17

Estruturas de dados

OBJETIVOS

- **Neste capítulo, você aprenderá:**
- **Como formar estruturas de dados encadeadas utilizando referências, classes auto-referenciais e recursão.**
- **Como as classes empacotadoras de tipo permitem aos programas processar valores dos dados primitivos como objetos.**
- **Como utilizar autoboxing para converter um valor primitivo em um objeto da classe empacotadora de tipo correspondente.**
- **Como utilizar auto-unboxing para converter um objeto de uma classe empacotadora de tipo em um valor primitivo.**
- **Como criar e manipular estruturas de dados dinâmicas, como listas encadeadas, filas, pilhas e árvores binárias.**
- **Vários aplicativos importantes de estruturas de dados encadeadas.**
- **Como criar estruturas de dados reutilizáveis com classes, herança e composição.**

Sumário

17.1 Introdução

- **Estruturas de dados dinâmicas:**
  - **Estruturas de dados linear:**
    - **listas vinculadas;**
    - **pilhas; e**
    - **filas.**
  - **Árvores binárias.**

17.2 Classes empacotadoras de tipo para tipos primitivos

- **Classes empacotadoras de tipo:**
  - **Também conhecidas como *classes invólucros de tipo*.**
  - **No pacote `java.lang`.**
  - **Permitem que programadores manipulem valores de tipo primitivo como objetos.**
  - **`Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` e `Short`.**

17.3 Autoboxing e auto-unboxing

- **Conversão boxing:**
  - **Converte um valor de um tipo primitivo em um objeto da classe empacotadora de tipo correspondente.**
- **Conversão unboxing:**
  - **Converte um objeto de uma classe empacotadora de tipo em um valor do tipo primitivo correspondente.**
- **O J2SE 5.0 realiza essas conversões automaticamente:**
  - **Chamadas *autoboxing* e *auto-unboxing*.**

17.4 Classes auto-referenciais

- **Classe auto-referencial:**
  - **Contém uma variável de instância que referencia um outro objeto do mesmo tipo de classe.**
    - **Essa variável de instância é chamada *link*.**
      - **Uma referência `null` indica que o link não referencia um outro objeto.**
        - **Ilustrada por uma barra invertida nos diagramas.**

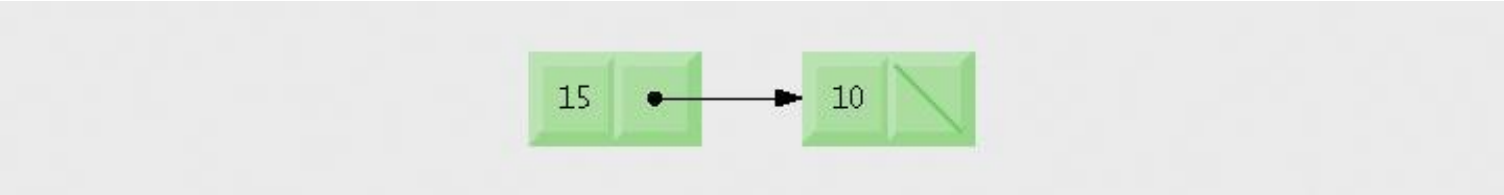

Figura 17.1 | Objetos de classe auto-referencial vinculados entre si.

17.5 Alocação dinâmica de memória

- **Alocação dinâmica de memória:**
  - **A capacidade de um programa para obter mais espaço de memória em tempo de execução a fim de armazenar novos nós e liberar espaço não mais necessário.**
    - **Em vez disso, o Java realiza coleta de lixo automática de objetos que não são mais referenciados em um programa.**
  - `Node nodeToAdd = new Node( 10 );`.
  - **Aloca a memória para armazenar um objeto `Node` e retornar uma referência ao objeto, que é atribuído a `nodeToAdd`.**
  - **Lança um `OutOfMemoryError` se não houver memória suficiente disponível.**

17.6 Listas vinculadas

- **Listas vinculadas:**
  - **Uma lista vinculada é uma coleção linear de nós:**
    - **Objetos de classe auto-referencial conectados por links de referência.**
    - **Podem conter dados de qualquer tipo.**
  - **Em geral, um programa acessa uma lista encadeada via uma referência ao primeiro nó na lista.**
    - **Um programa acessa cada nó subseqüente via a referência do link armazenado no nó anterior.**
  - **São dinâmicas:**
    - **O comprimento de uma lista pode aumentar ou diminuir conforme necessário.**
    - **Tornam-se cheias apenas quando o sistema tem memória insuficiente para satisfazer solicitações de alocação de armazenamento dinâmico.**

Dica de desempenho 17.1

Um array pode ser declarado para conter mais elementos que o número de itens esperado, mas isso desperdiça memória. Listas vinculadas fornecem melhor uso da memória nessas situações. As listas encadeadas permitem que o programa se adapte às necessidades de armazenamento em tempo de execução.

Dica de desempenho 17.2

A inserção em uma lista encadeada é rápida — somente duas referências precisam ser modificadas (depois da localização do ponto de inserção). Todos os objetos existentes de nó permanecem em suas localizações atuais na memória.

Dica de desempenho 17.3

A inserção e a exclusão em um array classificado podem consumir muito tempo — todos os elementos que se seguem ao elemento inserido ou excluído devem ser deslocados apropriadamente.

17.6 Listas vinculadas (*Cont.*)

- **Lista vinculada individualmente:**
  - **Cada nó contém uma referência ao próximo nó na lista.**
- **Lista vinculada duplamente:**
  - **Cada nó contém uma referência ao próximo nó na lista e uma referência ao nó anterior na lista.**
  - **A classe `LinkedList` do `java.utils` é uma implementação de lista duplamente vinculada.**

Dica de desempenho 17.4

Normalmente, os elementos de um array são contíguos na memória. Isso permite acesso imediato a qualquer elemento do array, pois seu endereço pode ser calculado diretamente como seu deslocamento a partir do início do array. Listas vinculadas não suportam esse tipo de acesso imediato a seus elementos — um elemento só pode ser acessado percorrendo a lista a partir do início (ou do final, em uma lista duplamente vinculada).

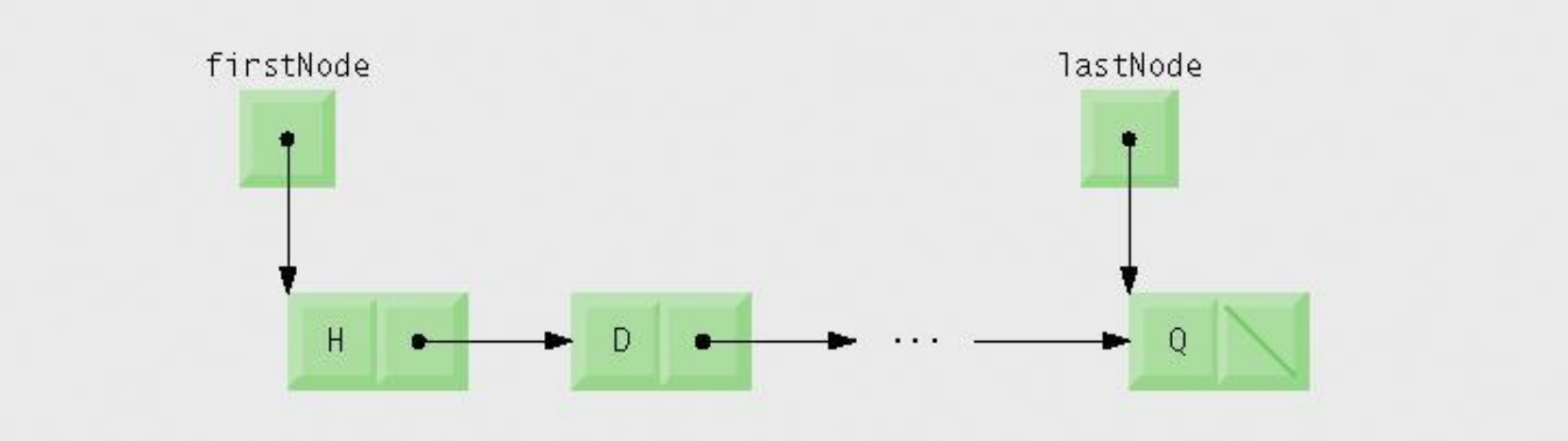

Figura 17.2 | Representação gráfica de uma lista encadeada.

