5
Listas encadeadas

Neste capítulo

- Encadeamentos
- Uma Lista Encadeada Simples
- Localizando e Eliminando Nós Especificados
- Listas com duas Extremidades
- Eficiência da Lista Encadeada
- Tipos Abstratos de Dados
- Listas Ordenadas
- Listas Duplamente Encadeadas
- Iteradores

No Capítulo 2, "Vetores", vimos que vetores tinham certas desvantagens como estruturas de armazenamento de dados. Em um vetor não ordenado, a busca é lenta, ao passo que em um vetor ordenado, a inserção é lenta. Em ambos os tipos de vetores, a eliminação é lenta. Além disso o tamanho de um vetor não pode ser alterado depois dele ter sido criado.

Neste capítulo, veremos uma estrutura de armazenamento de dados que resolve alguns desses problemas: a *lista encadeada*. Listas encadeadas são provavelmente as estruturas de armazenamento de dados de uso geral mais usadas depois de vetores.

A lista encadeada é um mecanismo versátil adequado para uso em muitos tipos de bases de dados de uso geral. Ela também pode substituir um vetor como a base para outras estruturas de armazenamento, tais como pilhas e filas. Na verdade, você pode usar uma lista encadeada em muitos casos nos quais você usa um vetor, a menos que precise de acesso aleatório freqüente a itens individuais usando um índice.

Listas encadeadas não são a solução para todos os problemas de armazenamento de dados, mas são surpreendentemente versáteis e conceitualmente mais simples do que algumas outras estruturas populares, tais como árvores. Analisaremos suas capacidades e fraquezas à medida que seguirmos.

Neste capítulo, veremos listas encadeadas simples, listas com duas extremidades, listas ordenadas, listas duplamente encadeadas e listas com iteradores (uma abordagem para acesso aleatório a elementos de uma lista). Também examinaremos a idéia de Tipos Abstratos de Dados (TADs), veremos como pilhas e filas podem ser vistas como TADs e como elas podem ser implementadas como listas encadeadas ao invés de vetores.

Nós

Em uma lista encadeada, cada item de dados é incorporado em uma *nó*. Um nó é um objeto de uma classe chamada Link. Como há muitos nós parecidos em uma lista, faz sentido usar uma classe separada para eles, distinta da lista encadeada em si. Cada objeto Link contém uma referência (geralmente chamada next) para o próximo nó da lista. Um campo da própria lista contém uma referência para o primeiro nó. Este relacionamento é mostrado na Figura 5.1.

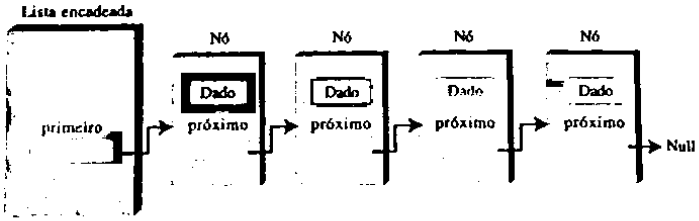

Figura 5.1 Nós em uma lista.

Eis uma parte da definição de uma classe Link. Ela contém alguns dados e uma referência ao próximo nó:

```
class  Link
  {
  public  int  iData;     // dado
  public  double  dData;  // dado
  public  Link  next;     // referência ao próximo nó
  }
```

Este tipo de definição de classe é algumas vezes chamado de *auto-referencial* porque ela contém um campo – chamado next neste caso – do mesmo tipo dela mesma.

Mostramos apenas dois itens de dados no nó: um int e um double. Em uma aplicação típica haveria muito mais. Um registro de funcionário, por exemplo, poderia ter nome, endereço, número do Seguro Social, título, salário e muitos outros campos. Em geral, um objeto de uma classe que contém esses dados é usado ao invés dos itens:

```
class  Link
  {
  public  inventoryItem  iI;   // objeto contendo dados
  public  Link  next;          // referência ao próximo nó
  }
```

Referências e tipos básicos

Você pode ficar facilmente confuso com as referências no contexto de listas encadeadas, portanto revisemos como elas funcionam.

Ser capaz de colocar um campo do tipo Link dentro da definição da classe desse mesmo tipo pode parecer estranho. O compilador não ficaria confuso? Como ele poderá descobrir com qual tamanho criar um objeto Link se um nó contém um link e o compilador não sabe ainda o tamanho de um objeto Link?

A resposta é que em Java, um objeto Link não contém de fato outro objeto Link, embora possa parecer que sim. O campo next do tipo Link é apenas uma *referência* a outro nó, não um objeto.

Uma referência é um número que se *refere a* um objeto. Ele é o endereço do objeto na memória do computador, mas você não precisa saber seu valor; você apenas irá tratá-lo como um número mágico que informa onde o objeto está. Em um dado computador/ sistema operacional, todas as referências, não importando a que se referem, são do mesmo tamanho. Assim não há nenhum problema para o compilador descobrir qual tamanho esse campo deverá ter e com isso construir um objeto Link inteiro.

Observe que em Java, tipos primitivos como int e double são armazenados de modo bem diferente que os objetos. Campos contendo tipos primitivos não contêm referências, mas valores numéricos reais como 7 ou 3.14159. Uma definição de variável como

```
double salary = 65000.00;
```

cria um espaço na memória e coloca o número 65000.00 nele. Porém, uma referência a um objeto como

```
Link aLink = someLink;
```

coloca uma referência a um objeto do tipo Link, chamada someLink, em aLink. O próprio objeto someLink está localizado em algum outro lugar. Ele não é movido ou mesmo criado por essa instrução; ele tem que ter sido criado antes. Para criar um objeto, você terá que usar sempre new:

```
Link someLink = new Link();
```

Mesmo o campo someLink não mantém um objeto; ele é ainda apenas uma referência. O objeto está em algum outro lugar na memória, como mostrado na Figura 5.2.

Outras linguagens, tais como C++, lidam com objetos de modo bem diferente de Java. Em C++, um campo como

```
Link next;
```

de fato contém um objeto do tipo Link. Você não pode escrever uma definição de classe auto-referencial em C++ (embora possa colocar um ponteiro para Link na classe Link; um ponteiro é similar a uma referência). Programadores C++ devem estar atentos a como Java lida com objetos; esse uso pode não ser intuitivo.

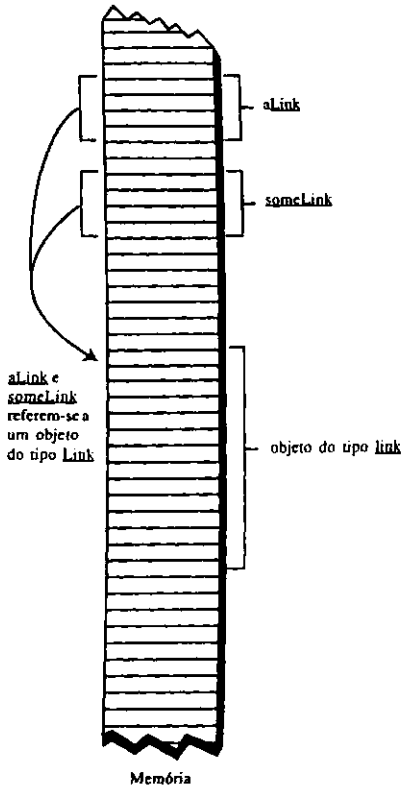

Figura 5.2 Objetos e referências em memória.

Relacionamento, não posição

Examinemos um dos principais modos no qual listas encadeadas diferem de vetores. Em um vetor, cada item ocupa uma certa posição. Essa posição pode ser acessada diretamente usando um número de índice. É como uma vila de casas: você pode encontrar uma determinada casa usando seu endereço.

Em uma lista, a única maneira de encontrar um elemento em particular é seguir a seqüência de elementos. É mais parecido com relações humanas. Talvez você pergunte a Harry onde Bob está. Harry não sabe, mas acha que Jane deveria saber, portanto, você vai e pergunta a Jane. Jane viu Bob sair do escritório com Sally, portanto você liga para o telefone celular de Sally. Ela deixou Bob no escritório de Peter, portanto... mas você tem uma idéia. Você não pode acessar um item de dados diretamente; você tem que usar os relacionamentos entre os itens para localizá-lo. Você começa com o primeiro item, vai para o segundo, então o terceiro, até encontrar o que está procurando.

O applet LinkList Workshop

O applet LinkList Workshop fornece três operações de lista. Você pode inserir um novo item de dados, buscar um item de dados com uma chave especificada e eliminar um item de dados com uma chave especificada. Essas operações são as mesmas que exploramos no applet Array Workshop no Capítulo 2; elas são adequadas para uma aplicação de base de dados de uso geral.

A Figura 5.3 mostra como o applet LinkList Workshop fica quando iniciado. Inicialmente, há 13 nós na lista.

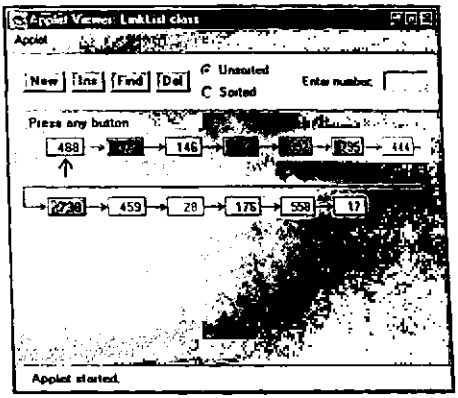

Figura 5.3 O applet LinkList Workshop.

O botão Insert

Se você acha que 13 não é um número de sorte, você pode inserir um novo nó. Pressione o botão Ins (Inserir) e você será solicitado a fornecer um valor-chave entre 0 e 999. Pressionares subseqüentes irão gerar um nó com este dado nele, como mostrado na Figura 5.4.

Nesta versão de uma lista encadeada, novos nós são sempre inseridos no início da lista. Esta é a abordagem mais simples, embora você também possa inserir nós em qualquer lugar na lista, como veremos posteriormente.

Um pressionar final em Ins irá redesenhar a lista para que o nó recém-inserido alinhe-se com os outros nós. Esse redesenhar não representa qualquer coisa ocorrendo no programa propriamente dito, apenas torna a exibição mais clara.

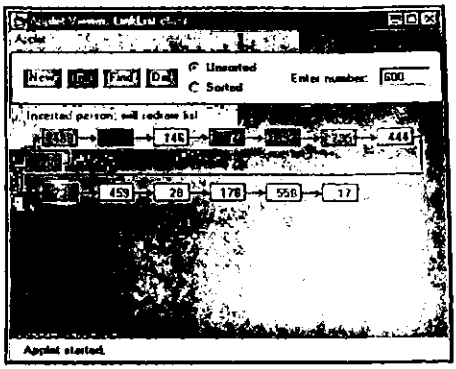

Figura 5.4 Um novo nó sendo inserido.

O botão Find

O botão Find (Localizar) permite encontrar um nó com um valor-chave especificado. Quando solicitado, digite o valor de um nó existente, de preferência um em algum lugar no meio da lista. Quando continuar a pressionar o botão, você verá a seta vermelha mover-se na lista, procurando o nó. Uma mensagem informa quando a seta encontra o nó. Se você digitar um valor-chave inexistente, a seta buscará até o final da lista antes de informar que o item não pode ser encontrado.

O botão Delete

Você também pode remover uma chave com um valor especificado. Digite o valor de um nó existente e pressione repetidamente Del (Apagar). Novamente, a seta mover-se-á na lista, procurando o nó. Quando a seta encontrar o nó, ela simplesmente irá removê-lo e conectar a seta do nó anterior diretamente ao nó seguinte. Isto é como nós são removidos: a referência ao nó anterior é alterada para apontar para o nó seguinte.

Um pressionar final irá redesenhar a figura, mas novamente o redesenho fornecerá apenas ligações igualmente espaçadas por razões estéticas; o comprimento das setas não corresponde a qualquer coisa no programa.

Nota

O applet LinkList Workshop pode criar tanto listas não ordenadas como ordenadas. Não ordenada é o padrão. Mostraremos como usar o applet para listas ordenadas quando as analisarmos mais tarde neste capítulo.

Uma lista encadeada simples

Nosso primeiro programa exemplo, linkList.java, demonstra uma lista encadeada simples. As únicas operações permitidas nesta versão de uma lista são

- Inserir um item no início da lista
- Eliminar o item no início da lista
- Fazer uma iteração na lista para exibir seu conteúdo

Estas operações são bem fáceis de executar, portanto começaremos com elas. (Como veremos mais tarde, essas operações são também tudo que você precisa para usar uma lista encadeada como a base para uma pilha.)

Antes de chegarmos ao programa linkList.java completo, veremos algumas partes importantes das classes Link e LinkList.

A classe Link

Você já viu a parte de dados da classe Link. Eis a definição completa da classe:

```
class Link
  {
  public int iData;                  // item de dado
  public double dData;               // item de dado
  public Link next;                  // próximo nó na lista
//-------------------------------------------------------------
  public Link(int id, double dd)     // construtor
    {
    iData = id;                      // inicializa dados
    dData = dd;                      // ('next' é automaticamente
    }                                // definido como null)
//-------------------------------------------------------------
  public void displayLink()          // nos exibe
    {
    System.out.print("{" + iData + ", " + dData + "} ");
    }
  } // fim da classe Link
```

Além dos dados, há um construtor e um método, displayLink(), que exibe os dados do nó no formato {22, 33.9}. Os puristas de objetos provavelmente teriam objeção em nomear esse método como displayLink(), argumentando que ele devia ser simplesmente display(). Usar

o nome mais curto estaria no espírito do polimorfismo, mas tornará a listagem um pouco mais difícil de entender quando você vir uma instrução como

```
current.display();
```

e tiver esquecido se current é um objeto Link, um objeto LinkList ou outra coisa.

O construtor inicializa os dados. Não há necessidade de inicializar o campo next porque ele será definido automaticamente como null quando for criado. (Porém, você poderia defini-lo como null explicitamente, por clareza.) O valor null significa que ele não se refere a qualquer coisa, que é a situação até que o nó seja conectado a outros nós.

Tomamos o tipo de armazenamento dos campos Link (iData etc.) public. Se eles fossem private, precisaríamos fornecer métodos públicos para acessá-los, o que iria requerer código extra, tornando assim a listagem mais longa e mais difícil de ler. Como ideal, por segurança provavelmente gostaríamos de limitar o acesso ao objeto Link aos métodos da classe LinkList. Porém, sem uma relação de herança entre essas classes, isso não é muito conveniente. Poderíamos usar o especificador de acesso padrão (sem palavra-chave) para fornecer aos dados *acesso de pacote* (acesso limitado às classes no mesmo diretório), mas isso não tem efeito nesses programas demo, que ocupam apenas um diretório de qualquer modo. O especificador public pelo menos torna claro que este dado não é privado. Em um programa mais sério, provavelmente você gostaria de tornar todos os campos de dados na classe Link privados.

A classe LinkList

A classe LinkList contém apenas um item de dados: uma referência para o primeiro nó da lista. Essa referência é chamada de first. Ela é a única informação permanente que a lista mantém sobre o local de qualquer dos nós. Ela encontra os outros nós seguindo a seqüência de referências a partir de first, usando o campo next de cada nó:

```
class  LinkList
  {
  private  Link  first;          // ref. ao primeiro nó da lista

//--------------------------------------------------------------
  public  void  LinkList()       //  construtor
    {
    first = null;                // sem itens na lista ainda
    }
//--------------------------------------------------------------
  public  boolean  isEmpty()   //verdadeiro se a lista estiver vazia
    {
    return  (first= =null);
    }
//--------------------------------------------------------------
// ...outros métodos entram aqui
  }
```

O construtor para LinkList define first como null. Isto não é realmente necessário porque, como mencionamos, referências são definidas como null automaticamente quando elas são criadas. Contudo, o construtor explícito torna claro que é assim que first começa.

Quando first tem o valor null, sabemos que não há itens na lista. Se houvesse qualquer item, first conteria uma referência para o primeiro. O método isEmpty() usa esse fato para determinar se a lista está vazia.

O método insertFirst()

O método insertFirst() de LinkList insere uma novo nó no início da lista. Este é o local mais fácil para inserir um nó porque first já aponta para o primeiro nó. Para inserir o novo nó, precisaremos apenas definir o campo next no nó recém-criado para apontar para o primeiro nó antigo e então alterar first para que ele aponte para o nó recém-criado. Esta situação é mostrada na Figura 5.5.

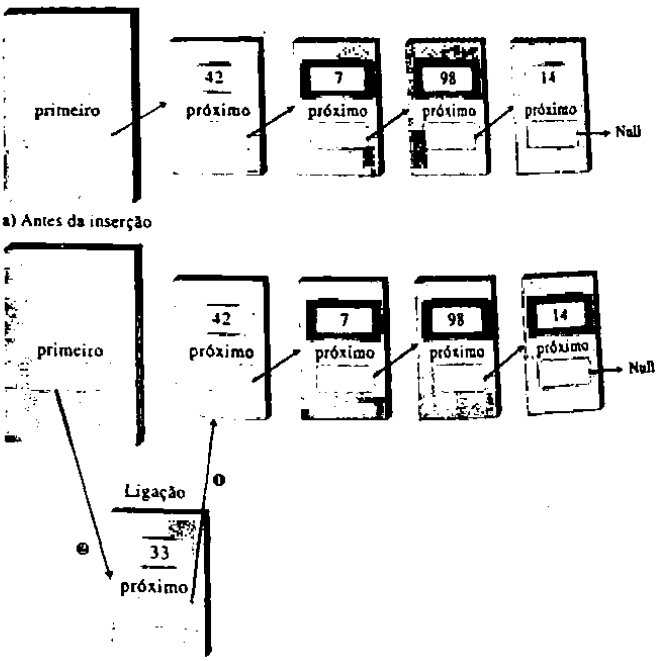

Figura 5.5 Inserindo um novo nó.

