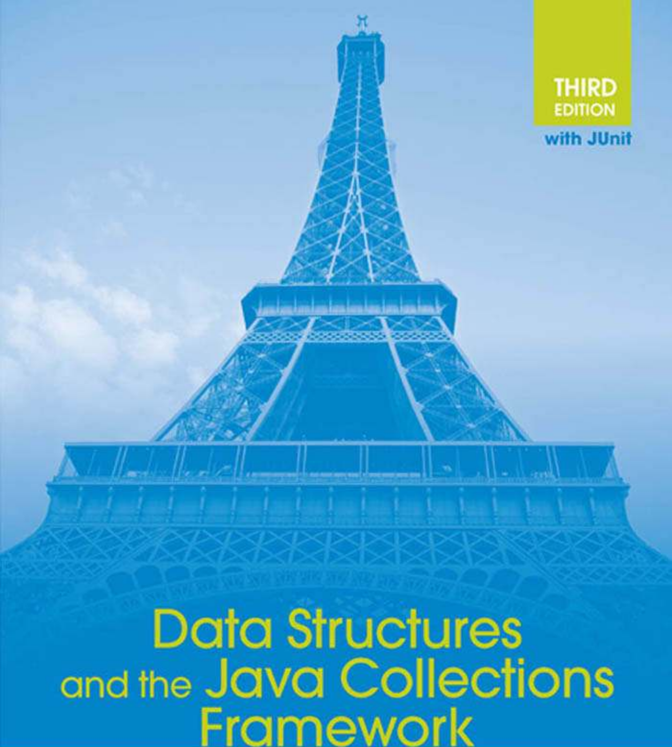


THIRD EDITION

with JUnit

Data Structures and the Java Collections Framework

WILLIAM J. COLLINS

This page intentionally left blank

Data Structures and the Java Collections Framework

Data Structures and the Java Collections Framework

Third Edition

William J. Collins
Lafayette College

WILEY
JOHN WILEY & SONS, INC.

Vice President and Executive Publisher:	Donald Fowley
Executive Editor:	Beth Lang Golub
Editorial Assistants:	Michael Berlin, Elizabeth Mills
Marketing Manager:	Chris Ruel
Senior Production Manager:	Janis Soo
Assistant Production Editor:	Annabelle Ang-Bok
Designers:	Seng Ping Ngieng
Cover Photo:	© Jose Ignacio Soto/iStockphoto

This book was set in 10/12 Times Roman by Laserwords and printed and bound by Malloy Lithographers. The cover was printed by Malloy Lithographers.

This book is printed on acid free paper.

Library of Congress Cataloging-in-Publication Data

Collins, William J. (William Joseph)
Data structures and the Java collections framework / William Collins. – 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-48267-4 (paperback)
1. Java (Computer program language) 2. Data structures (Computer science) I. Title.
QA76.73.J38C657 2011
005.7′3–dc22

2010043311

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

To Jayden—my fountain of youth

—W.J.C.

This page intentionally left blank

BRIEF CONTENTS

Preface xvii

CHAPTER 0
Introduction to Java 1

CHAPTER 1
Object-Oriented Concepts 27

CHAPTER 2
Additional Features of Programming and Java 59

CHAPTER 3
Analysis of Algorithms 105

CHAPTER 4
The Java Collections Framework 133

CHAPTER 5
Recursion 155

CHAPTER 6
Array-Based Lists 233

CHAPTER 7
Linked Lists 267

CHAPTER 8
Stacks and Queues 329

CHAPTER 9
Binary Trees 377

CHAPTER 10
Binary Search Trees 401

CHAPTER 11
Sorting 457

CHAPTER 12
Tree Maps and Tree Sets 501

CHAPTER 13
Priority Queues 551

CHAPTER 14
Hashing 599

CHAPTER 15
Graphs, Trees, and Networks 643

APPENDIX 1
Additional Features of the JAVA Collections Framework 701

APPENDIX 2
Mathematical Background 705

APPENDIX 3
Choosing a Data Structure 721

References 725

Index 727

CONTENTS

Preface xvii

CHAPTER 0
Introduction to Java 1

Chapter Objectives 1
0.1 Java Fundamentals 1
- *0.1.1 Primitive Types 2*
- *0.1.2 The `char` Type 2*

0.2 Classes 3
- *0.2.1 The `String` Class 4*
- *0.2.2 Using `javadoc` Notation for Method Specifications 5*
- *0.2.3 Equality of References and Equality of Objects 7*
- *0.2.4 Local Variables 9*
- *0.2.5 The `Scanner` Class 12*

0.3 Arrays 17
0.4 Arguments and Parameters 19
0.5 Output Formatting 22
Crossword Puzzle 24
Programming Exercises 25

CHAPTER 1
Object-Oriented Concepts 27

Chapter Objectives 27
1.1 Data Abstraction 27
1.2 Abstract Methods and Interfaces 28
- *1.2.1 Abstract Data Types and Data Structures 31*
- *1.2.2 An Interface and a Class that Implements the Interface 31*
- *1.2.3 Using the `FullTimeEmployee` Class 35*

1.3 Inheritance 37
- *1.3.1 The **`protected`** Visibility Modifier 39*
- *1.3.2 Inheritance and Constructors 43*
- *1.3.3 The Subclass Substitution Rule 43*
- *1.3.4 Is-a versus Has-a 47*

1.4 Information Hiding 48
1.5 Polymorphism 48
1.6 The Unified Modeling Language 49
Summary 52
Crossword Puzzle 54
Concept Exercises 55
Programming Exercises 56
Programming Project 1.1: A `CalendarDate` Class 58

CHAPTER 2
Additional Features of Programming and Java 59

Chapter Objectives 59
2.1 Static Variables, Constants and Methods 59
2.2 Method Testing 61
- *2.2.1 More Details on Unit Testing 64*

2.3 Exception Handling 68
- *2.3.1 Propagating Exceptions 71*
- *2.3.2 Unit Testing and Propagated Exceptions 74*
- *2.3.3 Checked Exceptions 77*
- *2.3.4 The `finally` Block 81*

2.4 File Output 81
2.5 System Testing 91
2.6 The Java Virtual Machine 92
- *2.6.1 Pre-Initialization of Fields 92*
- *2.6.2 Garbage Collection 92*

2.7 Packages 93

2.8 Overriding the `Object` Class's `equals` Method 94
Summary 97
Crossword Puzzle 98
Concept Exercises 99
Programming Exercises 100
Programming Project 2.1: An Integrated Web Browser and Search Engine, Part 1 102

CHAPTER 3
Analysis of Algorithms 105

Chapter Objectives 105
3.1 Estimating the Efficiency of Methods 105
3.1.1 Big-O Notation 106
3.1.2 Getting Big-O Estimates Quickly 110
3.1.3 Big-Omega, Big-Theta and Plain English 116
3.1.4 Growth Rates 117
3.1.5 Trade-Offs 119
3.2 Run-Time Analysis 121
3.2.1 Timing 121
3.2.2 Overview of the `Random` Class 122
Summary 126
Crossword Puzzle 127
Concept Exercises 128
Programming Exercises 130
Programming Project 3.1: Let's Make a Deal! 131

CHAPTER 4
The Java Collections Framework 133

Chapter Objectives 133
4.1 Collections 133
4.1.1 Collection Classes 134
4.1.2 Storage Structures for Collection Classes 136
4.2 Some Details of the Java Collections Framework 136
4.2.1 Abstract Classes 137
4.2.2 Parameterized Types 140
4.2.3 The `Collection` Interface 141
4.2.4 The `List` Interface 147
Summary 150
Crossword Puzzle 151
Concept Exercises 152
Programming Exercises 152
Programming Project 4.1: Wear a Developer's Hat and a User's Hat 153

CHAPTER 5
Recursion 155

Chapter Objectives 155
5.1 Introduction 155
5.2 Factorials 156
5.2.1 Execution Frames 159
5.3 Decimal to Binary 162
5.4 Towers of Hanoi 167
5.4.1 Analysis of the `move` Method 177
5.5 Searching an Array 179
5.6 Backtracking 191
5.6.1 An A-maze-ing Application 195
5.7 Indirect Recursion 208
5.8 The Cost of Recursion 209
Summary 210
Crossword Puzzle 211
Concept Exercises 212
Programming Exercises 214
Programming Project 5.1: Iterative Version of the Towers of Hanoi 219
Programming Project 5.2: Eight Queens 221
Programming Project 5.3: A Knight's Tour 222
Programming Project 5.4: Sudoku 225
Programming Project 5.5: Numbrix 227

CHAPTER 6
Array-Based Lists 233

Chapter Objectives 233
6.1 The List Interface 234
6.2 The ArrayList Class 234
6.2.1 Method Specifications for the ArrayList Class 235
6.2.2 A Simple Program with an ArrayList Object 244
6.2.3 The ArrayList Class's Heading and Fields 246
6.2.4 Definition of the One-Parameter add Method 248
6.3 Application: High-Precision Arithmetic 251
6.3.1 Method Specifications and Testing of the VeryLongInt Class 252
6.3.2 Fields in the VeryLongInt Class 253
6.3.3 Method Definitions of the VeryLongInt Class 254
Summary 257
Crossword Puzzle 258
Concept Exercises 259
Programming Exercises 259
Programming Project 6.1: Expanding the VeryLongInt Class 263
Programming Project 6.2: An Integrated Web Browser and Search Engine, Part 2 264

CHAPTER 7
Linked Lists 267

Chapter Objectives 267
7.1 What is a Linked List? 267
7.2 The SinglyLinkedList Class—A Singly-Linked, Toy Class! 268
7.2.1 Fields and Method Definitions in the SinglyLinkedList Class 273
7.2.2 Iterating through a SinglyLinkedList Object 276
7.3 Doubly-Linked Lists 281
7.3.1 A User's View of the LinkedList Class 282
7.3.2 The LinkedList Class versus the ArrayList Class 282
7.3.3 LinkedList Iterators 285
7.3.4 A Simple Program that uses a LinkedList Object 291
7.3.5 Fields and Heading of the LinkedList Class 294
7.3.6 Creating and Maintaining a LinkedList Object 296
7.3.7 Definition of the Two-Parameter add Method 298
7.4 Application: A Line Editor 300
7.4.1 Design and Testing of the Editor Class 304
7.4.2 Method Definitions for the Editor Class 308
7.4.3 Analysis of the Editor Class Methods 312
7.4.4 Design of the EditorUser Class 312
7.4.5 Implementation of the EditorUser Class 313
Summary 315
Crossword Puzzle 316
Concept Exercises 317
Programming Exercises 318
Programming Project 7.1: Expanding the SinglyLinkedList Class 320
Programming Project 7.2: Implementing the remove() Method in SinglyLinkedListIterator 322
Programming Project 7.3: Making a Circular Singly Linked List Class 322
Programming Project 7.4: Alternative Implementation of the LinkedList Class 323
Programming Project 7.5: Expanding the Line Editor 323
Programming Project 7.6: An Integrated Web Browser and Search Engine, Part 3 328

CHAPTER 8
Stacks and Queues 329

Chapter Objectives 329
8.1 Stacks 329
8.1.1 The `Stack` Class 329
8.1.2 A Fatal Flaw? 333
8.1.3 Stack Application 1: How Compilers Implement Recursion 334
8.1.4 Stack Application 2: Converting from Infix to Postfix 338
8.1.5 Prefix Notation 343
8.2 Queues 347
8.2.1 The `Queue` Interface 348
8.2.2 Implementations of the `Queue` Interface 349
8.2.3 Computer Simulation 350
8.2.4 Queue Application: A Simulated Car Wash 351
Summary 365
Crossword Puzzle 366
Concept Exercises 367
Programming Exercises 368
Programming Project 8.1: Making the Speedo's Car Wash Simulation More Realistic 369
Programming Project 8.2: Design, Test, and Implement a Program to Evaluate a Condition 371
Programming Project 8.3: Maze-Searching, Revisited 374
Programming Project 8.4: Fixing the `Stack` Class 375

CHAPTER 9
Binary Trees 377

Chapter Objectives 377
9.1 Definition of Binary Tree 377
9.2 Properties of Binary Trees 378
9.3 The Binary Tree Theorem 383
9.4 External Path Length 385
9.5 Traversals of a Binary Tree 386
Summary 393
Crossword Puzzle 394
Concept Exercises 395

CHAPTER 10
Binary Search Trees 401

Chapter Objectives 401
10.1 Binary Search Trees 402
10.1.1 The `BinarySearchTree` Implementation of the `Set` Interface 403
10.1.2 Implementation of the `BinarySearchTree` Class 411
10.2 Balanced Binary Search Trees 430
10.2.1 AVL Trees 435
10.2.2 The Height of an AVL Tree 436
10.2.3 The `AVLTree` Class 438
10.2.4 Runtime Estimates 441
Summary 442
Crossword Puzzle 443
Concept Exercises 444
Programming Exercises 448
Programming Project 10.1: An Alternate Implementation of the Binary-Search-Tree Data Type 449
Programming Project 10.2: Printing a `BinarySearchTree` Object 451
Programming Project 10.3: The `fixAfterInsertion` Method 451
Programming Project 10.4: The `fixAfterDeletion` Method 455

CHAPTER 11
Sorting 457

Chapter Objectives 457
11.1 Introduction 457

11.2 Simple Sorts 458
11.2.1 Insertion Sort 459
11.2.2 Selection Sort 461
11.2.3 Bubble Sort 463
11.3 The `Comparator` Interface 465
11.4 How Fast Can we Sort? 468
11.4.1 Merge Sort 470
11.4.2 The Divide-and-Conquer Design Pattern 476
11.4.3 Quick Sort 477
11.5 Radix Sort 489
Summary 493
Crossword Puzzle 494
Concept Exercises 495
Programming Exercises 497
Programming Project 11.1: Sorting a File into Ascending Order 497

CHAPTER 12 Tree `Maps` and Tree Sets 501

Chapter Objectives 501
12.1 Red-Black Trees 501
12.1.1 The Height of a Red Black Tree 503
12.2 The `Map` Interface 504
12.3 The `TreeMap` Implementation of the `SortedMap` Interface 509
12.3.1 The `TreeMap` Class's Fields and Embedded `Entry` Class 512
12.3.2 Method Definitions in the `TreeMap` Class 513
12.4 Application of the `TreeMap` Class: a Simple Thesaurus 517
12.4.1 Design, Testing, and Implementation of the `Thesaurus` Class 518
12.4.2 Design and Testing of the `ThesaurusUser` Class 521
12.4.3 Implementation of the `ThesaurusUser` Class 523
12.5 The `TreeSet` Class 525
12.5.1 Implementation of the `TreeSet` Class 528
12.5.2 Application: A Simple Spell Checker 530
Summary 536
Crossword Puzzle 537
Concept Exercises 538
Programming Exercises 539
Programming Project 12.1: Spell Check, Revisited 540
Programming Project 12.2: Word Frequencies 542
Programming Project 12.3: Building a Concordance 543
Programming Project 12.4: Approval Voting 545
Programming Project 12.5: An Integrated Web Browser and Search Engine, Part 4 548

CHAPTER 13 Priority Queues 551

Chapter Objectives 551
13.1 Introduction 551
13.2 The `PriorityQueue` Class 552
13.3 Implementation Details of the `PriorityQueue` Class 553
13.3.1 Fields and Method Definitions in the `PriorityQueue` Class 557
13.4 The `heapSort` Method 567
13.4.1 Analysis of `heapSort` 572
13.5 Application: Huffman Codes 573
13.5.1 Huffman Trees 575
13.5.2 Greedy Algorithm Design Pattern 578
13.5.3 The Huffman Encoding Project 578
Summary 590
Crossword Puzzle 591
Concept Exercises 592

Programming Exercises 594
Programming Project 13.1: Decoding a Huffman-Encoded Message 595
Programming Project 13.2: An Integrated Web Browser and Search Engine, Part 5 597

CHAPTER 14 Hashing 599

Chapter Objectives 599
14.1 A Framework to Analyze Searching 599
14.2 Review of Searching 600
14.2.1 Sequential Search 600
14.2.2 Binary Search 601
14.2.3 Red-Black-Tree Search 602
14.3 The `HashMap` Implementation of the `Map` Interface 603
14.3.1 Hashing 604
14.3.2 The Uniform Hashing Assumption 609
14.3.3 Chaining 609
14.3.4 Implementation of the `HashMap` Class 612
14.3.5 Analysis of the `containsKey` Method 614
14.3.6 The `HashIterator` Class 615
14.3.7 Creating a Symbol Table by Hashing 617
14.4 The `HashSet` Class 625
14.5 Open-Address Hashing (optional) 626
14.5.1 The `remove` Method 627
14.5.2 Primary Clustering 631
14.5.3 Double Hashing 632
Summary 635
Crossword Puzzle 636
Concept Exercises 637
Programming Exercises 639
Programming Project 14.1: The Double Hashing Implementation of the `HashMap` Class 640
Programming Project 14.2: An Integrated Web Browser and Search Engine, Part 6 640

CHAPTER 15 Graphs, Trees, and Networks 643

Chapter Objectives 643
15.1 Undirected Graphs 643
15.2 Directed Graphs 647
15.3 Trees 648
15.4 Networks 649
15.5 Graph Algorithms 650
15.5.1 Iterators 650
15.5.2 Connectedness 658
15.5.3 Generating a Minimum Spanning Tree 659
15.5.4 Finding the Shortest Path through a Network 663
15.5.5 Finding the Longest Path through a Network? 667
15.6 A `Network` Class 669
15.6.1 Method Specifications and Testing of the `Network` Class 671
15.6.2 Fields in the Network Class 680
15.6.3 Method Definitions in the `Network` Class 681
15.7 Backtracking Through A Network 686
Summary 689
Crossword Puzzle 690
Concept Exercises 691
Programming Exercises 693
Programming Project 15.1: The Traveling Salesperson Problem 694
Programming Project 15.2: Backtracking through a Network 695
Programming Project 15.3: Determining Critical Activities in a Project Network 697
Programming Project 15.4: An Integrated Web Browser and Search Engine, Part 7 698

APPENDIX 1
Additional Features of the JAVA Collections Framework 701

A1.1 Introduction 701
A1.2 Serialization 701
A1.3 Fail-Fast Iterators 703

APPENDIX 2
Mathematical Background 705

A2.1 Introduction 705
A2.2 Functions and Sequences 705
A2.3 Sums and Products 706
A2.4 Logarithms 707
A2.5 Mathematical Induction 708
A2.6 Induction and Recursion 719
Concept Exercises 719

APPENDIX 3
Choosing a Data Structure 721

A3.1 Introduction 721
A3.2 Time-Based Ordering 721
A3.3 Index-Based Ordering 721
A3.4 Comparison-Based Ordering 722
A3.5 Hash-Based Ordering 723
A3.6 Space Considerations 723
A3.7 The Best Data Structure? 724

References 725

Index 727

This page intentionally left blank

PREFACE

This book is intended for an object-oriented course in data structures and algorithms. The implementation language is Java, and it is assumed that students have taken a first course in programming, not necessarily in Java. That course should have covered the fundamental statements and data types, as well as arrays. Chapter 0 supplies the material on Java that is fundamental to the remaining chapters, so it serves as a review for those with previous experience in Java, and brings Java novices up to speed.

WHAT'S NEW IN THE THIRD EDITION

This edition presents two major changes from the second edition. First, the `Scanner` class has replaced the `BufferedReader` and `StringTokenizer` classes. The `Scanner` class's versatility supports pattern matching as well as keyboard input and file input. Second, there is an increased emphasis on testing. In particular, the unit-testing features of JUnit 4 are introduced in Chapter 2 and integrated in applications throughout the remaining chapters.

THE JAVA COLLECTIONS FRAMEWORK

One of the distinctive features of this text is its emphasis on the Java Collections Framework, part of the `java.util` package. Basically, the framework is a hierarchy with interfaces at each level except the lowest, and collection classes that implement those interfaces at the lowest level. The collection classes implement most of the data structures studied in a second computer-science course, such as a resizable array class, a linked-list class, a stack class, a queue class, a balanced binary-search-tree class, a priority-queue class, and a hash-map class.

There are several advantages to using the Java Collections Framework. First, students will be working with code that has been extensively tested; they need not depend on modules created by the instructor or textbook author. Second, the framework is available for later courses in the curriculum and beyond. Third, although the primary emphasis is on *using* the Java Collections Framework, the framework classes are not treated simply as "black boxes." For each such class, the heading and fields are provided, and one method definition is dissected. This exposition takes away some of the mystery that would otherwise surround the class, and allows students to see the efficiency and succinctness of professionals' code.

The version of the Java Collections Framework we will be working with includes type parameters. Type parameters, sometimes called "generic types," "generics," or "templates," were added to the Java language starting with Java 5.0. With type parameters, there is no need to downcast the return value from a collection, and many errors can be detected at compile-time that previously were discoverable only at run-time.

To complement generics, three other features have been added: boxing, unboxing, and an enhanced **`for`** statement. The elements in generic collections must be objects, often from a wrapper class such as `Integer`. If a primitive value appears where a collection method requires a wrapper element as an argument, ***boxing*** automatically converts the primitive value to the corresponding wrapper element. Conversely, if a wrapper-class element appears where a primitive value is needed, ***unboxing*** automatically converts that element to the corresponding primitive value. Finally, the enhanced `for` statement—often called a "for-each" statement—has a sleek structure for iterating through a collection. The net effect of these new features of Java is to improve productivity by relegating to the compiler many of the "boiler-plate" details related to casting and iterating.

OTHER IMPLEMENTATIONS CONSIDERED

As important as the Java Collections Framework implementations are, they cannot be the exclusive focus of such a fundamental course in data structures and algorithms. Approaches that differ from those in the framework deserve consideration. For example, the `HashMap` class utilizes chaining, so there is a separate section on open addressing, and a discussion of the trade-offs of one design over the other. Also, there is coverage of data structures (such as a weighted digraph class) and algorithms (such as Heap Sort) that are not yet included in the Java Collections Framework.

Sometimes, the complexity of the framework classes is mitigated by first introducing simpler versions of those classes. For example, the `SinglyLinkedList` class—not in the Java Collections Framework—helps to pave the way for the more powerful `LinkedList` class, which is in the framework. And the `BinarySearchTree` class prepares students to understand the framework's `TreeMap` class, based on red-black trees.

This text satisfies another important goal of a data structures and algorithms course: Students have the opportunity to develop their own data structures. There are programming projects in which data structures are either created "from the ground up" or extended from examples in the chapters. And there are many other projects to develop or extend applications that *use* the Java Collections Framework.

JUNIT AND TEST-FIRST DEVELOPMENT

Students realize that methods with no compile-time errors may still be a long way from correct, but they often need help in learning how to systematically test their methods. As described in Chapter 2, JUnit is an Open Source platform for the testing of units, that is, methods. For example, to test a `findMedian` method, a `FindMedianTest` class is developed. The `FindMedianTest` class consists mainly of methods that test `findMedian`. When all the test methods in `FindMedianTest` have been passed, the student's confidence in the correctness of `findMedian` is increased.

Instead of writing the tests after a method has been defined, we employ a "test-first" strategy. As soon as a method's specifications have been written, the tests for that method are coded. This ensures that the tests are based on the specifications only, not on the definition of the method. These tests are run on a "stub" version of the method to be tested, and all of the tests will fail. Then the method definition is written, and the tests are run on that version of the method. Corrections to the method are made as necessary until, eventually, all tests succeed. The test-first paradigm is introduced in Chapter 2 and utilized in subsequent chapters.

PEDAGOGICAL FEATURES

This text offers several features that may improve the teaching environment for instructors and the learning environment for students. Each chapter starts with a list of objectives, and most chapters conclude with several major programming assignments. Each chapter also has a crossword puzzle, from Crossword

Weaver—to help students learn the key words and phrases in an enjoyable setting—and a variety of exercises. The answers to all of the exercises are available to the instructor.

Each data structure is carefully described, with the specifications for each method given in javadoc notation. Also, there are examples of how to call the method, and the results of that call. To reinforce the important aspects of the material and to hone students' coding skills in preparation for programming projects, there is a suite of 23 lab experiments. The organization of these labs is described later in this preface.

SUPPORT MATERIAL

The website for all of the support material is www.wiley.com/college/collins/

That website has links to the following information for students:

- The suite of 23 labs. Lab 0 starts with a brief overview of the lab format.
- The source codes for all classes developed in the text.
- Applets for projects that have a strong visual component.

Additionally, instructors can obtain the following from the website:

- PowerPoint slides for each chapter (approximately 1500 slides).
- Answers to every exercise and lab experiment.

SYNOPSES OF THE CHAPTERS

Chapter 0 serves as an introduction to the Java language for those whose first course was in some other language. For students who are already familiar with Java, the chapter consists mostly of review material, but the treatment of the `Scanner` class is worth perusing.

Chapter 1 focuses on the fundamentals of object-oriented programming: encapsulation, inheritance and polymorphism. For a concrete illustration of these topics, an interface is created and implemented, and the implementation is extended. The relationship between abstract data types and interfaces is explored, as is the corresponding connection between data structures and classes. The Universal Modeling Language provides a design tool to depict the interrelationships among interfaces, classes and subclasses.

Chapter 2 introduces unit testing with the free package JUnit. This is a vital topic in programming, so method testing—before the method is defined—is emphasized in virtually all subsequent applications, programming assignments and lab experiments. The chapter also includes some additional features of the Java language. For example, there are sections on exception handling, file output, and the Java Virtual Machine. Also, there is a section on the `Object` class's `equals` method, why that method should be overridden, and how to accomplish the overriding.

Finally, Chapter 2 introduces a "theme" project: to develop an integrated web browser and search engine. This project, based on a paper by Newhall and Meeden [2002], continues through six of the remaining chapters, and clearly illustrates the practical value of understanding data structures. In the first part of the project, students develop a graphical user interface—a version of this interface is available to instructors who prefer to provide this part of the project to students. The remaining six parts involve an `ArrayList` object, a `LinkedList` object, a `TreeMap` object, a `PriorityQueue` object, a `HashMap` object, and a `Digraph` object.

Chapter 3, *Analysis of Algorithms*, starts by defining functions to estimate a method's execution-time requirements, both in the average and worst cases. Big-O notation provides a convenient tool for estimating these estimates. Because Big-O notation yields environment-independent estimates, these results are then compared with actual run-times, which are determined with the help of the `Random` class and the `nanoTime` method.

Chapter 4 outlines the Java Collections Framework. We start with some preliminary material on collection classes in general, type parameters and the iterator design-pattern. The remainder of the chapter presents part of the major interface hierarchy (`Collection` and `List`) and its implementations (`ArrayList` and `LinkedList`).

Chapter 5, on recursion, represents a temporary shift in emphasis from data structures to algorithms. There is a gradual progression from simple examples (factorial and decimal-to-binary) to more powerful examples (binary search and backtracking). The mechanism for illustrating the execution of recursive methods is the execution frame. Backtracking is introduced, not only as a design pattern, but as another illustration of creating polymorphic references through interfaces. And the same `BackTrack` class is used for maze-searching and solving eight queens, knight's tour, Sudoku, and Numbrix.

In Chapter 6, we study the Java Collections Framework's `ArrayList` class. An `ArrayList` object is a smart array: automatically resizable, and with methods to handle insertions and deletions at any index. The design starts with the method specifications for some of the most widely-used methods in the `ArrayList` class. There follows a brief outline of the implementation of the class. The application of the `ArrayList` class, high-precision arithmetic, is essential for public-key cryptography. This application is extended in a lab and in a programming project. Several JUnit 4 tests are included in the chapter, and the remaining tests are available from the book's website.

Chapter 7 presents linked lists. A discussion of singly-linked lists leads to the development of a primitive `SinglyLinkedList` class. This serves mainly to prepare students for the framework's `LinkedList` class. `LinkedList` objects are characterized by linear-time methods for inserting, removing or retrieving at an arbitrary position. This property makes a compelling case for *list iterator*s: objects that traverse a `LinkedList` object and have constant-time methods for inserting, removing or retrieving at the "current" position. The framework's design is doubly-linked and circular, but other approaches are also considered. The application is a small line-editor, for which list iterators are well suited. Testing entails an interesting feature: the testing of protected methods. The line-editor application is extended in a programming project.

Stacks and queues are the subjects of Chapter 8. The framework's `Stack` class has the expected `push`, `pop,` and `peek` methods. But the `Stack` class also allows elements to be inserted or removed anywhere in a `Stack` object, and this permission violates the definition. Students can use the `Stack` class—with care—or develop their own version that satisfies the definition of a stack. There are two applications of stacks: the implementation of recursion by a compiler, and the conversion from infix to postfix. This latter application is expanded in a lab, and forms the basis for a project on evaluating a condition.

The Java Collections Framework has a `Queue` interface, but that interface supports the removal of any element from a queue! As with the `Stack` class, students can tolerate this flaw and use a class—such as `LinkedList` —that implements the `Queue` interface. Or they can create their own implementation that does not violate the definition of a queue. The specific application of queues, calculating the average waiting time at a car wash, falls into the general category of *computer simulation*.

Chapter 9 focuses on binary trees in general, as a prelude to the material in Chapters 10 through 13. The essential features of binary trees are presented, including both botanical (root, branch, leaf) and familial (parent, child, sibling) terms. Binary trees are important for understanding later material on AVL trees, decision trees, red-black trees, heaps, and Huffman trees.

In Chapter 10, we look at binary search trees, including a `BinarySearchTree` class, and explain the value of balanced binary search trees. Rotations are introduced as the mechanism by which re-balancing is accomplished, and AVL trees are offered as examples of balanced binary search trees. An `AVLTree` class, as a subclass of `BinarySearchTree`, is outlined; the crucial methods, `fixAfterInsertion` and `fixAfterDeletion`, are left as programming projects.

Sorting is the theme of Chapter 11. Estimates of the lower bounds for comparison-based sorts are determined. A few simple sorts are defined, and then we move on to two sort methods provided by the framework. Quick Sort sorts an array of a primitive type, and Merge Sort works for an array of objects and for implementations of the `List` interface. A lab experiment compares all of these sort algorithms on randomly-generated integers.

The central topic of Chapter 12 is how to use the `TreeMap` class. A ***map*** is a collection in which each element has a unique key part and also a value part. In the `TreeMap` implementation of the `Map` interface, the elements are stored in a red-black tree, ordered by the elements' keys. There are labs to guide students through the details of re-structuring after an insertion or removal. The application consists of searching a thesaurus for synonyms, and JUnit 4 testing is again illustrated. The `TreeSet` class has a `TreeMap` field in which each element has the same, dummy value-part. The application of the `TreeSet` class is a simple spell-checker, which is also thoroughly tested.

Chapter 13 introduces the `PriorityQueue` class. This class is part of the Java Collections Framework and, like the Stack class and Queue interface in Chapter 8, allows methods that violate the definition of a priority queue. The class utilizes a heap to provide insertions in constant average time, and removal of the smallest-valued element in logarithmic worst time. The application is in the area of data compression: Given a text file, generate a minimal, prefix-free encoding. There is a project assignment to convert the encoded message back to the original text file.

Chapter 14 investigates hashing. The Java Collections Framework has a `HashMap` class for elements that consist of unique-key/value pairs. Basically, the average time for insertion, removal, and searching is constant! This average speed is exploited in an application (and JUnit 4 tests) to create a simple symbol table. The Java Collections Framework's implementation of hashing, using chained hashing, is compared to open-address hashing.

The most general data structures—graphs, trees, and networks—are presented in Chapter 15. There are outlines of the essential algorithms: breadth-first traversal, depth-first traversal, finding a minimum spanning tree, and finding the shortest or longest path between two vertices. The only class developed is the (directed) `Network` class, with an adjacency-map implementation. Other classes, such as `UndirectedGraph` and `UndirectedNetwork`, can be straightforwardly defined as subclasses of `Network`. The Traveling Salesperson Problem is investigated in a lab, and there is a programming project to solve that problem—not necessarily in polynomial time! Another backtracking application is presented, with the same `BackTrack` class that was introduced in Chapter 5.

The website includes all programs developed in each chapter, all JUnit 4 tests, and applets, where appropriate, to animate the concepts presented.

APPENDIXES

Appendix 1 has two additional features of the Java Collections Framework. Each of the collection classes in the framework is ***serializable***, that is, an instance of the class can be conveniently stored to an output stream, and the instance can later be re-constituted from an input stream (de-serialization). Framework iterators are ***fail-fast***: During an iteration through a collection, there should be no insertions into or removals from the collection except by the iterator. Otherwise, the integrity of that iterator may be compromised, so an exception will be thrown as soon as the iterator's unreliability has been established.

Appendix 2 contains the background that will allow students to comprehend the mathematical aspects of the chapters. Summation notation and the rudimentary properties of logarithms are essential, and the material on mathematical induction will lead to a deeper appreciation of recursion as well as the analysis of binary trees.

Appendix 3, "Choosing a Data Structure," can be viewed as a summary of the eight major data structures in the book. These collection classes are categorized according to their ordering of elements (for example, time-based for stacks and queues) and according to their performance of standard operations (for example, the `TreeMap` class requires logarithmic-in-n time to search for a key). Table A3.1 summarizes the summary.

ORGANIZATION OF THE LABS

There are 23 web labs associated with this text. For both students and instructors, the initial Uniform Resource Locator (URL) is www.wiley.com/college/collins.

The labs do not contain essential material, but provide reinforcement of the text material. For example, after the `ArrayList` and `LinkedList` classes have been investigated, there is a lab to perform some timing experiments on those two classes.

The labs are self-contained, so the instructor has considerable flexibility in assigning the labs:

a. they can be assigned as closed labs;

b. they can be assigned as open labs;

c. they can be assigned as ungraded homework.

In addition to the obvious benefit of promoting active learning, these labs also encourage use of the scientific method. Basically, each lab is set up as an experiment. Students *observe* some phenomenon, such as creating a greedy cycle to solve the Traveling Salesperson Problem. They then formulate and submit a *hypothesis*—with their own code—about the phenomenon. After *testing* and, perhaps, revising their hypothesis, they submit the *conclusions* they drew from the experiment.

ACKNOWLEDGEMENTS

Joshua Bloch, lead designer of the Java Collections Framework, gave valuable insights on type parameters and their impact on the Java Collections Framework.

Chun Wai Liew of Lafayette College helped to incorporate JUnit into this edition of the text.

I am indebted to the suggestions and corrections of the reviewers: Dean Hendrix (Auburn University), Benjamin Kuperman (Oberlin College), Andrew Haas (State University of New York—Albany), Kathy Liszka (University of Akron), Paul Bladek (Edmonds Community College), Siva Jasthi (Metropolitan State University), Hashem Anwari (Northern Virginia Community College), Alan Peslak (Nova Southeastern University), H. K. Dai (Oklahoma State University), Jiang Li (Austin Peay State University), Paul J. Wagner (University of Wisconsin—Eau Claire), John Mallozzi (Iona College), Robert Goldberg (Queens College), and Michael Clancy (University of California—Berkeley).

Introduction to Java

CHAPTER 0

This is a book about programming: specifically, about understanding and using data structures and algorithms. The Java Collections Framework has a considerable number of data structures and algorithms. Subsequent chapters will focus on what the framework is and how to use the framework in your programs. For this information to make sense to you, you will need to be familiar with certain aspects of Java that we present in this chapter. All of the material is needed, either for the framework itself or to enable you to use the framework in your programming projects.

CHAPTER OBJECTIVES

1. Learn (or review) the fundamentals of Java, including classes, objects and messages.
2. Be able to use javadoc in writing method specifications.
3. Incorporate the `Scanner` class into your programming.
4. Understand the significance of the fact that a copy of the argument is stored in the corresponding parameter when a method is called.
5. Understand the details of arrays and output formatting.

0.1 Java Fundamentals

Every Java program is a collection of classes. Basically, a ***class*** consists of variables, called ***fields***, together with functions, called ***methods***, that operate on those fields. A program is executed when a special method, the `main` method, is called by the run-time system (also known as the Java Virtual Machine). The heading of this method is fixed, as shown in the following program:

```
public class HelloWorld
{
      public static void main (String [ ] args)
      {
             System.out.println ("Hello, world!");
      }  // method main
} // class HelloWorld
```

The `main` method in this program calls another method, `println`, to produce the following output to the console window:

```
Hello, world!
```

Console output, that is, output to the console window on your monitor, is handled by the methods `System.out.print`, `System.out.println`, and `System.out.printf` (see Section 0.5).

0.1.1 Primitive Types

A ***primitive type*** is a collection of values, together with operations that can be performed on those values. For example, the reserved word **`int`** denotes the primitive type whose values are integers in the range from about −2 billion to 2 billion, and whose operations are addition, subtraction, and so on. A ***variable***—also called an ***instance***—of type **`int`** is a location in computer memory that can hold one value of type **`int`**. The term "variable" is used because the value stored can change during the execution of a program. Instead of specifying the location's address, we provide an identifier (that is, a name) and the Java compiler associates the identifier with a location. For example, here is a declaration for an **`int`** variable whose identifier is `score`:

```
int score;
```

By a standard abuse of language, we say that `score` is a variable instead of saying that `score` is an identifier for a variable. An assignment statement allows us to store a value in a variable. For example,

```
score = 0;
```

stores the value 0 in the variable `score`. A subsequent assignment can change the value stored:

```
score = 88;
```

The left-hand side of an assignment statement must be a variable, but the right-hand side can be an arbitrary ***expression***: any legal combination of symbols that has a value. For example, we can write

```
score = (score + 3) / 10;
```

If score had the value 88 prior to the execution of this assignment statement, then after its execution, score would have the value 9. Note that the division operator, /, returns the result of integer division because the two operands, 91 and 10, are both integers.

Another operator in the **`int`** type is %, the ***modulus operator***, which returns the integer remainder after integer division. For example, 91 % 10 returns the remainder, 1, when 91 is divided by 10. Similarly, 87 % 2 returns 1, (−37) % 5 returns −2, and 10 % 91 returns 10.

Java supports eight primitive types, summarized in Table 0.1.

0.1.2 The char Type

The **`char`** type stores characters in the Unicode collating sequence, which includes all of the ASCII characters such as `'a'`, `'A'`, `'?'`, and `' '`, the blank character. For example, suppose we write

```
char delimiter = ' ';
```

Then `delimiter` is a variable of type **`char`** and contains the value `' '`. The Unicode collating sequence also includes other—that is, non-Roman, alphabets—such as Greek, Cyrillic, Arabic, and Hebrew. The Unicode collating sequence holds up to 65,536 (= 2^{16}) distinct characters, but only about half of them have been assigned as of now. To include a character such as ☺ in a program, you provide an ***escape sequence***:

Table 0.1 The Primitive Types

Primitive Type	Range	Size
int	−2,147,483,648 to 2,147,483,647	4 bytes
long	-2^{63} to $2^{63} - 1$	8 bytes
short	−128 to 127	2 bytes
byte	−64 to 63	1 byte
double	$-1.7976931348623157 * 10^{308}$ to $1.7976931348623157 * 10^{308}$ (15 digits of precision)	8 bytes
float	$-3.4028235 * 10^{38}$ to $3.4028235 * 10^{38}$ (6 digits of precision)	4 bytes
char		2 bytes
boolean	**false, true**	1 byte

a sequence of symbols that starts with the backslash character, `'\'` and designates a single character. For example, the escape sequence for the smiley-face character, ☺, is `'\u263A'`, so we can write

```
char c = '\u263A';
```

to declare a **char** variable `c` and assign the smiley-face character to `c`.

More importantly, escape sequences are used for print control. For example, `'\n'` represents the new-line character and `'\t'` represents the tab character. The execution of

```
System.out.println ("We can control\n\noutput with \tprint-control\t\t characters");
```

will produce output of

```
We can control

output with    print-control          characters
```

An escape sequence is also needed to print a double quote—otherwise, the double quote would signify the end of the string to be output. For example, the execution of

```
System.out.println ("His motto was \"Don't sweat the nickels and dimes!\"");
```

will produce output of

```
His motto was "Don't sweat the nickels and dimes!"
```

0.2 Classes

In addition to primitive types such as **int** and **char**, Java provides programmers with the ability to create powerful new types called "classes." Given a problem, we develop classes—or utilize already existing classes—that correspond to components of the problem itself. A class combines the passive components (fields) and active components (methods) into a single entity. This grouping increases ***program modularity***:

the separation of a program into components that are coherent units. Specifically, a class is isolated from the rest of the program, and that makes the whole program easier to understand and to modify.

In Section 0.2.1, we investigate the class concept in more detail by looking at a specific example: the `String` class, the most widely used of Java's pre-declared classes.

0.2.1 The `String` Class

To start with a simple example, we consider the `String` class. Actually, the `String` class is somewhat intricate, with several fields and dozens of methods. But as we will focus on *using* the `String` class, we will ignore the fields, and look at only a few of the methods. In Chapter 1, we will introduce a new class and investigate its fields as well as its methods.

An ***object*** is an instance of a class; in other words, an object is a variable that contains fields and can call methods. In the context of using the `String` class, an object can be thought of as a variable that contains a ***string***—a sequence of characters—and can call the `String` class's methods. This gives rise to two questions:

1. How are `String` objects declared?
2. How do `String` objects call `String` methods?

The answer to the first question is somewhat surprising: `String` objects, in fact, objects of any class, cannot be declared in Java. Instead, we declare variables, called ***reference variables*****,** that can contain the address of an object. For example, we can declare

```
String s1;
```

Then `s1` is not a `String` object, but a variable that can contain the *address* of a `String` object.[1] In order for `s1` to actually contain such a reference, the space for a `String` object must be allocated, then the fields in that newly created `String` object must be initialized, and finally, the address of that `String` object must be assigned to `s1`. We combine these three steps into a single assignment statement. For example, if we want `s1` to be a reference to an empty `String` object, we write:

```
s1 = new String();
```

The right-hand side of this assignment statement accomplishes several tasks. The **`new`** operator allocates space in memory for a `String` object, calls a special method known as a "constructor" to initialize the fields in the object, and returns the address of that newly created object; that address is assigned to `s1`. A ***constructor*** is a method whose name is the same as the class's name and whose purpose is to initialize the object's fields. In this example, the fields are initialized to the effect that the newly created `String` object represents an empty string, that is, a string that contains no characters.

The constructor just mentioned has no parameters, and is called the ***default constructor***. The `String` class also has a constructor with a `String`-reference parameter. Here is the heading of that constructor:

```
public String (String original)
```

The parameter `original` is of type reference-to-`String`. When this constructor is called, the argument—inside the parentheses—will be assigned to the parameter, and then the body of the constructor (the statements inside the braces) will be executed. For an example of a call to this

[1] In the languages C and C++, a variable that can contain the address of another variable is called a ***pointer variable***.

constructor, the following statement combines the declaration of a reference variable and the assignment to that variable of a reference to a newly constructed `String` object:

```
String s2 = new String ("transparent");
```

When this statement is executed, the space for a new `String` object is allocated, the fields in that newly created `String` object are initialized to the effect that the new `String` object represents the string `"transparent"`, and the address of that new `String` object is assigned to the `String` reference `s2`.

Now that `s1` and `s2` contain live references, the objects referenced by `s1` and `s2` can invoke any `String` method.[2] For example, the `length` method takes no parameters and returns the number of characters in the calling object, that is, the object that invokes the `length` method. We can write

```
System.out.println (s1.length());
```

The output will be 0. If, instead, we write

```
System.out.println (s2.length());
```

then the output will be 11 because the calling object contains the string `"transparent"`.

The default constructor and the constructor with a `String`-reference parameter have the same name, `String`, but have different parameter lists. Java allows ***method overloading***: the ability of a class to have methods with the same method identifier but different parameter lists. In order to clarify exactly what method overloading entails, we define a method's ***signature*** to consist of the method identifier together with the number and types of parameters, in order. Method overloading is allowed for methods with different signatures. For example, consider the following method headings:

```
public String findLast (int n, String s)

public String findLast (String s, int n)
```

In this example, the first method's parameter list starts with an **`int`** parameter, but the second method's parameter list starts with a `String` parameter, so the two methods have different signatures. It is legal for these two methods to be defined in the same class; that is, method overloading is permitted. Contrast this example with the following:

```
public String findLast (int n, String s)

public int findLast (int j, String t)
```

Here the two methods have the same signature—notice that the return type is irrelevant in determining signature—so it would be *illegal* to define these two methods in the same class.

0.2.2 Using `javadoc` Notation for Method Specifications

The `String` class has a method that returns a copy of a specified substring—a contiguous part of—the calling string object. To make it easier for you to understand this method, we will supply the method's specification. A ***method specification*** is the explicit information a user will need in order to write code that invokes the method.

[2]Except `String` constructors, which are invoked by the **`new`** operator. For that reason, and the fact that constructors do not have a return type, the developers of the Java language do not classify a constructor as a method (see Arnold, 1996). But for the sake of simplicity, we lump constructors in with the methods of a class.

The method specification will include javadoc notation. ***javadoc*** is a program that converts Java source code and a specially formatted block of comments into Application Programming Interface (API) code in Hypertext Markup Language (HTML) for easy viewing on a browser. Because javadoc is available on any system that has Java, javadoc format has become the standard for writing method specifications. Each comment block starts with "/**", each subsequent line starts with "*", and the final line in a block has "*/". The complete specification consists of the javadoc comments and the method heading:

```
/**
 *  Returns a copy of the substring, between two specified indexes, of this String
 *  object.
 *
 *  @param beginIndex – the starting position (inclusive) of the substring.
 *  @param endIndex – the final position (exclusive) of the substring.
 *
 *  @return the substring of this String object from indexes beginIndex (inclusive)
 *   to endIndex (exclusive).
 *
 *  @throws IndexOutOfBoundsException – if beginIndex is negative, or if
 *          beginIndex is greater than endIndex, or if endIndex is greater than
 *          length().
 *
 */
public String substring (int beginIndex, int endIndex)
```

The first sentence in a javadoc comment block is called the ***postcondition***: the effect of a legal call to the method. The comment block also indicates parameters (starting with @param), the value returned (@return), and what exceptions can be thrown (@throws). An ***exception***, such as `IndexOutOfBounds Exception`, is an object created by an unusual condition, typically, an attempt at invalid processing. Section 2.2 covers the topic of exceptions, including how they are thrown and how they are caught. To avoid confusing you, we will omit `@throws` comments for the remainder of this chapter.

To illustrate the effect of calls to this method, here are several calls in which the calling object is either an empty string referenced by `s1` or the string "transparent" referenced by `s2`:

```
s1.substring (0,  0)   // returns reference to an empty string
s1.substring (0,  1)   // error: 2nd argument > length of calling object
s2.substring (1,  4)   // returns reference to copy of "ran", a 3-character string
s2.substring (5,  10)  // returns reference to copy of "paren", a 5-character string
s2.substring (5,  11)  // returns reference to copy of "parent", a 6-character string
```

There are several points worth mentioning about the comment block. In the postcondition and elsewhere, "this `String` object" means the calling object. What is returned is a reference to a copy of the substring. And the last character in the designated substring is at `endIndex -1`, not at `endIndex`.

The javadoc comment block just given is slightly simpler than the actual block for this `substring` method in the `String` class. The actual javadoc comment block includes several html tags: <pre>, <blockquote>, and <code>. And if you viewed the description of that method from a browser—that is, after the javadoc program had been executed for the `String` class—you would see the comments in an easier-to-read format. For example, instead of

```
 *  @return the substring of this String object from indexes beginIndex (inclusive)
 *          to endIndex (exclusive).
```

you would see

Returns:
the substring of this String object from indexes beginIndex (inclusive)
to endIndex (exclusive).

The on-line Java documentation is generated with javadoc. And the documentation about a method in one class may include a hyperlink to another class. For example, the heading for the `next()` method in the `Scanner` class is given as

```
public String next()
```

So if you are looking at the documentation of the `next()` method and you want to see some information on the `String` class, all you need to do is click on the `String` link.

Throughout the remainder of this text, we will regularly use javadoc to provide information about a method. You should try to use javadoc to describe your methods.

0.2.3 Equality of References and Equality of Objects

Reference variables represent an advance over the pointer mechanism of Java's predecessors, C and C++. A pointer variable could be assigned any memory address, and this often led to hard-to-find errors. In contrast, if a reference variable contains any address, it must be the address of an object created by the **`new`** operator. To indicate that a reference variable does not contain an address, we can assign to that variable a special value, indicated by the reserved word **`null`**:

```
String s3 = null;
```

At this point, `s3` does not contain an address, so it would be illegal to write

```
s3.length()
```

In object-oriented parlance, when a method is invoked, a ***message*** is being sent to the calling object. The term "message" is meant to suggest that a communication is being sent from one part of a program to another part. For example, the following message returns the length of the `String` object referenced by `s2`:

```
s2.length()
```

This message requests that the object referenced by `s2` return its length, and the value 11 is returned. The form of a message consists of a reference followed by a dot—called the ***member-selection operator***—followed by a method identifier followed by a parenthesized argument list.

Make sure you understand the difference between a **`null`** reference (such as `s3`), and a reference (such as `s1`) to an empty string. That distinction is essential to an understanding of Java's object-reference model.

The distinction between objects and references is prominent in comparing the `equals` method and the `==` operator. Here is the method specification for `equals`:

```
/**
 *  Compares this String object to a specified object:
```

```
 *  The result is true if and only if the argument is not null and is a String object
 *  that represents the same sequence of characters as this String object.
 *
 *  @param  anObject  - the object to compare this String against.
 *
 *  @return true - if the two String objects are equal; false otherwise.
 *
 */
public boolean equals (Object anObject)
```

The parameter's type suggests that the calling object can be compared to an object of any type, not just to a `String` object. Of course, **`false`** will be returned if the type is anything but `String`. The `Object` class is discussed in Chapter 1.

The **`==`** operator simply compares two *references*: **`true`** is returned if and only if the two references contain the same address. So if `str1` and `str2` are referencing identical `String` objects that are at different addresses,

```
str1.equals (str2)
```

will return **`true`** because the `String` objects are identical, but

```
str1 == str2
```

will return **`false`** because the `str1` and `str2` contain different addresses.

Finally, you can create a `String` object without invoking the **`new`** operator. For example,

```
String str0 = "yes",
       str3 = "yes";
```

Because the underlying strings are identical, only one `String` object is constructed, and both `str0` and `str3` are references to that object. In such cases, we say that the `String` object has been ***interned***.

Figure 0.1 has several examples, and contrasts the `String` method `equals` with the reference operator `==`.

The reason the output is different for the first and third calls to `println` in Figure 0.1 is that the `equals` method compares strings and the `==` operator compares references. Recall that each time the **`new`** operator is invoked, a new `String` object is created. So, as shown in Figure 0.2, `s4` is a reference to

```
String s4 = new String ("restful"),
       s5 = new String ("restful"),
       s6 = new String ("peaceful"),
       s7 = s4,
       s8 = "restful",
       s9 = "restful";

System.out.println (s4.equals (s5));       // the output is "true"
System.out.println (s4.equals (s6));       // the output is "false"
System.out.println (s4 == s5);             // the output is "false"
System.out.println (s4 == s7);             // the output is "true"
System.out.println (s4 == s8);             // the output is "false"
System.out.println (s8 == s9);             // the output is "true"
```

FIGURE 0.1 Illustration of the effect of the `equals` method and `==` operator

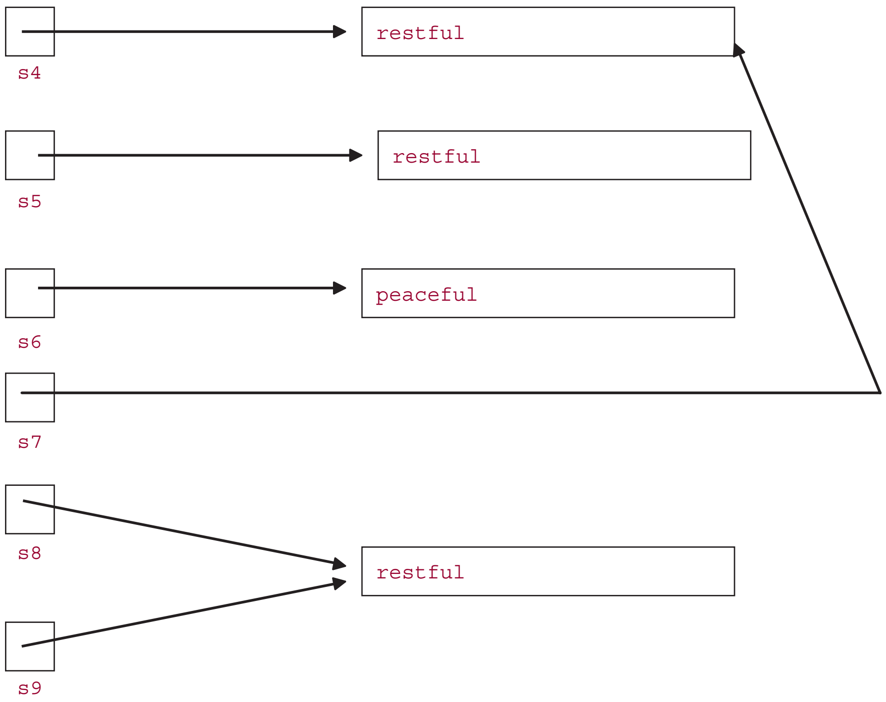


FIGURE 0.2 An internal view of the references and objects in Figure 0.1

a `String` object whose value is “restful”, and `s5` is a reference to a *different* `String` object whose value is also “restful”.

0.2.4 Local Variables

Variables declared within a method—including the method’s parameters—are called ***local variables***. For example, the following method has two local variables, `n` and `j`:

```
/**
 *  Determines if a specified integer greater than 1 is prime.
 *
 *  @param n – the integer whose primality is being tested.
 *
 *  @return true – if n is prime.
 *
 */
public static boolean isPrime (int n)
{
      if (n == 2)
             return true;
      if (n % 2 == 0)
             return false;
```

```
            for (int j = 3; j * j <= n; j = j + 2)
                   if (n % j == 0)
                           return false;
            return true;
      } // method isPrime
```

Local variables must be explicitly initialized before they are used. For example, suppose we have

```
      public void run()
      {
            int k;

            System.out.println (isPrime (k));
      } // method run
```

Compilation will fail, with an error message indicating that "variable `k` might not have been initialized." The phrase "might not have been initialized" in the error message suggests that the compiler does not perform a detailed analysis of the method's code to determine if, in fact, the variable has been properly initialized. For example, the same error message will be generated by the following method:

```
      public void run()
      {
            int k;

            boolean flag = true;

            if (flag)
                     k = 20;
            if (!flag)
                     k = 21;
            System.out.println (isPrime (k));
      } // method run
```

Clearly, if we look at this method as a whole, the variable `k` does receive proper initialization. But if each statement is considered on its own, no such guarantee can be made. The following slight variant is acceptable because the **`if-else`** statement is treated as a unit:

```
      public void run()
      {
            int k;

            boolean flag = true;

            if (flag)
                     k = 20;
            else
                     k = 21;
            System.out.println (isPrime (k));
      } // method run
```

The ***scope*** of an identifier is the region of a program to which that identifier's declaration applies. In the `isPrime` method, the scope of the parameter `n` is the entire function definition, but the scope of the variable `j` is only the **`for`** statement. A compile-time error results if an attempt is made to access an identifier outside of its scope: for example, if we tried to print out the value of `j` outside of the **`for`** statement in the `isPrime` method. This restriction of identifiers to specific code segments—and not, for example, to an entire method—promotes modularity.

To see how it is possible to declare the same identifier more than once in a class, we need to define what a "block" is. A ***block*** is an enclosed sequence of declarations and/or statements enclosed in curly braces { }. For a field identifier, its enclosing block is the entire class enclosed by the curly braces. It is permissible to re-declare the field identifier within a method in the class. But it is illegal to re-declare a local identifier within its block. Special case: for a variable identifier declared in the header of a **`for`** statement, its scope is the entire **`for`** statement. So it is possible to have two **`for`** statements in the same method with identical variable identifiers declared in the headers of those **`for`** statements (but that identifier cannot also be declared outside of those **`for`** statements as a local variable of the method). The following class illustrates the scopes of several identifiers.

```
public class Scope
{
   boolean t = true;

   int x = 99;

   double sample = 8.1;

   public void sample (double x)
   {
       double y = 5;

       x = 5.3;
       for (int t = 0; t < 3; t++)
       {
          int i = t + 4;

          System.out.println (i + t + x);
       } // end of int t's scope; end of i's scope

       for (int t = 0; t < 7; t++)
          x = t;    // end of the scope of this version of int t
   }  // method sample; end of double x's scope; end of y's scope

   public void original()
   {
       System.out.println (t + " " + x + " " + sample);
   } // method original

} // class Scope; end of boolean t's scope; end of int x's scope; end of double sample's scope
```

All three fields—`t`, `x` and `sample`—are re-declared within the method `sample`. But the scope of those three fields includes the method `original`. And note that there is no ambiguity when `sample` is used both as a field identifier and as a method identifier, because a method identifier is always followed by parentheses.

0.2.5 The Scanner Class

One of the recent improvements in Java is provided by the `Scanner` class in `java.util`. A `Scanner` object operates on ***text***: a sequence of lines, separated by end-of-line markers. The text may be input from the keyboard, input from a file, or a string. There are `Scanner` methods to return the rest of the current line, to return the next primitive value, and to return the next string.

Let's start with the three key constructors. Here is a declaration of a `Scanner` object that will read input from the keyboard:

```
Scanner sc = new Scanner (System.in);
```

The following declares a `Scanner` object that will read from the file named "myFile.dat":

```
Scanner scanner = new Scanner (new File ("myFile.dat"));
```

If, instead, we want to scan over the `String` object `line`, we can declare the following:

```
Scanner lineScanner = new Scanner (line);
```

We can now use the `Scanner` object `sc` declared above to read in an **`int`** value representing a test score:

```
int score = sc.nextInt( );
```

In order to understand how the `nextInt` method works, we need to introduce some terminology. A scanner subdivides the text into *tokens* separated by *delimiters*. In the case of the `nextInt` method, the delimiters are ***whitespace*** characters: blanks, end-of-line markers, newline characters, tabs, and so on. The tokens are everything else. The scanning proceeds as follows: First, all whitespace is skipped over. Then the token is read in. If the characters in the token represent an **`int`** value, that value is stored in the variable `score`. In Section 2.2 of Chapter 2, there is a discussion of what happens if the token does not represent an **`int`** value.

We can read in and add up scores until a sentinel of −1 is read. In the following program, we need to utilize the class `java.util.Scanner` in the package `java.util`. Because we will often want several classes from `java.util`, we specify that we want all of the classes from that package to be available. How? By denoting `java.util.*` in the `import` directive, we notify the compiler that the entire package `java.util` is to be made available.

In this program, and in all subsequent programs in this book, the `main` method consists of a single line. A new instance of the class is created with a call to the class's default constructor (automatically supplied by the compiler), and this new instance invokes its `run` method. Here is the complete file:

```
import java.util.*;  // for the Scanner class

public class Sum
{
     public static void main (String[ ] args)
      {
           new Sum().run();
      } // method main
```

```
        public void run()
        {
              final int SENTINEL = -1;

              final String INPUT_PROMPT = "\nPlease enter a test score (or " +
                                          SENTINEL + " to quit): ";

              final String SUM_MESSAGE = "\n\nThe sum of the scores is ";

              Scanner sc = new Scanner (System.in);

              int score,
                   sum = 0;

              while (true)
              {
                   System.out.print (INPUT_PROMPT);
                   score = sc.nextInt();
                   if (score == SENTINEL)
                        break;
                   sum += score;
              } // while
              System.out.println (SUM_MESSAGE + sum);
        } // method run

   } // class Sum
```

A noteworthy feature of this program is the structure of the **while** statement. The loop continues until the sentinel is read in. The execution of the **break** statement causes an abrupt exit of the loop; the next statement to be executed is the `println` that outputs the sum. The loop has only one entrance and only one exit, and that helps us to understand the action of the loop. Also, there is only one place where the prompt is printed, and only one place where input is read.

In that program, why did we use a sentinel instead of allowing the end user to terminate the loop by not entering more values? When `sc.next()` is called, the program will pause until a non-whitespace value is entered (followed by a pressing of the Enter key). In other words, a sentinel is needed to terminate keyboard input. For scanning a line or a file, there will rarely be a sentinel, so a call to the `next()` method should be preceded by a call to the `Scanner` class's `hasNext()` method, which returns true if there is still another token to be scanned in, and false otherwise. This will ensure that the call to `next()` will not cause an abnormal termination due to the lack of a next token.

In the `class Sum`, the end-user was prompted to enter a single **int** value per line. In general, a line may have several **int** values, or even no **int** values. For example, we could have the following:

```
   Scanner sc = new Scanner (System.in);

   int score1 = sc.nextInt(),
       score2 = sc.nextInt(),
       score3 = sc.nextInt();
```

Here is a sample sequence of lines, with the second line blank:

```
85

95 87
```

The variables `score1`, `score2`, and `score3` will now have the values 85, 95, and 87, respectively.

For a slightly more complicated example, the following program reads from a file. Each line in the file consists of a student's name and grade point average. The output is the name of the student with the highest grade point average. There is no sentinel. Instead, the scanning continues as long as the input file has another token, that is, any sequence of characters excluding whitespace. As indicated previously, the `hasNext( )` method returns true if and only if there are any tokens remaining in the file. The `next( )` method returns the next token as a string.

```
import java.util.*; // for the Scanner class

import java.io.*;  // for the File class

public class HighestGPA
{
      public static void main (String[ ] args) throws FileNotFoundException
      {
            new HighestGPA().run();
      } // method main

      public void run() throws FileNotFoundException
      {
            final double NEGATIVE_GPA = -1.0;

            final String NO_VALID_INPUT =
                 "Error: the given file has no valid input.";

            final String BEST_MESSAGE =
                 "\n\nThe student with the highest grade point average is ";

            Scanner fileScanner = new Scanner (new File ("students.dat"));

            String name,
                   bestStudent = null;

            double gpa,
                   highestGPA = NEGATIVE_GPA;

            while (fileScanner.hasNextLine())
            {
                 Scanner lineScanner = new Scanner (fileScanner.nextLine());

                 name = lineScanner.next();
                 gpa = lineScanner.nextDouble();
                 if (gpa > highestGPA)
```

```
                    {
                          highestGPA = gpa;
                          bestStudent = name;
                    } // if
               } // while
               if (highestGPA == NEGATIVE_GPA)
                    System.out.println (NO_VALID_INPUT);
               else
                    System.out.println (BEST_MESSAGE + bestStudent);
         } // method run

     } // class HighestGPA
```

The significance of the clause **throws** FileNotFoundException is explained in Section 2.3.3 of Chapter 2. As described in Section 0.4, the variable bestStudent is initialized to avoid a "might not have been initialized" error message when bestStudent is printed. Note, for example, that if the file "students.dat" is empty, the loop will not be executed, and so bestStudent will not be assigned a value in the loop.

Here are sample contents of the file "students.dat":

```
Larry 3.3
Curly 3.7
Moe 3.2
```

The corresponding output is:

```
The student with the highest grade point average is Curly
```

In the above program, the name of the input file was "hard-wired," that is, actually specified in the code. It is more realistic for the end-user to enter, from the keyboard, the input-file path. Then we need two scanner objects: one to read in the input-file path, and another to read the input file itself. Because a file path may contain blank spaces, we cannot invoke the next() method to read in the input-file path. Instead, we call the nextLine() method, which advances the scanner past the current line, and returns (the remainder of) the current line, excluding any end-of-line marker. Here is the code that replaces the declaration and assignment to fileScanner in that program:

```
final String IN_FILE_PROMPT = "Please enter the path for the input file: ";

Scanner keyboardScanner = new Scanner (System.in);

System.out.print (IN_FILE_PROMPT);

String inFileName = keyboardScanner.nextLine( );

Scanner fileScanner = new Scanner (new File (inFileName));
```

Here are sample input and output for the resulting program, with the input in boldface:

```
Please enter the path for the input file: students.dat

The student with the highest grade point average is Curly
```

Keep in mind that the `next()` method skips over whitespace and returns the next token, but the `nextLine()` method skips past the current line, and returns that current line, excluding any end-of-line marker. Similarly, the `hasNext()` method returns **`true`** if and only if there is another token in the text, while the `hasNextLine()` method returns **`true`** if and only if there is at least one more character—including whitespace—in the text. So if the text has a line of blanks remaining, or even an extra end-of-line marker, `hasNext()` will return **false**, but `hasNextLine( )` will return **true**.

The above example illustrates a pattern we will see over and over in the remaining chapters. A keyboard scanner scans in the path of an input file, a file scanner scans in the lines in the file, and a line scanner scans over a single line.

In the next example, a scanner retrieves each word in a line, and the word is converted to lower-case and printed. The scanner is declared in a method whose only parameter is a line to be parsed into words:

```
public void run()
{
     split ("Here today gone tomorrow");
} // method run


public void split (String line)
{
     Scanner sc = new Scanner (line);

     while (sc.hasNext())
           System.out.println (sc.next().toLowerCase());
} // method split
```

The output will be

```
here
today
gone
tomorrow
```

Unfortunately, if the input contains any non-alphabetic, non-whitespace characters, those characters will be included in the tokens. For example, if the call is

```
split ("Here today, gone tomorrow.");
```

The output will be

```
here
today,
gone
tomorrow.
```

We can override the default delimiter of whitespace with the `useDelimiter (String pattern)` method, which returns a (reference to a) `Scanner` object. For example, if we want the delimiter to be any positive number of non-alphabetic characters, we can explicitly indicate that as follows:

```
Scanner sc = new Scanner (line).useDelimiter ("[^a-zA-Z]+");
```

In the argument to the `useDelimiter method`, the brackets specify a group of characters, the '^' specifies the complement of the characters that follow, and '+' is shorthand for any positive number of occurrences of the preceding group. In other words, we are defining a delimiter as any sequence of one or more occurrences of characters that are non-alphabetic. So if we have included the above `useDelimiter` call, and we have

```
split ("Here today, gone tomorrow.");
```

then the output will be

```
here
today
gone
tomorrow
```

Finally, suppose we want to allow a word to have an apostrophe. Then we include the apostrophe in the class whose complement defines the delimiters:

```
Scanner sc = new Scanner (line).useDelimiter ("[^a-zA-Z']+");
```

If the call is

```
split ("You're 21??  I'll need to see some ID!");
```

then the output will be

```
you're
i'll
need
to
see
some
id
```

You are now prepared to do Lab 0: The `Scanner` Class

Cultural Note: The `Scanner` class enables a user to process a ***regular expression***: a format for identifying patterns in a text. The arguments to the `useDelimiter` methods shown previously are simple examples of regular expressions. In general, regular expressions provide a powerful but somewhat complex means of finding strings of interest in a text. For more information on handling regular expressions in Java, see http://www.txt2re.com and (Habibi, 2004).

0.3 Arrays

An ***array*** is a collection of elements, of the same type, that are stored contiguously in memory. ***Contiguous*** means "adjacent," so the individual elements are stored next to each other.[3] For example, we can create an array of five `String` objects as follows:

```
String [ ] names = new String [5];
```

[3]Actually, all that matters is that, to a user of an array, the elements are stored *as if* they were contiguous.

Here the **new** operator allocates space for an array of five `String` references (each initialized to **null** by the Java Virtual Machine), and returns a reference to the beginning of the space allocated. This reference is stored in `names`.

In order to specify one of the individual elements in an array, an index is used. An ***index*** is an integer expression in square brackets; the value of the expression determines which individual element is being denoted. The smallest allowable index is 0. For example,

```
names [0] = "Cromer";
```

will store a reference to "Cromer" at the zeroth entry in the array (referenced by) `names`.

The size of an array is fixed once the array has been created, but the size need not be determined at compile-time. For example, we can do the following:

```
public void processInput (String s)
{
     int n = new Scanner (s).nextInt();

     String [ ] names = new String [n];
     ...
```

When the `processInput` method is executed at run time, `names` will be assigned a reference to an array of `n` string references, all of which are initialized to **null**. An array is an object, even though there is no corresponding class. We loosely refer to `names` as an "array," even though "array reference" is the accurate term.

The capacity of an array, that is, the maximum number of elements that can be stored in the array, is stored in the array's `length` field. For example, suppose we initialize an array when we create it:

```
double [ ] weights = {107.3, 112.1, 114.4, 119.0, 117.4};
```

We can print out the capacity of this array as follows:

```
System.out.println (weights.length);
```

The output will be

```
5
```

For an array `x`, the value of any array index must be between `0` and `x.length-1`, inclusive. If the value of an index is outside that range, `ArrayIndexOutOfBoundsException` (See Section 2.3 of Chapter 2 for a discussion of exception handling) will be thrown, as in the following example:

```
final int MAX = 10;

double [ ] salaries = new double [MAX];

for (int i = 0; i <= MAX; i++)
       salaries [i] = 0.00;
```

For the first ten iterations of the loop, with `i` going from 0 through 9, each element in the array is initialized to 0.00. But when `i` gets the value 10, an `ArrayIndexOutOfBoundsException` is thrown because `i`'s value is greater than the value of `salaries.length-1`, which is 9.

0.4 Arguments and Parameters

In Java, the relationship between arguments (when a method is called) and parameters (in the called method's heading) is straightforward: a copy of the value of the argument is assigned to the parameter. So the argument is not affected by the method call. For example, consider the following program:

```
public class Nothing
{
      public static void main (String[ ] args)
      {
             new Nothing().run();
      } // method main

      public void run()
      {
             int k = 30;

             triple (k);

             System.out.println (k);
      } // method run

      public void triple (int n)
      {
             n = n * 3;
      } // method triple

} // class Nothing
```

The output will be 30. When the method `triple (int n)` is called, a copy of the value of the argument `k` is assigned to the parameter `n`. So at the beginning of the execution of the call, `n` has the value 30. At the end of the execution of the call, `n` has the value 90, but the argument `k` still has the value 30. Incidentally, the name of the parameter has no effect: the result would have been the same if the parameter had been named `k` instead of `n`.

Next, we look at what happens if the argument is of type reference. As with primitive arguments, a reference argument will not be affected by the method call. But a much more important issue is "What about the object referenced by the argument?" If the object is an array, then the array can be affected by the method call. For example, here is a simple program in which an array is modified during a method call:

```
public class Swap
{
      public static void main (String[ ] args)
      {
             new Swap().run();
      } // method main

      public void run()
      {
             double[ ] x = {5.0, 8.0, 12.2, 20.0};

             swap (x, 1, 2);

             System.out.println (x [1] + " " + x [2]);
      } // method run

      public void swap (double[ ] a, int i, int j)
       {
           double temp = a [i];
           a [i] = a [j];
           a [j] = temp;
       } // method swap

} // class Swap
```

The output will be

```
12.2 8.0
```

In the array referenced by `x`, the elements at indexes 1 and 2 have been swapped.

For objects other than arrays, whether the object can be affected by the method call depends on the class of which the object is an instance. For example, the `String` class is ***immutable***, that is, there are no methods in the `String` class that can modify an already constructed `String` object. So if a `String` reference is passed as an argument in a method call, the `String` object will not be affected by the call. The following program illustrates the immutability of `String` objects.

```
public class Immutable
{
     public static void main (String[ ] args)
     {
          new Immutable().run();
     } // method main

     public void run()
     {
          String s = "yes";
          flip (s);
          System.out.println (s);
     } // method run

     public void flip (String t)
     {
          t = new String ("no");
     } // method flip
} // class Immutable
```

The output will be “yes”. The `flip` method constructed a new `String` object, but did not affect the original `String` object referenced by `s`. Figure 0.3 shows the relationship between the argument `s` and the parameter `t` at the start of the execution of the call to the `flip` method. Figure 0.4 shows the relationship between `s` and `t` at the end of the execution of the method call.

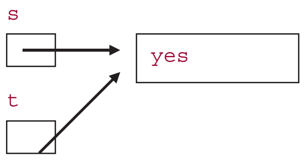


FIGURE 0.3 The relationship between the argument `s` and the parameter `t` at the start of the execution of the `flip` method in the class `Immutable`

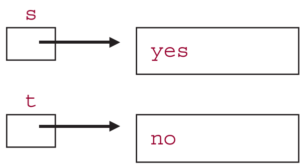


FIGURE 0.4 The relationship between the argument `s` and the parameter `t` at the end of the execution of the `flip` method in the class `Immutable`

Some built-in classes are mutable. For example, the following program illustrates the mutability of `Scanner` objects.

```
import java.util.*;   // for the Scanner class

public class Mutable
{
     public static void main (String[ ] args)
     {
            new Mutable().run();
     } // method main

     public void run()
     {
            Scanner sc = new Scanner ("yes no maybe");

            System.out.println (sc.next());
            advance (sc);
            System.out.println (sc.next());
     } // method run

       public void advance (Scanner scanner)
       {
             scanner.next();
       } // method advance

} // class Mutable
```

The output from this program will be

```
yes
maybe
```

The string "no" was returned when `scanner.next()` was called in the `advance` method, and that call advanced the current position of the `Scanner` object (referenced by both `sc` and `scanner`) beyond where "no" is. Because of this alteration of the `Scanner` object, the second call to `println` in the `run` method prints out "maybe".

To summarize this section, an argument is never affected by a method call. But if the argument is a reference, the corresponding object may (for example, an array object or a `Scanner` object) or may not (for example, a `String`, `Integer`, or `Double` object) be affected by the call.

0.5 Output Formatting

What do you think the output will be from the following code fragment?

```
double grossPay = 800.40;

System.out.println (grossPay);
```

The output will not be

```
800.40
```

but rather

```
800.4
```

To get two fractional digits printed, we need to convert `grossPay` to a string with two fractional digits. This is accomplished by first creating an instance of the `DecimalFormat` class (in the package `java.text`) with a fixed format that includes two fractional digits, and then applying that object's `format` method to `grossPay`. The code is

```
DecimalFormat d = new DecimalFormat ("0.00");

double grossPay = 800.40;

System.out.println (d.format (grossPay));
```

The fixed format, "0.00", is the argument to the `DecimalFormat` constructor. That format specifies that the `String` object returned by the `format` method will have at least one digit to the left of the decimal point and exactly two digits to the right of the decimal point. The fractional part will be rounded to two decimal digits; for example, suppose we have

```
DecimalFormat d = new DecimalFormat ("0.00");

double grossPay = 800.416;

System.out.println (d.format (grossPay));
```

Then the output will be

```
800.42
```

To make the output look more like a dollar-and-cents amount, we can ensure that `grossPay` is immediately preceded by a dollar sign, and a comma separates the hundreds digit from the thousands digit:

```
DecimalFormat d = new DecimalFormat (" $#,###.00");

double grossPay = 2800.4;

System.out.println (d.format (grossPay));
```

The output will be

```
$2,800.40
```

The `DecimalFormat` class is in the package `java.text`, so that package must be imported at the beginning of the file in which the formatting is performed.

An alternative to the output formatting just described is provided by the `printf` method, which is similar to the `printf` function in the C language. For example, the following will print `grossPay` with at least one digit to the left of the decimal point and exactly two digits (rounded) to the right of the decimal point:

```
System.out.printf ("%1.2f", grossPay);
```

The first field is called the ***format***, an expression in quotes that starts with a percent sign. The character 'f'—called a "flag"—signifies that `grossPay` will be printed with fractional digits. The "1.2"—the "width" and "precision"—signifies that there will be at least one digit (even if it is a zero) to the left of the decimal point, and exactly two digits (rounded) to the right of the decimal point. For example, suppose we have

```
double grossPay = 1234.567;

System.out.printf ("%1.2f", grossPay);
```

Then the output will be

```
1234.57
```

If there are additional values to be printed, the format for each value starts with a percent sign inside the format. For example,

```
double grossPay = 1234.567;

String name = "Jonathan Rowe";

System.out.printf ("%s %1.2f", name, grossPay);
```

The 's' flag indicates a string. The output will be

```
Jonathan Rowe 1234.57
```

We can ensure there will be a comma between the hundreds and thousands digits by including the comma flag, ','. For example, suppose we have

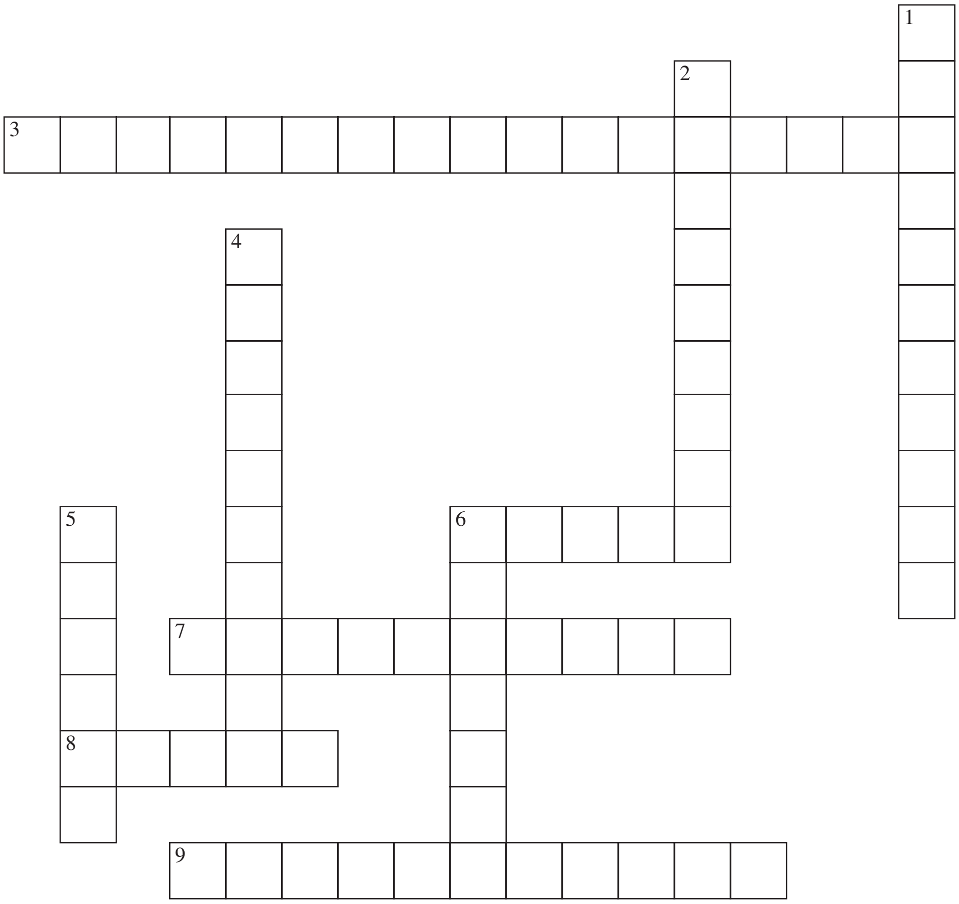
```
double grossPay = 1234.567;

String name = "Jonathan Rowe";

System.out.printf ("%s $%,1.2f", name, grossPay);
```

Then the output will be

```
Jonathan Rowe $1,234.57
```

More details on formatting in the `printf` method can be found in the `Format` class in the package java.util.

CROSSWORD PUZZLE



www.CrosswordWeaver.com

ACROSS

3. A variable that can contain the address of an object.

6. The region of a program to which an identifier's declaration applies.

7. The separation of a program into components that are coherent units.

8. A new, that is, non-primitive type.

9. In a class, a method whose name is the same as the class's name and whose purpose is to initialize a calling object's fields.

DOWN

1. The ability of a class to have methods with the same method identifier but different parameter lists.

2. A method's identifier together with the number and types of parameters in order.

4. Adjacent, as when individual elements are stored next to each other.

5. An instance of a class.

6. A class that is useful for parsing keyboard input, file input and strings.

PROGRAMMING EXERCISES

0.1 The package java.lang includes the `Integer` class. In that class, what is the definition of `MAX_VALUE` ? The number 0x7fffffff is in ***hexadecimal***, that is, base 16, notation. To indicate hexadecimal notation, start with "0x", followed by the hexadecimal value. In hexadecimal, there are sixteen digits: 0, 1, 2, . . ., 9, a, b, c, d, e, and f. The digits 0 through 9 have the same value as in decimal, and the letters 'a' through 'f' have the following decimal values:

a	10
b	11
c	12
d	13
e	14
f	15

The decimal representation of hexadecimal 7ffffff is

$$7 * 16^7 + 15 * 16^6 + 15 * 16^5 + 15 * 16^4 + 15 * 16^3 + 15 * 16^2 + 15 * 16^1 + 15$$
$$= 2147483647$$

Similarly, the decimal value of `MIN_VALUE` is -2147483648. Hypothesize the decimal value of each of the following:

```
Integer.MAX_VALUE + 1

Math.abs (Integer.MIN_VALUE)
```

Test your hypotheses with a small program that calls the `System.out.println` method.

0.2 Suppose we have the following:

```
int a = 37,
    b = 5;

System.out.println (a - a / b * b - a % b);
```

Hypothesize what the output will be. Test your hypothesis by executing a small program that includes that code. Can you find another pair of positive **`int`** values for `a` and `b` that will produce different output? Explain.

0.3 Hypothesize the output from the following:

```
System.out.println (1 / 0);
System.out.println (1.0 / 0);
```

Test your hypotheses by executing a small program that includes that code.

0.4 In the `String` class, read the specification for the `indexOf` method that takes a `String` parameter. Then hypothesize the output from the following

```
System.out.println ("The snow is now on the ground.".indexOf ("now"));
```

Test your hypothesis by executing a small program that includes that code.

0.5 In the `String` class, read the specification for the `indexOf` method that takes a `String` parameter and an **`int`** parameter. Then hypothesize the output from the following

```
System.out.println ("The snow is now on the ground.".indexOf ("now", 8));
```

Test your hypothesis by executing a small program that includes that code.

0.6 Write and run a small program in which an input string is read in and the output is the original string with each occurrence of the word “is” replaced by “was”. No replacement should be made for an embedded occurrence, such as in “this” or “isthmus”.

0.7 Write and run a small program in which an input string is read in and the output is the original string with each occurrence of the word “is” replaced by “is not”. No replacement should be made for an embedded occurrence, such as in “this” or “isthmus”.

0.8 Write and run a small program in which the end user enters three lines of input. The first line contains a string, the second line contains a substring to be replaced, and the third line contains the replacement substring. The output is the string in the first line with each occurrence of the substring in the second line replaced with the substring in the third line. No replacement should be made for an embedded occurrence, in the first line of the substring in the second line.

0.9 Study the following method:

```
public void mystery (int n)
{
       System.out.print ("For n = " + n);
       while (n > 1)
              if (n % 2 == 0)
                     n = n / 2;
              else
                     n = 3 * n + 1;
       System.out.println (", the loop terminated.");
} // method mystery
```

Trace the execution of this method when `n` = 7. In the same class as the method `mystery`, develop a `main` method and a `run` method to show that the **`while`** statement in the method mystery successfully terminates for any positive integer `n` less than 100. Cultural note: It can be shown that this method terminates for all **`int`** values greater than 0, but it is an open question whether the method terminates for all integer values greater than 0.

Object-Oriented Concepts

CHAPTER 1

CHAPTER OBJECTIVES

1. Compare a user's view of a class with a developer's view of that class.
2. Understand how inheritance promotes code re-use.
3. Understand how polymorphic references can be useful.
4. Be able to create class diagrams in the Unified Modeling Language.

1.1 Data Abstraction

A user of a class concentrates on the class's method **specifications**, that is, ***what*** a class provides. A developer of a class, on the other hand, focuses on the class's fields and method definitions, that is, ***how*** the class is defined. This separation—called ***data abstraction***—of what from how is an essential feature of object-oriented programming. For example, programmers who *use* the `String` class will not care about the fields that represent a string or how the methods are defined. Such details would be of no help when you are trying to develop a class that *uses* the `String` class, but were essential to the developers of the `String` class.

In general, suppose you are a programmer who is developing class `A`, and during development, you decide that you will need the services of class `B`. If someone else has already completed the definition of class `B`, you should simply use that class rather than re-inventing the wheel. But even if you must define the class, `B` yourself, you can simply create the method specifications for class `B` and postpone any further work on class `B` until after you have completed the development of class `A`. By working with class `B`'s method specifications, you increase the independence of class `A`: its effectiveness will not be affected by any changes to class `B` that do not affect the method specifications of class `B`.

When users focus on what a class provides to users rather than on the implementation details of that class, they are applying the ***Principle of Data Abstraction***: A user's code should not access the implementation details of the class used.

One important application of the Principle of Data Abstraction is that if class `A` simply uses class `B`, then class `A`'s methods should not access class `B`'s fields. In fact, class `B`'s fields should be accessed only in class `B`'s method definitions. This turns out to be a benefit to users of class `B` because they will be unaffected if the developer of class `B` decides to replace the old fields with new ones. For example, suppose the following definition is made outside of the `String` class:

```
String name;
```

Currently, one of the fields in the `String` class is an **`int`** field named `count`. But an expression such as

```
name.count
```

would be a violation of the Principle of Data Abstraction because whether or not the `String` class has a `count` field is an implementation detail. The developer of the `String` class is free to make any changes to the `String` class that do not affect the method specifications.

Most programming languages, including Java, have features that enable a developer of a class to force users to adhere to the Principle of Data Abstraction. These enforcement features are collectively known as "information hiding." We will discuss information hiding in Section 1.5 after we have introduced several of the relevant features.

We noted earlier that the Principle of Data Abstraction is a benefit to users of a class because they are freed from reliance on implementation details of that class. This assumes, of course, that the class's method specifications provide all of the information that a user of that class needs. The developer of a class should create methods with sufficient functionality that users need not rely on any implementation details. That functionality should be clearly spelled out in the method specifications. In particular, the user should be told under what circumstances a method call will be legal, and what the effect of a legal call will be.

In a method specification, the first sentence in a javadoc comment block is called the ***postcondition:*** the effect of a legal call to the method. The information relating to the legality of a method call is known as the ***precondition*** of the method. The precondition may be stated explicitly within or after the postcondition (for example, "The array must be sorted prior to making this call.") or implicitly from the exception information (for example, any call that throws an exception is illegal). The interplay between a method's precondition and postcondition determines a ***contract*** between the developer and the user. The terms of the contract are as follows:

> *If the user of the method ensures that the precondition is true before the method is invoked, the developer guarantees that the postcondition will be true at the end of the execution of the method.*

We can summarize our discussion of classes so far by saying that from the **developer's** perspective, a class consists of fields and the definitions of methods that act on those fields. A **user's** view is an abstraction of this: A class consists of method specifications.

The Java Collections Framework is, basically, a hierarchy of thoroughly tested classes that are useful in a variety of applications. The programs in this book will *use* the Java Collections Framework, so those programs will not rely on the definitions of the framework's methods. We will provide method specifications and an overview of most of the classes. Occasionally, to give you experience in reading the code of professional programmers, you will get to study the fields and some of the method definitions.

In Section 1.2, we see the value of classes that have undefined methods.

1.2 Abstract Methods and Interfaces

Up to now, we have used method specifications for the purpose of promoting data abstraction. That is, a user should focus on the method specifications in a class and ignore the class's fields and method definitions. Some methods don't even have a definition, and it turns out that this can be helpful to programmers. For example, suppose we want to create classes for circles, rectangles and other figures. In each class, there will be methods to draw the figure and to move the figure from one place on the screen to another place on the screen. The `Circle` class, for example, will have a `draw` method and a `move` method based on the center of the circle and its radius. Here are two method specifications and related constant identifiers that will apply to all of the figure classes:

```
final static int MAX_X_COORD = 1024;

final static int MAX_Y_COORD = 768;

/**
 *  Draws this Figure object centered at the given coordinates.
 *
 *  @param x - the X coordinate of the center point of where this Figure object
 *             will be drawn.
 *  @param y - the Y coordinate of the center point of where this Figure object
 *             will be drawn.
 *
 */
public void draw(int x, int y)


/**
 *  Moves this Figure object to a position whose center coordinates are specified.
 *
 *  @param x - the X coordinate of the center point of where this Figure object
 *             will be moved to.
 *  @param y - the Y coordinate of the center point of where this Figure object
 *             will be moved to.
 *
 */
public void move (int x, int y)
```

Each different type of figure will have to provide its own definitions for the `draw` and `move` methods. But by requiring that those definitions adhere to the above specifications, we introduce a consistency to any application that uses the figure classes. A user of one of those classes knows the exact format for the `draw` and `move` methods—and that will still be true for classes corresponding to new figure-types.

Java provides a way to enforce this consistency: the interface. Each method heading is followed by a semicolon instead of a definition. Such a method is called an ***abstract method***. An ***interface*** is a collection of abstract methods and constants. There are no defined methods and no fields. For example, here is the interface for figures:

```
public interface Figure
{
       final static int MAX_X_COORD = 1024;

       final static int MAX_Y_COORD = 768;

       /**
        *  Draws this Figure object centered at the given coordinates.
        *
        *  @param x - the X coordinate of the center point of where this Figure
        *       object will be drawn.
        *  @param y - the Y coordinate of the center point of where this Figure
        *             object will be drawn.
        *
        */
       void draw(int x, int y);


       /**
        *  Moves this Figure object to a position whose center coordinates are
        *  specified.
        *
```

```
         *  @param x - the X coordinate of the center point of where this Figure
         *             object will be moved to.
         *  @param y - the Y coordinate of the center point of where this Figure
         *             object will be moved to.
         *
         */
         void move (int x, int y);

     } // interface Figure
```

The **interface** Figure has two constants (MAX_X_COORD and MAX_Y_COORD) and two abstract methods, (draw and move). In any interface, all of the method identifiers and constant identifiers are public, so the declarations need not include the visibility modifier **public**.

When a class provides method definitions for an interface's methods, the class is said to ***implement*** the interface. The class may also define other methods. For example, here is part of a declaration of the Circle class:

```
public class Circle implements Figure
{
      // declaration of fields:
      private int xCoord,
                  yCoord,
                  radius;

      // constructors to initialize x, y and radius:
      ...
      /** (javadoc comments as above)
      */
      public draw (int x, int y)
      {
            xCoord = x;
            yCoord = y;

           // draw circle with center at (xCoord, yCoord) and radius:
            ...
      } // method draw

     // definitions for move and any other methods:
      ...

} // class Circle
```

The reserved word **implements** signals that class Circle provides method definitions for the methods whose specifications are in the interface Figure. Interfaces do not include constructors because constructors are always class specific. The incompleteness of interfaces makes them uninstantiable, that is, we cannot create an instance of a Figure object. For example, the following is illegal:

```
Figure myFig = new Figure();  // illegal
```

In the method specifications in the Figure interface, the phrase "this Figure object" means "this object in a class that implements the Figure interface."

Of what value is an interface? In general, an interface provides a common base whose method specifications are available to any implementing class. Thus, an interface raises the comfort level of users because they know that the specifications of any method in the interface will be adhered to in any class that implements the interface. In practice, once a user has seen an interface, the user knows a lot about any implementing class. Unlike the interface, the implementing class will have constructors, and may define other methods in addition to those specified in the interface.

1.2.1 Abstract Data Types and Data Structures

An ***abstract data type*** consists of a collection of values, together with a collection of operations on those values. In object-oriented languages such as Java, abstract data types correspond to interfaces in the sense that, for any class that implements the interface, a user of that class can:

a. create an instance of that class; (“instance” corresponds to “ value”)

b. invoke the public methods of that class (“public method” corresponds to “operation”).

A ***data structure*** is the implementation of an abstract data type. In object-oriented languages, a developer implements an interface with a class. In other words, we have the following associations:

general term	**object-oriented term**
abstract data type	Interface
data structure	class

A user is interested in abstract data types—interfaces—and a class’s method specifications, while a developer focuses on data structures, namely, a class’s fields and method definitions. For example, one of the Java Collections Framework’s interfaces is the `List` interface; one of the classes that implement that interface is `LinkedList`. When we work with the `List` interface or the `LinkedList` method specifications, we are taking a user’s view. But when we consider a specific choice of fields and method definitions in `LinkedList`, we are taking a developer’s view.

In Chapter 0, we viewed the `String` class from the user’s perspective: what information about the `String` class is needed by users of that class? A ***user*** of a class writes code that includes an instance of that class. Someone who simply executes a program that includes an instance of a class is called an ***end-user***. A ***developer*** of a class actually creates fields and method definitions. In Section 1.2.2, we will look at the developer’s perspective and compare the user’s and developer’s perspectives. Specifically, we create an interface, and utilize it and its implementing classes as vehicles for introducing several object-oriented concepts.

1.2.2 An Interface and a Class that Implements the Interface

This section is part of an extended example that continues through the rest of the chapter and illustrates several important object-oriented concepts. Let’s create a simple interface called `Employee` for the employees in a company. The information for each employee consists of the employee’s name and gross weekly pay. To lead into the method specifications, we first list the responsibilities of the interface, that is, the services provided to users of any class that implements the `Employee` interface. The responsibilities of the `Employee` interface are:

1. to return the employee’s name;
2. to return the employee’s gross pay;
3. to return a `String` representation of an employee.

These responsibilities are refined into the following interface:

```
import java.text.DecimalFormat;

public interface Employee
{

       final static DecimalFormat MONEY = new DecimalFormat (" $0.00");
             // a class constant used in formatting a value in dollars and cents

      /**
       *  Returns this Employee object's name.
       *
       *  @return this Employee object's name.
       *
       */
       String getName();


      /**
       *  Returns this Employee object's gross pay.
       *
       *  @return this Employee object's gross pay.
       *
       */
       double getGrossPay();


      /**
       *  Returns a String representation of this Employee object with the name
       *  followed by a space followed by a dollar sign followed by the gross
       *  weekly pay, with two fractional digits (rounded).
       *
       *  @return a String representation of this Employee object.
       *
       */
       String toString();

} // interface Employee
```

The identifier `MONEY` is a constant identifier—indicated by the reserved word **`final`**. The reason for declaring a `MONEY` object is to facilitate the conversion of a **`double`** value such as gross pay into a string in dollars-and-cents format suitable for printing. Instead of having a separate copy of the `MONEY` object for each instance of each class that implements the `Employee` interface, there is just one `MONEY` object shared by all instances of implementing classes. This sharing is indicated by the reserved word **`static`**.

The phrase "this `Employee` object" means "the calling object in a class that implements the `Employee` interface."

The `Employee` interface's method specifications are all that a user of any implementing class will need in order to invoke those methods. A developer of the class, on the other hand, must decide what fields to have and then define the methods. A convenient categorization of employees is full-time and part-time. Let's develop a `FullTimeEmployee` implementation of the `Employee` interface.

For example, a developer may well decide to have two fields: name (a String reference) and grossPay (a **double**). The complete method definitions are developed from the fields and method specifications. For example, here is a complete declaration of the FullTimeEmployee class; the next few sections of this chapter will investigate various aspects of the declaration.

```
import java.text.DecimalFormat;

public class FullTimeEmployee implements Employee
{
  private String name;

  private double grossPay;


  /**
  *  Initializes this FullTimeEmployee object to have an empty string for the
  *  name and 0.00 for the gross pay.
  *
  */
  public FullTimeEmployee()
  {
     final String EMPTY_STRING = "";

     name = EMPTY_STRING;
     grossPay = 0.00;
  } // default constructor


  /**
   *  Initializes this FullTimeEmployee object's name and gross pay from a
   *  a specified name and gross pay.
   *
   *  @param name - the specified name.
   *  @param grossPay - the specified gross pay.
   *
   */
  public FullTimeEmployee (String name, double grossPay)
  {
     this.name = name;
     this.grossPay = grossPay;
  } // 2-parameter constructor


  /**
   *   Returns the name of this FullTimeEmployee object.
   *
   *  @return the name of this FullTimeEmployee object.
   *
   */
  public String getName()
  {
```

```
            return name;
        } // method getName


        /**
         *   Returns the gross pay of this FullTimeEmployee object.
         *
         *  @return the gross pay of this FullTimeEmployee object.
         *
         */
        public double getGrossPay()
        {
            return grossPay;
        } // method getGrossPay


        /**
         *  Returns a String representation of this FullTimeEmployee object with the
         *  name followed by a space followed by a dollar sign followed by the gross
         *  weekly pay, with two fractional digits (rounded), followed by "FULL TIME".
         *
         *  @return a String representation of this FullTimeEmployee object.
         *
         */
        public String toString()
        {
            final String EMPLOYMENT_STATUS = "FULL TIME";

            return name + MONEY.format (grossPay) + EMPLOYMENT_STATUS;
                  // the format method returns a String representation of grossPay.
        } // method toString

      } // class FullTimeEmployee
```

In the two-parameter constructor, one of the parameters is `name`. The reserved word **`this`** is used to distinguish between the scope of the field identifier `name` and the scope of the parameter `name`. In any class, the reserved word **`this`** is a reference to the calling object, so **`this.`**`name` refers to the `name` field of the calling object, and `name` by itself refers to the parameter `name`. Similarly, **`this`**`.grossPay` refers to the calling object's `grossPay` field, and `grossPay` by itself refers to the parameter `grossPay`.

In the other methods, such as `toString()`, there is no `grossPay` parameter. Then the appearance of the identifier `grossPay` in the body of the `toString()` method refers to the `grossPay` field of the object that called the `toString()` method.

The same rule applies if a method identifier appears without a calling object in the body of a method. For example, here is the definition of the `hasNextLine()` method in the `Scanner` class:

```
      public boolean hasNextLine()
      {

            saveState();
            String result = findWithinHorizon(
                              ".*("+LINE_SEPARATOR_PATTERN+")|.+$", 0);
            revertState();
            return (result != null);
      }
```

There is no calling-object reference specified for the invocation of the `saveState()` method, and so the object assumed to be invoking `saveState()` is the object that called `hasNextLine()`. Similarly, the methods `findWithinHorizon` and `revertState` are being called by that same object. Here is the general rule, where a ***member*** is either a field or a method:

If an object has called a method and a member appears without an object reference in the method definition, the member is part of the calling object.

As you can see from the declaration of the `FullTimeEmployee` class, in each method definition there is at least one field that appears without an object reference. In fact, in almost every method definition in almost every class you will see in subsequent chapters, there will be at least one field that appears without an object (reference). Then the field is part of the calling object.

1.2.3 Using the `FullTimeEmployee` Class

As an example of how to use the `FullTimeEmployee` class, we can find the best-paid of the full-time employees in a company. The information for each employee will be on one line in a file, and the name of the file will be scanned in from System.in.

For convenience, the following `Company` class includes a `main` method. For the sake of an object orientation, that `main` method simply invokes the `Company` class's `run` method on a newly constructed `Company` instance; all of the `main` methods from here on will be one-liners. The `run` method calls a `findBestPaid` method to return the best-paid full-time employee, or **`null`** if there were no employees scanned in. Finally, the `findBestPaid` method invokes a `getNextEmployee` method to handle the details of constructing a `FullTimeEmployee` instance from a name and a gross pay.

Here is the complete program file, with three `Scanner` objects, one to scan the name of the file of employees, one to scan over that file, and one to scan a line in that file:

```
import java.util.*; // for the Scanner class

import java.io.*;  // for the FileNotFoundException class – see Section 2.3

public class Company
{
   public static void main (String[ ] args) throws FileNotFoundException
   {
      new Company().run();
   } // method main

   /**
   *  Determines and prints out the best paid of the full-time employees
   *  scanned in from a specified file.
   *
   */
   public void run() throws FileNotFoundException  // see Section 2.3
```

```
{
   final String INPUT_PROMPT = "Please enter the path for the file of employees: ";
   final String BEST_PAID_MESSAGE =
      "\n\nThe best-paid employee (and gross pay) is ";

   final String NO_INPUT_MESSAGE =
      "\n\nError: There were no employees scanned in.";

   String fileName;

   System.out.print (INPUT_PROMPT);
   fileName = new Scanner (System.in).nextLine();
   Scanner sc = new Scanner (new File (fileName));

   FullTimeEmployee bestPaid = findBestPaid (sc);

   if (bestPaid == null)
       System.out.println (NO_INPUT_MESSAGE);
   else
       System.out.println (BEST_PAID_MESSAGE + bestPaid.toString());
} // method run


/**
*  Returns the best paid of all the full-time employees scanned in.
*
*  @param sc - the Scanner object used to scan in the employees.
*
*  @return the best paid of all the full-time employees scanned in,
*          or null there were no employees scanned in.
*
*/
public FullTimeEmployee findBestPaid (Scanner sc)
{
   FullTimeEmployee full,
                    bestPaid = new FullTimeEmployee();

   while (sc.hasNext())
   {
       full = getNextEmployee (sc);
       if (full.getGrossPay() > bestPaid.getGrossPay())
           bestPaid = full;
   } //while
   if (bestPaid.getGrossPay() == 0.00)
        return null;
   return bestPaid;
} // method findBestPaid

/**
 *  Returns the next full-time employee from the file scanned by a specified Scanner
```

```
     *  object.
     *
     *  @param sc - the Scanner object over the file.
     *
     *  @return the next full-time employee scanned in from sc.
     *
     */
    private FullTimeEmployee getNextEmployee (Scanner sc)
    {
       Scanner lineScanner = new Scanner (sc.nextLine());

       String name = lineScanner.next();

       double grossPay = lineScanner.nextDouble();

       return new FullTimeEmployee (name, grossPay);
    } // method getNextEmployee

} // class Company
```

The above code is available from the Chapter 1 directory of the book's website. If the file name scanned in from System.in is "full.in1", and the corresponding file contains

```
a 1000.00
b 3000.00
c 2000.00
```

then the output will be

```
The best-paid employee (and gross pay) is b $3000.00  FULL TIME
```

As noted earlier, we should use existing classes whenever possible. What if a class has most, but not all, of what is needed for an application? We could simply scrap the existing class and develop our own, but that would be time consuming and inefficient. Another option is to copy the needed parts of the existing class and incorporate those parts into a new class that we develop. The danger with that option is that those parts may be incorrect or inefficient. If the developer of the original class replaces the incorrect or inefficient code, our class would still be erroneous or inefficient. A better alternative is to use inheritance, explained in Section 1.3.

1.3 Inheritance

We should write program components that are reusable. For example, instead of defining a method that calculates the average gross pay of 10 employees, we would achieve wider applicability by defining a method that calculates the average gross pay of any number of employees. By writing reusable code, we not only save time, but we also avoid the risk of incorrectly modifying the existing code.

One way that reusability can be applied to classes is through a special and powerful property of classes: inheritance. ***Inheritance*** is the ability to define a new class that includes all of the fields and some or all of the methods of an existing class. The previously existing class is called the ***superclass***. The new class, which may declare new fields and methods, is called the ***subclass***. A subclass may also ***override*** existing methods by giving them method definitions that differ from those in the superclass.[1] The subclass is said to ***extend*** the superclass.

For an example of how inheritance works, let's start with the class `FullTimeEmployee` defined in Section 1.2.2. Suppose that several applications use `FullTimeEmployee`. A new application involves finding the best-paid, full-time hourly employee. For this application, the information on an hourly employee consists of the employee's name, hours worked (an **`int`** value) and pay rate (a **`double`** value). Assume that each employee gets time-and-a-half for overtime (over 40 hours in a week). If the hourly employee did not work any overtime, the gross pay is the hours worked times the pay rate. Otherwise, the gross pay is 40 times the pay rate, plus the overtime hours times the pay rate times 1.5.

We could alter `FullTimeEmployee` by adding `hoursWorked` and `payRate` fields and modifying the methods. But it is risky to modify, for the sake of a new application, a class that is being used successfully in existing applications. The underlying concept is known as the **Open-Closed Principle**: Every class should be both open (extendible through inheritance) and closed (stable for existing applications).

Instead of rewriting `FullTimeEmployee`, we will create `HourlyEmployee`, a subclass of `FullTimeEmployee`. To indicate that a class is a subclass of another class, the subclass identifier is immediately followed by the reserved word **`extends`**. For example, we can declare the `HourlyEmployee` class to be a subclass of `FullTimeEmployee` as follows:

```
public class HourlyEmployee extends FullTimeEmployee
{
      . . .
```

Each `HourlyEmployee` object will have the information from `FullTimeEmployee`—name and gross pay—as well as hours worked and pay rate. These latter two will be fields in the `HourlyEmployee` class. To lead us into a discussion of the relationship between the `FullTimeEmployee` fields and the `HourlyEmployee` fields, here is a constructor to initialize an `HourlyEmployee` instance from a name, hours worked, and pay rate (`MAX_REGULAR_HOURS` is a constant identifier with a current value of 40, and `OVERTIME_FACTOR` is a constant identifier with a current value of 1.5).

```
/**
*  Initializes this full-time HourlyEmployee object's name, hours worked, pay rate, and
*  gross pay from a a specified name, hours worked and pay rate.  If the hours worked
*  is at most MAX_REGULAR_HOURS, the gross pay is the hours worked times
*  the pay rate. Otherwise, the gross pay is MAX_REGULAR_HOURS times the
*  pay rate, plus the pay rate times OVERTIME_FACTOR for all overtime hours.
*
*  @param name - the specified name.
*  @param hoursWorked - the specified hours worked.
*  @param payRate - the specified pay rate.
*
*/
public HourlyEmployee (String name, int hoursWorked, double payRate)
{
   this.name = name;
   this.hoursWorked = hoursWorked;
   this.payRate = payRate;
```

[1]Don't confuse method overriding with method overloading (discussed in Section 0.2.1 of Chapter 0): having two methods in the same class with the same method identifier but different signatures.

```
    if (hoursWorked <= MAX_REGULAR_HOURS)
    {
        regularPay = hoursWorked * payRate;
        overtimePay = 0.00;
    } // if
    else
    {
        regularPay = MAX_REGULAR_HOURS * payRate;
        overtimePay = (hoursWorked - MAX_REGULAR_HOURS) *
                      (payRate * OVERTIME_FACTOR);
    } // else
    grossPay = regularPay + overtimePay;
} // 3-parameter constructor
```

Notice that in the definition of this 3-parameter constructor for `HourlyEmployee`, the `name` and `grossPay` fields from the `FullTimeEmployee` class are treated as if they had been declared as fields in the `HourlyEmployee` class. The justification for this treatment is that an `HourlyEmployee` object is also a `FullTimeEmployee` object, so every `FullTimeEmployee` field is also an `HourlyEmployee` field. But the `name` and `grossPay` fields in the `FullTimeEmployee` class were given **`private`** visibility, which precludes their usage outside of the declaration of the `FullTimeEmployee` class. Can we change the visibility of those fields to **`public`**? That would be a bad choice, because then any user's code would be allowed to access (and modify) those fields. What we need is a visibility modifier for a superclass field that allows access by subclass methods but not by arbitrary user's code. The solution is found in the next section.

1.3.1 The `protected` Visibility Modifier

We noted above that subclass methods—but not user's code in general—should be able to access superclass fields. This suggests that we need a visibility modifier that is less restrictive than **`private`** (to allow subclass access) but more restrictive than **`public`** (to prohibit access by arbitrary user's code). The compromise between **`private`** and **`public`** visibility is **`protected`** visibility. We change the declaration of the `FullTimeEmployee` fields as follows:

```
protected String name;

protected double grossPay;
```

These declarations enable any subclass of `FullTimeEmployee` to access the `name` and `grossPay` fields as if they were declared within the subclass itself. This makes sense because an `HourlyEmployee` object is a `FullTimeEmployee` object as well. So the `HourlyEmployee` class has two inherited fields (`name` and `grossPay`) as well as those explicitly declared in `HourlyEmployee` (`hoursWorked`, `payRate`, and for convenience, `regularPay` and `overtimePay`).

The subclass `HourlyEmployee` can access all of the fields, from `FullTimeEmployee`, that have the **`protected`** modifier. Later on, if a subclass of `HourlyEmployee` is created, we would want that subclass's methods to be able to access the `HourlyEmployee` fields—as well as the `FullTimeEmployee` fields. So the declarations of the `HourlyEmployee` fields should also have the **`protected`** modifier:

```
protected int hoursWorked;

protected double payRate,
                 regularPay,
                 overtimePay;
```

The HourlyEmployee class will have a default constructor as well as the 3-parameter constructor defined earlier in Section 1.3. The FullTimeEmployee methods getName and getGrossPay are inherited as is by the HourlyEmployee class. The getHoursWorked, getPayRate, getRegularPay, and getOvertimePay methods are explicitly defined in the HourlyEmployee class.

The toString() method from the FullTimeEmployee class will be overridden in HourlyEmployee to include the word "HOURLY". The override can be accomplished easily enough: we copy the code from the toString() method in FullTimeEmployee, and append "HOURLY" to the String returned:

```
     return name + MONEY.format (grossPay) + "HOURLY";
```

But, as noted at the end of Section 1.2.3, copying code is dangerous. Instead, the definition of toString() in the HourlyEmployee class will call the toString() method in the FullTimeEmployee class. To call a superclass method, use the reserved word **super** as the calling object:

```
     return super.toString() + "HOURLY";
```

Here is the complete HourlyEmployee.java file:

```
import java.text.DecimalFormat;

public class HourlyEmployee extends FullTimeEmployee implements Employee
{
   // for full-time hourly employees

   public final static int MAX_REGULAR_HOURS = 40;

   public final static double OVERTIME_FACTOR = 1.5;

   protected int hoursWorked;

   protected double payRate,
                    regularPay,
                    overtimePay;

   /**
    *  Initializes this full-time HourlyEmployee object to have an empty string for
    *  the name, 0 for hours worked, 0.00 for the pay rate, grossPay, regularPay
    *  and overtimePay.
    *
    */
   public HourlyEmployee()
   {
       hoursWorked = 0;
       payRate = 0.00;
       regularPay = 0.00;
       overtimePay = 0.00;
   } // default constructor
```

```
/**
 *  Initializes this full-time HourlyEmployee object's name and gross pay from a
 *  a specified name, hours worked and pay rate.  If the hours worked is
 *  at most MAX_REGULAR_HOURS, the gross pay is the hours worked times
 *  the pay rate. Otherwise, the gross pay is MAX_REGULAR_HOURS time the
 *  pay rate, plus the pay rate times OVERTIME_FACTOR for all overtime hours.
 *
 *  @param name - the specified name.
 *  @param hoursWorked - the specified hours worked.
 *  @param payRate - the specified pay rate.
 *
 */
public HourlyEmployee (String name, int hoursWorked, double payRate)
{
   this.name = name;
   this.hoursWorked = hoursWorked;
   this.payRate = payRate;

   if (hoursWorked <= MAX_REGULAR_HOURS)
   {
       regularPay = hoursWorked * payRate;
       overtimePay = 0.00;
   } // if
   else
   {
       regularPay = MAX_REGULAR_HOURS * payRate;
       overtimePay = (hoursWorked - MAX_REGULAR_HOURS) *
                     (payRate * OVERTIME_FACTOR);
   } // else
       grossPay = regularPay + overtimePay;
   } // 3-parameter constructor

  /**
   *  Returns the hours worked by this full-time HourlyEmployee object.
   *
   *  @return the hours worked by this full-time HourlyEmployee object.
   *
   *
  public int getHoursWorked()
  {
      return hoursWorked;
  } // method getHoursWorked


  /**
   *  Returns the pay rate of this full-time HourlyEmployee object.
   *
   *  @return the pay rate this full-time HourlyEmployee object.
   *
   */
```

```
    public double getPayRate()
    {
         return payRate;
    } // method getPayRate


    /**
     *  Returns the regular pay of this full-time HourlyEmployee object.
     *
     *  @return the regular pay this full-time HourlyEmployee object.
     *
     */
    public double getRegularPay()
    {
        return regularPay;
    } // method getRegularPay


    /**
     *  Returns the overtime pay of this full-time HourlyEmployee object.
     *
     *  @return the overtime pay this full-time HourlyEmployee object.
     *
     */
    public double getOvertimePay()
    {
        return overtimePay;
    } // method getOvertimePay


    /**
     *  Returns a String representation of this full-time HourlyEmployee object with the
     *  name followed by a space followed by a dollar sign followed by the gross pay
     *  (with two fractional digits) followed by "FULL TIME HOURLY".
     *
     *  @return a String representation of this full-time HourlyEmployee object.
     *
     */
    public String toString()
    {
        final String FULL_TIME_STATUS = "HOURLY";

        return super.toString() + FULL_TIME_STATUS;
    } // method toString

} // class HourlyEmployee
```

A final note on the visibility modifier **`protected`**: It can be applied to methods as well as to fields. For example, the visibility modifier for the `getNextEmployee` method in the `Company` class should be changed from **`private`** to **`protected`** for the sake of potential subclasses of `Company`. One such subclass is introduced in Section 1.3.3.

Section 1.3.2 continues our discussion of inheritance by examining the interplay between inheritance and constructors.

1.3.2 Inheritance and Constructors

Constructors provide initialization for instances of a given class. For that reason, constructors are never inherited. But whenever a subclass constructor is called, the execution of the subclass constructor starts with an automatic call to the superclass's default constructor. This ensures that at least the default initialization of fields from the superclass will occur. For example, the `FullTimeEmployee` class's default constructor is automatically invoked at the beginning of a call to any `HourlyEmployee` constructor. That explains how the `name` and `grossPay` fields are initialized in the `HourlyEmployee` class's default constructor.

What if the superclass has a constructor but no default constructor? Then the first statement in any subclass constructor must explicitly call the superclass constructor. A call to a superclass constructor consists of the reserved word **`super`** followed by the argument list, in parentheses. For example, suppose some class `B`'s only constructor has an **`int`** parameter. If `C` is a subclass of `B` and `C` has a constructor with a `String` parameter, that constructor must start out by invoking `B`'s constructor. For example, we might have

```
public C (String s)
{
       super (s.length());  // explicitly calls B's int-parameter constructor
       ...

} // String-parameter constructor
```

So if a superclass explicitly defines a default (that is, zero-parameter) constructor, there are no restrictions on its subclasses. Similarly, if the superclass does not define any constructors, the compiler will automatically provide a default constructor, and there are no restrictions on the subclasses. But if a superclass defines at least one constructor and does not define a default constructor, the first statement in any subclass's constructor must explicitly invoke a superclass constructor.

1.3.3 The Subclass Substitution Rule

Just as the `Company` class used `FullTimeEmployee`, we can find the best-paid hourly employee with the help of an `HourlyCompany` class, which uses the `HourlyEmployee` class. `HourlyCompany`, a subclass of the `Company` class described in Section 1.2.3, differs only slightly from the `Company` class. Specifically, the `main` method invokes the `HourlyCompany` class's `run` method on a newly constructed `HourlyCompany` instance. Also, the `getNextEmployee` method is overridden to scan in the information for an `HourlyEmployee` object; to enable this overriding, we must change the visibility modifier of the `Company` class's `getNextEmployee` method from **`private`** to **`protected`**. Interestingly, the `run` and `findBestPaid` methods, which deal with full-time (not necessarily hourly) employees, are inherited, as is, from the `Company` class.

Assume the input file consists of

```
a 40 20
b 45 20
c 40 23
```

Then the output will be

The best-paid hourly employee (and gross pay) is b $950.00 FULL TIME HOURLY

Here is the HourlyCompany.java file:

```
import java.util.*;

import java.io.*;

public class HourlyCompany extends Company
{
    public static void main (String[ ] args) throws FileNotFoundException
    {
        new HourlyCompany().run();
    } // method main

   /**
    *  Returns the next hourly employee from the specified Scanner object.
    *
    *  @param sc – the Scanner object used to scan in the next employee.
    *
    *  @return the next hourly employee scanned in from sc.
    *
    */
   protected HourlyEmployee getNextEmployee (Scanner sc)
   {
      Scanner lineScanner = new Scanner (sc.nextLine());

      String name = lineScanner.next();

      int hoursWorked = lineScanner.nextInt();

      double payRate = lineScanner.nextDouble();

      return new HourlyEmployee (name, hoursWorked, payRate);
   } // method getNextEmployee

} // class HourlyCompany
```

Recall, from the inherited findBestPaid method, the following assignment:

```
full = getNextEmployee (sc);
```

The left-hand side of this assignment is (a reference to) a FullTimeEmployee object. But the value returned by the call to getNextEmployee is a reference to an HourlyEmployee object. Such an arrangement is legal because an HourlyEmployee is a FullTimeEmployee. This is an application of the Subclass Substitution Rule:

Subclass Substitution Rule

Whenever a reference-to-superclass-object is called for in an evaluated expression, a reference-to-subclass-object may be substituted.

Specifically, the left-hand side of the above assignment is a reference to a `FullTimeEmployee` object, so a reference to a `FullTimeEmployee` object is called for on the right-hand side of that assignment. So it is legal for that right-hand side expression to be a reference to an `HourlyEmployee` object; it is important to note that for an `HourlyEmployee` object, the `toString()` method includes "HOURLY". The returned reference is assigned to the `FullTimeEmployee` reference `full`, which is then used to update the `FullTimeEmployee` reference `bestPaid`. When the value of `bestPaid` is returned to the `run` method and the message `bestPaid.toString()` is sent, the output includes "HOURLY". Why? The reason is worth highlighting:

When a message is sent, the version of the method invoked depends on the *run-time type of the object referenced*, not on the compile-time type of the reference.

Starting with the construction of the new `HourlyEmployee` object in the `getNextEmployee` method, all of the subsequent references were to an `HourlyEmployee` object. So the version of the `toString()` method invoked by the message `bestPaid.toString()` was the one in the `HourlyEmployee` class.

Let's take a closer look at the Subclass Substitution Rule. Consider the following:

```
FullTimeEmployee full = new FullTimeEmployee ();

HourlyEmployee hourly = new HourlyEmployee();

full = hourly;
```

In this last assignment statement, a reference-to-`FullTimeEmployee` is called for in the evaluation of the expression on the right-hand side, so a reference-to-`HourlyEmployee` may be substituted: an `HourlyEmployee` is a `FullTimeEmployee`.

But the reverse assignment is illegal:

```
FullTimeEmployee full = new FullTimeEmployee ();

HourlyEmployee hourly = new HourlyEmployee ();

hourly = full; // illegal
```

On the right-hand side of this last assignment statement, the compiler expects a reference-to-`HourlyEmployee`, so a reference-to-`FullTimeEmployee` is unacceptable: a `FullTimeEmployee` is not necessarily an `HourlyEmployee`. Note that the left-hand side of an assignment statement must consist of a variable, which is an expression. But that left-hand-side variable is not evaluated in the execution of the assignment statement, so the Subclass Substitution Rule does not apply to the left-hand side.

Now suppose we had the following:

```
FullTimeEmployee full = new FullTimeEmployee ();

HourlyEmployee hourly = new HourlyEmployee ();

full = hourly;
hourly = full; // still illegal
```

After the assignment of `hourly` to `full`, `full` contains a reference to an `HourlyEmployee` object. But the assignment:

```
hourly = full;
```

still generates a compile-time error because the declared type of `full` is still reference-to-`FullTime Employee`. We can avoid a compile-time error in this situation with a ***cast***: the temporary conversion of an expression's type to another type. The syntax for a cast is:

```
(the new type)expression
```

Specifically, we will cast the type of `full` to `HourlyEmployee`:

```
FullTimeEmployee full = new FullTimeEmployee ();

HourlyEmployee hourly = new HourlyEmployee ();

full = hourly;
hourly = (HourlyEmployee) full;
```

To put it anthropomorphically, we are saying to the compiler, "Look, I know that the type of `full` is reference-to-`FullTimeEmployee`. But I promise that at run-time, the object referenced will, in fact, be an `HourlyEmployee` object." The cast is enough to satisfy the compiler, because the right-hand side of the last assignment statement now has type reference-to-`HourlyEmployee`. And there is no problem at run-time either because—from the previous assignment of `hourly` to `full`—the value on the right-hand side really is a reference-to-`HourlyEmployee`.

But the following, acceptable to the compiler, throws a `ClassCastException` at run-time:

```
FullTimeEmployee full = new FullTimeEmployee ();

HourlyEmployee hourly = new HourlyEmployee ();

hourly = (HourlyEmployee) full;
```

The run-time problem is that `full` is actually pointing to a `FullTimeEmployee` object, not to an `HourlyEmployee` object.

The complete project, `HourlyCompany`, is in the ch1 directory of the book's website. Lab 1's experiment illustrates another subclass of `FullTimEmployee`.

You are now prepared to do Lab 1: The **`SalariedEmployee`** Class

Before we can give a final illustration of the Subclass Substitution Rule, we need to introduce the `Object` class. The `Object` class, declared in the file `java.lang.Object.java`, is the superclass of all classes. `Object` is a bare-bones class, whose methods are normally overridden by its subclasses. For example, here are the method specification and definition of the `equals` method in the `Object` class:

```
/**
 *  Determines if the calling object is the same as a specified object.
 *
 *  @param obj - the specified object to be compared to the calling object.
 *
 *  @return true - if the two objects are the same.
```

```
     *
     */
    public boolean equals (Object obj)
    {
      return (this == obj);
    } // method equals
```

The definition of this method compares *references*, not objects, for equality. So true will be returned if and only if the calling object reference contains the same *address* as the reference `obj`. For example, consider the following program fragment:

```
Object obj1 = new Object(),
       obj2 = new Object(),
       obj3 = obj1;

System.out.println (obj1.equals (obj2) + "" + obj1.equals (obj3));
```

The output will be

```
false true
```

For that reason, this method is usually overridden by the `Object` class's subclasses. We saw an example of this with the `String` class's `equals` method in Section 0.2.3 of Chapter 0. In that `equals` method, the parameter's type is `Object`, and so, by the Subclass Substitution Rule, the argument's type can be `String`, a subclass of `Object`. For example, we can have

```
if (message.equals ("nevermore"))
```

1.3.4 Is-a versus Has-a

You will often encounter the following situation. You are developing a class `B`, and you realize that the methods of some other class, `A`, will be helpful. One possibility is for `B` to inherit all of `A`; that is, `B` will be a subclass of `A`. Then all of `A`'s **`protected`** methods are available to `B` (all of `A`'s **`public`** methods are available to `B` whether or not `B` inherits from `A`). An alternative is to define, in class `B`, a field whose class is `A`. Then the methods of `A` can be invoked by that field. It is important to grasp the distinction between these two ways to access the class `A`.

Inheritance describes an **is-a** relationship. An object in the subclass `HourlyEmployee` is also an object in the superclass `FullTimeEmployee`, so we can say that an `HourlyEmployee` is-a `FullTime Employee`.

On the other hand, the fields in a class constitute a **has-a** relationship to the class. For example, the `name` field in the `FullTimeEmployee` class is of type (reference to) `String`, so we can say a `FullTimeEmployee` has-a `String`.

Typically, if class `B` shares the overall functionality of class `A`, then inheritance of `A` by `B` is preferable. More often, there is only one aspect of `B` that will benefit from `A` 's methods, and then the better alternative will be to define an `A` object as a field in class `B`. That object can invoke the relevant methods from class `A`. The choice may not be clear-cut, so experience is your best guide. We will encounter this problem several times in subsequent chapters.

With an object-oriented approach, the emphasis is not so much on developing the program as a whole but on developing modular program-parts, namely, classes. These classes not only make the program easier to understand and to maintain, but they are reusable for other programs as well. A further advantage to this approach is that decisions about a class can easily be modified. We first decide what classes will be needed.

And because each class interacts with other classes through its method specifications, we can change the class's fields and method definitions as desired as long as the method specifications remain intact.

The next section of this chapter considers the extent to which a language can allow developers of a class to force users of that class to obey the Principle of Data Abstraction.

1.4 Information Hiding

The Principle of Data Abstraction states that a user's code should not access the implementation details of the class used. By following that principle, the user's code is protected from changes to those implementation details, such as a change in fields.

Protection is further enhanced if a user's code is *prohibited* from accessing the implementation details of the class used. ***Information hiding*** means making the implementation details of a class inaccessible to code that uses that class. The burden of obeying the Principle of Data Abstraction falls on users, whereas information hiding is a language feature that allows class developers to prevent users from violating the Principle of Data Abstraction.

As you saw in Section 1.3.1, Java supports information hiding through the use of the **`protected`** visibility modifier for fields. Through visibility modifiers such as **`private`** and **`protected`**, Java forces users to access class members only to the extent permitted by the developers. The term ***encapsulation*** refers to the grouping of fields and methods into a single entity—the class—whose implementation details are hidden from users.

There are three essential features of object-oriented languages: the *encapsulation* of fields and methods into a single entity with information-hiding capabilities, the *inheritance* of a class's fields and methods by subclasses, and *polymorphism*, discussed in Section 1.5.

1.5 Polymorphism

One of the major aids to code re-use in object-oriented languages is polymorphism. ***Polymorphism***—from the Greek words for "many" and "shapes"—is the ability of a reference to refer to different objects in a class hierarchy. For a simple example of this surprisingly useful concept, suppose that `sc` is a reference to an already constructed `Scanner` object. We can write the following:

```
FullTimeEmployee employee; // employee is of type reference-to-FullTimeEmployee

if (sc.nextLine().equals ("full time"))
    employee = new FullTimeEmployee ("Doremus", 485.00);
else
    employee = new HourlyEmployee ("Kokoska", 45, 20);
System.out.println (employee.toString());
```

Because the declared type of `employee` is reference-to-`FullTimeEmployee`, it is legal to write

```
employee = new FullTimeEmployee ("Doremus", 485.00);
```

So, by the Subclass Substitution Rule, it is also legal to write

```
employee = new HourlyEmployee ("Kokoska", 45, 20);
```

Now consider the meaning of the message

```
employee.toString()
```

The version of the `toString()` method executed depends on the type of the object that `employee` is referencing. If the scanned line consists of "full time", then `employee` is assigned a reference to an instance of class `FullTimeEmployee`, so the `FullTimeEmployee` class's version of `toString()` is invoked. On the other hand, if the scanned line consists of any other string, then `employee` is assigned a reference to an instance of class `HourlyEmployee`, so the `HourlyEmployee` class's version of `toString()` is invoked.

In this example, `employee` is a polymorphic reference: the object referred to can be an instance of class `FullTimeEmployee` or an instance of class `HourlyEmployee`, and the meaning of the message `employee.toString()` reflects the point made in Section 1.3.3: When a message is sent, the version of the method invoked depends on the type of the object, not on the type of the reference. What is important here is that polymorphism allows code re-use for methods related by inheritance. We need not explicitly call the two versions of the `toString()` method.

The previous code raises a question: how can the Java compiler determine which version of `toString()` is being invoked? Another way to phrase the same question is this: How can the method identifier `toString` be *bound* to the correct definition—in `FullTimeEmployee` or in `HourlyEmployee`—at compile time, when the necessary information is not available until run time? The answer is simple: The binding cannot be done at compile-time, but must be delayed until run time. A method that is bound to its method identifier at run time is called a ***virtual method***.

In Java, almost all methods are virtual. The only exceptions are for **`static`** methods (discussed in Section 2.1) and for **`final`** methods (the **`final`** modifier signifies that the method cannot be overridden in subclasses.) This delayed binding—also called ***dynamic binding*** or ***late binding***—of method identifiers to methods is one of the reasons that Java programs execute more slowly than programs in most other languages.

Polymorphism is a key feature of the Java language, and makes the Java Collections Framework possible. We will have more to say about this in Chapter 4, when we take a tour of the Java Collections Framework.

Method specifications are method-level documentation tools. Section 1.6 deals with class-level documentation tools.

1.6 The Unified Modeling Language

For each project, we will illustrate the classes and relationships between classes with the ***Unified Modeling Language*** (UML). UML is an industry-standardized language, mostly graphical, that incorporates current software-engineering practices that deal with the modeling of systems. The key visual tool in UML is the class diagram. For each class—except for widely used classes such as `String` and `Random`—the class diagram consists of a rectangle that contains information about the class. The information includes the name of the class, its attributes and operations. For the sake of simplicity, we will regard the UML term ***attribute*** as a synonym for field. Similarly, the UML term ***operation*** will be treated as a synonym for method. For example, Figure 1.1 shows the class diagram for the `FullTimeEmployee` class from Section 1.2.2. For both attributes and operation parameters, the type follows the variable (instead of preceding the variable, as in Java).

In a class diagram, a method's parenthesized parameter-list is followed by the return type, provided the method actually does return a value. Visibility information is abbreviated:

+, for public visibility

−, for private visibility

#, for protected visibility

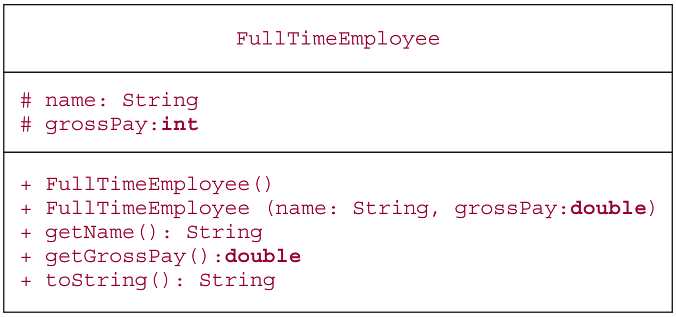


FIGURE 1.1 A UML class-diagram for the `FullTimeEmployee` class

Inheritance is illustrated by a *solid* arrow from the subclass to the superclass. For example, Figure 1.2 shows the relationship between the `HourlyEmployee` and `FullTimeEmployee` classes in Section 1.3.

A *dashed* arrow illustrates the relationship between a class and the interface that class implements. For example, Figure 1.3 augments the class diagrams from Figure 1.2 by adding the diagram for the `Employee` interface.

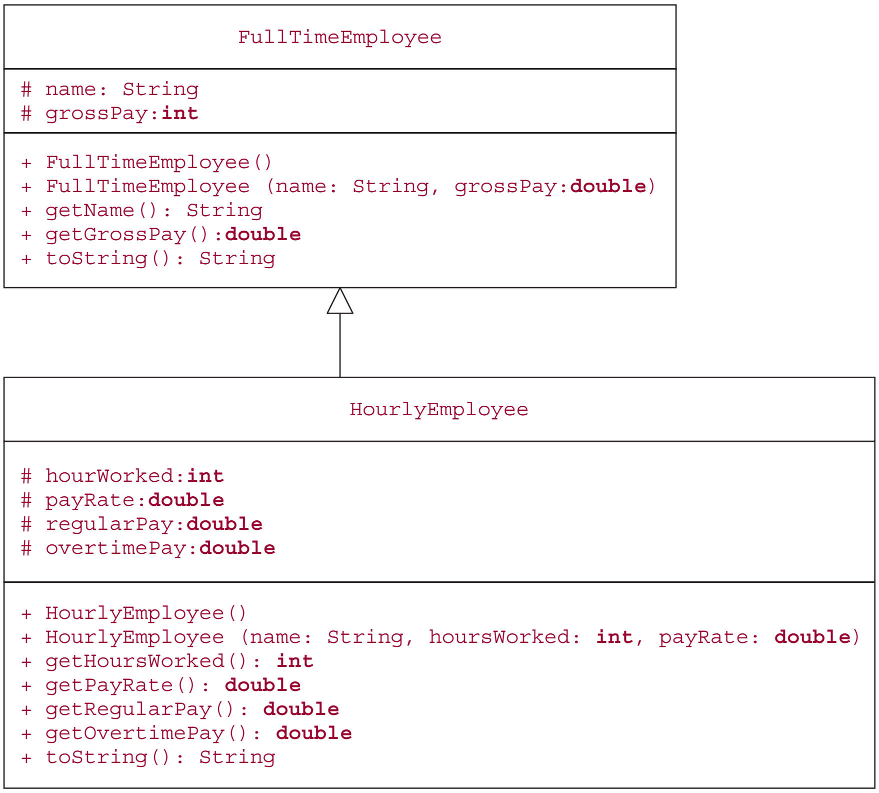


FIGURE 1.2 In UML, the notation for inheritance is a solid arrow from the subclass to the superclass

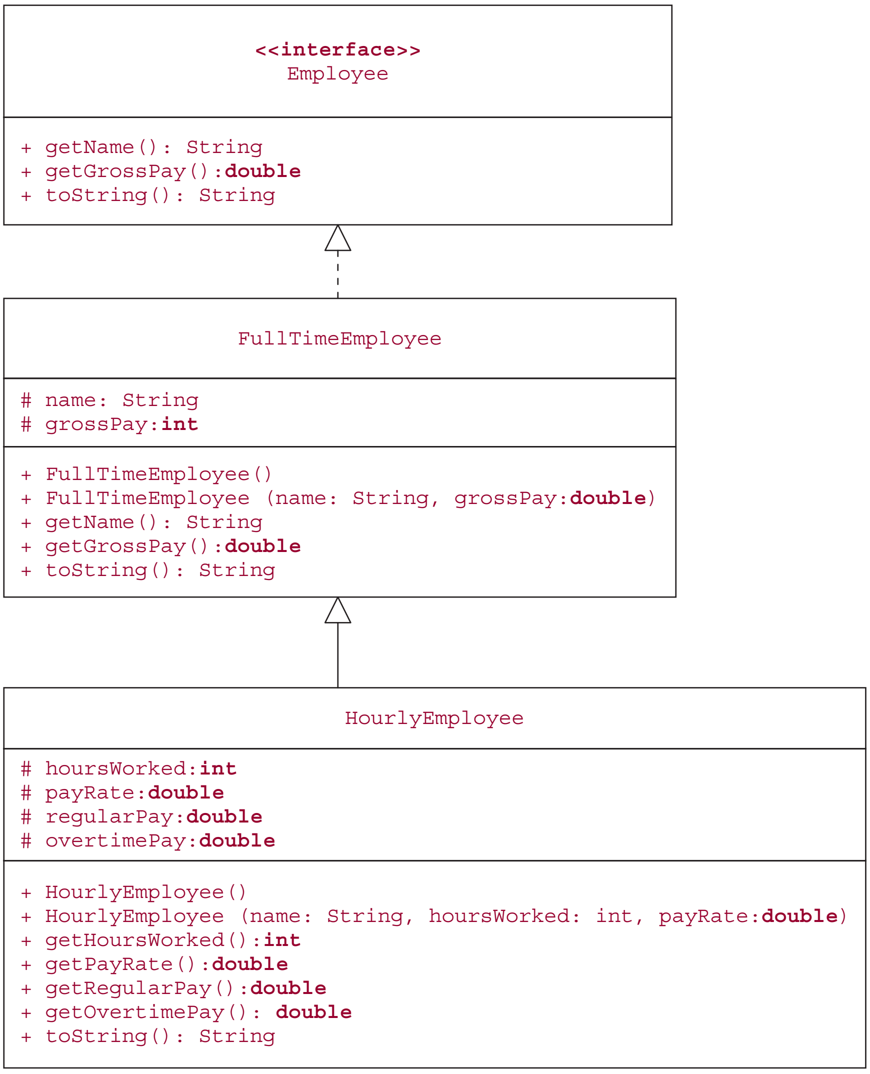


FIGURE 1.3 A UML illustration of the relationship between an interface, an implementing class, and a subclass

A non-inheritance relationship between classes is called an ***association***, and is represented by a solid line between the class diagrams. For example, Figure 1.4 shows an association between the `Company` and `FullTimeEmployee` classes in the find-best-paid-employee project in Section 1.2.3.

In Figure 1.4, the symbol '*' at the bottom of the association line indicates a company can have an arbitrary number of employees. The number 1 at the top of the association line indicates that an employee works for just one company.

Sometimes, we want to explicitly note that the class association is a has-a relationship, that is, an instance of one class is a field in the other class. In UML, a has-a relationship is termed an ***aggregation***, and is signified by a solid line between the classes, with a hollow diamond at the containing-class end.

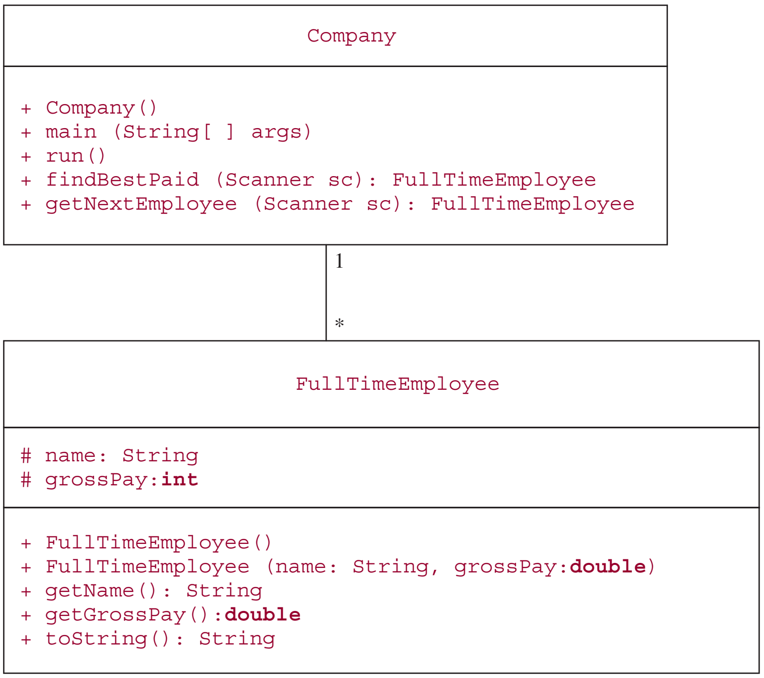


FIGURE 1.4 The UML representation of an association between two classes

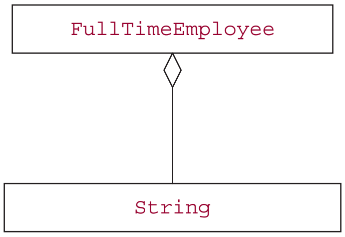


FIGURE 1.5 Aggregation in UML: the `FullTimeEmployee` class has a `String` field

For example, Figure 1.5 shows that the `FullTimeEmployee` class has a `String` field. To avoid clutter, the figure simply has the class name in each class diagram.

Graphical tools such as UML play an important role in outlining a project. We will be developing projects, starting in Chapter 5, as applications of data structures and algorithms. Each such project will include UML class diagrams.

SUMMARY

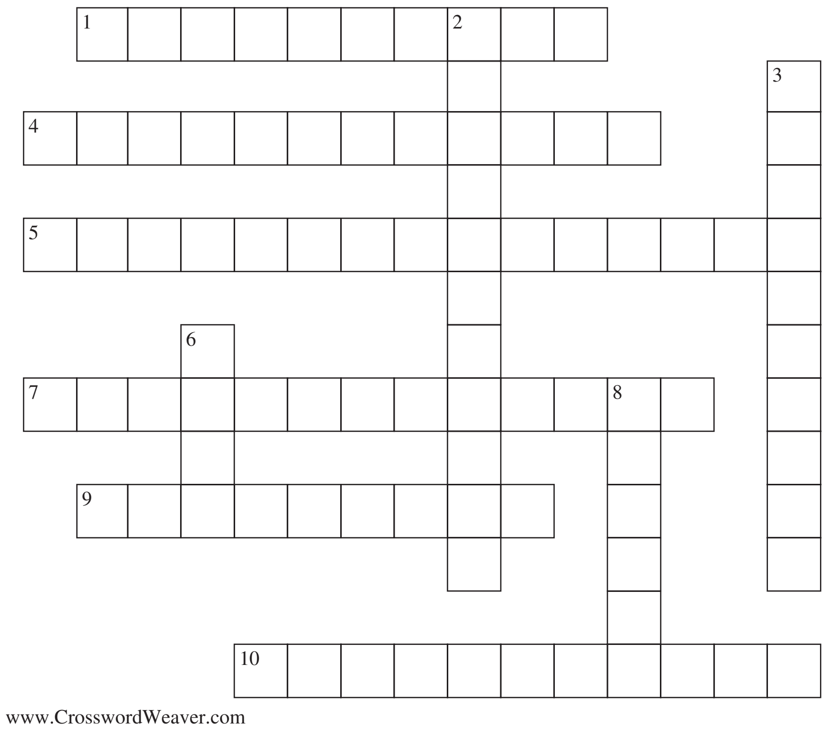
This chapter presents an overview of object-oriented programming. Our focus, on the use of classes rather than on their implementation details, is an example of data abstraction. ***Data abstraction***—the separation of method specifications from field and method definitions—is a way for users of a class to protect their code from being affected by changes in the implementation details of the class used.

The three essential features of an object-oriented language are:

1. *Encapsulation* of fields and methods into a single entity—the class—whose implementation details are hidden from users.
2. *Inheritance* of a class's fields and methods by subclasses.
3. *Polymorphism*: the ability of a reference to refer to different objects.

The Unified Modeling Language (UML) is an industry-standard, graphical language that illustrates the modeling of projects.

CROSSWORD PUZZLE



ACROSS

1. The ability of a subclass to give new definitions—applicable in the subclass—to methods defined in the superclass.

4. The ability of a reference to refer to different objects in a class hierarchy.

5. The separation of what a class provides to users from how the fields and methods are defined.

7. The grouping of fields and methods into a single entity—the class—whose implementation details are hidden from users.

9. A collection of abstract methods and constants; the object-oriented term for "abstract data type."

10. In the Unified Modeling Language, a non-inheritance relationship between classes.

DOWN

2. An example of an is–a relationship.

3. The principle that every class should be extendible through inheritance and still stable for existing applications.

6. The temporary conversion of an expression's type to another type.

8. The superclass of all classes.

CONCEPT EXERCISES

1.1 Given that HourlyEmployee and SalariedEmployee are subclasses of FullTimeEmployee, suppose we have:

```
FullTimeEmployee full = new FullTimeEmployee();
HourlyEmployee hourly = new HourlyEmployee ();
SalariedEmployee salaried = new SalariedEmployee ();

full = salaried;
```

Which one of the following assignments would be legal both at compile-time and at run-time?

a. `salaried = (SalariedEmployee) full;`

b. `salaried = full;`

c. `salaried = (FullTimeEmployee) full;`

d. `hourly = (HourlyEmployee) full;`

Create a small project to validate your claim.

1.2 Assume that the classes below are all in the file Polymorphism.java. Determine the output when the project is run. Would the output be different if the call to println were System.out.println (a.toString())?

```
import java.util.*;

public class Polymorphism
{
         public static void main (String args [ ])
         {
             new Polymorphism().run();
         } // method main

         public void run()
         {
                 Scanner sc = new Scanner (System.in));

                 A a;

                 int code = sc.nextInt();

                 if (code == 0)
                           a = new A();
                 else       // non-zero int entered
                           a = new D();
                 System.out.println (a);
         } // method run

} // class Polymorphism
```

```
class A
{
         public String toString ()
         {
                 return "A";
         } // method toString

} // class A


class D extends A
{
         public String toString ()
         {
                    return "D";
         } // method toString

} // class D
```

1.3 In the `Employee` class, modify the `toString` method so that the gross pay is printed with a comma to the left of the hundreds digit. For example, if the name is "O'Brien,Theresa" and the gross pay is 74400.00, the `toString` method will return

```
O'Brien,Theresa $74,400.00
```

1.4 What can you infer about the identifier `out` from the following message?

```
System.out.println ("Eureka!");
```

What is the complete declaration for the identifier `out`? Look in java.lang.System.java.

PROGRAMMING EXERCISES

1.1 Here is a simple class—but with method specifications instead of method definitions—to find the highest age from the ages scanned in:

```
public class Age
{
       protected int highestAge;

       /**
        *  Initializes this Age object.
        *
        */
       public Age ()

       /**
        *  Returns the highest age of the ages scanned in from the keyboard.
```

```
             *  The sentinel is -1.
             *
             * @param sc - The Scanner used to scan in the ages.
             *
             * @return the highest age of the ages scanned in from sc.
             *
             */
            public int findHighestAge (Scanner sc)

    } // class Age
```

a. Fill in the method definitions for the Age class.

b. Test your Age class by developing a project and running the project.

1.2 With the Age class in Programming Exercise 1.1.a. as a guide, develop a Salary class to scan in salaries from the input until the sentinel (−1.00) is reached, and to print out the average of those salaries. The average salary is the total of the salaries divided by the number of salaries.

1.3 This exercise presents an alternative to having **protected** fields. Modify the FullTimeEmployee class as follows: Change the visibility of the name and grossPay fields from **protected** to **private**, and develop **public** methods to get and set the values of those fields. A method that alters a field in an object is called a ***mutator***, and a method that returns a *copy* of a field is called an ***accessor***.

Here are the method specifications corresponding to the name field:

```
/**
 *  Returns this FullTimeEmployee object's name.
 *
 *  @return a (reference to a) copy of this FullTimeEmployee object's
 *                 name.
 *
 */
public String getName ()
/**
 *  Sets this FullTimeEmployee object's name to a specifed string.
 *
 *  @param nameIn - the String object whose value is assigned to this
 *                                 FullTimeEmployee object's name.
 *
 */
public void setName (String nameIn)
```

1.4 Create a class to determine which hourly employee in a file received the most overtime pay. The name of the file is to be scanned in from System.in.

1.5 In the toString() method of the FullTimeEmployee class, there is a call to the format method. The heading of that method is

```
public final String format(double number)
```

What is the definition of that method?

Programming Project 1.1

A `CalendarDate` Class

In this project, you will develop and test a `CalendarDate` class. Here are the responsibilities of the class, that is, the services that the class will provide to users:

1. to initialize a `CalendarDate` object to represent the date January 1, 2012;
2. to initialize a `CalendarDate` object from integers for the month, day-of-month and year; if the date is invalid (for example, if the month, day-of-month and year are 6, 31 and 2006, respectively), use 1, 1, 2012;
3. return, in `String` form, the next date after this `CalendarDate` object; for example, if this `CalendarDate` object represents January 31, 2012, the return value would be "February 1, 2012";
4. return, in `String` form, the date prior to this `CalendarDate` object; for example, if this `CalendarDate` object represents January 1, 2013, the return value would be "December 31, 2012";
5. return, in `String` form, the day of the week on which this `CalendarDate` object falls; for example, if this `CalendarDate` object represents the date December 20, 2012, the return value would be "Thursday";

Part a: Create method specifications for the above responsibilities.

Part b: Develop the `CalendarDate` class, that is, determine what fields to declare and then define the methods.

Part c: Create a project to test your `CalendarDate` class. Call each `CalendarDate` method at least twice.

Additional Features of Programming and Java

CHAPTER 2

In Chapter 1, the primary goal was to introduce object-oriented concepts, such as interfaces, inheritance and polymorphism, in the context of the Java programming language. This chapter introduces more topics on programming in general and Java in particular, and illustrates how they can aid your programming efforts. For example, Java's exception-handling mechanism provides programmers with significant control over what happens when errors occur.

CHAPTER OBJECTIVES

1. Distinguish between static members and instance members.
2. Be able to develop JUnit tests for a class's methods.
3. Be able to create `try` blocks and `catch` blocks to handle exceptions.
4. Compare file input/output with console input/output.
5. Understand the fundamentals of the Java Virtual Machine.
6. Be able to override the `Object` class's `equals` method.
7. Understand the interplay between packages and visibility modifiers.

2.1 Static Variables, Constants and Methods

Recall, from Section 1.2.2, that a class member is either a field or method in the class[1]. Let's look at some of the different kinds of members in Java. There are two kinds of fields. An ***instance variable*** is a field associated with an object—that is, with an instance of a class. For example, in the `FullTimeEmployee` class from Chapter 1, `name` and `grossPay` are instance variables. Each `FullTimeEmployee` object will have its own pair of instance variables. Suppose we declare

```
FullTimeEmployee oldEmployee,
                 currentEmployee,
                 newEmployee;
```

Then the object referenced by `oldEmployee` will have its own copy of the instance variables `name` and `grossPay`, and the objects referenced by `currentEmployee` and `newEmployee` will have their own copies also.

[1]In Section 4.2.3.1, we will see that a class may also have another class as a member.

In addition to instance variables, which are associated with a particular object in a class, we can declare ***static variables***, which are associated with the class itself. The space for a static variable—also known as a ***class variable***—is shared by all instances of the class. A field is designated as a static variable by the reserved modifier **`static`**. For example, if a `count` field is to maintain information about all objects of a class `Student`, we could declare the field `count` to be a static variable in the `Student` class:

```
protected static int count = 0;
```

This static variable could be incremented, for example, whenever a `Student` constructor is invoked. Then the variable `count` will contain the total number of `Student` instances created.

A class may also have constant identifiers, also called “symbolic constants” or “named constants”. A ***constant identifier*** is an identifier that represents a constant, which is a variable that can be assigned to only once. The declaration of a constant identifier includes the reserved word **`final`**—indicating only one assignment is allowed—as well as the type and value of the constant. For example, we can write

```
protected final static int SPEED_LIMIT = 65.0;
```

Constant identifiers promote both readability (SPEED_LIMIT conveys more information than 65.0) and maintainability (because SPEED_LIMIT is declared in only one place, it is easy to change its value throughout the class). There should be just one copy of the constant identifier for the entire class, rather than one copy for each instance of the class. So a constant identifier for a class should be declared as **`static`**; constants within a method cannot be declared as **`static`**. At the developer’s discretion, constant identifiers for a class may have **`public`** visibility. Here are declarations for two constant class identifiers:

```
public final static char COMMAND_START = ‘$’;

public final static String INSERT_COMMAND = "$Insert";
```

To access a static member *inside* its class, the member identifier alone is sufficient. For example, the above **`static`** field `count` could be accessed in a method in the `Student` class as follows:

```
count++;
```

In order to access a static member *outside* of its class, the class identifier itself is used as the qualifier. For example, outside of the wrapper class `Integer`, we can write:

```
if  (size == Integer.MAX_VALUE)
```

Here is the declaration for an often-used constant identifier in the `System` class:

```
public final static PrintStream out = nullPrintStream();
```

Because `out` is declared as **`static`**, its calls to the `PrintStream` class’s `println` method include the identifier `System` rather than an instance of the `System` class. For example,

```
System.out.println ("The Young Anarchists Club will hold a special election next week" +
                    "to approve the new constitution.");
```

The **`static`** modifier is used for any constant identifier that is defined outside of a class’s methods. The **`static`** modifier is *not* available within a method. For example, in the definition of the default constructor in the `FullTimeEmployee` class, we had:

```
final String EMPTY_STRING = "";
```

It would have been illegal to use the **`static`** modifier for this constant.

In Chapter 1, the `Employee` interface declared a constant:

```
final static DecimalFormat MONEY = new DecimalFormat (" $0.00");
                // a class constant used in formatting a value in dollars and cents
```

The constant identifier `MONEY` can be used in any class that implements the `Employee` interface. Recall that any constant or method identifier in an interface automatically has **`public`** visibility.

Java also allows static methods. For example, the `Math` class in the package java.lang has a `floor` method that takes a **`double`** argument and returns, as a **`double`**, the largest value that is less than or equal to the argument and equal to an integer. We can write

```
System.out.println (Math.floor (3.7));   // output: 3.0
```

When `floor` is called, there is no calling object because the effect of the method depends only on the **`double`** argument. To signify this situation, the class identifier is used in place of a calling object when `floor` is called. A method that is called without a calling object is known as a **`static`** method, as seen in the following heading

```
public static double floor (double a)
```

The execution of every Java application (excluding applets, servlets, and so on) starts with a static `main` method. And **`static`** methods are not virtual; that is, **`static`** methods are bound to method identifiers at compile time, rather than at run time. The reason is that **`static`** methods are associated with a class itself rather than an object, so the issue of which object is invoking the method does not arise.

2.2 Method Testing

A method is ***correct*** if it satisfies its method specification. The most widely used technique for increasing confidence in the correctness of a method is to test the method with a number of sample values for the parameters. We then compare the actual results with the results expected according to the method's specification.

The purpose of testing is to discover errors, which are then removed. When—eventually—no errors are discovered during testing, that does not imply that the method is correct, because there may be other tests that would reveal errors. In general, it is rarely feasible to run all possible tests, and so we cannot infer correctness based on testing alone. As E. W. Dijkstra has noted:

Testing can reveal the presence of errors but not the absence of errors.

The testing software we utilize is JUnit: the "J" stands for "Java," and each method in a project is referred to as a "unit." JUnit is an Open Source (that is, free) unit-testing product—available from www.junit.org—that allows the methods in a class to be tested systematically and without human intervention—for example, without keyboard input or Graphical User Interface (GUI) mouse clicks. The web page http://junit.sourceforge.net/README.html#Installation has information on installation. In general, the success or failure of an individual test is determined by whether the expected result of the test matches the actual result. The output from testing provides details for each failed test. For a simple example, here is a test of the `toString()` method in the `FullTimeEmployee` class:

```
@Test
public void toStringTest1()
{
     FullTimeEmployee full = new FullTimeEmployee ("a", 150.00);
```

```
        String expected = "a $150.00 FULL TIME";

        assertEquals (expected, full.toString());
    } // method testToString1
```

"@Test " is referred to as an annotation. The assertEquals method determines whether its arguments are equal. In this case, if the two strings are equal, the test is passed. Otherwise, the test fails.

The assertEquals method is an overloaded method in the Assert class of the org.junit package. The heading for the version called above is

```
public static void assertEquals (java.lang.String expected, java.lang.String actual)
```

There are other versions of the method to compare primitive values. For example, we could have

```
    int median = roster.findMedian();
    assertEquals (82, median);
```

Also, there is a version to compare any two objects. Here is the method heading:

```
public static void assertEquals (java.lang.Object expected, java.lang.Object actual)
```

According to the Subclass Substitution Rule, the arguments can be instances of any class—because any class is a subclass of Object. In fact, expected is a polymorphic reference when the AssertEquals method is invoked: the code executed depends on the types of the objects involved. For example, with String objects, this version has the same effect as the version in which both parameters have type String. (But this version is slightly slower than the String-parameter version due to the run-time test to make sure that actual is an instance of the String class.)

Finally, there are several assertArrayEquals methods for comparing two arrays of **int** values, two arrays of **double** values, two arrays of Object references, and so on.

The details of running JUnit test classes will depend on your computing environment. Typically, you will run the tests in an Integrated Development Environment (IDE) such as Eclipse or DrJava, and the output of testing may combine text and graphics (for example, a green bar if all tests were successful, and a red bar if one or more tests failed). For the sake of generality, the following test class is not tied to any IDE, but simply prints the string returned by the getFailures() method in the runClasses class of the package org.junit. If you are running tests in an IDE, the main method from the class below will be ignored. Here is a complete class for testing the toString( ) method in the FullTimeEmployee class—followed by a discussion of the details:

```
    import org.junit.*;
    import static org.junit.Assert.*;
    import org.junit.runner.Result;
    import static org.junit.runner.JUnitCore.runClasses;

    public class FullTimeEmployeeTest
    {
        public static void main(String[ ] args)
        {
            Result result = runClasses (ToStringTest.class);
            System.out.println ("Tests run = " + result.getRunCount() +
                                "\nTests failed = " + result.getFailures());
        } // method main

        protected FullTimeEmployee full;
        protected String expected;

        @Test
        public void toStringTest1()
        {
            full = new FullTimeEmployee ("a", 150.00);
            expected = "a $150.00 FULL TIME";
            assertEquals (expected, full.toString());
        } // method toStringTest1

        @Test
        public void toStringTest2()
        {
            full = new FullTimeEmployee ("b", 345.678);
            expected = "b $345.678 FULL TIME";     // error!
            assertEquals (expected, full.toString());
        } // method toStringTest2

        @Test
        public void toStringTest3()
        {
            full = new FullTimeEmployee();
            expected = " $0.00 FULL TIME";
            assertEquals (expected, full.toString());
        } // method toStringTest3

    } // class FullTimeEmployeeTest
```

The line

```
import static org.junit.Assert.*;
```

allows static methods—such as `assertEquals`—in the `Assert` class to be accessed without specifying the class name. The `main` method runs this cluster of tests, one after the other, in no particular order. Because of the mistake in test 2, the output from the program is

```
Tests run = 3
Tests failed = [toStringTest2(FullTimeEmployeeTest): expected:<b $345.6[7]8 FULL TIME>
but was:<b $345.6[]8 FULL TIME>]
```

Note that the mistake—the extra "7" in the expected value—was in the test, not in the method being tested, and was written just so you would know what is output when a test fails.

When should these tests be developed and run? Most unit-testing enthusiasts recommend that

A method's tests should be developed before the method is defined.

The advantage to pre-definition testing is that the testing will be based on the method specification only, and will not be influenced by the method definition. Furthermore, the tests should be run both before and after the method is defined (and after any subsequent changes to the method definition). That will illustrate the transition from a method that fails the tests to a method that passes the tests. But how can a method be compiled before it is defined? To satisfy the Java compiler, each method definition can be a ***stub***: a definition that has only enough code to avoid a compile-time error. For example, here is a stub for the `toString( )` method in the `FullTimeEmployee` class:

```
public String toString()
{
      return null;
} // method toString
```

When `FullTimeEmployeeTest` was run with this stub for the `toString( )` method, all three tests failed. (Of course, the mistake in `test2` ensures that test will fail anyway.) Because this chapter introduces unit testing intermingled with several important language features, the method testing in this chapter will be presented *after* the method has been fully defined. In subsequent chapters we will adhere to the test-first paradigm.

In Section 2.3.2 we will see how to create a stub that will fail any test even if the return type of the method to be tested is **`boolean`**.

2.2.1 More Details on Unit Testing

In Section 2.2, we developed a test suite for the `toString( )` method in the `FullTimeEmployee` class. There was so little that could go wrong with the `toString( )` method that we could barely justify the testing. (In fact, in the applications in subsequent chapters, a class's `toString( )` method will often be untested, but used in testing other methods.) The real purpose of the example was to show how a test suite could be developed. What about the other methods in the `FullTimeEmployee` class? Should they be tested also? Probably not. The constructors cannot be tested in isolation; in fact, you could argue that the suite in `FullTimeEmployeeTest` tests the constructors as much as testing the `toString( )` method. Also, there is no point in testing the accessor methods `getName( )` and `getGrossPay ( )`: they simply return the values assigned in a constructor.

So at this point, we can be fairly confident in the correctness of the methods in the `FullTime Employee` class. For the `Company` class from Chapter 1, which methods are suitable for testing? There is no point in testing the `main` method: it simply invokes the `run( )` method. The `run( )` method cannot be tested without human intervention because the end user must enter the input-file path from the keyboard. The **`protected`** method `getNextEmployee (Scanner sc)` can be tested—the `CompanyTest` class will be a subclass of the `Company` class. Finally, the `findBestPaid (Scanner sc)` method can and should be tested. In fact, that method was, originally, designed to facilitate testing: The reading of file name and printing of the best-paid employee were moved up to the `run( )` method. This illustrates an important aspect of method design:

In general, methods should be designed to facilitate testing.

Here is a `CompanyTest` class to test both the `getNextEmployee` and `findBestPaid` methods. Note that the `@Before` annotation precedes any method that is automatically invoked just prior to each test.

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;
import java.io.*;  // for IOException, see Section 2.3

public class CompanyTest extends Company
{
   public static void main(String[ ] args)
   {
        Result result = runClasses (CompanyTest.class);
        System.out.println ("Tests run = " + result.getRunCount() +
                            "\nTests failed = " + result.getFailures());
   } // method main

   protected Company company;

   protected FullTimeEmployee best;

   protected Scanner sc;

   protected String expected;

   @Before
   public void runBeforeEveryTest()
   {
        company = new Company();
   } // method runBeforeEveryTest

   @Test
   public void getNextEmployeeTest1()
   {
        sc = new Scanner ("Lucas 350.00");
        expected = "Lucas $350.00 FULL TIME";
        assertEquals (expected, company.getNextEmployee (sc).toString());
   } // method getNextEmployeeTest1

   @Test
   public void findBestPaidTest1() throws IOException
   {
        sc = new Scanner (new File ("company.in1"));
        best = company.findBestPaid (sc);
        expected = "b $150.00 FULL TIME";
        assertEquals (expected, best.toString());
   } // method findBestPaidTest1

  @Test
  public void findBestPaidTest2() throws IOException
  {
        sc = new Scanner (new File ("company.in2"));
        best = company.findBestPaid (sc);
        assertEquals (null, best);
  } // method findBestPaidTest2
} // class CompanyTest
```

The file `company.in1` contains

```
a 100
b 149.995
c 140
```

The file `company.in2` is empty. When the above tests were run with the versions of `getNextEmployee (Scanner sc)` and `findBestPaid (Scanner sc)` from Chapter 1, all three test cases were successful.

For the `HourlyEmployee` class, the only method worthy of testing is the three-parameter constructor. As noted earlier in this section, constructors cannot be tested in isolation. Instead, we will test the `getRegularPay`, `getOvertimePay`, and `getGrossPay` methods. These accessor methods are worthy of testing since they do more than simply returning values passed to a constructor. The important aspect of the following `HourlyEmployeeTest` class is the testing of a ***boundary condition***: the comparison (>=, >, <=, <) of a variable to some fixed value; the result of the comparison determines the action taken. Specifically, the `hoursWorked` is compared to 40 to determine the regular pay, overtime pay, and gross pay. To make sure that all boundary cases are covered, there are separate tests for `hoursWorked` equal to 39, 40, and 41. In comparing the expected result with the actual result for `regularPay`, `overTimePay`, and `grossPay`, we should not compare **`double`** values for exact equality. So we utilize a three-parameter `assertEquals` method, with the third parameter the (very small) difference we will allow between the expected and actual **`double`** values.

Make sure that any boundary conditions are thoroughly tested.

Here is the `HourlyEmployeeTest` class:

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;

public class HourlyEmployeeTest
{
    public static void main(String[ ] args)
    {
        Result result = runClasses (HourlyEmployeeTest.class);
        System.out.println ("Tests run = " + result.getRunCount() +
                            "\nTests failed = " + result.getFailures());
    } // method main

    public static final double DELTA = 0.0000001;

    protected HourlyEmployee hourly;

   @Test
   public void test1()
   {
        hourly = new HourlyEmployee ("andrew", 39, 10.00);
        assertEquals (390.00, hourly.getRegularPay(), DELTA);
        assertEquals (0.00, hourly.getOvertimePay(), DELTA);
        assertEquals (390.00, hourly.getGrossPay(), DELTA);
    } // method test1

    @Test
    public void test2()
    {
        hourly = new HourlyEmployee ("beth", 40, 20.00);
        assertEquals (800.00, hourly.getRegularPay(), DELTA);
        assertEquals (0.00, hourly.getOvertimePay(), DELTA);
        assertEquals (800.00, hourly.getGrossPay(), DELTA);
    } // method test2

    @Test
    public void test3()
    {
        hourly = new HourlyEmployee ("terry", 41, 20.00);
        assertEquals (800.00, hourly.getRegularPay(), DELTA);
        assertEquals (30.00, hourly.getOvertimePay(), DELTA);
        assertEquals (830.00, hourly.getGrossPay(), DELTA);
    } // method test3

    @Test
    public void test4()
    {
        hourly = new HourlyEmployee ("karen", 50, 10);
        assertEquals (400.00, hourly.getRegularPay(), DELTA);
        assertEquals (150.00, hourly.getOvertimePay(), DELTA);
        assertEquals (550.00, hourly.getGrossPay(), DELTA);
    } // method test4

} // class HourlyEmployeeTest
```

What about testing other methods? There is no rule to determine which methods in a class should be tested. The best strategy is to assume that a method contains subtle flaws that can be revealed only by rigorous testing.

Good testing requires great skepticism.

This can be a challenge to programmers, who tend to view their work favorably, even glowingly ("a thing of beauty and a joy forever"). As such, programmers are ill-suited to test their own methods because the purpose of testing is to uncover errors. Ideally the person who constructs test data should hope that the method will *fail* the test. If a method fails a test and the method is subsequently revised, *all* tests of that method should be re-run.

In the next section, we introduce Java's exception-handling facilities, and consider the interplay between exception-handling and testing.

2.3 Exception Handling

An ***exception*** is an object created by an unusual condition, typically, an attempt at invalid processing. When an exception object is constructed, the normal flow of control is halted; the exception is said to be ***thrown***. Control is immediately transferred to code—either in the current method or in some other method—that "handles" the exception. The exception handling usually depends on the particular exception, and may involve printing an error message, terminating the program, taking other action, or maybe doing nothing.

A ***robust*** program is one that does not terminate unexpectedly from invalid user-input. We almost always prefer programs that—instead of "crashing"—allow recovery from an error such as the input of 7.o instead of 7.0 for a **`double`**. Java's exception-handling feature allows the programmer to avoid almost all abnormal terminations.

For a simple introduction to exception handling, let's start with a method that takes as a parameter a (non-null reference to a) `String` object. The `String` represents a person's full name, which should be in the form "first-name middle-name last-name". The method returns the name in the form "last-name, first-name middle-initial.". For example, if we have

```
rearrange ("John Quincy Adams"))
```

The `String` returned will be

```
Adams, John Q.
```

Here is the method specification and a preliminary definition:

```
/**
 *  Returns a specified full name in the form "last-name, first-name middle-initial.".
 *
 *  @param fullName - a (non-null reference to a) String object that represents the
 *                    specified full name, which should be in the form
 *                    "first-name middle-name last-name".
 *
 *  @return the name in the form "last-name, first-name middle-initial.".
 *
 */
public String rearrange (String fullName)
{
       Scanner sc = new Scanner (fullName);

       String firstName = sc.next(),
              middleName = sc.next(),
              lastName = sc.next();

       return lastName + ", " + firstName + " " + middleName.charAt (0) + ".";
} // method rearrange
```

The problem with this method, as currently defined, is that the execution of the method can terminate abnormally. How? If the argument corresponding to `fullName` is a (reference to a) `String` object that does not have at least three components separated by whitespace, a `NoSuchElementException` object will be thrown. In this case, the execution of the method will terminate abnormally. Instead of an abnormal termination, we want to allow execution to continue even if the argument corresponding to `fullName` is not a reference to a `String` that consists of those three components. That is, we "try" to split up `fullName`, and "catch" the given exception. The revised specification and definition are

```
/**
 *  Returns a specified full name in the form "last-name, first-name middle-initial.".
 *
 *  @param fullName - a (non-null reference to a) String object that represents the
 *                    specified full name, which should be in the form
 *                    "first-name middle-name last-name".
 *
 *  @return the name in the form "last-name, first-name middle-initial." if fullName
 *          has three components.  Otherwise, return
 *          "java.util.NoSuchElementException: the name is not of the form
 *          "first-name middle-name last-name"".
 *
 */
public String rearrange (String fullName)
{
     String result;

     try
      {
        Scanner sc = new Scanner (fullName);

        String firstName = sc.next(),
               middleName = sc.next(),
               lastName = sc.next();

        result = lastName + ", " + firstName + " " + middleName.charAt (0) + ".";
      } // try
      catch (NoSuchElementException e)
      {
        result = e.toString() + ": " + ": The name is not of the form \"first-name " +
                 "middle-name last-name\"";
      } // catch
      return result;
} // method rearrange
```

In the execution of this method, the flow of control is as follows. Inside the **try** block, if `fullName` can be split into first, middle, and last names, the three calls to `sc.next( )` and the subsequent assignment to `result` will be executed. The entire **catch** block will be skipped over, and the **return** statement will be executed. But if `fullName` cannot be split into first, middle, and last names, one of the calls to `sc.next( )` will throw a `NoSuchElementException` object, the **try** block will be exited, and the statement inside the **catch** block will executed. Then the **return** statement will be executed, as before.

In the **catch** block, the parameter `e` is (a reference to) the `NoSuchElementException` object created and thrown during the execution of one of the calls to `sc.next( )`. Specifically, `e.toString( )` is the string `"java.util.NoSuchElementException"`. We will see shortly, in Section 2.3.1, how an exception can be thrown in one method and caught in another method.

Here is a test class for the `rearrange` method (assume that method is in the `NameChange` class, which may consist of nothing except the `rearrange` method):

```
     import org.junit.*;
     import static org.junit.Assert.*;
     import org.junit.runner.Result;
```

```
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;

public class NameChangeTest
{
   public static void main(String[ ] args)
   {
      Result result = runClasses (NameChangeTest.class);
      System.out.println ("Tests run = " + result.getRunCount() +
                          "\nTests failed = " + result.getFailures());
   } // method main
   public final static String EXCEPTION = "java.util.NoSuchElementException";

   public final static int EXCEPTION_LENGTH = EXCEPTION.length();

   protected NameChange change;

   protected String result;

   @Before
   public void runBeforeEveryTest( )
   {
      change = new NameChange();
   } // method runBeforeEveryTest

   @Test
   public void rearrangeTest1()
   {
      result = change.rearrange ("John Quincy Adams");
      assertEquals ("Adams, John Q.", result);
   } // method rearrangeTest1

   @Test
   public void rearrangeTest2()
   {
      result = change.rearrange ("John Adams");
      assertEquals (EXCEPTION, result.substring (0, EXCEPTION_LENGTH));
   } // method rearrangeTest2

   @Test
   public void rearrangeTest3()
   {
      result = change.rearrange ("John");
      assertEquals (EXCEPTION, result.substring (0, EXCEPTION_LENGTH));
   } // method rearrangeTest3

   @Test
   public void rearrangeTest4()
   {
      result = change.rearrange ("");
      assertEquals (EXCEPTION, result.substring (0, EXCEPTION_LENGTH));
   } // rearrangeTest4

} // class NameChangeTest
```

In this example, the exception was handled—in the **catch** block—of the `rearrange` method. In the next section, we see how to handle exceptions that are not caught within the method in which they are thrown.

2.3.1 Propagating Exceptions

What happens if an exception, such as `NoSuchElementException`, is thrown in a method that does not catch that exception? Then control is transferred back to the calling method: the method that called the method that threw the exception. This transferring of control is known as ***propagating the exception***. For example, the following method determines whether or not an integer scanned in is a leap year[2] (one of the exceptions is explicitly thrown, with a **throw** statement):

```
/**
 *  Determines if the integer scanned in is a leap year.
 *
 *  @param sc - a (reference to) a Scanner object from which
 *             the year is scanned in.
 *
 *  @return true - if the integer is a leap year; otherwise, returns false.
 *
 *  @throws InputMismatchException - if the string scanned in from sc is not
 *                                 empty but does not consist of an integer.
 *  @throws NoSuchElementException - if the value scanned in from sc is an
 *          empty string.
 *
 *  @throws NullPointerException - if sc is null.
 *  @throws IllegalArgumentException - if the value scanned in from
 *          sc is an integer less than 1582.
 */
public boolean isLeapYear (Scanner sc)
{
     final int FIRST_YEAR = 1582; // start of Gregorian Calendar

     int year = sc.nextInt();

     if (year < FIRST_YEAR)
         throw new IllegalArgumentException();
     if ((year % 4 == 0) && (year % 100 != 0 || year % 400 == 0))
         return true;
     return false;
} // method isLeapYear
```

What can go wrong in a call to this method? One possible error, as indicated in the `@throws` sections of the javadoc specification, if the string scanned in from `sc` is not empty but does not consist of an integer, `InputMismatchException` will be thrown. This exception is not caught in the `isLeapYear` method, so the exception is propagated back to the method that called `isLeapYear`. For example, the following `LeapYear` class has a `run()` method that scans five lines from `System.in`, and determines which lines contain leap years and which lines contain non-integers.

[2]Because the earth makes one full rotation around the sun in slightly less than 365.25 days, not every year divisible by 4 is a leap year. Specifically, a leap year must be both divisible by 4 and either not divisible by 100 or divisible by 400. So 2000 was a leap year, but 2100 will not be a leap year.

```
import java.util.*;  // for the Scanner class

public class LeapYear
{

   public static void main (String args [ ])
   {
       new LeapYear().run();
   } // method main


   public void run()
   {
       final String INPUT_PROMPT = "Please enter the year: ";

       Scanner sc = new Scanner (System.in);

       for (int i = 0; i < 5; i++)
           try
           {
               System.out.print (INPUT_PROMPT);
               System.out.println (isLeapYear (sc));
           } // try
           catch (InputMismatchException e)
           {
               System.out.println ("The input is not an integer.");
               sc.nextLine();
           } // catch InputMismatchException
   } // method run


   public boolean isLeapYear (Scanner sc)
   {
       final int FIRST_YEAR = 1582; // start of Gregorian Calendar

       int year = sc.nextInt();

       if (year < FIRST_YEAR)
              throw new IllegalArgumentException();

       if ((year % 4 == 0) && (year % 100 != 0 || year % 400 == 0))
           return true;
       return false;
   } // method isLeapYear

} // class LeapYear
```

For input of

```
2000
2100
201o
```

```
     2010
     2008
```

the output will be:

```
true
false
The input is not an integer.
false
true
```

The above **catch** block includes a call to `sc.nextLine()`. If that call had been omitted, the output for the above input would be

```
true
false
The input is not an integer.
The input is not an integer.
The input is not an integer.
```

Why? When the third call to `sc.nextInt()` in `isLeapYear` throws `InputMismatchException` for "`201o`", the scanner remains positioned on the third line instead of advancing to the fourth line. Then the next two calls to `sc.nextInt( )` also throw `InputMismatchException` for "`201o`". We needed to include `sc.nextLine( )` in the **catch** block to ensure that the scanner skips over the illegal input.

It is worth noting that in a method's specification, only propagated exceptions are included in the `@throws` javadoc comments. Any exception that is caught within the method definition itself (such as we did in the `rearrange` method of Section 2.3) is an implementation detail, and therefore not something that a user of the method needs to know about.

Incidentally, without too much trouble we can modify the above `run` method to accommodate an arbitrary number of input values. To indicate the end of the input, we need a value—called a ***sentinel***—that is not a legal year. For example, we can use `"***"` as the sentinel. When that value is entered from the keyboard, `InputMismatchException` is thrown in the `isLeapYear` method and caught in the `run` method, at which point a **break** statement terminates the execution of the scanning loop. Here is the revised `run` method:

```
     public void run()
     {
       final String SENTINEL = "***";

       final String INPUT_PROMPT =
          "Please enter the year (or " + SENTINEL + " to quit): ";

       Scanner sc = new Scanner (System.in);
       while (true)
       {
          try
          {
              System.out.print (INPUT_PROMPT);
              System.out.println (" " + isLeapYear (sc) + "\n");
          } // try
          catch (InputMismatchException e)
```

```
            {
                if (sc.nextLine().equals (SENTINEL))
                    break;
                System.out.println (" The input is not an integer.\n");
            } // catch
         } // while
      } // method run
```

If a propagated exception is not caught in a method, the exception is propagated back to the calling method. If the calling method does not handle the exception, then the exception is propagated back to the method that called the calling method itself. Ultimately, if the exception has not been caught even in the `main` method, the program will terminate abnormally and a message describing the exception will be printed. The advantage to propagating an exception is that the exception can be handled at a higher level in the program. Decisions to change how exceptions are handled can be made in one place, rather than scattered throughout the program. Also, the higher level might have facilities not available at lower levels, such as a Graphical User Interface (GUI) window for output.

2.3.2 Unit Testing and Propagated Exceptions

How can we test a method that propagates an exception? Right after the `@Test` annotation, we specify the expected exception. For example, a test of the `isLeapYear` method might have

```
@Test (expected = InputMismatchException.class)
public void isLeapYearTest()
{
      leap.isLeapYear (new Scanner ("201o"));
} // isLeapYearTest
```

For a complete test suite of the `isLeapYear` method, we cannot scan over `System.in` because such tests would require human intervention. Another option is to scan over a string of lines that contain the values to be tested. But in JUnit, test methods can be invoked in any order, and the results of one test do not affect any other test. So to ensure that the calls to the scanner would start on successive lines, we would have to place all tests in one method. This would be legal but inappropriate because the tests are independent of each other.

The following test suite for the `isLeapYear` method has each test in a separate method, and includes tests for `InputMismatchException`, `NoSuchElementException`, `NullPointerException` and `IllegalArgumentException`.

