Osa
V

19. oppitunti

Linkitetyt listat

Edellisissä luvuissa käsiteltiin periytymistä, monimuotoisuutta ja abstrakteja tietotyyppejä. Olet oppinut myös käyttämään taulukoita. Nyt on aika parantaa taulukoiden hyödyntämistä yhdessä oliopohjaisten perusperiaatteiden kanssa: periytymisen, monimuotoisuuden ja kapseloinnin yhteydessä. Tässä luvussa on seuraavia aiheita:

- ❏ Mikä on linkitetty lista
- ❏ Kuinka linkitetty lista luodaan
- ❏ Kuinka toiminnallisuus kapseloidaan periytymisen kautta

Linkitetyt listat ja muut rakenteet

Taulukot ovat kuin astioita. Niihin voidaan tallentaa kamaa, mutta ne ovat kiinteän kokoisia. Jos otat esille liian suuren astian, tuhlaat siinä olevaa tilaa. Liian pienen astian kohdalla taas tapahtuu ylivuotamista ja syntyy sotkua.

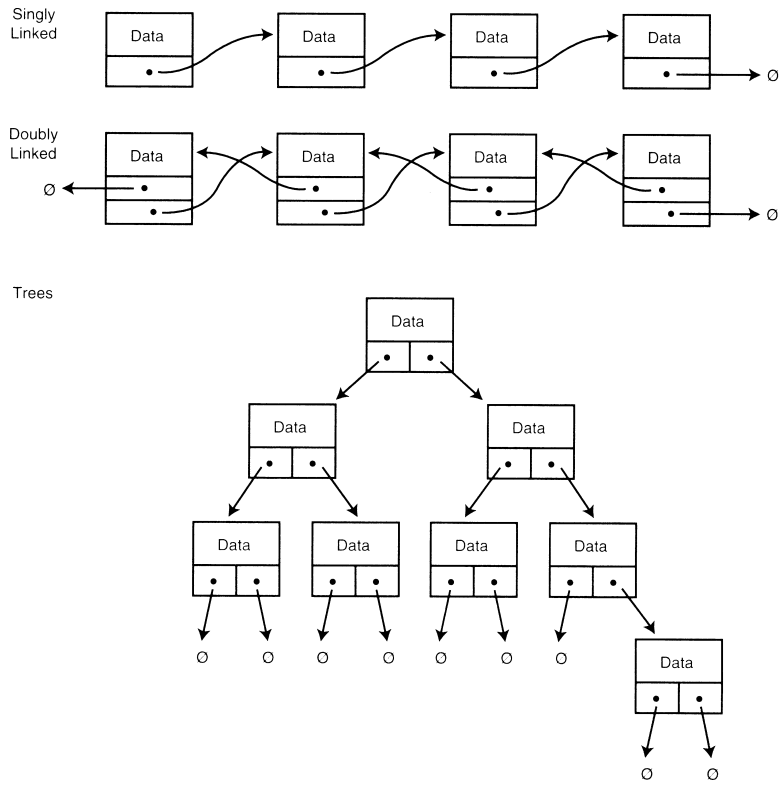
Eräs keino ratkaista tuo ongelma on käyttää linkitettyä listaa. Linkitetty lista on tietorakenne, joka sisältää pikkuastioita, jotka sopivat toisiinsa. Ideana on kirjoittaa luokka, joka tallentaa yhden tietokohteen tai olion, kuten CAT- tai Rectangle-olion, ja joka voi osoittaa listan seuraavaan astiaan. Luot siis yhden astian kullekin tallennettavalle kohteelle ja ketjutat ne toisiinsa.

Noita astioita kutsutaan solmuiksi. Ensimmäisen solmun nimi on head (listan alku) ja viimeisen taas tail (listan loppu).

Listoja on kolmea eri perusmuotoa. Yksinkertaisimmasta monimutkaisimpaan ne ovat:

- ❏ Yhteen suuntaan ketjutettu lista
- ❏ Kahteen suuntaan ketjutettu lista
- ❏ Puu

Yhteen suuntaan ketjutetussa listassa kukin solmu osoittaa seuraavaan, mutta ei takaisinpäin. Löytääksesi tietyn solmun on sinun aloitettava alusta ja kuljettava solmu solmulta eteenpäin, kunnes oikea löytyy. Kahteen suuntaan linkitetty lista sallii kulkemisen molempiin suuntiin. Puu on monimutkainen solmurakenne, jonka kukin solmu voi osoittaa kahteen tai kolmeen suuntaan. Kuva 19.1 havainnollistaa näitä listarakenteita.



