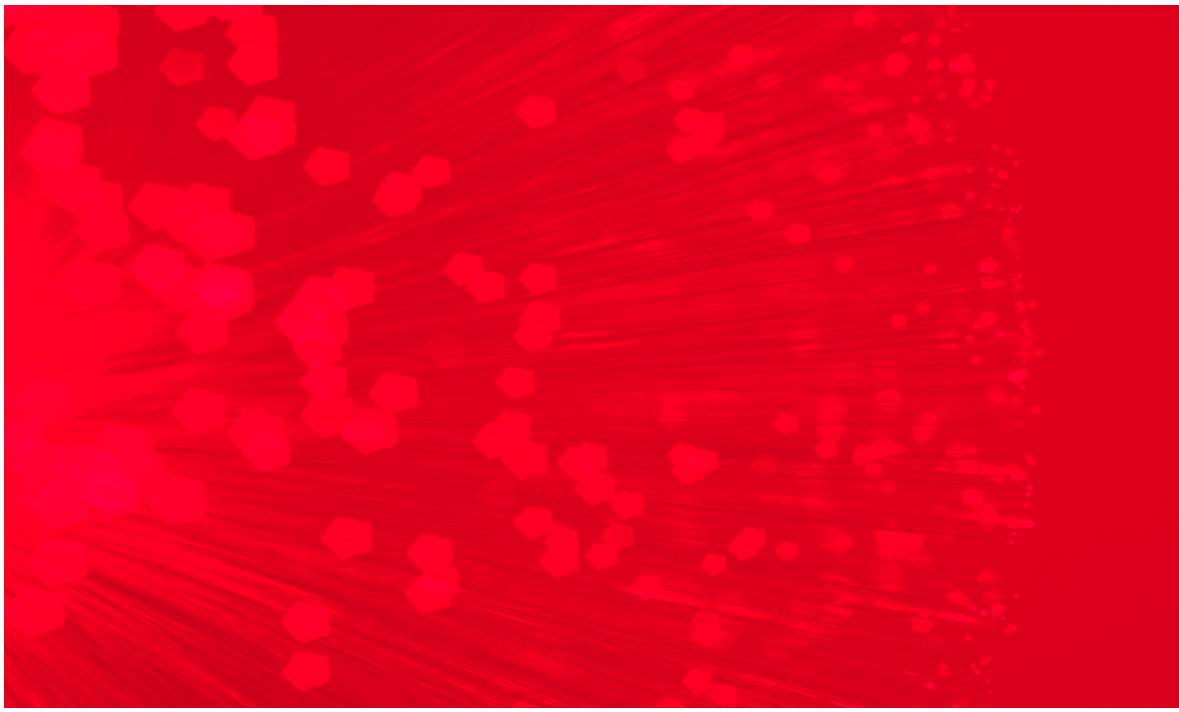
Data Structures

A Pseudocode Approach with C

Richard F. Gilberg & Behrouz A. Forouzan

SECOND EDITION



Data Structures: A Pseudocode Approach with C, Second Edition

Richard F. Gilberg & *Behrouz A. Forouzan*

Data Structures: A Pseudocode Approach with C, Second Edition
Richard F. Gilberg
Behrouz A. Forouzan

Senior Product Manager:
Alyssa Pratt

Senior Acquisitions Editor:
Amy Yarnevich

Production Editor:
BobbiJo Frasca

Senior Marketing Manager:
Karen Seitz

Associate Product Manager:
Mirella Misiaszek

Editorial Assistant:
Amanda Piantedosi

Senior Manufacturing Coordinator:
Trevor Kallop

Cover Designer:
Abby Scholz

Printed in the United States of America

1 2 3 4 5 6 7 8 9 QWT 09 08 07 06 05

For more information, contact Course Technology, 25 Thomson Place, Boston, Massachusetts, 02210.

Or find us on the World Wide Web at: www.course.com

ISBN-13: 978-0-534-39080-8
ISBN-10: 0-534-39080-3

Dedication Page

To my wife Evelyn for her support over the years.
Richard Gilberg

To Mark Bauer for his help when I needed it.
Behrouz Forouzan

Contents

Preface

The study of data structures is both exciting and challenging. It is exciting because it presents a wide range of programming techniques that make it possible to solve larger and more complex problems. It is challenging because the complex nature of data structures brings with it many concepts that change the way we approach the design of programs.

Because the study of data structures encompasses an abundant amount of material, you may find that it is not possible to cover all of it in one term. In fact, data structures is such a pervasive subject that you will find aspects of it taught in lower-division, upper-division, and graduate programs.

Features of This Book

Our primary focus in this text is to present data structures as an introductory subject, taught in a lower-division course. With this focus in mind, we present the material in a simple, straightforward manner with many examples and figures. We also de-emphasize the mathematical aspect of data structures, leaving the formal mathematical proofs of the algorithms for later courses.

Pseudocode

Pseudocode is an English-like presentation of the steps needed to solve a problem. It is written with a relaxed syntax that allows students to solve a problem at a level that hides the detail while they concentrate on the problem requirements. In other words, it allows students to concentrate on the big picture.

In addition to being an excellent design tool, pseudocode is also language independent. Consequently, we can and do use the same pseudocode design to implement an algorithm in different languages. The pseudocode syntax you will find in this text has evolved over the years. During our evolution of pseudocode, our students have implemented the same basic pseudocode algorithms in Pascal, C, C++, and Java. In this text, we use C for all of our code implementations.

Abstract Data Types

The second major feature of this text is its use of abstract data types (ADTs). Not every data structure should be implemented as an ADT. However, where

appropriate, we develop C implementations for the student's study and use. Specifically, students will find ADT implementations for Stacks (Chapter 3), Queues (Chapter 4), General Lists (Chapter 5), Binary Search Trees (Chapter 7), AVL Trees (Chapter 8), Heaps (Chapter 9), B-Trees (Chapter 10), and Graphs (Chapter 11).

Visual Approach

As we discuss the various data structures, we first present the general principles using figures and diagrams to help the student visualize the concept. If the data structure is large and complex enough to require several algorithms, we use a structure chart to present a design solution. Once the design and structure are fully understood, we present a pseudocode algorithm, followed as appropriate, by its C implementation.

A brief scan of the book demonstrates our visual approach. There are over 300 figures, 30 tables, 120 algorithms, 170 programs, and numerous code fragments. Although this amount of visual detail tends to create a large book, these materials make it much easier for students to understand and follow the concepts.

Practice Sets

End of chapter materials reenforce what the student has learned. The important topics in the chapter are summarized in bulleted lists. Following the summary are three practice sets.

Exercises

Questions covering the material in the chapter.

Problems

Short assignments that ask the student to develop a pseudocode algorithm or write a short program to be run on a computer. These problems can usually be developed in 2 to 3 hours.

Projects

Longer, major assignments that may take an average student 6 to 9 hours or more to develop.

Glossary

We include a comprehensive glossary. It contains definitions of all key words and other technical terms that we use in the text.

Teaching Tools

The following supplemental materials are available for download on the Course Technology Web site (http://www.course.com). Follow the links to Computer Science and then CS2 Data Structures — C.

Instructor Materials

The following instructor materials are available:

- Complete solutions to exercises and problems.
- Classroom-ready Microsoft PowerPoint presentations for each chapter, including:
  - Objectives
  - Figures
  - Tables
- Source code and any input files needed to run programs within the chapters, including the programs developed as problem solutions.

Student Materials

The following student materials are available:

- Solutions to all odd-numbered exercises.
- Source code and any input files needed to run programs within the chapters.

What's New in This Edition

The basic data structures found in the first edition are carried forward into this edition. You will find, however, that we have significantly changed the organization of the text as well as the presentation of several important concepts. The use of Abstract Data Types has also been extended. The most important changes follow.

1. The book has been organized into four Parts:

   I. Introduction
   This part covers basic concepts and recursion.

   II. Linear Lists
   This part covers stacks, queues, and general linear lists.

   III. Non-Linear Lists
   This part covers introduction to trees, binary search trees, AVL search trees, heaps, multiway trees, and graphs.

   IV. Sorting and Searching
   This part covers sorting and searching.

2. To help students understand and write ADTs using generic code, we added two sections to Chapter 1: Section 1.4, "ADT Implementations," and Section 1.5, "Generic Code for ADTs." Section 1.4 discusses arrays and linked lists as ADT data structure implementations. Section 1.5 discusses the two primary tools of ADTs, pointer to *void* and pointer to function.
3. The ADT concept has been extended to Binary Search Trees (Chapter 7) and Heaps (Chapter 9).

4. The level of the pseudocode has been raised to make it simpler and more conceptual, that is, less code oriented. We have also added end construct statements, such as *end if* and *end loop*.
5. All C programs have been revised to make them C-99 compliant.
6. The efficiency of algorithms has been expanded to include a mathematical discussion of the efficiency of quick sort in Chapter 12.
7. Appendix C, "Integer and Float Libraries," and Appendix D, "Selected C Libraries," have been revised to reflect the C-99 standard.
8. Appendix E, "Mathematical Series And Recursive Relations," has been added to provide a mathematical background for iterative and recursive algorithms.
9. The array implementations of stacks and queues have been removed from the chapters and placed in Appendix F, "Array Implementations of Stacks and Queues."
10. Ancillary Materials, which are found at the Course Technology web site, have been revised to make them easier to use.

Acknowledgments

Course Technology Staff

Our thanks go to our editors and staff at Course Technology, especially Senior Acquisitions Editor, Amy Yarnevich, Product Manager, Alyssa Pratt, and BobbiJo Frasca, Production Editor.

Reviewers

We would especially like to acknowledge the contributions of our reviewers:

First Edition

The following professors contributed to the publication of the first edition.

Naguib Attia, *Johnson C Smith University*
Kevin Croteau, *Francis Marion University*
Barbara Smith, *Conchise College*
William E. Toll, *Taylor University*

Second Edition

The following professors contributed to the publication of the second edition.

Jerry Adams, *Lee University*
Charles W. Bane, *Tarleton State University*
Alfred D. Benoit, *Johnson & Wales University*
Gerald Burgess, *Wilmington College*

Roman Erenshteyn, *Goldey-Beacom College*
Kamal Fernando, *Wilberforce University*
Barbara Guillott, *Louisiana State University*
Stefen Howard, *Mars Hill College*
Keenan D. Jackson, *Wichita State University*
Reza Kamali, *Pennsylvania College of Technology*
Herv Podnar, *Southern Connecticut State University*
Enrico Pontelli, *New Mexico State University*
Iren Valova, *University of Massachusetts Dartmouth*
Colin Ware, *University of New Hampshire*
Jesse Yu, *College of St. Elizabeth*

Richard F. Gilberg
Behrouz A. Forouzan

Part I

Introduction

There are several concepts that are essential to an understanding of this text. We discuss these concepts in the first two chapters. Chapter 1 "Basic Concepts," covers general materials that we use throughout the book. Chapter 2 "Recursion," discusses the concept of recursion. Figure I-1 shows the organization of Part I.

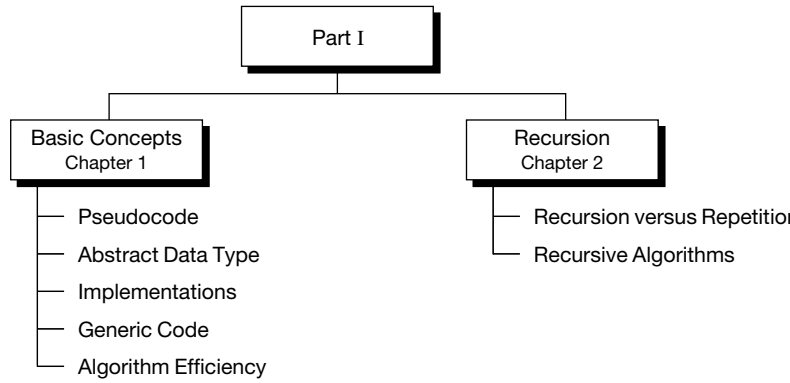


FIGURE I-1 Part I Contents

Chapters Covered

This part includes two chapters.

Chapter 1: Basic Concepts

The first chapter covers concepts that we use throughout the text. You may find that some of them are a review of material from your programming course. The major concepts are outlined below.

Pseudocode

In all of our texts, we use the basic tenet, "Design comes before code." Throughout the text, therefore, we separate algorithm design from the code that implements it in a specific language. Although the underlying language in this book is C, pseudocode allows us to separate the algorithm from the implementation.

Abstract Data Type

An abstract data type (ADT) implements a set of algorithms generically so that they can be applied to any data type or construct. The beauty of an ADT implementation is that the algorithms can handle any data type whether it is a simple integer or a complex record.

ADT Implementations

In general, there are two basic data structures that can be used to implement an abstract data type: arrays and linked lists. We discuss basic linked list concepts in Chapter 1 and expand on them as necessary in subsequent chapters.

Generic Code for ADTs

To implement the ADT concept, we need to use generic code. Each language provides a different set of tools to implement generic code. The C language uses two tools: pointer to *void* and pointer to function.

Algorithm Efficiency

While many authors argue that today's computers and compilers make algorithm efficiency an academic discussion, we believe that an understanding of algorithm efficiency provides the framework for writing better algorithms. Although we discuss the efficiency of specific algorithms when we develop them, in this chapter we discuss the basic concepts and tools for discussing algorithm efficiency.

Chapter 2: Recursion

In Chapter 2 we discuss recursion, a concept that is often skipped in an introductory programming course. We need to understand recursion to discuss data structures because many of the abstract data types are recursive by nature and algorithms that handle them can be better understood using recursion. We use recursive algorithms extensively, especially in Part III, "Non-Linear Lists."

Recursion versus Repetition

The first part of the chapter compares and contrasts recursion and repetition and when each should be used.

Recursive Algorithms

Although recursive algorithms are generally elegant, they can be difficult to understand. In the second part of the chapter, we introduce several algorithms to make the recursive concept clear and to provide a sense of design for creating recursive algorithms.

This page intentionally left blank

Chapter 1
Basic Concepts

This text assumes that the student has a solid foundation in structured programming principles and has written programs of moderate complexity. Although the text uses C for all of its implementation examples, the design and logic of the data structure algorithms are based on pseudocode. This approach creates a language-independent environment for the algorithms.

In this chapter we establish a background for the tools used in the rest of the text, most specifically pseudocode, the abstract data type, algorithm efficiency analysis, and the concepts necessary to create generic code.

1.1 Pseudocode

Although several tools are used to define algorithms, one of the most common is **pseudocode**. Pseudocode is an English-like representation of the algorithm logic. It is part English, part structured code. The English part provides a relaxed syntax that describes *what* must be done without showing unnecessary details such as error messages. The code part consists of an extended version of the basic algorithmic constructs—sequence, selection, and iteration.

> One of the most common tools for defining algorithms is pseudocode, which is part English, part structured code.

In this text we use pseudocode to represent both data structures and code. Data items do not need to be declared. The first time we use a data name in an algorithm, it is automatically declared. Its type is determined by context. The following statement declares a numeric data item named `count` and sets its value to zero.

```
set count to 0
```

The structure of the data, on the other hand, must be declared. We use a simple syntactical statement that begins with a structure name and concludes with the keyword *end* and the name of the structure. Within the structure we list the structural elements by indenting the data items as shown below.

```
node
       data
       link
end node
```

This data definition describes a node in a self-referential list that consists of a nested structure (`data`) and a pointer to the next node (link). An element's type is implied by its name and usage in the algorithm.

As mentioned, pseudocode is used to describe an algorithm. To facilitate a discussion of the algorithm statements, we number them using the hierarchical system shown in Algorithm 1-1. The following sections describe the components of an algorithm. Colored comments provide documentation or clarification when required.

ALGORITHM 1-1 Example of Pseudocode

```
Algorithm sample (pageNumber)
This algorithm reads a file and prints a report.
   Pre    pageNumber passed by reference
   Post   Report Printed
          pageNumber contains number of pages in report
   Return Number of lines printed
 1 loop (not end of file)
   1   read file
   2   if (full page)
       1   increment page number
       2   write page heading
   3   end if
   4   write report line
   5   increment line count
 2 end loop
 3 return line count
end sample
```

Algorithm Header

Each algorithm begins with a header that names it, lists its parameters, and describes any preconditions and postconditions. This information is important because it serves to document the algorithm. Therefore, the header information must be complete enough to communicate to the programmer everything he or she must know to write the algorithm. In Algorithm 1-1 there is only one parameter, the page number.

Purpose, Conditions, and Return

The purpose is a short statement about what the algorithm does. It needs to describe only the general algorithm processing. It should not attempt to describe all of the processing. For example, in Algorithm 1-1 the purpose does not need to state that the file will be opened or how the report will be printed. Similarly, in the search example the purpose does not need to state which of the possible array searches will be used.

The precondition lists any precursor requirements for the parameters. For example, in Algorithm 1-1 the algorithm that calls `sample` must pass the page number by reference. Sometimes there are no preconditions, in which case we still list the precondition with a statement that nothing is required, as shown below.

```
Pre    nothing
```

If there are several input parameters, the precondition should be shown for each. For example, a simple array search algorithm has the following header:

```
Algorithm search  (list, argument, location)
Search array for specific item and return index location.
   Pre    list contains data array to be searched
          argument contains data to be located in list
   Post   location contains matching index
          -or- undetermined if not found
   Return true if found, false if not found
```

In `search` the precondition specifies that the two input parameters, list and argument, must be initialized. If a binary search were being used, the precondition would also state that the array data must be sorted.

The postcondition identifies any action taken and the status of any output parameters. In Algorithm 1-1 the postcondition contains two parts. First, it states that the report has been printed. Second, the reference parameter, `pageNumber`, contains the updated number of pages in the report. In the search algorithm shown above, there is only one postcondition, which may be one of two different values.

If a value is returned, it is identified by a **return condition**. Often there is none, and no return condition is needed. In Algorithm 1-1 we return the number of lines printed. The search algorithm returns true if the argument was found, false if it was not found.

Statement Numbers

Statements are numbered using an abbreviated decimal notation in which only the last of the number sequence is shown on each statement. The expanded number of the statement in Algorithm 1-1 that reads the file is 1.1.

The statement that writes the page heading is 1.2.2. This technique allows us to identify an individual statement while providing statements that are easily read.

Variables

To ensure that the meaning is understood, we use **intelligent data names**—that is, names that describe the meaning of the data. However, it is not necessary to define the variables used in an algorithm, especially when the name indicates the context of the data.

The selection of the name for an algorithm or variable goes a long way toward making the algorithm and its coded implementation more readable. In general, you should follow these rules:

1. Do not use single-character names.
2. Do not use generic names in application programs. Examples of generic names are `count`, `sum`, `total`, `row`, `column`, and `file`. In a program of any size there are several counts, sums, and totals. Rather, add an intelligent qualifier to the generic name so that the reader knows exactly to which piece of data the name refers. For example, `studentCount` and `numberOfStudents` are both better than `count`.
3. Abbreviations are not excluded as intelligent data names. For example, `stuCnt` is a good abbreviation for `student count`, and `numOfStu` is a good abbreviation for `number of students`. Note, however, that `noStu` would not be a good abbreviation for `number of students` because it is too easily read as *no students*.

Statement Constructs

When he first proposed the structured programming model, Edsger Dijkstra stated that any algorithm could be written using only three programming **constructs**: sequence, selection, and loop. Our pseudocode contains only these three basic constructs. The implementation of these constructs relies on the richness of the implementation language. For example, the loop can be implemented as a *while, do…while,* or *for* statement in the C language.

Sequence

A **sequence** is one or more statements that do not alter the execution path *within an algorithm*. Although it is obvious that statements such as assign and add are sequence statements, it is not so obvious that a call to other algorithms is also considered a sequence statement. The reason calls are considered sequential statements lies in the structured programming concept that each algorithm has only one entry and one exit. Furthermore, when an algorithm completes, it returns to the statement immediately after the call that invoked it. Therefore, we can consider an algorithm call a sequence statement. In Algorithm 1-1 statements 1.2.1 and 1.2.2 are sequence statements.

Selection

A **selection statement** evaluates a condition and executes zero or more alternatives. The results of the evaluation determine which alternates are taken.

The typical selection statement is the two-way selection as implemented in an *if* statement. Whereas most languages provide for multiway selections, such as the *switch* in C, we provide none in the pseudocode. The parts of the selection are identified by indentation, as shown in the short pseudocode statement below.

```
1 if (condition)
  1    action1
2 else
  1    action2
3 end if
```

Statement 1.2 in Algorithm 1-1 is an example of a selection statement. The end of the selection is indicated by the *end if* in statement 1.3.

Loop

A **loop** statement iterates a block of code. The loop that we use in our pseudocode closely resembles the *while* loop. It is a pretest loop; that is, the condition is evaluated before the body of the loop is executed. If the condition is true, the body is executed. If the condition is false, the loop terminates.

In Algorithm 1-1 statement 1 is an example of a loop. The end of the loop is indicated by *end loop* in statement 2.

Algorithm Analysis

For selected algorithms, we follow the algorithm with an analysis section that explains some of its salient points. Not every line of code is explained. Rather, the analysis examines only those points that either need to be emphasized or that may require some clarification. The algorithm analysis also often introduces style or efficiency considerations.

Pseudocode Example

As another example of pseudocode, consider the logic required to calculate the deviation from a mean. In this problem we must first read a series of numbers and calculate their average. Then we subtract the mean from each number and print the number and its deviation. At the end of the calculation, we also print the totals and the average.

The obvious solution is to place the data in an array as they are read. Algorithm 1-2 contains the code for this simple problem as it would be implemented in a callable algorithm.

ALGORITHM 1-2 Print Deviation from Mean for Series

```
Algorithm deviation
   Pre    nothing
   Post   average and numbers with their deviation printed
 1 loop (not end of file)
   1   read number into array
   2   add number to total
   3   increment count
 2 end loop
 3 set average to total / count
 4 print average
 5 loop (not end of array)
   1   set devFromAve to array element - average
   2   print array element and devFromAve
 6 end loop
end deviation
```

Algorithm 1-2 Analysis There are two points worth mentioning in Algorithm 1-2. First, there are no parameters. Second, as previously explained, we do not declare variables. A variable's type and purpose should be easily determined by its name and usage.

1.2 The Abstract Data Type

In the history of programming concepts, we started with nonstructured, linear programs, known as **spaghetti code**, in which the logic flow wound through the program like spaghetti on a plate. Next came the concept of **modular programming**, in which programs were organized in functions, each of which still used a linear coding technique. In the 1970s, the basic principles of **structured programming** were formulated by computer scientists such as Edsger Dijkstra and Niklaus Wirth. They are still valid today.

Atomic and Composite Data

Atomic data are data that consist of a single piece of information; that is, they cannot be divided into other meaningful pieces of data. For example, the integer 4562 may be considered a single integer value. Of course, we can decompose it into digits, but the decomposed digits do not have the same characteristics of the original integer; they are four single-digit integers ranging from 0 to 9. In some languages atomic data are known as scalar data because of their numeric properties.

The opposite of atomic data is **composite data**. Composite data can be broken out into subfields that have meaning. As an example of a composite data item, consider your telephone number. A telephone number actually has three different parts. First, there is the area code. Then, what you consider to be your phone number is actually two different data items, a prefix consisting of a three-digit exchange and the number within the exchange,

consisting of four digits. In the past, these prefixes were names such as DAvenport and CYpress.

Data Type

A **data type** consists of two parts: a set of data and the operations that can be performed on the data. Thus we see that the integer type consists of values (whole numbers in some defined range) and operations (add, subtract, multiply, divide, and any other operations appropriate for the data).

Data Type
1. A set of values 2. A set of operations on values

Table 1-1 shows three data types found in all systems.

Type	Values	Operations
integer	-∞, … , -2, -1, 0, 1, 2,…,∞	*, +, -, %, /, ++, --, …
floating point	-∞, … , 0.0, … , ∞	*, +, -, /, …
character	\0, …, 'A', 'B', … , 'a', 'b', … , ~	<, >, …

TABLE 1-1 Three Data Types

Data Structure

A **data structure** is an aggregation of atomic and composite data into a set with defined relationships. In this definition *structure* means a set of rules that holds the data together. In other words, if we take a combination of data and fit them into a structure such that we can define its relating rules, we have made a data structure. Data structures can be nested. We can have a data structure that consists of other data structures. For example, we can define the two structures array and record as shown in Table 1-2.

Array	Record
Homogeneous sequence of data or data types known as elements	Heterogeneous combination of data into a single structure with an identified key
Position association among the elements	No association

TABLE 1-2 Data Structure Examples

Most of the programming languages support several data structures. In addition, modern programming languages allow programmers to create new data structures for an application.

Data Structure

1. A combination of elements in which each is either a data type or another data structure
2. A set of associations or relationships (structure) involving the combined elements

Abstract Data Type

Generally speaking, programmers' capabilities are determined by the tools in their tool kits. These tools are acquired by education and experience. A knowledge of data structures is one of those tools.

When we first started programming, there were no abstract data types. If we wanted to read a file, we wrote the code to read the physical file device. It did not take long to realize that we were writing the same code over and over again. So we created what is known today as an **abstract data type (ADT)**. We wrote the code to read a file and placed it in a library for all programmers to use.

This concept is found in modern languages today. The code to read the keyboard is an ADT. It has a data structure, a character, and a set of operations that can be used to read that data structure. Using the ADT we can not only read characters but we can also convert them into different data structures such as integers and strings.

With an ADT users are not concerned with *how* the task is done but rather with *what* it can do. In other words, the ADT consists of a set of definitions that allow programmers to use the functions while hiding the implementation. This generalization of operations with unspecified implementations is known as abstraction. We abstract the essence of the process and leave the implementation details hidden.

The concept of abstraction means:

1. We know *what* a data type can do.
2. *How* it is done is hidden.

Consider the concept of a list. At least four data structures can support a list. We can use a matrix, a linear list, a tree, or a graph. If we place our list in an ADT, users should not be aware of the structure we use. As long as they can insert and retrieve data, it should make no difference how we store the data. Figure 1-1 shows four logical structures that might be used to hold a list.

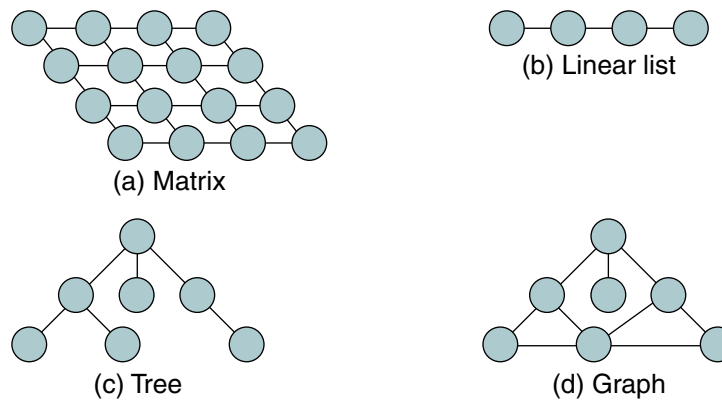


FIGURE 1-1 Some Data Structures

As another example, consider the system analyst who needs to simulate the waiting line of a bank to determine how many tellers are needed to serve customers efficiently. This analysis requires the simulation of a queue. However, queues are not generally available in programming languages. Even if a queue type were available, our analyst would still need some basic queue operations, such as enqueuing (insertion) and dequeuing (deleting), for the simulation.

There are two potential solutions to this problem: (1) we can write a program that simulates the queue our analyst needs (in this case, our solution is good only for the one application at hand) or (2) we can write a queue ADT that can be used to solve any queue problem. If we choose the latter course, our analyst still needs to write a program to simulate the banking application, but doing so is much easier and faster because he or she can concentrate on the application rather than the queue.

We are now ready to define ADT. An abstract data type is a data declaration packaged together with the operations that are meaningful for the data type. In other words, we **encapsulate** the data and the operations on the data, and then we hide them from the user.

Abstract Data Type
1. Declaration of data
2. Declaration of operations
3. Encapsulation of data and operations

We cannot overemphasize the importance of hiding the implementation. The user should not have to know the data structure to use the ADT. Referring to our queue example, the application program should have no knowledge of the data structure. All references to and manipulation of the data in the queue must be handled through defined interfaces to the

structure. Allowing the application program to directly reference the data structure is a common fault in many implementations. This keeps the ADT from being fully portable to other applications.

1.3 Model for an Abstract Data Type

The ADT model is shown in Figure 1-2. The colored area with an irregular outline represents the ADT. Inside the ADT are two different aspects of the model: data structures and functions (public and private). Both are entirely contained in the model and are not within the application program scope. However, the data structures are available to all of the ADT's functions as needed, and a function may call on other functions to accomplish its task. In other words, the data structures and the functions are within scope of each other.

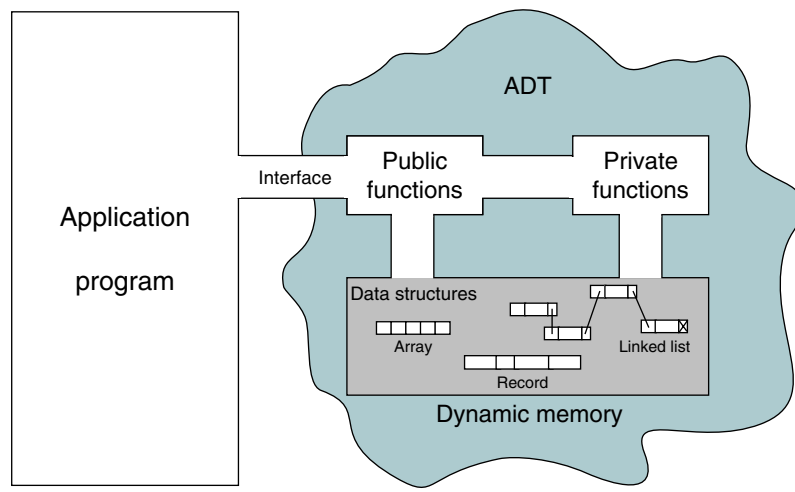


FIGURE 1-2 Abstract Data Type Model

ADT Operations

Data are entered, accessed, modified, and deleted through the external interface drawn as a passageway partially in and partially out of the ADT. Only the public functions are accessible through this interface. For each ADT operation there is an algorithm that performs its specific task. Only the operation name and its parameters are available to the application, and they provide the only interface to the ADT.

ADT Data Structure

When a list is controlled entirely by the program, it is often implemented using simple structures similar to those used in your programming class. Because the abstract data type must hide the implementation from the user, however, all data about the structure must be maintained inside the ADT. Just encapsulating the structure in an ADT is not sufficient. It is also necessary for multiple versions of the structure to be able to coexist. Consequently, we must hide the implementation from the user while being able to store different data.

In this text, we develop ADTs for stacks, queues, lists, binary search trees, AVL trees, B-trees, heaps, and graphs. If you would like a preview, look at the stack ADT in Chapter 3.

1.4 ADT Implementations

There are two basic structures we can use to implement an ADT list: arrays and linked lists.

Array Implementations

In an array, the sequentiality of a list is maintained by the order structure of elements in the array (indexes). Although searching an array for an individual element can be very efficient, addition and deletion of elements are complex and inefficient processes. For this reason arrays are seldom used, especially when the list changes frequently. In addition, array implementations of non-linear lists can become excessively large, especially when there are several successors for each element. Appendix F provides array implementations for two ADTs.

Linked List Implementations

A **linked list** is an ordered collection of data in which each element contains the location of the next element or elements. In a linked list, each element contains two parts: **data** and one or more **links**. The data part holds the application data—the data to be processed. Links are used to chain the data together. They contain pointers that identify the next element or elements in the list.

We can use a linked list to create linear and non-linear structures. In linear linked lists, each element has only zero or one successor. In non-linear linked lists, each element can have zero, one, or more successors.

The major advantage of the linked list over the array is that data are easily inserted and deleted. It is not necessary to shift elements of a linked list to make room for a new element or to delete an element. On the other hand, because the elements are no longer physically sequenced, we are limited to sequential searches:[1] we cannot use a binary search.[2]

1. Sequential and binary searches are discussed in Chapter 13.
2. When we examine trees, you will see several data structures that allow for easy updates and efficient searches.

Figure 1-3(a) shows a linked list implementation of a linear list. The link in each element, except the last, points to its unique successor; the link in the last element contains a null pointer, indicating the end of the list. Figure 1-3(b) shows a linked list implementation of a non-linear list. An element in a non-linear list can have two or more links. Here each element contains two links, each to one successor. Figure 1-3(c) contains an example of an **empty list**, linear or non-linear. We define an empty list as a null list pointer.

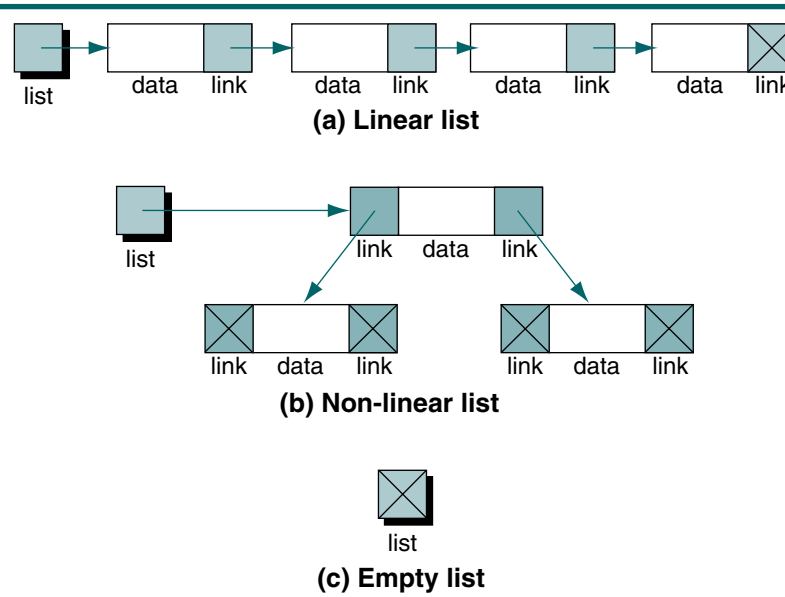


FIGURE 1-3 Linked Lists

In this section, we discuss only the basic concepts for linked lists. We expand on these concepts in future chapters.

Nodes

In linked list implementation, the elements in a list are called nodes. A **node** is a structure that has two parts: the data and one or more links. Figure 1-4 shows two different nodes: one for a linear list and the other for a non-linear list.

The nodes in a linked list are called **self-referential** structures. In a self-referential structure, each instance of the structure contains one or more pointers to other instances of the same structural type. In Figure 1-4, the colored boxes with arrows are the pointers that make the linked list a self-referential structure.

The data part in a node can be a single field, multiple fields, or a structure that contains several fields, but it always acts as a single field. Figure 1-5 shows three designs for a node of a linear list. The upper-left node contains a

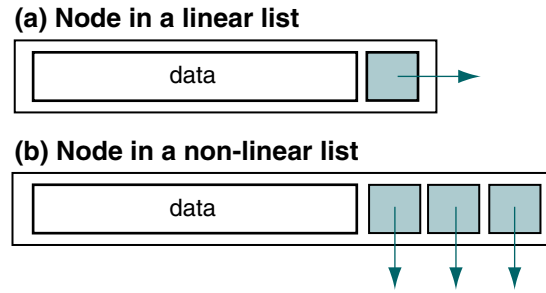


FIGURE 1-4 Nodes

single field, a number, and a link. The upper-right node is more typical. It contains three data fields: a name, an id, and grade points (`grdPts`)—and a link. The third example is the one we recommend. The fields are defined in their own structure, which is then put into the definition of a node structure.

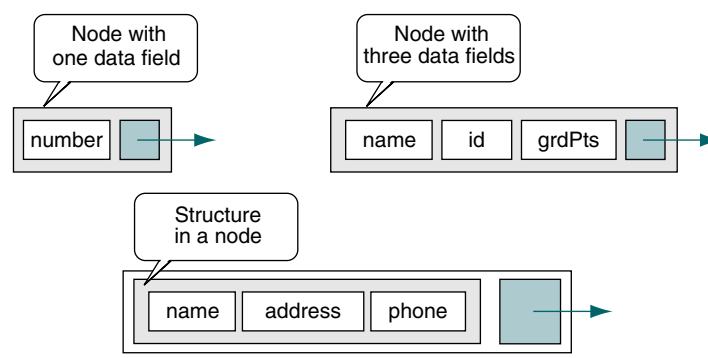


FIGURE 1-5 Linked List Node Structures

Pointers to Linked Lists

A linked list must always have a head pointer. Depending on how we use the list, we may have several other pointers as well. For example, if we are going to search a linked list, we will need an additional pointer to the location where we found the data we were looking for. Furthermore, in many structures, programming is more efficient if there is a pointer to the last node in the list as well as a head pointer.

1.5 Generic Code for ADTs

In data structures we need to create generic code for abstract data types. **Generic code** allows us to write one set of code and apply it to any data type. For

example, we can write generic functions to implement a stack structure. We can then use the generic functions to implement an integer stack, a float stack, a double stack, and so on. Although some high-level languages such as C++ and Java provide special tools to handle generic code, C has limited capability through two features: pointer to *void* and pointer to function.

Pointer to *void*

The first feature is **pointer to *void***. Because C is strongly typed, operations such as assign and compare must use compatible types or be cast to compatible types. The one exception is the pointer to *void*, which can be assigned without a cast. In other words, a pointer to *void* is a generic pointer that can be used to represent any data type during compilation or run time. Figure 1-6 shows the idea of a pointer to *void*. Note that a pointer to *void* is not a null pointer; it is pointing to a generic data type (*void*).

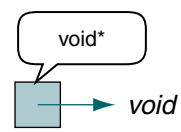


FIGURE 1-6 Pointer to *void*

Example Let us write a simple program to demonstrate the concept. It contains three variables: an integer, a floating-point number, and a *void* pointer. At different times in the program the pointer can be set to the address of the integer value or of the floating-point value. Figure 1-7 shows the situation.

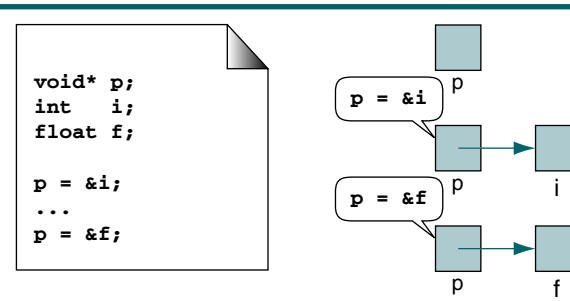


FIGURE 1-7 Pointers for Program 1-1

Program 1-1 uses a pointer to *void* that we can use to print either an integer or a floating-point number.

PROGRAM 1-1 Demonstrate Pointer to *void*

```
 1  /* Demonstrate pointer to void.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main ()
 8  {
 9  // Local Definitions
10     void* p;
11     int   i = 7 ;
12     float f = 23.5;
13
14  // Statements
15     p = &i;
16     printf ("i contains: %d\n", *((int*)p) );
17
18     p = &f;
19     printf ("f contains: %f\n", *((float*)p));
20
21     return 0;
22  }  // main
```

```
Results:
i contains 7
f contains 23.500000
```

Program 1-1 Analysis

The program is trivial, but it demonstrates the point. The pointer `p` is declared as a *void* pointer, but it can accept the address of an integer or floating-point number. However, we must remember a very import point about pointers to *void*: a pointer to *void* cannot be dereferenced unless it is cast. In other words, we cannot use `*p` without casting. That is why we need to cast the pointer in the print function before we use it for printing.

> A pointer to *void* cannot be dereferenced.

Example

As another example, let us look at a system function, *malloc*. This function returns a pointer to *void*. The designers of the *malloc* function needed to dynamically allocate any type of data. However, instead of using several *malloc* functions, each returning a pointer to a specific data type (`int*`, `float*`, `double*`, and so on), they designed a generic function that returns a pointer to *void* (`void*`). While it is not required, we recommend that the returned pointer be cast to the appropriate type. The following shows the use of *malloc* to create a pointer to an integer.

```
intPtr = (int*)malloc (sizeof (int));
```

Example Now let's look at an example that is similar to what we use to implement our ADTs. We need to have a generic function to create a node structure. The structure has two fields: data and link. The link field is a pointer to the node structure. The data field, however, can be any type: integer, floating point, string, or even another structure. To make the function generic so that we can store any type of data in the node, we use a *void* pointer to data stored in dynamic memory. We declare the node structure as shown in Figure 1-8.

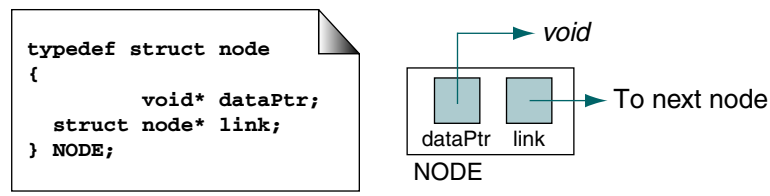


FIGURE 1-8 Pointer to Node

Now let's write the program that calls a function that accepts a pointer to data of any type and creates a node that stores the data pointer and a link pointer. Because we don't know where the link pointer will be pointing, we make it null. The pointer design is shown in Figure 1-9.

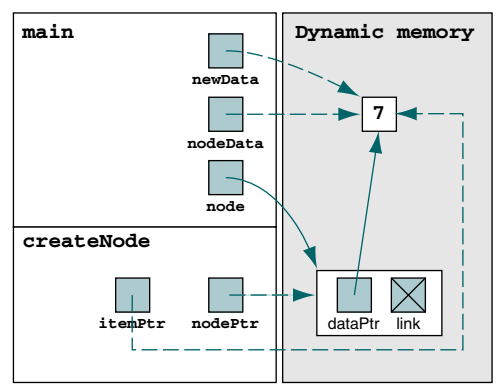


FIGURE 1-9 Pointers for Programs 1-2 and 1-3

Typically, ADTs are stored in their own header files. We begin, therefore, by writing the code for creating the node and placing the code in a header file. This code is shown in Program 1-2.

PROGRAM 1-2 Create Node Header File

```
1  /* Header file for create node structure.
```

continued

PROGRAM 1-2 Create Node Header File *(continued)*

```
 2        Written by:
 3        Date:
 4  */
 5  typedef struct node
 6  {
 7           void* dataPtr;
 8    struct node* link;
 9  } NODE;
10
11  /* ================== createNode ====================
12     Creates a node in dynamic memory and stores data
13     pointer in it.
14        Pre  itemPtr is pointer to data to be stored.
15        Post node created and its address returned.
16  */
17  NODE* createNode (void* itemPtr)
18  {
19     NODE* nodePtr;
20     nodePtr = (NODE*) malloc (sizeof (NODE));
21     nodePtr->dataPtr = itemPtr;
22     nodePtr->link    = NULL;
23     return nodePtr;
24  }  // createNode
```

Now that we've created the data structure and the create node function, we can write Program 1-3 to demonstrate the use of *void* pointers in a node.

PROGRAM 1-3 Demonstrate Node Creation Function

```
 1  /* Demonstrate simple generic node creation function.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include "P1-02.h"                        // Header file
 8
 9  int main (void)
10  {
11  // Local Definitions
12     int*  newData;
13     int*  nodeData;
14     NODE* node;
15
16  // Statements
17     newData  = (int*)malloc (sizeof (int));
18     *newData = 7;
```

continued

PROGRAM 1-3 Demonstrate Node Creation Function *(continued)*

```
19
20      node = createNode (newData);
21
22      nodeData = (int*)node->dataPtr;
23      printf ("Data from node: %d\n", *nodeData);
24      return 0;
25  }   // main
```

```
Results:
Data from node: 7
```

Program 1-3 Analysis

There are several important concepts in this program. First, the data to be stored in the node is represented by a *void* pointer. Because there are usually many instances of these nodes in a program, the data are stored in dynamic memory. The allocation and storage of the data are the responsibility of the programmer. We show these two steps in statements 17 and 18.

The **createNode** function allocates a node structure in dynamic memory, stores the data *void* pointer in the node, and then returns the node's address. In statement 22, we store the *void* pointer from the node into an integer pointer. Because C is strongly typed, this assignment must be cast to an integer pointer. So, while we can store an address in a *void* pointer without knowing its type, the reverse is not true. To use a *void* pointer, even in an assignment, it must be cast.

> Any reference to a *void* pointer must cast the pointer to the correct type.

Example

ADT structures generally contain several instances of a node. To better demonstrate the ADT concept, therefore, let's modify Program 1-3 to contain two different nodes. In this simple example, we point the first node to the second node. The pointer structure for the program is shown in Figure 1-10.

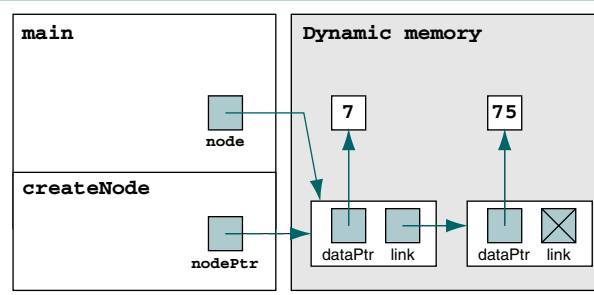


FIGURE 1-10 Structure for Two Linked Nodes

The pointer values in Figure 1-10 represent the settings at the end of Program 1-4.

PROGRAM 1-4 Create List with Two Linked Nodes

```
/* Create a list with two linked nodes.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include "P1-02.h"                          // Header file

int main (void)
{
// Local Definitions
   int*  newData;
   int*  nodeData;
   NODE* node;

// Statements
   newData  = (int*)malloc (sizeof (int));
   *newData = 7;
   node = createNode (newData);

   newData    = (int*)malloc (sizeof (int));
   *newData   = 75;
   node->link = createNode (newData);

   nodeData = (int*)node->dataPtr;
   printf ("Data from node 1: %d\n", *nodeData);

   nodeData = (int*)node->link->dataPtr;
   printf ("Data from node 2: %d\n", *nodeData);
   return 0;
}  // main
```

```
Results:
Data from node 1: 7
Data from node 2: 75
```

Program 1-4 Analysis This program demonstrates an important point. In a generic structure such as shown in the program, the nodes and the data must both be in dynamic memory. When studying the program, follow the code through Figure 1-10.

Pointer to Function

The second tool that is required to create C generic code is pointer to function. In this section we discuss how to use it.

Functions in your program occupy memory. The name of the function is a pointer constant to its first byte of memory. For example, imagine that you have four functions stored in memory: `main`, `fun`, `pun`, and `sun`. This

relationship is shown graphically in Figure 1-11. The name of each function is a pointer to its code in memory.

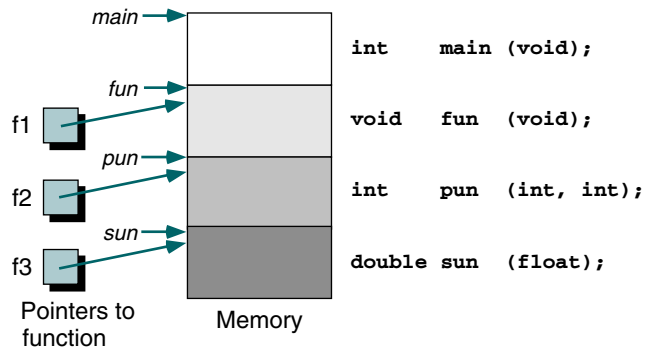


FIGURE 1-11 Functions in Memory

Defining Pointers to Functions

Just as with all other pointer types, we can define pointers to function variables and store the address of `fun`, `pun`, and `sun` in them. To declare a **pointer to function**, we code it as if it were a prototype definition, with the function pointer in parentheses. This format is shown in Figure 1-12. The parentheses are important: without them C interprets the function return type as a pointer.

Using Pointers to Functions

Now that you've seen how to create and use pointers to functions, let's write a generic function that returns the larger of any two pieces of data. The function uses two pointers to *void* as described in the previous section. While our function needs to determine which of the two values represented by the *void* pointers is larger, it cannot directly compare them because it doesn't know what type casts to use with the *void* pointers. Only the application program knows the data types.

The solution is to write simple compare functions for each program that uses our generic function. Then, when we call the generic compare function, we use a pointer to function to pass it the specific compare function that it must use.

Example

As we saw in our discussion of pointers to *void*, we place our generic function, which we call `larger`, in a header file so that it can be easily used. The program interfaces and pointers are shown in Figure 1-13.

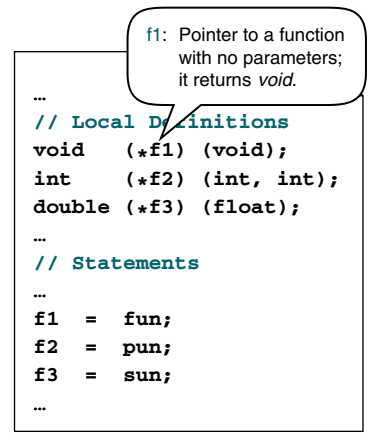


FIGURE 1-12 Pointers to Functions

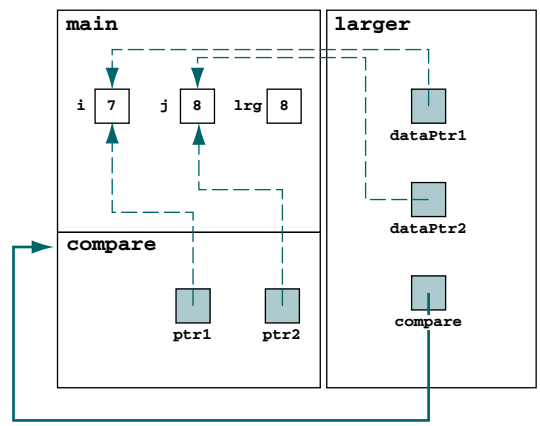


FIGURE 1-13 Design of Larger Function

The code is shown in Program 1-5.

PROGRAM 1-5 Larger Compare Function

```
1  /* Generic function to determine the larger of two
2     values referenced as void pointers.
3        Pre  dataPtr1 and dataPtr2 are pointers to values
4                of an unknown type.
5             ptrToCmpFun is address of a function that
6                knows the data types
7        Post data compared and larger value returned
```

continued

PROGRAM 1-5 Larger Compare Function *(continued)*

```
 8  */
 9  void* larger (void* dataPtr1,    void* dataPtr2,
10                int (*ptrToCmpFun)(void*, void*))
11  {
12     if ((*ptrToCmpFun) (dataPtr1, dataPtr2) > 0)
13           return dataPtr1;
14     else
15           return dataPtr2;
16  }  // larger
```

Program 1-6 contains an example of how to use our generic compare program and pass it a specific compare function.

PROGRAM 1-6 Compare Two Integers

```
 1  /* Demonstrate generic compare functions and pointer to
 2     function.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include "P1-05.h"                          // Header file
 9
10  int   compare (void* ptr1, void* ptr2);
11
12  int main (void)
13  {
14  // Local Definitions
15
16     int i = 7 ;
17     int j = 8 ;
18     int lrg;
19
20  // Statements
21     lrg = (*(int*) larger (&i, &j, compare));
22
23     printf ("Larger value is: %d\n", lrg);
24     return 0;
25  }  // main
26  /* ==================== compare ====================
27     Integer specific compare function.
28        Pre  ptr1 and ptr2 are pointers to integer values
29        Post returns +1 if ptr1 >= ptr2
30             returns -1 if ptr1 <  ptr2
31  */
32  int compare (void* ptr1, void* ptr2)
```

continued

PROGRAM 1-6 Compare Two Integers *(continued)*

```
33  {
34    if (*(int*)ptr1 >=  *(int*)ptr2)
35       return 1;
36    else
37       return -1;
38  }  // compare
```

```
Results:
Larger value is: 8
```

Example Now, let's write a program that compares two floating-point numbers. We can use our larger function, but we need to write a new compare function. We repeat Program 1-6, changing only the compare function and the data-specific statements in *main*. The result is shown in Program 1-7.

PROGRAM 1-7 Compare Two Floating-Point Values

```
 1  /* Demonstrate generic compare functions and pointer to
 2     function.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include "P1-05.h"                          // Header file
 9
10  int   compare (void* ptr1, void* ptr2);
11
12  int main (void)
13  {
14  // Local Definitions
15
16     float f1 = 73.4;
17     float f2 = 81.7;
18     float lrg;
19
20  // Statements
21     lrg = (*(float*) larger (&f1, &f2, compare));
22
23     printf ("Larger value is: %5.1f\n", lrg);
24     return 0;
25  }  // main
26  /* ==================== compare ====================
27     Float specific compare function.
28        Pre  ptr1 and ptr2 are pointers to integer values
29        Post returns +1 if ptr1 >= ptr2
```

continued

PROGRAM 1-7 Compare Two Floating-Point Values *(continued)*

```
30              returns -1 if ptr1 <  ptr2
31  */
32  int compare (void* ptr1, void* ptr2)
33  {
34    if (*(float*)ptr1 >=  *(float*)ptr2)
35       return 1;
36    else
37       return -1;
38  }  // compare
```

```
Results:
Larger value is: 81.7
```

1.6 Algorithm Efficiency

There is seldom a single algorithm for any problem. When comparing two different algorithms that solve the same problem, you often find that one algorithm is an order of magnitude more efficient than the other. In this case, it only makes sense that you be able to recognize and choose the more efficient algorithm.

Although computer scientists have studied algorithms and algorithm efficiency extensively, the field has not been given an official name. Brassard and Bratley coined the term **algorithmics**, which they define as "the systematic study of the fundamental techniques used to design and analyze efficient algorithms."[3] We use the term in this book.

If a function is linear—that is, if it contains no loops or recursions—its efficiency is a function of the number of instructions it contains. In this case, its efficiency depends on the speed of the computer and is generally not a factor in the overall efficiency of a program. On the other hand, functions that use loops or recursion vary widely in efficiency. The study of algorithm efficiency therefore focuses on loops. Our analysis concentrates on loops because recursion can always be converted to a loop.

As we study specific examples, we generally discuss the algorithm's efficiency as a function of the number of elements to be processed. The general format is

$$f(n) = \text{efficiency}$$

The basic concepts are discussed in this section.

3. Gilles Brassard and Paul Bratley, *Algorithmics Theory and Practice* (Englewood Cliffs, N.J.: Prentice Hall, 1988), xiii.

Linear Loops

Let us start with a simple loop. We want to know how many times the body of the loop is repeated in the following code:[4]

```
for (i = 0; i < 1000; i++)
     application code
```

Assuming `i` is an integer, the answer is 1000 times. The number of iterations is directly proportional to the loop factor, 1000. The higher the factor, the higher the number of loops. Because the efficiency is directly proportional to the number of iterations, it is

$$f(n) = n$$

However, the answer is not always as straightforward as it is in the above example. For instance, consider the following loop. How many times is the body repeated in this loop? Here the answer is 500 times. Why?

```
for (i = 0; i < 1000; i += 2)
     application code
```

In this example the number of iterations is half the loop factor. Once again, however, the higher the factor, the higher the number of loops. The efficiency of this loop is proportional to half the factor, which makes it

$$f(n) = n / 2$$

If you were to plot either of these loop examples, you would get a straight line. For that reason they are known as **linear loops**.

Logarithmic Loops

In a linear loop, the loop update either adds or subtracts. In a **logarithmic loop**, the controlling variable is multiplied or divided in each iteration. How many times is the body of the loops repeated in the following program segments?

Multiply Loops

```
for (i = 0; i < 1000; i *= 2)
     application code
```

Divide Loops

```
for (i = 0; i < 1000; i /= 2)
     application code
```

To help you understand this problem, Table 1-3 analyzes the values of *i* for each iteration. As you can see, the number of iterations is 10 in both cases. The reason is that in each iteration the value of `i` doubles for the multiply loop and is cut in half for the divide loop. Thus, the number of iterations

4. For algorithm efficiency analysis, we use C code so that we can clearly see the looping constructs.

Multiply		Divide	
Iteration	**Value of `i`**	**Iteration**	**Value of `i`**
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

TABLE 1-3 Analysis of Multiply and Divide Loops

is a function of the multiplier or divisor, in this case 2. That is, the loop continues while the condition shown below is true.

```
multiply   2^Iterations < 1000
divide     1000 / 2^Iterations >= 1
```

Generalizing the analysis, we can say that the iterations in loops that multiply or divide are determined by the following formula:

$$f(n) = \log n$$

Nested Loops

Loops that contain loops are known as **nested loops**. When we analyze nested loops, we must determine how many iterations each loop completes. The total is then the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

```
Iterations = outer loop iterations x inner loop iterations
```

We now look at three nested loops: linear logarithmic, quadratic, and dependent quadratic.

Linear Logarithmic

The inner loop in the following code is a loop that multiplies. To see the multiply loop, look at the update expression in the inner loop.

```
for (i = 0; i < 10; i++)
     for (j = 0; j < 10; j *= 2)
          application code
```

The number of iterations in the inner loop is therefore $\lceil \log 10 \rceil$. However, because the inner loop is controlled by an outer loop, the above formula must be multiplied by the number of times the outer loop executes, which is 10. This gives us

$$10 \log 10$$

which is generalized as

$$f(n) = n \log n$$

Quadratic

In a **quadratic loop**, the number of times the inner loop executes is the same as the outer loop. Consider the following example.

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
         application code
```

The outer loop (*for i*) is executed 10 times. For each of its iterations, the inner loop (*for j*) is also executed 10 times. The answer, therefore, is 100, which is 10×10, the square of the loops. This formula generalizes to

$$f(n) = n^2$$

Dependent Quadratic

In a **dependent quadratic loop**, the number of iterations of the inner loop depends on the outer loop. Consider the nested loop shown in the following example.

```
for (i = 0; i < 10; i++)
    for (j = 0; j < i; j++)
         application code
```

The outer loop is the same as the previous loop. However, the inner loop depends on the outer loop for one of its factors. It is executed only once the first iteration, twice the second iteration, three times the third iteration, and so forth. The number of iterations in the body of the inner loop is calculated as shown below.

```
1 + 2 + 3 + … + 9 + 10 = 55
```

If we compute the average of this loop, it is 5.5 (55/10), which is the same as the number of iterations (10) plus 1 divided by 2. Mathematically, this calculation is generalized to

$$\frac{(n+1)}{2}$$

Multiplying the inner loop by the number of times the outer loop is executed gives us the following formula for a dependent quadratic loop:

$$f(n) = n\left(\frac{n+1}{2}\right)$$

Big-O Notation

With the speed of computers today, we are not concerned with an exact measurement of an algorithm's efficiency as much as we are with its general order of magnitude. If the analysis of two algorithms shows that one executes 15 iterations while the other executes 25 iterations, they are both so fast that we can't see the difference. On the other hand, if one iterates 15 times and the other 1500 times, we should be concerned.

We have shown that the number of statements executed in the function for n elements of data is a function of the number of elements, expressed as $f(n)$. Although the equation derived for a function may be complex, a dominant factor in the equation usually determines the order of magnitude of the result. Therefore, we don't need to determine the complete measure of efficiency, only the factor that determines the magnitude. This factor is the big-O, as in "on the order of," and is expressed as O(n)—that is, on the order of n.

This simplification of efficiency is known as big-O analysis. For example, if an algorithm is quadratic, we would say its efficiency is

$$O(n^2)$$

or on the order of n squared.

The **big-O notation** can be derived from $f(n)$ using the following steps:

1. In each term, set the coefficient of the term to 1.

2. Keep the largest term in the function and discard the others. Terms are ranked from lowest to highest as shown below.

$$\log n \quad n \quad n\log n \quad n^2 \quad n^3 \ldots n^k \quad 2^n \quad n!$$

For example, to calculate the big-O notation for

$$f(n) = n\frac{(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

we first remove all coefficients. This gives us

$$n^2 + n$$

which after removing the smaller factors gives us

$$n^2$$

which in big-O notation is stated as

$$O(f(n)) = O(n^2)$$

To consider another example, let's look at the polynomial expression

$$f(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$$

We first eliminate all of the coefficients as shown below.

$$f(n) = n^k + n^{k-1} + \ldots + n^2 + n + 1$$

The largest term in this expression is the first one, so we can say that the order of a polynomial expression is

$$O(f(n)) = O(n^k)$$

Standard Measures of Efficiency

Computer scientists have defined seven categories of algorithm efficiency. We list them in Table 1-4 in order of decreasing efficiency and show the first five of them graphically in Figure 1-14.

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable

TABLE 1-4 Measures of Efficiency for n = 10,000

Any measure of efficiency presumes that a sufficiently large sample is being considered. If you are dealing with only 10 elements and the time required is a fraction of a second, there is no meaningful difference between two algorithms. On the other hand, as the number of elements being processed grows, the difference between algorithms can be staggering.

Returning for a moment to the question of why we should be concerned about efficiency, consider the situation in which you can solve a problem in three ways: one is linear, another is linear logarithmic, and a third is quadratic. The magnitude of their efficiency for a problem containing 10,000

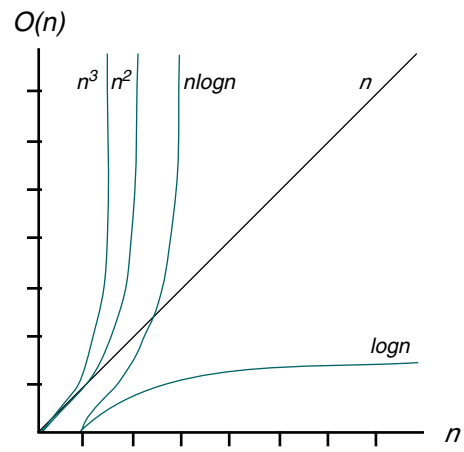


FIGURE 1-14 Plot of Effeciency Measures

elements shows that the linear solution requires a fraction of a second whereas the quadratic solution requires minutes (see Table 1-4).

Looking at the problem from the other end, if we are using a computer that executes a million instructions per second and the loop contains 10 instructions, we spend 0.00001 second for each iteration of the loop. Table 1-4 also contains an estimate of the time needed to solve the problem given different efficiencies.

Big-O Analysis Examples

To demonstrate the concepts we have been discussing, we examine two more algorithms: add and multiply two matrices.

Add Square Matrices

To add two square matrices, we add their corresponding elements; that is, we add the first element of the first matrix to the first element of the second matrix, the second element of the first matrix to the second element of the second matrix, and so forth. Of course, the two matrices must be the same size. This concept is shown in Figure 1-15.

4	2	1	+	6	1	7	=	10	3	8
0	–3	4		3	2	–1		3	–1	3
5	6	2		4	6	2		9	12	4

FIGURE 1-15 Add Matrices

The pseudocode to add two matrices is shown in Algorithm 1-3.

ALGORITHM 1-3 Add Two Matrices

```
Algorithm addMatrix (matrix1, matrix2, size, matrix3)
Add matrix1 to matrix2 and place results in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns or rows in matrix
   Post matrices added--result in matrix3
1  loop (not end of row)
   1   loop (not end of column)
       1   add matrix1 and matrix2 cells
       2   store sum in matrix3
   2   end loop
2  end loop
end addMatrix
```

Algorithm 1-3 Analysis

In this algorithm, we see that for each element in a row, we add all of the elements in a column. This is the classic quadratic loop. The efficiency of the algorithm is therefore $O(\text{size}^2)$ or $O(n^2)$.

Multiply Square Matrices

When two square matrices are multiplied, we must *multiply* each element in a row of the first matrix by its corresponding element in a column of the second matrix. The value in the resulting matrix is then the *sum* of the products. For example, given the matrix in our addition example above, the first element in the resulting matrix—that is, the element at `[0, 0]`—is the sum of the products obtained by multiplying each element in the first *row* (row 0) by its corresponding element in the first *column* (column 0). The value of the element at index location `[0, 1]` is the sum of the products of each element in the first row (again row 0) multiplied by its corresponding element in the second column (column 1). The value of the element at index location `[1, 2]` is the sum of the products of each element in the second *row* multiplied by the corresponding elements in the third *column*. Once again the square matrices must be the same size. Figure 1-16 graphically shows how two matrices are multiplied.

Generalizing this concept, we see that

```
matrix3 [row, col] =
            matrix1[row, 0] x matrix2[0, col]
      +     matrix1[row, 1] x matrix2[1, col]
      +     matrix1[row, 2] x matrix2[2, col]
      ...
      +     matrix1[row, s-1] x matrix2[s-1, col]
where s = size of matrix
```

The pseudocode used for multiplying matrices is provided in Algorithm 1-4.

ALGORITHM 1-4 Multiply Two Matrices

```
Algorithm multiMatrix (matrix1, matrix2, size, matrix3)
Multiply matrix1 by matrix2 and place product in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns and rows in matrix
   Post matrices multiplied--result in matrix3
1  loop (not end of row)
   1   loop (not end of column)
       1   loop (size of row times)
           1   calculate sum of
                   (all row cells) * (all column cells)
           2   store sum in matrix3
```

continued

ALGORITHM 1-4 Multiply Two Matrices *(continued)*

```
   2   end loop
 2  end loop
 3  return
end multiMatrix
```

Algorithm 1-4 Analysis

In this algorithm we see three nested loops. Because each loop starts at the first element, we have a cubic loop. Loops with three nested loops have a big-O efficiency of $O(size^3)$ or $O(n^3)$.

It is also possible to multiply two matrices if the number of rows in the first matrix is the same as the number of columns in the second. We leave the solution to this problem as an exercise (Exercise 21).

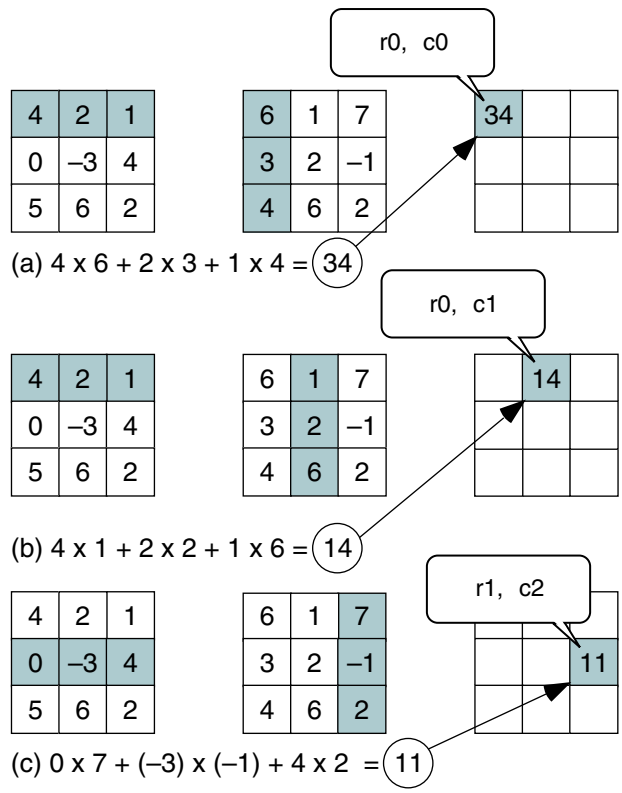


FIGURE 1-16 Multiply Matrices

1.7 Key Terms

abstract data type (ADT)
algorithmics
atomic data
big-O notation
composite data
construct
data
data structure
data type
dependent quadratic loop
empty list
encapsulation
generic code
intelligent data names
linear loop
link
linked list
logarithmic loop
loop
modular programming
nested loop
node
pointer to *void*
pointer to function
pseudocode
quadratic loop
return condition
self-referential
selection statement
sequence
spaghetti code
structured programming

1.8 Summary

- One of the most common tools used to define algorithms is pseudocode.
- Pseudocode is an English-like representation of the code required for an algorithm. It is part English, part structured code.
- Atomic data are data that are single, nondecomposable entities.
- Atomic data types are defined by a set of values and a set of operations that act on the values.
- A data structure is an aggregation of atomic and composite data with a defined relationship.
- An abstract data type (ADT) is a data declaration packaged together with the operations that are meaningful for the data type.
- There are two basic structures used to implement an ADT list: arrays and linked lists.
- In an array, the sequentiality of a list is preserved by the ordered structure of elements. Although searching an array is very efficient, adding and deleting is not.
- Although adding and deleting in a linked list is efficient, searching is not because we must use a sequential search.
- In a linked list, each element contains the location of the next element or elements.

- Abstract data types require generic algorithms, which allow an algorithm to be used with multiple data types.
- The C language has two features that allow the creation of generic code: pointer to *void* and pointer to function.
- A *void* pointer is a generic pointer that can be used to represent any data type.
- A pointer to *void* cannot be dereferenced, which means that nonassignment references to a *void* pointer must be cast to the correct type.
- The name of a function is a pointer constant to the first byte of a function.
- We can use pointer to function as a place holder for the name of a function in a parameter list of a generic function.
- Algorithm efficiency is generally defined as a function of the number of elements being processed and the type of loop being used.
- The efficiency of a logarithmic loop is $f(n) = \log n$.
- The efficiency of a linear loop is $f(n) = n$.
- The efficiency of a linear logarithmic loop is $f(n) = n\ (\log n)$.
- The efficiency of a quadratic loop is $f(n) = n^2$.
- The efficiency of a dependent quadratic loop is $f(n) = n(n + 1)/2$.
- The efficiency of a cubic loop is $f(n) = n^3$.
- The simplification of efficiency is known as big-O notation.
- The seven standard measures of efficiencies are $O(\log n)$, $O(n)$, $O(n(\log n))$, $O(n^2)$, $O(n^k)$, $O(c^n)$, and $O(n!)$.

1.9 Practice Sets

Exercises

1. Structure charts and pseudocode are two different design tools. How do they differ and how are they similar?
2. Using different syntactical constructs, write at least two pseudocode statements to add 1 to a number. For example, any of the following statements could be used to get data from a file:

```
read student file
read student file into student
read (studentFile into student)
```

3. Explain how an algorithm in an application program differs from an algorithm in an abstract data type.
4. Identify the atomic data types for your primary programming language.

5. Identify the composite data types for your primary programming language.
6. Reorder the following efficiencies from smallest to largest:
   a. 2^n
   b. $n!$
   c. n^5
   d. 10,000
   e. $n\log(n)$
7. Reorder the following efficiencies from smallest to largest:
   a. $n\log(n)$
   b. $n + n^2 + n^3$
   c. 24
   d. $n^{0.5}$
8. Determine the big-O notation for the following:
   a. $5n^{5/2} + n^{2/5}$
   b. $6\log(n) + 9n$
   c. $3n^4 + n\log(n)$
   d. $5n^2 + n^{3/2}$
9. Calculate the run-time efficiency of the following program segment:

```
for (i = 1; i <= n; i++)
     printf("%d ", i);
```

10. Calculate the run-time efficiency of the following program segment:

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        for (k = 1; k <= n; k++)
            print ("%d %d %d\n", i, j, k);
```

11. If the algorithm `doIt` has an efficiency factor of $5n$, calculate the run-time efficiency of the following program segment:

```
for (i = 1, i <= n; i++)
     doIt (...)
```

12. If the efficiency of the algorithm `doIt` can be expressed as $O(n) = n^2$, calculate the efficiency of the following program segment:

```
for (i = 1; i <= n;; i++)
    for (j = 1; j < n, j++)
         doIt (...)
```

13. If the efficiency of the algorithm `doIt` can be expressed as $O(n) = n^2$, calculate the efficiency of the following program segment:

```
for (i = 1; i < n; i *= 2)
    doIt (...)
```

14. Given that the efficiency of an algorithm is $5n^2$, if a step in this algorithm takes 1 nanosecond (10^{-9} seconds), how long does it take the algorithm to process an input of size 1000?

15. Given that the efficiency of an algorithm is n^3, if a step in this algorithm takes 1 nanosecond (10^{-9} seconds), how long does it take the algorithm to process an input of size 1000?
16. Given that the efficiency of an algorithm is $5n\log(n)$, if a step in this algorithm takes 1 nanosecond (10^{-9} seconds), how long does it take the algorithm to process an input of size 1000?
17. An algorithm processes a given input of size n. If n is 4096, the run time is 512 milliseconds. If n is 16,384, the run time is 2048 milliseconds. What is the efficiency? What is the big-O notation?
18. An algorithm processes a given input of size n. If n is 4096, the run time is 512 milliseconds. If n is 16,384, the run time is 8192 milliseconds. What is the efficiency? What is the big-O notation?
19. An algorithm processes a given input of size n. If n is 4096, the run time is 512 milliseconds. If n is 16,384, the run time is 1024 milliseconds. What is the efficiency? What is the big-O notation?
20. Three students wrote algorithms for the same problem. They tested the three algorithms with two sets of data as shown below:
    a. Case 1: $n = 10$
        - Run time for student 1: 1
        - Run time for student 2: 1/100
        - Run time for student 3: 1/1000

    b. Case 2: $n = 100$
        - Run time for student 1: 10
        - Run time for student 2: 1
        - Run time for student 3: 1

    What is the efficiency for each algorithm? Which is the best? Which is the worst? What is the minimum number of test cases (n) in which the best algorithm has the best run time?
21. We can multiply two matrices if the number of columns in the first matrix is the same as the number of rows in the second. Write an algorithm that multiplies an m × n matrix by a n × k matrix.
22. Write a compare function (see Program 1-6) to compare two strings.

Problems

23. Write a pseudocode algorithm for dialing a phone number.
24. Write a pseudocode algorithm for giving all employees in a company a cost-of-living wage increase of 3.2%. Assume that the payroll file includes all current employees.

25. Write a language-specific implementation for the pseudocode algorithm in Problem 24.
26. Write a pseudocode definition for a textbook data structure.
27. Write a pseudocode definition for a student data structure.

Projects

28. Your college bookstore has hired you as a summer intern to design a new textbook inventory system. It is to include the following major processes:

    a. Ordering textbooks
    b. Receiving textbooks
    c. Determining retail price
    d. Pricing used textbooks
    e. Determining quantity on hand
    f. Recording textbook sales
    g. Recording textbook returns

    Write the abstract data type algorithm headers for the inventory system. Each header should include name, parameters, purpose, preconditions, postconditions, and return value types. You may add additional algorithms as required by your analysis.

29. Write the pseudocode for an algorithm that converts a numeric score to a letter grade. The grading scale is the typical absolute scale in which 90% or more is an A, 80% to 89% is a B, 70% to 79% is a C, and 60% to 69% is a D. Anything below 60% is an F.
30. Write the pseudocode for an algorithm that receives an integer and then prints the number of digits in the integer and the sum of the digits. For example, given 12,345 it would print that there are 5 digits with a sum of 15.
31. Write the pseudocode for a program that builds a frequency array for data values in the range 1 to 20 and then prints their histogram. The data are to be read from a file. The design for the program is shown in Figure 1-17.

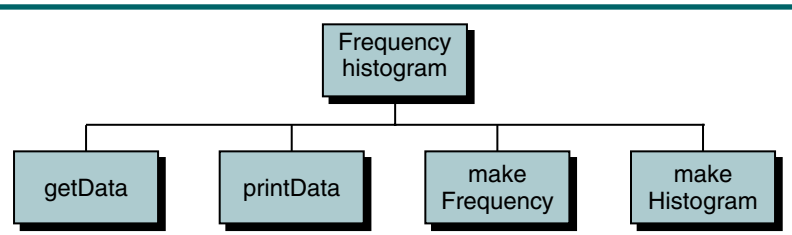


FIGURE 1-17 Design for Frequency Histogram Program

Each of the subalgorithms is described below.

a. The `getData` algorithm reads the file and stores the data in an array.
b. The `printData` algorithm prints the data in the array.
c. The `makeFrequency` algorithm examines the data in the array, one element at a time, and adds 1 to the corresponding element in a frequency array based on the data value.
d. The `makeHistogram` algorithm prints out a vertical histogram using asterisks for each occurrence of an element. For example, if there were five value 1s and eight value 2s in the data, it would print

```
1: *****
2: ********
```

32. Rewrite Program 1-4 to create a list of nodes. Each node consists of two fields. The first field is a pointer to a structure that contains a student id (integer) and a grade-point average (float). The second field is a link. The data are to be read from a text file.

    Then write a program to read a file of at least 10 students and test the function you wrote. You will also need to use the generic compare code in Program 1-6 in your program.

This page intentionally left blank

Chapter 2
Recursion

In general, there are two approaches to writing repetitive algorithms. One uses iteration; the other uses recursion. Recursion is a repetitive process in which an algorithm calls itself. Note, however, that some older languages do not support recursion.

In this chapter we study recursion. We begin by studying a classic recursive case—factorial. Once we explain how recursion works, we develop some principles for developing recursive algorithms and then use them to develop another recursive case study, Fibonacci numbers. We conclude the theory of recursion with a discussion of a classic recursive algorithm, the Towers of Hanoi. In the final section, we develop C implementations for Fibonacci numbers, prefix to postfix conversion, and the Towers of Hanoi.

2.1 Factorial—A Case Study

To begin with a simple example, let's consider the calculation of **factorial**. The factorial of a positive number is the product of the integral values from 1 to the number. This definition is shown in Figure 2-1.

$$\text{Factorial } (n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

FIGURE 2-1 Iterative Factorial Algorithm Definition

Note that this definition is iterative. A repetitive algorithm is defined iteratively whenever the definition involves only the algorithm parameter(s) and not the algorithm itself. We can calculate the value of factorial(4) using Figure 2-1, as follows:

```
factorial(4) = 4 × 3 × 2 × 1 = 24
```

Recursion Defined

A repetitive algorithm uses **recursion** whenever the algorithm appears within the definition itself. For example, the factorial algorithm can be defined recursively as shown in Figure 2-2.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

FIGURE 2-2 Recursive Factorial Algorithm Definition

The decomposition of factorial(3) using Figure 2-2 is shown in Figure 2-3. If you study Figure 2-3 carefully, you will note that the recursive solution for a problem involves a two-way journey: first we decompose the problem from the top to the bottom, then we solve it from the bottom to the top.

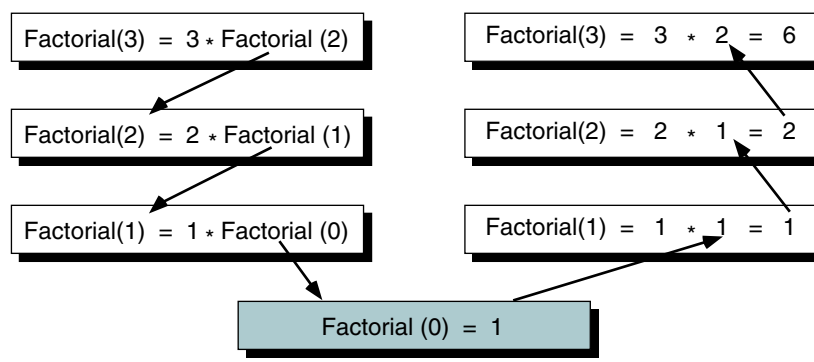


FIGURE 2-3 Factorial (3) Recursively

Judging by this example, the recursive calculation appears to be much longer and more difficult. So why would we want to use the recursive method? Although the recursive calculation looks more difficult when using

paper and pencil, it is often a much easier and more elegant solution when we use computers. Also, it offers a conceptual simplicity to the creator and the reader.

Recursion is a repetitive process in which an algorithm calls itself.

Iterative Solution

Let's write an algorithm to solve the factorial problem iteratively. This solution usually involves using a loop such as the one shown in Algorithm 2-1.

ALGORITHM 2-1 Iterative Factorial Algorithm

```
Algorithm iterativeFactorial (n)
Calculates the factorial of a number using a loop.
   Pre  n  is the number to be raised factorially
   Post n! is returned
 1 set i to 1
 2 set factN to 1
 3 loop (i <= n)
   1  set factN to factN * i
   2  increment i
 4 end loop
 5 return factN
end iterativeFactorial
```

Recursive Solution

Now let's write the same algorithm recursively. The recursive solution does not need a loop; recursion is itself repetition. In the recursive version, we let the factorial algorithm call itself, each time with a different set of parameters. The algorithm for recursive factorial is shown in Algorithm 2-2.

ALGORITHM 2-2 Recursive Factorial

```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
   Pre    n  is the number being raised factorially
   Post   n! is returned
 1 if (n equals 0)
   1  return 1
 2 else
   1  return (n * recursiveFactorial (n - 1))
 3 end if
end recursiveFactorial
```

Algorithm 2-2 Analysis

If you compare the iterative and recursive versions of factorial, you should be immediately struck by how much simpler the code is in the recursive version. First, there is no loop. The recursive version consists of a simple selection statement that returns either the value 1 or the product of two values, one of which is a call to factorial itself.

Figure 2-4 traces the recursion and the parameters for each individual call.

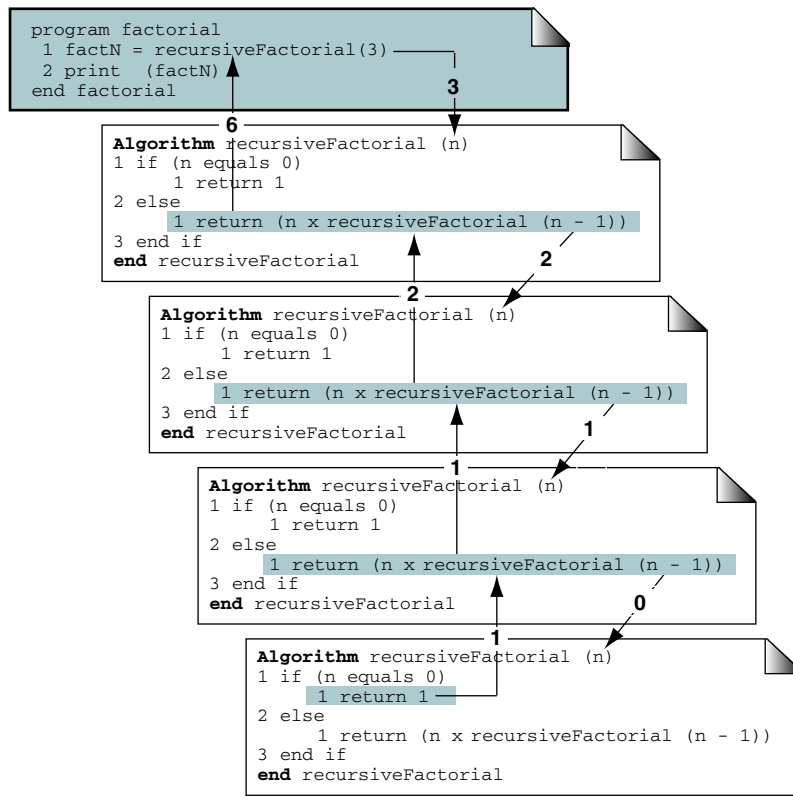


FIGURE 2-4 Calling a Recursive Algorithm

2.2 Designing Recursive Algorithms

Now that we have seen how recursion works, let's turn our attention to the steps for designing a recursive algorithm. We first look at the basic design methodology, then we discuss the limitations of recursion, and finally we design and implement another recursive algorithm.

The Design Methodology

If we were to examine all hypothetically possible recursive algorithms, we would see that they all have two elements: each call either *solves* one part of

the problem or it reduces *the size* of the problem. In Algorithm 2-2 statement 1.1 solves a small piece of the problem—factorial(0) is 1. Statement 2.1, on the other hand, reduces the size of the problem by recursively calling factorial with *n* - 1. Once the solution to factorial (n - 1) is known, statement 2.1 provides a part of the solution to the general problem by returning a value to the calling algorithm.

As we see in statement 2.1, the general part of the solution is the recursive call: statement 2.1 calls itself to solve the problem. We also see this in Figure 2-4. At each recursive call, the size of the problem is reduced, from the factorial of 3, to 2, to 1, and finally to factorial 0.

The statement that "solves" the problem is known as the **base case**. *Every recursive algorithm must have a base case.* The rest of the algorithm is known as the **general case**. In our factorial example, the base case is factorial (0); the general case is n × factorial (n - 1). The general case contains the logic needed to reduce the size of the problem.

Every recursive call must either solve a part of the problem or reduce the size of the problem.

In the factorial problem, once the base case has been reached, the solution begins. We now know one part of the answer and can return that part to the next, more general statement. Thus, in Algorithm 2-2, we know that factorial(0) is 1, and we return that value. This allows us to solve the next general case

```
factorial(1) ⇨ 1 × factorial(0) ⇨ 1 × 1 ⇨ 1
```

We can now return the value of `factorial(1)` to the more general case, factorial(2), which we know to be

```
factorial(2) ⇨ 2 × factorial(1) ⇨ 2 × 1 ⇨ 2
```

As we solve each general case in turn, we are able to solve the next-higher general case until we finally solve the most general case, the original problem.

Returning to the purpose of this section, we are now ready to state the rules for designing a recursive algorithm:

1. First, determine the base case.
2. Then determine the general case.
3. Combine the base case and the general cases into an algorithm.

In combining the base and the general case into an algorithm, we must pay careful attention to the logic. Each call must reduce the size of the problem and move it toward the base case. The base case, when reached, must terminate without a call to the recursive algorithm; that is, it must execute a return.

Limitations of Recursion

We have introduced only a brief explanation of recursion in this section. Recursion works best when the algorithm uses a data structure that naturally supports recursion. For example, in Chapter 6 we will study trees. Trees are a naturally recursive structure and recursion works well with them.

In other cases the algorithm is naturally suited to recursion. For example, the binary search algorithm (see Chapter 13) lends itself to a natural recursive algorithm, as does the Towers of Hanoi algorithm, which we discuss later in this chapter. On the other hand, not all looping algorithms can or should be implemented with recursion, as we discuss below.

Recursive solutions may involve extensive overhead (both time and memory) because they use calls. Each call takes time to execute. A recursive algorithm therefore generally runs more slowly than its nonrecursive implementation.[1]

You should not use recursion if the answer to any of the following questions is no:

1. Is the algorithm or data structure naturally suited to recursion?
2. Is the recursive solution shorter and more understandable?
3. Does the recursive solution run within acceptable time and space limits?

For reasons we will explain in Chapter 3, each time we make a call we use up some of our memory allocation. If the recursion is deep—that is, if there are many recursive calls—we may run out of memory.

Because of the time and memory overhead, algorithms such as factorial are better developed iteratively if large numbers are involved. As a general rule, recursive algorithms should be used only when their efficiency is logarithmic.

Design Implementation—Reverse Keyboard Input

Having studied the design methodology for recursive algorithms and their limitations, we are now ready to put the concepts into practice. Assume that we are reading data from the keyboard and need to print the data in reverse. The easiest way to print the list in reverse is to write a recursive algorithm.

It should be obvious that to print the list in reverse, we must first read all of the data. The base case, therefore, is that we have read the last piece of data. Similarly, the general case is to read the next piece of data. The question is, when do we print? If we print before we read all of the data, we print the list in sequence. If we print the list after we read the last piece of data—that is, if we print it as we back out of the recursion—we print it in reverse sequence. The code is shown in Algorithm 2-3.

1. Most of today's compilers optimize code when possible. When the recursion occurs at the end of a function (known as tail recursion), an optimized compiler turns the recursive code into a simple loop, thus eliminating the function call inefficiency.

ALGORITHM 2-3 Print Reverse

```
Algorithm printReverse (data)
Print keyboard data in reverse.
   Pre  nothing
   Post data printed in reverse
 1 if (end of input)
   1   return
 2 end if
 3 read data
 4 printReverse (data)
Have reached end of input: print nodes
 5 print data
 6 return
end printReverse
```

Algorithm 2-3 Analysis

As you study Algorithm 2-3, remember that statement 5 cannot be executed until we reach the end of the input. It is not executed immediately after statement 4 because statement 4 is a recursive call. We get to statement 5 only after we return from statement 1.1 or after we return from the end statement at the end of the algorithm. Figure 2-5 traces the execution of Algorithm 2-3.

This algorithm also demonstrates the use of local variables in recursion. Although there is only one variable, **data**, in the pseudocode, when we implement the algorithm in a recursive language, the computer creates a separate set of local variables for each call. These variables are kept in memory until we return from the call that created them, at which time they are recycled.

Now that we've designed the algorithm, we need to analyze it to determine whether it is really a good solution; that is, is the recursive algorithm a good candidate for recursion? To analyze this algorithm, we turn to the three questions we developed in the preceeding "Limitations of Recursion" section.

1. Is the algorithm or data structure naturally suited to recursion? A list, such as data read from the keyboard, is not a naturally recursive structure. Furthermore, the algorithm is not naturally suited to recursion because it is not a logarithmic algorithm.
2. Is the recursive solution shorter and more understandable? The answer to this question is yes.
3. Does the recursive solution run within acceptable time and space limits? The number of iterations in the traversal of a list can become quite large because the algorithm has a linear efficiency—that is, it is $O(n)$.

We thus see that the answer to two of the three questions is no. Therefore, although we can successfully write the algorithm recursively, we should not. It is not a good candidate for recursion and is better implemented as an iterative loop.

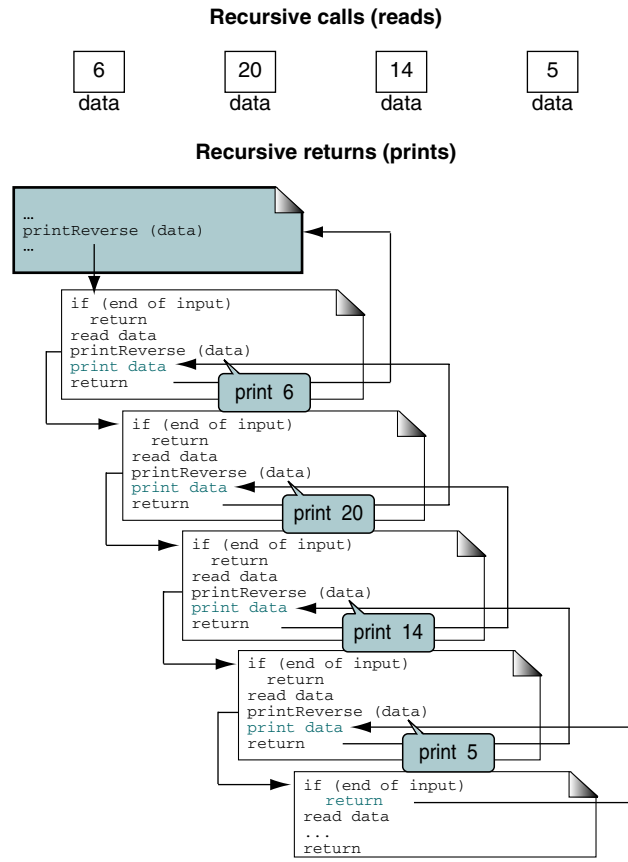


FIGURE 2-5 Print Keyboard Input in Reverse

2.3 Recursive Examples

In this section we discuss four recursive examples: greatest common divisor, Fibonacci numbers, prefix to postfix conversion, and the Towers of Hanoi. For each example, we start with a design and then implement it in C.

Greatest Common Divisor

A common mathematics function is to determine the **greatest common divisor (GCD)** for two numbers. For example, the greatest common divisor for 10 and 25 is 5.

GCD Design

We use the Euclidean algorithm to determine the greatest common divisor between two nonnegative integers. Given two integers, a and b, the greatest common divisor is recursively found using the formula in Figure 2-6.

$$\gcd(a, b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

FIGURE 2-6 Greatest Common Divisor Recursive Definition

The pseudocode design for the Euclidean algorithm is shown in Algorithm 2-4.

ALGORITHM 2-4 Euclidean Algorithm for Greatest Common Divisor

```
Algorithm gcd (a, b)
Calculates greatest common divisor using the Euclidean algo-
rithm.
   Pre  a and b are positive integers greater than 0
   Post greatest common divisor returned
1  if (b equals 0)
   1   return a
2  end if
3  if (a equals 0)
   2   return b
4  end if
5  return gcd (b, a mod b)
end gcd
```

GCD C Implementation

Our implementation of the GCD algorithm uses a driver that asks the user for two numbers. After editing the numbers, it calls a recursive implementation that returns the greatest common divisor.

PROGRAM 2-1 GCD Driver

```
1  /* This program determines the greatest common divisor
2     of two numbers.
3        Written by:
4        Date:
5  */
6  #include <stdio.h>
```

continued

PROGRAM 2-1 GCD Driver *(continued)*

```
 7  #include <ctype.h>
 8
 9  // Prototype Statements
10  int gcd (int a, int b);
11
12  int main (void)
13  {
14  // Local Declarations
15     int  gcdResult;
16
17  // Statements
18     printf("Test GCD Algorithm\n");
19
20     gcdResult = gcd (10, 25);
21     printf("GCD of 10 & 25 is %d", gcdResult);
22     printf("\nEnd of Test\n");
23     return 0;
24  }  // main
25  /* ================ gcd ================
26     Calculates greatest common divisor using the
27     Euclidean algorithm.
28        Pre  a and b are positive integers greater than 0
29        Post greatest common divisor returned
30  */
31  int gcd (int a, int b)
32  {
33     // Statements
34     if (b == 0)
35        return a;
36     if (a == 0)
37        return b;
38     return gcd (b, a % b);
39  }  // gcd
```

```
Results:
Test GCD Algorithm
GCD of 10 & 25 is 5
End of Test
```

Fibonacci Numbers

Fibonacci numbers are named after Leonardo Fibonacci, an Italian mathematician who lived in the early thirteenth century. In this series each number is the sum of the previous two numbers. The first few numbers in the Fibonacci series are

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

Design

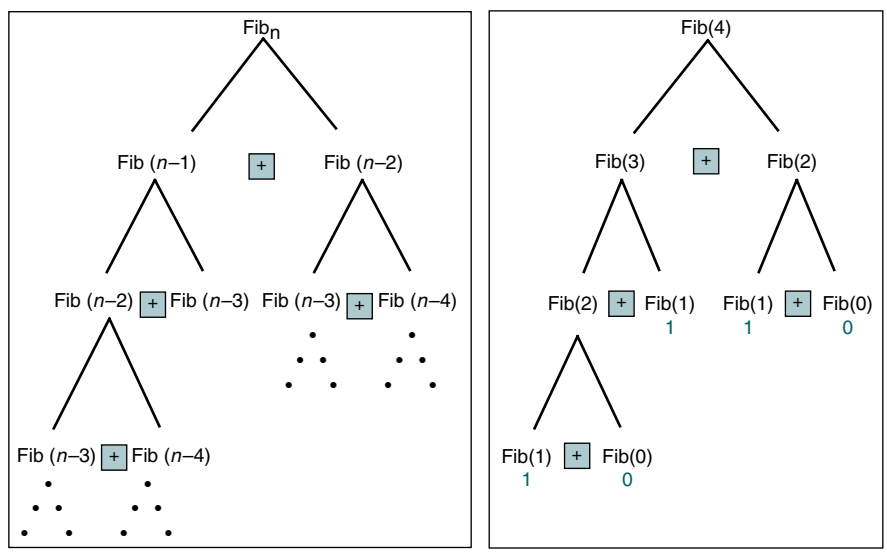
To start a Fibonacci series, we need to know the first two numbers. As you can see from the previous example, they are 0 and 1. Because we are discussing recursion, you should recognize these two numbers as the base cases.

We can generalize the definition of the Fibonacci series as shown in Figure 2-7.

$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) & \text{otherwise} \end{cases}$$

FIGURE 2-7 Fibonacci Numbers Recursive Definition

The generalization of *Fibonacci(4)* is shown in Figure 2-8. Figure 2-8(a) shows the components of *Fibonacci(4)* using a general notation. Figure 2-8(b) shows the components as they would be called to generate the numbers in the series.



(a) Fib(*n*) (b) Fib(4)

FIGURE 2-8 Fibonacci Numbers

Fibonacci C Implementation

In our Fibonacci program, we begin by asking the user how many numbers are needed. We then use a *for* loop that calls the Fibonacci function the prescribed number of times, printing the next number in the series in each loop.

In the C implementation of a Fibonacci series (Program 2-2), it is interesting to note that the bulk of the code in the algorithm is the mainline code to read the number of numbers to be generated and to print the results. There are actually only three statements in the recursive algorithm.

PROGRAM 2-2 Recursive Fibonacci Series

```
/* This program prints out a Fibonacci series.
      Written by:
      Date:
*/
#include <stdio.h>

// Prototype Statements
   long fib (long num);

int main (void)
{
// Local Declarations
   int seriesSize = 10;

// Statements
   printf("Print a Fibonacci series.\n");

   for (int looper = 0; looper < seriesSize; looper++)
       {
        if (looper % 5)
           printf(", %8ld", fib(looper));
        else
           printf("\n%8ld", fib(looper));
       } // for
   printf("\n");
   return 0;
}  // main

/* ================ fib =================
   Calculates the nth Fibonacci number
      Pre  num identifies Fibonacci number
      Post returns nth Fibonacci number
*/
long fib (long num)
{
// Statements
   if (num == 0 || num == 1)
```

continued

PROGRAM 2-2 Recursive Fibonacci Series *(continued)*

```
38          // Base Case
39          return num;
40       return (fib (num - 1) + fib (num - 2));
41    }  // fib
```

```
Results:
Print a Fibonacci series.

       0,        1,        1,        2,        3
       5,        8,       13,       21,       34
```

Program 2-2 Analysis If you have difficulty following the code, refer to Figure 2-8 and trace the first four Fibonacci numbers. The first 10 numbers are displayed at the end of the algorithm. Unless you have a very fast computer, we recommend that you don't try to calculate more than 30 to 35 numbers.

How many calls does it take to determine *Fibonacci(5)*? The answer is 15. As you can see from Table 2-1, the number of calls goes up quickly as we increase the size of the Fibonacci number we are calculating.

fib(*n*)	Calls	fib(n)	Calls
1	1	11	287
2	3	12	465
3	5	13	753
4	9	14	1219
5	15	15	1973
6	25	20	21,891
7	41	25	242,785
8	67	30	2,692,573
9	109	35	29,860,703
10	177	40	331,160,281

TABLE 2-1 Fibonacci Calls

Table 2-1 leads us to the obvious conclusion that a recursive solution to calculate Fibonacci numbers is not efficient for large numbers.

Prefix to Postfix Conversion

An arithmetic expression can be represented in three different formats: infix, postfix, and prefix. In an **infix** notation, the operator comes between the two operands, the basic format of the algebraic notation we learned in grammar

school. In **postfix** notation,[2] the operator comes after its two operands, and in **prefix** notation it comes before the two operands. These formats are shown below.

```
Prefix:   + A B
Infix:    A + B
Postfix:  A B +
```

Although some high-level languages use the infix and postfix notations, these expressions cannot be directly evaluated. Rather, they must be analyzed to determine the order in which the expressions are to be evaluated. One method to analyze them is to use recursion.

In *prefix* notation the operator comes *before* the operands.

In *infix* notation the operator comes *between* the operands.

In *postfix* notation the operator comes *after* the operands.

Design

In this section we use recursion to convert prefix expressions to the postfix format.[3]

Before looking at the algorithms, we review the basic concept. Given the following prefix expression

```
*AB
```

we convert it by moving the operator (`*`) after the operands (`A` and `B`). The postfix format of the expression is shown below.

```
AB*
```

To keep the algorithm as simple as possible, we assume that each operand is only one character and that there are only four operators: add (+), subtract (–), multiply (`*`), and divide (`/`).

As stated, to convert a prefix expression we must find its operator and move it after the two operands. By definition, the operator is always the first character in the prefix string. Following the operator are its two operands. As

2. Postfix notation is also known as reverse Polish notation (RPN) in honor of its originator, the Polish logician Jan Lukasiewicz.
3. We will return to this problem when we study stacks in Chapter 3.

the expression grows in complexity, however, we can have multiple operators and operands in an expression. Consider the following prefix expression, which we will use for our discussions.

```
-+*ABC/EF
```

This expression consists of several binary expressions combined into one complex expression. To parse it we begin at the left and work right until we isolate one binary expression, in this case `*AB`. Once we have a simple prefix expression, we can convert it to postfix and put it in the output. Figure 2-9 contains a decomposition of the complete process. The step that isolates `*AB` is on the left.

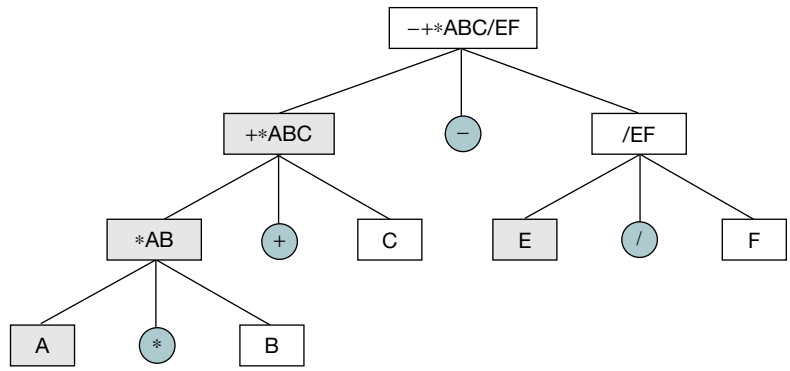


FIGURE 2-9 Decomposition of `-+*ABC/EF`

The first step in designing a recursive algorithm is determining the base case. The question, therefore, is "What is the base case?" Obviously, it is not finding an operator. Not so obviously, the base case turns out to be finding an operand—in our example, any of the alphabetic characters.

Because the operand can contain another expression, the general case is to find the operator and the left and right operands in the binary expression. We then concatenate them into one postfix expression.

To find the left and right operands in the prefix expression, we need a second algorithm that determines the length of a prefix expression. The length of an expression is the length of the first subexpression plus the length of the second expression plus 1 for the operator. We also use recursion for this algorithm. Because each operand is one character long, our base case is finding one operand.

We are now ready to design the algorithms. Algorithm 2-5 contains the pseudocode to build the postfix expression.

ALGORITHM 2-5 Convert Prefix Expression to Postfix

```
Algorithm preToPostFix (preFixIn, postFix)
Convert a preFix string to a postFix string.
   Pre  preFix is a valid preFixIn expression
        postFix is reference for converted expression
   Post postFix contains converted expression
 1 if (length of preFixIn is 1)
      Base case: one character string is an operand
   1  set postFix to preFixIn
   2  return
 2 end if
   If not an operand, must be an operator
 3 set operator to first character of preFixIn
   Find first expression
 4 set lengthOfExpr to findExprLen (preFixIn less first char)
 5 set temp to substring(preFixIn[2, lengthOfExpr])
 6 preToPostFix (temp, postFix1)
   Find second postFix expression
 7 set temp to prefixIn[lengthOfExpr + 1, end of string]
 8 preToPostFix (temp, postFix2)
   Concatenate postfix expressions and operator
 9 set postFix to postFix1 + postFix2 + operator
10 return
end preToPostFix
```

Algorithm 2-5 Analysis

The algorithm is rather straightforward. The most difficult part is finding the prefix expression. We find it by determining its length and then storing it in a temporary string. The notation in statement 5 indicates that the substring starting at the first location and through the second location is to be assigned to **temp**. Once we have identified a prefix expression, we recursively call Algorithm 2-5 to convert it to postfix.

Now let's look at the algorithm that determines the length of the prefix expression. As stated earlier, this algorithm needs to return the length of the expression. Once again the base case is finding an operand. From the definition of the problem, we know that each operand is one character long. The minimum length of a prefix expression is therefore three: one for the operator and one for each operand. As we recursively decompose the expression, we continue to add the size of its combined expressions until we have the total length.

The code is shown in Algorithm 2-6.

ALGORITHM 2-6 Find Length of Prefix Expression

```
Algorithm findExprLen (exprIn)
Recursively determine the length of a prefix expression.
   Pre  exprIn is a valid prefix expression
   Post length of expression returned
1  if (first character is operator)
   General Case: First character is operator
   Find length of first prefix expression
   1  set len1 to findExprLen (exprIn + 1)
   2  set len2 to findExprLen (exprIn + 1 + len2)
2  else
   Base case--first char is operand
   1  set len1 and len2 to 0
3  end if
4  return len1 + len2 + 1
end findExprLen
```

Algorithm 2-6 Analysis

We examine the first character of a string containing a prefix expression or a substring. If it is an operator, we recursively add the length of its two operands. The algorithm returns either 1 (the base case) or the length of an expression in the prefix expression (+*ABC in Figure 2-10).

Prefix to Postfix C Implementation

With the design complete, we are ready to write Program 2-3. The mainline is simply a test driver that calls `preToPostFix` and prints the results. The two functions parallel the pseudocode developed in the previous section. The results are shown at the end.

PROGRAM 2-3 Prefix to Postfix

```
 1  /* Convert prefix to postfix expression.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <string.h>
 7
 8  #define OPERATORS "+-*/"
 9
10  // Prototype Declarations
11  void preToPostFix  (char* preFixIn, char* exprOut);
12  int  findExprLen   (char* exprIn);
13
14  int main (void)
15  {
```

continued

PROGRAM 2-3 Prefix to Postfix *(continued)*

```
16  // Local Definitions
17     char  preFixExpr[256]  = "-+*ABC/EF";
18     char  postFixExpr[256] = "";
19
20  // Statements
21     printf("Begin prefix to postfix conversion\n\n");
22
23     preToPostFix (preFixExpr, postFixExpr);
24     printf("Prefix expr:  %-s\n", preFixExpr);
25     printf("Postfix expr: %-s\n", postFixExpr);
26
27     printf("\nEnd prefix to postfix conversion\n");
28     return 0;
29  }  // main
30
31  /* ================== preToPostFix ==================
32     Convert prefix expression to postfix format.
33        Pre  preFixIn is string prefix expression
34             expression can contain no errors/spaces
35             postFix is string variable for postfix
36        Post expression has been converted
37  */
38  void preToPostFix (char* preFixIn, char* postFix)
39  {
40  // Local Definitions
41     char  operator [2];
42     char  postFix1[256];
43     char  postFix2[256];
44     char  temp    [256];
45     int   lenPreFix;
46
47  // Statements
48     if (strlen(preFixIn) == 1)
49        {
50         *postFix       = *preFixIn;
51         *(postFix + 1) = '\0';
52         return;
53        } // if only operand
54
55     *operator       = *preFixIn;
56     *(operator + 1) = '\0';
57
58     // Find first expression
59     lenPreFix = findExprLen (preFixIn + 1);
60     strncpy (temp, preFixIn + 1, lenPreFix);
61     *(temp + lenPreFix) = '\0';
62     preToPostFix (temp, postFix1);
```

continued

PROGRAM 2-3 Prefix to Postfix *(continued)*

```
63
64    // Find second expression
65    strcpy (temp, preFixIn + 1 + lenPreFix);
66    preToPostFix (temp, postFix2);
67
68    // Concatenate to postFix
69    strcpy (postFix, postFix1);
70    strcat (postFix, postFix2);
71    strcat (postFix, operator);
72
73    return;
74 }  // preToPostFix
75
76 /* ================== findExprLen ====================
77    Determine size of first substring in an expression.
78       Pre  exprIn contains prefix expression
79       Post size of expression is returned
80 */
81 int findExprLen (char* exprIn)
82 {
83 // Local Definitions
84    int  len1;
85    int  len2;
86
87 // Statements
88    if (strcspn (exprIn, OPERATORS) == 0)
89          // General Case: First character is operator
90          // Find length of first expression
91       {
92          len1 = findExprLen(exprIn + 1);
93
94          // Find length of second expression
95          len2 = findExprLen(exprIn + 1 + len1);
96       } // if
97    else
98          // Base case--first char is operand
99          len1 = len2 = 0;
100   return len1 + len2 + 1;
101 }  // findExprLen
```

```
Results:
Begin prefix to postfix conversion

Prefix expr:  -+*ABC/EF
Postfix expr: AB*C+EF/-

End prefix to postfix conversion
```

Program 2-3 Analysis

The program design is quite complex and requires careful study. The recursive call sequences are shown in Figure 2-10. It determines the first expression on the left (+*ABC) in Figure 2-9. Each call shows the part of the expression that is passed in the parameter list. Note that the minus operator in the original infix expression is removed before the call.

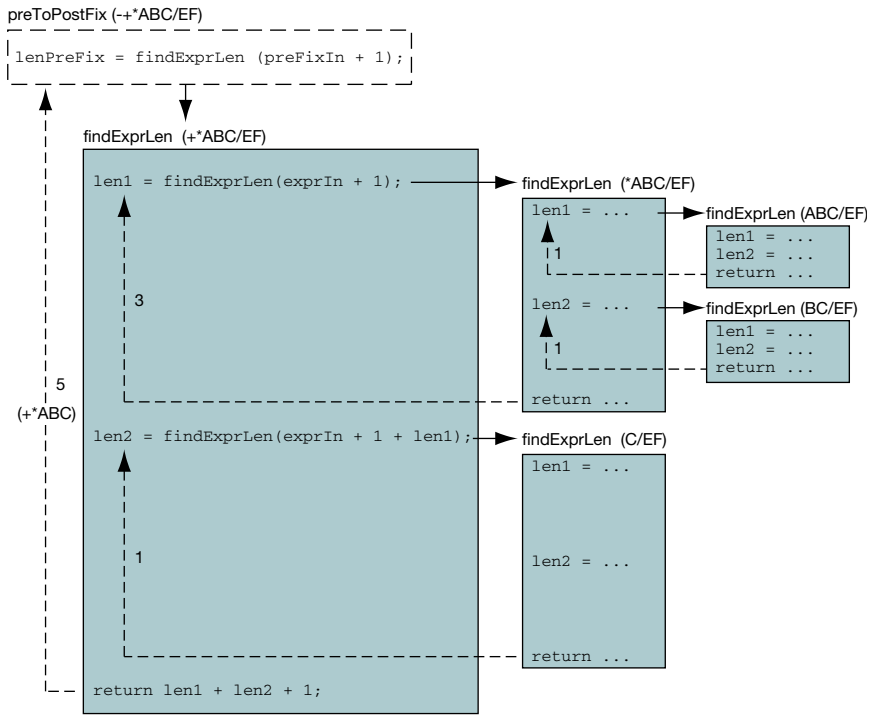


FIGURE 2-10 Recursive Call for Find Expression Length

Because the first character is an operator (**+**), we call recursively after stripping off the plus operator, leaving **`*ABC/EF`**. Once again the first character is an operator (*****), so we call recursively again, this time with **`ABC/EF`**. Because the first character is now an operand (**A**), we exit the function, returning **1**. We are now at the second recursive call (statement 95). We again call recursively, this time with the expression **`BC/EF`**, which returns with a length of **1**.

We now return from the call having parsed the expression **`*ABC/EF`**. The return value is **3**, representing **1** for the first operand (**A**) and **1** for the second operand (**B**) and **1** for the operator (*****).

The second recursive call in the decomposition of **`+*ABC/EF`** passes the expression after **`*AB`**, which is **`C/EF`**. Because the first character is an operand, which is a base case, the function terminates and returns **1**. This completes the isolation of the first expression, **`+*ABC`**, by returning its length, **5**. As shown in Figure 2-9, we are not done. The decomposition of the expression continues until the complete prefix expression has been converted.

The Towers of Hanoi

The Towers of Hanoi is a classic recursion problem that is relatively easy to follow, is efficient, and uses no complex data structures.

According to the legend, the monks in a remote mountain monastery knew how to predict when the world would end. They had a set of three diamond needles. Stacked on the first diamond needle were 64 gold disks of decreasing size. The monks moved one disk to another needle each hour, subject to the following rules:

1. Only one disk could be moved at a time. A larger disk must never be stacked above a smaller one.
2. One and only one auxiliary needle could be used for the intermediate storage of disks.

The legend said that when all 64 disks had been transferred to the destination needle, the stars would be extinguished and the world would end. Today we know that we need to have $2^{64} - 1$ moves to move all of the disks. Figure 2-11 shows the Towers of Hanoi with only three disks.

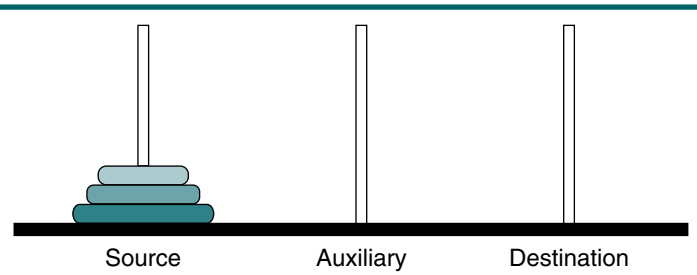


FIGURE 2-11 Towers of Hanoi—Start Position

This problem is interesting for two reasons. First, the recursive solution is much easier to code than the iterative solution would be, as is often the case with good recursive solutions. Second, its solution pattern is different from the simple examples we have been discussing. As you study the towers solution, note that after each base case, we return to a decomposition of the general case for several steps. In other words, the problem is divided into several subproblems, each of which terminates with the base case, moving one disk.

Recursive Towers of Hanoi Design

To solve this problem, we must study the moves to see if we can find a pattern. We will use only three disks because we do not want the world to end. First, imagine that we have only one disk to move. This is a very simple case, as shown on the following page.

Case 1: Move one disk from source to destination needle.

Now imagine that we have to move two disks. Figure 2-12 traces the steps for two disks.

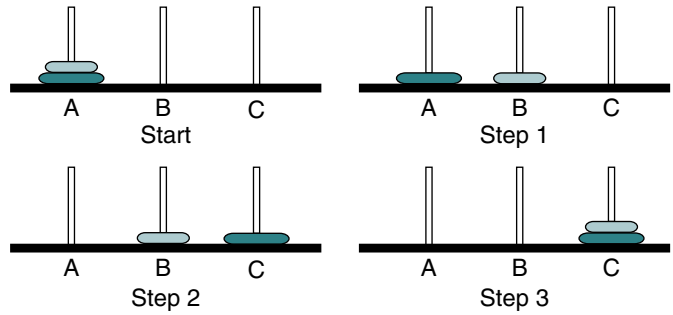


FIGURE 2-12 Towers Solution for Two Disks

First we move the top disk to the auxiliary needle. Then we move the second disk to the destination needle. Finally, we move the first disk to the top of the second disk on the destination needle. This gives us the second case, as shown below.

Case 2: 1. Move one disk to auxiliary needle.
2. Move one disk to destination needle.
3. Move one disk from auxiliary to destination needle.

We are now ready to study the case for three disks. Its solution is shown in Figure 2-13. The first three steps move the top two disks from the source to the auxiliary needle. (To see how to do this, refer to Case 2.) In step 4 we move the bottom disk to the destination needle. We now have one disk in place. This is an example of Case 1. We then need three more steps to move the two disks on the auxiliary needle to the destination needle.

These steps are summarized in Case 3 below.

Case 3: 1. Move two disks from source to auxiliary needle.
2. Move one disk from source to destination needle.
3. Move two disks from auxiliary to destination needle.

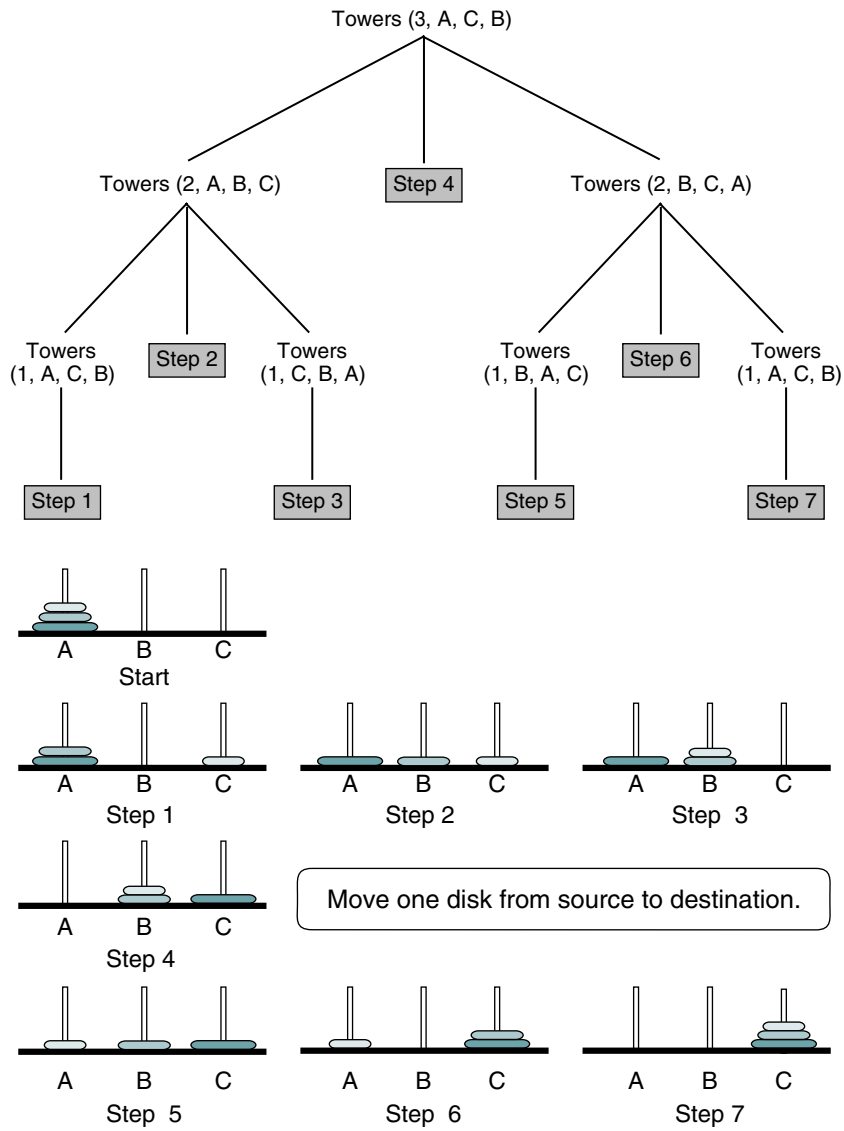


FIGURE 2-13 Towers Solution for Three Disks

We are now ready to generalize the problem.

1. Move $n-1$ disks from source to auxiliary.	**General case**
2. Move one disk from source to destination.	**Base case**
3. Move $n-1$ disks from auxiliary to destination.	**General case**

Our solution requires an algorithm with four parameters: the number of disks to be moved, the source needle, the destination needle, and the auxiliary needle. Using pseudocode the three moves in the generalization shown above are then

```
1. Call Towers (n - 1, source, auxiliary, destination)
2. Move one disk from source to destination
3. Call Towers (n - 1, auxiliary, destination, source)
```

Study the second call above carefully. After we complete the move of the first disk, the remaining disks are on the auxiliary needle. We need to move them from the auxiliary needle to the destination needle. In this case the original source needle becomes the auxiliary needle. (Remember that the positions of the parameters in the called algorithm are source, destination, and auxiliary. The calling algorithm must remember which of the three needles is the source and which is the destination for each call.)

We can now put these three calls together. In place of physical moves, we use print statements to show the moves that need to be made. The complete pseudocode is shown in Algorithm 2-7.

ALGORITHM 2-7 Towers of Hanoi

```
Algorithm towers (numDisks, source, dest, auxiliary)
Recursively move disks from source to destination.
   Pre  numDisks is number of disks to be moved
        source, destination, and auxiliary towers given
   Post steps for moves printed
 1 print("Towers: ", numDisks, source, dest, auxiliary)
 2 if (numDisks is 1)
   1  print ("Move from ", source, " to ", dest)
 3 else
   1  towers (numDisks - 1, source, auxiliary, dest, step)
   2  print ("Move from " source " to " dest)
   3  towers (numDisks - 1, auxiliary, dest, source, step)
 4 end if
end towers
```

Algorithm 2-7 Analysis

Each time we enter the algorithm, we print the current parameters; this helps us keep track of which tower is currently the source, which is the destination, and which is the auxiliary. Because the towers are constantly changing their role among the three, printing the call parameters helps us keep them straight.

Two statements print the move instructions. The first is at statement 2.1. It prints the instructions when there is only one disk left on a tower. The second move instruction is printed at statement 3.2. It is printed whenever we return from the recursion and provides the instructions to move a disk when there are more than one on a tower.

It is important to study the parameters in the two calls. Note how they are moving the towers' roles from destination to auxiliary in statement 3.1 and from auxiliary to source in statement 3.3.

The output from the algorithm is shown in Figure 2-14. You will undoubtedly need to trace the program with the output to follow the recursion.

Calls:	Output:
`Towers (3, A, C, B)`	
`Towers (2, A, B, C)`	
`Towers (1, A, C, B)`	
	`Move from A to C`
	`Move from A to B`
`Towers (1, C, B, A)`	
	`Move from C to B`
	`Move from A to C`
`Towers (2, B, C, A)`	
`Towers (1, B, A, C)`	
	`Move from B to A`
	`Move from B to C`
`Towers (1, A, C, B)`	
	`Move from A to C`

FIGURE 2-14 Tracing Algorithm 2-7, Towers of Hanoi

Towers of Hanoi C Implementation

Once again we see the elegance of recursion. With only 56 lines of code; we solve a relatively difficult problem. Furthermore, the recursive portion of the algorithm that does all of the work is only seven statements long—including three print statements! As is often the case with recursive algorithms, the analysis and design of the algorithm takes longer than the time to write it. The code is shown in Program 2-4. The output is shown in Figure 2-14.

PROGRAM 2-4 Towers of Hanoi

```
 1  /* Test Towers of Hanoi
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  // Prototype Statements
 8     void towers (int  n,     char source,
 9                  char dest,  char auxiliary);
10
11  int main (void)
12  {
13  // Local Declarations
14     int numDisks;
15
```

continued

PROGRAM 2-4 Towers of Hanoi *(continued)*

```
// Statements
   printf("Please enter number of disks: ");
   scanf ("%d", &numDisks);

   printf("Start Towers of Hanoi.\n\n");

   towers (numDisks, 'A', 'C', 'B');

   printf("\nI Hope you didn't select 64 "
            "and end the world!\n");
   return 0;
}  // main

/* ==================== towers ====================
   Move one disk from source to destination through
   the use of recursion.
      Pre  The tower consists of n disks
           Source, destination, & auxiliary towers
      Post Steps for moves printed
*/
void towers (int   n,    char  source,
             char  dest, char  auxiliary)
{
// Local Declarations
   static int step = 0;

// Statements
   printf("Towers (%d, %c, %c, %c)\n",
                  n, source, dest, auxiliary);
   if (n == 1)
      printf("\t\t\tStep %3d: Move from %c to %c\n",
             ++step, source, dest);
   else
      {
       towers (n - 1, source, auxiliary, dest);
       printf("\t\t\tStep %3d: Move from %c to %c\n",
              ++step, source, dest);
       towers (n - 1, auxiliary, dest, source);
      } // if … else
   return;
}  // towers
```

2.4 Key Terms

base case
factorial
Fibonacci numbers
general case
greatest common divisor (GCD)
infix
postfix
prefix
recursion

2.5 Summary

- There are two approaches to writing repetitive algorithms: iteration and recursion.
- Recursion is a repetitive process in which an algorithm calls itself.
- A repetitive algorithm uses recursion whenever the algorithm appears within the definition itself.
- When a program calls a subroutine, the current module suspends processing and the called subroutine takes over the control of the program. When the subroutine completes its processing and returns to the module that called it, the module wakes up and continues its processing.
- In a recursive algorithm, each call either solves part of the problem or reduces the size of the problem.
- The statement that solves the problem is known as the base case; every recursive algorithm must have a base case.
- The rest of the recursive algorithm is known as the general case.
- The general rule for designing a recursive algorithm is as follows:
  1. Determine the base case.
  2. Determine the general case.
  3. Combine the base case and the general case into an algorithm.
- You should not use recursion if the answer to any of the following questions is no:
  1. Is the algorithm or data structure naturally suited to recursion?
  2. Is the recursive solution shorter and more understandable?
  3. Does the recursive solution run within acceptable time and space limits?

2.6 Practice Sets

Exercises

1. Consider the following algorithm:

```
algorithm fun1 (x)
1  if (x < 5)
   1   return (3 * x)
2  else
   1   return (2 * fun1 (x - 5) + 7)
3  end if
end fun1
```

What would be returned if `fun1` is called as

a. `fun1 (4)`?
b. `fun1 (10)`?
c. `fun1 (12)`?

2. Consider the following algorithm:

```
algorithm fun2 (x, y)
1  if (x < y)
   1   return -3
2  else
   1   return (fun2 (x - y, y + 3) + y)
3  end if
end fun2
```

What would be returned if `fun2` is called as

a. `fun2 (2, 7)`?
b. `fun2 (5, 3)`?
c. `fun2 (15, 3)`?

3. Consider the following algorithm:

```
algorithm fun3 (x, y)
1  if (x > y)
   1   return -1
2  elseif (x equal y)
   1   return 1
3  else
   1   return (x * fun3 (x + 1, y))
4  end if
end fun3
```

What would be returned if `fun3` is called as

a. `fun3 (10,4)`?
b. `fun3 (4,3)`?
c. `fun3 (4,7)`?
d. `fun3 (0,0)`?

4. One of the methods to calculate the square root of a number is Newton's method. The formula for Newton's method is shown in Figure 2-15. Write the pseudocode for a recursive algorithm to compute a square root using Newton's method. Verify your algorithm by using it to manually calculate the following test cases: `squareRoot` (5, 2, 0.01) and `squareRoot` (4, 2, 0.01). *Note*: in the formula, tol is an abbreviation for *tolerance*.

$$\text{squareRoot}(num, ans, tol) = \begin{cases} ans & \text{if } |ans^2 - num| \le tol \\ \text{squareRoot}(num, (ans^2 + num)/(2 \times ans), tol) & \text{otherwise} \end{cases}$$

FIGURE 2-15 Newton's Method for Exercise 4

5. The combination of n objects, such as balls in a basket, taken k at a time can be calculated recursively using the formula shown in Figure 2-16. This is a useful formula. For example, several state lotteries require players to choose six numbers out of a series of possible numbers. This formula can be used to calculate the number of possible combinations, k, of n objects. For example, for 49 numbers, there are `C(49, 6)`, or 13,983,816, different combinations of six numbers. Write a recursive algorithm to calculate the combination of n objects taken k at a time.

$$C(n, k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k \\ C(n, k) = C(n-1, k) + C(n-1, k-1) & \text{if } n > k > 0 \end{cases}$$

FIGURE 2-16 Selection Algorithm for Exercise 5

6. Ackerman's number, used in mathematical logic, can be calculated using the formula shown in Figure 2-17. Write a recursive algorithm that calculates Ackerman's number. Verify your algorithm by using it to manually calculate the following test cases: `Ackerman(2, 3)`, `Ackerman(2, 5)`, `Ackerman(0, 3)`, and `Ackerman(3, 0)`.

$$\text{Ackerman}(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ \text{Ackerman}(m - 1, 1) & \text{if } n = 0 \text{ and } m > 0 \\ \text{Ackerman}(m - 1, \text{Ackerman}(m, n - 1)) & \text{otherwise} \end{cases}$$

FIGURE 2-17 Ackerman Formula for Problem 6

Problems

7. Write a recursive algorithm that calculates and returns the length of a list.
8. Write a recursive algorithm that converts a string of numerals to an integer. For example, "43567" will be converted to 43567.
9. Write a recursive algorithm to add the first *n* elements of the series

   ```
   1 + 1/2 + 1/3 + 1/4 + 1/5 + ... + 1/n
   ```

10. Write a recursive algorithm to determine whether a string is a palindrome. A string is a palindrome if it can be read forward and backward with the same meaning. Capitalization and spacing are ignored. For example, *anna* and *go dog* are palindromes. Test your algorithm with the following two palindromes and at least one case that is not a palindrome.

    Madam, I'm Adam

    Able was I ere I saw Elba

11. Write a recursive algorithm to check whether a specified character is in a string.
12. Write a recursive algorithm to count all occurrences of a specified character in a string.
13. Write a recursive algorithm that removes all occurrences of a specified character from a string.
14. Write a recursive algorithm that finds all occurrences of a substring in a string.
15. Write a recursive algorithm that changes an integer to a binary number.
16. Write a recursive C function to calculate the square root of a number using Newton's method. (See Exercise 4.) Test your function by printing the square root of 125, 763, and 997.
17. Write a recursive algorithm that reads a string of characters from the keyboard and prints them reversed.
18. The combination of *n* objects, such as balls in a basket, taken *k* at a time can be calculated recursively using the combination formula. (See Exercise 5.) Write a C function to calculate the number of combinations present in a number. Test your function by computing and printing `C(10, 3)`.

19. Ackerman's number, used in mathematical logic, can be calculated using the formula shown in Exercise 6. Write a recursive C function that calculates Ackerman's number. Verify your algorithm by using it to calculate the following test cases: `Ackerman(5,2)`, `Ackerman(2,5)`, `Ackerman(0,3)`, and `Ackerman(3,0)`.
20. Write the C function for the recursive algorithm that prints the elements of a list in reverse order. (See Algorithm 2-3, "Print Reverse.")

Projects

21. If a recursion call is the last executable statement in the algorithm, called tail recursion, it can easily be removed using iteration. Tail recursion is so named because the return point of each call is at the end of the algorithm. Thus, there are no executable statements to be executed after each call. To change a tail recursion to an iteration, we use the following steps:

    a. Use a variable to replace the procedure call.
    b. Use a loop with the limit condition as the base case (or its complement).
    c. Enclose all executable statements inside the loop.
    d. Change the recursive call to an appropriate assignment statement.
    e. Use appropriate statements to reassign values to parameters.
    f. Return the value of the variable defined in step a.

    Write the iterative version of the recursion factorial algorithm (Algorithm 2-2) and test it by printing the value of `factorial(5)` and `factorial(20)`.
22. Write the iterative version of the Fibonacci series algorithm using the hints given in Project 21. Note that step c in Project 21 will be different because factorial uses two recursive calls in the last statement.

This page intentionally left blank

Part II

Linear Lists

A **linear list** is a list in which each element has a unique successor. The sequential property of a linear list, as shown in Figure II-1, is basic to its definition and use.

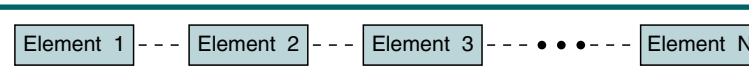


FIGURE II-1 Linear List

Linear lists can be divided into two categories: restricted lists and general lists. In a **restricted list**, addition and deletion of data are restricted to the ends of the list. We describe two restricted list structures: the last in–first out (LIFO) list, commonly called a stack, and the first in–first out (FIFO) list, commonly called a queue. We discuss stacks in Chapter 3 and queues in Chapter 4.

In a linear list, each element has a unique successor.

In a **general list**, data can be inserted or deleted anywhere in the list: at the beginning, in the middle, or at the end. We discuss general lists in Chapter 5. Figure II-2 shows the linear list categories as represented in this book.

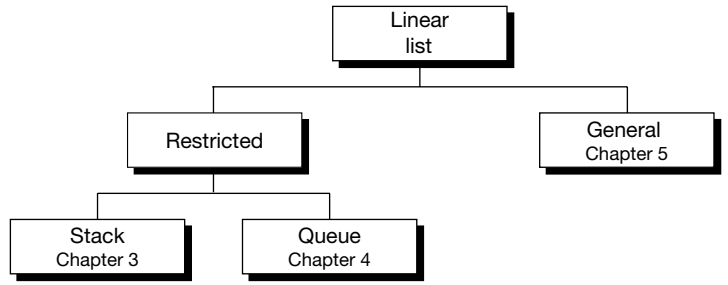


FIGURE II-2 Linear List Categories

Chapters Covered

This part includes three chapters:

Chapter 3: Stacks

In this chapter, we discuss the stack, a restricted linear list in which data can be inserted and deleted only at one end.

Chapter 4: Queues

In this chapter, we discuss the queue, a restricted linear list in which data can be inserted only at one end and deleted only at the other.

Chapter 5: General Linear Lists

In this chapter, we discuss general linear lists in which data can be inserted and deleted anywhere: beginning, middle, or end.

Chapter 3
Stacks

A **stack** is a linear list in which all additions and deletions are restricted[1] to one end, called the **top.** If you insert a data series into a stack and then remove it, the order of the data is reversed. Data input as {5, 10, 15, 20} is removed as {20, 15, 10, 5}. This reversing attribute is why stacks are known as the **last in–first out (LIFO)** data structure.

We use many different types of stacks in our daily lives. We often talk of a stack of coins or a stack of dishes. Any situation in which you can only add or remove an object at the top is a stack. If you want to remove any object other than the one at the top, you must first remove all objects above it. A graphic representation of a stack is shown in Figure 3-1.

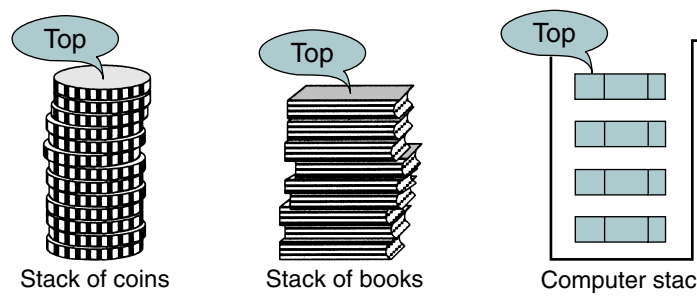


FIGURE 3-1 Stack

1. The stack is one of three data structures known collectively as *restrictive data structures* because the operations are restricted to the ends of the structure. The other two are the queue, which we study in Chapter 4, and the deque or double-ended queue. The deque, which is not covered in this text, allows insertions and deletions at both ends.

Although nothing prevents us from designing a data structure that allows us to perform other operations, such as moving the item at the top of the stack to the bottom, the result would not be a stack.

A stack is a last in–first out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.

3.1 Basic Stack Operations

The three basic stack operations are push, pop, and stack top. Push is used to insert data into the stack. Pop removes data from a stack and returns the data to the calling module. Stack top returns the data at the top of the stack without deleting the data from the stack.

Push

Push adds an item at the top of the stack. After the push, the new item becomes the top. The only potential problem with this simple operation is that we must ensure that there is room for the new item. If there is not enough room, the stack is in an **overflow** state and the item cannot be added. Figure 3-2 shows the push stack operation.

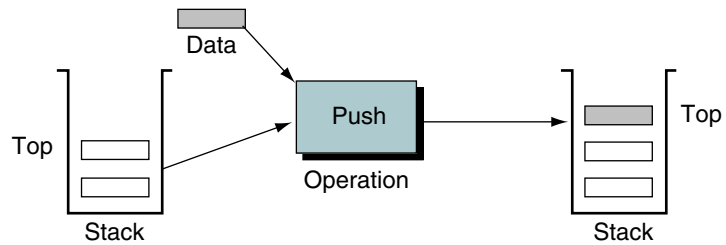


FIGURE 3-2 Push Stack Operation

Pop

When we **pop** a stack, we remove the item at the top of the stack and return it to the user. Because we have removed the top item, the next older item in the stack becomes the top. When the last item in the stack is deleted, the stack must be set to its empty state. If pop is called when the stack is empty, it is in an **underflow** state. The pop stack operation is shown in Figure 3-3.

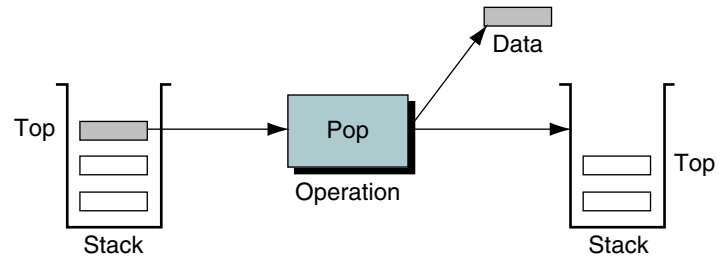


FIGURE 3-3 Pop Stack Operation

Stack Top

The third stack operation is **stack top**. Stack top copies the item at the top of the stack; that is, it returns the data in the top element to the user but does not delete it. You might think of this operation as reading the stack top. Stack top can also result in underflow if the stack is empty. The stack top operation is shown in Figure 3-4.

The three basic stack operations are push, pop, and stack top.

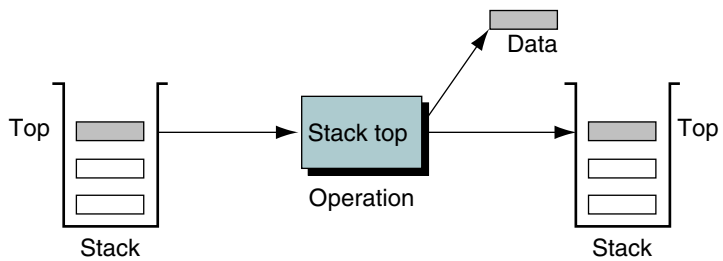


FIGURE 3-4 Stack Top Operation

Figure 3-5 traces these three operations in an example. We start with an empty stack and push *green* and *blue* into the stack. At this point the stack contains two entries. We then pop *blue* from the top of the stack, leaving *green* as the only entry. After pushing *red*, the stack again contains two entries. At this point we retrieve the top entry, *red*, using stack top. Note that stack top does not remove *red* from the stack; it is still the top element. We then pop *red*, leaving *green* as the only entry. When *green* is popped, the stack is again empty. Note also how this example demonstrates the last in–first out operation of a stack. Although *green* was pushed first, it is the last to be popped.

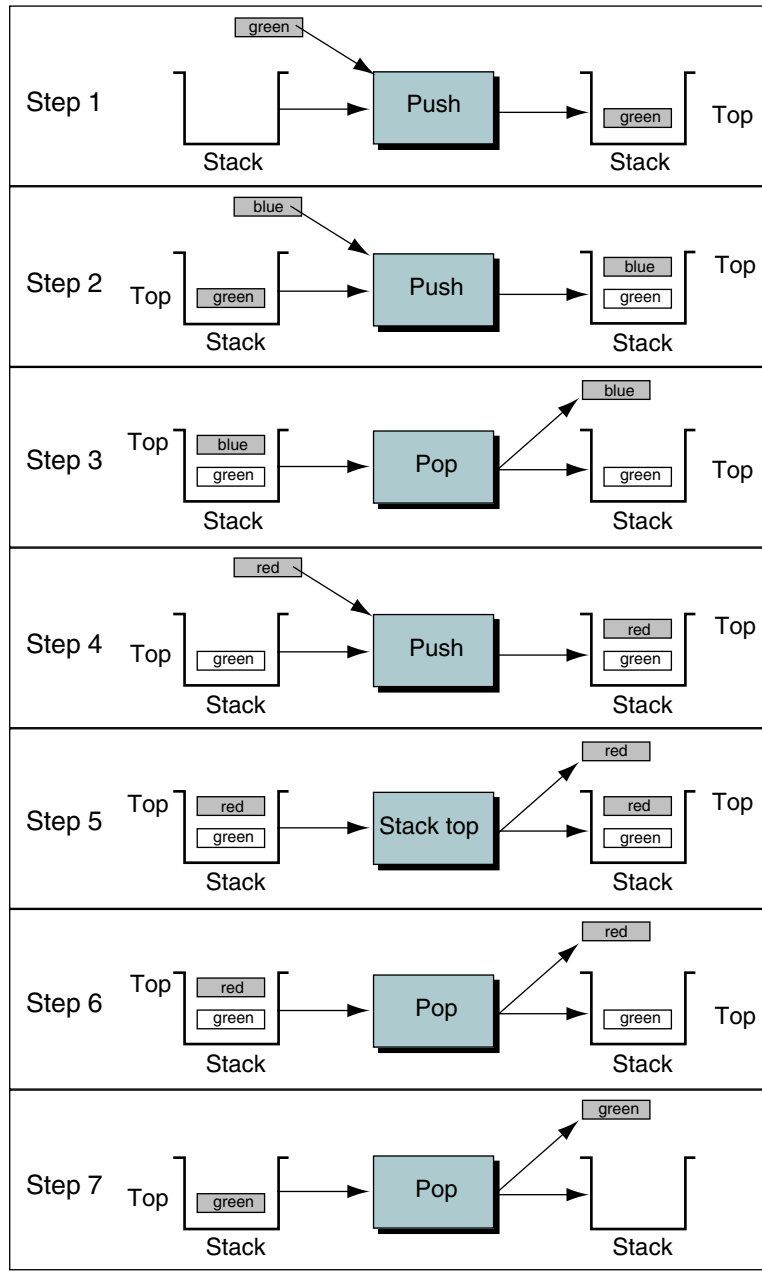


FIGURE 3-5 Stack Example

3.2 Stack Linked List Implementation

Several data structures can be used to implement a stack. In this section we implement the stack as a linked list. We introduced the concept of a linked list in Chapter 1, "Linked List Implementations."

Data Structure

To implement the linked list stack, we need two different structures, a head and a data node. The head structure contains **metadata**—that is, data about data—and a pointer to the top of the stack. The data structure contains data and a link pointer to the next node in the stack. The conceptual and physical implementations of the stack are shown in Figure 3-6.

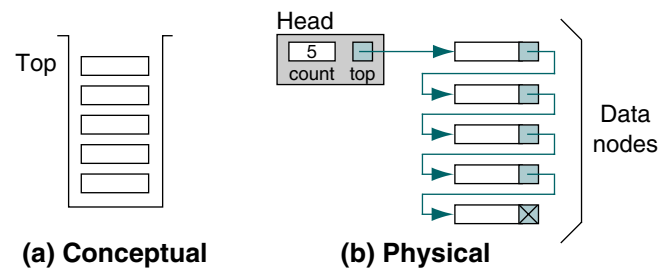


FIGURE 3-6 Conceptual and Physical Stack Implementations

Stack Head

Generally, the **head** for a stack requires only two attributes: a top pointer and a count of the number of elements in the stack. These two elements are placed in a structure. Other stack attributes can be placed here also. For example, it is possible to record the time the stack was created and the total number of items that have ever been placed in the stack. These two metadata items allow the user to determine the average number of items processed through the stack in a given period. Of course, we would do this only if such a statistic were required for some reason. A basic head structure is shown in Figure 3-7.

Stack Data Node

The rest of the data structure is a typical linked list **data node**. Although the application determines the data that are stored in the stack, the stack data node looks like any linked list node. In addition to the data, it contains a link pointer to other data nodes, making it a **self-referential data structure**. In a self-referential structure, each instance of the structure contains a pointer to another instance of the same structure. The stack data node is also shown in Figure 3-7.

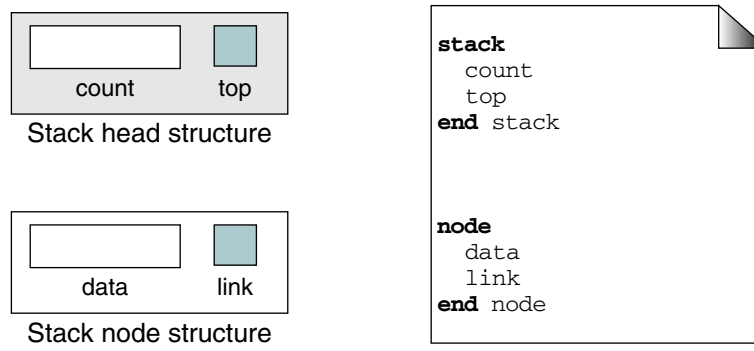


FIGURE 3-7 Stack Data Structure

Stack Algorithms

The eight stack operations defined in this section should be sufficient to solve any basic stack problem. If an application requires additional stack operations, they can be easily added. For each operation we give its name, a brief description, and then develop its algorithm.

Although the implementation of a stack depends somewhat on the implementation language, it is usually implemented with a stack head structure in C. We use the design shown in Figure 3-8, which demonstrates the four most common stack operations: create stack, push stack, pop stack, and destroy stack. Operations such as `stacktop` are not shown in the figure because they do not change the stack structure.

Create Stack

Create stack allocates memory for the stack structure and initializes its metadata. The pseudocode is shown in Algorithm 3-1.

ALGORITHM 3-1 Create Stack

```
Algorithm createStack
Creates and initializes metadata structure.
   Pre    Nothing
   Post   Structure created and initialized
   Return stack head
 1 allocate memory for stack head
 2 set count to 0
 3 set top to null
 4 return stack head
end createStack
```

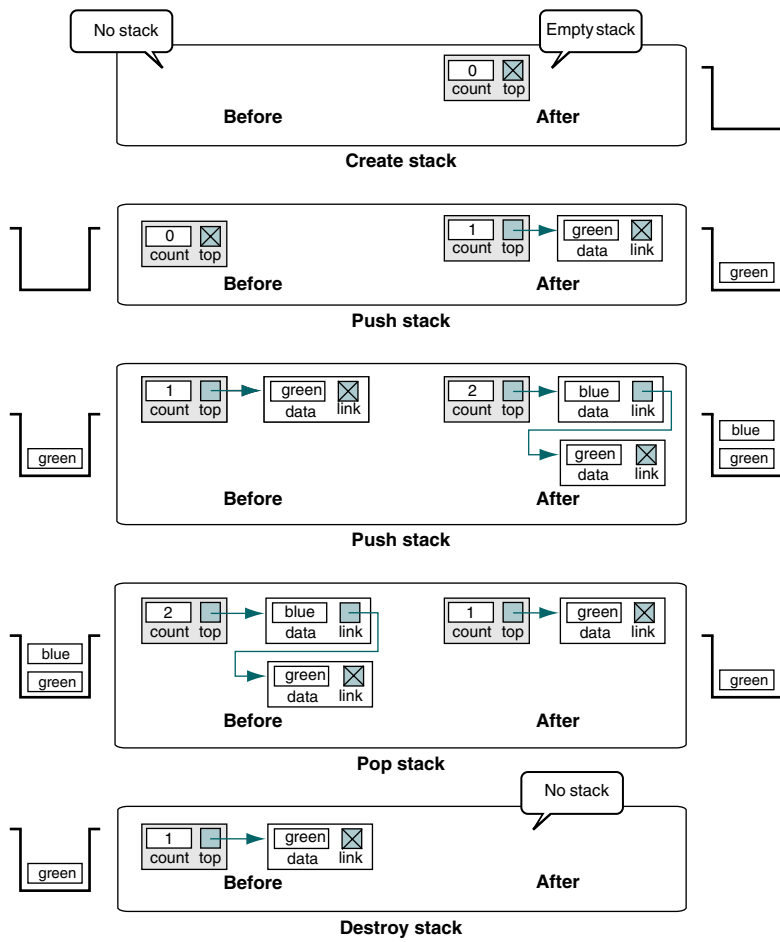


FIGURE 3-8 Stack Operations

Push Stack

Push stack inserts an element into the stack. The first thing we need to do when we push data into a stack is find memory for the node. We must therefore allocate a node from dynamic memory. Once the memory is allocated, we simply assign the data to the stack node and then set the link pointer to point to the node currently indicated as the stack top. We also need to update the stack top pointer and add 1 to the stack count field. Figure 3-9 traces a push stack operation in which a new pointer (`pNew`) is used to identify the data to be inserted into the stack.

To develop the insertion algorithm using a linked list, we need to analyze three different stack conditions: (1) insertion into an empty stack, (2) insertion into a stack with data, and (3) insertion into a stack when the available memory is exhausted. The first two of these situations are shown in Figure 3-8. The third is an error condition.

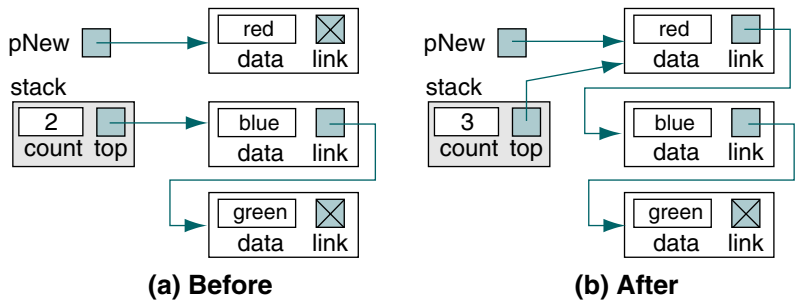


FIGURE 3-9 Push Stack Example

When we insert into a stack that contains data, the new node's link pointer is set to point to the node currently at the top, and the stack's top pointer is set to point to the new node. When we insert into an empty stack, the new node's link pointer is set to null and the stack's top pointer is set to point to the new node. However, because the stack's top pointer is null, we can use it to set the new node's link pointer to null. Thus the logic for inserting into a stack with data and the logic for inserting into an empty stack are identical.

ALGORITHM 3-2 Push Stack Design

```
Algorithm pushStack (stack, data)
Insert (push) one item into the stack.
   Pre  stack passed by reference
        data contain data to be pushed into stack
   Post data have been pushed in stack
 1  allocate new node
 2  store data in new node
 3  make current top node the second node
 4  make new node the top
 5  increment stack count
end pushStack
```

Pop Stack

Pop stack sends the data in the node at the top of the stack back to the calling algorithm. It then adjusts the pointers to logically delete the node. After the node has been logically deleted, it is physically deleted by recycling the memory, that is, returning it to dynamic memory. After the count is adjusted by subtracting 1, the algorithm returns the status to the caller: if the pop was successful, it returns true; if the stack is empty when pop is called, it returns false. The operations for pop stack are traced in Figure 3-10.

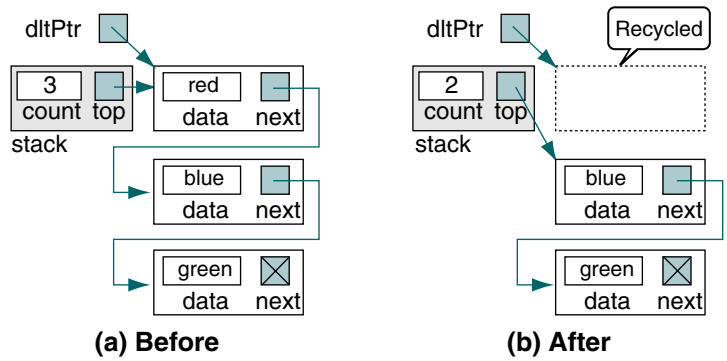


FIGURE 3-10 Pop Stack Example

The pop stack code is shown in Algorithm 3-3.