Resumo

List.java

(1 de 6)

```
1  // Fig. 17.3: List.java
2  // Definições da classe ListNode e List.
3  package com.deitel.jhtp6.ch17;
4
5  // classe para representar um nó em uma lista
6  class ListNode
7  {
8     // membros de acesso de pacote; List pode acessar esses diretamente
9     Object data;
10    ListNode nextNode;
11
12    // construtor cria um ListNode que referencia o objeto
13    ListNode( Object object )
14    {
15       this( object, null );
16    } // fim do construtor ListNode de um argumento
17
18    // construtor cria ListNode que se refere ao
19    // Object e ao próximo ListNode
20    ListNode( Object object, ListNode node )
21    {
22       data = object;
23       nextNode = node;
24    } // fim do construtor ListNode de dois argumentos
25
```

Campo **`data`** pode referenciar qualquer objeto

Armazena uma referência ao próximo objeto **`ListNode`** na lista vinculada

Resumo

List.java

(2 de 6)

```
26    // retoran referência aos dados no nó
27    Object getObject()
28    {
29       return data; // retorna Objeto neste nó
30    } // fim do método getObject
31
32    // retorna referência ao próximo nó na lista
33    ListNode getNext()
34    {
35       return nextNode; // obtém próximo nó
36    } // fim do método getNext
37 } // fim da classe ListNode
38
39 // definição da classe List
40 public class List
41 {
42    private ListNode firstNode;
43    private ListNode lastNode;
44    private String name; // string como "list" usada na impressão
45
46    // construtor cria List vazia com "list" como o nome
47    public List()
48    {
49       this( "list" );
50    } // fim do construtor List sem argumentos
51
```

Referências aos primeiros e últimos **`ListNodes`** em uma **`List`**

Construtor de um argumento

Resumo

```
52    // construtor cria uma List vazia com um nome
53    public List( String listName )
54    {
55       name = listName;
56       firstNode = lastNode = null;
57    } // fim do construtor List de um argumento
58
59    // insere Object na frente de List
60    public void insertAtFront( Object insertItem )
61    {
62       if ( isEmpty() ) // firstNode e lastNode referenciam o mesmo objeto
63          firstNode = lastNode = new ListNode( insertItem );
64       else // firstNode referencia o novo nó
65          firstNode = new ListNode( insertItem, firstNode );
66    } // fim do método insertAtFront
67
68    // insere Object no final de List
69    public void insertAtBack( Object insertItem )
70    {
71       if ( isEmpty() ) // firstNode e lastNode referenciam o mesmo objeto
72          firstNode = lastNode = new ListNode( insertItem );
73       else // nextNode do lastNode referencia o novo nó
74          lastNode = lastNode.nextNode = new ListNode( insertItem );
75    } // fim do método insertAtBack
76
```

Inicializa as duas referências para **`null`**

`List.java`

(3 de 6)

Resumo

List.java

(4 de 6)

```
77    // remove primeiro nó de List
78    public Object removeFromFront() throws EmptyListException
79    {
80       if ( isEmpty() ) // lança exceção se List estiver vazia
81          throw new EmptyListException( name );
82
83       Object removedItem = firstNode.data; // recupera dados sendo removidos
84
85       // atualiza referências firstNode e lastNode
86       if ( firstNode == lastNode )
87          firstNode = lastNode = null;
88       else
89          firstNode = firstNode.nextNode;
90
91       return removedItem; // retorna dados de nó removidos
92    } // fim do método removeFromFront
93
94    // remove último nó de List
95    public Object removeFromBack() throws EmptyListException
96    {
97       if ( isEmpty() ) // lança exceção se List estiver vazia
98          throw new EmptyListException( name );
99
100      Object removedItem = lastNode.data; // recupera dados sendo removidos
101
```

Resumo

List.java

(5 de 6)

```
102     // atualiza referências firstNode e lastNode
103     if ( firstNode == lastNode )
104        firstNode = lastNode = null;
105     else // localiza o novo último nó
106     {
107        ListNode current = firstNode;
108
109        // faz loop enquanto nó atual não referencia lastNode
110        while ( current.nextNode != lastNode )
111           current = current.nextNode;
112
113        lastNode = current; // atual é novo lastNode
114        current.nextNode = null;
115     } // fim de else
116
117     return removedItem; // retorna dados de nó removidos
118  } // fim do método removeFromBack
119
120  // determina se a lista está ou não vazia
121  public boolean isEmpty()
122  {
123     return firstNode == null; // retorna true se List estiver vazia
124  } // fim do método isEmpty
125
```

Método predicado que determina se a lista está vazia

Resumo

List.java

(6 de 6)

```
126   // output List contents
127   public void print()
128   {
129      if ( isEmpty() )
130      {
131         System.out.printf( "Empty %s\n", name );
132         return;
133      } // end if
134
135      System.out.printf( "The %s is: ", name );
136      ListNode current = firstNode;
137
138      // while not at end of list, output current node's data
139      while ( current != null )
140      {
141         System.out.printf( "%s ", current.data );
142         current = current.nextNode;
143      } // end while
144
145      System.out.println( "\n" );
146   } // end method print
147} // end class List
```

Exibe o conteúdo das listas

Exibe uma mensagem indicando que a lista está vazia

Gera saída de uma representação de string de **`current.data`**

Move-se para o próximo nó na lista

Resumo

EmptyListException.java

```
1  // Fig. 17.4: EmptyListException.java
2  // Definição da classe EmptyListException.
3  package com.deitel.jhtp6.ch17;
4
5  public class EmptyListException extends RuntimeException
6  {
7     // construtor sem argumentos
8     public EmptyListException()
9     {
10       this( "List" ); // chama outro construtor de EmptyListException
11    } // fim do construtor EmptyListException sem argumentos
12
13    // construtor de um argumento
14    public EmptyListException( String name )
15    {
16       super( name + " is empty" ); // chama construtor de superclasse
17    } // fim de construtor EmptyListException de um argumento
18 } // fim da classe EmptyListException
```

Resumo

ListTest.java

(1 de 3)

```
1  // Fig. 17.5: ListTest.java
2  // Classe ListTest para demonstrar as capacidades de List.
3  import com.deitel.jhtp6.ch17.List;
4  import com.deitel.jhtp6.ch17.EmptyListException;
5
6  public class ListTest
7  {
8     public static void main( String args[] )
9     {
10       List list = new List(); // cria o contêiner List
11
12       // insere inteiros na lista
13       list.insertAtFront( -1 );
14       list.print();
15       list.insertAtFront( 0 );
16       list.print();
17       list.insertAtBack( 1 );
18       list.print();
19       list.insertAtBack( 5 );
20       list.print();
21
```

Insere objetos no começo da lista utilizando o método **insertAtFront**

Insere objetos no final da lista utilizando o método **insertAtBack**

A JVM faz o autoboxing de cada valor literal convertendo-os em um objeto **Integer**