Here is the test class:

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;

public class LeapYearTest
{
   public static void main(String[ ] args)
   {
      Result result = runClasses (LeapYearTest.class);
      System.out.println ("Tests run = " + result.getRunCount() +
                  "\nTests failed = " + result.getFailures());
   } // method main
```

```
protected LeapYear leap;

protected boolean answer;

@Before
public void runBeforeEveryTest()
{
   leap = new LeapYear();
} // method runBeforeEveryTest

@Test
public void leapYearTest1()
{
   answer = leap.isLeapYear (new Scanner ("2000"));
   assertEquals (true, answer);
} // method leapYearTest1

@Test
public void leapYearTest2()
{
   answer = leap.isLeapYear (new Scanner ("2100"));
   assertEquals (false, answer);
} // method leapYearTest2

@Test
public void leapYearTest3()
{
   answer = leap.isLeapYear (new Scanner ("1582"));
   assertEquals (false, answer);
} // method leapYearTest3

@Test (expected = InputMismatchException.class)
public void leapYearTest4()
{
   leap.isLeapYear (new Scanner ("201o"));
} // method leapYearTest4

@Test (expected = NoSuchElementException.class)
public void leapYearTest5()
{
   leap.isLeapYear (new Scanner (""));
} // method leapYearTest5

@Test (expected = NullPointerException.class)
public void leapYearTest6()
{
     leap.isLeapYear (null);
} // method leapYearTest6

@Test (expected = IllegalArgumentException.class)
public void leapYearTest7()
{
     leap.isLeapYear (new Scanner ("1581"));
```

```
        } // method leapYearTest7

    } // class LeapYearTest
```

What if the exception propagated in the method being tested is not the exception expected in the testing method? Then the testing method will generate an error message that the exception thrown was not the one expected. Finally, what if the method being tested propagates an exception but no exception was expected by the testing method? For example, at the start of `leapYearTest5`, suppose we replaced

```
    @Test (expected = NoSuchElementException.class)
```

with

```
    @Test
```

Then the test would generate the following error message:

```
    Tests failed = [leapYearTest5(LeapYearTest): null]
```

The keyword **`null`** signifies that an exception was thrown but no exception was expected. In JUnit, an ***error*** in running a test method occurs if an unexpected exception is thrown or if an expected exception is not thrown. So it is an error if an exception is thrown but a different exception is expected. Errors are included in the string returned by the `getFailures()`method in the `runClasses` class, but the term ***failure*** is often applied only to those situations in which an assertion is tested and fails. Because testing assertions is what unit testing is all about, errors must be removed before serious testing can begin.

We defined the above `isLeapYear` method before we introduced the exception-propagation feature needed to test that method. What if, as is normally the case, we wanted to test a method before the method is defined? Specifically, how can we create a stub that will generated an error message for all of the above tests? If the stub returns **`true`**, `leapYearTest1()` will succeed, and if the stub returns **`false`**, `leapYearTest2()` will succeed. Clearly, the stub cannot return either **`true`** or **`false`**. Instead, the stub will throw an exception other than the exceptions thrown according to the specifications. For example,

```
    public boolean isLeapYear (Scanner sc)
    {
         throw new UnsupportedOperationException();
    } // method isLeapYear
```

When the test suite `LeapYearTest` was run on this stub, every test generated an error message (that is good news), and the output (formatted for readability) was

```
Tests run = 7
Tests failed =
        leapYearTest1(LeapYearTest): null
        leapYearTest2(LeapYearTest): null,
        leapYearTest3(LeapYearTest): null,
        leapYearTest4(LeapYearTest): Unexpected exception,
              expected<java.util.InputMismatchException> but
              was<java.lang.UnsupportedOperationException>,
        leapYearTest5(LeapYearTest): Unexpected exception,
              expected<java.util.NoSuchElementException> but
              was<java.lang.UnsupportedOperationException>,
        leapYearTest6(LeapYearTest): Unexpected exception,
              expected<java.lang.NullPointerException> but
              was<java.lang.UnsupportedOperationException>,
        leapYearTest7(LeapYearTest): Unexpected exception,
              expected<java.lang.IllegalArgumentException> but
              was<java.lang.UnsupportedOperationException>]
```

2.3.3 Checked Exceptions

Exceptions related to input or output, such as when a file is not found or the end-of-file marker is encountered while input is being read, are the most common examples of checked exceptions. With a ***checked exception***, the compiler checks that either the exception is caught within the method itself or—to allow propagation of the exception—that a **throws** clause is appended to the method heading. For an example of the latter, we might have

```
public void sample() throws IOException
{
```

This indicates that the `sample` method might throw an `IOException` object. If so, the exception will be propagated back to the method that called `sample`. That calling method *must* either catch `IOException` or append the same **throws** clause to its method heading. And so on. Checked exceptions are propagated for the same reason that other exceptions are propagated: It might be preferable to handle all exceptions at a higher level for the sake of uniformity, or there might be better facilities (such as a GUI window) available at the higher level.

For an example of how a checked exception can be handled in a method, we can revise the `run( )` method from Section 2.3.1 to scan lines from a file and determine which lines consist of leap years. The name of the file will be read from the keyboard in a loop that continues until the name corresponds to an existing file. Here is the revised `run( )` method:

```
public void run()
{
    final String INPUT_PROMPT = "Please enter the file name: ";

    Scanner keyboardScanner = new Scanner (System.in);

    String fileName;

    while (true)
    {
       System.out.print (INPUT_PROMPT);
       fileName = keyboardScanner.next();
       try
       {
           Scanner sc = new Scanner (new File (fileName));
           while (sc.hasNext())
              try
              {
                  System.out.println (isLeapYear (sc));
              } // try to scan a year
              catch (InputMismatchException e)
              {
                  System.out.println ("The input is not an integer.");
                  sc.nextLine();
              } // catch input mismatch
           break;
```

```
            } // try to scan the name of an existing file
            catch (FileNotFoundException e)
            {
                System.out.println (e);
            } // catch file not found
        } // while true
    } // method run
```

The **break** statement—to exit the outer loop—is executed when the file name scanned from the keyboard represents an existing file, and that file has been scanned for leap years. The inner-loop condition—`sc.hasNext()`—is slightly preferable to `sc.hasNextLine()`. In particular, if the last line in the file is blank, `sc.hasNext()` will return false and the execution of the inner loop will terminate, as desired. But if the last line in the file is blank, `sc.hasNextLine()` will return true, and the subsequent call to `sc.nextInt()` in the `isLeapYear` method will throw `NoSuchElementException`. Of course, if that exception is caught in the `run()` method, then `sc.hasNextLine()` will not be a problem.

A checked exception must be caught or must be specified in a **throws** clause, and the compiler "checks" to make sure this has been done. Which exceptions are checked, and which are unchecked? The answer is simple: run-time exceptions are not checked, and all other exceptions are checked. Figure 2.1 shows Java's exception hierarchy, including `IOException` with its subclasses (such as `FileNotFound Exception`), and `RuntimeException` with its subclasses (such as `NullPointerException`).

Why are run-time exceptions not checked? The motivation behind this is that an exception such as `NullPointerException` or `NumberFormatException`, can occur in almost any method. So appending a **throws** clause to the heading of such a method would burden the developer of the method without providing any helpful information to the reader of that method.

When an exception is thrown, the parameter classes of the subsequent **catch** blocks are tested, in order, until (unless) one is found for which the thrown exception is an instance of that class. So if you

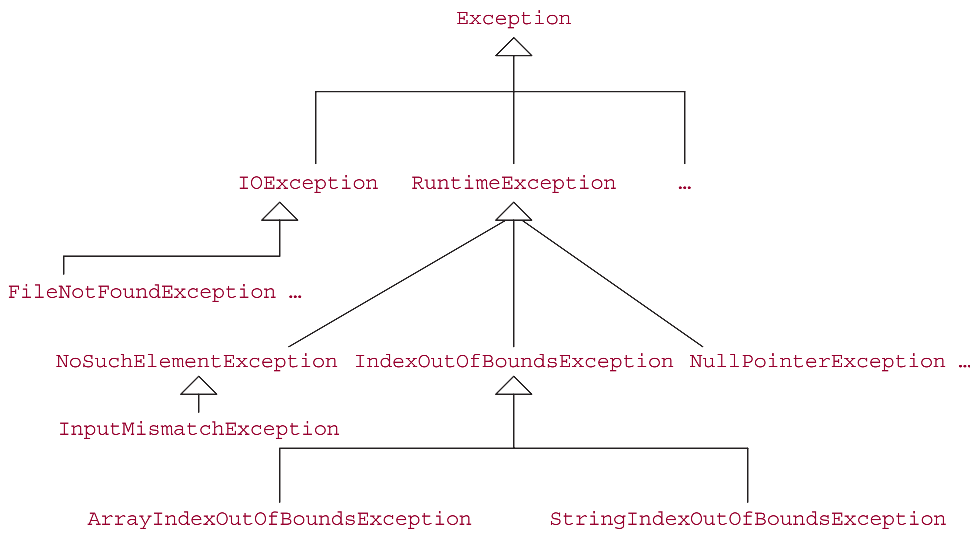


FIGURE 2.1 The exception hierarchy. In the unified modeling language, inheritance is represented with an arrow from a subclass to its superclass

want to ensure that all run-time exceptions are caught in a method, you can insert the following as the last **catch** block:

```
catch (RuntimeException e)
{
     // code to handle the exception
} // catch RuntimeException
```

If you do have a **catch** block for RuntimeException, make sure that **catch** block is not followed by a **catch** block for a subclass of RuntimeException. For example, because NullPointerException is a subclass of RuntimeException, the following sequence of **catch** blocks will generate a compile-time error:

```
catch (RuntimeException e)
{
     // code to handle the exception
} // catch RuntimeException
catch (NullPointerException e)     // error!
{
     // code to handle the exception
} // catch NullPointerException
```

The error message will inform you that the second **catch** block is unreachable code.

An exception can be explicitly thrown by the programmer, who gets to decide which exception class will be instantiated and under what circumstances the exception will be thrown. For example, suppose we want a method to return the smaller of two **double** values that represent prices obtained by comparison shopping. If the prices are too far apart—say, if the difference is greater than the smaller price—we throw an exception instead of returning the smaller price. The mechanism for explicitly throwing the exception is the **throw** statement, which can be placed anywhere a statement is allowed. For example, the code may be as in the following smaller method (the Math class's **static** method abs returns the absolute value of its argument):

```
public class Compare
{
     public static void main (String[ ] args)
     {
          new Compare().run();
     } // method main

     public void run()
     {
          System.out.println (smaller (5.00, 4.00));
          System.out.println (smaller (5.00, 20.00));
     } // method run

     public double smaller (double price1, double price2)
     {
          if (Math.abs (price1 - price2) > Math.min (price1, price2))
                throw new ArithmeticException ("difference too large");
          return Math.min (price1, price2);
     } // method smaller

} // class Compare
```

If the given comparison is true, the **throw** statement is executed, which creates a new instance of the exception class `ArithmeticException` by calling the constructor that takes a `String` argument. The exception will be propagated back to the method that called `smaller` and the execution of the `smaller` method will immediately terminate. In the above example, the exception is not caught, so the program terminates. The output is

```
4.0
java.lang.ArithmeticException: difference too large
```

The choice of `ArithmeticException` as the exception class to be instantiated is somewhat arbitrary.

A user can even create new exception classes. For example,

```
public class UnreasonablenessException extends RuntimeException
{
      public UnreasonablenessException (String s)
      {
            super (s);
      } // constructor with String parameter
} // class UnreasonablenessException
```

We can rewrite the `smaller` method to throw this exception:

```
public double smaller (double price1, double price2)
{
    if (Math.abs (price1 - price2) > Math.min (price1, price2))
          throw new UnreasonablenessException ("difference too large");
    return Math.min (price1, price2);
} // method smaller
```

This creates a new instance of the class `UnreasonablenessException`. The above program would terminate with the message:

```
UnreasonablenessException: difference too large
```

The `UnreasonablenessException` class is a subclass of `RuntimeException`. The `Runtime Exception` class handles[3] some of the low-level details of exception-handling, such as keeping track of the method the exception occurred in, the method that called that method, and so on. Such a "call stack" sequence can help a programmer to determine the root cause of an error.

An alternative to explicitly throwing an exception is to take a default action that overcomes the mistake. For example, here is a version of the 2-parameter constructor in the `FullTimeEmployee` class that replaces a negative value for gross pay with 0.00:

```
public FullTimeEmployee (String name, double grossPay)
{
     this.name = name;
     this.grossPay = Math.max (grossPay, 0.00);
} // 2-parameter constructor
```

[3]Actually, `RuntimeException` consists of several constructors, each of which merely invokes the corresponding constructor in `Exception`, the superclass of `RuntimeException`. The `Exception` class passes the buck to its superclass, `Throwable`, where the low-level details are dealt with.

2.3.4 The **finally** Block

Under normal circumstances, any code you place after the last **catch** block will be executed whether or not any exceptions were thrown in the **try** block. So you can place clean-up code—such as closing files—after the last **catch** block. There are two drawbacks to this approach. First, there may be an exception thrown in the **try** block that is not caught in a **catch** block. Another danger is that one of the **catch** blocks may itself throw an exception that is not caught. Then the clean-up code will not be executed. To avoid these pitfalls, Java allows a **finally** block after the last **catch** block. We can write

```
try
{
          ... // code that may throw an exception
} // try
catch (NumberFormatException e)
{
          ... // code to handle NumberFormatException
} // catch NumberFormatException
catch (IOException e)
{
          ... // code to handle IOException
} // catch IOException
finally
{
          ... // clean-up code; will be executed even if there are uncaught
              // exceptions thrown in the try block or catch blocks.
} // finally
```

If your **try** or **catch** blocks may throw uncaught exceptions, you should include a **finally** block—otherwise, any code placed after the last **catch** block may not be executed. Finally, a **finally** block is required by the Java language if you have a **try** block without a **catch** block.

Lab 2 provides the opportunity for you to practice exception-handling.

You are now prepared to do Lab 2: Exception Handling

The handling of input-output exceptions is one of the essential features of file processing, discussed in Section 2.4.

2.4 File Output

File output is only slightly different from console output. We first associate a `PrintWriter` reference with a file name. For example, to associate `printWriter` with `"scores.out"`:

```
PrintWriter printWriter = new PrintWriter  (new BufferedWriter
                         (new FileWriter ("scores.out")));
```

The `PrintWriter` object that is referenced by `printWriter` can now invoke the `print` and `println` methods. For example,

```
printWriter.println (line);
```

The output is not immediately stored in the file `"scores.out"`. Instead, the output is stored in a ***buffer***: a temporary storage area in memory. After all calls to `print` and `println` have been made by `printWriter`'s object, that object's `close` method must be called:

```
printWriter.close();
```

The `close` method flushes the buffer to the file `"scores.out"` and closes that file.

The file-processing program we will develop in this section is based on a program from Section 0.2.5 of Chapter 0. That program calculates the sum of scores read in from the keyboard. Here is a slightly modified version of that program, with a separate method to scan in and add up the scores:

```
import java.util.*; // for the Scanner class

public class Scores1
{
   public final int SENTINEL = -1;

   public static void main (String[ ] args)
   {
      new Scores1().run();
   } // method main

   public void run()
   {
      final String INPUT_PROMPT = "\nOn each line, enter a test score (or " +
                                   SENTINEL + " to quit): ";

      final String RESULT = "\n\nThe sum of the scores is ";

      Scanner sc = new Scanner (System.in);

      System.out.print (INPUT_PROMPT);

      int sum = addScores (sc);

      System.out.println (RESULT + sum);
   } // method run

   /**
    *  Returns the sum of the scores scanned in.
    *
    *  @param sc – a (non-null reference to a) Scanner object from
    *               which the scores are scanned in.
    *
    *  @return the sum of the scores scanned in from sc.
    *
    *  @throws InputMismatchException – if a value scanned in from sc is not an
    *                                    integer.
    *
    */
   public int addScores (Scanner sc)
   {
        int score,
            sum = 0;

        while (true)
        {
           score = sc.nextInt();
           if (score == SENTINEL)
                break;
           sum += score;
        } // while
        return sum;
      } // method addScores

   } // class Scores1
```

In the next version, the output goes to a file. To enable someone reading that file to confirm that the result is correct for the given input, each score is written to the output file. `IOException` is caught for output-file creation. The corresponding **`try`** block encompasses the creation of the output file and the input loop. For the sake of simplicity, there is no **`try`** block to catch input-mismatch exceptions (arising from input values that are not integers).

```
import java.util.*;

import java.io.*;

public class Scores2
{
    public final int SENTINEL = -1;

    public static void main (String [ ] args)
    {
        new Scores2().run();
    } // method main


    public void run()
    {
        final String INPUT_PROMPT =
            "\nOn each line, enter a test score (or " + SENTINEL + " to quit): ";

        final String RESULT = "\n\nThe sum of the scores is ";

        PrintWriter printWriter = null; // to ensure that printWriter is initialized
                                        // before it is closed in the finally block

        try
        {
            Scanner sc = new Scanner (System.in);
            printWriter = new PrintWriter (new BufferedWriter
                                  (new FileWriter ("scores.out")));

            System.out.print (INPUT_PROMPT);
            addScores (sc, printWriter);
        } // try
        catch (IOException e)
        {
            System.out.println (e);
        } // catch IOException
        finally
        {
            printWriter.println (RESULT + sum);
            printWriter.close();
        } // finally
    } // method run

    public int addScores (Scanner sc, PrintWriter printWriter)
    {

        int score,
            sum = 0;

        while (true)
        {
            score = sc.nextInt();
            if (score == SENTINEL)
               break;
            printWriter.println (score);
            sum += score;
        } // while
        return sum;
    } // method addScores

} // class Scores2
```

The simplification of ignoring input-mismatch exceptions leads to an unfortunate consequence: If an input-mismatch exception is thrown, the program will terminate without printing the final sum. The output file will be closed before the final sum is printed, and the `InputMismatchException` message—signifying abnormal termination—will be printed. We could add a **`catch`** block for `InputMismatchException` right after (or right before) the **`catch`** block for `IOException`. This change would not be much of an improvement: The program would still terminate without printing the final sum, but the termination would be normal.

To enable the program to continue after an input-mismatch exception, we create a new **`try`** block and a corresponding **`catch`** block inside the **`while`** loop. If the input contains no legal scores, we throw an exception related to that after the **`while`** loop. Here is the revised code:

```
boolean atLeastOneScore = false;
while (true)
{
      try
      {
           score = sc.nextInt();
           if (score == SENTINEL)
                   break;
           printWriter.println (score);
           sum += score;
           atLeastOneScore = true;
      } // try
```

```
            catch (InputMismatchException e)
            {
                 printWriter.println (e + " " + sc.nextLine());
            } // catch InputMismatchException
       } // while
       if (!atLeastOneScore)
           throw new RuntimeException ("The input contains no legal scores. ");
```

Here is a sample run of the resulting program, with input in boldface:

```
     Please enter a test score, or -1 to quit: 50
     Please enter a test score, or -1  to quit: x
     Please enter a test score, or -1  to quit: 80
     Please enter a test score, or -1  to quit: y
     Please enter a test score, or -1  to quit: -1
     The execution of this project has ended.
```

The file `scores.out` will now contain the following:

```
50
java.lang.InputMismatchException: x
80
java.lang.InputMismatchException: y

The sum of the scores is 130
```

The call to `nextLine( )` in the **`catch`** block of the `addScores` method allows the offending input to be printed to the output file, and also allows the scanner to skip over that line (otherwise, the input prompt will be continuously repeated, and the output file will continuously get copies of the exception message.

The most important fact to remember about file output is that the file writer must be explicitly closed, or else the file will be incomplete, and probably empty (depending on whether there was an intermediate flushing of the buffer). As we will illustrate in the next class, `Scores3`, we can ensure that a file writer is closed when (if) a program terminates by enveloping the construction of the file writer in a **`try`** block, which is followed by a **`finally`** block that closes the file writer.

For this final version of the program, we scan from an input file (with one score per line) instead of from the keyboard. As we saw in Section 0.2.5—file input is almost identical to console input. For example, to read from the file `"scores.in1"`, we start with

```
     Scanner fileScanner = new Scanner (new File ("scores.in1"));
```

Warning: This assumes that the file `scores.in1` is in the expected directory. For some Integrated Development Environments, the input file is assumed to be in the directory that is one level up from the source-file directory. Sometimes, you may need to specify a full path, such as

```
     Scanner fileScanner = new Scanner (new File
                           ("c:\\projects\\score_project\\scores.in1"));
```

Two back-slashes are required because a single back-slash would be interpreted as the escape character.

Input files seldom end with a sentinel because it is too easy to forget to add the sentinel at the end of the file. Instead, scanning continues as long as the `next()` or `nextLine()` method returns true. So for file input, we write

```
     while (fileScanner.hasNext())
```

For the sake of simplicity, if there is only one input file, we will not worry about closing that file at the end of the program: it will automatically be closed. And when it is re-opened in a subsequent program, its contents will be unchanged. A program that leaves many input files unclosed can run out of file descriptors, and an `IOException` will be thrown.

As noted earlier in this section, closing an output file entails copying the final contents of the file buffer to the file, so we should explicitly close each output file before the end of a program. Of course, if the program does not terminate—due to an infinite loop, for example—the file buffer will not be copied (unless the file was closed before the infinite loop).

The following program combines file input and file output. For the sake of generality, the program does not "hardwire" the file names (for example, `"scores.in"` and `"scores.out"`). In response to prompts, the end-user enters, from the keyboard, the names of the input and output files. If there is no input file with the given name, `FileNotFoundException` is caught, an error message is printed, and the end-user is re-prompted to enter the name of the input file. To allow this iteration, the **`try`** and **`catch`** blocks that involve throwing and handling `IOException` are placed in an outer **`while`** loop.

What if there is no file corresponding to the output file name? Normally, this is not a problem: an empty output file with that name will be created. But if file name is too bizarre for your system, such as

```
!@#$%^&*()
```

an `IOException` object (specifically, a `FileNotFoundException` object) will be thrown.

The following program has three **`try`** blocks:

1. an outermost **`try`** block to set up the files and process them, a **`catch`** block to handle a `Number FormatException` if the input contains no legal scores, followed by a **`finally`** block to close the file writer;
2. a **`try`** block/**`catch`** block sequence in an outer **`while`** loop to create the file scanner and file writer from file names scanned in from the keyboard;
3. a **`try`** block/**`catch`** sequence block in an inner **`while`** loop to scan each line from the input file and process that line, with output going to the file writer. If the input contains no legal scores, a `NumberFormatException` is thrown after this loop.

Here is the program, whose general structure is the same for all file-processing programs:

```
import java.util.*;

import java.io.*;

public class Scores3
{
   public static void main (String [ ] args)
   {
      new Scores3().run();
   } // method main

   public void run()
   {
      final String IN_FILE_PROMPT =
          "\nPlease enter the name of the input file: ";
```

```
    final String OUT_FILE_PROMPT =
        "\nPlease enter the name of the output file: ";

    final String RESULT = "\n\nThe sum of the scores is ";

    Scanner keyboardScanner = new Scanner (System.in),
            fileScanner;

    PrintWriter printWriter=null;  // to ensure that printWriter has been initialized
                                   // before it is closed in the finally block

    int sum = 0;

    try
    {
        while (true)
        {
            try
            {
                System.out.print (IN_FILE_PROMPT);
                fileScanner=new Scanner (new File (keyboardScanner.nextLine()));
                System.out.print (OUT_FILE_PROMPT);
                printWriter=new PrintWriter (new BufferedWriter
                                  (new FileWriter (keyboardScanner.nextLine())));
                sum = addScores (fileScanner, printWriter);
                break;
            } // try
            catch (IOException e)
            {
                System.out.println (e);
            } // catch
        } // while files not OK
    } // try
    catch (NumberFormatException e)
    {
        System.out.println (e);
    } // catch NumberFormatException
    finally
    {
        printWriter.println (RESULT + sum);
        printWriter.close();
    } // finally
} // method run

/**
 *  Returns the sum of the scores scanned in.
 *
 *  @param fileScanner – the Scanner object from which the scores are scanned
 *
 *  @param printWriter – the PrintWriter object to which the scores are written.
 *           If a score generates InputMismatchException, the message
```

```
     *            "java.util.InputMismatchException: " precedes the score.
     *
     *  @return the sum of the scores scanned in from fileScanner.
     *
     *  @throws NumberFormatException – if the values scanned in do not include
     *                                  an integer.
     *
     */
   public int addScores (Scanner fileScanner, PrintWriter printWriter)
   {
      final String NO_LEGAL_SCORES_MESSAGE=
          "The input contains no legal scores.";

      int score,
          sum=0;

      boolean atLeastOneScore=false;

      while (fileScanner.hasNext())
      {
          try
          {
              score=fileScanner.nextInt();
              printWriter.println (score);
              sum+=score;
              atLeastOneScore=true;
          } // try
          catch (InputMismatchException e)
          {
              printWriter.println (e+": "+fileScanner.nextLine());
          } // catch InputMismatchException
      } // while more scores in input file
      if (!atLeastOneScore)
          throw new NumberFormatException (NO_LEGAL_SCORES_MESSAGE);
      return sum;
   } // method addScores

} // class Scores3
```

Note that the message `printWriter.close()` is not in a **`catch`** block because the `printWriter` should be closed whether or not any exceptions are thrown.

Assume that the file `"scores.in1"` consists of the following four lines:

```
82
8z
77
99
```

Also, assume that there is no file named `"scores.in0"` or `"scores3.in"` in the working directory. Whether there is already a file named `"scores.out1"` or not is irrelevant. Here is a sample keyboard session, with input in boldface:

```
Please enter the name of the input file: scores.in0
java.io.FileNotFoundException: scores.in0 (The system cannot find the file specified)
Please enter the name of the input file: scores3.in
```

```
 java.io.FileNotFoundException: scores3.in (The system cannot find the file specified)

 Please enter the name of the input file: scores.in1

 Please enter the name of the output file: scores.out1
```

The final contents of the file `"scores.out1"` will be

```
82
java.util.InputMismatchException: 8z
77
99

The sum of the scores is 258
```

With file input, it is not sufficient that the file exist in order to associate a file scanner with that file. Your code must also account for the possibility that the file does not exist. The easy way to accomplish this is to include a **`throws`** `FileNotFoundException` clause immediately after the heading of the method that associates a file scanner with the file. The drawback to this approach is that if the file name is incorrect—if either the file does not exist or the file name is misspelled—then the end-user will not have the opportunity to correct the mistake.

A better alternative, as we did in the `run()` method of the class `Scores3`, is to include a **`try`** block and **`catch`** block for `FileNotFoundException`. To enable end-users to recover from incorrect file names, those blocks should be within a loop that continues until a correct file name has been entered.Similarly, to construct a file writer, `IOException` must be caught or declared in a **`throws`** clause. That is why, in the above program, the type of the relevant catch-block parameter is `IOException` instead of `FileNotFoundException`.

There is a common thread in the above examples. The `run()` method handles the aspects of the program that require the end user's intervention, such as input from the keyboard or from a GUI window, or interpretation, such as output to the console window or to a GUI window. Accordingly, the method called by the `run()` method should be *testable* in JUnit.

The major problem in testing the `addScores` method above is that the method outputs information to a file. So we will create an expected output file from a given input file, and check to make sure the expected output file matches the actual file generated by the `addScores` method. The expected file will have one line for each line in the input file, and will *not* include the final sum – because that value is not printed in the `addScores` method. We will also need an exception test for an input file with no legitimate scores, and exception tests if either the `fileScanner` or `printWriter` argument is **`null`**. Here is part of the `Scores3Test.java` file:

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;
import java.io.*;

public class Scores3Test
{
   public static void main(String[ ] args)
   {
       Result result = runClasses (Scores3Test.class);
```

```
        System.out.println ("Tests run = " + result.getRunCount() +
                            "\nTests failed = " + result.getFailures());
    } // method main

    protected Scores3 scores;
    @Before
    public void runBeforeEveryTest()
    {
        scores = new Scores3();
    } // method runBeforeEveryTest

    @Test
    public void scores3Test1() throws IOException
    {
        Scanner fileScanner = new Scanner (new File ("scores3.in1"));
        PrintWriter printWriter = new PrintWriter (new BufferedWriter
                                          (new FileWriter ("scores3.out1")));

        int actualSum = scores.addScores (fileScanner, printWriter);
        printWriter.close();
        Scanner scActual = new Scanner (new File ("scores3.out1")),
                scExpected = new Scanner (new File ("scores3.exp"));

        final int INPUT_LINES = 4;
        for (int i = 0; i < INPUT_LINES; i++)
                assertEquals (scExpected.nextLine(), scActual.nextLine());
        if (scExpected.hasNext())
                fail();
    } // method scores3Test1

    @Test (expected = NumberFormatException.class)
    public void scores3Test2() throws IOException
    {
        Scanner fileScanner = new Scanner (new File ("scores3.in2"));
        PrintWriter printWriter = new PrintWriter (new BufferedWriter
                                           (new FileWriter ("scores3.out2")));

        int actualSum = scores.addScores (fileScanner, printWriter);
    } // method scores3Test2

    @Test (expected = NullPointerException.class)
    public void scores3Test3() throws IOException
    {
        int actualSum = scores.addScores (null,
                                   new PrintWriter (new FileWriter("scores3.out3")));
    } // method scores3Test3

    @Test (expected = NullPointerException.class)
```

```
   public void scores3Test4() throws IOException
   {
       int actualSum = scores.addScores (new Scanner (new File("scores3.in1")), null);
   } // method scores3Test4

} // class Scores3Test
```

The relevant files are as follows:

```
     scores3.in1
     80
     x
     50
     y

     scores3.in2
     x
     y

     scores3.exp
     80
     java.util.InputMismatchException: x
     50
     java.uti.InputMismatchException: y
```

All tests were passed.

You are now prepared to do Lab 3:
More Practice on Unit Testing

2.5 System Testing

Just as it is unusual for a class to have a single method, it is unlikely that a project will have a single class. For a multi-class project, which class should be tested first? In an object-oriented environment, bottom-up testing is the norm. With ***bottom-up testing***, a project's low-level classes—those that are used by but do not use other classes—are tested and then integrated with higher-level classes and so on. After each of the component classes has satisfied its tests, we can perform ***system testing***, that is testing the project as a whole. Inputs for the system tests are created as soon as the project specifications are created. Note that system tests are not necessarily unit tests because system tests may entail human intervention—for example, to enter file path from the keyboard.

The purpose of testing is to detect errors in a program (or to increase confidence that no errors exist in the program). When testing reveals that there is an error in your program, you must then determine what brought about the error. This may entail some serious detective work. And the purpose of detection is correction. The entire process—testing, detection and correction—is iterative. Once an error has been corrected, the testing should start over, because the "correction" may have created new errors.

2.6 The Java Virtual Machine

Your Java classes are compiled into a low-level but machine-independent language called Java ***bytecode***. For example, the bytecode version of the file `HourlyEmployee.java` is stored in the file `HourlyEmployee.class`. The bytecode files are then interpreted and executed on your computer. The program that interprets and executes bytecode is the Java Virtual Machine. It is called a virtual machine because it executes what is almost machine-level code. There are several advantages to this arrangement

$$\text{source code} \longrightarrow \text{bytecode} \longrightarrow \text{Java Virtual Machine}$$

instead of

$$\text{source code} \longrightarrow \text{machine code}$$

The main advantage is platform independence. It doesn't matter whether your computer's operating system is Windows, Linux, or something else, the results of running your Java program will be exactly (well, almost exactly) the same on all platforms. A second benefit is customized security. For example, if the bytecode file is coming from the web, the virtual machine will not allow the application to read from or write to the local disk. But such activities would be allowed for a local application.

The Java Virtual Machine oversees all aspects of your program's run-time environment. In Sections 2.6.1 and 2.6.2, we investigate two tasks of the Java Virtual Machine.

2.6.1 Pre-Initialization of Fields

One of the Java Virtual Machine's duties is the initialization of fields just prior to the invocation of a constructor. For example, we might have the following:

```
new FullTimeEmployee ("Dilbert", 345.00)
```

First, the **`new`** operator allocates space for a `FullTimeEmployee` object. Then, to ensure that each field has at least a minimal level of initialization, the Java Virtual Machine initializes all of the class's fields according to their types. Reference fields are initialized to **`null`**, integer fields to 0, floating-point fields to 0.0, **`char`** fields to the character at position 0 in the Unicode collating sequence, and **`boolean`** fields to **`false`**. Then the specified constructor is called. Finally, the starting address of the newly constructed `FullTimeEmployee` object is returned.

There is an important consequence of this pre-initialization by the Java Virtual Machine. Even if a default constructor has an empty body—such as the one supplied by the Java compiler if your class does not declare any constructors—all fields in the class will still get initialized.

Unlike fields, local variables are *not* automatically initialized. Section 0.2.4 has the details.

2.6.2 Garbage Collection

The memory for objects is allocated when the **`new`** operator is invoked, but what about de-allocation? Specifically, what happens to the space allocated for an object that is no longer accessible? For example, suppose an object is constructed in a method, and at the end of the execution of that method, there are no references pointing to the object. The object is then inaccessible: ***garbage***, so to speak. If your program generates too much garbage, it will run out of memory, which is an error condition. Errors, unlike exceptions, should not be caught, so an error will force the abnormal termination of your program. Are you responsible for garbage collection, that is, for de-allocating inaccessible objects?

Fortunately, you need not worry about garbage collection. The Java run-time system includes a method that performs ***automatic garbage collection***. This method will be invoked if the **`new`** operator is invoked but there is not enough memory available for the object specified. With the supersizing of memory in recent years, this is an increasingly rare occurrence. To free up unused memory, the space for any object to which there are no references can be de-allocated. The garbage collector will seek out big chunks of garbage first, such as an array. In any event, this is all taken care of behind the scenes, so your overall approach to the topic of garbage collection should be "Don't worry. Be happy."

Section 2.6 investigates the relationship between packages and visibility modifiers.

2.7 Packages

A ***package*** is a collection of related classes. For each such class, the file in which the class is declared starts with the package declaration. For example, a file in a package of classes related to neural networks might start with

```
package neuralNetwork;
```

For another example, the `Scanner` class, part of the package `java.util`, is in the file `Scanner.java`, which starts with

```
package java.util;
```

If a file includes an instance of the `Scanner` class, that class can be "imported" into the file. This is done with an import directive, starting with the reserved word **`import`**:

```
import java.util.Scanner;
```

The advantage of importing is convenience: A declaration such as

```
Scanner sc;
```

can be used instead of the fully qualified name:

```
java.util.Scanner sc;
```

Many of the classes you create will utilize at least one class from the package java.util, so you can simply import the whole package:

```
import java.util.*; //the asterisk indicates that all files from java.util will be available
```

Occasionally, you may prefer to use the fully qualified name. For example, suppose your project uses two classes named `Widget`: one in the package `com.acme` and one in the package `com.doodads`. To declare (a reference to) an instance of the latter, you could write

```
com.doodads.Widget myWidget;
```

Every Java file must have a class with the visibility modifier **`public`**. Also, the name of that public class must be the same as the name of the file—without the `.java` extension. At the beginning of the file, there must be **`import`** directives for any package (or file) needed by the file but not part of the file. An exception is made for the package `java.lang`, which is automatically imported for any file.

A class member with no visibility modifier is said to have ***default visibility***. A member with default visibility can be accessed by any object (or class, in the case of a **`static`** member) in the same package as the class in which the member is declared. That is why default visibility is sometimes referred to as "package-friendly visibility." All classes without a package declaration are part of an unnamed package. But there may be more than one unnamed package so, as a general rule, if your project contains more than one class file, each file should include a package declaration.

Technically, it is possible for a Java file to have more than one class with public visibility; all but one of those classes must be ***nested***, that is, declared within another class. The Java Collections Framework, part of the package java.util, has many nested classes. Except for nested classes, a Java file is allowed to have *only one* class with public visibility. Every other non-nested class must have default visibility.

Because of the way the Java language was developed, **`protected`** visibility is not restricted to subclasses. In general, if an identifier in a class has protected visibility, that identifier can also be accessed in any class that is in the same package as the given class. For example, any class—whether or not a subclass—that is in the same package as `FullTimeEmployee` can access the `name` and `grossPay` fields of a `FullTimeEmployee` object.

In the Java Collections Framework, most of the fields have default visibility or **`private`** visibility. Almost no fields have **`protected`** visibility: Subclassing across package boundaries is discouraged in the Java Collections Framework. Why? The main reason is philosophical: a belief that the efficiency to users of the subclass is not worth the risk to the integrity of the subclass if the superclass is subsequently modified. This danger is not merely hypothetical. In Java 1.1, a class in java.security was a subclass of the `Hashtable` class. In Java 2, the `Hashtable` class was modified, and this opened a security hole in the subclass. Subclassing represents more of a commitment than mere use. So even if a class permits subclassing, it is not necessarily the wisest choice.

The bottom line is that **`protected`** visibility is even less restrictive than default visibility. This corruption of the meaning of protected visibility may make you reluctant to designate your fields as **`protected`**. An alternative is to designate the fields as **`private`**, but to create **`public`** methods to get and set the values of those **`private`** fields. As described in Programming Exercise 1.3, an ***accessor*** method returns a copy of a field (or a copy of the object referenced, if the field is a reference), and a ***mutator*** method alters a field (or the object referenced by the field). The usefulness of this approach diminishes as the number of fields increases.

The final topic in this chapter looks at the importance of overriding the `Object` class's `equals` method, the barriers to overriding that method, and how those barriers are overcome.

2.8 Overriding the `Object` Class's `equals` Method

In Section 1.3.3, we saw the method specification for the `equals` method in the `Object` class, the superclass of all classes. Here is that specification:

```
/**
 *  Determines if this Object object is the same as a specified Object
 *  object.
 *
 *  @param obj - the Object object to be compared to the calling Object object.
 *
 *  @return true - if the two objects are the same.
 *
 */
public boolean equals (Object obj)
```

This method, as with the other methods in the `Object` class, is intended to be overridden by subclasses, which can compare field values, for example. The object class has no fields, so what does it compare?

It compares references, specifically, the calling object reference with the argument reference. Here is the definition:

```
public boolean equals (Object obj)
{
     return this == obj;
} // method equals
```

As we saw in Section 1.3.2, in any class, the reserved word **this** is a reference to the calling object. For example, suppose the call is

```
obj1.equals (obj2)
```

Then in the definition of the `equals` method, **this** is a reference to the object that is also referenced by `obj1`, and `obj` is a reference to the object that is also referenced by `obj2`.

Because the `Object` class's `equals` method compares references, any class with an `equals` method should define its own version of that method. For example, suppose we decide to add an `equals` method to the `FullTimeEmployee` class. The first question is: Should we overload, that is,

```
public boolean equals (FullTimeEmployee full)
```

or override, that is,

```
    public boolean equals (Object obj)

?
```

Overloading `equals`—that is, having a different parameter list than the version inherited from the `Object` class—can be done fairly simply. The only obstacle is that **double** values should not be directly tested for equality; note, for example, that `System.out.println (.4==10.0 - 9.6)` outputs "false", (but `System.output.println (.4==1.0 - .6)` outputs "true"). Here is the definition:

```
public boolean equals (FullTimeEmployee full)
{
     return name.equals (full.name) ""
              MONEY.format (grossPay).equals (MONEY.format (full.grossPay));
} // overloading method equals
```

Recall that the `format` method rounds off the value in the `grossPay` field, so we need not compare `grossPay` and `full.grossPay` for equality. This version compares objects, not references, and so the value `true` would be printed by each of the following:

```
System.out.println (new FullTimeEmployee ("a", 100.00).equals
                    (new FullTimeEmployee ("a", 100.00)));

System.out.println (new HourlyEmployee ("a", 10, 10.00).equals
                    (new FullTimeEmployee ("a", 100.00)));
```

The overloaded version works well as long as the type of the calling object is known, at compile-time, to be `FullTimeEmployee` (or subclass of `FullTimeEmployee`). Sadly, that is not always the case. For example, many of the classes in the Java Collections Framework store a collection of objects. Those classes have a contains method to determine if a given object occurs in the collection. The contains method's heading is

```
public boolean contains (Object obj)
```

Typically, in testing for containment, the equals method is invoked, with obj as the calling object. For a given application, the collection may consist of FullTimeEmployee objects. But when the equals method—called by contains—is compiled, the only information available about the calling object is its declared type: Object. Therefore, the compiler generates bytecode for a call to the equals method in the Object class, which takes an Object parameter. At run time, when the class of the object (referenced by) obj is available, the version of the Object-parameter equals method executed will be the one in the Object class unless that method has been overridden. Whether the equals method has been overloaded is irrelevant!

Now that we have established the significance of overriding the Object class's equals method, let's see how to do it. We will take the FullTimeEmployee class as an example. The basic idea is simple: if the type of the argument object is not FullTimeEmployee, return **false**. Otherwise, as we did earlier in this section, compare the values returned by the toString( ) method of the calling object and the argument object. Here are some sample results:

```
System.out.println (new FullTimeEmployee ("a", 100.00).equals
                           ("yes"));                            // false
System.out.println (new FullTimeEmployee ("a", 100.00).equals
                           (new FullTimeEmployee ("a", 100.00)));  // true
System.out.println (new FullTimeEmployee ("a", 100.00).equals
                           (new FullTimeEmployee ("b", 100.00)));  // false
System.out.println (new FullTimeEmployee ("a", 100.00).equals
                           (new FullTimeEmployee ("a", 200.00)));  // false
```

Here is the full definition:

```
public boolean equals (Object obj)
{
       if (!(obj instanceof FullTimeEmployee))
                return false;
       FullTimeEmployee full = (FullTimeEmployee)obj;
       return name.equals (full.name) &&
                MONEY.format (grossPay).equals (MONEY.format (full.grossPay));
} // method equals
```

To summarize this section:

1. Every class whose instances might be elements of a collection should have an equals method that overrides the Object class's equals method.

2. The **instanceof** operator returns **true** if and only if, at run-time, the object referenced by the left operand is an instance of the class that is the right operand.

3. Before comparing the calling object with the argument object, cast the parameter type, Object, to the class in which equals is being defined.

Programming Exercise 2.11 has even more information about the equals method.

SUMMARY

The **static** modifier is used for identifiers that apply to a class as a whole, rather than to a particular instance of a class. Constants should be declared to be static, because then there will be only one copy of the constant, instead of one copy for each instance of the class. To access a static identifier outside of its class, the class identifier—rather than an object—is the qualifier.

JUnit is an Open Source software product that allows the methods in a class to be tested without human intervention. The tests are developed as soon as the method specifications are created. In general, methods should be designed to facilitate testing without human intervention, so input from System.in and output to System.out should be avoided in methods to be tested.

An **exception** is an object that signals a special situation, usually that an error has occurred. An exception can be handled with **try**/**catch** blocks. The sequence of statements in the **try** block is executed. If, during execution, an exception is thrown (indicating that an error has occurred), the appropriate **catch** block is executed to specify what, if anything, is to be done.

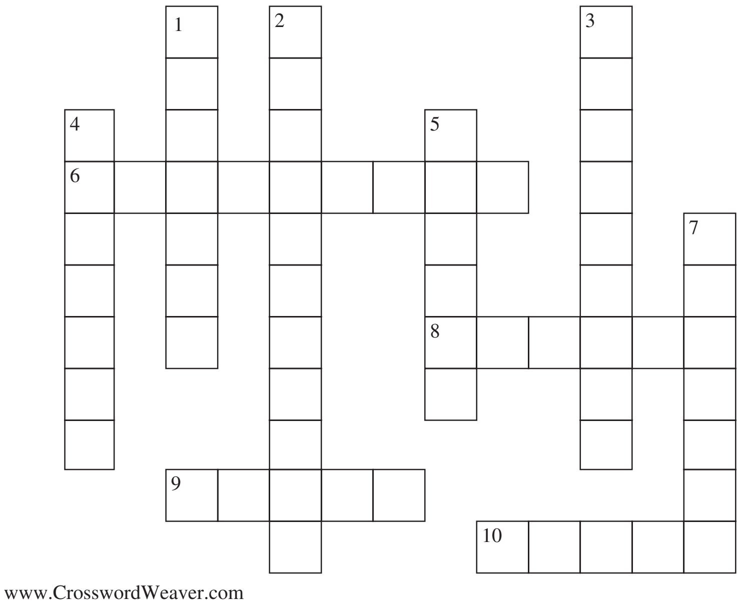
File output is similar to console-oriented output, except that a `PrintWriter` object is explicitly created to write to the specified output file. The output is not immediately sent to the output file, but rather to a buffer. At the conclusion of file processing, the buffer is flushed to the output file by a call to the **close** method.

The Java run-time, also known as the Java Virtual Machine, is a program that interprets and executes the bytecode output from a Java compiler. Among other tasks, the Java Virtual Machine is responsible for pre-initialization of fields, de-allocation of inaccessible objects, and managing threads.

A package is a collection of related classes. An identifier with no visibility modifier is said to have ***default visibility***. Java is "package friendly." That is, an identifier with default visibility can be accessed by any object (or class, in the case of a static member) in the same package as the class in which the identifier is declared. If a given class's identifier has **protected** visibility, that identifier can be accessed in any subclass of the given class, even in a different package. Unfortunately, that identifier may also be accessed in any class—even if not a subclass—within the given package's class.

The `equals` method in the `Object` class should be overridden for any class `C` whose instances might become elements of a collection. The overriding method invokes the **instanceof** method to return **false** for any argument object that is not an instance of class `C`, and then casts the `Object` class to class `C` in order to make the appropriate comparison(s).

CROSSWORD PUZZLE



ACROSS

6. An object created by an unusual condition, typically, an attempt at invalid processing.

8. An identifier associated with a class itself rather than with an instance of the class is called a _______ identifier.

9. A reserved-word modifier associated with a location that can be assigned to only once.

10. A method in the `PrintWriter` class that ensures a file is complete by flushing the output buffer.

DOWN

1. The kind of exception for which the compiler confirms that the exception is caught within the method or that a throws clause is appended to the method's heading.

2. When an exception is thrown in a method that does not catch the exception, the transferring of control back to the calling method is referred to as ___________ the exception.

3. A class member that can be accessed in any class within the same package as the given class or in any subclass of the given class is said to have ________ visibility.

4. A class member that can be accessed in any class within the same package as the given class, but not elsewhere, is said to have _________ visibility.

5. A program that does not terminate unexpectedly from invalid user-input is called a _________ program.

7. A collection of related classes.

CONCEPT EXERCISES

2.1 The `System` class in `java.lang` has a class constant identifier that has been extensively used in Chapters 0, 1 and 2. What is that constant identifier? Why should a class's constant identifiers be **`static`** ? Should a method's constant identifiers be **`static`** ? Explain.

2.2 Create a **`catch`** block that will handle any exception. Create a **`catch`** block that will handle any input/output exception. Create a **`catch`** block that will handle any run-time exception.

2.3 What is wrong with the following skeleton?

```
try
{
        ...
} // try
catch (IOException e)
{
        ...
} // catch IOException
catch (FileNotFoundException e)
{
        ...
} // catch FileNotFoundException
```

2.4 Suppose `fileScanner` is a `Scanner` object for reading from an input file, and `printWriter` is a `PrintWriter` object for writing to an output file. What will happen if, at the end of a program, you forget to close `fileScanner` ? What will happen if, at the end of a program, you do not close `printWriter` ?

2.5 What does "bottom-up" testing mean with respect to the classes in a project?

2.6 Suppose we create a two-dimensional array (literally, an array in which each element is an array). The following creates an **`int`** array with 50000 rows and 100000 columns:

```
int [ ][ ] a = new int [50000][100000];
```

If this code is executed, the program terminates abnormally, and the message is

```
java.lang.OutOfMemoryError
Exception in thread "main"
```

Why wasn't memory re-allocated by the garbage collector? Hypothesize whether this abnormal termination be handled with a **`try`**-block and **`catch`**-block. Test your hypothesis and explain.

2.7 Can a **`protected`** field be accessed outside of the class in which it is declared and subclasses of that class? What does the following statement mean? "Subclassing represents more of a commitment than mere use."

2.8 Arrays are strange objects because there is no array class. But an array object can call methods from the `Object` class. Determine and explain the output from the following code:

```
int [ ] a = new int [10];

int [ ] b = new int [10];

a [3] = 7;
b [3] = 7;
System.out.println (a.equals(b));
```

PROGRAMMING EXERCISES

2.1 Develop the specification for a method that scans one line that is supposed to contain three **double** values and returns the largest. Throw all possible exceptions. Start with a stub for your method and create a test class to test your method. Re-test your method as you define it. Finally, include a main method and a run( ) method that calls the method you developed.

2.2 Develop the specification for a method that scans (what are supposed to be) **double** values from a file and returns the largest. Throw all possible exceptions. Start with a stub for your method and create a test class to test your method. Re-test your method as you define it. Finally, include a main method and a run( ) method that calls the method you developed.

2.3 Modify the run method for the Company class to scan from an input file and write to an output file. Include a re-prompt if either the input or output path is incorrect.

2.4 Hypothesize what is wrong with the following method:

```
public static boolean isEven (int i)
{
    if (i % 2 == 0)
       return true;
    if (i % 2 != 0)
       return false;
} // method isEven
```

Test your hypothesis by calling this method from a run( ) method. Can a **try**-block and **catch**-block handle the problem? Explain.

2.5 Hypothesize the output from the following:

```
System.out.println (null + "null");
```

Test your hypothesis. Provide the code in the String class that explains why the output is what it is.

2.6 Give an example to show that **private** visibility is more restrictive than default visibility. Give an example to show that default visibility is more restrictive than **protected** visibility. Give an example to show that **protected** visibility is more restrictive than **public** visibility. In each case, test your code to make sure that the more restrictive choice generates a compile-time error message. No error message should be generated for the less restrictive choice.

2.7 Protectedness transfers across packages, but only within a subclass, and only for objects whose type is that subclass. For a bare-bones illustration, suppose we have class A declared in package APackage:

```
package APackage;

public class A
{
      protected int t;
} // class A
```

Also, suppose that classes C and D are subclasses of A and that C and D are in a different package from A. Then within class D, the t field is treated as if it were declared in D *instead of* in A. Here are possible declarations for classes C and D:

```
import APackage.*;

public class C extends A { }
```

Class D is declared in another file. For each of the four accesses of t in the following declaration of class D, hypothesize whether the access is legal or illegal:

```
import APackage.*;

public class D extends A
{

    public void meth()
     {
          D d = new D();
          d.t = 1;           // access 1
          t = 2;             // access 2
          A a = new A();
          a.t = 3;           // access 3
          C c = new C();
          c.t = 4;           // access 4
    } method meth

} // class D
```

Test your hypotheses by creating and running a project that includes the above files.

2.8 Re-do Programming Exercise 1.2 to print out the number of above-average salaries. Use an array field to hold the salaries, and assume there will be at most 10 salaries in the input.

2.9 Study the specification of the `arraycopy` method in the `System` class, and then write a short program that uses the `arraycopy` method to copy all the elements of an array to another array. Output the elements in the destination array to make sure the copying actually occurred.

2.10 Re-do Programming Exercise 2.8 if the input can contain an arbitrary number of salaries.
Hint: Start with an array of length 10. Whenever the number of salaries in the input exceeds the current length of the array field, create a new array of twice that length, copy the old array to the new array—see Programming Exercise 2.9—and then assign the new array (reference) to the old array (reference).

2.11 According to the full method specification in the `Object` class, any override of the `Object` class's `equals` method should satisfy the following five properties:

1. ***reflexivity***, that is, for any non-**`null`** reference `x`,

```
x.equals (x)
```

should return **`true`**.

2. ***symmetry***, that is, for any non-**`null`** references `x` and `y`,

```
x.equals (y)
```

should return the same result as

```
y.equals (x)
```

3. ***transitivity***, that is, for any references `x`, `y` and `z` if

```
x.equals (y)
```

returns **`true`**, and

```
y.equals (z)
```

returns **`true`**, then

```
x.equals (z)
```

should return **`true`**.

4. ***consistency***, that is, for any non-**`null`** references `x` and `y`, multiple invocations of

   ```
   x.equals (y)
   ```

   should consistently return **`true`** or consistently return **`false`**, provided no information used in `equals` comparisons on the objects is modified.

5. ***actuality***, that is, for any non-**null** reference `x`,

   ```
   x.equals (null)
   ```

   should return **`false`**.

   For the `FullTimeEmployee` class's `equals` method (see Section 2.7), provide examples to support the claim that the `equals` method satisfies those five properties. You are not being asked to *prove* that the `FullTimeEmployee` class's `equals` method satisfies those properties.

2.12 Create and run a test class for the `equals` method defined in Section 2.7 for the `FullTimeEmployee` class.

Programming Project 2.1

An Integrated Web Browser and Search Engine, Part 1

Note: This project assumes familiarity with developing graphical user interfaces.

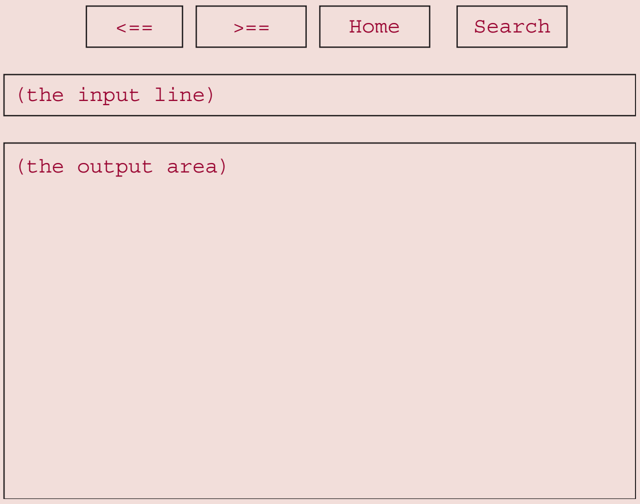
This is the first part of a seven-part project to create an integrated web browser and search engine. The remaining parts are in Chapters 6, 7, 12, 13, 14 and 15. The project is based on a paper by Newhall and Meeden [2002].

Basically, all the project does at this stage is to display web pages. Initially the output area of the Graphical User Interface (GUI) window displays the home page. That page has a link to another page, and if the end user clicks on the link, that page will be displayed in the output area. In addition to the output area, the GUI window will also have four buttons: one to go forward (currently disabled), one to go backward (currently disabled), one to display the home page (enabled), and one to conduct a search (currently disabled). Finally, the GUI window will have an input line that allows an end user to enter a file path; when the Enter key is pressed, that file (that is, that page) will be displayed.

Analysis The following specifications apply to the GUI:

1. The size of the window will be 700 pixels wide and 500 pixels high.
2. The upper-left-hand corner of the window will have x-coordinate 150 and y-coordinate 250.
3. Each of the four buttons on the top of the window will be 80 pixels wide and 30 pixels high. The foreground color of the Home button will be green, and the foreground color of the other three buttons will be red.
4. The input line will be 50 pixels wide.
5. The output area will be scrollable in both directions.

Here is a diagram of the GUI:



6. The only tags allowed in a page are link tags, for example,

```
< a href = browser.in4 > browser4 < /a >
```

   In this example, the only part that will appear in the output area is `browser4`.
7. For simplicity, all links (such as `browser.in4` above) will come from the same directory, and all link ‘‘nicknames’’ (such as `browser4` above) will consist of a single word.
8. In the output area, the foreground color of each link’s nickname should be blue.
9. A line in a page may have several link tags, but no tag will be split between two lines.
10. If a page clicked on or typed in does not exist, the following error message should appear in the output area:

```
Web page not found: HTTP 404
```

   At that point, the end user can click on the Home button, can enter a new file path in the input line, or can close the application.

Hints:

1. Use the layout manager `FlowLayout`.
2. Use a `JTextField` for the input line and a `JTextPane` (made scrollable as a `JScrollPane`) for the output area. A `JTextPane` component supports embedded links (as `JButton` objects).
3. Use a separate listener class so that there can be a listener object for each link that is clicked on.

Here is a sample home page:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
```

(continued on next page)

(continued from previous page)

```
Where Alph, the sacred river, ran
Through caverns <a href=browser.in2>browser2</a> measureless to man
Down to a sunless sea.
```

When that page is displayed in the output area, it will appear as

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns  browser2  measureless to man
Down to a sunless sea.
```

If the end user now clicks on browser2, the contents of browser.in2 will be displayed. Here are the contents of browser.in1, browser.in2, browser.in4, and browser.in5 (browser.in3 does not exist):
browser.in1:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down <a href=browser.in2>browser2</a> to a sun-
less <a href=browser.in4>browser4</a> sea.
```

browser.in2:

```
In Xanadu did Kubla Khan
A stately <a href=browser.in3>browser3</a> pleasure-dome decree:
Where Alph, the sacred river, <a href=browser.in4>the browser4</a> ran
Through caverns measureless to man
Down to a <a href=browser.in5>browser5</a> sunless sea.
```

browser.in4

```
In Xanadu did <a href=browser.in1>browser1</a> Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns  measureless to man
Down to a sunless sea.
```

browser.in5:

```
In Xanadu did <a href=browser.in2>browser2</a> Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns  measureless to man
Down to a sunless sea.
```

Analysis of Algorithms

CHAPTER 3

As noted in Section 2.2, a *correct* method is one that satisfies its specification. In defining a method, the first goal is to make sure the method is correct, and unit testing allows us to increase our confidence in a method's correctness. But if the method's execution time or memory requirements are excessive, the method may be of little value to the application. This chapter introduces two tools for measuring a method's efficiency. The first tool provides an estimate, based on studying the method, of the number of statements executed and number of variables allocated in a trace of the method. The second tool entails a run-time analysis of the method. Both tools are useful for comparing the efficiency of methods, and the tools can complement each other.

CHAPTER OBJECTIVES

1. Be able to use Big-O (and Big-Theta) notation to estimate the time and space requirements of methods.
2. Be able to conduct run-time analyses of methods.

3.1 Estimating the Efficiency of Methods

The correctness of a method depends only on whether the method does *what* it is supposed to do. But the efficiency of a method depends to a great extent on *how* that method is defined. How can efficiency be measured? We could test the method repeatedly, with different arguments. But then the analysis would depend on the thoroughness of the testing regimen, and also on the compiler, operating system and computer used. As we will see in Section 3.2, run-time analysis can have blaring weaknesses, mainly due to the "noise" of other processes that are executing at the same time as the method being tested.

At this stage, we prefer a more abstract analysis that can be performed by directly investigating the method's definition. We will ignore all memory restrictions, and so, for example, we will allow an `int` variable to take on any integer value and an array to be arbitrarily large. Because we will study the method without regard to a specific computer environment, we can refer to the method as an ***algorithm***, that is, a finite sequence of explicit instructions to solve a problem in a finite amount of time.

The question then is, how can we estimate the execution-time requirements of a method from the method's definition? We take the number of statements executed in a trace of a method as a measure of the execution-time requirements of that method. This measure will be represented as a function of the "size" of the problem. Given a method for a problem of size n, let ***worstTime(n)*** be the maximum—over all possible parameter/input values—number of statements executed in a trace of the method.

For example, let's determine worstTime(n) for the following method, which returns the number of elements greater than the mean of an array of non-negative `double` values. Here `n` refers to the length of the array.

```
/**
 *  Returns the number of elements in a non-empty  array that are greater than
 *  the mean of that array.
 *
 *  @param a - an array of double values
 *  @param mean - the sum of the  elements in a, divided by a.length.
 *
 *  @return the number of elements in a that are greater than mean
 *
 */
public static int aboveMeanCount (double[ ] a, double mean)
{
   int n = a.length,
       count = 0;

   for (int i = 0; i < n; i++)
      if (a [i] > mean)
           count++;
   return count;
}  // method aboveMeanCount
```

There are six statements that will be executed only once: the assignment of the arguments to the parameters `a` and `mean`; the initialization of `n, count` and `i`; and the return of `count`. Within the **`for`** statement, `i` will be compared to `n` a total of `n` + 1 times, `i` will be incremented `n` times and the comparison of `a [i]` to `mean` will be made `n` times. If `n` − 1 elements have the value 1.0 and the other element has the value 0.0, then `a [i]` will be greater than `mean` a total of `n` − 1 times, so `count` will be incremented `n` − 1 times. The total number of statements executed in the worst case, that is, worstTime(n), is

$$6 + (n + 1) + n + n + (n - 1) = 4n + 6$$

Sometimes we will also be interested in the average-case performance of a method. We define ***averageTime(n)*** to be the average number of statements executed in a trace of the method. This average is taken over all invocations of the method, and we assume that each set of n parameter/input values for a call is equally likely. For some applications, that assumption is unrealistic, so averageTime(n) may not be relevant.

In the **`for`** loop of the just completed example, `a [i]` will be greater than `mean`, on average, half of the time, so `count` will be incremented only $n/2$ times. Then averageTime(n) is $3.5n + 7$.

Occasionally, especially in Chapters 5 and 11, we will also be interested in estimating the space requirements of a method. To that end, we define ***worstSpace(n)*** to be the maximum number of variables allocated in a trace of the method, and ***averageSpace(n)*** to be the average number of variables allocated in a trace of the method. For an array, we treat each element—that is, indexed variable—to be a separate variable. So an array of length n would contribute n variables. The `aboveMeanCount` method does *not* create an array; worstSpace(n) = averageSpace(n) = 5.

3.1.1 Big-O Notation

We need not calculate worstTime(n) and averageTime(n)—or worstSpace(n) and averageSpace(n)—exactly since they are only crude approximations of the time requirements of the corresponding method. Instead, we approximate those functions by means of "Big-O" notation, defined in the next paragraph. Because we are looking at the method by itself, this "approximation of an approximation" is quite satisfactory for giving us an idea of how fast the method will be.

The basic idea behind Big-O notation is that we often want to determine an *upper bound* for the behavior of a function, that is, to determine how bad the performance of the function can get. For example, suppose we are given a function f. If some function g is, loosely speaking, an upper bound for f, then we say that f is Big-O of g. When we replace "loosely speaking" with specifics, we get the following definition:

Let g be a function that has non-negative integer arguments and returns a non-negative value for all arguments. A function f is said to be **O(*g*)** if for some positive constant C and some non-negative constant K,

$$f(n) \leq C\, g(n) \quad \text{for all} \quad n \geq K.$$

If f is O(g), pronounced "big-oh of g", we can also say "f is of order g".

The idea behind Big-O notation is that if f is O(g) then eventually f is bounded above by some constant times g, so we can use g as a crude upper-bound estimate of the function f.

By a standard abuse of notation, we often associate a function with the value it calculates. For example, let g be the function defined by

$$g(n) = n^3, \quad \text{for} \quad n = 0, 1, 2, \ldots$$

Instead of writing O(g) we write O(n^3).

The following three examples will help you to understand the details of Big-O notation. Then, in Section 3.1.2, we will describe how to arrive at Big-O estimates without going through the details.

Example 3.1

Let f be the function worstTime defined for the `aboveMeanCount` method in Section 3.1 and repeated here:

```
public static int aboveMeanCount (double[ ] a, double mean)
{
   int n = a.length,
       count = 0;

   for (int i = 0; i < n; i++)
      if (a [i] > mean)
          count++;
   return count;
}  // method aboveMeanCount
```

Then

$$f(n) = 4n + 6, \quad \text{for} \quad n = 0, 1, 2, \ldots$$

Show that f is O(n).

SOLUTION

We need to find non-negative constants C and K such that $f(n) \leq C^*n$ for all $n \geq K$. We will show that each term in the definition of f is less than or equal to some constant times n for n greater than or equal to some non-negative integer. Right away, we get:

$$\begin{aligned} 4n &\leq 4n \quad \text{for} \quad n \geq 0, \text{and} \\ 6 &\leq 6n \quad \text{for} \quad n \geq 1. \end{aligned}$$

So for any $n \geq 1$,

$$f(n) \leq 4n + 6n = 10n.$$

That is, for $C = 10$ and $K = 1, f(n) \leq C^*n$ for all $n \geq K$. This shows that f is $O(n)$.

In general, if f is a polynomial of the form

$$a_i n^i + a_{i-1} n^{i-1} + \cdots + a_1 n + a_0$$

then we can establish that f is $O(n^i)$ by choosing $K = 1,\ C = |a_i| + |a_{i-1}| + \cdots + |a_1| + |a_0|$ and proceeding as in Example 3.1.

The next example shows that we can ignore the base of a logarithm when determining the order of a function.

Example 3.2

Let a and b be positive constants. Show that if f is $O(\log_a n)$ then f is also $O(\log_b n)$.

SOLUTION

Assume that f is $O(\log_a n)$. Then there are non-negative constants C and K such that for all $n \geq K$,

$$f(n) \leq C * \log_a n$$

By a fundamental property of logarithms (see Section A2.4 of Appendix 2),

$$\log_a n = (\log_a b) * (\log_b n) \quad \text{for any} \quad n > 0.$$

Let $C_1 = C * \log_a b$.

Then for all $n \geq K$, we have

$$f(n) \leq C * \log_a n = C * \log_a b * \log_b n = C_1 * \log_b n,$$

and so f is $O(\log_b n)$.

Because the base of the logarithm is irrelevant in Big-O estimates, the base is usually omitted. For example, you will often see $O(\log n)$ instead of $O(\log_2 n)$ or $O(\log_{10} n)$.

The final example in this introduction to Big-O notation illustrates the significance of nested loops in estimating worstTime(n).

Example 3.3

Show that worstTime(n) is O(n^2) for the following nested loop:

```
for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)
          System.out.println (i + j);
```

SOLUTION

For this nested loop, every trace will entail the execution of the same number of statements. So worstTime(n) and averageTime(n) will be equal. And that is frequently the case.

In the outer loop, the initialization of `i` is executed once, the continuation condition, `i < n`, is executed $n + 1$ times, and `i` is incremented n times. So far, we have

$$1 + (n + 1) + n$$

statements executed. For each of the n iterations of the outer loop, the initialization of `j` is executed once, the continuation condition, `j < n`, is executed $n + 1$ times, `j` is incremented n times, and the call to `println` is executed n times. That increases the number of statements executed by

$$n(1 + (n + 1) + n)$$

The total number of statements executed is

$$1 + (n + 1) + n + n(1 + (n + 1) + n) = 2n^2 + 4n + 2$$

Since the same number of statements will be executed in every trace, we have

$$\text{worstTime}(n) = 2n^2 + 4n + 2$$

By the same technique used in Example 3.1,

$$\text{worstTime}(n) \le 8n^2 \quad \text{for all} \quad n \ge 1.$$

We conclude that worstTime(n) is O(n^2).

In Example 3.3, the number of statements executed in the outer loop is only $2n + 2$, while $2n^2 + 2n$ statements are executed in the inner loop. In general, most of the execution time of a method with nested loops is consumed in the execution of the inner(most) loop. So that is where you should devote your efforts to improving efficiency.

Note that Big-O notation merely gives an upper bound for a function. For example, if f is O(n^2), then f is also O($n^2 + 5n + 2$), O(n^3) and O($n^{10} + 3$). Whenever possible, we choose the *smallest* element from a hierarchy of orders, of which the most commonly used are shown in Figure 3.1. For example, if $f(n) = n + 7$ for $n = 0, 1, 2, \ldots$, it is most informative to say that f is O(n)—even though f is also O($n \log n$) and O(n^3). Similarly, we write O(n) instead of O($2n + 4$) or O($n - \log n$), even though O(n) = O($2n + 4$) = O($n - \log n$); see Exercise 3.9.

Figure 3.2 shows some more examples of functions and where they fit in the order hierarchy.

$$\boxed{O(1) \subset O(\log n) \subset O(n^{1/2}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \ldots \subset O(2^n) \ldots}$$

FIGURE 3.1 Some elements in the Big-O hierarchy. The symbol "$\subset$" means "is contained in". For example, every function that is in O(1) is also in O(log n)

Order	Sample Function
$O(1)$	$f(n) = 3000$
$O(\log n)$	$f(n) = (n * \log_2(n+1) + 2)/(n+1)$
$O(n)$	$f(n) = 500 \log_2 n + n/100000$
$O(n \log n)$	$f(n) = \log_2 n^n$ //see Section A2.4 of Appendix 2
$O(n^2)$	$f(n) = n * (n+1)/2$
$O(2^n)$	$f(n) = 3500\, n^{100} + 2^n$

FIGURE 3.2 Some sample functions in the order hierarchy

One danger with Big-O notation is that it can be misleading when the values of n are small. For example, consider the following two functions for $n = 0, 1, 2, \ldots$.

$$f(n) = 1000\,n \text{ is } O(n)$$

and

$$g(n) = n^2/10 \text{ is } O(n^2)$$

But $f(n)$ is larger than $g(n)$ for all values of n less than 10,000.

The next section illustrates how easy it can be to approximate worstTime(n)—or averageTime(n)—with the help of Big-O notation.

3.1.2 Getting Big-O Estimates Quickly

By estimating the number of loop iterations in a method, we can often determine at a glance an upper bound for worstTime(n). Let `S` represent any sequence of statements whose execution does not include a loop statement for which the number of iterations depends on n. The following method skeletons provide paradigms for determining an upper bound for worstTime(n).

Skeleton 1. worstTime(n) is O(1):

```
S
```

For example, for the following constructor from Chapter 1, worstTime(n) is O(1):

```
public HourlyEmployee (String name, int hoursWorked, double payRate)
{
  this.name = name;
  this.hoursWorked = hoursWorked;
  this.payRate = payRate;

  if (hoursWorked <= MAX_REGULAR_HOURS)
  {
      regularPay = hoursWorked * payRate;
      overtimePay = 0.00;
  } // if
  else
  {
    regularPay = MAX_REGULAR_HOURS * payRate;
    overtimePay = (hoursWorked - MAX_REGULAR_HOURS) * (payRate * 1.5);
  } // else
  grossPay = regularPay + overtimePay;
} // 3-parameter constructor
```

Because n is not specified or implied, the number of statements executed is constant—no matter what n is—so worstTime(n) is O(1). It follows that for the following method in the `HourlyCompany` class, worstTime(n) is also O(1):

```
protected FullTimeEmployee getNextEmployee (Scanner sc)
{
  String name = sc.next();

  int hoursWorked = sc.nextInt();

  double payRate = sc.nextDouble();
  return new HourlyEmployee (name, hoursWorked, payRate);
} // method getNextEmployee
```

Note that the execution of `S` may entail the execution of millions of statements. For example:

```
double sum = 0;
for (int i = 0; i < 10000000; i++)
        sum += Math.sqrt (i);
```

The reason that worstTime(n) is O(1) is that the number of loop iterations is constant and therefore independent of n. In fact, n does not even appear in the code. In any subsequent analysis of a method, n will be given explicitly or will be clear from the context, so you needn't worry about "What is n?"

Skeleton 2. worstTime(n) is O(*log* n):

```
while (n > 1)
{
    n = n / 2;
    S
} // while
```

Let $t(n)$ be the number of times that `S` is executed during the execution of the **`while`** statement. Then $t(n)$ is equal to the number of times that n can be divided by 2 until n equals 1. By Example A2.2 in Section A2.5 of Appendix 2, $t(n)$ is the largest integer $\le \log_2 n$. That is, $t(n) = \mathit{floor}(\log_2 n)$.[1] Since $\mathit{floor}(\log_2 n) \le \log_2(n)$ for any positive integer n, we conclude that $t(n)$ is O(log n) and so worstTime(n) is also O(log n).

The phenomenon of repeatedly splitting a collection in two will re-appear time and again in the remaining chapters. Be on the lookout for the splitting: it signals that worstTime(n) will be O(log n).

[1] `floor(x)` returns the largest integer that is less than or equal to `x`.

The Splitting Rule

In general, if during each loop iteration, *n* is divided by some constant greater than 1, worstTime(*n*) will be O(log *n*) for that loop.

As an example of the Splitting Rule, here—from the `Arrays` class in the package `java.util`—is the most widely known algorithm in computer science: the Binary Search Algorithm. Don't get hung up in the details; we will study a binary search algorithm carefully in Chapter 5. Here, *n* refers to the size of the array being searched.

```
/**
 * Searches the specified array of ints for the specified value using the
 * binary search algorithm.  The array must be sorted (as
 * by the sort method, above) prior to making this call.  If it
 * is not sorted, the results are undefined.  If the array contains
 * multiple elements with the specified value, there is no guarantee which
 * one will be found.
 *
 * @param a the array to be searched.
 * @param key the value to be searched for.
 * @return index of the search key, if it is contained in the list;
 *    otherwise, (-(insertion point) - 1).  The
 *    insertion point is defined as the point at which the
 *    key would be inserted into the array a: the index of the first
 *    element greater than the key, or a.length, if all
 *    elements in the array are less than the specified key.  Note
 *    that this guarantees that the return value will be greater than
 *    or equal to 0 if and only if the key is found.
 * @see #sort(int[ ])
 */
 public static int binarySearch(int[ ] a, int key)
 {
    int low = 0;
    int high = a.length-1;

    while (low <= high)
    {
        int mid = (low + high) >> 1;  // same effect as (low + high) / 2,
                                      // but see Programming Exercise 3.5
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
          high = mid - 1;
        else
          return mid; // key found
    } // while
    return -(low + 1);  // key not found.
 } // method binarySearch
```

At the start of each loop iteration, the area searched is from index `low` through index `high`, and the action of the loop reduces the area searched by half. In the worst case, the key is not in the array, and the loop continues until `low > high`. In other words, we keep dividing n by 2 until $n = 0$. (Incidentally, this repeated dividing by 2 is where the "binary" comes from in Binary Search Algorithm.) Then, by the Splitting Rule, worstTime(n) is O(log n) for the loop. And with just a constant number of statements outside of the loop, it is clear that worstTime(n) is O(log n) for the entire method.

Skeleton 3. worstTime(n) is O(n):

```
for (int i = 0; i < n; i++)
{
      S
} // for
```

The reason that worstTime(n) is O(n) is simply that the **`for`** loop is executed `n` times. It does not matter how many statements are executed during each iteration of the **`for`** loop: suppose the maximum is k statements, for some positive integer k. Then the total number of statements executed is $\leq kn$. Note that k must be positive because during each iteration, `i` is incremented and tested against `n`.

As we saw in Section 3.1, worstTime(n) is O(n) for the `aboveMeanCount` method. But now we can obtain that estimate simply by noting that the loop is executed n times.

```
public static int aboveMeanCount (double[ ] a, double mean)
{
   int n = a.length,
       count = 0;

   for (int i = 0; i < n; i++)
       if (a [i] > mean)
            count++;
   return count;
}  // method aboveMeanCount
```

For another example of a method whose worstTime(n) is O(n), here is another method from the `Arrays` class of the package `java.util`. This method performs a ***sequential search*** of two arrays for equality; that is, the search starts at index 0 of each array, and compares the elements at each index until either two unequal elements are found or the end of the arrays is reached.

```
/**
 * Returns true if the two specified arrays of longs are
 * equal to one another.  Two arrays are considered equal if both
 * arrays contain the same number of elements, and all corresponding pairs
 * of elements in the two arrays are equal.  In other words, two arrays
 * are equal if they contain the same elements in the same order.  Also,
 * two array references are considered equal if both are null.
 *
 * @param a one array to be tested for equality.
 * @param a2 the other array to be tested for equality.
 * @return true if the two arrays are equal.
 */
 public static boolean equals (long[ ] a, long[ ] a2)
 {
   if (a==a2)
       return true;
   if (a==null || a2==null)
       return false;

   int length = a.length;
   if (a2.length != length)
       return false;

   for (int i=0; i<length; i++)
     if (a[i] != a2[i])
        return false;

     return true;
 } // method equals
```

Skeleton 4. worstTime(n) is O(n log n):

```
int m;

for (int i = 0; i < n; i++)
{
    m = n;
    while (m > 1)
    {
         m = m / 2;
         S
    } // while
} // for
```

The **for** loop is executed `n` times. For each iteration of the **for** loop, the **while** loop is executed *floor*($\log_2 n$) times—see Example 2 above—which is $\leq \log_2 n$. Therefore worstTime(n) is O(n log n). We needed to include the variable `m` because if the inner loop started with **while** `(n > 1)`, the outer loop would have terminated after just one iteration.

In Chapter 11, we will encounter several sorting algorithms whose worstTime(n) is O(n log n), where n is the number of items to be sorted.

Skeleton 5. worstTime(n) is O(n^2):

a.
```
for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)
     {
          S
     } // for j
```

The number of times that `S` is executed is n^2. That is all we need to know to conclude that worstTime(n) is O(n^2). In Example 3.3, we painstakingly counted the exact number of statements executed and came up with the same result.

b.
```
for (int i = 0; i < n; i++)
     for (int k = i; k < n; k++)
     {
          S
     } // for k
```

The number of times that `S` is executed is

$$n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \sum_{k=1}^{n} k$$

As shown in Example A2.1 of Appendix 2, the above sum is equal to

$$n(n+1)/2,$$

which is O(n^2). That is, worstTime(n) is O(n^2).
The `selectionSort` method, developed in Chapter 11, uses the above skeleton. Here, n refers to the size of the array to be sorted.

```
/**
 *  Sorts a specified array of int values into ascending order.
 *  The worstTime(n) is O(n * n).
 *
 *  @param x - the array to be sorted.
 *
 */
 public static void selectionSort (int [ ] x)
 {
        // Make x [0 ... i] sorted and <= x [i + 1] ...x [x.length -1]:
        for (int i = 0; i < x.length -1; i++)
        {
                int pos = i;
                for (int j = i + 1; j < x.length; j++)
                        if (x [j] < x [pos])
                                   pos = j;
                int temp = x [i];
                x [i] = x [pos];
                x [pos] = temp;
        } // for i
 } // method selectionSort
```

There are $n - 1$ iterations of the outer loop; when the smallest values are at indexes x [0], x [1], ... x [$n-2$], the largest value will automatically be at index x [$n-1$]. So the total number of inner-loop iterations is

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$$

We conclude that worstTime(n) is O(n^2).

c.
```
for (int i = 0; i < n; i++)
{
    S
} // for i
for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)
     {
         S
     } // for j
```

For the first segment, worstTime(n) is O(n), and for the second segment, worstTime(n) is O(n^2), so for both segments together, worstTime(n) is O($n + n^2$), which is equal to O(n^2). In general, for the sequence

A

B

if worstTime(n) is O(f) for A and worstTime(n) is O(g) for B, then worstTime(n) is O($f + g$) for the sequence A, B.

3.1.3 Big-Omega, Big-Theta and Plain English

In addition to Big-O notation, there are two other notations that you should know about: Big-Omega and Big-Theta. Whereas Big-O provides a crude upper bound for a function, Big-Omega supplies a crude lower bound. Here is the definition:

Let g be a function that has non-negative integer arguments and returns a non-negative value for all arguments. We define **Ω(*g*)** to be the set of functions f such that for some positive constant C and some non-negative constant K,

$$f(n) \geq C\, g(n) \quad \text{for all} \quad n \geq K.$$

If f is in $\Omega(g)$ we say that f is "Big-Omega of g". Notice that the definition of Big-Omega differs from the definition of Big-O only in that the last line has $f(n) \geq Cg(n)$ instead of $f(n) \leq Cg(n)$, as we had for Big-O.

All of the Big-O examples from Section 3.1.1 are also Big-Omega examples. Specifically, in Example 3.1, the function f defined by

$$f(n) = 2n^2 + 4n + 2, \quad \text{for} \quad n = 0, 1, 2, \ldots$$

is $\Omega(n^2)$: for $C = 2$ and $K = 0$, $f(n) > Cn^2$ for all $n \geq K$. Also, for all of the code skeletons and methods in Section 3.1.2, we can replace O with Ω as a bound on worstTime(n).

Big-Omega notation is used less frequently than Big-O notation because we are usually more interested in providing an upper bound than a lower bound for worstTime(n) or averageTime(n). That is, "can't be any worse than" is often more relevant than "can't be any better than." Occasionally, knowledge of a theoretical lower bound can guide those trying to come up with an optimal algorithm. And in Chapter 11, we establish the important result that for any comparison-based sort method, averageTime(n)—and therefore, worstTime(n)—is $\Omega(n \log n)$.

A somewhat artificial example shows that Big-O and Big-Omega are distinct. Let f be the function defined by

$$f(n) = n, \quad \text{for} \quad n = 0, 1, 2, \ldots$$

Clearly, f is O(n), and therefore, f is also O(n^2). But f is not $\Omega(n^2)$. And that same function f is clearly $\Omega(n)$, and therefore $\Omega(1)$. But f is not O(1). In fact, the Big-Omega hierarchy is just the reverse of the Big-O hierarchy in Figure 3.1. For example,

$$\Omega(n^2) \subset \Omega(n \log n) \subset \Omega(n) \subset \Omega(1)$$

In most cases the same function will serve as both a lower bound and an upper bound, and this leads us to the definition of Big-Theta:

Let g be a function that has non-negative integer arguments and returns a non-negative value for all arguments. We define $\boldsymbol{\Theta(g)}$ to be the set of functions f such that for some positive constants C_1 and C_2, and some non-negative constant K,

$$C_1\, g(n) \leq f(n) \leq C_2\, g(n) \quad \text{for all} \quad n \geq K.$$

The idea is that if f is $\Theta(g)$, then eventually (that is, for all $n \geq K$), $f(n)$ is bounded below by some constant times $g(n)$ and also bounded above by some constant times $g(n)$. In other words, to say that a function f is $\Theta(g)$ is exactly the same as saying that f is both $O(g)$ and $\Omega(g)$. When we establish that a function f is $\Theta(g)$, we have "nailed down" the function f in the sense that f is, roughly, bounded above by g and also bounded below by g.

As an example of Big-Theta, consider the function f defined by

$$f(n) = 2n^2 + 4n + 2, \text{ for } n = 0, 1, 2, \ldots$$

We showed in Example 3.3 that f is $O(n^2)$, and earlier in this section we showed that f is $\Omega(n^2)$. We conclude that f is $\Theta(n^2)$.

For ease of reading, we adopt plain-English terms instead of Big-Theta notation for several families of functions in the Big-Theta hierarchy. For example, if f is $\Theta(n)$, we say that f is "linear in n". Table 3.1 shows some English-language replacements for Big-Theta notation.

We prefer to use plain English (such as "constant," "linear," and "quadratic") whenever possible. But as we will see in Section 3.1.5, there will still be many occasions when all we specify is an upper bound—namely, Big O—estimate.

3.1.4 Growth Rates

In this section, we look at the growth rate of functions. Specifically, we are interested in how rapidly a function increases based on its Big-Theta classification. Suppose we have a method whose worstTime(n) is linear in n. Then we can write:

$$\text{worstTime}(n) \approx C\, n, \quad \text{for some constant } C \text{ (and for sufficiently large values of } n).$$

What will be the effect of doubling the size of the problem, that is, of doubling n?

$$\begin{aligned} \text{worstTime}(2n) &\approx C\ 2\ n \\ &= 2\ C\ n \\ &\approx 2\ \text{worstTime}(n) \end{aligned}$$

In other words, if we double n, we double the estimate of worst time.

Table 3.1 Some English-language equivalents to Big-Theta notation

Big-Theta	English
$\Theta(c)$, for some constant $c \geq 0$	constant
$\Theta(\log n)$	logarithmic in *n*
$\Theta(n)$	linear in *n*
$\Theta(n \log n)$	linear-logarithmic in *n*
$\Theta(n^2)$	quadratic in *n*

Similarly, if a method has worstTime(n) that is quadratic in n, we can write:

worstTime(n) $\approx C\, n^2$, for some constant C (and for sufficiently large values of n).

Then

$$\begin{aligned}\text{worstTime}(2n) &\approx C\ (2n)^2\\ &= C\ 4\ n^2\\ &= 4\ C\ n^2\\ &\approx 4\ \text{worstTime}(n)\end{aligned}$$

In other words, if we double n, we quadruple the estimate of worst time. Other examples of this kind of relationship are explored in Concept Exercise 3.7, Concept Exercise 11.5 and in later labs.

Figure 3.3 shows the relative growth rates of worstTime(n) for several families of functions.

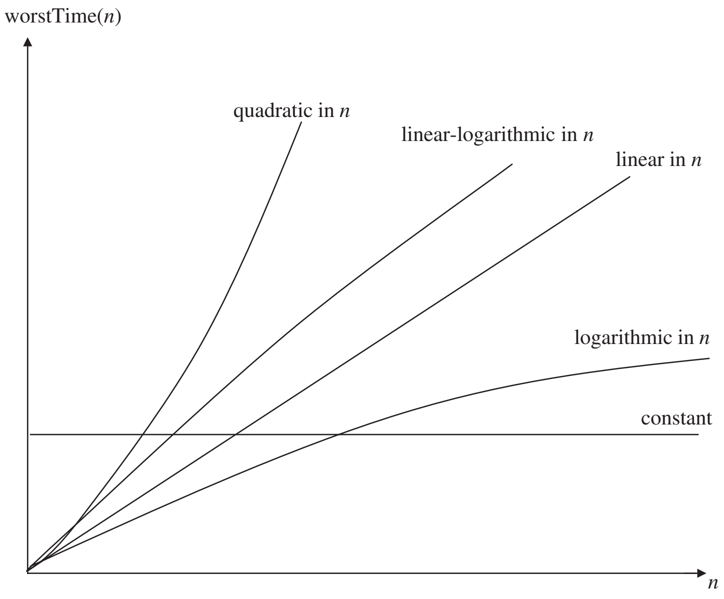


FIGURE 3.3 The graphs of worstTime(n) for several families of functions

Figure 3.4 indicates why Big-Theta differences eventually dominate constant factors in estimating the behavior of a function. For example, if n is sufficiently large, $t_1(n) = n^2/100$ is much greater than $t_2(n) = 100\ n\ \log_2 n$. But the phrase "if n is sufficiently large" should be viewed as a warning. Note that t_1 is *smaller* than t_2 for arguments less than 100,000. So whether Big-Theta (or Big-O or Big-Omega) is relevant may depend on how large the size of your problem might get.

Figure 3.4 has a concrete example of the differences between several families in the Big-Theta hierarchy. For a representative member of the family—expressed as a function of n—the time to execute that many statements is estimated when n equals one billion.

Some of the differences shown in Figure 3.4 are worth exploring. For example, there is a huge difference between the values of $\log_2 n$ and n. In Chapter 10, we will study a data structure—the binary search tree—for which averageTime(n) is logarithmic in n for inserting, removing, and searching, but worstTime(n) is linear in n for those methods.

Function of n	Time Estimate
$\log_2 n$	.0024 seconds
n	17 minutes
$n \log_2 n$	7 hours
n^2	300 years

FIGURE 3.4 Estimated time to execute a given number of statements for various functions of n when $n =$ 1,000,000,000 and 1,000,000 statements are executed per second. For example, to execute $n \log_2 n$ statements takes approximately 7 hours

Another notable comparison in Figure 3.4 is between $n \ \log_2 n$ and n^2. In Chapter 11, on sort methods, we will see tangible evidence of this difference. Roughly speaking, there are two categories of sort methods: fast sorts, whose averageTime(n) is linear-logarithmic in n, and simple sorts, whose averageTime(n) is quadratic in n.

All of the methods we have seen so far are polynomial-time methods. A ***polynomial-time*** method is one for which worstTime(n) is O(n^i) for some positive integer i. For example, a method whose worstTime(n) is (O n^2) is a polynomial-time method. Similarly, a method whose worstTime(n) is (O log n) is polynomial-time because (O log n) $\subset O$ (n).

When we try to develop a method to solve a given problem, we prefer polynomial-time methods whenever possible. For some methods, their run time is so long it is infeasible to run the methods for large values of n. Such methods are in the category of exponential-time methods. An ***exponential-time*** method is one whose worstTime(n) is $\Omega(x^n)$ for some real number $x > 0$. Then we say worstTime(n) is ***exponential in n***. For example, a method whose worstTime(n) is $\Omega(2^n)$ is an exponential-time method. Chapter 5 has an example of an exponential-time method, and Labs 7 and 9 have two more exponential-time methods. As you might expect, a polynomial-time method cannot also be exponential-time (Concept Exercise 3.10).

The existence of exponential-time methods gives rise to an interesting question: For a given exponential-time method, might there be a polynomial-time method to solve the same problem? In some cases, the answer is no. An ***intractable problem*** is one for which any method to solve the problem is an exponential-time method. For example, a problem that requires 2^n values to be printed is intractable because any method to solve that problem must execute at least $\Omega(2^n)$ statements. The problem in Chapter 5 for which an exponential-time method is supplied is an intractable problem. The problem in Lab 9 is also intractable, but the problem in Lab 7 has a polynomial-time solution.

Lab 23 investigates the Traveling Salesperson Problem, for which the only known methods to solve the problem are exponential-time methods. The most famous open question in computer science is whether the Traveling Salesperson Problem is intractable. There may be a polynomial-time method to solve that problem, but no one has found one, and most experts believe that no such method is possible.

If we are working on a single method only, it may be feasible to optimize that method's averageTime(n) and worstTime(n), with the intent of optimizing execution time. But for the management of an entire project, it is usually necessary to strike a balance. The next section explores the relevance of other factors, such as memory utilization and project deadlines.

3.1.5 Trade-Offs

In the previous section we saw how to estimate a method's execution-time requirements. The same Big-O (or Big-Omega or Big-Theta) notation can be used to estimate the memory requirements of a method. Ideally, we will be able to develop methods that are both fast enough *and* small enough. But in the real world, we seldom attain the ideal. More likely, we will encounter one or more of the following obstacles during programming:

1. The program's estimated execution time may be longer than acceptable according to the performance requirements. ***Performance requirements***, when given, state the time and space upper-bounds for all or part of a program.

2. The program's estimated memory requirements may be larger than acceptable according to the performance requirements. This situation frequently arises for hand-held devices.

3. The program may require understanding a technique about which the programmer is only vaguely familiar. This may create an unacceptable delay of the entire project.

Often, a trade-off must be made: a program that reduces one of the three obstacles may intensify the other two. For example, if you had to develop a project by tomorrow, you would probably ignore time and space constraints and focus on understanding the problem well enough to create a project. The point is that real-life programming involves hard decisions. It is not nearly enough that you can develop programs that run. Adapting to constraints such as those mentioned above will make you a better programmer by increasing your flexibility.

We can incorporate efficiency concerns into the correctness of a method by including performance requirements in the method's specification (but see Programming Exercise 3.5). For example, part of the specification for the `Quick Sort` method in Chapter 11 is:

> The worstTime (n) is $O(n^2)$.

Then for a definition of that method to be correct, worstTime(n) would have to be $O(n^2)$. Recall that the Big-O estimates provide *upper bounds* only. But the class developer is free to improve on the upper bounds for average time or worst time. For example, there is a way to define that sort method so that worstTime(n) is linear-logarithmic in n.

We want to allow developers of methods the flexibility to improve the efficiency of those methods without violating the contract between users and developers. So any performance requirements in method specifications will be given in terms of upper-bounds (that is, Big-O) only. Here are three conventions regarding the Big-O estimates in method specifications:

0. If a class stores a collection of elements, then unless otherwise noted, the variable n refers to the number of elements in the collection.

1. For many methods, worstTime(n) is $O(1)$. If no estimate of worstTime(n) is given, you may assume that worstTime(n) is $O(1)$.

2. Often, averageTime(n) has the same Big-O estimate as worstTime(n), and then we will specify the worstTime(n) estimate only. When they are different, we will specify both.

When we analyze the time (or space) efficiency of a specific method definition, we will determine lower as well as upper bounds, so we will use Big-Theta notation—or the English-language equivalent: constant, linear-in-n, and so on.

Up until now, we have separated concerns about correctness from concerns about efficiency. According to the Principle of Data Abstraction, the correctness of code that uses a class should be independent of that class's implementation details. But the efficiency of that code may well depend on those details. In other words, the developer of a class is free—for the sake of efficiency—to choose any combination of fields and method definitions, provided the correctness of the class's methods do not rely on those choices. For example, suppose a class developer can create three different versions of a class:

A: correct, inefficient, does not allow users to access fields;
B: correct, somewhat efficient, does not allow users to access fields;
C: correct, highly efficient, allows users to access fields.

In most cases, the appropriate choice is B. Choosing C would violate the Principle of Data Abstraction because the correctness of a program that uses C could depend on C's fields.

Big-O analysis provides a cross-platform estimate of the efficiency of a method. The following section explores an execution-time tool for measuring efficiency.

3.2 Run-Time Analysis

We have seen that Big-O notation allows us to estimate the efficiency of methods independently of any particular computing environment. For practical reasons, we may also want to estimate efficiency within some fixed environment. Why settle for estimates? For one thing,

> **In multi-programming environments such as Windows, it is very difficult to determine how long a single task takes**.

Why? Because there is so much going on behind the scenes, such as the maintaining the desktop clock, executing a wait-loop until a mouse click occurs, and updating information from your mailer and browser. At any given time, there might be dozens of such processes under control of the Windows Manager. And each process will get a time slice of several milliseconds. The bottom line is that the elapsed time for a task is seldom an accurate measure of how long the task took.

Another problem with seeking an exact measure of efficiency is that it might take a very long time—*O(forever)*. For example, suppose we are comparing two sorting methods, and we want to determine the average time each one takes to sort some collection of elements. The time may depend heavily on the particular arrangement of the elements chosen. Because the number of different arrangements of n distinct elements is $n!$, it is not feasible to generate every possible arrangement, run the method for each arrangement, and calculate the average time.

Instead, we will generate a sample ordering that is in "no particular order." The statistical concept corresponding to "no particular order" is randomness. We will use the time to sort a random sample as an estimate of the average sorting time. We start with a discussion of timing because, as we will see later, one aspect of randomness depends on the result of a timing method.

3.2.1 Timing

To assist in the timing of methods, Java supplies `nanoTime()`, a static method in the `System` class of java.lang. This method returns a **`long`** whose value is the number of nanoseconds—that is, billionths of a second—elapsed since some fixed but arbitrary time. To estimate how much execution time a task consumes, we calculate the time immediately before and immediately after the code for the task. The difference in the two times represents the elapsed time. As noted previously, elapsed time is a very, very crude estimate of the time the task consumed. The following code serves as a skeleton for estimating the time expended by a method:

```
final String ANSWER_1 = "The elapsed time was ";

final double NANO_FACTOR = 1000000000.0;  // nanoseconds per second
final String ANSWER_2 = " seconds.";

long startTime,
    finishTime,
    elapsedTime;

startTime = System.nanoTime();

// Perform the task:
...

// Calculate the elapsed time:
finishTime = System. nanoTime();
elapsedTime = finishTime - startTime;
System.out.println (ANSWER_1 + (elapsedTime / NANO_FACTOR) + ANSWER_2);
```

This skeleton determines the elapsed time for the task in seconds, with fractional digits. For example, if `startTime` has the value 885161724000 and `finishTime` has the value 889961724000, then `elapsedTime` has the value 4800000000, that is, four billion and eight hundred million. Then `elapsedTime/NANO_FACTOR` has the value 4.8 (seconds).

We will use the time to process a random sample of values as an estimate of the average processing time. Section 3.2.2 contains an introduction to—or review of—the `Random` class, part of the package java.util.

3.2.2 Overview of the `Random` Class

If each number in a sequence of numbers has the same chance of being selected, the sequence is said to be ***uniformly distributed***. A number so selected from a uniformly-distributed sequence is called a ***random number***. And a method that, when repeatedly called, returns a sequence of random numbers is called a ***random-number generator***.

The `Random` class in java.util supplies several random-number generators. We will look at three of those methods. Strictly speaking, the sequence of numbers returned by repeated calls to any one of those methods is a *pseudo-random-number* sequence because the numbers calculated are not random at all—they are determined by the code in the method. The numbers *appear* to be random if we do not see how they are calculated. If you look at the definition of this method in the `Random` class, the mystery and appearance of randomness will disappear.

Here is the method specification for one of the random-number generators:

```
/**
 *  Returns a pseudo-random int in the range from 0 (inclusive) to a specified int
 *  (exclusive).
 *
 *  @param n - the specified int, one more than the largest possible value
 *         returned.
 *
 *  @return a random int in the range from 0 to n -1, inclusive.
 *
 *  @throws IllegalArgumentException - if n is less than or equal to zero.
 *
 */
public int nextInt (int n)
```

For example, a call to `nextInt (100)` will return a random integer in the range from 0 to 99, inclusive.

For another example, suppose we want to simulate the roll of a die. The value from one roll of a die will be an integer in the range 1 . . . 6, inclusive. The call to `nextInt (6)` returns an **int** value in the range from 0 to 5, inclusive, so we need to add 1 to that returned value. Here is the code to print out that pseudo-random die roll:

```
Random die = new Random();

int oneRoll = die.nextInt (6) + 1;

System.out.println (oneRoll);
```

The value calculated by the `nextInt (int n)` method depends on the seed it is given. The variable `seed` is a **private long** field in the `Random` class. The initial value of `seed` depends on the constructor called. If, as above, the `Random` object is created with the default constructor, then `seed` is initialized to `System.nanoTime().` The other form of the constructor has a **long** parameter, and `seed` is initialized to the argument corresponding to that parameter. Each time the method `nextInt (int n)` is called, the current value of the seed is used to determine the next value of the seed, which determines the **int** returned by the method.

For example, suppose that two programs have

```
Random die = new Random (800);

for (int i = 0; i < 5; i++)
         System.out.println (die.nextInt (6) + 1);
```

The output from both programs would be exactly the same:

```
3
5
3
6
2
```

This repeatability can be helpful when we want to compare the behavior of programs, as we will in Chapters 5—15. In general, repeatability is an essential feature of the scientific method.

If we do not want repeatability, we use the default constructor. Recall that the default constructor initializes the seed to `System.nanoTime()`.

Here are two other random-number generators in the `Random` class:

```
/**
 *  Returns a pseudo-random int in the range from Integer.MIN_VALUE to
 *  Integer.MAX_VALUE.
 *
 *
 *  @return a pseudo-random int in the range from Integer.MIN_VALUE to
 *  Integer.MAX_VALUE.
 *
 */
 public int nextInt ()

/**
 *  Returns a pseudo-random double in the range from 0.0 (inclusive) to
 *  1.0 (exclusive).
 *
 */
 public double nextDouble ()
```

The following program combines randomness and timing with repeated calls to the `selectionSort` method of Section 3.1.2. The higher levels of the program—input, output, and exception handling—are handled in the `run( )` method. The `randomTimer` method generates an array of random integers, calls the `selectionSort` method, and calculates the elapsed time. The `randomTimer` method is not unit-tested because timing results will vary widely from computer to computer. The unit tests for `selectionSort` and other sort methods are in the Chapter 11 directory of the website's source code section.

```
import java.util.*;

public class TimingRandom
{
 public static void main (String[ ] args)
 {
   new TimingRandom().run();
 } // method main

 public void run()
 {
   final int SENTINEL = -1;

   final String INPUT_PROMPT = "\nPlease enter the number of"+
        " integers to be sorted (or " + SENTINEL + " to quit): ";

   final String ANSWER_1 = "The elapsed time was ";

   final double NANO_FACTOR = 1000000000.0; // nanoseconds per second

   final String ANSWER_2 = " seconds.";

   Scanner sc = new Scanner (System.in);

   long elapsedTime;

   while (true)
   {
     try
     {
        System.out.print (INPUT_PROMPT);
        int n = sc.nextInt();
        if (n == SENTINEL)
            break;
        elapsedTime = randomTimer (n);
        System.out.println (ANSWER_1 +
                    (elapsedTime / NANO_FACTOR) + ANSWER_2);
     } // try
     catch (Exception e)
     {
       System.out.println (e);
       sc.nextLine();
     } // catch
   } // while
} // method run

/**
 *  Determines the elapsed time to sort a randomly generated array of ints.
 *
 *  @param n – the size of the array to be generated and sorted.
 *
 *  @return the elapsed time to sort the randomly generated array.
 *
 *  @throws NegativeArraySizeException – if n is less than 0.
 *
 */
 public long randomTimer (int n)
 {
   Random r = new Random();

   long startTime,
        finishTime,
        elapsedTime;

   int[ ] x = new int [n];
   for (int i = 0; i < n; i++)
      x [i] = r.nextInt();
   startTime = System.nanoTime();

   // Sort x into ascending order:
   selectionSort (x);

   // Calculate the elapsed time:
   finishTime = System.nanoTime();
   elapsedTime = finishTime - startTime;
   return elapsedTime;
} // method randomTimer

/**
 *  Sorts a specified array of int values into ascending order.
 *  The worstTime(n) is O(n * n).
 *
 *  @param x – the array to be sorted.
 *
 */
 public static void selectionSort (int [ ] x)
 {
   // Make x [0 ... i] sorted and <= x [i + 1] ... x [x.length -1]:
   for (int i = 0; i < x.length - 1; i++)
   {
      int pos = i;
      for (int j = i + 1; j < x.length; j++)
         if (x [j] < x [pos])
             pos = j;
      int temp = x [i];
      x [i] = x [pos];
      x [pos] = temp;
    } // for i
  } // method selectionSort

 } // class TimingRandom
```

The number of iterations of the **while** loop is independent of n, so for the `run()` method, worstTime(n) is determined by the estimate of worstTime(n) for `randomTimer.` The randomTimer method has a loop to generate the array, and worstTime(n) for this generation is O(n). Then `randomTimer` calls `selection Sort`. In Section 3.1.2, we showed that worstTime(n) for `selectionSort` is O(n^2). Since the number of iterations is the same for any arrangement of the n elements, averageTime(n) is O(n^2). In fact, n^2 provides a crude lower bound as well as a crude upper bound, so averageTime(n) is quadratic in n. Then we expect the average run time—over all possible arrangements of n doubles—to be quadratic in n. As suggested in Section 3.2, we use the elapsed time to sort n pseudo-random doubles as an approximation of the average run time for all arrangements of n doubles.

The elapsed time gives further credence to that estimate: for $n = 50000$, the elapsed time is 19.985 seconds, and for $n = 100000$, the elapsed time is 80.766 seconds. The actual times are irrelevant since they depend on the computer used, but the relative times are significant: when n doubles, the elapsed time quadruples (approximately). According to Section 3.1.4 on growth rates, that ratio is symptomatic of quadratic time.

Randomness and timing are also combined in the experiment in Lab 4: You are given the unreadable (but runnable) bytecode versions of the classes instead of source code.

You are now prepared to do Lab 4:

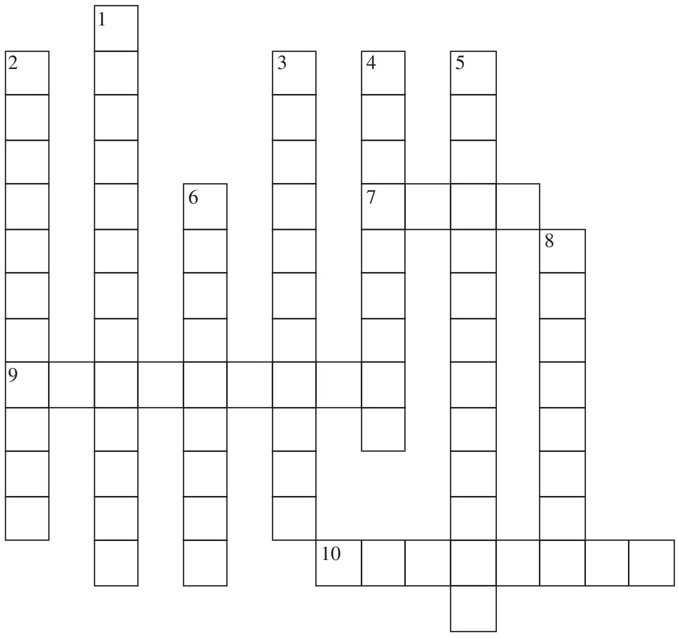
Randomness and Timing of Four Mystery Classes

SUMMARY

Big-O notation allows us to quickly estimate an upper bound on the time/space efficiency of methods. Because Big-O estimates allow function arguments to be arbitrarily large integers, we treat methods as algorithms by ignoring the space requirements imposed by Java and a particular computing environment. In addition to Big-O notation, we also looked at Big-Ω notation (for lower bounds) and Big-Θ notation (when the upper-bound and lower-bound are roughly the same).

Run-time analysis allows methods to be tested on a specific computer. But the estimates produced are often very crude, especially in a multiprogramming environment. Run-time tools include the `nanoTime()` method and several methods from the `Random` class.

CROSSWORD PUZZLE



www.CrosswordWeaver.com

ACROSS

7. The private long field in the `Random` class whose initial value depends on the constructor called.

9. A finite sequence of explicit instructions to solve a problem in a finite amount of time.

10. A function of *g* that is both Big O of *g* and Big Omega of *g* is said to be __________ of *g*.

DOWN

1. The rule that states "In general, if during each loop iteration, *n* is divided by some constant greater than 1, worstTime(*n*) will be O(log *n*) for that loop."

2. A problem for which for which any method to solve the problem is an exponential-time method is said to be _______________.

3. A method whose worstTime(*n*) is bounded below by *x* to the *n* for some real number *x* > 1.0 is said to be an ____________ time method.

4. A function of *n*, the problem size, that returns the maximum (over all possible parameter/input values) number of statements executed in a trace of the method.

5. A hallmark of the Scientific Method, and the reason we do not always want a random seed for the random number generator in the Random class.

6. A function of *n* that is Big Theta of *n*-squared is said to be __________ in *n*.

8. A static method in the System class that returns a long whose value is the number of billionths of a second elapsed since some fixed but arbitrary time.

CONCEPT EXERCISES

3.1 Create a method, `sample (int n)`, for which worstTime(n) is O(n) but worstTime(n) is not linear in n. **Hint:** O(n) indicates that n may be (crudely) viewed as an upper bound, but linear-in-n indicates that n may be (crudely) viewed as both an upper bound and a lower bound.

3.2 Study the following algorithm:

```
i = 0;
while (!a [i].equals (element))
    i++;
```

Assume that `a` is an array of `n` elements and that there is at least one index `k` in `0 ... n - 1` such that `a [k].equals (element)`.

Use Big-O notation to estimate worstTime(n). Use Big-Ω and Big-Θ notation to estimate worstTime(n). In plain English, estimate worstTime(n).

3.3 Study the following method:

```
/**
   *  Sorts a specified array of int values into ascending order.
   *  The worstTime(n) is O(n * n).
   *
   *  @param x - the array to be sorted.
   *
   */
  public static void selectionSort (int [ ] x)
  {
     // Make x [0 ... i] sorted and <= x [i + 1] ... x [x.length -1]:
    for (int i = 0; i < x.length - 1; i++)
    {
      int pos = i;
      for (int j = i + 1; j < x.length; j++)
         if (x [j] < x [pos])
             pos = j;
      int temp = x [i];
      x [i] = x [pos];
      x [pos] = temp;
    } // for i
  } // method selectionSort
```

a. For the inner **for** statement, when `i` $= 0$, `j` takes on values from 1 to `n - 1`, and so there are `n - 1` iterations of the inner **for** statement when `i` $= 0$. How many iterations are there when `i` $= 1$? When `i` $= 2$?

b. Determine, as a function of `n`, the total number of iterations of the inner **for** statement as `i` takes on values from 0 to `n - 2`.

c. Use Big-O notation to estimate worstTime(n). In plain English, estimate worstTime(n)—the choices are constant, logarithmic in n, linear in n, linear-logarithmic in n, quadratic in n and exponential in n.

3.4 For each of the following functions f, where $n = 0, 1, 2, 3, \ldots$, estimate f using Big-O notation and plain English:

a. $f(n) = (2 + n) * (3 + \log(n))$

b. $f(n) = 11 * \log(n) + n/2 - 3452$

c. $f(n) = 1 + 2 + 3 + \cdots + n$

d. $f(n) = n * (3 + n) - 7 * n$

e. $f(n) = 7 * n + (n - 1) * \log(n - 4)$

f. $f(n) = \log(n^2) + n$

g. $f(n) = \dfrac{(n+1) * \log(n+1) - (n+1) + 1}{n}$

h. $f(n) = \mathrm{n} + n/2 + n/4 + n/8 + n/16 + \cdots$

3.5 In the Order Hierarchy in Figure 3.1, we have ..., $O(\log n)$, $O(n^{1/2})$, ... Show that, for integers $n > 16$, $\log_2 n < n^{1/2}$. Hint from calculus: Show that for all real numbers $x > 16$, the slope of the function $\log_2 x$ is less than the slope of the function $x^{1/2}$. Since $\log_2(16) == 16^{1/2}$, we conclude that for all real numbers $x > 16$, $\log_2 x < x^{1/2}$.

3.6 For each of the following code segments, estimate worstTime(n) using Big Ω notation or plain English. In each segment, `S` represents a sequence of statements in which there are no n-dependent loops.

a.
```
for (int i = 0; i * i < n; i++)
       S
```

b.
```
for (int i = 0; Math.sqrt (i) < n; i++)
       S
```

c.
```
int k = 1;
for (int i = 0; i < n; i++)
     k *= 2;
for (int i = 0; i < k; i++)
     S
```

Hint: In each case, 2 is part of the answer.

3.7 **a.** Suppose we have a method whose worstTime(n) is linear in n. Estimate the effect of tripling n on run time—the actual time to execute the method in a particular computing environment. That is, estimate runTime($3n$) in terms of runTime(n).

b. Suppose we have a method whose worstTime(n) is quadratic in n. Estimate the effect of tripling n on run time—the actual time to execute the method in a particular computing environment. That is, estimate runTime($3n$) in terms of runTime(n).

c. Suppose we have a method whose worstTime(n) is constant. Estimate the effect of tripling n on run time—the actual time to execute the method in a particular computing environment. That is, estimate runTime($3n$) in terms of runTime(n).

3.8 This exercise proves that the Big-O families do not constitute a strict hierarchy. Consider the function f, defined for all non-negative integers as follows:

$$f(n) = \begin{cases} n, & \text{if } n \text{ is even;} \\ 0, & \text{if } n \text{ is odd} \end{cases}$$

Define a function g on all non-negative integers such that f is not $O(g)$ and g is not $O(f)$.

3.9 Show that $\mathrm{O}(n) = \mathrm{O}(n + 7)$. **Hint:** use the definition of Big-O.

3.10 Show that if $f(n)$ is polynomial in n, $f(n)$ cannot be exponential in n.

3.11 Suppose, for some method, worstTime(n) $= n^n$. Show that the method is an exponential-time method (that is, worstTime(n) is $\Omega(x^n)$ for some real number $x > 1.0$). But show that worstTime(n) is not $\Theta(x^n)$—that is, Big Theta of x^n—for any real number $x > 1.0$.

3.12 This exercise illustrates some anomalies of $\Theta(1)$.

a. Define $f(n)$ to be 0 for all $n \geq 0$. Show that f is not $\Theta(1)$, but f is $\Theta(0)$.

b. Define $f(n)$ to be $(n + 2)/(n + 1)$ for all $n \geq 0$. Show that f is $\Theta(1)$—and so can be said to be "constant"—even though f is not a constant function.

3.13 **a.** Assume that worstTime(n) $= C$ (statements) for some constant C and for all values of $n \geq 0$. Determine worstTime($2n$) in terms of worstTime(n).

b. Assume that worstTime(n) $= \log_2 n$ (statements) for all values of $n \geq 0$. Determine worstTime($2n$) in terms of worstTime(n).

c. Assume that worstTime(n) $= n$ (statements) for all values of $n \geq 0$. Determine worstTime($2n$) in terms of worstTime(n).

d. Assume that worstTime(n) $= n \log_2 n$ (statements) for all values of $n \geq 0$. Determine worstTime($2n$) in terms of worstTime(n).

e. Assume that worstTime(n) $= n^2$ (statements) for all values of $n \geq 0$. Determine worstTime($2n$) in terms of worstTime(n).

f. Assume that worstTime(n) $= 2^n$ (statements) for all values of $n \geq 0$. Determine worstTime($n + 1$) in terms of worstTime(n). Determine worstTime($2n$) in terms of worstTime(n).

3.14 If worstTime(n) is exponential in n for some method `sample`, which of the following must be true about that method?

a. worstTime(n) is $\mathrm{O}(2^n)$.

b. worstTime(n) is $\Omega(2^n)$.

c. worstTime(n) is $\Theta(2^n)$.

d. worstTime(n) is $\mathrm{O}(n^n)$.

e. none of the above.

PROGRAMMING EXERCISES

3.1 In mathematics, the absolute value function returns a non-negative integer for any integer argument. Develop a `run` method to show that the Java method `Math.abs (int a)` does not always return a non-negative integer.
Hint: See Programming Exercise 0.1.

3.2 Assume that `r` is (a reference to) an object in the `Random` class. Show that the value of the following expression is not necessarily in the range 0 . . . 99:

```
Math.abs (r.nextInt()) % 100
```

Hint: See Programming Exercise 3.1.

3.3 Develop a `run` method that initializes a `Random` object with the default constructor and then determines the elapsed time for the `nextInt()` method to generate 123456789.

3.4 Suppose a method specification includes a Big-O estimate for worstTime(n). Explain why it would be impossible to create a unit test to support the Big-O estimate.

3.5 In the `binarySearch` method in Section 3.1.2, the average of `low` and `high` was calculated by the following expression

```
(low + high) >> 1
```

Compare that expression to

```
low + ((high - low) >> 1)
```

The two expressions are mathematically equivalent, and the first expression is slightly more efficient, but will return an incorrect result for some values of `low` and `high`. Find values of `low` and `high` for which the first expression returns an incorrect value for the average of `low` and `high`. **Hint:** The largest possible **`int`** value is `Integer.MAX_VALUE`, approximately 2 billion.

Programming Project 3.1

Let's Make a Deal!

This project is based on the following modification—proposed by Marilyn Vos Savant—to the game show "Let's Make a Deal." A contestant is given a choice of three doors. Behind one door there is an expensive car; behind each of the other doors there is a goat.

After the contestant makes an initial guess, the announcer peeks behind the other two doors and eliminates one of them that does not have the car behind it. For example, if the initial guess is door 2 and the car is behind door 3, then the announcer will show that there is a goat behind door 1.

If the initial guess is correct, the announcer will randomly decide which of the other two doors to eliminate. For example, if the initial guess is door 2 and the car is behind door 2, the announcer will randomly decide whether to show a goat behind door 1 or a goat behind door 3. After the initial guess has been made and the announcer has eliminated one of the other doors, the contestant must then make the final choice.

Develop and test a program to determine the answer to the following questions:

1. Should the contestant stay with the initial guess, or switch?

2. How much more likely is it that an always-switching contestant will win instead of a never-switching contestant?

For the sake of repeatability, the following system tests used a seed of 100 for the random-number generator.

(continued on next page)

(continued from previous page)

System Test 1:

```
Please enter the number of times the game will be played: 10000


Please enter 0 for a never-switching contestant or 1 for always-switching: 0


The number of wins was 3330
```

System Test 2:

```
Please enter the number of times the game will be played: 10000


Please enter 0 for a never-switching contestant or 1 for always-switching: 1


The number of wins was 6628
```

Based on the output, what are your answers to the two questions given above?

Suppose, instead of working with three doors, the number of doors is input, along with the number of times the game will be played. Hypothesize how likely it is that the always-switching contestant will win. Modify and then run your project to confirm or reject your hypothesis. (Keep hypothesizing, and modifying and running your project until your hypothesis is confirmed.)

Hint for Hypothesis: Suppose the number of doors is n, where n can be any positive integer greater than 2. For an always-switching contestant to win, the initial guess must be incorrect, and then the final guess must be correct. What is the probability, with n doors, that the initial guess will be incorrect? Given that the initial guess is incorrect, how many doors will the always-switching contestant have to choose from for the final guess (remember that the announcer will eliminate one of those doors)? The probability that the always-switching contestant will win is the probability that the initial guess is incorrect times the probability that the final guess is then correct.

The Java Collections Framework

CHAPTER 4

The Java Collections Framework is an assortment of related interfaces and classes in the package `java.util`. For most of the classes in the Java Collections Framework, each instance is a collection, that is, each instance is composed of elements. The collection classes can have type parameters, a new feature of Java, so that a user can specify the type of the elements when declaring an instance of a collection class. In this chapter, we will take a brief tour of the Java Collection Framework's collection classes, along with the new features that enhance the utilization of those classes.

CHAPTER OBJECTIVES

1. Understand what a collection is, and how contiguous collections differ from linked collections.
2. Be able to create and manipulate parameterized collections.
3. Identify several of the methods in the `Collection` interface.
4. Describe a design pattern in general, and the iterator design pattern in particular.
5. Compare the `ArrayList` and `LinkedList` implementations of the `List` interface.
6. Be able to utilize boxing/unboxing and the enhanced **`for`** statement.

4.1 Collections

A ***collection*** is an object that is composed of elements. The elements can be either values in a primitive type (such as **`int`**) or references to objects. For a familiar example, an ***array*** is a collection of elements, of the same type, that are stored contiguously in memory. ***Contiguous*** means "adjacent," so the individual elements are stored next to each other[1]. For example, we can create an array of five `String` elements (strictly speaking, each element is a reference to a `String` object) as follows:

```
String [ ] names = new String [5];
```

Here the **`new`** operator allocates space for an array of five `String` references, (each initialized to **`null`** by the Java Virtual Machine), and returns a reference to the beginning of the space allocated. This reference is stored in `names`.

[1]Actually, all that matters is that, to a user of an array, the elements are stored *as if* they were contiguous, so an element can be accessed directly from its index.

There is an important consequence of the fact that arrays are stored contiguously: an individual element in an array can be accessed without first accessing any of the other individual elements. For example, `names [2]` can be accessed immediately—we need not access `names [0]` and `names [1]` first in order to reach `names [2]`. This ***random access*** property of arrays will come in handy in several subsequent chapters. In each case we will need a storage structure in which an element can be accessed quickly given its relative position, so an array will be appropriate in each case.

There are several drawbacks to arrays. First, the size of an array is fixed: Space for the entire array (of primitive values or references) must be allocated before any elements can be stored in the array. If that size is too small, a larger array must be allocated and the contents of the smaller array copied to the larger array.

Another problem with arrays is that the programmer must provide all the code for operating on an array. For example, inserting and deleting in an array may require that many elements be moved. Suppose an array's indexes range from 0 to 999, inclusive, and there are elements stored in order in the locations at indexes 0 to 755. To insert an element into the location with index 300, we must first move the elements at indexes 300 to 755 into the locations at indexes 301 to 756. Figure 4.1 shows the effect of such an insertion.

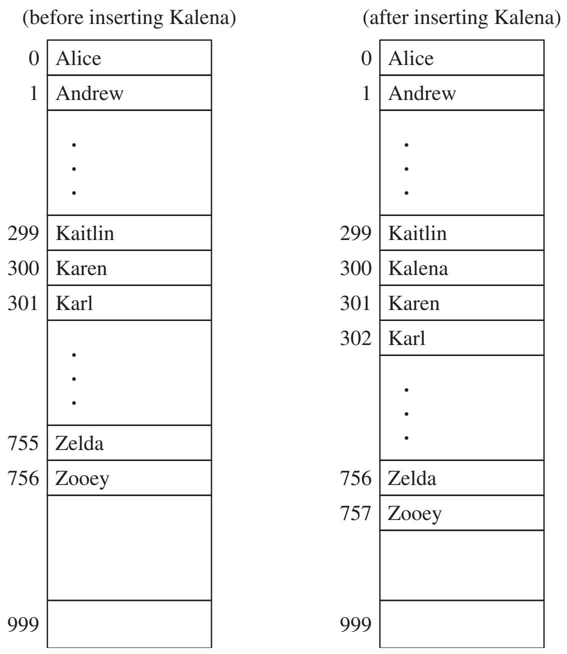


FIGURE 4.1 Insertion in an array: to insert "Kalena" at index 300 in the array on the left, the elements at indexes 300, 301, . . . , 756 must first be moved, respectively, to indexes 301, 302, . . . , 757

In your programming career up to now, you have had to put up with the above disadvantages of arrays. Section 4.1.1 describes an alternative that is almost always superior to arrays: instances of collection classes.

4.1.1 Collection Classes

Most of what we will do from here on involves collection classes. A ***collection class*** is a class in which each instance is a collection of elements, and each element is (a reference to) an object. For example, a `String` object can be an element, or a `FullTimeEmployee` object can be an element. Values in a primitive type are not objects, so we cannot create an instance of a collection class in which each element is of type **int**. But for each primitive type, there is a corresponding class, called a ***wrapper class***, whose purpose is to enable a primitive type to be represented by (that is, wrapped inside) a class. For example, there is an `Integer` class, and we can create an `Integer` object from an **int** `j` as follows:

```
new Integer (j)
```

The **new** operator returns a reference to an `Integer` object. Table 4.1 provides several important conversions.

Table 4.1 Some Important Conversion Formulas

```
int i;
Integer myInt;
String s;
Object obj;
```

TO OBTAIN	FROM	EXAMPLE
`Integer`	**int**	`myInt = i;  //see Section 4.2.2`
int	`Integer`	`i = myInt;  //see Section 4.2.2`
`String`	**int**	`s = Integer.toString(i);`
`String`	`Integer`	`s = myInt.toString();`
`Object`	`Integer`	`obj = myInt;  //by Subclass Substitution Rule`
`Object`	`String`	`obj = s;  //by Subclass Substitution Rule`
int	`String`	`i = new Integer (s);  //if s consists of an int`
`Integer`	`String`	`myInt = new Integer (s); // if s consists of an int`
`Integer`	`Object`	`myInt = (Integer)obj;//if obj references an Integer`
`String`	`Object`	`s = (String)obj;//if obj references a String`

The Java Collections Framework includes a number of collection classes that have wide applicability. All of those collection classes have some common methods. For example, each collection class has an `isEmpty` method whose method specification is:

```
/**
 *  Determines if this collection has no elements.
 *
 *  @return true - if this collection has no elements.
 *
 */
public boolean isEmpty()
```

Suppose `myList` is an instance of the collection class `ArrayList`, and `myList` has four elements. The execution of

```
System.out.println (myList.isEmpty());
```

will produce output of

```
false
```

Of course, a method specification does not indicate ***how*** the method's task will be accomplished. In subsequent chapters, we will investigate some of the details for several collection classes. But we can now introduce a simple classification of collection classes according to the way the elements are stored.

4.1.2 Storage Structures for Collection Classes

Instances of a collection class usually consume memory in proportion to the number of elements in the collection. So the way such a collection is stored in memory can have a substantial impact on the space efficiency of a program. One straightforward way to store a collection instance in memory is to store, in an array, a reference to each element in the collection. That is, an array could be a field in the collection class.

Such a class is called a ***contiguous-collection class***. For example, the `ArrayList` class in Chapter 6 has an array field, and (a reference to) each element in an `ArrayList` instance is stored in that instance's array. So `ArrayList` is a contiguous-collection class. We will study contiguous-collection classes in Chapters 6, 8 and 13. For many applications of contiguous-collection classes, the random-access feature of an array is a great asset.

What about the disadvantages, cited earlier, of an array: the size of an array is fixed, and the programmer is responsible for writing all the code that works with the array? With a contiguous-collection class, those are problems for the developer of the class, *not* for users of the class. Basically, the developer of a contiguous collection class writes the code—once—for methods that manipulate the array. Any user of that collection class simply invokes the appropriate methods for the given application. The user may not even be aware that there is an array field in the class, and by the Principle of Data Abstraction, would not rely on that field anyway.

You probably have not appreciated the random access feature of arrays. That's because you have probably not yet seen an alternative to arrays for storing a collection of elements in memory. We now briefly describe a structure that competes with the array for storing the elements in a collection object.

Instead of a contiguous relationship, the elements are related by links. A ***link*** is another name for a reference. Basically, each element is housed in a special object called an ***entry*** (sometimes called a ***node***). Within each entry object there will be at least one link to another entry object. In a linked-collection class, the elements in each instance are stored in entries. Figures 4.2–4.4 show parts of three linked collections. We will explore linked collections in Chapters 7, 10, 12, 14 and 15.

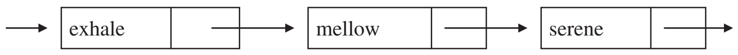


FIGURE 4.2 Part of a linked collection—a ***singly-linked list***—in which each entry contains an element and a reference to the next entry in the linked collection

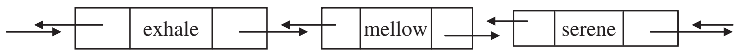


FIGURE 4.3 Part of a linked collection—a ***doubly-linked list***—in which each entry contains an element, a reference to the previous entry and a reference to the next entry

4.2 Some Details of the Java Collections Framework

In this section we present a little more information about the Java Collections Framework. The Java Collections Framework consists of a thoroughly tested assortment of interfaces and classes. The classes represent widely used data structures and algorithms. For most applications in which a collection is needed, the framework provides the appropriate class. By utilizing the framework classes, you improve your productivity by not "re-inventing the wheel."

One of the impediments to understanding the framework is its sheer size; over 200 methods in the eight classes we will study. Fortunately, there is a lot of duplication. For example, as noted in Section 4.1.1,

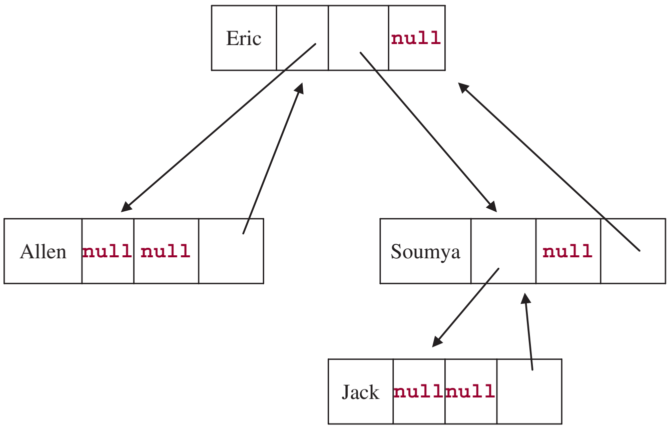


FIGURE 4.4 Part of a linked collection—a ***binary search tree***—in which each entry contains an element and references to three other entries

each of those classes has an `isEmpty` method. In fact, the definitions of many of the methods are the same in several classes. One of the unifying tools in the framework is the interface, which imposes method headings on implementing classes. Section 4.2.1 introduces another, similar unifying tool: the abstract class.

4.2.1 Abstract Classes

An ***abstract class*** is a class that is allowed to have abstract methods as well as defined methods. The abstract methods *must* be defined in each subclass (unless the subclass is also abstract). Here is a bare-bones example of an abstract class:

```
public abstract class Parent
{
      /**
       *  Returns the String object "I am".
       *
       *  @returns "I am".
       *
       */
       public String getPrefix()
       {
            return "I am";
       } // method getPrefix

      /**
       *  Returns a String object.
       *
       *  @return a String object.
       *
       */
       public abstract String getClassName();

} // class Parent
```

An abstract class is denoted by the **abstract** modifier at the beginning of its declaration. And within an abstract class, each abstract method's heading must include the modifier **abstract** before the return type, and a semicolon after the method heading. Because the `Parent` class lacks a definition for one of its methods, we cannot instantiate the `Parent` class. That is, we cannot define a `Parent` object:

```
Parent p = new Parent ();  // illegal because Parent is an abstract class
```

We can now declare two subclasses, `Child1` and `Child2`, of `Parent`.

```
public class Child1 extends Parent
{
      /**
       *  Returns the String object "Child1".
       *
       *  @return the String object "Child1".
       *
       */
       public String getClassName()
       {
             return "Child1";
       } // method getClassName

} // class Child1

public class Child2 extends Parent
{
      /**
       *  Returns the String object "Child2".
       *
       *  @return the String object "Child2".
       *
       */
       public String getClassName()
       {
             return "Child2";
       } // method getClassName

} // class Child2
```

The main benefit of abstract methods is that they promote flexibility (defined methods may be, but need not be, overridden in subclasses) and consistency (abstract-class headings must be identical in subclasses). For example, we can now do the following:

```
Parent p;

int code;

// Get the value for code;
...

if (code == 1)
      p = new Child1();
else
```

```
        p = new Child2();
    System.out.println (p.getPrefix() + p.getClassName());
```

The variable `p` is a polymorphic reference, so the version of `getClassName` called depends on the type—`Child1` or `Child2`—of the object referenced by `p`. The output will be "I am Child1" or "I am Child2", depending on the value of the variable `code`.

The Java Collections Framework has quite a few abstract classes: `AbstractCollection`, `AbstractList`, `AbstractSet`, and others. Typically, one of these classes will declare as abstract any method whose definition depends on fields in the subclasses, and define any method whose definition does not depend on those fields.

For now, a practical application of abstract classes is developed in Lab 5.

You are now prepared to do Lab 5: A Class for Regular Polygons

Here are a few more details on the relationship between interfaces, abstract classes and fully defined classes:

1. If a class implements some but not all of the methods in an interface, then the class would have to be declared as an abstract class—and therefore cannot be instantiated.
2. An interface can extend one or more other interfaces. For example, we could have:

   ```
   public interface Container extends Collection, Comparable
   {...
   ```

   `Container` has abstract methods of its own, and also inherits abstract methods from the interfaces `Collection` and `Comparable`.
3. A class can extend at most one other class; by default, the `Object` class is the superclass of every class. ***Multiple inheritance***—the ability of a class to have more than one immediate superclass—is illegal in Java. Multiple inheritance is illegal because of the danger of ambiguity. For example, viewing a teaching assistant as both a student and an employee, we could have a `TeachingAssistant` class that is the immediate subclass of classes `Student` and `StaffMember`. Now suppose classes `Student` and `StaffMember` each has its own `getHolidays()` method. If we define:

   ```
   TeachingAssistant teacher = new TeachingAssistant();
   ```

   which `getHolidays()` method does `teacher.getHolidays()` invoke? There is no way to tell, and that is why Java outlaws multiple inheritance. C++ allows multiple inheritance, but complex rules and disambiguating language are needed to make it work.
4. A class can implement more than one interface. For example, we could have:

   ```
   class NewClass implements Interface1, Interface2
   {...
   ```

This feature, especially when combined with feature 3, allows us to come close to achieving multiple inheritance. We can write:

```
class NewClass extends OldClass implements Interface1, Interface2
{
```

There is no ambiguity when a method is invoked because any methods in an interface are abstract, and any non-**final** superclass method can be explicitly ***overridden***—that is, re-defined—in the subclass. For example, suppose OldClass, Interface1, and Interface2 all have a writeOut() method, and we have

```
NewClass myStuff = new NewClass();

...

myStuff.writeOut();
```

Which version of the writeOut method will be invoked? Certainly not the version from Interface1 or Interface2, because those methods must be abstract. If NewClass implements a writeOut() method, that is the one that will be invoked. Otherwise, the version of writeOut defined in (or inherited by) OldClass will be invoked.

4.2.2 Parameterized Types

When collection classes were introduced in Section 4.1.1, we noted that the element type has to be a reference type: primitive types are not allowed. Starting with J2SE (that is, Java 2 Platform, Standard Edition) version 5.0, a class's element type can be specified, in angle brackets, when an instance of the class is declared. For example, suppose we want to declare and initialize an ArrayList object to hold a collection of grade point averages in which each grade point average will be stored as a Double. You don't have to know the details of the ArrayList class: You will learn some of those in Chapter 6. The declaration and initialization of the ArrayList object is as follows:

```
ArrayList <Double> gpaList = new ArrayList <Double>();
```

Only elements of type Double can be inserted into gpaList; an attempt to insert a String or Integer element will be disallowed by the compiler. As a result, you can be certain that any element retrieved from gpaList will be of type Double.

Let's see how elements can be inserted and retrieved from gpaList. In the ArrayList class, the add method inserts the element argument at the end of the ArrayList object. For example,

```
gpaList.add (new Double (2.7));
```

will append to the end of gpaList a (reference to a) Double object whose **double** value is 2.7.

For retrievals, the get method returns the element in the ArrayList object at a specified index. So we can access the element at index 0 as follows:

```
Double gpa = gpaList.get (0);
```

Notice that we don't need to cast the expression on the right-hand side to Double because the element at index 0 of gpaList must be of type Double.

Now suppose we want to add that grade point average to a **double** variable sum, initialized to 0.0. The method doubleValue() in the Double class returns the **double** value corresponding to the calling Double object. The assignment to sum is

```
sum = sum + gpa.doubleValue();
```

In this example, ArrayList<Double> is a parameterized type. A ***parameterized type*** consists of a class or interface identifier followed, in angle brackets, by a list of one or more class identifiers separated by commas. Typically, a parameterized type starts with a collection-class identifier, and the element type is enclosed in angle brackets. A parameterized type is sometimes called a “generic type”, and the language feature permitting parameterized types is called “generics”.

Parameterized collection classes improve your productivity as a programmer. You don’t have to remember what the element type of a collection is, because that type is specified when the collection is declared, as we did with `ArrayList<Double>`. If you make a mistake and try to insert an element of type `String` for example, you will be notified at compile-time. Without parameterized types, the insertion would be allowed, but the assignment of `(Double)gpaList.get(0)` to `gpa` would generate a `ClassCastException` at run time. And this exception, if uncaught, could crash a critical program.

In the previous example, the conversions from **`double`** to `Double` and from `Double` to **`double`** are annoyances. To simplify your working with parameterized collection classes, the Java compiler automatically translates primitive values into wrapper objects: the technical term is ***boxing***. For example, the insertion into `gpaList` can be accomplished as follows:

```
gpaList.add (2.7);              // instead of gpaList.add (new Double (2.7));
```

Unboxing translates a wrapper object into its primitive value. For example, to increment the above **`double`** variable `sum` by the value of the `Double` object `gpa`, we simply write

```
sum = sum + gpa;  // instead of sum = sum + gpa.doubleValue();
```

Unboxing eliminates the need for you to invoke the `doubleValue()` method, and that makes your code easier to read.

The general idea behind parameterized types and boxing/unboxing is to simplify the programmer’s work by assigning to the compiler several tasks that would otherwise have to be performed by the programmer.

Section 4.2.3 introduces the backbone of the Java Collections Framework: the `Collection` interface.

4.2.3 The `Collection` Interface

The Java Collections Framework consists basically of a hierarchy. There are interfaces and abstract classes at every level except the lowest, and the lowest level has implementations of interfaces and extensions of abstract classes. At the top of the hierarchy are two interfaces, `Collection` and `Map`.

In this section, we will focus on the `Collection` interface. For the sake of specificity, Figure 4.5 presents the `Collection` interface in UML notation, with the methods listed in alphabetical order. Don’t worry if some of the method headings are puzzling to you (or make no sense at all). You will learn all you will need to know in subsequent chapters, when we look at implementations of the interface.

As indicated in Figure 4.5, the `Collection` interface has `E`—for “element”—as the ***type parameter***. That is, `E` is replaced with an actual class, such as `Double` or `FullTimeEmployee`, in the declaration of an instance of any class that implements the interface. For example, part of the `ArrayList` heading is

```
public class ArrayList <E> implements Collection<E> ...
```

Here is an instance of the `ArrayList` class with `FullTimeEmployee` elements:

```
ArrayList<FullTimeEmployee>employeeList = new ArrayList <FullTimeEmployee>();
```

In this example, `FullTimeEmployee` is the actual class of the elements: the class that replaces the type parameter `E` when the `ArrayList` class is instantiated.

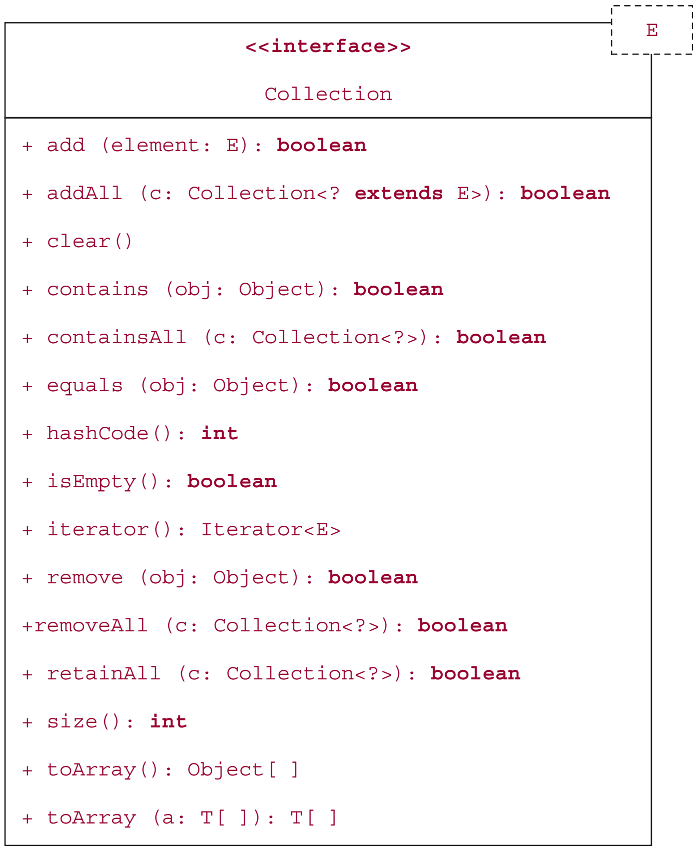

E
<<interface>>
Collection
+ add (element: E): boolean
+ addAll (c: Collection<? extends E>): boolean
+ clear()
+ contains (obj: Object): boolean
+ containsAll (c: Collection<?>): boolean
+ equals (obj: Object): boolean
+ hashCode(): int
+ isEmpty(): boolean
+ iterator(): Iterator<E>
+ remove (obj: Object): boolean
+removeAll (c: Collection<?>): boolean
+ retainAll (c: Collection<?>): boolean
+ size(): int
+ toArray(): Object[ ]
+ toArray (a: T[ ]): T[ ]

FIGURE 4.5 The `Collection` interface. In UML, a type parameter—in this case, `E`—is shown in a dashed rectangle in the upper-right-hand corner of the interface or class

If you wanted to, you could create your own class that fully implements the `Collection` interface. That is, sort of, what happens in Lab 6. Only a few methods are realistically defined; the others just throw an exception. For example,

```
public int hashCode()
{
          throw new UnsupportedOperationException();
}
```

Such definitions satisfy the compiler, so the resulting class, `ArrayCollection`, is instantiable. That is, we can create and initialize an `ArrayCollection` object:

```
ArrayCollection<Integer> collection = new ArrayCollection<Integer>();
```

You are now prepared to do Lab 6: The `ArrayCollection` Class

4.2.3.1 Iterators

The Collection interface provides a core of methods useful for applications. But each application will almost certainly have some specialized tasks that do not correspond to any method in the Collection interface. ***Important Note***: In the following examples, "Collection object" is shorthand for "object in a class that implements the Collection interface", and "Collection class" is shorthand for "class that implements the Collection interface."

1. Given a Collection object of students, print out each student who made the Dean's List.
2. Given a Collection object of words, determine how many are four-letter words.
3. Given a Collection object of club members, update the dues owed for each member.
4. Given a Collection object of full-time employees, calculate the average salary of the employees.

Surely, we cannot create a class that would provide a method for any task in any application—the number of methods would be limitless. But notice that in each of the four examples above, the task entails accessing each element in a Collection object. This suggests that we need to allow users of a Collection class to be able to construct a loop that accesses each element in a Collection object. As we will see when we look at classes that implement the Collection interface, *developers* can straightforwardly construct such a loop. Why? Because a developer has access to the fields in the class, so the developer knows how the class is organized. And that enables the developer to loop through each element in the instance.

According to the Principle of Data Abstraction, a user's code should not access the implementation details of a Collection class. The basic problem is this: How can any implementation of the Collection interface allow users to loop through the elements in an instance of that class without violating the Principle of Data Abstraction? The solution is in the use of iterators. ***Iterators*** are objects that allow the elements of Collection objects to be accessed in a consistent way without accessing the fields of the Collection class.

Inside each class that implements the Collection interface, there is an iterator class that allows a user to access each element in the collection. Each iterator class must itself implement the following Iterator interface:

```
public interface Iterator<E>
{
        /**
         *  Determines if this Iterator object is positioned at an element in
         *  this Collection object.
         *
         *  @return true - if this Iterator object is positioned at an element
         *          in this Collection object.
         *
         */
        boolean hasNext ();

        /**
         *  Advances this Iterator object, and returns the element this
         *  Iterator object was positioned at before this call.
         *
         *  @return the element this Iterator object was positioned at when
         *          this call was made.
         *
         *  @throws NoSuchElementException - if this Iterator object is not
         *          positioned at an element in the Collection object.
         *
         */
         E next ();

        /**
         *  Removes the element returned by the most recent call to next().
         *  The behavior of this Iterator object is unspecified if the underlying
         *  collection is modified - while this iteration is in progress - other
         *  than by calling this remove() method.
         *
         *  @throws IllegalStateException - if next() had not been called
         *          before this call to remove(), or if there had been an
         *          intervening call  to remove() between the most recent
         *          call to next() and this call.
         *
         void remove ();

} // interface Iterator<E>
```

For each class that implements the `Collection` interface, its iterator class provides the methods for traversing any instance of that `Collection` class. In other words, iterators are the behind-the-scenes workhorses that enable a user to access each element in any instance of a `Collection` class.

How can we associate an iterator object with a `Collection` object? The `iterator()` method in the `Collection` class creates the necessary connection. Here is the method specification from the `Collection` interface:

```
    /**
     *  Returns an Iterator object over this Collection object.
     *
     *  @return an Iterator object over this Collection object.
     *
     */
    Iterator<E> iterator( );
```

The value returned is (a reference to) an `Iterator` object, that is, an object in a class that implements the `Iterator` interface. With the help of this method, a user can iterate through a `Collection` object For example, suppose that `myColl` is (a reference to) an instance of a `Collection` object with `String` elements, and we want to print out each element in `myColl` that starts with the letter ‘a’. We first create an iterator object:

```
    Iterator<String> itr = myColl.iterator();
```

The variable `itr` is a polymorphic reference: it can be assigned a reference to an object in any class that implements the `Iterator<String>` interface. And `myColl.iterator()` returns a reference to an `Iterator<String>` object that is positioned at the beginning of the `myColl` object.

The actual iteration is fairly straightforward:

```
String word;
while (itr.hasNext ())
{
      word = itr.next();
      if (word.charAt (0) == 'a')
           System.out.println (word);
} // while
```

Incidentally, do you see what is wrong with the following?

```
// Incorrect!
while (itr.hasNext ())
      if (itr.next().charAt (0) == 'a')
                System.out.println (itr.next());
```

Because of the two calls to `itr.next()`, if the next word returned during a loop iteration starts with the letter 'a', the word *after* that word will be printed.

Very often, all we want to do during an iteration is to access each element in the collection. For such situations, Java provides an ***enhanced*** **`for`** statement (sometimes referred to as a ***for-each*** statement). For example, the previous (correct) iteration through `myColl` can be abbreviated to the following:

```
for (String word : myColl)
      if (word.charAt (0) == 'a')
              System.out.println (word);
```

The colon should be interpreted as "in", so the control part of this for statement can be read "For each `word` in `myColl`." The effect of this code is the same as before, but some of the drudgery—creating and initializing the iterator, and invoking the `hasNext()` and `next()` methods—has been relegated to the compiler.

Here is a complete example of iterating over a `Collection` object by using an enhanced **`for`** statement. You don't have to know the details of `ArrayList` class, the particular implementation of the `Collection` interface. You will learn those details in Chapter 6. For the sake of simplicity, `ArithmeticException` and `InputMismatchException` are caught in the same **`catch`** block.

```
// Calculates the mean grade-point-average

import java.util.*;

public class EnhancedFor
{
  public static void main (String [ ] args)
  {
    new EnhancedFor().run();
  } // method main

  public void run()
  {
    final double MIN_GPA = 0.0,
                 MAX_GPA = 4.0,
                 SENTINEL = -1.0;

    final String INPUT_PROMPT = "Please enter a GPA in the range" +
       " from " + MIN_GPA + " to " + MAX_GPA + ", inclusive (or " +
       SENTINEL + " to quit): ";

    final String RANGE_ERROR = "The grade point average must" +
       " be at least " + MIN_GPA + " and at most " + MAX_GPA + ".";

    final String MESSAGE = "\n\nThe mean GPA is ";

    final String NO_VALID_INPUT = "\n\nError: there were no valid " +
       "grade-point-averages in the input.";

    ArrayList<Double> gpaList = new ArrayList<Double>();

    Scanner sc = new Scanner (System.in);

    double oneGPA,
           sum = 0.0;

    while (true)
    {
      try
      {
         System.out.print (INPUT_PROMPT);
         oneGPA = sc.nextDouble();
         if (oneGPA == SENTINEL)
            break;
         if (oneGPA < MIN_GPA || oneGPA > MAX_GPA)
            throw new ArithmeticException (RANGE_ERROR);
         gpaList.add (oneGPA); // inserts at end of gpaList
      } // try
      catch (Exception e)
      {
        System.out.println (e + "\n");
        sc.nextLine();
      } // catch Exception
   } // while
   for (Double gpa : gpaList)
       sum += gpa;
   if (gpaList.size() > 0)
       System.out.println (MESSAGE +
                   (sum / gpaList.size()));
   else
      System.out.println(NO_VALID_INPUT);
  }  // method run

} // class EnhancedFor
```

The enhanced **for** statement simplifies your code, and that makes your programs easier to understand. So you should use an enhanced **for** statement whenever possible, that is, if you were to use an iterator instead, the only iterator methods invoked would be hasNext() and next(). You cannot use an enhanced **for** statement if the collection may be modified during the iteration. For example, if you wanted to delete, from gpaList, each grade-point-average below 1.0, you would need to explicitly set up an iterator:

```
Iterator<Double> itr = gpaList.iterator();
while (itr.hasNext())
       if (itr.next() < 1.0)
               itr.remove();
```

4.2.3.2 Design Patterns

In Section 4.2.3.1, we stated a problem, namely, how can the developer of a Collection class allow users to loop through one of its instances without violating the Principle of Data Abstraction? The solution to the problem was to employ an iterator. As such, the use of iterators is an example of a ***design pattern***: a generic programming technique that can be applied in a variety of situations. As we will see in subsequent chapters, the iterator pattern plays an important role in an assortment of applications.

Throughout the text, we will identify several design patterns and corresponding applications. The basic idea is that each design pattern provides you with a problem that occurs frequently and the outline of a solution. You may have to "tweak" the solution for a particular instance of the problem, but at least you will not be re-inventing the wheel.

In Section 4.2.4, we briefly introduce an extension of the Collection interface and three classes that implement that extension.

4.2.4 The List Interface

Java Collection Framework's List interface extends the Collection interface by providing some index-related methods. For example, there is a get method that returns the element at a given index. In any List object, that is, in any instance of a class that implements the List interface, the elements are stored in sequence, according to an index. For example, a List object pets might have the elements arranged as follows: "dog", "cat", "iguana", "gerbil", "cat". Here "dog" is at index 0, "gerbil" is at index 3. Duplicate elements are allowed: "cat" appears at index 1 and at index 4.

When viewed as a language-independent entity, a list is an abstract data type. Within Java, the List interface is abstract in the sense that it is not tied down to any particular implementation. In fact, in the Java Collections Framework, the List interface is not directly implemented. Instead, the abstract class AbstractList partially implements the List interface, and leaves the rest of the implementation to subclasses, namely, ArrayList and LinkedList. See Figure 4.6.

The ArrayList class implements the List interface with an underlying array[2], and the LinkedList class implements the List interface with the underlying linked structure shown in Figure 4.3. We will get to the details in Chapters 6 and 7, respectively. To give you an idea of some of the methods in both classes, the following class creates and manipulates a List of random Integer objects.

```
import java.util.*;

public class RandomList
```

[2]The Stack class also implements the List interface with an underlying array, but the definition of a stack severely restricts access to the array, so we will ignore the Stack class in this discussion.

```
{
  public static void main (String[ ] args)
  {
      new RandomList ().run();
  } // method main

  public void run()
  {
     final int SEED = 111;

     List<Integer> randList = new ArrayList<Integer>();

     Random r = new Random (SEED);

     // Insert 10 random integers, in the range 0...99, into randList:
     for (int i = 0; i < 10; i++)
             randList.add (r.nextInt(100));   // insertion

     // Print out randList:
     System.out.println (randList);

     // See if 22 is in randList:
     if (randList.contains (22))
             System.out.println ("Yes, 22 is in randList.");
     else
             System.out.println ("No, 22 is not in randList.");

     // Print out the Integer at index 3:
     System.out.println (randList.get (3) + "is at index 3");

     // Remove the Integer at index 6:
     randList.remove (6);

     // Insert a new random Integer at index 5:
     randList.add (5, r.nextInt (100));

     // Print out randList.
     System.out.println (randList);

     // Remove all even Integers:
     Iterator<Integer> itr = randList.iterator();
     while (itr.hasNext())
            if (itr.next() % 2 == 0)
                       itr.remove();

     // Print out randList;
     System.out.println (randList);
  } // method run

} // class RandomList
```

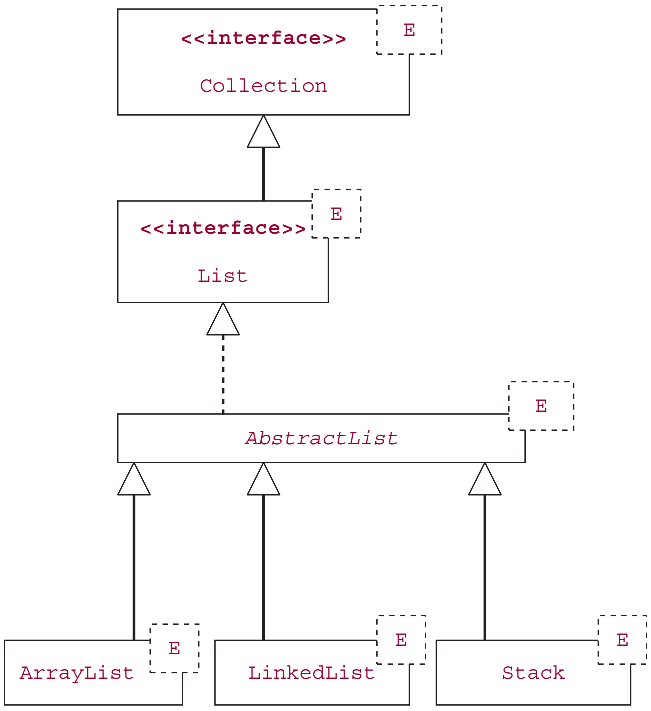


FIGURE 4.6 Part of the Java Collections Framework hierarchy dealing with the `List` interface. In UML, an abstract-class identifier is italicized

The line

```
System.out.println (randList);
```

is equivalent to

```
System.out.println (randList.toString());
```

The `toString` method returns a `String` representation of `randList`. Every class in the Java Collections Framework has a `toString()` method, so all the elements in an instance of one of those classes can be output with a single call to `println`.

Because an `ArrayList` object stores its elements in an underlying array, when the element at index 6 is removed, each element at a higher index is moved to the location at the next lower index. So the element that was at index 7 is then at index 6, the element that was at index 8 is then at index 7, and so on. When a new element is inserted at index 5, each element located at that index or higher is moved to the next higher index. So the element that was at index 5 is then at index 6, the element that was at index 6 is then at index 7, and so on.

The output is

```
[93, 70, 57, 97, 9, 20, 84, 12, 97, 65]
No, 22 is not in randList.
97 is at index 3
```

```
[93, 70, 57, 97, 9, 60, 20, 12, 97, 65]
[93, 57, 97, 9, 97, 65]
```

We could not use an enhanced `for` statement to iterate over `randList` because we needed to remove some of that object's elements, not merely access them.

In the program, `randList` is declared as a polymorphic reference and then immediately initialized as a reference to an `ArrayList` object. To re-run the program with a `LinkedList` object, the only change is the constructor call:

```
List<Integer> randList = new LinkedList<Integer>();
```

How do the two versions compare? Part of the program—printing the `Integer` at index 3—is executed more quickly with an `ArrayList` object because of the random-access ability of the underlying array. And part of it—removing all even `Integer` elements—is executed more quickly with a `LinkedList` object. That's because an entry in a linked list can be removed by adjusting links: no movement of elements is needed. In general, there is no "best" implementation of the `List` interface.

SUMMARY

A ***collection*** is an object that is composed of elements. The elements may be stored ***contiguously***, that is, at consecutive locations in memory. Another option is a ***linked*** structure, in which each element is stored in a special object called an ***entry*** that also includes a reference to another entry.

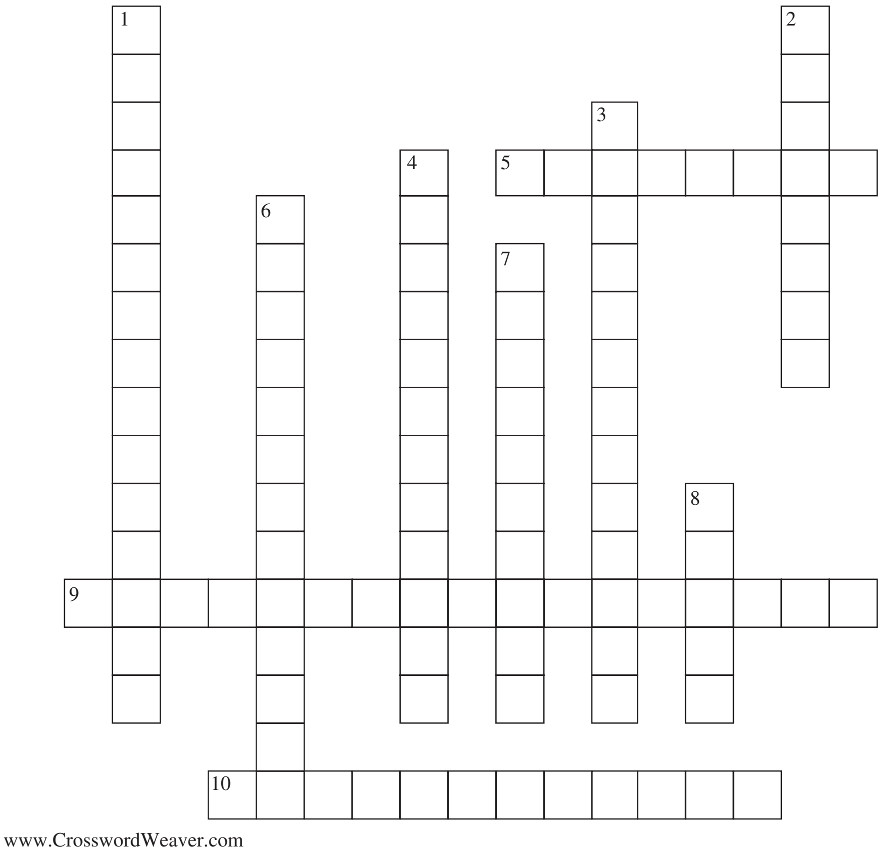
A ***collection class*** is a class of which each instance is a collection. The Java Collections Framework, part of the package java.util, includes a number of collection classes that have wide applicability. Each of those classes can be ***parameterized***, which means that the element class is specified when the collection-class object is created. And for any instance of one of those classes, an iterator can be defined. An ***iterator*** is an object that allows an instance of a collection class to loop through the elements in that class without violating the Principle of Data Abstraction.

To simplify the programmer's work of inserting elements into an instance of a parameterized class, Java automatically boxes primitive values into the corresponding wrapper elements. Similarly, wrapper elements retrieved from a parameter-class instance are automatically unboxed into the corresponding primitive value. A further simplification of Java is the ***enhanced*** **`for`** statement, which automates most of the routine code to access each element during an iteration.

The `Collection` interface consists of 15 method specifications for accessing and manipulating an instance of a class that implements the `Collection` interface.

The `List` interface adds several index-related methods to the `Collection` interface. The `List` interface is partially implemented by the `AbstractList` class, and fully implemented by the `ArrayList` and `LinkedList` classes.

CROSSWORD PUZZLE



ACROSS

5. Objects that allow the elements of `Collection` objects to be accessed in a consistent way without accessing the fields of the `Collection` class.

9. A class or interface identifier followed, in angle brackets, by a list of one or more class identifiers separated by commas.

10. A class whose purpose is to enable a primitive type to be represented by (that is, wrapped inside) a class.

DOWN

1. A class in which each instance is a collection of elements.

2. The translation, by the compiler, of a wrapper object into its primitive value.

3. A generic programming technique that can be applied in a variety of situations.

4. The property by which an individual element in an array can be accessed without first accessing any of the other individual elements.

6. A dummy type that is enclosed in angle brackets in the declaration of a class or interface.

7. An object that is composed of elements.

8. In a linked collection, a special object that houses an element and at least one link to another entry.

CONCEPT EXERCISES

4.1 What is a collection? What is a collection class? What is a `Collection` class? Give an example of a collection that is not an instance of a collection class. Programming Project 4.1 has an example of a collection class that is not a `Collection` class.

4.2 An array is a collection, even though there is no array class. But an array of objects can be converted into an instance of the `ArrayList` class. Look in the file Arrays.java in the package java.util to determine the generic algorithm (that is, static method) that converts an array of objects into an `ArrayList` of those objects. How can that `ArrayList` then be printed without a loop?

4.3 **a.** Identify each of the following as either an interface or a class:

```
Collection
LinkedList
Iterator
AbstractList
```

b. What is the difference between an interface and an abstract class?

c. Of what value is an abstract class? That is, to what extent can an abstract class make a programmer more productive?

4.4 What is a list?

PROGRAMMING EXERCISES

4.1 For each of the following, create and initialize a parameterized instance, add two elements to the instance, and then print out the instance:

a. an `ArrayList` object, `scoreList`, of `Integer` objects;

b. a `LinkedList` object, `salaryList`, of `Double` objects;

4.2 Develop a `main` method in which two `ArrayList` objects are created, one with `String` elements and one with `Integer` elements. For each list, add three elements to the list, remove the element at index 1, add an element at index 0, and print out the list.

4.3 Find an `ArrayList` method, other than a constructor, that is not also a method in the `LinkedList` class. Find a `LinkedList` method, other than a constructor, that is not also a method in the `ArrayList` class.

4.4 Suppose we have the following:

```
LinkedList<String> team = new LinkedList<String> ();
team.add ("Garcia");
Iterator<String> itr = team.iterator();
Integer player = itr.next ();
```

What error message will be generated? When (at compile-time or at run-time)? Test your hypotheses.

4.5 Use the `ArrayList` class three times. First, create an `ArrayList` object, `team1`, with elements of type `String`. Add three elements to `team1`. Second, create `team2`, another `ArrayList` object with elements of type `String`. Add four elements to `team2`. Finally, create an `ArrayList` object, `league`, whose elements are `ArrayList` objects in which each element is of type `String`. Add `team1` and `team2` to `league`.

Programming Project 4.1

Wear a Developer's Hat and a User's Hat

In this project, you will get to be a developer of a parameterized class, and then become a user of that class. To start with, here are method specifications for the parameterized class, `Sequence`, with `E` the type parameter:

```
/**
 *  Initializes this Sequence object to be empty, with an initial capacity of ten
 *  elements.
 *
 */
 public Sequence()

/**
 *  Initializes this Sequence object to be empty, with a specified initial
 *  capacity.
 *
 *  @param capacity – the initial capacity of this Sequence object.
 *
 *  @throw IllegalArgumentException – if capacity is non-positive.
 *
 */
 public Sequence (int n)

/**
 *  Returns the number of elements in this Sequence object.
 *
 *  @return the number of elements in this Sequence object.
 *
 */
 public int size()

/**
 *  Appends a specified element to this Sequence object.
 *
 *  @param element – the element to be inserted at the end of this
 *         Sequence object.
 *
 */
 public void append (E element)

/**
 *  Returns the element at a specified index in this Sequence object.
 *  The worstTime(n) is constant, where n is the number of elements in this
 *  Sequence object.
 *
 *  @param k – the index of the element returned.
```

(continued on next page)

(continued from previous page)

```
       *
       *  @return the element at index k in this Sequence object.
       *
       *  @throws IndexOutOfBoundsException – if k is either negative or greater
       *          than or equal to the number of elements in this Sequence
       *          Sequence object.
       *
       */

       public E get (int k)

      /**
       *  Changes the element at a specified index in this Sequence object.
       *  The worstTime(n) is constant, where n is the number of elements in this
       *  Sequence object.
       *
       *  @param k – the index of the element returned.
       *  @param newElement – the element to replace the element at index k in
       *         this Sequence object.
       *
       *  @throws IndexOutOfBoundsException – if k is either negative or greater
       *          than or equal to the number of elements in this Sequence
       *          object.
       *
       */
       public void set (int k, E newElement)
```

Part 1 Create unit tests based on the method specifications and stubs.

Part 2 Define the methods in the `Sequence` class.
Hint: use the following fields:

```
    protected E [ ] data;

    protected int size;   // the number of elements in the Sequence, not the
                          // capacity of the data array
```

Note 1: for the `append` method, if the `data` array is currently full, its capacity must be increased before the new element can be appended. See Programming Exercise 2.10 to see how to accomplish the expansion.
Note 2: for methods that may throw an exception, do not include **`catch`** blocks. Instead, the exception will be propagated, so the handling can be customized for the application.

Part 3 Test the method definitions in your `Sequence` class.

Recursion

CHAPTER 5

One of the skills that distinguish a novice programmer from an experienced one is an understanding of recursion. The goal of this chapter is to give you a feel for situations in which a recursive method is appropriate. Along the way you may start to see the power and elegance of recursion, as well as its potential for misuse. Recursion plays a minor role in the Java Collections Framework: two of the `sort` methods are recursive, and there are several recursive methods in the `TreeMap` class. But the value of recursion extends far beyond these methods. For example, one of the applications of the `Stack` class in Chapter 8 is the translation of recursive methods into machine code. The sooner you are exposed to recursion, the more likely you will be able to spot situations where it is appropriate—and to use it.

CHAPTER OBJECTIVES

1. Recognize the characteristics of those problems for which recursive solutions may be appropriate.
2. Compare recursive and iterative methods with respect to time, space, and ease of development.
3. Trace the execution of a recursive method with the help of execution frames.
4. Understand the backtracking design pattern.

5.1 Introduction

Roughly, a method is ***recursive*** if it contains a call to itself.[1] From this description, you may initially fear that the execution of a recursive method will lead to an infinite sequence of recursive calls. But under normal circumstances, this calamity does not occur, and the sequence of calls eventually stops. To show you how recursive methods terminate, here is the skeleton of the body of a typical recursive method:

```
if (simplest case)
     solve directly
else
     make a recursive call with a simpler case
```

This outline suggests that recursion should be considered whenever the problem to be solved has these two characteristics;

1. The simplest case(s) can be solved directly.
2. Complex cases of the problem can be reduced to simpler cases of the same form as the original problem.

[1]A formal definition of "recursive" is given later in this chapter.

Incidentally, if you are familiar with the Principle of Mathematical Induction, you may have observed that these two characteristics correspond to the base case and inductive case, respectively. In case you are not familiar with that principle, Section A2.5 of Appendix 2 is devoted to mathematical induction.

As we work through the following examples, do not be inhibited by old ways of thinking. As each problem is stated, try to frame a solution in terms of a simpler problem of the same form. Think recursively!

5.2 Factorials

Given a positive integer n, the ***factorial*** of n, written $n!$, is the product of all integers between n and 1, inclusive. For example,

$$4! = 4 * 3 * 2 * 1 = 24$$

and

$$6! = 6 * 5 * 4 * 3 * 2 * 1 = 720$$

Another way to calculate 4! is as follows:

$$4! = 4 * 3!$$

This formulation is not helpful unless we know what 3! is. But we can continue to calculate factorials in terms of smaller factorials (Aha!):

$$3! = 3 * 2!$$
$$2! = 2 * 1!$$

Note that 1! Can be calculated directly; its value is 1. Now we work backwards to calculate 4!:

$$2! = 2 * 1! = 2 * 1 = 2$$
$$3! = 3 * 2! = 3 * 2 = 6$$

Finally, we get

$$4! = 4 * 3! = 4 * 6 = 24$$

For $n > 1$, we reduce the problem of calculating $n!$ to the problem of calculating $(n - 1)!$. We stop reducing when we get to 1!, which is simply 1. For the sake of completeness[2], we define 0! to be 1.

There is a final consideration before we specify, test and define the `factorial` method: what about exceptions? If n is less than zero, we should throw an exception—`IllegalArgumentException` is appropriate. And because $n!$ is exponential in n, the value of $n!$ will be greater than `Long.MAX_VALUE` for not-very-large values of n. In fact, 21!> `Long.MAX_VALUE`, so we should also throw `IllegalArgumentException` for $n > 20$.

Here is the method specification:

```
/**
 * Calculates the factorial of a non-negative integer, that is, the product of all
 * integers between 1 and the given integer, inclusive. The worstTime(n) is O(n),
```

[2]The calculation of 0! occurs in the study of probability: The number of combinations of n things taken k at a time is calculated as $n!/(k!\,(n - k)!)$. When $n = k$, we get $n!/(n!)\,(0!)$, which has the value 1 because $0! = 1$. And note that 1 is the number of combinations of n things taken n at a time.

```
 * where n is the given integer.
 *
 * @param n the integer whose factorial is calculated.
 *
 * @return the factorial of n
 *
 * @throws IllegalArgumentException if n is less than 0 or greater than 20 (note
 *     that 21! > Long.MAX_VALUE).
 *
 */
public static long factorial (int n)
```

Note that `factorial` has a **`static`** modifier in its heading (see Section 2.1 in Chapter 2). Why? All the information needed by the method is provided by the parameter, and the only effect of a call to the method is the value returned. So a calling object would neither affect nor be affected by an invocation of the method. As noted in Chapter 2, we adhere to the test-first model. So the test class, based on the method specification only, is developed before the method itself is defined. Here is the test class, with special emphasis on boundary conditions, and including the usual stub within the test class itself:

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;

public class FactorialTest
{
  public static void main(String[ ] args)
  {
    Result result = runClasses (FactorialTest.class);
    System.out.println ("Tests run = " + result.getRunCount() +
                        "\nTests failed = " + result.getFailures());
  } // method main   import org.junit.*;

  @Test
  public void factorialTest1()
  {
     assertEquals (24,  factorial (4));
  } // method factorialTest1

  @Test
  public void factorialTest2()
  {
     assertEquals (1, factorial (0));
  } // method factorialTest2

  @Test
```

```
   public void factorialTest3()
   {
      assertEquals (1, factorial (1));
   } // method factorialTest3

   @Test
   public void factorialTest4()
   {
      assertEquals (2432902008176640000L, factorial (20));
   } // method factorialTest4

   @Test (expected = IllegalArgumentException.class)
   public void factorialTest5()
   {
      factorial (21);
   } // method factorialTest5

   @Test (expected = IllegalArgumentException.class)
   public void factorialTest6 ()
   {
      factorial (-1);
   } // method factorialTest6


   public static long factorial (int n)
   {
       throw new UnsupportedOperationException();
   } // method factorial

} // class FactorialTest
```

As expected, all tests of the `factorial` method failed.

We now define the `factorial` method. For the sake of efficiency, checking for values of n less than 0 or greater than 20 should be done just once, instead of during each recursive call. To accomplish this, we will define a wrapper method that throws an exception for out-of-range values of n, and (otherwise) calls a recursive method to actually calculate $n!$. Here are the method definitions.

```
public static long factorial (int n)
{
       final int MAX_INT = 20;      // because 21! > Long.MAX_VALUE

       final String ERROR_MESSAGE = "The value of n must be >= 0 and <= " +
               Integer.toString (MAX_INT);

       if (n < 0 || n > MAX_INT)
               throw new IllegalArgumentException (ERROR_MESSAGE);
       return fact (n);
} // method factorial
```

```
/**
 * Calculates n!.
 *
 *  @param n the integer whose factorial is calculated.
 *
 *  @return n!.
 *
 */
protected static long fact (int n)
{
      if (n <= 1)
              return 1;
      return n * fact (n - 1);
} // method fact
```

The testing of the wrapper method, `factorial`, incorporates the testing of the wrapped method, `fact`. The book's website has similar test classes for all recursive functions in this chapter.

Within the method `fact`, there is a call to the method `fact`, and so `fact`, unlike `factorial`, is a recursive method. The parameter `n` has its value reduced by 1 with each recursive call. But after the final call with `n` = 1, the previous values of `n` are needed for the multiplications. For example, when `n` = 4, the calculation of `n * fact (n - 1)` is postponed until the call to `fact (n - 1)` is completed. When this finally happens and the value 6 (that is, `fact (3)`) is returned, the value of 4 for `n` must be available to calculate the product.

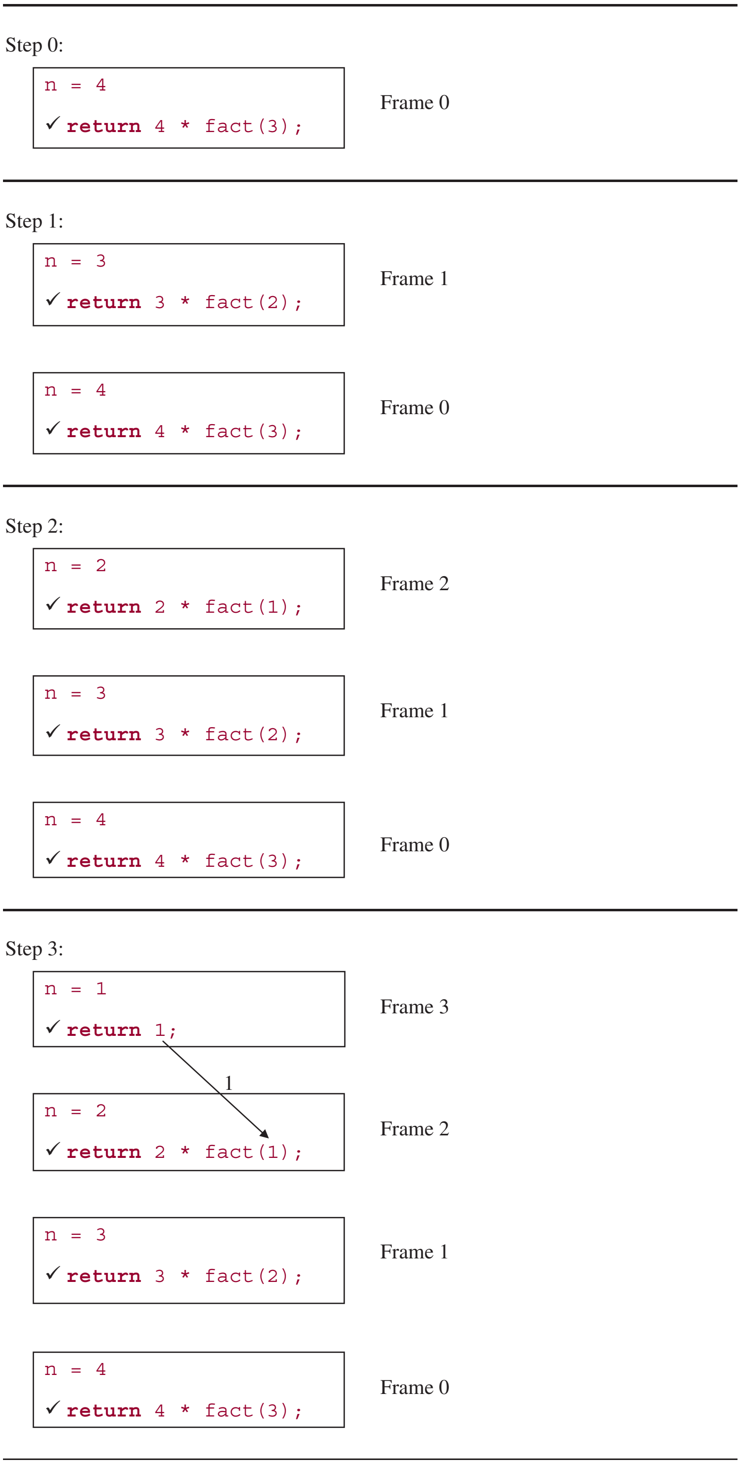
Somehow, the value of `n` must be saved when the call to `fact (n - 1)` is made. That value must be restored after the call to `fact (n - 1)` is completed so that the value of `n * fact (n - 1)` can be calculated. The beauty of recursion is that the programmer need not explicitly handle these savings and restorings; the compiler and computer do the work.

5.2.1 Execution Frames

The trace of a recursive method can be illustrated through ***execution frames***: boxes that contain information related to each invocation of the method. Each execution frame includes the values of parameters and other local variables. Each frame also has the relevant part of the recursive method's code—especially the recursive calls, with values for the arguments. When a recursive call is made, a new execution frame will be constructed on top of the current one; this new frame is destroyed when the call that caused its creation has been completed. A check mark indicates either the statement being executed in the current frame or the statement, in a previous frame, whose recursive call created (immediately or eventually) the current frame.

At any time, the top frame contains information relevant to the current execution of the recursive method. For example, here is a step-by-step, execution-frame trace of the `fact` method after an initial call of `fact (4)`:

The analysis of the `fact` method is fairly clear-cut. The execution-time requirements correspond to the number of recursive calls. For any argument `n`, there will be exactly `n - 1` recursive calls. During each recursive call, the **`if`** statement will be executed in constant time, so worstTime(n) is linear in n. Recursive methods often have an additional cost in terms of memory requirements. For example, when each recursive call to `fact` is made, the return address and a copy of the argument are saved. So worstSpace (n) is also linear in n.



Step 0:

```
n = 4
✓return 4 * fact(3);
```

Frame 0

Step 1:

```
n = 3
✓return 3 * fact(2);
```

Frame 1

```
n = 4
✓return 4 * fact(3);
```

Frame 0

Step 2:

```
n = 2
✓return 2 * fact(1);
```

Frame 2

```
n = 3
✓return 3 * fact(2);
```

Frame 1

```
n = 4
✓return 4 * fact(3);
```

Frame 0

Step 3:

```
n = 1
✓return 1;
```

Frame 3

```
n = 2
✓return 2 * fact(1);
```

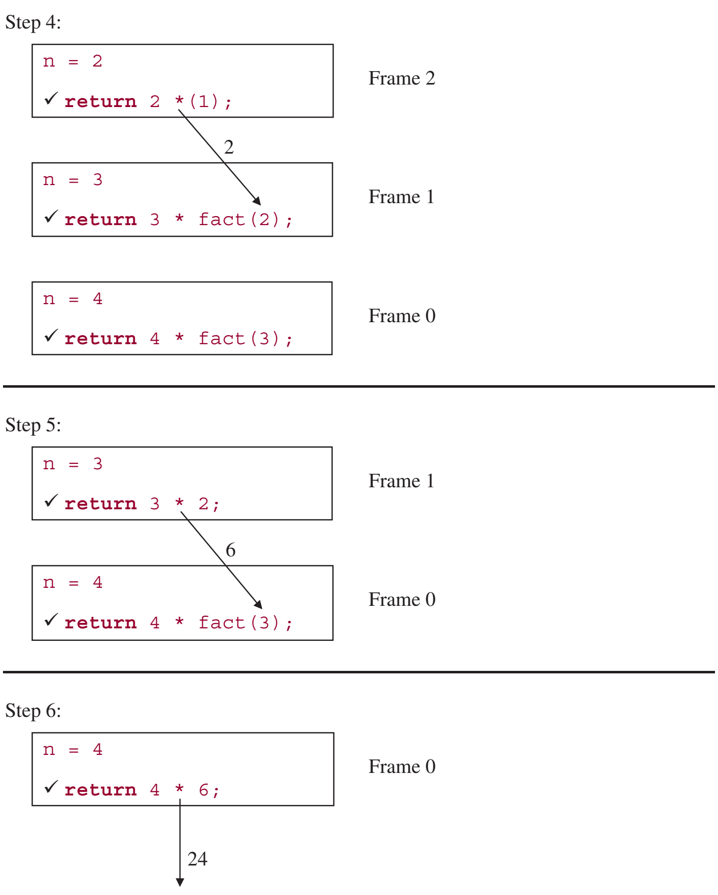
Frame 2

```
n = 3
✓return 3 * fact(2);
```

Frame 1

```
n = 4
✓return 4 * fact(3);
```

Frame 0



Recursion can often make it easier for us to solve problems, but **any problem that can be solved recursively can also be solved iteratively**. An ***iterative*** method is one that has a loop instead of a recursive call. For example, here is an iterative method to calculate factorials. No wrapper method is needed because there are no recursive calls.

```
/**
 * Calculates the factorial of a non-negative integer, that is, the product of all
 * integers between 1 and the given integer, inclusive.  The worstTime(n) is O(n),
 * where n is the given integer.
 *
 * @param n the non-negative integer whose factorial is calculated.
 *
 * @return the factorial of n
 *
 * @throws IllegalArgumentException if n is less than 0 or greater than 20 (note
 *      that 21! > Long.MAX_VALUE).
```

```
    *
    */
    public static long factorial (int n)
    {
           final int MAX_INT = 20;     // because 21! > Long.MAX_VALUE

           final String ERROR_MESSAGE = "The value of n must be >= 0 and <= " +
                                         Integer.toString (MAX_INT);

           if (n < 0 || n > MAX_INT)
                  throw new IllegalArgumentException (ERROR_MESSAGE);

           long product = n;

           if (n == 0)
                  return 1;
           for (int i = n-1; i > 1; i-)
                  product = product * i;
           return product;
    } // method factorial
```

This version of `factorial` passed all of the tests in `FactorialTest`. For this version of `factorial`, worstTime(n) is linear in n, the same as for the recursive version. But no matter what value `n` has, only three variables (`n`, `product` and `i`) are allocated in a trace of the iterative version, so worstSpace(n) is constant, versus linear in n for the recursive version. Finally, the iterative version follows directly from the definition of factorials, whereas the recursive version represents your first exposure to a new problem-solving technique, and that takes some extra effort.

So in this example, the iterative version of the `factorial` method is better than the recursive version. The whole purpose of the example was to provide a simple situation in which recursion was worth considering, even though we ultimately decided that iteration was better. In the next example, an iterative alternative is slightly less appealing.

5.3 Decimal to Binary

Humans count in base ten, possibly because we were born with ten fingers. Computers count in base two because of the binary nature of electronic switches. One of the tasks a computer performs is to convert from decimal (base ten) to binary (base two). Let's develop a method to solve a simplified version of this problem:

> Given a nonnegative integer n, determine its binary equivalent.

For example, if n is 25, the binary equivalent is $11001 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$. For a large **`int`** value such as one billion, the binary equivalent will have about 30 bits. But 30 digits of zeros and ones are too big for an **`int`** or even a **`long`**. So we will store the binary equivalent in a `String` object. The method specification is:

```
   /**
    *
    * Determines the binary equivalent of a non-negative integer.  The worstTime(n)
    * is O(log n), where n is the given integer.
```

```
 *
 * @param n the non-negative integer, in decimal notation.
 *
 * @return a String representation of the binary equivalent of n.
 *
 * @throws  IllegalArgumentException if n is negative.
 */
public static String getBinary (int n)
```

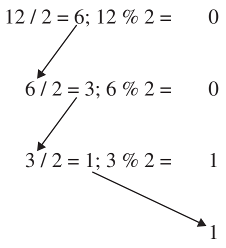
The test class for this method can be found on the book's website, and includes the following test:

```
@Test (expected = IllegalArgumentException.class)
public void getBinaryTest5()
{
    getBinary (-1);
} // method getBinaryTest5
```

There are several approaches to solving this problem. One of them is based on the following observation:

> The rightmost bit has the value of $n \% 2$; the other bits are the binary equivalent of $n/2$. (Aha!)

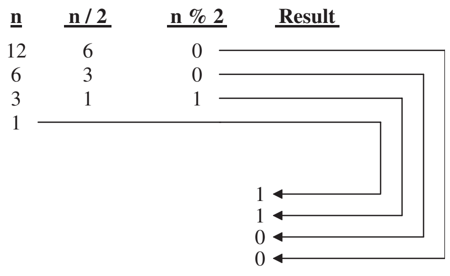
For example, if n is 12, the rightmost bit in the binary equivalent of n is 12% 2, namely, 0; the remaining bits are the binary equivalent of 12/2, that is, the binary equivalent of 6. So we can obtain all the bits as follows:



When the quotient is 1, the binary equivalent is simply 1. We concatenate (that is, join together) these bits from the bottom up, so that the rightmost bit will be joined last. The result would then be

```
1100
```

The following table graphically illustrates the effect of calling `getBinary (12)`:



This discussion suggests that we must perform all of the calculations before we return the result. Speaking recursively, we need to calculate the binary equivalent of `n/2` *before* we append the value of `n % 2`. In other words, we need to append the result of `n % 2` to the result of the recursive call.

We will stop when `n` is 1 or 0, and 0 will occur only if `n` is initially 0. As we did in Section 5.2, we make `getBinary` a wrapper method for the recursive `getBin` method. The method definitions are:

```
public static String getBinary (int n)
{
       if (n < 0)
               throw new IllegalArgumentException();
       return getBin (n);
} // method getBinary

public static String getBin (int n)
{
       if (n <= 1)
               return Integer.toString (n);
       return getBin (n / 2) + Integer.toString (n % 2);
} // method getBin
```

We are assured that the simple case of `n <= 1` will eventually be reached because in each execution of the method, the argument to the recursive call is at most half as big as the method parameter's value.

Here is a step-by-step, execution-frame trace of the `getBin` method after an initial call of `getBin (12)`:

The final value returned is the string:

```
1100
```

And that is the binary equivalent of 12.

As we noted earlier, the order of operands in the `String` expression of the last **`return`** statement in `getBin` enables us to postpone the final return until *all* of the bit values have been calculated. If the order had been reversed, the bits would have been returned in reverse order. Recursion is such a powerful tool that the effects of slight changes are magnified.

Step 0:

```
n = 12
✓ return getBin (6) + Integer.toString (0);
```

Frame 0

Step 1:

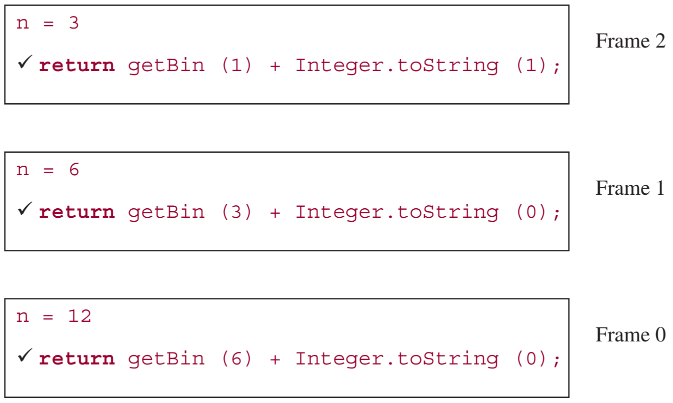
```
n = 6
✓ return getBin (3) + Integer.toString (0);
```

Frame 1

```
n = 12
✓ return getBin (6) + Integer.toString (0);
```

Frame 0

Step 2:

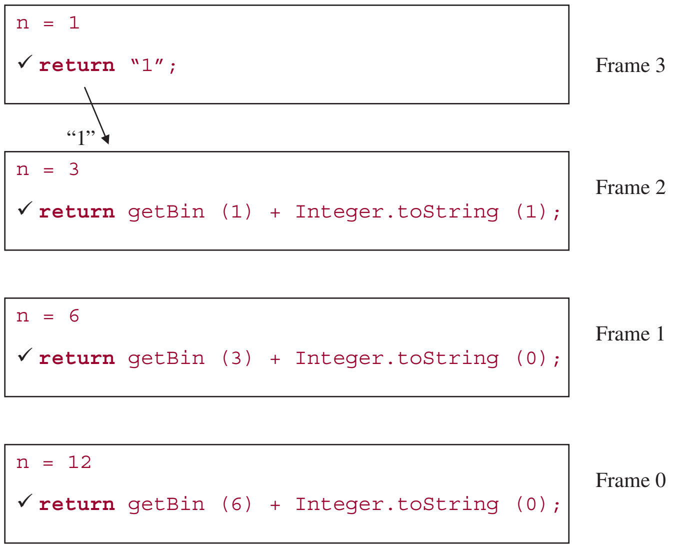


```
n = 3
✓ return getBin (1) + Integer.toString (1);
```

```
n = 6
✓ return getBin (3) + Integer.toString (0);
```

```
n = 12
✓ return getBin (6) + Integer.toString (0);
```

Step 3:



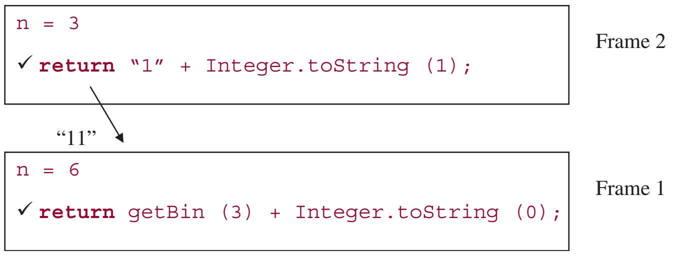
```
n = 1
✓ return "1";
```

```
n = 3
✓ return getBin (1) + Integer.toString (1);
```

```
n = 6
✓ return getBin (3) + Integer.toString (0);
```

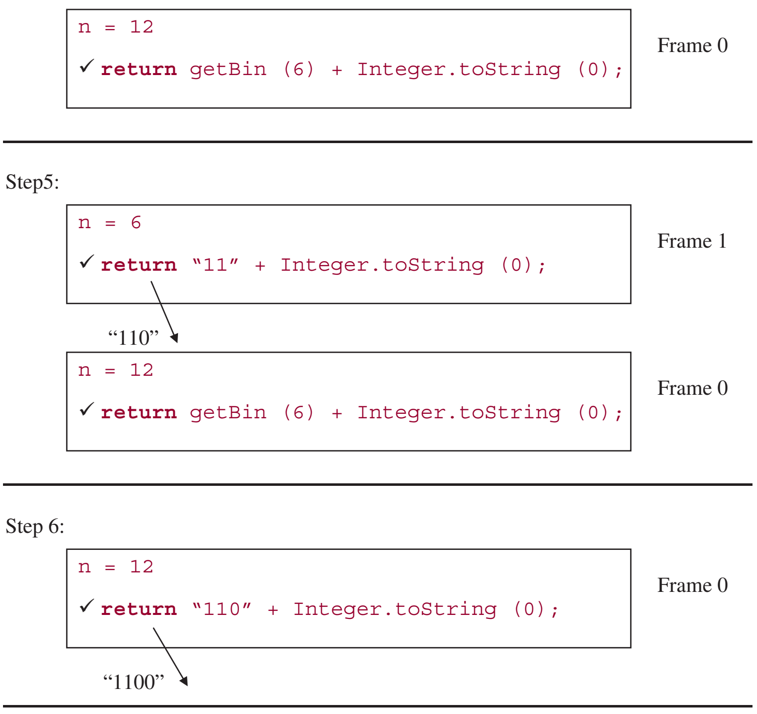
```
n = 12
✓ return getBin (6) + Integer.toString (0);
```

Step 4:



```
n = 3
✓ return "1" + Integer.toString (1);
```

```
n = 6
✓ return getBin (3) + Integer.toString (0);
```



```
n = 12
✓ return getBin (6) + Integer.toString (0);
```
Frame 0

Step5:

```
n = 6
✓ return "11" + Integer.toString (0);
```
Frame 1

"110"

```
n = 12
✓ return getBin (6) + Integer.toString (0);
```
Frame 0

Step 6:

```
n = 12
✓ return "110" + Integer.toString (0);
```
Frame 0

"1100"

As usually happens with recursive methods, the time and space requirements for `getBin` are estimated by the number of recursive calls. The number of recursive calls is the number of times that n can be divided by 2 until n equals 1. As we saw in Section 3.1.2 of Chapter 3, this value is $\mathit{floor}(\log_2 n)$, so worstTime(n) and worstSpace(n) are both logarithmic in n.

A user can call the `getBinary` method from the following `run` method (in a `BinaryUser` class whose `main` method simply calls **`new`** `BinaryUser().run()`):

```
public static void run()
{
        final int SENTINEL = -1;

        final String INPUT_PROMPT =
               "\nPlease enter a non-negative base-10 integer (or " +
               SENTINEL + " to quit): ";

        final String RESULT_MESSAGE = "The binary equivalent is ";

        Scanner sc = new Scanner (System.in);

        int n;

        while (true)
        {
              try
              {
```

```
                        System.out.print (INPUT_PROMPT);
                        n = sc.nextInt();
                        if (n == SENTINEL)
                                break;
                        System.out.println (RESULT_MESSAGE + getBinary (n));
                } // try
                catch (Exception e)
                {
                        System.out.println (e);
                        sc.nextLine();
                }// catch Exception
        }// while
} // method run
```

You are invited to develop an iterative version of the `getBinary` method. (See Programming Exercise 5.2.) After you have completed the iterative method, you will probably agree that it was somewhat harder to develop than the recursive method. This is typical, and probably obvious: recursive solutions usually flow more easily than iterative solutions to those problems for which recursion is appropriate. Recursion is appropriate when larger instances of the problem can be reduced to smaller instances that have the same form as the larger instances.

You are now prepared to do Lab 7: Fibonacci Numbers

For the next problem, an iterative solution is *much* harder to develop than a recursive solution.

5.4 Towers of Hanoi

In the Towers of Hanoi game, there are three poles, labeled 'A', 'B' and 'C', and several, different-sized, numbered disks, each with a hole in the center. Initially, all of the disks are on pole 'A', with the largest disk on the bottom, then the next largest, and so on. Figure 5.1 shows the initial configuration if we started with four disks, numbered from smallest to largest.

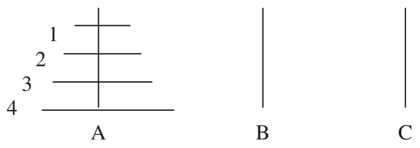


FIGURE 5.1 The starting position for the Towers of Hanoi game with four disks

The object of the game is to move all of the disks from pole 'A' to pole 'B'; pole 'C' is used for temporary storage[3]. The rules of the game are:

1. Only one disk may be moved at a time.
2. No disk may ever be placed on top of a smaller disk.
3. Other than the prohibition of rule 2, the top disk on any pole may be moved to either of the other two poles.

[3]In some versions, the goal is to move the disks from pole 'A' to pole 'C', with pole 'B' used for temporary storage.

