26

Data Structures

© 2009 Pearson Education, Inc. All rights reserved.

OBJECTIVES

In this chapter you will learn:

- To form linked data structures using references, self-referential classes and recursion.
- How boxing and unboxing enable simple-type values to be used where `object`s are expected in a program.
- To create and manipulate dynamic data structures, such as linked lists, queues, stacks and binary trees.
- Various important applications of linked data structures.
- To create reusable data structures with classes, inheritance and composition.

© 2009 Pearson Education, Inc. All rights reserved.

26.1 Introduction

- **Dynamic data structures** can grow and shrink at execution time.
- Linked lists are collections of data items that are “chained together”.
- Stacks have insertions and deletions made at only one end: the **top**.
- Queues represent waiting lines; insertions are made at the back and deletions are made from the front.
- **Binary trees** facilitate high-speed searching and sorting of data.

© 2009 Pearson Education, Inc. All rights reserved.

26.2 Simple-Type `struct`s, Boxing and Unboxing

- Each simple type is declared by a corresponding `struct`.
- A variable of a simple type can be declared using either the keyword for that simple type or the `struct` name.
- The methods related to a simple type are located in the corresponding `struct`.

© 2009 Pearson Education, Inc. All rights reserved.

26.2 Simple-Type `structs`, Boxing and Unboxing (Cont.)

- Simple-type `struct`s inherit from class `ValueType`, which inherits from class `object`.
- Any simple-type value can be assigned to an `object` variable; this is a **boxing conversion**.
- Boxing conversions can be performed either explicitly or implicitly:

```
int i = 5; // create an int value
object object1 = ( object ) i; // explicitly box the
int value
object object2 = i; // implicitly box the int value
```

© 2009 Pearson Education, Inc. All rights reserved.

26.2 Simple-Type structs, Boxing and Unboxing (Cont.)

- An **unboxing conversion** explicitly converts an `object` reference to a simple value,:

```
int int1 = ( int ) object1; // explicitly unbox the int value
```

- Unboxing an `object` that does not refer to the correct simple value type causes an `InvalidCastException`.

© 2009 Pearson Education, Inc. All rights reserved.

- A **self-referential class** contains a member that refers to an object of the same class type.
- In the class declaration in Fig. 26.1, `Next` references an object of type `Node`, an object of the same type being declared.
- `Next` is referred to as a **link**.

Fig26_01.cs

```
1  // Fig. 26.1: Fig26_01.cs
2  // Self-referential Node class declaration.
3  class Node
4  {
5     public int Data { get; set; } // store integer data
6     public Node Next { get; set; } // store reference to next Node
7
8     public Node( int dataValue )
9     {
10       // constructor body
11    } // end constructor
12 } // end class node
```

Fig. 26.1 | Self-referential `Node` class declaration.

© 2009 Pearson Education, Inc. All rights reserved.

26.3 Self-Referential Classes (Cont.)

- Self-referential objects can be linked together to form useful data structures.
- Figure 26.2 illustrates two self-referential objects linked together to form a linked list.
- A backslash indicates a `null` link.

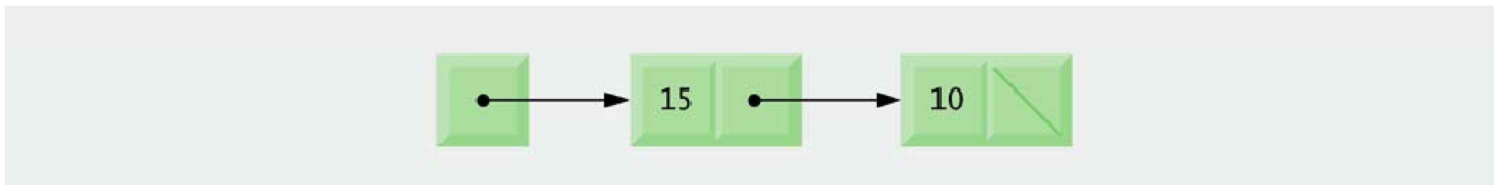

Fig. 26.2 | Self-referential class objects linked together.