Em insertFirst(), começamos criando o novo nó usando os dados passados como argumentos. Então alteramos as referências aos nós como acabamos de mencionar:

```
                                    // insere no início da lista
public void insertFirst(int id, double dd)
  {                                 // cria novo nó
  Link newLink = new Link(id, dd);
  newLink.next = first;             // newLink --> first antigo
  first = newLink;                  // first --> newLink
  }
```

As setas —> nos comentários nas duas últimas instruções significam que um nó (ou o campo first) conecta-se ao próximo (lista abaixo) nó. (Nas listas duplamente encadeadas, veremos conexões lista acima também, simbolizadas por setas <—.) Compare essas duas instruções com a Figura 5.5. Certifique-se de que tenha compreendido como as instruções fazem com que as ligações sejam alteradas, como mostrado na figura. Esse tipo de manipulação de referência é o centro dos algoritmos de listas encadeadas.

O método deleteFirst()

O método deleteFirst() é o inverso de insertFirst(). Ele desconecta o primeiro nó roteando de novo first para apontar para o segundo nó. Esse segundo nó é encontrado vendo o campo next no primeira nó:

```
public Link deleteFirst()           // elimina o primeiro item
  {                                 // (assume que a lista não está vazia)
  Link temp = first;                // salva referência ao nó
  first = first.next;               // elimina-o: first->next antigo
  return temp;                      // retorna nó eliminado
  }
```

A segunda instrução é tudo que você precisa para remover o primeiro nó da lista. Escolhemos também retornar o nó, por conveniência do usuário da lista encadeada, portanto salvamo-lo em temp antes de eliminá-lo e retornamos o valor de temp. A Figura 5.6 mostra como first é roteada de novo para eliminar o objeto.

Em C++ e em linguagens similares, você precisaria preocupar-se em eliminar o próprio nó depois dele ter sido desconectado da lista. Ele fica na memória em algum lugar, mas agora nada se refere a ele. O que aconteceria com ele? Em Java, o processo de coleta de lixo o destruirá em algum ponto no futuro; isto não é sua responsabilidade.

Note que o método deleteFirst() supõe que a lista não esteja vazia. Antes de chamá-lo, seu programa deve verificar esse fato com o método isEmpty().

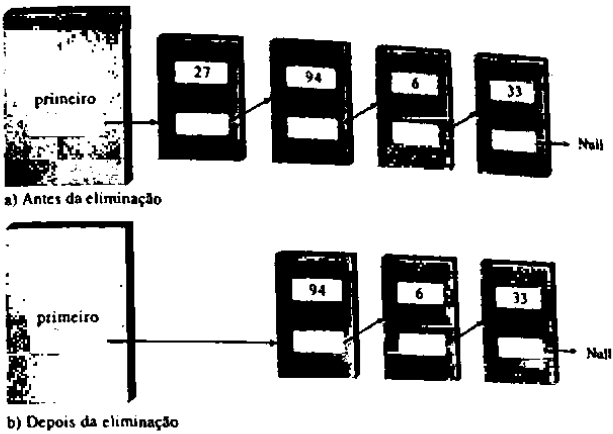

Figura 5.6 Eliminando um nó.

O método displayList()

Para exibir a lista, você começará em first e seguirá a seqüência de referências de nó em nó. Uma variável current aponta para (ou tecnicamente *refere-se* a) cada nó por vez. Ela começa apontando para first, que mantém uma referência para o primeiro nó. A instrução

```
current = current.next;
```

altera current para apontar para o próximo nó porque isto é o que está no campo next em cada nó. Eis o método displayList() inteiro:

```
public void displayList()
  {
  System.out.print("List (first-->last): ");
  Link current = first;          // começa no início da lista
  while(current != null)         // até o fim da lista,
    {
    current.displayLink();       // imprime dados
    current = current.next;      // move para o próximo nó
    }
  System.out.println("");
  }
```

O final da lista é indicado pelo campo next no último nó apontando para null ao invés de outro nó. Como esse campo ficou null? Ele começou assim quando o nó foi criado e nunca lhe foi fornecido qualquer outro valor porque ele sempre estava no final da lista. O laço while usa essa condição para terminar a si mesmo quando atinge o final da lista. A Figura 5.7 mostra como current avança na lista.

Listagem 5.1 O programa linkList.java (continuação)

```
 public  void  displayLink()          // nos exibe
   {
   System.out.print("{" + iData + ", " + dData + "}");
   }
 }   // fim da classe Link
////////////////////////////////////////////////////////////////
class  LinkList
 {
 private  Link  first;                //ref. ao primeiro nó na lista

//--------------------------------------------------------------
 public  LinkList()                   //  construtor
   {
   first = null;                      // sem itens na lista ainda
   }
//--------------------------------------------------------------
 public  boolean  isEmpty()//verdadeiro  se  a  lista  estiver  vazia
   {
   return  (first=  =null);
   }
//--------------------------------------------------------------
                                      // insere no início da lista
 public  void  insertFirst(int  id,  double  dd)
   {                                  // cria novo nó
   Link newLink  =  new  Link(id,  dd);
   newLink.next  =  first;            // newLink -> first antigo
   first  =  newLink;                 // first -> newLink
   }
//--------------------------------------------------------------
 public  Link  deleteFirst()          // elimina primeiro item
   {                                  // (assume que a lista não está vazia)
   Link temp  =  first;               // salva referência ao nó
   first  =  first.next;              // elimina-o:  first->next  antigo
   return  temp;                      // retorna nó eliminado
   }
//--------------------------------------------------------------
 public  void  displayList()
   {
   System.out.print("List  (first->last):  ");
   Link current  =  first;            // começa no início da lista
   while(current  !=  null)           // até o fim da lista,
     {
     current.displayLink();           // imprime dados
     current  =  current.next;        // move para o próximo nó
     }
   System.out.println("");
   }
//--------------------------------------------------------------
 } // fim da classe LinkList
////////////////////////////////////////////////////////////////
class  LinkListApp
 {
 public  static  void  main(String[  ]  args)
   {
   LinkList  theList  =  new  LinkList();   // criar nova lista
```

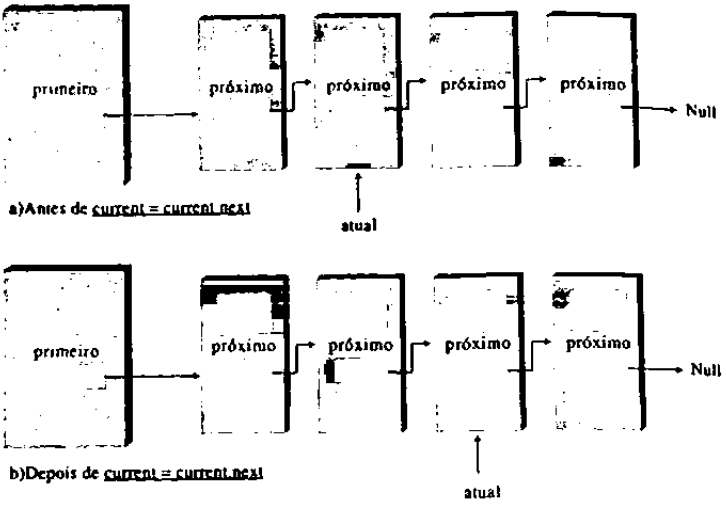

Figura 5.7 Avançando na lista.

Em cada nó, o método displayList() chama o método displayLink() para exibir os dados no nó.

O programa linkList.java

A Listagem 5.1 mostra o programa linkList.java completo. Você já viu todos os componentes, exceto a rotina main().

Listagem 5.1 O programa linkList.java

```
// linkList.java
// demonstra lista encadeada
// para executar este programa: C>java LinkListApp
////////////////////////////////////////////////////////////////
class Link
  {
  public int iData;              // item de dado (chave)
  public double dData;           // item de dado
  public Link next;              // próximo nó na lista
//-------------------------------------------------------------
  public Link(int id, double dd)  // construtor
    {
    iData = id;                  // inicializa dados
    dData = dd;                  // ('next' é automaticamente
    }                            // definido como null)
//-------------------------------------------------------------
```

Listagem 5.1 O programa linkList.java (continuação)

```
    theList.insertFirst(22,  2.99);       // inserir quatro itens
    theList.insertFirst(44,  4.99);
    theList.insertFirst(66,  6.99);
    theList.insertFirst(88,  8.99);

    theList.displayList();                // exibir lista

    while( !theList.isEmpty() )           // até ela estar vazia,
      {
      Link aLink = theList.deteleFirst();//  eliminar nó
      System.out.print("Deleted ");      //  exibi-lo
      aLink.displayLink();
      System.out.println("");
      }
    theList.displayList();                // exibir lista
    } // fim de main()
  } // fim da classe LinkListApp
////////////////////////////////////////////////////////////////
```

Em main(), criamos uma nova lista, inserimos quatro nós novos nela com insertFirst() e a exibimos. Então no laço while, removemos os itens um por um com deleteFirst() até que a lista estivesse vazia. A lista vazia é então exibida. Eis a saída de linkList.java:

```
List (first->last): (88, 8.99) (66, 6.99) (44, 4.99) (22, 2.99)
Deleted (88, 8.99)
Deleted (66, 6.99)
Deleted (44, 4.99)
Deleted (22, 2.99)
List (first->last):
```

Localizando e eliminando nós especificados

Nosso próximo programa exemplo adiciona métodos para buscar uma lista encadeada para obter um item de dados com um valor-chave especificado e para eliminar um item com um valor-chave especificado. Estas, junto com inserção no início da lista, são as mesmas operações executadas pelo applet LinkList Workshop. O programa linkList2.java completo é mostrado na Listagem 5.2.

Listagem 5.2 O programa linkList2.java

```
// linkList2.java
// demonstra lista encadeada
// para executar este programa: C>java LinkList2App
////////////////////////////////////////////////////////////////
class Link
  {
  public int iData;                   // item de dado (chave)
  public double dData;                // item de dado
  public Link next;                   // próximo nó na lista
//--------------------------------------------------------------
```

Listagem 5.2 O programa linkList2.java (continuação)

```
public Link(int id, double dd)  // construtor
  {
  iData = id;
  dData = dd;
  }
//--------------------------------------------------------------
public void displayLink()       // nos exibe
  {
  System.out.print("{" + iData + ", " + dData + "}");
  }
}  // fim da classe Link
////////////////////////////////////////////////////////////////
class LinkList
{
private Link first;             // ref. ao primeiro nó n a lista
//--------------------------------------------------------------
public LinkList()               // construtor
  {
  first = null;                 // sem nós na lista ainda
  }
//--------------------------------------------------------------
public void insertFirst(int id, double dd)
  {                             // cria novo nó
  Link newLink = new Link(id, dd);
  newLink.next = first;         //ele aponta p/ o prim. nó antigo
  first = newLink;              // agora first aponta para este
  }
//--------------------------------------------------------------
public Link find(int key)       // encontrar nó com chave dada
  {                             // (assume lista não vazia)
  Link current = first;           // começa em 'first'
  while(current.iData != key)     // enquanto não coincide,
    {
    if(current.next == null)      // se fim da lista,
      return null;                // não o encontrou
    else                          // não é fim da lista,
      current = current.next;     // ir p/ próximo nó
    }
  return current;                 // o encontrou
  }
//--------------------------------------------------------------
public Link delete(int key )    // delete link with given key
  {                             // (assume lista não vazia)
  Link current = first;           // buscar nó
  Link previous = first;
  while(current.iData != key)
    {
    if(current.next == null)
      return null;                // não o encontrou
    else
      {
      previous = current;         // ir p/ próximo nó
      current = current.next;
      }
    }                             // o encontrou
```

Listagem 5.2 O programa linkList2.java (continuação)

```
    if(current == first)                    // se primeiro nó,
      first = first.next;                   // alterar first
    else                                    // caso contrário,
      previous.next = current.next;         // contorná-lo
    return current;
    }
//--------------------------------------------------------------
  public void displayList()                 // exibe a lista
    {
    System.out.print("List (first->last): ");
    Link current = first;                   // começa no início da lista
    while(current != null)                  // até o fima da lista,
      {
      current.displayLink();                // imprime dados
      current = current.next;               // move p/ próximo nó
      }
    System.out.println("");
    }
//--------------------------------------------------------------
  } // fim da classe LinkList
////////////////////////////////////////////////////////////////
class LinkList2App
  {
  public static void main(String[ ] args)
    {
    LinkList theList = new LinkList();      // criar lista

    theList.insertFirst(22, 2.99);          // inserir 4 itens
    theList.insertFirst(44, 4.99);
    theList.insertFirst(66, 6.99);
    theList.insertFirst(88, 8.99);

    theList.displayList();                  // exibir lista

    Link f = theList.find(44);              // encontrar item
    if( f != null)
      System.out.println("Found link with key " + f.iData);
    else
      System.out.print("Can't find link");

    Link d = theList.delete(66);            // eliminar item
    if( d != null )
      System.out.println("Deleted link with key " + d.iData);
    else
      System.out.println("Can't delete link");

    theList.displayList();                  // exibir lista
    } // fim de main()
  } // fim da classe LinkList2App
////////////////////////////////////////////////////////////////
```

A rotina main() cria uma lista, insere quatro itens e exibe a lista resultante. Então busca o item com a chave 44, elimina o item com a chave 66 e exibe a lista novamente. Eis a saída:

```
List (first->last): (88, 8.99) (66, 6.99) (44, 4.99) (22, 2.99)
Found link with key 44
Deleted link with key 66
List (first->last): (88, 8.99) (44, 4.99) (22, 2.99)
```

O método find()

O método find() funciona de modo muito parecido com o método displayList() no programa linkList.java. A referência current aponta inicialmente para first e então percorre os nós definindo a si mesma repetidamente para current.next. Em cada nó, find() verifica se a chave do nó é aquela que ele está procurando. Se a chave for encontrada, ele retornará uma referência para esse nó. Se find() atingir o final da lista sem encontrar o nó desejado, retornará null.

O método delete()

O método delete() é similar a find() no sentido de que ele busca o nó a ser eliminado. Porém, ele precisa manter uma referência não apenas para o nó atual (current), mas para o nó que antecede o nó atual (previous). Ele faz isso porque, se ele eliminar o nó atual, terá que conectar o nó anterior ao nó seguinte, como mostrado na Figura 5.8. A única maneira de dizer onde o nó anterior está localizado é manter uma referência para ele.

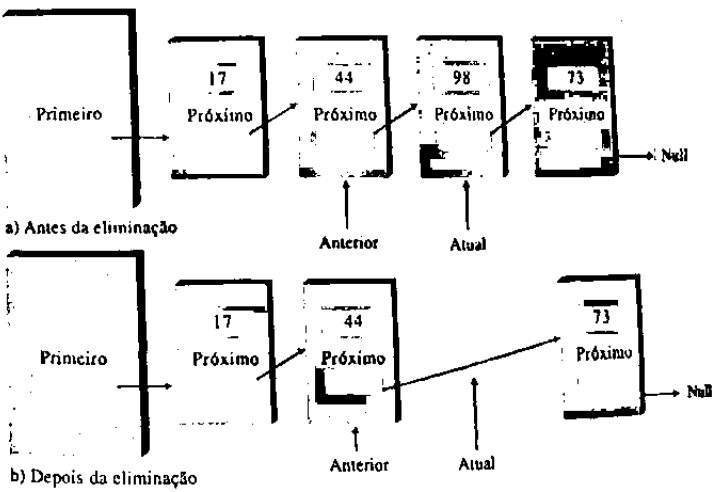

Figura 5.8 Eliminando um nó especificado.

A cada ciclo no laço while, logo antes de current ser definida para current.next, previous é definida para current. Isso a mantém apontando para o nó antes de current.

Para eliminar o nó atual assim que ele é encontrado, o campo next do nó anterior é definido para o próximo nó. Um caso especial surgirá se o nó atual for o primeiro nó, porque o primeiro nó é apontado pelo campo first de LinkList e não por outro nó. Neste caso, o nó é eliminado alterando first para apontar para first.next, como vimos no programa linkList.java com o método deleteFirst(). Eis o código que cobre essas duas possibilidades:

```
                                    // o encontrou
if(current == first)                // se primeiro nó,
  first = first.next;               // alterar first
else                                // caso contrário,
  previous.next = current.next;     // contornar nó
```

Outros métodos

Vimos métodos para inserir e eliminar itens no início de uma lista e para encontrar um item especificado e eliminar um item especificado. Você pode imaginar outros métodos úteis de lista. Por exemplo, um método insertAfter() poderia encontrar um nó com um valor-chave especificado e inserir uma novo nó depois dele. Veremos um tal método quando falarmos sobre iteradores de lista ao final deste capítulo.

Listas com extremidades duplas

Uma lista com extremidades duplas é similar a uma lista encadeada comum, mas ela tem um recurso adicional: uma referência para o último nó, assim como para o primeiro. A Figura 5.9 mostra uma tal lista.

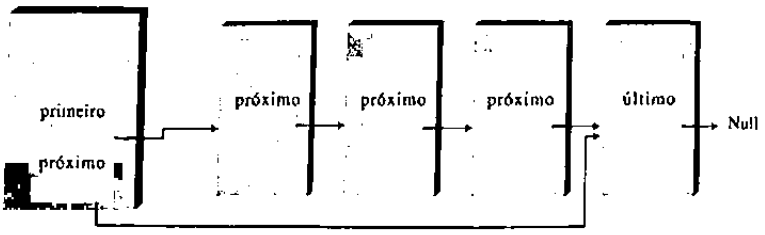

Figura 5.9 Uma lista com extremidades duplas.

A referência para o último nó permite inserir um novo nó diretamente no final da lista, assim como no início. Naturalmente, você pode inserir um novo nó no final de uma lista com extremidade simples fazendo uma iteração pela lista inteira até atingir o final, mas essa abordagem é ineficiente.