Resumo

```
22       // remove objetos da lista; imprime depois de cada remoção
23       try
24       {
25          Object removedObject = list.removeFromFront();
26          System.out.printf( "%s removed\n", removedObject );
27          list.print();
28
29          removedObject = list.removeFromFront();
30          System.out.printf( "%s removed\n", removedObject );
31          list.print();
32
33          removedObject = list.removeFromBack();
34          System.out.printf( "%s removed\n", removedObject );
35          list.print();
36
37          removedObject = list.removeFromBack();
38          System.out.printf( "%s removed\n", removedObject );
39          list.print();
40       } // fim do try
41       catch ( EmptyListException emptyListException )
42       {
43          emptyListException.printStackTrace();
44       } // fim do catch
45    } // fim do main
46 } // fim da classe ListTest
```

Exclui os objetos do início da lista usando o método **`removeFromFront`**

Exclui os objetos do final da lista usando o método **`removeFromBack`**

Chama o método **`List print`** para exibir o conteúdo da lista atual

Handler de exceção para **`EmptyListException`**

`ListTest.java`

(2 de 3)

```
The list is: -1

The list is: 0 -1

The list is: 0 -1 1

The list is: 0 -1 1 5

0 removed
The list is: -1 1 5

-1 removed
The list is: 1 5

5 removed
The list is: 1

1 removed
Empty list
```

Resumo

ListTest.java

(3 de 3)

17.6 Listas vinculadas (*Cont.*)

- **Passos do método `insertAtFront`:**
  1. **Chama `isEmpty` para determinar se a lista está vazia.**
  2. **Se a lista estiver vazia, atribui `firstNode` e `lastNode` ao novo `ListNode` que foi inicializado com `insertItem`.**
     - **A chamada ao construtor `ListNode` configura os dados para que referenciem o `insertItem` passado como um argumento e configura a referência a `nextNode` como `null`.**
  3. **Se a lista não estiver vazia, configura `firstNode` como um novo objeto `ListNode` e inicializa esse objeto com `insertItem` e `firstNode`.**
     - **A chamada do construtor `ListNode` configura os dados para que referenciem o `insertItem` passado como um argumento e configura a referência ao `ListNode` passado como o argumento que, anteriormente, era o primeiro nó.**

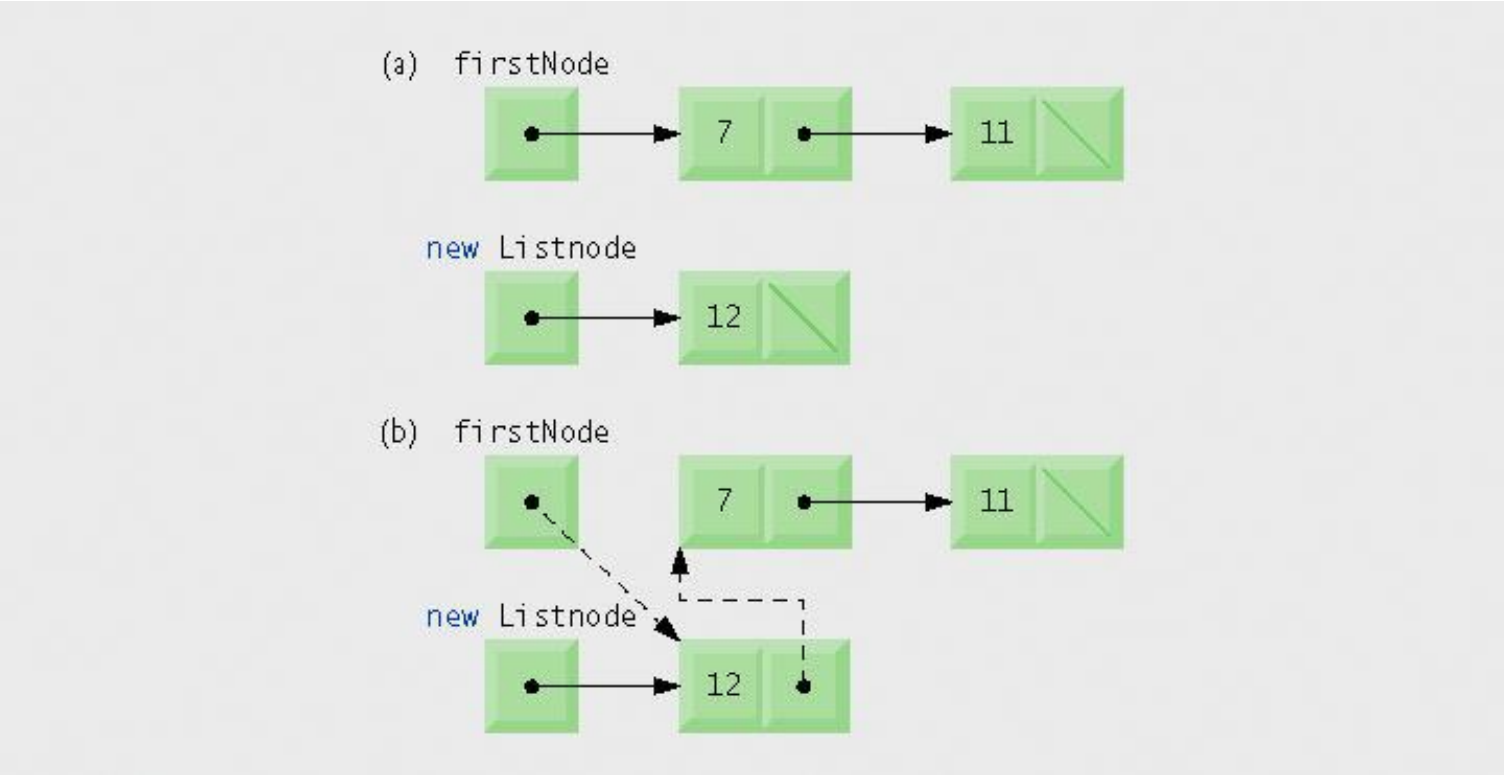

Figura 17.6 | Representação gráfica da operação `insertAtFront`.

17.6 Listas vinculadas (*Cont.*)

- **Passos do método `insertAtBack`:**
  1. **Chama `isEmpty` para determinar se a lista está vazia.**
  2. **Se a lista estiver vazia, atribui `firstNode` e `lastNode` ao novo `ListNode` que foi inicializado com `insertItem`.**
     - **A chamada ao construtor `ListNode` configura os dados para que referenciem o `insertItem` passado como um argumento e configura a referência a `nextNode` como `null`.**
  3. **Se a lista não estiver vazia, atribui a `lastNode` e `lastNode.nextNode` a referência ao novo `ListNode` que foi inicializado com `insertItem`.**
     - **O construtor `ListNode` configura os dados para que referenciem o `insertItem` passado como um argumento e configura a referência a `nextNode` como `null`.**