Common Programming Error 26.1

Not setting the link in the last node of a list to `null` is a logic error.

© 2009 Pearson Education, Inc. All rights reserved.

26.3 Self-Referential Classes (Cont.)

- Creating and maintaining dynamic data structures requires **dynamic memory allocation**.
- Programs obtain more memory space to hold new nodes and to release space no longer needed.

```
Node nodeToAdd = new Node( 10 );
```

- If no memory is available, `new` throws an `OutOfMemoryException`.

Good Programming Practice 26.1

When creating a large number of objects, test for an `OutOfMemoryException`. Perform appropriate error processing if the requested memory is not allocated.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists

- A **linked list** is a linear collection (i.e., a sequence) of **nodes**, connected by reference links.
- A linked list is appropriate when the number of data elements is unpredictable.
- Linked lists become full only when the system has insufficient memory.

© 2009 Pearson Education, Inc. All rights reserved.

- The program of Figs. 26.4 and 26.5 uses an object of class `List` to manipulate a list of miscellaneous object types.
- You should place the code of Fig. 26.4 in its own class library project.


```
1  // Fig. 26.4: LinkedListLibrary.cs
2  // ListNode, List and EmptyListException class declarations.
3  using System;
4
5  namespace LinkedListLibrary
6  {
7     // class to represent one node in a list
8     class ListNode
9     {
10       // automatic read-only property Data
11       public object Data { get; private set; }
12
13       // automatic property Next
14       public ListNode Next { get; set; }
15
```

`Data` can refer to any object.

`Next` stores a reference to the next `ListNode` object in the linked list.

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 1 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

```
16       // constructor to create ListNode that refers to dataValue
17       // and is last node in list
18       public ListNode( object dataValue )
19          : this( dataValue, null )
20       {
21       } // end default constructor
22
23       // constructor to create ListNode that refers to dataValue
24       // and refers to next ListNode in List
25       public ListNode( object dataValue, ListNode nextNode )
26       {
27          Data = dataValue;
28          Next = nextNode;
29       } // end constructor
30    } // end class ListNode
31
```

Overriding the default constructor to place a `ListNode` at the end of a `List`

Constructor which places a `ListNode` before a specific `ListNode`.

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 2 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

```
32    // class List declaration
33    public class List
34    {
35       private ListNode firstNode;
36       private ListNode lastNode;
37       private string name; // string like "list" to display
38
39       // construct empty List with specified name
40       public List( string listName )
41       {
42          name = listName;
43          firstNode = lastNode = null;
44       } // end constructor
45
46       // construct empty List with "list" as its name
47       public List()
48          : this( "list" )
49       {
50       } // end default constructor
51
```

LinkedListLibrary.cs

`private` instance variables `firstNode` and `lastNode` are endpoints of a linked list.

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 3 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

```
52       // Insert object at front of List. If List is empty,
53       // firstNode and lastNode will refer to same object.
54       // Otherwise, firstNode refers to new node.
55       public void InsertAtFront( object insertItem )
56       {
57          if ( IsEmpty() )
58             firstNode = lastNode = new ListNode( insertItem );
59          else
60             firstNode = new ListNode( insertItem, firstNode );
61       } // end method InsertAtFront
62
63       // Insert object at end of List. If List is empty,
64       // firstNode and lastNode will refer to same object.
65       // Otherwise, lastNode's Next property refers to new node.
66       public void InsertAtBack( object insertItem )
67       {
68          if ( IsEmpty() )
69             firstNode = lastNode = new ListNode( insertItem );
70          else
71             lastNode = lastNode.Next = new ListNode( insertItem );
72       } // end method InsertAtBack
73
```

`InsertAtFront` manipulates the `List` (this process is explained later)

`InsertAtBack` manipulates the `List` (this process is explained later)

