

26

Data Structures



26.5 Stacks

- A **stack** is referred to as a **last-in, first-out (LIFO)** data structure.
- Operation **push** adds a new node to the top of the stack.
- Operation **pop** removes a node from the top of the stack and returns the data item.



26.6 Queues

- A queue is similar to a checkout line in a supermarket.
- Queue nodes are removed only from the head (or front) of the queue and are inserted only at the tail (or end).
 - A queue is a **first-in, first-out (FIFO)** data structure.
 - The insert and remove operations are known as **enqueue** and **dequeue**.



26.7 Trees

- Linked lists, stacks and queues are **linear data structures** (i.e., **sequences**).
- A **tree** is a nonlinear, two-dimensional data structure.
- **Binary trees** (Fig. 26.18) have nodes which contains two links.



26.7 Trees (Cont.)

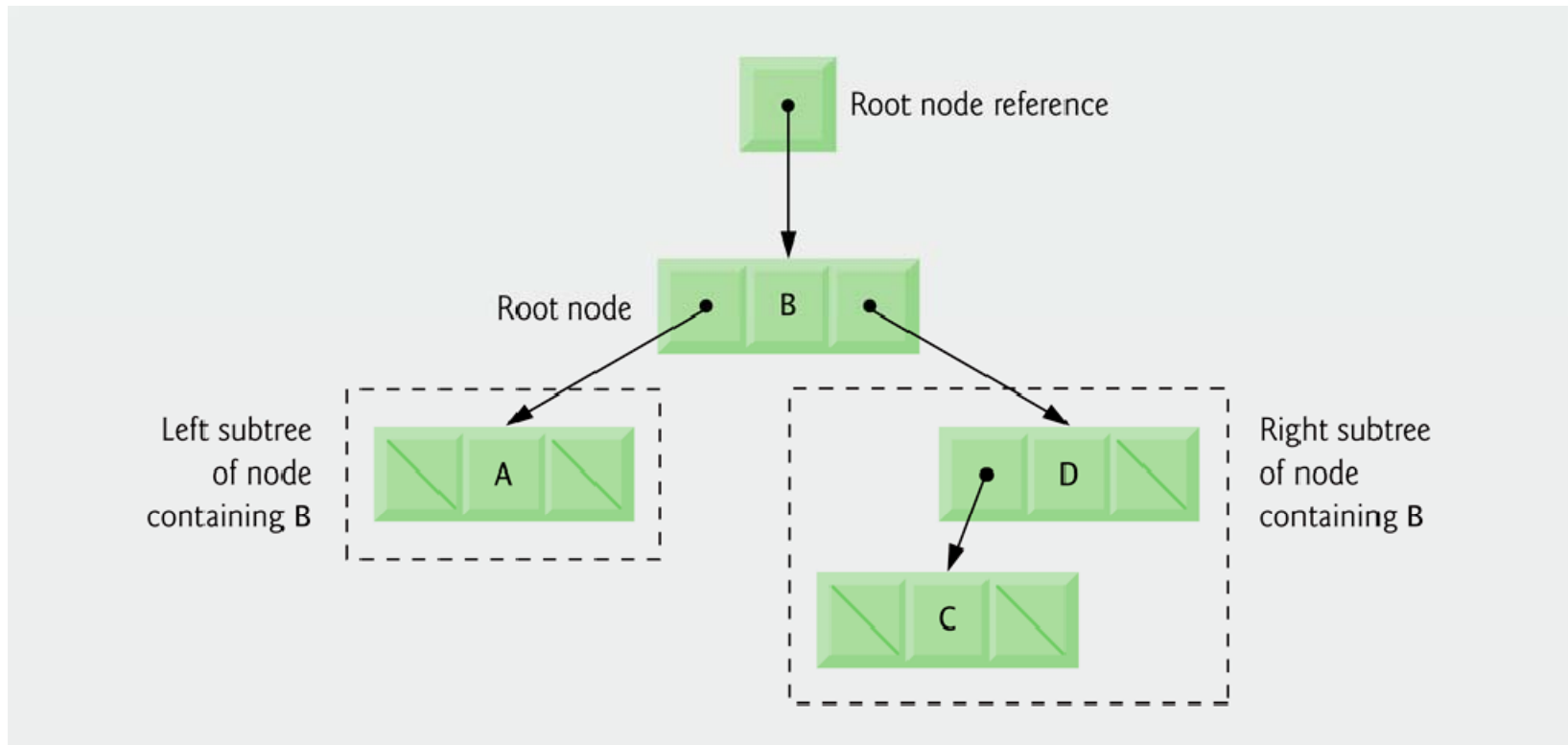


Fig. 26.18 | Binary-tree graphical representation.



26.7 Trees (Cont.)