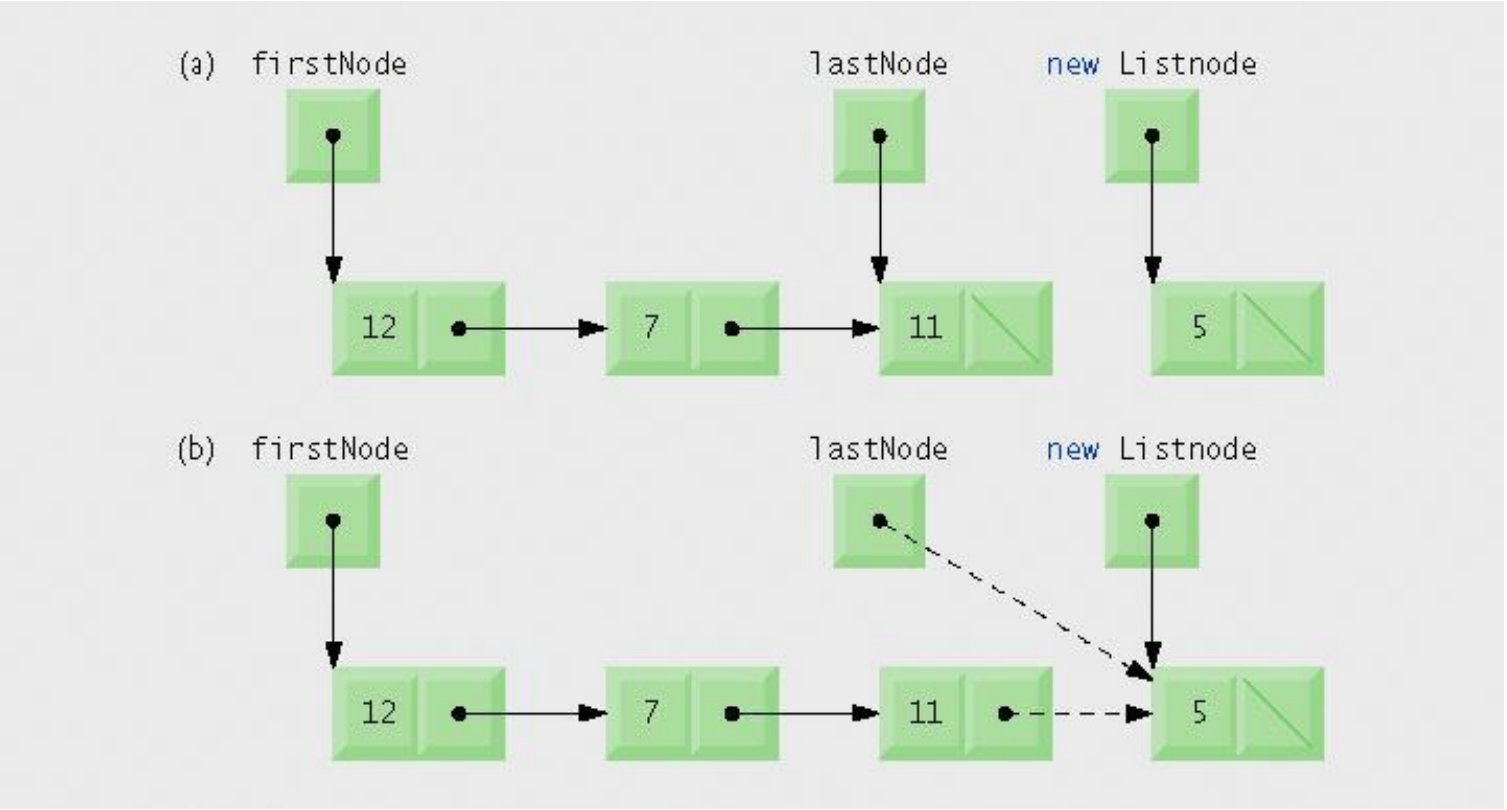

Figura 17.7 | Representação gráfica da operação `insertAtBack`.

17.6 Listas vinculadas (*Cont.*)

- **Passos do método `removeFromFront`:**
  1. **Lança uma `EmptyListException` se a lista estiver vazia.**
  2. **Atribui `firstNode.data` para que referenciem `removedItem`.**
  3. **Se `firstNode` e `lastNode` referenciarem o mesmo objeto, configura `firstNode` e `lastNode` como `null`.**
  4. **Se a lista tiver mais de um nó, atribui o valor do `firstNode.nextNode` ao `firstNode`.**
  5. **Retorna a referência `removeItem`.**

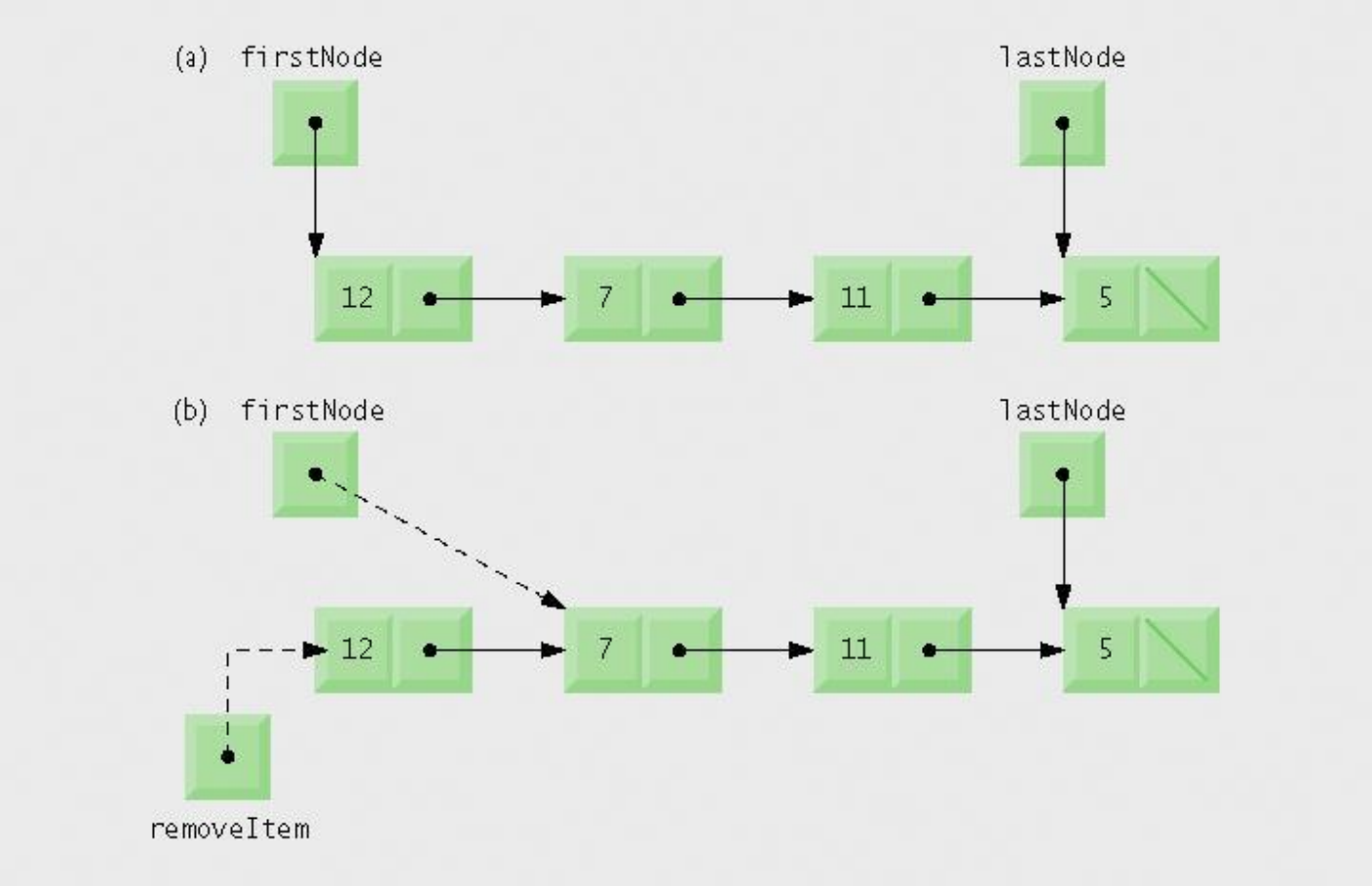

Figura 17.8 | Representação gráfica da operação `removeFromFront`.