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 4 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

```
74        // remove first node from List
75        public object RemoveFromFront()
76        {
77           if ( IsEmpty() )
78              throw new EmptyListException( name );
79
80           object removeItem = firstNode.Data; // retrieve data
81
82           // reset firstNode and lastNode references
83           if ( firstNode == lastNode )
84              firstNode = lastNode = null;
85           else
86              firstNode = firstNode.Next;
87
88           return removeItem; // return removed data
89        } // end method RemoveFromFront
90
91        // remove last node from List
92        public object RemoveFromBack()
93        {
94           if ( IsEmpty() )
95              throw new EmptyListException( name );
96
```

LinkedListLibrary.cs

(5 of 9)

RemoveFromFront manipulates the List (this process is explained later)

RemoveFromBack manipulates the List (this process is explained later)

Fig. 26.4 | ListNode, List and EmptyListException class declarations. (Part 5 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

```
97          object removeItem = lastNode.Data; // retrieve data
98
99          // reset firstNode and lastNode references
100         if ( firstNode == lastNode )
101            firstNode = lastNode = null;
102         else
103         {
104            ListNode current = firstNode;
105
106            // loop while current node is not lastNode
107            while ( current.Next != lastNode )
108               current = current.Next; // move to next node
109
110            // current is new lastNode
111            lastNode = current;
112            current.Next = null;
113         } // end else
114
115         return removeItem; // return removed data
116      } // end method RemoveFromBack
117
```

`LinkedListLibrary.cs`

(6 of 9)

`RemoveFromBack` manipulates the `List` (this process is explained later)

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 6 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

```
118     // return true if List is empty
119     public bool IsEmpty()
120     {
121        return firstNode == null;
122     } // end method IsEmpty
123
124     // output List contents
125     public void Display()
126     {
127        if ( IsEmpty() )
128        {
129           Console.WriteLine( "Empty " + name );
130        } // end if
131        else
132        {
133           Console.Write( "The " + name + " is: " );
134
135           ListNode current = firstNode;
136
```

`LinkedListLibrary.cs`

(7 of 9)

`IsEmpty` determines whether the list is empty.

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 7 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

LinkedListLibrary.cs

(8 of 9)

```
137             // output current node data while not at end of list
138             while ( current != null )
139             {
140                Console.Write( current.Data + " " );
141                current = current.Next;
142             } // end while
143
144             Console.WriteLine( "\n" );
145          } // end else
146       } // end method Display
147    } // end class List
148
149    // class EmptyListException declaration
150    public class EmptyListException : Exception
151    {
152       // parameterless constructor
153       public EmptyListException()
154          : base( "The list is empty" )
155       {
156          // empty constructor
157       } // end EmptyListException constructor
```

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 8 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

LinkedListLibrary .cs

(9 of 9)

```
158
159       // one-parameter constructor
160       public EmptyListException( string name )
161          : base( "The " + name + " is empty" )
162       {
163          // empty constructor
164       } // end EmptyListException constructor
165
166       // two-parameter constructor
167       public EmptyListException( string exception, Exception inner )
168          : base( exception, inner )
169       {
170          // empty constructor
171       } // end EmptyListException constructor
172    } // end class EmptyListException
173} // end namespace LinkedListLibrary
```

Fig. 26.4 | `ListNode`, `List` and `EmptyListException` class declarations. (Part 9 of 9.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

- Class `ListTest` (Fig. 26.5) uses the linked-list library to create and manipulate a linked list.
- You must add a reference to the class library containing the classes in Fig. 26.4.

`ListTest.cs`


```
1  // Fig. 26.5: ListTest.cs
2  // Testing class List.
3  using System;
4  using LinkedListLibrary;
5
6  // class to test List class functionality
7  class ListTest
8  {
9     public static void Main( string[] args )
10    {
11       List list = new List(); // create List container
12
13       // create data to store in List
14       bool aBoolean = true;
15       char aCharacter = '$';
16       int anInteger = 34567;
17       string aString = "hello";
```

Initializing a new `List` object.