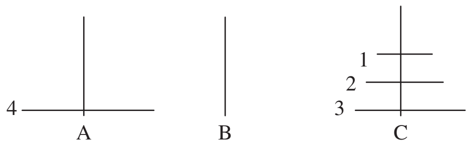


FIGURE 5.2 The game configuration for the Towers of Hanoi just before moving disk 4 from pole 'A' to pole 'B'

We will solve this problem in the following generalization: Show the steps to move n disks from an origin pole to a destination pole, using the third pole as a temporary. Here is the method specification for this generalization:

```
/**
 * Determines the steps needed to move n disks from an origin to a destination.
 * The worstTime(n) is O(2^n).
 *
 * @param n the number of disks to be moved.
 * @param orig the pole where the disks are originally.
 * @param dest the destination pole
 * @param temp the pole used for temporary storage.
 *
 * @return a String representation of the moves needed, where each
 * move is in the form "Move disk ? from ? to ?\n".
 *
 * @throws IllegalArgumentException if n is less than or equal to 0.
 */
public static String moveDisks (int n, char orig, char dest, char temp)
```

The test class for `moveDisks` can be found on the book's website, and includes the following test:

```
@Test
public void moveDisksTest1()
{
    assertEquals ("Move disk 1 from A to B\nMove disk 2 from A to C\n" +
                  "Move disk 1 from B to C\nMove disk 3 from A to B\n" +
                  "Move disk 1 from C to A\nMove disk 2 from C to B\n" +
                  "Move disk 1 from A to B\n", moveDisks (3, 'A', 'B', 'C'));
} // method moveDisksTest1
```

Let's try to play the game with the initial configuration given in Figure 5.1. We are immediately faced with a dilemma: Do we move disk 1 to pole 'B' or to pole 'C'? If we make the wrong move, we may end up with the four disks on pole 'C' rather than on pole 'B'.

Instead of trying to figure out where disk 1 should be moved initially, we will focus our attention on disk 4, the bottom disk. Of course, we can't move disk 4 right away, but eventually, disk 4 will have to be moved from pole 'A' to pole 'B'. By the rules of the game, the configuration just before moving disk 4 must be as shown in Figure 5.2.

Does this observation help us to figure out how to move 4 disks from 'A' to 'B'? Well, sort of. We still need to determine how to move three disks (one at a time) from pole 'A' to pole 'C'. We can then move disk 4 from 'A' to 'B'. Finally, we will need to determine how to move three disks (one at a time) from 'C' to 'B'.

The significance of this strategy is that we have reduced the problem from figuring how to move four disks to one of figuring how to move three disks. (Aha!) We still need to determine how to move three disks from one pole to another pole.

But the above strategy can be re-applied. To move three disks from, say, pole ‘A’ to pole ‘C’, we first move two disks (one at a time) from ‘A’ to ‘B’, then we move disk 3 from ‘A’ to ‘C’, and finally, we move two disks from ‘B’ to ‘C’. Continually reducing the problem, we eventually face the trivial task of moving disk 1 from one pole to another.

There is nothing special about the number 4 in the above approach. For any positive integer n we can describe how to move n disks from pole ‘A’ to pole ‘B’: if $n = 1$, we simply move disk 1 from pole ‘A’ to pole ‘B’. For $n > 1$,

1. First, move $n - 1$ disks from pole ‘A’ to pole ‘C’, using pole ‘B’ as a temporary.
2. Then move disk n from pole ‘A’ to pole ‘B’.
3. Finally, move $n - 1$ disks from pole ‘C’ to pole ‘B’, using pole ‘A’ as a temporary.

This does not quite solve the problem because, for example, we have not described how to move $n - 1$ disks from ‘A’ to ‘C’. But our strategy is easily generalized by replacing the constants ‘A’, ‘B’, and ‘C’ with variables *origin, destination*, and *temporary*. For example, we will initially have

```
origin = 'A'
destination = 'B'
temporary = 'C'
```

Then the general strategy for moving n disks from origin to destination is as follows:

If n is 1, move disk 1 from *origin* to *destination*.

Otherwise,

1. Move $n - 1$ disks (one at a time) from *origin* to *temporary*;
2. Move disk n from *origin* to *destination*;
3. Move $n - 1$ disks (one at a time) from *temporary* to *destination*.

The following recursive method incorporates the above strategy for moving n disks. If $n = 1$, the `String` representing the move, namely, `"Move disk 1 from " + orig + " to " + dest + "\n"` is simply returned. Otherwise, the `String` object returned consists of three `String` objects concatenated together, namely, the strings returned by

```
move (n - 1, orig, temp, dest)
"Move disk " + n + " from " + orig + " to " + dest + "\n"
move (n - 1, temp, dest, orig)
```

When the final return is made, the return value is the complete sequence of moves. This `String` object can then be printed to the console window, to a GUI window, or to a file. For the sake of efficiency, the test for $n \leq 0$ is made—once—in a wrapper method `moveDisks` that calls the `move` method. Here is the method specification for `moveDisks`:

```
/**
 * Determines the steps needed to move disks from an origin to a destination.
 * The worstTime(n) is O(2^n).
 *
 * @param n the number of disks to be moved.
 * @param orig the pole where the disks are originally.
 * @param dest the destination pole
 * @param temp the pole used for temporary storage.
 *
 * @return a String representation of the moves needed.
 *
 * @throws IllegalArgumentException if n is less than or equal to 0.
 */
public static String moveDisks (int n, char orig, char dest, char temp)
```

The test class for `moveDisks` can be found on the book's website. The definitions of `moveDisks` and `move` are as follows:

```
public static String moveDisks (int n, char orig, char dest, char temp)
{
    if (n <= 0)
            throw new IllegalArgumentException();
    return move (n, orig, dest, temp);
} // method moveDisks

/**
 * Determines the steps needed to move disks from an origin to a destination.
 * The worstTime(n) is O(2^n).
 *
 * @param n the number of disks to be moved.
 * @param orig the pole where the disks are originally.
 * @param dest the destination pole
 * @param temp the pole used for temporary storage.
 *
 * @return a String representation of the moves needed.
 *
 */
public static String move (int n, char orig, char dest, char temp)
{
    final String DIRECT_MOVE =
            "Move disk " + n + " from " + orig + " to " + dest + "\n";

    if (n == 1)
            return DIRECT_MOVE;
    String result = move (n - 1, orig, temp, dest);
    result += DIRECT_MOVE;
    result += move (n - 1, temp, dest, orig);
    return result;
} // method move
```

It is difficult to trace the execution of the `move` method because the interrelationship of parameter and argument values makes it difficult to keep track of which pole is currently the origin, which is the destination and which is the temporary. In the following execution frames, the parameter values are the argument values from the call, and the argument values for subsequent calls come from the method code and the current parameter values. For example, suppose the initial call is:

```
move (3, 'A', 'B', 'C');
```

Then the parameter values at step 0 will be those argument values, so we have:

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
```

Because `n` is not equal to 1, the recursive part is executed:

```
String result = move (n - 1, orig, temp, dest);
result += DIRECT_MOVE;
result += move (n - 1, temp, dest, orig);
return result;
```

The values of those arguments are obtained from the parameters' values, so the statements are equivalent to:

```
String result = move (2, 'A', 'C', 'B');
result = "Move disk 3 from A to B\n";
result = move (2, 'C', 'B', 'A');
return result;
```

Make sure you understand how to obtain the parameter values and argument values before you try to follow the trace given below.

Here is a step-by-step, execution-frame trace of the `move` method when the initial call is:

```
move (3, 'A', 'B', 'C');
```

```
                              value of result
```

Step 0:

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

✓String result = move (2, 'A', 'C', 'B');
 result += "Move disk 3 from A to B\n";
 result += move (2,'C','B','A');
 return result;
```

Step 1:

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'

✓String result = move (1, 'A', 'B', 'C');
 result += "Move disk 2 from A to C\n";
 result += move (1,'B','C','A');
 return result;
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

✓String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
  result += move (2,'C','B','A');
  return result;
```

Step 2:

```
n = 1
orig = 'A'
dest = 'B'
temp = 'C'

✓ return "Move disk 1 from A to B\n";
```

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'

✓String result = move (1, 'A', 'B', 'C');
 result += "Move disk 2 from A to C\n";
 result += move (1, 'B', 'C', 'A');
 return result;
```

Move disk 1 from A to B

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

✓String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
  result += move (2,'C','B','A');
  return result;
```

Step 3:

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'

  String result = move (1, 'A', 'B', 'C');
✓result += "Move disk 2 from A to C\n";
  result += move (1,'B','C','A');
  return result;
```

Move disk 1 from A to B
Move disk 2 from A to C

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

✓String result = move (2, 'A', 'C', 'B');
 result += "Move disk 3 from A to B\n";
 result += move (2,'C','B','A');
 return result;
```

Step 4:

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'

 String result = move (1, 'A', 'B', 'C');
 result += "Move disk 2 from A to C\n";
✓result += move (1, 'B', 'C', 'A');
 return result;
```

```
n = 3
orig = 'A'
dest = 'C'
temp = 'B'

✓String result = move (2, 'A', 'C', 'B');
 result += "Move disk 3 from A to B\n";
 result += move (2, 'C', 'B', 'A');
 return result;
```

Step 5:

```
n = 1
orig = 'B'
dest = 'C'
temp = 'A'

✓ return "Move disk 1 from B to C\n";
```

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'

 String result = move (1, 'A', 'B', 'C');
 result += "Move disk 2 from A to C\n";
✓result += move (1,'B','C','A');
 return result;
```

Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

 ✓String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
  result += move (2,'C','B','A' );
  return result;
```

Step 6:

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

  String result = move (2, 'A', 'C', 'B');
 ✓result += "Move disk 3 from A to B\n";
  result += move (2, 'C', 'B', 'A');
  return result;
```

```
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
```

Step 7:

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

  String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
 ✓result += move (2, 'C', 'B', 'A');
  return result;
```

Step 8:

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'

 ✓String result = move (1, 'C', 'A', 'B');
  result += "Move disk 2 from C to B";
  result += move (1,'A','B','C');
  return result;
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

  String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
 ✓result += move (2,'C','B','A');
  return result;
```

Step 9:

```
n = 1
orig = 'C'
dest = 'A'
temp = 'B'

 ✓return "Move disk 1 from C to A\n";
```

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'

 ✓String result = move (1, 'C', 'A', 'B');
  result += "Move disk 2 from C to B\n";
  result += move (1, 'A', 'B', 'C');
  return result;
```

```
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

  String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
 ✓result += move (2, 'C', 'B', 'A');
  return result;
```

Step 10:

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'

  String result = move (1, 'C', 'A', 'B');
 ✓result += "Move disk 2 from C to B\n";
  result += move (1,'A','B','C');
  return result;
```

```
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 1 from C to B
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

  String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
 ✓result += move (2,'C','B','A');
  return result;
```

Step 11:

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'

  String result = move (1, 'C', 'A', 'B');
  result += "Move disk 2 from C to B\n";
 ✓result += move (1, 'A', 'B', 'C');
  return result;
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

  String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
 ✓result += move (2, 'C', 'B', 'A');
  return result;
```

Step 12:

```
n = 1
orig = 'A'
dest = 'B'
temp = 'C'

✓ "Move disk 1 from A to B\n";
```

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'

  String result = move (1, 'C', 'A', 'B');
  result += "Move disk 2 from C to B\n";
 ✓result += move (1,'A','B','C');
  return result;
```

Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'

  String result = move (2, 'A', 'C', 'B');
  result += "Move disk 3 from A to B\n";
 ✓result += move (2,'C','B','A');
  return result;
```

Notice the disparity between the relative ease in developing the recursive method and the relative difficulty in tracing its execution. Imagine what it would be like to trace the execution of `move (15,'A','B','C')`. Fortunately, you need not undergo such torture. Computers handle this type of tedious detail very well. You "merely" develop the correct program and the computer handles the execution. For the `move` method—as well as for the other recursive methods in this chapter—you can actually *prove* the correctness of the method. See Exercise 5.5.

The significance of ensuring the precondition (see the `@throws` specification) is illustrated in the `move` method. For example, let's see what would happen if no exception were thrown and `move` were called with 0 as the first argument. Since `n` would have the value 0, the condition of the **`if`** statement would be false, and there would be a call to `move (-1,...).` Within that call, `n` would still be unequal to 1, so there would be a call to `move (-2,...)` then to `move (-3,....)`, `move (-4,...)`, `move (-5,...)`, and so on. Eventually, saving all those copies of `n` would overflow an area of memory called the ***stack***. This phenomenon known is as ***infinite recursion***. A `StackOverflowError`—not an exception—is generated, and the execution of the project terminates. In general, infinite recursion is avoided if each recursive call makes progress toward a "simplest" case. And, just to be on the safe side, the method should throw an exception if the precondition is violated.

A recursive method does not explicitly describe the considerable detail involved in its execution. For this reason, recursion is sometimes referred to as "the lazy programmer's problem-solving tool." If you want to appreciate the value of recursion, try to develop an iterative version of the `move` method. Programming Project 5.1 provides some hints.

5.4.1 Analysis of the `move` Method

What about worstTime(n)? In determining the time requirements of a recursive method, the number of calls to the method is of paramount importance. To get an idea of the number of calls to the `move` method, look at the tree in Figure 5.3.

As illustrated in Figure 5.3, the first call to the `move` method has `n` as the first argument. During that call, two recursive calls to the `move` method are made, and each of those two calls has `n - 1` as the first

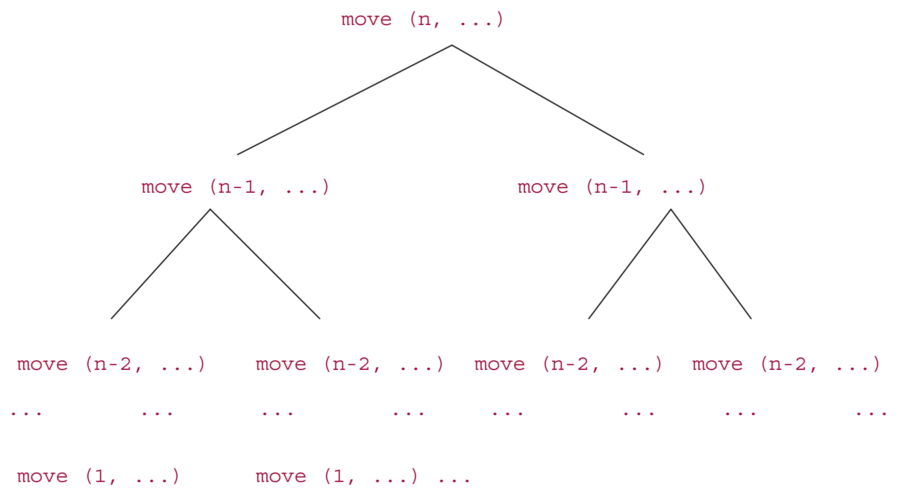


FIGURE 5.3 A schematic of the number of calls to the `move` method

argument. From each of those calls, we get two more calls to move, and each of those four calls has `n - 2` as the first argument. This process continues until, finally, we get calls with `1` as the first argument.

To calculate the total number of calls to the `move` method, we augment the tree in Figure 5.3 by identifying levels in the tree, starting with level 0 at the top, and include the number of calls at each level. At level 0, the number of calls is $1 (= 2^0)$. At level 1, the number of calls is $2\ (= 2^1)$. In general, at level k there are 2^k calls to the move method. Because there are n levels in the tree and the top is level 0, the bottom must be level $n - 1$, where there are 2^{n-1} calls to the `move` method. See Figure 5.4.

From Figure 5.4, we see that the total number of calls to the `move` method is

$$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{n-1} = \sum_{k=0}^{n-1} 2^k$$

By Example A2.6 in Appendix 2, this sum is equal to $2^n - 1$. That is, the number of calls to the `move` method is $2^n - 1$. We conclude that, for the `move` method, worstTime(n) is exponential in n; specifically, worstTime(n) is $\Theta(2^n)$. In fact, since *any* definition of the `move` method must return a string that has $2^n - 1$ lines, the Towers of Hanoi problem is intractable. That is, *any* solution to the Towers of Hanoi problem must take exponential time.

The memory requirements for `move` are modest because although space is allocated when `move` is called, that space is deallocated when the call is completed. So the amount of additional memory needed for `move` depends, not simply on the number of calls to `move`, but on the maximum number of started-but-not-completed calls. We can determine this number from the execution frames. Each time a recursive call is made, another frame is constructed, and each time a return is made, that frame is destroyed. For example, if $n = 3$ in the original call to `move`, then the maximum number of execution frames is 3. In general, the maximum number of execution frames is `n`. So worstSpace(n) is linear in n.

We now turn our attention to a widely known search technique: binary search. We will develop a recursive method to perform a binary search on an array. Lab 9 deals with the development of an iterative version of a binary search.

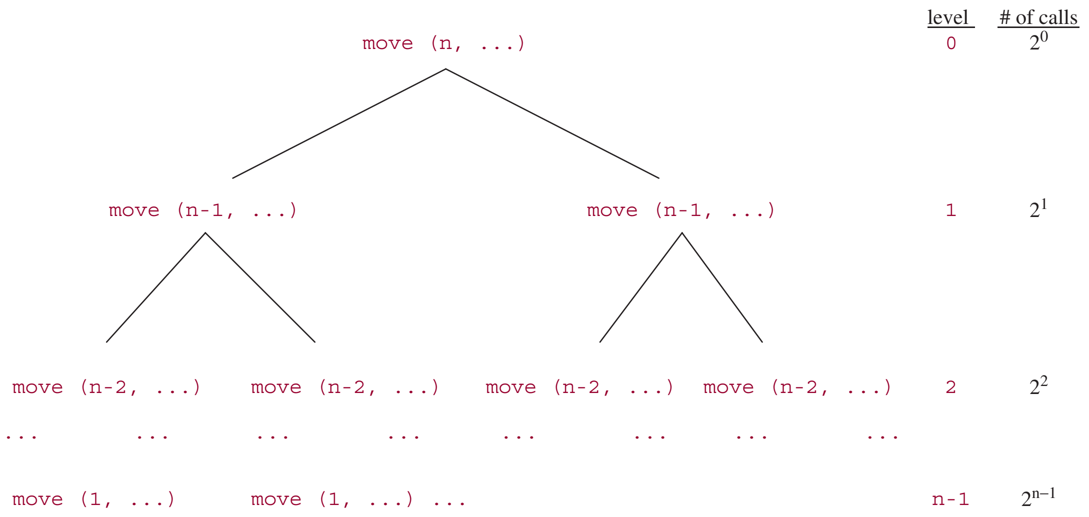


FIGURE 5.4 The relationship between level and number of calls to the `move` method in the tree from Figure 5.3

5.5 Searching an Array

Suppose you want to search an n-element array for an element. We assume that the element class implements the `Comparable<T>` interface (in java.lang):

```
public interface Comparable<T>
{
      /**
       *  Returns an int less than, equal to or greater than 0, depending on
       *  whether the calling object is less than, equal to or greater than a
       *  specified object.
       *
       *  @param obj – the specified object that the calling object is compared to.
       *
       *  @return an int value less than, equal to, or greater than 0, depending on
       *          whether the calling object is less than, equal to, or greater than
       *          obj, respectively.
       *
       *  @throws ClassCastException – if the calling object and obj are not in the
       *          same class.
       *
       */
       public int compareTo(T obj)
} // interface Comparable<T>
```

For example, the `String` class implements the `Comparable<String>` interface, so we can write the following:

```
String s = "elfin";

System.out.println (s.compareTo ("elastic"));
```

The output will be greater than 0 because “elfin” is lexicographically greater than “elastic”; in other words, “elfin” comes after “elastic” according to the Unicode values of the characters in those two strings. Specifically, the ‘f’ in “elfin” comes after the ‘a’ in “elastic”.

The simplest way to conduct the search is sequentially: start at the first location, and keep checking successively higher locations until either the element is found or you reach the end of the array. This search strategy, known as a ***sequential search***, is the basis for the following generic algorithm (that is, static method):

```
/**
 * Determines whether an array contains an element equal to a given key.
 * The worstTime(n) is O(n).
 *
 * @param a the array to be searched.
 * @param key the element searched for in the array a.
 *
 * @return the index of an element in a that is equal to key, if such an element
 *         exists; otherwise, -1.
 *
 * @throws ClassCastException, if the element class does not implement the
 *         Comparable interface.
 *
 */
 public static int sequentialSearch (Object[ ] a, Object key)
 {
   for (int i = 0; i < a.length; i++)
        if (((Comparable) a [i]).compareTo (key) == 0)
              return i;
   return -1;
} // sequentialSearch
```

Because the element type of the array parameter is `Object`, the element type of the array argument can be any type. But within the `sequentialSearch` method, the compiler requires that `a [i]` must be cast to a type that implements the `Comparable<Object>` interface. For the sake of simplicity, we use the "raw" type `Comparable` instead of the equivalent `Comparable<Object>`.

The `sequentialSearch` method is not explicitly included in the Java Collections Framework. But it is the basis for several of the method definitions in the `ArrayList` and `LinkedList` classes, which are in the framework.

For an unsuccessful sequential search of an array, the entire array must be scanned. So both worstTime(n) and averageTime(n) are linear in n for an unsuccessful search. For a successful sequential search, the entire array must be scanned in the worst case. In the average case, assuming each location is equally likely to house the element sought, we probe about $n/2$ elements. We conclude that for a successful search, both worstTime(n) and averageTime(n) are also linear in n.

Can we improve on these times? Definitely. In this section we will develop an array-based search technique for which worstTime(n) and averageTime(n) are only logarithmic in n. And in Chapter 14, we will encounter a powerful search technique—hashing—for which averageTime(n) is constant, but worstTime(n) is still linear in n.

Given an array to be searched and a value to be searched for, we will develop a ***binary search***, so called because the size of the region searched is divided by two at each stage until the search is completed. Initially, the first index in the region is index 0, and the last index is at the end of the array. One important restriction is this: *A binary search requires that the array be sorted*.

We assume, as above, that the array's element class implements the `Comparable` interface.

Here is the method specification, identical to one in the `Arrays` class in the package `java.util`:

```
/**
 * Searches the specified array for the specified object using the binary
 * search algorithm.  The array must be sorted into ascending order
 * according to the <i>natural ordering</i> of its elements (as by
 * <tt>Sort(Object[ ]</tt>), above) prior to making this call.  If it is
 * not sorted, the results are undefined.  If the array contains multiple
 * elements equal to the specified object, there is no guarantee which
 * one will be found.  The worstTime(n) is O(log n).
 *
 * @param a the array to be searched.
 * @param key the value to be searched for.
 *
 * @return index of the search key, if it is contained in the array;
 *       otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
 *       <i>insertion point</i> is defined as the point at which the
 *       key would be inserted into the array: the index of the first
 *       element greater than the key, or <tt>a.length</tt>, if all
 *       elements in the array are less than the specified key. Note
 *        that this guarantees that the return value will be >= 0 if
 *        and only if the key is found.
 *
 * @throws ClassCastException if the array contains elements that are not
 *       <i>mutually comparable</i> (for example, strings and integers),
 *      or the search key in not mutually comparable with the elements
 *      of the array.
 * @see Comparable
 * @see #sort(Object[ ])
 */
public static int binarySearch (Object[ ] a, Object key)
```

In javadoc, the html tag <tt> signifies code, <i> signifies italics, and &gt; signifies the greater than symbol, '>'. The symbol '>' by itself would be interpreted as part of an html tag. The "#" in one of the @see lines creates a link to the given sort method in the document generated through javadoc; that line expands to

See Also:

`sort(Object[])`

The `BinarySearchTest` class is available from the book's website, and includes the following test (`names` is the array of `String` elements from Figure 5.5):

```
@Test
public void binarySearchTest6()
{
    assertEquals (-11, binarySearch (names, "Joseph"));
} // method binarySearchTest6
```

For the sake of utilizing recursion, we will focus on the first and last indexes in the region being searched. Initially, `first` = 0 and `last` = `a.length - 1`. So the original version of `binarySearch` will be a wrapper that simply calls

```
return binarySearch (a, 0, a.length - 1, key);
```

The corresponding method heading is

```
public static int binarySearch (Object[ ] a, int first, int last, Object key)
```

For defining this version of the `binarySearch` method, the basic strategy is this: We compare the element at the middle index of the current region to the key sought. If the middle element is less than the key, we recursively search the array from the middle index + 1 to index `last`. If the middle element is greater than the key, we recursively search the array from index `first` to the middle index − 1. If the middle element is equal to the key, we are done.

Assume, for now, that `first <=  last`. Later on we'll take care of the case where `first > last`. Following the basic strategy given earlier, we start by finding the middle index:

```
int mid = (first + last) >> 1;
```

The right-hand-side expression uses the right-shift bitwise operator, >>, to shift the binary representation of `(first + last)` to the right by 1. This operation is equivalent to, but executes faster than

```
int mid = (first + last) / 2;
```

The middle element is at index mid in the array a. We need to compare (the element referenced by) a [mid] to (the element referenced by) key. The compareTo method is ideal for the comparison, but that method is not defined in the element class, Object. Fortunately, the compareTo method is defined in any class that implements the Comparable interface. So we cast a [mid] to a Comparable object and then call the method compareTo:

```
Comparable midVal = (Comparable)a [mid];
int comp = midVal.compareTo (key);
```

If the result of this comparison is <0, perform a binary search on the region from mid + 1 to last and return the result of that search. That is:

```
if  (comp < 0)
      return binarySearch (a, mid + 1, last, key);
```

Otherwise, if comp > 0, perform a binary search on the region from first to mid - 1 and return the result. That is,

```
if (comp > 0)
     return binarySearch (a, first, mid - 1, key);
```

Otherwise, return mid, because comp == 0 and so a [mid] is equal to key.

For example, let's follow this strategy in searching for "Frank" in the array names shown in Figure 5.5. That figure shows the state of the program when the binarySearch method is called to find "Frank".

The assignment:

```
mid = (first + last) >> 1;
```

gives mid the value (0 + 9)/2, which is 4.

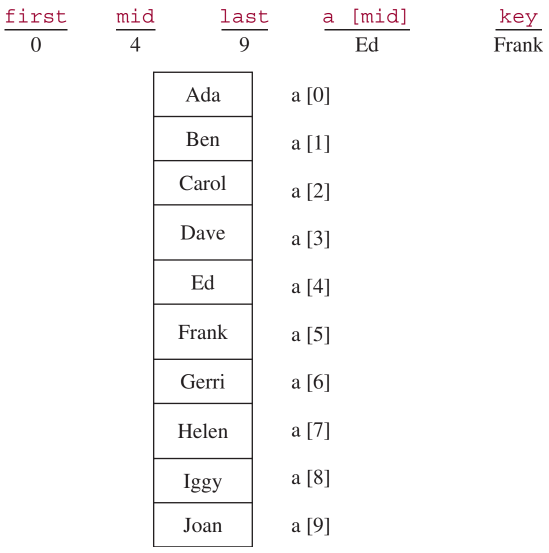


FIGURE 5.5 The state of the program at the beginning of the method called binarySearch (names, 0, 9, "Frank"). The parameter list is Object[ ] a, int first, int last and Object key. (For simplicity, we pretend that names is an array of Strings rather than an array of references to Strings)

The middle element, “Ed”, is less than “Frank”, so we perform a binary search of the region from `mid + 1` to `last`. The call is

```
binarySearch (a, mid + 1, last, key);
```

The parameter `first` gets the value of the argument `mid + 1`. During this execution of `binarySearch`, the assignment

```
mid = (first + last) >> 1;
```

gives `mid` the value (5 + 9)/2, which is 7, so `midVal` is “Helen”. See Figure 5.6.

The middle element, “Helen”, is greater than “Frank”, so a binary search is performed on the region from indexes 5 through 6. The call is

```
binarySearch (a, first, mid - 1, key);
```

The parameter `last` gets the value of the argument `mid - 1`. During this execution of `binarySearch`, the assignment

```
mid = (first + last) >> 1;
```

gives mid the value (5 + 6)/2, which is 5, so the middle element is “Frank”. See Figure 5.7.

Success! The middle element is equal to `key`, so the value returned is `mid`, the index of the middle element.

The only unresolved issue is what happens if the array does not have an element equal to `key`. In that case, we want to return `-insertion Point - 1`, where `insertionPoint` is the index where key could be inserted without disordering the array. The reason we don’t return `-insertionPoint` is that we would have an ambiguity if `insertionPoint` were equal to 0: a return of 0 could be interpreted as the index where `key` was found.

How can we determine what value to give `insertionPoint`? If `first > last` initially, we must have an empty region, with `first = 0` and `last = -1`, so `insertionPoint` should get the value of

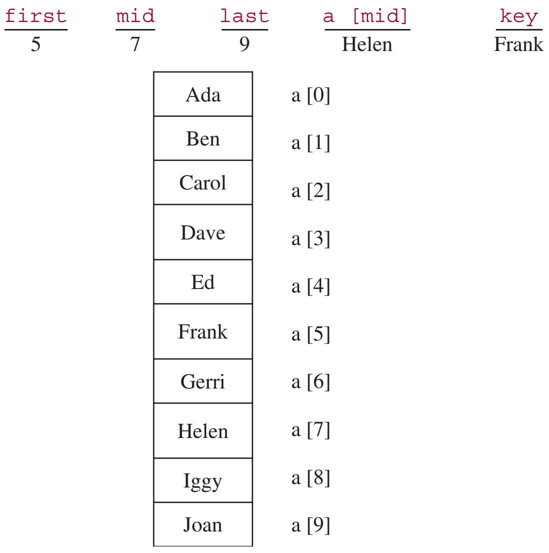


FIGURE 5.6 The state of the program at the beginning of the binary search for “Frank” in the region from indexes 5 through 9

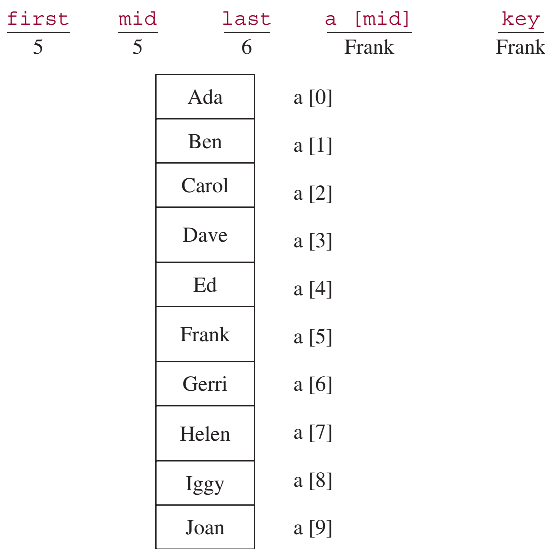


FIGURE 5.7 The state of the program at the beginning of the binary search for "Frank" in the region from indexes 5 through 6

`first`. Otherwise we must have `first <= last` during the first call to `binarySearch`. Whenever `first <= last` at the beginning of a call to `binarySearch`, we have

```
first <= mid <= last
```

So `mid + 1 < = last + 1` and `first - 1 < = mid - 1`.

If `comp < 0`, we call

```
binarySearch (a, mid + 1, last, key);
```

At the beginning of that call, we have

```
first <= last + 1
```

On the other hand, if `comp > 0`, we call

```
binarySearch (a, first, mid - 1, key);
```

At the beginning of that call, we have

```
first - 1 <= last
```

In either case, at the beginning of the call to `binarySearch`, we have

```
first <= last + 1
```

So when we finally get `first > last`, we must have

```
first = last + 1
```

But any element with an index less than `first` must be less than `key`, and any element with an index greater than `last` must be greater than `key`, so when we finish, `first` is the smallest index of any element greater than `key`. That is where `key` should be inserted.

Here is the complete definition:

```
public static  int binarySearch(Object[ ] a, int first, int last, Object key)
{
     if (first <= last)
     {
         int mid = (first + last) >> 1;
         Comparable midVal = (Comparable)a [mid];
         int comp = midVal.compareTo (key);
         if (comp < 0)
             return binarySearch (a, mid + 1, last, key);
         if (comp > 0)
             return binarySearch (a, first, mid - 1, key);
         return mid;  // key found
     } // if first <= last
     return -first - 1; // key not found; belongs at a[first]
} // method binarySearch
```

Here is a `BinarySearchUser` class that allows an end-user to enter names for which a given array will be searched binarily:

```
public class BinarySearchUser
{
    public static void main (String[ ] args)
    {
           new BinarySearchUser ().run();
    } // method main

    public void run()
    {
          final String ARRAY_MESSAGE =
                  "The array on which binary searches will be performed is:\n" +
                  "Ada, Ben, Carol, Dave, Ed, Frank, Gerri, Helen, Iggy, Joan";
          final String SENTINEL = "***";

          final String INPUT_PROMPT =
                  "\n\nPlease enter a name to be searched for in the array (or " +
                  SENTINEL + " to quit): ";

          final String[ ] names = {"Ada", "Ben", "Carol", "Dave", "Ed", "Frank",
                  "Gerri", "Helen", "Iggy", "Joan"};

          final String FOUND = "That name was found at index ";

          final String NOT_FOUND = "That name was not found, but could be " +
                  "inserted at index ";

          String name;

          Scanner sc = new Scanner (System.in);

          int index;

          System.out.println (ARRAY_MESSAGE);
```

```
            while (true)
            {
                System.out.print (INPUT_PROMPT);
                name = sc.next();
                if (name.equals(SENTINEL))
                    break;
                index = binarySearch (names, 0, names.length - 1, name);
                if (index >= 0)
                    System.out.println (FOUND + index);
                else
                    System.out.println (NOT_FOUND + (-index - 1));
            } // while
    } // method run
    public static  int binarySearch(Object[ ] a, int first, int last, Object key)
    {
        if (first <= last)
        {
            int mid = (first + last) >> 1;
            Comparable midVal = (Comparable)a [mid];
            int comp = midVal.compareTo (key);
            if (comp < 0)
                return binarySearch (a, mid + 1, last, key);
            if (comp > 0)
                return binarySearch (a, first, mid - 1, key);
            return mid;  // key found
        } // if first <= last
        return -first - 1; // key not found; belongs at a[first]
      } // method binarySearch

    } // class BinarySearchUser
```

Here is a step-by-step, execution-frame trace of the `binarySearch` method after an initial call of

```
binarySearch (names, 0, 9, "Dan");
```

Note that “Dan” is not in the array `names`.

Step 0:

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
     "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 9
key = "Dan"
mid = 4
midVal = "Ed"
comp is > 0

return binarySearch (a, 0, 3, "Dan");
```

Frame 0

Step 1:

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
       "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 3
key = "Dan"
mid = 1
midVal = "Ben"
comp is < 0

return binarySearch (a, 2, 3, "Dan");
```

Frame 1

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
       "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 9
key = "Dan"
mid = 4
midVal = "Ed"
comp is > 0

return binarySearch (a, 0, 3, "Dan");
```

Frame 0

Step 2:

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
       "Gerri", "Helen", "Iggy", "Joan"]

first = 2
last = 3
key = "Dan"
mid = 2
midVal = "Carol"
comp is < 0

return binarySearch (a, 3, 3, "Dan");
```

Frame 2

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
 "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 3
key = "Dan"
mid = 1
midVal = "Ben"
comp is < 0

return binarySearch (a, 2, 3, "Dan");
```

Frame 1

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
 "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 9
key = "Dan"
mid = 4
midVal = "Ed"
comp is > 0

return binarySearch (a, 0, 3, "Dan");
```

Frame 0

Step 3:

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
 "Gerri", "Helen", "Iggy", "Joan"]

first = 3
last = 3
key = "Dan"
mid = 3
midVal = "Dave"
comp is > 0

return binarySearch (a, 3, 2, "Dan");
```

Frame 3

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
 "Gerri", "Helen", "Iggy", "Joan"]

first = 2
last = 3
key = "Dan"
mid = 2
midVal = "Carol"
comp is < 0

return binarySearch (a, 3, 3, "Dan");
```

Frame 2

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
 "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 3
key = "Dan"
mid = 1
midVal = "Ben"
comp is < 0

return binarySearch (a, 2, 3, "Dan");
```

Frame 1

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
 "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 9
key = "Dan"
mid = 4
midVal = "Ed"
comp is > 0

return binarySearch (a, 0, 3, "Dan");
```

Frame 0

Step 4:

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
    "Gerri", "Helen", "Iggy", "Joan"]

first = 3
last = 2
key = "Dan"

return –3–1;
```

Frame 4

–4

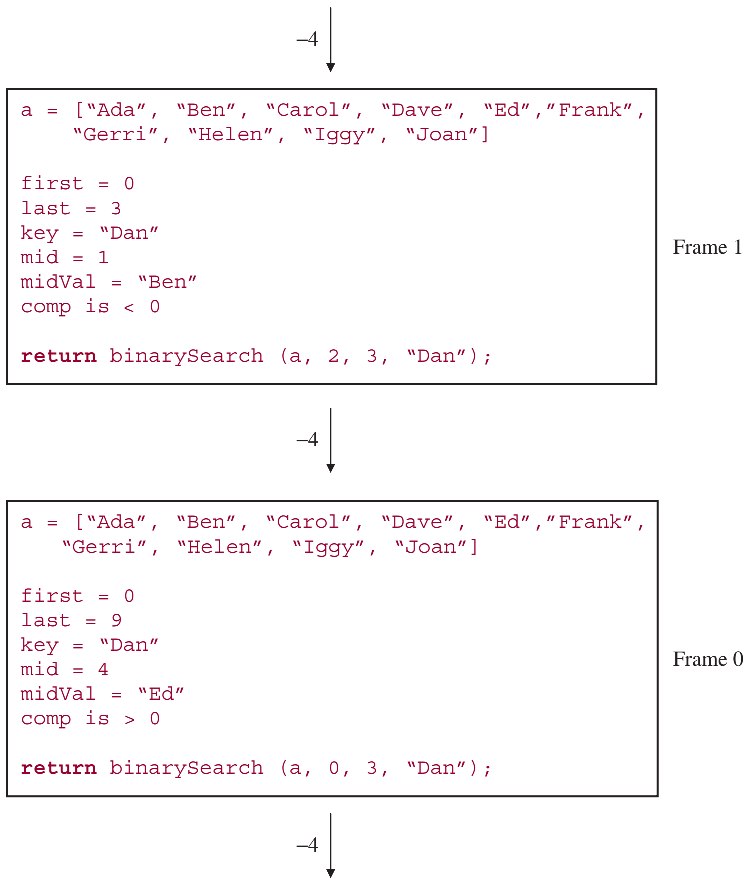
```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
    "Gerri", "Helen", "Iggy", "Joan"]

first = 3
last = 3
key = "Dan"
mid = 3
midVal = "Dave"
comp is > 0

return binarySearch (a, 3, 2, "Dan");
```

Frame 3

–4

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
    "Gerri", "Helen", "Iggy", "Joan"]

first = 2
last = 3
key = "Dan"
mid = 2
midVal = "Carol"
comp is < 0

return binarySearch (a, 3, 3, "Dan");
```

Frame 2



```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
    "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 3
key = "Dan"
mid = 1
midVal = "Ben"
comp is < 0

return binarySearch (a, 2, 3, "Dan");
```

```
a = ["Ada", "Ben", "Carol", "Dave", "Ed","Frank",
   "Gerri", "Helen", "Iggy", "Joan"]

first = 0
last = 9
key = "Dan"
mid = 4
midVal = "Ed"
comp is > 0

return binarySearch (a, 0, 3, "Dan");
```

How long does the `binarySearch` method take? We need to make a distinction between an unsuccessful search, in which the element is not found, and a successful search, in which the element is found. We start with an analysis of an unsuccessful search.

During each execution of the `binarySearch` method in which the middle element is not equal to `key`, the size of the region searched during the next execution is, approximately, halved. If the element sought is not in the array, we keep dividing by 2 as long as the region has at least one element. Let n represent the size of the region. The number of times n can be divided by 2 until $n = 0$ is logarithmic in n—this is, basically, the Splitting Rule from Chapter 3. So for a failed search, worstTime(n) is logarithmic in n. Since we are assuming the search is unsuccessful, the same number of searches will be performed in the average case as in the worst case, so averageTime(n) is logarithmic in n for a failed search.

The worst case for a successful search requires one less call to the `binarySearch` method than the worst case (or average case) for an unsuccessful search. So for a successful search, worstTime(n) is still logarithmic in n. In the average case for a successful search, the analysis—see Concept Exercise 5.15—is more complicated, but the result is the same: averageTime(n) is logarithmic in n.

During each call, a constant amount of information is saved: the entire array is not saved, only a reference to the array. So the space requirements are also logarithmic in n, for both successful and unsuccessful searches and for both the worst case and the average case.

In the `Arrays` class of the `java.util` package, there is an iterative version of the binary search algorithm. In Lab 8, you will conduct an experiment to compare the time to recursively search an array of **`int`** s, iteratively search an array of **`int`** s, and iteratively search an array of `Integer` s. Which of the three do you think will be slowest?

You are now prepared to do Lab 8: Iterative Binary Search

Lab 9 introduces another recursive method whose development is far easier than its iterative counterpart. The method for generating permutations is from Roberts' delightful book, *Thinking Recursively* [Roberts, 1986].

You are now prepared to do Lab 9: Generating Permutations

Section 5.6 deals with another design pattern (a general strategy for solving a variety of problems): backtracking. You have employed this strategy whenever you had to re-trace your steps on the way to some goal. The `BackTrack` class also illustrates the value of using interfaces.

5.6 Backtracking

The basic idea with backtracking is this: From a given starting position, we want to reach a goal position. We repeatedly choose, maybe by guessing, what our next position should be. If a given choice is valid—that is, the new position might be on a path to the goal—we advance to that new position and continue. If a choice leads to a dead end, we back up to the previous position and make another choice. ***Backtracking*** is the strategy of trying to reach a goal by a sequence of chosen positions, with a re-tracing in reverse order of positions that cannot lead to the goal.

For example, look at the picture in Figure 5.8. We start at position P0 and we want to find a path to the goal state, P14. We are allowed to move in only two directions: north and west. But we cannot "see" any farther than the next position. Here is a strategy: From any position, we first try to go north; if we are unable to go north, we try to go west; if we are unable to go west, we back up to the most recent position where we chose north and try to choose west instead. We never re-visit a position that has been discovered to be a dead end. The positions in Figure 5.8 are numbered in the order they would be tried according to this strategy.

Figure 5.8 casts some light on the phrase "re-tracing in reverse order." When we are unable to go north or west from position P4, we first back up to position P3, where west is not an option. So we back up to P2. Eventually, this leads to a dead end, and we back up to P1, which leads to the goal state.

When a position is visited, it is marked as possibly being on a path to the goal, but this marking must be undone if the position leads only to a dead end. That enables us to avoid re-visiting any dead-end position. For example, in Figure 5.8, P5 is not visited from P8 because by the time we got to P8, P5 had already been recognized as a dead end.

We can now refine our strategy. To try to reach a goal from a given position, enumerate over all positions directly accessible from the given position, and keep looping until either a goal has been reached or we can no longer advance to another position. During each loop iteration, get the next accessible position. If that position may be on a path to a goal, mark that position as possibly leading to a goal and, if it is a goal, the search has been successful; otherwise, attempt to reach a goal from that position, and mark the position as a dead end if the attempt fails.

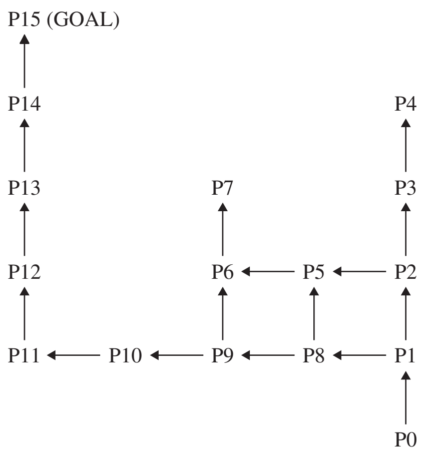


FIGURE 5.8 Backtracking to obtain a path to a goal. The solution path is P0, P1, P8, P9, P10, P11, P12, P13, P14, P15

Make sure you have a good understanding of the previous paragraph before you proceed. That paragraph contains the essence of backtracking. The rest of this section and Section 5.6.1 are almost superfluous by comparison.

Instead of developing a backtracking method for a particular application, we will utilize a generalized backtracking algorithm from Wirth [1976, p.138]. We then demonstrate that algorithm on a particular application, maze searching. Four other applications are left as programming projects in this chapter. And Chapter 15 has another application of backtracking: a programming project for searching a network. Backtracking is a design pattern because it is a generic programming technique that can be applied in a variety of contexts.

The `BackTrack` class below is based on one in Noonan [2000]. The details of the application class will be transparent to the `BackTrack` class, which works through an interface, `Application`. The `Application` interface will be implemented by the particular application.

A user (of the `BackTrack` class) supplies:

- the class implementing the `Application` interface (note: to access the positions available from a given position, the iterator design-pattern is employed, with a nested iterator class);
- a `Position` class to define what "position" means for this application;

The `Application` methods are generalizations of the previous outline of backtracking. Here is the `Appli cation` interface:

```
import java.util.*;

public interface Application
{
      /**
       * Determines if a given position is legal and not a dead end.
       *
       * @param pos - the given position.
       *
       * @return true if pos is a legal position and not a dead end.
       */
       boolean isOK (Position pos);
```

```
        /**
         * Indicates that a given position is possibly on a path to a goal.
         *
         * @param pos the position that has been marked as possibly being on a
         *                path to a goal.
         */
         void markAsPossible (Position pos);


        /**
         * Indicates whether a given position is a goal position.
         *
         * @param pos the position that may or may not be a goal position.
         *
         * @return true if pos is a goal position; false otherwise.
         */
         boolean isGoal (Position pos);


        /**
         * Indicates that a given position is not on any path to a goal position.
         *
         * @param pos the position that has been marked as not being on any path to
         *                a goal position.
         */
         void markAsDeadEnd (Position pos);


         /**
         * Converts this Application object into a String object.
         *
         * @return the String representation of this Application object.
         */
         String toString();


        /**
         * Produces an Iterator object that starts at a given position.
         *
         * @param pos the position the Iterator object starts at.
         *
         * @return an Iterator object that accesses the positions directly
         *               available from pos.
         */
         Iterator<Position> iterator (Position pos);

    } // interface Application
```

The BackTrack class has two responsibilities: to initialize a BackTrack object from a given application object, and to try to reach a goal position from a given position. The method specifications are

```
    /**
     * Initializes this BackTrack object from an application.
     *
     * @param app the application
     */
     public BackTrack (Application app)

    /**
     * Attempts to reach a goal through a given position.
     *
     * @param pos the given position.
     *
     * @return true if the attempt succeeds; otherwise, false.
     */
     public boolean tryToReachGoal (Position pos)
```

The only field needed is (a reference to) an `Application`. The definition of the constructor is straightforward. The definition of the `tryToReachGoal` method is based on the outline of backtracking given above: To "enumerate over all positions accessible from the given position," we create an iterator. The phrase "attempt to reach a goal from that position" becomes a recursive call to the method `tryToReachGoal`. The complete `BackTrack` class, without any application-specific information, is as follows:

```
import java.util.*;

public class BackTrack
{
  protected Application app;


  /**
   * Initializes this BackTrack object from an application.
   *
   * @param app the application
   */
   public BackTrack (Application app)
   {
     this.app = app;
   } // constructor


  /**
   * Attempts to reach a goal through a given position.
   *
   * @param pos the given position.
   *
   * @return true if the attempt succeeds; otherwise, false.
   */
   public boolean tryToReachGoal (Position pos)
   {
     Iterator<Position> itr = app.iterator (pos);

     while (itr.hasNext())
     {
```

```
     pos = itr.next();
     if (app.isOK (pos))
     {
       app.markAsPossible (pos);
       if (app.isGoal (pos) || tryToReachGoal (pos))
         return true;
       app.markAsDeadEnd (pos);
     } // pos may be on a path to a goal
   } // while
   return false;
 } // method tryToReachGoal

} // class BackTrack
```

Let's focus on the `tryToReachGoal` method, the essence of backtracking. We look at the possible choices of moves from the `pos` parameter. There are three possibilities:

1. One of those choices is a goal position. Then **`true`** is returned to indicate success.
2. One of those choices is valid but not a goal position. Then another call to `tryToReachGoal` is made, starting at the valid choice.
3. None of the choices is valid. Then the **`while`** loop terminates and **`false`** is returned to indicate failure to reach a goal position from the current position.

The argument to `tryToReachGoal` represents a position that has been marked as possibly being on a path to a goal position. Whenever a return is made from `tryToReachGoal`, the pre-call value of `pos` is restored, to be marked as a dead end if it does not lead to a goal position.

Now that we have developed a framework for backtracking, it is straightforward to utilize this framework to solve a variety of problems.

5.6.1 An A-maze-ing Application

For one application of backtracking, let's develop a program to try to find a path through a maze. For example, Figure 5.9 has a 7-by-13 maze, with a 1 representing a corridor and a 0 representing a wall. The only valid moves are along a corridor, and only horizontal and vertical moves are allowed; diagonal moves are prohibited. The starting position is in the upper left-hand corner and the goal position is in the lower-right-hand corner.

```
1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 1 0 1 0 1 0 1 0 1
1 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 1 1 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1
```

FIGURE 5.9 A maze: 1 represents a corridor and 0 represents a wall. Assume the starting position is in the upper left-hand corner, and the goal position is in the lower right-hand corner

A successful traversal of this maze will show a path leading from the start position to the goal position. We mark each such position with the number 9. Because there are two possible paths through this maze, the actual path chosen will depend on how the iterator class orders the possible choices. For the sake of specificity, assume the order of choices is north, east, south, and west. For example, from the position at coordinates (5, 8), the first choice would be (4, 8), followed by (5, 9), (6, 8), and (5, 7).

From the initial position at (0, 0), the following positions are recorded as possibly being on a solution-path:

```
(0, 1) // moving east
(0, 2) // moving east
(1, 2) // moving south
(1, 3) // moving east
(1, 4) // moving east
(0, 4) // moving north
(0, 5) // moving east;
```

This last position is a dead end, so we "undo" (0, 5) and (0, 4), backtrack to (1, 4) and then record the following as possibly leading to the goal:

```
(2, 4) // moving south
(3, 4) // moving south
(3, 5) // moving east;
```

From here we eventually reach a dead end. After we undo (3, 5) and re-trace to (3, 4), we advance—without any further backtracking—to the goal position. Figure 5.10 uses 9' to show the corresponding path through the maze of Figure 5.9, with dead-end positions marked with 2's.

For this application, a position is simply a pair: row, column. The `Position` class is easily developed:

```
public class Position
{
       protected int row,
                     column;


       /**
       * Initializes this Position object to (0, 0).
       */
       public Position ()
       {
             row = 0;
```

```
9 9 9 0 2 2 0 0 0 2 2 2 2
1 0 9 9 9 0 2 2 2 2 2 0 2
1 0 0 0 9 0 2 0 2 0 2 0 2
1 0 0 0 9 2 2 0 2 0 2 2 2
1 1 1 1 9 0 0 0 0 1 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 0
0 0 0 0 9 9 9 9 9 9 9 9 9
```

FIGURE 5.10 A path through the maze of Figure 5.9. The path positions are marked with 9's and the dead-end positions are marked with 2's

```
            column = 0;
        } // default constructor
       /**
        * Initializes this Position object to (row, column).
        *
        * @param row the row this Position object has been initialized to.
        * @param column the column this Position object has been initialized to.
        */
        public Position (int row, int column)
        {
              this.row = row;
              this.column = column;
        } // constructor
       /**
        * Determines the row of this Position object.
        *
        * @return the row of this Position object.
        */
        public int getRow ()
        {
              return row;
        } // method getRow
       /**
        * Determines the column of this Position object.
        *
        * @return the column of this Position object.
        */
        public int getColumn ()
        {
              return column;
        } // method getColumn

} // class Position
```

For this application, the `Application` interface is implemented in a `Maze` class. The only fields are a grid to hold the maze and start and finish positions. Figure 5.11 has the UML diagrams for the `Maze` class and `Application` interface.

Except for the `Maze` class constructor and the three accessors (`getGrid` was developed for the sake of testing), the method specifications for the `Maze` class are identical to those in the `Application` interfaces given earlier. For the embedded `MazeIterator` class, the constructor's specification is provided, but the method specifications for the `hasNext`, `next` and `remove` methods are boilerplate, so we need not list them. Here are the specifications for the `Maze` and `MazeIterator` constructors:

```
/**
 * Initializes this Maze object from a file scanner over a file.
 *
 * @param fileScanner - the scanner over the file that holds the
 *           maze information.
 *
 * @throws InputMismatchException - if any of the row or column values are non-
 *                  integers, or if any of the grid entries are non-integers.
 * @throws NumberFormatException - if the grid entries are integers but neither
 *                               WALL nor CORRIDOR
```

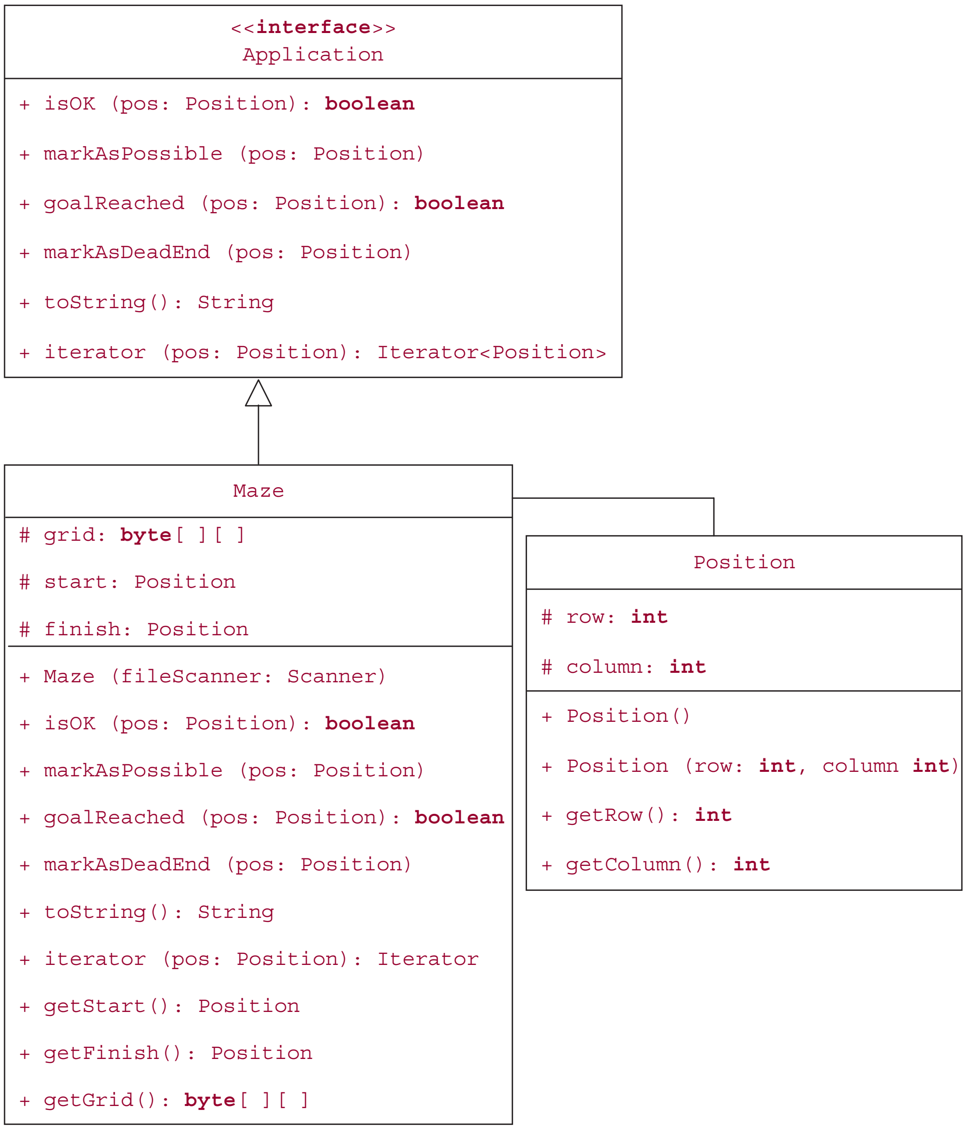

<<interface>>
Application
+ isOK (pos: Position): boolean
+ markAsPossible (pos: Position)
+ goalReached (pos: Position): boolean
+ markAsDeadEnd (pos: Position)
+ toString(): String
+ iterator (pos: Position): Iterator<Position>
Maze
# grid: byte[ ][ ]
# start: Position
# finish: Position
+ Maze (fileScanner: Scanner)
+ isOK (pos: Position): boolean
+ markAsPossible (pos: Position)
+ goalReached (pos: Position): boolean
+ markAsDeadEnd (pos: Position)
+ toString(): String
+ iterator (pos: Position): Iterator
+ getStart(): Position
+ getFinish(): Position
+ getGrid(): byte[ ][ ]
Position
# row: int
# column: int
+ Position()
+ Position (row: int, column int)
+ getRow(): int
+ getColumn(): int

FIGURE 5.11 The class diagram for the `Maze` class, which implements the `Application` interface and has `grid`, `start`, and `finish` fields

```
    */
    public Maze (Scanner fileScanner)

   /**
    * Initializes this MazeIterator object to start at a given position.
    *
    * @param pos the position the Iterator objects starts at.
    */
    public MazeIterator (Position pos)
```

The MazeTest class, available on the book's website, starts by declaring a maze field and then creating a maze (the one shown in Figure 5.9) from a file:

```
protected Maze maze;

@Before
public void runBeforeEachTest()  throws IOException
{
  fileScanner = new Scanner (new File ("maze.txt"));
  maze = new Maze (fileScanner);
} // method runBeforeEachTest
```

Here are four of the boundary-condition tests of the isOK method:

```
@Test
public void isOKTest1()
{
  Position pos = new Position (0, 0);
  assertEquals (true, maze.isOK (pos));
} // isOKTest1

@Test
public void isOKTest2()
{
  Position pos = new Position (6, 12);
  assertEquals (true, maze.isOK (pos));
} // isOKTest2

@Test
public void isOKTest3()
{
  Position pos = new Position (7, 0);
  assertEquals (false, maze.isOK (pos));
} // isOKTest3

@Test
public void isOKTest4()
{
  Position pos = new Position (0, 13);
  assertEquals (false, maze.isOK (pos));
} // isOKTest4
```

Here is the complete Maze class, including the embedded MazeIterator class:

```
import java.util.*;

public class Maze implements Application
{
       public static final byte WALL = 0;
       public static final byte CORRIDOR = 1;
       public static final byte PATH = 9;
       public static final byte DEAD_END = 2;

       protected Position start,
                          finish;
```

```
protected byte[ ][ ] grid;

/**
 * Initializes this Maze object from a file scanner over a file.
 *
 * @param fileScanner - the scanner over the file that holds the
 *             maze information.
 *
 * @throws InputMismatchException - if any of the row or column values are non-
 *                   integers, or if any of the grid entries are non-integers.
 * @throws NumberFormatException - if the grid entries are integers but neither
 *                                   WALL nor CORRIDOR
 */
 public Maze (Scanner fileScanner)
 {
   int rows = fileScanner.nextInt(),
      columns = fileScanner.nextInt();

   grid = new byte [rows][columns];

   start = new Position (fileScanner.nextInt(),
                         fileScanner.nextInt());

   finish = new Position (fileScanner.nextInt(),
                         fileScanner.nextInt());

   for (int i = 0; i < rows; i++)
      for (int j = 0; j < columns; j++)
      {
         grid [i][j] = fileScanner.nextByte();
         if (grid [i][j] != WALL "" grid [i][j] != CORRIDOR)
           throw new NumberFormatException ("At position (" + i + ", " + j + "), " +
                           grid [i][j] + " should be " +
                           WALL + " or " + CORRIDOR + ".");
      } // for j
} // constructor


/**
 * Determines if a given position is legal and not a dead end.
 *
 * @param pos - the given position.
 *
 * @return true if pos is a legal position and not a dead end.
 */
 public boolean isOK (Position pos)
 {
        return pos.getRow() >= 0 "" pos.getRow() < grid.length ""
               pos.getColumn() >= 0 "" pos.getColumn() < grid [0].length ""
               grid [pos.getRow()][pos.getColumn()] == CORRIDOR;
```

```
} // method isOK


/**
 * Indicates that a given position is possibly on a path to a goal.
 *
 * @param pos the position that has been marked as possibly being on a path
 *      to a goal.
 */
 public void markAsPossible (Position pos)
 {
        grid [pos.getRow ()][pos.getColumn ()] = PATH;
 } // method markAsPossible


 /**
  * Indicates whether a given position is a goal position.
  *
  * @param pos the position that may or may not be a goal position.
  *
  * @return true if pos is a goal position; false otherwise.
  */
  public boolean isGoal (Position pos)
  {
         return pos.getRow() == finish.getRow() ""
                pos.getColumn() == finish.getColumn();
  } // method isGoal


 /**
  * Indicates that a given position is not on any path to a goal position.
  *
  * @param pos the position that has been marked as not being on any path to a
  *      goal position.
  */
  public void markAsDeadEnd (Position pos)
  {
         grid [pos.getRow()][pos.getColumn()] = DEAD_END;
  } // method markAsDeadEnd


 /**
  * Converts this Application object into a String object.
  *
  * @return the String representation of this Application object.
  */
  public String toString ()
  {
         String result = "\n";
         result += start.getRow() + " " + start.getColumn() + "\n";
         result += finish.getRow() + " " + finish.getColumn() + "\n";
         for (int row = 0; row < grid.length; row++)
         {
                 for (int column = 0; column < grid [0].length; column++)
                         result += String.valueOf (grid [row][column]) + ' ';
                 result += "\n";
         } // for row = 0
         return result;
  } // method toString


 /**
  * Produces an Iterator object, over elements of type Position, that starts at a given
  * position.
  *
  * @param pos - the position the Iterator object starts at.
  *
  * @return the Iterator object.
  */
  public Iterator<Position> iterator (Position pos)
  {
         return new MazeIterator (pos);
  } // method iterator

 /**
  *   Returns the start position of this maze.
  *
  *  @return - the start position of this maze
  *
  */
  public Position getStart()
  {
         return start;
  } // method getStart

 /**
  *   Returns the finish position of this maze.
  *
  *  @return - the finish position of this maze
  *
  */
  public Position getFinish()
  {
         return finish;
  } // method getFinish

  /**
   * Returns a 2-dimensional array that holds a copy of the maze configuration.
   *
   * @return - a 2-dimensional array that holds a copy of the maze configuration.
   *
   */
```

```
public byte[ ][ ] getGrid()
{
       byte[ ][ ] gridCopy = new byte[grid.length][grid[0].length];

       for (int i = 0; i < grid.length; i++)
            for (int j = 0; j < grid[i].length; j++)
                  gridCopy[i][j] = grid[i][j];

       return gridCopy;
} // method getGrid

protected class MazeIterator implements Iterator<Position>
{

      protected static final int MAX_MOVES = 4;

      protected int row,
                    column,
                    count;

     /**
      * Initializes this MazeIterator object to start at a given position.
      *
      * @param pos the position the Iterator objects starts at.
      */
      public MazeIterator (Position pos)
      {
             row = pos.getRow();
             column = pos.getColumn();
             count = 0;
      } // constructor


     /**
      * Determines if this MazeIterator object can advance to another
      * position.
      *
      * @return true if this MazeIterator object can advance; false otherwise.
      */
      public boolean hasNext ()
      {
             return count < MAX_MOVES;
      } // method hasNext


     /**
      * Advances this MazeIterator object to the next position.
      *
      * @return the position advanced to.
      */
      public Position next ()
      {
             Position nextPosition = new Position();
```

```
                switch (count++)
                {
                    case 0: nextPosition = new Position (row-1, column); // north
                            break;
                    case 1: nextPosition = new Position (row, column+1); // east
                            break;
                    case 2: nextPosition = new Position (row+1, column); // south
                            break;
                    case 3: nextPosition = new Position (row, column-1); // west
                } // switch;
                return nextPosition;
            } // method next


            public void remove ()
            {
                // removal is illegal for a MazeIterator object
                throw new UnsupportedOperationException();
            } // method remove

     } // class MazeIterator

} // class Maze
```

To show how a user might utilize the `Maze` class, we develop a `MazeUser` class. The `MazeUser` class creates a maze from a file scanner. There is a method to search for a path through the maze. The output is either a solution or a statement that no solution is possible. The method specifications (except for the usual `main` method) are

```
     /**
      *  Runs the application.
      */
      public void run()


     /**
      * Searches for a solution path through the maze from the start position
      *
      *
      *  @param maze – the maze to be searched
      *
      *  @return true – if there is a path through the maze; otherwise, false.
      *
      */
      public boolean searchMaze (Maze maze)
```

Figure 5.12 has the UML class diagrams that illustrate the overall design. Because the `Position` class is quite simple and its diagram is in Figure 5.11, its class diagram is omitted.

The implementation of the `MazeUser` class is as follows:

```
import java.io.*;
```

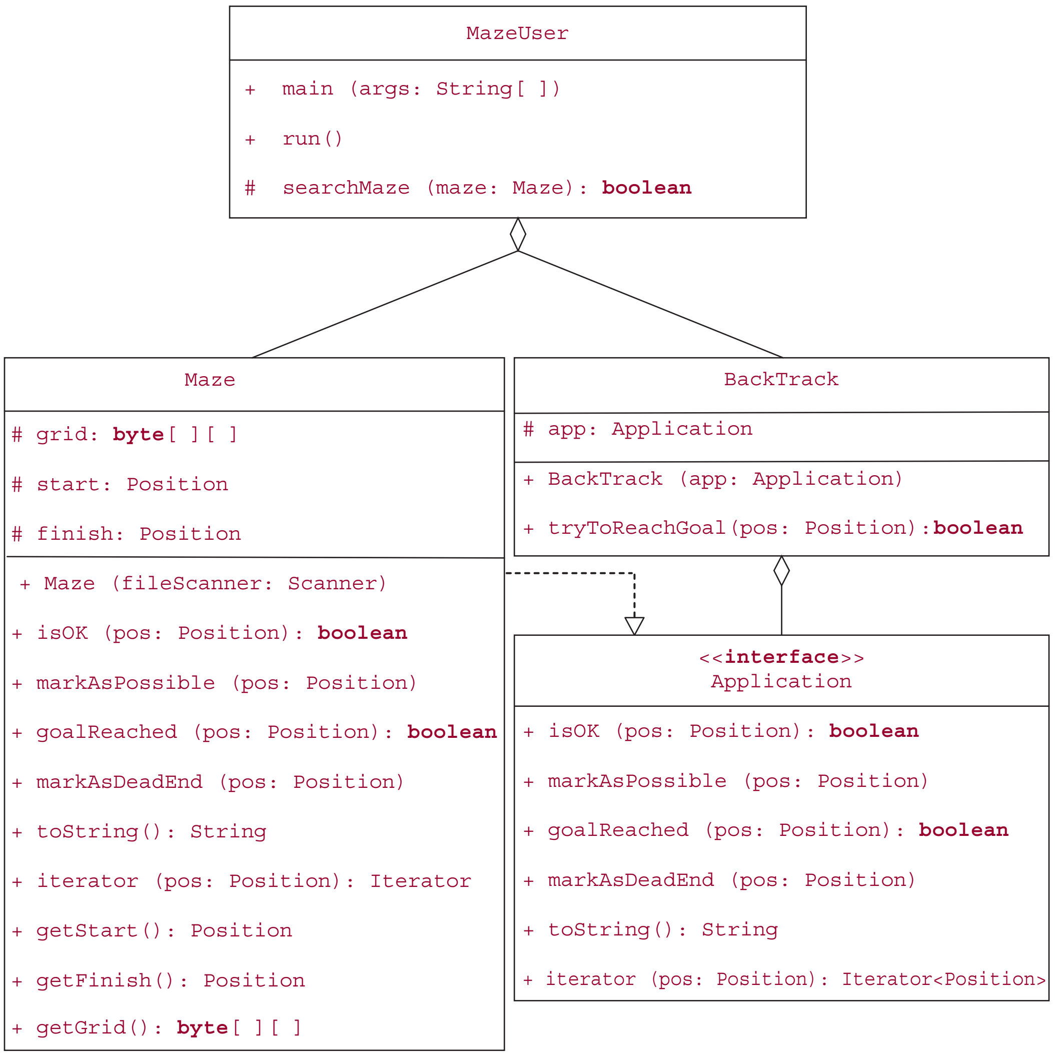

MazeUser
+ main (args: String[ ])
+ run()
# searchMaze (maze: Maze): boolean
Maze
# grid: byte[ ][ ]
# start: Position
# finish: Position
+ Maze (fileScanner: Scanner)
+ isOK (pos: Position): boolean
+ markAsPossible (pos: Position)
+ goalReached (pos: Position): boolean
+ markAsDeadEnd (pos: Position)
+ toString(): String
+ iterator (pos: Position): Iterator
+ getStart(): Position
+ getFinish(): Position
+ getGrid(): byte[ ][ ]
BackTrack
# app: Application
+ BackTrack (app: Application)
+ tryToReachGoal(pos: Position):boolean
<<interface>>
Application
+ isOK (pos: Position): boolean
+ markAsPossible (pos: Position)
+ goalReached (pos: Position): boolean
+ markAsDeadEnd (pos: Position)
+ toString(): String
+ iterator (pos: Position): Iterator<Position>

FIGURE 5.12 The UML class diagrams for the maze-search project

```
import java.util.*;

public class MazeUser
{

  public static void main (String[ ] args)
  {
    new MazeUser().run();
  } // method main

  public void run()
```

```
{
  final String INPUT_PROMPT =
     "\n\nPlease enter the path for the file whose first line contains the " +
     "number of rows and columns,\nwhose 2nd line the start row and column, " +
     "whose 3rd line the finish row and column, and then the maze, row-by-row: ";

  final String INITIAL_STATE =
     "\nThe initial state is as follows (0 = WALL, 1 = CORRIDOR):\n";

  final String START_INVALID = "The start position is invalid.";

  final String FINISH_INVALID = "The finish position is invalid.";

  final String FINAL_STATE =
     "The final state is as follows (2 = DEAD END, 9 = PATH):\n";

  final String SUCCESS = "\n\nA solution has been found:";

  final String FAILURE = "\n\nThere is no solution:";

  Maze maze = null;

  Scanner keyboardScanner = new Scanner (System.in),
          fileScanner = null;

  String fileName;

  while (true)
  {
    try
    {
       System.out.print (INPUT_PROMPT);
       fileName = keyboardScanner.next();
       fileScanner = new Scanner (new File (fileName));
       break;
    } // try
    catch (IOException e)
    {
      System.out.println ("\n" + e);
    } // catch IOException
  } // while
  try
  {
     maze = new Maze (fileScanner);
     System.out.println (INITIAL_STATE + maze);
     Scanner stringScanner = new Scanner (maze.toString());
     Position start = new Position (stringScanner.nextInt(), stringScanner.nextInt()),
              finish = new Position (stringScanner.nextInt(), stringScanner.nextInt());
     if (!maze.isOK (start))
         System.out.println (START_INVALID);
     else if (!maze.isOK (finish))
         System.out.println (FINISH_INVALID);
```

```
        else
        {
           if (searchMaze (maze, start))
               System.out.println (SUCCESS);
           else
               System.out.println (FAILURE);
           System.out.println (FINAL_STATE + maze);
        } // else valid search
     } // try
     catch (InputMismatchException e)
     {
       System.out.println ("\n" + e + ": " + fileScanner.nextLine());
     } // catch InputMismatchException
     catch (NumberFormatException e)
     {
       System.out.println ("\n" + e);
     } // catch NumberFormatException
     catch (RuntimeException e)
     {
       System.out.println ("\n" + e);
       System.out.println (FINAL_STATE + maze);
     } // catch NumberFormatException
  } // method run


  /**
   * Performs the maze search.
   *
   * @param maze - the maze to be searched.
   *
   *  @return true - if there is a path through this maze; otherwise, false
   *
   */
   public boolean searchMaze (Maze maze)
   {
     Position start = maze.getStart();
     maze.markAsPossible (start);
     BackTrack backTrack = new BackTrack (maze);
     if (maze.isGoal (start) || backTrack.tryToReachGoal (start))
        return true;
     maze.markAsDeadEnd (start);
     return false;
   } // method searchMaze

} // class MazeUser
```

In this project, and in general, the `run` method is not tested because it involves end-user input and output. All of the files, including the `Application` interface and the `BackTrack`, `Position`, `Maze`, `MazeTest`, `MazeUser`, and `MazeUserTest` (for the `searchMaze` method) classes, are available from the book's website.

```
1   0   1   1   0   1   1   0
1   0   1   1   0   1   1   0
1   0   1   1   0   1   1   0
.   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .
1   0   1   1   0   1   1   0
1   0   1   1   0   1   1   0
1   1   1   1   1   1   1   1
```

FIGURE 5.13 A worst-case maze: in columns 1, 4, 7, . . . , every row except the last contains a 0; every other position in the maze contains a 1. The start position is in the upper-left corner, and the finish position is in the lower-right corner

How long does the `tryToReachGoal` method in the `BackTrack` class take? Suppose the maze has n positions. In the worst case, such as in Figure 5.13, every position would be considered, so worstTime(n) is linear in n. And with more than half of the positions on a path to the goal position, there would be at least $n/2$ recursive calls to the `tryToReachGoal` method, so worstSpace(n) is also linear in n.

Projects 5.2, 5.3, 5.4, and 5.5 have other examples of backtracking. Because the previous project separated the backtracking aspects from the maze traversing aspects, the `BackTrack` class and `Application` interface are unchanged. The `Position` class for Projects 5.2, 5.3, and 5.5 is the same `Position` class declared earlier, and the `Position` class for Project 5.4 is only slightly different.

We will re-visit backtracking in Chapter 15 in the context of searching a network. And, of course, the `BackTrack` class and `Application` interface are the same as given earlier.

At the beginning of this chapter we informally described a recursive method as a method that called itself. Section 5.7 indicates why that description does not suffice as a definition and then provides a definition.

5.7 Indirect Recursion

Java allows methods to be indirectly recursive. For example, if method A calls method B and method B calls method A, then both A and B are recursive. Indirect recursion can occur in the development of a grammar for a programming language such as Java.

Because indirect recursion is legal, we cannot simply define a method to be recursive if it calls itself. To provide a formal definition of *recursive*, we first define *active*. A method is ***active*** if it is being executed or has called an active method. For example, consider a chain of method calls

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \arrow[r] & D \end{tikzcd}$$

That is, A calls B, B calls C, and C calls D. When D is being executed, the active methods are

D, because it is being executed;

C, because it has called D and D is active;

B, because it has called C and C is active;

A, because it has called B and B is active.

We can now define "recursive." A method is ***recursive*** if it can be called while it is active. For example, suppose we had the following sequence of calls:

$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] & C \arrow[r] & D \arrow[ll, bend left]
\end{tikzcd}$$

Then B, C, and D are recursive because each can be called while it is active.

When a recursive method is invoked, a certain amount of information must be saved so that information will not be written over during the execution of the recursive call. This information is restored when the execution of the method has been completed. This saving and restoring, and other work related to the support of recursion, carry some cost in terms of execution time and memory space. Section 5.8 estimates the cost of recursion, and attempts to determine whether that cost is justified.

5.8 The Cost of Recursion

We have seen that a certain amount of information is saved every time a method calls itself. This information is collectively referred to as an ***activation record*** because it pertains to the execution state of the method that is active during the call. In fact, an activation record is created whenever *any* method is called; this relieves the compiler of the burden of determining if a given method is indirectly recursive.

Essentially, an activation record is an execution frame without the statements. Each activation record contains:

a. the return address, that is, the address of the statement that will be executed when the call has been completed;

b. the value of each argument: a copy of the corresponding argument is made (if the type of the argument is reference-to-object, the reference is copied);

c. the values of the local variables declared within the body of the called method.

After the call has been completed, the previous activation record's information is restored and the execution of the calling method is resumed. For methods that return a value, the value is placed on top of the previous activation record's information just prior to the resumption of the calling method's execution. The calling method's first order of business is to get that return value.

There is an execution-time cost of saving and restoring these records, and the records themselves take up space. But these costs are negligible relative to the cost of a programmer's time to develop an iterative method when a recursive method would be more appropriate. Recursive methods, such as `move`, `tryToReachGoal`, and `permute` (from Lab 9) are far simpler and more elegant than their iterative counterparts.

How can you decide whether a recursive method or iterative method is more appropriate? Basically, if you can readily develop an iterative solution, go for it. If not, you need to decide if recursion is appropriate for the problem. That is, if complex cases of the problem can be reduced to simpler cases of the same form as the original and the simplest case(s) can be solved directly, you should try to develop a recursive method.

If an iterative method is not easy to develop, and recursion is appropriate, how does recursion compare with iteration? At worst, the recursive will take about as long (and have similar time/space performance) as the iterative version. At best, developing the recursive method will take far less time than the iterative version, and have similar time/space performance. See, for example, the `move`, `tryToReachGoal`, and `permute` methods. Of course, it is possible to design an inefficient recursive method, such as the original version of `fib` in Lab 7, just as iterative methods can have poor performance.

In this chapter we have focused on what recursion is. We postpone to Chapter 8 a discussion of the mechanism, called a ***stack***, by which the compiler implements the saving and restoring of activation records. As we saw in Chapter 1, this abstraction—the separation of what is done from how it is done—is critically important in problem solving.

SUMMARY

The purpose of this chapter was to familiarize you with the basic idea of recursion so you will be able to understand the recursive methods in subsequent chapters and to design your own recursive methods when the need arises.

A method is ***recursive*** if it can be called while it is active—an ***active*** method is one that either is being executed or has called an active method.

If an iterative method to solve a problem can readily be developed, then that should be done. Otherwise, recursion should be considered if the problem has the following characteristics:

1. Complex cases of the problem can be reduced to simpler cases of the same form as the original problem.
2. The simplest case(s) can be solved directly.

For such problems, it is often straightforward to develop a recursive method. Whenever any method (recursive or not) is called, a new activation record is created to provide a frame of reference for the execution of the method. Each activation record contains

a. the return address, that is, the address of the statement that will be executed when the call has been completed;

b. the value of each argument: a copy of the corresponding argument is made (if the type of the argument is reference-to-object, the reference is copied);

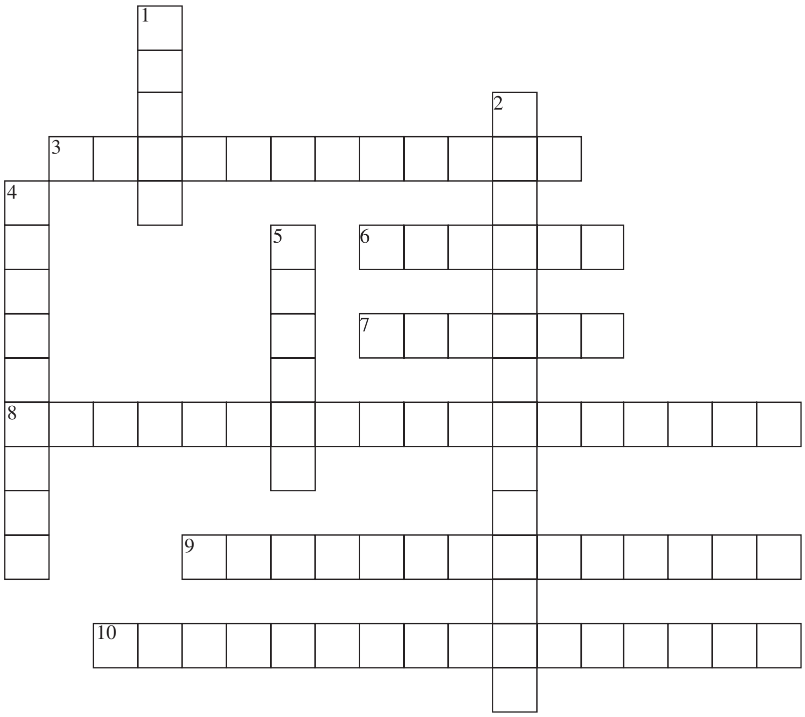
c. the values of the method's other local variables;

Activation records make recursion possible because they hold information that might otherwise be destroyed if the method called itself. When the execution of the current method has been completed, a return is made to the address specified in the current activation record. The previous activation record is then used as the frame of reference for that method's execution.

Any problem that can be solved with recursive methods can also be solved iteratively, that is, with a loop. Typically, iterative methods are slightly more efficient than their recursive counterparts because far fewer activation records are created and maintained. But the elegance and coding simplicity of recursion more than compensates for this slight disadvantage.

A *backtracking* strategy advances step-by-step toward a goal. At each step, a choice is made, but when a dead end is reached, the steps are re-traced in reverse order; that is, the most recent choice is discarded and a new choice is made. Backtracking was deployed for the maze-search application above, and can be used in Programming Projects 5.2 (eight-queens), 5.3 (knight's tour), 5.4 (Sudoku), and 5.5 (Numbrix).

CROSSWORD PUZZLE



www.CrosswordWeaver.com

ACROSS

3. The strategy of trying to reach a goal by a sequence of chosen positions, with a re-tracing in reverse order of positions that cannot lead to the goal.

6. For the `move` method, worstSpace(*n*) is __________ in *n*.

7. A precondition of the `binarySearch` method is that the array is ___________.

8. What is generated when infinite recursion occurs.

9. Boxes that contain information (both variables and code) related to each invocation of the method.

10. The information that is saved every time a method is called.

DOWN

1. The mechanism by which the compiler implements the saving and restoring of activation records.

2. In the `binarySearch` method, the index where the key could be inserted without disordering the array.

4. A method is ________ if it can be called while it is active.

5. A method is ______ if it is being executed or has called an active method.

CONCEPT EXERCISES

5.1 What is wrong with the following underlying method for calculating factorials?

```
/**
 * Calculates the factorial of a non-negative integer, that is, the product of all
 * integers between 1 and the given integer, inclusive.  The worstTime(n) is O(n),
 * where n is the given integer.
 *
 * @param n the non-negative integer whose factorial is calculated.
 *
 * @return the factorial of n
 *
 */
public static long fact (int n)
{
       if (n <= 1)
               return 1;
       return fact (n+1) / (n+1);
} // fact
```

5.2 Show the first three steps in an execution-frames trace of the `move` method after an initial call of

```
      move (4,  'A',  'B', 'C');
```

5.3 Perform an execution-frames trace to determine the output from the following ***incorrect*** version of the `recPermute` method (from Lab 9) after an initial call to

```
      permute ("ABC");
```

invokes

```
recPermute ([‘A’, ‘B’, ‘C’], 0);
```

```
/**
 *  Finds all permutations of a subarray from a given position to the end of the array.
 *
 *  @param c an array of characters
 *  @param k the starting position in c of the subarray to be permuted.
 *
 *  @return a String representation of all the permutations.
 *
 */
public static String recPermute (char[ ] c, int k)
{
       if (k == c.length - 1)
               return String.valueOf (c) + "\n";
       else
       {
           String allPermutations = new String();

           char temp;
```

```
            for (int i = k; i < c.length; i++)
            {
              temp = c [i];
              c [i] = c [k + 1];
              c [k + 1] = temp;
              allPermutations += recPermute (String.valueOf (c).toCharArray(), k+1);
            } // for
            return allPermutations;
        } // else
  } // method recPermute
```

5.4 Perform an execution-frames trace to determine the output from the following ***incorrect*** version of the `recPermute` method (from Lab 9) after an initial call to

```
permute ("ABC");
```

invokes

```
recPermute ([‘A’, ‘B’, ‘C’], 0);

/**
 *  Finds all permutations of a subarray from a given position to the end of the array.
 *
 *  @param c an array of characters
 *  @param k the starting position in c of the subarray to be permuted.
 *
 *  @return a String representation of all the permutations.
 *
 */
 public static String recPermute (char[ ] c, int k)
 {
        if (k == c.length - 1)
                return String.valueOf (c) + "\n";
        else
        {
            String allPermutations = new String();

            char temp;

            for (int i = k; i < c.length; i++)
            {
              allPermutations += recPermute (String.valueOf (c).toCharArray(), k+1);
              temp = c [i];
              c [i] = c [k];
              c [k] = temp;
            } // for
            return allPermutations;
        } // else
 } // method recPermute
```

5.5 Use the Principle of Mathematical Induction (Appendix 1) to prove that the `move` method in the Towers of Hanoi example is correct, that is, for any integer `n > = 1`, `move (n, orig, dest, temp)` returns the steps to move `n` disks from pole `orig` to pole `dest`.
Hint: for $n = 1, 2, 3, \ldots,$ let S_n be the statement:

> `move (n, orig, dest, temp)` returns the steps to move `n` disks from any pole `orig` to any other pole `dest`.

a. base case. Show that S_1 is true.

b. inductive case. Let n be any integer greater than 1 and assume S_{n-1} is true. Then show that S_n is true. According the code of the `move` method, what happens when `move (n, orig, dest, temp)` is called and `n` is greater than 1?

5.6 In an execution trace of the `move` method in the Towers of Hanoi application, the number of steps is equal to the number of recursive calls to the `move` method plus the number of direct moves. Because each call to the `move` method includes a direct move, the number of recursive calls to the `move` method is always one less than the number of direct moves. For example, in the execution trace shown in the chapter, $n = 3$. The total number of calls to move is $2^n - 1 = 7$. Then the number of recursive calls to `move` is 6, and the number of direct moves is 7, for a total of 13 steps (recall that we started at Step 0, so the last step is Step 12). How many steps would there be for an execution trace with $n = 4$?

5.7 Show that, for the recursive `binarySearch` method, averageTime(n) is logarithmic in n for a successful search.
Hint: Let n represent the size of the array to be searched. Because the average number of calls is a non-decreasing function of n, it is enough to show that the claim is true for values of n that are one less than a power of 2. So assume that

$$n = 2^k - 1, \text{for some positive integer } k.$$

In a successful search,

one call is sufficient if the item sought is half-way through the region to be searched;

two calls are needed if the item sought is one-fourth or three-fourths of the way through that region;

three calls are needed if the item sought is one-eighth, three-eighths, five-eighths or seven-eighths of the way through the region;

and so on.

The total number of calls for all successful searches is

$$(1 * 1) + (2 * 2) + (3 * 4) + (4 * 8) + (5 * 16) + \cdots + (k * 2^{k-1})$$

The average number of calls, and hence an estimate of averageTime(n), is this sum divided by n. Now use the result from Exercise 2.6 of Appendix 2 and the fact that

$$k = \log_2 (n + 1)$$

5.8 If a call to the `binarySearch` method is successful, will the index returned always be the smallest index of an item equal to the key sought? Explain.

PROGRAMMING EXERCISES

5.1 Develop an iterative version of the `getBinary` method in Section 5.3. Test that method with the same `BinaryTest` class (available on the book's website) used to test the recursive version.

5.2 Develop an iterative version of the permute method (from Lab 9). Here is the method specification:

```
/**
 * Finds all permutations of a specified String.
 *
 * @param s - the String to be permuted.
 *
 * @return a String representation of all the permutations, with a line separator
 *         (that is, "\n") after each permutation.
 */
 public static String permute (String  s)
```

For example, if the original string is “BADCGEFH”, the value returned would be

```
ABCDEFGH
ABCDEFHG
ABCDEGFH
ABCDEGHF
ABCDEHFG
```

and so on. Test your method with the same PermuteTest method developed in Lab 9 to test the recursive version.

Hint: One strategy starts by converting s to a character array c. Then the elements in c can be easily swapped with the help of the index operator, [ ]. To get the first permutation, use the static method sort in the Arrays class of java.util. To give you an idea of how the next permutation can be constructed from the current permutation, suppose, after some permutations have been printed,

c = ['A', 'H', 'E', 'G', 'F', 'D', 'C', 'B']

What is the smallest index whose character will be swapped to obtain the next permutation? It is index 2, because the characters at indexes 3 through 7 are already in reverse alphabetical order: ‘G’ > ‘F’ > ‘D’ > ‘C’ > ‘B’. We swap ‘E’ with ‘F’, the smallest character greater than ‘E’ at an index greater than 2. After swapping, we have

c = ['A', 'H', 'F', 'G', 'E', 'D', 'C', 'B']

We then reverse the characters at indexes 3 through 7 to get those characters into increasing order:

c = ['A', 'H', 'F', 'B', 'C', 'D', 'E', 'G'],

the next higher permutation after ‘A’, ‘H’, ‘E’, ‘G’, ‘F’, ‘D’, ‘C’, ‘B’.

Here is an outline:

```
public static String permute (String s)
{
   int n = s.length();

   boolean finished = false;

   char[ ] c = s.toCharArray();

   String perms = "";

   Arrays.sort (c);      // c is now in ascending order
```

```
    while (!finished)
    {

        perms += String.valueOf (c));

        // In 0 ... n-1, find the highest index p such that
        // p = 0 or c [p - 1] < c [p].
                ...

        if (p == 0)
                finished = true;
        else
        {
            // In p ... n-1, find the largest index i such that c [i] > c [p - 1].
                     ...

            // Swap c [i] with c [p - 1].


            // Swap c [p] with c [n-1], swap c [p+1] with c[n-2],
            // swap c [p+2] with c [n-3], ...
                  ...

        } // else
    } // while
    return perms;

} // method permute
```

In the above example, `p - 1 = 2` and `i = 4`, so `c [p - 1]`, namely, ‘E’ is swapped with `c [i]`, namely, ‘F’.

Explain how strings with duplicate characters are treated differently in this method than in the recursive version.

5.3 Given two positive integers i and j, the greatest common divisor of i and j, written

```
gcd (i, j)
```

is the largest integer k such that

$$(i \,\%\, k = 0) \quad \text{and} \quad (j \,\%\, k = 0).$$

For example, gcd (35, 21) = 7 and gcd (8, 15) = 1. Test and develop a wrapper method and a wrapped recursive method that return the greatest common divisor of i and j. Here is the method specification for the wrapper method:

```
/**
 *  Finds the greatest common divisor of two given positive integers
 *
 *  @param i - one of the given positive integers.
 *  @param j - the other given positive integer.
 *
 *  @return the greatest common divisor of iand j.
 *
 *  @throws IllegalArgumentException – if either i or j is not a positive integer.
 *
 */
public static int gcd (int i, int j)
```

Big hint: According to Euclid's algorithm, the greatest common divisor of i and j is j if $i \ \% \ j = 0$. Otherwise, the greatest common divisor of i and j is the greatest common divisor of j and *(i % j)*.

5.4 A ***palindrome*** is a string that is the same from right-to-left as from left-to-right. For example, the following are palindromes:

ABADABA

RADAR

OTTO

MADAMIMADAM

EVE

For this exercise, we restrict each string to upper-case letters only. (You are asked to remove this restriction in the next exercise.)

Test and develop a method that uses recursion to check for palindromes. The only parameter is a string that is to be checked for palindromity. The method specification is

```
/**
 * Determines whether a given string of upper-case letters is a palindrome.
 * A palindrome is a string that is the same from right-to-left as from left-to-right.
 *
 * @param s – (a reference to) the given string
 *
 * @return true – if the string s is a palindrome; otherwise, false.
 *
 * @throws NullPointerException – if s is null.
 * @throws IllegalArgumentException – if s is the empty string.
 *
 */
public static boolean isPalindrome (String s)
```

5.5 Expand the recursive method (and test class) developed in Programming Exercise 5.4 so that, in testing to see whether `s` is a palindrome, non-letters are ignored and no distinction is made between upper-case and lower-case letters. Throw `IllegalArgumentException` if `s` has no letters. For example, the following are palindromes:

Madam, I'm Adam.

Able was I 'ere I saw Elba.

A man. A plan. A canal. Panama!

Hint: The `toUpperCase()` method in the `String` class returns the upper-case `String` corresponding to the calling object.

5.6 **a.** Test and develop a wrapper method `power` and an underlying recursive method that return the result of integer exponentiation. The method specification of the wrapper method is

```
/**
 *  Calculates the value of a given integer raised to the power of a second integer.
 *  The worstTime(n) is O(n), where n is the second integer.
 *
 *  @param i – the base integer (to be raised to a power).
 *  @param n – the exponent (the power i is to be raised to).
 *
 *  @return the value of i to the nth power.
 *
 *  @throws IllegalArgumentException – if n is a negative integer or if i raised to
 *          to the n is greater than Long.MAX_VALUE.
 *
 */
 public static long power (long i, int n)
```

Hint: We define $0^0 = 1$, so for any integer $i, i^0 = 1$. For any integers $i > 0$ and $n > 0$,

$$i^n = \mathrm{i}^* i^{n-1}$$

b. Develop an iterative version of the `power` method.

c. Develop an underlying recursive version called by the `power` method for which worstTime(n) is logarithmic in n.
Hint: If `n` is even, `power (i, n)` = `power (i * i, n/2)`; if `n` is odd, `power (i, n) = i *` $\mathtt{i}^{\mathtt{n - 1}}$ = `i * power (i * i, n/2)`.
For testing parts b and c, use the same test suite you developed for part a.

5.7 Test and develop a recursive method to determine the number of distinct ways in which a given amount of money in cents can be changed into quarters, dimes, nickels, and pennies. For example, if the amount is 17 cents, then there are six ways to make change:

1 dime, 1 nickel and 2 pennies;

1 dime and 7 pennies;

3 nickels and 2 pennies;

2 nickels and 7 pennies;

1 nickel and 12 pennies;

17 pennies.

Here are some amount/ways pairs. The first number in each pair is the amount, and the second number is the number of ways in which that amount can be changed into quarters, dimes, nickels and pennies:

17	6
5	2
10	4
25	13
42	31
61	73
99	213

Here is the method specification:

```
/**
 *  Calculates the number of ways that a given amount can be changed
 *  into coins whose values are no larger than a given denomination.
 *
 *  @param amount - the given amount.
 *  @param denomination - the given denomination (1 = penny,
 *         2 = nickel, 3 = dime, 4 = quarter).
 *
 *  @return 0 - if amount is less than 0; otherwise, the number of ways
 *          that amount can be changed into coins whose values are no
 *          larger than denomination.
 *
 */
public static int ways (int amount, int denomination)
```

For the sake of simplifying the `ways` method, either develop an enumerated type `Coin` or develop a `coins` method that returns the value of each denomination. Thus, `coins (1)` returns 1, `coins (2)` returns 5, `coins (3)` returns 10, and `coins (4)` returns 25.
Hint: The number of ways that one can make change for an *amount* using coins no larger than a quarter is equal to the number of ways that one can make change for *amount*—25 using coins no larger than a quarter plus the number of ways one can make change for *amount* using coins no larger than a dime.

5.8 Modify the maze-search application to allow an end user to enter the maze information directly, instead of in a file. Throw exceptions for incorrect row or column numbers in the start and finish positions.

5.9 Modify the maze-search application so that diagonal moves would be valid.
Hint: only the `MazeIterator` class needs to be modified.

Programming Project 5.1

Iterative Version of the Towers of Hanoi

Develop an iterative version of the `moveDisks` method in the Towers of Hanoi game. Test your version with the same test suite, on the book's website, developed for the recursive version.
Hint: We can determine the proper move at each stage provided we can answer the following three questions:

1. Which disk is to be moved?
   To answer this question, we set up an *n-bit counter*, where *n* is the number of disks, and initialize that counter to all zeros. The counter can be implemented as an n-element array of zeros and ones, or as an n-element array of **`boolean`** values. That is the only array you should use for this project.
   For example, if $n = 5$, we would start with

   00000

   Each bit position corresponds to a disk: the rightmost bit corresponds to disk 1, the next rightmost bit to disk 2, and so on.

(continued on next page)

(continued from previous page)

At each stage, the rightmost zero bit corresponds to the disk to be moved, so the first disk to be moved is, as you would expect, disk 1.

After a disk has been moved, we increment the counter as follows: starting at the rightmost bit and working to the left, keep flipping bits (0 to 1, 1 to 0) until a zero gets flipped. For example, the first few increments and moves are as follows:

```
00000 // move disk 1
00001 // move disk 2
00010 // move disk 1
00011 // move disk 3
00100 // move disk 1
00101 // move disk 2
```

After 31 moves, the counter will contain all ones, so no further moves will be needed or possible. In general, $2^n - 1$ moves and $2^n - 1$ increments will be made.

2. In which direction should that disk be moved?
If n is odd, then odd-numbered disks move clockwise:

$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] & C \arrow[ll, bend left]
\end{tikzcd}$$

and even-numbered disks move counter clockwise:

$$\begin{tikzcd}
A \arrow[r] \arrow[rr, bend right] & B \arrow[r] & C
\end{tikzcd}$$

If n is even, even-numbered disks move clockwise and odd-numbered disks move counter clockwise.

If we number the poles 0, 1, and 2 instead of ‘A’, ‘B’, and ‘C’, then movements can be accomplished simply with modular arithmetic. Namely, if we are currently at pole k, then

```
k = (k + 1) % 3;
```

achieves a clockwise move, and

```
k = (k + 2) % 3;
```

achieves a counter-clockwise move. For the pole on which the just moved disk resides, we cast back to a character:

```
char(k + 'A')
```

3. Where is that disk now?
Keep track of where disk 1 is. If the counter indicates that disk 1 is to be moved, use the answer to question 2 to move that disk. If the counter indicates that the disk to be moved is not disk 1, then the answer to question 2 tells you where that disk is now. Why? Because that disk cannot be moved on top of disk 1 and cannot be moved from the pole where disk 1 is now.

Programming Project 5.2

Eight Queens

(This problem can be straightforwardly solved by using the `BackTrack` class and implementing the `Application` interface.) Test and develop an `EightQueens` class to place eight queens on a chess board in such a way that no queen is under attack from any other queen. Also, test and develop an `EightQueensUser` classsimilar to the `MazeUser` class in Section 5.6.1.

Analysis A chess board has eight rows and eight columns. In the game of chess, the queen is the most powerful piece: she can attack any piece in her row, any piece in her column, and any piece in either of her diagonals. See Figure 5.14.

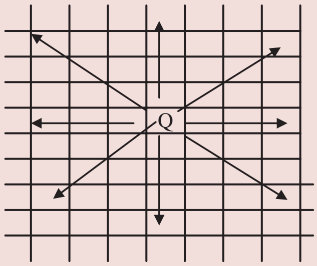


FIGURE 5.14 Positions vulnerable to a queen in chess. The arrows indicate the positions that can be attacked by the queen 'Q' in the center of the figure

The output should show the chess board after the placement of the eight queens. For example:

	0	1	2	3	4	5	6	7
0	Q							
1							Q	
2					Q			
3								Q
4		Q						
5				Q				
6						Q		
7			Q					

Hint: There must be exactly one queen in each row and exactly one queen in each column. There is no input: start with a queen at (0, 0), and place a queen in each column. A valid position is one that is not in the same row, column or diagonal as any queen placed in a previous column. The `QueensIterator` constructor should advance to row 0 of the next column. The `next` method should advance to the next row in the same column. So

(continued on next page)

(continued from previous page)

the first time the tryToReachGoal method in the BackTrack class (which cannot be modified by you) is called, the choices are:

(0, 1) // invalid: in the same row as the queen at (0, 0)
(1, 1) // invalid: in the same diagonal as the queen at (0, 0)
(2, 1) // valid

When the tryToReachGoal method is called again, the choices are:

(0, 2) // invalid: in the same row as the queen at (0, 0)
(1, 2) // invalid: in the same diagonal as the queen at (1, 2)
(2, 2) // invalid: in the same row as the queen at (1, 2)
(3, 2) // invalid: in the same diagonal as the queen at (1, 2)
(4, 2) // valid

Programming Project 5.3

A Knight's Tour

(This problem can be straightforwardly solved by using the BackTrack class and implementing the Application interface.) Test and develop a KnightsTour class to show the moves of a knight in traversing a chess board. Also, test and develop a KnightsTourUser class—similar to the MazeUser class in Section 5.6.1. The Backtrack class and Application interface are not to be modified by you.

Analysis A chess board has eight rows and eight columns. From its current position, a knight's next position will be either two rows and one column or one row and two columns from the current position. For example, Figure 5.15 shows the legal moves of a knight at position (5, 3), that is, row 5 and column 3.

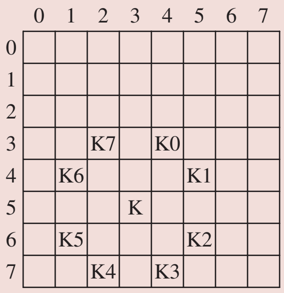


FIGURE 5.15 For a knight (K) at coordinates (5, 3), the legal moves are to the grid entries labeled K0 through K7

For simplicity, the knight starts at position (0, 0). Assume the moves are tried in the order given in Figure 5.15. That is, from position (row, column), the order tried is:

(row − 2, column + 1)

(row − 1, column + 2)

(row + 1, column + 2)

(row + 2, column + 1)

(row + 2, column − 1)

(row + 1, column − 2)

(row − 1, column − 2)

(row − 2, column − 1)

Figure 5.16 shows the first few moves.

	0	1	2	3	4	5	6	7
0	1				3			
1			2				4	
2								10
3								5
4							9	
5							6	
6						8		
7								7

FIGURE 5.16 The first few valid moves by a knight that starts at position (0, 0) and iterates according to the order shown in Figure 5.15. The integer at each filled entry indicates the order in which the moves were made

For the nine moves, starting at (0, 0), in Figure 5.16, no backtracking occurs. In fact, the first 36 moves are never backtracked over. But the total number of backtracks is substantial: over 3 million. The solution obtained by the above order of iteration is:

	0	1	2	3	4	5	6	7
0	1	38	55	34	3	36	19	22
1	54	47	2	37	20	23	4	17
2	39	56	33	46	35	18	21	10
3	48	53	40	57	24	11	16	5
4	59	32	45	52	41	26	9	12
5	44	49	58	25	62	15	6	27
6	31	60	51	42	29	8	13	64
7	50	43	30	61	14	63	28	7

Notice that the 37th move, from position (1, 3), does not take the first available choice—to position (3, 2)—nor the second available choice—to position (2, 1). Both of those choices led to dead ends, and backtracking occurred. The third available choice, to (0, 1), eventually led to a solution.

(continued on next page)

(continued from previous page)

System Test 1 (the input is in boldface)

Enter the starting row and column: **0 0**

Starting at row 0 and column 0, the solution is

	0	1	2	3	4	5	6	7
0	1	38	55	34	3	36	19	22
1	54	47	2	37	20	23	4	17
2	39	56	33	46	35	18	21	10
3	48	53	40	57	24	11	16	5
4	59	32	45	52	41	26	9	12
5	44	49	58	25	62	15	6	27
6	31	60	51	42	29	8	13	64
7	50	43	30	61	14	63	28	7

Note: The lines are not part of the output; they are included for readability.

System Test 2

Enter the starting row and column: **3 5**

Starting at row 3 and column 5, the solution is

	0	1	2	3	4	5	6	7
0	33	42	35	38	31	40	19	10
1	36	57	32	41	20	9	2	17
2	43	34	37	30	39	18	11	8
3	56	51	58	21	28	1	16	3
4	59	44	29	52	47	22	7	12
5	50	55	46	27	62	15	4	23
6	45	60	53	48	25	6	13	64
7	54	49	26	61	14	63	24	5

This solution requires 11 million backtracks. Some starting positions, for example (0, 1), require over 600 million backtracks. But for every possible starting position, there is a solution.

Programming Project 5.4

Sudoku

(This problem can be solved by using the `BackTrack` class and implementing the `Application` interface.) Sudoku (from the Japanese "single number") is a puzzle game in which the board is a 9-by-9 grid, further subdivided into nine 3-by-3 minigrids. Initially, each cell in the grid has either a single digit or a blank. For example, here (from http://en.wikipedia.org/wiki/Sudoku) is a sample initial configuration:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

The rules of the game are simple: Replace each blank cell with a single digit so that each row, each column, and each minigrid contain the digits 1 through 9. For example, in the above grid, what digit must be stored in the cell at row 6 and column 5? (The row numbers and column numbers start at 0, so the cell at (6, 5) is in the upper right-hand corner of the bottom center minigrid.) The value cannot be

1, because there is already a 1 in that minigrid

2, because there is already a 2 in row 6

3, because there is already a 3 in column 5

4, because there is already a 4 in that minigrid

5, because there is already a 5 in column 5

6, because there is already a 6 in row 6

8, because there is already an 8 in that minigrid (and in row 6)

9, because there is already a 9 in that minigrid

By a process of elimination, we conclude that the digit 7 should be placed in the cell at (6, 5). Using logic only, you can determine the complete solution to the puzzle. If you click on the link above, you will see the solution.

(continued on next page)

(continued from previous page)

Instead of solving Sudoku puzzles by logic, you can solve them with backtracking. You would not want to do this by hand, because for some Sudoku puzzles, over 100,000 backtracks would be needed. But you can solve any Sudoku puzzle with the help of the `BackTrack` class (which you are not allowed to modify). You will need to supply a `Sudoku` class that implements the `Application` interface, and a `SudokuUser` class (similar to the `MazeUser` class in Section 5.6.1). You may want to modify the `Position` class from Section 5.6.1 to include a `digit` field. Then the iteration will be over the digit values that a position can take on, and the `SudokuIterator` constructor will advance to the next position in the grid whose digit value is 0.

The initial configuration will be supplied from a file in which each line has a row, column, and digit-value. For example,

```
0  1  6        4  6  3
0  3  1        5  0  7
0  5  4        5  3  9
0  7  5        5  5  1
1  2  8        5  8  4
1  3  3        6  0  5
1  5  5        6  8  2
1  6  6        7  2  7
2  0  2        7  3  2
2  8  1        7  5  6
3  0  8        7  6  9
3  3  4        8  1  4
3  5  7        8  3  5
3  8  6        8  5  8
4  2  6        8  7  7
```

For the `SudokuTest` class, include a test to make sure the `iterator` method starts at the appropriate row and column, and the `next` method advances the position's digit. For one test of the `isOK` method, the initial configuration should have 2 in (0, 0) and 1 in (0, 1); after several calls to the `next()` method, the `isOK` method should return true. To test the `isGoal` method, the initial configuration should be a complete solution except for a blank in (8, 8); `isGoal` should eventually return true.

Also include tests for the following:

`InputMismatchException`, if the row or column is not an int, or if the value is not a byte.

`ArrayIndexOutOfBoundsException`, if the row or column is not in the range 0 . . . 8, inclusive.

`IllegalArgumentException`, if the value is not a single digit or duplicates the value in the same row, same column, or same minigrid.

`IllegalStateException`, if there is no solution to the puzzle.

Include tests in which the solution is found.

Programming Project 5.5

Numbrix

(This problem can be straightforwardly solved by using the `BackTrack` class and implementing the `Application` interface.) Numbrix, invented by Marilyn vos Savant, is a number-positioning puzzle. You start with a square grid—we will use a 9 × 9 grid for example. Initially, some of the grid has been filled in with integers in the range from 1 through 81. The goal is to complete the grid so that each number (except 81) is connected to the next higher number either horizontally or vertically.

Here is an initial state:

```
31  0   33  0   39  0   45  0   47
0   29  0   35  0   41  0   49  0
25  0   0   0   0   0   0   0   51
0   23  0   0   0   0   0   53  0
9   0   0   0   0   0   0   0   79
0   1   0   0   0   0   0   77  0
7   0   0   0   0   0   0   0   81
0   3   0   17  0   67  0   73  0
5   0   15  0   65  0   69  0   71
```

Here is the final state:

```
31  32  33  34  39  40  45  46  47
30  29  28  35  38  41  44  49  48
25  26  27  36  37  42  43  50  51
24  23  22  21  58  57  54  53  52
9   10  11  20  59  56  55  78  79
8   1   12  19  60  61  76  77  80
7   2   13  18  63  62  75  74  81
6   3   14  17  64  67  68  73  72
5   4   15  16  65  66  69  70  71
```

Design, test and write and program to solve any Numbrix puzzle. The first line of the input file will contain the grid's side length, and each subsequent line will contain the row, column and value of some initial entry in the grid.

System Test 1 (input in boldface)

Please enter the path for the file that contains, on the first line, the grid length, and then, on each other line, the row, column, and value of each non-zero entry in the initial state: **numbrix.in1**

(continued on next page)

(continued from previous page)

```
The initial state is as follows:

31  0   33  0   39  0   45  0   47
0   29  0   35  0   41  0   49  0
25  0   0   0   0   0   0   0   51
0   23  0   0   0   0   0   53  0
9   0   0   0   0   0   0   0   79
0   1   0   0   0   0   0   77  0
7   0   0   0   0   0   0   0   81
0   3   0   17  0   67  0   73  0
5   0   15  0   65  0   69  0   71

A solution has been found. The final state is as follows:

31  32  33  34  39  40  45  46  47
30  29  28  35  38  41  44  49  48
25  26  27  36  37  42  43  50  51
24  23  22  21  58  57  54  53  52
9   10  11  20  59  56  55  78  79
8   1   12  19  60  61  76  77  80
7   2   13  18  63  62  75  74  81
6   3   14  17  64  67  68  73  72
5   4   15  16  65  66  69  70  71
```

Note: The file numbrix.in1 consists of the following:

```
9
0 0 31
0 2 33
0 4 39
0 6 45
0 8 47
1 1 29
1 3 35
1 5 41
1 7 49
2 0 25
2 8 51
3 1 23
3 7 53
4 0 9
4 8 79
5 1 1
5 7 77
6 0 7
6 8 81
7 1 3
7 3 17
7 5 67
7 7 73
8 0 5
8 2 15
8 4 65
8 6 69
8 8 71
```

System Test 2 (input in boldface)

Please enter the path for the file that contains, on the first line, the grid length, and then, on each other line, the row, column, and value of each non-zero entry in the initial state: **numbrix.in2**

The initial state is as follows:

```
9   0   7   0   1   0   73  0   77
0   11  0   5   0   71  0   75  0
15  0   0   0   0   0   0   0   79
0   17  0   0   0   0   0   65  0
25  0   0   0   0   0   0   0   63
0   27  0   0   0   0   0   61  0
37  0   0   0   0   0   0   0   59
0   41  0   33  0   53  0   51  0
39  0   43  0   45  0   47  0   49
```

A solution has been found:

The final state is as follows:

```
9   8   7   6   1   72  73  76  77
10  11  12  5   2   71  74  75  78
15  14  13  4   3   70  81  80  79
16  17  18  19  20  69  66  65  64
25  24  23  22  21  68  67  62  63
26  27  28  29  30  55  56  61  60
37  36  35  34  31  54  57  58  59
38  41  42  33  32  53  52  51  50
39  40  43  44  45  46  47  48  49
```

Note: The file numbrix.in2 consists of the following:

```
9                4 8 63
0 0 9            5 1 27
0 2 7            5 7 61
0 4 1            6 0 37
0 6 73           6 8 59
0 8 77           7 1 41
1 1 11           7 3 33
1 3 5            7 5 53
1 5 71           7 7 51
1 7 75           8 0 39
2 0 15           8 2 43
2 8 79           8 4 45
3 1 17           8 6 47
3 7 65           8 8 49
4 0 25
```

(continued on next page)

(continued from previous page)

Also include tests for the following:

`InputMismatchException`, if the row or column is not an **`int`**, or if the value is not an **`int`**.

`ArrayIndexOutOfBoundsException`, if the row or column is not in the range 0 . . . grid length, inclusive.

`IllegalArgumentException`, if the value is greater than grid length squared or duplicates a value already in the grid.

`IllegalStateException`, if there is no solution to the puzzle.

Hint: The implementation is simplified if you assume that one of the original values in the grid is 1, as in System Tests 1 and 2. After you have tested your implementation with that assumption, remove the assumption. Here are two system tests in which 1 is not one of the original values in the grid:

System Test 3 (input in boldface)

Please enter the path for the file that contains, on the first line, the grid length, and then, on each other line, the row, column, and value of each non-zero entry in the initial state: **numbrix.in3**

```
      The initial state is as follows:

75  76  81  66  65  14  13  8   7
74  0   0   0   0   0   0   0   6
73  0   0   0   0   0   0   0   5
72  0   0   0   0   0   0   0   4
55  0   0   0   0   0   0   0   23
54  0   0   0   0   0   0   0   24
45  0   0   0   0   0   0   0   25
44  0   0   0   0   0   0   0   26
43  42  41  40  39  34  33  28  27

A solution has been found:
      The final state is as follows:

75  76  81  66  65  14  13  8   7
74  77  80  67  64  15  12  9   6
73  78  79  68  63  16  11  10  5
72  71  70  69  62  17  2   3   4
55  56  57  58  61  18  1   22  23
54  53  52  59  60  19  20  21  24
45  46  51  50  37  36  31  30  25
44  47  48  49  38  35  32  29  26
43  42  41  40  39  34  33  28  27
```

Note: The file numbrix.in3 consists of the following:

```
9           4 8 23
0 0 75      5 0 54
0 1 76      5 8 24
0 2 81      6 0 45
0 3 66      6 8 25
0 4 65      7 0 44
0 5 14      7 8 26
0 6 13      8 0 43
0 7 8       8 1 42
0 8 7       8 2 41
1 0 74      8 3 40
1 8 6       8 4 39
2 0 73      8 5 34
2 8 5       8 6 33
3 0 72      8 7 28
3 8 4       8 8 27
4 0 55
```

System Test 4 (input in boldface)

Please enter the path for the file that contains, on the first line, the grid length, and then, on each other line, the row, column, and value of each non-zero entry in the initial state: **numbrix.in4**

The initial state is as follows:

```
25  0  0  0   0
0   3  0  0   0
0   0  0  16  0
0   5  0  0   0
0   0  9  0   0
```

A solution has been found:

The final state is as follows:

```
25  24  23  22  21
2   3   18  19  20
1   4   17  16  15
6   5   10  11  14
7   8   9   12  13
```

Note: The file numbrix.in4 consists of the following:

```
5
0 0 25
1 1 3
2 3 16
3 1 5
4 2 9
```

This page intentionally left blank

Array-Based Lists

CHAPTER 6

We begin this chapter by introducing the Java Collection Framework's `List` interface, which extends the `Collection` interface by providing some index-related methods. For example, there is a `get` method that returns the element at a given index. In any `List` object, that is, in any instance of a class that implements the `List` interface, the elements are stored in sequence, according to an index. For example, a `List` object `pets` might have the elements arranged as follows: "dog", "cat", "iguana", "gerbil". Here "dog" is at index 0, and "gerbil" is at index 3.

The main focus of this chapter is the user's view of the `ArrayList` class. We start by investigating the method specifications. We then briefly turn to the developer's view: The Java Collection Framework's `ArrayList` class implements the `List` interface with an underlying array that allows constant-time access of any element from its index. We finish up the chapter with an application in the area of public-key cryptography.

As with all of the other collection classes in the Java Collections Framework, the `ArrayList` class is parameterized, and the element class is the type parameter, so it would be more appropriate to refer to the class as `ArrayList<E>`. When a user creates an instance of the `ArrayList<E>` class, the user specifies the element type that will replace the type parameter `E`. For example, to create an empty `ArrayList` object whose elements must be of type (reference to) `String`, we write

```
ArrayList<String> myList = new ArrayList<String>();
```

As we saw in Chapter 4, the only stipulation on the element type is that it cannot be a primitive type, such as **`int`** (but the wrapper class `Integer` is acceptable).

Chapter 7 covers another `List` implementation, the `LinkedList` class The `ArrayList` and `LinkedList` classes have their own advantages and disadvantages: there is no "best" `List` implementation. A major goal of the two chapters is to help you recognize situations in which one of the classes would be preferable to the other.

CHAPTER OBJECTIVES

1. Recognize the methods in the `List` interface that are not in the `Collection` interface.
2. Understand the user's view of the `ArrayList class`.
3. Be able to decide when an `ArrayList` is preferable to an array—and vice versa.
4. Understand the `VeryLongInt` class from both the user's view and the developers' view.

6.1 The List Interface

The List interface extends the Collection interface with methods that have an index as either a parameter or a return type. Here are thumbnail sketches of five of the methods. For each method below, E (for “element”) is the type parameter.

```
// Returns the element at position index in this List object.  The worstTime(n) is O(n).
E get (int index);

// Replaces the element that was at position index in this List object with the
// parameter element, and returns the previous occupant. The worstTime(n) is O(n).
E set (int index, E element);

// Returns the index of the first occurrence of obj in this List object, if obj
//  appears in this List object..  Otherwise, returns -1.  The worstTime(n) is O(n).
int indexOf (Object obj);

// Inserts element at position index in this List object; every element that
// was at a position >= index before this call is now at the next higher position.
// The worstTime(n) is O(n).
void add (int index, E element);

// Removes and returns the element at position index in this List object; every
//  element that was at a position > index before this call is now at the next lower
//  position.  The worstTime(n) is O(n).
E remove (int index);
```

Any implementation of this interface may improve on the time-estimate upper bounds for the methods; and, in fact, for the ArrayList class (see following), worstTime(n) is O(1) for both the get and set methods. We cannot give examples of calls to the List methods because interfaces cannot be instantiated, but the above five methods should give you the idea that many of the methods in a List object are index based. Of course, we also have some holdovers from the Collection interface: the methods size, isEmpty, contains, clear, and so on. And the add (E element) method specifies that the element is inserted at the *end* of the list.

Section 6.2 introduces the ArrayList class, which implements the List interface. We will emphasize the user’s perspective of the ArrayList class by studying the method specifications. In Section 6.3, we take a quick look at the developer’s perspective: the actual fields and method definitions in the Java Collections Framework. Then we return to the user’s view with an application of the ArrayList class.

6.2 The ArrayList Class

As we will see shortly, an ArrayList object can be thought of as an improved version of the one-dimensional array. Like an array, the ArrayList class supports random access of its elements, that is, any element can be accessed in constant time, given only the index of the element. But unlike an array, an ArrayList object’s size (as well as its capacity) is automatically maintained during the execution of a program. Also, there are ArrayList methods for inserting and deleting at any index—if you insert or delete in an array, you must write the code to open up or close up the space. Finally, if you want to insert an element into an array that is already full, you must write the code (to create a new array, copy the old array to the new array, and so on). With an ArrayList object, such expansions are handled automatically.

Figure 6.1 has the big picture from the user's perspective: the method heading for each public method in the ArrayList class. Except for the constructors, the headings are in alphabetical order by method identifier. The type parameter E may appear as the return type as well as the element type of a parameter.

Section 6.2.1 has more detail: the method specifications, with examples, for several ArrayList methods.

```
public ArrayList (int initialCapacity)
public ArrayList()
public ArrayList (Collection<? extends E> c) // See Section 6.2.1
public boolean add (E element) // inserts at back
public void add (int index, E element)
public boolean addAll (Collection<? extends E> c)
public boolean addAll (int index, Collection<? extends E> c)
public void clear() // worstTime (n) is O(n)
public Object clone()
public boolean contains (Object obj)
public boolean containsAll (Collection<?> c)
public void ensureCapacity (int minCapacity)
public boolean equals (Object obj)
public E get (int index) // worstTime (n) is constant
public int hashCode()
public int indexOf (Object obj)
public boolean isEmpty()
public Iterator<E> iterator()
public int lastIndexOf (Object element)
public ListIterator<E> listIterator( )
public ListIterator<E> listIterator (final int index)
public boolean remove (Object obj)
public E remove (int index)
public boolean removeAll (Collection<?> c)
public boolean retainAll (Collection<?> c)
public E set (int index, E element)
public int size( )
public List<E> subList (int fromIndex, int toIndex)
public Object[ ] toArray( )
public T[ ] toArray (T[ ] a)// ClassCastException unless T extends E
public String toString()
public void trimToSize( )
```

FIGURE 6.1 Public methods in the class ArrayList<E>, where E is the type parameter. Except for the constructors, the method headings are in alphabetical order by method identifier

6.2.1 Method Specifications for the ArrayList Class

The method specifications following use javadoc, and will yield specifications that are similar to, but shorter than, those provided with Sun's Application Programming Interface (API). You are strongly urged to consult that API to get the full details of each specification. The phrase "this ArrayList object" refers to the calling object.

Each method's time requirements are specified with Big-O notation because we are merely establishing an upper bound: a specific implementation of the method may reduce that upper bound. If no time estimate for a method is given, you may assume that worstTime(n) is constant. If a method's average-time estimate is the same as the worst-time estimate, only the worst-time estimate is given.

The following method specifications give you a user's view of the `ArrayList` class. For each method, we include an example and a comparison with an array.

1. **Constructor with initial-capacity parameter**

```
/**
 *  Initializes this ArrayList object to be empty, with the specified initial capacity.
 *
 *  @param initialCapacity the initial capacity of the list.
 *
 *  @throws IllegalArgumentException – if the specified initial capacity is negative
 *
 *
 */
public ArrayList (int initialCapacity)
```

Example The following creates an empty `ArrayList` object called `fruits`, with `String` elements and an initial capacity of 100:

```
ArrayList<String> fruits = new ArrayList<String> (100);
```

Note: There is also a default constructor. For example,

```
ArrayList<String> fruits = new ArrayList<String>();
```

simply constructs an empty `ArrayList` object with a default initial capacity (namely, 10).

Comparison to an array: An array object can be constructed with a specified initial capacity. For example,

```
String [ ] vegetables = new String [10];
```

makes `vegetables` an array object with **`null`** references at indexes 0 through 9. Unlike an `ArrayList` object, an array object can consist of primitive elements. For example,

```
double [ ] salaries = new double [200];
```

constructs an array object whose elements will be of type **`double`** and whose initial capacity is 200.

2. **Copy constructor**

```
/**
 *  Constructs a list containing the elements of the specified collection, in the order
 *  they are stored in the specified collection. This ArrayList object has an
 *  initial capacity of 110% the size of the specified collection.   The worstTime(n)
 *  is O(n), where n is the number of elements in the specified collection.
 *
 *  @param c - the specified collection whose elements this ArrayList object is
 *             initialized from.
 *
 */
public ArrayList (Collection<? extends E> c)
```

Example Suppose that myList is an ArrayList object whose elements are the Strings "yes", "no", and "maybe". We can create another ArrayList object that initially contains a copy of myList as follows:

```
ArrayList<String> newList = new ArrayList<String> (myList);
```

Note 1: This constructor is called the ***copy constructor***.

Note 2: The argument corresponding to the parameter c must be an instance of a class (not necessarily the ArrayList class) that implements the Collection interface. And the element type must be the same as the element type of the calling object or a subclass of that type.

For example, if intList is an ArrayList object whose elements are of type Integer (a subclass of Object), we can create an ArrayList object of type Object as follows:

```
ArrayList<Object> objList = new ArrayList<Object> (intList);
```

At this point, all of the elements in objList are of type Integer, but we can add elements of type Object (and, therefore, elements of type Integer) to objList.

It might seem that it would be sufficient for the parameter type to be Collection<E> instead of Collection<? **extends** E>. After all, an instance of the class ArrayList<Object> is legal as the argument corresponding to a parameter of type Collection<E>, so by the Subclass Substitution Rule, an instance of any subclass of ArrayList<Object> would also be legal. But even though Integer is a subclass of Object, ArrayList<Integer> is *not* allowed as a subclass of ArrayList<Object>.[1] Otherwise, the following code fragment would be able to violate the type restrictions by adding a string to an ArrayList of Integer:

```
ArrayList<Integer> intList = new ArrayList<Integer>();
ArrayList<Object> objList = intList;         // illegal!
objList.add ("oops");
```

Then intList would have "oops" at index 0.

Note 3: The new ArrayList object contains a copy of the elements in c. Strictly speaking, those elements are *references*, not objects; the objects referenced are not copied. For this reason, the copy constructor is said to produce a ***shallow copy***.

Note 4: The clone() method is an alternative, but less desirable way to obtain a shallow copy of an ArrayList object. Here is the method specification:

```
/**
 *  Returns a shallow copy of this ArrayList object.
```

[1]This phenomenon is called ***invariant subtyping***, and it is required for type safety. Why? The element type of a parameterized collection is not available at run time, so the type checking of elements cannot be done at run time. This is in contrast to arrays, whose element type *is* available at run time. As a result, arrays use ***covariant subtyping***; for example, String[ ] is a subclass of Object[ ].

```
 *  The worstTime(n) is O(n).
 *
 *  @return a shallow copy of this ArrayList object.
 */
public Object clone()
```

For example, if `myList` is an `ArrayList` object, we can create a shallow copy of `myList` as follows:

```
ArrayList<String> newList = (ArrayList<String>)myList.clone();
```

Unfortunately, there is no assurance of type safety, so the assignment will be made even if `myList` is an `ArrayList` object with `Integer` elements. See Programming Exercise 6.4 for details. For more discussion of `clone` drawbacks, see Bloch 2001, pages 45–52.

Comparison to an array: An array object can be copied with the **`static`** method `arraycopy` in the `System` of the package `java.lang`. For example,

```
System.arraycopy (vegetables, i, moreVeggies, 0, 3);
```

performs a shallow copy of the array object `vegetables`, starting at index `i`, to the array object `moreVeggies`, starting at index 0. A total of 3 elements are copied.

3. **One-parameter `add` method**

```
/**
 *  Appends the specified element to the end of this ArrayList object.
 *  The worstTime(n) is O(n) and averageTime(n) is constant.
 *
 *  @param element - the element to be appended to this ArrayList object
 *
 *  @return true (as per the general contract of the Collection.add method)
 *
 */
public boolean add (E element)
```

Note. According to the general contract of the `add` method in the `Collection` interface, **`true`** is returned if the element is inserted. So this `ArrayList` method will *always* return **`true`**. Then why bother to have it return a value? Because if we replace the return type **`boolean`** with **`void`**, then the `ArrayList` class would no longer implement the `Collection` interface. Incidentally, there are some implementations—the `TreeSet` class, for example—of the `Collection` interface that do not allow duplicate elements, so **`false`** will sometimes be returned when a `TreeSet` object calls this version of the `add` method.

Example We can insert items at the end of an `ArrayList` object as follows:

```
ArrayList<String> fruits = new ArrayList<String> (100);
fruits.add ("oranges");
fruits.add ("apples");
fruits.add ("durian");
fruits.add ("apples");
```

The `ArrayList` object `fruits` will now have “oranges” at index 0, “apples” at index 1, “durian” at index 2, and “apples” at index 3.

Comparison to an array: To insert into an array, an index must be specified:

```
String [ ] vegetables = new String [10];
vegetables [0] = "carrots";
vegetables [1] = "broccoli";
vegetables [2] = "spinach";
vegetables [3] = "corn";
```

4. **The `size` method**

```
/**
 *  Determines the number of elements in this ArrayList object.
 *
 *  @return the number of elements in this ArrayList object.
 *
 */
public int size()
```

Example Suppose we create an `ArrayList` object as follows:

```
ArrayList<String> fruits = new ArrayList<String> (100);
fruits.add ("oranges");
fruits.add ("apples");
fruits.add ("durian");
fruits.add ("apples");
```

Then

```
System.out.println (fruits.size());
```

will output 4.

Comparison to an array: Arrays have nothing that corresponds to a `size()` method. The `length` field contains the capacity of the array, that is, the *maximum* number of elements that can be inserted into the array, not the current number of elements in the array.

5. **The `get` method**

```
/**
 *  Returns the element at the specified index.
 *
 *  @param index - the index of the element to be returned.
 *
 *  @return the element at the specified index
 *
 *  @throws IndexOutOfBoundsException - if index is less than 0 or greater
 *                  than or equal to size()
 */
public E get (int index)
```

Note: Since no time estimates are given, you may assume that worstTime(n) is constant.

Example Suppose we start by constructing an `ArrayList` object:

```
ArrayList<String> fruits = new ArrayList<String> (100);
```

```
fruits.add ("oranges");
fruits.add ("apples");
fruits.add ("durian");
fruits.add ("apples");
```

Then

```
System.out.println (fruits.get (2));
```

will output “durian”.

Comparison to an array: The `get` method is similar to, but weaker than, the index operator for arrays. For example, suppose we start by constructing an array object:

```
String [ ] vegetables = new String [10];
vegetables [0] = "carrots";
vegetables [1] = "broccoli";
vegetables [2] = "spinach";
vegetables [3] = "corn";
```

Then

```
System.out.println (vegetables [1]);
```

Will output “broccoli”. But we can also overwrite that element:

```
vegetables [1] = "potatoes";
```

In contrast, the following is illegal if `fruits` is an `ArrayList` object:

```
fruits.get (1) = "pears";    // illegal
```

6. **The `set` method**

```
/**
 *
 *  Replaces the element at the specified index in this ArrayList object with the
 *  specified element.
 *
 *  @param index - the index of the element to be replaced.
 *  @param element - the element to be stored at the specified index
 *
 *  @return the element previously stored at the specified index
 *
 *  @throws IndexOutOfBoundsException - if index is less than 0 or greater
 *          than or equal to size()
 */
public E set (int index, E element)
```

Note: The worstTime(n) is constant.

Example Suppose we start by constructing an `ArrayList` object:

```
ArrayList<String> fruits = new ArrayList<String> (100);
fruits.add ("oranges");
fruits.add ("apples");
fruits.add ("durian");
fruits.add ("apples");
```

Then

```
System.out.println (fruits.set (2, "bananas"));
```

will change the element at index 2 to ‘‘bananas’’ and output ‘‘durian’’, the element that had been at index 2 before the `set` method was invoked.

Comparison to an array: As noted in the comparison for the `get` method, an array’s index operator can be used on the left-hand side of an assignment statement. For example, if `vegetables` is an array object,

```
vegetables [1] = "potatoes";
```

will change the element at index 1 to “potatoes”.

7. Two-parameter `add` method

```
/**
 *  Inserts the specified element at the specified index in this ArrayList object.
 *  All elements that were at positions greater than or equal to the specified
 *  index have been moved to the next higher position.  The worstTime(n) is
 *  O(n).
 *
 *  @param index - the index at which the specified element is to be inserted.
 *  @param element - the element to be inserted at the specified index
 *
 *  @throws IndexOutOfBoundsException - if index is less than 0 or greater
 *          than size().
 */
public void add (int index, E element)
```

Example Suppose we start by constructing an `ArrayList` object:

```
ArrayList<String> fruits = new ArrayList<String> (100);
fruits.add ("oranges");
fruits.add ("apples");
fruits.add ("durian");
fruits.add ("apples");
```

Then

```
fruits.add (1, "cherries");
for (int i = 0; i < fruits.size(); i++)
       System.out.println (fruits.get (i));
```

will produce output of

```
oranges
cherries
apples
durian
apples
```

Comparison to an array: For an insertion anywhere except at the end of the array object, the code must be written to open up the space. For example, suppose we start by constructing an array object:

```
String [ ] vegetables = new String [10];
vegetables [0] = "carrots";
vegetables [1] = "broccoli";
vegetables [2] = "spinach";
vegetables [3] = "corn";
```

We can insert "lettuce" at index 1 as follows:

```
for (int j = 4; j > 1; j--)
     vegetables [j] = vegetables [j - 1];
vegetables [1] = "lettuce";
```

The array `vegetables` now consists of "carrots", "lettuce", "broccoli", "spinach", "corn", **`null`**, **`null`**, **`null`**, **`null`**, **`null`**. Note that an insertion in a full array will throw an `ArrayIndexOutOfBounds` exception.

8. The `remove` method with an index parameter

```
/**
 *  Removes the element at the specified index in this ArrayList object.
 *  All elements that were at positions greater than the specified index have
 *  been moved to the next lower position.  The worstTime(n) is O(n).
 *
 *  @param index - the index of the element to be removed.
 *
 *  @return the element removed the specified index
 *
 *  @throws IndexOutOfBoundsException - if index is less than 0 or greater
 *          than or equal to size()
 */
public E remove (int index)
```

Example Suppose we start by constructing an `ArrayList` object:

```
ArrayList<String> fruits = new ArrayList<String> (100);
fruits.add ("oranges");
fruits.add ("apples");
fruits.add ("durian");
fruits.add ("apples");
```

Then we can remove (and return) the element at index 2 as follows:

```
System.out.println (fruits.remove (2));
```

The output will be ''durian'', and `fruits` will now contain ''oranges'', ''apples'', and ''apples''.

Comparison to an array: For removal anywhere except at the end of an array, the code must be written to close up the space. For example, suppose we start by creating an array object:

```
String [ ] vegetables = new String [10];
vegetables [0] = "carrots";
vegetables [1] = "broccoli";
vegetables [2] = "spinach";
vegetables [3] = "corn";
vegetables [4] = "potatoes";
vegetables [5] = "squash";
```

Then we can remove the element at index 2 as follows:

```
for (int j = 2; j < 5; j++)
       vegetables [j] = vegetables [j + 1];
```

The array `vegetables` now consists of "carrots", "broccoli", "corn", "potatoes", "squash", **`null`**, **`null`**, **`null`**, **`null`** and **`null`**.

9. **The `indexOf` method**

```
/**
 *  Searches for the first occurrence of a specified element, testing for equality with
 *  the equals method.  The worstTime(n) is O(n).
 *
 *  @param obj – the element to be searched for.
 *
 *  @return the index of the first occurrence of obj in this ArrayList object; if
 *          obj is not in this ArrayList object, -1 is returned.
 *
 */
public int indexOf (Object obj)
```

Example Suppose we start by constructing an `ArrayList` object:

```
ArrayList<String> fruits = new ArrayList<String> (100);
fruits.add ("oranges");
fruits.add ("apples");
fruits.add ("durian");
fruits.add ("apples");
```

Then

```
System.out.println (fruits.indexOf ("apples"));
```

will output 1, and

```
System.out.println (fruits.indexOf ("kiwi"));
```

will output −1.

Note: The type of the parameter `element` is `Object`, not `E`, so the following is legal:

```
System.out.println (fruits.indexOf (new Integer (8)));
```

Of course, the output will be −1, because all the elements in `fruits` are of type `String`.

Comparison to an array: An explicit search must be conducted to determine if an element occurs in an array. For example, suppose we start by creating an array object:

```
String [ ] vegetables = new String [10];
vegetables [0] = "carrots";
vegetables [1] = "broccoli";
vegetables [2] = "spinach";
vegetables [3] = "corn";
vegetables [4] = "potatoes";
vegetables [5] = "squash";
```

If myVeg is a String variable, we can print the index of the first occurrence of myVeg in the vegetables array as follows:

```
boolean found = false;
for (int j = 0; j < 6 && !found; j++)
       if (vegetables [j].equals (myVeg))
       {
               System.out.println (j);
               found = true;
       } // if
if (!found)
        System.out.println (-1);
```

If myVeg does not occur in the array object vegetables, −1 will be output.

These represent just a sampling of the ArrayList class's methods, but even at this point you can see that an ArrayList object is superior, in most respects, to an array object. For example, an ArrayList object's size and capacity are automatically maintained, but an array object's size and capacity must be explicitly maintained by the programmer.

6.2.2 A Simple Program with an ArrayList Object

Perhaps you need more convincing that ArrayList objects are more convenient than array objects. Here is a simple program that creates an ArrayList object from a file of words (one word per line), and then searches for a word in the ArrayList object, removes all instances of a word, appends a word and converts a word to upper case. The resulting ArrayList object is then printed out—with a single call to println. Because this is an illustrative program, there are no constant identifiers.

```
import java.util.*;

import java.io.*;

public class ArrayListExample
{
  public static void main (String[ ] args)
  {
    new ArrayListExample().run();
  } // method main

  public void run()
  {
     ArrayList<String> aList = new ArrayList<String>();

     Scanner keyboardScanner = new Scanner (System.in),
             fileScanner;

     String inFilePath,
            word;
```

```
try
 {
    System.out.print ("\n Please enter the path for the input file: ");
    inFilePath = keyboardScanner.nextLine();
    fileScanner = new Scanner (new File (inFilePath));
    while (fileScanner.hasNext())
    {
      word = fileScanner.next();
      System.out.println (word);
      aList.add (word);
    } // while not end of file

    System.out.print ("\n\n Please enter the word you want to search for: ");
    word = keyboardScanner.next();
    if (aList.indexOf (word) >= 0)
        System.out.println (word + " was found.\n\n");
    else
        System.out.println (word + " was not found.\n\n");

    System.out.print ("Please enter the word you want to remove: ");
    word = keyboardScanner.next();
    int removalCount = 0;
    while (aList.remove (word))
        removalCount++;
    if (removalCount == 0)
        System.out.println (word + " was not found, so not removed.\n\n");
    else if (removalCount == 1)
        System.out.println ("The only instance of " + word +
                            " was removed.\n\n");
    else
        System.out.println ("All " + removalCount + " instances of " +
                            word + " were removed.\n\n");

    System.out.print ("Please enter the word you want to append: ");
    word = keyboardScanner.next();
    aList.add (word);
    System.out.println (word + " was appended.\n\n");

    System.out.print ("Please enter the word you want to upper case: ");
    word = keyboardScanner.next();
    int position = aList.indexOf (word);
    if (position >= 0)
    {
        aList.set (position, word.toUpperCase());
        System.out.println (word + " was converted to upper-case.\n\n");
    } // if word is in aList
    else
        System.out.println (word +
                            " was not found, so not upper-cased.\n\n");
```

```
          System.out.println ("Here is the final version:\n" + aList);
                                        // same as aList.toString()
        } // try
        catch (IOException e)
        {
          System.out.println (e);
        } // catch
      } // method run

} // class ArrayListExample
```

When this program was run, the file `a.in1` contained the following words, one per line:

```
Don’t get mad Don’t get even Get over it all and get on with
```

Here is a sample run, with input in boldface:

```
Please enter the path for the input file: a.in1

Please enter the word you want to search for: even
even was found.


Please enter the word you want to remove: all
The only instance of all was removed.


Please enter the word you want to append: life
life was appended.


Please enter the word you want to convert to upper case: over
over was converted to upper-case.
```

Here is the final version:

```
[Don’t, get, mad, Don’t, get, even, Get, OVER, it, and, get, on, with, life]
```

In the above program, each removal takes linear time. Programming Exercise 6.8 suggests how to perform all removals in a single loop. And you are invited, in Programming Exercise 6.9, to endure the grind of converting the program from `ArrayList`-based to array-based.

In Sections 6.2.3 and 6.2.4, we briefly put on a developer’s hat and look at the `ArrayList` class heading, fields and a few method definitions. In Section 6.3, we return to a user’s perspective with an application of the `ArrayList` class.

6.2.3 The `ArrayList` Class’s Heading and Fields

Here is the heading of the `ArrayList` class:

```
public class ArrayList<E> extends AbstractList<E>
       implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

This says that the `ArrayList` class is a subclass of the class `AbstractList`, and implements four interfaces: `List`, `RandomAccess`, `Cloneable`, and `Serializable`. Figure 6.2 has a UML diagram to indicate where the `ArrayList` class fits in the Java Collections Framework, with a solid-line arrow from an extension (to a class or interface) and a dashed-line arrow from a class to an interface implemented by the class.

The `AbstractCollection` class provides a minimal implementation of the `Collection` interface, just as the `AbstractList` class provides a "bare bones" implementation of the `List` interface. As we saw in Section 6.1, the `List` interface extends the `Collection` interface by including some index-related methods, such as `get (int index)` and `remove (int index)`.

Basically, a class that implements the `Cloneable` interface must have a method that returns a shallow copy of the calling object. For a description of the `clone()` method, see Note 4 on the copy constructor (method number 2) in Section 6.2.1. The `RandomAccess` interface ensures that if an implementation of the `List` interface satisfies the random-access property (with an underlying array), then any sub-list of that list will also satisfy the random-access property. The `Serializable` interface, discussed in Appendix 1, has to do with saving objects to a stream (such as a disk file), which is called ***serialization***, and restoring those object from the stream, called ***deserialization***.

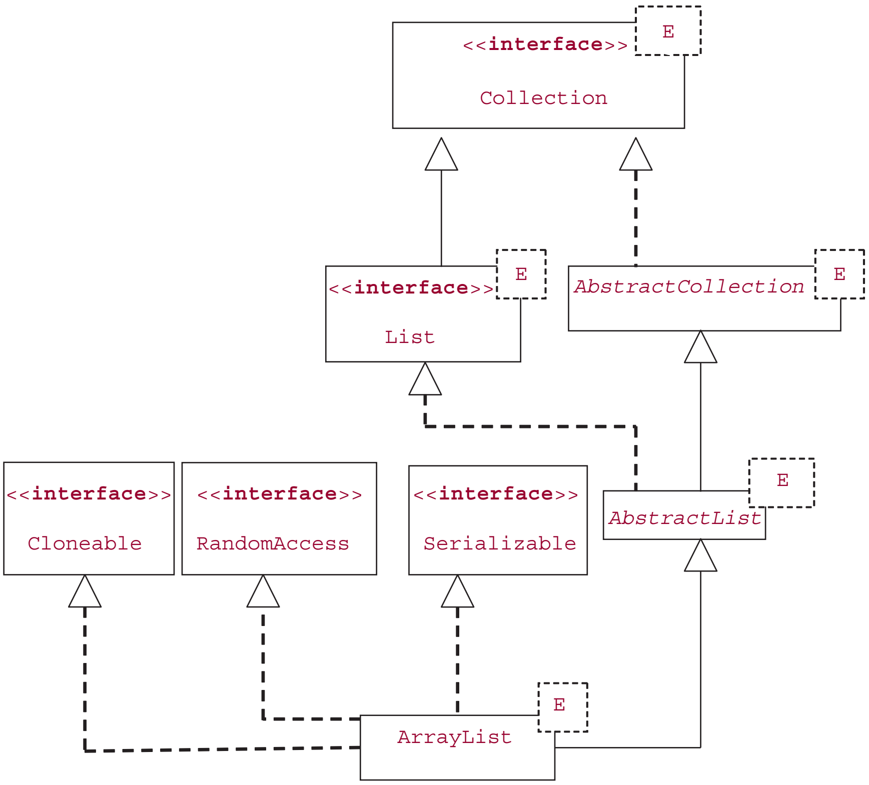


FIGURE 6.2 The UML diagram to illustrate the relative position of the `ArrayList<E>` class in the Java Collections Framework

It may come as no surprise to you that the ArrayList class has an array field:

```
private transient E[  ] elementData;
```

The reserved word **transient** indicates that this field is not saved during serialization (see Appendix 1). That is, each element would be saved, but not the entire array. The field is **private** instead of **protected** because the developers of the Java Collections Framework were opposed to giving users who subclass direct access to a superclass's fields. See Section 2.6 for a discussion of this choice.

The only other field defined in the ArrayList class is

```
private int size;
```

So an ArrayList object has an array field to store the elements and an **int** field to keep track of the number of elements.

We will finish up our developer's view of the ArrayList class by studying the implementation of the add method that appends an element to the end of the calling ArrayList object.

6.2.4 Definition of the One-Parameter add Method

To give you an idea of how expansion of an ArrayList object is accomplished, let's look at the definition of the one-parameter add method:

```
public boolean add (E element)
{
    ensureCapacity (size + 1);
    elementData [size++] = element;
    return true;
}
```

The call to the ensureCapacity method expands the underlying array, if necessary, to accommodate the new element; we'll get to the details of that method momentarily. Then the new element, element, is inserted at index size in the array, size is incremented, and **true** is returned. Suppose that fruits has been constructed as an empty ArrayList by a default-constructor call, and the next message is

```
fruits.add ("oranges");
```

After that message is processed, the elementData and size fields in fruits will have the contents shown in Figure 6.3.

Now let's get back to the ensureCapacity method. If the underlying array is not filled to capacity, then the call to ensureCapacity does nothing. But if size == elementData.length, then the argument size + 1 must be greater than elementData.length, so we need to expand the array. First, the array's current reference, elementData, is copied to oldData:

```
E oldData [ ] = elementData;
```

This does not make a copy of the array, just a copy of the reference. Then a new array object is constructed:

```
elementData = (E[ ]) new Object [newCapacity];
```

where (because the argument was size + 1) the variable newCapacity was given a value about 50% larger than oldData.length. The cast was necessary because the **new** operator must be applied to a "real" type, not to a type parameter (such as E). Finally, the arraycopy method in the System class is

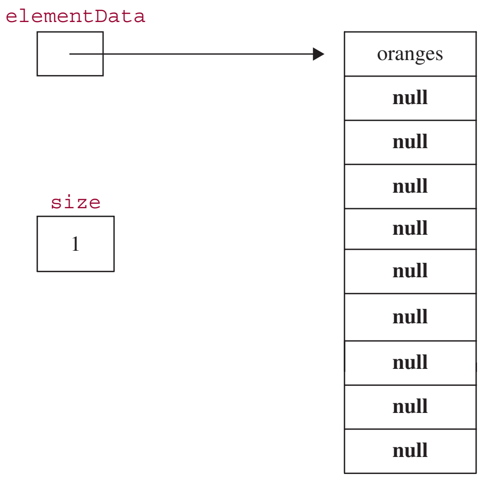


FIGURE 6.3 The contents of the elementData and size fields in the ArrayList object fruits after the message fruits.add ("oranges") is sent. As usual, we pretend that the non-null elements in the array are objects; in fact, the elements are references to objects

called to copy all the elements from oldData to elementData; the number of elements copied is the value of size.

Here is the complete definition:

```
public void ensureCapacity(int minCapacity)
{
       modCount++;   // discussed in Appendix 1
       int oldCapacity = elementData.length;
       if (minCapacity > oldCapacity)
       {
           E oldData[ ] = elementData;
           int newCapacity = (oldCapacity * 3) / 2 + 1;
           if (newCapacity < minCapacity)  // can't happen if argument is size + 1
                newCapacity = minCapacity;
           elementData = (E[ ]) new Object [newCapacity];
           System.arraycopy(oldData, 0, elementData, 0, size);
       }
}
```

To see the effect of an expansion, suppose that the ArrayList object fruits already has ten elements and the following message is sent:

```
fruits.add ("cantaloupes");
```

Figure 6.4 shows the effect of this message on the elementData and size fields of fruits.

What are the time estimates of the one-parameter add method? Let n represent the number of elements in the calling ArrayList object. In the worst case, we will have $n =$ elementData.length, and so, in the ensureCapacity method, we will have minCapacity > oldCapacity. Then the call to arrayCopy entails copying n elements from oldData to elementData. We conclude that worstTime(n) is linear in n.

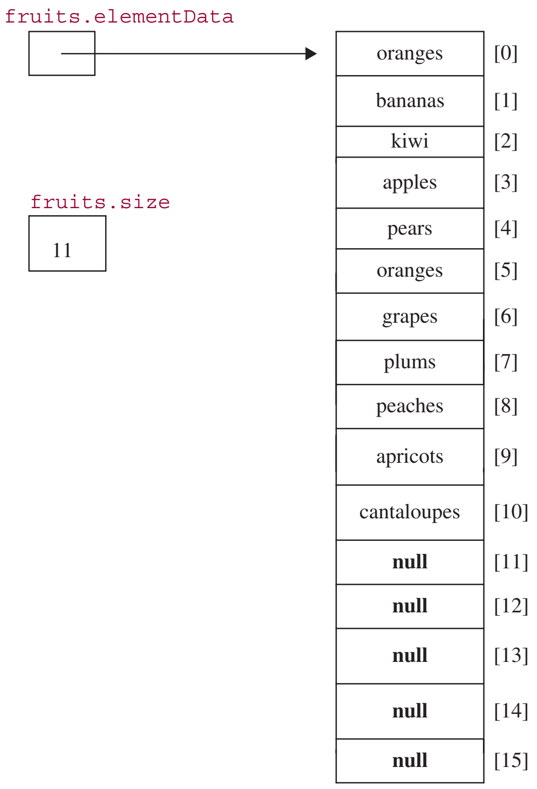


FIGURE 6.4 The contents of the `elementData` and `size` fields in the `ArrayList` object `fruits`, if `fruits` already had ten elements when the message `fruits.add ("cantaloupes")` was sent. As usual, we pretend that the non-null elements in the array are objects instead of references

What about the average case? The only occasion for copying occurs when n = `elementData.length`. But then, by the details of the `ensureCapacity` method, no copying would have occurred in the previous $n/3$ (approximately) calls to the one-parameter `add` method. So in $n/3 + 1$ calls to that `add` method, the total number of elements copied would be n, and the average number of elements copied per call would be about 3. We conclude, since the only non-constant-time code in the `ensureCapacity` method is in the initialization of `elementData` and in the call to `arrayCopy`, that averageTime(n) is constant for the one-parameter `add` method.

Incidentally, the developers of the `ArrayList` class could have doubled `oldCapacity` instead of increasing it by about 50%. There is a trade-off: with doubling, additional space is allocated immediately, but then there will be a longer period before the next re-sizing occurs. In fact, in the C++ analogue of the `ArrayList` class, the old capacity *is* doubled when a re-sizing occurs.

The previous examination of fields and implementation details is intended just to give you the flavor of the developer's view of the `ArrayList` class. A few more `ArrayList` method-definitions are covered in Lab 10. Of course, all of the `ArrayList` definitions are available in the `ArrayList` (or `AbstractList` or `AbstractCollection`) class of java.util.

You are now prepared to do Lab 10: More Details on the ArrayList Class

Section 6.3 presents an application of the ArrayList class, so the emphasis once again is on the user's viewpoint.

6.3 Application: High-Precision Arithmetic

We now introduce high-precision arithmetic as an application of the ArrayList class. We will get to the details shortly, but it is worth recalling that the ***use*** of a class is independent (except for efficiency) of ***how*** the class is implemented. So we are not locked in to any particular implementation of the ArrayList class.

In public-key cryptography (see Simmons [1992]), information is encoded and decoded using integers more than 100 digits long. The essential facts about the role of these ***very long integers*** in public-key cryptography are:

1. It takes relatively little time—$O(n^3)$—to generate a very long integer with n digits that is prime[2]. For example, suppose we want to generate a prime number that has 500 digits. Then the number of loop iterations required is approximately $500^3 = 125,000,000$.
2. It takes a very long time—currently, $\Omega(10^{n/2})$—to determine the prime factors of a very long integer with n digits that is not prime. For example, suppose we want to factor a non-prime with 500 digits. Then the number of loop iterations required is approximately $(10^{500/2}) = 10^{250}$.
3. Assume that you have generated p and q, two very long integers that are prime. You then calculate another prime e to be greater than pq. The product pq can be calculated quickly, and you supply this product, and e, to anyone who wants to send you a message, M. First, the sender splits the message M up into sequences of characters $M_1, M_2, \ldots$. The sequence M_i is then treated as a very long integer V_i by concatenating the bits in each character in M_i. The encrypted integer corresponding to V_i is V_i^e % pq. That is, we raise V_i to the power e and then take the remainder when the result of that exponentiation is divided by pq. This seems complicated, but in fact, the calculation can be performed relatively quickly. (See Simmons, [1992] for details.) The encoded message, as well as pq and e, are ***public***, that is, transmitted over an insecure channel such as a telephone, postal service, or computer network.
4. But decoding the message requires knowing the values of p and q. Since determining the factors p and q takes a prohibitively long time, only you can decode the message.

Very long integers require far greater precision than is directly available in programming languages. We will now design and implement a simple version of the VeryLongInt class. Exercise 6.5 asks you to amplify this version, Lab 12 involves the testing of the amplified version, and Programming Assignment 6.1 further expands the VeryLongInt class.

For an industrial-strength class that is applicable to public-key cryptography, see the BigInteger class in java.math. The BigInteger class includes efficient methods for primality testing, multiplication, and modular exponentiation.

[2]An integer $p > 1$ is **prime** if the only positive-integer factors of p are 1 and p itself.

6.3.1 Method Specifications and Testing of the VeryLongInt Class

There will be only three methods: A very long integer can be constructed from a string, converted to a string, or incremented by another very long integer. Here are the method specifications, with examples:

1.
```
/**
 * Initializes this VeryLongInt object from the digit characters in a
 * given String object.
 * There are no leading zeros, except for 0 itself, which has a single '0'.
 * The worstTime(n) is O(n), where n represents the number of characters in s.
 *
 * @param s - the given String object.
 *
 * @throws NullPointerException - if s is null.
 * @throws IllegalArgumentException - if s contains no digit characters.
 *
 */
public VeryLongInt (String s)
```

   Example Suppose we have

```
VeryLongInt veryLong = new VeryLongInt ("11223?344556677889900");
```

   Then `veryLong` will be initialized to the `VeryLongInt` object whose integer value is 112233445566778899 00. The '?' is ignored because it is not a digit character. The value is greater than the largest **int** value.

2.
```
/** Returns a String representation of this VeryLongInt object. The worstTime(n) is
 * O(n), where n represents the number of digits in this VeryLongInt object.
 *
 * @return a String representation of this VeryLongInt object in the form '[' followed
 *         by the digits, separated by commas and single spaces, followed by ']'.
 *
 */
public String toString()
```

   Example Suppose we have

```
VeryLongInt veryLong = new VeryLongInt ("52?481");
System.out.println (veryLong); // same as
                               // System.out.println (veryLong.toString());
```

   The output would be

```
[5, 2, 4, 8, 1]
```

3.
```
/**
 *  Increments this VeryLongInt object by a specified VeryLongInt object.
 *  The worstTime(n) is O(n), where n is the number of digits in the larger of this
 *  VeryLongInt object (before the call) and the specified VeryLongInt object.
 *
 *  @param otherVeryLong - the specified VeryLongInt object to be added to
 *         this VeryLongInt object.
 *
 *  @throws NullPointerException – if otherVeryLong is null.
 *
 */
public void add (VeryLongInt otherVeryLong)
```

Example Suppose that `newInt` and `oldInt` are `VeryLongInt` objects with values of 328 and 97, respectively, and the message sent is

```
newInt.add (oldInt);
```

Then the value of `newInt` has become 425.

Note: This method performs the arithmetic operation of addition. Contrast that to the `ArrayList` class's one-paramter `add` method, which appends the argument to the calling `ArrayList` object.

The book's website has a `VeryLongIntTest` class, with the following fields:

```
protected VeryLongInt very;

protected String answer;
```

That class also includes the following tests, one for a constructor call with an argument that has no digits, and one for a simple addition:

```
@Test (expected = IllegalArgumentException.class)
public void testConstructorWithNoDigits()
{
     very = new VeryLongInt ("x t?.o");
} // method testConstructorWithNoDigits

@Test
public void testAdd()
{
   very = new VeryLongInt ("99");
   VeryLongInt other = new VeryLongInt ("123");
   very.add (other);
   answer = very.toString();
   assertEquals ("[2, 2, 2]", answer);
} // method testAdd
```

6.3.2 Fields in the `VeryLongInt` Class

As often happens in developing a class, the major decision involves the field(s) to represent the class. Should we store a very long integer in an array-based structure such as an `ArrayList`, or would a linked structure be better? (An array itself is not a good idea because then we would have to write the code—for example, to keep track of the number of elements in the array—instead of simply calling methods). In this chapter, we will use the `ArrayList` class and represent each very long integer as a sequence of digits. In Chapter 7, we will consider a linked structure.

Which is the appropriate relationship between `VeryLongInt` and `ArrayList`: **is-a** (inheritance) or **has-a** (aggregation)? That is, should `VeryLongInt` be a subclass of `ArrayList`, or should `VeryLongInt` have a field of type `ArrayList` ? The primary purpose of the `VeryLongInt` class is to perform arithmetic; as such, it shares little functionality with the `ArrayList` class. So it makes more sense to say "a VeryLongInt object **has-an** ArrayList field" than "a VeryLongInt object **is-an** ArrayList object." The only field in the VeryLongInt class will be an ArrayList object whose elements are of type Integer:

```
protected ArrayList<Integer> digits;
```

Each element in the ArrayList object digits will be an Integer object whose value is a single digit (Exercise 6.6 expands each value to a five-digit integer).

Figure 6.5 has the UML diagram for the VeryLongInt class.

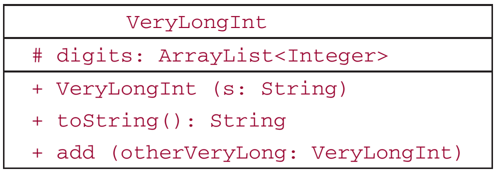

VeryLongInt
# digits: ArrayList<Integer>
+ VeryLongInt (s: String)
+ toString(): String
+ add (otherVeryLong: VeryLongInt)

FIGURE 6.5 The class diagram for the VeryLongInt class

6.3.3 Method Definitions of the VeryLongInt Class

The digits in the ArrayList field digits will be stored from left-to-right, that is, in their normal order. For example, if the underlying integer value of a VeryLongInt object is 328, we would store the 3 at index 0 in digits, the 2 at index 1, and the 8 at index 2.

Notice that by having the insertions take place at the end of the ArrayList object, we take advantage of the average-case speed of the ArrayList class's one-parameter add method, which is not related to the VeryLongInt method named add. If, instead, we stored a number in reverse order, we would be repeatedly inserting at the front of the ArrayList object digits. Exercise 6.7 explores the effect on efficiency if the digits are stored in reverse order.

We now define the String -parameter constructor, the toString() method and the add method. While we do we will keep in mind the strengths (fast random access and fast end-insertions) and weakness (slow insertions at other-than-the-end positions) of the ArrayList class.

For the String-parameter constructor, we loop through the characters in s. For each character in s, if the character is a digit character, we convert that character to the corresponding digit by subtracting the Unicode representation of '0' from the character. For example,

```
'7' - '0' = 7
```

Finally, we append that digit (as an Integer) to digits. The ArrayList field digits never needs resizing during the execution of this constructor because that field is constructed with initial capacity of s.length(). Here is the code:

```
public VeryLongInt (String s)
{
        final char LOWEST_DIGIT_CHAR = '0';

        digits = new ArrayList<Integer> (s.length());

        char c;

        int digit;
```

```
          boolean atLeastOneDigit = false;

          for (int i = 0; i < s.length(); i++)
          {
                 c = s.charAt (i);
                 if (Character.isDigit(c))
                 {
                         digit = c - LOWEST_DIGIT_CHAR;
                         digits.add (digit);   // digit is boxed to an Integer object
                         atLeastOneDigit = true;
                 } //  if a digit
          } // for
          if (!atLeastOneDigit)
                 throw new IllegalArgumentException();
   } // constructor with string parameter
```

How long will this method take? Assume that there are n characters in the input. Then the loop will be executed n times. For the `ArrayList` class's one-parameter `add` method, averageTime (n) is constant, so for this constructor, averageTime(n) is linear in n (that is, O(n) and $\Omega(n)$). As we saw in the analysis of that `add` method, if n represents the number of elements in the `ArrayList` object, worstTime(n) is O(n) for n calls to that `add` method, So for this constructor in the `VeryLongInt` class, worstTime(n) is O(n). In fact, because worstTime$(n) \geq$ averageTime(n) and averageTime(n) is $\Omega(n)$, worstTime(n) must be $\Omega(n)$. We conclude that worstTime(n) is linear in n.

For the `toString()` method, we simply invoke the `ArrayList` class's `toString()` method:

```
   public String toString()
   {
        return digits.toString();
   } // method toString
```

For an example of a call to this method, if `veryLong` is a `VeryLongInt` object with a value of 6713, the output from the call to

```
System.out.println (veryLong);    // same as System.out.println (veryLong.toString());
```

will be

```
[6, 7, 1, 3]
```

For this method, worstTime(n) is linear in n, the number of digits in the calling `VeryLongInt` object. To convince yourself of this estimate, look at the definition of the `toString()` method in the `AbstractCollection` class, a superclass of `ArrayList`.

Finally, we tackle the `add (VeryLongInt otherVeryLong)` method in the `VeryLongInt` class. We obtain partial sums by adding `otherVeryLong` to the calling object digit-by-digit, starting with the least significant digit in each number. Each partial sum, divided by 10, is appended to the end of the `ArrayList` object `sumDigits`, which is initially empty.

Because we will be using the `ArrayList` class's one-parameter `add` method on the partial sums, we must reverse `sumDigits` after adding so that the most significant digit will end up at index 0. For example, suppose `newInt` is a `VeryLongInt` object with the value 328 and `oldInt` is a `VeryLongInt` object with the value 47. If the message is

```
   newInt.add (oldInt);
```

then after adding and appending the partial sums to the VeryLongInt object sum, sum will have the value 573. When this is reversed—by the generic algorithm reverse in the Collections class of the package java.util—the sum will be correct. Note that the add method in the ArrayList class is used to append a digit to the end of sumDigits; the ArrayList class's add method does not perform arithmetic.

Here is the definition of the add method in the VeryLongInt class:

```
public void add (VeryLongInt otherVeryLong)
{
       final int BASE = 10;

       int largerSize,
           partialSum,
           carry = 0;

       if (digits.size() > otherVeryLong.digits.size())
                largerSize = digits.size();
       else
                largerSize = otherVeryLong.digits.size();

       ArrayList<Integer> sumDigits = new ArrayList<Integer> (largerSize + 1);

       for (int i = 0; i < largerSize; i++)
       {
              partialSum = least (i) + otherVeryLong.least (i) + carry;
              carry  = partialSum / BASE;
              sumDigits.add (partialSum % BASE);
       } // for

       if (carry == 1)
              sumDigits.add (carry);
       Collections.reverse (sumDigits);
       digits = sumDigits;
} // method add
```

The call to the least method with an argument of i returns the i^{th} least significant digit in the calling object's digits field. The units (rightmost) digit is considered the 0th least significant digit, the tens digit is considered the 1st least significant digit, and so on. For example, suppose that the calling VeryLongInt object has the value 3284971, and i has the value 2. Then the digit returned by the call to least (2) will be 9 because 9 is the 2nd least significant digit in the calling object's digits field; the 0th least-significant digit is 1 and the 1st least-significant digit is 7. The method definition is:

```
/** Returns the ith least significant digit in digits if i is a non-negative int less than
 *  digits.size().  Otherwise, returns 0.
 *
 *  @param i – the number of positions from the right-most digit in digits to the
 *             digit sought.
 *
 *  @return the ith least significant digit in digits, or 0 if there is no such digit.
 *
 *  @throws IndexOutOfBoundsException – if i is negative.
 *
 */
protected int least (int i)
{
       if (i >= digits.size())
               return 0;
       return digits.get (digits.size() - i - 1);
} // least
```

For the `least` method, worstTime(n) is constant because for the `size` and `get` methods in the `ArrayList` class, worstTime(n) is constant.

We can now estimate the time requirements for the `VeryLongInt` class's `add` method. Assume, for simplicity, that the calling object and `otherVeryLongInt` are very long integers with n digits. There will be n iterations of the **`for`** loop in the definition of the `add` method, and during each iteration, a digit is appended to `sumDigits`. For appending n elements to the end of an `ArrayList`, worstTime(n) is linear in n; see Exercise 6.2. The `reverse` generic algorithm also takes linear-in-n time, so for the `add` method in the `VeryLongInt` class, worstTime(n) is linear in n.

The book's website has a class, `VeryLongIntUser`, to demonstrate how an end user might work with the `VeryLongInt` class. The `run` method inputs a line from the keyboard, calls a `process` method to parse the line and invoke the appropriate method,, and outputs the result of processing to the screen. For the testing of that `process` method, see the test class, `VeryLongIntUserTest`, also on the book's website.

Programming Exercise 6.7 expands on the `VeryLongInt` class. You should complete that exercise before you attempt to do Lab 11.

You are now prepared to do Lab 11: Expanding the `VeryLongInt` Class

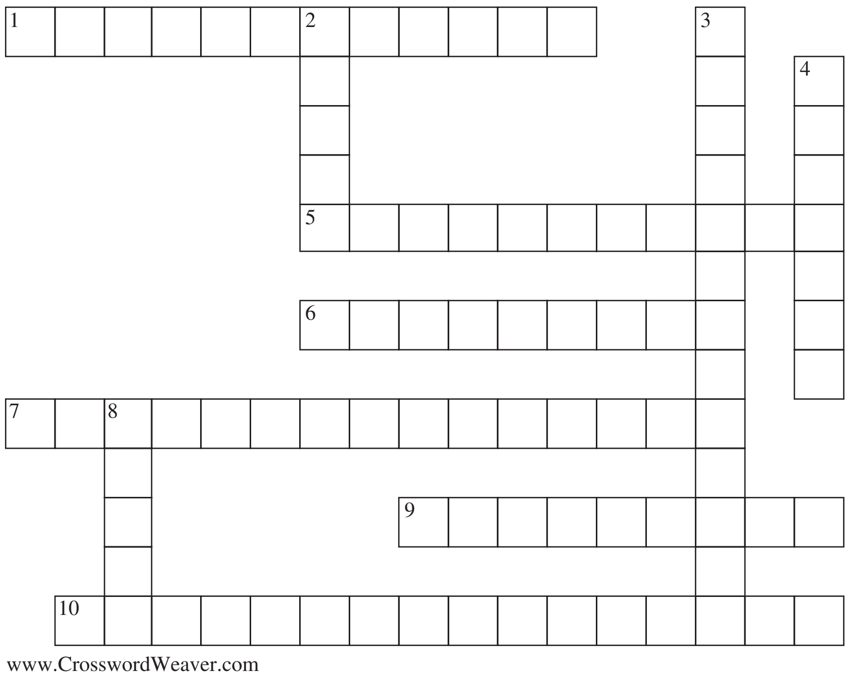
Exercise 7.7 explores the modifications needed to develop the `VeryLongInt` class with `digits` a `LinkedList` field instead of an `ArrayList` field.

SUMMARY

In this chapter we introduced the `List` interface, which extends the `Collection` interface by adding several index-based methods. We then studied the `ArrayList` class, an implementation of the `List` interface that allows random-access—that is, constant-time access—of any element from its index. Using an `ArrayList` object is similar to using an array, but one important difference is that `ArrayList` objects are automatically resizable. When an `ArrayList` outgrows the current capacity of its underlying array, an array of 1.5 times that size is created, and the old array is copied to the larger array. This is similar to what hermit crabs do each time they outgrow their shell. A further advantage of `ArrayList` object over arrays is that, for inserting and deleting, users are relieved of the burden of writing the code to make space for the new entry or to close up the space of the deleted entry.

The application of the `ArrayList` class was in high-precision arithmetic, an essential component of public-key cryptography.

CROSSWORD PUZZLE



ACROSS

1. The immediate superclass of `ArrayList`.

5. Currently, determining the prime factors of a non-prime very long integer of *n* digits requires__________ in *n* time.

6. The fact that `String[ ]` is a subclass of `Object[ ]` is an example of _____________ subtyping.

7. A constructor that initializes the calling object to a copy of the argument corresponding to the given parameter.

9. The fact that `ArrayList<String>` is not a subclass of `ArrayList<Object>` is an example of _____________ subtyping.

10. In public-key cryptography, information is encoded and decoded using _______________.

DOWN

2. The ___________ method, because it does not guarantee type safety, is inferior to the copy constructor for obtaining a copy of an `ArrayList` object.

3. An interface that extends the `Collection` interface with methods that have an index as either a parameter or a return type.

4. Because the elements in any `Collection` object are references, the `ArrayList`'s copy constructor is said to produce a _______ copy.

8. A positive integer greater than 1 that has no positive-integer factors other than 1 and itself is called a _____________ number.

CONCEPT EXERCISES

6.1 State two advantages, and one disadvantage, of using an `ArrayList` object instead of an array object.

6.2 Show that, for the task of appending *n* elements to an `ArrayList` object, worstTime(*n*) is linear in *n*.

6.3 The one-parameter `add` method in the `ArrayList` class always returns **`true`**. Would it make sense to change the return type from **`boolean`** to **`void`** ? Explain.

6.4 For the one-parameter `add` method in the `ArrayList` class, estimate worstSpace(*n*) and averageSpace(*n*).

6.5 In choosing fields for the `VeryLongInt` class, we decided to use, rather than inherit from, the `ArrayList` class. Why?
Hint: How much commonality is there between the methods in the `ArrayList` class and the methods in the `VeryLongInt` class?

6.6 Suppose you modified the `VeryLongInt` class as follows: each element in `digits` consists of a five-digit integer. What effect do you think this will have on Big-O time? What about run-time?

6.7 Suppose, in developing the `VeryLongInt` class, we decide that `digits` will contain the integer in reverse order. For example, if the constructor call is:

```
VeryLongInt veryLong = new VeryLongInt ("386");
```

we would have (`Integer` elements with values) 6, 8, 3 in positions 0 through 2, respectively of `digits`. Re-design this constructor so that worstTime(*n*) is still linear in `n`.

6.8 Which parts of the `VeryLongInt` methods would have to be re-written if `digits` were an array object of **`int`** elements instead of an `ArrayList` object of `Integer` elements?

6.9 How can a user of the `VeryLongInt` class easily create a `VeryLongInt` object that is a copy of an already existing `VeryLongInt` object?

PROGRAMMING EXERCISES

6.1 Hypothesize the output from the following code, and then test your hypothesis with a small program that includes the code:

```
ArrayList<String> letters = new ArrayList<String>();

letters.add ("f");
letters.add (1, "i");
letters.add ("e");
letters.add (1, "r");
letters.add ("e");
letters.add (4, "z");
System.out.println (letters);

letters.remove ("i");
int index = letters.indexOf ("e");
letters.remove (index);
letters.add (2, "o");
System.out.println (letters);
```

6.2 For each of the following program segments, hypothesize if the segment would generate a compile-time error, a run-time exception, or neither. Then test your hypotheses with a `main` method that includes each segment.

a.
```
ArrayList<String> myList = new ArrayList<String>();
myList.add ("yes");
myList.add (7);
```

b.
```
ArrayList<Double> original = new ArrayList<Double>();
original.add (7);
```

c.
```
ArrayList<Integer> original = new ArrayList<Integer>();
double x = 7;
original.add (x);
```

d.
```
ArrayList<String> newList = new ArrayList<String>();
newList.add ("yes");
Integer answer = (Integer)newList.get (0);
```

6.3 Suppose we have the following code:

```
ArrayList<String> myList = new ArrayList<String>();

myList.add ("Karen");
myList.add ("Don");
myList.add ("Mark");

ArrayList<String> temp = new ArrayList<String> (myList);
ArrayList<String> sameList = myList;

myList.add (1, "Courtney");
```

Hypothesize what the contents of `myList`, `temp`, and `sameList` will be after this last insertion. Then test your hypothesis with a `main` method that includes the code.

6.4 Hypothesize what will happen when the following code fragment is run, and then test your hypothesis:

```
ArrayList<String> original = new ArrayList<String>();
original.add ("yes");
ArrayList<Integer> copy = (ArrayList<Integer>)original.clone();

System.out.println (copy.get (0));
```

Hint: This exercise illustrates why the copy constructor is superior to the `clone()` method.

6.5 Expand the `VeryLongInt` class by testing and defining methods that have the following method specifications:

a.
```
/**
 *  Initializes this VeryLongInt object from a given int.
 *
 *  @param n - the int from which this VeryLongInt is initialized.
 *
 *  @throws IllegalArgumentException - if n is negative.
 *
 */
public VeryLongInt (int n)
```

b.

```
/**
 *  Returns the number of digits in this VeryLongInt object.
 *
 *  @return the number of digits in this VeryLongInt object.
 *
 */
public int size()
```

c.

```
/**
 *  Returns true if this VeryLongInt object is less than another VeryLongInt
 *  object. The worstTime(n) is O(n).
 *
 *  @param otherVeryLong – the other VeryLongInt object.
 *
 *  @return true – if this VeryLongInt is less than otherVeryLong.
 *
 *  @throws NullPointerException – if otherVeryLong is null
 *
 */
public boolean less (VeryLongInt otherVeryLong)
```

d.

```
/**
 *  Returns true if this VeryLongInt object is greater than another VeryLongInt
 *  object. The worstTime(n) is O(n).
 *
 *  @param otherVeryLong – the other VeryLongInt object.
 *
 *  @return true – if this VeryLongInt is greater than otherVeryLong.
 *
 *  @throws NullPointerException – if otherVeryLong is null
 *
 */
public boolean greater (VeryLongInt otherVeryLong)
```

e.

```
/**
 *  Returns true if this VeryLongInt object is equal to a specified object.
 *  The worstTime(n) is O(n).
 *
 *  @param obj – the specified object that this VeryLongInt is compared to.
 *
 *  @return true – if this VeryLongInt is equal to obj.
 *
 */
public boolean equals (Object obj)
```

f.

```
/**
 *  Stores a Fibonacci number in this VeryLongInt object.
 *
 *  @param n – the index in the Fibonacci sequence
 *
 *  @throws IllegalArgumentException – if n is not positive
 *
```

```
  */
public void fibonacci (int n)
```

Example Suppose the following message is sent

```
tempInt.fibonacci (100);
```

Then tempInt's value will be 354224848179261915075—the 100th Fibonacci number.

Hint: Mimic the iterative design of the Fibonacci function from Lab 7. Both i and n will be ordinary **int** variables, but previous, current and temp will be VeryLongInt objects. After the loop, instead of returning current, the calling object is modified by assigning to digits a copy of current.digits.

6.6 Assume that myList is (a reference to) an ArrayList<Double> object and that both i and j are int variables with values in the range from 0 to myList.size() -1, inclusive. Hypothesize what the following accomplishes, and then test your hypothesis.

```
myList.set (i, myList.set (j, myList.get (i)));
```

6.7 Describe how to find the method definitions for the ArrayList class in your computing environment.

6.8 Modify the simple program in Section 6.2.2 so that all removals are performed in a single loop.
Hint: Create a temporary ArrayList object to hold the un-removed elements. What is a drawback to this approach?

6.9 Convert the simple program in Section 6.2.2 into one that uses an array object instead of an ArrayList object.

6.10 Modify the simple program in Section 6.2.2 to use a binary search instead of the sequential search used in the call to the indexOf method. The Collections class in java.util has a binarySearch method and a sort method.

6.11 Suppose scoreList is an ArrayList object of Integer elements, and the following message is sent:

```
scoreList.remove (3);
```

Does this message remove the element at index 3, or remove the first occurrence of **new** Integer (3)? Test your hypothesis.

6.12 Suppose we create the following ArrayList instance:

```
ArrayList<String> words = new ArrayList<String>();
```

And then we insert several words into words. Write the code to print out each element of words that has exactly four letters. You should have three different versions of the code:

a. using an index;

b. using an explicit iterator;

c. using an enhanced **for** statement.

6.13 Test and define the following method

```
/**
 * In a given ArrayList, remove all duplicates.
 * The worstTime(n) is O(n^2).
 *
 * @param list - the given ArrayList.
 *
 * @return - An ArrayList that is identical to list except only the first
 *           occurrence of duplicate elements remains.
 *
 * @throws NullPointerException - if list is null.
 *
 */
public static <T> ArrayList <T> uniquefy (ArrayList <T> list)
```

For example, suppose `myList` consists of references to `Integer` objects with the following values, in sequence

3, 8, 6, 4, 8, 7, 8, 9, 4

Then the `ArrayList` returned by the call to `uniquefy (myList)` will consist of references to `Integer` objects with the following values, in sequence

3, 8, 6, 4, 7, 9

Programming Project 6.1

Expanding the `VeryLongInt` Class

In the `VeryLongInt` class, test and define a `multiply` method and a `factorial` method. Here is the method specification for `multiply`:

```
/** Stores in this VeryLongInt object the product of its pre-call value and the value
 *  of a specified VeryLongInt object.  The worstTime(n) is O(n * n), where n is
 *  the maximum of the number of digits in the pre-call value of this
 *  VeryLongInt object and the number of digits in the specified VeryLongInt object.
 *
 *  @param otherVeryLong - the specified VeryLongInt object to be multiplied by
 *         this VeryLongInt object.
 *
 *  @throws NullPointerException - if otherVeryLong is null
 *
 */
public void multiply (VeryLongInt otherVeryLong)
```

For `factorial`:

```
/**
 *  Stores, in this VeryLongInt object, the product of all integers between 1 and
 *  specified integer n.  The worstTime(n) is O(n log (n!)): n multiplications, and
 *  each product has fewer digits than log (n!), the number of digits in n!
 *
 *  @param n - the number whose factorial will be stored in this VeryLongInt
 *             object.
 *
 *  @throws IllegalArgumentException - if n is negative.
```

(continued on next page)

(continued from previous page)

```
 *
 */
public void factorial (int n)
```

Use unit testing to test your methods.

Programming Project 6.2

An Integrated Web Browser and Search Engine, Part 2

This is the second part of a sequence of related projects to create an integrated web browser and search engine.

Problem Convert a text file into a list of tokens, one per line.

Analysis A program that transforms an input file in one form into an output file in another form is called a ***filter***. For the sake of further parts of this project, you will transform an input file into a list. Develop a `Filter` class with the following two methods:

```
/**
 *  Initializes this Filter object from the paths for the input file
 *  and common words.
 *
 *  @param inFilePath - the path for the input file.
 *  @param commonFilePath - the path for the file with the common words.
 *
 *  @throws IOException - if either file does not exist.
 *
 */
public Filter (String inFilePath, String commonFilePath) throws IOException

/**
 *  Creates the ArrayList of tokens from the input file.
 *
 *  @return - an ArrayList of tokens.
 *
 *
 */
public ArrayList<String> createList()
```

1. The tokens in the returned `ArrayList` will not include common words such as “a”, “and”, and “in”. The end-user will specify a file of common words, one per line. Here are the (sample) contents of that file:

   a

   an

   and

   are

did

down

in

the

where

to

You should assume that the file of common words is large enough so that it should be read in only once, and stored without a lot of unused space (the title of this chapter is a hint for the storage structure). Each search of the common words should take O(log n) time in the worst case. The file of common words may not be in alphabetical order, but the stored common words should be in alphabetical order (see Collections.java in java.util).

2. The tokens will not include tags from the input file, that is, all characters between '<' and '>'. You may assume that each '<' will be followed on the same line by a matching '>'. You may assume that the text between two link tags consists of a single word. That is, you might have <a href = ... >singleword</a>.
For example, suppose a line in the input file consists of

Caverns are <a href =browser.in2>browser2</a> measureless to man

Then the tokens would be

caverns

browser2

measureless

man

3. Other than the restrictions of 1 and 2 above, the returned `ArrayList` will consist of all words, lowercased, in the input file; each word has only letters, digits, hyphens, and apostrophes.

4. After unit-testing your `Filter` class, develop a `FilterUser` class similar to the `VeryLongIntUser` class in Section 6.3.3. The `FilterUser` class scans in the path names for the input file and the common-words file. Include appropriate messages and re-prompts for incorrect input.
The filter you will be creating in this project is essential for a search engine because the relevance of a document is based on the words the document contains.
Here is sample input for the `FilterUser` class:

kubla.in1
common.in1

If the file "kubla.in1" consists of

Caverns are <a href=browser.in2>browser2</a> measureless to man.

and the file "common.in1" is as shown above, then the contents of the returned `ArrayList` will be

caverns

browser2

(continued on next page)

(continued from previous page)

measureless

man

Here is more sample input for the `FilterUser` class:

```
kubla.in3
common.in1
```

If the file “kubla.in3” consists of:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred <a href=browser.in4>browser4</a> river, ran
Through caverns <a href=browser.in2>browser2</a> measureless to man
Down to a sunless sea.
Sea’s sea and

down
```

then the contents of the returned `ArrayList` will be

xanadu

kubla

khan

stately

pleasure-dome

decree

alph

sacred

browser4

river

ran

through

caverns

browser2

measureless

man

sunless

sea

sea’s

sea

Hint for removing tags: Treat an entire tag as a delimiter. See http://www.txt2re.com for details.

Linked Lists

CHAPTER 7

In this chapter we continue our study of collection classes by introducing the `LinkedList` class, part of the Java Collections Framework. Like the `ArrayList` class, the `LinkedList` class implements the `List` interface, so you are already familiar with most of the `LinkedList` method headings. There are some significant performance differences between the two classes. For example, `LinkedList` objects lack the random-access feature of `ArrayList` objects: to access a `LinkedList`'s element from an *index* requires a loop. But `LinkedList` objects allow constant-time insertions and deletions, once the insertion-point or deletion-point has been accessed.

We will start with a general discussion of linked lists, and then introduce a simple linked structure, the `SinglyLinkedList` class. This toy class serves mainly to prepare you for the more powerful, and more complicated, `LinkedList` class. The application of the `LinkedList` class, a line editor, takes advantage of a `LinkedList` iterator's ability to insert or remove in constant time.

CHAPTER OBJECTIVES

1. Be able to develop new methods for the `SinglyLinkedList` class.
2. Understand the `LinkedList` class from a user's perspective.
3. Given an application that requires a list, be able to decide whether an `ArrayList` or a `LinkedList` would be more appropriate.
4. Compare several choices of fields for the `LinkedList` class and, for each choice, be able to create a `LinkedList` object.

7.1 What is a Linked List?

Before we start investigating the `LinkedList` class, let's spend a little time on the general concept of a linked list. A ***linked list*** is a `List` object (that is, an object in a class that implements the `List` interface) in which the following property is satisfied:

> Each element is contained in an object, called an `Entry` object, which also includes a reference, called a ***link***, to the `Entry` object that contains the next element in the list.

For the `Entry` object that holds the last element, there is no "next" element.

For example, Figure 7.1 shows part of a linked list.

Some linked lists also satisfy the following property:

> Each `Entry` object includes a link to the `Entry` object that contains the previous element in the list.

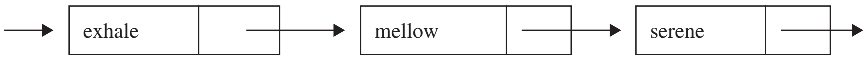


FIGURE 7.1 Part of a linked list

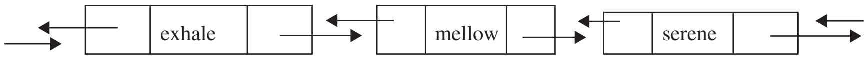


FIGURE 7.2 Part of a doubly-linked list

A linked list that satisfies the second property is called a ***doubly-linked list***. Otherwise, it is called a ***singly-linked list***. For example, Figure 7.2 shows part of a doubly-linked list with three elements, and they happen to be in alphabetical order.

We have intentionally omitted any indication of how the first and last elements are identified, and what is stored in their previous and next links, respectively. In Section 7.3, we'll see that there are several options.

Most of this chapter is devoted to doubly-linked lists, but we will start by studying singly-linked lists because, as you might imagine, they are easier to develop (they are also less powerful).

7.2 The `SinglyLinkedList` Class—A Singly-Linked, Toy Class!

We now create a class, `SinglyLinkedList`, that implements the `List` interface of the Java Collections Framework. As suggested by Figure 7.1, the basic idea is to link the elements together in a chain: with each element we will include a reference to the next element in the collection. You will have the opportunity to expand on this `SinglyLinkedList` class in Lab 12 and in three of the programming projects at the end of this chapter.

The `SinglyLinkedList` class has very little functionality, and is not part of the Java Collections Framework. You will never use it for application programs. Why bother to learn it in the first place? You should view the `SinglyLinkedList` class as a "toy" class that highlights the concepts of links and iterators, two essential features of the Java Collections Framework. And, like any other toy, you will have the opportunity to play with the `SinglyLinkedList` class: to add new fields and methods, and to alter the definitions of existing methods. You will study the `SinglyLinkedList` class mainly to make it easier for you to understand the `LinkedList` class, which *is* in the Java Collections Framework. The `LinkedList` class, doubly-linked, is quite powerful but also somewhat complex.

