CHAPTER

25

BINARY SEARCH TREES

Objectives

- To design and implement a binary search tree (§25.2).
- To represent binary trees using linked data structures (§25.2.1).
- To search an element in a binary search tree (§25.2.2).
- To insert an element into a binary search tree (§25.2.3).
- To traverse elements in a binary tree (§25.2.4).
- To design and implement the `Tree` interface, `AbstractTree` class, and the `BST` class (§25.2.5).
- To delete elements from a binary search tree (§25.3).
- To display a binary tree graphically (§25.4).
- To create iterators for traversing a binary tree (§25.5).
- To implement Huffman coding for compressing data using a binary tree (§25.6).

25.1 Introduction

A tree is a classic data structure with many important applications.

A *tree* provides a hierarchical organization in which data are stored in the nodes. This chapter introduces binary search trees. You will learn how to construct a binary search tree, how to search an element, insert an element, delete an element, and traverse elements in a binary search tree. You will also learn how to define and implement a custom data structure for a binary search tree.

25.2 Binary Search Trees

A binary search tree can be implemented using a linked structure.

binary tree
root
left subtree
right subtree

Recall that lists, stacks, and queues are linear structures that consist of a sequence of elements. A *binary tree* is a hierarchical structure. It either is empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*, either or both of which may be empty. Examples of binary trees are shown in Figure 25.1.

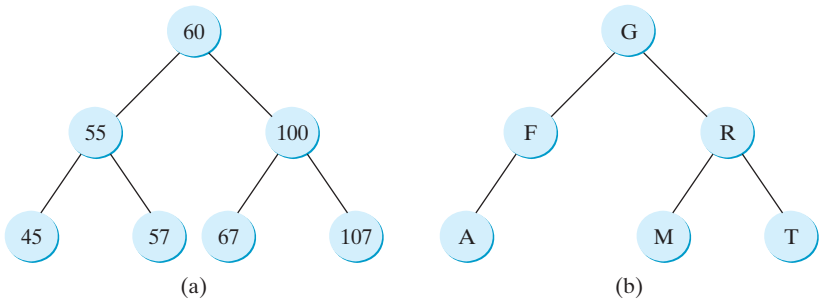


FIGURE 25.1 Each node in a binary tree has zero, one, or two subtrees.

length
depth
level
sibling
leaf
height

The *length* of a path is the number of the edges in the path. The *depth* of a node is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a *level* of the tree. *Siblings* are nodes that share the same parent node. The root of a left (right) subtree of a node is called a *left (right) child* of the node. A node without children is called a *leaf*. The height of a nonempty tree is the length of the path from the root node to its furthest leaf. The *height* of a tree that contains a single node is **0**. Conventionally, the height of an empty tree is **–1**. Consider the tree in Figure 25.1a. The length of the path from node 60 to 45 is **2**. The depth of node 60 is **0**, the depth of node 55 is **1**, and the depth of node 45 is **2**. The height of the tree is **2**. Nodes 45 and 57 are siblings. Nodes 45, 57, 67, and 107 are at the same level.

binary search tree

A special type of binary tree called a *binary search tree* (BST) is often useful. A BST (with no duplicate elements) has the property that for every node in the tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node. The binary trees in Figure 25.1 are all BSTs.

BST animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how a BST works, go to www.cs.armstrong.edu/liang/animation/web/BST.html, as shown in Figure 25.2.

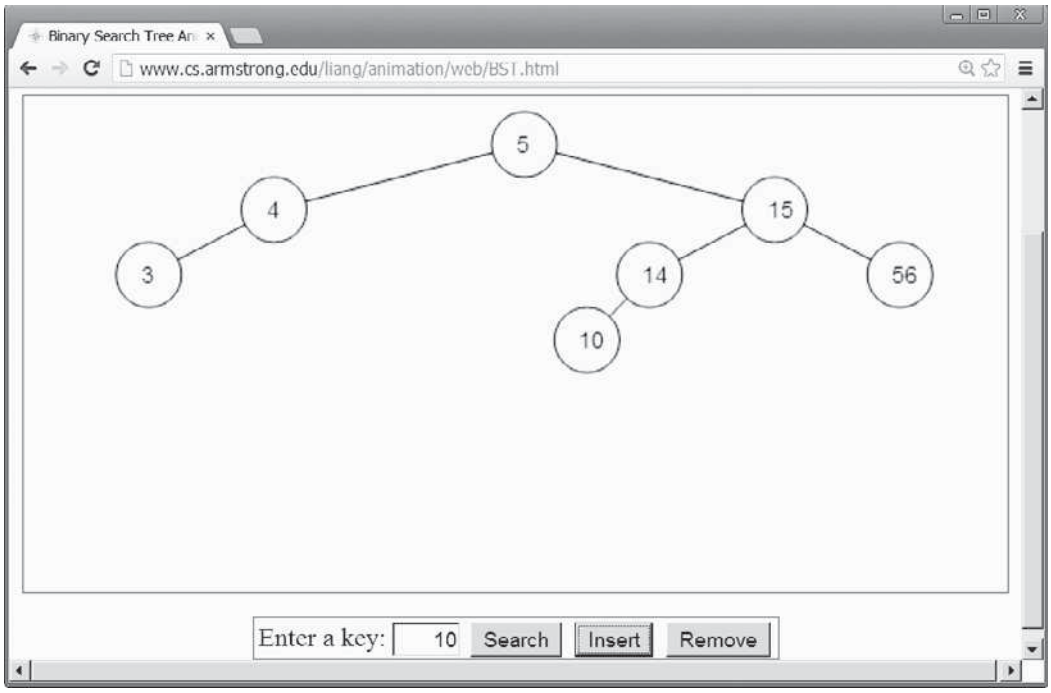


FIGURE 25.2 The animation tool enables you to insert, delete, and search elements.

25.2.1 Representing Binary Search Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in Figure 25.3.

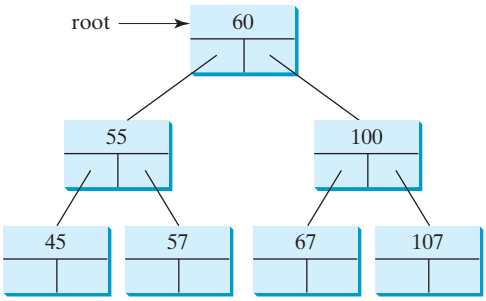


FIGURE 25.3 A binary tree can be represented using a set of linked nodes.

A node can be defined as a class, as follows:

```
class TreeNode<E> {
  protected E element;
  protected TreeNode<E> left;
  protected TreeNode<E> right;

  public TreeNode(E e) {
    element = e;
  }
}
```

The variable `root` refers to the root node of the tree. If the tree is empty, `root` is `null`. The following code creates the first three nodes of the tree in Figure 25.3.

```
// Create the root node
TreeNode<Integer> root = new TreeNode<>(60);

// Create the left child node
root.left = new TreeNode<>(55);

// Create the right child node
root.right = new TreeNode<>(100);
```

25.2.2 Searching for an Element

To search for an element in the BST, you start from the root and scan down from it until a match is found or you arrive at an empty subtree. The algorithm is described in Listing 25.1. Let `current` point to the root (line 2). Repeat the following steps until `current` is `null` (line 4) or the element matches `current.element` (line 12).

- If `element` is less than `current.element`, assign `current.left` to `current` (line 6).
- If `element` is greater than `current.element`, assign `current.right` to `current` (line 9).
- If `element` is equal to `current.element`, return `true` (line 12).

If `current` is `null`, the subtree is empty and the element is not in the tree (line 14).

LISTING 25.1 Searching for an Element in a BST

```
1  public boolean search(E element) {
2    TreeNode<E> current = root; // Start from the root
3
4    while (current != null)
5      if (element < current.element) {
6        current = current.left; // Go left
7      }
8      else if (element > current.element) {
9        current = current.right; // Go right
10     }
11     else // Element matches current.element
12       return true; // Element is found
13
14   return false; // Element is not in the tree
15 }
```

start from root

left subtree

right subtree

found

not found

25.2.3 Inserting an Element into a BST

To insert an element into a BST, you need to locate where to insert it in the tree. The key idea is to locate the parent for the new node. Listing 25.2 gives the algorithm.

LISTING 25.2 Inserting an Element into a BST

```
1  boolean insert(E e) {
2    if (tree is empty)
3      // Create the node for e as the root;
4    else {
5      // Locate the parent node
6      parent = current = root;
7      while (current != null)
```

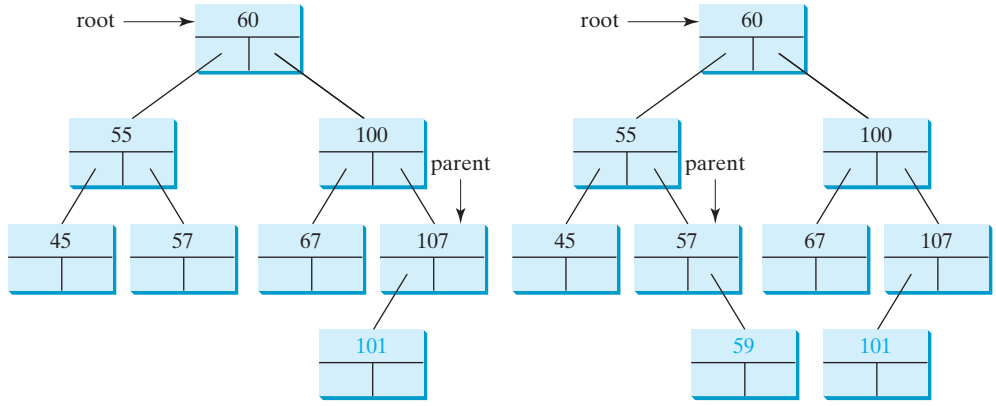
create a new node

locate parent

```
 8        if (e < the value in current.element) {
 9          parent = current;  // Keep the parent
10          current = current.left; // Go left                    left child
11        }
12        else if (e > the value in current.element) {
13          parent = current; // Keep the parent
14          current = current.right; // Go right                  right child
15        }
16        else
17          return false; // Duplicate node not inserted
18
19      // Create a new node for e and attach it to parent
20
21      return true; // Element inserted
22    }
23  }
```

If the tree is empty, create a root node with the new element (lines 2–3). Otherwise, locate the parent node for the new element node (lines 6–17). Create a new node for the element and link this node to its parent node. If the new element is less than the parent element, the node for the new element will be the left child of the parent. If the new element is greater than the parent element, the node for the new element will be the right child of the parent.

For example, to insert **101** into the tree in Figure 25.3, after the **while** loop finishes in the algorithm, **parent** points to the node for **107**, as shown in Figure 25.4a. The new node for **101** becomes the left child of the parent. To insert **59** into the tree, after the **while** loop finishes in the algorithm, the parent points to the node for **57**, as shown in Figure 25.4b. The new node for **59** becomes the right child of the parent.



(a) Inserting 101 (b) Inserting 59

FIGURE 25.4 Two new elements are inserted into the tree.

25.2.4 Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents *inorder*, *postorder, preorder*, *depth-first,* and *breadth-first* traversals. tree traversal

With *inorder traversal*, the left subtree of the current node is visited first recursively, then the current node, and finally the right subtree of the current node recursively. The inorder traversal displays all the nodes in a BST in increasing order. inorder traversal

With *postorder traversal*, the left subtree of the current node is visited recursively first, then recursively the right subtree of the current node, and finally the current node itself. An application of postorder is to find the size of the directory in a file system. As shown in postorder traversal

Figure 25.5, each directory is an internal node and a file is a leaf node. You can apply postorder to get the size of each file and subdirectory before finding the size of the root directory.

preorder traversal
depth-first traversal

With *preorder traversal*, the current node is visited first, then recursively the left subtree of the current node, and finally the right subtree of the current node recursively. Depth-first traversal is the same as preorder traversal. An application of preorder is to print a structured document. As shown in Figure 25.6, you can print a book's table of contents using preorder traversal.

