CHAPTER

25

Binary Search Trees

Objectives

- To design and implement a binary search tree (§25.2).
- To represent binary trees using linked data structures (§25.3).
- To search an element in a binary search tree (§25.4).
- To insert an element into a binary search tree (§25.5).
- To traverse elements in a binary tree (§25.6).
- To design and implement the `Tree` interface and the `BST` class (§25.7).
- To delete elements from a binary search tree (§25.8).
- To display a binary tree graphically (§25.9).
- To create iterators for traversing a binary tree (§25.10).
- To implement Huffman coding for compressing data using a binary tree (§25.11).

25.1 Introduction

A binary search tree is more efficient than a list for search, insertion, and deletion operations.

The preceding chapter gives the implementation for array lists and linked lists. The time complexity of search, insertion, and deletion operations in these data structures is $O(n)$. This chapter presents a new data structure called binary search tree, which takes $O(\log n)$ average time for search, insertion, and deletion of elements.

25.2 Binary Search Trees Basics

For every node in a binary search tree, the value of its left child is less than the value of the node, and the value of its right child is greater than the value of the node.

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, as shown in Figure 25.1a. Examples of binary trees are shown in Figures 25.1a and b.

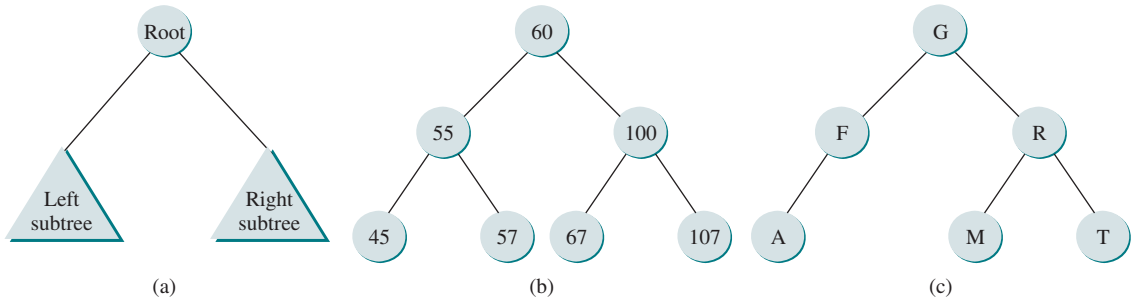


FIGURE 25.1 Each node in a binary tree has zero, one, or two subtrees.

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 25.1b. 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 its left child is less than the value of the node, and the value of its right child is greater than the value of the node. The binary trees in Figures 25.1b–c are all BSTs.

BST animation on Companion Website

Pedagogical Note

For an interactive GUI demo to see how a BST works, go to liveexample.pearsoncmg.com/dsanimation/BSTeBook.html, as shown in Figure 25.2.