Acesso ao final da lista, assim como ao início, torna a lista com extremidades duplas adequada para certas situações nas quais uma lista com extremidade simples não pode lidar com eficiência. Uma tal situação é implementar uma fila; veremos como essa técnica funciona na próxima seção.

A Listagem 5.3 contém o programa firstLastList.java, que demonstra uma lista com extremidades duplas. (Incidentalmente, não confunda lista com extremidades duplas com lista duplamente encadeada, que iremos explorar mais tarde neste capítulo.)

Listagem 5.3 O programa firstLastList.java

```
// firstLastList.java
// demonstra lista com primeira e última referências
// para executar este programa: C>java FirstLastApp
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                    // item de dado
  public Link next;                     // próximo no na lista
//--------------------------------------------------------------
  public Link(long d)                   // construtor
    { dData = d; }
//--------------------------------------------------------------
  public void displayLink()             // exibe este nó
    { System.out.print(dData + " "); }
//--------------------------------------------------------------
  }  // fim da classe Link
////////////////////////////////////////////////////////////////
class FirstLastList
  {
  private Link first;                   // ref. ao primeiro nó
  private Link last;                    // ref. ao último nó
//--------------------------------------------------------------
  public FirstLastList()                // construtor
    {
    first = null;                       // sem nós na lista ainda
    last = null;
    }
//--------------------------------------------------------------
  public boolean isEmpty()              // verdadeiro se não há nós
    { return first==null; }
//--------------------------------------------------------------
  public void insertFirst(long dd)      // insere na frente da lista
    {
    Link newLink = new Link(dd);        // cria novo nó

    if( isEmpty() )                     // se lista vazia,
      last = newLink;                   // newLink <- last
    newLink.next = first;               // newLink -> first antigo
    first = newLink;                    // first -> newLink
    }
//--------------------------------------------------------------
  public void insertLast(long dd)       // insere no fim da lista
    {
    Link newLink = new Link(dd);        // cria novo nó
```

Listagem 5.3 O programa firstLastList.java (continuação)

```
    if( isEmpty() )                 // se lista vazia,
      first = newLink;              // first -> newLink
    else
      last.next = newLink;          // last antigo -> newLink
    last = newLink;                 // newLink <- last
    }
  //--------------------------------------------------------------
    public long deleteFirst()       // elimina primeiro nó
    {                               // (assume lista vazia)
    long temp = first.dData;
    if(first.next == null)          // se apenas um item
      last = null;                  // null <- last
    first = first.next;             // first -> next antigo
    return temp;
    }
  //--------------------------------------------------------------
    public void displayList()
    {
    System.out.print("List (first->last): ");
    Link current = first;           // começa no início
    while(current != null)          // até o fim da lista,
      {
      current.displayLink();        // imprime dados
      current = current.next;       // move para prox. nó
      }
    System.out.println("");
    }
  //--------------------------------------------------------------
  }  // fim da classe FirstLastList
////////////////////////////////////////////////////////////////
class FirstLastApp
  {
  public static void main(String[ ] args)
    {                           // criar uma nova lista
    FirstLastList theList = new FirstLastList();

    theList.insertFirst(22);        // inserir na frente
    theList.insertFirst(44);
    theList.insertFirst(66);

    theList.insertLast(11);         // inserir no final
    theList.insertLast(33);
    theList.insertLast(55);

    theList.displayList();          // exibir a lista

    theList.deleteFirst();          //eliminar dois primeiros itens
    theList.deleteFirst();

    theList.displayList();          // exibir novamente
    } // fim de main()
  }  // fim da classe FirstLastApp
////////////////////////////////////////////////////////////////
```

Por simplicidade, neste programa reduzimos o número de itens de dados em cada nó de dois para um. Isso torna mais fácil exibir o conteúdo do nó. (Lembre-se que em um programa sério haveria muitos mais itens de dados ou uma referência para outro objeto contendo muitos itens de dados.)

Esse programa insere três itens na frente da lista, insere mais três no final e exibe a lista resultante. Ele então apaga os dois primeiros itens e exibe a lista novamente. Eis a saída:

```
List (first->last): 66 44 22 11 33 55
List (first->last): 22 11 33 55
```

Note como repetidas inserções na frente da lista invertem a ordem dos itens, ao passo que repetidas inserções no final preservam a ordem.

A classe da lista com extremidades duplas é chamada de FirstLastList. Como analisado, ela tem dois itens de dados, first e last, que apontam para o primeiro item e o último item na lista. Se houver apenas um item na lista, first e last apontarão para ele e, se não houver nenhum item, ambos serão null.

A classe tem um método novo, insertLast(), que insere um novo item no final da lista. Esse processo envolve modificar last.next para apontar para o novo nó e então alterar last para apontar para o novo nó, como mostrado na Figura 5.10.

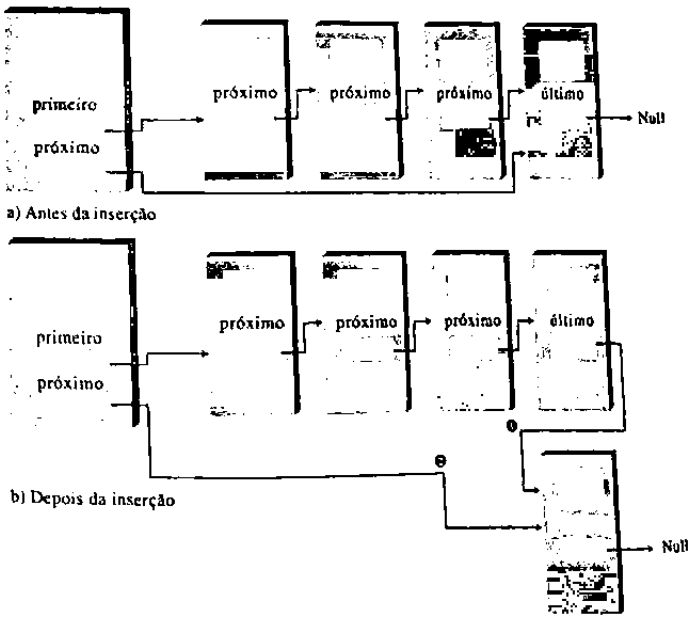

Figura 5.10 Inserção no final de uma lista.

As rotinas de inserção e de eliminação são similares àquelas em uma lista com uma extremidade. Porém, ambas as rotinas de inserção têm que estar atentas ao caso especial quando a lista estiver vazia antes da inserção. Ou seja, se isEmpty() for true, então insertFirst() terá que definir last para o novo nó e insertLast() terá que definir first para o novo nó.

Se inserindo no início com insertFirst(), first será definido para apontar para o novo nó, mas inserindo no final com insertLast(), last será definido para apontar para o novo nó. Eliminando no início da lista também será um caso especial se este for o último item na lista: last terá que ser definido para apontar para null neste caso.

Infelizmente, criar uma lista com extremidades duplas não lhe ajudará a apagar o último nó, porque ainda não há referência para o nó próximo ao último, cujo campo next precisaria ser alterado para null se o último nó fosse apagado. Para apagar convenientemente o último nó, você precisará de uma lista duplamente encadeada, que veremos em breve. (Naturalmente, você poderia também percorrer a lista inteira para encontrar o último nó, mas isto não é muito eficiente.)

Eficiência da lista encadeada

Inserção e eliminação no início de uma lista encadeada são muito rápidas. Elas envolvem alterar apenas uma ou duas referências, que leva tempo O(1).

Localizar, eliminar ou inserir próximo a um item específico requer buscar, em média, metade dos itens na lista. Isso requer O(N) comparações. Um vetor também é O(N) para estas operações, mas, entretanto, a lista encadeada é mais rápida porque nada precisa ser movido quando um item é inserido ou eliminado. O aumento de eficiência pode ser significativo, especialmente se uma cópia levar muito mais tempo que uma comparação.

Naturalmente, outra vantagem importante de listas encadeadas sobre vetores é que uma lista encadeada usa exatamente a memória da qual ela precisa e pode expandir-se para preencher toda a memória disponível. O tamanho de um vetor é fixado quando ele é criado; isso geralmente leva a uma ineficiência porque o vetor é grande demais ou fica sem espaço porque é pequeno demais. Vetores, que são vetores expansíveis, podem resolver esse problema até certo ponto, mas em geral eles se expandem em aumentos com tamanho fixo (como por exemplo dobrar o tamanho do vetor sempre que ele estiver para esgotar-se). Essa solução ainda não é um uso de memória tão eficiente quanto uma lista encadeada.

Tipos abstratos de dados

Nesta seção, mudaremos de direção e analisaremos um tópico que é mais geral que listas encadeadas: Tipos Abstratos de Dados (TADs). O que é TAD? A grosso modo, é uma maneira de ver uma estrutura de dados: concentrar-se no que ela faz e ignorar como ela faz seu serviço.

Pilhas e filas são exemplos de TADs. Já vimos que tanto pilhas quanto filas podem ser implementadas usando vetores. Antes de voltarmos para uma análise dos TADs, vejamos como pilhas e filas podem ser implementadas usando listas encadeadas. Essa análise demonstrará a natureza "abstrata" de pilhas e filas: como elas podem ser consideradas separadamente da sua implementação.

Uma pilha implementada por uma lista encadeada

Quando criamos uma pilha no Capítulo 4, "Pilhas e Filas", usamos um vetor Java comum para manter os dados da pilha. As operações push() e pop() da pilha foram de fato executadas por operações de vetor tais como

```
arr[++popl  =  data;
```

e

```
data  =  arr[top--];
```

que insere dados em, e retira de, um vetor.

Podemos também usar uma lista encadeada para manter dados de uma pilha. Neste caso, as operações push() e pop() seriam executadas por operações como

```
theList.insertFirst(data)
```

e

```
data  =  theList.deleteFirst()
```

O usuário da classe da pilha chama push() e pop() para inserir e eliminar itens sem saber, ou precisar saber, se a pilha foi implementada como um vetor ou como uma lista encadeada. A Listagem 5.4 mostra como uma classe da pilha chamada LinkStack pode ser implementada usando a classe LinkList, em vez de um vetor. (Os puristas de objetos argumentariam que o nome LinkStack deveria ser simplesmente Stack porque os usuários dessa classe não deveriam precisar saber que ela é implementada como uma lista.)

Listagem 5.4 O programa linkStack.java

```
// linkStack.java
// demonstra uma pilha implementada como uma lista
// para executar este programa: C>java LinkStackApp
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                    // item de dado
  public Link next;                     // próximo nó na lista
//-------------------------------------------------------------
  public Link(long dd)                  // construtor
    { dData = d; }
//-------------------------------------------------------------
  public void displayLink()             // mostramos nós mesmos
    { System.out.print(dData + " "); }
  }  // fim da classe Link
////////////////////////////////////////////////////////////////
```

Listagem 5.4 O programa linkStack.java (continuação)

```
class LinkList
  {
  private Link first;                    // ref. ao primeiro item na lista
//--------------------------------------------------------------
  public LinkList()                      //  construtor
    { first = null; }                    // sem itens na lista ainda
//--------------------------------------------------------------
  public boolean isEmpty()// verdadeiro se a lista estiver vazia
    { return (first= =null); }
//--------------------------------------------------------------
  public void insertFirst(long dd)       // insere no início da lista
    {                                    // cria novo nó
    Link newLink = new Link(dd);
    newLink.next = first;                // newLink -> antigo first
    first = newLink;                     // first -> newLink
    }
//--------------------------------------------------------------
  public long deleteFirst()              // elimina primeiro item
    {                                    // (assume lista não vazia)
    Link temp = first;                   // guarda referência ao nó
    first = first.next;                  //elimina-o:first->próximo antigo
    return temp.dData;                   // retorna nó eliminado
    }
//--------------------------------------------------------------
  public void displayList()
    {
    Link current = first;                // começa no início da lista
    while(current != null)               // vai até o final da lista,
      {
      current.displayLink();             // imprime dado
      current = current.next;            // move para o próximo nó
      }
    System.out.println("");
    }
//--------------------------------------------------------------
  } // fim da classe LinkList
////////////////////////////////////////////////////////////////
class LinkStack
  {
  private LinkList theList;
//--------------------------------------------------------------
  public LinkStack()                     //  construtor
    {
    theList = new LinkList();
    }
//--------------------------------------------------------------
  public void push(long j)               // coloca item no topo da pilha
    {
    theList.insertFirst(j);
    }
//--------------------------------------------------------------
  public long pop()                      // retira item do topo da pilha
    {
    return theList.deleteFirst();
    }
//--------------------------------------------------------------
```

Listagem 5.4 O programa linkStack.java (continuação)

```
public boolean isEmpty()          // verdadeiro se a pilha estiver vazia
  {
  return ( theList.isEmpty() );
  }
//--------------------------------------------------------------
public void displayStack()
  {
  System.out.print("Stack (top->bottom): ");
  theList.displayList();
  }
//--------------------------------------------------------------
} // fim da classe LinkStack
////////////////////////////////////////////////////////////////
class LinkStackApp
{
public static void main(String[ ] args)
  {
  LinkStack theStack = new LinkStack();  // criar pilha

  theStack.push(20);                     // empilhar itens
  theStack.push(40);

  theStack.displayStack();               // mostrar pilha

  theStack.push(60);                     // empilhar itens
  theStack.push(80);

  theStack.displayStack();               // mostrar pilha

  theStack.pop();                        // desempilhar itens
  theStack.pop();

  theStack.displayStack();               // mostrar pilha
  } // fim de main()
} // fim da classe LinkStackApp
////////////////////////////////////////////////////////////////
```

A rotina main() cria um objeto pilha, empilha dois itens, exibe a pilha, empilha mais dois itens e exibe a pilha novamente. Finalmente, desempilha dois itens e exibe a pilha uma terceira vez. Eis a saída:

```
Stack (top->bottom):  40 20
Stack (top->bottom):  80 60 40 20
Stack (top->bottom):  40 20
```

Note a organização geral deste programa. A rotina main() na classe LinkStackApp relaciona-se apenas com a classe LinkStack. A classe LinkStack relaciona-se apenas com a classe LinkList. Não há comunicação entre main() e a classe LinkList.

Mais especificamente, quando uma instrução em main() chama a operação push() na classe LinkStack, esse método por sua vez chama insertFirst() na classe LinkList para, de fato, inserir dados. Do mesmo modo, pop() chama deleteFirst() para eliminar um item e displayStack()

chama displayList() para exibir a pilha. Para o usuário da classe, escrevendo o código em main(), não há diferença entre usar a classe LinkStack baseada em listas e usar a classe stack baseada em vetores a partir do programa stack.java (Listagem 4.1) no Capítulo 4.

Uma fila implementada por uma lista encadeada

Eis um exemplo similar de um ADT implementado com uma lista encadeada. A Listagem 5.5 mostra uma fila implementada como uma lista encadeada com extremidades duplas.

Listagem 5.5 O programa linkQueue.java

```
// linkQueue.java
// demonstra fila implementada como uma lista com extremidades duplas
// para executar este programa: C>java LinkQueueApp
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                    // item de dado
  public Link next;                     // próximo nó na lista
//-------------------------------------------------------------
  public Link(long d)                   // construtor
    { dData = d; }
//-------------------
  public void displayLink()             // exibe este nó
    { System.out.print(dData + " "); }
//-------------------------------------------------------------
  }  // fim da classe Link
////////////////////////////////////////////////////////////////
class FirstLastList
  {
  private Link first;                   // ref. ao primeiro item
  private Link last;                    // ref. ao último item
//-------------------------------------------------------------
  public FirstLastList()                // construtor
    {
    first = null;                       // sem itens na lista ainda
    last = null;
    }
//-------------------------------------------------------------
  public boolean isEmpty()              // verdadeiro se não há nós
    { return (first==null); }
//-------------------------------------------------------------
  public void insertLast(long dd)       // insere no final da lista
    {
    Link newLink = new Link(dd);        // cria novo nó
    if( isEmpty() )                     // se lista vazia,
      first = newLink;                  // primeiro -> newLink
    else
      last.next = newLink;              // antigo last -> new Link
    last = newLink;                     // newLink <-last
    }
//-------------------------------------------------------------
  public long deleteFirst()             // elimina primeiro nó
    {                                   // (assume lista não vazia)
```

Listagem 5.5 O programa linkQueue.java (continuação)

```
   long temp = first.dData;
   if(first.next == null)              // se apenas um item
     last = null;                      // null <- last
   first = first.next;                 // first -> antigo next
   return temp;
   }
//--------------------------------------------------------------
 public void displayList()
   {
   Link current = first;               // começa no início
   while(current != null)              // vai até o final da lista,
     {
     current.displayLink();            // imprime dado
     current = current.next;           // move para o próximo nó
     }
   System.out.println("");
   }
//--------------------------------------------------------------
 } // fim da classe FirstLastList
////////////////////////////////////////////////////////////////
class LinkQueue
 {
 private FirstLastList theList;
//--------------------------------------------------------------
 public LinkQueue()                    // construtor
   { theList = new FirstLastList(); }  // cria uma lista 2-fim
//--------------------------------------------------------------
 public boolean isEmpty()              // verdadeiro se lista vazia
   { return theList.isEmpty(); }
//--------------------------------------------------------------
 public void insert(long j)            // insere, final da fila
   { theList.insertLast(j); }
//--------------------------------------------------------------
 public long remove()                  // remove, da frente da fila
   { return theList.deleteFirst(); }
//--------------------------------------------------------------
 public void displayQueue()
   {
   System.out.print("Queue (front->rear): ");
   theList.displayList();
   }
//--------------------------------------------------------------
 } // fim da classe LinkQueue
////////////////////////////////////////////////////////////////
class LinkQueueApp
 {
 public static void main(String[ ] args)
   {
   LinkQueue theQueue = new LinkQueue();
   theQueue.insert(20);                // inserir itens
   theQueue.insert(40);

   theQueue.displayQueue();            // exibir fila

   theQueue.insert(60);                // inserir itens
   theQueue.insert(80);
```

Listagem 5.5 O programa linkQueue.java (continuação)

```
    theQueue.displayQueue();              // exibir fila

    theQueue.remove();                    // remover itens
    theQueue.remove();

    theQueue.displayQueue();              // exibir fila
    } // fim de main()
//////////////////////////////////////////////////////////////
```

O programa cria uma fila, insere dois itens, insere mais dois itens e remove dois itens; depois de cada uma dessas operações a fila é exibida. Eis a saída:

```
Queue  (front->rear):  20  40
Queue  (front->rear):  20  40  60  80
Queue  (front->rear):  60  80
```

Aqui, os métodos insert() e remove() na classe LinkQueue são implementados pelos métodos insertLast() e deleteFirst() da classe FirstLastList. Substituímos o vetor usado para implementar a fila no programa queue.java (Listagem 4.4) do Capítulo 4 pela lista encadeada.