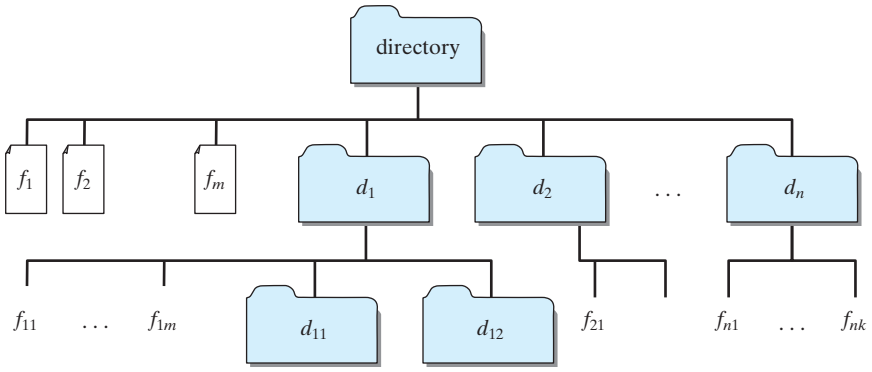


FIGURE 25.5 A directory contains files and subdirectories.

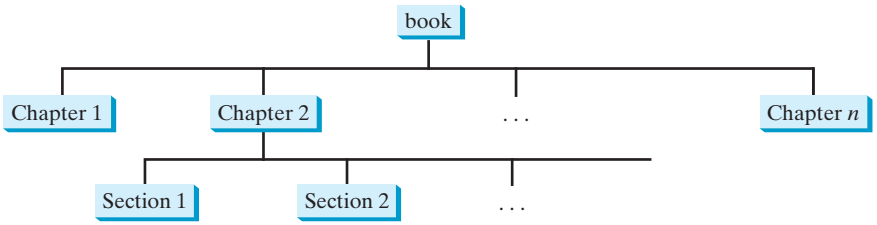


FIGURE 25.6 A tree can be used to represent a structured document such as a book and its chapters and sections.

Note

You can reconstruct a binary search tree by inserting the elements in their preorder. The reconstructed tree preserves the parent and child relationship for the nodes in the original binary search tree.

breadth-first traversal

With breadth-first traversal, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

For example, in the tree in Figure 25.4b, the inorder is

```
45 55 57 59 60 67 100 101 107
```

The postorder is

```
45 59 57 55 67 101 107 100 60
```

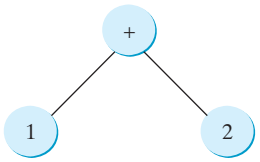
The preorder is

```
60 55 45 57 59 100 67 107 101
```

The breadth-first traversal is

```
60 55 100 45 57 67 107 59 101
```

You can use the following tree to help remember inorder, postorder, and preorder.



The inorder is `1 + 2`, the postorder is `1 2 +`, and the preorder is `+ 1 2`.

25.2.5 The BST Class

Following the design pattern of the Java Collections Framework API, we use an interface named `Tree` to define all common operations for trees and provide an abstract class named `AbstractTree` that partially implements `Tree`, as shown in Figure 25.7. A concrete `BST` class can be defined to extend `AbstractTree`, as shown in Figure 25.8.

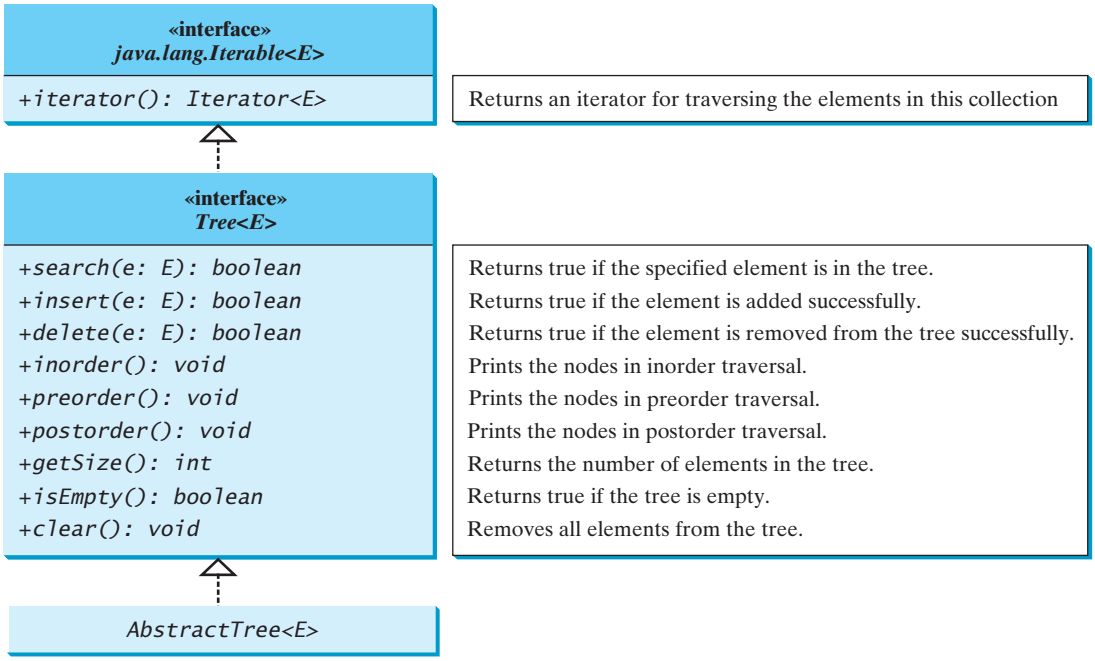


FIGURE 25.7 The `Tree` interface defines common operations for trees, and the `AbstractTree` class partially implements `Tree`.

Listings 25.3, 25.4, and 25.5 give the implementations for `Tree`, `AbstractTree`, and `BST`.

LISTING 25.3 Tree.java

```
1  public interface Tree<E> extends Iterable<E> {                    interface
2    /** Return true if the element is in the tree */
3    public boolean search(E e);                                      search
4
5    /** Insert element e into the binary search tree.
6     * Return true if the element is inserted successfully. */
7    public boolean insert(E e);                                      insert
8
```

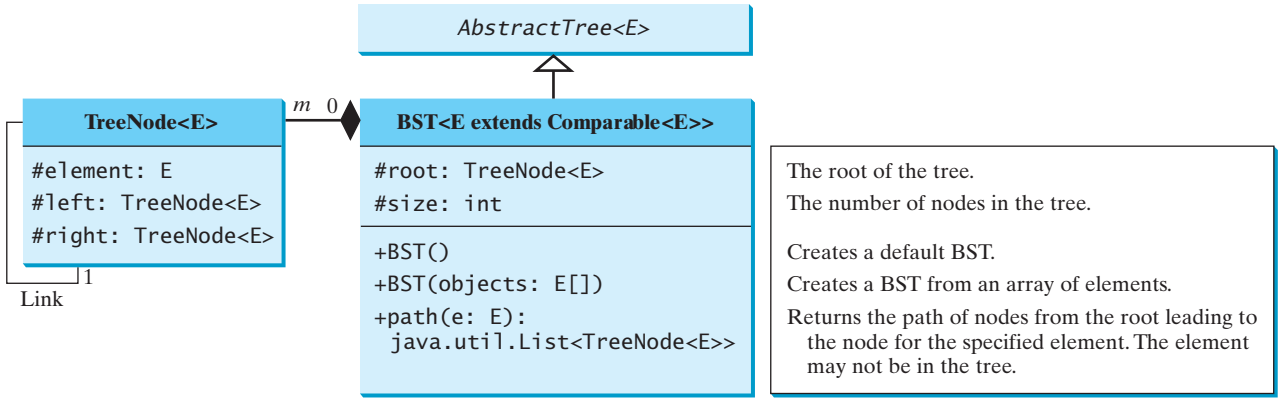


FIGURE 25.8 The `BST` class defines a concrete BST.

```
 9    /** Delete the specified element from the tree.
10     * Return true if the element is deleted successfully. */
11    public boolean delete(E e);
12
13    /** Inorder traversal from the root*/
14    public void inorder();
15
16    /** Postorder traversal from the root */
17    public void postorder();
18
19    /** Preorder traversal from the root */
20    public void preorder();
21
22    /** Get the number of nodes in the tree */
23    public int getSize();
24
25    /** Return true if the tree is empty */
26    public boolean isEmpty();
27  }
```

delete

inorder

postorder

preorder

getSize

isEmpty

LISTING 25.4 AbstractTree.java

```
1   public abstract class AbstractTree<E>
2       implements Tree<E> {
3     @Override /** Inorder traversal from the root*/
4     public void inorder() {
5     }
6
7     @Override /** Postorder traversal from the root */
8     public void postorder() {
9     }
10
11    @Override /** Preorder traversal from the root */
12    public void preorder() {
13    }
14
15    @Override /** Return true if the tree is empty */
```

abstract class

default `inorder` implementation

default `postorder` implementation

default `preorder` implementation

```
16    public boolean isEmpty() {                                    isEmpty implementation
17      return getSize() == 0;
18    }
19  }
```

LISTING 25.5 BST.java

```
 1  public class BST<E extends Comparable<E>>                       BST class
 2      extends AbstractTree<E> {
 3    protected TreeNode<E> root;                                   root
 4    protected int size = 0;                                       size
 5
 6    /** Create a default binary search tree */
 7    public BST() {                                                no-arg constructor
 8    }
 9
10    /** Create a binary search tree from an array of objects */
11    public BST(E[] objects) {                                     constructor
12      for (int i = 0; i < objects.length; i++)
13        insert(objects[i]);
14    }
15
16    @Override /** Return true if the element is in the tree */
17    public boolean search(E e) {                                  search
18      TreeNode<E> current = root; // Start from the root
19
20      while (current != null) {
21        if (e.compareTo(current.element) < 0) {                   compare objects
22          current = current.left;
23        }
24        else if (e.compareTo(current.element) > 0) {
25          current = current.right;
26        }
27        else // element matches current.element
28          return true; // Element is found
29      }
30
31      return false;
32    }
33
34    @Override /** Insert element e into the binary search tree.
35     * Return true if the element is inserted successfully. */
36    public boolean insert(E e) {                                  insert
37      if (root == null)
38        root = createNewNode(e); // Create a new root             new root
39      else {
40        // Locate the parent node
41        TreeNode<E> parent = null;
42        TreeNode<E> current = root;
43        while (current != null)
44          if (e.compareTo(current.element) < 0) {                 compare objects
45            parent = current;
46            current = current.left;
47          }
48          else if (e.compareTo(current.element) > 0) {
49            parent = current;
50            current = current.right;
51          }
52          else
53            return false; // Duplicate node not inserted
```

```
54
55          // Create the new node and attach it to the parent node
56          if (e.compareTo(parent.element) < 0)
57            parent.left = createNewNode(e);
58          else
59            parent.right = createNewNode(e);
60        }
61
62        size++;
63        return true; // Element inserted successfully
64      }
65
66      protected TreeNode<E> createNewNode(E e) {
67        return new TreeNode<>(e);
68      }
69
70      @Override /** Inorder traversal from the root */
71      public void inorder() {
72        inorder(root);
73      }
74
75      /** Inorder traversal from a subtree */
76      protected void inorder(TreeNode<E> root) {
77        if (root == null) return;
78        inorder(root.left);
79        System.out.print(root.element + " ");
80        inorder(root.right);
81      }
82
83      @Override /** Postorder traversal from the root */
84      public void postorder() {
85        postorder(root);
86      }
87
88      /** Postorder traversal from a subtree */
89      protected void preorder(TreeNode<E> root) {
90        if (root == null) return;
91        postorder(root.left);
92        postorder(root.right);
93        System.out.print(root.element + " ");
94      }
95
96      @Override /** Preorder traversal from the root */
97      public void preorder() {
98        preorder(root);
99      }
100
101     /** Preorder traversal from a subtree */
102     protected void postorder(TreeNode<E> root) {
103       if (root == null) return;
104       System.out.print(root.element + " ");
105       preorder(root.left);
106       preorder(root.right);
107     }
108
109     /** This inner class is static, because it does not access
110        any instance members defined in its outer class */
111     public static class TreeNode<E extends Comparable<E>> {
112       protected E element;
113       protected TreeNode<E> left;
```