Fig. 26.5 | Testing class `List`. (Part 1 of 4.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

ListTest.cs

(2 of 4)

```
18
19         // use List insert methods
20         list.InsertAtFront( aBoolean );
21         list.Display();
22         list.InsertAtFront( aCharacter );
23         list.Display();
24         list.InsertAtBack( anInteger );
25         list.Display();
26         list.InsertAtBack( aString );
27         list.Display();
28
29         // use List remove methods
30         object removedObject;
31
32         // remove data from list and display after each removal
33         try
34         {
35            removedObject = list.RemoveFromFront();
36            Console.WriteLine( removedObject + " removed" );
37            list.Display();
38
```

Demonstrating List insertion methods.

Removing objects via List deletion methods.

Fig. 26.5 | Testing class List. (Part 2 of 4.)

© 2009 Pearson Education, Inc. All rights reserved.

Outline

ListTest.cs

(3 of 4)

```
39            removedObject = list.RemoveFromFront();
40            Console.WriteLine( removedObject + " removed" );
41            list.Display();
42
43            removedObject = list.RemoveFromBack();
44            Console.WriteLine( removedObject + " removed" );
45            list.Display();
46
47            removedObject = list.RemoveFromBack();
48            Console.WriteLine( removedObject + " removed" );
49            list.Display();
50         } // end try
51         catch ( EmptyListException emptyListException )
52         {
53            Console.Error.WriteLine( "\n" + emptyListException );
54         } // end catch
55      } // end Main
56   } // end class ListTest
```

Removing objects via `List` deletion methods.

Fig. 26.5 | Testing class `List`. (Part 3 of 4.)

© 2009 Pearson Education, Inc. All rights reserved.

```
The list is: True
The list is: $ True
The list is: $ True 34567
The list is: $ True 34567 hello
$ removed
The list is: True 34567 hello
True removed
The list is: 34567 hello
```

```
hello removed
The list is: 34567
34567 removed
Empty list
```

Fig. 26.5 | Testing class `List`. (Part 4 of 4.)

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- Method `InsertAtFront` places a new node at the front of the list through the following process:

1. Determine whether the list is empty.
2. If the list is empty, set both `firstNode` and `lastNode` to refer to a new `ListNode`.
3. If the list is not empty, the new node is "linked" into the list by setting `firstNode` to refer to a new `ListNode`.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- In Fig. 26.6, part (a) shows a list and a new node during the `InsertAtFront` operation and before the new node is linked into the list.
- Part (b) illustrates *Step 3* of the `InsertAtFront` operation, which enables the node containing `12` to become the new list front.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

(a) firstNode

7

11

new ListNode

12

(b) firstNode

7

11

new ListNode

12

Fig. 26.6 | `InsertAtFront` operation.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- Method `InsertAtBack` places a new node at the back of the list through the following process:

1. Determine whether the list is empty.
2. If the list is empty, set both `firstNode` and `lastNode` to refer to a new `ListNode`.
3. If the list is not empty, link the new node into the list by setting `lastNode` and `lastNode.Next` to refer to a new `ListNode` object.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- In Fig. 26.7, part (a) shows a list and a new node during the `InsertAtBack` operation; before the new node has been linked into the list.
- The dashed lines and arrows in part (b) illustrate *Step 3* of method `Insert-AtBack`, which enables a new node to be added to the end of a list that is not empty.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

(a) firstNode lastNode new ListNode

12 7 11 5

(b) firstNode lastNode new ListNode

12 7 11 5

Fig. 26.7 | `InsertAtBack` operation.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- Method `RemoveFromFront` removes the front node of the list and returns a reference to the removed data.

1. Assign `firstNode.Data` to variable `removeItem`.
2. If `firstNode` and `lastNode` are the same object, the method sets `firstNode` and `lastNode` to `null`.
3. If the list has more than one node, the method assigns `firstNode.Next` to `firstNode`.
4. Return the `removeItem` reference.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- In Fig. 26.8, part (a) illustrates a list before a removal operation.
- Part (b) shows the reference manipulations.