Os programas linkStack.java e linkQueue.java enfatizam que as pilhas e as filas são entidades conceituais, separadas de suas implementações. Uma pilha pode ser implementada igualmente bem por um vetor ou por uma lista encadeada. O que é importante sobre uma pilha são as operações push() e pop() e como elas são usadas; não é o mecanismo subjacente usando para implementar essas operações.

Quando você usaria uma lista encadeada em oposição a um vetor como a implementação de uma pilha ou fila? Uma consideração é quão precisamente você poderá prever a quantidade de dados que a pilha ou fila precisará manter. Se isso não estiver claro, a lista encadeada fornecerá mais flexibilidade que um vetor. Ambas são rápidas, portanto, a velocidade provavelmente não é uma consideração maior.

Tipos de dados e abstração

De onde vem o termo *Tipo Abstrato de Dados*? Vejamos a parte *tipo de dados* primeiro e então *voltaremos* para *abstrato*.

Tipos de dados

A frase *tipo de dados* cobre muita coisa. Primeiramente, era aplicada aos tipos predefinidos como int e double. É provavelmente o que você pensa em primeiro lugar quando ouve o termo.

Quando se fala sobre um tipo primitivo, na verdade está se referindo a duas coisas: um item de dados com certas características e operações permitidas nesses dados. Por exemplo, variáveis do tipo int em Java podem ter valores inteiros entre –2.147.483.648 e +2.147.483.647

e os operadores +, -, *, /, etc. podem ser aplicados a eles. As operações permitidas ao tipo de dados são uma parte inseparável de sua identidade; compreender o tipo significa compreender quais operações podem ser executadas nele.

Com o advento da programação orientada a objetos, agora você pode criar seus próprios tipos de dados usando classes. Alguns desses tipos de dados representam quantidades numéricas que são usadas de maneiras parecidas com os tipos primitivos. Você pode, por exemplo, definir uma classe para tempo (com campos para horas, minutos, segundos), uma classe para frações (com campos numerador e denominador) e uma classe para números extra longos (caracteres em uma cadeia representam os dígitos). Todas essas classes podem ser adicionadas e subtraídas como int e double, exceto que em Java você tem que usar métodos com uma notação funcional como add() e sub(), ao invés de operadores como + e -.

A frase *tipos de dados* parece se adequar naturalmente a tais classes baseadas em quantidade. Porém, também é aplicada às classes que não têm esse aspecto quantitativo. Na verdade, *qualquer* classe representa um tipo de dados, no sentido de que uma classe é composta por dados (campos) e operações permitidas nesses dados (métodos).

Por extensão, quando uma estrutura de armazenamento de dados como uma pilha ou fila é representada por uma classe, ela também pode ser referida como um tipo de dados. Uma pilha é diferente em muitas maneiras de um int, mas ambos são definidos como uma certa organização de dados e um conjunto de operações nesses dados.

Abstração

A palavra *abstrato* significa "considerado à parte de especificações detalhadas ou implementação". Uma abstração é a essência ou características importantes de algo. O escritório de presidente, por exemplo, é uma abstração, considerada à parte do indivíduo que o ocupa. As capacidades e as responsabilidades do escritório permanecem iguais, ao passo que os ocupantes individuais do escritório vêm e vão.

Então, na programação orientada a objetos, um Tipo Abstrato de Dado é uma classe considerada sem relação com sua implementação. É uma descrição dos dados na classe (campos), uma lista de operações (métodos) que podem ser executadas nesses dados e instruções sobre como usar essas operações. Especificamente excluídos estão os detalhes de como os métodos executam suas tarefas. Como usuário da classe, você é informado sobre quais métodos usar, como chamá-los e os resultados que pode esperar, mas não como eles funcionam.

O significado de *Tipo Abstrato de Dado* é mais estendido quando aplicado a estruturas de dados como pilhas e filas. Como em qualquer classe, significa os dados e as operações que podem ser executadas neles, mas neste contexto até os fundamentos de como os dados são armazenados tornam-se invisíveis para o usuário. O usuário não apenas não sabe como os métodos funcionam, como também não conhece qual estrutura é usada para armazenar os dados.

Para a pilha, o usuário sabe que push() e pop() (e talvez alguns outros métodos) existem e como funcionam. O usuário não precisa saber (pelo menos não geralmente) como push() e pop() funcionam ou se os dados são armazenados em um vetor, lista encadeada ou alguma outra estrutura de dados como uma árvore.

A interface

Uma especificação TAD é geralmente chamada de *interface*. É o que o usuário da classe vê – em geral seus métodos públicos. Em uma classe pilha, push() e pop() e métodos semelhantes formam a interface.

Listas TAD

Agora que sabemos o que é um Tipo Abstrato de Dado, podemos mencionar outro: a lista. Uma lista (algumas vezes chamada de lista linear) é um grupo de itens organizados em uma ordem linear. Isto é, eles são alinhados de uma certa maneira, como as pérolas em um colar ou as casas em uma rua. Listas suportam certas operações fundamentais. Você pode inserir um item, eliminar um ittem e geralmente ler um item de um local especificado (o terceiro item, digamos).

Não confunda a lista TAD com a lista encadeada que analisamos neste capítulo. Uma lista é definida por sua interface: os métodos específicos usados para interagir com ela. Essa interface pode ser implementada por várias estruturas, inclusive vetores e listas encadeadas. A lista é uma abstração de tais estruturas de dados.

TAD como uma ferramenta de projeto

O conceito TAD é uma ajuda *útil* no processo de projeto de software. Se você precisar armazenar dados, comece considerando as operações que precisam ser executadas neles. Você precisa acessar o último item inserido? O primeiro? Um item com uma chave especificada? Um item em uma certa posição? Responder tais perguntas levará à definição de um TAD. Só depois do TAD estar completamente definido, você deve se preocupar com os detalhes de como representar os dados e como codificar os métodos que acessam os dados.

Separando a especificação do TAD dos detalhes de implementação, você poderá simplificar o processo de projeto. Também facilitará mudar a implementação em algum momento no futuro. Se um usuário se relacionar apenas à interface TAD, você deverá ser capaz de mudar a implementação sem "quebrar" o código do usuário.

Naturalmente, assim que o TAD tiver sido construído, a estrutura de dados subjacente terá que ser escolhida com cuidado para tornar as operações especificadas o mais eficiente possível. Se precisar de um acesso aleatório para o elemento N, por exemplo, a representação de lista encadeada não será boa, porque o acesso aleatório não é uma operação eficiente para uma lista encadeada. Seria melhor com um vetor.

Nota

Lembre-se que o conceito TAD é apenas uma ferramenta conceitual. As estruturas de armazenamento de dados não são divididas claramente em algo como que são TADs e algumas que são usadas para implementar TADs. Uma lista encadeada, por exemplo, não precisa ser integrada em uma interface de lista para ser útil; ela pode agir como um TAD por si só ou pode ser usada para implementar outro tipo de dados como uma fila. Uma lista encadeada pode ser

implementada usando um vetor e uma estrutura do tipo vetor pode ser implementada usando uma lista encadeada. O que é um TAD e o que é uma estrutura mais básica tem que ser determinado em um dado contexto.

Listas ordenadas

Nas listas encadeadas vistas até então, não houve nenhuma exigência para os dados serem armazenados em ordem. Porém, para certas aplicações será útil manter os dados em ordem classificada na lista. Uma lista com essa característica é chamada de *lista ordenada*.

Em uma lista ordenada, os itens são organizados na ordem classificada pelo valor-chave. A eliminação está geralmente limitada ao menor item (ou maior) na lista, que está no início da lista, embora algumas vezes métodos find() e delete(), que buscam a lista para obter nós específicos, sejam usados também.

Em geral, você pode usar uma lista ordenada na maioria das situações nas quais usa um vetor ordenado. As vantagens de uma lista ordenada sobre um vetor ordenado são a velocidade da inserção (porque os elementos não precisam ser movidos) e o fato de que uma lista pode se expandir para preencher a memória disponível, ao passo que um vetor está limitado a um tamanho fixo. Contudo, uma lista ordenada é de algum modo mais difícil de implementar que um vetor ordenado.

Posteriormente, veremos uma aplicação para listas ordenadas: ordenar dados. Uma lista ordenada pode também ser usada para implementar uma fila de prioridade, embora um heap (veja Capítulo 12, "Heaps") seja uma implementação mais comum.

O applet LinkList Workshop apresentado no início deste capítulo demonstra listas ordenadas, assim como não ordenadas. Para ver como listas ordenadas funcionam, use o botão New (Novo) para criar uma nova lista com cerca de 20 nós e quando solicitado, clique o botão Sorted (Ordenado). O resultado será uma lista com dados em ordem classificada, como mostrado na Figura 5.11.

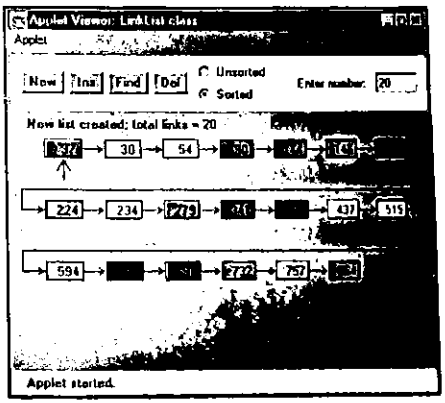

Figura 5.11 O applet LinkList Workshop com uma lista ordenada.

Use o botão Ins para inserir um novo item. Digite um valor que fique em algum lugar no meio da lista. Observe como o algoritmo percorre os nós, procurando o devido lugar da inserção. Quando encontrar o local correto, ele irá inserir o novo nó, como mostrado na Figura 5.12.

Com o próximo pressionar de Ins, a lista será redesenhada para regularizar sua aparência. Você poderá também encontrar um nó específico usando o botão Find e apagar um nó específico usando o botão Del.

Código Java para inserir um item em uma lista ordenada

Para inserir um item em uma lista ordenada, o algoritmo terá primeiro que buscar a lista até localizar o devido lugar para colocar o item: é logo antes do primeiro item que é maior, como mostrado na Figura 5.12.

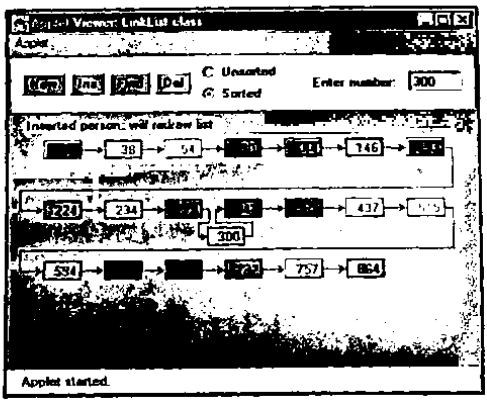

Figura 5.12 Uma ligação recém-inserida.

Quando o algoritmo encontrar onde colocá-lo, o item poderá ser inserido da maneira usual alterando next no novo nó para apontar para o próximo nó e alterando next no nó anterior para apontar para o novo nó. Porém, precisamos considerar alguns casos especiais: o nó poderia precisar ser inserido no início da lista ou poderia precisar ir para o final. Vejamos o código:

```
public void insert(long key)          // insere em ordem
  {
  Link newLink = new Link(key);       // cria novo nó
  Link previous = null;               // começa no primeiro
  Link current = first;
                                      // vai até o final da lista,
  while(current != null && key > current.dData)
    {                                 // ou até key > current,
    previous = current;
```

```
 current = current.next;                  // vai para o próximo item
 }
if(previous==null)                        // no início da lista
 first = newLink;                         // first -> newLink
else                                      // não no início
 previous.next = newLink;                 // old prev -> newLink
newLink.next = current;                   // newLink -> old current
} // fim de insert()
```

Precisamos manter uma referência previous quando nos movemos, portanto, podemos modificar o campo next do nó anterior para apontar para o novo nó. Depois de criar o novo nó, preparamo-nos para buscar o ponto de inserção definindo current para first da maneira usual. Também definimos previous para null; esta etapa é importante porque mais tarde usaremos esse valor null para determinar se ainda estamos no início da lista.

O laço while é similar àqueles usados antes para buscar o ponto de inserção, mas há uma condição adicionada. O laço termina quando a chave do nó atualmente sendo examinado (current.dData) não é mais menor que a chave do nó sendo inserido (key); é o caso mais usual, onde uma chave é inserida em algum lugar no meio da lista.

Porém, o laço while também terminará se current for null. Isso acontecerá no final da lista (o campo next do último elemento é null) ou se a lista estiver vazia para começar (first é null).

Então, quando o laço while termina podemos estar no início, no meio ou no final da lista, ou a lista pode estar vazia.

Se estivermos no início ou a lista estiver vazia, previous será null; portanto, definimos first para o novo nó. Do contrário, estaremos no meio da lista ou no final e definiremos previous.next para o novo nó.

Em qualquer caso, definimos o campo next do novo nó para current. Se estivermos no final da lista, current será null, portanto, o campo next do novo nó será definido adequadamente para esse valor.

O programa sortedList.java

O exemplo sortedList.java mostrado na Listagem 5.6 apresenta uma classe SortedList com os métodos insert(), remove() e displayList(). Apenas a rotina insert() é diferente de sua correspondente nas listas não ordenadas.

Listagem 5.6 O programa sortedList.java

```
// sortedList.java
// demonstrates sorted list
// to run this program: C>java SortedListApp
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                    // item de dado
  public Link next;                     // próximo nó na lista
//-------------------------------------------------------------
```

Listagem 5.6 O programa sortedList.java (continuação)

```
  public Link(long dd)                    // construtor
    { dData = dd; }
 //-----------------------------------------------------------------
  public void displayLink()               // exibe este nó
    { System.out.print(dData + " "); }
  } // fim da classe Link
////////////////////////////////////////////////////////////////
class SortedList
  {
  private Link first;                     // ref. ao primeiro item da lista
 //-----------------------------------------------------------------
  public SortedList()                     // construtor
    { first = null; }
 //-----------------------------------------------------------------
  public boolean isEmpty()                // verdadeiro, se não há nós
    { return (first= =null); }
 //-----------------------------------------------------------------
  public void insert(long key)            // insere, em ordem
    {
    Link newLink = new Link(key);         // cria novo nó
    Link previous = null;                 // começa no primeiro
    Link current = first;
                                          // até o final da lista,
    while(current != null && key > current.dData)
      {                                   // ou key > current,
      previous = current;
      current = current.next;             // vai p/ o próximo item
      }
    if(previous= =null)                   // no início da lista
      first = newLink;                    // first -> newLink
    else                                  // não é início
      previous.next = newLink;            //antigo prev -> newLink
    newLink.next = current;               //newLink -> antigo current
    } // fim de insert()
 //-----------------------------------------------------------------
  public Link remove()                    //retorna & elimina primeiro nó
    {                                     // (assume lista não vazia)
    Link temp = first;                    // salva first
    first = first.next;                   // elimina first
    return temp;                          // retorna valor
    }
 //-----------------------------------------------------------------
  public void displayList()
    {
    System.out.print("List (first->last): ");
    Link current = first;                 // começa no início da lista
    while(current != null)                // até o final da lista,
      {
      current.displayLink();              // imprime dado
      current = current.next;             // move p/ o próximo nó
      }
    System.out.println("");
    }
  } // fim da classe SortedList
////////////////////////////////////////////////////////////////
```

Listagem 5.6 O programa sortedList.java (continuação)

```
class SortedListApp
{
public static void main(String[ ] args)
  {                                    // criar nova lista
  SortedList theSortedList = new SortedList();
  theSortedList.insert(20);            // inserir 2 itens
  theSortedList.insert(40);

  theSortedList.displayList();         // exibir lista

  theSortedList.insert(10);            // inserir mais 3 itens
  theSortedList.insert(30);
  theSortedList.insert(50);

  theSortedList.displayList();         // exibir lista

  theSortedList.remove();              // remover um item

  theSortedList.displayList();         // exibir lista
  } // fim de main()
} // fim da classe SortedListApp
////////////////////////////////////////////////////////////////
```

Em main(), inserimos dois itens com os valores-chaves 20 e 40. Então, inserimos mais três itens, com os valores 10, 30 e 50. Esses valores são inseridos no início da lista, no meio e no final, mostrando que a rotina insert() lida corretamente com esses casos especiais. Finalmente, removemos um item, para mostrar que a remoção é sempre a partir da frente da lista. Depois de cada alteração, a lista é exibida. Eis a saída de sortedList.java:

```
List (first->last): 20 40
List (first->last): 10 20 30 40 50
List (first->last): 20 30 40 50
```

Eficiência das listas encadeadas ordenadas

Inserção e eliminação de itens arbitrários na lista encadeada ordenada requer O(N) comparações (N/2 em média), porque o devido local tem que ser encontrado percorrendo a lista. Porém, o valor mínimo pode ser encontrado ou eliminado, em tempo O(1) porque ele está no início da lista. Se uma aplicação acessar com freqüência o item mínimo e uma inserção rápida não for crítica, então uma lista encadeada ordenada será uma opção eficiente. Uma fila de prioridade poderia ser implementada por uma lista encadeada ordenada, por exemplo.