- The **root node** is the first node in a binary tree.
- The **left child** is the first node in the **left subtree**, and the **right child** is the first node in the **right subtree**.
- The children of a specific node are called **siblings**.
- A node with no children is called a **leaf node**.

Common Programming Error 26.2

Not setting to null the links in leaf nodes of a tree is a common logic error.



26.7 Trees (Cont.)

- A **binary search tree** (Fig. 26.19) is organized such that:
 - the values in any left subtree are less than the value in the subtree's **parent node**
 - the values in any right subtree are greater than the value in the subtree's parent node.

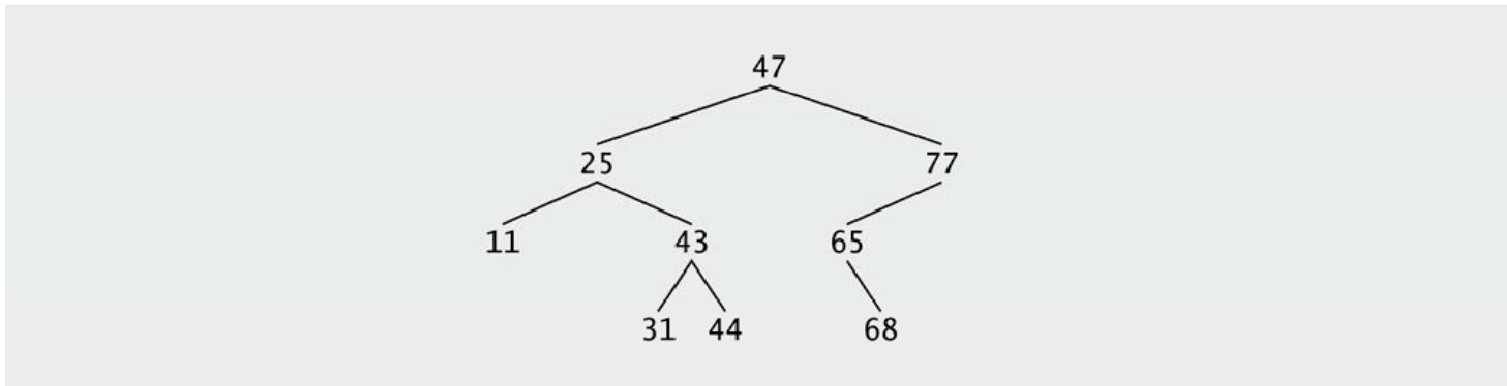


Fig. 26.19 | Binary search tree containing 9 values.



Outline

- The application of Figs. 26.20 and 26.21 creates a binary search tree and traverses it using recursive **inorder**, **preorder** and **postorder** traversals.