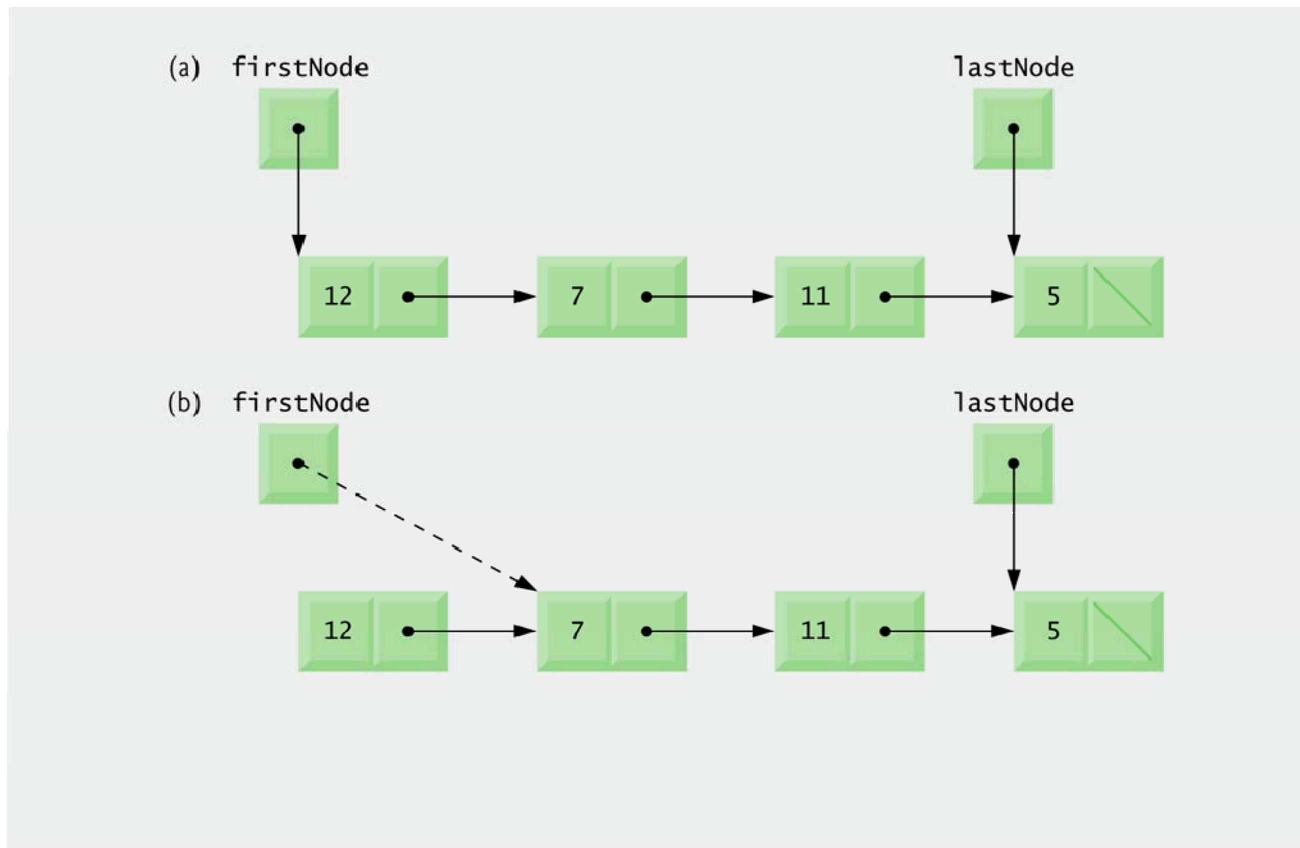

Fig. 26.8 | `RemoveFromFront` operation.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- Method `RemoveFromBack` removes the last node of a list and returns a reference to the removed data.

1. Assign `lastNode.Data` to variable `removeItem`.
2. If `firstNode` and `lastNode` refer to the same object, set `firstNode` and `lastNode` to `null`.
3. If the list has more than one node, create `ListNode` variable `current` and assign it `firstNode`.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

4. Now “walk the list” with `current` until it references the node before the last node.
5. After locating the second-to-last node, assign `current` to `lastNode`.
6. Set `current.Next` to `null` to remove the last node from the list.
7. Return the `removeItem` reference.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- In Fig. 26.9, part (a) illustrates a list before a removal operation.
- Part (b) shows the reference manipulations.