Kuva 19.1. Linkitetyt listat.

Tieteilijät ovat kehittäneet vielä monimutkaisempia ja älykkäämpiä tietorakenteita, joista suurin osa liittyy solmujen sisäisiin kytkemisiin.

Tapaustutkimus

Tässä jaksossa tutkimme linkitettyä listaa tarkemmin. Luomme monimutkaisia rakenteita ja, mikä vielä tärkeämpää tutkimme, kuinka periytymistä, monimuotoisuutta ja kapselointia hyödynnetään laajojen projektien hallinnassa.

Vastuun jakaminen

Oliopohjaisen ohjelmoinnin eräs perusperiaate on se, että kukin kohde tekee yhden asian hyvin eikä jaa muille kohteille mitään, mikä ei ole sen tehtäväalueella.

Auto on täydellinen esimerkki tästä laitealueella: moottorin tehtävänä on tuottaa tehoa. Voiman jakelu ei ole moottorin tehtävä, vaan voimansiirtomekanismi hoitaa sen. Kääntäminen ei ole moottorin tai voimansiirron tehtävänä vaan se tapahtuu pyörien avulla.

Hyvin suunnitellussa moottorissa on runsaasti pieniä, käytännöllisiä osia, joista kukin suorittaa oman työnsä ja ne toimivat yhdessä aikaansaaden vielä enemmän hyötyä. Hyvin suunniteltu ohjelma on samanlainen: kukin luokkaa hoitaa omat työnsä, mutta yhdessä ne luovat todella toimivan yhteisön.

Listan osat

Linkitetty lista koostuu solmuista. Itse solmu on abstrakti; käytämme kolmea alityyppiä työn suorittamiseen. Listassa on alkusolmu (head, pää), jonka tehtävänä on hoitaa listan alku, loppusolmu (tail, häntä) (arvaa sen tehtävä!) ja sisäisiä solmuja. Sisäiset solmut hallitsevat listan todellista tietoa.

Huomaa, että tieto ja lista ovat aivan erillisiä. Teoriassa voit tallentaa mitä tahansa tietoa listaan. Tietoa ei siis linkitetä yhteen; ketjutetut solmut tallentavat tiedot.

Pääohjelma ei tiedä mitään solmuista; se vain käsittelee listaa. Lista tekee kuitenkin hieman töitä; se yksinkertaisesti jakaa vastuuta solmuille.

Listaus 19.1 käsittelee linkitettyä listaa; tutkimme koodin melko tarkkaan.