The elements in a `SinglyLinkedList` object are not stored contiguously, so with each element we must provide information on how to get to the next element in the collection. First, we create a class to hold a reference to an element and a "next" reference. In this `Entry` class, there are no specified methods (of course, there is a default constructor) and two fields, with `E` the type parameter:

```
protected class Entry<E>
{
      protected E element;
      protected Entry<E> next;
} // class Entry
```

The `next` field in an `Entry` holds a reference to another `Entry` object. A reference to an `Entry` object is called a ***link***. For example, Figure 7.3 depicts a sequence of linked entries; each element is a (reference to a) `String` object. We use an arrow to indicate that the `next` field at the base of the arrow contains

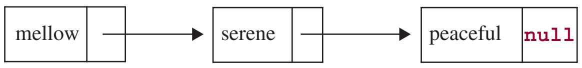


FIGURE 7.3 Part of a singly-linked list of three String elements

a reference to the Entry object pointed to by the tip of the arrow. And, for the sake of simplicity, we pretend that the type of element is String rather than reference-to-String. In the last Entry object, the next field has the value **null**, which indicates that there is no subsequent Entry object.

The Entry class will be embedded in the SinglyLinkedList class. A class that is embedded in another class is called a ***nested class***. This embedding allows the SinglyLinkedList class to access the two fields in an Entry object directly (a good thing, too, because the Entry class has no methods). The Entry class has **protected** visibility for the sake of future subclasses of SinglyLinkedList.

The SinglyLinkedList class will implement the Collection interface in the Java Collections Framework. As with all other Collection classes, the SinglyLinkedList class is parameterized, with E as the type parameter:

```
public class SinglyLinkedList<E> extends AbstractCollection<E>
                                 implements List<E>
```

We need not provide realistic implementations for each of the abstract methods in the List interface. For methods we are not interested in, their definitions will simply throw an exception. To start with, we will implement only five methods: a default constructor, isEmpty, addToFront, size, and contains. Here are the method specifications:

1. **Default constructor**

```
/**
 * Initializes this SinglyLinkedList object to be empty, with elements to be of
 *   type E.
 *
 */
public SinglyLinkedList()
```

Note. Saying that a SinglyLinkedList object is empty means that the collection has no elements in it.

2. **The isEmpty method**

```
/**
 *  Determines if this SinglyLinkedList object has no elements.
 *
 *  @return true - if this SinglyLinkedList object has no elements; otherwise,
 *                 false.
 *
 */
public boolean isEmpty ()
```

Example If we start with

```
SinglyLinkedList<Double> myLinked = new SinglyLinkedList<Double>();

System.out.println (myLinked.isEmpty ());
```

The output would be

```
true
```

because the object referenced by `myLinked` has no elements.

3. **The `addToFront` method**

```
/**
 *  Adds a specified element to the front of this SinglyLinkedList object.
 *
 *  @param element – the element to be inserted (at the front).
 *
 */
public void addToFront (E element)
```

Note 1. Elements are inserted only ***at the front*** of a `SinglyLinkedList` object (This allows for a simpler implementation). For example, suppose the `SinglyLinkedList` object referenced by `myLinked` consists of "yes", "no", and "maybe" in that order, and the message is

```
myLinked.addToFront ("simple");
```

Then the `SinglyLinkedList` object referenced by `myLinked` will consist of "simple", "yes", "no", and "maybe" in that order.

Note 2. The method identifier `addToFront` is used instead of `add` because the `add (E element)` in the `List` interface specifies that `element` must be inserted at the end of the list, and that is somewhat more difficult than inserting at the front.

4. **The `size` method**

```
/**
 *  Determines the number of elements in this SinglyLinkedList object.
 *  The worstTime(n) is O(n).
 *
 *  @return the number of elements.
 *
 */
public int size ()
```

Example Suppose the `SinglyLinkedList` object referenced by `myLinked` consists of the elements "simple", "yes", "no", and "maybe" in that order. If the message is

```
System.out.println (myLinked.size ());
```

then the output will be

```
4
```

5. **The `contains` method**

```
/**
 *  Determines if this SinglyLinkedList object contains a specified element.
 *  The worstTime(n) is O(n).
 *
 *  @param obj – the specified element being sought.
 *
 *  @return true - if this SinglyLinkedList object contains obj; otherwise,
 *          false.
 *
 */
public boolean contains (Object obj)
```

Note. The user of this method is responsible for ensuring that the `equals` method is explicitly defined for the class that includes `obj` and the elements in the `SinglyLinkedList`. Otherwise, as noted in Section 2.7, the `Object` class's version of `equals` will be applied:

```
public boolean equals (Object obj)
{
     return (this == obj);
}
```

This methods test whether the reference to the calling object contains the same address as the reference `obj`. Because equality-of-references is tested instead of equality-of-elements, **`false`** will be returned if the calling-object reference and `obj` are references to distinct but identical objects!

Here, from the book's website, is a test suite for these methods:

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;

import java.util.*;

public class SinglyLinkedTest
{
    public static void main(String[ ] args)
    {
       Result result = runClasses (SinglyLinkedTest.class);
       System.out.println ("Tests run = " + result.getRunCount() +
                           "\nTests failed = " + result.getFailures());
    } // method main

    protected SinglyLinkedList<String> list;

    @Before
    public void runBeforeEachTest()
    {
       list = new SinglyLinkedList<String>();
    } // method runBeforeEachTest

    @Test
    public void testSize1()
    {
       assertEquals (0, list.size());
    } // method testSize1
    @Test
    public void testAdd()
    {
        list.addToFront ("Greg");
        list.addToFront ("Brian");
        list.addToFront ("Berkin");
        assertEquals ("[Berkin, Brian, Greg]", list.toString());
                     // Note: AbstractCollection implements toString()
    } // testAdd

    @Test
    public void testSize2()
    {
        list.addToFront ("Greg");
        list.addToFront ("Brian");
        list.addToFront ("Berkin");
        assertEquals (3, list.size());
    } // testSize2

    @Test
    public void testContains1()
    {
        list.addToFront ("Greg");
        list.addToFront ("Brian");
        list.addToFront ("Berkin");
        assertEquals (true, list.contains("Brian"));
    } // testContains1

   @Test
    public void testContains2()
    {
        list.addToFront ("Greg");
        list.addToFront ("Brian");
        list.addToFront ("Berkin");
        assertEquals (false, list.contains("Jack"));
    } // testContains2

   @Test
    public void testContains3()
    {
        list.addToFront ("Greg");
        list.addToFront ("Brian");
        list.addToFront ("Berkin");
        assertEquals (false, list.contains(7));
    } // testContains2

} // class SinglyLinkedTest
```

All tests failed initially, with the usual stub (`throw new UnsupportedOperationException();`) for each `SinglyLinkedList` method.

These few methods do not provide much in the way of functionality: we cannot remove an element and, what's worse, we cannot even retrieve the elements. But we have enough to consider fields and method definitions.

7.2.1 Fields and Method Definitions in the SinglyLinkedList Class

Something is missing from Figure 7.3: a reference to the first Entry object. This missing "link" will be a field in the SinglyLinkedList class, in fact, the only field:

```
protected Entry<E> head;
```

Suppose a SinglyLinkedList object is constructed as follows:

```
SinglyLinkedList<Integer> scoreList = new SinglyLinkedList<Integer>();
```

To make scoreList a reference to an empty SinglyLinkedList object, all the default constructor has to do is to initialize the head field to **null**. Since a reference field is automatically initialized to **null**, we need not define the default constructor, but we will do so for the sake of being explicit:

```
public SinglyLinkedList()
{
       head = null;
} // default constructor
```

Now we can move on to the definitions of the isEmpty, addToFront, size, and contains methods. How can the isEmpty() method determine if the list has no elements? By testing the head field:

```
public boolean isEmpty ()
{
       return head == null;
} // method isEmpty
```

The definition of the addToFront (E element) method is not quite so easy to develop. For inspiration, suppose we add a fourth element to the front of a singly-linked list consisting of the three elements from Figure 7.3. Figure 7.4 shows the picture before the fourth element is added.

According to the method specification for the addToFront method, each new element is inserted at the front of a SinglyLinkedList object. So if we now add "calm" to *the front of* this list, we will get the list shown in Figure 7.5.

In general, how should we proceed if we want to insert the element at the front of the calling SinglyLinkedList object? We start by constructing a new Entry object and assigning (a reference to) the new element to that Entry object's element field. What about the Entry object's next field? The reference we assign to the next field should be a reference to what had been the first Entry before this

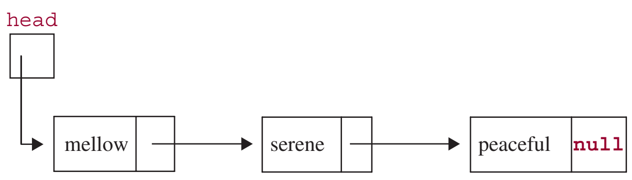


FIGURE 7.4 A SinglyLinkedList object of three String elements

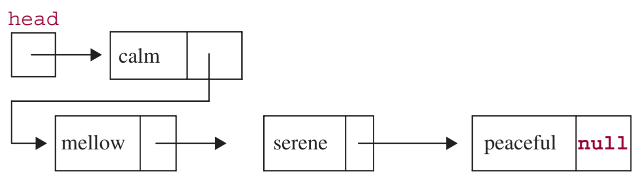


FIGURE 7.5 The SinglyLinkedList object from Figure 2.5 after inserting "calm" at the front of the list

call to addToFront. In other words, we should assign head to the next field of the new Entry object. Finally, we adjust head to reference the new Entry object. The complete definition is:

```
public void addToFront (E element)
{
    Entry<E> newEntry = new Entry<E>();
    newEntry.element = element;
    newEntry.next = head;
    head = newEntry;
} // method addToFront
```

Figures 7.6a through 7.6d show the effect of executing the first four statements in this method when "calm" is inserted at the front of the SinglyLinkedList object shown in Figure 7.4.

For the definition of the size method, we initialize a local int variable, count, to 0 and a local Entry reference, current, to head. We then loop until current is null, and increment count and current during each loop iteration. Incrementing count is a familiar operation, but what does it mean to

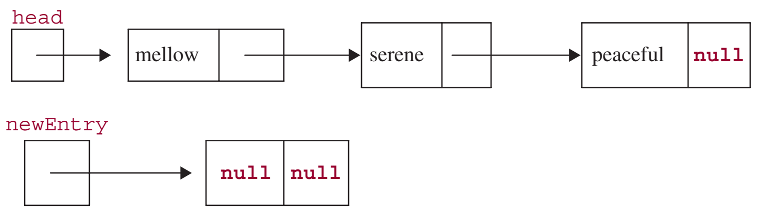


FIGURE 7.6a The first step in inserting "calm" at the front of the SinglyLinkedList object of Figure 7.4: constructing a new Entry object (whose two fields are automatically pre-initialized to null)

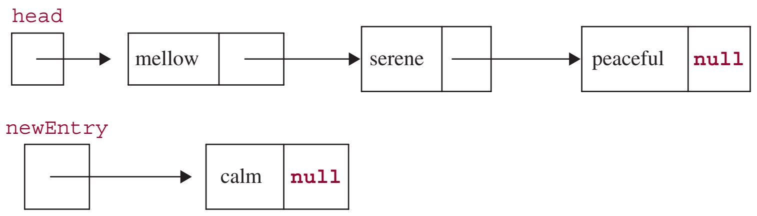


FIGURE 7.6b The second step in inserting "calm" at the front of the SinglyLinkedList object of Figure 7.4: assigning the object-reference element to the element field of the newEntry object

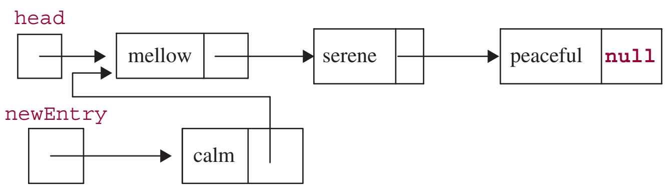


FIGURE 7.6c The third step in inserting “calm” at the front of the SinglyLinkedList object of Figure 7.4: assigning head to the next field of the newEntry object

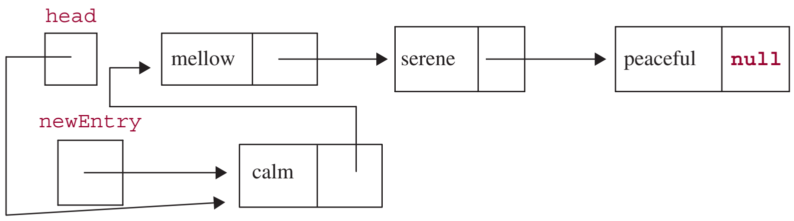


FIGURE 7.6d The fourth step in inserting “calm” at the front of the SinglyLinkedList object of Figure 7.4. The SinglyLinkedList object is now as shown in Figure 7.5

“increment current ”? That means to change current so that current will reference the next Entry after the one current is now referencing. That is, we set

```
current = current.next;
```

Here is the definition of the size method:

```
public int size ()
{
    int count = 0;

    for (Entry<E> current = head; current != null; current = current.next)
           count++;
    return count;
} // method size
```

The loop goes through the entire SinglyLinkedList object, and so worstTime(n) is linear in n (as is averageTime(n)). Note that if we add a size field to the SinglyLinkedList class, the definition of the size() method becomes a one-liner, namely,

```
return size;
```

But then the definition of the addToFront method would have to be modified to maintain the value of the size field. See Programming Exercise 7.7.

Finally, for now, we develop the contains method. The loop structure is similar to the one in the definition of the size method, except that we need to compare element to current.element. For the sake of compatibility with the LinkedList class in the Java Collections Framework, we allow **null** elements in a `SinglyLinkedList` object. So we need a separate loop for the case where `element` is **`null`**. Here is the code:

```
public boolean contains (Object obj)
{
     if (obj == null)
     {
          for (Entry<E> current = head; current != null; current = current.next)
                 if (obj == current.element)
                         return true;
     } // if obj == null
     else
          for (Entry<E> current = head; current != null; current = current.next)
                 if (obj.equals (current.element))
                         return true;
     return false;
} // method contains
```

With these definitions, all tests passed in `SinglyLinkedTest`.

As we discussed in Section 7.2, in the note following the method specification for `contains (Object obj)`, make sure that the definition of the `equals` method in the element's class compares elements for equality. We needed a special case for `obj == null` because the message `obj.equals (current.element)` will throw `NullPointerException` if `obj` is **`null`**.

One important point of the `SinglyLinkedList` class is that a linked structure for storing a collection of elements is different from an array or `ArrayList` object in two key respects:

1. The size of the collection need not be known in advance. We simply add elements at will. So we do not have to worry, as we would with an array, about allocating too much space or too little space. But it should be noted that in each `Entry` object, the `next` field consumes extra space: it contains program information rather than problem information.
2. Random access is not available. To access some element, we would have to start by accessing the `head` element, and then accessing the next element after the `head` element, and so on.

7.2.2 Iterating through a `SinglyLinkedList` Object

We have not yet established a way for a user to loop through the elements in a `SinglyLinkedList` object. The solution, as we saw in Section 4.2.3.1, is to develop an `Iterator` class for `SinglyLinkedList` objects, that is, a class that implements the `Iterator` interface, and of which each instance will iterate over a `SinglyLinkedList` object.

The `SinglyLinkedListIterator` class will be a **`protected`** class embedded within the `SinglyLinkedList` class. We want our `Iterator` object to be positioned at an `Entry` so we can easily get the next element and determine if there are any more elements beyond where the `Iterator` object is positioned. For now[1], the `SinglyLinkedListIterator` class will have only one field:

```
protected Entry<E> next;
```

[1]In Project 7.1, two additional fields are added to the `SinglyLinkedListIterator` class.

We will fully implement only three methods: a default constructor, next() and hasNext(). The remove() method will simply throw an exception. Here is an outline of the class:

```
protected class SinglyLinkedListIterator implements Iterator<E>
{
   protected Entry<E> next;

   /**
    *  Initializes this SinglyLinkedListIterator object
    *
    */
   protected SinglyLinkedListIterator()
   {
        ...
   } // default constructor


   /**
    *  Determines if this Iterator object is positioned at an element in this
    *  SinglyLinkedIterator object.
    *
    *  @return true - if this Iterator object is positioned at an element;
    *          otherwise, false.
    */
   public boolean hasNext()
   {
        ...
   } // method hasNext


   /**
    *  Returns the element this Iterator object was (before this call)
    *  positioned at, and advances this Iterator object.
    *
    *  @return - the element this Iterator object was positioned at.
    *
    *  @throws NullPointerException – if this Iterator object was
    *          not postioned at an element before this call.
    */
   public E next()
   {
        ...
   } // method next


   public void remove()
   {
        throw new Unsupported1OperationException();
   } // method remove
} // class SinglyLinkedListIterator
```

Note that the SinglyLinkedIterator class has E as its type parameter because SinglyLinkedList Iterator implements Iterator<E>.

In defining the three methods, there are three "next"s we will be dealing with:

a next field in the SinglyLinkedListIterator class;

a next() method in the SinglyLinkedListIterator class;

a next field in the Entry class.

You will be able to determine the correct choice based on the context—and the presence or absence of parentheses.

An interface does not have any constructors because an interface cannot be instantiated, so the Iterator interface had no constructors. But we will need a constructor for the SinglyLinkedList Iterator class. Otherwise, the compiler would generate a default constructor and the Java Virtual Machine would simply, but worthlessly, initialize the next field to **null**. What should the constructor initialize the next field to? Where we want to start iterating? At the head of the SinglyLinkedList:

```
protected SinglyLinkedListIterator()
{
       next = head;
} // default constructor
```

This method can access head because the SinglyLinkedListIterator class is embedded in the SinglyLinkedList class, where head is a field.

The hasNext() method should return true as long as the next field (in the SinglyLinkedList Iterator class, not in the Entry class) is referencing an Entry object:

```
public boolean hasNext ()
{
       return next != null;
} // method hasNext
```

The definition of the remove() method will simply throw an exception, so all that remains is the definition of the next() method. Suppose we have a SinglyLinkedList of two String elements and the default constructor for the SinglyLinkedListIterator has just been called. Figure 7.7 shows the current situation (as usual, we pretend that a String object itself, not a reference, is stored in the element field of an Entry object):

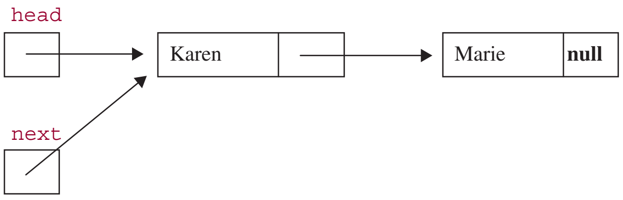


FIGURE 7.7 The contents of the next field in the SinglyLinkedListIterator class just after the SinglyLinkedListIterator's constructor is called

Since we are just starting out, what element should the next() method return? The element returned should be “Karen”, that is, next.element. And then the next field should be advanced to point to the next Entry object That is, the SinglyLinkedListIterator’s next field should get the reference stored in the next field of the Entry object that the SinglyLinkedListIterator’s next field is currently pointing to. We can’t do anything after a return, so we save next.element before advancing next, and then we return (a reference to) the saved element. Here is the definition:

```
public E next()
{
       E theElement = next.element;
       next = next.next;
       return theElement;
} // method next
```

Now that we have a SinglyLinkedListIterator class, we can work on the problem of iterating through a SinglyLinkedList object. First, we have to associate a SinglyLinkedListIterator object with a SinglyLinkedList object. The iterator() method in the SinglyLinkedList class creates the necessary connection:

```
/**
 *  Returns a SinglyLinkedListIterator object to iterate over this
 *  SinglyLinkedList object.
 *
 */
public Iterator<E> iterator()
{
       return new SinglyLinkedListIterator();
} // method iterator
```

The value returned is a (reference to a) SinglyLinkedListIterator. The specified return type has to be Iterator<E> because that is what the iterator() method in the Iterator interface calls for. Any class that implements the Iterator interface—such as SinglyLinkedListIterator—can be the actual return type.

With the help of this method, a user can create the appropriate iterator. For example, if myLinked is a SinglyLinkedList object of Boolean elements, we can do the following:

```
Iterator<Boolean> itr = myLinked.iterator();
```

The variable itr is a polymorphic reference: it can be assigned a reference to an object in any class (for example, SinglyLinkedListIterator) that implements the Iterator<Boolean> interface. And myLinked.iterator() returns a reference to an object in the SinglyLinkedListIterator class, specifically, to an object that is positioned at the beginning of the myLinked object.

The actual iteration is straightforward. For example, to print out each element:

```
while (itr.hasNext ())
       System.out.println (itr.next ());
```

Or, even simpler, with the enhanced **for** statement:

```
for (Boolean b : myList)
     System.out.println (b);
```

Now that we have added a new method to the SinglyLinkedList class, we need to test this method, and that entails testing the next() method in the SinglyLinkedListIterator class. The book's website includes these tests. The other methods in the SinglyLinkedClass were re-tested, and passed all tests.

For a complete example of iteration, the following program method reads in a non-empty list of grade-point-averages from the input, stores each one at the front of a SinglyLinkedList object, and then iterates through the collection to calculate the sum. Finally, the average grade-point-average is printed.

```
import java.util.*;

public class SinglyLinkedExample
{
    public static void main (String [ ] args)
    {
        new SinglyLinkedExample().run();
    } // method main

    public void run()
    {
        final double SENTINEL = -1.0;

        final String INPUT_PROMPT = "\nPlease enter a GPA (or " +
              SENTINEL + " to quit): ";

        final String AVERAGE_MESSAGE = "\n\nThe average GPA is ";

        final String NO_VALID_INPUT =
              "\n\nError: there were no valid grade-point-averages in the input.";

        SinglyLinkedList<Double> gpaList = new SinglyLinkedList<Double>();

        Scanner sc = new Scanner (System.in);

        double oneGPA,
               sum = 0.0;

        while (true)
        {
               try
               {
                    System.out.print (INPUT_PROMPT);
                    oneGPA = sc.nextDouble();
                    if (oneGPA == SENTINEL)
                        break;
                    gpaList.addToFront (oneGPA);
               } // try
               catch (Exception e)
               {
                    System.out.println (e);
                    sc.nextLine();
               } // catch
```

```
            } // while
            for (Double gpa : gpaList)
                   sum += gpa;
            if (gpaList.size() > 0)
                   System.out.println(AVERAGE_MESSAGE + (sum/gpaList.size ()));
            else
                   System.out.println (NO_VALID_INPUT);
         }  // method run

    } // class SinglyLinkedExample
```

The `SinglyLinkedList` class, including its embedded `Entry` and `SinglyLinkedListIterator` classes, is available from the course website, as is `SinglyLinkedTest` and `SinglyLinkedExample`.

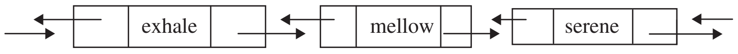
In Lab 12, you have the opportunity to define several other methods in the `SinglyLinkedList` class.

You are now prepared to do Lab 12:
Expanding the `SinglyLinkedList` Class

Also, there are several Programming Exercises and a Programming Project related to the `SinglyLinkedList` class. But now that you have some familiarity with links, we turn to the focal point of this chapter: doubly-linked lists.

7.3 Doubly-Linked Lists

Suppose we want to insert "placid" in front of "serene" in the doubly-linked list partially shown in Figure 7.2 and repeated here:



First we need to get a reference to the `Entry` object that holds "serene"; that will take linear-in-*n* time, on average, where *n* is the size of the linked list. After that, as shown in Figure 7.8, the insertion entails constructing a new `Entry` object, storing "placid" as its element, and adjusting four links (the previous and next links for "placid", the next link for the predecessor of "serene", and the previous link for "serene"). In other words, once we have a reference to an `Entry` object, we can insert a new element in front of that `Entry` object in constant time.

The process for removal of an element in a doubly-linked list is similar. For example, suppose we want to remove "mellow" from the partially shown linked list in Figure 7.8. First, we get a reference to the `Entry` object that houses "mellow", and this takes linear-in-*n* time, on average. Then, as shown in Figure 7.9, we adjust the `next` link of the predecessor of "mellow" and the `previous` link of the successor of "mellow". Notice that there is no need to adjust either of the links in the `Entry` object that houses "mellow" because that object is not pointed to by any other `Entry` object's links.

The bottom line in the previous discussion is that it takes linear-in-*n* time to get a reference to an `Entry` object that houses an element, but once the reference is available, any number of insertions, removals or retrievals can be accomplished in constant time for each one.

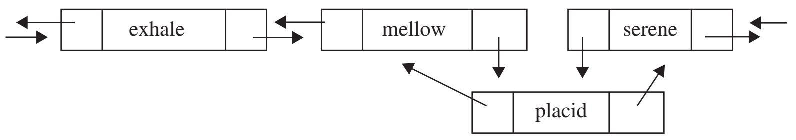


FIGURE 7.8 The partially shown doubly-linked list from Figure 7.2 after "placid" is inserted in front of "serene"

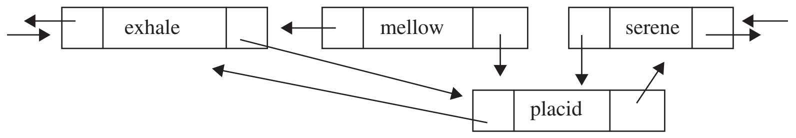


FIGURE 7.9 The partially shown linked list from Figure 7.8 after "mellow" removed

Now that you have a rough idea of what a doubly-linked list looks like and can be manipulated, we are ready to study the `LinkedList` class, the Java Collection Framework's design and implementation of doubly-linked lists. First as always, we start with a user's perspective, that is, with the method specifications.

7.3.1 A User's View of the `LinkedList` Class

Because the `LinkedList` class implements the `List` interface, `LinkedList` objects support a variety of index-based methods such as `get` and `indexOf`. The indexes *always start at 0,* so a `LinkedList` object with three elements has its first element at index 0, its second element at index 1, and its third element at index 2.

As we did with the `ArrayList` class, let's start with the big picture from the user's point of view. Figure 7.10 has the method headings for the public methods in the `LinkedList` class. The `LinkedList` class is a parameterized type, with `E` as the type parameter representing the type of an element. Method specifications can be obtained from the Application Programmer Interface (API).

Section 7.3.2 has more details (still from the user's view): those `LinkedList` methods that are in some way different from those of the `ArrayList` class.

7.3.2 The `LinkedList` Class versus the `ArrayList` Class

Let's compare the `LinkedList` and `ArrayList` classes from a user's perspective. The `LinkedList` class does not have a constructor with an initial-capacity parameter because `LinkedList` objects grow and shrink as needed. And the related methods `ensureCapacity` and `trimToSize` are also not need for the `LinkedList` class.

The `LinkedList` class has several methods, such as `removeFirst()` and `getLast()`, that are not in the `ArrayList` class. For each of these methods, worstTime(n) is constant, where n represents the number of elements in the calling object. These methods are provided only for the convenience of users. They do not represent an increase in the functionality of `LinkedList` objects. For example, if `myList` is a non-empty `LinkedList` object, the message

```
myList.removeFirst()
```

can be replaced with

```
myList.remove (0)
```

```
public LinkedList()
public LinkedList (Collection<? extends E> c )
public boolean add (E element) // element inserted at back; worstTime(n) is constant
public void add (int index, E element)
public void addAll (Collection<? extends E> c)
public boolean addAll (int index, Collection<? extends E> c)
public boolean addFirst (E element)
public boolean addLast (E element)
public void clear( ) // worstTime(n) is O(n)
public Object clone()
public boolean contains (Object obj)
public boolean containsAll (Collection<?> c)
public E element()
public boolean equals (Object obj)
public E get (int index) // worstTime(n) is O(n)
public E getFirst ()
public E getLast ()
public int hashCode()
public int indexOf (Object obj)
public boolean isEmpty()
public Iterator<E> iterator()
public int lastIndexOf (Object obj)
public ListIterator<E> listIterator() // iterate backward or forward
public ListIterator<E> listIterator (final int index)
public boolean offer (E element)
public E peek()
public E poll()
public E pop()
public void push (E e)
public E remove()
public boolean remove (Object obj)
public E remove (int index)
public boolean removeAll (Collection<?> c)
public E removeFirst() // worstTime(n) is constant
public E removeLast() // worstTime(n) is constant
public boolean retainAll (Collection<?> c)
public E set (int index, E element) // worstTime(n) is O(n)
public int size( )
public List<E> subList (int fromIndex, int toIndex)
public Object[ ] toArray( )
public T[ ] toArray (T[ ] a)
public String toString()
```

FIGURE 7.10 Method headings for the public methods in the `LinkedList` class. Except for the constructors, the headings are in alphabetical order by method identifier

And this last message takes only constant time. But if `myList` were a non-empty `ArrayList` object, the same message,

```
myList.remove (0)
```

requires linear-in-*n* time.

There are some other performance differences between `LinkedList` objects and `ArrayList` objects. Here are method specifications of some other `LinkedList` methods that have different worst times than their `ArrayList` counterparts.

1. **The one-parameter `add` method**

```
/**
 *  Appends a specified element to (the back of) this LinkedList object.
 *
 *  @param element - the element to be appended.
 *
 *  @return true - according to the general contract of the Collection interface's
 *                 one-parameter add method.
 *
 */
public boolean add (E element)
```

Note. The worstTime(n) is constant. With an `ArrayList` object, the worstTime(n) is linear in n for the one-parameter `add` method, namely, when the underlying array is at full capacity. This represents a significant difference for a single insertion. For multiple back-end insertions, the time estimates for the `ArrayList` and `LinkedList` classes are similar. Specifically, for n back-end insertions, worstTime(n) is linear in n for both the `ArrayList` class and the `LinkedList` class. And averageTime(n), for a single call to `add`, is constant for both classes.

Example Suppose we have the following:

```
LinkedList<String> fruits = new LinkedList<String>();
fruits.add ("apples");
fruits.add ("kumquats");
fruits.add ("durian");
fruits.add ("limes");
```

The `LinkedList` object `fruits` now contains, in order, ‘‘apples’’, ‘‘kumquats’’, ‘‘durian’’, and ‘‘limes’’.

2. **The `get` method**

```
/**
 *  Finds the element at a specified position in this LinkedList object.
 *  The worstTime(n) is O(n).
 *
 *  @param index - the position of the element to be returned.
 *
 *  @return the element at position index.
 *
 *  @throws IndexOutOfBoundsException - if index is less than 0 or greater than
 *          or equal to size().
 */
public E get (int index)
```

Note. This method represents a major disadvantage of the `LinkedList` class compared to the `ArrayList` class. As noted in the method specification, the `LinkedList` version of this method has worstTime(n) in O(n)—in fact, worstTime(n) is linear in n in the current implementation. But for the `ArrayList` version, worstTime(*n*) is constant. So if your application has a preponderance of list accesses, an `ArrayList` object is preferable to a `LinkedList` object.

Example Suppose the `LinkedList` object `fruits` consists of “apples”, “kumquats”, “durian”, and “limes”, in that order. Then the message

```
fruits.get (1)
```

would return “kumquats”; recall that list indexes start at zero.

3. **The `set` method**

```
/**
 *  Replaces the element at a specified index with a specified element.
 *  The worstTime(n) is O(n).
 *
 *  @param index - the specified index where the replacement will occur.
 *  @param element - the element that replaces the previous occupant at
 *         position index.
 *
 *  @return the previous occupant (the element replaced) at position index.
 *
 *   @throws IndexOutOfBoundsException - if index is either less than 0 or
 *          greater than or equal to size().
 *
 */
public E set (int index, E element)
```

Note. For the `ArrayList` version of this method, worstTime(*n*) is constant.

Example Suppose the `LinkedList` object `fruits` consists of “apples”, “kumquats”, “durian”, and “limes”, in that order. We can change (and print) the element at index 2 with the following:

```
System.out.println (fruits.set (2, "kiwi"));
```

The elements in the `LinkedList` object `fruits` are now “apples”, “kumquats”, “kiwi”, and “limes”, and the output will be “durian”.

When we looked at the `ArrayList` class, iterators were ignored. That neglect was due to the fact that the random-access property of `ArrayList` objects allowed us to loop through an `ArrayList` object in linear time by using indexes. `LinkedList` objects do not support random access (in constant time), so iterators are an essential component of the `LinkedList` class.

7.3.3 `LinkedList` Iterators

In the `LinkedList` class, the iterators are bi-directional: they can move either forward (to the next element) or backward (to the previous element). The name of the class that defines the iterators is `ListItr`. The `ListItr` class—which implements the `ListIterator` interface—is embedded as a **`private`** class in the `LinkedList` class. So a `ListItr` object cannot be directly constructed by a user; instead there are `LinkedList` methods to create a `ListItr` object, just as there was a `SinglyLinkedList` method (namely, `iterator()`), to create a `SinglyLinkedListIterator` object.

There are two `LinkedList` methods that return a (reference to a) `ListIterator` object, that is, an object in a class that implements the `ListIterator` interface. Their method specifications are as follows:

1. **The start-at-the-beginning `listIterator` method**

```
/**
 *  Returns a ListIterator object positioned at the beginning of this LinkedList
 *  object.
 *
 *  @return a ListIterator object positioned at the beginning of this LinkedList
 *          object.
 *
 */
public ListIterator<E> listIterator()
```

Example Suppose that `fruits` is a `LinkedList` object. Then we can create a `ListItr` object to iterate through `fruits` as follows:

```
ListIterator<String> itr1 = fruits.listIterator();
```

2. **The start anywhere `listIterator` method**

```
/**
 *  Returns a ListIterator object positioned at a specified index in this LinkedList
 *  object.  The worstTime(n) is O(n).
 *
 *  @param index – the specified index where the returned iterator is positioned.
 *
 *  @return a ListIterator object positioned at index.
 *
 *  @throws IndexOutOfBoundsException – if index is either less than zero or
 *          greater than size().
 *
 */
public ListIterator<E> listIterator (final int index)
```

Example Suppose the `LinkedList` object `fruits` consists of ‘‘apples’’, ‘‘kumquats’’, ‘‘durian’’, and ‘‘limes’’, in that order. The following statement creates a `ListIterator` object positioned at ‘‘durian’’:

```
ListIterator<String> itr2 = fruits.listIterator (2);
```

Figure 7.11 has the method headings for all of the methods in the `ListItr` class. We will look at some of the details—from a user's viewpoint—of these methods shortly.

We can iterate forwardly with an enhanced **`for`** statement (or the pair `hasNext()` and `next()`), just as we did with the `SinglyLinkedList` class in Section 7.2.2. For example, suppose the `LinkedList` object `fruits` consists of “kumquats”, “bananas”, “kiwi”, and “apples”, in that order. We can iterate through `fruits` from the first element to the last element as follows:

```
for (String s : fruits)
       System.out.println (s);
```

```
public void add (E element)
public boolean hasNext()
public boolean hasPrevious()
public E next()
public int nextIndex()
public E previous()
public int previousIndex()
public void remove( )
public void set (E element)
```

FIGURE 7.11 Method headings for all of the **public** methods in the ListItr class. *For each method, worstTime(n) is constant*!

The output will be:

kumquats

bananas

kiwi

apples

For backward iterating, there is a `hasPrevious()` and `previous()` pair. Here are their method specifications:

IT1. The hasPrevious method

```
/**
 *  Determines whether this ListIterator object has more elements when traversing
 *  in the reverse direction.
 *
 *  @return true - if this ListIterator object has more elements when traversing
 *                 in the reverse direction; otherwise, false.
 *
 */
public boolean hasPrevious()
```

Example Suppose the `LinkedList` object `fruits` consists of the elements ‘‘kumquats’’, ‘‘bananas’’, ‘‘kiwi’’, and ‘‘apples’’, in that order. The output from

```
ListIterator<String> itr = listIterator (2);
System.out.println (itr.hasPrevious());
```

will be

```
true
```

But the output from

```
ListIterator<String> itr = listIterator();  // itr is positioned at index 0
System.out.println (itr.hasPrevious());
```

will be

```
false
```

IT2. The `previous` method

```
/**
 *  Retreats this ListIterator object to the previous element, and returns that
 *  element.
 *
 *  @return the element retreated to in this ListIterator object.
 *
 *  @throws NoSuchElementException - if this ListIterator object has no
 *                previous element.
 *
 */
public E previous()
```

Example Suppose the `LinkedList` object `fruits` consists of ''kumquats'', ''bananas'', ''kiwi'', and ''apples'', in that order. If we have

```
ListIterator<String> itr = fruits.listIterator();
System.out.println (itr.next() + " " + itr.next() + " " + itr.previous());
```

the output will be

```
kumquats bananas bananas
```

Think of the "current" position in a `LinkedList` object as the index where the `ListIterator` is positioned. Here is how the `next()` and `previous()` methods are related to the current position:

- The `next()` method *advances* to the next position in the `LinkedList` object, but returns the element that had been at the current position before the call to `next()`.
- The `previous()` method *first retreats to the position before the current position, and then returns* the element at that retreated-to position.

The `next()` method is similar to the post-increment operator `++`, and the `previous()` method is similar to the pre-decrement operator `--`. Suppose, for example, we have

```
int j = 4,
    k = 9;

System.out.println (j++);
System.out.println (--k);
```

The output will be

```
4
8
```

Because the `previous()` method returns the previous element, we must start "beyond" the end of a `LinkedList` object to iterate in reverse order. For example, suppose the `LinkedList` object `fruits` consists of "kumquats", "bananas", "kiwi", and "apples", in that order, and we have

```
ListIterator itr = fruits.listIterator (fruits.size());  // fruits has size() - 1 elements
while (itr.hasPrevious())
       System.out.println (itr.previous());
```

The output will be:

```
apples
kiwi
bananas
kumquats
```

Of course, the LinkedList object fruits has not changed. It still consists of "kumquats", "bananas", "kiwi", and "apples", in that order.

We can do even more. The ListItr class also has add, remove and set methods. Here are the method specifications and examples (as usual, E is the type parameter representing the class of the elements in the LinkedList object):

IT3. The add method

```
/**
 * Inserts a specified element into the LinkedList object in front of (before) the element
 * that would be returned by next(), if any, and in back of (after) the element that would
 * be returned by previous(), if any.  If the LinkedList object was empty before this
 * call, then the specified element is the only element in the LinkedList object.
 *
 * @param element - the element to be inserted.
 *
 */
public void add (E element)
```

Example Suppose the LinkedList object fruits consists of "kumquats", "bananas", "kiwi", and "apples", in that order. We can insert repeatedly insert "pears" *after* each element in fruits as follows:

```
ListIterator<String> itr = fruits.listIterator();
while (itr.hasNext())
{
       itr.next();
       itr.add ("pears");
} // while
```

During the first iteration of the above **while** loop, the call to next() returns "kumquats" and (before returning) advances to "bananas". The first call to add ("pears") inserts "pears" in front of "bananas". During the second iteration, the call to next() returns "bananas" and advances to "kiwi". The second call to add ("pears") inserts "pears" in front of "kiwi". And so on. At the completion of the **while** statement, the LinkedList object fruits consists of

"kumquats", "pears", "bananas", "pears", "kiwi", "pears", "apples", "pears"

Note. If the ListItr is not positioned at any element (for example, if the LinkedList object is empty), each call to the ListIterator class's add method will insert an element at the end of the LinkedList object.

IT4. The **`remove`** method

```
/**
 *  Removes the element returned by the most recent call to next() or previous().
 *  This method can be called only once per call to next() or previous(), and can
 *  can be called only if this ListIterator's add method has not been called since
 *  the most recent call to next() or previous().
 *
 *  @throws IllegalStateException – if neither next() nor previous() has been
 *          called, or if either this ListIterator's add or remove method has
 *          been called since the most recent call to next() or previous().
 *
 */
public void remove()
```

Example Suppose the `LinkedList` object `fruits` consists of ‘‘kumquats’’, ‘‘pears’’, ‘‘bananas’’, ‘‘pears’’, ‘‘kiwi’’, ‘‘pears’’, ‘‘apples’’, and ‘‘pears’’, in that order. We can remove every other element from `fruits` as follows:

```
ListIterator<String> itr = fruits.listIterator (1);  // NOTE: starting index is 1
while (itr.hasNext())
{
       itr.next();
       itr.remove();
       if (itr.hasNext())
                itr.next();
} // while
```

Now `fruits` consists of ‘‘kumquats’’, ‘‘bananas’’, ‘‘kiwi’’, and ‘‘apples’’, in that order. If we eliminate the **`if`** statement from the above loop, every element except the first element will be removed.

IT5. The **`set`** method

```
/**
 *  Replaces the element returned by the most recent call to next() or previous() with
 *  the specified element.  This call can be made only if neither this ListIterator's add
 *  nor remove method have been called since the most recent call to next() or
 *  previous().
 *
 *  @param element – the element to replace the element returned by the most
 *         recent call to next() or previous().
 *
 *  @throws IllegalStateException – if neither next() nor previous() have been
 *          called, or if either this ListIterator's add or remove method have been
 *          called since the most recent call to next() or previous().
 *
 */
public void set (E element)
```

Example Suppose the `LinkedList` object `fruits` consists of ‘‘kumquats’’, ‘‘bananas’’, ‘‘kiwi’’, and ‘‘apples’’, in that order. We can iterate through `fruits` and capitalize the first letter of each fruit as follows:

```
String aFruit;

char first;

ListIterator<String> itr = fruits.listIterator();

while (itr.hasNext())
{
       aFruit = itr.next();
       first = Character.toUpperCase (aFruit.charAt (0));
       aFruit = first + aFruit.substring (1); // substring from index 1 to end
       itr.set (aFruit);
} // while
```

The `LinkedList` object `fruits` now consists of “Kumquats”, “Bananas”, “Kiwi”, and “Apples”.

Programming Exercise 7.4 considers all possible sequences of calls to the `add`, `next`, and `remove` methods in the `ListItr` class.

As noted in Figure 7.11, all of the `ListItr` methods take only constant time. So if you iterate through a `LinkedList` object, for each call to the `ListItr` object’s `add` or `remove` method, worstTime(n) is constant. With an `ArrayList` object, for each call to `add (int index, E element)` or `remove (int index)`, worstTime(n) is linear in n. And the same linear worst-time would apply for adding and removing if you decided to iterate through an `ArrayList`. The bottom line here is that a `LinkedList` object is faster than an `ArrayList` object when you have a lot of insertions or removals.

What if you need to access or replace elements at different indexes in a list? With an `ArrayList` object, for each call to `get (int index)` or `set (int index, E element)`, worstTime(n) is constant. With a `LinkedList` object, for each call to `get (int index)` or `set (int index, E element)`, worstTime(n) is linear in n. If instead, you iterate through a `LinkedList` object, and use the `ListItr` methods `next()` and `set (E element)` for accessing and replacing elements, each iteration takes linear-in-n time. So if the elements to be accessed or replaced are at indexes that are far apart, an `ArrayList` object will be faster than a `LinkedList` object.

To summarize the above discussion:

✓ If a large part of the application consists of iterating through a list and making insertions and/or removals during the iterations, a `LinkedList` object can be much faster than an `ArrayList` object.

✓ If the application entails a lot of accessing and/or replacing elements at widely varying indexes, an `ArrayList` object will be much faster than a `LinkedList` object.

7.3.4 A Simple Program that uses a `LinkedList` Object

The following program accomplishes the same tasks as the simple `ArrayList` program in Section 6.2.2. But the code has been modified to take advantage of the `LinkedList` class’s ability to perform constant-time insertions or removals during an iteration.

```
import java.util.*;

import java.io.*;

public class LinkedListExample
{
   public static void main (String[ ] args)
   {
      new LinkedListExample().run();
   } // method main

   public void run()
   {
      LinkedList<String> aList = new LinkedList<String>();

      Scanner keyboardScanner = new Scanner (System.in),
              fileScanner;

      String inFilePath,
             word;

      try
      {
             System.out.print ("\n\nPlease enter the path for the input file: ");
             inFilePath = keyboardScanner.nextLine();
             fileScanner = new Scanner (new File (inFilePath));
             while (fileScanner.hasNext())
                    aList.add (fileScanner.next());
             System.out.print ("\nPlease enter the word you want to search for: ");
             word = keyboardScanner.next();
             if (aList.indexOf (word) >= 0)
                 System.out.println (word + " was found.\n\n");
             else
                 System.out.println (word + " was not found.\n\n");

             System.out.print ("Please enter the word you want to remove: ");
             word = keyboardScanner.next();
             int removalCount = 0;
             ListIterator<String> itr = aList.listIterator();
             while (itr.hasNext())
                 if (itr.next().equals (word))
                 {
                    itr.remove();
                    removalCount++;
                 } // if another instance of word has been discovered
             if (removalCount == 0)
                    System.out.println (word +
                         " was not found, so not removed.\n\n");
```

```
                else if (removalCount == 1)
                       System.out.println ("The only instance of " + word +
                            " was removed.\n\n");
                else
                       System.out.println ("All " + removalCount + " instances of " +
                            word + " were removed.\n\n");

                System.out.print ("Please enter the word you want to append: ");
                word = keyboardScanner.next();
                aList.add (word);
                System.out.println (word + " was appended.\n\n");

                System.out.print (
                       "Please enter the word you want to convert to upper case: ");
                word = keyboardScanner.next();

                String currentWord;
                boolean found = false;
                itr = aList.listIterator();
                while (itr.hasNext() && !found)
                {
                       currentWord = itr.next();
                       if (word.equals (currentWord))
                       {
                           itr.set (word.toUpperCase());
                           System.out.println (word +
                              " was converted to upper case.\n\n");
                           found = true;
                       } // found word to convert to upper case
                } // while
                if (!found)
                       System.out.println (word +
                              " was not found, so not upper-cased.\n\n");
                System.out.println ("Here is the final version:\n" + aList);
         } // try
         catch (IOException e)
         {
                System.out.println (e);
         } // catch
      } // method run

  } // class LinkedListExample
```

For removing all instances of a word, the iterator-based version above is clearly faster than repeatedly invoking `aList.remove (word)`. For converting a word to upper case, the iterator-based version above requires only one iteration. The version in Section 6.2.2 requires two iterations: one to get the index, in `aList`, of the word to by upper cased, and one more for the call to `aList.set (index, word.toUpperCase())`.

Lab 13 has an experiment on LinkedList iterators.

You are now prepared to do Lab 13:
Working with LinkedList Iterators

Now that you have seen both the ArrayList and LinkedList classes, you can run a timing experiment on them.

You are now prepared to do Lab 14:
Timing the ArrayList and LinkedList Classes

In Section 7.3.5, we briefly look at a developer's view of the LinkedList class. Specifically, we compare various alternatives for the fields in the LinkedList class. For the choice made in the Java Collections Framework, we develop a LinkedList object, and then, to give you the flavor of that implementation, we investigate the definition of the two-parameter add method.

7.3.5 Fields and Heading of the LinkedList Class

For the implementation of the LinkedList class, the primary decision is what the fields will be. For the sake of code re-use (beneficial laziness), we first consider the SinglyLinkedList class. Can we expand that class to satisfy all of the method specifications for the LinkedList class? The problem comes with the upper bounds of worstTime(n) for some of the LinkedList methods.

For example, the addLast method's postcondition states that any implementation of that method should take constant time. Recall that the SinglyLinkedList class had one field only:

```
protected Entry<E> head;  // reference to first entry
```

The embedded Entry class had two fields, an element and a reference to the next entry:

```
protected E element;

protected Entry<E> next;
```

Clearly, it will take linear-in-n time to add an element to the back of a SinglyLinkedList object. We can get around this difficulty by adding to the SinglyLinkedList class a tail field that holds a reference to last entry in a SinglyLinkedList object. Figure 7.12 shows an example of a SinglyLinkedList object with these fields.

We can now define the addLast method without much difficulty (see Programming Exercise 7.3.a). Implementing the removeLast presents a much more serious problem. We would need to change the (reference stored in the) next field of the Entry object *preceding* the Entry object referenced by tail. And for that task, a loop is needed, so worstTime(n) would be linear in n. That would violate the performance requirement of the removeLast method that specifies worstTime (n) must be constant.

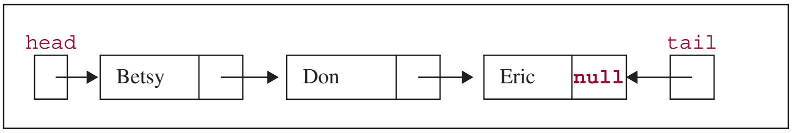


FIGURE 7.12 A singly-linked list with `head` and `tail` fields

So we must abandon a singly-linked implementation of the `LinkedList` class because of the given performance specifications. But the idea mentioned previously—having `head` and `tail` fields—suggests a viable alternative. The nested `Entry` class will support a doubly-linked list by having three fields:

```
protected E element;

protected Entry<E> previous,   // reference to previous entry
                   next;   // reference to next entry
```

Figure 7.13 shows this doubly-linked version of the three-element list from Figure 7.12.

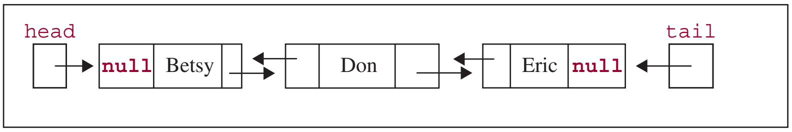


FIGURE 7.13 A doubly-linked list with `head` and `tail` fields

With this version, we can implement the `LinkedList` class with method definitions that satisfy the given performance specifications. You will get to flesh out the details of the doubly-linked, head&tail implementation if you undertake Project 7.4.

The Java Collection Framework's implementation of the `LinkedList` class is doubly-linked, but does not have `head` and `tail` fields. Instead, there is a `header` field, which contains a reference to a special `Entry` object, called a "dummy entry" or "dummy node." We will discuss the significance of the dummy entry shortly. The class starts as follows:

```
public class LinkedList<E> extends AbstractSequentialList<E>,
                           implements List<E>,
                                      Queue<E>,
                                      java.lang.Cloneable,
                                      java.io.Serializable
{
        private transient int size = 0;

        private transient Entry<E> header = new Entry<E> (null, null, null);
```

The `size` field keeps track of the number of elements in the calling `LinkedList` object. As noted in Chapter 6, the **`transient`** modifier merely indicates that this field is not saved if the elements in a `LinkedList` object are serialized, that is, saved to an output stream. (Appendix 1 discusses serialization.)

The nested `Entry` class has three fields, one for an element, and two for links. The only method in the `Entry` class is a constructor that initializes the three fields. Here is the complete `Entry` class

```
private static class Entry<E>
{
      E element;
      Entry<E> next;
      Entry<E> previous;

      Entry(E element, Entry<E> next, Entry<E> previous) {
         this.element = element;
         this.next = next;
         this.previous = previous;
      } // constructor
}  // class Entry<E>
```

The `element` field will hold (a reference to) the `Entry` object's element; `next` will contain a reference to the `Entry` one position further in the `LinkedList` object, and `previous` will contain a reference to the `Entry` one position earlier in the `LinkedList` object.

Under normal circumstances, an object in a nested class has implicit access back to the enclosing object. For example, the nested `ListItr` class accesses the `header` field of the enclosing `LinkedList` object. But if the nested class is declared to be **`static`**, no such access is available. The `Entry` class is a stand-alone class, so it would have been a waste of time and space to provide such access.

We can now make sense of the definition of the `header` field in the `LinkedList` class. That field initially references an `Entry` in which all three fields are **`null`**; see Figure 7.14.

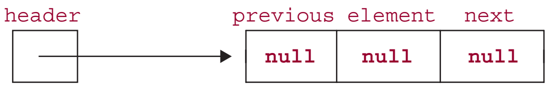


FIGURE 7.14 The `header` field in the `LinkedList` class

The `header` field always points to the same dummy entry, and the dummy entry's `element` field always contains **`null`**. The `next` field will point to the `Entry` object that houses the first element in the `LinkedList` object, and the `previous` field will point to the `Entry` object that houses the last element in the `LinkedList` object. Having a dummy entry instead of `head` and `tail` fields ensures that every `Entry` object in a linked list will have both a previous `Entry` object and a next `Entry` object. The advantage to this approach is that insertions and removals can be made without making a special case for the first and last elements in the list.

7.3.6 Creating and Maintaining a `LinkedList` Object

To get a better idea of how the fields in the `LinkedLinked` class and `Entry` class work in concert, let's create and maintain a `LinkedList` object. We start with a call to the default constructor:

```
LinkedList<String> names = new LinkedList<String>();
```

As shown in Figure 7.15, the default constructor makes the `previous` and `next` fields in the dummy entry point to the dummy entry itself. It turns out that this simplifies the definitions of the methods that insert or delete elements.

Next, we append an element to that empty `LinkedList` object:

```
names.add ("Betsy");
```

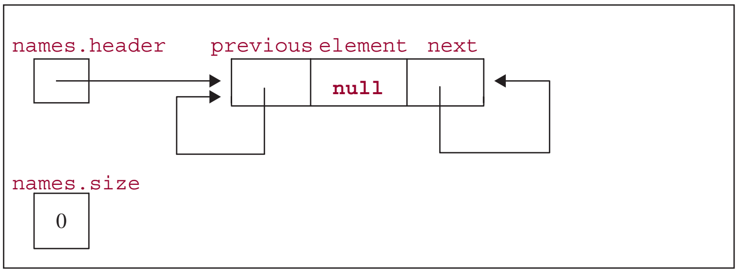


FIGURE 7.15 An empty `LinkedList` object, `names`

At this point, "Betsy" is both the first element in `names` and the last element in `names`. So the dummy entry both precedes and follows the `Entry` object that houses "Betsy". Figure 7.16 shows the effect of the insertion.

In general, adding an element at the end of a `LinkedList` object entails inserting the corresponding `Entry` object just before the dummy entry. For example, suppose the following message is now sent to the `LinkedList` object in Figure 7.16:

```
names.add ("Eric");
```

What is the effect of appending "Eric" to the end of the `LinkedList` object `names`? Eric's `Entry` object will come before the dummy entry and after Betsy's `Entry` object. See Figure 7.17.

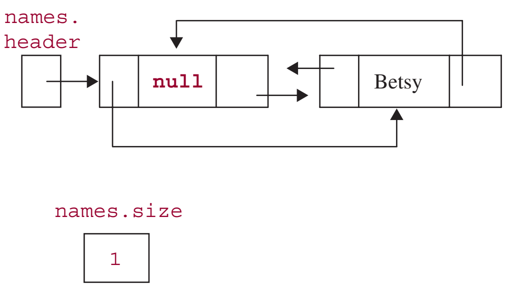


FIGURE 7.16 The effect of inserting "Betsy" at the back of the empty `LinkedList` object in Figure 7.15

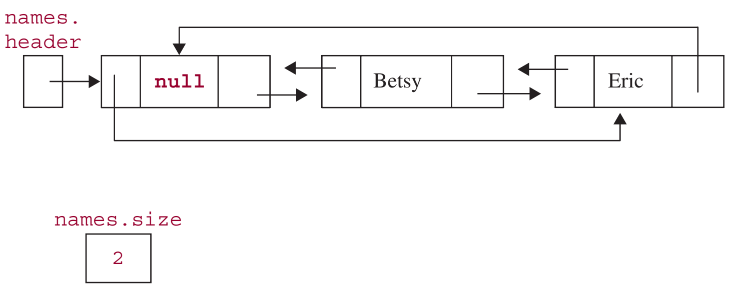


FIGURE 7.17 A two-element `LinkedList` object. The first element is "Betsy" and the second element is "Eric"

As you can see from Figure 7.17, a LinkedList object is stored circularly. The dummy entry precedes the first entry and follows the last entry. So we can iterate through a LinkedList object in the forward direction by starting at the first entry and repeatedly calling the next() method until we get to the dummy entry. Or we can iterate through a LinkedList object in the reverse direction by starting at the dummy entry and repeatedly calling the previous() method until we get to the first entry.

Finally, let's see what happens when the two-parameter add method is invoked. Here is a sample call:

```
names.add (1, "Don");
```

To insert "Don" at index 1, we need to insert "Don" in front of "Eric". To accomplish this, we need to create an Entry object that houses "Don", and adjust the links so that Entry object follows the Entry object that houses "Betsy" and precedes the Entry object that houses "Eric". Figure 7.18 shows the result.

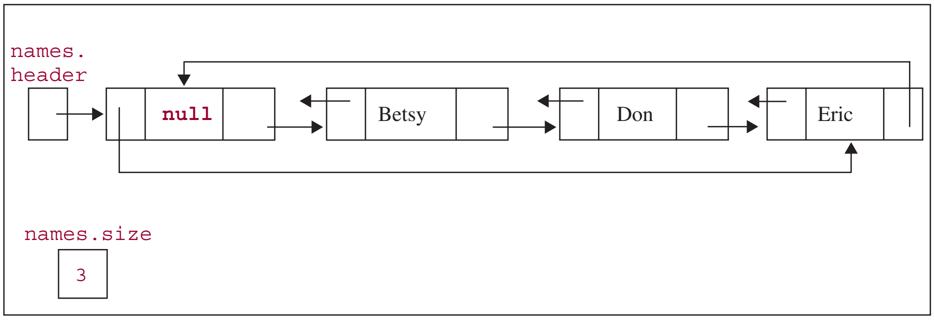


FIGURE 7.18 The LinkedList object from Figure 7.17 after the insertion of "Don" in front of "Eric" by the call names.add (1, "Don")

From the above examples, you should note that when an element is inserted in a LinkedList object, *no other elements in the list are moved*. In fact, when an element is appended with the LinkedList class's one-parameter add method, there are no loops or recursive calls, so worstTime(n) is constant. What about an insertion at an index? Section 7.4 investigates the definition of the two-parameter add method.

7.3.7 Definition of the Two-Parameter add Method

To finish up this foray into the developer's view of the LinkedList class, we will look at the definition of the two-parameter add method. Here is the method specification:

```
/**
 *  Inserts a specified element at a specified index.
 *  All elements that were at positions greater than or equal to the specified index
 *  before this call are now at the next higher position.  The worstTime(n) is O(n).
 *
 *  @param index – the specified index at which the element is to be inserted.
 *  @param element – the specified element to be inserted.
 *
 *  @throws IndexOutOfBoundsException – if index is less than zero or greater
 *          than size().
 *
 */
public void add (int index, E element)
```

For inserting an element at position `index`, the hard work is getting a reference to the `Entry` object that is currently at position `index`. This is accomplished—in the **`private`** method `entry`—in a loop that starts at `header` and moves forward or backward, depending on whether `index < size/2`.

Once a reference, `e`, to the appropriate `Entry` has been obtained, `element` is stored in a new entry that is put in front of `e` by adjusting a few `previous` and `next` references. These adjustments are accomplished in the **`private`** `addBefore` method:

```
/**
 * Inserts an Entry object with a specified element in front of a specified Entry object.
 *
 * @param element - the element to be in the inserted Entry object.
 * @param e - the Entry object in front of which the new Entry object is to be
 * inserted.
 *
 * @return - the Entry object that houses the specified element and is in front
 *                  of the specified Entry object.
 *
 */
private Entry<E> addBefore (E element, Entry<E> e)
{
   Entry<E> newEntry = new Entry<E>(element, e, e.previous);// insert newEntry in
                                              // front of e
   newEntry.previous.next = newEntry;    // make newEntry follow its predecessor
   newEntry.next.previous = newEntry;    // make newEntry precede its successor, e
   size++;
   modCount++;    // discussed in Appendix 1
   return newEntry;
}
```

Here, basically, is the definition of the two-parameter add method

```
public void add (int index, E element)
{
   if (index == size)
       addBefore (element, header);
   else
       addBefore (element, entry (index));
}
```

The bottom line in all of this is that to insert an `Entry` object in front of another `Entry` object, worstTime(n) is constant, but to get (a reference to) the `Entry` object at a given `index`, worstTime(n) is linear in n (because of the loop in the `entry` method). The insertion is accomplished by adjusting references, not by moving elements.

The actual definition of the two-parameter `add` method is a one-liner:

```
public void add (int index, A element)
{
    addBefore(element, (index==size ? header : entry(index)));
}
```

The '?' and ':' are part of the shorthand for the usual **`if/else`** statement[2]. The advantage of having a dummy entry is that every entry, even the first or last entry, has a predecessor and a successor, so there is no need for a special case to insert at the front or back of a `LinkedList` object.

The flow of the `remove (`**`int`**` index)` method is similar to that of `add (`**`int`**` index, E element)`. We first get a reference, `e`, to the `Entry` object at position `index`, and then adjust the predecessor and successor of `e`'s `Entry`.

As an application of the `LinkedList` class, we develop a line editor in Section 7.5.

7.4 Application: A Line Editor

A ***line editor*** is a program that manipulates text, line by line. At one time, line editors were state-of-the-art, but with the advent of full-screen editors (you move the cursor to the line you want to edit), line editors are seldom used. Linux/Unix and Windows still have a line editor, but they are used only when full-screen editors are unavailable—for example, after a system crash.

We assume that each line is at most 75 characters long. The first line of the text is thought of as line 0 (just as Java programmers refer to their zero-th child), and one of the lines is designated as the ***current line***. Each editing command begins with a dollar sign, and only editing commands begin with a dollar sign. There are seven editing commands. Here are four of the commands; the remaining three, and two system tests, are specified in Programming Project 7.5.

1. **$Insert**
   Each subsequent line, up to the next editing command, will be inserted in the text. If there is a designated current line, each line is inserted before that current line. Otherwise, each line is inserted at the end of the text; that is, the current line is then considered to be a dummy line beyond the last line of text. For example, suppose the text is empty and we have the following:

   $Insert
   Water, water every where,
   And all the boards did shrink;
   Water, water every where,
   Nor any drop to drink.

[2]For example, instead of writing

```
if (first > second)
   big = first;
else
   big = second;
```

we can simply write:

```
 big = (first > second) ? first : second;
```

This can be read as "If `first` is greater than `second`, assign to `big` the value of `first`. Otherwise, assign to `big` the value of `second`."

The syntax for a conditional expression is:

```
 condition ? expression_t : expression_f
```

The semantics is this: if the condition has the value **`true`**, then the value of the conditional expression is the value of *`expression_t`*. Otherwise, the value of the conditional expression is the value of *`expression_f`*. If you like to write cryptic code, you'll love the conditional operator, one of the legacies that Java got from C. Note that it is the only ***ternary*** operator in Java. That means it has three operands.

Then after the insertions, the text would be as follows, with a caret '> ' indicating the current line:

```
  Water, water every where,
  And all the boards did shrink;
  Water, water every where,
  Nor any drop to drink.
>
```

For another example, suppose the text is:

```
  Now is the
  time for
>citizens to come to
  the
  aid of their country.
```

The sequence

```
$Insert
all
good
```

will cause the text to become

```
  Now is the
  time for
  all
  good
>citizens to come to
  the
  aid of their country.
```

2. **$Delete *m n***
   Each line in the text between lines m and n, inclusive, will be deleted. The current line becomes the first line after the last line deleted. So if the last line of text is deleted, the current line is beyond any line in the text.
   For example, suppose the text is

```
  Now is the
  time for
  all
>good
  citizens to come to
  the
  aid of their country.
```

Then the command

```
$Delete 2 4
```

will cause the text to become

```
 Now is the
 time for
```

```
>the
  aid of their country.
```

If the next command is

```
$Delete 3 3
```

then the text becomes:

```
  Now is the
  time for
  the
>
```

The following error messages should be printed when appropriate:

Error: The first line number is greater the second.

Error: The first line number is less than 0.

Error: The second line number is greater than the last line number.

Error: The command should be followed by two integers.

3. **$Line m**
   Line *m* becomes the current line. For example, if the text is

```
  Mairzy doats
  an dozy doats
>an liddle lamsy divy.
```

   then the command

```
$Line 0
```

   will make line 0 the current line:

```
>Mairzy doats
  an dozy doats
  an liddle lamsy divy.
```

   An error message should be printed if *m* is either less than 0 or greater than the number of lines in the text or if no integer is entered. See command 2 above.

4. **$Done**
   This terminates the execution of the text editor. The entire text is printed.
   An error message should be printed for any illegal command, such as "$End", "$insert", or "Insert".

System Test 1 (Input is boldfaced):

Please enter a line; a command must start with a $.
$Insert

Please enter a line; a command must start with a $.
Yesterday, upon the stair,

Please enter a line; a command must start with a $..
I shot an arrow into the air.

```
Please enter a line; a command must start with a $.
It fell to earth, I know not where.

Please enter a line; a command must start with a $.
I met a man who wasn’t there.

Please enter a line; a command must start with a $.
$Delete 1 2

Please enter a line; a command must start with a $.
$Line 2

Please enter a line; a command must start with a $.
$Insert

Please enter a line; a command must start with a $.
He wasn’t there again today.

Please enter a line; a command must start with a $.
Oh how I wish he’d go away.

Please enter a line; a command must start with a $.
$Done
*********************
Here is the final text:
  Yesterday, upon the stair,
  I met a man who wasn’t there.
  He wasn’t there again today.
  Oh how I wish he’d go away.
>
```

System Test 2 (Input is boldfaced):

```
Please enter a line; a command must start with a $.
Insert
Error: not one of the given commands.

Please enter a line; a command must start with a $.
$Insert

Please enter a line; a command must start with a $.
There is no patch for stupidity.

Please enter a line; a command must start with a $.
$Line
Error: the command must be followed by a blank, followed by an integer.

Please enter a line; a command must start with a $.
$Line 2
Error: the number is greater than the number of lines in the text.

Please enter a line; a command must start with a $.
$Line 0
```

```
Please enter a line; a command must start with a $.
$Insert

Please enter a line; a command must start with a $.
As Kevin Mittnick said,

Please enter a line; a command must start with a $.
$Delete 0
Error: the command must be followed by a space, followed by an integer,
followed by a space, followed by an integer.

Please enter a line; a command must start with a $.
$Done
************************
Here is the final text:
  As Kevin Mittnick said,
> There is no patch for stupidity.
```

7.4.1 Design and Testing of the `Editor` Class

We will create an `Editor` class to solve this problem. To separate the editing aspects from the input/output aspects, there will be an `EditorUser` class that reads from an input file and prints to an output file. Then the same `Editor` class could later be used in an ***interactive program***, that is, a program in which the input is entered in response to outputs. That later program could have a graphical user interface or use console input/output. The design and implementation of the `EditorUser` class will be developed after we complete work on the `Editor` class.

Before we decide what fields and methods the `Editor` class should contain, we ask what does an editor have to do? From the commands given above, some responsibilities can be determined:

- to interpret whether the line contains a legal command, an illegal command or a line of text
- to carry out each of the four commands

When one of the errors described occurs, the offending method cannot print an error message because we want to separate editing from input/output. We could have the method return the error message as a `String`, but what if a method that is supposed to return a `String` has an error? You will encounter such a method in Project 7.1. For the sake of consistency, each error will throw a `RunTimeException`; the argument will have the specific error message. For example, in the `Editor` class we might have

```
throw new RunTimeException ("Error: not one of the given commands.\n");
```

Each error message can then be printed in the `EditorUser` class when the exception is caught. `RunTime Exception` is the superclass of most the exceptions thrown during execution: `NullPointerException`, `NumberFormatException`, `NoSuchElementException`, and so on.

Here are the method specifications for a default constructor and the five methods outlined previously. To ensure that each command method is properly invoked, a user has access only to the `interpret` method, which in turn invokes the appropriate command method.

```
/**
 *  Initializes this Editor object.
 *
 */
public Editor()

/**
 *  Intreprets whether a specified line is a legal command, an illegal command
 *  or a line of text.
 *
 *  @param s - the specified line to be interpreted.
 *
 *  @return the result of carrying out the command, if s is a legal command, and
 *          return null, if s is a line of text.
 *
 *  @throws RunTimeException - if s is an illegal command; the argument
 *          indicates the specific error.
 *
 */
public String interpret (String s)


/**
 *  Inserts a specified line in front of the current line.
 *
 *  @param s - the line to be inserted.
 *
 *  @throws RunTimeException - if s has more than MAX_LINE_LENGTH
 *          characters.
 *
 */
protected void insert (String s)


/**
 *  Deletes a specified range of lines from the text, and sets the current line
 *  to be the line after the last line deleted.
 *
 *  @param m - the beginning index of the range of lines to be deleted.
 *  @param n - the ending index of the range of lines to be deleted.
 *
 *  @throws RunTimeException - if m is less than 0 or if n is less than m or if
 *          n is greater than or equal to the number of lines of text.
 *
 */
protected void delete (int m, int n)


/**
 *  Makes a specified index the index of the current line in the text.
 *
 *  @param m - the specified index of the current line.
 *
```

```
     *  @throws RunTimeException - if m is less than 0 or greater than the
     *          number of lines of text.
     *
     */
    protected void setCurrentLineNumber (int m)


    /**
     *  Returns the final version of the text.
     *
     *  @return the final version of the text.
     *
     */
    protected String done()
```

The book's website includes a test suite, `EditorTest`. `EditorTest` is a subclass of `Editor` to allow the testing of the **`protected`** methods in the `Editor` class. For example, here is a simple test of the `insert` method:

```
    @Test
    public void testInsert()
    {
        editor.interpret ("$Insert");
        editor.insert ("a");
        editor.insert ("b");
        String actual = editor.interpret ("$Done"),
               expected = "   a\n   b\n>  \n";
        assertEquals (expected, actual);
    } // method testInsert
```

Note that this test does not access the **`protected`** *fields* of the `Editor` class because there is no guarantee that those fields will be relevant to the definition of the `insert` method. Recall from Chapter 2 that unit testing applies only to a method's specification.

Another interesting feature of the `EditorTest` class is that tests that expect a `RuntimeException` to be thrown must have a **`catch`** block to ensure that the appropriate exception message is included, and must throw `RuntimeException` within that **`catch`** block! For example, here is one of the tests:

```
    @Test (expected = RuntimeException.class)
    public void testInterpretBadLine()
    {
             try
             {
                     editor.interpret ("$Delete 7 x");
             } // try
             catch (RuntimeException e)
             {
                     assertEquals ("java.lang.RuntimeException: " +
                                     Editor.TWO_INTEGERS_NEEDED, e.toString());
                     throw new RuntimeException();
             } // catch RuntimeException
    } // method testInterpretBadLine
```

There is one more issue related to EditorTest: what can we use as a stub for each method in Editor so that all of the tests will initially fail? We cannot use the normal stub

```
throw new UnsupportedOperationException();
```

because UnsupportedOperationException is a subclass of RuntimeException. So, instead the stub will be

```
throw new OutOfMemoryError();
```

As expected, all tests initially failed.

In order to define the Editor methods, we have to decide what fields we will have. One of the fields will hold the text, so we'll call it text. The text will be a sequence of strings, and we will often need to make insertions/deletions in the interior of the text, so text should be (a reference to) an instance of the LinkedList class (surprise!). To keep track of the current line, we will have a ListIterator field, current. A **boolean** field, inserting, will determine whether the most recent command was $Insert.

Here are the constant identifiers and fields:

```
public final static char COMMAND_START = '$';

public final static String INSERT_COMMAND = "$Insert";

public final static String DELETE_COMMAND = "$Delete";

public final static String LINE_COMMAND = "$Line";

public final static String DONE_COMMAND = "$Done";

public final static String BAD_LINE_MESSAGE =
       "Error: a command should start with " + COMMAND_START + ".\n";

public final static String BAD_COMMAND_MESSAGE =
       "Error: not one of the given commands.\n";

public final static String INTEGER_NEEDED =
       "Error: The command should be followed by a blank space, " +
       "\nfollowed by an integer.\n";

public final static String TWO_INTEGERS_NEEDED =
       "Error: The command should be followed by a blank space, " +
       "\nfollowed by an integer, followed by a blank space, " +
       "followed by an integer.\n";

public final static String FIRST_GREATER =
       "Error: the first line number given is greater than the second.\n";

public final static String FIRST_LESS_THAN_ZERO =
       "Error: the first line number given is less than 0.\n";

public final static String SECOND_TOO_LARGE =
       "Error: the second line number given is greater than the " +
       "\nnumber of the last line in the text.\n";
```

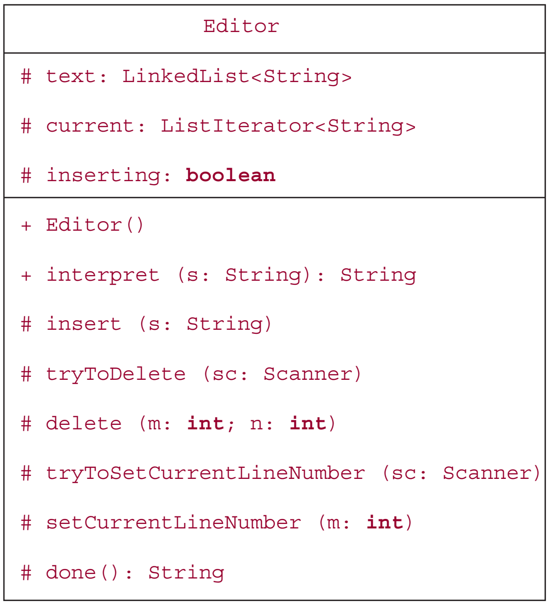
```
public final static String M_LESS_THAN_ZERO =
       "Error: the number is less than 0.\n";

public final static String M_TOO_LARGE =
       "Error: the number is larger than the number of lines in the text.\n";

public final static String LINE_TOO_LONG =
       "Error: the line exceeds the maximum number of characters allowed, ";

public final static int MAX_LINE_LENGTH = 75;


protected LinkedList<String> text;

protected ListIterator<String> current;

protected boolean inserting;
```

The `delete` method can be invoked only if the command line has two integers. So we will have an auxiliary method, **`protected void`** `tryToDelete (Scanner sc)`, which calls `delete` provided there are two integers in the command line. There is a similar auxiliary method for the `setCurrentLineNumber` method. The Figure 7.19 has the UML diagram for the `Editor` class.

7.4.2 Method Definitions for the `Editor` Class

As usual, the default constructor initializes the fields:

```
public Editor()
{
      text = new LinkedList<String>();
```



```
Editor
-----------------------------------------
# text: LinkedList<String>
# current: ListIterator<String>
# inserting: boolean
-----------------------------------------
+ Editor()
+ interpret (s: String): String
# insert (s: String)
# tryToDelete (sc: Scanner)
# delete (m: int; n: int)
# tryToSetCurrentLineNumber (sc: Scanner)
# setCurrentLineNumber (m: int)
# done(): String
```

FIGURE 7.19 The class diagram for the `Editor` class

```
        current = text.listIterator();
        inserting = false;
    } // default constructor
```

We can estimate the time requirements for this method because it does not call any other methods in the `Editor` class. In general, the time requirements for a given method depend on the time for the methods called by the given method. The worstTime(n), where n is the number of lines of text, is constant. For the remainder of the `Editor` class's methods, we postpone an estimate of worstTime(n) until all of the methods have been defined.

The `interpret` method proceeds as follows. There are special cases if the line is blank or if the first character in the line is not '$': the `insert` method is invoked if `inserting` is true; otherwise, a bad-line exception is thrown. If the first character in the line is '$', the line is scanned and action appropriate to the command is taken. For the $Delete and $Line commands, the remaining tokens must first be checked—to make sure they are integers—before the `delete` and `setCurrentLineNumber` methods can be called. That allows the `delete` and `setCurrentLineNumber` methods to have **`int`** parameters.

Here is the definition of the `interpret` method:

```
    public String interpret (String s)
    {
        Scanner sc = new Scanner (s);

        String command;

        if (s.length() == 0 || s.charAt (0) != COMMAND_START)
               if (inserting)
                   insert (s);
               else
                   throw new RuntimeException (BAD_LINE_MESSAGE);
        else
        {
               command = sc.next();
               if (command.equals (INSERT_COMMAND))
                   inserting = true;
               else
               {
                   inserting = false;
                   if (command.equals (DELETE_COMMAND))
                          tryToDelete (sc);
                   else if (command.equals (LINE_COMMAND))
                          tryToSetCurrentLineNumber (sc);
                   else if (command.equals (DONE_COMMAND))
                          return done();
                   else
                          throw new RuntimeException (BAD_COMMAND_MESSAGE);
               } // command other than insert
        } // a command
        return null;
    } // method interpret
```

The definition of the insert method is straightforward. The only error checking is for a too-long line; otherwise, the parameter s is inserted into the text in front of the current line. The method definition is:

```
protected void insert (String s)
{
   if (s.length() > MAX_LINE_LENGTH)
       throw new RuntimeException (LINE_TOO_LONG +
                                   MAX_LINE_LENGTH + "\n");
   current.add (s);
} // insert
```

The $Delete command can fail syntactically, if the line does not have two integers, or semantically, if the first line number is either greater than the second or less than zero, or if the second line number is greater than the last line in the text. The tryToDelete method checks for syntax errors:

```
protected void tryToDelete (Scanner sc)
{
     int m = 0,
         n = 0;

     try
     {
         int m = sc.next();
         int n = sc.next();
     }// try
     catch (RuntimeException e)
     {
         throw new RuntimeException (TWO_INTEGERS_NEEDED);
     } // not enough integer tokens
     delete (m, n);
} // method tryToDelete
```

The call to the delete method must be outside of the **try** block so that the run-time exceptions thrown within the delete method will pass through tryToDelete and back to the method that calls interpret, instead of being caught in tryToDelete's **catch** block.

The delete method checks for semantic errors. If there are no errors, The ListIterator object current is positioned at line m, and a loop removes lines m through n. Then current will automatically be positioned beyond the last line removed. Here is the definition of the delete method:

```
protected void delete (int m, int n)
{
      if (m > n)
             throw new RuntimeException (FIRST_GREATER);
      if (m < 0)
             throw new RuntimeException (FIRST_LESS_THAN_ZERO);
      if (n >= text.size())
             throw new RuntimeException (SECOND_TOO_LARGE);
      current = text.listIterator (m);
      for (int i = m; i <= n; i++)
```

```
        {
               current.next();
               current.remove ();
        } // for
  } // method delete
```

The tryToSetCurrentLineNumber method is similar to tryToDelete, except there is only one integer expected on the command line:

```
  protected void tryToSetCurrentLineNumber (Scanner sc)
  {
        int m = 0;

        try
        {
               int m = sc.next();
        } // try
        catch (RuntimeException e)
        {
               throw new RuntimeException (INTEGER_NEEDED);
        } // no next token or token not an integer
        setCurrentLineNumber (m);
  } // method tryToSetCurrentLineNumber
```

The setCurrentLineNumber method, called if there are no syntactic errors, checks for semantic errors, and if none are found, re-positions current to the line whose line number is m. Here is the definition of the setCurrentLineNumber method:

```
  protected void setCurrentLineNumber (int m)
  {
        if (m < 0)
               throw new RuntimeException (M_LESS_THAN_ZERO);
        if (m > text.size())
               throw new RuntimeException (M_TOO_LARGE);
        current = text.listIterator (m);
  } // method setCurrentLineNumber
```

Finally, the done method returns a String representation of the text: suitable for printing. We create itr, a ListIterator object (specifically, a ListItr object) to iterate through the LinkedList object text. The current line should have a '> ' in front of it. But how can we determine when the line that itr is positioned at is the same as the line that current is positioned at? Here is one possibility:

```
  itr.equals (current)
```

The ListItr class does not define an equals method, so the Object class's version of equals is invoked. But that method compares references, not objects. The references will never be the same since they were, ultimately, allocated by different calls to the ListItr constructor. So that approach will not work.

Alternatively, we could compare elements:

```
  itr.next().equals (current.next())
```

But this could give incorrect information if the text had duplicate lines. The safe way to compare is by the `nextIndex()` method, which returns the index of the element that the iterator is positioned at. Here is the method definition:

```
protected String done()
{
      ListIterator<String> itr = text.listIterator();

      while (itr.hasNext())
               if (itr.nextIndex() == current.nextIndex())
                      s = s + ">  " + itr.next() + '\n';
               else
                      s = s + "   " + itr.next() + '\n';
      if (!current.hasNext())
               s = s + ">   " + '\n';
      return s;
} // method done
```

7.4.3 Analysis of the `Editor` Class Methods

To estimate the time requirements for the methods in the `Editor` class, let *n* represent the size of the text—this is not necessarily the same as the `n` used as a parameter in several methods. The `delete` method calls the one-parameter `listIterator (int index)` method, for which worstTime(*n*) is linear in *n*. There is then a loop in which some elements in the text are removed; each removal takes constant time. This number of elements is certainly less than or equal to *n*, the total number of elements in the text. So for the `delete` method, worstTime(*n*) is linear in *n*.

The `setCurrentLineNumber` method also calls the `listIterator (int index)` method, and that makes the worstTime(*n*) linear in *n* for the `setCurrentLineNumber` method. The `done` method loops through the text, so its worstTime(*n*) is also linear in *n*. All other methods take constant time, except those whose worstTime(*n*) is linear in *n* owing to their calling the `delete`, `setCurrentLineNumber`, or `done` methods.

7.4.4 Design of the `EditorUser` Class

We will create another class, `EditorUser`, to illustrate the use of input and output files for editing text. The paths for the input and output files are scanned in from the keyboard, and a file scanner and file writer for those two files are declared and opened in the `run( )` method. The only other responsibility of the `EditorUser` class is to edit the input file. Here are the method specifications for the `editText()` method.

```
/**
 *  Edits the text by performing the input-file commands
 *
 *  @param fileScanner - a Scanner object over the input file
 *  @param printWriter - a PrintWriter object that holds the edited text.
 *
 */
public void editText (Scanner fileScanner, PrintWriter printWriter)
```

Figure 7.20 has all the UML class diagrams for this project.

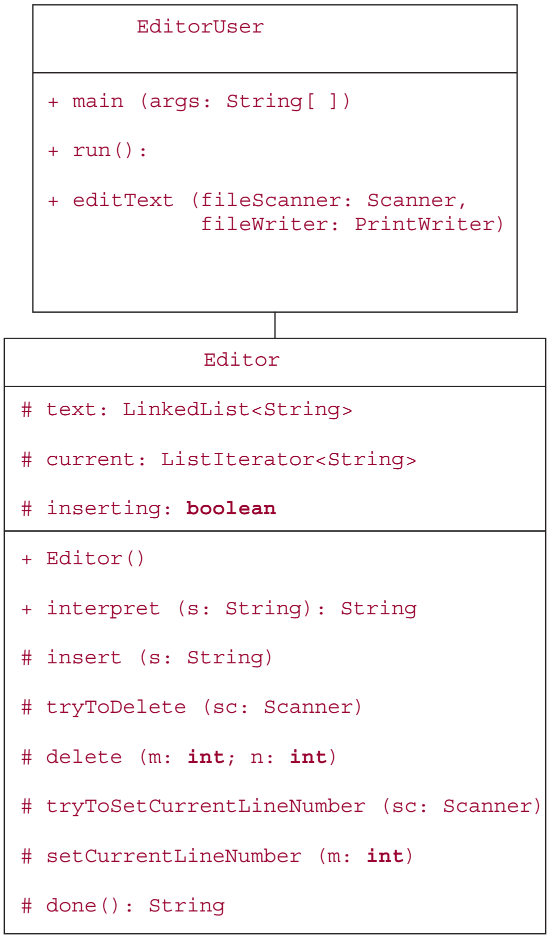

EditorUser
+ main (args: String[ ])
+ run():
+ editText (fileScanner: Scanner, fileWriter: PrintWriter)
Editor
# text: LinkedList<String>
# current: ListIterator<String>
# inserting: boolean
+ Editor()
+ interpret (s: String): String
# insert (s: String)
# tryToDelete (sc: Scanner)
# delete (m: int; n: int)
# tryToSetCurrentLineNumber (sc: Scanner)
# setCurrentLineNumber (m: int)
# done(): String

FIGURE 7.20 Class diagrams for the Editor project

The book's website has the `EditorUserTest` class to test the `editText` method.

7.4.5 Implementation of the `EditorUser` Class

The `run()` method is similar to the file-oriented `run()` method of the `Scores3` class in Chapter 2:

```
public void run()
{
   Scanner fileScanner = null;

   PrintWriter printWriter = null;

   final String IN_FILE_PROMPT =
          "\n\nPlease enter the path for the input file: ";
```

```
        final String OUT_FILE_PROMPT =
               "\n\nPlease enter the path for the output file: ";

        final String IO_EXCEPTION_MESSAGE = "The file was not found.\n\n";

        Scanner keyboardScanner = new Scanner (System.in);

        String inFilePath,
               outFilePath;

        boolean pathsOK = false;

        while (!pathsOK)
         {
             try
             {
                 System.out.print (IN_FILE_PROMPT);
                 inFilePath = keyboardScanner.nextLine();
                 fileScanner = new Scanner (new File (inFilePath));
                 System.out.print (OUT_FILE_PROMPT);
                 outFilePath = keyboardScanner.nextLine();
                 printWriter = new PrintWriter (new FileWriter (outFilePath));
                 pathsOK = true;
             } // try
             catch (IOException e)
             {
                 System.out.println (IO_EXCEPTION_MESSAGE + e);
             } // catch I/O exception
         } // while
         editText (fileScanner, printWriter);
         printWriter.close();
     } // method run
```

The `editText()` method loops through the input file, with a **try**-block and a **catch**-block to handle all of the editing errors that may occur. During each loop iteration, the `interpret` method is called. The return value from this call will be **null** unless the command is “$Done”, in which case the final text is printed.

Here is the definition:

```
     public void editText (Scanner fileScanner, PrintWriter printWriter)
     {
           final String FINAL_MESSAGE =
                "\n\n***********************\nHere is the final text:\n";

           String line = new String(),
                result = new String();

           while (true)
```

```
        {
               try
               {
                      line = fileScanner.nextLine();
                      printWriter.println (line);
                      result = editor.interpret (line);
                } // try
                catch (RuntimeException e)
                {
                       printWriter.println (e);
                } // catch RuntimeException
                if (line.equals (Editor.DONE_COMMAND))
                {
                       printWriter.println (FINAL_MESSAGE + result);
                       break;
                } // line is done command
        } // while
  } // method editText
```

This method accesses the **public** constant DONE_COMMAND from the Editor class. That enables us to avoid the dangerous practice of defining the same constant identifier twice. The danger is that this identifier might be re-defined in a subsequent application, for example, if the developer of the Editor class decided to change the command-start symbol from '$' to '#'.

SUMMARY

A ***linked list*** is a List object (that is, an object in a class that implements the List interface) in which the following property is satisfied:

> Each element is contained in an object, called an Entry object, that also includes a reference, called a *link*, to another Entry object. For each Entry object except the one that holds the last element in the collection, the link is to the Entry object that contains the next element in the collection.

A linked list that also satisfies the following property:

> Each Entry object except the first also includes a link to the Entry object that contains the previous element.

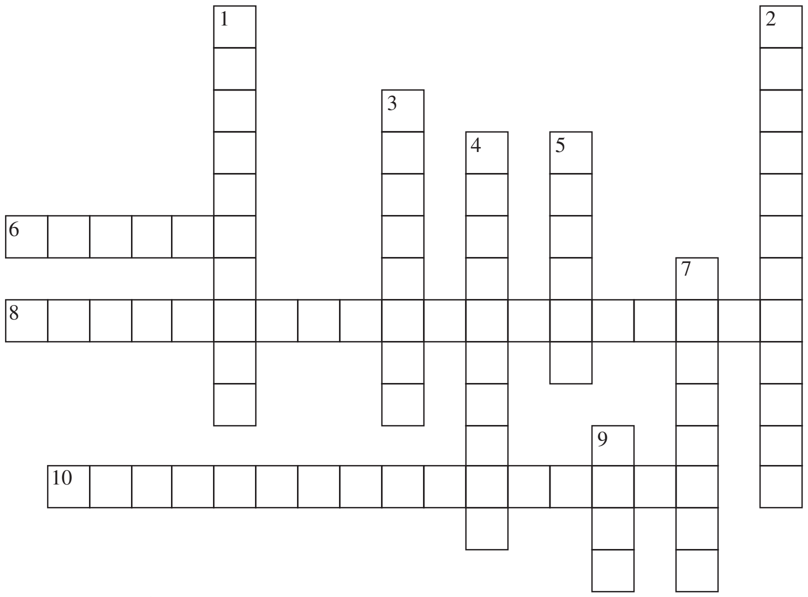
is called a ***doubly-linked list***. Otherwise, it is called a ***singly-linked list***.

The SinglyLinkedList class implements a singly-linked list. The purpose of developing the SinglyLinkedList class is to introduce you to the topics of links and iterators, and thus to prepare you for the focal point of the chapter: the LinkedList class, part of the Java Collections Framework. LinkedList objects lack the random-access ability of ArrayList objects. But, by using an iterator, an element can be added to or removed from a LinkedList in only constant time; for adding to or removing from an ArrayList object, worstTime(n) is linear in n. This advantage of LinkedList objects is best suited for consecutive insertions and deletions because, for the task of getting to the index of the first insertion or deletion, worstTime(n) is linear in n.

The Java Collection Framework's implementation of the LinkedList class stores the elements in a circular, doubly-linked structure with a dummy entry. Another possible implementation is a non-circular, doubly-linked structure with head and tail fields.

The application, a simple line editor, took advantage of the LinkedList class's ability to quickly make consecutive insertions and deletions anywhere in a LinkedList object.

CROSSWORD PUZZLE



ACROSS

6. A class that is embedded in another class is called a ___________ class.

8. The only operator in Java that has three operands (separated by '?' and ':').

10. A linked list in which each `Entry` object includes a link to the `Entry` object that contains the previous element in the list.

DOWN

1. A program that edits text, line-by-line.

2. In the **`LinkedList`** class, the method that associated the calling object with an iterator that can move forward or backward is ___________ ().

3. The worstTime(n) for the public methods in the `Listltr` class.

4. The class that catches the expectations thrown in the `Editor` class.

5. The field in the `LinkedList` class that contains a reference to a special `Entry` object, called a "dummy entry."

7. In the `SinglyLinkedList` or `LinkedList` class, the method that associated the calling object with an iterator that can move forward only is ___________ ().

9. In a linked list entry, a reference to the entry that contains the next element in the linked list.

CONCEPT EXERCISES

7.1 In the SinglyLinkedList class, define the following method without using an iterator.

```
/**
 *  Finds the element at a specified position in this LinkedList object.
 *  The worstTime(n) is O(n).
 *
 *  @param index – the position of the element to be returned.
 *
 *  @return the element at position index.
 *
 *  @throws IndexOutOfBoundsException – if index is less than 0 or greater than
 *                   or equal to size().
 */
public E get (int index)
```

7.2 Re-do Concept Exercise 7.1 by using an iterator.

7.3 Suppose we added each of the following methods to the ArrayList class:

```
public boolean addFirst (E element);
public boolean addLast (E element);
public E getFirst();
public E getLast();
public E removeFirst();
public E removeLast();
```

Estimate worstTime(n) for each method.

7.4 The listIterator() method can be called by a LinkedList object, but is not defined within the LinkedList class. In what class is that listIterator() method defined? What is that definition?

7.5 One of the possibilities for fields in the LinkedList class was to have head and tail fields, both of type Entry, where the Entry class had element and next fields, but no previous field. Then we would have a singly-linked list.

a. Define the addLast method for this design. Here is the method specification:

```
/**
 *  Appends a specified element to (the back of) this LinkedList object.
 *
 *  @param element – the element to be appended.
 *
  *  @return true.
 *
 */
public boolean addLast (E element)
```

b. The definition of the removeLast() method would need to make **null** the next field in the Entry object before the Entry object tail. Could we avoid a loop in the definition of removeLast() if, in the LinkedList class, we added a beforeTail field that pointed to the Entry object before the Entry object tail ? Explain.

7.6 How can you distinguish between a call to the `add (E element)` method in the `LinkedList` class and a call to the `add (E element)` method in the `ListItr` class?

7.7 Explain how to remove "Don" from the `LinkedList` object in Figure 7.18. Explain why, for the definition of the method `remove (Object obj)`, worstTime(*n*) is linear in *n*?

7.8 In the Java Collections Framework, the `LinkedList` class is designed as a circular, doubly-linked list with a dummy entry (pointed to by the `header` field). What is the main advantage of this approach over a circular, doubly-linked list with `head` and `tail` fields?

7.9 For the three methods in the `EditorUser` class, estimate worstTime(*n*), where *n* represents the number of lines of text.

PROGRAMMING EXERCISES

7.1 Use the `SinglyLinkedList` class three times. First, create a `SinglyLinkedList` object, `team1`, with elements of type `String`. Add three elements to `team1`. Second, create `team2`, another `SinglyLinkedList` object with elements of type `String`. Add four elements to `team2`. Finally, create a `SinglyLinkedList` object, `league`, whose elements are `SinglyLinkedList` objects of teams. Add `team1` and `team2` to `league`.

7.2 Hypothesize the output from the following method segment:

```
LinkedList<Character> letters = new LinkedList<Character>();

ListIterator<Character> itr = letters.listIterator();

itr.add ('f');
itr.add ('t');
itr.previous();
itr.previous();
itr.add ('e');
itr.add ('r');
itr.next();
itr.add ('e');
itr.add ('c');
itr = letters.listIterator();
itr.add ('p');
System.out.println (letters);
```

Test your hypothesis.

7.3 Rewrite the code in Programming Exercise 7.2 without using an iterator. For example, you would start with:

```
LinkedList<Character> letters = new LinkedList<Character>();

letters.add (0, 'f');
```

Test your revision.

7.4 Rewrite the code in Exercise 7.2 with a native array. For example, you would start with:

```
char [ ] letters = new char [10];

letters [0] = 'f';
```

Test your revision.

7.5 Hypothesize the error in the following code:

```
LinkedList<Double> duesList = new LinkedList<Double>();

ListItr<Double> itr = duesList.listIterator();
```

Test your hypothesis.

7.6 Suppose we have the following:

```
LinkedList<Double> weights = new LinkedList<Double>();

ListIterator<Double> itr;

weights.add (5.3);
weights.add (2.8);
itr = weights.listIterator();
```

Hypothesize which of the following sequences of messages would now be legal:

a. `itr.add (8.8); itr.next(); itr.remove();`

b. `itr.add (8.8); itr.remove(); itr.next();`

c. `itr.next(); itr.add (8.8); itr.remove();`

d. `itr.next(); itr.remove(); itr.add (8.8);`

e. `itr.remove(); itr.add (8.8); itr.next();`

f. `itr.remove(); itr.next(); itr.add (8.8);`

Test your hypotheses.

7.7 Suppose you decided to rewrite the `VeryLongInt` class, from Chapter 6, with a `LinkedList` instead of an `ArrayList`. The main change is to replace each occurrence of the identifier `ArrayList` with `LinkedList`. Another change, in the `String` -parameter constructor, is to replace

```
digits = new ArrayList<Integer> (s.length());
```

with

```
digits = new LinkedList<Integer>();
```

But the `add` method will now take quadratic time because the `least` method will now take linear time. Modify the `least` method—including its heading—so that its worstTime(n) will be constant. Make the corresponding changes to the `add` method so that method will take only linear time.

7.8 Rewrite the `insert` method in the `Editor` class to insert the given line *after* the current line. For example, if the text is

```
>I was
  older then
```

and the command is

```
$Insert
so much
```

then the text becomes

```
  I was
>so much
older then
```

The newly added line becomes the current line.

7.9 Modify the `EditorUser` class to work with commands entered from the keyboard instead of a file. The output should go to the console window. Test your changes by entering, from the keyboard, the lines in editor.in1 from the Editor directory on the book's website.

7.10 Unit test and define the following method:

```
/**
 *  Removes the first and last 4-letter word from a given LinkedList<String> object.
 *  Each word will consist of letters only.
 *  The worstTime(n) is O(n).
 *
 *  @param list - the LinkedList<String> object.
 *
 *  @throws NullPointerException - if list is null.
 *  @throws NoSuchElementException - if list is not null, but list has no 4-letter
 *                                   words or only one 4-letter word.
 *
 */
public static void bleep (LinkedList<String> list)
```

Programming Project 7.1

Expanding the `SinglyLinkedList` Class

Expand the `SinglyLinkedList` class from Lab 12 by providing genuine definitions—not just thrown exceptions—for each of the following methods:

```
/**
 *  Adds a specified element at the back of this SinglyLinkedList object.
 *
 *  @param element - the element to be inserted.
 *
```

```
     *  @return true.
     *
     */
    public boolean add (E element)


    /**
     *  Inserts the elements of this SinglyLinkedList object into an array in the same
     *  order as in this SinglyLinkedList object.  The worstTime(n) is O(n).
     *
     *  @return a reference to the array that holds the same elements, in the same
     *                order, as this SinglyLinkedList object.
     *
     */
    public Object [ ] toArray ()


    /**
     *  Determines if this SinglyLinkedList object contains all of the elements from a
     *  specified collection.
     *
     *  @param c - the specified collection.
     *
     *  @return true - if this SinglyLinkedList object contains each element of c;
     *                 otherwise, return false.
     *
     *  @throws NullPointerException - if c is null.
     *
     */
    public boolean containsAll (Collection<?> c)


    /**
     *  Determines if this SinglyLinkedList object is equal to obj.
     *
     *  @param obj - an object whose equality to this SinglyLinkedList object is
     *               being tested.
     *

     *  @return true - if obj is a SinglyLinkedList object of the same size as this
     *                SinglyLinkedList object, and at each index, the element in this
     *                SinglyLinkedList object is equal to the element at the same
     *                index in obj.
     *
     */
    public boolean equals (Object obj)
```

For unit testing, modify the `SinglyLinkedTest` class from the book's website.

Programming Project 7.2

Implementing the `remove()` Method in `SinglyLinkedListIterator`

1. Modify the `SinglyLinkedListIterator` class by implementing the `remove()` method. Here are revised fields that class and a revised definition of the `next()` method:

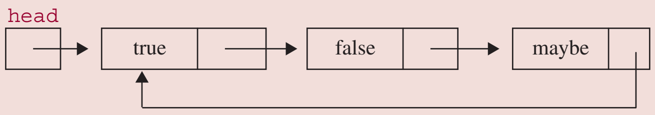
```
protected Entry previous,    // reference to Entry before lastReturned Entry
                lastReturned,   // reference to Entry with element returned
                               // by most recent call to next( ) method.
                next;          // reference to Entry with element that will be
                               // returned by subsequent call to next( ) method


public E next ()
{
    if (lastReturned != null)
         previous = lastReturned;
    lastReturned = next;
    next = next.next;
    return lastReturned.element;
} // method next
```

2. For unit testing of your `remove()` method, update the `SinglyLinkedTest` class.

Programming Project 7.3

Making a Circular Singly Linked List Class

Modify the `SinglyLinkedList` class to be circular. That is, the entry after the last entry should be the entry referenced by `head`. Here is an example, with three elements:



The only methods you need to implement are the five methods listed in Section 7.2 and the `iterator()` method from Section 7.2.2. You will also need to test those methods.

Programming Project 7.4

Alternative Implementation of the LinkedList Class

Implement the `LinkedList` class with `head` and `tail` fields instead of a `header` field that points to a dummy entry. Your implementation should be doubly-linked, that is, each `Entry` object should have a reference to the previous `Entry` object and a reference to the next `Entry` object. You get to choose whether your implementation will be circular. The `Entry` class will be unchanged from the header implementation, but the `ListItr` class will need to be modified.

Create a `LinkedListTest` class to test your `LinkedList` and `ListItr` classes.

Programming Project 7.5

Expanding the Line Editor

Expand the Line Editor project by implementing the following additional commands:

5. `$Change  %X%Y%`

Effect: In the current line, each occurrence of the string given by X will be replaced by the string given by Y.

Example Suppose the current line is

```
bear ruin'd choirs, wear late the sweet birds sang
```

Then the command

```
$Change %ear%are%
```

will cause the current line to become

```
bare ruin'd choirs, ware late the sweet birds sang
```

If we then issue the command

```
$Change %wa%whe%
```

the current line will be

```
bare ruin'd choirs, where late the sweet birds sang
```

Notes:

a. If either X or Y contains a percent sign, it is the end-user's responsibility to choose another delimiter. For example,

```
$Change #0.16#16%#
```

(continued on next page)

(continued from previous page)

b. The string given by Y may be the null string. For example, if current line is

```
aid of their country.
```

then the command

```
$Change  %of %%
```

will change the current line to

```
aid their country.
```

c. If the delimiter occurs fewer than three times, the error message to be generated is

```
*** Error:Delimiter must occur three times. Please try again.
```

6. `$Last`

Effect: The line number of the last line in the text has been returned.

Example Suppose the text is

```
  I heard a bird sing
> in the dark of December.
  A magical thing
  and a joy to remember.
```

The command

```
$Last
```

will cause 3 to be returned. The text and the designation of the current line are unchanged.

7. `$GetLines` **`m n`**

Effect: Each line number and line in the text, from lines m through n, inclusive, will be returned.

Example Suppose the text is

```
  Winston Churchill once said that
> democracy is the worst
  form of government
  except for all the others.
```

The command

```
$GetLines 0 2
```

will cause the following to be returned:

```
0  Winston Churchill once said that
1  democracy is the worst
2  form of government
```

The text and the designation of the current line are unchanged.

Note: If no line numbers are entered, the entire text should be returned. For example,

```
$GetLines
```

is the command to return the entire text, with line numbers.

As with the delete command, an error message should be generated if (1) *m* is greater than *n* or if (2) *m* is less than 0 or if (3) *n* is greater than the last line number in the text.

Expand the `EditorTest` class to validate your changes.

System Test 1 (For simplicity, prompts are omitted. Error messages and the values returned by $GetLines and $Last are shown in boldface):

```
    $Insert
    You can fool
    some of the people
    some of the times,
    but you cannot foul
    all of the people
    all of the time.
    $Line 2
    $GetLines 2 1
    Error: The first line number is greater than the second.

    $ GetLines 2 2
    2 some of the times,
    $Change %s%%
    $GetLines 2 2
    2 some of the time,
    $Change %o%so
    Error: Delimiter must occur three times.  Please try again.

    $Change %o%so%
    $GetLines 2 2
    2 some sof the time,
    Change
    Error: a command should start with $.

    $Change  %sof%of%
    $GetLines 2 2
    2  some of the time,
    $Line 0
    $Insert
    Lincoln once said that
    you can fool
    some of the people
    all the time and
    all of the time and
```

(continued on next page)

(continued from previous page)

```
$Last
10
$GetLines  0 10
0  Lincoln once said that
1  you can fool
2  some of the people
3  all the time and
4  all of the time and
5  You can fool
6  some of the people
7  some of the time,
8  but you cannot foul
9  all of the people
10 all of the time.
$Line 5
$Change %Y%y%
$GetLines  5 5
5  you can fool
$Line 6
$Change %some%all%
$GetLines 6  6
6  all of the people
$Line  8
$Change  %ul%ol%
$GetLines 8 8
8  but you cannot fool
$Line 9
$Change %ee%eo%
$GetLines 9 9
9  all of the people
$Delete 3 3
$GetLines 0 10
Error: The second line number is greater than the number of the last line in
            the text.

$Last
9
$GetLines 0  9
0  Lincoln once said that
1  you can fool
2  some of the people
3  all of the time and
4  you can fool
```

```
5  all of the people
6  some of the time,
7  but you cannot fool
8  all of the people
9  all of the time.
$Done
Here is the final text:
    Lincoln once said that
    you can fool
    some of the people
  > all of the time and
    you can fool
    all of the people
    some of the time,
    but you cannot fool
    all of the people
    all of the time.
```

System Test 2

```
$Insert
Life is full of
successes and lessons.
$Delete 1 1
$Insert
wondrous oppurtunities disguised as
hopeless situations.
$Last
2
$GetLines
0  Life is full of
1  wondrous oppurtunities disguised as
2  hopeless situations.
$Line 1
$Change %ur%or%
$GetLines  0  2
0  Life is full of
1  wondrous opportunities disguised as
2  hopeless situations.
$Done
Here is the final text:
  Life is full of
> wondrous opportunities disguised as
  hopeless situations.
```

Programming Project 7.6

An Integrated Web Browser and Search Engine, Part 3

In this part of the project, you will add functionality to the forward and backward buttons in your browser. Up to now, the end user can type a URL in the input line, can click on the home button, and can click on a link (if there is one) in the currently displayed page. Your revisions will allow the end user to go backward or forward to other web pages in the current chain.

According to standard web-browser protocol, you cannot go where you already are. So, for example, if you are on the home page, nothing happens if you click on the Home button. Also, whenever a new page is printed, all forward links are removed. For example, if you click on browser2, then browser4, then back, then home, the forward button would now be disabled (and colored red), so you could not click Forward to get to browser4.

The only class you will be altering is your listener class.

The web pages you can use to test your project—home.in1, browser.in1, browser.in2, browser.in4, and browser.in5—are the same as in Programming Project 2.1 from Chapter 2.

When a button is enabled, its color should be green.

When a button is disabled, its color should be red.

Here are some system tests that your project must pass.

System Test 1:
click on browser2, browser4, back (browser2 appears), enter browser.in5 in the
input line, click on back (browser2 appears), forward (browser5 appears)

At this point, the Forward button is disabled.

System Test 2:

click on browser2, browser4, back (browser2 appears), home, back
(browser2 appears), back (home appears), forward (browser2 appears),
forward (home appears)

At this point, the Forward button is disabled.

Stacks and Queues

CHAPTER 8

In this chapter we introduce two more abstract data types: stacks and queues. Stacks and queues can be modified in only a very limited way, so there are straightforward implementations, that is, data structures, of the corresponding data types. Best of all, stacks and queues have a wide variety of applications. We'll start with stacks because a stack is somewhat easier to implement than a queue.

CHAPTER OBJECTIVES

1. Understand the defining properties of stacks and queues, and how these properties are violated by the Java Collections Framework's `Stack` class and `Queue` interface.
2. For both stacks and queues, be able to develop contiguous and linked implementations that do not violate their defining properties.
3. Explore the use of stacks in the implementation of recursion and in converting from infix notation to postfix notation.
4. Examine the role of queues in computer simulation.

8.1 Stacks

A ***stack*** is a finite sequence of elements in which the only element that can be removed is the element that was most recently inserted. That element is referred to as the ***top*** element on the stack.

For example, a tray-holder in a cafeteria holds a stack of trays. Insertions and deletions are made only at the top. To put it another way, the tray that was most recently put on the holder will be the next one to be removed. This defining property of stacks is sometimes called "Last In, First Out," or LIFO. In keeping with this view, an insertion is referred to as a ***push***, and a removal as a ***pop***. For the sake of alliteration, a retrieval of the top element is referred to as a ***peek***.

Figure 8.1a shows a stack with three elements and Figures 8.1b, c and d show the effect of two pops and then a push.

In Section 8.1.1, we define the `Stack` class, and note its assets and liabilities.

8.1.1 The `Stack` Class

The Java Collection Framework's `Stack` class is a ***legacy*** class: It was created even before there was a Java Collections Framework. It is a subclass of another legacy class, `Vector`, which was retrofitted to implement the `List` interface. In fact, the `Vector` class is virtually equivalent to the `ArrayList` class that we studied in Chapter 6.

The `Stack` class's essential methods—`push`, `pop`, and `peek`—were easily defined once the developers decided whether the top of the stack should be at the front or back of the underlying array. Which

17			
13	13		21
28	28	28	28
(a)	(b) pop	(c) pop	(d) push 21

FIGURE 8.1 A stack through several stages of pops and pushes: 17 and 13 are popped and then 21 is pushed. In each figure, the highest element is the top element

do you think would be faster for the `pop` and `push` methods? For removing the front element in an array, worstTime(n) and averageTime(n) are both linear in n, whereas removing the last element in an array takes only constant time in both the worst and average cases. For inserting at the front of an array, worstTime(n) and averageTime(n) are both linear in n, whereas inserting at the back of an array takes only constant time on average, but linear-in-n time in the worst case (when the array is full). So it is clearly faster for the top element to be at the back of the underlying array. See Figure 8.2.

Here are the `Stack` class's heading and method specifications for the only constructor and the `push`, `pop` and `peek` methods[1]:

```
public class Stack<E> extends Vector<E>

 /**
  * Creates an empty Stack.
  */
  public Stack()
 /**

  * Pushes an element onto the top of this stack.
  * The worstTime(n) is O(n) and averageTime(n) is constant.
  *
  * @param  element:   the element to be pushed onto this stack.
  * @return  the element argument.
  */
  public E push (E element)

 /**
  * Removes the element at the top of this stack and returns that
  * element.
  * The worstTime(n) and averageTime(n) are constant.
  *
  * @return  the element at the top of this stack.
  * @throws EmptyStackException  if this stack is empty.
  */
  public E pop()

 /**
  * Returns the element at the top of this stack without removing it
  * from the stack.
```

[1]Strictly speaking, the `pop` and `peek` method headings include the modifier **`synchronized`**: a keyword related to concurrent programming, which is beyond the scope of this book and irrelevant to our discussion. For more details on synchronization and concurrent programming, see the Java Tutorials at java.sun.com.

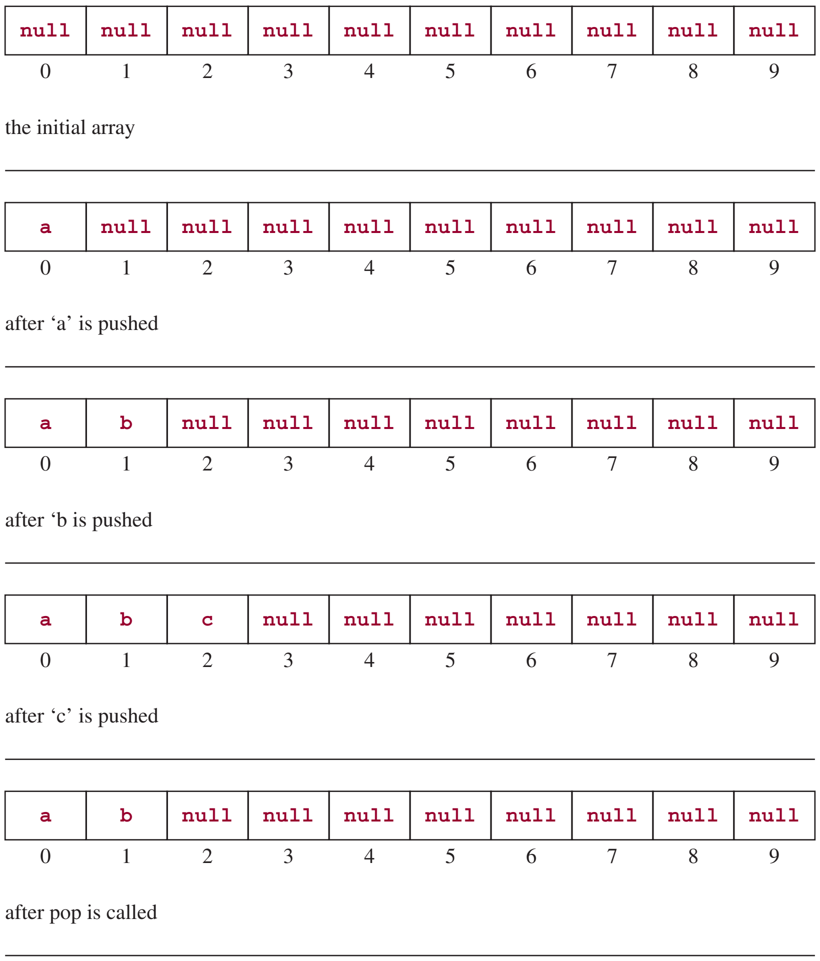

null null null null null null null null null null
0 1 2 3 4 5 6 7 8 9
the initial array
a null null null null null null null null null
0 1 2 3 4 5 6 7 8 9
after 'a' is pushed
a b null null null null null null null null
0 1 2 3 4 5 6 7 8 9
after 'b is pushed
a b c null null null null null null null
0 1 2 3 4 5 6 7 8 9
after 'c' is pushed
a b null null null null null null null null
0 1 2 3 4 5 6 7 8 9
after pop is called

FIGURE 8.2 The effect of three pushes and a pop on the underlying array of an instance of the Java Collection Framework's `Stack` class. The `Stack` class's constructor automatically creates an array of length 10

```
    * The worstTime(n) and averageTime(n) are constant.
    *
    * @return  the element at the top of this stack.
    * @throws  EmptyStackException  if this stack is empty.
    */
    public E peek()
```

The following program utilizes each of the above methods, as well as several `List` methods you saw in Chapters 6 and 7.

```
import java.util.*;

public class StackExample
{
  public static void main (String[] args)
  {
     new StackExample().run();
  } // method main

  public void run()
  {
    Stack<Character> myStack = new Stack<Character>();

    System.out.println ("push " + myStack.push ('a'));
    System.out.println ("push " + myStack.push ('b'));
    System.out.println ("push " + myStack.push ('c'));
    System.out.println ("pop " + myStack.pop());
    System.out.println ("top = " + myStack.peek());
    System.out.println ("push " + myStack.push ('d'));
    System.out.println ("The stack now has " + myStack.size() + " elements.");

    System.out.println ("\nHere are the contents of the stack, from top to bottom:");
    for (int i = myStack.size() - 1; i >= 0; i-)
      System.out.println (myStack.get (i));

    System.out.println ("\nHere are the contents of the stack, starting from index 0:");
    for (Character c : myStack)
      System.out.println (c);

    System.out.println (
         "\nHere are the contents of the stack, from top to bottom, during destruction:");
    while (!myStack.isEmpty())
       System.out.println (myStack.pop());
    System.out.println ("The stack now has " + myStack.size() + " elements.");
  } // method run

}  // class StackExample
```

And here is the corresponding output:

```
push a
push b
push c
pop c
top = b
push d
The stack now has 3 elements.

Here are the contents of the stack, from top to bottom:
d
b
a

Here are the contents of the stack, starting at index 0:
a
b
d
```

```
Here are the contents of the stack, from top to bottom, during destruction:
d
b
a
The stack now has 0 elements.
```

The first iteration in the above program uses a list index and the `get` method to access each element, starting at the top of the stack. The second iteration illustrates a curious feature of the `Stack` class: The standard iteration—with an enhanced **`for`** statement—accesses the elements starting from the bottom of the stack! The same bottom-up access would apply if you called

```
System.out.println (myStack);
```

The third iteration represents destructive access: the top element is repeatedly popped and printed until the stack is empty.

8.1.2 A Fatal Flaw?

The `Stack` class is part of the Java Collections Framework, so that class is always available to you whenever you use the Java language. And the fact that the `Stack` class implements the `List` interface can occasionally come in handy. For example, as in the above program, you can access each of the elements without removing them, and each access (or modification) takes constant time. Unfortunately, you can also remove an element other than the element that was most recently inserted, and *this violates the definition of a stack*. For example, you can have the following:

```
Stack<String> badStack = new Stack<String>();

badStack.push ("Larry");
badStack.push ("Curly");
badStack.push ("Moe");

badStack.remove (1);  // removes Curly
```

We will let the convenience of the `Stack` class override our disapproval of the just noted defect. But if you regard the violation as a fatal flaw, you have (at least) two options, one utilizing inheritance and one utilizing aggregation. For the first option, you can undertake Programming Project 8.4, and create a `PureStack` class that extends `Stack`, but throws `UnsupportedOperationException` for any attempt to remove an element that is not at the top of the stack. And the iterator goes from top to bottom, instead of from bottom to top. This is, essentially, the approach used for the `Stack` class in C#, a member of Microsoft's. NET family of languages.

Another option for a `PureStack` class is to allow access, modification or removal only of the most recently inserted element. This stringency would allow only a few methods: a default constructor, a copy constructor (so you can non-destructively access a *copy* of a stack, instead of the original), `push`, `pop`, `peek`, `size`, and `isEmpty`. This is the idea behind the `stack` class in C++, a widely used language. A straightforward way to create such a `PureStack` class is with aggregation: the only field is a list whose type is `ArrayList` or `LinkedList`. For example, the definition of the `pop` method—in either implementation—is as follows:

```
public E pop()
{
      return list.remove (list.size() - 1);
} // method pop
```

In fact, all the definitions—in either implementation—are one-liners. See Programming Exercise 8.1 for an opportunity to develop the `LinkedList` implementation, and Programming Exercise 8.2 for the `ArrayList` implementation.

Now let's look at a couple of important applications.

8.1.3 Stack Application 1: How Compilers Implement Recursion

We saw several examples of recursive methods in Chapter 5. In adherence to the Principle of Abstraction, we focused on what recursion did and ignored the question of how recursion is implemented by a compiler or interpreter. It turns out that the visual aids—execution frames—are closely related to this implementation. We now outline how a stack is utilized in implementing recursion and the time-space implications for methods, especially recursive methods.

Each time a method call occurs, whether it is a recursive method or not, the return address in the calling method is saved. This information is saved so the computer will know where to resume execution in the calling method after the execution of the called method has been completed. Also, the values of the called method's local variables must be saved. This is done to prevent the destruction of that information in the event that the method is—directly or indirectly—recursive. As we noted in Chapter 5, the compiler saves this method information for all methods, not just the recursive ones (this relieves the compiler of the burden of determining if a given method is indirectly recursive). This information is collectively referred to as an ***activation record*** or ***stack frame***.

Each activation record includes:

a. the return address, that is, the address of the statement that will be executed when the call has been completed;

b. the value of each argument: a copy of the corresponding argument is made (if the type of the argument is reference-to-object, the reference is copied);

c. the values of the called method's other local variables;

Part of main memory—the ***stack***—is allocated for a run-time stack onto which an activation record is pushed when a method is called and from which an activation record is popped when the execution of the method has been completed. During the execution of that method, the top activation record contains the current state of the method. For methods that return a value, that value—either a primitive value or a reference—is pushed onto the top of the stack just before there is a return to the calling method.

How does an activation record compare to an execution frame? Both contain values, but an activation record has no code. Of course, the entire method, in bytecode, is available at run-time. So there is no need for a checkmark to indicate the method that is currently executing.

For a simple example of activation records and the run-time stack, let's trace the execution of a `getBinary` method that invokes the `getBin` method from Chapter 5. The return addresses have been commented as RA1 and RA2.

```
/**
 *
 * Determines the binary equivalent of a non-negative integer.  The worstTime(n)
 * is O(log n).
 *
 * @param n the non-negative integer, in decimal notation.
 *
 * @return a String representation of the binary equivalent of n.
 *
 * @throws  IllegalArgumentException if n is negative.
 */
 public static String getBinary (int n)
 {
       if (n < 0)
               throw new IllegalArgumentException();
                return getBin (n);  // RA1
 } // method getBinary

 public static String getBin (int n)
 {
        if (n <= 1)
                 return Integer.toString (n);
        return getBin (n / 2) + Integer.toString (n % 2); // RA2
 } // method getBin
```

The `getBin` method has the formal parameter `n` as its only local variable, and so each activation record will have two components:

a. the return address;

b. the value of the formal parameter `n`.

Also, because the `getBin` method returns a `String` reference, a copy of that `String` reference is pushed onto the stack just before a return is made. For simplicity, we will pretend that the `String` object itself is pushed.

Assume that the value of `n` is 6. When `getBin` is called from the `getBinary` method, an activation record is created and pushed onto the stack, as shown in Figure 8.3.

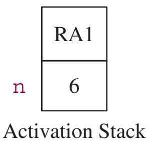


FIGURE 8.3 The activation stack just prior to `getBin`'s first activation. RA1 is the return address

Since $n > 1$, `getBin` is called recursively with 3 (that is, 6/2) as the value of the argument. A second activation record is created and pushed onto the stack. See Figure 8.4.

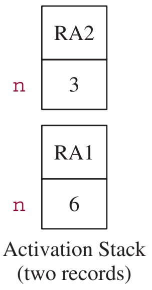


FIGURE 8.4 The activation stack just prior to the second activation of `getBin`

Since `n` is still greater than 1, `getBin` is called again, this time with 1 (that is, 3/2) as the value of the argument. A third activation record is created and pushed. See Figure 8.5.

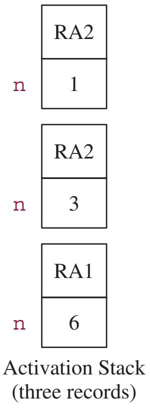


FIGURE 8.5 The activation stack just prior to the third activation of `getBin`

Since $n \leq 1$, the top activation record is popped, the `String` "1" is pushed onto the top of the stack and a return is made to the address RA2. The resulting stack is shown in Figure 8.6.

The concatenation at RA2 in `getBin` is executed, yielding the `String "1" + Integer.toString (3 % 2), namely, "11"`. The top activation record on the stack is popped, the `String` "11" is pushed, and another return to RA2 is made, as shown in Figure 8.7.

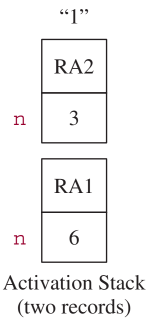


FIGURE 8.6 The activation stack just after the completion of the third activation of `getBin`

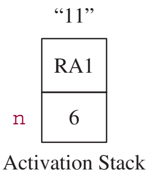


FIGURE 8.7 The activation stack just after the completion of the second activation of `getBin`

The concatenation at RA2 is `"11" + Integer.toString (6 % 2)`, and the value of that `String` object is "110". The stack is popped once more, leaving it empty, and "110", the binary equivalent of 6, is pushed. Then a return to RA1—at the end of the `getBinary` method—is made.

The above discussion should give you a general idea of how recursion is implemented by the compiler. The same stack is used for all method calls. And so the size of each activation record must be saved with each method call. Then the correct number of bytes can be popped. For the sake of simplicity, we have ignored the size of each activation record in the above discussion.

The compiler must generate code for the creation and maintenance, at run time, of the activation stack. Each time a call is made, the entire local environment must be saved. In most cases, this overhead pales to insignificance relative to the cost in programmer time of converting to an iterative version, but this conversion is always feasible.

On those rare occasions when you must convert a recursive method to an iterative method, one option is to simulate the recursive method with an iterative method that creates and maintains its own stack of information to be saved. For example, Project 8.3 requires an iterative version of the (recursive) `tryToReachGoal` method in the backtracking application from Chapter 5. When you create your own stack, you get to decide what is saved. For example, if the recursive version of the method contains a single recursive call, you need not save the return address. Here is an iterative, stack-based version of the `getBinary` method (see Programming Exercise 5.1 for an iterative version that is not stack-based, and see Programming Exercise 8.3 for a related exercise).

```
/**
 *
 * Determines the binary equivalent of a non-negative integer.  The worstTime(n)
 * is O(log n).
 *
 * @param n the non-negative integer, in decimal notation.
 *
 * @return a String representation of the binary equivalent of n.
 *
 * @throws  IllegalArgumentException if n is negative.
 */
 public static String getBinary (int n)
 {
       Stack<Integer> myStack = new Stack<Integer>();

       String binary = new String();

       if (n < 0)
               throw new IllegalArgumentException( );
       myStack.push (n % 2);
       while (n > 1)
       {
              n /= 2;
              myStack.push (n % 2);
       } // pushing
       while (!myStack.isEmpty())
              binary += myStack.pop();
       return binary;
 } // method getBinary
```

What is most important is that you not overlook the cost, in terms of programmer time, of making the conversion from a recursive method to an iterative method. Some recursive methods, such as the `factorial` method, can easily be converted to iterative methods. Sometimes the conversion is nontrivial, such as for the `move` and `tryToReachGoal` methods of Chapter 5 and the `permute` method of Lab 9. Furthermore, the iterative version may well lack the simple elegance of the recursive version, and this may complicate maintenance.

You certainly should continue to design recursive methods when circumstances warrant. That is, whenever the problem is such that complex instances of the problem can be reduced to simpler instances of the same form, and the simplest instance(s) can be solved directly. The above discussion on the activation stack enables you to make better-informed tradeoff decisions.

8.1.4 Stack Application 2: Converting from Infix to Postfix

In Section 8.1.3, we saw how a compiler or interpreter could implement recursion. In this section we present another "internal" application: the translation of arithmetic expressions from infix notation into postfix notation. This can be one of the key tasks performed by a compiler as it creates machine-level code, or by an interpreter as it evaluates an arithmetic expression.

In ***infix*** notation, a binary operator is placed between its operands. For example, Figure 8.8 shows several arithmetic expressions in infix notation.

```
a + b
b - c * d
(b - c) * d
a - c - h / b * c
a - (c - h) / (b * c)
```

FIGURE 8.8 Several arithmetic expressions in infix notation

For the sake of simplicity, we initially restrict our attention to expressions with single-letter identifiers, parentheses and the binary operators $+$, $-$, $*$, and /.

The usual rules of arithmetic apply:

1. Operations are normally carried out from left to right. For example, if we have

   ```
   a + b - c
   ```

   then the addition will be performed first.

2. If the current operator is $+$ or $-$ and the next operator is $*$ or /, then the next operator is applied before the current operator. For example, if we have

   ```
   b + c * d
   ```

   then the multiplication will be carried out before the addition. For

   ```
   a - b + c * d
   ```

   the subtraction is performed first, then the multiplication and, finally, the addition.

   We can interpret this rule as saying that multiplication and division have "higher precedence" than addition and subtraction.

3. Parentheses may be used to alter the order indicated by rules 1 and 2. For example, if we have

   ```
   a - (b + c)
   ```

   then the addition is performed first. Similarly, with

   ```
   (a - b) * c
   ```

   the subtraction is performed first.

Figure 8.9 shows the order of evaluation for the last two expressions in Figure 8.8.

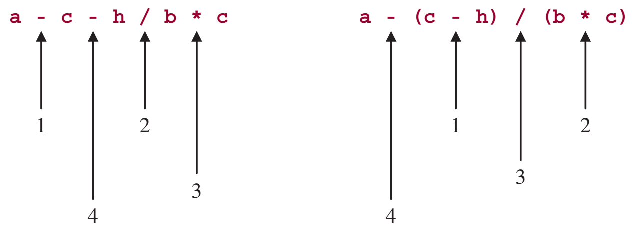


FIGURE 8.9 The order of evaluation for the last two expressions in Figure 8.8

The first widely used programming language was FORTRAN (from FORmula TRANslator), so named because its compiler could translate arithmetic formulas into machine-level code. In early (pre-1960) compilers, the translation was performed directly. But direct translation is awkward because the machine-level code for an operator cannot be generated until both of its operands are known. This requirement leads to difficulties when either operand is a parenthesized subexpression.

8.1.4.1 Postfix Notation

Modern compilers do not translate arithmetic expressions directly into machine-level code. Instead, they can utilize an intermediate form known as ***postfix notation***. In postfix notation, an operator is placed immediately after its operands. For example, given the infix expression `a + b`, the postfix form is `a b +`. For `a + b * c`, the postfix form is `a b c * +` because the operands for `+` are `a` and the product of `b` and `c`. For `(a + b) * c`, the postfix form is `a b + c *`. Since an operator immediately follows its operands in postfix notation, parentheses are unnecessary and therefore not used. Figure 8.10 shows several arithmetic expressions in both infix and postfix notation.

Infix	Postfix
`a - b + c * d`	`a b - c d * +`
`a + c - h/b * r`	`a c + h b/r * -`
`a + (c - h)/(b * r)`	`a c h - b r * /+`

FIGURE 8.10 Several arithmetic expressions in both infix and postfix notation

How can we convert an arithmetic expression from infix notation into postfix notation? Let's view the infix notation as a string of characters and try to produce the corresponding postfix string. The identifiers in the postfix string will be in the same order as they are in the infix string, so each identifier can be appended to the postfix string as soon as it is encountered. But in postfix notation, operators must be placed after their operands. So when an operator is encountered in the infix string, it must be saved somewhere temporarily.

For example, suppose we want to translate the infix string

```
a - b + c * d
```

into postfix notation. (The blanks are for readability only—they are not, for now, considered part of the infix expression.) We would go through the following steps:

```
'a' is appended to postfix, which is now the string "a"

'-' is stored temporarily

'b' is appended to postfix, which is now the string "ab"
```

When `'+'` is encountered, we note that since it has the same precedence as `'-'`, the subtraction should be performed first by the left-to-right rule (Rule 1, above). So the `'-'` is appended to the postfix string, which is now `"ab-"` and `'+'` is saved temporarily. Then `'c'` is appended to postfix, which now is `"ab-c"`.

The next operator, `'*'`, must also be saved somewhere temporarily because one of its operands (namely `'d'`) has not yet been appended to postfix. But `'*'` should be retrieved and appended to postfix *before* `'+'` since multiplication has higher precedence than addition.

When `'d'` is appended to postfix, the postfix string is `"ab-cd"`. Then `'*'` is appended, making the postfix string `"ab-cd*"`. Finally, `'+'` is appended to postfix, and the final postfix representation is

```
"ab-cd*+"
```

The temporary storage facility referred to in the previous paragraph is handled conveniently with a stack to hold operators. The rules governing this `operatorStack` are:

R1. Initially, `operatorStack` is empty.

R2. For each operator in the infix string,
Loop until the operator has been pushed onto `operatorStack`:

```
If operatorStack is empty or the operator has greater precedence than
   the operator on the top of operatorStack then
       Push the operator onto operatorStack.

   else

Pop operatorStack and append that popped operator to the
   postfix string.
```

R3. Once the end of the input string is encountered,
Loop until operatorStack is empty:

```
Pop operatorStack and append that popped operator to the postfix
    string.
```

For example, Figure 8.11 shows the history of the operator stack during the conversion of

```
a + c - h / b * r
```

to its postfix equivalent.

Infix Expression: a + c - h/b * r

infix	**operatorStack**	**postfix**
a	(empty)	a
+	+	a
c	+	ac
-	-	ac+
h	-	ac+h
/	-/	ac+h
b	-/	ac+hb
*	-*	ac+hb/
r	-*	ac+hb/r
	-	ac+hb/r*
	(empty)	ac+hb/r*-

FIGURE 8.11 The conversion of `a + c - h/b * r` to postfix notation. At each stage, the top of `operatorStack` is shown as the *rightmost* element

How are parentheses handled? When a left parenthesis is encountered in the infix string, it is immediately pushed onto `operatorStack`, but its precedence is defined to be ***lower*** than the precedence of any binary operator. When a right parenthesis is encountered in the infix string, `operatorStack` is repeatedly popped, and the popped element appended to the postfix string, until the operator on the top of the stack is a left parenthesis. Then that left parenthesis is popped but not appended to postfix, and the scan of the infix string is resumed. This process ensures that parentheses will never appear in postfix notation.

For example, when we translate `a * (b + c)` into postfix, the operators `'*'`, `'('`, and `'+'` would be pushed and then all would be popped (last-in, first-out) when the right parenthesis is encountered. The postfix form is

```
a b c + *
```

For a more complex example, Figure 8.12 (next page) illustrates the conversion of

```
x - (y * a / b - (z + d * e) + c) / f
```

into postfix notation.

8.1.4.2 Transition Matrix

At each step in the conversion process, we know what action to take as long as we know the current character in the infix string and the top character on the operator stack. We can therefore create a matrix to summarize the conversion. The row indexes represent the possible values of the current infix character. The column indexes represent the possible values of the top character on the operator stack. The matrix entries represent the action to be taken. Such a matrix is called a ***transition matrix*** because it directs the transition of information from one form to another. Figure 8.13 (page 343) shows the transition matrix for converting a simple expression from infix notation to postfix notation.

The graphical nature of the transition matrix in Figure 8.13 enables us to see at a glance how to convert simple expressions from infix to postfix. We could now design and implement a program to do just that. The program may well incorporate the transition matrix in Figure 8.13 for the sake of extensibility.

Infix Expression: x - (y ∗ a / b - (z + d ∗ e) + c) / f

Infix	***operatorStack***	**postfix**
x	(empty)	x
-	-	x
(	-(	x
y	-(	xy
*	-(∗	xy
a	-(∗	xya
/	-(/	xya∗
b	-(/	xya∗b
-	-(-	xya∗b/
(	-(-(	xya∗b/
z	-(-(	xya∗b/z
+	-(-(+	xya∗b/z
d	-(-(+	xya∗b/zd
*	-(-(+∗	xya∗b/zd
e	-(-(+∗	xya∗b/zde
)	-(-(+	xya∗b/zde∗
	-(-(	xya∗b/zde∗+
	-(-	xya∗b/zde∗+
+	-(+	xya∗b/zde∗+-
c	-(+	xya∗b/zde*+-c
)	-(	xya*b/zde*+-c+
	-	xya*b/zde*+-c+
/	-/	xya*b/zde*+-c+
f	-/	xya*b/zde*+-c+f
	-	xya*b/zde*+-c+f/
	(empty)	xya*b/zde*+-c+f/-

Postfix Expression: x y a * b / z d e * + - c + f / -

FIGURE 8.12 The conversion of `x - (y * a/b - (z + d * e) + c)/f` from infix to postfix. At each stage, the top of `operatorStack` is shown as the *rightmost* element

More complex expressions can be accommodated by expanding the matrix. For the conversion, there would be a **`switch`** statement with one case for each matrix entry.

8.1.4.3 Tokens

A program that utilized a transition matrix would probably not work with the characters themselves because there are too many possible (legal) values for each character. For example, a transition matrix that used a row for each legal infix character would need 52 rows just for an identifier. And if we changed the specifications to allow multi-character identifiers, we would need millions of rows!

Instead, the legal characters would usually be grouped together into "tokens." A ***token*** is the smallest meaningful unit in a program. Each token has two parts: a generic part that holds its category and a specific

ACTION TAKEN		Top character on operator stack: `(`	`+,-`	`*,/`	empty
Infix character	Identifier	Append to postfix	Append to postfix	Append to postfix	Append to postfix
	`)`	Pop; pitch `'('`	Pop to postfix	Pop to postfix	Error
	`(`	Push	Push	Push	Push
	`+,-`	Push	Pop to postfix	Pop to postfix	Push
	`*,/`	Push	Push	Pop to postfix	Push
	empty	Error	Pop to postfix	Pop to postfix	Done

FIGURE 8.13 The transition matrix for converting simple expressions from infix notation to postfix notation

part that enables us to recapture the character(s) tokenized. For converting simple infix expressions to postfix, the token categories would be: `identifier`, `rightPar`, `leftPar`, `addOp` (for `'+'` and `'-'`), `multOp` (for `'*'` and `'/'`), and empty (for a dummy value). The specific part would contain the index, in the infix string, of the character tokenized. For example, given the infix string

```
(first + last) / sum
```

to tokenize `"last"`, we would set its category to `identifier` and its index to 9.

The structure of tokens varies widely among compilers. Typically, the specific part of a variable identifier's token contains an address into a table, called a ***symbol table***. At that address would be stored the identifier, an indication that it is a variable identifier, its type, initial value, the block it is declared in and other information helpful to the compiler. There is a symbol table in the project of Lab 15, and the creation of a symbol table is the subject of an application in Chapter 14.

In Lab 15, a complete infix-to-postfix project is developed, with tokens and massive input-editing.

You are now prepared to do Lab 15: Converting from Infix to Postfix

8.1.5 Prefix Notation

In Section 8.1.4 we described how to convert an infix expression into postfix notation. Another possibility is to convert from infix into ***prefix notation***, in which each operator immediately precedes its operands[2]. Figure 8.14 shows several expressions in both infix and prefix notation.

[2]Prefix notation was invented by Jan Lukasiewicz, a Polish logician. It is sometimes referred to as *Polish Notation*. Postfix notation is then called *Reverse Polish Notation*.

Infix	Prefix
`a - b`	`- a b`
`a - b * c`	`- a * b c`
`(a - b) * c`	`* - a b c`
`a - b + c * d`	`+ - a b * c d`
`a + c - h / b * d`	`- + a c * / h b d`
`a + (c - h) / (b * d)`	`+ a / - c h * b d`

FIGURE 8.14 Several arithmetic expressions in both infix and prefix notation

How can we convert an arithmetic expression from infix to prefix? As in infix-to-postfix, we will need to save each operator until both of its operands have been obtained. But we cannot simply append each identifier to the prefix string as soon as it is encountered. Instead, we will need to save each identifier, in fact, each operand, until its operator has been obtained.

The saving of operands and operators is easily accomplished with the help of two stacks, `operandStack` and `operatorStack`. The precedence rules for `operatorStack` are exactly the same as we saw in converting from infix to postfix. Initially, both stacks are empty. When an identifier is encountered in the infix string, that identifier is pushed onto `operandStack`. When an operator is encountered, it is pushed onto `operatorStack` if that stack is empty. Otherwise, one of the following cases applies:

1. If the operator is a left parenthesis, push it onto `operatorStack` (but left parenthesis has lowest precedence).
2. If the operator has higher precedence than the top operator on `operatorStack`, push the operator onto `operatorStack`.
3. If the operator's precedence is equal to or lower than the precedence of the top operator on `operatorStack`, pop the top operator, `opt1`, from `operatorStack` and pop the top two operands, `opnd1` and `opnd2`, from `operandStack`. Concatenate (join together) `opt1`, `opnd2`, and `opnd1` and push the result string onto `operandStack`. Note that `opnd2` is in front of `opnd1` in the result string because `opnd2` was encountered in the infix string—and pushed onto `operandStack`—before `opnd1`.
4. If the operator is a right parenthesis, treat it as having lower priority than $+$, $-$, $*$, and /. Then Case 3 will apply until a left parenthesis is the top operator on `operatorStack`. Pop that left parenthesis.

The above process continues until we reach the end of the infix expression. We then repeat the following actions from case 3 (above) until *operatorStack* is empty:

Pop opt1 from operatorStack.
Pop opnd1 and opnd2 from operandStack.
Concatenate opt1, opnd2 and opnd1 together and push the result onto operandStack.

When `operatorStack` is finally empty, the top (and only) operand on `operandStack` will be the prefix string corresponding to the original infix expression.

For example, if we start with

```
a + b ∗ c
```

then the history of the two stacks would be as follows:

```
                a                            a
1.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack

                +                            a                           +
2.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack

                                             b
                b                            a                           +
3.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack

                                             b                           *
                *                            a                           +
4.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack

                                             c
                                             b                           *
                c                            a                           +
5.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack

                                            *bc
                                             a                           +
6.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack
                                           +a*bc
7.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack
```

The prefix string corresponding to the original string is

```
+ a * b c
```

For a more complex example, suppose the infix string is

```
a + (c - h) / (b * d)
```

Then the elements on the two stacks during the processing of the first right parenthesis would be as follows:

```
                                             h                           -
                                             c                           (
                )                            a                           +
1.   ---------------------------  ---------------------------  ---------------------------
              infix                   operandStack                 operatorStack
```

```
                             -ch                         (
                              a                          +
2.                   ----------------------    ----------------------
                         operandStack               operatorStack

                             -ch
                              a                          +
3.                   ----------------------    ----------------------
                         operandStack               operatorStack
```

During the processing of the second right parenthesis in the infix string, we would have

```
                                        d                *
                                        b                (
                                       -ch               /
                 )                      a                +
1.   ----------------------    ----------------------    ----------------------
             infix                 operandStack               operatorStack

                                       *bd               (
                                       -ch               /
                                        a                +
2.                             ----------------------    ----------------------
                                   operandStack               operatorStack
```

The end of the infix expression has been reached, so `operatorStack` is repeatedly popped.

```
                             *bd
                             -ch                         /
                              a                          +
3.                   ----------------------    ----------------------
                         operandStack               operatorStack

                           /-ch*bd
                              a                          +
4.                   ----------------------    ----------------------
                         operandStack               operatorStack

                           +a/-ch*bd
5.                   ----------------------    ----------------------
                         operandStack               operatorStack
```

The prefix string is

```
+ a / - c h * b d
```

8.2 Queues

A ***queue*** is a finite sequence of elements in which:

1. insertion is allowed only at the back of the sequence;
2. removal is allowed only at the front of the sequence.

The term ***enqueue*** (or ***add***) is used for inserting an element at the back of a queue, ***dequeue*** (or ***remove***) for removing the first element from a queue, and ***front*** (or ***element***) for the first element in a queue. A queue imposes a chronological order on its elements: the first element enqueued, at the back, will eventually be the first element to be dequeued, from the front. The second element enqueued will be the second element to be dequeued, and so on. This defining property of queues is sometimes referred to as "First Come, First Served," "First In, First Out," or simply FIFO.

Figure 8.15 shows a queue through several stages of insertions and deletions.

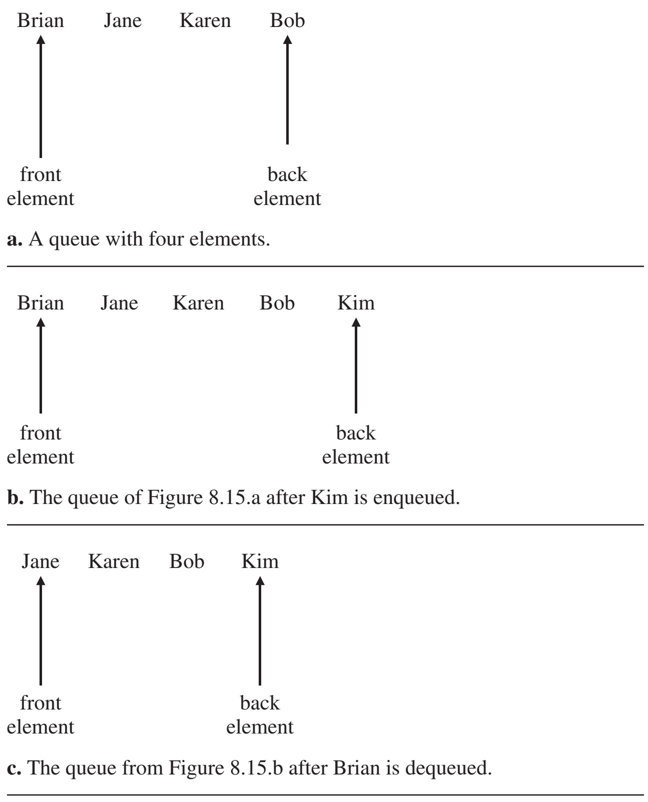
The examples of queues are widespread:

cars in line at a drive-up window

fans waiting to buy tickets to a ball game

customers in a check-out line at a supermarket

airplanes waiting to take off from an airport.



a. A queue with four elements.

b. The queue of Figure 8.15.a after Kim is enqueued.

c. The queue from Figure 8.15.b after Brian is dequeued.

FIGURE 8.15 A queue through several stages of insertions and deletions

We could continue giving queue examples almost indefinitely. Later in this chapter we will develop an application of queues in the field of computer simulation.

Section 8.2.1 presents the `Queue` interface—part of the Java Collections Framework.

8.2.1 The Queue Interface

The `Queue` interface, with type parameter `E`, extends the `Collection` interface by specifying `remove()` and `element()` methods:

```
public interface Queue<E> extends Collection<E>
{
      /**
       * Retrieves and removes the head of this queue.
       *
       * @return the head of this queue.
       * @throws NoSuchElementException if this queue is empty.
       */
       E remove();

      /**
       * Retrieves, but does not remove, the head of this queue.
       *
       * @return the head of this queue.
       * @throws NoSuchElementException if this queue is empty.
       */
       E element();

} // interface Queue
```

Also, the `Queue` interface inherits the following `add` method from the `Collection` interface:

```
/**
 * Ensures that this collection contains the specified element.
 *
 * @param element:  element whose presence in this collection is to be ensured.
 * @return true if this collection changed as a result of the call; otherwise, false.
 *
 * @throws ClassCastException class of the specified element prevents it
 *     from being added to this collection.
 * @throws NullPointerException if the specified element is null and this
 *     collection does not support null elements.
 *
 */
 boolean add (E element);
```

So the `Queue` interface includes the essential methods of the queue data type[3].

[3]The `Queue` interface also has a `poll()` method that is equivalent to the `remove()` method except that `poll()` returns **`null`** if the queue is empty, a `peek()` method that is equivalent to the `element()` method except that `peek()` returns **`null`** if the queue is empty, and an `offer (E element)` method that is equivalent to the `add (E element)` method except that `offer (E element)` is better suited if the queue imposes insertion restrictions and a full queue is commonplace.

8.2.2 Implementations of the Queue Interface

The LinkedList class, from Chapter 7, implements the Queue interface, with the front of the queue at index 0. Here is the code to create the queue in Figure 8.15:

```
Queue<String> queue = new LinkedList<String>();

// Figure 8.15.a.
queue.add ("Brian");
queue.add ("Jane");
queue.add ("Karen");
queue.add ("Bob");

// Figure 8.15.b.
queue.add ("Kim");

// Figure 8.15.c.
queue.remove();
```

For the three key methods—add (E element), remove(), and element()—worstTime(n) is constant. That is because in a LinkedList object, insertion at the back and removal or access of the front element take only constant time. And we have the same flaw as we had for the Stack class earlier in this chapter: There are LinkedList methods—such as remove (**int** index) and add (**int** index, E element)—that violate the definition of a queue. Whenever a LinkedList object implements the Queue interface, we declare the object with the Queue interface as in the previous example. This is a heads-up to users that insertions should occur only at the back and deletions only at the front.

You can overcome this defective implementation of the Queue interface by extending the LinkedList class to a PureQueue class that throws UnsupportedOperationException for any of the offending methods. Or you can take an austere approach and create a PureQueue class by aggregation: The only field will be a LinkedList (why not ArrayList ?) object, list, and the only methods will be a default constructor, a copy constructor, isEmpty(), size(), add (E element), remove(), and element(). For example, the definition of the remove() method is

```
/**
 * Retrieves and removes the head of this queue.
 * The worstTime(n) is constant and averageTime(n) is constant.
 *
 * @return the head of this queue.
 * @throws NoSuchElementException if this queue is empty.
 */
 public E remove()
 {
       return list.removeFirst();
 } // method remove()
```

The definition of each of the other six methods is also a one-liner. Observe that if list were an ArrayList object, worstTime(n) and averageTime(n) for this method would be linear in n.

Now that we have seen several possible implementations of the Queue interface, we turn our attention to applications.

8.2.3 Computer Simulation

A ***system*** is a collection of interacting parts. We are often interested in studying the behavior of a system, for example, an economic system, a political system, an ecological system or even a computer system. Because systems are usually complicated, we may utilize a model to make our task manageable. A ***model***, that is, a simplification of a system, is designed so that we may study the behavior of the system.

A ***physical model*** is similar to the system it represents, except in scale or intensity. For example, we might create a physical model of tidal movements in the Chesapeake Bay or of a proposed shopping center. War games, spring training, and scrimmages are also examples of physical models. Unfortunately, some systems cannot be modeled physically with currently available technology—there is, as yet, no physical substance that could be expected to behave like the weather. Often, as with pilot training, a physical model may be too expensive, too dangerous, or simply inconvenient.

Sometimes we may be able to represent the system with a **mathematical model**: a set of assumptions, variables, constants, and equations. Often, a mathematical model is easier to develop than a physical model. For example, such equations as distance = rate ∗ time and the formula for the Pythagorean Theorem can be solved analytically in a short amount of time. But sometimes, this is not the case. For example, most differential equations cannot be solved analytically, and an economic model with thousands of equations cannot be solved by hand with any hope of correctness.

In such cases, the mathematical model is usually represented by a computer program. Computer models are essential in complex systems such as weather forecasting, space flight, and urban planning. The use of computer models is called ***computer simulation***. There are several advantages to working with a computer model rather than the original system:

1. **Safety**. Flight simulators can assail pilot trainees with a welter of dangerous situations such as hurricanes and hijackings, but no one gets hurt.[4]

2. **Economy**. Simulation games in business-policy courses enable students to run a hypothetical company in competition with other students. If the company goes "belly up," the only recrimination is a lower grade for the students.

3. **Speed**. The computer usually makes predictions soon enough for you to act on them. This feature is essential in almost every simulation, from the stock market to national defense to weather forecasting.

4. **Flexibility**. If the results you get do not conform to the system you are studying, you can change your model. This is an example of ***feedback***: a process in which the factors that produce a result are themselves affected by that result. After the computer model is developed and run, the output is interpreted to see what it says about the original system. If the results are invalid—that is, if the results do not correspond to the known behavior of the original system—the computer model is changed. See Figure 8.16.

The above benefits are so compelling that computer simulation has become a standard tool in the study of complex systems. This is not to say that computer simulation is a panacea for all systems problems. The simplification required to model a system necessarily introduces a disparity between the model and the system. For example, suppose you had developed a computer simulation of the earth's ecosystem

[4]According to legend, a trainee once panicked because one of his simulated engines failed during a simulated blizzard. He "bailed out" of his simulated cockpit and broke his ankle when he hit the unsimulated floor.

$$\begin{tikzcd}
\text{System} \arrow[r, "\text{develop}"] & \text{Computer Model} \arrow[d, "\text{run}"] \\
\text{Interpretation} \arrow[u, "\text{validate}"] & \text{Output} \arrow[l, "\text{interpret}"]
\end{tikzcd}$$

FIGURE 8.16 Feedback in computer simulation

30 years ago. You probably would have disregarded the effects of aerosol sprays and refrigerants that released chlorofluorocarbons. Many scientists now suspect that chlorofluorocarbons may have a significant impact on the ozone layer and thus on all land organisms.

Another disadvantage of computer simulation is that its results are often interpreted as predictions, and prediction is always a risky business. For this reason, a disclaimer such as the following usually precedes the results of a computer simulation: "If the relationships among the variables are as described and if the initial conditions are as described, then the consequences will probably be as follows . . . "

8.2.4 Queue Application: A Simulated Car Wash

Queues are employed in many simulations. For example, we now illustrate the use of a queue in simulating traffic flow at Speedo's Car Wash.

Problem Given the arrival times at the car wash, calculate the average waiting time per car.

Analysis We assume that there is one station in the car wash, that is, there is one ''server.'' Each car takes exactly ten minutes to get washed. At any time there will be at most five cars waiting—in a queue—to be washed. If an arrival occurs when there is a car being washed and there are five cars in the queue, that arrival is turned away as an ''overflow'' and not counted. Error messages should be printed for an arrival time that is not an integer, less than zero, greater than the sentinel, or less than the previous arrival time.

The average waiting time is determined by adding up the waiting times for each car and dividing by the number of cars. Here are the details regarding arrivals and departures:

1. If an arrival and departure occur during the same minute, the departure is processed first.
2. If a car arrives when the queue is empty and no cars are being washed, the car starts getting washed immediately; it is not put on the queue.
3. A car leaves the queue, and stops waiting, once the car starts through the ten-minute wash cycle.

The following is a sample list of arrival times:

5 5 7 12 12 13 14 18 19 25 999 (a sentinel)

To calculate the waiting time for each car that is washed, we subtract its arrival time from the time when it entered the car wash. The first arrival, at time 5, entered the wash station right away, so its waiting time was 0. For the second arrival, also at time 5, it was enqueued at time 5, and then dequeued and entered the wash station when the first car left the wash station—at time 15. So the waiting time for the second arrival was 10 minutes. Here is the complete simulation:

Arrival Time	Time Dequeued	Waiting Time
5		0
5	15	10
7	25	18
12	35	23
12	45	33
13	55	42
14	–	Overflow
18	65	47
19	–	Overflow
25	75	50

The sum of the waiting times is 223. The number of cars is 8 (the two overflows at 14 and 19 minutes are not counted), so the average waiting time is 27.875 minutes.

Formally, we supply system tests to specify the expected behavior (that is, in terms of input and output) of the program. The system tests are created before the program is written and provide an indication of the program's correctness. But as we noted in Section 2.4, testing can establish the incorrectness—but not the correctness—of a program. JUnit testing of any class can commence after the class has been designed, that is, after the method specifications have been developed.

System Test 1 (the input is in boldface):

Please enter the next arrival time. The sentinel is 999: **5**

Please enter the next arrival time. The sentinel is 999: **5**

Please enter the next arrival time. The sentinel is 999: **7**

Please enter the next arrival time. The sentinel is 999: **12**

Please enter the next arrival time. The sentinel is 999: **12**

Please enter the next arrival time. The sentinel is 999: **13**

Please enter the next arrival time. The sentinel is 999: **14**

Please enter the next arrival time. The sentinel is 999: **18**

Please enter the next arrival time. The sentinel is 999: **19**

Please enter the next arrival time. The sentinel is 999: **25**

Please enter the next arrival time. The sentinel is 999: **999**

Here are the results of the simulation:

Time	Event	Waiting Time
5	Arrival	
5	Arrival	
7	Arrival	
12	Arrival	
12	Arrival	
13	Arrival	
14	Arrival (Overflow)	
15	Departure	0
18	Arrival	
19	Arrival (Overflow)	
25	Departure	10
25	Arrival	
35	Departure	18
45	Departure	23
55	Departure	33
65	Departure	42
75	Departure	47
85	Departure	50

The average waiting time, in minutes, was 27.875

System Test 2 (input is in boldface):

```
Please enter the next arrival time.  The sentinel is 999: -3
java.lang.IllegalArgumentException   The input must consist of a non-negative integer
less than the sentinel.


Please enter the next arrival time.  The sentinel is 999: 5

Please enter the next arrival time.  The sentinel is 999: m
java.util.InputMismatchException:

Please enter the next arrival time.  The sentinel is 999: 3
java.lang.IllegalArgumentException   The next arrival time must not be less than the
current time.

Please enter the next arrival time.  The sentinel is 999: 1000
java.lang.IllegalArgumentException   The input must consist of a non-negative integer
less than the sentinel.

Please enter the next arrival time.  The sentinel is 999: 10

Please enter the next arrival time.  The sentinel is 999: 999
```

Here are the results of the simulation:

Time	Event	Waiting Time
5	Arrival	
10	Arrival	
15	Departure	0
25	Departure	5

The average waiting time, in minutes, was 2.5

8.2.4.1 Program Design and Testing

As usual, we will separate the processing concerns from the input/output concerns, so we will have two major classes: `CarWash` and `CarWashUser`. The simulation will be ***event driven***, that is, the pivotal decision in processing is whether the next event will be an arrival or a departure. After each of the next-arrival times has been processed, we need to wash any remaining cars and return the results of the simulation.

For now, four methods can be identified. Here are their specifications

```java
/**
 *  Initializes this CarWash object.
 *
 */
public CarWash()
/**
 *  Handles all events from the current time up through the specified time for the next
 *  arrival.
 *
 *  @param nextArrivalTime – the time when the next arrival will occur.
 *
 *  @throws IllegalArgumentException – if nextArrivalTime is less than the
 *          current time.
 *
 *  @return - a string that represents the history of the car wash
 *        through the processing of the car at nextArrivalTime.
 *
 */
public LinkedList<String> process (int nextArrivalTime)


/**
  *  Washes all cars that are still unwashed after the final arrival.
  *
  *  @return - a string that represents the history of the car wash
  *            after all arrivals have been processed (washed or turned away).

  */
  public LinkedList<String> finishUp()
```

```
/**
  *  Returns the history of this CarWash object's arrivals and departures, and the
  *  average waiting time.
  *
  *  @return the history of the simulation, including the average waiting time.
  *
  */
public LinkedList<String> getResults()
```

The `process` and `finishUp` methods return the history of the car wash (times, events, and waiting times), not because that is needed for the definitions of the `process` and `finishUp` methods, but for the sake of *testing* those methods. Recall the maxim from Chapter 2: In general, methods should be designed to facilitate testing.

In the `CarWashTest` class, `carWash` is a field, and `CarWashTest` extends `CarWash` to enable protected methods in `CarWash` to be accessed in `CarWashTest`. Here, for example, are tests for arrivals and overflow:

```
     @Test
     public void twoArrivalsTest()
     {
       carWash.processArrival (5);
       results = carWash.processArrival (7);
       assertTrue (results.indexOf ("5\tArrival") != -1);
       assertTrue (results.indexOf ("7\tArrival") > results.indexOf ("5\tArrival"));
     } // method twoArrivalsTest
     @Test
     public void overflowTest()
     {
       carWash.processArrival (5);
       carWash.processArrival (7);
       carWash.processArrival (8);
       carWash.processArrival (12);
       carWash.processArrival (12);
       assertTrue (carWash.processArrival (13).toString().
                   indexOf (CarWash.OVERFLOW) == -1); // no overflow for arrival at 13
       assertTrue (carWash.processArrival (14).toString().
                   indexOf (CarWash.OVERFLOW) > 0);   // overflow for arrival at 14
     } // method overflowTest
```

The complete project, including all test classes, is available from the book's website.

8.2.4.2 Fields in the `CarWash` Class

The next step in the development of the `CarWash` class is to choose its fields. We'll start with a list of variables needed to solve the problem and then select the fields from this list.

As noted at the beginning of the design stage in Section 8.2.4.1, the essential feature of processing is the determination of whether the next event is an arrival or a departure? We can make this decision based on the values of `nextArrivalTime` (which is read in) and `nextDepartureTime`. The variable `nextArrivalTime` holds the time when the next arrival will occur, and `nextDepartureTime` contains the time when the washing of the car now being washed will be finished. For example, suppose at some point in the simulation, nextArrivalTime contains 28 and nextDepartureTime contains 24. Then the next event in the simulation will be a departure at time 24. If the two times are the same, the next event will be an arrival (see note 1 of Analysis). What if there is no car being washed? Then the next event will be an arrival. To make sure the next event is an arrival no matter what nextArrivalTime holds, we will set nextDepartureTime to a large number—say 10000—when there is no car being washed.

The cars waiting to be washed should be saved in chronological order, so one of the variables needed will be the queue carQueue. Each element in carQueue is a Car object, so we temporarily suspend development of the CarWash class in order to determine the methods the Car class should have.

When a car leaves the queue to enter the wash station, we can calculate that car's waiting time by subtracting the car's arrival time from the current time. So the Car class will provide, at least, a getArrivalTime() method that returns the arrival time of the car that was just dequeued. Beyond that, all the Car class needs is a constructor to initialize a Car object from nextArrivalTime when a Car object is enqueued. The method specifications for the Car class are:

```
/**
 *  Initializes this Car object from the specified time of the next arrival.
 *
 */
public Car (int nextArrivalTime)


/**
 *  Determines the arrival time of this Car object.
 *
 *  @return the arrival time of this Car object.
 *
 */
public int getArrivalTime()
```

We now resume the determination of variables in CarWash. As indicated in the previous paragraph, we should have waitingTime and currentTime variables. To calculate the average waiting time, we need numberOfCarsWashed and sumOfWaitingTimes. Finally, we need a variable, results, to hold each line of output of the simulation. We could simply make results a String variable, but then the concatenation operations would become increasingly expensive. Instead, each line will be appended to a linked list:

```
LinkedList<String> results;
```

At this point, we have amassed eight variables. Which of these should be fields? A simple heuristic (rule of thumb) is that most of a class's public, non-constructor methods should access most of the class's fields (see Riel, [1996] for more details). Clearly, the process method will need all of the variables. The finishUp method will handle the remaining departures, so that method must have access to carQueue, results, sumOfWaitingTimes, waitingTime, currentTime, and nextDepartureTime; these will be fields. The only other field is numberOfCars, needed by the getResults method. There is no need to make nextArrivalTime a field (it is needed only in the process method). Here are the constant identifiers and fields in the CarWash class:

```
public final String OVERFLOW = " (Overflow)\n";

protected final String HEADING =
      "\n\nTime\tEvent\t\tWaiting Time\n";
```

```
protected static final int INFINITY = 10000; // indicates no car being washed

protected static final int MAX_SIZE = 5; // maximum cars allowed in carQueue

protected static final int WASH_TIME = 10; // minutes to wash one car

protected Queue<Car> carQueue;

protected LinkedList<String> results;  // the sequence of events in the simulation

protected int currentTime,
              nextDepartureTime,     // when car being washed will finish
              numberOfCars,
              waitingTime,
              sumOfWaitingTimes;
```

Figure 8.17 has the UML diagrams for the CarWash, Car, and LinkedList classes. For the sake of brevity, the LinkedList fields and methods are not shown, and the Queue interface's methods are not shown because they are implemented by the LinkedList class.

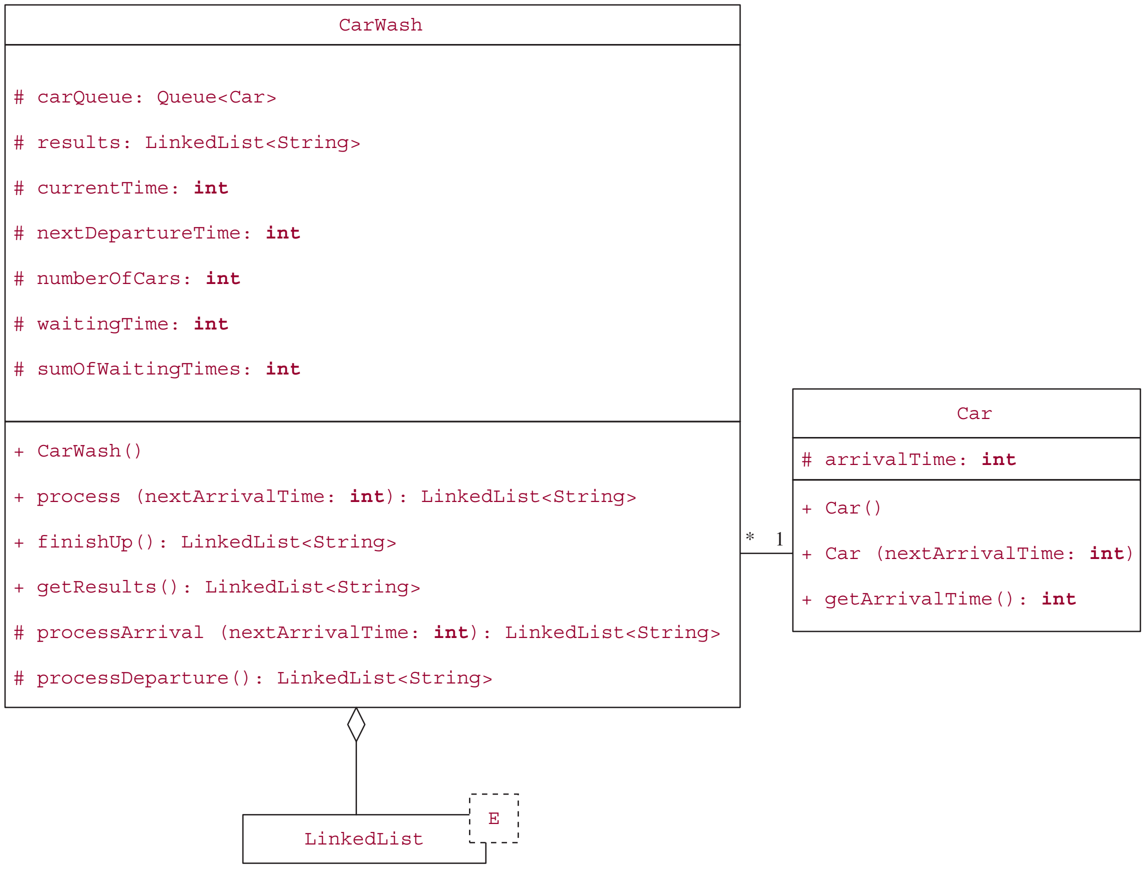

CarWash
# carQueue: Queue<Car>
# results: LinkedList<String>
# currentTime: int
# nextDepartureTime: int
# numberOfCars: int
# waitingTime: int
# sumOfWaitingTimes: int
+ CarWash()
+ process (nextArrivalTime: int): LinkedList<String>
+ finishUp(): LinkedList<String>
+ getResults(): LinkedList<String>
# processArrival (nextArrivalTime: int): LinkedList<String>
# processDeparture(): LinkedList<String>
*
1
Car
# arrivalTime: int
+ Car()
+ Car (nextArrivalTime: int)
+ getArrivalTime(): int
LinkedList
E

FIGURE 8.17 The class diagrams for CarWash and associated classes

8.2.4.3 Method Definitions of the CarWash Class

We now start on the method definitions of the CarWash class. The constructor is straightforward:

```
public CarWash()
{
      carQueue = new LinkedList<Car>();
      results = new LinkedList<String>();
      results.add (HEADING);
      currentTime = 0;
      numberOfCars = 0;
      waitingTime = 0;
      sumOfWaitingTimes = 0;
      nextDepartureTime = INFINITY;   // no car being washed
} // constructor
```

The process method takes the nextArrivalTime read in from the calling method. Then the decision is made, by comparing nextArrivalTime to nextDepartureTime, whether the next event is an arrival or departure. According to the specifications of the problem, we keep processing departures until the next event is an arrival, that is, until nextArrivalTime < nextDepartureTime. Then the arrival at nextArrivalTime is processed. By creating processArrival and processDeparture methods, we avoid getting bogged down in details, at least for now.

```
public LinkedList<String> process (int nextArrivalTime)
{
      final String BAD_TIME =
              "The time of the next arrival cannot be less than the current time.";

      if (nextArrivalTime < currentTime)
             throw new IllegalArgumentException (BAD_TIME);
      while (nextArrivalTime >= nextDepartureTime)
             processDeparture();
      return processArrival (nextArrivalTime);
} // process
```

To process the arrival given by nextArrivalTime, we first update currentTime and check for an overflow. If this arrival is not an overflow, numberOfCars is incremented and the car either starts getting washed (if the wash station is empty) or is enqueued on carQueue. Here is the code:

```
/**
 *  Moves the just arrived car into the car wash (if there is room on the car queue),

 *  or turns the car away (if there is no room on the car queue).
 *
 *  @param nextArrivalTime – the arrival time of the just-arrived car.
 *
 */
protected LinkedList<String> processArrival (int nextArrivalTime)
{
      final String ARRIVAL = "\tArrival";

      currentTime = nextArrivalTime;
```

```
        results.add (Integer.toString (currentTime) + ARRIVAL);
        if (carQueue.size() == MAX_SIZE)
                results.add (OVERFLOW);
        else
        {
              numberOfCars++;
              if (nextDepartureTime == INFINITY)  // if no car is being washed
                      nextDepartureTime = currentTime + WASH_TIME;
              else
                      carQueue.add (new Car (nextArrivalTime));
              results.add ("\n");
        } // not an overflow
        return results;
  } // method processArrival
```

This method reveals how the `Car` class gets involved: there is a constructor with `nextArrivalTime` as its argument. Here is the complete definition of the `Car` class:

```
  public class Car
  {
        protected int arrivalTime;


       /**
        *  Initializes this Car object.
        *
        */
       public Car() { }   // for the sake of subclasses of Car


       /**
        *  Initializes this Car object from the specified time of the next arrival.
        *
        */
        public Car (int nextArrivalTime)
        {
              arrivalTime = nextArrivalTime;
        } // constructor with int parameter


       /**
        *  Determines the arrival time of this Car object.
        *
        *  @return the arrival time of this Car object.
        *
        */
        public int getArrivalTime()
        {
              return arrivalTime;
        } // method getArrivalTime

  } // class Car
```

For this project, we could easily have avoided the Car class, but a subsequent extension of the project might relate to more information about a car: the number of axles, whether it is a convertible, and so on.

To process a departure, we first update currentTime and results. Note that the waiting time for the departing car was calculated when that car entered the wash station—during the previous call to the processDeparture method. We check to see if there are any cars on carQueue. If so, we dequeue the front car, calculate its waiting time, add that to sumOfWaitingTimes, and begin washing that car. Otherwise, we set waitingTime to 0 and nextDepartureTime to a large number to indicate that no car is now being washed. Here is the definition:

```
/**
 *  Updates the simulation to reflect the fact that a car has finished getting washed.
 *
 */
 protected LinkedList<String> processDeparture()
 {
       final String DEPARTURE = "\tDeparture\t\t";

       int arrivalTime;

       currentTime = nextDepartureTime;
       results.add (Integer.toString (currentTime) + DEPARTURE +
                     Integer.toString (waitingTime) + "\n");
       if (!carQueue.isEmpty())
       {
               Car car = carQueue.remove();
               arrivalTime = car.getArrivalTime();
               waitingTime = currentTime - arrivalTime;
               sumOfWaitingTimes += waitingTime;
               nextDepartureTime = currentTime + WASH_TIME;
       } // carQueue was not empty
       else
       {
               waitingTime = 0;
               nextDepartureTime = INFINITY;  // no car is being washed
       } // carQueue was empty
       return results;
} // method processDeparture
```

The finishUp and getResults methods are straightforward:

```
    public LinkedList<String> finishUp()
    {
          while (nextDepartureTime < INFINITY)  // while there are unwashed cars
                 processDeparture();
          return results;
    } // finishUp


    public LinkedList<String> getResults()
    {
          final String NO_CARS_MESSAGE = "There were no cars in the car wash.\n";
```

```
        final String AVERAGE_WAITING_TIME_MESSAGE =
               "\n\nThe average waiting time, in minutes, was ";

        if (numberOfCars == 0)
              results.add (NO_CARS_MESSAGE);
        else
             results.add (AVERAGE_WAITING_TIME_MESSAGE + Double.toString (
                      (double) sumOfWaitingTimes / numberOfCars));
        return results;
  } // method getResults
```

8.2.4.4 The CarWashUser Class

The CarWashUser class has a run method and a printResults method (in addition to the usual main method). Here are the method specifications for the run method and the printResults method :

```
/**
 *  Reads in all of the arrival times, runs the simulation, and calculates the average
 *  waiting time.
 *
 */
 public void run()
/**
 *  Prints the results of the simulation.
 *
 */
 public void printResults()
```

There is no CarWashUserTest class because the CarWashUser class has no testable methods. Figure 8.18 has the UML class diagrams for this project.

The run method repeatedly—until the sentinel is reached—reads in a value for nextArrivalTime. Unless an exception is thrown (for example, if the value is not an **int**), carWash.process (nextArrivalTime) is called. When the sentinel is read in, the loop is exited and carWash.finishUp() and printResults (carWash) are called. Here is the code:

```
  public void run()
  {
        final int SENTINEL = 999;

        final String INPUT_PROMPT = "\nPlease enter the next arrival time (or " +
               SENTINEL + " to quit): ";

        final String OUT_OF_RANGE = "The input must consist of a non-" +
              "negative integer less than the sentinel.";

        CarWash carWash = new CarWash();

        Scanner sc = new Scanner (System.in);
```

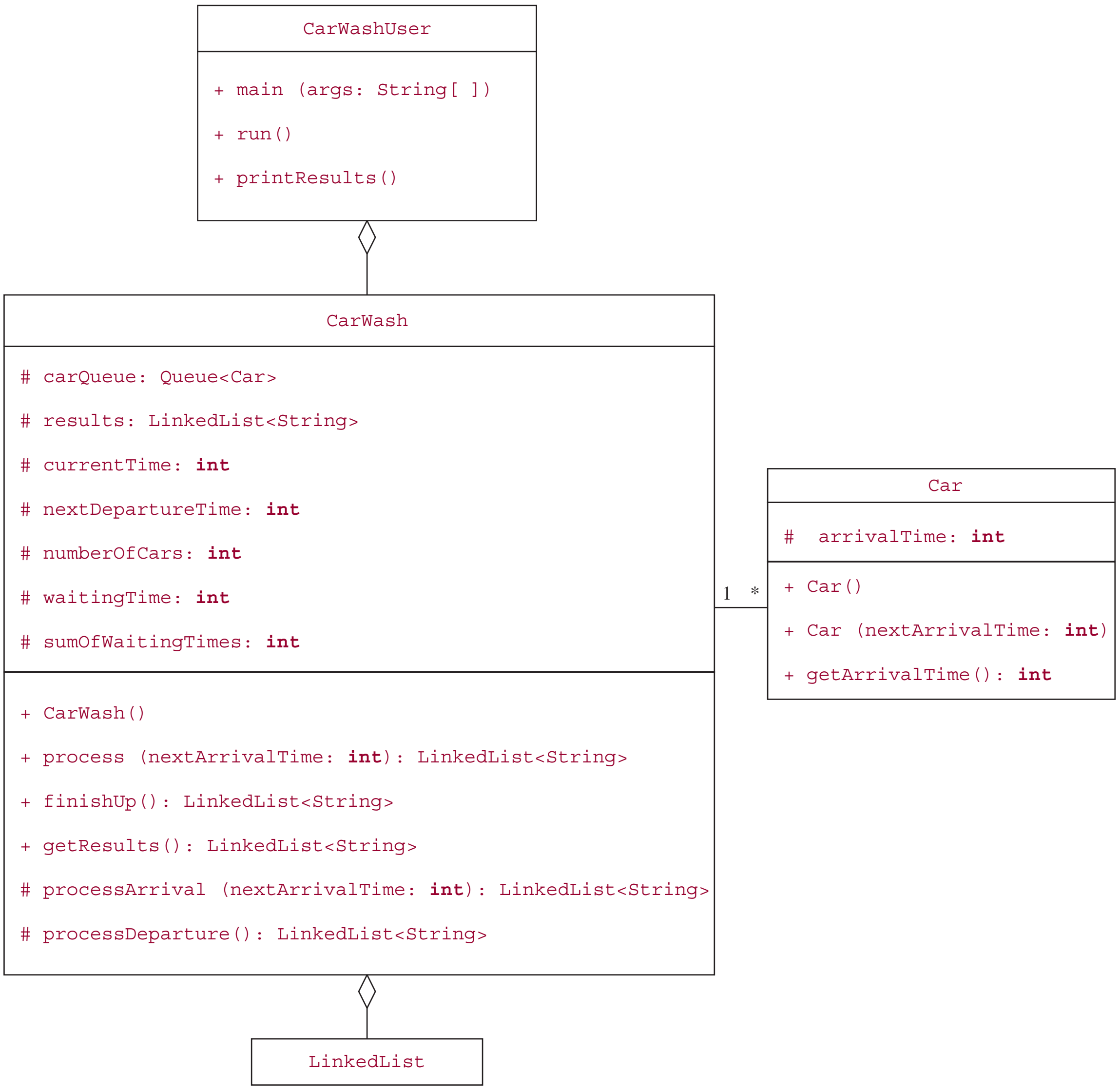

CarWashUser
+ main (args: String[ ])
+ run()
+ printResults()
CarWash
# carQueue: Queue<Car>
# results: LinkedList<String>
# currentTime: int
# nextDepartureTime: int
# numberOfCars: int
# waitingTime: int
# sumOfWaitingTimes: int
+ CarWash()
+ process (nextArrivalTime: int): LinkedList<String>
+ finishUp(): LinkedList<String>
+ getResults(): LinkedList<String>
# processArrival (nextArrivalTime: int): LinkedList<String>
# processDeparture(): LinkedList<String>
1
*
Car
# arrivalTime: int
+ Car()
+ Car (nextArrivalTime: int)
+ getArrivalTime(): int
LinkedList

FIGURE 8.18 UML class diagrams for the `CarWash` project

```
int nextArrivalTime;

while (true)
{
        System.out.print (INPUT_PROMPT);
        try
        {
                nextArrivalTime = sc.nextInt();
                if (nextArrivalTime == SENTINEL)
```

```
                        break;
                if (nextArrivalTime < 0 || nextArrivalTime > SENTINEL)
                        throw new NumberFormatException (OUT_OF_RANGE);
                carWash.process (nextArrivalTime);
            } // try
            catch (Exception e)
            {
                System.out.println(e);
                sc.nextLine();
            } // catch
        } // while
        carWash.finishUp();
        printResults (carWash);
    } // method run
```

The definition of the `printResults` method needs no explanation:

```
public void printResults()
{
      final String RESULTS_HEADING =
            "\nHere are the results of the simulation:\n";

      LinkedList<String> results = carWash.getResults();
      System.out.println (RESULTS_HEADING);
      for (String s : results)
            System.out.print (s);
} // method printResults
```

For the `run` method, worstTime(n) is linear in n, where n is the number of lines of input. There are loops in the definitions of the `CarWash` methods `process` and `finishUp`, but those loops are independent of `n`; in fact, the number of iterations of either loop is at most 5, the maximum size of the car queue.

8.2.4.5 Randomizing the Arrival Times

It is not necessary that the arrival times be read in. They can be generated by your simulation program, provided the input includes the ***mean arrival time***, that is, the average time between arrivals for the population. In order to generate the list of arrival times from the mean arrival time, we need to know the distribution of arrival times. We now define a function that calculates the distribution, known as the ***Poisson distribution***, of times between arrivals. The mathematical justification for the following discussion is beyond the scope of this book—the interested reader may consult a text on mathematical statistics.

Let x be any time between arrivals. Then $F(x)$, the probability that the time until the next arrival will be at least x minutes from now, is given by

```
F(x) = exp(-x / meanArrivalTime)
```

For example, $F(0) = \exp(0) = 1$; that is, it is certain that the next arrival will occur at least 0 minutes from now. Similarly, $F(\text{meanArrivalTime}) = \exp(-1) \sim 0.4.F(10000 * \text{meanArrivalTime})$ is approximately 0. The graph of the function F is shown in Figure 8.19.

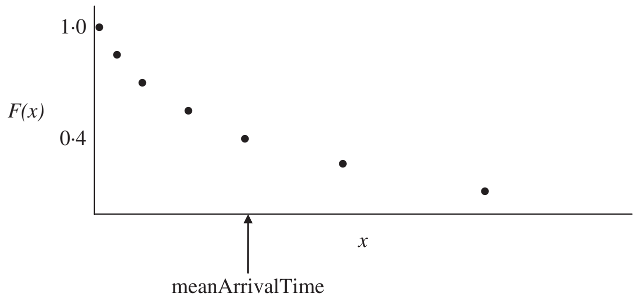


FIGURE 8.19 Graph of the Poisson distribution of interarrival times

To generate the arrival times randomly, we introduce an integer variable called `timeTillNext`, which will contain the number of minutes from the current time until the next arrival. We determine the value for `timeTillNext` as follows. According to the distribution function *F* given previously, the probability that the next arrival will take at least `timeTillNext` minutes is given by

```
exp(-timeTillNext / meanArrivalTime)
```

This expression represents a probability, specifically, a floating point number that is greater than 0.0 and less than or equal to 1.0. To randomize this probability, we associate the expression with the value of a random variable, `randomDouble`, in the same range. So we set

```
randomDouble = random.nextDouble();
```

Then `randomDouble` contains a **`double`** value that is greater than or equal to 0.0 and less than 1.0. So 1—`randomDouble` will contain a value that is greater than 0.0 and less than or equal to 1.0. This is what we want, so we equate 1—randomDouble with exp(–timeTillNext/meanArrivalTime):

1 –randomDouble = exp (–timeTillNext / meanArrivalTime)

To solve this equation for timeTillNext, we take logs of both sides:

log (1 –randomDouble) = –timeTillNext / meanArrivalTime

Now each side is multiplied by—meanArrivalTime, to yield

timeTillNext = –meanArrivalTime ∗ log (1 –randomDouble)

In Java code, we get:

```
timeTillNext = (int)Math.round (-meanArrivalTime * Math.log (1 - randomDouble));
```

We round the result so that `timeTillNext` will be an integer.

To illustrate how the values would be calculated, suppose that the mean arrival time is 3 minutes and the list of values of `1 - randomDouble` starts with 0.715842, 0.28016, and 0.409589. Then the first three, randomized values of `timeTillNext` will be

```
1, that is, (int)Math.round (-3 * log (0.715842)),

4, that is, (int)Math.round (-3 * log (0.28016)), and

3, that is, (int)Math.round (-3 * log (0.409589)).
```

The first car will arrive one minute after the car wash opens and the second car will arrive four minutes later, at minute 5. The third car will arrive three minutes later, at minute 8.

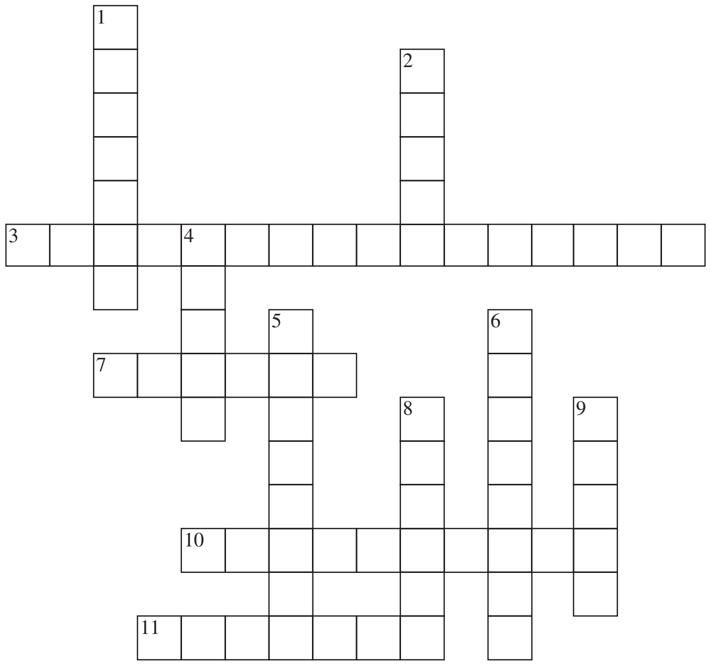
You are now prepared to do Lab 16: Randomizing the Arrival Times

SUMMARY

A ***stack*** is a finite sequence of elements in which insertions and deletions can take place only at one end of the sequence, called the *top* of the stack. Because the most recently inserted element is the next element to be removed, a stack is a last-in-first-out (LIFO) structure. Compilers implement recursion by generating code for pushing and popping activation records onto a run-time stack whose top record holds the state of the method currently being executed. Another stack application occurs in the translation of infix expressions into machine code. With the help of an operator stack, an infix expression can be converted into a postfix expression, which is an intermediate form between infix and machine language. For this conversion, worstTime(n) is linear in n, the size of the infix expression.

A ***queue*** is a finite sequence of elements in which insertions can take place only at the back, and removals can take place only at the front. Because the first element inserted will be the first element to be removed, a queue is a first-in-first-out (FIFO) structure. The inherent fairness of this *first-come-first-served* restriction has made queues important components of many systems. Specifically, queues play a key role in the development of computer models to study the behavior of those systems.

CROSSWORD PUZZLE



www.CrosswordWeaver.com

ACROSS

3. A two-dimensional array that directs the conversion from infix notation to postfix notation.

7. The immediate superclass the `Stack` class.

10. The information saved whenever a method is called.

11. A notation in which each operator immediately follows its operands.

DOWN

1. The first widely used programming language.

2. The smallest meaningful unit in a program.

4. The area of main memory that is allocated for a run-time stack.

5. The worstTime(*n*) for the `add (E element)`, `remove()` and `element()` methods in the `LinkedList` implementation of the `Queue` interface.

6. A process in which the factors that produce a result are themselves affected by that result.

8. A notation in which each operator immediately precedes its operands.

9. A simplification of a system.

CONCEPT EXERCISES

8.1 What advantage was obtained by implementing the `List` interface before declaring the `Stack` class?

8.2 Suppose we define:

```
Queue<Integer> queue = new LinkedList<Integer>();
```

Show what the `LinkedList` object (referenced by) `queue` will look like after each of the following messages is sent:

a. `queue.add (2000);`

b. `queue. add (1215);`

c. `queue. add (1035);`

d. `queue. add (2117);`

e. `queue.remove();`

f. `queue. add (1999);`

g. `queue.remove();`

8.3 Re-do Exercise 8.2, parts a through g, for a stack instead of a queue. Start with

```
Stack<Integer> stack = new Stack<Integer>();

stack.push (2000);
```

8.4 Suppose that elements `"a"`, `"b"`, `"c"`, `"d"`, `"e"` are pushed, in that order, onto an initially empty stack, which is then popped four times, and as each element is popped, it is enqueued into an initially empty queue. If one element is then dequeued from the queue, what is the *next* element to be dequeued?

8.5 Use a stack of activation records to trace the execution of the recursive `fact` method after an initial call of `fact (4)`. Here is the method definition:

```
/**
/**
 * Calculates n!.
 *
 *  @param n the integer whose factorial is calculated.
 *
 *  @return n!.
 *
 */
 protected static long fact (int n)
 {
       if (n <= 1)
               return 1;
       return n * fact (n - 1);
 } // method fact
```

8.6 Translate the following expressions into postfix notation:

1. $x + y * z$
2. $(x + y) * z$
3. $x - y - z * (a + b)$
4. $(a + b) * c - (d + e * f/((g/h + i - j) * k))/ r$

Test your answers by running the `InfixToPostfix` project in Lab 15.

8.7 Translate each of the expressions in Programming Exercise 8.6 into prefix notation.

8.8 An expression in postfix notation can be evaluated at run time by means of a stack. For simplicity, assume that the postfix expression consists of integer values and binary operators only. For example, we might have the following postfix expression:

$$8\ 5\ 4\ +\ *\ 7\ -$$

The evaluation proceeds as follows: When a value is encountered, it is pushed onto the stack. When an operator is encountered, the first—that is, top—and second elements on the stack are retrieved and popped, the operator is applied (the second element is the left operand, the first element is the right operand) and the result is pushed onto the stack. When the postfix expression has been processed, the value of that expression is the top (and only) element on the stack.

For example, for the preceding expression, the contents of the stack would be as follows:

```
                                      4
                          5           5
              8           8           8
 -----      -----       -----       -----


  9                       7
  8           72          72          65
 -----      -----       -----       -----
```

Convert the following expression into postfix notation and then use a stack to evaluate the expression:

$$5 + 2 * \ (30 - 10/5)$$

PROGRAMMING EXERCISES

8.1 Declare and test the `PureStack` class (see Section 8.1.2) with a `LinkedList` field.
Hint: Each of the definitions is a one-liner.

8.2 Declare and test the `PureStack` class (see Section 8.1.2) with an `ArrayList` field.
Hint: For the sake of efficiency, the top of the stack should be at index `size() - 1`.

8.3 Develop an iterative, stack-based version of the `ways` method in Programming Exercise 5.7. Test your method with unit testing.

Programming Project 8.1

Making the Speedo's Car Wash Simulation More Realistic

Problem Expand the Car Wash Simulation Project with random arrival and service times. Use unit testing of new methods after you have specified those methods.

Analysis The arrival times—with a Poisson distribution—should be generated randomly from the mean arrival time. Speedo has added a new feature: The service time is not necessarily 10 minutes, but depends on what the customer wants done, such as wash only, wash and wax, wash and vacuum, and so on. The service time for a car should be calculated just before the car enters the wash station—that's when the customer knows how much time will be taken until the customer leaves the car wash. The service times, also with a Poisson distribution, should be generated randomly from the mean service time with the same random-number generator used for arrival times.

The input consists of three positive integers: the mean arrival time, the mean service time, and the maximum arrival time. Repeatedly re-prompt until each value is a positive integer.

Calculate the average waiting time and the average queue length, both to one fractional digit[5]. The average waiting time is the sum of the waiting times divided by the number of customers.

The average queue length is the sum of the queue lengths for each minute of the simulation divided by the number of minutes until the last customer departs. To calculate the sum of the queue lengths, we add, for each minute of the simulation, the total number of customers on the queue during that minute. We can calculate this sum another way: we add, for each customer, the total number of minutes that customer was on the queue. But this is the sum of the waiting times! So we can calculate the average queue length as the sum of the waiting times divided by the number of minutes of the simulation until the last customer departs. And we already calculated the sum of the waiting times for the average waiting time.

Also calculate the number of overflows. Use a seed of 100 for the random-number generator, so the output you get should have the same values as the output from the following tests.

System Test 1: (the input is in boldface)
Please enter the mean arrival time: **3**

Please enter the mean service time: **5**

Please enter the maximum arrival time: **25**

Time	Event	Waiting Time
4	Arrival	
5	Departure	0
7	Arrival	
10	Arrival	
13	Arrival	
14	Arrival	
15	Departure	0
16	Arrival	

Time	Event	Waiting Time
18	Arrival	
20	Arrival	
21	Departure	5
32	Departure	8
34	Departure	18
37	Departure	18
41	Departure	19
46	Departure	21

[5] Given a **`double`** `d`, you can print `d` rounded to one fractional digit as follows: `System.out.println (Math.round (d * 10)/10.0);`

(continued on next page)

(continued from previous page)

```
The average waiting time was 11.1 minutes per car.
The average queue length was 1.9 cars per minute.
The number of overflows was 0.
```

System Test 2: (input in boldface)

```
Please enter the mean arrival time: 1

Please enter the mean service time: 1

Please enter the maximum arrival time: 23

Here are the results of the simulation:

Time   Event                 Waiting Time      Time   Event        Waiting Time

  1    Arrival
  1    Departure                  0             13    Arrival
  2    Arrival                                  14    Departure          5
  3    Arrival                                  14    Departure          5
  4    Departure                  0             14    Departure          4
  4    Departure                  1             14    Arrival
  4    Arrival                                  15    Departure          3
  5    Departure                  0             15    Departure          4
  5    Arrival                                  15    Arrival
  6    Departure                  0             16    Departure          2
  6    Arrival                                  16    Departure          2
  8    Arrival                                  18    Arrival
  8    Arrival                                  19    Departure          1
  9    Arrival                                  20    Departure          1
 10    Arrival                                  22    Arrival
 11    Departure                  0             22    Departure          0
 11    Arrival                                  22    Arrival
 11    Arrival                                  23    Arrival
 11    Arrival (Overflow)                       23    Arrival
 12    Arrival (Overflow)                       24    Departure          0
 12    Arrival (Overflow)                       24    Departure          1
 13    Departure                  3             24    Departure          1
```

The average waiting time was 1.7 minutes per car. The average queue length was 1.4 cars per minute. The number of overflows was 3.

Programming Project 8.2

Design, Test, and Implement a Program to Evaluate a Condition

Analysis The input will consist of a condition (that is, a Boolean expression) followed by the values—one per line—of the variables as they are first encountered in the condition. For example:

```
b * a > a + c
6
2
7
```

The variable `b` gets the value 6, `a` gets 2, and `c` gets 7. The operator `*` has precedence over `>`, and `+` has precedence over `>`, so the value of the above expression is true (12 is greater than 9).

Each variable will be given as an identifier, consisting of lower-case letters only. All variables will be integer-valued. There will be no constant literals. The legal operators and precedence levels—high to low—are:

```
*, /, %
+, - (that is, integer addition and subtraction)
>, >=, <=, <
==, !=
&&
|| <
```

Parenthesized subexpressions are legal. You need not do any input editing, that is, you may assume that the input is a legal expression.

System Test 1 (Input in boldface):
Please enter a condition, or $ to quit: **b * a > a + c**

Please enter a value: **6**

Please enter a value: **2**

Please enter a value: **7**
The value of the condition is true.

Please enter a condition, or $ to quit: **b * a < a + c**

Please enter a value: **6**

Please enter a value: **2**

Please enter a value: **7**
The value of the condition is false.

Please enter a condition, or $ to quit: **m + j * next == current * (next - previous)**

(continued on next page)

(continued from previous page)

```
Please enter a value: 6

Please enter a value: 2

Please enter a value: 7

Please enter a value: 5

Please enter a value: 3
The value of the condition is true.