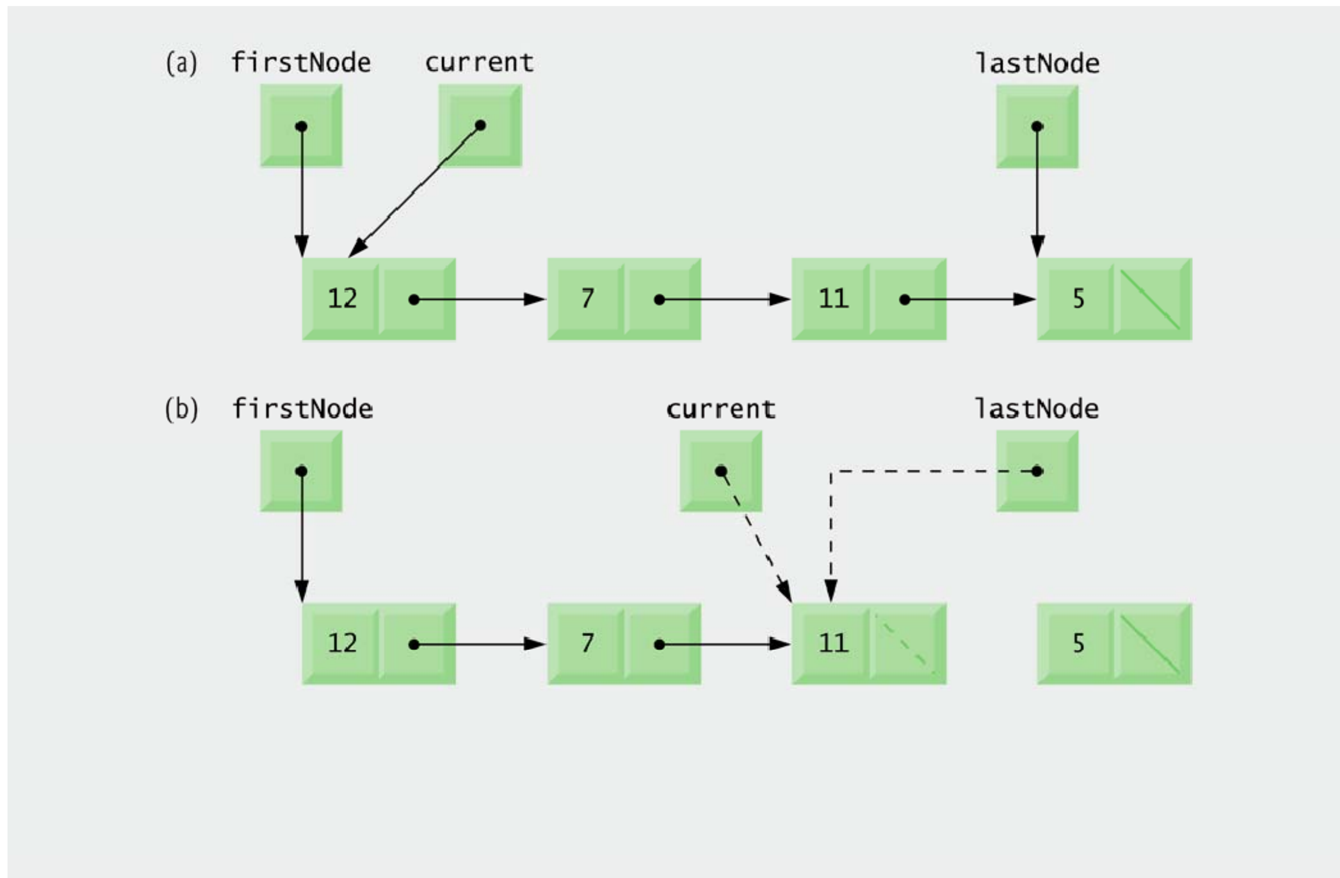

Fig. 26.9 | `RemoveFromBack` operation.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- A **singly linked list** may be traversed in only one direction.
- In a **circular, singly linked list** (Fig. 26.10), the "last node" points back to the first node.