BinaryTreeLibrary.cs

(1 of 7)

```
1 // Fig. 26.20: BinaryTreeLibrary.cs
2 // Declaration of class TreeNode and class Tree.
3 using System;
4
5 namespace BinaryTreeLibrary
6 {
7     // class TreeNode declaration
8     class TreeNode
9     {
10         // automatic property LeftNode
11         public TreeNode LeftNode { get; set; }
12
13         // automatic property Data
14         public int Data { get; set; }
15
16         // automatic property RightNode
17         public TreeNode RightNode { get; set; }
18
```

Fig. 26.20 | Declaration of class TreeNode and class Tree. (Part 1 of 7.)



Outline

```

19 // initialize Data and make this a leaf node
20 public TreeNode( int nodeData )
21 {
22     Data = nodeData;
23     LeftNode = RightNode = null; // node has no children
24 } // end constructor
25
26 // insert TreeNode into Tree that contains nodes;
27 // ignore duplicate values
28 public void Insert( int insertValue )
29 {
30     if ( insertValue < Data ) // insert in left subtree
31     {
32         // insert new TreeNode
33         if ( LeftNode == null )
34             LeftNode = new TreeNode( insertValue );
35         else // continue traversing left subtree
36             LeftNode.Insert( insertValue );
37     } // end if

```

BinaryTree
Library.cs

(2 of 7)

Initially, every TreeNode is a leaf node with null children.

Insert recursively determines the location for the new node in the binary search tree.

Fig. 26.20 | Declaration of class TreeNode and class Tree. (Part 2 of 7.)



Outline

```

38     else if ( insertValue > Data ) // insert in right subtree
39     {
40         // insert new TreeNode
41         if ( RightNode == null )
42             RightNode = new TreeNode( insertValue );
43         else // continue traversing right subtree
44             RightNode.Insert( insertValue );
45     } // end else if
46 } // end method Insert
47 } // end class TreeNode
48
49 // class Tree declaration
50 public class Tree
51 {
52     private TreeNode root;
53
54     // construct an empty Tree of integers
55     public Tree()
56     {
57         root = null;
58     } // end constructor

```

BinaryTree
Library.cs

(3 of 7)

Insert recursively determines the location for the new node in the binary search tree.

root stores the root node of the tree.

Fig. 26.20 | Declaration of class TreeNode and class Tree. (Part 3 of 7.)



Outline

BinaryTree Library.cs

(4 of 7)