17.6 Listas vinculadas (*Cont.*)

- **Passos do método `removeFromBack`:**
  1. **Lança uma `EmptyListException` se a lista estiver vazia.**
  2. **Atribui `lastNode.data` a `removedItem`.**
  3. **Se `firstNode` e `lastNode` referenciarem o mesmo objeto, configura `firstNode` e `lastNode` como `null`.**
  4. **Se a lista tiver mais de um nó, cria a referência ao `ListNode` atual e lhe atribui `firstNode`.**
  5. **'Percorre a lista' que contém o nó atual até referenciar o nó antes do último nó.**
     - **O loop `while` atribui `current.nextNode` a `current` desde que `current.nextNode` não seja `lastNode`.**

17.6 Listas vinculadas (*Cont.*)

- **Passos do método `removeFromBack`:**
  6. **Atribui `current` a `lastNode`.**
  7. **Configura `current.nextNode` como `null`.**
  8. **Retorna a referência `removeItem`.**

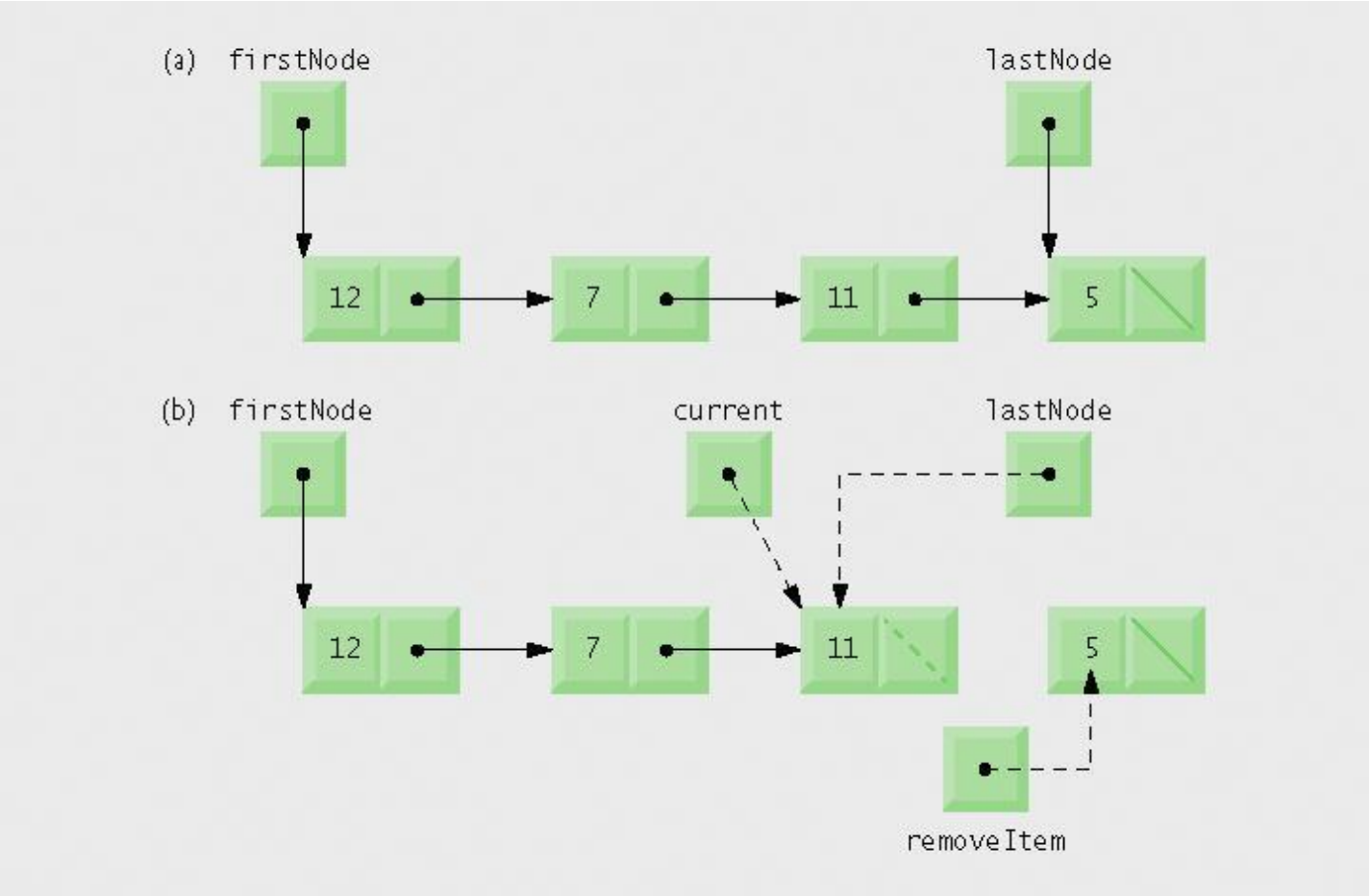

Figura 17.9 | Representação gráfica da operação `removeFromBack`.

17.7 Pilhas

- **Pilhas:**
  - **Estrutura de dados último a entrar, primeiro a sair (*last-in-first-out* — LIFO)**
    - **O método `push` adiciona um novo nó à parte superior da pilha.**
    - **O método `pop` remove um nó da parte superior da pilha e retorna os dados do nó removido.**
  - **Pilha de execução do programa:**
    - **Contém os endereços de retorno dos métodos chamadores.**
    - **Também contém as variáveis locais para os métodos chamados.**
  - **Utilizado pelo compilador para avaliar expressões aritméticas.**

17.7 Pilhas (*Cont.*)

- **Classe de pilha que herda de `List`:**
  - **Os métodos de pilha `push`, `pop`, `isEmpty` e `print` são realizados pelos métodos herdados `insertAtFront`, `removeFromFront`, `isEmpty` e `print`.**
    - `push` **chama** `insertAtFront`.
    - `pop` **chama** `removeFromFront`.
    - **Os métodos** `isEmpty` **e** `print` **podem ser chamados como herdados.**
  - **Os outros métodos `List` também são herdados.**
    - **Incluindo os métodos que não devem estar na interface `public` da classe de pilha.**

Resumo

StackInheritance.java

```
1  // Fig. 17.10: StackInheritance.java
2  // Derivado da classe List.
3  package com.deitel.jhtp6.ch17;
4
5  public class StackInheritance extends List
6  {
7     // construtor sem argumento
8     public StackInheritance()
9     {
10       super( "stack" );
11    } // fim do construtor sem argumento StackInheritance
12
13    // adiciona objeto à pilha
14    public void push( Object object )
15    {
16       insertAtFront( object );
17    } // fim do método push
18
19    // remove objeto da pilha
20    public Object pop() throws EmptyListException
21    {
22       return removeFromFront();
23    } // fim do método pop
24 } // fim da classe StackInheritance
```

A classe **StackInheritance** estende a classe **List**

O método **push** chama o método herdado **insertAtFront**

O método **pop** chama o método herdado **removeFromFront**

Resumo

StackInheritance Test.java

(1 de 3)

```
1  // Fig. 17.11: StackInheritanceTest.java
2  // Classe StackInheritanceTest.
3  import com.deitel.jhtp6.ch17.StackInheritance;
4  import com.deitel.jhtp6.ch17.EmptyListException;
5
6  public class StackInheritanceTest
7  {
8     public static void main( String args[] )
9     {
10       StackInheritance stack = new StackInheritance();
11
12       // utiliza método push
13       stack.push( -1 );
14       stack.print();
15       stack.push( 0 );
16       stack.print();
17       stack.push( 1 );
18       stack.print();
19       stack.push( 5 );
20       stack.print();
21
```

Cria um objeto **StackInheritance**

Insere inteiros na pilha

Resumo

```
22       // remove itens da pilha
23       try
24       {
25          Object removedObject = null;
26
27          while ( true )
28          {
29             removedObject = stack.pop(); // utiliza método pop
30             System.out.printf( "%s popped\n", removedObject );
31             stack.print();
32          } // fim do while
33       } // fim do try
34       catch ( EmptyListException emptyListException )
35       {
36          emptyListException.printStackTrace();
37       } // fim do catch
38    } // fim do main
39 } // fim da classe StackInheritanceTest
```