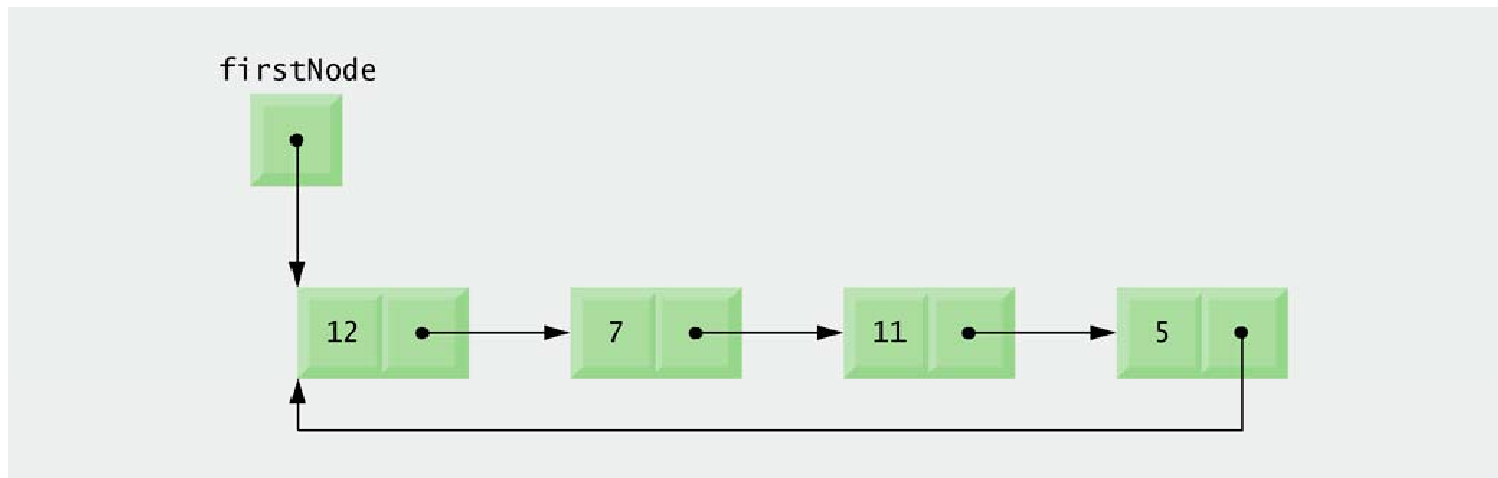

Fig. 26.10 | Circular, singly linked list.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- A **doubly linked list** (Fig. 26.11) allows traversals both forward and backward.
- Each node has both a forward reference to the next node in the list and a backward reference to the previous node.