Listaus 19.1. Linkitetty lista.

```
1:  // ********************************************
2:  //    Tiedosto: Listaus 19.1
3:  //
4:  //    Tarkoitus: Linkitetyt listat
5:  //    Huomautuksia:
6:  //
7:  //  COPYRIGHT:  Copyright (C) 1997 Liberty Associates, Inc.
8:  //                All Rights Reserved
9:  //Esittelee oliopohjaista näkökulmaa linkitettyjen
10: //listojen käsittelyssä. Lista jakaa vastuun solmulle.
11: // Solmu on abstrakti tietotyyppi.
12: // Solmuja ovat head, tail js sisäinen solmu.
13: // Vain sisäinen solmu tallentaa
14: // tietoa.
15: //
16: // Data-luokka toimii kohteena, joka
17: // tallennetaan listaan.
18: //
19: // ********************************************
20:
21:
22: #include <iostream.h>
23:
24: enum { kIsSmaller, kIsLarger, kIsSame};
25:
26: // Data-luokka, joka sijoitetaan listaan.
27: // Jokaisen luokan on toteutettava seur. metodit:
28: // Show (esittää arvon) ja Compare (palauttaa sijainnin)
29: class Data
30: {
31: public:
32:   Data(int val):myValue(val){}
33:   ~Data(){}
34:   int Compare(const Data &);
35:   void Show() { cout << myValue << endl; }
36: private:
37:   int myValue;
38: };
39:
40:   // Compare päättää, minnen
41:   // tietty kohde sijoitetaan.
42:   int Data::Compare(const Data & theOtherData)
43:   {
44:     if (myValue < theOtherData.myValue)
45:       return kIsSmaller;
46:     if (myValue > theOtherData.myValue)
47:       return kIsLarger;
48:     else
49:       return kIsSame;
50:   }
51:
52: // ennalta esittelyt
53: class Node;
54: class HeadNode;
55: class TailNode;
56: class InternalNode;
57:
58: // ADT edustaa listan solmua
59: // Jokaisen johd. luokan on korvattava Insert ja Show
60: class Node
```

```
61: {
62: public:
63:   Node(){}
64:   virtual ~Node(){}
65:   virtual Node * Insert(Data * theData)=0;
66:   virtual void Show() = 0;
67: private:
68: };
69:
70: // Tämä solmu tallentaa tiedon.
71: // Tieto on nyt Data-luokan olio.
72: // Yleistämme tämän, kun käsittelemme
73: // malleja.
74: class InternalNode: public Node
75: {
76: public:
77:   InternalNode(Data * theData, Node * next);
78:   ~InternalNode(){ delete myNext; delete myData; }
79:   virtual Node * Insert(Data * theData);
80:   virtual void Show() { myData->Show(); myNext->Show(); }  // delegate!
81:
82: private:
83:   Data * myData;  // itse tieto
84:   Node * myNext;    // Osoittaa seuraav. solmuun
85: };
86:
87: // Muodostin vain alustaa
88: InternalNode::InternalNode(Data * theData, Node * next):
89: myData(theData),myNext(next)
90: {
91: }
92:
93: // listan ydin
94: // Kun laitat uuden tiedon listaan,
95: // se viedään solmulle, joka päättää,
96: // minne se sijoitetaan ja tekee sijoituksen
97: Node * InternalNode::Insert(Data * theData)
98: {
99:
100:   // Onko uusi kaveri pienempi vai isompi?
101:   int result = myData->Compare(*theData);
102:
103:
104:   switch(result)
105:   {
106:     // Jos se on sama, laitetaan edelle
107:     case kIsSame:        // eteenpäin
108:     case kIsLarger:    // uusi tieto tulee eteen
109:   {
110:   InternalNode * dataNode = new InternalNode(theData, this);
111:   return dataNode;
112: }
113:
114: // Se on isompi, joten viedään seuraavalle
115: // solmulle käsiteltäväksi.
116:     case kIsSmaller:
117:     myNext = myNext->Insert(theData);
118:   return this;
119: }
120:   return this;  // appease MSC
121: }
```

```
122:
123:
124: // Tail-solmu
125:
126: class TailNode : public Node
127: {
128: public:
129:   TailNode(){}
130:   ~TailNode(){}
131:   virtual Node * Insert(Data * theData);
132:   virtual void Show() { }
133:
134: private:
135:
136: };
137:
138: // Jos tieto tulee mulle, on se laitettava ennen
139: // Olen häntä, eikä jälkeeni tule mitään
140: Node * TailNode::Insert(Data * theData)
141: {
142:   InternalNode * dataNode = new InternalNode(theData, this);
143:   return dataNode;
144: }
145:
146: // Headilla ei ole dataa.
147: // se näyttää listan alun.
148: class HeadNode : public Node
149: {
150: public:
151:   HeadNode();
152:   ~HeadNode() { delete myNext; }
153:   virtual Node * Insert(Data * theData);
154:   virtual void Show() { myNext->Show(); }
155: private:
156:   Node * myNext;
157: };
158:
159: // As soon as the head is created
160: // it creates the tail
161: HeadNode::HeadNode()
162: {
163:   myNext = new TailNode;
164: }
165:
166: // Mitään ei tule Headia ennen, joten
167: // viedään tieto eteenpäin
168: Node * HeadNode::Insert(Data * theData)
169: {
170:   myNext = myNext->Insert(theData);
171:   return this;
172: }
173:
174: // Minä saan tulokset
175: class LinkedList
176: {
177: public:
178:   LinkedList();
179:   ~LinkedList() { delete myHead; }
180:   void Insert(Data * theData);
181:   void ShowAll() { myHead->Show(); }
182: private:
```

```
183:   HeadNode * myHead;
184: };
185:
186: // Aluksi luon Headin.
187: // Head luo Tailin.
188: // Tyhjä lista osoittaa Headiin, joka
189: // osoittaa Tailiin. Välissä ei mitään.
190: LinkedList::LinkedList()
191: {
192:   myHead = new HeadNode;
193: }
194:
195: // Delegate, delegate, delegate
196: void LinkedList::Insert(Data * pData)
197: {
198:   myHead->Insert(pData);
199: }
200:
201: // testiohjelma
202: int main()
203: {
204:   Data * pData;
205:   int val;
206:   LinkedList ll;
207:
208: // Pyydä käyttäjältä arvoja
209: // Laita ne listaan
210:   for (;;)
211:   {
212:     cout << "What value? (0 to stop): ";
213:     cin >> val;
214:     if (!val)
215:       break;
216:     pData = new Data(val);
217:     ll.Insert(pData);
218:   }
219:
220: // Käy lista läpi.
221:   ll.ShowAll();
222:   return 0;  // ll menee näkyvyysalueelta ja tuhotaan!
223: }
```