Please enter a condition, or $ to quit: m + j * next ! =  current * (next - previous)

Please enter a value: 6

Please enter a value: 2

Please enter a value: 7

Please enter a value: 5

Please enter a value: 3
The value of the condition is false.

Please enter a condition, or $ to quit: a * (b + c / (d - b)  * e) > =  a + b + c + d + e

Please enter a value: 6

Please enter a value: 2

Please enter a value: 7

Please enter a value: 5

Please enter a value: 3
The value of the condition is true.

Please enter a condition, or $ to quit: a * (b + c / (d - b)  * e) < =  a + b + c + d + e

Please enter a value: 6

Please enter a value: 2

Please enter a value: 7

Please enter a value: 5

Please enter a value: 3
```

```
The value of the condition is false.

Please enter a condition, or $ to quit: $
```

System Test 2 (Input in boldface):

```
Please enter a condition, or $ to quit: b < c && c < a

Please enter a value: 10

Please enter a value: 20

Please enter a value: 30
The value of the condition is true.

Please enter a condition, or $ to quit: b < c && a < c

Please enter a value: 10

Please enter a value: 20

Please enter a value: 30
The value of the condition is false.

Please enter a condition, or $ to quit: b < c || a < c

Please enter a value: 10

Please enter a value: 20

Please enter a value: 30
The value of the condition is true.

Please enter a condition, or $ to quit: c < b || c > a

Please enter a value: 10

Please enter a value: 20

Please enter a value: 30
The value of the condition is true.

Please enter a condition, or $ to quit: b ! =  a || b < =  c && a > =  c
```

(continued on next page)

(continued from previous page)

Please enter a value: **10**

Please enter a value: **20**

Please enter a value: **30**
The value of the condition is true.

Please enter a condition, or $ to quit: **(b != a || b <= c) && a >= c**

Please enter a value: **10**

Please enter a value: **20**

Please enter a value: **30**
The value of the condition is false.

Please enter a condition, or $ to quit: **a / b * b + a % b == a**

Please enter a value: **17**

Please enter a value: **5**
The value of the condition is true.

Please enter a condition, or $ to quit: **$**

Hint See Lab 15 on converting infix to postfix, and Concept Exercise 8.8. After constructing the `postfix` queue, create `values`, an `ArrayList` object with `Integer` elements. The `values` object corresponds to `symbolTable`, the `ArrayList` of identifiers. Use a stack, `runTimeStack`, for pushing and popping `Integer` and `Boolean` elements. Because `runTimeStack` contains both `Integer` and `Boolean` elements, it should not have a type argument.

Programming Project 8.3

Maze-Searching, Revisited

Re-do and re-test the maze-search project in Chapter 5 by replacing `tryToReachGoal` with an iterative method. **Hint:** use a stack to simulate the recursive calls to `tryToReachGoal`.

The original version of the project is in the Chapter 5 subdirectory on the book's website.

Programming Project 8.4

Fixing the `Stack` Class

Design, test, and implement a subclass of the `Stack` class, `PureStack`, that satisfies the definition of a stack. Specifically, `PureStack` will throw `UnsupportedOperationException` for any method call that attempts to remove an element that is not the top element on the stack or attempts to insert an element except onto the top of the stack. Furthermore, a forward iteration (using the `next()` method, either explicitly or implicitly) through a `PureStack` object will start at the top of the stack and work down to the bottom of the stack. And a reverse iteration will start at the bottom of the stack.

Binary Trees

CHAPTER 9

In this chapter we "branch" out from the linear structures of earlier chapters to introduce what is essentially a non-linear construct: the binary tree. This brief chapter focuses on the definition and properties of binary trees, and that will provide the necessary background for the next four chapters. Chapters 10 through 13 will consider various specializations of binary trees: binary search trees, AVL trees, decision trees, red-black trees, heaps, and Huffman trees. There is no question that the binary tree is one of the most important concepts in computer science. Finally, to round out the picture, Chapter 15 presents the topic of trees in general.

CHAPTER OBJECTIVES

1. Understand binary-tree concepts and important properties, such as the Binary Tree Theorem and the External Path Length Theorem.
2. Be able to perform various traversals of a binary tree.

9.1 Definition of Binary Tree

The following definition sets the tone for the whole chapter:

A ***binary tree*** *t* is either empty or consists of an element, called the ***root element***, and two distinct binary trees, called the ***left subtree*** and ***right subtree*** of *t*.

We denote those subtrees as leftTree(t) and rightTree(t), respectively. Functional notation, such as leftTree(t), is utilized instead of object notation, such as `t.leftTree()`, because there is no binary-tree data structure. Why not? Different types of binary trees have widely differing methods—even different parameters lists—for such operations as inserting and removing. Note that the definition of a binary tree is recursive, and many of the definitions associated with binary trees are naturally recursive.

In depicting a binary tree, the root element is shown at the top, by convention. To suggest the association between the root element and the left and right subtrees, we draw a southwesterly line from the root element to the left subtree and a southeasterly line from the root element to the right subtree. Figure 9.1 shows several binary trees.

The binary tree in Figure 9.1a is different from the binary tree in Figure 9.1b because B is in the left subtree of Figure 9.1a but B is not in the left subtree of Figure 9.1b. As we will see in Chapter 15, those two binary trees *are* equivalent when viewed as general trees.

A subtree of a binary tree is itself a binary tree, and so Figure 9.1a has seven binary trees: the whole binary tree, the binary tree whose root element is B, the binary tree whose root element is C, and four

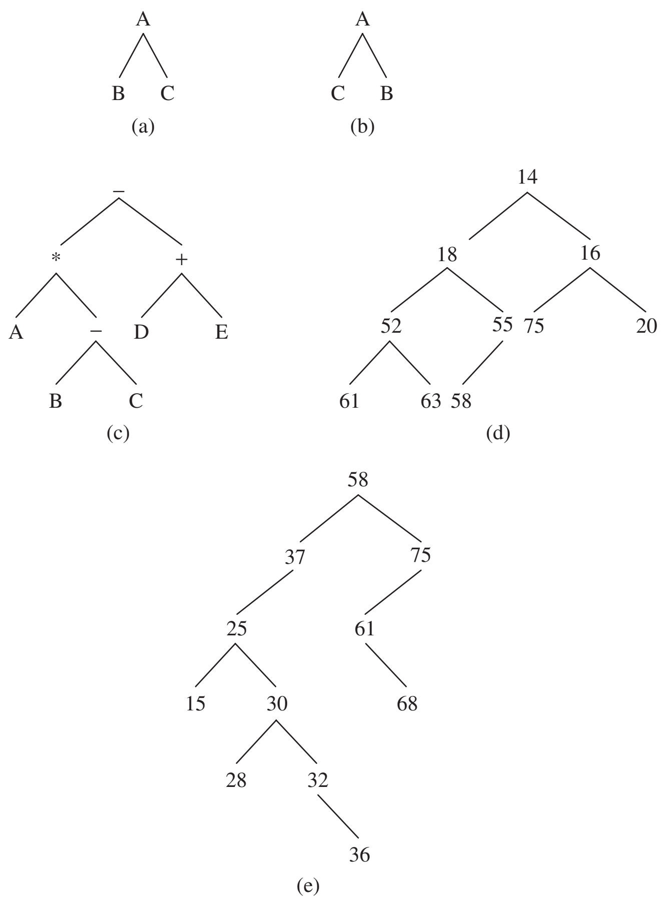


FIGURE 9.1 Several binary trees

empty binary trees. Try to calculate the total number of subtrees for the tree in Figures 9.1 c, d, and e, and hypothesize the formula to calculate the number of subtrees as a function of the number of elements.

The next section develops several properties of binary trees, and most of the properties are relevant to the material in later chapters.

9.2 Properties of Binary Trees

In addition to "tree" and "root," botanical terms are used for several binary-tree concepts. The line from a root element to a subtree is called a ***branch***. An element whose associated left and right subtrees are both empty is called a ***leaf***. A leaf has no branches going down from it. In the binary tree shown in Figure 9.1e, there are four leaves: 15, 28, 36, and 68. We can determine the number of leaves in a binary tree recursively. Let t be a binary tree. The number of leaves in t, written ***leaves(t)***, can be defined as follows:

```
if $t$ is empty
        leaves($t$) = 0
else if $t$ consists of a root element only
```

```
        leaves($t$) = 1
    else
        leaves($t$) = leaves(leftTree($t$)) + leaves(rightTree($t$))
```

This is a mathematical definition, not a Java method. The last line in the above definition states that the number of leaves in t is equal to the number of leaves in t's left subtree plus the number of leaves in t's right subtree. Just for practice, try to use this definition to calculate the number of leaves in Figure 9.1a. Of course, you can simply look at the whole tree and count the number of leaves, but the above definition of leaves(t) is atomic rather than holistic.

Each element in a binary tree is uniquely determined by its location in the tree. For example, let t be the binary tree shown in Figure 9.1c. There are two elements in t with value '–'. We can distinguish between them by referring to one of them as "the element whose value is '–' and whose location is at the root of t" and the other one as "the element whose value is '–' and whose location is at the root of the right subtree of the left subtree of t." We loosely refer to "an element" in a binary tree when, strictly speaking, we should say "the element at such and such a location."

Some binary-tree concepts use familial terminology. Let t be the binary tree shown in Figure 9.2. We say that x is the ***parent*** of y and that y is the ***left child*** of x. Similarly, we say that x is the ***parent*** of z and that z is the ***right child*** of x.

In a binary tree, each element has zero, one, or two children. For example, in Figure 9.1d, 14 has two children, 18 and 16; 55 has 58 as its only child; 61 is childless, that is, it is a leaf. For any element w in a tree, we write parent(w) for the parent of w, left(w) for the left child of w and right(w) for the right child of w.

In a binary tree, the root element does not have a parent, and every other element has exactly one parent. Continuing with the terminology of a family tree, we could define sibling, grandparent, grandchild, first cousin, ancestor and descendant. For example, an element A is an ***ancestor*** of an element B if B is in the subtree whose root element is A. To put it recursively, A is an ***ancestor*** of B if A is the parent of B or if A is an ancestor of the parent of B. Try to define "descendant" recursively.

If A is an ancestor of B, the ***path*** from A to B is the sequence of elements, starting with A and ending with B, in which each element in the sequence (except the last) is the parent of the next element. For example, in Figure 9.1e, the sequence 37, 25, 30, 32 is the path from 37 to 32.

Informally, the height of a binary tree is the number of branches between the root and the farthest leaf, that is, the leaf with the most ancestors. For example, Figure 9.3 has a binary tree of height 3.

The height of the tree in Figure 9.3 (next page) is 3 because the path from E to S has three branches. Suppose for some binary tree, the left subtree has a height of 12 and the right subtree has a height of 20. What is the height of the whole tree? The answer is 21.

In general, the height of a tree is one more than the maximum of the heights of the left and right subtrees. This leads us to a recursive definition of the height of a binary tree. But first, we need to know what the base case is, namely, the height of an empty tree. We want the height of a single-element tree to be 0: there are no branches from the root element to itself. But that means that 0 is one more than the maximum heights of the left and right subtrees, which are both empty. So we need to define the height of an empty subtree to be, strangely enough, -1.

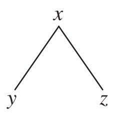


FIGURE 9.2 A binary tree with one parent and two children

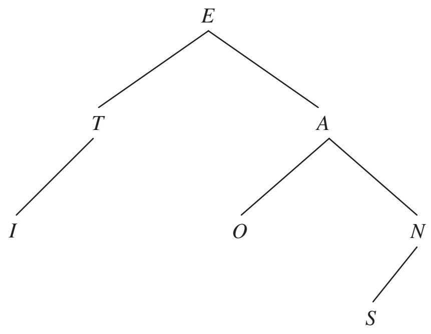


FIGURE 9.3 A binary tree of height 3

Let t be a binary tree. We define height(t), the ***height*** of t, recursively as follows:

```
if $t$ is empty,
    height($t$) $= -1$
else
    height($t$) $= 1+$ max (height(leftTree($t$)), height(rightTree($t$)))
```

It follows from this definition that a binary tree with a single element has height 0 because each of its empty subtrees has a height of −1. Also, the height of the binary tree in Figure 9.1a is 1. And, if you want to try some recursive gymnastics, you can verify that the height of the binary tree in Figure 9.1e is 5.

Height is a property of an entire binary tree. For each element in a binary tree, we can define a similar concept: the level of the element. Speaking non-recursively, if x is an element in a binary tree, we define level(x), the ***level*** of element x, to be the number of branches between the root element and element x. Figure 9.4 shows a binary tree, with levels.

In Figure 9.4, level(N) is 2. Notice that the level of the root element is 0, and the height of a tree is equal to the highest level in the tree. Here is a recursive definition of the level of an element. For any element x in a binary tree, we define level(x), the ***level*** of element x, as follows:

```
if $x$ is the root element,
    level($x$) $= 0$
```

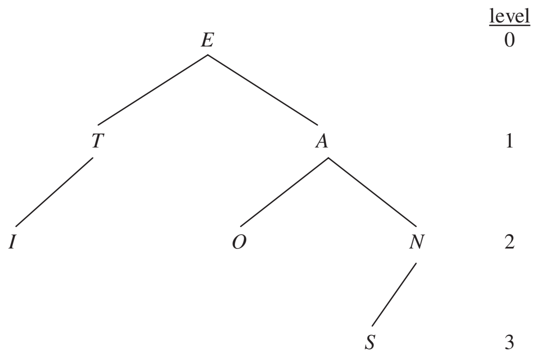


FIGURE 9.4 A binary tree, with the levels of elements shown

```
else
    level(x) = 1+ level(parent(x))
```

An element's level is also referred to as that element's ***depth***. Curiously, the *height* of a non-empty binary tree is the *depth* of the farthest leaf.

A ***two-tree*** is a binary tree that either is empty or in which each non-leaf has 2 branches going down from it. For example, Figure 9.5a has a two-tree and the tree in Figure 9.5b is not a two-tree.

Recursively speaking, a binary tree t is a ***two-tree*** if:

t has at most one element

or

leftTree(t) and rightTree(t) are non-empty two-trees.

A binary tree t is ***full*** if t is a two-tree with all of its leaves on the same level. For example, the tree in Figure 9.6a is full and the tree in Figure 9.6b is not full.

Recursively speaking, a binary tree t is ***full*** if:

t is empty

or

t's left and right subtrees have the same height and both are full.

Of course, every full binary tree is a two-tree but the converse is not necessarily true. For example, the tree in Figure 9.6b is a two-tree but is not full. For full binary trees, there is a relationship between the

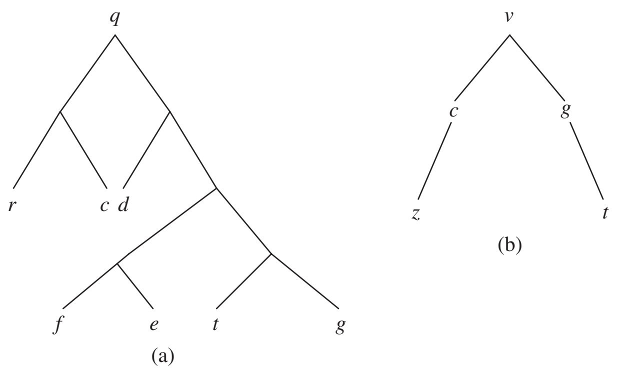


FIGURE 9.5 (a) a two tree; (b) a binary tree that is not a two tree

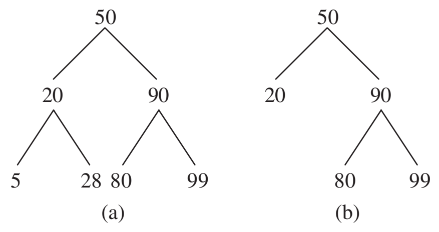


FIGURE 9.6 (a) a full binary tree; (b) a binary tree that is not full

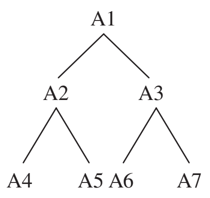


FIGURE 9.7 A full binary tree of height 2; such a tree must have exactly 7 elements

height and number of elements in the tree. For example, the full binary tree in Figure 9.7 has a height of 2, so the tree must have exactly 7 elements:

How many elements must there be in a full binary tree of height 3? Of height 4? For a full binary tree t, can you conjecture the formula for the number of elements in t as a function of height(t)? The answer can be found in Section 9.3.

A binary tree t is ***complete*** if t is full through the next-to-lowest level and all of the leaves at the lowest level are as far to the left as possible. By "lowest level," we mean the level farthest from the root.

Any full binary tree is complete, but the converse is not necessarily true. For example, Figure 9.8a has a complete binary tree that is not full. The tree in Figure 9.8b is not complete because it is not full at the next-to-lowest level: C has only one child. The tree in Figure 9.8c is not complete because leaves I and J are not as far to the left as they could be.

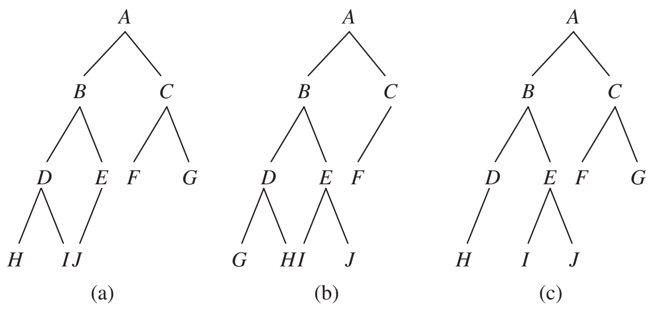


FIGURE 9.8 Three binary trees, of which only (a) is complete

In a complete binary tree, we can associate a "position" with each element. The root element is assigned a position of 0. For any nonnegative integer i, if the element at position i has children, the position of its left child is $2i + 1$ and the position of its right child is $2i + 2$. For example, if a complete binary tree has ten elements, the positions of those elements are as indicated in Figure 9.9.

If we use a left shift (that is, operator $<<$) of one bit, we can achieve the same effect as multiplying by 2, but much faster. Then the children of the element at position i are at positions

```
(i << 1) + 1 and (i << 1) + 2
```

In Figure 9.9, the parent of the element at position 8 is in position 3, and the parent of the element in position 5 is in position 2. In general, if i is a positive integer, the position of the parent of the element in position i is in position $(i - 1) >> 1$; we use a right shift of one bit instead of division by 2.

The position of an element is important because we can implement a complete binary tree with a contiguous collection such as an array or an `ArrayList`. Specifically, we will store the element that is at position i in the tree at index i in the array. For example, Figure 9.10 shows an array with the elements from Figure 9.8a.

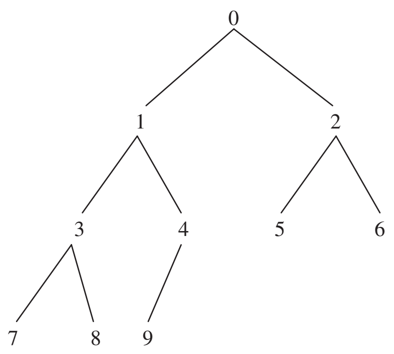


FIGURE 9.9 The association of consecutive integers to elements in a complete binary tree

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

FIGURE 9.10 An array that holds the elements from the complete binary tree in Figure 9.8a

If a complete binary tree is implemented with an `ArrayList` object or array object, the random-access property of arrays allows us to access the children of a parent (or parent of a child) in constant time. That is exactly what we will do in Chapter 13.

We have shown how we can recursively calculate leaves(t), the number of leaves in a binary tree t, and height(t), the height of a binary tree t. We can also recursively calculate the number of elements, $n(t)$, in t:

```
if $t$ is empty
    $n(t) = 0$
else
    $n(t) = 1 + n(\text{leftTree}(t)) + n(\text{rightTree}(t))$
```

9.3 The Binary Tree Theorem

For any non-empty binary tree t, leaves(t) $\leq n(t)$, and leaves(t) $= n(t)$ if and only if t consists of one element only. The phrase "if and only if" indicates that each of the individual statements follows from the other. Namely, for a non-empty binary tree t, if t consists of a single element only, then leaves(t) $= n(t)$; and if leaves(t) $= n(t)$, then t consists of a single element only.

The following theorem characterizes the relationships among leaves(t), height(t) and $n(t)$.

Binary Tree Theorem For any non-empty binary tree t,

1. leaves(t) $\leq \dfrac{n(t) + 1}{2.0}$
2. $\dfrac{n(t) + 1}{2.0} \leq 2^{\text{height}(t)}$
3. Equality holds in part 1 if and only if t is a two-tree.
4. Equality holds in part 2 if and only if t is full.

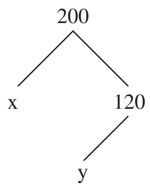


FIGURE 9.11 A binary tree t with $(n(t) + 1)/2$ leaves that is not a two tree

Note: Because 2.0 is the denominator of the division in part 1, the quotient is a floating point value. For example, 7/2.0 = 3.5. We cannot use integer division because of part 3: let t be the binary tree in Figure 9.11.

For the tree in Figure 9.11, leaves(t) $= 2 = (n(t) + 1)/2$ if we use integer division. But t is not a two-tree. Note that $(n(t) + 1)/2.0 = 2.5$.

Parts 3 and 4 each entail two sub-parts. For example, for part 3, we must show that if t is a non-empty two-tree, then

$$\text{leaves}(t) = \frac{n(t) + 1}{2.0}$$

And if

$$\text{leaves}(t) = \frac{n(t) + 1}{2.0}$$

then t must be a non-empty two-tree.

All six parts of this theorem can be proved by induction on the height of t. As it turns out, most theorems about binary trees can be proved by induction on the *height* of the tree. The reason for this is that if t is a binary tree, then both leftTree(t) and rightTree(t) have height less than height(t), and so the Strong Form of the Principle of Mathematical Induction (see Section A2.5 of Appendix 2) often applies. For example, Example A2.5 of Appendix 2 has a proof of part 1. The proofs of the remaining parts are left as Exercises 9.13 and 9.14, with a hint for each exercise.

Suppose t is a full binary tree (possibly empty), then from part 4 of the Binary Tree Theorem, and the fact that any empty tree has height of -1, we have the equation

$$\frac{n(t) + 1}{2.0} = 2^{\text{height}(t)}$$

If we solve this equation for height(t), we get

$$\begin{aligned} \text{height}(t) &= \log_2((n(t) + 1)/2.0) \\ &= \log_2(n(t) + 1) - 1 \end{aligned}$$

So we can say that the height of a full tree is logarithmic in n, where n is the number of elements in the tree; we often use n instead of $n(t)$ when it is clear which tree we are referring to. Even if t is merely complete, its height is still logarithmic in n. See Concept Exercise 9.7. On the other hand, t could be a chain. A ***chain*** is a binary tree in which each non-leaf has exactly one child. For example, Figure 9.12 has an example of a binary tree that is a chain.

If t is a chain, then height(t) $= n(t) - 1$, so for chains the height is linear in n. Much of our work with trees in subsequent chapters will be concerned with maintaining logarithmic-in-n height and avoiding linear-in-n height. Basically, for inserting into or removing from certain kinds of binary trees whose height is logarithmic in n, worstTime(n) is logarithmic in n. That is why, in many applications, binary trees are preferable to lists. Recall that with both `ArrayList` objects and `LinkedList` objects, for inserting or removing at a specific index, worstTime(n) is linear in n.

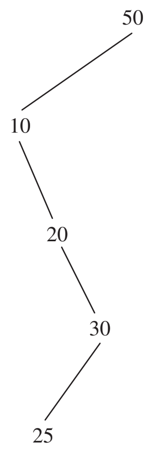


FIGURE 9.12 A binary tree that is a chain: each non-leaf has exactly one child

9.4 External Path Length

You may wonder why we would be interested in adding up all the root-to-leaf path lengths, but the following definition does have some practical value. Let t be a non-empty binary tree. E(t), the ***external path length*** of t, is the sum of the depths of all the leaves in t. For example, in Figure 9.13, the sum of the depths of the leaves is $2 + 4 + 4 + 4 + 5 + 5 + 1 = 25$.

The following lower bound on external path lengths yields an important result in the study of sorting algorithms (see Chapter 11).

External Path Length Theorem Let t be a binary tree with $k > 0$ leaves. Then

$$E(t) \geq (k/2) \text{ floor } (\log_2 k)$$

Proof It follows from the Binary Tree Theorem that if a nonempty binary tree is full and has height h, then the tree has 2^h leaves. And we can obtain any binary tree by ‘‘pruning’’ a full binary tree of the same height;

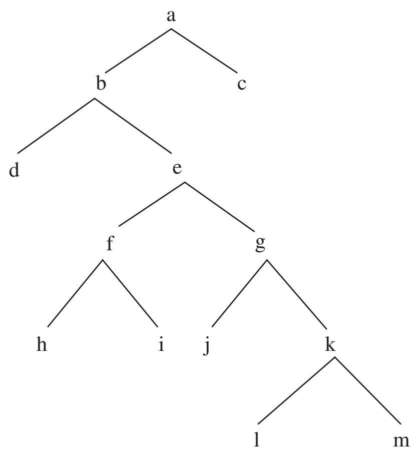


FIGURE 9.13 A binary tree whose external path length is 25

in so doing we reduce the number of leaves in the tree. So any non-empty binary tree of height h has no more than 2^h leaves. To put that in a slightly different way, if k is any positive integer, any nonempty binary tree of height floor($\log_2 k$) has no more than k leaves. (We have to use the *floor* function because the height must be an integer, but $log_2 k$ might not be an integer.)

Now suppose t is a nonempty binary tree whose height is floor($\log_2 k$) for some positive integer k. By the previous paragraph, t has no more than k leaves. How many of those leaves will be at level floor($\log_2 k$), the level farthest from the root? To answer that question, we ask how many leaves must be at a level less than floor($\log_2 k$). That is, how many leaves must there be in the subtree t' of t formed by removing all leaves at level floor($\log_2 k$)? The height of t' is floor($\log_2 k$) − 1. Note that

$$\begin{aligned}\text{floor}(\log_2 k) - 1 &= \text{floor}(\log_2 k - 1)\\ &= \text{floor}(\log_2 k - \log_2 2)\\ &= \text{floor}(\log_2 (k/2))\end{aligned}$$

By the previous paragraph, the total number of leaves in t' is no more than $k/2$. But every leaf in t that is at a level less than floor($\log_2 k$) is also a leaf in t'. And so there must be, at least, $k/2$ leaves at level floor($\log_2 k$).

Each of those $k/2$ leaves contributes floor($\log_2 k$) to the external path length, so we must have

$$E(t) \geq (k/2)\ \text{floor}\ (\log_2 k)$$

Note: This result is all we will need in Chapter 11, but at a cost of a somewhat more complicated proof, we could show that $E(t) \geq k \log_2 k$ for any non-empty *two-tree* with k leaves. (See Cormen, [2002] for details.)

9.5 Traversals of a Binary Tree

A ***traversal*** of a binary tree t is an algorithm that processes each element in t exactly once. In this section, we restrict our attention to algorithms only; there are no methods here. We make no attempt to declare a `BinaryTree` class or interface: it would not be flexible enough to support the variety of insertion and removal methods already in the Java Collections Framework. But the traversals we discuss in this section are related to code: specifically, to iterators. One of the iterators will turn up in Section 10.1.2.6, and two other iterators will appear in Chapter 15.

We identify four different kinds of traversal.

Traversal 1. inOrder Traversal: Left-Root-Right The basic idea of this recursive algorithm is that we first perform an inOrder traversal of the left subtree, then we process the root element, and finally, we perform an inOrder traversal of the right subtree. Here is the algorithm—assume that t is a binary tree:

```
inOrder (t)
{
  if (t is not empty)
  {
      inOrder (leftTree (t));
      process the root element of t;
      inOrder (rightTree (t));
  } // if
} // inOrder traversal
```

Let n represent the number of elements in the tree. Corresponding to each element there are 2 subtrees, so there will be $2n$ recursive calls to inOrder(t). We conclude that worstTime(n) is linear in n. Ditto for averageTime(n).

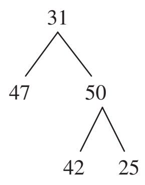


FIGURE 9.14 A binary tree

We can use this recursive description to list the elements in an inOrder traversal of the binary tree in Figure 9.14.

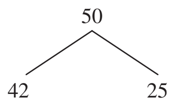
The tree *t* in Figure 9.14 is not empty, so we start by performing an inOrder traversal of leftTree(*t*), namely,

47

This one-element tree becomes the current version of *t*. Since its left subtree is empty, we process the root element of this *t*, namely 47. That completes the traversal of this version of *t* since rightTree(*t*) is empty. So now *t* again refers to the original tree. We next process *t*'s root element, namely,

31

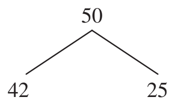
After that, we perform an inOrder traversal of rightTree(*t*), namely,



This becomes the current version of *t*. We start by performing an inOrder traversal of leftTree(*t*), namely,

42

Now this tree with one element becomes the current version of *t*. Since its left subtree is empty, we process *t*'s root element, 42. The right subtree of this *t* is empty. So we have completed the inOrder traversal of the tree with the single element 42, and now, once again, *t* refers to the binary tree with 3 elements:



We next process the root element of this version of *t*, namely,

50

Finally, we perform an inOrder traversal of rightTree(*t*), namely,

25

Since the left subtree of this single-element tree *t* is empty, we process the root element of *t*, namely 25. We are now done since *t*'s right subtree is also empty.

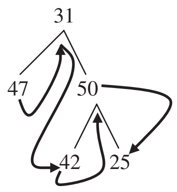


FIGURE 9.15 An inOrder traversal of a binary tree

The complete listing is

47 31 42 50 25

Figure 9.15 shows the original tree, with arrows to indicate the order in which elements are processed:

The inOrder traversal gets its name from the fact that, for a special kind of binary tree—a binary search tree—an inOrder traversal will process the elements in order. For example, Figure 9.16 has a binary search tree:

An inOrder traversal processes the elements of the tree in Figure 9.16 as follows:

25, 31, 42, 47, 50

In a binary search tree, all of the elements in the left subtree are less than the root element, which is less than all of the elements in the right subtree. What recursive property do you think will be part of the definition of a binary search tree so that an inOrder traversal processes the elements in order? **Hint:** the binary tree in Figure 9.17 is *not* a binary search tree:

We will devote Chapters 10 and 12 to the study of binary search trees.

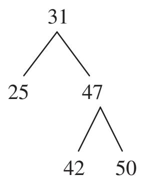


FIGURE 9.16 A binary search tree

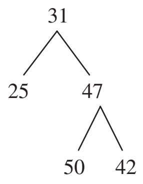


FIGURE 9.17 A binary tree that is not a binary search tree

Traversal 2. postOrder Traversal: Left-Right-Root The idea behind this recursive algorithm is that we perform postOrder traversals of the left and right subtrees before processing the root element. The algorithm, with t a binary tree, is:

```
postOrder (t)
{
    if (t is not empty)
    {
        postOrder (leftTree (t));
        postOrder (rightTree (t));
        process the root element of t;
    } // if
} // postOrder traversal
```

Just as with an inOrder traversal, the worstTime(n) for a postOrder traversal is linear in n because there are $2n$ recursive calls to postOrder(t).

Suppose we conduct a postOrder traversal of the binary tree in Figure 9.18.

A postOrder traversal of the binary tree in Figure 9.18 will process the elements in the path shown in Figure 9.19.

In a linear form, the postOrder traversal shown in Figure 9.19 is

A B C + ∗

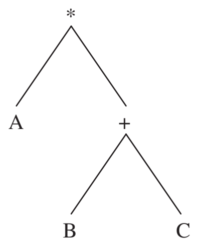


FIGURE 9.18 A binary tree

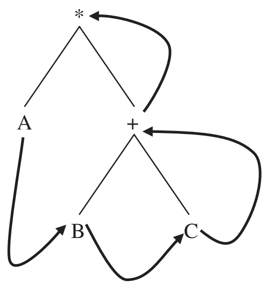


FIGURE 9.19 The path followed by a postOrder traversal of the binary tree in Figure 9.18

We can view the above binary tree as an ***expression tree***: each non-leaf is a binary operator whose operands are the associated left and right subtrees. With this interpretation, a postOrder traversal produces postfix notation.

Traversal 3. preOrder Traversal: Root-Left-Right Here we process the root element and then perform preOrder traversals of the left and right subtrees. The algorithm, with t a binary tree, is:

```
preOrder (t)
{
  if (t is not empty)
  {
       process the root element of t;
       preOrder (leftTree (t));
       preOrder (rightTree (t));
  } // if
} // preOrder traversal
```

As with the inOrder and postOrder algorithms, worstTime(n) is linear in n.

For example, a preOrder traversal of the binary tree in Figure 9.20 will process the elements in the order indicated in Figure 9.21.

If we linearize the path in Figure 9.21, we get

$* \ \ A + B \ \ C$

For an expression tree, a preOrder traversal produces prefix notation.

A search of a binary tree that employs a preOrder traversal is called a ***depth-first-search*** because the search goes to the left as deeply as possible before searching to the right. The search stops when (if) the

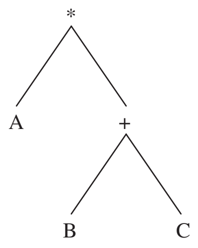


FIGURE 9.20 An expression tree

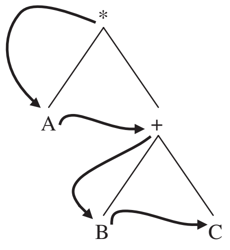


FIGURE 9.21 The path followed by a preOrder traversal of the elements in the expression tree of Figure 9.20

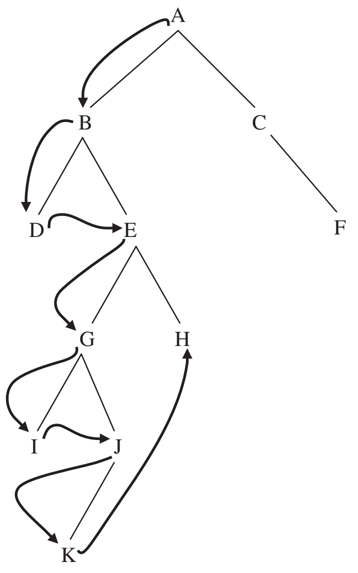


FIGURE 9.22 A depth-first search for H

element sought is found, so the traversal may not be completed. For an example of a depth-first search, Figure 9.22 shows a binary tree and the path followed by a depth-first search for H.

The *backtracking* strategy from Chapter 5 includes a depth-first search, but at each stage there may be more than two choices. For example, in the maze-search, the choices are to move north, east, south, or west. Because moving north is the first option, that option will be repeatedly applied until either the goal is reached or moving north is not possible. Then a move east will be taken, if possible, and then as many moves north as possible or necessary. And so on. In Chapter 15, we will re-visit backtracking for a generalization of binary trees.

Traversal 4. breadthFirst Traversal: Level-By-Level To perform a breadth-first traversal of a non-empty binary tree *t*, first process the root element, then the children of the root, from left to right, then the grandchildren of the root, from left to right, and so on.

For example, suppose we perform a breadth-first traversal of the binary tree in Figure 9.23.

The order in which elements would be processed in a breadthFirst traversal of the tree in Figure 9.23 is

A B C D E F G H I J K

One way to accomplish this traversal is to generate, level-by-level, a list of (references to) non-empty subtrees. We need to retrieve these subtrees in the same order they were generated so the elements can be processed level-by-level. What kind of collection allows retrievals in the same order as insertions? A queue! Here is the algorithm, with *t* a binary tree:

```
breadthFirst (t)
{
    // queue is a queue of (references to) binary trees
    // tree is a (reference to a) binary tree
```

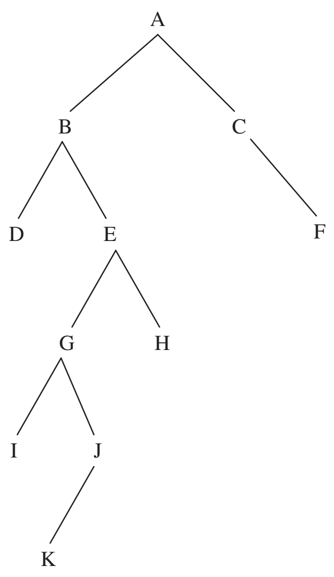


FIGURE 9.23 A binary tree

```
    if ($t$ is not empty)
    {
        queue.enqueue ($t$);
        while (queue is not empty)
        {
            tree = queue.dequeue();
            process tree's root;
            if (leftTree (tree) is not empty)
                queue.enqueue (leftTree (tree));
            if (rightTree (tree) is not empty)
                queue.enqueue (rightTree (tree));
        } // while
    } // if t not empty
} // breadthFirst traversal
```

During each loop iteration, one element is processed, so worstTime(n) is linear in n.

We used a queue for a breadth-first traversal because we wanted the subtrees retrieved in the same order they were saved (First-In, First-Out). With inOrder, postOrder, and preOrder traversals, the subtrees are retrieved in the reverse of the order they were saved in (Last-In, First-Out). For each of those three traversals, we utilized recursion, which, as we saw in Chapter 8, can be replaced with an iterative, stack-based algorithm.

We will encounter this type of traversal again in Chapter 15 when we study breadth-first traversals of structures less restrictive than binary trees. Incidentally, if we are willing to be more restrictive, specifically, if we require a complete binary tree, then the tree can be implemented with an array, and a breadth-first traversal is simply an iteration through the array. The root element is at index 0, the root's left child at index 1, the root's right child at index 2, the root's leftmost grandchild at index 3, and so on.

SUMMARY

A ***binary tree*** t is either empty or consists of an element, called the ***root element***, and two distinct binary trees, called the ***left subtree*** and ***right subtree*** of t. This is a recursive definition, and there are recursive definitions for many of the related terms: height, number of leaves, number of elements, two-tree, full tree, and so on. The inter-relationships among some of these terms is given by the

Binary Tree Theorem: For any non-empty binary tree t,

$$\text{leaves}(t) \leq \frac{n(t)+1}{2.0} \leq 2^{\text{height}(t)}$$

Equality holds for the first relation if and only if t is a two-tree.
Equality holds for the second relation if and only if t is a full tree.

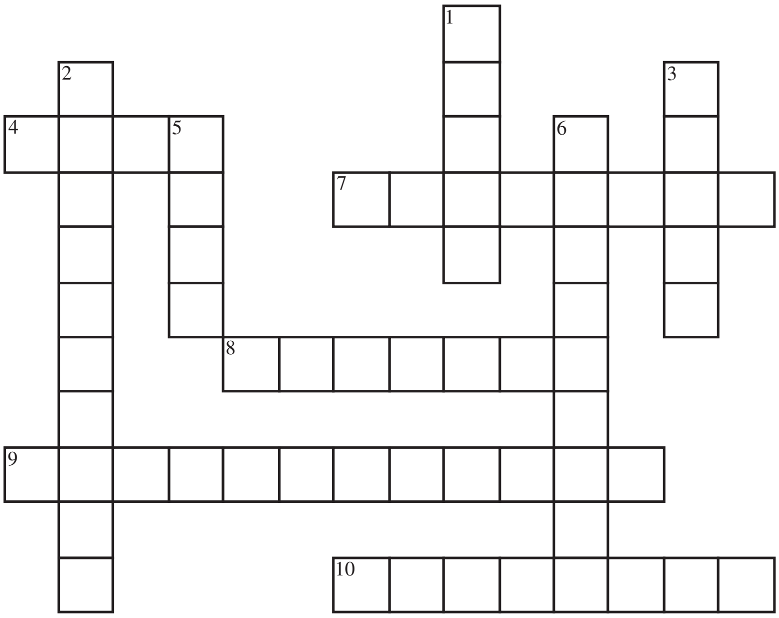
For a binary tree t, the ***external path length*** of t, written $E(t)$, is the sum of the distances from the root to the leaves of t. A lower bound for comparison-based sorting algorithms can be obtained from the

External Path Length Theorem: Let t be a binary tree with $k > 0$ leaves. Then

$$E(t) \geq (k/2) \text{ floor } (\log_2 k)$$

There are four commonly-used traversals of a binary tree: inOrder (recursively: left subtree, root element, right subtree), postOrder (recursively: left subtree, right subtree, root element), preorder (recursively: root element, left subtree, right subtree) and breadth-first (that is, starting at the root, level-by-level, and left-to-right at each level).

CROSSWORD PUZZLE



www.CrosswordWeaver.com

ACROSS

4. In a binary tree, an element with no children.

7. The ______ path length of a binary tree is the sum of the depths of all the leaves in the tree.

8. A binary tree *t* is a ______ if *t* has at most one element or leftTree(*t*) and rightTree(*t*) are non-empty two-trees.

9. The only traversal in this chapter described non-recursively.

10. A binary tree is ________ if *t* is full through the next-to-lowest level and all of the leaves at the lowest level ara as far to the left as possible.

DOWN

1. A synonym of "level," the function that calculates the length of the path from the root to a given element.

2. Another name for "preOrder" traversal.

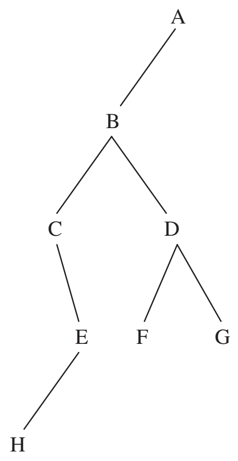
3. A __________ is a binary tree in which each non-leaf has exactly one child.

5. A binary tree *t* is __________ if *t* is a two-tree with all of its leaves on the same level.

6. An algorithm that processes each element in a binary tree exactly once.

CONCEPT EXERCISES

9.1 Answer the questions below about the following binary tree:



a. What is the root element?

b. How many elements are in the tree?

c. How many leaves are in the tree?

d. What is the height of the tree?

e. What is the height of the left subtree?

f. What is the height of the right subtree?

g. What is the level of F?

h. What is the depth of C?

i. How many children does C have?

j. What is the parent of F?

k. What are the descendants of B?

l. What are the ancestors of F?

m. What would the output be if the elements were written out during an inOrder traversal?

n. What would the output be if the elements were written out during a postOrder traversal?

o. What would the output be if the elements were written out during a preOrder traversal?

p. What would the output be if the elements were written out during a breadth-first traversal?

9.2 **a.** Construct a binary tree of height 3 that has 8 elements.

b. Can you construct a binary tree of height 2 that has 8 elements?

c. For n going from 1 to 20, determine the minimum height possible for a binary tree with n elements.

d. Based on your calculations in part c, try to develop a formula for the minimum height possible for a binary tree with n elements, where n can be any positive integer.

e. Use the Principle of Mathematical Induction (Strong Form) to prove the correctness of your formula in part d.

9.3 **a.** What is the maximum number of leaves possible in a binary tree with 10 elements? Construct such a tree.

b. What is the minimum number of leaves possible in a binary tree with 10 elements? Construct such a tree.

9.4 **a.** Construct a two-tree that is not complete.

b. Construct a complete tree that is not a two-tree.

c. Construct a complete two-tree that is not full.

d. How many leaves are there in a two-tree with 17 elements?

e. How many leaves are there in a two-tree with 731 elements?

f. A non-empty two-tree must always have an odd number of elements. Why?
Hint: Use the Binary Tree Theorem and the fact that the number of leaves must be an integer.

g. How many elements are there in a full binary tree of height 4?

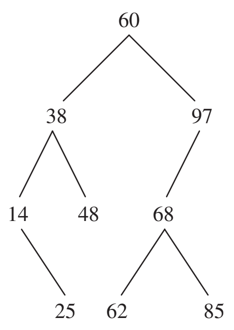
h. How many elements are there in a full binary tree of height 12?

i. Use induction (original form) on the height of the tree to show that any full binary tree is a two tree.

j. Use the results from part i and the Binary Tree Theorem to determine the number of leaves in a full binary tree with 63 elements.

k. Construct a complete two-tree that is not full, but in which the heights of the left and right subtrees are equal.

9.5 For the following binary tree, show the order in which elements would be visited for an inOrder, postOrder, preOrder, and breadthFirst traversal.



9.6 Show that a binary tree with n elements has $2n + 1$ subtrees (including the entire tree). How many of these subtrees are empty?

9.7 Show that if t is a complete binary tree, then

$$\text{height}(t) = \text{floor}(\log_2(n(t)))$$

Hint: Let t be a complete binary tree of height $k \geq 0$, and let $t1$ be a full binary tree of height $k - 1$. Then $n(t1) + 1 \leq n(t)$. Use Part 4 of the Binary Tree Theorem to show that $\text{floor}(\log_2(n(t1) + 1)) = k$, and use Part 1 of the Binary Tree Theorem to show that $\text{floor}(\log_2(n(t))) < k + 1$.

9.8 The Binary Tree Theorem is stated for non-empty binary trees. Show that parts 1, 2, and 4 hold even for an empty binary tree.

9.9 Give an example of a non-empty binary tree that is *not* a two-tree but

$$\text{leaves}(t) = (n(t) + 1) / 2$$

Hint: The denominator is 2, not 2.0, so integer division is performed.

9.10 Let t be a non-empty tree. Show that if

$$\text{leaves}(t) = \frac{n(t) + 1}{2.0}$$

then either both subtrees of t are empty or both subtrees of t are non-empty.
Note: Do not use Part 3 of the Binary Tree Theorem. This exercise can be used in the proof of Part 3.

9.11 Show that in any complete binary tree t, at least half of the elements are leaves.

Hint: if t is empty, there are no elements, so the claim is vacuously true. If the leaf at the highest index is a right child, then t is a two-tree, and the claim follows from part 3 of the Binary Tree Theorem. Otherwise, t was formed by adding a left child to *the* complete two-tree with $n(t) - 1$ elements.

9.12 Compare the `inOrder` traversal algorithm in Section 9.5 with the `move` method from the Towers of Hanoi application in Section 5.4 of Chapter 5. They have the same structure, but worstTime(n) is linear in n for the `inOrder` algorithm and exponential in n for the `move` method. Explain.

9.13 Let t be a nonempty binary tree. Use the Strong Form of the Principle of Mathematical Induction to prove each of the following parts of the Binary Tree Theorem:

a. $\dfrac{n(t)+1}{2.0} \leq 2^{\text{height}(t)}$

b. If t is a two-tree, then leaves$(t) = \dfrac{n(t)+1}{2.0}$

c. If t is a full tree, then $\dfrac{n(t)+1}{2.0} = 2^{\text{height}(t)}$

Hint: The outline of the proof is the same as in Example A2.5 of Appendix 2.

9.14 Let t be a nonempty binary tree. Use the Strong Form of the Principle of Mathematical Induction to prove each of the following parts of the Binary Tree Theorem:

a. If leaves$(t) = \dfrac{n(t)+1}{2.0}$ then t is a two-tree.

b. If $\dfrac{n(t)+1}{2.0} = 2^{\text{height}(t)}$ then t is a full tree.

Hint: The proof for both parts has the same outline. For example, here is the outline for part a:

For $h = 0, 1, 2, \ldots$, let S_h be the statement

If t is a binary tree of height h and leaves$(t) = \dfrac{n(t)+1}{2.0}$

then t is a two-tree.

In the inductive case, let h be any nonnegative integer and assume that $S_0, S_1, \ldots, S_h$ are all true. To show that S_{h+1} is true, let t be a binary tree of height $h+1$ such that

$$\text{leaves}(t) = \frac{n(t)+1}{2.0}$$

First, show that

$$\text{leaves}(\text{leftTree}(t)) + \text{leaves}(\text{rightTree}(t)) = \frac{n(\text{leftTree}(t))+1}{2.0} + \frac{n(\text{rightTree}(t))+1}{2.0}$$

For any non-negative integers a, b, c, and d, if

$$a + b = c + d \text{ and } a \leq c \text{ and } b \geq d, \text{ then } a = c \text{ and } b = d.$$

Then, using Exercise 8.10, show that leftTree(t) and rightTree(t) are two-trees. Then, using Exercise 8.8, show that both leftTree(t) and rightTree(t) are nonempty. Conclude, from the definition of a two-tree, that t must be a two-tree.

9.15 For any positive integer n, we can construct a binary tree of n elements as follows:

at level 0, there will be 1 element (the root);

at level 1, there will be 2 elements;

at level 2, there will be 3 elements;

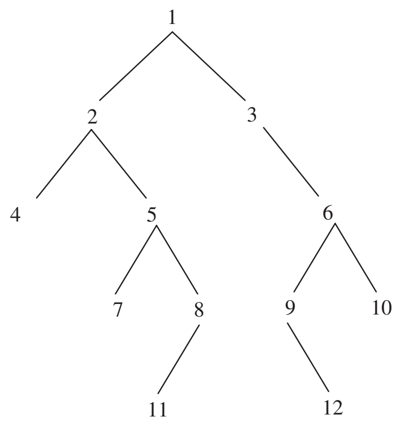
at level 3, there will be 4 elements;

at level 4, there will be 5 elements;

. . .

At the level farthest from the root, there will be just enough elements so the entire tree will have n elements.

For example, if $n = 12$, we can construct the following tree:



Provide a Θ estimate of the height as a function of n.

Hint: Since Θ is just an estimate, we can ignore the elements at the lowest level. We seek an integer h such that

$$1 + 2 + 3 + 4 + \ldots + (h + 1) = n$$

See Example A2.1 in Appendix 2.

Note: This exercise is contrived but, in fact, the Θ estimate of the average height of a binary tree is the same as the answer to this exercise (see Flajolet, [1981]).

This page intentionally left blank

Binary Search Trees

CHAPTER 10

In Chapter 9, you studied an important conceptual tool: the binary tree. This chapter presents the binary search tree data type and a corresponding data structure, the `BinarySearchTree` class. `BinarySearchTree` objects are valuable collections because they require only logarithmic time, on average, for inserting, removing, and searching (but linear time in the worst case). This performance is far better than the linear average-case time for insertions, removals and searches in an array, `ArrayList` or `LinkedList` object. For example, if $n = 1,000,000,000$, $\log_2 n < 30$.

The `BinarySearchTree` class is not part of the Java Collections Framework. The reason for this omission is that the framework already includes the `TreeSet` class, which boasts logarithmic time for inserting, removing, and searching even in the worst case. The `BinarySearchTree` class requires linear-in-n time, in the worst case, for those three operations. So the `BinarySearchTree` class should be viewed as a "toy" class that is simple enough for you to play with and will help you to better understand the `TreeSet` class. The implementation of the `TreeSet` class is based on a kind of "balanced" binary search tree, namely, the red-black tree.

To further prepare you for the study of red-black trees and the `TreeSet` class, this chapter also explains what it means to say that a binary search tree is balanced. To add some substance to that discussion, we introduce the AVL tree data type, which is somewhat simpler to understand than the red-black tree data type.

CHAPTER OBJECTIVES

1. Compare the time efficiency of the `BinarySearchTree` class's insertion, removal and search methods to that of the corresponding methods in the `ArrayList` and `LinkedList` classes.
2. Discuss the similarities and differences of the `BinarySearchTree` class's `contains` method and the `binarySearch` methods in the `Arrays` and `Collections` classes.
3. Explain why the `BinarySearchTree` class's `remove` method and the `TreeIterator` class's `next` method are somewhat difficult to define.
4. Be able to perform each of the four possible rotations.
5. Understand why the height of an AVL tree is always logarithmic in n.

10.1 Binary Search Trees

We start with a recursive definition of a binary search tree:

A ***binary search tree*** *t* is a binary tree such that either *t* is empty or

1. each element in leftTree(*t*) is less than the root element of *t*;
2. each element in rightTree(*t*) is greater than the root element of *t*;
3. both leftTree(*t*) and rightTree(*t*) are binary search trees.

Figure 10.1 shows a binary search tree.

An inOrder traversal of a binary search tree accesses the items in increasing order. For example, with the binary search tree in Figure 10.1, an inOrder traversal accesses the following sequence:

15 25 28 30 32 36 37 50 55 59 61 68 75

As we have defined a binary search tree, duplicate elements are not permitted. Some definitions have "less than or equal to" and "greater than or equal to" in the above definition. For the sake of consistency with some classes in the Java Collections Framework that are based on binary search trees, we opt for strictly less than and strictly greater than.

Section 10.1.1 describes the `BinarySearchTree` class. As we noted at the beginning of this chapter, this is a "toy" class that provides a gentle introduction to related classes in the Java Collections Framework. Much of the code for the `BinarySearchTree` class is either identical to or a simplification of code in the `TreeMap` class of the Java Collections Framework. Overall, the `BinarySearchTree` class's performance is not good enough for applications—because its worst-case height is linear in *n*—but you will be asked to add new methods and to modify existing methods.

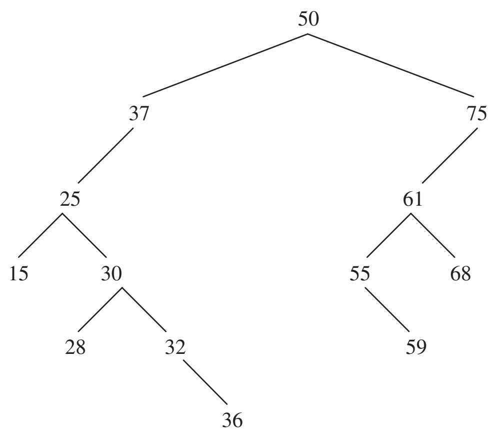


FIGURE 10.1 A binary search tree

10.1.1 The BinarySearchTree Implementation of the Set Interface

We will study the binary-search-tree data type through the method specifications of the BinarySearchTree class. The BinarySearchTree class is not part of the Java Collections Framework, but it implements the Collection interface. In fact, the BinarySearchTree class implements a slight extension of the Collection interface: the Set interface. The Set interface does not provide any new methods. The only difference between the Collection and Set interfaces is that duplicate elements are not allowed in a Set, and this affects the specifications of the default constructor and the add method in the BinarySearchTree class. We will see these differences shortly.

Another feature that distinguishes the BinarySearchTree class from previous Collection classes we have seen is that the elements in a BinarySearchTree must be maintained in order. For simplicity, we assume that the elements are instances of a class that implements the Comparable interface, which was introduced in Section 5.5. Thus, **null** elements are not allowed in a BinarySearchTree object. Here is that interface:

```
public interface Comparable
{
     /**
      *  Returns an int less than, equal to or greater than 0, depending on
      *  whether the calling object is less than, equal to or greater than a
      *  specified object.
      *
      *  @param obj – the specified object that the calling object is compared to.
      *
      *  @return an int value less than, equal to, or greater than 0, depending on
      *          whether the calling object is less than, equal to, or greater than
      *          obj, respectively.
      *
      *  @throws ClassCastException – if the calling object and obj are not in the
      *          same class.
      *
      */
      public int compareTo (T obj)
} // interface Comparable
```

From now on, when we refer to the "natural" order of elements, we assume that the element's class implements the Comparable interface; then the natural order is the order imposed by the compareTo method in the element's class. For example, the Integer class implements the Comparable interface, and the "natural" order of Integer objects is based on the numeric comparison of the underlying **int** values. To illustrate this ordering, suppose we have the following:

```
Integer myInt = 25;

System.out.println (myInt.compareTo (107));
```

The output will be less than 0 because 25 is less than 107. Specifically, the output will be −1 because when the compareTo method is used for numeric comparisons, the result is either −1, 0, or 1, depending on whether the calling object is less than, equal to, or greater than the argument.

For a more involved example, the following `Student` class has `name` and `gpa` fields. The `compareTo` method will use the alphabetical ordering of names; for equal names, the ordering will be by *decreasing* grade point averages. For example,

```
new Student ("Lodato", 3.8).compareTo (new Student ("Zsoldos", 3.5))
```

returns −1 because “Lodato” is, alphabetically, less than “Zsoldos”. But

```
new Student ("Dufresne", 3.4).compareTo (new Student ("Dufresne", 3.6))
```

returns 1 because 3.4 is less than 3.6, and the return value reflects decreasing order. Similarly,

```
new Student ("Dufresne", 3.8).compareTo (new Student ("Dufresne", 3.6))
```

returns −1, because 3.8 is greater than 3.6.

Here is the `Student` class:

```
public class Student implements Comparable<Student>
{
   public final double DELTA = 0.0000001;

   protected String name;

   protected double gpa;

   public Student() { }

   /**
    *  Initializes this Student object from a specified name and gpa.
    *
    *  @param name – the specified name.
    *  @param gpa – the specified gpa.
    *
    */
    public Student (String name, double gpa)
    {
          this.name = name;
          this.gpa = gpa;
    } // constructor


   /**
    *  Compares this Student object with a specified Student object.
    *  The comparison is alphabetical; for two objects with the same name,
    *  the comparison is by grade point averages.
    *
    *  @param otherStudent – the specified Student object that this Student
    *         object is being compared to.
    *
    *  @return -1, if this Student object's name is alphabetically less than
    *              otherStudent's name, or if the names are equal and this
    *              Student object's grade point average is at least DELTA
    *              greater than otherStudent's grade point average;
```

```
 *           0, if this Student object's name the same as
 *              otherStudent's name and the grade point
 *              averages are within DELTA;
 *           1, if this Student object's name is alphabetically greater
 *              than otherStudent's name, or if the names are equal and
 *              this Student object's grade point average is at least DELTA
 *              less than otherStudent's grade point average;
 *
 */
 public int compareTo (Student otherStudent)
 {
      final double DELTA = 0.0000001;

      if (name.compareTo (otherStudent.name) < 0)
             return -1;
      if (name.compareTo (otherStudent.name) > 0)
             return 1;
      if (gpa - otherStudent.gpa > DELTA)
             return -1;
      if (otherStudent.gpa - gpa > DELTA)
             return 1;
      return 0;
 } // method compareTo


/**
 *  Determines if this Student object's name and grade point average are
 *  the same as some specified object's.
 *
 *  @param obj - the specified object that this Student object is being
 *   compared to.
 *
 *  @return true - if obj is a Student object and this Student object has the
 *   same name and almost (that is, within DELTA) the same point average as obj.
 *
 */
 public boolean equals (Object obj)
 {
      if (! (obj instanceof Student))
             return false;
      return this.compareTo ((Student)obj) == 0;
 } // method equals


/**
 *  Returns a String representation of this Student object.
 *
 *  @return a String representation of this Student object: name, blank,
        *            grade point average.
        *
        */
        public String toString()
        {
               return name + " " + gpa;
        } // method toString

    } // class Student
```

Rather than starting from scratch, we will let the BinarySearchTree class extend some class already in the Framework. Then we need implement only those methods whose definitions are specific to the BinarySearchTree class. The AbstractSet class is a good place to start. That class has garden-variety implementations for many of the Set methods: isEmpty, toArray, clear, and the bulk operations (addAll, containsAll, removeAll and retainAll). So the class heading is

```
    public class BinarySearchTree<E> extends AbstractSet<E>
```

Figure 10.2 shows the relationships among the classes and interfaces we have discussed so far.

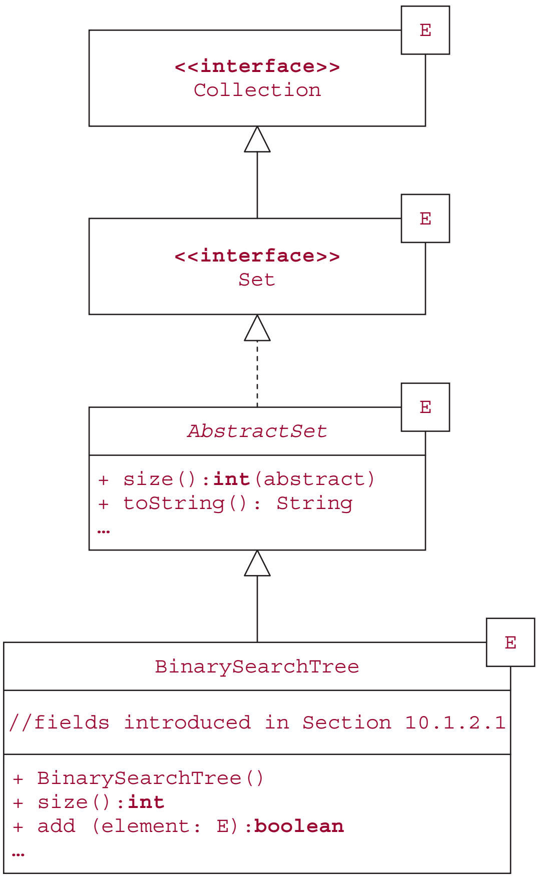


FIGURE 10.2 A UML diagram that includes part of the BinarySearchTree class

Here are the method specifications for the methods we will explicitly implement:

```
/**
 *  Initializes this BinarySearchTree object to be empty, to contain only elements
 *  of type E, to be ordered by the Comparable interface, and to contain no
 *  duplicate elements.
 *
 */
public BinarySearchTree()


/**
 * Initializes this BinarySearchTree object to contain a shallow copy of
 * a specified BinarySearchTree object.
 * The worstTime(n) is O(n), where n is the number of elements in the
 * specified BinarySearchTree object.
 *
 * @param otherTree - the specified BinarySearchTree object that this
 *       BinarySearchTree object will be assigned a shallow copy of.
 *
 */
public BinarySearchTree (BinarySearchTree<? extends E> otherTree)


/**
 *  Returns the size of this BinarySearchTree object.
 *
 * @return the size of this BinarySearchTree object.
 *
 */
public int size( )


/**
 *  Returns an iterator positioned at the smallest element in this
 *  BinarySearchTree object.
 *
 *  @return an iterator positioned at the smallest element (according to
 *          the element class's implementation of the Comparable
 *          interface) in this BinarySearchTree object.
 *
 */
public Iterator<E> iterator()


/**
 *  Determines if there is an element in this BinarySearchTree object that
 *  equals a specified element.
 *  The worstTime(n) is O(n) and averageTime(n) is O(log n).
 *
 *  @param obj - the element sought in this BinarySearchTree object.
 *
```

```
     *  @return true - if there is an element in this BinarySearchTree object that
     *                 equals obj; otherwise, return false.
     *
     *  @throws ClassCastException - if obj is not null but cannot be compared to the
     *                 elements already in this BinarySearchTree object.
     *  @throws NullPointerException - if obj is null.
     *
     */
    public boolean contains (Object obj)



    /**
     *  Ensures that this BinarySearchTree object contains a specified element.
     *  The worstTime(n) is O(n) and averageTime(n) is O(log n).
     *
     *  @param element - the element whose presence is ensured in this
     *         BinarySearchTree object.
     *
     *  @return true - if this BinarySearchTree object changed as a result of this
     *          method call (that is, if element was actually inserted); otherwise,
     *          return false.
     *
     *  @throws ClassCastException - if element is not null but cannot be compared
     *          to the elements of this BinarySearchTree object.
     *  @throws NullPointerException - if element is null.
     *
     */
    public boolean add (E element)


    /**
     *  Ensures that this BinarySearchTree object does not contain a specified
     *  element.
     *  The worstTime(n) is O(n) and averageTime(n) is O(log n).
     *
     *  @param obj - the object whose absence is ensured in this
     *         BinarySearchTree object.
     *
     *  @return true - if this BinarySearchTree object changed as a result of this
     *          method call (that is, if obj was actually removed); otherwise,
     *          return false.
     *
     *  @throws ClassCastException - if obj is not null but cannot be compared to the
     *          elements of this BinarySearchTree object.
     *  @throws NullPointerException - if obj is null.
     *
     */
    public boolean remove (Object obj)
```

These method specifications, together with the method specifications of methods not overridden from the `AbstractSet` class, constitute the abstract data type Binary Search Tree: all that is needed for *using* the `BinarySearchTree` class. For example, here is a small class that creates and manipulates three `BinarySearchTree` objects: two of `String` elements and one of `Student` elements (for the `Student` class declared earlier in this section):

```
public class BinarySearchTreeExample
{
   public static void main (String[ ] args)
   {
        new BinarySearchTreeExample().run();
   } // method main

   public void run()
   {
          BinarySearchTree<String> tree1 = new BinarySearchTree<String>();

          tree1.add ("yes");
          tree1.add ("no");
          tree1.add ("maybe");
          tree1.add ("always");
          tree1.add ("no");   // not added: duplicate element
          if (tree1.remove ("often"))
              System.out.println ("How did that happen?");
          else
              System.out.println (tree1.remove ("maybe"));
          System.out.println (tree1);

          BinarySearchTree<String> tree2 =
                       new BinarySearchTree<String> (tree1);

          System.out.println (tree2);

          BinarySearchTree<Student> tree3 =
                       new BinarySearchTree<Student>();

          tree3.add (new Student ("Jones", 3.17));
          tree3.add (new Student ("Smith", 3.82));
          tree3.add (new Student ("Jones", 3.4));
          if (tree3.contains (new Student ("Jones", 10.0 - 6.6)))
              System.out.println ("The number of elements in tree3 is " +
                                  tree3.size());
          System.out.println (tree3);
   } // method run

} // class BinarySearchTreeExample
```

The output is

```
true
[always, no, yes]
```

```
[always, no, yes]
The number of elements in tree3 is 3
[Jones 3.4, Jones 3.17, Smith 3.82]
```

Here is part of the `BinarySearchTreeTest` class:

```
protected BinarySearchTree<String> tree;

@Before
public void RunBeforeEachTest()
{
    tree = new BinarySearchTree<String>();
} // method RunBeforeEachTest

@Test
public void testAdd()
{
    tree.add ("b");
    tree.add ("a");
    tree.add ("c");
    tree.add ("e");
    tree.add ("c");
    tree.add ("d");
    assertEquals ("[a, b, c, d, e]", tree.toString());
} // method testAdd

@Test
public void testContains()
{
    tree.add ("a");
    tree.add ("b");
    tree.add ("c");
    assertEquals (true, tree.contains ("a"));
    assertEquals (true, tree.contains ("b"));
    assertEquals (true, tree.contains ("c"));
    assertEquals (false, tree.contains ("x"));
    assertEquals (false, tree.contains (""));
} // method testContains

@Test (expected = NullPointerException.class)
public void testContainsNull()
{
    tree.add ("a");
    tree.add ("b");
    tree.add ("c");
    tree.contains (null);
} // method testContainsNull

@Test
public void testRemove()
{
    tree.add ("b");
    tree.add ("a");
    tree.add ("c");
    assertEquals (true, tree.remove ("b"));
    assertEquals (2, tree.size());
    assertEquals (false, tree.remove ("b"));
    assertEquals (2, tree.size());
} // method testRemove
```

The complete test class, available from the book's website, includes an override of the `AbstractSet` class's `equals (Object obj)` method to return **`false`** when comparing two `BinarySearchTree` objects that have the same elements but different structures.

10.1.2 Implementation of the `BinarySearchTree` Class

In this section, we will develop an implementation of the `BinarySearchTree` class. To confirm that Binary Search Tree is an abstract data type that has several possible implementations, you will have the opportunity in Project 10.1 to develop an array-based implementation.

The only methods we need to implement are the seven methods specified in Section 10.1.1. But to implement the `iterator()` method, we will need to develop a class that implements the `Iterator` interface, with `hasNext()`, `next()` and `remove()` methods. So we will create a `TreeIterator` class embedded within the `BinarySearchTree` class.

The definitions of the default constructor, `size()`, `iterator()`, and `hasNext()` are one-liners. Also, the definition of the copy constructor is fairly straightforward, and the `TreeIterator` class's `remove` method is almost identical to the `BinarySearchTree` class's `remove` method. But the remaining four methods—`contains`, `add`, and `remove` in `BinarySearchTree`, and `next` in `TreeIterator`—get to the heart of how the implementation of the `BinarySearchTree` class differs from that of, say, the `LinkedList` class.

10.1.2.1 Fields and Nested Classes in the `BinarySearchTree` Class

What fields and embedded classes should we have in the `BinarySearchTree` class? We have already noted the need for a `TreeIterator` class. From the fact that a binary search tree is a binary tree—with a root item and two subtrees—and from our earlier investigation of the `LinkedList` class, we can see the value of having an embedded `Entry` class. The only fields in the `BinarySearchTree` class are:

```
protected Entry<E> root;

protected int size;
```

The `Entry` class, as you may have expected, has an `element` field, of type (reference to) `E`, and `left` and `right` fields, of type (reference to) `Entry<E>`. To facilitate going back up the tree during an iteration, the `Entry` class will also have a `parent` field, of type (reference to) `Entry<E>`. Figure 10.3 shows the representation of a `BinarySearchTree` object with elements of type (reference to) `String`. To simplify the figure, we pretend that the elements are of type `String` instead of reference to `String`.

Here is the nested `Entry` class:

```
protected static class Entry<E>
{
    protected E element;

    protected Entry<E> left = null,
```

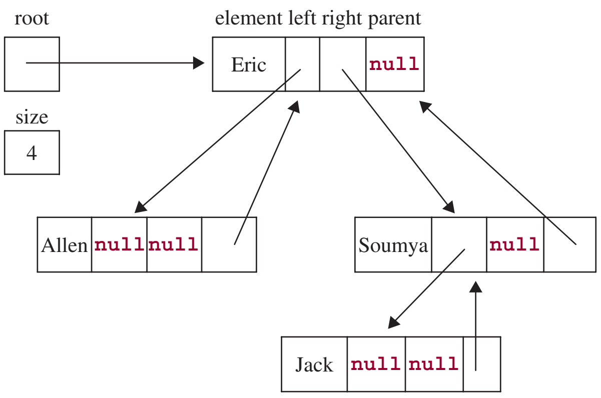


FIGURE 10.3 A `BinarySearchTree` object with four elements

```
                        right = null,
                        parent;
        /**
         *  Initializes this Entry object.
         *
         *  This default constructor is defined for the sake of subclasses of
         *  the BinarySearchTree class.
         */
        public Entry() { }

        /**
         *  Initializes this Entry object from element and parent.
         *
         */
        public Entry (E element, Entry<E> parent)
        {
               this.element = element;
               this.parent = parent;
        } // constructor

    } // class Entry
```

Recall, from Section 7.3.5, that the `Entry` class nested in the Java Collection Framework's `LinkedList` class was given the **`static`** modifier. The `BinarySearchTree`'s nested `Entry` class is made static for the same reason: to avoid the wasted time and space needed to maintain a reference back to the enclosing object.

Now that we have declared the fields and one of the nested classes, we are ready to tackle the `BinarySearchTree` method definitions.

10.1.2.2 Implementation of Simple Methods in the `BinarySearchTree` Class

We can immediately develop a few method definitions:

```
    public BinarySearchTree()
    {
         root = null;
         size = 0;
    } // default constructor
```

We could have omitted the assignments to the `root` and `size` fields because each field is given a default initial value (**`null`** for a reference, 0 for an **`int`**, **`false`** for a **`boolean`**, and so on) just prior to the invocation of the constructor. But explicit assignments facilitate understanding; default initializations do not.

```
    public int size()
    {
          return size;
    } // method size()


    public Iterator<E> iterator()
    {
          return new TreeIterator();
    } // method iterator
```

It is easy enough to define a copy constructor that iterates through `otherTree` and adds each element from `otherTree` to the calling object. But then the iteration over `otherTree` will be inOrder, so the new tree will be a chain, and worstTime(n) will be quadratic in n. To obtain linear-in-n time, we construct the new tree one entry at a time, starting at the root. Because each entry has an element, a parent, a left child, and a right child, we create the new entry from `otherTree`'s entry and the new entry's parent. The new entry's left (or right) child is then copied recursively from `otherTree`'s left (or right) child and from the new entry—the parent of that child.

We start with a wrapper method that calls the recursive method:

```
    public BinarySearchTree (BinarySearchTree<? extends E> otherTree)
    {
          root = copy (otherTree.root, null);
          size = otherTree.size;
    } // copy constructor


    protected Entry<E> copy (Entry<<? extends E> p, Entry<E> parent)
    {
          if (p != null)
          {
                Entry<E> q = new Entry<E> (p.element, parent);
                q.left = copy (p.left, q);
                q.right = copy (p.right, q);
                return q;
          } // if
          return null;
    } // method copy
```

10.1.2.3 Definition of the `contains` Method

The elements in a binary search tree are stored in the "natural" order, that is, the order imposed by the `compareTo` method in the element class. The definition of the `contains (Object obj)` method takes advantage of that fact that by moving down the tree in the direction of where `obj` is or belongs. Specifically, an `Entry` object `temp` is initialized to `root` and then `obj` is compared to `temp.element` in a loop. If they are equal, **`true`** is returned. If `obj` is less than `temp.element`, `temp` is replaced with `temp.left`. Otherwise, `temp` is replaced with `temp.right`. The loop continues until `temp` is **`null`**. Here is the complete definition:

```
public boolean contains (Object obj)
{
     Entry<E> temp = root;

     int comp;

     if (obj == null)
            throw new NullPointerException();

     while (temp != null)
     {
            comp = ((Comparable)obj).compareTo (temp.element);
            if (comp == 0)
                   return true;
            else if (comp < 0)
                   temp = temp.left;
            else
                   temp = temp.right;
     } // while
     return false;
} // method contains
```

The cast of `obj` to a `Comparable` object is necessary for the compiler because the `Object` class does not have a `compareTo` method.

How long does this method take? For this method, indeed, for all of the remaining methods in this section, the essential feature for estimating worstTime(n) or averageTime(n) is the *height* of the tree. Specifically, for the `contains` method, suppose the search is successful; a similar analysis can be used for an unsuccessful search. In the worst case, we will have a chain, and will be seeking the leaf. For example, suppose we are seeking 25 in the binary search tree in Figure 10.4.

In such a case, the number of loop iterations is equal to the height of the tree. In general, if n is the number of elements in the tree, and the tree is a chain, the height of the tree is $n - 1$, so worstTime(n) is linear in n for the `contains` method.

We now determine averageTime(n) for a successful search. Again, the crucial factor is the height of the tree. For binary search trees constructed through random insertions and removals, the average height H is logarithmic in n—(see Cormen, [2002]). The `contains` method starts searching at level 0, and each loop iteration descends to the next lower level in the tree. Since averageTime(n) requires no more than H iterations, we immediately conclude that averageTime(n) is O(log n). That is, averageTime(n) is less than or equal to some function of log n.

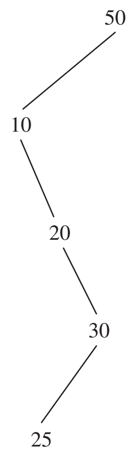


FIGURE 10.4 A binary search tree

To establish that averageTime(n) is logarithmic in n, we must also show that averageTime(n) is greater than or equal to some function of log n. The average—over all binary search trees—number of iterations is greater than or equal to the average number of iterations for a complete binary search tree with n elements. In a complete binary tree t, at least half of the elements are leaves (see Concept Exercise 9.11), and the level of each leaf is at least height(t) $-$ 1. So the average number of iterations by the `contains` method must be at least (height(t) $-$ 1)/2, which, by Concept Exercise 9.7, is (floor($\log_2(n(t))$) $-$ 1)/2. That is, the average number of iterations for the `contains` method is greater than or equal to a function of log n. So averageTime(n) is greater than or equal to some function of log n.

We conclude from the two previous paragraphs that averageTime(n) is logarithmic in n. Incidentally, that is why we defined the `contains` method above instead of inheriting the one in the `AbstractColl ection` class (the superclass of `AbstractSet`). For that version, an iterator loops through the elements in the tree, starting with the smallest, so its averageTime(n) is linear in n.

The binary search tree gets its name from the situation that arises when the `contains` method is invoked on a full tree. For then the `contains` method accesses the same elements, in the same order, as a binary search of an array with the same elements. For example, the root element in a full binary search tree corresponds to the middle element in the array.

You may have been surprised (and even disappointed) that the definition of the `contains` method was not recursive. Up to this point in our study of binary trees, most of the concepts—including binary search tree itself—were defined recursively. But when it comes to method definitions, looping is the rule rather than the exception. Why is that? The glib answer is that `left` and `right` are of type `Entry`, not of type `BinarySearchTree`, so we cannot call

```
left.contains(obj)  // illegal
```

But we can make `contains` a wrapper method for a protected, recursive `containsElement` method:

```
public boolean contains (Object obj)
{
   return containsElement (root, obj);
} // method contains
```

```
protected boolean containsElement (Entry<E> p, Object obj)
{
      if (p == null)
           return false;
      int comp = ((Comparable)obj).compareTo (p.element);

      if (comp == 0)
           return true;
      if (comp < 0)
           return containsElement (p.left, obj);
      return containsElement (p.right, obj);
} // method containsElement
```

This recursive version would be nominally less efficient—in both time and space—than the iterative version. And it is this slight difference that sinks the recursive version. For the iterative version is virtually identical to one in the `TreeMap` class, part of the Java Collections Framework, where efficiency is prized above elegance. Besides, some of the luster of recursion is diminished by the necessity of having a wrapper method.

10.1.2.4 Definition of the `add` Method

The definition of the `add (E element)` method is only a little more complicated than the definition of `contains (Object obj)`. Basically, the `add` method starts at the root and branches down the tree searching for the element; if the search fails, the element is inserted as a leaf.

Specifically, if the tree is empty, we construct a new `Entry` object and initialize that object with the given `element` and a **`null`** parent, then increment the `size` field and return **`true`**. Otherwise, as we did in the definition of the `contains` method, we initialize an `Entry` object, `temp`, to `root` and compare `element` to `temp.element` in a loop. If `element` equals `temp.element`, we have an attempt to add a duplicate, so we return **`false`**. If `element` is less than `temp.element`, replace `temp` with `temp.left` unless `temp.left` is **`null`**, in which case we insert `element` in an `Entry` object whose parent is `temp`. The steps are similar when `element` is greater than `temp.element`.

For example, suppose we are trying to insert 45 into the binary search tree in Figure 10.5:

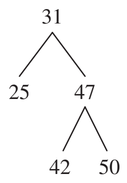


FIGURE 10.5 A binary search tree into which 45 will be inserted

The insertion is made in a loop that starts by comparing 45 to 31, the root element. Since $45 > 31$, we advance to 47, the right child of 31. See Figure 10.6.

Because $45 < 47$, we advance to 42, the left child of 47, as indicated in Figure 10.7.

At this point, $45 > 42$, so we would advance to the right child of 42 if 42 had a right child. It does not, so 45 is inserted as the right child of 42. See Figure 10.8.

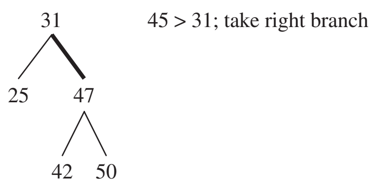


FIGURE 10.6 The effect of comparing 45 to 31 in the binary search tree of Figure 10.5

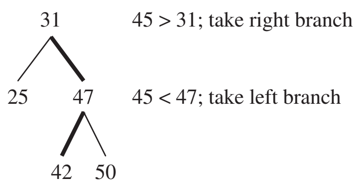


FIGURE 10.7 The effect of comparing 45 to 47 in the binary search tree of Figure 10.6

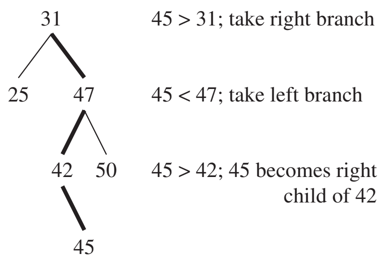

31
25
47
42
50
45
45 > 31; take right branch
45 < 47; take left branch
45 > 42; 45 becomes right child of 42

FIGURE 10.8 The effect of inserting 45 into the binary search tree in Figure 10.7

In general, the search fails if the element to be inserted belongs in an empty subtree of `temp`. Then the element is inserted as the only element in that subtree. That is, the inserted element *always becomes a leaf in the tree*. This has the advantage that the tree is not re-organized after an insertion.

Here is the complete definition (the loop continues indefinitely until, during some iteration, **`true`** or **`false`** is returned):

```
public boolean add (E element)
{
      if (root == null)
      {
              if (element == null)
                    throw new NullPointerException();
              root = new Entry<E> (element, null);
              size++;
              return true;
      } // empty tree
      else
      {
              Entry<E> temp = root;
```

```
                int comp;

                while (true)
                {
                       comp =  ((Comparable)element).compareTo (temp.element);
                       if (comp == 0)
                             return false;
                       if (comp < 0)
                             if (temp.left != null)
                                    temp = temp.left;
                             else
                             {
                                    temp.left = new Entry<E> (element, temp);
                                    size++;
                                    return true;
                             } // temp.left == null
                       else if (temp.right != null)
                             temp = temp.right;
                       else
                       {
                             temp.right = new Entry<E> (element, temp);
                             size++;
                             return true;
                       } // temp.right == null
                } // while
        } // root not null
   } // method add
```

The timing estimates for the `add` method are identical to those for the `contains` method, and depend on the height of the tree. To insert at the end of a binary search tree that forms a chain, the number of iterations is one more than the height of the tree. The height of a chain is linear in n, so worstTime(n) is linear in n. And, with the same argument we used for the `contains` method, we conclude that averageTime(n) is logarithmic in n.

In Programming Exercise 10.4, you get the opportunity to define the `add` method recursively.

10.1.2.5 The Definition of the `remove` Method

The only other `BinarySearchTree` method to be defined is

```
public boolean remove (Object obj)
```

The definition of the `remove` method in the `BinarySearchTree` class is more complicated than the definition of the `add` method from Section 10.1.2.4. The reason for the extra complexity is that the `remove` method requires a re-structuring of the tree—unless the element to be removed is a leaf. With the `add` method, the inserted element always becomes a leaf, and no re-structuring is needed.

The basic strategy is this: we first get (a reference to) the `Entry` object that holds the element to be removed, and then we delete that `Entry` object. Here is the definition:

```
public boolean remove (Object obj)
{
     Entry<E> e = getEntry (obj);
```

```
        if (e == null)
                return false;
        deleteEntry (e);
        return true;
    } // method remove
```

Of course, we need to postpone the analysis of the `remove` method until we have developed both the `getEntry` and `deleteEntry` methods. The **`protected`** method `getEntry` searches the tree—in the same way as the `contains` method defined in Section 10.1.2.3—for an `Entry` object whose element is `obj`. For example, Figure 10.9 shows what happens if the `getEntry` method is called to get a reference to the `Entry` whose element is 50.

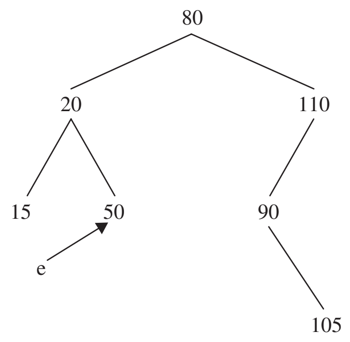


FIGURE 10.9 The effect of calling the `getEntry` method to get a reference to the `Entry` whose element is 50. A copy of the reference `e` is returned

Here is the definition of the `getEntry` method:

```
/**
 *  Finds the Entry object that houses a specified element, if there is such an Entry.
 *  The worstTime(n) is O(n), and averageTime(n) is O(log n).
 *
 *  @param obj - the element whose Entry is sought.
 *
 *  @return the Entry object that houses obj - if there is such an Entry;
 *          otherwise, return null.
 *
 *  @throws ClassCastException - if obj is not comparable to the elements
 *          already in this BinarySearchTree object.
 *  @throws NullPointerException - if obj is null.
 *
 */
protected Entry<E> getEntry (Object obj)
{
      int comp;

      if (obj == null)
            throw new NullPointerException();
      Entry<E> e = root;
      while (e != null)
      {
            comp = ((Comparable)obj).compareTo (e.element);
            if (comp == 0)
                  return e;
            else if (comp < 0)
                  e = e.left;
            else
                  e = e.right;
      } // while
      return null;
} // method getEntry
```

The analysis of the `getEntry` method is the same as for the `contains` method in Section 10.1.2.3: worstTime(n) is linear in n and averageTime(n) is logarithmic in n.

The structure of the **while** loop in the `getEntry` method is identical to that in the `contains` method. In fact, we can re-define the `contains` method to call `getEntry`. Here is the new definition, now a one-liner:

```
public boolean contains (Object obj)
{
     return (getEntry (obj) != null);
} // method contains
```

For the `deleteEntry` method, let's start with a few simple examples of how a binary search tree is affected by a deletion; then we'll get into the details of defining the `deleteEntry` method. As noted, removal of a leaf requires no re-structuring. For example, suppose we remove 50 from the binary search tree in Figure 10.10:

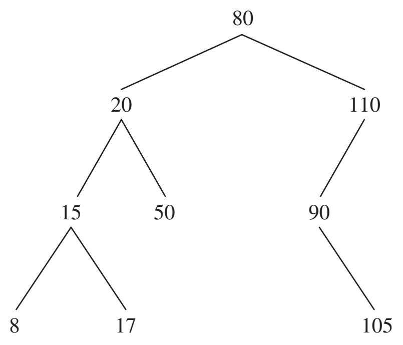


FIGURE 10.10 A binary search tree from which 50 is to be removed

To delete 50 from the tree in Figure 10.10, all we need to do is change to **`null`** the right field of 50's parent `Entry`—the `Entry` object whose element is 20. We end up with the binary search tree in Figure 10.11.

In general, if `p` is (a reference to) the `Entry` object that contains the leaf element to be deleted, we first decide what to do if `p` is the root. In that case, we set

```
root = null;
```

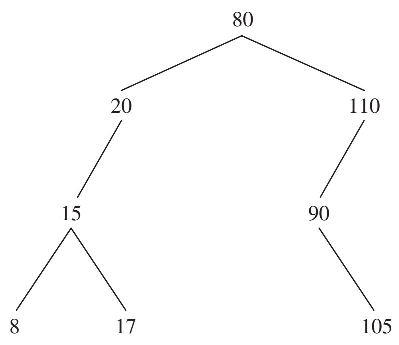


FIGURE 10.11 The binary search tree from Figure 10.10 after the removal of 50

Otherwise, the determination of which child of `p.parent` gets the value **`null`** depends on whether `p` is a left child or a right child:

```
if (p == p.parent.left)
         p.parent.left = null;
else
         p.parent.right = null;
```

Notice how we check to see if `p` is a (reference to) a left child: if `p` equals `p`'s parent's left child.

It is almost as easy to remove an element that has only one child. For example, suppose we want to remove 20 from the binary search tree in Figure 10.11. We cannot leave a hole in a binary search tree, so we must replace 20 with some element. Which one? The best choice is 15, the child of the element to be removed. So we need to link 15 to 20's parent. When we do, we get the binary search tree shown in Figure 10.12.

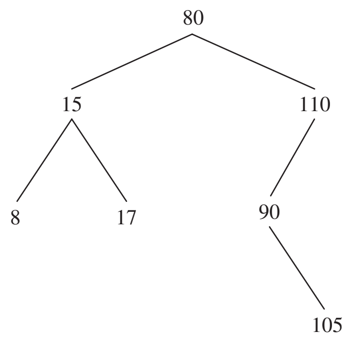


FIGURE 10.12 The binary search tree from Figure 10.11 after 20 was removed by replacing 20's entry with 15's entry

In general, let `replacement` be the `Entry` that replaces `p`, which has exactly one child. Then `replacement` should get the value of either `p.left` or `p.right`, whichever is not empty—they cannot both be empty because `p` has one child. We can combine this case, where `p` has one child, with the previous case, where `p` has no children:

```
Entry<E> replacement;

if (p.left != null)
        replacement = p.left;
else
        replacement = p.right;

// If p has at least one child, link replacement to p.parent.
if (replacement != null)
{
        replacement.parent = p.parent;
        if (p.parent == null)
                root = replacement;
        else if (p == p.parent.left)
                p.parent.left  = replacement;
        else
                p.parent.right = replacement;
} // p has at least one child
else if (p.parent == null)     // p is the root and has no children
        root = null;
else         // p has a parent and has no children
{
        if (p == p.parent.left)
                 p.parent.left = null;
        else
                 p.parent.right = null;
} // p has a parent but no children
```

Finally, we come to the interesting case: when the element to be removed has two children. For example, suppose we want to remove 80 from the binary search tree in Figure 10.12.

As in the previous case, 80 must be replaced with some other element in the tree. But which one? To preserve the ordering, a removed element should be replaced with either its immediate predecessor (in this case, 17) or its immediate successor (in this case, 90). We will, in fact, need a successor method for an inOrder iterator. So assume we already have a `successor` method that returns an `Entry` object's immediate successor. In general, the immediate successor, `s`, of a given `Entry` object `p` is the leftmost `Entry` object in the subtree `p.right`. Important: the left child of this leftmost `Entry` object will be **`null`**. (Why?)

The removal of 80 from the tree in Figure 10.12 is accomplished as follows: first, we copy the successor's element to `p.element`, as shown in Figure 10.13.

Next, we assign to `p` the value of (the reference) `s`. See Figure 10.14.

Then we delete `p`'s `Entry` object from the tree. As noted earlier, the left child of `p` must now be **`null`**, so the removal follows the replacement strategy of removing an element with one no children or one child. In this case, `p` has a right child (105), so 105 replaces `p`'s element, as shown in Figure 10.15.

Because the 2-children case reduces to the 0-or-1-child case developed earlier, the code for removal of any `Entry` object starts by handling the 2-children case, followed by the code for the 0-or-1-child case.

As we will see in Section 10.2.3, it is beneficial for subclasses of `BinarySearchTree` if the `deleteEntry` method returns the `Entry` object that is actually deleted. For example, if the `deleteEntry` method is called to delete an entry that has two children, the successor of that entry is actually removed from the tree and returned.

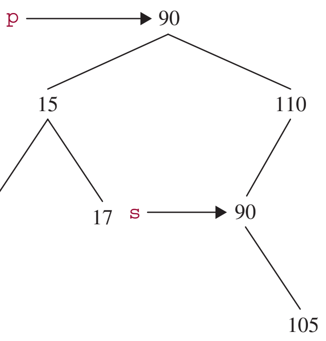


FIGURE 10.13 The first step in the removal of 80 from the binary search tree in Figure 10.12: 90, the immediate successor of 80, replaces 80

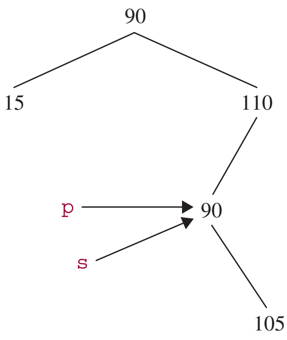


FIGURE 10.14 The second step in the removal of 80 in the binary search tree of Figure 10.12: p points to the successor entry

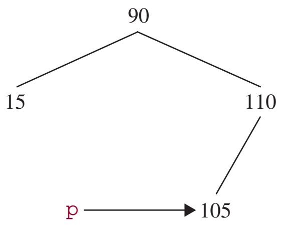


FIGURE 10.15 The final step in the removal of 80 from the binary search tree in Figure 10.12: `p`'s element (90) is replaced with that element's right child (105)

Here is the complete definition:

```
/**
 *  Deletes the element in a specified Entry object from this BinarySearchTree.
 *
 *  @param p – the Entry object whose element is to be deleted from this
 *          BinarySearchTree object.
 *
 *  @return the Entry object that was actually deleted from this BinarySearchTree
 *          object.
 *
*/
protected Entry<E> deleteEntry (Entry<E> p)
{
        size–;

        // If p has two children, replace p’s element with p’s successor’s
        // element, then make p reference that successor.
        if (p.left != null && p.right != null)
        {
                Entry<E> s = successor (p);
                p.element = s.element;
                p = s;
        } // p had two children


        // At this point, p has either no children or one child.

        Entry<E> replacement;

        if (p.left != null)
                replacement = p.left;
        else
                replacement = p.right;

        // If p has at least one child, link replacement to p.parent.
        if (replacement != null)
        {
                replacement.parent = p.parent;
                if (p.parent == null)
                       root = replacement;
                else if (p == p.parent.left)
                       p.parent.left  = replacement;
                else
                       p.parent.right = replacement;
        } // p has at least one child
        else if (p.parent == null)
                root = null;
        else
        {
                if (p == p.parent.left)
                       p.parent.left = null;
                else
                       p.parent.right = null;
        } // p has a parent but no children

        return p;
} // method deleteEntry
```

We still have the successor method to develop. Here is the method specification:

```
/**
 *  Finds the successor of a specified Entry object in this BinarySearchTree.
 *  The worstTime(n) is O(n) and averageTime(n) is constant.
 *
 *  @param e - the Entry object whose successor is to be found.
 *
 *  @return the successor of e, if e has a successor; otherwise, return null.
 *
 */
protected Entry<E> successor (Entry<E> e)
```

This method has **protected** visibility to reflect the fact that Entry—the return type and parameter type—has **protected** visibility.

How can we find the successor of an Entry object? For inspiration, look at the binary search tree in Figure 10.16.

In the tree in Figure 10.16, the successor of 50 is 55. To get to this successor from 50, we move right (to 75) and then move left as far as possible. Will this always work? Only for those entries that have a non-null right child. What if an Entry object—for example, the one whose element is 36—has a null right child? If the right child of an Entry object e is **null**, we get to e's successor by going back up the tree to the left as far as possible; the successor of e is the parent of that leftmost ancestor of e. For example, the successor of 36 is 37. Similarly, the successor of 68 is 75. Also, the successor of 28 is 30; since 28 is a left child, we go up the tree to the left zero times—remaining at 28—and then return that Entry object's parent, whose element is 30. Finally, the successor of 75 is **null** because its leftmost ancestor, 50,has no parent.

Here is the method definition:

```
protected Entry<E> successor (Entry<E> e)
{
     if (e == null)
               return null;
```

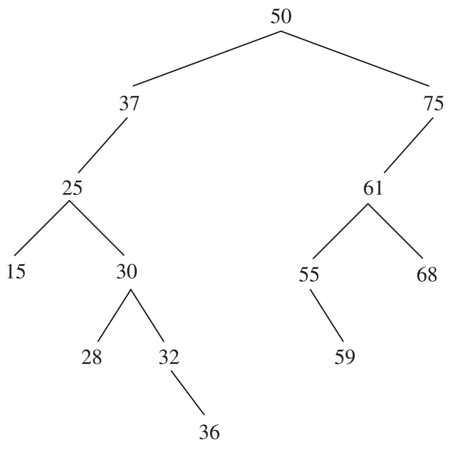


FIGURE 10.16 A binary search tree

```
        else if (e.right != null)
        {
                 // successor is leftmost Entry in right subtree of e
                 Entry<E> p = e.right;
                 while (p.left != null)
                        p = p.left;
                 return p;

        } // e has a right child
        else
        {
                 // go up the tree to the left as far as possible, then go up
                 // to the right.
                 Entry<E> p = e.parent;
                 Entry<E> ch = e;
                 while (p != null && ch == p.right)
                 {
                       ch = p;
                       p = p.parent;
                 } // while
                 return p;
        } // e has no right child
   } // method successor
```

To estimate worstTime(n) for the `successor` method, suppose the following elements are inserted into an initially empty binary search tree: $n, 1, 2, 3, \ldots, n-1$. The shape of the resulting tree is as shown in Figure 10.17.

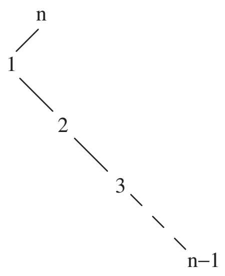


FIGURE 10.17 A binary search tree in which finding the successor of $n - 1$ requires $n - 3$ iterations

For the binary search tree in Figure 10.17, obtaining the successor of $n - 1$ requires $n - 3$ iterations, so worstTime(n) is linear in n. For averageTime(n), note that an element in the tree will be reached at most 3 times: once to get to its left child, once as the successor of that left child, and once in going back up the tree to get the successsor of its rightmost descendant. So the total number of loop iterations to access each element is at most $3n$, and the average number of loop iterations is $3n/n = 3$. That is, averageTime(n) is constant.

We can briefly summarize the steps needed to delete an entry:

a. If the entry has no children, simply set to **`null`** the corresponding subtree-link from the entry's parent (if the entry is the root, set the root to **`null`**).

b. If the entry has one child, replace the parent-entry link and the entry-child link with a parent-child link.

c. If the entry has two children, copy the element in the entry's immediate successor into the entry to be deleted, and then delete that immediate successor (by part a or part b).

Finally, we can estimate the time for the `remove` method. The `remove` method has no loops or recursive calls, so the time for that method is determined by the time for the `getEntry` and `deleteEntry` methods called by the `remove` method. As noted above, for the `getEntry` method, worstTime(n) is linear in n and averageTime(n) is logarithmic in n. The `deleteEntry` method has no loops or recursive calls, but calls the `successor` method, whose worstTime(n) is linear in n and whose averageTime(n) is constant. We conclude that for the `remove` method, worstTime(n) is linear in n and averageTime(n) is logarithmic in n.

To complete the development of the `BinarySearchTree` class, we develop the embedded `TreeIterator` class in Section 10.1.2.6.

10.1.2.6 The `TreeIterator` Class

All we have left to implement is the `TreeIterator` class, nested in the `BinarySearchTree` class. The method specifications for `hasNext()`, `next()`, and `remove()` were given in the `Iterator` interface back in Chapter 4. The only fields are a reference to the element returned by the most recent call to the `next()` method, and a reference to the element to be returned by the next call to the `next()` method. The declaration of the `TreeIterator` class starts out with

```
protected class TreeIterator implements Iterator<E>
{

        protected Entry<E> lastReturned = null,
                           next;
```

Before we get to defining the three methods mentioned above, we should define a default constructor. Where do we want to start? That depends on how we want to iterate. For a preOrder or breadthFirst iteration, we would start at the root `Entry` object. For an inOrder or postOrder iteration, we would start at the leftmost `Entry` object. We will want to iterate over the elements in a `BinarySearchTree` in ascending order, so we want to initialize the `next` field to the leftmost (that is, smallest) `Entry` object in the `BinarySearchTree`. To obtain that first `Entry` object, we start at the root and go left as far as possible:

```
/**
 *  Positions this TreeIterator to the smallest element, according to the Comparable
 *  interface, in the BinarySearchTree object.
 *  The worstTime(n) is O(n) and averageTime(n) is O(log n).
 *
 */
protected TreeIterator()
{
       next = root;
       if (next != null)
               while (next.left != null)
                      next = next.left;
} // default constructor
```

To estimate the time for this default constructor, the situation is the same as for the `contains` and `add` methods in the `BinarySearchTree` class: worstTime(n) is linear in n (when the tree consists of a chain of left children), and averageTime(n) is logarithmic in n.

The `hasNext()` method simply checks to see if the `next` field has the value **`null`**:

```
/**
 *  Determines if there are still some elements, in the BinarySearchTree object this
 *  TreeIterator object is iterating over, that have not been accessed by this
 *  TreeIterator object.
 *
 *  @return true - if there are still some elements that have not been accessed by
 *          this TreeIterator object; otherwise, return false.
 *
 */
public boolean hasNext()
{
       return next != null;
} // method hasNext
```

The definition of the `next()` method is quite simple because we have already defined the `successor` method:

```
/**
 *  Returns the element in the Entry this TreeIterator object was positioned at
 *  before this call, and advances this TreeIterator object.
 *  The worstTime(n) is O(n) and averageTime(n) is constant.
 *
 *  @return the element this TreeIterator object was positioned at before this call.
 *
 *  @throws NoSuchElementException - if this TreeIterator object was not
 *          positioned at an Entry before this call.
 *
 */
public E next()
{
       if (next == null)
               throw new NoSuchElementException();
       lastReturned = next;
       next = successor (next);
       return lastReturned.element;
} // method next
```

Finally, the `TreeIterator` class's `remove` method deletes the `Entry` that was last returned. Basically, we call `deleteEntry (lastReturned)`. A slight complication arises if `lastReturned` has two children. For example, suppose `lastReturned` references the `Entry` object whose element is (the `Integer` whose value is) 40 in the `BinarySearchTree` object of Figure 10.18.

For the `BinarySearchTree` object of Figure 10.18, if we simply call

```
deleteEntry (lastReturned);
```

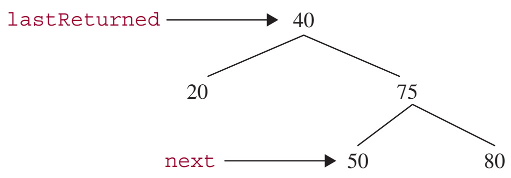


FIGURE 10.18 A binary search tree from which 40 is to be removed

then `next` will reference an `Entry` object that is no longer in the tree. To avoid this problem, we set

```
next = lastReturned;
```

before calling

```
deleteEntry (lastReturned);
```

Then the tree from Figure 10.18 would be changed to the tree in Figure 10.19.

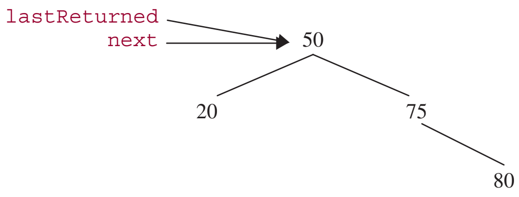


FIGURE 10.19 A binary search tree in which the element referenced by `lastReturned` is to be removed. Before `deleteEntry (lastReturned)` is called, `next` is assigned the value of `lastReturned`

After `deleteEntry (lastReturned)` is called for the tree in Figure 10.19, we get the tree in Figure 10.20.

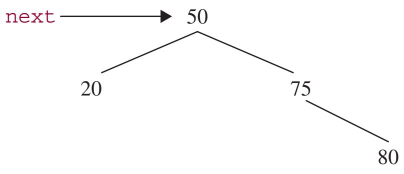


FIGURE 10.20 The tree from Figure 10.19 after `deleteEntry (lastReturned)` is called

For the tree in Figure 10.20, `next` is positioned where it should be positioned. We then set `lastReturned` to **`null`** to preclude a subsequent call to `remove()` before a call to `next()`.

Here is the method definition:

```
/**
 *  Removes the element returned by the most recent call to this TreeIterator
 *  object's next() method.
 *  The worstTime(n) is O(n) and averageTime(n) is constant.
```

```
     *
     *  @throws IllegalStateException – if this TreeIterator’s next() method was not
     *          called before this call, or if this TreeIterator’s remove() method was
     *          called between the call to the next() method and this call.
     *
     */
    public void remove()
    {
           if (lastReturned == null)
                  throw new IllegalStateException();
           if (lastReturned.left != null && lastReturned.right != null)
                  next = lastReturned;
           deleteEntry(lastReturned);
           lastReturned = null;
    } // method remove
```

Lab 17 provides run-time support for the claim made earlier that the average height of a BinarySearchTree is logarithmic in n.

You are now prepared to do Lab 17:
A Run-Time Estimate of the Average Height of a BinarySearchTree Object

10.2 Balanced Binary Search Trees

Keep in mind that the height of a BinarySearchTree is the determining factor in estimating the time to insert, remove or search. In the average case, the height of a BinarySearchTree object is logarithmic in n (the number of elements), so inserting, removing, and searching take only logarithmic-in-n time. This implies that BinarySearchTree objects represent an improvement, on average, over ArrayList objects and LinkedList obects, for which inserting, removing, or searching take linear time.[1] But in the worst case, a BinarySearchTree object’s height can be linear in n, which leads to linear-in-n worstTime(n) for inserting, removing, or searching.

We do not include any applications of the BinarySearchTree class because any application would be superseded by re-defining the tree instance from one of the classes in Chapter 12: TreeMap or TreeSet. For either of those classes, the height of the tree is *always* logarithmic in n, so insertions, removals, and searches take logarithmic time, even in the worst case. As we noted at the beginning of this chapter, the TreeMap and TreeSet classes are based on a somewhat complicated concept: the red-black tree. This section and the following two sections will help prepare you to understand red-black trees.

A binary search tree is ***balanced*** if its height is logarithmic in n, the number of elements in the tree. Three widely known data structures in this category of balanced binary search trees are AVL trees, red-black trees and splay trees. AVL trees are introduced in Section 10.3. Red-black trees are investigated in Chapter 12. For information on splay trees, the interested reader may consult Bailey [2003].

[1]The corresponding methods in the ArrayList and LinkedList classes are `add (int index, E element)`, `remove (Object obj)` and `contains (Object obj)`. Note that ArrayList objects and LinkedList objects are not necessarily in order.

For all of these balanced binary search trees, the basic mechanism that keeps a tree balanced is the rotation. A ***rotation*** is an adjustment to the tree around an element such that the adjustment maintains the required ordering of elements. The ultimate goal of rotating is to restore some balance property that has temporarily been violated due to an insertion or removal. For example, one such balance property is that the heights of the left and right subtrees of any element should differ by at most 1. Let's start with a simple classification of rotations: left and right.

In a ***left rotation***, some adjustments are made to the element's parent, left subtree and right subtree. The main effect of these adjustments is that the element becomes the left child of what had been the element's right child. For a simple example, Figure 10.21 shows a left rotation around the element 50. Note that before and after the rotation, the tree is a binary search tree.

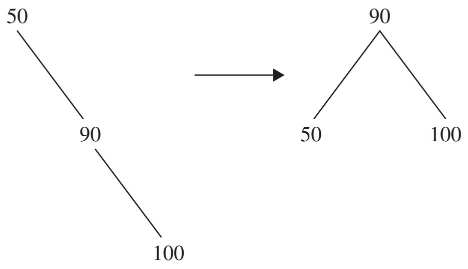


FIGURE 10.21 A left rotation around 50

Figure 10.22 has another example of a left rotation, around the element 80, that reduces the height of the tree from 3 to 2.

The noteworthy feature of Figure 10.22 is that 85, which was in the right subtree of the before-rotation tree, ends up in the left subtree of the after-rotation tree. This phenomenon is common to all left rotations around an element x whose right child is y. The left subtree of y becomes the right subtree of x. This adjustment is necessary to preserve the ordering of the binary search tree: Any element that was in y's left subtree is greater than x and less than y. So any such element should be in the right subtree of x (and in the left subtree of y). Technically, the same phenomenon also occurred in Figure 10.21, but the left subtree of 50's right child was empty.

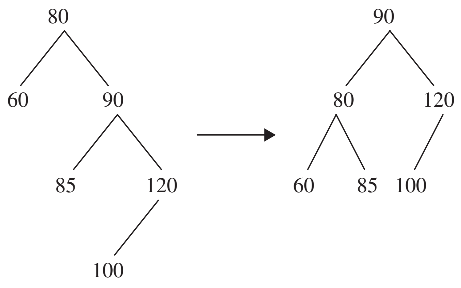


FIGURE 10.22 A left rotation around 80

Figure 10.23 shows the rotation of Figure 10.22 in a broader context: the element rotated around is not the root of the tree. Before and after the rotation, the tree is a binary search tree.

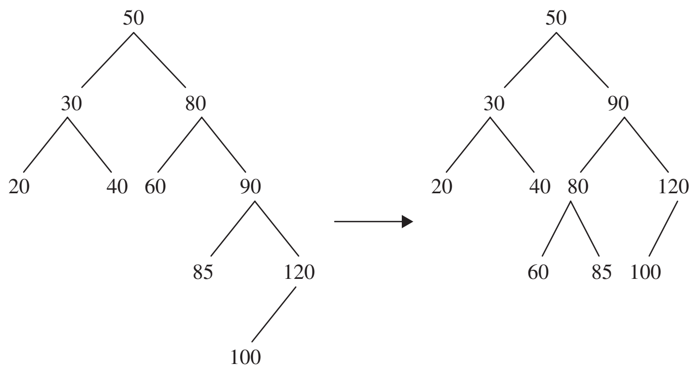
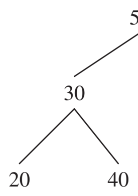


FIGURE 10.23 The left rotation around 80 from Figure 10.22, but here 80 is not the root element

Figure 10.23 illustrates another aspect of all rotations: all the elements that are not in the rotated element's subtree are unaffected by the rotation. That is, in both trees, we still have:



If we implement a rotation in the `BinarySearchTree` class, no elements are actually moved; only the references are manipulated. Suppose that `p` (for "parent") is a reference to an `Entry` object and `r` (for "right child") is a reference to `p`'s right child. Basically, a left rotation around `p` can be accomplished in just two steps:

```
p.right = r.left;  // for example, look at 85 in Figure 10.22
r.left = p;
```

Unfortunately, we also have to adjust the parent fields, and that adds quite a bit of code. Here is the complete definition of a `leftRotate` method in the `BinarySearchTree` class (a similar definition appears in the `TreeMap` class in Chapter 12):

```
/**
 *  Performs a left rotation in this BinarySearchTre object around a specified
 *  Entry object.
 *
 *  @param p – the Entry object around which the left rotation is performed
 *
 *  @throws NullPointerException – if p is null or p.right is null.
 *
 *  @see Cormen, 2002.
protected void rotateLeft (Entry<E> p)
{
       Entry<E> r = p.right;
       p.right = r.left;
       if (r.left != null)
           r.left.parent = p;
       r.parent = p.parent;
       if (p.parent == null)
           root = r;
       else if (p.parent.left == p)
           p.parent.left = r;
       else
           p.parent.right = r;
       r.left = p;
       p.parent = r;
} // method rotateLeft
```

This indicates how much of a bother parents can be! But on the bright side, no elements get moved, and the time is constant.

What about a right rotation? Figure 10.24 shows a simple example: a right rotation around 50.

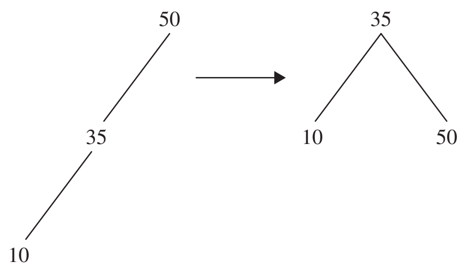


FIGURE 10.24 A right rotation around 50

Does this look familiar? Figure 10.24 is just Figure 10.21 with the direction of the arrow reversed. In general, if you perform a left rotation around an element and then perform a right rotation around the new parent of that element, you will end up with the tree you started with.

Figure 10.25 shows a right rotation around 80, in which the right child of 80's left child becomes the left child of 80. This is analogous to the left rotation in Figure 10.22.

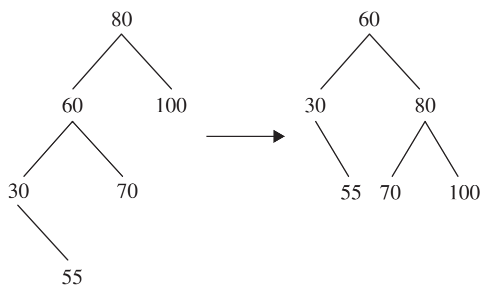


FIGURE 10.25 A right rotation around 80

Here are details on implementing right rotations in the `BinarySearchTree` class. Let `p` be a reference to an `Entry` object and let l (for "left child") be a reference to the left child of `p`. Basically, a right rotation around `p` can be accomplished in just two steps:

```
p.left = l.right;  // for example, look at 70 in the rotation of Figure 10.8
l.right = p;
```

Of course, once we include the parent adjustments, we get a considerably longer—but still constant time—method. In fact, if you interchange "left" with "right" in the definition of the `leftRotate` method, you get the definition of `rightRotate`.

In all of the rotations shown so far, the height of the tree was reduced by 1. That is not surprising; in fact, reducing height is the motivation for rotating. But it is not necessary that every rotation reduce the height of the tree. For example, Figure 10.26 shows a left rotation—around 50—that does not affect the height of the tree.

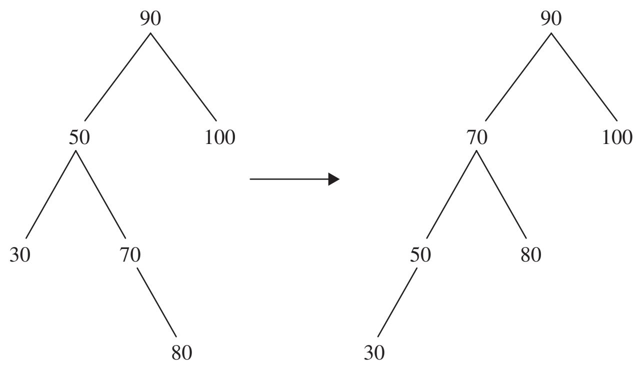


FIGURE 10.26 A left rotation around 50. The height of the tree is still 3 after the rotation

It is true that the left rotation in Figure 10.26 did not reduce the height of the tree. But a few minutes of checking should convince you that no single rotation can reduce the height of the tree on the left side of Figure 10.26. Now look at the tree on the right side of Figure 10.26. Can you figure out a rotation that will reduce the height of *that* tree? Not a right rotation around 70; that would just get us back where we started. How about a right rotation around 90? Bingo! Figure 10.27 shows the effect.

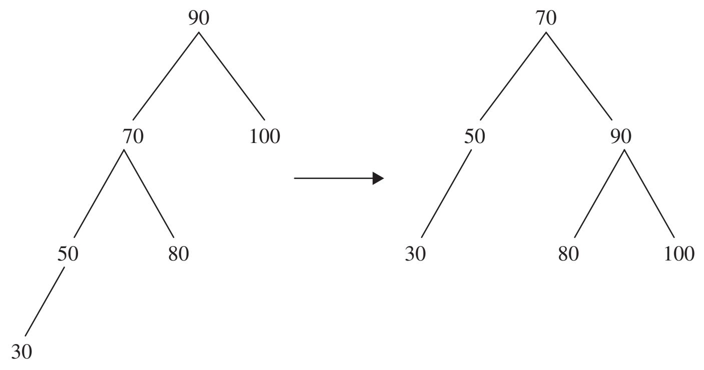


FIGURE 10.27 A right rotation around 90. The height of the tree has been reduced from 3 to 2

The rotations in Figures 10.26 and 10.27 should be viewed as a package: a left rotation around 90's left child, followed by a right rotation around 90. This is referred to as a ***double rotation***. In general, if `p` is a reference to an `Entry` object, then a double rotation around `p` can be accomplished as follows:

```
leftRotate (p.left);
rightRotate (p);
```

Figure 10.28 shows another kind of double rotation: a right rotation around the right child of 50, followed by a left rotation around 50.

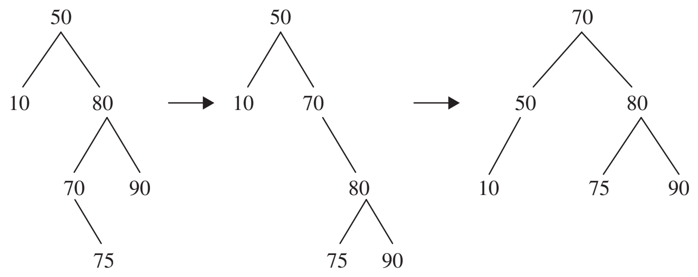


FIGURE 10.28 Another kind of double rotation: a right rotation around 50's right child, followed by a left rotation around 50

Before we move on to Section 10.2.1 with a specific kind of balanced binary search tree, let's list the major features of rotations:

1. There are four kinds of rotation:
   - **a.** Left rotation;
   - **b.** Right rotation;
   - **c.** A left rotation around the left child of an element, followed by a right rotation around the element itself;
   - **d.** A right rotation around the right child of an element, followed by a left rotation around the element itself.
2. Elements not in the subtree of the element rotated about are unaffected by the rotation.
3. A rotation takes constant time.
4. Before and after a rotation, the tree is still a binary search tree.
5. The code for a left rotation is symmetric to the code for a right rotation: simply swap the words "left" and "right".

Section 10.2.1 introduces the AVL tree, a kind of binary search tree that employs rotations to maintain balance.

10.2.1 AVL Trees

An ***AVL tree*** is a binary search tree that either is empty or in which:

1. the heights of the root's left and right subtrees differ by at most 1, and
2. the root's left and right subtrees are AVL trees.

AVL trees are named after the two Russian mathematicians, Adelson-Velski and Landis, who invented them in 1962. Figure 10.29 shows three AVL trees, and Figure 10.30 shows three binary search trees that are not AVL trees.

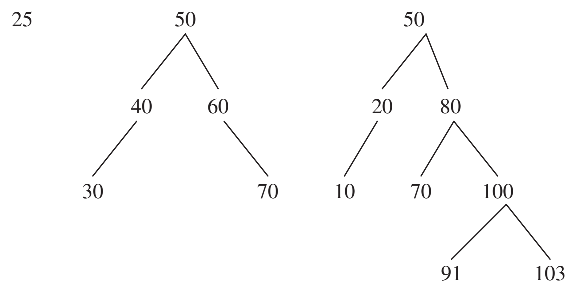


FIGURE 10.29 Three AVL trees

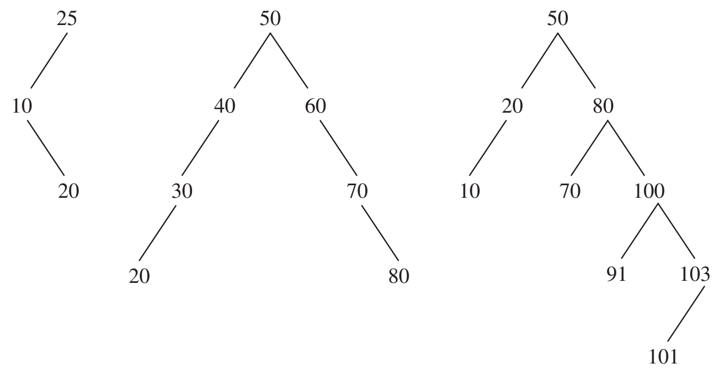


FIGURE 10.30 Three binary search trees that are not AVL trees

The first tree in Figure 10.30 is not an AVL tree because its left subtree has height 1 and its right subtree has height −1. The second tree is not an AVL tree because its left subtree is not an AVL tree; neither is its right subtree. The third tree is not an AVL tree because its left subtree has height 1 and its right subtree has height 3.

In Section 10.2.2, we show that an AVL tree is a balanced binary search tree, that is, that the height of an AVL tree is always logarithmic in n. This compares favorably to a binary search tree, whose height is linear in n in the worst case (namely, a chain). The difference between linear and logarithmic can be huge. For example, suppose $n = 1,000,000,000,000$. Then $\log_2 n$ is less than 40. The practical import of this difference is that insertions, removals and searches for the `AVLTree` class take far less time, in the worst case, than for the `BinarySearchTree` class.

10.2.2 The Height of an AVL Tree

We can prove that an AVL tree's height is logarithmic in n, and the proof relates AVL trees back to, of all things, Fibonacci numbers.

Claim If t is a non-empty AVL tree, height(t) is logarithmic in n, where n is the number of elements in t.

Proof We will show that, even if an AVL tree t has the maximum height possible for its n elements, its height will still be logarithmic in n. How can we determine the maximum height possible for an AVL tree with n elements? As Kruse (see Kruse [1987]) suggests, rephrasing the question helps us get the answer. Given a height h, what is the minimum number of elements in any AVL tree of that height?

For $h = 0, 1, 2, \ldots$, let min_h be the minimum number of elements in an AVL tree of height h. Clearly, $\text{min}_0 = 1$ and $\text{min}_1 = 2$. The values of min_2 and min_3 can be seen from the AVL trees in Figure 10.31.

In general, if $h1 > h2$, then min_{h1} is greater than the number of elements needed to construct an AVL tree of height $h2$. That is, if $h1 > h2$, then $\text{min}_{h1} > \text{min}_{h2}$. In other words, min_h is an increasing function of h.

Suppose that t is an AVL tree with h height and min_h elements, for some value of $h > 1$. What can we say about the heights of the left and right subtrees of t? By the definition of height, one of those subtrees must have height $h - 1$. And by the definition of an AVL tree, the other subtree must have height of $h - 1$ or $h - 2$. In fact, because t has the minimum number of elements for its height, one of its subtrees must have height $h - 1$ and min_{h-1} elements, and the other subtree must have height $h - 2$ and min_{h-2} elements.

A tree always has one more element than the number of elements in its left and right subtrees. So we have the following equation, called a ***recurrence relation***:

$$\text{min}_h = \text{min}_{h-1} + \text{min}_{h-2} + 1, \text{ for any integer } h > 1$$

Now that we can calculate min_h for any positive integer h, we can see how the function min_h is related to the maximum height of an AVL tree. For example, because $\text{min}_6 = 33$ and $\text{min}_7 = 54$, the maximum height of an AVL tree with 50 elements is six.

The above recurrence relation looks a lot like the formula for generating Fibonacci numbers (see Lab 7). The term ***Fibonacci tree*** refers to an AVL tree with the minimum number of elements for its height. From the above recurrence relation and the values of min_0 and min_1, we can show, by induction on h, that

$$\text{min}_h = \text{fib}(h + 3) - 1, \text{ for any nonnegative integer } h.$$

We can further show, by induction on h (see Concept Exercise 10.8),

$$\text{fib}(h + 3) - 1 \geq (3/2)^h, \text{ for any nonnegative integer } h.$$

Combining these results,

$$\text{min}_h \geq (3/2)^h, \text{ for any nonnegative integer } h.$$

Taking logs in base 2 (but any base will do), we get

$$\log_2(\text{min}_h) \geq h * \log_2(3/2), \text{ for any nonnegative integer } h.$$

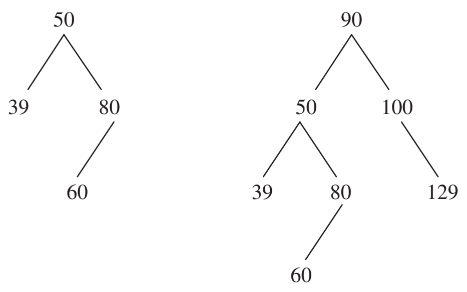


FIGURE 10.31 AVL trees of heights 2 and 3 in which the number of elements is minimal

Rewriting this in a form suitable for a Big-O claim, with $1/\log_2(3/2) < 1.75$:

$$h \le 1.75 * \log_2(\min_h), \text{ for any nonnegative integer } h.$$

If t is an AVL tree with h height and n elements, we must have $\min_h \le n$, so for any such AVL tree,

$$h \le 1.75 * \log_2(n).$$

This implies that the height of any AVL tree is $O(\log n)$. Is $O(\log n)$ a tight upper bound; that is, is the height of any AVL tree logarithmic in n? Yes, and here's why. For any binary tree of height h with n elements,

$$h \ge \log_2(n+1) - 1$$

by part 2 of the Binary Tree Theorem. We conclude that any AVL tree with n elements has height that is logarithmic in n, even in the worst case.

To give you a better idea of how AVL trees relate to binary search trees, we sketch the design and implementation of the `AVLTree` class in Section 10.2.3. To complete the implementation, you will need to tackle Programming Projects 10.3 and 10.4. Those projects deal with some of the details of the `add` and `remove` methods, respectively.

10.2.3 The `AVLTree` Class

The `AVLTree` class will be developed as a subclass of the `BinarySearchTree` class. There will not be any additional fields in the `AVLTree` class, but each entry object has an additional field:

```
char balanceFactor = '=';
```

The purpose of this additional field in the `Entry` class is to make it easier to maintain the balance of an `AVLTree` object. If an `Entry` object has a `balanceFactor` value of '=', the `Entry` object's left subtree has the same height as the `Entry` object's right subtree. If the `balanceFactor` value is 'L', the left subtree's height is one greater than the right subtree's height. And a `balanceFactor` value of 'R' means that the right subtree's height is one greater than the left subtree's height. Figure 10.32 shows an AVL tree with each element's balance factor shown below the element.

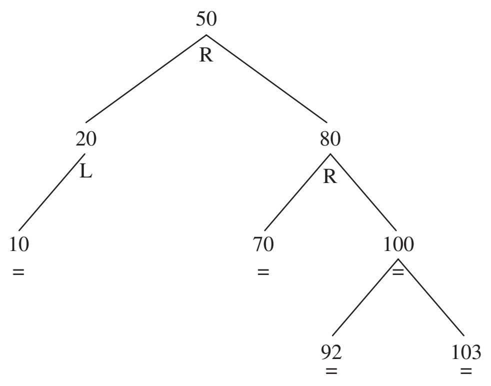


FIGURE 10.32 An AVL tree with the balance factor under each element

Here is the new entry class, nested in the AVLTree class:

```
protected static class AVLEntry<E> extends BinarySearchTree.Entry<E>
{
      protected char balanceFactor = '=';


      /**
        *  Initializes this AVLEntry object from a specified element and a
        *  specified parent AVLEntry.
        *
        *  @param element - the specified element to be housed in this
        *         AVLEntry object.
        *  @param parent - the specified parent of this AVLEntry object.
        *
        */
        protected AVLEntry (E element, AVLEntry<E> parent)
        {
                 this.element = element;
                 this.parent = parent;
        } // constructor
      } // class AVLEntry
```

The only methods that the AVLTree class overrides from the BinarySearchTree class are those that involve the AVLEntry class's balanceFactor field. Specifically, the AVLTree class will override the add and deleteEntry methods from the BinarySearchTree class. The re-balancing strategy is from Sahni (see Sahni [2000]).

One intriguing feature of the AVLTree class is that the contains method is not overridden from the BinarySearchTree class, but worstTime(n) is different: logarithmic in n, versus linear in n for the BinarySearchTree class. This speed reflects the fact that the height of an AVL tree is always logarithmic in n.

The definition of the add method in the AVLTree class resembles the definition of the add method in the BinarySearchTree class. But as we work our way down the tree from the root to the insertion point, we keep track of the inserted AVLEntry object's closest ancestor whose balanceFactor is 'L' or 'R'. We refer to this Entry object as imbalanceAncestor. For example, if we insert 60 into the AVLTree object in Figure 10.33, imbalanceAncestor is the Entry object whose element is 80:

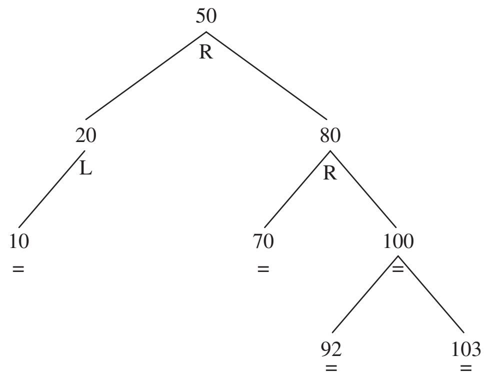


FIGURE 10.33 An AVLTree object

After the element has been inserted, BinarySearchTree-style, into the AVLTree object, we call a fix-up method to handle rotations and balanceFactor adjustments. Here is the definition of the add method:

```
public boolean add (E element)
{
        if (root == null)
        {
              if (element == null)
                    throw new NullPointerException();
              root = new AVLEntry<E> (element, null);
              size++;
              return true;
        } // empty tree
        else
        {
              AVLEntry<E>  temp = (AVLEntry<E>)root,
                           imbalanceAncestor = null;  // nearest ancestor of
                                                       // element with
                                                       // balanceFactor not '='
              int comp;

              while (true)
              {
                    comp = ((Comparable)element).compareTo (temp.element);
                    if (comp == 0)
                           return false;
                    if (comp < 0)
                    {
                           if (temp.balanceFactor != '=')
                                 imbalanceAncestor = temp;
                           if (temp.left != null)
                                 temp = (AVLEntry<E>)temp.left;
                           else
                           {
                                 temp.left = new AVLEntry<E>  (element, temp);
                                 fixAfterInsertion ((AVLEntry<E>)temp.left,
                                                   imbalanceAncestor);

                                 size++;
                                 return true;
                           } // temp.left == null
                    } // comp < 0
                    else
                    {
                           if (temp.balanceFactor != '=')
                                 imbalanceAncestor = temp;
                           if (temp.right != null)
                                 temp = (AVLEntry<E>)temp.right;
                           else
```

```
                            {
                                    temp.right = new AVLEntry<E>(element, temp);
                                    fixAfterInsertion ((AVLEntry<E>)temp.right,
                                                            imbalanceAncestor);

                                    size++;
                                    return true;
                            } // temp.right == null
                    } // comp > 0
            } // while
        } // root not null
} // method add
```

This code differs from that of the `BinarySearchTree` class's `add` method in three respects:

1. The new entry is an instance of `AVLEntry<E>`.
2. The `imbalanceAncestor` variable is maintained.
3. The `fixAfterInsertion` method is called to re-balance the tree, if necessary.

The definition of the `fixAfterInsertion` method is left as Programming Project 10.3. The bottom line is that, for the `add` method, worstTime(n) is O($\log n$). In fact, because the **`while`** loop in the `add` method can require as many iterations as the height of the AVL tree, worstTime(n) is logarithmic in n.

The definition of the `deleteEntry` method (called by the inherited `remove` method) starts by performing a `BinarySearchTree` -style deletion, and then invokes a `fixAfterDeletion` method. Fortunately, for the sake of code re-use, we can explicitly call the `BinarySearchTree` class's `deleteEntry` method, so the complete definition of the `AVLTree` class's `deleteEntry` method is simply:

```
protected Entry<E>  deleteEntry (Entry<E>  p)
{
      AVLEntry<E>  deleted = (AVLEntry<E>)super.deleteEntry (p);
      fixAfterDeletion (deleted.element, (AVLEntry<E>)deleted.parent);
      return deleted;
} // method deleteEntry
```

Of course, we are not done yet; we are not even close: The definition of the `fixAfterDeletion` method is left as Programming Project 10.4. For the `remove` method, worstTime(n) is O($\log n$). In fact, because we start with a `BinarySearchTree` -style deletion, worstTime(n) is logarithmic in n.

The book's website includes an applet that will help you to visualize insertions in and removals from an `AVLTree` object.

10.2.4 Runtime Estimates

We close out this chapter with a brief look at some run-time issues. For the `AVLTree` class's `add` method, worstTime(n) is logarithmic in n, whereas for the `BinarySearchTree` class's `add` method, worstTime(n) is linear in n. And so, as you would expect, the worst case run-time behavior of the `AVLTree` class's `add` method is much faster than that of the `BinarySearchTree` class's `add` method. Programming Exercise 10.5 confirms this expectation.

What about averageTime(n)? The averageTime(n) is logarithmic in n for the `add` method in those two classes. Which do you think will be faster in run-time experiments? Because quite a bit of effort goes into maintaining the balance of an `AVLTree` object, the average height of a `BinarySearchTree` object is about 50% larger than the average height of an `AVLTree` object: 2.1 $\log_2 n$ versus 1.44 $\log_2 n$. But the extra maintenance makes `AVLTree` insertions slightly slower, in spite of the height advantage, than `BinarySearchTree` insertions.

In Chapter 12, we present another kind of balanced binary search tree: the red-black tree. Insertions in red-black trees are slightly faster (but less intuitive), on average, than for AVL trees, and that is why the red-black tree was selected as the underlying structure for the Java Collection Framework's `TreeMap` and `TreeSet` classes. Both of these classes are extremely useful; you will get some idea of this from the applications and Programming Projects in Chapter 12.

SUMMARY

A ***binary search tree*** t is a binary tree such that either t is empty or

1. each element in leftTree(t) is less than the root element of t;
2. each element in rightTree(t) is greater than the root element of t;
3. both leftTree(t) and rightTree(t) are binary search trees.

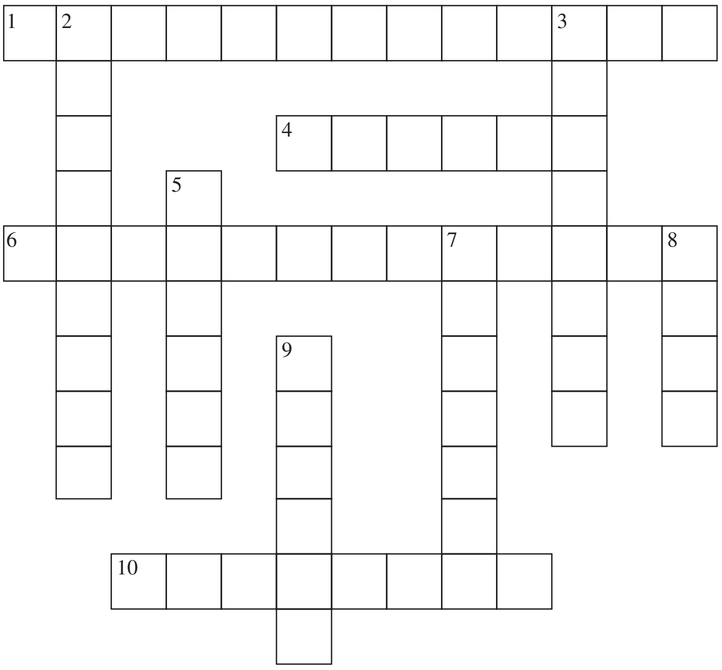
The `BinarySearchTree` class maintains a sorted collection of `Comparable` elements. The time estimates for searching, inserting, and deleting depend on the height of the tree. In the worst case—if the tree is a chain—the height is linear in n, the number of elements in the tree. The average height of a binary search tree is logarithmic in n. So for the `contains`, `add`, and `remove` methods, worstTime(n) is linear in n and averageTime(n) is logarithmic in n.

A binary search tree is ***balanced*** if its height is logarithmic in n, the number of elements in the tree. The balancing is maintained with rotations. A ***rotation*** is an adjustment to the tree around an element such that the adjustment maintains the required ordering of elements. This chapter introduced one kind of balanced binary search tree: the AVL tree. An ***AVL tree*** is a binary search tree that either is empty or in which:

1. the heights of the root's left and right subtrees differ by at most 1, and
2. the root's left and right subtrees are AVL trees.

The `AVLTree` class is a subclass of the `BinarySearchTree` class. The only overridden methods are those related to maintaining balance: `add` and `deleteEntry`.

CROSSWORD PUZZLE



www.CrosswordWeaver.com

ACROSS

1. An AVL tree with the minimum number of elements for its height

4. The feature of a `BinarySearchTree` object most important in estimating worstTime(*n*) and averageTime(*n*) for the `contains`, `add` and `remove` methods

6. The only field in the `AVLEntry` class that is not inherited from the nested `Entry` class of the `BinarySearchTree` class

10. A binary search tree is ________ if its height is logarithmic in *n*, the number of elements in the tree.

DOWN

2. The given definitions of the `contains`, `add` and `remove` methods in the `BinarySearchTree` class are _____.

3. An adjustment to a binary search tree around an element that maintains the required ordering of elements.

5. The fourth field in the nested `Entry` class of the `BinarySearchTree` class. The other three fields are `element`, `left` and `right`.

7. A balanced binary search tree in which, if the tree is not empty, the heights of the left and right subtrees differ by at most one

8. The only field, other than `size`, in the `BinarySearchTree` class

9. The worstTime(*n*) for the `copy` method in the `BinarySearchTree` class is __________ in *n*.

CONCEPT EXERCISES

10.1 **a.** Show the effect of making the following insertions into an initially empty binary search tree:

30, 40, 20, 90, 10, 50, 70, 60, 80

b. Find a different ordering of the above elements whose insertions would generate the same binary search tree as in part a.

10.2 Describe in English how to remove each of the following from a binary search tree:

a. an element with no children

b. an element with one child

c. an element with two children

10.3 **a.** For any positive integer n, describe how to arrange the integers $1, 2, \ldots, n$ so that when they are inserted into a `BinarySearchTree` object, the height of the tree will be linear in n.

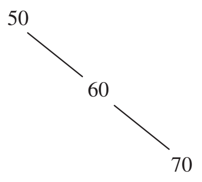
b. For any positive integer n, describe how to arrange the integers $1, 2, \ldots, n$ so that when they are inserted into a `BinarySearchTree` object, the height of the tree will be logarithmic in n.

c. For any positive integer n, describe how to arrange the integers $1, 2, \ldots, n$ so that when they are inserted into an `AVLTree` object, the height of the tree will be logarithmic in n.

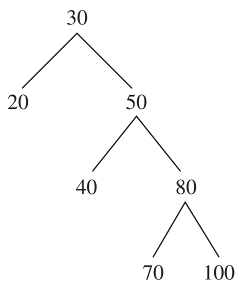
d. For any positive integer n, is it possible to arrange the integers $1, 2, \ldots, n$ so that when they are inserted into an `AVLTree` object, the height of the tree will be linear in n? Explain.

10.4 In each of the following binary search trees, perform a left rotation around 50.

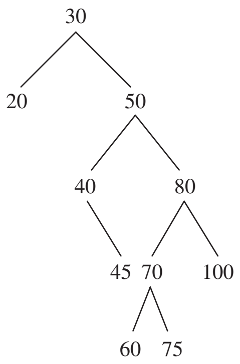
a.



b.

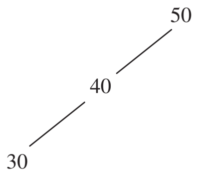


c.

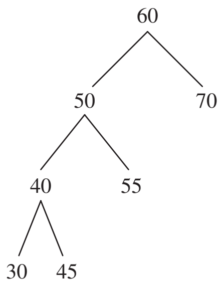


10.5 In each of the following binary search trees, perform a right rotation around 50.

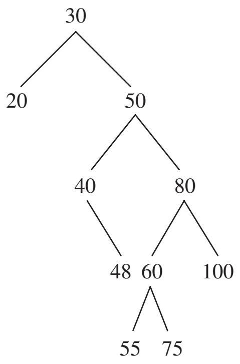
a.



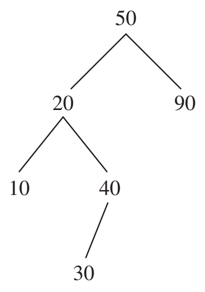
b.



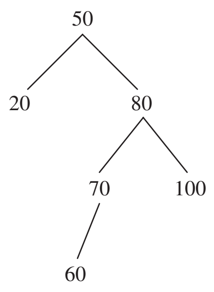
c.



10.6 In the following binary search tree, perform a double rotation (a left rotation around 20 and then a right rotation around 50) to reduce the height to 2.



10.7 In the following binary search tree, perform a "double rotation" to reduce the height to 2:



10.8 Show that for any nonnegative integer h,

$$\text{fib}(h+3) - 1 \geq (3/2)^h$$

Hint: Use the Strong Form of the Principle of Mathematical Induction and note that, for $h > 1$,

$$(3/2)^{h-1} + (3/2)^{h-2} = (3/2)^{h-2} * (3/2+1) > (3/2)^{h-2} * 9/4.$$

10.9 Suppose we define max_h to be the maximum number of elements in an AVL tree of height h.

a. Calculate max_3.

b. Determine the formula for max_h for any $h \geq 0$.
Hint: Use the Binary Tree Theorem from Chapter 9.

c. What is the maximum height of an AVL tree with 100 elements?

10.10 Show that the height of an AVL tree with 32 elements must be exactly 5.
Hint: calculate max_4 (see Concept Exercise 10.9) and min_6.

10.11 For the `contains` method in the `BinarySearchTree` class, worstTime(n) is linear in n. The `AVLTree` class does not override that method, but for the `contains` method in the `AVLTree` class, worstTime(n) is logarithmic in n. Explain.

10.12 The following program generates a BinarySearchTree object of n elements. Draw the tree when $n = 13$. For any $n \geq 0$, provide a Θ (that is, Big Theta) estimate of height(n), that is, the height of the BinarySearchTree object as a function of n.

```
import java.util.*;

public class weirdBST
{
     public static void main (String[] args)
     {
         new weirdBST().run();
     } // method main

     public void run()
     {
        BinarySearchTree<Double> tree = new BinarySearchTree<Double>();

        LinkedList<Double> list =new LinkedList<Double>();

        System.out.println ("Enter n > 0");

        int n = new Scanner (System.in).nextInt();

        tree.add (1.0);
        list.add (1.0);
        int k = 2;
        while (tree.size() < n)
             addLevel (tree, n, k++, list);
        System.out.println (tree.height());
     } // method run

     public void addLevel (BinarySearchTree<Double> tree, int n, int k,
                           LinkedList<Double> list)
     {
        final double SMALL = 0.00000000001;

        LinkedList<Double> newList = new LinkedList<Double>();

        Iterator<Double> itr = list.iterator();

        double d = itr.next();
        tree.add (d - 1.0);
        newList.add (d - 1.0);
        for (int i = 0; i < k && tree.size() < n; i++)
        {
           tree.add (d + SMALL );
           newList.add (d + SMALL);
           if (itr.hasNext())
            d = itr.next();
        } // for
```

```
            list.clear();
            list.addAll (newList);
         } // method addLevel

    } // class weirdBST
```

PROGRAMMING EXERCISES

10.1 In the BinarySearchTree class, test and develop a leaves method. Here is the method specification:

```
/**
 *  Returns the number of leaves in this BinarySearchTree object.
 *  The worstTime(n) is O(n).
 *
 *  @return - the number of leaves in this BinarySearchTree object.
 *
 */
public int leaves()
```

Test your method by adding tests to the BinarySearchTreeTest class available from the book's website. **Hint:** A recursive version, invoked by a wrapper method, can mimic the definition of *leaves(t)* from Section 9.1. Or, you can also develop an iterative version by creating a new iterator class in which the next method increments a count for each Entry object whose left and right fields are **null**.

10.2 Modify the BinarySearchTree class so that the iterators are fail-fast (see Appendix 1 for details on fail-fast iterators). Test your class by adding tests to the BinarySearchTreeTest class available from the book's website.

10.3 Modify the BinarySearchTree class so that BinarySearchTree objects are serializable (see Appendix 1 for details on serializability). Test your class by adding tests to the BinarySearchTreeTest class available from the book's website.

10.4 Create a recursive version of the add method.
Hint: Make the add method a wrapper for a recursive method. Test your version with the relevant tests in the BinarySearchTreeTest class available from the book's website.

10.5 In the BinarySearchTree class, modify the getEntry method so that it is a wrapper for a recursive method. Test your version with the relevant tests in the BinarySearchTreeTest class available from the book's website.

10.6 (This exercise assumes you have completed Programming Projects 10.3 and 10.4.) Create a test suite for the AVLTree class.
Hint: Make very minor modifications to the BinarySearchTreeTest class available from the book's website. Use your test suite to increase your confidence in the correctness of the methods you defined in Programming Projects 10.3 and 10.4.

10.7 (This exercise assumes you have completed Programming Projects 10.3 and 10.4.) In the AVLTree class, test and define the following method:

```
/**
 *  The height of this AVLTree object has been returned.
 *  The worstTime(n) is O(log n).
```

```
     *
     *  @return the height of this AVLTree object.
     *
     */
   public int height()
```

Hint: Use the `balanceFactor` field in the `AVLEntry` class to guide you down the tree. Test your method by adding tests to the `BinarySearchTreeTest` class available from the book's website.

Programming Project 10.1

An Alternate Implementation of the Binary-Search-Tree Data Type

This project illustrates that the binary-search-tree data type has more than one implementation. You can also use the technique described below to save a binary search tree (in fact, any binary tree) to disk so that it can be subsequently retrieved with its original structure.

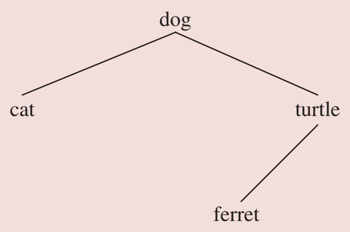
Develop an array-based implementation of the binary-search-tree data type. Your class, `BinarySearchTreeArray<E>`, will have the same method specifications as the `BinarySearchTree<E>` class but will use indexes to simulate the parent, left, and right links. For example, the fields in your embedded `Entry<E>` class might have:

```
E element;
int parent,
     left,
     right;
```

Similarly, the `BinarySearchTreeArray<E>` class might have the following three fields:

```
Entry<E> [ ] tree;

int root,
     size;
```

The root `Entry` object is stored in `tree [0]`, and a **`null`** reference is indicated by the index −1. For example, suppose we create a binary search tree by entering the `String` objects "dog", "turtle", "cat", "ferret". The tree would be as follows:



(continued on next page)

(continued from previous page)

The array representation, with the elements stored in the order in which they are entered, is

	Element	parent	left	right
0	dog	–1	2	1
1	turtle	0	3	–1
2	cat	0	–1	–1
3	ferret	1	–1	–1
. . .				

The method definitions are very similar to those in the `BinarySearchTree` class, except that an expression such as

```
root.left
```

is replaced with

```
tree [root].left
```

For example, here is a possible definition of the `getEntry` method:

```
protected Entry<E> getEntry (Object obj)
{
       int temp = root,
           comp;

       while (temp != -1)
       {
             comp = ((Comparable)obj).compareTo (tree [temp].element);
             if (comp == 0)
                   return tree [temp];
             else if (comp < 0)
                   temp = tree [temp].left;
             else
                   temp = tree [temp].right;
       } // while
       return  null;
} // method getEntry
```

You will also need to modify the `TreeIterator` class.

Programming Project 10.2

Printing a `BinarySearchTree` Object

In the `BinarySearchTree` class, implement the following method:

```
/**
 *  Returns a String representation of this BinarySearchTree object.
 *  The worstTime(n) is linear in n.
 *
 *  @return a String representation – that incorporates the structure-of this
 *          BinarySearchTree object.
 *
 */
public String toTreeString()
```

Note 1: The `String` returned should incorporate the structure of the tree. For example, suppose we have the following:

```
BinarySearchTree<Integer> tree = new BinarySearchTree<Integer>();

tree.add (55);
tree.add (12);
tree.add (30);
tree.add (97);
System.out.println (tree.toTreeString());
```

The output would be:

```
                          55


                12                    97



                    30
```

In what sense is the above approach better than developing a `printTree` method in the `BinarySearchTree` class?

Programming Project 10.3

The `fixAfterInsertion` Method

Test and define the `fixAfterInsertion` method in the `AVLTree<E>` class. Here is the method specification:

```
/**
 *  Restores the AVLTree properties, if necessary, by rotations and balance-
 *  factor adjustments between a specified inserted entry and the specified nearest
 *  ancestor of inserted that has a balanceFactor of 'L' or 'R'.
```

(continued on next page)

(continued from previous page)

```
 *  The worstTime(n) is O(log n).
 *
 *  @param inserted – the specified inserted AVLEntry object.

 *  @param imbalanceAncestor – the specified AVLEntry object that is the
 *         nearest ancestor of inserted.
 *
 */
protected void fixAfterInsertion (AVLEntry<E> inserted,
                                  AVLEntry<E> imbalanceAncestor)
```

Hint: If `imbalanceAncestor` is **`null`**, then each ancestor of the inserted `AVLEntry` object has a `balanceFactor` value of '='. For example, Figure 10.34 shows the before-and-after for this case.

There are three remaining cases when the `balanceFactor` value of `imbalanceAncestor` is 'L'. The three cases when that value is 'R' can be obtained by symmetry.

Case 1: `imbalanceAncestor.balanceFactor` is 'L' and the insertion is made in the right subtree of `imbalanceAncestor`. Then no rotation is needed. Figure 10.35 shows the before-and-after for this case.

Case 2: `imbalanceAncestor.balanceFactor` is 'L' and the insertion is made in the left subtree of the left subtree of `imbalanceAncestor`. The restructuring can be accomplished with a right rotation around `imbalanceAncestor`. Figure 10.36 shows the before-and-after in this case.

Case 3: `imbalanceAncestor.balanceFactor` is 'L' and the inserted entry is in the right subtree of the left subtree of `imbalanceAncestor`. The restructuring can be accomplished with a left rotation around the left child of `imbalanceAncestor` followed by a right rotation around `imbalanceAncestor`. There are three subcases to determine the adjustment of balance factors:

3a: `imbalanceAncestor`'s post-rotation parent is the inserted entry. Figure 10.37 shows the before-and-after in this case.

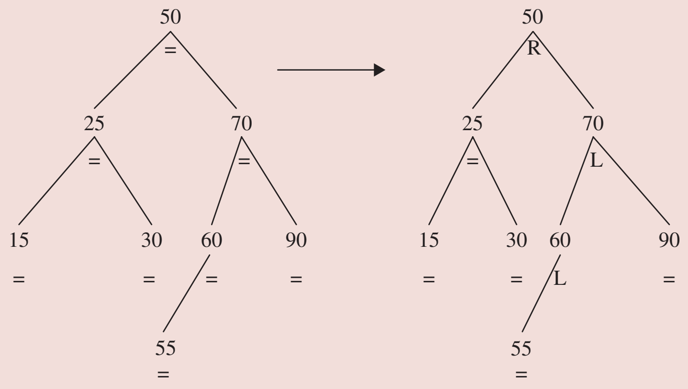


FIGURE 10.34 On the left-hand side, an `AVLTree` object just before the call to `fixAfterInsertion`; the element inserted was 55, and all of its ancestors have a balance factor of '='. On the right-hand side, the `AVLTree` object with adjusted balance factors

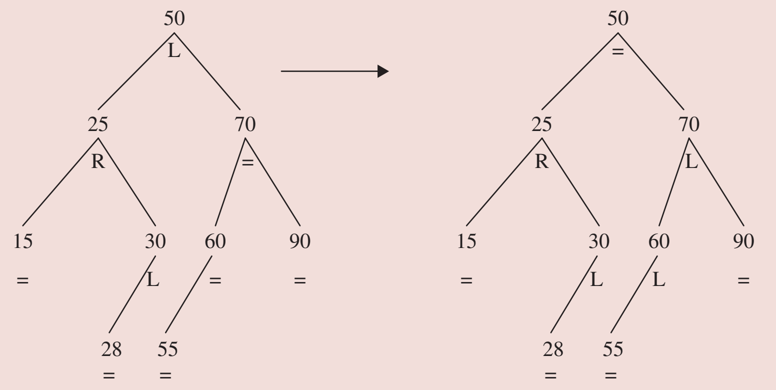


FIGURE 10.35 On the left-hand side, an AVL tree into which 55 has just been inserted. The balance factors of the other entries are *pre-insertion*. On the right-hand side, the same AVL tree after the balance factors have been adjusted. The only balance factors adjusted are those in the path between 55 (exclusive) and 50 (inclusive)

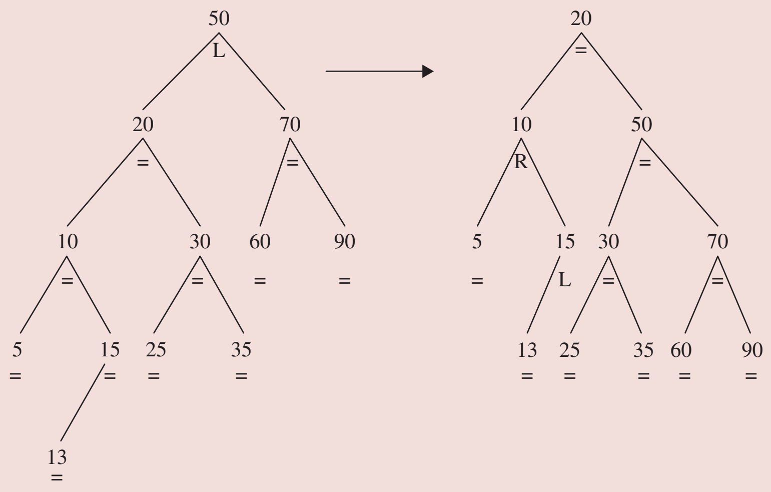


FIGURE 10.36 On the left side, what was an AVL tree has become imbalanced by the insertion of 13. The balance factors of the other entries are *pre-insertion*. In this case, `imbalanceAncestor` is the `AVLEntry` object whose element is 50. On the right side, the restructured AVL tree with adjusted balanced factors

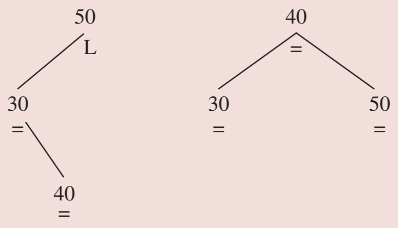


FIGURE 10.37 On the left side, what was an AVL tree has become imbalanced by the insertion of 40. The balance factors of the other entries are *pre-insertion*. In this sub-case, `imbalanceAncestor` is the `AVLEntry` object whose element is 50. On the right side, the restructured AVL tree with adjusted balanced factors

(continued on next page)

(continued from previous page)

3b: The inserted element is less than `imbalanceAncestor`'s post-rotation parent. Figure 10.38 shows the before-and-after in this subcase.

3c: The inserted element is greater than `imbalanceAncestor`'s post-rotation parent. Figure 10.39 shows the before-and-after in this subcase.

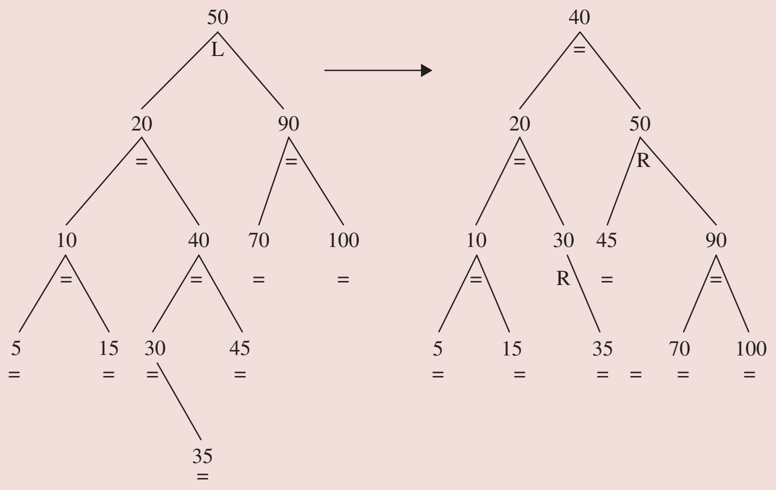


FIGURE 10.38 On the left side, what was an AVL tree has become imbalanced by the insertion of 35. The balance factors of the other entries are *pre-insertion*. In this case, *imbalanceAncestor* is the *AVLEntry* object whose element is 50. On the right side, the restructured AVL tree with adjusted balanced factors

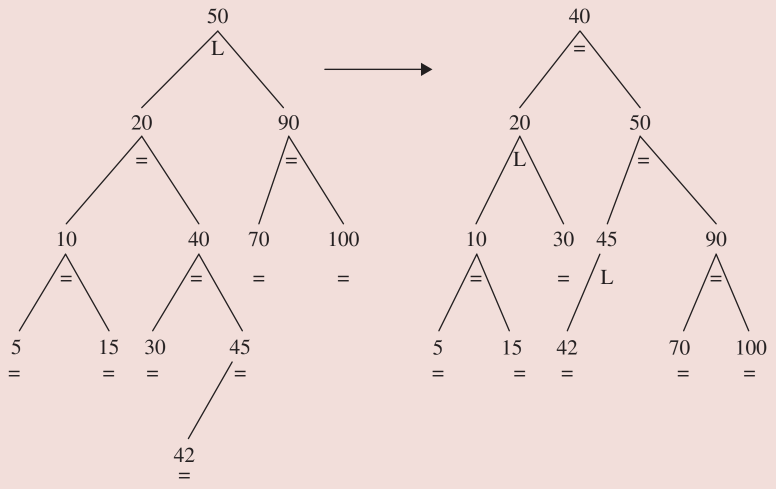


FIGURE 10.39 On the left side, what was an AVL tree has become imbalanced by the insertion of 42. The balance factors of the other entries are *pre-insertion*. In this case, `imbalanceAncestor` is the `AVLEntry` object whose element is 50. On the right side, the restructured AVL tree with adjusted balanced factors

Programming Project 10.4

The `fixAfterDeletion` Method

Test and define the `fixAfterDeletion` method in the `AVLTree<E>` class. Here is the method specification:

```
/**
 *  Restores the AVL properties, if necessary, by rotations and balance-factor
 *  adjustments between the element actually deleted and a specified ancestor
 *  of the AVLEntry object actually deleted.
 *  The worstTime(n) is O(log n).
 *
 *  @param element – the element actually deleted from this AVLTree object.
 *  @param ancestor – the specified ancestor (initially, the parent) of the
 *         element actually deleted.
 *
 */
protected void fixAfterDeletion (E element, AVLEntry<E> ancestor)
```

Hint: Loop until the tree is an AVL tree with appropriate balance factors. Within the loop, suppose the element removed was in the right subtree of `ancestor` (a symmetric analysis handles the left-subtree case). Then there are three subcases, depending on whether `ancestor.balanceFactor` is '=', 'R', or 'L'. In all three subcases, `ancestor.balanceFactor` must be changed. For the '=' subcase, the loop then terminates. For the 'R' subcase, `ancestor` is replaced with `ancestor.parent` and the loop continues. For the 'L' subcase, there are three sub-subcases, depending on whether `ancestor.left.balanceFactor` is '=', 'R', or 'L'. And the 'R' sub-subcase has three sub-sub-subcases.

Sorting

CHAPTER 11

One of the most common computer operations is *sorting*, that is, putting a collection of elements in order. From simple, one-time sorts for small collections to highly efficient sorts for frequently used mailing lists and dictionaries, the ability to choose among various sort methods is an important skill in every programmer's repertoire.

CHAPTER OBJECTIVES

1. Compare the `Comparable` interface to the `Comparator` interface, and know when to use each one.
2. Be able to decide which sort algorithm is appropriate for a given application.
3. Understand the limitations of each sort algorithm.
4. Explain the criteria for a divide-and-conquer algorithm.

11.1 Introduction

Our focus will be on comparison-based sorts; that is, the sorting entails comparing elements to other elements. Comparisons are not necessary if we know, in advance, the final position of each element. For example, if we start with an unsorted list of 100 distinct integers in the range 0 . . . 99, we know without any comparisons that the integer 0 must end up in position 0, and so on. The best- known sort algorithm that is not comparison-based is Radix Sort: see Section 11.5.

All of the sort algorithms presented in this chapter are generic algorithms, that is, **`static`** methods: they have no calling object and operate on the parameter that specifies the collection to be sorted. Two of the sort methods, Merge Sort and Quick Sort, are included in the Java Collections Framework, and can be found in the `Collections` or `Arrays` classes in the package java.util.

The parameter list may include an array of primitive values (**`int`**s or **`double`**s, for example), an array of objects, or a `List` object—that is, an instance of a class that implements the `List` interface. In illustrating a sort algorithm, we gloss over the distinction between an array of **`int`**s, an array of `Integer` objects and a `List` object whose elements are of type `Integer`. In Section 11.3, we'll see how to sort objects by a different ordering than that provided by the `compareTo` method in the `Comparable` interface.

In estimating the efficiency of a sorting method, our primary concerns will be averageTime(n) and worstTime(n). In some applications, such as national defense and life-support systems, the worst-case performance of a sort method can be critical. For example, we will see a sort algorithm that is quite fast, both on average and in the worst case. And we will also look at a sort algorithm that is extremely fast, on average, but whose worst-case performance is achingly slow.

The space requirements will also be noted, because some sort algorithms make a copy of the collection that is to be sorted, while other sort algorithms have only negligible space requirements.

Another criterion we'll use for measuring a sort method is stability. A ***stable*** sort method preserves the relative order of equal elements. For example, suppose we have an array of students in which each student consists of a last name and the total quality points for that student, and we want to sort by total quality points. If the sort method is stable and before sorting, ("Balan", 28) appears at an earlier index than ("Wang", 28), then after sorting ("Balan" 28) will still appear at an earlier index than ("Wang" 28). Stability can simplify project development. For example, assume that the above array is already in order by name, and the application calls for sorting by quality points; for students with the same quality points the ordering should be alphabetical. A stable sort will accomplish this without any additional work to make sure students with the same quality points are ordered alphabetically.

Table 11.1 at the end of the chapter provides a summary of the sorting methods we will investigate. Each sort method will be illustrated on the following collection of 20 `int` values:

59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Here is the method specification and one of the test cases for the `?Sort` method, where `?Sort` can represent any of the sort methods from Section 11.2:

```
/**
  *  Sorts a specified array of int values into ascending order.
  *  The worstTime(n) is O(n * n).
  *
  *  @param x - the array to be sorted.
  *
  *  @throws NullPointerException - if x is null.
  *
  */
public static void ?Sort (int[ ] x)

public void testSample()
{
      int [ ] expected = {12,  16,  17,  32,  33,  40,  43,  44,  46,  46,
                      50,  55,  59,  61,  75,  80,  80,  81,  87,  95};
      int [ ] actual = {59,  46,  32,  80,  46,  55,  50,  43,  44,  81,
                      12,  95,  17,  80,  75,  33,  40,  61,  16,  87};
      Sorts.?Sort (actual);
      assertArrayEquals (expected, actual);
} // method testSample
```

The remaining test cases can be found on the book's website.

11.2 Simple Sorts

We'll start with a few sort algorithms that are fairly easy to develop, but provide slow execution time when n is large. In each case, we will sort an array of `int` values into ascending order, and duplicates will be allowed. These algorithms could easily be modified to sort an array of `double`s, for example, into ascending order or into descending order. We could also sort objects in some class that implements the `Comparable` interface. The ordering, provided by the `compareTo` method, would be "natural": for example, `String` objects would be ordered lexicographically. In Section 11.3, we'll see how to achieve a different ordering of objects than the one provided by the `compareTo` method.

11.2.1 Insertion Sort

Insertion Sort repeatedly sifts out-of-place elements down into their proper indexes in an array. Given an array `x` of **`int`** values, `x [1]` is inserted where it belongs relative to `x [0]`, so `x [0]`, and `x [1]` will be swapped if `x [0]` > `x [1]`. At this point, we have `x [0]` <= `x [1]`. Then `x [2]` will be inserted where it belongs relative to `x [0]` and `x [1]`; there will be 0, 1 or 2 swaps. At that point, we have `x [0]` <= `x [1]` <= `x [2]`. Then `x [3]` will be inserted where it belongs relative to `x [0]` . . . `x [2]`. And so on.

Example Suppose the array `x` initially has the following values:

59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

We first place `x [1]`, 46, where it belongs relative to `x [0]`, 59. One swap is needed, and this gives us:

46 59 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

The underlined values are in their correct order. We then place `x [2]`, 32, where it belongs relative to the sorted subarray `x [0] ... x [1]`, and two swaps are required. We now have

32 46 59 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Next, we place `x [3]`, 80, where it belongs relative to the sorted subarray `x [0] ... x [2]`; this step does not require any swaps, and the array is now

32 46 59 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

We then place `x [4]`, 46, where it belongs relative to the sorted subarray `x [0] ... x [3]`. Two swaps are required, and we get

32 46 46 59 80 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

This process continues until, finally, we place `x [19]`, 87, where it belongs relative to the sorted subarray `x [0] ... x [18]`. The array is now sorted:

12 16 17 32 33 40 43 44 46 46 50 55 59 61 75 80 80 81 87 95

At each stage in the above process, we have an **`int`** variable `i` in the range 1 through 19, and we place `x [i]` into its proper position relative to the sorted subarray `x [0]`, `x [1]`,..., `x [i-1]`. During each iteration, there is another loop in which an **`int`** variable `k` starts at index `i` and works downward until either `k` = 0 or `x [k - 1]` <= `x[k]`. During each inner-loop iteration, `x [k]` and `x [k -1]` are swapped.

Here is the method definition:

```
/**
 *  Sorts a specified array of int values into ascending order.
 *  The worstTime(n) is O(n * n).
 *
 *  @param x - the array to be sorted.
 *
 *  @throws NullPointerException - if x is null.
 *
 */
public static void insertionSort (int[ ] x)
```

```
{
      for (int i = 1; i < x.length; i++)
             for (int k = i; k > 0 && x [k -1] > x [k]; k--)
                   swap (x, k, k -1);
} // method insertionSort
```

The definition of the swap method is:

```
/**
 *  Swaps two specified elements in a specified array.
 *
 *  @param x - the array in which the two elements are to be swapped.
 *  @param a - the index of one of the elements to be swapped.
 *  @param b - the index of the other element to be swapped.
 *
 */
public static void swap (int [ ] x, int a, int b)
{
      int t = x[a];
      x[a] = x[b];
      x[b] = t;
} // method swap
```

For example, if scores is an array of **int** values, we could sort the array with the following call:

```
insertionSort (scores);
```

Analysis Let n be the number of elements to be sorted. The outer **for** loop will be executed exactly $n{-}1$ times. For each value of i, the number of iterations of the inner loop is equal to the number of swaps required to sift x [i] into its proper position in x [0], x [1], ..., x [i-1]. In the worst case, the collection starts out in decreasing order, so i swaps are required to sift x [i] into its proper position. That is, the number of iterations of the inner **for** loop will be

$$1 + 2 + 3 + \cdots + n - 2 + n - 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$$

The total number of outer-loop and inner-loop iterations is $n-1 + n(n-1)/2$, so worstTime (n) is quadratic in n. In practice, what really slows down insertionSort in the worst case is that the number of *swaps* is quadratic in n. But these can be replaced with single assignments (see Concept Exercise 11.9).

To simplify the average-time analysis, assume that there are no duplicates in the array to be sorted. The number of inner-loop iterations is equal to the number of swaps. When x [1] is sifted into its proper place, half of the time there will be a swap and half of the time there will be no swap.[1] Then the expected number of inner-loop iterations is (0 + 1)/2.0, which is 1/2.0. When x [2] is sifted into its proper place, the expected number of inner-loop iterations is (0 + 1 + 2)/3.0, which is 2/2.0. In general, when sifting x [i] to its proper position, the expected number of loop iterations is

$$(0 + 1 + 2 + \cdots + \mathrm{i})/(\mathrm{i} + 1.0) = \mathrm{i}/2.0$$

The total number of inner-loop iterations, on average, is

$$1/2.0 + 2/2.0 + 3/2.0 + \cdots + (n - 1)/2.0 = \sum_{i=1}^{n-1} i/2.0 = n(n-1)/4.0$$

[1]As always in averaging, we assume that each event is equally likely.

We conclude that averageTime (n) is quadratic in n. In the Java Collections Framework, Insertion Sort is used for sorting subarrays of fewer than 7 elements. Instead of a method call, there is inline code (`off` contains the first index of the subarray to be sorted, and `len` contains the number of elements to be sorted):

```
// Insertion sort on smallest arrays
if (len < 7)
{
        for (int i=off; i<len+off; i++)
           for (int k=i; k>off && x[k-1]>x[k]; k--)
               swap(x, j, j-1);
        return;
}
```

For small subarrays, other sort methods—usually faster than Insertion Sort—are actually slower because their powerful machinery is designed for large-sized arrays. The choice of 7 for the cutoff is based on empirical studies described in Bentley [1993]. The best choice for a cutoff will depend on machine-dependent characteristics.

An interesting aspect of Insertion Sort is its best-case behavior. If the original array happens to be in ascending order—of course the sort method does not "know" this—then the inner loop will not be executed at all, and the total number of iterations is linear in n. In general, if the array is already in order or nearly so, Insertion Sort is very quick. So it is sometimes used at the tail end of a sort method that takes an arbitrary array of elements and produces an "almost" sorted array. For example, this is exactly what happens with the `sort` method in C++'s Standard Template Library.

The space requirements for Insertion Sort are modest: a couple of loop-control variables, a temporary for swapping, and an activation record for the call to `swap` (which we lump together as a single variable). So worstSpace (n) is constant; such a sort is called an ***in-place*** sort.

Because the inner loop of Insertion Sort swaps `x [k-1]` and `x [k]` only if `x [k-1] > x [k]`, equal elements will not be swapped. That is, Insertion Sort is stable.

11.2.2 Selection Sort

Perhaps the simplest of all sort algorithms is Selection Sort: Given an array `x` of **`int`** values, swap the smallest element with the element at index 0, swap the second smallest element with the element at index 1, and so on.

Example Suppose the array `x` initially has the usual values, with an arrow pointing to the element at the current index, and the smallest value from that index on in boldface:

59 46 32 80 46 55 50 43 44 81 **12** 95 17 80 75 33 40 61 16 87
↑

The smallest value in the array, 12, is swapped with the value 59 at index 0, and we now have (with the sorted subarray underlined)

$\underline{12}$ 46 32 80 46 55 50 43 44 81 59 95 17 80 75 33 40 61 **16** 87
      ↑

Now 16, the smallest of the values from index 1 on, is swapped with the value 46 at index 1:

$$\underline{12\ \ 16}\ \ \underset{\uparrow}{32}\ \ 80\ \ 46\ \ 55\ \ 50\ \ 43\ \ 44\ \ 81\ \ 59\ \ 95\ \ \mathbf{17}\ \ 80\ \ 75\ \ 33\ \ 40\ \ 61\ \ 46\ \ 87$$

Then 17, the smallest of the values from index 2 on, is swapped with the value 32 at index 2:

$$\underline{12\ \ 16\ \ 17}\ \ \underset{\uparrow}{80}\ \ 46\ \ 55\ \ 50\ \ 43\ \ 44\ \ 81\ \ 59\ \ 95\ \ \mathbf{32}\ \ 80\ \ 75\ \ 33\ \ 40\ \ 61\ \ 46\ \ 87$$

Finally, during the 19th loop iteration, 87 will be swapped with the value 95 at index 18, and the whole array will be sorted:

$$\underline{12\ \ 16\ \ 17\ \ 32\ \ 33\ \ 40\ \ 43\ \ 44\ \ 46\ \ 46\ \ 50\ \ 55\ \ 59\ \ 61\ \ 75\ \ 80\ \ 80\ \ 81\ \ 87\ \ 95}$$

In other words, for each value of `i` between 0 and `x.length - 1`, the smallest value in the subarray from `x [i]` to `x [x.length -1]` is swapped with `x [i]`. Here is the method definition:

```
/**
 *  Sorts a specified array of int values into ascending order.
 *  The worstTime(n) is O(n * n).
 *
 *  @param x - the array to be sorted.
 *
 *  @throws NullPointerException - if x is null.
 *
 */
public static void selectionSort (int [ ] x)
{
     // Make x [0 ... i] sorted and <= x [i + 1] ...x [x.length -1]:
     for (int i = 0; i < x.length -1; i++)
     {
           int pos = i;
           for (int k = i + 1; k < x.length; k++)
                 if (x [k] < x [pos])
                         pos = k;
           swap (x, i, pos);
     } // for i
} // method selectionSort
```

Analysis First, note that the number of loop iterations is independent of the initial arrangement of elements, so worstTime(n) and averageTime(n) will be identical. There are n–1 iterations of the outer loop; when the smallest values are at indexes x[0], x[1], . . . x[n–2], the largest value will automatically be at index x[n–1]. During the first iteration, with `i` = 0, there are n–1 iterations of the inner loop. During the second iteration of the outer loop, with `i` = 1, there are n–2 iterations of the inner loop. The total number of inner-loop iterations is

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$$

We conclude that worstTime(n) is quadratic in n. For future reference, note that only n–1 swaps are made.

The worstSpace(n) is constant: only a few variables are needed. But Selection Sort is not stable; see Concept Exercise 11.14.

As we noted in Section 11.2.1, Insertion Sort requires only linear-in-n time if the array is already sorted, or nearly so. That is a clear advantage over Selection Sort, which always takes quadratic-in-n time. In the average case or worst case, Insertion Sort takes quadratic-in-n time, and so a run-time experiment is needed to distinguish between Insertion Sort and Selection Sort. You will get the opportunity to do this in Lab 18.

11.2.3 Bubble Sort

Warning: Do not use this method. Information on Bubble Sort is provided to illustrate a very inefficient algorithm with an appealing name. In this section, you will learn why Bubble Sort should be avoided, so you can illuminate any unfortunate person who has written, used, or even mentioned Bubble Sort.

Given an array `x` of **`int`** values, compare each element to the next element in the array, swapping where necessary. At this point, the largest value will be at index `x.length -1`. Then start back at the beginning, and compare and swap elements. To avoid needless comparisons, go only as far as the last interchange from the previous iteration. Continue until no more swaps can be made: the array will then be sorted.

Example Suppose the array `x` initially has the following values:

59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Because 59 is greater than 46, those two elements are swapped, and we have

46 59 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

Then 59 and 32 are swapped, 59 and 80 are not swapped, 80 and 46 (at index 4) are swapped, and so on. After the first iteration, `x` contains

46 32 59 46 55 50 43 44 80 12 81 17 80 75 33 40 61 16 87 95

The last swap during the first iteration was of the elements 95 and 87 at indexes 18 and 19, so in the second iteration, the final comparison will be between the elements at indexes 17 and 18. After the second iteration, the array contains

32 46 46 55 50 43 44 59 12 80 17 80 75 33 40 61 16 81 87 95

The last swap during the second iteration was of the elements 81 and 16 at indexes 16 and 17, so in the third iteration, the final comparison will be between the elements at indexes 15 and 16.

Finally, after 18 iterations, and many swaps, we end up with

12 16 17 32 33 40 43 44 46 46 50 55 59 61 75 80 80 81 87 95

Here is the method definition:

```
/**
 *  Sorts a specified array of int values into ascending order.
 *  The worstTime(n) is O(n * n).
 *
 *  @param x - the array to be sorted.
 *
```

```
     *  @throws NullPointerException - if x is null.
     *
     */
    public static void bubbleSort (int[ ] x)
    {
            int finalSwapPos = x.length - 1,
                swapPos;
            while (finalSwapPos > 0)
            {
                   swapPos = 0;
                   for (int i = 0; i < finalSwapPos; i++)
                         if (x [i] > x [i + 1])
                         {
                                swap (x, i, i + 1);
                                swapPos = i;
                         } // if
                   finalSwapPos = swapPos;
            } // while
     } // method bubbleSort
```

Analysis If the array starts out in reverse order, then there will be n–1 swaps during the first outer-loop iteration, n–2 swaps during the second outer-loop iteration, and so on. The total number of swaps, and inner-loop iterations, is

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$$

We conclude that worstTime(n) is quadratic in n.

What about averageTime(n)? The average number of inner-loop iterations, as you probably would have guessed (!), is

$$(n^2-n)/2-(n+1)\ \ln(n+1)/2$$
$$+ (n+1)/2 * (\ln 2 + \lim_{k\to\infty}\left(\sum_{i=1}^{k}(1/i)-\ln k\right)$$
$$+ (2/3)\sqrt{(2\pi(n+1))} + 31/36 + \text{some terms in } O(n^{-1/2}).$$

What is clear from the first term in this formula is that averageTime(n) is quadratic in n.

It is not a big deal, but Bubble Sort is very efficient if the array happens to be in order. Then, only n inner-loop iterations (and no swaps) take place. What if the entire array is in order, except that the smallest element happens to be at index `x.length -1` ? Then $n(n-1)/2$ inner-loop iterations still occur!

Swaps take place when, for some index i, the element at index `i` is greater than the element at index `i + 1`. This implies that Bubble Sort is stable. And with just a few variables needed (the space for the array was allocated in the calling method), worstSpace(n) is constant.

What drags Bubble Sort down, with respect to run-time performance, is the large number of swaps that occur, even in the average case. You will get first-hand experience with Bubble Sort's run-time sluggishness if you complete Lab 18. As Knuth [1973] says, "In short, the bubble sort seems to have nothing going for it, except a catchy name and the fact that it leads to some interesting theoretical problems."

11.3 The Comparator Interface

Insertion Sort, Selection Sort and Bubble Sort produce an array of **int** values in ascending order. We could easily modify those methods to sort into descending order. Similarly straightforward changes would allow us to sort arrays of values from other primitive types, such as **long** or **double**. What about sorting objects? For objects in a class that implements the Comparable interface, we can sort by the "natural" ordering, as described in Section 10.1.1. For example, here is the heading for a Selection Sort that sorts an array of objects:

```
/**
 *  Sorts a specified array of objects into ascending order.
 *  The worstTime(n) is O(n * n).
 *
 *  @param x - the array to be sorted.
 *
 *  @throws NullPointerException - if x is null.
 *
 */
public static void selectionSort (Object [ ] x)
```

For the definition of this version, we replace the line

```
if (x [k] < x [pos])
```

in the original version with

```
if (((Comparable)x [k]).compareTo (x [pos]) < 0)
```

and change heading of the swap method and the type of temp in that method. See Programming Exercise 11.3.

As we saw in Section 10.1.1, the String class implements the Comparable interface with a compareTo method that reflects a lexicographic ordering. If names is an array of String objects, we can sort names into lexicographical order with the call

```
selectionSort (names);
```

This raises an interesting question: What if we did not want the "natural" ordering? For example, what if we wanted String objects ordered by the length of the string?

For applications in which the "natural" ordering—through the Comparable interface—is inappropriate, elements can be compared with the Comparator interface. The Comparator interface, with type parameter T (for "type") has a method to compare two elements of type T:

```
/**
 *  Compares two specified elements.
 *
 *  @param element1 - one of the specified elements.
 *  @param element2 - the other specified element.
 *
 *  @return a negative integer, 0, or a positive integer, depending on
 *          whether element1 is less than, equal to, or greater than
 *          element2.
 *
 */
int compare (T element1, T element2);
```

We can implement the `Comparator` interface to override the natural ordering. For example, we can implement the `Comparator` interface with a `ByLength` class that uses the "natural" ordering for `String` objects of the same length, and otherwise returns the difference in lengths. Then the 3-character string "yes" is considered greater than the 3-character string "and," but less than the 5-character string "maybe." Here is the declaration of `ByLength`:

```
public class ByLength implements Comparator<String>
{
     /**
      *  Compares two specified String objects lexicographically if they have the
      *  same length, and otherwise returns the difference in their lengths.
      *
      *  @param s1 - one of the specified String objects.
      *  @param s2 - the other specified String object.
      *
      *  @return s1.compareTo (s2) if s1 and s2 have the same length;
      *          otherwise, return s1.length() - s2.length().
      *
      */
     public int compare (String s1, String s2)
     {
            int len1 = s1.length(),
                len2 = s2.length();
            if (len1 == len2)
                     return s1.compareTo (s2);
            return len1 - len2;
     } // method compare
} // class ByLength
```

One advantage to using a `Comparator` object is that no changes need be made to the element class: the `compare` method's parameters are the two elements to be ordered. Leaving the element class unchanged is especially valuable when, as with the `String` class, users are prohibited from modifying the class.

Here is a definition of Selection Sort, which sorts an array of objects according to a comparator that compares any two objects:

```
/**
 *  Sorts a specified array into the order specified by a specified Comparator
 *  object.
 *  The worstTime(n) is O(n * n).
 *
 *  @param x - the array to be sorted.
 *  @param comp - the Comparator object used for ordering.
 *
 *  @throws NullPointerException - if x and/or comp is null.
 *
 */
```

```
public static  void selectionSort (T [ ] x, Comparator comp)
{
      // Make x [0 ... i] sorted and <= x [i + 1] ...x [x.length -1]:
      for (int i = 0; i < x.length -1; i++)
      {
             int pos = i;
             for (int k = i + 1; k < x.length; k++)
                    if (comp.compare (x [k], x [pos]) < 0)
                          pos = k;
             swap (x, i, pos);
      } // for i
} // method selectionSort
```

The corresponding swap method is:

```
public static void swap (Object[ ] x, int a, int b)
{
      Object temp = x [a];
      x [a] = x [b];
      x [b] = temp;
} // swap
```

To complete the picture, here is a small program method that applies this version of `selectionSort` (note that the enhanced **for** statement also works for arrays):

```
import java.util.*;  // for the Comparator interface

public class SelectionSortExample
{
      public static void main(String[] args)
      {
            new SelectionSortExample().run();
      } // method main

      public void run()
      {
            String[ ] words = {"Jayden", "Jack", "Rowan", "Brooke"};

            selectionSort (words, new ByLength());
            for (String s : words)
                  System.out.print (s + "  ");
            } // method run

} // class SelectionSortExample
```

The output will be

```
 Jack Rowan Brooke Jayden
```

The material in this section will be helpful in Section 11.4.1, where we will encounter one of the sort methods in the Java Collections Framework. For this method, called Merge Sort, the element type cannot be primitive; it must be (reference to) `Object`, or subclass of `Object`. The method comes in four flavors:

the collection to be sorted can be either an array object or a `List` object, and the ordering may be according to the `Comparable` interface or the `Comparator` interface. And Chapter 13 has a sort method with similar flexibility.

In Section 11.4, we consider two important questions: For comparison-based sorts, is there a lower bound for worstTime(n)? Is there a lower bound for averageTime(n)?

11.4 How Fast Can we Sort?

If we apply Insertion Sort, Selection Sort or (perish the thought) Bubble Sort, worstTime(n) and averageTime(n) are quadratic in n. Before we look at some faster sorts, let's see how much of an improvement is possible. The tool we will use for this analysis is the decision tree. Given n elements to be sorted, a ***decision tree*** is a binary tree in which each non-leaf represents a comparison between two elements and each leaf represents a sorted sequence of the n elements. For example, Figure 11.1 shows a decision tree for applying Insertion Sort in which the elements to be sorted are stored in the variables a_1, a_2, and a_3:

A decision tree must have one leaf for each permutation of the elements to be sorted[2]. The total number of permutations of n elements is $n!$, so if we are sorting n elements, the corresponding decision tree must have $n!$ leaves. According to the Binary Tree Theorem, the number of leaves in any non-empty binary tree t is $<= 2^{\text{height(t)}}$. Thus, for a decision tree t that sorts n elements,

$$n! <= 2^{\text{height}(t)}$$

Taking logs, we get

$$\text{height}(t) >= \log_2(n!)$$

In other words, for any comparison-based sort, there must be a leaf whose depth is at least $\log_2(n!)$. In the context of decision trees, that means that there must be an arrangement of elements whose sorting requires at least $\log_2(n!)$ comparisons. That is, worstTime(n) $>= \log_2(n!)$. According to Concept Exercise 11.7, $\log_2(n!) >= n/2 \log_2(n/2)$, which makes $n/2 \log_2(n/2)$ a lower bound of worstTime(n) for any comparison-based sort. According to the material on lower bounds from Chapter 3, $n/2 \log_2(n/2)$ is $\Omega(n \log n)$. So we can say, crudely, that $n \log n$ is a lower bound for worstTime(n). Formally, we have

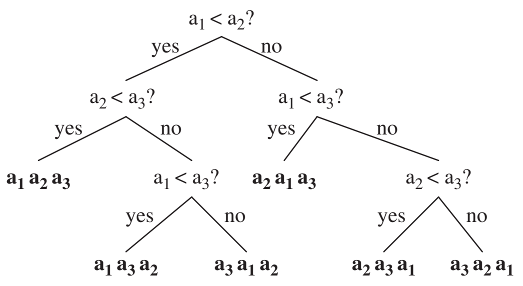


FIGURE 11.1 A decision tree for sorting 3 elements by Insertion Sort

[2]For the sake of simplicity, we assume the collection to be sorted does not contain any duplicates.

Sorting Fact 1:

For comparison-based sorts, worstTime(n) is $\Omega(n \log n)$.

What does Sorting Fact 1 say about upper bounds? We can say, for example, that for any comparison-based sort, worstTime(n) is not O(n). But can we say that for any comparison-based sort, worstTime(n) is O($n \log n$)? No, because for each of the sorts in Section 11.2, worstTime(n) is O(n^2). We cannot even be sure, at this point, that there are any comparison-based sorts whose worstTime(n) is O($n \log n$). Fortunately, this is not some lofty, unattainable goal. For the comparison-based sort algorithms in Sections 11.4.1 and 13.4, worstTime(n) is O($n \log n$). When we combine that upper bound with the lower bound from Sorting Fact 1, we will have several sort methods whose worstTime(n) is linear-logarithmic in n.

What about averageTime(n)? For any comparison-based sort, $n \log n$ is a lower bound of averageTime(n) as well. To obtain this result, suppose t is a decision tree for sorting n elements. Then t has $n!$ leaves. The average, over all $n!$ permutations, number of comparisons to sort the n elements is the total number of comparisons divided by $n!$. In a decision tree, the total number of comparisons is the sum of the lengths of all paths from the root to the leaves. This sum is the external path length of the decision tree. By the External Path Length Theorem in Chapter 9, E(t) >= ($n!/2$) floor ($\log_2(n!)$). So we get, for any positive integer n:

$$\begin{aligned}
\text{averageTime}(n) &>= \text{average number of comparisons}\\
&= \text{E}(t)/n!\\
&>= (n!/2)\text{floor}(\log_2(n!))/n!\\
&= (1/2)\text{floor}(\log_2(n!))\\
&>= (1/4)\log_2(n!)\\
&>= (n/8)\log_2(n/2) \text{ [by Concept Exercise 11.7]}
\end{aligned}$$

We conclude that $n \log n$ is a lower bound of averageTime(n). That is,

Sorting Fact 2:

For comparison-based sorts, averageTime(n) is Ω ($n \log n$).

We noted that there are several sort methods whose worstTime(n) is linear-logarithmic in n. We can use that fact to show that their averageTime(n) must also be linear-logarithmic in n. Why? Suppose we have a sort algorithm for which worstTime(n) is linear-logarithmic in n. That is, crudely, $n \log n$ is both an upper bound and a lower bound of worstTime(n). But averageTime(n) <= worstTime(n), so if $n \log n$ is an upper bound of worstTime(n), $n \log n$ must also be an upper bound of averageTime(n). According to Sorting Fact 2, $n \log n$ is a lower bound on averageTime(n). Since $n \log n$ is both an upper bound and a lower bound of averageTime(n), we conclude that averageTime(n) must be linear-logarithmic in n. That is,

Sorting Fact 3:

For comparison-based sorts, if worstTime(n) is linear-logarithmic in n, then averageTime(n) must be linear-logarithmic in n.

In Sections 11.4.1 and 11.4.3, we will study two sort algorithms, Merge Sort and Quick Sort, whose averageTime(n) is linear-logarithmic in n. For Merge Sort, worstTime(n) is also linear-logarithmic in n, while for Quick Sort, worstTime(n) is quadratic in n. Strangely enough, Quick Sort is generally considered the most efficient all-around sort. Quick Sort's worst-case performance is bad, but for average-case, run-time speed, Quick Sort is the best of the lot.

11.4.1 Merge Sort

The Merge Sort algorithm, in the `Arrays` class of the package `java.util`, sorts a collection of objects. We start with two simplifying assumptions, which we will then dispose of. First, we assume the objects to be sorted are in an array. Second, we assume the ordering is to be accomplished through the `Comparable` interface.

The basic idea is to keep splitting the n-element array in two until, at some step, each of the subarrays has size less than 7 (the choice of 7 is based on run-time experiments, see Bentley [1993]). Insertion Sort is then applied to each of two, small-sized subarrays, and the two, sorted subarrays are merged together into a sorted, double-sized subarray. Eventually, that subarray is merged with another sorted, double-sized subarray to produce a sorted, quadruple-sized subarray. This process continues until, finally, two sorted subarrays of size $n/2$ are merged back into the original array, now sorted, of size n.

Here is the method specification for Merge Sort:

```
/**
 *  Sorts a specified array of objects according to the compareTo method
 *  in the specified class of elements.
 *  The worstTime(n) is O(n log n).
 *
 *  @param a - the array of objects to be sorted.
 *
 */
public static void sort (Object[ ] a)
```

This method has the identifier `sort` because it is the only method in the `Arrays` class of the package `java.util` for sorting an array of objects according to the `Comparable` interface. Later in this chapter we will encounter a different sort method—also with the identifier `sort`—for sorting an array of values from a primitive type. The distinction is easy to make from the context, namely, whether the argument is an array of objects or an array from a primitive type such as **`int`** or **`double`**.

Example To start with a small example of Merge Sort, here are the **`int`** values in an array of `Integer` objects:

59 46 32 80 46 55 87 43 44 81

We use an auxiliary array, `aux`. We first clone the parameter `a` into `aux`. (Cloning is acceptable here because an array object cannot invoke a copy constructor.) We now have two arrays with (separate references to) identical elements. The recursive method `mergeSort` is then called to sort the elements in `aux` back into `a`. Here is the definition of the `sort` method:

```
public static void sort (Object[ ] a)
{
       Object aux[ ] = (Object [ ])a.clone();
       mergeSort (aux, a, 0, a.length);
} // method sort
```

The method specification for `mergeSort` is

```
/**
 *  Sorts, by the Comparable interface, a specified range of a specified array
 *  into the same range of another specified array.
 *  The worstTime(k) is O(k log k), where k is the size of the subarray.
 *
 *  @param src - the specified array whose elements are to be sorted into another
 *         specified array.
 *  @param dest - the specified array whose subarray is to be sorted.
 *  @param low - the smallest index in the range to be sorted.
 *  @param high - 1 + the largest index in the range to be sorted.
 *
 */
private static void mergeSort (Object src[ ], Object dest[ ], int low, int high)
```

The reason we have two arrays is to make it easier to merge two sorted subarrays into a larger subarray. The reason for the **`int`** parameters `low` and `high` is that their values will change when the recursive calls are made. Note that `high`'*s value is one greater than the largest index* of the subarray being mergeSorted.[3]

When the initial call to `mergeSort` is made with the example data, Insertion Sort is not performed because `high` – `low` >= 7. (The number 7 was chosen based on run-time experiments.) Instead, two recursive calls are made:

```
mergeSort (a, aux, 0, 5);
mergeSort (a, aux, 5, 10);
```

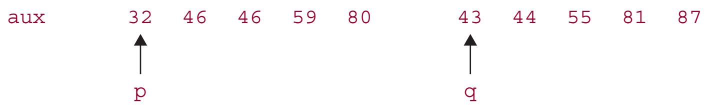
When the first of these calls is executed, `high` –`low` = 5 – 0 < 7, so Insertion Sort is performed on the first five elements of `aux`:

```
a [0 ... 4] = {59, 46, 32, 80, 46}

aux [0 ... 4] = {32, 46, 46, 59, 80}
```

In general, the two arrays will be identical until an Insertion Sort is performed. When the second recursive call is made, `high` –`low` = 10–5, so Insertion Sort is performed on the second five elements of `aux`:

```
a [5 ... 9] = {55, 87, 43, 44, 81}

aux [5 ... 9] = {43, 44, 55, 81, 87}
```

Upon the completion of these two calls to `mergeSort`, the ten elements of `aux`, in two sorted subarrays of size 5, are merged back into `a`, and we are done. The merging is accomplished with the aid of two indexes, `p` and `q`. In this example, `p` starts out as 0 (the low index of the left subarray of `aux`) and `q` starts out as 5 (the low index of the right subarray of `aux`). In the following figure, arrows point from `p` and `q` to the elements at `aux [p]` and `aux [q]`:



[3]Technically, there is a fifth parameter. But since we assume that the entire array is being sorted, we can ignore that parameter.

The smaller of `aux [p]` and `aux [q]` is copied to `a [p]` and then the index, either `p` or `q`, of that smaller element is incremented:

```
aux      32  46  46  59  80      43  44  55  81  87
             ↑                   ↑
             p                   q

a        32
```

The process is repeated: the smaller of `aux [p]` and `aux [q]` is copied to the next location in the array `a`, and then the index, either `p` or `q`, of that smaller element is incremented:

```
aux      32  46  46  59  80      43  44  55  81  87
             ↑                       ↑
             p                       q

a        32  43
```

The next three iterations will copy 44, 46, and 46 into `a`. The process continues until all of the elements from both subarrays have been copied into `a`. Since each iteration copies one element to `a`, merging two subarrays of the same size requires exactly twice as many iterations as the size of either subarray. In general, to merge n/k subarrays, each of size k, requires exactly n iterations.

Figure 11.2 summarizes the sorting the above array of ten elements.

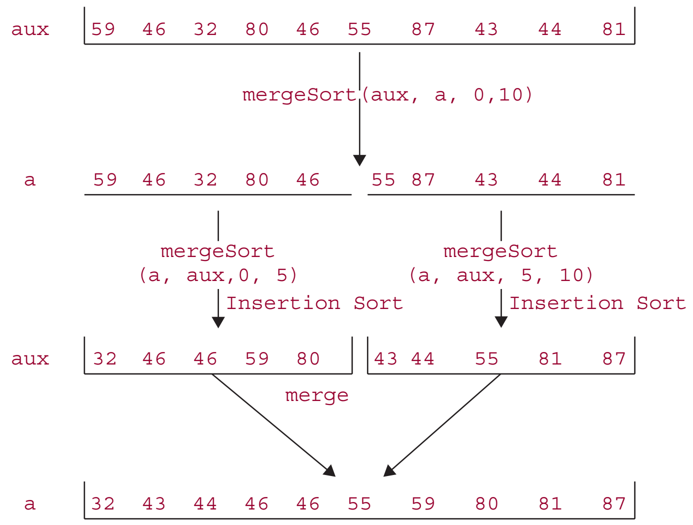


FIGURE 11.2 The effect of a call to `mergeSort` on an array of 10 elements

Figure 11.3 incorporates the above example in merge sorting an array of 20 elements. Two pairs of recursive calls are required to merge sort 20 elements.

The successive calls to `mergeSort` resemble a ping-pong match:

`aux` ⟶ `a` ⟶ `aux` ⟶ `a` ⟶ `aux` ⟶ `a`

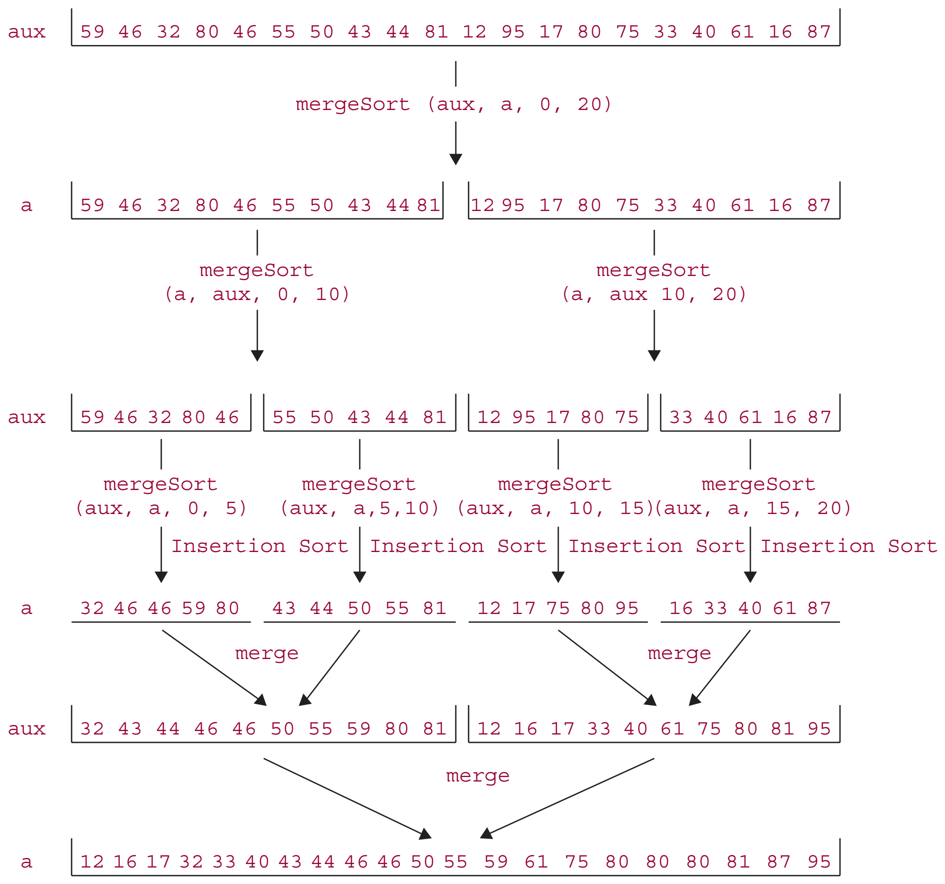


FIGURE 11.3 The effect of a call to `mergeSort` on an array of 20 elements

The original call, from within the `sort` method, is always of the form

```
mergeSort (aux, a, ...);
```

So after all of the recursive calls have been executed, the sorted result ends up in `a`. After the Insertion Sorting has been completed for the two successive subarrays in the recursive call to `mergeSort`, a merge of those sorted subarrays is performed, and that completes a recursive call to `mergeSort`.

Here is the complete `mergeSort` method, including an optimization (starting with `//If left subarray` . . .) that you can safely ignore.