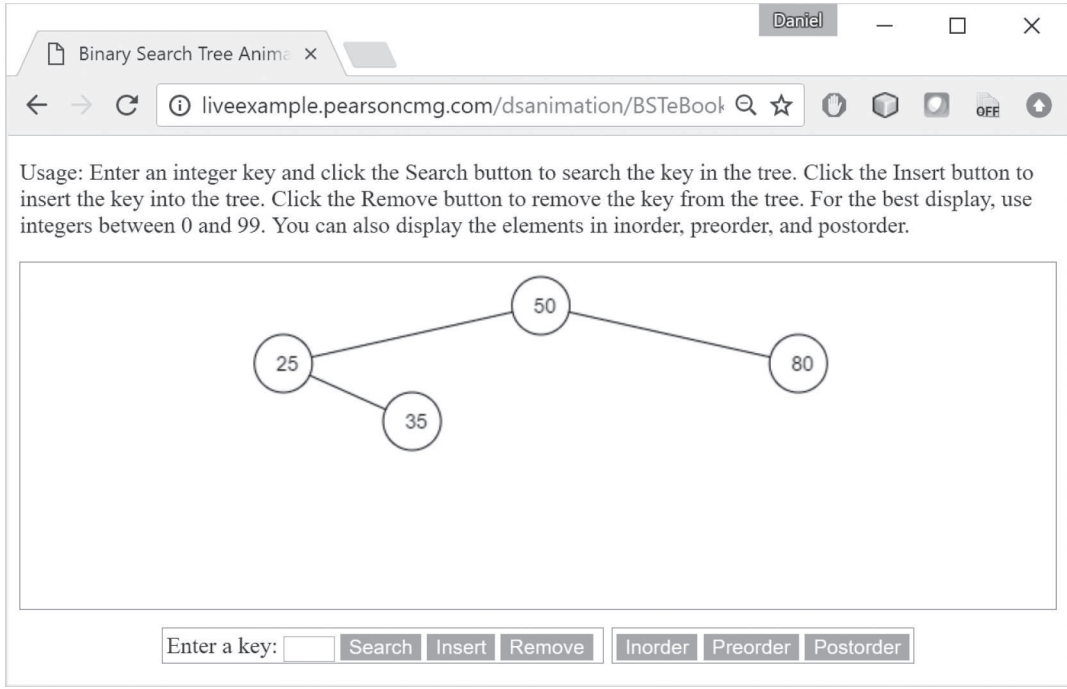


FIGURE 25.2 The animation tool enables you to insert, delete, and search elements. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

25.3 Representing Binary Search Trees

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

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.

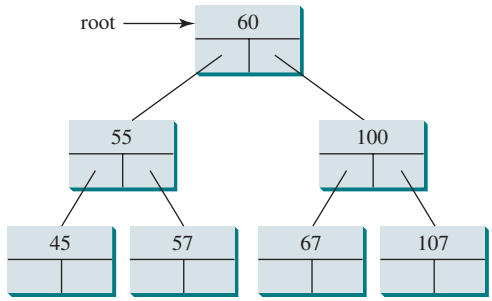


FIGURE 25.3 A binary tree can be represented using a set of linked nodes.

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;
  }
}
```

We use the variable `root` to refer 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);
```

25.4 Searching for an Element

BST enables an efficient search that resembles a binary search.

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 `e` is less than `current.element`, assign `current.left` to `current` (line 6).
- If `e` is greater than `current.element`, assign `current.right` to `current` (line 9).
- If `e` 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 e in a BST

start from root
left subtree
right subtree
found
not found

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

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

25.5 Inserting an Element into a BST

The new element is inserted at a leaf node.

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

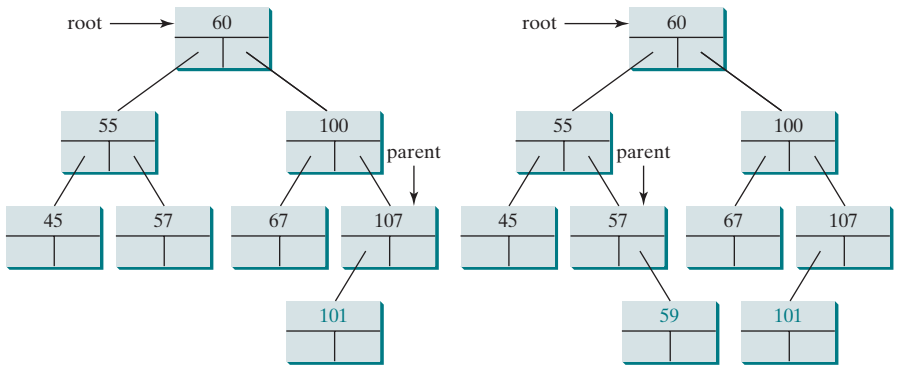
create a new node

```
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)                                  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 and 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.

25.5.1 Show the result of inserting 44 into Figure 25.4b.

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

Check Point

25.6 Tree Traversal

Inorder, preorder, postorder, depth-first, and breadth-first are common ways to traverse the elements in a binary tree.

Key Point

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

inorder 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, as shown in Figure 25.5.

postorder 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.

preorder 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.

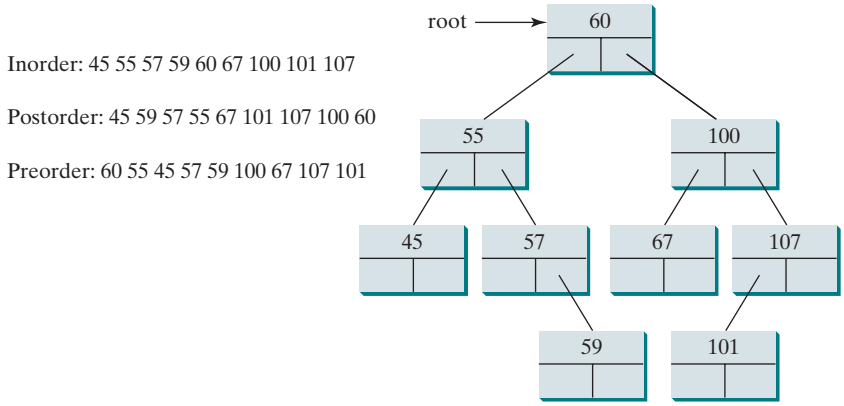


FIGURE 25.5 A tree traversal visits each node in a tree in a certain order.

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.

reconstruct a tree

depth-first traversal

Depth-first traversal is to visit the root then recursively visit its left subtree and right subtree in an arbitrary order. The preorder traversal can be viewed as a special case of depth-first traversal, which recursively visits its left subtree then its right subtree.

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 25.5, 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
```