Tulostus

```
What value? (0 to stop): 5
What value? (0 to stop): 8
What value? (0 to stop): 3
What value? (0 to stop): 9
What value? (0 to stop): 2
What value? (0 to stop): 10
What value? (0 to stop): 0
2
3
5
8
9
10
```

Analyysi
Huomaa ensiksi luetteloitu vakio, jossa on kolme vakioarvoa: kIsSmaller, kIsLarger ja kIsSame. Jokaisessa linkitetyn listan oliossa tulee olla Compare()-metodi. Nuo vakiot ovat Compare()-metodin palautusarvoja.

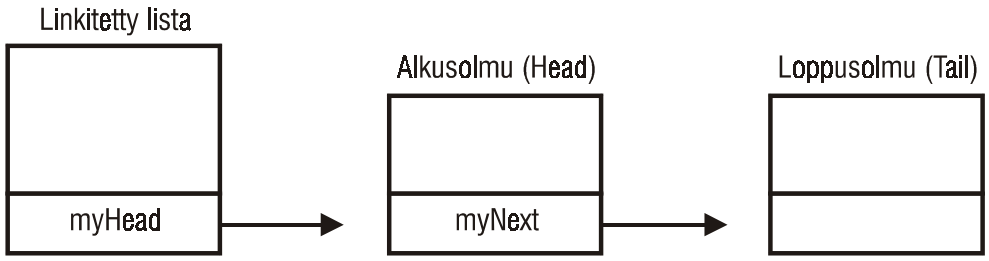
Luokka nimeltä Data luodaan riveillä 29-38 havainnollistamisen vuoksi ja Compare()-metodi toteutetaan riveillä 42-50. Jokainen Data-olio tallentaa arvon ja voi verrata itseään muihin Data-olioihin. Siinä on myös Show()-metodi, joka tulostaa Data-olion arvon.

Helpoin tapa oppia ymmärtämään linkitettyä listaa on tutkia ohjelmaa, jossa sellaista käytetään. Rivillä 202 alkaa pääohjelma. Rivillä 204 esitellään osoitin Data-olioon ja rivillä 206 määritellään paikallinen linkitetty lista.

Kun linkitetty lista on luotu, kutsutaan muodostinta rivillä 190. Muodostin varaa tilaa HeadNote-oliolle ja sijoittaa olion osoitteen linkitetyn listan osoittimeen rivillä 183.

Tässä HeadNode-varauksessa käytetään HeadNode-muodostinta (rivi 161). Se varaa vuorostaan tilaa TailNode-oliolle ja sijoittaa sen osoitteen alkusolmun myNext-osoittimeen. TailNode-olion luominen kutsuu TailNode-muodostinta (rivi 129), joka on inline-tyyppinen eikä tee mitään.

Täten muistin varaaminen pinosta linkitetylle listalle on luonut linkitetyn listan alku-ja loppusolmuineen. Samalla on solmujen yhteydet toteutettu, kuten kuva 19.2 osoittaa.



Kuva 19.2. Linkitetty lista luomisen jälkeen.