link to parent

increase size

create new node

inorder

recursive helper method

postorder

recursive helper method

preorder

recursive helper method

inner class

```
114       protected TreeNode<E> right;
115
116       public TreeNode(E e) {
117         element = e;
118       }
119     }
120
121     @Override /** Get the number of nodes in the tree */
122     public int getSize() {                                          getSize
123       return size;
124     }
125
126     /** Returns the root of the tree */
127     public TreeNode<E> getRoot() {                                  getRoot
128       return root;
129     }
130
131     /** Returns a path from the root leading to the specified element */
132     public java.util.ArrayList<TreeNode<E>> path(E e) {             path
133       java.util.ArrayList<TreeNode<E>> list =
134         new java.util.ArrayList<>();
135       TreeNode<E> current = root; // Start from the root
136
137       while (current != null) {
138         list.add(current); // Add the node to the list
139         if (e.compareTo(current.element) < 0) {
140           current = current.left;
141         }
142         else if (e.compareTo(current.element) > 0) {
143           current = current.right;
144         }
145         else
146           break;
147       }
148
149       return list; // Return an array list of nodes
150     }
151
152     @Override /** Delete an element from the binary search tree.
153      * Return true if the element is deleted successfully.
154      * Return false if the element is not in the tree. */
155     public boolean delete(E e) {                                    delete
156       // Locate the node to be deleted and also locate its parent node
157       TreeNode<E> parent = null;                                    locate parent
158       TreeNode<E> current = root;                                   locate current
159       while (current != null) {
160         if (e.compareTo(current.element) < 0) {
161           parent = current;
162           current = current.left;
163         }
164         else if (e.compareTo(current.element) > 0) {
165           parent = current;
166           current = current.right;
167         }
168         else
169           break; // Element is in the tree pointed at by current     current found
170       }
171
172       if (current == null)                                          not found
173         return false; // Element is not in the tree
```

```
174
175       // Case 1: current has no left child
Case 1    176       if (current.left == null) {
177         // Connect the parent with the right child of the current node
178         if (parent == null) {
179           root = current.right;
180         }
181         else {
182           if (e.compareTo(parent.element) < 0)
reconnect parent  183             parent.left = current.right;
184           else
reconnect parent  185             parent.right = current.right;
186         }
187       }
188       else {
Case 2    189         // Case 2: The current node has a left child.
190         // Locate the rightmost node in the left subtree of
191         // the current node and also its parent.
locate parentOfRightMost  192         TreeNode<E> parentOfRightMost = current;
locate rightMost  193         TreeNode<E> rightMost = current.left;
194
195         while (rightMost.right != null) {
196           parentOfRightMost = rightMost;
197           rightMost = rightMost.right; // Keep going to the right
198         }
199
200         // Replace the element in current by the element in rightMost
replace current  201         current.element = rightMost.element;
202
203         // Eliminate rightmost node
204         if (parentOfRightMost.right == rightMost)
205           parentOfRightMost.right = rightMost.left;
reconnect parentOfRightMost  206         else
207           // Special case: parentOfRightMost == current
208           parentOfRightMost.left = rightMost.left;
209       }
210
reduce size  211       size--;
successful deletion  212       return true; // Element deleted successfully
213     }
214
215     @Override /** Obtain an iterator. Use inorder. */
iterator  216     public java.util.Iterator<E> iterator() {
217       return new InorderIterator();
218     }
219
220     // Inner class InorderIterator
iterator class  221     private class InorderIterator implements java.util.Iterator<E> {
222       // Store the elements in a list
internal list  223       private java.util.ArrayList<E> list =
224         new java.util.ArrayList<>();
current position  225       private int current = 0; // Point to the current element in list
226
227       public InorderIterator() {
228         inorder(); // Traverse binary tree and store elements in list
229       }
230
231       /** Inorder traversal from the root*/
232       private void inorder() {
obtain inorder list  233         inorder(root);
```

```
234       }
235
236       /** Inorder traversal from a subtree */
237       private void inorder(TreeNode<E> root) {
238         if (root == null) return;
239         inorder(root.left);
240         list.add(root.element);
241         inorder(root.right);
242       }
243
244       @Override /** More elements for traversing? */
245       public boolean hasNext() {                                  hasNext in iterator?
246         if (current < list.size())
247           return true;
248
249         return false;
250       }
251
252       @Override /** Get the current element and move to the next */
253       public E next() {                                           get next element
254         return list.get(current++);
255       }
256
257       @Override /** Remove the current element */
258       public void remove() {                                      remove the current
259         delete(list.get(current)); // Delete the current element
260         list.clear(); // Clear the list
261         inorder(); // Rebuild the list                            refresh list
262       }
263     }
264
265     /** Remove all elements from the tree */
266     public void clear() {                                         clear
267       root = null;
268       size = 0;
269     }
270   }
```

The `insert(E e)` method (lines 36–64) creates a node for element `e` and inserts it into the tree. If the tree is empty, the node becomes the root. Otherwise, the method finds an appropriate parent for the node to maintain the order of the tree. If the element is already in the tree, the method returns `false`; otherwise it returns `true`.

The `inorder()` method (lines 71–81) invokes `inorder(root)` to traverse the entire tree. The method `inorder(TreeNode root)` traverses the tree with the specified root. This is a recursive method. It recursively traverses the left subtree, then the root, and finally the right subtree. The traversal ends when the tree is empty.

The `postorder()` method (lines 84–94) and the `preorder()` method (lines 97–107) are implemented similarly using recursion.

The `path(E e)` method (lines 132–150) returns a path of the nodes as an array list. The path starts from the root leading to the element. The element may not be in the tree. For example, in Figure 25.4a, `path(45)` contains the nodes for elements `60`, `55`, and `45`, and `path(58)` contains the nodes for elements `60`, `55`, and `57`.

The implementation of `delete()` and `iterator()` (lines 155–269) will be discussed in Sections 25.3 and 25.5.

Listing 25.6 gives an example that creates a binary search tree using `BST` (line 4). The program adds strings into the tree (lines 5–11), traverses the tree in inorder, postorder, and preorder (lines 14–20), searches for an element (line 24), and obtains a path from the node containing `Peter` to the root (lines 28–31).

LISTING 25.6 TestBST.java

```
1  public class TestBST {
2    public static void main(String[] args) {
3      // Create a BST
4      BST<String> tree = new BST<>();
5      tree.insert("George");
6      tree.insert("Michael");
7      tree.insert("Tom");
8      tree.insert("Adam");
9      tree.insert("Jones");
10     tree.insert("Peter");
11     tree.insert("Daniel");
12
13     // Traverse tree
14     System.out.print("Inorder (sorted): ");
15     tree.inorder();
16     System.out.print("\nPostorder: ");
17     tree.postorder();
18     System.out.print("\nPreorder: ");
19     tree.preorder();
20     System.out.print("\nThe number of nodes is " + tree.getSize());
21
22     // Search for an element
23     System.out.print("\nIs Peter in the tree? " +
24       tree.search("Peter"));
25
26     // Get a path from the root to Peter
27     System.out.print("\nA path from the root to Peter is: ");
28     java.util.ArrayList<BST.TreeNode<String>> path
29       = tree.path("Peter");
30     for (int i = 0; path != null && i < path.size(); i++)
31       System.out.print(path.get(i).element + " ");
32
33     Integer[] numbers = {2, 4, 3, 1, 8, 5, 6, 7};
34     BST<Integer> intTree = new BST<>(numbers);
35     System.out.print("\nInorder (sorted): ");
36     intTree.inorder();
37   }
38 }
```

create tree
insert
inorder
postorder
preorder
getSize
search

```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7
Is Peter in the tree? true
A path from the root to Peter is: George Michael Tom Peter
Inorder (sorted): 1 2 3 4 5 6 7 8
```

The program checks `path != null` in line 30 to ensure that the path is not `null` before invoking `path.get(i)`. This is an example of defensive programming to avoid potential runtime errors.

The program creates another tree for storing `int` values (line 34). After all the elements are inserted in the trees, the trees should appear as shown in Figure 25.9.

If the elements are inserted in a different order (e.g., Daniel, Adam, Jones, Peter, Tom, Michael, George), the tree will look different. However, the inorder traversal prints elements in the same order as long as the set of elements is the same. The inorder traversal displays a sorted list.

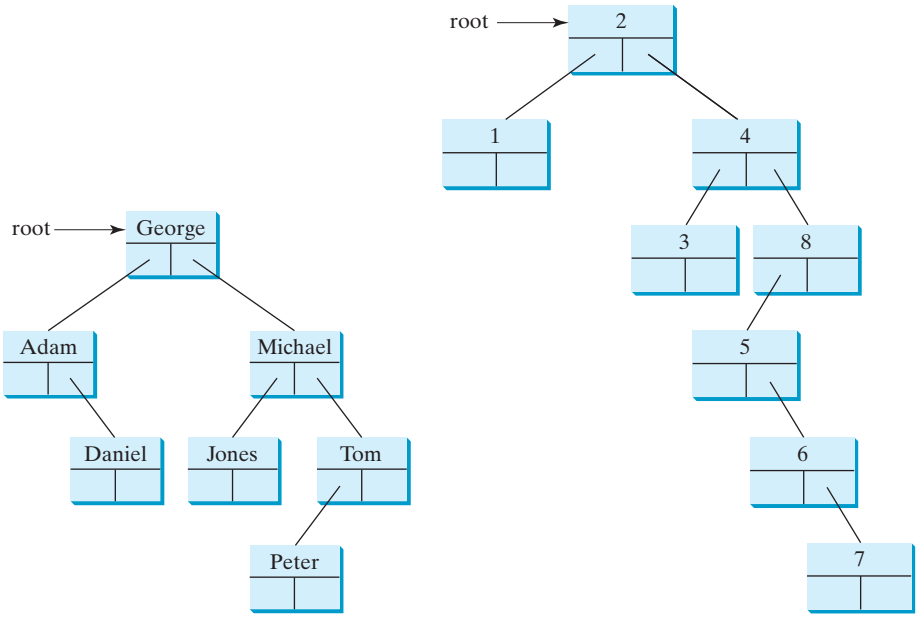


FIGURE 25.9 The BSTs in Listing 25.6 are pictured here after they are created.

25.1 Show the result of inserting `44` into Figure 25.4b.

25.2 Show the inorder, preorder, and postorder of traversing the elements in the binary tree shown in Figure 25.1b.

25.3 If a set of elements is inserted into a BST in two different orders, will the two corresponding BSTs look the same? Will the inorder traversal be the same? Will the postorder traversal be the same? Will the preorder traversal be the same?

25.4 What is the time complexity of inserting an element into a BST?

25.5 Implement the `search(element)` method using recursion.

25.3 Deleting Elements from a BST

To delete an element from a BST, first locate it in the tree and then consider two cases—whether or not the node has a left child—before deleting the element and reconnecting the tree.

The `insert(element)` method was presented in Section 25.2.3. Often you need to delete an element from a binary search tree. Doing so is far more complex than adding an element into a binary search tree.

To delete an element from a binary search tree, you need to first locate the node that contains the element and also its parent node. Let `current` point to the node that contains the element in the binary search tree and `parent` point to the parent of the `current` node. The `current` node may be a left child or a right child of the `parent` node. There are two cases to consider.

locating element

Case 1: The current node does not have a left child, as shown in Figure 25.10a. In this case, simply connect the parent with the right child of the current node, as shown in Figure 25.10b.