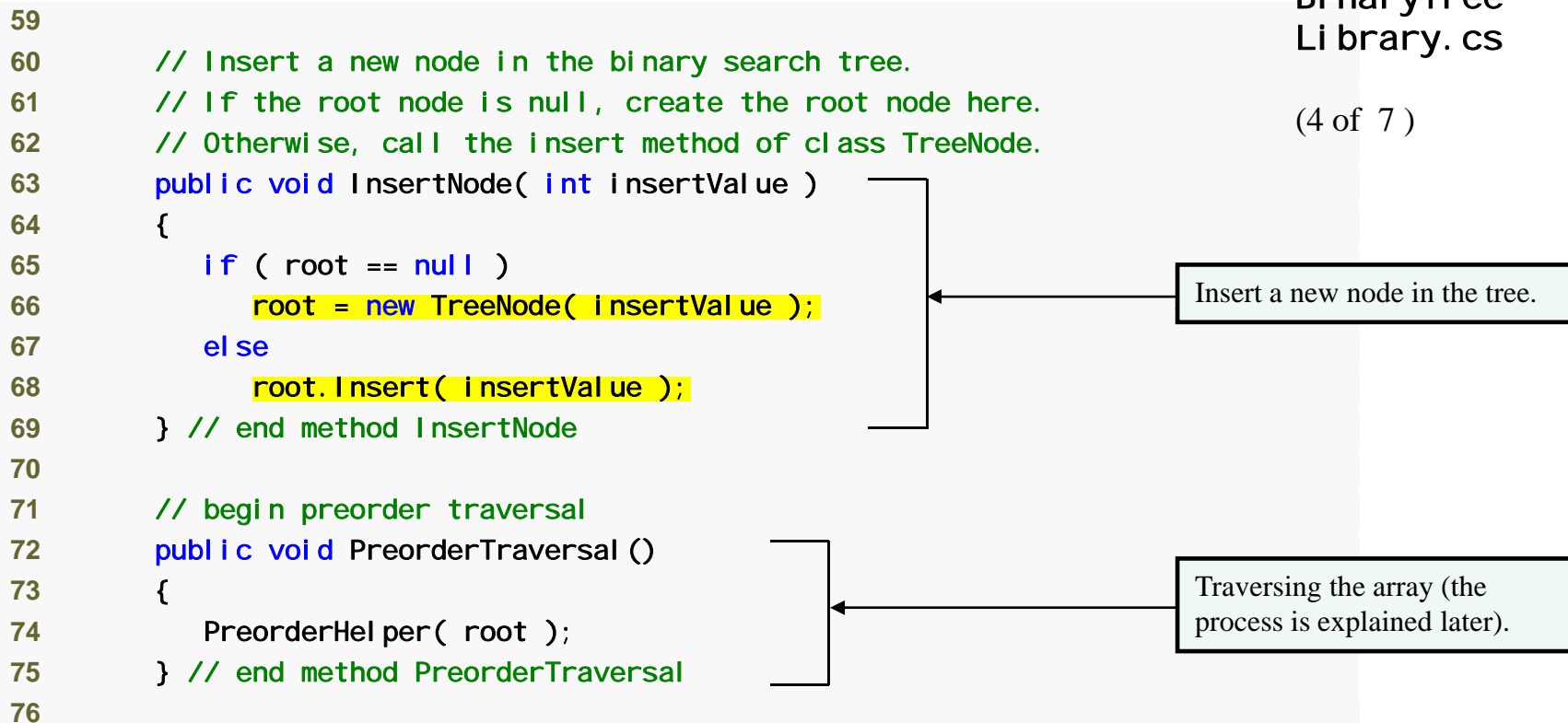


Fig. 26.20 | Declaration of class TreeNode and class Tree. (Part 4 of 7.)



Outline

BinaryTree
Library.cs

(5 of 7)

```

77 // recursive method to perform preorder traversal
78 private void PreorderHelper( TreeNode node )
79 {
80     if ( node != null )
81     {
82         // output node Data
83         Console.WriteLine( node.Data + " " );
84
85         // traverse left subtree
86         PreorderHelper( node.LeftNode );
87
88         // traverse right subtree
89         PreorderHelper( node.RightNode );
90     } // end if
91 } // end method PreorderHelper
92
93 // begin inorder traversal
94 public void InorderTraversal ()
95 {
96     InorderHelper( root );
97 } // end method InorderTraversal
98

```

Traversing the array (the process is explained later).

Fig. 26.20 | Declaration of class TreeNode and class Tree. (Part 5 of 7.)



Outline

```

99 // recursive method to perform inorder traversal
100 private void InorderHelper( TreeNode node )
101 {
102     if ( node != null )
103     {
104         // traverse left subtree
105         InorderHelper( node.LeftNode );
106
107         // output node data
108         Console.WriteLine( node.Data + " " );
109
110         // traverse right subtree
111         InorderHelper( node.RightNode );
112     } // end if
113 } // end method InorderHelper
114
115 // begin postorder traversal
116 public void PostorderTraversal ()
117 {
118     PostorderHelper( root );
119 } // end method PostorderTraversal
120

```

BinaryTree
Library.cs

(6 of 7)

Traversing the array (the process is explained later).

Fig. 26.20 | Declaration of class TreeNode and class Tree. (Part 6 of 7.)