```
/**
 *  Sorts, by the Comparable interface, a specified range of a specified array
 *  into the same range of another specified array.
 *  The worstTime(k) is O(k log k), where k is the size of the subarray.
 *
 *  @param src - the specified array whose range is to be sorted into another
 *         specified array.
```

```
     *  @param dest - the specified array whose subarray is to be sorted.
     *  @param low: the smallest index in the range to be sorted.
     *  @param high: 1 + the largest index in the range to be sorted.
     *
     */
    private static void mergeSort (Object src[ ], Object dest[ ], int low, int high)
    {
        int length = high - low;

        // Use Insertion Sort for small subarrays.
        if (length < INSERTIONSORT_THRESHOLD /* = 7*/)
        {
        for (int i = low; i < high; i++)
                   for (int j = i; j >low && ((Comparable)dest[j - 1])
                                             .compareTo(dest[j]) > 0;  j--)
                          swap (dest, j, j-1);
             return
        } // if length < 7

        // Sort left and right halves of src into dest.
        int mid = (low + high) >> 1;   // >> 1 has same effect as / 2, but is faster
        mergeSort (dest, src, low, mid);
        mergeSort (dest, src, mid, high);

        // If left subarray less than right subarray, copy src to dest.
        if (((Comparable)src [mid-1]).compareTo (src [mid]) <= 0)
        {
             System.arraycopy (src, low, dest, low, length);
             return;
        }

        // Merge sorted subarrays in src into dest.
        for (int i = low, p = low, q = mid; i < high; i++)
             if (q>=high || (p<mid && ((Comparable)src[p]).compareTo (src[q])<= 0))
                    dest [i] = src [p++];
             else
                    dest[i] = src[q++];
    } // method mergeSort
```

Analysis We want to get an upper bound on worstTime(*n*), where *n* is the number of elements to be sorted. There are four phases: cloning, calls to `mergeSort`, Insertion Sorting, and merging.

The cloning requires n iterations.

Let L (for "levels") be the number of pairs of recursive calls to `mergeSort`. The initial splitting into subarrays requires approximately L statements (that is, L pairs of recursive calls). L is equal to the number of times that n is divided by 2. By the Splitting Rule from Chapter 3, the number of times that n can be divided by 2 until n equals 1 is $\log_2 n$. But when the size of a subarray is less than 7, we stop dividing by 2. So L is approximately $\log_2(n/6)$.

For the Insertion Sort phase, we have fewer than n subarrays, each of size less than 7. The maximum number of iterations executed when Insertion Sort is applied to each one of these subarrays is less than

36 because Insertion Sort's worst time is quadratic in the number of elements. So the total number of iterations for this phase is less than $36n$.

Finally, the merging back into double-sized subarrays takes, approximately, L times the number of iterations per level, that is, $\log_2(n/6)$ times the number of iterations executed at any level. At any level, *exactly* n elements are copied from `a` to `aux` (or from `aux` to `a`). So the total number of iterations is, approximately, $n \log_2(n/6)$.

The total number of iterations is less than

$$n + \log_2(n/6) + 36n + \log_2(n/6) * n$$

From this we conclude that worstTime(n) is less than

$$n + \log_2(n/6) + 36n + n\log_2(n/6).$$

That is, worstTime(n) is O(n log n). By Sorting Fact 1 in Section 13.3, for any comparison-based sort, worstTime(n) is $\Omega(n \log n)$. Since $n \log n$ is both an upper bound and a lower bound of worstTime(n), worstTime(n) must be linear-logarithmic in n (that is, $\Theta(n \log n)$: Big Theta of $n \log n$) That implies, by Sorting Fact 3, that averageTime(n) is also linear-logarithmic in n.

Not only is `mergeSort` as good as you can get in terms of estimates of worstTime(n) and averageTime(n), but the actual number of comparisons made is close to the theoretical minimum (see Kruse [1987], pages 251–254).

What is worstSpace(n)? The temporary array `aux`, of size n, is created before `mergeSort` is called. During the execution of `mergeSort`, activation records are created at each level. At the first level, two activation records are created; at the second level, four activation records are created; and so on. The total number of activation records created is

$$2 + 4 + 8 + 16 + \ldots + 2^L = \sum_{i=1}^{L} 2^i = 2^{L+1} - 2$$

(The result on the sum of powers of 2 is from Exercise A2.6 in Appendix 2). Since L $\sim \log_2(n/7)$, and

$$2^{\log_2 n} = n$$

we conclude that the total number of activation records created is linear in n. When we add up the linear-in-n space for `aux` and the linear-in-n space for activation records, we conclude that worstSpace(n) is linear in n.

Both the Insertion Sorting phase and the merging phase preserve the relative order of elements. That is, `mergeSort` is a stable sort.

11.4.1.1 Other Merge Sort Methods

The `Arrays` class also has a version of Merge Sort that takes a `Comparator` parameter:

```
public static  void sort (T [ ], Comparator<? super T> c)
```

The essential difference between this version and the `Comparable` version is that an expression such as

```
(Comparable)dest[j-1]).compareTo(dest[j])>0
```

is replaced with

```
c.compare(dest[j-1], dest[j])>0
```

For example, suppose `words` is an array of `String` objects. To perform Merge Sort on `words` by the lengths of the strings (but lexicographically for equal-length strings), we utilize the `ByLength` class from Section 11.3:

```
Arrays.sort (words, new ByLength());
```

The `Collections` class, also in the package `java.util`, has two versions—depending on whether or not a comparator is supplied—of a Merge Sort method. Each version has a `List` parameter and starts by copying the list to an array. Then the appropriate version of `sort` from the `Arrays` class is called. Finally, during an iteration of the list, each element is assigned the value of the corresponding element in the array. Here, for example, is the `Comparator` version:

```
/**
 *  Sorts a specified List object of elements from class E according to a
 *  specified Comparator object.
 *  The worstTime(n) is O(n log n).
 *
 *  @param list - the List object to be sorted.
 *  @param c - the Comparator object that determines the ordering of elements.
 *
 */
public static  void sort (List list, Comparator<? super T> c)
{
     Object a[ ] = list.toArray();
     Arrays.sort(a, c);
     ListIterator i = list.listIterator();
     for (int j=0; j<a.length; j++)
     {
            i.next();
            i.set(a[j]);
     } // for
} // method sort
```

Both versions of Merge Sort in the `Collections` class work for any class that implements the `List` interface, such as `ArrayList` and `LinkedList`. The run-time will be somewhat slower than for the `Arrays` -class versions because of the copying from the list to the array before sorting and the copying from the array to the list after sorting.

One limitation to the current versions of Merge Sort is that they do not allow an array of primitives to be merge sorted. The effect of this restriction can be overcome by merge sorting the corresponding array of objects. For example, to Merge Sort an array of **`int`** values, create an array of `Integer` objects, convert each **`int`** value to the corresponding `Integer` object, apply Merge Sort to the array of `Integer` objects, then convert the `Integer` array back to an array of **`int`** values. But this roundabout approach will increase the run time for merge sorting.

11.4.2 The Divide-and-Conquer Design Pattern

The `mergeSort` method is an example of the Divide-and-Conquer design pattern. Every ***divide-and-conquer*** algorithm has the following characteristics:

- the method consists of at least two recursive calls to the method itself;

- the recursive calls are independent and can be executed in parallel;
- the original task is accomplished by combining the effects of the recursive calls.

In the case of `mergeSort`, the sorting of the left and right subarrays can be done separately, and then the left and right subarrays are merged, so the requirements of a divide-and-conquer algorithm are met.

How does a divide-and-conquer algorithm differ from an arbitrary recursive algorithm that includes two recursive calls? The difference is that, for an arbitrary recursive algorithm, the recursive calls need not be independent. For example, in the Towers of Hanoi problem, *n–1* disks had to be moved from the source to the temporary pole *before* the same *n–1* disks could be moved from the temporary pole to the destination. Note that the original Fibonacci method from Lab 7 was a divide-and-conquer algorithm, but the fact that the two method calls were independent was an indication of the method's gross inefficiency.

Section 11.4.3 has another example of the divide-and-conquer design pattern.

11.4.3 Quick Sort

One of the most efficient and, therefore, widely used sorting algorithms is Quick Sort, developed by C.A.R. Hoare [1962]. The generic algorithm `sort` is a Quick Sort algorithm based on "Engineering a Sort Function" (see Bentley [1993]). In the `Arrays` class, Quick Sort refers to any method named `sort` whose parameter is an array of primitive values, and Merge Sort refers to any method named `sort` whose parameter is an array of objects.

There are seven versions[4] of Quick Sort in the `Arrays` class: one for each primitive type (**`int`**, **`byte`**, **`short`**, **`long`**, **`char`**, **`double`**, and **`float`**) except **`boolean`**. The seven versions are identical, except for the specific type information; there is no code re-use. We will illustrate Quick Sort on the **`int`** version and, for simplicity, assume that the entire array is to be sorted. The actual code, somewhat harder to follow, allows a specified subarray to be sorted. Here is the (simplified) method specification and definition:

```
/**
 *  Sorts a specified array of int values into ascending order.
 *  The worstTime(n) is O(n * n), and averageTime(n) is O(n log n).
 *
 *  @param a - the array to be sorted.
 *
 */
public static void sort (int[ ] a)
{
     sort1(a, 0, a.length);
} // method sort
```

The **`private`** `sort1` method has the following method specification:

```
/**
 *  Sorts into ascending order the subarray of a specified array, given
 *  an initial index and subarray length.
 *  The worstTime(n) is O(n * n) and averageTime(n) is O(n log n),
```

[4]Actually, there are fourteen versions, because for each primitive type, there is a version that allows Quick Sort to be applied to an entire array, and another version for a specified subarray.

```
     *  where n is the length of the subarray to be sorted.
     *
     *  @param x - the array whose subarray is to be sorted.
     *  @param off - the start index in x of the subarray to be sorted.
     *  @param len - the length of the subarray to be sorted.
     *
     */
    private static void sort1(int x[ ], int off, int len)
```

The basic idea behind the `sort1` method is this: we first partition the array `x` into a left subarray and a right subarray so that each element in the left subarray is less than or equal to each element in the right subarray. The sizes of the subarrays need not be the same. We then Quick Sort the left and right subarrays, and we are done. Since this last statement is easily accomplished with two recursive calls to `sort1`, we will concentrate on the partitioning phase.

Let's start with the essentials; in Section 11.4.3.1, we'll look at some of the finer points. The first task in partitioning is to choose an element, called the ***pivot***, that each element in `x` will be compared to. Elements less than the pivot will end up in the left subarray, and elements greater than the pivot will end up in the right subarray. Elements equal to the pivot may end up in either subarray.

What makes Quick Sort fast? With other sorts, it may take many comparisons to put an element in the general area where is belongs. But with Quick Sort, a partition can move many elements close to where they will finally end up. This assumes that the value of the pivot is close to the median[5] of the elements to be partitioned. We could, of course, sort the elements to be partitioned and then select the median as the pivot. But that begs the question of how we are going to sort the elements in the first place.

How about choosing `x [off]`—the element at the start index of the subarray to be sorted—as the pivot? If the elements happen to be in order (a common occurrence), that would be a bad choice. Why? Because the left subarray would be empty after partitioning, so the partitioning would reduce the size of the array to be sorted by only one. Another option is to choose `x [off + len/2]` as the pivot, that is, the element in the middle position. If the range happens to be in order, that is the perfect choice; otherwise, it is as good a blind choice as any other.

With a little extra work, we can substantially increase the likelihood that the pivot will split the range into two subarrays of approximately equal size. The pivot is chosen as the median of the elements at indexes `off`, `off + len/2`, and `off + len -1`. The median of those three elements is taken as a simply calculated estimate of the median of the whole range.

Before looking at any more details, let's go through an example.

Example We start with the usual sample of twenty values given earlier:

59 46 32 80 46 55 50 43 44 81 12 95 17 80 75 33 40 61 16 87

In this case, we choose the median of the three **`int`** values at indexes 0, 10, and 19. The median of 59, 12, and 87 is 59, so that is the original pivot.

[5]The *median* of a collection of values is the value that would be in the middle position if the collection were sorted. For example, the median of

100 32 77 85 95

is 85. If the collection contains an even number of values, the median is the average of the two values that would be in the two middle positions if the collection were sorted. For example, the median of

100 32 77 85 95 80

is 82.5.

We now want to move to the left subarray all the elements that are less than 59 and move to the right subarray all the elements that are greater than 59. Elements with a value of 59 may end up in either subarray, and the two subarrays need not have the same size.

To accomplish this partitioning, we create two counters: `b`, which starts at `off` and moves upward, and `c`, which starts at `off + len - 1` and moves downward. There is an outer loop that contains two inner loops. The first of these inner loops increments `b` until `x [b] >= pivot`. Then the second inner loop decrements `c` until `x [c] <= pivot`. If `b` is still less than or equal to `c` when this second inner loop terminates, `x [b]` and `x [c]` are swapped, `b` is incremented, `c` is decremented, and the outer loop is executed again. Otherwise, the outer loop terminates.

The reason we loop until `x [b] >= pivot` instead of `x [b] > pivot` is that there might not be any element whose value is greater than the pivot. In Section 11.4.3.1, we'll see a slightly different loop condition to avoid stopping at, and therefore swapping, the pivot.

For the usual sample of values, `pivot` has the value 59, `b` starts at 0 and `c` starts at 19. In Figure 11.4, arrows point from an index to the corresponding element in the array:

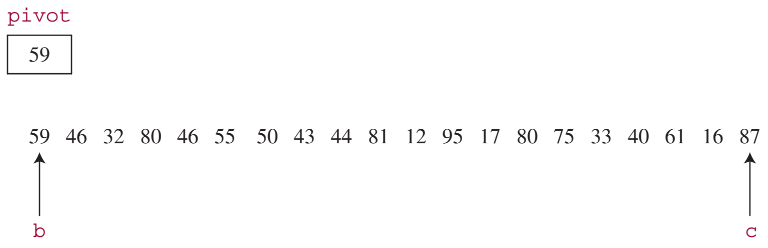


FIGURE 11.4 The start of partitioning

The first inner loop terminates immediately because `x [b]` = 59 and 59 is the pivot. The second inner loop terminates when `c` is decremented to index 18 because at that point, `x [c]` = 16 < 59. When 59 and 16 are swapped and `b` and `c` are bumped, we get the situation shown in Figure 11.5.

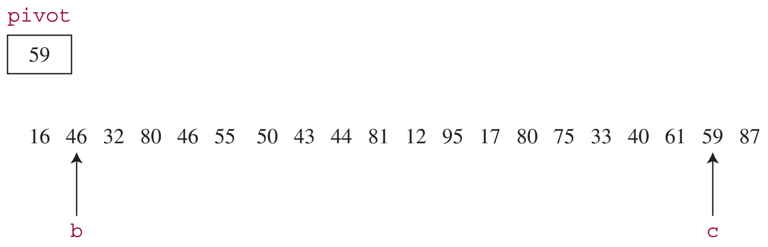


FIGURE 11.5 The state of partitioning after the first iteration of the outer loop

Now `b` is incremented twice more, and at that point we have `x [b]` = 80 > 59. Then `c` is decremented once more, to where `x [c]` = 40 < 59. After swapping `x [b]` with `x [c]` and bumping `b` and `c`, we have the state shown in Figure 11.6.

During the next iteration of the outer loop, `b` is incremented five more times, `c` is not decremented, 81 and 33 are swapped, then the two counters are bumped, and we have the state shown in Figure 11.7.

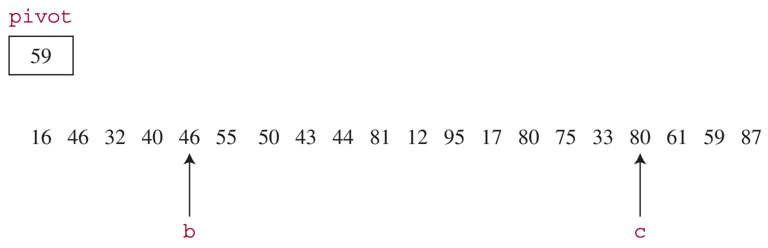


FIGURE 11.6 The state of partitioning after the second iteration of the outer loop

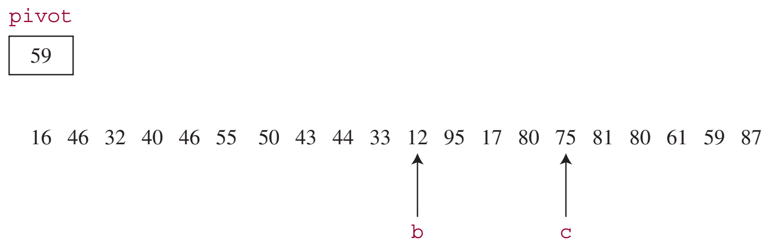


FIGURE 11.7 The state of partitioning after the third iteration of the outer loop

During the next iteration of the outer loop, `b` is incremented once (`x [b]` = 95), and `c` is decremented twice (`x [c]` = 17). Then 95 and 17 are swapped, and `b` and `c` are bumped. See Figure 11.8.

All of the elements in the subarray at indexes 0 through `c` are less than or equal to the pivot, and all the elements in the subarray at indexes `b` through 19 are greater than or equal to the pivot.

We finish up by making two recursive calls to `sort1`:

```
sort1 (x, off, c + 1 - off);           // for this example, sort1 (x, 0, 12);
sort1 (x, b, off + len -b);            // for this example, sort1 (x, 12, 8);
```

In this example, the call to `sort1 (x, off, c + 1 - off)` will choose a new pivot, partition the subarray of 12 elements starting at index 0, and make two calls to `sort1`. After those two calls (and *their* recursive calls) are completed, the call to `sort1 (x, b, off + len -b)` will choose a new pivot, and so on. If we view each pair of recursive calls as the left and right child of the parent call, the execution of the calls in the corresponding binary tree follows a preOrder traversal: the original call, then the left child of that call, then the left child of *that* call, and so on. This leftward chain stops when the subarray to be sorted has fewer than two elements.

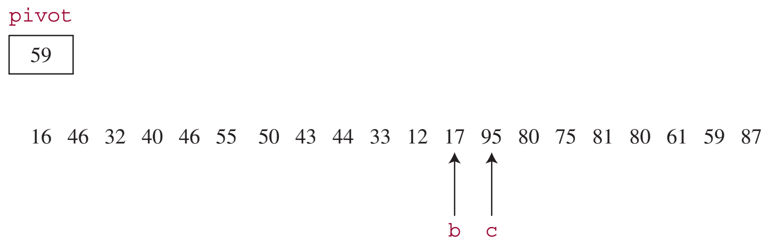


FIGURE 11.8 The state of partitioning after the outer loop is exited

After partitioning, the left subarray consists of the elements from indexes off through c, and the right subarray consists of the elements from indexes b through off + len -1. The pivot need not end up in either subarray. For example, suppose at some point in sorting, the subarray to be partitioned contains

15 45 81

The pivot, at index 1, is 45, and both b and c move to that index in searching for an element greater than or equal to the pivot and less than or equal to the pivot, respectively. Then (wastefully) x [b] is swapped with x [c], b is incremented to 2, and c is decremented to 0. The outer loop terminates, and no further recursive calls are made because the left subarray consists of 15 alone, and the right subarray consists of 81 alone. The pivot is, and remains, where it belongs.

Similarly, one of the subarrays may be empty after a partitioning. For example, if subarray to be partitioned is

15 45

The pivot is 45, both b and c move to that index, 45 is swapped with itself, b is incremented to 2 and c is decremented to 0. The left subarray consists of 15 alone, the pivot is in neither subarray, and the right subarray is empty.

Here is the method definition for the above-described version of sort1 (the version in the Arrays class has a few optimizations, discussed in Section 11.4.3.1):

```
/**
 *  Sorts into ascending order the subarray of a specified array, given
 *  an initial index and subarray length.
 *  The worstTime(k) is O(n * n) and averageTime(n) is O(n log n),
 *  where n is the length of the subarray to be sorted.
 *
 *  @param x - the array whose subarray is to be sorted.
 *  @param off - the start index in x of the subarray to be sorted.
 *  @param len - the length of the subarray to be sorted.
 *
 */
private static void sort1(int x[ ], int off, int len)
{
      // Choose a pivot element, v
      int m = off + (len >> 1),
          l = off,
          n = off + len - 1;

      m = med3 (x, l, m, n); // median of 3
      int v = x [m];   // v is the pivot

      int b = off,
          c = off + len - 1;
      while(true)
      {
              while (b <= c && x [b] < v)
                   b++;
              while (c >= b && x [c] > v)
                   c--;
```

```
                if (b > c)
                     break;
                swap (x, b++, c--);
        } // while true

        if (c + 1 -off > 1)
                sort1 (x, off, c + 1 -off);
        if (off + len -b > 1)
                sort1 (x, b, off + len -b);
} // method sort1


/**
 *  Finds the median of three specified elements in a given array..
 *
 *  @param x - the given array.
 *  @param a - the index of the first element.
 *  @param b - the index of the second element.
 *  @param c - the index of the third element
 *
 *  @return the median of x [a], x [b], and x [c].
 *
 */
private static int med3(int x[], int a, int b, int c) {
     return (x[a] < x[b] ?
             (x[b] < x[c] ? b : x[a] < x[c] ? c : a) :
             (x[b] > x[c] ? b : x[a] > x[c] ? c : a));
} // method med3


/**
 *  Swaps two specified elements in a specified array.
 *
 *  @param x – the array in which the two elements are to be swapped.
 *  @param a – the index of one of the elements to be swapped.
 *  @param b – the index of the other element to be swapped.
 *
 */
private static void swap(int x[], int a, int b) {
     int t = x[a];
     x[a] = x[b];
     x[b] = t;
} // method swap
```

Analysis We can view the effect of `sort1` as creating an imaginary binary search tree, whose root element is the pivot and whose left and right subtrees are the left and right subarrays. For example, suppose we call Quick Sort for the following array of 15 integers

68 63 59 77 98 87 84 51 17 12 8 25 42 35 31

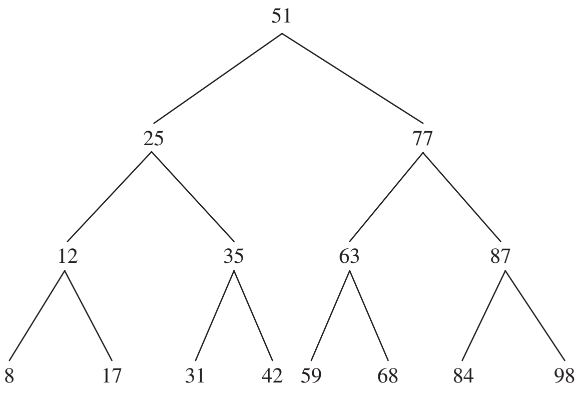


FIGURE 11.9 The imaginary binary search tree created by repeated partitions, into equal sized subarrays, of the array [68 63 59 77 98 87 84 51 17 12 8 25 42 35 31]

The first pivot chosen is 51; after the first partitioning, the pivot of the left subarray is 25 and the pivot of the right subarray is 77. Figure 11.9 shows the full binary-search-tree induced during the sorting of the given array. In general, we get a full binary search tree when each partition splits its subarray into two subarrays that have the same size. We would also get such a tree if, for example, the elements were originally in order or in reverse order, because then the pivot would always be the element at index `off + len/2`, and that element would always be the actual median of the whole sequence.

Contrast the above tree with the tree shown in Figure 11.10. The tree in Figure 11.10 represents the partitioning generated, for example, by the following sequence of 38 elements (the pivot is the median of 1, 37, and 36):

1, 2, 3, . . ., 17, 18, 0, 37, 19, 20, 21, . . , 35, 36

The worst case will occur when, during each partition, the pivot is either the next-to-smallest or next-to-largest element. That is what happens for the sequence that generated the tree in Figure 11.10.

For any array to be sorted, the induced binary search tree can help to determine how many comparisons are made in sorting the tree. At level 0, each element is compared to the original pivot, for a total of approximately n loop iterations (there will be an extra iteration just before the counters cross). At level 1, there are two subarrays, and each element in each subarray is compared to its pivot, for a total of about n iterations. In general, there will be about n iterations at each level, and so the total number of iterations will be, approximately, n times the number of levels in the tree.

We can now estimate the average time for the method `sort1`. The average is taken over all $n!$ initial arrangements of elements in the array. At each level in the binary search tree that represents the partitioning, about n iterations are required. The number of levels is the average height of that binary search tree. Since the average height of a binary search tree is logarithmic in n, we conclude that the total number of iterations is linear-logarithmic in n. That is, averageTime(n) is linear-logarithmic in n. By Sorting Fact 2 in Section 11.3.1, the averageTime(n) for Quick Sort is optimal.

In the worst case, the first partition requires about n iterations and produces a subarray of size $n - 2$ (and another subarray of size 1). When this subarray of size $n - 2$ is partitioned, about $n - 2$ iterations are required, and a subarray of size $n - 4$ is produced. This process continues until the last subarray, of size 2 is partitioned. The total number of iterations is approximately $n + (n - 2) + (n - 4) + \cdots + 4 + 2$, which is, approximately, $n^2/4$. We conclude that worstTime(n) is quadratic in n.

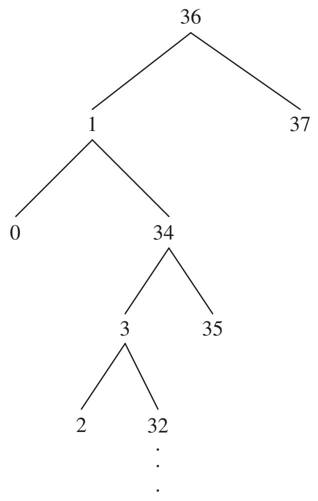


FIGURE 11.10 Worst-case partitioning: each partition reduces by only 2 the size of the subarray to be Quick Sorted. The corresponding binary search tree has a leaf at every non-root level. The subtrees below 32 are not shown

Quick Sort's space needs are due to the recursive calls, so the space estimates depend on the longest chain of recursive calls, because that determines the maximum number of activation records in the run-time stack. In turn, the longest chain of recursive calls corresponds to the height of the induced binary search tree. In the average case, that height is logarithmic in n, and we conclude that averageSpace(n) is logarithmic in n. In the worst case, that height is linear in n, so worstSpace(n) is linear in n.

Quick Sort is not a stable sort. For example, in the example given at the beginning of this section, there are two copies of 80. The one at index 3 is swapped into index 16, and the one at index 13 remains where it starts.

Quick Sort is another example of the Divide-and-Conquer design pattern. Each of the recursive calls to `sort1` can be done in parallel, and the combined effect of those calls is a sorted array.

11.4.3.1 Optimizations to the Quick Sort Algorithm

The `Arrays` class's `sort1` method has several modifications to the definition given in Section 11.4.3. The modifications deal with handling small subarrays, handling large subarrays, and excluding the pivot (and elements equal to the pivot) from either subarray.

The partitioning and Quick Sorting continues only for subarrays whose size is at least 7. For subarrays of size less than 7, Insertion Sort is applied. This avoids using the partitioning and recursive-call machinery for a task that can be handled efficiently by Insertion Sort. The choice of 7 for the pivot is based on empirical tests described in Bentley [1993]. For arrays of size 7, the pivot is chosen as the element at the middle index, that is, `pivot =  x [off + (len >> 1)]`. (Recall that `len >> 1` is a fast way to calculate `len/2`.) For arrays of size 8 through 40, the pivot is chosen—as we did in Section 11.4.3—as the median of the three elements at the first, middle, and last indexes of the subarray to be partitioned.

For subarrays of size greater than 40, an extra step is made to increase the likelihood that the pivot will partition the array into subarrays of about the same size. The region to be sorted is divided into three parts, the median-of-three technique is applied to each part, and then the median of those three medians becomes the pivot. For example, if `off` = 0 and `len` = 81, Figure 11.11 shows how the pivot would

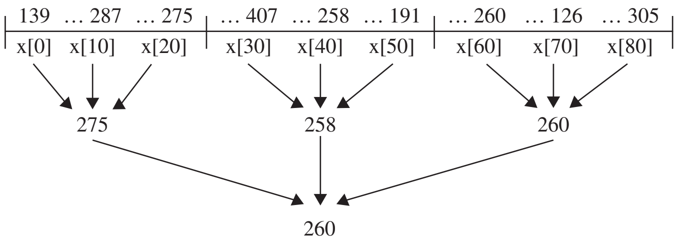


FIGURE 11.11 The calculation of the pivot as the median of three medians. The median of (139, 287, 275) is 275; the median of (407, 258, 191) is 258; the median of (260, 126, 305) is 260. The median of (275, 258, 260) is 260, and that is chosen as the pivot

be calculated for a sample arrangement of the array `x`. These extra comparisons have a price, but they increase the likelihood of an even split during partitioning.

There are two additional refinements, both related to the pivot. Instead of incrementing `b` until an element greater than or equal to the pivot is found, the search is given by

```
while (b <= c && x[b] <= v)
```

A similar modification is made in the second inner loop. This appears to be an optimization because the pivot won't be swapped, but the inner-loop conditions also test `b <= c`. That extra test may impede the speed of the loop more than avoiding needless swaps would enhance speed.

That refinement enables another pivot-related refinement. For the `sort1` method defined above, the pivot may end up in one of the subarrays and be included in subsequent comparisons. These comparisons can be avoided if, after partitioning, the pivot is always stored where it belongs. Then the left subarray will consist of elements strictly less than the pivot, and the right subarray will consist of elements strictly greater than the pivot. Then the pivot—indeed, all elements equal to the pivot—will be ignored in the rest of the Quick Sorting.

To show you how this can be accomplished, after the execution of the outer loop the relation of segments of the subarray to the pivot `v` will be as shown in Figure 11.12.

At this point, equal-to-pivot elements are swapped back into the middle of the subarray. The left and right subarrays in the recursive calls do not include the equal-to-pivot elements.

As the partitioning of a subarray proceeds, the equal-to-pivot elements are moved to the beginning and end of the subarray with the help of a couple of additional variables:

```
int a = off,

d = off + len – 1;
```

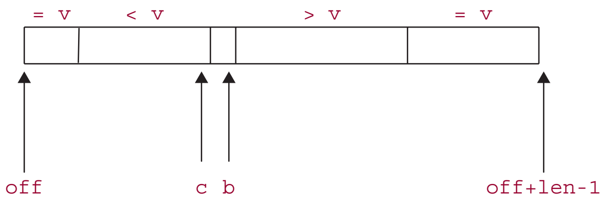


FIGURE 11.12 The relationship of the pivot `v` to the elements in the subarray to be partitioned. The leftmost segment and rightmost segment consist of copies of the pivot

The index `a` will be one more than highest index of an equal-to-pivot element in the left subarray, and `d` will be one less than the lowest index of an equal-to-pivot element in the right subarray. In the first inner loop, if `x [b] = v`, we call

```
swap (x, a++, b);
```

Similarly, in the second inner loop, if `x [c] = v`, we call

```
swap (x, c, d–);
```

Figure 11.13 indicates where indexes `a` and `d` would occur in Figure 11.12.

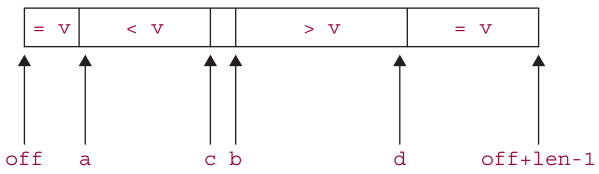


FIGURE 11.13 A refinement of Figure 11.12 to include indexes `a` and `d`

For an example that has several copies of the pivot, suppose we started with the following array of 20 **int**s:

59 46 59 80 46 55 87 43 44 81 95 12 17 80 75 33 40 59 16 50

During partitioning, copies of 59 are moved to the leftmost and rightmost part of the array. After b and c have crossed, we have the arrangement shown in Figure 11.14.

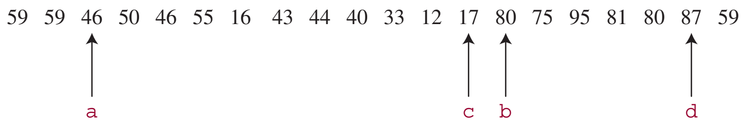


FIGURE 11.14 The status of the indexes a, b, c, and d after partitioning

Now all the duplicates of 59 are swapped into the middle, as shown in Figure 11.15.

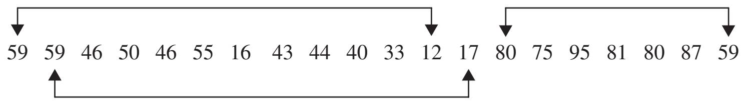


FIGURE 11.15 The swapping of equal-to-pivot elements to the middle of the array

We now have the array shown in Figure 11.16:

12 17 46 50 46 55 16 43 44 40 33 59 59 59 75 95 81 80 87 80

FIGURE 11.16 The array from Figure 11.15 after the swapping

The next pair of recursive calls is:

```
sort1 (x, 0, 11);
sort1 (x, 14, 6);
```

The duplicates of 59, at indexes 11, 12, and 13, are in their final resting place.

Here, from the `Arrays` class, is the complete definition (the `swap` and `med3` method definitions were given in Section 11.4.3):

```
/**
 *  Sorts into ascending order the subarray of a specified array, given
 *  an initial index and subarray length.
 *  The worstTime(n) is O(n * n) and averageTime(n) is O(n log n),
 *  where n is the length of the subarray to be sorted.
 *
 *  @param x - the array whose subarray is to be sorted.
 *  @param off - the start index in x of the subarray to be sorted.
 *  @param len - the length of the subarray to be sorted.
 *
 */
private static void sort1(int x[ ], int off, int len)
{

      // Insertion sort on smallest arrays
      if (len < 7) {
         for (int i=off; i<len+off; i++)
            for (int j=i; j>off && x[j-1]>x[j]; j--)
                swap(x, j, j-1);
         return;
      }

      // Choose a partition element, v
      int m = off + (len >> 1);       // Small arrays, middle element
      if (len > 7) {
            int l = off;
            int n = off + len - 1;
            if (len > 40) {         // Big arrays, pseudomedian of 9
               int s = len/8;
               l = med3(x, l,     l+s, l+2*s);
               m = med3(x, m-s,   m,   m+s);
               n = med3(x, n-2*s, n-s, n);
        }
        m = med3(x, l, m, n); // Mid-size, med of 3
      }
      int v = x[m];    // v is the pivot

      // Establish Invariant: = v; < v; > v; = v
      int a = off, b = a, c = off + len - 1, d = c;
      while(true) {
         while (b <= c && x[b] <= v) {
              if (x[b] == v)
```

```
                    swap(x, a++, b);
                b++;
            }
            while (c >= b && x[c] >= v) {
                if (x[c] == v)
                    swap(x, c, d--);
                c--;
            }
            if (b > c)
                break;
            swap(x, b++, c--);
        }

        // Swap partition elements back to middle
        int s, n = off + len;
        s = Math.min(a-off, b-a  );  vecswap(x, off, b-s, s);
        s = Math.min(d-c,   n-d-1);  vecswap(x, b,   n-s, s);

        // Recursively sort non-partition-elements
        if ((s = b-a) > 1)
            sort1(x, off, s);
        if ((s = d-c) > 1)
            sort1(x, n-s, s);
    }


    /**
     *  Swaps the elements in two specified subarrays of a given array.
     *  The worstTime(n) is O(n), where n is the number of pairs to be swapped.
     *
     *  @param x - the array whose subarrays are to be swapped.
     *  @param a - the start index of the first subarray to be swapped.
     *  @param b - the start index of the second subarray to be swapped.
     *  @param n - the number of elements to be swapped from each subarray.
     *
     */
    private static void vecswap(int x[], int a, int b, int n) {
         for (int i=0; i<n; i++, a++, b++)
              swap(x, a, b);
    } // method vecswap
```

With these optimizations, the results of the analysis in Section 11.4.3 still hold. For example, we now show that if Quick Sort is applied to a large array, worstTime(n) will still be quadratic in n. For $n > 40$, the worst case occurs when the 9 elements involved in the calculation of the median of medians are the five smallest and four largest elements. Then the fifth-smallest element is the best pivot possible, and the partitioning will reduce the size of the subarray by 5. In partitioning the subarray of size $n - 5$, we may have the four smallest and five largest tested for median of medians, and the size will again be reduced by only 5. Since the number of iterations at each level is, approximately, the size of the subarray to be partitioned, the total number of iterations is, approximately,

$$n + (n - 5) + (n - 10) + (n - 15) + \cdots + 45$$

which is, approximately, $n^2/10$. That is, worstTime(n) is quadratic in n.

For a discussion of how, for any positive integer n, to create an array of **`int`** values for which Quick Sort takes quadratic time, see McIlroy [1999].

One debatable issue with the above Quick Sort algorithm is its approach to duplicates of the chosen pivot. The overall algorithm is significantly slowed by the test for equality in the inner loops of `sort1`. Whether this approach enhances or diminishes efficiency depends on the number of multiple pivot copies in the array.

We finish up this chapter with a sort method that is not comparison-based, and therefore essentially different from the other sort methods we have seen.

11.5 Radix Sort

Radix Sort is unlike the other sorts presented in this chapter. The sorting is based on the internal representation of the elements to be sorted, not on comparisons between elements. For this reason, the restriction that worstTime(n) can be no better than linear-logarithmic in n no longer applies.

Radix Sort was widely used on electromechanical punched-card sorters that appear in old FBI movies. The interested reader may consult Shaffer [1998].

For the sake of simplicity, suppose we want to sort an array of non-negative integers of at most two decimal digits. The representation is in base 10, also referred to as radix 10—this is how Radix Sort gets its name. In addition to the array to be sorted, we also have an array, `lists`, of 10 linked lists, with one linked list for each of the ten possible digit values.

During the first outer-loop iteration, each element in the array is appended to the linked list corresponding to the units digit (the least-significant digit) of that element. Then, starting at the beginning of each list, the elements in `lists [0]`, `lists [1]`, and so on are stored back in the original array. This overwrites the original array. In the second outer-loop iteration, each element in the array is appended to the linked list corresponding to the element's tens digit. Then, starting at the beginning of each list, the elements in `lists [0]`, `lists [1]`, and so on are stored back in the original array.

Here is the method specification:

```
/**
 *  Sorts a specified array into ascending order.
 *  The worstTime(n) is O(n log N), where n is the length of the array, and N
 *  is the largest number (in absolute value) of the numbers in the array.
 *
 *  @param a – the array to be sorted.
 *
 *  @throws NullPointerException - if a is null.
 *
 */
public static void radixSort (int[ ] a)
```

Example Suppose we start with the following array of 12 **`int`** values:

85 3 19 43 20 55 42 91 21 85 73 29

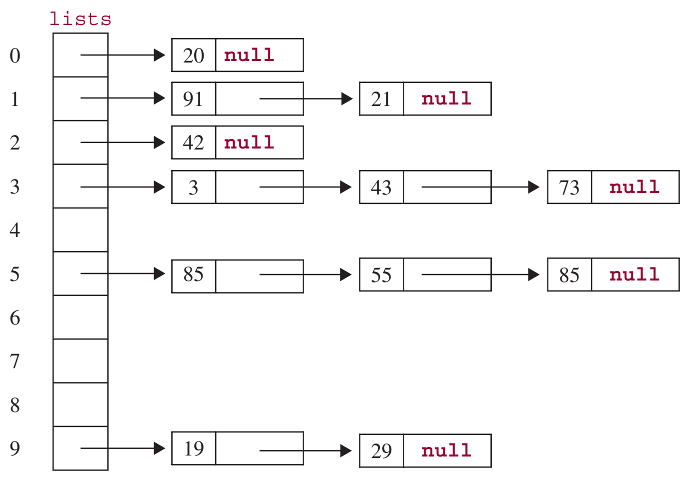


FIGURE 11.17 An array of linked lists after each element in the original array is appended to the linked list that corresponds to the element's units digit

After each of these is appended to the linked list corresponding to its units (that is, rightmost) digit, the array of linked lists will be as shown in Figure 11.17.

Then, starting at the beginning of each list, elements in `lists [0]`, `lists [1]`, and so on are stored back in `a`. See Figure 11.18.

20 91 21 42 3 43 73 85 55 85 19 29

FIGURE 11.18 The contents of the array `a`, with the elements ordered by their units digits

The elements in the array `a` have now been ordered by their units digits.

In the next outer-loop iteration, each element in `a` is appended to the list corresponding to the element's tens digit, as shown in Figure 11.19.

Finally (because the integers had at most two digits), starting at the beginning of each list, the integers in `lists [0]`, `lists [1]`, and so on are stored back in `a`:

3 19 20 21 29 42 43 55 73 85 85 91

The elements in `a` have been ordered by their tens digits, and for numbers with the same tens digits, they have been ordered by their units digits. In other words, the array `a` is sorted.

What happens in general? Suppose we have two integers x and y, with $x < y$. Here's how Radix Sort ensures that x ends up at a smaller index in the array. If x has fewer digits than y, then in the final iteration of the outer loop, x will be placed in `lists [0]` and y will be placed in a higher-indexed list because y's leftmost digit is not zero. Then when the lists are stored back in the array, x will be at a smaller index than y.

If x and y have the same number of digits, start at the leftmost digit in each number and, moving to the right, find the first digit in x that is smaller than the corresponding digit in y. (For example, if $x = 28734426$ and $y = 28736843$, the thousands digit in x is less than the thousands digit in y.) Then at the start of that iteration of the outer loop, x will be placed in a lower indexed list than y. Then when the

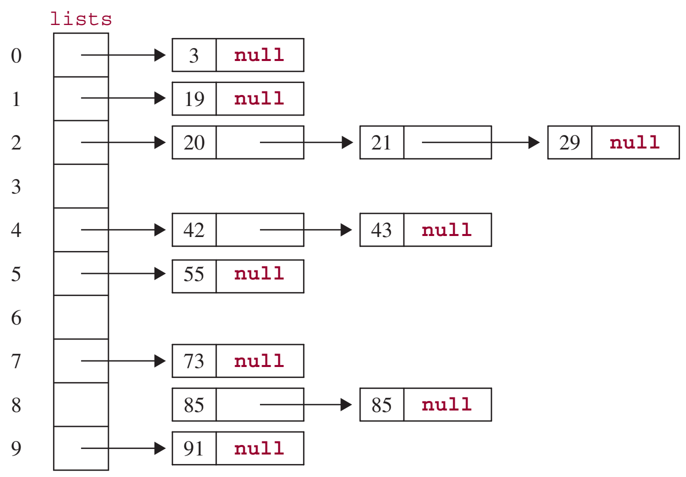


FIGURE 11.19 The array of linked lists after each element in the array `a` has been appended to the linked list corresponding to its tens digit

lists are stored back in the array, *x* will be at a smaller index than *y*. And from that point on, the relative positions of *x* and *y* in the array will not change because they agree in all remaining digits.

There is a slight difficulty in converting the outline in the previous paragraphs into a Java method definition. The following statement is illegal:

```
LinkedList<Integer>[ ] lists = new LinkedList<Integer> [10];
```

The reason is that arrays use ***covariant subtyping*** (for example, the array `Double[ ]` is a subtype of `Object[ ]`), but parameterized types use ***invariant subtyping*** (for example, `LinkedList<Double>` is *not* a subtype of `LinkedList<Object>`). We cannot create an array whose elements are parameterized collections. But it is okay to create an array whose elements are raw (that is, unparameterized) collections. So we will create the array with the raw type `LinkedList`, and then construct the individual linked lists with a parameterized type.

Here is the method definition:

```
/**
 *  Sorts a specified array into ascending order.
 *  The worstTime(n) is O(n log N), where n is the length of the array, and N is the largest
 *  number (in absolute value) of the numbers in the array.
 *
 *  @param a - the array to be sorted.
 *
 */
public static void radixSort (int [ ] a)
{
     final int RADIX = 10;

     int biggest = a [0],
          i;
```

```
      for (i = 1; i < a.length; i++)
          if (a [i] > biggest)
              biggest = a [i];

      int maxDigits = (int)Math.floor (Math.log (biggest) / Math.log (10)) + 1;

      long quotient = 1;        // the type is long because the largest number may have
                                // 10 digits; the successive quotients are 1, 10, 100, 1000,
                                // and so on. 10 to the 10th is too large for an int value.

     LinkedList[ ] lists = new LinkedList [RADIX];

     for (int m = 0; m < RADIX; m++)
          lists [m] = new LinkedList<Integer>();

    // Loop once for each digit in the largest number:
    for (int k = 0; k < maxDigits; k++)
    {
       // Store each int in a as an Integer in lists at the index of a [i]'s kth-smallest digit:
       for (i = 0; i < a.length; i++)
            ((LinkedList<Integer>)lists [(int)(a [i] / quotient) % RADIX]).add (a [i]);
       i = 0;

       // Store each Integer in list [0], list [1], ...,  as an int in a:
       for (int j = 0; j < RADIX; j++)
       {
            for (Integer anInt :  (LinkedList<Integer>)lists [j])
                  a [i++] = anInt;   // unboxing
            lists [j].clear();
       } // for j
       quotient *= RADIX;
    } // for k
} // method radixSort
```

Analysis Suppose N is the largest integer in the array. The number of outer-loop iterations must be at least ceil ($\log_{10} N$), so worstTime(n, N) is O($n \log N$). If the array also includes negative integers, N is chosen as the largest number in absolute value. Each array element is also stored in a linked list, and so worstSpace(n) is linear in n.

The elements are stored first-in, first-out in each list, and that makes Radix Sort stable.

Note: The elements in this example of Radix Sort are of type `int`, but with a slight change, the element type could also be `String`, for example. There would be one list for each possible character in the `String` class. Because each Unicode character occupies 16-bits, the number of distinct characters is $2^{16} = 65,536$ characters. That would require 65,536 linked lists! Instead, the allowable character set would probably be reduced to ASCII, an 8-bit code, so there would be only $2^8 = 256$ characters, and therefore 256 lists.

Lab 18 includes Radix Sort in a run-time experiment on sort methods.

You are now prepared to do Lab 18: Run-times for Sort Methods.

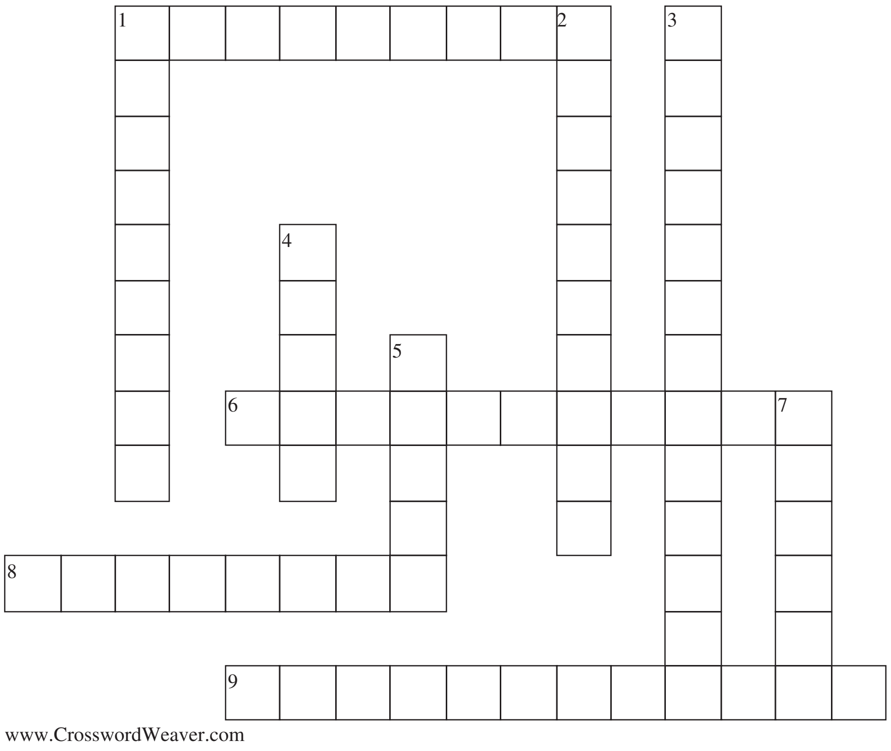
SUMMARY

Table 11.1 provides a thumbnail sketch of the sort algorithms presented in this chapter.

Table 11.1 Important features of sort algorithms from Chapter 11. Run-time rank is based on the time to sort n randomly-generated integers. The restrictions on element type are for the versions of Merge Sort and Quick Sort in the Java Collections Framework (JCF). For Radix Sort, N refers to the largest number in the collection

Sort Algorithm	Element Type Restriction	Stable?	worstTime(n)	averageTime(n);	run-time rank	worstSpace(n)
Insertion Sort		yes	quadratic	quadratic;	**4**	Constant
Selection Sort		no	quadratic	quadratic;	**5**	Constant
Bubble Sort		yes	quadratic	quadratic;	**6**	Constant
Merge Sort	reference (in JCF)	yes	linear- logarithmic	linear- logarithmic;	**2**	Linear
Quick Sort	primitive (in JCF)	no	quadratic	linear-logarithmic;	**1**	Linear
Radix Sort		yes	n log N	n log N;	**3**	Linear

CROSSWORD PUZZLE



ACROSS

1. The worstTime(*n*) for the three simple sorts is ______ in *n*.

6. The class in the Java Collections Framework that has exactly two versions of the Merge Sort algorithm

8. The number of different versions of the Quick Sort algorithm in the `Arrays` class

9. Given *n* elements to be sorted, a ______ is a binary tree in which each non-leaf represents a comparison between two elements and each leaf represents a sorted sequence of the *n* elements.

DOWN

1. The sorting algorithm whose average run-time performance is fastest

2. An interface whose implementation allows "unnatural" comparisons of elements

3. The only one of the three simple sorts that is not stable

4. In Quck Sort partitioning, the element that every element in a subarray is compared to

5. For comparison-based sorts, averageTime(*n*) is BigOmega (__________).

7. A __________ sort method preserves the relative order of equal elements.

CONCEPT EXERCISES

11.1 Trace the execution of each of the six sort methods—Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort, and Radix Sort—with the following array of values:

10 90 45 82 71 96 82 50 33 43 67

11.2 **a.** For each sort method, rearrange the list of values in Concept Exercise 11.1 so that the minimum number of element-comparisons would be required to sort the array.

b. For each sort method, rearrange the list of values in Concept Exercise 11.1 so that the maximum number of element-comparisons would be required to sort the sequence.

11.3 Suppose you want a sort method whose worstTime(n) is linear-logarithmic in n, but requires only linear-in-n time for an already sorted collection. None of the sorts in this chapter have those properties. Create a sort method that does have those properties.

Hint: Add a front end to Merge Sort to see if the collection is already sorted.

11.4 For the optimized Quick Sort in Section 11.4.3.1, find an arrangement of the integers 0 ... 49 for which the first partition will produce a subarray of size 4 and a subarray of size 44. Recall that because the number of values is greater than 40, the pivot is the "super-median," that is, the median of the three median-of-threes.

11.5 **a.** Suppose we have a sort algorithm whose averageTime(n) is linear-logarithmic in n. For example, either Merge Sort or Quick Sort would qualify as such an algorithm. Let runTime(n) represent the time, in seconds, for the implementation of the algorithm to sort n random integers. Then we can write:

$$\text{runTime}(n) \approx k(c) * n * \log_c n \text{ seconds},$$

where c is a an integer variable and k is a function whose value depends on c. Show that runTime(cn) $\approx$ runTime(n) $*(c + c/\log_c n)$.

b. Use the technique in Concept Exercise 11.5.a to estimate runTime(200000) if runTime(100000) = 10.0 seconds.

11.6 Show that seven comparisons are sufficient to sort any collection of five elements.

Hint: Compare the first and second elements. Compare the third and fourth elements. Compare the two larger elements from the earlier comparisons. With three comparisons, we have an ordered chain of three elements, with the fourth element less than (or equal to) one of the elements in the chain. Now compare the fifth element to the middle element in the chain. Complete the sorting in three more comparisons. Note that ceil($\log_2 5!$) = 7, so some collections of five elements cannot be sorted with 6 comparisons.

11.7 Show that $\log_2 n! >= n/2 \log_2 (n/2)$ for any positive integer n.

Hint: For any positive integer n,

$$n! = \prod_{i=1}^{n} i >= \prod_{i=1}^{n/2} (n/2) = (n/2)^{n/2}$$

11.8 Show how Quick Sort's partitioning can be used to develop a method, `median`, that finds the median of an array of **`int`** values. For the method `median`, averageTime(n) must be linear in n.

Hint: Suppose we want to find the median of x [0 ... 10000]. Of course, if we Quick Sort the array, the median would be in x [5000], but then averageTime(n) would be linear-logarithmic in n. To get an idea of how to proceed, let's say that the first partition yields a left subarray x [0 ... 3039] and a right subarray x [3055 ... 10000], with copies of the pivot in x [3040 ... 3054]. Since every **`int`** value in the left subarray is less than every **int** value in the right subarray, which subarray *must* contain the median? The other subarray can be ignored from then on, so the array is not completely sorted.

11.9 Consider the following, consecutive improvements to Insertion Sort:

a. Replace the call to the method swap with in-line code:

```
public static void insertionSort (int[ ] x)
{
    int temp;

    for (int i = 1; i < x.length; i++)
        for (int k= i; k > 0 && x [k -1] > x [k]; j--)
        {
                temp = x [k];
                x [k] = x [k -1];
                x [k -1] = temp;
        } // inner for
} // method insertionSort
```

b. Notice that in the inner loop in part a, temp is repeatedly assigned the original value of x [i]. For example, suppose the array x has

32 46 59 80 35

and j starts at 4. Then 35 hops its way down the array, from index 4 to index 1. The only relevant assignment from temp is that last one. Instead, we can move the assignments to and from temp out of the inner loop:

```
int temp,
      k;

for (int i = 1; i < x.length; i++)
{
     temp = x [i];
     for (k = i; k > 0 && x [k -1] > temp; k--)
         x [k] = x [k -1];
         x [k] = temp;
} // outer for
```

Will these changes affect the estimates for worstTime(n) and averageTime(n)?

11.10 If x is an array, Arrays.sort (x) can be called. Will x be Merge Sorted or Quick Sorted? How is the determination made?

11.11 Show how Merge Sort can be used to sort an array of primitives with the help of the wrapper classes.

11.12 The Java Collection Framework's version of Quick Sort can be applied only to an array of a primitive type, such as **int** or **double**. Exactly what would have to be changed to create a Quick Sort method that could be applied to an array of objects?

11.13 If Merge Sort is applied to a collection with 25 elements, what are the values of the index arguments for the first two recursive calls?

11.14 Give an example to show that Selection Sort is not a stable sort.

Hint: you need only three elements.

PROGRAMMING EXERCISES

11.1 For Concept Exercise 11.9, conduct a timing experiment to estimate the run-time effect of the changes made.

11.2 In the Java Collections Framework version of Quick Sort, special care is taken during partitioning to make sure that the pivot, and elements equal to the pivot, are not in either of the subarrays created. Estimate—in percentage terms—how much faster Quick Sort would run, on average, if this special care were not taken. Conduct a timing experiment to test your hypothesis.

11.3 In the `med3` method, replace the two applications of the conditional operator with **`if`** statements.

11.4 For the original version of Quick Sort in Section 11.4.3, replace the inner-loop conditions from

`while (x [b] < v)` and `while (x [c] > v)`

to

`while (b <= c && x [b] <= v)` and `while (c >= b && x [c] >= v)`

Create a small program method to apply this version of Quick Sort to the following array of **`int`** values:

46 59

Explain the results.

11.5 Develop a version of Radix Sort to sort an array of `String` objects. You may assume that each `String` object contains only ASCII characters, and that the maximum size of any `String` object is 30. Use JUnit to test your `radixSort` method.

Hint: Instead of the `quotient` variable, use the `charAt` method in the `String` class.

11.6 Modify the `radixSort` method in Section 11.5 to use an `ArrayList` instead of an array.

Hint: Start with

```
ArrayList<LinkedList<Integer>> lists = new ArrayList<LinkedList<Integer>>(RADIX);
```

Then append 10 empty linked lists to `lists`.

Programming Project 11.1

Sorting a File into Ascending Order

Analysis The input line will contain the path to the file to be sorted. Each element in the file will consist of a name—last name followed by a blank followed by first name followed by a blank followed by middle name—and social security number. The file is to be sorted by name; equal names should be ordered by social security number. For example, after sorting, part of the file might be as follows:

```
Jones Jennifer Mary  222222222
Jones Jennifer Mary  644644644
```

For convenience, you may assume that each name will have a middle name.

Suppose the file persons.dat consists of the following:

```
Kiriyeva Marina Alice      333333333
Johnson Kevin Michael      555555555
Misino John Michael        444444444
```

(continued on next page)

(continued from previous page)

```
Panchenko Eric Sam         888888888
Taoubina Xenia Barbara     111111111
Johnson Kevin Michael      222222222
Deusenbery Amanda May      777777777
Dunn Michael Holmes        999999999
Reiley Timothy Patrick     666666666
```

System Test 1:

Please enter the path for the file to be sorted.

```
persons.dat
The file persons.dat has been sorted.
The file persons.dat will now consist of

Deusenbery Amanda May      777777777
Dunn Michael Holmes        999999999
Johnson Kevin Michael      222222222
Johnson Kevin Michael      555555555
Kiriyeva Marina Alice      333333333
Misino John Michael        444444444
Panchenko Eric Sam         888888888
Reiley Timothy Patrick     666666666
Taoubina Xenia Barbara     111111111
```

For a larger system test, randomly generated, use the same name for each person. The social security numbers will be randomly generated **int**s in the range 0 . . . 999999999. For example, part of the file might have

```
a a a   238749736
a a a   701338476
a a a   408955917
```

Use unit testing to increase your confidence in the correctness of your methods.

Hint: This would be a fairly simple problem if we could be certain that the entire file would fit in main memory. Unfortunately, this is not the case. Suppose we want to sort a large file of objects from the class `Person`. For specificity, we assume that an object in the `Person` class occupies 50 bytes and that the maximum storage for an array is 500,000 bytes. So the maximum size of an array of `Person` objects is 10,000.

We start by reading in the file of persons, in blocks of k persons each. Each block is Merge Sorted and stored, in an alternating fashion, on one of two temporary files: leftTop and leftBottom. Figure 11.20 illustrates the effect of this first stage in file sorting.

We then go through an alternating process which continues until all of the elements are sorted and in a single file. The temporary files used are leftTop, leftBottom, and rightBottom; personsFile itself plays the role of rightTop. At each stage, we merge a top and bottom pair of files, with the resulting double-sized blocks stored alternately on the other top and bottom pair. The code for merging sorted blocks in two files into sorted, double-sized blocks in another file is essentially what was done—using subarrays instead of file blocks—at the end of Merge Sort. Here is that code

```java
// Merge sorted subarrays in src into dest.
for (int i = low, p = low, q = mid; i < high; i++) {
    if (q>=high || (p<mid && ((Comparable)src[p]).compareTo (src[q])<= 0))
        dest [i] = src [p++];
    else
        dest[i] = src[q++];
}
```

Figure 11.21 illustrates the first merge pass.

If rightBottom is still empty after a left-to-right merge, then the sort is complete and personsFile holds the sorted file. Otherwise a right-to-left merge is performed, after which we check to see if leftBottom is still empty. If so, leftTop is copied onto personsFile and the sort is complete.

How much time will this take? Suppose that we have n elements in n/k blocks, each of size k. In the Merge Sort phase, creating each of the n/k sorted blocks takes, roughly, $k \log_2 k$ time, on average. Each Merge phase takes about n iterations, and there are about $\log_2(n/k)$ Merge phases. The total time is the sum of the times for all phases: roughly,

$$\begin{aligned}(n/k) * \mathrm{k} \log_2 k + n * \log_2(n/k) &= n \log_2 k + n \log_2(n/k) \\ &= n \log_2 k + n \log_2 n - n \log_2 k \\ &= n \log_2 n\end{aligned}$$

Because the averageTime(n) is optimal, namely linear-logarithmic in n, a sorting method such as this is often used for a system sort utility.

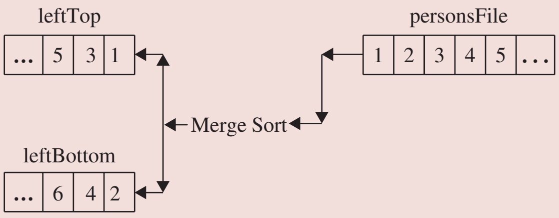


FIGURE 11.20 The first stage in file sorting: each of the unsorted blocks in personsFile is Merge Sorted and stored in leftTop or leftBottom

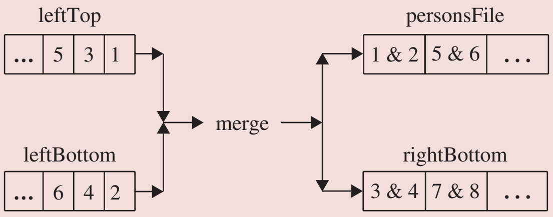


FIGURE 11.21 The first merge pass in file sorting. The files leftTop and leftBottom contain sorted blocks, and personsFile and rightBottom contain double-sized sorted blocks

Tree Maps and Tree Sets

CHAPTER 12

We begin this chapter by introducing another kind of balanced binary tree: the red-black tree. Red-black trees provide the underpinning for two extremely valuable classes: the `TreeMap` class and the `TreeSet` class, both of which are in the Java Collections Framework. Each element in a `TreeMap` object has two parts: a *key* part—by which the element is compared to other elements—and a *value* part consisting of the rest of the element. No two elements in a `TreeMap` object can have the same key. A `TreeSet` object is a `TreeMap` object in which all the elements have the same value part. There are applications of both the `TreeMap` class (a simple thesaurus) and the `TreeSet` class (a spell-checker). `TreeMap` objects and `TreeSet` objects are close to ideal: For inserting, removing and searching, worstTime(n) is logarithmic in n.

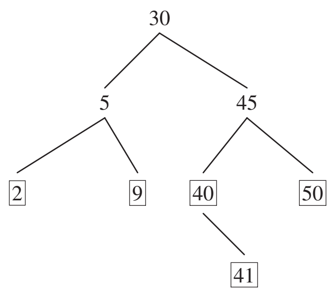
CHAPTER OBJECTIVES

1. Be able to define what a red-black tree is, and be able to distinguish between a red-black tree and an AVL tree.
2. Understand the `Map` interface and the overall idea of how the `TreeMap` implementation of the `Map` interface is based on red-black trees.
3. Compare `TreeMap` and `TreeSet` objects.

12.1 Red-Black Trees

Basically, a red-black tree is a binary search tree in which we adopt a coloring convention for each element in the tree. Specifically, with each element we associate a color of either red or black, according to rules we will give shortly. One of the rules involves paths. Recall, from Chapter 9, that if element A is an ancestor of element B, the ***path*** from A to B is the sequence of elements, starting with A and ending with B, in which each element in the sequence (except the last) is the parent of the next element. Specifically, we will be interested in paths from the root to elements with no children *or with one child*.[1] For example, in the following tree, there are five paths from the root to an element (boxed) with no children or one child.

[1]Equivalently, we could define the rule in terms of paths from the root element to an empty subtree, because an element with one child also has an empty subtree, and a leaf has two empty subtrees. When this approach is taken, the binary search tree is expanded to include a special kind of element, a stub leaf, for each such empty subtree.



Note that one of the paths is to the element 40, which has one child. So the paths described are not necessarily to a leaf.

A ***red-black tree*** is a binary search tree that is empty or in which *the root element is colored black*, every other element is colored red or black and the following properties are satisfied:

Red Rule: If an element is colored red, none of its children can be colored red.

Path Rule: The number of black elements must be the same in all paths from the root element to elements with no children **or with one child**.

For example, Figure 12.1 shows a red-black tree in which the elements are values of `Integer` objects and colored red or black

Observe that this is a binary search tree with a black root. Since no red element has any red children, the Red Rule is satisfied. Also, there are two black elements in each of the five paths (one path ends at 40) from the root to an element with no children or one child, so the Path Rule is satisfied. In other words, the tree is a red-black tree.

The tree in Figure 12.2 is *not* a red-black tree even though the Red Rule is satisfied and every path from the root to a leaf has the same number of black elements. The Path Rule is violated because, for example, the path from 70 to 40 (an element with one child) has three black elements, but the path from 70 to 110 has four black elements. That tree is badly unbalanced: most of its elements have only one child. The Red and Path rules preclude most single children in red-black trees. In fact, if a red element has any children, it must have two children and they must be black. And if a black element has only one child, that child must be a red leaf.

The red-black tree in Figure 12.1 is fairly evenly balanced, but not every red-black tree has that characteristic. For example, Figure 12.3 shows one that droops to the left.

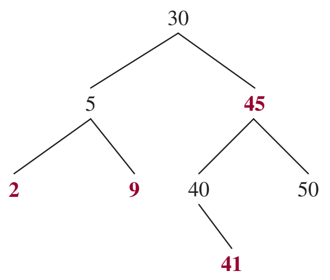


FIGURE 12.1 A red-black tree with eight elements

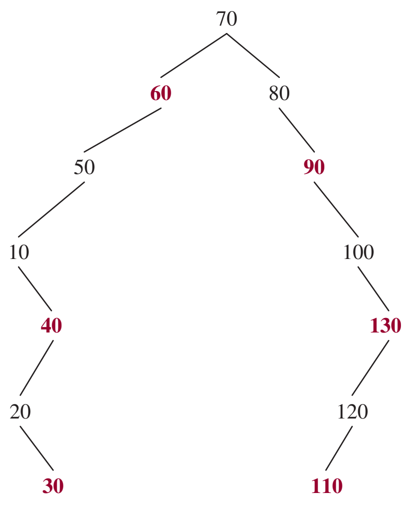


FIGURE 12.2 A binary search tree that is not a red-black tree

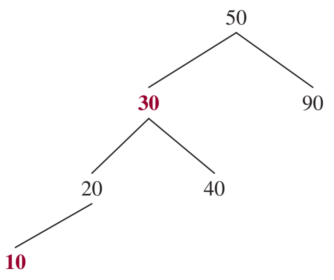


FIGURE 12.3 A red-black tree that is not "evenly" balanced

You can easily verify that this is a black-rooted binary search tree and that the Red Rule is satisfied. For the Path Rule, there are exactly two black elements in any path from the root to an element with no children or with one child. That is, the tree is a red-black tree. But there are limits to how unbalanced a red-black tree can be. For example, we could not hang another element under element 10 without re-balancing the tree. For if we tried to add a red element, the Red Rule would no longer be satisfied. And if we tried to add a black element, the Path Rule would fail.

If a red-black tree is complete, with all black elements except for red leaves at the lowest level, the height of that tree will be minimal, approximately $\log_2 n$. To get the maximum height for a given n, we would have as many red elements as possible on one path, and all other elements black. For example, Figure 12.3 contains one such tree, and Figure 12.4 contains another. The path with all of the red elements will be about twice as long as the path(s) with no red elements. These trees lead us to hypothesize that the maximum height of a red-black tree is less than $2 \log_2 n$.

12.1.1 The Height of a Red Black Tree

Red-black trees are fairly bushy in the sense that almost all non-leaves have two children. In fact, as noted earlier, if a parent has only one child, that parent must be black and the child must be a red leaf. This

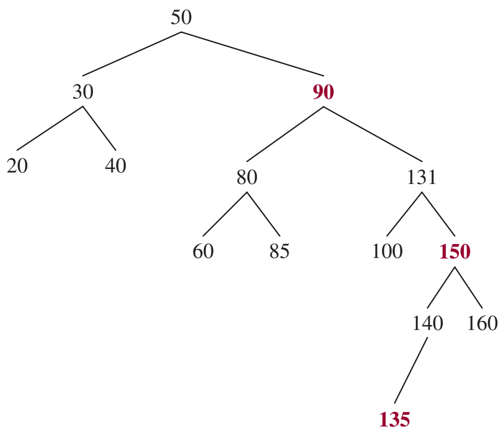


FIGURE 12.4 A red-black tree of 14 elements with maximum height, 5

bushiness leads us to believe that a red-black tree is balanced, that is, has height that is logarithmic in n, even in the worst case. Compare that with the worst-case height that is linear in n for a binary search tree. As shown in Example A2.6 of Appendix 2,

The height of a red-black tree is always logarithmic in *n*, the size of the tree.

How do red-black trees compare to AVL trees? The height of an AVL tree is also logarithmic in n. The definition of a red-black tree is slightly more "relaxed" than the definition of an AVL tree. So any AVL tree can be colored to become a red-black tree, but the converse is not true (see Concept Exercises 12.6 and 12.7). That is, red-black trees can have larger heights than AVL trees with the same number of elements. It can be shown (see Weiss [2002]) that the average height of an AVL tree with n elements is, approximately, $1.44 \log_2 n$, versus $2 \log_2 n$ for a red-black tree. For example, if n is one million, the average height of an AVL tree with n elements is about 29, and the average height of a red-black tree with n elements is about 40.

In Section 12.2, we introduce the `Map` interface, and in Section 12.3, a class that implements the `Map` interface. That class, the `TreeMap` class, is based on a red-black tree, and is part of the Java Collections Framework. The developers of the framework found that using a red-black tree for the underlying structure of the `TreeMap` class provided slightly faster insertions and removals than using an AVL tree.

12.2 The `Map` Interface

A ***map*** is a collection[2] in which each element has two parts: a unique ***key*** part and a ***value*** part. The idea behind this definition is that there is a "mapping" from each key to the corresponding value. For example, we could have a map of social security numbers and names. The keys will be social security numbers and the values will be names. The social security numbers are unique: no two elements in the collection are allowed to have the same social security number. But two elements may have the same name. For

[2]Recall, from Chapter 4, that a ***collection*** is an object that is composed of elements. A collection is not necessarily a `Collection` object, that is, a collection need not implement the `Collection` interface. For example, an array is a collection but not a `Collection` object.

example, we could have the following map, in which all of the social security numbers are unique, but two elements have the same name:

123-45-6789	Builder, Jay
222-22-2222	Johnson, Alan
555-55-5555	Nguyen, Viet
666-66-6666	Chandramouli, Soumya
888-88-8888	Kalsi, Navdeep
999-99-9999	Johnson, Alan

A dictionary is another example of a map. The key is the word being defined and the value consists of the definition, punctuation, and etymology. The term ***dictionary*** is sometimes used as a synonym for “map”. In this sense, a dictionary is simply a collection of key-value pairs in which there are no duplicate keys.

The Java Collections Framework has a `Map` interface that provides method headings for the abstract-data-type map. The `Map` interface does not extend the `Collection` interface because many `Map` methods are oriented towards the key-value relationship. In fact, the type parameters are `K` (for the key class) and `V` (for the value class). But the `Map` interface has some standard methods such as `size`, `equals`, and `clear`. Here are specifications for several of the other methods in the `Map` interface—no time estimates are given because different implementations have substantially different estimates:

1. **The `put` method**

```
/**
 * Associates the specified value with the specified key in this map.
 * If the map previously contained a mapping for the key, the old
 * value is replaced.
 *
 * @param key key with which the specified value is to be associated
 * @param value value to be associated with the specified key
 *
 * @return the previous value associated with key, or
 *         null if there was no mapping for key.
 *         (A null return can also indicate that the map
 *         previously associated null with key.)
 * @throws ClassCastException if the specified key cannot be compared
 *         with the keys currently in the map
 * @throws NullPointerException if the specified key is null
 *         and this map uses natural ordering, or its comparator
 *         does not permit null keys
 */
V put (K key, V value);
```

Note 1: The phrase “this `map` ” refers to an object in a class that implements the `Map` interface.
Note 2: The `put` method is somewhat more versatile than an `add` method because the `put` method handles replacement—of the values associated with a given key—as well as insertion of a new key-value pair.

2. **The `containsKey` method**

```
/**
 *  Determines if this Map object contains a mapping for a specified key.
 *
 *  @param key - the specified key.
 *
 *  @return true – if there is at least one mapping for the specified key in
 *          this Map object; otherwise, return false.
 *
 *  @throws ClassCastException – if key cannot be compared with the keys
 *          currently in the map.
 *
 *  @throws NullPointerException – if key is null and this Map object uses
 *          the natural order, or the comparator does not allow null keys.
 *
 */
boolean containsKey (Object key);
```

3. **The `containsValue` method**

```
/**
 *  Determines if there is at least one mapping with a specified value in this
 *  Map object.
 *
 *  @param value – the specified value for which a mapping is sought.
 *
 *  @return true – if there is at least one mapping with the specified value
 *          in this Map object; otherwise, return false.
 *
 */
boolean containsValue (Object value);
```

4. **The `get` method**

```
/**
 *  Returns the value to which a specified key is mapped in this Map
 *  object.
 *
 *  @param key - the specified key.
 *
 *  @return the value to which the specified key is mapped, if the specified
 *          key is mapped to a value; otherwise, return null.
 *
 *  @throws ClassCastException – if key cannot be compared with the keys
 *          currently in the map.
 *
 *  @throws NullPointerException – if key is null and this Map object uses
 *          the natural order, or the comparator does not allow null keys.
 */
V get (Object key);
```

Note: The value **`null`** might also be returned if the given key maps to **`null`** . To distinguish between this situation and the no-matching-key situation, the `containsKey` method can be used. For example, if `persons` is an object in a class that implements the `Map` interface and `key` is an object in the key class, we can do the following:

```
if (persons.get (key) == null)
   if (persons.containsKey (key))
       System.out.println (key + " maps to null");
   else
       System.out.println (key + " does not match any key in this map.");
```

5. **The `remove` method**

```
/**
 *  Removes the mapping with a specified key from this Map object, if there
 *  was such a mapping.
 *
 *  @param key - the specified key whose mapping, if present, is to be
 *         removed from this Map object.
 *
 *  @return the value to which the specified key is mapped, if there is such
 *          a mapping; otherwise, return null (note that null could also be the
 *          previous value associated with the specified key).
 *
 *  @throws ClassCastException - if key cannot be compared with the keys
 *          currently in the map.
 *
 *  @throws NullPointerException - if key is null and this Map object uses
 *          the natural order, or the comparator does not allow null keys.
 */
V remove (Object key);
```

6. **The `entrySet` method**

```
/**
 *  Returns a Set view of the key-map pairs in this Map object.
 *
 *  @return a Set view of the key-map pairs in this Map object.
 *
 */
Set entrySet();
```

Note: Recall, from Chapter 10, that a ***set*** is a collection of elements in which duplicates are not allowed. We can view a `Map` object as just a set of key-value pairs. The advantage to this view is that we can then iterate over the `Map` object, and the elements returned will be the key-value pairs of the `Map` object. Why is this important? The `Map` interface does not have an `iterator()` method, so you cannot iterate over a `Map` object except through a view. And the `Map` interface has a **`public`**, nested `Entry` interface that has `getKey()` and `getValue()` methods.

For example, suppose that `persons` is an instance of a class that implements the `Map` interface, and that the element class has a social security number as the (`Integer`) key and a name as the (`String`) value. Then we can print out the name of each person whose social security number begins with 555 as follows:

```
for (Map.Entry<Integer, String> entry : persons.entrySet())
      if (entry.getKey() / 1000000 == 555)
            System.out.println (entry.getValue());
```

There are also keySet() and values() methods that allow iterating over a Map viewed as a set of keys and as a collection of values, respectively. The term “collection of values” is appropriate instead of “set of values” because there may be duplicate values.

Section 12.3 has an implementation of the Map interface, namely, the TreeMap class. Chapter 14 has another implementation, the HashMap class. The TreeMap class, since it is based on a red-black tree, boasts logarithmic time, even in the worst case, for insertions, removals, and searches. The HashMap class’s claim to fame is that, on average, it takes only *constant* time for insertions, removals and searches. But its worst-case performance is poor: linear in n.

The TreeMap class actually implements a slight extension of the Map interface, namely, the SortedMap interface. The SortedMap interface mandates that for any instance of any implementing class, the elements will be in “ascending” order of keys (for example, when iterating over an entry-set view). The ordering is either the natural ordering—if the key class implements the Comparable interface—or an ordering supplied by a comparator. Here are several new methods:

```
/**
 *  Returns the comparator for this sorted map, or null, if the map uses the
 *  keys' natural ordering.  The comparator returned, if not null, must implement
 *  the Comparator interface for elements in any superclass of the key class.
 *
 *  @return the comparator for this sorted map, or null, if this sorted
 *              map uses the keys' natural ordering.
 *
 */
Comparator<? super K> comparator();


/**
 *  Returns the first (that is, smallest) key currently in this sorted map.
 *
 *  @return the first (that is, smallest) key currently in this sorted map.
 *
 *  @throws NoSuchElementException, if this sorted map is empty.
 *
 */
K firstKey();


/**
 *  Returns the last (that is, largest) key currently in this sorted map.
 *
 *  @return the last (that is, largest) key currently in this sorted map.
 *
 *  @throws NoSuchElementException, if this sorted map is empty.
 *
 */
K lastKey();
```

12.3 The TreeMap Implementation of the SortedMap Interface

The Java Collection Framework's TreeMap class implements the SortedMap interface. For the put, containsKey, get, and remove methods, worstTime(n) is logarithmic in n. Why? In a TreeMap object, the key-value pairs are stored in a red-black tree ordered by the keys. Can you figure out why, for the containsValue method, worstTime(n) is linear in n instead of logarithmic in n?

We will look at the fields and method definitions in Section 12.3.2. But our main emphasis is on the *use* of data structures, so let's start with a simple example. The following class creates a TreeMap object of students. Each student has a name and a grade point average; the ordering is alphabetical by student names. The method prints each student, each student whose grade point average is greater than 3.9, and the results of several removals and searches.

```java
import java.util.*;

public class TreeMapExample
{
       public static void main (String[ ] args)
       {
            new TreeMapExample().run();
       } // method main

       public void run()
       {
            TreeMap<String,Double> students = new TreeMap<String,Double>();

            students.put ("Bogan, James", 3.85);
            students.put ("Zawada, Matt", 3.95);
            students.put ("Balan, Tavi", 4.00);
            students.put ("Nikolic, Lazar", 3.85);

            System.out.println (students);

            for (Map.Entry<String, Double> entry : students.entrySet())
               if (entry.getValue() > 3.9)
                  System.out.println (entry.getKey() + " " + entry.getValue());

            System.out.println (students.remove ("Brown, Robert"));
            System.out.println (students.remove ("Zawada, Matt"));
            System.out.println (students.containsKey ("Tavi Balan"));
            System.out.println (students.containsKey ("Balan, Tavi"));
            System.out.println (students.containsValue (3.85));
       } // method run

} // class TreeMapExample
```

The output will be

```
{Balan, Tavi=4.0, Bogan, James=3.85, Nikolic, Lazar=3.85, Zawada, Matt=3.95}
Balan, Tavi 4.0
```

```
Zawada, Matt 3.95
null
3.95
false
true
true
```

The reason that the `students` object is alphabetically ordered by student names is that the key class is `String`. As we saw in Section 10.1.1, the `String` class implements the `Comparable` interface with a `compareTo` method that reflects an alphabetical ordering. For applications in which the "natural" ordering—through the `Comparable` interface—is inappropriate, elements can be compared with the `Comparator` interface, discussed in Section 11.3. In the `TreeMap` class, there is a special constructor:

```
/**
 *  Initializes this TreeMap object to be empty, with keys to be compared
 *  according to a specified Comparator object.
 *
 *  @param c - the Comparator object by which the keys in this TreeMap
 *         object are to be compared.
 *
 */
public TreeMap (Comparator<? super K> c)
```

We can implement the `Comparator` interface to override the natural ordering. For example, suppose we want to create a `TreeMap` of `Integer` keys (and `Double` values) in *decreasing* order. We cannot rely on the `Integer` class because that class implements the `Comparable` interface with a `compareTo` method that reflects increasing order. Instead, we create a class that implements the `Comparator` interface by reversing the meaning of the `compareTo` method in the `Integer` class:

```
public class Decreasing implements Comparator<Integer>
{
       /**
        *  Compares two specified Integer objects.
        *
        *  @param i1 - one of the Integer objects to be compared.
        *  @param i2 - the other Integer object.
        *
        *  @return the value of i2's int - the value of i1's int.
        *
        */
        public int compare (Integer i1, Integer i2)
        {
                 return i2.compareTo (i1);
        } // method compare

} // class Decreasing
```

Notice that the `Decreasing` class need not specify a type parameter since that class is implementing the `Comparator` interface parameterized with `Integer`.

A TreeMap object can then be constructed as follows:

```
TreeMap<Integer, Double> inventory =
            new TreeMap<Integer, Double>(new Decreasing());
```

For another example, here is the ByLength class from Section 11.3:

```
public class ByLength implements Comparator<String>
{
     /**
      *  Compares two specified String objects lexicographically if they have the
      *  same length, and otherwise returns the difference in their lengths.
      *
      *  @param s1 - one of the specified String objects.
      *  @param s2 - the other specified String object.
      *
      *  @return s1.compareTo (s2) if s1 and s2 have the same length;
      *          otherwise, return s1.length() - s2.length().
      *
      */
      public int compare (String s1, String s2)
      {
          int len1 = s1.length(),
              len2 = s2.length();
          if (len1 == len2)
                  return s1.compareTo (s2);
          return len1 - len2;
      } // method compare

} // class ByLength
```

The following class utilizes the ByLength class with a TreeMap object in which the keys are words—stored in order of increasing word lengths—and the values are the number of letters in the words.

```
import java.util.*;

public class TreeMapByLength
{
       public static void main (String[ ] args)
       {
               new TreeMapByLength().run();
       } // method main

       public void run()
       {
               TreeMap<String, Integer> wordLengths =
                            new TreeMap<String, Integer>(new ByLength());

               wordLengths.put ("serendipity", 11);
               wordLengths.put ("always", 6);
               wordLengths.put ("serenity", 8);
               wordLengths.put ("utopia", 6);
```

```
                System.out.println (wordLengths);
        } // method run

} // class TreeMapByLength
```

The output will be

```
{always=6, utopia=6, serenity=8, serendipity=11}
```

Now that we have seen a user's view of the TreeMap class, Sections 12.3.1 and 12.3.2 will spend a little time looking "under the hood" at the fields, the embedded Entry class and the method definitions. Then Section 12.4 will present an application of the TreeMap class: creating a thesaurus.

12.3.1 The TreeMap Class's Fields and Embedded Entry Class

In the design of a class, the critical decision is the choice of fields. For the TreeMap class, two of the fields are the same as in the BinarySearchTree class of Chapter 10 (except that the framework designers prefer **private** visibility to **protected** visibility):

```
private transient Entry root = null;

private transient int size = 0;
```

To flag illegal modifications (see Appendix 1) to the structure of the tree during an iteration:

```
private transient int modCount = 0;
```

The only other field in the TreeMap class is used for comparing elements:

```
private Comparator comparator = null;
```

This field gives a user of the TreeMap class a choice. If the user wants the "natural" ordering, such as alphabetical order for String keys or increasing order for Integer keys, the user creates a TreeMap instance with the default constructor. Then the keys' class must implement the Comparable interface, so comparisons are based on the compareTo method in the key class. Alternatively, as we saw in Section 12.3, a user can override the "natural" ordering by supplying a Comparator object in the constructor call:

```
TreeMap<String, Integer> wordLengths =
                          new TreeMap<String, Integer>(new ByLength());
```

The designers of the Java Collections Framework's TreeMap class chose a red-black tree as the underlying structure because it had a slight speed advantage over an AVL tree for insertions, removals, and searches. We will now start to get into the red-black aspects of the TreeMap class. There are two constant identifiers that supply the colors:

```
private static final boolean RED = false;

private static final boolean BLACK = true;
```

These constant identifiers apply, not to the tree as a whole, but to the Entry objects in the tree. The Entry class, embedded in the TreeMap class, is similar to the Entry class that is embedded in the BinarySearchTree class, except that the TreeMap class's Entry class has key and value fields (instead of just an element field), and a color field:

```
static class Entry<K, V> implements Map.Entry<K, V>
{
     K key;
     V value;
     Entry<K, V>  left = null;
     Entry<K, V>  right = null;
     Entry<K, V>  parent;
     boolean color = BLACK; // ensures that root's color will start out BLACK
```

Every `Entry` object's `color` field is initialized to `BLACK`. But during an insertion, the inserted `Entry` object is colored `RED`; this simplifies the maintenance of the Path Rule. The `Entry` class also has a default-visibility constructor to initialize the `key`, `value`, and `parent` fields. And there are a few **`public`** methods, such as `getKey()` and `getValue()`, which are useful in iterating over the entries in a `TreeMap` object after a call to the `entrySet()`, `keySet()`, or `values()`, methods.

To finish up our overview of the `TreeMap` implementation of the `Map` interface, we consider a few method definitions in Section 12.3.2.

12.3.2 Method Definitions in the TreeMap Class

We will focus on the definitions of the `put` and `remove` methods. As you might expect, the definitions of those methods are quite similar to the definitions of the `add` and `remove` methods in the `AVLTree` class. But one obvious difference is that, for the sake of simplicity, we restricted `AVLTree` elements to the "natural" ordering with an implementation of the `Comparable` interface. Users of the `TreeMap` class can guarantee the elements in a `TreeMap` instance will be ordered "naturally" by invoking the default constructor. Or, as we saw in Section 12.3.1, a user can override the natural ordering by invoking the constructor that takes a `Comparator` argument.

The definition of the `put (K key, V value)` method starts by initializing an `Entry`:

```
Entry<K, V>  t = root;
```

Then, just as we did in the `add` method of the `AVLTree` class, we work our way down the tree until we find where `key` is or belongs. Except that the `put` method:

- splits off the `Comparator` case from the `Comparable` case;
- returns `t.setValue (value)` if `key` and `t.key` are the same, and then `value` replaces `t.value` and the old value is returned;
- after an insertion, calls a special method, `fixAfterInsertion`, to re-color and rotate the tree if the Red Rule is no longer satisfied (the Path Rule will still be satisfied because the newly inserted entry is colored `RED` at the start of `fixAfterInsertion`).

Here is the complete definition:

```
public V put (K key, V value)
{
     Entry<K, V>  t = root;

     if (t == null)
     {
           root = new Entry<K,V>(key, value, null);
           size = 1;
```

```
            modCount++;
            return null;
      }
      int cmp;
      Entry<K,V> parent;

      // split comparator and comparable paths
      Comparator<? super K> cpr = comparator;
      if (cpr != null)
      {
            do
            {
                   parent = t;
                   cmp = cpr.compare(key, t.key);
                   if (cmp < 0)
                           t = t.left;
                   else if (cmp > 0)
                           t = t.right;
                   else
                           return t.setValue(value);
            } while (t != null);
      } // the keys are ordered by the comparator
      else
      {
            if (key == null)
                   throw new NullPointerException();
            Comparable<? super K> k = (Comparable<? super K>) key;
            do
            {
                   parent = t;
                   cmp = k.compareTo(t.key);
                   if (cmp < 0)
                           t = t.left;
                   else if (cmp > 0)
                           t = t.right;
                   else
                           return t.setValue(value);
            } while (t != null);
      } // the keys are ordered "naturally"
      Entry<K,V> e = new Entry<K,V>(key, value, parent);
      if (cmp < 0)
              parent.left = e;
      else
              parent.right = e;
      fixAfterInsertion(e);
      size++;
      modCount++;
      return null;
 } // method put
```

Notice that the fixAfterInsertion method is not called when an insertion is made at the root. So root remains BLACK in that case.

The definition of the fixAfterInsertion method is not intuitively obvious. In fact, even if you study the code, it makes no sense. Red-black trees were originally developed in Bayer [1972]. The algorithms for inserting and removing in these trees, called "2-3-4 trees," were lengthy but the overall strategy was easy to understand. Shorter but harder to follow methods were supplied when the red-black coloring was imposed on these structures in Guibas [1978].

Lab 19 investigates the fixAfterInsertion method in some detail.

You are now ready for Lab 19: The fixAfterInsertion Method

In Section 12.1.1, we stated that the height of any red-black tree is logarithmic in n, the number of elements in the tree. So for the part of the put method that finds where the element is to be inserted, worstTime(n) is logarithmic in n. Then a call to fixAfterInsertion is made, for which worstTime(n) is also logarithmic in n. We conclude that, for the entire put method, worstTime(n) is logarithmic in n.

The remove method, only slightly changed from that of the BinarySearchTree class, gets the Entry object corresponding to the given key and then deletes that Entry object from the tree:

```
public V remove (K key)
{
     Entry<K, V>  p = getEntry (key);
     if (p == null)
            return p;
     V oldValue = p.value;
     deleteEntry (p);
     return oldValue;
} // method remove
```

The getEntry method is almost identical to the BinarySearchTree class's getEntry method except—as we saw with the put method—that there is a split of the Comparator and Comparable cases.

The deleteEntry method mimics the BinarySearchTree class's (and AVLTree class's) delete Entry method, except now we must ensure that the Path Rule is still satisfied after the deletion. To see how we might have a problem, suppose we want to delete the entry with key 50 from the TreeMap object in Figure 12.5. The value parts are omitted because they are irrelevant to this discussion, and we pretend that the keys are of type **int** ; they are actually of type reference-to-Integer.

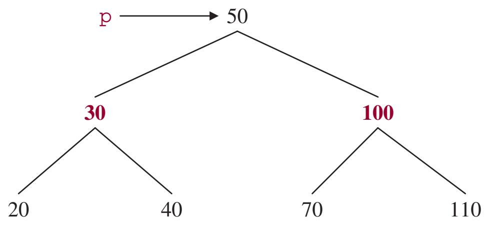


FIGURE 12.5 A TreeMap (with value parts not shown) from which 50 is to be deleted

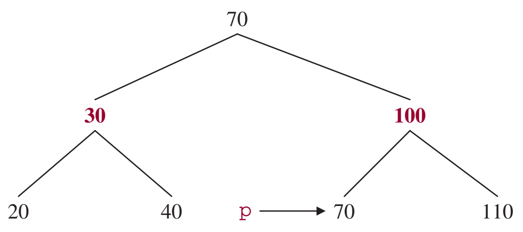


FIGURE 12.6 An intermediate stage in the deletion of 50 from the TreeMap object of Figure 12.5

Just as we did with the BinarySearchTree class's deleteEntry method, the successor's key (namely, 70) replaces 50 and then p references that successor. See Figure 12.6 above.

If we were performing a BinarySearchTree -style deletion, we would simply unlink p's Entry object and be done. But if we unlink that Entry object from the TreeMap object of Figure 12.5, the Path Rule would be violated. Why? There would be only one black element in the path from the root to 100 (an element with one child), and two black elements in the path from the root to the leaf 110. To perform any necessary re-coloring and re-structuring, there is a fixAfterDeletion method.

Here is the definition of the deleteEntry method, which is very similar to the definition of the deleteEntry method in both the BinarySearchTree and AVLTree classes:

```
private void deleteEntry (Entry<K, V>  p)
{
      modCount++;
      size-;

      // If strictly internal, replace p's element with its successor's element
      // and then make p reference that successor.
      if (p.left != null && p.right != null)
      {
            Entry<K, V>  s = successor (p);
            p.key = s.key;
            p.value = s.value;
            p = s;
      } // p has two children

      // Start fixup at replacement node, if it exists.
      Entry<K, V>  replacement = (p.left != null? p.left : p.right);
      if (replacement != null)
            {
            // Link replacement to parent
            replacement.parent = p.parent;
            if (p.parent == null)
                root = replacement;
            else if (p == p.parent.left)
                p.parent.left  = replacement;
            else
                p.parent.right = replacement;
            // Fix replacement
            if (p.color == BLACK)
                fixAfterDeletion(replacement);
      }
      else if (p.parent == null)
      { // return if we are the only node.
            root = null;
            }
            else
            { // No children. Use self as phantom replacement and unlink.
                if (p.color == BLACK)
                    fixAfterDeletion(p);
                if (p.parent != null)
                {
                    if (p == p.parent.left)
                        p.parent.left = null;
                    else if (p == p.parent.right)
                        p.parent.right = null;
                } // non-null parent
            } // p has no children
      } // method deleteEntry
```

The `fixAfterDeletion` method, the subject of Lab 20, has even more cases than the `fixAfterInsertion` method.

You are now ready for Lab 20: The `fixAfterDeletion` Method

The ch12 directory on the book's website includes an applet that will help you to visualize insertions in and removals from a red-black tree:

In Section 12.4, we develop an application of the `TreeMap` class to print out the synonyms of given words.

12.4 Application of the `TreeMap` Class: a Simple Thesaurus

A ***thesaurus*** is a dictionary of synonyms. For example, here is a small thesaurus, with each word followed by its synonyms:

```
close near confined
confined close cramped
correct true
cramped confined
near close
one singular unique
singular one unique
true correct
unique singular one
```

The problem we want to solve is this: given a thesaurus file and a file of words whose synonyms are requested, print the synonym of each word entered to a file

Analysis If there is no file for either path input, or if the output file path is illegal, an error message should be printed, followed by a re-prompt. The thesaurus file will be in alphabetical order. For each word entered from the requests file, the synonyms of that word should be output to the synonyms file, provided the word's synonyms are in the thesaurus file. Otherwise, a synonyms-not-found message should be printed. In the following system test, assume that the thesaurus shown earlier in this section is in the file ‘‘thesaurus.in1’’, and ‘‘requests.in1’’ consists of

```
one
two
close
```

```
   System Test (input is boldfaced):
   Please enter the path for the thesaurus file: thesaraus.in1
   java.io.FileNotFoundException: thesaraus.in1 (The system cannot find the file
   specified)

   Please enter the path for the thesaurus file: thesaurus.in1


   Please enter the path for the requests file: requests.in1

   Please enter the path for the synonyms file: synonyms.ou1


   The contents of synonyms.ou1 are now:

   The synonyms of one are [singular, unique]

   two does not appear in the thesaurus.

   The synonyms of close are [near, confined]
```

We will create two classes to solve this problem: a `Thesaurus` class to store the synonym information, and a `ThesaurusUser` class to handle the input/output.

12.4.1 Design, Testing, and Implementation of the `Thesaurus` Class

The `Thesaurus` class will have four responsibilities: to initialize a thesaurus object, to add a line of synonyms to a thesaurus, to return the synonyms of a given word, and—for the sake of testing—to return a string representation of the thesaurus. The synonyms will be returned in a `LinkedList` object. In the specifications, *n* refers to the number of lines in the thesaurus file.

Here are the method specifications:

```
     /**
      *  Initializes this Thesaurus object.
      *
      */
     public Thesaurus( )
```

```
/**
 *  Adds a specified line of synonyms to this Thesaurus object.
 *  The worstTime(n) is O(log n).
 *
 *  @param line – the specified line of synonyms to be added to this
 *         Thesaurus object.
 *  @throws NullPointerException – if line is null.
 *
 */
public void add (String line)


/**
 *  Finds the LinkedList of synonyms of a specified word in this Thesaurus.
 *  The worstTime(n) is O(log n).
 *
 *  @param word – the specified word, whose synonyms are to be
 *         returned.
 *
 *  @return the LinkedList of synonyms of word.
 *
 *  @throws NullPointerException – if word is null.
 *
 */
public LinkedList<String> getSynonyms (String word)

/**
 *  Returns a String representation of this Thesaurus object.
 *  The worstTime(n) is O(n).
 *
 *  @return a String representation of this Thesaurus object in the
 *          form {word1=[syn11, syn12,...], word2=[syn21, syn22,...],...}.
 *
 */
public String toString()
```

Here are two tests in the ThesaurusTest class, which has thesaurus (an instance of the Thesaurus class) as a field:

```
@Test (expected = NullPointerException.class)
public void testAddLineNull()
{
     thesaurus.add (null);
} // method testAddLineNull


@Test
public void testAdd1()
{
     thesaurus.add ("therefore since because ergo");
```

```
          assertEquals ("{therefore=[since, because, ergo]}", thesaurus.toString());
     } // method testAdd1
```

The complete test suite is available from the book's website.

The only field in the `Thesaurus` class is a `TreeMap` object in which the key is a word and the value is the linked list of synonyms of the word:

```
protected TreeMap<String, LinkedList<String>> thesaurusMap;
```

The implementation of the `Thesaurus` class is fairly straightforward; most of the work is done in the `put` and `get` methods of the `TreeMap` class. The `Thesaurus` class's `add` method tokenizes the line, saves the first token as the key, and saves the remaining tokens in a `LinkedList` object as the value.

Here are the method definitions and time estimates:

```
public Thesaurus()
{
      thesaurusMap = new TreeMap<String, LinkedList<String>>();
 } // default constructor


public void add (String line)
{
      LinkedList<String> synonymList = new LinkedList<String>();

      Scanner sc = new Scanner (line);

      if (sc.hasNext())
      {
           String word = sc.next();

           while (sc.hasNext())
                 synonymList.add (sc.next());
           thesaurusMap.put (word, synonymList);
      } // if
} // method add
```

For the `put` method in the `TreeMap` class, worstTime(n) is logarithmic in n, and so that is also the time estimate for the `add` method. Note that the **`while`** loop takes constant time because it is independent of n, the number of lines in the thesaurus.

Here is the one-line `getSynonyms` method:

```
public LinkedList<String> getSynonyms (String word)
{
        return thesaurusMap.get (word);
} // method getSynonyms
```

For the `getSynonyms` method, worstTime(n) is logarithmic in n because that is the time estimate for the `TreeMap` class's `get` method.

The definition of the `toString()` method is just as simple:

```
public String toString()
{
    return thesaurusMap.toString();
} // method toString
```

For this method, worstTime(n) is linear in n—the iteration over the entries accesses a key and value in constant time.

12.4.2 Design and Testing of the ThesaurusUser Class

The ThesaurusUser class's run method creates a thesaurus from a file whose file-path is scanned in from the keyboard, and then creates, from paths scanned in, a scanner for a requests file and a print writer for synonyms file. Then the findSynonyms method produces the synonyms file from the thesaurus and the requests file. Here are the corresponding method specifications:

```
/**
 *  Constructs a thesaurus from a file whose path is read in from the keyboard
 *  and creates, from paths scanned in, a scanner for a requests file and a print
 *  writer for the synonyms file.
 *  The worstTime(n) is O(n log n).
 *
 */
public void run()


/**
 *  Outputs the synonyms of the words in the file scanned to a specified file.
 *  The worstTime(n, m) is O(m log n), where n is the number of lines in
 *  the thesaurus, and m is the number of words in the file scanned.
 *
 *  @param thesaurus - the thesaurus of words and synonyms.
 *  @param requestFileScanner - the Scanner over the file that holds the
 *         words whose synonyms are requested.
 *  @param synonymPrintWriter - the PrintWriter for the file that will hold
 *         the synonyms of the words in the request file.
 *
 */
public void findSynonyms (Thesaurus thesaurus, Scanner requestFileScanner,
                          PrintWriter synonymPrintWriter)
```

The run method is not testable because it deals mainly with end-user input and output. The findSynonyms method is testable, and here is one of those tests (user, of type ThesaurusUser and line, of type String, are fields in ThesaurusUserTest):

```
@Test
public void testProcessFilesNormal() throws IOException
{
    Scanner thesaurusFileScanner = new Scanner (new File ("thesaurus.in1")),
            requestFileScanner = new Scanner (new File ("requests.in1"));

    PrintWriter synonymPrintWriter = new PrintWriter (new BufferedWriter
                                     (new FileWriter ("synonyms.ou1")));

    Thesaurus thesaurus = new Thesaurus();
    while (thesaurusFileScanner.hasNext())
              thesaurus.add (thesaurusFileScanner.nextLine());
```

```
        user.findSynonyms (thesaurus, requestFileScanner, synonymFileWriter);
        synonymFileWriter.close();
        Scanner sc = new Scanner (new File ("synonyms.ou1"));

        line = sc.nextLine();
        assertEquals ("Here are the synonyms of confined: [close, cramped]", line);
        line = sc.nextLine();
        assertEquals ("Here are the synonyms of near: [close]", line);
        line = sc.nextLine();
        assertEquals ("x does not appear in the thesaurus.", line);
        line = sc.nextLine();
        assertEquals ("Here are the synonyms of singular: [one, unique]", line);
    } // method testProcessFilesNormal
```

The book's website includes the complete ThesaurusUserTest class.

Figure 12.7 has the UML class diagrams for this project. The line just below the diagram for ThesaurusUser signifies that the ThesaurusUser class has an association—specifically, a method parameter—with the Thesaurus class.

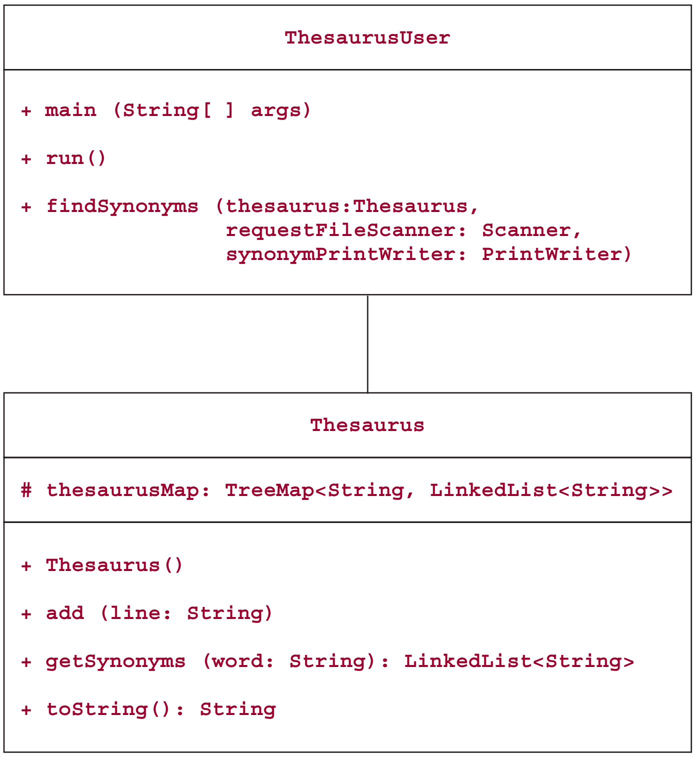

ThesaurusUser
+ main (String[ ] args)
+ run()
+ findSynonyms (thesaurus:Thesaurus, requestFileScanner: Scanner, synonymPrintWriter: PrintWriter)
Thesaurus
# thesaurusMap: TreeMap<String, LinkedList<String>>
+ Thesaurus()
+ add (line: String)
+ getSynonyms (word: String): LinkedList<String>
+ toString(): String

FIGURE 12.7 Class diagrams for the Thesaurus project

12.4.3 Implementation of the ThesaurusUser Class

As always, the main method simply invokes the run method on a newly constructed ThesaurusUser object:

```
public static void main (String[] args)
{
        new ThesaurusUser().run();
} // method main
```

The ThesaurusUser's run method scans a file path (and keeps scanning until a legal file path is scanned in), adds each line in the file to the thesaurus, and then finds the synonyms of each word in a file whose path is scanned in:

```
public void run()
{
  final String THESAURUS_FILE_PROMPT =
       "\nPlease enter the path for the thesaurus file: ";

  final String REQUEST_FILE_PROMPT =
       "\nPlease enter the path for the file with the words " +
       "whose synonyms are requested: ";

  final String SYNONYM_FILE_PROMPT =
       "\nPlease enter the path for the file that will " +
       "hold the synonyms of each word in the request file: ";

  final String NO_INPUT_FILE_FOUND_MESSAGE =
       "Error: there is no file with that path.\n\n";

  Thesaurus thesaurus = new Thesaurus();

  Scanner keyboardScanner = new Scanner (System.in),
          thesaurusFileScanner,
          requestFileScanner;

  PrintWriter synonymPrintWriter;

  String thesaurusFilePath,
         requestFilePath,
         synonymFilePath;

  boolean pathsOK = false;

  while (!pathsOK)
  {
       try
       {
           System.out.print (THESAURUS_FILE_PROMPT);
           thesaurusFilePath = keyboardScanner.nextLine();
```

```
                thesaurusFileScanner = new Scanner (new File (thesaurusFilePath));
                while (thesaurusFileScanner.hasNext())
                    thesaurus.add (thesaurusFileScanner.nextLine());

                System.out.print (REQUEST_FILE_PROMPT);
                requestFilePath = keyboardScanner.nextLine();
                requestFileScanner = new Scanner (new File (requestFilePath));

                System.out.print (SYNONYM_FILE_PROMPT);
                synonymFilePath = keyboardScanner.nextLine();
                synonymPrintWriter = new PrintWriter (new BufferedWriter
                    (new FileWriter (synonymFilePath)));

                pathsOK = true;
                findSynonyms (thesaurus, requestFileScanner, synonymPrintWriter);
                synonymFileWriter.close();
            } // try
            catch (IOException e)
            {
                System.out.println (e);
            } // catch
       } // while !pathsOK
     } // method run
```

Intuitively, since it takes logarithmic-in-n time for each insertion into the thesaurus, it should take linear-logarithmic-in-n time for n insertions. But the first insertion is into an empty tree, the second insertion is into a tree with one element, and so on. Specifically, for $i = 1, 2, \ldots, n$, it takes approximately $\log_2 i$ loop iterations to insert the ith element into a red-black tree. To insert n elements, the total number of iterations is, approximately,

$$\sum_{i=2}^{n} \log_2 i = \log_2 n! \qquad //\text{sum of logs} = \text{log of product}$$

$$\approx n \log_2 n \qquad //\text{by the logarithmic form of Stirling's}$$
$$//\text{approximation of factorials (see Zwilliger [2000])}$$

In other words, our intuition is correct, and worstTime(n) is linear-logarithmic in n for filling in the thesaurus. To estimate the worst time for the entire `run` method, we first need to develop and then estimate the worst time for the `findSynonyms` method, because `findSynonyms` is called by the `run` method.

The `findSynonyms` method consists of a read-loop that continues as long as there are words left in the requests file. For each word scanned, the synonyms of that word are fetched from the thesaurus and output to the synonyms file; an error message is output if the word is not in the thesaurus. Here is the method definition:

```
     public void findSynonyms (Thesaurus thesaurus, Scanner requestFileScanner,
                               PrintWriter synonymPrintWriter)
     {
         final String WORD_NOT_FOUND_MESSAGE =
             " does not appear in the thesaurus.";
```

```
        final String SYNONYM_MESSAGE = "Here are the synonyms of ";

        String word;

        LinkedList<String> synonymList;

        while (requestFileScanner.hasNext())
        {
            word = requestFileScanner.next();
            synonymList = thesaurus.getSynonyms (word);
            if (synonymList == null)
                synonymPrintWriter.println (word + WORD_NOT_FOUND_MESSAGE);
            else
                synonymPrintWriter.println (SYNONYM_MESSAGE + word +
                        ": " + synonymList);
        } // while
    } // method findSynonyms
```

To estimate how long the `findSynonyms` method takes, we must take into account the number of words entered from the requests file as well as the size of `thesaurusMap`. Assume there are m words entered from the requests file. (We cannot use n here because that represents the size of `thesaurusMap`.) Then the **`while`** loop in `findSynonyms` is executed $O(m)$ times. During each iteration of the **`while`** loop, there is a call to the `get` method in the `TreeMap` class, and that call takes $O(\log n)$ time. So worstTime(n, m) is $O(m \log n)$; in fact, to utilize the notation from Chapter 3, worstTime(n, m) is $\Theta(m \log n)$ because $m \log n$ provides a lower bound as well as an upper bound on worstTime(n, m). The worstTime function has two arguments because the total number of statements executed depends on both n and m.

We can now estimate the worst time for the `run` method. For the loop to fill in the thesaurus, worstTime(n) is linear-logarithmic in n, and for the call to `findSynonyms`, worstTime(n, m) is $\Theta(m \log n)$. We assume that the number of requests will be less than the size of `thesaurusMap`. Then for the `run` method, worstTime(n, m) is $\Theta(n \log n)$, that is, linear logarithmic in n.

The next topic in this chapter is the `TreeSet` class, which is implemented as a `TreeMap` in which each `Entry` object has the same dummy value-part.

12.5 The TreeSet Class

We need to go through a little bit of background before we can discuss the `TreeSet` class. Recall, from Chapter 10, that the `Set` interface extends the `Collection` interface by stipulating that duplicate elements are not allowed. The `SortedSet` interface extends the `Set` interface in two ways:

1. by stipulating that its iterator must traverse the `Set` in order of ascending elements;
2. by including a few new methods relating to the ordering, such as `first()`, which returns the smallest element in the instance, and `last()`, which returns the largest element in the instance.

The `TreeSet` class implements the `SortedSet` interface, and extends the `AbstractSet` class, which has a bare-bones implementation of the `Set` interface.

The bottom line is that a `TreeSet` is a `Collection` in which the elements are ordered from smallest to largest, and there are no duplicates. Most importantly, for the `TreeSet` class's `contains`, `add`, and `remove` methods, worstTime(n) is logarithmic in n. So if these criteria suit your application, use a `TreeSet` instead of an ArrayList, LinkedList, BinarySearchTree, or array. For those four collections, if the elements are to be maintained in order from smallest to largest, worstTime(n) is linear in n for insertions and removals.

How does the TreeSet class compare with the AVLTree class? The TreeSet class is superior because it is part of the Java Collections Framework. As a result, the class is available to you on any Java compiler. Also, the methods have been thoroughly tested. Finally, you are not restricted to the "natural" ordering of elements: You can override that ordering with a comparator.

We already saw most of the TreeSet methods when we studied the BinarySearchTree and AVLTree classes in Chapter 10.

The following class illustrates both the default constructor and the constructor with a comparator parameter, as well as a few other methods:

```
import java.util.*;

public class  TreeSetExample
{
    public static void main (String[ ] args)
    {
        new TreeSetExample().run();
    } // method main

    public void run()
    {
        final String START = "Here is the TreeSet:\n";

        final String ADD =
              "\nAfter adding \"tranquil\", here is the TreeSet:\n";

        final String REMOVE =
              "\nAfter removing \"serene\", here is the TreeSet:\n";

        final String REVERSE =
              "\n\nHere are the scores in decreasing order:\n";

        final String SUM = "The sum of the scores is ";

        TreeSet<String> mySet = new TreeSet<String>();

        TreeSet<Integer> scores = new TreeSet<Integer>
                                 (new Decreasing ());

        mySet.add ("happy");
        mySet.add ("always");
        mySet.add ("yes");
        mySet.add ("serene");
        System.out.println (START + mySet);

        if (mySet.add ("happy"))
              System.out.println ("ooops");
        else
```

```
                System.out.println ("\n\"happy\" was not added " +
                                    "because it was already there");
          mySet.add ("tranquil");
          System.out.println (ADD + mySet);
          System.out.println ("size = " + mySet.size());
          if (mySet.contains ("no"))
                System.out.println ("How did \"no\" get in there?");
          else
                System.out.println ("\n\"no\" is not in the TreeSet");
          if (mySet.remove ("serene"))
                System.out.println (REMOVE + mySet);

          for (int i = 0; i < 5; i++)
                scores.add (i);
          System.out.println (REVERSE + scores);

          int sum = 0;
          for (Integer i : scores)
                sum += i;
                    System.out.println (SUM + sum);
      } // method run

  } // class TreeSetExample
```

Here is the output:

```
Here is the TreeSet:
[always, happy, serene, yes]

"happy" was not added because it was already there

After adding "tranquil", here is the TreeSet:
[always, happy, serene, tranquil, yes]
size = 5

"no" is not in the TreeSet

After removing "serene", here is the TreeSet:
[always, happy, tranquil, yes]


Here are the scores in decreasing order:
[4, 3, 2, 1, 0]

The sum of the scores is 10
```

After we take a brief look at the implementation of the `TreeSet` class, we will return to a user's view by developing an application on spell checking.

12.5.1 Implementation of the TreeSet Class

The TreeSet class is based on the TreeMap class, which implements a red-black tree. Basically, a TreeSet object is a TreeMap object in which each element has the same dummy value. Recall that it is legal for different TreeMap elements to have the same values; it would be illegal for different TreeMap elements to have the same keys. Here is the start of the declaration of the TreeSet class:

```
public class TreeSet<E>
           extends AbstractSet<E>
           implements NavigableSet<E>, Cloneable, java.io.Serializable
{
      private transient NavigableMap<E, Object> m;  // The backing Map

      // Dummy value to associate with an Object in the backing Map
      private static final Object PRESENT = new Object();
```

"Navigable" means that the set (or map) can be iterated over from back to front as well as from front to back.

To explicitly construct a TreeSet object from a given SortedSet object, usually a TreeSet object, there is a constructor with default visibility (that is, accessible only from within the package java.util):

```
/**
 *  Initializes this TreeSet object from a specified NavigableMap object.
 *
 *  @param m – the NavigableMap that this TreeSet object is initialized from.
 *
 */
TreeSet<E> (NavigableMap<E, Object> m)
{
    this.m = m;
} // constructor with map parameter
```

Given the TreeSet fields and this constructor, we can straightforwardly implement the rest of the TreeSet methods. In fact, most of the definitions are one-liners. For example, here are the definitions of the default constructor, the constructor with a comparator parameter, and the contains, add, and remove methods:

```
/**
 * Initializes this TreeSet object to be empty, with the elements to be
 * ordered by the Comparable interface.
 *
 */
public TreeSet()
{
      this (new TreeMap<E, Object>());
} // default constructor


/**
 *  Initializes this TreeSet object to be empty, with elements to be ordered
 *  by a specified Comparator object.
 *
 *  @param c – the specified Comparator object by which the elements in
 *         this TreeSet object are to be ordered.
 *
 */
public TreeSet (Comparator<? super E> c)
{
      this (new TreeMap<E, Object>(c));
}


/**
 *  Determines if this TreeSet object contains a specified element.
 *  The worstTime(n) is O(log n).
 *
 *  @param obj – the specified element sought in this TreeSet object.
 *
 *  @return true – if obj is equal to at least one of the elements in this
 *          TreeSet object; otherwise, return false.
 *
 *  @throws ClassCastException – if obj cannot be compared to the
 *          elements in this TreeSet object.
 *
 */
public boolean contains (Object obj)
{
      return m.containsKey (obj);
} // method contains


/**
 *  Inserts a specified element where it belongs in this TreeSet object,
 *  unless the element is already in this TreeSet object.
 *  The worstTime(n) is O(log n).
 *
 *  @param element – the element to be inserted, unless already there, into
 *         this TreeSet object.
 *
 *  @return true – if this element was inserted; return false – if this element
 *          was already in this TreeSet object.
 *
 */
public boolean add (E element)
{
      return m.put (element, PRESENT) == null;
} // method add


/**
 *  Removes a specified element from this TreeSet object, unless the
 *  element was not in this TreeSet object just before this call.
 *  The worstTime(n) is O (log n).
 *
 *  @param element – the element to be removed, unless it is not there,
 *         from this TreeSet object.
 *
 *  @return true – if element was removed from this TreeSet object;
 *          otherwise, return false.
 *
public boolean remove (Object element)
{
      return m.remove (element) == PRESENT;
} // method remove
```

Section 12.5.2 has an application of the `TreeSet` class: developing a spell checker.

12.5.2 Application: A Simple Spell Checker

One of the most helpful features of modern word processors is spell checking: scanning a document for possible misspellings. We say "possible" misspellings because the document may contain words that are legal but not found in a dictionary. For example, "iterator" and "postorder" were cited as *not found* by the word processor (Microsoft Word) used in typing this chapter.

The overall problem is this: Given a dictionary and a document, in files whose names are provided by the end user, print out all words in the document that are not found in the dictionary.

Analysis We make some simplifying assumptions:

1. The dictionary consists of lower-case words only, one per line (with no definitions).
2. Each word in the document consists of letters only—some or all may be in upper-case.
3. The words in the document are separated from each other by at least one non-alphabetic character (such as a punctuation symbol, a blank, or an end-of-line marker).
4. The dictionary file is in alphabetical order. The document file, not necessarily in alphabetical order, will fit in memory (along with the dictionary file) if duplicates are excluded.

Here are the contents of a small dictionary file called "dictionary.dat", a small document file called "document.dat" and the words in the latter that are not in the former.

```
// the dictionary file:
a
algorithms
asterisk
coat
equal
he
pied
pile
plus
programs
separate
she
```

```
structures
wore

// the document file:
Alogrithms plus Data Structures equal Programs.
She woar a pide coat.

// the possibly misspelled words:
alogrithms, data, pide, woar
```

To isolate the spell-checking details from the input/output aspects, we create two classes: `SpellChecker` and `SpellCheckerUser`.

12.5.2.1 Design, Testing, and Implementation of the SpellChecker Class

The `SpellChecker` class will have four responsibilities:

- to initialize a `SpellChecker` object
- to add a word to the set of dictionary words;
- to add the words in a line to the set of document words;
- to return a `LinkedList` of words from the document that are not in the dictionary.

The use of the term "set" in the second and third responsibilities implies that there will be no duplicates in either collection; there may have been duplicates in the dictionary or document files. Here are the method specifications for the methods in the `SpellChecker` class:

```
/**
 *  Initializes this SpellChecker object.
 *
 */
public SpellChecker()

/**
 *  Inserts a specified word into the dictionary of words.
 *  The worstTime(n) is O(log n), where n is the number of words in the
 *  dictionary of words.
 *
 *  @param word - the word to be inserted into the dictionary of words.
 *
 *  @return a String representation of the dictionary.
 *
 *  @throws NullPointerException - if word is null.
 *
 */
public String addToDictionary (String word)


/**
 *  Inserts all of the words in a specified line into the document of words.
 *  The worstTime(m) is O(log m), where m is the number of (unique) words
 *  in the document of words.
 *
 *  @param line - the line whose words are added to the document of words.
 *
 *  @return a String representation of the document
 *
 *  @throws NullPointerException - if line is null.
 *
 */
public String addToDocument (String line)


/**
 *  Determines all words that are in the document but not in the dictionary.
 *  The worstTime(m, n) is O(m log n), where m is the number of words
 *  in the document, and n is the number of words in the dictionary.
 *
 *  @return a LinkedList consisting of all the words in the document that
 *          are not in the dictionary.
 *
 */
public LinkedList<String> compare()
```

Here are two of the tests in `SpellCheckerTest`, which has `spellChecker` (always initialized to an empty `SpellChecker` instance) as a field:

```
@Test
public void testAddToDocument1()
{
     String actual = spellChecker.addToDocument ("A man, a plan, a canal. Panama!");
     assertEquals ("[a, canal, man, panama, plan]", actual);
} // method testAddToDocument1

@Test
public void testSeveralMisspellings1()
{
  spellChecker.addToDictionarySet ("separate");
  spellChecker.addToDictionarySet ("algorithms");
  spellChecker.addToDictionarySet ("equals");
  spellChecker.addToDictionarySet ("asterisk");
  spellChecker.addToDictionarySet ("wore");
  spellChecker.addToDictionarySet ("coat");
  spellChecker.addToDictionarySet ("she");
  spellChecker.addToDictionarySet ("equals");
  spellChecker.addToDictionarySet ("plus");
  spellChecker.addToDictionarySet ("he");
  spellChecker.addToDictionarySet ("pied");
  spellChecker.addToDictionarySet ("a");
  spellChecker.addToDictionarySet ("pile");
  spellChecker.addToDictionarySet ("programs");
  spellChecker.addToDictionarySet ("structures");
```

```
      spellChecker.addToDocumentSet ("Alogrithms plus Data Structures equal Programs.");
      String expected = "[alogrithms, data, equal]",
             actual = spellChecker.compare().toString();
      assertEquals (expected, actual);
    } // method testSeveralMisspellings1
```

The `SpellChecker` class has only two fields:

```
    protected TreeSet<String> dictionarySet,
                              documentSet;
```

The `dictionarySet` field holds the words in the dictionary file. The `documentSet` field holds each unique word in the document file—there is no purpose in storing multiple copies of any word.

The definitions of the default constructor and `addToDictionarySet` methods hold no surprises:

```
    public SpellChecker()
    {
        dictionarySet = new TreeSet<String>();
        documentSet = new TreeSet<String>();
    } // default constructor


    public String addToDictionary (String word)
    {
         dictionarySet.add (word);
         return dictionarySet.toString();
    } // method addToDictionary
```

The definition of `addToDocumentSet (String line)` is slightly more complicated. The line is tokenized, with delimiters that include punctuation symbols. Each word, as a token, is converted to lower-case and inserted into `documentSet` unless the word is already in `documentSet`. Here is the definition (see Section 0.2.5 of Chapter 0 for a discussion of the `useDelimiter` method):

```
    public String addToDocument (String line)
    {
       final String DELIMITERS = "[^a-zA-Z]+";

       Scanner sc = new Scanner (line).useDelimiter (DELIMITERS);

       String word;

       while (sc.hasNext())
        {
               word = sc.next().toLowerCase();
               documentSet.add (word);
        } // while line has more tokens
        return documentSet.toString();
    } // method addToDocument
```

Let m represent the number of words in `documentSet`. Each call to the `TreeSet` class's `add` method takes logarithmic-in-m time. The number of words on a line is independent of m, so for the `addToDocument` method, worstTime(m) is logarithmic in m.

The compare method iterates through documentSet; each word that is not in dictionarySet is appended to a LinkedList object of (possibly) misspelled words. Here is the definition:

```
public LinkedList<String> compare()
{
       LinkedList<String> misspelled = new LinkedList<String>();

       for (String word : documentSet)
            if (!dictionarySet.contains (word))
                 misspelled.add (word);
       return misspelled;
} // method compare
```

For iterating through documentSet, worstTime(m) is linear in m, and for each call to the TreeSet class's contains method, worstTime(n) is logarithmic in n. So for the compare method in the SpellChecker class, worstTime(n, m) is $O(m \log n)$. In fact, worstTime(n, m) is $\Theta(m \log n)$.

In Chapter 14, we will encounter another class that implements the Set interface: the HashSet class. In this class, the average time for insertions, removals and searches is constant! So we can re-do the above problem with HashSet object for dictionarySet and documentSet. No other changes need be made! For that version of the spell-check project, averageTime(n) would be constant for the addToDictionary method, and averageTime(m) would be constant for the addToDocument method. For the compare method, averageTime(m, n) would be linear in m. But don't sell your stock in TreeSets-R-Us. For the HashSet version of the SpellChecker class, the worstTime(m, n) for compare, for example, would be $\Theta(mn)$.

12.5.2.2 Design of the SpellCheckerUser Class

The SpellCheckerUser class has the usual main method that invokes a run method, which handles end-user input and output, creates the dictionary and document, and compares the two.

Figure 12.8 shows the class diagrams for this project.

12.5.2.3 Implementation of the SpellCheckerUser Class

The run method scans a file path from the keyboard and constructs a file scanner for that file. Then, depending on whether fileType is "dictionary" or "document," each line from the file is read and added to the dictionary set or the document set, respectively. Finally, the possibly misspelled words are printed. Here is the method definition:

```
public void run()
{
       final int FILE_TYPES = 2;

       final String DICTIONARY = "dictionary";

       final String DOCUMENT = "document";

       final String ALL_CORRECT =
                 "\n\nAll the words are spelled correctly.";

       final String MISSPELLED =
                 "\n\nThe following words are misspelled:";
```

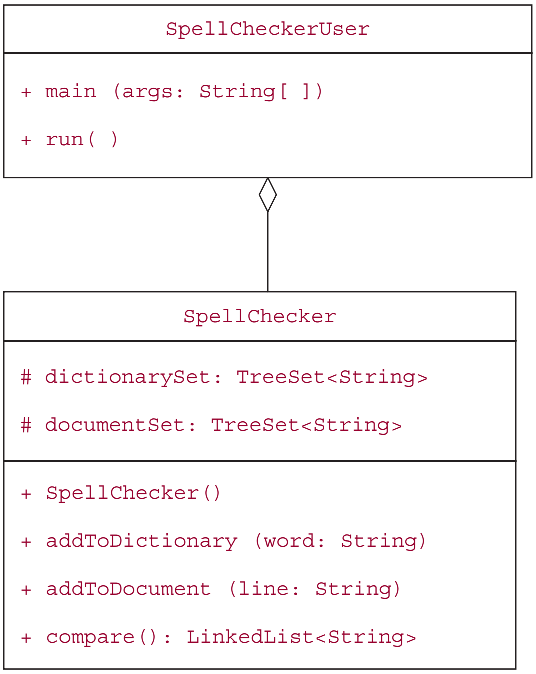

SpellCheckerUser
+ main (args: String[ ])
+ run( )
SpellChecker
# dictionarySet: TreeSet<String>
# documentSet: TreeSet<String>
+ SpellChecker()
+ addToDictionary (word: String)
+ addToDocument (line: String)
+ compare(): LinkedList<String>

FIGURE 12.8 Class diagrams for the Spell Checker project

```
SpellChecker spellChecker = new SpellChecker();

Scanner keyboardScanner = new Scanner (System.in),
        fileScanner;

String fileType = DICTIONARY,
       filePath;

for (int i = 0; i < FILE_TYPES; i++)
{
    final String FILE_PROMPT =
        "\nPlease enter the path for the " + fileType + " file: ";

    boolean pathOK = false;

    while (!pathOK)
    {
        try
        {
             System.out.print (FILE_PROMPT);
             filePath = keyboardScanner.nextLine();
             fileScanner = new Scanner (new File (filePath));
             pathOK = true;
             if (fileType.equals (DICTIONARY))
```

```
                        while (fileScanner.hasNext())
                           spellChecker.addToDictionary (fileScanner.nextLine());
                     else if (fileType.equals (DOCUMENT))
                        while (fileScanner.hasNext())
                           spellChecker.addToDocument (fileScanner.nextLine());
                 } // try
                 catch (IOException e)
                 {
                     System.out.println (e);
                 } // catch
              } // while

              fileType = DOCUMENT;
           } // for
           LinkedList<String> misspelled = spellChecker.compare();
           if (misspelled == null)
              System.out.println (ALL_CORRECT);
           else
              System.out.println (MISSPELLED + misspelled);
      } // method run
```

To create the dictionary, worstTime(n) is linear-logarithmic in n because each of the n words in the dictionary file is stored in a `TreeSet` object, and for each insertion in a `TreeSet` object, worstTime(n) is logarithmic in n. By the same reasoning, to create the document, worstTime(m) is linear-logarithmic in m (the size of the document file). Also, worstTime (m, n) is $\Theta(m \log n)$ for the call to the `SpellChecker` class's `compare` method.

We conclude that, for the `run` method, worstTime (m, n) is linear-logarithmic in max (m, n).

The book's website includes the `SpellCheckerTest` class. There is no `SpellCheckerUserTest` class because `SpellCheckerUser`'s `main` and `run` methods are untestable.

SUMMARY

A ***red-black tree*** is a binary search tree that is empty or in which the root element is colored black, every other element is colored either red or black, and for which the following two rules hold:

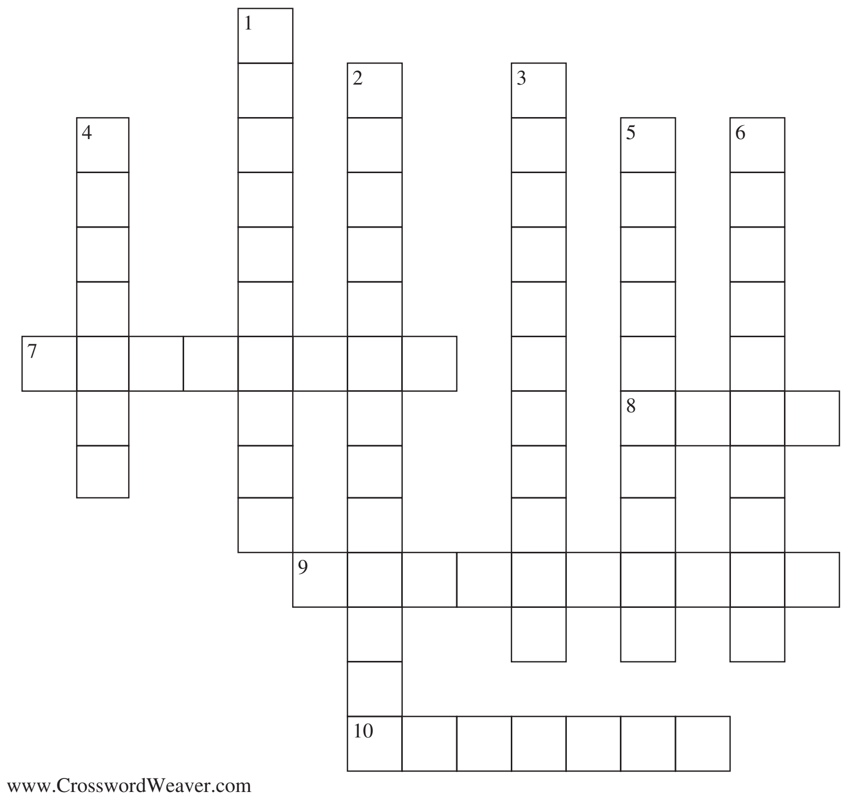
1. **Red Rule:** if an element is colored red, none of its children can be colored red.
2. **Path Rule:** the number of black elements is the same in all paths from the root to elements with one child or with no children.

The height of a red-black tree is always logarithmic in n, the number of elements in the tree.

The Java Collections Framework implements red-black trees in the `TreeMap` class. In a `TreeMap` object each element has a ***key*** part—by which the element is identified—and a ***value*** part, which contains the rest of the element. The elements are stored in a red-black tree in key-ascending order, according to their "natural" order (implementing the `Comparable` interface) or by the order specified by a user-supplied `Comparator` object (there is a constructor in which the `Comparator` object is supplied). For the `containsKey`, `get`, `put`, and `remove` methods, worstTime(n) is logarithmic in n.

A `TreeSet` object is a `Collection` object in which duplicate elements are not allowed and in which the elements are stored in order (according to the `Comparable` ordering or a user-supplied `Comparator`). The `TreeSet` class is implemented in the Java Collections Framework as a `TreeMap` in which each element has the same dummy value-part. For the `contains`, `add`, and `remove` methods, worstTime(n) is logarithmic in n.

CROSSWORD PUZZLE



ACROSS

7. The two components of each element in a map collection

8. In the `TreeMap` class, the value returned by the put method when an element with a new key is inserted

9. What a `TreeSet` object never has

10. In a red-black tree, what the only child of a black element must be

DOWN

1. The `TreeMap` field used to compare elements (by their keys)

2. In the simple thesaurus project, the class that handles input and output

3. For the `containskey`, `get`, `put` and `remove` methods in the `TreeMap` class, worstTime(*n*) is ______ in *n*.

4. The type of the constant identifiers RED and BLACK

5. The kind of search in the contains Value method of the `TreeMap` class

6. The Path Rule states that the number of black elements must be the same in all paths from the root element to elements with ______ or with one child.

CONCEPT EXERCISES

12.1 Construct a red-black tree of height 2 with six elements. Construct a red-black tree of height 3 with six elements.

12.2 Construct two different red-black trees with the same three elements.

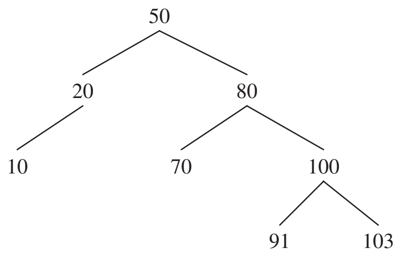
12.3 What is the maximum number of black elements in a red-black tree of height 4? What is the minimum number of black elements in a red-black tree of height 4?

12.4 It is impossible to construct a red-black tree of size 20 with no red elements. Explain.

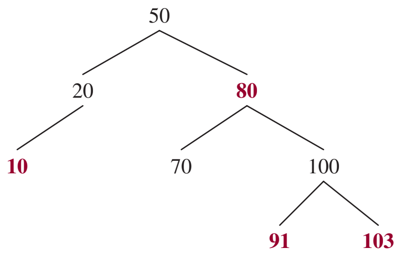
12.5 Suppose *v* is an element with one child in a red-black tree. Explain why *v* must be black and *v*'s child must be a red leaf.

12.6 Construct a red-black tree that (when the colors are ignored) is not an AVL tree.

12.7 Guibas and Sedgewick [1978] provide a simple algorithm for coloring any AVL tree into a red-black tree: For each element in the AVL tree, if the height of the subtree rooted at that element is an even integer and the height of its parent's subtree is odd, color the element red; otherwise, color the element black.
For example, here is an AVL tree from Chapter 10:



In this tree, 20's subtree has a height of 1, 80's subtree has a height of 2, 10's subtree has a height of 0 and 70's subtree has a height of 0. Note that since the root of the entire tree has no parent, this algorithm guarantees that the root will be colored black. Here is that AVL tree, colorized to a red-black tree:



Create an AVL tree of height 4 with $\min_4$ elements (that is, the minimum number of elements for an AVL tree of height 4), and then colorize that tree to a red-black tree.

12.8 Suppose, in the definition of red-black tree, we replace the Path Rule with the following:

> **Pathetic Rule: The number of black elements must be the same in all paths from the root element to a leaf.**

a. With this new definition, describe how to construct a red-black tree of 101 elements whose height is 50.

b. Give an example of a binary search tree that cannot be colored to make it a red-black tree (even with this new definition).

12.9 Show the effect of making the following insertions into an initially empty `TreeSet` object:

30, 40, 20, 90, 10, 50, 70, 60, 80

12.10 Delete 20 and 40 from the `TreeSet` object in Exercise 12.8. Show the complete tree after each deletion.

12.11 Pick any integer $h \geq 1$, and create a `TreeSet` object as follows:

$$\text{let } k = 2^{h+1} + 2^h - 2.$$

Insert 1, 2, 3, $\dots, k$. Remove $k, k-1, k-2, \dots, 2^h$. Try this with $h = 1, 2$ and 3. What is unusual about the red-black trees that you end up with? Alexandru Balan developed this formula.

12.12 From a user's point of view, what is the difference between the `TreeMap` class and the `TreeSet` class?

12.13 From a developer's point of view, what is the relationship between the `TreeMap` class and the `TreeSet` class?

PROGRAMMING EXERCISES

12.1 Suppose we are given the name and division number for each employee in a company. There are no duplicate names. We would like to store this information alphabetically, by name. For example, part of the input might be the following:

Misino, John	8
Nguyen, Viet	14
Panchenko, Eric	6
Dunn, Michael	6
Deusenbery, Amanda	14
Taoubina, Xenia	6

We want these elements stored in the following order:

Deusenbery, Amanda	14
Dunn, Michael	6
Misino, John	8
Nguyen, Viet	14
Panchenko, Eric	6
Taoubina, Xenia	6

How should this be done? `TreeMap`? `TreeSet`? `Comparable`? `Comparator`? Develop a small unit-test to test your hypotheses.

12.2 Re-do Programming Exercise 12.1, but now the ordering should be by increasing division numbers, and within each division number, by alphabetical order of names. For example, part of the input might be the following:

Misino, John	8
Nguyen, Viet	14
Panchenko, Eric	6
Dunn, Michael	6
Deusenbery, Amanda	14
Taoubina, Xenia	6

We want these elements stored in the following order:

Dunn, Michael	6
Panchenko, Eric	6
Taoubina, Xenia	6
Misino, John	8
Deusenbery, Amanda	14
Nguyen, Viet	14

How should this be done? `TreeMap`? `TreeSet`? `Comparable`? `Comparator`? Develop a small unit-test to test your hypotheses.

12.3 Declare two `TreeSet` objects, `set1` and `set2`, whose elements come from the same `Student` class. Each student has a name and grade point average. In `set1`, the students are in alphabetical order. In `set2`, the students are in decreasing order of GPAs. Insert a few students into each set and then print out the set. Include everything needed for this to work, including the two declarations of `TreeSet` objects, the insertion messages, the declaration of the `Student` class, and any other necessary class(es).

Programming Project 12.1

Spell Check, Revisited

Modify the spell-check project and unit-test your modified methods. If document word x is not in the dictionary but word y is in the dictionary and x differs from y either by an adjacent transposition or by a single letter, then y should be proposed as an alternative for x. For example, suppose the document word is “asteriks” and the dictionary contains “asterisk.” By transposing the adjacent letters “s” and “k” in “asteriks,” we get “asterisk.” So “asterisk” should be proposed as an alternative. Similarly, if the document word is “seperate” or “seprate” and the dictionary word is “separate,” then “separate” should be offered as an alternative in either case.

Here are the dictionary words for both system tests:

a

algorithms

asterisk

coat

equal

he

pied

pile

plus

programs

separate

structures

wore

Here is document file doc1.dat:

She woar a pide coat.

And here is document file doc2.dat

Alogrithms plus Data Structures equal Pograms

System Test 1 (with input in boldface):

```
Please enter the name of the dictionary file.
dictionary.dat
Please enter the name of the document file.
doc1.dat

Possible Misspellings          Possible Alternatives
pide                           pied, pile
she                            he
woar
```

System Test 2:

```
Please enter the name of the dictionary file.
dictionary.dat
In the Input line, please enter the name of the document file.
doc2.dat

Possible Misspellings          Possible Alternatives
alogrithms                     algorithms
data
pograms                        programs
```

Programming Project 12.2

Word Frequencies

Design, test, and implement a program to solve the following problem: Given a text, determine the frequency of each word, that is, the number of times each word occurs in the text. Include Big-Θ time estimates of all method definitions.

Analysis

1. The first line of input will contain the path to the text file. The second line of input will contain the path to the output file.
2. Each word in the text consists of letters only (some or all may be in upper-case), except that a word may also have an apostrophe.
3. Words in the text are separated from each other by at least one non-alphabetic character, such as a punctuation symbol, a blank, or an end-of-line marker.
4. The output should consist of the words, lower-cased and in alphabetical order; each word is followed by its frequency.
5. For the entire program, worstTime(n) is O(n log n), where n is the number of distinct words in the text.

Assume that doc1.in contains the following file:

```
This program counts the
number of words in a text.
The text may have many words
in it, including big words.
```

Also, assume that doc2.in contains the following file:

```
Fuzzy Wuzzy was a bear.
Fuzzy Wuzzy had no hair.
Fuzzy Wuzzy wasn't fuzzy.
Was he?
```

System Test 1:

```
Please enter the path to the text file.
doc1.in
Please enter the path to the output file.
doc1.out
```

(Here are the contents of doc1.out after the completion of the program.)
Here are the words and their frequencies:

```
a: 1
big: 1
counts: 1
```

```
have: 1
in: 2
including: 1
it: 1
many: 1
may: 1
number: 1
of: 1
program: 1
text: 2
the: 2
this: 1
words: 3
```

System Test 2:

```
Please enter the path to the text file.
doc2.in
Please enter the path to the output file.
doc2.out
```

(Here are the contents of doc2.out after the completion of the program.)
Here are the words and their frequencies:

```
a: 1
bear: 1
fuzzy: 4
had: 1
hair: 1
he: 1
no: 1
was: 2
wasn't: 1
wuzzy: 3
```

Programming Project 12.3

Building a Concordance

Design, test, and implement a program to solve the following problem: Given a text, develop a concordance for the words in the text. A ***concordance*** consists of each word in the text and, for each word, each line number that the word occurs in. Include Big-Θ time estimates of all methods.

(continued on next page)

(continued from previous page)

Analysis

1. The first line of input will contain the path to the text file. The second line of input will contain the path to the output file.
2. Each word in the text consists of letters only (some or all may be in upper-case), except that a word may also have an apostrophe.
3. The words in the text are separated from each other by at least one non-alphabetic character, such as a punctuation symbol, a blank, or an end-of-line marker.
4. The output should consist of the words, lower-cased and in alphabetical order; each word is followed by each line number that the word occurs in. The line numbers should be separated by commas.
5. The line numbers in the text start at 1.
6. For the entire program, worstTime(n) is O(n log n), where n is the number of distinct words in the text.

Assume that doc1.in contains the following file:

```
This program counts the
number of words in a text.
The text may have many words
in it, including big words.
```

Also, assume that doc2.in contains the following file:

```
Fuzzy Wuzzy was a bear.
Fuzzy Wuzzy had no hair.
Fuzzy Wuzzy wasn't fuzzy.
Was he?
```

System Test 1:

```
Please enter the path to the text file.
doc1.in
Please enter the path to the output file.
doc1.out
```

(Here are the contents of doc1.out after the completion of the program.)
Here is the concordance:

```
a: 2
big: 4
counts: 1
have: 3
in: 2, 4
including: 4
it: 4
many: 3
may: 3
number: 2
```

```
of: 2
program: 1
text: 2, 3
the: 1, 3
this: 1
words: 2, 3, 4
```

System Test 2:

```
Please enter the path to the text file.
doc2.in
Please enter the path to the output file.
doc2.out
```

(Here are the contents of doc2.out after the completion of the program.)
Here is the concordance:

```
a: 1
bear: 1
fuzzy: 1, 2, 3
had: 2
hair: 2
he: 4
no: 2
was: 1, 4
wasn't: 3
wuzzy: 1, 2, 3
```

Programming Project 12.4

Approval Voting

Design, test, and implement a program to handle approval voting. In *approval voting*, each voter specifies which candidates are acceptable. The winning candidate is the one who is voted to be acceptable on the most ballots. For example, suppose there are four candidates: Larry, Curly, Moe, and Gerry. The seven voters' ballots are as follows:

Voter	Ballot
1	Moe, Curly
2	Curly, Larry
3	Larry
4	Gerry, Larry, Moe
5	Larry, Moe, Gerry, Curly
6	Gerry, Moe
7	Curly, Moe

(continued on next page)

(continued from previous page)

The number of ballots on which each candidate was listed is as follows:

Candidate	Number of Ballots
Curly	4
Gerry	3
Larry	4
Moe	5

In this example, the winner is Moe. If there is a tie for most approvals, all of those tied candidates are considered winners—there may then be a run-off election to determine the final winner.

For a given file of ballots, the output—in the console window—should have:

1. the candidate(s) with the most votes, in alphabetical order of candidates' names;

2. the vote totals for all candidates, in alphabetical order of candidates' names;

3. the vote totals for all candidates, in decreasing order of votes; for candidates with equal vote total, the ordering should be alphabetical.

Let C represent the number of candidates, and V represent the number of voters. For the entire program, worstTime(C, V) is O(V log C). You may assume that C is (much) smaller than V.
Assume that p5.in1 contains the following:

```
Moe, Curly
Curly, Larry
Larry
Gerry, Larry, Moe
Larry, Moe, Gerry, Curly
Gerry, Moe
Curly, Moe
```

Assume that p5.in2 contains the following:

```
Karen
Tara, Courtney
Courtney, Tara
```

System Test 1 (Input in boldface):

```
Please enter the input file path: p5.in1
```

The winner is Moe, with 5 votes.

Here are the totals for all candidates, in alphabetical order:

Candidate	Number of Ballots
Curly	4
Gerry	3
Larry	4
Moe	5

Here are the vote totals for all candidates, in decreasing order of vote totals:

Candidate	Number of Ballots
Moe	5
Curly	4
Larry	4
Gerry	3

System Test 2 (Input in boldface):

```
Please enter the input file path: p5.in2
```

The winners are Courtney and Tara, with 2 votes.
Here are the totals for all candidates, in alphabetical order:

Candidate	Number of Ballots
Courtney	2
Karen	1
Tara	2

Here are the vote totals for all candidates, in decreasing order of vote totals:

Candidate	Number of Ballots
Courtney	2
Tara	2
Karen	1

Keep re-prompting until a legal file path is entered.

The input file may have millions of ballots, so do not re-read the input.

Programming Project 12.5

An Integrated Web Browser and Search Engine, Part 4

In this part of the project, you will add functionality to the Search button. Assume the file search.in1 consists of file names (for web pages), one per line. Each time the end-user clicks on the Search button, the output window is cleared and then a prompt is printed in the Output window to request that a search string be entered in the Input window followed by a pressing of the Enter key.

For each file name in search.in1, the web page corresponding to that file is then searched for the individual words in the search string. Then the link is printed in the Output window, along with the relevance count: the sum of the word frequencies of each word in the search string. For example, suppose the search string is "neural network", the file name is "browser.in6", and that web page has

> A network is a network, neural or not. If every neural network were
> combined, that would be a large neural network for networking.

The output corresponding to that web page would be

```
browser.in6  7
```

because "neural" appears 3 times on the web page, and "network" appears 4 times ("networking" does not count). The end user may now click on browser.in6 to display that page.

Start with your definition of the two specified methods in your `Filter` class from Part 2 of this project. Use those methods to get each word in the web page—excluding an expanded file of common words, and so on—and determine the frequency of each such word on that web page. Then, for each word in the search string, add up the frequencies.

The only class you will modify is your listener class. Here is the expanded file of common words, which will be stored in the file common.in1:

a

all

an

and

be

but

did

down

for

if

in

is

not

or

that

the

through

to

were

where

would

For each of the n words in the web page, the worstTime(n) for incrementing that word's frequency must be logarithmic in n. Also, for each word in the search string, calculating its frequency in the web page must also take logarithmic-in-n time, even in the worst case.

For testing, assume search.in1 contains just a few files, for example,

browser.in6
browser.in7
browser.in8

Here are the contents of those files:
browser.in6:

```
A network is a network, neural or not. If every neural network were
combined,
that would be a
large neural network for networking.
```

browser.in7:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns<a href=browser.in2>browser2</a> measureless to man
Down to a sunless sea.
```

browser.in8:

```
In Xanadu did Kubla Khan
A stately network pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns <a href=browser.in2>browser2</a>measureless to man
Down to a sunless neural.
```

(continued on next page)

(continued from previous page)

System Test 1(user input in boldface):

```
Please enter a search string in the input line and then press the Enter key.
neural network

Here are the files and relevance frequencies
browser.in6  7
browser.in7  0
browser.in8  2

Please enter a search string in the input line and then press the Enter key.
neural

Here are the files and relevance frequencies
browser.in6  3
browser.in7  0
browser.in8  1
```

If the end-user now clicks on the Back button, the input prompt should appear in the output window. If the end-user clicks on the Back button again, the search page for "neural network" should re-appear.

System Test 2:

```
Please enter a search string in the input line and then press the Enter key.
network

Here are the files and relevance frequencies:
browser.in6  4
browser.in7  0
browser.in8  1
```

NOTE: The Search button should be green as soon as the GUI window is opened, and only one search string can be entered for each press of the Search button.

Priority Queues

CHAPTER 13

In this chapter, we examine the priority queue data type. A *priority queue* is a collection in which only the element with highest priority can be removed, according to some method for comparing elements. This restriction allows an implementation—the Java Collections Framework's `PriorityQueue` class—with an `add` method whose average time is constant. The `PriorityQueue` class can be enhanced by including `heapSort`, a fast sort method for which worstSpace(n) is constant. The chapter concludes by using a priority queue to generate a Huffman Tree—a necessary component of a popular data-compression technique called Huffman compression.

CHAPTER OBJECTIVES

1. Understand the defining property of a priority queue, and how the Java Collections Framework's `PriorityQueue` class violates that property.
2. Be able to perform the heap operations of `siftUp` and `siftDown`.
3. Compare Heap Sort to Merge Sort with respect to time and space requirements.
4. Examine the Huffman data-compression algorithm.
5. Determine the characteristic of a greedy algorithm.

13.1 Introduction

A variation of the queue, the priority queue is a commonplace structure. The basic idea is that we have elements waiting in line for service, as with a queue. But removals are not strictly on a first-in-first-out basis. For example, patients in an emergency room are treated according to the severity of their injuries, not according to when they arrived. Similarly, in air-traffic control, when there is a queue of planes waiting to land, the controller can move a plane to the front of the queue if the plane is low on fuel or has a sick passenger.

A shared printer in a network is another example of a resource suited for a priority queue. Normally, jobs are printed based on arrival time, but while one job is printing, several others may enter the service queue. Highest priority could be given to the job with the fewest pages to print. This would optimize the average time for job completion. The same idea of prioritized service can be applied to any shared resource: a central processing unit, a family car, the courses offered next semester, and so on.

Here is the definition:

A ***priority queue*** is collection in which removal is of the highest-priority element in the collection, according to some method for comparing elements.

For example, if the elements are of type (reference to) `Integer` and comparisons use the "natural" ordering, then the highest-priority element is the one whose corresponding **`int`** has the *smallest* value in the priority queue. But if elements are of type `Integer` and the comparisons use the reverse of the natural ordering, then the highest-priority element is the one whose corresponding **`int`** has the largest value in the priority queue. By default, the smallest-valued element has highest priority.

This definition says nothing about insertions. In the `PriorityQueue` class, the method `add (E element)` takes only constant time on average, but linear-in-*n* time in the worst case. What underlying structure do these times suggest?

You might wonder what happens if two or more elements are tied for highest priority. In the interest of fairness, the tie should be broken in favor of the element that has been in the priority queue for the longest time. This appeal to fairness is not part of the definition, and is not incorporated into the `PriorityQueue` class. Lab 21 provides a solution to this problem.

Most of this chapter is devoted to an important application of priority queues: Huffman encoding. Also, Chapter 15 has two widely used priority-queue applications: Prim's minimum-spanning-tree algorithm and Dijkstra's shortest-path algorithm. For a lively discussion of the versatility of priority queues, see Dale [1990].

13.2 The `PriorityQueue` Class

The `PriorityQueue` class, which complements the `Queue` interface, is in the package `java.util`. That fact has the significant benefit that the `PriorityQueue` class is available to you whenever you program in Java. But the `PriorityQueue` class also has a flaw: It allows methods—such as `remove (Object obj)`—that violate the definition of a priority queue. If you want a `PurePriorityQueue` class, you have the same two choices as were available for the `PureStack` and `PureQueue` classes in Chapter 8. You can extend the `PriorityQueue` class to a `PurePriorityQueue` class that throws `UnsupportedOperationException` for the removal of any element except the highest-priority element. Or you can create a minimalist `PurePriorityQueue` class that allows only a few methods: a couple of constructors, `isEmpty()`, `size()`, `add (E element)`, `remove()`, and `element()`.

The `PriorityQueue` class has some unusual features. For example, **`null`** values are not permitted. Why? Because some removal methods, such as `poll()`, and some access methods (such as `peek()`) return **`null`** if the priority queue is empty. If **`null`** values were allowed, there would be ambiguity if `poll()` returned **`null`**: That value could signify an empty queue, or could simply be the value of the highest priority element.

The creator of a `PriorityQueue` object can order the elements "naturally" (with the element class's `Comparable` interface) or else provide a `Comparator` class for comparisons.

Here are the specifications for two constructors and three other essential methods:

```
/**
 * Initializes a PriorityQueue object with the default initial capacity (11) that orders its
 * elements according to their natural ordering (using Comparable).
 */
public PriorityQueue()


/**
 * Initializes a PriorityQueue object with the specified initial capacity
 * that orders its elements according to the specified comparator.
 *
 * @param initialCapacity the initial capacity for this priority queue.
 * @param comparator the comparator used to order this priority queue.
 *        If null then the order depends on the elements' natural ordering.
 * @throws IllegalArgumentException if initialCapacity is less than 1
 */
 public PriorityQueue(int initialCapacity, Comparator<? super E> comparator)

/**
 *  Inserts a specified element into this PriorityQueue object.
 *  The worstTime(n) is O(n) and averageTime(n) is constant.
 *
 *  @param element -the element to be inserted into this PriorityQueue object.
 *  @return true
 *  @throws NullPointerException -if element is null.
 *  @throws ClassCastException if the specified element cannot be compared
 *                   with elements currently in the priority queue according
 *                   to the priority queue's ordering.
 *
 */
public boolean add (E element)

/**
 * Retrieves and removes the highest priority element of this PriorityQueue object.
 * The worstTime(n) is O(log n).
 *
 * @return the highest priority element of this PriorityQueue object.
 * @throws NoSuchElementException if this queue is empty.
 */
public E remove()

/**
 * Retrieves, but does not remove, the highest priority element of this PriorityQueue
 * object.
 * The worstTime(n) is constant.
 *
 * @return the highest priority element of this PriorityQueue object.
 * @throws NoSuchElementException if this queue is empty.
 */
public E element()
```

13.3 Implementation Details of the PriorityQueue Class

In any instance of the PriorityQueue class, the elements are stored in a heap. A ***heap*** t is a complete binary tree such that either t is empty or

1. the root element is the smallest element in t, according to some method for comparing elements;
2. the left and right subtrees of t are heaps.

(The heap we have defined is called a "minHeap". In a "maxHeap", the root element is the largest element.)

Recall from Chapter 9 that a complete binary tree is full except, possibly, at the level farthest from the root, and at that farthest level, the leaves are as far to the left as possible. Figure 13.1 shows a heap

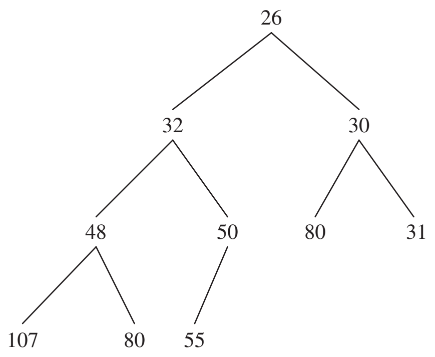


FIGURE 13.1 A heap with ten (reference to) `Integer` elements; the **`int`** values of the `Integer` elements are shown

of ten `Integer` elements with the "natural" ordering. Notice that a heap is *not* a binary search tree. For example, duplicates are allowed in a heap, unlike the prohibition against duplicates for binary search trees.

The ordering in a heap is top-down, but not left-to-right: the root element of each subtree is less than or equal to each of its children, but some left siblings may be less than their right siblings and some may be greater than or equal to. In Figure 13.1, for example, 48 is less than its right sibling, 50, but 107 is greater than its right sibling, 80. Can you find an element that is smaller than its parent's sibling?

A heap is a complete binary tree. We saw in Chapter 9 that a complete binary tree can be implemented with an array. Figure 13.2 shows the array version of Figure 13.1, with each index under its element. An iteration over a `PriorityQueue` object uses the index-ordering of the array, so the iteration would be 26, 32, 30, 48, 50, 80, 31, 107, 80, 55.

Recall from Chapter 9 that the random-access feature of arrays is convenient for processing a complete binary tree: Given the index of an element, that element's children can be accessed *in constant time*. For example, with the array as shown in Figure 13.2, the children of the element at index `i` are at indexes `(i << 1) + 1` and `(i << 1) + 2`. And the parent of the element at index `j` is at index `(j - 1) >> 1`.

26	32	30	48	50	80	30	107	80	55	. . .
0	1	2	3	4	5	6	7	8	9	

FIGURE 13.2 The array representation of the heap from Figure 13.1

As we will see shortly, the ability to quickly swap the values of a parent and its smaller-valued child makes a heap an efficient storage structure for the `PriorityQueue` class.

Here is program that creates and maintains two `PriorityQueue` objects in which the elements are of type `Student`, declared below. Each line of input—except for the sentinel—consists of a name and the corresponding grade point average (GPA). The highest-priority student is the one whose GPA is lowest. In the `Student` class, the `Comparable` interface is implemented by specifying increasing order of grade point averages. To order another heap by alphabetical order of student names, a `ByName` comparator class is defined.

```
import java.util.*;

public class PriorityQueueExample
{
    public static void main (String[ ] args)
    {
        new PriorityQueueExample().run();
    } // method main

    public void run()
    {
        final int DEFAULT_INITIAL_CAPACITY = 11;

        final String PROMPT1 = "Please enter student's name and GPA, or " ;

        final String PROMPT2 = " to quit: ";

        final String SENTINEL = "***";

        final String RESULTS1 = "\nHere are the student names and GPAs, " +
                                "in increasing order of GPAs:";

        final String RESULTS2 = "\nHere are the student names and GPAs, " +
                                "in alphabetical order of names:";

        PriorityQueue<Student> pq1 = new PriorityQueue<Student>(),
                               pq2 = new PriorityQueue<Student>
                                 (DEFAULT_INITIAL_CAPACITY, new ByName());

        Scanner sc = new Scanner (System.in);

        String line;

        while (true)
         {
             System.out.print (PROMPT1 + SENTINEL + PROMPT2);
             line = sc.nextLine();
             if (line.equals (SENTINEL))
                 break;
             pq1.add (new Student (line));
             pq2.add (new Student (line));
         } // while
         System.out.println (RESULTS1);
         while (!pq1.isEmpty())
             System.out.println (pq1.remove());

        System.out.println (RESULTS2);
         while (!pq2.isEmpty())
             System.out.println (pq2.remove());
     } // method run

} // class PriorityQueueExample

// in another file

import java.util.*;
public class Student implements Comparable<Student>
```

```
{
    protected String name;

    protected double gpa;


     /**
      *  Initializes this Student object from a specified String object.
      *
      *  @param s - the String object used to initialize this Student object.
      *
      */
     public Student (String s)
     {
         Scanner sc = new Scanner (s);

         name = sc.next();
         gpa = sc.nextDouble();
     } // constructor

     /**
      *  Compares this Student object to a specified Student object by
      *  grade point average.
      *
      *  @param otherStudent - the specified Student object.
      *
      *  @return a negative integer, 0, or a positive integer, depending
      *  on whether this Student object's grade point average is less than,
      *  equal to, or greater than otherStudent's grade point average.
      *
      */
     public int compareTo (Student otherStudent)
     {
         if (gpa < otherStudent.gpa)
             return -1;
         if (gpa > otherStudent.gpa)
             return 1;
                 return 0;

     } // method compareTo


    /**
     *  Returns a String representation of this Student object.
     *
     *  @return  a String representation of this Student object: name " " gpa
     *
     */
    public String toString()
    {
        return name + "  " + gpa;
    } // method toString
} // class Student
// in another file:
import java.util.*;

public class ByName implements Comparator<Student>
{
    public int compare (Student stu1, Student stu2)
    {
        String name1 = new Scanner (stu1.toString()).next(),
               name2 = new Scanner (stu2.toString()).next();

       return name1.compareTo (name2);
    } // method compare

} // class ByName
```

The Student class implements the Comparable interface with a compareTo method that returns −1, 0, or 1, depending on whether the calling Student object's grade point average is less than, equal to, or greater than another student's grade point average. So, as you would expect, the Student with the lowest grade point average is the highest-priority element. Suppose the input for the PriorityQueueExample program is as follows:

```
Soumya 3.4
Navdeep  3.5
Viet 3.5
***
```

Here is the output of the students as they are removed from pq1:

```
Soumya 3.4
Viet 3.5
Navdeep  3.5
```

Notice that Viet is printed before Navdeep even though they have the same grade point average, and Navdeep was input earlier than Viet. As mentioned earlier, the PriorityQueue class—like life—is unfair. In Section 13.3.1, we look at more details of the PriorityQueue class, which will help to explain how this unfairness comes about. Lab 21 will show you how to remedy the problem.

For pq2 in the PriorityQueueExample program above, the output of students would be

```
Navdeep  3.5
Soumya   3.4
Viet  3.5
```

13.3.1 Fields and Method Definitions in the PriorityQueue Class

The critical field in the PriorityQueue class will hold the elements:

```
private transient Object[ ] queue;
```

And—similar to the TreeMap class—we also have size, comparator, and modCount fields:

```
/**
 * The number of elements in the priority queue.
```

```
      */
     private int size = 0;

     /**
      * The comparator, or null if priority queue uses elements'
      * natural ordering.
      */
     private final Comparator<? super E> comparator;

     /**
      * The number of times this priority queue has been
      * structurally modified.  See AbstractList class and Appendix 1 for gory details.
      */
     private transient int modCount = 0;
```

The following constructor definition creates a priority queue with a specified initial capacity and an ordering supplied by a specified comparator:

```
     /**
     * Initializes a PriorityQueue object with the specified initial capacity
     * that orders its elements according to the specified comparator.
     *
     * @param initialCapacity the initial capacity for this priority queue.
     * @param comparator the comparator used to order this priority queue.
     *        If null then the order depends on the elements' natural ordering.
     * @throws IllegalArgumentException if initialCapacity is less than 1
     */
     public PriorityQueue (int initialCapacity,  Comparator<? super E> comparator)
     {
              if (initialCapacity < 1)
                      throw new IllegalArgumentException();
              queue = new Object[initialCapacity];
              this.comparator = comparator;
     } // constructor
```

The definition of the default constructor follows easily:

```
     /**
      * Initializes a PriorityQueue object with the default initial capacity (11)
      * that orders its elements according to their natural ordering (using Comparable).
      */
     public PriorityQueue( )
     {
          this (DEFAULT_INITIAL_CAPACITY, null);
     } // default constructor
```

Now let's define the `add (E element)`, `element()`, and `remove()` methods. The `add (E element)` method expands that array if it is full, and calls an auxiliary method, `siftUp`, to insert the element and restore the heap property. Here are the specifications and definition:

```
     /**
      * Inserts the specified element into this priority queue.
      * The worstTime(n) is O(n) and averageTime(n) is constant.
      *
      * @return {@code true} (as specified by {@link Collection#add})
      * @throws ClassCastException if the specified element cannot be
      *         compared with elements currently in this priority queue
      *         according to the priority queue's ordering
      * @throws NullPointerException if the specified element is null
      */
     public boolean add(E e) {
         if (e == null)
             throw new NullPointerException();
         modCount++;
         int i = size;
         if (i >= queue.length)
             grow(i + 1);
         size = i + 1;
         if (i == 0)
             queue[0] = e;
         else
             siftUp(i, e);
         return true;
     }
```

For example, Figure 13.3 shows what the heap in Figure 13.1 will look like just before `siftUp (10, new Integer (28))` is called.

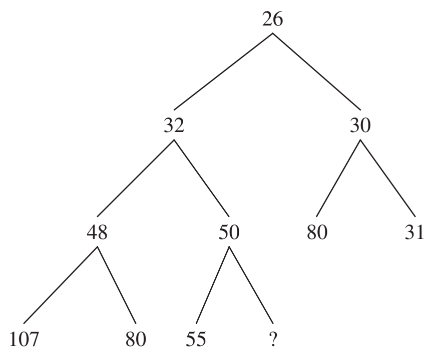


FIGURE 13.3 The heap from Figure 13.1 just before 28 is inserted. The size is 10.