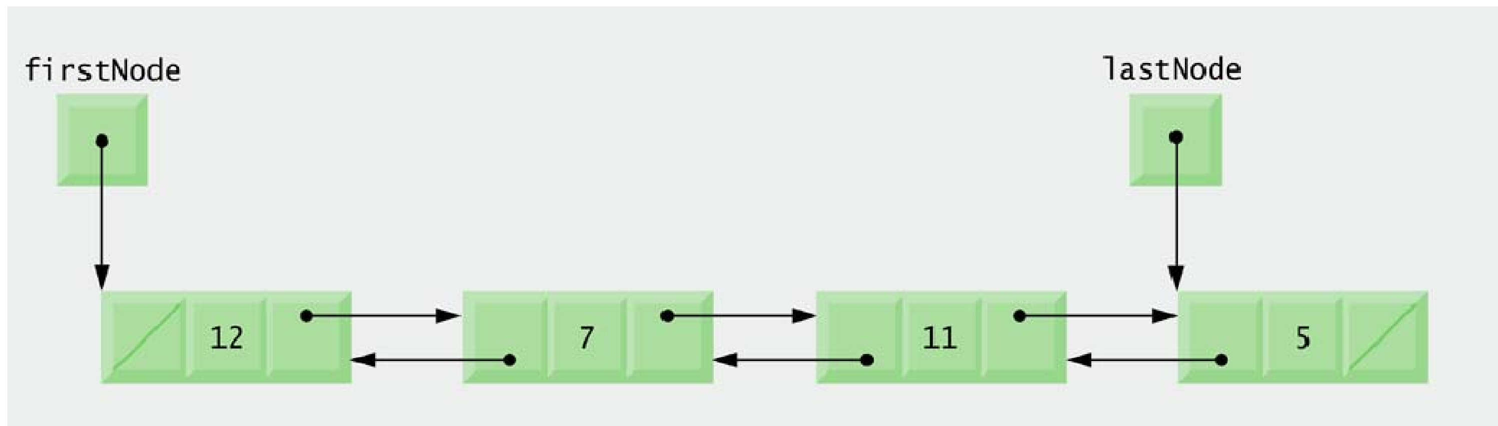

Fig. 26.11 | Doubly linked list.

© 2009 Pearson Education, Inc. All rights reserved.

26.4 Linked Lists (Cont.)

- In a **circular, doubly linked list** (Fig. 26.12), the forward reference of the last node refers to the first node, and the backward reference of the first node refers to the last node.

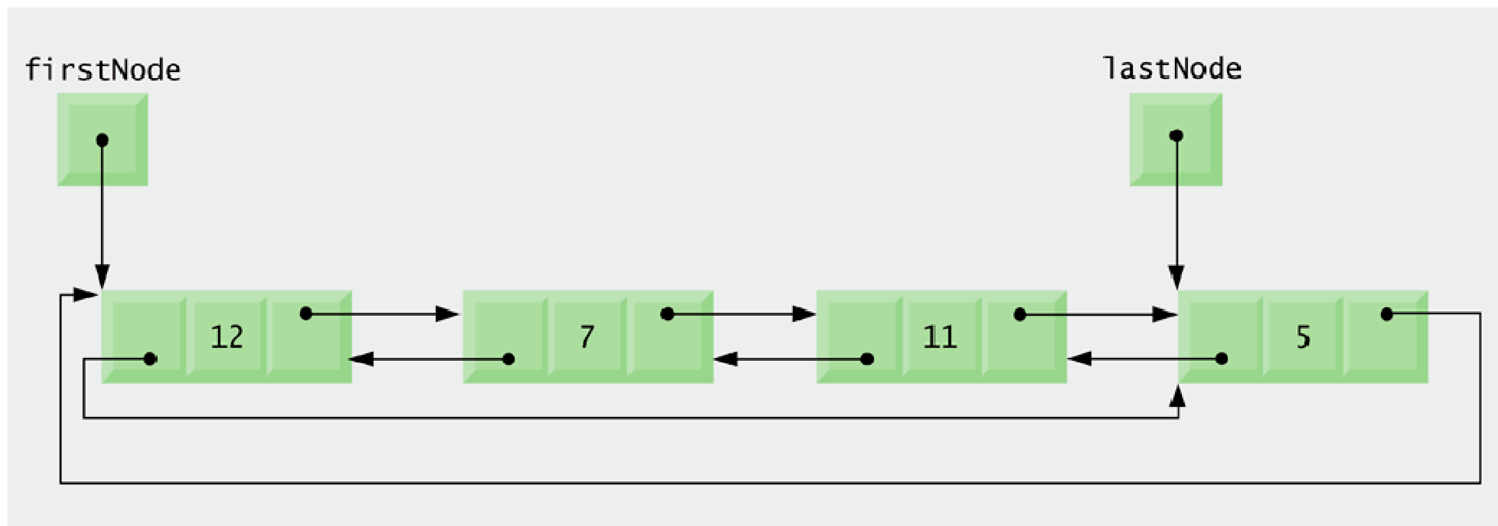

Fig. 26.12 | Circular, doubly linked list.

© 2009 Pearson Education, Inc. All rights reserved.