For example, to delete node `10` in Figure 25.11a, you would connect the parent of node `10` with the right child of node `10`, as shown in Figure 25.11b.

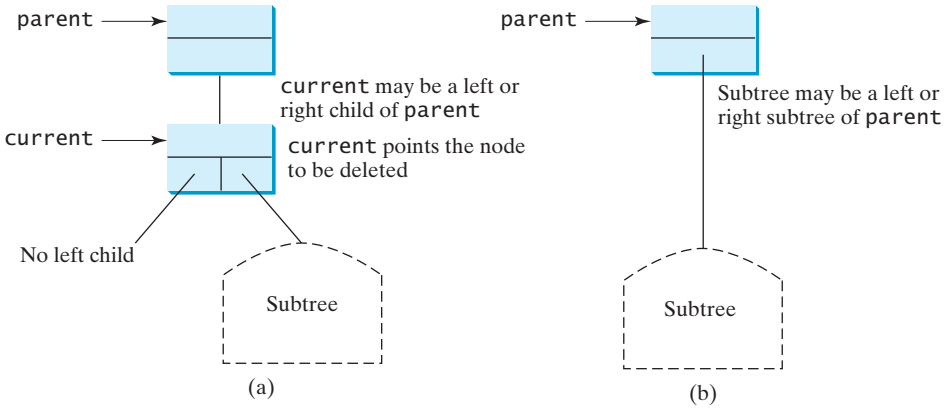


FIGURE 25.10 Case 1: The current node has no left child.

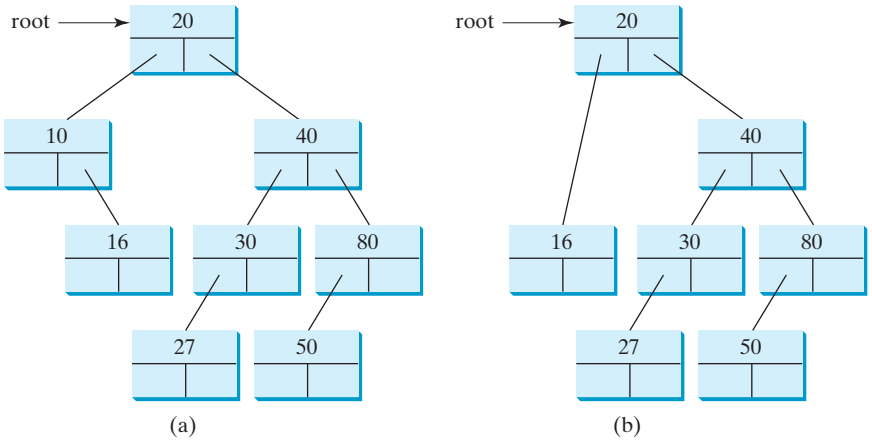


FIGURE 25.11 Case 1: Deleting node 10 from (a) results in (b).

delete a leaf

Note

If the current node is a leaf, it falls into Case 1. For example, to delete element `16` in Figure 25.11a, connect its right child (in this case, it is `null`) to the parent of node `16`.

Case 2: The `current` node has a left child. Let `rightMost` point to the node that contains the largest element in the left subtree of the `current` node and `parentOfRightMost` point to the parent node of the `rightMost` node, as shown in Figure 25.12a. Note that the `rightMost` node cannot have a right child but may have a left child. Replace the element value in the `current` node with the one in the `rightMost` node, connect the `parentOfRightMost` node with the left child of the `rightMost` node, and delete the `rightMost` node, as shown in Figure 25.12b.

For example, consider deleting node `20` in Figure 25.13a. The `rightMost` node has the element value `16`. Replace the element value `20` with `16` in the `current` node and make node `10` the parent for node `14`, as shown in Figure 25.13b.

Note

If the left child of `current` does not have a right child, `current.left` points to the large element in the left subtree of `current`. In this case, `rightMost` is `current.left` and `parentOfRightMost` is `current`. You have to take care of this special case to reconnect the right child of `rightMost` with `parentOfRightMost`.

special case

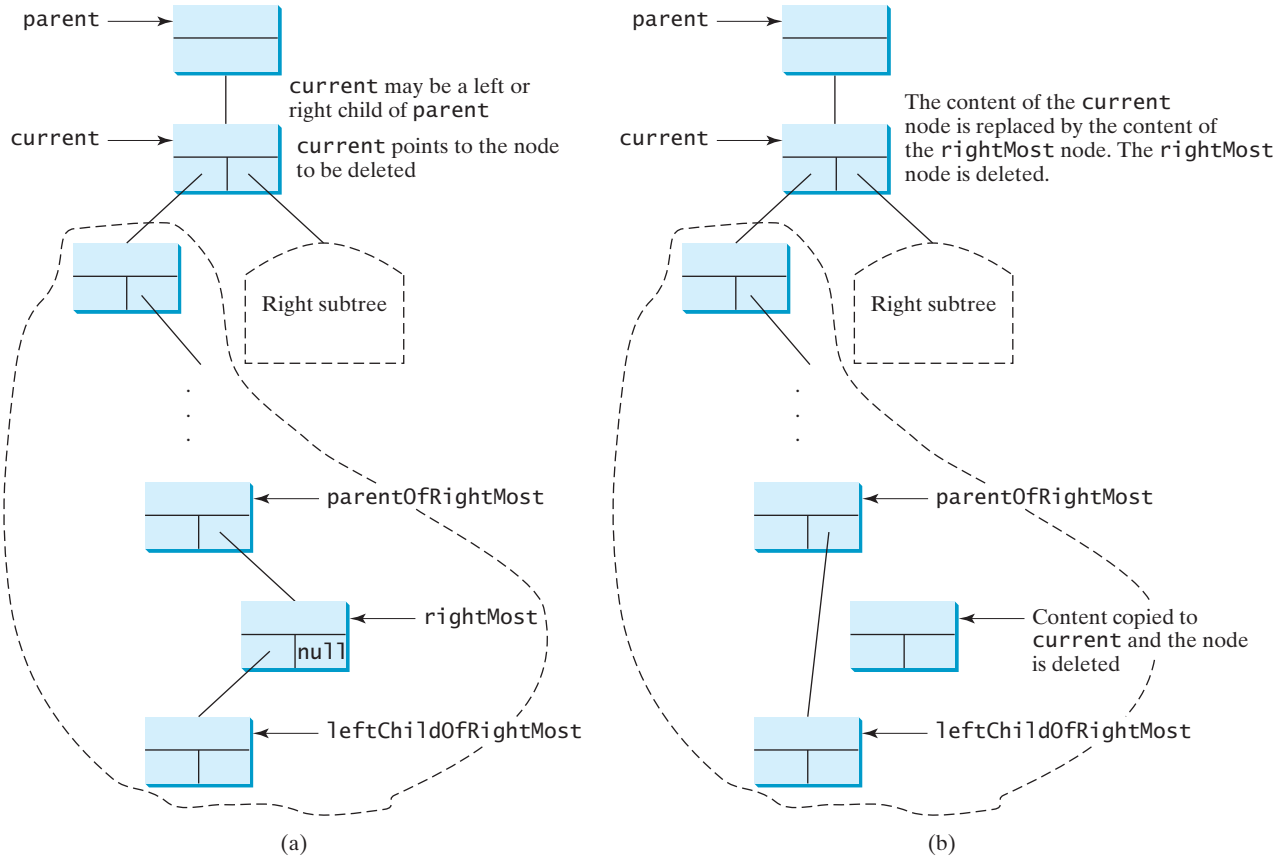


FIGURE 25.12 Case 2: The current node has a left child.

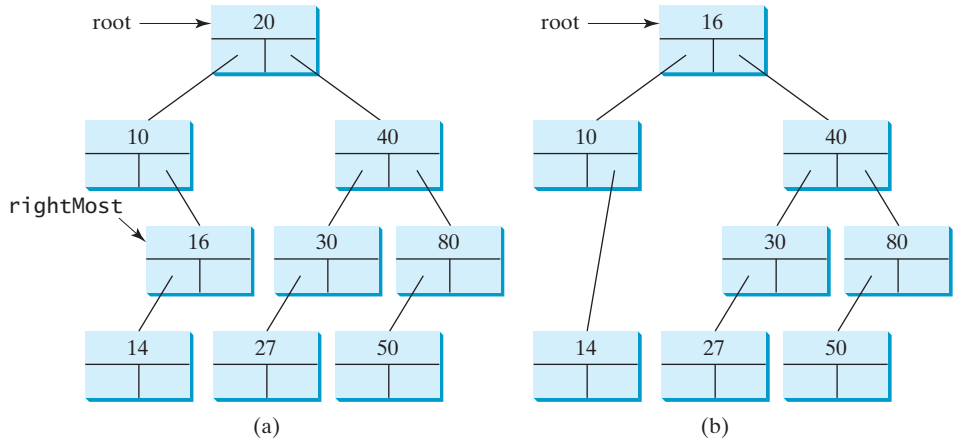


FIGURE 25.13 Case 2: Deleting node 20 from (a) results in (b).

The algorithm for deleting an element from a binary search tree can be described in `delete` method Listing 25.7.

LISTING 25.7 Deleting an Element from a BST

```
1  boolean delete(E e) {
2    Locate element e in the tree;                    not in the tree
3    if element e is not found
```

```
      return true;

    Let current be the node that contains e and parent be
      the parent of current;

    if (current has no left child) // Case 1
      Connect the right child of
        current with parent; now current is not referenced, so
      it is eliminated;
    else // Case 2
      Locate the rightmost node in the left subtree of current.
      Copy the element value in the rightmost node to current.
      Connect the parent of the rightmost node to the left child
        of rightmost node;

    return true; // Element deleted
  }
```

locate current
locate parent
Case 1
Case 2

The complete implementation of the `delete` method is given in lines 155–213 in Listing 25.5. The method locates the node (named `current`) to be deleted and also locates its parent (named `parent`) in lines 157–170. If `current` is `null`, the element is not in the tree. Therefore, the method returns `false` (line 173). Please note that if `current` is `root`, `parent` is `null`. If the tree is empty, both `current` and `parent` are `null`.

Case 1 of the algorithm is covered in lines 176–187. In this case, the `current` node has no left child (i.e., `current.left == null`). If `parent` is `null`, assign `current.right` to `root` (lines 178–180). Otherwise, assign `current.right` to either `parent.left` or `parent.right`, depending on whether `current` is a left or right child of `parent` (182–185).

Case 2 of the algorithm is covered in lines 188–209. In this case, `current` has a left child. The algorithm locates the rightmost node (named `rightMost`) in the left subtree of the current node and also its parent (named `parentOfRightMost`) (lines 195–198). Replace the element in `current` by the element in `rightMost` (line 201); assign `rightMost.left` to either `parentOfRightMost.right` or `parentOfRightMost.left` (lines 204–208), depending on whether `rightMost` is a right or left child of `parentOfRightMost`.

Listing 25.8 gives a test program that deletes the elements from the binary search tree.