ALGORITHM 3-3 Pop Stack

```
Algorithm popStack (stack, dataOut)
This algorithm pops the item on the top of the stack and
returns it to the user.
   Pre    stack passed by reference
          dataOut is reference variable to receive data
   Post   Data have been returned to calling algorithm
   Return true if successful; false if underflow
 1 if (stack empty)
   1  set success to false
 2 else
   1  set dataOut to data in top node
   2  make second node the top node
   3  decrement stack count
   4  set success to true
 3 end if
 4 return success
end popStack
```

Algorithm 3-3 Analysis

It is interesting to follow the logic when the last node is being deleted. In this case, the result is an empty stack. No special logic is required; however, the empty stack is created automatically because the last node has a null link pointer, which when moved to top indicates that the stack is empty. A count of zero, which automatically occurs when we decrement the count, is also an indication of an empty stack.

Stack Top

The stack top algorithm (Algorithm 3-4) sends the data at the top of the stack back to the calling module without deleting the top node. This is

included in case the application needs to know what will be deleted with the next pop stack.

ALGORITHM 3-4 Stack Top Pseudocode

```
Algorithm stackTop  (stack, dataOut)
This algorithm retrieves the data from the top of the stack
without changing the stack.
   Pre    stack is metadata structure to a valid stack
          dataOut is reference variable to receive data
   Post   Data have been returned to calling algorithm
   Return true if data returned, false if underflow
 1 if (stack empty)
   1  set success to false
 2 else
   1  set dataOut to data in top node
   2  set success to true
 3 end if
 4 return success
end stackTop
```

Algorithm 3-4 Analysis

The logic for the stack top is virtually identical to that for the pop stack except for the delete logic. As with the pop, there is only one potential problem with the algorithm: the stack may be empty. If the algorithm is successful, it returns true; if the stack is empty, it returns false.

Empty Stack

Empty stack is provided to implement the structured programming concept of data hiding. If the entire program has access to the stack head structure, it is not needed. However, if the stack is implemented as a separately compiled program to be linked with other programs, the calling program may not have access to the stack head node. In these cases it is necessary to provide a way to determine whether the stack is empty. The pseudocode for empty stack is shown in Algorithm 3-5.

ALGORITHM 3-5 Empty Stack

```
Algorithm emptyStack (stack)
Determines if stack is empty and returns a Boolean.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack status
   Return true if stack empty, false if stack contains data
 1 if (stack count is 0)
   1  return true
 2 else
   1  return false
 3 end if
end emptyStack
```

Full Stack

Full stack is another structured programming implementation of data hiding. Depending on the language, it may also be one of the most difficult algorithms to implement. ANSI C, for instance, provides no direct way to implement it. The pseudocode for full stack is shown in Algorithm 3-6.

ALGORITHM 3-6 Full Stack

```
Algorithm fullStack  (stack)
Determines if stack is full and returns a Boolean.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack status
   Return true if stack full, false if memory available
 1 if (memory not available)
   1   return true
 2 else
   1   return false
 3 end if
end fullStack
```

Stack Count

Stack count returns the number of elements currently in the stack. It is another implementation of the data-hiding principle of structured programming. The pseudocode is shown in Algorithm 3-7.

ALGORITHM 3-7 Stack Count

```
Algorithm stackCount (stack)
Returns the number of elements currently in stack.
   Pre    stack is metadata structure to a valid stack
   Post   returns stack count
   Return integer count of number of elements in stack
 1 return (stack count)
end stackCount
```

Destroy Stack

Destroy stack deletes all data in a stack. Figure 3-8 graphically shows the results of destroy stack, and Algorithm 3-8 contains the pseudocode.

ALGORITHM 3-8 Destroy Stack

```
Algorithm destroyStack (stack)
This algorithm releases all nodes back to the dynamic memory.
   Pre    stack passed by reference
   Post   stack empty and all nodes deleted
 1 if (stack not empty)
```

continued

ALGORITHM 3-8 Destroy Stack *(continued)*

```
   1  loop (stack not empty)
      1  delete top node
   2  end loop
 2 end if
 3 delete stack head
end destroyStack
```

Algorithm 3-8 Analysis It is only necessary to destroy a stack when you no longer need it and the program is not complete. If the program is complete, the stack is automatically cleared when the program terminates.

3.3 C Language Implementations

As previously stated, there are two approaches to implementing stacks. We can write unique C programs, which are easy to write but not reusable, or we can create an abstract data type (ADT). In this section we develop simple, but not very reusable, programs. In the next section, we develop the stack ADT.

To demonstrate the push and pop stack algorithms, we write a program that inserts data into a stack. To keep it simple, the data is random uppercase characters. After the characters have been inserted, they are popped and printed. When the stack is empty, the program terminates. While we are not using a stack header, we are using the basic push and pop algorithms discussed in Section 3.2, "Stack Linked List Implementation." The design for this program is shown in Figure 3-11.

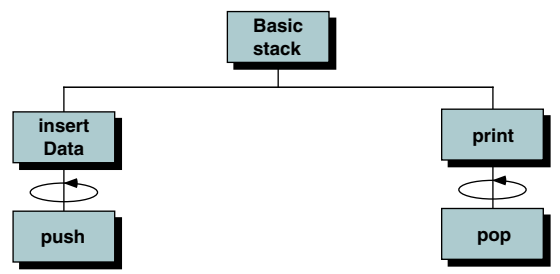


FIGURE 3-11 Design for Basic Stack Program

The node declaration, prototype statements, and test driver are contained in Program 3-1.

PROGRAM 3-1 Simple Stack Application Program

```
/* This program is a test driver to demonstrate the
   basic operation of the stack push and pop functions.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Structure Declarations
typedef struct node
   {
    char         data;
    struct node* link;
   } STACK_NODE;

// Prototype Declarations
void insertData (STACK_NODE** pStackTop);
void print      (STACK_NODE** pStackTop);

bool push       (STACK_NODE** pList, char  dataIn);
bool pop        (STACK_NODE** pList, char* dataOut);

int main (void)
{
// Local Definitions
STACK_NODE* pStackTop;

// Statements
   printf("Beginning Simple Stack Program\n\n");

   pStackTop = NULL;
   insertData  (&pStackTop);
   print       (&pStackTop);

   printf("\n\nEnd Simple Stack Program\n");
   return 0;
}  // main
```

```
Results:
   Beginning Simple Stack Program

   Creating characters: QMZRHLAJOE
   Stack contained:     EOJALHRZMQ

   End Simple Stack Program
```

Program 3-1 Analysis To verify that the program works correctly, we print the characters as they are generated. This allows us to verify that the stack output was in fact correct. Note that this simple program verifies the LIFO operation of a stack.

Insert Data

The insert data function loops while creating characters and inserting them into the stack. To create random characters, we use the random number generator and scale the return value to the uppercase alphabetic range. This code is developed in Program 3-2.

PROGRAM 3-2 Insert Data

```
/* ================== insertData ==================
   This program creates random character data and
   inserts them into a linked list stack.
      Pre  pStackTop is a pointer to first node
      Post Stack has been created
*/
void insertData (STACK_NODE** pStackTop)
{
// Local Definitions
   char  charIn;
   bool  success;

// Statements
   printf("Creating characters: ");
   for (int nodeCount = 0; nodeCount < 10; nodeCount++)
       {
        // Generate uppercase character
        charIn  = rand() % 26 + 'A';
        printf("%c", charIn);
        success = push(pStackTop, charIn);
        if (!success)
           {
            printf("Error 100: Out of Memory\n");
            exit (100);
           } // if
       } // for
   printf("\n");
   return;
}  // insertData
```

Program 3-2 Analysis

The insert logic is straightforward. The generation of the uppercase characters requires a little explanation. Because there are 26 alphabetic characters, we scale the random number to the range 0 to 25. To map the resulting random number to the uppercase characters, we then add the value of the character `'A'` to the random number.

Also note that even in this small example, we test for overflow. Although the probability of an overflow occurring is small, we always test for it. Remember Murphy's Law.[2]

2. Murphy's Law: If something can go wrong, it will.

Push Stack

The insert data function calls the push function. With the exception of testing for overflow technique and the error-handling code, and maintaining the header metadata, it is a direct implementation of Algorithm 3-2. The code is developed in Program 3-3.

PROGRAM 3-3 Push Stack

```
/* ================== push ====================
   Inserts node into linked list stack.
      Pre     pStackTop is pointer to valid stack
      Post    charIn inserted
      Return  true  if successful
              false if underflow
*/
bool push (STACK_NODE** pStackTop, char charIn)
{
// Local Definitions
   STACK_NODE* pNew;
   bool        success;

// Statements
   pNew = (STACK_NODE*)malloc(sizeof (STACK_NODE));
   if (!pNew)
       success = false;
   else
      {
       pNew->data =  charIn;
       pNew->link = *pStackTop;
       *pStackTop =  pNew;
       success = true;
      } // else
   return success;
}  // push
```

Print Stack

Once the stack has been built, we print it to verify the output. The print function calls the pop function until the stack is empty. The print code is found in Program 3-4.

PROGRAM 3-4 Print Stack

```
/* ==================== print ====================
   This function prints a singly linked stack.
      Pre     pStackTop is pointer to valid stack
      Post    data in stack printed
*/
void print (STACK_NODE** pStackTop)
{
```

continued

PROGRAM 3-4 Print Stack *(continued)*

```
 8  // Local Definitions
 9     char printData;
10
11  // Statements
12     printf("Stack contained:     ");
13     while (pop(pStackTop, &printData))
14         printf("%c", printData);
15     return;
16  }  // print
```

Pop Character

The pop character function copies the character at the top of the stack to the output location specified in the second parameter. With the exception of the code to maintain the header metadata, it is an exact implementation of Algorithm 3-3. The implementation is shown in Program 3-5.

PROGRAM 3-5 Pop Stack

```
 1  /* ================== pop ====================
 2     Delete node from linked list stack.
 3        Pre  pStackTop is pointer to valid stack
 4        Post charOut contains deleted data
 5        Return  true  if successful
 6                false if underflow
 7  */
 8  bool pop (STACK_NODE** pStackTop, char* charOut)
 9  {
10  // Local Definitions
11     STACK_NODE* pDlt;
12     bool        success;
13
14  // Statements
15     if (*pStackTop)
16        {
17         success    = true;
18         *charOut   = (*pStackTop)->data;
19         pDlt       = *pStackTop;
20         *pStackTop = (*pStackTop)->link;
21         free (pDlt);
22        } // else
23     else
24        success = false;
25     return success;
26  }  // pop
```

3.4 Stack ADT

Each of the algorithms in the previous section used several stack operations. While we could write them each time we needed them, this would certainly be an inefficient use of our time. What we need, therefore, is a stack abstract data type that we can put in a library and call whenever we need it. We develop a stack abstract data type in this section.[3]

Data Structure

The stack ADT implementation in C is straightforward. Rather than store data in each node, we store a pointer to the data. It is the application program's responsibility to allocate memory for the data and pass the address to the stack ADT. Within the ADT, the stack node looks like any linked list node except that it contains a pointer to the data rather than the actual data. Because the data pointer type is unknown, it is stored as a pointer to *void*.

The head node and the data nodes are encapsulated in the ADT. The calling function's only view of the stack is a pointer to the stack structure in the ADT, which is declared as a type definition. We name this stack type STACK. This design is very similar to C's FILE structure.

To create a stack, the programmer defines a pointer to a stack as shown in the following example and then calls the create stack function. The address of the stack structure is returned by create stack and assigned to the pointer in the calling function.

```
STACK* stack;
…
stack = createStack ();
```

The stack ADT structure is found in Figure 3-12.

ADT Implementation

It takes more than a data structure to make an abstract data type: there must also be operations that support the stack. We develop the C functions in the sections that follow.

Stack Structure

The stack abstract data type structure is shown in Program 3-6. The node structure consists only of a data pointer and a link node pointer. The stack head structure also contains only two elements, a pointer to the top of the stack and a count of the number of entries currently in the stack.

3. The array implementation for a stack can be found in Appendix F.

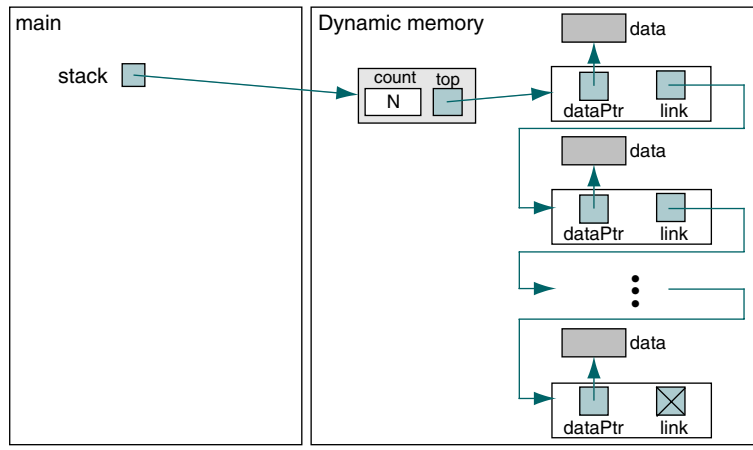


FIGURE 3-12 Stack ADT Structural Concepts

The type definition for the stack, as shown in Program 3-6, is included in a header file so that any function that needs to define a stack can do so easily.

PROGRAM 3-6 Stack ADT Definitions

```
1  // Stack ADT Type Defintions
2     typedef struct node
3        {
4         void*        dataPtr;
5         struct node* link;
6        } STACK_NODE;
7
8     typedef struct
9        {
10        int          count;
11        STACK_NODE* top;
12       } STACK;
```

Create Stack

Create stack allocates a stack head node, initializes the top pointer to null, and zeros the count field. The address of the node in the dynamic memory is then returned to the caller. The call to create a stack must assign the return pointer value to a stack pointer as shown below.

```
stack = createStack ( );
```

Figure 3-8 graphically shows the results of creating a stack. The code for the create stack function is shown in Program 3-7.

PROGRAM 3-7 ADT Create Stack

```
1  /* ============== createStack ==============
2     This algorithm creates an empty stack.
3        Pre  Nothing
4        Post Returns pointer to a null stack
5                -or- NULL if overflow
6  */
7  STACK* createStack (void)
8  {
9  // Local Definitions
10    STACK* stack;
11
12 // Statements
13    stack = (STACK*) malloc( sizeof (STACK));
14    if (stack)
15       {
16        stack->count = 0;
17        stack->top   = NULL;
18       } // if
19    return stack;
20 }  // createStack
```

Push Stack

The first thing that we need to do when we push data into a stack is to find a place for the data. This requires that we allocate memory from the heap using *malloc*. Once the memory is allocated, we simply assign the data pointer to the node and then set the link to point to the node currently indicated as the stack top. We also need to add one to the stack count field. Figure 3-8 shows several pushes into the stack. Program 3-8 is an implementation of push stack.

PROGRAM 3-8 Push Stack

```
1  /* ================ pushStack ================
2     This function pushes an item onto the stack.
3        Pre     stack is a pointer to the stack
4                dataPtr pointer to data to be inserted
5        Post    Data inserted into stack
6        Return  true  if successful
7                false if underflow
8  */
9  bool pushStack (STACK* stack, void* dataInPtr)
10 {
11 // Local Definitions
12    STACK_NODE* newPtr;
13
14 // Statements
15    newPtr = (STACK_NODE* ) malloc(sizeof( STACK_NODE));
16    if (!newPtr)
```

continued

PROGRAM 3-8 Push Stack *(continued)*

```
17          return false;
18
19       newPtr->dataPtr = dataInPtr;
20
21       newPtr->link    = stack->top;
22       stack->top      = newPtr;
23
24       (stack->count)++;
25       return true;
26    }  // pushStack
```

Program 3-8 Analysis

The ADT implementation of push differs from a traditional implementation only in its generality. Because it is an abstract data type, we need to handle any type of data pointer; therefore, we use a *void* pointer in the stack node. Recall that a *void* pointer can be used to store any type of pointer, and, conversely, it can be assigned to any pointer type. About the only thing we can't do with a *void* pointer is dereference it. In the stack, however, there is no need to refer to the actual data, so we never need to dereference it. As you study the rest of the stack algorithms, note the use of *void* pointers.

The last point also addresses generic code. Because we are writing an ADT, we cannot assume that heap overflow is an error. That decision belongs to the application. All we can do is report status: either the push was successful or it wasn't. If the allocation is successful, which is the normal case, we report that the push was successful by returning true. If the heap is full, we report that the push failed by returning false. It is then the calling function's responsibility to detect and respond to an overflow.

Pop Stack

Pop stack returns the data in the node at the top of the stack. It then deletes and recycles the node. After the count is adjusted by subtracting 1, the function returns to the caller. Note the way underflow is reported. In statement 15 we set the data pointer to NULL. If the stack is empty, when we return the data pointer in statement 24 we return NULL. Refer to Figure 3-8 for a graphic example of a stack pop in a linked list environment. The code is developed in Program 3-9.

PROGRAM 3-9 ADT Pop Stack

```
 1  /* ================== popStack ==================
 2     This function pops item on the top of the stack.
 3        Pre  stack is pointer to a stack
 4        Post Returns pointer to user data if successful
 5                     NULL if underflow
 6  */
 7  void* popStack (STACK* stack)
 8  {
 9  // Local Definitions
10     void*       dataOutPtr;
```

continued

PROGRAM 3-9 ADT Pop Stack *(continued)*

```
11      STACK_NODE* temp;
12
13  // Statements
14      if (stack->count == 0)
15          dataOutPtr = NULL;
16      else
17         {
18          temp       = stack->top;
19          dataOutPtr = stack->top->dataPtr;
20          stack->top = stack->top->link;
21          free (temp);
22          (stack->count)--;
23         } // else
24      return dataOutPtr;
25  }  // popStack
```

Program 3-9 Analysis

The code for the pop is rather straightforward. Two local pointers are required, one for the data pointer to be returned to the caller, and one that is used to free the deleted node.

It is interesting to follow the logic when the last node is being deleted. In this case the result is an empty stack. No special logic is required: the empty stack is created automatically because the last node has a null pointer, which when assigned to top indicates that the stack is empty.

Because a null pointer is false and a pointer with an address is true, we don't need a separate success flag. Rather, we just return the pointer with the address allocated by the new function. If memory was allocated successfully, it contains an address, which is true. If the allocation failed, it contains a null pointer, which is false.

Study the code that updates the stack top carefully. Note that it takes two levels of indirection to access the link field of the top node. If these references are confusing, trace them using Figure 3-8.

Finally, when we delete a node from the stack, the pop stack releases its memory.

Stack Top

The stack top function returns the data at the top of the stack without deleting the top node. It allows the user to see what will be deleted when the stack is popped. The code is developed in Program 3-10.

PROGRAM 3-10 Retrieve Stack Top

```
1   /* ================== stackTop =================
2      Retrieves data from the top of stack without
3       changing the stack.
4         Pre  stack is a pointer to the stack
5         Post Returns data pointer if successful
6                      null pointer if stack empty
7   */
```

continued

PROGRAM 3-10 Retrieve Stack Top *(continued)*

```
 8  void* stackTop (STACK* stack)
 9  {
10  // Statements
11     if (stack->count == 0)
12         return NULL;
13     else
14         return stack->top->dataPtr;
15  }  // stackTop
```

Program 3-10 Analysis The logic is virtually identical to the pop stack except for the delete and recycle. The code, however, is significantly different. As with the pop, there is only one potential problem with the retrieve function—the stack may be empty. If the stack is empty, we return a null pointer.

Empty Stack

Because the calling function has no access to the data structure, it cannot determine if there are data in the stack without actually trying to retrieve them. We therefore provide empty stack, a function that simply reports that the stack has data or that it is empty. (See Program 3-11.)

PROGRAM 3-11 Empty Stack

```
 1  /* ================ emptyStack ================
 2     This function determines if a stack is empty.
 3        Pre  stack is pointer to a stack
 4        Post returns 1 if empty; 0 if data in stack
 5  */
 6  bool emptyStack (STACK* stack)
 7  {
 8  // Statements
 9     return (stack->count == 0);
10  }  // emptyStack
```

Full Stack

Full stack is one of the most complex of the supporting functions. There is no straightforward way to tell if the next memory allocation is going to succeed or fail. All we can do is try it. But by trying to make an allocation, we use up part of the heap. Therefore, after allocating space for a node, we immediately free it so that it will be there when the program requests memory. This logic is shown in Program 3-12.

PROGRAM 3-12 Full Stack

```
 1  /* ================= fullStack =================
 2     This function determines if a stack is full.
```

continued

PROGRAM 3-12 Full Stack *(continued)*

```
 3    Full is defined as heap full.
 4       Pre    stack is pointer to a stack head node
 5       Return true if heap full
 6              false if heap has room
 7 */
 8 bool fullStack (STACK* stack)
 9 {
10 // Local Definitions
11 STACK_NODE* temp;
12
13 // Statements
14    if ((temp =
15       (STACK_NODE*)malloc (sizeof(*(stack->top)))))
16       {
17        free (temp);
18        return false;
19       } // if
20
21    // malloc failed
22    return true;
23 }  // fullStack
```

Stack Count

Stack count returns the number of items in the stack. Because this count is stored in the stack head node, we do not need to traverse the stack to determine how many items are currently in it. We simply return the head count.

PROGRAM 3-13 Stack Count

```
 1 /* ================== stackCount =================
 2    Returns number of elements in stack.
 3       Pre  stack is a pointer to the stack
 4       Post count returned
 5 */
 6 int stackCount (STACK* stack)
 7 {
 8 // Statements
 9    return stack->count;
10 }  // stackCount
```

Destroy Stack

Destroy stack deletes the nodes in a stack and returns a null pointer. It is the user's responsibility to set the stack pointer in the calling area to NULL by assigning the return value to the local stack pointer. Because the stack is implemented as a dynamic data structure in the heap, the memory is also released for reuse. Figure 3-8 graphically shows the results of destroying a

stack. Note that the two stack nodes and the stack head are all freed. The code for `destroyStack` is identical to its implementation in Program 3-14.

PROGRAM 3-14 Destroy Stack

```
/* ================== destroyStack =================
   This function releases all nodes to the heap.
      Pre  A stack
      Post returns null pointer
*/
STACK* destroyStack (STACK* stack)
{
// Local Definitions
   STACK_NODE* temp;

// Statements
   if (stack)
      {
       // Delete all nodes in stack
       while (stack->top != NULL)
          {
           // Delete data entry
           free (stack->top->dataPtr);

           temp = stack->top;
           stack->top = stack->top->link;
           free (temp);
          } // while

       // Stack now empty. Destroy stack head node.
       free (stack);
      } // if stack
   return NULL;
}  // destroyStack
```

Program 3-14 Analysis

Note that we guard against a call to destroy a stack that does not exist. Only if there is a stack do we execute the *while* loop. In the loop we walk through the stack, first recycling the user's data nodes and then recycling the stack nodes. When the last node has been freed, the stack top becomes NULL and the loop terminates. After all data nodes have been deleted, we also free the head node and return a null stack pointer.

3.5 Stack Applications

Stack applications can be classified into four broad categories: reversing data, parsing data, postponing data usage, and backtracking steps. For each of these applications we provide one or two examples. For reversing data we reverse a list. We also use reversing data to convert a decimal number to its binary equivalent. For parsing we show how to match the parentheses in a

source program. We use postponement to convert infix to postfix notation and also to evaluate a postfix expression. Finally, we use backtracking to choose between two or more paths. Each of these applications uses the stack ADT described in the previous section. You should make sure that you fully understand these algorithms before studying the following applications.

Four common stack applications are: reversing data, parsing, postponing, and backtracking.

Reversing Data

Reversing data requires that a given set of data be reordered so that the first and last elements are exchanged, with all of the positions between the first and last being relatively exchanged also. For example, {1 2 3 4} becomes {4 3 2 1}. We examine two different reversing applications: reverse a list and convert decimal to binary.

Reverse a List

One of the applications of a stack is to reverse a list of items. For example, Program 3-15 reads a list of integers and prints them in reverse.

PROGRAM 3-15 Reverse a Number Series

```
/* This program reverses a list of integers read
   from the keyboard by pushing them into a stack
   and retrieving them one by one.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdbool.h>
#include "stacksADT.h"

int main (void)
{
// Local Definitions
   bool done = false;
   int* dataPtr;

   STACK* stack;

// Statements
   // Create a stack and allocate memory for data
   stack = createStack ();

   // Fill stack
   while (!done)
      {
```

continued

PROGRAM 3-15 Reverse a Number Series *(continued)*

```
26          dataPtr = (int*) malloc (sizeof(int));
27          printf ("Enter a number: <EOF> to stop: ");
28          if ((scanf ("%d" , dataPtr)) == EOF
29                  || fullStack (stack))
30             done = true;
31          else
32             pushStack (stack, dataPtr);
33         } // while
34
35   // Now print numbers in reverse
36      printf ("\n\nThe list of numbers reversed:\n");
37      while (!emptyStack (stack))
38         {
39          dataPtr = (int*)popStack (stack);
40          printf ("%3d\n", *dataPtr);
41          free (dataPtr);
42         } // while
43
44   // Destroying Stack
45      destroyStack (stack);
46      return 0;
47   }  // main
```

```
Results:
Enter a number: <EOF> to stop: 3
Enter a number: <EOF> to stop: 5
Enter a number: <EOF> to stop: 7
Enter a number: <EOF> to stop: 16
Enter a number: <EOF> to stop: 91
Enter a number: <EOF> to stop:

The list of numbers reversed:
 91
 16
  7
  5
  3
```

Program 3-15 Analysis

This program is very simple. After creating a stack, it reads a series of numbers and pushes them into the stack. When the user keys end of file, the program then pops the stack and prints the numbers in reverse order.

The important point to note in this simple program is that we never referenced the stack structure directly. All stack references were through the stack ADT interface. This is an important concept in the structured programming principles of encapsulation and function reusability.

Figure 3-13 follows the program as it executes. As you study Program 3-15, use Figure 3-13 to trace its operations. Make sure that you have the same results at each different point in the figure.

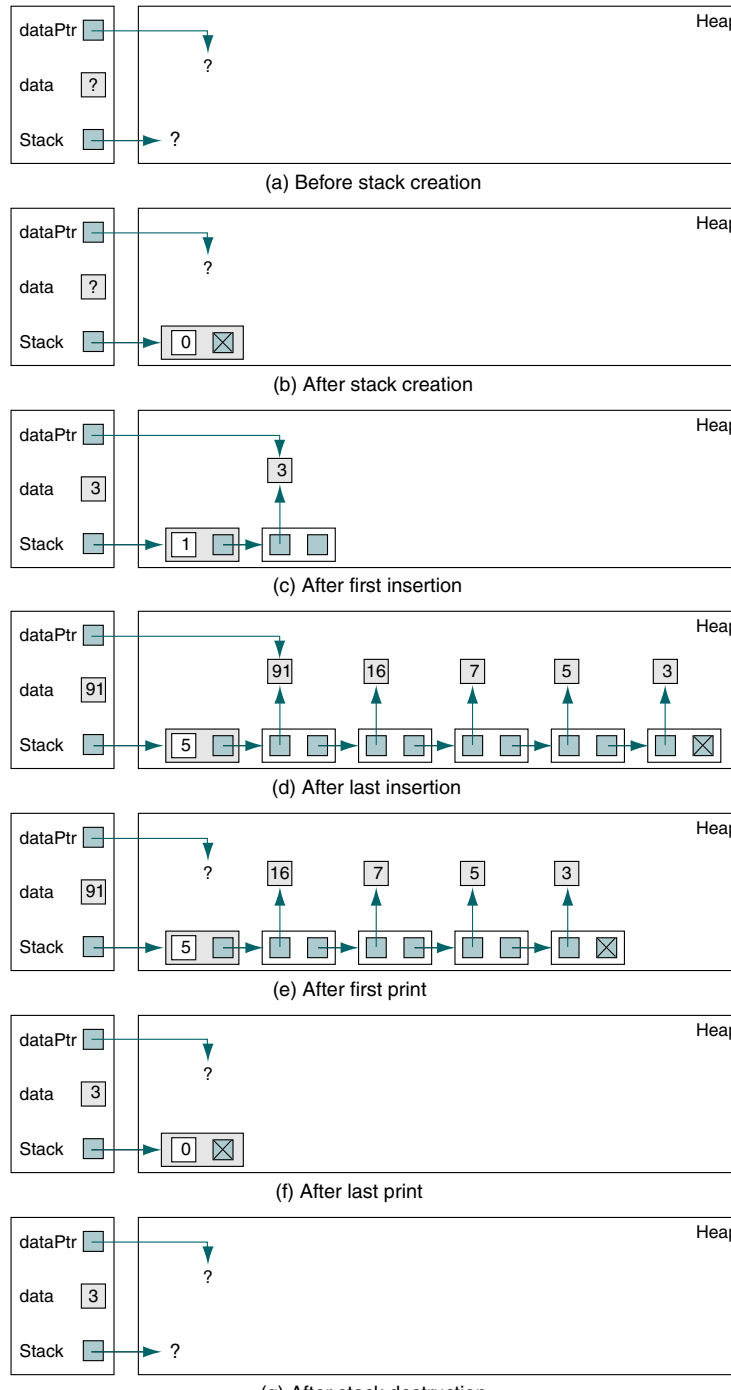


FIGURE 3-13 Program 3-15 Step-by-Step Execution

Convert Decimal to Binary

The idea of reversing a series can be used in solving classical problems such as transforming a decimal number to a binary number. The following simple code segment transforms a decimal number into a binary number:

```
1  read (number)
2  loop (number > 0)
   1  set digit to number modulo 2
   2  print (digit)
   3  set number to quotient of number / 2
3  end loop
```

This code has a problem, however. It creates the binary number backward. Thus, 19 becomes `11001` rather than `10011`.

We can solve this problem by using a stack. Instead of printing the binary digit as soon as it is produced, we push it into the stack. Then, after the number has been completely converted, we simply pop the stack and print the results one digit at a time in a line. This program is shown in Program 3-16.

PROGRAM 3-16 Convert Decimal to Binary

```
1  /* This program reads an integer from the keyboard
2     and prints its binary equivalent. It uses a stack
3     to reverse the order of 0s and 1s produced.
4        Written by:
5        Date:
6  */
7  #include <stdio.h>
8  #include "stacksADT.h"
9
10 int main (void)
11 {
12 // Local Definitions
13    unsigned int    num;
14             int*   digit;
15             STACK* stack;
16
17 // Statements
18    // Create Stack
19    stack = createStack ();
20
21    // prompt and read a number
22    printf ("Enter an integer:      ");
23    scanf ("%d", &num);
24
25    // create 0s and 1s and push them into the stack
26    while (num > 0)
```

continued

PROGRAM 3-16 Convert Decimal to Binary *(continued)*

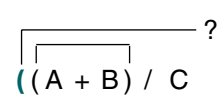
```
27          {
28           digit  = (int*) malloc (sizeof(int));
29           *digit = num % 2;
30           pushStack (stack, digit);
31           num = num /2;
32          } // while
33
34       // Binary number created. Now print it
35       printf ("The binary number is : ");
36       while (!emptyStack (stack))
37          {
38           digit = (int*)popStack (stack);
39           printf ("%1d", *digit);
40          } // while
41       printf ("\n");
42
43       // Destroying Stack
44       destroyStack (stack);
45       return 0;
46   }  // main
```

```
Results:
Enter an integer:      45
The binary number is : 101101
```

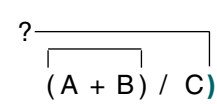
Parsing

Another application of stacks is parsing. **Parsing** is any logic that breaks data into independent pieces for further processing. For example, to translate a source program to machine language, a compiler must parse the program into individual parts such as keywords, names, and tokens.

One common programming problem is unmatched parentheses in an algebraic expression. When parentheses are unmatched, two types of errors can occur: the opening parenthesis can be missing or the closing parenthesis can be missing. These two errors are shown in Figure 3-14.



(a) Opening parenthesis not matched



(b) Closing parenthesis not matched

FIGURE 3-14 Unmatched Parentheses Examples

In Algorithm 3-9, we parse a source program to ensure that all of the parentheses are properly paired.

ALGORITHM 3-9 Parse Parentheses

```
Algorithm parseParens
This algorithm reads a source program and parses it to make
sure all opening-closing parentheses are paired.
 1  loop (more data)
    1   read (character)
    2   if (opening parenthesis)
        1   pushStack (stack, character)
    3   else
        1   if (closing parenthesis)
            1   if (emptyStack (stack))
                1   print (Error: Closing parenthesis not matched)
            2   else
                1   popStack(stack)
            3   end if
        2   end if
    4   end if
 2  end loop
 3  if (not emptyStack (stack))
    1   print (Error: Opening parenthesis not matched)
end parseParens
```

The implementation is shown in Program 3-17.

PROGRAM 3-17 Verify Parentheses Paired in Source Program

```
 1  /* This program reads a source program and parses it to
 2     make sure all opening-closing parentheses are paired.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7  #include "stacksADT.h"
 8
 9  // Error Messages
10  const char closMiss[] = "Close paren missing at line";
11  const char openMiss[] = "Open paren missing at line";
12
13  int main (void)
14  {
15  // Local Definitions
16     STACK* stack;
17     char   token;
18     char*  dataPtr;
```

continued

PROGRAM 3-17 Verify Parentheses Paired in Source Program *(continued)*

```
19      char   fileID[25];
20      FILE*  fpIn;
21      int    lineCount = 1;
22
23  // Statements
24      // Create Stack
25      stack = createStack ();
26      printf ("Enter file ID for file to be parsed: ");
27      scanf  ("%s", fileID);
28
29      fpIn = fopen (fileID, "r");
30      if (!fpIn)
31          printf("Error opening %s\n", fileID), exit(100);
32
33      // read characters from the source code and parse
34      while ((token = fgetc (fpIn)) != EOF )
35         {
36          if (token == '\n')
37             lineCount++;
38          if (token == '(' )
39             {
40              dataPtr = (char*) malloc (sizeof (char));
41              pushStack (stack, dataPtr);
42             } // if
43          else
44             {
45              if (token == ')')
46                 {
47                  if (emptyStack (stack))
48                     {
49                      printf ("%s %d\n",
50                              openMiss, lineCount);
51                      return 1;
52                     } // if true
53                  else
54                     popStack (stack);
55                 } // token ==
56             } // else
57         } // while
58
59      if (!emptyStack (stack))
60         {
61          printf ("%s %d\n", closMiss, lineCount);
62          return 1;
63         } // if
64
65      // Now destroy the stack
66      destroyStack (stack);
```

continued

PROGRAM 3-17 Verify Parentheses Paired in Source Program *(continued)*

```
67      printf ("Parsing is OK: %d Lines parsed.\n",
68                lineCount);
69      return 0;
70  }  // main
```

```
Results:
Run 1:
   Enter file ID for file to be parsed: no-errors.txt
   Parsing is OK: 65 Lines parsed.
Run 2:
   Enter file ID for file to be parsed: close-match.txt
   Close paren missing at line 46
Run 3:
   Enter file ID for file to be parsed: open-match.txt
   Open paren missing at line 23
```

Program 3-17 Analysis

Whenever we find an opening parenthesis in a program, we push it into the stack. When we find a closing parenthesis, we pop its matching opening parenthesis from the stack. Note that there are two different pairing errors in the program: (1) a closing parenthesis without a matching opening parenthesis (see statement 49) and (2) an opening parenthesis without a matching closing parenthesis (see statement 61).

To test the program, we ran it three times against three different files. While this testing demonstrates all three situations, it does not fully test the program. Additional test situations, such as missing first opening parenthesis, must also be run.

Finally, note how we placed the error messages at the beginning of the program. This is a common documentation technique that not only provides good documentation but makes the program easier to read.

Postponement

When we used a stack to reverse a list, the entire list was read before we began outputting the results. Often the logic of an application requires that the usage of data be deferred until some later point. A stack can be useful when the application requires that the use of data be postponed for a while. We develop two stack postponement applications in this section: infix to postfix transformation and postfix expression evaluation.

Infix to Postfix Transformation

One of the disadvantages of the infix notation[4] is that we need to use parentheses to control the evaluation of the operators. We thus have an evaluation method that includes parentheses and two operator priority classes. In the postfix and prefix notations, we do not need parentheses; each provides only one evaluation rule.

Although some high-level languages use infix notation, such expressions cannot be directly evaluated. Rather, they must be analyzed to determine the

4. For a discussion of arithmetic notations, see "Prefix to Postfix Conversion" in Chapter 2.

order in which the expressions are to be evaluated. A common evaluation technique is to convert the expressions to postfix notation before generating the code to evaluate them. We demonstrate this concept here. We first examine a manual method for converting infix to postfix expressions and then develop an algorithm that can be implemented on a computer.

Manual Transformation

The rules for manually converting infix to postfix expressions are as follows:

1. Fully parenthesize the expression using any explicit parentheses and the arithmetic precedence—multiply and divide before add and subtract.
2. Change all infix notations in each parenthesis to postfix notation, starting from the innermost expressions. Conversion to postfix notation is done by moving the operator to the location of the expression's closing parenthesis.
3. Remove all parentheses.

For example, for the following infix expression:

```
A + B * C
```

Step 1 results in

```
( A + ( B * C ) )
```

Step 2 moves the multiply operator after C

```
( A + ( B C * ) )
```

and then moves the addition operator to between the last two closing parentheses. This change is made because the closing parenthesis for the plus sign is the last parenthesis. We now have

```
( A (B C * ) +)
```

Finally, step 3 removes the parentheses.

```
A B C * +
```

Let's look at a more complex example. This example is not only longer but it already has one set of parentheses to override the default evaluation order.

```
( A + B ) * C  + D + E * F - G
```

Step 1 adds parentheses.

```
( ( ( ( ( A + B ) * C ) + D ) + ( E * F ) ) - G )
```

Step 2 then moves the operators.

```
( ( ( ( ( A B + ) C * ) D + ) ( E F * ) + ) G - )
```

Step 3 removes the parentheses.

```
A B + C * D + E F * + G -
```

Algorithmic Transformation

This manual operation would be difficult to implement in a computer. Let's look at another technique that is easily implemented with a stack.

We start with a very simple example, transforming a multiplication operation. The multiplication of two variables is shown below, first in infix notation and then in postfix notation.

```
A * B    converts to    A B *
```

We can obviously read the operands and output them in order. The problem becomes how to handle the multiply operator; we need to postpone putting it in the output until we have read the right operand, `B`. In this simple case, we push the operator into a stack and, after the whole infix expression has been read, pop the stack and put the operator in the postfix expression.

Now let's look at a more complex expression. Again, the infix expression is given on the left and the equivalent postfix expression on the right.

```
A * B + C    converts to    A B * C +
```

Here again we can read the infix operators and copy them to the postfix expression. If we were to simply put the operators into the stack as we did earlier and then pop them to the postfix expression after all of the operands had been read, we would get the wrong answer. Somehow we must pair the two operators with their correct operands. One possible rule might be to postpone an operator only until we get another operator. Then, before we push the second operator, we could pop the first one and place it in the output expression. This logic works in this case, but it won't for others. Consider the following example.

```
A + B * C    converts to    A B C * +
```

As we discussed previously, infix expressions use a precedence rule to determine how to group the operands and operators in an expression. We can use the same rule when we convert infix to postfix. When we need to push an operator into the stack, if its priority is higher than the operator at the top of the stack, we go ahead and push it into the stack. Conversely, if the operator at the top of the stack has a higher priority than the current operator, it is popped and placed in the output expression. Using this rule with the above expression, we would take the following actions:

1. Copy operand `A` to output expression.
2. Push operator + into stack.
3. Copy operand `B` to output expression.
4. Push operator `*` into stack. (Priority of `*` is higher than +.)
5. Copy operand `C` to output expression.

6. Pop operator * and copy to output expression.
7. Pop operator + and copy to output expression.

We need to cover one more rule to complete the logic. When a current operator with a lower or equal priority forces the top operator to be popped from the stack, we must check the new top operator. If it is also greater than the current operator, it is popped to the output expression. Consequently, we may pop several operators to the output expression before pushing the new operator into the stack.

Let's work on one more example before we formally develop the algorithm.

```
A + B * C - D / E    converts to    A B C * + D E / -
```

The transformation of this expression is shown in Figure 3-15. Because it uses all of the basic arithmetic operators, it is a complete test.

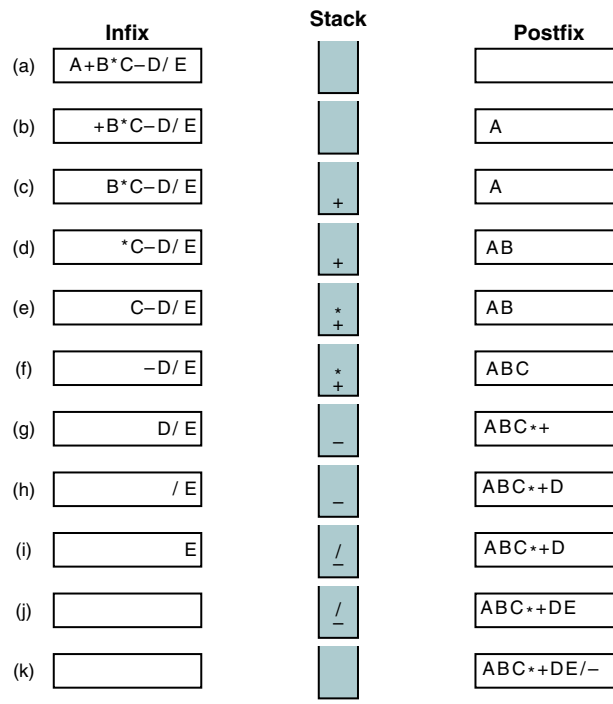


FIGURE 3-15 Infix Transformations

We begin by copying the first operand, `A`, to the postfix expression. See Figure 3-15(b). The add operator is then pushed into the stack and the second operand is copied to the postfix expression. See Figure 3-15(d). At this point we are ready to insert the multiply operator into the stack. As we see in

Figure 3-15(e), its priority is higher than that of the add operator at the top of the stack, so we simply push it into the stack. After copying the next operand, C, to the postfix expression, we need to push the minus operator into the stack. Because its priority is lower than that of the multiply operator, however, we must first pop the multiply and copy it to the postfix expression. The plus sign is now popped and appended to the postfix expression because the minus and plus have the same priority. The minus is then pushed into the stack. The result is shown in Figure 3-15(g). After copying the operand D to the postfix expression, we push the divide operator into the stack because it is of higher priority than the minus at the top of the stack in Figure 3-15(i). After copying E to the postfix expression, we are left with an empty infix expression and two operators in the stack. See Figure 3-15(j). All that is left at this point is to pop the stack and copy each operator to the postfix expression. The final expression is shown in Figure 3-15(k).

We are now ready to develop the algorithm. We assume only the operators shown below. They have been adapted from the standard algebraic notation.

```
Priority 2:    *  /
Priority 1:    +  -
Priority 0:    (
```

The design is shown in Algorithm 3-10.

ALGORITHM 3-10 Convert Infix to Postfix

```
Algorithm inToPostFix (formula)
Convert infix formula to postfix.
   Pre    formula is infix notation that has been edited
          to ensure that there are no syntactical errors
   Post   postfix formula has been formatted as a string
   Return postfix formula
 1 createStack (stack)
 2 loop (for each character in formula)
   1  if (character is open parenthesis)
      1  pushStack (stack, character)
   2  elseif (character is close parenthesis)
      1  popStack (stack, character)
      2  loop (character not open parenthesis)
         1  concatenate character to postFixExpr
         2  popStack (stack, character)
      3  end loop
   3  elseif (character is operator)
         Test priority of token to token at top of stack
      1  stackTop (stack, topToken)
      2  loop (not emptyStack (stack)
           AND priority(character) <= priority(topToken))
         1  popStack (stack, tokenOut)
         2  concatenate tokenOut to postFixExpr
         3  stackTop (stack, topToken)
```

continued

ALGORITHM 3-10 Convert Infix to Postfix *(continued)*

```
      3  end loop
      4  pushStack (stack, token)
   4  else
         Character is operand
      1 Concatenate token to postFixExpr
   5  end if
3  end loop
   Input formula empty. Pop stack to postFix
4  loop (not emptyStack (stack))
   1  popStack (stack, character)
   2  concatenate token to postFixExpr
5  end loop
6  return postFix
end inToPostFix
```

The code follows in Program 3-18.

PROGRAM 3-18 Convert Infix to Postfix

```
/* This program converts an infix formula to a postfix
   formula. The infix formula has been edited to ensure
   that there are no syntactical errors.
       Written by:
       Date:
*/
#include <stdio.h>
#include <string.h>
#include "stacksADT.h"

// Prototype Declarations
int  priority (char token);
bool isOperator (char token);

int main (void)
{
// Local Definitions
   char   postfix [80] = {0};
   char   temp [2] = {0};
   char   token;
   char*  dataPtr;
   STACK* stack;

// Statements
   // Create Stack
   stack = createStack ();

   // read infix formula and parse char by char
   printf("Enter an infix formula: ");
```

continued

PROGRAM 3-18 Convert Infix to Postfix *(continued)*

```
30
31    while ((token = getchar ())!= '\n')
32       {
33        if (token == '(')
34           {
35            dataPtr  = (char*) malloc (sizeof(char));
36            *dataPtr = token;
37            pushStack (stack, dataPtr);
38           } // if
39        else if (token == ')')
40           {
41            dataPtr = (char*)popStack (stack);
42            while (*dataPtr != '(')
43               {
44                temp [0]= *dataPtr;
45                strcat (postfix , temp);
46                dataPtr = (char*)popStack (stack);
47               } // while
48           } // else if
49        else if (isOperator (token))
50           {
51            // test priority of token at stack top
52            dataPtr = (char*)stackTop (stack);
53            while (!emptyStack (stack)
54              && priority (token) <= priority (*dataPtr))
55                {
56                 dataPtr  = (char*)popStack (stack);
57                 temp [0] = *dataPtr;
58                 strcat (postfix , temp);
59                 dataPtr  = (char*)stackTop (stack);
60                } // while
61            dataPtr  = (char*) malloc (sizeof (char));
62            *dataPtr = token;
63            pushStack (stack , dataPtr);
64           } // else if
65        else                    // character is operand
66           {
67            temp[0]= token;
68            strcat (postfix , temp);
69           } // else
70       } // while get token
71
72    // Infix formula empty. Pop stack to postfix
73    while (!emptyStack (stack))
74       {
75        dataPtr = (char*)popStack (stack);
76        temp[0] = *dataPtr;
77        strcat (postfix , temp);
```

continued

PROGRAM 3-18 Convert Infix to Postfix *(continued)*

```
78          } // while
79
80       // Print the postfix
81       printf ("The postfix formula is: ");
82       puts (postfix);
83
84       // Now destroy the stack
85       destroyStack (stack);
86       return 0;
87    }  // main
88    /* ================== priority ====================
89       Determine priority of operator.
90          Pre  token is a valid operator
91          Post token priority returned
92    */
93    int priority (char token)
94    {
95    // Statements
96       if (token == '*' || token == '/')
97           return 2;
98       if (token == '+' || token == '-')
99           return 1;
100      return 0;
101   }  // priority
102   /* ================== isOperator ====================
103      Determine if token is an operator.
104         Pre  token is a valid operator
105         Post return true if operator; false if not
106   */
107   bool isOperator (char token)
108   {
109   // Statements
110      if (token == '*'
111          || token == '/'
112          || token == '+'
113          || token == '-')
114           return true;
115      return false;
116   }  // isOperator
```

```
Results:
Run 1
   Enter an infix formula: 2+4
   The postfix formula is: 24+
Run 2
   Enter an infix formula: (a+b)*(c-d)/e
   The postfix formula is: ab+cd-*e/
```

Program 3-18 Analysis An infix expression can contain only three objects: a parenthetical set of operators, a variable identifier or a constant, and an operator. If we have an opening parenthesis,

we put it in the stack. It is eventually paired with a closing parenthesis that signals the end of an expression or a subexpression. Variables and constants go immediately to the output string; thus only operators require more analysis.

Given an operator to be processed, the question is: Should it be pushed into the stack or placed in the output formula? If the stack is empty, the operator is pushed into the stack and we continue.

Infix operators have two priorities. The multiply and divide operators have a higher priority (2) than the add and subtract operators (1). Because the opening parenthesis is also pushed into the operator stack, we give it a priority of 0. We check the priority of the new current token against the priority of the token at the top of the stack at statement 54.

If the new operator's priority is lower than or equal to that of the operator at the top of the stack, the token at the top of the stack is moved to the output string and we loop to recheck the operator at the top of the stack. If the new operator has a higher priority than the operator at the top of the stack, it goes into the stack and we move to the next token in the input formula.

Evaluating Postfix Expressions

Now let's see how we can use stack postponement to evaluate the postfix expressions we developed earlier. For example, given the expression shown below,

```
A B C + *
```

and assuming that `A` is 2, `B` is 4, and `C` is 6, what is the expression value?

The first thing you should notice is that the operands come before the operators. This means that we will have to postpone the use of the operands this time, not the operators. We therefore put them into the stack. When we find an operator, we pop the two operands at the top of the stack and perform the operation. We then push the value back into the stack to be used later. Figure 3-16 traces the operation of our expression. (Note that we push the operand values into the stack, not the operand names. We therefore use the values in the figure.)

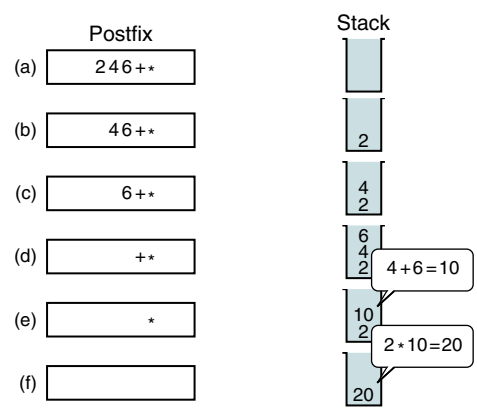


FIGURE 3-16 Evaluation of Postfix Expression

When the expression has been completely evaluated, its value is in the stack. See Algorithm 3-11. The implementation is shown in Program 3-19.

ALGORITHM 3-11 Evaluation of Postfix Expressions

```
Algorithm postFixEvaluate (expr)
This algorithm evaluates a postfix expression and returns its
value.
   Pre    a valid expression
   Post   postfix value computed
   Return value of expression
 1 createStack (stack)
 2 loop (for each character)
   1  if (character is operand)
      1  pushStack (stack, character)
   2  else
      1  popStack (stack, oper2)
      2  popStack (stack, oper1)
      3  operator = character
      4  set value to calculate (oper1, operator, oper2)
      5  pushStack (stack, value)
   3  end if
 3 end loop
 4 popStack (stack, result)
 5 return (result)
end postFixEvaluate
```

PROGRAM 3-19 Evaluate Postfix Expression

```
 1  /* This program evaluates a postfix expression and
 2     returns its value. The postfix expression must be
 3     valid with each operand being only one digit.
 4        Written by:
 5        Date:
 6  */
 7  #include <stdio.h>
 8  #include <stdlib.h>
 9  #include "stacksADT.h"
10
11  // Prototype Declarations
12  bool isOperator (char token);
13  int  calc       (int operand1, int oper, int operand2);
14
15  int main (void)
16  {
17  // Local Definitions
18     char   token;
19     int    operand1;
20     int    operand2;
21     int    value;
```

continued

PROGRAM 3-19 Evaluate Postfix Expression *(continued)*

```
22       int*   dataPtr;
23       STACK* stack;
24
25    // Statements
26       // Create Stack
27       stack = createStack ();
28
29       // read postfix expression, char by char
30       printf("Input formula: ");
31       while ((token = getchar ())!= '\n')
32          {
33           if (!isOperator (token))
34              {
35               dataPtr  = (int*) malloc (sizeof (int));
36               *dataPtr = atoi (&token);
37               pushStack (stack, dataPtr);
38              } // while
39
40           else
41              // character is operand
42              {
43               dataPtr  = (int*)popStack (stack);
44               operand2 = *dataPtr;
45               dataPtr  = (int*)popStack (stack);
46               operand1 = *dataPtr;
47               value    = calc(operand1, token, operand2);
48               dataPtr  = (int*) malloc (sizeof (int));
49               *dataPtr = value;
50               pushStack (stack, dataPtr);
51              } // else
52          } // while
53
54       // The final result is in stack. Pop it print it
55       dataPtr = (int*)popStack (stack);
56       value = *dataPtr;
57       printf ("The result is: %d\n", value);
58
59
60       // Now destroy the stack
61       destroyStack (stack);
62       return 0;
63    }  // main
64    /* ==================== isOperator ==================
65       Validate operator.
66          Pre  token is operator to be validated
67          Post return true if valid, false if not
68    */
69    bool isOperator (char token)
```

continued

PROGRAM 3-19 Evaluate Postfix Expression *(continued)*

```
{
// Statements
   if (token == '*'
       || token == '/'
       || token == '+'
       || token == '-')
      return true;
   return false;
}  // isOperator
/* ================== calc ==================
   Given two values and operator, determine value of
   formula.
      Pre  operand1 and operand2 are values
           oper is the operator to be used
      Post return result of calculation
*/
int calc (int operand1, int oper, int operand2)
{
// Local Declaration
   int result;

// Statements
   switch (oper)
      {
       case '+' : result  = operand1 + operand2;
                  break;
       case '-' : result  = operand1 - operand2;
                  break;
       case '*' : result  = operand1 * operand2;
                  break;
       case '/' : result  = operand1 / operand2;
                  break;
      } // switch
   return result;
}  // calc
```

```
Results:
Input formula: 52/4+5*2+
The result is 32
```

Program 3-19 Analysis

We have omitted all of the error handling in this algorithm. We assume that the expression is valid and that there is sufficient memory to evaluate it. An implementation obviously needs to guard against these problems.

Given these assumptions, the algorithm is relatively simple. We create a stack and then push operands into the stack until we find an operator. As we find each operator, we pop the two top operands and perform the operation. The result is then pushed into the stack and the next element of the expression is examined. At the end, we pop the value from the stack and return it.

Backtracking

Backtracking is another stack use found in applications such as computer gaming, decision analysis, and expert systems. We examine two backtracking applications in this section: goal seeking and the eight queens problem.

Goal Seeking

Figure 3-17 is an example of a goal-seeking application. One way to portray the problem is to lay out the steps in the form of a graph that contains several alternate paths. Only one of the paths in the figure leads to a desired goal. Whereas we can immediately see the correct path when we look at the figure, the computer needs an algorithm to determine the correct path.

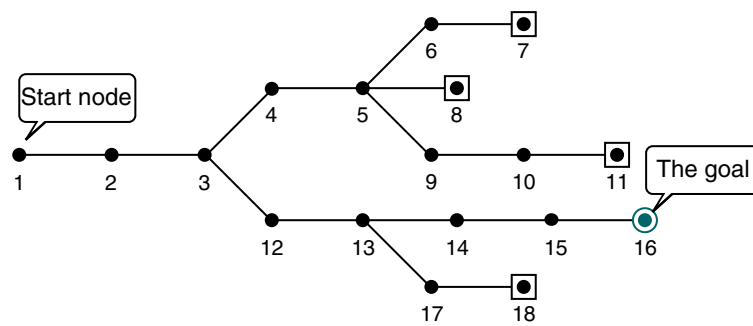


FIGURE 3-17 Backtracking Example

We start at node 1 and move right until we hit a branching node, 3. At this point we take the upper path. We continue until we get to node 5, at which time we again take the upper path. When we arrive at node 7, we can go no further. However, we have not yet reached our goal. We must therefore backtrack to node 5 and take the next path. At node 8 we again backtrack to node 5 and take the third path. Once again, at node 11, we must backtrack, this time way back to node 3. As we follow the path from node 12 we finally arrive at node 16, our goal. Note that we did not examine the path that contains nodes 17 and 18.

By now you should have begun to formulate an algorithm in your head. Every time we get to a decision point, we need to remember where it is so that we can get back to it if necessary. When we backtrack we want to go back to the nearest point before we continue; that is, we don't want to start at the beginning again. For this problem we use the LIFO data structure, the stack.

The question now is what to put into the stack. If we only needed to locate the node that contains the goal, we would just put the branch point nodes into the stack. However, when we are finished we want to print out the path that leads to our goal. Therefore, we must also put the nodes in the valid

path into the stack. Because we are putting two things into the stack, we need to be able to tell them apart. We do that with a flag. If the node is in the path, we push a path token. If we are storing a backtracking point, we set the flag to a backtracking token.

Figure 3-18 contains our stack tracing of the path to our goal in Figure 3-17. We use B to indicate that we have stored a backtracking node. If there is no token, the stack contains a path node.

We start by pushing 1, 2, and 3 into the stack. Because 3 is not our goal, we follow the upper path. However, we need to remember that we made a decision here. Therefore, we push the branch point, 12, into a stack with a backtracking token, as shown in Figure 3-18(a). At 5 we must make another decision. Again, we follow the upper path. This time we have two continue points, 8 and 9, that we need to remember. So, we push both into the stack with backtracking tokens. This point is shown in Figure 3-18(b). At 7 we have reached the end of the path without finding our goal. See Figure 3-15(c). To continue we pop the stack until we are at a backtracking point. We then push the backtracking point into the stack as a path node and move on toward 8. In our diagram we immediately hit another dead end, as shown in Figure 3-18(d). After backtracking to 9, we continue until we get to the dead end at node 11. See Figure 3-18(e). Once again we backtrack, this time to node 12. Following the path to 13, we push a decision node into the stack (B17) and continue on until we find our goal, at node 16.

We now know the path to our goal, we only need to print it out. The print loop simply pops the stack and prints the path. We ignore any backtracking points left in the stack, such as B17 in Figure 3-18(f).

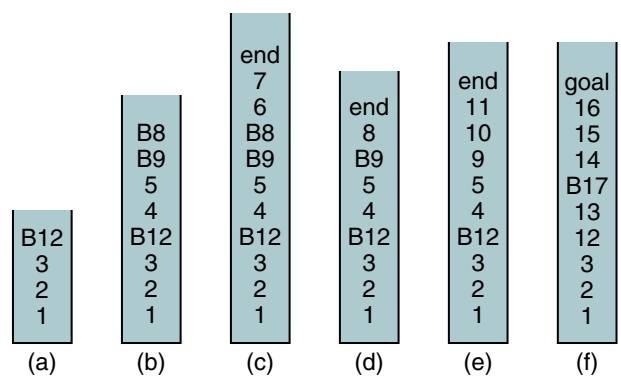


FIGURE 3-18 Backtracking Stack Operation

We present the solution in algorithmic form. Until we study the creation and storage of graphs in Chapter 12, we cannot develop the C code. The pseudocode solution is found in Algorithm 3-12.

ALGORITHM 3-12 Print Path to Goal

```
Algorithm seekGoal (map)
This algorithm determines the path to a desired goal.
   Pre  a graph containing the path
   Post path printed
 1 createStack (stack)
 2 set pMap to starting point
 3 loop (pMap not null AND goalNotFound)
   1  if (pMap is goal)
      1  set goalNotFound to false
   2  else
      1  pushStack (stack, pMap)
      2  if (pMap is a branch point)
         1  loop (more branch points)
            1  create branchPoint node
            2  pushStack (stack, branchPoint)
         2  end loop
      3  end if
      4  advance to next node
   3  end if
 4 end loop
 5 if (emptyStack (stack))
   1  print (There is no path to your goal)
 6 else
   1  print (The path to your goal is:)
   2  loop (not emptyStack (stack))
      1  popStack (stack, pMap)
      2  if (pMap not branchPoint)
         1  print(map point)
      3  end if
   3  end loop
   4  print (End of Path)
 7 end if
 8 return
end seekGoal
```

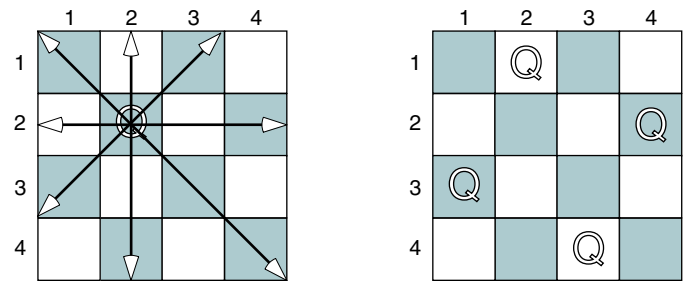
Algorithm 3-12 Analysis

This algorithm demonstrates one of the values of pseudocode and why it is popular when discussing algorithms with users. You will not have enough data structure knowledge until Chapter 11 (graphs) to solve this problem on a computer. Yet even though we don't have a data structure to solve it, we can still develop and understand the algorithm.

Eight Queens Problem

A classic chess problem requires that you place eight queens on the chessboard in such a way that no queen can capture another queen. There are actually several different solutions to this problem. The computer solution to the problem requires that we place a queen on the board and then analyze all of the attack positions to see if there is a queen that could capture the new queen. If there is, then we try another position.[5]

To demonstrate a sample solution, let's analyze how we would place four queens on a 4 × 4 chessboard. The queen's capture rules and one solution are shown in Figure 3-19.



(a) Queen capture rules (b) First four queens solution

FIGURE 3-19 Four Queens Solution

We can solve this problem using a stack and backtracking logic. Because only one queen can be placed in any row, we begin by placing the first queen in row 1, column 1. This location is then pushed into a stack, giving the position shown in Figure 3-20, step 1.

After placing a queen in the first row, we look for a position in the second row. Position 2,1 is not possible because the queen in the first row is guarding this location on the vertical. Likewise, position 2,2 is guarded on the diagonal. We therefore place a queen in the third column in row 2 and push this location into the stack. This position is shown in Figure 3-20, step 2.

We now try to locate a position in row 3, but none are possible. The first column is guarded by the queen in row 1, and the other three positions are guarded by the queen in row 2. At this point we must backtrack to the second row by popping the stack and continue looking for a position for the second-row queen. Because column 4 is not guarded, we place a queen there and push its location into the stack (step 3 in Figure 3-20).

5. A queen can attack another queen if it is in the same row, in the same column, or in the same diagonal.

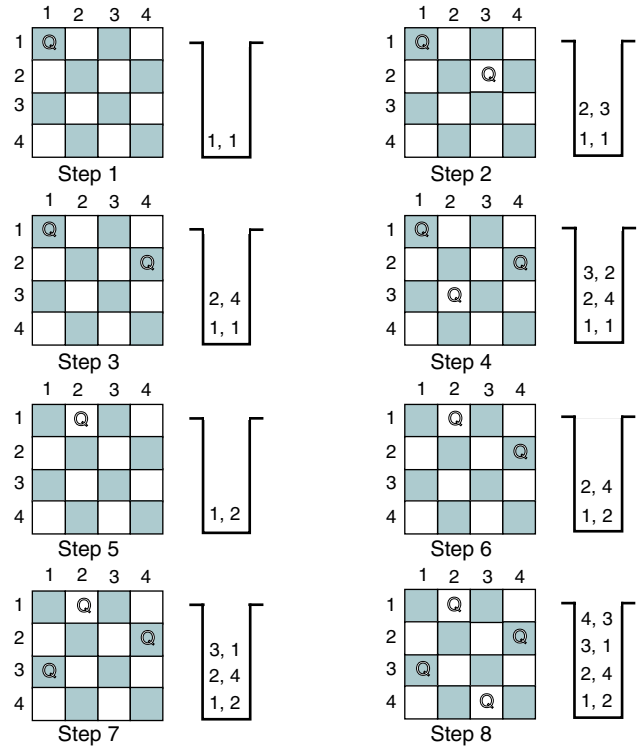


FIGURE 3-20 Four Queens Step-by-Step Solution

Looking again at row 3, we see that the first column is still guarded by the queen in row 1 but that we can place a queen in the second column. We do so and push this location into the stack (step 4 in Figure 3-20).

When we try to place a queen in row 4, however, we find that all positions are guarded. Column 1 is guarded by the queen in row 1 and the queen in row 3. Column 2 is guarded by the queen in row 2 and the queen in row 3. Column 3 is guarded by the queen in row 3, and column 4 is guarded by both the queen in row 1 and the queen in row 2. We therefore backtrack to the queen in row 3 and try to find another place for her. Because the queen in row 2 is guarding both column 3 and column 4, there is no position for a queen in row 3. Once again we backtrack by popping the stack and find that the queen in row 2 has nowhere else to go, so we backtrack to the queen in row 1 and move her to column 2. This position is shown in Figure 3-20, step 5.

Analyzing row 2, we see that the only possible position for a queen is column 4 because the queen in row 1 is guarding the first three positions. We therefore place the queen in this location and push the location into the stack (step 6 in Figure 3-20).

Column 1 in the third row is unguarded, so we place a queen there (step 7 in Figure 3-20). Moving to row 4, we find that the first two positions are guarded, the first by the queen in row 3 and the second by all three queens. The third column is unguarded, however, so we can place the fourth queen in this column for a solution to the problem.

Generalizing the solution, we see that we place a queen in a position in a row and then examine all positions in the next row to see if a queen can be placed there. If we can't place a queen in the next row, we backtrack to the last-positioned queen and try to position her in the next column. If there is no room in the next column, we fall back again. Given that there is a solution,[6] this trial-and-error method works well. A pseudocode solution using our stack abstract data type is shown in Algorithm 3-13.

ALGORITHM 3-13 Eight Queens Problem

```
Algorithm queens8 (boardSize)
Position chess queens on a game board so that no queen can
capture any other queen.
   Pre  boardSize is number of rows & columns on board
   Post queens' positions printed
1  createStack (stack)
2  set row to 1
3  set col to 0
4  loop (row <= boardSize)
   1   loop (col <= boardSize AND row <= boardSize)
       1   increment col
       2   if (not guarded (row, col))
           1   place queen at row-col intersection on board
           2   pushStack (row-col into stack)
           3   increment row
           4   set col to 0
       3   end if
           At end of row. Back up to previous position.
       4   loop (col >= boardSize)
           1   popStack (row-col from stack)
           2   remove queen from row-col intersection on board
       5   end loop
   2   end loop
5  end loop
6  printBoard (stack)
7  return
end queens8
```

6. There are no solutions for boards less than 4 x 4 positions. All boards from 4 x 4 to 8 x 8 have at least one solution.

We now develop a C program to implement the eight queens problem. The program begins by requesting the board size from the user and then creating a stack to store the queens' positions. After creating the stack, the main line calls a function that fills the board with the maximum number of queens for the board size. It then prints the results. The design is shown in Figure 3-21.

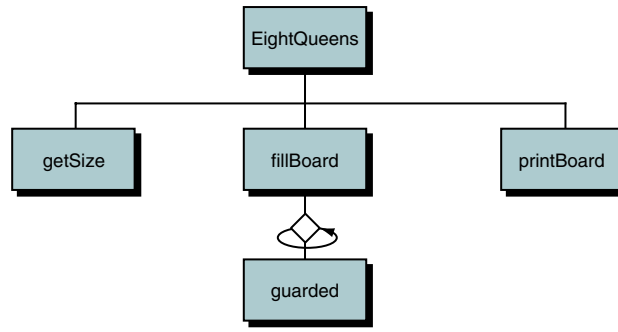


FIGURE 3-21 Design for Eight Queens

As shown in the design, only one other function, `guarded`, is needed. It is called as each position surrounding the potential placement of a new queen is examined.

When designing a program, it is often necessary to resolve differences between the technical and user perspectives of the problem. In the eight queens program, this difference is seen in the terminology used to place the queens on the board. Users number the rows and columns starting at 1; C numbers them starting with 0. To resolve this difference, we added a dummy row and column to the board array, creating a 9×9 board. We then ignore row 0 and column 0 in the processing.

The global declarations and main line code are shown in Program 3-20.

PROGRAM 3-20 Eight Queens Mainline

```
/* This program tests the eight queens algorithm. Eight
   queens is a classic chess problem in which eight
   queens are placed on a chess board in positions
   such that no queen can capture another queen.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include "P4StkADT.h"

// Structure Declarations
```

continued

PROGRAM 3-20 Eight Queens Mainline *(continued)*

```
13    typedef struct
14       {
15        int  row;
16        int  col;
17       } POSITION;
18
19 // Prototype Declarations
20 int  getSize   (void);
21
22 void fillBoard  (STACK* stack, int boardSize);
23 void printBoard (STACK* stack, int boardSize);
24
25 bool guarded (int board[][9], int row,
26               int col,        int boardSize);
27
28 int main (void)
29 {
30 // Local Definitions
31    int  boardSize;
32
33    STACK* stack ;
34
35 // Statements
36    boardSize = getSize ();
37    stack     = createStack ();
38
39    fillBoard    (stack, boardSize);
40    printBoard   (stack, boardSize);
41    destroyStack (stack);
42
43    printf("\nWe hope you enjoyed Eight Queens.\n");
44    return 0;
45 }  // main
```

The function to read the board size from the keyboard is shown in Program 3-21.

PROGRAM 3-21 Eight Queens: Get Board Size

```
1  /* ==================== getSize =====================
2     Prompt user for a valid board size.
3        Pre  nothing
4        Post valid board size returned
5  */
6  int getSize (void)
7  {
```

continued

PROGRAM 3-21 Eight Queens: Get Board Size *(continued)*

```
// Local Definitions
   int boardSize;

// Statements
   printf("Welcome to Eight Queens. You may select\n"
          "a board size from 4 x 4 to 8 x 8. I will\n"
          "then position a queen in each row of the\n"
          "board so no queen may capture another\n"
          "queen. Note: There are no solutions for \n"
          "boards less than 4 x 4.\n");
   printf("\nPlease enter the board size: ");
   scanf ("%d", &boardSize);
   while (boardSize < 4 || boardSize > 8)
      {
       printf("Board size must be greater than 3 \n"
              "and less than 9. You entered %d.\n"
              "Please re-enter. Thank you.\a\a\n\n"
              "Your board size: ", boardSize);
       scanf ("%d", &boardSize);
      } // while
   return boardSize;
}  // getSize
```

The longest function in the program is `fillBoard`. It loops using a trial-and-error concept to place the maximum number of queens on the board. Each possible position in a row is tested by calling `guarded`. If the position is good, its location is placed in the stack and the next row is analyzed. If it is not good, the stack is popped and we back up to try another position in the previous row. When all of the rows have a queen, the problem is solved. The code is shown in Program 3-22.

PROGRAM 3-22 Eight Queens: Fill Board

```
/* ================== fillBoard ====================
   Position chess queens on game board so that no queen
   can capture any other queen.
      Pre  boardSize number of rows & columns on board
      Post Queens' positions filled
*/
void fillBoard (STACK* stack, int boardSize)
{
// Local Definitions
   int  row;
   int  col;
   int  board[9][9] = {0};  // 0 no queen: 1 queen
                            // row 0 & col 0 !used
```

continued

PROGRAM 3-22 Eight Queens: Fill Board *(continued)*

```
14      POSITION* pPos;
15
16  // Statements
17      row = 1;
18      col = 0;
19
20      while (row <= boardSize)
21        {
22         while (col <= boardSize && row <= boardSize)
23           {
24            col++;
25            if (!guarded(board, row, col, boardSize))
26               {
27                board[row][col] = 1;
28
29               pPos = (POSITION*)malloc(sizeof(POSITION));
30                pPos->row = row;
31                pPos->col = col;
32
33                pushStack(stack, pPos);
34
35                row++;
36                col = 0;
37               } // if
38             while (col >= boardSize)
39               {
40                pPos = popStack(stack);
41                row  = pPos->row;
42                col  = pPos->col;
43                board[row][col] = 0;
44                free (pPos);
45               } // while col
46           } // while col
47        } // while row
48      return;
49  }  // fillBoard
```

Program 3-22 Analysis

In this backtracking problem, we need to back up to the row and column that contains the last queen we placed on the board. In statements 30 and 31, we save the row and column in a structure and then pass the position to **`pushStack`** in statement 33.

Examine the *while* in statement 38 carefully. It is used to backtrack when no positions in a row are found. We must backtrack when the column becomes greater than or equal to the board size, indicating that we have examined all of the positions in the row and found none that satisfies the requirements. (If we had found an unguarded position, we would have advanced the row in statement 35.)

The logic for `guarded` is shown in Program 3-23. It uses a brute force, trial-and-error method to determine whether the new queen is safe. Note that we only need to test for queens attacking from above. There can be no

queens below the current row. If the queen is not safe, we return true—her current position is guarded. If she is safe, we return false.

PROGRAM 3-23 Eight Queens: Guarded

```
/* ===================== guarded ==================
   Checks rows, columns, diagonals for guarding queens

      Pre  board contains current positions for queens
           chkRow & chkCol are position for new queen
           boardSize is number of rows & cols in board
      Post returns true if guarded; false if not
*/
bool guarded (int board[][9], int chkRow,
              int chkCol,     int boardSize)
{
// Local Definitions
   int row;
   int col;

// Statements

   // Check current col for a queen
   col = chkCol;
   for (row = 1; row <= chkRow; row++)
       if (board[row][col] == 1)
          return true;

   // Check diagonal right-up
   for (row = chkRow - 1, col = chkCol + 1;
        row > 0 && col <= boardSize;
        row--, col++)
       if (board[row][col] == 1)
          return true;

   // Check diagonal left-up
   for (row = chkRow - 1, col = chkCol - 1;
        row > 0 && col > 0;
        row--, col--)
       if (board[row][col] == 1)
          return true;

   return false;
}  // guarded
```

The `printBoard` function simply prints out the positioning of the queens on the board. A sample output is shown at the end of Program 3-24.

PROGRAM 3-24 Eight Queens: Print Board

```
/* ================== printBoard ====================
   Print positions of chess queens on a game board
   Pre  stack contains positions of queen
        boardSize is the number of rows and columns
   Post Queens' positions printed
*/
void printBoard (STACK* stack, int boardSize)
{
// Local Definitions
   int col;

   POSITION* pPos;
   STACK*    pOutStack;

// Statements
   if (emptyStack(stack))
       {
       printf("There are no positions on this board\n");
        return;
       } // if

   printf("\nPlace queens in following positions:\n");

   // Reverse stack for printing
   pOutStack = createStack ();
   while (!emptyStack (stack))
      {
       pPos = popStack (stack);
       pushStack (pOutStack, pPos);
      } // while

   // Now print board
   while (!emptyStack (pOutStack))
      {
       pPos = popStack (pOutStack);
       printf("Row %d-Col %d: \t|",
               pPos->row, pPos->col);
       for (col = 1; col <= boardSize; col++)
          {
           if (pPos->col == col)
               printf(" Q |");
           else
               printf("   |");
          } // for
       printf("\n");
      } // while
```

continued

PROGRAM 3-24 Eight Queens: Print Board *(continued)*

```
47      destroyStack(pOutStack);
48      return;
49  }   // printBoard
```

```
Results:
Welcome to Eight Queens. You may select
a board size from 4 x 4 to 8 x 8. I will
then position a queen in each row of the
board so no queen may capture another
queen. Note: There are no solutions for
boards less than 4 x 4.

Please enter the board size: 4

Place queens in following positions:
Row 1-Col 2:     | Q |   |   |
Row 2-Col 4:     |   |   | Q |
Row 3-Col 1:   Q |   |   |   |
Row 4-Col 3:     |   | Q |   |

We hope you enjoyed Eight Queens.
```

3.6 How Recursion Works

In Chapter 2 we discussed recursion but did not explain how it physically works. Now that we've discussed stacks, we can discuss the physical operation of recursion.

To understand how recursion works, we need to first explore how any call works. When a program calls a subroutine—for example, a function in C—the current module suspends processing and the called subroutine takes over control of the program. When the subroutine completes its processing and returns to the module that called it, the module wakes up and continues its processing. One important point in this interaction is that, unless changed through call by reference, all local data in the calling module are unchanged. Every local variable must be in the same state when processing resumes as it was when processing suspended. Similarly, the parameter list must not be changed. The value of the parameters must be the same before and after a call, unless they are reference parameters.

Let's look at an example of a simple call and see how it works. Figure 3-22 contains an algorithm called `testPower` that prints the value of a number `x` raised to a power `y`. To illustrate the function call, we use a separate algorithm, `power`, to determine the actual value of x^y.

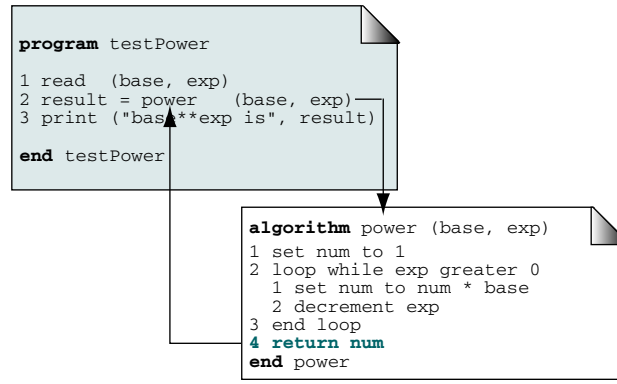


FIGURE 3-22 Call and Return

Our little program contains three local variables: the base number, `base`; the exponent, `exp`; and the answer, `result`. The base and exponent values must not be changed by the call to `power`.[7] When `power` terminates, it returns the value of x^y, which is stored in the local variable, `result`.

When `power` begins to execute, it must know the values of the parameters so that it can process them. It must also know where it needs to return when its processing is done. Finally, because `power` returns a value, it must know where the value is to be placed when it terminates. The physical implementation is determined by the compiler writer, but these data are conceptually placed in a **stackframe**. When `power` is called, the stackframe is created and pushed into a system stack. When it concludes, the stackframe is popped, the local variables are replaced, the return value is stored, and processing resumes in the calling algorithm.

A stackframe contains four different elements:

1. The parameters to be processed by the called algorithm
2. The local variables in the calling algorithm
3. The return statement in the calling algorithm
4. The expression that is to receive the return value (if any)

Now that you understand how a call works, you are ready to see how recursion works. Algorithm 3-14 contains a recursive version of the power algorithm discussed earlier.

7. In this discussion, we assume a recursive compiler. Languages such as COBOL and FORTRAN operate under a different, nonrecursive design in which there are no local variables.

ALGORITHM 3-14 Recursive Power Algorithm

```
Algorithm power (base, exp)
This algorithm computes the value of a number, base, raised
to the power of an exponent, exp.
   Pre    base is the number to be raised
          exp is the exponent
   Post   value of base raised to power exp computed
   Return value of base raised to power exp returned
 1 if (exp is 0)
   1   return (1)
 2 else
   1   return (base * power (base, exp - 1))
 3 end if
end power
```

In Figure 2-4, we traced the calls for the recursive factorial algorithm. This example is logically correct, but it oversimplifies the calls. Figure 3-23 traces the execution of Algorithm 3-14 in its recursive version and shows the contents of the system stack for each call.

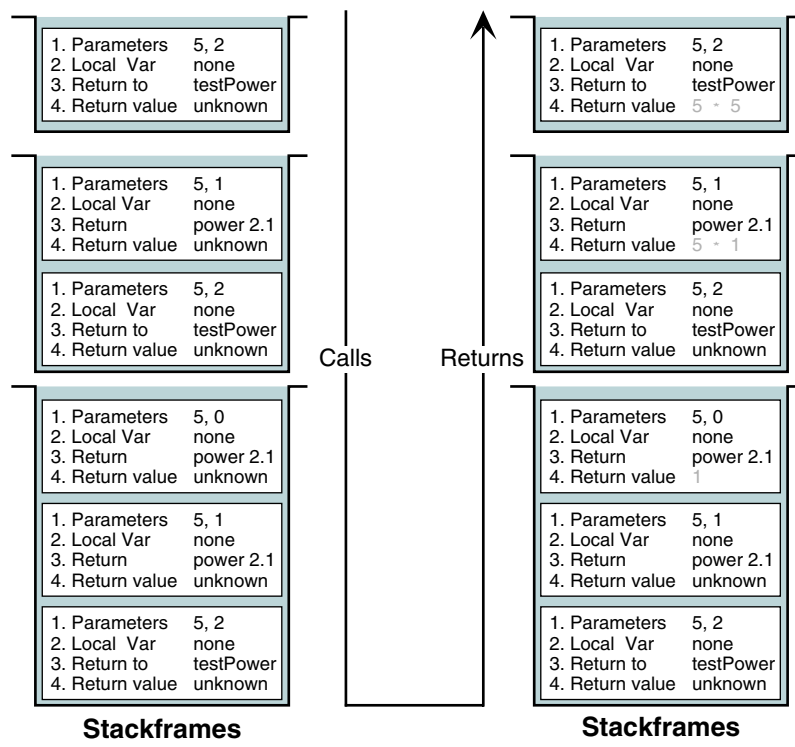


FIGURE 3-23 Stackframes for Power (5, 2)

Referring to Figure 3-23, `power` is called initially by `testPower` with the base and exponent set to 5 and 2, respectively. The first stackframe was created by the original call and contains a return location in the calling algorithm, `testPower`. In the second call, the base is unchanged but the exponent becomes 1. This call creates the second stackframe in Figure 3-23. Note that the parameters are changed and the return address is now `power` 2.1. In the third iteration, the exponent becomes 0. This call is shown in the third stackframe.

At this point we have reached the base case and begin backing out of the recursion. As each return statement is executed, we return to statement 2.1 in `power`, first with the value of 1 (the base case), then with a value of 5 (5×1), and finally with a value of 25 (5×5). When we return the third time, we return to the original calling algorithm, `testPower`.

3.7 Key Terms

backtracking
create stack
data node
destroy stack
empty stack
full stack
head
last in–first out (LIFO)
metadata
overflow
parsing
pop
pop stack
push
push stack
self-referential data structure
stack
stack count
stackframe
stack top
top
underflow

3.8 Summary

- A stack is a linear list in which all additions are restricted to one end, called the top. A stack is also called a LIFO list.
- We have defined eight operations for a stack: create stack, push stack, pop stack, stack top, empty stack, full stack, stack count, and destroy stack.
- Create stack initializes the stack metadata.
- Push stack adds an item to the top of the stack. After the push, the new item becomes the top.
- Each push must ensure that there is room for the new item. If there is no room, the stack is in an overflow state.
- Pop stack removes the item at the top of the stack. After the pop the next item, if any, becomes the top. Each pop must ensure that there is at least one item in the stack. If there is not at least one item, the stack is in an underflow state.
- The stack top operation only retrieves the item at the top of the stack. Each top execution must ensure that there is at least one item in the stack. If there is not at least one item, the stack is in an underflow state.
- Empty stack determines whether the stack is empty and returns a Boolean true if it is.
- Full stack determines whether there is room for at least one more item and returns a Boolean false if there is.
- Stack count returns the number of elements currently in the stack.

- Destroy stack releases all allocated data memory to dynamic memory.
- One of the applications of a stack is reversing data. The nature of a stack (last in–first out) allows us to push items into a stack and pop them in reverse order.
- Stacks are commonly used to parse data, postpone the use of data, and backtrack steps in a path.
- When a module calls a subroutine recursively, in each call all of the information needed by the subroutine is pushed in the stackframe. The information is popped in the reverse order when subroutines are terminated one after another, until finally control is returned to the initial calling module.
- A stack is used to facilitate recursive calls. When a program calls a subroutine, the system creates a stackframe to store:
  1. The value of the parameters
  2. The value of the local variables
  3. The return address in the calling module
  4. The return value

3.9 Practice Sets

Exercises

1. Imagine we have two empty stacks of integers, `s1` and `s2`. Draw a picture of each stack after the following operations:

```
1 pushStack  (s1, 3)
2 pushStack  (s1, 5)
3 pushStack  (s1, 7)
4 pushStack  (s1, 9)
5 pushStack  (s1, 11)
6 pushStack  (s1, 13)
7 loop not emptyStack (s1)
    1  popStack  (s1, x)
    2  pushStack (s2, x)
8 end loop
```

2. Imagine we have two empty stacks of integers, `s1` and `s2`. Draw a picture of each stack after the following operations:

```
1 pushStack (s1, 3)
2 pushStack (s1, 5)
3 pushStack (s1, 7)
4 pushStack (s1, 9)
5 pushStack (s1, 11)
6 pushStack (s1, 13)
7 loop not emptyStack (s1)
    1  popStack  (s1, x)
    2  popStack  (s1, x)
    3  pushStack (s2, x)
8 end loop
```

3. Using manual transformation, write the following infix expressions in their postfix and prefix forms:

   a. D – B + C
   b. A * B + C * D
   c. (A + B) * C – D * F + C
   d. (A – 2 * (B + C) – D * E) * F

4. Using manual transformation, change the following postfix or prefix expressions to infix:

   a. A B * C – D +
   b. A B C + * D –
   c. + – * A B C D
   d. – * A + B C D

5. If the values of A, B, C, and D are 2, 3, 4, and 5, respectively, manually calculate the value of the following postfix expressions:

   a. A B * C – D +
   b. A B C + * D –

6. If the values of A, B, C, and D are 2, 3, 4, and 5, respectively, manually calculate the value of the following prefix expressions:

   a. + – * A B C D
   b. – * A + B C D

7. Change the following infix expressions to postfix expressions using the algorithmic method (a stack):

   a. D – B + C
   b. A * B + C * D
   c. (A + B) * C – D * F + C
   d. (A – 2 * (B + C) – D * E) * F

8. Determine the value of the following postfix expressions when the variables have the following values: A is 2, B is 3, C is 4, and D is 5.

   a. A B C D * – +
   b. D C * B A + –

Problems

9. Write a program to implement Algorithm 3-15, "Reverse a Number Series." Test your program with the number series 1, 3, 5, 7, 9, 2, 4, 6, 8.
10. Write the pseudocode for an algorithm that prints a decimal number as an octal number.
11. Write a program that changes a decimal number to a hexadecimal number. (*Hint:* If the remainder is 10, 11, 12, 13, 14, or 15, print A, B, C, D, E, or F, respectively.)
12. Write a program to implement Algorithm 3-9, "Parse Parentheses," matching braces rather than parentheses. In your implementation, push the line number into the stack rather than the opening brace. When an error occurs, print the line number for the unmatched opening brace or unmatched closing brace. Test your program by running the source code through itself (there should be no errors) and then test it with the following small program:

```
Test brace errors.
} line 2 closing brace is not paired
No braces.
   {opening brace is paired on same line}
No braces.
   {opening brace paired later
    No braces.
   } Closing brace paired two lines up.
{{{ Line 9. Three braces--only two paired.
  } First closing brace
 } Second closing brace.
End of program. One opening brace left.
```

13. Write a program that implements the infix-to-postfix notation (see Algorithm 3-10, "Convert Infix to Postfix"). The program should read an infix string consisting of single alphabetic characters for variables, parentheses, and the +, –, *, and / operators; call the conversion algorithm; and then print the resulting postfix expression. After transforming an algorithm, it should loop and convert another infix string. To test your program, transform the expressions in Exercise 3 with your program.

14. Change Problem 13 to allow multicharacter variable identifiers and numeric constants as shown in the following expression:

```
num + 18 * factor
```

15. Write a program to implement the postfix evaluation (see Algorithm 3-11, "Evaluation of Postfix Expressions"). The program should read a postfix string consisting of only multidigit numeric data and the +, –, *, and / operators; call the evaluation algorithm; and then print the result. After each evaluation it should loop and process another evaluation. Evaluate the following expressions with your program:

```
25 7 * 14 - 6 +
1 24 3 + * 41 -
2 37 4 + * 15 -
```

16. One of the applications of a stack is to backtrack—that is, to retrace its steps. As an example, imagine we want to read a list of items, and each time we read a negative number we must backtrack and print the five numbers that come before the negative number and then discard the negative number.

    Use a stack to solve this problem. Read the numbers and push them into the stack (without printing them) until a negative number is read. At this time, stop reading and pop five items from the stack and print them. If there are fewer than five items in the stack, print an error message and stop the program.

    After printing the five items, resume reading data and placing them in the stack. When the end of the file is detected, print a message and the items remaining in the stack.

    Test your program with the following data:

    `1 2 3 4 5 -1 1 2 3 4 5 6 7 8 9 10 -2 11 12 -3 1 2 3 4 5`

17. Write the pseudocode for an algorithm called `copyStack` that copies the contents of one stack into another. The algorithm passes two stacks, the source stack and the destination stack. The order of the stacks must be identical. (*Hint:* Use a temporary stack to preserve the order.)

18. Write a new ADT function, `catStack`, that concatenates the contents of one stack on top of another. Test your function by writing a program that uses the ADT to create two stacks and prints them. It should then concatenate the stacks and print the resulting stack.

19. A palindrome is a string that can be read backward and forward with the same result. For example, the following is a palindrome:

    Able was I ere I saw Elba.

Write a function to test if a string is a palindrome using a stack. You can push characters in the stack one by one. When you reach the end of the string, you can pop the characters and form a new string. If the two strings are exactly the same, the string is a palindrome. Note that palindromes ignore spacing, punctuation, and capitalization. Test your program with the following test cases:

Go dog
Madam, I'm Adam
Madam, I'm not a palindrome

20. Write a program that reads a text file, one line at a time, and prints the line as it was read and then prints the line with its text reversed. Print a blank line after each reversed line.

21. Write the pseudocode for an algorithm that reverses the contents of a stack (the top and bottom elements exchange positions, the second and the element just before the bottom exchange positions, and so forth until the entire stack is reversed). (*Hint:* Use temporary stacks.)

22. Write a function to check whether the contents of two stacks are identical. Neither stack should be changed. You need to write a function that prints the contents of a stack to verify that your function works.

Projects

23. Given a square matrix, write a program that determines the number of white blocks and total number of squares in each of the white blocks. By definition the outside boundaries of the matrix must be shaded. A block of white squares consists of all of the white squares whose side boundaries are adjacent to another white square. White squares that touch only at a diagonal point are not adjacent.

    In Figure 3-24, we have numbered the white blocks. Block 1 contains three squares, and block 4 contains nine squares. Note that block 3 contains only one square. It touches block 1 only on the diagonal.

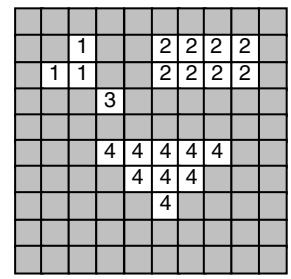


FIGURE 3-24 Project 23: Find White Blocks

Obtain the square definition from a text file. The file should contain a square matrix composed of zeros for shaded squares and nonzero values for white squares. The input for Figure 3-24 is shown below. The matrix should be allocated from dynamic memory.

```
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 2 2 2 2 0
0 1 1 0 0 2 2 2 2 0
0 0 0 3 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 4 4 4 4 4 0 0
0 0 0 0 4 4 4 0 0 0
0 0 0 0 0 4 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

At the end of the program, print a report showing the number of white blocks and the number of squares in each.

24. Write a program that simulates a mouse in a maze. The program must print the path taken by the mouse from the starting point to the final point, including all spots that have been visited and backtracked. Thus, if a spot is visited two times, it must be printed two times; if it is visited three times, it must be printed three times.

    The maze is shown in Figure 3-25. The entrance spot, where the mouse starts its journey, is chosen by the user who runs the program. It can be changed each time.

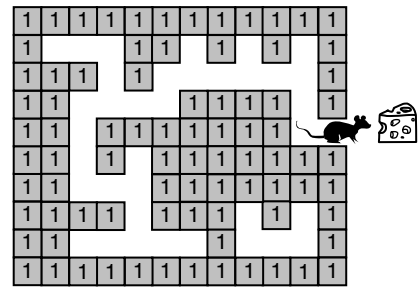


FIGURE 3-25 Mouse Maze for Project 24

A two-dimensional array can be used as a supporting data structure to store the maze. Each element of the array can be black or white. A black element is a square that the mouse cannot enter. A white element is a

square that can be used by the mouse. In the array a black element can be represented by a 1 and a white element by a 0.

When the mouse is traversing the maze, it visits the elements one by one. In other words, the mouse does not consider the maze as an array of elements; at each moment of its journey, it is only in one element. Let's call this element the `currentSpot`. It can be represented by a structure of two integer fields. The first field is the row and the second is the column coordinate of the spot in the maze. For example, the exit in Figure 3-25 is at (5,12)—that is, row 5, column 12.

The program begins by creating the maze. It then initializes the exit spot and prompts the user for the coordinates of the entrance spot. The program must be robust. If the user enters coordinates of a black spot, the program must request new coordinates until a white spot is entered. The mouse starts from the entrance spot and tries to reach the exit spot and its reward. Note, however, that some start positions do not lead to the exit.

As the mouse progresses through its journey, print its path. As it enters a spot, the program determines the class of that spot. The class of a spot can be one of the following:

a. Continuing—A spot is a continuing spot if one and only one of the neighbors (excluding the last spot) is a white spot. In other words, the mouse has only one choice.
b. Intersection—A spot is an intersection spot if two or more of the neighbors (excluding the last spot) is a white spot. In other words, the mouse has two or more choices.
c. Dead end—A spot is a dead-end spot if none of the neighbors (excluding the last spot) is a white spot. In other words, the mouse has no spot to choose. It must backtrack.
d. Exit—A spot is an exit spot if the mouse can get out of the maze. When the mouse finds an exit, it is free and receives a piece of cheese for a reward.

To solve this problem, you need two stacks. The first stack, the visited stack, contains the path the mouse is following. Whenever the mouse arrives at a spot, it first checks to see whether it is an exit. If not, its location is placed in the stack. This stack is used if the mouse hits a dead end and must backtrack. Whenever the mouse backtracks to the last decision point, also print the backtrack path.

When the mouse enters an intersection, the alternatives are placed in a second stack. This decision point is also marked by a special decision token that is placed in the visited stack. The decision token has coordinates of (–1,–1). To select a path, an alternative is then popped from the alternatives stack and the mouse continues on its path.

While backtracking, if the mouse hits a decision token, the token is discarded and the next alternative is selected from the alternatives stack. At this point print an asterisk (*) next to the location to show that the next alternative path is being selected.

If the mouse arrives at a dead end and both stacks are empty, the mouse is locked in a portion of the maze with no exit. In this case, print a trapped message and terminate the search for an exit.

After each trial, regardless of the outcome, the user should be given the opportunity to stop or continue.

25. Write a program that implements the stack ADT described in Section 3.4, "Stack ADT." To test your implementation, write a menu-driven user interface to test each of the operations in the ADT. For the test, use integer data as described below. Error conditions should be printed as a part of the test results. A suggested menu is shown below.

```
A. Push data into stack
B. Pop and print data
C. Print data at top of stack
D. Print entire stack (top to base)
E. Print stack status: Empty
F. Print stack status: Full
G. Print number of elements in stack
H. Destroy stack and quit
```

Test your program with the following test case. You may include additional test cases. The menu operation is shown at the end of the test.

a. Print stack status: Empty [E]
b. Pop and print data (should return error) [B]
c. Push data into stack: 1 [A]
d. Push data into stack: 2 [A]
e. Print stack status: Empty [E]
f. Print stack status: Full [F]
g. Print data at top of stack [C]
h. Print entire stack (top to base) [D]
i. Print number of elements in stack [G]
j. Pop and print data [B]
k. Pop and print data [B]
l. Pop and print data (should return empty) [B]
m. Push data into stack: 3 [A]
n. Print data at top of stack [C]
o. Destroy stack and quit [H]

26. Write a program to find all solutions to the eight queens problem. Your program will need to be able to handle a search for a configuration that has no solution.

27. Modify the user interface in Project 25 to manipulate two different stacks with two different types of data. The first stack is to contain integer data. The second stack is to contain alphabetic data. In addition to the test data in Project 25, create a set of alphabetic test data that tests the alphabetic stack with a similar set of operations.

Chapter 4
Queues

A queue is a linear list in which data can only be inserted at one end, called the **rear**, and deleted from the other end, called the **front**. These restrictions ensure that the data are processed through the queue in the order in which they are received. In other words, a queue is a **first in–first out (FIFO)** structure.

A queue is the same as a line. In fact, if you were in England, you would not get into a line, you would get into a queue. A line of people waiting for the bus at a bus station is a queue, a list of calls put on hold to be answered by a telephone operator is a queue, and a list of waiting jobs to be processed by a computer is a queue.

Figure 4-1 shows two representations of a queue: one a queue of people and the other a computer queue. Both people and data enter the queue at the rear and progress through the queue until they arrive at the front. Once they are at the front of the queue, they leave the queue and are served.

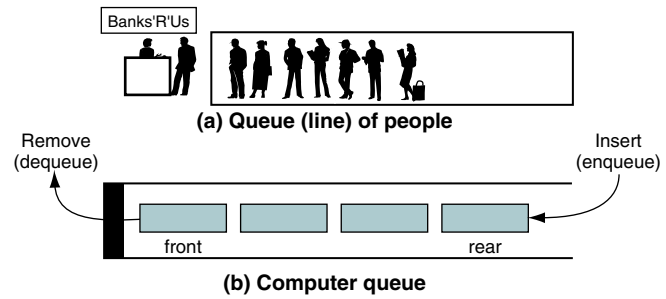


FIGURE 4-1 Queue Concept

4.1 Queue Operations

There are four basic queue operations. Data can be inserted at the rear, deleted from the front, retrieved from the front, and retrieved from the rear. Although there are many similarities between stacks and queues, one significant structural difference is that the queue implementation needs to keep track of the front and the rear of the queue, whereas the stack only needs to worry about one end: the top.

A queue is a linear list in which data can be inserted at one end, called the rear, and deleted from the other end, called the front. It is a first in–first out (FIFO) restricted data structure.

Enqueue

The queue insert operation is known as **enqueue**. After the data have been inserted into the queue, the new element becomes the rear. As we saw with stacks, the only potential problem with enqueue is running out of room for the data. If there is not enough room for another element in the queue, the queue is in an overflow state.

Enqueue inserts an element at the rear of the queue.

Figure 4-2 shows the enqueue operation.

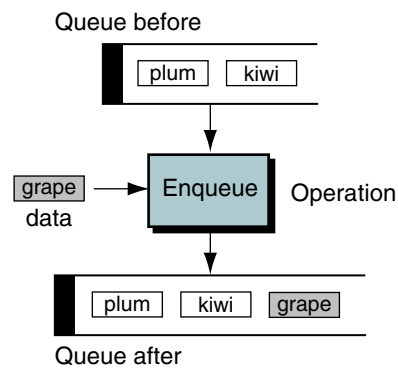


FIGURE 4-2 Enqueue

Dequeue

The queue delete operation is known as **dequeue**. The data at the front of the queue are returned to the user and removed from the queue. If there are no data in the queue when a dequeue is attempted, the queue is in an underflow state.

Dequeue deletes an element at the front of the queue.

The dequeue operation is shown in Figure 4-3.

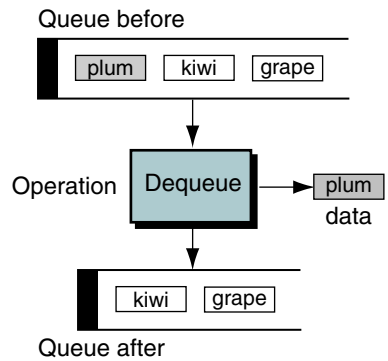


FIGURE 4-3 Dequeue

Queue Front

Data at the front of the queue can be retrieved with **queue front**. It returns the data at the front of the queue without changing the contents of the queue. If there are no data in the queue when a queue front is attempted, then the queue is in an underflow state.

Queue front retrieves the element at the front of the queue.

The queue front operation is shown in Figure 4-4.

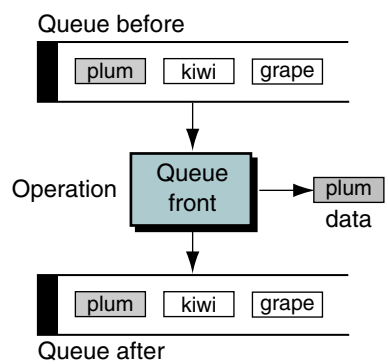


FIGURE 4-4 Queue Front

Queue Rear

A parallel operation to queue front retrieves the data at the rear of the queue. It is known as **queue rear.** As with queue front, if there are no data in the queue when a queue rear is attempted, the queue is in an underflow state.

Queue rear retrieves the element at the rear of the queue.

The queue rear operation is shown in Figure 4-5.

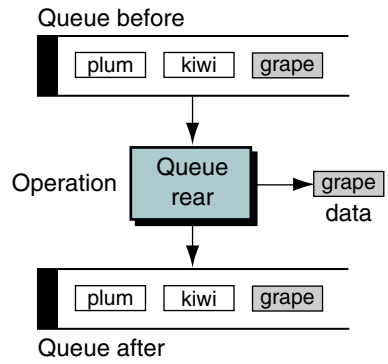


FIGURE 4-5 Queue Rear

Queue Example

Figure 4-6 traces these four operations in an example. We start with an empty queue and enqueue `green` and `blue`. At this point the queue contains two entries. We then dequeue, which removes the entry at the front of the queue, leaving `blue` as the only entry. After enqueuing `red` a queue front operation returns `blue` to the caller but leaves it at the front of the queue. The next operation, queue rear, returns `red` but leaves it in the queue. A dequeue then removes `blue`, leaving `red` as the only entry in the queue. Finally, we dequeue `red`, which results in an empty queue.

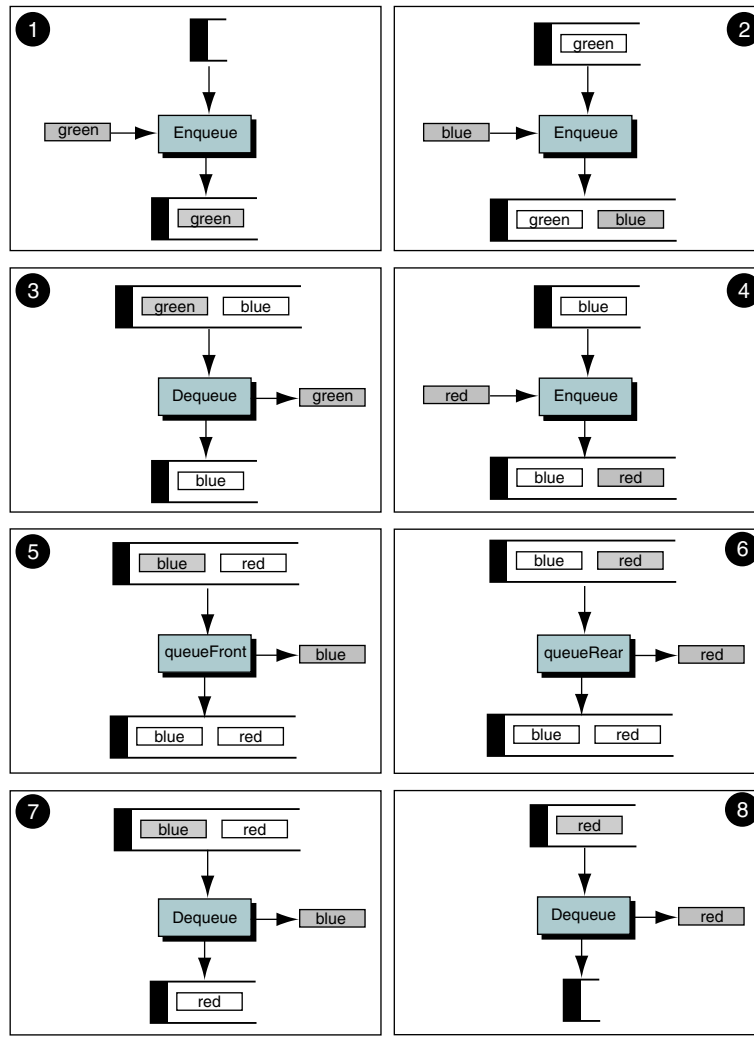


FIGURE 4-6 Queue Example

4.2 Queue Linked List Design

As with a stack, we implement our queue using a linked list. As we have emphasized, the actual implementation may be different. For example, in Appendix F we provide an array implementation for the queue ADT.

Data Structure

We need two different structures to implement the queue: a queue head structure and a data node structure. After it is created, the queue will have one head node and zero or more data nodes, depending on its current state. Figure 4-7 shows the conceptual and physical implementations for our queue structure.

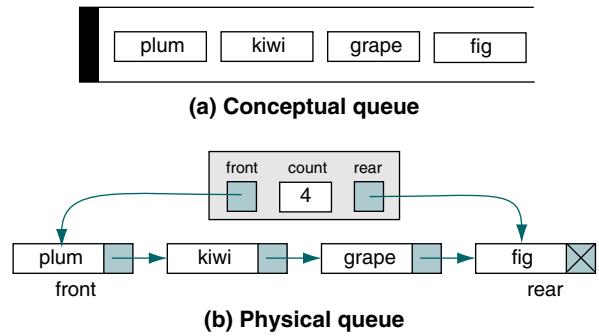


FIGURE 4-7 Conceptual and Physical Queue Implementations

Queue Head

The queue requires two pointers and a count. These fields are stored in the **queue head** structure. Other queue attributes, such as the maximum number of items that were ever present in the queue and the total number of items that have been processed through the queue, can be stored in the head node if such data are relevant to an application. The queue head structure is shown in Figure 4-8.

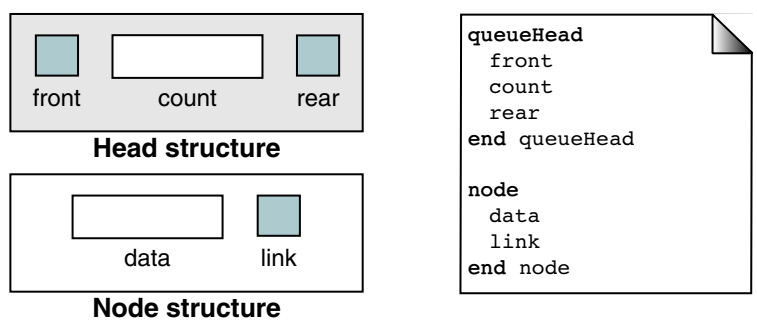


FIGURE 4-8 Queue Data Structure

Queue Data Node

The **queue data node** contains the user data and a link field pointing to the next node, if any. These nodes are stored in dynamic memory and are inserted and deleted as requested by the using program. Its structure is also shown in Figure 4-8.

Queue Algorithms

The queue operations parallel those for a stack, with the addition of an algorithm to look at the data at the rear of the queue. We define these operations here and in the sections that follow. The four basic queue operations are shown in Figure 4-9.

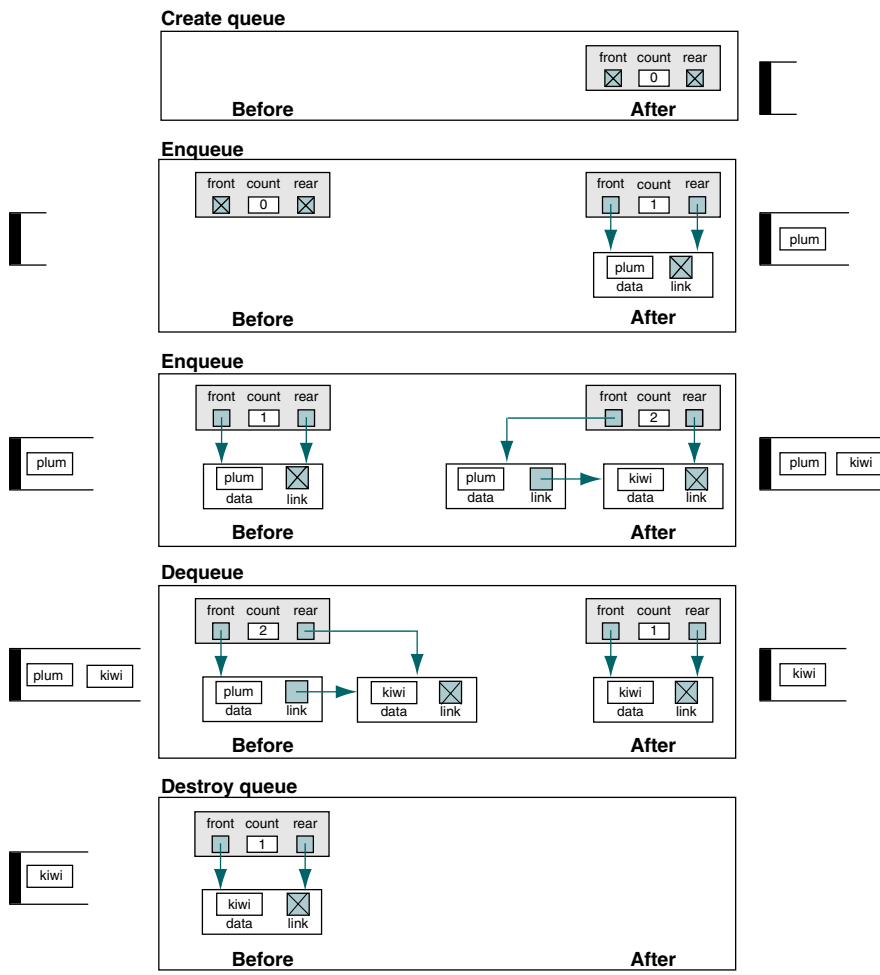


FIGURE 4-9 Basic Queue Functions

Create Queue

The **create queue** operation is rather simple. All we have to do is set the metadata pointers to null and the count to 0. The pseudocode for create queue is shown in Algorithm 4-1.

ALGORITHM 4-1 Create Queue

```
Algorithm createQueue
Creates and initializes queue structure.
  Pre    queue is a metadata structure
  Post   metadata elements have been initialized
  Return queue head
1 allocate queue head
2 set queue front to null
3 set queue rear  to null
4 set queue count to 0
5 return queue head
end createQueue
```

Enqueue

The enqueue is a little more complex than inserting data into a stack. To develop the insertion algorithm, we need to analyze two different queue conditions: insertion into an empty queue and insertion into a queue with data. These operations are shown in Figure 4-10.

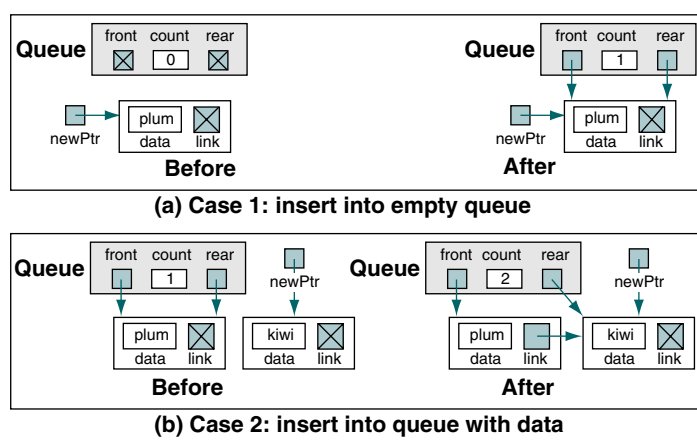


FIGURE 4-10 Enqueue Example

When we insert data into an empty queue, the queue's front and rear pointers must both be set to point to the new node. When we insert data into a queue with data already in it, we must point both the link field in the last

node and the rear pointer to the new node. If the insert was successful, we return a Boolean true; if there is no memory left for the new node, we return a Boolean false. The pseudocode is shown in Algorithm 4-2.

ALGORITHM 4-2 Insert Data into Queue

```
Algorithm enqueue (queue, dataIn)
This algorithm inserts data into a queue.
   Pre    queue is a metadata structure
   Post   dataIn has been inserted
   Return true if successful, false if overflow
 1 if (queue full)
   1  return false
 2 end if
 3 allocate (new node)
 4 move dataIn to new node data
 5 set new node next to null pointer
 6 if (empty queue)
   Inserting into null queue
   1  set queue front to address of new data
 7 else
   Point old rear to new node
   1  set next pointer of rear node to address of new node
 8 end if
 9 set queue rear to address of new node
10 increment queue count
11 return true
end enqueue
```

Dequeue

Although dequeue is also a little more complex than deleting data from a stack, it starts out much the same. We must first ensure that the queue contains data. If the queue is empty, we have underflow and we return false, indicating that the dequeue was not successful.

Given that there are data to be dequeued, we pass the data back through the parameter list and then set the front pointer to the next item in the queue. If we have just dequeued the last item, the queue front pointer automatically becomes a null pointer by assigning it the null pointer from the link field of the last node. However, if the queue is now empty, we must also set the rear pointer to null. These cases are shown in Figure 4-11.

The pseudocode is shown in Algorithm 4-3.

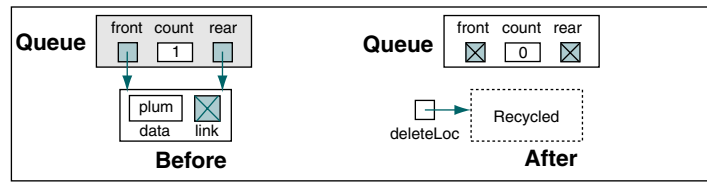
ALGORITHM 4-3 Delete Data from Queue

```
Algorithm dequeue (queue, item)
This algorithm deletes a node from a queue.
  Pre    queue is a metadata structure
```

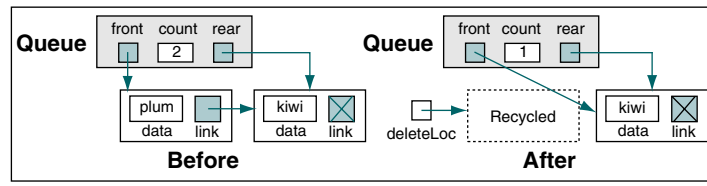
continued

ALGORITHM 4-3 Delete Data from Queue *(continued)*

```
        item is a reference to calling algorithm variable
  Post   data at queue front returned to user through item
         and front element deleted
  Return true if successful, false if underflow
1 if (queue empty)
  1  return false
2 end if
3 move front data to item
4 if (only 1 node in queue)
  Deleting only item in queue
  1  set queue rear to null
5 end if
6 set queue front to queue front next
7 decrement queue count
8 return true
end dequeue
```



(a) Case 1: delete only item in queue



(b) Case 2: delete item at front of queue

FIGURE 4-11 Dequeue Examples

Retrieving Queue Data

The only difference between the two retrieve queue operations—queue front and queue rear—is which pointer is used, front or rear. Let's look at queue front. Its logic is identical to that of dequeue except that the data are not deleted from the queue. It first checks for an empty queue and returns false if the queue is empty. If there are data in the queue, it passes the data back through `dataOut` and returns true. The pseudocode is shown in Algorithm 4-4.

ALGORITHM 4-4 Retrieve Data at Front of Queue

```
Algorithm queueFront (queue, dataOut)
Retrieves data at the front of the queue without changing
queue contents.
  Pre    queue is a metadata structure
         dataOut is a reference to calling algorithm variable
  Post   data passed back to caller
  Return true if successful, false if underflow
1 if (queue empty)
  1  return false
2 end if
3 move data at front of queue to dataOut
4 return true
end queueFront
```

To implement queue rear, copy Algorithm 4-4 and move the data at the rear in statement 3.

Empty Queue

Empty queue returns true if the queue is empty and false if the queue contains data. There are several ways to test for an empty queue. Checking the queue count is the easiest. The pseudocode is shown in Algorithm 4-5.

ALGORITHM 4-5 Queue Empty

```
Algorithm emptyQueue (queue)
This algorithm checks to see if a queue is empty.
  Pre    queue is a metadata structure
  Return true if empty, false if queue has data
1 if (queue count equal 0)
  1  return true
2 else
  1  return false
end emptyQueue
```

Full Queue

Like the stack, **full queue** is another structured programming implementation of data hiding. Depending on the language, it may be one of the most difficult algorithms to implement—ANSI C provides no direct way to implement it. The pseudocode is shown in Algorithm 4-6.

ALGORITHM 4-6 Full Queue

```
Algorithm fullQueue (queue)
This algorithm checks to see if a queue is full. The queue
```

continued

ALGORITHM 4-6 Full Queue *(continued)*

```
is full if memory cannot be allocated for another node.
  Pre    queue is a metadata structure
  Return true if full, false if room for another node
1 if (memory not available)
  1  return true
2 else
  1  return false
3 end if
end fullQueue
```

Queue Count

Queue count returns the number of elements currently in the queue by simply returning the count found in the queue head node. The code is shown in Algorithm 4-7.

ALGORITHM 4-7 Queue Count

```
Algorithm queueCount (queue)
This algorithm returns the number of elements in the queue.
  Pre    queue is a metadata structure
  Return queue count
1 return queue count
end queueCount
```

Destroy Queue

Destroy queue deletes all data in the queue. Algorithm 4-8 contains the code to destroy a queue.

ALGORITHM 4-8 Destroy Queue

```
Algorithm destroyQueue (queue)
This algorithm deletes all data from a queue.
  Pre     queue is a metadata structure
  Post    all data have been deleted
1 if (queue not empty)
  1  loop (queue not empty)
     1  delete front node
  2  end loop
2 end if
3 delete head structure
end destroyQueue
```

4.3 Queue ADT

The queue abstract data type (ADT) follows the basic design of the stack abstract data type. We begin by developing the queue data structure and then we develop each of the queue algorithms in Section 4.2.

Queue Structure

The **queue data structure** is shown in Program 4-1. The node structure is identical to the structure we used for a stack. Each node contains a *void* pointer to the data and a link pointer to the next element in the queue. As with the stack, the using program is responsible for allocating memory to store the data.

Program 4-1 also contains the prototype statements for the queue functions. In a production environment, it would be a queue header file, such as `queues.h`.

PROGRAM 4-1 Queue ADT Data Structures

```
1  //Queue ADT Type Defintions
2    typedef struct node
3      {
4       void*        dataPtr;
5       struct node* next;
6      } QUEUE_NODE;
7    typedef struct
8      {
9       QUEUE_NODE* front;
10      QUEUE_NODE* rear;
11      int         count;
12     } QUEUE;
13
14 //Prototype Declarations
15   QUEUE* createQueue  (void);
16   QUEUE* destroyQueue (QUEUE* queue);
17
18   bool  dequeue    (QUEUE* queue, void** itemPtr);
19   bool  enqueue    (QUEUE* queue, void*  itemPtr);
20   bool  queueFront (QUEUE* queue, void** itemPtr);
21   bool  queueRear  (QUEUE* queue, void** itemPtr);
22   int   queueCount (QUEUE* queue);
23
24   bool  emptyQueue (QUEUE* queue);
25   bool  fullQueue  (QUEUE* queue);
26 //End of Queue ADT Definitions
```

Queue ADT Algorithms

This section contains the C code for each of the queue algorithms. As you study them, trace them through Figure 4-6, "Queue Example."

Create Queue

Create queue allocates a node for the queue header. It then initializes the front and rear pointers to null and sets the count to zero. If overflow occurs, the return value is null; if the allocation is successful, it returns the address of the queue head. The pseudocode design is shown in Algorithm 4-1. The code for create queue is developed in Program 4-2.

PROGRAM 4-2 Create Queue

```
/*================= createQueue ================
  Allocates memory for a queue head node from dynamic
  memory and returns its address to the caller.
     Pre    nothing
     Post   head has been allocated and initialized
     Return head if successful; null if overflow
*/
QUEUE* createQueue (void)
{
//Local Definitions
  QUEUE* queue;

//Statements
  queue = (QUEUE*) malloc (sizeof (QUEUE));
  if (queue)
     {
      queue->front  = NULL;
      queue->rear   = NULL;
      queue->count  = 0;
     } // if
  return queue;
} // createQueue
```

Enqueue

The logic for enqueue, as shown in Algorithm 4-2, is straightforward. If memory is available, it creates a new node, inserts it at the rear of the queue, and returns true. If memory is not available, it returns false. The code is shown in Program 4-3.

PROGRAM 4-3 Enqueue

```
/*================= enqueue ================
  This algorithm inserts data into a queue.
     Pre    queue has been created
     Post   data have been inserted
     Return true if successful, false if overflow
*/
```

continued

PROGRAM 4-3 Enqueue *(continued)*

```
 7  bool enqueue (QUEUE* queue, void* itemPtr)
 8  {
 9  //Local Definitions
10    QUEUE_NODE* newPtr;
11
12  //Statements
13    if (!(newPtr =
14       (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE))))
15       return false;
16
17    newPtr->dataPtr = itemPtr;
18    newPtr->next    = NULL;
19
20    if (queue->count == 0)
21       // Inserting into null queue
22       queue->front  = newPtr;
23    else
24       queue->rear->next = newPtr;
25
26    (queue->count)++;
27    queue->rear = newPtr;
28    return true;
29  } // enqueue
```

Program 4-3 Analysis Because we must maintain both a front and a rear pointer, we need to check to see if we are inserting into a null queue. If we are, we must set both pointers to the data just inserted. If there are already data in the queue, we need to set the next field of the node at the rear of the queue and the rear pointer to the new node. In this case the front pointer is unchanged. Because the rear pointer is updated in either case, we changed it after the *if* statement (see statement 27).

Dequeue

Dequeue follows the basic design developed in Algorithm 4-3. It begins by checking to make sure there are data in the queue. If there are, it passes the address of the data being deleted back to the calling function through a reference parameter, adjusts the pointers, and subtracts one from the queue count. If the delete is successful, it returns true; if it is empty, it returns false. The code is shown in Program 4-4.

PROGRAM 4-4 Dequeue

```
 1  /*================= dequeue ================
 2    This algorithm deletes a node from the queue.
 3       Pre    queue has been created
 4       Post   Data pointer to queue front returned and
```

continued

PROGRAM 4-4 Dequeue *(continued)*

```
 5                front element deleted and recycled.
 6            Return true if successful; false if underflow
 7   */
 8   bool dequeue (QUEUE* queue, void** itemPtr)
 9   {
10   //Local Definitions
11     QUEUE_NODE* deleteLoc;
12
13   //Statements
14     if (!queue->count)
15         return false;
16
17     *itemPtr  = queue->front->dataPtr;
18     deleteLoc = queue->front;
19     if (queue->count == 1)
20        // Deleting only item in queue
21        queue->rear  = queue->front = NULL;
22     else
23        queue->front = queue->front->next;
24     (queue->count)--;
25     free (deleteLoc);
26
27     return true;
28   } // dequeue
```

Program 4-4 Analysis

The logic for dequeue is rather basic. The major concern is that we may have deleted the last element in the queue, in which case we must set it to a null state. This requires that the queue rear be set to a null pointer. Note that we always set the queue front to the next pointer in the node being deleted. If the last node is being deleted, the deleted node's next pointer is guaranteed to be null (because it is the last node in the queue). In this case queue front is automatically set to null. If it is not the last node in the queue, queue front points to the new node at the front of the queue.

Although the code for dequeue is simple enough, the call to it is a little tricky. Because the dequeue data parameter is a *void* pointer to a pointer, the call must cast the data pointer to *void*. The call is shown below.

```
dequeue(queue, (void*)&dataPtr)
```

Because dequeue returns a Boolean success flag, the call should be an expression in a selection statement to test for success or failure.

```
if (dequeue(queue, (void*)&dataPtr))
```

Queue Front

Queue front also passes the address of the data at the front of the queue back to the calling function. It differs from dequeue only in that the queue is not changed. The code is shown in Program 4-5; the pseudocode design is shown in Algorithm 4-4.

PROGRAM 4-5 Queue Front

```
/*================== queueFront =================
  This algorithm retrieves data at front of the
  queue without changing the queue contents.
     Pre    queue is pointer to an initialized queue
     Post   itemPtr passed back to caller
     Return true if successful; false if underflow
*/
bool queueFront (QUEUE* queue, void** itemPtr)
{
//Statements
  if (!queue->count)
      return false;
  else
     {
      *itemPtr = queue->front->dataPtr;
       return true;
     } // else
} // queueFront
```

Queue Rear

The code for queue rear, shown in Program 4-6, is identical to queue front except that the address of the data at the rear of the queue is sent back to the calling function.

PROGRAM 4-6 Queue Rear

```
/*================== queueRear =================
  Retrieves data at the rear of the queue
  without changing the queue contents.
     Pre    queue is pointer to initialized queue
     Post   Data passed back to caller
     Return true if successful; false if underflow
*/
bool queueRear (QUEUE* queue, void** itemPtr)
{
//Statements
  if (!queue->count)
      return true;
```

continued

PROGRAM 4-6 Queue Rear *(continued)*

```
13    else
14       {
15        *itemPtr = queue->rear->dataPtr;
16        return false;
17       } // else
18  } // queueRear
```

Queue Empty

Queue empty is one of the simplest functions in the library. It simply uses the count in the queue header to determine if the queue is empty or contains data, as shown in Program 4-7.

PROGRAM 4-7 Empty Queue

```
1   /*================== emptyQueue =================
2     This algorithm checks to see if queue is empty.
3     Pre    queue is a pointer to a queue head node
4     Return true if empty; false if queue has data
5   */
6   bool emptyQueue (QUEUE* queue)
7   {
8   //Statements
9     return (queue->count == 0);
10  } // emptyQueue
```

Full Queue

As mentioned in the discussion of Algorithm 4-6, full queue is one of the more difficult algorithms to implement because C provides no way to test the amount of space left in dynamic memory. The only way we can tell if there is any room left is to allocate a node and test for success. If the node was allocated successfully, we free it and return false for not full. If there was no memory left, we return true—the queue is full. The code for full queue is shown in Program 4-8.

PROGRAM 4-8 Full Queue

```
1   /*================== fullQueue =================
2     This algorithm checks to see if queue is full. It
3     is full if memory cannot be allocated for next node.
4        Pre    queue is a pointer to a queue head node
5        Return true if full; false if room for a node
6   */
7   bool fullQueue (QUEUE* queue)
8   {
```

continued

PROGRAM 4-8 Full Queue *(continued)*

```
 9  //Local Definitions
10  QUEUE_NODE* temp;
11
12  //Statements
13    temp = (QUEUE_NODE*)malloc(sizeof(*(queue->rear)));
14    if (temp)
15       {
16        free (temp);
17        return true;
18       } // if
19    // Heap full
20    return false;
21  } // fullQueue
```

Queue Count

Queue count simply returns the count found in the queue head node. Its code is shown in Program 4-9.

PROGRAM 4-9 Queue Count

```
 1  /*================== queueCount =================
 2    Returns the number of elements in the queue.
 3       Pre    queue is pointer to the queue head node
 4       Return queue count
 5  */
 6  int queueCount(QUEUE* queue)
 7  {
 8  //Statements
 9    return queue->count;
10  } // queueCount
```

Destroy Queue

The design of **destroy queue** in Algorithm 4-8 is relatively simple; the implementation in Program 4-10 is relatively complex. While it is apparent that we must delete all elements in the queue and free each of their nodes, it is easy to forget to recycle the data space. We begin by checking to make sure that the queue exists. If it doesn't, we simply return a null pointer. If the queue is valid, we cycle through it, deleting all of the elements.

PROGRAM 4-10 Destroy Queue

```
 1  /*================== destroyQueue =================
 2    Deletes all data from a queue and recycles its
 3    memory, then deletes & recycles queue head pointer.
 4       Pre    Queue is a valid queue
```

continued

PROGRAM 4-10 Destroy Queue *(continued)*

```
 5        Post   All data have been deleted and recycled
 6        Return null pointer
 7  */
 8  QUEUE* destroyQueue (QUEUE* queue)
 9  {
10  //Local Definitions
11    QUEUE_NODE* deletePtr;
12
13  //Statements
14    if (queue)
15       {
16        while (queue->front != NULL)
17           {
18            free (queue->front->dataPtr);
19            deletePtr    = queue->front;
20          queue->front = queue->front->next;
21            free (deletePtr);
22           } // while
23        free (queue);
24       } // if
25    return NULL;
26  } // destroyQueue
```

Program 4-10 Analysis In the normal course of events, the queue being destroyed is empty. If it is, the queue front is null and all we need to do is free the queue header and return. If there are data in the queue, however, we first free the data space using the data pointer stored in the queue node, then we delete the queue node itself. Eventually, we get to the end of the queue, at which time we free the queue head node and return.

4.4 Queuing Theory

Queuing theory is a field of applied mathematics that is used to predict the performance of queues. In this section we review a few basic queuing theory concepts. We leave the mathematics of queuing theory to advanced courses on the subject.

> Queuing theory is a field of applied mathematics that is used to predict the performance of queues.

Queues can be divided into single-server and multiserver types. A **single-server queue** can provide service to only one customer at a time. An example of a single-server queue is the hot-food vendor found on street corners in most large cities. **Multiserver queues**, on the other hand, can provide service to many customers at a time. An example of a multiserver queue is a bank in which there is one line with many bank tellers providing service. Note that grocery stores are examples of multiple single-server queues, or *multiqueues*.

The two common elements to all queues are one or more customers who need a service and the a server who provides the service. A **customer** is any person or thing needing service. Besides the examples cited, customers can be as diverse as jobs waiting to be run in a computer and packages being sent by express delivery. The **service** is any activity needed to accomplish the required result. The hot-food vendor provides food as a service; the package handler provides transportation and delivery of packages.

The two factors that most dramatically affect the queue are the arrival rate and the service time. The rate at which customers arrive in the queue for service is known as the **arrival rate**. Depending on the service being provided, the arrival rate may be random or regular. The hot-food vendor's customers more than likely arrive randomly, although the rate may vary widely during the day. Jobs to be processed in the computer, however, may arrive at some regular rate created by a job-scheduling system.

Service time is the average time required to complete the processing of a customer request. If you have ever observed different bank customers while you stood in line, you noted the wide range in time required to process each customer. Obviously, the faster customers arrive and the higher the service time, the longer the queue.

In an ideal situation, customers would arrive at a rate that matches the service time. However, as we all know, things are seldom ideal. Sometimes the server will be idle because there are no customers to be served. At other times there will be many customers waiting to be served. If we can predict the patterns, we may be able to minimize idle servers and waiting customers. Doing so is especially important when customers who have to wait for a long time are apt to switch to an alternative server, such as the hot-food vendor a block down the street.

Queuing theory attempts to predict such patterns. Specifically, it attempts to predict *queue time* (which is defined as the average length of time customers wait in the queue), the average size of the queue, and the maximum queue size. To make these predictions, it needs two factors: the arrival rate and the average service time, which is defined as the average of the total service time between idle periods.

Given queue time and service time, we know *response time*,—a measure of the average time from the point at which customers enter the queue until the moment they leave the server. Response time is an important statistic, especially in online computer systems. A queuing theory model is shown in Figure 4-12.

Once a model of a queue has been built, it can be used to study proposed changes to the system. For example, in the banking queue, if we were able to use automation to reduce the average service time by 15%, how many fewer tellers would we need? Or, given a model of a growing system that is currently under capacity, how long will it be before we need to add another server?

The two factors that most affect the performance of queues are the arrival rate and the service time.

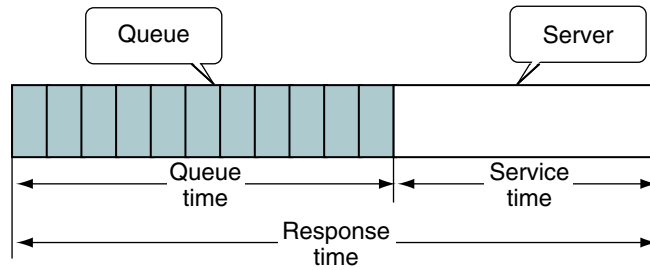


FIGURE 4-12 Queuing Theory Model

4.5 Queue Applications

Queues are one of the more common data-processing structures. They are found in virtually every operating system and every network and in countless other areas. For example, queues are used in business online applications such as processing customer requests, jobs, and orders. In a computer system, a queue is needed to process jobs and for system services such as print spools.

Queues can become quite complex. In this section we demonstrate two queue implementations: The first is an application that is useful in many problems—categorizing data. For this application we show both the design and the implementation. The second is a queue simulation that can be used to study the performance of any queue application. Only the design is developed; the implementation is left as a project at the end of the chapter.

Categorizing Data

It is often necessary to rearrange data without destroying their basic sequence. As a simple example, consider a list of numbers. We need to group the numbers while maintaining their original order in each group. This is an excellent multiple-queue application.

To demonstrate, consider the following list of numbers:

```
3 22 12 6 10 34 65 29 9 30 81 4 5 19 20 57 44 99
```

We want to categorize them into four different groups:

- Group 1: less than 10
- Group 2: between 10 and 19
- Group 3: between 20 and 29
- Group 4: 30 and greater

In other words, we want the list rearranged as shown below.

```
| 3 6 9 4 5 | 12 10 19 | 22 29 20| 34 65 30 81 57 44 99 |
```

This is not sorting. The result is not a sorted list but rather a list categorized according to the specified rules. The numbers in each group have kept their original order.

Categorizing Data Design

The solution is simple. We build a queue for each of the four categories. We then store the numbers in the appropriate queue as we read them. After all the data have been processed, we print each queue to demonstrate that we categorized the data correctly. This design is shown in Algorithm 4-9.

ALGORITHM 4-9 Category Queues

```
Algorithm categorize
  Group a list of numbers into four groups using four queues.
     Written by:
     Date:
 1 createQueue (q0to9)
 2 createQueue (q10to19)
 3 createQueue (q20to29)
 4 createQueue (qOver29)
 5 fillQueues  (q0to9, q10to19, q20to29, qOver29)
 6 printQueues (q0to9, q10to19, q20to29, qOver29)
end categorize
```

Algorithm 4-9 Analysis

The mainline for the category queue simply creates the queues and then calls algorithms to fill and print them. Note that when we complete, we do not destroy the queues. We leave this process for the operating system. If we had more work to do, however, we would delete them to release their memory.

The pseudocode to fill the queue is shown in Algorithm 4-10. We demonstrate the logic for printing the queues when we write the C implementation in the next section.

ALGORITHM 4-10 Fill Category Queues

```
Algorithm fillQueues (q0to9, q10to19, q20to29, qOver29)
This algorithm reads data from the keyboard and places them in
one of four queues.
  Pre  all four queues have been created
  Post queues filled with data
 1 loop (not end of data)
```

continued

ALGORITHM 4-10 Fill Category Queues *(continued)*

```
   1  read (number)
   2  if (number < 10)
      1  enqueue (q0to9, number)
   3  elseif (number < 20)
      1  enqueue (q10to19, number)
   4  elseif (number < 30)
      1  enqueue (q20to29, number)
   5  else
      1  enqueue (qOver29, number)
   6  end if
 2 end loop
end fillQueues
```

Algorithm 4-10 Analysis

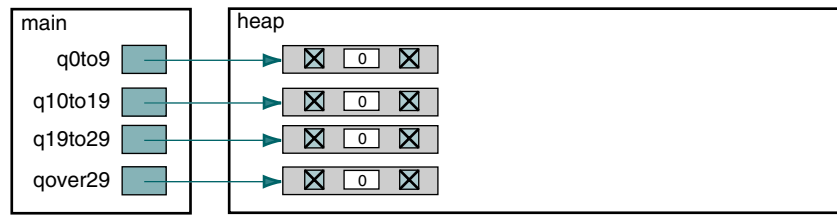
What sounded like a long and difficult program is really a short and simple algorithm, thanks to code reusability. The most difficult decision in this algorithm is determining how to write the categorizing code. We simply used a multiway selection implemented with an *elseif* construct. Once we made that decision, the algorithm practically wrote itself.

Categorizing Data—C Implementation

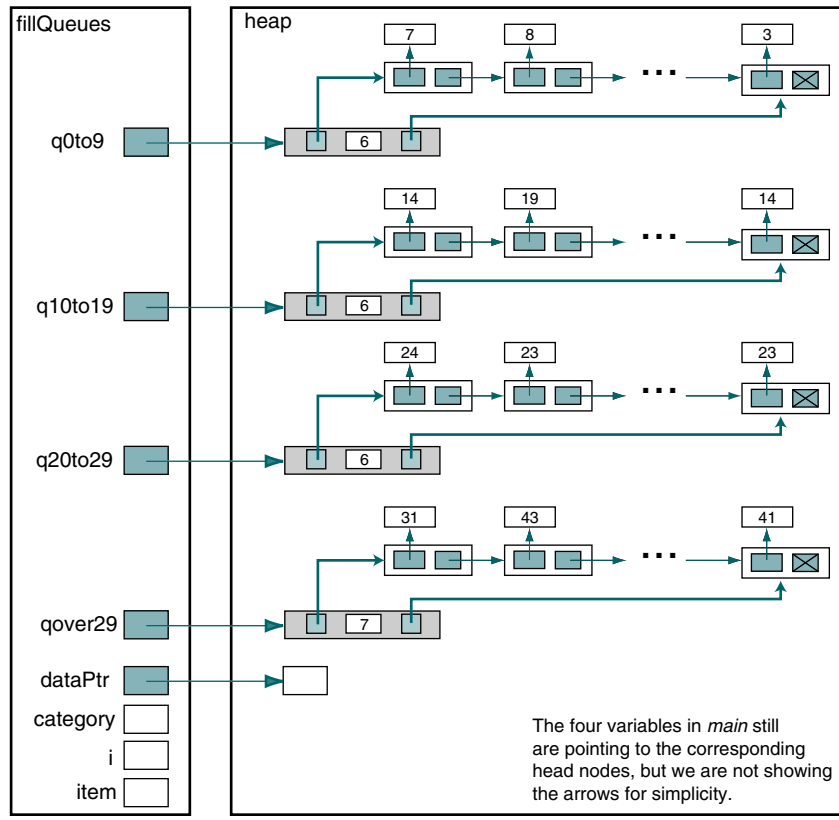
Rather than read the data from a file, however, we simply generate 25 random numbers in the range 0 to 50. The C functions use the abstract data type created in Section 4.3. To help you follow the program, we include Figure 4-13, which shows the structures after they have been created.

Mainline Logic

The mainline logic for the categorizing program begins by creating the queues. Once the queues have been created, it calls a function to create and enqueue the data and another function to print it. The code is shown in Program 4-11.



(a) Before calling fillQueues



(b) After calling fillQueues

FIGURE 4-13 Structures for Categorizing Data

PROGRAM 4-11 Categorizing Data Mainline

```
1  /*Groups numbers into four groups using four queues.
2       Written by:
3       Date:
```

continued

PROGRAM 4-11 Categorizing Data Mainline *(continued)*

```
 4  */
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include <stdbool.h>
 8  #include "queues.h"
 9
10  //Prototype Statements
11  void fillQueues  (QUEUE*, QUEUE*, QUEUE*, QUEUE*);
12  void printQueues (QUEUE*, QUEUE*, QUEUE*, QUEUE*);
13
14  void printOneQueue (QUEUE* pQueue);
15
16  int main (void)
17  {
18  //Local Definitions
19    QUEUE* q0to9;
20    QUEUE* q10to19;
21    QUEUE* q20to29;
22    QUEUE* qOver29;
23
24  //Statements
25    printf("Welcome to a demonstration of categorizing\n"
26           "data. We generate 25 random numbers and then\n"
27           "group them into categories using queues.\n\n");
28
29    q0to9   = createQueue ();
30    q10to19 = createQueue ();
31    q20to29 = createQueue ();
32    qOver29 = createQueue ();
33
34    fillQueues  (q0to9, q10to19, q20to29, qOver29);
35    printQueues (q0to9, q10to19, q20to29, qOver29);
36
37    return 0;
38  } // main
```

Fill Queues

Fill queues uses a random-number generator to create 25 random numbers between 0 and 50. As each number is generated, it is displayed so that we can verify the accuracy of the program, and then each number is inserted into the appropriate queue. The code is in Program 4-12.

PROGRAM 4-12 Categorizing Data: Fill Queues

```
/*================= fillQueues ================
  This function generates data using rand() and
  places them in one of four queues.
     Pre:  All four queues have been created
      Post: Queues filled with data
*/
void fillQueues (QUEUE* q0to9,   QUEUE* q10to19,
                 QUEUE* q20to29, QUEUE* qOver29)
{
//Local Definitions
  int  category;
  int  item;
  int* dataPtr;

//Statements
  printf("Categorizing data:\n");
  srand(79);

  for (int i = 1; i <= 25; i++)
      {
      if (!(dataPtr = (int*) malloc (sizeof (int))))
          printf("Overflow in fillQueues\a\n"),
                 exit(100);

       *dataPtr = item = rand() % 51;
       category = item / 10;
       printf("%3d", item);
       if (!(i % 11))
          // Start new line when line full
          printf("\n");

       switch (category)
          {
           case 0 : enqueue (q0to9, dataPtr);
                    break;
           case 1 : enqueue (q10to19, dataPtr);
                    break;
           case 2 : enqueue (q20to29, dataPtr);
                    break;
           default: enqueue (qOver29, dataPtr);
                    break;
          } // switch
      } // for
  printf("\nEnd of data categorization\n\n");
  return;
} // fillQueues
```

Print Queues

Print queues is a simple driver that calls a function to print one queue for each queue. Its code is shown in Program 4-13.

PROGRAM 4-13 Print Queues

```
/*================= printQueues =================
  This function prints the data in each of the queues.
     Pre  Queues have been filled
     Post Data printed and dequeued
*/
void printQueues (QUEUE* q0to9,   QUEUE* q10to19,
                  QUEUE* q20to29, QUEUE* qOver29)
{
//Statements
  printf("Data   0.. 9:");
  printOneQueue (q0to9);

  printf("Data  10..19:");
  printOneQueue (q10to19);

  printf("Data  20..29:");
  printOneQueue (q20to29);

  printf("Data over 29:");
  printOneQueue (qOver29);

  return;
} // printQueues
```

Print One Queue

Print one queue uses a straightforward approach. Each call prints a queue by dequeuing a node and printing it. The loop continues as long as there are data in the queue. Because we don't have access to the data structure, however, we use empty queue to test for more data to be printed. We could also have used queue count and tested for greater than zero. The code is shown in Program 4-14. It also contains the results from a sample run.

PROGRAM 4-14 Print One Queue

```
/*================= printOneQueue =================
  This function prints the data in one queue,
  ten entries to a line.
     Pre  Queue has been filled
     Post Data deleted and printed. Queue is empty
*/
void printOneQueue (QUEUE* pQueue)
```

continued

PROGRAM 4-14 Print One Queue *(continued)*

```
{
//Local Definitions
  int  lineCount;
  int* dataPtr;

//Statements
  lineCount = 0;
  while (!emptyQueue (pQueue))
      {
       dequeue (pQueue, (void*)&dataPtr);
       if (lineCount++ >= 10)
          {
           lineCount = 1;
           printf ("\n             ");
          } // if
       printf("%3d ", *dataPtr);
      } // while !emptyQueue
  printf("\n");

  return;
} // printOne Queue
```

```
Results:
Welcome to a demonstration of categorizing
data. We generate 25 random numbers and then
group them into categories using queues.