Outline

BinaryTree
Library.cs

(7 of 7)

```
121 // recursive method to perform postorder traversal
122 private void PostorderHelper( TreeNode node )
123 {
124     if ( node != null )
125     {
126         // traverse left subtree
127         PostorderHelper( node.LeftNode );
128
129         // traverse right subtree
130         PostorderHelper( node.RightNode );
131
132         // output node Data
133         Console.WriteLine( node.Data + " " );
134     } // end if
135 } // end method PostorderHelper
136 } // end class Tree
137 } // end namespace BinaryTreeLibrary
```

Fig. 26.20 | Declaration of class TreeNode and class Tree. (Part 7 of 7.)



Outline

TreeTest.cs

(1 of 3)

```
1 // Fig. 26.21: TreeTest.cs
2 // Testing class Tree with a binary tree.
3 using System;
4 using BinaryTreeLibrary;
5
6 // class TreeTest declaration
7 public class TreeTest
8 {
9     // test class Tree
10    public static void Main( string[] args )
11    {
12        Tree tree = new Tree();
13        int insertValue;
14
15        Console.WriteLine( "Inserting values: " );
16        Random random = new Random();
17
```

Fig. 26.21 | Testing class Tree with a binary tree. (Part 1 of 3.)



Outline

```
18 // insert 10 random integers from 0-99 in tree
19 for ( int i = 1; i <= 10; i++ )
20 {
21     insertValue = random.Next( 100 );
22     Console.Wri te( insertValue + " " );
23
24     tree.InsertNode( insertValue );
25 } // end for
26
27 // perform preorder traversal of tree
28 Console.Wri teLi ne( "\n\nPreorder traversal " );
29 tree.PreorderTraversal ();
30
31 // perform inorder traversal of tree
32 Console.Wri teLi ne( "\n\nInorder traversal " );
33 tree.InorderTraversal ();
34
35 // perform postorder traversal of tree
36 Console.Wri teLi ne( "\n\nPostorder traversal " );
37 tree.PostorderTraversal ();
38 Console.Wri teLi ne();
39 } // end Mai n
40 } // end class TreeTest
```

TreeTest.cs

(2 of 3)

Fig. 26.21 | Testing class Tree with a binary tree. (Part 2 of 3.)



Outline

TreeTest.cs

(3 of 3)

Inserting values:

39 69 94 47 50 72 55 41 97 73

Preorder traversal

39 69 47 41 50 55 94 72 73 97

Inorder traversal

39 41 47 50 55 69 72 73 94 97

Postorder traversal

41 55 50 47 73 72 97 94 69 39

Fig. 26.21 | Testing class Tree with a binary tree. (Part 3 of 3.)



26.7 Trees (Cont.)

- Consider the binary search tree shown in Fig. 26.22.

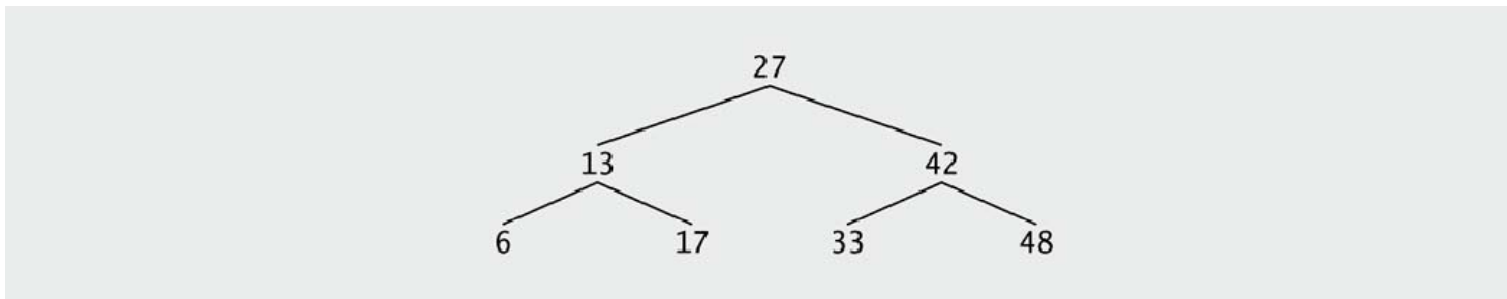


Fig. 26.22 | Binary search tree



26.7 Trees (Cont.)

- The inorder traversal of a binary search tree displays the node values in ascending order.
- Method `InorderHelper` defines the steps for an inorder traversal:
 1. If the argument is `null`, do not process the tree.
 2. Traverse the left subtree with a call to `InorderHelper`.
 3. Process the value in the node.
 4. Traverse the right subtree with a call to `InorderHelper`.