LISTING 25.8 TestBSTDelete.java

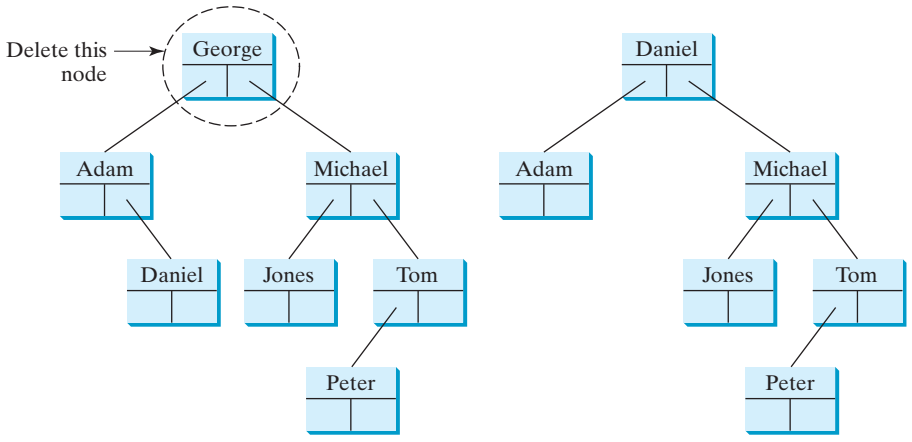
```
1  public class TestBSTDelete {
2    public static void main(String[] args) {
3      BST<String> tree = new BST<>();
4      tree.insert("George");
5      tree.insert("Michael");
6      tree.insert("Tom");
7      tree.insert("Adam");
8      tree.insert("Jones");
9      tree.insert("Peter");
10     tree.insert("Daniel");
11     printTree(tree);
12
13     System.out.println("\nAfter delete George:");
14     tree.delete("George");
15     printTree(tree);
16
17     System.out.println("\nAfter delete Adam:");
18     tree.delete("Adam");
19     printTree(tree);
20
21     System.out.println("\nAfter delete Michael:");
22     tree.delete("Michael");
```

delete an element
delete an element
delete an element

```
23      printTree(tree);
24    }
25
26    public static void printTree(BST tree) {
27      // Traverse tree
28      System.out.print("Inorder (sorted): ");
29      tree.inorder();
30      System.out.print("\nPostorder: ");
31      tree.postorder();
32      System.out.print("\nPreorder: ");
33      tree.preorder();
34      System.out.print("\nThe number of nodes is " + tree.getSize());
35      System.out.println();
36    }
37  }
```

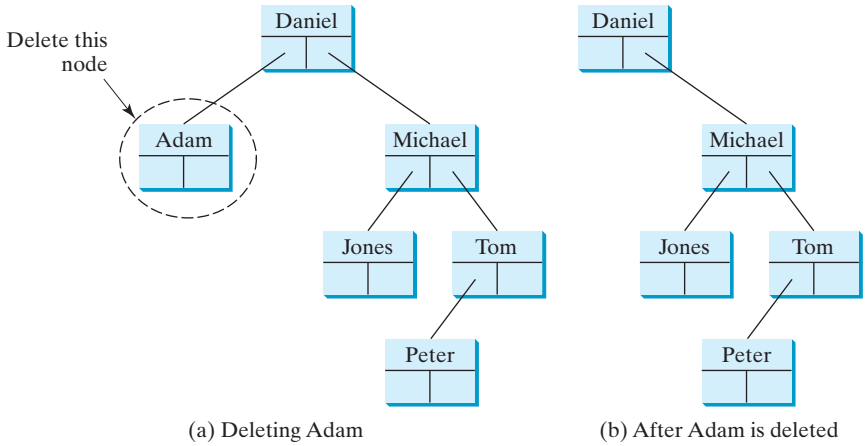
```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7

After delete George:
Inorder (sorted): Adam Daniel Jones Michael Peter Tom
Postorder: Adam Jones Peter Tom Michael Daniel
Preorder: Daniel Adam Michael Jones Tom Peter
The number of nodes is 6

After delete Adam:
Inorder (sorted): Daniel Jones Michael Peter Tom
Postorder: Jones Peter Tom Michael Daniel
Preorder: Daniel Michael Jones Tom Peter
The number of nodes is 5

After delete Michael:
Inorder (sorted): Daniel Jones Peter Tom
Postorder: Peter Tom Jones Daniel
Preorder: Daniel Jones Tom Peter
The number of nodes is 4
```

Figures 25.14–25.16 show how the tree evolves as the elements are deleted from it.



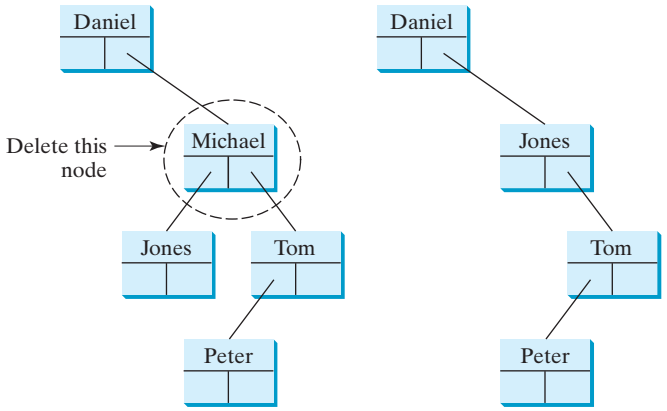
(a) Deleting George (b) After George is deleted

FIGURE 25.14 Deleting George falls into Case 2.



(a) Deleting Adam (b) After Adam is deleted

FIGURE 25.15 Deleting Adam falls into Case 1.



(a) Deleting Michael (b) After Michael is deleted

FIGURE 25.16 Deleting Michael falls into Case 2.

BST time complexity

Note

It is obvious that the time complexity for the inorder, preorder, and postorder is $O(n)$, since each node is traversed only once. The time complexity for search, insertion, and deletion is the height of the tree. In the worst case, the height of the tree is $O(n)$. If a tree is well-balanced, the height would be $O(\log n)$. We will introduce well-balanced binary trees in Chapter 26 and bonus Chapters 40 and 41.

25.6 Show the result of deleting `55` from the tree in Figure 25.4b.

25.7 Show the result of deleting `60` from the tree in Figure 25.4b.

25.8 What is the time complexity of deleting an element from a BST?

25.9 Is the algorithm correct if lines 204–208 in Listing 25.5 in Case 2 of the `delete()` method are replaced by the following code?

```
parentOfRightMost.right = rightMost.left;
```

25.4 Tree Visualization and MVC

You can use recursion to display a binary tree.

Pedagogical Note

One challenge facing the data-structure course is to motivate students. Displaying a binary tree graphically will not only help you understand the working of a binary tree but perhaps also stimulate your interest in programming. This section introduces the techniques to visualize binary trees. You can also apply visualization techniques to other projects.

How do you display a binary tree? It is a recursive structure, so you can display a binary tree using recursion. You can simply display the root, then display the two subtrees recursively. The techniques for displaying the Sierpinski triangle (Listing 18.9, SierpinskiTriangle.java) can be applied to displaying a binary tree. For simplicity, we assume the keys are positive integers less than 100. Listings 25.9 and 25.10 give the program, and Figure 25.17 shows some sample runs of the program.

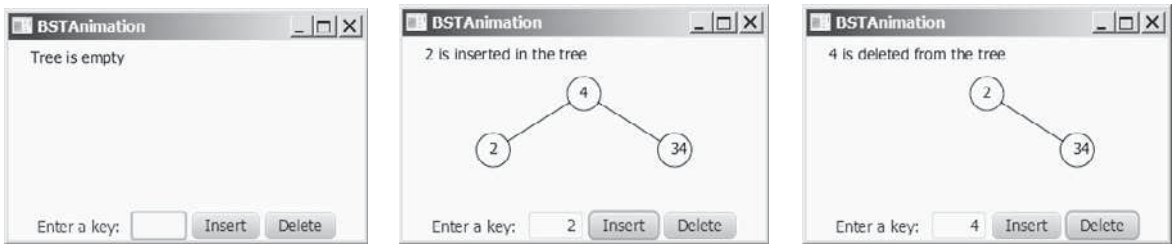


FIGURE 25.17 A binary tree is displayed graphically.

LISTING 25.9 BSTAnimation.java

```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.stage.Stage;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.Label;
7  import javafx.scene.control.TextField;
8  import javafx.scene.layout.BorderPane;
9  import javafx.scene.layout.HBox;
10
11 public class BSTAnimation extends Application {
12   @Override // Override the start method in the Application class
13   public void start(Stage primaryStage) {
14     BST<Integer> tree = new BST<>(); // Create a tree                    create a tree
15
16     BorderPane pane = new BorderPane();
17     BTView view = new BTView(tree); // Create a BTView                   view for tree
18     pane.setCenter(view);                                               place tree view
19
20     TextField tfKey = new TextField();
21     tfKey.setPrefColumnCount(3);
22     tfKey.setAlignment(Pos.BASELINE_RIGHT);
23     Button btInsert = new Button("Insert");
24     Button btDelete = new Button("Delete");
25     HBox hBox = new HBox(5);
26     hBox.getChildren().addAll(new Label("Enter a key: "),
```

```
27              tfKey, btInsert, btDelete);
28      hBox.setAlignment(Pos.CENTER);
29      pane.setBottom(hBox);
30
31      btInsert.setOnAction(e -> {
32        int key = Integer.parseInt(tfKey.getText());
33        if (tree.search(key)) { // key is in the tree already
34          view.displayTree();
35          view.setStatus(key + " is already in the tree");
36        } else {
37          tree.insert(key); // Insert a new key
38          view.displayTree();
39          view.setStatus(key + " is inserted in the tree");
40        }
41      });
42
43      btDelete.setOnAction(e -> {
44        int key = Integer.parseInt(tfKey.getText());
45        if (!tree.search(key)) { // key is not in the tree
46          view.displayTree();
47          view.setStatus(key + " is not in the tree");
48        } else {
49          tree.delete(key); // Delete a key
50          view.displayTree();
51          view.setStatus(key + " is deleted from the tree");
52        }
53      });
54
55      // Create a scene and place the pane in the stage
56      Scene scene = new Scene(pane, 450, 250);
57      primaryStage.setTitle("BSTAnimation"); // Set the stage title
58      primaryStage.setScene(scene); // Place the scene in the stage
59      primaryStage.show(); // Display the stage
60    }
61  }
```

place hBox

handle insertion

insert key
display the tree

handle deletion

delete key
display the tree

LISTING 25.10 BTView.java

```
1  import javafx.scene.layout.Pane;
2  import javafx.scene.paint.Color;
3  import javafx.scene.shape.Circle;
4  import javafx.scene.shape.Line;
5  import javafx.scene.text.Text;
6
7  public class BTView extends Pane {
8    private BST<Integer> tree = new BST<>();
9    private double radius = 15; // Tree node radius
10   private double vGap = 50; // Gap between two levels in a tree
11
12   BTView(BST<Integer> tree) {
13     this.tree = tree;
14     setStatus("Tree is empty");
15   }
16
17   public void setStatus(String msg) {
18     getChildren().add(new Text(20, 20, msg));
19   }
20
```

tree to display

set a tree

```
21    public void displayTree() {
22      this.getChildren().clear(); // Clear the pane                    clear the display
23      if (tree.getRoot() != null) {
24        // Display tree recursively
25        displayTree(tree.getRoot(), getWidth() / 2, vGap,              display tree recursively
26          getWidth() / 4);
27      }
28    }
29
30    /** Display a subtree rooted at position (x, y) */
31    private void displayTree(BST.TreeNode<Integer> root,
32        double x, double y, double hGap) {
33      if (root.left != null) {
34        // Draw a line to the left node
35        getChildren().add(new Line(x - hGap, y + vGap, x, y));         connect two nodes
36        // Draw the left subtree recursively
37        displayTree(root.left, x - hGap, y + vGap, hGap / 2);          draw left subtree
38      }
39
40      if (root.right != null) {
41        // Draw a line to the right node
42        getChildren().add(new Line(x + hGap, y + vGap, x, y));         connect two nodes
43        // Draw the right subtree recursively
44        displayTree(root.right, x + hGap, y + vGap, hGap / 2);         draw right subtree
45      }
46
47      // Display a node
48      Circle circle = new Circle(x, y, radius);
49      circle.setFill(Color.WHITE);
50      circle.setStroke(Color.BLACK);
51      getChildren().addAll(circle,                                     display a node
52        new Text(x - 4, y + 4, root.element + ""));
53    }
54  }
```

In Listing 25.9, BSTAnimation.java, a tree is created (line 14) and a tree view is placed in the pane (line 18). After a new key is inserted into the tree (line 37), the tree is repainted (line 38) to reflect the change. After a key is deleted (line 49), the tree is repainted (line 50) to reflect the change.