Categorizing data:
 24  7 31 23 26 14 19  8  9  6 43
 16 22  0 39 46 22 38 41 23 19 18
 14  3 41
End of data categorization

Data   0.. 9:  7   8   9   6   0   3
Data  10..19: 14  19  16  19  18  14
Data  20..29: 24  23  26  22  22  23
Data over 29: 31  43  39  46  38  41  41
```

Queue Simulation

An important application of queues is **queue simulation**, a modeling activity used to generate statistics about the performance of queues. Let's build a model of a single-server queue—for example, a saltwater taffy store on a beach boardwalk. The store has one window, and a clerk can service only one customer at a time. The store also ships boxes of taffy anywhere in the country. Because there are many flavors of saltwater taffy and because it takes longer to serve a

customer who requests that the taffy be mailed, the time to serve customers varies between 1 and 10 minutes.

We want to study the store's activity over a hypothetical day. The store is open 8 hours per day, 7 days a week. To simulate a day, we build a model that runs for 480 minutes (8 hours times 60 minutes per hour).

The simulation uses a digital clock that lets events start and stop in 1- minute intervals. In other words, customers arrive on the minute, wait an integral number of minutes in the queue, and require an integral number of minutes to be served. In each minute of operation, the simulation needs to check three events: the arrival of customers, the start of customer processing, and the completion of customer processing.

Events

The arrival of a new customer is determined in a module we name *new customer*. To determine the arrival rate, the store owner used a stopwatch and studied customer patterns over several days. The owner found that, on average, a customer arrives every 4 minutes. We simulate an arrival rate using a random-number generator that returns a value between 1 and 4. If the number is 4, a customer has arrived. If the number is 1, 2, or 3, no customer has arrived.

We start processing a customer when the server is idle. In each minute of the simulation, therefore, the simulator needs to determine whether the clerk (the server) is busy or idle. In the simulation this is done with a module called *server free*. If the clerk is idle, the next waiting customer in line (the queue) can be served. If the clerk is busy, the waiting customers remain in the queue.

Finally, at the end of each minute, the simulation determines whether it has completed the processing for the current customer. The processing time for the current customer is determined by a random-number generator when the customer processing is started. Then, for each customer, we loop the required number of minutes to complete the transaction. When the customer has been completely served, we gather statistics about the sale and set the server to an idle state.

Data Structures

Four data structures are required for the queue simulation: a queue head, a queue node, a current customer status, and a simulation statistics structure. These structures are described below and shown in Figure 4-14.

Queue Head

We use the standard head node structure for the queue. It contains two node pointers—front and rear—and a count of the number of elements currently in the queue.

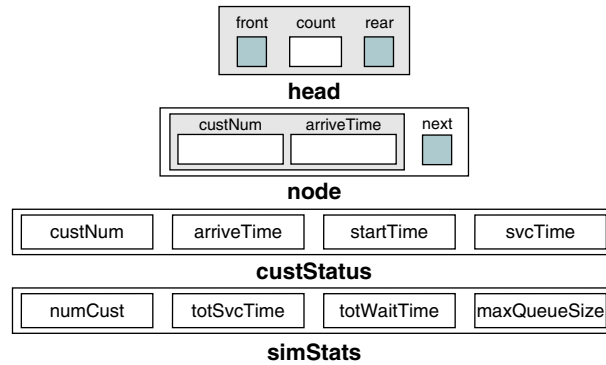


FIGURE 4-14 Queue Data Structures

Queue Node

The queue node contains only two elements: the customer data and a next node pointer. The customer data consist of a sequential customer number and the arrival time. The customer number is like the number you take when you go into a busy store with many customers waiting for service. The arrival time is the time the customer arrived at the store and got in line (was enqueued). The next node pointer is used to point to the next customer in line.

Customer Status

While we are processing the customer's order, we use a customer status structure to keep track of four pieces of data. First we store the customer's number and arrival time. Because we need to know what time we started processing the order, we store the start time. Finally, as we start the processing, we use a random-number generator to calculate and store the time it will take to fill the customer's order.

Simulation Statistics

At the conclusion of the simulation, we need to report the total number of customers processed in the simulation, the total and average service times, the total and average wait times, and the greatest number of customers in the queue at one time. We use a simulation statistics structure to store these data.

Output

At the conclusion of the simulation, we print the statistics gathered during the simulation along with the average queue wait time and average queue service time. To verify that the queue is working properly, we also print the following statistics each time a customer is completely served: arrival time, start time, wait time, service time, and the number of elements in the queue.

Simulation Algorithm

We are now ready to write the simulation algorithm. The design consists of a driver module that calls three processing modules, as shown in Figure 4-15. It parallels the requirements discussed earlier. New customer determines whether a customer has arrived and needs to be enqueued. Server free determines whether the server is idle, and if so, it starts serving the next customer, if one exists. Service complete determines whether the current customer has been completely served, and if so, collects the necessary statistical data and prints the data about the customer. At the end of the simulation, we print the statistics using the print stats module.

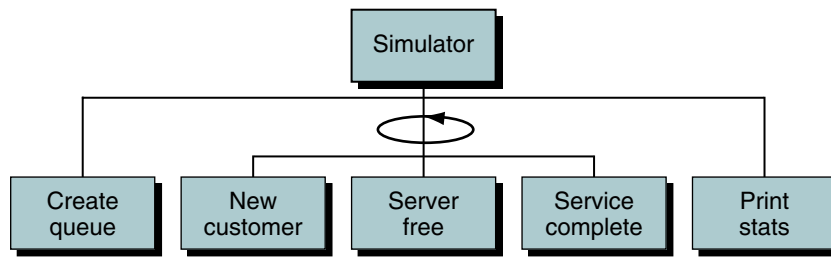


FIGURE 4-15 Design for Queue Simulation

Simulator

The pseudocode for the simulation driver is shown in Algorithm 4-11.

ALGORITHM 4-11 Queue Simulation: Driver

```
Algorithm taffySimulation
This program simulates a queue for a saltwater taffy store.
      Written by:
      Date:
 1 createQueue (queue)
 2 loop (clock <= endTime OR moreCusts)
   1  newCustomer  (queue, clock, custNum)
   2  serverFree   (queue, clock, custStatus, moreCusts)
   3  svcComplete  (queue, clock, custStatus,
                    runStats,     moreCusts)
   4  if (queue not empty)
      1  set moreCusts true
   5  end if
   6  increment clock
 3 end loop
 4 printStats (runStats)
end taffySimulation
```

Algorithm 4-11 Analysis

The driver creates the queue and then loops until the simulation is complete. Each loop tests to see whether a new customer needs to be enqueued, checks to see whether the server is idle so that a new customer can be started, and checks to see whether the current customer's service is complete.

To determine whether the simulation is complete, we need to test two conditions. We can only stop the simulation when we have run for the allocated time *and* there are no more customers. If there is more time left in the simulation, we are not finished. Even when we reach the end of the processing time, however, we are not finished if a customer is being served or if customers are waiting in the queue. We use a more customers flag to determine whether either of these conditions is true.

The server free logic sets the more customers flag to true when it starts a call (statement 2.2). The service complete logic sets it to false when it completes a call (statement 2.3). In the driver loop, we need to set it to true if there are calls waiting in the queue (statement 2.4). This test ensures that calls waiting in the queue are handled after the clock time has been exhausted.

At the end of the simulation, the driver calls a print algorithm that calculates the averages and prints all of the statistics.

New Customer

The pseudocode for new customer is shown in Algorithm 4-12.

ALGORITHM 4-12 Queue Simulation: New Customer

```
Algorithm  newCustomer (queue, clock, custNum)
This algorithm determines if a new customer has arrived.
  Pre    queue is a structure to a valid queue
         clock is the current clock minute
         custNum is the number of the last customer
  Post   if new customer has arrived, placed in queue
1 randomly determine if customer has arrived
2 if (new customer)
  1  increment custNum
  2  store custNum in custData
  3  store arrival time in custData
  4  enqueue (queue, custData)
3 end if
end newCustomer
```

Algorithm 4-12 Analysis

It is important to note that we could not simply add a customer every fourth call. For the simulation study to work, the customers must arrive in a random fashion, not exactly every 4 minutes. By randomly determining when a customer has arrived (using a random-number generator), we may go several minutes without any customers and then have several customers arrive in succession. This random distribution is essential to queuing theory.

Server Free

The pseudocode to determine whether we can start serving a new customer is shown in Algorithm 4-13.

ALGORITHM 4-13 Queue Simulation: Server Free

```
Algorithm serverFree (queue, clock, status, moreCusts)
This algorithm determines if the server is idle and if so
starts serving a new customer.
  Pre   queue is a structure for a valid queue
        clock is the current clock minute
        status holds data about current/previous customer
  Post  moreCusts is set true if a call is started
1 if (clock > status startTime + status svcTime - 1)
     Server is idle.
  1  if (not emptyQueue (queue))
     1  dequeue (queue, custData)
     2  set status custNum    to custData number
     3  set status arriveTime to custData arriveTime
     4  set status startTime  to clock
     5  set status svcTime    to random service time
     6  set moreCusts true
  2  end if
2 end if
end serverFree
```

Algorithm 4-13 Analysis

The question in this algorithm is, "How can we tell if the server is free?" The customer status record holds the answer. It represents either the current customer or, if there is no current customer, the previous customer. If there is a current customer, we cannot start a new customer. We can tell if the record represents the new customer by comparing the clock time with the end time for the customer. The end time is the start time plus the required service time. If the clock time is greater than the calculated end time, then, if the queue is not empty, we start a new customer.

One question remains to be answered: "How do we start the first call?" When we create the customer status structure, we initialize everything to 0. Now, using the formula in statement 1, we see that a 0 start time plus a 0 service time minus 1 must be less than the clock time, so we can start the first customer.

Service Complete

The logic for service complete is shown in Algorithm 4-14.

ALGORITHM 4-14 Queue Simulation: Service Complete

```
Algorithm svcComplete (queue, clock, status,
                       stats, moreCusts)
This algorithm determines if the current customer's
processing is complete.
  Pre    queue is a structure for a valid queue
         clock is the current clock minute
         status holds data about current/previous customer
         stats contains data for complete simulation
```

continued

ALGORITHM 4-14 Queue Simulation: Service Complete *(continued)*

```
  Post   if service complete, data for current customer
         printed and simulation statistics updated
         moreCusts set to false if call completed
1 if (clock equal status startTime + status svcTime - 1)
  Current call complete
  1  set waitTime to status startTime - status arriveTime
  2  increment stats numCust
  3  set stats totSvcTime  to stats totSvcTime + status svcTime
  4  set stats totWaitTime to stats totWaitTime + waitTime
  5  set queueSize to queueCount (queue)
  6  if (stats maxQueueSize < queueSize)
     1  set stats maxQueueSize to queueSize
  7  end if
  8  print (status custNum   status arriveTime
            status startTime status svcTime
            waitTime         queueCount (queue))
  9  set    moreCusts to false
2 end if
end svcComplete
```

Algorithm 4-14 Analysis

The question in this algorithm is similar to the server free question. To help analyze the logic, let's look at a few hypothetical service times for the simulation. They are shown in Table 4-1.

Start time	Service time	Time completed	Minutes served
1	2	2	1 and 2
3	1	3	3
4	3	6	4, 5, and 6
7	2	8	7 and 8

TABLE 4-1 Hypothetical Simulation Service Times

Because we test for a new customer in server free before we test for the customer being complete, it is possible to start and finish serving a customer who needs only 1 minute in the same loop. This is shown in minute 3 above. The correct calculation for the end of service time is therefore

```
start time + service time – 1
```

This calculation is borne out in each of the examples in Table 4-1. We therefore test for the clock time equal to the end of service time, and if they

are equal, we print the necessary data for the current customer and accumulate the statistics needed for the end of the simulation.

Print Stats

The last algorithm in the simulation prints the statistics for the entire simulation. The code is shown in Algorithm 4-15.

ALGORITHM 4-15 Queue Simulation: Print Statistics

```
Algorithm printStats (stats)
This algorithm prints the statistics for the simulation.
  Pre    stats contains the run statistics
  Post   statistics printed
 1 print (Simulation Statistics:)
 2 print ("Total customers: "    stats numCust)
 3 print ("Total service time: " stats totSvcTime)
 4 set avrgSvcTime to stats totSvcTime / stats numCust
 5 print ("Average service time: " avrgSvcTime)
 6 set avrgWaitTime to stats totWaitTime / stats numCust
 7 print ("Average wait time: " avrgWaitTime)
 8 print ("Maximum queue size: " stats maxQueueSize)
end printStats
```

4.6 Key Terms

arrival rate
create queue
customer
dequeue
destroy queue
empty queue
enqueue
first in–first out (FIFO)
front
full queue
multiserver queues
queue count
queue data node
queue data structure
queue front
queue head
queue rear
queue simulation
rear
service
service time
single-server queue

4.7 Summary

- A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.
- A queue is a first in–first out (FIFO) structure.
- There are four basic queue operations: enqueue, dequeue, queue front, and queue rear.
  1. The enqueue operation inserts an element at the rear of the queue.
  2. The dequeue operation deletes the element at the front of the queue.
  3. The queue front operation examines the element at the front of the queue without deleting it.
  4. The queue rear operation examines the element at the rear of the queue without deleting it.
- Queues can be implemented using linked lists or arrays.
- To implement the queue using a linked list, we use two types of structures: a head and a node.
- Queuing theory is a field of applied mathematics that is used to predict the performance of queues.
- Queue applications can be divided into single servers and multiservers.
  1. A single-server queue application provides services to only one customer at a time.
  2. A multiserver queue application provides service to several customers at a time.

- The two features that most affect the performance of queues are the arrival rate and the service time.
  1. The rate at which the customers arrive in the queue for service is known as the arrival rate.
  2. Service time is the average time required to complete the processing of a customer request.
- The queue time is the average length of time customers wait in the queue.
- The response time is a measure of average time from the point at which customers enter the queue until the moment they leave the server. It is queue time plus service time.
- One application of queues is queue simulation, which is a modeling activity used to generate statistics about the performance of a queue.
- Another application of queues is categorization. Queues are used to categorize data into different groups without losing the original ordering of the data.

4.8 Practice Sets

Exercises

1. Imagine you have a stack of integers, `S`, and a queue of integers, `Q`. Draw a picture of `S` and `Q` after the following operations:

```
 1 pushStack (S, 3)
 2 pushStack (S, 12)
 3 enqueue (Q, 5)
 4 enqueue (Q, 8)
 5 popStack (S, x)
 6 pushStack (S, 2)
 7 enqueue (Q, x)
 8 dequeue (Q, y)
 9 pushStack (S, x)
10 pushStack (S, y)
```

2. What would be the value of queues `Q1` and `Q2`, and stack `S` after the following algorithm segment:

```
 1 S  = createStack
 2 Q1 = createQueue
 3 Q2 = createQueue
 4 enqueue (Q1, 5)
 5 enqueue (Q1, 6)
 6 enqueue (Q1, 9)
 7 enqueue (Q1, 0)
 8 enqueue (Q1, 7)
 9 enqueue (Q1, 5)
10 enqueue (Q1, 0)
11 enqueue (Q1, 2)
12 enqueue (Q1, 6)
13 loop (not emptyQueue (Q1))
   1 dequeue (Q1, x)
   2 if (x == 0)
     1 z = 0
     2 loop (not emptyStack (S))
       1 popStack (S, y)
       2 z = z + y
     3 end loop
     4 enqueue (Q2, z)
   3 else
     1 pushStack (S, x)
   4 end if
14 end loop
```

3. What would be the contents of queue `Q` after the following code is executed and the following data are entered?

```
1 Q = createQueue
2 loop (not end of file)
  1 read number
  2 if (number not 0)
    1 enqueue (Q, number)
  3 else
    1 x = queuerear (Q)
    2 enqueue (Q, x)
  4 end if
3 end loop
```

The data are: 5, 7, 12, 4, 0, 4, 6, 8, 67, 34, 23, 5, 0, 44, 33, 22, 6, 0.

4. What would be the contents of queue Q1 and queue Q2 after the following code is executed and the following data are entered?

```
1 Q1 = createQueue
2 Q2 = createQueue
3 loop (not end of file)
  1 read number
  2 enqueue (Q1, number)
  3 enqueue (Q2, number)
  4 loop (not empty Q1)
    1 dequeue (Q1, x)
    2 enqueue (Q2, x)
  5 end loop
4 end loop
```

   The data are 5, 7, 12, 4, 0, 4, 6.

5. What would be the contents of queue Q1 after the following code is executed and the following data are entered?

```
1 Q1 = createQueue
2 S1 = createStack
3 loop (not end of file)
  1 read number
  2 if (number not 0)
    1 pushStack (S1, number)
  3 else
    1 popStack (S1, x)
    2 popStack (S1, x)
    3 loop (not empty S1)
      1 popStack (S1, x)
      2 enqueue (Q1, x)
    4 end loop
  4 end if
4 end loop
```

   The data are 5, 7, 12, 4, 0, 4, 6, 8, 67, 34, 23, 5, 0, 44, 33, 22, 6, 0.

6. Imagine that the contents of queue Q1 and queue Q2 are as shown. What would be the contents of Q3 after the following code is executed? The queue contents are shown front (left) to rear (right).

   Q1: 42 30 41 31 19 20 25 14 10 11 12 15

   Q2: 4 5 4 10 13

```
1 Q3 = createQueue
2 count = 0
3 loop (not empty Q1 and not empty Q2)
  1 count = count + 1
  2 dequeue (Q1, x)
  3 dequeue (Q2, y)
  4 if (y equal count)
    1 enqueue (Q3, x)
  5 end if
1 end loop
```

Problems

7. Using only the algorithms in the queue ADT, write an application algorithm called `copyQueue` that copies the contents of one queue to another.
8. It doesn't take much analysis to determine that the solution for Problem 7 is not very efficient. It would be much more efficient to write a new ADT method that copies a queue using its knowledge of the ADT. Rewrite Problem 7 as a new ADT algorithm.
9. Using only the algorithms in the queue ADT, write an algorithm called `catQueue` that concatenates two queues together. The second queue should be put at the end of the first queue.
10. Write a C function to implement Problem 9.
11. Write an algorithm called `stackToQueue` that creates a queue from a stack. After the queue has been created, the top of the stack should be the front of the queue and the base of the stack should be the rear of the queue. At the end of the algorithm, the stack should be empty.
12. Write an algorithm called `queueToStack` that creates a stack from a queue. At the end of the algorithm, the queue should be unchanged; the front of the queue should be the top of the stack, and the rear of the queue should be the base of the stack.
13. Write an algorithm that compresses a string by deleting all space characters in the string. One way to do so is to use a queue of characters. Insert nonspace characters from the string into the queue. When you reach the end of the string, dequeue the characters from the queue and place them back into the string.
14. Given a queue of integers, write an algorithm that, using only the queue ADT, calculates and prints the sum and the average of the integers in the queue without changing the contents of the queue.
15. Given a queue of integers, write an algorithm that deletes all negative integers without changing the order of the remaining elements in the queue.

16. Write an algorithm that reverses the contents of a queue.
17. Using only the algorithms in the queue ADT, write an algorithm that checks the contents of two queues and returns true if they are identical and false if they are not.
18. Implement Problem 17 as a C function.

Projects

19. Using the linked list abstract data type described in Section 4.3 "Queue ADT," write a menu-driven user interface to test each of the operations in the ADT. Any errors discovered during the processing should be printed as a part of the test result. A suggested menu is shown below.

```
A. Enqueue data into queue
B. Dequeue and print data
C. Print data at the front
D. Print data at the rear
E. Print entire queue
F. Print queue status: Empty
G. Print queue status: Full
H. Print number of elements and quit
```

Test your program with the following test cases. You may include additional test cases after you have executed the tests shown below.

a. Print queue status, Empty (F).
b. Dequeue and print data (B). Should return error.
c. Enqueue data into queue: 5 (A).
d. Enqueue data into queue: 8 (A).
e. Print queue status, Empty (F).
f. Print queue status, Full (G).
g. Print data at the front (C).
h. Print data at the rear (D).
i. Print entire queue (E).
j. Print number of elements in queue (H).
k. Dequeue and print data (B).
l. Dequeue and print data (B).
m. Dequeue and print data (B). Should return error.
n. Enqueue data into queue: 14 (A).
o. Print data at the front (C).
p. Print data at the rear (D).
q. Enqueue data into queue: 32 (A).
r. Print data at the front (C).
s. Print data at the rear (D).
t. Destroy queue and quit (I).

20. One way to evaluate a prefix expression is to use a queue. To evaluate the expression, scan it repeatedly until you know the final expression value. In each scan read the tokens and store them in a queue. In each scan replace an operator that is followed by two operands with their calculated values. For example, the following expression is a prefix expression that is evaluated to 159:

```
- + * 9 + 2 8 * + 4 8 6 3
```

We scan the expression and store it in a queue. During the scan when an operator is followed by two operands, such as + 2 8, we put the result, 10, in the queue.

After the first scan, we have

```
- + * 9 10 * 12 6 3
```

After the second scan, we have

```
- + 90 72 3
```

After the third scan, we have

```
- 162 3
```

After the fourth scan, we have

```
159
```

Write a C program to evaluate a prefix expression.

21. Write the C implementations for the saltwater taffy store described in "Queue Simulation," starting on page 175.

22. Using the C ADT, write a program that simulates the operation of a telephone system that might be found in a small business, such as your local pizza parlor. Only one person can answer the phone (a single-server queue), but there can be an unlimited number of calls waiting to be answered.

Queue analysis considers two primary elements, the length of time a requester waits for service (the queue wait time—in this case, the customer calling for pizza) and the service time (the time it takes the customer to place the order). Your program should simulate the operation of the telephone and gather statistics during the process.

The program requires two inputs to run the simulation: (1) the length of time in hours that the service is provided and (2) the maximum time it takes for the operator to take an order (the maximum service time).

Four elements are required to run the simulation: a timing loop, a call simulator, a call processor, and a start call function.

a. **Timing loop:** This is simply the simulation loop. Every iteration of the loop is considered 1 minute in real time. The loop continues until the service has been in operation the requested amount of time (see input above). When the operating period is complete, however, any waiting calls must be answered before ending the simulation. The timing loop has the following subfunctions:

   - Determine whether a call was received (call simulator)
   - Process active call
   - Start new call

   This sequence allows a call to be completed and another call to be started in the same minute.

b. **Call simulator:** The call simulator uses a random-number generator to determine whether a call has been received. Scale the random number to an appropriate range, such as 1 to 10.

   The random number should be compared with a defined constant. If the value is less than the constant, a call was received; if it is not, no call was received. For the simulation set the call level to 50%; that is, on the average, a call is received every 2 minutes. If a call is received, place it in a queue.

c. **Process active call:** If a call is active, test whether it has been completed. If completed, print the statistics for the current call and gather the necessary statistics for the end-of-job report.

d. **Start new call:** If there are no active calls, start a new call if there is one waiting in the queue. Note that starting a call must calculate the time the call has been waiting.

   During the processing, print the data shown in Table 4-2 after each call is completed. (*Note*: you will not get the same results.)

Clock time	Call number	Arrival time	Wait time	Start time	Service time	Queue size
4	1	2	0	2	3	2
6	2	3	2	5	2	4

TABLE 4-2 Sample Output for Project 22

At the end of the simulation, print out the following statistics gathered during the processing. Be sure to use an appropriate format for each statistic, such as a float for averages.

a. Total calls: calls received during operating period
b. Total idle time: total time during which no calls were being serviced
c. Total wait time: sum of wait times for all calls
d. Total service time: sum of service time for all calls
e. Maximum queue size: greatest number of calls waiting during simulation
f. Average wait time: total wait time/number of calls
g. Average service time: total service time/number of calls

Run the simulator twice. Both runs should simulate 2 hours. In the first simulation, use a maximum service time of 2 minutes. In the second run, use a maximum service time of 5 minutes.

23. Repeat the queue simulation in Project 22, using multiple queue servers such as you might find in a bank. There should be only one queue. The number of servers, to be read from the keyboard, may range from 2 to 5.

To simulate multiple servers, you need to provide a separate customer status structure for each server. In the processing loop, each server should be tested for completion. Similarly, in start new call, a call should be started for each idle server.

You also need to modify the call simulator to handle up to 3 customers arriving at the same time. To determine the number of customers, generate a random number in the range of 0 to 3. If the number is 0, no customer arrived. If the number is not 0, enqueue the number of customers indicated by the random number.

24. Write a stack and queue test driver. A test driver is a program created to test functions that are to be placed in a library. Its primary purpose is to completely test functions; therefore, it has no application use.

The functions to be tested are create stack, create queue, push stack, pop stack, enqueue, and dequeue. You may include other stack and queue functions as required. All data should be integers. You need two stacks and two queues in the program, as described below.

a. Input stack: used to store all user input
b. Input queue: used to store all user input
c. Output stack: used to store data deleted from input queue
d. Output queue: used to store data deleted from input stack

Use a menu-driven user interface that prompts the user to select either insert or delete. If an insert is requested, the system should prompt the user for the integer to be inserted. The data are then inserted into the input stack and input queue. If a delete is requested, the data are deleted from both structures: the data popped from the input stack are enqueued in the output queue, and the data dequeued from the input queue are pushed into the output stack.

Processing continues until the input structures are empty. At this point print the contents of the output stack while deleting all of its data. Label this output "Output Stack," then print all of the data in the output queue while deleting all of its data. Label this output "Output Queue." Your output should be formatted as shown below.

```
Output Stack: 18  9 13  7  5  1
Output Queue:  7 13  9 18  5  1
```

Test your program with the following operations:

```
1 input 1   5 delete     9 input 6    13 input 8
2 input 2   6 input 0   10 delete     14 delete
3 delete    7 input 5   11 input 7    15 delete
4 input 3   8 delete    12 delete     16 delete
```

In addition to the computer output from your test, write a short report (less than one page) describing what structural concepts were demonstrated by your output.

25. Queues are commonly used in network systems. For example, e-mail is placed in queues while it is waiting to be sent and after it arrives at the recipient's mailbox. A problem occurs, however, if the outgoing mail processor cannot send one or more of the messages in the queue. For example, a message might not be sent because the recipient's system is not available.

    Write an e-mail simulator that processes mail at an average of 40 messages per minute. As messages are received, they are placed in a queue. For the simulation assume that the messages arrive at an average rate of 30 messages per minute. Remember, the messages must arrive randomly, so you need to use a random-number generator to determine when messages are received (see "Queue Simulation," starting on page 175).

    Each minute, you can dequeue up to 40 messages and send them. Assume that 25% of the messages in the queue cannot be sent in any processing cycle. Again, you need to use a random number to determine whether a given message can be sent. If it can't be sent, put it back at the end of the queue (enqueue it).

    Run the simulation for 24 hours, tracking the number of times each message had to be requeued. At the end of the simulation, print the statistics that show:

    a. The total messages processed
    b. The average arrival rate
    c. The average number of messages sent per minute
    d. The average number of messages in the queue in a minute
    e. The number of messages sent on the first attempt, the number sent on the second attempt, and so forth
    f. The average number of times messages had to be requeued (do not include the messages sent the first time in this average)

Chapter 5
General Linear Lists

A general linear list is a list in which operations, such as retrievals, insertions, changes, and deletions, can be done anywhere in the list, that is, at the beginning, in the middle, or at the end of the list. We use many different types of general lists in our daily lives. We use lists of employees, student lists, and lists of our favorite songs. When we process our song list, we need to be able to search for a song, add a new song, or delete one we gave away. This chapter describes how we can maintain and process general lists. For simplicity, from now on we refer to general linear lists as *lists*.

5.1 Basic Operations

The four basic list operations are insertion, deletion, retrieval, and traversal. Insertion is used to add a new element to the list. Deletion is used to remove an element from the list. Retrieval is used to get the information related to an element without changing the structure of the list. Traversal is used to traverse the list while applying a process to each element.

Insertion

List **insertion** can be ordered or random. **Ordered lists** are maintained in sequence according to the data or, when available, a key that identifies the data. A **key** is one or more fields within a structure that identifies the data. Examples of keys are Social Security number and the universal product code on retail merchandise.

In **random lists** there is no sequential relationship between two elements. Although there are no restrictions on inserting data into a random list, computer algorithms generally insert data at the end of the list. Thus, random lists are sometimes called **chronological lists**. Applications that use random lists are found either in data-gathering applications, in which case the lists are

chronological lists, or in situations in which the applications require randomness, such as simulation studies or games.

Data must be inserted into ordered lists so that the ordering of the list is maintained. Maintaining the order may require inserting the data at the beginning or at the end of the list, but most of the time data are inserted somewhere in the middle of the list. To determine where the data are to be placed, computer scientists use a search algorithm. Insertions are graphically shown in Figure 5-1. The inserted data are identified by the shaded element; in this example it is the third element of the revised list.

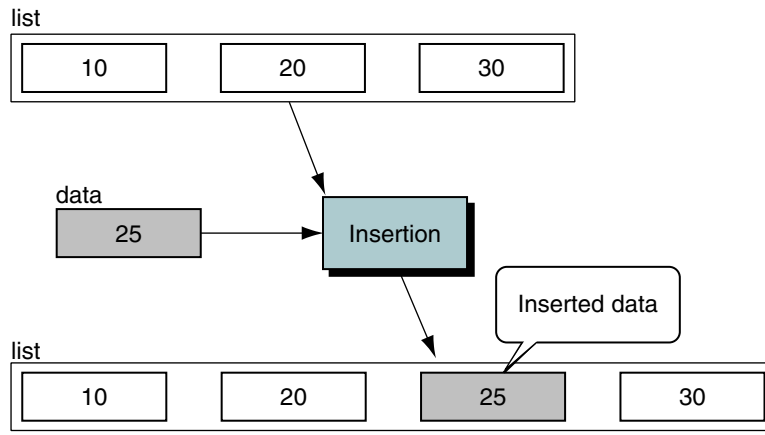


FIGURE 5-1 Insertion

Deletion

Deletion from a list requires that the list be searched to locate the data being deleted. Once located, the data are removed from the list. Figure 5-2 depicts a deletion from a list.

Retrieval

List **retrieval** requires that data be located in a list and presented to the calling module without changing the contents of the list. As with both insertion and deletion, a search algorithm can be used to locate the data to be retrieved from a list. Retrieving data from a list is shown in Figure 5-3.

Traversal

List **traversal** processes each element in a list in sequence. It requires a looping algorithm rather than a search. Each execution of the loop processes one element in the list. The loop terminates when all elements have been processed.

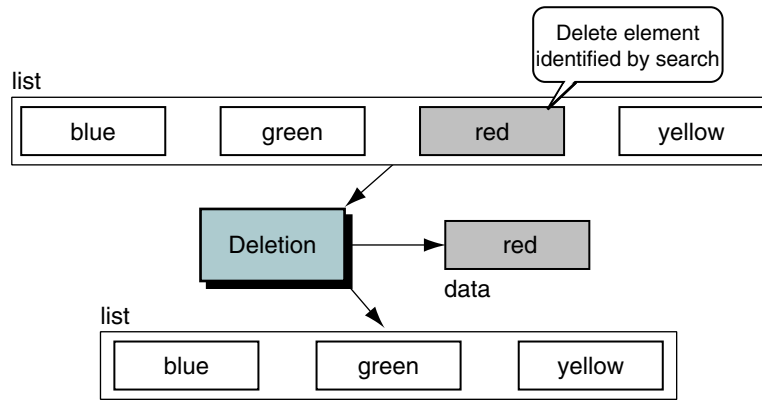


FIGURE 5-2 Deletion

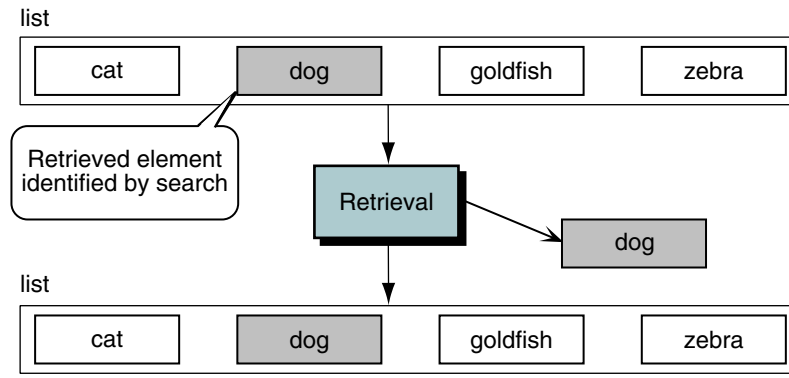


FIGURE 5-3 Retrieval

5.2 Implementation

Several data structures can be used to implement a list; we use a linked list. A linked list is a good structure for a list because data are easily inserted and deleted at the beginning, in the middle, or at the end of the list. Figure 5-4 shows the conceptual view of a list and its implementation as a linked list.

Data Structure

To implement a list, we need two different structures, a head node and data node, as shown in Figure 5-5.

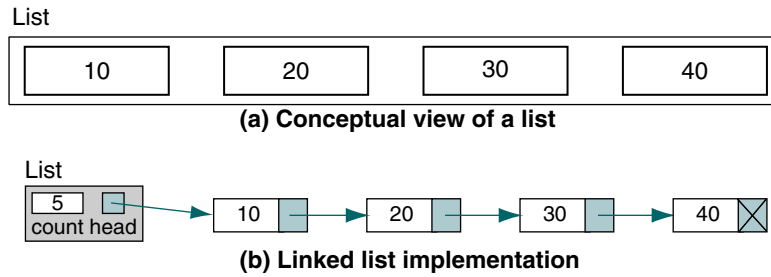


FIGURE 5-4 Linked List Implementation of a List

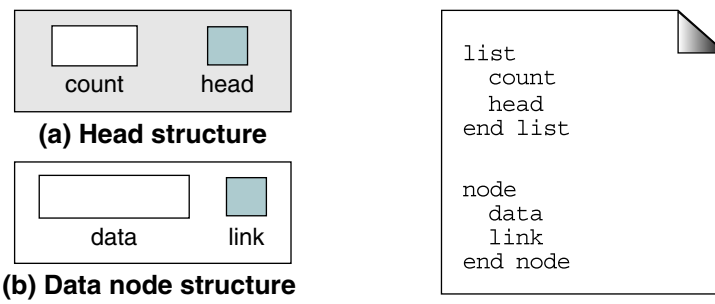


FIGURE 5-5 Head Node and Data Node

Head Node

Although only a single pointer is needed to identify the list, we often find it convenient to create a head node structure that stores the head pointer and other data about the list. When a node contains data about a list, the data are known as **metadata**; that is, they are data about the data in the list. For example, the head structure in Figure 5-4 contains one piece of metadata: `count`, an integer that contains the number of nodes currently in the list. Other metadata, such as the greatest number of nodes during the processing of the list, are often included when they serve a useful purpose.

Data Node

The data type for the list depends entirely on the application. A typical data type is shown below. The data types must be tailored to the particular application being created. We include a key field for applications that require searching by key.

```
data
   key
   field1
   field2
      ...
   fieldN
end data
```

Algorithms

We define 10 operations for a list, which should be sufficient to solve any problem. If an application requires additional list operations, they can be easily added. For each operation we define its name and provide a brief description and its calling parameters. We then develop algorithms for each operation.

Create List

Create list allocates the head structure and initializes the metadata for the list. At this time there are only two metadata entries; later we will add more to expand the capabilities of the list. Figure 5-6 shows the header before and after it is initialized by create list.

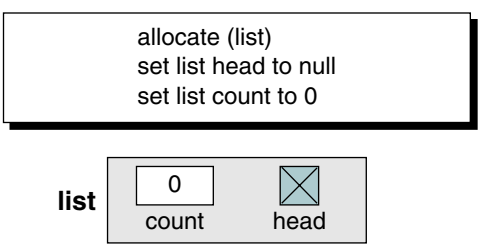


FIGURE 5-6 Create List

The pseudocode for create list is shown in Algorithm 5-1.

ALGORITHM 5-1 Create List

```
Algorithm createList (list)
Initializes metadata for list.
  Pre    list is metadata structure passed by reference
  Post   metadata initialized
1 allocate (list)
2 set list head  to null
3 set list count to 0
end createList
```

Insert Node

Insert node adds data to a list. We need only its logical predecessor to insert a node into the list. Given the predecessor, there are three steps to the insertion:

1. Allocate memory for the new node and move data to the node.
2. Point the new node to its successor.
3. Point the new node's predecessor to the new node.

These steps appear to be simple, but a little analysis is needed to fully understand how to implement them. To insert a node into a list, we need to know the location of the node that precedes the new node. This node is identified by a predecessor pointer that can be in one of two states: it can contain the address of a node or it can be null. When the predecessor pointer is null, it means that there is no predecessor to the data being added. The logical conclusion is that we are either adding to an empty list or are at the beginning of the list. If the predecessor is not null, we are adding somewhere after the first node—that is, in the middle of the list or at the end of the list. Let's discuss each of these situations in turn.

Insert into Empty List

When the head pointer of the list is null, the list is empty. This situation is shown in Figure 5-7. All that is necessary to add a node to an empty list is to assign the list head pointer the address of the new node and make sure that its link field is a null pointer. Although we could use a constant to set the link field of the new node, we use the null pointer contained in the list head. Why we use the list head null pointer will become apparent in the next section.

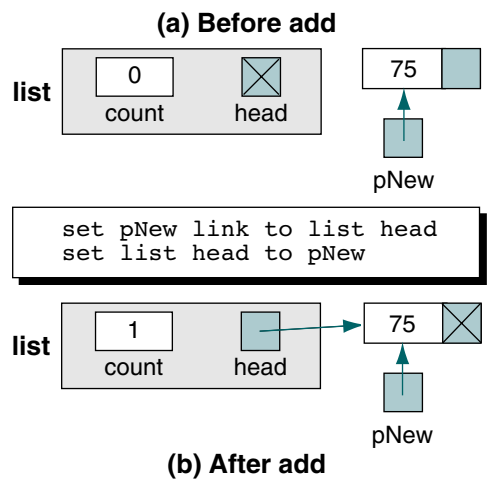


FIGURE 5-7 Add Node to Empty List

The pseudocode statements to insert a node into an empty list are shown here.

```
set pNew link to list head                (Null pointer)
set list head to pNew                     (First node)
```

Note the order of the statements. We must first point the new node to its successor; then we can change the head pointer. If we reverse these statements, we end up with the new node pointing to itself, which puts our program into a never-ending loop when we process the list.

Insert at Beginning

We add at the beginning of the list anytime we need to insert a node before the first node of the list. We determine that we are adding at the beginning of the list by testing the predecessor pointer. If it is a null pointer, there is no predecessor, so we are at the beginning of the list.

To insert a node at the beginning of the list, we simply point the new node to the first node of the list and then set the head pointer to point to the new first node. We know the address of the new node. The question at this point is how we can find the address of the first node currently in the list so that we can point the new node to it. The answer is simple: the first node's address is stored in the head pointer. The pseudocode statements to insert at the beginning of the list are shown here.

```
set pNew link to list head        (To current first node)
set list head to pNew             (To new first node)
```

If we compare these two statements with the statements to insert into an empty list, we will see that they are the same. They are the same because, logically, inserting into an empty list is the same as inserting at the beginning of a list. We can therefore use the same logic to cover both situations. Adding at the beginning of the list is shown in Figure 5-8.

Insert in Middle

When we add a node anywhere in the middle of the list, the predecessor pointer (`pPre`) contains an address. This case is shown in Figure 5-9.

To insert a node between two nodes, we point the new node to its successor and then point its predecessor to the new node. The address of the new node's successor can be found in the predecessor's link field. The pseudocode statements to insert a node in the middle of the list are shown here:

```
set pNew link to pPre link          (New to successor)
set pPre link to pNew               (Predecessor to new)
```

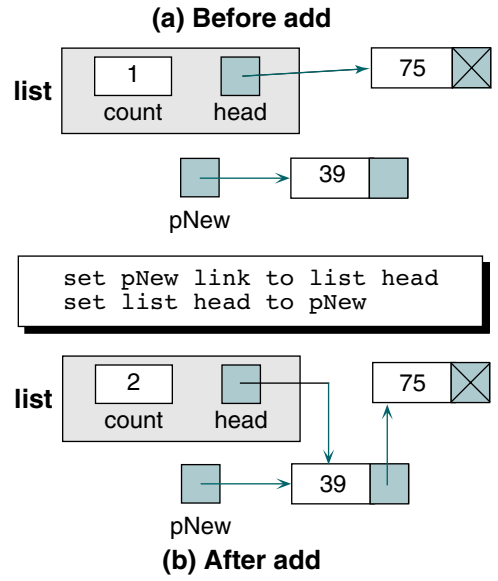


FIGURE 5-8 Add Node at Beginning

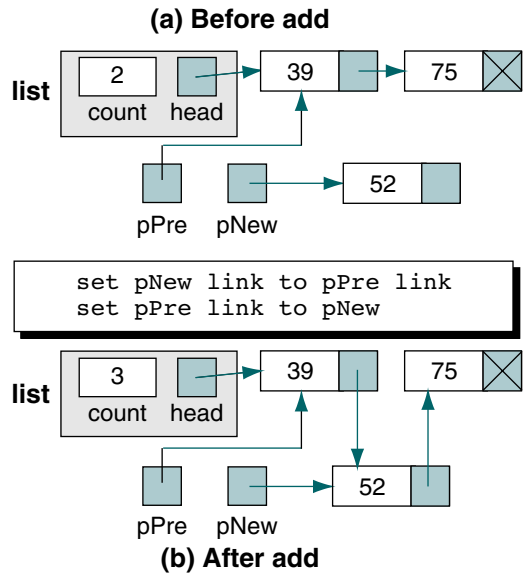


FIGURE 5-9 Add Node in Middle

Insert at End

When we are adding at the end of the list, we only need to point the predecessor to the new node. There is no successor to point to. It is necessary, however, to set the new node's link field to a null pointer. The statements to insert a node at the end of a list are shown here:

```
set pNew link to null pointer
set pPre link to pNew                (Predecessor to new)
```

Rather than have special logic in the algorithm for inserting at the end, we can take advantage of the existing linked list structure. We know that the last node in the list has a null link pointer. If we use this pointer rather than a null pointer constant, the code becomes exactly the same as the code for inserting in the middle of the list. The revised code follows. Compare it with the code for inserting in the middle of the list.

```
set pNew link to pPre link           (New to null)
set pPre link to pNew                (Predecessor to new)
```

Figure 5-10 shows the logic for inserting at the end of a list.

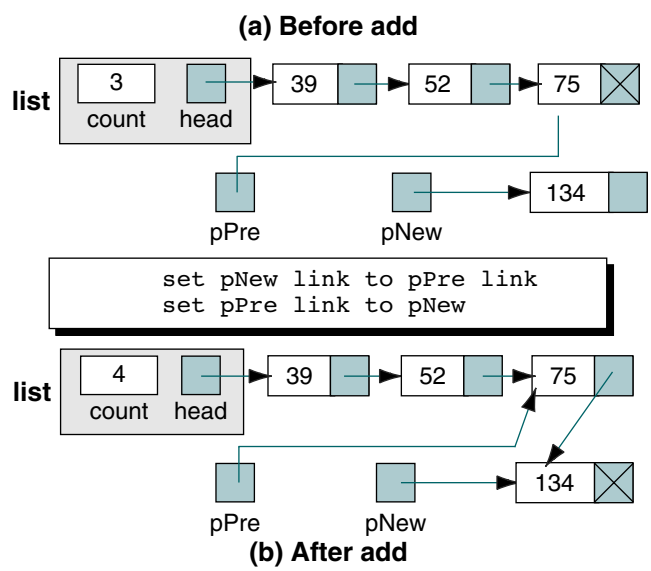


FIGURE 5-10 Add Node at End

Insert Node Algorithm

Now let's write the algorithm that puts it all together and inserts a node into the list. We are given a pointer to the list, the predecessor, and the data to be

inserted. We allocate memory for the new node and adjust the link pointers appropriately. When the algorithm is complete, it returns a Boolean—true if it was successful and false if there was no memory for the insert. The pseudocode is shown in Algorithm 5-2.

ALGORITHM 5-2 Insert List Node

```
Algorithm insertNode (list, pPre, dataIn)
Inserts data into a new node in the list.
  Pre    list is metadata structure to a valid list
         pPre is pointer to data's logical predecessor
         dataIn contains data to be inserted
  Post   data have been inserted in sequence
  Return true if successful, false if memory overflow
 1 allocate (pNew)
 2 set pNew data to dataIn
 3 if (pPre null)
   Adding before first node or to empty list.
   1  set pNew link to list head
   2  set list head to pNew
 4 else
   Adding in middle or at end.
   1  set pNew link to pPre link
   2  set pPre link to pNew
 5 end if
 6 return true
end insertNode
```

Algorithm 5-2 Analysis

We have discussed all of the logic in this algorithm except for the memory allocation. When memory is exhausted, the insert is in an **overflow** state. The action taken depends on the application. Although the application generally needs to abort the program, that decision should not be made in the insert algorithm. We therefore return a Boolean indicating whether we were successful and let the calling module decide whether it needs to abort or whether some other action is appropriate.

Delete Node

The **delete node** algorithm logically removes a node from the list by changing various link pointers and then physically deleting the node from dynamic memory. To logically delete a node, we must first locate the node itself. A delete node is located by knowing its address and its predecessor's address. We will discuss location concepts shortly. Once we locate the node to be deleted, we can simply change its predecessor's link field to point to the deleted node's successor. We then recycle the node back to dynamic memory. We need to be concerned, however, about deleting the only node in a list. Deleting the only node results in an empty list, and so we must be careful that in this case the head is set to a null pointer.

The delete situations parallel those for add. We can delete the only node, the first node, a node in the middle of the list, or the last node of a list. As we

explain below, these four situations reduce to only two combinations: delete the first node and delete any other node. In all cases the node to be deleted is identified by a pointer (`pLoc`).

Delete First Node

When we delete the first node, we must reset the head pointer to point to the first node's successor and then recycle the memory for the deleted note. We can tell we are deleting the first node by testing the predecessor. If the predecessor is a null pointer, we are deleting the first node. This situation is diagrammed in Figure 5-11.

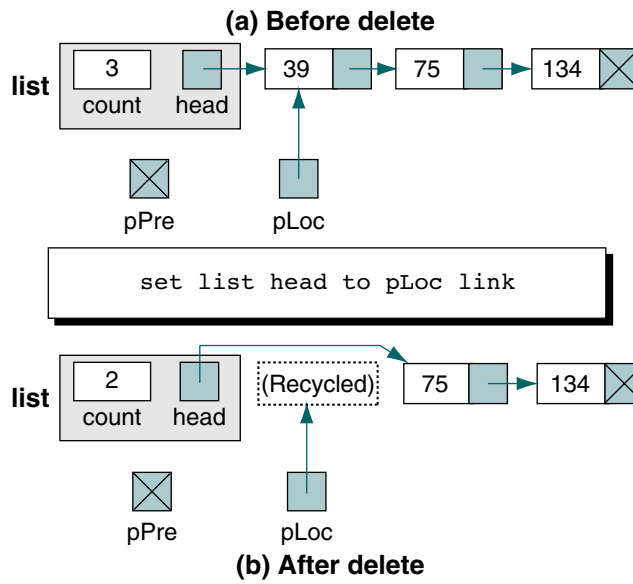


FIGURE 5-11 Delete First Node

The statements to delete the first node are shown in the next example. `Recycle` is the pseudocode command to return a node's space to dynamic memory.

```
set list head to pLoc link
recycle (pLoc)
```

If we examine this logic carefully, we note that it also applies when we are deleting the only node in the list. If the first node is the only node, then its link field is a null pointer. Because we move its link field (a null pointer) to the head pointer, the result is by definition an empty list.

General Delete Case

We call deleting any node other than the first node a general case because the same logic applies to deleting any node either in the middle or at the end of the list. For both of these cases, we simply point the predecessor node to the successor of the node being deleted. The logic is shown in Figure 5-12.

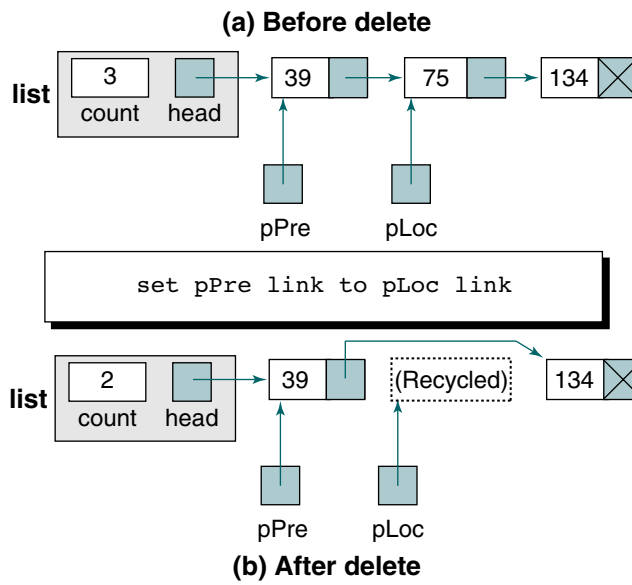


FIGURE 5-12 Delete General Case

We delete the last node automatically. When the node being deleted is the last node of the list, its null pointer is moved to the predecessor's link field, making the predecessor the new logical end of the list. After the pointers have been adjusted, the current node is recycled.

The delete general case pseudocode is shown here:

```
set pPre link to pLoc link
recycle (pLoc)
```

Delete Node Algorithm

The logic to delete a node is shown in Algorithm 5-3. The algorithm is given a pointer to the list, to the node to be deleted, and to the delete node's predecessor. It copies the deleted node's data to a data out area in the calling program and then adjusts the pointers before releasing the node's memory.

ALGORITHM 5-3 List Delete Node

```
Algorithm deleteNode (list, pPre, pLoc, dataOut)
Deletes data from list & returns it to calling module.
  Pre  list is metadata structure to a valid list
       Pre is a pointer to predecessor node
       pLoc is a pointer to node to be deleted
       dataOut is variable to receive deleted data
  Post data have been deleted and returned to caller
1 move pLoc data to dataOut
2 if (pPre null)
  Deleting first node
  1  set list head to pLoc link
3 else
  Deleting other nodes
  1  set pPre link to pLoc link
4 end if
5 recycle (pLoc)
end deleteNode
```

Algorithm 5-3 Analysis

We need to discuss two points in this algorithm. The first, and most important, is that the node to be deleted must be identified before it is called. The algorithm assumes that the predecessor and current pointers are properly set. If they aren't, the algorithm will most likely fail. Even if it doesn't fail, the data will be wrong. (It's better that the algorithm fail than that it report invalid results.)

Second, when we discussed the individual logic cases earlier, we placed the recycle statement after each delete statement. In the implementation we moved it to the end of the algorithm. When the same statements appear in both the true and false blocks of a selection statement, they should be moved out of the selection logic. Moving common statements is similar to factoring common expressions in algebra. The result is a program that is smaller and easier to maintain.

List Search

A list search is used by several algorithms to locate data in a list. To insert data, we need to know the logical predecessor to the new data. To delete data, we need to find the node to be deleted and identify its logical predecessor. To retrieve data from a list, we need to search the list and find the data. In addition, many user applications require that lists be searched to locate data.

We must use a sequential search because there is no physical relationship among the nodes. The classic sequential search[1] returns the location of an element when it is found and the address of the last element when it is not found. Because our list is ordered, we need to return the location of the element when it is found and the location *where it should be placed* when it is not found. Knuth[2] calls this search "sequential search in ordered table." We simply call it an ordered list search.

---

1. See Chapter 13.
2. Donald E. Knuth, *The Art of Computer Programming,* Vol. 3, *Sorting and Searching,* Second Edition (Reading, MA: Addison-Wesley, 1998), 398.

To search a list on a key, we need a key field. For simple lists the key and the data can be the same field. For more complex structures, we need a separate key field. We reproduce the data node structure from "Data Node" in Section 5-2 for your convenience:

```
data
   key
   field1
   field2
      ...
   fieldN
end data
```

Given a target key, the ordered list search attempts to locate the requested node in the list. If a node in the list matches the target value, the search returns true; if there are no key matches, it returns false. The predecessor and current pointers are set according to the rules in Table 5-1.

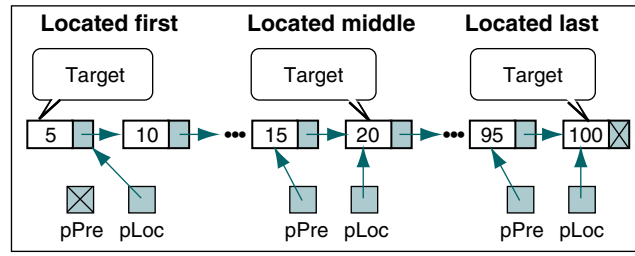
Condition	pPre	pLoc	Return
Target < first node	Null	First node	False
Target = first node	Null	First node	True
First < target < last	Largest node < target	First node > target	False
Target = middle node	Node's predecessor	Equal node	True
Target = last node	Last's predecessor	Last node	True
Target > last node	Last node	Null	False

TABLE 5-1 List Search Results

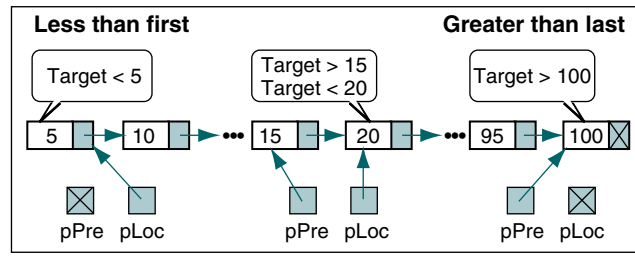
Each of these conditions is also shown in Figure 5-13.

We start at the beginning and search the list sequentially until the target value is no longer greater than the current node's key. At this point the target value is either less than or equal to the current node's key while the predecessor is pointing to the node immediately before the current node. We then test the current node and set the return value to true if the target value is equal to the list value or false if it is less (it cannot be greater) and terminate the search. The pseudocode for this search is shown in Algorithm 5-4.

In the previous discussion we assumed that the list was sequentially ordered on a key. It is often necessary, however, to search the list on a list attribute rather than on the key. For example, given a list of information about the employees in a company, we might want to find employees who have an engineering degree or employees who speak Japanese.



(a) Successful searches (return true)



(b) Unsuccessful searches (return false)

FIGURE 5-13 Ordered List Search

ALGORITHM 5-4 Search List

```
Algorithm searchList (list, pPre, pLoc, target)
Searches list and passes back address of node containing
target and its logical predecessor.
  Pre   list is metadata structure to a valid list
        pPre is pointer variable for predecessor
        pLoc is pointer variable for current node
        target is the key being sought
  Post  pLoc points to first node with equal/greater key
        -or- null if target > key of last node
        pPre points to largest node smaller than key
        -or- null if target < key of first node
  Return true if found, false if not found
1 set pPre to null
2 set pLoc to list head
3 loop (pLoc not null AND target > pLoc key)
  1  set pPre to pLoc
  2  set pLoc to pLoc link
4 end loop
5 if (pLoc null)
  Set return value
  1  set found to false
```

continued

ALGORITHM 5-4 Search List *(continued)*

```
 6 else
   1  if (target equal pLoc key)
      1  set found to true
   2  else
      1  set found to false
   3  end if
 7 end if
 8 return found
end searchList
```

Algorithm 5-4 Analysis

Examine the loop statement carefully. Note that there are two tests. The first test protects us from running off the end of the list; the second test stops the loop when we find the target or, if the target doesn't exist, when we find a node larger than the target. It is important that the null pointer test be done first. If the loop is at the end of the list, the current pointer is no longer valid. Testing the key first would give unpredictable results.[3]

We could make the search slightly more efficient if we had a rear pointer. A rear pointer is a metadata field that contains the address of the last node in the list. With a rear pointer, we could test the last node to make sure that the target wasn't larger than its key value. If the target were larger, we would simply skip the loop, setting the predecessor pointer to the last node and the current pointer to a null pointer. Once we know that the target is not greater than the last node, we don't need to worry about running off the end of the list.

To search a list on any field other than the key, we use a simple sequential search. The problem with nonkey searches, however, is that multiple elements often satisfy the search criteria. Although we might have only one employee who speaks Japanese, we might just as well have zero or many who do. One simple solution is to return a list of all elements that satisfy the criteria.

Retrieve Node

Now that we know how to locate a node in the list, we are ready to study retrieve node. Retrieve node uses search node to locate the data in the list. If the data are found, it moves the data to the output area in the calling module and returns true. If they are not found, it returns false. The pseudocode is shown in Algorithm 5-5.

Empty List

Processing logic often depends on there being data in a list. We provide **empty list**, a simple module that returns a Boolean indicating that there are data in the list or that it is empty (Algorithm 5-6).

3. Note that some languages, such as Pascal, test both expressions in the condition before evaluating the results. In these situations the loop limit test condition must be broken up into two separate expressions.

ALGORITHM 5-5 Retrieve List Node

```
Algorithm retrieveNode (list, key, dataOut)
Retrieves data from a list.
  Pre    list is metadata structure to a valid list
         key is target of data to be retrieved
         dataOut is variable to receive retrieved data
  Post   data placed in dataOut
          -or- error returned if not found
  Return true if successful, false if data not found
1 set found to searchList (list, pPre, pLoc, key)
2 if (found)
  1 move pLoc data to dataOut
3 end if
4 return found
end retrieveNode
```

ALGORITHM 5-6 Empty List

```
Algorithm emtpyList (list)
Returns Boolean indicating whether the list is empty.
  Pre    list is metadata structure to a valid list
  Return true if list empty, false if list contains data
1 if (list count equal 0)
  1 return true
2 else
  1 return false
end emptyList
```

Algorithm 5-6 Analysis

One of the questions often raised by students studying program design is "Why write a module when it contains only one statement—isn't it more efficient to simply code the statement inline?" Although it is definitely more efficient to write the code inline, it is not a better design for a generalized module. Remember that we are implementing generalized code in which the user does not necessarily know the data structure. For example, virtually all systems have an algorithm that tests for end of file. This test is quite simple, most likely only one line of code. It is necessary, however, because the programmer doesn't know the file structure and therefore cannot check the end-of-file status without support.

Full List

At first glance, **full list** appears to be as simple as empty list. As we saw with stacks, however, it turns out to be a relatively complex algorithm to implement. Very few languages provide the programmer with the capability to test how much memory is left in dynamic memory—C does not. The pseudocode is shown in Algorithm 5-7.

ALGORITHM 5-7 Full List

```
Algorithm fullList (list)
Returns Boolean indicating whether or not the list is full.
  Pre    list is metadata structure to a valid list
  Return false if room for new node; true if memory full
 1 if (memory full)
   1  return true
 2 else
   2  return false
 3 end if
 4 return true
end fullList
```

List Count

List count is another simple, one-line module. It is necessary because the calling module has no direct access to the list structure. Its implementation is shown in Algorithm 5-8.

ALGORITHM 5-8 List Count

```
Algorithm listCount (list)
Returns integer representing number of nodes in list.
  Pre    list is metadata structure to a valid list
  Return count for number of nodes in list
 1 return (list count)
end listCount
```

Traverse List

Algorithms that traverse a list start at the first node and examine each node in succession until the last node has been processed. Traversal logic is used by several different types of algorithms, such as changing a value in each node, printing the list, summing a field in the list, or calculating the average of a field. Any application that requires that the entire list be processed uses a traversal.

To traverse the list, we need a walking pointer, a pointer that moves from node to node as each element is processed. Assuming a list with a head structure, the following pseudocode uses a walking pointer to traverse the list. Each loop modifies the pointer to move to the next node in sequence as we traverse the list.

```
set pWalker to list head
loop (more nodes)
   process (pWalker data)
   set pWalker to next link
end loop
```

We begin by setting the walking pointer to the first node in the list. Then, using a loop, we continue until all of the data have been processed. Each loop calls a process module and passes it the data and then advances the walking pointer to the next node. When the last node has been processed, the loop terminates.

We have two possible approaches in designing the traverse list implementation. In one approach the user controls the loop, calling traverse to get the next element in the list. In the other approach, the traverse module controls the loop, calling a user-supplied algorithm to process the data. We implement the first option because it provides the programmer with more flexibility. For example, if the application needs to process only half the list, the loop can simply terminate. The second design would loop through the second half of the data unnecessarily. Figure 5-14 is a graphic representation of a list traversal.

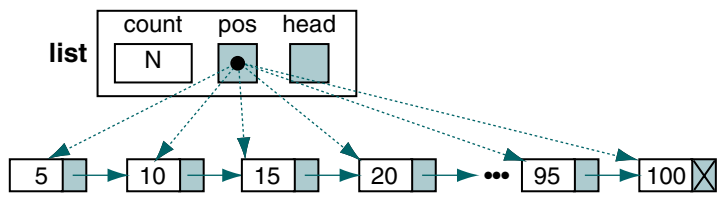


FIGURE 5-14 List Traversal

Because we need to remember where we are in the list from one call to the next, we need to add a current position pointer to the head structure. It keeps track of the node processed after the last call. The head structure is shown in Figure 5-14. The figure also shows how the position pointer is modified to move from one node to the next as the traversal algorithm is called.

Each call also needs to know whether we are starting from the beginning of the list or continuing from the last node processed. This information is communicated through the parameter list. The basic logic to traverse a list is shown in Algorithm 5-9. We name it `getNext` because it is called to get the next node in the traversal.

ALGORITHM 5-9 Traverse List

```
Algorithm getNext (list, fromWhere, dataOut)
Traverses a list. Each call returns the location of an
element in the list.
  Pre    list is metadata structure to a valid list
         fromWhere is 0 to start at the first element
         dataOut is reference to data variable
  Post   dataOut contains data and true returned
         -or- if end of list, returns false
```

continued

ALGORITHM 5-9 Traverse List *(continued)*

```
  Return true if next element located
         false if end of list
1 if (empty list)
  1  return false
2 if (fromWhere is beginning)
  Start from first
  1  set list pos to list head
  2  move current list data to dataOut
  3  return true
3 else
  Continue from pos
  1  if (end of list)
     End of List
     1  return false
   2 else
     1  set list pos to next node
     2  move current list data to dataOut
     3  return true
   3 end if
4 end if
end getNext
```

Algorithm 5-9 Analysis There are two major blocks of code in this algorithm. Statement 2 handles the processing when we start from the beginning of the list; statement 3 handles the processing when we continue from the current location. When we are starting at the beginning, we set the head structure's position pointer to the first node in the list and pass the first node's data back to the calling module.

When we are continuing the list traversal, we must ensure that there are more data in the list. If we are at the end of the list, we set success to false and terminate. If there are more data, we set the current position to the next node and pass its data back to the calling module.

One final word of caution: This design supports only one traversal of any given list at a time. If an application needs more than one traversal at a time, a different design is needed.

Destroy List

When a list is no longer needed but the application is not done, the list should be destroyed. **Destroy list** deletes any nodes still in the list and recycles their memory. It then sets the metadata to a null list condition. The code for destroy list is shown in Algorithm 5-10.

ALGORITHM 5-10 Destroy List

```
Algorithm destroyList (pList)
Deletes all data in list.
  Pre    list is metadata structure to a valid list
```

continued

ALGORITHM 5-10 Destroy List *(continued)*

```
  Post   All data deleted
1 loop (not at end of list)
  1  set list head to successor node
  2  release memory to heap
2 end loop
  No data left in list. Reset metadata.
3 set list pos to null
4 set list count to 0
end destroyList
```

5.3 List ADT

In this section we create a list ADT based on the linked list implementation. Recall that an ADT consists of a data type and the operations that manipulate the data. When we implement an ADT in C, the application data are controlled by the programmer. This requires that the application program allocate memory for the data and pass the data node's address to the ADT. Because C is strongly typed, the data node pointer must be passed as a *void* pointer. Using *void* pointers allows us to store the data node pointers in the ADT's data structure without knowing any of the details about the data.

But, being able to store the data node pointer does not solve all of our problems. Lists and many other data structures require that we be able to sequence the data. In a list, the data are generally stored in key sequence. The ADT does not have the necessary information to do this sequencing. Furthermore, each type of data requires different functions to compare two keys. The solution to this problem is found in pointers to functions. The application programmer writes a function to compare two keys in the data structure, for example to compare two integers, and passes the address of the compare function to the ADT. The ADT then stores the compare function address as metadata in the list head structure and uses it whenever data need to be compared. Before you continue you may want to review Chapter 1, "Generic Code for ADTs" and your C text for a discussion of *void* pointers and for pointers to functions.

For the list, we use a simple head structure that contains a count, several pointers, and the address of the compare function needed to compare the data in the list. The head structure is shown in Figure 5-15.

The data nodes contain the pointer to a data structure created by the programmer and a self-referential pointer to the next node in the list. These data structures are shown in Program 5-1. We will explain each of the structure variables as we use them.

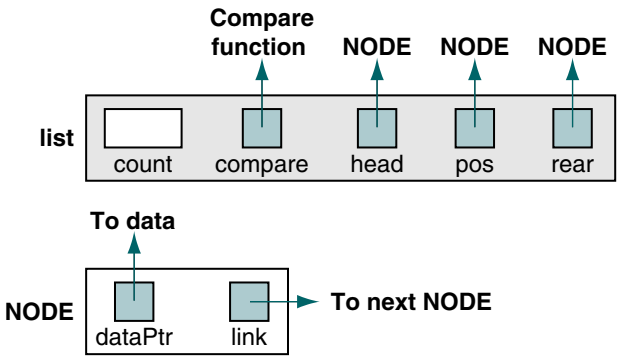


FIGURE 5-15 ADT Structure

PROGRAM 5-1 List ADT Type Definitions

```
1  //List ADT Type Defintions
2    typedef struct node
3       {
4        void*        dataPtr;
5        struct node* link;
6       } NODE;
7
8    typedef struct
9       {
10       int   count;
11       NODE* pos;
12       NODE* head;
13       NODE* rear;
14       int    (*compare) (void* argu1, void* argu2);
15      } LIST;
```

ADT Functions

The prototype declarations for the ADT functions are graphically described in Figure 5-16. They include all of the basic algorithms described in Section 5.2, "Algorithms."

Program 5-2 contains the prototype statements for the ADT. We discuss each function in the following sections. Note that the last three functions have identifiers that start with an underscore. These are internal functions that are not used in application code; they are for internal ADT use only. For this reason we have also declared them as static functions.[4] The code is shown in Program 5-2.

4. Static functions are not exported to the linker. They prevent internal functions from being used outside the ADT.

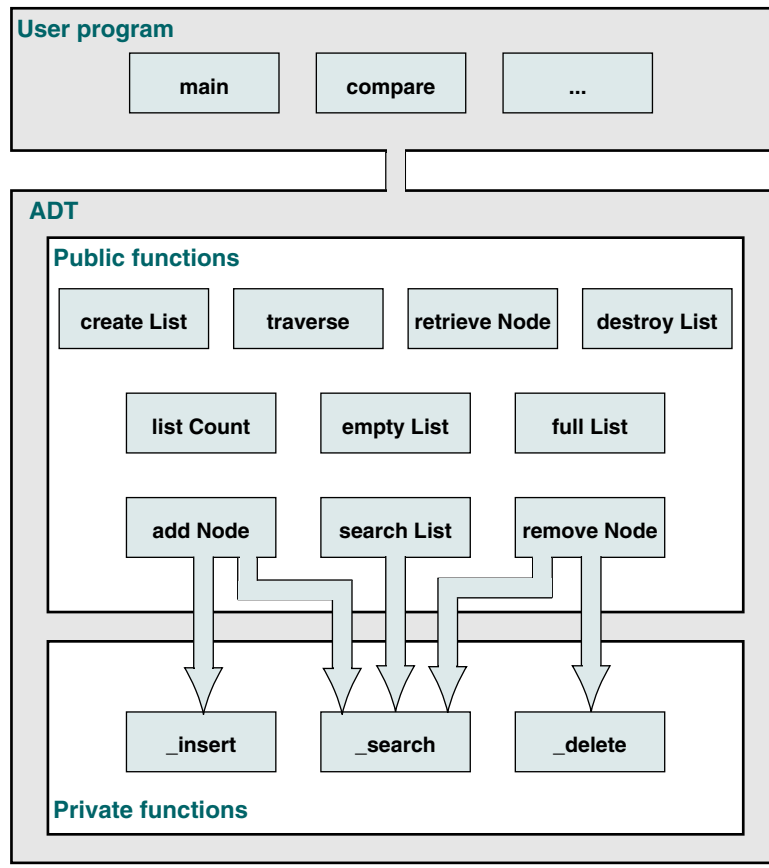


FIGURE 5-16 List ADT Functions

PROGRAM 5-2 List ADT Prototype Declarations

```
1  //Prototype Declarations
2    LIST* createList   (int (*compare)
3                        (void* argu1, void* argu2));
4    LIST* destroyList  (LIST* list);
5
6    int   addNode   (LIST* pList, void* dataInPtr);
7
8    bool  removeNode   (LIST*  pList,
9                        void*  keyPtr,
10                       void** dataOutPtr);
11
12   bool  searchList   (LIST*  pList,
13                       void*  pArgu,
14                       void** pDataOut);
```

continued

PROGRAM 5-2 List ADT Prototype Declarations *(continued)*

```
15
16    bool  retrieveNode (LIST*  pList,
17                        void*  pArgu,
18                        void** dataOutPtr);
19
20    bool  traverse     (LIST*  pList,
21                        int    fromWhere,
22                        void** dataOutPtr);
23
24    int   listCount    (LIST*  pList);
25    bool  emptyList    (LIST*  pList);
26    bool  fullList     (LIST*  pList);
27
28    static int _insert   (LIST* pList,
29                          NODE* pPre,
30                          void* dataInPtr);
31
32    static void _delete  (LIST*  pList,
33                          NODE*  pPre,
34                          NODE*  pLoc,
35                          void** dataOutPtr);
36    static bool _search  (LIST*  pList,
37                          NODE** pPre,
38                          NODE** pLoc,
39                          void*  pArgu);
40  //End of List ADT Definitions
```

Create List

The create list function creates the abstract data type in a null list state. Its only parameter is the compare function required to compare two keys in the data structure. It allocates memory for the head structure, initializes the pointers (and the list count, stores the address of the compare function for later use), and returns the address of the structure to the calling function. If the memory allocation fails, it returns a null pointer. The code is shown in Program 5-3.

PROGRAM 5-3 Create List

```
1  /*=============== createList ==============
2    Allocates dynamic memory for a list head
3    node and returns its address to caller
4       Pre    compare is address of compare function
5              used to compare two nodes.
6       Post   head has allocated or error returned
7       Return head node pointer or null if overflow
```

continued

PROGRAM 5-3 Create List *(continued)*

```
 8  */
 9  LIST* createList
10       (int (*compare) (void* argu1, void* argu2))
11  {
12  //Local Definitions
13    LIST* list;
14
15  //Statements
16    list = (LIST*) malloc (sizeof (LIST));
17    if (list)
18       {
19        list->head    = NULL;
20        list->pos     = NULL;
21        list->rear    = NULL;
22        list->count   = 0;
23        list->compare = compare;
24       } // if
25
26    return list;
27  } // createList
```

Program 5-3 Analysis With the exception of the compare function pointer, this algorithm is basic and straightforward. The pointer to the compare function passed to create list is simply the address of the function. When you examine the data structure in Program 5-1, however, note that the compare function is fully declared to receive two *void* pointers as its parameters and to return an integer. Should any of these types be declared incorrectly, the compiler returns an error message. This assures us that the programmer has correctly written the compare function to agree with our requirements.

Add Node

Add node is actually a higher-level, user-interface function that receives the data to be inserted into the list and searches the list for the insertion point. It then calls the low-level delete function, which parallels Algorithm 5-2, "Insert List Node."

In this ADT we have chosen to prevent duplicate keys from being inserted into the list. This leads to three possible return values: `-1` indicates dynamic memory overflow, `0` indicates success, and `+1` indicates a duplicate key. It is the using programmer's responsibility to correctly interpret these return values. If an application requires duplicate key insertions, the insertion and several of the other algorithms would need to be rewritten to handle duplicate data. The code is shown in Program 5-4.

Internal Insert Function

Internal insert function (`_insert`) is responsible for physically placing the new data into the list. Its code is shown in Program 5-5.

PROGRAM 5-4 Add Node

```
/*================== addNode =================
  Inserts data into list.
     Pre     pList is pointer to valid list
             dataInPtr pointer to insertion data
     Post    data inserted or error
     Return  -1 if overflow
             0 if successful
             1 if dupe key
*/
int addNode (LIST* pList, void* dataInPtr)
{
//Local Definitions
  bool found;
  bool success;

  NODE* pPre;
  NODE* pLoc;

//Statements
  found = _search (pList, &pPre, &pLoc, dataInPtr);
  if (found)
     // Duplicate keys not allowed
     return (+1);

  success = _insert (pList, pPre, dataInPtr);
  if (!success)
     // Overflow
     return (-1);
  return (0);
} // addNode
```

Program 5-4 Analysis

Add node begins by searching the structure to find the insertion position, as identified by the predecessor (**pPre**) and the current location (**pLoc**) pointers returned by the search. If the key of the new data matches a node in the list, the insertion is rejected and **+1** is returned. Assuming that the new data contains a unique key, the node is inserted.

PROGRAM 5-5 Internal Insert Function

```
/*================== _insert ==================
  Inserts data pointer into a new node.
     Pre    pList pointer to a valid list
            pPre  pointer to data's predecessor
            dataInPtr data pointer to be inserted
     Post   data have been inserted in sequence
     Return boolean, true  if successful,
                     false if memory overflow
```

continued

PROGRAM 5-5 Internal Insert Function *(continued)*

```
 9  */
10  static bool _insert (LIST* pList, NODE* pPre,
11                       void* dataInPtr)
12  {
13  //Local Definitions
14    NODE* pNew;
15
16  //Statements
17    if (!(pNew = (NODE*) malloc(sizeof(NODE))))
18       return false;
19
20    pNew->dataPtr   = dataInPtr;
21    pNew->link      = NULL;
22
23    if (pPre == NULL)
24       {
25        // Adding before first node or to empty list.
26        pNew->link       = pList->head;
27        pList->head      = pNew;
28        if (pList->count == 0)
29         // Adding to empty list. Set rear
30           pList->rear = pNew;
31       } // if pPre
32    else
33       {
34        // Adding in middle or at end
35        pNew->link  = pPre->link;
36        pPre->link  = pNew;
37
38       // Now check for add at end of list
39        if (pNew->link     == NULL)
40             pList->rear   =  pNew;
41       } // if else
42
43    (pList->count)++;
44    return true;
45  } // _insert
```

Remove Node

Remove node is also a high-level, user-interface function. It calls search list and delete node to complete the deletion. There are two possible completion states in remove node: either we were successful (true) or we were unsuccessful because the data to be deleted could not be found (false). The code is shown in Program 5-6.

PROGRAM 5-6 Remove Node

```
/*================= removeNode ================
  Removes data from list.
     Pre    pList pointer to a valid list
            keyPtr pointer to key to be deleted
            dataOutPtr pointer to data pointer
     Post   Node deleted or error returned.
     Return false not found; true deleted
*/
bool removeNode   (LIST*  pList, void*  keyPtr,
                   void** dataOutPtr)
{
//Local Definitions
  bool found;

  NODE* pPre;
  NODE* pLoc;

//Statements
  found = _search (pList, &pPre, &pLoc, keyPtr);
  if (found)
     _delete (pList, pPre, pLoc, dataOutPtr);

  return found;
} // removeNode
```

Program 5-6 Analysis While remove node is rather easy to follow, `dataOutPtr` must be cast as a *void* pointer in the calling function or you get a compile error. A typical call is shown here:

```
removeNode (pList, &partNum, (void*)&partPtr);
```

Internal Delete Function

Internal delete function (`_delete`) is called by remove node to physically delete the identified node from dynamic memory. When the data are deleted, a pointer to the data is returned to the calling function and placed in the variable location (`dataOutPtr`) specified by the last parameter in the call. The code is shown in Program 5-7.

PROGRAM 5-7 Internal Delete Function

```
/*================= _delete ================
  Deletes data from a list and returns
  pointer to data to calling module.
     Pre    pList pointer to valid list.
            pPre  pointer to predecessor node
            pLoc  pointer to target node
            dataOutPtr pointer to data pointer
```

continued

PROGRAM 5-7 Internal Delete Function *(continued)*

```
 8        Post   Data have been deleted and returned
 9               Data memory has been freed
10  */
11  void _delete (LIST* pList, NODE*  pPre,
12                NODE* pLoc,  void** dataOutPtr)
13  {
14  //Statements
15    *dataOutPtr = pLoc->dataPtr;
16    if (pPre == NULL)
17       // Deleting first node
18       pList->head = pLoc->link;
19    else
20       // Deleting any other node
21       pPre->link = pLoc->link;
22
23    // Test for deleting last node
24    if (pLoc->link == NULL)
25        pList->rear = pPre;
26
27    (pList->count)--;
28    free (pLoc);
29
30    return;
31  } // _delete
```

Search List

Search list is a high-level, user-interface function that locates a given node in the list and passes back its address and the address of its predecessor to the calling function. Because it is an application interface, it needs only three parameters: the list to be searched, the search argument, and the address of the pointer to receive the data pointer. To accomplish its function, it calls the internal ADT search function. Once the search is complete, it sets the output parameter to the address of the located data and returns the found Boolean. The code is shown in Program 5-8.

PROGRAM 5-8 Search User Interface

```
1   /*================== searchList =================
2     Interface to search function.
3        Pre    pList pointer to initialized list.
4               pArgu pointer to key being sought
5        Post   pDataOut contains pointer to found data
6          -or- NULL if not found
7        Return boolean true successful; false not found
8   */
```

continued

PROGRAM 5-8 Search User Interface *(continued)*

```
 9  bool searchList (LIST*  pList, void* pArgu,
10                   void** pDataOut)
11  {
12  //Local Definitions
13    bool  found;
14
15    NODE* pPre;
16    NODE* pLoc;
17
18  //Statements
19    found = _search (pList, &pPre, &pLoc, pArgu);
20    if (found)
21        *pDataOut = pLoc->dataPtr;
22    else
23        *pDataOut = NULL;
24    return found;
25  } // searchList
```

Internal Search Function

The actual search work is done with an internal search function (`_search`) available only within the ADT. It uses the compare function created by the using programmer to determine if the search argument is equal or not equal to the key in a node. The search logic is shown in Program 5-9.

PROGRAM 5-9 Internal Search Function

```
 1  /*================== _search ================
 2    Searches list and passes back address of node
 3    containing target and its logical predecessor.
 4       Pre    pList pointer to initialized list
 5              pPre  pointer variable to predecessor
 6              pLoc  pointer variable to receive node
 7              pArgu pointer to key being sought
 8       Post   pLoc points to first equal/greater key
 9         -or- null if target > key of last node
10              pPre points to largest node < key
11         -or- null if target < key of first node
12        Return boolean true found; false not found
13
14  */
15  bool _search (LIST*  pList, NODE** pPre,
16                NODE** pLoc,  void*  pArgu)
17  {
18  //Macro Definition
19  #define COMPARE \
20    ( ((* pList->compare) (pArgu, (*pLoc)->dataPtr)) )
```

continued

PROGRAM 5-9 Internal Search Function *(continued)*

```
21
22  #define COMPARE_LAST \
23    ((* pList->compare) (pArgu, pList->rear->dataPtr))
24
25  //Local Definitions
26    int result;
27
28  //Statements
29    *pPre  = NULL;
30    *pLoc  = pList->head;
31    if (pList->count == 0)
32        return false;
33
34    // Test for argument > last node in list
35    if ( COMPARE_LAST > 0)
36       {
37        *pPre = pList->rear;
38        *pLoc = NULL;
39        return false;
40       } // if
41
42    while ( (result = COMPARE) > 0 )
43       {
44        // Have not found search argument location
45        *pPre = *pLoc;
46        *pLoc = (*pLoc)->link;
47       } // while
48
49    if (result == 0)
50       // argument found--success
51       return true;
52    else
53       return false;
54  } // _search
```

Program 5-9 Analysis

You need to study several aspects of this function carefully. First, study the two macros we define at the beginning of the function. The compare code is quite long and tends to confuse the logic flow in the code when it is coded in line. By coding it as a macro, the function logic is much easier to follow. We need two different macros because we use two different pointers to nodes in the list.

Now let's study the compare closely (shown again here):

```
(* pList->compare) (pArgu, (*pLoc)->dataPtr)
```

The first expression in the compare is the address of the compare function written by the application program and stored in the ADT head when it was created. This is the equivalent of the function name. We get the address through the list header, **`pList->compare`**. The second parenthetical expression is the parameter list for the function.

In this case it contains a *void* pointer to the search argument (**pArgu**) and a *void* pointer to the data node (**dataPtr**). Because the compare function knows the data structure for the node, it can cast the pointers appropriately. The compare function returns **-1** when a search argument is less than the list key, **0** for equal, or **+1** for greater. This is the same design that C uses for string compares, so you are already familiar with it.

In the first function statement, we use the compare to determine if the search argument is greater than the last key in the list, identified by **rear** in Program 5-1. If it is, there is no need to search the list. If the search argument is less than or equal to the last entry, the compare in the *while* loop searches from the beginning of the list until the search argument is not greater than the list argument. At that point we have either found a matching key or the smallest key greater than the search argument.

Retrieve Node

Retrieve node searches the list and returns the address of the data whose key matches the search argument. Its code is shown in Program 5-10.

PROGRAM 5-10 Retrieve Node

```
/*================== retrieveNode ================
  This algorithm retrieves data in the list without
  changing the list contents.
     Pre    pList pointer to initialized list.
            pArgu pointer to key to be retrieved
     Post   Data (pointer) passed back to caller
     Return boolean true success; false underflow
*/
static bool retrieveNode (LIST*  pList,
                          void*  pArgu,
                          void** dataOutPtr)
{
//Local Definitions
  bool  found;

  NODE* pPre;
  NODE* pLoc;

//Statements
  found = _search (pList, &pPre, &pLoc, pArgu);
  if (found)
     {
      *dataOutPtr = pLoc->dataPtr;
      return true;
     } // if

  *dataOutPtr = NULL;
  return false;
} // retrieveNode
```

Empty List

Because the application programmer does not have access to the list structure, we provide three status functions that can be used to determine the list status. The first, empty list, is shown in Program 5-11.

PROGRAM 5-11 Empty List

```
/*================= emptyList ================
  Returns boolean indicating whether or not the
  list is empty
     Pre    pList is a pointer to a valid list
     Return boolean true empty; false list has data
*/
bool emptyList (LIST* pList)
{
//Statements
  return (pList->count == 0);
} // emptyList
```

Full List

The second status function, **full list**, determines if there is enough room in dynamic memory for another node. It is available for those applications that need to know if there is room available before an insert. The code is shown in Program 5-12.

PROGRAM 5-12 Full List

```
/*================== fullList =================
  Returns boolean indicating no room for more data.
  This list is full if memory cannot be allocated for
  another node.
     Pre    pList pointer to valid list
     Return boolean true if full
                    false if room for node
*/
bool fullList (LIST* pList)
{
//Local Definitions
NODE* temp;

//Statements
  if ((temp = (NODE*)malloc(sizeof(*(pList->head)))))
     {
      free (temp);
      return false;
     } // if

```

continued

PROGRAM 5-12 Full List *(continued)*

```
21    // Dynamic memory full
22    return true;
23
24  } // fullList
```

Program 5-12 Analysis This is one of the more difficult functions to write. Because C does not provide a facility to determine free space in dynamic memory, we can only allocate a node; if it works, we know there is room for at least one more node. But, as the allocation process may use the last available space, we must free the node before we return. Of course, if the allocation fails, we know there is no more room in dynamic memory.

List Count

List count is the last of the status functions. It returns an integer count for the number of nodes currently in the list. Its code is shown in Program 5-13.

PROGRAM 5-13 List Count

```
1   /*================== listCount ==================
2     Returns number of nodes in list.
3        Pre    pList is a pointer to a valid list
4        Return count for number of nodes in list
5   */
6   int listCount(LIST* pList)
7   {
8   //Statements
9
10    return pList->count;
11
12  } // listCount
```

Traverse

At one time or another, every list needs to be traversed. Because the programmer doesn't have access to the list structure, we need to provide a function to traverse the list. This algorithm is shown in Program 5-14.

PROGRAM 5-14 Traverse List

```
1   /*================== traverse =================
2     Traverses a list. Each call either starts at the
3     beginning of list or returns the location of the
4     next element in the list.
5        Pre    pList       pointer to a valid list
6               fromWhere   0 to start at first element
7               dataPtrOut  address of pointer to data
8        Post   if more data, address of next node
```

continued

PROGRAM 5-14 Traverse List *(continued)*

```
 9        Return true node located; false if end of list
10  */
11  bool traverse (LIST*  pList,
12                 int    fromWhere,
13                 void** dataPtrOut)
14  {
15  //Statements
16    if (pList->count == 0)
17        return false;
18
19    if (fromWhere == 0)
20       {
21        // Start from first node
22        pList->pos  = pList->head;
23        *dataPtrOut = pList->pos->dataPtr;
24        return true;
25       } // if fromwhere
26    else
27       {
28        // Start from current position
29        if (pList->pos->link == NULL)
30            return false;
31        else
32           {
33            pList->pos  = pList->pos->link;
34            *dataPtrOut = pList->pos->dataPtr;
35            return true;
36           } // if else
37       } // if fromwhere else
38  } // traverse
```

Program 5-14 Analysis

The traverse function needs to know if the traversal is just starting or if we are in the middle of a traversal. This logic is controlled through the **`fromWhere`** flag, a technique similar to the position flag in C's seek function. Each time the function is called, it stores the address of the current node being returned in the position pointer in the head node. Then the next time, if we are not starting from the beginning of the list, we can use the position pointer to locate the next node. If a node is available, we return true; if we are at the end of the list, we return false.

Destroy List

Destroy list is needed only for those situations in which a list needs to be deleted so that it can be built again. If the program is complete, it is not necessary to destroy the list before returning to the operating system. When called, destroy list deletes not only the nodes in the ADT but also all of the nodes allocated by the application programmer. At the end it deletes the list head node and returns a null pointer. The code is shown in Program 5-15.

PROGRAM 5-15 Destroy List

```
/*================== destroyList =================
  Deletes all data in list and recycles memory
     Pre    List is a pointer to a valid list.
     Post   All data and head structure deleted
     Return null head pointer
*/
LIST* destroyList (LIST* pList)
{
//Local Definitions
  NODE* deletePtr;

//Statements
  if (pList)
     {
      while (pList->count > 0)
         {
          // First delete data
          free (pList->head->dataPtr);

          // Now delete node
          deletePtr    = pList->head;
          pList->head  = pList->head->link;
          pList->count--;
          free (deletePtr);
         } // while
      free (pList);
     } // if
  return NULL;
} // destroyList
```

5.4 Application

To demonstrate how easily we can implement a list once we have an abstract data type, we implement a list of award-winning pictures and their directors. The program has three major functions: print instructions, build the list, and process user inquiries. In addition, the process function calls three functions: get the user's choice, print the entire list, and search for a requested year. One other function is required: a compare function to compare two years (keys). The complete design is shown in Figure 5-17.

Data Structure

The application data structure contains three fields: the year the movie was made, the name of the movie, and the name of the director. It is shown in Program 5-16.

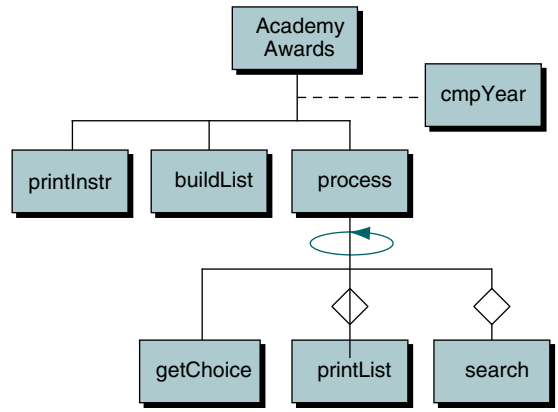


FIGURE 5-17 Academy Awards List Design

PROGRAM 5-16 Data Structure for Academy Awards

```
 1  /*Data Structure for Academy Awards
 2       Written by:
 3       Date:
 4  */
 5
 6  const short STR_MAX = 41;
 7
 8  typedef struct
 9    {
10     short   year;
11     char    picture [STR_MAX];
12     char    director[STR_MAX];
13    } PICTURE;
```

Application Functions

In this section we describe the functions and their interface with the abstract data type.

Mainline

Mainline defines the list variable and then calls the three functions that do the job. It has no interface with the abstract data type. Its code is shown in Program 5-17.

Print Instructions

Print instructions is a simple function that explains how the program works. Its code is shown in Program 5-18.

PROGRAM 5-17 Mainline for Academy Awards

```
/*This program maintains and displays a list of
  Academy Awards Motion Pictures.
     Written by:
     Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <cType.h>
#include <stdbool.h>
#include "P5-16.h"          // Data Structure
#include "linkListADT.h"

//Prototype Declarations
  void  printInstr (void);
  LIST* buildList  (void);
  void  process    (LIST* list);
  char  getChoice  (void);
  void  printList  (LIST* list);
  void  search     (LIST* list);

  int   cmpYear    (void* pYear1, void* pYear2);

int main (void)
{
// Local Definitions
   LIST* list;

// Statements
   printInstr ();
   list = buildList ();
   process (list);

   printf("End Best Pictures\n"
          "Hope you found your favorite!\n");
   return 0;
}  // main
```

PROGRAM 5-18 Print Instructions for User

```
/*================== printInstr =================
  Print instructions to user.
     Pre    nothing
     Post   instructions printed
*/
void printInstr (void)
{
//Statements
  printf("This program prints the Academy Awards \n"
```

continued

PROGRAM 5-18 Print Instructions for User *(continued)*

```
10      "Best Picture of the Year and its director.\n"
11      "Your job is to enter the year; we will do\n"
12      "the rest. Enjoy.\n");
13    return;
14  } // printInstr
```

Build List

Build list is one of the two major processing modules in the program. The data for the list are contained in a text file. This module reads the text file and inserts the data into the list. Program 5-19 contains the code for build list; Figure 5-18 diagrams its status after the second insert. As you study the figure, pay close attention to the pointers. First note how the list pointer in the program references the list header structure in the ADT. Then note how the header structure references the nodes in the list. Also note that each node references a different picture structure. Finally, note how the ADT can reference both nodes and pictures, but the application can reference only pictures.

PROGRAM 5-19 Build List

```
 1  /*==================== buildList ====================
 2    Reads a text data file and loads the list
 3    Pre    file exists in format: yy \t 'pic' \t 'dir'
 4    Post   list contains data
 5           -or- program aborted if problems
 6  */
 7  LIST* buildList (void)
 8  {
 9  //Local Definitions
10    FILE* fpData;
11    LIST* list;
12
13    short  yearIn;
14    int    addResult;
15
16    PICTURE* pPic;
17
18  //Statements
19    list   = createList (cmpYear);
20    if (!list)
21        printf("\aCannot create list\n"),
22              exit (100);
23    fpData = fopen("pictures.dat", "r");
24    if (!fpData)
25        printf("\aError opening input file\n"),
26              exit (110);
```

continued

PROGRAM 5-19 Build List *(continued)*

```
27    while (fscanf(fpData, " %hd", &yearIn) == 1)
28       {
29        pPic = (PICTURE*) malloc(sizeof(PICTURE));
30        if (!(pPic))
31            printf("\aOut of Memory in build list\n"),
32                   exit (100);
33        pPic->year = yearIn;
34
35        // Skip tabs and quote
36        while ((fgetc(fpData)) != '\t')
37           ;
38        while ((fgetc(fpData)) != '"')
39           ;
40        fscanf(fpData, " %40[^\"], %*c", pPic->picture);
41        while ((fgetc(fpData)) != '\t')
42           ;
43        while ((fgetc(fpData)) != '"')
44           ;
45        fscanf(fpData, " %40[^\"], %*c", pPic->director);
46
47        // Insert into list
48        addResult = addNode (list, pPic);
49        if (addResult != 0)
50            if (addResult == -1)
51              printf("Memory overflow adding movie\a\n"),
52                        exit (120);
53            else
54               printf("Duplicate year %hd not added\n\a",
55                        pPic->year);
56        while (fgetc(fpData) != '\n')
57              ;
58       } // while
59    return list;
60  } // buildList
```

Program 5-19 Analysis

Program 5-19 contains the first module that interfaces with the abstract data type. It has some interesting design aspects.

When we create the list, we pass the address of the function that compares two years. Remember, the ADT doesn't have visibility to the data types, so we must write and pass a function for any function that processes data.

Because the picture and director data both contain embedded spaces, we format the file with the year as a short integer and the picture title and the director as delimited strings. The delimiter for the picture title and the director is a double quote. This formatting allows us to use the *scanf* edit set format operator.

The loop is controlled by reading the year from the file. If there is no year, we are at the end of the file. If there is a year, we continue by reading the picture and the director in turn.

After reading the data into the picture variable structure, we insert it into the list. There are three possible results from add node. If the add is successful, add node returns 0. After calling add node, therefore, we check the results. If the return value is not 0, we check for memory overflow, in which case we exit the program; if there is a duplicate add, we just print an error message and continue.

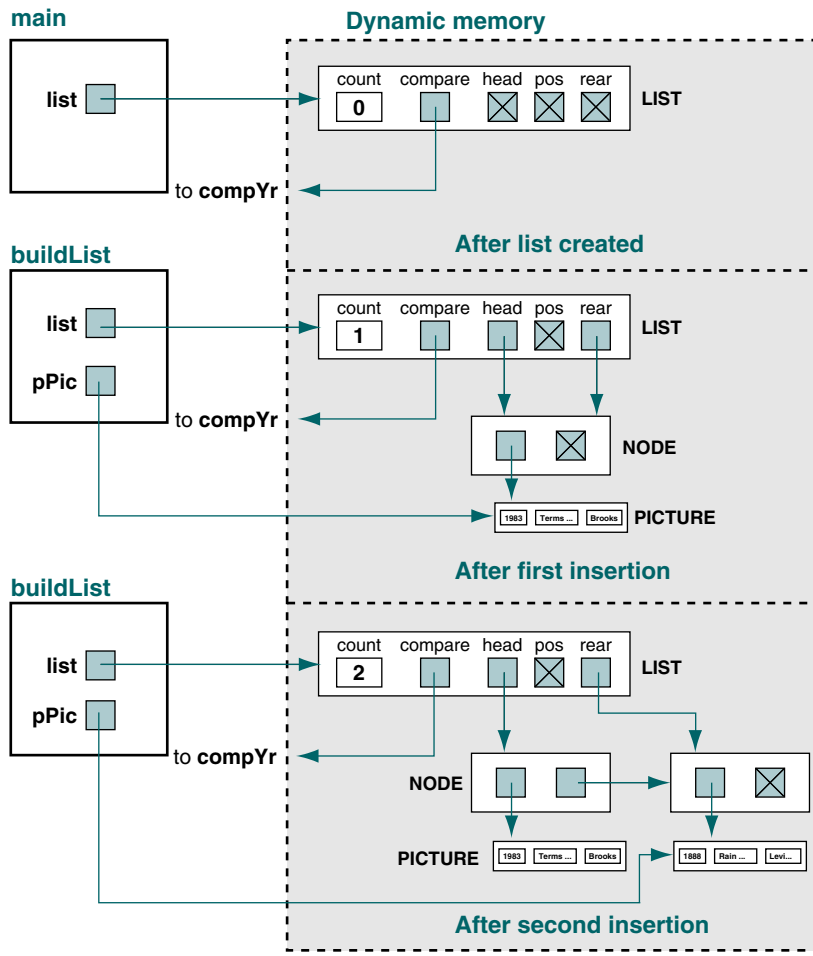


FIGURE 5-18 Build List Status after Second Insert

Process User Requests

After the list has been built, we are ready to process the user requests. The user has three options, as shown in a menu displayed by get choice: print the entire list, search for a particular year, or quit the program. The code for processing the list is shown in Program 5-20.

PROGRAM 5-20 Process User Choices

```
/*================== process ==================
  Process user choices
  Pre    list has been created
  Post   all of user's choice executed
*/
void process (LIST* list)
{
//Local Definitions
  char choice;

//Statements
  do
     {
      choice = getChoice ();

      switch (choice)
         {
          case 'P': printList (list);
                    break;
          case 'S': search (list);
          case 'Q': break;
         } // switch
     } while (choice != 'Q');
  return;
}  // process
```

Program 5-20 Analysis There are no calls to the abstract data type in this function. The function simply calls an application function to get the user's choice and then calls print list or search as appropriate.

Get User Choice

Get user choice is a simple function similar to many you have written. It reads the user's choice and ensures that it is valid. If the choice is not valid, the function loops until the user enters a correct choice. The code is shown in Program 5-21.

PROGRAM 5-21 Get User's Choice

```
/*================== getChoice ==================
  Prints the menu of choices.
     Pre    nothing
     Post   menu printed and choice returned
*/
char getChoice (void)
{
```

continued

PROGRAM 5-21 Get User's Choice *(continued)*

```
 8  //Local Definitions
 9    char choice;
10    bool valid;
11
12  //Statements
13    printf("======== MENU ======= \n"
14           "Here are your choices:\n"
15           "  S: Search for a year\n"
16           "  P: Print all years  \n"
17           "  Q: Quit             \n\n"
18           "Enter your choice: ");
19    do
20       {
21        scanf(" %c", &choice);
22        choice = toupper(choice);
23        switch (choice)
24           {
25            case 'S':
26            case 'P':
27            case 'Q': valid = true;
28                      break;
29            default:  valid = false;
30                      printf("\aInvalid choice\n"
31                             "Please try again: ");
32                      break;
33           } // switch
34       } while (!valid);
35    return choice;
36  } // getChoice
```

Print List

Print list traverses the list, printing one movie with each call to the abstract data type traverse function. We begin by checking to make sure that there are data in the list and printing an appropriate message if the list is empty. Once we have verified that the list contains data, we call get next to read the first movie. We then use a *do…while* loop to process the rest of the data. The code is shown in Program 5-22.

PROGRAM 5-22 Print List

```
1  /*================== printList ==================
2    Prints the entire list
3       Pre    list has been created
4       Post   list printed
5  */
6  void printList (LIST* list)
```

continued

PROGRAM 5-22 Print List *(continued)*

```
{
//Local Definitions
  PICTURE* pPic;

//Statements

  // Get first node
  if (listCount (list) == 0)
     printf("Sorry, nothing in list\n\a");
  else
     {
      printf("\nBest Pictures List\n");
     traverse (list, 0, (void**)&pPic);
      do
        {
         printf("%hd %-40s %s\n",
                pPic->year,      pPic->picture,
                pPic->director);
        } while (traverse (list, 1, (void**)&pPic));
     } // else
  printf("End of Best Pictures List\n\n");
} // printList
```

Program 5-22 Analysis

The interesting logic in this function relates to the traversing of the list. The traverse function in the abstract data type needs a "where from" flag to tell it to either start at the beginning of the list or continue from its last location. Therefore, before the loop we tell traverse that we want the first movie's data returned. Once we have a movie, we obtain the rest of the list using a loop. Because **`traverse`** returns a success flag, we can use it in the *do…while* expression. When all of the movies have been displayed, we display an end-of-list message and terminate.

Search List

The last function in our application allows the user to search for and display a movie for any year. The code is shown in Program 5-23.

PROGRAM 5-23 Search List

```
/*==================== search ====================
  Searches for year and prints year, picture, and
  director.
     Pre    list has been created
            user has selected search option
     Post   year printed or error message
*/
void search (LIST* list)
{
```

continued

PROGRAM 5-23 Search List *(continued)*

```
10  //Local Definitions
11    short    year;
12    bool     found;
13
14    PICTURE  pSrchArgu;
15    PICTURE* pPic;
16
17  //Statements
18    printf("Enter a four digit year: ");
19    scanf ("%hd", &year);
20    pSrchArgu.year = year;
21
22    found = searchList (list, &pSrchArgu,
23                        (void**)&pPic);
24
25    if (found)
26        printf("%hd %-40s %s\n",
27               pPic->year, pPic->picture, pPic->director);
28    else
29        printf("Sorry, but %d is not available.\n", year);
30    return;
31  } // search
```

Compare Year

As we discussed in the analysis for build list, we must write a compare function that compares two years. Our design is similar to the compare string function found in C. If the first year is less than the second, we return –1; if the two years are equal, we return 0; if the first year is greater than the second, we return +1. The code is shown in Program 5-24.

PROGRAM 5-24 Compare Year Function

```
1   /*==================== cmpYear ====================
2     Compares two years in PICTURE structures
3        Pre  year1 is a pointer to the first structure
4             year2 is a pointer to the second structure
5        Post two years compared and result returned
6     Return -1 if year1 less; 0 if equal; +1 greater
7   */
8   int cmpYear (void* pYear1, void* pYear2)
9   {
10  //Local Definitions
11    int   result;
12    short year1;
13    short year2;
14
```

continued

PROGRAM 5-24 Compare Year Function *(continued)*

```
15  //Statements
16    year1 = ((PICTURE*)pYear1)->year;
17    year2 = ((PICTURE*)pYear2)->year;
18
19    if (year1 < year2)
20        result = -1;
21    else if (year1 > year2)
22        result = +1;
23    else
24        result = 0;
25    return result;
26  } // cmpYear
```

Program 5-24 Analysis Compare year receives two picture structures and compares the key in the first to the key in the second. For simplicity we assign the years in the picture structure to local variables and then make the comparison. This step is unnecessary, but the resulting code is a little easier to understand.

Note that because the abstract data type passes a *void* pointer to the compare function, we must cast the structure as a picture type. If we did not do this, we get a compile error.

Testing Insert and Delete Logic

Testing list algorithms requires careful planning. We discuss testing insert logic and delete logic.

Testing Insert Logic

Because the list is ordered, we need at least four test cases to validate the insert logic:

1. *Insert a node into a null list*—This test is always done automatically because the list starts out in a null state.
2. *Insert a node before the first data node*—This test is not automatic; therefore, we need to arrange the input so that this case is tested.
3. *Insert between two data nodes*—Again, this test is not automatic. We need to make sure that the test cases include the insertion of at least one node between two existing nodes.
4. *Insert after the last node*—Because this case is the same as the insert into a null list, it is automatic. Nevertheless, we recommend a test case in which the new data are inserted after the last node in the list.

Testing Delete Logic

Testing the delete logic is similar to testing the insert logic:

1. *Delete to a null list*—To fully test the delete logic, one test case should delete all of the data from the list and then insert new data.

2. *Delete the first data node in the list*—When dealing with lists, the most common locations for errors are the first and last elements in the list. Make sure that one of the test cases deletes the first data node in the list.
3. *Delete a node between two data nodes*—Because this is the most common case, it is usually tested. We need to analyze the test data, however, to ensure that at least one of the delete cases deletes a node between two data nodes.
4. *Delete the node at the end of the list*—This test is not automatic. Make sure that one of the delete cases deletes the node at the end of the list.
5. *Try to delete a node that doesn't exist*—This test is not obvious. Some programs erroneously delete a node in the list even when the target node does not exist. It is thus very important that we also test the not-found conditions.
6. *Try to delete a node whose key is less than the first data node's key*—A subtle error in the program could result in the first node being deleted when the target does not exist. Make sure this case is tested.
7. *Try to delete a node whose key is greater than the last data node's key.*
8. *Try to delete from an empty list*—This is the opposite of the previous condition. It needs to be included in the test cases.

5.5 Complex Implementations

The implementation we have used to this point is known as a **singly linked list** because it contains only one link to a single successor. In this section we introduce three other useful implementations: the circularly linked list, the doubly linked list, and the multilinked list. They are not fully developed. To be included in the list ADT, they would need additional structures, such as a pointer to a compare function and a position pointer as well as additional functions.

Circularly Linked Lists

In a **circularly linked list** implementation, the last node's link points to the first node of the list, as shown in Figure 5-19. Circularly linked lists are primarily used in lists that allow access to nodes in the middle of the list without starting at the beginning. We will see one of these structures when we discuss the multilinked list.

Insertion into and deletion from a circularly linked list follow the same logic patterns used in a singly linked list except that the last node points to the first node. Therefore, when inserting or deleting the last node, in addition to updating the rear pointer in the header we must also point the link field to the first node.

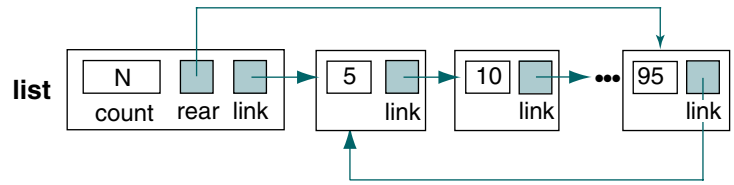


FIGURE 5-19 Circularly Linked List

Given that we can directly access a node in the middle of the list through its data structure, we are then faced with a problem when searching the list. If the search target lies before the current node, how do we find it? In a singly linked list implementation, when we arrive at the end of the list the search is complete. In a circular list implementation, however, we automatically continue the search from the beginning of the list.

The ability to continue the search presents another problem. What if the target does not exist? In the singly linked list implementation, if we didn't find the data we were looking for, we stopped when we hit the end of the list or when the target was less than the current node's data. With a circular list implementation, we save the starting node's address and stop when we have circled around to it, as shown in the code below:

```
loop (target not equal to pLoc key
  AND pLoc link not equal to startAddress)
```

Doubly Linked Lists

One of the most powerful implementations is the doubly linked list. A **doubly linked list** is a linked list structure in which each node has a pointer to both its successor and its predecessor. Figure 5-20 is a presentation of a doubly linked list.

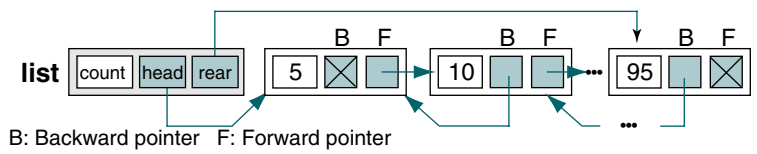


FIGURE 5-20 Doubly Linked List

There are three pieces of metadata in the head structure: a count, a position pointer for traversals, and a rear pointer. Although a **rear pointer** is not required in all doubly linked lists, it makes some of the list algorithms, such as insert and search, more efficient.

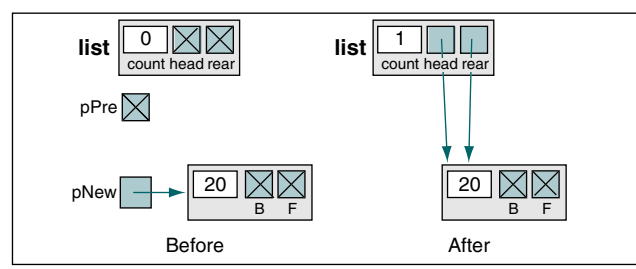
Each node contains two pointers: a **backward pointer** to its predecessor and a **forward pointer** to its successor. In Figure 5-20 these pointers are designated *B* and *F*, respectively.

Another variation on the doubly linked list is the doubly linked circularly linked list. In this variation the last forward pointer points to the first node of the list and the backward pointer of the first node points to the last node. If there is only one node in the list, both the forward and the backward pointers point to the node itself.

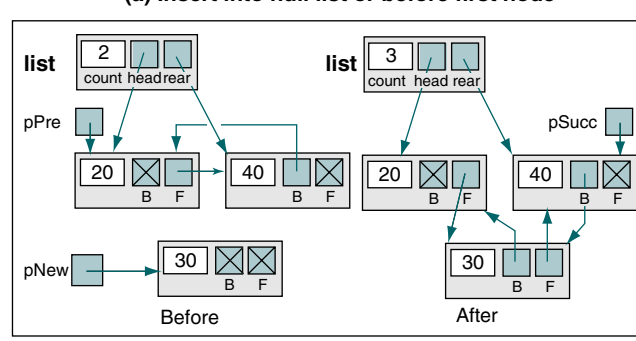
Insertion

Inserting follows the basic pattern of inserting a node into a singly linked list, but we also need to connect both the forward and the backward pointers.

A null doubly linked list's head and rear pointers are null. To insert a node into a null list, we simply set the head and rear pointers to point to the new node and set the forward and backward pointers of the new node to null. The results of inserting a node into a null list are shown in Figure 5-21(a).



(a) Insert into null list or before first node



(b) Insert between two nodes

FIGURE 5-21 Doubly Linked List Insert

Figure 5-21(b) shows the case for inserting between two nodes. The new node needs to be set to point to both its predecessor and its successor, and they need to be set to point to the new node. Because the insert is in the middle of the list, the head structure is unchanged.

Inserting at the end of the list requires that the new node's backward pointer be set to point to its predecessor. Because there is no successor, the forward pointer is set to null. The rear pointer in the head structure must also be set to point to the new rear node.

Algorithm 5-11 contains the code to insert a node into a doubly linked list.

ALGORITHM 5-11 Doubly Linked List Insert

```
Algorithm insertDbl (list, dataIn)
This algorithm inserts data into a doubly linked list.
  Pre    list is metadata structure to a valid list
         dataIn contains the data to be inserted
  Post   The data have been inserted in sequence
  Return 0: failed--dynamic memory overflow
         1: successful
         2: failed--duplicate key presented
 1 if (full list)
   1 return 0
 2 end if
   Locate insertion point in list.
 3 set found to searchList
        (list, predecessor, successor, dataIn key)
 4 if (not found)
   1 allocate new node
   2 move dataIn to new node
   3 if (predecessor is null)
     Inserting before first node or into empty list
     1 set new node back pointer to null
     2 set new node fore pointer to list head
     3 set list head to new node
   4 else
       Inserting into middle or end of list
     1 set new node fore pointer to predecessor fore pointer
     2 set new node back pointer to predecessor
   5 end if
     Test for insert into null list or at end of list
   6 if (predecessor fore null)
       Inserting at end of list--set rear pointer
     1 set list rear to new node
   7 else
       Inserting in middle of list--point successor to new
     1 set successor back to new node
   8 end if
   9 set predecessor fore to new node
   10 return 1
 5 end if
   Duplicate data. Key already exists.
 6 return 2
end insertDbl
```

Algorithm 5-11 Analysis

We must look at several points in this algorithm. First, rather than returning a simple success or failure, we have three different conditions and thus return three different values. If dynamic memory is full, we return a 0, indicating memory overflow. If we are successful, we return a 1. Finally, if the insert matches data already in the list, we return a 2.

The search algorithm provides the location of the target's logical predecessor and either: (1) the location of the node with a key that matches the target, (2) the location of the first node with a key greater than the target, or (3) null if the target is greater than the last node. We named the last parameter in the search successor because that was what we expected. (It could be confusing to the reader to use a pointer name that indicated we were looking for something other than the location of the successor.)

Finally, notice the comments. Although it is a short algorithm, the different conditions that can occur may be confusing. The comments should clarify the logic for the reader.

Deletion

Deleting requires that the deleted node's predecessor, if present, be pointed to the deleted node's successor and that the successor, if present, be set to point to the predecessor. This rather straightforward logic is shown in Figure 5-22. Once we locate the node to be deleted, we simply change its predecessor's and successor's pointers and recycle the node.

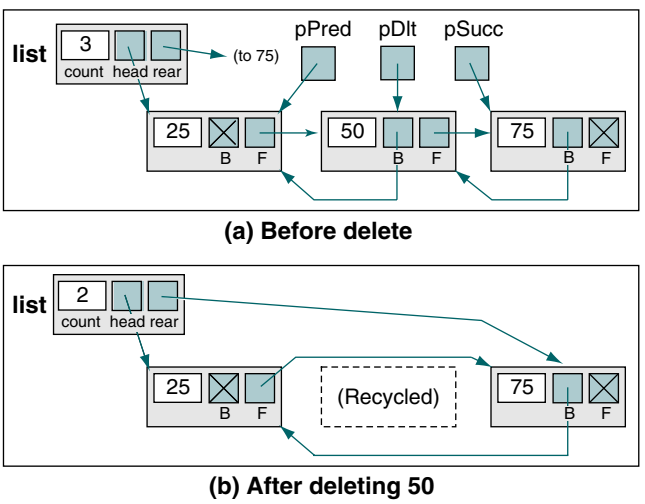


FIGURE 5-22 Doubly Linked List Delete

The pseudocode is shown in Algorithm 5-12.

ALGORITHM 5-12 Doubly Linked List Delete

```
Algorithm deleteDbl (list, deleteNode)
This algorithm deletes a node from a doubly linked list.
   Pre    list is metadata structure to a valid list
          deleteNode is a pointer to the node to be deleted
   Post   node deleted
 1 if (deleteNode null)
   1  abort ("Impossible condition in delete double")
 2 end if
 3 if (deleteNode back not null)
   Point predecessor to successor
   1  set predecessor      to deleteNode back
   2  set predecessor fore to deleteNode fore
 4 else
   Update head pointer
   1  set list head to deleteNode fore
 5 end if
 6 if (deleteNode fore not null)
   Point successor to predecessor
   1  set successor      to deleteNode fore
   2  set successor back to deleteNode back
 7 else
   Point rear to predecessor
   1  set list rear to deleteNode back
 8 end if
 9 recycle (deleteNode)
end deleteDbl
```

Algorithm 5-12 Analysis

Two points in this algorithm require further comment. First, the search was done in the calling algorithm. When we enter the doubly linked list delete algorithm, we already know the location of the node to be deleted.

Second, because it should never be called with a null delete pointer, we abort the program if we detect one. This is a logic error. Whenever a logic error is detected in a program, the program should be aborted and fixed.

Multilinked Lists

A **multilinked list** is a list with two or more logical key sequences. For example, consider the list of the first 10 presidents of the United States, shown in Table 5-2.

The data in Table 5-2 are listed chronologically by the date the president first assumed office (year). Two additional sequences could be of interest. The data could be ordered by the president's name or by his wife's name. Better yet, why not be able to traverse the list in any of these sequences? This is the power of the multilinked list: the same set of data can be processed in multiple sequences. It is important to understand that in a multilinked list the data are not replicated. The data exist only once, but multiple paths connect the one set of data.

President	Year	First lady
Washington, George	1789	Custis, Martha Dandridge
Adams, John	1797	Smith, Abigail
Jefferson, Thomas	1801	Skelton, Martha Wayles
Madison, James	1809	Todd, Dorothy Payne
Monroe, James	1817	Kortright, Elizabeth
Adams, John Quincy	1825	Johnson, Louisa Catherine
Jackson, Andrew	1829	Robards, Rachel Donelson
Van Buren, Martin	1837	Hoes, Hannah
Harrison, William H.	1841	Symmes, Anna
Tyler, John	1841	Christian, Letitia

TABLE 5-2 First 10 Presidents of the United States

To process the data in multiple sequences, we create a separate set of links for each sequence. Each link structure can be singly linked, doubly linked, or circularly linked. To demonstrate the concept, let's create a singly linked multilinked list of the presidents and their first ladies. Figure 5-23 diagrams the data for the first three presidents.

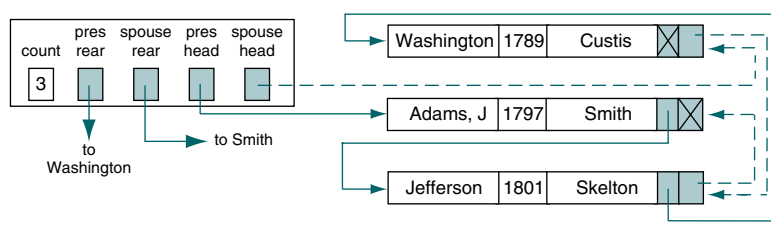


FIGURE 5-23 Multilinked List Implementation of Presidents List

Because there are two logical lists in the one physical list, we need two different link fields. Each node therefore contains two link fields: one for the president and one for the spouse. The president links are represented by solid lines. The spouse links are represented by dashed lines. If we follow the president links, we traverse the list through Adams, Jefferson, and Washington. If we follow the spouse links, we traverse the list through Custis (Mrs. Washington), Skelton (Mrs. Jefferson), and Smith (Mrs. Adams). Of course, in a real application there would be many more data fields. We use just three to represent the structure.

With minor adjustments the basic algorithms needed to maintain a multilinked list are the same as the algorithms for the singly linked or doubly linked lists.

Insert

Let's look first at the design to build a multilinked list. Using the president list as defined in Table 5-2, we see that there are two logical lists. Therefore when we add a node to the list, we need to insert it into each of the lists. We need to be careful, however, to make sure that we store the data only once. The design for the add node algorithm is shown in Figure 5-24.

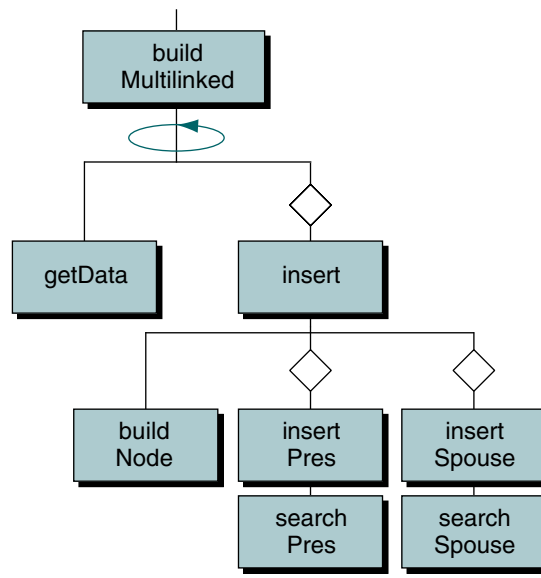


FIGURE 5-24 Multilinked List Add Node

This design assumes that the list is built from a file. The buildMultilinked module loops, reading the data for one president and inserting a node into the list. It then loops again, reading the data for the next president, and so on until the entire file has been read and the list built.

Each insert begins by allocating space for the data in `buildNode` and then building the president links in the insert president module and the spouse links in the insert spouse module. Because they are different logical lists with different keys, we use separate algorithms for each insertion and search.

There are three conditional calls in this design. We call insert as long as we successfully read data. If there is an error or when we get to the end of the file, we don't call it. Similarly, we call the inserts as long as the memory allocation in the build node module is successful. If it fails, then we cannot insert the data.

If you compare the insert design for the singly linked and doubly linked lists with the logic for the multilinked list, you will see one major difference: in the multilinked list insert, the memory allocation is separate from the searching and pointer connections because the memory is allocated only once. Another minor variation is the count increment, which should also be coded in the build node module. We add to the count only once because we are inserting only one physical node, even though we inserted it into two logical lists.

Delete

The major algorithm variation for the delete is that we need to reconnect the pointers for each logical list. Thus, if we delete a president's record, we need to adjust the spouse's as well as the president's successor pointer.

Assuming that we follow the president's pointers to delete a president, the delete logic follows the same pattern that the other linked list deletions use. But how do we delete the spouse? One alternative is to use the spouse's name from the president search and then search the spouse list to find the pointers that need to be updated. When the lists are long, doing so can be very inefficient.

The standard solution uses a doubly linked list for the spouse links. Having arrived in the middle of the spouse list through our search of the president list, we can easily set the spouse predecessor's pointers by following the backward pointer to the predecessor. This application is one of the more common uses of doubly linked lists.

5.6 Key Terms

chronological list
circularly linked list
create list
delete node
deletion
destroy list
doubly linked list
empty list
full list
insertion
key
metadata
multilinked list
ordered list
overflow
random list
rear pointer
retrieval
search list
singly linked list
traversal

5.7 Summary

- In a general linear list, data can be inserted and deleted anywhere, and there are no restrictions on the operations that can be used to process the list. In this chapter we refer to general linear lists simply as *lists*.
- Lists can be further divided into random lists and ordered lists. In a random list, there is no ordering of the data. In an ordered list, the data are arranged according to a key, which is one or more fields used to identify the data or control their use.
- Four common operations are associated with lists: insertion, deletion, retrieval, and traversal.
- Lists are usually implemented using linked lists.
- A head node is a data structure that contains metadata about the list, such as a count, a head pointer to the first node, and a rear pointer to the last node. It may contain any other data required by the use of the structure.
- When we want to insert into a list, we must consider four cases: adding to the empty list, adding at the beginning, adding to the middle, and adding at the end.
- When we want to delete a node from a list, we must consider two cases: delete the first node or delete any other node.
- To search a list for an item, we use the ordered list search.
- Traversing a list means going through the list, node by node, and processing each node. Three examples of list traversals are counting the number of nodes, printing the contents of nodes, and summing the values of one or more fields.
- A list can be implemented using a circularly linked list in which the last node's link points to the first node of the list.

- A list can be implemented using a doubly linked list in which each node has a pointer to both its successor and its predecessor.
- A list can be implemented using a multilinked list to create two or more logical lists.

5.8 Practice Sets

Exercises

1. Imagine we have the list shown in Figure 5-25 implemented as a linked list.

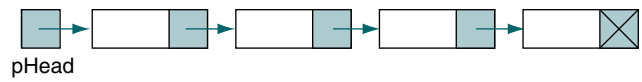


FIGURE 5-25 Linked List Implementation for Exercise 1

Show what happens if we use the following statement in a search of this list:

```
pHead = pHead->link
```

What is the problem with using this kind of statement? Does it justify the two walking pointers (`pPre` and `pLoc`) we introduced in the text?

2. Imagine we have the list shown in Figure 5-26 implemented as a linked list. As discussed in "List Search," in Section 5.2, the search needs to be able to pass back both the location of the predecessor (`pPre`) and the location of the current (`pLoc`) node based on search criteria.

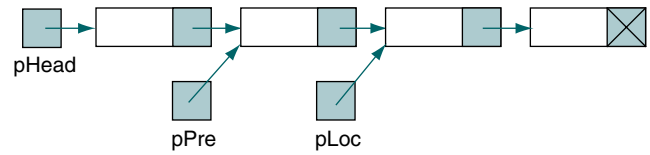


FIGURE 5-26 Linked List Implementation for Exercise 2

The following code to set `pPre` and `pLoc` contains a common error. What is it and how should it be corrected?

```
pLoc = pLoc->link
pPre = pPre->link
```

(*Hint:* What are the contents of these pointers at the beginning of the search?)

3. Imagine we implement a list using a dummy node at the beginning of the list. The dummy node does not carry any data. It is not the first data node, it is an empty node. Figure 5-27 shows a list with a dummy node.

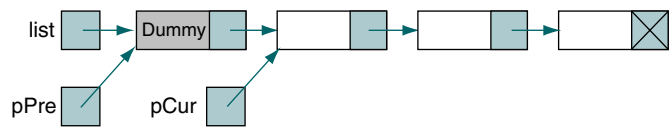


FIGURE 5-27 Linked List Implementation for Exercise 3

Write the code to delete the first node (the node after the dummy node) in the list.

4. Write the code to delete a node in the middle of a list implemented as a linked list with the dummy node (see Exercise 3). Compare your answer with the answer to Exercise 3. Are they the same? What do you conclude? Does the dummy node simplify the operation on a list? How?
5. Figure 5-28 shows an empty list with a dummy node. Write the code to add a node to this empty list.

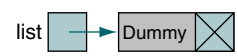


FIGURE 5-28 List for Exercise 5

6. Write the statements to add a node in the middle of a list with the dummy node (see Exercise 3). Compare your answer with the answer to Exercise 5. Are they the same? What do you conclude? Does the dummy node simplify the operation on a list? How?
7. Imagine we have the two lists shown in Figure 5-29. What would happen if we applied the following statement to these two lists?

```
listl = list2
```

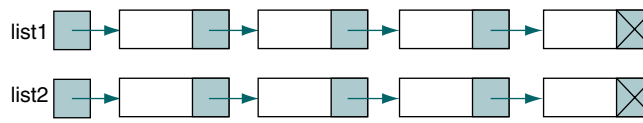


FIGURE 5-29 Linked Lists for Exercise 7

8. What would happen if we applied the following statements to the two lists in Exercise 7?

```
1 set temp to list1
2 loop (temp link not null)
  1 set temp to temp link
3 end loop
4 set temp link to list2
```

9. Imagine we have the list shown in Figure 5-30.

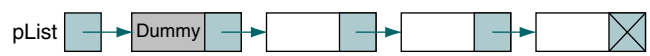


FIGURE 5-30 List Implementation for Exercise 9

What would happen if we applied the following statements to this list?

```
1 set temp to pList
2 loop (temp link not null)
  1 set temp to temp link
3 end loop
4 set temp link to pList
```

Problems

10. Write an algorithm that reads a list of integers from the keyboard, creates a list of them using linked list implementation, and prints the result.
11. Write an algorithm that accepts a list implemented using a linked list, traverses it, and returns the data in the node with the smallest key value.
12. Write an algorithm that traverses a list implemented using a linked list and deletes all nodes whose keys are negative.
13. Write an algorithm that traverses a list implemented using a linked list and deletes the node following a node with a negative key.

14. Write an algorithm that traverses a list implemented using a linked list and deletes the node immediately preceding a node with a negative key.
15. Write a program that creates a two-dimensional array using a linked list implementation. The nodes in the first column contain only two pointers, as shown in Figure 5-31. The left pointer points to the next row. The right pointer points to the data in the row.

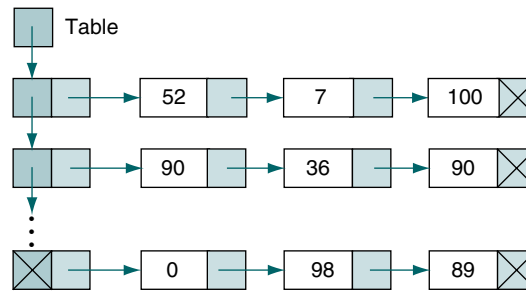


FIGURE 5-31 List for Problem 15

16. We can simplify most of the algorithms in the text using a linked list implementation with a dummy node at the beginning, as shown in Figure 5-32.

FIGURE 5-32 Implementation for Problem 16

Write an algorithm `insertNode` (see "Internal Insert Function," in Section 5.4) using a linked list implementation with a dummy node.

17. Write an algorithm `deleteNode` (Algorithm 5-3, "List Delete Node") using a linked list implementation with a dummy node.
18. Write an algorithm `searchList` (Algorithm 5-4, "Search List") using a linked list implementation with a dummy node.
19. Write an algorithm that returns a pointer to the last node in a linked list implementation of a list.
20. Write an algorithm that appends two lists together. Use linked list implementation.
21. Write an algorithm that appends a list to itself. Use linked list implementation.

22. Write an algorithm that swaps (exchanges) two nodes in a list. The nodes are identified by number and are passed as parameters. For example, to exchange nodes 5 and 8, you would call `swap (5, 8)`. If the exchange is successful, the algorithm is to return true. If it encounters an error, such as an invalid node number, it returns false. Use linked list implementation.
23. Write a new ADT algorithm to merge two lists. Use linked list implementation.

Projects

24. Write a program that reads a file and builds a list. After the list is built, display it on the monitor. You may use any appropriate data structure, but it should have a key field and data. Two possibilities are a list of your favorite CDs or your friends' telephone numbers. Use a linked list implementation.
25. Modify the program you wrote in Project 24. After you create the file, the program should present the user with a menu to insert new data, remove existing data, or print a list of all data.
26. Write a program to read a list of students from a file and create a list. The program should use a linked list for implementation. Each node in the linked list should have the student's name, a pointer to the next student, and a pointer to a linked list of scores. There may be up to four scores for each student. The structure is shown in Figure 5-33.

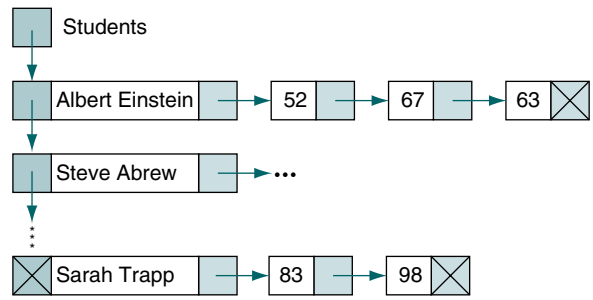


FIGURE 5-33 Data Structure for Project 26

The program should initialize the student list by reading the students' names from the text file and creating null score lists. It should then loop through the list, prompting the user to enter the scores for each student. The scores' prompt should include the name of the student.

After all scores have been entered, the program should print the scores for each student along with the score total and the average score. The average should include only those scores present.

The data for each student are shown in Table 5-3.

Student name	Score 1	Score 2	Score 3	Score 4
Albert Einstein	52	67	63	
Steve Abrew	90	86	90	93
David Nagasake	100	85	93	89
Mike Black	81	87	81	85
Andrew Dijkstra	90	82	95	87
Joanne Nguyen	84	80	95	91
Chris Walljasper	86	100	96	89
Fred Albert	70	68		
Dennis Dudley	74	79	77	81
Leo Rice	95			
Fred Flintstone	73	81	78	74
Frances Dupre	82	76	79	
Dave Light	89	76	91	83
Hua Tran	91	81	87	94
Sarah Trapp	83	98	94	93

TABLE 5-3 Data for Project 26

27. Modify Project 26 to insert the data into the student list in key (student name) sequence. Because the data are entered in a first name–last name format, you need to write a special compare algorithm that reformats the name into a last name–first name format and then does a string compare. All other algorithms should work as previously described.
28. Write a function that merges two ordered lists into one list. When two lists are merged,[5] the data in the resulting list are also ordered. The two original lists should be left unchanged; that is, the merged list should be a new list. Use linked list implementation.
29. Write a program that adds and subtracts polynomials. Each polynomial should be represented as a list with linked list implementation. The first node in the list represents the first term in the polynomial, the second node represents the second term, and so forth.

5. For more information about merge concepts, see Chapter 12, "Advanced Sort Concepts."

Each node contains three fields. The first field is the term's coefficient. The second field is the term's power, and the third field is a pointer to the next term. For example, consider the polynomials shown in Figure 5-34. The first term in the first polynomial has a coefficient of 5 and an exponent of 4, which is then interpreted as $5x^4$.

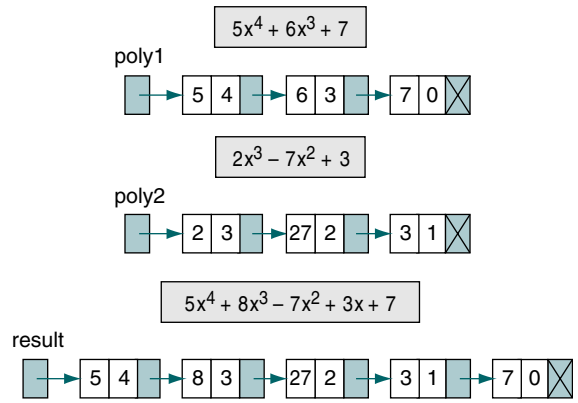


FIGURE 5-34 Example of Polynomials for Project 29

The rules for the addition of polynomials are as follows:

a. If the powers are equal, the coefficients are algebraically added.
b. If the powers are unequal, the term with the higher power is inserted in the new polynomial.
c. If the exponent is 0, it represents x^0, which is 1. The value of the term is therefore the value of the coefficient.
d. If the result of adding the coefficients results in 0, the term is dropped (0 times anything is 0).

A polynomial is represented by a series of lines, each of which has two integers. The first integer represents the coefficient; the second integer represents the exponent. Thus, the first polynomial in Figure 5-35 is

```
5    4
6    3
7    0
```

To add two polynomials, the program reads the coefficients and exponents for each polynomial and places them into a linked list. The input can be read from separate files or entered from the keyboard with appropriate user prompts. After the polynomials have been stored, they are added and the results are placed in a third linked list.

The polynomials are added using an operational merge process. An operational merge combines the two lists while performing one or more operations—in our case, addition. To add we take one term from each of the polynomials and compare the exponents. If the two exponents are equal, the coefficients are added to create a new coefficient. If the new coefficient is 0, the term is dropped; if it is not 0, it is appended to the linked list for the resulting polynomial. If one of the exponents is larger than the other, the corresponding term is immediately placed into the new linked list, and the term with the smaller exponent is held to be compared with the next term from the other list. If one list ends before the other, the rest of the longer list is simply appended to the list for the new polynomial.

Print the two input polynomials and their sum by traversing the linked lists and displaying them as sets of numbers. Be sure to label each polynomial.

Test your program with the two polynomials shown in Table 5-4.

Polynomial 1		Polynomial 2	
Coefficient	**Exponent**	**Coefficient**	**Exponent**
7	9	–7	9
2	6	2	8
3	5	–5	7
4	4	2	4
2	3	2	3
6	2	9	2
6	0	–7	1

TABLE 5-4 Text Data for Project 29

30. In older personal computers, the largest integer is 32,767 and the largest long integer is 2,147,483,647. Some applications, such as cryptography and security algorithms, may require an unbounded integer. One way to store and manipulate integers of unlimited size is by using a linked list. Each digit is stored in a node of the list. For example, Figure 5-35 shows how we could store a five-digit number in a list.

Although the list in Figure 5-35 is represented as moving from right to left, there is no physical direction in a list. We represent it in this way to clarify the problem.

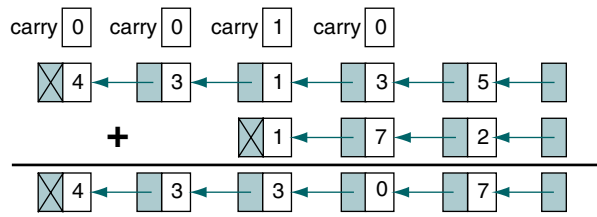


FIGURE 5-35 Integer Stored in a List for Project 30

To add two numbers, we simply add the corresponding digit in the same location in their respective lists with the carry from the previous addition. With each addition, if the sum is greater than 10, we need to subtract 10 and set the carry to 1. Otherwise, the carry is set to 0.

Write an algorithm to add two integer lists. Design your solution so that the same logic adds the first numbers (units position) as well as the rest of the number. In other words, do not have special one-time logic for adding the units position.

31. Once you have written the function for adding two numbers, multiplication is relatively easy. You can use the basic definition of multiplication, repetitive addition. In other words, if we need to multiply two numbers, such as 45×6, we simply add 45 six times. Write an algorithm that multiplies two numbers using the algorithm developed in Project 30.

32. We have shown the list as being implemented as a linked list in dynamic memory, but it is possible to implement it in an array. In this case the array structure has two basic fields: the data and the next index location. The next index field allows the array structure to take on the attributes of a list. A list array is shown in Figure 5-36.

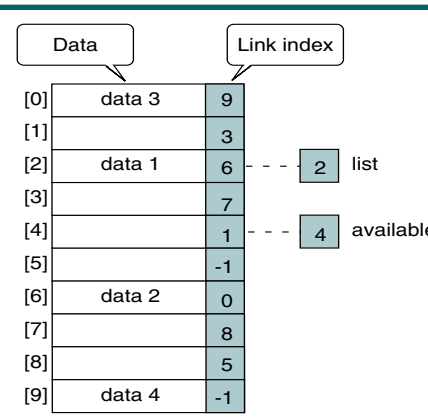


FIGURE 5-36 Linked List Array for Project 32

Note that there are actually two lists in the array. The data list starts at element 2 and progresses through 6 and 0 and to the end of the data at element 9. The second list links all of the empty or available elements together. It starts at element 4, progresses through 1, 3, 7, and 8, and ends at 5.

Write a program that implements a list using an array. Your program should be menu driven and prompt the user to insert data, delete data, print the contents of the list, or search the list for a given piece of data.

To insert new data, you must delete an element from the available list and then insert it into the data list. Conversely, when you delete a node from the data list, you must insert it into the available list.

33. Rework the list ADT to be implemented as a doubly linked list. Include a backward traversal as an additional algorithm.

34. Write a program to process stock data. The stock data should be read from a text file containing the following data: stock code, stock name, amount invested (*xxx.xx*), shares held, and current price. Use the Internet or your local paper to gather data on at least 20 stocks. (You may use mutual funds in place of stocks.)

    As each stock is read, insert it into a doubly linked multilinked list. The first logical list should be ordered on the stock code. The second logical list should be ordered on the gain or loss for the stock. Gain or loss is calculated as the current value of the stock (shares held times current price) minus the amount invested. Include at least one loss in your test data.

    After building the lists, display a menu that allows the user to display each logical list forward or backward (a total of four options). Each display should contain an appropriate heading and column captions.

    Run your program and submit a list of your input data and a printout of each display option.

35. Modify Project 34 to include the following additional processes: search for a stock using the stock code and print the data for the stock, insert data for a new stock, modify data for an existing stock, and write the data back to a file when the program terminates.

36. You have been assigned to a programming team writing a system that will maintain a doubly linked, multilinked list containing census data about cities in the United States. The first logical list maintains data in sequence by the 2000 population data. The second logical list maintains the same data using the 1990 census data. Your assignment is to write the insert routine(s). To completely test your program, you also need to write routines to print the contents of the lists.

The structure for the data is shown here:

```
Logical definition:
Metropolitan area                50 characters
Population - 2000                   integer
Population - 1900                   integer
```

The input is a file containing the census data in Table 5-5.

Metropolitan area	Census population	
	2000	1990
New York–No. NJ	21,199,865	19,549,649
Los Angeles area	16,373,645	14,531,529
Chicago area	9,157,540	8,239,820
Washington–Baltimore	7,608,070	6,727,050
San Francisco area	7,039,362	6,253,311
Philadelphia–Atlantic City area	6,188,463	5,892,937
Boston area	5,819,100	5,455,403
Detroit area	5,456,428	5,187,171
Dallas–Fort Worth	5,221,801	4,037,282
Houston–Galveston area	4,669,571	3,731,131
Atlanta area	4,112,198	2,959,950
Miami–Fort Lauderdale	3,876,380	3,192,582
Seattle area	3,554,760	2,970,328
Phoenix area	3,251,876	2,238,480
Minneapolis–St. Paul	2,968,806	2,538,834
Cleveland area	2,945,831	2,859,644
San Diego area	2,813,833	2,498,016
St. Louis area	2,603,607	2,492,525
Denver area	2,581,506	1,980,140
San Juan, PR, area	2,450,292	2,270,808

TABLE 5-5 Twenty Largest Metropolitan Areas in the United States

To verify your program, print the list in sequence, both forward and backward, for both census years. To conserve paper print the data two-up, as shown in Table 5-6.

Census Data for 1990	Population	Census Data for 2000	Population
Metropolitan area		Metropolitan area	
01 San Juan, PR, area	2450292	01 San Juan, PR, area	2270808
02 Denver area	2581506	02 Denver area	1980140
...		...	
19 Los Angeles area	16373645	19 Los Angeles	14531529
20 New York–No. NJ	21199865	20 New York–No. NJ	19549649
20 New York–No. NJ	21199865	20 New York–No. NJ	19549649
19 Los angeles Area	16373645	19 Los Angeles	14531529
...		...	
02 Denver area	2581506	02 Denver area	1980140
01 San Juan, PR, area	2450292	01 San Juan, PR, area	2270808

TABLE 5-6 Example of PrintOut

37. Modify Project 36 to include the ability to delete an element from the list.
38. Write a program that builds two linked lists (do not use the ADT) and then append the second one to the end of the first one. A pictorial representation of the data is shown in Figure 5-37.
39. Write a program that builds an array of linked lists (do not use the ADT). The data for each linked list are to be read from a set of existing files and inserted into the list. The program is to accept a variable number of files that are determined at run time by asking the user for the number of files and their names. The data structure is shown in Figure 5-38.

    After the program builds the lists, print them to verify that they are complete. Test the program with the following four files:

    a. 150 110 130 100 140
    b. 200 280 240 220 260
    c. 390 300 330 360
    d. 480 440 400

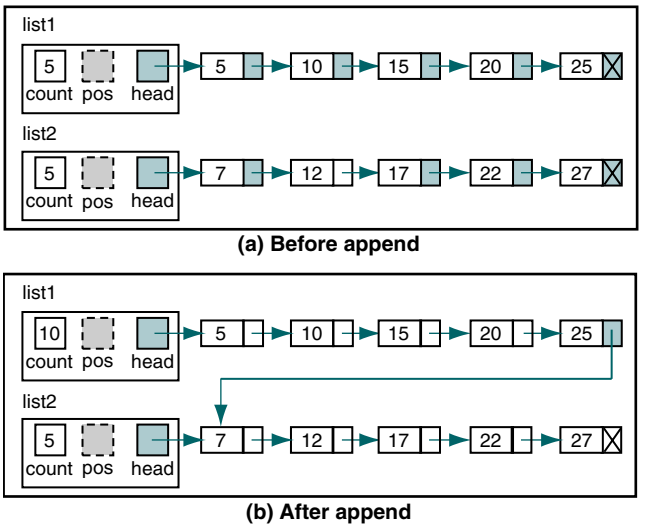


FIGURE 5-37 Append Linked Lists

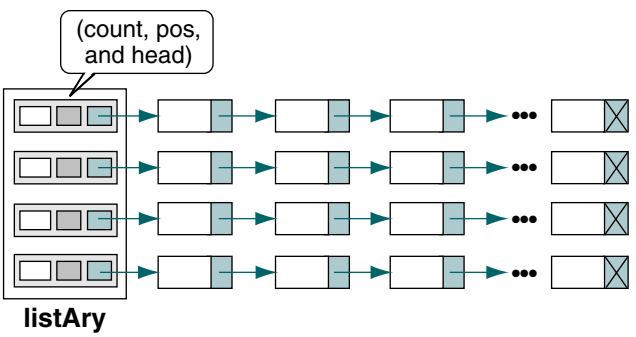


FIGURE 5-38 Structure for Array of Linked Lists

This page intentionally left blank

Part III

Non-Linear Lists

A **non-linear list** is a list in which each element can have more than one successor. Non-linear lists can be divided into two categories: trees and graphs. In a tree each element, except the root, can have only one predecessor; in a graph each element can have more than one predecessor. Figure III-1 shows how the discussion of non-linear lists is organized in this book.

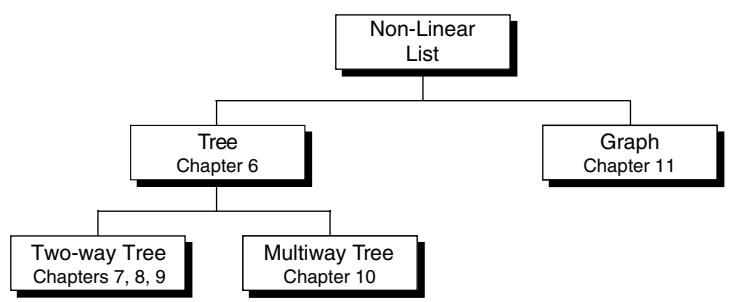


FIGURE III-1 Grouping of the Topic of Non-Linear Lists

In computer science two types of trees has been defined: two-way trees, better known as binary trees, and multiway trees. In a binary tree, each element can have up to two successors; in a multiway tree, there is no limitation on the number of successors.

In a non-linear list, each element can have more than one successor.

In a tree, an element can have only one predecessor.

In a graph, an element can have one or more predecessors.

Implementation

A non-linear list, like a linear list, can be implemented as an abstract data type (ADT). We use both arrays and linked lists to implement the different lists discussed in this part. We use an array to implement a heap because it is more convenient, as you will see. We use linked lists for the other structures because it is more efficient.

Chapters Covered

This part includes six chapters. The first five chapters discuss trees, or non-linear lists where each element has only one predecessor. The last chapter discusses graphs, or non-linear lists with one or more predecessors.

Chapter 6: Introduction to Trees

In this chapter we discuss general concepts that are related to all trees.

Chapter 7: Binary Search Trees

In this chapter we discuss the binary search tree (BST). The BSTs are simple structures that are easy to implement but are inefficient when the tree is unbalanced.

Chapter 8: AVL Trees

In this chapter we discuss AVL trees: a balanced variation on the binary tree. AVL trees are more efficient than BSTs but are also more complex.

Chapter 9: Heaps

Heaps are binary trees that may be stored in an array or in a linked-list structure. We discuss heap creation and two applications in Chapter 9: selection algorithms and priority queues. In Chapter 12 we discuss a third application, sorting.

Chapter 10: Multiway Trees

Multiway trees, or trees with an unlimited number of successors, are used for dictionary searching, spell checking, and other applications.

Chapter 11: Graphs

Graphs can be used to solve complex routing problems, such as designing and routing airlines among the airports they serve. Similarly, graphs can be used to route messages over a computer network from one node to another.

Chapter 6
Introduction to Trees

The study of trees in mathematics can be traced to Gustav Kirchhoff in the middle nineteenth century and several years later to Arthur Cayley, who used trees to study the structure of algebraic formulas. Cayley's work undoubtedly laid the framework for Grace Hopper's use of trees in 1951 to represent arithmetic expressions. Hopper's work bears a strong resemblance to today's binary tree formats.[1]

Trees are used extensively in computer science to represent algebraic formulas; as an efficient method for searching large, dynamic lists; and for such diverse applications as artificial intelligence systems and encoding algorithms. In this chapter we discuss the basic concepts behind the computer science application of trees. Then, in the following four chapters, we develop the application of trees for specific problems.

6.1 Basic Tree Concepts

A **tree** consists of a finite set of elements, called **nodes**, and a finite set of directed lines, called **branches**, that connect the nodes. The number of branches associated with a node is the **degree** of the node. When the branch is directed toward the node, it is an **indegree** branch; when the branch is directed away from the node, it is an **outdegree** branch. The sum of the indegree and outdegree branches is the degree of the node.

A tree consists of a finite set of elements, called nodes, and a finite set of directed lines, called branches, that connect the nodes.

1. Donald E. Knuth, *The Art of Computer Programming* Vol. 1, *Fundamental Algorithms,* 2nd ed. (Reading, MA: Addison-Wesley, 1972), 405, 458.

If the tree is not empty, the first node is called the **root**. The indegree of the root is, by definition, zero. With the exception of the root, all of the nodes in a tree must have an indegree of exactly one; that is, they may have only one predecessor. All nodes in the tree can have zero, one, or more branches leaving them; that is, they may have an outdegree of zero, one, or more (zero or more successors). Figure 6-1 is a representation of a tree.

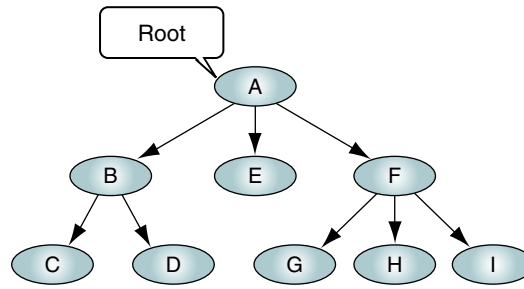


FIGURE 6-1 Tree

Terminology

In addition to *root*, many different terms are used to describe the attributes of a tree. A **leaf** is any node with an outdegree of zero, that is, a node with no successors. A node that is not a root or a leaf is known as an **internal node** because it is found in the middle portion of a tree.

A node is a **parent** if it has successor nodes—that is, if it has an outdegree greater than zero. Conversely, a node with a predecessor is a **child**. A child node has an indegree of one. Two or more nodes with the same parent are **siblings**. Fortunately, we don't have to worry about aunts, uncles, nieces, nephews, and cousins. Although some literature uses the term *grandparent*, we do not. We prefer the more general term *ancestor*. An **ancestor** is any node in the path from the root to the node. A **descendent** is any node in the path below the parent node; that is, all nodes in the paths from a given node to a leaf are descendents of that node. Figure 6-2 shows the usage of these terms.

Several terms drawn from mathematics or created by computer scientists are used to describe attributes of trees and their nodes. A **path** is a sequence of nodes in which each node is adjacent to the next one. Every node in the tree can be reached by following a unique path starting from the root. In Figure 6-2 the path from the root to the leaf `I` is designated as `AFI`. It includes two distinct branches, `AF` and `FI`.

The **level** of a node is its distance from the root. Because the root has a zero distance from itself, the root is at level 0. The children of the root are at level 1, their children are at level 2, and so forth. Note the relationship between levels and siblings in Figure 6-2. Siblings are always at the same level, but all nodes in a level are not necessarily siblings. For example, at level 2, `C` and `D` are siblings, as are `G`, `H`, and `I`. However, `D` and `G` are not siblings because they have different parents.

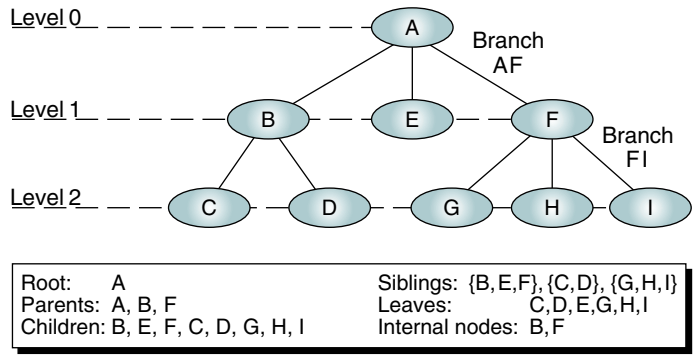


FIGURE 6-2 Tree Nomenclature

The **height** of the tree is the level of the leaf in the longest path from the root plus 1. By definition the height of an empty tree is –1. Figure 6-2 contains nodes at three levels: 0, 1, and 2. Its height is 3. Because the tree is drawn upside down, some texts refer to the **depth** of a tree rather than its height.

The level of a node is its distance from the root. The height of a tree is the level of the leaf in the longest path from the root plus 1.

A tree may be divided into subtrees. A **subtree** is any connected structure below the root. The first node in a subtree is known as the root of the subtree and is used to name the subtree. Subtrees can also be further subdivided into subtrees. In Figure 6-3, BCD is a subtree, as are E and FGHI. Note that by this definition, a single node is a subtree. Thus, the subtree B can be divided into two subtrees, C and D, and the subtree F contains the subtrees G, H, and I.

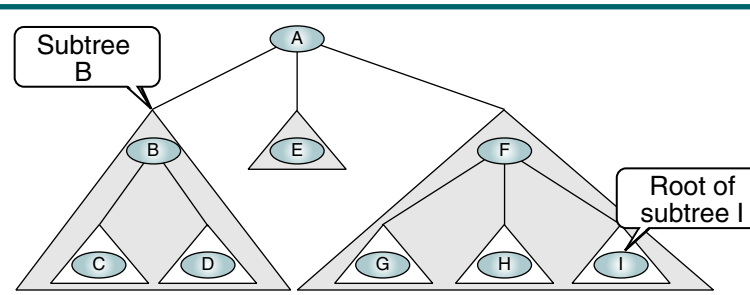


FIGURE 6-3 Subtrees

The concept of subtrees leads us to a recursive definition of a tree: A tree is a set of nodes that either: (1) is empty or (2) has a designated node, called

the root, from which hierarchically descend zero or more subtrees, which are also trees.

A tree is a set of nodes that either:
1. Is empty, or
2. Has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees

User Representation

There are three different user representations for trees. The first is the organization chart format, which is basically the notation we use to represent trees in our figures. The term we use for this notation is **general tree**. The general tree representation of a computer's components is shown in Figure 6-4; it is discussed in Section 6.3.

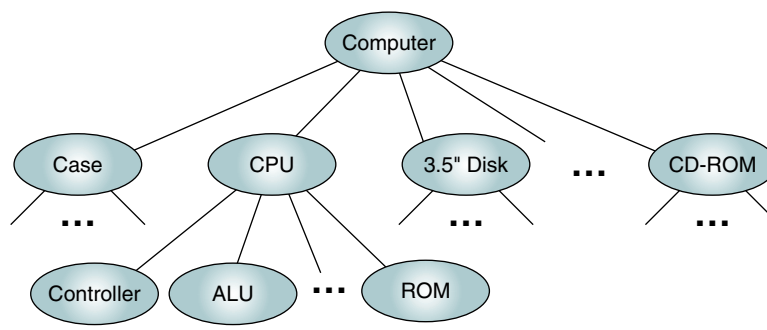


FIGURE 6-4 Computer Parts List as a General Tree

The second representational notation is the indented list. You will find it most often used in bill-of-materials systems in which a parts list represents the assembly structure of an item. The graphical representation of a computer system's components in Figure 6-4 clearly shows the relationship among the various components of a computer, but graphical representations are not easily generated from a database system. The bill-of-materials format was therefore created to show the same information using a textual parts list format. In a bill of materials, each assembly component is shown indented below its assembly. Some bills of materials even show the level number of each component. Because a bill of materials shows which components are assembled into each assembly, it is sometimes called a **goezinta** (goes into) list. Table 6-1 shows the computer bill of materials in an indented parts list format.

There is another common bill of materials with which you should be familiar. When you write a structured program, the entire program can be considered an assembly of related functions. Your structure chart is a general tree representing the relationship among the functions.

Part number	Description
301	Computer
301-1	Case
…	…
301-2	CPU
301-2-1	Controller
301-2-2	ALU
…	…
301-2-9	ROM
301-3	3.5" Disk
…	…
301-9	CD-ROM
…	…

TABLE 6-1 Computer Bill of Materials

The third user format is the parenthetical listing. This format is used with algebraic expressions. When a tree is represented in parenthetical notation, each open parenthesis indicates the start of a new level; each closing parenthesis completes the current level and moves up one level in the tree. Consider the tree shown in Figure 6-1. Its parenthetical notation is

```
A (B (C D) E F (G H I))
```

To convert a general tree to its parenthetical notation, we use the code in Algorithm 6-1.

ALGORITHM 6-1 Convert General Tree to Parenthetical Notation

```
Algorithm ConvertToParen (root, output)
Convert a general tree to parenthetical notation.
   Pre  root is a pointer to a tree node
   Post output contains parenthetical notation
 1 Place root in output
 2 if (root is a parent)
   1  Place an open parenthesis in the output
   2  ConvertToParen (root's first child)
   3  loop (more siblings)
      1  ConvertToParen (root's next child)
```

continued

ALGORITHM 6-1 Convert General Tree to Parenthetical Notation *(continued)*