26.7 Trees (Cont.)

- The preorder traversal processes the value in each node, then the values in the left subtree, then the values in the right subtree.
- Method `PreorderHelper` defines the steps for a preorder traversal:
 1. If the argument is `null`, do not process the tree.
 2. Process the value in the node.
 3. Traverse the left subtree with a call to `PreorderHelper`.
 4. Traverse the right subtree with a call to `PreorderHelper`.



26.7 Trees (Cont.)

- The postorder traversal processes the value in each node after the values of all that node's children are processed.
- Method `PostorderHelper` defines the steps for a postorder traversal:
 1. If the argument is `null`, do not process the tree.
 2. Traverse the left subtree with a call to `PostorderHelper`.
 3. Traverse the right subtree with a call to `PostorderHelper`.
 4. Process the value in the node.



26.7 Trees (Cont.)

- A binary search tree facilitates **duplicate elimination**.
- The insertion operation recognizes a duplicate value, because a duplicate is placed the same as the original value.
- Searching a binary tree for a value that matches a key value is fast, especially for **tightly packed** binary trees.
- In a tightly packed binary tree, each level contains about twice as many elements as the previous level.



26.7 Trees (Cont.)

26.7.2 Binary Search Tree of Comparable Objects

- We would like to define the functionality of a binary tree once and reuse it for many data types.
- Classes that implement Comparable define method `compareTo`, which compares the object that invokes the method with the object received as an argument.



Outline

- The program of Figs. 26.23 and 26.24 enhances the program to manipulate `IComparable` objects.

BinaryTree
Library2. cs

(1 of 7)

```

1 // Fig. 26.23: BinaryTreeLibrary2.cs
2 // Declaration of class TreeNode and class Tree.
3 using System;
4
5 namespace BinaryTreeLibrary2
6 {
7     // class TreeNode declaration
8     class TreeNode
9     {
10        // automatic property LeftNode
11        public TreeNode LeftNode { get; set; }
12
13        // automatic property Data
14        public IComparable Data { get; set; }
15
16        // automatic property RightNode
17        public TreeNode RightNode { get; set; }
18

```

A few lines are changed to switch to use `IComparable` objects.

Fig. 26.23 | Declaration of class `TreeNode` and class `Tree`. (Part 1 of 7.)



Outline

BinaryTree Library2. cs

(2 of 7)

```

19 // initialize Data and make this a leaf node
20 public TreeNode( IComparable nodeData )
21 {
22     Data = nodeData;
23     LeftNode = RightNode = null; // node has no children
24 } // end constructor
25
26 // insert TreeNode into Tree that contains nodes;
27 // ignore duplicate values
28 public void Insert( IComparable insertValue )
29 {
30     if ( insertValue.CompareTo(Data) < 0 ) // insert in left subtree
31     {
32         // insert new TreeNode
33         if ( LeftNode == null )
34             LeftNode = new TreeNode( insertValue );
35         else // continue traversing left subtree
36             LeftNode.Insert( insertValue );
37     } // end if

```

Method CompareTo is now used to place items in the tree.

Fig. 26.23 | Declaration of class TreeNode and class Tree. (Part 2 of 7.)