In Listing 25.10, BTView.java, the node is displayed as a circle with **`radius 15`** (line 48). The distance between two levels in the tree is defined in **`vGap 50`** (line 25). **`hGap`** (line 32) defines the distance between two nodes horizontally. This value is reduced by half (**`hGap / 2`**) in the next level when the **`displayTree`** method is called recursively (lines 44, 51). Note that **`vGap`** is not changed in the tree.

The method **`displayTree`** is recursively invoked to display a left subtree (lines 33–38) and a right subtree (lines 40–45) if a subtree is not empty. A line is added to the pane to connect two nodes (lines 35, 42). Note that the method first adds the lines to the pane and then adds the circle into the pane (line 52) so that the circles will be painted on top of the lines to achieve desired visual effects.

The program assumes that the keys are integers. You can easily modify the program with a generic type to display keys of characters or short strings.

Tree visualization is an example of the model-view-controller (MVC) software architecture. This is an important architecture for software development. The model is for storing and handling data. The view is for visually presenting the data. The controller handles the user interaction with the model and controls the view, as shown in Figure 25.18.

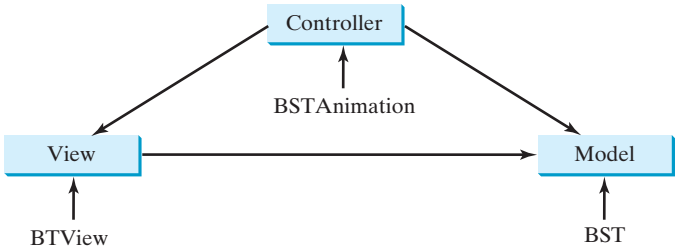


FIGURE 25.18 The controller obtains data and stores it in a model. The view displays the data stored in the model.

The MVC architecture separates data storage and handling from the visual representation of the data. It has two major benefits:

- It makes multiple views possible so that data can be shared through the same model. For example, you can create a new view that displays the tree with the root on the left and tree grows horizontally to the right (see Programming Exercise 25.11).
- It simplifies the task of writing complex applications and makes the components scalable and easy to maintain. Changes can be made to the view without affecting the model, and vice versa.

Check Point

25.10 How many times will the `displayTree` method be invoked if the tree is empty? How many times will the `displayTree` method be invoked if the tree has `100` nodes?

25.11 In what order are the nodes in the tree visited by the `displayTree` method: inorder, preorder, or postorder?

25.12 What would happen if the code in lines 47–52 in BTView.java is moved to line 33?

25.13 What is MVC? What are the benefits of the MVC?

25.5 Iterators

Key Point

`BST` is iterable because it is defined as a subtype of the `java.lang.Iterable` interface.

The methods `inorder()`, `preorder()`, and `postorder()` display the elements in `inorder`, `preorder`, and `postorder` in a binary tree. These methods are limited to displaying the elements in a tree. If you wish to process the elements in a binary tree rather than display them, these methods cannot be used. Recall that an iterator is provided for traversing the elements in a set or list. You can apply the same approach in a binary tree to provide a uniform way of traversing the elements in a binary tree.

iterator

The `java.lang.Iterable` interface defines the `iterator` method, which returns an instance of the `java.util.Iterator` interface. The `java.util.Iterator` interface (see Figure 25.19) defines the common features of iterators.

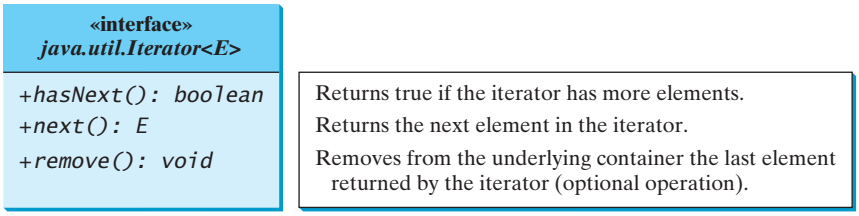


FIGURE 25.19 The `Iterator` interface defines a uniform way of traversing the elements in a container.

The `Tree` interface extends `java.lang.Iterable`. Since `BST` is a subclass of `AbstractTree` and `AbstractTree` implements `Tree`, `BST` is a subtype of `Iterable`. The `Iterable` interface contains the `iterator()` method that returns an instance of `java.util.Iterator`.

You can traverse a binary tree in inorder, preorder, or postorder. Since inorder is used frequently, we will use inorder for traversing the elements in a binary tree. We define an iterator class named `InorderIterator` to implement the `java.util.Iterator` interface in Listing 25.5 (lines 221–263). The `iterator` method simply returns an instance of `InorderIterator` (line 217).

how to create an iterator

The `InorderIterator` constructor invokes the `inorder` method (line 228). The `inorder(root)` method (lines 237–242) stores all the elements from the tree in `list`. The elements are traversed in `inorder`.

Once an `Iterator` object is created, its `current` value is initialized to `0` (line 225), which points to the first element in the list. Invoking the `next()` method returns the current element and moves `current` to point to the next element in the list (line 253).

The `hasNext()` method checks whether `current` is still in the range of `list` (line 246).

The `remove()` method removes the current element from the tree (line 259). Afterward, a new list is created (lines 260–261). Note that `current` does not need to be changed.

Listing 25.11 gives a test program that stores the strings in a BST and displays all strings in uppercase.

LISTING 25.11 TestBSTWithIterator.java

```
1  public class TestBSTWithIterator {
2    public static void main(String[] args) {
3      BST<String> tree = new BST<>();
4      tree.insert("George");
5      tree.insert("Michael");
6      tree.insert("Tom");
7      tree.insert("Adam");
8      tree.insert("Jones");
9      tree.insert("Peter");
10     tree.insert("Daniel");
11
12     for (String s: tree)                                    use an iterator
13       System.out.print(s.toUpperCase() + " ");              get uppercase letters
14   }
15 }
```

```
ADAM DANIEL GEORGE JONES MICHAEL PETER TOM
```

The foreach loop (lines 12–13) uses an iterator to traverse all elements in the tree.

Design Guide

Iterator is an important software design pattern. It provides a uniform way of traversing the elements in a container, while hiding the container's structural details. By implementing the same interface `java.util.Iterator`, you can write a program that traverses the elements of all containers in the same way.

iterator pattern
advantages of iterators

Note

`java.util.Iterator` defines a forward iterator, which traverses the elements in the iterator in a forward direction, and each element can be traversed only once. The Java API also provides the `java.util.ListIterator`, which supports traversing in both forward and backward directions. If your data structure warrants flexible traversing, you may define iterator classes as a subtype of `java.util.ListIterator`.

variations of iterators

The implementation of the iterator is not efficient. Every time you remove an element through the iterator, the whole list is rebuilt (line 261 in Listing 25.5 BST.java). The client should always use the `delete` method in the `BinraryTree` class to remove an element. To prevent the user from using the `remove` method in the iterator, implement the iterator as follows:

```
public void remove() {
  throw new UnsupportedOperationException
    ("Removing an element from the iterator is not supported");
}
```

After making the `remove` method unsupported by the iterator class, you can implement the iterator more efficiently without having to maintain a list for the elements in the tree. You can use a stack to store the nodes, and the node on the top of the stack contains the element that is to be returned from the `next()` method. If the tree is well-balanced, the maximum stack size will be $O(\log n)$.

25.14 What is an iterator?

25.15 What method is defined in the `java.lang.Iterable<E>` interface?

25.16 Suppose you delete `extends Iterable<E>` from line 1 in Listing 25.3, Tree.java. Will Listing 25.11 still compile?

25.17 What is the benefit of being a subtype of `Iterable<E>`?

25.6 Case Study: Data Compression

Huffman coding compresses data by using fewer bits to encode characters that occur more frequently. The codes for the characters are constructed based on the occurrence of the characters in the text using a binary tree, called the Huffman coding tree.

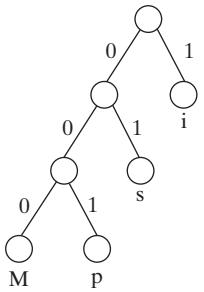
Compressing data is a common task. There are many utilities available for compressing files. This section introduces Huffman coding, invented by David Huffman in 1952.

In ASCII, every character is encoded in 8 bits. If a text consists of 100 characters, it will take 800 bits to represent the text. The idea of Huffman coding is to use a fewer bits to encode frequently used characters in the text and more bits to encode less frequently used characters to reduce the overall size of the file. In Huffman coding, the characters' codes are constructed based on the characters' occurrence in the text using a binary tree, called the *Huffman coding tree*. Suppose the text is `Mississippi`. Its Huffman tree can be shown as in Figure 25.20a. The left and right edges of a node are assigned a value `0` and `1`, respectively. Each character is a leaf in the tree. The code for the character consists of the edge values in the path from the root to the leaf, as shown in Figure 25.20b. Since `i` and `s` appear more than `M` and `p` in the text, they are assigned shorter codes.

Huffman coding

Based on the coding scheme in Figure 25.20,

```
          is encoded to                    is decoded to
Mississippi ========> 00010101110101100100111 ========> Mississippi
```



(a) Huffman coding tree

Character	Code	Frequency
M	000	1
p	001	2
s	01	4
i	1	4

(b) Character code table

FIGURE 25.20 The codes for characters are constructed based on the occurrence of characters in the text using a coding tree.

decoding

The coding tree is also used for decoding a sequence of bits into characters. To do so, start with the first bit in the sequence and determine whether to go to the left or right branch of the tree's root based on the bit value. Consider the next bit and continue to go down to the left or right branch based on the bit value. When you reach a leaf, you have found a character. The next bit in the stream is the first bit of the next character. For example, the stream `011001` is decoded to `sip`, with `01` matching `s`, `1` matching `i`, and `001` matching `p`.

construct coding tree

To construct a *Huffman coding tree*, use an algorithm as follows:

1. Begin with a forest of trees. Each tree contains a node for a character. The weight of the node is the frequency of the character in the text.
2. Repeat the following action to combine trees until there is only one tree: Choose two trees with the smallest weight and create a new node as their parent. The weight of the new tree is the sum of the weight of the subtrees.
3. For each interior node, assign its left edge a value `0` and right edge a value `1`. All leaf nodes represent characters in the text.

Here is an example of building a coding tree for the text `Mississippi`. The frequency table for the characters is shown in Figure 25.20b. Initially the forest contains single-node trees, as shown in Figure 25.21a. The trees are repeatedly combined to form large trees until only one tree is left, as shown in Figure 25.21b–d.