```
   4   end loop
   5   Place close parenthesis in the output
 3 end if
 4 return
end ConvertToParen
```

Using this algorithm with Figure 6-1, we start by placing the root (A) in the output. Because A is a parent, we insert an opening parenthesis in the output and then process the subtree B by first placing its root (B) in the output. Again, because B is a parent, we place an open parenthesis in the output and recursively call ConvertToParen to insert C. This time, however, the root has no children. The algorithm therefore terminates and returns to statement 2.4. After placing D in the output, we return to the loop statement to discover that there are no more siblings. We now place a closing parenthesis in the output and return to statement 2.4 to continue placing B's siblings in the output.

After I's children have been completely processed, we add a closing parenthesis (2.5), which takes us up one level to complete the processing of B's siblings and the addition of a closing parenthesis. Note that when the processing is done correctly, the parentheses are balanced. If they are not balanced, you have made a mistake in the conversion.

6.2 Binary Trees

A **binary tree** is a tree in which no node can have more than two subtrees; the maximum outdegree for a node is two. In other words, a node can have zero, one, or two subtrees. These subtrees are designated as the **left subtree** and the **right subtree**. Figure 6-5 shows a binary tree with its two subtrees. Note that each subtree is itself a binary tree.

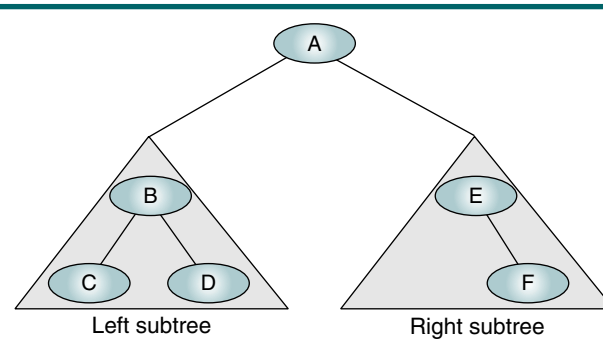


FIGURE 6-5 Binary Tree

To better understand the structure of binary trees, study Figure 6-6.

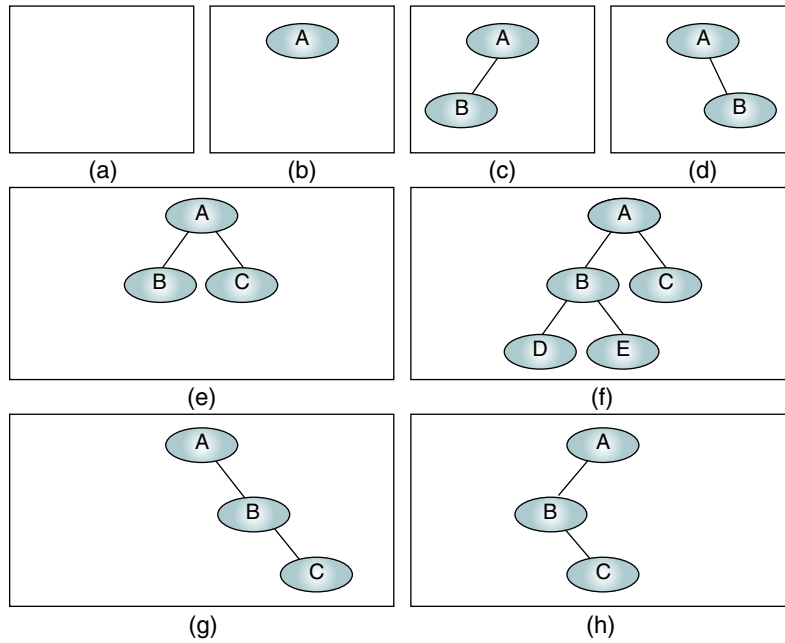


FIGURE 6-6 Collection of Binary Trees

Figure 6-6 contains eight binary trees, the first of which is a null tree. A **null tree** is a tree with no nodes, as shown in Figure 6-6(a). As you study this figure, note that symmetry is not a tree requirement.

A node in a binary tree can have no more than two subtrees.

Properties

We now define several properties for binary trees that distinguish them from general trees.

Height of Binary Trees

The height of binary trees can be mathematically predicted.

Maximum Height

Given that we need to store N nodes in a binary tree, the maximum height, H_{max}, is

$$H_{max} = N$$

Example Given three nodes to be stored in a binary tree, what is the maximum height? In this example N is 3. Therefore, the maximum height is 3. There are four different trees that can be drawn to satisfy a maximum height of 3. A tree with a maximum height is rare. It occurs when all of the nodes in the entire tree have only one successor, as shown in Figure 6-6(g) and Figure 6-6(h).

Minimum Height

The minimum height of the tree, H_{min}, is determined by the following formula:

$$H_{min} = \lfloor \log_2 N \rfloor + 1$$

Example Given three nodes to be stored in a binary tree, what is the minimum height? Again, N is 3. Therefore, the minimum height is 2.

Minimum Nodes. We can turn the calculation around and determine the minimum number of nodes in a tree of a specified height. Given a height of the binary tree, H, the minimum number of nodes in the tree are given as

$$N_{min} = H$$

Example Given a tree of height 3, what is the minimum number of nodes that can be stored? Using the formula for N_{min}, the answer is H, or 3 nodes. This case is seen in Figure 6-6(g) and 6-6(h).

Maximum Nodes

The formula for the maximum number of nodes is derived from the fact that each node can have only two descendents. Given a height of the binary tree, H, the maximum number of nodes in the tree is given as

$$N_{max} = 2^H - 1$$

Example Given a tree of height 3, what is the maximum number of nodes that can be stored? N_{max} is 7.

Balance

The distance of a node from the root determines how efficiently it can be located. The children of any node in a tree can be accessed by following only one branch path, the one that leads to the desired node. The nodes at level 1, which are children of the root, can be accessed by following only one branch. Similarly, the nodes at level 2 of a tree can all be accessed by following only two branches from the root. It stands to reason, therefore, that the shorter the tree, the easier it is to locate any desired node in the tree.

This concept leads us to a very important characteristic of a binary tree—, its **balance**. To determine whether a tree is balanced, we calculate its balance factor. The **balance factor** of a binary tree is the difference in height between its left and right subtrees. If we define the height of the left subtree as H_L and the height of the right subtree as H_R, the balance factor of the tree, B, is determined by the following formula:

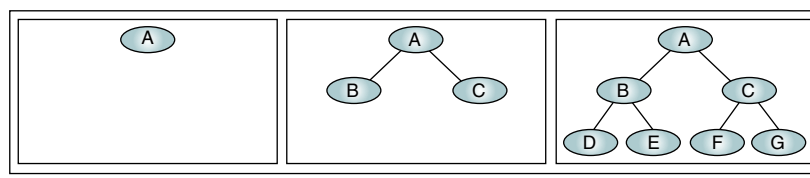
$$B = H_L - H_R$$

Using this formula, the balances of the eight trees in Figure 6-6 are (a) 0 by definition, (b) 0, (c) 1, (d) –1, (e) 0, (f) 1, (g) –2, and (h) 2.

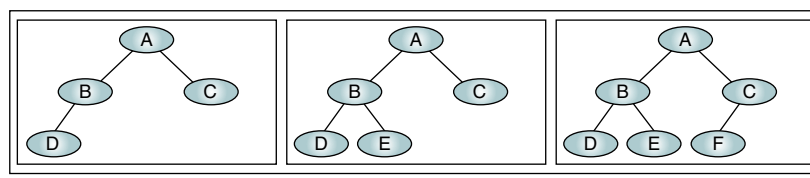
In a **balanced binary tree**, the height of its subtrees differs by no more than one (its balance factor is –1, 0, or +1), and its subtrees are also balanced. As we shall see, this definition was created by Adelson-Veskii and Landis in their definition of an AVL tree.

Complete and Nearly Complete Binary Trees

A **complete tree** has the maximum number of entries for its height (see formula N_{max} in "Height of Binary Trees"). The maximum number is reached when the last level is full (see Figure 6-7). A tree is considered **nearly complete** if it has the minimum height for its nodes (see formula H_{min}) and all nodes in the last level are found on the left. Complete and nearly complete trees are shown in Figure 6-7.



(a) Complete trees (at levels 0, 1, and 2)



(b) Nearly complete trees (at level 2)

FIGURE 6-7 Complete and Nearly Complete Trees

Binary Tree Traversals

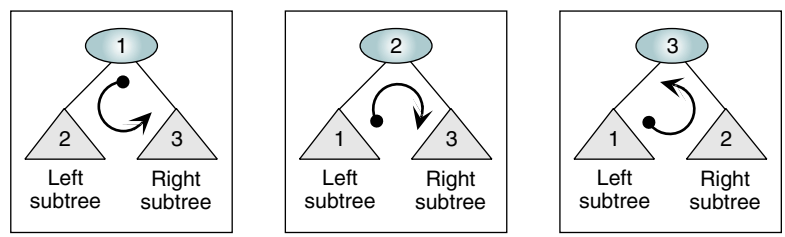
A **binary tree traversal** requires that each node of the tree be processed once and only once in a predetermined sequence. The two general approaches to the traversal sequence are depth first and breadth first. In the **depth-first traversal**, the

processing proceeds along a path from the root through one child to the most distant descendent of that first child before processing a second child. In other words, in the depth-first traversal, we process all of the descendents of a child before going on to the next child.

In a **breadth-first traversal**, the processing proceeds horizontally from the root to all of its children, then to its children's children, and so forth until all nodes have been processed. In other words, in the breadth-first traversal, each level is completely processed before the next level is started.

Depth-first Traversals

Given that a binary tree consists of a root, a left subtree, and a right subtree, we can define six different depth-first traversal sequences. Computer scientists have assigned three of these sequences standard names in the literature; the other three are unnamed but are easily derived. The standard traversals are shown in Figure 6-8.



(a) Preorder traversal **(b) Inorder traversal** **(c) Postorder traversal**

FIGURE 6-8 Binary Tree Traversals

The traditional designation of the traversals uses a designation of `node` (N) for the root, `left` (L) for the left subtree, and `right` (R) for the right subtree. To demonstrate the different traversal sequences for a binary tree, we use Figure 6-9.

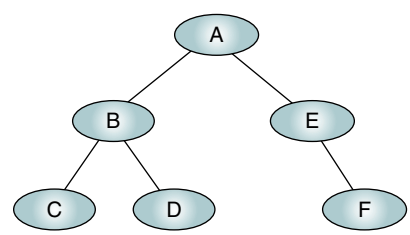


FIGURE 6-9 Binary Tree for Traversals

Preorder Traversal (NLR)

In the **preorder traversal**, the root node is processed first, followed by the left subtree and then the right subtree. It draws its name from the Latin prefix *pre*, which means to go before. Thus, the root goes before the subtrees.

> In the preorder traversal, the root is processed first, before its subtrees.

Given the recursive characteristics of trees, it is only natural to implement tree traversals recursively. First we process the root, then the left subtree, and then the right subtree. The left subtree is in turn processed recursively, as is the right subtree. The code for the preorder traversal is shown in Algorithm 6-2.

ALGORITHM 6-2 Preorder Traversal of a Binary Tree

```
Algorithm preOrder (root)
Traverse a binary tree in node-left-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
 1 if (root is not null)
   1  process (root)
   2  preOrder (leftSubtree)
   3  preOrder (rightSubtree)
 2 end if
end preOrder
```

Figure 6-9 contains a binary tree with each node named. The processing sequence for a preorder traversal processes this tree as follows: First we process the root A. After the root, we process the left subtree. To process the left subtree, we first process its root, B, then its left subtree and right subtree in order. When B's left and right subtrees have been processed in order, we are then ready to process A's right subtree, E. To process the subtree E, we first process the root and then the left subtree and the right subtree. Because there is no left subtree, we continue immediately with the right subtree, which completes the tree.

Figure 6-10 shows another way to visualize the traversal of the tree. Imagine that we are walking around the tree, starting on the left of the root and keeping as close to the nodes as possible. In the preorder traversal we process the node when we meet it for the first time (on the left of the node). This is shown as a black box on the left of the node. The path is shown as a line following a route completely around the tree and back to the root.

Figure 6-11 shows the recursive algorithmic traversal of the tree. The first call processes the root of the tree, A. It then recursively calls itself to process the root of the subtree B, as shown in Figure 6-11(b). The third call, shown in Figure 6-11(c), processes node C, which is also subtree C. At this point we call preorder with a null pointer, which results in an immediate return to subtree C

to process its right subtree. Because C's right subtree is also null, we return to node B so that we can process its right tree, D, in Figure 6-11(d). After processing node D, we make two more calls, one with D's null left pointer and one with its null right pointer. Because subtree B has now been completely processed, we return to the tree root and process its right subtree, E, in Figure 6-11(e). After a call to E's null left subtree, we call E's right subtree, F, in Figure 6-11(f). Although the tree is completely processed at this point, we still have two more calls to make: one to F's null left subtree and one to its null right subtree. We can now back out of the tree, returning first to E and then to A, which concludes the traversal of the tree.

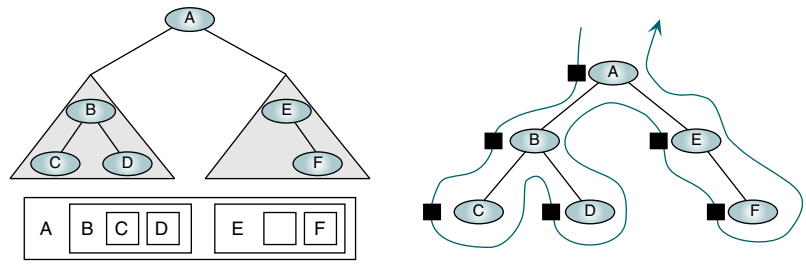


FIGURE 6-10 Preorder Traversal—A B C D E F

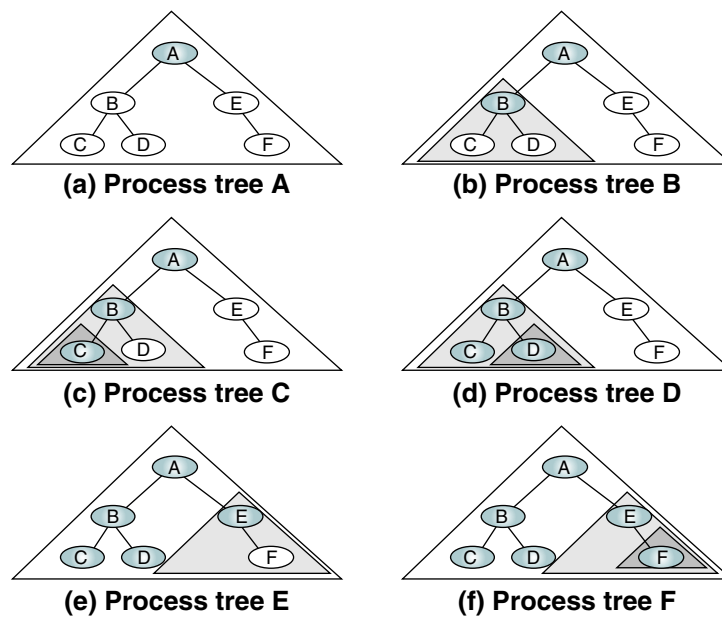


FIGURE 6-11 Algorithmic Traversal of Binary Tree

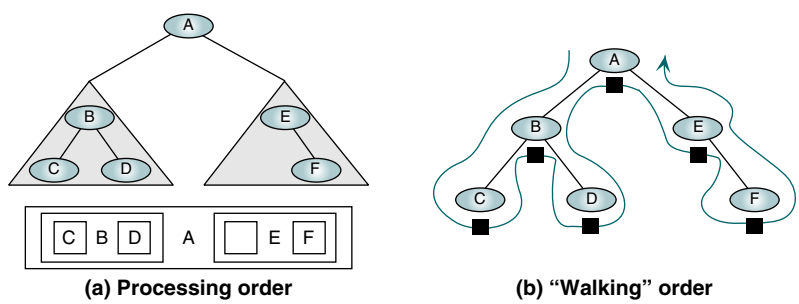
Inorder Traversal (LNR)

The **inorder traversal** processes the left subtree first, then the root, and finally the right subtree. The meaning of the prefix *in* is that the root is processed *in between* the subtrees. Once again we implement the algorithm recursively, as shown in Algorithm 6-3.

ALGORITHM 6-3 Inorder Traversal of a Binary Tree

```
Algorithm inOrder (root)
Traverse a binary tree in left-node-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
 1  if (root is not null)
    1   inOrder (leftSubTree)
    2   process (root)
    3   inOrder (rightSubTree)
 2  end if
end inOrder
```

Because the left subtree must be processed first, we trace from the root to the far-left leaf node before processing any nodes. After processing the far-left subtree, C, we process its parent node, B. We are now ready to process the right subtree, D. Processing D completes the processing of the root's left subtree, and we are now ready to process the root, A, followed by its right subtree. Because the right subtree, E, has no left child, we can process its root immediately followed by its right subtree, F. The complete sequence for inorder processing is shown in Figure 6-12.



(a) Processing order (b) "Walking" order

FIGURE 6-12 Inorder Traversal—C B D A E F

To walk around the tree in inorder sequence, we follow the same path but process each node when we meet it for the second time (the bottom of the node). This processing route is shown in Figure 6-12(b).

In the inorder traversal, the root is processed between its subtrees.

Postorder Traversal (LRN)

The last of the standard traversals is the **postorder traversal**. It processes the root node after (*post*) the left and right subtrees have been processed. It starts by locating the far-left leaf and processing it. It then processes its right sibling, including its subtrees (if any). Finally, it processes the root node.

> In the postorder traversal, the root is processed after its subtrees.

The recursive postorder traversal logic is shown in Algorithm 6-4.

ALGORITHM 6-4 Postorder Traversal of a Binary Tree

```
Algorithm postOrder (root)
Traverse a binary tree in left-right-node sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
 1 if (root is not null)
   1  postOrder (left subtree)
   2  postOrder (right subtree)
   3  process (root)
 2 end if
end postOrder
```

In the tree walk for a postorder traversal, we move the processing block to the right of the node so that we process it as we meet the node for the third time. The postorder traversal is shown in Figure 6-13. Note that we took the same path in all three walks; only the time of the processing changed.

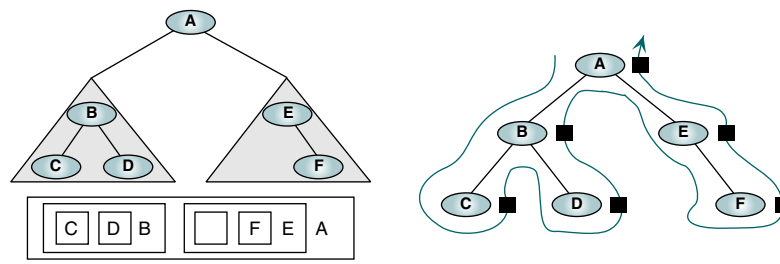


FIGURE 6-13 Postorder Traversal—C D B F E A

Breadth-first Traversals

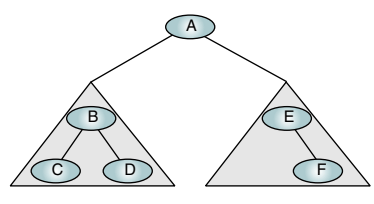
In the breadth-first traversal of a binary tree, we process all of the children of a node before proceeding with the next level. In other words, given a root at level *n*, we process all nodes at level *n* before proceeding with the nodes at

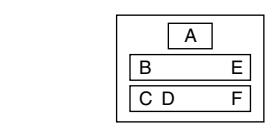
level $n + 1$. To traverse a tree in depth-first order, we used a stack. (Remember that recursion uses a stack.) To traverse a tree in breadth-first order, we use a queue. The pseudocode for a breadth-first traversal of our binary tree is shown in Algorithm 6-5.

ALGORITHM 6-5 Breadth-first Tree Traversal

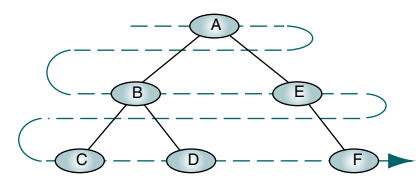
```
Algorithm breadthFirst (root)
Process tree using breadth-first traversal.
   Pre    root is node to be processed
   Post   tree has been processed
 1 set currentNode to root
 2 createQueue (bfQueue)
 3 loop (currentNode not null)
   1  process (currentNode)
   2  if (left subtree not null)
      1  enqueue (bfQueue, left subtree)
   3  end if
   4  if (right subtree not null)
      1  enqueue (bfQueue, right subtree)
   5  end if
   6  if (not emptyQueue(bfQueue))
      1  set currentNode to dequeue (bfQueue)
   7  else
      1  set currentNode to null
   8  end if
 4 end loop
 5 destroyQueue (bfQueue)
end breadthFirst
```

Like the depth-first traversals, we can trace the traversal with a walk. This time, however, the walk proceeds in a horizontal fashion, first across the root level, then across level 1, then across level 2, and so forth until the entire tree is traversed. The breadth-first traversal is shown in Figure 6-14.





(a) Processing order



(b) "Walking" order

FIGURE 6-14 Breadth-first Traversal

Expression Trees

One interesting application of binary trees is expression trees. An **expression** is a sequence of tokens that follow prescribed rules. A **token** may be either an operand or an operator. In this discussion we consider only binary arithmetic operators in the form operand–operator–operand. The standard arithmetic operators are +, -, *, and /.

An **expression tree** is a binary tree with the following properties:

- Each leaf is an operand.
- The root and internal nodes are operators.
- Subtrees are subexpressions, with the root being an operator.

Figure 6-15 contains an infix expression and its expression tree.

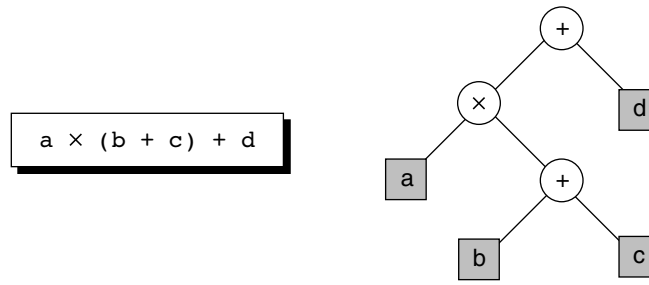


FIGURE 6-15 Infix Expression and Its Expression Tree

For an expression tree, the three standard depth-first traversals represent the three different expression formats: infix, postfix, and prefix. The inorder traversal produces the infix expression, the postorder traversal produces the postfix expression, and the preorder traversal produces the prefix expression.

Infix Traversal

To demonstrate the **infix traversal** of an expression tree, let's write an algorithm that traverses the tree and prints the expression. When we print the infix expression tree, we must add an opening parenthesis at the beginning of each expression and a closing parenthesis at the end of each expression. Because the root of the tree and each of its subtrees represents a subexpression, we print the opening parenthesis when we start a tree or a subtree and the closing parenthesis when we have processed all of its children. Figure 6-16 shows the placement of the parentheses as we walk through the tree using an inorder traversal.

The pseudocode for the infix expression tree traversal is shown in Algorithm 6-6.

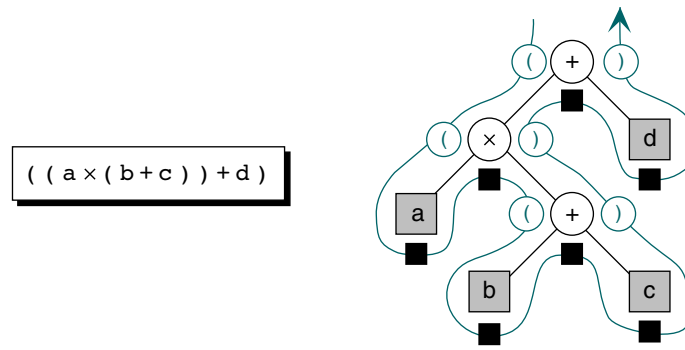


FIGURE 6-16 Infix Traversal of an Expression Tree

ALGORITHM 6-6 Infix Expression Tree Traversal

```
Algorithm infix (tree)
Print the infix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the infix expression has been printed
1  if (tree not empty)
   1   if (tree token is an operand)
       1   print (tree-token)
   2   else
       1   print (open parenthesis)
       2   infix (tree left subtree)
       3   print (tree token)
       4   infix (tree right subtree)
       5   print (close parenthesis)
   3   end if
2  end if
end infix
```

Postfix Traversal

The **postfix traversal** of an expression uses the basic postorder traversal of any binary tree. Note that it does not require parentheses. The pseudocode is shown in Algorithm 6-7.

ALGORITHM 6-7 Postfix Traversal of an Expression Tree

```
Algorithm postfix (tree)
Print the postfix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the postfix expression has been printed
1  if (tree not empty)
```

continued

ALGORITHM 6-7 Postfix Traversal of an Expression Tree *(continued)*

```
   1  postfix (tree left subtree)
   2  postfix (tree right subtree)
   3  print   (tree token)
 2 end if
end postfix
```

Prefix Traversal

The final expression tree traversal is the **prefix traversal**. It uses the standard preorder tree traversal. Again, no parentheses are necessary. The pseudocode is shown in Algorithm 6-8.

ALGORITHM 6-8 Prefix Traversal of an Expression Tree

```
Algorithm prefix (tree)
Print the prefix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the prefix expression has been printed
 1 if (tree not empty)
   1  print  (tree token)
   2  prefix (tree left subtree)
   3  prefix (tree right subtree)
 2 end if
end prefix
```

Huffman Code

The American Standard Code for Information Interchange (ASCII) is a fixed-length code; that is, the character length does not vary. Each ASCII character consists of 7 bits.[2] Although the character E occurs more frequently than the character Z, both are assigned the same number of bits. This consistency means that every character uses the maximum number of bits.

Huffman code, on the other hand, makes character storage more efficient. In Huffman code we assign shorter codes to characters that occur more frequently and longer codes to those that occur less frequently. For example, E and T, two characters that occur frequently in the English language, could be assigned one bit each. A, O, R, and N, which also occur frequently but less frequently than E and T, could be assigned two bits each. S, U, I, D, M, C, and G are the next most frequent and could be assigned three bits each, and so forth. In a given piece of text, only some of the characters require the maximum bit length. When used in a network transmission, the overall length of the transmission is shorter if Huffman-encoded characters are transmitted

2. When ASCII code is stored in an 8-bit byte, only the first 128 values are considered ASCII. The second 128 characters are used for special characters and graphics and are considered "extended ASCII."

rather than fixed-length encoding; Huffman code is therefore a popular **data compression** algorithm.

Before we can assign bit patterns to each character, we assign each character a weight based on its frequency of use. In our example, we assume that the frequency of the character E in a text is 15% and the frequency of the character T is 12%. Table 6-2 shows the weighted values of the characters.

Character	Weight	Character	Weight	Character	Weight
A	10	I	4	R	7
C	3	K	2	S	5
D	4	M	3	T	12
E	15	N	6	U	5
G	2	O	8		

TABLE 6-2 Character Weights for a Sample of Huffman Code

Once we have established the weight of each character, we build a tree based on those values. The process for building this tree is shown in Figures 6-17 through 6-20. It follows three basic steps:

1. First we organize the entire character set into a row, ordered according to frequency from highest to lowest (or vice versa). Each character is now a node at the leaf level of a tree.
2. Next we find the two nodes with the smallest combined frequency weights and join them to form a third node, resulting in a simple two-level tree. The weight of the new node is the combined weights of the original two nodes. This node, one level up from the leaves, is eligible to be combined with other nodes. Remember, the sum of the weights of the two nodes chosen must be smaller than the combination of any other possible choices.
3. We repeat step 2 until all of the nodes, on every level, are combined into a single tree.

Figure 6-17 shows the first part of this process. The first row of the figure shows step 1, with the leaf-level nodes representing the original characters arranged in descending order of value; then, as explained in step 2, we locate the two nodes with the smallest values and combine them. This step is shown in the second row. As you can see, this process results in the creation of a new node (represented by a solid circle). The frequency value (weight) of this new node is the sum of the weights of the two nodes. In the third row, we combine two more nodes, and so on.

In the sixth row, the nodes with the lowest values are found one level up from the characters rather than among the characters themselves. We combine them into a node two levels up from the leaves.

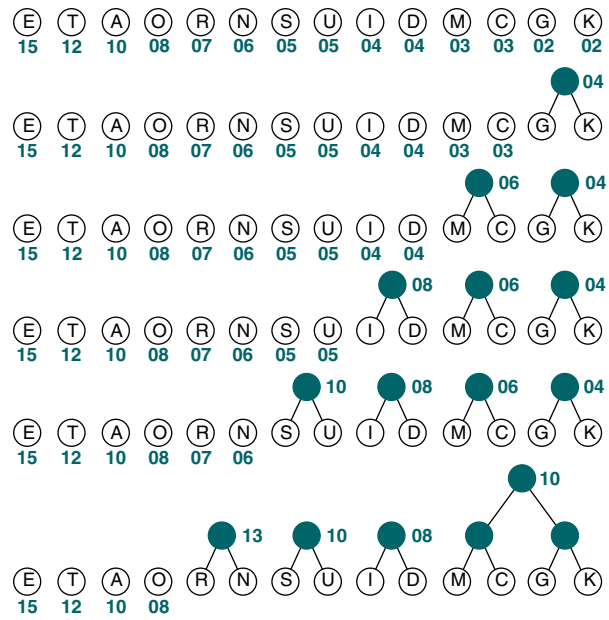


FIGURE 6-17 Huffman Tree, Part 1

Also in the sixth row, the lowest-value node is 08 (O) and the second-lowest value is 10 (A). But there are three 10s—one at the leaf level (A), one a level up from the leaves (S-U), and one two levels up from the leaves (M-C-G-K). Which should we choose? We choose whichever of the 10s is adjacent to the 8. This decision keeps the branch lines from crossing and allows us to preserve the legibility of the tree.

If none of the higher values is adjacent to the lower value, we can rearrange the nodes for clarity (see Figure 6-18). In the figure (third row), we have moved the character T from the left side of the tree to the right to combine it with a node on that side. We move the character E for the same reason.

Figure 6-19 shows the rest of the process. As you can see, the completed tree results in a single node at the root level (with a value of 86).

Once the tree is complete, we use it to assign codes to each character. First, we assign a bit value to each branch (see Figure 6-20). Starting from the root (top node), we assign 0 to the left branch and 1 to the right branch and repeat this pattern at each node. Which branch becomes 0 and which becomes 1 is left to the designer—as long as the assignments are consistent throughout the tree.

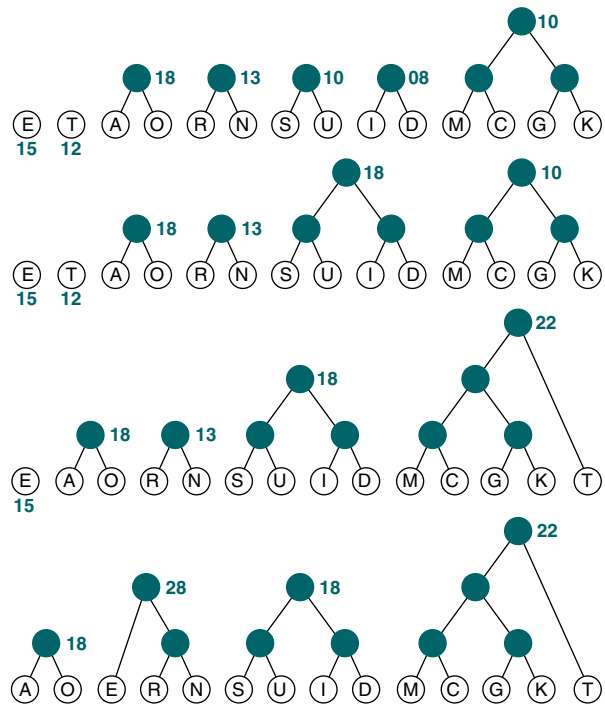


FIGURE 6-18 Huffman Tree, Part 2

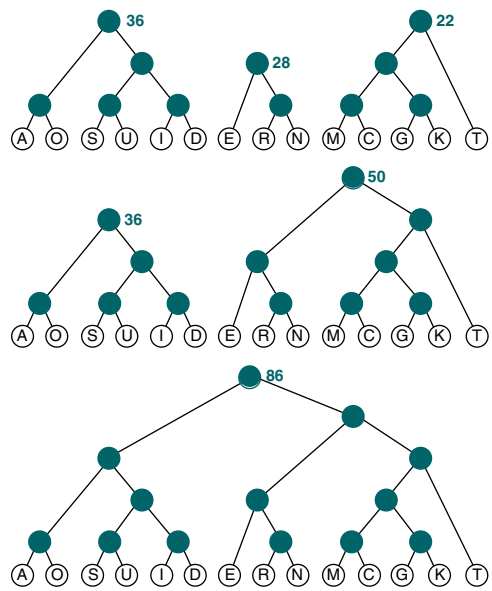


FIGURE 6-19 Huffman Tree, Part 3

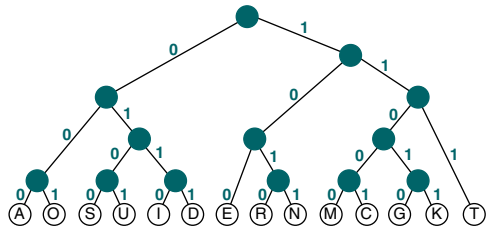


FIGURE 6-20 Huffman Code Assignment

A character's code is found by starting at the root and following the branches that lead to that character. The code itself is the bit value of each branch on the path taken in sequence. In our example, for instance, A = 000, G = 11010, and so on. The code for each character and the frequency of the character are shown in Figure 6-20. If you examine the codes carefully, you will note that the leading bits of each code are unique; that is, no code is the prefix of any other code because each has been obtained by following a different path from the root.

Huffman code is widely used for data compression; it reduces the number of bits sent or stored.

Example One common use of Huffman code is data compression for communications. Because it is a variable-length encoding system in which no character is longer than its ASCII equivalent, it saves transmission time. Assume that the following bit string uses the previously developed Huffman code.

```
000110100010010111110000001010110111001111
```

After receiving the first bit, we start from the root and follow the `000` path as we read the next two bits, arriving at the leaf `A`. Because we reached a leaf, we have decoded the first character. We then start the next character with `1` and, starting with the right branch, follow the path (`11010`) to the leaf `G`. As each character is decoded, we start at the root again until all of the message is received. The complete transmission is shown below. Because we have no spaces in our code, the words are run together.

```
AGOODMARKET
```

6.3 General Trees

A general tree is a tree in which each node can have an unlimited outdegree. Each node may have as many children as is necessary to satisfy its requirements. The bill of materials discussed earlier is an example of a general tree.

Insertions into General Trees

To insert a node into a general tree, the user must supply the parent of the node. Given the parent, three different rules may be used: (1) first in–first out (FIFO) insertion, (2) last in–first out (LIFO) insertion, and (3) key-sequenced insertion.

FIFO Insertion

When using **FIFO insertion**, we insert the nodes at the end of the sibling list, much as we insert a new node at the rear of a queue. When the list is then processed, the siblings are processed in FIFO order. FIFO order is used when the application requires that the data be processed in the order in which they were input. Figure 6-21 shows two FIFO insertions into a general tree. Given its parent as `A`, node `N` has been inserted into level 1 after node `F`; and, given its parent as `B`, node `M` has been inserted at level 2 after node `D`.

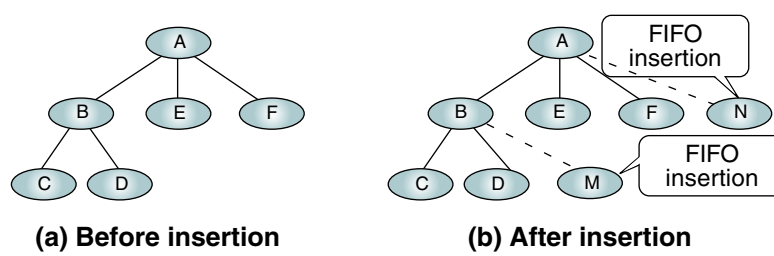


FIGURE 6-21 FIFO Insertion into General Trees

LIFO Insertion

To process sibling lists in the opposite order in which they were created, we use **LIFO insertion**. LIFO insertion places the new node at the beginning of the sibling list. It is the equivalent of a stack. Figure 6-22 shows the insertion points for a LIFO tree.

Key-sequenced Insertion

Perhaps the most common of the insertion rules, **key-sequenced insertion** places the new node in key sequence among the sibling nodes. The logic for inserting in key sequence is similar to that for insertion into a linked list. Starting at the parent's first child, we follow the sibling (right) pointers until we locate the correct insertion point and then build the links with the predecessors and successors (if any). Figure 6-23 shows the correct key-sequenced insertion locations for several different values in a general tree.

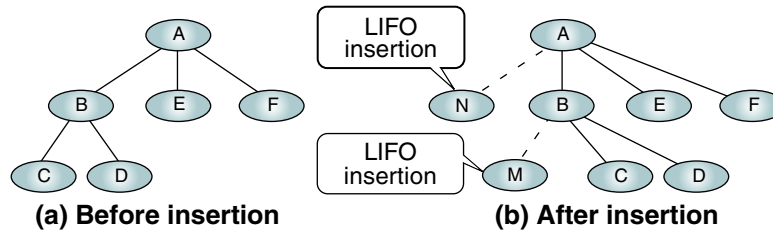


FIGURE 6-22 LIFO Insertion into General Trees

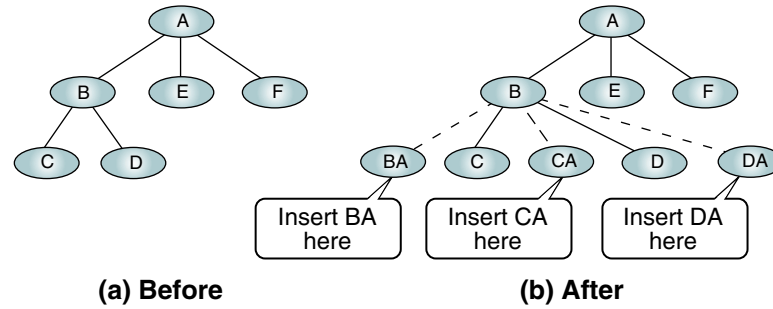


FIGURE 6-23 Key-sequenced Insertion into General Tree

General Tree Deletions

Although we cannot develop standard rules for general tree insertions, we can develop standard deletion rules. The first rule is: a node may be deleted only if it is a leaf. In the general tree, this means a node cannot be deleted if it has any children. If the user tries to delete a node that has children, the program provides an error message that the node cannot be deleted until its children are deleted. It is then the user's responsibility to first delete any children. As an alternative, the application could be programmed to delete the children first and then delete the requested node. If this alternative is used, it should be with a different user option, such as purge node and children, and not the simple delete node option.

Changing a General Tree to a Binary Tree

It is considerably easier to represent binary trees in programs than it is to represent general trees. We would therefore like to be able to represent general trees using a binary tree format. The binary tree format can be adopted by changing the meaning of the left and right pointers. In a general tree, we use two relationships: parent to child and sibling to sibling. Using these two relationships, we can represent any general tree as a binary tree.

Consider the tree shown in Figure 6-24. To change it to a binary tree, we first identify the branch from the parent to its first child. These branches from each parent become left pointers in the binary tree. They are shown in Figure 6-24(b). Then we connect siblings, starting with the far-left child, using a branch for each sibling to its right sibling. These branches, shown in Figure 6-24(c), are the right pointers in the binary tree. The third and last step in the conversion process is to remove all unneeded branches from the parent to its children. The resulting binary tree is shown in Figure 6-24(d). Although this is a valid tree structure, it does not have a traditional binary tree format. We therefore redraw it as shown in Figure 6-24(e).

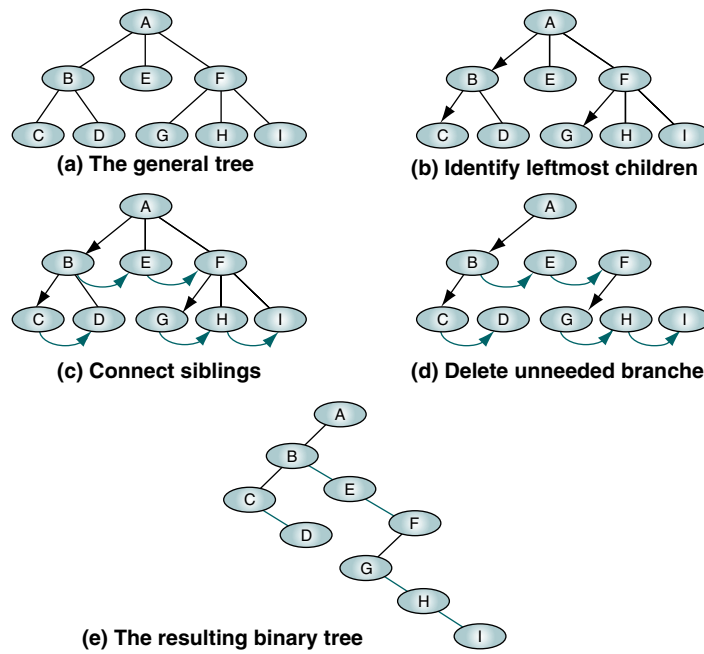


FIGURE 6-24 Converting General Trees to Binary Trees

6.4 Key Terms

ancestor
balance
balance factor
balanced binary tree
binary tree
binary tree traversal
branch
breadth-first traversal
child
complete tree
data compression
degree
depth
depth-first traversal
descendent
expression
expression tree
FIFO insertion
general tree
goezinta
height
Huffman code
indegree
infix traversal
inorder traversal
internal node
key-sequenced insertion
leaf
left subtree
level
LIFO insertion
nearly complete tree
node
null tree
outdegree
parent
path
postfix traversal
postorder traversal
prefix traversal
preorder traversal
right subtree
root
sibling
subtree
token
tree

6.5 Summary

- A tree consists of a finite set of elements called nodes and a finite set of directed lines called branches that connect the nodes.
- The number of branches associated with a node is the degree of the node.
- When the branch is directed toward the node, it is an indegree branch; when the branch is directed away from the node, it is an outdegree branch. The sum of indegree and outdegree branches is the degree of the node.
- If the tree is not empty, the first node is called the root, which has the indegree of zero.
- All nodes in the tree, except the root, must have an indegree of one.
- A leaf is a node with an outdegree of zero.
- An internal node is a node that is neither the root nor a leaf.
- A node can be a parent, a child, or both.
- Two or more nodes with the same parent are called siblings.
- A path is a sequence of nodes in which each node is adjacent to the next one.

- An ancestor is any node in the path from the root of a given node. A descendent is any node in all of the paths from a given node to a leaf.
- The level of a node is its distance from the root.
- The height of a tree is the level of the leaf in the longest path from the root plus 1; the height of an empty tree is –1.
- A subtree is any connected structure below the root.
- A tree can be defined recursively as a set of nodes that either: (1) is empty or (2) has a designated node called the root from which hierarchically descend zero or more subtrees, which are also trees.
- A binary tree is a tree in which no node can have more than two children.
- The minimum and maximum height of a binary tree can be related to the number of nodes:

$$H_{min} = \lfloor \log_2 N \rfloor + 1 \qquad H_{max} = N$$

- Given the height of a binary tree, the minimum and maximum number of nodes in the tree can be calculated as

$$N_{min} = H \qquad N_{max} = 2^H - 1$$

- The balance factor of a binary tree is the difference in height between its left and right subtrees.

$$B = H_L - H_R$$

- A binary tree is balanced if the heights of its subtrees differ by no more than 1 and its subtrees are also balanced.
- A complete binary tree has the maximum number of entries for its height; a tree is complete when the last level is full.
- A nearly complete binary tree is a tree that has the minimum height for its nodes and all nodes in the last level are found on the left.
- A binary tree traversal visits each node of the tree once and only once in a predetermined sequence.
- The two approaches to binary tree traversal are depth first and breadth first.
- Using the depth-first approach, we may traverse a binary tree in six different sequences; however, only three of these sequences are given standard names: preorder, inorder, and postorder.
- In the preorder traversal, we process the root first, followed by the left subtree and then the right subtree.
- In the inorder traversal, we process the left subtree first, followed by the root and then the right subtree.

- In the postorder traversal, we process the left subtree first, followed by the right subtree and then the root.
- In the breadth-first approach, we process all nodes in a level before proceeding to the next level.
- A general tree is a tree in which each node can have an unlimited outdegree.
- To change a general tree to a binary tree, we identify the first child of each node, connect the siblings from left to right, and delete the connection between each parent and all children except the first.
- The three approaches for inserting data into general trees are FIFO, LIFO, and key sequenced.
- To delete a node in a general tree, we must ensure that it does not have a child.
- Huffman code is an encoding method that uses a variable-length code to represent characters.
- In Huffman code we assign shorter codes to characters that occur more frequently and longer codes to those that occur less frequently.

6.6 Practice Sets

Exercises

1. Show the tree representation of the following parenthetical notation:

```
a (b (c d) e f (g h))
```

2. In Figure 6-25 find the:
   a. root
   b. leaves
   c. internal nodes
   d. ancestors of H
   e. descendents of F
3. In Figure 6-25 find the:
   a. indegree of node F
   b. outdegree of node B
   c. siblings of H
   d. parent of K
   e. children of C
4. In Figure 6-25 find the:
   a. height of the tree
   b. height of subtree G
   c. level of node G
   d. level of node A
   e. height of subtree E

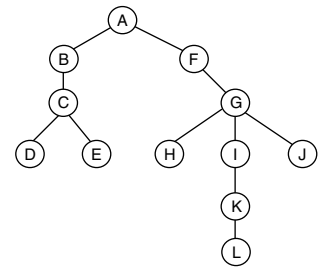


FIGURE 6-25 Tree for Exercises 2, 3, 4, 5, 6, and 7

5. In Figure 6-25 show the subtrees of node F.
6. In Figure 6-25 show the indented list representation of the tree.
7. In Figure 6-25 show the parenthetical representation of the tree.
8. Find a binary tree whose preorder and inorder traversals create the same result.
9. What are the maximum and minimum heights of a tree with 28 nodes?
10. In a binary tree, what is the maximum number of nodes that can be found in level 3? In level 4? In level 12?
11. What is the balance factor of the tree in Figure 6-26?

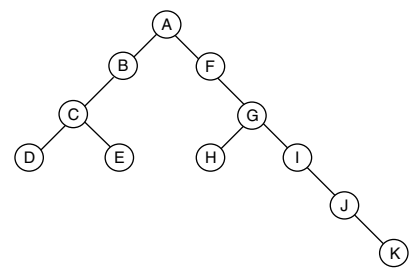


FIGURE 6-26 Binary Tree for Exercises 11, 14, 15, and 30

12. Draw a complete tree to level 4.
13. How many different nearly complete trees can exist in a tree of height 4?
14. Show the depth-first traversals (preorder, inorder, and postorder) of the binary tree in Figure 6-26.
15. Show the breadth-first traversal of the tree in Figure 6-26.
16. Find the root of each of the following binary trees:
    a. tree with postorder traversal: FCBDG
    b. tree with preorder traversal: IBCDFEN
    c. tree with inorder traversal: CBIDFGE

17. A binary tree has 10 nodes. The preorder and inorder traversals of the tree are shown below. Draw the tree.

    Preorder: JCBADEFIGH

    Inorder: ABCEDFJGIH

18. A binary tree has eight nodes. The postorder and inorder traversals of the tree are given below. Draw the tree.

    Postorder: FECHGDBA

    Inorder: FCEABHDG

19. A binary tree has seven nodes. The preorder and postorder traversals of the tree are given below. Can you draw the tree? If not, explain.

    Preorder: GFDABEC

    Postorder: ABDCEFG

20. A nearly complete binary tree has nine nodes. The breadth traversal of the tree is given below. Draw the tree.

    Breadth: JCBADEFIG

21. Draw all possible nonsimilar binary trees with three nodes (A, B, C).

22. Draw the corresponding binary tree of Figure 6-21(b).

23. What is the minimum number of levels a binary tree with 42 nodes can have?

24. What is the minimum number of levels a ternary tree, that is, a tree with an outdegree of 3, with 42 nodes can have?

25. What is the maximum number of nodes at level five of a binary tree?

26. Find the infix, prefix, and postfix expressions in the expression tree of Figure 6-27.

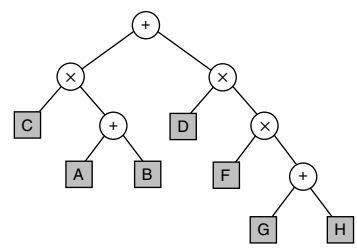


FIGURE 6-27 Expression Tree for Exercise 26

27. Draw the expression tree and find the prefix and postfix expressions for the following infix expression:

```
(C + D + A × B) × (E + F)
```

28. Draw the expression tree and find the infix and postfix expressions for the following prefix expression:

```
× - A B + × C D / E F
```

29. Draw the expression tree and find the infix and prefix expressions for the following postfix expression:

```
A B × C D / + E F - ×
```

30. Show the result of the recursive function in Algorithm 6-9 using the tree in Figure 6-26.

ALGORITHM 6-9 Tree Traversal for Exercise 30

```
Algorithm treeTraversal (tree)
1  if tree is null
   1   print "Null"
2  else
   1   treeTraversal (right subtree)
   2   print "right is done"
   3   treeTraversal (left subtree)
   4   print (tree data)
3  end if
end treeTraversal
```

31. Supply the missing factor (the question mark) in the following recursive definition (Figure 6-28) of the maximum number of nodes based on the height of a binary tree.

$$N(H) = \begin{cases} 1 & \text{if } H = 1 \\ N(H - 1) + ? & \text{if } H \geq 2 \end{cases}$$

FIGURE 6-28 Recursive Definition for Exercise 31

Problems

32. Write an algorithm that counts the number of nodes in a binary tree.
33. Write an algorithm that counts the number of leaves in a binary tree.
34. Write an algorithm to delete all the leaves from a binary tree, leaving the root and intermediate nodes in place. (*Hint*: Use a preorder traversal.)
35. Write an algorithm that, given the number of nodes in a complete or nearly complete binary tree, finds the height of the tree.

36. Write an algorithm that determines whether a binary tree is complete.
37. Write an algorithm that determines whether a binary tree is nearly complete.
38. Rewrite the binary tree preorder traversal algorithm using a stack instead of recursion.
39. Rewrite the binary tree inorder traversal algorithm using a stack instead of recursion.
40. Rewrite the binary tree postorder traversal algorithm using a stack instead of recursion.
41. Write the FIFO insertion algorithm for general trees.
42. Write the LIFO insertion algorithm for general trees.
43. Write the key-sequenced insertion algorithm for general trees.
44. Write the deletion algorithm for a general tree.
45. Write an algorithm that creates a mirror image of a binary tree. All left children become right children and vice versa.

Projects

46. Write a C function to compute the balance factor of a binary tree. If it is called initially with the root pointer, it should determine the balance factor of the entire tree. If it is called with a pointer to a subtree, it should determine the balance factor for the subtree.
47. Write a pseudocode algorithm to build a Huffman tree. Use the alphabet as shown in Table 6-3.

Character	Weight	Character	Weight	Character	Weight
A	7	J	1	S	6
B	2	K	1	T	8
C	2	L	4	U	4
D	3	M	3	V	1
E	11	N	7	W	2
F	2	O	9	X	1
G	2	P	2	Y	2
H	6	Q	1	Z	1
I	6	R	6		

TABLE 6-3 Huffman Character Weights for Project 47

48. Write the C implementation for the Huffman algorithm developed in Project 47. After it has been built, print the code. Then write a C program to read characters from the keyboard and convert them to your Huffman code. Include a function in your program that converts Huffman code back to text. Use it to verify that the code entered from the keyboard was converted correctly.

This page intentionally left blank

Chapter 7
Binary Search Trees

We now turn our attention to search trees, with an in-depth discussion of two standard tree structures: binary search trees in this chapter and AVL trees in Chapter 8. Both are used when data need to be ordered. They differ primarily in that AVL trees are balanced, whereas binary search trees are not.

In the design of the linear list structure, we had two choices: an array or a linked list. The array structure provides a very efficient search algorithm, the binary search, but its insertion and deletion algorithms are very inefficient. On the other hand, the linked list structure provides efficient insertion and deletion, but its search algorithm is very inefficient. What we need is a structure that provides an efficient search algorithm and at the same time efficient insert and delete algorithms. The binary search tree and the AVL tree provide that structure.

7.1 Basic Concepts

A **binary search tree (BST)** is a binary tree with the following properties:

- All items in the left subtree are less than the root.
- All items in the right subtree are greater than or equal to the root.
- Each subtree is itself a binary search tree.

> In a binary search tree, the left subtree contains key values less than the root and the right subtree contains key values greater than or equal to the root.

Generally, the information represented by each node is a record rather than a single data element. When the binary search tree definition is applied to a record, the sequencing properties refer to the key of the record. Figure 7-1 reflects the properties of a binary tree in which K is the key.

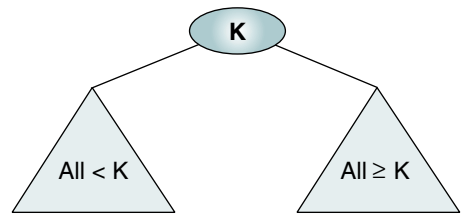


FIGURE 7-1 Binary Search Tree

Figure 7-2 contains five binary search trees. As you study them, note that the trees in Figures 7-2(a) and (b) are complete and balanced, the tree in Figure 7-2(d) is nearly complete and balanced, and the trees in Figures 7-2(c) and (e) are neither complete nor balanced.

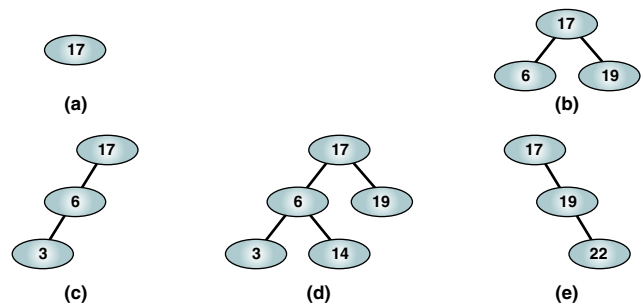


FIGURE 7-2 Valid Binary Search Trees

Now let's look at some binary trees that do not have the properties of a binary search tree. Examine the binary trees in Figure 7-3. The first tree, Figure 7-3(a), breaks the first rule: all items in the left subtree must be less than the root. The key in the left subtree (22) is greater than the key in the root (17). The second tree, Figure 7-3(b), breaks the second rule: all items in the right subtree must be greater than or equal to the root. The key in the right subtree (11) is less than the key in the root (17). Figure 7-3(c) breaks the third rule: each subtree must be a binary search tree. In this tree the left subtree key (6) is less than the root (17), and the right subtree key (19) is greater than the root. However, the left subtree is not a valid binary search tree because it breaks the first rule: its left subtree (11) is greater than the root (6). Figure 7-3(d) also breaks one of the three rules. Do you see which one? (*Hint:* What is the largest key in the left subtree?)

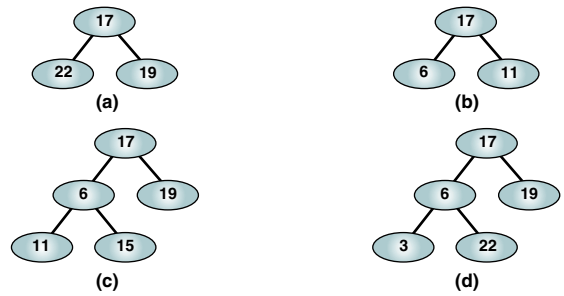


FIGURE 7-3 Invalid Binary Search Trees

7.2 BST Operations

We now examine several operations used on binary search trees. We start with the traversal algorithms we studied in Chapter 6. We then look at some simple search algorithms and conclude with the algorithms that build a binary search tree.

Traversals

The **binary tree traversal** operations are identical to the ones in Chapter 6. Our interest here is not in the operation itself but rather in the results it produces. Let's begin by traversing the tree in Figure 7-4.

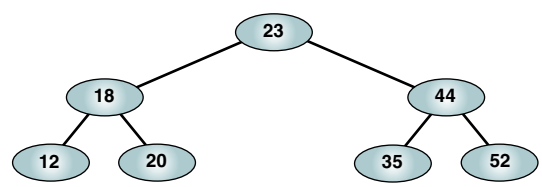


FIGURE 7-4 Example of a Binary Search Tree

If we traverse the tree using a preorder traversal, we get the results shown below.

```
23 18 12 20 44 35 52
```

Although this traversal is valid, it is not very useful. Let's try a postorder traversal and see if it is more useful.

```
12 20 18 35 52 44 23
```

Again, this sequence holds little promise of being useful. Let's try an inorder traversal.

```
12 18 20 23 35 44 52
```

This traversal has some very practical use: the inorder traversal of a binary search tree produces a sequenced list. What happens if you traverse the tree using a right-node-left sequence? Try it and see.[1]

> The inorder traversal of a binary search tree produces a sequenced list.

Searches

In this section we study three search algorithms: find the smallest node, find the largest node, and find a requested node (BST search).

Find the Smallest Node

As we examine the binary search tree in Figure 7-4, we note that the node with the smallest value (12) is the far-left leaf node in the tree. The **find smallest node** operation, therefore, simply follows the left branches until we get to a leaf. Algorithm 7-1 contains the pseudocode to find the smallest node in a binary search tree.

ALGORITHM 7-1 Find Smallest Node in a BST

```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
1  if (left subtree empty)
   1   return (root)
2  end if
3  return findSmallestBST (left subtree)
end findSmallestBST
```

Algorithm 7-1 Analysis As is typical with trees, this algorithm is recursive. The first call starts with the root of the tree, as shown in Figure 7-5. We then follow a path down the left subtrees. If the left

1. As you can see, the right-node-left traversal traverses the tree in a descending sequence. This is not a standard traversal, but it can be very useful.

subtree is not null, we must keep looking farther to the left. We do so with a recursive call to `findSmallestBST` with the left subtree. The base case in this algorithm occurs when we find an empty left subtree. At this point we return the address of the current node, which is the node containing 12 as the base case. Because the recursive call is part of a return statement, as we move back up the tree we continue to return the node address to the smallest node until we finally return it to the initiating module.

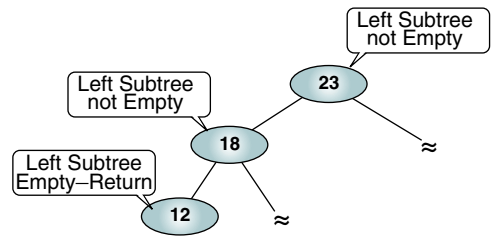


FIGURE 7-5 Find Smallest Node in a BST

Find the Largest Node

The logic to the **find largest node** operation in a binary search tree is the reverse of finding the smallest node. This time we start at the tree root and follow the right branches to the far-right node in the tree, which by definition must be the largest. The pseudocode is shown in Algorithm 7-2.

ALGORITHM 7-2 Find Largest Node in a BST

```
Algorithm findLargestBST (root)
This algorithm finds the largest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of largest node returned
1  if (right subtree empty)
   1   return (root)
2  end if
3  return findLargestBST (right subtree)
end findLargestBST
```

BST Search

We now examine the most important feature of binary search trees, the **binary tree search**, which locates a specific node in the tree. To help us understand how the BST search works, let's revisit the binary search algorithm, as shown in Figure 7-6. This figure traces each of the possible search paths from the middle element in the array. Starting with 23, the binary search examines either 18 or 44, depending on the search key. From 18 it examines either 12 or 20; from 44 it examines either 35 or 52. As is clear in the figure, tracing all possible search paths follows the same paths we see in a binary search tree.

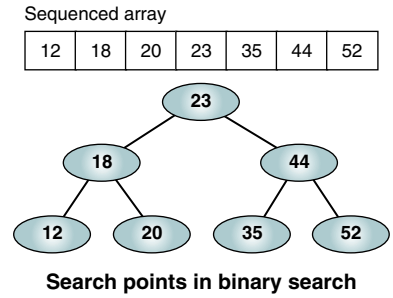


FIGURE 7-6 BST and the Binary Search

Now let's reverse the process. Find a given node in a binary search tree. Assume we are looking for node 20. We begin by comparing the search argument, 20, with the value in the tree root. Because 20 is less than the root value, 23, and because we know that all values less than the root lie in its left subtree, we go left. We now compare the search argument with the value in the subtree, 18. This time the search argument is greater than the root value, 18. Because we know that values greater than the tree root must lie in its right subtree, we go right and find our desired value. This logic is shown in Algorithm 7-3.

ALGORITHM 7-3 Search BST

```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
   Pre    root is the root to a binary tree or subtree
          targetKey is the key value requested
   Return the node address if the value is found
          null if the node is not in the tree
 1 if (empty tree)
      Not found
   1  return null
 2 end if
 3 if (targetKey < root)
   1  return searchBST (left subtree, targetKey)
 4 else if (targetKey > root)
   1  return searchBST (right subtree, targetKey)
 5 else
      Found target key
   1  return root
 6 end if
end searchBST
```

Algorithm 7-3 Analysis We implement the BST search using recursion. In this algorithm there are two base cases: either we find the search argument in the tree, in which case we return the

address of its node (statement 5.1), or the search argument doesn't exist, in which case we return null (statement 1.1).

Study the returns at statements 3.1 and 4.1 carefully. Note that they are returning the value given by the recursive call, which as we saw earlier is either null or the address of the node we are trying to locate. These statements are necessary to pass the located address back through the recursion to the original requester. Figure 7-7 traces the path through the binary search tree from Figure 7-4, as we search for node 20 using Algorithm 7-3.

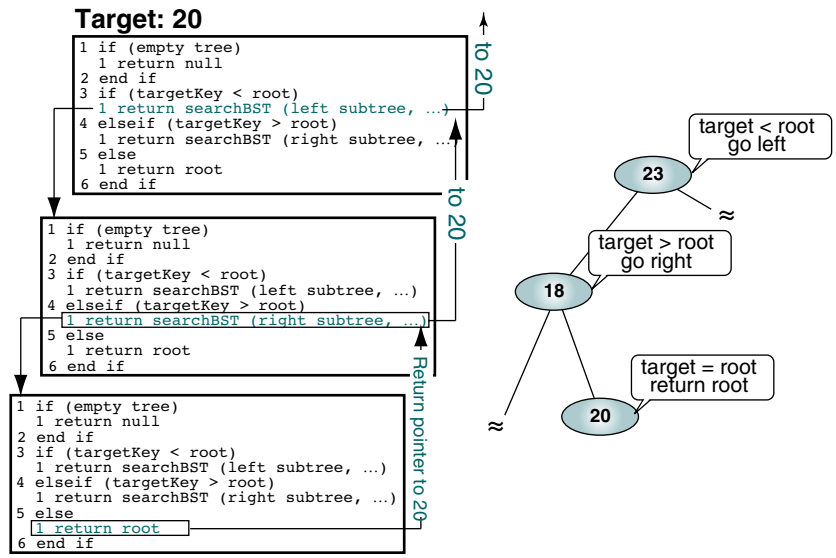


FIGURE 7-7 Searching a BST

Insertion

The **insert node** function adds data to a BST. To insert data all we need to do is follow the branches to an empty subtree and then insert the new node. In other words, all inserts take place at a leaf or at a leaflike node—a node that has only one null subtree.

All BST insertions take place at a leaf or a leaflike node.

Figure 7-8 shows our binary search tree after we have inserted two nodes. We first added node 19. To locate its insertion point, we searched the tree through the path 23, 18, and 20 to a null left branch. After locating the insertion point, we inserted the new node as the left subtree of 20. We then added 38. This time we searched the tree through 23, 44, and 35 to a null right subtree and inserted the new node.

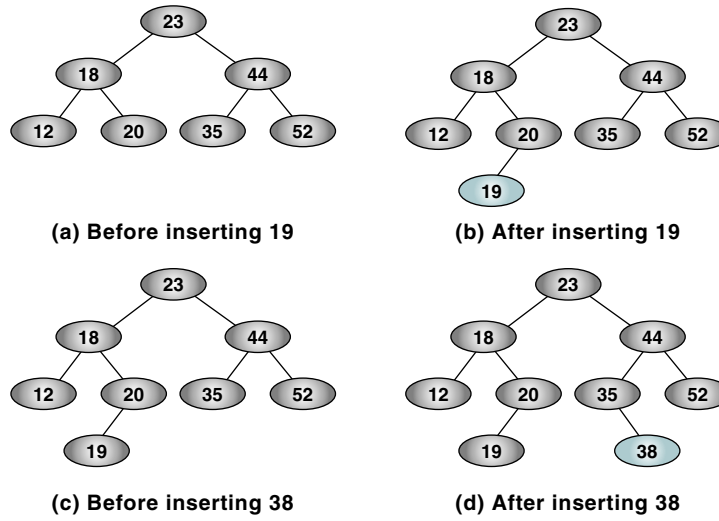


FIGURE 7-8 BST Insertion

Insertions of both 19 and 38 were made at a leaf node. If we inserted a duplicate of the root, 23, it would become the left subtree of 35. Remember that in a binary search tree, nodes with equal values are found in the right subtree. The path for its insertion would therefore be 23, 44, and 35. In this case the insertion takes place at a leaflike node. Although 35 has a right subtree, its left subtree is null. We would therefore place the new node, 23, as the left subtree of 35.

We are now ready to develop the insert algorithm. We can write an elegant algorithm that inserts the data into the tree using recursion. If the tree or subtree is empty, we simply insert the data at the root. If we are not at an empty tree, we determine which branch we need to follow and call recursively to determine whether we are at a leaf yet. The pseudocode is shown in Algorithm 7-4.

ALGORITHM 7-4 Add Node to BST

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
   Pre    root is address of current node in a BST
          newNode is address of node containing data
   Post   newNode inserted into the tree
   Return address of potential new tree root
1  if (empty tree)
   1   set root to newNode
   2   return newNode
2  end if
```

continued

ALGORITHM 7-4 Add Node to BST *(continued)*

```
   Locate null subtree for insertion
3  if (newNode < root)
   1   return addBST (left subtree, newNode)
4  else
   1   return addBST (right subtree, newNode)
5  end if
end addBST
```

Algorithm 7-4 Analysis The algorithm is quite elegant, but it is not easy to see how the new node is inserted at the correct location. To help, let's insert a node into a tree as shown in Figure 7-9.

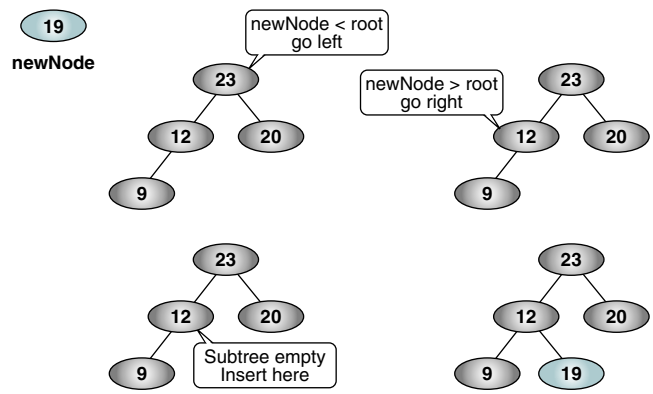


FIGURE 7-9 Trace of Recursive BST Insert

To insert 19 into the tree, we start with `root`. Because the new node's key is less than the root key, we recursively call `addBST` using the left subtree (12). The new node's key is now greater than the root key, so we again call recursively with the right subtree, which is null. At this point we discover that the subtree is null, so we insert the new node, replacing the null subtree as 12's right subtree.

Deletion

To **delete a node** from a binary search tree, we must first locate it. There are four possible cases when we delete a node:

1. The node to be deleted has no children. In this case, all we need to do is delete the node.
2. The node to be deleted has only a right subtree. We delete the node and attach the right subtree to the deleted node's parent.
3. The node to be deleted has only a left subtree. We delete the node and attach the left subtree to the deleted node's parent.
4. The node to be deleted has two subtrees. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced

trees. Rather than simply delete the node, therefore, we try to maintain the existing structure as much as possible by finding data to take the place of the deleted data. This can be done in one of two ways: (1) we can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data or (2) we can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data. Regardless of which logic we use, we will be moving data from a leaf or a leaflike node that can then be deleted. Prove to yourself that either of these moves preserves the integrity of the binary search tree.

The pseudocode for the binary search tree delete is shown in Algorithm 7-5.

ALGORITHM 7-5 Delete Node from BST

```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
   Pre    root is reference to node to be deleted
          dltKey is key of node to be deleted
   Post   node deleted
          if dltKey not found, root unchanged
   Return true if node deleted, false if not found
 1 if (empty tree)
   1  return false
 2 end if
 3 if (dltKey < root)
   1  return deleteBST (left subtree, dltKey)
 4 else if (dltKey > root)
   1  return deleteBST (right subtree, dltKey)
 5 else
      Delete node found--test for leaf node
   1  If (no left subtree)
      1  make right subtree the root
      2  return true
   2  else if (no right subtree)
      1  make left subtree the root
      2  return true
   3  else
         Node to be deleted not a leaf. Find largest node on
         left subtree.
      1  save root in deleteNode
      2  set largest to largestBST (left subtree)
      3  move data in largest to deleteNode
      4  return deleteBST (left subtree of deleteNode,
                           key of largest
   4  end if
 6 end if
end deleteBST
```

Algorithm 7-5 Analysis

You need to study this algorithm carefully to fully understand it. First, note that it is a recursive algorithm. There are two base cases: First, we do not find the node. In that

case the root pointer is null. This case is handled in statement 1.1. The second case occurs after we have deleted the node, at either statement 5.1.2 or statement 5.2.2.

The first two cases on page 313 have been combined in one case in the actual implementation of the algorithm. If the left subtree is null, we can simply connect the right subtree to the parent (`root`). If the right subtree is null, we are connecting a null subtree, which is correct. If the right subtree is not null, its data are connected to the deleted node's parent, which is Case 2. On the other hand, if the left subtree is not null, we test to see whether the right subtree is null. If the right subtree is null, we can move the left subtree pointer to its parent.

The most difficult logic in this algorithm occurs when the node is not a leaf. You need to study this situation carefully to fully understand it. We begin by searching for the largest node on the left subtree and move its data to replace the data to be deleted. We then call the algorithm recursively, giving it a new delete target, the key of the leaf node that contains the data we moved to the internal or root node. This guarantees that when we find the target key this time, at least one of its subtrees is null. We trace this logic in Figure 7-10.

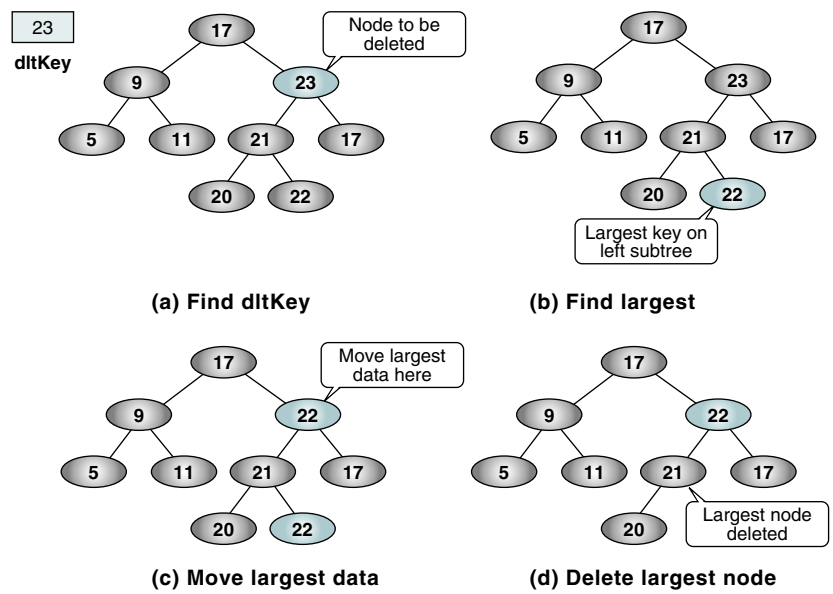


FIGURE 7-10 Delete BST Test Cases

7.3 Binary Search Tree ADT

No programming language has intrinsic operations for a binary search tree. Its operations must be simulated using C functions. The model for the BST abstract data type is the ADT we developed for lists in Chapter 5. Both structures use dynamic memory to store nodes containing a pointer to the application data. Both structures require pointers to identify each node's successor. Whereas the list contains only one pointer to each successor, the BST tree uses two—one for the left subtree and one for the right subtree. Because there is

order in the BST tree, just as there was in the list, we need an application-dependent search function that must be written by the application programmer.

A major difference between the two structures lies in the traversal of the structure. In the list, we provided a function to retrieve the next node in the list. Because we use recursion in the BST traversal, this design is not possible. Because the ADT must process the data in a traversal, the design requires that the application programmer develop the traversal-processing function and pass its address to the ADT. Given that there may be more than one process required for any application, the function must be passed when the traversal is initiated, not when the tree is created as we do with the search algorithm.

With the basic design understood, we are ready to describe the ADT. We define nine public functions and five private functions; that is, functions that are available only within the ADT. These functions make up the basic set needed to build and maintain BST trees. The private functions begin with an underscore character. The design is shown in Figure 7-11 and described in the following sections.

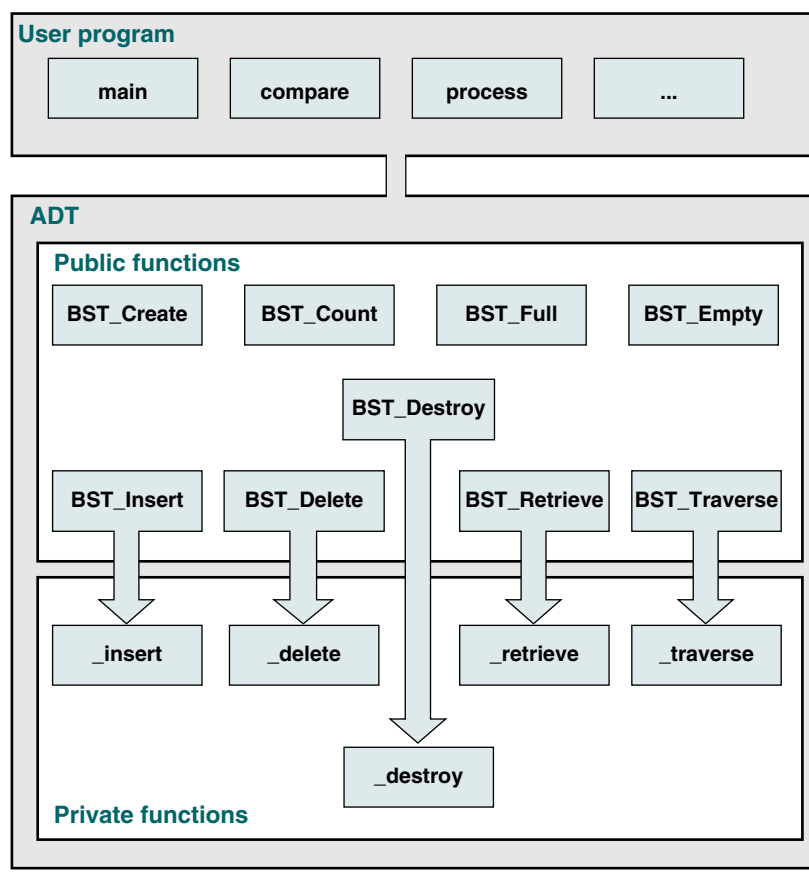


FIGURE 7-11 BST ADT Design

Data Structure

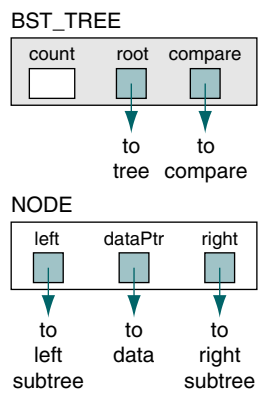
The BST needs two separate data structures: one for the head and one for the node.

Head Structure

As with the linear list, the BST data structure uses a simple head structure, `BST_TREE`, that contains a count, a root pointer, and the address of the compare function needed to search the list. The application program's only view of the tree is a pointer to the head structure, which is allocated from dynamic memory when the tree is created.

Node Structure

The BST nodes contain a data pointer and two self-referential pointers to the left and right subtrees. These data structures are shown in Figure 7-12.




```
typedef struct
  {
   int count;
   int  (*compare)
          (void* argu1,
           void* argu2);
   NODE* root;
  } BST_TREE;

typedef struct node
  {
   void*        dataPtr;
   struct node* left;
   struct node* right;
  } NODE;
```

FIGURE 7-12 BST Tree Data Structure

Algorithms

A basic set of BST tree algorithms is covered in this section. Depending on the application, other algorithms could be required. For example, some applications may need to include descending-key traversals. The ADT data structures and the prototype declarations are shown in Program 7-1.

PROGRAM 7-1 BST Declarations

```
1  /* Header file for binary search tree (BST). Contains
2     structural definitions and prototypes for BST.
3        Written by:
4        Date:
5  */
```

continued

PROGRAM 7-1 BST Declarations *(continued)*

```
 6  #include <stdbool.h>
 7
 8  // Structure Declarations
 9  typedef struct node
10     {
11      void*        dataPtr;
12      struct node* left;
13      struct node* right;
14     } NODE;
15
16  typedef struct
17     {
18      int   count;
19      int  (*compare) (void* argu1, void* argu2);
20      NODE*  root;
21     } BST_TREE;
22
23  // Prototype Declarations
24     BST_TREE* BST_Create
25              (int (*compare) (void* argu1, void* argu2));
26     BST_TREE* BST_Destroy (BST_TREE* tree);
27
28     bool  BST_Insert   (BST_TREE* tree, void* dataPtr);
29     bool  BST_Delete   (BST_TREE* tree, void* dltKey);
30     void* BST_Retrieve (BST_TREE* tree, void* keyPtr);
31     void  BST_Traverse (BST_TREE* tree,
32                         void (*process)(void* dataPtr));
33
34     bool BST_Empty (BST_TREE* tree);
35     bool BST_Full  (BST_TREE* tree);
36     int  BST_Count (BST_TREE* tree);
37
38     static NODE* _insert
39                    (BST_TREE* tree, NODE* root,
40                     NODE* newPtr);
41     static NODE* _delete
42                    (BST_TREE* tree,    NODE* root,
43                     void*     dataPtr, bool* success);
44     static void* _retrieve
45                    (BST_TREE* tree,
46                     void* dataPtr, NODE* root);
47     static void _traverse
48                    (NODE* root,
49                     void (*process) (void* dataPtr));
50     static void _destroy (NODE* root);
```

Create a BST

The abstract data type must be able to support multiple structures in one program. This is accomplished by allocating the tree head structure in dynamic memory. The create tree operation allocates the structure, sets its count to zero and the root pointers to null, and stores the address of the compare function. It then returns the tree pointer. The code is shown in Program 7-2.

PROGRAM 7-2 Create BST Application Interface

```
/* ================ BST_Create ================
   Allocates dynamic memory for an BST tree head
   node and returns its address to caller
      Pre    compare is address of compare function
             used when two nodes need to be compared
      Post   head allocated or error returned
      Return head node pointer; null if overflow
*/
BST_TREE* BST_Create
        (int  (*compare) (void* argu1, void* argu2))
{
// Local Definitions
   BST_TREE* tree;

// Statements
   tree = (BST_TREE*) malloc (sizeof (BST_TREE));
   if (tree)
      {
       tree->root    = NULL;
       tree->count   = 0;
       tree->compare = compare;
      } // if

   return tree;
}  // BST_Create
```

Insert a BST

The BST add node function is the module called by the application program. It receives a pointer to the tree structure and a pointer to the data to be inserted into the tree. After creating a node, it calls a recursive insert function to make the physical insertion. The code is shown in Program 7-3.

PROGRAM 7-3 Insert BST Application Interface

```
/* ================ BST_Insert ==================
   This function inserts new data into the tree.
      Pre    tree is pointer to BST tree structure
      Post   data inserted or memory overflow
      Return Success (true) or Overflow (false)
```

continued

PROGRAM 7-3 Insert BST Application Interface *(continued)*

```
 6  */
 7  bool BST_Insert (BST_TREE* tree, void* dataPtr)
 8  {
 9  // Local Definitions
10     NODE* newPtr;
11
12  // Statements
13     newPtr = (NODE*)malloc(sizeof(NODE));
14     if (!newPtr)
15        return false;
16
17     newPtr->right   = NULL;
18     newPtr->left    = NULL;
19     newPtr->dataPtr = dataPtr;
20
21     if (tree->count == 0)
22         tree->root  =  newPtr;
23     else
24         _insert(tree, tree->root, newPtr);
25
26     (tree->count)++;
27     return true;
28  }  // BST_Insert
```

Program 7-3 Analysis Because all additions take place at a leaf node, we store the data pointer in the newly created node and initialize the subtree pointers to null. We then call the recursive insert function. When it returns, we update the tree count and return success.

Note the name of the recursive insert module. Following the guide for system software, we call it `_insert` so that the name is not duplicated by application programmers when they use the ADT in their programs.

Internal Insert Function

The internal insert function, called initially by `BST_Insert`, requires three parameters: a pointer to the tree structure, a pointer to the root of the tree or the subtree, and a pointer to the node being inserted. The function is shown in Program 7-4. It follows the design in Algorithm 7-4.

PROGRAM 7-4 Internal Insert Function

```
1  /* ==================== _insert ====================
2     This function uses recursion to insert the new data
3     into a leaf node in the BST tree.
4        Pre    Application has called BST_Insert, which
5               passes root and data pointer
6        Post   Data have been inserted
7        Return pointer to [potentially] new root
```

continued

PROGRAM 7-4 Internal Insert Function *(continued)*

```
 8  */
 9  NODE* _insert (BST_TREE* tree, NODE* root, NODE* newPtr)
10  {
11  // Statements
12     if (!root)
13        // if NULL tree
14        return newPtr;
15
16     // Locate null subtree for insertion
17     if (tree->compare(newPtr->dataPtr,
18                       root->dataPtr) < 0)
19        {
20         root->left = _insert(tree, root->left, newPtr);
21         return root;
22        } // new < node
23     else
24        // new data >= root data
25        {
26         root->right = _insert(tree, root->right, newPtr);
27         return root;
28          } // else new data >= root data
29     return root;
30  }  // _insert
```

Program 7-4 Analysis This algorithm must be carefully studied to fully understand its logic. It begins with a recursive search to locate the correct insertion point in a leaf node. A leaf node is identified by a subtree pointer, either right or left, that is null. When we find a leaf pointer, we return the new node so that it can be inserted into the parent pointer (statement 14).

Because this is a recursive function, it must have a base case. Can you see it? The base case occurs when we locate a leaf and return **`newPtr`** in statement 14. At this point we begin to back out of the tree.

Delete a BST

We use the same design for deletion that we used for insertion. The application program interface sees only the BST head structure. It passes the tree and a pointer to a variable containing the key to be deleted. If the deletion is successful, the BST delete function returns true; if the node cannot be found, it returns false. The code for the application interface is shown in Program 7-5.

PROGRAM 7-5 Delete BST Application Interface

```
1  /* ================== BST_Delete ==================
2     This function deletes a node from the tree and
3     rebalances it if necessary.
4        Pre    tree initialized--null tree is OK
```

continued

PROGRAM 7-5 Delete BST Application Interface *(continued)*

```
5                dltKey is pointer to data structure
6                       containing key to be deleted
7          Post   node deleted and its space recycled
8                 -or- An error code is returned
9          Return Success (true) or Not found (false)
10   */
11   bool BST_Delete (BST_TREE* tree, void* dltKey)
12   {
13   // Local Definitions
14      bool  success;
15      NODE* newRoot;
16
17   // Statements
18      newRoot = _delete (tree, tree->root, dltKey,
19                         &success);
20      if (success)
21         {
22          tree->root = newRoot;
23          (tree->count)--;
24          if (tree->count == 0)
25              // Tree now empty
26              tree->root = NULL;
27         } // if
28      return success;
29   }  // BST_Delete
```

Program 7-5 Analysis The application interface delete function simply accepts the parameters from the user and reformats them for the recursive delete function. When the delete has been completed, it updates the tree count and, if the tree is empty, sets the root to null. It then returns true if the delete function was successful or false if the key could not be located.

Internal Delete Function

The real work is done by the internal delete function shown in Program 7-6. It uses the design in Algorithm 7-5.

PROGRAM 7-6 Internal Delete Function

```
1    /* ==================== _delete ====================
2       Deletes node from the tree and rebalances
3       tree if necessary.
4          Pre    tree initialized--null tree is OK
5                 dataPtr contains key of node to be deleted
6          Post   node is deleted and its space recycled
7                 -or- if key not found, tree is unchanged
8                success is true if deleted; false if not
9        Return pointer to root
```

continued

PROGRAM 7-6 Internal Delete Function *(continued)*

```
10  */
11  NODE*  _delete (BST_TREE* tree,    NODE* root,
12                  void*     dataPtr, bool* success)
13  {
14  // Local Definitions
15     NODE* dltPtr;
16     NODE* exchPtr;
17     NODE* newRoot;
18     void* holdPtr;
19
20  // Statements
21     if (!root)
22        {
23         *success = false;
24         return NULL;
25        } // if
26
27     if (tree->compare(dataPtr, root->dataPtr) < 0)
28          root->left  = _delete (tree,    root->left,
29                                 dataPtr, success);
30     else if (tree->compare(dataPtr, root->dataPtr) > 0)
31          root->right = _delete (tree,    root->right,
32                                 dataPtr, success);
33     else
34         // Delete node found--test for leaf node
35         {
36          dltPtr = root;
37          if (!root->left)
38              // No left subtree
39              {
40               free (root->dataPtr);       // data memory
41               newRoot = root->right;
42               free (dltPtr);              // BST Node
43               *success = true;
44               return newRoot;             // base case
45              } // if true
46          else
47              if (!root->right)
48                  // Only left subtree
49                  {
50                   newRoot = root->left;
51                   free (dltPtr);
52                   *success = true;
53                   return newRoot;         // base case
54                  } // if
55              else
56                  // Delete Node has two subtrees
57                  {
```

continued

PROGRAM 7-6 Internal Delete Function *(continued)*

```
58                  exchPtr = root->left;
59                  // Find largest node on left subtree
60                  while (exchPtr->right)
61                      exchPtr = exchPtr->right;
62
63                  // Exchange Data
64                  holdPtr          = root->dataPtr;
65                  root->dataPtr    = exchPtr->dataPtr;
66                  exchPtr->dataPtr = holdPtr;
67                  root->left       =
68                     _delete (tree,   root->left,
69                              exchPtr->dataPtr, success);
70                } // else
71          } // node found
72      return root;
73  }  // _delete
```

Program 7-6 Analysis

As we saw with the recursive insert, the first part of the function searches for the node to be deleted. If it reaches a leaf before finding the node to be deleted, the key cannot be found, so it returns false. This is the first of three base cases.

Once the node to be deleted is found, we determine if it is at a leaf or a leaflike node. Remember that deletions can take place only at a leaf. If a node has two subtrees, we must search for a leaf node to take its place. We first check to see if there is a left subtree. If there is none, we simply save the right subtree pointer as the pointer to take the root's place. If there is a left subtree, we check the right subtree pointer. If there is none, we save the left subtree pointer as the pointer to take the root's place. Assuming for the moment that there is zero or one subtree, we free the deleted node's memory and return true. These are the second (statement 44) and third (statement 53) base cases.

If the node to be deleted has two subtrees, we must find a node to take its place. Our design searches the left subtree for its largest node. When we find it, we move its data to replace the deleted data and then recursively call the delete function to delete what we know to be a valid leaf node. This logic is shown at statements 57 through 70.

Retrieve a BST

The retrieve function follows the left-right structure of the tree until the desired node is found. When it is located, the address of the data is returned to the calling function. If the data are not located, a null pointer is returned.

The design is similar to the insert and delete algorithms described above. An application interface function provides a pointer to the tree and a pointer to the key. We then call an ADT recursive function to locate the data. The code for the ADT retrieve function is shown in Program 7-7.

PROGRAM 7-7 Retrieve BST Application Interface

```
/* ================== BST_Retrieve ==================
   Retrieve node searches tree for the node containing
   the requested key and returns pointer to its data.
      Pre     Tree has been created (may be null)
              data is pointer to data structure
                   containing key to be located
      Post    Tree searched and data pointer returned
      Return  Address of matching node returned
              If not found, NULL returned
*/
void* BST_Retrieve  (BST_TREE* tree, void* keyPtr)
{
// Statements
   if (tree->root)
       return _retrieve (tree, keyPtr, tree->root);
   else
       return NULL;
}  // BST_Retrieve
```

Program 7-7 Analysis

The retrieve data function is quite simple. It is needed only because the recursive function needs an additional parameter, of which the using application is not aware. It simply passes the pointer to the tree structure and the key, adding the pointer to the tree root.

Internal Retrieve Function

The code for the internal retrieve function is shown in Program 7-8.

PROGRAM 7-8 Internal Retrieve Function

```
/* ==================== _retrieve ====================
   Searches tree for node containing requested key
   and returns its data to the calling function.
      Pre     _retrieve passes tree, dataPtr, root
              dataPtr is pointer to data structure
                 containing key to be located
      Post    tree searched; data pointer returned
      Return  Address of data in matching node
              If not found, NULL returned
*/
void* _retrieve (BST_TREE* tree,
                 void* dataPtr, NODE* root)
{
// Statements
   if (root)
```

continued

PROGRAM 7-8 Internal Retrieve Function *(continued)*

```
16          {
17           if (tree->compare(dataPtr, root->dataPtr) < 0)
18               return _retrieve(tree, dataPtr, root->left);
19          else if (tree->compare(dataPtr,
20                        root->dataPtr) > 0)
21              return _retrieve(tree, dataPtr, root->right);
22           else
23               // Found equal key
24               return root->dataPtr;
25          }  // if root
26       else
27          // Data not in tree
28          return NULL;
29    }  // _retrieve
```

Program 7-8 Analysis The retrieve function uses the compare function stored in the tree structure when the tree was created. If the data in the search argument, **dataPtr**, is less than the root data's key, it calls itself with the left subtree as the root. If the search argument is greater than the root data's key, it calls itself with the right subtree as the root. If the argument is not greater and not less than the root, it must be equal, so it returns the root's data pointer.

Carefully study the recursive call's functions in statements 18 and 21. Note that they return the address returned by the recursive call. In this fashion the address returned by the base cases is passed back up the tree structure until it is finally returned to the original application call by the interface function.

Traverse a BST

The traverse function uses an inorder traversal of the tree, calling the application-dependent process function when the node is to be processed. Although the traversal is standard, the application-processing function is not. Therefore, whenever the using application calls the traverse function, it must also pass the address of the function that processes the data. The processing function uses only one parameter: the address of the node to be processed. It in turn calls the ADT internal function that actually traverses the tree. The code is shown in Program 7-9.

PROGRAM 7-9 Traverse BST Application Interface

```
1   /* ================== BST_Traverse ===================
2      Process tree using inorder traversal.
3         Pre   Tree has been created (may be null)
4               process "visits" nodes during traversal
5         Post  Nodes processed in LNR (inorder) sequence
6   */
7   void BST_Traverse (BST_TREE* tree,
8                      void (*process) (void* dataPtr))
9   {
```

continued

PROGRAM 7-9 Traverse BST Application Interface *(continued)*

```
10  // Statements
11     _traverse (tree->root, process);
12     return;
13  }  // end BST_Traverse
```

Internal Traverse Function

The code for the internal search function is shown in Program 7-10.

PROGRAM 7-10 Internal Traverse Function

```
 1  /* ================== _traverse ==================
 2     Inorder tree traversal. To process a node, we use
 3     the function passed when traversal was called.
 4        Pre   Tree has been created (may be null)
 5        Post  All nodes processed
 6  */
 7  void _traverse (NODE* root,
 8                  void (*process) (void* dataPtr))
 9  {
10  // Statements
11  if  (root)
12      {
13       _traverse (root->left, process);
14       process   (root->dataPtr);
15       _traverse (root->right, process);
16      } // if
17  return;
18  }  // _traverse
```

Empty a BST

The empty funtion simply checks the tree count. If it is zero, it returns true; otherwise, it returns false. The code is seen in Program 7-11.

PROGRAM 7-11 Empty BST Application Interface

```
 1  /* ================== BST_Empty ==================
 2     Returns true if tree is empty; false if any data.
 3        Pre      Tree has been created. (May be null)
 4        Returns  True if tree empty, false if any data
 5  */
 6  bool BST_Empty (BST_TREE* tree)
 7  {
 8  // Statements
 9     return (tree->count == 0);
10  }  // BST_Empty
```

Full BST

Because there is no way to test for available memory in C, we must actually try to allocate a node. If we are successful, we delete it and return false—the list is not full. If the allocation fails, we return true; there is not enough memory for another node. The code is shown in Program 7-12.

PROGRAM 7-12 Full BST Application Interface

```
/* ===================== BST_Full ====================
   If there is no room for another node, returns true.
      Pre      tree has been created
      Returns  true if no room for another insert
               false if room
*/
bool BST_Full (BST_TREE* tree)
{
// Local Definitions
   NODE* newPtr;

// Statements
   newPtr = (NODE*)malloc(sizeof (*(tree->root)));
   if (newPtr)
      {
       free (newPtr);
       return false;
      } // if
   else
        return true;
}  // BST_Full
```

BST Count

The count function, shown in Program 7-13, simply returns the number of nodes currently in the tree.

PROGRAM 7-13 BST Count Application Interface

```
/* ================== BST_Count ==================
   Returns number of nodes in tree.
      Pre     tree has been created
      Returns tree count
*/
int BST_Count (BST_TREE* tree)
{
// Statements
   return (tree->count);
}  // BST_Count
```

Destroy a BST

The last function in the BST tree abstract data type is the destroy function. It is used to physically delete and free all of the data nodes and the tree head structure when the tree is no longer needed. Because we need to traverse the tree to find all of the data and nodes that need to be deleted, we call a recursive function to do the physical deletions.

The logic for the destroy function parallels the destroy functions we have discussed previously. The code is shown in Program 7-14.

PROGRAM 7-14 Destroy BST Application Interface

```
/* =============== BST_Destroy ==============
   Deletes all data in tree and recycles memory.
   The nodes are deleted by calling a recursive
   function to traverse the tree in inorder sequence.
      Pre      tree is a pointer to a valid tree
      Post     All data and head structure deleted
      Return   null head pointer
*/
BST_TREE* BST_Destroy (BST_TREE* tree)
{
// Statements
   if (tree)
      _destroy (tree->root);

   // All nodes deleted. Free structure
   free (tree);
   return NULL;
}  // BST_Destroy
```

Program 7-14 Analysis

The logic is simple. We first make sure that we have a valid tree by testing the tree pointer. If it is valid—that is, if it is not null—we call the recursive subfunction that does the physical deletions. When we return we delete the tree structure itself and return a null pointer.

Internal Destroy Function

The code for the internal deletion function is shown in Program 7-15.

PROGRAM 7-15 Internal Destroy Function

```
/* =============== _destroy ==============
   Deletes all data in tree and recycles memory.
   It also recycles memory for the key and data nodes.
   The nodes are deleted by calling a recursive
   function to traverse the tree in inorder sequence.
```

continued

PROGRAM 7-15 Internal Destroy Function *(continued)*

```
 6        Pre      root is pointer to valid tree/subtree
 7        Post     All data and head structure deleted
 8        Return   null head pointer
 9  */
10  void _destroy (NODE* root)
11  {
12  // Statements
13     if (root)
14        {
15         _destroy (root->left);
16         free (root->dataPtr);
17         _destroy (root->right);
18         free (root);
19        } // if
20     return;
21  }  // _destroy
```

Program 7-15 Analysis The logic in the recursive deletion function is a little more complex. The big question is: When do we delete the data, and when do we delete the node? We have to make sure that we do each only once and at the right time.

The data are deleted as we return from the left subtree. This is done at statement 16. This is the logical point at which we need to process a BST's data. However, we are not yet ready to delete the node because we have not processed the right subtree. We must wait until we return from the right subtree traversal to delete it (see statement 18).

7.4 BST Applications

Now that we've developed the BST ADT, let's write two applications that use it. The first is a BST that contains only integer data. The second is a simple application that lists the students in a class by their student number.

Integer Application

The BST tree integer application reads integers from the keyboard and inserts them into the BST. Figure 7-13 traces the first two insertions, 18 and 33, into a null BST. Use it to help follow the program.

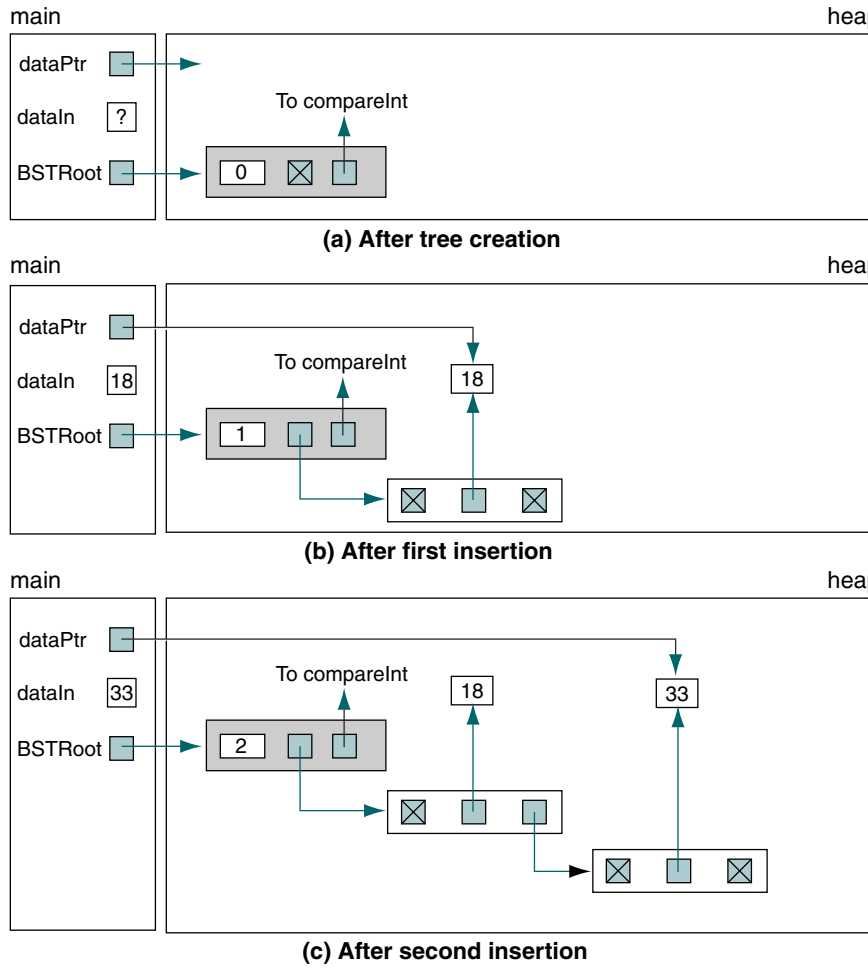


FIGURE 7-13 Insertions into a BST

The program is very simple. After creating a tree, we loop reading data from the keyboard and calling `BST_Insert` to insert it into the tree. When the tree is complete, we call the BST traverse function to print it. The code is shown in Program 7-16. Besides *main*, it also contains the compare and print functions.

PROGRAM 7-16 BST Integer Application

```
/* This program builds and prints a BST.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include "P7-BST-ADT.h"

// Prototype Declarations
int  compareInt (void* num1, void* num2);
void printBST   (void* num1);

int main (void)
{
// Local Definitions
   BST_TREE* BSTRoot;
   int*      dataPtr;
   int       dataIn = +1;

// Statements
   printf("Begin BST Demonstation\n");

   BSTRoot = BST_Create (compareInt);

// Build Tree
   printf("Enter a list of positive integers;\n");
   printf("Enter a negative number to stop.\n");

   do
      {
       printf("Enter a number: ");
       scanf ("%d", &dataIn);
       if (dataIn > -1)
          {
           dataPtr = (int*) malloc (sizeof (int));
           if (!dataPtr)
               {
                printf("Memory Overflow in add\n"),
                exit(100);
               } // if overflow
           *dataPtr = dataIn;
           BST_Insert (BSTRoot, dataPtr);
          } // valid data
      } while (dataIn > -1);

```

continued

PROGRAM 7-16 BST Integer Application *(continued)*

```
46      printf("\nBST contains:\n");
47      BST_Traverse (BSTRoot, printBST);
48
49      printf("\nEnd BST Demonstration\n");
50      return 0;
51   }  // main
52
53   /* ==================== compareInt ====================
54      Compare two integers and return low, equal, high.
55          Pre  num1 and num2 are valid pointers to integers
56          Post return low (-1), equal (0), or high (+1)
57   */
58   int compareInt (void* num1, void* num2)
59   {
60   // Local Definitions
61      int key1;
62      int key2;
63
64   // Statements
65      key1 = *(int*)num1;
66      key2 = *(int*)num2;
67      if (key1 < key2)
68          return -1;
69      if (key1 == key2)
70          return 0;
71      return +1;
72   }  // compareInt
73
74   /* ==================== printBST ====================
      Print one integer from BST.
75          Pre  num1 is a pointer to an integer
76          Post integer printed and line advanced
77   */
78   void printBST (void* num1)
79   {
80   // Statements
81      printf("%4d\n", *(int*)num1);
82      return;
83   }  // printBST
84
```

```
Results:
Begin BST Demonstation
Enter a list of positive integers;
Enter a negative number to stop.
Enter a number: 18
```

continued

PROGRAM 7-16 BST Integer Application *(continued)*

```
Enter a number: 33
Enter a number: 7
Enter a number: 24
Enter a number: 19
Enter a number: -1

BST contains:
   7
  18
  19
  24
  33

End BST Demonstration
```

Program 7-16 Analysis There are times when an application needs to use an ADT, but the ADT's design doesn't quite fit the application. This is one of those cases. The ADT expects a key and data. Our tree, however, needs only data. We solve the problem by sending the data both as a key and as data.

Student List Application

The second application creates a student list. It requires three pieces of data: the student's ID, the student's name, and the student's grade-point average. Students are added and deleted from the keyboard. They can be retrieved individually or as a list. The student structures and their relationships to the ADT are shown in Figure 7-14.

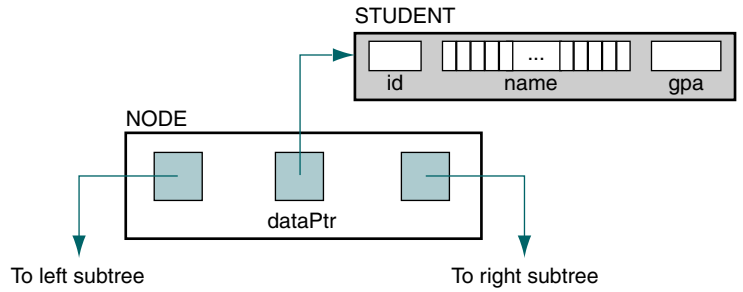


FIGURE 7-14 Student Data in ADT

Program Design

The program design is quite simple. We use a menu to insert, delete, print one student's data, or print a student list. To fully exercise the ADT, we also add one nonapplication function, test utilities, which calls the three utility functions.

Because this is a simple example, we use a loop in *main* to control the processing. In addition to the loop, *main* creates the file by calling the ADT and deletes it at the end, again by calling the ADT. Figure 7-15 shows the program design. We also show the two processing functions that are required by the ADT: compare and process. Although they are not directly used in the program, they are called by the ADT. We show their indirect usage with dashed lines. As is customary, the ADT calls are not shown in the design.

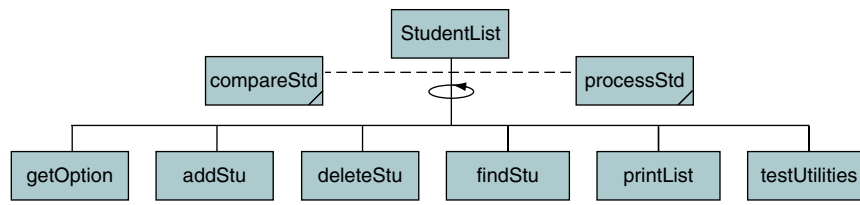


FIGURE 7-15 Student List Design

The program's data structure, its prototype declarations, its mainline, and all of the code are shown in Program 7-17. Although the program is long, the code is simple and straightforward.

PROGRAM 7-17 BST Student Application

```
/* This program builds and prints a student list.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#include "P7-BST-ADT.h"

// Structures
typedef struct
   {
    int   id;
    char  name[40];
    float gpa;
```

continued

PROGRAM 7-17 BST Student Application *(continued)*

```
   } STUDENT;

// Prototype Declarations
char getOption    (void);
void addStu       (BST_TREE* list);
void deleteStu    (BST_TREE* list);
void findStu      (BST_TREE* list);
void printList    (BST_TREE* list);
void testUtilties (BST_TREE* tree);
int  compareStu   (void* stu1, void* stu2);
void processStu   (void* dataPtr);

int main (void)
{
// Local Definitions
   BST_TREE* list;
   char      option = ' ';

// Statements
   printf("Begin Student List\n");
   list = BST_Create (compareStu);

   while ( (option = getOption ()) != 'Q')
      {
       switch (option)
          {
           case 'A': addStu (list);
                     break;
           case 'D': deleteStu (list);
                     break;
           case 'F': findStu (list);
                     break;
           case 'P': printList (list);
                     break;
           case 'U': testUtilties (list);
                     break;
         } // switch
      } // while
   list = BST_Destroy (list);

   printf("\nEnd Student List\n");
   return 0;
}  // main

/* ================== getOption ====================
   Reads and validates processing option from keyboard.
       Pre  nothing
```

continued

PROGRAM 7-17 BST Student Application *(continued)*

```
64          Post valid option returned
65    */
66    char getOption  (void)
67    {
68    // Local Definitions
69       char option;
70       bool error;
71
72    // Statements
73       printf("\n ======  MENU  ======\n");
74       printf(" A   Add Student\n");
75       printf(" D   Delete Student\n");
76       printf(" F   Find Student\n");
77       printf(" P   Print Class List\n");
78       printf(" U   Show Utilities\n");
79       printf(" Q   Quit\n");
80
81       do
82          {
83           printf("\nEnter Option: ");
84           scanf(" %c", &option);
85           option = toupper(option);
86           if   (option == 'A' || option == 'D'
87              || option == 'F' || option == 'P'
88              || option == 'U' || option == 'Q')
89                 error = false;
90           else
91                {
92               printf("Invalid option. Please re-enter: ");
93                error = true;
94               } // if
95          } while (error == true);
96       return option;
97    }  // getOption
98
99    /* ===================== addStu =====================
100      Adds a student to tree.
101          Pre  nothing
102          Post student added (abort on memory overflow)
103   */
104   void addStu (BST_TREE* list)
105   {
106   // Local Definitions
107      STUDENT* stuPtr;
108
109   // Statements
110      stuPtr = (STUDENT*)malloc (sizeof (STUDENT));
```

continued

PROGRAM 7-17 BST Student Application *(continued)*

```
111     if (!stuPtr)
112         printf("Memory Overflow in add\n"), exit(101);
113
114     printf("Enter student id:   ");
115     scanf ("%d",  &(stuPtr->id));
116     printf("Enter student name: ");
117     scanf ("%39s", stuPtr->name);
118     printf("Enter student gpa:  ");
119     scanf ("%f",  &(stuPtr->gpa));
120
121     BST_Insert (list, stuPtr);
122  }  // addStu
123
124  /* ==================== deleteStu ====================
125     Deletes a student from the tree.
126         Pre  nothing
127         Post student deleted or error message printed
128  */
129  void deleteStu (BST_TREE* list)
130  {
131  // Local Definitions
132     int  id;
133     int* idPtr = &id;
134
135  // Statements
136     printf("Enter student id: ");
137     scanf ("%d", idPtr);
138
139     if (!BST_Delete (list, idPtr))
140        printf("ERROR: No Student: %0d\n", *idPtr);
141  }  // deleteStu
142
143  /* ===================== findStu =====================
144     Finds a student and prints name and gpa.
145         Pre  student id
146         Post student data printed or error message
147  */
148  void findStu (BST_TREE* list)
149  {
150  // Local Definitions
151     int      id;
152     STUDENT* stuPtr;
153
154  // Statements
155     printf("Enter student id: ");
156     scanf ("%d", &id);
157
```

continued

PROGRAM 7-17 BST Student Application *(continued)*

```
158      stuPtr = (STUDENT*)BST_Retrieve (list, &id);
159      if (stuPtr)
160         {
161          printf("Student id:   %04d\n",  id);
162          printf("Student name: %s\n",    stuPtr->name);
163          printf("Student gpa:  %4.1f\n", stuPtr->gpa);
164         } // if
165      else
166         printf("Student %d not in file\n", id);
167   }  // findStu
168
169   /* ==================== printList ====================
170      Prints a list of students.
171          Pre  list has been created (may be null)
172          Post students printed
173   */
174   void printList  (BST_TREE* list)
175   {
176   // Statements
177      printf("\nStudent List:\n");
178      BST_Traverse (list, processStu);
179      printf("End of Student List\n");
180      return;
181   }  // printList
182
183   /* ================== testUtilties ==================
184      This function tests the ADT utilities by calling
185      each in turn and printing the results.
186         Pre  tree has been created
187         Post results printed
188   */
189   void testUtilties (BST_TREE* tree)
190   {
191   // Statements
192      printf("Tree contains %3d nodes: \n",
193             BST_Count(tree));
194      if (BST_Empty(tree))
195          printf("The tree IS empty\n");
196      else
197          printf("The tree IS NOT empty\n");
198      if (BST_Full(tree))
199          printf("The tree IS full\a\n");
200      else
201          printf("The tree IS NOT full\n");
202      return;
203   }  // testUtilities
204
```

continued

PROGRAM 7-17 BST Student Application *(continued)*

```
/* ==================== compareStu ====================
   Compare two student id's and return low, equal, high.
       Pre  stu1 and stu2 are valid pointers to students
       Post return low (-1), equal (0), or high (+1)
*/
int  compareStu (void* stu1, void* stu2)
{
// Local Definitions
   STUDENT s1;
   STUDENT s2;

// Statements
   s1 =  *(STUDENT*)stu1;
   s2 =  *(STUDENT*)stu2;

   if ( s1.id < s2.id)
         return -1;

   if ( s1.id == s2.id)
         return 0;

   return +1;
}  // compareStu

/* ==================== processStu ====================
   Print one student's data.
       Pre  stu is a pointer to a student
       Post data printed and line advanced
*/
void processStu (void* stuPtr)
{
// Local Definitions
   STUDENT aStu;

// Statements
   aStu = *(STUDENT*)stuPtr;
   printf("%04d  %-40s %4.1f\n",
          aStu.id, aStu.name, aStu.gpa);
   return;
}  // processStu
```

7.5 Threaded Trees

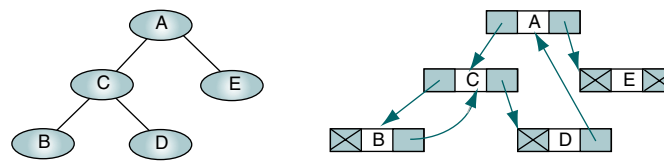
Binary tree traversal algorithms are written using either recursion or programmer-written stacks. If the tree must be traversed frequently, using stacks rather than recursion may be more efficient. A third alternative is a

threaded tree. In a threaded tree, null pointers are replaced with pointers to their successor nodes.

Using a stack for each call makes the binary tree traversal relatively inefficient, particularly if the tree must be traversed frequently. The reason we use recursion or a stack is that, at each step, we cannot access the next node in the sequence directly and we must use backtracking. The traversal is more efficient if the tree is a threaded tree.

For example, in the inorder traversal of a binary tree, we must traverse the left subtree, the node, and the right subtree. Because an inorder traversal is a depth-first traversal, we follow left pointers to the far-left leaf.

When we find the far-left leaf, we must begin backtracking to process the right subtrees we passed on our way down. This is especially inefficient when the parent node has no right subtree. Consider the tree shown in Figure 7-16(a).



(a) Binary tree (b) Threaded binary tree

FIGURE 7-16 Threaded Binary Tree

The inorder traversal of this tree is BCDAE. When we traverse the tree in inorder sequence, we follow the left pointers to get the first node, `B`. However, after locating the far-left node, we must go back to `C`, which is why we need recursion or a stack. Note that when we are processing `B`, we do not need recursion or a stack because `B`'s right subtree is empty.

Similarly, when we finish processing node `D` using recursion or stacks, we must return to node `C` before we go to `A`. But again, we do not need to pass through `C`. The next node to be processed is the root, `A`.

From this small example, it should be clear that the nodes whose right subtrees are empty create more work when we use recursion or stacks. This leads to the threaded concept: when the right subtree pointer is empty, we can use the pointer to point to the successor of the node. In other words, we can use the right null pointer as a thread. The threaded tree is shown in Figure 7-16(b).

To build a threaded tree, first build a standard binary search tree. Then traverse the tree, changing the null right pointers to point to their successors.

The traversal for a threaded tree is straightforward. Once you locate the far-left node, you loop, following the thread (the right pointer) to the next node. No recursion or stack is needed. When you find a null thread (right pointer), the traversal is complete.

7.6 Key Terms

binary search tree (BST)
binary tree search
binary tree traversal
delete a node
find largest node
find smallest node
insert node
threaded tree

7.7 Summary

- A binary search tree is a binary tree with the following properties:
  - All items in the left subtree are less than the root.
  - All items in the right subtree are greater than or equal to the root.
  - Each subtree is itself a binary search tree.
- The inorder traversal of a binary search tree produces an ordered list.
- In a binary search tree, the node with the smallest value is the far-left node in the tree. To find the smallest node, we simply follow the left branches until we get to a null left pointer.
- In a binary search tree, the node with the largest value is the far-right node. To find the largest node, we follow the right branches until we get to a null right pointer.
- To search for a value in a binary search tree, we first compare the target value with the root. If the target value is smaller than the root, we repeat the procedure for the left subtree. If the target value is greater than the root, we repeat the procedure for the right subtree. If the target value is equal to the root, the search is complete.
- To insert a node in a binary search tree, we follow the left or right branch down the tree, depending on the value of the new node, until we find a null subtree.
- To delete a node from a subtree, we must consider four cases:
  1. The node to be deleted has no children.
  2. The node to be deleted has only a left subtree.
  3. The node to be deleted has only a right subtree.
  4. The node to be deleted has two subtrees.
- To facilitate the traversal of a BST, threaded BSTs were created.
- In a threaded BST, null pointers are replaced with pointers to the successor node in a specific traversal order.

7.8 Practice Sets

Exercises

1. Draw all possible binary search trees for the data elements 5, 9, and 12.
2. Create a binary search tree using the following data entered as a sequential set:

```
14 23 7 10 33 56 80 66 70
```

3. Create a binary search tree using the following data entered as a sequential set:

```
7 10 14 23 33 56 66 70 80
```

4. Create a binary search tree using the following data entered as a sequential set:

```
80 70 66 56 33 23 14 10 7
```

5. Insert 44 and 50 into the tree created in Exercise 2.
6. Insert 44 and 50 into the tree created in Exercise 3.
7. Insert 44 and 50 into the tree created in Exercise 4.
8. Which one of the trees in Figure 7-17 is a valid binary search tree and which one is not? Explain your answer.

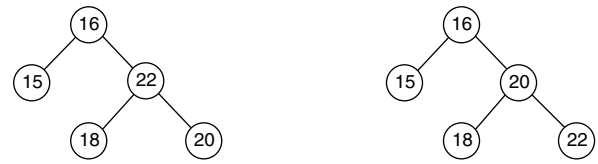


FIGURE 7-17 Figure for Exercise 8

9. Traverse the binary search tree in Figure 7-18 using an inorder traversal.

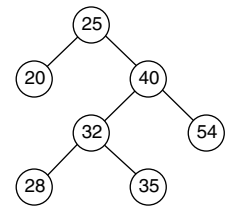


FIGURE 7-18 Figure for Exercise 9

10. The binary search tree in Figure 7-19 was created by starting with a null tree and entering data from the keyboard. In what sequence were the data entered? If there is more than one possible sequence, identify the alternatives.

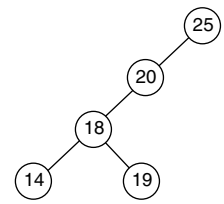


FIGURE 7-19 Figure for Exercise 10

11. The binary search tree in Figure 7-20 was created by starting with a null tree and entering data from the keyboard. In what sequence were the data entered? If there is more than one possible sequence, identify the alternatives.

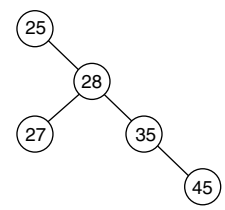


FIGURE 7-20 Figure for Exercise 11

12. Insert 44, 66, and 77 into the binary search tree in Figure 7-21.

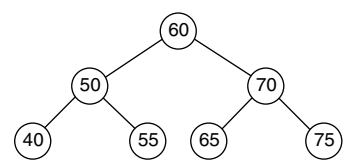


FIGURE 7-21 Figure for Exercise 12

13. Delete the node containing 60 from the binary search tree in Figure 7-22.

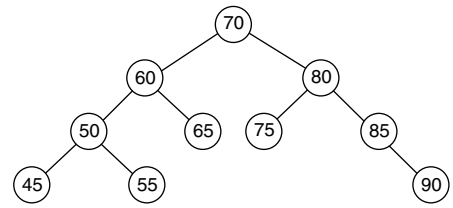


FIGURE 7-22 Figure for Exercises 13 and 14

14. Delete the node containing 85 from the binary search tree in Figure 7-22.

Problems

15. Develop a nonrecursive algorithm for Algorithm 7-1, "Find Smallest Node in a BST."
16. Develop a nonrecursive algorithm for Algorithm 7-3, "Search BST."
17. Write the C code for Algorithm 7-1, "Find Smallest Node in a BST."
18. Write the C code for Algorithm 7-4, "Add Node to BST."
19. Write the C code for Algorithm 7-5, "Delete Node from BST."

Projects

20. When writing BST algorithms, you need to be able to print the tree in a hierarchical order to verify that the algorithms are processing the data correctly. Write a print function that can be called to print the tree. The printed output should contain the node level number and its data. Present the tree using a bill-of-materials format, as shown in Figure 7-23, for several popular breeds of dogs recognized by the American Kennel Club (AKC).

```
1. Labrador
   2. German Shepherd
      3. Cocker Spaniel
         4. Beagle
         4. Dachshund
            5. Dalmatian
      3. Golden Retriever
   2. Rottweiler
      3. Poodle
      3. Shetland Sheepdog
```

FIGURE 7-23 Top Popular Dog Breeds Recognized by the AKC

21. Rewrite the print algorithm in Project 20 to print the data hierarchically. A vertical presentation of the data in Figure 7-23 is shown in Figure 7-24. If you tilt the page sideways, the data are presented in the binary tree format.

```
            3. Shetland Sheepdog
         2. Rottweiler
            3. Poodle
   1. Labrador
            3. Golden Retriever
         2. German Shepherd
                  5. Dalmatian
               4. Dachshund
            3. Cocker Spaniel
               4. Beagle
```

FIGURE 7-24 AKC Data Presented Hierarchically

22. Write a program that reads a list of names and telephone numbers from a text file and inserts them into a BST tree. Once the tree has been built, present the user with a menu that allows him or her to search the list for a specified name, insert a new name, delete an existing name, or print the entire phone list. At the end of the job, write the data in the list back to the file. Test your program with at least 10 names.
23. Create the ADT for a binary search tree using the array implementation. In an array implementation, the pointers become indexes to the subtree elements. When you create the tree, you need to know the maximum number of nodes to be stored in the tree. To test the ADT, use it to run the program in Project 22.
24. Write a program that processes a threaded binary tree. The program should first build the tree, then use an iterative traversal to process it using the threads.
25. Rework Project 24 using a preorder traversal.
26. Rework Project 24 using a postorder traversal.

Chapter 8
AVL Search Trees

While the binary search tree is simple and easy to understand, it has one major problem: it is not balanced. In this chapter we discuss the AVL tree, which is balanced. We develop the AVL tree structure and algorithms and give examples of its use. Then, at the end of the chapter, we discuss the AVL tree as an abstract data type and write a program that uses it.

8.1 AVL Tree Basic Concepts

In 1962, two Russian mathematicians, G. M. Adelson-Velskii and E. M. Landis, created the balanced binary tree structure that is named after them—the AVL tree. An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1. It is thus a balanced binary tree. To understand the significance of the trees being balanced, let's look at two different trees containing the same data. The first is a search tree in which each node is larger than its predecessor. The second is an AVL tree. These trees are shown in Figure 8-1.

As you study the tree in Figure 8-1(a), you should quickly note that it is the equivalent of a linear list in a binary tree's clothing. It takes two tests to locate 12. It takes three tests to locate 14. It takes eight tests to locate 52. In other words, the search effort for this particular binary search tree is $O(n)$.

Now consider the tree in Figure 8-1(b). Although we have labeled it an AVL tree, it looks like a binary search tree because both trees are built on the same structure. Locating nodes 8 and 14 requires four tests. Locating nodes 20 and 52 requires three tests. In other words, the maximum search effort for this tree is either three or four. Its search effort is $O(\log n)$. In this sample of two small trees, we see that the worst search effort was reduced from eight to four by simply balancing the tree. For a tree with 1000 nodes, the worst case for a completely unbalanced tree is 1000, whereas the worst case for a nearly

complete tree is 10. We thus see that balancing a tree can lead to significant performance improvements.

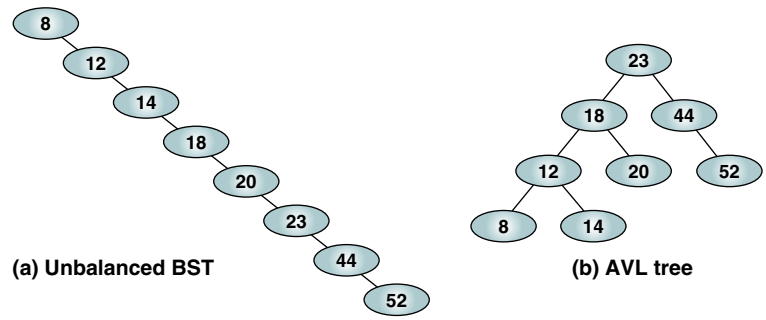


FIGURE 8-1 Two Binary Trees

We are now ready to define an AVL tree. An **AVL tree** is a binary tree that either is empty or consists of two AVL subtrees, T_L, and T_R, whose heights differ by no more than 1, as shown below.

$$|H_L - H_R| <= 1$$

In this formula, H_L is the height of the left subtree and H_R is the height of the right subtree (the bar symbols indicate absolute value). Because AVL trees are balanced by working with their height, they are also known as **height-balanced trees**.

An AVL tree is a height-balanced binary search tree.

Referring back to Figure 8-1(b), we see that the tree appears to be balanced when we look at the root. The height of the left subtree (18) is 3; the height of the right subtree (44) is 2. Therefore, if the subtrees 18 and 44 are balanced, the tree is balanced, as shown in Figure 8-2(a).

Now examine the left subtree, 18, in Figure 8-2(b). It also appears to be balanced because the heights of its subtrees differ by only 1. If its subtrees 12 and 20 are balanced, it is a balanced tree. Looking at Figure 8-2(c), we see that the right subtree (44) is balanced because the heights of its subtrees differ by only 1. Prove to yourself that the remaining subtrees are also balanced, which makes the tree balanced.

AVL Tree Balance Factor

In Chapter 6 we defined the balance factor as the height of the left subtree minus the height of the right subtree. Because the balance factor for any

node in an AVL tree must be +1, 0, or –1, we use the descriptive identifiers `LH` for left high (+1) to indicate that the left subtree is higher than the right subtree, `EH` for even high (0) to indicate that the subtrees are the same height, and `RH` for right high (–1) to indicate that the left subtree is shorter than the right subtree. We find that this system allows us to more easily concentrate on the structure; and because most programming languages provide methods for implementing identifiers for constant values, it should prove no problem when implementing the algorithms. These balance factors are shown in Figure 8-2.

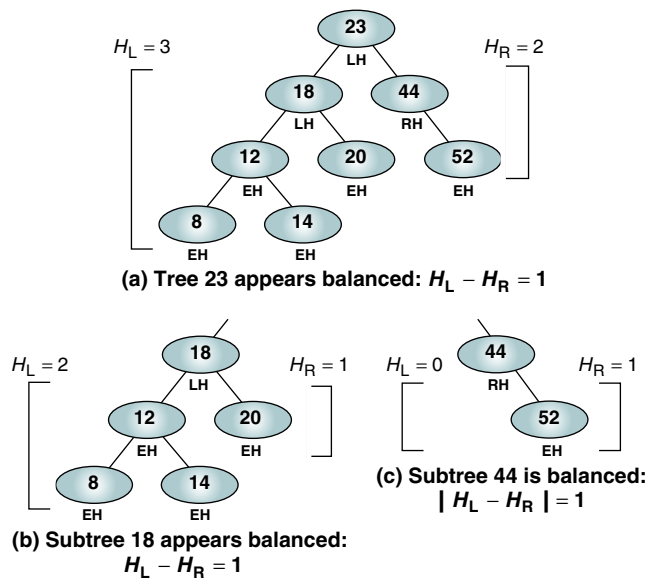


FIGURE 8-2 AVL Tree

Balancing Trees

Whenever we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced. When we detect that a tree has become unbalanced, we must rebalance it. AVL trees are balanced by rotating nodes either to the left or to the right. In this section we discuss the basic balancing algorithms.

We consider four cases that require rebalancing. All unbalanced trees fall into one of these four cases:

1. **Left of left**—A subtree of a tree that is left high has also become left high.
2. **Right of right**—A subtree of a tree that is right high has also become right high.
3. **Right of left**—A subtree of a tree that is left high has become right high.
4. **Left of right**—A subtree of a tree that is right high has become left high.

These four cases, created by insertions, are shown in Figure 8-3. Similar cases can be created for deletions.

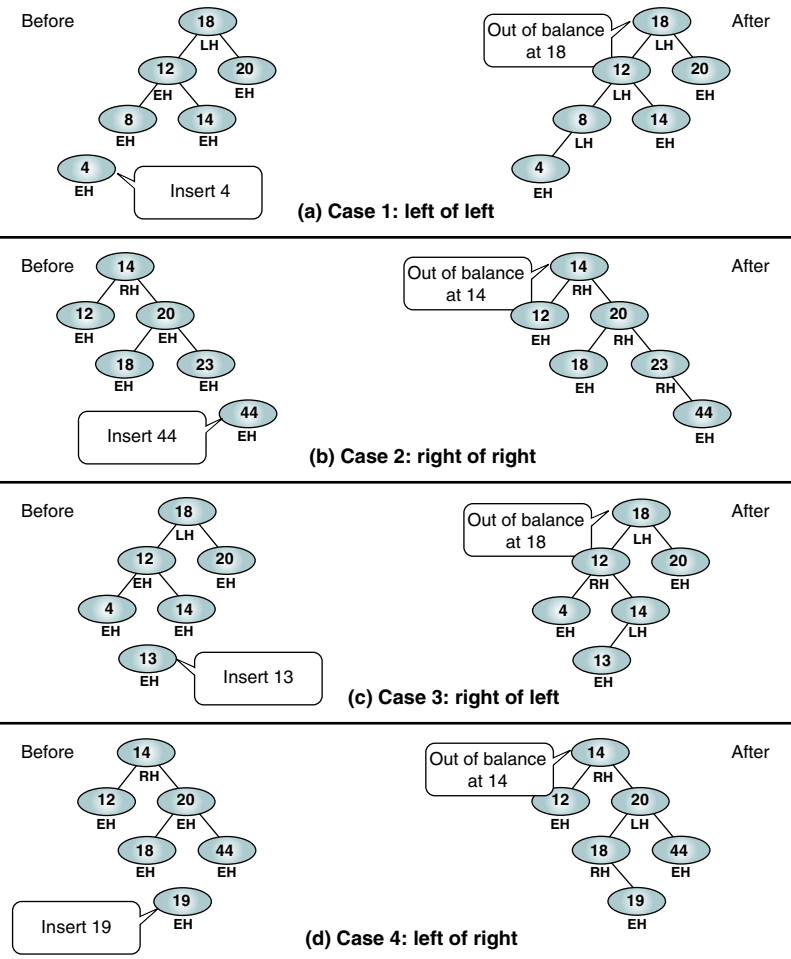


FIGURE 8-3 Out-of-balance AVL Trees

Case 1: Left of Left

When the out-of-balance condition has been created by a left high subtree of a left high tree, we must balance the tree by rotating the out-of-balance node to the right. Let's begin with a simple case. In Figure 8-4(a) node 20 (the tree) is out of balance because the left subtree 18 is left high and it is on the left branch of node 20, which is also left high. In this case we balance the tree by rotating the root, 20, to the right so that it becomes the right subtree of 18.

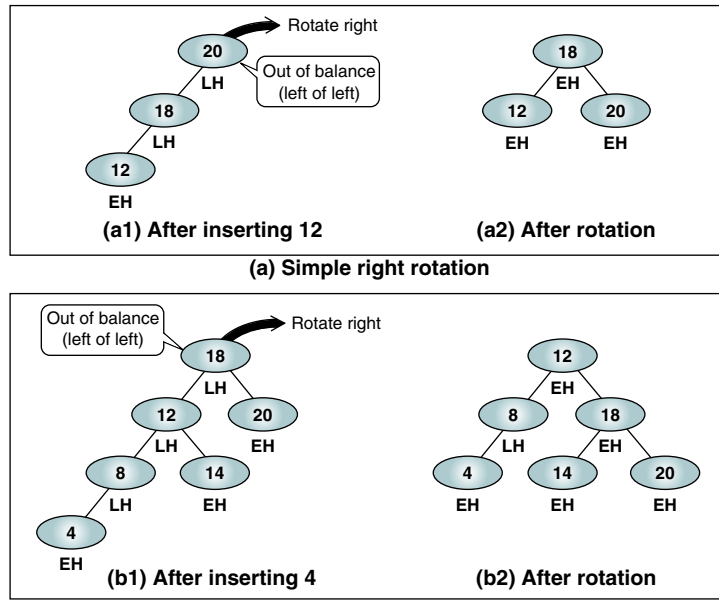


FIGURE 8-4 Left of Left—Single Rotation Right

The tree in Figure 8-4(b) presents a more complex problem. The subtree 12 is balanced, but the whole tree (18) is not. Therefore, we must rotate 18 to the right, making it the right subtree of the new root, 12. This creates a problem, however. What can we do with node 14, the current right subtree of 12? If you study the tree carefully, you will note that the old root, 18, loses its left subtree (12) in the rotation (12 becomes the root). We can therefore use 18's empty left subtree to attach 14, which also preserves the search tree relationship that all nodes on the right of the root must be greater than or equal to the root.

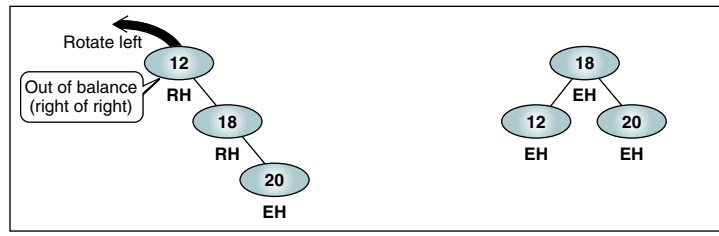
In both cases in Figure 8-4, after the rotation the rotated node is even high. Other balance factors may also have changed along the way, depending on the complexity of the tree and how the out-of-balance condition was created. We will see how the balance factors are changed when we study the insert and delete algorithms.

Case 2: Right of Right

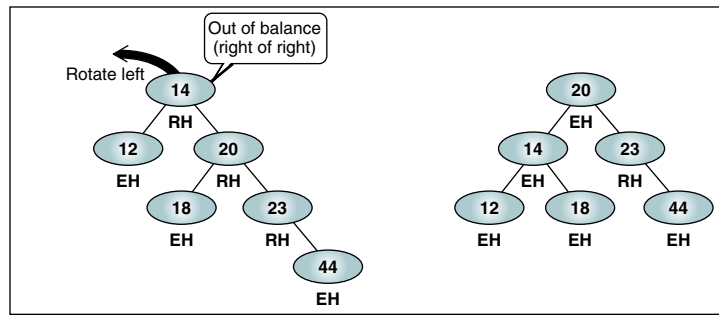
Case 2 is the mirror of Case 1. Figure 8-5(a) contains a simple left rotation. The subtree 18 is balanced, but the root is not. We therefore rotate the root to the left, making it the left subtree of the new root, 18.

Now let's look at the more complex case shown in Figure 8-5(b). In this case we have a right high root with a right high subtree and thus a right of

right, out-of-balance condition. To correct the imbalance, we rotate the root to the left, making the right subtree, 20, the new root. In the process 20's left subtree is connected as the old root's right subtree, preserving the order of the search tree.



(a) Simple left rotation



(b) Complex left rotation

FIGURE 8-5 Right of Right—Single Rotation Left

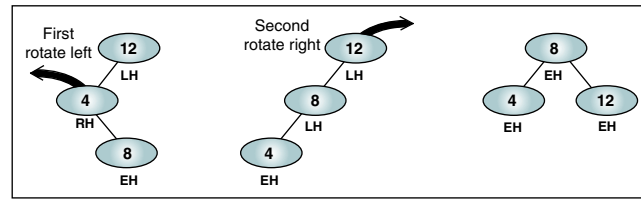
Case 3: Right of Left

The first two cases required single rotations to balance the trees. We now study two out-of-balance conditions in which we need to rotate two nodes, one to the left and one to the right, to balance the tree.

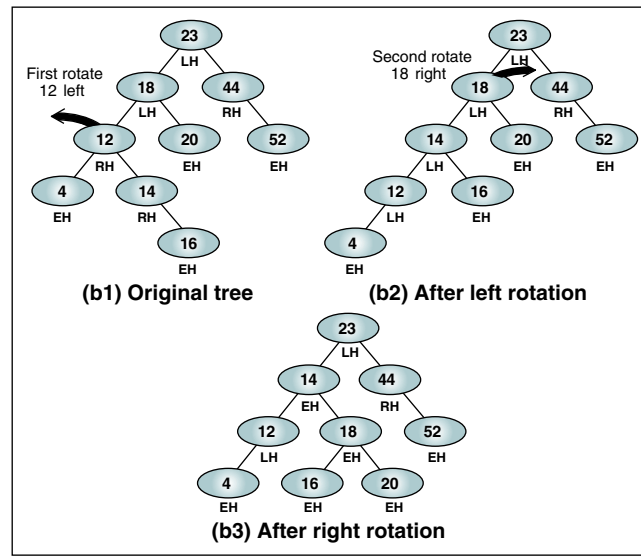
Again, let's start with a relatively simple case. In Figure 8-6(a) we see an out-of-balance tree in which the root is left high and the left subtree is right high—a right of left tree. To balance this tree, we first rotate the left subtree to the left, then we rotate the root to the right, making the left node the new root.

Now let's examine the complex case. In Figure 8-6(b) we see an out-of-balance tree in which a node (18) is left high and its left subtree (12) is right high. To balance this tree, we first rotate the left subtree (12) of the node that is out of balance (18) to the left, which aligns the subtree nodes in search tree sequence. This rotation is shown in Figure 8-6(b2). We now rotate the left subtree (18) to the right, which results in the new left subtree's (14) being balanced.

In this example the resulting tree is still left high, and the newly balanced subtree, 14, is even high.



(a) Simple double rotation right



(b) Complex double rotation right

FIGURE 8-6 Right of Left—Double Rotation Right

Case 4: Left of Right

Case 4 is also complicated. Figure 8-7(a) shows a simple case. To balance the tree, we first rotate the right subtree (44) right and then rotate the root (12) left.

Figure 8-7(b) shows the complex case. In this example the subtree 44 is balanced, but the tree is not. Because the out-of-balance condition is created by a left high subtree on a right high branch, we must double rotate. We begin by rotating the right subtree (44) to the right, which creates the tree in Figure 8-7(b2). We then rotate the root left, which gives us the balanced tree in Figure 8-7(b3). Note that the root of the balanced tree has an even high balance factor.

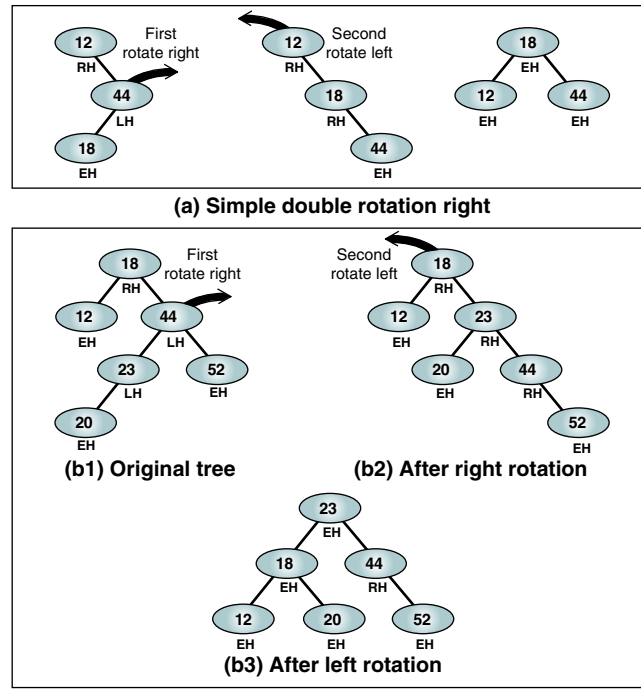


FIGURE 8-7 Left of Right—Double Rotation Right

8.2 AVL Tree Implementations

In this section we discuss the AVL tree insert and delete algorithms, including the algorithms to maintain the tree balance.

Insert into AVL Tree

The search and retrieval algorithms are the same as for any binary tree. Naturally, we use an inorder traversal because AVL trees are search trees. The insert and delete algorithms, however, must be rewritten.

As with the binary search tree, all inserts take place at a leaf node. To find the appropriate leaf node, we follow the path from the root, going left when the new data's key is less than a node's key and right otherwise. Once we have found a leaf, we connect it to its parent node and begin to back out of the tree. Here is where the AVL tree differs from the binary search tree. *As we back out of the tree, we constantly check the balance of each node*. When we find that a node is out of balance, we balance it and then continue up the tree.

Not all inserts create an out-of-balance condition. When we add a node on the right branch of a left high node, automatic balancing occurs and the node

is now even high, as shown in Figure 8-8. Conversely, when we add a node on the left branch of a right high node, the node is automatically balanced.

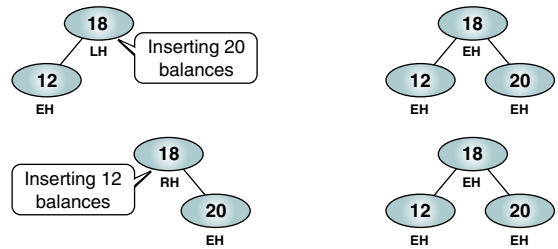


FIGURE 8-8 Automatic AVL Tree Balancing

When the leaf is evenly balanced before the add, however, the subtree that the leaf is on grows one level. Whether this addition creates an out-of-balance condition depends on the balance of its ancestor's nodes. As we back out of the insertion, if the tree has grown, we need to rotate to rebalance.

The design of the AVL tree insert algorithm is shown in Figure 8-9.[1] It clearly shows that inserting on the right branch of a tree is a mirror of inserting on the left. On the left in the structure chart is the design for inserting on a left branch. On the right is the design for inserting on a right branch. Both start with a recursive call to locate a leaf. Once we locate the leaf and begin to back out of the recursion, we call either left balance or right balance if the insert created a taller tree.

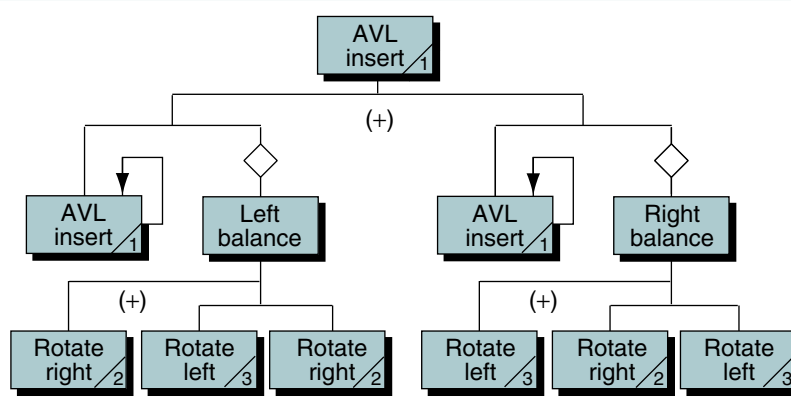


FIGURE 8-9 AVL Tree Insert Design

1. For an explanation of structure chart notations, refer to Appendix B.

To balance a left subtree node, we rotate either singly to the right or doubly, first to the left and then to the right. To balance a right subtree node, we rotate either singly to the left or doubly, first to the right and then to the left. Before you move on to the algorithms, compare this design with the balancing figures in Section 8.1, "AVL Tree Basic Concepts" (Figures 8-3 through 8-7).

AVL Tree Insert Algorithm

The AVL tree insert pseudocode is shown in Algorithm 8-1. This algorithm requires that the space be allocated for the new node before it is called. The calling algorithm therefore manages memory.

ALGORITHM 8-1 AVL Tree Insert

```
Algorithm AVLInsert (root, newData)
Using recursion, insert a node into an AVL tree.
   Pre    root is pointer to first node in AVL tree/subtree
          newData is pointer to new node to be inserted
   Post   new node has been inserted
   Return root returned recursively up the tree
 1 if (subtree empty)
   Insert at root
   1  insert newData at root
   2  return root
 2 end if
 3 if (newData < root)
   1  AVLInsert (left subtree, newData)
   2  if (left subtree taller)
      1  leftBalance (root)
   3  end if
 4 else
   New data >= root data
   1  AVLInsert (right subtree, newPtr)
   2  if(right subtree taller)
      1  rightBalance (root)
   3  end if
 5 end if
 6 return root
end AVLInsert
```

Algorithm 8-1 Analysis

As we discussed earlier, the algorithm begins much the same as the binary search tree add node (Algorithm 7-4). If we are at a leaf, we set the root to point to the new node.

When we return from the recursive call, there are two possibilities: either the tree has grown taller or it hasn't. If it is not taller, no balancing is required. If the tree is taller, there are again two possibilities: we need to balance if the tree is left high and we inserted on the left (statement 3.2) or if the tree is right high and we inserted on the right (statement 4.2).

One final note: This algorithm does not check for duplicate insertions. If an application does not allow duplicates, the algorithm must search before it inserts to ensure that the insertion is not a duplicate.

AVL Tree Left Balance Algorithm

Because the logic for balancing a left subtree and the logic for balancing a right subtree are mirrors of each other, we show only the logic to balance a left subtree. If you understand it, you should be able to construct the algorithm for balancing a right subtree. The pseudocode for balancing a left high node is shown in Algorithm 8-2.

ALGORITHM 8-2 AVL Tree Left Balance

```
Algorithm leftBalance (root)
This algorithm is entered when the root is left high (the
left subtree is higher than the right subtree).
   Pre    root is a pointer to the root of the [sub]tree
   Post   root has been updated (if necessary)
 1 if (left subtree high)
   1  rotateRight (root)
 2 else
   1  rotateLeft (left subtree)
   2  rotateRight (root)
 3 end if
end leftBalance
```

Rotate Algorithms

To rotate a node, we simply exchange the root and the appropriate subtree pointers. This process takes four steps: three for the exchange and one to reset the root pointer. The steps to rotate a left high tree are shown in Figure 8-10. Note that in this figure we are dealing with just a portion of a tree. Therefore, we cannot determine the final balance factors.

The pseudocode for rotate right and rotate left shown in Algorithm 8-3.

ALGORITHM 8-3 Rotate AVL Tree Right and Left

```
Algorithm rotateRight (root)
This algorithm exchanges pointers to rotate the tree right.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange left subtree with right subtree of left subtree
 2 make left subtree new root
end rotateRight

Algorithm rotateLeft (root)
This algorithm exchanges pointers to rotate the tree left.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange right subtree with left subtree of right subtree
 2 make right subtree new root
end rotateLeft
```

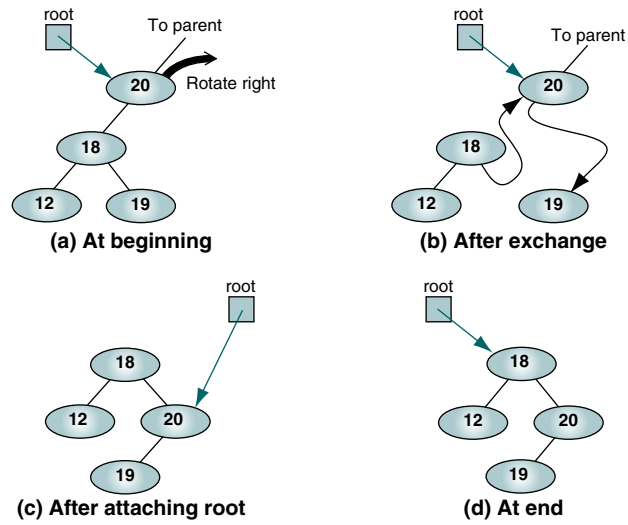


FIGURE 8-10 AVL Tree Rotate Right

AVL Tree Delete Algorithm

As we saw with the binary search tree, all deletions must take place at a leaf node. AVL tree deletion (Algorithm 8-4) follows the basic logic of the binary search tree deletion with the addition of the logic to balance the tree. As with the insert logic, the balancing occurs as we back out of the tree.

ALGORITHM 8-4 AVL Tree Delete

```
Algorithm AVLDelete (root, dltKey, success)
This algorithm deletes a node from an AVL tree and
rebalances if necessary.
   Pre    root is a pointer to a [sub]tree
          dltKey is the key of node to be deleted
          success is reference to boolean variable
   Post   node deleted if found, tree unchanged if not
          success set true (key found and deleted)
             or false (key not found)
   Return pointer to root of [potential] new subtree
1  if Return (empty subtree)
      Not found
   1  set success to false
   2  return null
2  end if
3  if (dltKey < root)
   1  set left-subtree to AVLDelete(left subtree, dltKey,
                                    success)
```

continued

ALGORITHM 8-4 AVL Tree Delete *(continued)*

```
   2  if (tree shorter)
      1  set root to deleteRightBalance(root)
   3  end if
 4 elseif (dltKey > root)
   1  set right subtree to AVLDelete(root->right, dltKey,
                                     success)
   2  if (tree shorter)
      1  set root to deleteLeftBalance (root)
   3  end if
 5 else
      Delete node found--test for leaf node
   1  save root
   2  if (no right subtree)
      1  set success to true
      2  return left subtree
   3  elseif (no left subtree)
         Have right but no left subtree
      1  set success to true
      2  return right subtree
   4  else
         Deleted node has two subtrees
         Find substitute--largest node on left subtree
      1  find largest node on left subtree
      2  save largest key
      3  copy data in largest to root
      4  set left subtree to AVLDelete(left subtree,
                                       largest key, success)
      5  if (tree shorter)
         1  set root to dltRightBal (root)
      6  end if
   5  end if
 6 end if
 7 return root
end AVLDelete
```

Algorithm 8-4 Analysis

Although the basic logic parallels that of the binary search tree delete algorithm, there are significant differences. It begins much the same: if we find a null tree, the node we are trying to delete does not exist. In this case we set success to false. We then return a null root. Returning a null root is necessary because the tree's or subtree's root may have changed during rotation. The original calling algorithm must therefore save the new root and then test for success before automatically changing the tree's root, just in case the delete key was not found. This logic is shown below.

```
1  set newTree to AVLDelete (root, dltKey, success)
2  if (success is true)
   1  set root to newTree
1  else
   2  error (delete key "not found")
2  end if
```

Once we find the node to be deleted (statement 5), we need to determine whether the deleted node is a leaf. If there is no right subtree, we set success to true and return the left subtree (which, as we saw in the binary search tree delete algorithm, may be null also). If there is a right subtree but not a left subtree, we set success to true and return the right subtree.

If there is a left subtree and a right subtree, we need to find a node to take the deleted node's place. After it is located, we copy its data to the root node. We then continue the delete operation at the root node's left subtree, this time looking to delete the largest key's node, which is now redundant. If after deleting the copied data from its leaf the tree is shorter, call delete right to balance it.

Delete Right Balance

Again, we show only one side of the balancing algorithm, delete right balance. We also exclude the logic for resetting the balance factors. We develop this logic in the ADT implementation. The logic for delete left balance mirrors that for Algorithm 8-5.

ALGORITHM 8-5 AVL Tree Delete Right Balance

```
Algorithm deleteRightBalance (root)
The [sub]tree is shorter after a deletion on the left branch.
If necessary, balance the tree by rotating.
   Pre    tree is shorter
   Post   balance restored
   Return new root
 1 if (tree not balanced)
      No rotation required if tree left or even high
   1  set rightOfRight to right subtree
   2  if (rightOfRight left high)
         Double rotation required
      1  set leftOfRight to left subtree of rightOfRight
         Rotate right then left
      2  right subtree = rotateRight (rightOfRight)
      3  root          = rotateLeft (root)
   3  else
         Single rotation required
      1  set root to rotateLeft (root)
   4  end if
 2 end if
 3 return root
end deleteRightBalance
```

Algorithm 8-5 Analysis

We begin by determining whether we need to balance the tree. Remember, we do not necessarily need to rebalance the tree just because we deleted a node. For example, if the root was left high, it is now even high and we can exit after setting the balance factor.

If we have deleted on the left and the tree was already right high, it is now doubly right high (the delete operation is right of right). We therefore need to rotate the right subtree to the left.

If the right subtree is left high, we need to rotate twice, first to the right and then to the left. This case is shown in Figure 8-7. If the right subtree is not left high, we need to

rotate only once. This situation is shown in Figure 8-5. Because we deleted on the left, we need to rotate the root to the left. These rotations are shown in Figure 8-11.

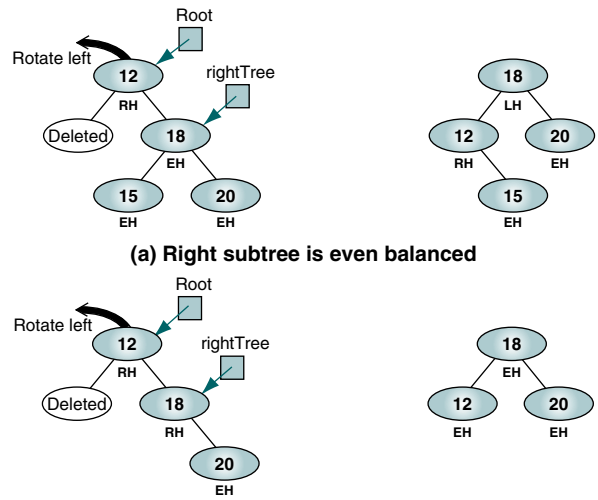


FIGURE 8-11 AVL Tree Delete Balancing

Duplicate Keys in AVL Trees

If duplicate keys are allowed in an AVL tree, it is possible for an equal key to be rotated to the left subtree. Consider the following example.

This example demonstrates that while insert operations always add duplicates on the right, delete operations can move them to the left.

The problem then becomes, how do we determine which node to delete? The recursive inorder search always locates the first node on a right branch; it cannot locate a node in the left subtree. The solution is to write an iterative search that locates the node just before the node with the desired key.

Adjusting the Balance Factors

After an insertion or deletion, as we balance the tree we must adjust the balance factors. Although the adjustments to the balance factors must be analyzed individually for each case, there is a general pattern:

1. If the root was even balanced before an insert, it is now high on the side in which the insert was made.

2. If an insert was in the shorter subtree of a tree that was not even balanced, the root is now even balanced.
3. If an insert was in the higher subtree of a tree that was not even balanced, the root must be rotated.

We develop the code for adjusting the balance factors fully in the implementation of the AVL tree abstract data type.

8.3 AVL Tree Abstract Data Type

With the basic design understood, we are ready to develop the ADT. We define nine functions that provide the basic user interface. As with the BST, there are several internal functions needed to implement the tree. The ADT design is shown in Figure 8-12.