Ordenação por inserção na lista

Uma lista ordenada pode ser usada como um mecanismo de ordenação bem eficiente. Suponha que você tenha um vetor de itens de dados não ordenados. Se você tirar os itens do vetor e inseri-los um por um na lista ordenada, eles serão colocados em ordem

classificada automaticamente. Se então removê-los da lista e colocá-los de volta no vetor, o vetor estará ordenado.

Esse tipo de ordenação acaba sendo substancialmente mais eficiente do que a mais usual ordenação por inserção em um vetor, descrita no Capítulo 3, "Ordenação Simples", porque menos cópias são necessárias. É ainda um processo $O(N^2)$ porque inserir cada item na lista ordenada envolve comparar um novo item com uma média de metade dos itens já na lista e há N itens a inserir, resultando em cerca de $N^2/4$ comparações. Porém, cada item é copiado apenas duas vezes: uma vez do vetor para a lista e outra vez da lista para o vetor. N^2 cópias comparam-se favoravelmente com a ordenação por inserção em um vetor, onde há cerca de N^2 cópias.

A Listagem 5.7 mostra o programa listInsertionSort.java, que começa com um vetor de itens não ordenados do tipo *link*, insere-os em uma lista ordenada (usando um construtor), então os remove e os coloca de volta no vetor.

Listagem 5.7 O programa listInsertionSort.java

```
// listInsertionSort.java
// demonstra lists ordenada usada para ordenar
// para executar este programa: C>java ListIsertionSortApp
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                    // item de dado
  public Link next;                     // próximo nó na lista
//--------------------------------------------------------------
  public Link(long dd)                  // construtor
    { dData = dd; }
//--------------------------------------------------------------
  }   // fim da classe Link
////////////////////////////////////////////////////////////////
class SortedList
  {
  private Link first;                   // ref. ao primeiro item na lista
//--------------------------------------------------------------
  public SortedList()                   // construtor (sem argumentos)
    { first = null; }                       // inicializa lista
//--------------------------------------------------------------
  public SortedList(Link[ ] linkArr)    // construtor (vetor
    {                                   // como argumento)
    first = null;                           // inicializa lista
    for(int j=0; j<linkArr.length; j++)     // copia vetor
      insert( linkArr[j] );                 // para lista
    }
//--------------------------------------------------------------
  public void insert(Link k)            // insere (em ordem)
    {
    Link previous = null;                   // começa no primeiro
    Link current = first;
                                            // até o final da lista,
    while(current != null && k.dData > current.dData)
      {                                     // ou key > current,
      previous = current;
```

Listagem 5.7 O programa listInsertionSort.java (continuação)

```
    current = current.next;                    // vai p/ próximo item
    }
  if(previous==null)                           // no início da lista
    first = k;                                 // first -> k
  else                                         // não no início
    previous.next = k;                         // antigo prev ->
  k.next = current;                            // k -> antigo current
  } fim de insert()
//--------------------------------------------------------------
 public Link remove()                          // retorna & elimina primeiro nó
  {                                            // (assume lista não vazia)
  long temp = first;                           // salva first
  first = first.next;                          // elimina first
  return temp;                                 // retorna valor
  }
//--------------------------------------------------------------
 } // fim da classe SortedList
////////////////////////////////////////////////////////////////
class ListInsertionSortApp
 {
 public static void main(String[ ] args)
  {
  int size = 10;
                                               // criar vetor de links
  Link[ ] linkArray = new Link[size];

  for(int j=0; j<size; j++)                    // preencher vetor com links
    {                                          // número aleatório
    int n = (int)(java.lang.Math.random()*99);
    Link newLink = new Link(n);                // criar link
    linkArray[j] = newLink;                    // colocar no vetor
    }
                                               // exibir conteúdo do vetor
  System.out.print("Unsorted array: ");
  for(int j=0; j<size; j++)
    System.out.print( linkArray[j].dData + " " );
  System.out.println("");
                                               // criar nova lista
                                               // inicializada com vetor
  SortedList theSortedList = new SortedList(linkArray);

  for(int j=0; j<size; j++)                    // links da lista para vetor
    linkArray[j] = theSortedList.remove();
                                               // exibir conteúdo do vetor
  System.out.print("Sorted Array: ");
  for(int j=0; j<size; j++)
    System.out.print(linkArray[j].dData + " ");
  System.out.println("");
  } fim de main()
 } // fim da classe ListInsertionSortApp
////////////////////////////////////////////////////////////////
```

Este programa exibe os valores no vetor antes da operação de ordenação e novamente depois. Eis algum exemplo de saída:

```
Unsorted  array:  59 69 41 56 84 15 86 81 37 35
Sorted  array:    15 35 37 41 56 59 69 81 84 86
```

A saída será diferente sempre porque os valores iniciais são gerados aleatoriamente.

Um novo construtor para SortedList obtém um vetor de objetos Link como um argumento e insere o conteúdo inteiro desse vetor na lista recém-criada. Fazendo isso, ele ajuda a tornar as coisas mais fáceis para o cliente (a rotina main()).

Também fizemos uma alteração na rotina insert() desse programa. Ela agora aceita um objeto Link como um argumento, ao invés de um long. Fazemos isso para poder armazenar objetos Link no vetor e inseri-los diretamente na lista. No programa sortedList.java (Listagem 5.6), foi mais conveniente fazer com que a rotina insert() criasse cada objeto Link, usando o valor long passado como um argumento.

A desvantagem da ordenação por inserção na lista, comparada com uma ordenação por inserção baseada em vetor, é que ela ocupa mais ou menos duas vezes a memória: o vetor e a lista encadeada têm que estar na memória ao mesmo tempo. Porém, se você tiver uma classe da lista encadeada ordenada à mão, a ordenação por inserção na lista será uma maneira conveniente de ordenar vetores que não sejam grandes demais.

Listas duplamente encadeadas

Examinemos outra variação na lista encadeada: a lista *duplamente* encadeada (não confundir com a lista com extremidades duplas). Qual é a vantagem de uma lista duplamente encadeada? Um problema potencial com as listas encadeadas comuns é que é difícil percorrê-la de volta. Uma instrução como:

```
current=current.next
```

vai de modo conveniente para o próximo nó, mas não há maneira correspondente de ir para o nó anterior. Dependendo da aplicação, essa limitação poderia trazer problemas.

Por exemplo, imagine um editor de texto no qual uma lista encadeada é usada para armazenar o texto. Cada linha de texto na tela é armazenada como um objeto String incorporado em um nó. Quando o usuário do editor move o cursor para baixo na tela, o programa vai para o próximo nó para manipular ou exibir a nova linha. Mas o que acontecerá se o usuário mover o cursor para cima? Em uma lista encadeada comum, você precisaria voltar current (ou seu equivalente) para o início da lista e então percorrer todo o caminho de novo até o novo nó atual. Isto não é muito eficiente. Você deseja dar um único passo para cima.

A lista duplamente encadeada fornece essa capacidade. Ela permite percorrer a lista para trás, assim como para frente. O segredo é que cada nó tem duas referências para outros nós, ao invés de uma. A primeira é com o próximo nó, como nas listas comuns. A segunda é com o nó anterior. Esse tipo de lista é mostrado na Figura 5.13.

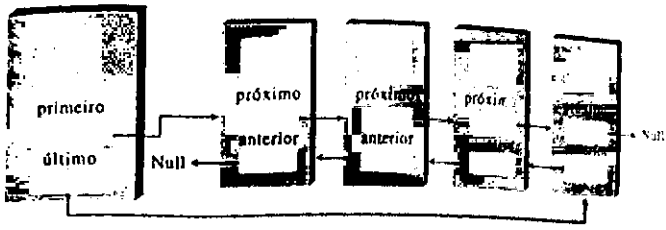

Figura 5.13 Uma lista duplamente encadeada.

O início da especificação para a classe Link em uma lista duplamente encadeada fica assim:

```
class Link
  {
  public long dData;              // item de dado
  public Link next;               // próximo nó na lista
  public Link previous;           // nó anterior na lista
  ...
  }
```

A desvantagem das listas duplamente encadeadas é que toda vez que você insere ou elimina um nó, tem que lidar com quatro ligações, ao invés de duas: duas conexões com o nó anterior e duas conexões com o seguinte. E claro, cada nó é um pouco maior por causa da referência extra.

Uma lista duplamente encadeada não precisa necessariamente ser uma lista com extremidades duplas (mantendo uma referência para o último elemento na lista), mas criá-la assim é útil, portanto, iremos incluí-la em nosso exemplo.

Mostraremos a listagem completa para o programa doublyLinked.java em breve, mas primeiro iremos examinar alguns métodos em sua classe doublyLinkedList.

Travessia

Dois métodos de exibição demonstram a travessia de uma lista duplamente encadeada. O método displayForward() é igual ao método displayList() visto nas listas encadeadas comuns. O método displayBackward() é similar, mas começa no último elemento da lista e avança em direção ao início da lista, indo para o campo previous de cada elemento. Este fragmento de código mostra como o processo funciona:

```
Link current = last;                  // começa no fim
while(current != null)                // até o início da lista,
  current = current.previous;         // move para o nó anterior
```

Ocasionalmente, algumas pessoas têm a opinião que, como você pode ir para qualquer lado igualmente com facilidade em uma lista duplamente encadeada, não há nenhuma direção preferida e, portanto, termos como previous e next são inadequados. Se preferir, poderá substituir por termos neutros em relação a direção como left e right.

Inserção

Incluímos várias rotinas de inserção na classe DoublyLinkedList. O método insertFirst() insere no início da lista, insertLast() insere no final e insertAfter() insere depois de um elemento com uma chave especificada.

A menos que a lista esteja vazia, a rotina insertFirst() mudará o campo previous no antigo primeiro nó para apontar para o novo nó e mudará o campo next no novo nó para apontar para o antigo primeiro nó. Finalmente, definirá first para apontar para o novo nó. Este processo é mostrado na Figura 5.14.

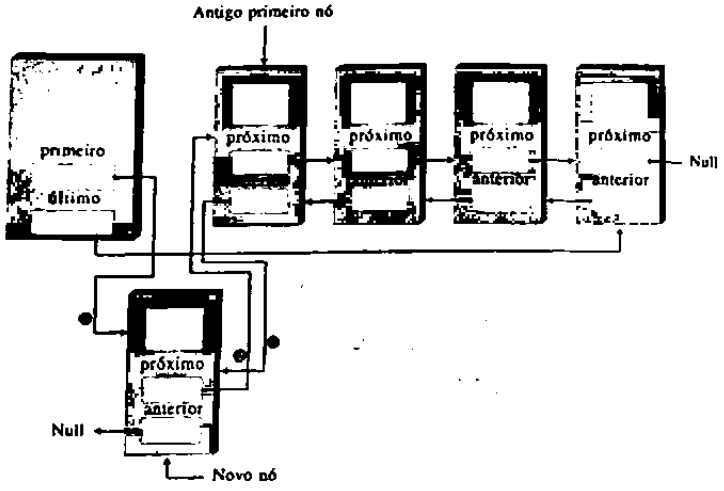

Figura 5.14 Inserção no início.

Se a lista estiver vazia, o campo last terá que ser alterado, ao invés do campo first.previous. Eis o código:

```
if( isEmpty() )                 // se lista vazia,
  last = newLink;               // newLink <- last
else
  first.previous = newLink;     // newLink <- antigo first
newLink.next = first;           // newLink -> antigo first
first = newLink;                // first -> newLink
```

O método insertLast() é o mesmo processo aplicado no final da lista; é uma imagem espelhada de insertFirst().

O método insertAfter() insere um novo nó depois do nó com um valor-chave especificado. É um pouco mais complicado porque quatro conexões têm que ser feitas. Primeiro, o nó com o valor-chave especificado tem que ser encontrado. Esse procedimento é lidado do mesmo

modo que a rotina find() no programa linkList2.java (Listagem 5.2). Então, supondo que não estamos no final da lista, duas conexões têm que ser feitas entre o novo nó e o próximo, e mais duas entre current e o novo nó. Este processo é mostrado na Figura 5.15.

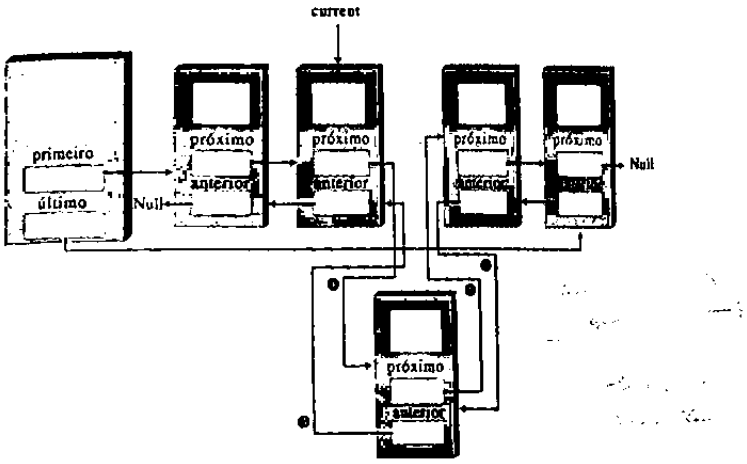

Figura 5.15 Inserção em um local arbitrário.

Se o novo nó for inserido no final da lista, seu campo next terá que apontar para null e last terá que apontar para o novo nó. Eis o código insertAfter() que lida com os nós:

```
if(current==  =last)                    // se último nó,
  {
  newLink.next  =  null;                // newLink -> null
  last  =  newLink;                     // newLink <- last
  }
else                                    // não é o último nó,
  {
  newLink.next  =  current.next;        // newLink -> antigo next
                                        // newLink <- antigo next
  current.next.previous  =  newLink;
  }
newLink.previous  =  current;           // old current <- newLink
current.next  =  newLink;               // old current -> newLink
```

Talvez, você não esteja familiarizado com o uso de dois operadores de ponto na mesma expressão. É uma extensão natural do operador com um ponto. A expressão:

```
current.next.previous
```

significa o campo previous do nó referido pelo campo next do nó current.

Eliminação

Há três rotinas de eliminação: deleteFirst(), deleteLast() e deleteKey(). As duas primeiras são bem simples. Em deleteKey(), a chave sendo apagada é current. Supondo que o nó a ser eliminado não seja o primeiro nem o último da lista, o campo next de current.previous (o nó antes daquele que está sendo eliminado) é definido para apontar para current.next (o nó depois daquele que está sendo eliminado) e o campo previous de current.next é definido para apontar para current.previous. Isso desconecta o nó atual da lista. A Figura 5.16 mostra como fica essa desconexão e as duas instruções seguintes executam-na:

```
current.previous.next  =  current.next;
current.next.previous  =  current.previous;
```

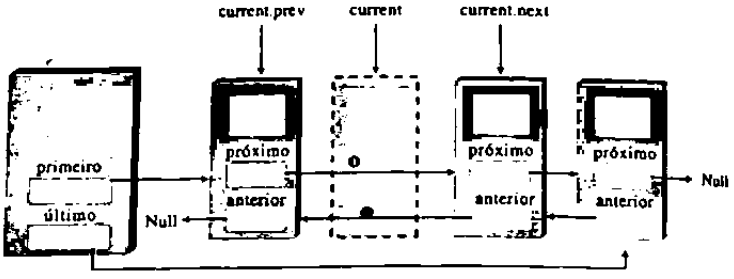

Figura 5.16 Eliminando um nó arbitrário.

Surgirão casos especiais se o nó a ser eliminado for o primeiro ou o último da lista, porque first ou last tem que ser definido para apontar para o próximo nó ou o anterior. Eis o código de deleteKey() para lidar com as conexões do nó:

```
if(current==first)                    // primeiro item?
  first = current.next;               // first -> antigo next
else                                  // não é o primeiro
                                      // antigo previous -> antigo next
  current.previous.next = current.next;

if(current==last)                     // último item?
  last = current.previous;            // antigo previous <- last
else                                  // não é o último
                                      // antigo previous <- antigo next
  current.next.previous = current.previous;
```

O programa doublyLinked.java

A Listagem 5.8 mostra o programa doublyLinked.java completo, que inclui todas as rotinas recém-analisadas.