Rivi 209 aloittaa ikuisen silmukan. Käyttäjältä pyydetään arvoja lisättäviksi linkitettyyn listaan. Arvolla 0 syöttäminen lopetetaan. Rivi 213 tutkii annetun arvon, jos se on 0, silmukka lopetetaan.

Jos arvo ei ole 0, luodaan uusi Data-olio rivillä 215 ja se laitetaan listaan rivillä 216. Oletetaan, että käyttäjä antaa arvon 15. Tällöin kutsutaan Insert()-metodia rivillä 195.

Linkitetty lista jakaa olion sijoittamisvastuun alkusolmulleen. Se taas kutsuu metodia Insert() rivillä 167. Alkusolmu antaa vastuun heti sille solmulle, johon sen myNext-osoitin osoittaa. Tässä (ensimmäisessä)

tapauksessa se osoittaa loppusolmuun (muista, että alkusolmun luonnin yhteydessä luotiin linkki loppusolmuun). Nyt loppusolmu käyttää siis Insert()-metodia rivillä 139.

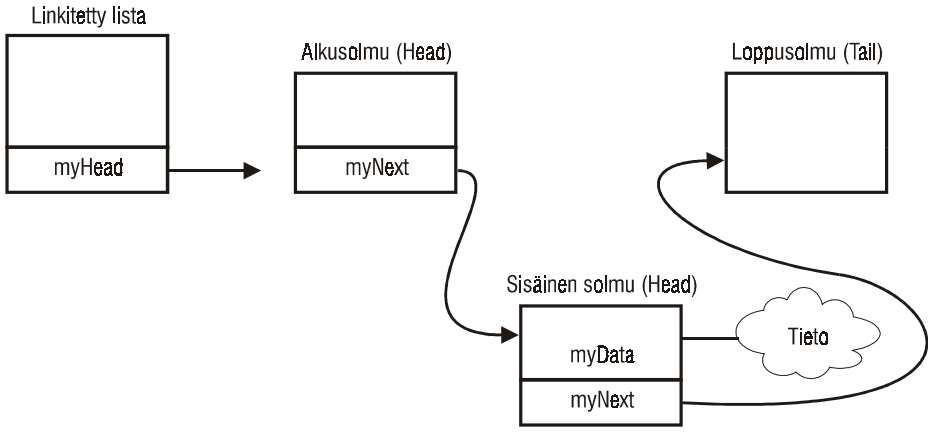
TailNode::Insert tietää, että sille annettu olio on sijoitettava välittömästi sitä ennen. Siksi se luo rivillä 141 uuden InternalNode-olion vieden sille annetun arvon ja samalla osoittimen itselleen. Tällöin käytetään muodostinta InternalNode-oliolle (rivi 87).

InternalNode-muodostin ei tee muuta kuin alustaa sen Data-osoittimen sille viedyn Data-olion osoitteella ja myNext-osoittimen sille viedyn solmun osoitteella. Tässä tapauksessa sen osoittama solmu on loppusolmu (muista, että loppusolmu vei sisään oman this-osoittimensa).

Nyt, kun InternalNode on luotu, sijoitetaan uuden solmun osoite dataNode-osoittimeen rivillä 141 ja tuo osoite puolestaan palautetaan TailNode::Insert()-metodista. Se palauttaa meidät HeadNode::Insert()-metodiin, jossa InternalNode-osoite sijoitetaan HeadNoden myNext-osoittimeen (rivillä 169). Lopuksi palautetaan HeadNoden osoite linkitettyyn listaan, josta se heitetään pois rivillä 197 (sillä ei tehdä mitään, koska linkitetty lista tietää jo alkusolmun osoitteen).

Miksi osoite sitten palautetaan, jos sitä ei käytetä? Insert()-metodi esitellään perusluokassa, Node. Muut toteutukset tarvitsevat palautusarvoa. Jos muutat HeadNode::Insert()-metodin palautusarvoa, saat kääntämisvirheen; on yksinkertaisempaa vain palauttaa HeadNode ja antaa linkitetyn listan heittää osoitteensa pois.