Outline

```
38     else if ( insertValue.CompareTo( Data ) > 0 ) // insert in right
39     {
40         // insert new TreeNode
41         if ( RightNode == null )
42             RightNode = new TreeNode( insertValue );
43         else // continue traversing right subtree
44             RightNode.Insert( insertValue );
45     } // end else if
46 } // end method Insert
47 } // end class TreeNode
48
49 // class Tree declaration
50 public class Tree
51 {
52     private TreeNode root;
53
54     // construct an empty Tree of integers
55     public Tree()
56     {
57         root = null;
58     } // end constructor
```

BinaryTree
Library2.cs

(3 of 7)

Fig. 26.23 | Declaration of class TreeNode and class Tree. (Part 3 of 7.)



Outline

BinaryTree Library2. cs

(4 of 7)

```
59
60 // Insert a new node in the binary search tree.
61 // If the root node is null, create the root node here.
62 // Otherwise, call the insert method of class TreeNode.
63 public void InsertNode( IComparable insertValue )
64 {
65     if ( root == null )
66         root = new TreeNode( insertValue );
67     else
68         root.Insert( insertValue );
69 } // end method InsertNode
70
71 // begin preorder traversal
72 public void PreorderTraversal ()
73 {
74     PreorderHelper( root );
75 } // end method PreorderTraversal
76
```

Fig. 26.23 | Declaration of class TreeNode and class Tree. (Part 4 of 7.)



Outline

BinaryTree
Library2.cs

(5 of 7)

```

77 // recursive method to perform preorder traversal
78 private void PreorderHelper( TreeNode node )
79 {
80     if ( node != null )
81     {
82         // output node Data
83         Console.WriteLine( node.Data + " " );
84
85         // traverse left subtree
86         PreorderHelper( node.LeftNode );
87
88         // traverse right subtree
89         PreorderHelper( node.RightNode );
90     } // end if
91 } // end method PreorderHelper
92
93 // begin inorder traversal
94 public void InorderTraversal ()
95 {
96     InorderHelper( root );
97 } // end method InorderTraversal
98

```

Fig. 26.23 | Declaration of class TreeNode and class Tree. (Part 5 of 7.)



Outline

BinaryTree
Library2.cs

(6 of 7)

```

99 // recursive method to perform inorder traversal
100 private void InorderHelper( TreeNode node )
101 {
102     if ( node != null )
103     {
104         // traverse left subtree
105         InorderHelper( node.LeftNode );
106
107         // output node data
108         Console.WriteLine( node.Data + " " );
109
110         // traverse right subtree
111         InorderHelper( node.RightNode );
112     } // end if
113 } // end method InorderHelper
114
115 // begin postorder traversal
116 public void PostorderTraversal ()
117 {
118     PostorderHelper( root );
119 } // end method PostorderTraversal

```

Fig. 26.23 | Declaration of class TreeNode and class Tree. (Part 6 of 7.)



Outline

BinaryTree
Library2. cs

(7 of 7)

```
120
121 // recursive method to perform postorder traversal
122 private void PostorderHelper( TreeNode node )
123 {
124     if ( node != null )
125     {
126         // traverse left subtree
127         PostorderHelper( node.LeftNode );
128
129         // traverse right subtree
130         PostorderHelper( node.RightNode );
131
132         // output node Data
133         Console.WriteLine( node.Data + " " );
134     } // end if
135 } // end method PostorderHelper
136 } // end class Tree
137} // end namespace BinaryTreeLibrary
```

Fig. 26.23 | Declaration of class TreeNode and class Tree. (Part 7 of 7.)



Outline

- Class `TreeTest` (Fig. 26.24) creates `Tree` objects to store `int`, `double` and `string` values.
- The `Tree` of `strings` is stored in alphabetical order.