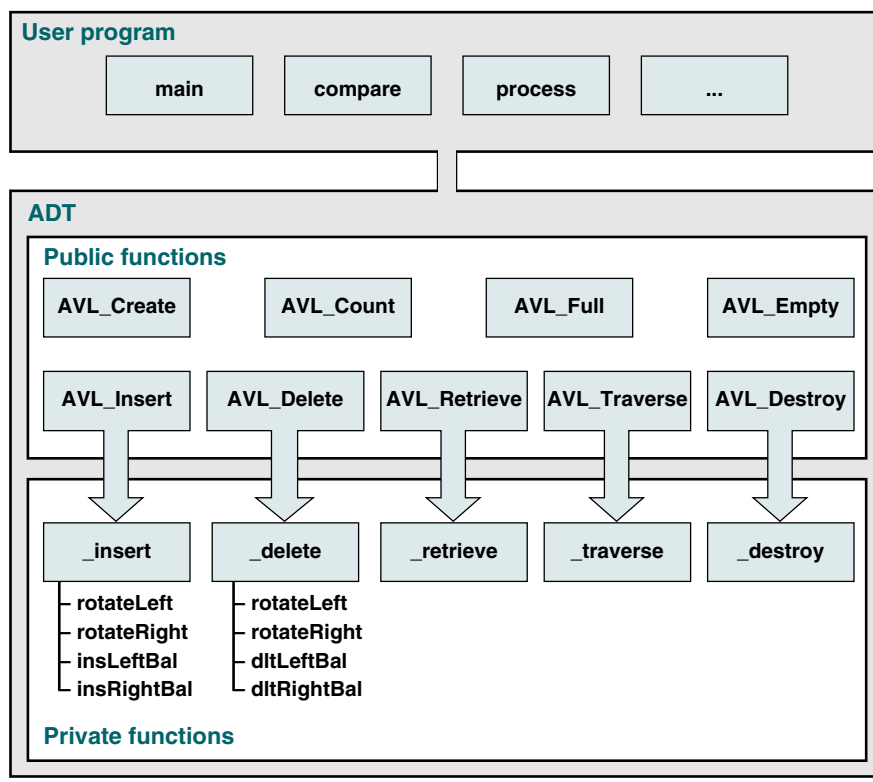


FIGURE 8-12 AVL Tree Design

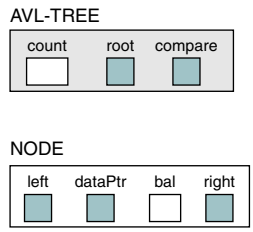
AVL Tree Data Structure

As with the other ADT structures, we implement the AVL tree ADT with a head structure and a node structure.

Head Structure

The **AVL tree head structure**, `AVL_TREE`, contains a count, a root pointer, and the address of the compare function needed to search the list. The application program's only view of the tree is a pointer to the head structure, which is allocated from dynamic memory when the tree is created.

The data structures are shown in Figure 8-13.




```
typedef struct
{
  int   count;
  NODE* root;
  int   (*compare) (void* argu1, void* argu2);
} AVL_TREE;

typedef struct node
{
  struct node* left;
  void*        dataPtr;
  int          bal;
  struct node* right;
} NODE;
```

FIGURE 8-13 AVL Tree Data Structure

Node Structure

The AVL tree node structure follows the same design as the binary search tree, with the addition of a balance factor. Each node must contain a *void* pointer to the data to be stored, which includes a key and attributes, a left and a right subtree pointer, and a balance factor.

AVL Tree Algorithms

A basic set of AVL tree algorithms is covered in this section. Depending on the application, other algorithms could be required. For example, some applications might need to include descending-key traversals. We begin with the ADT data structures and the prototype declarations in Program 8-1.

PROGRAM 8-1 AVL Tree Declarations

```
#define LH +1                                   // Left High
#define EH  0                                   // Even High
#define RH -1                                   // Right High

// Structure Declarations
typedef struct node
   {
    void*        dataPtr;
    struct node* left;
    int          bal;
    struct node* right;
   } NODE;

typedef struct
   {
    int    count;
    int   (*compare) (void* argu1, void* argu2);
    NODE*   root;
   } AVL_TREE;

// Prototype Declarations
   AVL_TREE* AVL_Create
            (int (*compare)(void* argu1, void* argu2));
   AVL_TREE* AVL_Destroy (AVL_TREE* tree);

   bool  AVL_Insert   (AVL_TREE* tree,
                       void* dataInPtr);
   bool  AVL_Delete   (AVL_TREE* tree,  void* dltKey);
   void* AVL_Retrieve (AVL_TREE* tree,  void* keyPtr);
   void  AVL_Traverse (AVL_TREE* tree,
                       void (*process)(void* dataPtr));
   int  AVL_Count     (AVL_TREE* tree);
   bool AVL_Empty     (AVL_TREE* tree);
   bool AVL_Full      (AVL_TREE* tree);

   static NODE* _insert
                  (AVL_TREE* tree, NODE* root,
                   NODE* newPtr,   bool* taller);
   static NODE* _delete
                  (AVL_TREE* tree,
                   NODE* root,      void* dltKey,
                   bool* shorter,   bool* success);
   static void* _retrieve
                  (AVL_TREE* tree,
                   void* keyPtr,    NODE* root);
   static void _traversal
                  (NODE* root,
                   void (*process)(void* dataPtr));
```

continued

PROGRAM 8-1 AVL Tree Declarations *(continued)*

```
49    static void _destroy     (NODE* root);
50
51    static NODE* rotateLeft  (NODE* root);
52    static NODE* rotateRight (NODE* root);
53    static NODE* insLeftBal  (NODE* root, bool* taller);
54    static NODE* insRightBal (NODE* root, bool* taller);
55    static NODE* dltLeftBal  (NODE* root, bool* shorter);
56    static NODE* dltRightBal (NODE* root, bool* shorter);
```

Create an AVL Tree

The abstract data type must be able to support multiple structures in one program. This is accomplished by allocating the tree head structure in dynamic memory. The create tree function allocates the structure, sets its count to zero and the root pointer to null, and stores the address of the compare function. It then returns the tree pointer. The code is shown in Program 8-2.

PROGRAM 8-2 Create AVL Tree Application Interface

```
 1  /* ================ AVL_Create ================
 2     Allocates dynamic memory for an AVL tree head
 3     node and returns its address to caller.
 4        Pre    compare is address of compare function
 5               used when two nodes need to be compared
 6        Post   head allocated or error returned
 7        Return head node pointer; null if overflow
 8  */
 9  AVL_TREE* AVL_Create
10          (int  (*compare) (void* argu1, void* argu2))
11  {
12  // Local Definitions
13     AVL_TREE* tree;
14
15  // Statements
16     tree = (AVL_TREE*) malloc (sizeof (AVL_TREE));
17     if (tree)
18        {
19         tree->root    = NULL;
20         tree->count   = 0;
21         tree->compare = compare;
22        } // if
23
24     return tree;
25  }  // AVL_Create
```

Insert an AVL Tree

The AVL tree insert function is the module called by the application program. It receives a pointer to the tree structure and a pointer to the data to be inserted into the tree. After allocating a node, it calls a recursive insert function to make the physical insertion. The insert function that the application called is shown in Program 8-3.

PROGRAM 8-3 Insert AVL Tree Application Interface

```
/* ================ AVL_Insert ==================
   This function inserts new data into the tree.
      Pre    tree is pointer to AVL tree structure
      Post   data inserted or memory overflow
      Return Success (true) or Overflow (false)
*/
bool AVL_Insert (AVL_TREE* tree, void* dataInPtr)
{
// Local Definitions
   NODE* newPtr;
   bool  forTaller;

// Statements
   newPtr = (NODE*)malloc(sizeof(NODE));
   if (!newPtr)
      return false;

   newPtr->bal     = EH;
   newPtr->right   = NULL;
   newPtr->left    = NULL;
   newPtr->dataPtr = dataInPtr;

   tree->root = _insert(tree,  tree->root,
                newPtr,      &forTaller);
   (tree->count)++;
   return true;
}  // AVL_Insert
```

Program 8-3 Analysis

Because all additions take place at a leaf node, we initialize the balance factor to even height and the subtree pointers to null. We then call the recursive insert function. When it returns, we update the tree count and return success.

Note the name of the recursive insert module. Following the guide for system software, we call it `_insert` so that the name will not be duplicated by application programmers.

Internal Insert Function

The internal insert function, called initially by `AVL_Insert`, requires four parameters: a pointer to the tree structure, a pointer to the root of the tree or the subtree, a pointer to the node being inserted, and a Boolean to tell if the tree has grown taller. The taller flag is used for rotation processing as we back

out of the tree after the insertion. If the tree is taller, the function calls one of two subfunctions: one to balance a left subtree or another to balance a right subtree. The changes to the balance factors are also handled as we back out of the tree. The function is shown in Program 8-4. Its design follows Algorithm 8-1.

PROGRAM 8-4 Internal Insert Function

```
/* ==================== _insert ====================
   This function uses recursion to insert the new data
   into a leaf node in the AVL tree.
   Pre   Application has called AVL_Insert, which passes
         root and data pointers.
   Post  Data have been inserted.
   Return pointer to [potentially] new root.
*/
NODE* _insert (AVL_TREE* tree,   NODE* root,
               NODE*     newPtr, bool* taller)
{
// Statements
   if (!root)
      {
       // Insert at root
       root     = newPtr;
       *taller  = true;
       return   root;
      } // if NULL tree

   if (tree->compare(newPtr->dataPtr,
                     root->dataPtr) < 0)
      {
       // newData < root -- go left
       root->left = _insert(tree,   root->left,
                            newPtr, taller);
       if (*taller)
          // Left subtree is taller
          switch (root->bal)
             {
              case LH:  // Was left high--rotate
                        root = insLeftBal (root, taller);
                        break;

              case EH:  // Was balanced--now LH
                        root->bal = LH;
                        break;

              case RH:  // Was right high--now EH
                        root->bal = EH;
```

continued

PROGRAM 8-4 Internal Insert Function *(continued)*

```
41                          *taller   = false;
42                          break;
43                  } // switch
44                return root;
45             } // new < node
46          else
47              // new data >= root data
48              {
49               root->right = _insert (tree,   root->right,
50                                      newPtr, taller);
51               if (*taller)
52                  // Right subtree is taller
53                  switch (root->bal)
54                      {
55                       case LH:  // Was left high--now EH
56                                 root->bal = EH;
57                                 *taller   = false;
58                                 break;
59
60                       case EH:  // Was balanced--now RH
61                                 root->bal = RH;
62                                 break;
63
64                       case RH:  // Was right high--rotate
65                                 root = insRightBal
66                                           (root, taller);
67                                 break;
68                      } // switch
69               return root;
70               } // else new data >= root data
71          return root;
72      }  // _insert
```

Program 8-4 Analysis

This nontrivial algorithm must be carefully studied to fully understand its logic. It begins with a recursive search to locate the correct insertion point in a leaf node. After completing the insertion (statement 16) and setting the taller flag, we begin to back out of the tree, examining the balance as we go. If we are out of balance after inserting on the left (statement 25), we call left balance. If we are out of balance after inserting on the right (statement 49), we call right balance.

The adjustment of the balance factors takes place in three functions. The current root's balance factor is adjusted in the insert function. The subtree balance factors are adjusted in left and right balance as the tree is rotated. All follow the general rules discussed in the "Adjusting the Balance Factors" section of this chapter.

After each node has been adjusted as necessary, we return the root. This is necessary because the root of a tree may change as we add data. This is one of the major differences between the binary search tree and the AVL tree. In the binary search tree, the first node inserted is always the root unless it is deleted.

Because this is a recursive function, it must have a base case. Can you see it? The base case occurs when we locate a leaf and return the root in statement 18.

Internal Left Balance Function

If the tree is taller after an insertion on the left subtree, we need to check to see if a rotation is necessary. As we saw above, it is necessary only if the tree was already left high. In this case we call the left balance function to restore the balance by rotating the left subtree. This algorithm is shown in Program 8-5. Its design is shown in Algorithm 8-2.

PROGRAM 8-5 Internal Insert Left Balance Function

```
/* ================= insLeftBalance =================
   Tree out-of-balance to the left. This function
   rotates the tree to the right.
      Pre  The tree is left high
      Post Balance restored; return potentially new root
*/
   NODE* insLeftBal (NODE* root, bool* taller)
{
// Local Definitions
   NODE* rightTree;
   NODE* leftTree;

// Statements
   leftTree = root->left;
   switch (leftTree->bal)
      {
       case LH: // Left High--Rotate Right
                root->bal     = EH;
                leftTree->bal = EH;

                // Rotate Right
                root     = rotateRight (root);
                *taller  = false;
                break;
       case EH: // This is an error
                printf ("\n\aError in insLeftBal\n");
                exit (100);
       case RH: // Right High-Requires double
                // rotation: first left, then right
                rightTree = leftTree->right;
                switch (rightTree->bal)
                   {
                    case LH: root->bal     = RH;
                             leftTree->bal = EH;
                             break;
                    case EH: root->bal     = EH;
                             leftTree->bal = LH;
```

continued

PROGRAM 8-5 Internal Insert Left Balance Function *(continued)*

```
38                                  break;
39                          case RH: root->bal     = EH;
40                                   leftTree->bal = LH;
41                                   break;
42                          } // switch rightTree
43
44                       rightTree->bal = EH;
45                       // Rotate Left
46                       root->left = rotateLeft (leftTree);
47
48                       // Rotate Right
49                       root     = rotateRight (root);
50                       *taller  = false;
51             } // switch
52          return root;
53       }  // leftBalance
```

Program 8-5 Analysis The function examines the left subtree's balance factor to determine what action must be taken. If the subtree is left high—that is, if it is left of left—a simple right rotation is all that is required. If it is right high, it is right of left and a double rotation is required. The third possible balance factor, even high, is not possible. Our action if it should occur, therefore, is to abort the program because it must contain a logic error to arrive at this impossible situation.

Internal Rotation Functions

The rotation of a subtree requires an exchange of pointers among four subtree pointers. The logic is a relatively simple extension of the logic to exchange two values. The code to rotate left and right is shown in Program 8-6. Its design is shown in Algorithm 8-3. As you study the logic, you may find it helpful to refer to Figure 8-10.

PROGRAM 8-6 Internal Insert Rotate Left and Right Function

```
 1  /* ================== rotateLeft ===================
 2     Exchanges pointers to rotate the tree to the left.
 3        Pre  root points to tree to be rotated
 4        Post Node rotated and new root returned
 5  */
 6  NODE* rotateLeft (NODE* root)
 7  {
 8  // Local Definitions
 9     NODE* tempPtr;
10
11  // Statements
12     tempPtr        = root->right;
13     root->right    = tempPtr->left;
```

continued

PROGRAM 8-6 Internal Insert Rotate Left and Right Function *(continued)*

```
14    tempPtr->left  = root;
15
16    return tempPtr;
17 }  // rotateLeft
18
19 /* ================== rotateRight ==================
20    Exchange pointers to rotate the tree to the right.
21       Pre  root points to tree to be rotated
22       Post Node rotated and new root returned
23 */
24 NODE* rotateRight (NODE* root)
25 {
26 // Local Definitions
27    NODE* tempPtr;
28
29 // Statements
30    tempPtr         = root->left;
31    root->left      = tempPtr->right;
32    tempPtr->right  = root;
33
34    return tempPtr;
35 }  // rotateRight
```

Delete an AVL Tree

Deleting from an AVL tree encounters many of the same problems we saw when we inserted data. Interestingly, however, only the rotation algorithms can be reused; the balancing algorithms must be rewritten for deletion.

We use the same design for deletion that we used for insertion. The application program interface sees only the AVL tree head structure. It passes a *void* pointer to the delete key so that the search function can be called to locate the delete node in the tree. If the deletion is successful, the AVL tree delete function returns true; if the node cannot be found, it returns false.

The code for the application interface is shown in Program 8-7.

PROGRAM 8-7 Delete AVL Tree Application Interface

```
1  /* ==================== AVL_Delete ====================
2     This function deletes a node from the tree and
3     rebalances it if necessary.
4        Pre    tree initialized--null tree is OK
5               dltKey is pointer to key to be deleted
6        Post   node deleted and its space recycled
7               -or- An error code is returned
8        Return Success (true) or Not found (false)
9  */
```

continued

PROGRAM 8-7 Delete AVL Tree Application Interface *(continued)*

```
10  bool AVL_Delete (AVL_TREE* tree, void* dltKey)
11  {
12  // Local Definitions
13     bool  shorter;
14     bool  success;
15     NODE* newRoot;
16
17  // Statements
18  newRoot = _delete (tree,    tree->root, dltKey,
19                     &shorter, &success);
20  if (success)
21     {
22      tree->root = newRoot;
23      (tree->count)--;
24      return true;
25     } // if
26  else
27     return false;
28  }  // AVL_Delete
```

Program 8-7 Analysis The application interface delete function simply accepts the parameters from the user and reformats them for the recursive delete function. When the deletion has been completed, the function updates the tree count and passes the status back to the calling function.

Internal Delete Function

The real work is done by the recursive AVL tree delete function shown in Program 8-8. Its design is shown in Algorithm 8-4.

PROGRAM 8-8 Internal Delete Function

```
 1  /* ==================== _delete ====================
 2     Deletes node from the tree and rebalances
 3     tree if necessary.
 4        Pre    tree initialized--null tree is OK.
 5               dltKey contains key of node to be deleted
 6               shorter indicates tree is shorter
 7        Post   node is deleted and its space recycled
 8               -or- if key not found, tree is unchanged
 9        Return true if deleted; false if not found
10               pointer to root
11  */
12
13  NODE* _delete (AVL_TREE* tree,   NODE* root,
14                 void*     dltKey, bool* shorter,
15                 bool*     success)
16  {
```

continued

PROGRAM 8-8 Internal Delete Function *(continued)*

```
17  // Local Definitions
18     NODE* dltPtr;
19     NODE* exchPtr;
20     NODE* newRoot;
21
22  // Statements
23     if (!root)
24        {
25         *shorter = false;
26         *success = false;
27         return NULL;
28        } // if
29
30     if (tree->compare(dltKey, root->dataPtr) < 0)
31         {
32          root->left = _delete (tree,
33                                root->left, dltKey,
34                                shorter,    success);
35          if (*shorter)
36              root   = dltRightBal (root, shorter);
37         } // if less
38     else if (tree->compare(dltKey, root->dataPtr) > 0)
39         {
40          root->right = _delete (tree,
41                                 root->right, dltKey,
42                                 shorter,     success);
43          if (*shorter)
44              root = dltLeftBal (root, shorter);
45         } // if greater
46     else
47         // Found equal node
48         {
49          dltPtr = root;
50          if (!root->right)
51              // Only left subtree
52              {
53               newRoot  = root->left;
54               *success = true;
55               *shorter = true;
56               free (dltPtr);
57               return newRoot;              // base case
58              } // if true
59          else
60              if (!root->left)
61                  // Only right subtree
62                  {
63                   newRoot  = root->right;
```

continued

PROGRAM 8-8 Internal Delete Function *(continued)*

```
64                    *success = true;
65                    *shorter = true;
66                    free (dltPtr);
67                    return newRoot;          // base case
68                 } // if
69              else
70                 // Delete Node has two subtrees
71                 {
72                  exchPtr = root->left;
73                  while (exchPtr->right)
74                      exchPtr = exchPtr->right;
75                  root->dataPtr = exchPtr->dataPtr;
76                  root->left = _delete (tree,
77                        root->left, exchPtr->dataPtr,
78                        shorter,    success);
79                  if (*shorter)
80                     root = dltRightBal (root, shorter);
81                 } // else
82           } // equal node
83      return root;
84   }  // _delete
```

Program 8-8 Analysis

Again we see a long, relatively complex implementation. Although this function is definitely longer than the structured programming guideline of one page, it does not readily lend itself to decomposition.

As we saw with the recursive insert, the first part of the function searches for the node to be deleted. If it reaches a leaf before finding the node to be deleted, it sets the success flag false and returns a null pointer. This is the first of three base cases.

Once the node to be deleted is found, we determine if it is a leaf or a leaflike node. Remember that delete operations can take place only at a leaf. If a node has two subtrees, we must search for a leaf node to take its place. We first check to see if there is a right subtree. If there is none, we simply save the left subtree pointer as the pointer to take the root's place. If there is a right subtree, we check the left subtree pointer. If there is none, we save the right subtree pointer as the pointer to take the root's place. Assuming for the moment that there is zero or one subtree, we set success and shorter true, free the deleted node's memory, and return the new root pointer. These are the second (statement 57) and the third (statement 67) base cases.

If the node to be deleted has two subtrees, we must find a node to take its place. Our design searches the left subtree for its largest node. When we find it, we move its data to replace the deleted data and then recursively call the delete function to delete what we know to be a valid leaf node. This logic is shown in statements 72 through 80.

As we back out of the tree, we must check to make sure that it is still balanced. If we have deleted a node on the left and the tree is shorter, we call delete right balance (statements 36 and 80). If we have deleted a node on the right and the tree is shorter, we call delete left balance (statement 44). Of course, if the tree is not shorter, we don't need to balance it.

Internal Delete Right Balance Function

Delete right balance is used when a deletion has occurred on a left subtree and the tree needs to be rebalanced. We need to rebalance only if the root is right high. In this case we have deleted a node on the left side of a right high tree, making it out of balance on the right subtree by two levels. Conversely, if the root is left or even high, no rotation is needed. The code is shown in Program 8-9. Its design is shown in Algorithm 8-5.

PROGRAM 8-9 Internal Delete Right Balance Function

```
/* ================== dltRightBal ==================
   The tree is shorter after a delete on the left. This
   function adjusts the balance factors and rotates
   the tree to the left if necessary.
      Pre     tree shorter
      Post    Balance factors reset-balance restored
      Returns potentially new root
*/
NODE* dltRightBal (NODE* root, bool* shorter)
{
// Local Definitions
   NODE* rightTree;
   NODE* leftTree;

// Statements
   switch (root->bal)
       {
        case LH:       // Deleted Left--Now balanced
             root->bal  = EH;
             break;

        case EH:       // Now Right high
             root->bal  = RH;
             *shorter   = false;
             break;

        case RH:       // Right High - Rotate Left
             rightTree = root->right;
             if (rightTree->bal == LH)
                 // Double rotation required
                 {
                  leftTree  = rightTree->left;

                  switch (leftTree->bal)
                      {
                       case LH: rightTree->bal = RH;
                                root->bal      = EH;
                                break;
```

continued

PROGRAM 8-9 Internal Delete Right Balance Function *(continued)*

```
39                          case EH: root->bal      = EH;
40                                   rightTree->bal = EH;
41                                   break;
42                          case RH: root->bal      = LH;
43                                   rightTree->bal = EH;
44                                   break;
45                              } // switch
46
47                      leftTree->bal = EH;
48
49                      // Rotate Right then Left
50                      root->right =
51                             rotateRight (rightTree);
52                      root = rotateLeft  (root);
53                     } // if rightTree->bal == LH
54                 else
55                     {
56                      // Single Rotation Only
57                      switch (rightTree->bal)
58                          {
59                           case LH:
60                           case RH: root->bal      = EH;
61                                    rightTree->bal = EH;
62                                    break;
63                           case EH: root->bal      = RH;
64                                    rightTree->bal = LH;
65                                    *shorter       = false;
66                                    break;
67                          } // switch rightTree->bal
68                      root = rotateLeft (root);
69                     } // else
70             } // switch
71       return root;
72   }  // dltRightBal
```

Program 8-9 Analysis As we said above, we need to rotate only if the right tree's height is two larger than the left tree's height. Therefore, if the tree was left high or even high before the delete, no rotation is necessary. If it was right high, it needs to be rotated. If its right subtree is left high, a double rotation is necessary; otherwise, only a single rotation is necessary. In either case we first adjust the balance factors and then rotate. As you study the logic, you might find it helpful to refer back to Figure 8-11.

Retrieve an AVL Tree

The retrieve function follows the left-right structure of the tree until the desired node is found. When it is located, the address of the data is returned to the calling function. If the data are not located, a null pointer is returned.

The design is similar to the insert and delete algorithms described above. An application interface function provides a pointer to the tree and a pointer

to the key. We then call an ADT recursive function to locate the data. The code for the ADT retrieve function is shown in Program 8-10. The code for the recursive function is shown in Program 8-11.

PROGRAM 8-10 Retrieve AVL Tree Application Interface

```
/* ================== AVL_Retrieve =================
   Retrieve node searches tree for node containing
   the requested key and returns pointer to its data.
      Pre     Tree has been created (may be null)
              data is pointer to data structure
                   containing key to be located
      Post    Tree searched and data pointer returned
      Return  Address of matching node returned
              If not found, NULL returned
*/
void* AVL_Retrieve  (AVL_TREE* tree, void* keyPtr)
{
// Statements
   if (tree->root)
       return _retrieve (tree, keyPtr, tree->root);
   else
       return NULL;
}  // AVL_Retrieve
```

Program 8-10 Analysis

The retrieve data function is quite simple. It is needed only because the recursive function needs an additional parameter, of which the using application is not aware. It simply passes the pointer to the tree structure and the key, adding the pointer to the tree root.

PROGRAM 8-11 Internal Retrieve Function

```
/* ================== _retrieve ==================
   Searches tree for node containing requested key
   and returns its data to the calling function.
      Pre     AVL_Retrieve passes tree, keyPtr, root
              keyPtr is pointer to data structure
                 containing key to be located
      Post    tree searched; data pointer returned
      Return  Address of matching node returned
                  if not found, NULL returned
*/
void*  _retrieve (AVL_TREE* tree,
                  void*     keyPtr, NODE* root)
{
// Statements
   if (root)
       {
```

continued

PROGRAM 8-11 Internal Retrieve Function *(continued)*

```
17          if (tree->compare(keyPtr, root->dataPtr) < 0)
18              return _retrieve(tree, keyPtr, root->left);
19          else if (tree->compare(keyPtr, root->dataPtr) > 0)
20              return _retrieve(tree, keyPtr, root->right);
21          else
22              // Found equal key
23              return root->dataPtr;
24          }  // if root
25      else
26          // Data not in tree
27          return NULL;
28  }  // _retrieve
```

Program 8-11 Analysis The retrieve function uses the compare function stored in the tree structure when the tree was created. If the search argument, **keyPtr**, is less than the root data's key, it calls itself with the left subtree as the root. If the search argument is greater than the root data's key, it calls itself with the right subtree as the root. If the argument is not greater or less than the root, it must be equal, so it returns the root's data pointer.

Carefully study the recursive function calls in statements 18 and 20. Note that they return the address returned by the recursive call. In this fashion the address returned by the base cases is passed back up the tree structure until it is finally returned to the application by the interface function.

Traverse an AVL Tree

The traversal uses an inorder traversal of the tree, calling the application-dependent process function when the node is to be processed. While the traversal is standard, the application-processing function is not. Therefore, whenever the using application calls the traversal, it must also pass the address of the function that processes the data. The processing function uses only one parameter: the address of the node to be processed. The application interface for the traverse function is shown in Program 8-12.

PROGRAM 8-12 Traverse AVL Tree Application Interface

```
1   /* ================== AVL_Traverse ==================
       Process tree using inorder traversal.
2         Pre   Tree has been created (may be null)
3               process "visits" nodes during traversal
4         Post  Nodes processed in LNR (inorder) sequence
5   */
6   void AVL_Traverse (AVL_TREE* tree,
7                      void (*process) (void* dataPtr))
8   {
9
```

continued

PROGRAM 8-12 Traverse AVL Tree Application Interface *(continued)*

```
// Statements
   _traversal (tree->root, process);
   return;
}  // end AVL_Traverse
```

Internal Traverse Function

The traverse function in turn calls the ADT internal function that actually traverses the tree. The code is shown in Program 8-13.

PROGRAM 8-13 Internal Traverse Function

```
/* ================== _traversal ==================
   Inorder tree traversal. To process a node, we use
   the function passed when traversal was called.
      Pre   Tree has been created (may be null)
      Post  All nodes processed
*/
void _traversal (NODE* root,
                 void (*process) (void* dataPtr))
{
// Statements
if  (root)
    {
     _traversal  (root->left, process);
     process     (root->dataPtr);
     _traversal  (root->right, process);
    } // if
return;
}  // _traversal
```

Empty an AVL Tree

The empty function simply checks the tree count. If it is zero, it returns true; otherwise, it returns false. The code is shown in Program 8-14.

PROGRAM 8-14 Empty AVL Tree Application Interface

```
/* ================== AVL_Empty ==================
   Returns true if tree is empty; false if any data.
      Pre      Tree has been created. May be null
      Returns  True if tree empty, false if any data
*/
bool AVL_Empty (AVL_TREE* tree)
{
// Statements
   return (tree->count == 0);
}  // AVL_Empty
```

Full AVL Tree

Because there is no way to test for available memory in C, we must actually try to allocate a node. If we are successful, we delete it and return false—the list is not full. If the allocation fails, we return true—there is not enough memory for another node. The code is shown in Program 8-15.

PROGRAM 8-15 Full AVL Tree Application Interface

```
/* =============== AVL_Full ==============
   If there is no room for another node, returns true.
   Pre      Tree has been created
   Returns  True if no room for another insert,
            false if room.
*/
bool AVL_Full (AVL_TREE* tree)
{
// Local Definitions
   NODE* newPtr;

// Statements
   newPtr = (NODE*)malloc(sizeof (*(tree->root)));
   if (newPtr)
      {
       free (newPtr);
       return false;
      } // if
   else
        return true;
}  // AVL_Full
```

AVL Tree Count

The count function, shown in Program 8-16, simply returns the number of nodes currently in the tree.

PROGRAM 8-16 AVL Tree Count Application Interface

```
/* =============== AVL_Count ==============
   Returns number of nodes in tree.
      Pre     Tree has been created
      Returns tree count
*/
int AVL_Count (AVL_TREE* tree)
{
// Statements
   return (tree->count);
}  // AVL_Count
```

Destroy AVL Tree

The last function in the AVL tree abstract data type is the destroy AVL tree function. It is used to physically delete and free all of the data nodes and the tree head structure when the tree is no longer needed. Because we need to traverse the tree to find all of the data and nodes that need to be deleted, we call a recursive function to do the physical deletions.

The logic for the destroy function parallels the destroy functions we have seen previously. The code is shown in Program 8-17.

PROGRAM 8-17 Destroy AVL Tree Application Interface

```
/* =============== AVL_Destroy ==============
   Deletes all data in tree and recycles memory.
   The nodes are deleted by calling a recursive
   function to traverse the tree in inorder sequence.
      Pre      tree is a pointer to a valid tree
      Post     All data and head structure deleted
      Return   null head pointer
*/
AVL_TREE* AVL_Destroy (AVL_TREE* tree)
{
// Statements
   if (tree)
      _destroy (tree->root);

   // All nodes deleted. Free structure
   free (tree);
   return NULL;
}  // AVL_Destroy
```

Program 8-17 Analysis

The logic is simple. We first make sure that we have a valid tree by testing the tree pointer. If it is valid—that is, if it is not null—we call the recursive subfunction that does the physical deletions. When we return we then delete the tree structure itself and return a null pointer.

Internal Destroy Function

The code for the recursive deletion function is shown in Program 8-18.

PROGRAM 8-18 Internal Destroy Function

```
/* ================= _destroy ================
   Deletes all data in tree and recycles memory.
   The nodes are deleted by calling a recursive
   function to traverse the tree in inorder sequence.
      Pre      root is pointer to valid tree/subtree
      Post     All data and head structure deleted
      Return   null head pointer
```

continued

PROGRAM 8-18 Internal Destroy Function *(continued)*

```
 8  */
 9  void _destroy (NODE* root)
10  {
11  // Statements
12     if (root)
13        {
14         _destroy (root->left);
15         free (root->dataPtr);
16         _destroy (root->right);
17         free (root);
18        } // if
19     return;
20  }  // _destroy
```

Program 8-18 Analysis The logic in the recursive deletion function is a little more complex. The big question is: When do we delete the data, and when do we delete the node?. We have to make sure that we do each only once and at the right time.

The data are deleted as we return from the left subtree. This is the logical point at which we need to process an AVL tree's data. This is done at statement 15. However, we are not yet ready to delete the node because we have not processed the right subtree. We must wait until we return from the right subtree traversal to delete it (see statement 17).

8.4 Application—Count Words

In this section we create an AVL tree application that uses a tree structure containing all of the words in a document, with a count of the number of times each word is used. This program uses the AVL tree abstract data type found in Section 8.3, "AVL Tree Abstract Data Type."

Data Structure

The application data structure is shown in Figure 8-14. Each entry in the AVL tree contains a word from the document and a pointer to an integer that contains a count of the number of times the word appears in the document. Had the AVL tree contained a function to write a node back to the tree—that is, to update it—we could have simply stored the count in the node.

Program Design

As we process the file, we parse a word and then search the tree to see if we can find it. If it's already in the tree, we simply add to its counter. If it's not in the tree, we insert a new entry into the tree. Figure 8-15 shows the program design. We have indicated the AVL tree ADT functions where they are used. The code for each of the functions is shown in the following sections.

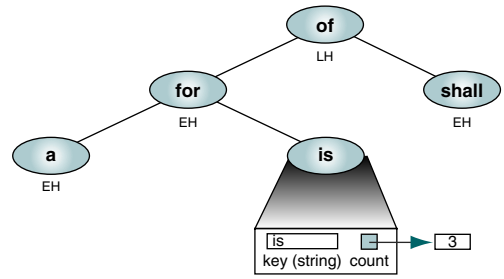


FIGURE 8-14 Count Words Data Structure

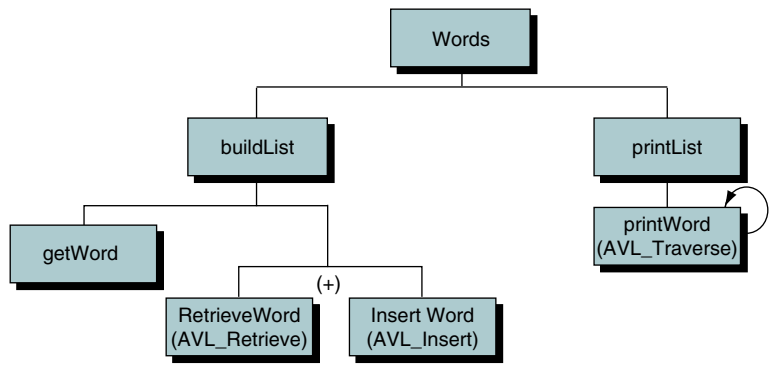


FIGURE 8-15 Count Words Design

Count Words Program

The program's data structure, its prototype declarations, and its mainline are shown in Program 8-19. As you study it, pay particular attention to the create list function call (statement 35). Because we are using the AVL tree ADT, we must write a compare function that compares two instances of the data stored in the tree. This compare function is then passed to the AVL tree when we create it.

PROGRAM 8-19 Count Words

```
1  /* This program counts the words in a file.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <string.h>
7  #include <ctype.h>
8  #include <stdlib.h>
```

continued

PROGRAM 8-19 Count Words *(continued)*

```
 9  #include <stdbool.h>
10  #include "P8AVLADT.h"
11
12  // Structures
13  typedef struct
14     {
15      char word[51];                      // One word
16      int  count;
17     } DATA;
18
19  // Prototype Declarations
20  void  buildList    (AVL_TREE*      wordList);
21  void  insertWord   (AVL_TREE*      words);
22  void  deleteWord   (AVL_TREE*      words);
23  void  printList    (AVL_TREE*      wordList);
24  void  printWord    (void* aWord);
25  bool  getWord      (DATA* aWord,   FILE* fpWords);
26  int   compareWords (void* arguPtr, void* listPtr);
27
28  int main (void)
29  {
30  // Local Definitions
31     AVL_TREE* wordList;
32
33  // Statements
34     printf("Start count words in document\n");
35     wordList = AVL_Create (compareWords);
36
37     buildList (wordList);
38     printList (wordList);
39
40     printf("End count words\n");
41     return 0;
42  }  // main
```

Program 8-19 Analysis A good design keeps the code in the mainline function to a minimum. In this program we start with a hello message; call **`AVL_Create`**, **`buildList`**, and **`printList`**; and then conclude with a good-bye message. Like a good manager, *main* delegates all of the processing to subordinate functions.

Build List

The build list function reads one word from the file, looks it up in the tree, and, if the word is found, simply adds 1 to the word's counter. If the word is not yet in the tree, it creates a counter, sets it to 1, and adds the word to the tree. The code is shown in Program 8-20.

PROGRAM 8-20 Build List

```
/* ================== buildList ==================
   Reads file and creates AVL tree containing list
   of all words used in the file with count of the
   number of times each word is found in the file.
      Pre   wordList has been created
      Post  AVL tree (list) built or error returned
*/
void buildList (AVL_TREE* wordList)
{
// Local Definitions
   char  fileName[25];
   FILE* fpWords;

   bool  success;
   DATA* aWord;
   DATA  newWord;

// Statements
   printf("Enter name of file to be processed: ");
   scanf ("%24s", fileName);

   fpWords = fopen (fileName, "r");
   if (!fpWords)
      {
       printf("%-s could not be opened\a\n",fileName);
       printf("Please verify name and try again.\n");
       exit (100);
      } // !fpWords

   while (getWord (&newWord, fpWords))
      {
       aWord = AVL_Retrieve(wordList, &(newWord));
       if (aWord)
          (aWord->count)++;
       else
          {
           aWord = (DATA*) malloc (sizeof (DATA));
           if (!aWord)
              {
               printf("Error 120 in buildList\n");
               exit (120);
              } // if
           // Add word to list
           *aWord       = newWord;
           aWord->count = 1;
           success      = AVL_Insert (wordList, aWord);
           if (!success)
              {
```

continued

PROGRAM 8-20 Build List *(continued)*

```
49                printf("Error 121 in buildList\a\n");
50                exit (121);
51              } // if overflow test
52           } // else
53        } // while
54
55     printf("End AVL Tree\n");
56     return;
57  } // buildList
```

Program 8-20 Analysis

As you study this function, pay close attention to four important points. First, we allow the user to enter the name of the file. This program is designed to support any text file, and by reading the filename from the keyboard we make it as flexible as possible.

One of the most difficult loops to write in any program is a read file loop. We use a simple technique in this program: we call a read function that returns true if it was able to parse a word and false if there was no word to parse. This makes the control of the processing loop very simple.

Note the "intelligent" data names we use for words. The first, **aWord**, is used to retrieve a word from the list. The second, **newWord**, is used to create a new entry in the tree. Good names make a program more readable.

Finally, note that we check for memory failures. When we allocate memory for the new word's counter, we test for an overflow. Again, when we insert the new word into the tree, we test the return from the insert function. If either function fails, we print an error message and exit the program.

Get Word

The get word function parses the file input and extracts one word from the file. Its code is shown in Program 8-21.

PROGRAM 8-21 Get Word

```
 1  /* ================== getWord ==================
 2     Reads one word from file.
 3        Pre  nothing
 4        Post word read into reference parameter
 5  */
 6  bool getWord (DATA* aWord, FILE* fpWords)
 7  {
 8  // Local Definitions
 9     char strIn[51];
10     int  ioResult;
11     int  lastChar;
12
13  // Statements
14     ioResult = fscanf(fpWords, "%50s", strIn);
15     if (ioResult != 1)
16         return false;
```

continued

PROGRAM 8-21 Get Word *(continued)*

```
17      // Copy and remove punctuation at end of word.
18      strcpy (aWord->word, strIn);
19      lastChar = strlen(aWord->word) - 1;
20      if (ispunct(aWord->word[lastChar]))
21         aWord->word[lastChar] = '\0';
22      return true;
23   }  // getWord
```

Program 8-21 Analysis

Parsing words simply means copying all of the characters between two or more spaces to a string. Because we are processing a text document, however, we had to remove any punctuation from the end of the word. We did this by simply testing the last character to make sure it was alphabetic. If it wasn't, we deleted it from the word by moving a null character over it.

Depending on the application, it might be necessary to use a different set of editing criteria. For example, to parse the words in a program, we need to keep any numbers and the underscore character as a part of the word. We might also want to eliminate the reserved words. To keep the rest of the program as simple as possible, these changes would be programmed in `getWord`.

Compare Words

As we pointed out in our description of Program 8-19, the AVL tree doesn't know how to compare the data. The application programmer must therefore supply a compare function and pass it to the tree when it is created. Program 8-22 contains the compare function for our count words application. Note that the compare is made on the word, which is a string and therefore requires the string compare function.

PROGRAM 8-22 Compare Words Function

```
 1  /* ================== compareWords ================
 2     This function compares two integers identified
 3     by pointers to integers.
 4        Pre    arguPtr and listPtr are pointers to DATA
 5        Return -1: arguPtr value <  listPtr value
 6               -0: arguPtr value == listPtr value
 7               +1: arguPtr value >  listPtr value
 8  */
 9  int compareWords (void* arguPtr, void* listPtr)
10  {
11  // Local Declarations
12     DATA arguValue;
13     DATA listValue;
14
15  // Statements
16     arguValue = *(DATA*)arguPtr;
17     listValue = *(DATA*)listPtr;
18
```

continued

PROGRAM 8-22 Compare Words Function *(continued)*

```
19      return (strcmp(arguValue.word, listValue.word));
20   }  // compare
```

Print Words

Because the implementation of the tree is hidden in the ADT, we cannot directly traverse the tree to print the list. Rather, we must call on the tree traversal function, `AVL_Traverse`. Now we have another problem: the traverse function doesn't know how to process the data as it traverses the tree. For this reason we pass the process function to the traverse function when we call it. Program 8-23 contains both the call to the traverse function and the function that prints one word. The results of counting the words in the Gettysburg Address are shown at the end of the program. In the interest of space, we show only the beginning and the end of the word list.

Figure 8-16 traces the first two insertions, "Four" and "score," into a null AVL tree. Use it to help follow the program.

PROGRAM 8-23 Print Words List

```
 1  /* ================== printList ==================
 2     Prints the list with the count for each word.
 3        Pre  list has been built
 4        Post list printed
 5  */
 6  void printList (AVL_TREE* wordList)
 7  {
 8  // Statements
 9     printf("\nWords found in list\n");
10     AVL_Traverse (wordList, printWord);
11     printf("\nEnd of word list\n");
12     return;
13  }  // printList
14
15  /* ================== printWord ==================
16     Prints one word from the list with its count.
17        Pre  ADT calls function to print data
18        Post data printed
19  */
20  void printWord (void* aWord)
21  {
22  // Statements
23     printf("%-25s %3d\n",
24           ((DATA*)aWord)->word, ((DATA*)aWord)->count);
25     return;
26  }  // printWord
```

```
Results:
Start count words in document
```

continued

PROGRAM 8-23 Print Words List *(continued)*

```
Enter name of file to be processed: gtsybrg.txt
End AVL Tree

Words found in list
But                          1
Four                         1
God                          1
It                           3
...
years                        1

End of word list
End count words
```

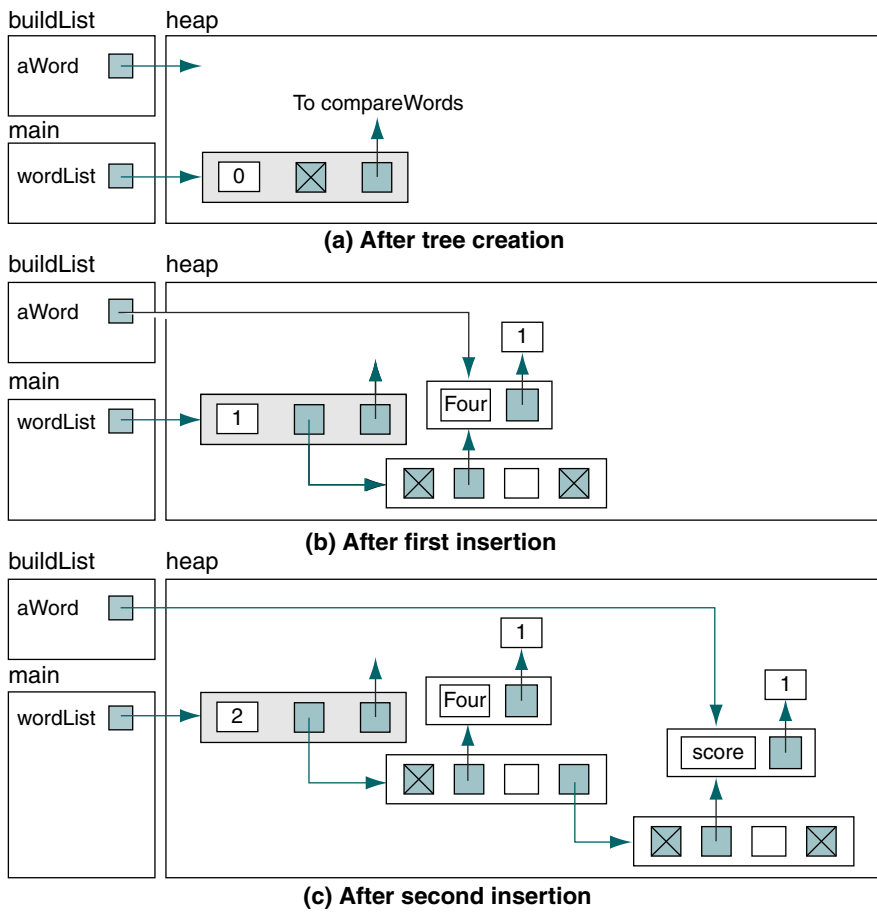


FIGURE 8-16 Insertions into AVL Tree

8.5 Key Terms

AVL tree
AVL tree head structure
AVL tree node structure
height-balanced trees
left of left
left of right
right of left
right of right

8.6 Summary

- An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1, which means that the tree is balanced.
- We consider four different cases when we want to rebalance a tree after deletion or insertion: left of left, right of right, right of left, and left of right.
- We must balance a left of left, unbalanced AVL tree by rotating the out-of-balance node to the right.
- We must balance a right of right, unbalanced tree by rotating the out-of-balance node to the left.
- We must balance a right of left, unbalanced tree by double rotation: first the left subtree to the left and then the out-of-balance node to the right.
- We must balance a left of right, unbalanced tree by double rotation: first the right subtree to the right and then the out-of-balance node to the left.
- Inserting a new node into an AVL tree is the same as inserting a new node into a binary search tree, except that when we back out of the tree we must constantly check the balance of each node and rebalance if necessary.
- Deleting a node from an AVL tree is the same as deleting a node from a binary search tree, except that when we back out of the tree we must constantly check the balance of each node and rebalance if necessary.

8.7 Practice Sets

Exercises

1. Balance the AVL tree in Figure 8-17. Show the balance factors in the result.
2. Balance the AVL tree in Figure 8-18. Show the balance factors in the result.
3. Add 49 to the AVL tree in Figure 8-19. The result must be an AVL tree. Show the balance factors in the resulting tree.
4. Add 68 to the AVL tree in Figure 8-19. The result must be an AVL tree. Show the balance factors in the resulting tree.

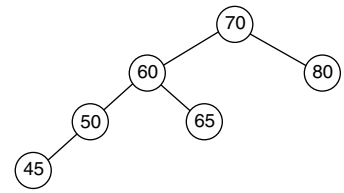


FIGURE 8-17 Figure for Exercise 1

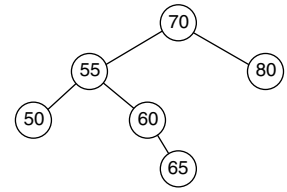


FIGURE 8-18 Figure for Exercise 2

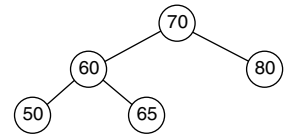


FIGURE 8-19 Figure for Exercises 3 and 4

5. Insert 44, 66, and 77 into the AVL tree in Figure 8-20. The result must be an AVL tree. Show the balance factors in the resulting tree.

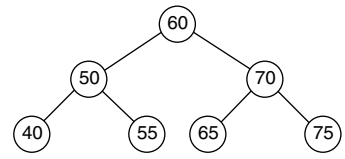


FIGURE 8-20 Figure for Exercise 5

6. Create an AVL tree using the following data entered as a sequential set. Show the balance factors in the resulting tree:

```
14 23 7 10 33 56 80 66 70
```

7. Create an AVL tree using the following data entered as a sequential set. Show the balance factors in the resulting tree:

```
7 10 14 23 33 56 66 70 80
```

8. Create an AVL tree using the following data entered as a sequential set. Show the balance factors in the resulting tree:

```
80 70 66 56 33 23 14 10 7
```

9. Insert 44 and 50 into the tree created in Exercise 6.
10. Insert 44 and 50 into the tree created in Exercise 7.
11. Insert 44 and 50 into the tree created in Exercise 8.
12. Delete the node containing 80 from the AVL tree in Figure 8-21.

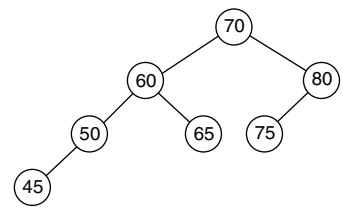


FIGURE 8-21 Figure for Exercises 12 and 13

13. Delete the node containing 70 from the AVL tree in Figure 8-21.

Problems

14. Write an iterative version of Algorithm 8-1, "AVL Tree Insert."
15. Write an iterative version of Algorithm 8-4, "AVL Tree Delete."

Projects

16. Write the C code for Problem 14.
17. Write the C code for Problem 15.
18. Write a program that reads a list of names and telephone numbers from a text file and inserts them into an AVL tree. Once the tree has been built, present the user with a menu that allows him or her to search the list for a specified name, insert a new name, delete an existing name, or print the entire phone list. At the end of the job, write the data in the list back to the file. Test your program with at least 10 names.

19. When writing AVL tree algorithms, we need to be able to print the tree in a hierarchical order to verify that the algorithms are processing the data correctly. Write a print function that can be called to print the tree. The printed output should contain the node level number in parentheses, its data, and its balance factor. So that it can handle any size tree, print the data with the root on the left of the page and the descendents on the right. When viewed sideways, the left subtrees should be on the bottom of the page and the right subtrees on the top of the page. For example, Figure 8-21 would be printed as shown in Figure 8-22.

```
                (2) 80 RH
                       (3) 75 EH
          (1) 70 LH
                       (3) 65 EH
                (2) 60 LH
                       (3) 50 LH
                            (4) 45 EH
```

FIGURE 8-22 Output for Project 19

20. Modify the search for the AVL tree ADT to locate the immediate predecessor for a node. Once located, it tests the next node and returns found or not found and a pointer to the node (null if not found).
21. Create the ADT for an AVL tree using the array implementation. In an array implementation, the pointers become indexes to the subtree elements. When you create the tree, you need to know the maximum number of nodes to be stored in the tree. To test the ADT, use it to run the program in Project 18.
22. Build an index for the words in a document. Each word is to be identified by its page number. The program is to keep track of page numbers by counting pages identified by a form-feed character at the end of each page. (*Hint:* See Section 8-4, "Application—Count Words.") Print the index at the end of the program.
23. An index generally contains only keywords. Write a program that uses a file of keywords to index a document. It begins by reading the keyword file and inserting the words into an AVL tree. It then reads the document and builds an AVL tree index of the keywords in the document. Print the keyword index at the end of the document.

This page intentionally left blank

Chapter 9
Heaps

A third type of tree is a heap. A heap is a binary tree whose left and right subtrees have values less than their parents. The root of a heap is guaranteed to hold the largest node in the tree; its subtrees contain data that have lesser values. Unlike the binary search tree, however, the lesser-valued nodes of a heap can be placed on either the right or the left subtree. Therefore, both the left and the right branches of the tree have the same properties.

Heaps have another interesting facet: they are often implemented in an array rather than a linked list. When we implement a heap in an array, we are able to calculate the location of the left and right subtrees. Conversely, given the address of a node, we can calculate the address of its parent. This makes for very efficient processing.

9.1 Basic Concepts

In this section we define a heap and the two operations necessary for the operation of the heap.

Definition

A **heap**, as shown in Figure 9-1, is a binary tree structure with the following properties:

1. The tree is complete or nearly complete.
2. The key value of each node is greater than or equal to the key value in each of its descendents.

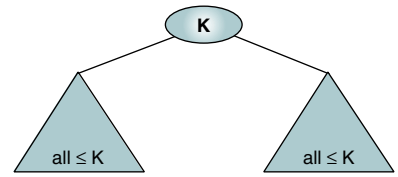
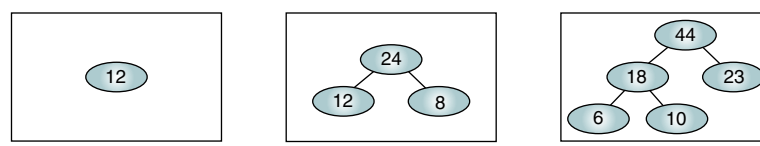


FIGURE 9-1 Heap

Sometimes this structure is called a **max-heap**. The second property of a heap—the key value is greater than the keys of the subtrees—can be reversed to create a **min-heap**. That is, we can create a minimum heap in which the key value in a node is *less than* the key values in all of its subtrees. Generally speaking, whenever the term *heap* is used by itself, it refers to a max-heap.

A heap is a complete or nearly complete binary tree in which the key value in a node is greater than or equal to the key values in all of its subtrees, and the subtrees are in turn heaps.

To better understand the structure of a heap, let's examine the heaps in Figure 9-2. Study the two- and three-level heaps. Note that the left node of two siblings can be either larger or smaller than the right node. Compare this ordering with the binary search tree that we studied earlier. It is obviously different.[1] Finally, in the three-level heap, note that the third level is being filled from the left. This is the definition of a nearly complete tree and is a requirement for a heap.



(a) Root-only heap (b) Two-level heap (c) Three-level heap

FIGURE 9-2 Heap Trees

To complete our understanding of a heap, let's look at some structures that are not heaps. Figure 9-3 shows four examples of structures that are not heaps. The first two structures are not heaps because they are not complete or nearly complete trees. Although the third and fourth examples are nearly complete, the keys of the nodes are not always greater than the keys of their descendents.

1. Another interesting difference is that in a complete binary search tree with no equal nodes, the root always contains the median value of the keys, whereas in the heap it contains the largest value.

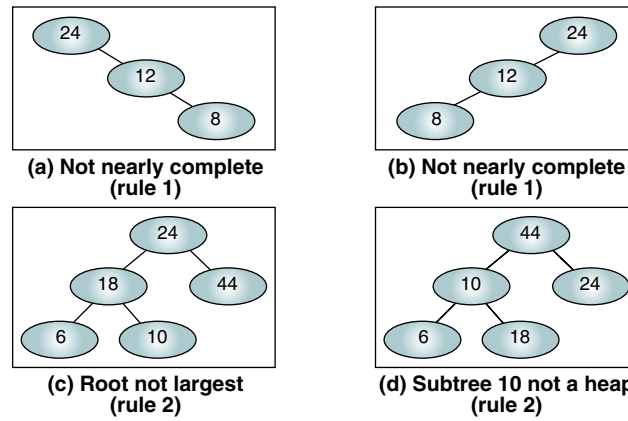


FIGURE 9-3 Invalid Heaps

Maintenance Operations

Two basic maintenance operations are performed on a heap: insert a node and delete a node. Although the heap structure is a tree, it is meaningless to traverse it, search it, or print it out. In these respects it is much like a restricted data structure. To implement the insert and delete operations, we need two basic algorithms: reheap up and reheap down. All of the other heap algorithms build on these two. We therefore study them first.

Reheap Up

Imagine that we have a nearly complete binary tree with N elements whose first $N - 1$ elements satisfy the order property of heaps, but the last element does not. In other words, the structure would be a heap if the last element were not there. The **reheap up** operation repairs the structure so that it is a heap by floating the last element up the tree until that element is in its correct location in the tree. We show this restructuring graphically in Figure 9-4. Before we reheap up, the last node in the heap was out of order. After the reheap, it is in its correct location and the heap has been extended one node.

Like the binary search tree, inserts into heaps take place at a leaf. Furthermore, because the heap is a complete or nearly complete tree, the node must be placed in the last leaf level at the first empty position. This creates the situation we see in Figure 9-4. If the new node's key is larger than that of its parent, it is floated up the tree by exchanging the child and parent keys and data. The data eventually rise to the correct place in the heap by repeatedly exchanging child/parent keys and data.

The reheap up operation reorders a "broken" heap by floating the last element up the tree until it is in its correct location in the heap.

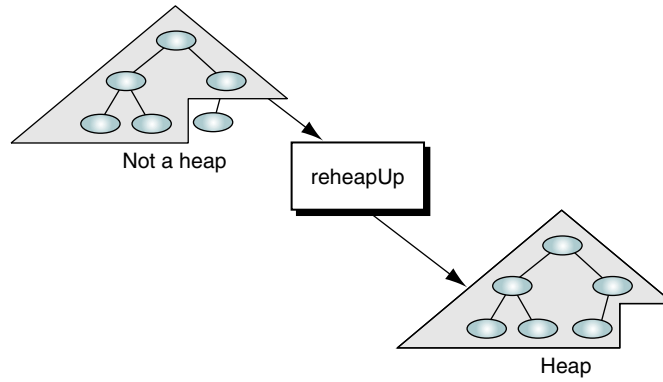


FIGURE 9-4 Reheap Up Operation

Figure 9-5 traces the reheap up operation in a heap. At the beginning we observe that 25 is greater than its parent's key, 12. Because 25 is greater than 12, we also know from the definition of a heap that it is greater than the parent's left subtree keys. We therefore exchange 25 and 12 and call reheap up to test its current position in the heap. Once again, 25 is greater than its parent's key, 21. Therefore, we again exchange the nodes' data. This time, when reheap up is called, the value of the current node's key is less than the value of its parent key, indicating that we have located the correct position and the operation stops.

Reheap Down

Now let's examine the reverse situation. Imagine we have a nearly complete binary tree that satisfies the heap order property except in the root position. This situation occurs when the root is deleted from the tree, leaving two disjointed heaps. To correct the situation, we move the data in the last tree node to the root. Obviously, this action destroys the tree's heap properties.

> Reheap down reorders a "broken" heap by pushing the root down the tree until it is in its correct position in the heap.

To restore the heap property, we need an operation that sinks the root down until it is in a position where the heap-ordering property is satisfied. We call this operation **reheap down**. The reheap down operation is shown in Figure 9-6.

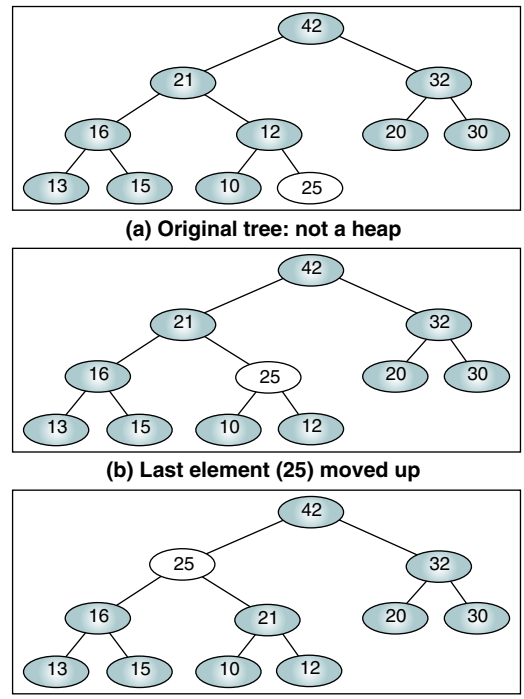


FIGURE 9-5 Reheap Up Example

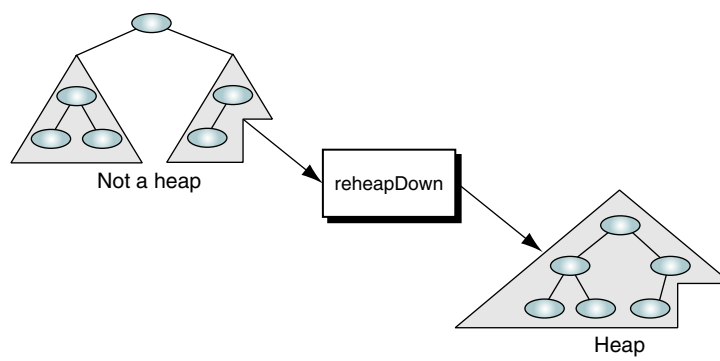
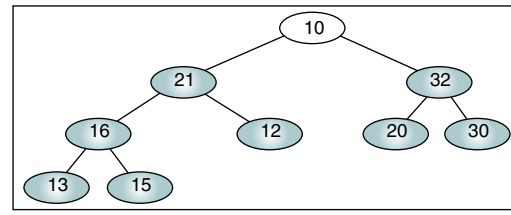


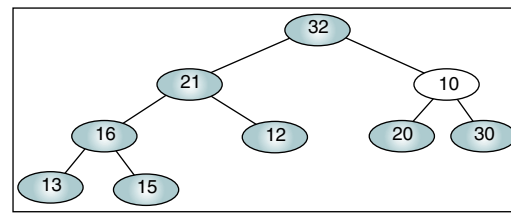
FIGURE 9-6 Reheap Down Operation

Figure 9-7 is an example of the reheap down operation. When we start, the root (10) is smaller than its subtrees. We examine them and select the larger of the two to exchange with the root, in this case 32. Having made the

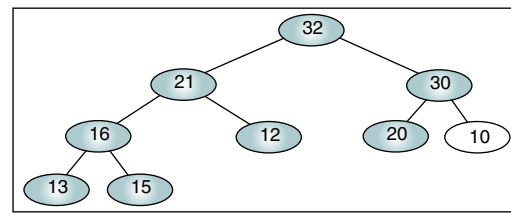
exchange in Figure 9-7(b), we check the subtrees to see if we are finished and see that 10 is smaller than their keys. Once again we exchange 10 with the larger of the subtrees, 30. At this point we have reached a leaf and are finished.



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

FIGURE 9-7 Reheap Down Example

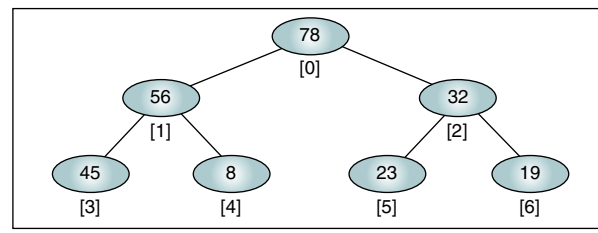
9.2 Heap Implementation

Although a heap can be built in a dynamic tree structure, it is most often implemented in an array. This implementation is possible because the heap is, by definition, complete or nearly complete. Therefore, the relationship between a node and its children is fixed and can be calculated as shown below.

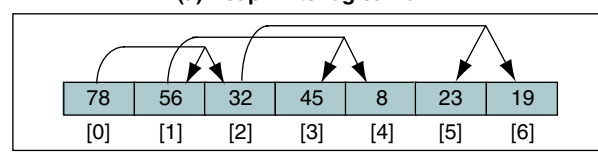
1. For a node located at index i, its children are found at:
   a. Left child: $2i + 1$
   b. Right child: $2i + 2$
2. The parent of a node located at index i is located at $\lfloor (i - 1) / 2 \rfloor$.
3. Given the index for a left child, j, its right sibling, if any, is found at $j + 1$. Conversely, given the index for a right child, k, its left sibling, which must exist, is found at $k - 1$.

4. Given the size, n, of a complete heap, the location of the first leaf is $\lfloor (n / 2) \rfloor$. Given the location of the first leaf element, the location of the last nonleaf element is one less.

To demonstrate these relationships, let's examine Figure 9-8.



(a) Heap in its logical form



(b) Heap in an array

FIGURE 9-8 Heaps in Arrays

In Figure 9-8 we have the following relationships:

1. The index of 32 is 2, so the index of its left child, 23, is 2 / 2 + 1, or 5. The index of its right child, 19, is 2 / 2 + 2, or 6 (Relationship 1).
2. The index of 8 is 4, so the index of its parent, 56, is $\lfloor (4 - 1) / 2 \rfloor$, or 1 (Relationship 2).
3. In the first example, we found the address of the left and the right children. To find the right child, we could also have used the location of the left child (5) and added 1 (Relationship 3).
4. The total number of elements is 7, so the index of the first leaf element, 45, is $\lfloor (7 / 2) \rfloor$, or 3 (Relationship 4).
5. The location of the last nonleaf element, 32, is 3 – 1, or 2 (Relationship 5).

Finally, these heap relationships are unique to C and other languages that use base-zero index addressing for their arrays. They would need to be modified slightly for other languages that use base-one index addressing.

A heap can be implemented in an array because it must be a complete or nearly complete binary tree, which allows a fixed relationship between each node and its children.

Figure 9-8 shows a heap in its tree form and in its array form. Study both of these representations carefully. In the logical form, note that each node's

data are greater than the data in its descendents. In the array format, follow the arrows to both successors for a node and confirm that the array properly represents the logical (tree) format. In Figure 9-8 the tree is complete and the array is full. Therefore, to add another node to the tree we would have to add a new level to the tree and double the size of the array because the physical array should always represent the complete tree structure.

Algorithms

There are two ways to build a heap. We can start with an empty array and insert elements into the array one at a time, or, given an array of data that are not a heap, we can rearrange the elements in the array to form a heap. After looking at the two basic algorithms, reheap up and reheap down, we examine both approaches. Then we look at the logic for deleting data from a heap.

Reheap Up

Reheap up uses recursion to move the new node up the tree. It begins by determining the parent's address using the relationships we discussed earlier in this section. If the key of the new data is greater than the key of its parent, it exchanges the nodes and recursively calls itself (Figure 9-5). The base case is determined when either there is no parent, meaning we are at the heap's root, or the nodes are in the proper heap sequence. The logic is shown in Algorithm 9-1.

ALGORITHM 9-1 Reheap Up

```
Algorithm reheapUp (heap, newNode)
Reestablishes heap by moving data in child up to its
correct location in the heap array.
   Pre  heap is array containing an invalid heap
        ¬newNode is index location to new data in heap
   Post heap has been reordered
 1 if (newNode not the root)
   1  set parent to parent of newNode
   2  if (newNode key > parent key)
      1  exchange newNode and parent)
      2  reheapUp (heap, parent)
   3  end if
 2 end if
end reheapUp
```

Reheap Down

The logic for reheap down is a little more complex. As we push nodes down the heap, we need to determine whether the current entry is less than either of its children (one or both). If it is, we need to exchange it with the larger entry (see Figure 9-7).

To determine whether the current entry is less than either of its children, we first determine which of the two subtree keys is larger. Once we know

which one is larger, we compare the current node to it. If the current node is smaller than the larger node, we exchange them and recursively call reheap down. As you study Algorithm 9-2, you will see that most of the pseudocode is used to determine which of the subtree keys is larger.

ALGORITHM 9-2 Reheap Down

```
Algorithm reheapDown (heap, root, last)
Reestablishes heap by moving data in root down to its
correct location in the heap.
   Pre    heap is an array of data
          root is root of heap or subheap
          last is an index to the last element in heap
   Post   heap has been restored
   Determine which child has larger key
1  if (there is a left subtree)
   1  set leftKey to left subtree key
   2  if (there is a right subtree)
      1  set rightKey to right subtree key
   3  else
      1  set rightKey to null key
   4  end if
   5  if (leftKey > rightKey)
      1  set largeSubtree to left subtree
   6  else
      1  set largeSubtree to right subtree
   7  end if
      Test if root > larger subtree
   8  if (root key < largeSubtree key)
      1  exchange root and largeSubtree
      2  reheapDown (heap, largeSubtree, last)
   9  end if
2  end if
end reheapDown
```

Algorithm 9-2 Analysis

In statement 8 we need to test only one subtree because we know from logic that if the root is greater than the left subtree and the left subtree is greater than the right subtree, the root is also greater than the right subtree. This argument is logically stated as "If A > B and if B > C, then A > C."

There are two base cases in this algorithm. First, if there are no leaves, we are finished. This case is shown in statement 1 (if there is not a left subtree, there cannot be a right subtree because a heap is a complete tree). The second base case is shown in statement 1.8. If the root is greater than the larger subtree, we have a valid heap.

Build a Heap

Given a filled array of elements in random order, to build the heap we need to rearrange the data so that each node in the heap is greater than its children. We begin by dividing the array into two parts, the left being a heap and the right being data to be inserted into the heap. At the beginning the root

(the first node) is the only node in the heap and the rest of the array are data to be inserted. This structure is shown in Figure 9-9, which is a worst-case example—an ordered list. Note the "wall" between the first and second nodes. The lines at the top of the figures point to the nodes' children. Each iteration of the insertion algorithm uses reheap up to insert the next element into the heap and moves the wall separating the elements one position to the right.

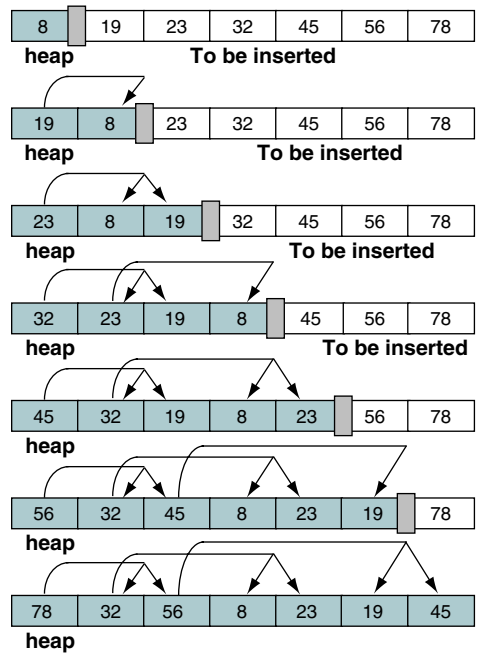


FIGURE 9-9 Building a Heap

To insert a node into the heap, we follow the parent path up the heap, swapping nodes that are out of order. If the nodes are in order, the insertion terminates and the next node is selected and inserted into the heap. This process is sometimes referred to as **heapify**.

The build heap algorithm is very simple. We walk through the array that we need to convert to a heap, starting at the second element, calling reheap up for each array element to be inserted into the heap. The pseudocode is shown in Algorithm 9-3.

Insert a Node into a Heap

Once we have built the heap, we can insert a node so long as there is room in the array. To insert a node, we need to locate the first empty leaf in the array. We find it immediately after the last node in the tree, which is given as a parameter. To insert a node, we move the new data to the first empty leaf and reheap up. The concept is shown in Figure 9-10.

ALGORITHM 9-3 Build Heap

```
Algorithm buildHeap (heap, size)
Given an array, rearrange data so that they form a heap.
   Pre    heap is array containing data in nonheap order
          size is number of elements in array
   Post   array is now a heap
 1 set walker to 1
 2 loop (walker < size)
   1   reheapUp(heap, walker)
   2   increment walker
 3 end loop
end buildHeap
```

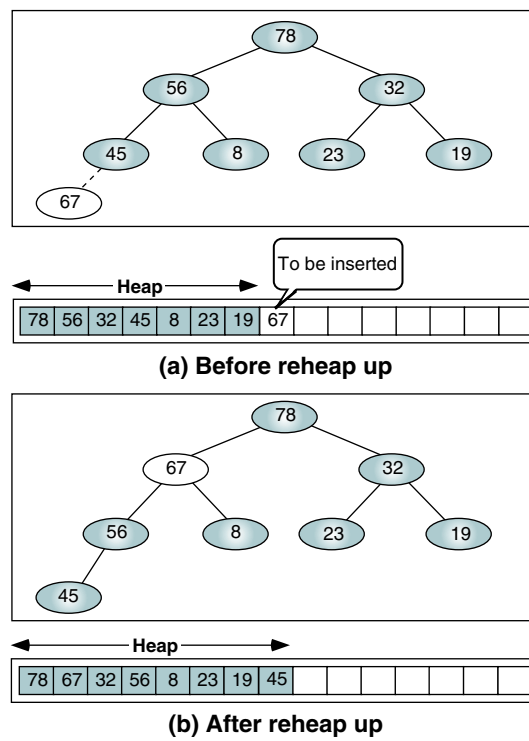


FIGURE 9-10 Insert Node

The algorithm for insert heap is straightforward. It moves the data to be inserted into the heap to the first leaf and calls reheap up. The pseudocode is shown in Algorithm 9-4.

ALGORITHM 9-4 Insert Heap

```
Algorithm insertHeap (heap, last, data)
Inserts data into heap.
   Pre    heap is a valid heap structure
          last is reference parameter to last node in heap
          data contains data to be inserted
   Post   data have been inserted into heap
   Return true if successful; false if array full
 1 if (heap full)
   1  return false
 2 end if
 3 increment last
 4 move data to last node
 5 reheapUp (heap, last)
 6 return true
end insertHeap
```

Delete a Node from a Heap

When deleting a node from a heap, the most common and meaningful logic is to delete the root. In fact, the rationale for a heap is to determine and extract the largest element, the root. After it has been deleted, the heap is thus left without a root. To reestablish the heap, we move the data in the last heap node to the root and reheap down. The concept is shown in Figure 9-11.

The logic to delete the root from a heap is shown in Algorithm 9-5. Note that we return the data at the top of the heap to the calling algorithm for processing.

ALGORITHM 9-5 Delete Heap Node

```
Algorithm deleteHeap (heap, last, dataOut)
Deletes root of heap and passes data back to caller.
   Pre    heap is a valid heap structure
          last is reference parameter to last node in heap
          dataOut is reference parameter for output area
   Post   root deleted and heap rebuilt
          root data placed in dataOut
   Return true if successful; false if array empty
 1 if (heap empty)
   1  return false
 2 end if
 3 set dataOut to root data
 4 move last data to root
 5 decrement last
 6 reheapDown (heap, 0, last)
 7 return true
end deleteHeap
```

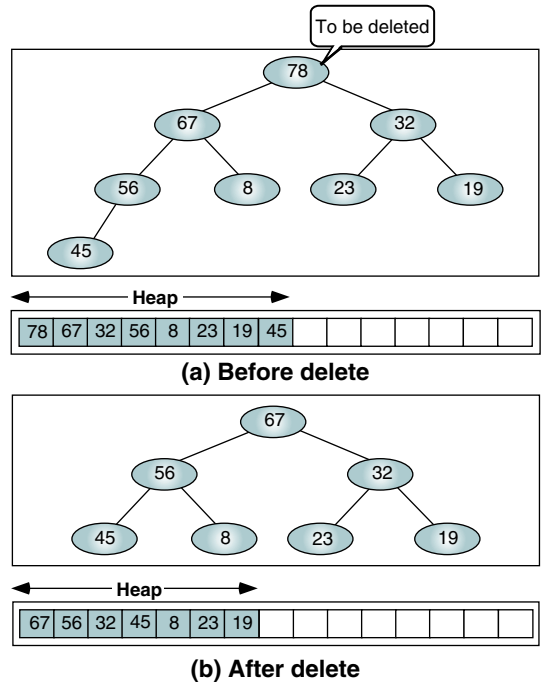


FIGURE 9-11 `deleteHeap` Node

9.3 Heap ADT

No programming language has intrinsic heap operators; therefore, to use a heap, we must write the heap operations. In this section we create an abstract data type for a heap.

We define nine functions that provide the basic user interface. As with the BST and AVL trees, there are several internal functions needed to implement the tree. The ADT design is shown in Figure 9-12.

Heap Structure

Before developing the ADT, we need to determine the data structure we will use. As we mentioned above, we use an array (see Figure 9-8). Therefore, all that we need is the array itself and three pieces of metadata: the index location of the last element in the heap, the maximum size of the heap, and the number of elements in the heap. The data structure design is shown in Figure 9-13.

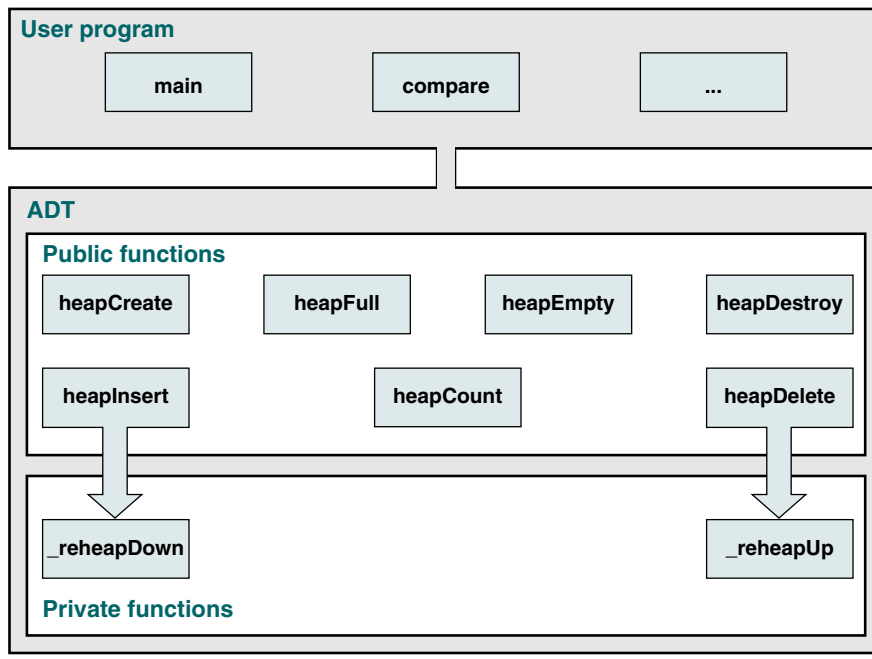


FIGURE 9-12 Heap ADT Design

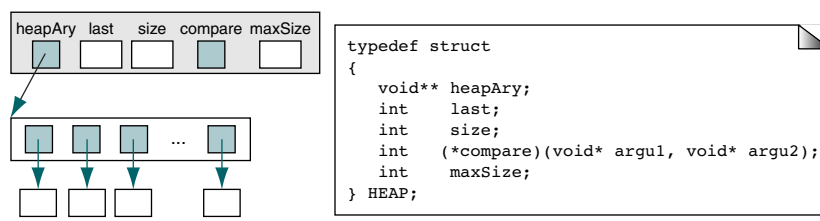


FIGURE 9-13 Heap ADT Structure

Heap Algorithms

Program 9-1 contains the data structures for the heap functions.

PROGRAM 9-1 Heap Declaration

```
1  /* Data Structures for heap ADT
2        Created by:
3        Date:
4  */
```

continued

PROGRAM 9-1 Heap Declaration *(continued)*

```
 5  #include <stdbool.h>
 6
 7  typedef struct
 8  {
 9     void** heapAry;
10     int    last;
11     int    size;
12     int    (*compare) (void* argu1, void* argu2);
13     int    maxSize;
14  } HEAP;
15
16  // Prototype Definitions
17  HEAP* heapCreate (int maxSize,
18              int (*compare) (void* arg1, void* arg2));
19  bool  heapInsert  (HEAP* heap, void*  dataPtr);
20  bool  heapDelete  (HEAP* heap, void** dataOutPtr);
21  int   heapCount   (HEAP* heap);
22  bool  heapFull    (HEAP* heap);
23  bool  heapEmpty   (HEAP* heap);
24  void  heapDestroy (HEAP* heap);
25
26  static void _reheapUp   (HEAP* heap, int childLoc);
27  static void _reheapDown (HEAP* heap, int root);
```

Create a Heap

The create heap function receives a count for the maximum size of the heap and returns the address of the heap structure. The code is shown in Program 9-2.

PROGRAM 9-2 Create Heap Application Interface

```
 1  /* ==================== heapCreate ====================
 2     Allocates memory for heap and returns address of
 3     heap head structure.
 4        Pre  Nothing
 5        Post heap created and address returned
 6                  if memory overflow, NULL returned
 7  */
 8  #include <math.h>
 9
10  HEAP* heapCreate (int maxSize,
11                int  (*compare) (void* argu1, void* argu2))
12  {
13  // Local Definitions
14     HEAP* heap;
15
```

continued

PROGRAM 9-2 Create Heap Application Interface *(continued)*

```
16  // Statements
17     heap = (HEAP*)malloc(sizeof (HEAP));
18     if (!heap)
19        return NULL;
20
21     heap->last    = -1;
22     heap->compare = compare;
23
24     // Force heap size to power of 2 -1
25     heap->maxSize =
26              (int) pow (2, ceil(log2(maxSize))) - 1;
27     heap->heapAry = (void*)
28                     calloc(heap->maxSize, sizeof(void*));
29     return heap;
30  }  // createHeap
```

Insert a Heap

The insert heap function implements Algorithm 9-4. It inserts one entry into the heap and reheaps up to reestablish the heap. Its code is seen in Program 9-3.

PROGRAM 9-3 Insert Heap Application Interface

```
 1  /* ================== heapInsert ====================
 2     Inserts data into heap.
 3        Pre    Heap is a valid heap structure
 4               last is pointer to index for last element
 5               data is data to be inserted
 6        Post   data have been inserted into heap
 7        Return true if successful; false if array full
 8  */
 9  bool heapInsert (HEAP* heap, void* dataPtr)
10  {
11  // Statements
12     if (heap->size == 0)                  // Heap empty
13        {
14         heap->size                = 1;
15         heap->last                = 0;
16         heap->heapAry[heap->last] = dataPtr;
17         return true;
18        } // if
19     if (heap->last == heap->maxSize - 1)
20       return false;
21     ++(heap->last);
22     ++(heap->size);
23     heap->heapAry[heap->last] = dataPtr;
24     _reheapUp (heap, heap->last);
```

continued

PROGRAM 9-3 Insert Heap Application Interface *(continued)*

```
   return true;
} // heapInsert
```

Internal Reheap Up Function

Program 9-4 parallels Algorithm 9-1.

PROGRAM 9-4 Internal Reheap Up Function

```
/* ==================== reheapUp ====================
   Reestablishes heap by moving data in child up to
   correct location heap array.
      Pre  heap is array containing an invalid heap
           newNode is index to new data in heap
      Post newNode inserted into heap
*/
void  _reheapUp  (HEAP* heap, int childLoc)
{
// Local Definitions
   int    parent;
   void** heapAry;
   void*  hold;

// Statements
   // if not at root of heap -- index 0
   if (childLoc)
      {
       heapAry = heap->heapAry;
       parent = (childLoc - 1)/ 2;
       if (heap->compare(heapAry[childLoc],
                         heapAry[parent]) > 0)
           // child is greater than parent -- swap
           {
            hold             = heapAry[parent];
            heapAry[parent]  = heapAry[childLoc];
            heapAry[childLoc] = hold;
           _reheapUp (heap, parent);
           } // if heap[]
      } // if newNode
   return;
} // reheapUp
```

Delete a Heap

The delete heap function implements Algorithm 9-5. It deletes the element at the top of the heap and returns it to the caller. It then calls reheap down to reestablish the heap. The code is shown in Program 9-5.

PROGRAM 9-5 Delete Heap Application Interface

```
/*  ================== heapDelete ===================
   Deletes root of heap and passes data back to caller.
      Pre    heap is a valid heap structure
             last is reference to last node in heap
             dataOut is reference to output area
      Post   last deleted and heap rebuilt
             deleted data passed back to user
      Return true if successful; false if array empty
*/
bool heapDelete (HEAP* heap, void** dataOutPtr)
{
// Statements
   if (heap->size == 0)
       // heap empty
       return false;
   *dataOutPtr = heap->heapAry[0];
   heap->heapAry[0]  = heap->heapAry[heap->last];
   (heap->last)--;
   (heap->size)--;
   _reheapDown (heap, 0);
   return true;
}  // heapDelete
```

Internal Reheap Down Function

Although reheap down is a rather long algorithm, it follows the pseudocode closely. The major difference is that we chose to code the swap inline rather than call a swap function. The logic, which parallels Algorithm 9-2, is shown in Program 9-6.

PROGRAM 9-6 Internal Reheap Down Function

```
/* ================== reheapDown ===================
   Reestablishes heap by moving data in root down to its
   correct location in the heap.
      Pre  heap is array of data
           root is root of heap or subheap
           last is an index to last element in heap
      Post heap has been restored
*/
void _reheapDown (HEAP* heap, int root)
{
// Local Definitions
   void* hold;
   void* leftData;
   void* rightData;
   int   largeLoc;
```

continued

PROGRAM 9-6 Internal Reheap Down Function *(continued)*

```
16      int   last;
17
18  // Statements
19      last = heap->last;
20      if ((root * 2 + 1) <= last)          // left subtree
21          // There is at least one child
22         {
23          leftData   = heap->heapAry[root * 2 + 1];
24          if ((root * 2 + 2) <= last)  // right subtree
25             rightData = heap->heapAry[root * 2 + 2];
26          else
27             rightData = NULL;
28
29          // Determine which child is larger
30          if ((!rightData)
31               || heap->compare (leftData, rightData) > 0)
32             {
33              largeLoc = root * 2 + 1;
34             } // if no right key or leftKey greater
35          else
36             {
37              largeLoc = root * 2 + 2;
38             } // else
39          // Test if root > larger subtree
40          if (heap->compare (heap->heapAry[root],
41              heap->heapAry[largeLoc]) < 0)
42              {
43               // parent < children
44               hold = heap->heapAry[root];
45               heap->heapAry[root] =
46                     heap->heapAry[largeLoc];
47               heap->heapAry[largeLoc] = hold;
48               _reheapDown (heap, largeLoc);
49              } // if root <
50      } // if root
51      return;
52  }  // reheapDown
```

9.4 Heap Applications

Three common applications of heaps are selection algorithms, priority queues, and sorting. We discuss heap sorting in Chapter 12 and selection algorithms and priority queues here.

Selection Algorithms

There are two solutions to the problem of determining the k^{th} element in an unsorted list. We could first sort the list and select the element at location k, or we could create a heap and delete $k - 1$ elements from it, leaving the desired element at the root. Selecting the k^{th} element is then easy; it is at the root of the heap. Because we are studying heaps, let's look at the second solution, using a heap.

Creating the heap is simple enough. We have already discussed the heap creation algorithms. Once we have a heap, however, how do we select the desired element? Rather than simply discard the elements at the top of the heap, a better solution is to place the deleted element at the end of the heap and reduce the heap size by 1. After the k^{th} element has been processed, the temporarily removed elements can then be reinserted into the heap.

For example, if we want to know the fourth-largest element in a list, we can create the heap shown in Figure 9-14. After deleting three times, we have the fourth-largest element, 21, at the top of the heap. After selecting 21 we reheap to restore the heap so that it is complete and we are ready for another selection.

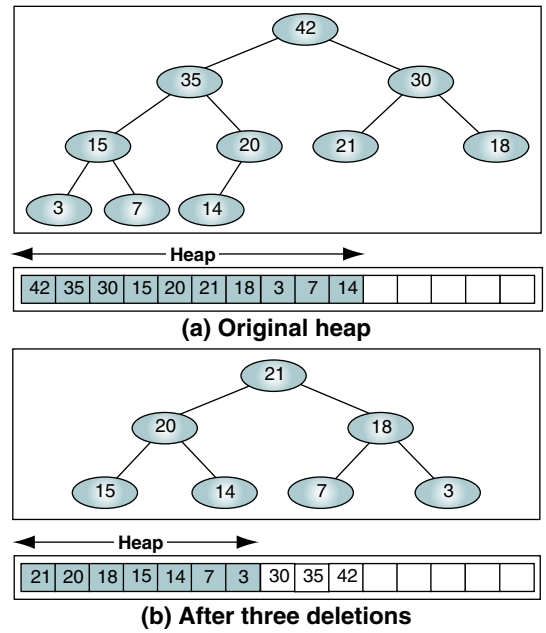


FIGURE 9-14 Heap Selection

The heap selection logic is shown in Algorithm 9-6.

ALGORITHM 9-6 Heap Selection

```
Algorithm selectK (heap, k, heapLast)
Select the k-th largest element from a list
   Pre    heap is an array implementation of a heap
          k is the ordinal of the element desired
          heapLast is reference parameter to last element
   Post   k-th largest value returned
 1 if (k > heap size)
   1  return false
 2 end if
 3 set origHeapSize to heapLast + 1
 4 loop (k times)
   1  set tempData to root data
   2  deleteHeap (heap, heapLast, dataOut)
   3  move tempData to heapLast + 1
 5 end loop
   Desired element is now at top of heap
 6 move root data to holdOut
   Reconstruct heap
 7 loop (while heapLast < origHeapSize)
   1  increment heapLast
   2  reheapUp (heap, heapLast)
 8 end loop
 9 return holdOut
end selectK
```

Priority Queues

The queue structure that we studied in Chapter 4 uses a linear list in which each node travels serially through the queue. It is not possible for any element to advance faster than the others. Although this system may be very equitable, it is often not very realistic. Oftentimes, for various reasons, we want to prioritize one element over the others. As a rather trivial example, consider the first-class line at the airport. Most people wait in one long line but if you have first-class tickets or belong to the airline's executive club, the line is very short if not empty.

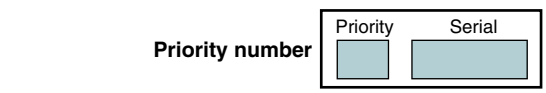
Design

The heap is an excellent structure to use for a **priority queue**. As an event enters the queue, it is assigned a priority number that determines its position relative to the other events already in the queue. It is assigned a priority number even though the new event can enter the heap in only one place at any given time, the first empty leaf. Once in line, however, the new event quickly rises to its correct position relative to all other events in the heap. If it has the highest priority, it rises to the top of the heap and becomes the next event to be processed. If it has a low priority, it remains relatively low in the heap, waiting its turn.

The key in a priority queue must be carefully constructed to ensure that the queue works properly. One common technique is to use an encoded priority

number that consists of the priority plus a sequential number representing the event's place within the queue. For example, given a queue with five priority classes, we can construct a key in which the first digit of the priority number represented the queue priority, 1 through 5, and the rest of the number represented the serial placement within the priority. Because we are using a priority heap, however, the serial number must be in descending order—that is, 999 down to 0 within each priority.

If we assume that there will be a maximum of 1000 events for any priority at any one time, we could assign the lowest priority to the sequential numbers in the range 1999 down to 1000, the second-lowest priority to the sequential numbers in the range 2999 to 2000, the third-lowest priority to the numbers in the range 3999 to 3000, and so forth. This concept is shown in Figure 9-15.



Priority	Serial	Priority	Serial	Priority	Serial
1	999	3	999	5	999
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
1	000	3	000	5	000
2	999	4	999		
.	.	.	.		
.	.	.	.		
.	.	.	.		
2	000	4	000		

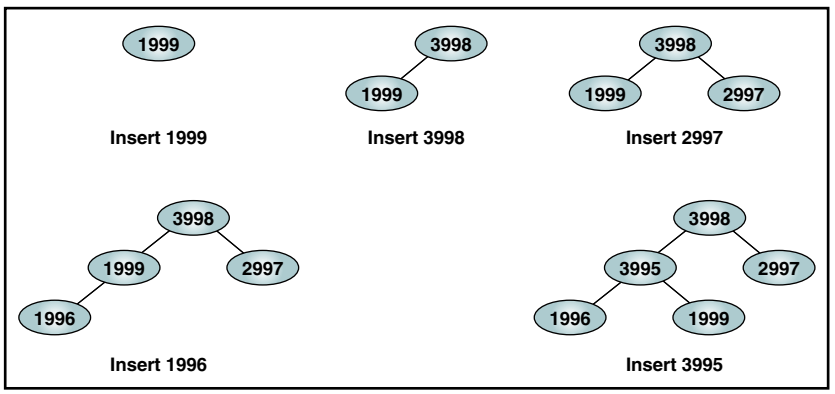
FIGURE 9-15 Priority Queue Priority Numbers

Example Assume that we have a priority queue with three priorities: high (3), medium (2), and low (1). Of the first five customers who arrive, the second and the fifth are high-priority customers, the third is medium priority, and the first and the fourth are low priority. They are assigned priority numbers as indicated in Table 9-1.

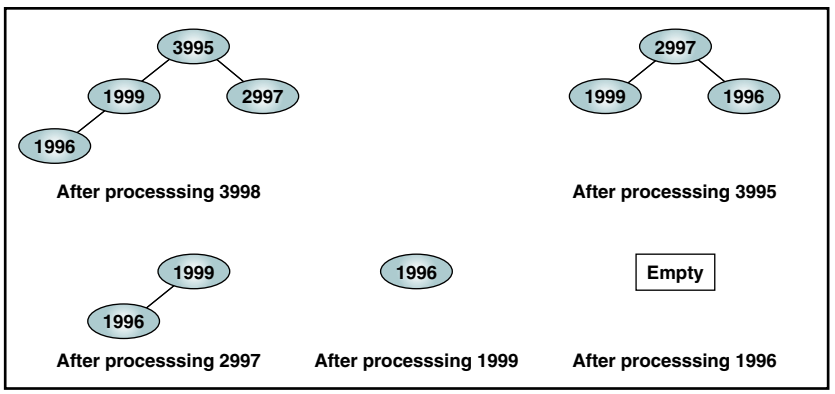
Arrival	Priority	Priority
1	low	1999 (1 & (1000 – 1)
2	high	3998 (3 & (1000 – 2)
3	medium	2997 (2 & (1000 – 3)
4	low	1996 (1 & (1000 – 4)
5	high	3995 (3 & (1000 – 5)

TABLE 9-1 Priority Number Assignments

Figure 9-16 traces the priority queue as it is built and processed, assuming that all of the customers arrive before the first customer is served. As you study the figure, note that the customers are served according to their priority and within equal priorities, according to their arrival. Thus we see that customer 2 (3998) is served first, followed by customer 5 (3995), customer 3 (2997), customer 1 (1999), and customer 4 (1996).



(a) Insert customers



(b) Process customers

FIGURE 9-16 Priority Queue Example

Implementation

The design of the priority queue is shown in Figure 9-17.

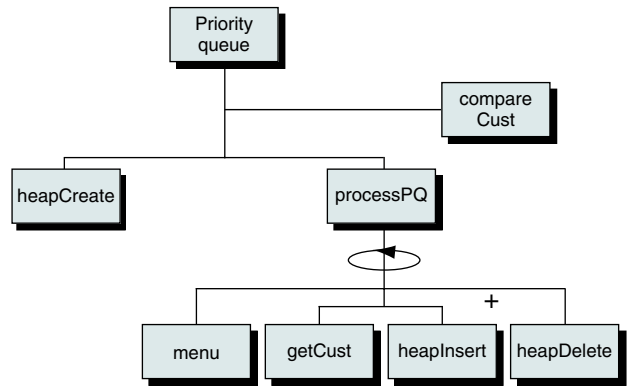


FIGURE 9-17 Priority Queue Design

As shown in the design, we call three functions from the head ADT: heap create, heap insert, and heap delete. After creating the priority queue, we use a menu function to determine the next action and then either insert a new customer into the priority queue or delete the next customer from the queue. When a customer is deleted, we display the customer ID to verify that the customers are processed first in, first out within their priorities. The code is shown in Program 9-7.

PROGRAM 9-7 Priority Queue Implementation

```
 1  /* Implement priority queue using heap.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <ctype.h>
 7  #include <stdbool.h>
 8
 9  #include "P9-heap.h"
10
11  // Constant Definitions
12     const int maxQueue = 20;
13
14  // Structure Declarations
15     typedef struct
16        {
17         int  id;
18         int  priority;
19         int  serial;
20        } CUST;
```

continued

PROGRAM 9-7 Priority Queue Implementation *(continued)*

```
21
22  // Prototype Declarations
23  int   compareCust (void* cust1, void* cust2);
24  void  processPQ   (HEAP* heap);
25  char  menu        (void);
26  CUST* getCust     (void);
27
28  int main (void)
29  {
30  // Local Definitions
31     HEAP* prQueue;
32
33  // Statements
34     printf("Begin Priority Queue Demonstration\n");
35
36     prQueue = heapCreate(maxQueue, compareCust);
37     processPQ (prQueue);
38
39     printf("End Priority Queue Demonstration\n");
40     return 0;
41  }  // main
42
43  /* ==================== compare ====================
44     Compare priority of two customers to determine
45     who has higher priority.
46        Pre  Given two customer structures
47        Post if cust1 >  cust2 return +1
48             if cust1 == cust2 return 0
49             if cust1 <  cust2 return -1
50  */
51  int compareCust (void* cust1, void* cust2)
52  {
53  // Local Definitions
54     CUST c1;
55     CUST c2;
56
57  // Statements
58     c1 = *(CUST*)cust1;
59     c2 = *(CUST*)cust2;
60
61     if (c1.serial < c2.serial)
62         return -1;
63     else if (c1.serial == c2.serial)
64         return 0;
65     return +1;
66  }  // compareCust
67
```

continued

PROGRAM 9-7 Priority Queue Implementation *(continued)*

```
/* ==================== processPQ ====================
   Compare priority of two customers to determine
   who has higher priority.
      Pre  Given two customer structures
      Post if cust1 >  cust2 return +1
           if cust1 == cust2 return 0
           if cust1 <  cust2 return -1
*/
void processPQ (HEAP* prQueue)
{
// Local Definitions
   CUST* cust;
   bool  result;
   char  option;
   int   numCusts = 0;

// Statements
   do
      {
       option = menu ();
       switch (option)
          {
           case 'e':
           cust = getCust ();
           numCusts++;
           cust->serial =
              cust->priority * 1000 + (1000 - numCusts);
              result = heapInsert (prQueue, cust);
              if (!result)
                  printf("Error inserting into heap\n"),
                      exit (101);
              break;
           case 'd':
            result = heapDelete (prQueue, (void**)&cust);
              if (!result)
                 printf("Error: customer not found\n");
              else
                {
                 printf("Customer %4d deleted\n",
                         cust->id);
                 numCusts--;
                } // else
          } // switch
      } while (option != 'q');
   return;
}  // processPQ

```

continued

PROGRAM 9-7 Priority Queue Implementation *(continued)*

```
/* ==================== menu ====================
   Display menu and get action.
      Pre  nothing
      Post action read and validated
*/
char menu (void)
{
// Local Declarations
   char option;
   bool valid;

// Statements
   printf( "\n============ Menu =============\n" );
   printf( " e :     Enter Customer Flight   \n" );
   printf( " d :     Delete Customer Flight  \n" );
   printf( " q :     Quit.                   \n" );
   printf( "================================\n" );
   printf( "Please enter your choice:  " );

   do
      {
       scanf(" %c", &option);
       option = tolower (option);
       switch (option)
          {
           case 'e':
           case 'd':
           case 'q': valid = true;
                     break;

          default:  printf("Invalid choice. Re-Enter: ");
                     valid = false;
                     break;
           } // switch
      }  while (!valid);
   return option;
}  // menu

/* ==================== getCust ====================
   Reads customer data from keyboard.
      Pre  nothing
      Post data read and returned in structure
*/
CUST* getCust (void)
{
// Local Definitions
   CUST* cust;
```

continued

PROGRAM 9-7 Priority Queue Implementation *(continued)*

```
162
163 // Statements
164    cust = (CUST*)malloc(sizeof (CUST));
165    if (!cust)
166       printf("Memory overflow in getCust\n"),
167           exit (200);
168
169    printf("Enter customer id:       ");
170    scanf ("%d", &cust->id);
171    printf("Enter customer priority: ");
172    scanf ("%d", &cust->priority);
173    return cust;
174 }  // getCust
```

9.5 Key Terms

heap
heapify
max-heap
min-heap
priority queue
reheap down
reheap up

9.6 Summary

- A heap is a complete or nearly complete binary tree structure in which the key value of each node is greater than or equal to the key value in each of its descendents.
- A heap can be recursively defined as a complete or nearly complete binary tree in which the key value in a node is greater than or equal to the key values in all of its subtrees and each subtree is itself a heap.
- The only operations that generally apply to a heap are insert and delete.
- To be able to insert or delete from a heap, we need two algorithms: reheap up and reheap down.
- The reheap up algorithm repairs a nonheap structure that is a heap except for the last element. It floats the last element up the tree until that element is in its correct location. The reheap up algorithm is needed when we insert an element into the heap and when we build a heap from an array.
- The reheap down algorithm repairs a nonheap structure that is made of two heaps and a root. The algorithm floats the root element down the tree until that element is in its correct location. We need a reheap down algorithm when we delete an element from a heap.
- A heap can be implemented in an array because a heap is a complete or nearly complete binary tree in which there is a fixed relationship between each node and its children.
- Three common applications of heaps are selection, priority queue, and sorting.
- A heap can be used to select the k^{th} largest (or smallest) element in an unsorted list.
- A heap can be used to implement a priority queue, in which every element has a priority number that determines its relationship to other elements.
- A heap can also be used to implement a sorting algorithm called the heap sort.

9.7 Practice Sets

Exercises

1. Show which of the structures in Figure 9-18 are heaps and which are not.

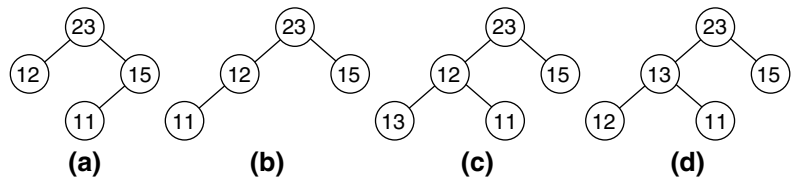


FIGURE 9-18 Heaps for Exercise 1

2. Make a heap out of the following data read from the keyboard:

```
23 7 92 6 12 14 40 44 20 21
```

3. Apply the reheap up algorithm to the nonheap structure shown in Figure 9-19.

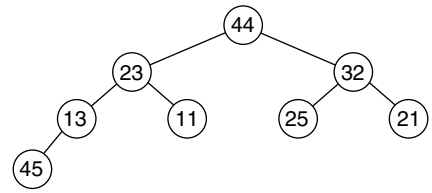


FIGURE 9-19 Nonheap Structure for Exercise 3

4. Apply the reheap down algorithm to the partial heap structure shown in Figure 9-20.

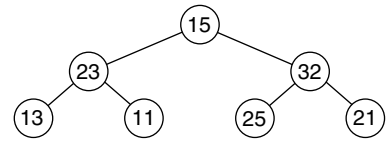


FIGURE 9-20 Partial Heap for Exercise 4

5. Show the array implementation of the heap in Figure 9-21.

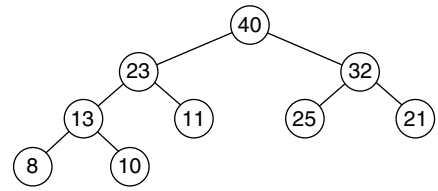


FIGURE 9-21 Heap for Exercises 5, 6, and 7

6. Apply the delete operation to the heap in Figure 9-21. Repair the heap after the deletion.
7. Insert 38 into the heap in Figure 9-21. Repair the heap after the insertion.
8. Show the left and right children of 32 and 27 in the heap in Figure 9-22. Also show the left children of 14 and 40.

40	27	32	15	14	20	25	11
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

FIGURE 9-22 Heap Array for Exercises 8 and 9

9. In the heap in Figure 9-22, show the parent of 11, the parent of 20, and the parent of 25.
10. If a node is at index 25, what is the index of its right child? What is the index of its left child? Assume the indexes start from 0.
11. If a node is at index 37, what is the index of its parent?
12. Which of the following sequences are heaps?
    a. 42 35 37 20 14 18 7 10
    b. 42 35 18 20 14 30 10
    c. 20 20 20 20 20 20
13. Show which item would be deleted from the following heap after calling the delete algorithm three times:

```
50 30 40 20 10 25 35 10 5
```

14. Show the resulting heap after 33, 22, and 8 are added to the following heap:

```
50 30 40 20 10 25 35 10 5
```

15. Draw a tree that is both a heap and a binary search tree.

16. Create a priority queue using the following data. The first number is the priority, and the letter is the data.

```
3-A 5-B 3-C 2-D 1-E 2-F 3-G 2-H 2-I 2-J
```

17. Show the contents of the priority queue in Exercise 16 after deleting three items from the queue.
18. Show the contents of the original priority queue in Exercise 16 after the following operations:

    Insert 4-K, Insert 3-L, Delete, Insert 2-M, Delete

Problems

19. Rewrite Algorithm 9-1, "Reheap Up," to build a minimum heap.
20. Rewrite Algorithm 9-2, "Reheap Down," to recreate a minimum heap.
21. Rewrite Algorithm 9-4, "Insert Heap," to build a minimum heap.
22. Rewrite Algorithm 9-5, "Delete Heap Node," to recreate a minimum heap.
23. Write an algorithm to combine two heaps and produce a third heap.
24. Write the C code for Problem 19.
25. Write the C code for Problem 20.
26. Write the C code for Problem 21.
27. Write the C code for Problem 22.
28. Write the C code for Problem 23.

Projects

29. Our study of tree algorithmics has shown that most tree structures are quite efficient. Let's examine the efficiency of heaps. Modify the heap ADT developed in Section 9.3 to determine the complexity of building a heap. For this program measure efficiency as the number of data moves necessary to build the heap.

    To determine a pattern, run your program with arrays filled with random numbers. Use five different array sizes: 100, 200, 500, 1000, and 2000. Then analyze the heuristics developed in these runs and determine which big-O notation best applies. Prepare a short report of your findings with appropriate tables and graphs.
30. Modify Project 29 to determine the efficiency of the reheap up and reheap down algorithms only. Again, analyze the data and prepare a short report of your conclusions regarding their efficiency.

31. Algorithm 9-3, Build Heap, uses the reheap up algorithm to build a heap from an array. Another, slightly faster way to build a heap from an array is to use reheap down in a loop starting from the middle of the array $\lfloor N / 2 \rfloor - 1$, which is the root of the last nonempty subtree, and working up the tree to the root. This concept is shown in Figure 9-23.

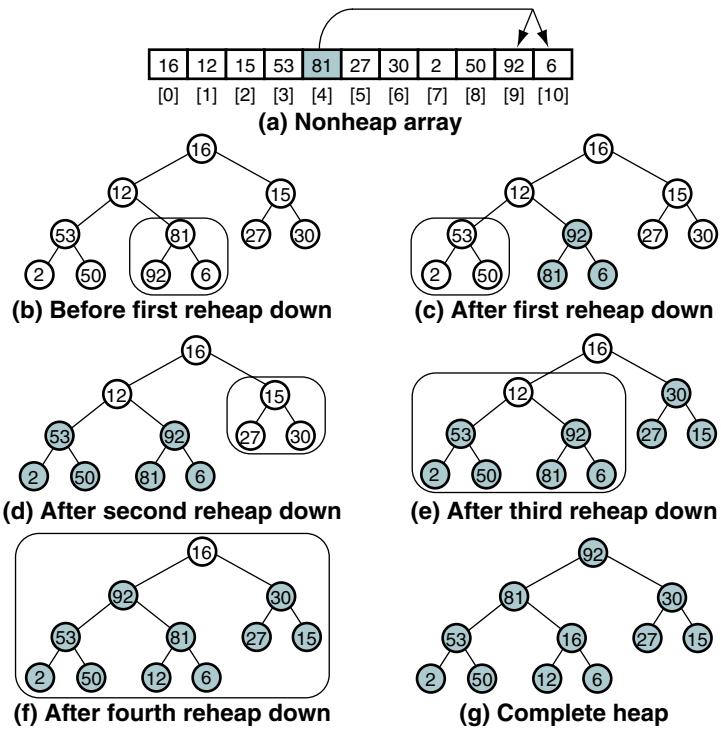


FIGURE 9-23 Build Heap from Last Subtree (Project 31)

Rewrite the build heap algorithm and then write the C code to implement it using the above approach.

32. Add code to Project 31 to determine the efficiency of building the heap from the middle up.

33. An airline company uses the formula shown below to determine the priority of passengers on the waiting list for overbooked flights.

$$\text{priority number} = A / 1000 + B - C$$

where
A is the customer's total mileage in the past year
B is the number of years in his or her frequent flier program
C is a sequence number representing the customer's arrival position when he or she booked the flight

Given a file of overbooked customers as shown in Table 9-2, write a program that reads the file and determines each customer's priority number. The program then builds a priority queue using the priority number and prints a list of waiting customers in priority sequence.

Name	Mileage	Years	Sequence
Bryan Devaux	53,000	5	1
Amanda Trapp	89,000	3	2
Baclan Nguyen	93,000	3	3
Sarah Gilley	17,000	1	4
Warren Rexroad	72,000	7	5
Jorge Gonzales	65,000	2	6
Paula Hung	34,000	3	7
Lou Mason	21,000	6	8
Steve Chu	42,000	4	9
Dave Lightfoot	63,000	3	10
Joanne Brown	33,000	2	11

TABLE 9-2 Data for Project 33

Chapter 10
Multiway Trees

We studied the general concepts of trees in Chapter 6. We also studied three types of binary trees: binary search trees (Chapter 7), AVL trees (Chapter 8), and heaps (Chapter 9). As binary trees grow in size, their heights can become significant. For example, a tree with 1000 entries has a height of at least 10, and a tree with 100,000 entries has a height of at least 17. If the trees are unbalanced, their heights can be significantly larger.

In this chapter we explore trees whose outdegree is not restricted to 2 but that retain the general properties of binary search trees. Whereas each node in a binary tree has only one entry, multiway trees have multiple entries in each node and thus may have multiple subtrees. You will find these structures in applications such as internal search trees, spell checkers, and external file indexes.

10.1 *M*-way Search Trees

An ***m*-way tree** is a search tree in which each node can have from 0 to m subtrees, where m is defined as the **B-tree order.** Given a nonempty multiway tree, we can identify the following properties:

1. Each node has 0 to m subtrees.
2. A node with $k < m$ subtrees contains k subtrees and $k - 1$ data entries.
3. The key values in the first subtree are all less than the key value in the first entry; the key values in the other subtrees are all greater than or equal to the key value in their parent entry.
4. The keys of the data entries are ordered $key_1 \le key_2 \le \ldots \le key_k$.
5. All subtrees are themselves multiway trees.

Figure 10-1 is a diagrammatic representation of an m-way tree of order 4.

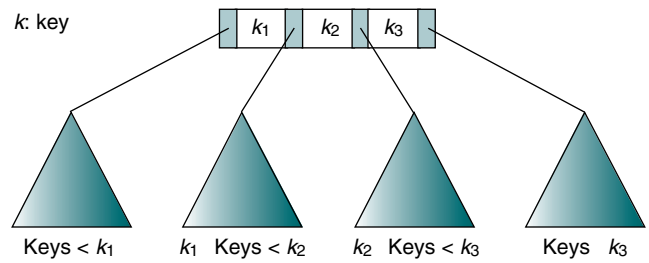


FIGURE 10-1 *M*-way Tree of Order 4

Study Figure 10-1 carefully. The first thing to note is that it has the same structure as the binary search tree: subtrees to the left of an entry contain data with keys that are less than the key of the entry, and subtrees to the right of an entry contain data with keys that are greater than or equal to the entry's key. This ordering is easiest to see in the first and last subtrees.

Now study the second subtree. Note that its keys are greater than or equal to k_1 and less than k_2. It serves as the right subtree for K_1 and at the same time the left subtree for k_2. In other words, whether a subtree is a left or a right subtree depends on which node entry you are viewing.

Also note that there is one more subtree than there are entries in the node; there is a separate subtree at the beginning of each node. This first subtree identifies all of the subtrees that contain keys less than the first entry in the node.

Because each node has a variable number of entries, we need some way to keep track of how many entries are currently in the node. This is done with an entry count, which is not shown in Figure 10-1. Figure 10-2 is a four-way tree.

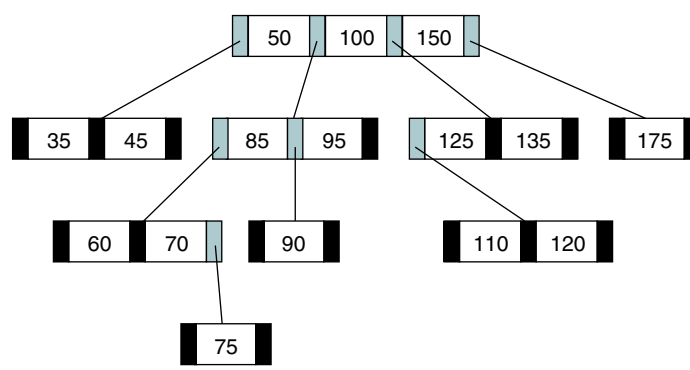


FIGURE 10-2 Four-way Tree

We are now ready to code the description of an *m*-way tree node. Let's first build a structure for the entries. Because the number of entries varies up to a specified maximum, the best structure in which to store them is an array. Each entry needs to hold the key of the data, the data itself (or a pointer to the data if stored elsewhere), and a pointer to its right subtree. Figure 10-3(a) depicts the structure.

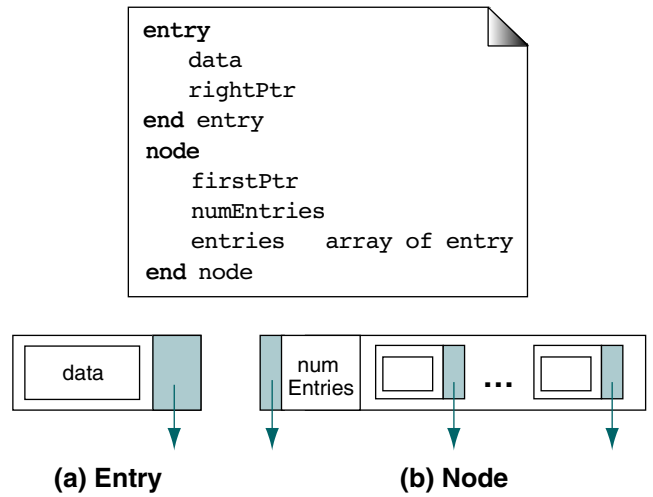


FIGURE 10-3 M-way Node Structure

The node structure contains the first pointer to the subtree with entries less than the key of the first entry, a count of the number of entries currently in the node, and the array of entries. The array must have room for $m - 1$ entries. The node structure is shown in Figure 10-3(b).

Before leaving the discussion of *m*-way trees, there is one more thought we would like to leave with you. The binary search tree is an *m*-way tree of order 2.

The binary search tree is an *m*-way tree of order 2.

10.2 B-trees

The *m*-way tree has the potential to greatly reduce the height of a tree. However, it still has one major deficiency: it is not balanced. In 1970 two computer scientists working for the Boeing Company in Seattle, Washington,

created a new tree structure they called the B-tree.[1] A **B-tree** is an m-way search tree with the following additional properties:

1. The root is either a leaf or it has 2 … m subtrees.
2. All internal nodes have at least $\lceil m / 2 \rceil$ nonnull subtrees and at most m nonnull subtrees.
3. All leaf nodes are at the same level; that is, the tree is perfectly balanced.
4. A leaf node has at least $\lceil m / 2 \rceil - 1$ and at most $m - 1$ entries.

From the definition of a B-tree, it should be apparent that a B-tree is a perfectly balanced m-way tree in which each node, with the possible exception of the root, is at least half full. Table 10-1 defines the minimum and maximum numbers of subtrees in a nonroot node for B-trees of different orders.

	Number of subtrees		Number of entries	
Order	Minimum	Maximum	Minimum	Maximum
3	2	3	1	2
4	2	4	1	3
5	3	5	2	4
6	3	6	2	5
…	…	…	…	…
m	$\lceil m / 2 \rceil$	m	$\lceil m / 2 \rceil - 1$	$m - 1$

TABLE 10-1 Entries in B-trees of Various Orders

Figure 10-4 contains a B-tree of order 5. Let's examine its design. First, by the m-way rules in Section 10.1, the tree is a valid m-way tree. Now let's look at the B-tree rules. The root is not a leaf—it has two subtrees (Rule 1). The left internal node has the minimum number of subtrees (three), and the right internal node has the maximum number of subtrees (five) (Rule 2). All leaf nodes are at level 2 (Rule 3). The leaves all have between the minimum (two) and the maximum (four) entries (Rule 4). Therefore, the tree is a valid B-tree.

1. R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica 1,* no. 3 (1972): 173–189. This paper does not explain why they called it *B-tree*. Two possibilities are *B* is for Bayer or that *B* is just a sequence similar to the C Language name.

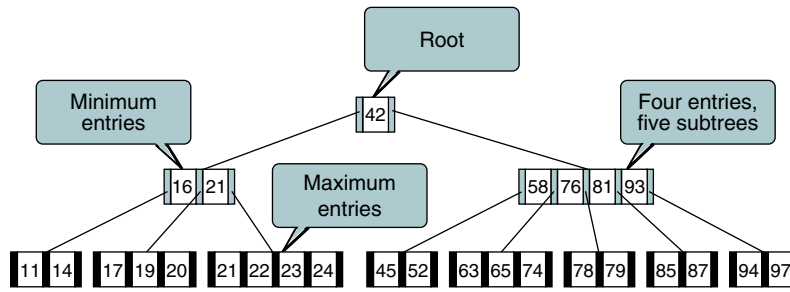


FIGURE 10-4 A B-tree of Order 5

B-tree Implementation

The four basic operations for B-trees are: insert, delete, traverse, and search. In this section we discuss these four algorithms and any algorithms they call.

B-tree Insertion

Like the binary search tree, B-tree insertion takes place at a leaf node. The first step, therefore, is to locate the leaf node for the data being inserted. If the node is not full—that is, if it has fewer than $m - 1$ entries, the new data are simply inserted in sequence in the node.

A B-tree grows from the bottom up.

When the leaf node is full, we have a condition known as **overflow**. Overflow requires that the leaf node be split into two nodes, each containing half of the data. To split the node, we allocate a new node from the available memory and then copy the data from the *end* of the full node to the new node. After the data have been split, the new entry is inserted into either the original or the new node, depending on its key value. Then the median data entry is inserted into the parent node.

To help us understand how to build a B-tree, let's run through a simple example. Given a B-tree structure with an order of 5, we begin by inserting 11, 21, 14, and 78. The first insertion creates a node that becomes the root. The next three insertions simply place the data in the node in ascending key sequence. At this point we have a tree that looks like the one in Figure 10-5(f).

When we try to insert 97, we discover that the node is full. We therefore create a new right subtree and move the upper half of the data to it, leaving the rest of the data in the original node. This situation is shown in Figure 10-5(b). Note that the new value (97) has been placed in the new node because it logically belongs to the upper half of the data.

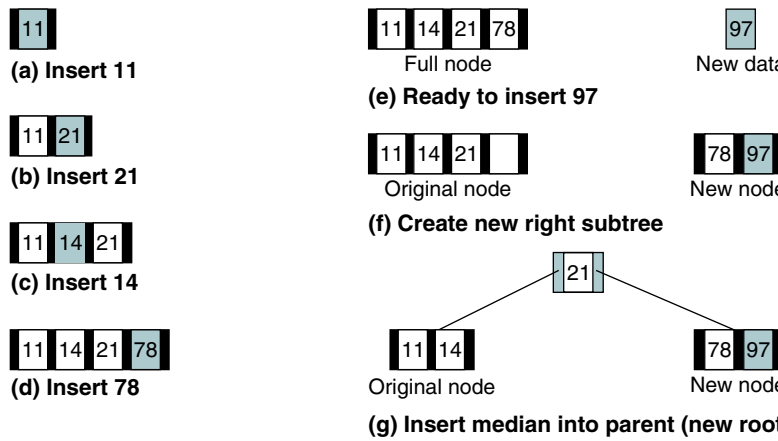


FIGURE 10-5 B-tree Insert Overview

After creating the new node, we insert the median-valued data (21) into the parent of the original node. Because the original node was a root, we create a new root and insert 21 into it. This step completes the insertion of 97 into the B-tree, which has now grown by one level. The resulting B-tree is shown in Figure 10-5(g).

The B-tree insert design is shown in Figure 10-6. The process of inserting an entry into the parent provides an interesting contrast to the binary search tree. Recall that the binary search tree grew in an unbalanced fashion from the top down. B-trees grow in a balanced fashion from the bottom up. When the root node of a B-tree overflows and the median entry is pushed up, a new root node is created and the tree grows one level.

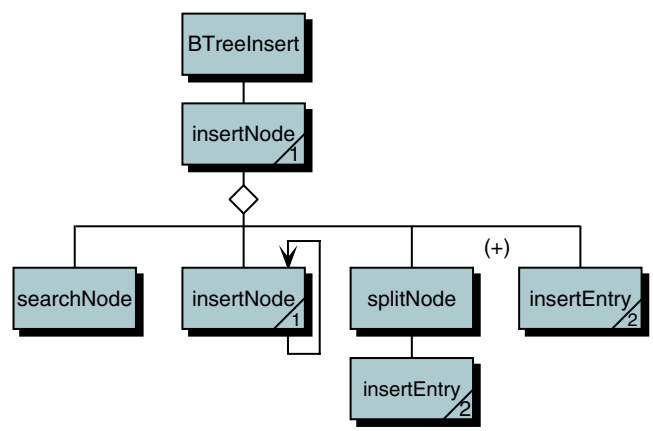


FIGURE 10-6 B-tree Insert Design

The B-tree insert pseudocode is shown in Algorithm 10-1.

ALGORITHM 10-1 B-tree Insert

```
Algorithm BTreeInsert (tree,  data)
Inserts data into B-Tree. Equal keys placed on right branch.
   Pre    tree is reference to B-Tree; may be null
   Post   data inserted
1  if (tree null)
   Empty tree. Insert first node.
   1   create new node
   2   set left subtree of node to null
   3   move data to first entry in new node
   4   set subtree of first entry to null
   5   set tree root to address of new node
   6   set number of entries to 1
2  end if
   Insert data in existing tree
3  insertNode (tree, data, upEntry)
4  if (tree higher)
      Tree has grown; create new root
   1   create new node
   2   move upEntry to first entry in new node
   3   set left subtree of node to tree
   4   set tree root to new node
   5   set number of entries to 1
   6   end if
end BTreeInsert
```

Algorithm 10-1 Analysis

The B-tree insert algorithm has just three processes. First, if the tree is null, it creates a root. Second, if the tree is not null, it calls the insert node algorithm. As we will see, insert node is a recursive algorithm that not only inserts data into a leaf but also inserts any overflow entries into their parent nodes. If the root overflows, however, there is no parent in which to insert the median entry. The third process, therefore, creates a new root when the tree overflows. Whenever there is an overflow, the insert node algorithm passes the median entry back to the B-tree insert algorithm, through the parameter `upEntry`. This median entry becomes the entry for the new root.

Insert Node

The insert node algorithm is the heart of the B-tree insert. Because it is complex, you need to walk through an example to understand it. Let's trace the building of a B-tree of order 5. We have already shown an overview of the first part of the process. Let's pick it up where we are ready to insert 57 into the tree. We begin by calling B-tree insert (Algorithm 10-1) and passing it the root and the new data to be inserted (57), as shown in Figure 10-7.

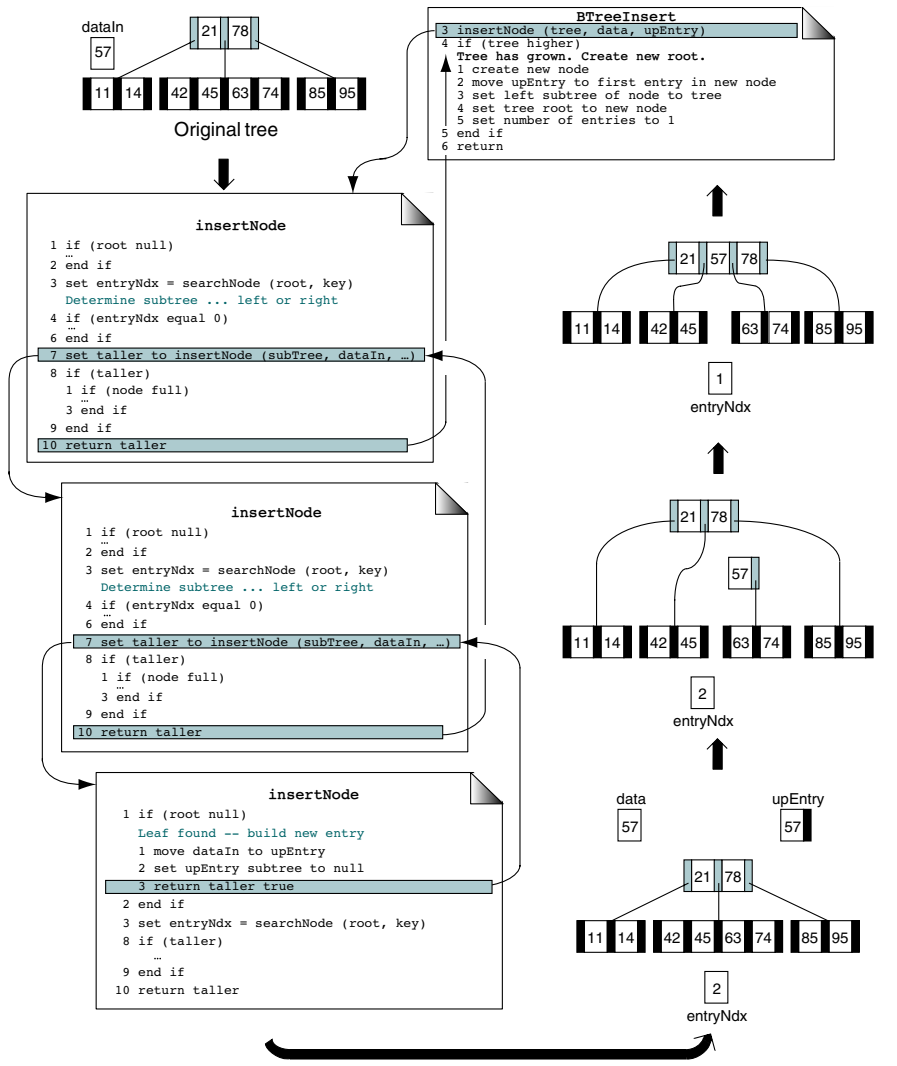


FIGURE 10-7 Build B-tree, First Overflow

We now trace Figure 10-7 through insert node, as shown in Algorithm 10-2.

ALGORITHM 10-2 B-tree Insert Node

```
Algorithm insertNode (root, dataIn, upEntry)
Recursively searches tree to locate leaf for data. If node
overflows, inserts median key's data into parent.
   Pre    root is reference to tree or subtree; may be null
          dataIn contains data to be inserted
          upEntry is reference to entry structure
   Post   data inserted
          upEntry is overflow entry to be inserted into
             parent
   Return boolean (taller)
 1 if (root null)
   Leaf found -- build new entry
   1  move dataIn to upEntry
   2  set upEntry subtree to null
   3  return taller true
 2 end if
   Search for entry point (leaf)
 3 set entryNdx to searchNode (root, key)
   Determine subtree ... left or right
 4 if (entryNdx equal 0)
   1  if (data key < key in first entry)
      1  set subtree to left subtree of node
   2  else
      1  set subtree to subtree of entry
   3  end if
 5 else
   1  set subtree to entryNdx rightPtr
 6 end if
 7 set taller to insertNode (subTree, dataIn, upEntry)
   Data inserted -- back out
 8 if (taller)
   1  if (node full)
      1  splitNode (root, entryNdx, newEntryLow, upEntry)
      2  set taller to true
   2  else
      1  insertEntry (root, entryNdx, upEntry)
      1  insert upEntry in root
      2  set taller to false
   3  end if
 9 end if
10 return taller
end insertNode
```

Algorithm 10-2 Analysis

B-tree insert node (**insertNode**) is a recursive algorithm (see statement 7). Let's follow the code through Figure 10-7. The base case is a null subtree, indicating that the leaf node was located in the previous call. In the first call, the root is not null, so **insertNode** calls **searchNode** to determine the correct subtree branch, found at 21, and recursively calls itself using 21's right subtree pointer.

The third call reaches the base case for this insertion. Because the root is null, **insertNode** moves the new data and a null pointer to the new entry (statements 1.1 and 1.2) and returns true, indicating that the new entry is ready to be inserted. Note that the new entry must be passed by reference.

We then return to **insertNode** in statement 7 and are ready to insert the new entry into the leaf node as we back out of the recursion. If the node is full, we split it and then insert the median entry into the current node's parent (statement 8.1.1). The logic for **splitNode** is shown in Algorithm 10-4 and will be discussed in detail shortly. For now it is sufficient to understand that **splitNode** creates a new right subtree node and moves data to it. On the other hand, if the node is not full, we simply insert it into the current node.

Because the node is full, we insert the median valued entry (57) upward into the parent node containing 21 and 78. This step completes the insertion because there is room for 57 in the parent node. We must still complete the recursion, however, by backing all of the way out.

Search Node

Search node first checks to see if the target is less than the first entry's key. If it is, it returns entry 0. If the target is not less than the first node, it locates the target's node by starting at the end and working toward the beginning of the entries. This design is more efficient than starting at the beginning (see Algorithm 10-3 Analysis below). The pseudocode code is shown in Algorithm 10-3.

ALGORITHM 10-3 B-tree Search Node

```
Algorithm searchNode (nodePtr, target)
Search B-Tree node for data entry containing key <= target.
   Pre    nodePtr is reference to nonnull node
          target is key to be located
   Return index to entry with key <= target
          -or- 0 if key < first entry in node
 1 if (target < key in first entry)
   1  return 0
 2 end if
 3 set walker to number of entries – 1
 4 loop (target < entry key[walker])
   1  decrement walker
 5 end loop
 6 return walker
end searchNode
```

Algorithm 10-3 Analysis

This algorithm uses a very interesting search. It starts at the end of the entry array and works toward the beginning. We use this search technique because each entry points to a subtree with data whose keys are greater than or equal to the current entry's key and less than the next entry's key. If we searched from the beginning, we would overshoot our target and have to back up. By searching from the end, when we find an entry less than or equal to the target, we have also found the subtree pointer to the subtree that contains the desired target.

A close analysis of the algorithm reveals that there are two conditions when it returns an index of 0. The first occurs when the target is less than the first entry (statement 1). The second situation occurs when the target is greater than or equal to the first entry and less than the second entry. In this case the data are found in the first entry's right subtree.

Split Node

Split node takes a full node, an entry that needs to be inserted into the node, and the index location for the new entry and splits the data between the existing node, a median entry, and a new node. The code is shown in Algorithm 10-4.

ALGORITHM 10-4 B-tree Split Node

```
Algorithm splitNode (node, entryNdx, newEntryLow, upEntry)
Node has overflowed. Split node into two nodes.
   Pre    node is reference to node that overflowed
          entryNdx contains index location of parent
          newEntryLow true if new data < entryNdx data
          upEntry is reference to entry being inserted
   Post   upEntry contains entry to be inserted into parent
1  create new node
   Build right subtree node
2  move high entries to new node
3  if (entryNdx < minimum entries)
   1  Insert upEntry in node
4  else
   1  insert upEntry in new node
5  end if
   Build entry for parent
6  move median data to upEntry
7  make new node firstPtr the right subtree of median data
8  make new node the right subtree of upEntry
end splitNode
```

Algorithm 10-4 Analysis

We need to analyze three different insert positions: the new key is less than the median key, the new key is the median key, and the new key is greater than the median key. The first two reduce to the same case. Figure 10-8(a) shows the step-by-step logic for splitting a node when the new entry is less than or equal to the median, and Figure 10-8(b) shows the steps when the new entry is greater than the median.

In both cases we begin by allocating a new node and copying data from the end of the original node to the beginning of the new node. Then we insert the new entry, found in **upEntry**, into either the original node or the new node. Finally, we copy the median entry to **upEntry**. Note that when we enter Algorithm 10-4, **upEntry** contains the data being inserted into the overflow node; after the node has been split, **upEntry** contains the data to be inserted into the parent when we return.

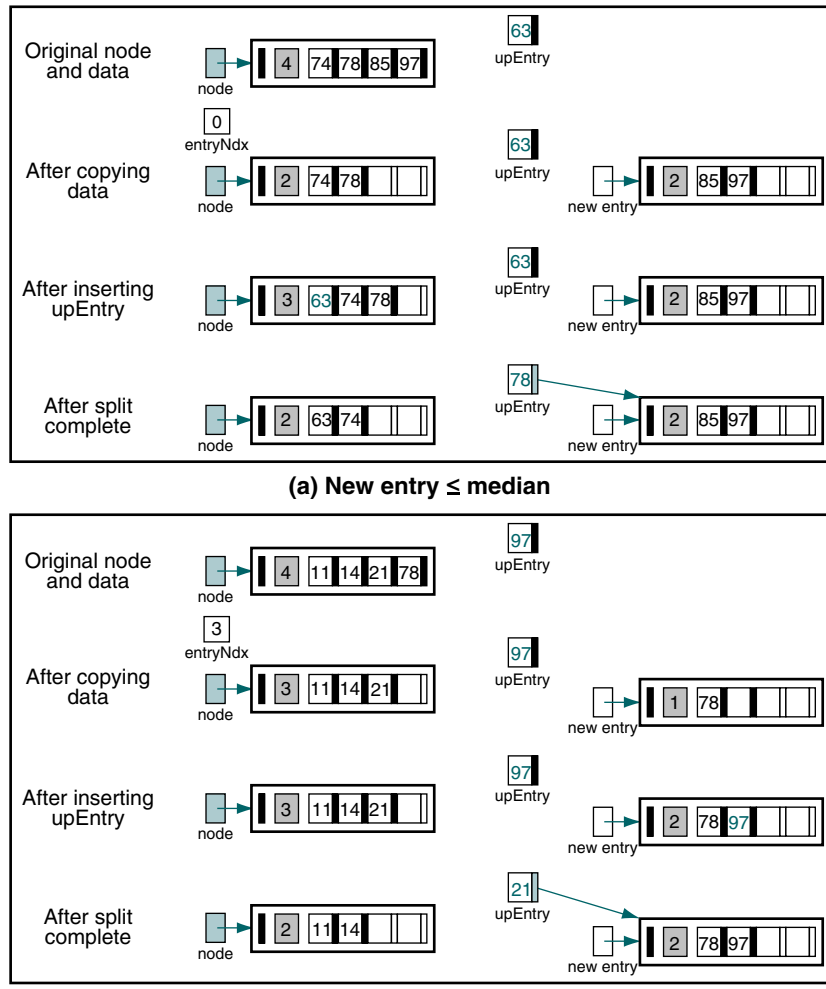


FIGURE 10-8 Split Node B-tree Order of 5

When the new key is less than or equal to the median key, the new data belong in the left or original node. On the other hand, if the new key is greater than the median key, the new data belong in the new node. At statement 3, therefore, we test the location for the new entry (**`entryNdx`**) and call the insert entry algorithm, passing it either the original node or the new node.

If you study Figure 10-8 carefully, you will note that regardless of which situation occurred, the median value is always in the same place, identified by the minimum number of entries. Also, the right pointer in the median entry is always the new node's pointer. It is thus easy to build the median entry.

The test cases in Figure 10-8 are found in a larger context in Figure 10-9. Figure 10-8(a) is the last insertion in Figure 10-9(c), and Figure 10-8(b) is the same as Figure 10-9(b).

Insertion Summary

We have now looked at all of the algorithms necessary to insert data into a B-tree. As a summary we build a complete B-tree in Figure 10-9. You have already seen some of the trees in previous figures. Study this figure carefully to ensure that you can build a similar tree with different data.

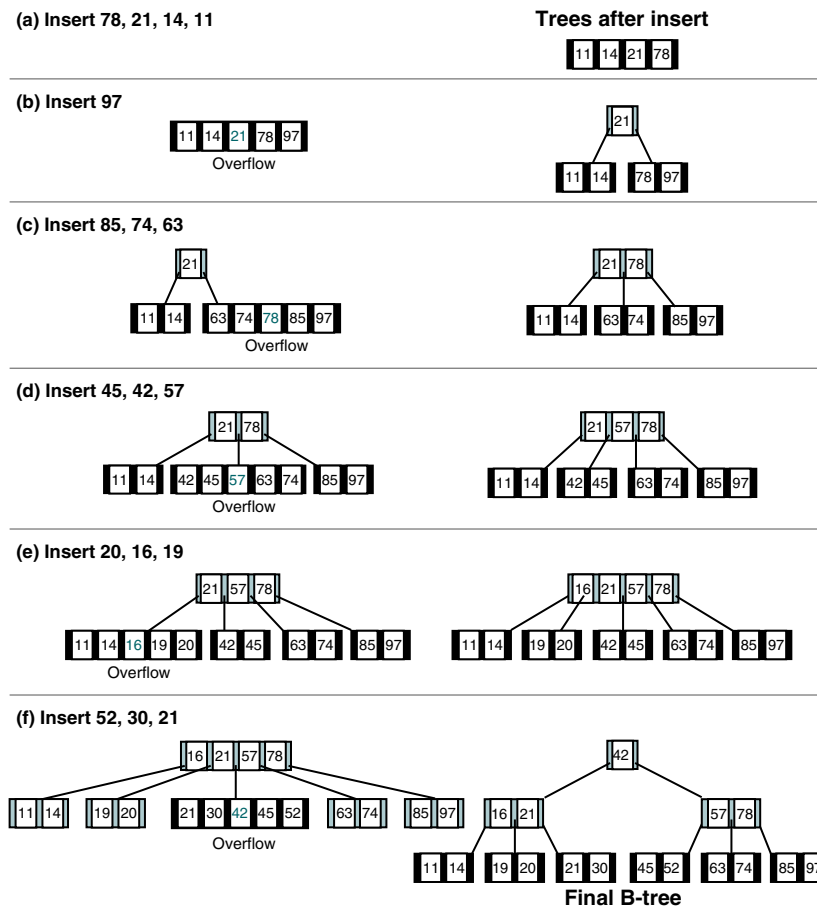


FIGURE 10-9 Building a B-tree of Order 5

B-tree Deletion

There are three considerations when deleting a data entry from a B-tree node. First, we must ensure that the data to be deleted are actually in the tree. Second, if the node does not have enough entries after the deletion, we need to correct the structural deficiency. A deletion that results in a node with fewer than the minimum number of entries is an **underflow**. Finally, as we saw with the binary search tree, we can delete only from a leaf node. Therefore, if the data to be deleted are in an internal node, we must find a data entry to take their

place. The design for the B-tree deletion is shown in Figure 10-10. We suggest that you study this design before proceeding with its algorithms.

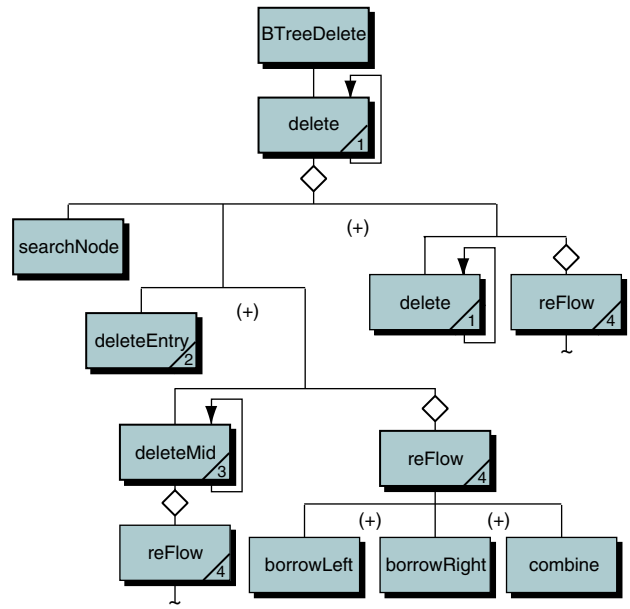


FIGURE 10-10 B-tree Delete Design

Note that there are two levels of recursion. The first, Algorithm 10-6, traverses the tree, looking for the node to be deleted and, if it is found at a leaf, simply deletes it. For the second, Algorithm 10-8 is called recursively when the data to be deleted are not at a leaf. The algorithm could be written with only one recursion, but it is more complex to follow.

Now study the node delete logic. It searches for the data to be deleted. If they are not found, it prints an error message and terminates the recursion. This is the first base case. The second base case occurs when the data have been found and deleted. In either case the delete must determine whether it has caused an underflow and set a return Boolean—true if underflow occurred or false if the node is okay.

The underflow processing takes place as delete node backs out of the recursion. After each return it checks the node to make sure it has not underflowed. If an underflow occurred, it corrects the underflow and continues. With this understanding of the big picture, we are now ready to look at the algorithms.

Like the insert, the B-tree delete requires two algorithms: a user interface and a recursive delete function. The high-level user interface calls the recursive delete and then checks for root underflow. It is shown in Algorithm 10-5.

ALGORITHM 10-5 B-tree Delete

```
Algorithm BTreeDelete (tree, dltKey)
Delete entry with key target from B-tree.
   Pre    tree is a reference to a B-tree
          dltKey is the key of the entry to be deleted
   Post   data deleted or false returned
   Return success (found) or failure (not found)
 1 if (tree empty)
   1  return false
 2 end if
 3 delete (tree, dltKey, success)
 4 if (success)
   1  if (tree number of entries zero)
         Tree is shorter--delete root
      1  set tree to left subtree
   2  end if
 5 end if
 6 return success
end BTreeDelete
```

Algorithm 10-5 Analysis

B-tree delete has two responsibilities. First, it starts the delete process by calling the recursive algorithm, **`delete`**. Then, when the delete process is complete, it determines whether the root has underflowed. If it has, the address of the new root is found in the node's first pointer.

Delete Node

The heart of the delete process is shown in the recursive delete algorithm. If delete reaches a null subtree, it has not found the data to be deleted and returns underflow false.

If the root is not null, delete searches the node to see if the data to be deleted are in it. This is the same algorithm we used for the insert search (see Algorithm 10-3, "B-tree Search Node"). The search returns either 0, indicating that the delete key is less than the first entry in the node, or an entry index for a node that is less than or equal to the delete key. When the delete key is not in the current node, delete calls itself recursively with a new subtree. If the delete key has been found, it is deleted by calling either `deleteEntry` or `deleteMid`. After deleting the node, delete checks for underflow and if necessary repairs the underflow by calling `reFlow`. The pseudocode is shown in Algorithm 10-6.

ALGORITHM 10-6 B-tree Delete Node

```
Algorithm delete (root, deleteKey, success)
Recursively locates node containing deleteKey; deletes data.
   Pre    root is a reference to a nonnull B-Tree
          deleteKey is key to entry to be deleted
          success is reference to boolean
   Post   data deleted--success true; or success false
          underflow--true or false
   Return success true or false
 1 if (root null)
      Leaf node found--deleteKey key does not exist
   1  set success false
   2  return false
 2 end if
 3 set entryNdx to searchNode (root, deleteKey)
 4 if (deleteKey found)
      Found entry to be deleted
   1  set success to true
   2  if (leaf node)
      1  set underflow to deleteEntry (root, entryNdx)
   3  else
         Entry is in internal node
      1  if (entryNdx > 0)
         1  set leftPtr to rightPtr of previous entry
      2  else
         1  set leftPtr to root firstPtr
      3  end if
      4  set underflow to deleteMid (root, entryNdx, leftPtr)
      5  if (underflow)
         1  set underflow to reFlow (root, entryNdx)
      6  end if
   4  end if
 5 else
   1  if (deleteKey less key in first entry)
      1  set subtree to root firstPtr
   2  else
         deleteKey is in right subtree
      1  set subtree to entryNdx rightPtr
   3  end if
   4  set underflow to delete (subtree, deleteKey, success)
   5  if (underflow)
      1  set underflow to reFlow (root, entryNdx)
   6  end if
```

continued

ALGORITHM 10-6 B-tree Delete Node *(continued)*

```
 6  end if
 7  return underflow
end delete
```

Algorithm 10-6 Analysis

This delete algorithm calls four different algorithms, including itself. The recursive call is found in statement 5.4. Note that after the recursive calls, we must test for underflow. This logic is important. Although the underflow is detected when a node underflows, it is handled *after the logic flow returns to the parent*. It is handled at this point because to resolve an underflow we need the parent, the left subtree, and the right subtree.

Delete Entry

Delete entry removes the entry from a node and compresses it; that is, it moves the entries on the right of the deleted entry to the left. After deleting the entry, it tests for underflow and returns a Boolean—true if underflow occurred and false if the node is okay. The code is shown in Algorithm 10-7.

ALGORITHM 10-7 B-tree Delete Entry

```
Algorithm deleteEntry (node, entryNdx)
Deletes entry at entryNdx from leaf node.
   Pre    node is reference to node with data to be deleted
          entryNdx is index of entry in node
   Post   entry deleted
   Return underflow <Boolean>
 1 shift entries after delete to left
 2 if (number of entries less minimum entries)
   1  return true
 3 else
   1  return false
 4 end if
end deleteEntry
```

Delete Mid

As we stated earlier, all deletions must take place at a leaf node. When the data to be deleted are not in a leaf node, we must find substitute data. There are two choices for substitute data: either the immediate predecessor or the immediate successor. Either will do, but it is more efficient to use the immediate predecessor because it is always the last entry in a node and no shifting is required when it is deleted. We therefore use the immediate predecessor.

The immediate predecessor is the largest node on the left subtree of the entry to be deleted. The initial calling algorithm, `delete`, determines the left subtree and passes it as a parameter along with the node containing the data to be deleted and the index to its entry in the array. `deleteMid` recursively follows the left subtree's right pointer in the last entry until it comes to a leaf node.

Finding a leaf node is the recursion base case for `deleteMid`. At this point it replaces the data to be deleted with the data in the leaf's last entry and then calls itself to delete the predecessor entry in the leaf node. The pseudocode for `deleteMid` is shown in Algorithm 10-8.

ALGORITHM 10-8 B-tree Delete Mid

```
Algorithm deleteMid (node, entryNdx, subtree)
Deletes entry from internal node in tree.
   Pre    node is reference to node containing delete entry
          entryNdx is index to entry to be deleted in node
          subtree is reference to node's subtree
   Post   delete data replaced with immediate predecessor
          and predecessor deleted from leaf node
   Return underflow true or false
   Find entry to replace node being deleted
1  if (no rightmost subtree)
      Leaf located. Replace data and delete leaf entry.
   1  move predecessor's data to delete entry
   2  set underflow if node entries less minimum
2  else
      Not located. Traverse right to locate predecessor.
   1  set underflow to deleteMid
                          (node, entryNdx, right subtree)
   2  if (underflow)
      1  set underflow to reFlow (root, entryNdx)
   3  end if
3  end if
4  return underflow
end deleteMid
```

Algorithm 10-8 Analysis

This algorithm is rather simple. As we have seen in other algorithms, the base case is handled first in statement 1. The recursive search for a leaf node is found in statement 2. Note that both the base case and the general case determine whether underflow has occurred and pass the underflow status to the return statement at the end of the code.

Figure 10-11 shows the interaction of the B-tree delete, delete, and delete mid algorithms. You may want to study it before going on.

Reflow

When underflow has occurred, we need to bring the underflowed node up to a minimum state by adding at least one entry to it. This process is known as **reflow.** This is perhaps the most difficult logic in the B-tree delete. To understand it we need to review two concepts: balance and combine. **Balance** shifts data among nodes to reestablish the integrity of the tree. Because it does not change the structure of the tree, it is less disruptive and thereforepreferred. **Combine** joins the data from an underflowed entry, a minimal sibling, and a parent in one node. Combine thus results in one node with the maximum number of entries and an empty node that must be recycled.

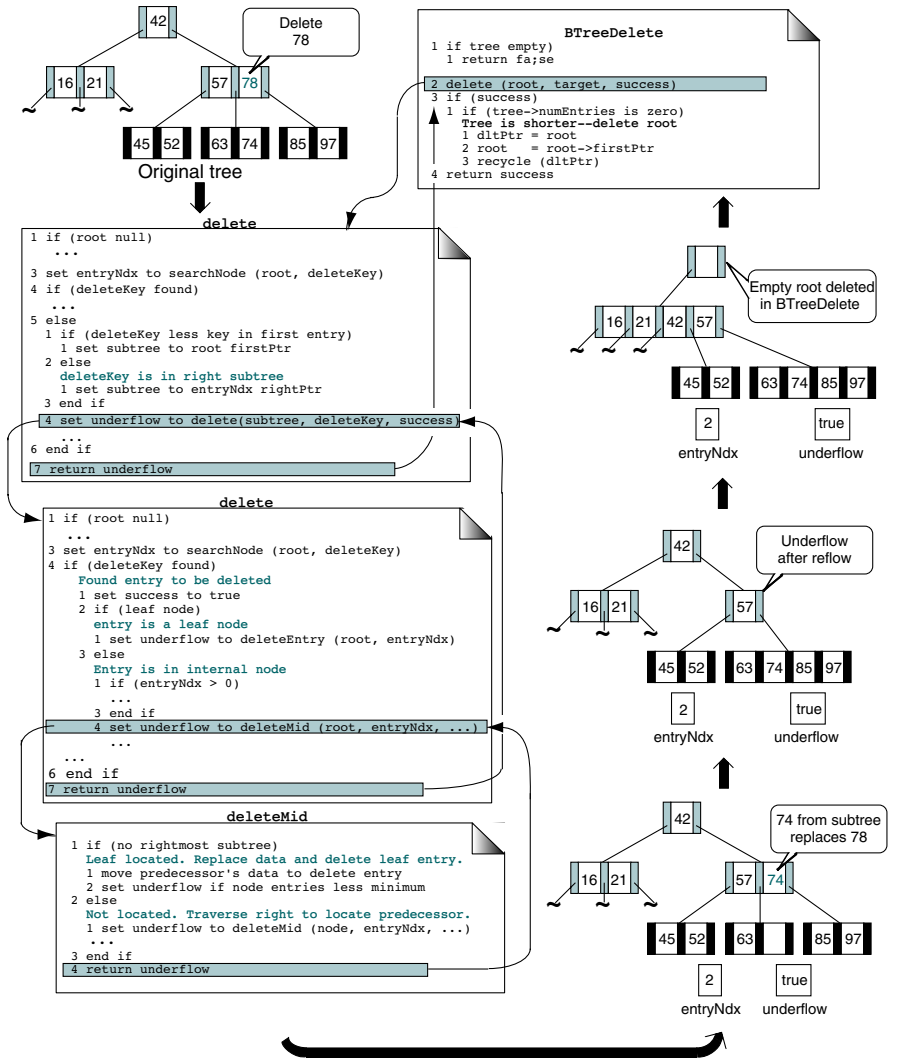


FIGURE 10-11 B-tree Deletions

When we enter `reFlow`, root contains the parent to the node that underflowed, identified by the entry index; that is, the entry index identifies the subtree that is an underflow state. We begin by examining the left and right subtrees of the root to determine whether one has more than the minimum number of entries. If either one does, we can order the tree by balancing. If neither the left nor the right subtree has more than the minimum entries, we must combine them with the parent. The pseudocode is shown in Algorithm 10-9.