Mitäpä tapahtui? Tieto sijoitettiin listaan. Lista vei tiedon alkusolmulle. Alkusolmu taas vei tiedon solmulle, johon se sattui osoittamaan. Tässä (ensimmäisessä) tapauksessa alkusolmu osoitti loppusolmuun. Loppusolmu loi välittömästi uuden sisäisen solmun, jonka se alusti osoittamaan loppusolmuun. Loppusolmu palautti sitten uuden solmun osoitteen alkusolmulle, joka laittoi myNext-osoittimensa osoittamaan uuteen solmuun. Voila! Listan tieto on omalla paikallaan, kuten kuva 19.3 osoittaa.



Kuva 19.3. Linkitetty lista, kun siihen on sijoitettu uusi solmu.

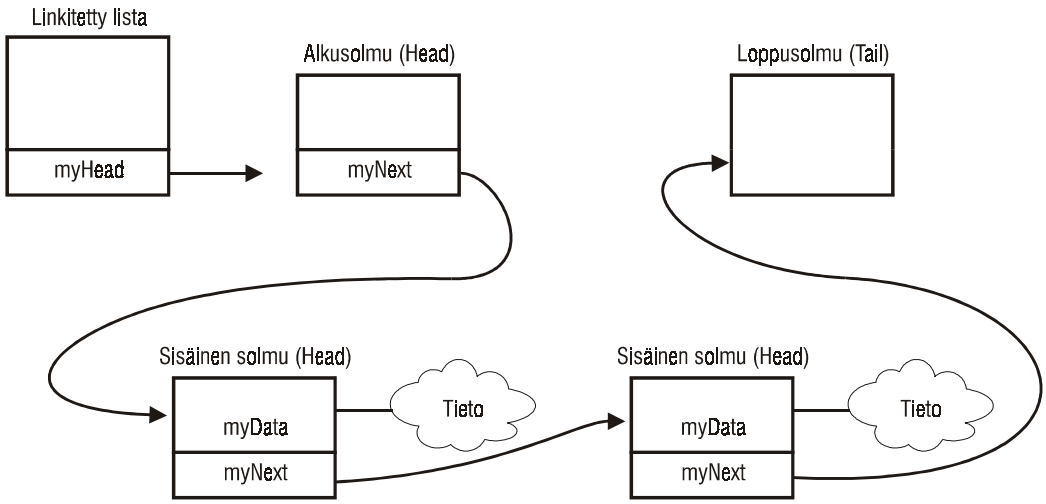
Ensimmäisen solmun sijoittamisen jälkeen ohjelman suoritus palaa riville 211. Jälleen tutkitaan käyttäjän antama arvo. Olettakaamme, että käyttäjä antaa nyt arvon 3. Tällöin luodaan rivillä 215 uusi Data-olio, joka sijoitetaan listaan rivillä 216.

Rivillä 197 vie lista luvun HeadNode-solmulleen. HeadNode::Insert()-metodi vie uuden arvon solmulle, johon sen myNext osoittaa. Nythän se osoitti solmuun, jossa on aiempi arvo, 15. Tuo Data-olio kutsuu nyt InternalNode::Insert()-metodia rivillä 96.

Rivillä 100 käyttää InternalNode myData-osoitintaan pyytämään Data-oliotaan (sitä, jonka arvo on 15) kutsumaan Compare()-metodia vieden sille uuden Data-olion (jonka arvo on 3). Seuraavaksi kutsutaankin Compare()-metodia (rivi 41).

Noita kahta arvoa verrataan toisiinsa ja koska myValue on 15 ja theOtherData.myValue taas 3, on palautusarvona kIsLarger. Tämä aiheuttaa ohjelman suorituksen hyppäämisen riville 109.

Uusi InternalNode luodaan uudelle Data-oliolle. Uusi solmu osoittaa nykyiseen InternalNode-olioon ja uuden InternalNoden osoite palautetaan InternalNode::Insert()-metodista HeadNodelle. Siten uusi solmu, jonka olion arvo oli siis pienempi kuin nykyisen solmun olion arvo, sijoitetaan listaan. Nyt lista näyttää kuvan 19.4 mukaiselta.



Kuva 19.4. Linkitetty lista sen jälkeen, kun toinen solmu on sijoitettu.

Kolmannella silmukkakierroksella antaa käyttäjä arvon 8. Se on suurempi kuin 3, mutta pienempi kuin 15, joten se tulisi sijoittaa aiemman kahden solmun väliin. Sijoittaminen tapahtuu edellä kuvatulla tavalla. Nyt vertailun tuloksena vain palautetaan arvo kIsSmaller.