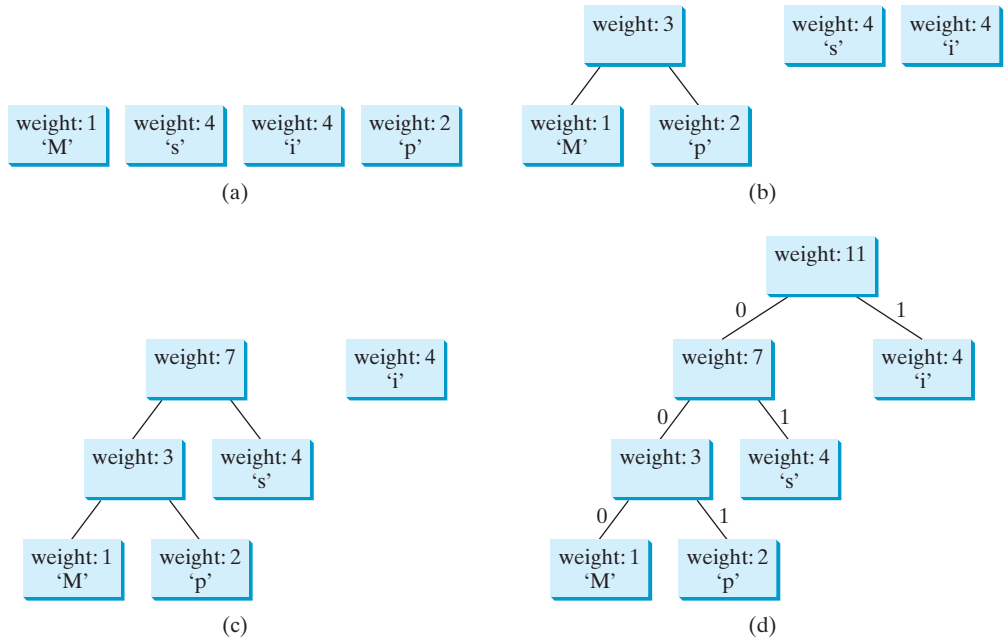


FIGURE 25.21 The coding tree is built by repeatedly combining the two smallest-weighted trees.

prefix property

It is worth noting that no code is a prefix of another code. This property ensures that the streams can be decoded unambiguously.

Pedagogical Note

For an interactive GUI demo to see how Huffman coding works, go to www.cs.armstrong.edu/liang/animation/HuffmanCodingAnimation.html, as shown in Figure 25.22.

Huffman coding animation on Companion Website

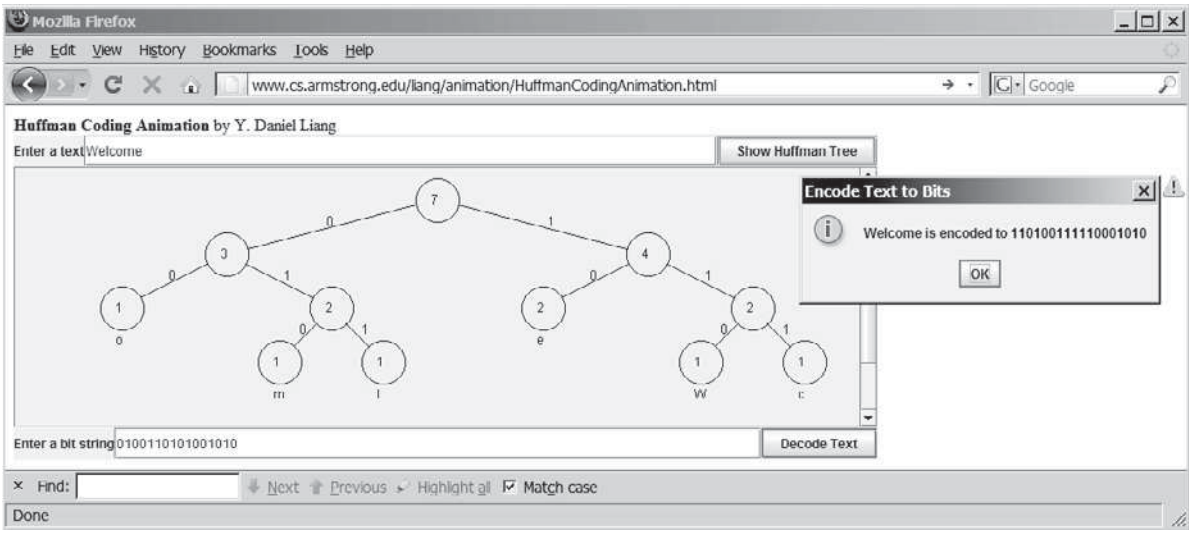


FIGURE 25.22 The animation tool enables you to create and view a Huffman tree, and it performs encoding and decoding using the tree.

greedy algorithm

The algorithm used here is an example of a *greedy algorithm*. A greedy algorithm is often used in solving optimization problems. The algorithm makes the choice that is optimal locally in the hope that this choice will lead to a globally optimal solution. In this case, the algorithm always chooses two trees with the smallest weight and creates a new node as their parent. This intuitive optimal local solution indeed leads to a final optimal solution for constructing a Huffman tree. As another example, consider changing money into the fewest possible coins. A greedy algorithm would take the largest possible coin first. For example, for 98¢, you would use three quarters to make 75¢, additional two dimes to make 95¢, and additional three pennies to make the 98¢. The greedy algorithm finds an optimal solution for this problem. However, a greedy algorithm is not always going to find the optimal result; see the bin packing problem in Programming Exercise 25.22.

Listing 25.12 gives a program that prompts the user to enter a string, displays the frequency table of the characters in the string, and displays the Huffman code for each character.

LISTING 25.12 HuffmanCode.java

```
1  import java.util.Scanner;
2
3  public class HuffmanCode {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      System.out.print("Enter text: ");
7      String text = input.nextLine();
8
9      int[] counts = getCharacterFrequency(text); // Count frequency
10
11     System.out.printf("%-15s%-15s%-15s%-15s\n",
12       "ASCII Code", "Character", "Frequency", "Code");
13
14     Tree tree = getHuffmanTree(counts); // Create a Huffman tree
15     String[] codes = getCode(tree.root); // Get codes
16
17     for (int i = 0; i < codes.length; i++)
18       if (counts[i] != 0) // (char)i is not in text if counts[i] is 0
19         System.out.printf("%-15d%-15s%-15d%-15s\n",
```

count frequency

get Huffman tree
code for each character

```
20            i, (char)i + "", counts[i], codes[i]);
21    }
22
23    /** Get Huffman codes for the characters
24     * This method is called once after a Huffman tree is built
25     */
26    public static String[] getCode(Tree.Node root) {                    getCode
27      if (root == null) return null;
28      String[] codes = new String[2 * 128];
29      assignCode(root, codes);
30      return codes;
31    }
32
33    /* Recursively get codes to the leaf node */
34    private static void assignCode(Tree.Node root, String[] codes) {    assignCode
35      if (root.left != null) {
36        root.left.code = root.code + "0";
37        assignCode(root.left, codes);
38
39        root.right.code = root.code + "1";
40        assignCode(root.right, codes);
41      }
42      else {
43        codes[(int)root.element] = root.code;
44      }
45    }
46
47    /** Get a Huffman tree from the codes */
48    public static Tree getHuffmanTree(int[] counts) {                   getHuffmanTree
49      // Create a heap to hold trees
50      Heap<Tree> heap = new Heap<>(); // Defined in Listing 23.9
51      for (int i = 0; i < counts.length; i++) {
52        if (counts[i] > 0)
53          heap.add(new Tree(counts[i], (char)i)); // A leaf node tree
54      }
55
56      while (heap.getSize() > 1) {
57        Tree t1 = heap.remove(); // Remove the smallest-weight tree
58        Tree t2 = heap.remove(); // Remove the next smallest
59        heap.add(new Tree(t1, t2)); // Combine two trees
60      }
61
62      return heap.remove(); // The final tree
63    }
64
65    /** Get the frequency of the characters */
66    public static int[] getCharacterFrequency(String text) {            getCharacterFrequency
67      int[] counts = new int[256]; // 256 ASCII characters
68
69      for (int i = 0; i < text.length(); i++)
70        counts[(int)text.charAt(i)]++; // Count the characters in text
71
72      return counts;
73    }
74
75    /** Define a Huffman coding tree */
76    public static class Tree implements Comparable<Tree> {              Huffman tree
77      Node root; // The root of the tree
78
79      /** Create a tree with two subtrees */
```

```
80      public Tree(Tree t1, Tree t2) {
81        root = new Node();
82        root.left = t1.root;
83        root.right = t2.root;
84        root.weight = t1.root.weight + t2.root.weight;
85      }
86
87      /** Create a tree containing a leaf node */
88      public Tree(int weight, char element) {
89        root = new Node(weight, element);
90      }
91
92      @Override /** Compare trees based on their weights */
93      public int compareTo(Tree t) {
94        if (root.weight < t.root.weight) // Purposely reverse the order
95          return 1;
96        else if (root.weight == t.root.weight)
97          return 0;
98        else
99          return -1;
100     }
101
102     public class Node {
103       char element; // Stores the character for a leaf node
104       int weight; // weight of the subtree rooted at this node
105       Node left; // Reference to the left subtree
106       Node right; // Reference to the right subtree
107       String code = ""; // The code of this node from the root
108
109       /** Create an empty node */
110       public Node() {
111       }
112
113       /** Create a node with the specified weight and character */
114       public Node(int weight, char element) {
115         this.weight = weight;
116         this.element = element;
117       }
118     }
119   }
120 }
```

tree node

```
Enter text: Welcome ↵Enter
ASCII Code     Character      Frequency      Code
87             W              1              110
99             c              1              111
101            e              2              10
108            l              1              011
109            m              1              010
111            o              1              00
```

getCharacterFrequency

The program prompts the user to enter a text string (lines 5–7) and counts the frequency of the characters in the text (line 9). The **`getCharacterFrequency`** method (lines 66–73) creates an array **`counts`** to count the occurrences of each of the 256 ASCII characters in the text. If a character appears in the text, its corresponding count is increased by **`1`** (line 70).

The program obtains a Huffman coding tree based on `counts` (line 14). The tree consists of linked nodes. The `Node` class is defined in lines 102–118. Each node consists of properties `element` (storing character), `weight` (storing weight of the subtree under this node), `left` (linking to the left subtree), `right` (linking to the right subtree), and `code` (storing the Huffman code for the character). The `Tree` class (lines 76–119) contains the root property. From the root, you can access all the nodes in the tree. The `Tree` class implements `Comparable`. The trees are comparable based on their weights. The compare order is purposely reversed (lines 93–100) so that the smallest-weight tree is removed first from the heap of trees.

`Node` class

`Tree` class

The `getHuffmanTree` method returns a Huffman coding tree. Initially, the single-node trees are created and added to the heap (lines 50–54). In each iteration of the `while` loop (lines 56–60), two smallest-weight trees are removed from the heap and are combined to form a big tree, and then the new tree is added to the heap. This process continues until the heap contains just one tree, which is our final Huffman tree for the text.

`getHuffmanTree`

The `assignCode` method assigns the code for each node in the tree (lines 34–45). The `getCode` method gets the code for each character in the leaf node (lines 26–31). The element `codes[i]` contains the code for character `(char)i`, where `i` is from `0` to `255`. Note that `codes[i]` is `null` if `(char)i` is not in the text.

`assignCode`
`getCode`

25.18 Every internal node in a Huffman tree has two children. Is it true?

25.19 What is a greedy algorithm? Give an example.

25.20 If the `Heap` class in line 50 in Listing 25.10 is replaced by `java.util.PriorityQueue`, will the program still work?

Key Terms

binary search tree 930
binary tree 930
breadth-first traversal 934
depth-first traversal 934
greedy algorithm 956
Huffman coding 954
inorder traversal 933
postorder traversal 933
preorder traversal 934
tree traversal 933

Chapter Summary

1. A *binary search tree* (BST) is a hierarchical data structure. You learned how to define and implement a BST class, how to insert and delete elements into/from a BST, and how to traverse a BST using *inorder*, *postorder*, *preorder*, depth-first, and breadth-first searches.

2. An iterator is an object that provides a uniform way of traversing the elements in a container, such as a set, a list, or a *binary tree*. You learned how to define and implement iterator classes for traversing the elements in a binary tree.

3. *Huffman coding* is a scheme for compressing data by using fewer bits to encode characters that occur more frequently. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree, called the *Huffman coding tree*.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/intro10e/quiz.html.