ALGORITHM 10-9 B-tree Underflow Reflow

```
Algorithm reflow (root, entryNdx)
An underflow has occurred in one of the subtrees to the
root, identified by the entry index parameter. Correct
underflow by either balancing or combining subtrees.
   Pre    underflow has occurred in a subtree in root
          entryNdx identifies parent of underflow subtree
   Post   underflow corrected
   Return underflow true if node has underflowed
   Try to borrow first. Try right subtree first.
1  if (rightTree entries greater minimum entries)
   1  borrowRight (root, entryNdx, leftTree, rightTree)
   2  set underflow to false
2  else
      Can't balance from right. Try left.
   1  if (leftTree entries greater minimum entries)
      1  borrowLeft (root, entryNdx, leftTree, rightTree)
      2  set underflow to false
   2  else
         Can't borrow. Must combine entries.
      1  combine (root, entryNdx, leftTree, rightTree)
      2  if (root numEntries less minimum entries)
         1  set underflow to true
      3  else
         1  set underflow to false
      4  end if
   3  end if
3  end if
4  return underflow
end reflow
```

Algorithm 10-9 Analysis

The first part of the algorithm determines whether it is possible to correct the tree by balancing. We know that one of the subtrees is below the minimum. If we can find one that is above the minimum, we can balance. We first check the right subtree, and if it is above the minimum we call the borrow right algorithm. If it is not, we check the left subtree. Again, if it is above the minimum, we call the borrow left algorithm to restore order. If neither of the subtrees has an extra entry, we must combine them by calling a separate combine algorithm.

Underflow can occur only when the nodes are combined. Because borrowing takes place from a node with more than the minimum number of entries, it cannot underflow. Therefore, when we borrow, either from the left or from the right, underflow is set to false. On the other hand, when we combine nodes, we delete the parent entry from the root. We must therefore check to make sure it has not underflowed. If it has, we return underflow true.

Balance

We balance a tree by rotating an entry from one sibling to another through the parent. In `reFlow` we determine the direction of the rotation. The simplest implementation, therefore, is to write separate algorithms, one for rotating from the left and one for rotating from the right. The rotations are graphically shown in Figure 10-12.

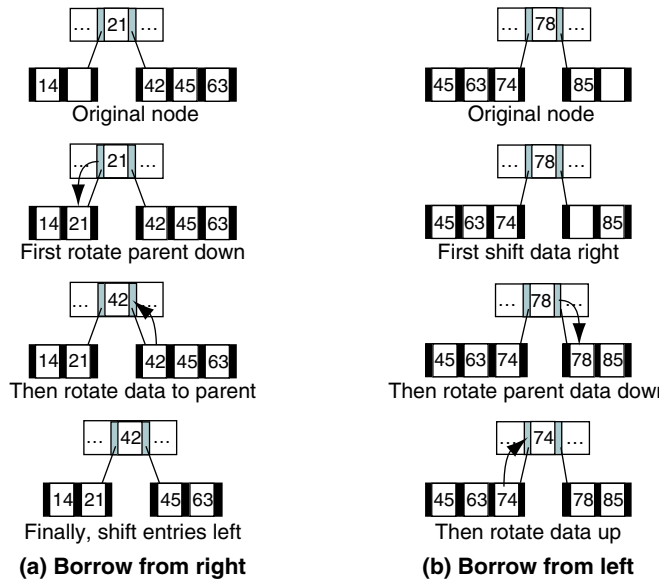


FIGURE 10-12 Restoring Order by Borrowing

The pseudocode for borrow left is shown in Algorithm 10-10, and the pseudocode for borrow right is shown in Algorithm 10-11.

ALGORITHM 10-10 B-tree Borrow Left

```
Algorithm borrowLeft (root, entryNdx, left, right)
It has been determined that the right subtree of root has
underflowed. Borrow a node from the left subtree and rotate
through the parent.
   Pre    root is parent of node that underflowed
          entryNdx is parent entry
          left a subtree that contains extra node(s)
          right a subtree that underflowed
   Post   subtrees are balanced
   Shift entries right to make room for new data in entry 0
 1 shift all elements one to the right
   Move parent data down and reset right pointer
 2 move root data to first entry in right
 3 move right first pointer to right subtree of first entry
   Moved entry's rightPtr becomes right tree first pointer
 4 move left last right pointer to right first pointer
   Move data from left to parent
 5 move left last entry data to root at entryNdx
end borrowLeft
```

ALGORITHM 10-11 B-tree Borrow Right

```
Algorithm borrowRight (root, entryNdx, left, right)
It has been determined that the left subtree of root has
underflowed. Borrow a node from the right subtree and rotate
through the parent.
   Pre    root is parent of node that underflowed
          entryNdx is parent entry
          left a subtree that underflowed
          right a subtree that contains extra node(s)
   Post   subtrees are balanced
   Move parent and subtree pointer to left tree
 1 move root data to left last entry
 2 move right first subtree to left last entry right subtree
   Move right data to parent
 3 move right first entry data to root at entryNdx
   Set right tree first pointer and shift data
 4 set right firstPtr to right first entry right subtree
 5 shift all entries one to the left
end borrowRight
```

Algorithm 10-10 and Algorithm 10-11 Analysis

The code for these two algorithms is rather straightforward. When we borrow from the right, the parent node entry's data, but not its subtree, moves down to become the last entry in the left subtree. We must therefore find a new right subtree for it. We find it in the first subtree in the right subtree (see Algorithm 10-10, statement 2). To replace the moved first subtree, we use the right subtree of the first entry (see statement 4).

Similarly, when we borrow from the left, the data in the last entry of the left subtree is copied to the root entry's data, but its pointer becomes the first pointer of the right subtree. We suggest that you create several balancing situations and follow the algorithms carefully until you are comfortable with the rotation and pointer manipulation.

Combine

When we can't balance, we must combine nodes. Figure 10-13 shows the logic for combining when there is an underflow. As you study it, note that the two subtrees and the parent entry are all combined into the left node. Then the parent entry is deleted from its node, which may result in an underflow. What is not apparent from the figure is that the right subtree node is then recycled.

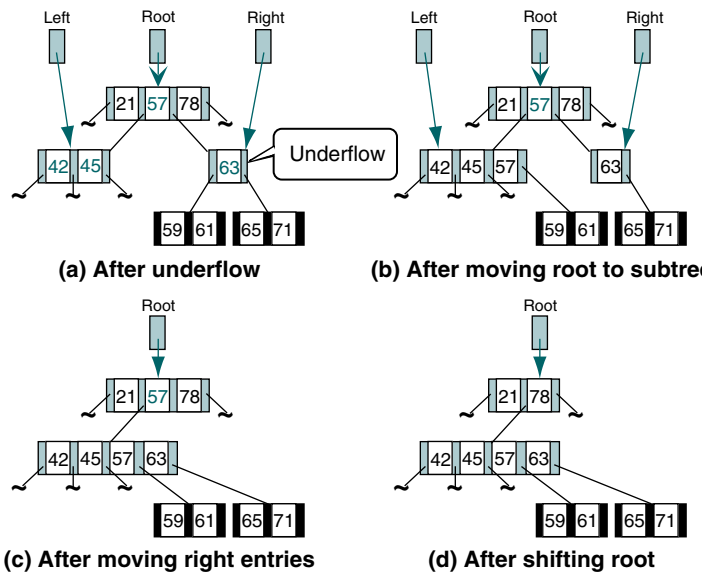


FIGURE 10-13 B-tree Combine

The pseudocode for combine is shown in Algorithm 10-12.

ALGORITHM 10-12 B-tree Combine

```
Algorithm combine (root, entryNdx, left, right)
Underflow has occurred, and we are unable to borrow an entry.
The two subtrees must be combined with the parent.
   Pre    root contains the parent of the underflowed entry
          entryNdx identifies the parent entry
          left and right are pointers to subtrees
   Post   parent and subtrees combined
          right tree node has been recycled
   Move parent and set its rightPtr from right tree
 1 move parent entry to first open entry in left subtree
 2 move right subtree first subtree
        to moved parent left subtree
   Move data from right tree to left tree
 3 move entries from right subtree to end of left subtree
   Now shift data in root to the left
 4 shift root data to left
end combine
```

Algorithm 10-12 Analysis Once again the most difficult part of this algorithm is the subtree manipulation. Regardless of which subtree has underflowed, we combine all nodes into the left subtree so we can recycle the right subtree.

Study statement 2 carefully. We have just moved the parent data to the first empty entry in the left subtree. Its right subtree then becomes the first subtree of the right subtree. Remember that the first subtree identifies the subtree whose entries are greater than the parent and less than the first entry. This subtree must therefore become the right subtree when the parent is moved to the left subtree.

Figure 10-14 is a summary of the B-tree deletions. It contains an example of each of the major algorithms required to delete data from a B-tree. Study it carefully until you understand the concepts.

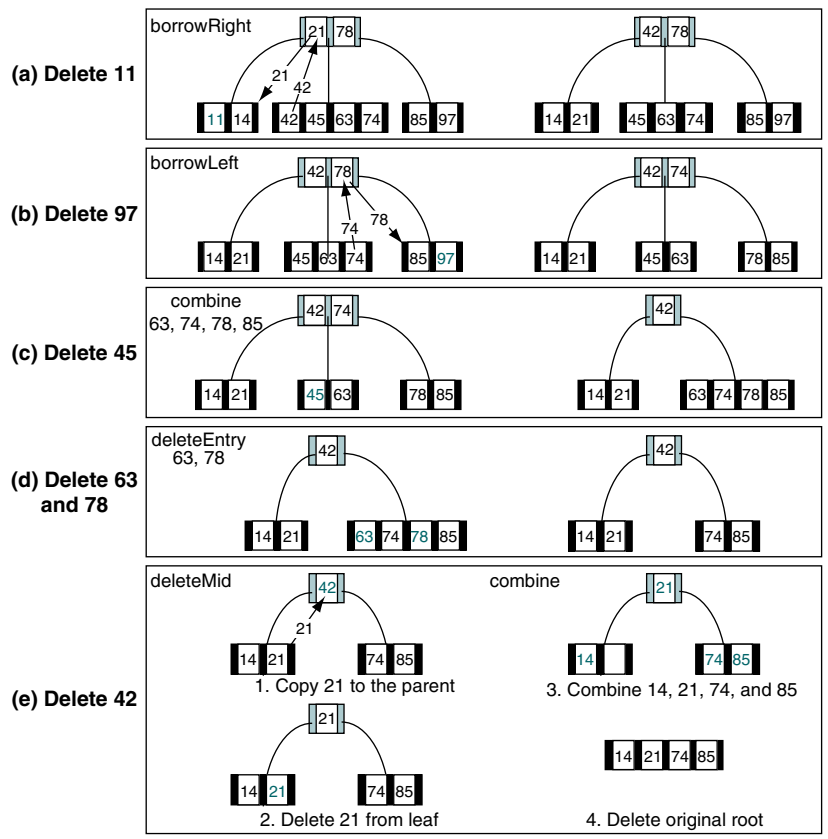


FIGURE 10-14 B-tree Deletion Summary

Traverse B-tree

Now that we have a B-tree, let's look at the logic to traverse it. Because a B-tree is built on the same structure as the binary search tree, we can use the same basic traversal design—the inorder traversal. The major difference, however, is that with the exception of leaf nodes, we don't process all of the data in a node

at the same time. Therefore, we must remember which entry was processed whenever we return to a node and continue from that point. With recursion this logic is relatively simple. Figure 10-15 shows the processing order of the entries as we "walk around" the tree. In this figure each entry is processed as we walk below it. Note that, as expected, the data are processed in sequential order.

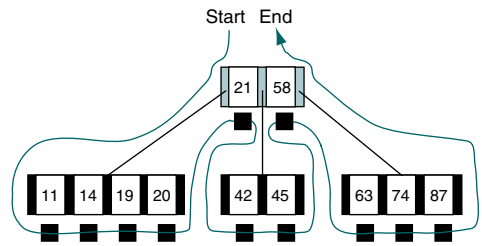


FIGURE 10-15 Basic B-tree Traversal

The traversal logic is shown in Algorithm 10-13.

ALGORITHM 10-13 B-tree Traversal

```
Algorithm BTreeTraversal (root)
Process tree using inorder traversal.
   Pre   root is pointer to B-Tree; it may not be null
   Post  every entry has been processed in order
1  set scanCount to 0
2  set nextSubTree to root left subtree
3  loop (scanCount <= number of entries)
      Test for subtree
   1  if (nextSubTree not null)
      1  BTreeTraversal (nextSubTree)
   2  end if
      Subtree processed--get next entry
   3  if (scanCount < number of entries)
      1  process (entry[scanCount])
      2  set nextSubTree to current entry right subtree
   4  end if
   5  increment scanCount
4  end loop
end BTreeTraversal
```

Algorithm 10-13 Analysis In the inorder traversal, data are processed after the left subtree and before the right subtree. The left subtrees are processed in the recursive call found in statement 3.1.1. When we return from processing a subtree, therefore, we are ready to process the parent entry. We do so by checking the scan count and, if it is less than the number of entries in the node, we call an algorithm to process the data (statement 3.3.1). We

then process the right subtree by setting the next subtree to the right subtree and continuing the loop in statement 3.

It is tempting to think that the loop can terminate when the scan count becomes equal to the number of entries in the node. Although this is true when we are processing a leaf node, it is not true for internal nodes. The loop cannot terminate because when we have processed the data in the last entry of a node, we still need to process its right subtree.

B-tree Search

We have seen an algorithm that searches a node, but we have not seen one that traverses the entire tree looking for a target key. When we searched a binary search tree, we simply returned the node that contained the target data. Because there are multiple entries in a B-tree node, however, returning the node is not sufficient; we must also return the entry that contains the target. Algorithm 10-14 therefore uses reference parameters for the located node and its entry index.

ALGORITHM 10-14 B-tree Search

```
Algorithm BTreeSearch (root, srchKey, node, foundLoc)
Recursively search a B-Tree for the srchKey key.
   Pre    root is a reference to a tree or subtree
          srchKey is the data to be located
          node is a reference to a found subtree
          foundLoc is reference to index for found entry
   Post   if found--
             node contains address of located node
             foundLoc contains index entry within node
          if not found--
             returns false
   Return found -- true or false
1  if (empty tree)
   1  return false
2  end if
3  if (srchKey < first entry)
   1  return BTreeSearch (root first subtree,
                          srchKey, node, foundLoc)
4  end if
   Search from last (rightmost) entry down to first
5  set foundLoc to number of entries - 1
6  loop (srchKey < key of current entry)
   1  decrement foundLoc
```

continued

ALGORITHM 10-14 B-tree Search *(continued)*

```
 7  end loop
 8  if (srchKey equal to key of current entry)
    1   set node to root
    2   return true
 9  end if
    Not less than or equal so search right subtree
10 return BTreeSearch (current entry right subtree,
                       srchKey, node, foundLoc)
end BTreeSearch
```

Algorithm 10-14 Analysis The logic for Figure 10-15 is similar to the design we saw in Algorithm 10-3, "B-tree Search Node." Both algorithms search from the end of the node toward the beginning. The major difference is that we must search the subtrees as well as the current node. Thus we need two different recursive calls in the algorithm, the first when the target is less than the first entry (statement 3.1) and the second when it is greater than the current entry (statement 10).

There are two base cases: we reach a null subtree, which indicates that the target doesn't exist, or we find the target entry. The first base case is handled in statement 1. In this case we simply return false.

The equal base case is shown in statement 8. Because we have set the entry number in the loop, we need only set the node address to the current root pointer and return true.

One final point: study statement 6 carefully. What prevents the loop from running off the beginning of the array? The answer to this question is found in statement 3. The target's key must be equal to or greater than the first entry's; if it were less, we would have followed the first pointer to the left subtree. Because we know that the target's key is equal to or greater than the first entry's, it becomes a sentinel that stops the loop. Therefore, we don't need to test for the beginning of the loop, which makes the loop very efficient—it need test only one condition.

10.3 B-tree ADT

With an understanding of how a B-tree works, it should be no surprise that the B-tree implementation is rather complex. In this section we develop code for the key algorithms. Some are left for you to develop. Figure 10-16 provides a basic list of functions. One that is not provided is update—that is, changing the contents of an entry. Under the current design, to change the contents of an entry we would have to delete and reinsert it.

B-tree Data Structure

As with the other ADT structures, we implement the B-tree ADT with a head structure and a node structure.

Head Structure

The B-tree head structure, `BTREE`, contains a count, a root pointer, and the address of the compare function needed to search the list. The application

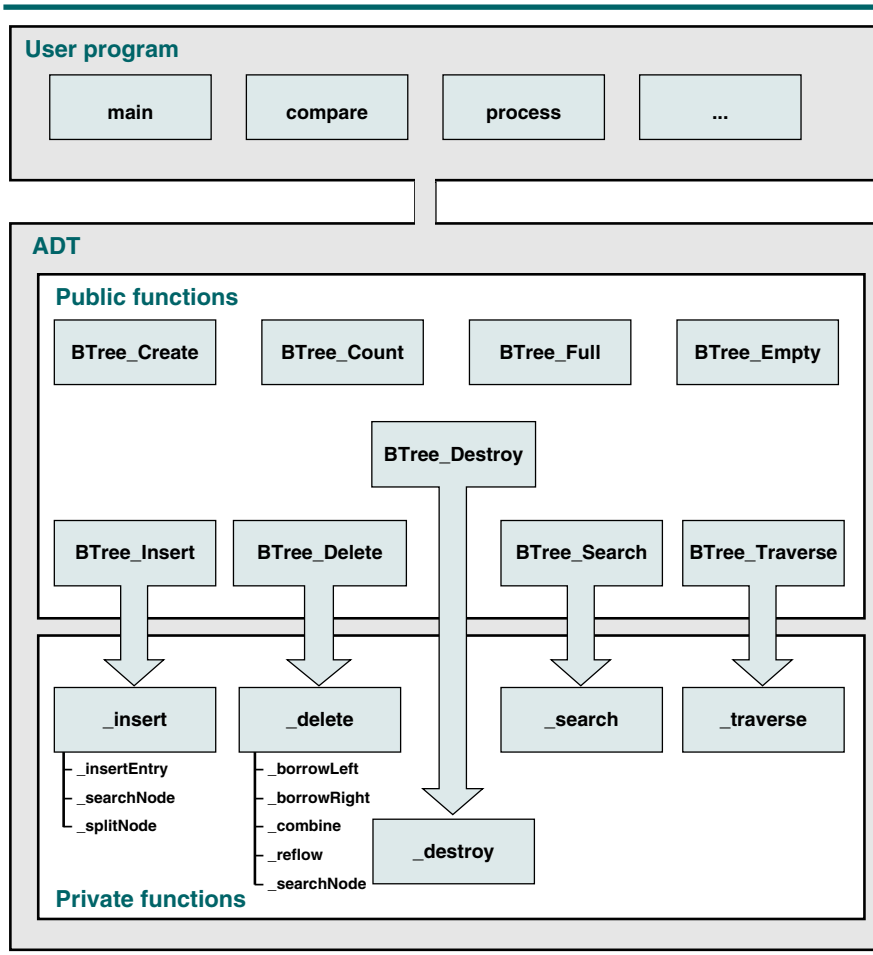
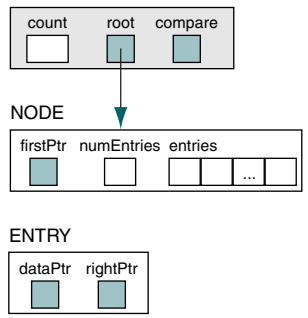


FIGURE 10-16 B-tree Function Family

program's only view of the tree is a pointer to the head structure, which is allocated from dynamic memory when the tree is created. The head data structure is shown in Figure 10-17.

Node Structure

The B-tree node structure requires three separate structures: a tree head structure, a node structure, and an entry structure. The tree head structure contains the root pointer, any required metadata, and a pointer to the compare function. The node structure stores a pointer to the node's left subtree, an array of entries, and a count of the current number of entries in the node. Each data entry contains a data pointer to a structure with a key and attributes and a pointer to the entry's right subtree. The node structure is shown in Figure 10-17.




```
typedef struct
 {
  void*        dataPtr;
  struct node* rightPtr;
 } ENTRY;

typedef struct node
 {
  struct node* firstPtr;
  int          numEntries;
  ENTRY        entries[ORDER - 1];
 } NODE;

typedef struct
   {
    int   count;
    NODE* root;
    int  (*compare) (void* argu1, void* argu2);
   } BTREE;
```

FIGURE 10-17 B-tree Data Structure

Header File

The header file for the B-tree ADT contains the data structures and the prototype declarations for the user interface. It is shown in Program 10-1.

PROGRAM 10-1 B-tree Declaration

```
/* ==================== B-Tree.h ====================
   This header file contains the functions for the AVL
   Tree abstract data type.
      Written by:
      Date:
*/
#include <stdlib.h>
#include <stdbool.h>

// =============== CONSTANTS & MACROS ==============
const int ORDER = 5;
const int MIN_ENTRIES = (((ORDER + 1) / 2) - 1);

// ================== STRUCTURES ==================
struct node;

typedef struct
   {
    void*        dataPtr;
    struct node* rightPtr;
   } ENTRY;

typedef struct node
```

continued

PROGRAM 10-1 B-tree Declaration *(continued)*

```
   {
    struct node* firstPtr;
    int          numEntries;
    ENTRY        entries[ORDER - 1];
   } NODE;

typedef struct
   {
    int   count;
    NODE* root;
    int  (*compare) (void* argu1, void* argu2);
   } BTREE;

// =========== Prototype Declarations ===========

// User interfaces
BTREE* BTree_Create
       (int  (*compare) (void* argu1, void* argu2));
void   BTree_Traverse
       (BTREE* tree, void (*process) (void* dataPtr));
BTREE* BTree_Destroy (BTREE* tree);
void   BTree_Insert  (BTREE* tree, void* dataInPtr);
bool   BTree_Delete  (BTREE* tree, void* dltKey);
void*  BTree_Search  (BTREE* tree, void* dataPtr);
bool   BTree_Empty   (BTREE* tree);
bool   BTree_Full    (BTREE* tree);
int    BTree_Count   (BTREE* tree);

// Internal BTree functions
static void* _search
               (BTREE* tree, void*  targetPtr,
                NODE*  root);
static int   _searchNode
               (BTREE* tree, NODE* nodePtr,
                void*  target);
static bool  _delete
               (BTREE* tree,      NODE* root,
                void*  dltKeyPtr, bool* success);
static bool _insert
               (BTREE* tree,      NODE*  root,
                void*  dataInPtr, ENTRY* upEntry);
static void _traverse
               (NODE* root,
                void  (*process)(void* dataPtr));
static void _splitNode
               (NODE*  root,    int    entryNdx,
                int compResult, ENTRY* upEntry);
static void _insertEntry
```

continued

PROGRAM 10-1 B-tree Declaration *(continued)*

```
72                  (NODE*  root, int entryNdx,
73                   ENTRY  upEntry);
74  static bool _deleteEntry
75                  (NODE*  node, int entryNdx);
76  static bool _deleteMid
77                  (NODE*  root, int entryNdx,
78                   NODE*  leftPtr);
79  static bool _reFlow
80                  (NODE*  root, int entryNdx);
81  static void _borrowLeft
82                  (NODE*  root,     int   entryNdx,
83                   NODE*  leftTree, NODE* rightTree);
84  static void _borrowRight
85                  (NODE*  root,     int   entryNdx,
86                   NODE*  leftTree, NODE* rightTree);
87  static void _combine
88                  (NODE*  root,       int   entryNdx,
89                   NODE*  leftTree,   NODE* rightTree);
90  static void _destroy (NODE* root);
```

Algorithms

In this section we implement the basic algorithms for a B-tree.

B-tree Search

The search B-tree function is interesting in that it requires an ADT interface function and an internal recursive search function. From the calling function's point of view, all it needs to supply is the tree and the data to be located. Whereas the pseudocode algorithm returned the key's node address and entry index, in the ADT we return the data pointer. (Remember that the application does not have access to the node structure.) Internally, however, we need to know which tree node we are processing at any given time in the search. The code for the ADT interface is shown in Program 10-2.

PROGRAM 10-2 B-tree Search Application Interface

```
1   /* ================== BTree_Search ==================
2      Search the tree for the node containing
3      requested key and returns pointer to its data.
4         Pre    tree has been created (may be null)
5                targetPtr is pointer to data structure
6                     containing key to be located
7         Post   tree searched and data pointer returned
8         Return pointer to data
9   */
10  void* BTree_Search (BTREE* tree, void* targetPtr)
11  {
```

continued

PROGRAM 10-2 B-tree Search Application Interface *(continued)*

```
12  // Statements
13     if (tree->root)
14         return _search
15                   (tree, targetPtr, tree->root);
16     else
17         return NULL;
18  }  // BTree_Search
```

Internal Search Function

The internal search implementation is shown in Program 10-3. It parallels Algorithm 10-14.

PROGRAM 10-3 Internal Search Function

```
 1  /* ================= _search =================
 2     Search tree for node containing requested
 3     key and returns its data to the calling function.
 4        Pre    BTree_Search passes tree, targetPtr, root
 5               targetPtr is pointer to data structure
 6                  containing key to be located
 7        Post   tree searched and data pointer returned
 8        Return address of matching node returned
 9               If not found, NULL returned
10  */
11  void* _search (BTREE* tree, void* targetPtr,
12                 NODE*  root)
13  {
14  // Local Definitions
15     int   entryNo;
16
17  // Statements
18     if (!root)
19        return NULL;
20
21     if (tree->compare(targetPtr,
22                       root->entries[0].dataPtr) < 0)
23         return _search (tree,
24                         targetPtr,
25                         root->firstPtr);
26
27     entryNo = root->numEntries - 1;
28     while (tree->compare(targetPtr,
29                     root->entries[entryNo].dataPtr) < 0)
30        entryNo--;
31     if (tree->compare(targetPtr,
32                    root->entries[entryNo].dataPtr) == 0)
33         return (root->entries[entryNo].dataPtr);
```

continued

PROGRAM 10-3 Internal Search Function *(continued)*

```
34
35       return (_search (tree,
36            targetPtr, root->entries[entryNo].rightPtr));
37    }  // _search
```

Program 10-3 Analysis

First, note the name of the function. It begins with an underscore. It is reasonable to assume that a programmer using the ADT may decide to use the same name as one of our ADT functions. Therefore, to prevent duplicates, we prefix all of our internal names with the underscore.

We need to receive the B-tree structure because it contains the compare function written by the user. We use the compare function first in statement 21 and again in statements 28 and 31. Study these statements carefully. If you are not familiar with passing a function, you may want to review the concept in Chapter 1.

The search begins by testing to see if the search argument found in the key pointer is less than the first entry in the node. If it is, we follow the first pointer to the left. If it is not, we must search the entry array for the matching entry or the correct right subtree to continue the search. The entry array search we use is an adaptation of the ordered list search (see Chapter 13). The major difference is that it starts from the last entry and works forward. We search backward because each entry points to a subtree with data whose keys are greater than or equal to the current entry's key and less than the next entry's key. If we searched from the beginning, we would overshoot our target and have to back up. By searching from the end, when we find an entry less than or equal to the target, we have also found the subtree pointer to the subtree that contains the desired target.

B-tree Traverse

The B-tree traversal is interesting for several reasons. First, once again we require two functions: one for the ADT interface and one for the internal traversal. Second, we must pass the name of the function that is to be used to process each node. The user interface code, including receiving the process function's address as a parameter, is shown in Program 10-4.

PROGRAM 10-4 B-tree Traverse Application Interface

```
1  /* ================ BTree_Traverse ================
2     Process tree using inorder traversal.
3        Pre  Tree has been created (may be null)
4             tree is pointer to B-Tree
5             process used to "visit" nodes in traversal
6        Post Entries processed in LNR sequence
7  */
8  void BTree_Traverse (BTREE* tree,
9                       void (*process) (void* dataPtr))
```

continued

PROGRAM 10-4 B-tree Traverse Application Interface *(continued)*

```
{
// Statements
   if (tree->root)
        _traverse (tree->root, process);
   return;
}  // end BTree_Traverse
```

Internal Traverse Function

Another interesting point in the traversal logic is that as we traverse the tree, using a variation of the inorder traversal, we must deblock each entry. We studied the deblocking logic in Algorithm 10-13, "B-tree Traversal." The code is shown in Program 10-5.

PROGRAM 10-5 Internal Traverse Function

```
/* ================== _traverse ==================
   Traverse tree using inorder traversal. To "process"
   node, using function passed when traversal called
      Pre  tree validated in BTree_Traversal
           root is pointer to B-Tree node
           process is function to process an entry
      Post All nodes processed
*/
 void _traverse (NODE* root,
                 void (*process) (void* dataPtr))
{
// Local Definitions
   int   scanCount;
   NODE* ptr;

// Statements

   scanCount = 0;
   ptr = root->firstPtr;

   while (scanCount <= root->numEntries)
     {
      // Test for subtree
      if (ptr)
          _traverse  (ptr, process);

      // Subtree processed -- get next entry
      if (scanCount < root->numEntries)
```

continued

PROGRAM 10-5 Internal Traverse Function *(continued)*

```
29          {
30           process  (root->entries[scanCount].dataPtr);
31           ptr     = root->entries[scanCount].rightPtr;
32          } // if scanCount
33       scanCount++;
34     } // if
35   return;
36 }  // _traverse
```

B-tree Insert

The insertion logic requires several algorithms. We develop all but two of them in this section. Insertion requires a search that returns the index entry of the matching node. Because the logic is the same as we saw for Program 10-3, "Internal Search Function," we do not develop it here. The second function we do not develop inserts an entry into the entry array.

The implementation closely parallels the design shown in Algorithm 10-1, "B-tree Insert." The major difference is that the root is not returned—it is hidden in the ADT structure. The ADT interface function is shown in Program 10-6.

PROGRAM 10-6 B-tree Insert Application Interface

```
/* ================== BTree_Insert ==================
   This function inserts new data into the tree.
      Pre  tree is pointer to valid B-Tree structure
      Post data inserted or abort if memory O/F
*/
void BTree_Insert (BTREE* tree, void* dataInPtr)
{
// Local Definitions
   bool  taller;
   NODE* newPtr;
   ENTRY upEntry;

// Statements
   if (tree->root == NULL)
       // Empty Tree. Insert first node
       if (newPtr = (NODE*)malloc(sizeof (NODE)))
          {
           newPtr->firstPtr            = NULL;
           newPtr->numEntries          = 1;
           newPtr->entries[0].dataPtr  = dataInPtr;
           newPtr->entries[0].rightPtr = NULL;
           tree->root                  = newPtr;
           (tree->count)++;
```

continued

PROGRAM 10-6 B-tree Insert Application Interface *(continued)*

```
24
25              for (int i = 1; i < ORDER - 1; i++)
26                  {
27                   newPtr->entries[i].dataPtr  = NULL;
28                   newPtr->entries[i].rightPtr = NULL;
29                  } // for *
30              return;
31             } // if malloc
32           else
33              printf("Error 100 in BTree_Insert\a\n"),
34                      exit (100);
35
36       taller = _insert (tree,       tree->root,
37                         dataInPtr, &upEntry);
38       if (taller)
39          {
40           // Tree has grown. Create new root
41           newPtr = (NODE*)malloc(sizeof(NODE));
42           if (newPtr)
43              {
44               newPtr->entries[0]    = upEntry;
45               newPtr->firstPtr      = tree->root;
46               newPtr->numEntries    = 1;
47               tree->root            = newPtr;
48              } // if newPtr
49           else
50              printf("Overflow error 101\a\n"),
51                      exit (100);
52          } // if taller
53
54       (tree->count)++;
55       return;
56    }  // BTree_Insert
```

Program 10-6 Analysis

The insert ADT interface is more complex than those we have seen so far. It performs four different processes. First, it handles the insert into a null tree. This code is shown in statements 14 through 34.

The second process involves inserting all nodes after the first. It consists of a single call (statement 36) to the internal insert function. This code is identical in the pseudocode and in the ADT.

Once the new node has been inserted, the insert interface ADT must determine if a new root needs to be created. The internal insert function returns a Boolean indicating whether the height of the tree has grown. If it has, the new root is built in statements 38 through 52. Again, the code parallels the pseudocode.

Finally, if the insert was successful, the B-tree count must be updated. If other metadata, such as the maximum number of elements ever held in the B-tree, were necessary, they would also be handled here. This logic is not required in the pseudocode.

Internal Insert Function

The insert design is shown in Figure 10-6, "B-tree Insert Design." You might want to take a minute to review it before studying the functions in this section. Although the functions are rather long, they closely follow the design in Algorithm 10-2, "B-tree Insert Node," and should be easy to follow. The code for the internal insert function is shown in Program 10-7.

PROGRAM 10-7 Internal Insert Function

```
/* ==================== _insert ==================
   This function uses recursion to insert the new data
   into a leaf node in the B-Tree.
      Pre    Application has called BTree_Insert,
                which passes root and data pointers
      Post   Data have been inserted
      Return taller boolean
*/
bool _insert (BTREE* tree,      NODE*  root,
              void*  dataInPtr, ENTRY* upEntry)
{
// Local Declarations
   int  compResult;
   int  entryNdx;
   bool taller;

   NODE* subtreePtr;

// Statements
   if (!root)
      {
       // Leaf found -- build new entry
       (*upEntry).dataPtr  = dataInPtr;
       (*upEntry).rightPtr = NULL;
       return true;                    // tree taller
      } // if NULL tree

   entryNdx   = _searchNode (tree, root, dataInPtr);
   compResult = tree->compare(dataInPtr,
                    root->entries[entryNdx].dataPtr);
   if (entryNdx <= 0 && compResult < 0)
       // in node's first subtree
       subtreePtr = root->firstPtr;
   else
       // in entry's right subtree
```

continued

PROGRAM 10-7 Internal Insert Function *(continued)*

```
36          subtreePtr = root->entries[entryNdx].rightPtr;
37       taller = _insert (tree,      subtreePtr,
38                         dataInPtr, upEntry);
39
40       // Entry inserted -- back out of tree
41       if (taller)
42          {
43           if (root->numEntries >= ORDER - 1)
44              {
45               // Need to create new node
46               _splitNode (root,       entryNdx,
47                           compResult, upEntry);
48               taller = true;
49              } // node full
50           else
51              {
52               if (compResult >= 0)
53                 // New data >= current entry -- insert after
54                  _insertEntry(root, entryNdx + 1, *upEntry);
55               else
56                  // Insert before current entry
57                   _insertEntry(root, entryNdx, *upEntry);
58               (root->numEntries)++;
59               taller = false;
60              } // else
61          } // if taller
62
63       return taller;
64    }  // _insert
```

Program 10-7 Analysis

There are four major parts to the function.

(1) If the root is null, which is a base case, we have found the leaf node for the new data. At this point we build the new entry but do not insert it.

(2) If we are not at a leaf, we search the node for the entry that contains the target. If the entry is 0, the subtree may be either the node's first pointer's subtree or the entry's left subtree. After determining which, we recursively call insert node.

(3) After returning from the base case, we determine if the tree is taller or if there is room in the current node for the new entry. If the tree is taller, we must split the node.

(4) If the node is not taller, we insert the new entry either before or after the current entry, depending on their relationship.

Internal Split Node Function

The code for splitting a node is shown in Program 10-8.

PROGRAM 10-8 Internal Split Node Function

```
/* ================== _splitNode ==================
   Splits node when node is full.
      Pre  node has overflowed--Split node
           entryNdx is index location for new data
           compResult new data < or > entryNdx key
           upEntry pointer to data to be inserted
      Post node split and upEntry contains entry to
              be inserted into parent
       or- Program aborted if memory overflow
*/
void _splitNode  (NODE* node,       int    entryNdx,
                  int   compResult, ENTRY* upEntry)

{
// Local Definitions
   int   fromNdx;
   int   toNdx;
   NODE* rightPtr;

// Statements
   rightPtr  = (NODE*)malloc(sizeof (NODE));
   if (!rightPtr)
      printf("Overflow Error 101 in _splitNode\a\n"),
              exit (100);

   // Build right subtree node
   if (entryNdx < MIN_ENTRIES)
       fromNdx  = MIN_ENTRIES;
   else
       fromNdx  = MIN_ENTRIES + 1;
   toNdx = 0;
   rightPtr->numEntries = node->numEntries - fromNdx;
   while (fromNdx < node->numEntries)
       rightPtr->entries[toNdx++]
                    = node->entries[fromNdx++];
   node->numEntries = node->numEntries
                      - rightPtr->numEntries;

   // Insert new entry
   if (entryNdx < MIN_ENTRIES)
      {
       if (compResult < 0)
           _insertEntry (node, entryNdx, *upEntry);
       else
           _insertEntry (node, entryNdx + 1, *upEntry);
      } // if
```

continued

PROGRAM 10-8 Internal Split Node Function *(continued)*

```
47    else
48       {
49        _insertEntry (rightPtr,
50                      entryNdx - MIN_ENTRIES,
51                      *upEntry);
52        (rightPtr->numEntries)++;
53        (node->numEntries)--;
54       } // else
55
56    upEntry->dataPtr=node->entries[MIN_ENTRIES].dataPtr;
57    upEntry->rightPtr  = rightPtr;
58    rightPtr->firstPtr
59                  = node->entries[MIN_ENTRIES].rightPtr;
60
61    return;
62 }  // _splitNode
```

Program 10-8 Analysis

There are several interesting logical elements to this algorithm that take some thought as you study them. For example, when we insert **upEntry** into the right node (statements 49 to 53), we must add one to the node's number of entries, but when we insert it on the left (statements 42 to 45), we don't. The reason is that the left node contains the median entry to be inserted up into the parent, and therefore we don't need to add one. If we did, we would just have to subtract it after we deleted the median entry.

In a similar vein, when we insert an entry into the left node, we need to know if the new data key is less than the entry key. If it is, we pass the entry index to insert entry; if it is not, we pass the entry index plus one. To fully understand the reason for this difference, you need to construct two examples and follow the insertion logic carefully.

B-tree Delete

Deletion from a B-tree is potentially much more work than insertion. In this section we describe five of the seven functions required by delete. As we have seen several times, the delete requires both an ADT interface function and an internal recursive function. The delete interface function, however, is quite a bit simpler than the one we saw for the insert. Its code is shown in Program 10-9.

PROGRAM 10-9 B-tree Delete Application Interface

```
1  /* ================= BTree_Delete ====================
2     Delete entry with key target from B-Tree
3        Pre    tree must be initialized. Null tree OK
4               dltKey is pointer to key to be deleted
5        Post   Node entry deleted & data space freed
6               -or- An error code is returned
```

continued

PROGRAM 10-9 B-tree Delete Application Interface *(continued)*

```
 7        Return Success (true) or Not found (false)
 8  */
 9  bool BTree_Delete (BTREE* tree, void* dltKey)
10  {
11  // Local Definitions
12     bool  success;
13     NODE* dltPtr;
14
15  // Statements
16     if (!tree->root)
17         return false;
18
19     _delete (tree,
20              tree->root,
21              dltKey,
22             &success);
23
24     if (success)
25        {
26         (tree->count)--;
27         if (tree->root->numEntries == 0)
28            {
29             dltPtr     = tree->root;
30             tree->root = tree->root->firstPtr;
31             free (dltPtr);
32            } // root empty
33        } // success
34     return success;
35  }  // BTree_Delete
```

Program 10-9 Analysis Whereas the recursive delete function uses an underflow Boolean to determine when a node is below the minimum, the root has no minimum. Therefore, in this function our only concern is that all entries in the root may have been deleted. We handle this situation in statements 27 to 31.

Internal Delete Function

The internal delete function searches the B-tree for the entry to be deleted. If it is found in a leaf node, the delete is simple. If it is found in an internal node, however, a substitute in a leaf must be found. As we explained earlier, we search the left subtree for the immediate predecessor of the entry to be deleted. Once the substitution has been made, we can then delete the entry we know to be in the leaf. The pseudocode design is shown in Algorithm 10-6. The code is shown in Program 10-10.

PROGRAM 10-10 Internal Delete Function

```
/* ==================== _delete ====================
   Delete entry with key dltKey from B-Tree
      Pre    tree must be initialized--Null tree OK
             root is pointer to tree or subtree
             dltKey is key of entry to be deleted
             success indicates entry deleted or failed
      Post   node is deleted and its space recycled
             -or- if key not found, tree is unchanged.
             success is true / false
      Return underflow true / false
*/
bool _delete (BTREE* tree,        NODE* root,
              void*  dltKeyPtr,   bool* success)
{
// Local Definitions
   NODE* leftPtr;
   NODE* subTreePtr;
   int   entryNdx;
   int   underflow;

// Statements
   if (!root)
      {
       *success = false;
       return     false;
      } // null tree

   entryNdx  = _searchNode (tree, root, dltKeyPtr);
   if (tree->compare(dltKeyPtr,
       root->entries[entryNdx].dataPtr) == 0)
       {
        // found entry to be deleted
        *success = true;
        if (root->entries[entryNdx].rightPtr == NULL)
            // entry is a leaf node
            underflow = _deleteEntry (root, entryNdx);
        else
            // entry is in an internal node
            {
             if (entryNdx > 0)
                leftPtr =
                  root->entries[entryNdx - 1].rightPtr;
             else
                leftPtr = root->firstPtr;
             underflow = _deleteMid
                  (root, entryNdx, leftPtr);
```

continued

PROGRAM 10-10 Internal Delete Function *(continued)*

```
47                if (underflow)
48                   underflow = _reFlow (root, entryNdx);
49               } // else internal node
50           } // else found entry
51       else
52           {
53            if (tree->compare (dltKeyPtr,
54                   root->entries[0].dataPtr) < 0)
55               // delete key < first entry
56               subTreePtr = root->firstPtr;
57            else
58               // delete key in right subtree
59              subTreePtr = root->entries[entryNdx].rightPtr;
60
61            underflow = _delete (tree,      subTreePtr,
62                                 dltKeyPtr, success);
63            if (underflow)
64               underflow = _reFlow (root, entryNdx);
65           } // else not found *
66
67       return underflow;
68   }  // _delete
```

Internal Delete Middle Function

Called by the internal delete function, delete middle (Program 10-11) locates the predecessor node on the left subtree and substitutes it for the deleted node in the root.

PROGRAM 10-11 Internal Delete Middle Function

```
 1  /* ================== _deleteMid ==================
 2     Deletes entry from internal node in B-Tree
 3        Pre    Tree initialized--null tree OK
 4               node points to node data to be deleted
 5               subtreePtr is pointer to root's subtree
 6               entryNdx is entry to be deleted
 7        Post   predecessor's data replaces delete data
 8               predecessor deleted from tree
 9        Return underflow true / false
10  */
11  bool _deleteMid   (NODE* root,
12                     int   entryNdx,
13                     NODE* subtreePtr)
14  {
15  // Local Definitions
16     int  dltNdx;
```

continued

PROGRAM 10-11 Internal Delete Middle Function *(continued)*

```
17    int  rightNdx;
18    bool underflow;
19
20 // Statements
21    if (subtreePtr->firstPtr == NULL)
22        {
23        // leaf located. Exchange data & delete leaf
24         dltNdx = subtreePtr->numEntries - 1;
25         root->entries[entryNdx].dataPtr =
26               subtreePtr->entries[dltNdx].dataPtr;
27         --subtreePtr->numEntries;
28        underflow  = subtreePtr->numEntries < MIN_ENTRIES;
29        } // if leaf
30    else
31       {
32        // Not located. Traverse right for predecessor
33        rightNdx  = subtreePtr->numEntries - 1;
34        underflow = _deleteMid (root, entryNdx,
35               subtreePtr->entries[rightNdx].rightPtr);
36        if (underflow)
37           underflow = _reFlow (subtreePtr, rightNdx);
38       } // else traverse right
39    return underflow;
40 }  // _deleteMid
```

Program 10-11 Analysis Delete middle contains one of the more sophisticated pieces of logic in the B-tree ADT. Study the logic in statements 34 through 38 carefully. We recursively call delete middle until we find a leaf. When we return we know that we have deleted the substitute entry. Now we must back out of the tree, testing at each node to see if we have underflowed. If we have, we must reflow the nodes and then proceed up the tree until we reach the node that originally contained the data to be deleted. At that point we return to delete, which also continues backing out of the tree until it reaches the root.

Internal Reflow Function

When an underflow occurs, we must reflow the nodes to make sure they are valid. Reflow first tries to balance by borrowing from the right, then it tries to borrow from the left; if both fail, it combines the siblings. The code is shown in Program 10-12.

PROGRAM 10-12 Internal Reflow Function

```
1  /* ===================== _reFlow ====================
2     An underflow has occurred in a subtree to root.
3     Correct by balancing or concatenating.
4        Pre    root is pointer to underflow tree/subtree
5               entryNdx is parent of underflow subtree
```

continued

PROGRAM 10-12 Internal Reflow Function *(continued)*

```
 6        Post   Underflow corrected
 7        Return underflow true / false
 8  */
 9  bool _reFlow  (NODE* root, int entryNdx)
10  {
11  // Local Definitions
12     NODE* leftTreePtr;
13     NODE* rightTreePtr;
14     bool  underflow;
15
16  // Statements
17     if (entryNdx == 0)
18        leftTreePtr = root->firstPtr;
19     else
20       leftTreePtr = root->entries[entryNdx - 1].rightPtr;
21     rightTreePtr   = root->entries[entryNdx].rightPtr;
22
23     // Try to borrow first
24     if (rightTreePtr->numEntries > MIN_ENTRIES)
25           {
26            _borrowRight (root,        entryNdx,
27                          leftTreePtr, rightTreePtr);
28            underflow   = false;
29           } // if borrow right
30     else
31        {
32         // Can't borrow from right--try left
33         if (leftTreePtr->numEntries > MIN_ENTRIES)
34            {
35             _borrowLeft (root,        entryNdx,
36                          leftTreePtr, rightTreePtr);
37             underflow  = false;
38            } // if borrow left *
39        else
40            {
41             // Can't borrow. Must combine nodes.
42            _combine (root,        entryNdx,
43                      leftTreePtr, rightTreePtr);
44            underflow = (root->numEntries < MIN_ENTRIES);
45            } // else combine
46        } // else borrow right
47     return underflow;
48  }  // _reFlow
```

Internal Borrow Left or Right

When we underflow we first try to balance the tree by borrowing a node from the right sibling. If that doesn't work, we try to borrow from the left. These

two functions are mirror logic; they are identical except for the direction. The logic for borrow right is shown in Program 10-13.

PROGRAM 10-13 Internal Borrow Right Function

```
/* ================= _borrowRight ================
   Root left subtree underflow. Borrow from right
   and rotate.
      Pre    root is parent node to underflow node
             entryNdx is parent entry
             leftTreePtr is underflowed subtree
             rightTreePtr is subtree w/ extra entry
      Post   Underflow corrected
*/
void _borrowRight (NODE* root,
                   int   entryNdx,
                   NODE* leftTreePtr,
                   NODE* rightTreePtr)
{
// Local Definitions
   int toNdx;
   int shifter;

// Statements
   // Move parent and subtree pointer to left tree
   toNdx = leftTreePtr->numEntries;
   leftTreePtr->entries[toNdx].dataPtr
         = root->entries[entryNdx].dataPtr;
   leftTreePtr->entries[toNdx].rightPtr
         = rightTreePtr->firstPtr;
   ++leftTreePtr->numEntries;

   // Move right data to parent
   root->entries[entryNdx].dataPtr
         = rightTreePtr->entries[0].dataPtr;

   // Set right tree first pointer. Shift entries left
   rightTreePtr->firstPtr
         = rightTreePtr->entries[0].rightPtr;
   shifter = 0;
   while (shifter < rightTreePtr->numEntries - 1)
      {
       rightTreePtr->entries[shifter]
             = rightTreePtr->entries[shifter + 1];
       ++shifter;
      } // while
   --rightTreePtr->numEntries;
   return;
}  // _borrowRight
```

Internal Combine Nodes Function

If we can't borrow a node from a sibling, we must combine two nodes. The logic to combine nodes is shown in Program 10-14.

PROGRAM 10-14 Internal Combine Nodes Function

```
/* ==================== _combine ====================
   Underflow cannot be corrected by borrowing.
   Combine two subtrees.
      Pre  root contains parent to underflow node
           entryNdx is parent entry
           leftTreePtr & rightTreePtr are subtree ptrs
      Post Parent & subtrees combined-right node freed
*/
void _combine (NODE* root,        int   entryNdx,
               NODE* leftTreePtr, NODE* rightTreePtr)
{
// Local Definitions
   int toNdx;
   int fromNdx;
   int shifter;

// Statements
   // Move parent & set its right pointer from right tree
   toNdx = leftTreePtr->numEntries;
   leftTreePtr->entries[toNdx].dataPtr
           = root->entries[entryNdx].dataPtr;
   leftTreePtr->entries[toNdx].rightPtr
           = rightTreePtr->firstPtr;
   ++leftTreePtr->numEntries;
   --root->numEntries;

   // move data from right tree to left tree
   fromNdx = 0;
   toNdx++;
   while (fromNdx < rightTreePtr->numEntries)
       leftTreePtr->entries[toNdx++]
            = rightTreePtr->entries[fromNdx++];
   leftTreePtr->numEntries += rightTreePtr->numEntries;
   free (rightTreePtr);

   // Now shift data in root to the left
   shifter = entryNdx;
   while (shifter < root->numEntries)
      {
     root->entries[shifter] =
              root->entries[shifter + 1];
       shifter++;
      } // while
   return;
}  // _combine
```

10.4 Simplified B-trees

Computer scientists have assigned unique names to two specialized B-trees: 2-3 trees and 2-3-4 trees. We discuss them briefly. Both are well suited to internal search trees.

2-3 Tree

The **2-3 tree** is a B-tree of order 3. It gets its name because each nonroot node has either two or three subtrees (the root may have zero, two, or three subtrees). Figure 10-18 contains two 2-3 trees. The first is complete; that is, it has the maximum number of entries for its height. The second has more than twice as many entries, but some of the entries are empty. Note also that subtree 94 has only two descendents.

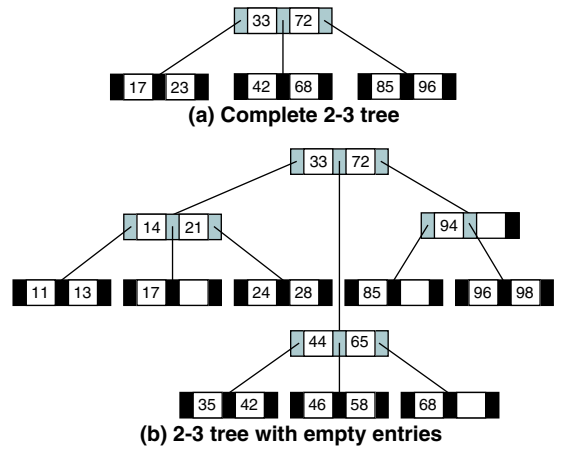


FIGURE 10-18 2-3 Trees

2-3-4 Tree

Figure 10-19 contains a B-tree of order 4. This type of tree is sometimes called a **2-3-4 tree** because each node can have two, three, or four children.

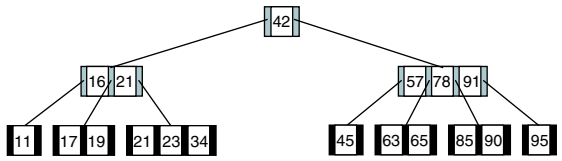


FIGURE 10-19 2-3-4 Tree

10.5 B-tree Variations

There are two popular variations on the B-tree. Although their manipulation is beyond the scope of this text, you should be aware of the variations.

B*tree

When we use a B-tree to store a large number of entries, the space requirements can become excessive because up to 50% of the entries can be empty. The first variation, the **B*tree**, addresses the space usage for large trees. Rather than each node containing a minimum of one-half the maximum entries, the minimum is set at two-thirds.[2]

In a B*tree, when a node overflows, instead of being split immediately, the data are redistributed among the node's siblings, delaying the creation of a new node. Splitting occurs only when all of the siblings are full. Furthermore, when the nodes are split, data from two full siblings are divided among the two full nodes and a new node, with the result that all three nodes are two-thirds full. Figure 10-20 shows how redistribution is handled when a node in a B*tree of order 5 overflows.

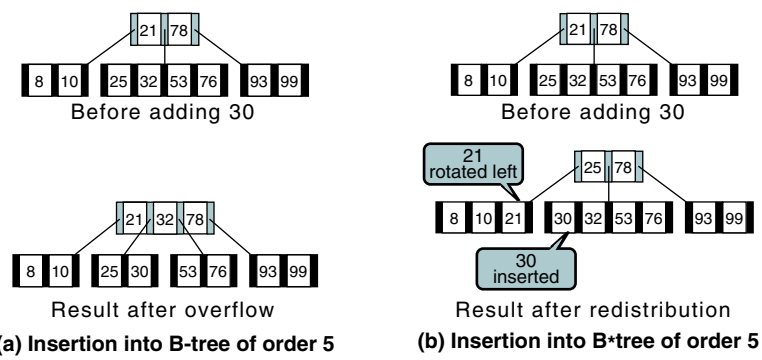


FIGURE 10-20 B*tree Insertion

B+tree

In large file systems, data need to be processed both randomly and sequentially. In these situations the most popular file organization methods use the B-tree to process the data randomly. However, much processing time is taken up moving up and down the tree structure when the data need to be processed sequentially. This inefficiency has led to the second B-tree variation, the **B+tree**.

2. Note that when the root in a root-only tree is split, its two subtrees are only half full.

There are two differences between the B-tree and the B+tree:

1. Each data entry must be represented at the leaf level, even though there may be internal nodes with the same keys. Because the internal nodes are used only for searching, they generally do not contain data.
2. Each leaf node has one additional pointer, which is used to move to the next leaf node in sequence. This structure is shown in Figure 10-21.

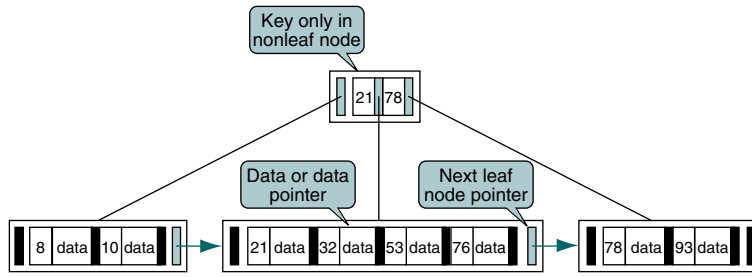


FIGURE 10-21 B+tree

When we process the data randomly, we modify the tree search to find the target data only at a leaf. The search time is thus increased slightly. However, to process the data sequentially, we simply locate the far-left entry and then process the data as though we were processing a linked list in which each node is an array.

10.6 Lexical Search Tree

Instead of searching a tree using the entire value of a key, we can consider the key to be a sequence of characters, such as a word or a nonnumeric identifier (e.g., a telephone number). When placed in a tree, each node has a place for each of the possible values that the characters in the lexical tree can assume. For example, if a key can contain the complete alphabet, each node has 26 entries, one for each of the letters of the alphabet. This is known as a lexical 26-ary tree.

Each entry in the **lexical search tree** contains a pointer to the next level. In addition, each node of a 26-ary tree contains 26 pointers, the first representing the letter *A*, the second the letter *B*, and so forth until the last pointer, which represents Z. Because each letter in the first level must point to a complete set of values, the second level contains 26×26 entries, one node of 26 entries for each of the 26 letters in the first level. Similarly, the third level has $26 \times 26 \times 26$ entries. Finally, we store the actual key at a leaf.

If a key has three letters, there are at least three levels in the tree. If a key has 10 letters, there are 10 levels in the tree. Because a lexical tree can

contain many different keys, the largest word determines the height of the tree. Figure 10-22 illustrates a lexical tree.

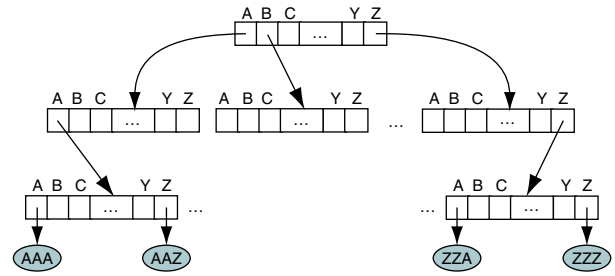


FIGURE 10-22 Lexical Tree Structure

Tries

The problem with the lexical *m*-ary tree is that after a few levels it becomes very large. To prevent this, we **prune** the tree; that is, we cut all of the branches that are not needed. We identify a pruned branch with a null pointer. For example, if no key starts with the letter *X*, then at level 0 the X pointer is null, thus eliminating its subtree completely. Similarly, after the letter Q, the only valid letter is *U* (we will not worry about the very few exceptions). All of the pointers in the Q branch except U are therefore set to null, again eliminating all but the U subtree from level 1 down. The resulting structure is called a **trie** (short for *reTRIEval* and pronounced "try").

As an example, let's create a spell checker using a trie. A spell checker is not a dictionary; it contains only words and not their definitions. Each word in our spell checker is one entry in the trie, the correct spelling of a word. To demonstrate the idea, let's build a small spell checker trie that checks only the spelling of a few words containing the letters *A*, *B*, *C*, *E*, and *T*.

Here we need a 5-ary trie because we have a total of five characters in these words (*A*, *B*, *C*, *E*, and *T*). The trie must have four levels because each word has at most three characters. Our trie is shown in Figure 10-23.

Although Figure 10-23 contains the complete spelling for each word, we have eliminated the third and fourth levels of the trie for the letter *A* because it can be fully identified by the first level under A. This is another form of pruning. Similarly, we have eliminated the third and fourth levels for *EAT* because it's the only word starting with *E*.

To search our spell checker for the word *CAB*, we check the first level of the trie at the letter C. Because it is not null, we follow the pointer to the second level, this time checking for the letter *A*. Again, there is a valid pointer at A, so we follow it to the node representing *CA* and check the B position. At this point we check the entry pointer and find CAB.

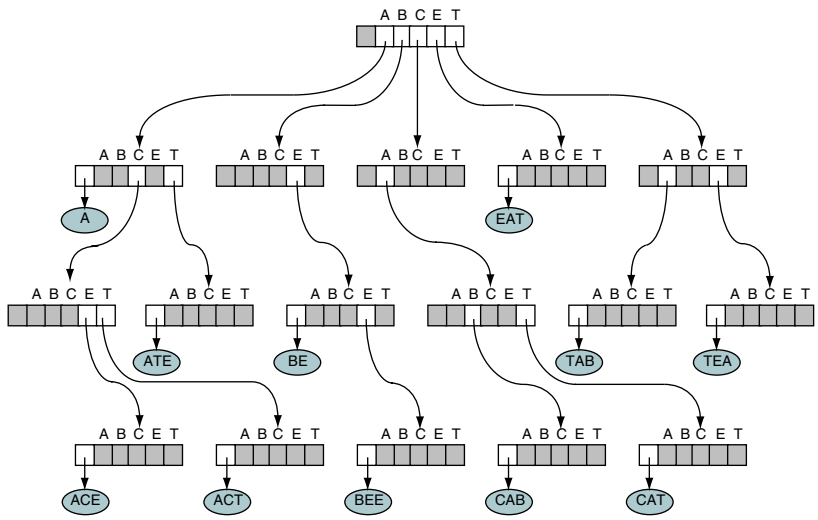


FIGURE 10-23 Spell Checker Trie

Now let's try to find *CTA* in our spell checker. We begin by checking the C position in the first node and find a valid pointer to the C node. However, when we check the T location in the second level, we find that the pointer is null. We then check the entry pointer, which is null, indicating that there is no valid word *CTA* in our spell checker.

Trie Structure

From the above discussion, we derive the structure for a trie. Each node needs two pointer types: one to the subtrees of the trie and one to the data. Each letter in the alphabet for the trie must be represented in each node. This gives us the node structure shown below.

```
trie
   entryPtr
   ltrPtrs
end trie
```

Trie Search

Let's write an algorithm to search our trie in Figure 10-23. We need two parameters: one for the dictionary trie and one for the word to be looked up. We use the trie structure shown above for the dictionary and a simple string for the word. The pseudocode is shown in Algorithm 10-15.

ALGORITHM 10-15 Trie Search

```
Algorithm searchTrie (dictionary, word)
Search the dictionary trie for word.
   Pre    dictionary is a valid trie with alphabet ABCET
   Return true if word in dictionary, false if not
1  set root   to dictionary
2  set ltrNdx to 0
3  loop (root not null)
   1   if (root entry equals word)
       1  return true
   2   end if
   3   if (ltrNdx >= word length)
       1  return false
   4   end if
   5   set chNdx to word[ltrNdx]
   6   set root  to chNdx subtree
   7   increment ltrNdx
4  end loop
5  return false
end searchTrie
```

Algorithm 10-15 Analysis

This algorithm's simplicity demonstrates the power of trees. As we search down the tree, we first compare the word associated with the trie node with our target word to see if we have found it. If not, we test to see if the dictionary word length at the current level, represented by the variable index **`ltrNdx`**, is greater than the word length. If it is, we know that the word is not in the dictionary.

The implementation of the code in statements 3.5 and 3.6 varies depending on the language. Most languages have a method of turning a letter into an index. Once the next letter in the word has been converted to an index (**`chNdx`**), we can use it to pick up the pointer to the next trie level.

10.7 Key Terms

2-3 tree
2-3-4 tree
balance
B*tree
B+tree
B-tree
B-tree order
combine
lexical search tree
m-way tree
overflow
prune
reflow
trie
underflow

10.8 Summary

- An *m*-way tree is a search tree in which:
  1. Each node has 0 to *m* subtrees.
  2. Given a node with $k < m$ subtrees, the node contains k subtree pointers, some of which may be null, and $k - 1$ data entries.
  3. The key values in the first subtree are all less than the key in the first entry; the key values in the other subtrees are all greater than or equal to the key in their parent entry.
  4. The keys of the data entries are ordered $key_1 <= key_2 <= \ldots <= key_k$.
  5. All subtrees are multiway trees.
- A B-tree is an *m*-way tree in which:
  1. The root is either a leaf or it has 2, 3, … *m* subtrees.
  2. All internal nodes have at least $\lceil m / 2 \rceil$ nonnull subtrees and at most *m* nonnull subtrees.
  3. All leaf nodes are at the same level; that is, the tree is perfectly balanced.
  4. A leaf node has at least $\lceil m / 2 \rceil - 1$ and at most $m - 1$ entries.
- B-tree insertion takes place at a leaf node. An insert to a full node creates a condition known as overflow. Overflow requires that the leaf node be split into two nodes, each containing half of the data.
- Three points must be considered when we delete an entry from a B-tree. First, we must ensure that the data to be deleted are actually in the tree. Second, if the node does not have enough entries after the deletion, we need to correct the structure deficiency (underflow). Third, we can delete only from a leaf node.
- Because a B-tree is a search tree, we use the inorder traversal to traverse the tree and visit each node in order.
- There are two special B-trees: 2-3 trees and 2-3-4 trees.

- The 2-3 tree is a B-tree of order 3. Each node except the root can have two or three subtrees.
- The 2-3-4 tree is a B-tree of order 4. Each node except the root can have two, three, or four subtrees.
- There are two popular variations of the B-tree: B*tree and B+tree.
- In a B*tree, each node contains a minimum of two-thirds of the maximum entries allowed for each node.
- In a B+tree, data entries are found only at the leaf level, and each leaf node has an additional pointer that is used to connect to the next leaf.
- In the lexical *m*-ary tree, the key is represented as a sequence of characters. Each entry in the lexical tree contains a pointer to the next level.
- A trie is a lexical *m*-ary tree in which the pointers pointing to nonexisting characters are replaced by null pointers.

10.9 Practice Sets

Exercises

1. Calculate the maximum number of data entries in a:
   a. 3-way tree of height 3
   b. 4-way tree of height 5
   c. *m*-way tree of height *h*
2. Calculate the maximum number of data entries in a:
   a. B-tree of order 5 with a height of 3
   b. B-tree of order 5 with a height of 5
   c. B-tree of order 5 with a height of *h*
3. Draw the B-tree of order 3 created by inserting the following data arriving in sequence:

```
92 24 6 7 11 8 22 4 5 16 19 20 78
```

4. Draw the B-tree of order 4 created by inserting the following data arriving in sequence:

```
92 24 6 7 11 8 22 4 5 16 19 20 78
```

5. Draw two B-trees of order 3 created by inserting data arriving in sequence from the two sets shown below. Compare the two B-trees to determine whether the order of data creates different B-trees.

```
89 78 8 19 20 33 56 44
44 56 33 20 19 8 78 89
```

6. Draw two different B-trees of order 3 that can store seven entries.
7. Create a B*tree of order 5 for the following data arriving in sequence:

```
92 24 6 7 11 8 22 4 5 16 19 20 78
```

8. Create a B+tree of order 5 for the following data arriving in sequence:

```
92 24 6 7 11 8 22 4 5 16 19 20 78
```

9. Draw a trie made from all 3-bit binary numbers (000 to 111).
10. Using the B-tree of order 3 shown in Figure 10-24, add 50, 78, 101, and 232.

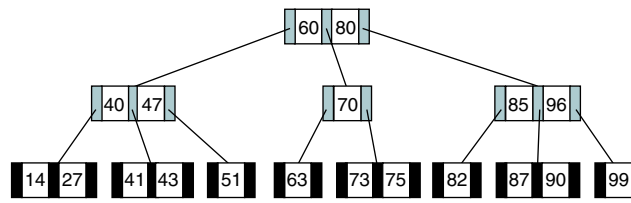


FIGURE 10-24 B-tree for Exercises 10 and 11

11. Using the B-tree of order 3 (without the updates in Exercise 10) shown in Figure 10-24, delete 63, 90, 41, and 60.

Problems

12. Rewrite the B-tree insertion algorithm using a stack instead of recursion.
13. Rewrite the B-tree deletion algorithm using a stack instead of recursion.
14. Write the search algorithm for a B+tree.
15. Write an algorithm that traverses a trie and prints all of its words in lexical order.
16. Write the C code for a B-tree count algorithm.
17. Write the C code for `_insertEntry`.
18. Write the C code for `_searchNode`.
19. Write the C code for `_borrowLeft`.
20. Write the C code for `_deleteEntry`.

Projects

21. Using the B-tree ADT, create a B-tree of order 7 that has 100 entries. Use a random-number generator to randomly create the keys between 1 and 1000. Then create a menu-driven user interface that allows the user to delete, insert, and retrieve data or to print the tree.

22. Create an ADT for a B+tree. In the tree structure, provide an additional metadata variable that identifies the address of the far-left node in the file. Then modify the traversal function to use the address of the far-left node and the next node pointers to traverse the tree.

23. The B-tree structure we studied can be used to create an indexed file. An indexed file contains an index structure to search for data in the file. Each entry in the index contains the data key from the file and the address of the data record in the file. The index can be created when the file is opened or it can be stored as a separate file on the disk.

    Write a program that uses the B-tree ADT to create a file index in dynamic memory. When the program starts, it reads the file and creates the B-tree index. After the index has been created, provide a menu-driven user interface that allows the user to retrieve a specified record, insert new records, delete records, and traverse the file, printing all of the data. You may use any appropriate application data, such as a collection of CDs or library books, for the file.

24. In Project 23 we created a B-tree index by reading the file. Rather than read the file each time the program starts, we could store the index as a separate file on the disk. In this case, when the program starts, the index is read and inserted into the B-tree. When the program is done, the updated B-tree index is written back to the disk. Modify the program from Project 23 to store the index on the disk.

25. The B-tree index project can be maintained on a disk rather than in dynamic memory. The first record on the index file should contain metadata about the index, including the location of the root entry in the file and the address of the compare function for the data. Modify the B-tree ADT to maintain the B-tree on a disk. In this version of the ADT, the create B-tree function is replaced by an open file function. The compare function must be defined when the file is opened.

26. As a final variation on the B-tree index project, rework Project 25 as a B+tree.

This page intentionally left blank

Chapter 11
Graphs

We have studied many different data structures. We started by looking at several data structures that deal with linear lists in which each node has a single successor. Then we looked at tree structures in which each node could have multiple successors but just one predecessor. In this last chapter, we turn our attention to a data structure—graphs—that differs from all of the others in one major concept: each node may have multiple predecessors as well as multiple successors.

Graphs are very useful structures. They can be used to solve complex routing problems, such as designing and routing airlines among the airports they serve. Similarly, they can be used to route messages over a computer network from one node to another.

11.1 Basic Concepts

A **graph** is a collection of nodes, called **vertices**, and a collection of segments, called **lines**, connecting pairs of vertices. In other words, a graph consists of two sets, a set of vertices and a set of lines.

Graphs may be either directed or undirected. A **directed graph**, or **digraph** for short, is a graph in which each line has a direction (arrow head) to its successor. The lines in a directed graph are known as **arcs**. In a directed graph, the flow along the arcs between two vertices can follow only the indicated direction. An **undirected graph** is a graph in which there is no direction (arrow head) on any of the lines, which are known as **edges**. In an undirected graph, the flow between two vertices can go in either direction. Figure 11-1 contains an example of both a directed graph (a) and an undirected graph (b).

A **path** is a sequence of vertices in which each vertex is adjacent to the next one. In Figure 11-1, {A, B, C, E} is one path and {A, B, E, F} is another. Note that both directed and undirected graphs have paths. In an undirected graph, you may travel in either direction.

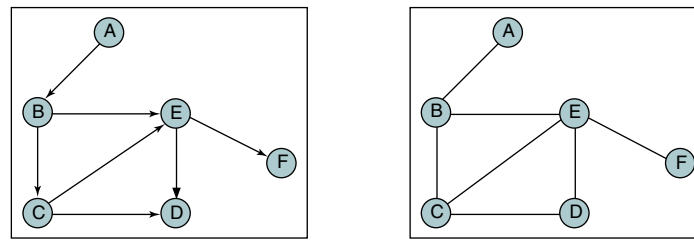


FIGURE 11-1 Directed and Undirected Graphs

Graphs may be directed or undirected. In a directed graph, each line, called an arc, has a direction indicating how it may be traversed. In an undirected graph, the line is known as an edge, and it may be traversed in either direction.

Two vertices in a graph are said to be **adjacent vertices** (or neighbors) if there is a path of length 1 connecting them. In Figure 11-1(a), B is adjacent to A, whereas E is not adjacent to D; on the other hand, D is adjacent to E. In Figure 11-1(b), E and D are adjacent, but D and F are not.

A graph is a collection of nodes, called vertices, and line segments, called arcs or edges, that connect pairs of nodes.

A **cycle** is a path consisting of at least three vertices that starts and ends with the same vertex. In Figure 11-1(b), B, C, D, E, B is a cycle. Note, however, that the same vertices in Figure 11-1(a) do not constitute a cycle because in a digraph a path can follow only the direction of the arc, whereas in an undirected graph a path can move in either direction along the edge. A **loop** is a special case of a cycle in which a single arc begins and ends with the same vertex. In a loop the end points of the line are the same.

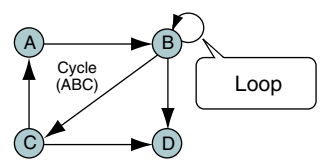


FIGURE 11-2 Cycles and Loops

Two vertices are said to be **connected** if there is a path between them. A graph is said to be connected if, ignoring direction, there is a path from any vertex to any other vertex. Furthermore, a directed graph is **strongly connected**

if there is a path from each vertex to every other vertex in the digraph. A directed graph is **weakly connected** if at least two vertices are not connected. (A connected undirected graph would always be strongly connected, so the concept is not normally used with undirected graphs.) A graph is a **disjoint graph** if it is not connected. Figure 11-3 contains a weakly connected graph (a), a strongly connected graph (b), and a disjoint graph (c).

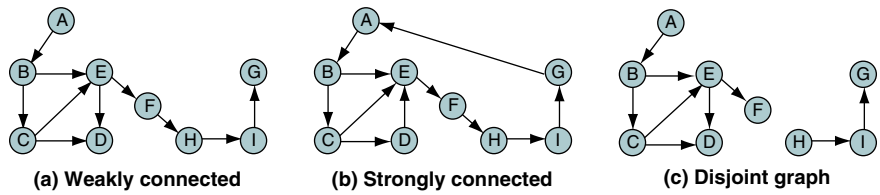


FIGURE 11-3 Connected and Disjoint Graphs

The **degree** of a vertex is the number of lines incident to it. In Figure 11-3(a) the degree of vertex B is 3 and the degree of vertex E is 4. The **outdegree** of a vertex in a digraph is the number of arcs leaving the vertex; the **indegree** is the number of arcs entering the vertex. Again, in Figure 11-3(a) the indegree of vertex B is 1 and its outdegree is 2; in Figure 11-3(b) the indegree of vertex E is 3 and its outdegree is 1.

One final point: a tree is a graph in which each vertex has only one predecessor; however, a graph is not a tree. We will see later in the chapter that some graphs have one or more trees in them that can be algorithmically determined.

11.2 Operations

In this section we define six primitive graph operations that provide the basic modules needed to maintain a graph: insert a vertex, delete a vertex, add an edge, delete an edge, find a vertex, and traverse a graph. As we will see, the graph traversal involves two different traversal methods.

Insert Vertex

Insert vertex adds a new vertex to a graph. When a vertex is inserted, it is disjoint; that is, it is not connected to any other vertices in the list. Obviously, inserting a vertex is just the first step in the insertion process. After a vertex is inserted, it must be connected. Figure 11-4 shows a graph before and after a new vertex is added.

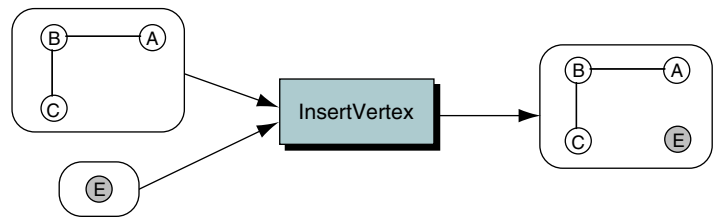


FIGURE 11-4 Insert Vertex

Delete Vertex

Delete vertex removes a vertex from the graph. When a vertex is deleted, all connecting edges are also removed. Figure 11-5 shows an example of deleting a vertex.

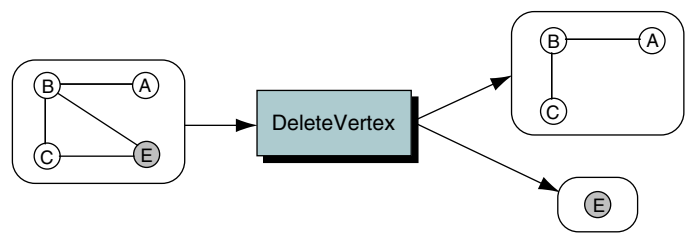


FIGURE 11-5 Delete Vertex

Add Edge

Add edge connects a vertex to a destination vertex. If a vertex requires multiple edges, add an edge must be called once for each adjacent vertex. To add an edge, two vertices must be specified. If the graph is a digraph, one of the vertices must be specified as the source and one as the destination. Figure 11-6 shows an example of adding an edge, {A, E}, to the graph.

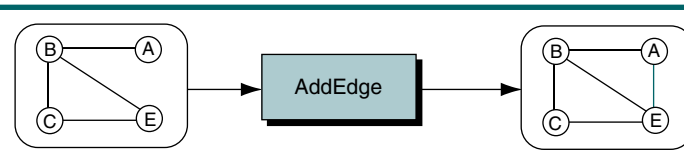


FIGURE 11-6 Add Edge

Delete Edge

Delete edge removes one edge from a graph. Figure 11-7 shows an example that deletes the edge {A, E} from the graph.

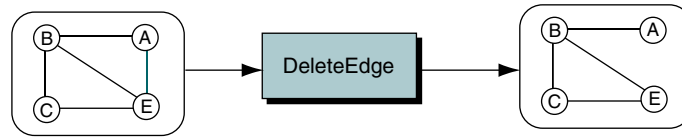


FIGURE 11-7 Delete Edge

Find Vertex

Find vertex traverses a graph, looking for a specified vertex. If the vertex is found, its data are returned. If it is not found, an error is indicated. In Figure 11-8 find vertex traverses the graph, looking for vertex C.

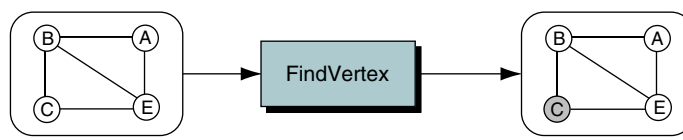


FIGURE 11-8 Find Vertex

Traverse Graph

There is always at least one application that requires that all vertices in a given graph be visited; that is, there is at least one application that requires that the graph be traversed. Because a vertex in a graph can have multiple parents, the traversal of a graph presents some problems not found in the traversal of linear lists and trees. Specifically, we must somehow ensure that we process the data in each vertex only once. However, because there are multiple paths to a vertex, we may arrive at it from more than one direction as we traverse the graph. The traditional solution to this problem is to include a visited flag at each vertex. Before the traversal we set the visited flag in each vertex to *off*. Then, as we traverse the graph, we set the visited flag to *on* to indicate that the data have been processed.

There are two standard graph traversals: depth first and breadth first. Both use the visited flag.

Depth-first Traversal

In the **depth-first traversal**, we process all of a vertex's descendents before we move to an adjacent vertex. This concept is most easily seen when the graph is a

tree. In Figure 11-9 we show the tree preorder traversal-processing sequence, one of the standard depth-first traversals.

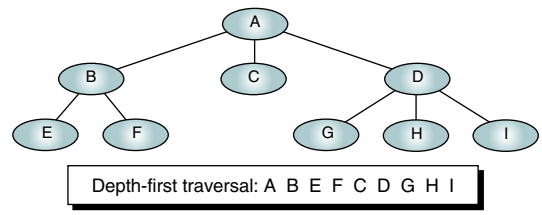


FIGURE 11-9 Depth-first Traversal of a Tree

In a similar manner, the depth-first traversal of a graph starts by processing the first vertex of the graph. After processing the first vertex, we select any vertex adjacent to the first vertex and process it. As we process each vertex, we select an adjacent vertex until we reach a vertex with no adjacent entries. This is similar to reaching a leaf in a tree. We then back out of the structure, processing adjacent vertices as we go. It should be obvious that this logic requires a stack (or recursion) to complete the traversal.

The order in which the adjacent vertices are processed depends on how the graph is physically stored. When we discuss the insertion logic later in the chapter, you will see that we insert the arcs in ascending key sequence. Because we are using a stack, however, the traversal processes adjacent vertices in descending, or last in–first out (LIFO), order.

> In the depth-first traversal, all of a node's descendents are processed before moving to an adjacent node.

Let's trace a depth-first traversal through the graph in Figure 11-10. The number in the box next to a vertex indicates the processing order. The stacks below the graph show the stack contents as we work our way down the graph and then as we back out.

1. We begin by pushing the first vertex, A, into the stack.
2. We then loop, pop the stack, and, after processing the vertex, push all of the adjacent vertices into the stack. To process X at step 2, therefore, we pop X from the stack, process it, and then push G and H into the stack, giving the stack contents for step 3 as shown in Figure 11-10(b)—H G.
3. When the stack is empty, the traversal is complete.

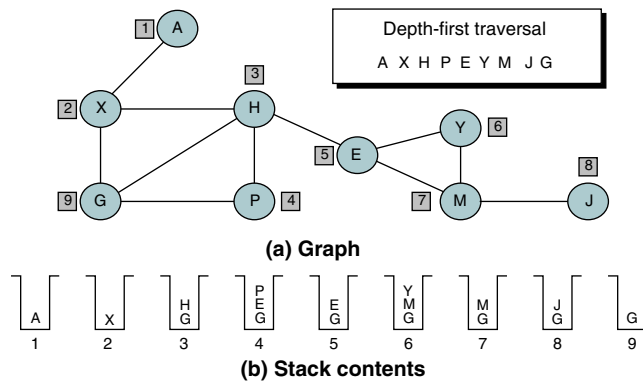


FIGURE 11-10 Depth-first Traversal of a Graph

Breadth-first Traversal

In the **breadth-first traversal** of a graph, we process all adjacent vertices of a vertex before going to the next level. We first saw the breadth-first traversal of a tree in Chapter 6. Looking at the tree in Figure 11-11, we see that its breadth-first traversal starts at level 0 and then processes all the vertices in level 1 before going on to process the vertices in level 2.

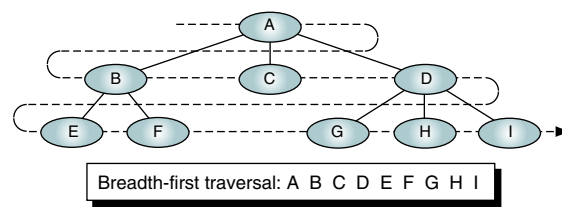


FIGURE 11-11 Breadth-first Traversal of a Tree

The breadth-first traversal of a graph follows the same concept. We begin by picking a starting vertex (A); after processing it we process all of its adjacent vertices (BCD). After we process all of the first vertex's adjacent vertices, we pick its first adjacent vertex (B) and process all of its vertices, then the second adjacent vertex (C) and all of its vertices, and so forth until we are finished.

In Chapter 6 we saw that the breadth-first traversal uses a queue rather than a stack. As we process each vertex, we place all of its adjacent vertices in the queue. Then, to select the next vertex to be processed, we delete a vertex from the queue and process it. Let's trace this logic through the graph in Figure 11-12.

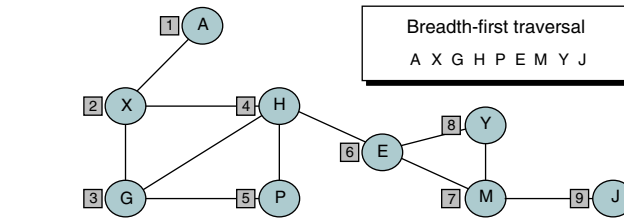


FIGURE 11-12 Breadth-first Traversal of a Graph

1. We begin by enqueuing vertex A in the queue.
2. We then loop, dequeuing the queue and processing the vertex from the front of the queue. After processing the vertex, we place all of its adjacent vertices into the queue. Thus, at step 2 in Figure 11-12(b), we dequeue vertex X, process it, and then place vertices G and H in the queue. We are then ready for step 3, in which we process vertex G.
3. When the queue is empty, the traversal is complete.

In the breadth-first traversal, all adjacent vertices are processed before processing the descendents of a vertex.

11.3 Graph Storage Structures

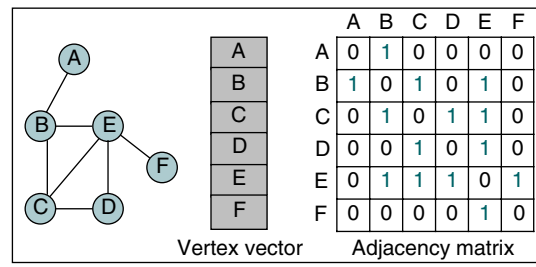
To represent a graph, we need to store two sets. The first set represents the vertices of the graph, and the second set represents the edges or arcs. The two most common structures used to store these sets are arrays and linked lists. Although the arrays offer some simplicity and processing efficiencies, the number of vertices must be known in advance. This is a major limitation.

Adjacency Matrix

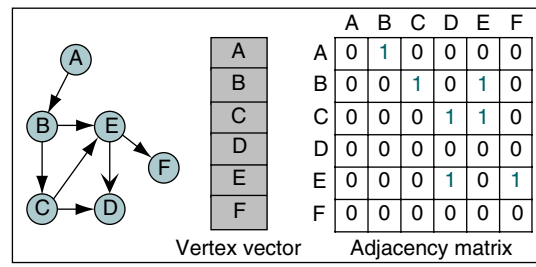
The **adjacency matrix** uses a vector (one-dimensional array) for the vertices and a matrix (two-dimensional array) to store the edges (see Figure 11-13). If two vertices are adjacent—that is, if there is an edge between them—the matrix intersect has a value of 1; if there is no edge between them, the intersect is set to 0.

If the graph is directed, the intersection in the adjacency matrix indicates the direction. For example, in Figure 11-13(b) there is an arc from source vertex B to destination vertex C. In the adjacency matrix, this arc is seen as a

1 in the intersection from B (on the left) to C (on the top). Because there is no arc from C to B, however, the intersection from C to B is 0. On the other hand, in Figure 11-13(a) the edge from B to C is bidirectional; that is, you can traverse it in either direction because the graph is nondirected. Thus, the nondirected adjacency matrix uses a 1 in the intersection from B to C as well as in the intersection from C to B. In other words, the matrix reflects the fact that you can use the edge to go either way.



(a) Adjacency matrix for nondirected graph



(b) Adjacency matrix for directed graph

FIGURE 11-13 Adjacency Matrix

In the adjacency matrix representation, we use a vector to store the vertices and a matrix to store the edges.

In addition to the limitation that the size of the graph must be known before the program starts, there is another serious limitation in the adjacency matrix: only one edge can be stored between any two vertices. Although this limitation does not prevent many graphs from using the matrix format, some network structures require multiple lines between vertices.

Adjacency List

The **adjacency list** uses a two-dimensional ragged array to store the edges. An adjacency list is shown in Figure 11-14.

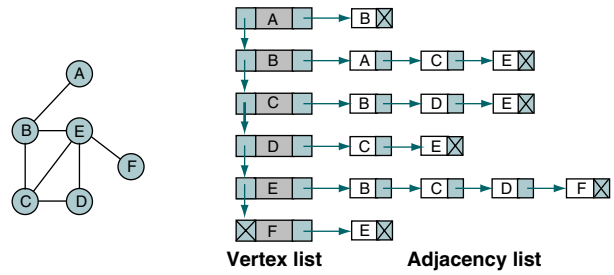


FIGURE 11-14 Adjacency List

The **vertex list** is a singly linked list of the vertices in the list. Depending on the application, it could also be implemented using doubly linked lists or circularly linked lists. The pointer at the left of the list links the vertex entries. The pointer at the right in the vertex is a head pointer to a linked list of edges from the vertex. Thus, in the nondirected graph on the left in Figure 11-14, there is a path from vertex B to vertices A, C, and E. To find these edges in the adjacency list, we start at B's vertex list entry and traverse the linked list to A, then to C, and finally to E.

In the adjacency list, we use a linked list to store the vertices and a two-dimensional linked list to store the arcs.

11.4 Graph Algorithms

In this section we develop a minimum set of algorithms that are needed to create and maintain a directed graph. In addition to the operations described in Section 11-2, "Operations," we include several others, such as create graph, that are required in programming graph applications. The nature of the application determines which other operations are required. For example, it may be necessary to write algorithms that return the vertex count or a vertex's indegree or outdegree.

Before we discuss the algorithms, we need to design the data structure we use for storing the graph. The most flexible structure is the adjacency list implemented as a singly linked list. In addition to the vertex and adjacency structures shown in Figure 11-14, we include a head structure. The head structure stores metadata about the list. For our algorithms we store only a count of the number of vertices in the graph. Examples of other metadata that can be stored include a rear pointer to the end of the vertex list and a count of the total arcs in the graph. The graph data structure is shown in Figure 11-15.

Note that the graph data, if any, are stored in the vertex node. These data pertain only to the vertex. Later we will see a structure that requires that we store data about an arc. In that case we store the arc data in the arc vertex. The pseudocode for the graph structure is shown in Algorithm 11-1.

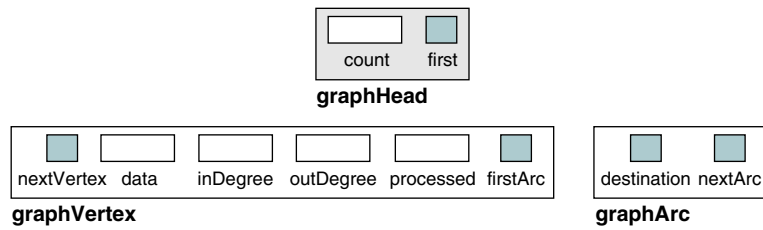


FIGURE 11-15 Graph Data Structure

ALGORITHM 11-1 Data Structure for Graph

```
graphHead
    count
    first
end graphHead

graphVertex
    nextVertex
    data
    inDegree
    outDegree
    processed
    firstArc
end graphVertex

graphArc
    destination
    nextArc
end graphArc
```

In addition to the data and pointers, each vertex entry contains a count of the number of arcs pointing to it—the indegree—and the number of arcs leaving it—the outdegree. The indegree count serves a very useful purpose. When we delete a vertex, we must ensure that there are no arcs pointing to it. If there are, any reference to the deleted vertex using the arc pointer causes the program to fail. Therefore, our delete algorithm does not allow a vertex to be deleted if any arcs are pointing to it.

Finally, we have a field, `processed`, that is used only for traversals. It indicates that the data in a vertex are waiting to be processed or have already been processed during the current traversal.

Create Graph

Create graph initializes the metadata elements for a graph head structure. The code is shown in Algorithm 11-2.

ALGORITHM 11-2 Create Graph

```
Algorithm createGraph
Initializes the metadata elements of a graph structure.
      Pre  graph is a reference to metadata structure
      Post metadata elements have been initialized
1  set count to 0
2  set first to null
3  return graph head
end createGraph
```

Insert Vertex

Insert vertex adds a disjoint—that is, an unconnected—vertex to the graph. The arcs associated with the vertex must be inserted separately. The pseudocode is shown in Algorithm 11-3.

ALGORITHM 11-3 Insert Vertex

```
Algorithm insertVertex (graph, dataIn)
Allocates memory for a new vertex and copies the data to it.
   Pre    graph is a reference to graph head structure
          dataIn contains data to be inserted into vertex
   Post   new vertex allocated and data copied
1  allocate memory for new vertex
2  store dataIn in new vertex
3  initialize metadata elements in newNode
4  increment graph count
   Now find insertion point
5  if (empty graph)
   1   set graph first to newNode
6  else
   1   search for insertion point
   2   if (inserting before first vertex)
       1   set graph first to new vertex
   3   else
       1   insert new vertex in sequence
7  end if
end insertVertex
```

Algorithm 11-3 Analysis

This is the basic singly linked list insertion situation. After allocating memory for the new vertex, we move in the data and set all of its metadata values to null.

Because the graph does not contain an integrated header structure, we must handle insertions into an empty graph and insertions before the first vertex as special cases. The insertion into a null list is handled in statement 5.1. Inserting before the first vertex is handled in statement 6.2.1.

Delete Vertex

Like any other delete algorithm, the first thing we have to do to delete a vertex is find it. Once we have found it, however, we also need to make sure that it is disjoint; that is, we need to ensure that there are no arcs leaving or entering the vertex. If there are, we reject the deletion. The pseudocode for delete vertex is shown in Algorithm 11-4.

ALGORITHM 11-4 Delete Vertex

```
Algorithm deleteVertex (graph, key)
Deletes an existing vertex only if its degree is 0.
   Pre    graph is reference pointer to graph head
          key is the key of the vertex to be deleted
   Post   vertex deleted (if degree zero)
   Return +1 if successful
          -1 if degree not zero
          -2 if key not found
 1 if (empty graph)
   1   return -2
 2 end if
   Locate vertex to be deleted
 3 search for vertex to be deleted
 4 if (not found)
   1   return -2
 5 end if
   Found vertex to be deleted. Test degree.
 6 if (vertex inDegree > 0 OR outDegree > 0)
   1   return -1
 7 end if
   Okay to delete vertex
 8 delete vertex
 9 decrement graph count
10 return 1
end deleteVertex
```

Algorithm 11-4 Analysis

This is a basic singly linked list delete situation. The only complexity is that we can't delete a vertex if its degree is greater than 0. This requirement is easily handled by testing the indegree and the outdegree in the vertex, as shown in statement 6.

Insert Arc

Once we have a vertex, we can connect it to other vertices. Insert arc requires two points in the graph: the source vertex (`fromPtr`) and the destination vertex (`toPtr`). Each vertex is identified by its key value rather than by its physical address. This system of identification gives us more flexibility in working with the graph and provides a degree of data structure hiding that makes it easier to implement the algorithms. The insertion logic is shown in Algorithm 11-5.

ALGORITHM 11-5 Insert Arc

```
Algorithm insertArc (graph, fromKey, toKey)
Adds an arc between two vertices.
   Pre    graph is reference to graph head structure
          fromKey is the key of the originating vertex
          toKey is the key of the destination vertex
   Post   arc added to adjacency list
   Return +1 if successful
          -2 if fromKey not found
          -3 if toKey not found
 1 allocate memory for new arc
   Locate source vertex
 2 search and set fromVertex
 3 if (from vertex not found)
   1   return -2
 4 end if
   Now locate to vertex
 5 search and set toVertex
 6 if (to vertex not found)
   1   return -3
 7 end if
   From and to vertices located. Insert new arc.
 8 increment fromVertex outDegree
 9 increment toVertex   inDegree
10 set arc destination to toVertex
11 if (fromVertex arc list empty)
       Inserting first arc
   1   set fromVertex firstArc to new arc
   2   set new arc nextArc to null
   3   return 1
12 end if
   Find insertion point in adjacency (arc) list
13 find insertion point in arc list
14 if (insert at beginning of arc list)
       Insertion before first arc
   1   set fromVertix firstArc to new arc
15 else
   1   insert in arc list
16 end if
17 return 1
end insertArc
```

Algorithm 11-5 Analysis

After allocating memory for the new arc, we locate the from vertex and the to vertex. This logic involves simple linked list searches. If either search fails, we return the appropriate value, −2 for from vertex not found and −3 for to vertex not found.

After we locate both vertices, we update their degree counts and set the new arc's destination pointer to the destination vertex. We are then ready to insert the new arc into the adjacency list. Our design for the adjacency list requires that the arc vertices be in sequence by destination key. We therefore search the arc list for the insertion point. Note that duplicate arcs are placed in the list in FIFO order. On the other hand, some

applications may not allow duplicate arcs. If they are not allowed, the logic needs to be changed and an additional error code created.

Delete Arc

The delete arc algorithm removes one arc from the adjacency list. To identify an arc, we need two vertices. The vertices are identified by their key. The algorithm therefore first searches the vertex list for the start vertex and then searches its adjacency list for the destination vertex. After locating and deleting the arc, the degree in the from and to vertices is adjusted and the memory recycled. The pseudocode is shown in Algorithm 11-6.

ALGORITHM 11-6 Delete Arc

```
Algorithm  deleteArc (graph, fromKey, toKey)
Deletes an arc between two vertices.
   Pre    graph is reference to a graph head structure
          fromKey is the key of the originating vertex
          toKey is the key of the destination vertex
   Post   vertex deleted
   Return +1 if successful
          -2 if fromKey not found
          -3 if toKey not found
 1 if (empty graph)
   1   return -2
 2 end if
   Locate source vertex
 3 search and set fromVertex to vertex with key equal to
      fromKey
 4 if (fromVertex not found)
   1   return -2
 5 end if
   Locate destination vertex in adjacency list
 6 if (fromVertex arc list null)
   1   return -3
 7 end if
 8 search and find arc with key equal to toKey
 9 if (toKey not found)
      1   return -3
10 end if
   fromVertex, toVertex, and arc all located. Delete arc.
11 set toVertex to arc destination
12 delete arc
13 decrement fromVertex outDegree
14 decrement toVertex   inDegree
15 return 1
end deleteArc
```

Algorithm 11-6 Analysis

There are three processes in this algorithm: (1) locate the source vertex, (2) locate the to vertex, and (3) delete the arc. The source vertex is located by searching the vertex list. Once we have located it, we search the adjacency list for an arc that points to the

destination vertex. Once we find the correct arc, we adjust the degree fields in the from and to vertex entries and then delete the arc.

Retrieve Vertex

Retrieve vertex returns the data stored in a vertex. Given the key of the vertex, the data are placed in the output area specified in the call. The pseudocode is shown in Algorithm 11-7.

ALGORITHM 11-7 Retrieve Vertex

```
Algorithm retrieveVertex (graph, key, dataOut)
Data contained in vertex identified by key passed to caller.
   Pre    graph is reference to a graph head structure
          key is the key of the vertex data
          dataOut is reference to data variable
   Post   vertex data copied to dataOut
   Return +1 if successful
          -2 if key not found
 1 if (empty graph)
   1  return -2
 2 end if
 3 search for vertex
 4 if (vertex found)
   1  move locnPtr data to dataOut
   2  return 1
 5 else
   1  return -2
 6 end if
end retrieveVertex
```

Algorithm 11-7 Analysis

Retrieve vertex is a typical linked list search algorithm. If the search is successful, the data in the vertex are placed in the output area specified in the calling sequence and success (+1) is returned. If the list is null or the data can't be located, key not located (–2) is returned.

Depth-first Traversal

The depth-first traversal was described in Section11.2, "Breadth-first Traversal." It visits all of the vertices in a graph by processing a vertex and all of its descendents before processing an adjacent vertex.

There are two ways to write the depth-first traversal algorithm: we can write it recursively or we can write it using a stack. We have decided to use the stack in this implementation. When we reach a vertex, we push its address into a stack. Then we pop the stack, process the vertex, and push all of its adjacent vertices into the stack.

The problem is that a vertex may be reached by more than one path through the graph. Consequently, we must ensure that each vertex is processed only once. When we created the structure for the vertex in Algorithm 11-1, we included a processed flag. When we begin the traversal,

we set the processed flag to 0. When we push a vertex into the stack, we set its processed flag to 1, indicating that it is in the stack, awaiting its turn. This prevents us from pushing it more than once. Finally, when we process the vertex, we set its flag to 2. Now, if we arrive at a vertex more than once, we know that we either pushed it or processed it earlier and not push or process it a second time. The traversal logic is shown in Algorithm 11-8.

ALGORITHM 11-8 Depth-first Traversal

```
Algorithm depthFirst (graph)
Process the keys of the graph in depth-first order.
   Pre  graph is a pointer to a graph head structure
   Post vertices "processed"
1   if (empty graph)
   1  return
   Set processed flags to not processed
2  set walkPtr to graph first
3  loop (through all vertices)
   1  set processed to 0
4  end loop
   Process each vertex in list
5  createStack (stack)
6  loop (through vertix list)
   1  if (vertix not processed and not in stack)
         Push and set flag to stack
      1  pushStack (stack, walkPtr)
      2  set walkPtr processed to 1
      3  end if
         Process vertex at stack top
      4  loop (not emptyStack(stack))
         1  set vertex to popStack(stack)
         2  process (vertix)
         3  set vertex to processed
            Push non-processed vertices from adjacency list
         4  loop (through arc list)
            1  if (arc destination not in stack)
               1  pushStack(stack, destination)
               2  set destination to in stack
            2  end if
            3  get next destination
         5  end loop
      5  end loop
   2  end if
   3  get next vertix
7  end loop
8  destroyStack(stack)
end depthFirst
```

Algorithm 11-8 Analysis

Depth-first traversal begins by setting all vertices to not processed. We then loop through the vertix list in statement 6. If we are guaranteed that the graph is strongly

connected, this loop is not necessary. However, if there is not a path from the first vertex to all other vertices, or if there are disjoint vertices, we need to ensure that all vertices have been processed by looping through the vertex list. That is the primary purpose of the loop in statement 6.

As a vertex is selected for processing, we first ensure that it has not been previously put into the sack or processed as a descendent of an earlier vertex. If not, we push it into the stack in statement 6.1.1. We then process the vertex at the top of the stack and push the nonprocessed adjacent vertices into the stack. Note that because we are using a stack to hold the vertices, we not only process them in depth-first order but also process the adjacent vertices in descending order. (Remember that we build the adjacency list in ascending key sequence, so adjacent vertices are pushed in ascending sequence and popped in descending sequence.)

When the stack is empty, we have processed all of the vertices strongly connected to the original vertex. We then advance to the next vertex and return to statement 6 to ensure that any weakly connected or disjoint vertices are processed.

Breadth-first Traversal

The breadth-first traversal, described in Section 11.2, processes a vertex and then processes all of its adjacent vertices. Whereas we used a stack for the depth-first traversal, we use a queue to traverse a graph breadth first. The pseudocode is shown in Algorithm 11-9.

ALGORITHM 11-9 Breadth-first Traversal

```
Algorithm breadthFirst (graph)
Processes the keys of the graph in breadth-first order.
   Pre  graph is pointer to graph head structure
   Post vertices processed
1  if (empty graph)
   1  return
2  end if
   First set all processed flags to not processed
3  createQueue (queue)
4  loop (through all vertices)
   1  set vertix to not processed
5  end loop
   Process each vertex in vertex list
6  loop (through all vertices)
   1  if (vertix not processed)
      1  if (vertix not in queue)
            Enqueue and set process flag to queued (1)
         1  enqueue (queue, walkPtr)
         2  set vertix to enqueued
      2  end if
         Now process descendents of vertex at queue front
      3  loop (not emptyQueue (queue))
         1  set vertex to dequeue (queue)
            Process vertex and flag as processed
```

continued

ALGORITHM 11-9 Breadth-first Traversal *(continued)*

```
         2  process (vertex)
         3  set vertix to processed
            Enqueue non-processed vertices from adjacency list
         4  loop (through adjacency list)
            1  if (destination not enqueued or processed)
               1  enqueue (queue, destination)
               2  set destination to enqueued
            2  end if
            3  get next destination
         5  end loop
      4  end loop
   2  end if
   3  get next vertix
7  end loop
8  destroyQueue (queue)
end breadthFirst
```

Algorithm 11-9 Analysis

The code in a breadth-first traversal begins as the depth-first algorithm does, by setting all of the processed flags to not processed. Note, however, that three process states are used in this algorithm: not processed, in queue, and processed. This prevents a vertex from being placed in the queue more than once. When it is processed, we set the flag to 2. After setting the processed flags to 0, we create a queue using the standard algorithms we developed in Chapter 4.

As we loop through the vertex list (statement 6), we first check whether the vertex has been processed (statement 6.1). If it has, we advance to the next vertex (statement 6.3) and loop back to check again. As we saw in the depth-first traversal, this loop allows us to pick up disjoint vertices or other vertices in a weakly connected graph.

If the vertex has not been processed, we test to see if it has been placed in the queue already (statement 6.1.1); if not, we enqueue it. We then process all descendents of the vertex at the front of the queue with a loop (statement 6.1.3) that dequeues and processes the vertex at the beginning of the queue, places all of its unprocessed adjacent vertices in the queue, and repeats the loop until the queue is empty.

After processing all of the descendents of the vertex at the front of the queue, we advance to the next vertex and loop back to statement 6 to complete the processing of the vertex list.

Destroy Graph

When we discussed deleting a vertex from a graph, we saw that the vertex must be disjoint before it could be deleted. This rule does not apply to destroying the graph. The logic for destroy graph must therefore first delete all arcs from the vertex before it can delete the vertex. We don't need to worry about arcs pointing to the vertex because they are deleted when their corresponding vertex is deleted. The design for destroy graph is shown in Algorithm 11-10.

ALGORITHM 11-10 Destroy Graph

```
Algorithm destroyGraph (graph)
Traverses graph deleting all vertices and arcs.
   Pre  Nothing
   Post All vertices and arcs deleted
1  if (empty graph)
   1   return
2  loop (more vertices)
   1   loop (vertex outDegree > 0)
       1   delete vertex firstArc
       2   subtract 1 from vertex outDegree
   2   end loop
3  end loop
end destroyGraph
```

11.5 Graph ADT

We are now ready to discuss the abstract data type for a graph. You will find this one very simple, especially when compared with the ADT for B-trees. It uses simple linked list concepts, with the stack and queue ADTs incorporated for the graph traversals. The ADT design is shown in Figure 11-16.

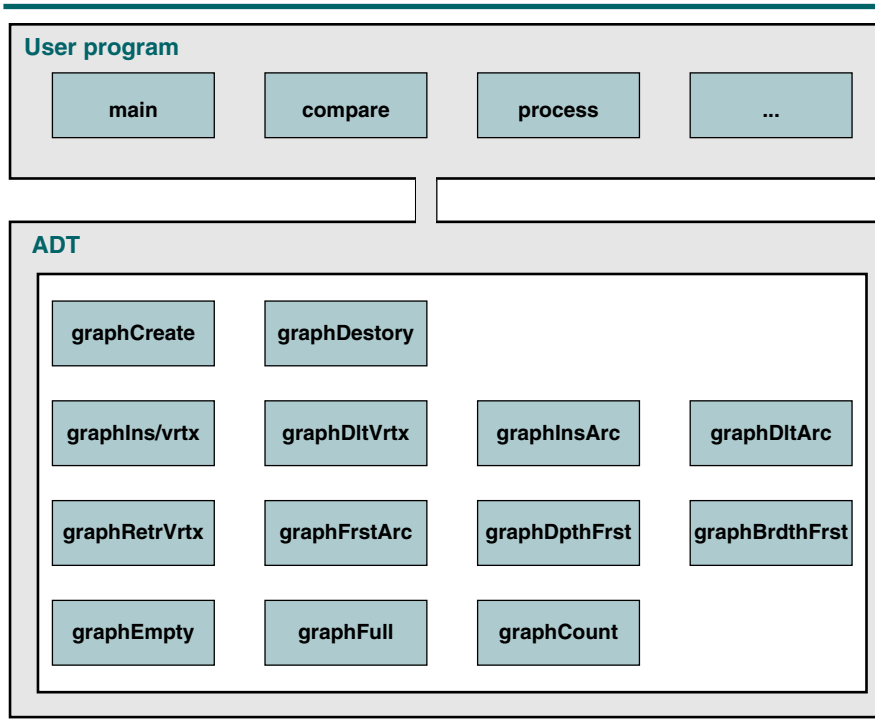


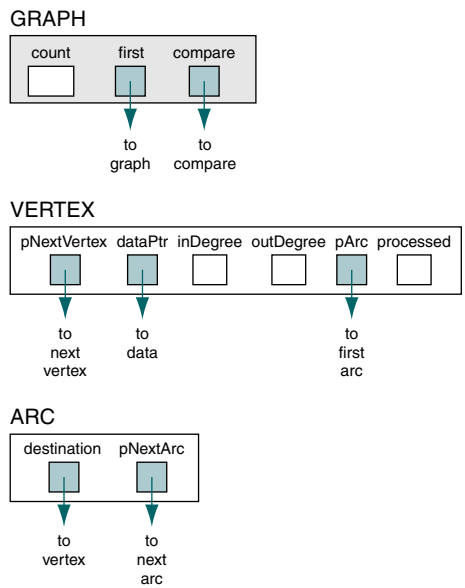
FIGURE 11-16 Graph ADT Design

Data Structure

As we have seen in other designs, a graph needs three separate data structures: one for the head, one for vertices, and one for arcs.

Head Structure

The graph data structure uses a simple head structure, `GRAPH`, that contains a count, a pointer to the first vertex, and the address of the compare function needed to search the graph. The application program's only view of the graph is a pointer to the head structure, which is allocated from dynamic memory when the graph is created. The head structure is shown in Figure 11-17.




```
typedef struct
   {
    int            count;
    struct vertex* first;
    int (*compare)
          (void* argu1,
           void* argu2);
   } GRAPH;

typedef struct vertex
 {
  struct vertex*  pNextVertex;
  void*           dataPtr;
  int             inDegree;
  int             outDegree;
  short           processed;
  struct arc*     pArc;
 } VERTEX;

typedef struct arc
 {
  struct vertex*  destination;
  struct arc*     pNextArc;
 } ARC;
```

FIGURE 11-17 Graph Data Structure

Vertex Structure

The graph vertex node stores data about the vertex. It contains a data pointer and a pointer to the next vertex in sequence, a pointer to the first arc from the vertex, and three metadata attributes—indegree, outdegree, and a processed flag. The vertex structure is also shown in Figure 11-17.

Arc Structure

The arc node stores data about the path from one vertex to another. It contains a pointer to the destination vertex and a pointer to the next arc in the path from the related vertex. It may also contain arc metadata, such as the weight of the arc. The arc structure is also shown in Figure 11-17.

Algorithms

The graph data structure is an adjacency list as described in Section 11.3, "Adjacency List." The data structures and the prototype declarations are declared in a graph header structure which we named `graph.h`. The header file is shown in Program 11-1.

PROGRAM 11-1 Graph Declaration

```
/* ==================== graphs.h ====================
   This header file contains the declarations and
   prototype functions for graphs as developed in
   this text.
      Written by:
      Date:
*/
#include "queueADT.h"
#include "stackADT.h"
#include "stdbool.h"

// ================== STRUCTURES =================
typedef struct
   {
    int            count;
    struct vertex* first;
    int (*compare) (void* argu1, void* argu2);
   } GRAPH;

typedef struct vertex
   {
    struct vertex*  pNextVertex;
    void*           dataPtr;
    int             inDegree;
    int             outDegree;
    short           processed;
    struct arc*     pArc;
   } VERTEX;

typedef struct arc
   {
    struct vertex*  destination;
    struct arc*     pNextArc;
   } ARC;

// ============ Prototype Declarations =============

GRAPH* graphCreate
          (int  (*compare) (void* argu1, void* argu2));
GRAPH* graphDestroy (GRAPH* graph);

```

continued

PROGRAM 11-1 Graph Declaration *(continued)*

```
42 void graphInsVrtx   (GRAPH* graph, void*  dataInPtr);
43 int graphDltVrtx    (GRAPH* graph, void*  dltKey);
44 int graphInsArc     (GRAPH* graph, void*  pFromKey,
45                                    void*  pToKey);
46 int graphDltArc     (GRAPH* graph, void*  pFromKey,
47                                    void*  pToKey);
48
49 int graphRetrVrtx   (GRAPH* graph, void*  pKey,
50                                    void** pDataOut);
51 int graphFrstArc    (GRAPH* graph, void*  pKey,
52                                    void** pDataOut);
53
54 void graphDpthFrst  (GRAPH* graph,
55                      void (*process) (void* dataPtr));
56 void graphBrdthFrst (GRAPH* graph,
57                      void (*process) (void* dataPtr));
58
59 bool graphEmpty (GRAPH* graph);
60 bool graphFull  (GRAPH* graph);
61 int  graphCount (GRAPH* graph);
```

We discuss most of the ADT functions in the following sections. Several of them,[1] however, are very simple and parallel functions developed for other ADTs. They are not developed here.

Graph Insert Vertex

In the building of a graph, the first process is to insert the vertices. This is done with insert vertex, as shown in Program 11-2.

PROGRAM 11-2 Graph Insert Vertex

```
 1 /*  ================= graphInsVrtx ==================
 2    This function inserts new data into the graph.
 3       Pre  graph is pointer to valid graph structure
 4       Post data inserted or abort if memory O/F
 5 */
 6 void graphInsVrtx (GRAPH* graph, void* dataInPtr)
 7 {
 8 // Local Definitions
 9    VERTEX* newPtr;
10    VERTEX* locPtr;
11    VERTEX* predPtr;
```

continued

1. Specifically: `graphCreate`, `graphEmpty`, `graphFull`, `graphCount`, `graphRetrVrtx`, `graphFrstArc`, and `graphDestroy` are not included.

PROGRAM 11-2 Graph Insert Vertex *(continued)*

```
12
13 // Statements
14    newPtr = (VERTEX*)malloc(sizeof (VERTEX));
15    if (newPtr)
16       {
17        newPtr->pNextVertex  = NULL;
18        newPtr->dataPtr      = dataInPtr;
19        newPtr->inDegree     = 0;
20        newPtr->outDegree    = 0;
21        newPtr->processed    = 0;
22        newPtr->pArc         = NULL;
23        (graph->count)++;
24       } // if malloc
25     else
26        printf("Overflow error 100\a\n"),
27           exit (100);
28
29    // Now find insertion point
30    locPtr = graph->first;
31    if (!locPtr)
32        // Empty graph. Insert at beginning
33        graph->first = newPtr;
34    else
35       {
36        predPtr = NULL;
37        while (locPtr && (graph->compare
38                      (dataInPtr, locPtr->dataPtr) > 0))
39           {
40            predPtr = locPtr;
41            locPtr  = locPtr->pNextVertex;
42           } // while
43        if (!predPtr)
44            // Insert before first vertex
45            graph->first = newPtr;
46        else
47            predPtr->pNextVertex = newPtr;
48        newPtr->pNextVertex = locPtr;
49       } // else
50    return;
51 }  // graphInsVrtx
```

Program 11-2 Analysis

Insert graph parallels the pseudocode found in Algorithm 11-3. The only significant change is that we allocated memory for the new vertex in statement 15 rather than later. This was done to facilitate the test for memory overflow at the beginning of the function.

Although it is not logically necessary to set the processed flag to 0 at this time (see statement 21), it was done for consistency. The rest of the logic is the function inserting the vertex into a singly linked list.

Graph Delete Vertex

Delete vertex deletes a vertex from a graph, provided that its degree is 0. If another vertex points to it, or if it points to another vertex, it cannot be deleted. Program 11-3 follows the design in Algorithm 11-4.

PROGRAM 11-3 Graph Delete Vertex

```
/* ================== graphDltVrtx =================
   Deletes existing vertex only if its degree is zero.
      Pre    graph is pointer to graph head structure
             dltKey is key of vertex to be deleted
      Post   Vertex deleted if degree zero
             -or- An error code is returned
      Return Success +1 if successful
                     -1 if degree not zero
                     -2 if dltKey not found
*/
int graphDltVrtx (GRAPH* graph, void* dltKey)
{
// Local Definitions
   VERTEX* predPtr;
   VERTEX* walkPtr;

// Statements
   if (!graph->first)
       return -2;

   // Locate vertex to be deleted
   predPtr = NULL;
   walkPtr = graph->first;
   while (walkPtr
     && (graph->compare(dltKey, walkPtr->dataPtr) > 0))
      {
       predPtr = walkPtr;
       walkPtr = walkPtr->pNextVertex;
      } // walkPtr &&
   if (!walkPtr
     || graph->compare(dltKey, walkPtr->dataPtr) != 0)
      return -2;

   // Found vertex. Test degree
   if ((walkPtr->inDegree > 0)
        || (walkPtr->outDegree > 0))
       return -1;

   // OK to delete
   if (!predPtr)
       graph->first = walkPtr->pNextVertex;
   else
```

continued

PROGRAM 11-3 Graph Delete Vertex *(continued)*

```
43          predPtr->pNextVertex = walkPtr->pNextVertex;
44       --graph->count;
45       free(walkPtr);
46       return 1;
47    }  // graphDltVrtx
```

Graph Insert Arc

This rather lengthy function is really quite simple. To insert an arc, we need to know the source vertex and the destination vertex. Much of the code locates the source and destination vertices. Once we are sure that they both exist, we simply insert a new arc in the source adjacency list in destination vertex sequence. The code is shown in Program 11-4, which follows the design in Algorithm 11-5.

PROGRAM 11-4 Graph Insert Arc

```
 1  /* ================== graphInsArc ==================
 2     Adds an arc vertex between two vertices.
 3        Pre    graph is a pointer to a graph
 4               fromKey is pointer to start vertex key
 5               toKey   is pointer to dest'n vertex key
 6        Post   Arc added to adjacency list
 7        Return success  +1 if successful
 8                        -1 if memory overflow
 9                        -2 if fromKey not found
10                        -3 if toKey not found
11  */
12  int graphInsArc (GRAPH* graph, void* pFromKey,
13                   void*  pToKey)
14  {
15  // Local Definitions
16     ARC*    newPtr;
17     ARC*    arcPredPtr;
18     ARC*    arcWalkPtr;
19     VERTEX* vertFromPtr;
20     VERTEX* vertToPtr;
21
22  // Statements
23     newPtr = (ARC*)malloc(sizeof(ARC));
24     if (!newPtr)
25        return (-1);
26
27     // Locate source vertex
28     vertFromPtr = graph->first;
29     while (vertFromPtr && (graph->compare(pFromKey,
30                            vertFromPtr->dataPtr) > 0))
```

continued

PROGRAM 11-4 Graph Insert Arc *(continued)*

```
31          {
32           vertFromPtr = vertFromPtr->pNextVertex;
33          } // while vertFromPtr &&
34       if (!vertFromPtr || (graph->compare(pFromKey,
35                            vertFromPtr->dataPtr) != 0))
36          return (-2);
37
38       // Now locate to vertex
39       vertToPtr   = graph->first;
40       while (vertToPtr
41           && graph->compare(pToKey, vertToPtr->dataPtr) > 0)
42          {
43           vertToPtr   = vertToPtr->pNextVertex;
44          } // while vertToPtr &&
45       if (!vertToPtr ||
46          (graph->compare(pToKey, vertToPtr->dataPtr) != 0))
47          return (-3);
48
49       // From and to vertices located. Insert new arc
50       ++vertFromPtr->outDegree;
51       ++vertToPtr  ->inDegree;
52       newPtr->destination = vertToPtr;
53       if (!vertFromPtr->pArc)
54          {
55           // Inserting first arc for this vertex
56           vertFromPtr->pArc = newPtr;
57           newPtr-> pNextArc = NULL;
58           return 1;
59          } // if new arc
60
61       // Find insertion point in adjacency (arc) list
62       arcPredPtr = NULL;
63       arcWalkPtr = vertFromPtr->pArc;
64       while (arcWalkPtr
65              && graph->compare(pToKey,
66                     arcWalkPtr->destination->dataPtr) >= 0)
67          {
68           arcPredPtr = arcWalkPtr;
69           arcWalkPtr = arcWalkPtr->pNextArc;
70          } // arcWalkPtr &&
71
72       if (!arcPredPtr)
73           // Insertion before first arc
74           vertFromPtr->pArc    = newPtr;
75       else
76           arcPredPtr->pNextArc = newPtr;
77       newPtr->pNextArc = arcWalkPtr;
78        return 1;
79    } // graphInsArc
```

Graph Delete Arc

This is another lengthy function for a relatively simple process. As with the insert arc function, we identify an arc by its source and destination vertices. Once we locate the source vertex and verify that there is an arc to its destination, we use a simple linked list deletion algorithm to delete the arc. The code, shown in Program 11-5, follows the design in Algorithm 11-6.

PROGRAM 11-5 Graph Delete Arc

```
/* ================== graphDltArc ==================
   Deletes an existing arc.
      Pre    graph is pointer to graph head structure
             fromKey is key of start vertex; toKey is
             toKey is key of dest'n of delete vertex
      Post   Arc deleted
      Return Success +1 if successful
                     -2 if fromKey not found
                     -3 if toKey not found
*/
int graphDltArc (GRAPH* graph,
                 void* fromKey, void* toKey)
{
// Local Definitions
   VERTEX* fromVertexPtr;
   VERTEX* toVertexPtr;
   ARC*    preArcPtr;
   ARC*    arcWalkPtr;

// Statements
   if (!graph->first)
       return -2;

   // Locate source vertex
   fromVertexPtr = graph->first;
   while (fromVertexPtr && (graph->compare(fromKey,
                       fromVertexPtr->dataPtr) > 0))
       fromVertexPtr = fromVertexPtr->pNextVertex;

   if (!fromVertexPtr || graph->compare(fromKey,
                       fromVertexPtr->dataPtr) != 0)
      return -2;

   // Locate destination vertex in adjacency list
   if (!fromVertexPtr->pArc)
       return -3;

   preArcPtr = NULL;
   arcWalkPtr = fromVertexPtr->pArc;
```

continued

PROGRAM 11-5 Graph Delete Arc *(continued)*

```
40      while (arcWalkPtr && (graph->compare(toKey,
41                arcWalkPtr->destination->dataPtr) > 0))
42         {
43          preArcPtr  = arcWalkPtr;
44          arcWalkPtr = arcWalkPtr->pNextArc;
45         } // while arcWalkPtr &&
46      if (!arcWalkPtr || (graph->compare(toKey,
47               arcWalkPtr->destination->dataPtr) != 0))
48          return -3;
49      toVertexPtr = arcWalkPtr->destination;
50
51      // from, toVertex & arcPtr located. Delete arc
52      --fromVertexPtr->outDegree;
53      --toVertexPtr -> inDegree;
54      if (!preArcPtr)
55          // Deleting first arc
56          fromVertexPtr->pArc  = arcWalkPtr->pNextArc;
57      else
58          preArcPtr->pNextArc = arcWalkPtr->pNextArc;
59      free (arcWalkPtr);
60      return 1;
61  }  // graphDltArc
```

Graph Depth-first Traversal

As you study the depth-first traversal, you may want to refer back to Figure 11-10, "Depth-first Traversal of a Graph." One of the interesting aspects of this algorithm is its use of the stack ADT. Recall that we need to completely process a vertex and all of its descendents from start to end before we process any adjacent vertices. This requires that we place a vertex into a stack and then pop the stack to process it. The design is shown in Algorithm 11-8, the code in Program 11-6.

PROGRAM 11-6 Graph Depth-first Traversal

```
 1  /* ================== graphDpthFrst =================
 2     Process data in graph in depth-first order.
 3        Pre  graph is the a pointer to graph head
 4        Post vertices "processed".
 5
 6        Processed Flag: 0 = not processed
 7                        1 = pushed into stack
 8                        2 = processed
 9  */
10  void graphDpthFrst (GRAPH* graph,
11                      void (*process) (void* dataPtr))
12  {
```

continued

PROGRAM 11-6 Graph Depth-first Traversal *(continued)*

```
 // Local Definitions
   bool    success;
   VERTEX* walkPtr;
   VERTEX* vertexPtr;
   VERTEX* vertToPtr;
   STACK * stack;
   ARC*    arcWalkPtr;

// Statements
   if (!graph->first)
       return;

   // Set processed flags to not processed
   walkPtr = graph->first;
   while (walkPtr)
      {
       walkPtr->processed = 0;
       walkPtr            = walkPtr->pNextVertex;
      } // while

   // Process each vertex in list
   stack = createStack ();
   walkPtr = graph->first;
   while (walkPtr)
     {
      if (walkPtr->processed < 2)
        {
         if (walkPtr->processed < 1)
           {
            // Push & set flag to pushed
            success = pushStack (stack, walkPtr);
            if (!success)
                printf("\aStack overflow 100\a\n"),
                      exit (100);
            walkPtr->processed = 1;
           } // if processed < 1
        } // if processed < 2
      // Process descendents of vertex at stack top
      while (!emptyStack (stack))
        {
         vertexPtr = popStack(stack);
         process (vertexPtr->dataPtr);
         vertexPtr->processed = 2;

         // Push all vertices from adjacency list
         arcWalkPtr = vertexPtr->pArc;
         while (arcWalkPtr)
           {
```

continued

PROGRAM 11-6 Graph Depth-first Traversal *(continued)*

```
61              vertToPtr = arcWalkPtr->destination;
62              if (vertToPtr->processed == 0)
63                {
64                 success = pushStack(stack, vertToPtr);
65                 if (!success)
66                   printf("\aStack overflow 101\a\n"),
67                       exit (101);
68                 vertToPtr->processed = 1;
69                } // if vertToPtr
70              arcWalkPtr = arcWalkPtr->pNextArc;
71             } // while pWalkArc
72
73          } // while !emptyStack
74        walkPtr = walkPtr->pNextVertex;
75       } // while walkPtr
76    destroyStack(stack);
77    return;
78 } // graphDpthFrst
```

Graph Breadth-first Traversal

Again, we suggest that you study Figure 11-12, "Breadth-first Traversal of a Graph," as you work with this function. Because we want to process all of the adjacent vertices of a vertex before moving down the structure, we use a queue. The design is shown in Algorithm 11-9, the code in Program 11-7.

PROGRAM 11-7 Graph Breadth-first Traversal

```
 1 /* ================== graphBrdthFrst ====================
 2     Process the data of the graph in breadth-first order.
 3        Pre  graph is pointer to graph head structure
 4        Post graph has been processed
 5        Processed Flag: 0 = not processed
 6                        1 = enqueued
 7                        2 = processed
 8 */
 9 void graphBrdthFrst (GRAPH* graph,
10                       void (*process) (void* dataPtr))
11 {
12 // Local Definitions
13    bool    success;
14    VERTEX* walkPtr;
15    VERTEX* vertexPtr;
16    VERTEX* vertToPtr;
17    QUEUE*  queue;
18    ARC*    arcWalkPtr;
19
```

continued

PROGRAM 11-7 Graph Breadth-first Traversal *(continued)*

```
20  // Statements
21     if (!graph->first)
22         return;
23
24     // Set processed flags to not processed
25     walkPtr = graph->first;
26     while (walkPtr)
27        {
28         walkPtr->processed = 0;
29         walkPtr            = walkPtr->pNextVertex;
30        } // while
31
32     // Process each vertex in list
33     queue = createQueue ();
34     walkPtr = graph->first;
35     while (walkPtr)
36        {
37         if (walkPtr->processed < 2)
38            {
39             if (walkPtr->processed < 1)
40                {
41                 // Enqueue & set flag to queue
42                 success = enqueue(queue, walkPtr);
43                 if (!success)
44                    printf("\aQueue overflow 100\a\n"),
45                    exit (100);
46                 walkPtr->processed = 1;
47                } // if processed < 1
48            } // if processed < 2
49         // Process descendents of vertex at que front
50         while (!emptyQueue (queue))
51            {
52             dequeue(queue, (void**)&vertexPtr);
53             process (vertexPtr->dataPtr);
54             vertexPtr->processed = 2;
55
56             // Enqueue vertices from adjacency list
57             arcWalkPtr = vertexPtr->pArc;
58             while (arcWalkPtr)
59               {
60                vertToPtr = arcWalkPtr->destination;
61                if (vertToPtr->processed == 0)
62                  {
63                   success = enqueue(queue, vertToPtr);
64                   if (!success)
65                     printf("\aQueue overflow 101\a\n"),
66                        exit (101);
67                   vertToPtr->processed = 1;
```

continued

PROGRAM 11-7 Graph Breadth-first Traversal *(continued)*

```
68                  } // if vertToPtr
69                arcWalkPtr = arcWalkPtr->pNextArc;
70              } // while pWalkArc
71            } // while !emptyQueue
72          walkPtr = walkPtr->pNextVertex;
73        } // while walkPtr
74    destroyQueue(queue);
75    return;
76  } // graphBrdthFrst
```

11.6 Networks

A **network** is a graph whose lines are weighted. It is also known as a **weighted graph**. The meaning of the weights depends on the application. For example, an airline might use a graph to represent the routes between cities that it serves. In this example the vertices represent the cities and the edge a route between two cities. The edge's weight could represent the miles between the two cities or the price of the flight. A network for a small hypothetical airline is shown in Figure 11-18. In this case the weights represent the mileage between the cities.

A network is a graph whose lines are weighted.

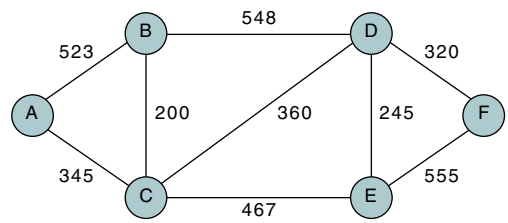


FIGURE 11-18 City Network

Because the weight is an attribute of an edge, it is stored in the structure that contains the edge. In an adjacency matrix, the weight is stored as the intersection value. In an adjacency list, it is stored as the value in the adjacency linked list. The representation of the city network in these two formats is shown in Figure 11-19.

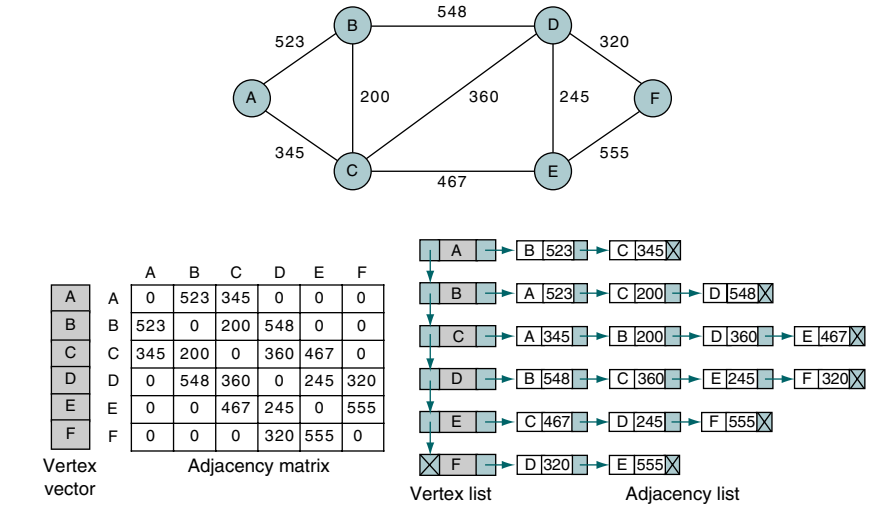


FIGURE 11-19 Storing Weights in Graph Structures

We now turn our attention to two applications of networks: the minimum spanning tree and the shortest path through a network.

Minimum Spanning Tree

We can derive one or more spanning trees from a connected network. A **spanning tree** is a tree that contains all of the vertices in the graph.

One interesting algorithm derives the **minimum spanning tree** of a network such that the sum of its weights is guaranteed to be minimal. If the weights in the network are unique, there is only one minimum spanning tree. If there are duplicate weights, there may be one or more minimum spanning trees.

There are many applications for minimum spanning trees, all with the requirement to minimize some aspect of the graph, such as the distance among all of the vertices in the graph. For example, given a network of computers, we can create a tree that connects all of the computers. The minimum spanning tree gives us the shortest length of cable that can be used to connect all of the computers while ensuring that there is a path between any two computers.

A spanning tree contains all of the vertices in a graph. A minimum spanning tree is a spanning tree in which the total weight of the lines is guaranteed to be the minimum of all possible trees in the graph.

To create a minimum spanning tree in a strongly connected network—that is, in a network in which there is a path between any two vertices—the edges for the minimum spanning tree are chosen so that the following properties exist:

1. Every vertex is included.
2. The total edge weight of the spanning tree is the minimum possible that includes a path between any two vertices.

Minimum Spanning Tree Example

Before going into the formal algorithm definition, let's manually determine the minimum spanning tree shown in Figure 11-20.

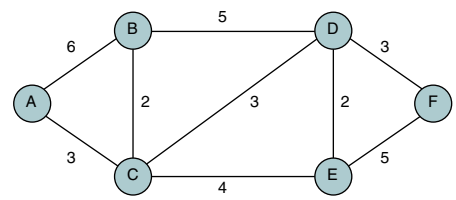


FIGURE 11-20 Spanning Tree

We can start with any vertex. Because the vertex list is usually key sequenced, let's start with A. Then we add the vertex that gives the minimum-weighted edge with A in Figure 11-20, AC. From the two vertices in the tree, A and C, we now locate the edge with the minimum weight. The edge AB has a weight of 6, the edge BC has a weight of 2, the edge CD has a weight of 3, and the edge CE has a weight of 4. The minimum-weighted edge is therefore BC. Note that in this analysis we do not consider any edge to a vertex that is already in the list. Thus, we did not consider the edge AC.

To generalize the process, we use the following rule: from all of the vertices currently in the tree, select the edge with the minimal value to a vertex not currently in the tree and insert it into the tree. Using this rule, we add CD (weight 3), DE (weight 2), and DF (weight 3) in turn. The steps are graphically shown in Figure 11-21.

Figure 11-21(g) shows the minimum spanning tree within the original network. If we sum the weights of the edges in the tree, the total is 13. Because there are duplicate weights, a different spanning tree may have the same weight, but none has a lesser weight.

Minimum Spanning Tree Data Structure

To develop the algorithm for the minimum spanning tree, we need to decide on a storage structure. Because it is the most flexible, we use the adjacency list.

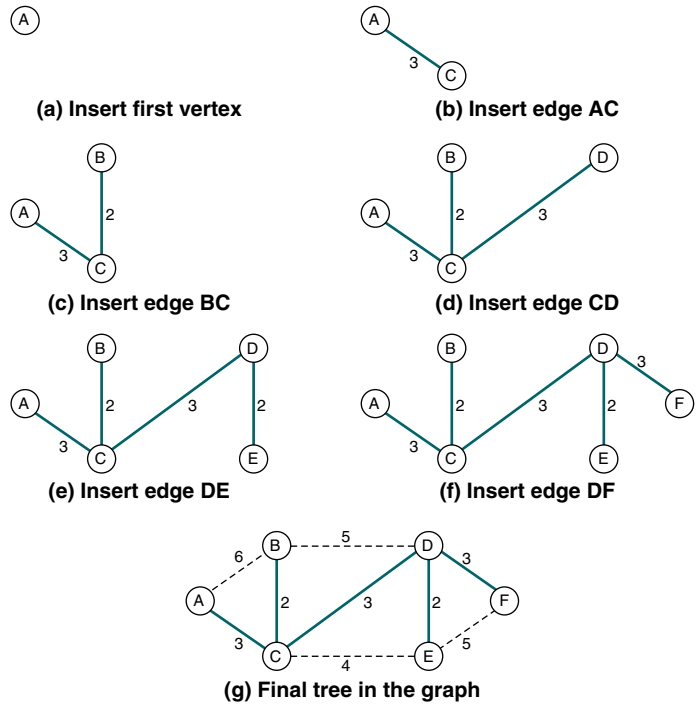


FIGURE 11-21 Develop a Minimum Spanning Tree

We need to add some additional elements to determine the minimum spanning tree. Each vertex node needs a Boolean flag indicating that the vertex has been inserted into the spanning tree. In addition, each arc needs an in-tree Boolean. We name both of these fields `inTree`. Finally, because the structure represents a network, each edge must have a weight. The resulting structure is shown in Algorithm 11-11.

ALGORITHM 11-11 Data Structure for Spanning Tree Graph

```
graphHead
   count
   first
end graphHead

graphVertex
   nextVertex
   data
   inDegree
   outDegree
```

continued

ALGORITHM 11-11 Data Structure for Spanning Tree Graph *(continued)*

```
    inTree
    firstEdge
end graphVertex

graphEdge
    destination
    weight
    nextEdge
    inTree
end graphEdge
```

Minimum Spanning Tree Pseudocode

The minimum spanning tree algorithm follows the concept outlined earlier. It begins by inserting the first vertex into the tree. It then loops until all vertices are in the tree, each time inserting the edge with the minimum weight into the tree. When the algorithm concludes, the minimum spanning tree is identified by the in-tree flags in the edges. The pseudocode is shown in Algorithm 11-12.

ALGORITHM 11-12 Minimum Spanning Tree of a Graph

```
Algorithm spanningTree (graph)
Determine the minimum spanning tree of a network.
    Pre  graph contains a network
    Post spanning tree determined
1  if (empty graph)
   1   return
2  end if
3  loop (through all vertices)
       Set inTree flags false.
   1   set vertex inTree flag to false
   2   loop (through all edges)
       1   set edge inTree flag to false
       2   get next edge
   3   end loop
   4   get next vertex
4  end loop
   Now derive spanning tree.
5  set first vertex to in tree
6  set treeComplete to false
7  loop (not treeComplete)
   1   set treeComplete to true
   2   set minEdge to maximum integer
   3   set minEdgeLoc to null
   4   loop (through all vertices)
           Walk through graph checking vertices in tree.
```

continued

ALGORITHM 11-12 Minimum Spanning Tree of a Graph *(continued)*

```
      1  if (vertex in tree AND vertex outDegree > 0)
         1  loop (through all edges)
            1  if (destination not in tree)
                     set destination inTree flag to false)
               1  set treeComplete to false
               2  if (edge weight < minEdge)
                  1  set minEdge to edge weight
                  2  set minEdgeLoc to edge
               3  end if
            2  end if
            3  get next edge
         2  end loop
      2  end if
      3  get next vertex
   5  end loop
   6  if (minEdgeLoc not null)
         Found edge to insert into tree.
      1  set minEdgeLoc  inTree flag to true
      2  set destination inTree flag to true
   7  end if
 8  end loop
end spanningTree
```

Algorithm 11-12 Analysis This rather long algorithm is easily broken into two sections. The first section prepares the graph for processing by setting all of the in-tree flags to false.

The second section loops (see statement 7) through the vertex graph, inserting edges into the spanning tree. As we begin the loop, we set the tree completed flag to true. If we find a vertex that is not yet in the tree, we set it to false (statement 7.4.1.1.1.1). On the last pass through the graph, no new edges are added and the tree completed flag remains true, thus terminating the vertex loop. Because we need to remember the edge with the minimum weight, at the beginning of the loop we also set a minimum edge variable to a value larger than any possible weight, positive infinity in the pseudocode. At the same time, we set a pointer to the minimum edge to null.

Within the loop the edges from each vertex already in the tree are tested with an inner loop that traverses the edges, looking for the minimum valued edge. Each edge that is not in the tree is tested to determine whether its weight is less than the current minimum we have located. If it is, we save the weight and the location of the edge. At the end of the loop (statement 7.6), we set the flags for both the newly inserted vertex and its associated edge.

Shortest Path Algorithm

Another common application used with graphs requires that we find the shortest path between two vertices in a network. For example, if the network represents the routes flown by an airline, when we travel we want to find the least expensive route between home and our destination. When the weights in the route graph are the flight fare, our minimum cost is the shortest path between our origin and our destination. Edsger Dijkstra developed a classic

algorithm for just this problem in 1959.[2] His algorithm is generally known simply as Dijkstra's shortest path algorithm.

The Dijkstra algorithm is used to find the shortest path between any two nodes in a graph.

Shortest Path Manual Example

Before we develop the formal algorithm, let's walk through an example. We use the same graph we used for the minimum spanning tree. In this example we want to find the shortest path from vertex A to any other vertex in the graph. The result is a tree with a root of vertex A. The result of our analysis is shown in Figure 11-22.

The algorithm is similar to the minimum spanning tree algorithm. We begin by inserting the starting point into the tree. We then examine the paths to all vertices adjacent to the starting point and insert the path with the minimum weight into the tree. We now have two vertices in the tree, as shown in Figure 11-22(a2).

We then examine all of the paths from the two vertices in the tree, A and C, to all of their adjacent vertices. Rather than determine the shortest path to the next vertex as we did in the spanning tree, however, we determine the total path length to the adjacent vertices. The total path length to each vertex is shown in Figure 11-22 with the notation `T:`*n* next to the destination vertex. If you examine Figure 11-22(b1), you see that there are four paths: AB with a total weight of 6, CB with a total weight of 5, CD with a total weight of 6, and CE with a total weight of 7. We select the minimum path length, CB (`T:5`), and place it in the tree. The resulting tree is shown in Figure 11-22(b2).

In the third pass through the graph, we examine the paths from the vertices in the tree (A, C, and B) to vertices not already in the tree (D and E). There are no paths from A. From B we have one path, BD, with a total weight of 10. From C we have two paths, CD with a total weight of 6 and CE with a total weight of 7. We select the minimum path, CD, with a total weight of 6.

Let's try to generalize the steps we have been following:

1. Insert the first vertex into the tree.
2. From every vertex already in the tree, examine the total path length to all adjacent vertices not in the tree. Select the edge with the minimum total path weight and insert it into the tree.
3. Repeat step 2 until all vertices are in the tree.

Following these steps, we next insert edge CE shown in Figure 11-22(d2), and last we add edge DF as shown in Figure 11-22(e2).

2. E. W. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik* 1 (1959): 269–271.

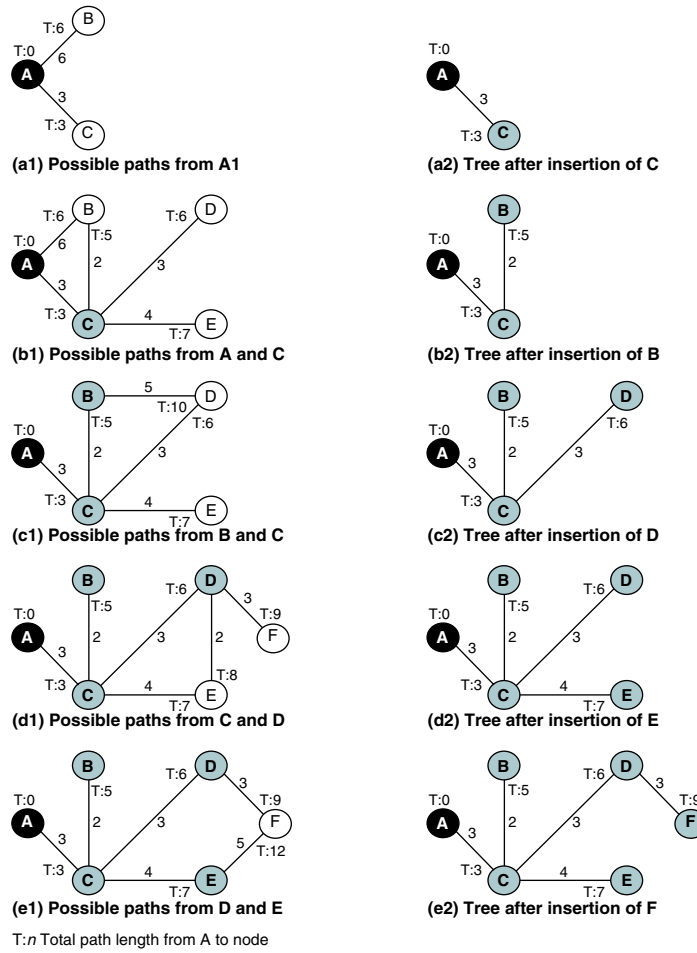


FIGURE 11-22 Determining Shortest Path

Shortest Path Pseudocode

We are now ready to develop the algorithm. To determine the minimum path, we need to add one more field to the vertex structure, a total path length to the vertex. We call it `pathLength`. The pseudocode is shown in Algorithm 11-13.

ALGORITHM 11-13 Shortest Path

```
Algorithm shortestPath (graph)
Determine shortest path from a network vertex to other
vertices.
   Pre    graph is pointer to network
   Post   minimum path tree determined
```

continued

ALGORITHM 11-13 Shortest Path *(continued)*

```
1  if (empty graph)
   1  return
2  end if
3  loop (through all vertices)
      Initialize inTree flags and path length.
   1  set vertex inTree flag to false
   2  set vertex pathLength to maximum integer
   3  loop (through all edges)
      1  set edge inTree flag to false
      2  get next edge
   4  end loop
   5  get next vertex
4  end loop
   Now derive minimum path tree.
5  set first vertex inTree flag to true
6  set vertex pathLength to 0
7  set treeComplete to false
8  loop (not treeComplete)
   1  set treeComplete to true
   2  set minEdgeLoc to null
   3  set pathLoc    to null
   4  set newPathLen to maximum integer
   5  loop (through all vertices)
         Walk through graph checking vertices in tree.
      1  if (vertex in tree AND outDegree > 0)
         1  set edgeLoc to firstEdge
         2  set minPath to pathLength
         3  set minEdge to maximum integer
         4  loop (through all edges)
               Locate smallest path from this vertex.
            1  if (destination not in tree)
               1  set treeComplete to false
               2  if (edge weight < minEdge)
                  1  set minEdge to edge weight
                  2  set minEdgeLoc to edgeLoc
               3  end if
            2  end if
            3  set edgeLoc to edgeLoc nextEdge
         5  end loop
            Test for shortest path.
         6  if (minPath + minEdge < newPathLen)
            1  set newPathLen to minPath + minEdge
            2  set pathLoc    to minEdgeLoc
         7  end if
      2  end if
      3  get next vertex
   6  end loop
```

continued

ALGORITHM 11-13 Shortest path *(continued)*

```
   7  if (pathLoc not null)
         Found edge to insert into tree.
      1  set pathLoc             inTree flag to true
      2  set pathLoc destination inTree flag to true
      3  set pathLoc destination pathLength to newPathLen
   8  end if
 9 end loop
end shortestPath
```

Algorithm 11-13 Analysis

As we saw in the minimum spanning tree, the shortest path algorithm begins with an initialization loop that sets the in-tree flags to false and initializes the vertex path length to a number greater than the maximum possible length (maximum integer).

Once we have initialized the vertices, we are ready to determine the shortest path to all vertices. We start with the first vertex in the graph, as indicated by the graph's first pointer. If we need to determine the shortest path from another vertex, we have to pass the starting point to the algorithm.

Each loop through the graph (statement 8) inserts the vertex with the shortest path to any connected vertex, initially only A. For each vertex in the tree (statement 8.5), we test all of its adjacent vertices and determine the minimum weight to a vertex not already in the tree (statement 8.5.1.4). As we locate each path edge, we test to see whether it is less than any previous path edge (statement 8.5.1.6); if it is, we save the new path length and the pointer to the edge.

When we locate the shortest path edge, we insert it (statement 8.7) and loop back to check the graph again (statement 8).

11.7 Key Terms

adjacency list
adjacency matrix
adjacent vertices
arc
breadth-first traversal
connected
cycle
degree
depth-first traversal
digraph
directed graph (digraph)
disjoint graph
edge
graph
indegree
line
loop
minimum spanning tree
network
outdegree
path
spanning tree
strongly connected
undirected graph
vertex (vertices)
vertex list
weakly connected
weighted graph

11.8 Summary

- A graph is a collection of nodes, called vertices, and a collection of line segments connecting pairs of nodes, called edges or arcs.
- Graphs may be directed or undirected. In a directed graph, or digraph, each line has a direction. In an undirected graph, there is no direction on the lines. A line in a directed graph is called an arc.
- In a graph two vertices are said to be adjacent if an edge directly connects them.
- A path is a sequence of vertices in which each vertex is adjacent to the next one.
- A cycle is a path of at least three vertices that starts and ends with the same vertex.
- A loop is a special case of a cycle in which a single arc begins and ends with the same node.
- A graph is said to be connected if, for any two vertices, there is a path from one to the other. A graph is disjointed if it is not connected.
- The degree of a vertex is the number of vertices adjacent to it. The outdegree of a vertex is the number of arcs leaving the node; the indegree of a vertex is the number of arcs entering the node.
- Six operations have been defined for a graph: insert a vertex, delete a vertex, add an edge, delete an edge, find a vertex, and traverse a graph.
  - Add a vertex operation inserts a new vertex into a graph without connecting it to any other vertex.

- Delete a vertex operation removes a vertex from a graph.
- Add an edge operation inserts an edge between a source vertex and a destination vertex in a graph.
- Delete an edge operation removes an edge connecting the source vertex to the destination vertex in a graph.
- Find a vertex operation traverses a graph, looking for a specified vertex.
- Traverse a graph visits all nodes in the graph and processes them one by one.

- There are two standard graph traversals: depth first and breadth first.
  - In the depth-first traversal, all of a node's descendents are processed before moving to an adjacent node.
  - In the breadth-first traversal, all of the adjacent vertices are processed before processing the descendents of a vertex.
- To represent a graph in a computer, we need to store two sets of information: the first set represents the vertices, and the second set represents the edges.
- The most common methods used to store a graph are the adjacency matrix method and the adjacency list method.
  - In the adjacency matrix method, we use a vector to store the vertices and a matrix to store the edges.
  - In the adjacency list method, we use a linked list to store the vertices and a two-dimensional linked list to store the edges.
- A network is a graph whose lines are weighted.
- A spanning tree contains all of the vertices in the graph.
- A minimum spanning tree is a spanning tree in which the total weight of the edges is the minimum.
- Another common algorithm used with graphs finds the shortest path between two vertices.

11.9 Practice Sets

Exercises

1. In the graph in Figure 11-23, find:
   a. all noncyclic paths from A to H
   b. all noncyclic paths from C to E
   c. all noncyclic paths from B to F

2. In the graph in Figure 11-23, find all nodes adjacent to:
   a. node A
   b. node F
   c. node G

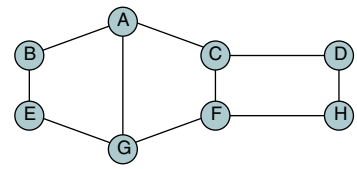


FIGURE 11-23 Graph for Exercises 1 through 8

3. In the graph in Figure 11-23, find the degree, outdegree, and indegree of vertices A, E, F, G, and H.
4. Give the depth-first traversal of the graph in Figure 11-23, starting from vertex A.
5. Give the breadth-first traversal of the graph in Figure 11-23, starting from vertex A.
6. Draw three spanning trees that can be found in the graph in Figure 11-23.
7. Give the adjacency matrix representation of the graph in Figure 11-23.
8. Give the adjacency list representation of the graph in Figure 11-23.
9. Find the minimum spanning tree of the graph in Figure 11-24.

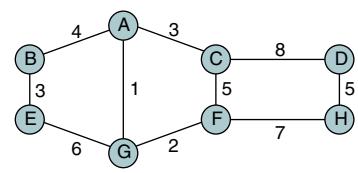


FIGURE 11-24 Graph for Exercises 9 through 12

10. Find the shortest path between node A and all other nodes in the graph in Figure 11-24.
11. Give the adjacency matrix representation of the graph in Figure 11-24.
12. Give the adjacency list representation of the graph in Figure 11-24.
13. Draw the directed graph for the adjacency matrix representation in Figure 11-25.