Listagem 5.8 O programa doublyLinked.java

```
// doublyLinked.java
// demonstra lista duplamente encadeada
// para executar este programa: C>java DoublyLinkedApp
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                    // item de dado
  public Link next;                     // próximo nó na lista
  public Link previous;                 // nó anterior na lista
//-------------------------------------------------------------
  public Link(long d)                   // construtor
    { dData = d; }
//-------------------------------------------------------------
  public void displayLink()             // exibe este nó
    { System.out.print(dData + " "); }
//-------------------------------------------------------------
  }  // fim da classe Link
////////////////////////////////////////////////////////////////
class DoublyLinkedList
  {
  private Link first;                   // ref. ao primeiro item
  private Link last;                    // ref. ao último item
//-------------------------------------------------------------
  public DoublyLinkedList()             // construtor
    {
    first = null;                       // sem item na lista ainda
    last = null;
    }
//-------------------------------------------------------------
  public boolean isEmpty()              // verdadeiro se não há nós
    { return first==null; }
//-------------------------------------------------------------
  public void insertFirst(long dd)      // insere na frente da lista
    {
    Link newLink = new Link(dd);        // cria novo nó

    if( isEmpty() )                     // se lista vazia,
      last = newLink;                   // newLink <- last
    else
      first.previous = newLink;         // newLink <- antigo first
    newLink.next = first;               // newLink -> antigo first
    first = newLink;                    // first -> newLink
    }
//-------------------------------------------------------------
  public void insertLast(long dd)       // insere no final da lista
    {
    Link newLink = new Link(dd);        // cria novo nó
    if( isEmpty() )                     // se lista vazia,
      first = newLink;                  // first <- newLink
    else
      {
      last.next = newLink;              // antigo last <- newLink
      newLink.previous = last;          // antigo last <- newLink
      }
    last = newLink;                     // newLink <- last
```

Listagem 5.8 O programa doublyLinked.java (continuação)

```
    }
//--------------------------------------------------------------
  public  Link  deletePirst()           // elimina primeiro nó
    {                                   // (assume lista não vazia)
    Link temp = first;
    if(first.next  ==  null)            // se somente um item
      last = null;                      // null <- last
    else
      first.next.previous  =  null;     // null <- antigo next
    first  =  first.next;               // first -> antigo next
    return  temp;
    }
//--------------------------------------------------------------
  public  Link  deleteLast()            // elimina ultimo nó
    {                                   // (assume lista não vazia)
    Link temp = last;
    if(first.next  ==  null)            // se somente um item
      first = null;                     // first -> null
    else
      last.previous.next  =  null;      // antigo previous <- null
    last  =  last.previous;             // antigo previous -> last
    return  temp;
    }
//--------------------------------------------------------------
                                        // insere dd logo após key
  public  boolean  insertAfter(long  key,  long  dd)
    {                                   // (assume lista não vazia)
    Link current = first;               // começa no início
    while(current.dData  !=  key)       // até encontrar coincidência,
      {
      current  =  current.next;         // move p/ o próximo nó
      if(current  ==  null)
        return  false;                  // não o encontrou
      }
    Link  newLink  =  new  Link(dd);    // cria novo nó

    if(current==last)                   // se ultimo nó,
      {
      newLink.next  =  null;            // newLink -> null
      last  =  newLink;                 // newLink <- last
      }
    else                                // não é o ultimo nó,
      {
      newLink.next  =  current.next;    // newLink -> antigo next
                                        // newLink <- antigo next
      current.next.previous  =  newLink;
      }
    newLink.previous  =  current;       // antigo current <- newLink
    current.next  =  newLink;           // antigo current -> newLink
    return  true;                       // o encontrou, inseriu
    }
//--------------------------------------------------------------
  public  Link  deleteKey(long  key)    // elimina item c/ chave dada
    {                                   // (assume lista não vazia)
    Link current = first;               // começa no início
```

Listagem 5.8 O programa doublyLinked.java (continuação)

```
  while(current.dData  !=  key)        // até encontrar coincidência,
    {
    current  =  current.next;          // move p/ próximo nó
    if(current  ==  null)
      return  null;                    // não o encontrou
    }
   if(current==first)                  // o encontrou; primeiro item?
    first  =  current.next;            // first - > antigo next
   else                                // not first
                                       // antigo previous -> antigo next
    current.previous.next  =  current.next;

   if(current==  last)                 // último item?
    last  =  current.previous;         // antigo previous <- last
   else                                // não é o último
                                       // antigo previous <- antigo next
    current.next.previous  =  current.previous;
   return  current;                    // retorna valor
   }
//--------------------------------------------------------------------------
  public  void  displayForward()
    {
    System.out.print("List  (first->last):  ");
    Link current  =  first;            // começa no início
    while(current  !=  null)           // até o final da lista,
      {
      current.displayLink();           // exibe dado
      current  =  current.next;        // move p/ o próximo nó
      }
     System.out.println("");
     }
//--------------------------------------------------------------------------
  public  void  displayBackward()
    {
    System.out.print("List  (last->first):  ");
    Link current  =  last;      // começa no fim
    while(current  !=  null) // até o início da lista,
      {
      current.displayLink(); // exibe dado
      current  =  current.previous;// move p/ nó anterior
      }
     System.out.println("");
     }
//--------------------------------------------------------------------------
  } // fim da classe DoublyLinkedList
////////////////////////////////////////////////////////////////
class  DoublyLinkedApp
  {
  public  static  void  main(String[  ]  args)
    {                                  // criar nova lista
    DoublyLinkedList  theList  =  new  DoublyLinkedList();

    theList.insertFirst(22);           // inserir na frente
    theList.insertFirst(44);
    theList.insertFirst(66);
```

Listagem 5.8 O programa doublyLinked.java (continuação)

```
      theList.insertLast(11);          // inserir no final
      theList.insertLast(33);
      theList.insertLast(55);

      theList.displayForward();        // exibir lista para frente
      theList.displayBackward();       // exibir lista para trás

      theList.deleteFirst();           // eliminar primeiro item
      theList.deleteLast();            // eliminar último item
      theList.deleteKey(11);           // eliminar item com chave 11
      theList.displayForward();        // exibir lista para frente

      theList.insertAfter(22, 77);     // inserir 77 depois de 22
      theList.insertAfter(33, 88);     // inserir 88 depois de 33

      theList.insertForward();         // exibir lista para frente
      } fim de main()
   } // fim da classe DoublyLinkedApp
////////////////////////////////////////////////////////////////
```

Em main(), inserimos alguns itens no início da lista e no final, exibimos os itens indo para frente e para trás, eliminamos os primeiro e último itens e o item com a chave 11, exibimos a lista novamente (para frente apenas), inserimos dois itens usando o método insertAfter() e exibimos a lista de novo. Eis a saída:

```
List  (first->last): 66 44 22 11 33 55
List  (last->first): 55 33 11 22 44 66
List  (first->last): 44 22 33
List  (first->last): 44 22 77 33 88
```

Os métodos de eliminação e o método insertAfter() supõem que a lista não esteja vazia. Embora por simplicidade não o tenhamos mostrado em main(), isEmpty() deve ser usado para verificar se há algo na lista antes de tentar estas inserções e eliminações.

Lista duplamente encadeada como base para deques

Uma lista duplamente encadeada pode ser usada como base para uma deque, mencionada no capítulo anterior. Em uma deque, você pode inserir e eliminar em qualquer extremidade e a lista duplamente encadeada fornece essa capacidade.

Iteradores

Vimos como o usuário de uma lista pode encontrar um nó com uma certa chave usando um método find(). O método começa no início da lista e examina cada nó até que encontre um que coincida com a chave de busca. Outras operações vistas, como eliminar um nó especificado ou inserir antes ou depois de um nó especificado, também envolvem buscar

a lista para encontrar o nó especificado. Porém, esses métodos não fornecem ao usuário nenhum controle sobre a travessia até o item especificado.

Suponha que você queira percorrer uma lista, executando alguma operação em certos nós. Por exemplo, imagine um arquivo de funcionários armazenado como uma lista encadeada. Você poderia querer aumentar o salário de todos os funcionários que estavam recebendo um salário mínimo, sem afetar os funcionários já acima do mínimo. Ou suponha que em uma lista de clientes por reembolso postal, você tenha decidido eliminar todos os clientes que não tenham feito algum pedido em seis meses.

Em um vetor, tais operações são fáceis porque você pode usar um índice do vetor para controlar sua posição. Pode operar em um item, então aumentar o índice para apontar para o próximo item e ver se esse item é um candidato adequado para a operação. Porém, em uma lista encadeada, os nós não têm números de índice fixos. Como podemos fornecer a um usuário de uma lista algo análogo ao índice de um vetor? Você poderia usar find() repetidamente para buscar os devidos itens em uma lista, mas essa abordagem requer muitas comparações para encontrar cada nó. É bem mais eficiente ir de nó em nó, verificando se cada uma satisfaz certos critérios e executando a devida operação se ele satisfizer.

Uma referência na lista em si mesma?

Como usuários de uma classe de lista, o que precisamos é acesso a uma referência que possa apontar para qualquer nó arbitrário. Assim, podemos examinar ou modificar o nó. Devemos ser capazes de aumentar a referência para que possamos percorrer a lista, vendo cada nó por vez e devemos ser capazes de acessar o nó apontado pela referência.

Assumindo que criamos tal referência, onde ela será instalada? Uma possibilidade é usar um campo na lista em si mesma, chamado current ou algo assim. Você poderá acessar um nó usando current e incrementar current para ir para o próximo nó.

Um problema com essa abordagem é que você poderia precisar de mais de uma referência, exatamente como você usa em geral vários índices de vetor ao mesmo tempo. Quantas seriam adequadas? Não há maneira de saber quanto o usuário poderia precisar. Assim, parece mais fácil permitir ao usuário criar quantas referências forem necessárias. Para tornar isso possível em uma linguagem orientada a objetos, é natural incorporar cada referência em um objeto de classe. Esse objeto não poderá ser o mesmo que a classe da lista porque há apenas um objeto de lista, portanto, ele é normalmente implementado como uma classe separada.

Uma classe iterador

Objetos contendo referências aos itens em estruturas de dados, usados para percorrer essas estruturas, são comumente chamados de *iteradores* (ou algumas vezes, como em certas classes Java, *enumeradores*). Eis uma idéia preliminar de como eles são:

```
class  ListIterator()
  {
  private  Link  current;
  ...
  }
```

O campo current contém uma referência para o nó para o qual o iterador aponta atualmente. (O termo *aponta* como usado aqui não se refere aos ponteiros em C++; estamos usando-o em seu sentido genérico significando "referir-se a".)

Para usar um tal iterador, o usuário poderá criar uma lista e então criar um objeto iterador associado à lista. Na verdade, como resultado, permitir à lista criar o iterador é mais fácil, pois ela pode passar ao iterador certas informações, como uma referência para seu campo first. Assim, adicionamos um método getIterator() à classe da lista; esse método retorna um objeto iterador adequado ao usuário. Eis algum código abreviado em main() que mostra como o usuário da classe chamaria um iterador:

```
public static void main(...)
  {
  LinkList theList = new LinkList();           // cria lista
  ListIterator iter1 = theList.getIterator(); // cria iterador
  Link aLink = iter1.getCurrent();             // acessa nó em iterador
  iter1.nextLink();                            // move iter. p/ próximo nó
  }
```

Depois de termos criado o objeto iterador, podemos usá-lo para acessar o nó para o qual ele aponta ou incrementá-lo para que ele aponte para o próximo nó, como mostrado nas duas segundas instruções. Podemos chamar o objeto iterador de iter1 para enfatizar que você poderia criar mais iteradores (iter2, etc.) da mesma maneira.

O iterador sempre aponta para algum nó da lista. Ele é associado à lista, mas não é o mesmo que a lista ou o mesmo que um nó. A *Figura* 5.17 mostra dois iteradores apontando para os nós em uma lista.

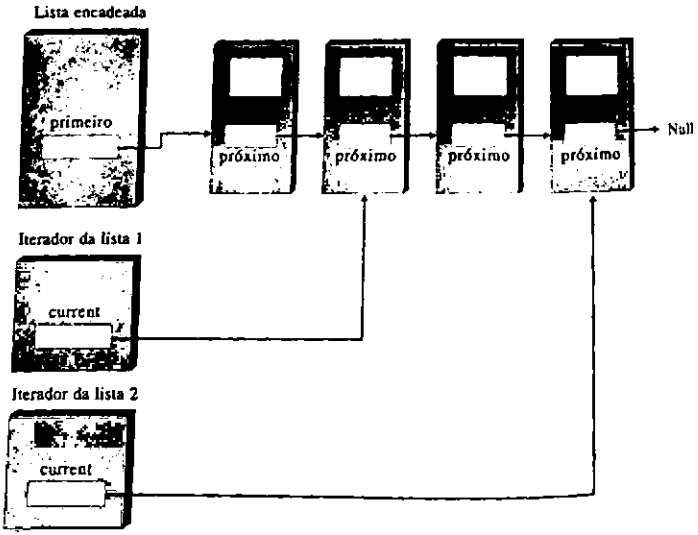

Figura 5.17 Iteradores de lista.

Recursos adicionais de um iterador

Vimos vários programas nos quais o uso de um campo previous torna mais simples executar certas operações, tais como eliminar um nó de um local arbitrário. Tal campo é também útil em um iterador.

Além disso, pode ser que o iterador precise mudar o valor do campo first da lista – por exemplo, se um item for inserido ou eliminado no início da lista. Se o iterador for um objeto de uma classe separada, como ele poderá acessar um campo privado, como first, na lista? Uma solução é a lista passar uma referência de si mesma para o iterador quando ela criar o iterador. Essa referência é armazenada em um campo no iterador.

A lista então terá que fornecer métodos públicos que permitam ao iterador alterar first. Esses métodos de LinkList são getFirst() e setFirst(). (A desvantagem dessa abordagem é que esses métodos permitem que qualquer um altere first, o que introduz um elemento de risco.)

Eis uma classe do iterador revisada (embora ainda incompleta) que incorpora esses campos adicionais, junto com os métodos reset() e nextLink():

```
class ListIterator()
  {
  private Link current;          // reference ao nó atual
  private Link previous;         // reference ao nó anterior
  private LinkList ourList;      // reference à lista "pai"
  public void reset()            // vai para o inícios da lista
    {
    current = ourList.getFirst(); // current -> first
    previous = null;              // previous -> null
    }
  public void nextLink()         // vai para o próximo nó
    {
    previous = current;          // faz anterior ser este
    current = current.next;      // faz este ser o próximo
    }
  ...
  }
```

Podemos notar, para vocês antigos programadores C++, que em C++ a conexão entre o iterador e a lista é geralmente fornecida tornando a classe do iterador *amiga* da classe da lista. Porém, Java não tem classes amigas, que são controversas em qualquer caso, porque são uma abertura na armadura da ocultação de dados.

Métodos do iterador

Métodos adicionais podem tornar o iterador uma classe flexível e poderosa. Todas as operações executadas anteriormente pela classe que envolvem fazer uma iteração na lista, tais como insertAfter(), são executadas mais naturalmente pelo iterador. Em nosso exemplo, o iterador inclui os seguintes métodos:

- reset() – Leva o iterador para o início da lista
- nextLink() – Move o iterador para o próximo nó

- getCurrent() – Retorna o nó no iterador
- atEnd() – Retorna true se o iterador estiver no final da lista
- insertAfter() – Insere um novo nó depois do iterador
- insertBefore() – Insere um novo nó antes do iterador
- deleteCurrent() – Elimina o nó no iterador

O usuário pode posicionar o iterador usando reset() e nextLink(), verificar se está no final da lista com atEnd() e executar as outras operações mostradas.

Decidir quais tarefas devem ser executadas por um iterador e quais pela lista em si mesma nem sempre é fácil. Um método insertBefore() funciona melhor no iterador, mas uma rotina insertFirst() que sempre insere no início da lista poderia ser mais adequada na classe da lista. Mantivemos uma rotina displayList() na lista, mas essa operação poderia também ser lidada com as chamadas a getCurrent() e a nextLink() para o iterador.

O programa interIterator.java

O programa interIterator.java inclui uma interface interativa que permite ao usuário controlar o iterador diretamente. Depois de você ter iniciado o programa, poderá executar as seguintes ações digitando a devida letra:

- s – Mostra o conteúdo da lista
- r – Redefine o iterador para o início da lista
- n – Vai para o próximo nó
- g – Obtém o conteúdo do nó atual
- b – Insere antes do nó atual
- a – Insere um novo nó depois do nó atual
- d – Elimina o nó atual

A Listagem 5.9 mostra o programa interIterator.java completo.