`TreeTest.cs`

(1 of 5)

```
1 // Fig. 26.24: TreeTest.cs
2 // Testing class Tree with IComparable objects.
3 using System;
4 using BinaryTreeLibrary2;
5
6 // class TreeTest declaration
7 public class TreeTest
8 {
9     // test class Tree
10    public static void Main( string[] args )
11    {
12        int[] intArray = { 8, 2, 4, 3, 1, 7, 5, 6 };
13        double[] doubleArray = { 8.8, 2.2, 4.4, 3.3, 1.1, 7.7, 5.5, 6.6 };
14        string[] stringArray = { "eight", "two", "four",
15                                "three", "one", "seven", "five", "six" };
16    }
```

Fig. 26.24 | Testing class `Tree` with `IComparable` objects. (Part 1 of 5.)



Outline

TreeTest.cs

(2 of 5)

```

17 // create int Tree
18 Tree intTree = new Tree();
19 populateTree( intArray, intTree, "IntTree" );
20 traverseTree( intTree, "IntTree" );
21
22 // create double Tree
23 Tree doubleTree = new Tree();
24 populateTree( doubleArray, doubleTree, "doubleTree" );
25 traverseTree( doubleTree, "doubleTree" );
26
27 // create string Tree
28 Tree stringTree = new Tree();
29 populateTree( stringArray, stringTree, "stringTree" );
30 traverseTree( stringTree, "stringTree" );
31 } // end Main
32
33 // populate Tree with array elements
34 private static void populateTree( Array array, Tree tree, string name )
35 {
36     Console.WriteLine( "\n\nInserting into " + name + ":" );
37

```

PopulateTree adds an Array of values to a Tree.

Fig. 26.24 | Testing class Tree with I Comparable objects. (Part 2 of 5.)



Outline

```

38     foreach ( IComparable data in array )
39     {
40         Console.Wri te( data + " " );
41         tree.InsertNode( data );
42     } // end foreach
43 } // end method populateTree
44
45 // insert perform traversals
46 private static void traverseTree( Tree tree, string treeType )
47 {
48     // perform preorder traversal of tree
49     Console.Wri teLi ne( "\n\nPreorder traversal of " + treeType );
50     tree.PreorderTraversal ();
51
52     // perform inorder traversal of tree
53     Console.Wri teLi ne( "\n\nInorder traversal of " + treeType );
54     tree.InorderTraversal ();
55
56     // perform postorder traversal of tree
57     Console.Wri teLi ne( "\n\nPostorder traversal of " + treeType );
58     tree.PostorderTraversal ();
59 } // end method traverseTree
60 } // end class TreeTest

```

TreeTest.cs

(3 of 5)

PopulateTree adds an Array of values to a Tree.

TraverseTree outputs the preorder, inorder and postorder traversals of the Tree.

Fig. 26.24 | Testing class Tree with I Comparable objects. (Part 3 of 5.)



Outline

TreeTest.cs

(4 of 5)

```
Inserting into intTree:
```

```
8 2 4 3 1 7 5 6
```

```
Preorder traversal of intTree
```

```
8 2 1 4 3 7 5 6
```

```
Inorder traversal of intTree
```

```
1 2 3 4 5 6 7 8
```

```
Postorder traversal of intTree
```

```
1 3 6 5 7 4 2 8
```

(continued on next page...)

Fig. 26.24 | Testing class Tree with IComparable objects. (Part 4 of 5.)



Outline*(continued from previous page...)*

Inserting into doubleTree:

8.8 2.2 4.4 3.3 1.1 7.7 5.5 6.6

Preorder traversal of doubleTree

8.8 2.2 1.1 4.4 3.3 7.7 5.5 6.6

Inorder traversal of doubleTree

1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8

Postorder traversal of doubleTree

1.1 3.3 6.6 5.5 7.7 4.4 2.2 8.8

Inserting into stringTree:

eight two four three one seven five six

Preorder traversal of stringTree

eight two four five three one seven six

Inorder traversal of stringTree

eight five four one seven six three two

Postorder traversal of stringTree

five six seven one three four two eight

TreeTest.cs

(5 of 5)

Fig. 26.24 | Testing class Tree with IComparable objects. (Part 5 of 5.)