	A	B	C	D	E	F
A	0	3	4	0	2	1
B	0	0	2	0	0	3
C	0	0	0	2	6	1
D	2	6	1	0	1	2
E	0	0	0	0	0	3
F	0	0	0	0	0	0

FIGURE 11-25 Adjacency Matrix for Exercise 13

14. A graph can be used to show relationships. For example, from the following list of people belonging to the same club (vertices) and their friendships (edges), find:

```
People = {George, Jim, Jean, Frank, Fred, John, Susan}
Friendship = {(George, Jean), (Frank, Fred),
              (George, John), (Jim, Fred),
              (Jim, Frank),   (Jim, Susan),
              (Susan, Frank)}
```

a. all friends of John
b. all friends of Susan
c. all friends of friends of Jean
d. all friends of friends of Jim

Problems

15. Write an algorithm that determines whether a node is disjoint.
16. Write an algorithm that disjoints a node.
17. Write an algorithm that finds the sum of the degrees for a node, using the adjacency list representation.
18. Both the depth-first and breadth-first traversals process disjoint graphs. Write an algorithm that traverses a graph and returns true if the graph is connected and false if it is disjoint.
19. Write an algorithm that determines whether there is at least one arc pointing to a specified vertex. (*Hint:* This is a very simple algorithm.)
20. In a weakly connected graph, it may not be possible to start at one vertex and reach another. Write an algorithm that, when given the graph, a source vertex, and a destination vertex, determines whether there is at least one path from the source to the destination.
21. Algorithm 11-8, "Depth-first Traversal," uses a stack to traverse the graph. Rewrite the algorithm to use recursion. (*Hint:* You need two algorithms, one to interface with the using algorithm and one for the recursion.)

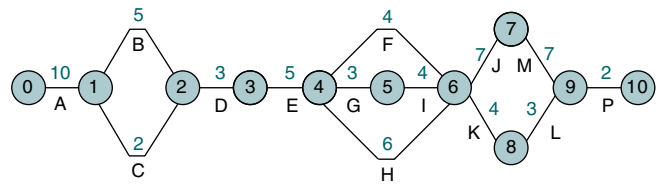
Projects

22. Write the C code for Algorithm 11-12, "Minimum Spanning Tree of a Graph," using the ADT given in the text.
23. Write the C code for Algorithm 11-13, "Shortest Path," using the ADT given in the text.
24. Write an algorithm that prints the minimum spanning tree of a graph. At the end, print the weight of the spanning tree. A suggested report format is shown in the following example.

```
Source Vertex              To Vertex      Weight
      A                          B           2
      A                          C           4
      B                          D           3
      D                          E           1
Total weight of spanning tree: 10
```

25. Revise the graph ADT using an adjacency matrix as the data structure.
26. One of the tools used to manage large projects is known as the critical path method (CPM). In CPM the manager builds a network of all phases of a project and then evaluates the network to determine critical aspects of the project.

    In a CPM network, each vertex is an event, such as the start or completion of a task. The arcs connecting the vertices represent the duration of the activity. Unlike the examples in the text, they also store the name of the activity. To better understand the concept, let's look at a possible CPM plan to build a house. The network for this project is shown in Figure 11-26.



A. Prepare plans (10)
B. Building permits (5)
C. Select contractor (2)
D. Foundation (3)
E. Walls (5)
F. Electrical (4)
G. Roof (3)
H. Plumbing (6)
I. Drywall (4)
J. Paint outside (7)
K. Paint inside (4)
L. Fixtures/carpet (3)
M. Landscape (7)
P. Final inspection (2)

FIGURE 11-26 Project 26: Steps for Building a House

In the plan we see that it will take 10 days to prepare the building plan (A) and 5 days to get it approved (B). Furthermore, we can't start building until we have selected the contractor (C).

We could construct the shortest path from the start to the end for our plan, but it would be of little value. On the other hand, if we determined

the maximum path—that is, the path with the greatest sum of the weights—we would know which steps in our plan are critical. If a critical step slips even one day, we slip our end date. We can slip noncritical dates, however, without slipping our end date, so long as the slip does not change the critical path for the project.

Modify Algorithm 11-12, "Minimum Spanning Tree of a Graph," to determine the maximum path through the graph. Then provide a menu that allows the project manager to answer the following questions:

a. What is the shortest possible completion time (SPCT)? The SPCT is the longest path through the graph from beginning to end.
b. What is the earliest start time (EST) for each activity? The EST is the sum of the weights in the maximum spanning tree up to the activity.
c. What is the latest start time (LST) for each activity? The LST is the SPCT for the whole project minus the SPCT for the rest of the project (starting from the current activity).
d. What is the slack time for each activity? The slack time is LST – EST.
e. Is an activity a critical path item? (Critical path items have a slack time of zero.)
f. What is the critical path for the project? (The critical path is the subgraph consisting of the maximum spanning tree.)

27. The graph is another structure that can be used to solve the maze problem (see Project 24 in Chapter 3). Every start point, dead end, goal, and decision point can be represented by a node. The arcs between the nodes represent one possible path through the maze. A graph maze is shown in Figure 11-27.

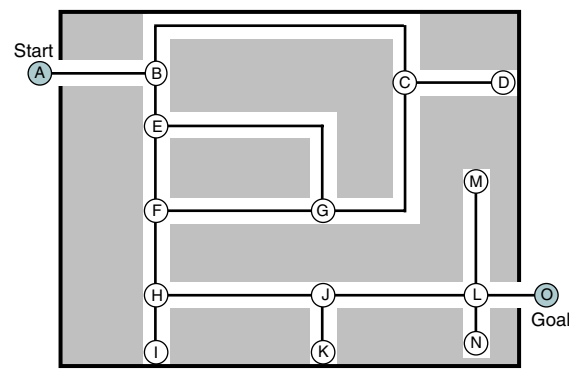


FIGURE 11-27 Graph Maze for Project 27

Write a program that simulates a mouse's movement through the maze, using a graph and a depth-first traversal. When the program is complete, print the path through the maze.

28. A computer company in the Silicon Valley area (see Figure 11-28) needs to route delivery vehicles between cities on the shortest route. Having studied data structures, you recognize that this is an application for Dijkstra's shortest path algorithm. To demonstrate your proposal, you decide to implement it on your computer. To do so you must complete the following tasks:

    a. Convert the map in Figure 11-28 to a network and present it to management.
    b. Modify the graph ADT to store weights in the arc nodes.
    c. Write an interactive program that when given the start and destination displays the shortest route between them.

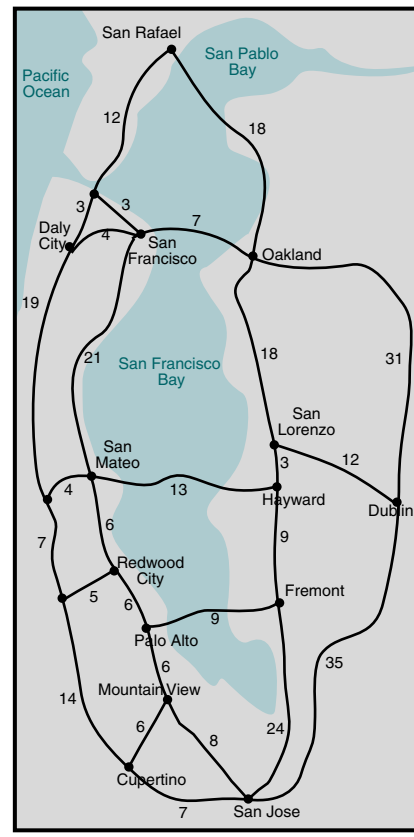


FIGURE 11-28 Map of Silicon Valley Area

This page intentionally left blank

Part IV

Sorting and Searching

The last part of the book touches on two subjects that are not considered abstract data types but are very closely related: sorting and searching. Although sorting and searching are briefly discussed in an introductory programming course, the discussion does not normally get into advanced concepts. In this part we discuss advanced sorting and searching algorithms. Figure IV-1 shows the chapters included in this part and the topics discussed in each chapter.

Included in the two chapters we find discussions about the complexity of sorting and searching. The ideas developed in Chapter 1 are applied to each sorting and searching algorithm, and some informal but simple proofs are given. Complexity analysis enables us to understand the difference between the fast sorts developed in this part compared with slow sorts discussed in an introductory programming course.

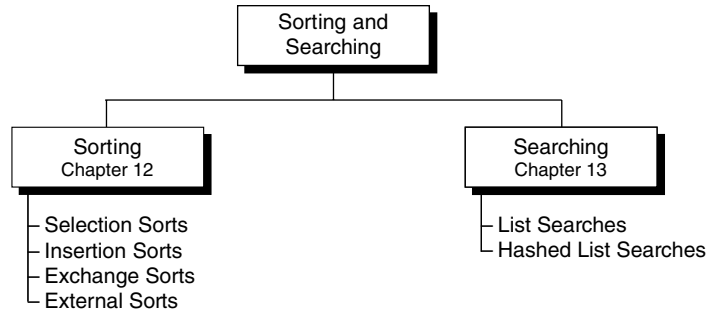


FIGURE IV-1

Implementation

The sorting and searching algorithms developed in this part of the book are directly implemented as stand-alone functions in C. Although abstract data types can be used to develop generic functions, we do not use them in this section. Rather, we keep the algorithm simple to allow us to concentrate on the algorithm logic.

Chapters Covered

This part includes two chapters.

Chapter 12: Sorting

In this chapter we discuss sorting. The chapter begins with a general discussion of sort concepts. We then introduce four sort categories: selection sort, insertion sort, exchange sort, and external sort.

Chapter 13: Searching

In this chapter we discuss searching. We begin with a discussion of the basic sequential search, including several variations, and the binary search. We conclude with a discussion of hashed list searches and collison resolution.

Chapter 12
Sorting

One of the most common applications in computer science is **sort**, the process through which data are arranged according to their values. We are surrounded by data. If data were not ordered, we would spend hours trying to find a single piece of information. Imagine the difficulty of finding someone's telephone number in a telephone book that had no internal order.

The history of sorting goes back to the roots of computing. Herman Hollerith's electric tabulating machine was used to tally the 1890 U.S. Census and was one of the first modern sorting machines. Sorting was also on the scene when general-purpose computers first came into use. According to Knuth, "There is evidence that a sorting routine was the first program ever written for a stored program computer."[1] Although computer scientists have not developed a major new algorithm in more than 30 years (the newest algorithm in this book is heap sort, which was developed in 1964), sorting is still one of the most important concepts in computing today.

12.1 Sort Concepts

We discuss six internal sorts in this chapter: insertion sort, bubble sort, selection sort, shell sort, heap sort, and quick sort. The first three are useful only for sorting very small lists, but they form the basis of the last three, which are useful general-purpose sorting concepts. After discussing the internal sorts, we will introduce the basic concepts used in external sorts.

Sorting is one of the most common data-processing applications.

1. Donald E. Knuth, *The Art of Computer Programming,* vol. 3, *Sorting and Searching*, Second Edition (Reading, MA: Addison-Wesley, 1998), 385.

Sorts are generally classified as either internal or external sorts. In an **internal sort**, all of the data are held in primary memory during the sorting process. An **external sort** uses primary memory for the data currently being sorted and secondary storage for any data that does not fit in primary memory. For example, a file of 20,000 records may be sorted using an array that holds only 1000 records. During the sorting process, only 1000 records are therefore in memory at any one time; the other 19,000 are kept in a file in secondary storage.

Sorting algorithms are classified as either internal or external.

Internal sorting algorithms have been grouped into several different classifications depending on their general approach to sorting. Knuth identified five different classifications: insertion, selection, exchanging, merging, and distribution sorts.[2] In this text we cover the first three. Distribution sorts, although interesting, have minimal use in computers. The different sorts are shown in Figure 12-1.

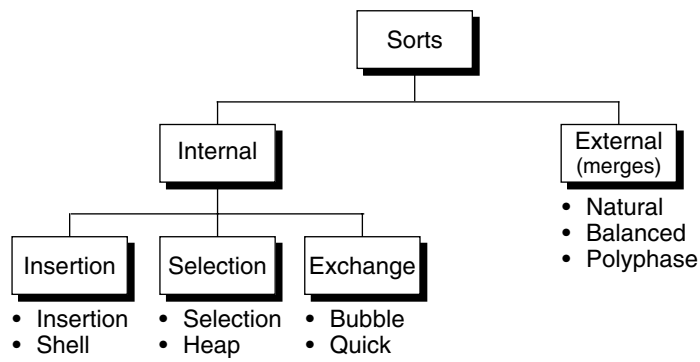


FIGURE 12-1 Sort Classifications

Sort Order

Data may be sorted in either ascending sequence or descending sequence. The **sort order** identifies the sequence of the sorted data, ascending or descending. If the order of the sort is not specified, it is assumed to be ascending. Examples of common data sorted in ascending sequence are the dictionary and the telephone book. Examples of common descending data are percentages of games won in a sporting event such as baseball or grade-point averages for honor students.

2. See Knuth, *Art of Computer Programming,* 73–179.

Sort Stability

Sort stability is an attribute of a sort, indicating that data with equal keys maintain their relative input order in the output. Stability is best seen in an example. In Figure 12-2(a) the unsorted data contain three entries with identical keys (212). If the data are sorted with a stable sort, the order in Figure 12-2(b) is guaranteed. That is, 212 `green` is guaranteed to be the first of the three in the output, 212 `yellow` is guaranteed to be the second, and 212 `blue` is guaranteed to be the last. If the sort is not stable, records with identical keys may occur in any order, including the stable order shown in Figure 12-2(b). Figure 12-2(c) is one example of the six different sequences that could occur in an unstable sort. Note that in this example `blue` comes out first even though it was the last of the equal keys. Of the sort algorithms we will discuss in this text, the straight insertion sort and the bubble sort are stable; the others are unstable.

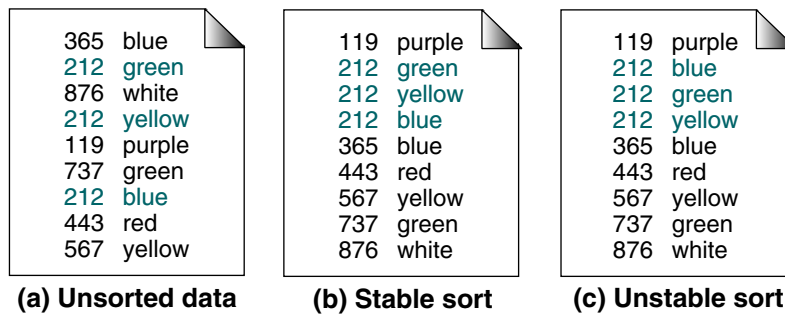


FIGURE 12-2 Sort Stability

Sort Efficiency

Sort efficiency is a measure of the relative efficiency of a sort. It is usually an estimate of the number of comparisons and moves required to order an unordered list. We discuss the sort efficiency of each of the internal sorts we cover in this chapter. Generally speaking, however, the best possible sorting algorithms are on the order of $n \log n$; that is, they are $O(n \log n)$ sorts.[3] Three of the sorts we study are $O(n^2)$. The best, quick sort, is $O(n \log n)$.

Passes

During the sorting process, the data are traversed many times. Each traversal of the data is referred to as a **sort pass**. Depending on the algorithm, the sort

3. As a project, we discuss an interesting sort—the radix sort—which is $O(n)$. However, its extensive overhead limits its general use.

pass may traverse the whole list or just a section of the list. Also, a characteristic of a sort pass is the placement of one or more elements in a sorted list.

Sorts and ADTs

Sort algorithms can be easily adapted to an ADT implementation using the techniques discussed in Section 1.5, "Generic Code for ADTs." There is one significant difference between the ADTs in previous chapters and sort ADTs: in sort ADTs the data type is defined and filled in the application. It is then passed to the ADT for sorting.

To sort data of an unknown type and size, the ADT needs four parameters, as shown in the following prototype statement:

```
bool sortADT (void* ary,
              int   sizeofElem,
              int   numElem,
              int  (*compare)(void* arg1, void* arg2));
```

The data to be sorted is received as a *void* pointer. To locate an element in the array, we can treat the array as an array of characters and use pointer arithmetic to move from one element to another. For example, to address the fifth element of the array, we use the following statement:

```
(char*)ary + (walker * sizeofElem)
```

The number of elements entry (`numElem`) is needed to identify the physical end of the list. And, finally, a compare function is needed, as we have seen in all of the key-sequenced ADTs.

All sort algorithms require that array elements be exchanged. Because we don't know the physical type, the exchanges must treat each element as an array of characters and move them one at a time. The following code segment shows how to exchange two elements:

```
// Move smallest to hold area
for (int i = 0; i < sizeofElem; i++)
     *(holdData + i) = *(smallest + i);
// Move current to smallest location
movePtr = (char*)ary + (current * sizeofElem);
for (int i = 0; i < sizeofElem; i++)
     *(smallest + i) = *(movePtr + i);
// Move hold area current location
for (int i = 0; i < sizeofElem; i++)
     *(movePtr + i)  = *(holdData + i);
```

In this chapter we implement the sort algorithms using integer data so that we can concentrate on the sort design. We leave ADT implementations for problems and projects at the end of the chapter.

12.2 Selection Sorts

Selection sorts are among the most intuitive of all sorts. Given a list of data to be sorted, we simply select the smallest item and place it in a sorted list. These steps are then repeated until we have sorted all of the data. In this section we study two selection sorts: the straight selection sort and the heap sort.

> In each pass of the selection sort, the smallest element is selected from the unsorted sublist and exchanged with the element at the beginning of the unsorted list.

Straight Selection Sort

In the **straight selection sort**, the list at any moment is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall. We select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data. After each selection and exchange, the wall between the two sublists moves one element, increasing the number of sorted elements and decreasing the number of unsorted ones. Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed one sort pass. If we have a list of n elements, therefore, we need $n - 1$ passes to completely rearrange the data. The selection sort is graphically presented in Figure 12-3.

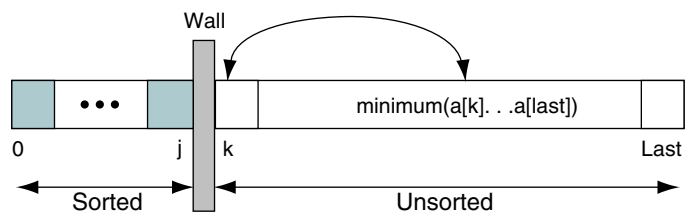


FIGURE 12-3 Selection Sort Concept

Figure 12-4 traces our set of six integers as we sort them. It shows how the wall between the sorted and unsorted sublists moves in each pass. As you study the figure, you see that the array is sorted after five passes, one less than the number of elements in the array. Thus, if we use a loop to control the sorting, our loop has one less iteration than the number of elements in the array.

Selection Sort Algorithm

If you knew nothing about sorting and were asked to sort a list on paper, you would undoubtedly scan the list to locate the smallest item and then copy it to a second list. You would then repeat the process of locating the smallest remaining item in the list and copying it to the new list until you had copied all items to the sorted list.

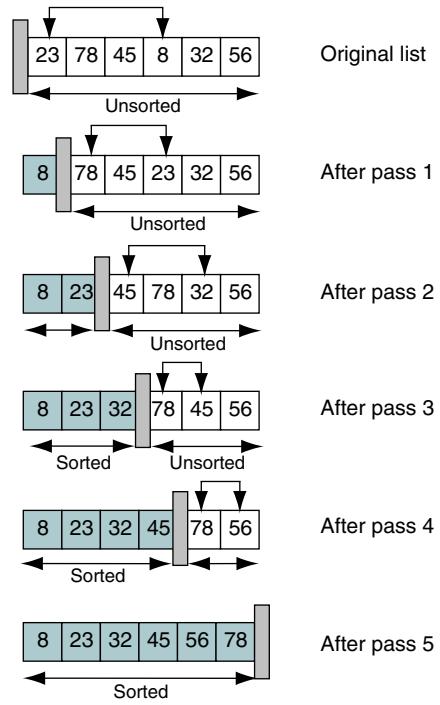


FIGURE 12-4 Selection Sort Example

With the exception of using two areas, this is exactly how the selection sort works. Starting with the first item in the list, the algorithm scans the list for the smallest element and exchanges it with the item at the beginning of the list. Each selection and exchange is one sort pass. After advancing the index (wall), the sort continues until the list is completely sorted. The pseudocode is shown in Algorithm 12-1.

ALGORITHM 12-1 Selection Sort

```
Algorithm selectionSort (list, last)
Sorts list array by selecting smallest element in
unsorted portion of array and exchanging it with element
at the beginning of the unsorted list.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged smallest to largest
1  set current to 0
2  loop (until last element sorted)
   1   set smallest to current
```

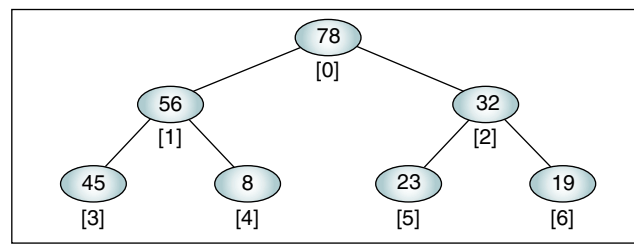
continued

ALGORITHM 12-1 Selection Sort *(continued)*

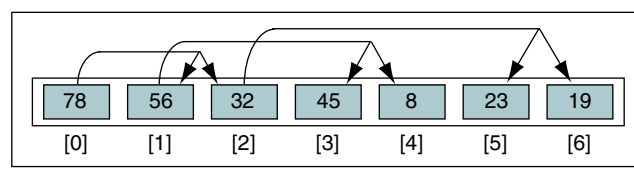
```
   2  set walker   to current + 1
   3  loop (walker <= last)
      1  if (walker key < smallest key)
         1  set smallest to walker
      2  increment walker
   4  end loop
      Smallest selected: exchange with current element.
   5  exchange (current, smallest)
   6  increment current
3  end loop
end selectionSort
```

Heap Sort

In Chapter 9 we studied heaps. Recall that a heap is a tree structure in which the root contains the largest (or smallest) element in the tree. (You may want to review Chapter 9 before studying heap sort.) As a quick review, Figure 12-5 shows a heap in its tree form and in its array form.



(a) Heap in its tree form



(b) Heap in its array form

FIGURE 12-5 Heap Representations

The **heap sort** algorithm is an improved version of the straight selection sort. The straight selection sort algorithm scans the unsorted elements and selects the smallest element. Finding the smallest element among the n elements requires $n - 1$ comparisons. This part of the selection sort makes it very slow.

The heap sort also selects an element from the unsorted portion of the list, but it is the largest element. Because the heap is a tree structure, however, we don't need to scan the entire list to locate the largest key. Rather, we

reheap, which moves the largest element to the root by following tree branches. This ability to follow branches makes the heap sort much faster than the straight selection sort.

Heap sort begins by turning the array to be sorted into a heap. The array is turned into a heap only once for each sort. We then exchange the root, which is the largest element in the heap, with the last element in the unsorted list. This exchange results in the largest element's being added to the beginning of the sorted list. We then reheap down to reconstruct the heap and exchange again. The reheap and exchange process continues until the entire list is sorted.

> The heap sort is an improved version of the selection sort in which the largest element (the root) is selected and exchanged with the last element in the unsorted list.

The heap sort process is shown in Figure 12-6. We first turn the unordered list into a heap. You should verify for yourself that the array is actually a heap.

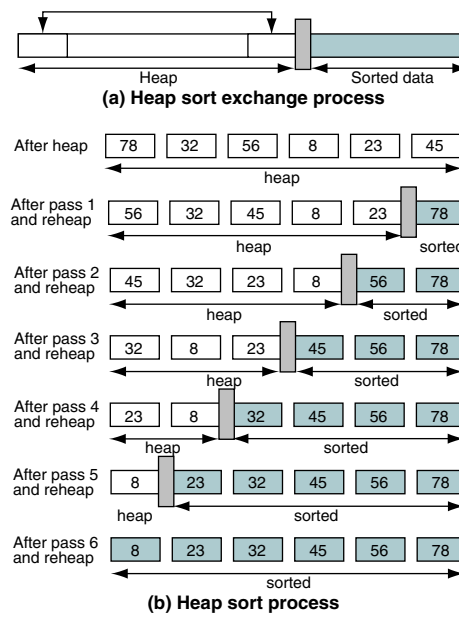


FIGURE 12-6 Heap Sort Process

Because the largest element (78) is at the top of the heap, we can exchange it with the last element in the heap and move the heap wall one element to the left. This exchange places the largest element in its correct location at the end of the array, but it destroys the heap. We therefore rebuild the heap. The smaller heap now has its largest element (56) at the top. We

exchange 56 with the element at the end of the heap (23), which places 56 in its correct location in the sorted array. The reheap and exchange processing continues until we have sorted the entire array.

Heap Sort Algorithm

Algorithm 12-2 contains the pseudocode for heap sort. It uses two algorithms defined in Chapter 9: Algorithm 9-1, "Reheap Up," and Algorithm 9-2, "Reheap Down."

ALGORITHM 12-2 Heap Sort

```
Algorithm heapSort (heap, last)
Sort an array, using a heap.
   Pre  heap array is filled
        last is index to last element in array
   Post heap array has been sorted
   Create heap
 1 set walker to 1
 2 loop (heap built)
   1  reheapUp (heap, walker)
   2  increment walker
 3 end loop
   Heap created. Now sort it.
 4 set sorted to last
 5 loop (until all data sorted)
   1  exchange (heap, 0, sorted)
   2  decrement sorted
   3  reheapDown (heap, 0, sorted)
 6 end loop
end heapSort
```

Algorithm 12-2 Analysis Whereas the heap-array structure is relatively complex, the heap sort algorithm is simple. The algorithm begins by using the reheap up algorithm to turn the array into a heap. It then sorts the array by exchanging the element at the top of the heap with the element at the end of the heap and rebuilding the heap using the reheap down algorithm.

Selection Sort Efficiency

In this section we examine the sort efficiency for the selection sorts.

Straight Selection Sort

The code for the straight selection sort is found in Algorithm 12-1, "Selection Sort." It contains the two loops shown below.

```
2  loop (until last element sorted)
   ...
   3  loop (walker <= last)
```

The outer loop executes $n - 1$ times. The inner loop also executes $n - 1$ times. This is a classic example of the quadratic loop. Its search effort therefore, using big-O notation, is $O(n^2)$.

The straight selection sort efficiency is $O(n^2)$.

Heap Sort

The heap sort pseudocode is shown in Algorithm 12-2. Ignoring the effort required to build the heap initially, the sort contains two loops. The first is a simple iterative loop; the second is a recursive loop:

```
5   loop (until all data sorted)
    ...
    3   reheapDown (heap, 0, sorted)
```

The outer loop starts at the end of the array and moves through the heap one element at a time until it reaches the first element. It therefore loops n times. The inner loop follows a branch down a binary tree from the root to a leaf or until the parent and child data are in heap order. The probability of the data being in order before we reach the leaf is very small, so we ignore it. The difficult part of this analysis is that for each of the outer loops, the heap becomes smaller, shortening the path from the root to a leaf. Again, except for the largest of heaps, this factor is rather minor and is eliminated in big-O analysis; therefore, we ignore it also.

Following the branches of a binary tree from a root to a leaf requires log n loops. The sort effort, the outer loop times the inner loop, for the heap sort is therefore

$$n(\log n)$$

When we include the processing to create the original heap, the big-O notation is the same. Creating the heap requires n log n loops through the data. When factored into the sort effort, it becomes a coefficient, which is then dropped to determine the final sort effort.

The heap sort efficiency is $O(n \log n)$.

Summary

Our analysis leads to the conclusion that the heap sort, $O(n \log n)$, is more efficient than the straight selection sort, $O(n^2)$. Table 12-1 offers a comparison of these two sorts. Depending on the algorithm's implementation, the run time can be affected.

n	Number of loops	
	Straight selection	Heap
25	625	116
100	10,000	664
500	250,000	4482
1000	1,000,000	9965
2000	4,000,000	10,965

TABLE 12-1 Comparison of Selection Sorts

Selection Sort Implementation

We now turn our attention to implementing the selection sort algorithms in C. We first implement the straight selection sort and then the heap sort.

Selection Sort C Code

The code for the selection sort is shown in Program 12-1. Its design is found in Algorithm 12-1.

PROGRAM 12-1 Selection Sort

```
/* ================== selectionSort ===================
   Sorts list [1…last] by selecting smallest element in
   unsorted portion of array and exchanging it with
   element at beginning of the unsorted list.
      Pre  list must contain at least one item
           last contains index to last list element
      Post list has been sorted smallest to largest
*/
void selectionSort  (int list[ ], int last)
{
// Local Declarations
   int smallest;
   int holdData;

// Statements
   for (int current = 0; current < last; current++)
       {
        smallest = current;
        for (int walker = current + 1;
                 walker <= last;
                 walker++)
            if (list[ walker ] < list[ smallest ])
```

continued

PROGRAM 12-1 Selection Sort *(continued)*

```
23                 smallest = walker;
24
25              // Smallest selected: exchange with current
26              holdData        = list[ current ];
27              list[current]   = list[ smallest ];
28              list[smallest]  = holdData;
29             } // for current
30      return;
31   }  // selectionSort
```

Heap Sort C Code

The heap sort program requires a driver function, which we call heap sort, and two functions discussed earlier, reheap up and reheap down. We have tailored them here for the sorting process.

Heap Sort Function

The heap sort function accepts an array of unsorted data and an index to the last element in the array. It then creates a heap and sorts it using reheap up and reheap down. The heap sort algorithms are shown in Program 12-2.

PROGRAM 12-2 Heap Sort

```
 1  /* ==================== heapSort ====================
 2     Sort an array, [list0 .. last], using a heap.
 3        Pre  list must contain at least one item
 4             last contains index to last element in list
 5        Post list has been sorted smallest to largest
 6  */
 7  void heapSort (int  list[ ], int  last)
 8  {
 9  // Local Definitions
10     int sorted;
11     int holdData;
12
13  // Statements
14     // Create Heap
15     for (int walker = 1; walker <= last; walker++)
16        reheapUp (list, walker);
17
18     // Heap created. Now sort it.
19     sorted = last;
20     while (sorted > 0)
21        {
22         holdData     = list[0];
23         list[0]      = list[sorted];
24         list[sorted] = holdData;
```

continued

PROGRAM 12-2 Heap Sort *(continued)*

```
25          sorted--;
26          reheapDown (list, 0, sorted);
27         } // while
28      return;
29   }  // heapSort
30
31   /* ==================== reheapUp ====================
32      Reestablishes heap by moving data in child up to
33      correct location heap array.
34         Pre  heap is array containing an invalid heap
35              newNode is index location to new data in heap
36         Post newNode has been inserted into heap
37   */
38   void reheapUp (int* heap, int newNode)
39   {
40   // Local Declarations
41      int parent;
42      int hold;
43
44   // Statements
45      // if not at root of heap
46      if (newNode)
47         {
48          parent = (newNode - 1)/ 2;
49          if ( heap[newNode] > heap[parent] )
50             {
51              // child is greater than parent
52              hold          = heap[parent];
53              heap[parent]  = heap[newNode];
54              heap[newNode] = hold;
55              reheapUp (heap, parent);
56             } // if heap[]
57         } // if newNode
58      return;
59   }  // reheapUp
60
61   /* ==================== reheapDown ====================
62      Reestablishes heap by moving data in root down to its
63      correct location in the heap.
64         Pre  heap is an array of data
65              root is root of heap or subheap
66              last is an index to the last element in heap
67         Post heap has been restored
68   */
69   void reheapDown   (int* heap, int root, int last)
70   {
71   // Local Declarations
72      int hold;
```

continued

PROGRAM 12-2 Heap Sort *(continued)*

```
   int leftKey;
   int rightKey;
   int largeChildKey;
   int largeChildIndex;

// Statements
   if ((root * 2 + 1) <= last)
       // There is at least one child
      {
       leftKey   = heap[root * 2 + 1];
       if ((root * 2 + 2) <= last)
          rightKey  = heap[root * 2 + 2];
       else
          rightKey  = -1;

       // Determine which child is larger
       if (leftKey > rightKey)
          {
           largeChildKey   = leftKey;
           largeChildIndex = root * 2 + 1;
          } // if leftKey
       else
          {
           largeChildKey   = rightKey;
           largeChildIndex = root * 2 + 2;
          } // else
       // Test if root > larger subtree
       if (heap[root] < heap[largeChildIndex])
           {
            // parent < children
            hold       = heap[root];
            heap[root] = heap[largeChildIndex];
            heap[largeChildIndex] = hold;
            reheapDown (heap, largeChildIndex, last);
           } // if root <
   } // if root
   return;
}  // reheapDown
```

Program 12-2 Analysis

The heap sort implementation follows the pseudocode closely. Using an index, we walk through the array, calling reheap up to create the heap. Once the heap has been created, we exchange the element at the top of the heap with the last element in the heap and adjust the heap size down by 1. We then call reheap down to re-create the heap by moving the root element down the tree to its correct location.

12.3 Insertion Sorts

Insertion sorting is one of the most common sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand.[4] The concept extends well into computer sorting. In each pass of an insertion sort, one or more pieces of data are inserted into their correct location in an ordered list. In this section we study two insertion sorts: the straight insertion sort and the shell sort.

Straight Insertion Sort

In the **straight insertion sort**, the list is divided into two parts: sorted and unsorted. In each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place. If we have a list of n elements, it will take at most $n - 1$ passes to sort the data. This concept is shown in Figure 12-7. In this figure we have placed a visual wall between the sorted and the unsorted portions of the list.

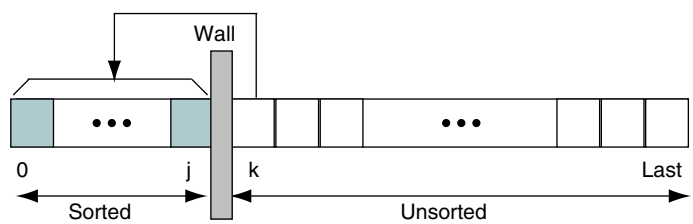


FIGURE 12-7 Insertion Sort Concept

Straight Insertion Sort Example

Figure 12-8 traces the insertion sort through a list of six numbers. Sorting these data requires five sort passes. Each pass moves the wall one element to the right as an element is removed from the unsorted sublist and inserted into the sorted sublist.

Insertion Sort Algorithm

The design of the insertion sort follows the pattern in the example. Each execution of the outer loop inserts the first element from the unsorted list into the sorted list. The inner loop steps through the sorted list, starting at the high end, looking for the correct insertion location. The pseudocode is shown in Algorithm 12-3.

4. Card sorting is an example of a sort that uses two pieces of data to sort: suit and rank.

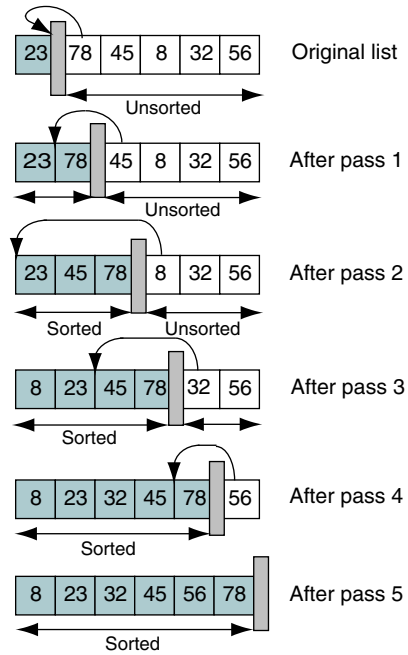


FIGURE 12-8 Insertion Sort Example

ALGORITHM 12-3 Straight Insertion Sort

```
Algorithm insertionSort (list, last)
Sort list array using insertion sort. The array is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list.
   Pre  list must contain at least one element
        last is an index to last element in the list
   Post list has been rearranged
 1 set current to 1
 2 loop (until last element sorted)
   1  move current element to hold
   2  set walker to current - 1
   3  loop (walker >= 0 AND hold key < walker key)
      1  move walker element right one element
      2  decrement walker
   4  end loop
   5  move hold to walker + 1 element
   6  increment current
 3 end loop
end insertionSort
```

Algorithm 12-3 Analysis

Two design concepts need to be explored in this simple algorithm. At some point in their execution, all sort algorithms must exchange two elements. Each exchange takes three statements, which can greatly impact the sort efficiency when many elements need to be exchanged. To improve the efficiency, therefore, split the exchange into different parts of the algorithm. The first step of the loop begins the exchange by moving to the hold area the data currently being sorted. This typical first move in any exchange is shown in statement 2.1. The inner loop (statement 2.3) shifts elements to the right until it finds the correct insertion location. Each of the shifts repeats the second step in an exchange. Finally, when the correct location is found, the hold area is moved back to the array (statement 2.5). This is the third step in the exchange logic and completes the exchange.

> In a straight insertion sort, the list at any moment is divided into sorted and unsorted sublists. In each pass the first element of the unsorted sublist is inserted into the sorted sublist.

Another point you should study is the workings of the inner loop. To make the sort as efficient as possible, we start with the high end of the sorted list and work toward the beginning of the sorted area. For the first insertion, this approach requires a maximum of one element to be shifted. For the last, it requires a minimum of zero and a maximum of $n - 1$ elements be shifted. The result is that only a portion of the list may be examined in each sort pass.

Shell Sort

The **shell sort** algorithm, named after its creator, Donald L. Shell, is an improved version of the straight insertion sort. It was one of the first fast sorting algorithms.[5]

> The shell sort is an improved version of the straight insertion sort in which diminishing partitions are used to sort the data.

In the shell sort, a list of N elements is divided into K segments, where K is known as the increment. Each segment contains a minimum of integral N/K; if N/K is not an integral, some of the segments will contain an extra element. Figure 12-9 contains a graphic representation of the segments in a shell sort. Note that the segments are dispersed throughout the list. In Figure 12-9 the increment is 3; the first, fourth, seventh, and tenth elements make up segment 1; the second, fifth, and eighth elements make up segment 2; and the third, sixth, and ninth elements make up segment 3. After each pass through the data, the data in each segment are ordered. Thus, when there are three segments, as we see in Figure 12-9, there are three different ordered lists. When there are two segments, there are two ordered lists; when there is only one segment, the list is sorted.

5. Shell's algorithm was first published in *Communications of the ACM*, vol. 2, no. 7 (1959): 30–32.

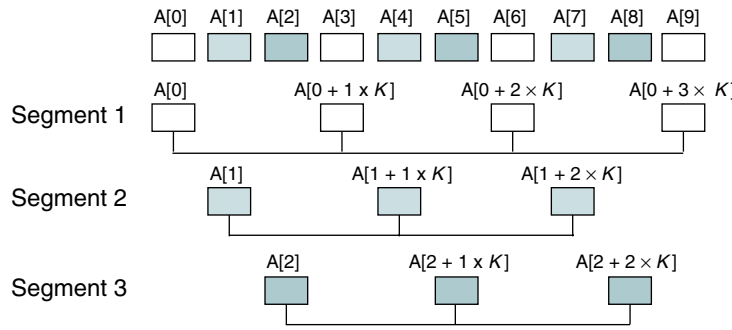


FIGURE 12-9 Segmented Array

Shell Sort Algorithm

Each pass through the data starts with the first element in the array and progresses through the array, comparing adjacent elements in each segment. In Figure 12-9 we begin by comparing elements A[0] and A[3] from the first segment, then A[1] and A[4] from the second segment, and then A[2] and A[5] from the third segment. We then compare A[3] and A[6] from the first segment, A[4] and A[7] from the second segment, and so forth until finally we compare elements A[6] and A[9]. If the elements are out of sequence, they are exchanged. Study Figure 12-9 carefully until you see each of these comparisons. Be sure to note also that the first segment has four elements, whereas the other two have only three. The number of elements in the segments varies because the list size (10) is not evenly divisible by the increment (3).

To compare adjacent keys in a segment, we add the increment to the current index, as shown below.

```
list[cur] : list[cur + incre]
```

After each pass through the data, the increment is reduced until, in the final pass, it is 1. Although the only absolute is that the last increment must be 1, the size of the increments influences the efficiency of the sort. We will discuss this issue separately later. The diminishing increment[6] is shown for an array of 10 elements and increments of 5, 2, and 1 in Figure 12-10.

6. Knuth gave this sort the name *diminishing increment sort*, but it is better known simply as the shell sort.

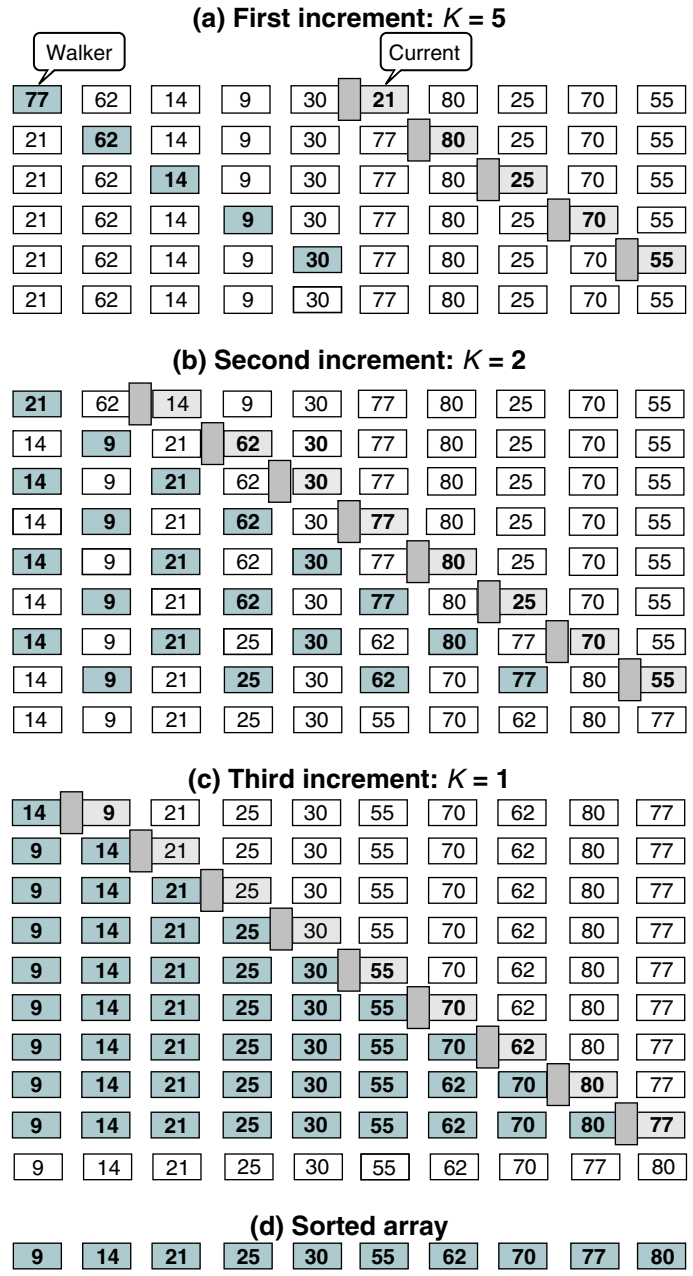


FIGURE 12-10 Diminishing Increments in Shell Sort

After each pass through the data, the elements in each segment are ordered. To ensure that each segment is ordered at the end of the pass, whenever an exchange is made we drop back one increment and test the adjacent elements. If they are out of sequence, we exchange them and drop back again. If necessary, we keep exchanging and dropping back until we find two elements are ordered. We now have all of the elements of the shell sort. Its pseudocode is shown in Algorithm 12-4.

ALGORITHM 12-4 Shell Sort

```
Algorithm shellSort (list, last)
Data in list array are sorted in
place. After the sort, their keys will be in order,
list[0] <= list[1] <= … <= list[last].
   Pre  list is an unordered array of records
        last is index to last record in array
   Post list is ordered on list[i].key
 1 set incre to last / 2
   Compare keys "increment" elements apart.
 2 loop (incre not 0)
   1  set current to incre
   2  loop (until last element sorted)
      1  move current element to hold
      2  set walker to current - incre
      3  loop (walker >= 0 AND hold key < walker key)
            Move larger element up in list.
         1  move walker element one increment right
            Fall back one partition.
         2  set walker to walker - incre
      4  end loop
         Insert hold record in proper relative location.
      5  move hold to walker + incre element
      6  increment current
   3  end loop
      End of pass--calculate next increment.
   4  set incre to incre / 2
 3 end loop
end shellSort
```

Algorithm 12-4 Analysis

Let's look at the shell sort carefully to see how it qualifies as an insertion sort. Recall that insertion sorts insert the new data into their correct location in the ordered portion of the list. This concept is found in the shell sort: the ordered portion of the list is a segment with its members separated by the increment size. To see this more clearly, look at Figure 12-11. In this figure the segment is shown as the colored elements in an array. The new element, $A[0 + 3 \times K]$, is being inserted into its correct position in the ordered portion of the segment.

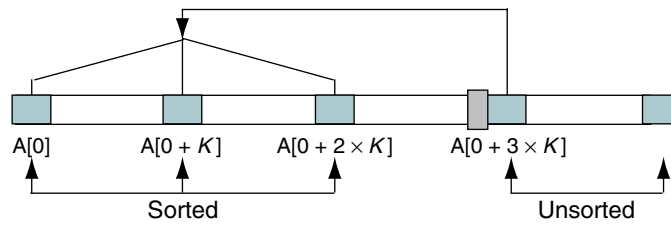


FIGURE 12-11 Ordered Segment in a Shell Sort

Furthermore, if you compare the code in Algorithm 12-4 with the code in Algorithm 12-3, "Straight Insertion Sort," you see that, other than the increment, the code is virtually identical.

One of the most important parts of the shell sort is falling back to ensure that the segment is ordered. This logic takes place in statement 2.2.3.2. If this logic is not included, the resulting list is not completely sorted.

Selecting the Increment Size

First, recognize that no increment size is best for all situations. The overriding considerations in the sort are to complete the sort with the minimum number of passes (increments) and to minimize the number of elements that appear in more than one segment. One method to ensure that an element is not in more than one segment is to use prime numbers. Unfortunately, the dynamic calculation of prime numbers is a relatively slow process.

Most texts use the simple series we proposed in Algorithm 12-4, setting the increment to half the list size and dividing by 2 on each pass. Knuth suggests, however, that we should not start with an increment greater than one-third of the list size.[7] Other computer scientists have suggested that the increments be a power of 2 minus 1 or a Fibonacci series. These variations may result in a slightly more efficient sort, but they are relatively complex. One simple variation of the division-by-2 approach is to add 1 whenever the increment is even. Doing so tends to reduce the number of elements that appear in multiple segments.

Although we can use more-complex increment-setting algorithms, the efficiency of a shell sort never approaches that of the quick sort discussed in the next section. Therefore, if the objective is to obtain the most efficient sort, the solution is to use the quick sort rather than try to optimize the increment size in the shell sort. On the other hand, the shell sort is a much simpler sort and at the same time is reasonably efficient.

7. Knuth, *Art of Computer Programming,* vol. 3, 94.

Insertion Sort Efficiency

Sort algorithms determine the sort effort for a given sort. Sort effort is defined as the relative efficiency of a sort. It can be determined in several ways, but we use the number of loops in the sort. Another common measure is the number of moves and comparisons needed to sort the list. Of course, the best measure is the time it takes to actually run the sort. Time, however, varies by the efficiency of the program implementation and the speed of the computer being used. For analyzing different sorts, therefore, the first two measures are more meaningful. Let's now analyze the straight insertion sort and the shell sort algorithms to determine their relative efficiency.

Straight Insertion Sort

Referring to Algorithm 12-3, "Straight Insertion Sort," we find the following two loops:

```
2   loop (until last element sorted)
    ...
    2   set walker to current - 1
    3   loop (walker >= 0 AND hold key < walker key)
```

The outer loop executes $n - 1$ times, from 1 through the last element in the list. For each outer loop, the inner loop executes from 0 to `current` times, depending on the relationship between the hold key and the walker key. On the average, we can expect the inner loop to process through the data in half of the sorted list. Because the inner loop depends on the setting for `current`, which is controlled by the outer loop, we have a dependent quadratic loop, which is mathematically stated as

$$f(n) = n\left(\frac{n+1}{2}\right)$$

In big-O notation the dependent quadratic loop is $O(n^2)$.

The straight insertion sort efficiency is $O(n^2)$.

Shell Sort

Now let's look at Algorithm 12-4, "Shell Sort." This algorithm contains the nested loops shown below:

```
2   loop (incre not 0)
    ...
    2   loop (until last element sorted)
        ...
        3 loop (walker >= 0 AND hold key < walker key)
```

Because we are dividing the increment by 2 in each loop, the outer loop is logarithmic; it is executed log n times. The first inner loop executes n – increment times for each of the outer loops; the first time it loops through 50% of the array ($n - (n / 2)$), the second time it loops through 75% of the array ($n - (n / 4)$), and so forth until it loops through all of the elements. The total number of iterations for the outer loop and the first inner loop is shown below:

$$\log n \times \left[\left(n - \frac{n}{2}\right) + \left(n - \frac{n}{4}\right) + \left(n - \frac{n}{8}\right) + \cdots + 1\right] = n \log n$$

The innermost loop is the most difficult to analyze. The first limit keeps us from falling off the beginning of the array. The second limit determines whether we have to loop at all: we loop only when the data are out of order. Sometimes the inner loop is executed zero times; sometimes it is executed anywhere from one to increment times. If we were able to derive a formula for the third factor, the total sort effort would be the product of the three loops. The first two loops have a combined efficiency of $O(n \log n)$. However, we still need to include the third loop. We can see, therefore, that the result is something greater than $O(n \log n)$.

Knuth[8] tells us that the sort effort for the shell sort cannot be derived mathematically. He estimates from his empirical studies that the average sort effort is $15n^{1.25}$. Reducing Knuth's analysis to a big-O notation, we see that the shell sort is $O(n^{1.25})$.

The shell sort efficiency is $O(n^{1.25})$.

Summary

Our analysis leads to the conclusion that the heap sort is more efficient than the other sorts we have discussed. The straight insertion and the straight selection sorts are both $O(n^2)$ sorts, the shell sort is $O(n^{1.25})$, and the heap sort is $O(n \log n)$. Table 12-2 offers a comparison of the sorts. Note that according to a strict mathematical interpretation of the big-O notation, heap sort surpasses shell sort in efficiency as we approach 2000 elements to be sorted. Remember two points, however: First, big-O is a rounded approximation; all coefficients and many of the factors have been removed. Second, big-O is based on an analytical evaluation of the algorithms, not an evaluation of the code. Depending on the algorithm's implementation, the actual run time can be affected.

8. Knuth, *Art of Computer Programming,* 382.

n	Number of loops		
	Straight insertion Straight selection	Shell	Heap
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	10,965

TABLE 12-2 Comparison of Insertion and Selection Sorts

Insertion Sort Implementation

In this section we write C implementations of the straight insertion sort and the shell sort.

Straight Insertion Sort

The straight insertion sort's implementation follows the pseudocode in Algorithm 12-3 very closely. To test the sort, we created an array of random integers. The code is shown in Program 12-3.

PROGRAM 12-3 Insertion Sort

```
 1  /* ================== insertionSort ===================
 2     Sort using Insertion Sort. The list is divided into
 3     sorted/unsorted list. In each pass, first element
 4     in unsorted list is inserted into sorted list.
 5        Pre  list must contain at least one element
 6             last contains index to last element in list
 7        Post list has been rearranged
 8  */
 9  void insertionSort (int list[], int last)
10  {
11  // Local Definitions
12     int hold;
13     int walker;
14
15  // Statements
16     for (int current = 1; current <= last; current++)
17        {
18         hold = list[current];
19         for (walker = current - 1;
```

continued

PROGRAM 12-3 Insertion Sort *(continued)*

```
20              walker >= 0 && hold < list[walker];
21              walker--)
22                 list[walker + 1] = list[walker];
23
24           list [walker + 1] = hold;
25        } // for current
26     return;
27  }  // insertionSort
```

Program 12-3 Analysis We implement both loops using a *for* statement. When we begin the sort, the sorted list contains the first element. Therefore, we set the loop start to position 1 for the unsorted list (statement 16). In the inner loop, the limiting condition is the beginning of the array, as shown in statement 20.

Shell Sort

The shell sort implementation, which also uses an array of integers, is shown in Program 12-4. Its design is shown in Algorithm 12-4.

PROGRAM 12-4 Shell Sort

```
 1  /* List[1], list[2], …, list[last] are sorted in place
 2     so that the keys are ordered, list[1].key <=
 3     list[2].key, <= … <= list[last].key.
 4        Pre  list is an unordered array of integers
 5             last is index to last element in array
 6        Post list is ordered
 7  */
 8  void shellSort (int list [], int last)
 9  {
10  // Local Definitions
11     int hold;
12     int incre;
13     int walker;
14
15  // Statements
16     incre = last / 2;
17     while (incre != 0)
18        {
19         for (int curr = incre; curr <= last; curr++)
20            {
21             hold = list [curr];
22             walker = curr - incre;
23             while (walker >= 0 && hold < list [walker])
24                {
25                 // Move larger element up in list
26                 list [walker + incre] = list [walker];
27                 // Fall back one partition
```

continued

PROGRAM 12-4 Shell Sort *(continued)*

```
28                walker = ( walker - incre );
29               } //  while
30             // Insert hold in proper position
31             list [walker + incre] = hold;
32            } // for walk
33          // End of pass--calculate next increment.
34          incre = incre / 2;
35        } // while
36    return;
37  } // shellSort
```

Program 12-4 Analysis Although more complex than the straight insertion sort, the shell sort is by no means a difficult algorithm. There are only two additional complexities over the straight insertion sort. First, rather than a single array, we have an array of partitions. Second, whenever an exchange is made, we must fall back and verify the order of the partition.

12.4 Exchange Sorts

The third category of sorts, **exchange sorts**, contains the most common sort taught in computer science—the bubble sort—and the most efficient general-purpose sort, quick sort. In exchange sorts we exchange elements that are out of order until the entire list is sorted. Although virtually every sorting method uses some form of exchange, the sorts in this section use it extensively.

Bubble Sort

In the **bubble sort**, the list at any moment is divided into two sublists: sorted and unsorted. The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist. After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the number of unsorted ones (Figure 12-12). Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed. Given a list of n elements, the bubble sort requires up to $n - 1$ passes to sort the data.

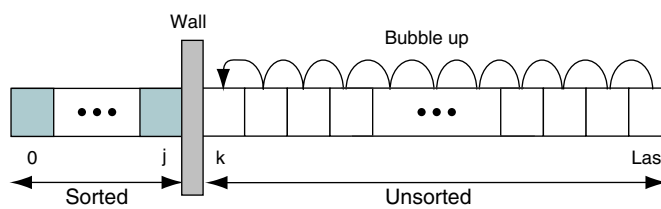


FIGURE 12-12 Bubble Sort Concept

Figure 12-13 shows how the wall moves one element in each pass. Looking at the first pass, we start with 32 and compare it with 56. Because 32 is less than 56, we exchange the two and step down one element. We then compare 32 and 8. Because 32 is not less than 8, we do not exchange these elements. We step down one element and compare 45 and 8. They are out of sequence, so we exchange them and step down again. Because we moved 8 down, it is now compared with 78, and these two elements are exchanged. Finally, 8 is compared with 23 and exchanged. This series of exchanges places 8 in the first location, and the wall is moved up one position.

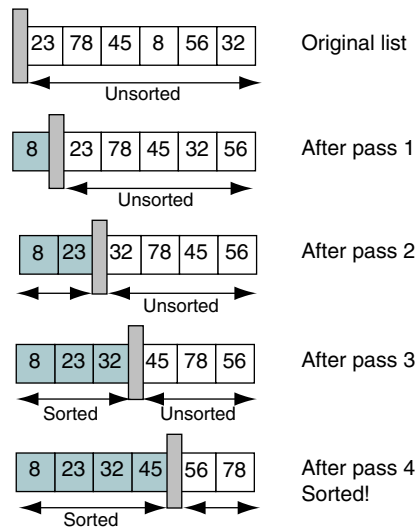


FIGURE 12-13 Bubble Sort Example

Bubble Sort Algorithm

Like the insertion and selection sorts, the bubble sort is quite simple. In each pass through the data, the smallest element is bubbled to the beginning of the unsorted segment of the array. In the process, adjacent elements that are out of order are exchanged, partially ordering the data. When the smallest element is encountered, it is automatically bubbled to the beginning of the unsorted list. The sort then continues by making another pass through the unsorted list. The code for the bubble sort is shown in Algorithm 12-5.

ALGORITHM 12-5 Bubble Sort

```
Algorithm  bubbleSort (list, last)
Sort an array using bubble sort. Adjacent
elements are compared and exchanged until list is
completely ordered.
```

continued

ALGORITHM 12-5 Bubble Sort *(continued)*

```
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged in sequence low to high
 1 set current to 0
 2 set sorted  to false
 3 loop (current <= last AND sorted false)
      Each iteration is one sort pass.
   1  set walker to last
   2  set sorted to true
   3  loop (walker > current)
      1  if (walker data < walker - 1 data)
            Any exchange means list is not sorted.
         1  set sorted to false
         2  exchange (list, walker, walker - 1)
      2  end if
      3  decrement walker
   4  end loop
   5  increment current
 4 end loop
end bubbleSort
```

Algorithm 12-5 Analysis

If you have studied other bubble sort algorithms, you may have noticed a slight improvement in this version of the sort. If an exchange is not made in a pass (statement 3), we know that the list is already sorted and the sort can stop. At statement 3.2 we set a Boolean, **`sorted`**, to true. If at any time during the pass an exchange is made, **`sorted`** is changed to false, indicating that the list was not sorted when the pass began.

Another difference you may have noticed is that we started from the high end and bubbled down. As a historical note, the bubble sort was originally written to "bubble up" the highest element in the list. From an efficiency point of view, it makes no difference whether the largest element is bubbled down or the smallest element is bubbled up. From a consistency point of view, however, comparisons between the sorts are easier if all three of our basic sorts (insertion, selection, and exchange) work in the same manner. For that reason we have chosen to bubble the lowest key in each pass.

Quick Sort

In the bubble sort, consecutive items are compared and possibly exchanged on each pass through the list, which means that many exchanges may be needed to move an element to its correct position. **Quick sort** is an exchange sort developed by C. A. R. Hoare in 1962. It is more efficient than the bubble sort because a typical exchange involves elements that are far apart, so fewer exchanges are required to correctly position an element.

Each iteration of the quick sort selects an element, known as **pivot**, and divides the list into three groups: a partition of elements whose keys are less than the pivot's key, the pivot element that is placed in its ultimately correct location in the list, and a partition of elements greater than or equal to the pivot's key. The sorting then continues by quick sorting the left

partition followed by quick sorting the right partition. This partitioning is shown in Figure 12-14.

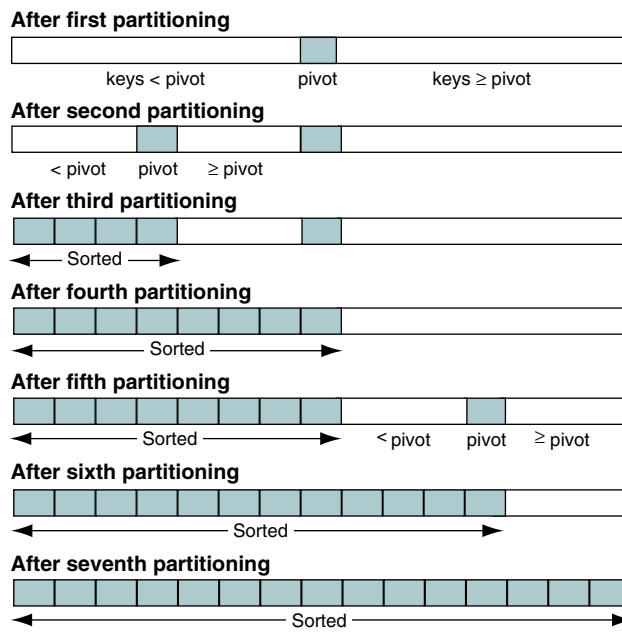


FIGURE 12-14 Quick Sort Partitions

Hoare's original algorithm selected the pivot key as the first element in the list. In 1969, R. C. Singleton improved the sort by selecting the pivot key as the median value of three elements: left, right, and an element in the middle of the list. Once the median value is determined, it is exchanged with the left element. We implement Singleton's variation of quick sort.

Quick sort is an exchange sort in which a pivot key is placed in its correct position in the array while rearranging other elements widely dispersed across the list.

Knuth suggests that when the sort partition becomes small a straight insertion sort be used to complete the sorting of the partition.[9] Although the mathematics to optimally choose the minimum partition size are quite complex, the partition size should be relatively small; we recommend 16.

9. Knuth, *Art of Computer Programming*, "Algorithm Q," 115.

Quick Sort Algorithm

We are now ready to develop the algorithm. As you may have anticipated from our discussion, we use a recursive algorithm for the quick sort. In addition to the basic algorithm, two supporting algorithms are required, one to determine the pivot element and one for the straight insertion sort. We discuss these two algorithms first.

Straight Insertion Module

The straight insertion sort is a minor variation on the algorithm developed earlier, which always sorted a list beginning at location 0. Because the partition to be sorted in the quick sort can be found anywhere in the array, we must be able to sort partitions beginning at locations other than 0. We therefore add a parameter that specifies the starting location of the location to be sorted in addition to its ending location. The modified code is shown in Algorithm 12-6.

ALGORITHM 12-6 Quick Sort's Straight Insertion Sort Module

```
Algorithm quickInsertion (list, first, last)
Sort array list using insertion sort. The list is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list. This is a special version of the insertion
sort modified for use with quick sort.
   Pre  list must contain at least one element
        first is an index to first element in the list
        last is an index to last element in the list
   Post list has been rearranged
 1 set current to first + 1
 2 loop (until current partition sorted)
   1  move current element to hold
   2  set walker to current - 1
   3  loop (walker >= first AND hold key < walker key)
      1  move walker element one element right
      2  decrement walker
   4  end loop
   5  move hold to walker + 1 element
   6  increment current
 3 end loop
end quickInsertion
```

Determine Median of Three

The logic to select the median location requires three tests. First we test the left and middle elements; if they are out of sequence, we exchange them. Then we test the left and right elements; if they are out of sequence, we exchange them. Finally, we test the middle and right elements; if they are out of sequence, we exchange them. At this point the three elements are in order.

```
left element <= middle element <= right element
```

We see, therefore, that this logic is based on the logical proposition that if *a* is less than *b,* and *b* is less than *c*, then *a* is less than *c*. Finally, before we leave the algorithm we exchange the left and middle elements, thus positioning the median valued element at the left of the array. The pseudocode is shown in Algorithm 12-7.

ALGORITHM 12-7 Median Left

```
Algorithm medianLeft (sortData, left, right)
Find the median value of an array and place it in the first
(left) location.
   Pre   sortData is an array of at least three elements
         left and right are the boundaries of the array
   Post  median value located and placed left
   Rearrange sortData so median value is in middle location.
1  set mid to (left + right) / 2
2  if (left key > mid key)
   1   exchange (sortData, left, mid)
3  end if
4  if (left key > right key)
   1   exchange (sortData, left, right)
5  end if
6  if (mid key > right key)
   1   exchange (sortData, mid, right)
7  end if
   Median is in middle location. Exchange with left.
8  exchange (sortData, left, mid)
end medianLeft
```

Algorithm 12-7 Analysis

The logic to determine a median value can become unintelligible very quickly. The beauty of this algorithm is its simplicity. It approaches the determination of the median value by performing a very simple sort on the three elements, placing the median in the middle location. It then exchanges the middle element with the left element.

Quick Sort Algorithm

We now turn our attention to the quick sort algorithm itself. It contains an interesting program design that you will find useful in other array applications. To determine the correct position for the pivot element, we work from the two ends of the array toward the middle. Because we have used the median value of three elements to determine the pivot element, the pivot may end up near the middle of the array, although this is not guaranteed. Quick sort is most efficient when the pivot's location is the middle of the array. The technique of working from the ends to the middle is shown in Figure 12-15.

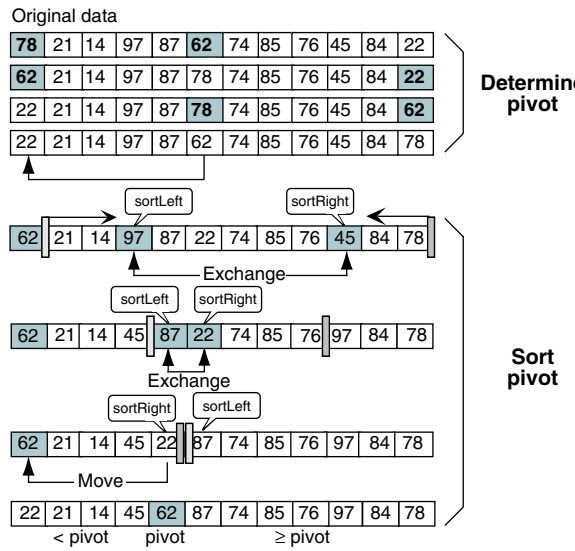


FIGURE 12-15 Quick Sort Pivot

As you study Figure 12-15, note that before the exchanges start, the median element is in the middle position and the smallest of the three elements used to determine the median is in the right location. After calling the median left algorithm, the median is in the left position and the smallest is in the middle location. The pivot key is then moved to a hold area to facilitate the processing. This move is actually the first part of an exchange that puts the pivot key in its correct location.

To help follow the sort, envision two walls, one on the left just after the pivot and one on the right (see Figure 12-15). We start at the left wall and move right while looking for an element that belongs on the right of the pivot. We locate one at element 97. After finding an element on the left that belongs on the right, we start at the right wall and move left while looking for an element that belongs on the left of the pivot. We find one at element 45. At this point we exchange the two elements, move the walls to the left of element 87 and the right of element 76, and repeat the process.

This time, as we move to the right, the first left element, 87, belongs on the right. When we move down from the right, we find that element 22 belongs on the left. Again, we exchange the left and right elements and move the wall. At this point we note that the walls have crossed. We have thus located the correct position for the pivot element. We move the data in the pivot's location to the first element of the array and then move the pivot back to the array, completing the exchange we started when we moved the pivot key to the hold area. The list has now been rearranged so that the pivot element (62) is in its correct location in the array relative to all of the other

elements in the array. The elements on the left of the pivot are all less than the pivot, and the elements on the right of the pivot are all greater than the pivot. Any equal data is found on the right of the pivot.

After placing the pivot key in its correct location, we recursively call quick sort to sort the left partition. When the left partition is completely sorted, we recursively call quick sort to sort the right partition. When both the left partition and the right partition have been sorted, the list is completely sorted. The data in Figure 12-15 are completely sorted using only quick sort in Figure 12-16.

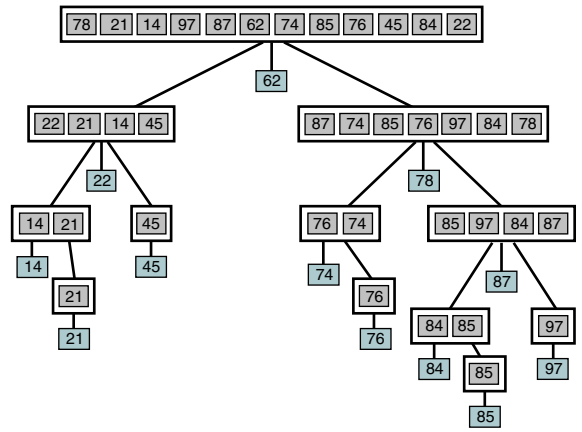


FIGURE 12-16 Quick Sort Operation

The quick sort pseudocode is shown in Algorithm 12-8.

ALGORITHM 12-8 Quick Sort

```
Algorithm quickSort (list, left, right)
An array, list, is sorted using recursion.
   Pre  list is an array of data to be sorted
        left and right identify the first and last
        elements of the list, respectively
   Post list is sorted
 1 if ((right - left) > minSize)
      Quick sort
   1  medianLeft (list, left, right)
   2  set pivot     to left element
   3  set sortLeft  to left + 1
   4  set sortRight to right
   5  loop (sortLeft    <= sortRight)
         Find key on left that belongs on right
```

continued

ALGORITHM 12-8 Quick Sort *(continued)*

```
      1  loop (sortLeft key < pivot key)
         1  increment sortLeft
      2  end loop
         Find key on right that belongs on left
      3  loop (sortRight key >= pivot key)
         1  decrement sortRight
      4  end loop
      5  if (sortLeft <= sortRight)
         1  exchange(list, sortLeft, sortRight)
         2  increment sortLeft
         3  decrement sortRight
      6  end if
   6  end loop
      Prepare for next pass
   7  move sortLeft - 1 element to left element
   8  move pivot element to sortLeft - 1 element
   9  if  (left < sortRight)
      1  quickSort (list, left, sortRight - 1)
  10  end if
  11  if (sortLeft < right)
      1  quickSort (list, sortLeft, right)
  12  end if
 2 else
   1  insertionSort (list, left, right)
 3 end if
end quickSort
```

Algorithm 12-8 Analysis

In addition to the design that works from both ends to the middle, two aspects of this algorithm merit further discussion. The loops used to determine the sort left and sort right elements (statements 1.5.1 and 1.5.3) test only one condition. Most loops that process an array must also test for the end of the array, but we can omit the test in this algorithm based on selection of the pivot key. The pivot key is guaranteed to be the median value of three elements: the first, the last, and the one in the middle. Therefore, the median key cannot be less than the far-left key nor greater than the far-right key. At worst it is equal to the far-left or far-right key.

Assume that we have the worst case: all of the elements in a partition have the same key value. In this case the pivot key is equal to the far-left and the far-right keys. When we start on the left to move up the list, we stop immediately because the pivot key is equal to the **`sortLeft`** key. When we then start on the right and move down, we move beyond the beginning of the list when all keys are equal. However, because the two elements have crossed, we do not exchange elements. Rather, we move to statement 1.7. In this case we never use the sort right index that has moved off the beginning of the array.

Hoare's original algorithm was not recursive. Because he had to maintain a stack, he incorporated logic to ensure that the stack size was kept to a minimum. Rather than simply sort the left partition, he determined which partition was larger and put it in the stack while he sorted the smaller partition. We are not concerned with minimizing the stack size for two reasons. First, we have chosen the pivot key to be the median value.

Therefore, the size of the two partitions should be generally the same, thus minimizing the number of recursive calls. More important, because we are using recursion, we do not need to determine the size of the stack in advance. We simply call on the system to provide stack space.

Exchange Sort Efficiency

In the exchange sorts, we find what Knuth called the best general-purpose sort: quick sort. Let's determine their sort efforts to see why.

Bubble Sort

The code for the bubble sort is shown in Algorithm 12-5, "Bubble Sort." As we saw with the straight insertion and the straight selection sorts, it uses two loops to sort the data. The loops are shown below:

```
3   loop (current <= last AND sorted false)
    ...
    3   loop (walker > current)
```

The outer loop tests two conditions: the current index and a sorted flag. Assuming that the list is not sorted until the last pass, we loop through the array n times. The number of loops in the inner loop depends on the current location in the outer loop. It therefore loops through half the list on the average. The total number of loops is the product of both loops, making the bubble sort efficiency

$$n\left(\frac{n+1}{2}\right)$$

In big-O notation the bubble sort efficiency is $O(n^2)$.

The bubble sort efficiency is $O(n^2)$.

Quick Sort

The code for the quick sort is shown in Algorithm 12-8, "Quick Sort." A quick look at the algorithm reveals that there are five loops (three iterative loops and two recursive loops). The algorithm also contains the straight insertion sort as a subroutine. Because it is possible to use the quick sort to completely sort the data—that is, it is possible to eliminate the insertion sort—we analyze the quick sort portion as though the insertion sort weren't used. The quick sort loops are shown below:

```
    5  loop (sortLeft <= sortRight)
       1   loop (sortLeft key < pivot key)
           ...
```

continued

```
      2   end loop
      3   loop (sortRight key >= pivot key)
          ...
      4   end loop
      5   ...
   6  end loop
      ...
   9  if (left < sortRight)
      1   quickSort (list, left, sortRight - 1)
  10  end if
  11  if (sortLeft < right)
      1   quickSort (list, sortLeft, right)
  12  if (sortLeft < right)
```

Recall that each pass in the quick sort divides the list into three parts: a list of elements smaller than the pivot key, the pivot key, and a list of elements greater than the pivot key. The first loop (statement 5), in conjunction with the two nested loops (statements 5.1 and 5.3), looks at each element in the portion of the array being sorted. Statement 5.1 loops through the left portion of the list; statement 5.3 loops through the right portion of the list. Together, therefore, they loop through the list n times.

Similarly, the two recursive loops process one portion of the array each, either on the left or the right of the pivot key. The question is how many times they are called. Recall that we said that quick sort is most efficient when the pivot key is in the middle of the array. That's why we used a median value. Assuming that it is located relatively close to the center, we see that we have divided the list into two sublists of roughly the same size. Because we are dividing by 2, the number of loops is logarithmic. The total sort effort is therefore the product of the first loop times the recursive loops, or $n \log n$.

The quick sort efficiency is $O(n \log n)$.

Algorithmics does not explain why we use the straight insertion sort when the list is small. The answer lies in the algorithm code and the overhead of recursion. When the list becomes sufficiently small, it is simply more efficient to use a straight insertion sort.

Summary

Our analysis leads to the conclusion that the quick sort's efficiency is the same as that of the heap sort. This is true because big-O notation is only an approximation of the actual sort efficiency. Although both are on the order of $n \log n$, if we were to develop more-accurate formulas to reflect their actual efficiency, we would see that the quick sort is actually more efficient. Table 12-3 summarizes the six sorts we discuss in this chapter.

n	Number of loops		
	Straight insertion Straight selection Bubble sorts	Shell	Heap and quick
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	10,965

TABLE 12-3 Sort Comparisons

We recommend that you use the straight insertion sort for small lists and the quick sort for large lists. Although the shell sort is an interesting sort that played an important role in the history of sorts, we would not recommend it in most cases. As you will see when we discuss external sorting, algorithms such as the heap sort play an important niche role in special situations and for that reason also belong in your sorting tool kit.

Exchange Sort Implementation

Let's look at the C code for the bubble sort and the quick sort algorithms.

Bubble Sort Code

The bubble sort code is shown in Program 12-5.

PROGRAM 12-5 Bubble Sort

```
 1  /* ================== bubbleSort ====================
 2     Sort list using bubble sort. Adjacent elements are
 3     compared and exchanged until list is ordered.
 4        Pre  list must contain at least one item
 5             last contains index to last list element
 6        Post list has been sorted in ascending sequence
 7  */
 8  void bubbleSort (int list [], int last)
 9  {
10  // Local Definitions
11     int temp;
12
```

continued

PROGRAM 12-5 Bubble Sort *(continued)*

```
// Statements
   // Each iteration is one sort pass
   for (int current = 0, sorted = 0;
            current <= last && !sorted;
            current++)
        for (int walker = last, sorted = 1;
                 walker > current;
                 walker--)
          if (list[ walker ]  < list[ walker - 1 ])
              // Any exchange means list is not sorted
              {
               sorted = 0;
               temp             = list[walker];
               list[walker]     = list[walker - 1];
               list[walker - 1] = temp;
              } // if
   return;
}  // bubbleSort
```

Quick Sort Code

In this section we implement the three quick sort algorithms described earlier. The sort array is once again a simple array of integers.

The first function is quick sort, shown in Program 12-6. The modified version of the straight insertion sort is shown in Program 12-7, and the median function is shown in Program 12-8.

PROGRAM 12-6 Quick Sort

```
/* ==================== quickSort ====================
   Array, sortData[left..right] sorted using recursion.
      Pre  sortData is an array of data to be sorted
           left identifies first element of sortData
           right identifies last element of sortData
      Post sortData array is sorted
*/
void quickSort (int sortData[ ], int left, int right)
{
#define MIN_SIZE 16

// Local Definitions
int sortLeft;
int sortRight;
int pivot;
int hold;

```

continued

PROGRAM 12-6 Quick Sort *(continued)*

```
18  // Statements
19     if ((right - left) > MIN_SIZE)
20       {
21        medianLeft (sortData, left, right);
22        pivot     = sortData [left];
23        sortLeft  = left + 1;
24        sortRight = right;
25        while (sortLeft <= sortRight)
26          {
27           // Find key on left that belongs on right
28           while (sortData [sortLeft] < pivot)
29              sortLeft = sortLeft + 1;
30           // Find key on right that belongs on left
31           while (sortData[sortRight] >= pivot)
32              sortRight = sortRight - 1;
33           if (sortLeft <= sortRight)
34             {
35              hold                = sortData[sortLeft];
36              sortData[sortLeft]  = sortData[sortRight];
37              sortData[sortRight] = hold;
38              sortLeft            = sortLeft  + 1;
39              sortRight           = sortRight - 1;
40             } // if
41          } // while
42        // Prepare for next pass
43        sortData [left]         = sortData [sortLeft - 1];
44        sortData [sortLeft - 1] = pivot;
45        if (left < sortRight)
46            quickSort (sortData, left, sortRight - 1);
47        if (sortLeft < right)
48            quickSort (sortData, sortLeft, right);
49       } // if right
50     else
51       quickInsertion (sortData, left, right);
52      return;
53  }  // quickSort
```

PROGRAM 12-7 Modified Straight Insertion Sort for Quick Sort

```
1  /* ================ quickInsertion ================
2     Sort data[1…last] using insertion sort. Data is
3     divided into sorted and unsorted lists. With each
4     pass, the first element in unsorted list is inserted
5     into the sorted list. This is a special version of the
6     insertion sort modified for use with quick sort.
7        Pre  data must contain at least one element
8             first is index to first element in data
```

continued

PROGRAM 12-7 Modified Straight Insertion Sort for Quick Sort *(continued)*

```
 9              last is index to last element in data
10         Post data has been rearranged
11   */
12   void quickInsertion (int data[], int first, int last)
13   {
14   // Local Definitions
15      int hold;
16      int walker;
17
18   // Statements
19      for (int current = first + 1;
20               current <= last;
21               current++)
22         {
23          hold   = data[current];
24          walker = current - 1;
25          while (walker >= first
26              && hold < data[walker])
27                {
28                 data[walker + 1] = data[walker];
29                 walker = walker - 1;
30                } // while
31          data[walker + 1] = hold;
32         } // for
33      return;
34   }  // quickInsertion
```

PROGRAM 12-8 Median Left for Quick Sort

```
 1   /* ================== medianLeft ====================
 2      Find the median value of an array,
 3      sortData[left..right], and place it in the
 4      location sortData[left].
 5         Pre  sortData is array of at least three elements
 6              left and right are boundaries of array
 7         Post median value placed at sortData[left]
 8   */
 9   void medianLeft (int sortData[], int left, int right)
10   {
11   // Local Definitions
12      int mid;
13      int hold;
14
15   // Statements
16      // Rearrange sortData so median is middle location
17      mid = (left + right) / 2;
```

continued

PROGRAM 12-8 Median Left for Quick Sort *(continued)*

```
18       if (sortData[left] > sortData[mid])
19          {
20           hold            = sortData[left];
21           sortData[left]  = sortData[mid];
22           sortData[mid]   = hold;
23          } // if
24       if (sortData[left] > sortData[right])
25          {
26           hold            = sortData[left];
27           sortData[left]  = sortData[right];
28           sortData[right] = hold;
29          } // if
30       if (sortData[mid]  > sortData[right])
31          {
32           hold            = sortData[mid];
33           sortData[mid]   = sortData[right];
34           sortData[right] = hold;
35          } // if
36
37       // Median is in middle. Exchange with left
38       hold           = sortData[left];
39       sortData[left] = sortData[mid];
40       sortData[mid]  = hold;
41
42       return;
43    }  // medianLeft
```

12.5 External Sorts

All of the algorithms we have studied so far have been internal sorts—that is, sorts that require all of the data to be in primary memory during the sorting process. We now turn our attention to external sorting—sorts that allow portions of the data to be stored in secondary memory during the sorting process.

The term *external sort* is somewhat of a misnomer. Most of the work spent ordering large files is not sorting but actually merging. As we will see, external sorts begin by sorting blocks of data internally and writing them to files. Once all of the data have been through the internal sort phase, the building of one completely sorted file is done through merging files. To understand external sorting, therefore, we must first understand the merge concept.

Merging Ordered Files

A **merge** is the process that combines two files sorted on a given key into one sorted file on the same given key. A simple example is shown in Figure 12-17.

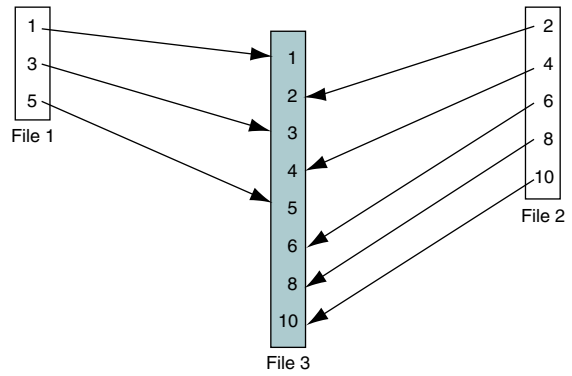


FIGURE 12-17 Simple Merge

File 1 and file 2 are to be merged into file 3. To merge the files, we compare the first record in file 1 with the first record in file 2 and write the smaller one, 1, to file 3. We then compare the second record in file 1 with the first record in file 2 and write the smaller one, 2, to file 3. The process continues until all data have been merged in order into file 3. The logic to merge these two files, which is relatively simple, is shown in Algorithm 12-9.

ALGORITHM 12-9 Merge Files

```
Algorithm mergeFiles
Merge two sorted files into one file.
   Pre    input files are sorted
   Post   input files sequentially combined in output file
1  open files
2  read (file1 into record1)
3  read (file2 into record2)
4  loop (not end file1 OR not end file2)
   1   if (record1 key <= record2 key)
       1  write  (record1 to file3)
       2  read   (file1 into record1)
       3  if (end of file1)
          1  set record1 key to infinity
       4  end if
   2   else
       1  write  (record2 to file3)
       2  read   (file2 into record2)
       3  if (end of file2)
          1  set record2 key to infinity
       4  end if
   3   end if
```

continued

ALGORITHM 12-9 Merge Files *(continued)*

```
 5  end loop
 6  close files
end mergeFiles
```

Algorithm 12-9 Analysis

Although merge files is a relatively simple algorithm, one point is worth discussing. When one file reaches the end, there may be more data in the second file. We therefore need to keep processing until both files are at the end. We could write separate blocks of code to handle the end-of-file processing for each file, but there is a simpler method. When one file hits its end, we simply set its key value to an artificially high value. In the algorithm this value is identified as infinity (see statements 4.1.3.1 and 4.2.3.1). When we compare the two keys (statement 4.1), the file at the end is forced high and the other file is processed. The high value is often called a **sentinel**. The only limitation to this logic is that the sentinel value cannot be a valid data value.

Merging Unordered Files

In a **merge sort**, however, we usually have a different situation than that shown in Algorithm 12-9. Because the files are unsorted, the data runs in sequence, and then there is a sequence break followed by another series of data in sequence. This situation is demonstrated in Figure 12-18.

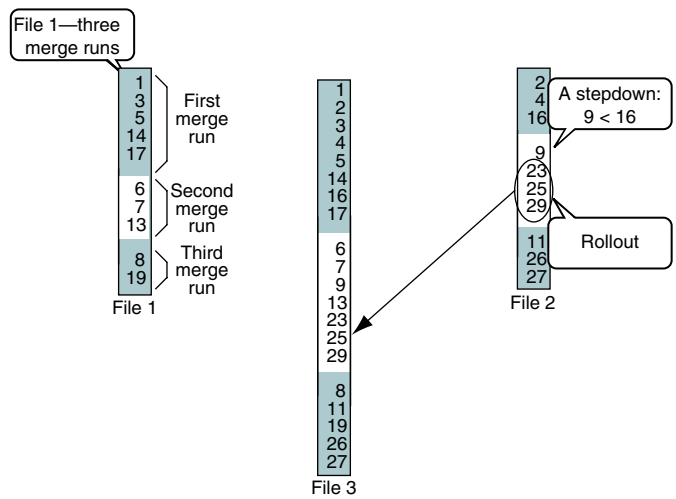


FIGURE 12-18 Merging Files Example

The series of consecutively ordered data in a file is known as a **merge run**. In Figure 12-18 all three files have three merge runs. To make them easy to see, we have spaced and colored the merge runs in each file. A **stepdown** occurs

when the sequential ordering of the records in a merge file is broken. The end of each merge run is identified by a stepdown. For example, in Figure 12-18 there is a stepdown in file 2 when key 16 in the first merge run is followed by key 9 in the second merge run. An end of file is also considered a stepdown.

In the merging of a file, the process of copying a consecutive series of records to the merge output file after a stepdown in the alternate merge input is known as a **rollout**. In Figure 12-18, after record 13 has been copied to the output file and a stepdown occurs, the remaining records (23, 25, and 29) in the second merge run in file 2 are rolled out to the merge output.

Now let's look closely at the merge process in Figure 12-18. When merging files that are not completely ordered, we merge the corresponding merge runs from each file into a merge run in the output file. Thus, we see that merge run 1 in file 1 merges with merge run 1 in file 2 to produce merge run 1 in file 3. Similarly, merge runs 2 and 3 in the input files merge to produce merge runs 2 and 3 in the output.

When a stepdown is detected in an input merge file, the merge run in the alternate file must be rolled out to synchronize the two files. Thus, in merge run 2, when the stepdown between record 13 and record 8 is detected, we must roll out the remaining three records in file 2 so that we can begin merging the third merge runs.

Finally, it is important to note that the merge output is not a completely ordered file. In this particular case, two more merge runs are required. To see the complete process, we turn to a higher-level example.

The Sorting Process

Assume that a file of 2300 records needs to be sorted. In an external sort, we begin by sorting as many records as possible and creating two or more merge files. Assuming that the record size and the memory available for our sort program allow a maximum sort array size of 500 records, we begin by reading and sorting the first 500 records and writing them to a merge output file. As we sort the first 500 records, we keep the remaining 1800 records in secondary storage. After writing out the first merge run, we read the second 500 records, sort them, and write them to an alternate merge output file. We continue the sort process, writing 500 sorted records (records 1001 to 1500) to the first merge output file and another 500 sorted records (records 1501 to 2000) to the second merge output file. Finally, we sort the last 300 records and write them to the first output merge file. At this point we have created the situation we see in Figure 12-19. This first processing of the data into merge runs is known as the **sort phase**.

After completing the sort phase of an external sort, we proceed with the merge phase. Each complete reading and merging of the input merge files to one or more output merge files is considered a separate **merge phase**. Depending on how many merge runs are created, there are zero or more merge phases. If all of the data fit into memory at one time, or if the file was sorted to begin with, there is only one merge run and the data on the first merge output file

are completely sorted. This situation is the exception, however; several merge phases are generally required.

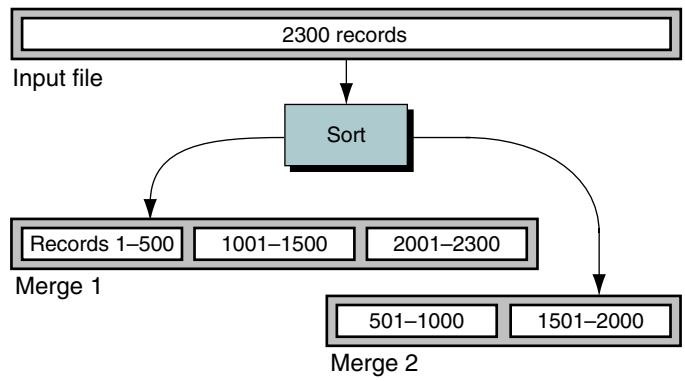


FIGURE 12-19 Sort Phase in an External Sort

Computer scientists have developed many different merge concepts over the years. We present three that are representative: natural merge, balanced merge, and polyphase merge.

Natural Merge

A **natural merge** sorts a constant number of input merge files to one merge output file. Between each merge phase, a **distribution phase** is required to redistribute the merge runs to the input files for remerging. Figure 12-20 is a diagram of our 2300 records as they would be sorted using a natural two-way merge—that is, a natural merge with two input merge files and one output merge file. To diagram the process, we use the classic symbol for a stencil file: the tape symbol. Recognize, however, that the data can be on either a tape file or a disk file. The only requirement is that it must be a sequential file.

In the natural merge, all merge runs are written to one file. Therefore, unless the file is completely ordered, the merge runs must be distributed to two merge files between each merge phase. This processing is very inefficient, especially because reading and writing records are among the slowest of all data processing. The question, therefore, is how can we make the merge process more efficient. The answer is found in the balanced merge.

In the natural merge, each phase merges a constant number of input files into one output file.

Balanced Merge

A **balanced merge** uses a constant number of input merge files and the same number of output merge files. Any number of merge files can be used,

although more than four is uncommon. Because multiple merge files are created in each merge phase, no distribution phase is required. Figure 12-21 sorts our 2300 records using a balanced two-way merge.

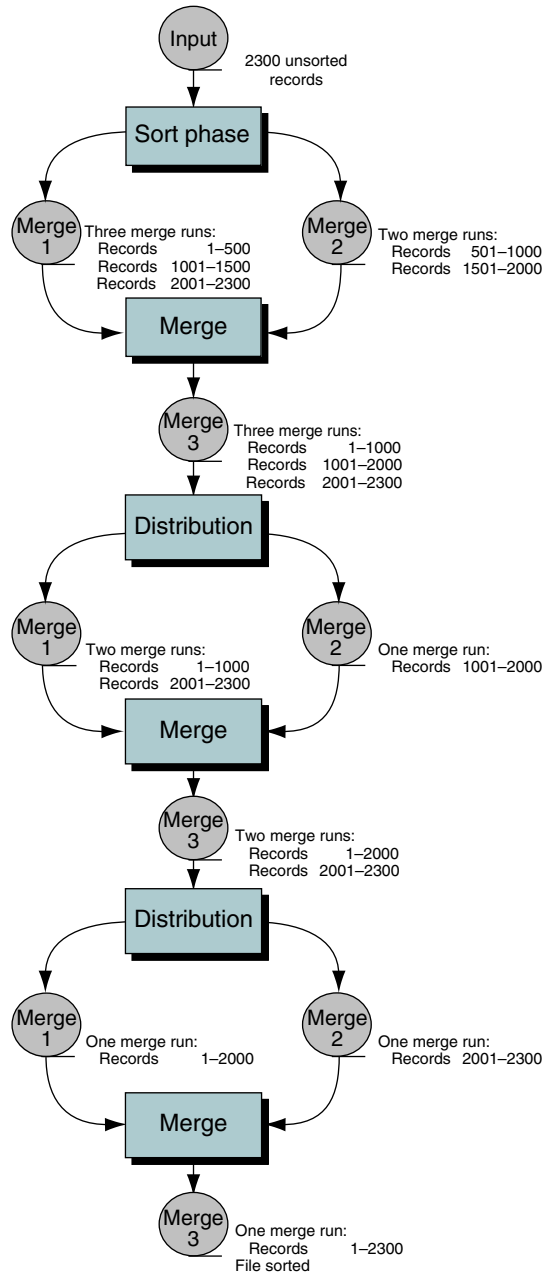


FIGURE 12-20 Natural Two-way Merge Sort

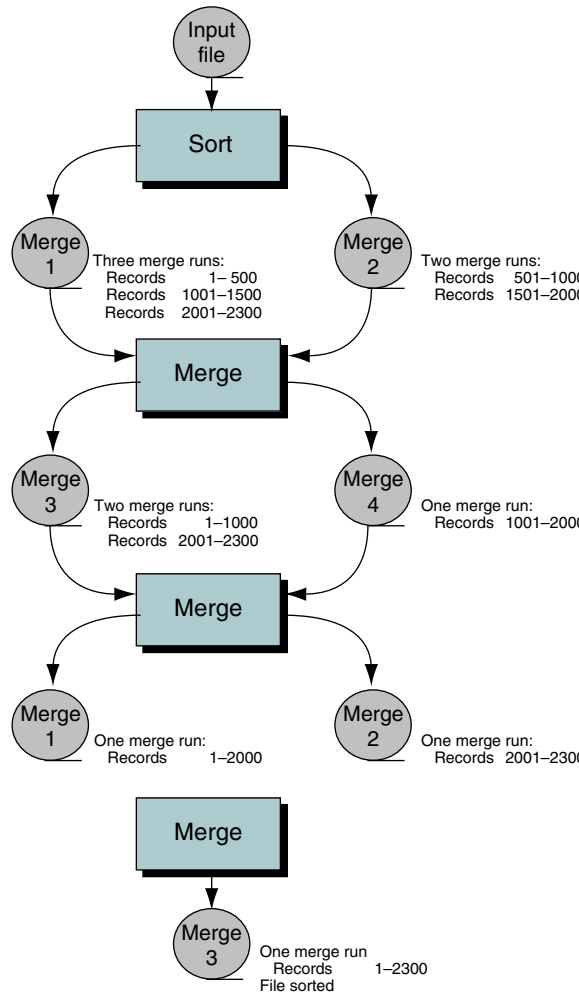


FIGURE 12-21 Balanced Two-way Merge Sort

Four merge files are required in the balanced two-way merge. The first merge phase merges the first merge run on file 1 with the first merge run on file 2 and writes it to file 3. It then merges the second merge run on file 1 with the second merge run on file 2 and writes it to file 4. At this point all of the merge runs on file 2 have been processed, so we roll out the remaining merge run on file 1 to merge file 3. This rollout of 300 records is wasted effort. We want to eliminate this step and make the merge processing as efficient as possible. We can if we use the polyphase merge.

The balanced merge eliminates the distribution phase by using the same number of input and output merge files.

Polyphase Merge

In the **polyphase merge**, a constant number of input merge files are merged to one output merge file, and input merge files are immediately reused when their input has been completely merged. Polyphase merge is the most complex of the merge sorts we have discussed.

In the polyphase merge, a constant number of input files are merged to one output file. As the data in each input file are completely merged, it immediately becomes the output file and what was the output file becomes an input file.

To demonstrate the process, study Figure 12-22 carefully. The processing begins as it does for the natural two-way merge. The first merge run on file 1 is merged with the first merge run on file 2. Then the second merge run on file 1 is merged with the second merge run on file 2. At this point merge 2 is empty and the first merge phase is complete. We therefore close merge 2 and open it as output and close merge 3 and open it as input. The third merge run on file 1 is then merged with the first merge run on file 3, with the merged data being written to merge 2. Because merge 1 is empty, merge phase 2 is complete. We therefore close merge 1 and open it as output while closing merge 2 and opening it as input. Because there is only one merge run on each of the input files, the sort is complete when these two merge runs have been merged to merge 1.

Sort Phase Revisited

With the polyphase sort, we have improved the merge phases as much as possible. We now return to the sort phase and try to improve it. Let's assume that we used the fastest possible sort in the sort phase. Even if we were able to double its internal sort speed, little would be gained. With today's modern computers operating in picosecond speeds and file processing operating in microsecond speeds, little is gained by improving the sort speed. In fact, the opposite is actually true. If we slow up the sort speed a little using a slower sort, we can actually improve overall speed. Let's see how this is done.

One class of sorts, tree sorts, allows us to start writing data to a merge file before the sort is complete. By using tree sorts, then, we can write longer merge runs, which reduces the number of merge runs and therefore speeds up the sorting process. Because we have studied one tree sort, the heap sort, let's see how it can be used to write longer merge runs.

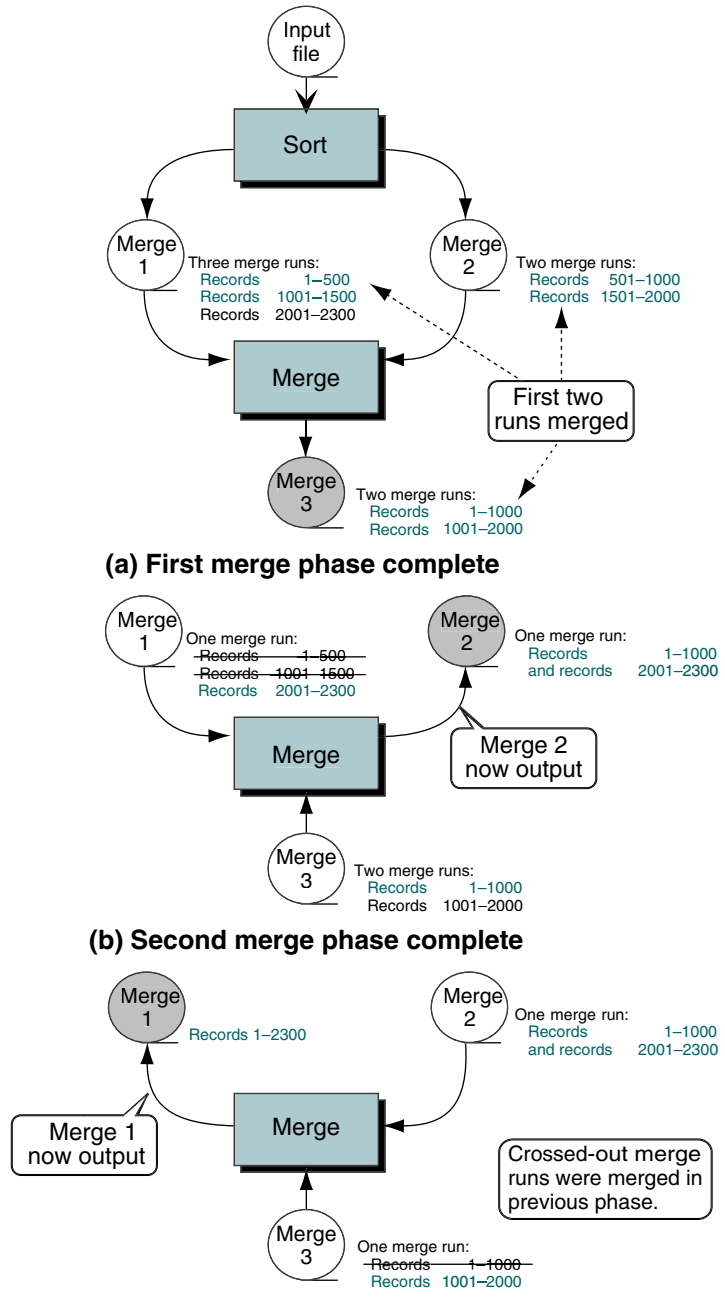


FIGURE 12-22 Polyphase Merge Sort

Figure 12-23 shows how we can use the heap sort to write long merge runs. Given a list of 12 elements to be sorted and using a heap of three nodes, we sort the data into two merge runs. We begin by filling the sort array and then turning it into a **minimum heap**. After creating the heap, we write the smallest element, located at the root, to the first merge file and read the fourth element (97) into the root node. We again reheap and this time write 21 to the merge file. After reading 87 and rebuilding the heap, we write 78 to the merge file.

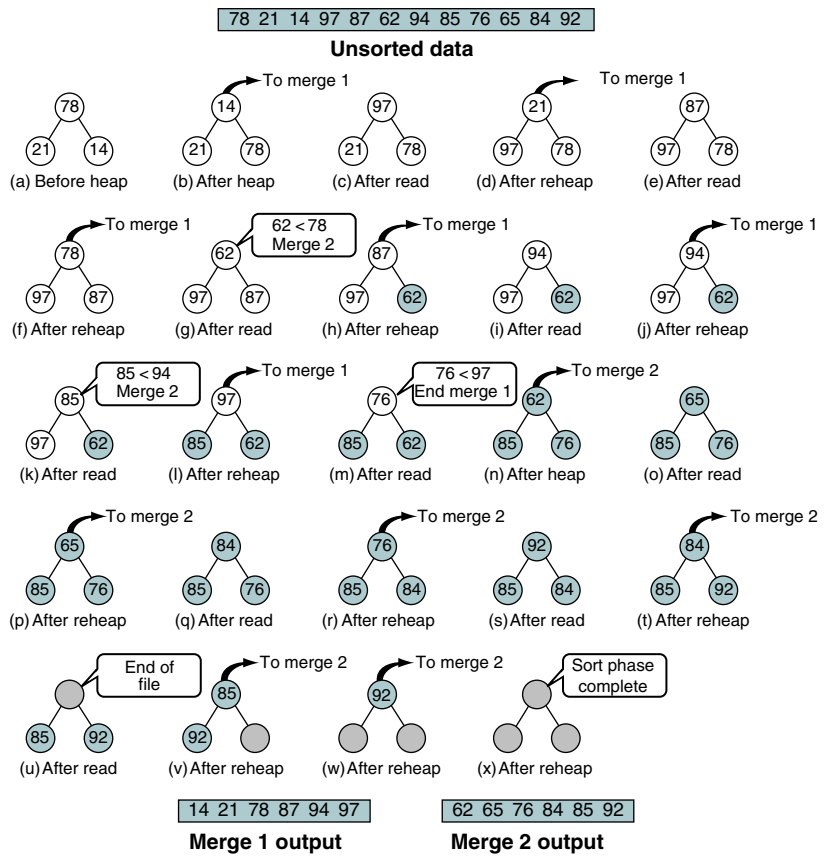


FIGURE 12-23 External Sort Phase Using Heap Sort

After reading 62 we have the heap shown in Figure 12-23(g). At this point the data we just read (62) is smaller than the last element we wrote to the merge file (78). Thus it cannot be placed in the current merge run in merge 1. By definition, it belongs to the next merge run. Therefore, we exchange it with the last element in the heap (87) and subtract 1 from the heap size, making the heap only two elements. We indicate that the third element, containing 62, is not currently in the heap by shading it. After we reheap we write 87 to merge 1. We then read 94, reheap, and write it to merge 1. When we read 85,

we discover that it is less than the largest key in merge 1 (94), so it must also be eliminated from the heap. After exchanging it with the last element in the heap, we have the situation shown in Figure 12-23(l).

After writing 97 to merge 1, we read 76. Because 76 is less than the last element written to the merge file, it also belongs to the next merge run. At this point all elements in the array belong to the second merge run. The first merge run is therefore complete, as shown in Figure 12-23(m).

After rebuilding the heap, we write the first element to merge file 2. We continue reading, reheaping, and writing until we write 84 to the merge file. At this point we have read all of the data to be sorted, and the input file is at the end of the file, as shown in Figure 12-23(u). We therefore move the last element to the heap, subtract 1 from the heap size, and reheap. After writing 85 to the merge run, we move the last element in the heap to the root, reheap, and write it to the merge file. Because the heap is now empty, the sort is complete.

If you examine the merge runs created in this example, you will note that they are both twice as long as the array size. As a rule of thumb, our heuristic studies have indicated that the average merge run size is twice the size of the array being used for the sort. These longer merge runs eliminate one merge pass, which is a significant time saver.

12.6 Quick Sort Efficiency

Quick sort is considered the best general-purpose sort known today. We would expect, therefore, that it has a low sort effort. In Section 12.4 we discussed the efficiency of quick sort using big-O notation and showed that it was indeed fast. In this section we provide a background for you to analyze recursive algorithms mathematically.

To calculate the complexity of quick sort, imagine that the number of comparisons to sort an array of n elements (index ranges from 0 to $n - 1$) is f(n). We observe the following:

- An array of zero or one element is already sorted. This means f(0) = f(1) = 0.
- If we choose the pivot to be at index i, we have two subarrays. The left subarray has (i) elements, and the right subarray has ($n - 1 - i$) elements. The number of comparisons to sort the left subarray is f(i), and the number of comparisons to sort the right subarray is f ($n - 1 - i$), where i can be between 0 to $n - 1$.
- To continuously divide the array into subarrays, we need n comparisons.

Now we can write the general formula for the number of comparisons:

$$\begin{aligned} &\mathbf{f(}n\mathbf{) = 0} && \textbf{if } n < 2 \\ &\mathbf{f(}n\mathbf{) = f(i) + f(}n\mathbf{ - i - 1) + }n && \textbf{if } n \geq 2 \end{aligned}$$

This formula can be used to find the complexity of the quick sort in three situations: worst case, best case, and average case.

Worst Case

There is no worst case for the modern quick sort. Singleton's modification in 1969 effectively eliminated it. In the original sort, however, the worst case occurred when the data were already sorted. In this case *i* is always 0, which means that the array is divided into the pivot cell and the rest of the array. When the list is sorted, either ascending or descending, the formula can be simplified to:

$$f(n) = f(i) + f(n - i - 1) + n$$
$$f(n) = f(n - 1) + n \qquad \text{Because } f(i) = f(0) \text{ and } i = 0$$

This is a recursion calculation that defines f(n) in terms of f(n – 1). If we evaluate all of the factors in the recursion, we observe:

$$\begin{aligned}
f(n) &= f(n-1) + n \\
f(n-1) &= f(n-2) + n - 1 \\
f(n-2) &= f(n-3) + n - 2 \\
f(n-3) &= f(n-4) + n - 3 \\
f(n-4) &= f(n-5) + n - 4 \\
&\dots \\
f(1) &= f(0) + 1 \\
f(0) &= 0
\end{aligned}$$

If we add all right terms with the left terms, f(n – 1) on the right side of the first line is canceled by the f(n – 1) on the left side of the second line; f(n – 2) on the right side of the second line is canceled by the f(n – 2) on the left side of the third line, and so on. What is left is

$$f(n) = n + (n-1) + (n-2) + (n-3) + \dots + 1 + 0$$

This is the arithmetic series,[10] which evaluates to $n(n + 1) / 2$. When we eliminate the parentheses by multiplying the terms, we have:

$$f(n) = n\frac{(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

But this is n^2, therefore

$$O(n) = n^2 \qquad \text{Worst Case}$$

Best Case

In the best case, the arrays are always divided into two subarrays of equal sizes. In other words, the pivot key belongs exactly in the middle of the

10. See Appendix E.

array. For this to happen, the number of elements in the original array must be a power of two minus one. In other words, the number of elements can be $n = 2^k - 1$ or, conversely, $k = \log_2 (n - 1)$. So, given an array that meets this requirement, $i = 2^{2k-1} - 1$ and

$$\mathbf{f(n) = f(i) + f(n - i - 1) + n}$$
$$\mathbf{f(2^k - 1) = f(2^{k-1} - 1) + f(2^k - 1 - (2^{k-1} - 1) - 1) + 2^k - 1}$$
$$\mathbf{f(2^k - 1) = 2f(2^{k-1} - 1) + 2^k - 1}$$

To solve this recursive definition, we can replace k with $k - 1$ repeatedly and multiply both sides of the equation by a factor of 2. This gives us:

$$\mathbf{f(2^k - 1) = 2^1 f(2^{k-1} - 1) + 2^0(2^k - 1)}$$
$$\mathbf{2^1 f(2^{k-1} - 1) = 2^2 f(2^{k-2} - 1) + 2^1(2^{k-1} - 1)}$$
$$\mathbf{2^2 f(2^{k-2} - 1) = 2^3 f(2^{k-3} - 1) + 2^2(2^{k-2} - 1)}$$
$$\mathbf{\ldots}$$
$$\mathbf{2^{k-2} f(2^2 - 1) = 2^{k-1} f(2^1 - 1) + 2^{k-1}(2^1 - 1)}$$

Now we add all of the right side together and all of the left side together; after cancelling equal terms from different sides, we get

$$\mathbf{f(2^k - 1) = 2^{k-1} f(1) + 2^0(2^k - 1) + 2^1(2^{k-1} - 1) + 2^2(2^{k-2} - 1) + \ldots + 2^{k-1}(2^1 - 1)}$$

But, $f(1) = 0$, so

$$\mathbf{f(2^k - 1) = (2^k - 1) + (2^k - 2^1) + (2^k - 2^2) + \ldots + (2^k - 2^{k-1})}$$
$$\mathbf{f(2^k - 1) = (k - 1)2^k - (2^0 + 2^1 + 2^2 + \ldots + 2^{k-1})}$$

The term $(2^0 + 2^1 + 2^2 + \ldots + 2^{k-1})$ is a geometric series[11] that reduces to

$$\mathbf{(2^k - 1) / (2 - 1) \text{ or } 2^k - 1}$$

So we have

$$\mathbf{f(2^k - 1) = (k - 1)2^k - (2^k - 1)}$$

If we replace $n = 2^k - 1$ or $k = \log (n + 1)$, we get

$$\mathbf{f(n) = (\log (n + 1) - 1)(n + 1) - (n)}$$
$$\mathbf{f(n) = (n + 1) \log (n + 1) - n - 1 - n}$$
$$\mathbf{= (n + 1) \log (n + 1) - 2n - 1}$$

11. See Appendix E.

But, this is $n \log n$, therefore

$$O(n) = n \log n \qquad \text{Best Case}$$

Average Case

In the average case, i can be anywhere. In each partition it is possible that: (1) i is at the beginning. With Singleton's variation this is highly improbable. It would require that two keys are equal and smaller than or equal to a third key. (2) i is somewhere in the middle area of the array. This is the most probable. The three values used to select the median value are randomly representative of the list. And, finally, (3) i is at the end of the array. Again, for the same reasons as cited in (1), this is highly improbable. We therefore assume that i is somewhere in the middle portion and take an average.

Assuming an average, we see that

$$f(n) = f(i) + f(n - i - 1) + n$$
$$f(n) = (1 / n)\{f(0) + f(1) + \ldots + f(n - 1)\} + (1 / n)\{f(n - 1) + f(n - 2) + \ldots + f(0)\} + n$$

We can observe that the first two terms are exactly the same (the functions inside the braces are listed in reverse). So we have

$$f(n) = (2 / n)\{f(0) + f(1) + \ldots + f(n - 1)\} + n$$

To solve this recursion definition, we multiply both sides of equation by n

$$nf(n) = 2\{f(0) + f(1) + \ldots + f(n-1)\} + n^2$$

Now we replace n with $n - 1$, which gives us the following equation:

$$(n - 1)f(n - 1) = 2\{f(0) + f(1) + \ldots + f(n - 2)\} + (n - 1)^2$$

Now we subtract the new equation from the previous one, giving us

$$nf(n) - (n - 1)f(n - 1) = 2f(n - 1) + n^2 - (n - 1)^2$$

Simplifying, we get

$$nf(n) = (n + 1)f(n - 1) + 2n - 1$$

We divide both sides by $n\ (n + 1)$, and we get

$$\frac{1}{n+1} f(n) = \frac{1}{n} f(n - 1) + \frac{2}{n+1} - \frac{1}{n(n+1)}$$

Now we substitute n with $n - 1$, $n - 2$, …, 2 continuously

$$\frac{1}{n+1} f(n) = \frac{1}{n} f(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)}$$

$$\frac{1}{n} f(n-1) = \frac{1}{n-1} f(n-2) + \frac{2}{n} - \frac{1}{(n-1)(n)}$$

$$\frac{1}{n-1} f(n-2) = \frac{1}{n-2} f(n-3) + \frac{2}{n-1} - \frac{1}{(n-2)(n-1)}$$

$$\vdots$$

$$\frac{1}{3} f(2) = \frac{1}{2} f(1) + \frac{2}{3} - \frac{1}{(2)(3)}$$

Now we add both sides and cancel similar terms

$$(1 / (n + 1))f(n) =$$
$$(1 / 2)f(1) +$$
$$(2 / (n + 1) + 2 / n - 1) + \ldots + 2 / 3) -$$
$$\{(1 / n(n + 1) + 1 / (n - 2)(n) + 1 / (n - 2)(n - 1) + \ldots + 1 / 6)\}$$

But, f(1) = 0, so

$$1 / (n + 1)f(n) =$$
$$(2 / (n + 1) + 2 / n - 1) + \ldots + 2 / 3) -$$
$$\{(1 / n(n + 1) + 1 / (n - 2)(n) + 1 / (n - 2)(n - 1) + \ldots + 1 / 6)\}$$
$$f(n) =$$
$$(n + 1)(2 / (n + 1) + 2 / n - 1) + \ldots + 2 / 3) -$$
$$\{(n + 1)(1 / n(n + 1) + 1 / (n - 2)(n) + 1 / (n - 2)(n - 1) + \ldots + 1 / 6)\}$$

The expression $(2 / (n + 1) + 2 / n - 1) + \ldots + 2 / 3)$ is a natural logarithm; the second expression is negligible because it is of order $1 / n$ and can be ignored by the rules of big-O notation. Therefore, the result is

$$f(n) = (n + 1) \ln n$$

or in big-O notation

$$O(n) = (n \log n)$$

12.7 Key Terms

balanced merge
bubble sort
distribution phase
exchange sort
external sort
heap sort
internal sort
merge
merge phase
merge run
merge sort
minimum heap
natural merge
pivot
polyphase merge
quick sort
rollout
selection sort
sentinel
shell sort
sort
sort efficiency
sort order
sort pass
sort phase
sort stability
stepdown
straight insertion sort
straight selection sort

12.8 Summary

- One of the most common applications in computer science is sorting.
- Sorts are generally classified as either internal or external.
  - In an internal sort, all of the data are held in primary storage during the sorting process.
  - An external sort uses primary storage for the data currently being sorted and secondary storage for any data that does not fit in primary memory.
- Data may be sorted in either ascending or descending order.
- Sort stability is an attribute of a sort indicating that data with equal keys maintain their relative input order in the output.
- Sort efficiency is a measure of the relative efficiency of a sort.
- Each traversal of the data during the sorting process is referred to as a pass.
- Internal sorting can be divided into three broad categories: insertion, selection, and exchange.
- Two methods of insertion sorting were discussed in this chapter: straight insertion sort and shell sort.
  - In the straight insertion sort, the list at any moment is divided into two sublists: sorted and unsorted. In each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.

- The shell sort algorithm is an improved version of the straight insertion sort, in which the process uses different increments to sort the list. In each increment the list is divided into segments, which are sorted independent of each other.

- Two methods of selection sorting are discussed in this chapter: straight selection sort and heap sort.
  - In the straight selection sort, the list at any moment is divided into two sublists: sorted and unsorted. In each pass the process selects the smallest element from the unsorted sublist and exchanges it with the element at the beginning of the unsorted sublist.
  - The heap sort is an improved version of the straight selection sort. In the heap sort, the largest element in the heap is exchanged with the last element in the unsorted sublist. However, selecting the largest element is much easier in this sort method because the largest element is the root of the heap.
- Two methods of exchange sorting are discussed in this chapter: bubble sort and quick sort.
  - In the bubble sort, the list at any moment is divided into two sublists: sorted and unsorted. In each pass the smallest element is bubbled up from the unsorted sublist and moved to the sorted sublist.
  - The quick sort is the new version of the exchange sort in which the list is continuously divided into smaller sublists and exchanging takes place between elements that are out of order. Each pass of the quick sort selects a pivot and divides the list into three groups: a partition of elements whose key is less than the pivot's key, the pivot element that is placed in its ultimate correct position, and a partition of elements greater than or equal to the pivot's key. The sorting then continues by quick sorting the left partition followed by quick sorting the right partition.
- The efficiency of straight insertion, straight selection, and bubble sort is $O(n^2)$.
- The efficiency of shell sort is $O(n^{1.25})$, and the efficiency of heap and quick sorts is $O(n \log n)$.
- External sorting allows a portion of the data to be stored in secondary storage during the sorting process.
- External sorting consists of two phases: the sort phase and the merge phase.
- The merge phase uses one of three methods: natural merge, balanced merge, and polyphase merge.
  - In natural merge each phase merges a constant number of input files into one output file. The natural merge requires a distribution process between each merge phase.

- In the balanced merge, the distribution processes are eliminated by using the same number of input and output files.
- In polyphase merge, a number of input files are merged into one output file. However, the input file, which is exhausted first, is immediately used as an output file. The output file in the previous phase becomes one of the input files in the next phase.

- To improve the efficiency of the sort phase in external sorting, we can use a tree sort, such as the minimum heap sort. This sort allows us to write to a merge file before the sorting process is complete, which results in longer merge runs.

12.9 Practice Sets

Exercises

1. An array contains the elements shown below. The first two elements have been sorted using a straight insertion sort. What would be the value of the elements in the array after three more passes of the straight insertion sort algorithm?

```
3 13 7 26 44 23 98 57
```

2. An array contains the elements shown below. Show the contents of the array after it has gone through a one-increment pass of the shell sort. The increment factor is $k = 3$.

```
23 3 7 13 89 7 66 2 6 44 18 90 98 57
```

3. An array contains the elements shown below. The first two elements have been sorted using a straight selection sort. What would be the value of the elements in the array after three more passes of the selection sort algorithm?

```
7 8 26 44 13 23 98 57
```

4. An array contains the elements shown below. What would be the value of the elements in the array after three passes of the heap sort algorithm?

```
44 78 22 7 98 56 34 2 38 35 45
```

5. An array contains the elements shown below. The first two elements have been sorted using a bubble sort. What would be the value of the elements in the array after three more passes of the bubble sort algorithm? Use the version of bubble sort that starts from the end and bubbles up the smallest element.

```
7 8 26 44 13 23 57 98
```

6. An array contains the elements shown below. Using a quick sort, show the contents of the array after the first pivot has been placed in its correct location. Identify the three sublists that exist at that point.

```
44 78 22 7 98 56 34 2 38 35 45
```

7. After two passes of a sorting algorithm, the following array:

```
47 3 21 32 56 92
```

has been rearranged as shown below.

```
3 21 47 32 56 92
```

Which sorting algorithm is being used (straight selection, bubble, or straight insertion)? Defend your answer.

8. After two passes of a sorting algorithm, the following array:

```
80 72 66 44 21 33
```

has been rearranged as shown below.

```
21 33 80 72 66 44
```

Which sorting algorithm is being used (straight selection, bubble, or straight insertion)? Defend your answer.

9. After two passes of a sorting algorithm, the following array:

```
47 3 66 32 56 92
```

has been rearranged as shown below.

```
3 47 66 32 56 92
```

Which sorting algorithm is being used (straight selection, bubble, or straight insertion)? Defend your answer.

10. Show the result after each merge phase when merging the following two files:

```
6 12 19 23 34 · 8 11 17 20 25 · 9 10 15 25 35
```

```
13 21 27 28 29 · 7 30 36 37 39
```

11. Starting with the following file, show the contents of all of the files created using external sorting and the natural merge method (do not include a sort phase):

```
37 9 23 56 4 5 12 45 78 22 33 44 14 17 57 11 35 46 59
```

12. Rework Exercise 11 using the balanced merge method.
13. Rework Exercise 11 using the polyphase merge method with an array size of seven and an insertion sort.

Problems

14. Modify Program 12-3, "Insertion Sort," to count the number of data moves needed to order an array of 1000 random numbers. A data move is a movement of an element of data from one position in the array to another, to a hold area, or from a hold area back to the array. Display the array before and after the sort. At the end of the program, display the total moves needed to sort the array.
15. Repeat Problem 14 using the shell sort (see Program 12-4).
16. Repeat Problem 14 using the selection sort (see Program 12-1).
17. Repeat Problem 14 using the heap sort (see Program 12-2).
18. Repeat Problem 14 using the bubble sort (see Program 12-5).
19. Repeat Problem 14 using the quick sort (see Program 12-8).
20. Change the bubble sort algorithm (Program 12-5) as follows: Use two-directional bubbling in each pass. In the first bubbling, the smallest element is bubbled up; in the second bubbling, the largest element is bubbled down. This sort is known as the shaker sort.
21. Using the techniques discussed in "Sorts and ADTs" at the end of Section 12.1, create an ADT using the selection sort (Program 12-1). Then write a test driver to test it once with an array of integers and once with an array of floating-point numbers.
22. Write an algorithm that applies the incremental idea of the shell sort to a selection sort. The algorithm first applies the straight section sort to items $n / 2$ elements apart (first, middle, and last). It then applies it to $n / 3$ elements apart, then to elements $n / 4$ apart, and so on.

23. Write a recursive version of the selection sort algorithm (see Program 12-4).
24. Rewrite the insertion sort algorithm (Program 12-3) using a singly linked list instead of an array.

Projects

25. Merge sorting is an example of a divide-and-conquer paradigm. In our discussions, we used merge only as an external sorting method. It can also be used for internal sorting. Let's see how it works. If we have a list of only two elements, we can simply divide the list into two halves and then merge them. In other words, the merge sort is totally dependent on two processes, distribution and merge. This elementary process is shown in Figure 12-24.

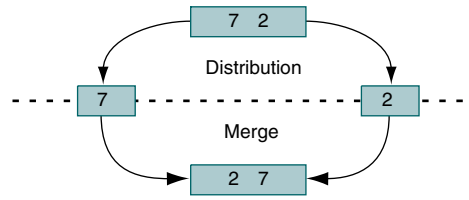


FIGURE 12-24 Split and Merge for Project 25

Given a list longer than two elements, we can sort by repeating the distribution and merge processes. Because we don't know how many elements are in the input list when we begin, we can distribute the list originally by writing the first element to one output list, the second element to a second output list, and then continue writing alternatively to the first list and then the second list until the input list has been divided into two output lists. The output lists can then be sorted using a balanced two-way merge. This process is shown in Figure 12-25. It could be called the "sortless sort" because it sorts without ever using a sort phase.

Write a C program to sort an array of 500 random numbers, using this approach. Print the data before and after the sort.

26. Write a program that sorts an array of random numbers, using the shell sort and the quick sort. Both sorts should use the same data. Each sort should be executed twice. For the first sort, fill the array with random numbers between 1 and 999. For the second sort, fill the array with a nearly ordered list. Construct your nearly ordered list by reversing elements 19 and 20 in the sorted random-number list. For each sort, count the number of comparisons and moves necessary to order this list.

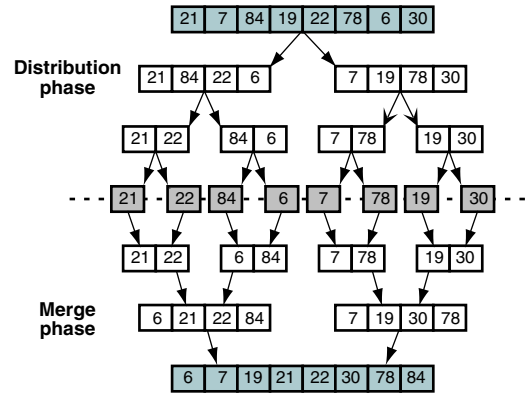


FIGURE 12-25 Sortless Sort

Run the program three times, once with an array of 100 items, once with an array of 500 items, and once with an array of 1000 items. For the first execution only (100 elements), print the unsorted data followed by the sort data in 10-by-10 matrixes (10 rows of 10 numbers each). For all runs print the number of comparisons and the number of moves required to order the data.

To make sure your statistics are as accurate as possible, you must analyze each loop limit condition test and each selection statement in your sort algorithms. The following notes should help with this analysis:

a. All loops require a count increment in their body.
b. Pretest loops (*while* and *for*) also require a count increment either before (recommended) or after the loop to count the last test.
c. Remember that C uses the shortcut rule for evaluating Boolean and/or expressions. The best way to count them is with a comma expression, as shown below. Use similar code for the selection statements.

```
while ((count++, a) && (count++, b))
```

Analyze the statistics you generated and write a short report (less than one page) concerning what you discovered about these two sorts. Include the data in Table 12-4, one for the random data and one for the nearly ordered data. Calculate the ratio to one decimal place.

27. Repeat Project 26 adding heap sort and using your computer's internal clock. For each sort algorithm, start the clock as the last statement before calling the sort and read the clock as the first statement after the sort. Write a short report comparing the run times with the suggested algorithmics in the text. If your results do not agree with the algorithmics, increase the array size in element increments of 1000 to get a better picture.

List Size	Shell	Quick	Ratio (shell/quick)
Compares			
100			
500			
1000			
Moves			
100			
500			
1000			

TABLE 12-4 Sorting Statistics Format for Project 26

28. Radix sorting—also known as digit, pocket, and bucket sorting—is a very efficient sort for large lists whose keys are relatively short. If fact, if we consider only its big-O notation, which is $O(n)$, it is one of the best. Radix sorts were used extensively in the punched-card era to sort cards on electronic accounting machines (EAMs).

    In a radix sort, each pass through the list orders the data one digit at a time. The first pass orders the data on the units (least significant) digit. The second pass orders the data on the tens digit. The third pass orders the data on the hundreds digit, and so forth until the list is completely sorted by sorting the data on the most significant digit.

    In EAM sorts the punched cards were sorted into pockets. In today's systems we would sort them into a linked list, with a separate linked list for each digit. Using a pocket or bucket approach, we sort eight four-digit numbers in Figure 12-26.[12]

    If you analyze this sort, you will see that there are k sort passes, where k is the number of digits in the key. For each sort pass, we have n operations, where n is the number of elements to be sorted. This gives us an efficiency of kn, which in big-O notation is $O(n)$.

    Write a program that uses the radix sort to sort 1000 random digits. Print the data before and after the sort. Each sort bucket should be a linked list. At the end of the sort, the data should be in the original array.

29. Modify Project 27 to include the radix sort. Build a table of sort times and write a short paper explaining the different sort timings and their relationship to their sort efficiency.

12. *Note:* For efficiency, radix sorts are not recommended for small lists or for keys containing a large number of digits.

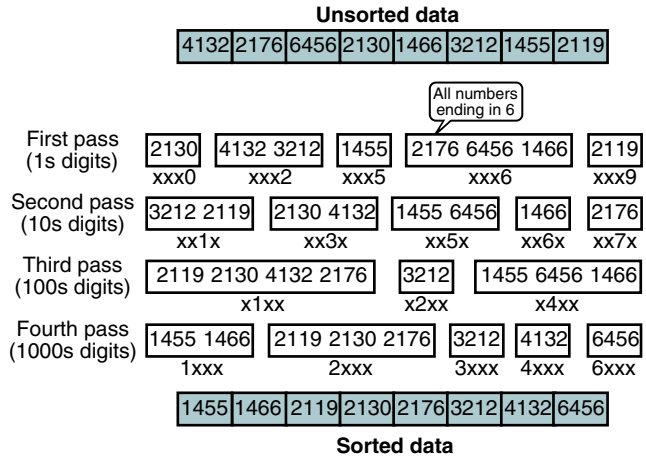


FIGURE 12-26 Radix Sorting Example for Project 27

30. Using the techniques discussed in "Sorts and ADTs" at the end of Section 12.1, create an ADT using the straight insertion sort (Program 12-3). Then write a test driver to test it once with an array of integers and once with an array of floating-point numbers.

31. Using the techniques discussed in "Sorts and ADTs" at the end of Section 12.1, create an ADT using the bubble sort (Program 12-5). Then write a test driver to test it once with an array of integers and once with an array of floating-point numbers.

32. Using the techniques discussed in "Sorts and ADTs" at the end of Section 12.1, create an ADT using the shell sort (Program 12-4). Then write a test driver to test it once with an array of integers and once with an array of floating-point numbers.

33. Using the techniques discussed in "Sorts and ADTs" at the end of Section 12.1, create an ADT using the heap sort (Program 12-2). Then write a test driver to test it once with an array of integers and once with an array of floating-point numbers.

34. Using the techniques discussed in "Sorts and ADTs" at the end of Section 12.1, create an ADT using the quick sort (Program 12-6). Then write a test driver to test it once with an array of integers and once with an array of floating-point numbers.

Chapter 13
Searching

One of the most common and time-consuming operations in computer science is **searching,** the process used to find the location of a target among a list of objects. In this chapter we study several search algorithms. We begin with list searching and a discussion of the two basic search algorithms: the sequential search—including three interesting variations—and the binary search.

After reviewing the concepts of list searches, we discuss hashed list searching, in which the data key is algorithmically manipulated to calculate the location of the data in the list. An integral part of virtually all hashed list algorithms is collision resolution, which we discuss in the last section.

Although we discuss the list search algorithms using an array structure, the same concepts can be found in linked list searches. The sequential search, along with the ordered list variation, is most commonly used to locate data in a list (Chapter 5). The binary search tree (Chapter 7) is actually a structure built to provide the efficiency of the binary search of a tree structure. These searches are covered in their respective chapters.

13.1 List Searches

The algorithm used to search a list depends to a large extent on the structure of the list. In this section we study searches that work with arrays. The two basic searches for arrays are the sequential search and the binary search. The sequential search can be used to locate an item in any array. The binary search, on the other hand, requires an ordered list. The basic search concept is shown in Figure 13-1.

Sequential Search

The **sequential search** is used whenever the list is not ordered. Generally, you use this technique only for small lists or lists that are not searched often. In other cases you should first sort the list and then search it using the binary search, discussed later.

FIGURE 13-1 Search Concept

In the sequential search, we start searching for the target at the beginning of the list and continue until we find the target or we are sure that it is not in the list. This approach gives us two possibilities: either we find it or we reach the end of the list. In Figure 13-2 we trace the steps to find the value 14. We first check the data at index 0, then 1, and then 2 before finding 14 in the fourth element (index 3).

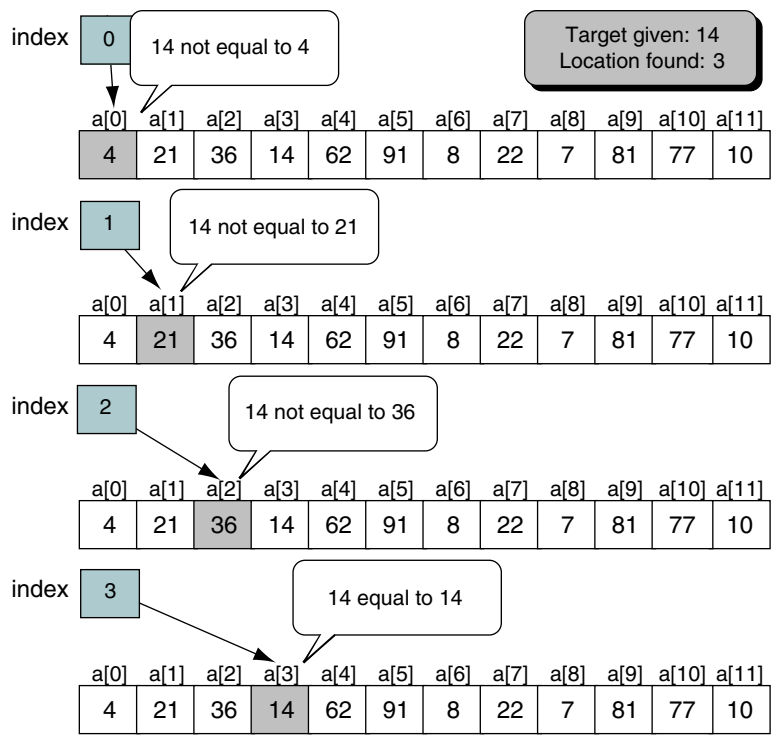
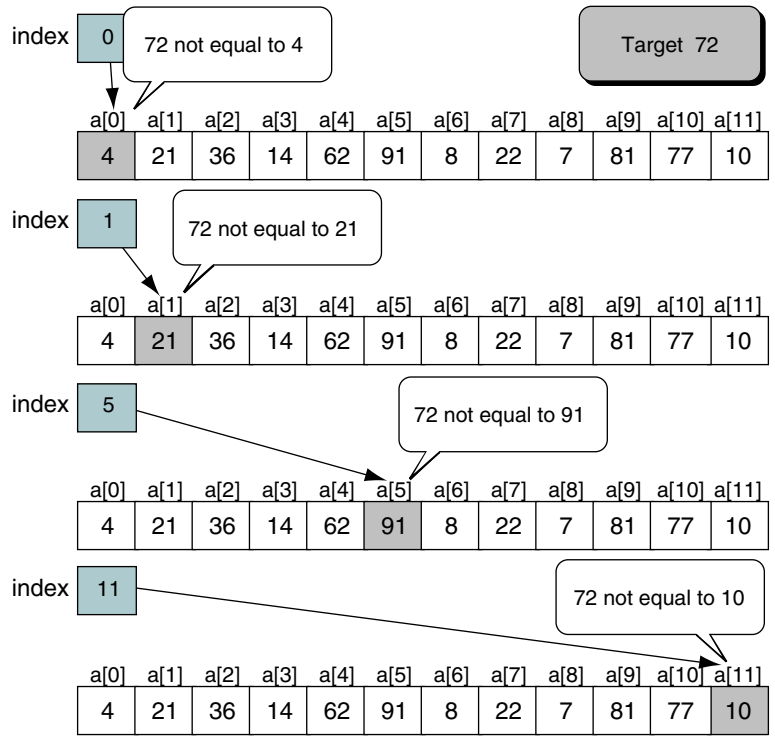


FIGURE 13-2 Successful Search of an Unordered List

But what if the target were not in the list? In that case we would have to examine each element until we reach the end of the list. Figure 13-3 traces the search for a target of 72. At the end of the list, we discover that the target does not exist.



Note: Not all test points are shown.

FIGURE 13-3 Unsuccessful Search in Unordered List

Sequential Search Algorithm

The sequential search algorithm needs to tell the calling algorithm two things: First, did it find the data it was looking for? And, second, if it did, at what index are the target data found? To answer these questions, the search algorithm requires four parameters: (1) the list we are searching, (2) an index to the last element in the list,[1] (3) the target, and (4) the address where the found element's index location is to be stored. To tell the calling algorithm whether the data were found, we return a Boolean—true if we found it or false if we didn't find it.

Although we could write a sequential search algorithm without passing the index to the last element, if we did so the search would have to know how

1. As an alternative to the index to the last element, the size of the list may be passed.

many elements are in the list. To make the function as flexible as possible, we pass the index of the last data value in the array. Generalizing the algorithm by passing the index to the last item is also a good structured design technique. With this information we are now ready to code Algorithm 13-1.

ALGORITHM 13-1 Sequential Search

```
Algorithm seqSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
   Pre    list must contain at least one element
          last is index to last element in the list
          target contains the data to be located
          locn is address of index in calling algorithm
   Post   if found: index stored in locn & found true
          if not found: last stored in locn & found false
   Return found true or false
 1 set looker to 0
 2 loop (looker < last AND target not equal list[looker])
   1   increment looker
 3 end loop
 4 set locn to looker
 5 if (target equal list[looker])
   1   set found to true
 6 else
   1   set found to false
 7 end if
 8 return found
end seqSearch
```

Algorithm 13-1 Analysis We have only one comment. We could have used the location parameter for the search rather than create the `looker` index. However, it is generally not a good idea to use a parameter as a working variable because doing so destroys any initial value that may be needed later in the algorithm.

Variations on Sequential Searches

Three useful variations on the sequential search algorithm are: (1) the sentinel search, (2) the probability search, and (3) the ordered list search. We look at each briefly in the following sections.

Sentinel Search

If you examine the search algorithm carefully, you note that the loop tests two conditions: the end of the list and the target's not being found. Knuth states, "When the inner loop of a program tests two or more conditions, we should try to reduce the testing to just one condition."[2] If we know that the target will be found in the list, we can eliminate the test for the end of the list. The

2. Donald E. Knuth, *The Art of Computer Programming,* vol. 3, *Sorting and Searching,* Second Edition (Reading, MA: Addison-Wesley, 1998), 398.

only way we can ensure that a target is actually in the list is to put it there ourself. A target is put in the list by adding an extra element (sentinel entry) at the end of the array and placing the target in the sentinel. We can then optimize the loop and determine after the loop completes whether we found actual data or the sentinel.[3] The obvious disadvantage is that the rest of the processing must be careful to never look at the sentinel element at the end of the list. The pseudocode for the sentinel search is shown in Algorithm 13-2.

ALGORITHM 13-2 Sentinel Search

```
Algorithm SentinelSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
   Pre    list must contain element at the end for sentinel
          last is index to last data element in the list
          target contains the data to be located
          locn is address of index in calling algorithm
   Post   if found--matching index stored in locn & found
             set true
          if not found--last stored in locn & found false
   Return found true or false
 1 set list[last + 1] to target
 2 set looker to 0
 3 loop (target not equal list[looker])
   1   increment looker
 4 end loop
 5 if (looker <= last)
   1   set found to true
   2   set locn  to looker
 6 else
   1   set found to false
   2   set locn  to last
 7 end if
 8 return found
end SentinelSearch
```

Probability Search

One of the more useful variations of the sequential search is known as the **probability search**. In the probability search, the data in the array are arranged with the most probable search elements at the beginning of the array and the least probable at the end. It is especially useful when relatively few elements are the targets for most of the searches. To ensure that the probability ordering is correct over time, in each search we exchange the located element with the element immediately before it in the array. A typical implementation of the probability search is shown in Algorithm 13-3.

3. It is not always possible to reduce a loop to only one test. We will see many loops that require two or more tests to satisfy the logic.

ALGORITHM 13-3 Probability Search

```
Algorithm ProbabilitySearch (list, last, target, locn)
Locate the target in a list ordered by the probability of each
element being the target--most probable first, least probable
last.
   Pre    list must contain at least one element
          last is index to last element in the list
          target contains the data to be located
          locn is address of index in calling algorithm
   Post   if found--matching index stored in locn,
             found true, and element moved up in priority.
          if not found--last stored in locn & found false
   Return found true or false
 1 find target in list
 2 if (target in list)
   1   set found to true
   2   set locn to index of element containing target
   3   if (target after first element)
       1  move element containing target up one location
   4   end if
 3 else
   1   set found to false
 4 end if
 5 return found
end ProbabilitySearch
```

Ordered List Search

Although we generally recommend a binary search when searching a list ordered on the key (target), if the list is small it may be more efficient to use a sequential search. When searching an ordered list sequentially, however, it is not necessary to search to the end of the list to determine that the target is not in the list. We can stop when the target becomes less than or equal to the current element we are testing. In addition, we can incorporate the sentinel concept by bypassing the search loop when the target is greater than the last item. In other words, when the target is less than or equal to the last element, the last element becomes a sentinel, allowing us to eliminate the test for the end of the list.

Although it can be used with array implementations, the ordered list search is more commonly used when searching linked list implementations. The pseudocode for searching an ordered array is found in Algorithm 13-4.

ALGORITHM 13-4 Ordered List Search

```
Algorithm OrderedListSearch (list, last, target, locn)
Locate target in a list ordered on target.
Pre    list must contain at least one element
          last is index to last element in the list
          target contains the data to be located
```

continued

ALGORITHM 13-4 Ordered List Search *(continued)*

```
            locn is address of index in calling algorithm
   Post     if found--matching index stored in locn-found true
            if not found--locn is index of first element >
               target or locn equal last & found is false
   Return found true or false
 1 if (target less than last element in list)
   1   find first element less than or equal to target
   2   set locn to index of element
 2 else
   1   set locn to last
 3 end if
 4 if (target in list)
   1   set found to true
 5 else
   1   set found to false
 6 end if
 7 return found
end OrderedListSearch
```

Binary Search

The sequential search algorithm is very slow. If we have an array of 1000 elements, we must make 1000 comparisons in the worst case. If the array is not sorted, the sequential search is the only solution. However, if the array is sorted, we can use a more efficient algorithm called the **binary search**. Generally speaking, we should use a binary search whenever the list starts to become large. Although the definition of *large* is vague, we suggest that you consider binary searches whenever the list contains more than 16 elements.

The binary search starts by testing the data in the element at the middle of the array to determine if the target is in the first or the second half of the list. If it is in the first half, we do not need to check the second half. If it is in the second half, we do not need to test the first half. In other words, we eliminate half the list from further consideration with just one comparison. We repeat this process, eliminating half of the remaining list with each test, until we find the target or determine that it is not in the list.

To find the middle of the list, we need three variables: one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list. We analyze two cases here: the target is in the list and the target is not in the list.

Target Found

Figure 13-4 traces the binary search for a target of 22 in a sorted array.

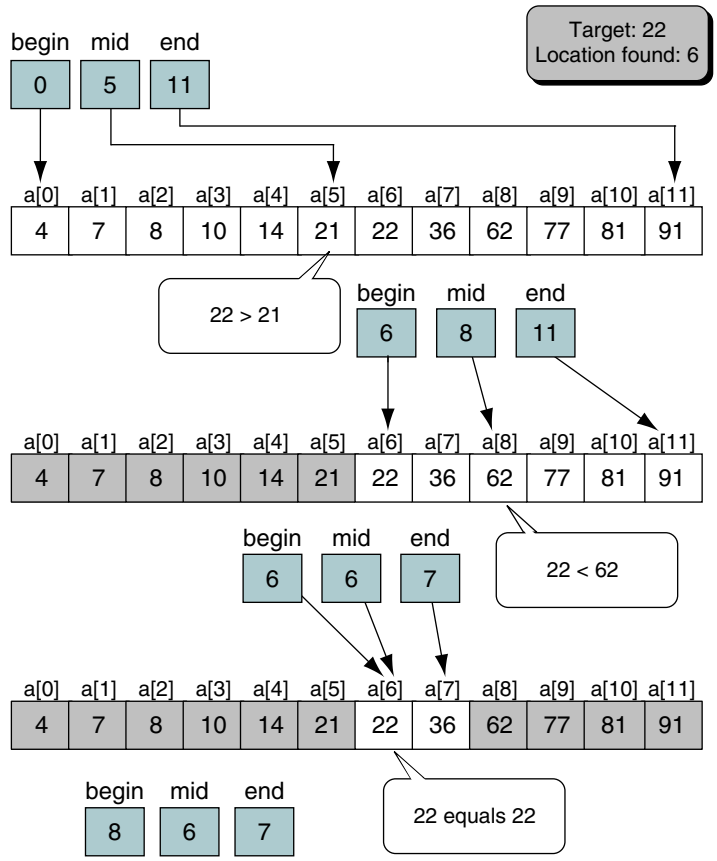


FIGURE 13-4 Successful Binary Search Example

We descriptively call our three indexes `begin`, `mid`, and `end`. Given `begin` as 0 and `end` as 11, we can calculate `mid` as follows:

$$\texttt{mid} = \lfloor (\texttt{begin} + \texttt{end}) \ / \ 2 \rfloor = \lfloor (0 + 11) \ / \ 2 \rfloor = 5$$

At index location 5, we discover that the target is greater than the list value (22 > 21). We can therefore eliminate the array locations 0 through 5. (Note that `mid` is automatically eliminated.) To narrow our search, we set `mid + 1` to `begin` and repeat the search.

The next loop calculates `mid` with the new value for `begin` (6) and determines that the midpoint is now 8.

```
mid = ⌊(6 + 11) / 2⌋ = ⌊17 / 2⌋ = 8
```

When we test the target to the value at `mid` a second time, we discover that the target is less than the list value (22 < 62). This time we adjust the end of the list by setting `end` to `mid - 1` and recalculating `mid`.

```
mid = ⌊(6 + 7) / 2⌋ = ⌊13 / 2⌋ = 6
```

This step effectively eliminates elements 8 through 11 from consideration. We have now arrived at index location 6, whose value matches our target. To stop the search in the algorithm (see page 607), we force `begin` to be greater than `end`.

Target Not Found

A more interesting case occurs when the target is not in the list. We must construct our search algorithm so that it stops when we have checked all possible locations. We do this in the binary search by testing for the `begin` and `end` indexes crossing; that is, we are done when `begin` becomes greater than `end`. We now see that two conditions terminate the binary search algorithm: the target is found or it is not found. To terminate the loop when it is found, we force `begin` to be greater than `end`. When the target is not in the list, `begin` becomes larger than `end` automatically.

Let's demonstrate the target's not being found with an example. In Figure 13-5 we search for a target of 11, which doesn't exist in the array.

In this example the loop continues to narrow the range as we saw in the successful search until we are examining the data at index locations 3 and 4. These settings of `begin` and `end` set the `mid` index to 3.

```
mid = ⌊(3 + 4) / 2⌋ = ⌊7 / 2⌋ = 3
```

The test at index location 3 indicates that the target is greater than the list value, so we set `begin` to `mid + 1`, or 4. We now test the data at location 4 and discover that 11 < 14.

```
mid = ⌊(4 + 4) / 2⌋ = ⌊8 / 2⌋ = 4
```

At this point we have discovered that the target should be between two adjacent values; in other words, it is not in the list. We see this algorithmically because `end` is set to `mid - 1`, which makes `begin` greater than `end`, the signal that the value we are looking for is not in the list.

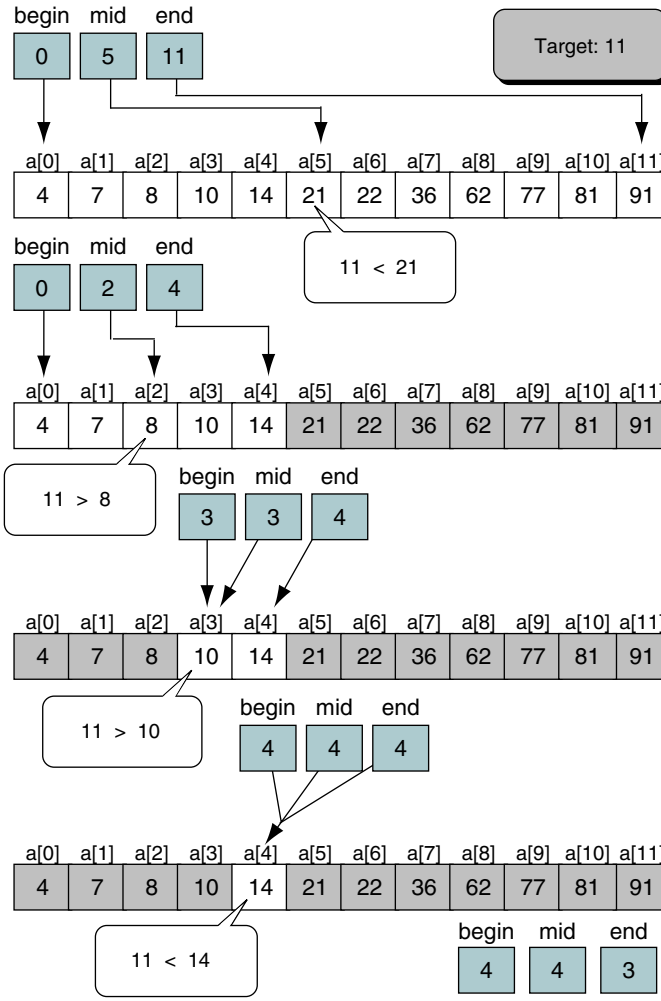


FIGURE 13-5 Unsuccessful Binary Search Example

Binary Search Algorithm

Algorithm 13-5 contains the implementation of the binary search algorithm we have been describing. It is constructed along the same design we saw for the sequential search. The first three parameters describe the list and the target we are looking for, and the last one contains the address into which we place the located index. One point is worth noting: when we terminate the loop with a not-found condition, the index returned is unpredictable—it may indicate that the node is greater than or less than the value in the target.

ALGORITHM 13-5 Binary Search

```
Algorithm binarySearch (list, last, target, locn)
Search an ordered list using Binary Search
   Pre    list is ordered; it must have at least 1 value
          last is index to the largest element in the list
          target is the value of element being sought
          locn is address of index in calling algorithm
   Post   FOUND: locn assigned index to target element
                 found set true
          NOT FOUND: locn = element below or above target
                     found set false
   Return found true or false
1  set begin to 0
2  set end to last
3  loop (begin <= end)
   1   set mid to (begin + end) / 2
   2   if (target > list[mid])
          Look in upper half
       1   set begin to (mid + 1)
   3   else if (target < list[mid])
          Look in lower half
       1   set end to mid - 1
   4   else
          Found: force exit
       1   set begin to (end + 1)
   5   end if
4  end loop
5  set locn to mid
6  if (target equal list [mid])
   1   set found to true
7  else
   1   set found to false
8  end if
9  return found
end binarySearch
```

Analyzing Search Algorithms

Of the search algorithms discussed, which is the best? An application often determines which algorithm should be used, but we analyze the algorithms to determine which is most efficient.

Sequential Search

The basic loop for the sequential search is shown below.

```
2 loop (looker < last AND target not equal list[looker])
      1 increment looker
```

This is a classic example of a linear algorithm. In fact, in some of the literature this search is known as a **linear search**. Because the algorithm is linear, its efficiency is $O(n)$.

> The efficiency of the sequential search is $O(n)$.

The search efficiency for the sentinel search is basically the same as for the sequential search. Although the sentinel search saves a few instructions in the loop, its design is identical. Therefore, it is also an $O(n)$ search. Likewise, the efficiency of the ordered list search is also $O(n)$. If we know the probability of a search's being successful, we can construct a more accurate formula for searching an ordered list. This improved accuracy, however, turns out to be a coefficient in the formula, which, as you recall, is dropped when using big-O analysis.

It is not possible to generalize the efficiency of the probability search without knowing the probability of each element in the list. On the other hand, if the probability of the first few elements of a list totals more than 90%, it can be a very efficient search, even considering the additional overhead required to maintain the list. In general, however, we recommend the binary search, especially for large lists.

Binary Search

The binary search locates an item by repeatedly dividing the list in half. Its loop is:

```
3  loop (begin <= end)
   1  set mid to (begin + end) / 22  if (target > list[mid])
      1  set begin to (mid + 1)
   3  else if (target < list[mid])
      1  set end to mid - 1
   4  else
      1  set begin to (end + 1)
   5  end if
4  end loop
```

This loop obviously divides, and it is therefore a logarithmic loop. The efficiency is thus $O(\log n)$, which you should recognize as one of the most efficient of all the measures.

> The efficiency of the binary search is $O(\log n)$.

Comparing the two searches, we see that, disregarding the time required to order the list, the binary search is obviously more efficient for searching a list of any significant size (see Table 13-1). For this reason the binary search is recommended for all but the smallest of lists (i.e., lists with fewer than 16 elements).

List size	Iterations	
	Binary	Sequential
16	4	16
50	6	50
256	8	256
1000	10	1000
10,000	14	10,000
100,000	17	100,000
1,000,000	20	1,000,000

TABLE 13-1 Comparison of Binary and Sequential Searches

13.2 Search Implementations

In this section we develop the C code for the sequential and binary searches. As we saw with sorts, we can write a generic search algorithm for any type of data. To allow us to concentrate on the search algorithms, we develop them using only integer data.

Sequential Search in C

The sequential search in Program 13-1 parallels the pseudocode implementation in Algorithm 13-1.