Listagem 5.9 O programa interIterator.java

```
// interIterator.java
// demonstra iteradores em uma listListIterator encadeada
// para executar este programa: C>java InterIterApp
import java.io.*;          // para E/S
////////////////////////////////////////////////////////////////class
Link
  {
  public long dData;                    // item de dado
  public Link next;                     // próximo nó na lista
//-------------------------------------------------------------
  public Link(long dd)                  // construtor
    { dData = dd; }
//-------------------------------------------------------------
  public void displayLink()             // exibe a nos mesmos
    { System.out.print(dData + " "); }
  }   // fim da classe Link
```

Listagem 5.9 O programa interIterator.java (continuação)

```
////////////////////////////////////////////////////////////////
class LinkList
  {
  private Link first;                    // ref. ao primeiro item na lista
//--------------------------------------------------------------
  public LinkList()                      // construtor
    { first = null; }                    // sem itens na lista ainda
//--------------------------------------------------------------
  public Link getFirst()                 // obtém valor do primeiro
    { return first; }
//--------------------------------------------------------------
  public void setFirst(Link f)           // faz primeiro ser um novo nó
    { first = f; }
//--------------------------------------------------------------
  public boolean isEmpty()               // verdadeiro, se lista vazia
    { return first==null; }
//--------------------------------------------------------------
  public ListIterator getIterator()      // retorna iterador
    {
    return new ListIterator(this);       // inicializado com
    }                                    // esta lista
//--------------------------------------------------------------
  public void displayList()
    {
    Link current = first;                // começa no início da lista
    while(current != null)               // até o fim da lista,
      {
      current.displayLink();             // imprime dado
      current = current.next;            // move p/ próximo nó
      }
    System.out.println("");
    }
//--------------------------------------------------------------
  } // fim da classe LinkList
////////////////////////////////////////////////////////////////
class ListIterator
  {
  private Link current;                  // nó atual
  private link previous;                 // nó anterior
  private LinkList ourList;              // nossa lista encadeada
//--------------------------------------------------------------
  public ListIterator(LinkList list)     // construtor
    {
    ourList = list;
    reset();
    }
//--------------------------------------------------------------
  public void reset()                    // começa em 'first'
    {
    current = ourList.getFirst();
    previous = null;
    }
//--------------------------------------------------------------
  public boolean atEnd()                 // verdadeiro se último nó
    { return (current.next==null); }
```

Listagem 5.9 O programa interIterator.java (continuação)

```
//--------------------------------------------------------------
  public void nextLink()                    // vai para próximo nó
    {
    previous = current;
    current = current.next;
    }
//--------------------------------------------------------------
  public Link getCurrent()                  // obtém nó atual
    { return current; }
//--------------------------------------------------------------
  public void insertAfter(long dd)          // insere depois
    {                                       // do nó atual
    Link newLink = new Link (dd);

    if( ourList.isEmpty() )                 // lista vazia
      {
      ourList.setFirst(newLink);
      current = newLink;
      }
    else                                    // não vazia
      {
      newLink.next = current.next;
      current.next = newLink;
      nextLink();                           // aponta para o novo nó
      }
    }
//--------------------------------------------------------------
  public void insertBefore(long dd)         // insere antes
    {                                       // do nó atual
    Link newLink = new Link(dd);

    if(previous = = null)                   // início da lista
      {                                     // (ou lista vazia)
      newLink.next = ourList.getFirst();
      ourList.setFirst(newLink);
      reset();
      }
    else                                    // não início
      {
      newLink.next = previous.next;
      previous.next = newLink;
      current = newLink;
      }
    }
//--------------------------------------------------------------
  public long deleteCurrent()               // elimina item em current
    {
    long value = current.dData;
    if(previous = = null)                   // início da lista
      {
      ourList.setFirst(current.next);
      reset();
      }
    else                                    // não início
      {
```

Listagem 5.9 O programa interIterator.java (continuação)

```
    previous.next = current.next;
    if( atEnd() )
      reset();
    else
      current = current.next;
    }
   return value;
   }
//--------------------------------------------------------------
  }  // fim da classe ListIterator
////////////////////////////////////////////////////////////////
class InterIterApp
  {
  public static void main(String[ ] args) throws IOException
    {
    LinkList theList = new LinkList();      // nova lista
    ListIterator iter1 = theList.getIterator();// novo iter.
    long value;

    iter1.insertAfter(20);                  // inserir itens
    iter1.insertAfter(40);
    iter1.insertAfter(80);
    iter1.insertBefore(60);

    while(true)
      {
      System.out.print("Enter first letter of show, reset, ");
      System.out.print("next, get, before, after, delete: ");
      System.out.flush();
      int choice = getChar();               // obter opção do usuário
      switch(choice)
        {
        case 's':
          if( !theList.isEmpty() )
            theList.displayList();
          else
            System.out.println("List is empty");
          break;
        case 'r':                           // retornar (ao primeiro)
          iter1.reset();
          break;
        case 'n':                           // avançar p/ próximo item
          if( !theList.isEmpty() && !iter1.atEnd() )
            iter1.nextLink();
          else
            System.out.println("Can't go to next link");
          break;
        case 'g':                           // obter item atual
          if( !theList.isEmpty() )
            {
            value = iter1.getCurrent().dData;
            System.out.println("Returned " + value);
            }
          else
            System.out.println("List is empty");
```

Listagem 5.9 O programa interIterator.java (continuação)

```
            break;
          case 'b':                              // inserir antes do atual
            System.out.print("Enter value to insert: ");
            System.out.flush();
            value = getInt();
            iter1.insertBefore(value);
            break;
          case 'a':                              // inserir depois do atual
            System.out.print("Enter value to insert: ");
            System.out.flush();
            value = getInt();
            iter1.insertAfter(value);
            break;
          case 'd':                              // eliminar item atual
            if( !theList.isEmpty() )
              {
              value = iter1.deleteCurrent();
              System.out.println("Deleted " + value);
              }
            else
              System.out.println("Can't delete");
            break;
          default:
            System.out.println("Invalid entry");
          } // fim do switch
        } // fim do while
      } fim de main()
//--------------------------------------------------------------
public static String getString() throws IOException
  {
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  String s = br.readLine();
  return s;
  }
//--------------------------------------------------------------
  public static char getChar() throws IOException
    {
    String s = getString();
    return s.charAt(0);
    }
//--------------------------------------------------------------
  public static int getInt() throws IOException
    {
    String s = getString();
    return Integer.parseInt(s);
    }
//--------------------------------------------------------------
  } // fim da classe InterIterApp
////////////////////////////////////////////////////////////////
```

A rotina main() insere quatro itens na lista, usando um iterador e seu método insertAfter(). Então aguarda que o usuário interaja com ela. No exemplo de interação a seguir, o usuário exibe a lista, redefine o iterador para o início, avança dois nós, obtém o valor-chave do nó atual (que é 60), insere 100 antes deste, insere 7 depois de 100 e exibe a lista novamente:

```
Enter first letter of
 show, reset, next, get, before, after, delete: s
20 40 60 80
Enter first letter of
 show, reset, next, get, before, after, delete: r
Enter first letter of
 show, reset, next, get, before, after, delete: n
Enter first letter of
 show, reset, next, get, before, after, delete: n
Enter first letter of
 show, reset, next, get, before, after, delete: g
Returned 60
Enter first letter of
 show, reset, next, get, before, after, delete: b
Enter value to insert: 100
Enter first letter of
 show, reset, next, get, before, after, delete: a
Enter value to insert: 7
Enter first letter of
 show, reset, next, get, before, after, delete: s
20 40 100 7 60 80
```

Experimentar o programa interIterator.java dará uma idéia de como o iterador se move ao longo dos nós e como ele pode inserir e eliminar nós em qualquer lugar da lista.

Para onde o iterador aponta?

Um dos problemas de projeto em uma classe iterador é decidir para onde o iterador deve apontar depois de várias operações.

Quando você elimina um item com deleteCurrent(), o iterador deve acabar apontando para o próximo item, para o item anterior ou de volta para o início da lista? Manter o iterador na vizinhança do item eliminado é conveniente porque há chances de o usuário da classe executar outras operações lá. Porém, você não poderá movê-lo para o item anterior porque não há maneira de redefinir o campo previous da lista para o item anterior. (Precisaria de uma lista duplamente encadeada para essa tarefa.) Nossa solução é mover o iterador para o nó posterior ao nó eliminado. Se acabamos de eliminar o item no final da lista, o iterador será definido para o início da lista.

Depois de chamadas para insertBefore() e insertAfter(), retornamos com current apontando para o item recém-inserido.

O método atEnd()

Há uma outra questão sobre o método atEnd(). Ele poderá retornar true quando o iterador apontar para o último nó válido na lista ou poderá retornar true quando o iterador apontar para *depois* do último nó (e estará assim não apontando para um nó válido).

Com a primeira abordagem, uma condição de laço usada para fazer uma iteração na lista torna-se inadequada porque você precisa executar uma operação no último nó antes de verificar se ele é o último nó (e terminar o laço se ele for).

Contudo, a segunda abordagem não permite descobrir se você está no final da lista até que seja tarde demais para fazer qualquer coisa com o último nó. (Você não poderia procurar o último nó e então eliminá-lo, por exemplo.) Isto é porque quando atEnd() tornou-se true, o iterador não apontava mais para o último nó (ou na verdade qualquer nó válido) e você não pode fazer um "retorno" do iterador em uma lista encadeada simples.

Adotamos a primeira abordagem. Assim, o iterador sempre aponta para um nó válido, embora você tenha que ter cuidado ao escrever um laço que faz uma iteração na lista, como veremos em seguida.

Operações iterativas

Como mencionamos, um iterador permite percorrer a lista, executando operações em certos itens de dados. Eis um fragmento de código que exibe o conteúdo da lista, usando um iterador ao invés do método displayList() da lista:

```
iter1.reset();                          // começa no primeiro
long value = iter1.getCurrent().dData;  // exibe nó
System.out.println(value + " ");
while( !iter1.atEnd() )                 // até o fim,
  {
  iter1.nextLink();                     // vai p/ próximo nó,
  long value = iter1.getCurrent().dData;//  exibe-o
  System.out.println(value + " ");
  }
```

Embora não façamos isso aqui, você deveria verificar com isEmpty() para assegurar-se que a lista não esteja vazia antes de chamar getCurrent().

O seguinte código mostra como você poderia eliminar todos os itens com chaves que são múltiplos de 3. Mostramos apenas a rotina main() revisada; tudo mais é igual como em interIterator.java (Listagem 5.9).

```
class InterIterApp
  {
  public static void main(String[ ] args) throws IOException
    {
    LinkList theList = new LinkList();          // nova lista
    ListIterator iter1 = theList.getIterator();// novo iter.

    iter1.insertAfter(21);                  // inserir nós
    iter1.insertAfter(40);
    iter1.insertAfter(30);
    iter1.insertAfter(7);
    iter1.insertAfter(45);

    theList.displayList();                  // exibir lista

    iter1.reset();                          // começar no primeiro nó
    Link aLink = iter1.getCurrent();        // obtê-lo
    if(aLink.dData % 3 == 0)                // se divisível por 3,
      iter1.deleteCurrent();                // eliminá-lo
```

```
while( !iterl.atEnd() )                 // até o final da lista,
  {
  iterl.nextLink();                     // ir para próximo nó

  aLink = iterl.getCurrent();           // obter nó
  if(aLink.dData % 3 == 0)              // se divisível por 3,
    iterl.deleteCurrent();              //  eliminá-lo
  }
 theList.displayList();                 // exibir lista
 } // fim de main()
} // fim da classe InterIterApp
```

Inserimos cinco nós e exibimos a lista. Então fizemos uma iteração na lista, eliminando os nós com chaves divisíveis por 3 e exibimos a lista novamente. Eis a saída:

```
21 40 30 7 45
40 7
```

De novo, embora não tenhamos mostrado aqui, é importante verificar se a lista está vazia antes de chamar deleteCurrent().

Outros métodos

Você poderia criar outros métodos úteis para a classe ListIterator. Por exemplo, um método find() retornaria um item com um valor-chave especificado, como vimos quando find() é um método da lista. Um método replace() poderia substituir itens que tinham certos valores-chaves com outros itens.

Como é uma lista encadeada simples, você pode fazer uma iteração nela apenas para frente. Se uma lista duplamente encadeada fosse usada, poderia ir para qualquer lado, permitindo operações como a eliminação a partir do final da lista, exatamente como os não iteradores. Essa capacidade provavelmente seria uma conveniência em algumas aplicações.

Resumo

- Uma lista encadeada consiste em um objeto linkedList e vários objetos Link.
- O objeto linkedList contém uma referência, geralmente chamada first, para o primeiro nó da lista.
- Cada objeto Link contém dados e uma referência, geralmente chamada next, para o próximo nó da lista.
- Um valor null em next sinaliza o final da lista.
- Inserir um item no início de uma lista encadeada envolve alterar o campo next do novo nó para apontar para o primeiro nó antigo e alterar first para apontar para o novo item.
- Eliminar um item no início de uma lista envolve definir first para apontar para first.next.
- Para percorrer uma lista encadeada, você inicia em first e então vai de nó em nó, usando o campo next de cada nó para encontrar o próximo nó.

- Um nó com um valor-chave especificado pode ser encontrado percorrendo-se a lista. Assim que encontrado, um item pode ser exibido, eliminado ou operado de várias maneiras.
- Um novo nó pode ser inserido antes ou depois de um nó com um valor-chave especificado, depois de uma travessia para encontrar esse nó.
- Uma lista com extremidades duplas mantém um ponteiro para o último nó da lista, geralmente chamado de last, assim como para o primeiro.
- Uma lista com extremidades duplas permite inserção no final da lista.
- Um Tipo Abstrato de Dado (TAD) é uma classe de armazenamento de dados considerada sem referência para sua implementação.
- Pilhas e filas são TADs. Eles podem ser implementados usando vetores ou listas encadeadas.
- Em uma lista encadeada ordenada, os nós são organizadas na ordem do valor-chave ascendente (ou algumas vezes descendente).
- Inserção em uma lista ordenada leva tempo O(N) porque o ponto de inserção correto tem que ser encontrado. Eliminação do menor nó leva tempo O(1).
- Em uma lista duplamente encadeada, cada nó contém uma referência para o nó anterior, assim como para o próximo nó.
- Uma lista duplamente encadeada permite travessia para trás e eliminação a partir do final da lista.
- Um iterador é uma referência, encapsulada em um objeto de classe, que aponta para um nó em uma lista associada.
- Métodos do iterador permitem ao usuário mover o iterador ao longo da lista e acessar o nó apontado atualmente.
- Um iterador pode ser usado para percorrer uma lista, executando alguma operação nos nós selecionados (ou todas os nós).

Questões

Estas questões têm a intenção de ser um autoteste para os leitores. Respostas podem ser encontradas no Apêndice C.

1. Qual a seguir *não* é verdadeira? Uma referência para um objeto de classe
   a. pode ser usada para acessar métodos públicos no objeto.
   b. tem um tamanho dependente de sua classe.
   c. tem o tipo de dados da classe.
   d. não mantém o objeto propriamente dito.
2. Acesso aos nós em uma lista encadeada é geralmente através do _____ nó.
3. Quando você cria uma referência para um nó em uma lista encadeada, ela
   a. tem que se referir ao primeiro nó.
   b. tem que se referir ao nó apontado por current.

c. tem que se referir ao nó apontado por next.
d. pode referir-se a qualquer nó desejado.

4. Quantas referências você tem que alterar para inserir um nó no meio de uma lista encadeada simples?

5. Quantas referências você tem que alterar para inserir um nó no final de uma lista encadeada simples?

6. No método insertFirst() no programa linkList.java (Listagem 5.1), a instrução newLink.next=first; significa que
   a. o próximo novo nó a ser inserido irá referir-se a first.
   b. first referir-se-á ao novo nó.
   c. o campo next do novo nó referir-se-á ao antigo primeiro nó.
   d. newLink.next referir-se-á ao novo primeiro nó da lista.

7. Assumindo-se que current aponte para o nó próximo ao último *em* uma lista encadeada simples, qual instrução eliminará o último nó da lista?

8. Quando todas as referências para um nó são alteradas para referirem-se a outra coisa, o que acontece com o nó?

9. Uma lista com extremidades duplas
   a. pode ser acessada a partir de qualquer extremidade.
   b. é um nome diferente para lista duplamente encadeada.
   c. tem ponteiros indo tanto para frente como para trás entre os nós.
   d. tem seu primeiro nó conectado ao seu último nó.

10. Um caso especial geralmente ocorre para rotinas de inserção e eliminação quando uma lista é _____.

11. Assumindo-se que uma cópia leve mais tempo que uma comparação, será mais rápido eliminar um item com uma certa chave de uma lista encadeada ou de um vetor não ordenado?

12. Quantas vezes você precisará percorrer uma lista encadeada simples para eliminar o item com a maior chave?

13. Das listas analisadas neste capítulo, qual seria melhor para implementar uma fila?

14. Qual a seguir *não* é verdadeira? Iteradores seriam úteis se você quisesse
   a. fazer uma ordenação por inserção em uma lista encadeada.
   b. inserir um novo nó *no início* de uma lista.
   c. trocar dois nós em locais arbitrários.
   d. eliminar todos os nós com um certo valor-chave.

15. Qual você considera uma melhor opção para implementar uma pilha: uma lista encadeada simples ou um vetor?

Experimentos

Conduzirr estes experimentos ajudará a fornecer uma compreensão sobre os tópicos tratados no capítulo. Nenhuma programação é envolvida.

1. Use o applet LinkList Workshop para executar operações de inserção, localização e eliminação tanto em listas ordenadas quanto em não ordenadas. Para as operações demonstradas por esse applet, há alguma vantagem para a lista ordenada?
2. Modifique main() no programa linkList.java (Listagem 5.1) para que ele insira continuamente nós na lista até que a memória tenha se esgotado. Depois de cada 1.000 itens, faça com que exiba o número de itens inseridos até então. Assim, você poderá aprender aproximadamente quantos nós uma lista pode manter em sua máquina particular. (Naturalmente, o número irá variar, dependendo de quais outros programas estão na memória e de muitos outros fatores.) Não tente esse experimento se ele paralisar a rede de sua instituição.

Projetos de programação

Escrever programas para resolver os Projetos de Programação ajuda a solidificar sua compreensão do material e demonstra como os conceitos do capítulo são aplicados. (Como mencionado na Introdução, instrutores qualificados podem obter soluções completas para os Projetos de Programação no site Web da editora.)

5.1 Implemente uma fila de prioridade baseada em uma lista encadeada ordenada. A operação de remoção na fila de prioridade deve remover o item com a menor chave.

5.2 Implemente uma deque baseada em uma lista duplamente encadeada. (Veja o Projeto de Programação 4.2 no capítulo anterior.) O usuário deve ser capaz de executar as operações normais na deque.

5.3 Uma lista circular é uma lista encadeada na qual o último nó aponta de volta para o primeiro nó. Há muitas maneiras de projetar uma lista circular. Algumas vezes, há um ponteiro para o "início" da lista. Porém, isso torna a lista menos parecida com um círculo real e mais com uma lista comum, que tem seu final conectado ao seu início. Crie uma classe para uma lista circular encadeada simples que não tenha final e início. O único acesso à lista é uma referência simples, current, que possa apontar para qualquer nó na lista. Essa referência pode mover-se ao longo da lista quando necessário. (Veja o Projeto de Programação 5.5 para obter uma situação na qual uma tal lista circular é bem adequada.) Sua lista deve lidar com inserção, busca e eliminação. Você pode achar conveniente se essas operações ocorrerem em um nó no fluxo inferior em relação ao nó apontado por current.(Como o nó no fluxo superior é encadeado de modo simples, você não pode obtê-lo sem percorrer todo o caminho no círculo.) Você deve também ser capaz de exibir a lista (embora precise quebrar o círculo em algum ponto arbitrário para imprimi-la na tela). Um método step() que mova current para o próximo nó poderá ser útil também.

5.4 Implemente uma classe pilha baseada na lista circular do Projeto de Programação 5.3. Este exercício não é muito difícil. (Porém, implementar uma fila pode ser mais difícil, a menos que você torne a lista circular duplamente encadeada.)