Tämä saa InternalNode::Insert()-metodin haarautumaan rivillä 116. Uuden solmun luomisen ja sijoittamisen sijaan InternalNode vie uuden arvon sen

solmun Insert()-metodille, johon myNext-osoitin sattuu osoittamaan. Tässä tapauksessa se käyttää sen InternalNode-solmun InsertNode-metodia, jonka Data-olion arvo on 15.

Vertailu suoritetaan uudelleen ja luodaan uusi InternalNode. Tämä uusi InternalNode osoittaa InternalNodeen, jonka Data-olion arvo on 15 ja sen osoite viedään takaisin InternalNodelle, jonka Data-olion arvo on 3 (rivi 116).

Tuon kaiken tuloksena sijoitetaan uusi solmu listaan juuri oikeaan paikkaan.

Jos mahdollista, sijoita uusia solmuja listaan ja tutki koodin suorittamista askeltaen debuggerillasi. Sinun tulisi nähdä, kuinka metodit kutsuvat toisiaan ja osoittimet säädetään oikealla tavalla.

Mitä olet oppinut, Paavo?

"Jos saan tehdä sen, mitä sydämeni ensi sijassa halajaa, menen katselemaan takapihaani". Oma koti on kullan kallis, se tiedetään. Samoin on totta se, että lausekielistä ohjelmointia ei voita mikään. Lausekielisessä ohjelmoinnissa tutkisi valvontamenettely tiedon ja kutsuisi sitten funktioita.

Tässä oliopohjaisessa lähestymistavassa annetaan kullekin yksittäiselle oliolle kapea ja hyvin määritelty vastuualueensa. Linkitetty lista on vastuussa alkusolmun ylläpidosta. Alkusolmu vie uuden tiedon solmulle, johon se osoittaa.

Loppusolmu luo uuden solmun ja sijoittaa sen listaan aina, kun sillä on tietoa. Se tietää vain yhden asian: jos tämä tuli minulle, on se laitettava juuri ennen minua.

Sisäiset solmut ovat hieman monimutkaisempia; ne pyytävät olemassaolevaa oliotaan vertaamaan itseään uuteen olioon. Tuloksesta riippuen ne joko sijoittavat uuden olion tai vievät sen eteenpäin.

Huomaa, että sisäinen solmu ei osaa tehdä vertailua; se jätetään oliolle itselleen tehtäväksi. Ainoa, mitä sisäinen solmu osaa, on pyytää olioita vertaamaan itseään ja odottaa yhtä kolmesta mahdollisesta vastauksesta. Kun vastaus on saatu, se sijoittaa solmun; muutoin se vain vie sen eteenpäin tietämättä tai välittämättä siitä, mitä sille tapahtuu.

Kuka on sitten vastuussa? Hyvin suunnitellussa oliopohjaisessa ohjelmassa ei kukaan ole vastuussa. Kukin olio tekee oman pienen työnsä ja kaiken tuloksena on hyvin toimiva kokonaisuus.

Yhteenveto

Tässä luvussa sait tietoa linkitetyistä listoista.

Linkitetty lista on dynaamisesti kasvava kokoelma. Kukin tietokohde pidetään listan solmussa. Tiettyyn solmuun päästään aloittamalla listan alusta ja kulkemalla listaa loppua kohti.

Luvussa esitettiin myös, kuinka rajallista toiminnallisuutta kapseloidaan kuhunkin listan luokkaan ja kuinka hyvin suunnitellut oliot voivat tuoda synergiaa.

Kysymyksiä ja Vastauksia

K
Miksi käyttäisin linkitettyjä listoja, kun voin käyttää taulukoita?
V
Taulukon koko on kiinteä, kun taas linkitettyä listaa voidaan kasvattaa dynaamisesti ajon aikana.

K
Miksi tieto-olio erotetaan solmusta?
V
Kun solmukohteet toimivat oikein, voidaan tuota koodia käyttää uudelleen kaikkiin muihin listan olioihin.

K
Jos haluan lisätä muita olioita listaan, onko minun luotava uusi listatyyppi ja uusi solmutyyppi?
V
Toistaiseksi kyllä. Ratkaisemme tuon ongelman, kun pääsemme malleihin.