PROGRAM 13-1 Sequential Search

```
 1  /* Locate target in an unordered list of size elements.
 2        Pre  list must contain at least one item
 3             last is index to last element in list
 4             target contains the data to be located
 5             locn is address of index in calling function
 6        Post FOUND: matching index stored in locn address
 7                    return true (found)
 8             NOT FOUND: last stored in locn address
 9                        return false (not found)
10  */
11  bool seqSearch (int list[ ], int  last,
12                  int target,  int* locn)
13  {
14  // Local Definitions
15     int looker;
```

continued

PROGRAM 13-1 Sequential Search *(continued)*

```
16
17  // Statements
18     looker = 0;
19     while (looker < last && target != list[looker])
20        looker++;
21
22     *locn = looker;
23     return ( target == list[looker] );
24  }  // seqSearch
```

Program 13-1 Analysis Although this program is simple, it merits some discussion. First, why did we use a *while* statement rather than a *for* loop? Even though we know the limits of the array, it is still an event-controlled loop. We search until we find what we are looking for or reach the end of the list. Finding something is an event, so we use an event loop.

Next, note that there are two tests in the limit expression of the loop. We have coded the test for the end of the array first. In this case it doesn't make any difference which test is first from an execution standpoint, but in other search loops it might. Therefore, you should develop the habit of coding the limit test first because it doesn't use an indexed value and is therefore safer.

The call-by-address use for **`locn`** also merits discussion. Because we need to pass the found location back to the variable in the calling program, we need to pass its address to the search.

Notice how succinct this function is. In fact, there are more lines of documentation than there are lines of code. The entire search is contained in one *while* statement. With this short code, you might be tempted to ask, "Why write the function at all? Why not just put the one line of code wherever it is needed?" The answer lies in the structured programming concepts that each function should do only one thing and in the concept of reusability. By isolating the search process in its own function, we separate it from the process that needs the search. This is a better approach to structured programming and also makes the code reusable in other parts of the program and portable to other programs that require searches.

Binary Search In C

The C implementation of the binary search algorithm is shown in Program 13-2.

PROGRAM 13-2 Binary Search

```
1  /* Search an ordered list using Binary Search.
2       Pre  list must contain at least one element
3            last is index to largest element in list
4            target is value of element being sought
5            locn is address of index in calling function
6       Post FOUND: locn = index to target element
7                    return true (found)
8            NOT FOUND: locn = index below/above target
9                       return false (not found)
```

continued

PROGRAM 13-2 Binary Search *(continued)*

```
10  */
11  bool binarySearch   (int list[ ], int  last,
12                       int target,  int* locn )
13  {
14  // Local Definitions
15     int begin;
16     int mid;
17     int end;
18
19  // Statements
20     begin = 0;
21     end  = last;
22     while (begin <= end)
23        {
24        mid = ( begin + end ) / 2;
25        if ( target > list[ mid ] )
26           // look in upper half
27           begin = mid + 1;
28           else if ( target < list[ mid ] )
29              // look in lower half
30              end = mid - 1;
31        else
32              // found: force exit
33              begin = end + 1;
34        } // end while
35     *locn = mid;
36     return (target == list [mid]);
37  }  // binarySearch
```

13.3 Hashed List Searches

The search techniques discussed in Section 13.1, "List Searches," require several tests before we can find the data. In an ideal search, we would know exactly where the data are and go directly there. This is the goal of a hashed search: to find the data with only one test. For our discussion we use an array of data. The general concept is easily extended to other structures, such as files stored on a disk. These structures are beyond the scope of this text, however, and their discussion is left to other books.

Basic Concepts

In a **hashed search**, the key, through an algorithmic function, determines the location of the data. Because we are searching an array, we use a hashing algorithm to transform the key into the index that contains the data we need to locate. Another way to describe hashing is as a key-to-address transformation in which the keys map to addresses in a list. This mapping transformation is

shown in Figure 13-6. At the top of the figure is a general representation of the hashing concept. The rest of the figure shows how three keys might hash to three different addresses in the list.

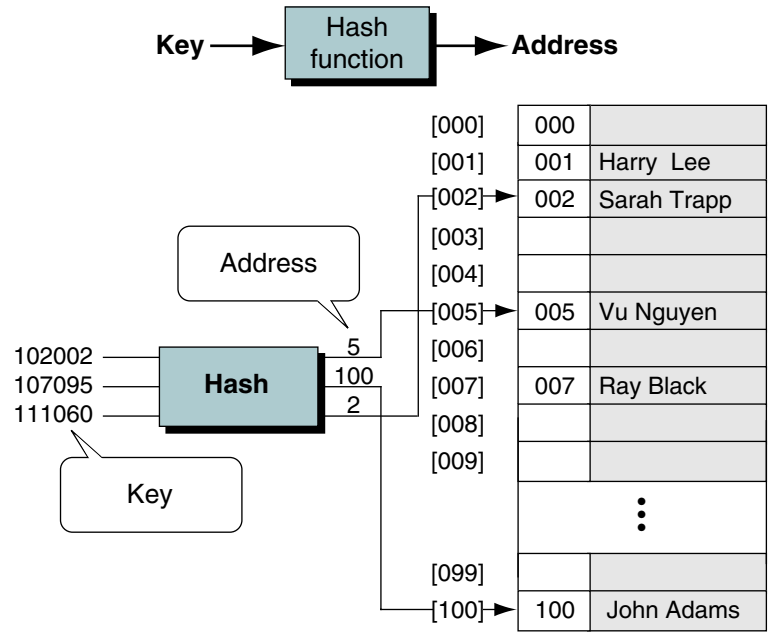


FIGURE 13-6 Hash Concept

Generally, the population of keys for a hashed list is greater than the storage area for the data. For example, if we have an array of 50 students for a class in which the students are identified by the last four digits of their Social Security numbers, there are 200 possible keys for each element in the array (10,000 / 50). Because there are many keys for each index location in the array, more than one student may hash to the same location in the array. We call the set of keys that hash to the same location in our list **synonyms**.

Hashing is a key-to-address mapping process.

If the actual data that we insert into our list contain two or more synonyms, we can have collisions. A **collision** occurs when a hashing algorithm produces an address for an insertion key and that address is already occupied. The address produced by the hashing algorithm is known as the **home address**. The memory that contains all of the home addresses is known as the **prime area**.

When two keys collide at a home address, we must resolve the collision by placing one of the keys and its data in another location. The collision resolution concept is shown in Figure 13-7.

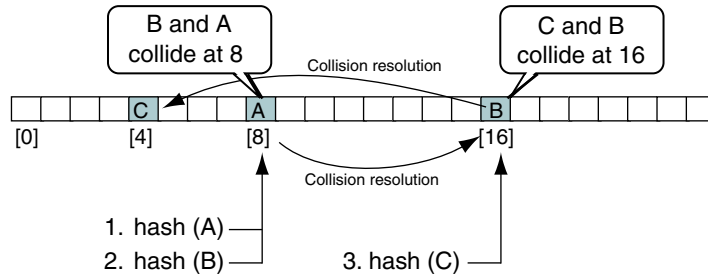


FIGURE 13-7 Collision Resolution Concept

In Figure 13-7 we hash key `A` and place it at location 8 in the list. At some later time, we hash key `B`, which is a synonym of `A`. Because they are synonyms, they both hash to the same home address and a collision results. We resolve the collision by placing key `B` at location 16. When we hash key `C`, its home address is 16. Although `B` and `C` are not synonyms, they still collide at location 16 because we placed key `B` there when we resolved the earlier collision. We must therefore find another location for key `C`, which we place at location 4.

It should be obvious that when we need to locate an element in a hashed list, we must use the same algorithm that we used to insert it into the list. Consequently, we first hash the key and check the home address to determine whether it contains the desired element. If it does, the search is complete. If not, we must use the collision resolution algorithm to determine the next location and continue until we find the element or determine that it is not in the list. Each calculation of an address and test for success is known as a **probe.**

Hashing Methods

We are now ready to study several hashing methods. After we look at the different methods, we create a simple hashing algorithm that incorporates several of them. The hashing techniques that we study are shown in Figure 13-8.

Direct Method

In **direct hashing** the key is the address without any algorithmic manipulation. The data structure must therefore contain an element for every possible key. The situations in which you can use direct hashing are limited, but it can be very powerful because it guarantees that there are no synonyms and therefore no collisions. Let's look at two applications.

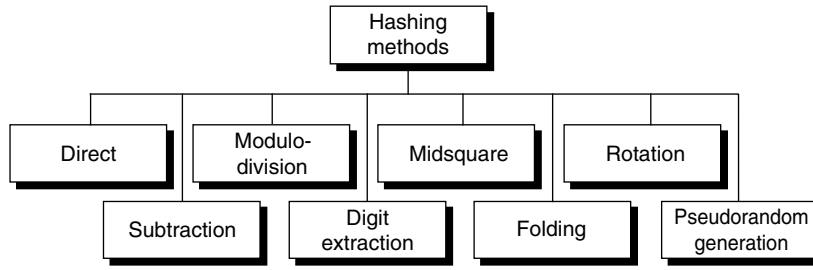


FIGURE 13-8 Basic Hashing Techniques

First consider the problem in which we need to total monthly sales by the days of the month. For each sale we have the date and the amount of the sale. In this case we create an array of 31 accumulators. As we read the sales records for the month, we use the day of the month as the key for the array and add the sale amount to the corresponding accumulator. The accumulation code is shown in the following example.[4]

```
dailySales[sale.day] = dailySales[sale.day]
                     + sale.amount;
```

Now let's consider a more complex example. Imagine that a small organization has fewer than 100 employees. Each employee is assigned an employee number between 1 and 100. In this case, if we create an array of 101 employee records (location 0 is not used), the employee number can be directly used as the address of any individual record. This concept is shown in Figure 13-9.

As you study Figure 13-9, note that not every element in the array contains an employee's record. Although every element was used in our daily sales example, more often than not there are some empty elements in hashed lists. In fact, as we will see later, all hashing techniques other than direct hashing require that some of the elements be empty to reduce the number of collisions.

As you may have noticed, although this is the ideal method, its application is very limited. For example, we cannot have the Social Security number as the key using this method because Social Security numbers are nine digits. In other words, if we use the Social Security number as the key, we need an array as large as 1,000,000,000 entries, but we would use fewer than 100 of them.

We now turn our attention to hashing techniques that map a large population of possible keys into a small address space.

4. Because C arrays start at location 0, we need to have 32 elements in our array. Element 0 is not used.

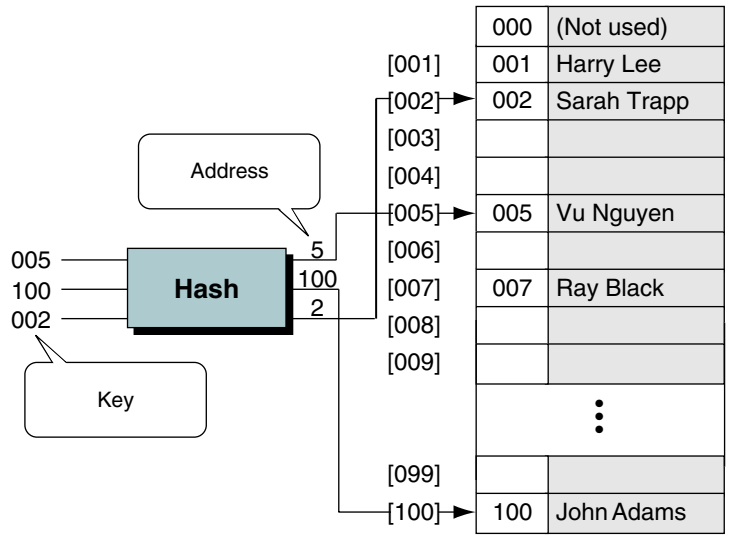


FIGURE 13-9 Direct Hashing of Employee Numbers

Subtraction Method

Sometimes keys are consecutive but do not start from 1. For example, a company may have only 100 employees, but the employee numbers start from 1001 and go to 1100. In this case we use subtraction hashing, a very simple hashing function that subtracts 1000 from the key to determine the address. The beauty of this example is that it is simple and guarantees that there will be no collisions. Its limitations are the same as direct hashing: it can be used only for small lists in which the keys map to a densely filled list.

> The direct and subtraction hash functions both guarantee a search effort of one with no collisions. They are one-to-one hashing methods: only one key hashes to each address.

Modulo-division Method

Also known as **division remainder**, the **modulo-division** method divides the key by the array size and uses the remainder for the address. This method gives us the simple hashing algorithm shown below in which `listSize` is the number of elements in the array:

```
address = key MODULO listSize
```

This algorithm works with any list size, but a list size that is a prime number produces fewer collisions than other list sizes. We should therefore try, whenever possible, to make the array size a prime number.

As our little company begins to grow, we realize that soon we will have more than 100 employees. Planning for the future, we create a new employee numbering system that can handle employee numbers up to 1,000,000. We also decide that we want to provide data space for up to 300 employees. The first prime number greater than 300 is 307. We therefore choose 307 as our list (array) size, which gives us a list with addresses that range from 0 through 306. Our new employee list and some of its hashed addresses are shown in Figure 13-10.

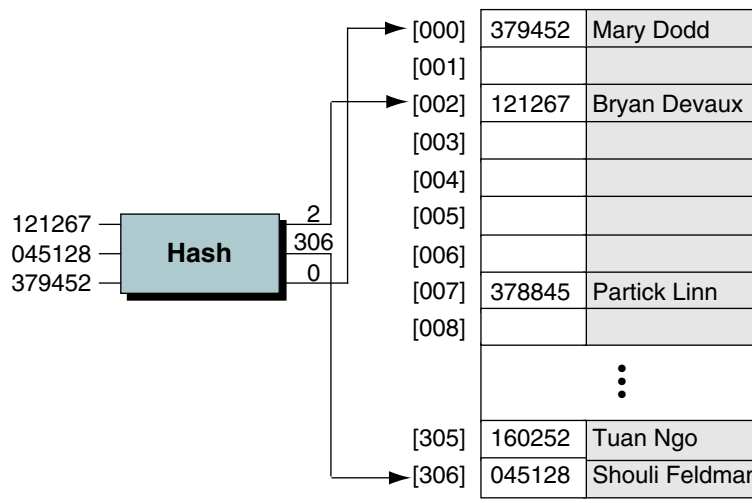


FIGURE 13-10 Modulo-division Hashing

To demonstrate, let's hash Bryan Devaux's employee number, 121267.

```
121267/307 = 395 with remainder of 2

Therefore: hash(121267) = 2
```

Digit-extraction Method

Using **digit extraction** selected digits are extracted from the key and used as the address. For example, using our six-digit employee number to hash to a three-digit address (000–999), we could select the first, third, and fourth digits (from the left) and use them as the address. Using the keys from Figure 13-10, we would hash them to the addresses shown below:

```
379452 ⇨ 394
121267 ⇨ 112
378845 ⇨ 388
160252 ⇨ 102
045128 ⇨ 051
```

Midsquare Method

In **midsquare hashing** the key is squared and the address is selected from the middle of the squared number. The most obvious limitation of this method is the size of the key. Given a key of six digits, the product will be 12 digits, which is beyond the maximum integer size of many computers. Because most personal computers can handle a nine-digit integer, let's demonstrate the concept with keys of four digits. Given a key of 9452, the midsquare address calculation is shown below using a four-digit address (0000–9999).

```
9452² = 89340304: address is 3403
```

As a variation on the midsquare method, we can select a portion of the key, such as the middle three digits, and then use them rather than the whole key. Doing so allows the method to be used when the key is too large to square. For example, for the keys in Figure 13-10, we can select the first three digits and then use the midsquare method as shown below. (We select the third, fourth, and fifth digits as the address.)

```
379452: 379² = 143641 ⇨ 364
121267: 121² = 014641 ⇨ 464
378845: 378² = 142884 ⇨ 288
160252: 160² = 025600 ⇨ 560
045128: 045² = 002025 ⇨ 202
```

Note that in the midsquare method the same digits must be selected from the product. For that reason we consider the product to have sufficient leading zeros to make it the full six digits.

Folding Methods

Two folding methods are used: fold shift and fold boundary. In **fold shift** the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part. For example, imagine that we want to map Social Security numbers into three-digit addresses. We divide the nine-digit Social Security number into three three-digit numbers, which are then added. If the resulting sum is greater than 999, we discard the leading digit. This method is shown in Figure 13-11(a).

In **fold boundary** the left and right numbers are folded on a fixed boundary between them and the center number. The two outside values are thus reversed, as shown in Figure 13-11(b), where 123 is folded to 321 and 789 is folded to 987. It is interesting to note that the two folding methods give different hashed addresses.

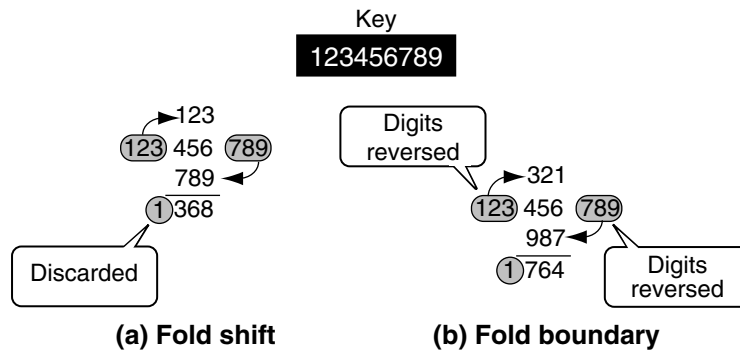


FIGURE 13-11 Hash Fold Examples

Rotation Method

Rotation hashing is generally not used by itself but rather is incorporated in combination with other hashing methods. It is most useful when keys are assigned serially, such as we often see in employee numbers and part numbers. A simple hashing algorithm tends to create synonyms when hashing keys are identical except for the last character. Rotating the last character to the front of the key minimizes this effect. For example, consider the case of a six-digit employee number that might be used in a large company (see Figure 13-12).

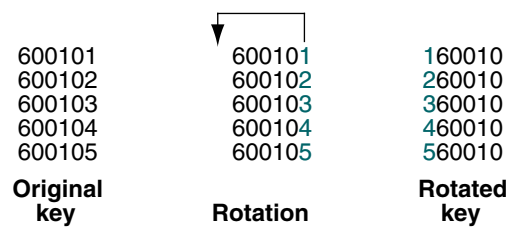


FIGURE 13-12 Rotation Hashing

Examine the rotated key carefully. Because all keys now end in 60010, they would obviously not work well with modulo-division. On the other hand, if we used a simple fold shift hash on the original key and a two-digit address, the addresses would be sequential starting with 62. Using a shift hash on the rotated key results in the series of addresses 26, 36, 46, 56, 66, which has the desired effect of spreading the data more evenly across the address space. Rotation is often used in combination with folding and pseudorandom hashing.

Pseudorandom Hashing

In **pseudorandom hashing** the key is used as the seed in a pseudorandom-number generator, and the resulting random number is then scaled into the possible address range using modulo-division (see "Modulo-division method," earlier in this section). Do not confuse pseudorandom numbers with random numbers. Given a fixed seed, pseudorandom-number generators always generate the same series of numbers. That is what allows us to use them in hashing.

A common random-number generator is shown below.

```
y = ax + c
```

To use the pseudorandom-number generator as a hashing method, we set x to the key, multiply it by the coefficient `a`, and then add the constant `c`. The result is then divided by the list size, with the remainder being the hashed address. For maximum efficiency, the factors `a` and `c` should be prime numbers. Let's demonstrate the concept with an example from Figure 13-10. To keep the calculation reasonable, we use 17 and 7 for factors `a` and `c`, respectively. Also, the list size in the example is the prime number 307.

```
y = ((17 * 121267) + 7) modulo 307
y = (2061539 + 7) modulo 307
y = 2061546 modulo 307
y = 41
```

We will see this pseudorandom-number generator again when we discuss collision resolution.

All hash functions except direct hashing and subtraction hashing are many-to-one functions: many keys hash to one address.

One Hashing Algorithm

Before we conclude our discussion of hashing methods, we need to describe a complete hashing algorithm. Although a hashing method may work well when we hash a key to an address in an array, hashing to large files is generally more complex. It often requires extensive analysis of the population of keys to be hashed to determine the number of synonyms and the length of the collision series produced by the algorithm. The study of such analysis is beyond the scope of this text.

In this section we present the algorithm that could be used for a large file. It is a simplification of an algorithm used to hash keys in an industrial database system. As you study it, note that it uses three different hashing methods: fold shift, rotation, and modulo-division.

Assume that we have an alphanumeric key consisting of up to 30 bytes that we need to hash into a 32-bit address. The first step is to convert the alphanumeric key into a number by adding the American Standard Code for Information Interchange (ASCII) value for each character to an accumulator

that will be the address. As each character is added, we rotate the bits in the address to maximize the distribution of the values. After the characters in the key have been completely hashed, we take the absolute value of the address and then map it into the address range for the file. This logic is shown in Algorithm 13-6.

ALGORITHM 13-6 Hashing Algorithm

```
Algorithm hash (key, size, maxAddr, addr)
This algorithm converts an alphanumeric key of size
characters into an integral address.
   Pre   key is a key to be hashed
         size is the number of characters in the key
         maxAddr is maximum possible address for the list
   Post  addr contains the hashed address
 1 set looper to 0
 2 set addr to 0
   Hash key
 3 for each character in key
   1   if (character not space)
       1   add character to address
       2   rotate addr 12 bits right
   2   end if
 4 end loop
   Test for negative address
 5 if (addr < 0)
   1   addr = absolute(addr)
 6 end if
 7 addr = addr modulo maxAddr
end hash
```

Algorithm 13-6 Analysis

Two points merit discussion in this algorithm. First, the rotation in statement 3.1.2 can often be accomplished by an assembly language instruction. If the algorithm is written in a high-level language, the rotation is accomplished by a series of bitwise operations. For our purposes it is sufficient that the 12 bits at the end of the address are shifted to be the 12 bits at the beginning of the address, and the bits at the beginning are shifted to occupy the bit locations at the right.

Second, this algorithm actually uses three of the hashing methods discussed previously. We use fold shift when we add the individual characters to the address. We use rotation when we rotate the address after each addition. Finally, we use modulo-division when we map the hashed address into the range of available addresses.

13.4 Collision Resolution

With the exception of the direct and subtraction methods, none of the methods used for hashing is one-to-one mapping. Thus, when we hash a new key to an address, we may create a collision. There are several methods for handling

collisions, each of them independent of the hashing algorithm. That is, each hashing method can be used with each of the collision resolution methods. In this section we discuss the **collision resolution** methods shown in Figure 13-13.

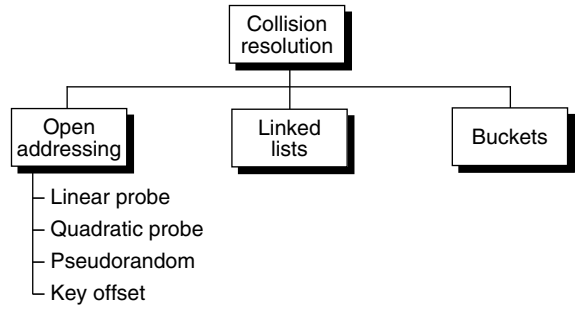


FIGURE 13-13 Collision Resolution Methods

Before we discuss the collision resolution methods, however, we need to cover three more concepts. Because of the nature of hashing algorithms, there must be some empty elements in a list at all times. In fact, we define a full list as a list in which all elements except one contain data. As a rule of thumb, a hashed list should not be allowed to become more than 75% full. This guideline leads us to our first concept: load factor. The **load factor** of a hashed list is the number of elements in the list divided by the number of physical elements allocated for the list, expressed as a percentage. Traditionally, load factor is assigned the symbol alpha (α). The formula in which k represents the number of filled elements in the list and n represents the total number of elements allocated to the list is

$$\alpha = \frac{k}{n} \times 100$$

As data are added to a list and collisions are resolved, some hashing algorithms tend to cause data to group within the list. This tendency of data to build up unevenly across a hashed list is known as **clustering**, our second concept. Clustering is a concern because it is usually created by collisions. If the list contains a high degree of clustering, the number of probes to locate an element grows and reduces the processing efficiency of the list.

Computer scientists have identified two distinct types of clusters. The first, **primary clustering**, occurs when data cluster around a home address. Primary clustering is easy to identify. Consider, for example, the population clusters found in the United States. If you have ever flown across the country on a clear night, you noticed that amid the darkness towns and cities were identified by their lights. If the whole country were a hashed list, and the

lights each represented an element of data, we would be looking at primary clustering—clustering around a home address in our list.

Secondary clustering occurs when data become grouped along a collision path throughout a list. This type of clustering is not easy to identify. In secondary clustering the data are widely distributed across the whole list, so the list appears to be well distributed. If the data all lie along a well-traveled collision path, however, the time to locate a requested element of data can increase.

Figure 13-14 diagrams both types of clustering. In Figure 13-14(a) we see that K, P, and Y cluster around K's home address. In this example the collision resolution is based on the home address. Q, on the other hand, hashes to its own address remotely located from K's home address. In Figure 13-14(b) the same four keys are inserted into the list. The collision resolution in this example is not based on the home address. It spreads the collisions across the entire list. Thus, we see that while K, P, and Y are still synonyms that hash to the same home address, they are not clustered around K's home address; they are spread across the file. Also note that because P was placed in Q's home address, a secondary collision was created.

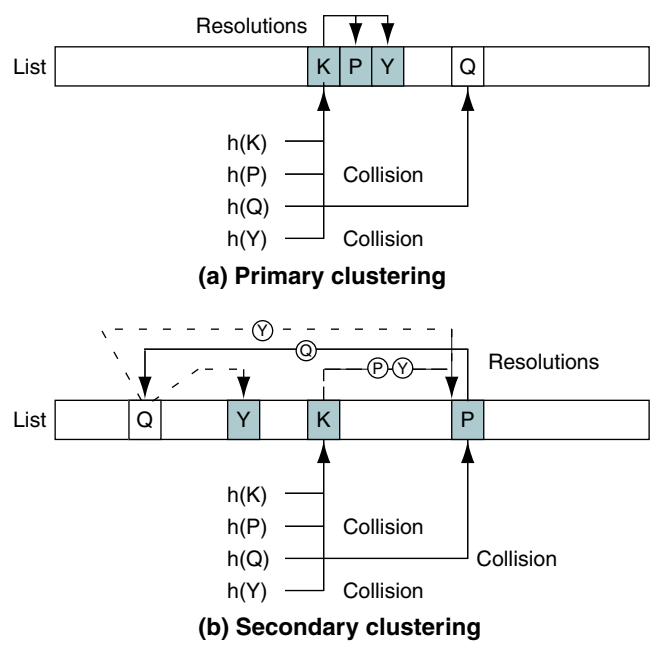


FIGURE 13-14 Clustering

To better understand the effects of secondary clustering, consider an extreme example: Assume that we have a hashing algorithm that hashes each key to the same home address. Locating the first element inserted into the list

takes only one probe. Locating the second element takes two probes. Carrying the analogy to its conclusion, locating the *n*th element added to the list takes *n* probes, even if the data are widely distributed across the addresses in the list.

From this discussion it should be apparent that we need to design our hashing algorithms to minimize clustering, both primary and secondary. However, note that with the exception of the direct and subtraction methods, we cannot eliminate collisions.

One of the surprises of hashing methods is how few elements need to be inserted into a list before a collision occurs. This concept is easier to understand if we recall a common party game used to encourage mingling. The host prepares a list of topics, and each guest is required to find one or more guests who satisfy each topic. One topic is often birthdays and requires that the guests find two people who have the same birthday (the year is not counted). If there are more than 23 party guests, chances are better than 50% that two of them have the same birthday.[5] Extrapolating this phenomenon to our hashing algorithms, if we have a list with 365 addresses, we can expect to get a collision within the first 23 inserts more than 50% of the time.

Our final concept is that the number of elements examined in the search for a place to store the data must be limited. The traditional limit of examining all elements of the list presents three difficulties. First, the search is not sequential, so finding the end of the list doesn't mean that every element has been tested. Second, examining every element would be excessively time-consuming for an algorithm that has as its goal a search effort of one. Third, some of the collision resolution techniques cannot physically examine all of the elements in a list. (For an example, see "Quadratic Probe," in the next section.)

Computer scientists therefore generally place a collision limit on hashing algorithms. What happens when the limit is reached depends on the application. One simple solution is to abort the program. A more elegant solution is to store the data in an area separate from the list (see "Linked List Collision Resolution" later in the chapter). Whatever the solution it is important to use the same algorithmic limit when searching for data in the list.

We are now ready to look at some collision resolution methods. Generally, there are two different approaches to resolving collisions: open addressing and linked lists. A third concept—buckets—defers collisions but does not prevent them.

Open Addressing

The first collision resolution method, **open addressing**, resolves collisions in the prime area—that is, the area that contains all of the home addresses. This

5. This mathematical fact was first documented by von Mises in the 1930s and is known as the von Mises birthday paradox.

technique is opposed to linked list resolution, in which the collisions are resolved by placing the data in a separate overflow area.

When a collision occurs, the prime area addresses are searched for an open or unoccupied element where the new data can be placed. We discuss four different methods: linear probe, quadratic probe, double hashing, and key offset.

Linear Probe

In a **linear probe**, which is the simplest, when data cannot be stored in the home address we resolve the collision by adding 1 to the current address. For example, let's add two more elements to the modulo-division method example in Figure 13-10. The results are shown in Figure 13-15. When we insert key 070918, we find an empty element and insert it with no collision. When we try to insert key 166702, however, we have a collision at location 001. We try to resolve the collision by adding 1 to the address and inserting the new data at location 002. However, this address is also filled. We therefore add another 1 to the address and this time find an empty location, 003, where we can place the new data.

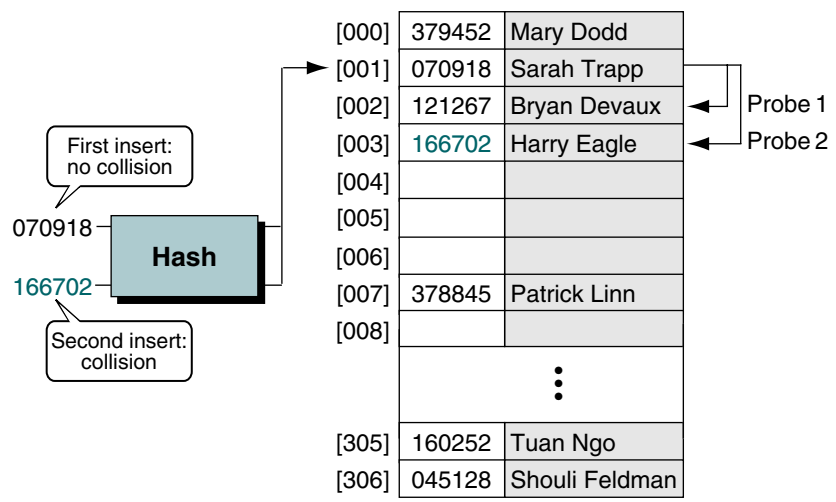


FIGURE 13-15 Linear Probe Collision Resolution

As an alternative to a simple linear probe, we can add 1, subtract 2, add 3, subtract 4, and so forth until we locate an empty element. For example, given a collision at location 341, we would try 342, 340, 343, 339, and so forth until we located an empty element.

In either method the code for the linear probe must ensure that the next collision resolution address lies within the boundaries of the list. Thus, if a key hashes to the last location in the list, adding 1 must produce the address of

the first element in the list. Similarly, if the key hashes to the first element of the list, subtracting 1 must produce the address of the last element in the list.

Linear probes have two advantages. First, they are quite simple to implement. Second, data tend to remain near their home address. This tendency can be important in implementations for which being near the home address is important, such as when we hash to a disk address. On the other hand, linear probes tend to produce primary clustering. Additionally, they tend to make the search algorithm more complex, especially after data have been deleted.

Quadratic Probe

Primary clustering, although not necessarily secondary clustering, can be eliminated by adding a value other than 1 to the current address. One easily implemented method is to use the **quadratic probe**. In the quadratic probe, the increment is the collision probe number squared. Thus for the first probe we add 1^2, for the second collision probe we add 2^2, for the third collision probe we add 3^2, and so forth until we either find an empty element or we exhaust the possible elements. To ensure that we don't run off the end of the address list, we use the modulo of the quadratic sum for the new address. This sequence is shown in Table 13-2, which for simplicity assumes a collision at location 1 and a list size of 100.

Probe number	Collision location	Probe^2 and increment	New address
1	1	$1^2 = 1$	1 + 1 ⇨ 02
2	2	$2^2 = 4$	2 + 4 ⇨ 06
3	6	$3^2 = 9$	6 + 9 ⇨ 15
4	15	$4^2 = 16$	15 + 16 ⇨ 31
5	31	$5^2 = 25$	31 + 25 ⇨ 56
6	56	$6^2 = 36$	56 + 36 ⇨ 92
7	92	$7^2 = 49$	92 + 49 ⇨ 41
8	41	$8^2 = 64$	41 + 64 ⇨ 05
9	5	$9^2 = 81$	5 + 81 ⇨ 86
10	86	$10^2 = 100$	86 + 100 ⇨ 86

TABLE 13-2 Quadratic Collision Resolution Increments

A potential disadvantage of the quadratic probe is the time required to square the probe number. We can eliminate the multiplication factor, however, by using an increment factor that increases by 2 with each probe. Adding the increment factor to the previous increment gives us the next increment, which as you can see by the last column in Table 13-2 is the equivalent of the probe squared.

The quadratic probe has one limitation: it is not possible to generate a new address for every element in the list. For example, in Table 13-2 only 59 of the probes can generate unique addresses. The other 41 locations in the list are not probed. The first duplicate address is found in probe 10. To see more examples

of duplicate addresses, extend the table several probes. The solution to this problem is to use a list size that is a prime number. When the list size is a prime number, at least half of the list is reachable, which is a reasonable number.

Pseudorandom Collision Resolution

The last two open addressing methods are collectively known as **double hashing**. In each method, rather than use an arithmetic probe function, the address is rehashed. As will be apparent from the discussion, both methods prevent primary clustering.

Pseudorandom collision resolution uses a pseudorandom number to resolve the collision. We saw the pseudorandom-number generator as a hashing method in the "Pseudorandom Hashing" section earlier in the chapter. We now use it as a collision resolution method. In this case, rather than use the key as a factor in the random-number calculation, we use the collision address. Consider the collision we created in Figure 13-15. We now resolve the collision using the following pseudorandom-number generator, where `a` is 3 and `c` is 5:

```
y = (ax + c) modulo listSize
  = (3 × 1 + 5) Modulo 307
  = 8
```

In this example we resolve the collision by placing the new data in element 008 (Figure 13-16). We have to keep the coefficients small to fit our example. A better set of factors would use a large prime number for `a`, such as 1663.

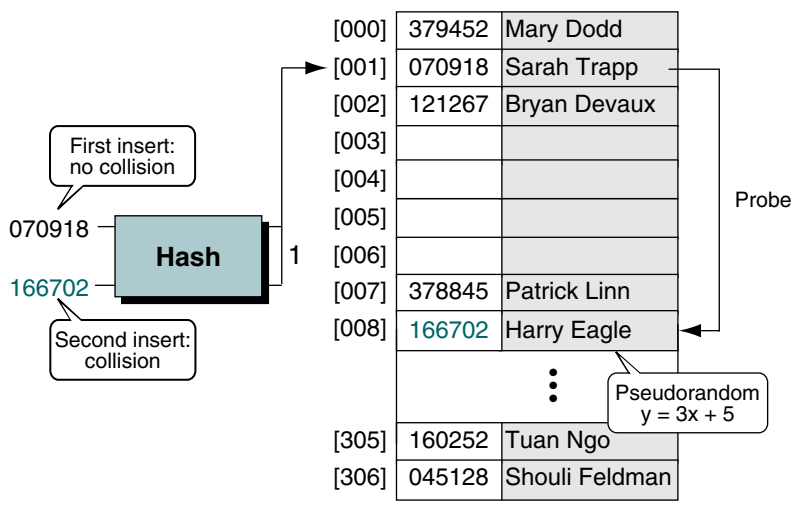


FIGURE 13-16 Pseudorandom Collision Resolution

Pseudorandom numbers are a relatively simple solution, but they have one significant limitation: all keys follow only one collision resolution path through the list. (This deficiency also occurs in the linear and quadratic probes.) Because pseudorandom collision resolution can create significant secondary clustering, we should look for a method that produces different collision paths for different keys.

Key Offset

Key offset is a double hashing method that produces different collision paths for different keys. Whereas the pseudorandom-number generator produces a new address as a function of the previous address, key offset calculates the new address as a function of the old address and the key. One of the simplest versions simply adds the quotient of the key divided by the list size to the address to determine the next collision resolution address, as shown in the formula below.

```
offSet  = ⌊key/listSize⌋
address = ((offSet + old address) modulo listSize)
```

For example, when the key is 166702 and the list size is 307, using the modulo-division hashing method generates an address of 1. As shown in Figure 13-16, this synonym of 070918 produces a collision at address 1. Using key offset to calculate the next address, we get 237, as shown below.

```
offSet  = ⌊166702/307⌋ = 543
address = ((543 + 001) modulo 307) = 237
```

If 237 were also a collision, we would repeat the process to locate the next address, as shown below.

```
offSet  = ⌊166702/307⌋ = 543
address = ((543 + 237) modulo 307) = 166
```

To really see the effect of key offset, we need to calculate several different keys, all hashing to the same home address. In Table 13-3 we calculate the next two collision probe addresses for three keys that collide at address 001.

Key	Home address	Key offset	Probe 1	Probe 2
166702	1	543	237	166
572556	1	1865	024	047
067234	1	219	220	132

TABLE 13-3 Key-offset Examples

Note that each key resolves its collision at a different address for both the first and the second probes.

Linked List Collision Resolution

A major disadvantage to open addressing is that each collision resolution increases the probability of future collisions. This disadvantage is eliminated in the second approach to collision resolution: linked lists. A linked list is an ordered collection of data in which each element contains the location of the next element. For example, in Figure 13-17 array element 001, Sarah Trapp, contains a pointer to the next element, Harry Eagle, which in turn contains a pointer to the third element, Chris Walljasper (when the link location is crossed out, the link is null). We studied the maintenance of linked lists in Chapter 5.

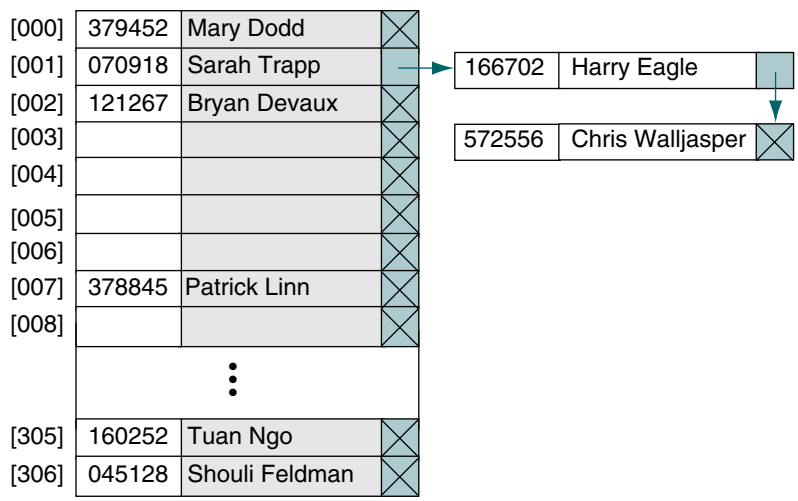


FIGURE 13-17 Linked List Collision Resolution

Linked list collision resolution uses a separate area to store collisions and chains all synonyms together in a linked list. It uses two storage areas: the prime area and the **overflow area**. Each element in the prime area contains an additional field—a link head pointer to a linked list of overflow data in the overflow area. When a collision occurs, one element is stored in the prime area and chained to its corresponding linked list in the overflow area. Although the overflow area can be any data structure, it is typically implemented as a linked list in dynamic memory. Figure 13-17 shows the linked list from Figure 13-16 with the three synonyms for address 001.

The linked list data can be stored in any order, but a last in–first out (LIFO) sequence or a key sequence is the most common. The LIFO sequence

is the fastest for inserts because the linked list does not have to be scanned to insert the data. The element being inserted into overflow is simply placed at the beginning of the linked list and linked to the node in the prime area. Key-sequenced lists, with the key in the prime area being the smallest, provide for faster search retrieval. Which sequence (LIFO or key sequence) is used depends on the application.

Bucket Hashing

Another approach to handling the collision problems is **bucket hashing**, in which keys are hashed to **buckets**, nodes that accommodate multiple data occurrences. Because a bucket can hold multiple data, collisions are postponed until the bucket is full. Assume, for example, that in our Figure 13-17 list each address is large enough to hold data about three employees. Under this assumption a collision would not occur until we tried to add a fourth employee to an address. There are two problems with this concept. First, it uses significantly more space because many of the buckets are empty or partially empty at any given time. Second, it does not completely resolve the collision problem. At some point a collision occurs and needs to be resolved. When it does, a typical approach is to use a linear probe, assuming that the next element has some empty space. Figure 13-18 demonstrates the bucket approach.

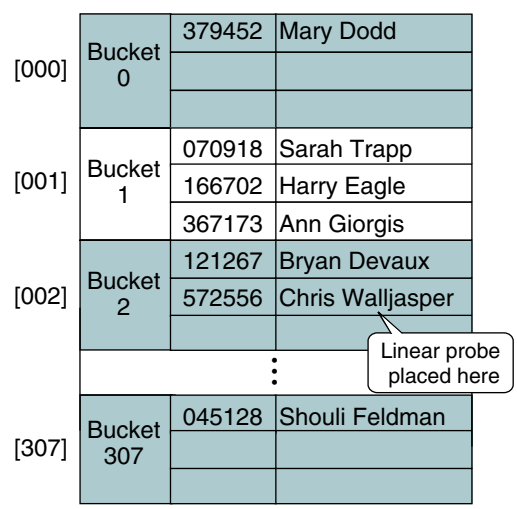


FIGURE 13-18 Bucket Hashing

Study the second bucket in Figure 13-18. Note that it contains the data for three entries, all of which hashed to address 1. We do not get a collision until the fourth key, 572556 in our example, is inserted into the list. When a collision finally occurs—that is, when the bucket is full—any of the collision

resolution methods may be used. For example, in Figure 13-18 a collision occurrs when we insert 572556 because bucket 1 was full. We then use a linear probe to insert it into location 2. Also note that for efficiency we place the keys within a bucket in ascending key sequence.

Combination Approaches

There are several approaches to resolving collisions. As we saw with the hashing methods, a complex implementation often uses multiple steps. For example, one large database implementation hashes to a bucket. If the bucket is full, it uses a set number of linear probes, such as three, to resolve the collision and then uses a linked list overflow area.

Hashed List Example

To demonstrate hashed searches, let's create a program that looks up telephone numbers. Our input is a simple text file that contains names and phone numbers. When the program begins, it reads the file and loads the data into an array. We then use a hash to find phone numbers. We use a variation of Algorithm 13-6 for the hashing and a linear probe for the collision resolution.

We use a list size of 53 because it is a prime number. We also include two sets of names that are synonyms. Julie and Chris both hash to location 38; Wayn and Wyan both hash to location 52, the last location in our hash list. We can thus test our collision resolution at the end of the list. Our code is shown in Program 13-3.

PROGRAM 13-3 Hashed List Search

```
/* The telephone lookup program using hashing.
   The input is a file of names and phone numbers.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Global Declarations
typedef struct
   {
    char  name [31];
    char  phone[16];
   } LISTING;

const int cMax_Size = 53;

// Prototype Declarations
   void buildList  (LISTING  phoneList[], int* last);
```

continued

PROGRAM 13-3 Hashed List Search *(continued)*

```
21    void hashSearch (LISTING* phoneList,   int  last);
22    int hashKey     (char*    key,         int  last);
23    int collision   (int      last,        int  locn);
24
25 int main (void)
26 {
27 // Local Definitions
28    LISTING  phoneList[cMax_Size];
29    int      last;
30
31 // Statements
32    printf("Begin Phone Listing\n");
33
34    last = cMax_Size - 1;
35    buildList (phoneList, &last);
36    hashSearch(phoneList,  last);
37
38    printf("\nEnd Phone Listing\n");
39    return 0;
40 }  // main
41
42 /* ==================== buildList ====================
43    Read phone number file and load into array.
44       Pre  phoneList is array to be filled
45            last is index to last element loaded
46       Post array filled
47 */
48 void buildList (LISTING phoneList[], int* last)
49    {
50 // Local Definitions
51    FILE*   fpPhoneNums;
52    LISTING aListing;
53    int     locn;
54    int     cntCol;
55    int     end;
56
57 // Statements
58    fpPhoneNums = fopen ("P13-03.TXT", "r");
59    if (!fpPhoneNums)
60       {
61        printf("Can't open phone file\a\n");
62        exit (100);
63       } // if
64
65    // Set keys to null
66    for (int i = 0; i <= *last; i++)
67        phoneList[i].name[0] = '\0';
68
```

continued

PROGRAM 13-3 Hashed List Search *(continued)*

```
   while (!feof(fpPhoneNums))
 {
       fscanf(fpPhoneNums, " %30[^;]%*c %[^;]%*c",
              aListing.name, aListing.phone);
       locn = hashKey(aListing.name, *last);

       if (phoneList[locn].name[0] != '\0')
          {
           // Collision
           end     = *last;
           cntCol  =  0;
           while (phoneList[locn].name[0] != '\0'
               && cntCol++ <= *last)
                  locn = collision(*last, locn);

           if (phoneList[locn].name[0] != '\0')
              {
               printf("List full. Not all read.\n");
               return;
              } // if full list
           } // if collision
       phoneList[locn] = aListing;
      } // while
   return;
}  // buildList

/* ==================== hashKey ====================
   Given key, hash key to location in list.
      Pre  phoneList is hash array
           last is last index in list
           key is string to be hashed
      Post returns hash location
*/
int hashKey (char* key, int last)
{
// Local Definitions
   int  addr;
   int  keyLen;

// Statements
   keyLen = strlen(key);
   addr   = 0;

   for (int i = 0; i < keyLen; i++)
       if (key[i] != ' ')
          addr += key[i];
   return (addr % last + 1);
}  // hashKey
```

continued

PROGRAM 13-3 Hashed List Search *(continued)*

```
117
118 /* ==================== collision ====================
119    Have a collision. Resolve.
120       Pre  phoneList is hashed list
121            last is index of last element in list
122            locn is address of collision
123       Post returns next address in list
124 */
125 int collision (int  last, int  locn)
126 {
127 // Statements
128     return locn < last ? ++locn : 0;
129 }  // collision
130
131 /* ==================== hashSearch ====================
132    Prompt user for name and lookup in array.
133       Pre  phoneList has been initialized
134       Post User requested quit
135 */
136 void hashSearch (LISTING* phoneList, int last)
137 {
138 // Local Definitions
139    char  srchName[31];
140    char  more;
141    int   locn;
142    int   maxSrch;
143    int   cntCol;
144
145 // Statements
146    do
147      {
148       printf("Enter name: ");
149       scanf ("%s", srchName);
150
151       locn = hashKey (srchName, last);
152        if (strcmp(srchName, phoneList[locn].name) != 0)
153          {
154           // treat as collision
155           maxSrch = last;
156           cntCol  = 0;
157           while (strcmp (srchName,
158                          phoneList[locn].name) != 0
159              &&  cntCol++ <= maxSrch)
160                locn = collision(last, locn);
161          } // if
162
163       // Test for success
164       if (strcmp (srchName, phoneList[locn].name) == 0)
```

continued

PROGRAM 13-3 Hashed List Search *(continued)*

```
165             printf("%-32s (%02d) %-15s\n",
166                     phoneList[locn].name,
167                     locn,
168                     phoneList[locn].phone);
169         else
170             printf("%s not found\n", srchName);
171
172         printf("\nLook up another number <Y/N>? ");
173         scanf (" %c", &more);
174       } while (more == 'Y' || more == 'y');
175   }  // hashSearch
```

```
Results:
   Begin Phone Listing
   Enter name: Julie
   Julie (38)  (555) 916-1212

   Look up another number <Y/N>? y
   Enter name: Chris
   Chris (39)  (555) 946-2859

   Look up another number <Y/N>? y
   Enter name: Wyan
   Wyan (52)  (555) 866-1234

   Look up another number <Y/N>? y
   Enter name: Wayn
   Wayn (0)  (555) 345-0987

   Look up another number <Y/N>? y
   Enter name: Bill
   Bill not found

   Look up another number <Y/N>? n
   End Phone Listing
```

Program 13-3 Analysis

Several points in this program require discussion, although we tried to keep it as simple as possible. As we load the file, we use a function to hash the key. We use the same function to search the list. After hashing the key, we test for a collision. If the name in the hashed location is not a null string, we have a collision. To resolve the collision, we loop until we find an empty location. We use a separate collision function even though there is only one statement for two reasons. First, collision resolution is a separate process that should be placed in its own function because it is used by multiple functions in the program. Second, if we later decide to change the collision resolution algorithm to key offset, for example, we can easily isolate and modify the code.

In the search function, we get the search argument from the user and then call the hash function to determine its location. If the search argument doesn't match the name in the location, we assume a collision. In this case we loop until we find a matching key or until we have examined every location in the list.

The major problem with this simple search algorithm is that it must look in all locations to determine that a key is not in the list. In a productional program with a lot of data, we would have used a slightly different approach. If we used a flag to determine that a location had never been occupied, rather than one that was empty but had previously contained data, we could stop the search when we found an empty location.

13.5 Key Terms

binary search
bucket
bucket hashing
clustering
collision
collision resolution
digit extraction
direct hashing
division remainder
double hashing
fold boundary
fold shift
hashed search
home address
key offset
linear probe
linear search
linked list collision resolution
load factor
midsquare hashing
modulo-division
open addressing
overflow area
primary clustering
prime area
probability search
probe
pseudorandom collision resolution
pseudorandom hashing
quadratic probe
rotation hashing
searching
secondary clustering
sequential search
synonym

13.6 Summary

- Searching is the process of finding the location of a target among a list of objects.
- There are two basic searching methods for arrays: sequential and binary search.
- The sequential search is normally used when a list is not sorted. It starts at the beginning of the list and searches until it finds the data or hits the end of the list.
- One of the variations of the sequential search is the sentinel search. With this method the condition ending the search is reduced to only one by artificially inserting the target at the end of the list.
- The second variation of the sequential search is called the probability search. With this method the list is ordered with the most probable elements at the beginning of the list and the least probable at the end.
- The sequential search can also be used to search a sorted list. In this case we can terminate the search when the target is less than the current element.
- If an array is sorted, we can use a more efficient algorithm called the binary search.
- The binary search algorithm searches the list by first checking the middle element. If the target is not in the middle element, the algorithm eliminates the upper half or the lower half of the list, depending on the value of

the middle element. The process continues until the target is found or the reduced list length becomes zero.

- The efficiency of a sequential list is $O(n)$.
- The efficiency of a binary search is $O(\log n)$.
- In a hashed search, the key, through an algorithmic transformation, determines the location of the data. It is a key-to-address transformation.
- There are several hashing functions, including direct, subtraction, modulo-division, digit extraction, midsquare, folding, rotation, and pseudorandom generation.
  - In direct hashing the key is the address without any algorithmic manipulation.
  - In subtraction hashing the key is transformed to an address by subtracting a fixed number from it.
  - In modulo-division hashing the key is divided by the list size (which is recommended to be a prime number), and the remainder plus 1 is used as the address.
  - In digit-extraction hashing selected digits are extracted from the key and used as an address.
  - In midsquare hashing the key is squared and the address is selected from the middle of the result.
  - In fold shift hashing, the key is divided into parts whose sizes match the size of the required address. Then the parts are added to obtain the address.
  - In fold boundary hashing, the key is divided into parts whose sizes match the size of the required address. Then the left and right parts are reversed and added to the middle part to obtain the address.
  - In rotation hashing the far-right digit of the key is rotated to the left to determine an address. However, this method is usually used in combination with other methods.
  - In the pseudorandom generation hashing, the key is used as the seed to generate a pseudorandom number. The result is then scaled to obtain the address.
- Except in the direct and subtraction methods, collisions are unavoidable in hashing. Collisions occur when a new key is hashed to an address that is already occupied.
- Clustering is the tendency of data to build up unevenly across a hashed list.
  - Primary clustering occurs when data build up around a home address.
  - Secondary clustering occurs when data build up along a collision path in the list.
- To solve a collision, a collision resolution method is used.
- Three general methods are used to resolve collisions: open addressing, linked lists, and buckets.

- The open addressing method can be subdivided into linear probe, quadratic probe, pseudorandom rehashing, and key-offset rehashing.
  - In the linear probe method, when the collision occurs the new data are stored in the next available address.
  - In the quadratic method, the increment is the collision probe number squared.
  - In the pseudorandom rehashing method, a random-number generator is used to rehash the address.
  - In the key-offset rehashing method, an offset is used to rehash the address.
- In the linked list technique, separate areas store collisions and chain all synonyms together in a linked list.
- In bucket hashing a bucket accommodates multiple data occurrences.

13.7 Practice Sets

Exercises

1. An array contains the elements shown below. Using the binary search algorithm, trace the steps followed to find 88. At each loop iteration, including the last, show the contents of `first`, `last`, and `mid`.

```
18 13 17 26 44 56 88 97
```

2. An array contains the elements shown below. Using the binary search algorithm, trace the steps followed to find 20. At each loop iteration, including the last, show the contents of `first`, `last`, and `mid`.

```
18 13 17 26 44 56 88 97
```

3. Using the modulo-division method and linear probing, store the keys shown below in an array with 19 elements. How many collisions occurred? What is the density of the list after all keys have been inserted?

```
224562 137456 214562
140145 214576 162145
144467 199645 234534
```

4. Repeat Exercise 3 using a linked list method for collisions. Compare the results in this exercise with the results you obtained in Exercise 3.

5. Repeat Exercise 3 using the digit-extraction method (first, third, and fifth digits) and quadratic probing.
6. Repeat Exercise 5 using a linked list method for collisions. Compare the results in this exercise with the results you obtained in Exercise 5.
7. Repeat Exercise 3 using the midsquare method, with the center two digits, for hashing. Use a pseudorandom-number generator for rehashing if a collision occurs. Use `a` = 3 and `c` = –1 as the factors.
8. Repeat Exercise 7 using a key-offset method for collisions. Compare the results in this exercise with the results you obtained in Exercise 7.
9. Repeat Exercise 3 using the fold shift method and folding two digits at a time and then use modulo-division on the folded sum.
10. Repeat Exercise 9 using the fold boundary method.
11. Repeat Exercise 3 using the rotation method for hashing. First rotate the far-right digits two to the left and then use digit extraction (first, third, and fifth digits). Use the linear probe method to resolve collisions.
12. Repeat Exercise 11 using a key-offset method for collisions. Compare the results in this exercise with the results you obtained in Exercise 11.

Problems

13. Write a program that creates an array of 100 random integers in the range 1 to 200 and then, using the sequential search, searches the array 100 times using randomly generated targets in the same range. At the end of the program, display the following statistics:

    a. The number of searches completed
    b. The number of successful searches
    c. The percentage of successful searches
    d. The average number of tests per search

    To determine the average number of tests per search, you need to count the number of tests for each search.

    After you run your program, write a paragraph on the similarities or differences between the expected efficiency (big-O) and your calculated results.

14. Repeat Problem 13 using an ordered list search. Rather than use a pseudorandom-number generator, generate a sequenced array of numbers starting with 1 and alternately add 1 and then add 2 to create the next numbers in the series, as shown below.

```
1 3 4 6 7 9 10 12 13 15 6 ... 145 147 148 150
```

    For the search arguments, generate the 100 numbers in the range of 1 to 150.

15. Repeat Problem 14 using a binary search.

Projects

16. Modify Program 13-3 to determine the efficiency of the hashed list search. Run your program with a list of at least 50 names and at least 10 searches. Include at least three search arguments that are not in the list.
17. Run your program from Problem 16 four times. The only differences between the runs should be the load factor. For the first program, use a 60% load factor and then increase it by 10% for each of the following runs. Draw a graph that plots the search efficiency and write a short report about the differences.
18. Modify Program 13-3 to use pseudorandom-number generation to resolve collisions. Write a short report on the differences between the two methods.
19. Write a program that uses a hashing algorithm to create a list of inventory parts and their quantities sold in the past month. After creating the hashed list, write a simple menu-driven user interface that allows the user to select from the following options:

    a. Search for an inventory item and report its quantity sold
    b. Print the inventory parts and their quantities sold
    c. Analyze the efficiency of the hashing algorithm

    The parts data are contained in a text file, as shown in Table 13-4. The key is the three-digit part number. The quantity represents the units sold during the past month.

Part number	Quantity
112	12
130	30
156	56
173	17
197	19
150	50
166	66
113	13
123	12
143	14
167	16
189	18
193	19
117	11
176	76

TABLE 13-4 Data for Hashing Problem

Three outputs are required from your program.

a. Test the following searches and return appropriate messages. You may test other part numbers if you desire, but the following tests must be completed first:
   - Search for 112
   - Search for 126
   - Search for 173

b. When requested, analyze the efficiency of the hashing algorithm for this set of data. Your printout should follow the report format shown below.

```
Percentage of Prime Area Filled:xx%
Average nodes in linked lists:  nn
Longest linked list             nn
```

c. The printout of the entire contents of the list should use the following format:

```
       Home Addr  Prime Area   Overflow List
          0         130/30
          1
          2         112/12
          3         123/12     143/14, 173/17, 193/19
          .
          .
          .
```

20. Create a sequential search ADT. The array to be searched is to be maintained by the application program in its own area. The target may be any type and may be included in a structure. The prototype for the ADT interface is:

```
bool seqSearch (void* ary,
           int   sizeofElem,
           int   numElem,
           int  (*compare)(void* arg1, void* arg2));
```

where `ary` contains the data to be searched, `sizeofElem` is the size of one element in the array, `numElem` is the number of elements in the array, and `compare` is the application function to compare two array elements. The compare function returns –1 if `arg1` < `arg2`, 0 if `arg1` = `arg2`, and +1 if `arg1` > `arg2`.

21. Rewrite Project 20 using a binary search.

This page intentionally left blank

Appendix A
ASCII Tables

This appendix contains the American Standard Code for Information Interchange (ASCII). Table A-1 indicates the decimal, hexadecimal, octal, and symbolic codes with an English interpretation, if appropriate. Table A-2 gives a hexadecimal matrix of all values.

A.1 ASCII Codes (Long Form)

Decimal	Hexadecimal	Octal	Symbol	Interpretation
0	00	00	NUL	NULL value
1	01	01	SOH	Start of heading
2	02	02	STX	Start of text
3	03	03	ETX	End of text
4	04	04	EOT	End of transmission
5	05	05	ENQ	Enquiry
6	06	06	ACK	Acknowledgment
7	07	07	BEL	Ring bell
8	08	10	BS	Backspace
9	09	11	HT	Horizontal tab
10	0A	12	LF	Line feed

continued

TABLE A-1 ASCII Codes

Decimal	Hexadecimal	Octal	Symbol	Interpretation
11	0B	13	VT	Vertical tab
12	0C	14	FF	Form feed
13	0D	15	CR	Carriage return
14	0E	16	SO	Shift out
15	0F	17	SI	Shift in
16	10	20	DLE	Data link escape
17	11	21	DC1	Device control 1
18	12	22	DC2	Device control 2
19	13	23	DC3	Device control 3
20	14	24	DC4	Device control 4
21	15	25	NAK	Negative acknowledgment
22	16	26	SYN	Synchronous idle
23	17	27	ETB	End-of-transmission block
24	18	30	CAN	Cancel
25	19	31	EM	End of medium
26	1A	32	SUB	Substitute
27	1B	33	ESC	Escape
28	1C	34	FS	File separator
29	1D	35	GS	Group separator
30	1E	36	RS	Record separator
31	1F	37	US	Unit separator
32	20	40	SP	Space
33	21	41	!	
34	22	42	"	Double quote
35	23	43	#	
36	24	44	$	

continued

TABLE A-1 ASCII Codes *(continued)*

Decimal	Hexadecimal	Octal	Symbol	Interpretation
37	25	45	%	
38	26	46	&	
39	27	47	'	Apostrophe
40	28	50	(	
41	29	51	)	
42	2A	52	*	
43	2B	53	+	
44	2C	54	,	Comma
45	2D	55	–	Minus
46	2E	56	.	
47	2F	57	/	
48	30	60	0	
49	31	61	1	
50	32	62	2	
51	33	63	3	
52	34	64	4	
53	35	65	5	
54	36	66	6	
55	37	67	7	
56	38	70	8	
57	39	71	9	
58	3A	72	:	Colon
59	3B	73	;	Semicolon
60	3C	74	<	
61	3D	75	=	
62	3E	76	>	
63	3F	77	?	

continued

TABLE A-1 ASCII Codes *(continued)*

Decimal	Hexadecimal	Octal	Symbol	Interpretation
64	40	100	@	
65	41	101	A	
66	42	102	B	
67	43	103	C	
68	44	104	D	
69	45	105	E	
70	46	106	F	
71	47	107	G	
72	48	110	H	
73	49	111	I	
74	4A	112	J	
75	4B	113	K	
76	4C	114	L	
77	4D	115	M	
78	4E	116	N	
79	4F	117	O	
80	50	120	P	
81	51	121	Q	
82	52	122	R	
83	53	123	S	
84	54	124	T	
85	55	125	U	
86	56	126	V	
87	57	127	W	
88	58	130	X	
89	59	131	Y	
90	5A	132	Z	

continued

TABLE A-1 ASCII Codes *(continued)*

Decimal	Hexadecimal	Octal	Symbol	Interpretation
91	5B	133	[	Open bracket
92	5C	134	\	Backslash
93	5D	135	]	Close bracket
94	5E	136	^	Caret
95	5F	137	_	Underscore
96	60	140	`	Grave accent
97	61	141	a	
98	62	142	b	
99	63	143	c	
100	64	144	d	
101	65	145	e	
102	66	146	f	
103	67	147	g	
104	68	150	h	
105	69	151	i	
106	6A	152	j	
107	6B	153	k	
108	6C	154	l	
109	6D	155	m	
110	6E	156	n	
111	6F	157	o	
112	70	160	p	
113	71	161	q	
114	72	162	r	
115	73	163	s	
116	74	164	t	
117	75	165	u	

continued

TABLE A-1 ASCII Codes *(continued)*

Decimal	Hexadecimal	Octal	Symbol	Interpretation
118	76	166	v	
119	77	167	w	
120	78	170	x	
121	79	171	y	
122	7A	172	z	
123	7B	173	{	Open brace
124	7C	174	|	Bar
125	7D	175	}	Close brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

TABLE A-1 ASCII Codes *(continued)*

A.2 ASCII Table (Short Form)

Right / Left	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	l	}	~	DEL

TABLE A-2 Short ASCII Table (Hexadecimal)

Appendix B
Structure Charts

This appendix documents the structure chart concepts and styles used in this book. It includes the basic symbology for our structure charts and some guidelines on their use. Obviously, it is not a tutorial on the design process.

The structure chart is the primary design tool for a program. As a design tool, it is used before you start writing your program. An analogy can help you understand the importance of designing before you start coding.

Assume that you have decided to build a house. You will spend a lot of time thinking about exactly what you want. How many rooms does it need? Do you want a family room or a great room? Should the laundry be inside the house or in the garage? To make sure that everyone understands what you want, you prepare formal blueprints that describe everything in detail. Even if you are building something small, such as a doll house for your child or a tool shed for your backyard, you make some sketches or plans.

Deciding what you want in your house is comparable to determining the requirements for a large system. Drawing up a set of blueprints parallels the structure chart in the design of a program. All require advance planning; only the level of detail changes.

Professionals use the structure chart for another purpose. When you work in a project team environment, you must have your design reviewed before you start writing your program. This review process is called a *structured walk-through*. The review team consists of the systems analyst responsible for your area of the project, a representative of the user community, a system test engineer, and one or two programmers from the project.

Your design walk-through serves three purposes. First, it ensures that you understand how your program fits into the system by communicating your design to the team. If there are any omissions or communication errors, they should be detected here. If you invite programmers who must interface with your program, the walk-through also ensures that the interprogram communication linkages are correct.

Second, it validates your design. In creating your design, you will have considered several alternative approaches to writing your program. The review team expects to see and understand the different designs you considered and hear why you chose the design you are proposing. They will challenge aspects of the design and suggest approaches you may not have considered. The result of the review is the best possible design.

Third, it gives the test engineer the opportunity to assess the *testability* of your program. This step in turn ensures that the final program is robust and as error free as possible.

B.1 Structure Chart Symbols

Figure B-1 shows the various symbols used to write a structure chart. We describe each of these symbols in this section. In addition to discussing the symbols themselves, we cover how to read a structure chart and several rules to follow in your structure charts.

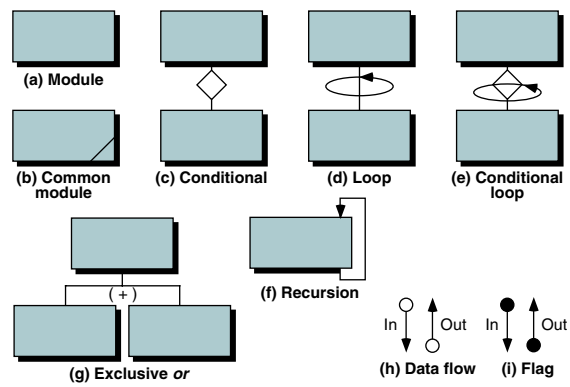


FIGURE B-1 Structure Chart Symbols

Modules

Each rectangle in a structure chart (see Figure B-2) represents a module *that you will write*. Modules that are a part of the implementation language, such as read, write, and square root, are not shown in your structure chart. The name in the rectangle is the name you give to the module. It should be meaningful. The software engineering principle known as *intelligent names* states that the names used in a program should be self-documenting; that is, they should convey their intended usage to the reader. Intelligent names should be used for both modules and data names within your program.

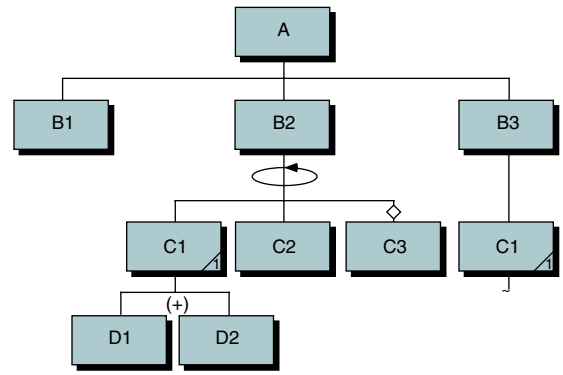


FIGURE B-2 Structure Chart

Although all names should be descriptive, we are going to break our own rule because we want to concentrate on the format of a structure chart rather than a particular program application. The names you see in Figure B-2 identify the various symbols for discussion.

Structure charts show only module flow; they contain no code.

At this point it is helpful to discuss an important structure chart rule: no code is contained in a structure chart. A structure chart shows only the module flow through the program. It is not a block diagram or a flowchart. As a map of your program, the structure chart shows only the logical flow of the modules. The algorithm design shows exactly how each module does its job. Another way of looking at it is that a structure chart shows the big picture; the details are left to algorithm design.

Reading Structure Charts

Before discussing the rest of the symbols, let's look at how to read structure charts. Structure charts are read *top-down, left-right*. Thus, referring to Figure B-2, Module `A` (the first rectangle at the top) consists of three submodules, `B1`, `B2`, and `B3`. According to the left-right rule, the first call in the program is to `B1`. After `B1` is complete, the program calls `B2`. When `B2` is complete, the program calls `B3`. In other words, the modules on the same level of a structure chart are called in order from the left to the right.

The concept of top-down is demonstrated by `B2`. When `B2` is called, it calls `C1`, `C2`, and `C3` in turn. Module `C2` does not start running, however, until `C1` is finished. While `C1` is running, it calls `D1` or `D2`. In other words, all modules in a line from `C1` to `D2` must be called before Module `C2` can start.

Common Modules

There are 10 modules in Figure B-2. Two of them, however, are the same module (C1). When a module is used in more than one place in your design, it is repeated. In other words, if you need to call a common module from several different places within a program, each call is represented by a rectangle in your structure chart. To identify a common module, draw a line in the lower-right corner of the rectangle; see Figure B-1(b). If multiple modules are used multiple times, it helps to add a unique number for each module. We have numbered C1 in Figure B-2, even though it is not necessary in this example, just to illustrate this technique.

One final point about common modules: when common modules call several other modules, their design can become quite large. Rather than redraw all of the submodules each time a common module is called, you can simply indicate that modules are called by including a line below the rectangle and a cut (~) symbol. This concept is also shown in Figure B-2. In this hypothetical design, Module C1 is called by both Module B2 and Module B3. Rather than repeat the design of C1, however, we simply indicate that it calls other modules with a cut symbol when we call it from B3.

Conditional Calls

In Figure B-2, Modules B1, B2, and B3 are always called. Often, however, a module is sometimes called and sometimes passed over. We call this a *conditional call*. It is shown in Figure B-2 when Module B2 calls Module C3. To identify a module as conditional, we use a small diamond, the same symbol used in a flowchart to indicate a selection statement. The code that implements this design is a simple *if* statement, as shown below.

```
1 if (condition)
  1   call C3
2 end if
```

Exclusive *Or*

Two or more modules are often called exclusively. For example, the design for our program in Figure B-2 requires that we call either D1 *or* D2 but not both. This design can be implemented with the following statements in Module C1:

```
1 if (condition)
  1   call D1
2 else
  2   call D2
3 end if
```

It is obvious from the code that either D1 or D2 is called but not both. We need some way to describe this design in the structure chart. The symbol that we use to indicate exclusive calls is (+). Note that if several calls are grouped exclusively, such as in a multiway selection (a *switch* or nested *if* statements in C), we would use the same symbol.

Loops

Very often a design requires that a module be called in a loop. For example, assume that in Figure B-2, B2 calls C1, C2, and C3 while processing a file. It must loop, therefore, until all of the data have been read. The loop symbol is an oval. We use an arrow on our oval, but it is not required. The implementation code for our loop would look something similar to the following piece of code:

```
1 loop (not end of file)
  1  call C1
  2  call C2
  3  if (condition)
     1  call C3
  4  end if
2 end loop
```

Conditional Loops

Sometimes we loop conditionally. For example, consider the following piece of code in which we test for a null linked list before beginning a search. The structure chart symbol for this code combines the conditional diamond with the loop oval, as shown in Figure B-1(e).

```
1 if (list->first not null)
  1  walker = list->first
  2  loop (walker not null)
     1  ...
2 end if
```

Recursion

The last of the module symbols represents a recursive module. It is shown in Figure B-1(f). To see a use of the recursion symbol and as a final example of a structure chart, consider the design of the AVL tree insert as shown in Chapter 8. It is repeated in Figure B-3. This structure chart contains all of the possible structure chart symbols except for a loop. It also presents some interesting design points.

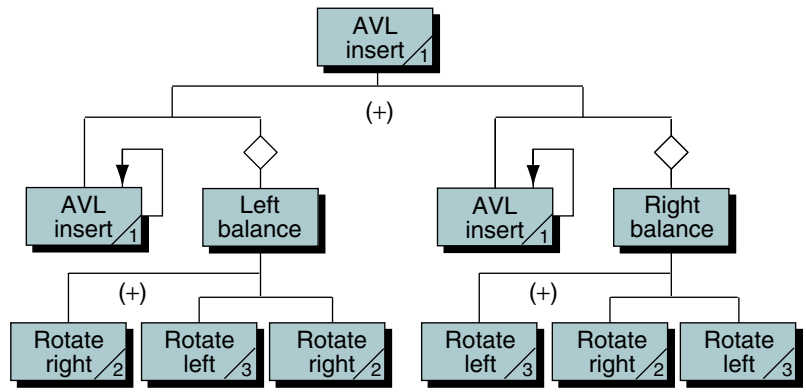


FIGURE B-3 Structure Chart Design

Note that there are two recursive calls in the module, both in the second level. We need two calls because the first one has the left subtree as a parameter and the second one has the right subtree as a parameter. We have used an optional variation of the recursion symbol in this figure, one that doesn't come all the way to the bottom of the rectangle. Also, you should note that the recursion symbol is not generally used on the top rectangle that identifies the name of the module, although it is not considered wrong to do so.

In this design we are inserting a node on either the left or the right branch of the tree. (If you haven't studied trees yet, you may find it a little difficult to understand the following discussion.) Because this is an exclusive *or* design, we place the (+) between the two vertical lines below *AVL insert*.

Looking at the design for inserting on the left, we see that we have a recursive call and then a conditional call to balance the tree. If the tree is still in balance after the insert, we don't need to balance it. To show that these two calls are grouped together, we connect them with a horizontal line, which also separates the exclusive *or* design.

Now look at the design for *Left balance*. Once again we see an exclusive *or,* this time indicating that we have a single rotation to the right or a double rotation to the left and the right. Again, note the use of the horizontal line to group the design steps and isolate the exclusive *or* logic.

Data Flows and Flags

At this point we have covered all of the symbols except for data flows and flags. As a general rule, we do not use them because they quickly break down and are often not maintained. We include them here for completeness.

Data flows and flags represent parameters passed to and from the module. Input parameters are shown on the left of the vertical line connecting the modules, and output parameters are shown on the right of the line. The only

difference between them symbolically is that the data flows have a hollow circle, whereas the flags have a solid circle. In either case, the names of the data flows and the flags should be included on the structure chart.

B.2 Structure Chart Rules

The rules described in this section are summarized below:

1. Each rectangle in a structure chart represents a module written by the programmer. Standard modules provided as a part of the language translator are not included.
2. The name in the rectangle is an intelligent name that communicates the purpose of the module. It is the name that is used in coding the module.
3. The structure chart contains only module flow. No code is indicated.
4. Common modules are indicated by a crosshatch or shading in the lower-right corner of the module's rectangle.
5. Common calls are shown in a structure wherever they are found in the program. If they contain submodule calls, the complete structure need be shown only once.
6. Data flows and flags are optional. When used, they should be named.
7. Input flows and flags are shown on the left of the vertical line; output flows and flags are shown on the right.

This page intentionally left blank

Appendix C
Integer and Float Libraries

This appendix documents two of the more important C libraries. Note that the values shown here are representative only and change from hardware to hardware. Libraries contain unformatted numbers (no commas), often expressed in hexadecimal. We use decimally formatted numbers for readability.

C.1 limits.h

Table C-1 contains hardware specific values for the integer types.

Identifier	Meaning	Minimum Value
CHAR_BIT	bits in a char	8
SCHAR_MIN	short char minimum	–127
SCHAR_MAX	short char maximum	127
UCHAR_MAX	unsigned char maximum	255
CHAR_MIN	char minimum	See SCHAR_MIN
CHAR_MAX	char maximum	See SCHAR_MAX
SHRT_MIN	short int minimum	–32,767
SHRT_MAX	short int maximum	32,767
USHRT_MAX	unsigned short maximum	65,535

continued

TABLE C-1 Partial Contents of Limits Library

Identifier	Meaning	Minimum Value
INT_MIN	int minimum	–32,767
INT_MAX	int maximum	32,767
UINT_MAX	unsigned int maximum	65,535
LONG_MIN	long minimum	–2,147,483,647
LONG_MAX	long maximum	2,147,483,647
ULONG_MAX	unsigned long maximum	4,294,967,295
LLONG_MIN	long long int minimum	$-(2^{63} - 1)$
LLONG_MAX	long long int maximum	$2^{63} - 1$
ULLONG_MAX	unsigned long long maximum	$2^{64} - 1$

TABLE C-1 Partial Contents of Limits Library *(continued)*

C.2 float.h

Table C-2 contains hardware specific values for the floating point types.

Identifier	Meaning	Minimum Value
FLT_DIG	digits of precision	6
DBL_DIG		10
LDBL_DIG		10
DECIMAL_DIG	decimal digits needed to represent floating-point value	10
FLT_MANT_DIG	size of mantissa	none
DBL_MANT_DIG		none
LDBL_MANT_DIG		none
FLT_MIN_EXP	largest integer for negative exponent (float radix)	none
DBL_MIN_EXP		none
LDBL_MIN_EXP		none

continued

TABLE C-2 Partial Contents of Float Library

Identifier	Meaning	Minimum Value
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	largest integer for negative exponent (base 10)	–37 –37 –37
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	largest integer for positive exponent (float radix)	37 37 37
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	largest integer for positive exponent (base 10)	37 37 37
FLT_MAX DBL_MAX LDBL_MAX	largest possible floating–point number	10^{37} 10^{37} 10^{37}
FLT_MIN DBL_MIN LDBL_MIN	smallest possible floating–point number	10^{-37} 10^{-37} 10^{-37}

TABLE C-2 Partial Contents of Float Library *(continued)*

This page intentionally left blank

Appendix D
Selected C Libraries

In this appendix we list most of the standard functions found in the C Language. We have grouped them by library so that related functions are grouped together. We have also listed them alphabetically in Section D.1 for your convenience. Note that not all functions are covered in the text and that there are functions in the libraries that are not covered in this appendix.

D.1 Function Index

Function	Page	Library	Function	Page	Library	Function	Page	Library
_Exit	671	*stdlib*	*fwrite*	670	*stdio*	*remainderf*	667	*math*
abort	671	*stdlib*	*getc*	669	*stdio*	*remainderl*	667	*math*
abs	671	*stdlib*	*getchar*	669	*stdio*	*remquo*	667	*math*
acos	664	*math*	*gets*	670	*stdio*	*remquof*	667	*math*
aosf	664	*math*	*gmtime*	674	*time*	*remquol*	667	*math*
acosl	664	*math*	*ilogb*	666	*math*	*remove*	670	*stdio*
asctime	674	*time*	*ilogbf*	666	*math*	*rename*	670	*stdio*
asin	664	*math*	*ilogbl*	666	*math*	*rewind*	670	*stdio*
asinf	664	*math*	*isalnum*	664	*ctype*	*rint*	667	*math*
asinl	664	*math*	*isalpha*	664	*ctype*	*rintf*	667	*math*
atan	664	*math*	*isascii*	664	*ctype*	*rintl*	667	*math*
atanf	664	*math*	*iscntrl*	664	*ctype*	*round*	667	*math*

continued

continued

Function	Page	Library	Function	Page	Library	Function	Page	Library
expm1l	665	math	log10	666	math	strncpy	673	string
exp2	665	math	log10f	666	math	strpbrk	673	string
exp2f	665	math	log10l	666	math	strrchr	673	string
exp2l	665	math	lround	666	math	strspn	673	string
fabs	665	math	lroundf	666	math	strstr	673	string
fabsf	665	math	lroundl	666	math	strtod	672	stdlib
fabsl	665	math	llround	666	math	strtof	672	stdlib
fclose	668	stdio	llroundf	667	math	strtok	673	string
feof	668	stdio	llroundl	667	math	strtol	672	stdlib
ferror	668	stdio	malloc	671	stdlib	strtold	672	stdlib
fgetc	669	stdio	memchr	672	string	strtoll	672	stdlib
fgets	670	stdio	memcmp	672	string	strtoul	672	stdlib
floor	665	math	memcpy	673	string	system	672	stdlib
floorf	665	math	memmove	673	string	tan	668	math
floorl	665	math	mktime	673	time	tanf	668	math
fmod	665	math	modf	667	math	tanl	668	math
fmodf	665	math	modff	667	math	tanh	668	math
fmodl	665	math	modfl	667	math	tanhf	668	math
fopen	669	stdio	nearbyint	667	math	tanhl	668	math
freopen	669	stdio	nearbyintf	667	math	time	674	time
fprintf	669	stdio	nearbyintl	667	math	tmpfile	670	stdio
fputc	669	stdio	pow	667	math	tmpnam	670	stdio
fputs	670	stdio	powf	667	math	trunc	668	math
fread	670	stdio	powl	667	math	truncf	668	math
free	671	stdlib	printf	669	stdio	truncl	668	math
frexp	666	math	putc	669	stdio	toint	664	ctype
frexpf	666	math	putchar	669	stdio	tolower	664	ctype
frexpl	666	math	puts	670	stdio	toupper	664	ctype
fscanf	669	stdio	rand	671	stdlib	ungetc	669	stdio
fseek	670	stdio	realloc	671	stdlib			
ftell	670	stdio	remainder	667	math			

D.2 Type Library

The following functions are found in `ctype.h`.

```
isalnum     int          isalnum  (int a_char);
isalpha     int          isalpha  (int a_char);
isascii     int          isascii  (int a_char);[a]
iscntrl     int          iscntrl  (int a_char);
isdigit     int          isdigit  (int a_char);
isgraph     int          isgraph  (int a_char);
islower     int          islower  (int a_char);
isprint     int          isprint  (int a_char);
ispunct     int          ispunct  (int a_char);
isspace     int          isspace  (int a_char);
isupper     int          isupper  (int a_char);
isxdigit    int          isxdigit (int a_char);
toint       int          toint    (int a_char);
tolower     int          tolower  (int a_char);
toupper     int          toupper  (int a_char);
```

a. Not standard C. Traditional extension included in most implementations.

D.3 Math Library

The following functions are found in `math.h`.

```
acos     double       acos   (double      number);
acosf    float        acosf  (float       number);
acosl    long double  acosl  (long double number);
asin     double       asin   (double      number);
asinf    float        asinf  (float       number);
asinl    long double  asinl  (long double number);
atan     double       atan   (double      number);
atanf    float        atanf  (float       number);
atanl    long double  atanl  (long double number);
```

continued

```
atan2    double       atan2  (double      number1, double number2);
atan2f   float        atan2f (float       number1, float  number2);
atan2l   long double  atan2l (long double number1,
                              long double number2);
---------------------------------------------------------------------
ceil     double       ceil   (double      number);
ceilf    float        ceilf  (float       number);
ceill    long double  ceill  (long double number);
---------------------------------------------------------------------
cos      double       cos    (double      number);
cosf     float        cosf   (float       number);
cosl     long double  cosl   (long double number);
---------------------------------------------------------------------
cosh     double       cosh   (double      number);
coshf    float        coshf  (float       number);
coshl    long double  coshl  (long double number);
---------------------------------------------------------------------
exp      double       exp    (double      number);
expf     float        expf   (float       number);
expl     long double  expl   (long double number);
---------------------------------------------------------------------
expm1    double       expm1  (double      number);
expm1f   float        expm1f (float       number);
expm1l   long double  expm1l (long double number);
---------------------------------------------------------------------
exp2     double       exp2   (double      number);
exp2f    float        exp2f  (float       number);
exp2l    long double  exp2l  (long double number);
---------------------------------------------------------------------
fabs     double       fabs   (double      number);
fabsf    float        fabsf  (float       number);
fabsl    long double  fabsl  (long double number);
---------------------------------------------------------------------
floor    double       floor  (double      number);
floorf   float        floorf (float       number);
floorl   long double  floorl (long double number);
---------------------------------------------------------------------
fmod     double       fmod   (double number1, double number2);
fmodf    float        fmodf  (float  number1, float  number2);
fmodl    long double  fmodl  (long double number1,
                              long double number2);
---------------------------------------------------------------------
```

continued

```
frexp      double       frexp    (double number, int* exponent);
frexpf     float        frexpf   (float  number, int* exponent);
frexpl     long double  frexpl   (long double number,
                                   int*        exponent);
-----------------------------------------------------------------
ilogb      double       ilogb    (double       number);
ilogbf     float        ilogbf   (float        number);
ilogbl     long double  ilogbl   (long double  number);
-----------------------------------------------------------------
ldexp      double       ldexp    (double       number, int power);
ldexpf     float        ldexpf   (float        number, int power);
ldexpl     long double  ldexpl   (long double  number, int power);
-----------------------------------------------------------------
lint       double       lint     (double       number);
lintf      float        lintf    (float        number);
lintl      long double  lintl    (long double  number);
-----------------------------------------------------------------
lrint      long         lrint    (double       number);
lrintf     long         lrintf   (float        number);
lrintl     long         lrintl   (long double  number);
-----------------------------------------------------------------
llrint     long long    llrint   (double       number);
llrintf    long long    llrintf  (float        number);
llrintl    long long    llrintl  (long double  number);
-----------------------------------------------------------------
log        double       log      (double       number);
logf       float        logf     (float        number);
logl       long double  logl     (long double  number);
-----------------------------------------------------------------
log2       double       log2     (double       number);
log2f      float        log2f    (float        number);
log2l      long double  log2l    (long double  number);
-----------------------------------------------------------------
log10      double       log10    (double       number);
log10f     float        log10f   (float        number);
log10l     long double  log10l   (long double  number);
-----------------------------------------------------------------
lround     long         lround   (double       number);
lroundf    long         lroundf  (float        number);
lroundl    long         lroundl  (long double  number);
-----------------------------------------------------------------
llround    long long    llround  (double       number);
```

continued

```
llroundf    long long    llroundf (float        number);
llroundl    long long    llroundl (long double  number);
-----------------------------------------------------------------------
modf        double       modf     (double number, double* integral);
modff       float        modff    (float  number, float*  integral);
modfl       long double  modfl    (long double  number,
                                    long double* integral);
-----------------------------------------------------------------------
nearbyint   double       nearbyint  (double      number);
nearbyintf  float        nearbyintf (float       number);
nearbyintl  long double  nearbyintl (long double number);
-----------------------------------------------------------------------
pow         double       pow  (double      base, double power);
powf        float        powf (float       base, float  power);
powl        long double  powl (long double base, long double power);
-----------------------------------------------------------------------
remainder   double       remainder  (double number1, double number2);
remainderf  float        remainderf (float  number1, float  number2);
remainderl  long double  remainderl (long double number1,
                                     long double number2);
-----------------------------------------------------------------------
remquo      double       remquo     (double number1, double quotient);
remquof     float        remquof    (float  number1, float  quotient);
remquol     long double  remquol    (long double number1,
                                     long double quotient);
-----------------------------------------------------------------------
rint        double       rint       (double       number);
rintf       float        rintf      (float        number);
rintl       long double  rintl      (long double  number);
-----------------------------------------------------------------------
round       double       round      (double      number);
roundf      float        roundf     (float       number);
roundl      long double  roundl     (long double number);
-----------------------------------------------------------------------
scalebn     double       scalebn    (double      number, int  factor);
scalebnf    float        scalebnf   (float       number, int  factor);
scalebnl    long double  scalebnl   (long double number, int  factor);
-----------------------------------------------------------------------
scalebln    double       scalebln   (double      number, long factor);
scaleblnf   float        scalebnlf  (float       number, long factor);
scalblnl    long double  scalebnll  (long double number, long factor);
-----------------------------------------------------------------------
```

continued

```
sin         double        sin       (double      number);
sinf        float         sinf      (float       number);
sinl        long double   sinl      (long double number);
sinh        double        sinh      (double      number);
sinhf       float         sinhf     (float       number);
sinhl       long double   sinhl     (long double number);
sqrt        double        sqrt      (double      number);
sqrtf       float         sqrtf     (float       number);
sqrtl       long double   sqrtl     (long double number);
tan         double        tan       (double      number);
tanf        float         tanf      (float       number);
tanl        long double   tanl      (long double number);
tanh        double        tanh      (double      number);
tanhf       float         tanhf     (float       number);
tanhl       long double   tanhl     (long double number);
trunc       double        trunc     (double      number);
truncf      float         truncf    (float       number);
truncl      long double   truncl    (long double number);
```

D.4 Standard I/O Library

We have divided the system input/output library (`stdio.h`) by the type of data being read or written.

General I/O

General input/output contains functions that apply to all files.

```
clearerr    void          clearerr (FILE* fp);
fclose      int           fclose   (FILE* fp);
feof        int           feof     (FILE* fp);
ferror      int           ferror   (FILE* fp);
```

continued

```
fopen      FILE*       fopen    (const char* extn_name,
                                  const char* file_mode);

freopen    FILE*       freopen  (const char* extn_name,
                                  const char* file_mode,
                                        FILE* stream);
```

Formatted I/O

Convert text data to/from internal memory formats.

```
fprintf    int         fprintf  (      FILE* fileOut,
                                  const char* format_string, ...);

printf     int         printf   (const char* format_string, ...);

sprintf    int         sprintf  (      char* to_loc,
                                  const char* format_string, ...);

snprintf   int         snprintf (      char* to_loc, size_t n,
                                  const char* format_string, ...);

fscanf     int         fscanf   (      FILE* fileIn,
                                  const char* format_string, ...);

scanf      int         scanf    (const char* format_string, ...);

sscanf     int         sscanf   (const char* from_loc,
                                  const char* format_string, ...);
```

Character I/O

Read and write one character at a time.

```
fgetc      int         fgetc    (FILE* fp);

fputc      int         fputc    (int   char_out, FILE* fp);

getc       int         getc     (FILE* fp);

getchar    int         getchar  (void);

putc       int         putc     (int   char_out, FILE* fp);

putchar    int         putchar  (int   char_out);

ungetc     int         ungetc   (int   char_out, FILE* fp);
```

File I/O

These functions work with binary files.

```
fread     size_t     fread  (void*  in_area, size_t size,
                             size_t count,    FILE* fp);
fseek     int        fseek  (FILE* fp, long offset, int from_loc);
ftell     long       ftell  (FILE* fp);
fwrite    size_t     fwrite (const void*  out_data, size_t  size,
                                 size_t count,    FILE*   fp);
rewind    void       rewind (FILE* fp);
```

String I/O

Read and write strings.

```
fgets     char*      fgets (      char* string, int   size, FILE* fp);
fputs     int        fputs (const char* string, FILE* fp);
gets      char*      gets  (      char* string);
puts      int        puts  (const char* string);
```

System File Control

System commands that create and delete files on the disk.

```
remove    int        remove  (const char* file_name);
rename    int        rename  (const char* old_name,
                              const char* new_name);
tmpfile   FILE*      tmpfile (void);
tmpnam    char*      tmpnam  (char* file_name);
```

D.5 Standard Library

The following functions are found in `stdlib.h`.

Math Functions

The following math functions are found in `stdlib`.

```
abs       int        abs   (int number);
div       div_t      div   (int numerator, int divisor);
labs      long       labs  (long number);
llabs     long long  llabs (long number);
ldiv      ldiv_t     ldiv  (long numerator, long divisor);
lldiv     lldiv_t    lldiv (long long numerator,
                            long long divisor);
rand      int        rand  (void);
srand     void       srand (unsigned seed);
```

Memory Functions

The following are memory allocation functions.

```
calloc    void*      calloc  (size_t num_elements,
                              size_t element_size);
free      void       free    (void*);
malloc    void*      malloc  (size_t num_bytes);
realloc   void*      realloc (void* stge_ptr, size_t element_size);
```

Program Control

The following functions control the program flow.

```
abort     void       abort  (void);
atexit    int        atexit (void (*) function_name (void));
exit      void       exit   (int exit_code);
_Exit     void       _Exit  (int exit_code);
```

System Communication

The following function communicates with the operating system.

```
system    int       system(const char* system_command);
```

Conversion Functions

The following functions convert data from one type to another.

```
atof      double       atof      (const char* string);
atoi      int          atoi      (const char* string);
atol      long         atol      (const char* string);
atoll     long long    atoll     (const char* string);
strtod    double       strtod    (const char* str, char** next_str);
strtof    float        strtof    (const char* str, char** next_str);
strtol    long         strtol    (const char* str, char** next_str,
                                  int   base);
strtold   long double  strtold   (const char* str, char** next_str);
strtoll   long long    strtoll   (const char* str, char** next_str,
                                  int   base);
strtoul   unsigned     strtoul   (const char* str, char** next_str,
          long                    int   base);
strtoull  unsigned     strtoull  (const char* str, char** next_str,
          long long               int   base);
```

D.6 String Library

The following functions are found in **`string.h`**.

Memory Functions

The following functions manipulate block data in memory.

```
memchr    void*      memchr  (const void*  mem, int a_char,
                                size_t bytes);
memcmp    int        memcmp  (const void* mem1, const void* mem2,
                                size_t bytes);
```

continued

```
memcpy   void*    memcpy  (void* to_mem, const void* fr_mem,
                           size_t bytes);

memmove  void*    memmove (void* to_mem, const void* fr_mem,
                           size_t bytes);
```

String Functions

The following functions manipulate strings.

```
strcat   char*    strcat  (char* to_str, const char* fr_str);

strchr   char*    strchr  (const char* str, int a_char);

strcmp   int      strcmp  (const char* str1, const char* str2);

strcpy   char*    strcpy   (char* to_str, const char* fr_str);

strlen   size_t   strlen  (const char* str);

strncat  char*    strncat (char*  to_str, const char* fr_str,
                           size_t bytes);

strncmp  int      strncmp (const  char*  str1,  const char* str2,
                           size_t bytes);

strncpy  char*    strncpy (char*  to_str, const char* fr_str,
                           size_t bytes);

strpbrk  char*    strpbrk (const char* str1, const char* str2);

strcspn  size_t   strcspn (const char* str1, const char* str2);

strrchr  char*    strrchr (const char* str, int a_char);

strspn   size_t   strspn  (const char* str1, const char* str2);

strstr   char*    strstr  (const char* str1, const char* str2);

strtok   char*    strtok  (      char* str1, const char* str2);
```

D.7 Time Library

The following functions are found in `time.h`.

```
clock     clock_t    clock     (void);

difftime  double     difftime  (time_t time_start,
                                time_t time_end);

mktime    time_t     mktime    (struct tm* cal_time);
```

continued

```
time      time_t      time      (time_t* num_time);
asctime   char*       asctime   (const struct tm* cal_time);
ctime     char*       ctime     (const time_t* num_time);
gmtime    struct tm*  gmtime    (const time_t* num_time);
localtime struct tm*  localtime (const time_t* num_time);
strftime  size_t      strftime  (      char*      str, size_t maxsize,
                                 const char*      format,
                                 const struct tm* timeptr);
```

Appendix E
Mathematical Series and Recursive Relations

To understand how the efficiency of an algorithm is computed, you need to have a basic understanding of how a mathematical series works. Because so much of algorithmic design involves recursion, the effect of recursion on a series must also be understood.

This appendix explores four basic concepts behind mathematical series and recursive relations: arithmetic series, geometric series, harmonic series, and recursive relations.

E.1 Arithmetic Series

In mathematics an arithmetic series is defined as shown in Figure E-1.

$$S_a = \sum_{i=1}^{n} i \,.\, x = x + 2x + \ldots + (n-1)x + nx$$

FIGURE E-1 Arithmetic Series

We present an informal solution to this series by writing it twice, first in its normal order and then in the reverse order of terms. To get equal terms, we add them as shown in Figure E-2.

$$
\begin{aligned}
S_a &= x + 2x + \ldots + (n-1)x + nx \\
S_a &= nx + (n-1)x + \ldots + 2x + x \\
\hline
2S_a &= \underbrace{(n+1)x + (n+1)x + \ldots + (n+1)x + (n+1)x}_{n \text{ terms}}
\end{aligned}
$$

Result: $S_a = \dfrac{n(n+1)x}{2}$

FIGURE E-2 Arithmetic Series Solution

Example Find the sum of the following series:

```
1 + 2 + 3 + 4 + ... + 100
```

This is the arithmetic series with `x = 1` and `n = 100`. Although we could add all of the numbers, it is much easier to apply the series formula.

```
Sa = 100(101) / 2 = 5050
```

Example In this example we use a series where `x` is not `1`.

```
5 + 10 + 15 + 20 + ... + 50
```

In this series, we see that each number increases by 5, which means `x = 5`. We use the general formula

```
Sa = xn (n + 1) / 2 = 5 (10)(11) / 2 = 275
```

Example For our final example, we find the sum of a series where `x` is not `1` and `n` does not start with `1`. Find the following sum:

```
100 + 105 + 110 + 115 + ... + 200
```

To be able to use our formula, we need to write this series as the combination of two series.

```
(5 + 10 + 15 + ... + 200) - (5 + 10 + 15 + ... + 95)
```

The first series (from 5 to 200) is an arithmetic series with `x = 5` and `n = 40`. The second series is an arithmetic series with `x = 5` and `n = 19`. Using our formula, we see that

```
Sa = ( 5 (40)(41) / 2) - 5(19)(20) / 2 = 3150
```

E.2 Geometric Series

In mathematics a geometric series is defined as shown in Figure E-3.

$$S_g = \sum_{i=0}^{n} x^i = 1 + x + x^2 + \ldots + x^{(n-1)} + x^n$$

FIGURE E-3 Geometric Series

Again, we present an informal solution to this series as shown in Figure E-4.

Given

$$S_g = 1 + x + x^2 + \ldots + x^{(n-1)} + x^n$$

We multiply both sides by x

$$x \cdot S_g = x + x^2 + x^3 + \ldots + x^n + x^{(n+1)}$$

We add and subtract 1 from right side

$$x \cdot S_g = \underbrace{1 + x + x^2 + x^3 + \ldots + x^n}_{S_g} + x^{(n+1)} - 1$$

After substituting part of the right side with S_g

$$x \cdot S_g = S_g + x^{(n+1)} - 1$$

After reordering and factoring

$$S_g = \frac{x^{(n+1)} - 1}{x - 1}$$

FIGURE E-4 Geometric Series Solution

Example Find the sum of the following series:

```
1 + 2 + 4 + 8 + ... + 1024
```

We recognize that this is a geometric series in which x is 2 (the series is powers of 2) and n is 10 (2^{10} is 1024). Using the geometric formula, we get

$$S_g = (x^{n+1} - 1) / (x - 1) = (2^{11} - 1) / (2 - 1) = 2047$$

Example Find the sum of the following series:

$$1 + 1/2 + 1/4 + 1/8 + \ldots$$

This is a case of the geometric formula in which x is 1/2. One property of the geometric series is that it converges if x is less than 1 and n approaches infinity. In this case x^{n+1} becomes 0 because a number less than 1 to the power infinity is 0. This is shown in this example in which x is less than 1 and n is infinity.

$$\begin{aligned} S_g &= (x^{n+1} - 1) / (x - 1) \\ &= (-1) / (x - 1) \\ &= 1 / (1 - x) \\ &= 1 / (1 - 1/2) \\ &= 2 \end{aligned}$$

E.3 Harmonic Series

Figure E-5 shows a harmonic series. This series does not converge, which means that the value of H_n does not get close to 0 when n gets close to infinity. Although the exact calculation of this series is beyond the scope of this text, we provide the approximation shown in the figure.

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Result: $H_n = \ln n$

FIGURE E-5 Harmonic Series

Example Find the sum of the following series.

$$1 + 1/2 + 1/3 + \ldots + 1/100$$

The answer is ln 100 or 4.60.

E.4 Recursive Relations

To find the complexity of recursive algorithms, we need to solve a recursive relation. A recursive relation is one in which the value of a function is defined in terms of itself. For example,

```
f(n) = f(n - 1) + 1
```

is a recursive relation because the value of `f(n)` is defined in terms of `f(n - 1)`. Although there is not a general formula for solving these types of relations, the general approach requires that we develop the relations for `n`, `n - 1`, `n - 2`, `n - 3`, and so on. In this process we try to eliminate terms that involve `f(i)` so that on the right side we have only `f(n)` and on the left side only constants. Note that for a relation to be solvable, some initial values of `f` must be provided. The most common given is the value of `f(0)` or `f(1)`. We develop the concept heuristically through examples.

Example Figure E-6 shows the recursive solution for a recursive relationship.

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1) + 1 & \text{Otherwise} \end{cases}$$

We substitute n with n – 1 repeatedly

$$\begin{aligned} f(n) &= f(n-1) + 1 \\ f(n-1) &= f(n-2) + 1 \\ f(n-2) &= f(n-3) + 1 \\ &\dots \\ f(1) &= f(0) + 1 \end{aligned}$$

When we add the equations, some terms are canceled

$$f(n) = f(0) + \underbrace{1 + 1 + \dots + 1}_{n}$$

The base case, f(0) = 0, so we have

$$f(n) = n$$

FIGURE E-6 Solution for First Recursive Relation

Example Figure E-7 shows another recursive relation and its solution.

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1) + n & \text{Otherwise} \end{cases}$$

We substitute n with n – 1 repeatedly

$$\begin{aligned} f(n) &= f(n-1) + n \\ f(n-1) &= f(n-2) + n - 1 \\ f(n-2) &= f(n-3) + n - 2 \\ &\dots \\ f(1) &= f(0) + 1 \end{aligned}$$

When we add the equations, some terms are canceled

$$f(n) = \underbrace{f(0)}_{0} + \underbrace{1 + 2 + \dots + n}_{\text{Arithmetic series}}$$

Result: $$f(n) = \frac{n(n-1)}{2}$$

FIGURE E-7 Solution for Second Recursive Relation

Example Figure E-8 shows a third recursive relation.

$$f(n) = \begin{cases} 0 & \text{if } n = 1 \\ f(n/2) + 1 & \text{Otherwise} \end{cases}$$

Where n is 2^k

We substitute n with 2^k and repeatedly replace k by k –1

$$\begin{aligned} f(2^k) &= f(2^{k-1}) + 1 \\ f(2^{k-1}) &= f(2^{k-2}) + 1 \\ &\dots \\ f(2^1) &= f(2^0) + 1 \end{aligned}$$

If we add right sides and left sides, some terms are canceled

$$f(2^k) = \underbrace{f(1)}_{0} + \underbrace{1 + 1 + \dots + 1}_{k \text{ times}}$$

$$f(n) = k = \log_2 n$$

The value of k is $\log_2 n$

$$f(n) = \log_2 n$$

FIGURE E-8 Recursive Solution for an n log n Series

Appendix F
Array Implementations of Stacks and Queues

In Chapter 3 we developed a stack ADT using dynamic memory and a linked list implementation. In Chapter 4 we similarly developed a queue ADT. There are times, however, when we either desire to or need to use an array implementation. We discuss stack and queue array implementations in this appendix.

F.1 Stack ADT

If a stack's maximum size can be calculated before the program is written, an array implementation of a stack is more efficient than the dynamic implementation using a linked list. In addition, an array stack is a more easily understood and natural picture of a stack.

When implementing a stack in an array, the base is found at the first stack element, index 0 in C. The top then moves up and down the array as data are inserted and deleted. To push an element into the stack, we add 1 to top and use it as the array index for the new data. To pop an element from the stack, we copy the data at index location top and then subtract 1 from top. One additional metadata element is required: the maximum number of elements in the stack. The structure for the array implementation with a maximum size of 5 elements is shown in Figure F-1.

In this section we redesign the eight basic stack algorithms, using an array rather than a linked list implementation.

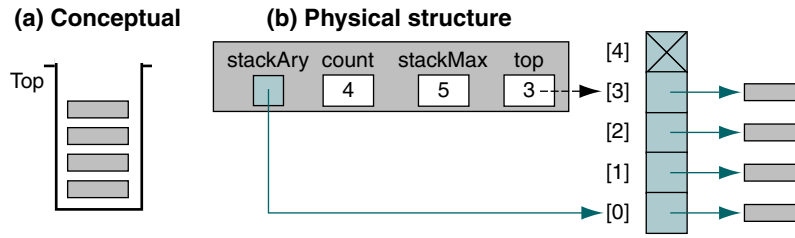


FIGURE F-1 Stack Array Implementation

Array Data Structure

There are three differences in the data structure for an array implementation: first, the stack top is an index rather than a pointer; second, we need to store the maximum number of elements allowed in the stack; and, third, we don't need next fields. Because the stack is a LIFO structure, each element has a physical adjacency to its predecessor.

Although the data structure is significantly different, the array implementation of a stack requires the same basic algorithms used in the linked list implementation. The ADT declaration is shown in Program F-1. The algorithms are developed in the sections that follow.

PROGRAM F-1 Stack Array Definition

```
1  // Stack Definitions for Array Implementation
2     typedef struct
3       {
4        void**  stackAry;
5        int     count;
6        int     stackMax;
7        int     top;
8       } STACK;
9
10 // Prototype Declarations
11    STACK* createStack  (int maxSize);
12    STACK* destroyStack (STACK* queue);
13
14    bool  pushStack  (STACK* stack, void* itemPtr);
15    void* popStack   (STACK* stack);
16    void* stackTop   (STACK* stack);
17    int   stackCount (STACK* stack);
18    bool  emptyStack (STACK* stack);
19    bool  fullStack  (STACK* stack);
20 // End of Stack ADT Definitions
```

Create Stack

The ATD implementation of create stack is shown in Program F-2. The application program specifies the stack maximum size when the stack is created.

PROGRAM F-2 Create Stack

```
 1  /* ================== createStack =================
 2     This algorithm creates an empty stack by allocating
 3     the head structure and the array from dynamic memory.
 4        Pre    maxSize is max number of elements
 5        Post   returns pointer to stack head structure
 6             -or- NULL if overflow
 7  */
 8  STACK* createStack (int maxSize)
 9  {
10  // Local Definitions
11     STACK* stack;
12
13  // Statements
14     stack = (STACK*) malloc( sizeof (STACK));
15     if (!stack)
16        return NULL;
17
18     // Head allocated. Initialize & allocate stack.
19     stack->count    =  0;
20     stack->top      = -1;
21     stack->stackMax = maxSize;
22     stack->stackAry =  (void**)calloc(stack->stackMax,
23                                sizeof(void*));
24     if (!stack->stackAry)
25        {
26         free (stack);
27         return NULL;
28        } // if
29     return stack;
30  }  // createStack
```

Program F-2 Analysis

There are two possible errors in the create stack function; both involve memory allocation. In statement 14 we allocate memory for the stack head structure. If this allocation fails, we return a null pointer. Then, in statement 22, we use *calloc* to allocate the array of *void* pointers and store its address in the stack head structure. Again, if this allocation fails, we return a null pointer. Note that the stack array in the head structure is a pointer to an array of *void* pointers. This means that when we cast the pointer type, it must be cast as a pointer to a pointer to *void*.

Because the newly created stack is by definition empty, we set the stack top index to −1. We must use −1 as the value for a null stack because a stack with only one item in it has a stack top of zero.

Push Stack

The logic for the array implementation of push stack is very simple. We first test for overflow by comparing the count of the elements in the stack with the maximum stack size. If the stack is full, we return false, indicating that the push failed. If there is room, we copy the data to the stack, increase the top index and the stack count, and return true for success. The function code is shown in Program F-3.

PROGRAM F-3 Push Stack

```
/* ================== pushStack =================
   This function pushes an item onto the stack.
      Pre  stack is pointer to stack head structure
           dataInPtr is pointer to be inserted
      Post returns true if success; false if overflow
*/
bool pushStack (STACK* stack, void* dataInPtr)
{
// Statements
   if (stack->count == stack->stackMax)
      return false;

   (stack->count)++;
   (stack->top)++;
   stack->stackAry[stack->top] = dataInPtr;

   return true;
}  // pushStack
```

Pop Stack

The array implementation of pop stack is also quite simple and parallels the linked list implementation. The code is shown in Program F-4.

PROGRAM F-4 Pop Stack

```
/* ================== popStack ==================
   This function pops the item on the top of the stack.
      Pre  stack is pointer to stack head structure
      Post returns pointer to user data if successful
                   NULL if underflow
*/
void* popStack (STACK* stack)
{
// Local Declarations
   void* dataPtrOut;

```

continued

PROGRAM F-4 Pop Stack *(continued)*

```
12  // Statements
13     if (stack->count == 0)
14         dataPtrOut = NULL;
15     else
16        {
17         dataPtrOut = stack->stackAry[stack->top];
18         (stack->count)--;
19         (stack->top)--;
20        } // else
21
22     return dataPtrOut;
23  }  // popStack
```

Program F-4 Analysis

When working with data structures, you should always test what happens when you delete the only item in the structure. For our array stack, we need to ensure that when the only item in the stack is deleted, the stack is properly set to a null status. We have defined a null stack as a stack with a top index of −1. If there is only one element in the stack, the top is 0. Subtracting 1 from 0 gives us −1, our designated flag for a null stack.

Stack Top

Stack top is a simple function. We test for data in the stack and return the data pointer in the top element if the stack is not empty or a null pointer if it is empty. The code is shown in Program F-5.

PROGRAM F-5 Stack Top

```
1   /* ==================== stackTop ====================
2      This function retrieves the data from the top of the
3      stack without changing the stack.
4         Pre  stack is pointer to the stack
5         Post returns data pointer if successful
6                 -or- null pointer if stack empty
7   */
8   void* stackTop (STACK* stack)
9   {
10  // Statements
11     if (stack->count == 0)
12        return NULL;
13     else
14        return stack->stackAry[stack->top];
15  }  // stackTop
```

Empty Stack

Empty stack logic is the same regardless of the implementation structure. It simply tests the stack count in the head structure and returns true if it is 0 or false if it is not. The code is shown in Program F-6.

PROGRAM F-6 Empty Stack

```
/* ================= emptyStack ================
   This function determines if a stack is empty.
      Pre  stack is a pointer to the stack
      Post returns true if empty; false if data in stack
*/
bool emptyStack (STACK* stack)
{
// Statements
   return (stack->count == 0);
}  // emptyStack
```

Full Stack

To determine whether the stack is full, we simply compare the count with the maximum number of elements allocated for the array. If they are equal, the stack is full; if not, the stack has room for more data. The code is shown in Program F-7.

PROGRAM F-7 Full Stack

```
/* ================= fullStack ================
   This function determines if a stack is full.
      Pre  stack is a pointer to a stack head structure
      Post returns true if full; false if empty elements
*/
bool fullStack (STACK* stack)
{
// Statements
   return (stack->top == stack->stackMax);
}  // fullStack
```

Stack Count

Like empty stack, the logic for stack count is identical in all implementations. We simply return the stack count found in the stack head structure. The code is shown in Program F-8.

PROGRAM F-8 Stack Count

```
/* ================== stackCount =================
   Returns number of elements in stack.
      Pre  stack is a pointer to the stack
      Post count returned
*/
int stackCount(STACK* stack)
{
// Statements
   return stack->count;
}  // stackCount
```

Destroy Stack

To destroy a stack implemented using an array, we must first free the array structure and then the stack head structure. We then return a null pointer, which the calling function should store in the application stack pointer. The code is shown in Program F-9.

PROGRAM F-9 Destroy Stack

```
/* ================= destroyStack ================
   This function releases all memory to the heap.
   Pre  stack is pointer to stack head structure
   Post returns null pointer
*/
STACK* destroyStack (STACK* stack)
{
// Statements
   if (stack)
      {
       // Release data memory
       for (int i = 0; i < stack->count; i++)
           free (stack->stackAry[i]);

       // Release stack array
       free (stack->stackAry);

       // Now release memory for stack head
       free (stack);
      } // if stack
   return NULL;
}  // destroyStack
```

Program F-9 Analysis Our only concern is that the stack pointer may be null. If it is, we do nothing except return a null pointer. If the stack pointer is not null, we free the array memory first, then the head structure memory, and return a null pointer.

F.2 Queue ADT

Although the linked list implementation is very popular, it is not the only way to implement a queue. Queues can also be implemented in arrays and, for very large queues, in files. Because a file can be thought of as an array stored on a disk, the array and file implementations have similar solutions. In this section we rewrite the queue ADT using an array.

An enqueue to an empty queue is placed in the first element of the array, which becomes both the front and the rear. Subsequent enqueues are placed at the array location following rear; that is, each enqueue stores the queue data in the next element after the current queue rear. Thus, if the last element in the queue is stored at array location 11, the data for the next enqueue is placed in element 12.

Dequeues take place at the front of the queue. As an element is deleted from the queue, the queue front is advanced to the next location; that is, queue front becomes queue front plus one. Figure F-2(a) shows a conceptual queue; Figure F-2(b) shows a physical queue after it has been in operation for a period of time. At the point shown, the data have migrated from the front of the array to its center.

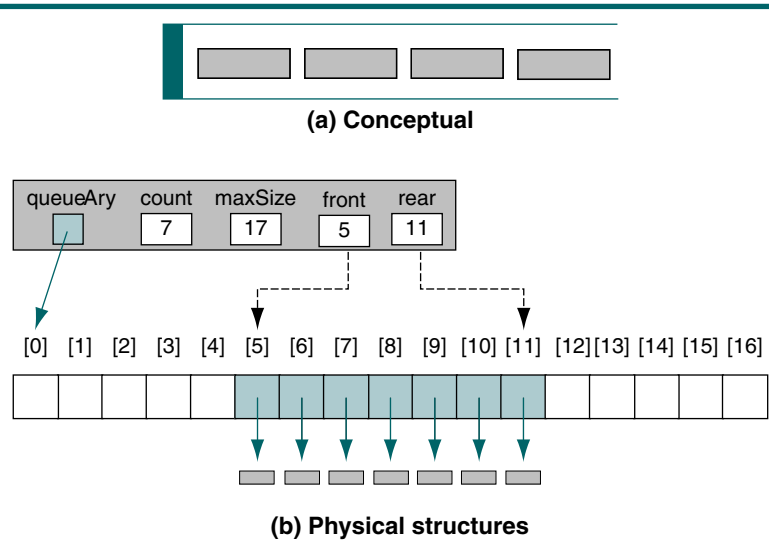


FIGURE F-2 Queue Stored in an Array

When we implement a queue in an array, we use indexes rather than pointers. Thus, the front of the queue in Figure F-2 is 5 and the rear is 11.

The definition of a full queue changes when we implement a queue in an array. A full queue is defined as every element filled. Because arrays have a

finite number of elements, we can determine that the queue is full by testing the queue count against the maximum number of elements.[1]

When data arrive faster than the queue service time, the queue tends to advance to the end of the array. This leads to the situation where the last element in the array is occupied but the queue is not full because there are empty elements at the beginning of the array. This situation is shown in Figure F-3.

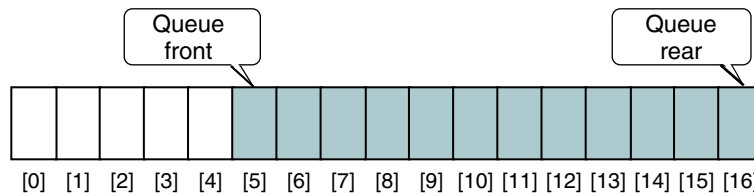


FIGURE F-3 Array Queue with Last Element Filled

When the data are grouped at the end of the array, we need to find a place for the new element when we enqueue data. One solution is to shift all of the elements from the end to the beginning of the array. For example, in Figure F-3 element 5 is shifted to element 0, element 6 to 1, 7 to 2, and so forth until element 16 is shifted to 11.

A more efficient alternative is to use a circular array. In a circular array, the last element is logically followed by the first element. This is done by testing for the last element and, rather than adding one, setting the index to zero. Given our array queue above, the next element after 16 is 0. A circular queue is shown in Figure F-4.

With this understanding of a circular queue, we are ready to rewrite the abstract data type. The only difference in the calling sequence between the two implementations is the addition of a maximum queue size in the create queue function.

Array Queues Implementation

Two changes are needed to the data structure for the queue head. We need to store the address of the queue array, which we allocate from dynamic memory. We also need to store the maximum number of elements in the array. The revised data structure is shown in Program F-10. We also include the prototype declarations for the queue functions. As you study the data structure, compare it with the linked list structure in Program 4-1, "Queue ADT Data Structures."

1. Even though we are implementing the queue in C, we ignore the possibility of dynamically changing the array size with *realloc*.

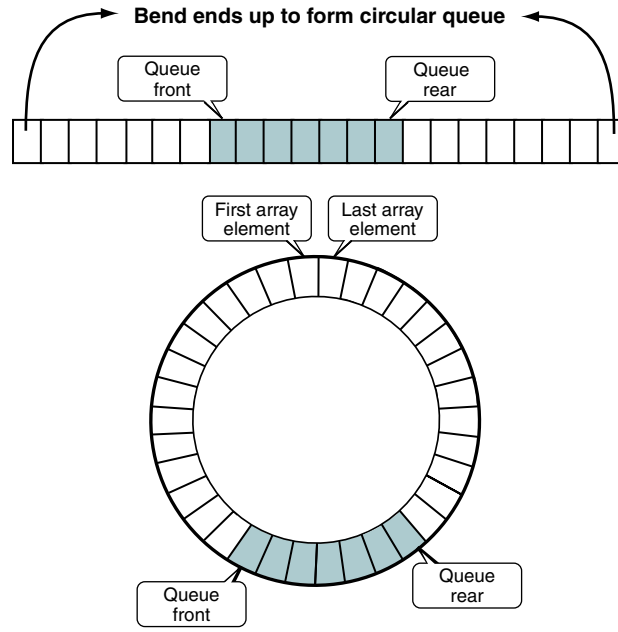


FIGURE F-4 Circular Queue

PROGRAM F-10 Queue Array Definition

```
1  // Queue ADT Defintions
2     typedef struct
3        {
4         void**   queueAry;
5         int      maxSize;
6         int      count;
7         int      front;
8         int      rear;
9        } QUEUE;
10
11 // Prototype Declarations
12    QUEUE* createQueue  (int maxSize);
13    QUEUE* destroyQueue (QUEUE* queue);
14
15    bool  enqueue    (QUEUE* queue, void* itemPtr);
16    void* dequeue    (QUEUE* queue);
17    void* queueFront (QUEUE* queue);
18    void* queueRear  (QUEUE* queue);
19    int   queueCount (QUEUE* queue);
20    bool  emptyQueue (QUEUE* queue);
21    bool  fullQueue  (QUEUE* queue);
22 // End of Queue ADT Definitions
```

Create Queue

The array implementation of create queue allocates memory for a queue head structure and the array itself in dynamic memory and returns the address of the head structure to the caller. If there is not enough memory for the queue structure, it returns a null pointer. The code for create queue is shown in Program F-11.

PROGRAM F-11 Create Queue

```
 1  /* =============== createQueue ================
 2     Allocates memory for a queue head node from dynamic
 3     memory and returns its address to the caller.
 4        Pre    nothing
 5        Post   head has been allocated and initialized
 6        Return head's address if successful;
 7               null if overflow
 8  */
 9  QUEUE* createQueue (int maxSize)
10  {
11  // Local Definitions
12     QUEUE* queue;
13
14  // Statements
15     queue = (QUEUE*) malloc (sizeof (QUEUE));
16     if (queue)
17        {
18         // head structure created. Now allocate queue
19         queue->queueAry =
20              (void**)calloc(maxSize, sizeof(void*));
21         queue->count   =  0;
22         queue->front   = -1;
23         queue->rear    = -1;
24         queue->maxSize = maxSize;
25        } // if
26
27     return queue;
28  }  // createQueue
```

Program F-11 Analysis

The most difficult code in create queue is found in statement 19. We must cast the array pointer as a pointer to a *void* pointer. This is because we cannot store the data directly in the ADT. The calling function must pass a pointer to data in dynamic memory. We can then store it in the queue array as a *void* pointer. Because the array is an array of *void* pointers, its name is a pointer constant to a *void* pointer. This means that the pointer returned by *calloc* must be a pointer to a *void* pointer.

A second point worth noting is that we set the front and rear indexes to −1. We have chosen −1 as the index for a null queue to make the enqueue logic as simple as possible. Whenever we insert an element into the queue, we add one to the rear index. When the queue is null, this automatically sets rear to zero when the queue is null. The initial value for the front queue could be anything, but we make it −1 for consistency.

Enqueue

The parameter list for the array implementation of enqueue is identical to the linked list implementation. The logic, however, differs considerably. First, there is no memory to be allocated. We simply need to determine in which array element to store the data pointer.

The test for a full queue has also been changed. To determine if the queue is full, we compare the queue count to the maximum queue size. If they are equal, we have a full queue.

To complete the insert, we add one to the rear index. If rear is at the last element, we need to wrap around to the first element of the queue. The code is shown in Program F-12.

PROGRAM F-12 Enqueue

```
1  /* ================ enqueue ================
2     This algorithm inserts data into a queue.
3        Pre    queue has been created
4        Post   data have been inserted
5        Return true if successful; false if overflow
6  */
7  bool enqueue (QUEUE* queue, void* itemPtr)
8  {
9  // Statements
10    if (queue->count == queue->maxSize)
11       return false;
12
13    (queue->rear)++;
14    if (queue->rear  == queue->maxSize)
15        // Queue wraps to element 0
16        queue->rear  =  0;
17    queue->queueAry[queue->rear] = itemPtr;
18
19    if (queue->count == 0)
20       {
21        // Inserting into null queue
22        queue->front  = 0;
23        queue->count  = 1;
24       } // if
25    else
26       (queue->count)++;
27    return true;
28 }  // enqueue
```

Program F-12 Analysis

You need to study two points in this function carefully. First, note how the wraparound logic in statements 14 to 16 is handled. At this point in the function, we know that the queue is not full. If the rear index is equal to the maximum queue size, however, we need to wrap around to the first element. (Remember, C uses zero indexing, so when the new rear is equal to the count, the last element has been filled.)

Second, if we are inserting into a null queue, identified by a count of zero, we must set the front index to 0. The rear index is set to 0 automatically when we add one to the null rear pointer, which is set to −1 whenever the queue is empty.

Dequeue

Dequeue deletes data at the front of the queue and returns it to the calling function. Its code is shown in Program F-13.

PROGRAM F-13 Dequeue

```
1  /* ================== dequeue ==================
2     This algorithm deletes a node from the linked list.
3        Pre  queue has been created
4        Post returns pointer to user data if successful
5                     NULL if underflow
6  */
7  void* dequeue (QUEUE* queue)
8  {
9  // Local Declarations
10    void* dataPtrOut;
11
12 // Statements
13    if (!queue->count)
14       return NULL;
15
16    dataPtrOut = queue->queueAry[queue->front];
17    (queue->front)++;
18    if (queue->front == queue->maxSize)
19       // queue front has wrapped to element 0
20       queue->front =  0;
21    if (queue->count == 1)
22       // Deleted only item in queue
23       queue->rear = queue->front = -1;
24    (queue->count)--;
25
26    return dataPtrOut;
27 }  // dequeue
```

Program F-13 Analysis

Take note of two pieces of code in this function. First, once again we must wrap around from the end of the queue. For dequeue we test for a wrap after we have deleted the data from the queue. If front was in the last element of the array, we must set the new front to the first element, 0.

The second point is that we need to reset both front and rear to −1 when we delete the last item in the queue. This is done in statements 21 to 23.

Queue Front

The queue front logic parallels dequeue except that the status of the queue is not changed. Its logic is shown in Program F-14.

PROGRAM F-14 Queue Front

```
/* ================== queueFront =================
   Retrieve the data at the front of the queue
   without changing the queue contents.
      Pre  queue is a pointer to initialized queue
      Post returns pointer to user data if successful;
                   NULL if underflow
*/
void* queueFront (QUEUE* queue)
{
// Statements
   if (!queue->count)
       return NULL;
   return queue->queueAry[queue->front];
}  // queueFront
```

Queue Rear

Queue rear returns a pointer to the data at the rear of the queue without changing the contents of the queue. It's logic is shown in Program F-15.

PROGRAM F-15 Queue Rear

```
/* ================== queueRear =================
   Retrieves the data at the rear of the queue
   without changing the queue contents.
      Pre  queue is pointer to initialized queue
      Post returns pointer to user data if successful;
                   NULL if underflow
*/
void* queueRear (QUEUE* queue)
{
// Statements
   if (!queue->count)
       return NULL;
   return queue->queueAry[queue->rear];
}  // queueRear
```

Full Queue

The full queue logic is much simpler in the array implementation than in the linked list. We simply test the count to the maximum queue size. If they are equal, the queue if full. The code is shown in Program F-16.

Empty Queue

The empty queue logic is much simpler in the array implementation than in the linked list. We simply test the count to the maximum queue size. If they are equal, the queue is full. The code is shown in Program F-17.

PROGRAM F-16 Full Queue

```
/* ================== fullQueue =================
   Checks to see if a queue is full. The queue is full
   if the array is full.
      Pre    queue is pointer to a queue head node
      Return true if full; false if room for more
*/
bool fullQueue (QUEUE* queue)
{
// Statements
   return (queue->count == queue->maxSize);
}  // fullQueue
```

PROGRAM F-17 Empty Queue

```
/* ==================== emptyQueue ==================
   Checks to see if a queue is empty.
      Pre    queue is pointer to a queue head node
      Return true if empty; false if data in queue
*/
bool emptyQueue (QUEUE* queue)
{
// Statements
   return (queue->count == 0);
}  // emptyQueue
```

Queue Count

The queue count returns the number of elements in the queue. The code is shown in Program F-18.

PROGRAM F-18 Queue Count

```
/* ==================== queueCount ==================
   Returns the number of elements in the queue.
      Pre    queue is pointer to a queue head node
      Return queue count
*/
int queueCount (QUEUE* queue)
{
// Statements
   return queue->count;
}  // emptyQueue
```

Destroy Queue

The last function for the array implementation is destroy queue. The queue count and empty queue functions are identical for the linked list implementation and the array implementation, so we won't repeat them.

To destroy the queue, we simply free the array and head structures and return a null pointer. The code is shown in Program F-19.

PROGRAM F-19 Destroy Queue

```
/* ================= destroyQueue ================
   This function deletes the queue and array memory.
      Pre    queue is a valid queue
      Post   all data have been deleted and recycled
      Return null pointer
*/
QUEUE* destroyQueue (QUEUE* queue)
{
// Local Definitions
   int locn = queue->front;

// Statements
   if (queue)
      {
       // Free data memory
       while (queue->count != 0)
          {
           free (queue->queueAry[locn++]);
           queue->count--;
           if (locn == queue->maxSize)
               locn = 0;
          } // while
       free (queue->queueAry);
       free (queue);
      } // if
   return NULL;
}  // destroyQueue
```

Glossary

2-3 tree: a B-tree of order 3.

2-3-4 tree: a B-tree of order 4.

A

abstract data type (ADT): a data declaration packaged together with the operations that are meaningful on the data type.

adjacency list: a method of representing a graph that uses a linked list to store the vertices and a two-dimensional, linked list array to store the lines.

adjacency matrix: a method of representing a graph that uses a vector for the vertices and a matrix (square two-dimensional array) to store the lines.

adjacent vertices: two vertices in a graph that are connected by a line.

ADT: *see* abstract data type.

algorithm: the logical steps necessary to solve a problem; a module or a part of a module.

algorithmics: the term created by Brassard and Bratley that refers to the study of techniques used to create efficient algorithms.

ancestor: any node in the path from the current node to the root of a tree.

arc: a directed line in a graph; contrast *edge*.

array: a fixed-size, sequenced collection of elements of the same data type.

arrival rate: the rate at which customers arrive in the queue.

ascending sequence: a list order in which each element in the list has a key greater than or equal to those of its predecessors.

ascending sort: a sort that orders a list in ascending sequence.

ASCII: the American Standard Code for Information Interchange. An encoding scheme that defines control characters and graphic characters for the first 128 values in a byte.

atomic data: data that cannot be meaningfully subdivided.

AVL tree: a height-balanced binary search tree that uses a balance factor to control the balance.

AVL tree head structure: the node in a AVL tree implementation that contains the tree metadata, such as the root pointer, the compare function pointer, and a node count.

AVL tree node structure: the node in a AVL tree implementation that contains the tree application data and structural metadata, such as the left and right branches, and the balance factor for the node.

B

backtracking: an algorithmic process, usually implemented with a stack or through recursion, that remembers the path through a data structure and can retrace the path in reverse order.

balance: a tree node attribute representing the difference in height between the node's subtrees.

balance factor: in an AVL tree, metadata used to maintain the tree's balance.

balanced binary tree: a binary tree in which the height of the subtrees differs by no more than one and the subtrees are balanced.

balanced merge: a merge that uses a constant number of input merge files and the same number of output merge files.

base case: in a recursive function, the statement that "solves" the problem.

big-O notation: a measure of the efficiency of an algorithm in which only the dominant factor is considered.

bill of materials: a tree representation of a product showing which components are assembled into which parts. A structure chart is an example of a bill of materials.

binary search: a search algorithm in which a target is located by repeatedly dividing the list in half.

binary search tree (BST): a binary tree in which: (1) the keys of the left subtree are all less than the root key, (2) the keys of the right subtree are greater than or equal to the root key, and (3) the subtrees are binary search trees.

binary tree: a tree in which no node can have more than two children; a tree with a maximum outdegree of 2.

binary tree search: a binary tree with the following properties: (1) All items in the left subtree are less than the root. (2) All items in the right subtree are greater than or equal to the root. And (3) each subtree is itself a binary search tree.

binary tree traversal: a binary tree processing sequence that processes each node of the tree once and only once in a predetermined sequence.

bool: the C keyword for the Boolean type.

Boolean: a type whose values can be only true or false.

branch: a line in a tree that connects two adjacent nodes.

breadth-first traversal: a graph traversal in which nodes adjacent to the current (siblings) node are processed before their descendents.

BST: *see* binary search tree.

B*tree: a B-tree in which each node is at least two-thirds full.

B+tree: a B-tree in which all data are represented at the leaf level, and each leaf node has a pointer to the next sequential leaf node in the tree.

B-tree: an *m*-way tree in which all nodes except the root have a minimum number of entries, and except for leaves each node entry has a nonnull subtree.

B-tree order: the maximum number of subtrees from a B-tree node.

bubble sort: a sort algorithm in which each pass through the data moves (bubbles) the lowest element to the beginning of the unsorted portion of the list.

bucket: in a hashing algorithm, a node that can accommodate multiple data occurrences.

bucket hashing: a hashing technique in which keys are hashed to nodes that accommodate multiple data occurrences.

C

child: a node in a tree or a graph that has a predecessor.

chronological list: a list that is organized by time—that is, the data are stored in the order in which they were received; *see also* first-in–first out (FIFO) *and* last in–last out (LIFO).

circularly linked list: a linked list in which the last node's link points to the first node in the list.

clustering: the tendency of data to build up unevenly across a hashed list.

collision: an event that occurs when a hashing algorithm produces an address for an insertion and that address is already occupied.

collision resolution: an algorithmic processing that determines an alternative address after a collision.

combine: a B-tree underflow balancing technique that joins the data from an underflowed entry, a minimal sibling, and a parent in one node.

complete tree: a tree with a restricted outdegree that has the maximum number of nodes for its height.

complete binary tree: a complete tree with a outdegree of 2.

composite data: data that are built on other data structures; that is, data that can be broken down into discrete atomic elements.

connected graph: a graph is connected if, when direction is suppressed, there is a path from any vertex to any other vertex.

construct: a basic programming design statement, such as, sequence, selection, or loop.

customer: in a queue, a person or thing needing service.

cycle: a graph path whose length is greater than 1 and that starts and ends at the same vertex.

D

data: in a linked list, the part of the structure that stores application information.

data compression: any data storage method that reduces the number of bits required to store data. See Huffman code.

data hiding: the principle of structured programming in which data are available to a function only if it needs them to complete its processing; data that are not needed are "hidden" from view; *see also* encapsulation.

data name: an identifier given to data in a program.

data node: in a data structure, a node that contains application data or a pointer to application data; contrast *header node*.

data structure: an aggregation of atomic and composite data types into a set with defined relationships.

data type: a named set of values and operations defined to manipulate the values, such as character and integer.

data validation: the process of verifying and validating data read from an external source.

degree: the number of lines incident to a node in a graph.

deletion: in a list, any process that removes data.

dependent quadratic loop: any loop in which the number of iterations of the inner loop depends on the outer loop.

depth (of tree): *see* height.

depth-first traversal: a traversal in which all of a node's descendents are processed before any adjacent nodes (siblings).

dequeue: delete an element from a queue.

descendent: any node in the path from the current node to a leaf.

descending sequence: a list order in which each element in a list has a key less than or equal to that of its predecessor.

descending sort: a sort that orders a list in descending sequence.

digit extraction: a hashing method in which selected digits are extracted from the key and used as the address.

digraph: a directed graph.

direct hashing: a hashing method in which the key is used without modification.

directed graph: a graph in which direction is indicated on the lines (arcs).

disjoint graph: a graph that is not connected.

distribution phase: in a natural merge sort, the pass through the data that redistributes the merge runs to the input files for remerging.

division remainder: a hashing method that divides the key by the array size and uses the remainder for the address.

double hashing: a hashing collision resolution method in which the collision address is hashed to determine the next address.

doubly linked list: a linked list structure in which each node has a pointer to both its successor and its predecessor; contrast *singly linked list*.

dynamic memory: memory whose use can change during the execution of the program (often referred to as *heap*).

E

edge: a line in an undirected graph.

empty list: a list that has been defined but that contains no data. Also known as a null list.

encapsulation: the design concept in which data, functions, and objects, such as text files or linear lists, are maintained separately from the application using them; *see also* data hiding.

enqueue: insert an element into a queue.

exchange sort: the sort algorithm that exchanges elements that are out of order until the entire list is sorted.

exponential efficiency: a category of program/module efficiency in which the run time is a function of the power of the number of elements being processed, as in $O(n) = c^n$.

expression tree: a binary tree representation of an expression.

external sort: a sort that uses primary memory for the data currently being sorted and secondary storage for any data that does not fit in primary memory.

F

factorial: an arithmetic function whose value is the product of the integral values from 1 to the number.

factorial efficiency: a measure of the efficiency of a module in which the run time is proportional to the number of elements factorial, as in $O(n) = n!$.

Fibonacci numbers: a number series discovered by Leonardo Fibonacci in which each number is the sum of the previous two numbers.

FIFO: *see* first in–first out.

FIFO insertion: an insertion method that places data in a list such that a subsequent traversal processes data in first in–first out sequence.

first-in–first-out (FIFO): a data structure processing sequence in which data are processed in the order that they are retrieved. A queue.

fold boundary: a hashing algorithm in which the left and right folds are reversed before they are added to determine the key.

fold shift: a hashing algorithm in which the key is divided into parts that are added to determine the key.

front: in a list, a reference that identifies the first element. In a queue, the next element to be removed.

G

general case: in recursion, the case that reduces the problem.

general list: a list in which data can be inserted and deleted anywhere in the list.

general tree: a tree with an unlimited outdegree.

generic code: a program design that uses one set of code that can be used to process any data type.

goezinta: a colloquial term derived from "goes into" for a bill of material.

graph: a non-linear list in which each element can have zero, one, or more predecessors and zero, one, or more successors.

greatest common divisor (GCD): in mathematics, the largest integral divisor of two numbers.

H

hashed list: a list in which the location of the data is determined by a hashing algorithm.

hashed search: any of the methods used to locate data in a hashed list.

hashing: a key-to-address transformation in which the key, through an algorithmic transformation, directly determines the location of the data.

head: in an ADT, a structure that contains the ADT's metadata, such as the address of the first node and the count of the number of elements in the ADT.

head pointer: a pointer that identifies the first element of a linked list.

header node: *see* head.

heap: a binary tree in which the root is the largest node in the tree and the subtrees are also heaps; *See also* minimum heap.

heap memory: a pool of memory that can be used to dynamically allocate space for data while the program is running; *see also* dynamic memory.

heap sort: a selection sort algorithm that uses a heap to determine the largest element in the unsorted area of an array.

heapify: the heap-processing algorithm that bubbles the largest element to the top of the heap.

height: a tree attribute indicating the length of the path from the root to the last level; the level of the leaf in the longest path from the root plus 1.

height-balanced tree: a balanced tree in which the balance is controlled through the height of its subtrees. An AVL tree is a height-balanced tree.

home address: in a hashed list, the first address produced by the hashing algorithm.

Huffman code: a compression algorithm that assign shorter codes to characters that occur more frequently and longer codes to those that occur less frequently.

I

indegree: in a tree or graph, the number of lines entering a node.

index: the address of an element within an array.

infix: a binary arithmetic notation in which the operator is placed between two operands.

infix traversal: in an expression tree, the traversal that results in the operator being placed between two operands. See inorder.

information hiding: a structured programming concept in which the user does not know the data structure or the implementation of its operations.

inorder: a binary tree traversal in which the root is processed after the left subtree and before the right subtree; an LNR traversal.

inorder traversal: see inorder.

insert: the addition of an element in a data structure or a file.

insertion: in a list, any process that adds data.

insertion sort: a sort algorithm in which the first element from the unsorted portion of the list is inserted into its proper position relative to the data in the sorted portion of the list.

intelligent data name: a data or algorithm name that describes the meaning of the data or the purpose of the algorithm.

internal node: any tree node except the root and the leaves; a node in the middle of a tree.

internal sort: a sort in which all of the data are held in primary storage during the sorting process.

K

key: one or more fields in a data structure that are used to identify the data or otherwise control its use.

key offset: a hashed list collision resolution method in which the next address is a function of the current address and the key.

key-sequenced insertion: in a general tree, the insertion method that places the new node in key sequence among the sibling nodes.

L

last in–first out (LIFO): a data structure processing sequence in which data are processed in the reverse order that they are retrieved; a stack.

leaf: a graph or tree node with an outdegree of 0.

leaf subtree: in a binary tree, the subtree on the left branch from a node.

left high: in an AVL tree, a tree or subtree whose left subtree has a height greater than its right subtree.

left of left: a temporary state of an unbalanced AVL tree in which a left high node has a left high subtree.

left of right: a temporary state of an unbalanced AVL tree in which a right high node has a left high subtree.

level: in a tree or graph, an attribute of a node, indicating its distance from the root.

lexical search tree: a lexicographical search tree in which each node contains only one character rather than an entire key, and the complete key is constructed by following a path through the tree.

lexicographical: a data order based on the dictionary; *see also* ascending sequence.

LIFO: see *last in–first out*.

LIFO insertion: an insertion method that places data in a list such that a subsequent traversal processes data in last in-first out sequence.

line: a graph element that connects two vertices in the graph; *see also* arc *and* edge.

linear efficiency: a measure of the efficiency of a module in which the run time is proportional to the number of elements being processed, as in $O(n) = n$.

linear list: a list structure in which each element, except the last, has a unique successor.

linear loop: a loop whose execution is a function of the number of elements being processed; *see also* linear efficiency.

linear probe: the collision resolution algorithm that determines the next candidate address by adding or subtracting 1.

linear search: any search algorithm used to locate data in a linear list.

link: in a list structure, the field that identifies the next element in the list.

linked list: a structure in which the ordering of the elements is determined by link fields.

linked list collision resolution: a hashed list collision resolution method that uses a separate area for synonyms, which are maintained in a linked list.

list: an ordered set of data contained in main memory.

list traversal: the list-processing algorithm that processes each element in a list in sequence.

LNR: left, node, right; *see* inorder.

load factor: in a hashed list, the ratio of the number of data nodes in the list to the number of physical elements in the list, expressed as a percentage.

logarithmic efficiency: a measure of the efficiency of a module in which the run time is proportional to the log of the number of elements being processed, as in $O(n) = \log n$.

logarithmic loop: a loop whose efficiency is a function of the log of the number of elements being processed; *see also* logarithmic efficiency.

logical data: data whose values can be only true or false; *see also* Boolean.

loop: a statement that iterates a block of code.

LRN: left, right, node; *see* postorder.

M

***m*:** the order in an *m*-way tree.

***m*-way tree:** a search tree structure with multiple data entries and subtrees per node; the maximum number of subtrees is known as the order of the *m*-way tree.

max-heap: *see* heap.

merge: to combine two or more sequential files into one sequential file based on a common key and structure format.

merge phase: in an external sort, a pass through the data that combines the data from two or more files.

merge run: a set of consecutively ordered data in a merge file.

merge sort: any sort technique that orders data by the repetitive merging of files.

metadata: data about the list or other data structure stored within the data structure itself.

midsquare hashing: a hashing method in which the key is squared and the address is selected from the middle of the squared number.

minimum heap: a binary tree in which the root is the smallest node in the tree and the subtrees are also min-heaps; contrast *heap* and *max-heap*.

minimum spanning tree: a spanning tree in which the sum of the weights is the minimum of all possible trees contained in the graph.

modular programming: the program design in which the program is written in a series of modules that are linear in their design; contrast *structured programming*.

modulo-division: a hashing method that divides the key by the array size and uses the remainder for the address.

multilinked list: one physical linked list structure with two or more logical key sequences.

multiserver queue: a queue design that can provide service to many customers at a time.

N

natural merge: a merge with a constant number of input merge files and one output merge file.

nearly complete tree: a tree with a limited outdegree that has the minimum height for its nodes and in which the leaf level is being filled from the left.

nested loop: a loop whose efficiency is a function of the efficiency of a controlling loop.

network: *see* weighted graph.

NLR: node, left, right; *see* preorder.

node: in a data structure, an element that contains both data and structural elements used to process the list.

non-linear list: a list in which each element can have more than one successor.

null tree: a tree with no nodes.

O

object: in object-oriented programming, any instantiation of a class, including its members and methods.

open addressing: a collision resolution method in which the new address is in the home area.

order: in an *m*-way tree, the maximum number of subtrees allowed for a node.

order (sort): *see* sort order.

ordered list: a list in which the elements are arranged so that the key values are placed in ascending or descending sequence.

outdegree: the number of lines leaving a node in a tree or a graph.

overflow: the condition that results when an attempt is made to insert data into a full list.

overflow area: in a hashed list, an area separate from the prime area that is used to store synonyms in a linked list.

P

parent: a tree or graph node with an outdegree greater than 0; that is, with successors.

parsing: logic that breaks data into independent pieces.

path: in a tree or graph, a sequence of nodes in which each node is adjacent to the next one.

pivot: in quicksort, the key value used to arrange the data such that all of the data with keys less than the pivot are on the left of the pivot in the array, and all of the data with keys greater than or equal to the pivot are on the right of the pivot.

pointer: a constant or variable that contains the address of a location in memory.

pointer to function: a pointer that identifies the entry point to a function. It is used to pass a function's address as a parameter.

pointer to *void*: a generic pointer type that can store a pointer to any type.

polynomic efficiency: a measure of the efficiency of a module in which the run time is proportional to the number of elements raised to the highest factor in a polynomial, as in $O(n) = n^k$.

polyphase merge: a merge in which a constant number of input merge files are merged into one output merge file and each input merge file is immediately reused when its input has been completely merged.

pop: the stack delete operation.

postfix: a binary arithmetic notation in which the operator is placed after its operands.

postfix traversal: in an expression tree, the traversal that results in the operator being placed after its two operands. See postorder.

postorder: a binary tree traversal in which the left subtree is processed before the right subtree and both are processed before the root; an LRN traversal.

postorder traversal: see postorder.

prefix: a binary arithmetic notation in which the operator is placed before the operands.

prefix traversal: in an expression tree, the traversal that results in the operator being placed before its two operands. See preorder.

preorder: a binary tree traversal in which the root is processed before the left subtree and the left subtree is processed before the right subtree; an NLR traversal.

preorder traversal: see preorder.

primary clustering: the buildup of data around the home address in a hashed list; contrast *secondary clustering*.

prime area: in a hashed list, the memory that contains the home addresses.

priority queue: a queue in which the elements are organized into groups according to priority numbers and processed such that the highest-priority elements are output first. Items with the same priority are processed in first in–first out (FIFO) order.

probability search: a search in which the list is ordered according to the probability of the list data being the target of a search, with the most probable targets first.

probe: in a hashing algorithm, the calculation of an address and a test for success; in a search algorithm, one iteration of the loop that includes the test for the search argument.

prune: in a trie, the process of removing all of the branches that are not needed.

pseudocode: English-like statements that follow a loosely defined syntax and are used to convey the design of an algorithm or a function.

pseudorandom collision resolution: a collision resolution method that uses a pseudorandom-number generator to determine the next address after a collision.

pseudorandom hashing: a hashing method that uses the key as the variable factor in a pseudorandom-number generator to determine the address.

pseudorandom numbers: a repeatable number series with random properties.

push: the stack insert operation.

Q

quadratic efficiency: a measure of the efficiency of a module in which the run time is proportional to the number of elements squared. Quadratic efficiency is one of the polynomic factors, as in $O(n) = n^2$.

quadratic loop: a nested loop in which each loop has a linear efficiency.

quadratic probe: a collision resolution method in which the increment is the collision probe number squared, giving the series 1^2, 2^2, 3^2, and so forth.

queue: a linear list in which data can be inserted only at one end, called the rear, and deleted from the other end, called the front.

queue data node: a node that contains the user data and a link field pointing to the next node.

queue data structure: the node in a queue structure that contains queue metadata, such as the front and rear pointers or a queue count.

queue simulation: a modeling activity used to generate statistics about the performance of a queue.

quick sort: the exchange sort developed by C. A. R. Hoare that sorts by repetitively dividing the list into three groups: a partition of elements whose keys are less than the pivot's key, the pivot element that is placed in its ultimately correct location in the list, and a partition of elements greater than or equal to the pivot's key.

R

random list: a list with no ordering of the data.

random number: a number selected from a set in which all members have the same probability of being selected.

rear: in a linked list, a pointer that identifies the last element; in a queue, the most recent element inserted into the structure.

rear pointer: in a list, a pointer that points to the last entry in the list.

recursion: a repetitive process in which an algorithm calls itself.

reflow: in a B-tree, the module that corrects an underflow either by moving data from one node to another or by combining nodes.

reheap down: in heap processing, the logic that flows an entry down the heap tree structure until it is in its proper location in the heap.

reheap up: in heap processing, the logic that flows an entry up the heap tree structure until it is in its proper location in the heap.

restricted list: a list in which data can be added or deleted only at the ends of the list and processing is restricted to operations on the data at the ends.

retrieval: the location and return of an element in a list.

return condition: in algorithm documentation, an explanation of any value returned by the algorithm.

right high: in an AVL tree, a tree or subtree whose right subtree has a height greater than its left subtree.

right of left: a temporary state of an unbalanced AVL tree in which a left high node has a right high subtree.

right of right: a temporary state of an unbalanced AVL tree in which a right high node has a right high subtree.

right subtree: in a binary tree, the subtree on the right branch from a node.

rollout: merge processing in which a consecutive series of merge input data are copied to the merge output after a stepdown in the alternate merge input.

root: the first node of a tree.

rotation: a hashing method in which the end portion of a key is copied to the front of the key.

S

search: the process that examines a list to locate one or more elements containing a designated value known as a search argument.

search argument: the key value being looked for in a search.

secondary clustering: the buildup of data along a collision path through a hashed list; contrast *primary clustering*.

selection sort: the sort algorithm in which the smallest value in the unsorted portion of a list is selected and placed at the end of the sorted portion of the list.

selection statement: a statement that evaluates a condition and executes zero or more alternatives.

self-referential data structure: a structure that contains a reference to itself.

sentinel: a flag that guards the end of a list or a file. The sentinel is usually the maximum value for a key field and cannot be a valid data value.

sentinel search: a search algorithm in which the search argument is placed in an extra element at the end of the list.

sequence: in structured programming, one or more statements that do not alter the execution path within an algorithm.

sequential file: a file structure in which data must be processed serially from the first entry in the file.

sequential search: a search technique used with a linear list in which the searching begins at the first element and continues until the value of an element equal to the value being sought is located, or until the end of the list is reached.

service: in a queue, any activity needed to accomplish the required result.

service time: in a queue, the average time required to complete the processing of a customer request.

shell sort: an insertion sort that uses a list divided into partitions. Each sort pass orders the data in their respective partitions. After each pass, the partition size is reduced and the sort repeated until the list is completely sorted.

siblings: two or more tree nodes with a common parent.

single-server queue: a queue that can process only one customer at a time.

singly linked list: a collection of nodes in which each element contains data and only one reference, which is the location of the next element; contrast *doubly linked list*.

sort: the process that orders a list or a file.

sort efficiency: a measure of sort performance.

sort order: the arrangement of data in a list or a file, either ascending or descending.

sort pass: in a sort, one traversal of the data.

sort phase: the first pass through the data in an external sort in which a sort algorithm is used to sort data into merge runs for further processing.

sort stability: an attribute of a sort in which input data with equal keys retain their relative order in the sort output.

spaghetti code: an archaic, unstructured coding style in which the logic flow wound through the program like spaghetti on a plate.

spanning tree: a tree extracted from a connected graph that contains all of the vertices in the graph.

stability: *see* sort stability.

stack: a restricted data structure in which data can be inserted and deleted at only one end, called the top.

stackframe: a logical structure used in a function call that contains the parameters, the local variable values for the calling function, the return statement in the calling function, and the address of the variable to receive any return value.

stepdown: an event that occurs when the sequential ordering of the data in a merge file is broken; the end of a merge run. End of file is also considered a stepdown.

straight insertion sort: a sort in which in each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate location.

strongly connected graph: a graph in which there is a path from every node to every other node; contrast *weakly connected graph*.

structure chart: a design and documentation tool that represents a program as a hierarchical flow of algorithms or functions.

structured programming: the program design technique in which a program is decomposed into modules that have one entry and one exit.

subtraction hashing: a hashing method in which only a constant value is subtracted from the key to determine the address.

subtree: any connected structure below the root of the tree.

synonym: in a hashed list, two or more keys that hash to the same home address.

T

threaded tree: a tree structure in which null (leaf) pointers are replaced with pointers to their successor nodes.

token: any syntactical construct that represents an operation or a flag, such as the plus sign (+).

top: in a stack, the next element to be removed.

traversal: an algorithmic process in which each element in a structure is processed once and only once.

tree: a non-linear list in which each node has only one predecessor.

trie: a lexical search tree in which null subtrees are pruned—that is, in which null subtrees are deleted.

U

underflow: an event that occurs when an attempt is made to delete data from a data structure and it is empty.

undirected graph: a graph in which there is no indication of direction on the lines.

V

vertex: a node in a graph.

vertex list: in a graph, a list of vertices.

W

weakly connected graph: a graph in which there is at least one node with no path to any other node; contrast *strongly connected graph*.

weighted graph: a graph whose lines are weighted.

This page intentionally left blank

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W