5.5 O problema de Josephus é um quebra-cabeça matemático famoso que remonta aos tempos antigos. Há muitas histórias sobre ele. Uma é que Josephus fazia parte de um grupo de judeus que estava para ser capturado pelos romanos. Ao invés de serem escravizados, eles escolheram cometer suicídio. Eles se organizaram em círculo e, começando em uma certa pessoa, começaram a contar no círculo. Cada enésima pessoa tinha que sair do círculo e cometer suicídio. Josephus decidiu que não queria morrer, portanto, organizou as regras para que ele fosse a última pessoa a sair. Se houvesse (digamos) 20 pessoas e ele fosse a sétima pessoa a partir do início do círculo, qual número deveria dizer para eles usarem para contagem? O problema fica muito mais complicado porque o círculo diminui quando a contagem prossegue.

Crie uma aplicação que use uma lista encadeada circular (como no Projeto de Programação 5.3) para modelar esse problema. As entradas são o número de pessoas no círculo, o número usado para a contagem e o número da pessoa onde a contagem começa (geralmente 1). A saída é a lista de pessoas sendo eliminadas. Quando uma pessoa sai do círculo, a contagem começa novamente a partir da pessoa que estava à sua esquerda (supondo que você vai para a direita). Eis um exemplo. Há sete pessoas numeradas de 1 a 7, você começa em 1 e conta em três. As pessoas serão eliminadas na ordem 4, 1, 6, 5, 7, 3. O número 2 ficará.

5.6 Experimentemos algo um pouco diferente: uma lista encadeada bidimensional, que chamaremos de matriz. É uma lista análoga a um vetor bidimensional. Poderia ser útil em aplicações como programas de planilha. Se uma planilha for baseada em um vetor e você inserir uma nova linha perto da parte superior, terá que mover cada célula nas linhas inferiores em N*M células, o que é potencialmente um processo lento. Se a planilha for implementada por uma matriz, você precisará alterar apenas N ponteiros.

Por simplicidade, asumiremos uma abordagem encadeada simples (embora uma abordagem duplamente encadeada provavelmente fosse mais apropriada para uma planilha). Cada nó (exceto aqueles na linha superior e no lado esquerdo) é apontado pelo nó diretamente acima dele e pelo nó à sua esquerda. Você poderá começar no nó superior esquerdo e navegar para, digamos, o nó na terceira linha e quinta coluna seguindo os ponteiros duas linhas para baixo e quatro colunas à direita. Assuma que sua matriz seja criada com dimensões especificadas (7 por 10, por exemplo). Você deverá ser capaz de inserir valores nos nós especificados e exibir o conteúdo da matriz.

6

Recursão

NESTE CAPÍTULO

- NÚMEROS TRIANGULARES
- FATORIAIS
- ANAGRAMAS
- UMA BUSCA BINÁRIA RECURSIVA
- AS TORRES DE HANOI
- ORDENAÇÃO POR INTERCALAÇÃO
- ELIMINANDO A RECURSÃO
- ALGUMAS APLICAÇÕES RECURSIVAS INTERESSANTES

Recursão é uma técnica de programação na qual um método (função) chama a si mesmo. Isso poderia parecer algo estranho a fazer ou mesmo um erro catastrófico. Recursão, porém, é uma das técnicas mais interessantes e uma das mais surpreendetemente efetivas em programação. Como melhorar a si mesmo, a recursão parece incrível quando você a encontra pela primeira vez. Porém, ela não só funciona como também fornece uma estrutura conceitual única para resolver muitos problemas.

Neste capítulo, examinaremos vários exemplos para mostrar a grande variedade de situações nas quais a recursão pode ser aplicada. Calcularemos números triangulares e fatoriais, iremos gerar anagramas, executaremos uma busca binária recursiva, resolveremos o quebra-cabeça das Torres de Hanoi e investigaremos uma técnica de ordenação chamada ordenação por intercalação. Applets Workshop são fornecidos para demonstrar as Torres de Hanoi e ordenação por intercalação.

Também analisaremos as capacidades e fraquezas da recursão e mostraremos como uma abordagem recursiva pode ser transformada em uma abordagem baseada em pilhas.

Números triangulares

É dito que os Pitagorianos, um grupo de matemáticos na Grécia antiga que trabalhava com Pitágoras (do famoso teorema de Pitágoras), sentiram uma conexão mística com a série de números 1, 3, 6, 10, 15, 21, ... (onde ... significa que a série continua indefinidamente). Você pode descobrir o próximo membro dessa série?

O enésimo termo na série é obtido adicionando n ao termo anterior. Assim, o segundo termo é encontrado adicionando 2 ao primeiro (que é 1), dando 3. O terceiro termo é 3 adicionado ao segundo termo (que é 3) dando 6, etc.

Os números nesta série são chamados de *números triangulares* porque eles podem ser visualizados como uma organização triangular de objetos, mostrados como pequenos quadrados na Figura 6.1.

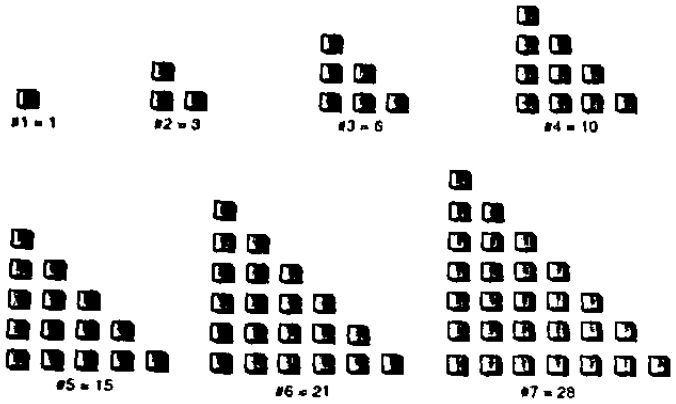

Figura 6.1 Os números triangulares.

Encontrando o enésimo termo usando um laço

Suponha que você queira encontrar o valor de algum enésimo termo arbitrário na série – digamos o quarto termo (cujo valor é 10). Como o calcularia? Vendo a Figura 6.2, você pode decidir que o valor de qualquer termo pode ser obtido somando todas as colunas verticais de quadrados.

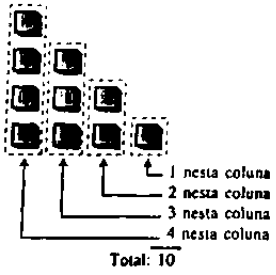

Figura 6.2 Número triangular como colunas.

No quarto item, a primeira coluna tem quatro quadrados pequenos, a segunda coluna tem três etc. Somar 4+3+2+1 dará 10.

O seguinte método triangle() usa essa técnica baseada em colunas para encontrar um número triangular. Ele soma todas as colunas, de uma altura n a uma altura 1:

```
int triangle(int n)
 {
 int total = 0;

 while(n > 0)              // até n ser 1
  {
  total = total + n;  // some n (altura da coluna) ao total
  --n;                // decrementa a altura da coluna
  }
 return total;
 }
```

O método fica no laço n vezes, adicionando n a total na primeira vez, n-1 na segunda vez e assim em diante até 1, saindo do laço quando n chega a 0.

Encontrando o enésimo termo usando recursão

A abordagem de laço pode parecer imediata, mas há outra maneira de ver esse problema. O valor do enésimo termo pode ser considerado como a soma de apenas duas coisas, ao invés de uma série inteira. Elas são

1. A primeira coluna (mais alta), que tem o valor n.
2. A soma de todas as colunas restantes.

Isto é mostrado na Figura 6.3.

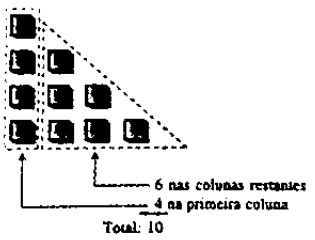

Figura 6.3 Número triangular como uma coluna mais um triângulo.

Encontrando as colunas restantes

Se conhecêssemos um método que encontrasse a soma de todas as colunas restantes, poderíamos escrever nosso método triangle(), que retorna o valor do enésimo número triangular, assim:

```
int  triangle(int  n)
  {
  return  ( n  +  sumRemainingColumns(n)  );    //  (versão  incompleta)
  }
```

Mas o que obtivemos aqui? Parece que escrever o método sumRemainingColumns() é tão difícil quanto escrever o método triangle() em primeiro lugar.

Porém, observe na Figura 6.3 que a soma de todas as colunas restantes para o termo n é igual à soma de *todas* as colunas para o termo n-1. Assim, se conhecêssemos um método que somou todas as colunas para o termo n, poderíamos chamá-lo com um argumento n-1 para encontrar a soma de todas as colunas restantes para o termo n:

```
int  triangle(int  n)
  {
  return  ( n  +  sumAllColumns(n-1)  );        //  (versão  incompleta)
  }
```

Mas quando você o considerá, o método sumAllColumns() está fazendo exatamente a mesma coisa que o método triangle(): somar todas as colunas para algum número n passado como um argumento. Então por que não usar o próprio método triangle(), ao invés de algum outro método? Isso ficaria assim:

```
int  triangle(int  n)
  {
  return  ( n  +  triangle(n-1)  );             //  (versão  incompleta)
  }
```

Você pode ficar surpreso com o fato de que um método possa chamar a si mesmo, mas por que não deveria ser capaz? Uma chamada a um método é (entre outras coisas) uma transferência de controle para o início do método. Essa transferência de controle pode ocorrer de dentro do método, assim como de fora.

Passando a responsabilidade

Todas estas abordagens podem parecer estar passando a responsabilidade. Alguém me pede para encontrar o 9º número triangular. Eu sei que isto é 9 mais o 8º número triangular, portanto, chamo Harry e peço a ele para encontrar o 8º número triangular. Quando ele me disser, adicionarei 9 àquilo que ele me diga e esta será a resposta.

Harry sabe que o 8º número triangular é 8 mais o 7º número triangular, portanto ele chama Sally e pede a ela para encontrar o 7º número triangular. Esse processo continua com cada pessoa passando a responsabilidade para outra.

Onde essa passagem de responsabilidade termina? Alguém em algum ponto tem que ser capaz de descobrir uma resposta que não envolva pedir ajuda a outra pessoa. Se isso não acontecesse, haveria uma cadeia infinita de pessoas fazendo perguntas a outras pessoas - um tipo de esquema aritmético Ponzi que nunca acabaria. No caso de triangle(), isso significaria o método chamando a si mesmo sempre em uma série infinita, que finalmente paralisaria o programa.

A responsabilidade pára aqui

Para evitar um retorno infinito, a pessoa que é solicitada a encontrar o primeiro número triangular da série, quando n é 1, deve saber, sem perguntar a outra pessoa, que a resposta é 1. Não há número menor para perguntar a outra pessoa, não há o que adicionar a qualquer outra coisa, portanto, a responsabilidade pára aqui. Podemos expressar isso adicionando uma condição ao método triangle():

```
int triangle(int n)
 {
 if(n==1)
  return 1;
 else
  return ( n + triangle(n-1) );
 }
```

A condição que leva um método recursivo a retornar sem fazer outra chamada recursiva é referida como *caso base*. É fundamental que todo método recursivo tenha um caso base para impedir a recursão infinita e o conseqüente fim do programa.

O programa triangle.java

Recursão funciona de fato? Se você executar o programa triangle.java, verá que sim. Forneça um valor para o número do termo, n, e o programa exibirá o valor do número triangular correspondente. A Listagem 6.1 mostra o programa triangle.java.

Listagem 6.1 O programa triangle.java

```
// triangle.java
// avalia números triangulares
// para executar este programa: C>java TriangleApp
import java.io.*;            // para E/S
////////////////////////////////////////////////////////////////
class TriangleApp
 {
 static int theNumber;
```

Listagem 6.1 O programa triangle.java (continuação)

```
  public static void main(String[ ] args) throws IOException
    {
    System.out.print("Enter a number: ");
    theNumber = getInt();
    int theAnswer = triangle(theNumber);
    System.out.println("Triangle= "+theAnswer);
    } // fim de main()
//-------------------------------------------------------------
  public static int triangle(int n)
    {
    if(n==1)
      return1;
    else
      return( n + triangle(n-1) );
    }
//-------------------------------------------------------------
  public static String getString() throws IOException
    {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
    }
//-------------------------------------------------------------
  public static int getInt() throws IOException
    {
    String s = getString();
    return Integer.parseInt(s);
    }
//-------------------------------------------------------------
  } // fim da classe TriangleApp
////////////////////////////////////////////////////////////////
```

A rotina main() solicita ao usuário um valor para n, chama triangle() e exibe o valor de retorno. O método triangle() chama a si mesmo repetidamente para fazer todo o trabalho.

Eis um exemplo de saída:

```
Enter a number: 1000
Triangle: 500500
```

Ocasionalmente, se você for cético com os resultados retornados a partir de triangle(), poderá verificá-los usando a seguinte fórmula:

enésimo número triangular = $(n^2+n)/2$

O que está realmente acontecendo?

Modifiquemos o método triangle() para fornecer uma compreensão sobre o que está acontecendo quando ele executa. Iremos inserir algumas instruções de saída para acompanhar os argumentos e os valores de retorno:

```
public static int triangle(int n)
 {
 System.out.println("Entering: n=" + n);
 if(n==1)
  {
  System.out.println("Returning 1");
  return 1;
  }
 else
  {
  int temp = n + triangle(n-1);
  System.out.println("Returning " + temp);
  return temp;
  }
 }
```

Eis a interação quando o método triangle() anterior é substituído por este método e o usuário fornece 5:

```
Enter a number: 5

Entering: n=5
Entering: n=4
Entering: n=3
Entering: n=2
Entering: n=1
Returning 1
Returning 3
Returning 6
Returning 10
Returning 15

Triangle = 15
```

Sempre que o método triangle() chama a si mesmo, seu argumento, que começa em 5, é reduzido em 1. O método mergulha em si mesmo sucessivamente até que seu argumento seja reduzido a 1. Então ele retorna. Isso inicializa uma série inteira de retornos. O método surge de novo, como uma fênix, das versões descartadas de si mesmo. Sempre que retorna, adiciona o valor n com o qual foi chamado ao valor de retorno do método que ele chamou.

Os valores de retorno recapitulam a série de números triangulares, até que a resposta seja retornada para main(). A Figura 6-4 mostra como cada chamada do método triangle() pode ser imaginada como estando "dentro" da anterior.

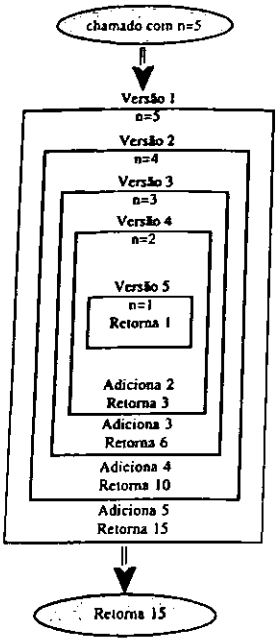

Figura 6.4 O método triangle() recursivo.

Note que, logo antes da versão mais interna retornar 1, há de fato cinco encarnações diferentes de triangle() em existência ao mesmo tempo. A mais externa recebeu o argumento 5; a mais interna recebeu o argumento 1.

Características de métodos recursivos

Embora seja curto, o método triangle() possui os principais recursos comuns a todas as rotinas recursivas:

- Chama a si mesmo.
- Quando chama a si mesmo, faz isso para resolver um problema menor.
- Há alguma versão do problema que é simples o bastante para que a rotina possa resolvê-lo e retornar sem chamar a si mesma.

Em cada chamada sucessiva de um método recursivo a si mesmo, o argumento fica menor (ou talvez uma faixa descrita por diversos argumentos fique menor), refletindo o fato de

que o problema tornou-se "menor" ou mais fácil. Quando o argumento ou a faixa atinge um certo tamanho mínimo, uma condição é disparada e o método retorna sem chamar a si mesmo.

Recursão é eficiente?

Chamar um método envolve um certo overhead. O controle tem que ser transferido do local da chamada para o início do método. Além disso, os argumentos para o método e o endereço para o qual o método deve retornar têm que ser colocados em uma pilha interna para que o método possa acessar os valores do argumento e saiba onde retornar.

No caso do método triangle(), é provável que, como resultado desse overhead, a abordagem do laço while seja executada mais rapidamente que a abordagem recursiva. A desvantagem pode não ser significante, mas se houver um número grande de chamadas do método como resultado de um método recursivo, poderia ser desejável eliminar a recursão. Falaremos sobre essa questão mais no final deste capítulo.

Outra ineficiência é que a memória é usada para armazenar todos os argumentos intermediários e valores de retorno na pilha interna do sistema. Isso poderá causar problemas se houver uma grande quantidade de dados, levando a pilha a estourar.

A recursão é geralmente usada porque simplifica um problema conceitualmente, não porque é inerentemente mais eficiente.

Indução matemática

Recursão é o equivalente em programação à indução matemática. Indução matemática é uma maneira de definir algo em termos de si mesmo. (O termo é também usado para descrever uma abordagem relacionada a provar teoremas.) Usando indução, poderíamos definir os números triangulares matematicamente dizendo:

tri(n) = 1 se n = 1

tri(n) = n + tri(n-1) se n > 1

Definir algo em termos de si mesmo pode parecer circular, mas na verdade é perfeitamente válido (contanto que haja um case base).

Fatoriais

Os fatoriais são similaress em conceito aos números triangulares, exceto que multiplicação é usada, em vez de adição. O número triangular correspondente a n é encontrado adicionando n ao número triangular n-1, ao passo que o fatorial de n é encontrado multiplicando n pelo fatorial n-1. Isto é, o quinto número triangular é 5+4+3+2+1, enquanto o fatorial de 5 é 5*4*3*2*1, que é igual a 120. A Tabela 6.1 mostra os fatoriais dos 10 primeiros números.