13.3.1.1 The `siftUp` Method

The `siftUp` method determines where an element is to be inserted, and performs the insertion while preserving the heap property. Here is the method specification and definition for `siftUp`:

```
    /**
     * Inserts item x at position k, maintaining heap invariant by
     * promoting x up the tree until it is greater than or equal to
     * its parent, or is the root.
     * The worstTime(n) is O(log n) and averageTime(n) is constant.
     *
     * To simplify and speed up coercions and comparisons, the
     * Comparable and Comparator versions are separated into different
     * methods that are otherwise identical. (Similarly for siftDown.)
     *
     * @param k the position to fill
     * @param x the item to insert
     */
    private void siftUp(int k, E x) {
        if (comparator != null)
           siftUpUsingComparator(k, x);
        else
           siftUpComparable(k, x);
    }
```

And here is the specification and definition for `siftUpComparable` (the definition for `siftUsing-Comparator` differs only in the use of the `compare` method instead of `compareTo`):

```
    /**
     * Inserts item x at position k, maintaining heap invariant by
     * promoting x up the tree until it is greater than or equal to
     * (according to the elements' implementation of the Comparable
     * interface) its parent, or is the root.
     * The worstTime(n) is O(log n) and averageTime(n) is constant.
     *
     * @param k the position to fill
     * @param x the item to insert
     */
     private void siftUpComparable(int k, E x) {
          Comparable<? super E> key = (Comparable<? super E>) x;
          while (k > 0) {
              int parent = (k - 1) >>> 1;
              Object e = queue[parent];
              if (key.compareTo((E) e) >= 0)
                  break;
              queue[k] = e;
              k = parent;
          }
          queue[k] = key;
     }
```

The `siftUpComparable (int k, E x)` method restores the heap property by repeatedly replacing `queue [k]` with its parent and dividing `k` by 2 (with a right shift of 1 to accomplish dividing by 2 faster) until `x` is greater than or equal to the new parent. Then the **`while`** loop is exited and `x` is stored at `queue[k]`. Figure 13.4 shows the result of a call to `siftUpComparable (10, new Integer (28))` on the complete binary tree in Figure 13.3:

When called by the `add (E element)` method, the `siftUp` method starts at `queue [size - 1]` as the child and replaces that child with its parent until the parent is greater than or equal to the element to

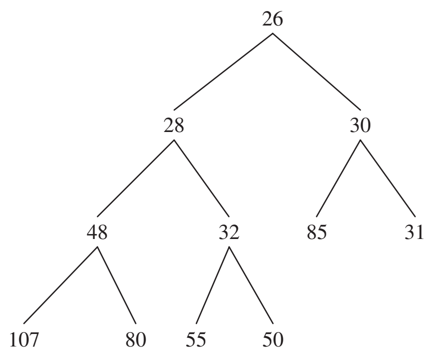
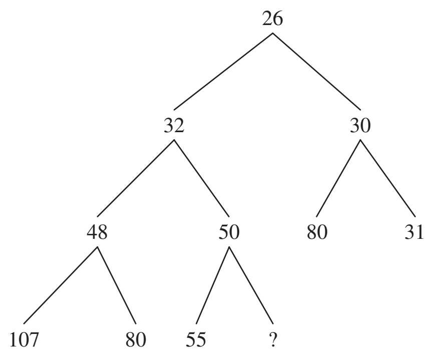


FIGURE 13.4 The heap formed by invoking `siftUp (10, new Integer(28))` on the complete binary tree in Figure 13.3

be inserted. Then the element is inserted at that parent index. For example, let's begin with the complete binary tree in Figure 13.3, repeated here, with 28 to be inserted:



The child index is 10. Because 28 is less than 50, we replace queue[10] with 50, and set the child index to 4(= (10 – 1) >>> 1). See Figure 13.5.

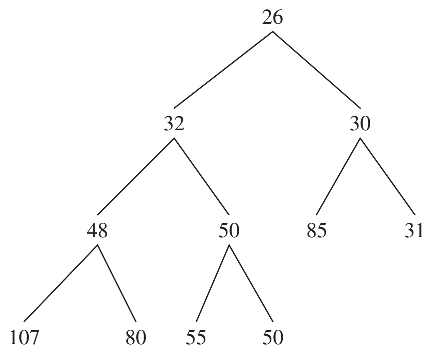


FIGURE 13.5 The heap of Figure 13.3 after replacing `queue[10]` with `queue[4]`

We now set the parent index to 1(= (4 − 1) >>> 1), and compare 28 with the parent's value, namely 32. Because 28 is less than 32, we replace `queue[4]` with `queue[1]`, and set the parent index to 0(= (1 − 1) >>> 1). Because 28 is <= `queue[0]`, 28 is inserted at queue[1] and that gives us the heap shown in Figure 13.6.

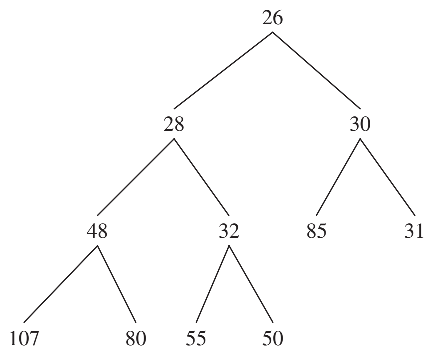


FIGURE 13.6 The heap formed when 28 is added to the heap in Figure 13.1

The key to the efficiency of the `siftUp` method is that a parent's index is readily computable from either of its children's indexes: if `k` contains a child's index, then its parent's index must be

```
(k -1) / 2
```

or, to be slightly more efficient

```
(k -1) >>> 1 // >> performs a right shift of 1 bit on the binary representation of k - 1
```

And because `queue` is an array object, the element at that parent index can be accessed or modified in constant time.

In the worst case for the `siftUp` method, the element inserted will have a smaller value than any of its ancestors, so the number of loop iterations will be, approximately, the height of the heap. As we saw in Chapter 9, the height of a complete binary tree is logarithmic in n. So worstTime(n) is logarithmic in n, which satisfies the worst-time estimate from the specification of `siftUp`.

In the average case, about half of the elements in the heap will have a smaller value than the element inserted, and about half will have a larger value. But heaps are very bushy: at least half of the elements are leaves. And because of the heap properties, most of the larger-valued elements will be at or near the leaf level. In fact, the average number of loop iterations is less than 3 (see Schaffer [1993]), which satisfies the specification that averageTime(n) be constant. Programming Exercise 13.3 outlines a run-time experiment to support this claim.

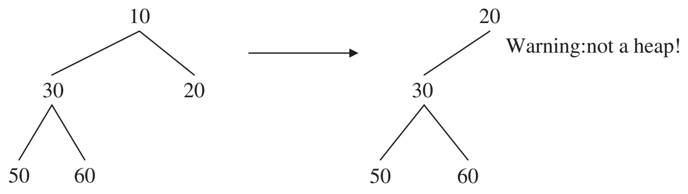
Now we can analyze the `add (E element)` method. The worst case occurs when the array `queue` is full. Then a new, larger array is constructed and the old array is copied over. So worstTime(n) is linear in n. But this doubling occurs infrequently—once in every n insertions—so averageTime(n) is constant, just as for `siftUp`.

13.3.1.2 The `element()` and `remove()` Methods

The `element()` method simply returns the element at index 0, and so its worstTime(n) is constant.

```
/**
 *  Returns the smallest-valued element in this PriorityQueue object.
 *
 *  @return the smallest-valued element in this PriorityQueue object.
 *
 *  @throws NoSuchElementException -if this PriorityQueue object is empty.
 *
 */
public E element()
{
    if (size == 0)
         throw new NoSuchElementException();
    return (E) queue [0];
} // method element()
```

The `remove()` method's main task is to delete the root element. But simply doing this would leave a hole where the root used to be. So, just as we did when we removed a `BinarySearchTree` object's element that had two children, we replace the removed element with some other element in the tree. Which one? The obvious answer is the smaller of its children. And ultimately, that must happen in order to preserve the heap properties. But if we start by replacing the root element with its smaller child, we could open up another hole, as is shown in the following picture:



When 10 is replaced with 20, its smaller child, the resulting tree is not a heap because it is not a complete binary tree.

A few minutes reflection should convince you that, to preserve the completeness of the binary tree, there is only one possible replacement for the root element: the last element in the heap—at index `size -1`. Of course, we will then have a lot of adjustments to make; after all, we have taken one of the largest elements and put it at the top of the heap. These adjustments are handled in a `siftDown` method that starts at the new root and moves down to the leaf level. So the `remove()` method decrements `size` by 1, saves the element at index 0 in `result`, saves the element (no longer in the heap) at index `size` in `x`, calls `siftDown(0, x)` to place `x` where it now belongs, and returns `result`.

Here is the method definition:

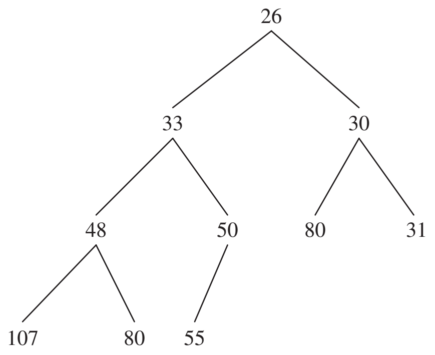
```
/**
 *  Removes the smallest-valued element from this PriorityQueue object.
 *  The worstTime(n) is O(log n).
 *
 *  @return the element removed.
 *
 *  @throws NoSuchElementException - if this PriorityQueue object is empty.
 *
 */
 public E remove ()
 {
      if (size == 0)
           throw new NoSuchElementException();
      int s = --size;
```

```
            modCount++;
            E result = (E) queue[0];
            E x = (E) queue[s];
            queue[s] = null;    // to prevent memory leak
            if (s != 0)
                 siftDown(0, x);
            return result;
      } // method remove
```

The definition of the `siftDown (int k, E x)` method is developed in Section 13.3.1.3.

13.3.1.3 The siftDown Method

To see how to define `siftDown`, let's start with the heap from Figure 13.1, repeated below:



If we now call `remove()`, then 26 is saved in `result` and 55 is saved in `x`, just before the call to `siftDown`. See Figure 13.7.

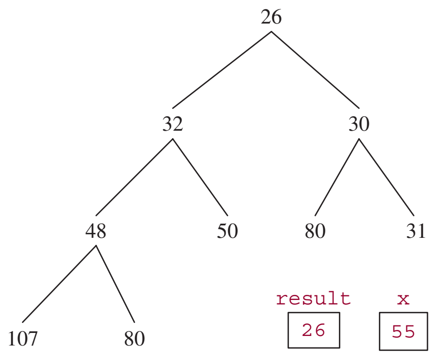


FIGURE 13.7 A heap from which 26 is to be removed

Notice that since we started with a heap, the heap property guarantees that the root's left and right subtrees are still heaps. So `siftDown's` task is to maintain the heap property while putting `x` where it belongs.

In contrast to `siftUp`, the `siftDown` method works its way down the tree, starting at the index supplied as the argument, in this case, 0. Each parent is replaced with its smaller-valued child until, eventually, either x's value is less than or equal to the smaller-valued child (and then is stored at the parent index) or the smaller-valued child is a leaf (and then x is stored at that child index). For example, in the tree of Figure 13.7, 30 is smaller than 32, so 30 becomes the root element, and we have the tree shown in Figure 13.8.

We still need one more iteration because 55 is greater than the smaller of the children (namely, 80 and 31 at indexes 5 and 6. So the element at index 6 (namely, 31) replaces the element at index 2 (namely, 30). That smaller-valued child was a leaf, so 55 is inserted at that leaf index and we end up with the heap shown in Figure 13.9.

If k contains the parent index, then k << 1 is the index of the left child and (k << 1) + 1 is the index of the right child (if the parent has a right child). And because queue is an array object, the elements at those indexes can be accessed or modified in constant time.

Here is the definition of siftDown:

```
private void siftDown(int k, E x)
{
    if (comparator != null)
        siftDownUsingComparator(k, x);
```

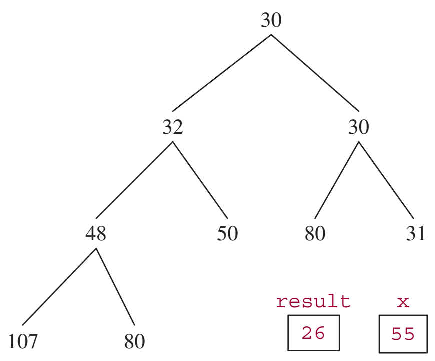


FIGURE 13.8 The heap from Figure 13.7 after the smaller-valued of the root's children is replaces the root. The element 55 has not yet been removed

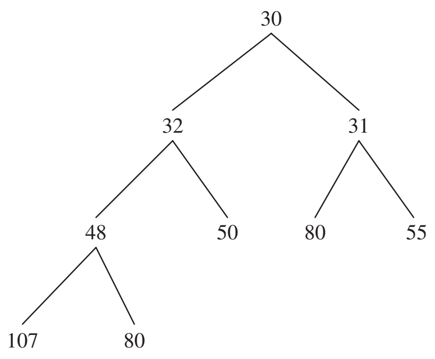


FIGURE 13.9 The heap formed from the heap in Figure 13.8 by replacing the element at index 2 with 31 and then inserting 55 at index 6

```
        else
            siftDownComparable(k, x);
    }
```

And here is `siftDownComparable` (which differs from `siftDownUsingComparator` in that the elements' class implements the `Comparable` interface):

```
    /**
     *  Maintains the heap properties in this PriorityQueue object while, starting at a
     *  specified index, inserts a specified element where it belongs.
     *  The worstTime(n) is O(log n).
     *
     *  @param k -the specified position where the restoration of the heap
     *                     will begin.
     *  @param x -the specified element to be inserted.
     *
     */
    private void siftDownComparable(int k, E x) {
         Comparable<? super E> key = (Comparable<? super E>)x;
         int half = size >>> 1;        // loop while a non-leaf
         while (k < half) {
             int child = (k << 1) + 1; // assume left child is least
             Object c = queue[child];
             int right = child + 1;
             if (right < size &&
                 ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)
                    c = queue[child = right];
             if (key.compareTo((E) c) <= 0)
                    break;
             queue[k] = c;
             k = child;
         }
         queue[k] = key;
    }
```

In the worst case for `siftDownComparable`, the loop will continue until a leaf replaces its parent. Since the height of a heap is logarithmic in n, worstTime(n) is logarithmic in n. Because a heap is a complete binary tree, the average number of loop iterations will be nearly $\log_2 n$, so averageTime(n) is also logarithmic in n.

In Lab 21, you will create a descendant, `FairPQ`, of the `PriorityQueue` class. As its name suggests, the `FairPQ` class resolves ties for highest-priority element in favor of seniority. For elements with equal priority, the highest-priority is chosen as the element that has been on the heap for the longest time.

You are now ready for Lab 21: Incorporating Fairness in Priority Queues

Section 13.4 shows how to modify the `PriorityQueue` class to obtain the `heapSort` method. Actually, the `PriorityQueue` class was created many years after the invention of the `heapSort` method, which was originally based on the idea of using an array to store a heap.

13.4 The heapSort Method

The heapSort method was invented by J.W.J. Williams (see Williams [1964]). Here is the method specification:

```
/**
 *  Sorts a specified array into the order given by the comparator in the
 *  constructor (natural, that is, Comparable order if default constructor;
 *  unnatural, that is, Comparator order if constructor specifies non-null
 *  comparator).
 *  The worstTime(n) is O(n log n) and worstSpace(n) is constant.
 *
 *  @param a -the array object to be sorted.
 *
 *  @throws NullPointerException - if a is null.
 *
 */
public void heapSort (Object[ ] a)
```

Assume, for now, that this method is in the PriorityQueue class. Then the following is a test of that method:

```
public void testSample()
{

     Integer [ ] expected ={12,  16,  17,  32,  33,  40,  43,  44,  46,  46,
                           50,  55,  59,  61,  75,  80,  80,  81,  87,  95};
     Integer [ ] actual = {59,  46,  32,  80,  46,  55,  50,  43,  44,  81,
                           12,  95,  17,  80,  75,  33,  40,  61,  16,  87};
     new PriorityQueue().heapSort (actual);
     assertArrayEquals (expected, actual);
} // method testSample
```

Here is the basic idea of the heapSort method: We first initialize the queue and size fields:

```
queue = a;
int length = queue.length;
size = length;
```

We then ***heapify*** the array queue, that is, convert the array into a heap. After queue has become a heap, we sort queue into reverse order. **Important:** After each iteration of this loop, the heap will consist of the elements of queue from index 0 through index size—1, not through index queue.length-1. After the first sift down, the smallest element (the one with highest priority) will be at index queue.length - 1, and the array from indexes 0 through queue.length - 2 will constitute a heap. After the second sift down, the second smallest element will be at index queue.length - 2, and the array from indexes 0 through queue.length - 3 will constitute a heap. After all of the swaps and fix downs, the elements in the array queue will be in reverse order, so by reversing the order of the elements, the array will be in order.

For an example, let's start with an array of (references to Integer objects with the following) int values:

59 46 32 80 46 55 87 43 44 81 95 12 17 80 75 33 40 61 16 50

The value 59 is stored at index 0, the value 46 at index 1, and so on. Figure 13.10 (next page) shows the field queue viewed as a complete binary tree.

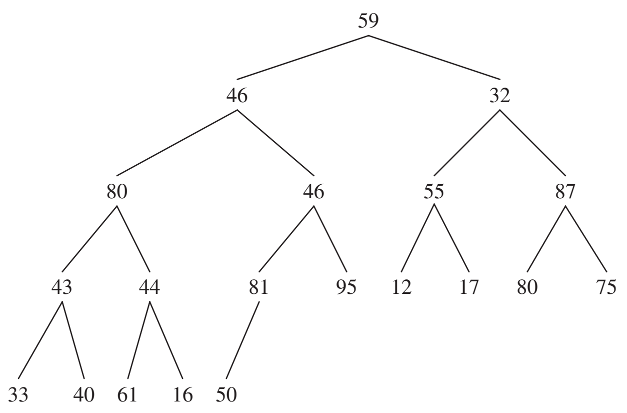


FIGURE 13.10 The array of 20 **int**s viewed as a complete binary tree

Of course, `queue` is not yet a heap. But we can accomplish that subgoal by looping as `i` goes from `size - 1` down to 0. At the start of each iteration, the complete binary tree rooted at index `i` is a heap except, possibly, at index `i` itself. The call to `siftDown (i, queue [i])` method in the `PriorityQueue` class converts the complete binary tree rooted at index `i` into a heap. So, basically, all we need to do is to repeatedly call `siftDown (i, queue [i])` to create the heap from the bottom up.

We can reduce the number of loop iterations by half by taking advantage of the fact that a leaf is automatically a heap, and no call to `siftDown` is needed. So the loop will start at the last non-leaf index. For the complete binary tree rooted at that index, its left and right subtrees will be heaps, and we can call `siftDown` with that index as argument. In the above example, the last non-leaf index is 9, that is, 20/2 – 1. Since `queue [19]` is less than `queue [9]`—that is, since 50 is less than 81—the effect of the first iteration is that those two elements are swapped, giving the complete binary tree in Figure 13.11.

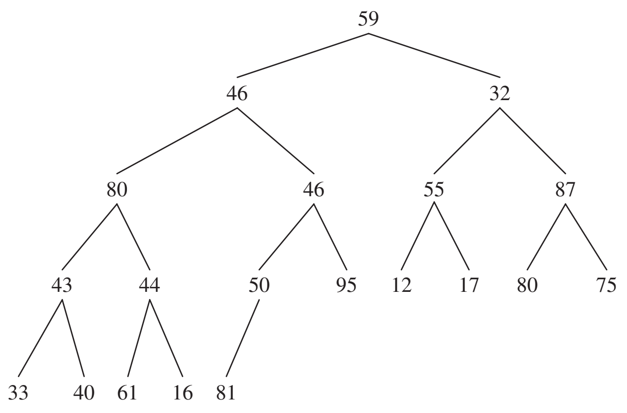
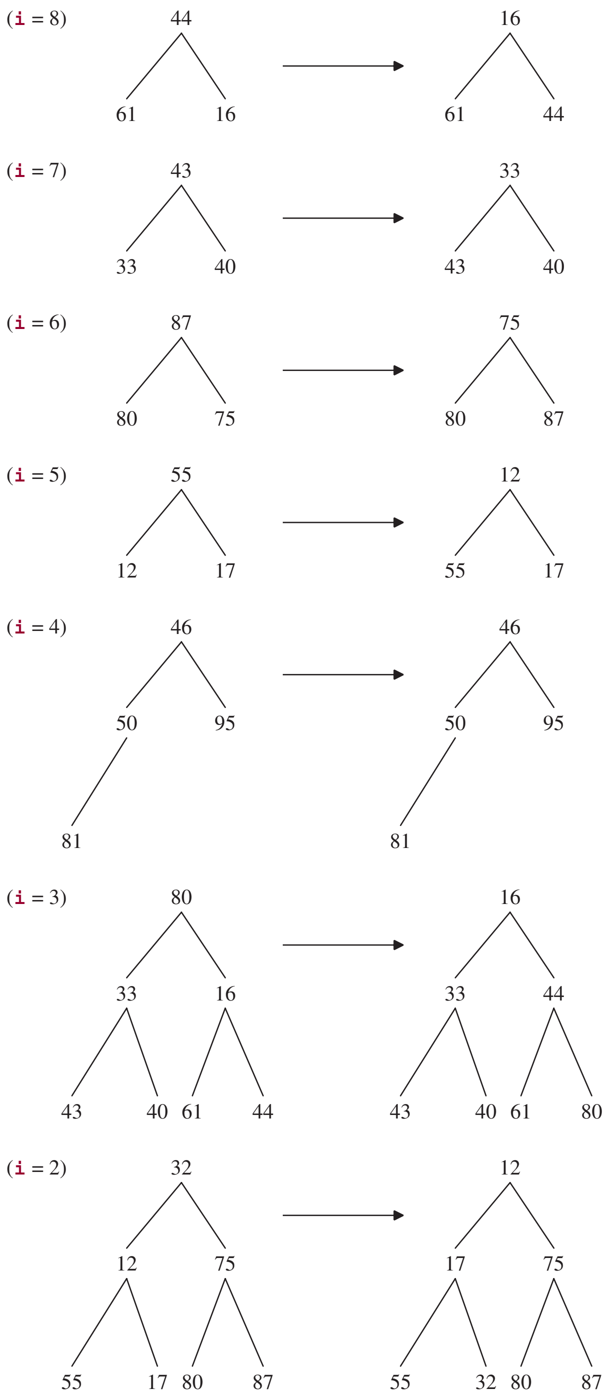
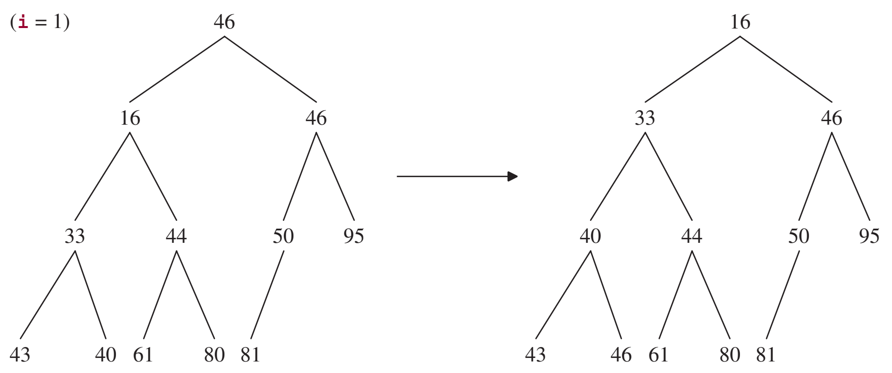


FIGURE 13.11 The complete binary tree of Figure 13.10 after swapping 50 with 81

During the remaining 9 loop iterations (at indexes 8 through 0), we wind our way back up the tree. For the `i`th iteration, `siftDown (i, queue [i])` is applied to the complete binary tree rooted at index `i`, with the left and right subtrees being heaps. The effects of the calls to `siftDown (i, queue [i])` are as follows:


(i = 8)
44
61
16
16
61
44
(i = 7)
43
33
40
33
43
40
(i = 6)
87
80
75
75
80
87
(i = 5)
55
12
17
12
55
17
(i = 4)
46
50
95
81
46
50
95
81
(i = 3)
80
33
16
43
40
61
44
16
33
44
43
40
61
80
(i = 2)
32
12
75
55
17
80
87
12
17
75
55
32
80
87



After the execution of `siftDown (i, queue [i])` at index 0 in the final iteration, we get the heap shown in Figure 13.12.

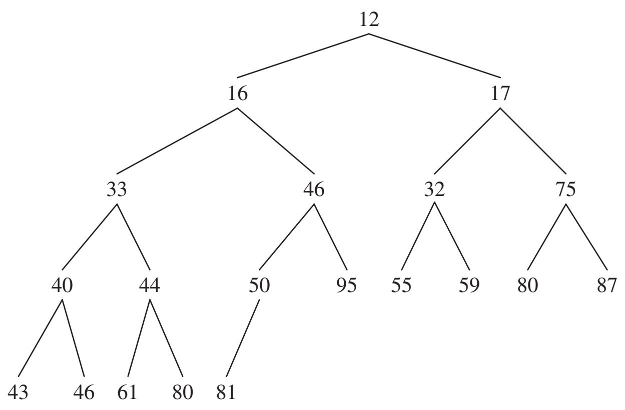


FIGURE 13.12 The effect of making a heap from the complete binary tree in Figure 13.10

We now sort the array into reverse order. To accomplish this, we have another loop, this one with `i` going from index 0 to index `queue.length - 1`. During each loop iteration, we save the element at `queue [-size]` in `x`, store `queue [0]` in `queue [size]` and then call `siftDown (0, x)`. For example, if `i` = 0 and `size` = 20, the element at `queue [19]` is saved in `x`, the smallest element (at index 0) is stored at `queue [19]`, and `siftDown (0, x)` is called. The complete binary tree from index 0 through index 18 is a complete binary tree except at index 0, so the call to `siftDown (0, x)` will restore heapity to this subarray, without affecting the smallest element (now at index 19). The result of 20 of these loop iterations is to put the 20 elements in descending order in the array. Figure 13.13 shows this array in the form of a complete binary tree.

When we treat the complete binary tree in Figure 13.13 as an array, and reverse the elements in this array, the elements are in ascending order:

12 16 17 32 33 40 43 44 46 46 50 55 59 61 75 80 80 81 87 95

Here is the `heapSort` method:

```
public void heapSort (Object[ ] a)
{
    queue = a;
```

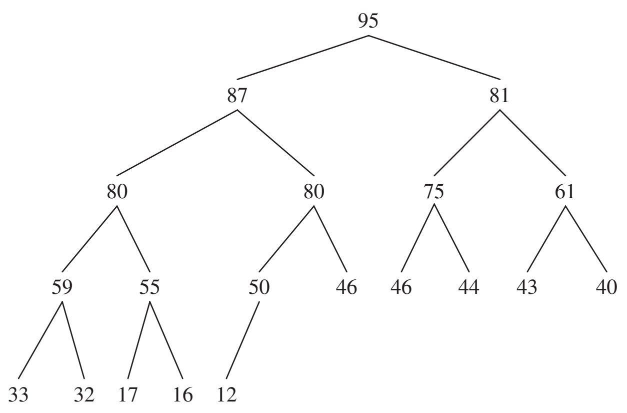


FIGURE 13.13 The complete binary tree resulting from the heap of Figure 13.12 after the 20 swaps and sift downs

```
        int length = queue.length;
        size = length;

       // Convert queue into a heap:
       for (int i = (size >> 1) - 1; i >= 0; i--)
            siftDown (i, (E)queue [i]);

       // Sort queue into reverse order:
       E x;
       for (int i = 0; i < length; i++)
       {
           x = (E)queue [--size];
           queue [size] = queue [0];
           siftDown (0, x);
       } // sort queue into reverse order

       // Reverse queue:
       for (int i = 0; i < length / 2; i++)
       {
           x = (E)queue [i];
           queue [i] = queue [length - i - 1];
           queue [length - i - 1] = x;
       } // reverse queue
    } // method heapSort
```

We earlier assumed that the `heapSort` method was part of the `PriorityQueue` class. Currently, at least, that is not the case. So how can we test and perform run-time experiments on that method? Surely, we cannot modify the `PriorityQueue` class in `java.util`. And we cannot make `heapSort` a method in an extension of the `PriorityQueue` class because the `queue` and `size` fields in that class have **`private`** visibility. What we do is copy the `PriorityQueue` class into a local directory, replace

```
    package java.util;
```

with

```
    import java.util.*;
```

And add the heapSort method to this copy of the PriorityQueue class. The method can then be tested and experimented with.

13.4.1 Analysis of heapSort

As with mergeSort in Chapter 11, we first estimate worstTime(n) for heapSort. The worst case occurs when the elements in the array argument are in reverse order. We'll look at the three explicit loops in heapSort separately:

1. To convert queue to a heap, there is a loop with $n/2 - 1$ iterations and with i as the loop-control variable, and in each iteration, siftDown (i, queue [i]) is called. In the worst case, siftDown (i, queue [i]) requires $\log_2(n)$ iterations, so the total number of iterations in converting a to a heap is $O(n \log n)$. In fact, it can be shown (see Concept Exercise 13.12) that this total number of iterations is linear in n.
2. For each of the n calls to siftDown with 0 as the first argument, worstTime(n) is $O(\log n)$. So the total number of iterations for this loop is $O(n \log n)$ in the worst case.
3. The reversing loop is executed $n/2$ times.

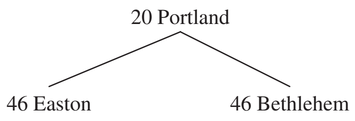
The total number of iterations in the worst case is $O(n) + O(n \log n) + O(n)$. That is, worstTime($n$) is $O(n \log n)$ and so, by Sorting Fact 1 from Section 11.4, worstTime(n) is linear-logarithmic in n. Therefore, by Sorting Fact 3, averageTime(n) is also linear-logarithmic in n.

The space requirements are meager: a few variables. There is no need to make a copy of the elements to be sorted. Such a sort is called an ***in-place*** sort.

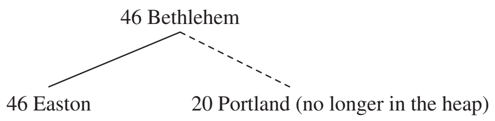
That heapSort is *not* stable can be seen if we start with the following array object of quality-of-life scores and cities ordered by increasing quality-of-life scores:

20	Portland
46	Easton
46	Bethlehem

Converting this array object to a heap requires no work at all because the heap property is already satisfied:



In the second loop of heapSort, Portland and Bethlehem are swapped and siftDown is called with 0 as the first argument:



Notice that, in the call to `siftDown`, (46 Bethlehem) is not swapped with (46 Easton) because its child (46 Easton) is *not less than* (46 Bethlehem). After two more iterations of the second loop and a reversal of the elements of the array, we have

(20 Portland) (46 Bethlehem) (46 Easton)

The positions of (46 Bethlehem) and (46 Easton) have flipped from the original array object, so the `heapSort` method is not stable.

13.5 Application: Huffman Codes

Suppose we have a large file of information. It would be advantageous if we could save space by compressing the file without losing any of the information. Even more valuable, the time to transmit the information might be significantly reduced if we could send the compressed version instead of the original.

Let's consider how we might encode a message file so that the encoded file has smaller size—that is, fewer bits—than the message file. For the sake of simplicity and specificity, assume the message file M contains 100,000 characters, and each character is either 'a', 'b', 'c', 'd', 'e', 'f', or 'g'. Since there are seven characters, we can encode each character uniquely with ceil $(\log_2 7)$ bits,[1] which is 3 bits. For example, we could use 000 for 'a', 001 for 'b', 010 for 'c', and so on. A word such as "cad" would be encoded as 010000011. Then the encoded file E need take up only 300,000 bits, plus an extra few bits for the encoding itself: 'a' = 000, and so on.

We can save space by reducing the number of bits for some characters. For example, we could use the following encoding:

a = 0

b = 1

c = 00

d = 01

e = 10

f = 11

g = 000

This would reduce the size of the encoded file E by about one-third (unless the character 'g' occurred very frequently). But this encoding leads to ambiguities. For example, the bit sequence 001 could be interpreted as "ad" or as "cb" or as "aab", depending on whether we grouped the first two bits together or the last two bits together, or treated each bit individually.

The reason the above encoding scheme is ambiguous is that some of the encodings are prefixes of other encodings. For example, 0 is a prefix of 00, so it is impossible to determine whether 00 should be interpreted as "aa" or "c". We can avoid ambiguities by requiring that the encoding be ***prefix-free***, that is, no encoding of a character can be a prefix of any other character's encoding.

[1]Recall that ceil(x) returns the smallest integer greater than or equal to x.

One way to guarantee prefix-free bit encodings is to create a binary tree in which a left branch is interpreted as a 0 and a right branch is interpreted as a 1. If each encoded character is a *leaf* in the tree, then the encoding for that character could not be the prefix of any other character's encoding. In other words, the path to each character provides a prefix-free encoding. For example, Figure 13.14 has a binary tree that illustrates a prefix-free encoding of the characters 'a' through 'g'.

To get the encoding for a character, start with an empty encoding at the root of the binary tree, and continue until the leaf to be encoded is reached. Within the loop, append 0 to the encoding when turning left and append 1 to the encoding when turning right. For example, 'b' is encoded as 01 and 'f' is encoded as 1110. Because each encoded character is a leaf, the encoding is prefix-free and therefore unambiguous. But it is not certain that this will save space or transmission time. It all depends on the frequency of each character. Since three of the encodings take up two bits and four encodings take up four bits, this encoding scheme may actually take up more space than the simple, three-bits-per-character encoding introduced earlier.

This suggests that if we start by determining the frequency of each character and then make up the encoding tree based on those frequencies, we may be able to save a considerable amount of space. The idea of using character frequencies to determine the encoding is the basis for ***Huffman encoding*** (Huffman [1952]). Huffman encoding is a prefix-free encoding strategy that is guaranteed to be optimal—among prefix-free encodings. Huffman encoding is the basis for the Unix `compress` utility, and also part of the JPEG (Joint Photographic Experts Group) encoding process.

We begin by calculating the frequency of each character in a given message M. Note that the time for these calculations is linear in the length of M. For example, suppose that the characters in M are the letters 'a' . . . 'g' as shown previously, and their frequencies are as given in Figure 13.15.

The size of M is 100,000 characters. If we ignored frequencies and encoded each character into a unique 3-bit sequence, we would need 300,000 bits to encode the message M. We'll soon see how far this is from an optimal encoding.

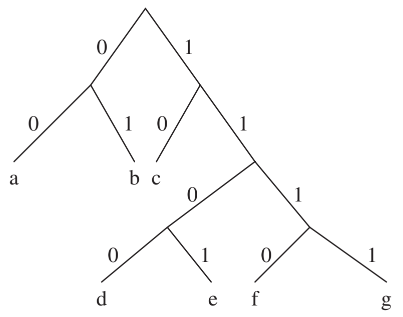


FIGURE 13.14 A binary tree that determines a prefix-free encoding of 'a' . . . 'g'

```
a:  5,000
b:  2,000
c: 10,000
d:  8,000
e: 22,000
f: 49,000
g:  4,000
```

FIGURE 13.15 Sample character-frequencies for a message of 100,000 characters from 'a' . . . 'g'

Once we have calculated the frequency of each character, we will insert each character-frequency pair into a priority queue ordered by increasing frequencies. That is, the front character-frequency pair in the priority queue will have the *least* frequently occurring character. These characters will end up being farthest from the root in the prefix-free tree, so their encoding will have the most bits. Conversely, characters that occur most frequently will have the fewest bits in their encodings.

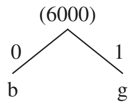
Initially we insert the following pairs into the priority queue:

(a:5000) (b:2000) (c:10000) (d:8000) (e:22000) (f:49000) (g:4000)

In observance of the Principle of Data Abstraction, we will not rely on any implementation details of the `PriorityQueue` class. So all we know about this priority queue is that an access or removal would be of the pair (b:2000). We do not assume any order for the remaining elements, but for the sake of simplicity, we will henceforth show them in increasing order of frequencies—as they would be returned by repeated calls to `remove()`.[2] Note that there is no need to sort the priority queue.

13.5.1 Huffman Trees

The binary tree constructed from the priority queue of character-frequency pairs is called a ***Huffman tree***. We create the Huffman tree from the bottom up because the front of the priority queue has the least-frequently-occurring character. We start by calling the `remove()` method twice to get the two characters with lowest frequencies. The first character removed, 'b', becomes the left leaf in the binary tree, and 'g' becomes the right leaf. The sum of their frequencies becomes the root of the tree and is added to the priority queue. We now have the following Huffman tree

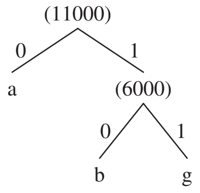


The priority queue contains

(a:5000) (:6000) (d:8000) (c:10000) (e:22000) (f:49000)

Technically, the priority queue and the Huffman tree consist of character-frequency pairs. But the character can be ignored when two frequencies are summed, and the frequencies can be ignored in showing the leaves of the Huffman tree. And anyway, the algorithm works with references to the pairs rather than the pairs themselves. This allows references to represent the typical binary-tree constructs—left, right, root, parent—that are needed for navigating through the Huffman tree.

When the pairs (a:5000) and (:6000) are removed from the priority queue, they become the left and right branches of the extended tree whose root is the sum of their frequencies. That sum is added to the priority queue. We now have the Huffman tree:

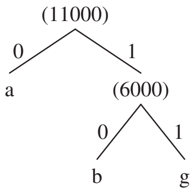


[2]In fact, if we did store the elements in a heap represented as an array, the contents of that array would be in the following order: (b:2000),(a:5000),(g:4000),(d:8000),(e:22000),(f:49000),(c:10000).

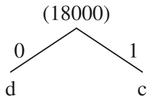
The priority queue contains

(d:8000) (c:10000) (:11000) (e:22000) (f:49000)

When ‘d’ and ‘c’ are removed, they cannot yet be connected to the main tree, because neither of their frequencies is at the root of that tree. So they become the left and right branches of another tree, whose root—their sum—is added to the priority queue. We temporarily have two Huffman trees:



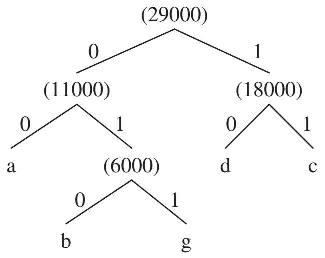
and



The priority queue now contains

(:11000) (:18000) (e:22000) (f:49000)

When the pair (:11000) is removed, it becomes the left branch of the binary tree whose right branch is the next pair removed, (:18000). The sum becomes the root of this binary tree, and that sum is added the priority queue, so we have the following Huffman tree:



The priority queue contains

(e:22000) (:29000) (f:49000)

When the next two pairs are removed, ‘e’ becomes the left branch and (:29000) the right branch of the Huffman tree whose root, (:51000), is added to the priority queue. Finally the last two pairs, (f:49000) and (:51000) are removed and become the left and right branches of the final Huffman tree. The sum of those two frequencies is the frequency of the root, (:100000), which is added as the sole element into the priority queue. The final Huffman tree is shown in Figure 13.16.

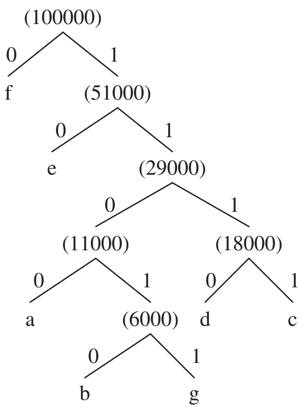


FIGURE 13.16 The Huffman tree for the character-frequencies in Figure 13.15

To get the Huffman encoding for ‘d’, for example, we start at the leaf ‘d’ and work our way back up the tree to the root. As we do, each bit encountered is pre-pended—placed at the front of—the bit string that represents the Huffman encoding. So the encoding in stages, is

```
0
10
110
1110
```

That is, the encoding for ‘d’ is 1110. Here are the encodings for all of the characters:

```
a: 1100
b: 11010
c: 1111
d: 1110
e: 10
f: 0
g: 11011
```

It is now an easy matter to translate the message M into an encoded message E. For example, if the message M starts out with

```
fad...
```

Then the encoded message E starts out with the encoding for ‘f’ (namely, 0) followed by the encoding for ‘a’ (namely, 1100) followed by the encoding for ‘d’ (namely, 1110). So E has

```
011001110...
```

What happens on the receiving end? How easy is it to decode E to get the original message M? Start at the root of the tree and the beginning of E, take a left branch in the tree for a 0 in E, and take a right branch for a 1. Continue until a leaf is reached. That is the first character in M. Start back at the top of the tree and continue reading E. For example, if M starts with “cede”, then E starts with 111110111010. Starting at the root of the tree, the four ones at the beginning of E lead us to the leaf ‘c’. Then we go back to the root, and continue reading E with the fifth 1. We take right branch for that 1 and then a left branch for the 0, and we are at ‘e’. The next few bits produce ‘d’ and ‘e’. In other words, we now have “cede”, and that, as expected, is the start of M.

The size of the message E is equal to the sum, over all characters, of the number of bits in the encoding of the character times the frequency of that character in M. So to get the size of E in this example, we take the product of the four bits in the encoding of ‘a’ and the 5000 occurrences of ‘a’, add to that the product of the five bits in the encoding of ‘b’ and the 2000 occurrences of ‘b’, and so on. We get:

$$\begin{aligned}&(4 * 5000) + (5 * 2000) + (4 * 10000) + (4 * 8000) +\\&(2 * 22000) + (1 * 49000) + (5 * 4000)\\&\quad = 215,000\end{aligned}$$

This is about 30% less than the 300,000 bits required with the fixed-length, 3-bits-per-character encoding discussed earlier. So the savings in space required and transmission time is significant. But it should be noted that a fixed-length encoding can usually be decoded more quickly than a Huffman encoding; for example, the encoded bits can be interpreted as an array index—the entry at that index is the character encoded.

13.5.2 Greedy Algorithm Design Pattern

Huffman’s algorithm for encoding a message is an example of the Greedy Algorithm design pattern. In a ***greedy algorithm***, locally optimal choices are made, in the hope that these will lead to a globally optimal solution. In the case of Huffman’s algorithm, during each loop iteration the two smallest-valued elements are removed from the priority queue and made the left and right branches of a binary tree. Choosing the smallest-valued elements is locally optimal, that is, greedy. And the end result—globally optimal—is a minimal prefix-free encoding. So greed succeeds. We will encounter two more examples of greedy algorithms in Chapter 15; they also involve priority queues.

13.5.3 The Huffman Encoding Project

To add substance to the prior discussion, let’s develop a project that handles the encoding of a message. The decoding phase is covered in Programming Project 13.1. The input will consist of a file path for the original message, and a file path denoting where the character codes and encoded message should be saved. For example, suppose that the file `huffman.in1` contains the following message:

```
more money needed
```

System Test 1 (input in boldface):
Please enter the path for the input file: **huffman.in1**
Please enter the path for the output file: **huffman.ou1**

After the execution of the program, the file huffman.ou1 will consist of two parts. The first part will have each character in the original message and that character’s code. The second part, separated from the first by “**”, will have the encoded message. Here is the complete file:

```
                // the first line in the file is blank (see next paragraph)
 0110           // the encoding for \n is 0110
  1011          // the encoding for the blank character is 1011
d 100
e 11
m 001
n 000
o 010
```

```
r 0111
y 1010
**
001010011111101100101000011101010110001111100111000110
```

In the encoding, the first character is the new-line marker, `'\n'`. When the output file is viewed, the first line is blank because a new-line feed is carried out. The second line starts with a space, followed by the code for the new-line. This may seem a bit strange because the first line in the encoding is

```
\n 0110
```

But when this line is printed, the new-line feed is carried out instead of `'\n'` being printed. So the space between `'\n'` and `0110` starts the *second* line. Subsequent lines start with the character encoded, a space, and the code for that character.

For System Test 2, assume that the file huffman.in2 contains the following message:

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

System Test 2 (input in boldface):
Please enter the path for the input file: **huffman.in2**
Please enter the path for the output file: **huffman.ou2**

Here is the complete file huffman.ou2 after execution of the program:

```
 10001
  101
, 001000
- 0100000
. 0100001
: 0010100
A 001011
D 0100110
I 0100101
K 001110
T 0011011
W 0010101
X 0011000
a 000
b 0011001
c 110010
d 11110
e 011
g 0011010
h 11111
i 001111
l 11101
m 110011
n 0101
o 11000
p 001001
```

```
r 1001
s 1101
t 10000
u 11100
v 010001
w 0100111
y 0100100
**
0100101010110100110000000101000111101110010111110001111111010100111
0111000011001111010001010011101111100001011000100101110111011000000
1000001111101010010010100100111101011000110111100100101101000001111 0
1100011001101110111110011110010100101101100101001000100101011111101 1
1001011101001011111010010011111100100010110000111110111011101000110 0
1010010111111010110010011110100010111001001000101100100001011000100 1
1011111111001110001110000110101111110111001000001000101110010101110 1
1011100110110001101111001001011111010111101110110110000110001011100 1
1000010110001010011011000010011101011011000011000101000101110111100 0
1011110101111011101101110101100001000011000 1
```

As always, we embrace modularity by separating the input-output details, in a `HuffmanUser` class, from the Huffman-encoding details, in a `Huffman` class. We start with the lower-level class, `Huffman`, in Section 13.5.3.1.

13.5.3.1 Design and Testing of the `Huffman` Class

The `Huffman` class has several responsibilities. First, it must initialize a `Huffman` object. Also, it must determine the frequency of each character. This must be done line-by-line so that the `Huffman` class can avoid entanglement in input-output issues. The other responsibilities are to create the priority queue and Huffman tree, to calculate the Huffman codes, and to return both the character codes and, line-by-line, the encoded message. From the discussion in Section 13.5.1, we want the Huffman class to be able to access the entries, that is, elements, in a priority queue or Huffman tree. For this we will create an `Entry` class. The `Entry` class has two fields related to the encoding, three fields related to binary-tree traversal, a `compareTo` method based on frequencies, and `getFreq` and `getCode` methods for the sake of testing:

```java
public class Entry implements Comparable<Entry>
{
    int freq;

    String code;

    Entry left,
          right,
          parent;

    public int compareTo (Entry entry)
    {
            return freq - entry.freq;
    } // compareTo

   public int getFreq()
   {
           return freq;
   } // method getFreq
```

```
    public String getCode()
    {
             return code;
    } // method getCode

} // class Entry
```

We can now present the method specifications for the `Huffman` class. For the second through fifth of the following method specifications, the return values are only for the sake of testing.

```
/**
 *  Initializes this Huffman object.
 *
 */
public Huffman()

/**
 *  Updates the frequencies of the characters in a scanned-in line.
 *
 *  @param line - the line scanned in.
 *
 *  @return - a cumulative array of type Entry in which the frequencies have been
 *       updated for each character in the line.
 *
 */
public Entry[ ] updateFrequencies (String line)

/**
 *  Creates the priority queue from the frequencies.
 *
 *  @return - the priority queue of frequencies (in increasing order).
 *
 */
public PriorityQueue<Entry> createPQ()

/**
 *  Creates the Huffman tree from the priority queue.
 *
 *  @return - an Entry representing the root of the Huffman tree.
 *
 */
public Entry createHuffmanTree()

/**
 *  Calculates and returns the Huffman codes.
 *
 *  @return - an array of type Entry, with the Huffman code
 *       for each character.
 *
 */
public Entry[ ] calculateHuffmanCodes()

/**
 *  Returns, as a String object, the characters and their Huffman codes.
```

```
     *
     *  @return the characters and their Huffman codes.
     *
     */
    public String getCodes()

    /**
     *  Returns a String representation of the encoding of a specified line.
     *
     *  @param line – the line whose encoding is returned.
     *
     *  @return a String representation of the encoding of line.
     *
     */
    public String getEncodedLine (String line)
```

By convention, since no time estimates are given, you may assume that for each method, worstTime(n) is constant, where n is the size of the original message.

The `Huffman` methods must be tested in sequence. For example, testing the `getCodes` method assumes that the `updateFrequencies`, `createPQ`, `createHuffmanTree`, and `calculateHuffmanCodes` are correct. Here is a test of the `getCodes` method, with `huffman` a field in the `HuffmanTest` class:

```
    @Test
     public void testGetCodes()
     {
        huffman.updateFrequencies ("aaaabbbbbbbbbbbbbbbbccdddddddd");
        huffman.createPQ();
        huffman.createHuffmanTree();
        huffman.calculateHuffmanCodes();
        assertEquals ("\n 0000\na 001\nb 1\nc 0001\nd 01\n", huffman.getCodes());
     } // method testGetCodes
```

All of the files, including test files, are available from the book's website.

At this point, we can determine the fields that will be needed. Clearly, from the discussion in section 13.5.1, we will need a priority queue. Given a character in the input, it will be stored in a leaf in the Huffman tree, and we want to be able to access that leaf-entry quickly. To allow random-access, we can choose an `ArrayList` field or an array field. Here the nod goes to an array field because of the speed of directly utilizing the index operator, `[ ]`, versus indirect access with the `ArrayList`'s `get` and `set` methods. Note that if the original message file is very large, most of the processing time will be consumed in updating the frequencies of the characters and, later, in calculating the codes corresponding to the characters in the message file. Both of these tasks entail accessing or modifying entries.

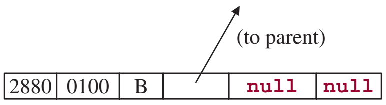
That gives us two fields:

```
    protected Entry [ ] leafEntries;

    protected PriorityQueue<Entry> pq;
```

For the sake of simplicity, we restrict ourselves to the 256 characters in the extended ASCII character set, so we have one array slot for each ASCII character:

```
    public final static int SIZE = 256;
```

For example, if the input character is 'B', the information for that character is stored at index (**int**)'B', which is 66. If the encoding for 'B' is 0100 and the frequency of 'B' in the input message is 2880, then the entry at `leafEntries [66]` would be



The left and right references are **`null`** because leaves have no children. The reason that we use references rather than indexes for `parent`, `left`, and `right` is that some of the entries represent sums, not leaf-characters, so those entries are not in `leafEntries`. Every leaf is an entry, but not every entry is a leaf.

13.5.3.2 Definition of Methods in the `Huffman` Class

The definition of the default constructor is straightforward:

```
public Huffman()
{
    Entry entry;

    leafEntries = new Entry [SIZE];
    for (int i = 0; i < SIZE; i++)
    {
         leafEntries [i] = new Entry();
         entry = leafEntries [i];
         entry.freq = 0;
         entry.left = null;
         entry.right = null;
         entry.parent = null 1;
    } // initializing leafEntries

    pq = new PriorityQueue<Entry>();
} // default constructor
```

The `updateFrequencies` method adds 1 to the frequency for each character in the parameter `line`. The new-line character is included in this updating to ensure that the line structure of the original message will be preserved in the encoding. Here is the method definition:

```
public Entry[ ] updateFrequencies (String line)
{
    Entry entry;

    for (int j = 0; j < line.length(); j++)
    {
         entry = leafEntries [(int)(line.charAt (j))];
         entry.freq++;
    } // for

    // Account for the end-of-line marker:
    entry = leafEntries [(int)'\n'];
    entry.freq++;
```

```
        return leafEntries;
    } // method updateFrequencies
```

As noted earlier, worstTime(n) for this method is constant because n refers to the size of the entire message file, and this method works on a single line.

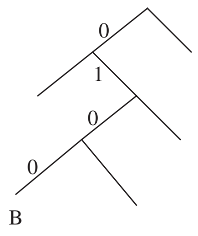
The priority queue is created from the entries with non-zero frequencies:

```
    public PriorityQueue<Entry> createPQ()
    {
        Entry entry;

        for (int i = 0; i < SIZE; i++)
        {
             entry = leafEntries [i];
             if (entry.freq > 0)
                 pq.add (entry);
        } // for
        return pq;
    } // createPQ
```

The Huffman tree is created "on the fly." Until the priority queue consists of a single entry, a pair of entries is removed from the priority queue and becomes the left and right children of an entry that contains the sum of the pair's frequencies; the sum entry is added to the priority queue. The root of the Huffman tree is returned (for the sake of testing). Here is the definition:

```
    public Entry createHuffmanTree()
    {
        Entry left,
              right,
              sum;

        while (pq.size() > 1)
        {
             left = pq.remove();
             left.code = "0";

             right = pq.remove();
             right.code = "1";

             sum = new Entry();
             sum.parent = null;
             sum.freq = left.freq + right.freq;
             sum.left = left;
             sum.right = right;
             left.parent = sum;
             right.parent = sum;

             pq.add (sum);
        } // while
        return pq.element();  // the root of the Huffman tree
    } // method createHuffmanTree
```

The Huffman codes are determined for each leaf entry whose frequency is nonzero. We create the code for that entry as follows: starting with an empty string variable `code`, we pre-pend the entry's `code` field (either “0” or “1”) to code, and then replace the entry with the entry’s parent. The loop stops when the entry is the root. The final value of code is then inserted as the code field for that entry in leafEntries. For example, suppose part of the Huffman tree is as follows:



Then the code for ‘B’ would be “0100”, and this value would be stored in the code field of the Entry at index 66 of leafEntries—recall that (int)'B' = 66 in the ASCII (and Unicode) collating sequence.

Here is the method definition:

```
public Entry[ ] calculateHuffmanCodes()
{
    String code;

    Entry entry;

    for (int i = 0; i < SIZE; i++)
    {
         code = "";
         entry = leafEntries [i];
         if (entry.freq > 0)
         {
             while (entry.parent != null)
             {
                 code = entry.code + code;  // current bit prepended to
                 entry = entry.parent;      // code as we go up the tree
             } // while
             leafEntries [i].code = code;
         } // if
    } // for
    return leafEntries;
} // calculateHuffmanCodes
```

In the getCodes method, a String object is constructed from each character and its code, and that String object is then returned:

```
public String getCodes()
{
      Entry entry;

      String codes = new String();

      for (int i = 0; i < SIZE; i++)
       {
         entry = leafEntries [i];
         if (entry.freq > 0)
```

```
                    codes += (char)i + " " + entry.code + "\n";
        } // for
        return codes;
    } // method getCodes
```

Finally, in the `getEncodedLine` method, a `String` object is constructed from the code for each character in `line`, appended by the new-line character, and that `String` object is returned:

```
    public String getEncodedLine (String line)
    {
        Entry entry;

        String encodedLine = new String();

        for (int j = 0; j < line.length(); j++)
        {
          entry = leafEntries [(int)(line.charAt (j))];
          encodedLine += entry.code;
        } // for
        entry = leafEntries [(int)'\n'];
        encodedLine += entry.code;
        return encodedLine;
    } // method getEncodedLine
```

13.5.3.3 The `HuffmanUser` Class

The `HuffmanUser` class has a `main` method to invoke a `run` method, a `run` method to open the input and output files from paths read in from the keyboard, a method to create the Huffman encoding, and a method to save the encoded message to the output file. Here are the method specifications for the `createEncoding` and `saveEncodedMessage` methods:

```
    /**
     *  Creates the Huffman encoding by scanning over a file to be encoded.
     *  The worstTime(n) is O(n).
     *
     *  @param fileScanner -a scanner over the file to be encoded.
     *  @param huffman -an instance of the Huffman class.
     *
     * @return - a String consisting of each character and its encoding
     *
     */
    public String createEncoding (Scanner fileScanner, Huffman huffman)

    /**
     *  Saves the Huffman codes and the encoded message to a file.
     *  The worstTime(n) is O(n).
     *
     *  @param printWriter - the PrintWriter object that holds the Huffman codes
     *                       and the encoded message.
     *  @param inFilePath - the String object that holds the path for the file
     *                       that contains the original message.
     *  @param huffman – an instance of the Huffman class.
     *
     */
    public void saveEncodedMessage (PrintWriter printWriter,  String inFilePath,
                                    Huffman huffman)
```

The book's website has unit tests for the `createEncoding` and `saveEncodedMessage` methods, and Figure 13.17 has the UML diagram for the `HuffmanUser` class.

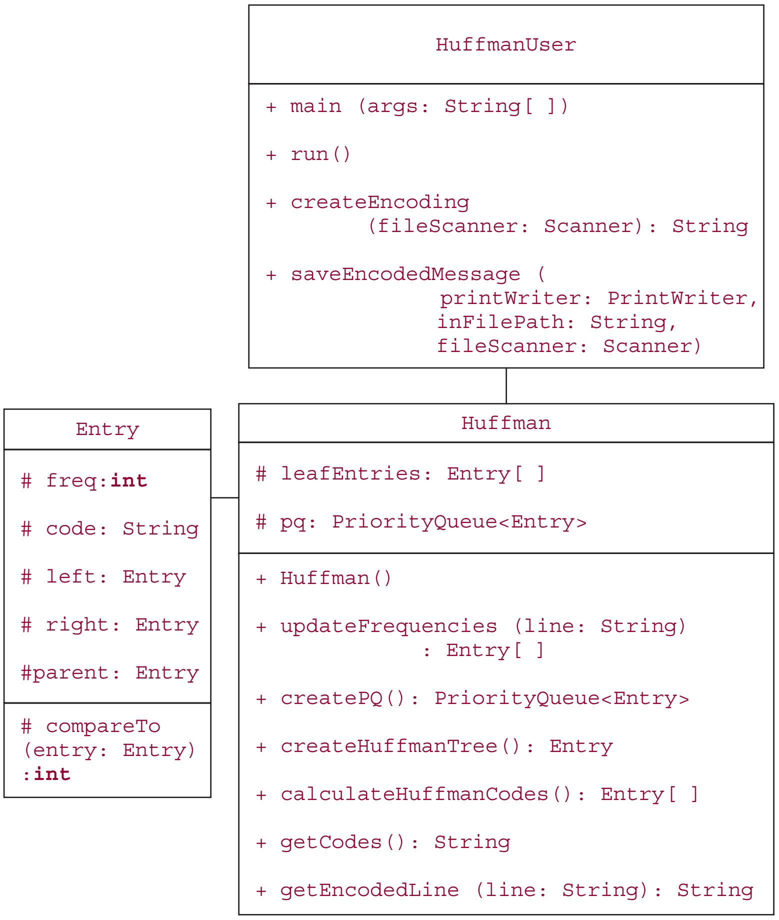

HuffmanUser
+ main (args: String[ ])
+ run()
+ createEncoding (fileScanner: Scanner): String
+ saveEncodedMessage (printWriter: PrintWriter, inFilePath: String, fileScanner: Scanner)
Entry
# freq:int
# code: String
# left: Entry
# right: Entry
#parent: Entry
# compareTo (entry: Entry) :int
Huffman
# leafEntries: Entry[ ]
# pq: PriorityQueue<Entry>
+ Huffman()
+ updateFrequencies (line: String) : Entry[ ]
+ createPQ(): PriorityQueue<Entry>
+ createHuffmanTree(): Entry
+ calculateHuffmanCodes(): Entry[ ]
+ getCodes(): String
+ getEncodedLine (line: String): String

FIGURE 13.17 UML diagram for the `HuffmanUser` class

13.5.3.4 Method Definitions for the `HuffmanUser` Class

Here are the definitions of the `main` method, the default constructor, and the `run` method:

```
    public static void main (String[] args)
    {
        new Huffman().run();
    } // method main


    public void run()
    {
        final String IN_FILE_PROMPT =
            "\nPlease enter the path for the input file: ";

        final String OUT_FILE_PROMPT =
            "\nPlease enter the path for the output file: ";

        Huffman huffman = new Huffman();

        PrintWriter printWriter = null;

        Scanner keyboardScanner = new Scanner (System.in),
                       fileScanner = null;

       String inFilePath = null,
                   outFilePath,
                   line;

        boolean pathsOK = false;

        while (!pathsOK)
        {
            try
            {
                 System.out.print (IN_FILE_PROMPT);
                 inFilePath = keyboardScanner.nextLine();
                 fileScanner = new Scanner(new File (inFilePath));
                 System.out.print (OUT_FILE_PROMPT);
                 outFilePath = keyboardScanner.nextLine();
                 printWriter = new PrintWriter (new FileWriter (outFilePath));
                 pathsOK = true;
            } // try
            catch (IOException e)
            {
                System.out.println (e);
            } // catch
        } // while !pathOK
        createEncoding (fileScanner, huffman);
        saveEncodedMessage (printWriter, inFilePath, huffman);
    } // method run
```

In that `run` method, the **`null`** assignments are needed to avoid a compile-time (" . . . may not have been initialized") error for the arguments to the `saveEncodedMessage` method. That method is called outside of the **`try`** block in which `inFilePath` and `printWriter` are initialized.

In order to create the encoding, we will scan the input file and update the frequencies line-by-line. We then create the priority queue and Huffman tree, and calculate the Huffman codes. Most of those tasks are handled in the `Huffman` class, so we can straightforwardly define the `createEncoding` method:

```
public String createEncoding (Scanner fileScanner, Huffman huffman)
{
    String line;

   while (fileScanner.hasNextLine())
    {
        line = fileScanner.nextLine();
        huffman.updateFrequencies (line);
    } // while
    fileScanner.close(); // re-opened in saveEncodedMessage
    huffman.createPQ();
    huffman.createHuffmanTree();
    huffman.calculateHuffmanCodes();
    return getCodes();
} // method createEncoding
```

For the `saveEncodedMessage` method, we first write the codes to the output file, and then, for each line in the input file, write the encoded line to the output file. Again, the hard work has already been done in the `Huffman` class. Here is the definition of `saveEncodedMessage`:

```
public void saveEncodedMessage (PrintWriter printWriter, String inFilePath,
                                Huffman huffman)
{
    String line;

    try
    {
        printWriter.print (huffman.getCodes());
        printWriter.println ("**"); // to separate codes from encoded message
        Scanner fileScanner = new Scanner (new File (inFilePath));

        while (fileScanner.hasNextLine())
        {
            line = fileScanner.nextLine();
            printWriter.println (huffman.getEncodedLine (line));
        } // while
        printWriter.close();
    } // try
    catch (IOException e)
    {
        System.out.println (e);
    } // catch IOException
} // method saveEncodedMessage
```

Later, another user may want to decode the message. This can be done in two steps:

1. The encoding is read in and the Huffman tree is re-created. For this step, only the essential structure of the Huffman tree is needed, so the `Huffman` class is not used and the `Entry` class will have only three fields:

   ```
   Entry left,
         right;

   char id; // the character that is a leaf of the tree
   ```

   The root entry is created, and then each encoding is iterated through, which will create new entries and, ultimately, a leaf entry.

2. The encoded message is read in, decoded through the Huffman tree, and the output is the decoded message, which should be identical to the original message. Decoding the encoded message is the subject of Programming Project 13.1.

SUMMARY

This chapter introduced the ***priority queue***: a collection in which removal allowed only of the highest-priority element in the sequence, according to some method for comparing elements. A priority queue may be implemented in a variety of ways. The most widely used implementation is with a heap. A ***heap*** *t* is a complete binary tree such that either *t* is empty or

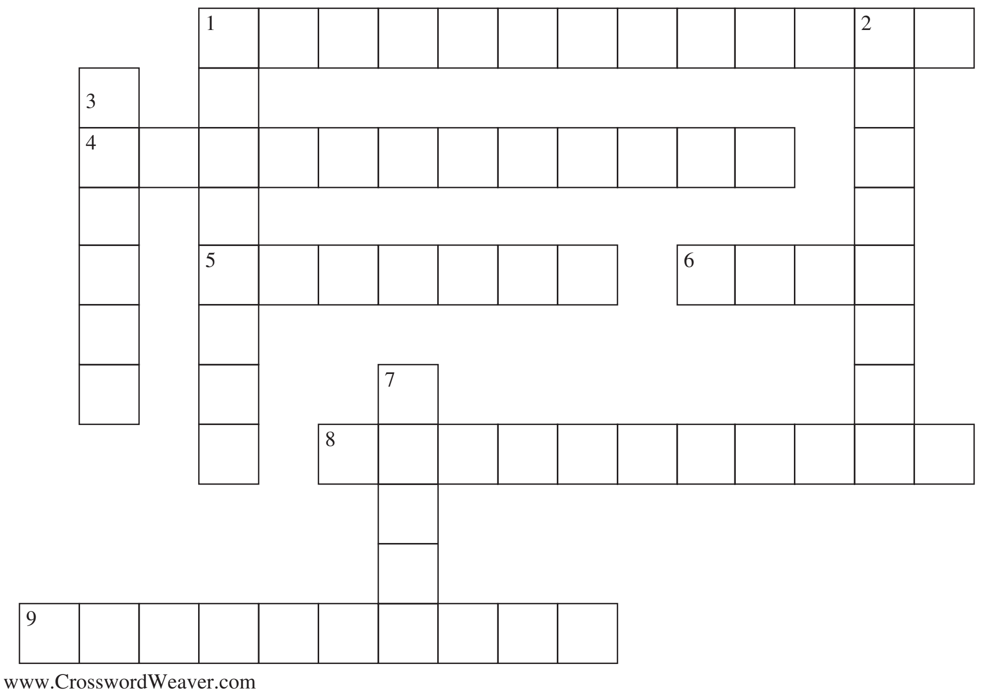
1. the root element of *t* is smallest element in t, according to some method for comparing elements;
2. the left and right subtrees of *t* are heaps.

Because array-based representations of complete binary trees allow rapid calculation of a parent's index from a child's index and vice versa, the heap can be represented as an array. This utilizes an array's ability to randomly access the element at a given index.

The `PriorityQueue` class's methods can be adapted to achieve the `heapSort` method, whose worstTime(n) is linear logarithmic in n and whose worstSpace(n) is constant.

An application of priority queues is in the area of data compression. Given a message, it is possible to encode each character, unambiguously, into a minimum number of bits. One way to achieve such a minimum encoding is with a Huffman tree. A ***Huffman tree*** is a two-tree in which each leaf represents a distinct character in the original message, each left branch is labeled with a 0, and each right branch is labeled with a 1. The Huffman code for each character is constructed by tracing the path from that leaf character back to root, and pre-pending each branch label in the path.

CROSSWORD PUZZLE



ACROSS

1. The main advantage of Heap Sort over Merge Sort
4. Why an array is used to implement a heap
5. Every Huffman Tree is a _____________.
6. A `PriorityQueue` object is not allowed to have any ____ elements.
8. The prefix-free binary tree constructed from the priority queue of character-frequency pairs
9. Why $(k - 1)$ >>> 1 is used instead of $(k - 1)/2$

DOWN

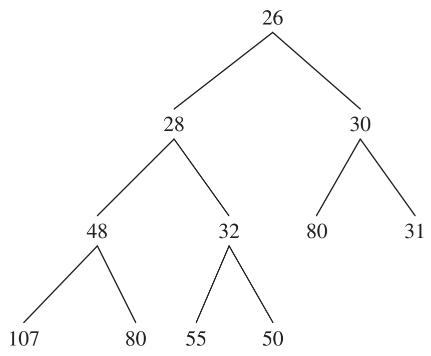
1. The averageTime(n) for the `add (E element)` in the `PriorityQueue` class
2. Every heap must be a _____ binary tree.
3. In a ___________ algorithm, locally optimal choices lead to a globally optimal solution.
7. The field in the `PriorityQueue` class that holds the elements

CONCEPT EXERCISES

13.1 Declare a `PriorityQueue` object—and the associated `Comparator` -implementing class—in which the highest-priority element is the `String` object of greatest length in the priority queue; for elements of equal length, use lexicographical order. For example, if the elements are “yes”, “maybe”, and “no”, the highest-priority element would be “maybe”.

13.2 In practical terms, what is the difference between the `Comparable` interface and the `Comparator` interface? Give an example in which the `Comparator` interface must be used.

13.3 Show the resulting heap after each of the following alterations is made, consecutively, to the following heap:



a. `add (29);`

b. `add (30);`

c. `remove ();`

d. `remove ();`

13.4 For the following character frequencies, create the heap of character-frequency pairs (highest priority = lowest frequency):

a: 5,000

b: 2,000

c: 10,000

d: 8,000

e: 22,000

f: 49,000

g: 4,000

13.5 Use the following Huffman code to translate “faced” into a bit sequence:

a: 1100

b: 1101

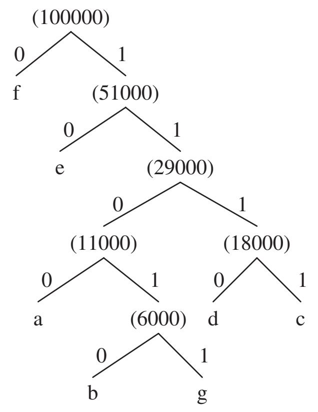
c: 1111

d: 1110

e: 10

f: 0

13.6 Use the following Huffman tree to translate the bit sequence 11101011111100111010 back into letters 'a' . . . 'g':



13.7 If each of the letters 'a' through 'f' appears at least once in the original message, explain why the following cannot be a Huffman code:

a: 1100

b: 11010

c: 1111

d: 1110

e: 10

f: 0

13.8 Must a Huffman tree be a two-tree? Explain.

13.9 Provide a message with the alphabet 'a' . . . 'e' in which two of the letters have a Huffman code of 4 bits. Explain why it is impossible to create a message with the alphabet 'a' . . . 'e' in which two of the letters have a Huffman code of 5 bits. Create a message with the alphabet 'a' . . . 'h' in which all of the letters have a Huffman code of 3 bits.

13.10 In Figure 13.16, the sum of the frequencies of all the non-leaves is 215,000. This is also the size of the encoded message E. Explain why in any Huffman tree, the sum of the frequencies of all non-leaves is equal to the size of the encoded message.

13.11 Give an example of a `PriorityQueue` object of ten unique elements in which, during the call to `remove()`, the call to `siftDown` would entail only one swap of parent and child.

13.12 This exercise deals with Heap Sort, specifically, the number of iterations required to create a heap from an array in reverse order. Suppose the elements to be sorted are in an array of `Integer` s with the following **int** values:

15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

a. Calculate the total number of loop iterations in the **`while`** loop of `siftDownComparable` or `siftDownComparator` to convert the array `queue` into a heap.

b. Suppose $n = 31$ instead of 15, and the elements are still in reverse order. Calculate the total number of loop iterations in the the **`while`** loop of `siftDownComparable` or `siftDownComparator`.

c. Let n be one less than a power of 2—the complete binary tree will be full—and suppose the n elements are in reverse order. If we let $h = \text{floor}(\log_2 n)$, we can develop a formula for calculating the number of **while** -loop iterations in the $n/2$ calls to siftDown. (For clarity, we will start calculating at the root level, even though the calls to siftDown start at level $h - 1$.) At the root level, there is one element, and the call to siftDown entails h loop iterations. At the next level down, there are two elements, and each of the corresponding calls to siftDown entails $h - 1$ loop iterations. At the next lowest level down, there are 4 elements, and each call to siftDown entails $h - 2$ iterations. And so on. Finally, at level $h - 1$, the next-to-leaf level, there are 2^{h-1} elements, and each call to siftDown (i) entails one loop iteration. The total number of **while** -loop iterations is:

$$1 * h + 2 * (h - 1) + 4 * (h - 2) + 8 * (h - 3) + \ldots + 2^{h-1} * 1 = \sum_{i=0}^{h-1} 2^i (h - i)$$

Show that this sum is equal to $n - \text{floor}(\log_2 n) - 1$; there are also n/2 calls to siftDown. That is, for creating a heap from a full binary tree, worstTime(n) is linear in n.

Hint: By Exercise A2.6 in Appendix 2,

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

By Exercise A2.3 in Appendix 2,

$$\sum_{i=0}^{h-1} i2^i = (h - 2)2^h + 2$$

By the Binary Tree Theorem, if t is a (non-empty) full binary tree of n elements and height h,

$$(n + 1)/2 = 2^h$$

d. Let t be a complete binary tree with n elements. Show that to make a heap from t, worstTime(n) is linear in n.

Hint: Let h be the height of t. For the number of **while** -loop iterations in siftDownComparable or siftDownComparator, compare the number of iterations, in the worst case, to make a heap from t with

i. the number of iterations, in the worst case, to make a heap from the full binary tree t_1 of height $h - 1$, and

ii. the number of iterations, in the worst case, to make a heap from the full binary tree t_2 of height h.

PROGRAMMING EXERCISES

13.1 In Section 13.3.1, the PriorityQueueExample class creates a heap, pq1, of Student objects; the Student class implements the Comparable interface. Re-write the project so that the Comparator interface is implemented instead for pq1. Re-run the project to confirm your revisions.

13.2 Conduct a run-time experiment to support the claim that, on average, Heap Sort takes more time and uses less space than Merge Sort.

13.3 Conduct a run-time experiment to support the claim that averageTime(n) is constant for the add method in the PriorityQueue class. **Hint:** Copy the PriorityQueue class into one of your directories; replace the line `package java.util;` with `import java.util.*;`. Add a siftUpCount field, initialize it to 0, and increment it by 1 during the **while** loop in siftUpComparable (and in siftUpComparator). Define a public getSiftUpCount method that returns the value of siftUpCount. Create another class whose run method scans in a value for n, initializes a PriorityQueue<Integer> object with an initial capacity of n, fills in that PriorityQueue object with n randomly generated integer values, and prints out the value of siftUpCount divided (not integer division) by n. That quotient should be slightly less than 2.3.

Programming Project 13.1

Decoding a Huffman-Encoded Message

Suppose a message has been encoded by a Huffman encoding. Design, test, and implement a project to decode the encoded message and thus retrieve the original message.

Analysis For the entire project, worstTime(*n*) must be O(*n*), where *n* is the size of the original message. As shown in the output file in the Huffman project of this chapter, the input file will consist of two parts:

each character and its encoding;

the encoded message.

A sample input file, huffman.ou1, is shown in the following. The first line contains the carriage-return character, followed by a space, followed by its encoding. The carriage-return character, when viewed, forces a line-feed. So the first line below is blank, and the second starts with a space, followed by the code for the carriage-return character.

```
 0010
  0111
a 000
b 1
c 0011
d 010
e 0110
**
1000010011100110110010011001110011000101110100001001001001100000100111
0100000100111001100001000101111111111111111111111111111110010
```

System Test 1: (Input in boldface)

Please enter the name of the input file: **huffman.ou1**

Please enter the name of the output file: **decode.ou1**

The file decode.ou1 will contain:

```
bad cede cab dab
dead dad cad
bbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Suppose the file huffman.ou2 contains the following (the message is from Coleridge's "Rubiyat of Omar Khayam"):

```
10001
  101
, 001000
- 0100000
. 0100001
```

(continued on next page)

(continued from previous page)

```
: 0010100
A 001011
D 0100110
I 0100101
K 001110
T 0011011
W 0010101
X 0011000
a 000
b 0011001
c 110010
d 11110
e 011
g 0011010
h 11111
i 001111
l 11101
m 110011
n 0101
o 11000
p 001001
r 1001
s 1101
t 10000
u 11100
v 010001
w 0100111
y 0100100
**
01001010101101001100000001010001111011100101111100011111
11101010011101110000110011110100010100111011111000010110
00100101110111011000000010000011111010100100101001001111
01011000110111100100101101000001111011000110011011101111
10011110010100101101100101001000100101011111101110010111
01001011111010010011111100100010110000111110111011101000
11001010010111111010110010011110100010111001001000101100
10000101100010011011111111001110001110000110101111110111
001000000100010111001010111011011100110110001101111001001
011111101011110111011011000011000101110011000010110001010
01101100001001110101101100001100010100010111011110001011
11010111101110110111010110000100001100001
```

System Test 2:

Please enter the name of the input file: **huffman.ou2**

Please enter the name of the output file: **decode.ou2**

The file decode.ou2 will contain

```
In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Note: For the sake of overall efficiency, re-creating the Huffman tree will be better than sequentially searching the codes for a match of each sequence of bits in the encoded message.

Programming Project 13.2

An Integrated Web Browser and Search Engine, Part 5

In Part 4, you printed the search-results as they were obtained. In this part of the project, you will save the results, and then print them in decreasing order of frequencies. For example, if you ran System Test 1 from Part 4 (Programming Project 12.5), the output would now be

```
Here are the files and relevance frequencies, in decreasing order:
browser.in6  7
browser.in8  2
browser.in7  0
```

In saving the results, the data structure must satisfy the following:

to insert a result into the data structure, worstTime(n) must be O(n), and averageTime(n) must be constant;

to remove a result from the data structure, worstTime(n) and averageTime(n) must be O($\log n$);

System Test 1: (Assume search.in1 contains browser.in6, 7, 8 from Programming Project 12.5)

```
Please enter a search string in the input line and then press the Enter key.
neural network

Here are the files and relevance frequencies, in decreasing order:

browser.in6  7
browser.in8  2
browser.in7  0
```

System Test 2: (Assume search.in1 contains browser.in10, 11, 12, 13 –shown below)

```
Please enter a search string in the input line and then press the Enter key.
```

(continued on next page)

(continued from previous page)

neural network

neural network

Here are the files and relevance frequencies, in decreasing order:

browser.in10	8
browser.in13	8
browser.in11	6
browser.in12	6

Here are the contents of the files from System Test 2:
browser.in10:

```
In Xanadu did Kubla Khan
A stately <a href=browser.in3>browser3</a> pleasure-dome decree:
Where Alph, the sacred river, <a href=browser.in4>browser4</a> ran
Through caverns measureless to man
Down to a <a href=browser.in5>browser5</a> sunless sea.
neural network neural network neural network neural network
And so it goes.
```

browser.in11:

```
In Xanadu did <a href=browser.in1>browser1</a> Kubla Khan
A stately pleasure-dome decree:
Where Alph, the neural network sacred river, ran
Through caverns neural network measureless to man
Down to a network sunless sea.
network
```

browser.in12:

```
Neural surgeons have a network.  But the decree is a decree is
a network and a network is a network, neural or not.
```

browser.in13:

```
In Xanadu did Kubla Khan
A stately <a href=browser.in3>browser3</a> pleasure-dome decree:
Where Alph, the sacred river, <a href=browser.in4>browser4</a> ran
Through caverns measureless to man
Down to a <a href=browser.in5>browser5</a> sunless sea.
neural network neural network neural network neural network
```

Note 1: In your code, do not use complete path names, such as “h:\Project3\home.in1”. Instead, use “home.in1”.

Hashing

CHAPTER 14

We start this chapter by reviewing some search algorithms from earlier chapters as a prelude to the introduction of a new search technique: hashing. The basic idea with hashing is that we perform some simple operation on an element's key to obtain an index in an array. The element is stored at that index. With an appropriate implementation, hashing allows searches (as well as insertions and removals) to be performed in constant average time. Our primary focus will be on understanding and using the `HashMap` class in the Java Collections Framework, but we will also consider other approaches to hashing. The application will use hashing to insert identifiers into a symbol table.

CHAPTER OBJECTIVES

1. Understand how hashing works, when it should be used, and when it should not be used.
2. Explain the significance of the Uniform Hashing Assumption.
3. Compare the various collision handlers: chaining, offset-of-1, quotient-offset.

14.1 A Framework to Analyze Searching

Before we begin looking at hashing, we need a systematic way to analyze search methods in general. Because the search may be successful or unsuccessful, the analysis of search methods should include both possibilities. For each search method we estimate $\text{averageTime}_S(n)$, the average time—over all n elements in the collection—of a successful search. As always for average time, we make the simplifying assumption that each element in the collection is equally likely to be sought.

We will also be interested in $\text{worstTime}_S(n)$, the largest number of statements needed to successfully search for an element. That is, for a given value of n, we look at all permutations of the n elements and all possible choices of the element to be successfully sought. For each permutation and element, we determine the number of iterations (or recursive calls) to find that element. Then $\text{worstTime}_S(n)$ corresponds to the largest number of iterations attained.

We also estimate $\text{averageTime}_U(n)$, the average time of an unsuccessful search, and $\text{worstTime}_U(n)$. For an unsuccessful search, we assume that on average, every possible failure is equally likely. For example, in an unsuccessful search of a sequence of n elements, there are $n + 1$ possibilities for where the given element can occur:

before the first element in the sequence;

between the first and second elements;

between the second and third elements;

...

between the (n - 1)st and nth elements;

after the nth element.

The next section reviews the search methods employed so far.

14.2 Review of Searching

Up to this point, we have seen three different kinds of searches: sequential search, binary search, and red-black-tree search. Let's look at each one in turn.

14.2.1 Sequential Search

A ***sequential search***—also called a "linear search"—of a collection starts at the beginning of the collection and iterates until either the element sought is found or the end of the collection is reached. For example, the `contains` method in the `AbstractCollection<E>` class uses a sequential search:

```
/**
 *  Determines if this AbstractCollection object contains a specified element.
 *  The worstTime(n) is O(n).
 *
 *  @param obj -the element searched for in this AbstractCollection object.
 *
 *  @return true -if this AbstractionCollection object contains obj; otherwise,
 *          return false.
 */
public boolean contains(Object obj)
{
       Iterator<E> e = iterator();   // E is the type parameter for this class
       if (obj == null)
       {
           while (e.hasNext())
             if (e.next()==null)
                 return true;
       }  // if obj == null
       else
       {
           while (e.hasNext())
             if (obj.equals(e.next()))
                 return true;
       } // obj!= null
       return false;
} // method contains
```

The `contains` method in the `ArrayList` class and the `containsValue` method in the `TreeMap` class use sequential searches similar to the above.

For a successful sequential search of a collection, we assume that each of the n elements in the collection is equally likely to be sought. So the average number of loop iterations is $(1 + 2 + \ldots + n)/n$, which is $(n + 1)/2$, and the largest possible number of loop iterations is n. We conclude that both $\text{averageTime}_S(n)$ and $\text{worstTime}_S(n)$ are linear in n.

For an unsuccessful sequential search, even if the collection happens to be ordered, we must access all n elements before concluding that the given element is not in the collection, so averageTime$_U(n)$ and worstTime$_U(n)$ are both linear in n. As we will see in Sections 14.2.2 and 14.2.3, ordered collections can improve on these times by employing non-sequential searches.

14.2.2 Binary Search

Sometimes we know beforehand that the collection is sorted. For sorted collections, we can perform a ***binary search***, so called because the size of the segment searched is repeatedly divided by two. Here, as investigated in Lab 8, is the `binarySearch` method from the `Arrays` class of the Java Collections Framework:

```
/**
 *  Searches a specified array for a specified element.
 *  The array must be sorted in ascending order according to the natural ordering
 *  of its elements (that is, by the compareTo method); otherwise, the results are
 *  undefined.
 *  The worstTime(n) is O(log n).
 *
 *  @param a -the array to be searched.
 *  @param key -the element to be searched for in the array.
 *
 *  @return the index of an element equal to key - if the array contains at least one
 *          such element; otherwise, -insertion point -1, where insertion point is
 *          the index where key would be inserted.  Note that the return value is
 *          greater than or equal to zero only if the key is found in the array.
 *
 */
public static int binarySearch (Object[ ] a, Object key)
{
   int low = 0;
   int high = a.length-1;


   while (low <= high)
   {
       int mid =(low + high) >> 1;
       Comparable midVal = (Comparable)a [mid];
       int cmp = midVal.compareTo(key);

       if (cmp < 0)
           low = mid + 1;
       else if (cmp > 0)
           high = mid - 1;
       else
           return mid; // key found
   } // while
   return  -(low + 1);  // key not found
} // method binarySearch
```

A binary search is much faster than a sequential search. For either a successful or unsuccessful search, the number of elements to be searched is $n =$ `high + 1 - low`. The **`while`** loop will continue to divide n

by 2 until either `key` is found or `high + 1 = low`, that is, until $n = 0$. The number of loop iterations will be the number of times n can be divided by 2 until $n = 0$. By the Splitting Rule in Chapter 3, that number is, approximately, $\log_2 n$ (also, see Example A2.2 of Appendix 2). So we get averageTime$_S(n) \approx$ worstTime$_S(n) \approx$ averageTime$_U(n) \approx$ worstTime$_U(n)$, which is logarithmic in n.

There is also a `Comparator` -based version of this method; the heading is

```
public static <T> int binarySearch (T[ ] a, T key, Comparator<? super T> c)
```

For example, if `words` is an array of `String` objects ordered by the length of the string, we can utilize the `ByLength` class from Section 11.3 and call

```
System.out.println (Arrays.binarySearch (words, "misspell", new ByLength()));
```

14.2.3 Red-Black-Tree Search

Red-black trees were introduced in Chapter 12. One of that class's **`public`** methods, `containsKey`, has a one-line definition: All it does is call the **`private`** `getEntry` method. The `getEntry` method, which returns the `Entry` corresponding to a given key, has a definition that is similar to the definition of the `binarySearch` method. Here is the definition of `getEntry`:

```
final Entry<K,V> getEntry(Object key) {
      // Offload comparator-based version for sake of performance
      if (comparator != null)
          return getEntryUsingComparator(key);
      if (key == null)
          throw new NullPointerException();
      Comparable<? super K> k = (Comparable<? super K>) key;
      Entry<K,V> p = root;
      while (p != null) {
          int cmp = k.compareTo(p.key);
          if (cmp < 0)
              p = p.left;
          else if (cmp > 0)
              p = p.right;
          else
              return p;
      }
      return null;
}
```

The height of a red-black tree t is logarithmic in n, the number of elements (see Example 2.6 in Appendix 2). For an unsuccessful search, the `getEntry` method iterates from the root to an empty subtree. In the worst case, the empty subtree will be a distance of height(t) branches from the root, and the number of iterations will be logarithmic in n. That is, worstTime$_U(n)$ is logarithmic in n.

The number of iterations in the average case depends on bh(root): the number of black elements in the path from the root to an element with no children or with one child. The path length from the root to such an element is certainly less than or equal to the height of the tree. As shown in Example 2.6 of Appendix 2, bh(root) $\geq$ height(t)/2. So the number of iterations for an unsuccessful search is between height(t)/2 and height(t). That is, averageTime$_U(n)$ is also logarithmic in n.

For a successful search, the worst case occurs when the element sought is a leaf, and this requires only one less iteration than an unsuccessful search. That is, worstTime$_S(n)$ is logarithmic in n. It is somewhat more difficult to show that averageTime$_S(n)$ is logarithmic in n. But the strategy and result are the same as in Concept Exercise 5.7, which showed that for the recursive version of the `binarySearch` method, averageTime$_S(n)$ is logarithmic in n.

Section 14.3 introduces a class that allows us to break through the log n barrier for insertions, removals and searches.

14.3 The HashMap Implementation of the Map Interface

The `HashMap` class implements the `Map` interface, so you saw most of the method headings when we studied the `TreeMap` class in Chapter 12. The main changes have to do with the timing estimates in some of the method specifications. Basically, for the `put`, `remove`, and `containsKey` methods, the average number of loop iterations is constant. There are also `size()`, `isEmpty()`, `clear()`, `toString()`, `entry Set()`, `keySet()`, and `valueSet()` methods, whose specifications are the same as for their `TreeMap` counterparts.

The class heading, with "K" for key and "V" for value, is

```
public class HashMap<K,V>
       extends AbstractMap<K,V>
       implements Map<K,V>, Cloneable, Serializable
```

Here is an example of the creation of a simple `HashMap` object. The entire map, both keys and values, is printed. Then the keys alone are printed by iterating through the map viewed as a `Set` object with keys as elements. Finally, the values alone are printed by iterating through the map as if it were a `Collection` of values.

```
import java.util.*;

public class HashExample
{
       public static void main (String[ ] args)
       {
              new HashExample().run();
       } // method main

       public void run()
       {
              HashMap<String, Integer> ageMap = new HashMap<String, Integer>();

              ageMap.put ("dog", 15);
              ageMap.put ("cat", 20);
              ageMap.put ("human", 75);
              ageMap.put ("turtle", 100);
              System.out.println (ageMap);

              for (String s : ageMap.keySet())
                  System.out.println (s);
```

```
                for (Integer i : ageMap.values())
                    System.out.println (i);
        } // method run

    } // class HashExample
```

The output will be as follows:

```
{cat=20, dog=15, turtle=100, human=75}
cat
dog
turtle
human
20
15
100
75
```

Notice that the output is not in order of increasing keys, nor in order of increasing values. (When we get to the details of the `HashMap` class, we'll see how the order of elements is determined.) This unsortedness is one of the few drawbacks to the `HashMap` class. The `HashMap` class is outstanding for insertions, removals, and searches, on average, but if you also need a sorted collection, you should probably use a `TreeMap` instead.

In trying to decide on fields for the `HashMap` class, you might at first be tempted to bring back a contiguous or linked design from a previous chapter. But if the elements are unordered, we will need sequential searches, and these take linear time. Even if the elements are ordered and we utilize that ordering, the best we can get is logarithmic time for searching.

The rest of this chapter is devoted to showing how, through the miracle of hashing, we can achieve searches—and insertions and removals—in constant time, on average. After we have defined what hashing is and how it works, we will spend a little time on the Java Collection Framework's implementation of the `HashMap` class. Then we will consider an application—hashing identifiers into a symbol table—before we consider alternate implementations.

14.3.1 Hashing

We have already ruled out a straightforward contiguous design, but just to ease into hashing, let's look at a contiguous design with the following two fields:

```
transient Entry[ ] table;  // an array of entries;

transient int size; // number of mappings in the HashMap object
```

Recall, from Chapter 6, that the **`transient`** modifier indicates that the field will not be saved during serialization. Appendix 1 includes a discussion of serialization.

We first present a simple example and then move on to something more realistic. Suppose that the array is constructed to hold 1024 mappings, that is, 1024 key&value pairs. The key is (a reference to) an `Integer` object that contains a three-digit **`int`** for a person's ID. The value will be a `String` holding a person's name, but the values are irrelevant to this discussion, so they will be ignored. We start by calling a constructor that initializes the table length to 1024:

```
HashMap<Integer, String> persons = new HashMap<Integer, String>(1024);
```

At this point, we have the state shown in Figure 14.1.

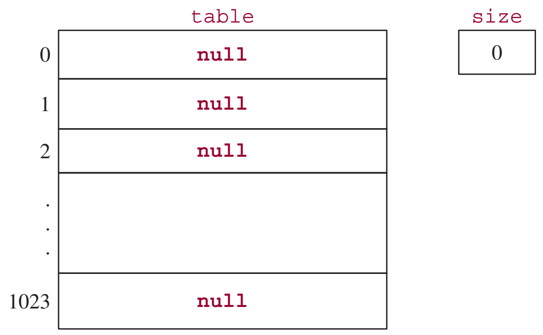


FIGURE 14.1 The design representation of an empty `HashMap` object (simplified design)

At what index should we store the element with key 251? An obvious choice is index 251. The advantages are several:

a. The element can be directly inserted into the array without accessing any other elements.

b. Subsequent searches for the element require accessing only location 251.

c. The element can be removed directly, without first accessing other elements.

We now add three elements to this `HashMap` object:

```
persons.put (251, "Smolenski");
persons.put (118, "Schwartz");
persons.put (335, "Beh-Forrest");
```

Figure 14.2 shows the contents of the `HashMap` object. We show the **`int`** values, rather than the `Integer` references, for the keys of inserted elements, and the names are omitted.

So far, this is no big deal. Now, for a slightly different application, suppose that `table.length` is still 1024, but each element has a social-security-number as the key. We need to transform the key into an index in the array, and we want this transformation to be accomplished quickly, that is, with few loop iterations. To allow fast access, we perform a bit-wise "and" of the social security number and `table.length - 1`. For example, the element with a key value of 214-30-3261—the hyphens are for readability only—has a binary representation of

00001100110001100000001000011101

The binary representation of 1023 is

00000000000000000000001111111111

The result of applying the bit-wise and operator, **`&`** , to a pair of bits is 1 if both bits are 1, and otherwise 0. We get

$$\begin{array}{rl} & 00001100110001100000001000011101 \\ \& & \underline{00000000000000000000001111111111} \\ & 00000000000000000000001000011101 \end{array}$$

The decimal value of the result is 541, and so the element whose key is 214-30-3261 is stored at index 541. Similarly, the element with a key value of 033-51-8000 would be stored at location 432.

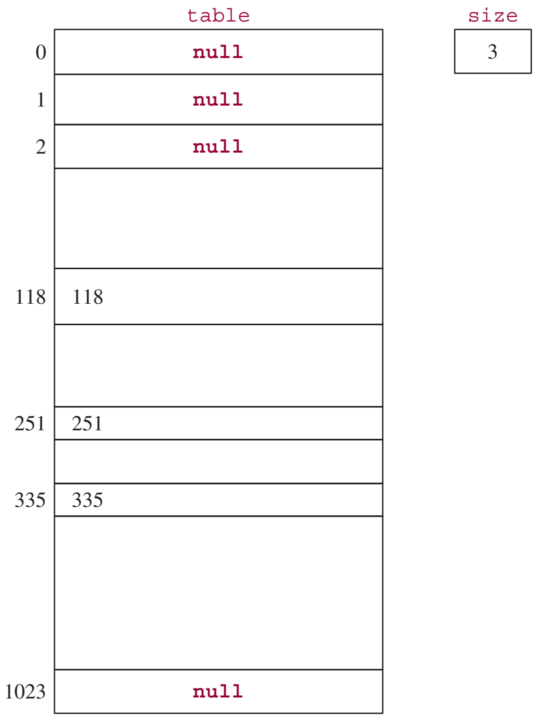


FIGURE 14.2 The `HashMap` object of Figure 14.1 after three elements have been inserted. For each non-null element, all that is shown is the **`int`** corresponding to the `Integer` key. The value parts are omitted.

Figure 14.3 shows the resulting `HashMap` object after the following two insertions:

```
persons.put (214303261, "Albert");
persons.put (033518000, "Schranz");
```

You might already have noticed a potential pitfall with this scheme: two distinct keys might produce the same index. For example, 214-30-3261 and 323-56-8157 both produce the index 541. Such a phenomenon is called a ***collision***, and the colliding keys are called ***synonyms***. We will deal with collisions shortly. For now we simply acknowledge that the possibility of collisions always exists when the size of the key space, that is, the number of legal key values, is larger than the table capacity.

Hashing is the process of transforming a key into an index in a table. One component of this transformation is the key class's `hashCode()` method. The purpose of this method is to perform some easily computable operation on the key object. The key class, which may be `Integer`, `String`, `FullTimeEmployee`, or whatever, usually defines its own `hashCode()` method. Alternatively, the key class can inherit a `hashCode()` method from one of its ancestors; ultimately, the `Object` class defines a `hashCode()` method.[1] Here is the definition of the `String` class's `hashCode()` method:

```
/**
 * Returns a hash code for this String object.
 *
```

[1]But the `hashCode()` method in the `Object` class will most likely return the reference itself, that is, the machine address, as an **`int`** . Then the `hashCode()` method applied to two equivalent objects would return different **`int`** values!

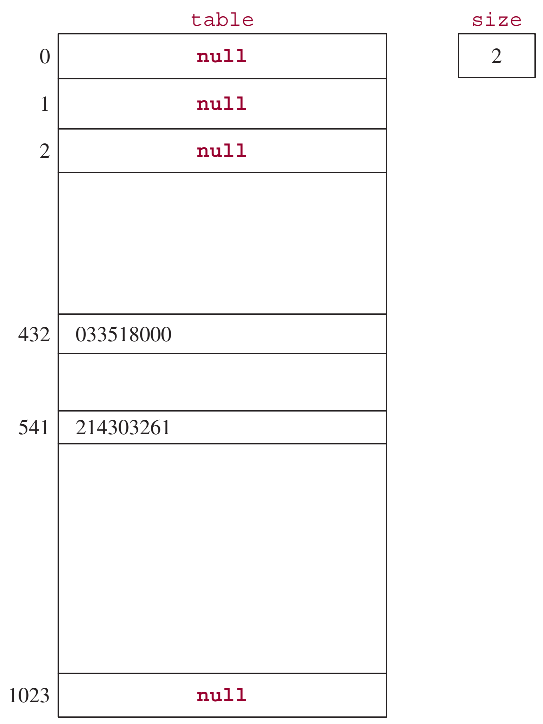


FIGURE 14.3 A HashMap object with two elements. The key is a social security number. The value is a name, but that is irrelevant, so it is not shown

```
   * @return  a hash code value for this String object.
   */
  public int hashCode()
  {
     int h = 0;
     int off = offset;      // index of first character in array value
     char val[ ] = value;   // value is the array of char that holds the String
     int len = count;       // count holds the number of characters in the String

     for (int i = 0; i < len; i++)
               h = 31*h + val[off++];

     return h;
  } // method hashCode
```

The multiplication of partial sums by 31 increases the likelihood that the **int** value returned will be greater than `table.length`, so the resulting indexes can span the entire table. For example, suppose we have

```
System.out.println ("graduate".hashCode());
```

The output will be

```
90004811
```

In the `HashMap` class, the `hashCode()` method for a key is supplemented by a static `hash` method whose only parameter is the `int` returned by the `hashCode()` method. The `hash` function scrambles that **`int`** value. In fact, the basic idea of hashing is to "make hash" out of the key. The additional scrambling is accomplished with a few right shifts and bit-wise "exclusive-or" operators (return 1 if the two bits are different, otherwise 0).

Here is the `hash` method:

```
static int hash(int h)
{
     h ^= (h >>> 20) ^ (h >>> 12);
     return h ^ (h >>> 7) ^ (h >>> 4);
} // method hash
```

For example,

```
"graduate".hashCode()
```

returns

```
90004811
```

and `hash (90004811)` returns

```
84042326
```

This value is bit-wise "anded" with `table.length - 1` to obtain an index in the range from 0 to `table.length - 1`. For example, if `table.length` = 1024,

```
84042326 & 1023
```

returns

```
598
```

And that is the index "graduate" hashes to. The overall strategy is

$$\texttt{key} \xrightarrow{\texttt{hash (key.hashCode()) \& table.length - 1}} \texttt{index}$$

One of the requirements of the `HashMap` class is that `table.length` must be a power of 2. (This requirement is specific to the Java Collections Framework, and does not apply generally.) For example, if `table.length` $= 1024 = 2^{10}$, then the index returned by

```
hash (key.hashCode()) & table.length - 1
```

consists of the rightmost 10 bits of `hash (key)`. Because a poor-quality `hashCode()` method may not adequately distinguish the rightmost bits in the keys, the additional scrambling from the bit-wise operators in the `hash` method is usually enough to prevent a large number of collisions from occurring.

You may have noticed that, because the table size is a power of 2, the result of

```
hash (key.hashCode()) & table.length - 1
```

is the same as the result of

```
hash (key.hashCode()) % table.length
```

The advantage of the former expression is that its calculation is quite a bit faster than for the latter expression. Modular arithmetic is computationally expensive, but is commonly utilized in hashing (outside of the Java Collections Framework) because its effectiveness does not require that the table size be a power of 2.

How would you go about developing your own hashCode() method? To see how this might be done, let's develop a simple hashCode() method for the FullTimeEmployee class from Chapter 1. The method should involve the two fields—name and grossPay—because these distinguish one FullTimeEmployee object from another. We have already seen the String class's hashCode() method. All of the wrapper classes, including Double, have their own hashCode() method, so we are led to the following method:

```
public int hashCode()
{
      return name.hashCode() + new Double (grossPay).hashCode();
} // method hashCode in class FullTimeEmployee
```

For example, if a FullTimeEmployee object has the name "Dominguez" and a gross pay of 4000.00, the value returned by the hashCode() method will be −319687574. When this hashCode() method is followed by the additional bit manipulation of the HashMap class's hash method, the result will probably appear to be quite random.

The final ingredient in hashing is the collision handler, which determines what happens when two keys hash to the same index. As you might expect, if a large number of keys hash to the same index, the performance of a hashing algorithm can be seriously degraded. Collision handling will be investigated in Section 14.3.3. But first, Section 14.3.2 reveals the major assumption of hashing: basically, that there will not be a lot of keys hashing to the same index.

14.3.2 The Uniform Hashing Assumption

For each value of key in an application, a table index is calculated as

```
int hash = hash (key.hashCode()),
   index = hash & table.length - 1;  // table.length will be a power of 2
```

Hashing is most efficient when the values of index are spread throughout the table. This notion is referred to as the ***Uniform Hashing Assumption***. Probabilistically speaking, the Uniform Hashing Assumption states that the set of all possible keys is uniformly distributed over the set of all table indexes. That is, each key is equally likely to hash to any one of the table indexes.

No class's hashCode method will satisfy the Uniform Hashing Assumption for all applications, although the supplemental scrambling in the hash function makes it extremely likely (but not guaranteed) that the assumption will hold.

Even if the Uniform Hashing Assumption holds, we must still deal with the possibility of collisions. That is, two distinct keys may hash to the same index. In the HashMap class, collisions are handled by a simple but quite effective technique called "chaining". Section 14.3.3 investigates chaining, and an alternate approach to collisions is introduced in Section 14.5.

14.3.3 Chaining

To resolve collisions, we store at each table location the singly-linked list of all elements whose keys have hashed to that index in table. This design is called ***chained hashing*** because the elements in each list form a chain. We still have the table and size fields from Section 14.3.1:

```
/**
 *  An array; at each index, store the singly-linked list of entries whose keys hash
```

```
     *  to that index.
     *
     */
    transient Entry[ ] table;

    /**
     *  The number of mappings in this HashMap object.
     */
    transient int size;
```

Each element is stored in an `Entry` object, and the embedded `Entry` class starts as follows:

```
    static class Entry<K,V> implements Map.Entry<K,V>
    {
           final K key;

           V value;

           final int hash;    // to avoid re-calculation

           Entry<K,V> next;   // reference to next Entry in linked list
```

The **`final`** modifier mandates that the `key` and `hash` fields can be assigned to only once. A singly-linked list, instead of the `LinkedList` class, is used to save space: There is no header, and no `previous` field. The `Entry` class also has a four-parameter constructor to initialize each of the fields.

To see how chained hashing works, consider the problem just stated of storing persons with social security numbers as keys. Since each key is (a reference to) an `Integer` object, the `hashCode()` method simply returns the corresponding **`int`**, but the `hash` method sufficiently garbles the key that the **`int`** returned seems random. Initially, each location contains an empty list, and insertions are made at the *front* of the linked list. Figure 14.4 shows what we would have after inserting elements with the following `Integer` keys (each is shown as an **`int`**, followed by the index the key hashed to) into a table of length 1024:

key	index
62488979	743
831947084	440
1917270349	911
1842336783	208
1320358464	440
1102282446	319
1173431176	440
33532452	911

The keys in Figure 14.4 were “rigged” so that there would be some collisions. If the keys were chosen randomly, it is unlikely there would have been any collisions, so there would have been eight linked lists, each with a single entry. Of course, if the number of entries is large enough, there will be collisions and multi-entry lists. And that raises an interesting question: Should the table be subject to re-sizing? Suppose the initial table length is 1024. If n represents the number of entries and n can continue to increase, the average size of each linked list will be $n/1024$, which is linear in n. But then, for searching, inserting, and removing, worstTime(n) will be linear in n—a far cry from the constant time that was promised earlier.

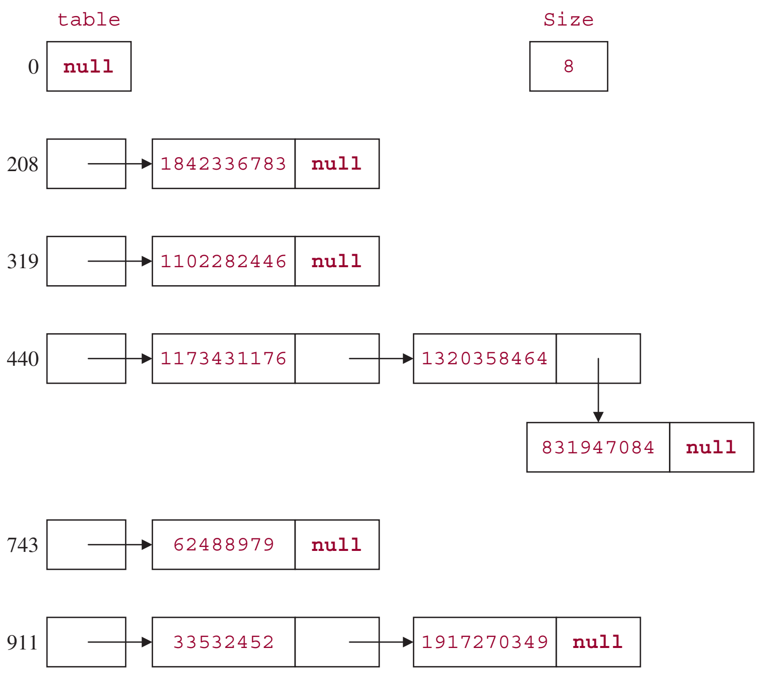


FIGURE 14.4 A `HashMap` object into which eight elements have been inserted. For each `Entry` object, only the key and `next` fields are shown. At all indexes not shown there is a **`null`** `Entry`.

Given that re-sizing must be done under certain circumstances, what are those circumstances? We will re-size whenever the size of the map reaches a pre-set threshold. In the `HashMap` class, the default threshold is 75% of `table.length`. So when `size` is >= `(int)(table.length * 0.75)`, the table will be re-sized. That means that the average size of each linked list will be less than one, and that is how constant average time can be achieved for inserting, deleting, and searching.

There are two additional fields relating to this discussion: `loadFactor` (how large can the ratio of `size` to table length get before resizing will occur) and `threshold` (how large can `size` get before resizing will occur). Specifically, we have

```
/**
 *  the maximum[2] ratio of  size / table.length before re-sizing will occur
 *
 */
final float loadFactor

/**
 *  (int)(table.length * loadFactor); when size++ >= threshold, re-size table
 *
 */
int threshold;
```

[2]This definition is non-standard. In hashing terminology, ***load factor*** is simply the ratio of the number of elements in the collection to its capacity. In the Framework's `HashMap` and `HashSet` classes, load factor is the upper bound of that ratio.

There are three constant identifiers related to `table` and `loadFactor`:

```
/**
 * The default initial capacity - MUST be a power of two.
 */
static final int DEFAULT_INITIAL_CAPACITY = 16;

/**
 * The maximum capacity, 2 to the 30th power, used if a higher value
 * is implicitly specified by either of the constructors with arguments.
 *
 */
static final int MAXIMUM_CAPACITY = 1 << 30; // = 2^30

/**
 * The load factor used when none specified in constructor.
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

The value for `loadFactor` presents an interesting time-space tradeoff. With a low value (say, less than 1.0), searches and removals are fast, and insertions are fast until an insertion triggers a re-sizing. That insertion will require linear-in-*n* time and space. On the other hand, with a high value for `loadFactor`, searches, removals, and insertions will be slower than with a low `loadFactor`, but there will be fewer re-sizings. Similary, if `table.length` is large, there will be fewer re-sizings, but more space consumed. Ultimately, the application will dictate the need for speed (how small a load factor) and the memory available (how large a table).

Now that we have decided on the fields in the `HashMap` and `Entry` classes, we can don our developer's hat to tackle the implementation of a couple of `HashMap` methods: a constructor and `containsKey`.

14.3.4 Implementation of the `HashMap` Class

All of the constructors deal with the load factor and the size of the table. The fundamental constructor—the one called by the others—is the following:

```
/**
 * Constructs an empty HashMap with the specified initial
 * capacity and load factor.
 *
 * @param  initialCapacity The initial capacity.
 * @param  loadFactor      The load factor.
 * @throws IllegalArgumentException if the initial capacity is negative
 *       or the load factor is nonpositive.
 */
public HashMap(int initialCapacity, float loadFactor)
{
     if (initialCapacity < 0)
         throw new IllegalArgumentException("Illegal initial capacity:  " +
                                             initialCapacity);
     if (initialCapacity > MAXIMUM_CAPACITY)
         initialCapacity = MAXIMUM_CAPACITY;
```

```
    if (loadFactor <= 0 || Float.isNaN(loadFactor))      // Not a Number
        throw new IllegalArgumentException("Illegal load factor:  " +
                                            loadFactor);
    // Find a power of 2 >= initialCapacity
    int capacity = 1;
    while (capacity < initialCapacity)
       capacity <<= 1;    // same as capacity = capacity << 1;

    this.loadFactor = loadFactor;
    threshold = (int)(capacity * loadFactor);
    table = new Entry[capacity];
    init();  // allows readObject to handle subclasses of HashMap (Appendix 1, A1.2)
}
```

This constructor is utilized, for example, in the definition of the default constructor:

```
    /**
     * Constructs an empty <tt>HashMap</tt> with the default initial capacity
     * (16) and the default load factor (0.75).
     */
    public HashMap()
    {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        threshold = (int)(DEFAULT_INITIAL_CAPACITY *
                          DEFAULT_LOAD_FACTOR);
        table = new Entry[DEFAULT_INITIAL_CAPACITY];
        init();
    }
```

We finish up this section by looking at the definition of the containsKey method. The other signature methods for a HashMap object—put, get, and remove—share the same basic strategy as containsKey: hash the key to an index in table, and then search the linked list at table [index] for a matching key. The containsKey method simply calls the getEntry method:

```
    /**
     *  Determines if this HashMap object contains a mapping for the
     *  specified key.
     *  The worstTime(n) is O(n).  If the Uniform Hashing Assumption holds,
     *  averageTime(n) is constant.
     *
     *  @param   key   The key whose presence in this HashMap object is to be tested.
     *
     *  @return true - if this map contains a mapping for the specified key.
     *
     */
    public boolean containsKey(Object key)
    {
      return getEntry(key) != null;
    }
```

Here is the definition of the `getEntry` method (the `indexFor` method simply returns `hash & table.length - 1`):

```
/**
  * Returns the entry associated with the specified key in the
  * HashMap. Returns null if the HashMap contains no mapping
  * for the key.
  */
final Entry<K,V> getEntry(Object key)
{
     int hash = (key == null) ? 0 : hash(key.hashCode());
     for (Entry<K,V> e = table[indexFor(hash, table.length)];
          e != null;
          e = e.next)
     {
         Object k;
         if (e.hash == hash  &&
             ((k = e.key) == key || (key != null  && key.equals(k))))
             return e;
     }
     return null;
}
```

Notice that both the `hash` and `key` fields are compared. Concept Exercise 14.4 indicates why both fields are checked.

Before we leave this section on the implementation of the `HashMap` class, there is an important detail about insertions that deserves mention. When a re-sizing takes place, that is, when

```
size++ >= threshold
```

the table is doubled in size. But the old entries cannot simply be copied to the new table: They must be re-hashed. Why? Because the index where an entry is stored is calculated as

```
hash  & table.length -1
```

When `table.length` changes, the index of the entry changes, so a re-hashing is required. For example,

```
hash ("myList")  & 1023                                  // 1023 = 2^10 -1
```

returns 231, but

```
hash ("myList")  & 2047                                 // 2047 = 2^11 -1
```

returns 1255.

After the keys have been re-hashed, subsequent calls to the `containsKey` method, for example, will search the linked list at the appropriate index.

In Section 14.3.5, we show that the `containsKey` method takes only constant time, on average.

14.3.5 Analysis of the `containsKey` Method

For the sake of brevity, we let $n =$ `size` and $m =$ `table.length`. We assume that the Uniform Hashing Assumption holds, that is, we expect the n elements to be fairly evenly distributed over the m lists. Then a successful search will examine a list that has about n/m elements[3], on average. A successful, sequential search of such a list requires, approximately, $n/2m$ loop iterations, so the average time appears to depend on both n and m. We get

$$\text{averageTime}_S(n, m) \approx n/2m \text{ iterations} <= \texttt{loadFactor / 2 < loadFactor}$$

The value of loadFactor is set in the HashMap object's constructor, and is fixed for the lifetime of the HashMap object. That is, averageTime$_S(n, m)$ is less than a constant. Therefore averageTime$_S(n, m)$ must be constant (in the "plain English" sense of Θ notation). In other words, the averageTime is independent of m (and even independent of n). So we conclude that averageTime$_S(n)$ is constant. Similarly, averageTime$_U(n)$ is constant. We will avoid the subscript and simply write averageTime(n) whenever the Θ estimates are the same for both successful and unsuccessful searches. See Section 14.5 for a situation where the Θ estimates are different.

So far we have blithely ignored any discussion of worstTime. This is the Achilles' heel of hashing, just as worstTime was the vulnerable aspect of Quick Sort. The Uniform Hashing Assumption is more often a hope than a fact. If the hashCode method is inappropriate for the keys in the application, the additional scrambling in the hash function may not be enough to prevent an inordinate number of keys to hash to just a few locations, leading to linear-in-n searches. Even if the Uniform Hashing Assumption holds, we could have a worst-case scenario: the given key hashes to index, and the number of keys at table [index] is linear in n. So worstTime (n, m), for both successful and unsuccessful searches, is linear in n. The independence from m allows us to write that worstTime(n) is linear in n for both successful and unsuccessful searches for the containsKey method.

Similarly, for the get and remove methods, worstTime(n) is linear in n. What about worstTime(n, m) for the put method? In the worst case, the size of the underlying table will be at the threshold, so we will need to double the size of the table, and then iterate through all m of the linked lists (many will be empty) to re-hash the n elements into the new table. So it appears that worstTime(n, m) is $\Theta(n + m)$. But at the threshold, $n/m =$ loadFactor (a constant), so $m = n/$loadFactor. Then worstTime(n, m) $=$ worstTime(n) $= \Theta(n + n/\texttt{loadFactor})$, since m is a function of n. We conclude that worstTime(n) is $\Theta(n)$, or in plain English, worstTime(n) is linear in n.

The bottom line is this: unless you are confident that the Uniform Hashing Assumption is reasonable for the key space in the application or you are not worried about linear-in-n worst time, use a TreeMap object, with its guarantee of logarithmic-in-n worst time for searching, inserting, and deleting.

The final topic in our introduction to the HashMap class is the HashIterator class.

14.3.6 The HashIterator Class

This section will clear up a mystery from when we first introduced the HashMap class: in what order are iterations performed? The answer is fairly straightforward. Starting at index table.length-1 and working down to index 0, the linked list at each index is traversed in sequential order. Starting at the back makes the continuation condition slightly more efficient: index > 0 (a constant) instead of index < table.length (a variable). In each singly-linked list, the sequential order is *last-in-first-out*. For example, Figure 14.5 (next page) is a repeat of the HashMap object from Figure 14.4.

[3] In common hashing terminology, the load factor is simply the ratio of n to m. In the Java Collections Framework, load factor is the maximum ratio of n to m before resizing takes place.

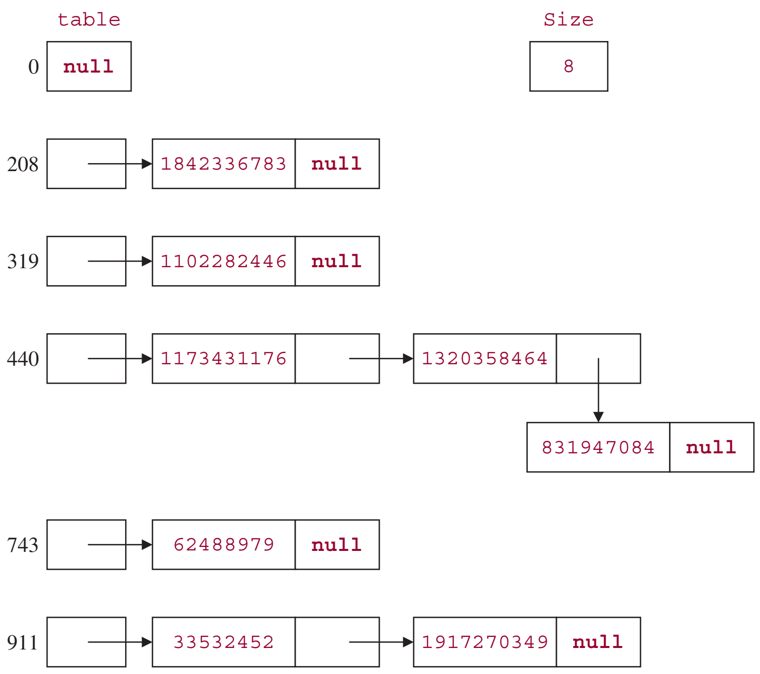


FIGURE 14.5

Here is the order in which the keys would appear in an iteration:

```
33532452
1917270349
62488979
1173431176
1320358464
831947084
1102282446
1842336783
```

Just as with the `TreeMap` class, the iteration can be by keys, by values, or by entries. For example:

```
for (K key   : myMap.keySet())
for (V value : myMap.values())
for (Map.Entry<K, V> entry  : myMap.entrySet())
```

Or, if the iteration might entail removals:

```
Iterator<K> itr1 = myMap.keySet().iterator();
Iterator<V> itr2 = myMap.values().iterator();
Iterator<Map.Entry<K, V>>  itr3 = myMap.entrySet().iterator();
```

An iteration through a `HashMap` object must peruse the singly linked list at each index in the table. Even if most of those linked lists are empty, each one must be checked. The total number of loop iterations is `size + table.length`, and this indicates a drawback to having a very large table. If the Uniform Hashing

Assumption holds, then the time estimates for the `next()` method are the same as for the `containsKey` method in the `HashMap` class. In the terminology of Section 14.3.5, for example,

$$\text{averageTime}_S(n, m) \approx (n + m)/n = 1 + m/n = 1 + 1/\texttt{loadFactor}$$

Since `loadFactor` is fixed, we conclude that averageTime(n, m) = averageTime(n) is constant for the `next()` method.

In the worst case, a call to the `next()` method will start at the end of the linked list at `table [table.length - 1]`, with empty linked lists at indexes `table.length - 2`, `table.length - 3`, . . ., 2, 1, and a non-empty linked list at `table [0]`. We conclude that worstTime(n, m) is linear in m.

If you would like to have control about the order in which elements in a hash map are iterated over, Programming Exercise 14.6 will show you the way.

In Section 14.3.7, we look at the premiere application of the `HashMap` class: the hashing of identifiers into a symbol table by a compiler.

14.3.7 Creating a Symbol Table by Hashing

Most compilers use hashing to create a ***symbol table***: an array that contains information about word-symbols in a program. A word-symbol can be either a reserved word or an identifier. An ***identifier*** is a name for a declared entity—such as a variable, method or class. An identifier consists of a letter, underscore or dollar sign, followed by any number of letters, digits, underscores or dollar signs. Here are some legitimate identifiers:

```
n
tax_2010
$stack_pointer
_left
```

The following are not legal identifiers

```
2010_tax    // an identifier must not start with a digit
time?       // an identifier must not contain a ?
while       // an identifier must not be a reserved word
```

Maintaining a symbol table is a good environment for hashing: Almost all of the activity consists of retrieving a symbol-table entry, given a key. And a reasonable upper bound on the number of identifiers is the size of the program, so we can initialize the size of the symbol table to avoid re-sizing.

To avoid getting bogged down in the myriad details of symbol-table creation and maintenance, we will solve the following restricted problem: Given a file of reserved words and a program in a source file, print to a file a symbol table of word symbols, with each word symbol designated as a reserved word or identifier. One interesting feature of this project is that the input is itself a program.

The main issue is how to separate identifiers from reserved words. The list of legal reserved words[4] is fixed, so we will read in a file that contains the reserved words before we read in the source file. For example, suppose the file "reserved.dat" contains the list, one per line, of reserved words in Java. And the file "Sample.java" contains the following program:

```
import java.io.*;

public class Sample
```

[4]Technically, **`true`**, **`false`**, and **`null`** are not reserved words. They are literals, that is, constant values for a type, just as "exhale" and 574 are literals. But we include them in with the reserved words because they cannot be used as identifiers.

```
{
      public static void main (String[ ] args)
      {
             int i = 75;
             Integer top,
                       bottom;
             /* all */ int z1 = 1;
             /* int z2 = 2;
                    int z3 = 3;
                    int z4 = 4;
             */
            int z5 = 5;
            /* int z6 = 6;*/
            String ans =  "All string literals, such as this, are ignored.";
            char x = 'x';
            for (int j = 0; j < i; j++)
                    System.out.println (i + "   " + j);
      } // method main
} // class Sample
```

System Test 1 (the input is boldfaced):

```
Please enter the path for the file that holds the reserved words: reserved.dat
Please enter the path for the file that holds the source code: Sample.java
Please enter the path for the file that will hold the symbol table: hasher.out
```

At the end of the execution of the program the file hasher.out will have:

```
Here is the symbol table:

ans=identifier
top=identifier
boolean=reserved word
interface=reserved word
rest=reserved word
for=reserved word
continue=reserved word
long=reserved word
abstract=reserved word
double=reserved word
instanceof=reserved word
println=identifier
throws=reserved word
super=reserved word
throw=reserved word
short=reserved word
do=reserved word
byte=reserved word
import=reserved word
if=reserved word
future=reserved word
package=reserved word
switch=reserved word
```

```
catch=reserved word
return=reserved word
x=identifier
outer=reserved word
String=identifier
System=identifier
z5=identifier
transient=reserved word
out=identifier
j=identifier
synchronized=reserved word
else=reserved word
args=identifier
while=reserved word
Double=identifier
goto=reserved word
_yes=identifier
var=reserved word
extends=reserved word
operator=reserved word
Integer=identifier
case=reserved word
final=reserved word
Sample=identifier
native=reserved word
null=reserved word
$name=identifier
float=reserved word
class=reserved word
implements=reserved word
private=reserved word
false=reserved word
main=identifier
char=reserved word
volatile=reserved word
const=reserved word
cast=reserved word
bottom=identifier
protected=reserved word
this=reserved word
static=reserved word
generic=reserved word
i=identifier
z1=identifier
void=reserved word
int=reserved word
byvalue=reserved word
break=reserved word
new=reserved word
default=reserved word
```

```
inner=reserved word
true=reserved word
public=reserved word
finally=reserved word
try=reserved word
```

We will filter out comments, string literals, and **`import`** statements. Then each line (what remains of it) will be tokenized. The delimiters will include punctuation, parentheses, and so on. For the sake of simplicity, both reserved words and identifiers will be saved in a symbol table, implemented as a `HashMap` object. Each key will be a word, either a reserved word or an identifier, and each value will be an indication of the word's type, either "reserved word" or "identifier".

We will create a `Hasher` class to solve this problem. The responsibilities of the `Hasher` class are as follows:

1. Scan the reserved words and hash each one (with "reserved word" as the value part) to the symbol table.
2. Scan the lines from the source file. For each identifier in each line, post the identifier (with "identifier" as the value part) to the symbol table, unless that identifier already appears as a key in the symbol table.
3. For each mapping in the symbol table, print the mapping to the output file.

14.3.7.1 Design and Testing of the `Hasher` Class

The responsibilities enunciated in Section 14.3.7 easily lead to method specifications. The `run()` method calls the following three methods (the `symbolTable` parameter is included for the sake of testing):

```
/**
 * Reads in the reserved words and posts them to the symbol table.
 *
 * @param reservedFileScanner - a Scanner object for the file that
 *                              contains the reserved words.
 * @param symbolTable - the HashMap that holds the symbol table.
 */
public void readReservedWords (Scanner reservedFileScanner,
                                  HashMap<String, String> symbolTable)

/**
 * Reads the source file and posts identifiers to the symbol table.
 * The averageTime(n, m) is O(n), and worstTime(n, m) is O(n * n), where
 *  n is the number of identifiers and m is the size of the symbol table.
 *
 *  @param sourceFileScanner -a scanner over the source file.
 *                              contains the reserved words.
 *  @param symbolTable - the HashMap that holds the symbol table.
 */
public void readSourceCode (Scanner sourceFileScanner,
 *                          HashMap<String, String> symbolTable)


/**
 * Outputs the symbol table to a file.
 * The worstTime(n, m) is O(n + m), where n is the number of word symbols
 * and m is the size of the symbol table.
 *
 * @param symbolTablePrintWriter - a PrintWriter object for the file that
 *                                  contains the symbol table.
 * @param symbolTable - the HashMap that holds the symbol table.
 */
public void printSymbolTable  (PrintWriter symbolTablePrintWriter,
                               HashMap<String, String> symbolTable)
```

Here is a test of ignoring string literals in the `readSourceCode` method:

```
@Test
public void testIgnoreStringLiterals() throws FileNotFoundException
{
    Scanner fileScanner = new Scanner (new File ("source.in1"));

    String s = hasher.readSourceCode (fileScanner);
    assertEquals (-1, s.indexOf ("This is a String literal."));
    assertEquals (-1, s.indexOf ("Heading"));
    assertEquals (-1, s.indexOf ("Salaries by Department: "));
} // method testIgnoreStringLiterals
```

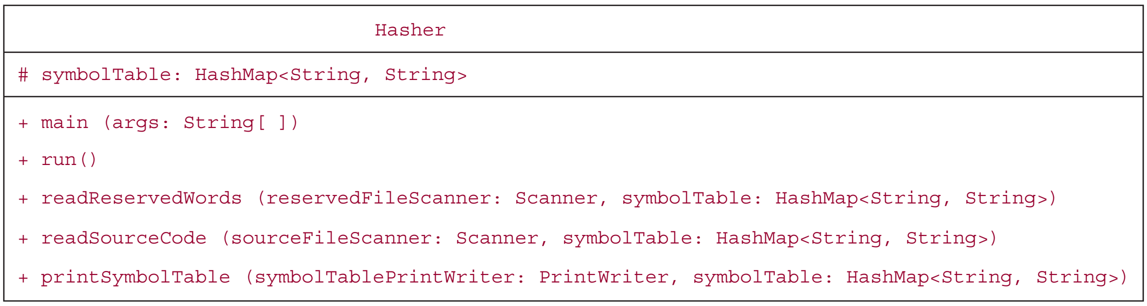
The file `source.in1` consists of the following:

```
String front = "Heading";

String nonsense = "This is a String literal.";

String report = "Salaries by Department: ";
```

The `Hasher` class has two constant identifiers and one field:

```
protected final String IDENTIFIER =  "identifier";

protected final String RESERVED_WORD =  "reserved word";

protected HashMap<String, String> symbolTable;
```

Figure 14.6 has the class diagram for the `Hasher` class:



```
Hasher
# symbolTable: HashMap<String, String>
+ main (args: String[ ])
+ run()
+ readReservedWords (reservedFileScanner: Scanner, symbolTable: HashMap<String, String>)
+ readSourceCode (sourceFileScanner: Scanner, symbolTable: HashMap<String, String>)
+ printSymbolTable (symbolTablePrintWriter: PrintWriter, symbolTable: HashMap<String, String>)
```

FIGURE 14.6

14.3.7.2 Implementation of the Hasher Class

The main method's definition is as expected. The definition of the run method is also straightforward, except for the initialization of the symbol table. To avoid the need for re-sizing, the symbol table is created with an initial capacity of the size of the source file. This number is returned by the length() method of a File object, so we first create (a reference to) such an object from the input path.

```
public void run()
{
   final String RESERVED_FILE_PROMPT =
        "\nPlease enter the path for the file that holds the reserved words:  "

   final String SOURCE_FILE_PROMPT =
        "\nPlease enter the path for the file that holds the source code:  ";

   final String SYMBOL_TABLE_FILE_PROMPT =
        "\nPlease enter the path for the output file that will hold the symbol table:  ";

   Scanner keyboardScanner,
           reservedFileScanner = null,
           sourceFileScanner = null;

   PrintWriter symbolTablePrintWriter = null;

   boolean pathsOK = false;

   while (!pathsOK)
   {
        try
        {
            keyboardScanner = new Scanner (System.in);

            System.out.print (RESERVED_FILE_PROMPT);
            String reservedFilePath = keyboardScanner.nextLine();
            reservedFileScanner = new Scanner (new File (reservedFilePath));

            System.out.print (SOURCE_FILE_PROMPT);
            String sourceFilePath = keyboardScanner.nextLine();
            File sourceFile = new File (sourceFilePath);
            sourceFileScanner = new Scanner (sourceFile);

            symbolTable = new HashMap<String, String> ((int)sourceFile.length());

            System.out.print (SYMBOL_TABLE_FILE_PROMPT);
            String symbolTablePrintPath = keyboardScanner.nextLine();
            symbolTablePrintWriter = new PrintWriter (
                                         new FileWriter (symbolTablePrintPath));
            pathsOK = true;
        } // try
        catch (Exception e)
        {
            System.out.println (e);
```

```
      } // catch
   } // while !pathsOK
   readReservedWords (reservedFileScanner, symbolTable);
   readSourceCode (sourceFileScanner, symbolTable);
   printSymbolTable (symbolTablePrintWriter, symbolTable);
}  // method run
```

The `reservedFileScanner`, `sourceFileScanner`, and `symbolTablePrintWriter` variables are initialized to **`null`** to avoid a compile-time error when they are used outside of the **`try`** block in which they are actually initialized.

The `readReservedWords` method loops through the file of reserved words and posts each one as a key in the symbol table; the value is the `String` “reserved word”.

```
public void readReservedWords (Scanner reservedFileScanner,
                                  HashMap<String, String> symbolTable)
{
    String reservedWord;

    while (true)
    {
          if (!reservedFileScanner.hasNextLine())
                break;
          reservedWord = reservedFileScanner.nextLine();
          symbolTable.put (reservedWord, RESERVED_WORD);
    } // while not end of file
} // method readReservedWords
```

It is easy enough to determine if a token is the first occurrence of an identifier: It must start with a letter, and not already be in `symbolTable` (recall that the reserved words were posted to `symbolTable` earlier). The hard part of the method is filtering out comments, such as in the following:

```
/* all */ int z1 = 1;
/* int z2 = 2;
   int z3 = 3;
   int z4 = 4; */ int z5 = 5; /* int z6 = 6;*/
```

Here is the method definition:

```
public String readSourceCode (Scanner sourceFileScanner,
                                  HashMap<String, String> symbolTable)
{
   final String DELIMITERS =  "[∧a-zA-Z0-9$_]+";

   String line,
          word;

   int start,
       finish;

   boolean skip = false ;

   while (true)
   {
       if (!sourceFileScanner.hasNextLine())
```

```
            break;
        line = sourceFileScanner.nextLine();
        line = line.trim();

        // Ignore lines beginning with  "import".
        if (line.indexOf("import  ") == 0)
           continue;    // start another iteration of this loop
        // Ignore string literals.
        while ((start = line.indexOf ("\"")) >= 0)
        {
             finish = line.indexOf("\"", 1 + start);
             while (line.charAt (finish - 1) == ‘\’)
                    finish = line.indexOf ("\"’’, finish + 1);
             line = line.substring(0, start) + line.substring(finish + 1);
        } // while line still has a string literal

        // Ignore // comments
        if ((start = line.indexOf("//")) >= 0)
           line = line.substring(0, start);

        // Ignore any line between /* and */.
        if ((line.indexOf ("*/") == -1)  && skip)
            continue;

        // Remove substring up to */ if matching /* on earlier line.
        if ((start = line.indexOf("*/")) >= 0  && skip)
        {
            skip = false;
            line = line.substring (start + 2);
        } // first part of line a comment

        // Handle lines that have /*.
        while ((start = line.indexOf ("/*")) >= 0)
            if ((finish = line.indexOf("*/", start + 2)) >= 0)
                line = line.substring(0, start) + line.substring(finish + 2);
            else
            {
                line = line.substring(0, start);
                skip = true;
            } // matching */ not on this line
        // Tokenize line to find identifiers.
        Scanner lineScanner = new Scanner (line).useDelimiter (DELIMITERS);
        while (lineScanner.hasNext())
        {
            word = lineScanner.next();
            if (!Character.isDigit (word.charAt (0))  &&
                     !symbolTable.containsKey (word) == null)
               symbolTable.put (word, IDENTIFIER);
        } // while not at end of line
    } // while not end of file
```

```
        return symbolTable.toString();
    } // method readSourceCode
```

Let n be the number of identifiers in the source file, and let m be the size of the source file. Because m is fixed, only n is relevant for estimating. Based on the analysis of containsKey method in Section 14.3.5, we extrapolate that each call to the HashMap methods get and put will take constant time on average, and linear-in-n time in the worst case. We conclude that for processing all n identifiers in the readSourceCode method, averageTime(n) is linear in n and worstTime(n) is quadratic in n.

Finally, the printSymbolTable method iterates through the entries in symbolTable and prints each one to the output file:

```
    public void printSymbolTable (PrintWriter symbolTablePrintWriter,
                                  HashMap<String, String> symbolTable)
    {
          final String HEADING =  "Here is the symbol table:\n";

          symbolTablePrintWriter.println (HEADING);
          for (Map.Entry<String, String> entry : symbolTable.entrySet())
                symbolTablePrintWriter.println (entry);
    } // method printSymbolTable
```

An iteration through a HashMap object requires $n + m$ iterations in all cases, so worstTime(n, m) and averageTime(n, m) are both $\Theta(n + m)$. We can say, crudely, that $n + m$ is both a lower bound and an upper bound of worstTime(n, m) and averageTime(n, m) for the printSymbolTable method.

One interesting feature of the above program is that Hasher.java itself can be the input file. The output file will then contain the symbol table of all word symbols in the Hasher class.

The word symbols in the output file are not in order. To remedy this, instead of printing out each entry, we could insert the entry into a TreeMap object, and then print out the TreeMap object, which would be in order. This sorting would take linear-logarithmic in n time; on average, that would be longer than the rest of the application.

To finish up the Java Collections Framework treatment of hashing, the next section takes a brief look at the HashSet class.

14.4 The HashSet Class

With hashing, all of the work involves the key-to-index relationship. It doesn't matter if the value associated with a key has meaning in the application or if each key is associated with the same dummy value. In the former case, we have a HashMap object and in the latter case, we have a HashSet object. The method definitions in the HashSet class are almost identical to those of the TreeSet class from Chapter 12. The major difference—besides efficiency—is that several of the constructor headings for the HashSet class involve the initial capacity and load factor.

Here is a program fragment that creates and maintains a HashSet instance in which the elements are of type String, the initial capacity is 100 and the load factor is 0.8F (that is, 0.8 as a **float**):

```
HashSet<String> names = new HashSet<String>(100, 0.8F);

names.add ("Kihei");
names.add ("Kula");
names.add ("Kaanapali");
System.out.println (names.contains ("Kapalua"));                   // Output: false
System.out.println (names.remove ("Kula") + "  " + names.size());   // Output: true 2
```

Lab 22 covers the crucial aspect of `HashMap` objects and `HashSet` objects: their speed.

You are now prepared to do Lab 22: Timing the Hash Classes

As indicated, the Java Collections Framework's implementation of hashing uses chaining to handle collisions. Section 14.5 explores another important collision handler, one that avoids linked lists.

14.5 Open-Address Hashing (optional)

To handle collisions with chaining, the basic idea is this: when a key hashes to a given `index` in `table`, that key's entry is inserted at the front of the linked list at `table [index]`. Each entry contains, not only `hash`, `key`, and `value` fields, but a `next` field that points to another entry in the linked list. The total number of `Entry` references is equal to `size` + `table.length`. For some applications, this number may be too large.

Open addressing provides another approach to collision handling. With ***open addressing***, each table location contains a single entry; there are no linked lists, and the total number of `Entry` references is equal to `table.length`. To insert an entry, if the entry's key hashes to an index whose entry contains a different key, the rest of the table is searched systematically until an empty—that is, "open"—location is found.

The simplest open-addressing strategy is to use an offset of 1. That is, to insert an entry whose key hashes to index `j`, if `table [j]` is empty, the entry is inserted there. Otherwise, the entry at index `j + 1` is investigated, then at index `j + 2`, and so on until an open slot is found. Figure 14.7 shows the table created when elements with the following `Integer` keys are inserted:

key	index
587771904	754
081903292	919
033520048	212
735668100	919
214303261	212
214303495	212
301336785	80
298719753	529

In this example, table.length is 1024, but in general, we will not require that the table length be a power of two. In fact, we will see in Section 14.5.2 that we may want the table length to be a prime number.

There are a couple of minor details with open addressing:

a. to ensure that an open location will be found if one is available, the table must wrap around: if the location at index `table.length - 1` is not open, the next index tried is 0.

b. the number of entries cannot exceed the table length, so the load factor cannot exceed 1.0. It will simplify the implementation and efficiency of the `containsKey`, `put`, and `remove` methods if the table always has at least one open (that is, empty) location. So we require that the load factor be strictly less than 1.0. Recall that with chaining, the load factor can exceed 1.0.

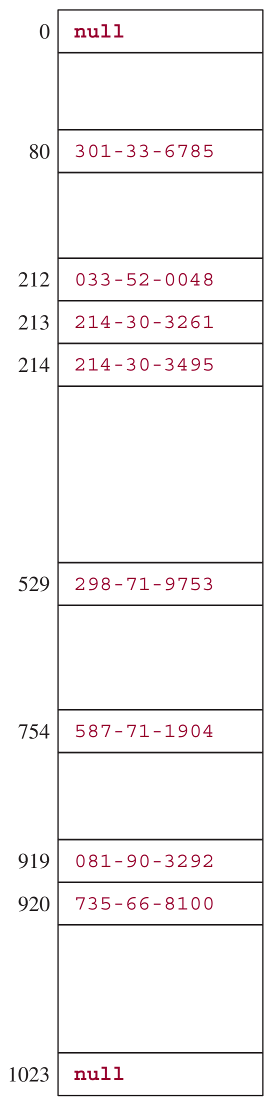


FIGURE 14.7 A table to which 8 elements have been inserted. Open addressing, with an offset of 1, handles collisions

Let's see what is involved in the design and implementation of a `HashMap` class with open addressing and an offset of 1. We'll have many of the same fields as in the chained-hashing design: `table`, `size`, `loadFactor`, and `threshold`. The embedded `Entry` class will have `hash`, `key`, and `value` fields, but no `next` field. We will focus on the `containsKey`, `put`, and `remove` methods: They will have to be re-defined because now there are no linked lists.

14.5.1 The `remove` Method

We need to consider the `remove` method before the `containsKey` and `put` method because the details of removing elements have a subtle impact on searches and insertions. To see what this is all about, suppose we want to remove the entry with key 033-52-0048 from the table in Figure 14.7. If we simply make that entry **`null`**, we will get the table in Figure 14.8.

Do you see the pitfall with this removal strategy? The path taken by synonyms of 033-52-0048 has been blocked. A search for the entry with key 214-30-3495 would be unsuccessful, even though there is such an entry in the table.

Instead of nulling out a removed entry, we will add another field to the `Entry` class:

```
boolean markedForRemoval;
```

This field is initialized to **`false`** when an entry is inserted into the table. The `remove` method sets this field to **`true`** . The `markedForRemoval` field, when **`true`** , indicates that its entry is no longer part of

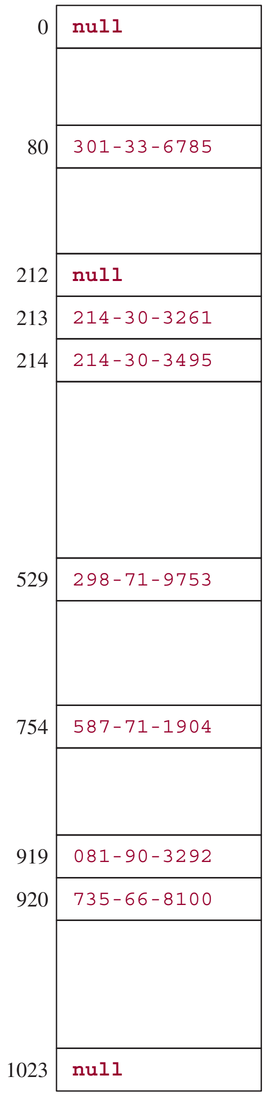


FIGURE 14.8 The effect of removing the entry with key 033-52-0048 from the table in Figure 14.7 by nulling out the entry at index 212

the hash map, but allows the offset-of-1 collision handler to continue along its path. Figure 14.9 shows a table after 8 insertions, and Figure 14.10 shows the subsequent effect of the message

```
remove (new Integer (033-52-0048))
```

A search for the entry with the key 214303495 would now be successful. In the definition of the `remove` method, a loop is executed until a **`null`** or matching entry is found. Note that an entry's `key` is examined only if that entry is not marked for removal.

The `containsKey` method loops until an empty or matching entry is found. As with the `remove` method, an entry's key is checked only if that entry is not marked for removal. The definition of the

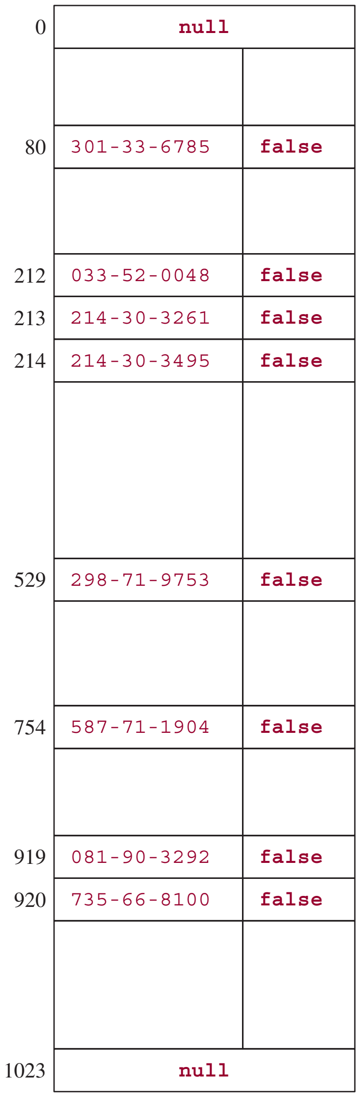


FIGURE 14.9 A table to which 8 elements have been inserted. Open addressing, with an offset of 1, handles collisions. In each entry, only the `key` and `markedForRemoval` fields are shown.

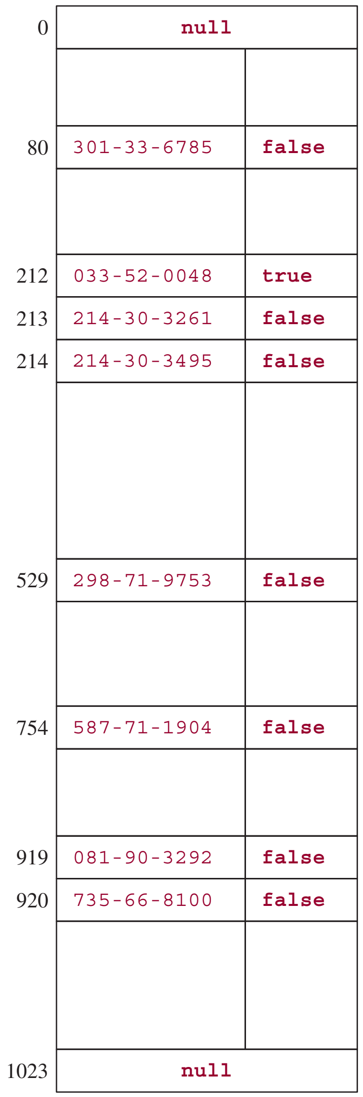


FIGURE 14.10 The table from Figure 14.9 after the message `remove (new Integer (033520048))` has been sent

`containsKey` method is only slightly revised from the chained-hashing version. For example, here we use modular arithmetic instead of the & operator because the table length need not be a power of 2.

```
/**
 *  Determines if this HashMap object contains a mapping for the
 *  specified key.
 *  The worstTime(n) is O(n).  If the Uniform Hashing Assumption holds,
 *  averageTime(n) is constant.
 *
 *  @param   key   The key whose presence in this HashMap object is to be tested.
 *
 *  @return true - if this map contains a mapping for the specified key.
 *
 */
public boolean containsKey (Object key)
{
      Object k = maskNull (key);  // use NULL_KEY if key is null
      int hash = hash (k);
      int i = indexFor (hash, table.length);
      Entry e = table [i];
      while (e != null)
      {
             if (!e.markedForRemoval  && e.hash == hash  && eq (k, e.key))
                    return true;
             e = table [(++i) % table.length]; // table.length may not be a power of 2
      } // while
      return false;
} // method containsKey
```

With the help of the `markedForRemoval` field, we solved the problem of removing an element without breaking the offset-of-1 path. The `put` method hashes a key to an index and stores the key and value at that index if that location is unoccupied: either a **`null`** key or marked for removal.

14.5.2 Primary Clustering

There is still a disturbing feature of the offset-of-1 collision handler: all the keys that hash to a given `index` will probe the same path: `index`, `index + 1`, `index + 2`, and so on. What's worse, all keys that hash to any index in that path will follow the same path from that index on. For example, Figure 14.11 shows part of the table from Figure 14.9:

In Figure 14.11, the path traced by keys that hash to 212 is 212, 213, 214, 215, ... And the path traced by keys that hash to 213 is 213, 214, 215, ... A ***cluster*** is a sequence of non-empty locations (assume the elements at those locations are not marked for removal). With the offset-of-1 collision handler, clusters are formed by synonyms, including synonyms from different collisions. In Figure 14.11, the locations at indexes 212, 213, and 214 form a cluster. As each entry is added to a cluster, the cluster not only gets bigger, but also grows faster, because any keys that hash to that new index will follow the same path as keys already stored in the cluster. ***Primary clustering*** is the phenomenon that occurs when the collision handler allows the growth of clusters to accelerate.

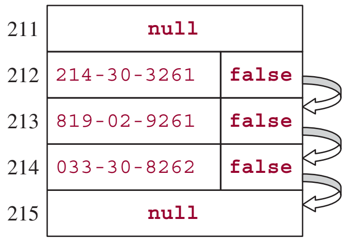


FIGURE 14.11 The path traced by keys that hash to 212 overlaps the paths traced by keys that hash to 213 or 214

Clearly, the offset-of-1 collision handler is susceptible to primary clustering. The problem with primary clustering is that we get ever-longer paths that are sequentially traversed during searches, insertions, and removals. Long sequential traversals are the bane of hashing, so we should try to solve this problem.

What if we choose an offset of, say, 32 instead of 1? We would still have primary clustering: keys that hashed to index would overlap the path traced by keys that hashed to index + 32, index + 64, and so on. In fact, this could create an even bigger problem than primary clustering. For example, suppose the table size is 128 and a key hashes to index 45. Then the only locations that would be allowed in that cluster have the following indexes:

45, 77, 109, 13

Once those locations fill up, there would be no way to insert any entry whose key hashed to any one of those indexes. The reason we have this additional problem is that the offset and table size have a common factor. We can avoid this problem by making the table size a prime number, instead of a power of 2. But then we would still have primary clustering.

14.5.3 Double Hashing

We can avoid primary clustering if, instead of using the same offset for all keys, we make the offset dependent on the key. Basically, the offset will be `hash/table.length`. There is a problem here: `hash` may be negative. To remedy this, we perform a bit-wise "and" with `hash` and the largest positive **`int`** value, written as `0x7FFFFFFF` in hexadecimal (base 16) notation. The result is guaranteed to be non-negative. The assignment for `offset` is

```
int offset = (hash  & 0x7FFFFFFF) / table.length;
```

And then, whether searching, inserting, or removing, replace

```
e = table [(++i) % table.length];  // offset of 1
```

with

```
e = table [(i + offset) % table.length];
```

To see how this works in a simple setting (ignoring the `hash` method), let's insert the following keys into a table of size 19:

```
33
72
71
55
112
109
```

These keys were created, not randomly, but to illustrate that keys from different collisions, even from the same collision, do not follow the same path. Here are the relevant moduli and quotients:

key	key % 19	key/19
33	14	1
72	15	3
71	14	3
112	17	5
55	17	2
109	14	5

The first key, 33, is stored at index 14, and the second key, 72, is stored at index 15. The third key, 71, hashes to 14, but that location is occupied, so the index 14 is incremented by the offset 3 to yield index 17; 71 is stored at that location. The fourth key, 112, hashes to 17 (occupied); the index 17 is incremented by the offset 5. Since 22 is beyond the range of the table, we try index 22% 19, that is, 3, an unoccupied location. The key 112 is stored at index 3. The fifth key, 55, hashes to 17 (occupied) and then to $(17 + 2)$ % 19, that is, 0, an empty location. The sixth key, 109, hashes to 14 (occupied) and then to $(14 + 5)$ % 19, that is 0 (occupied), and then to $(0 + 5)$ % 19, that is, 5, an unoccupied location.

Figure 14.12 shows the effect of the insertions:

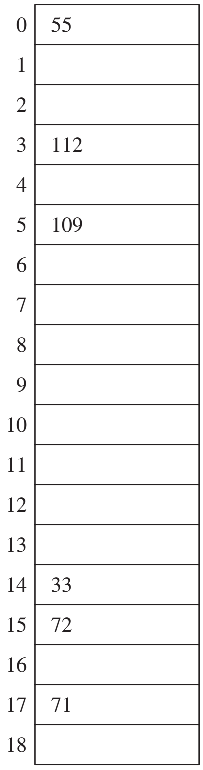


FIGURE 14.12 The effect of inserting six entries into a table; the collision handler uses the hash value divided by the table size as the offset

This collision handler is known as ***double hashing***, or the ***quotient-offset*** collision handler. There is one last problem we need to address before we get to the analysis: what happens if the offset is a multiple of the table size? For example, suppose we try to add the entry whose key is 736 to the table in Figure 14.12. We have

$$736 \,\%\, 19 = 14$$

$$736/19 = 38$$

Because location 14 is occupied, the next location tried is $(14 + 38)$ % 19, which is 14 again. To avoid this impasse, we use 1 as the offset whenever key/table.length is a multiple of table.length. This is an infrequent occurrence: it happens, on average, once in every m keys, where $m =$ `table.length`. Concept

Exercise 14.5 shows that if this collision handler is used and the table size is a prime number, the sequence of offsets from any key covers the whole table.

Because the table size must be a prime number, the call to the `resize` method—within the `put` method—must be changed from the offset-of-1 version:

```
resize (2 * table.length);
```

With the quotient-offset collision handler, we have

```
resize (nextPrime (2 * table.length));
```

The static method `nextPrime` returns the smallest prime larger than the argument.

If you undertake Programming Project 14.1, you will get to fill in the details of double hashing. If we make the Uniform Hashing Assumption, the average time for insertions, removals and searches is constant (see Collins [2003, pp 554–556]). Figure 14.13 summarizes the time estimates for successful and unsuccessful searches (that is, invocations of the `containsKey` method). For purposes of comparison, the information in Section 14.3.5 on chained hashing is included.

Figure 14.14 provides some specifics: the expected number of loop iterations for various ratios of n to m. For purposes of comparison, the information in Section 14.6 on chained hashing is included.

Chaining:

$$\text{averageTime}_S(n,m) \approx n/2m \text{ iterations}$$

$$\text{averageTime}_U(n,m) \approx n/m \text{ iterations}$$

Double Hashing:

$$\text{averageTime}_S(n,m) \approx (m/n)\ \ln(1/(1-n/m)) \text{ iterations}$$

$$\text{averageTime}_U(n,m) \approx 1/(1-n/m) \text{ iterations}$$

FIGURE 14.13 Estimates of average times for successful and unsuccessful calls to the `containsKey` method, under both chaining and double hashing. In the figure, n = number of elements inserted; m = `table.length`

n/m	0.25	0.50	0.75	0.90	0.99
Chained Hashing:					
successful	0.13	0.25	0.38	0.45	0.50
unsuccessful	0.25	0.50	0.75	0.90	0.99
Double Hashing:					
successful	1.14	1.39	1.85	2.56	4.65
unsuccessful	1.33	2.00	4.00	10.00	100.00

FIGURE 14.14 Estimated average number of loop iterations for successful and unsuccessful calls to the `containsKey` method, under both chained hashing and double hashing. In the figure, n = number of elements inserted; m = `table.length`

A cursory look at Figure 14.14 suggests that chained hashing is much faster than double hashing. But the figures given are mere estimates of the number of loop iterations. Run-time testing may, or may not, give a different picture. Run-time comparisons are included in Programming Project 14.1.

The main reason we looked at open-address hashing is that it is widely used; for example, some programming languages do not support linked lists. Also, open-address hashing can save space relative to chained hashing, which requires $n + m$ references versus only m references for open-address hashing.

Even if the Uniform Hashing Assumption applies, we could still, in the worst case, get every key to hash to the same index and yield the same offset. So for the `containsKey` method under double hashing, $\text{worstTime}_S(n, m)$ and $\text{worstTime}_U(n, m)$ are linear in n.

For open addressing, we eliminated the threat of primary clustering with double hashing. Another way to avoid primary clustering is through ***quadratic probing***: the sequence of offsets is $1^2, 2^2, 3^2, \ldots$. For details, the interested reader may consult Weiss [2002].

SUMMARY

In this chapter, we studied the Java Collections Framework's `HashMap` class, for which key-searches, insertions, and removals can be very fast, on average. This exceptional performance is due to ***hashing***, the process of transforming a key into an index in a table. A hashing algorithm must include a collision handler for the possibility that two keys might hash to the same index. A widely used collision handler is chaining. With ***chaining***, the `HashMap` object is represented as an array of singly linked lists. Each list contains the elements whose keys hashed to that index in the array.

Let n represent the number of elements currently in the table, and let m represent the capacity of the table. The ***load factor*** is the maximum ratio of n to m before rehashing will take place. The load factor is an upper-bound estimate of the average size of each list, assuming that the `hash` method scatters the keys uniformly throughout the table. With that assumption—the **Uniform Hashing Assumption**—the average time for successful and unsuccessful searches depends only on the ratio of n to m. The same is true for insertions and removals. If we double the table size whenever the ratio of n to m equals or exceeds the load factor, then the size of each list, on average, will be less than or equal to the load factor. This shows that with chained hashing, the average time to insert, remove, or search is constant.

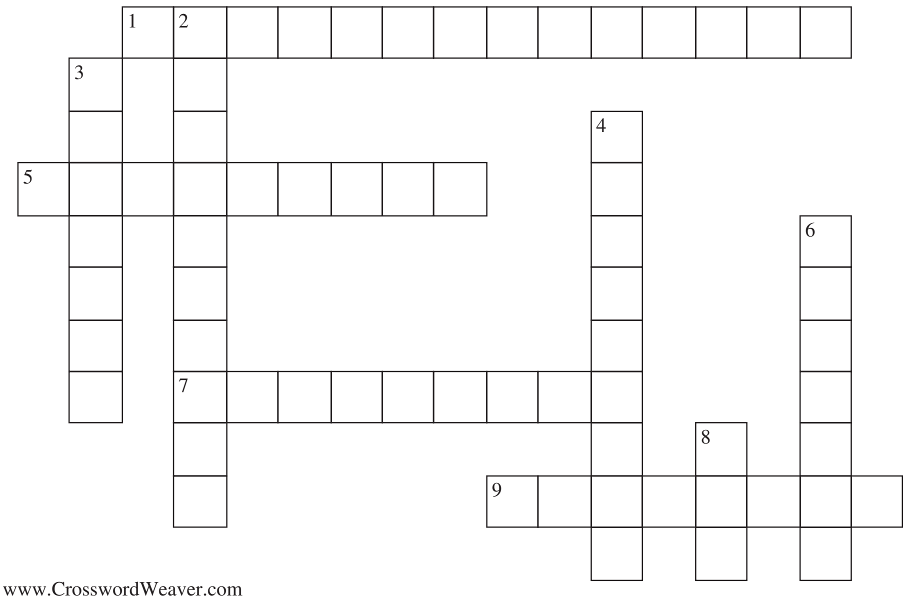
A `HashSet` object is simply a `HashMap` in which each key has the same dummy value. Almost all of the `HashSet` methods are one-liners that invoke the corresponding `HashMap` method.

An alternative to chaining is open addressing. When open addressing is used to handle collisions, each location in the table consists of entries only; there are no linked lists. If a key hashes to an index at which another key already resides, the table is searched systematically until an open address, that is, empty location is found. With an offset of 1, the sequence searched if the key hashes to `index`, is

```
index, index + 1, index + 2, ...,
table.length - 1, 0, 1, ..., index - 1.
```

The offset-of-1 collision handler is susceptible to ***primary clustering***: the phenomenon of having accelerating growth in the size of collision paths. Primary clustering is avoided with ***double hashing***: the offset is the (positivized) hash value divided by `table.length`. If the Uniform Hashing Assumption holds and the table size is prime, the average time for both successful and unsuccessful searches with double hashing is constant.

CROSSWORD PUZZLE



ACROSS

1. An alternative to chained hashing is _____ hashing.

5. For any iteration through a `HashMap` object worstTime(*n*, *m*) for the `next()` method is _______.

7. `(int)(table.length * loadFactor)`

9. The technique for resolving collisions by storing at each table location the linked list of all elements whose keys hashed to that index in the table

DOWN

2. The capacity of a `HashMap` object must be a _________.

3. The ______ Hashing Assumption states that the set of all possible keys is uniformly distributed over the set of all table indexes.

4. The interface implemented by the `TreeMap` class but not by the `HashMap` class

6. The process of transforming a key into a table index

8. The type of the value returned by the `hashCode()` method

CONCEPT EXERCISES

14.1 Why does the `HashMap` class use singly-linked lists instead of the `LinkedList` class?

14.2 Suppose you have a `HashMap` object, and you want to insert an element unless it is already there. How could you accomplish this?

Hint: The `put` method will insert the element even if it is already there (in which case, the new value will replace the old value).

14.3 For each of the following methods, estimate averageTime(n) and worstTime(n):

a. making a successful call—that is, the element was found—to the `contains` method in the `LinkedList` class;

b. making a successful call to the `contains` method in the `ArrayList` class;

c. making a successful call to the generic algorithm `binarySearch` in the `Arrays` class; assume the elements in the array are in order.

d. making a successful call to the `contains` method in the `BinarySearchTree` class;

e. making a successful call to the `contains` method in the `TreeSet` class;

f. making a successful call to the `contains` method in the `HashSet` class—you should make the Uniform Hashing Assumption.

14.4 This exercise helps to explain why both the `hash` and `key` fields are compared in the `containsKey` and `put` (and `remove`) methods of the `HashMap` class.

a. Suppose the keys in the `HashMap` are from a class, `Key`, that overrides the `Object` class's `equals` method but does not override the `Object` class's `hashCode` method. (This is in violation of the contract for the `Object` class's `hashCode` method, namely, equal objects should have the same hash code). The `Object` class's `hashCode` method converts the `Object` reference to an **`int`** value. So it would be possible for a key to be constructed with a given reference, and an identical key constructed with a different reference. For example, we could have:

```
Key key1 = new Key ("Webucation"),
    key2 = new Key ("Webucation");

HashMap<Key, String> myMap = new HashMap<Key, String>();

myMap.put (key1, "");    // the value part is the empty String
```

If the `hash` fields were *not* compared in the `containsKey` and `put` methods, what would be returned by each of the following messages:

```
myMap.containsKey (key2)

myMap.put (key2, "")
```

b. In some classes, the `hashCode` method may return the same **`int`** value for two distinct keys. For example, in the `String` class, we can have two distinct `String` objects—even with the same characters—that have the same hash code:

```
String key1 = "! @";  // exclamation point, blank, at sign

String key2 = " @!";  // blank, at sign, exclamation point

HashMap<String, String> myMap = new HashMap<String, String>();

myMap.put (key1, "");   // the value part is the empty String

System.out.println (key1.hashCode() + " " + key2.hashCode());
```

The output will be

```
32769 32769
```

If the `key` fields were *not* compared in the `containsKey` and `put` methods, what would be returned by each of the following messages:

```
myMap.containsKey (key2)

myMap.put (key2, "")
```

14.5 Assume that p is a prime number. Use modular algebra to show that for any positive integers *index* and *offset* (with offset not a multiple of p), the following set has exactly p elements:

$$\{(\text{index} + \text{k} * \text{offset})\% \ \text{p};\ \text{k} \ = 0, 1, 2, \ldots, \ \text{p} \ - 1\}$$

14.6 Compare the space requirements for chained hashing and open-address hashing with quotient offsets. Assume that a reference occupies four bytes and a **`boolean`** value occupies one byte. Under what circumstances (`size`, `loadFactor`, `table.length`) will chained hashing require more space? Under what circumstances will double hashing require more space?

14.7 We noted in Chapter 12 that the term ***dictionary***, viewed as an arbitrary collection of key-value pairs, is a synonym for ***map***. If you were going to create a real dictionary, would you prefer to store the elements in a `TreeMap` object or in a `HashMap` object? Explain.

14.8 In open-addressing, with the quotient-offset collision handler, insert the following keys to a table of size 13 (ignore the `hash` method):

20

33

49

22

26

202

140

508

9

Here are the relevant remainders and quotients:

key	key % 13	key/13
20	7	1
33	7	2
49	10	3
22	9	1
26	0	2
202	7	15
140	10	10
508	1	39
9	9	0

PROGRAMMING EXERCISES

14.1 Construct a `HashMap` object of the 25,000 students at Excel University. Each student's key will be that student's unique 6-digit ID. Each student's value will be that student's grade point average. Which constructor should you use and why?

14.2 Construct a `HashMap` object of the 25,000 students at Excel University. Each student's key will be that student's unique 6-digit ID. Each student's value will be that student's grade point average and class rank. Insert a few elements into the `HashMap` object. Note that the value does not consist of a single component.

14.3 Construct a `HashSet` object with `Integer` elements and an initial capacity of 2000. What is the load factor? What is the table length (**Hint**: It is greater than 2000)? Insert a few random `Integer` elements into the `HashSet` object.

14.4 As a programmer who *uses* the `HashSet` class, test and define test a `toSortedString` method:

```
/**
 * Returns a String representation of a specified HashSet object, with the natural
 * ordering,
 * The worstTime(n) is O(n log n).
 *
 * @param hashSet –the HashSet object whose String representation, sorted, is
 *        to be returned.
 *
 * @return a String representation of hashSet, with the natural ordering.
 *
 */
public static <E> String toSortedString (HashSet<E> hashSet)
```

Note: As a user, you cannot access any of the fields in a `HashSet` class.

14.5 Given the following program fragment, add the code to print out all of the keys, and then add the code to print out all of the values:

```
HashMap<String, Integer> ageMap = new HashMap<String, Integer>();
```

```
ageMap.put ("dog", 15);
ageMap.put ("cat", 20);
ageMap.put ("turtle", 100);
ageMap.put ("human", 75);
```

14.6 The `LinkedHashMap` class superimposes a `LinkedList` object on a `HashMap` object, that is, there is a doubly-linked list of all of the elements in the `HashMap` object. For example, suppose we have

```
LinkedHashMap<String, Integer> ageMap =
             new LinkedHashMap<String, Integer>();

ageMap.put ("dog", 15);
ageMap.put ("cat", 20);
ageMap.put ("turtle", 100);
ageMap.put ("human", 75);

System.out.println (ageMap);
```

The output will reflect the order in which elements were inserted, namely,

```
{dog=15, cat=20, turtle=100, human=75}
```

Revise the above code segment so that only the pets are printed, and in alphabetical order.

Programming Project 14.1

The Double Hashing Implementation of the `HashMap` Class

Test and develop the implementation of the `HashMap` class that uses double hashing—see Sections 14.5 and following. Define only a small number of methods: a default constructor, a constructor with an initial-capacity parameter, `containsKey`, `put`, and `remove`. The `toString` method is inherited from `AbstractMap`.

For system tests, run the `Hasher` program with input files `Sample.java` and `Hasher.java` itself. Use your `HashMap` class for Lab 22 and compare the times you get to the times you got in that lab.

Programming Project 14.2

An Integrated Web Browser and Search Engine, Part 6

At this point, your search engine created a `TreeMap` object in which each element was a web page: the key was a word, and the value was the number of occurrences of the word. This approach worked well for a small number of web pages, but would not scale up to the billions (even trillions) of web pages. In fact, the approach would be infeasible even if we used a `HashMap` object instead of a `TreeMap` object.

To enable searches to be completed quickly, a search engine must have a web crawler that operates all the time. The web crawler more or less randomly searches web pages and saves ("caches") the results of the searches. For example, for each possible search word, each file in which the search word occurs and the frequency of occurrences are saved. When an end-user clicks on the Search button and then enters a search string, the cached results allow for quick searches.

Initially, you will be given a file, "search.ser", that contains a search-string `HashMap` in serialized form (see Appendix 1 for details on serializing and de-serializing). Each key—initially—is a search word, and each value is a file-count `HashMap` in which the key is a file name and the value is the frequency of the search word in the file.

You will need to de-serialize the search-string `HashMap`. When the end-user enters a search string, there are two possibilities. If the search string already appears as a key in the search-string `HashMap`, the file names and counts in the file-count `HashMap` are put into a priority queue and Part 5 of the project takes over. If the search string does not appear as a key in the search-string `HashMap`, the individual words in the search string are used to search the search-string `HashMap`, and the combined result becomes a new entry in the search-string `HashMap`.

For example, suppose the universe of web pages consists of the files browser.in10, in11, in12, and in13 from Part 5 (Programming Project 13.2). The search-string `HashMap` starts out with keys that include "neural", "decree", and "network". If the end-user clicks on the Search button and then enters "network", the output will be

```
Here are the results of the old search for network:

browser.in13  4
browser.in12  4
browser.in11  4
browser.in10  4
```

But if the end-user searches for "neural network", that string is not in the search-string `HashMap`. So the results from the individual words "neural" and "network" are combined, and the output will be

```
Here are the results of the new search for neural network:

browser.in13  8
browser.in10  8
browser.in12  6
browser.in11  6
```

The search string "neural network" and the results will now be added to the search-string `HashMap`. If, later in the same run, the end-user searches for "neural network", the output will be

```
Here are the results of the old search for neural network:

browser.in13  8
browser.in10  8
browser.in12  6
browser.in11  6
```

System Test 1:

(Assume the end-user searches for "neural network".)

(continued on next page)

(continued from previous page)

Here are the results of the new search for neural network:

```
browser.in13  8
browser.in10  8
browser.in12  6
browser.in11  6
```

(Assume the end-user searches for "network".)

Here are the results of the old search for network:

```
browser.in13  4
browser.in12  4
browser.in11  4
browser.in10  4
```

(Assume the end-user searches for "neural network".)

Here are the results of the old search for neural network:

```
browser.in13  8
browser.in10  8
browser.in12  6
browser.in11  6
```

System Test 2:

(Assume the end-user searches for "network decree".)

Here are the results of the new search for network decree:

```
browser.in12  6
browser.in13  5
browser.in11  5
browser.in10  5
```

NOTE 1: By the end of each execution of the project, the file "search.ser" should contain the updated search-string `HashMap` in serialized form. For example, by the end of System Test 1, "neural network" will be one of the search strings serialized in "search.ser". You may update "search.ser" after each search.

NOTE 2: The original file search.ser is available from the ch14 directory of the book's website.

NOTE 3: If the search string does not occur in any of the files, the output should be "The search string does not occur in any of the files."

Graphs, Trees, and Networks

CHAPTER 15

There are many situations in which we want to study the relationship between objects. For example, in Facebook, the objects are individuals and the relationship is based on friendship. In a curriculum, the objects are courses and the relationship is based on prerequisites. In airline travel, the objects are cities; two cities are related if there is a flight between them. It is visually appealing to describe such situations graphically, with points (called *vertices*) representing the objects and lines (called *edges*) representing the relationships. In this chapter, we will introduce several collections based on vertices and edges. Finally, we will design, test, and implement one of those collections in a class, `Network`, from which the other structures can be defined as subclasses. That class uses several classes—`TreeMap`, `PriorityQueue,` and `LinkedList`—that are in the Java Collections Framework. But neither the `Network` class nor the subclasses are currently part of the Java Collections Framework.

CHAPTER OBJECTIVES

1. Define the terms ***graph*** and ***tree*** for both directed/undirected and weighted/unweighted collections.
2. Compare breadth-first iterators and depth-first iterators.
3. Understand Prim's greedy algorithm for finding a minimum-cost spanning tree and Dijkstra's greedy algorithm for finding the minimum-cost path between vertices.
4. Be able to find critical paths in a project network.
5. Be able to utilize, expand, and extend the `Network` class.

15.1 Undirected Graphs

An ***undirected graph*** consists of a collection of distinct elements called ***vertices***, connected to other elements by distinct vertex-pairs called ***edges***. Here is an example of an undirected graph:

Vertices: A, B, C, D, E

Edges: (A,B), (A,C), (B,D), (C,D), (C,E)

The vertex pair in an edge is enclosed in parentheses to indicate the pair of vertices is unordered. For example, to say there is an edge from A to B is the same as saying there is an edge from B to A. That's why "undirected" is part of the term defined. Figure 15.1 depicts this undirected graph, with each edge represented as a line connecting its vertex pair.

From the illustration in Figure 15.1 we could obtain the original formulation of the undirected graph as a collection of vertices and edges. And furthermore, Figure 15.1 gives us a better grasp of the undirected

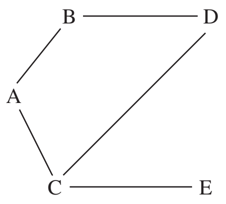


FIGURE 15.1 A visual representation of an undirected graph

graph than the original formulation. From now on we will use illustrations such as Figure 15.1 instead of the formulations.

Figure 15.2a contains several additional undirected graphs. Notice that the number of edges can be fewer than the number of vertices (Figures 15.2a and b), equal to the number of vertices (Figure 15.1) or greater than the number of vertices (Figure 15.2c).

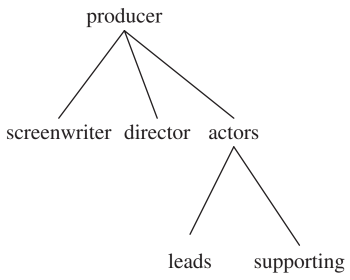


FIGURE 15.2a An undirected graph with six vertices and five edges

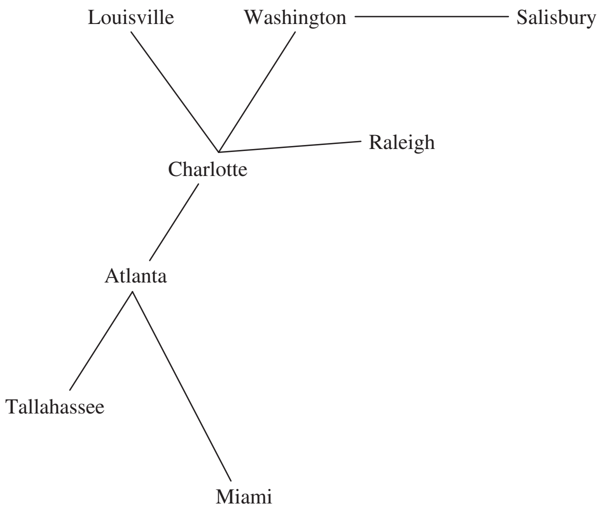


FIGURE 15.2b An undirected graph with eight vertices and seven edges

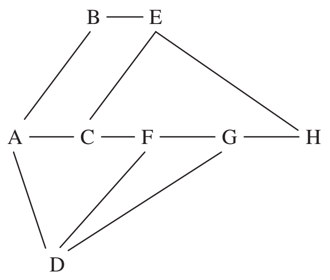


FIGURE 15.2c An undirected graph with eight vertices and eleven edges

An undirected graph is ***complete*** if it has as many edges as possible. What is the number of edges in a complete undirected graph? Let n represent the number of vertices. Figure 15.3 shows that when $n = 6$, the maximum number of edges is 15.

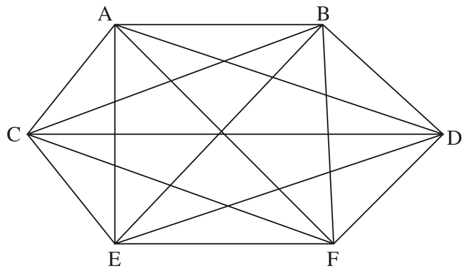


FIGURE 15.3 An undirected graph with 6 vertices and the maximum number (15) of edges for any undirected graph with 6 vertices

Can you determine a formula that holds for any positive integer n? In general, start with any one of the n vertices, and construct an edge to each of the remaining $n - 1$ vertices. Then from any one of those $n - 1$ vertices, construct an edge to each of the remaining $n - 2$ vertices (the edge to the first vertex was constructed in the previous step). Then from any one of those $n - 2$ vertices, construct an edge to each of the remaining $n - 3$ vertices. This process continues until, at step $n - 1$, a final edge is constructed. The total number of edges constructed is:

$$(n-1)+(n-2)+(n-3)+\ldots+2+1=\sum_{i=1}^{n-1} i = n(n-1)/2$$

This last equality can either be proved directly by induction on n or can be derived from the proof—in Example A2.1 of Appendix 2—that the sum of the first n positive integers is equal to $n\ (n + 1)/2$.

Two vertices are ***adjacent*** if they form an edge. For example, in Figure 15.2b, Charlotte and Atlanta are adjacent, but Atlanta and Raleigh are not adjacent. Adjacent vertices are called ***neighbors***.

A ***path*** is a sequence of vertices in which each successive pair is an edge. For example, in Figure 15.2c,

A, B, E, H

is a path from A to H because (A, B), (B, E), and (E, H) are edges. Another path from A to H is

A, C, F, D, G, H

For a path of k vertices, the ***length*** of the path is $k - 1$. In other words, the path length is the number of *edges* in the path. For example, in Figure 15.2c the following path from C to A has a length of 3:

C, F, D, A

There is, in fact, a path with fewer edges from C to A, namely,

C, A

In general, there may be several paths with fewest edges between two vertices. For example, in Figure 15.2c,

A, B, E

and

A, C, E

are both paths with fewest edges from A to E.

A ***cycle*** is a path in which the first and last vertices are the same and there are no repeated edges. For example, in Figure 15.1,

A, B, D, C, A

is a cycle. In Figure 15.2c,

B, E, H, G, D, A, B

is a cycle, as is

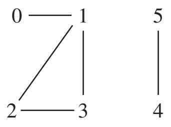
E, C, A, B, E

The undirected graph in Figures 15.2a and b are ***acyclic***, that is, they do not contain any cycles. In Figure 15.2a,

producer, director, producer

is *not* a cycle since the edge (producer, director) is repeated—recall that an edge in an undirected graph is an unordered pair.

An undirected graph is ***connected*** if, for any given vertex, there is a path from that vertex to any other vertex in the graph. Informally, an undirected graph is connected if it is "all one piece." For example, all of the graphs in the previous figures are connected. The following undirected graph, with six vertices and five edges, is not connected:



15.2 Directed Graphs

Up to now we have not concerned ourselves with the direction of edges. If we could go from vertex V to vertex W, we could also go from vertex W to vertex V. In many situations this assumption may be unrealistic. For example suppose the edges represent streets and the vertices represent intersections. If the street connecting vertex V to vertex W is one-way going from V to W, there would be no edge connecting W to V.

A ***directed graph***—sometimes called a ***digraph***—is a collection of vertices and edges in which the edges are **ordered** pairs of vertices. For example, here is a directed graph, with edges in *angle* brackets to indicate an ordered pair:

Vertices: A, T, V, W, Z

Edges: ⟨A, T⟩, ⟨A, V⟩, ⟨T, A⟩, ⟨V, A⟩, ⟨V, W⟩, ⟨W, Z⟩, ⟨Z, W⟩, ⟨Z, T⟩

Pictorially, these edges are represented by arrows, with the arrow's direction going from the first vertex in the ordered pair to the second vertex. For example, Figure 15.4 contains the directed graph just defined.

A path in a directed graph must follow the direction of the arrows. Formally a ***path*** in a directed graph is a sequence of $k > 1$ vertices $V_0, V_1, \ldots, V_{k-1}$ such that $\langle V_0, V_1\rangle, \langle V_1, V_2\rangle, \ldots, \langle V_{k-2}, V_{k-1}\rangle$ are edges in the directed graph. For example, in Figure 15.4,

A, V, W, Z

is a path from A to Z because $\langle A, V\rangle, \langle V, W\rangle$, and $\langle W, Z\rangle$ are edges in the directed graph. But

A, T, Z, W

is not a path because there is no edge from T *to* Z, (In other words, Z is *not* a neighbor of T, although T *is* a neighbor of Z.) A few minutes checking should convince you that for any two vertices in Figure 15.4, there is a path from the first vertex to the second.

A digraph D is ***connected*** if, for any pair of distinct vertices x and y, there is a path from x to y. Figure 15.4 is a connected digraph, but the digraph in Figure 15.5 is not connected (try to figure out why):

From these examples, you can see that we actually could have defined the term "undirected graph" from "directed graph": an ***undirected graph*** is a directed graph in which, for any two vertices V and W,

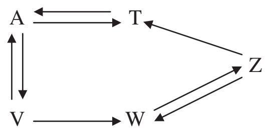


FIGURE 15.4 A directed graph

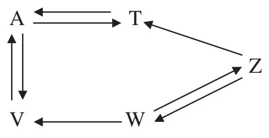


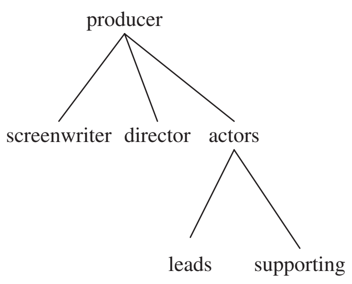
FIGURE 15.5 A digraph that is not connected

if there is an edge from V to W, there is also an edge from W to V. This observation will come in handy when we get to developing Java classes.

In the next two sections, we will look at specializations of graphs: trees and networks.

15.3 Trees

An ***undirected tree*** is a connected, acyclic, undirected graph with one element designated as the ***root element***. Note that any element in the graph can be designated as the root element. For example, here is the undirected tree from Figure 15.2a; producer is designated as the root element.



On most occasions, we are interested in directed trees, that is, trees that have arrows from a parent to its children. A ***tree***, sometimes called a ***directed tree***, is a directed graph that either is empty or has an element, called the ***root element*** such that:

a. there are no edges coming into the root element;

b. every non-root element has exactly one edge coming into it;

c. there is a path from the root to every other element.

For example, Figure 15.6 shows that we can easily re-draw the undirected tree as a directed tree:

Many of the binary-tree terms from Chapter 9—such as “child”, “leaf”, “branch”—can be extended to apply to arbitrary trees. For example, the tree in Figure 15.6 has four leaves and height 2. But “full” does not apply to trees in general because there is no limit to the number of children a parent can have. In fact, we cannot simply define a binary tree to be a tree in which each element has at most two children. Why not? Figure 15.7 has two distinct binary trees that are equivalent as trees.

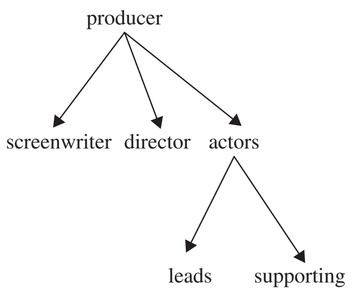


FIGURE 15.6 A (directed) tree

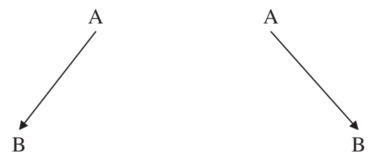


FIGURE 15.7 Two distinct binary trees, one with an empty right subtree and one with an empty left subtree

We can define a binary tree to be a (directed) tree in which each vertex has at most two children, labeled the "left" child and the "right" child of that vertex.

Trees allow us to study hierarchical relationships such as parent-child and supervisor-supervisee. With arbitrary trees, we are not subject to the at-most-two-children restriction of binary trees.

15.4 Networks

Sometimes we associate a non-negative number with each edge in a graph (which can be directed or undirected). The non-negative numbers are called ***weights***, and the resulting structure is called a ***weighted graph*** or ***network***. For example, Figure 15.8 has an undirected network in which each weight represents the distance between cities for the graph of Figure 15.2b.

Of what value is a weighted digraph, that is, why might the direction of a weighted edge be significant? Even if one can travel in either direction on an edge, the weight for going in one direction may be different from the weight going in the other direction. For example, suppose the weights represent the

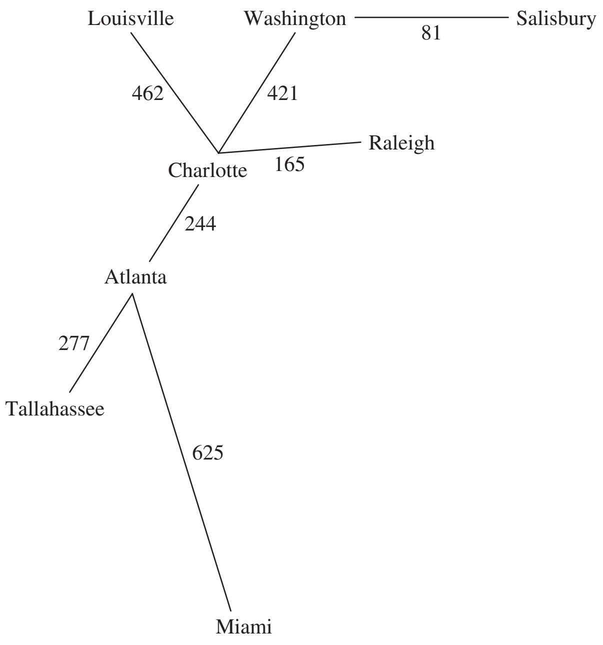


FIGURE 15.8 An undirected network in which vertices represent cities, and each edge's weight represents the distance between the two cities in the edge

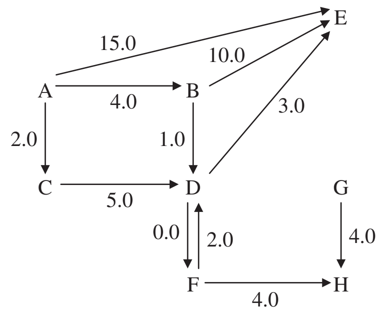


FIGURE 15.9 A weighted digraph with 8 vertices and 11 edges

time for a plane flight between two cities. Due to the prevailing westerly winds, the time to fly from New York to Los Angeles is usually longer than the time to fly from Los Angeles to New York.

Figure 15.9 shows a weighted digraph in which the weight of the edge from vertex D to vertex F is different from the weight of the edge going in the other direction.

With each path between two vertices in a network, we can calculate the total weight of the path. For example, in Figure 15.9, the path A, C, D, E has a total weight of 10.0. Can you find a shorter path from A to E, that is, a path with smaller total weight[1]? The shortest path from A to E is A, B, D, E with total weight 8.0. Later in this chapter we will develop an algorithm to find a shortest path (there may be several of them) between two vertices in a network.

The weighted digraph in Figure 15.9 is not connected because, for example, there is no path from B to C. Recall that a path in a digraph must follow the direction of the arrows.

Now that we have seen how to define a graph or tree (directed or undirected, weighted or unweighted), we can outline some well-known algorithms. The implementation of those algorithms, and their analysis, will be handled in Section 15.6.3.

15.5 Graph Algorithms

A prerequisite to other graph algorithms is being able to iterate through a graph, so we start by looking at iterators in general. We focus on two kinds of iterators: breadth-first and depth-first. These terms may ring a bell; in Chapter 9, we studied breadth-first and depth-first (also known as pre-order) traversals of a binary tree.

15.5.1 Iterators

There are several kinds of iterators associated with directed or undirected graphs, and these iterators can also be applied to trees and networks (directed or undirected). First, we can simply iterate over all of the vertices in the graph. The iteration need not be in any particular order. For example, here is an iteration over the vertices in the weighted digraph of Figure 15.9:

A, B, D, F, G, C, E, H

[1]This is different from the meaning of "shorter" in the graph sense, namely, having fewer edges in the path.

In addition to iterating over all of the vertices in a graph, we are sometimes interested in iterating over all vertices reachable from a given vertex. For example, in the weighted digraph of Figure 15.9, we might want to iterate over all vertices ***reachable*** from A, that is, over all vertices that are in some path that starts at A. Here is one such iteration:

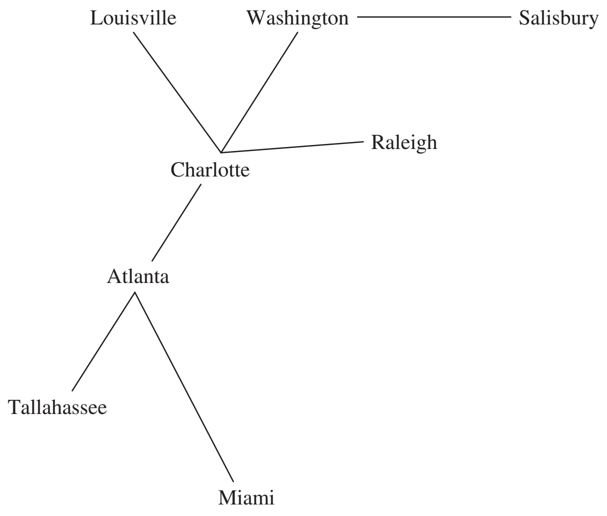
A, B, C, D, E, F, H

The vertex G is not reachable from A, so G will not be in any iteration from A.

15.5.1.1 Breadth-First Iterators

A ***breadth-first iterator***, similar to a breadth-first traversal in Chapter 9, visits vertices in order, beginning with some vertex identified as the "start vertex." What is the order? Roughly, it can be described as "neighbors first." As soon as a vertex has been visited, each neighbor of that vertex is marked as "reachable," and vertices are visited in the order in which they are marked as reachable. The first vertex marked as reachable and visited is the start vertex, then the neighbors of the start vertex are marked as reachable, so they are visited, then the neighbors of those neighbors, and so on.

No vertex is visited more than once, and any vertex that is not on some path from the start vertex will not be visited at all. For example, assume that the vertices in the following graph—from Figure 15.2b—were entered in alphabetical order:



We will perform a breadth-first iteration starting at Atlanta and visit neighbors in alphabetical order. So we start with

$\overset{r,\ v}{\text{Atlanta}}$

The *r* and *v* above Atlanta indicate that Atlanta has been marked as reachable and visited. The neighbors of Atlanta are now marked as reachable:

$\overset{r,\ v}{\text{Atlanta}}$, $\overset{r}{\text{Charlotte}}$, $\overset{r}{\text{Miami}}$, $\overset{r}{\text{Tallahassee}}$

Each of those neighbors of Atlanta is then visited in turn, and when a city is visited, each of its neighbors is marked as reachable—unless that neighbor was marked as reachable previously. When we visit Charlotte, we have

$\overset{r,\,v}{\text{Atlanta}}$, $\overset{r,\,v}{\text{Charlotte}}$, $\overset{r}{\text{Miami}}$, $\overset{r}{\text{Tallahassee}}$, $\overset{r}{\text{Louisville}}$, $\overset{r}{\text{Raleigh}}$, $\overset{r}{\text{Washington}}$

Notice that the first neighbor of Charlotte, Atlanta, is ignored because we have already marked as reachable (and visited) Atlanta. Then Miami is visited (its only neighbor has already been marked as reachable), Tallahassee (its only neighbor has already been marked as reachable), Louisville (its only neighbor has already been marked as reachable), Raleigh (its only neighbor has already been marked as reachable), and Washington, whose neighbor Salisbury is now marked as reachable. We now have

$\overset{r,\,v}{\text{Atlanta}}$, $\overset{r,\,v}{\text{Charlotte}}$, $\overset{r,\,v}{\text{Miami}}$, $\overset{r,\,v}{\text{Tallahassee}}$, $\overset{r,\,v}{\text{Louisville}}$, $\overset{r,\,v}{\text{Raleigh}}$, $\overset{r,\,v}{\text{Washington}}$ $\overset{r}{\text{Salisbury}}$

When we visit Salisbury we are done because Salisbury has no not-yet-reached neighbors marked as not reachable. In other words, we have iterated through all cities reachable from Atlanta, starting at Atlanta:

Atlanta, Charlotte, Miami, Tallahassee, Louisville, Raleigh, Washington, Salisbury

The order of visiting cities would be different if we started a breadth-first iteration at Louisville:

Louisville, Charlotte, Atlanta, Raleigh, Washington, Miami, Tallahassee, Salisbury

Let's do some preliminary work on the design of the `BreadthFirstIterator` class. When a vertex has been marked as reachable, that vertex will, eventually, be visited. To make sure that vertex is not re-visited, we will keep track of which vertices have already been marked as reachable. To do this, we will store the marked-as-reachable vertices in some kind of collection. Specifically, we want to visit the neighbors of the current vertex in the order in which those neighbors were initially stored in the collection. Because we want the vertices removed from the collection in the order in which they were added to the collection, a *queue* is the appropriate collection. And we will also want to keep track of the current vertex.

We can now develop high-level algorithms for the `BreadthFirstIterator` methods—the details will have to be postponed until we create a class—such as `Network`, in which `BreadthFirstIterator` will be embedded. The constructor enqueues the start vertex and marks all other vertices as not reachable:

```
public BreadthFirstIterator (Vertex start)
{
     for every vertex in the graph:
            mark that vertex as not reachable.
     mark start as reachable.
     queue.enqueue (start);
} // algorithm for constructor
```

The `hasNext()` method returns `!queue.isEmpty()`. The `next()` method removes the front vertex from the queue, makes that vertex the current vertex, and enqueues each neighbor of the current vertex that has not yet been marked as reachable:

```
public Vertex next()
{
       current = queue.dequeue();
```

```
        for each vertex that is a neighbor of current:
              if that vertex has not yet been marked as reachable
              {
                     mark that vertex as reachable;
                     enqueue that vertex;
              } // if
        return current;
 } // algorithm for method next
```

The analysis of this algorithm is postponed until Section 15.6.3, when we will have all of the details associated with the `BreadthFirstIterator` class.

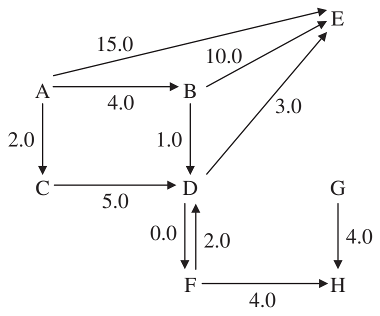
The `remove()` method deletes from the graph the current vertex, that is, the vertex most recently returned by a call to the `next()` method. All edges going into or out of that current vertex are also deleted from the graph.

For an example of how the queue and the `next()` method work together in a breadth-first iteration, suppose we create a weighted digraph by entering the sequence of edges and weights in Figure 15.10.

A	B	4.0
A	C	2.0
A	E	15.0
B	D	1.0
B	E	10.0
C	D	5.0
D	E	3.0
D	F	0.0
F	D	0.0
F	H	4.0
G	H	4.0

FIGURE 15.10 A sequence of edges and weights to generate the weighted digraph in Figure 15.9

The weighted digraph created is the same one shown in Figure 15.9:



To conduct a breadth-first iteration starting at A, for example, we first enqueue A in the constructor. The first call to `next()` dequeues A, enqueues B, C and E, and returns A. The second call to `next()` dequeues B, enqueues D, and returns B. Figure 15.11 shows the entire queue generated by a breadth-first iteration starting at A.

queue	vertex returned by next()
A	A
B, C, E	B
C, E, D	C
E, D	E
D	D
F	F
H	H

FIGURE 15.11 A breadth-first iteration of the vertices starting at A. The vertices are enqueued—and therefore dequeued—in the same order in which they were entered in Figure 15.10

Notice that vertex G is missing from Figure 15.11. The reason is that G is not reachable from A, that is, there is no path from A to G. If we performed a breadth-first iteration from any other vertex, there would be even fewer vertices visited than in Figure 15.11. For example, a breadth-first iteration starting at vertex B would visit

B, D, E, F, H

in that order.

Breadth-first iterators are especially useful in iterating over a (directed) tree. The start vertex is the root and, as we saw in Chapter 9, the vertices are visited level-by-level: the root, the root's children, the root's grandchildren, and so on.

15.5.1.2 Depth-First Iterators

The other specialized iterator is a ***depth-first iterator***. A depth-first iterator is a generalization of the pre-order traversal of Chapter 9. To refresh your memory, here is the algorithm for a pre-order traversal of a binary tree *t*:

```
preOrder (t)
{
     if (t is not empty)
     {
             access the root element of t;
             preOrder (leftTree (t));
             preOrder (rightTree (t));
     } // if
} // preOrder traversal
```

To further help you recall how a pre-order traversal works, Figure 15.12 shows a binary tree and a pre-order traversal of its elements.

We can describe a pre-order traversal of a binary tree as follows: start with a leftward path from the root. Once the end of a path is reached, the algorithm *backtracks* to an element that has an as-yet-unreachable right child. Another leftward path is begun, starting with that right child.

A depth-first iteration of a graph proceeds similarly to a breadth-first iteration. We first mark as not reachable each vertex, then mark as reachable, and visit, the start vertex. Then we mark as reachable each neighbor of the start vertex. Next, we visit the most recently marked as reachable vertex of those

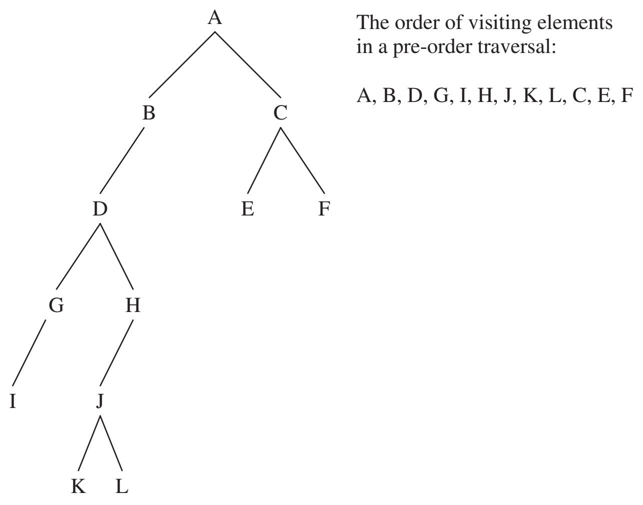
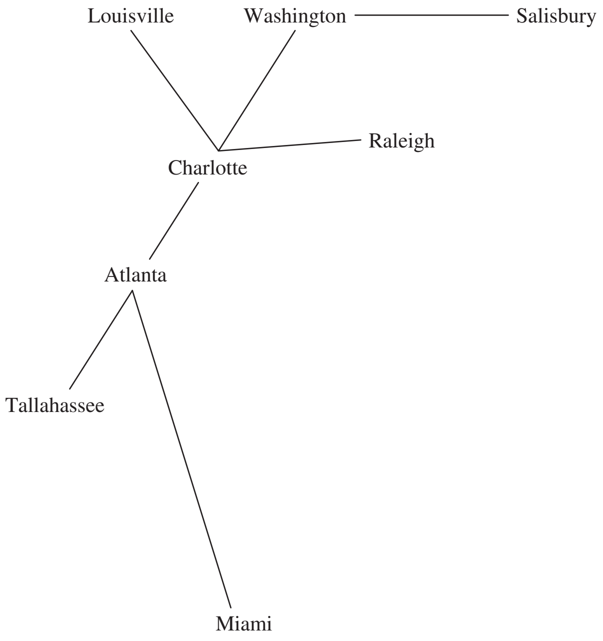

A
B
C
D
E
F
G
H
I
J
K
L
The order of visiting elements
in a pre-order traversal:
A, B, D, G, I, H, J, K, L, C, E, F

FIGURE 15.12 A binary tree and the order in which its elements would be visited during a pre-order traversal

neighbors, and mark as reachable each not-yet-marked-as-reachable neighbor of that vertex. Another path is begun starting with that unvisited vertex. This continues as long as there are vertices reachable from the start vertex that have not yet been found to be reachable.

For example, let's perform a depth-first iteration of the following graph, starting at Atlanta—we assume that the vertices were initially entered in alphabetical order:



We first visit the start vertex:

$\overset{r,\,v}{\text{Atlanta}}$

Then we mark as reachable the neighbors of Atlanta:

$\overset{r,\,v}{\text{Atlanta}}$, $\overset{r}{\text{Charlotte}}$, $\overset{r}{\text{Miami}}$, $\overset{r}{\text{Tallahassee}}$

We then visit the *most recently marked as reachable* vertex, namely Tennessee, *not* Charlotte. Tallahassee's only neighbor has already been marked as reachable, so we visit the next-most-recently marked-as-reachable vertex: Miami. Miami's only neighbor has already been marked as reachable, so we visit Charlotte, and mark as reachable Louisville, Raleigh and Washington, in that order. We now have

$\overset{r,\,v}{\text{Atlanta}}$, $\overset{r,\,v}{\text{Charlotte}}$, $\overset{r,\,v}{\text{Miami}}$, $\overset{r,\,v}{\text{Tallahassee}}$, $\overset{r}{\text{Louisville}}$, $\overset{r}{\text{Raleigh}}$, $\overset{r}{\text{Washington}}$

Washington, the most recently marked-as-reachable vertex, is visited, and its only not yet marked-as-reachable neighbor, Salisbury, is marked as reachable. We now have

$\overset{r,\,v}{\text{Atlanta}}$, $\overset{r,\,v}{\text{Charlotte}}$, $\overset{r,\,v}{\text{Miami}}$, $\overset{r,\,v}{\text{Tallahassee}}$, $\overset{r}{\text{Louisville}}$, $\overset{r}{\text{Raleigh}}$, $\overset{r,\,v}{\text{Washington}}$ $\overset{r}{\text{Salisbury}}$

Now Salisbury is the most recently marked-as-reachable vertex, so Salisbury is visited. Finally, Raleigh and then Louisville are visited. The order in which vertices are visited is as follows:

Atlanta, Tallahassee, Miami, Charlotte, Washington, Salisbury, Raleigh, Louisville

For a depth-first iteration starting at Charlotte, the order in which vertices are visited is:

Charlotte, Washington, Salisbury, Raleigh, Louisville, Atlanta, Tallahassee, Miami

With a breadth-first iteration, we saved vertices in a queue so that the vertices were visited in the order in which they were saved. With a depth-first iteration, the next vertex to be visited is the ***most recently reached*** vertex. So the vertices will be *stacked* instead of queued. Other than that, the basic strategy of a depth-first iterator is exactly the same as the basic strategy of a breadth-first iterator. Here is the high-level algorithm for `next()`:

```
public Vertex next()
{
       current = stack.pop();

       for each vertex that is a neighbor of current:
             if that vertex has not yet been marked as reachable
             {
                    mark that vertex as reachable;
                    push that vertex onto stack;
             } // if
       return current;
} // algorithm for method next
```

The analysis of this algorithm is the same as that of the `next()` method in the `BreadthFirstIterator` class: see Section 15.6.3.

Suppose, as we did above, we create a weighted digraph from the following input, in the order given:

A	B	4.0
A	C	2.0
A	E	15.0
B	D	1.0
B	E	10.0
C	D	5.0
D	E	3.0
D	F	0.0
F	D	0.0
F	H	4.0
G	H	4.0

The weighted digraph created is the same one shown in Figure 15.9:

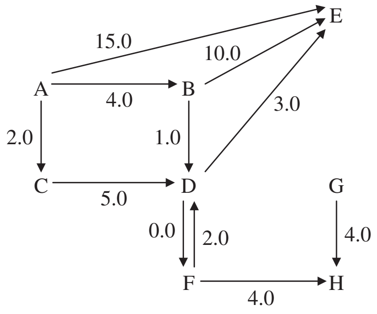


Figure 15.13 shows the sequence of stack states and the vertices returned by `next()` for a depth-first iteration of the weighted digraph in Figure 15.9, as generated from the input in Figure 15.10.

We could have developed a backtrack version of the `next()` method. The vertices would be visited in the same order, but recursion would be utilized instead of an explicit stack.

When should you use a breadth-first iterator and when should you use a depth-first iterator? If you are looping through all vertices reachable from the start vertex, there's not much reason to pick. The only

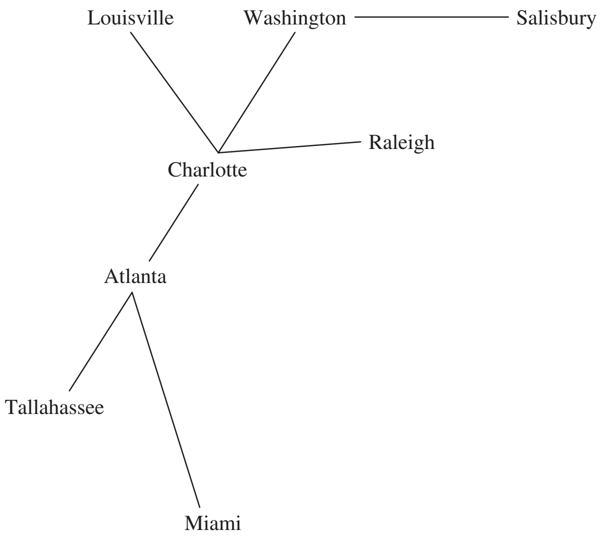
stack (top vertex is shown it *leftmost*)	**vertex returned by next()**
A	A
E, C, B	E
C, B	C
D, B	D
F, B	F
H, B	H
B	B

FIGURE 15.13 A depth-first iteration of the vertices reachable from A. We assume the vertices were entered as in Figure 15.10

difference is the order in which the vertices will be visited. But if you are at some start vertex and you are searching for a specific vertex reachable from that start vertex, there can be a difference. If, somehow, you know that there is a short path from the start vertex to the vertex sought, a breadth-first iterator is preferable: the vertices on a path of length 1 from the start vertex are visited, then the vertices on a path of length 2, and so on. On the other hand, if you know that the vertex sought may be very far from the start vertex, a depth-first search will probably be quicker (see Figure 15.12).

15.5.2 Connectedness

In Section 15.1, we defined an undirected graph to be ***connected*** if, for any given vertex, there is a path from that vertex to any other vertex in the graph. For example, the following is a connected, undirected graph:



For a digraph, that is, a directed graph, connectedness means that for any two distinct vertices, there is a path—that follows the directions of the arrows—between them. A breadth-first or depth-first iteration over all vertices in a graph can be performed only if the graph is connected. In fact, we can use the ability to iterate between any two vertices as a test for connectedness of a digraph.

Given a digraph, let `itr` be an iterator over the digraph. For each vertex `v` returned by `itr.next()`, let `bfitr` be a breadth-first iterator starting at `v`. We check to make sure that the number of vertices reachable from `v` (including `v` itself) is equal to the number of vertices in the digraph.

Here is a high-level algorithm to determine connectedness in a digraph:

```
public boolean isConnected()
{
    for each Vertex v in this digraph
    {
      // Count the number of vertices reachable from v.
      Construct a BreadthFirstIterator, bfItr, starting at v.
      int count = 0;
      while (bfItr.hasNext())
```

```
          {
                 bfItr.next();
                 count++;
          } // while
          if (count < number of vertices in this digraph)
            return false;
        } // for
        return true;
    } // algorithm for isConnected
```

For an undirected graph, the `isConnected()` algorithm is somewhat simpler: there is no need for an outer loop to iterate through the entire graph. See Concept Exercise 15.4.

In the next two sections, we outline the development of two important network algorithms. Each algorithm is sufficiently complex that it is named after the person (Prim, Dijkstra) who invented the algorithm.

15.5.3 Generating a Minimum Spanning Tree

Suppose a cable company has to connect a number of houses in a community. Given the costs, in hundreds of dollars, to lay cable between the houses, determine the minimum cost of connecting the houses to the cable system. This can be cast as a connected network problem; each weight represents the cost to lay cable between two neighbors. Figure 15.14 gives a sample layout. Some house-to-house distances are not given because they represent infeasible connections (over a mountain, for example).

In a connected, undirected network, a ***spanning tree*** is a weighted tree that consists of all of the vertices and some of the edges (and their weights) from the network. Because a tree must be connected, a spanning tree connects all of the vertices of the original graph. For example, Figures 15.15 and 15.16 show two spanning trees for the network in Figure 15.14. For the sake of specificity, we designate vertex A as the root of each tree, but any other vertex would serve equally well.

A ***minimum spanning tree*** is a spanning tree in which the sum of all the weights is no greater than the sum of all the weights in any other spanning tree. The original problem about laying cable can be re-stated in the form of constructing a minimum spanning tree for a connected network. To give you an idea of how difficult it is to solve this problem, try to construct a minimum spanning tree for the network in Figure 15.14. How difficult would it be to "scale up" your solution to a community with 1,000 houses?

An algorithm to construct a minimum spanning tree is due to R. C. Prim [1957]. Here is the basic strategy. Start with an empty tree T and pick any vertex v in the network. Add v to T. For each vertex w such

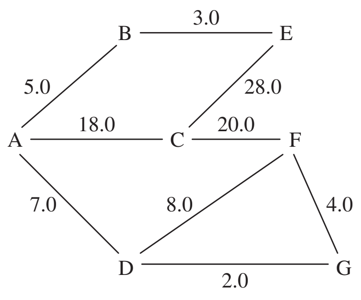


FIGURE 15.14 A connected network in which the vertices represent houses and the weights represent the cost, in hundreds of dollars, to connect the two houses

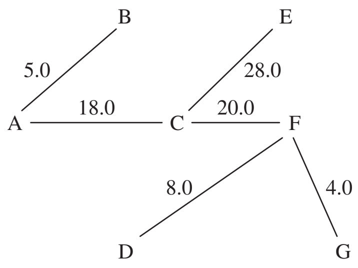


FIGURE 15.15 A spanning tree for the network in Figure 15.14

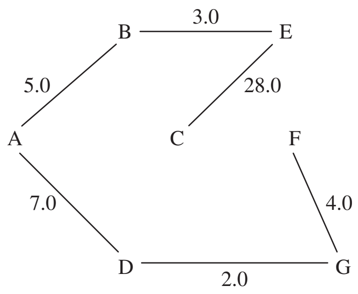
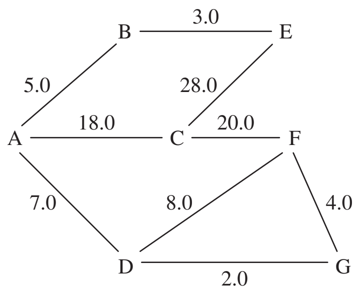


FIGURE 15.16 Another spanning tree for the network in Figure 15.14

that (v, w) is an edge with weight wweight, save the ordered triple ⟨v, w, wweight⟩ in a collection—we'll see what kind of collection shortly. Then loop until T has as many vertices as the original network. During each loop iteration, remove from the collection the triple ⟨x, y, yweight⟩ for which yweight is the smallest weight of all triples in the collection; if y is not already in T, add y and the edge (x, y) to T and save in the collection every triple ⟨y, z, zweight⟩ such that z is not already in T and (y, z) is an edge with weight zweight.

What kind of collection should we have? The collection should be ordered by weights; we need to be able to add an element, that is, a triple, to the collection and to remove the triple with lowest weight. A *priority queue* will perform these tasks quickly. Recall from Chapter 13 that for the `PriorityQueue` class, averageTime(n) for the `add (E element)` method is constant, and averageTime(n) for the `remove()` method is logarithmic in n.

To see how Prim's algorithm works, let's start with the network in Figure 15.14, repeated here:

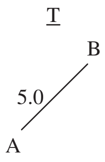


Initially, the tree T and the priority queue `pq` are both empty. Add A to T, and add to `pq` each triple of the form ⟨A, w, wweight⟩ where (A, w) is an edge with weight wweight. Figure 15.17 shows the contents

T	pq
A	<A, B, 5.0>
	<A, D, 7.0>
	<A, C, 18.0>

FIGURE 15.17 The contents of T and `pq` at the start of Prim's algorithm as applied to the network in Figure 15.14

of T and `pq` at this point. For the sake of readability, the triples in `pq` are shown in increasing order of weights; strictly speaking, all we know for sure is that the `element()` and `remove()` methods return the triple with smallest weight.
When the lowest-weighted triple, ⟨A, B, 5.0⟩ is removed from `pq`, the vertex B and the edge (A, B) are added to T, and the triple ⟨B, E, 3.0⟩ is added to `pq`. See Figure 15.18.

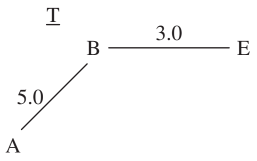


`pq`

<B, E, 3.0>
<A, D, 7.0>
<A, C, 18.0>

FIGURE 15.18 The contents of T and `pq` during the application of Prim's algorithm to the network in Figure 15.14

During the next iteration, the triple ⟨B, E, 3.0⟩ is removed from `pq`, the vertex E and edge (B, E) are added to T, and the triple ⟨E, C, 28.0⟩ is added to `pq`. See Figure 15.19.

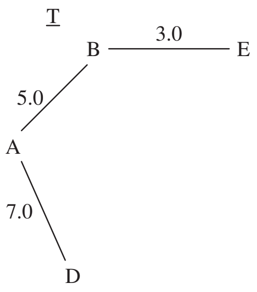


`pq`

<A, D, 7.0>
<A, C, 18.0>
<E, C, 28.0>

FIGURE 15.19 The contents of T and `pq` during the application of Prim's algorithm to the network in Figure 15.14

During the next iteration, the triple ⟨A, D, 7.0⟩ is removed from `pq`, the vertex D and the edge (A, D) are added to T, and the triples ⟨D, F, 8.0⟩ and ⟨D, G, 2.0⟩ are added to pq. See Figure 15.20.

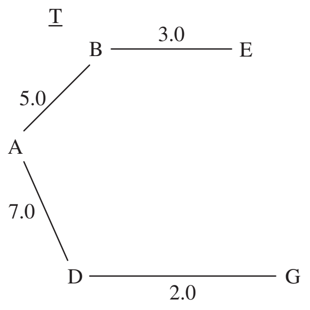


`pq`

<D, G, 2.0>
<D, F, 8.0>
<A, C, 18.0>
<E, C, 28.0>

FIGURE 15.20 The contents of T and `pq` during the application of Prim's algorithm to the network in Figure 15.14

During the next iteration, the triple ⟨D, G, 2.0⟩ is removed from pq, the vertex G and the edge (D, G) are added to T, and the triple ⟨G, F, 4.0⟩ is added to pq. See Figure 15.21.

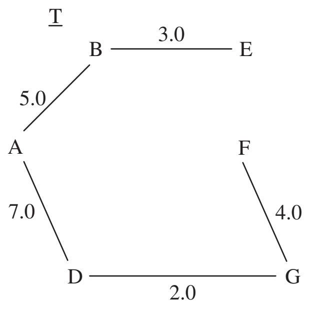


pq

<G, F, 4.0>
<D, F, 8.0>
<A, C, 18.0>
<E, C, 28.0>

FIGURE 15.21 The contents of T and pq during the application of Prim's algorithm to the network in Figure 15.14

During the next iteration, the triple ⟨G, F, 4.0⟩ is removed from pq, the vertex F and the edge (G, F) are added to T, and the triple ⟨F, C, 20.0⟩ is added to pq. See Figure 15.22.



pq

<D, F, 8.0>
<A, C, 18.0>
<F, C, 20.0>
<E, C, 28.0>

FIGURE 15.22 The contents of T and pq during the application of Prim's algorithm to the network in Figure 15.14

During the next iteration, the triple ⟨D, F, 8.0⟩ is removed from pq. But *nothing* is added to T or pq because F is already in T!

During the next iteration, the triple ⟨A, C, 18.0⟩ is removed from pq, the vertex C and the edge (A, C) are added to T, and nothing is added to pq. The reason nothing is added to pq is that, for all of C's edges, (C, A), (C, E) and (C, F), the second element in the pair is already in T. See Figure 15.23.

Even though pq is not empty, we are finished because every vertex in the original network is also in T.

From the way T is constructed, we know that T is a spanning tree. We can show, by contradiction, that T is a minimum spanning tree. In general, a proof by contradiction assumes some statement to be false, and shows that some other statement—known to be true—must also be false. We then conclude that the original statement must have been true.

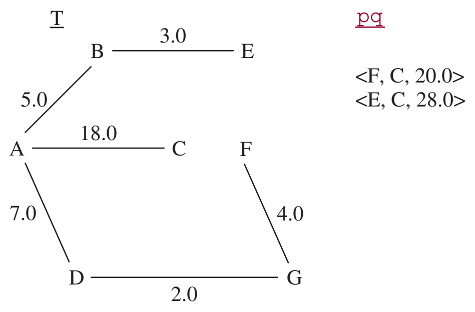
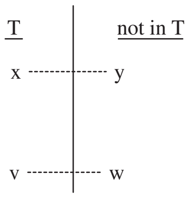


FIGURE 15.23 The contents of T and `pq` after the last iteration in the application of Prim's algorithm to the network in Figure 15.14

Assume that T is not a minimum spanning tree. Then during some iteration, a triple ⟨x, y, yweight⟩ is removed from `pq` and the edge (x, y) is added to T, but there is some vertex v, already in T, and w, not in T, such that edge (v, w) has lower weight than edge (x, y). Pictorially:



Note that since v is already in T, the triple starting with ⟨v, w, ?⟩ must have been added to `pq` earlier. But the edge (v, w) could not have a lower weight than the edge (x, y) because the triple ⟨x, y, y weight⟩ was removed from `pq`, not the triple starting with ⟨v, w, ?⟩. This contradicts the claim that (v, w) had lower edge weight than (x, y). So T, with edge (x, y) added, must still be minimum.

Can Prim's algorithm be applied to a connected, directed network? Consider the following network: {<a, b, 8.0>, <b, c, 5.0>, <c, a, 10.0>}. Prim's algorithm would give a different result depending on which vertex was chosen as the root. We can—and will—apply Prim's algorithm to a connected directed network provided it is equivalent to its undirected counterpart. That is, for any pair of vertices u and v, if v is a neighbor of u then u is a neighbor of v, and the weights of the two edges are the same.

Prim's algorithm is another example of the greedy-algorithm design pattern (the Huffman encoding algorithm in Chapter 13 is also a greedy algorithm). During each loop iteration, the locally optimal choice is made: the edge with lowest weight is added to T. This sequence of locally optimal—that is, greedy—choices leads to a globally optimal solution: T is a minimum spanning tree.

Another greedy algorithm appears in Section 15.5.4. Concept Exercise 15.7 and Lab 23 show that greed does not always succeed.

15.5.4 Finding the Shortest Path through a Network

In Section 15.4.3, we developed a strategy for constructing a minimum spanning tree in a network. A similar strategy applies to finding a shortest path in a network (directed or undirected) from some vertex v1 to some other vertex v2. In this context, "shortest" means having the lowest total weight. Both algorithms are greedy, and both use a priority queue. The shortest-path algorithm, due to Edsgar Dijkstra [1959], is essentially a breadth-first iteration that starts at v1 and stops as soon as v2's pair is removed from the priority queue `pq`. (Dijkstra's algorithm actually finds a shortest path from v1 to every other vertex in the network.) Each pair consists of a vertex w and the sum of the weights of all edges on the shortest path so far from v1 to w.

The priority queue is ordered by lowest *total* weights. To keep track of total weights, we have a map, `weightSum`, in which each key is a vertex w and each value is the sum of the weights of all the edges on the shortest path so far from v1 to w. To enable us to re-construct the shortest path when we are through, there is another map, `predecessor`, in which each key is a vertex w, and each value is the vertex that is the immediate predecessor of w on the shortest path so far from v1 to w.

Basically, `weightSum` maps each given vertex to the minimum total weight, so far, of the path from v1 to the given vertex. Initially, `pq` consists of vertex v1 and its weight, 0.0. On each iteration we greedily choose the vertex-weight pair ⟨x, total weight⟩ in `pq` that has the minimum total weight among all vertex-weight pairs in `pq`. If there is a neighbor y of x whose total weight can be reduced by the path ⟨v1,. . ., x, y⟩ , then y's path and minimum weight are altered, and y (and its new total weight) is added to `pq`. For example, we might have the partial network shown in Figure 15.24.

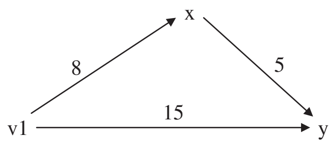
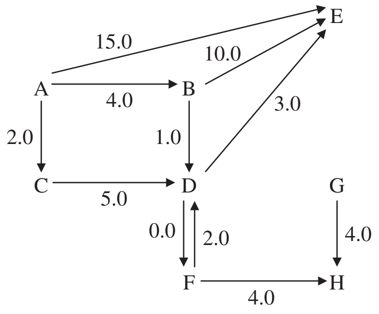


FIGURE 15.24 The minimum-weight path from vertex v1 to vertex y had been 15, but the path from vertex v1 through vertex x to vertex y has a lower total weight

Then the total weight between v1 and y is reduced to 13, the pair ⟨y, 13⟩ is added to `pq` and y's predecessor becomes x. Eventually, this yields the shortest path from v1 to v2, if there is a path between those vertices.

To start, `weightSum` associates with each vertex a very large total weight, and `predecessor` associates with each vertex the value `null`. We then refine those initializations by mapping v1's `weightSum` to 0, mapping v1's `predecessor` to v1 itself, and adding (v1, 0.0) to `pq`. This completes the initialization phase.

Suppose we want to find the shortest path from A to E in the network from Figure 15.9, repeated here:



For simplicity, we ignore G because, in fact, G is not reachable from A. Initially, we have

`weightSum`	`predecessor`	`pq`
A, 0.0	A	⟨A, 0.0⟩
B, 10000.0	`null`	
C, 10000.0	`null`	
D, 10000.0	`null`	
E, 10000.0	`null`	
F, 10000.0	`null`	
H, 10000.0	`null`	

After initializing, we keep looping until E is removed from `pq` (that is, until the shortest path is found) or `pq` is empty (that is, there is no path from A to E). During the first iteration of this loop, the minimum (and only) pair, ⟨A, 0.0⟩, is removed from `pq`. Since that pair's weight, 0.0, is less than or equal to A's `weightSum` value, we iterate, in an inner loop, over the neighbors of A. For each neighbor of A, if A's `weightSum` value plus that neighbor's weight is less than the neighbor's `weightSum` value, `weightSum,` and `predecessor` are updated, and the neighbor and its total weight (so far) are added to `pq`. The effects are shown in Figure 15.25.

`weightSum`	`predecessor`	`pq`
A, 0.0	A	⟨C, 2.0⟩
B, 4.0	A	⟨B, 4.0⟩
C, 2.0	A	⟨E, 15.0⟩
D, 10000.0	`null`	
E, 15.0	A	
F, 10000.0	`null`	
H, 10000.0	`null`	

FIGURE 15.25 Dijkstra's shortest-path algorithm, after the first iteration of the outer loop

After the processing shown in Figure 15.25, the outer loop is executed for a second time: the pair ⟨C, 2.0⟩ is removed from `pq` and we iterate (the inner loop) over the neighbors of C. The only vertex on an edge from C is D, and the weight of that edge is 5.0. This weight plus 2.0 (C's weight sum) is 7.0, which is less than D's weight sum, 10000.0). So in `weightSum`, D's weight sum is upgraded to 7.0. Figure 15.26 shows the effect on `weightSum`, `predecessor,` and `pq`.

Figure 15.26 indicates that at this point, the lowest-weight path from A to D has a total weight of 7.0. During the third iteration of the outer loop, ⟨B, 4.0⟩ is removed from `pq` and we iterate over the neighbors of B, namely, D and E. The effects are shown in Figure 15.27.

At this point, the lowest-weight path to D has a total weight of 5.0 and the lowest-weight path to E has a total weight of 14.0. During the fourth iteration of the outer loop, ⟨D, 5.0⟩ is removed from `pq`, and we iterate over the neighbors of D, namely, F and E. Figure 15.28 shows the effects of this iteration.

During the fifth outer-loop iteration, ⟨F, 5.0⟩ is removed from `pq`, the neighbors of F, namely D and H are examined, and the collections are updated. See Figure 15.29.

`weightSum`	`predecessor`	`pq`
A, 0.0	A	⟨B, 4.0⟩
B, 4.0	A	⟨D, 7.0⟩
C, 2.0	A	⟨E, 15.0⟩
D, 7.0	C	
E, 15.0	A	
F, 10000.0	`null`	
H, 10000.0	`null`	

FIGURE 15.26 The state of the application of Dijkstra's shortest-path algorithm after the second iteration of the outer loop

`weightSum`	`predecessor`	`pq`
A, 0.0	A	⟨D, 5.0⟩
B, 4.0	A	⟨D, 7.0⟩
C, 2.0	A	⟨E, 14.0⟩
D, 5.0	B	⟨E, 15.0⟩
E, 14.0	B	
F, 10000.0	`null`	
H, 10000.0	`null`	

FIGURE 15.27 The state of the application of Dijkstra's shortest-path algorithm after the third iteration of the outer loop

`weightSum`	`predecessor`	`pq`
A, 0.0	A	⟨F, 5.0⟩
B, 4.0	A	⟨D, 7.0⟩
C, 2.0	A	⟨E, 8.0⟩
D, 5.0	B	⟨E, 14.0⟩
E, 8.0	D	⟨E, 15.0⟩
F, 5.0	D	
H, 10000.0	`null`	

FIGURE 15.28 The state of the application of Dijkstra's shortest-path algorithm after the fourth iteration of the outer loop

During the sixth iteration of the outer loop, ⟨D, 7.0⟩ is removed from `pq`. The minimum total weight, so far, from A to D is recorded in `weightSum` as 5.0. So there is no inner-loop iteration.

During the seventh iteration of the outer loop, ⟨E, 8.0⟩ is removed from `pq`. Because E is the vertex we want to find the shortest path to, we are done. How can we be sure there are no shorter paths to E? If there were another path to E with total weight t less than 8.0, then the pair ⟨E, t⟩ would have been removed from `pq` before the pair ⟨E, 8.0⟩.

`weightSum`	`predecessor`	`pq`
A, 0.0	A	⟨D, 7.0⟩
B, 4.0	A	⟨E, 8.0⟩
C, 2.0	A	⟨H, 9.0⟩
D, 5.0	B	⟨E, 14.0⟩
E, 8.0	D	⟨E, 15.0⟩
F, 5.0	D	
H, 9.0	F	

FIGURE 15.29 The state of the application of Dijkstra's shortest-path algorithm after the fifth iteration of the outer loop

We construct the shortest path, as a `LinkedList` of vertices, from `predecessor`: starting with an empty `LinkedList` object, we prepend E; then prepend D, the predecessor of E; then prepend B, the predecessor of D; finally, prepend A, the predecessor of B. The final contents of the `LinkedList` object are, in order,

A, B, D, E

There are a few details that are missing in the above description of Dijkstra's algorithm. For example, how will the vertices, edges, and neighbors be stored? What are worstTime(n) and averageTime(n)? To answer these questions, we will develop a class in Section 15.6.3, and fill in the missing details, not only of Dijkstra's algorithm, but of all our graph-related work.

15.5.5 Finding the Longest Path through a Network?

Interestingly, it is often important to be able to find a longest path through a network. A ***project network*** is an acyclic, directed network with a single ***source*** (no arrows coming into it) and a single ***sink*** (no arrows going out from it). The source and sink are usually labeled S (for start) and T (for terminus), respectively. Figure 15.30 shows a project network.

We can interpret the edges in a project network as activities to be performed, the weights as the time in days (or cost in dollars) required for the activities, and the vertices as events that indicate the completion of all activities coming into the event. With that interpretation, the ***project length***, that is,

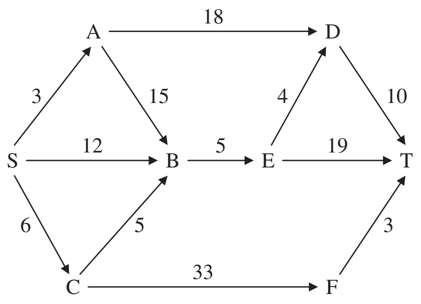


FIGURE 15.30 A project network

the length of a longest path, represents the number of days required to complete the project. For example, in Figure 15.30, there are two longest paths:

S, A, B, E, T

and

S, C, F, T

Each of those paths has a length of 42, so the project, as scheduled, will take 42 days. Dijkstra's algorithm can easily be modified to find a longest path in an acyclic network. The next section, on topological sorting, can be used to determine if a network is acyclic.

In a project network, some activities can occur concurrently, and some activities can be delayed without affecting the project length. A project manager may want to know *which* activities can be delayed without delaying the entire project. First, we need to make some calculations.

For each event w, we can calculate ET(w), the ***earliest time*** that event w can be completed. For event S, ET(S) is 0. For any other event w, ET(w) is the maximum of {ET(v) + weight (v, w)} for any event v such that ⟨v, w⟩ forms an edge. For example, in Figure 15.30, ET(C) = 6, ET (B) = max {12, 18, 11} = 18, ET(E) = 23, ET (D) = max {21, 27} = 27, and so on.

For each event w, we can also calculate LT(w), the ***latest time*** by which w must be completed to avoid delaying the entire project. For event T, LT(T) is the project length; for any other event w, LT(w) is the minimum of {LT(x) − weight(w,x)} for any event x such that ⟨w, x⟩ forms an edge. For example, in Figure 15.30, LT(T) = 42, LT(D) = 32, LT(E) = min {28, 23} = 23, LT(B) = 18, LT (C) = min {6, 13} = 6, and so on.

Finally, ST(y, z), the ***slack time*** of activity ⟨y, z⟩, is calculated as follows:

ST(y, z) = LT(z) – ET(y) – weight(y, z)

For example, in Figure 15.30,

ST(S, C) = LT(C) – ET(S) – weight(S, C) = 6 – 0 – 6 = 0

ST(A, B) = LT(B) – ET(A) – weight(A, B) = 18 – 3 – 15 = 0

ST(A, D) = LT(D) – ET(A) – weight(A, D) = 32 – 3 – 18 = 11

An activity with a slack time of zero is called a ***critical activity***, and any path that consists only of critical activities is called a ***critical path***. The idea is that special attention should be given to any critical activity because any delay in that activity will increase the project length.

Activities that are not on a critical path are less constrained: They may be delayed or take longer than scheduled without affecting the length of the entire project. For example, in Figure 15.30, the activity ⟨A, D⟩ has a slack time of 11 days: That activity can be delayed or take longer than scheduled, as long as the activity is completed by day 32. The significance of this is that resources allocated to activity ⟨A, D⟩ can be diverted to a critical task, and thereby speed up the completion of the entire project.

Programming Project 15.3 entails finding the slack time for each project in a project network. In order to calculate the earliest times for each event, the events must be ordered so that for example, when ET(w) is to be calculated for some vertex w, the value of ET(v) must already be available for each vertex v such that ⟨v, w⟩ forms an edge. Section 15.5.5.1 describes that kind of ordering.

15.5.5.1 Topological Sorting

A network with a cycle cannot have a longest path because we could continuously loop through the cycle to create a path whose total weight is arbitrarily large. We will soon see how to determine if a network is acyclic.

The calculation of earliest times and latest times for the project network in Section 15.5.5 requires that the events be in order. Specifically, to calculate ET(w) for some vertex w, we need to know ET(v), for any vertex v such that ⟨v, w⟩ forms an edge. If the project network is small and the calculations are being done by hand, this ordering can be made implicitly. But for a large project, or for a program to perform the calculations, the ordering must be explicit. To calculate LT(w) for any vertex w, the reverse of this ordering is needed.

The ordering is not, necessarily, the one provided by a breadth-first iterator, a depth-first iterator, or an iterator over the entire network. The ordering is called a "topological ordering." The vertices $v_1, v_2, \ldots$ of a digraph are in ***topological order*** if v_i precedes v_j in the ordering whenever $\langle v_i, v_j \rangle$ forms an edge in the digraph. The arranging of vertices into a topological ordering is called ***topological sorting***. For example, for the digraph in Figure 15.30, here is a topological order:

S, A, C, B, E, F, D, T

Notice that D had to come after both A and E because ⟨A, D⟩ and ⟨E, D⟩ are edges in the digraph. Another topological order is

S, A, C, F, B, E, D, T

Any network that can be put into topological order must be acyclic. Concept Exercise 15.12 has more information about topological order, and Programming Exercise 15.5 suggests how to perform a topological sort.

15.6 A Network Class

In this chapter, we have introduced eight different data types: a graph and a tree, each of which can be directed or undirected, and weighted or unweighted. We would like to develop classes for these collections with a maximum amount of code sharing. The object-oriented solution is to utilize inheritance, but exactly how should this be done? If we make the `Digraph` class a subclass of `UndirectedGraph`, then virtually all of the code relating to edges will have to be overridden. That's because in an undirected graph, each edge A, B represents two links: from A to B and from B to A. Similarly, code written for graphs would have to be re-written for networks.

A better approach is to define the (directed) `Network` class, and make the other classes subclasses of that class. For example, we can view an undirected network as a directed network in which all edges are two-way. So to add an edge A, B in an undirected network, the method definition is:

```
/**
 *  Ensures that a given edge with a given weight is in this UndirectedNetwork
 *  object.
 *
 *  @param v1 - the first vertex of the edge.
 *  @param v2 -  the second vertex of the edge (the neighbor of v1).
 *  @param weight - the weight of the edge (v1, v2).
 *
 *  @return true - if this UndirectedNetwork object changed as a result of this call.
 *
 */
public boolean addEdge (Vertex v1, Vertex v2, double weight)
{
     return super.addEdge (v1, v2, weight)  && super.addEdge (v2, v1, weight);
} // method addEdge in UndirectedNetwork class, a subclass of Network
```

Furthermore, we can view a digraph as a network in which all weights have the value of 1.0. The `Digraph` class will have the following method definition:

```
/**
 *  Ensures that a given edge with a given weight is in this UndirectedNetwork
 *  object.
 *
 *  @param v1 - the first vertex of the edge.
 *  @param v2 - the second vertex of the edge (the neighbor of v1).
 *
 *  @return true - if this UndirectedNetwork object changed as a result of this call.
 *
 */
public boolean addEdge (Vertex v1, Vertex v2)
{
     return super.addEdge (v1, v2, 1.0);
} // method addEdge in DiGraph class, a subclass of Network
```

Figure 15.31 shows the inheritance hierarchy. Technically, the hierarchy is not a tree because in the Unified Modeling Language, the arrows go from the subclass to the superclass.

In the following section, we develop a (directed) `Network` class, that is, a weighted digraph class. The development of the subclasses—`DirectedWeightedTree`, `DirectedTree`, `Digraph`, `UndirectedNetwork`, `UndirectedWeightedTree`, `UndirectedGraph`, `Tree` and `UndirectedTree`—are provided on the book's website or are programming exercises.

The class heading, with `Vertex` as the type parameter, is

```
public class Network<Vertex> implements Iterable<Vertex>, java.io.Serializable
```

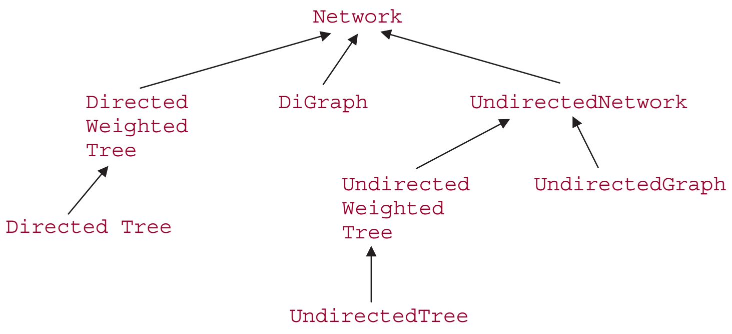


FIGURE 15.31 The inheritance hierarchy for the collections in this chapter

The Iterable interface has only one method, iterator(), which returns an iterator that implements hasNext(), next(), and remove(). By implementing the Iterable interface, we are able to utilize the enhanced **for** statement for Network objects. We have not mentioned the Iterable interface up to now because we have applied the enhanced **for** statement only to instances of classes that implement the Collection interface. The start of that interface is

```
public interface Collection<E> extends Iterable<E>
```

An important issue in developing the (directed) Network class is to decide what public methods the class should have: these constitute the abstract data-type network, that is, the user's view of the Network class. For the Network class, we have vertex-related methods, edge-related methods, and network-as-a-whole methods. In the method specifications, V represents the number of vertices and E represents the number of edges.

15.6.1 Method Specifications and Testing of the Network Class

Here are the vertex-related method specifications:

```
/**
 *  Determines if this Network object contains a specified Vertex object.
 *  The worstTime(V, E) is O(log V).
 *
 *  @param vertex – the Vertex object whose presence is sought.
 *
 *  @return true – if vertex is an element of this Network object.
 *
 *  @throws NullPointerException - if vertex is null.
 *
 */
public boolean containsVertex (Vertex vertex)


/**
 *  Ensures that a specified Vertex object is an element of this Network object.
 *  The worstTime(V, E) is O(log V).
 *
 *  @param vertex – the Vertex object whose presence is ensured.
 *
 *  @return true – if vertex was added to this Network object by this call; returns
 *          false if vertex was already an element of this Network object when
 *          this call was made.
 *
 *  @throws NullPointerException - if vertex is null.
 *
 */
public boolean addVertex (Vertex vertex)


/**
 *  Ensures that a specified Vertex object is not an element of this Network object.
```

```
     *  The worstTime(V, E) is O(V log V).
     *
     *  @param v – the Vertex object whose absence is ensured.
     *
     *  @return true – if v was removed from this Network object by this call;
     *          returns false if v is not an element of this Network object
     *          when this call is made.
     *
     *  @throws NullPointerException – if vertex is null.
     *
     */
    public boolean removeVertex (Vertex v)


     .
```

Here are the edge-related method specifications:

```
    /**
     *  Returns the number of edges in this Network object.
     *  The worstTime(V, E) is O(V).
     *
     *  @return the number of edges in this Network object.
     *
     */
    public int getEdgeCount()


    /**
     *  Determines the weight of an edge in this Network object.
     *  The worstTime(V, E) is O(log V).
     *
     *  @param v1 – the beginning Vertex object of the edge whose weight is sought.
     *  @param v2 – the ending Vertex object of the edge whose weight is sought.
     *
     *  @return the weight of edge <v1, v2>, if <v1, v2> forms an edge; return – 1.0 if
     *          <v1, v2> does not form an edge in this Network object.
     *
     *  @throws NullPointerException – if v1 is null and/or v2 is null.
     *
     */
    public double getEdgeWeight (Vertex v1, Vertex v2)


    /**
     *  Determines if this Network object contains an edge specified by two vertices.
     *  The worstTime(V, E) is O(log V).
     *
     *  @param v1 – the beginning Vertex object of the edge sought.
```

```
 *  @param v2 – the ending Vertex object of the edge sought.
 *
 *  @return true – if this Network object contains the edge <v1, v2>.
 *
 *  @throws NullPointerException – if v1 is null and/or v2 is null.
 *
 */
public boolean containsEdge (Vertex v1, Vertex v2)


/**
 *  Ensures that an edge is in this Network object.
 *  The worstTime(V, E) is O(log V).
 *
 *  @param v1 – the beginning Vertex object of the edge whose presence
 *              is ensured.
 *  @param v2 – the ending Vertex object of the edge whose presence is
 *              ensured.
 *  @param weight – the weight of the edge whose presence is ensured.
 *
 *  @return true – if the given edge (and weight) were added to this Network
 *                 object by this call; return false, if the given edge (and weight)
 *                 were already in this Network object when this call was made.
 *
 *  @throws NullPointerException – if v1 is null and/or v2 is null.
 *
 */
public boolean addEdge (Vertex v1, Vertex v2, double weight)


/**
 *  Ensures that an edge specified by two vertices is absent from this Network
 *  object.
 *  The worstTime (V, E) is O (V log V).
 *
 *  @param v1 – the beginning Vertex object of the edge whose absence is
 *              ensured.
 *  @param v2 – the ending Vertex object of the edge whose absence is
 *              ensured.
 *
 *  @return true – if the edge <v1, v2> was removed from this Network object
 *                 by this call; return false if the edge <v1, v2> was not in this
 *                 Network object when this call was made.
 *
 *  @throws NullPointerException – if v1 is null and/or v2 is null.
 *
 */
public boolean removeEdge (Vertex v1, Vertex v2)
```

Finally, we have the method specifications for those methods that apply to the network as a whole, including three flavors of iterators:

```
/**
 *  Initializes this Network object to be empty, with the ordering of
 *  vertices by an implementation of the Comparable interface.
 */
public Network()


/**
 *  Initializes this Network object to a shallow copy of a specified Network
 *  object.
 *
 *  @param network – the Network object that this Network object is
 *                   initialized to a shallow copy of.
 *
 *  @throws NullPointerException – if network is null.
 *
 */
public Network (Network<Vertex> network)


/**
 *  Determines if this Network object contains no vertices.
 *
 *  @return true – if this Network object contains no vertices.
 *
 */
public boolean isEmpty()


/**
 *  Determines the number of vertices in this Network object.
 *
 *  @return the number of vertices in this Network object.
 *
 */
public int size()


/**
 *  Determines if this Network object is equal to a given object.
 *
 *  @param obj – the object this Network object is compared to.
 *
 *  @return true – if this Network object is equal to obj.
 *
 */
public boolean equals (Object obj)


/**
 *  Returns a LinkedList<Vertex> object of the neighbors of a specified Vertex object.
 *  The worstTime(V, E) is O(V).
 *
 *  @param v – the Vertex object whose neighbors are returned.
 *
 *  @return a LinkedList<Vertex> object of the vertices that are neighbors of v.
 *
 *  @throws NullPointerException – if v is null.
 *
 */
public LinkedList<Vertex> neighbors (Vertex v)

/**
 * Returns an Iterator object over the vertices in this Network object.
 *
 *  @return an Iterator object over the vertices in this Network object.
 *
 */
public Iterator<Vertex> iterator()


/**
 *  Returns a breadth-first Iterator object over all vertices reachable from
 *  a specified Vertex object.
 *  The worstTime(V, E) is O(V log V).
 *
 *  @param v – the start Vertex object for the Iterator object returned.
 *
 *  @return a  breadth-first Iterator object over all vertices reachable from v.
 *
 *  @throws IllegalArgumentException – if v is not an element of this Network
 *          object.
 *
 *  @throws NullPointerException – if vertex is null.
 *
 */
public Iterator<Vertex> breadthFirstIterator (Vertex v)


/**
 *  Returns a depth-first Iterator object over all vertices reachable from
 *  a specified Vertex object.
 *  The worstTime(V, E) is O(V log V).
 *
 *  @param v - the start Vertex object for the Iterator object returned.
 *
 *  @return a  depth - first Iterator object over all vertices reachable from v.
 *
 *  @throws IllegalArgumentException - if v is not an element of this Network
 *          object.
 *
 *  @throws NullPointerException - if vertex is null.
 *
 */
public Iterator<Vertex> depthFirstIterator (Vertex v)


/**
 *  Determines if this (directed) Network object is connected.
 *  The worstTime(V, E) is O(V * V * log V).
 *
 *  @return true - if this (directed) Network object is connected.
 *
 */
public boolean isConnected()


/**
 *  Returns a minimum spanning tree for this connected Network object
 *  in which for any vertices u and v, if v is a neighbor of u then u is
 *  a neighbor of v, and the weights of those two edges are the same.
 *  The worstTime(V, E) is O(E log V).
 *
 *  @return a minimum spanning tree for this connected Network object.
 *
 */
public UndirectedWeightedTree<Vertex> getMinimumSpanningTree()


/**
 *  Finds a shortest path between two specified vertices in this Network
 *  object, and the total weight of that path.
 *  The worstTime(V, E) is O(E log V).
 *
 *  @param v1 - the beginning Vertex object.
 *  @param v2 - the ending Vertex object.
 *
 *  @return a LinkedList object containing the vertices in a shortest path
 *          from Vertex v1 to Vertex v2.  The last element in the Linked
 *          List object is the total weight of the path, or -1.0 if there is no path.
```

```
     *
     *  @throws NullPointerException – if v1 is null and/or v2 is null.
     *
     */
    public LinkedList<Object> getShortestPath (Vertex v1, Vertex v2)


    /**
     *  Returns a String representation of this Network object.
     *  The averageTime(V, E) is O(V * V).
     *
     *  @return a String representation of this Network object, with
     *          each vertex v, each neighbor w of that vertex, and the
     *          weight of the corresponding edge.  The format is
     *          {v1=[w11  weight11, w12  weight12,...],
     *          v2=[w21  weight21, w22  weight22,...],
     *          ...
     *          vn=[wn1  weightn1, wn2  weightn2,...]}
     *
     */
    public String toString()
```

Here is a test of the getShortestPath method (network is a field in the NetworkTest class):

```
@Test
public void testShortest3()
{
    network.addEdge ("S",  "A", 2);
    network.addEdge ("S",  "B", 6);
    network.addEdge ("S",  "C", 5);
    network.addEdge ("A",  "D", 8);
    network.addEdge ("B",  "C", 2);
    network.addEdge ("B",  "D", 3);
    network.addEdge ("B",  "E", 2);
    network.addEdge ("D",  "T", 5);
    network.addEdge ("D",  "E", 3);
    network.addEdge ("E",  "T", 1);
    network.addEdge ("C",  "F", 2);
    network.addEdge ("F",  "T", 10);

   assertEquals ("[S, B, E, T, 9.0]", network.getShortestPath ("S",  "T").toString());
} // method testShortest3
```

The book's website has the NetworkTest class, along with the UndirectedNetwork and UndirectedWeightedTree classes.

Before we turn to developer's issues in Section 15.6.2, we illustrate some of the Network class's methods in a class whose run method essentially consists of one method call after another. Each edge in the file network.in1 is two-way for the sake of the getMinimumSpanningTree method. (There is no attempt at modularization.)

```
import java.util.*;

import java.io.*;

public class NetworkExample
{
    public static void main (String[ ] args)
    {
        new NetworkExample().run();
    } // main

    public void run()
    {
        final String SHORTEST_PATH_MESSAGE1 = "\n\nThe shortest path from ";
        final String SHORTEST_PATH_MESSAGE2 = " and its total weight are ";
        final String REMOVAL_MESSAGE =
                     "\niterating over network after removing B-E and D:";

        Network<String> network = new Network<String>();

        try
        {
            Scanner sc = new Scanner (new File ("network.in1"));

            String start,
                   finish,
                   vertex1,
                   vertex2;

            double weight;

            // Get start and finish vertices.
            start = sc.next();
            finish = sc.next();

            // Get edges and weights.
            while (sc.hasNext())
            {
                vertex1 = sc.next();
                vertex2 = sc.next();
                weight = sc.nextDouble();
                network.addEdge (vertex1, vertex2, weight);
            } // while

            LinkedList<Object> pathList = network.getShortestPath (start, finish);
            System.out.println (SHORTEST_PATH_MESSAGE1 + start + " to " +
                                finish + SHORTEST_PATH_MESSAGE2 + pathList);

            boolean networkIsConnected = network.isConnected();
            System.out.println ("is connected: " + networkIsConnected);
        if (networkIsConnected)
            System.out.println ("spanning tree: " + network.getMinimumSpanningTree());

        System.out.println ("neighbors of " + start + ": "
                            + network.neighbors (start));
        System.out.println ("is empty: " + network.isEmpty());
        System.out.println ("vertex count: " + network.size());
        System.out.println ("edge count: " + network.getEdgeCount());
        System.out.println ("contains Q:  " + network.containsVertex ("Q"));
        System.out.println ("contains edge B-D: " + network.containsEdge ("B", "D"));
        System.out.println ("contains edge F-C: " + network.containsEdge ("F", "C"));
        System.out.println ("edge weight of A-B: "
                            +network.getEdgeWeight ("A", "B"));

        System.out.println ("\nbreadth-first iterating from " + start + ": ");
        Iterator<String> itr = network.breadthFirstIterator (start);
        while (itr.hasNext())
            System.out.print (itr.next() + " ");

        System.out.println ("\ndepth-first iterating from " + start + ": ");
        itr = network.depthFirstIterator (start);
        while (itr.hasNext())
            System.out.print (itr.next() + " ");

        System.out.println ("\niterating over network:");
        for (String s : network)
        System.out.print (s + " ");

        network.removeEdge ("B", "E");
        network.removeVertex ("D");
        System.out.println (REMOVAL_MESSAGE);
        for (String s : network)
        System.out.print (s + " ");

        pathList = network.getShortestPath (start, finish);
        System.out.println (SHORTEST_PATH_MESSAGE1 + start + " to " +
                            finish + SHORTEST_PATH_MESSAGE2 + pathList);
        } // try
        catch (FileNotFoundException e)
        {
            System.out.println (e);
        } // catch
    } // method run

} // class NetworkExample
```

Here is the file network.in1:

```
A F
A B 5
B A 5
```

```
A C 18
C A 18
A D 7
D A 7
B E 3
E B 3
C E 28
E C 28
C F 20
F C 20
D F 8
F D 8
D G 2
G D 2
G F 4
F G 4
```

And here is the output when the program was run:

```
The shortest path from A to F and its total weight are [A, D, G, F, 13.0]
is connected: true
spanning tree: {A={B=5.0, C=18.0, D=7.0}, B={A=5.0, E=3.0}, C={A=18.0},
               D={A=7.0, G=2.0}, E={B=3.0}, F={G=4.0}, G={D=2.0, F=4.0}}
neighbors of A: [B, C, D]
is empty: false
vertex count: 7
edge count: 18
contains Q:  false
contains edge B-D: false
contains edge F-C: true
edge weight of A-B: 5.0

breadth-first iterating from A:
A B C D E F G
depth-first iterating from A:
A D G F C E B
iterating over network:
A B C D E F G
iterating over network after removing B-E and D:
A B C E F G

The shortest path from A to F and its total weight are [A, C, F, 38.0]
```

15.6.2 Fields in the Network Class

As usual, the fundamental decisions about implementing a class involve selecting its fields. In the (directed) `Network` class, we will associate each vertex v with the collection of all vertices that are adjacent to v. That is, a vertex w will be included in this collection if ⟨v, w⟩ forms an edge. And to simplify determining the weight of that edge, we will associate each adjacent vertex w with the weight of the edge ⟨v, w⟩. In other words, we will associate each vertex v in the `Network` object with a map that associates each vertex w adjacent to v with the weight of the edge ⟨v, w⟩.

This suggests that we will need a map in which each value is itself a map. We have two `Map` implementations in the Java Collections Framework: `HashMap` and `TreeMap`. For inserting, removing, and searching in a `HashMap`, averageTime(n) is constant (if the Uniform Hashing Assumption holds), but worstTime(n) is linear in n for those three operations (even if the Uniform Hashing Assumption holds). It is the user's responsibility to ensure that the Uniform Hashing Assumption holds. Furthermore, the time (on average and in the worst case) to iterate through a `HashMap` object is linear in $n + m$, where m is the capacity of the hash table.

For a `TreeMap` object, the operations of insertion, removal, and search take logarithmic in n time, both on average and in the worst case. The elements can be ordered "naturally" by implementing the `Comparable` interface, or unnaturally with an implementation of an instance of the `Comparator` interface.

For both maps we choose the `TreeMap` class, whose performance in inserting, removing, and searching is superb even in the worst case, over the `HashMap` class, whose average-time performance is spectacular, but whose worst-time performance is pitiful.

In summary, the only field in the `Network` class is

```
protected TreeMap<Vertex, TreeMap<Vertex, Double>> adjacencyMap;
```

For each key v in `adjacencyMap`, each `value` will be a `TreeMap` object in which each key is a neighbor w of v and each value is the weight of the edge ⟨v, w⟩. For the sake of simplicity, we assume the vertices will be ordered "naturally". That is, the class corresponding to the type parameter `Vertex` must implement the `Comparable` interface.

Figure 15.32 shows a simple network, and Figure 15.33 shows an internal representation (which depends on the order in which vertices are inserted).

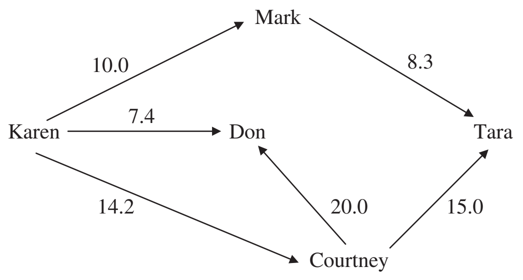


FIGURE 15.32 A network whose internal representation is shown in Figure 15.33

15.6.3 Method Definitions in the Network Class

Because the only field in the `Network` class is a `TreeMap` object, there are several one-line method definitions. For example, here is the definition of the `containsVertex` method:

```
public boolean containsVertex (Vertex vertex)
{
```

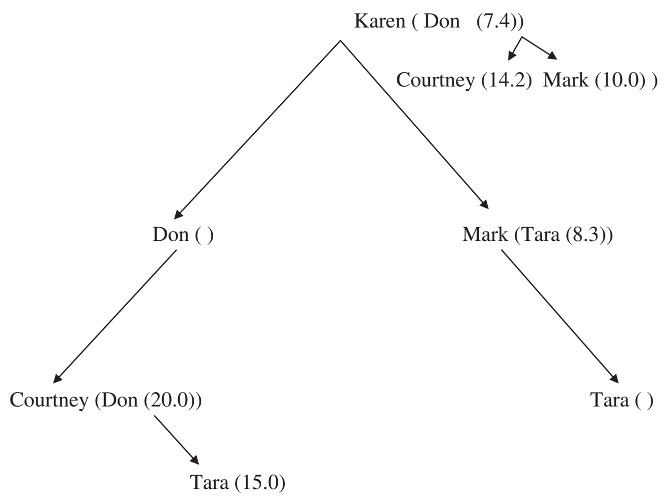


FIGURE 15.33 The internal representation of the network in Figure 15.32. The value associated with a key is shown in parentheses. "Don (_)", the left child of the root, indicates that "Don" has no neighbors

```
        return adjacencyMap.containsKey (vertex);
} // method containsVertex
```

In the `TreeMap` class, the worstTime(n) for the `containsKey` method is logarithmic in n, and so the worstTime(V, E) for the `containsVertex` method is logarithmic in V, where V represents the number of vertices and E represents the number of edges. The averageTime(V, E) for this method—and for all the other methods in the `Network` class—is the same as worstTime(V, E).

Adding a vertex to a `Network` object is straightforward:

```
public boolean addVertex (Vertex vertex)
{
        if (adjacencyMap.containsKey (vertex))
                return false;
        adjacencyMap.put (vertex, new TreeMap<Vertex, Double>());
        return true;
} // method addVertex
```

In the `TreeMap` class, the timing of the `containsKey` and `put` methods is logarithmic in the number of elements in the map, and so worstTime(V, E) is logarithmic in V.

The definition of `removeVertex` requires some work. It is easy to remove a `Vertex` object `v` from `adjacencyMap`, and thus remove its associated `TreeMap` object of edges going out from `vertex`. But each edge going *into* `v` must also be removed. To accomplish this latter task, we will iterate over the entries in `adjacencyMap` and remove from the associated neighbor map any edge whose vertex is `v`. Here is the definition:

```
public boolean removeVertex (Vertex v)
{
     if (!adjacencyMap.containsKey (v))
          return false;
```

```
        for (Map.Entry<Vertex, TreeMap<Vertex, Double>> entry: adjacencyMap.entrySet())
        {
             TreeMap<Vertex, Double> neighborMap = entry.getValue();

             neighborMap.remove (v);
        } // for each vertex in the network
        adjacencyMap.remove (v);
        return true;
   } // removeVertex
```

How long does this take? Iterating over the entries in adjacencyMap takes linear-in-V time, and removing a vertex from the neighbor map takes logarithmic-in-V time, so for the removeVertex method, worstTime(V, E) is linear-logarithmic in V.

Next, we'll develop the definition of an edge-related method. To count the number of edges in a Network object, we use the fact that the size of any vertex's associated neighbor map represents the number of edges going out from that vertex. So we iterate over all the entries in adjacencyMap, and accumulate the sizes of the associated neighbor maps. Here is the definition:

```
public int getEdgeCount()
{
   int count = 0;

  for (Map.Entry<Vertex, TreeMap<Vertex, Double>> entry : adjacencyMap.entrySet())
         count += entry.getValue().size();
   return count;
} // method getEdgeCount
```

The getEdgeCount method iterates over all entries in adjacencyMap, so worstTime(V, E) is linear in V.

Before we get to the methods that deal with the Network object as a whole, we need to say a few words about the BreadthFirstIterator class (similar comments apply to the DepthFirstIterator class).

The main issue with regard to the BreadthFirstIterator class is how to ensure a quick determination of whether a given vertex has been reached. To this end, we make reachable a TreeMap object, with Vertex keys and Boolean values. The heading and fields of this embedded class are:

```
protected class BreadthFirstIterator implements Iterator<Vertex>
{
       protected Queue<Vertex> queue;

       protected TreeMap<Vertex, Boolean> reachable;

       protected Vertex current;
```

From this point, the method definitions in the BreadthFirstIterator class closely follow the algorithms in Section 15.5.1 For the next() method, we iterate over the current vertex's neighbor map, and check if each neighbor has been marked as reachable. For the next() method, worstTime(V, E) is linear in $V \log V$.

Last, but by no means least, is the getShortestPath method to find the shortest path from vertex v1 to vertex v2. We start by filling in some of the details from the earlier outline. For the sake of speed, weightSum will be a TreeMap object that associates each Vertex object w with the sum of the weights of all the edges on the shortest path so far from v1 to w. Similarly, predecessor will be a TreeMap object that associates each vertex w with the vertex that is the immediate predecessor of w on the shortest path so far from v1 to w. The only unusual field declaration is for the priority queue of vertex-weight pairs. The PriorityQueue class has a single type parameter, so we create a VertexWeightPair class (nested in the Network class) for the type argument. The VertexWeightPair class will have vertex and weight fields, a two-parameter constructor to initialize those fields, and a compareTo method to order pairs by their weights.

The heading and variables in the getShortestPath method are as follows:

```
/**
 *  Finds a shortest path between two specified vertices in this Network
 *  object.
 *  The worstTime(V, E) is O(E log E).
 *
 *  @param v1 – the beginning Vertex object.
 *  @param v2 – the ending Vertex object.
 *
 *  @return a LinkedList object containing the vertices in a shortest path
 *          from Vertex v1 to Vertex v2.
 *
 */
public LinkedList<Object> getShortestPath (Vertex v1, Vertex v2)
{
        final double MAX_PATH_WEIGHT = Double.MAX_VALUE;

        TreeMap<Vertex,Double> weightSum = new TreeMap<Vertex,Double>();

        TreeMap<Vertex,Vertex> predecessor = new TreeMap<Vertex,Vertex>();

        PriorityQueue<VertexWeightPair> pq =
                                     new PriorityQueue<VertexWeightPair>();

        Vertex vertex,
                   to = null,
                   from;

        VertexWeightPair vertexWeightPair;

        double weight;
```

If either v1 or v2 is not in the Network, we return an empty LinkedList and we are done. Otherwise, we perform the initializations referred to in Section 15.5.4. Here is this initialization code:

```
if (v1 == null || v2 == null)
            throw new NullPointerException();
if (! (adjacencyMap.containsKey (v1)  && adjacencyMap.containsKey (v2)))
            return new LinkedList<Object>();
Iterator<Vertex> netItr = breadthFirstIterator(v1);
while (netItr.hasNext())
{
            vertex = netItr.next();
            weightSum.put (vertex, MAX_PATH_WEIGHT);
            predecessor.put (vertex, null);
} // initializing weightSum and predecessor
weightSum.put (v1, 0.0);
predecessor.put (v1, v1);
pq.add (new VertexWeightPair (v1, 0.0));
```

Now we find the shortest path, if there is one, from v1 to v2. As noted in Section 15.5.4, we have an outer loop that removes a vertex-weight pair <from, totalWeight> from pq and then an inner loop that iterates over the neighbors of from. The purpose of this inner loop is to see if any of those neighbors to can have its weightSum value reduced to from's weightSum value plus the weight of the edge <from, to>. Here are these nested loops:

```
   boolean pathFound = false;
   while (!pathFound  && !pq.isEmpty())
   {
      vertexWeightPair = pq.remove();
      from = vertexWeightPair.vertex;
      if (from.equals (v2))
             pathFound = true;
      else if (vertexWeightPair.weight <= weightSum.get(from))
      {
          for (Map.Entry<Vertex, Double> entry : adjacencyMap.get (from).entrySet())
          {
               to = entry.getKey();
               weight = entry.getValue();
               if (weightSum.get (from) + weight < weightSum.get (to))
               {
                  weightSum.put (to, weightSum.get (from) + weight);
                  predecessor.put (to, from);
                  pq.add (new VertexWeightPair (to,weightSum.get (to)));
               } // if
          } // while from's neighbors have not been processed
      } // else path not yet found
   } // while not done and priority queue not empty
```

All that remains is to create the path. We start by inserting v2 into an empty LinkedList object, and then, using the predecessor map, keep *prepending* predecessors until v1 is prepended. Finally, we add v2's total weight, as a Double object, to the end of this LinkedList object, and return the LinkedList object. Note that this LinkedList object has Object as the type parameter because the list contains both vertices and Double values. Here is the remaining code in the getShortestPath method:

```
   LinkedList<Object> path = new LinkedList<Object>();
   if (pathFound)
   {
         Vertex current = v2;
         while (!(current.equals (v1)))
         {
                path.addFirst (current);
                current = predecessor.get (current);
         } // while not back to v1
         path.addFirst (v1);
         path.addLast (weightSum.get (v2));
```

```
} // if path found
else
        path.addLast (-1.0);
return path;
```

We now estimate worstTime(V, E) for the getShortestPath method. We'll first establish upper bounds for worstTime(V, E). For each edge ⟨x, y⟩, y will be added to pq provided weightSum.get (x) + weight of ⟨x, y⟩ is less than weightSum.get (y)

Because the vertex at the end of each edge may be added to pq, the size of pq—and therefore the number of outer-loop iterations—is O(E). During each outer loop iteration, one vertex-weight pair is removed from pq, and this requires O(log E) iterations (in the remove() method). Also, for each removal of a vertex y, there is an inner loop in which each edge ⟨y, z⟩ is examined to see if z should be added to pq. The total number of edges examined during all of these inner-loop iterations is O(E).

From the previous paragraph, we see that the total number of iterations, even in the worst case, is O(E log $E + E$). We conclude that worstTime(V, E) is O(E log $E + E$). Also,

$$O(E \log E + E) = O(E \log E) = O(E \log V)$$

The last equality follows from the fact that log E $<= \log V^2 = 2 \log V$. We have worstTime(V, E) is O(E log V). If the network is connected (even as an undirected graph), $V - 1 <= E$, and so log $E >=$ log($V - 1$). Then all of the upper bounds on iterations from above are also lower bounds, and we conclude that worstTime(V, E) is $\Theta(E \log V)$.

Lab 23 introduces the best-known network problem, further explores the greedy-algorithm design pattern, and touches on the topic of *very hard* problems.

You are now ready for Lab 23: The Traveling Salesperson Problem

The final topic in this chapter is backtracking. In Chapter 5, we saw how backtracking could be used to solve a variety of applications. Now we expand the application domain to include networks and therefore, graphs and trees.

15.7 Backtracking Through A Network

When backtracking was introduced in Chapter 5, we saw four applications in which the basic framework did not change. Specifically, the same BackTrack class and Application interface were used for

1. searching a maze;
2. placing eight queens—none under attack by another queen—on a chess board (Programming Assignment 5.2);
3. illustrating that a knight could traverse every square in a chess board without landing on any square more than once (Programming Assignment 5.3);
4. solving a Sudoku puzzle (Programming Assignment 5.4);
5. solving a Numbrix puzzle (Programming Assignment 5.5).

A network (or graph or tree) is also suitable for backtracking. For example, suppose we have a network of cities. Each edge weight represents the distance, in miles, between its two cities. Given a start city and a finish city, find a path *in which each edge's distance is less than the previous edge's distance*. Figure 15.34 has sample data; the start and finish cities are given first, followed by each edge:

Figure 15.35 depicts the network generated by the data in Figure 15.34.

One solution to this problem is the following path:

$$\text{Boston} \xrightarrow{214} \text{NewYork} \xrightarrow{168} \text{Harrisburg} \xrightarrow{123} \text{Washington}$$

A lower-total-weight solution is:

$$\text{Boston} \xrightarrow{279} \text{Trenton} \xrightarrow{178} \text{Washington}$$

```
Boston Washington
Albany Washington 371
Boston Albany 166
Boston Hartford 101
Boston NewYork 214
Boston Trenton 279
Harrisburg Philadelphia 106
Harrisburg Washington 123
NewYork Harrisburg 168
NewYork Washington 232
Trenton Washington 178
```

FIGURE 15.34 A network: The first line contains the start and finish cities; each other line contains two cities and the distance from the first city to the second city

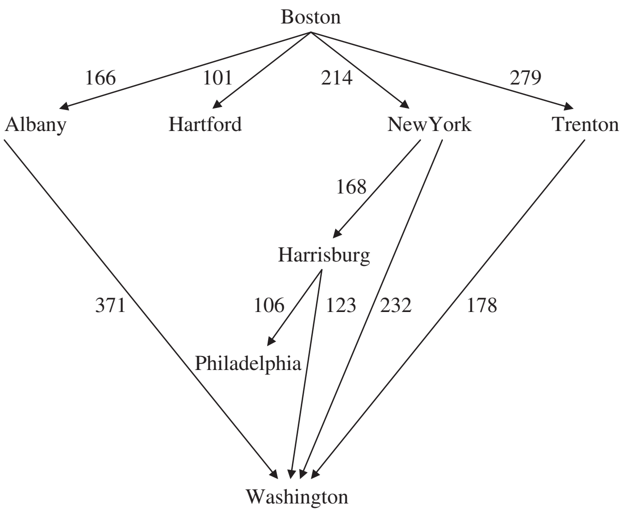


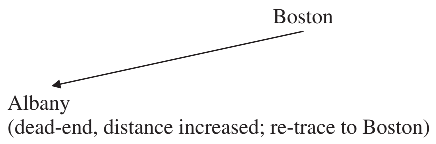
FIGURE 15.35 A network of cities; each edge weight represents the distance between the cities in the edge

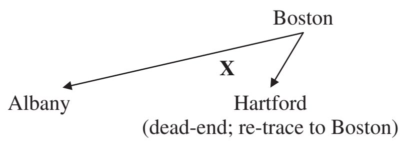
The lowest-total-weight path is illegal for this problem because the distances increase (from 214 to 232):

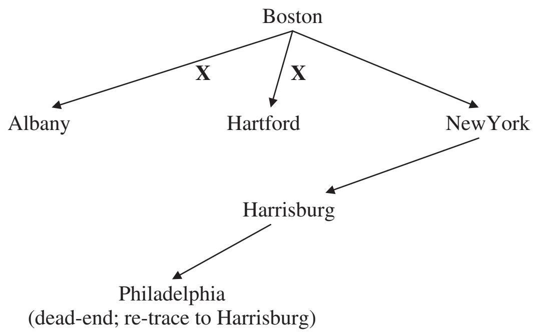
$$\begin{tikzcd}\text{Boston} \arrow[r, "214"] & \text{NewYork} \arrow[r, "232"] & \text{Washington}\end{tikzcd}$$

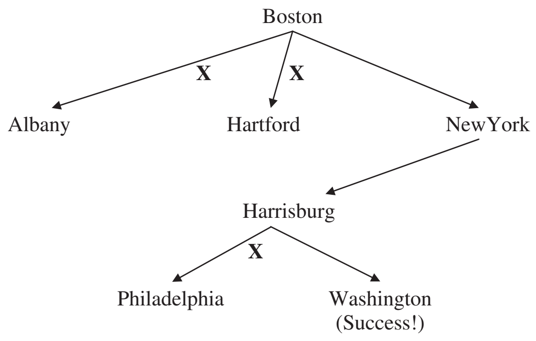
When a dead-end is reached, we can backtrack through the network. The basic strategy with backtracking is to utilize a depth-first search starting at the start position. At each position, we iterate through the neighbors of that position. The order in which neighboring positions are visited is the order in which the corresponding edges are initially inserted into the network. So we are guaranteed to find a solution path if one exists, but not necessarily the lowest-total-weight solution path.

Here is the sequence of steps in the solution generated by backtracking:









The framework introduced in Chapter 5 supplies the `BackTrack` class and `Application` interface. What about the `Position` class? That class must be modified: row and column have no meaning in a network. And the `MazeUser` and `Maze` classes must be revised as well. The details are left as Programming Project 15.2.

SUMMARY

An ***undirected graph*** consists of a collection of distinct elements called ***vertices***, connected to other elements by distinct vertex-pairs called ***edges***. If the pairs are ordered, we have a ***directed graph***. A ***tree***, sometimes called a ***directed tree***, is a directed graph that either is empty or has an element, called the ***root element*** such that:

1. There are no edges coming into the root element;
2. Every non-root element has exactly one edge coming into it;
3. There is a path from the root to every other element.

A ***network*** (or undirected network) is a directed graph (or undirected graph) in which each edge has an associated non-negative number called the ***weight*** of the edge. A network is also referred to as a "weighted digraph" (or "weighted undirected graph").

Some of the important graph/network algorithms are:

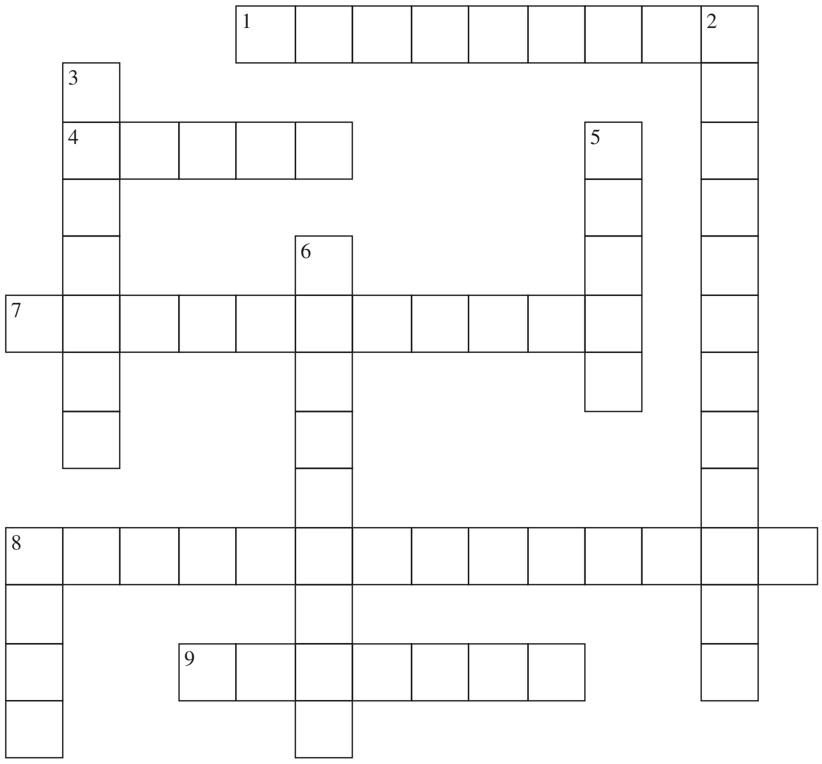
1. Breadth-first iteration of all vertices reachable from a given vertex;
2. Depth-first iteration of all vertices reachable from a given vertex;
3. Determining if a given graph is connected, that is, if for any two vertices, there is a path from the first vertex to the second;
4. Finding a minimum spanning tree for a network;
5. Finding a shortest path between two vertices in a network.

One possible design of the `Network` class associates, in a `TreeMap` object, each vertex with its neighbors. Specifically, we declare

```
TreeMap<Vertex, TreeMap<Vertex,
  Double>> adjacencyMap;
```

Here, `adjacencyMap` is a `TreeMap` object in which each key is a `Vertex` object `v` and each `value` is a `TreeMap` object in which each key is a `Vertex` object `w` and each value is a `Double` object `weight`, where `weight` is the weight of edge ⟨`v`, `w`⟩. Some network problems can be solved through backtracking. The iteration around a given position corresponds to an iteration through the linked list of vertex-weight pairs that comprise the neighbors of a given vertex.

CROSSWORD PUZZLE



www.CrosswordWeaver.com

ACROSS

1. Adjacent vertices in a graph

4. For the `getShortestPath` method in the `Network` class,worstTime(V, E) is BigTheta(___________).

7. The vertices v1, v2, … of a digraph are in _____order if vi precedes vj in the ordering whenever <vi, vj> forms an edge in the digraph.

8. An acyclic, directed network with a single source and a single sink

9. The type of the only field in the `Network` class

DOWN

2. In a connected, undirected network, a _______ is a weighted tree that consists of all of the vertices and some of the edges (and their weights) from the network.

3. Another name for a weighted graph

5. A path in which the first and last vertices are the same and there are no repeated edges

6. A graph is _____ if, for any given vertex, there is a path from that vertex to any other vertex in the graph.

8. In a graph, a sequence of vertices in which each successive pair is an edge

CONCEPT EXERCISES

15.1 Draw a picture of the following undirected graph:

Vertices: A, B, C, D, E

Edges: {A, B}, {C, D}, {D, A}, {B, D}, {B, E}

15.2 **a.** Draw an undirected graph that has four vertices and as many edges as possible. How many edges does the graph have?

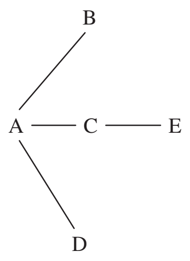
b. Draw an undirected graph that has five vertices and as many edges as possible. How many edges does the graph have?

c. What is the maximum number of edges for an undirected graph with V vertices, where V is any non-negative integer?

d. Prove the claim you made in part (c).
Hint: Use induction on V.

e. What is the maximum number of edges for a directed graph with V vertices?

15.3 Suppose we have the following undirected graph:



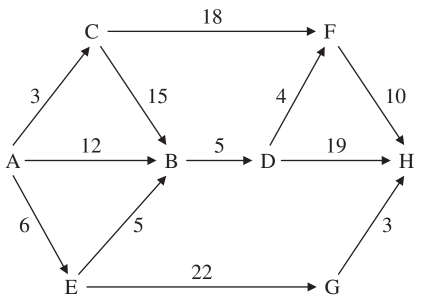
Assume the vertices were inserted into the graph in alphabetical order.

a. Perform a breadth-first iteration of the undirected graph.

b. Perform a depth-first iteration of the undirected graph.

15.4 Develop a high-level algorithm, based on the `isConnected( )` algorithm in Section 15.5.2, to determine if an undirected graph is connected.

15.5 For the network given below, determine the shortest path from A to H by brute force, that is, list all paths and see which one has the lowest total weight.



15.6 For the network given in Exercise 15.5, use Dijkstra's algorithm (`getShortestPath`) to find the shortest path from A to H.

15.7 Prim's algorithm (`getMinimumSpanningTree`) and Dijkstra's algorithm (`getShortestPath`) are ***greedy***: the locally optimal choice has the highest priority. In these cases, greed succeeds in the sense that the locally optimal choice led to the globally optimal solution. Do all greedy algorithms succeed for all inputs? In this exercise we explore coin-changing algorithms. In one situation, the greedy algorithm succeeds for all inputs. In the other situation, the greedy algorithm succeeds for some inputs and fails for some inputs.

Suppose you want to provide change for any amount under a dollar using as few coins as possible. Since "fewest" is best, the greedy (that is, locally optimal) choice at each step is the coin with largest value whose addition will not surpass the original amount. Here is a method to solve this problem:

```
/**
 *  Prints the change for a given amount, with as few coins (quarters, dimes,
 *  nickels and pennies) as possible.
 *
 *  @param amount - the amount to be given in change.
 *
 *  @throws NumberFormatException - if amount is less than 0 or greater than
 *          99.
 *
 */
public static void printFewest (int amount)
{
     if (amount < 0 || amount > 99)
          throw new NumberFormatException();

     int coin[ ] = {25, 10, 5, 1};

     final String RESULT =
          "With as few coins as possible, here is the change for ";

     System.out.println (RESULT + amount + ":");
     for (int i = 0; i< 4; i++)
          while (coin [i] <= amount)
          {
               System.out.println (coin [i]);
               amount -= coin [i];
          } // while
 } // printFewest
```

For example, suppose that `amount` has the value 62. Then the output will be

```
25
25
10
1
1
```

Five is the minimum number of coins needed to make 62 cents from quarters, nickels, dimes, and pennies.

a. Show that the above algorithm is optimal for any amount between 0 and 99 cents, inclusive. **Hint:** First, consider an amount of 0. Then amounts of 5, 10, or 25; then amounts of 15 or 20. Then add 25 to any of the amounts in the previous sentence. After all legal amounts divisible by 5 have been considered, consider all legal amounts that are not divisible by 5.

b. Give an example to show that a greedy algorithm is not optimal for all inputs if nickels are not available. That is, if we have

```
int coins[] = {25, 10, 1}

…
for (int i = 0; i < 3; i++)
…
```

then the algorithm will not be optimal for some inputs.

15.8 Ignore the direction of arrows in the figure for Exercise 15.5. Then that figure depicts an undirected network. Use Prim's algorithm to find a minimum spanning tree for that undirected network.

15.9 Ignore the direction of arrows and assume all weights are 1.0 in the figure for Exercise 15.5. Use Dijkstra's algorithm to find a shortest path (fewest edges) from A to H.

15.10 From the given implementation of the `Network` class, define the `removeEdge` method in the `UndirectedNetwork` class.

15.11 Re-order the edges in Figure 15.35 so that the solution path generated by backtracking is different from the lowest-total-weight solution path.

15.12 For the digraph in Figure 15.30, there were two topological orders given. Find three other topological orders. If a digraph has a cycle, can its vertices be put into a topological order? Explain.

PROGRAMMING EXERCISES

15.1 Modify the specification of Dijkstra's algorithm to find the shortest paths from a given vertex `v` to all other vertices in a network. Unit-test and define your method.

15.2 Modify the specification of Dijkstra's algorithm to find a longest path between two vertices. Unit-test and define your method. The method assumes that the network is acyclic. Why?

15.3 In the program in Section 15.6.1, the `getMinimumSpanningTree` method is not called if the network is not connected. In that program, comment out the line

```
if (networkIsConnected)
```

Run that program to create a network that is not connected but for which the `getMinimumSpanningTree` method still succeeds.
Hint: Not all edges are used in obtaining a minimum spanning tree.

15.4 In the `Network` class, unit-test and define a method to produce a topological order. Here is the specification:

```
/**
 *  Sorts this acyclic Network object into topological order.
 *  The worstTime(V, E) is O(V log V).
 *
 *  @return an ArrayList object of the vertices in topological order.  Note: if the
 *     size of the ArrayList object is less than the size of this Network object,
 *     the Network object must contain a cycle, and the ArrayList will not
 *     contain all the network's vertices in topological order.
```

```
     *
     */
    public ArrayList<vertex> sort()
```

Hint: First, construct a `TreeMap` object, `inCount`, that maps each vertex w to the number of vertices to which w is adjacent. For example, if the `Network` object has three edges of the form <`?`, `w`>, then `inCount` will map w to 3 (technically, to **`new`** **Integer (3)**). After `inCount` has been constructed, push onto a stack (or enqueue onto a queue) each vertex that `inCount` maps to 0. Then loop until the stack is empty. During each loop iteration,

1. pop the stack;
2. append the popped vertex `v` to an `ArrayList` object, `orderedVertices`;
3. decrease the value, in `inCount` 's mapping, of any vertex `w` such that <`v`, `w`> forms an edge;
4. if `inCount` now maps `w` to 0, push `w` onto the stack.

After the execution of the loop, the `ArrayList` object, containing the vertices in a topological order, is returned.

Programming Project 15.1

The Traveling Salesperson Problem

(This project assumes you are familiar with the material in Labs 9 and 23). Design, test, and implement a program to solve the Traveling Salesperson Problem. Your program will probably take exponential time. If, somehow, your program takes only polynomial time, be sure to claim your rightful place among the greatest computer scientists in the history of computing!

ANALYSIS: Each line of each input file will contain a pair of cities and the weight of the edge connecting the first city to the second city.

Assume that the input file tsp.in1 contains the following edges and weights:

```
a b 6
b a 6
a c 3
c a 3
a d 5
d a 5
b c 4
c b 4
b d 7
d b 7
c d 9
d c 9
```

System Test 1 (input in boldface):

Please enter the path for the file name that will hold the input: **tsp.in1**

```
The minimal-weight cycle is dbcad.
Its total weight is 19.
```

Assume that the input file tsp.in2 contains the following edges and weights:

```
a b 12
b a 12
a c 4
c a 4
a d 10
d a 10
a e 3
e a 3
b c 99
c b 9
b d 7
d b 7
b e 5
e b 5
c d 9
d c 9
c e 19
e c 19
d e 4
e d 4
```

System Test 2 (input in boldface):

Please enter the path for the file name that will hold the input: **tsp.in2**

```
The minimal-weight cycle is dbeacd.
Its total weight is 28.
```

Programming Project 15.2

Backtracking through a Network

Given a network in which each vertex is a city and each weight represents the distance between two cities, determine a path from a start city to a finish city in which *each edge's distance is less than the previous edge's distance*.

Analysis: Each city will be given as a String of at most 14 characters, with no embedded blanks. The first line of input will contain the start city and finish city. Each subsequent line—until the sentinel of "***"—will consist of two cities and the distance in miles from the first of those two cities to the second.

There is no input editing to be done.

The initial output will be the network. If there is no solution, the final output will be:

```
There is no solution.
```

Otherwise the final output will be:

```
There is a solution:
```

followed by the edges (from-city, to-city, distance) corresponding to the solution.

(continued on next page)

(continued from previous page)

System Test 1 (input in boldface):

```
        Please enter the start and finish cities, separated by a blank. Each city name should have
        no blanks and be at most 14 characters in length.
        Boston Washington

        Please enter two cities and their distance; the sentinel is ***
        Boston NewYork 214

        Please enter two cities and their distance; the sentinel is ***
        Boston Trenton 279

        Please enter two cities and their distance; the sentinel is ***
        Harrisburg Washington 123

        Please enter two cities and their distance; the sentinel is ***
        NewYork Harrisburg 168

        Please enter two cities and their distance; the sentinel is ***
        NewYork Washington 232

        Please enter two cities and their distance; the sentinel is ***
        Trenton Washington 178

        Please enter two cities and their distance; the sentinel is ***
        ***
The initial state is as follows:

        {Trenton = [Washington  178.0], NewYork = [Harrisburg  168.0,
        Washington 232.0], Washington = [ ], Harrisburg = [Washington  123.0],
        Boston = [NewYork   214.0, Trenton 279.0]}

A solution has been found:

                          FROM CITY        TO CITY         DISTANCE
                          Boston           NewYork           214.0
                          NewYork          Harrisburg        168.0
                          Harrisburg       Washington        123.0
```

System Test 2 (input in boldface):

```
        Please enter the start and finish cities, separated by a blank. Each city name should have
        no blanks and be at most 14 characters in length.
        Boston Washington

        Please enter two cities and their distance; the sentinel is ***
        Boston Trenton 279

        Please enter two cities and their distance; the sentinel is ***
        Boston NewYork 214
```

Please enter two cities and their distance; the sentinel is ***
Harrisburg Washington 123

Please enter two cities and their distance; the sentinel is ***
NewYork Harrisburg 168

Please enter two cities and their distance; the sentinel is ***
NewYork Washington 232

Pease enter two cities and their distance; the sentinel is ***
Trenton Washington 178

Please enter two cities and a weight; the sentinel is ***
****

The initial state is as follows:

{Trenton = [Washington 178.0], NewYork = [Harrisburg 168.0, Washington 232.0], Washington = [ ], Harrisburg = [Washington 123.0], Boston = [Trenton 279.0, NewYork 214.0]}

A solution has been found:

FROM CITY	TO CITY	DISTANCE
Boston	Trenton	279.0
Trenton	Washington	178.0

Note: The solution to this System Test is different from the solution to System Test 1 because in this System Test, the Boston-Trenton edge is entered before the Boston-NewYork edge.

Programming Project 15.3

Determining Critical Activities in a Project Network

Note: This project assumes the completion of Programming Exercise 15.5.

In the `Network` class, test and define a method to calculate the slack time for each activity in a project network. Here is the method specification:

```
/**
 *  Determines the slack time for each activity in this Project Network object.
 *  The worstTime(V, E) is O(V log V).
 *
 *  @return a TreeMap object that maps each activity, that is, each
 *           edge triple <v1, v2, weight>, to the slack time for that activity.
```

(continued on next page)

(continued from previous page)

```
         *
         */
        public TreeMap<EdgeTriple, Double> getSlackTimes()
```

Hint: First, create a `TreeMap` object, `inMap`, that maps each vertex `v` to the `TreeMap` object of vertex-weight pairs <`w`, `weight`> such that <`w`, `v`> forms an edge with weight `weight`. That is, `inMap` is similar to `adjacencyMap` except that each vertex `v` is mapped to the vertices coming into `v`; in `adjacencyMap`, each vertex `v` is mapped to the vertices going out from `v`.

Then loop through the `ArrayList` object (from Programming Exercise 15.4) in topological order. For each vertex `v`, use `inMap` to calculate ET(`v`). (The functional notation, "ET(`v`)", suggests a mapping, and `earliestTime` can be another `HashMap` object!) Then loop through the `ArrayList` object in reverse order to calculate LT(`v`) for each vertex `v`. Then calculate the slack time for each vertex.

Programming Project 15.4

An Integrated Web Browser and Search Engine, Part 7

This final part of the project involves increasing the relevance count of a given web page for each other web page that has a link to the given web page. For example, suppose the search is for "neural network", and the word "network" appears 4 times in the web page "browser.in11" and "neural" appears 2 times in "browser.in11". If the web pages "browser.in12" and "browser.in13" have a link to "browser.in11", that indicates that "browser.in11" is more relevant than if it had no links to it. So we will increase the relevance count for "browser.in11". For simplicity, we increase the relevance count by 1 for each file that has a link to "browser.in11". So the new relevance count for "browser.in11" will be 8: 4 for "network", 2 for "neural", and 2 for having two web pages that had links to "browser.in11".

To accomplish this change to the search engine, we need to determine, for each given web page, the list of other web pages that have a link to the given web page. So we will create a directed graph in which there is an arrow from vertex A to vertex B *if web page B has a link to web page A*. Then the number of web pages that have links to a web page A is just the number of neighbors of A.

To start with, develop a `NetworkConnectivity` class that scans each web page in search.in1 searching for links, and adds edges to a graph (for simplicity, an instance of the `Network` class) as described in the previous paragraph. After creating the graph, serialize it to "network.ser". This should be done just once, because the connectivity graph is independent of the search string. After "network.ser" has been created, you will need to de-serialize it for your search engine. Unlike "search.ser", you do not re-serialize "network.ser" because the network does not change as the result of your searches.

The contents of the web pages used in the System Tests are as follows:

home.in1:

```
        This is my home page
        Through caverns <a href = browser.in11>browser11</a> numberless to man
        Down to a neural network sunless sea.
```

browser.in10:

```
        In Xanadu did Kubla Khan
        A stately <a href = browser.in13>browser13</a> pleasure-dome decree:
```

```
        Where Alph, the sacred river, ran
        Through caverns measureless to man
        Down to a  sunless sea.
        neural network neural network neural network neural network
        And so it goes.
```

browser.in11:

```
        In Xanadu did <a href = browser.in12>browser12</a> Kubla Khan
        A stately pleasure-dome decree:
        Where Alph, the neural <a href =
        browser.in10>browser10</a> network sacred river, ran
        Through caverns neural network measureless to man
        Down to a network sunless sea.
        network
```

browser.in12:

```
        Neural surgeons have a network.  But the decree is a decree is
        a network <a href = browser.in11>browser11</a> and a network is a
        network, <a href = browser.in10>browser10</a> neural or not.
```

browser.in13:

```
        In Xanadu did Kubla Khan
        A stately <a href = browser.in11>browser11</a> pleasure-dome decree:
        Where Alph, the sacred river, ran
        Through caverns measureless to man
        Down to a sunless sea.
        neural network neural network neural network neural network
```

browser.in14:

```
        In <a href = browser.in12>browser12</a> Xanadu <a href =
        browser.in10>browser10</a>
        Did Kubla Khan
        A stately pleasure-dome decree:
        Where Alph, the neural network sacred river, ran
        Through caverns neural network measureless to man
        Down to a network sunless sea.
        network
```

Note: The above files are different from the same-named files in Programming Project 13.2. The original version of search.ser is available from the book's web site.

Incorporating hyperlink connectivity into a search engine was one of the innovations of Google, and the graduate students who created Google (Sergei Brin and Larry Page) are decabillionaires.

System Test 1:

(The end-user searches for "neural network")

(continued on next page)

(continued from previous page)

```
    Here are the results of the new search for "neural network"

      browser.in10  11
      browser.in13  9
      browser.in11  8
      browser.in12  8
      browser.in14  6

(The end-user searches for "network")
    Here are the results of the old search for "network"

      browser.in10  7
      browser.in12  6
      browser.in11  6
      browser.in13  5
      browser.in14  4

(The end-user clicks on the Back button twice)
    Here are the results of the new search for "neural network"

      browser.in10  11
      browser.in13  9
      browser.in11  8
      browser.in12  8
      browser.in14  6
```

System Test 2:

```
(The end-user searches for "neural network")
    Here are the results of the old search for "neural network"

      browser.in10  11
      browser.in13  9
      browser.in11  8
      browser.in12  8
      browser.in14  6

(The end-user searches for "network decree")
    Here are the results of the new search for "network decree"

      browser.in10  8
      browser.in12  8
      browser.in11  7
      browser.in13  6
      browser.in14  5
```

In System Test 2, the search for "neural network" is referred to as an "old" search because search.ser was updated when System Test 1 ended. The file search.ser contains search information, but not connectivity information.

APPENDIX 1

Additional Features of the JAVA Collections Framework

A1.1 Introduction

The Java Collections Framework has several features beyond those encountered so far in this book. This appendix focuses on two of those features: serialization and fail-fast iterators.

A1.2 Serialization

Suppose we have gone to the trouble of creating a large and complex `HashMap` object as part of a project. After we have used that `HashMap` object in an execution of the project, we might want to save the `HashMap`, on file, so that we can later resume the execution of the project without having to re-construct the `HashMap`. Fortunately, this is easily done with just a couple of statements.

How? All of the collection classes in the Java Collections Framework implement the `Serializable` interface that is in the package java.io. This interface has no methods, but merely provides information to the Java virtual machine about sending instances of the class to/from a stream (a sequence of bytes). Specifically, any class that implements the `Serializable` interface will be able to copy any object in the class to an output stream—that is, to "serialize" the elements in the object. "Deserialization" reconstructs the original object from an input stream.

For a simple example, suppose we have created an `ArrayList` object named `fruits`, whose elements are of type `String`. We can create an `ObjectOutputStream` and then write `fruits` to the `FileOutputStream` object whose path is "fruits.ser" as follows:

```
try
{
     ObjectOutputStream oos = new ObjectOutputStream (
                                  new FileOutputStream ("fruits.ser"));
     oos.writeObject (fruits);
} // try
catch (IOException e)
{
     System.out.println (e);
} // catch
```

The `ArrayList` object `fruits` has been ***serialized***, that is, is saved as a stream of bytes. The file qualifier, "ser", is an abbreviation of "serializable," but you are free to use any qualifier you want, or no qualifier. The definition of the `writeObject` method depends on the class of the object serialized. For example, here is the definition in the `ArrayList` class:

```
/**
 *  Save the state of the <tt>ArrayList</tt> instance to a stream (that
 *  is, serialize it).
```

```
     *  The worstTime(n) is O(n).
     *
     *  @serialData The length of the array backing the <tt>ArrayList</tt>
     *         instance is emitted (int), followed by all of its elements
     *         (each an <tt>Object</tt>) in the proper order.
     */
    private void writeObject (java.io.ObjectOutputStream s)
                 throws java.io.IOException
    {
           // Write out element count, and any hidden stuff
           s.defaultWriteObject();

           // Write out array length
           s.writeInt (elementData.length);

           // Write out all elements in the proper order.
           for (int i=0; i<size; i++)
                s.writeObject (elementData [i]);
    }
```

The size of the `ArrayList` object is saved first, and then the length of the `elementData` array field, so that an array of the exact same capacity can be created when the `ArrayList` object is reconstructed from the file. Finally, the elements in the `ArrayList` object are saved to the file.

In another program, or in a later execution of the same program, we can ***de-serialize*** `fruits`. To accomplish that task, we create an `ObjectInputStream` object to read the stream of bytes, from the `FileInputStream` whose path is "fruits.ser", into `fruits`:

```
    try
    {
           ObjectInputStream ois = new ObjectInputStream (
                                       new FileInputStream ("fruits.ser"));
           fruits = (ArrayList<String>)ois.readObject ();
    } // try
    catch (Exception e)
    {
           System.out.println (e);
    } // catch
```

In the `readObject` method, the first value read from the stream represents the size of the `ArrayList` object, and the next value represents the length of the underlying array, and finally, the individual elements are read, one at a time.

An object that is saved and then retrieved in another program (or later execution of the same program) is called ***persistent***. In this example, the `ArrayList` object `fruits` is persistent. And the same mechanism shown above can be used to create persistent instances of any of the other collection classes in the Java Collections Framework.

You can make your own classes serializable. Right after the class heading, add

```
    implements java.io.Serializable;
```

If all that have to be saved are fields, you needn't define `writeObject` and `readObject` methods: the default serialization/deserialization will handle fields. In the `ArrayList` example just listed, the defaults were not enough; the length of the array `elementData` and the elements themselves had to be saved. That is why the `ArrayList` class explicitly defines `writeObject` and `readObject` methods.

A1.3 Fail-Fast Iterators

Once an iterator has started iterating over a collection, that collection should not be structurally modified except by that iterator. A ***structural modification*** is either an insertion or removal; accessors and mutators do not structurally modify a collection. First, we'll see how this prohibition against structural modification can be helpful to users, and then we'll look at how the prohibition is enforced in the Java Collections Framework.

The following `main` method creates a small `LinkedList` object. During an iteration over that object, the object modifies itself, and then the iteration continues.

```
public static void main (String[ ] args)
{
      LinkedList<String> list = new LinkedList<String>();

      list.add ("humble");
      list.add ("meek");
      list.add ("modest");

      Iterator<String> itr = list.iterator();
      System.out.println (itr.next());   // prints "humble"
      list.remove ("modest");
      System.out.println (itr.next());   // prints "meek"?
      System.out.println (itr.next());   // ???
} // method main
```

The program constructs a `LinkedList` object, `list`, of three elements. An iterator, `itr`, starts iterating over `list`. When `itr` calls its `next()` method, the element "humble" at index 0 is returned, and `itr` advances to index 1. At this point, `list` removes the element at index 2. The second call to the `next()` method should be flagged as illegal. Why? The element "meek" at index 1 *could* be returned, but `itr` should not be allowed to advance to and print the element at index 2 because there is no longer an element at index 2. So what is best for the programmer is to have the error detected when the second call to the `next()` method is made, rather than having the error detected later in the program.

And that is exactly what happens. When this program was run, the value "humble" was output, and there was an exception thrown: `ConcurrentModificationException`. This exception was thrown when the second call to the `next()` method was made.

The idea is this: Once you start iterating through a collection in the Java Collections Framework, you should not modify the collection except with messages to that iterator. Otherwise, the integrity of that iterator may be compromised, so `ConcurrentModificationException` is thrown. Such iterators are ***fail-fast***: the exception is thrown as soon as the iterator may be invalid. The alternative, waiting until the iterator is known to be invalid, may be far more difficult to detect.

The mechanism for making iterators fail-fast involves two fields: one in the collection, and one in the iterator. We have studied six `Collection` classes within the Java Collections Framework: `ArrayList`, `LinkedList`, `Stack`, `PriorityQueue`, `TreeSet`, and `HashSet`. Each of these classes has a `modCount` field[1] that is initialized to 0. Each time the collection is structurally modified, `modCount`—for “modification count”—is incremented by 1.

The iterator class embedded in the `Collection` class has an `expectedModCount` field, which is initialized to `modCount` in the iterator’s constructor. Whenever the iterator structurally modifies the collection (for example, with a call to `itr.remove()`), both `expectedModCount` and `modCount` are incremented by 1. Also, whenever the iterator object itself is modified (for example, with a call to `itr.next()` or `itr.remove()`), there is a test:

```
if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
```

If `modCount` and `expectedModCount` are unequal, that means the collection has been structurally modified, but not by the iterator. Then for example, as in the program at the beginning of this section, a call to the `next()` method might advance to an element that is no longer in the collection. Instead of returning a possibly incorrect value, the `next()` method throws `ConcurrentModificationException`.

If, for some reason, you want to bypass this fail-fast protection, you can catch `ConcurrentModificationException` and do nothing in the **`catch`** block.

[1]For the `ArrayList` and `LinkedList` classes, `modCount` is inherited from `AbstractList`. `TreeMap` and `HashMap` explicitly declare the `modCount` field. `TreeSet` and `HashSet` utilize the `modCount` field in the backing `map` field (an instance of `TreeMap` and `HashMap`, respectively).

Mathematical Background

APPENDIX 2

A2.1 Introduction

Mathematics is one of the outstanding accomplishments of the human mind. Its abstract models of real-life phenomena have fostered advances in every field of science and engineering. Most of computer science is based on mathematics, and this book is no exception. This appendix provides an introduction to those mathematical concepts referred to in the chapters. Some exercises are given at the end of the appendix, so that you can practice the skills while you are learning them.

A2.2 Functions and Sequences

An amazing aspect of mathematics, first revealed by Whitehead and Russell [1910], is that only two basic concepts are required. Every other mathematical term can be built up from the primitives ***set*** and ***element***. For example, an ordered pair < a, b > can be defined as a set with two elements:

$$< a, b > = \{a, \{a, b\}\}$$

The element a is called the ***first component*** of the ordered pair, and b is called the ***second component***.

Given two sets A and B, we can define a ***function*** f from A to B, written

$$f : A \rightarrow B$$

as a set of ordered pairs $< a, b >$, where a is an element of A, b is an element of B, and each element in A is the first component of exactly one ordered pair in f. Thus no two ordered pairs in a function have the same first element. The sets A and B are called the ***domain*** and ***co-domain***, respectively.

For example,

$$f = \{< -2, 4 >, < -1, 1 >, < 0, 0 >, < 1, 1 >, < 2, 4 >\}$$

defines the "square" function with domain { -2, -1, 0, 1, 2 } and co-domain { 0, 1, 2, 4 }. No two ordered pairs in the function have the same first component, but it is legal for two ordered pairs to have the same second component. For example, the pairs

<-1, 1> and <1, 1>

have the same second component, namely, 1.

If$< a, b>$ is an element of f, we write $f(a) = b$. This gives us a more familiar description of the above function: the function f is defined by

$$f(i) = i^2, \text{ for } i \text{ in } -2 \ldots 2.$$

Another name for a function is a ***map***. This is the term used in Chapters 12, 14, and 15 to describe a collection of elements in which each element has a unique ***key*** part and a ***value*** part. There is, in effect, a function from the keys to the values, and that is why the keys must be unique.

A ***finite sequence*** t is a function such that for some positive integer k, called the *length* of the sequence, the domain of t is the set { 0, 1, 2, . . . , k-1 }. For example, the following defines a finite sequence of length 4:

t(0) = "Karen"

t(1) = "Don"

t(2) = "Mark"

t(3) = "Courtney"

Because the domain of each finite sequence starts at 0, the domain is often left implicit, and we write

t = "Karen", "Don", "Mark", "Courtney"

A2.3 Sums and Products

Mathematics entails quite a bit of symbol manipulation. For this reason, brevity is an important consideration. An example of abbreviated notation can be found in the way that sums are represented. Instead of writing

$$x_0 + x_1 + x_2 + \cdots + x_{n-1}$$

we can write

$$\sum_{i=0}^{n-1} x_i$$

This expression is read as "the sum, as i goes from 0 to $n-1$, of *x sub i*." We say that i is the ***count index***. A count index corresponds to a loop-control variable in a **for** statement. For example, the following code will store in `sum` the sum of components 0 through `n-1` in the array `x`:

```
double sum = 0.0;

for (i = 0; i < n; i++)
   sum += x [i];
```

Of course, there is nothing special about the letter "i." We can write, for example,

$$\sum_{j=1}^{10} (1/j)$$

as shorthand for

$$1 + 1/2 + 1/3 + \cdots + 1/10$$

Similarly, if $n >= m$,

$$\sum_{k=m}^{n} (k2^{-k})$$

is shorthand for

$$m2^{-m} + (m+1)2^{-(m+1)} + \cdots + n2^{-n}$$

Another abbreviation, less frequently seen than summation notation, is product notation. For example,

$$\prod_{k=0}^{4} a[k]$$

is shorthand for

```
a [0] * a [1] * a [2] * a [3] * a [4]
```

A2.4 Logarithms

John Napier, a Scottish baron and part-time mathematician, first described logarithms in a paper he published in 1614. From that time until the invention of computers, the principal value of logarithms was in number-crunching: logarithms enabled multiplication (and division) of large numbers to be accomplished through mere addition (and subtraction).

Nowadays, logarithms have only a few computational applications—for example, the Richter scale for measuring earthquakes. But logarithms provide a useful tool for analyzing algorithms, as you saw (or will see) in Chapter 3 through 15.

We define logarithms in terms of exponents, just as subtraction can be defined in terms of addition, and division can be defined in terms of multiplication.

Given a real number $b > 1$, we refer to b as the ***base***. The ***logarithm***, base b, of any real number $x > 0$, written

$$\log_b x$$

is defined to be that real number y such that

$$b^y = x$$

For example, $\log_2 16 = 4$ because $2^4 = 16$. Similarly, $\log_{10} 100 = 2$ because $10^2 = 100$. What is $\log_2 64$? What is $\log_8 64$? Estimate $\log_{10} 64$.

The following relations can be proved from the above definition and the corresponding properties of exponents. For any real value $b > 1$ and for any positive real numbers x and y,

Properties of Logarithms

1. $\log_b 1 = 0$
2. $\log_b b = 1$
3. $\log_b (xy) = \log_b x + \log_b y$
4. $\log_b (x/y) = \log_b x - \log_b y$
5. $\log_b b^x = x$
6. $b^{\log_b x} = x$
7. $\log_b x^y = y \log_b x$

From these equations, we can obtain the formula for converting logarithms from one base to another. For any bases a and $b > 1$ and for any $x > 0$,

$$\begin{aligned}\log_b x &= \log_b a^{\log_a x} \{\text{by property 5}\}\\ &= (\log_a x)(\log_b a)\{\text{by property 7}\}\end{aligned}$$

The base e (≈ 2.718) has special significance in calculus; for this reason logarithms with base e are called *natural logarithms* and are written in the shorthand ln instead of $\log_e$.

To convert from a natural logarithm to a base 2 logarithm, we apply the base-conversion formula just derived. For any $x > 0$,

$$\ln x = (\log_2 x)(\ln 2)$$

Dividing both sides of this equation by ln 2, we get

$$\log_2 x = \ln x / \ln 2$$

We assume the function ln is predefined, so this equation can be used to approximate $\log_2 x$. Similarly, using base-10 logarithms,

$$\log_2 x = \log_{10} x / \log_{10} 2$$

The function ln and its inverse *exp* provide one way to perform exponentiation. For example, suppose we want to calculate x^y where x and y are of type **`double`** and $x > 0$. We first rewrite x^y:

$$\begin{aligned}x^y &= e^{\ln(x)^y} \{\text{by Property 6, above}\}\\ &= e^{y \ln x} \{\text{by Property 7}\}\end{aligned}$$

This last expression can be written in Java as

```
Math.exp (y * Math.ln (x))
```

or, equivalently, and without any derivation,

```
Math.pow (x, y)
```

A2.5 Mathematical Induction

Many of the claims in the analysis of algorithms can be stated as properties of integers. For example, for any positive integer n,

$$\sum_{i=1}^{n} i = n(n+1)/2$$

In such situations, the claims can be proved by the Principle of Mathematical Induction.

Principle of Mathematical Induction

Let $S_1, S_2, \ldots$ be a sequence of statements. If both of the following cases hold:

1. S_1 is true
2. For any positive integer n, whenever S_n is true, S_{n+1} is true

then the statement S_n is true for any positive integer n.

To help you to understand why this principle makes sense, suppose that $S_1, S_2, \ldots$ is a sequence of statements for which cases 1 and 2 are true. By case 1, S_1 must be true. By case 2, since S_1 is true, S_2 must be true. Applying case 2 again, since S_2 is true, S_3 must be true. Continually applying case 2 from this point, we conclude that S_4 is true, and then that S_5 is true, and so on. This indicates that the conclusion in the principle is reasonable.

To prove a claim by mathematical induction, we first state the claim in terms of a sequence of statements $S_1, S_2, \ldots$. We then show that S_1 is true—this is called the ***base case***. Finally, we need to prove case 2, the ***inductive case***.

Here is an outline of the strategy for a proof of the inductive case: let n be any positive integer and assume that S_n is true. To show that S_{n+1} is true, relate S_{n+1} back to S_n, which is assumed to be true. The remainder of the proof often utilizes arithmetic or algebra.

Example A2.1 Sum of Initial Sequence of Positive Integers

We will use the Principle of Mathematical Induction to prove the following:

Claim For any positive integer n,

$$\sum_{i=1}^{n} i = n(n+1)/2$$

Proof We start by stating the claim in terms of a sequence of statements. For $n = 1, 2, \ldots$, let S_n be the statement

$$\sum_{i=1}^{n} i = n(n+1)/2$$

1. *Base case*.

$$\sum_{i=1}^{1} i = 1 = 1(2)/2$$

Therefore S_1 is true.

2. *Inductive case*. Let n be any positive integer and assume that S_n is true. That is,

$$\sum_{i=1}^{n} i = n(n+1)/2$$

We need to show that S_{n+1} is true, namely,

$$\sum_{i=1}^{n+1} i = (n+1)(n+2)/2$$

We relate S_{n+1} back to S_n by making the following observation: The sum of the first $n+1$ integers is the sum of the first n integers plus $n+1$. That is,

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= \sum_{i=1}^{n} i + (n+1) \\
&= n(n+1)/2 + (n+1) && \text{//because } S_n \text{is assumed true} \\
&= n(n+1)/2 + 2(n+1)/2 \\
&= (n(n+1) + 2(n+1))/2 \\
&= (n+2)(n+1)/2
\end{aligned}$$

We conclude that S_{n+1} is true (whenever S_n is true). So, by the Principle of Mathematical Induction, the statement S_n is true for any positive integer n.

An important variant of the Principle of Mathematical Induction is the following:

Principle of Mathematical Induction—Strong Form

Let $S_1, S_2, \ldots$ be a sequence of statements. If both of the following cases hold:

1. S_1 is true
2. For any positive integer n, whenever $S_1, S_2, \ldots, S_n$ are true, S_{n+1} is true

then the statement S_n is true for any positive integer n.

The difference between this version and the previous version is in the inductive case. Here, when we want to establish that S_{n+1} is true, we can assume that $S_1, S_2, \ldots, S_n$ are true.

Before you go any further, try to convince (or at least, persuade) yourself that this version of the principle is reasonable. At first glance, you might think that the strong form is more powerful than the original version. But in fact, they are equivalent.

We now apply the strong form of the Principle of Mathematical Induction to obtain a simple but important result.

Example A2.2 Number of Iterations of "Halving" Loop

Show that for any positive integer n, the number of iterations of the following loop statement is $floor(\log_2 n)$:

```
while (n > 1)
     n = n / 2;
```

Proof For $n = 1, 2, \ldots$, let $t(n)$ be the number of loop iterations. For $n = 1, 2, \ldots$, let S_n be the statement:

$$t(n) = floor(\log_2 n)$$

1. *Base case*. When $n = 1$, the loop is not executed at all, and so $t(n) = 0 = floor(\log_2 n)$. That is S_1 is true.
2. *Inductive case*. Let n be any positive integer and assume that $S_1, S_2, \ldots, S_n$ are all true. We need to show that S_{n+1} is true. There are two cases to consider:
   a. $n+1$ is even. Then the number of iterations after the first iteration is equal to $t((n + 1)/2)$. Therefore, we have

$$\begin{aligned} t(n+1) &= 1 + t((n+1)/2) \\ &= 1 + floor(\log_2((n+1)/2)) && \text{\{by the induction hypothesis\}} \\ &= 1 + floor(\log_2(n+1) - \log_2(2)) && \text{\{because log of quotient equals difference of logs\}} \\ &= 1 + floor(\log_2(n+1) - 1) \\ &= 1 + floor(\log_2(n+1)) - 1 \\ &= floor(\log_2(n+1)) \end{aligned}$$

   Thus S_{n+1} is true.

b. $n+1$ is odd. Then the number of iterations after the first iteration is equal to $t(n/2)$. Therefore, we have

$$\begin{aligned} t(n+1) &= 1 + t(n/2) \\ &= 1 + floor(\log_2(n/2)) && \{\text{by the induction hypothesis}\} \\ &= 1 + floor(\log_2 n - \log_2 2) && \{\text{log of quotient equals difference of logs}\} \\ &= 1 + floor(\log_2 n - 1) \\ &= 1 + floor(\log_2 n) - 1 \\ &= floor(\log_2 n) \\ &= floor(\log_2(n+1) && \{\text{since } \log_2(n+1) \text{ cannot be an integer}\} \end{aligned}$$

Thus S_{n+1} is true.

Therefore, by the strong form of the Principle of Mathematical Induction, S_n is true for any positive integer n.

Before we leave this example, we note that an almost identical proof shows that in the worst case for an unsuccessful binary search, the number of iterations is

$$floor(\log_2 n) + 1$$

In the original and "strong" forms of the Principle of Mathematical Induction, the base case consists of a proof that S_1 is true. In some situations we may need to start at some integer other than 1. For example, suppose we want to show that

$$n! > 2^n$$

for any $n >= 4$. (Notice that this statement is false for $n = 1$, 2, and 3.) Then the sequence of statements is S_4, S_5, ... For the base case we need to show that S_4 is true.

In still other situations, there may be several base cases. For example, suppose we want to show that

$$fib(n) < 2^n$$

for any positive integer n. (The method `fib`, defined in Lab 7, calculates Fibonacci numbers.) The base cases are:

$$fib(1) < 2^1$$

and

$$fib(2) < 2^2$$

These observations lead us to the following:

Principle of Mathematical Induction—General Form

Let K and L be any integers such that $K <= L$ and let S_K, S_{K+1}, ... be a sequence of statements. If both of the following cases hold:

1. S_K, S_{K+1},...,S_L are true
2. For any integer $n >= L$, if S_K, S_{K+1}, ..., S_n are true, then S_{n+1} is true.

then the statement S_n is true for any integer $n >= K$.

The general form extends the strong form by allowing the sequence of statements to start at any integer (K) and to have any number of base cases (S_K, S_{K+1},. . .,S_L). If $K = L = 1$, the general form reduces to the strong form.

The next two examples use the general form of the Principle of Mathematical Induction to prove claims about Fibonacci numbers.

Example A2.3 Upper Bound on nth Fibonacci Number

Show that

$$fib(n) < 2^n$$

for any positive integer n.

Proof For $n = 1, 2, \ldots$, let S_n be the statement

$$fib(n) < 2^n$$

In the terminology of the general form of the Principle of Mathematical Induction, $K = 1$ because the sequence starts at 1; $L = 2$ because there are two base cases.

1. $fib(1) = 1 < 2 = 2^1$, and so S_1 is true.
   $fib(2) = 1 < 4 = 2^2$, and so S_2 is true.

2. Let n be any integer ≥ 2 and assume that $S_1, S_2, \ldots, S_n$ are true. We need to show that S_{n+1} is true (that is, we need to show that $fib(n+1) < 2^{n+1}$).

By the definition of Fibonacci numbers,

$$fib(n+1) = fib(n) + fib(n-1), for n \geq 2.$$

Since $S_1, S_2, \ldots, S_n$ are true, we know that S_{n-1} and S_n are true. Thus

$$fib(n-1) < 2^{n-1}$$

and

$$fib(n) < 2^n$$

We then get

$$\begin{aligned} fib(n+1) &= fib(n) + fib(n-1) \\ &< 2^n + 2^{n-1} \\ &< 2^n + 2^n \\ &= 2^{n+1} \end{aligned}$$

And so fib (n+1) is true.

We conclude, by the general form of the Principle of Mathematical Induction, that

$$fib(n) < 2^n$$

for any positive integer n.

You could now proceed, in a similar fashion, to develop the following lower bound for Fibonacci numbers:

$$fib(n) > (6/5)^n$$

for all integers $n \geq 3$.

Hint: Use the general form of the Principle of Mathematical Induction, with $K = 3$ and $L = 4$.

Now that lower and upper bounds for Fibonacci numbers have been established, you might wonder if we can improve on those bounds. We will do even better. In the next example we verify an exact, closed formula for the nth Fibonacci number. A ***closed formula*** is one that is neither recursive nor iterative.

Example A2.4 Closed-Form Formula for nth Fibonacci Number

Show that for any positive integer n,

$$fib(n) = \frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^n - \left[\frac{1-\sqrt{5}}{2}\right]^n\right]$$

Before you look at the proof below, calculate a few values to convince yourself that the formula actually does provide the correct values.

Proof For $n = 1, 2, \ldots$, let S_n be the statement

$$fib(n) = \frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^n - \left[\frac{1-\sqrt{5}}{2}\right]^n\right]$$

Let $x = \frac{1+\sqrt{5}}{2}$ and let $y = \frac{1-\sqrt{5}}{2}$.

Note that

$$x^2 = \frac{(1+\sqrt{5})^2}{4} = \frac{1+2\sqrt{5}+5}{4} = \frac{3+\sqrt{5}}{2} = x+1$$

Similarly, $y^2 = y + 1$.

We now proceed with the proof.

1.

$$fib(1) = \frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^1 - \left[\frac{1-\sqrt{5}}{2}\right]^1\right] = 1, \text{ so } S_1 \text{ is true}$$

To show that S_2 is true, we proceed as follows:

$$\frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^2 - \left[\frac{1-\sqrt{5}}{2}\right]^2\right]$$
$$= \frac{1}{\sqrt{5}}(x^2 - y^2)$$
$$= \frac{1}{\sqrt{5}}(x + 1 - (y + 1))$$
$$= \frac{1}{\sqrt{5}}(x - y)$$
$$= \frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^1 - \left[\frac{1-\sqrt{5}}{2}\right]^1\right]$$

$=1$, which $fib(2)$ equals, by definition, and so S_2 is also true.

2. Let n be any positive integer greater than 1 and assume that $S_1, S_2, \ldots, S_n$ are true. We need to show that S_{n+1} is true; that is, we need to show that

$$fib(n+1) = \frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^{n+1} - \left[\frac{1-\sqrt{5}}{2}\right]^{n+1}\right]$$

By the definition of Fibonacci numbers,

$$fib(n+1) = fib(n) + fib(n-1)$$

Since S_n and S_{n-1} are true by the induction hypothesis, we have (using *x* and *y*)

$$fib(n) = \frac{1}{\sqrt{5}}(x^n - y^n)$$

and

$$fib(n-1) = \frac{1}{\sqrt{5}}(x^{n-1} - y^{n-1})$$

Substituting, we get

$$\begin{aligned} fib(n+1) &= \frac{1}{\sqrt{5}}(x^n + x^{n-1} - y^n - y^{n-1}) \\ &= \frac{1}{\sqrt{5}}(x^{n-1}(x+1) - y^{n-1}(y+1)) \\ &= \frac{1}{\sqrt{5}}(x^{n-1}x^2 - y^{n-1}y^2) \\ &= \frac{1}{\sqrt{5}}(x^{n+1} - y^{n+1}) \end{aligned}$$

Therefore S_{n+1} is true.

We conclude, by the general form of the Principle of Mathematical Induction, that S_n is true for any positive integer *n*.

The next example establishes part 1 of the Binary Tree Theorem in Chapter 9: the number of leaves is at most the number of elements in the tree plus one, all divided by 2.0. The induction is on the height of the tree and so the base case is for single-item tree, that is, a tree of height 0.

Example A2.5 Upper Bound on Number of Leaves in a Non-Empty Binary Tree

Let *t* be a non-empty binary tree, with leaves(*t*) leaves and *n*(*t*) elements. We claim that

$$leaves(t) \le \frac{n(t) + 1}{2.0}$$

Proof For $k = 0, 1, 2, \ldots$, let S_k be the statement: For any nonempty binary tree *t* of height *k*,

$$leaves(t) \le \frac{n(t) + 1}{2.0}$$

1. If *t* has height 0, then leaves(*t*) = n(*t*) = 1, and so

$$1 = leaves(t) \le \frac{n(t) + 1}{2.0} = 1$$

Therefore S_0 is true.

2. Let k be any integer ≥ 0, and assume that $S_0, S_1, \ldots, S_k$ are true. We need to show that S_{k+1} is true. Let t be a non-empty binary tree of height $k+1$. Both leftTree(t) and rightTree(t) have height $\leq k$, so both satisfy the induction hypothesis. That is,

$$leaves(leftTree(t)) \leq \frac{n(leftTree(t)) + 1}{2.0}$$

and

$$leaves(rightTree(t)) \leq \frac{n(rightTree(t)) + 1}{2.0}$$

But each leaf in t is either in leftTree(t) or in rightTree(t). That is,

$$leaves(t) = leaves(leftTree(t)) + leaves(rightTree(t))$$

Then we have

$$\begin{aligned} leaves(t) &\leq \frac{n(leftTree(t)) + 1}{2.0} + \frac{n(rightTree(t)) + 1}{2.0} \\ &= \frac{n(leftTree(t)) + n(rightTree(t)) + 1 + 1}{2.0} \end{aligned}$$

Except for the root element of t, each element in t is either in leftTree(t) or in rightTree(t), and so

$$n(t) = n(leftTree(t)) + n(rightTree(t)) + 1$$

Substituting this equation's left-hand side for its right-hand side in the previous inequality, we get

$$leaves(t) \leq \frac{n(t) + 1}{2.0}$$

That is, S_{k+1} is true.

Therefore, by the general form of the Principle of Mathematical Induction, S_k is true for any nonnegative integer k. This completes the proof of the claim.

The next example, relevant to Chapter 12, shows that the height of any red-black tree is logarithmic in n.

Example A2.6 The Height of a Red-Black Tree

As noted in Chapter 12, the `TreeMap` and `TreeSet` classes require only logarithmic time—even in the worst case—to insert, remove, or search. The reason for this speed is that those classes are based on red-black trees, whose height is always logarithmic in the number of elements in the tree. The proof that red-black trees are balanced utilizes the General Form of the Principle of Mathematical Induction.

Theorem The height of any red-black tree is logarithmic in n, the number of elements in the tree.

In order to prove this theorem, we first need a couple of preliminary results.

Claim 1. Let y be the root of a subtree of a red-black tree. The number of black elements is the same in a path from y to any one of its descendants with no children or one child.

Suppose x is the root of a red-black tree, and y is the root of a subtree. Let b_0 be the number of black elements from x (inclusive) to y (exclusive). For example, in Figure A2.1, if x is 50 and y is 131, $b_0 = 1$, the number of black elements in the path from 50 through 90; 131 is not counted in b_0.

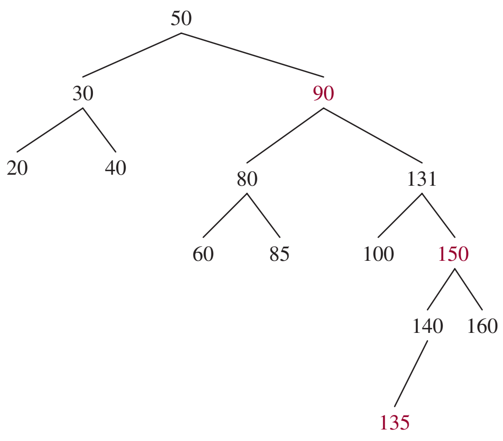


FIGURE A2.1 A red-black tree of 14 elements with maximum height, 5

Let b_1 be the number of black elements from y (inclusive) to any one of its descendants with no children or one child (inclusive), and let b_2 be the number of black elements from y (inclusive) to any other one of its descendants with no children or one child (inclusive). Figure A2.2 depicts this situation.

For example, in Figure A2.1, suppose that y is 131 and the two descendants of y are 100 and 135. Then $b_0 = 1$ because 50 is black; $b_1 = 2$ because 131 and 100 are black; $b_2 = 2$ because 131 and 140 are black.

In general, by the Path Rule for the whole tree, we must have $b_0 + b_1 = b_0 + b_2$. This implies that $b_1 = b_2$. In other words, the number of black elements is the same in any path from y to any of its descendants that have no children or one child. We have established Claim 1.

Now that Claim 1 has been verified, we can make the following definition. Let y be an element in a red-black tree; we define the ***black-height*** of y, written bh(y), as follows:

bh(y) = the number of black elements in any path from y to any descendant of y that has no children or one child.

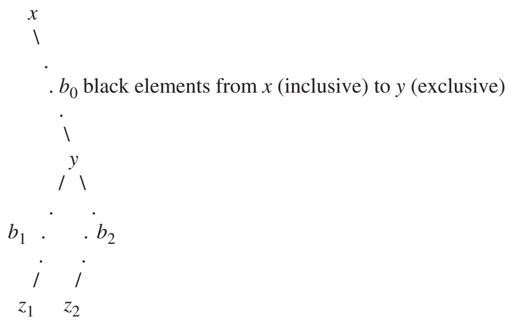

x
. b_0 black elements from x (inclusive) to y (exclusive)
y
b_1
b_2
z_1
z_2

FIGURE A2.2 Part of a red-black tree rooted at x; y is a descendant of x, and z_1, and z_2 are two arbitrarily chosen descendants of y that have no children or one child. Then b_0 represents the number of black descendants in the path from x up to but not including y; b_1 and b_2 represent the number of black elements in the path from y to z_1 and z_2, respectively

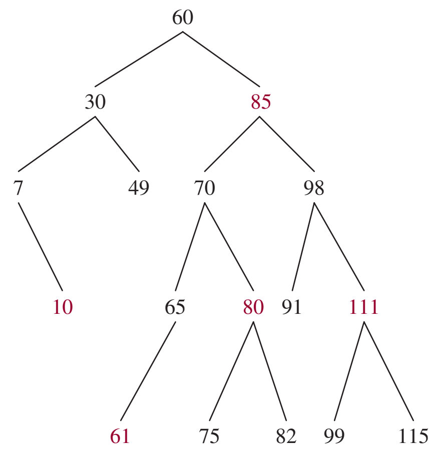


FIGURE A2.3 A red-black tree whose root has a black height of 3

By Claim 1, the number of black elements must be the same in any path from an element to any of its descendants with no children or one child. So black-height is well defined. For an example of black height, in Figure 12.3 from Chapter 12, the black-height of 50 is 2, the black height of 20, 30, 40, or 90 is 1, and the black-height of 10 is 0. Figure A2.3 shows a red-black tree in which 60 has a black height of 3 and 85 has a black height of 2

Claim 2. For any non-empty subtree *t* of a red-black tree,

$$n(t) \le 2^{bh(root(t))} - 1$$

(In the claim, $n(t)$ is the number of elements in *t*, and root(*t*) is the root element of *t*.) The proof of this claim is, as usual, by induction on the height of *t*.

Base case: Assume that height(t) = 0. Then $n(t) = 1$, and bh(root(t)) = 1 if the root is black and 0 if the root is red. In either case, $1 \ge bh(root(t))$. We have

$$n(t) = 1 = 2^1 - 1 \ge 2^{bh(root(t))} - 1$$

This proves Claim 2 for the base case.

Inductive case: Let *k* be any non-negative integer, and assume Claim 2 is true for any subtree whose height is $\le k$. Let *t* be a subtree of height $k + 1$.

If the root of *t* has one child, then the root must be black and the child must be red, and so $bh(root(t)) = 1$. Therefore,

$$n(t) \ge 1 = 2^1 - 1 = 2^{bh(root(t))} - 1$$

This completes the proof of the inductive case if the root of *t* has only one child.

Otherwise, the root of *t* must have a left child, v_1, and a right child, v_2. If the root of *t* is red, bh(root(t)) = bh(v_1) = bh(v_2). If the root of *t* is black, bh(root(t)) = bh(v_1) + 1 = bh(v_2) + 1. In either case,

$$\text{bh}(v_1) \ge \text{bh}(\text{root}(t)) - 1$$

and

$$\text{bh}(v_2) \ge \text{bh}(\text{root}(t)) - 1$$

The left and right subtrees of *t* have height $\leq k$, and so the induction hypothesis applies and we have

$$n(leftTree(t)) \geq 2^{bh(v_1)} - 1$$

and

$$n(rightTree(t)) \geq 2^{bh(v_2)} - 1$$

The number of elements in *t* is one more than the number of elements in leftTree(*t*) plus the number of elements in rightTree(*t*).

Putting all of the above together, we get:

$$\begin{aligned} n(t) &= n(leftTree(t)) + n(rightTree(t)) + 1 \\ &\geq 2^{bh(v_1)} - 1 + 2^{bh(v_2)} - 1 + 1 \\ &\geq 2^{bh(root(t))-1} - 1 + 2^{bh(root(t))-1} - 1 + 1 \\ &= 2 * 2^{bh(root(t))-1} - 1 \\ &= 2^{bh(root(t))} - 1 \end{aligned}$$

This complete the proof of the inductive case when the root of *t* has two children.

Therefore, by the Principle of Mathematical Induction, Claim 2 is true for all non-empty subtrees of red-black trees.

Finally, we get to show the important result that the height of any non-empty red-black tree is logarithmic in *n*, where *n* represents the number of elements in the tree.

Theorem For any non-empty red-black tree *t* with *n* elements, height(*t*) is logarithmic in *n*.

Proof Let *t* be a non-empty red-black tree. By the Red Rule, at most half of the elements in the path from the root to the farthest leaf can be red, so at least half of those elements must be black. That is,

$$\text{bh(root}(t)) \geq \text{height}(t)/2$$

From Claim 2,

$$\begin{aligned} n(t) &\geq 2^{bh(root(t))} - 1 \\ &\geq 2^{height(t)/2} - 1 \end{aligned}$$

From this we obtain

$$height(t) \leq 2 \log_2(n(t) + 1)$$

This inequality implies that height(*t*) is O(log *n*). By the Binary Tree Theorem in Chapter 9,

$$height(t) \geq \log_2((n(t) + 1)/2.0)$$

Combining these two inequalities, we conclude that height(*t*) is logarithmic in *n*.

This theorem states that red-black trees never get far out of balance. For an arbitrary binary search tree on the other hand, the height can be linear in *n*—for example, if the tree is a chain.

A2.6 Induction and Recursion

Induction is similar to recursion. Each has a number of base cases. Also, each has a general case that reduces to one or more simpler cases, which, eventually, reduce to the base case(s). But the direction is different. With recursion, we start with the general case and, eventually, reduce it to the base case. With induction, we start with the base case and use it to develop the general case.

CONCEPT EXERCISES

A2.1 Use mathematical induction to show that, in the Towers of Hanoi game from Chapter 5, moving n disks from pole a to pole b requires a total of $2^n - 1$ moves for any positive integer n.

A2.2 Use mathematical induction to show that for any positive integer n,

$$\sum_{i=1}^{n} Af(i) = A\sum_{i=1}^{n} f(i)$$

where A is a constant and f is a function.

A2.3 Use mathematical induction to show that for any positive integer n,

$$\sum_{i=1}^{n} (i * 2^{i-1}) = (n-1) * 2^n + 1$$

A2.4 Let n_0 be the smallest positive integer such that

$$fib(n_0) > n_0^2$$

1. Find n_0.

2. Use mathematical induction to show that, for all integers $n \geq n_0$,

$$fib(n) > n^2$$

A2.5 Show that fib(n) is exponential in n, specifically, $\Omega(((1 + \sqrt{5})/2)^n)$.

Hint: See the formula in Example A1.4 above. Note that the absolute value of

$$\left[\frac{1-\sqrt{5}}{2}\right]^n < 1$$

and so

$$\left[\frac{1-\sqrt{5}}{2}\right]^n$$

becomes insignificant for sufficiently large n.

A2.6 Show that

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

for any nonnegative integer n.

A2.7 Find the flaw in the following proof.

Claim All dogs have the same hair color.

Proof For $n = 1, 2, \ldots$, let S_n be the statement:
In any set of n dogs, all dogs in the set have the same hair color.

Base Case: If $n = 1$, there is only one dog in the set, so all dogs in that set have the same hair color. Thus, S_1 is true.

Inductive Case: Let n be any positive integer and assume that S_n is true; that is, in any set of n dogs, all the dogs in the group have the same hair color. We need to show that S_{n+1} is true. Suppose we have a set of $n + 1$ dogs:

$$d_1, d_2, d_3, \ldots, d_n, d_{n+1}$$

The set d_1, d_2, $d_3, \ldots$, d_n has size n, so by the Induction hypothesis, all the dogs in that set have the same hair color.

The set d_2, $d_3, \ldots$, d_n, d_{n+1} also has size n, so by the Induction hypothesis, all the dogs in that set have the same hair color.

But the two sets have at least one dog, d_n, in common. So whatever color that dog's hair is must be the color of all dogs in both sets, that is, of all $n + 1$ dogs. In other words, S_{n+1} is true.

Therefore, by the Principle of Mathematical Induction, S_n is true for any non-negative integer n.

Choosing a Data Structure

APPENDIX 3

A3.1 Introduction

This appendix serves as a brief summary of much of the material from Chapters 6–8 and 12–15. In particular, we will categorize the eight major collection classes (`ArrayList`, `LinkedList`, `Stack`, `Queue`[1], `TreeMap`, `PriorityQueue`, `HashMap`, `Network`) from those chapters in two different ways. The first classification will be by the ordering of the elements in the collection: time-based, index-based, comparison-based and hash-based. The second classification will be by the time to perform common operations on an element in the collection: access, insertion, deletion and search. For each such operation, averageTime(n) will be provided, that is, the average time (assuming each event is equally likely) to perform the operation in a collection of n elements. Whenever the estimate of averageTime(n) differs from the corresponding estimate for worstTime(n), both estimates will be given.

A3.2 Time-Based Ordering

For applications that utilize a time-based ordering, elements are removed from the collection in the same order they were inserted (First In, First Out), or in the reverse of that order (Last In, First Out). In the first case, an instance of the `LinkedList` class (Section 8.2.2) is appropriate, and an instance of the `Stack` class (Section 8.1) is called for in the second case.

For a `LinkedList` object, accessing the front element takes constant time, as does inserting at the back and deleting from the front. To search for a specific element in a `LinkedList` object, averageTime(n) is linear in n (as is worstTime(n)).

For a `Stack` object—implemented with an array in which the bottom of the stack is at index 0—accessing the top element takes constant time, as does popping the top element. To push an element onto the top of the stack, averageTime(n) is constant, but worstTime(n) is linear in n; the worst time occurs when the underlying array is already full before the insertion. To search for a specific element in a `Stack` object, averageTime(n) is linear in n (as is worstTime(n)).

A3.3 Index-Based Ordering

If the index of an element—0, 1, 2, and so on—is critical to the application, you have two choices: the `ArrayList` class (Section 6.2) or the `LinkedList` class (Section 7.3.1).

In the Java Collections Framework, the `ArrayList` class has an underlying array, and so accessing (or replacing) the element at a given index takes constant time in both the average and worst cases. Also, inserting or deleting at a given index entails moving the elements at higher indexes up (to make room for the insertion) or down (to close up the space of the deleted element). That is why insertion or deletion at a given index in an `ArrayList` object takes linear-in-n time in both the average and worst cases. Finally,

[1]Implemented by `LinkedList`.

searching for an element in an `ArrayList` object takes linear-in-*n* time, but only logarithmic-in-*n* time if the elements in the `ArrayList` object are in order according to a comparator.

Superficially, a `LinkedList` object is a pitiful choice: Both averageTime(*n*) and worstTime(*n*) are linear-in-*n* for accessing (or replacing) the element at a given index, as well as for inserting an element at—or deleting an element from—a given index. But `LinkedList` objects sparkle during an iteration: It takes only constant time to access (or replace) the "current" element, or to insert an element in front of the current element or to delete the current element. Finally, searching for an arbitrary element in a `LinkedList` object takes linear-in-n time.

The bottom line, as noted in Section 7.3.3 of Chapter 7, is this:

If the application entails a lot of accessing and/or replacing elements at widely varying indexes, an `ArrayList` object will be much faster than a `LinkedList` object. But if a large part of the application consists of iterating through a list and making insertions and/or removals during the iterations, a `LinkedList` object can be much faster than an `ArrayList` object.

A3.4 Comparison-Based Ordering

For many applications, the elements will be stored according to how they compare with each other. If the elements' class implements the `Comparable` interface, the comparisons are said to use the "natural" order. For an "unnatural" order—such as `Integer` elements in decreasing order, or `String` elements ordered by the length of the string—an implementation of the `Comparator` interface is appropriate. Section 11.3 of Chapter 11 has the details. In the Java Collections Framework's `PriorityQueue` class (Section 13.2), the focus of the comparisons is to identify the lowest valued (that is, highest priority) element. The framework's `TreeMap` (and `TreeSet`) class in Section 12.3 (and 12.5) utilizes a red-black tree to order all the elements in the collection.

A `PriorityQueue` object accesses the highest-priority element in constant time. Inserting an element takes constant time, on average, but worstTime(*n*) is linear in *n*: If the `add (E element)` method is called and the underlying array is full, that array must be resized. Removal of the highest-priority element takes logarithmic-in-*n* time in both the average and worst cases. The search for an arbitrary element takes linear-in-*n* time in both average and worst cases.

In the `TreeMap` class, each element is composed of a key—on which the ordering is based—and a value (the rest of the element). For both the average and worst cases, each of the following operations takes logarithmic-in-*n* time:

a. accessing a value, given a key;

b. putting a new element into the collection;

c. removing an element from the collection;

d. searching the collection for an element with a given key.

The `Network` (that is, weighted digraph) class (Section 15.6) is not part of the Java Collections Framework, but is well suited for a variety of applications, from scheduling to circuit-board wiring to analyzing web searches or social networks. In a network, the elements are called "vertices" or "nodes." The essential aspect in a network is the relationship, called an "edge" between two neighboring vertices. In the `Network` class, the only field is a `TreeMap` object in which each key is a vertex, and each value is the `TreeMap` object in which each key is a neighbor of the vertex, and each value is the weight of the edge connecting the vertex to the neighbor.

As you might expect, most of the usual operations have times that are similar to those of the `TreeMap` class. Specifically, with V representing the number of vertices and E the number of edges, the worstTime(V, E) and averageTime(V,E) are logarithmic in V for each of the following operations:

a. accessing the neighbors of a given vertex;

b. adding a new vertex to the network;

c. searching the network for a given vertex.

In removing a vertex from a network, we must also remove all edges going to that vertex, and for this operation worstTime(V, E) and averageTime(V, E) are linear-logarithmic in V.

A3.5 Hash-Based Ordering

In some applications—such as the maintenance of a symbol table by a compiler—it is important to be able to access, insert, and search for elements in constant time on average, even if these operations might rarely take linear-in-n time. The elements need not be stored in any recognizable order, such as chronological, indexed, or comparator-based.

In the Java Collection Framework's `HashMap` class (Section 14.3), each element consists of a key/value pair. If the Uniform Hashing Assumption (Section 14.3.2) holds, the averageTime(n) is constant for each of the following operations:

a. accessing a value, given a key;

b. putting a new element into the collection;

c. removing an element from the collection;

d. searching the collection for an element with a given key.

Unfortunately, even if the Uniform Hashing Assumption holds, worstTime(n) is sluggish: linear-in-n for each of those four operations.

Table A3.1 encapsulates the preceding information.

A3.6 Space Considerations

The space requirements of your application may play a role in your choice of a data structure, and here are a few points worth noting. If the data structure has an underlying array (namely `Stack`, `ArrayList`, `PriorityQueue`, and `HashMap`), a too-large capacity may waste space. And a too-small capacity can entail frequent resizing, each of which takes linear-in-n time. A further complication involves the `HashMap` class: The time-efficiency of hashing is proportional to the unused space!

For the `LinkedList` classes, each entry includes previous and next fields as well as the element field, so the entry consumes three times as much space as the element itself. For a `TreeMap`, each entry includes left, right, parent, and color fields as well as the element field, so the entry consumes more than four times as much space as the element itself (the color field can take up just one byte).

Table A3.1 Summary of Time Estimates in Appendix 3

Legend: con = constant;
lin = linear in *n*;
log = logarithmic in *n* (or in *V* for `Network`);
lin-log = linear-logarithmic in *V*;
worst times in boldface unless the same as average time

ORDERING	Access	Insertion	Deletion	Search
Time-Based				
FIFO (`LinkedList`)	con (front)	con (back)	con (front)	lin (any)
LIFO (`Stack`)	con (top)	con **[lin]** (top)	con (top)	lin (any)
Index-Based				
`ArrayList`	con (	lin at given index	lin)	lin* (any)
	* log if elements in comparison-based order			
`LinkedList`	con (at current index while iterating)	con	con	lin (any)
Comparison Based				
`TreeMap`	log	log	log	log
`PriorityQueue`	con (	con**[lin]** highest priority element	log)	lin
`Network`	log	log	lin-log	log
Hash Based				
`HashMap`	con **[lin]**	con **[lin]**	con **[lin]**	con **[lin]**

A3.7 The Best Data Structure?

As you can see from Table A3.1 and Section A3.6, there is no perfect data structure. Each data structure will be ideal for some applications and horrible for others. If even the possibility of linear-in-*n* time for the four common operations is unacceptable and space is not a factor, your best bet is the `TreeMap` class, whose worst times are logarithmic in *n*. And recall from Chapter 3 that the difference between constant time and logarithmic time is relatively small, but the difference between logarithmic time and linear time is relatively huge.

REFERENCES

ACM/IEEE-CS Joint Curriculum Task Force, *Computing Curricula 1991*, Association for Computing Machinery, New York, 1991.

Adel'son-Vel'skii, G.M., and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Mathematics*, Vol. 3, 1962: 1259–1263.

Albir, S. S., *UML in a Nutshell*, O'Reilly & Associates, Inc., Sebastopol, CA, 1998.

Andersson, A., T. Hagerup, S. Nilsson and R. Raman, "Sorting in Linear Time?", *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, 1995.

Arnold, K., and J. Gosling, *The Java Programming Language*, Addison-Wesley Publishing Company, Reading, MA, 1996.

Bailey, D. A., *Data Structures in Java for the Principled Programmer*, Second Edition, The McGraw-Hill Companies, Inc., Burr Ridge, IL, 2003.

Bayer, R. "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", *Acta Informatica*, **1**(4), 1972: 290–306.

Bentley, J. L. and M. D. McIlroy, "Engineering a Sort Function," *Software–Practice and Experience* , **23**(11), November 1993: 1249–1265.

Bloch, J., *Effective Java Programming Language Guide*, Addison-Wesley, Boston, 2001.

Collins, W. J., *Data Structures and the Standard Template Library*, McGraw-Hill, New York, NY, 2003.

Cormen, T., C. Leierson and R. Rivest, *Introduction to Algorithms*, Second Edition, McGraw-Hill, New York, NY, 2002.

Dale, N. "If You Were Lost on a Desert Island, What One ADT Would You Like to Have with You?", *Proceedings of the Twenty-First SIGCSE Technical Symposium*, **22**(1), March 1991: 139–142.

Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik* **1**, 1959: 269–271.

Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

Flajolet, P. and A. Odlyzko, "The Average Height of Binary Trees and Other Simple Trees," *Raports de Recherche*, #56, Institut National de Recherche en Informatique et en Informatique, February 1981.

Fowler, M., and K. Scott, *UML Distilled*, Second Edition, Addison-Wesley, Reading, MA, 2000.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA, 1995.

Goodrich, M., and R. Tamassia, *Data Structures and Algorithms in Java*, Second Edition, John Wiley & Sons, Inc., New York, 2001.

Gries, D., *Science of Programming*, Springer-Verlag, New York, 1981.

Guibas, L., and R. Sedgewick, "A Diochromatic Framework for Balanced Trees", *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, 1978: 8–21.

Habibi, M., *Java Regular Expressions: Taming the java.util.regex Engine*, Apress, Berkeley, CA, 2004.

Heileman, G. L., *Data Structures, Algorithms and Object-Oriented Programming*, The McGraw-Hill Companies, Inc., New York, 1996.

Hoare, C. A. R., "Quicksort," *Computer Journal*, **5**(4), April 1962: 10–15.

Huffman, D. A., "A Model for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, **40**, 1952: 1098–1101.

Knuth, D. E., *The Art of Computer Programming*, Volume 1: "Fundamental Algorithms," Second Edition, Addison-Wesley Publishing Company, Reading, MA, 1973.

Knuth, D. E., *The Art of Computer Programming*, Volume 2: "Seminumerical Algorithms," Second Edition, Addison-Wesley Publishing Company, Reading, MA, 1973.

Knuth, D. E., *The Art of Computer Programming*, Volume 3: "Sorting and Searching," Addison-Wesley Publishing Company, Reading, MA, 1973.

Kruse, R. L., *Data Structures and Program Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

Lewis, J., and W. Loftus, *Java Software Solutions: Foundations of Program Design*, Second Edition, Addison Wesley Longman, Inc., Reading, MA, 2000.

McIlroy, M., "A Killer Adversary for Quicksort," *Software–Practice and Experience* **29**(0), 1999: 1–4.

Meyer, B., *Object-oriented Software Construction*, Prentice-Hall International, London, 1988.

Newhall, T., and L. Meeden, "A Comprehensive Project for CS2: Combining Key Data Structures and Algorithms into an Integrated Web Browser and Search Engine," *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, March 2002: 386–290.

Noonan, R. E., "An Object-Oriented View of Backtracking," *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, March 2000: 362–366.

Pfleeger, S. L., *Software Engineering: Theory and Practice*, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1998.

Pohl, I., and C. McDowell, *Java by Dissection*, Addison Wesley Longman, Inc., Reading, MA, 2000.

Prim, R. C., "Shortest Connection Networks and Some Generalizations", *Bell System Technical Journal* **36**, 1957: 1389–1401.

Rawlins, G. J., *Compared to What? An Introduction to the Analysis of Algorithms*, Computer Science Press, New York, NY, 1992.

Riel, A. J., *Object-Oriented Design Heuristics*, Addison-Wesley Publishing Company, Reading, MA, 1996.

Roberts, S., *Thinking Recursively*, John Wiley & Sons, Inc., New York, 1986.

Sahni, S., *Data Structures, Algorithms, and Applications in Java*, The McGraw-Hill Companies, Inc., Burr Ridge, IL, 2000.

Schaffer, R., and R. Sedgewick, "The Analysis of Heapsort," *Journal of Algorithms* **14**, 1993: 76–100.

Shaffer, C., *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1998.

Simmons, G. J. (Editor), *Contemporary Cryptology: The Science of Information Integrity*, IEEE Press, New York, NY, 1992.

Wallace, S. P., *Programming Web Graphics*, O'Reilly & Associates, Sebastopol, CA, 1999.

Weiss, M. A., *Data Structures and Problem Solving Using Java*, Second Edition, Addison Wesley Longman, Inc., Reading, MA, 2002.

Whitehead, A. N., and B. Russell, *Principia Mathematica*, Cambridge University Press, Cambridge, England, 1910 (Volume 1), 1912 (Volume 2), 1913 (Volume 3).

Williams, J. W., "Algorithm 232: Heapsort", *Communications of the ACM* **7**(6), 1964: 347–348.

Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

Zwillinger, Daniel, *CRC Standard Mathematical Tables and Formulae*, Thirty-First Edition, Chemical Rubber Company, Cleveland, OH, 2002.

INDEX

A

Abstract class, 137–139
    versus fully defined class and interface, 139–140
Abstract data type, 31
Abstract method, 29
abstract modifier, 138
Abstraction,
    data, 27–28
Accessor method, 57
Activation record, 209, 334
Active method, 208
Acyclic graph, 646, 648, 667, 669, 693–694
Adjacent vertices, 644
Aggregation, 51
Algorithm, 105
    connectedness, 658–659
    divide-and-conquer, 476–477
    longest path, 667–669
    minimum spanning tree, 659–663
    shortest path, 663–667
Algorithm analysis, 105–121
Alternate implementation(s)
    of the BinarySearchTree class, 449–451
    of the HashMap class, 626–635, 640
    of the LinkedList class, 294–295, 323
A-maze-ing application, 195–208
Ancestor, 379, 439
Application Programming Interface (API), 6
Applications
    backtracking through a maze, 195–208
    backtracking through a network, 686–689, 695–697
    converting from infix to postfix, 338–343
    creating a symbol table, 617–625
    high-precision arithmetic, 251–257
    how compilers implement recursion, 334–337
    Huffman encoding, 573–590
    line editor, 300–315
    simulated car wash, 351–365
    spell checker, 530–536
    thesaurus, 517–525
Approval voting (project), 545–547
Argument, 19
ArrayList class, 234–251
    fields in, 248
    heading of, 246–247
    versus LinkedList class, 150, 282–285
ArrayList objects,
    serializing, 247, 701–702
    simple program using, 244–246
Arrays class
    binarySearch, 112–113, 180
    sort (Merge Sort), 470–476
    (Quick Sort), 477–489
assertArrayEquals, 62, 458
assertEquals, 62
Association, 51
Attribute in Unified Modeling Language, 49
Automatic garbage collection, 93
averageSpace(n), 106
averageTime(n), 106
    important assumption for, 106
AVL tree, 435–436
    height of, 436–438
AVLTree class, 438–441
    fixAfterDeletion method (project), 451–454
    fixAfterInsertion method (project), 455

B

BackTrack class, 193–195
Backtracking, 191–195
    a-maze-ing application, 195–198
    Eight Queens (project), 221–222
    Knight’s Tour (project), 222–224
    Numbrix (project), 227–231
    Sudoku (project), 225–226
    through a network, 686–689, 695–697
Balance factor, 438–439
Balanced binary search tree, 430–435
    AVL tree, 435–436
    red-black tree, 501–504
    rotations, 431–435
Base of a logarithm, 707
Base case, 709
Big-O notation, 106–116
Big-Omega notation, 116
Big-Theta notation, 117–119

BigInteger class, 251
Binary search, 112–113, 180–185
Binary search tree, 402
  AVL tree, 435–436
  red-black tree, 501–504
  rotations, 431–435
Binary tree, 377–392
  branches and leaves, 378–379
  chain, 384
  complete, 381–383
  decision tree, 468
  definition, 377
  external path length, 385
  full, 381
  Huffman tree, 576
  properties, 378–386
  traversals, 386–392
  two-tree, 381
Binary Tree Theorem, 383
BinarySearchTree class, 403–427
  add method, 416–418
  alternative implementation, 449–450
  contains method, 414, 420
  embedded Entry class, 411–412
  embedded TreeIterator class, 427–430
  fields, 411
  method specifications, 407–408
  remove method, 418–426
  testing, 410–411
  TreeIterator embedded class, 427–430
BinarySearchTreeTest class, 410–411
Binding
  dynamic, 49
  late, 49
Bit-wise operators,
  & (and), 605, 608
  ∧ (exclusive or), 608
  << (left shift), 382
  >> (right shift, 112
  >>> (unsigned right shift), 608
Black-height, 716–718
Bottom-up testing, 91
Boundary condition, 66
Boxing, 141
Branch, 378
Breadth-first iterator, 651–654, 683
break statement, 73
Bubble Sort, 463–464
Bytecode, 92

C

CalendarDate (project), 58
Car wash (application), 351–365
  randomizing the arrival times, 362–365
CarWash class, 354–361
  Car class, 356, 359
  CarWashTest class, 355
  CarWashUser class, 361–362
  fields, 355–357
  UML diagram, 357
catch block, 68–69, 73
  with **finally** block, 81
  with **try** block, 68–69
chain in a binary tree, 384
Chained hashing, 609–610
char type, 2–3
Checked exception, 77–78
Circularity in LinkedList class, 298
Circularity in SinglyLinkedList class (project), 322
Class, 3–7
  abstract, 137–139
  ArrayList, 234–251
  Arrays, 112–113, 180, 470–475, 477–489
  AVLTree, 438–441
  BackTrack, 193–195
  BigInteger, 251
  CalendarDate, 58
  Car, 356, 359
  CarWash, 354–361
  CarWashTest, 355
  CarWashUser, 361–362
  Collections, 256, 476
  Company, 35–37
  CompanyTest, 64–66
  constant, 60–61
  constructor, 4–5, 43, 92
  contiguous, 133
  DecimalToBinaryTest, 163
  Editor, 304–312
  EditorTest, 306–307
  EditorUser, 312–315
  embedded, 269
  Entry, 268–269
    in BinarySearchTree class, 412
    in HashMap class, 610
    in Huffman class, 580
    in LinkedList class, 295
    in SinglyLinkedList class, 268–269
    in TreeMap class, 513
  FactorialTest, 157–158
  Field, 1
  FullTimeEmployee, 32–34
  FullTimeEmployeeTest, 62–63
  Hasher, 620–625
  HashIterator, 615–619
  HashMap, 603–615
  HashSet, 625–626
  HourlyEmployee, 38–43
  HourlyEmployeeTest, 66–67

    Huffman, 581–586
    HuffmanTest, 582
    HuffmanUser, 586–589
    HuffmanUserTest, 587
    Immutable, 20
    implementing an interface, 32–34
    inheritance, 37–43
    instance variables, 59
    LinkedList, 295–300
    ListItr, 285–291
    Mutable, 21–22
    Network, 669–686
    NetworkTest, 677
    Nothing, 19
    PriorityQueue, 552–566
    Random, 122–126
    SalariedEmployee, 46
    SinglyLinkedList, 268–276
    SinglyLinkedListIterator, 276–279
    SpellChecker, 531–534
    SpellCheckerTest, 533
    SpellCheckerUser, 534–536
    String, 4–9
    Swap, 20–21
    Thesaurus, 518–521
    ThesaurusTest, 519–520
    ThesaurusUser, 521–525
    ThesaurusUserTest, 521–522
    TreeIterator, 427–430
    TreeMap, 509–517
    TreeSet, 525–530
    VeryLongInt, 252–257
    VeryLongIntTest, 253
    VeryLongIntUser, 257
    VeryLongIntUserTest, 257
Class constant, 60–61
Class variable, 60
clone method, 237–238
Cloneable interface, 246–247
close() method in PrintWriter class, 82, 85
Clustering in open-address hashing
    primary, 631–632
Co-domain (of a function), 705
Collection interface, 141–142
Collections class, 256, 476
Collision, 606
Collision handling
    with chained hashing, 609–611
    with open-address hashing, 626–634
Company class, 35–37
CompanyTest class, 64–66
Comparable interface, 179
Comparator interface, 465–468
Compiler: implementing recursion, 334–338
Complete binary tree, 381–383
Computer simulation, 350–351
Concordance (project), 543–545
Conditional operator (**? :**), 300
Connectedness, 646–647, 658–659
Constant (independent of *n*), 111, 117
Constant identifier, 60
Constructor, 4–5, 30, 43
    copy, 236–237
    default, 4, 43
    inheritance and, 43
Contiguous-collection class, 136, 234–251, 329–332, 552–566
Contiguous elements, 17, 136
Conversion formulas, 135
Converting from infix notation,
    to postfix notation, 338–343
    to prefix notation, 343–346
    tokens, 342–343
    transition matrix, 341–342
Copy
    shallow, 236–237
Copy constructor, 236–237
Correctness, 61
Cost of recursion, 209–210
Covariant subtyping, 237
Critical activity, 668
Critical path, 668
Current line (in line editor), 300
Cycle, 646, 669

D

Data abstraction, 27
    principle of, 27–28
Data structure, 31
Dead end, in backtracking, 191
Decimal to binary (application), 162–167
Decision tree, 468–469
Decoding a Huffman-encoded message (project), 595–597
Default constructor, 4, 43
Default visibility, 93
    versus visibility modifiers, 93–94, 100
Depth of an element, 381
Depth-first iterator, 654–658
Depth-first search, 390–391
Dequeue, 347
Deserialize, 702
Design pattern, 147
    divide-and-conquer, 476–477, 484
    greedy, 578, 663, 664, 686, 692–693
    iterator, 147
Dictionary, 505
Dijkstra's Algorithm, 663–667, 683–686
Directed graph (digraph), 647–648
Directed tree, 648–649
Directed weighted graph, 649

Distribution
  Poisson, 362–365
  uniform, 122
Divide-and-conquer algorithms, 476–477, 484
Domain (of a function), 705
Double hashing, 632–635, 638, 640
Double rotation, 434–435
Doubly linked list, 267–268
Dummy entry, 295
Dynamic binding, 49

E

Earliest time for an event, 668
Edge, 643
Editor class, 304–312
EditorTest class, 306–307
EditorUser class, 312–315
EditorUserTest class, 313
Efficiency of methods (estimating), 105–119
Eight Queens (project), 221–222
Embedded class, 269
Employee interface, 31–32
Encapsulation, 48
Encoding, 573
  prefix-free, 573–575
Enhanced **for** statement, 145–147
Enqueue, 347
Entry class
  in BinarySearchTree class, 412
  in HashMap class, 610
  in Huffman class, 580
  in LinkedList class, 295
  in SinglyLinkedList class, 268–269
  in TreeMap class, 513
  reason for **static** modifier, 296
Entry interface, 610
Equality
  of objects, 7–9
  of references, 8–9
  untestability of **double** values for, 66–67
equals method, 8, 94, 411
  in Object class, 94–95
  overriding, 95–96
Error
  in JUnit, 76
Evaluation of a condition (project), 371–374
Event-driven simulation, 354
Exception, 6, 68
  checked, 77–78
  handling of, 68–74
  propagating, 71–74
  testing, 69–71, 74–77
  thrown, 68–69
Execution frame, 159–161
Exponential-time method, 119
Expression tree, 389–390
Expression, 44–45
extends, 38
External path length, 385
External Path Length Theorem, 385–386

F

Factorial, 156–159
  execution frames, 159–161
FactorialTest class, 157–158
Failure
  in JUnit, 76
Fail-fast iterators, 703–704
Fairness in priority queue, 557, 566
Fast sorts
  Heap Sort, 567–573
  Merge Sort, 470–475
  Quick Sort, 477–489
Feedback, 350
Fibonacci numbers, 167
Fibonacci tree, 437
Field(s), 1
  in the ArrayList class, 248
  in the BinarySearchTree class, 411
  in the CarWash class, 355–357
  in the Editor class, 307–308
  in the Hasher class, 621
  in the HashMap class, 610–611
  in the LinkedList class, 295
  in the Network class, 680–681
  in the SinglyLinkedList class, 273
  in the SpellCheck class, 533
  in the Thesaurus class, 520
  in the TreeMap class, 512
  in the VeryLongInt class, 254
  pre-initialization of, 92
File
  input, 12–16
  output, 81–91
  sorting (project), 497–499
final modifier, 60
finally block, 81
Finite sequence, 706
First-in, first-out (FIFO)
  in a queue, 347
fixAfterDeletion method
  in AVLTree class (project), 455
  in TreeMap class (lab), 517
fixAfterInsertion method
  in AVLTree class (project), 451–454
  in TreeMap class (lab), 515
Full binary tree, 381
FullTimeEmployee class, 32–34
Function, 705

growth rate of, 117–119
and map, 705

G

Garbage, 92
automatic collection of, 92–93
Generating
a minimum spanning tree, 659–663
permutations
exercise, 215–216
lab experiment, 191
pseudorandom values, 122
Generic type, 140–141
Graph
acyclic, 646, 648, 667, 669, 693–694
algorithms, 651–669
cycle, 646, 669
directed, 647–648
edge, 643
neighbor, 644
path, 644
undirected, 643
vertex, 643
weighted, 649
Graph algorithms
breadth-first traversal, 651–654
connectedness, 658–659
depth-first traversal, 654–658
longest path, 667–669
minimum spanning tree, 659–663
shortest path, 663–667
Greedy algorithm, 578
Dijkstra's algorithm, 663–667, 683–686
Huffman's algorithm, 573–577
Prim's algorithm, 659–663
Growth rates, 117–119

H

Has-a, 47
Hash classes
HashMap, 603–615
HashSet, 625–626
Hasher class, 620–625
Hashing, 606
chained, 606–612
double, 632–635, 638, 640
identifiers, 617–625
Uniform Hashing Assumption, 609
HashIterator class, 615–619
HashMap class, 603–615
chaining, 606–612
fields in, 610–611
implementation of, 612–614
views (entrySet(), keyset(), values()), 616
HashSet class, 625–626
Heading
of the ArrayList class, 246
of the BinarySearchTree class, 406
of the HashMap class, 603
of the LinkedList class, 295
of the Network class, 670
Heap, 553–554
Heap Sort, 567–573
Height
black height, 716–718
of an AVL tree, 436–438
of a binary search tree, 434–436
of a binary tree, 339, 447–448
of a red-black tree, 503–504
Hexadecimal notation (exercise), 25
High-precision arithmetic (application), 251–257
HourlyCompany class, 44
HourlyEmployee class, 38–42
HourlyEmployeeTest class, 66–67
How fast can we sort, 468–469
Huffman
Huffman class, 581–586
Huffman encoding, 574
Huffman tree, 574
HuffmanTest class, 582
HuffmanUser class, 586–589
HuffmanUserTest class, 587

I

Identifier, 617
Imbalance ancestor, 439
Immutable, 20
Immutable class, 20
import directive, 12
In-place sorting, 461, 572
Index-related method, 147
Indirect recursion, 208–209
Induction and recursion, 719
Infinite recursion, 177
Infix notation, 338
converting to postfix, 338–343
converting to prefix, 343–346
Information hiding, 48
Inheritance, 37–43
and constructors, 43
multiple inheritance (not allowed), 139
Insertion Sort, 459–461
Instance variables, 59
instanceof operator, 96
Integrated Web Browser and Search Engine (project),
ArrayList, 264–266
GUI, 102–104
HashMap, 640–642
LinkedList, 328
Network, 698–700

PriorityQueue, 595–598
TreeMap, 548–550
Interface
Application Programming Interface, 6
Collection interface, 141–142
Comparable interface, 179
Comparator interface, 465–468
Employee interface, 31–32
Iterator interface, 143–144
List interface, 147
ListIterator interface, 285–287
Map interface, 504–508
Queue interface, 348
Set interface, 403
SortedMap interface, 508
SortedSet interface, 525
Interning of strings, 8
Intractable problem, 119
Invariant subtyping, 237
Is-a, 47
Iterative method, 161
Iterator, 143
bi-directional, 285
breadth-first, 651–654, 682
depth-first, 654–658
design pattern, 143
fail fast, 703–704

J

Java Collections Framework, 133
Java Virtual Machine, 92
java.util, 12
javadoc notation, 6–7
JUnit, 61–67, 69–70, 74–77
assertArrayEquals, 62, 458
assertEquals, 62
error, 76
exception testing, 74–75
fail(), 90
failure, 76
no testing of Big-O claims, 131
no testing **double** values for equality, 66
stub testing, 64, 76, 307
test first, 63–64

K

Key part in a map, 504
Knight's Tour (project), 222–224

L

Last-in, first-out (LIFO),
in a stack, 329
Late binding, 49
Latest time (for an event), 668
Leaf, 378
Leap year, 71
LeapYear class, 72
LeapYearTest class, 74–76
Left child, 379
Left rotation
followed by right rotation, 434–435
Length
of a path in a graph, 645
of a path in a network, 650
Let's Make a Deal (project), 131–132
Level of an element, 380–381
Line editor (application), 300–315
Linear in n, 117
Linear-logarithmic in n, 117
Link, 267
Linked list, 267
doubly linked list, 268
singly linked list, 268
LinkedList class, 295–300
alternate implementation (project), 323
iterators, 285–291
versus ArrayList class, 282–285, 291
List interface, 147
loadFactor, 611–612
Local variable, 9–12
explicit initialization, 10
Logarithmic in n, 117
Logarithms, 707–708
Longest path through a network, 667–668
Lower bound on sorting, 468–469

M

main method, 1, 35
Map, 504
Map interface, 504–508
embedded Entry interface, 610
Mathematical background
functions and sequences, 705–719
logarithms, 707–708
mathematical induction, 708–719
sums and products, 706–707
Mathematical induction. *See* Principle of Mathematical Induction
and recursion, 719
Mathematical model, 350
Maze (application), 195–208
Mean arrival time, 362
Median, 478
finding in linear-in-n time (exercise), 495–496
of medians, 488
Member of a class, 35
Member-selection operator, 7

Merge Sort
- in `Arrays` class, 470–476
- in `Collections` class, 476

Message
- in Huffman encoding, 573
- in object-oriented language, 7

Method, 1
- abstract, 29
- accessor, 57
- active, 208
- exponential-time, 119
- final, 61
- iterative, 161
- local variables in, 9–12
- mutator, 57
- overloading, 5
- overriding, 37–38
- polynomial-time, 119
- private, 39, 94
- protected, 39, 94
- public, 39, 94
- recursive, 208–209
- signature, 5
- static, 61
- testing of, 61–67, 69–70, 74–77
- timing of, 121–122
- virtual, 49

Method specification, 5–6
- in `javadoc` notation, 6–7

Minimum spanning tree, 659–663

Model, 350
- mathematical, 350
- physical, 350

Modifier. *See* Visibility modifier,
- **`abstract`**, 138
- **`final`**, 60–61
- **`private`**, 39, 94
- **`protected`**, 39, 94
- **`public`**, 39, 94
- **`static`**, 60–61
- **`transient`**, 248

Modularity, 3–4

Modulus operator (**`%`**), 2

`Mutable` class, 21–22

Mutator method, 57

N

Natural logarithm, 708

Neighbor, 644

Nested class, 269

Network, 649–689
- backtracking through, 686–689, 695–697
- finding longest path, 667–669
- finding minimum spanning tree, 659–663
- finding shortest path, 663–667
- `Network` class, 669–686
- `NetworkTest` class, 677
- project network, 667
- Traveling Salesperson Problem (project), 694–695

`new` operator, 4

node, 136

Notation
- Big-O, 106–116
- Big-Omega, 116
- Big-Theta, 117–119
- infix, 338
- `javadoc`, 5–7
- postfix, 339
- prefix, 343

`null` keyword, 7

`NullPointerException`, 78

O

`Object` class, 46–47
- overriding `equals` method of, 94–96

Object-oriented concepts, 27–49

Open-address hashing, 626–637
- double hashing, 632–635, 638, 640
- primary clustering, 631–632

Open-Closed Principle, 38

Operator
- bitwise and (**`&`**), 605, 608
- bitwise exclusive or (∧), 608
- bitwise left shift (**`<<`**), 382
- bitwise right shift (**`>>`**), 112
  - unsigned (**`>>>`**), 608
- conditional (**`? :`**), 300
- equality (**`==`**), 8
- **`instanceof`**, 96
- modulus (%), 2
- **`new`**,4,

Output file, 81–91

Overloading of methods, 5

Overriding of methods, 37–38

P

Package, 93

Parameterized type, 140–141

Parent, 379

Path, 381, 644
- cycle, 646, 669
- length, 645

Path length
- external, 385

Path Rule, 502

Performance requirements, 120

Permutation, generating
- exercise, 215–216
- lab experiment, 191

Persistent object, 702

Physical model, 350
Poisson distribution, 362–365
Polymorphism, 48–49
Polynomial-time method, 119
Postcondition, 6, 28
Postfix notation, 338
  converting from infix to, 338–343
Pre-initialization of fields, 92
Precondition, 28
Prefix notation, 343
  converting from infix to, 343–346
Prefix-free encoding, 573–575
Prim's Algorithm, 659–663
Primary clustering, 631–632
Primitive type, 2
Principle of Data Abstraction, 27–28
Principle of Mathematical Induction, 708
  general form, 711
  strong form, 710
Priority queue, 551
  heap implementation of, 553–556
  in Huffman encoding, 575
  in minimum-spanning-tree algorithm, 660
  in shortest-path algorithm, 664
`PriorityQueue` class, 552–566
`private` visibility, 39, 94
  versus other visibility modifiers, 93–94, 100
Probing
  quadratic, 635
Program modularity, 3–4
Project network, 667–669
  critical activity, 668
  critical path, 668
  earliest time of an event, 668
  latest time of an event, 668
  project length, 667–668
  sink, 667
  slack time of an activity, 668
  source, 667
  topological order, 669
  topological sorting, 669
Propagating exceptions, 71–74
`protected` visibility, 39, 94
  fields, 39
  methods, 42
  testing protected methods, 306
  versus other visibility modifiers, 39, 94
Pseudo-random number generator, 122
`public` visibility, 39, 94
  versus other visibility modifiers, 39, 94
Public-key cryptography, 251
Punched-card sorters, 489
`PurePriorityQueue` class, 552
`PureQueue` class, 349
`PureStack` class, 333

Q

Quadratic in n, 117
Quadratic probing, 635
Queue, 347
  computer simulation, 350–365
  dequeue, 347
  enqueue, 347
`Queue` interface, 348
  implemented by `LinkedList` class, 348
  implemented by `PriorityQueue` class, 552
Quick Sort, 477–484
  optimizations, 484–489
Quotient-offset collision handler, 632–633

R

Radix Sort, 489–492
Random access, 134
`Random` class, 122–126
Random number, 122
Random-number generator, 122
Randomizing
  arrival times in simulation, 362–365
  variable seed in `Random` class, 123
Reachable vertices, 651
Recursion, 155–210
  activation record, 209, 334
  backtracking, 191–195
  cost of, 209–210
  decimal to binary (application), 162–167
  definition, 209
  divide and conquer, 476–477
  execution frame, 159–161
  factorial, 156–159
  indirect, 208–209
  induction and, 719
  infinite, 177
  maze searching (application), 195–208
  stack frame, 304
  stack-based implementation, 334–338
  Towers of Hanoi (application), 167–179
  versus iteration, 161
Recursive definition, 377
Red Rule, 502
Red-black tree, 502
  height of, 504, 715–718
  in `TreeMap` implementation, 504
Reference variables, 4
  testing equality of (==), 8
Right child, 379
Right rotation, 433
  followed by left rotation, 435
Robust program, 68
Root element, 377, 648
Rotation, 431

double, 434–435
left, 431–433
properties of, 435
right, 433–434
Run-time analysis, 121–126
of sort methods (lab), 492

S

SalariedEmployee class (lab), 46
Scanner class, 12–17
delimiter, 12
InputMismatchException, 71
scanning over a string, 12
scanning over file input, 12
scanning over keyboard input, 12
token, 12
useDelimiter method, 16
whitespace, 12
Scope of an identifier, 11
Search engine. *See* Integrated Web Browser and Search Engine
Searching
an array, 179–181
binary search, 180–181, 601–602
framework to analyze, 599–600
in HashMap class, 614–615
red-black tree, 602–603
sequential, 179, 600–601
seed, 123
Selection Sort, 461–463
Sequence, 706
Sequential searching, 179, 600–601
Serialization, 701–703
Set interface, 403
Shallow copy, 236–237
Shortest path through a network, 663–667
Signature of a method, 5
Simple sorts, 458–464
Simulation, 350
event driven, 354
Singly linked list, 268
SinglyLinkedList class, 268–281
embedded SinglyLinkedListIterator class, 276–279
embedded Entry class, 268
expanding (projects), 319–322
field, 273
method definitions, 273–276
testing, 271–272
SinglyLinkedTest class, 271–272
Sink, 667
Slack time (for an activity), 668
SortedMap interface, 508
TreeMap implementation of, 509
Sorting
Bubble Sort, 463–464
fast sorts, 470–475, 477–489, 567–573
file sorting (project), 497–499
Heap Sort, 567–573
in-place sorting, 461, 572
Insertion Sort, 459–461
Merge Sort, 470–475
Quick Sort, 477–489
Radix Sort, 489–492
Selection Sort, 461–463
simple sorts, 458–464
stable sort, 458
summary, 493
topological sorting, 669
Source, 667
Spanning tree, 659
Specification of methods, 5
Speedo's car wash (application), 351–354
making more realistic, 362–365
Spell Checker (application), 530–531
SpellChecker class, 531–534
SpellCheckerTest class, 532–533
SpellCheckerUser class, 534–536
Splitting Rule, 112
Stable sort, 458
Stack, 329
in converting from infix-to-postfix, 334–338
in implementing recursion, 338–341
Stack class, 329–333
Stack frame, 334
Static constant, 60
static modifier, 60
Static variable, 60
Storage structures for collection classes, 136
String class, 4–9
constructors, 4
testing equality of String objects, 7–8
Stub, 64
Subclass, 37–38
Subclass-Substitution Rule, 44
Sums and products, 706–707
super, 40
Superclass, 37
Symbol table, 617
Synonym, 606
System test, 91

T

Testing
bottom up, 91
unit testing. *See* JUnit
system tests, 91
Thesaurus (application), 517–518
Thesaurus class, 518–521
ThesaurusTest class, 519–520
ThesaurusUser class, 521–525

ThesaurusUserTest class, 521–522
throw statement, 71
throws clause, 67–68
Timing, 121–122
Token, 342
Topological order, 669, 693–694
Topological sorting, 669
Towers of Hanoi (application), 167–178
  iterative version (project), 219–220
Trade-off, 119–121
  performance requirements, 120
  time *versus* space, 120
transient modifier, 248
Transition matrix, 341–342
Traveling Salesperson Problem (lab), 684
Traversals of a binary tree, 383–392
  breadthFirst, 391–392
  inOrder, 386–388
  postOrder, 387–390
  preOrder (= depth-first), 390–391
Tree
  balanced binary search tree, 430
  binary search tree, 402
  binary tree, 377
  complete tree (binary), 382
  decision tree (binary), 468
  directed tree, 648
  expression tree (binary), 389–390
  Fibonacci tree (binary), 437
  full tree (binary), 381
  Huffman tree (binary), 575
  red-black tree (binary), 502
  root, 377, 648
  spanning tree, 659
  subtree, 377
  two-tree (binary), 381
  undirected tree, 648
TreeMap class, 509–517
  approval voting (problem), 545–547
  building a concordance (problem), 543–545
  building a thesaurus (application), 518–525
  Comparable and Comparator interfaces, 510–511
  determining word frequencies (problem), 542–543
  Entry class, 513
  field in Network class, 680–681
  fields in, 512
  red-black tree, 504
  SortedMap interface, 509
  views (entrySet(), keyset(), values()), 507–508
TreeSet class, 525–530
  spell checker (application), 530–536
try block, 68–69
  with **catch** block, 68–69
  with **finally** block, 81
Two-tree, 381
Type parameter, 141

U
UML, *See* Unified Modeling Language
Unboxing, 141
Undirected graph, 643–646
  acyclic, 646
  adjacent, 644
  complete, 644
  connected, 646
  cycle, 646
  edge, 643
  neighbor, 644
  path, 644
  vertex, 643
Undirected, weighted tree, 676
Unified Modeling Language (UML), 49–52
Uniform distribution, 112
Uniform Hashing Assumption, 609
Upper bound, 107

V
Value part in a map, 504
Variable, 1, 2
  reference variable, 4
Vertex, 643
  comparing vertices, 680
  reachable, 651
Very long integers (application), 251–257
VeryLongInt class, 251–257
  field in, 254
VeryLongIntTest class, 253
VeryLongIntUser class, 257
VeryLongIntUserTest class, 257
Virtual method, 49
Visibility modifier, 39, 94
  default visibility, 93
  private, 39, 94
  protected, 39, 94
  public, 39, 94

W
Web browser. *See* Integrated Web Browser and
  Search Engine
  Weighted graph, 649
Word frequencies (problem), 542–543
worstSpace(*n*), 105
worstTime(*n*), 106
Wrapper class, 135
  conversions from/to, 135
Wrapper method, 158, 164, 169, 181