Remove os objetos da pilha em um loop **`while`** infinito

Chama implicitamente o método **`print`** herdado

Exibe o rastreamento de pilha da exceção

`StackInheritanceTest.java`

(2 de 3)

```
The stack is: -1

The stack is: 0 -1

The stack is: 1 0 -1

The stack is: 5 1 0 -1

5 popped
The stack is: 1 0 -1

1 popped
The stack is: 0 -1

0 popped
The stack is: -1

-1 popped
Empty stack
com.deitel.jhtp6.ch17.EmptyListException: stack is empty
        at com.deitel.jhtp6.ch17.List.removeFromFront(List.java:81)
        at com.deitel.jhtp6.ch17.StackInheritance.pop(StackInheritance.java:22)
        at StackInheritanceTest.main(StackInheritanceTest.java:29)
```

Resumo

StackInheritanceTest.java

(3 de 3)

17.7 Pilhas (*Cont.*)

- **Classe de pilha que contém uma referência a uma `List`:**
  - **Permite ocultar os métodos `List` que não devem estar na interface `public` de nossa pilha.**
  - **Cada método de pilha invocado delega a chamada ao método `List` apropriado:**
    - **O método `push` delega ao método `List insertAtFront`.**
    - **O método `pop` delega ao método `List removeFromFront`.**
    - **O método `isEmpty` delega ao método `List isEmpty`.**
    - **O método `print` delega ao método `List print`.**

Resumo

StackComposition.java

(1 de 2)

```
1  // Fig. 17.12: StackComposition.java
2  // Definição da classe StackComposition com o objeto List composto.
3  package com.deitel.jhtp6.ch17;
4
5  public class StackComposition
6  {
7     private List stackList;
8
9     // construtor sem argumentos
10    public StackComposition()
11    {
12       stackList = new List( "stack" );
13    } // fim do construtor StackComposition sem argumentos
14
15    // adiciona objeto à pilha
16    public void push( Object object )
17    {
18       stackList.insertAtFront( object );
19    } // fim do método push
20
```

Referência **private List**

O método **push** delega a chamada ao método **insertAtFront List**

Resumo

```
21    // remove objeto da pilha
22    public Object pop() throws EmptyListException
23    {
24       return stackList.removeFromFront();
25    } // fim do método pop
26
27    // determina se a pilha está vazia
28    public boolean isEmpty()
29    {
30       return stackList.isEmpty();
31    } // fim do método isEmpty
32
33    // gera saída do conteúdo da pilha
34    public void print()
35    {
36       stackList.print();
37    } // fim do método print
38 } // fim da classe StackComposition
```

O método **pop** delega chamadas ao método **List removeFromFront**

O método **isEmpty** delega chamadas ao método **List isEmpty**

O método **print** delega chamadas ao método **List print**

StackComposition.java

(2 de 2)

17.8 Filas

- **Fila:**
  - **Semelhante a uma fila no caixa de um supermercado.**
  - **É uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* — FIFO).**
    - **O enfileiramento insere nós no final da fila.**
    - **O desenfileiramento remove nós do início da fila.**
  - **Utilizado para suportar spooling de impressão.**
    - **Um programa spooler gerencia a fila de trabalhos de impressão.**

17.8 Filas (*Continuação*)

- **Classe de fila que contém uma referência a uma `List`:**
  - **O método `enqueue` chama o método `List insertAtBack`.**
  - **O método `dequeue` chama o método `List removeFromFront`.**
  - **O método `isEmpty` chama o método `List isEmpty`.**
  - **O método `print` chama o método `List print`.**

Resumo

```
1  // Fig. 17.13: Queue.java
2  // Classe Queue.
3  package com.deitel.jhtp6.ch17;
4
5  public class Queue
6  {
7     private List queueList;
8
9     // construtor sem argumentos
10    public Queue()
11    {
12       queueList = new List( "queue" );
13    } // fim do construtor Queue sem argumentos
14
15    // adiciona objeto à fila
16    public void enqueue( Object object )
17    {
18       queueList.insertAtBack( object );
19    } // fim do método enqueue
20
```

Um objeto da classe **`List`**

O método **`enqueue`** chama o método **`List insertAtBack`**

`Queue.java`

(1 de 2)

Resumo

```
21    // remove objeto da pilha
22    public Object dequeue() throws EmptyListException
23    {
24       return queueList.removeFromFront();
25    } // fim do método dequeue
26
27    // determina se a fila está vazia
28    public boolean isEmpty()
29    {
30       return queueList.isEmpty();
31    } // fim do método isEmpty
32
33    // gera saída do conteúdo da fila
34    public void print()
35    {
36       queueList.print();
37    } // fim do método print
38 } // fim da classe Queue
```

O método **queue** chama o método **List removeFromFront**

O método **isEmpty** chama o método **List isEmpty**

O método **print** chama o método **List print**

Queue.java

(2 de 2)

Resumo

QueueTest.java

(1 de 3)

```
1  // Fig. 17.14: QueueTest.java
2  // Classe QueueTest.
3  import com.deitel.jhtp6.ch17.Queue;
4  import com.deitel.jhtp6.ch17.EmptyListException;
5
6  public class QueueTest
7  {
8     public static void main( String args[] )
9     {
10       Queue queue = new Queue();
11
12       // utiliza método enqueue
13       queue.enqueue( -1 );
14       queue.print();
15       queue.enqueue( 0 );
16       queue.print();
17       queue.enqueue( 1 );
18       queue.print();
19       queue.enqueue( 5 );
20       queue.print();
21
```

Cria um objeto **Queue**

Enfileira quatro inteiros

Sumo

Queue.java

(2 de 3)

```
22       // remove objetos da fila
23       try
24       {
25          Object removedObject = null;
26
27          while ( true )
28          {
29             removedObject = queue.dequeue(); // utiliza método dequeue
30             System.out.printf( "%s dequeued\n", removedObject );
31             queue.print();
32          } // fim de while
33       } // fim de try
34       catch ( EmptyListException emptyListException )
35       {
36          emptyListException.printStackTrace();
37       } // fim de catch
38    } // fim de main
39 } // fim da classe QueueTest
```

Desenfileira os objetos na ordem primeiro a entrar, primeiro a sair

Exibe o rastreamento de pilha da exceção

```
The queue is: -1

The queue is: -1 0

The queue is: -1 0 1

The queue is: -1 0 1 5

-1 dequeued
The queue is: 0 1 5

0 dequeued
The queue is: 1 5

1 dequeued
The queue is: 5

5 dequeued
Empty queue
com.deitel.jhtp6.ch17.EmptyListException: queue is empty
        at com.deitel.jhtp6.ch17.List.removeFromFront(List.java:81)
        at com.deitel.jhtp6.ch17.Queue.dequeue(Queue.java:24)
        at QueueTest.main(QueueTest.java:29)
```

Resumo

QueueTest.java

(3 de 3)

17.9 Árvores

- **Árvore:**
  - **O nó-raiz é o primeiro nó em uma árvore.**
  - **Cada link referencia um filho:**
    - **O filho à esquerda é a raiz da subárvore esquerda.**
    - **O filho à direita é a raiz da subárvore direita.**
    - **Os irmãos são os filhos de um nó específico.**
  - **Um nó-folha não tem nenhum filho.**

17.9 Árvores (*Cont.*)

- **Árvore de pesquisa binária:**
  - **Os valores na subárvore esquerda são menores que o valor no nó-pai dessa subárvore; e os valores na subárvore direita são maiores que o valor no nó-pai dessa subárvore.**
- **Percorrendo uma árvore:**
  - **Na ordem – percorre a subárvore esquerda, depois processa a raiz e então percorre a subárvore direita.**
  - **Pré-ordem – processa a raiz, depois percorre a subárvore esquerda e então percorre a subárvore direita.**
  - **Pós-ordem – percorre a subárvore esquerda, depois percorre a subárvore direita e então processa a raiz.**