MyProgrammingLab™

PROGRAMMING EXERCISES

Sections 25.2–25.6

*25.1 (*Add new methods in BST*) Add the following new methods in BST.

```
/** Displays the nodes in a breadth-first traversal */
public void breadthFirstTraversal()

/** Returns the height of this binary tree */
public int height()
```

*25.2 (*Test full binary tree*) A full binary tree is a binary tree with the leaves on the same level. Add a method in the BST class to return true if the tree is a full binary tree. (*Hint*: The number of nodes in a full binary tree is $2^{\text{depth}} - 1$.)

```
/** Returns true if the tree is a full binary tree */
boolean isFullBST()
```

**25.3 (*Implement inorder traversal without using recursion*) Implement the inorder method in BST using a stack instead of recursion. Write a test program that prompts the user to enter 10 integers, stores them in a BST, and invokes the inorder method to display the elements.

**25.4 (*Implement preorder traversal without using recursion*) Implement the preorder method in BST using a stack instead of recursion. Write a test program that prompts the user to enter 10 integers, stores them in a BST, and invokes the preorder method to display the elements.

**25.5 (*Implement postorder traversal without using recursion*) Implement the postorder method in BST using a stack instead of recursion. Write a test program that prompts the user to enter 10 integers, stores them in a BST, and invokes the postorder method to display the elements.

**25.6 (*Find the leaves*) Add a method in the BST class to return the number of the leaves as follows:

```
/** Returns the number of leaf nodes */
public int getNumberOfLeaves()
```

**25.7 (*Find the nonleaves*) Add a method in the BST class to return the number of the nonleaves as follows:

```
/** Returns the number of nonleaf nodes */
public int getNumberofNonLeaves()
```

***25.8 (*Implement bidirectional iterator*) The java.util.Iterator interface defines a forward iterator. The Java API also provides the java.util.ListIterator interface that defines a bidirectional iterator. Study ListIterator and define a bidirectional iterator for the BST class.

**25.9 (*Tree clone and equals*) Implement the clone and equals methods in the BST class. Two BST trees are equal if they contain the same elements. The clone method returns an identical copy of a BST.

25.10 (*Preorder iterator*) Add the following method in the `BST` class that returns an iterator for traversing the elements in a BST in preorder.

```
/** Returns an iterator for traversing the elements in preorder */
java.util.Iterator<E> preorderIterator()
```

25.11 (*Display tree*) Write a new view class that displays the tree horizontally with the root on the left as shown in Figure 25.23.

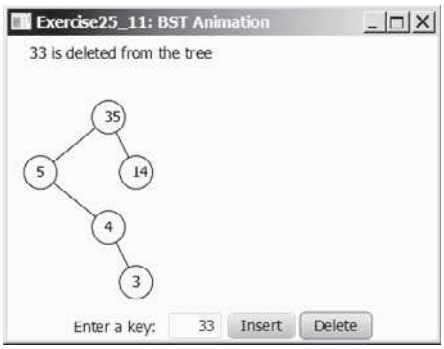


FIGURE 25.23 A binary tree is displayed horizontally.

****25.12** (*Test* `BST`) Design and write a complete test program to test if the `BST` class in Listing 25.5 meets all requirements.

****25.13** (*Add new buttons in* `BSTAnimation`) Modify Listing 25.9, BSTAnimation.java, to add three new buttons—*Show Inorder, Show Preorder, and Show Postorder*—to display the result in a label, as shown in Figure 25.24. You need also to modify BST.java to implement the `inorderList()`, `preorderList()`, and `postorderList()` methods so that each of these methods returns a `List` of the node elements in inorder, preorder, and postorder, as follows:

```
public java.util.List<E> inorderList();
public java.util.List<E> preorderList();
public java.util.List<E> postorderList();
```

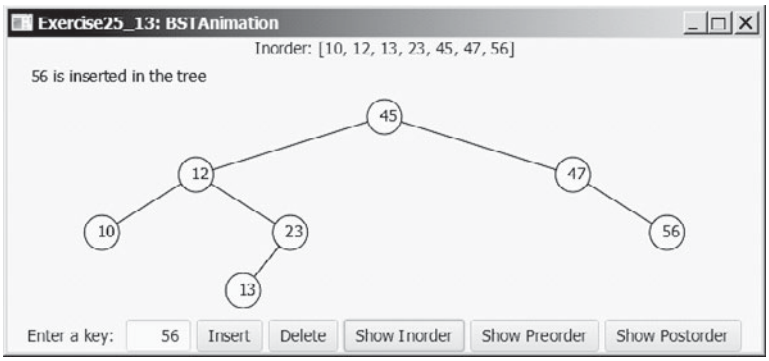
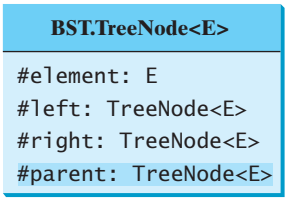


FIGURE 25.24 When you click the Show Inorder, Show Preorder, or Show Postorder button, the elements are displayed in an inorder, preorder, or postorder in a label.

***25.14** (*Generic BST* using `Comparator`) Revise `BST` in Listing 25.5, using a generic parameter and a `Comparator` for comparing objects. Define a new constructor with a `Comparator` as its argument as follows:

```
BST(Comparator<? super E> comparator)
```

***25.15** (*Parent reference for BST*) Redefine `TreeNode` by adding a reference to a node's parent, as shown below:



Reimplement the `insert` and `delete` methods in the `BST` class to update the parent for each node in the tree. Add the following new method in `BST`:

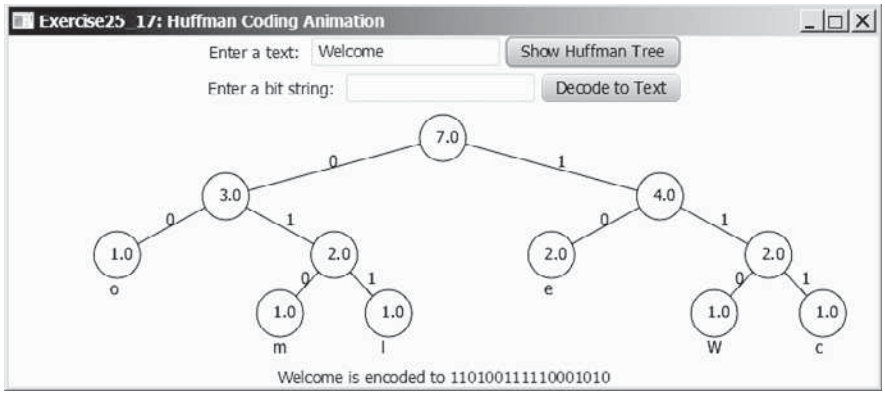
```
/** Returns the node for the specified element.
  * Returns null if the element is not in the tree. */
private TreeNode<E> getNode(E element)

/** Returns true if the node for the element is a leaf */
private boolean isLeaf(E element)

/** Returns the path of elements from the specified element
  * to the root in an array list. */
public ArrayList<E> getPath(E e)
```

Write a test program that prompts the user to enter 10 integers, adds them to the tree, deletes the first integer from the tree, and displays the paths for all leaf nodes. Here is a sample run:

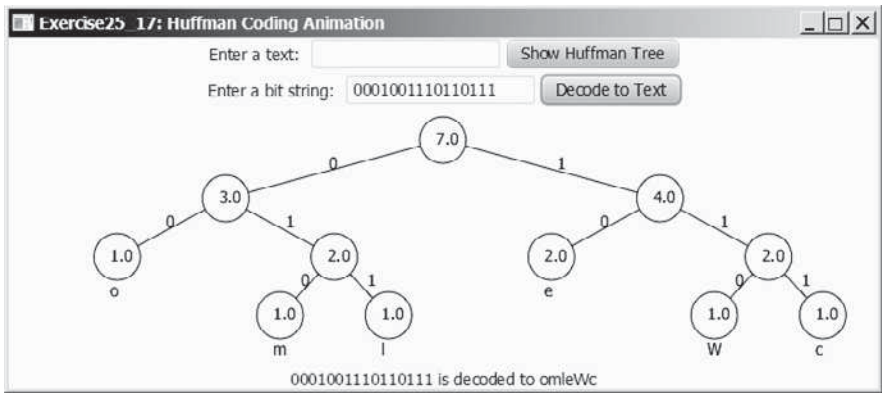
```
Enter 10 integers: 45 54 67 56 50 45 23 59 23 67 Enter
[50, 54, 23]
[59, 56, 67, 54, 23]
```

*****25.16** (*Data compression: Huffman coding*) Write a program that prompts the user to enter a file name, then displays the frequency table of the characters in the file and displays the Huffman code for each character.

*****25.17** (*Data compression: Huffman coding animation*) Write a program that enables the user to enter text and displays the Huffman coding tree based on the text, as shown in Figure 25.25a. Display the weight of the subtree inside the subtree's root circle. Display each leaf node's character. Display the encoded bits for the text in a label. When the user clicks the *Decode Text* button, a bit string is decoded into text displayed in the label, as shown in Figure 25.25b.



(a)



(b)

FIGURE 25.25 (a) The animation shows the coding tree for a given text string and the encoded bits for the text are displayed in the label; (b) You can enter a bit string to display its text in the label.

*****25.18** (*Compress a file*) Write a program that compresses a source file into a target file using the Huffman coding method. First use `ObjectOutputStream` to output the Huffman codes into the target file, and then use `BitOutputStream` in Programming Exercise 17.17 to output the encoded binary contents to the target file. Pass the files from the command line using the following command:

```
java Exercise25_18 sourcefile targetfile
```

*****25.19** (*Decompress a file*) The preceding exercise compresses a file. The compressed file contains the Huffman codes and the compressed contents. Write a program that decompresses a source file into a target file using the following command:

```
java Exercise25_19 sourcefile targetfile
```

25.20 (*Bin packing using first fit*) Write a program that packs the objects of various weights into containers. Each container can hold a maximum of 10 pounds. The program uses a greedy algorithm that places an object into the first bin in which it would fit. Your program should prompt the user to enter the total number of

objects and the weight of each object. The program displays the total number of containers needed to pack the objects and the contents of each container. Here is a sample run of the program:

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 7 2
Container 2 contains objects with weight 5 3
Container 3 contains objects with weight 5
Container 4 contains objects with weight 8
```

Does this program produce an optimal solution, that is, finding the minimum number of containers to pack the objects?

25.21 (*Bin packing with smallest object first*) Rewrite the preceding program that uses a new greedy algorithm that places an object with the *smallest weight* into the first bin in which it would fit. Your program should prompt the user to enter the total number of objects and the weight of each object. The program displays the total number of containers needed to pack the objects and the contents of each container. Here is a sample run of the program:

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 2 3 5
Container 2 contains objects with weight 5
Container 3 contains objects with weight 7
Container 4 contains objects with weight 8
```

Does this program produce an optimal solution, that is, finding the minimum number of containers to pack the objects?

25.22 (*Bin packing with largest object first*) Rewrite the preceding program that places an object with the *largest weight* into the first bin in which it would fit. Give an example to show that this program does not produce an optimal solution.

25.23 (*Optimal bin packing*) Rewrite the preceding program so that it finds an optimal solution that packs all objects using the smallest number of containers. Here is a sample run of the program:

```
Enter the number of objects: 6 Enter
Enter the weights of the objects: 7 5 2 3 5 8 Enter
Container 1 contains objects with weight 7 3
Container 2 contains objects with weight 5 5
Container 3 contains objects with weight 2 8
The optimal number of bins is 3
```

What is the time complexity of your program?

CHAPTER

26

AVL TREES

Objectives

- To know what an AVL tree is (§26.1).
- To understand how to rebalance a tree using the LL rotation, LR rotation, RR rotation, and RL rotation (§26.2).
- To design the `AVLTree` class by extending the `BST` class (§26.3).
- To insert elements into an AVL tree (§26.4).
- To implement tree rebalancing (§26.5).
- To delete elements from an AVL tree (§26.6).
- To implement the `AVLTree` class (§26.7).
- To test the `AVLTree` class (§26.8).
- To analyze the complexity of search, insertion, and deletion operations in AVL trees (§26.9).