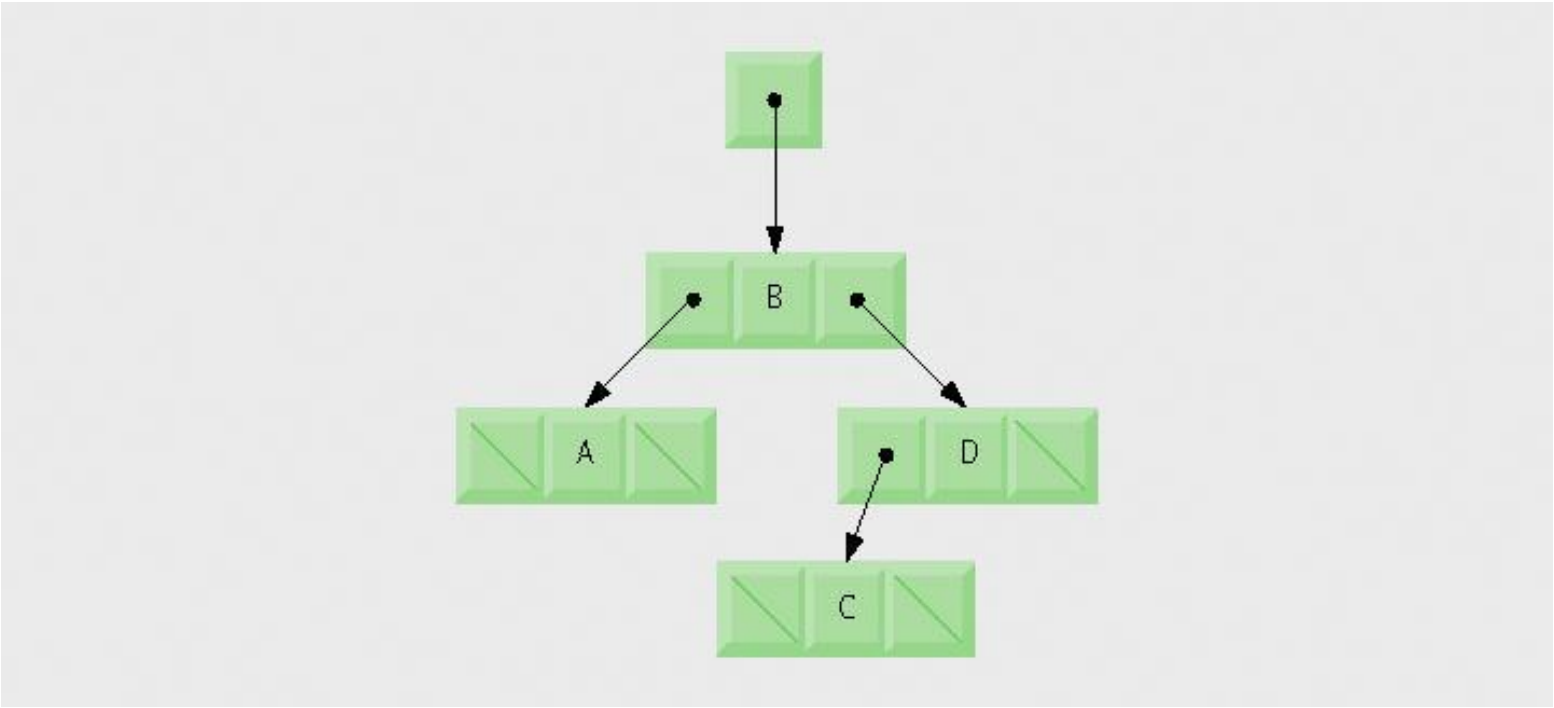

Figura 17.15 | Representação gráfica de árvore binária.

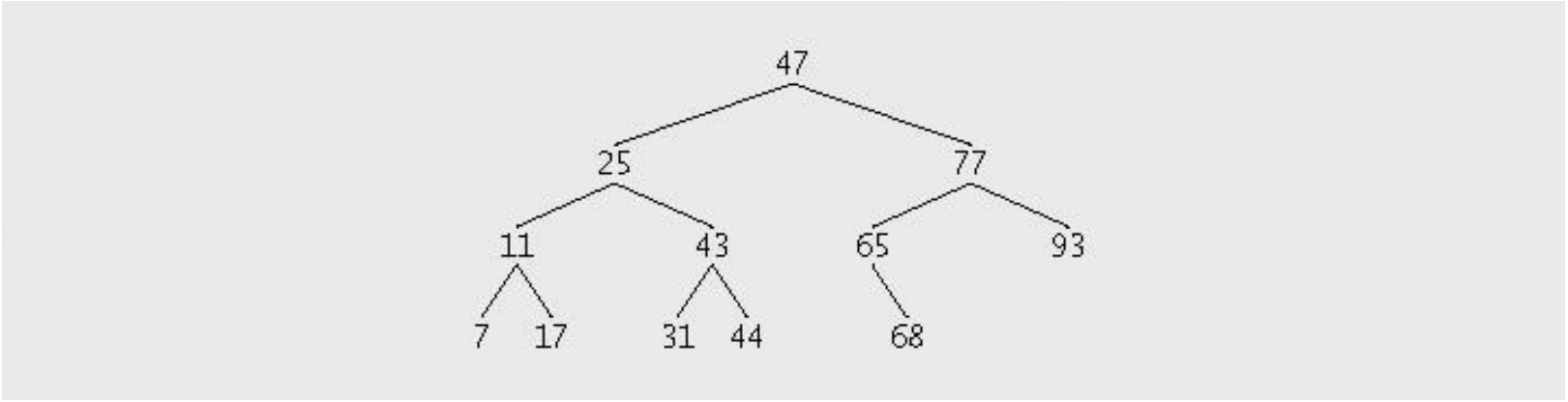

Figura 17.16 | Árvore de pesquisa binária contendo 12 valores.

Resumo

Tree.java

(1 de 5)

```
1  // Fig. 17.17: Tree.java
2  // Definição da classe TreeNode e classe Tree.
3  package com.deitel.jhtp6.ch17;
4
5  // definição da classe TreeNode
6  class TreeNode
7  {
8     // membros de acesso de pacote
9     TreeNode leftNode; // nó esquerdo
10    int data; // valor do nó
11    TreeNode rightNode; // nó direito
12
13    // construtor inicializa dados e os torna um nó-folha
14    public TreeNode( int nodeData )
15    {
16       data = nodeData;
17       leftNode = rightNode = null; // nó não tem nenhum filho
18    } // construtor sem argumento TreeNode
19
```

Declara o filho à esquerda, o valor do nó e o filho à direita

Resumo

```
20    // localiza ponto de inserção e insere novo nó; ignora os valores duplicados
21    public void insert( int insertValue )
22    {
23       // insere na subárvore esquerda
24       if ( insertValue < data )
25       {
26          // insere novo TreeNode
27          if ( leftNode == null )
28             leftNode = new TreeNode( insertValue );
29          else // continua percorrendo subárvore esquerda
30             leftNode.insert( insertValue );
31       } // end if
32       else if ( insertValue > data ) // insere na subárvore direita
33       {
34          // insere novo TreeNode
35          if ( rightNode == null )
36             rightNode = new TreeNode( insertValue );
37          else // continua a percorrer a subárvore direita
38             rightNode.insert( insertValue );
39       } // fim de else if
40    } // fim do método insert
41 } // fim da classe TreeNode
42
```

Aloca um novo **`TreeNode`** e lhe atribui a referência **`leftNode`**

Aloca um novo **`TreeNode`** e lhe atribui a referência **`rightNode`**

`Tree.java`

(2 de 5)

esumo

```
43 // definição da classe Tree
44 public class Tree
45 {
46    private TreeNode root;
47
48    // construtor inicializa uma Tree de inteiros vazia
49    public Tree()
50    {
51       root = null;
52    } // fim do construtor Tree sem argumentos
53
54    // insere um novo nó na árvore de pesquisa binária
55    public void insertNode( int insertValue )
56    {
57       if ( root == null )
58          root = new TreeNode( insertValue ); // cria o nó raiz aqui
59       else
60          root.insert( insertValue ); // chama o método insert
61    } // fim do método insertNode
62
63    // inicia o percurso na pré-ordem
64    public void preorderTraversal()
65    {
66       preorderHelper( root );
67    } // fim do método preorderTraversal
68
```

Referência **TreeNode** para o nó da raiz da árvore

Aloca um novo **TreeNode** e lhe atribui a referência **root**

Chama o método **TreeNode insert**

Chama o método auxiliar **Tree preorderHelper**

Tree.java

(3 de 5)

Resumo

Tree.java

(4 de 5)

```
69    // método recursivo para realizar percurso na pré-ordem
70    private void preorderHelper( TreeNode node )
71    {
72       if ( node == null )
73          return;
74
75       System.out.printf( "%d ", node.data ); // gera saída de dados do nó
76       preorderHelper( node.leftNode );       // percorre subárvore esquerda
77       preorderHelper( node.rightNode );      // percorre subárvore direita
78    } // fim do método preorderHelper
79
80    // inicia percurso na ordem
81    public void inorderTraversal()
82    {
83       inorderHelper( root );
84    } // fim do método inorderTraversal
85
86    // método recursivo para realizar percurso na ordem
87    private void inorderHelper( TreeNode node )
88    {
89       if ( node == null )
90          return;
91
92       inorderHelper( node.leftNode );        // percorre subárvore esquerda
93       System.out.printf( "%d ", node.data ); // gera saída de dados do nó
94       inorderHelper( node.rightNode );       // percorre subárvore direita
95    } // fim do método inorderHelper
96
```

Chama o método auxiliar **`Tree inorderHelper`**

esumo

Tree.java

(5 de 5)

```
97    // inicia percurso na pós-ordem
98    public void postorderTraversal()
99    {
100      postorderHelper( root );
101   } // fim do método postorderTraversal
102
103   // método recursivo para realizar percurso na pós-ordem
104   private void postorderHelper( TreeNode node )
105   {
106      if ( node == null )
107         return;
108
109      postorderHelper( node.leftNode );       // percorre subárvore esquerda
110      postorderHelper( node.rightNode );      // percorre subárvore direita
111      System.out.printf( "%d ", node.data ); // gera saída de dados do nó
112   } // fim do método postorderHelper
113} // fim da classe Tree
```

Chama o método auxiliar `Tree` `postorderHelper`

Resumo

TreeTest.java

(1 de 2)

```
1  // Fig. 17.18: TreeTest.java
2  // Este programa testa a classe Tree.
3  import java.util.Random;
4  import com.deitel.jhtp6.ch17.Tree;
5
6  public class TreeTest
7  {
8     public static void main( String args[] )
9     {
10       Tree tree = new Tree();
11       int value;
12       Random randomNumber = new Random();
13
14       System.out.println( "Inserting the following values: " );
15
16       // insere 10 inteiros aleatórios de 0-99 na árvore
17       for ( int i = 1; i <= 10; i++ )
18       {
19          value = randomNumber.nextInt( 100 );
20          System.out.print( value + " " );
21          tree.insertNode( value );
22       } // end for
23
24       System.out.println ( "\n\nPreorder traversal" );
25       tree.preorderTraversal(); // realiza percurso na pré-ordem da árvore
26
```

Cria um objeto **Tree**

Insere valores em **tree**

Resumo

TreeTest.java

(2 de 2)

```
27       System.out.println ( "\n\nInorder traversal" );
28       tree.inorderTraversal(); // realiza percurso na ordem da árvore
29
30       System.out.println ( "\n\nPostorder traversal" );
31       tree.postorderTraversal(); // realiza percurso na pós-ordem da árvore
32       System.out.println();
33    } // fim do main
34 } // fim da classe TreeTest
```

```
Inserting the following values:
92 73 77 16 30 30 94 89 26 80

Preorder traversal
92 73 16 30 26 77 89 80 94

Inorder traversal
16 26 30 73 77 80 89 92 94

Postorder traversal
26 30 16 80 89 77 73 94 92
```

17.9 Árvores (*Cont.*)

- **Passos do percurso na ordem:**
  1. **Retornar imediatamente se o parâmetro de referência for nulo.**
  2. **Percorrer a subárvore esquerda com uma chamada a `inorderHelper`.**
  3. **Processar o valor no nó.**
  4. **Percorrer a subárvore direita com uma chamada a `inorderHelper`.**
- **Classificação em árvore binária:**
  - **O percurso na ordem de uma árvore de pesquisa binária imprime os valores de nó na ordem crescente.**

17.9 Árvores (*Cont.*)

- **Passos do percurso na pré-ordem:**
  1. **Retorna imediatamente se o parâmetro de referência for `null`.**
  2. **Processa o valor no nó.**
  3. **Percorre a subárvore esquerda com uma chamada a `preorderHelper`.**
  4. **Percorre a subárvore direita com uma chamada a `preorderHelper`.**

17.9 Árvores (*Cont.*)

- **Passos do percurso na pós-ordem:**
  1. **Retorna imediatamente se o parâmetro de referência for `null`.**
  2. **Percorre a subárvore esquerda com uma chamada a `postorderHelper`.**
  3. **Percorre a subárvore direita com uma chamada a `postorderHelper`.**
  4. **Processa o valor no nó.**

17.9 Árvores (*Cont.*)

- **Eliminação de duplicatas:**
  - **Como os valores duplicados seguem as decisões de direcionamento, a operação de inserção por fim compara a duplicata com um nó de mesmo valor.**
  - **A duplicata pode então ser ignorada.**
- **Árvores fortemente empacotadas (ou equilibradas):**
  - **Cada nível contém cerca de duas vezes o número de elementos do nível anterior.**
  - **Encontrar uma correspondência ou determinar se não há nenhuma correspondência entre *n* elementos requer no máximo log *n* comparações.**
- **Percurso na ordem de nível de uma árvore binária:**
  - **Verifica os nós da árvore linha por linha, iniciando no nível do nó raiz.**

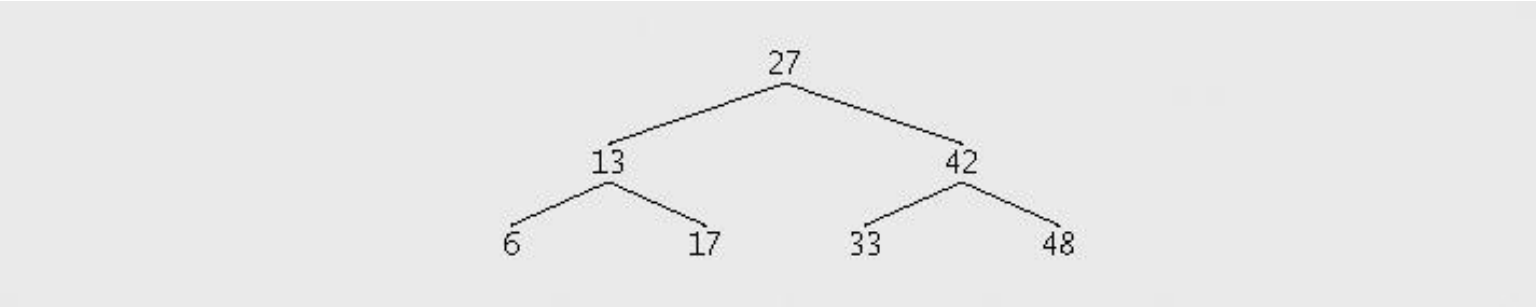

Figura 17.19 | Árvore de pesquisa binária contendo sete valores.