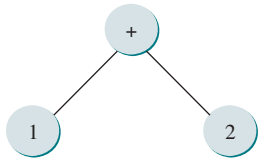
The depth-first is

```
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 simple tree to help remember inorder, postorder, and preorder.



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

25.6.1 Show the inorder, preorder, and postorder of traversing the elements in the binary tree shown in Figure 25.1c?

25.6.2 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?

25.7 The BST Class

The **BST** *class defines a data structure for storing and manipulating data in a binary search tree.*

Following the design pattern for Java Collections Framework and utilizing the default methods in Java 8, we use an interface named **Tree** to define all common operations for trees and define **Tree** to be a subtype of **Collection** so we can use common operations in **Collection** for trees, as shown in Figure 25.6. A concrete **BST** class can be defined to implement **Tree**, as shown in Figure 25.7.

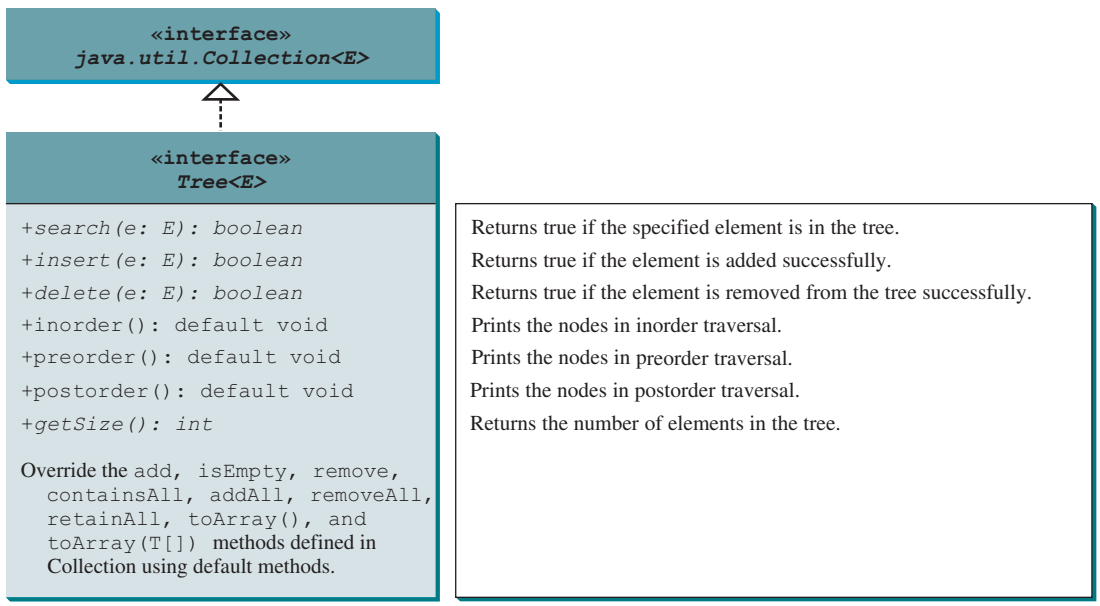


FIGURE 25.6 The **Tree** interface defines common operations for trees, and partially implements **Collection**.

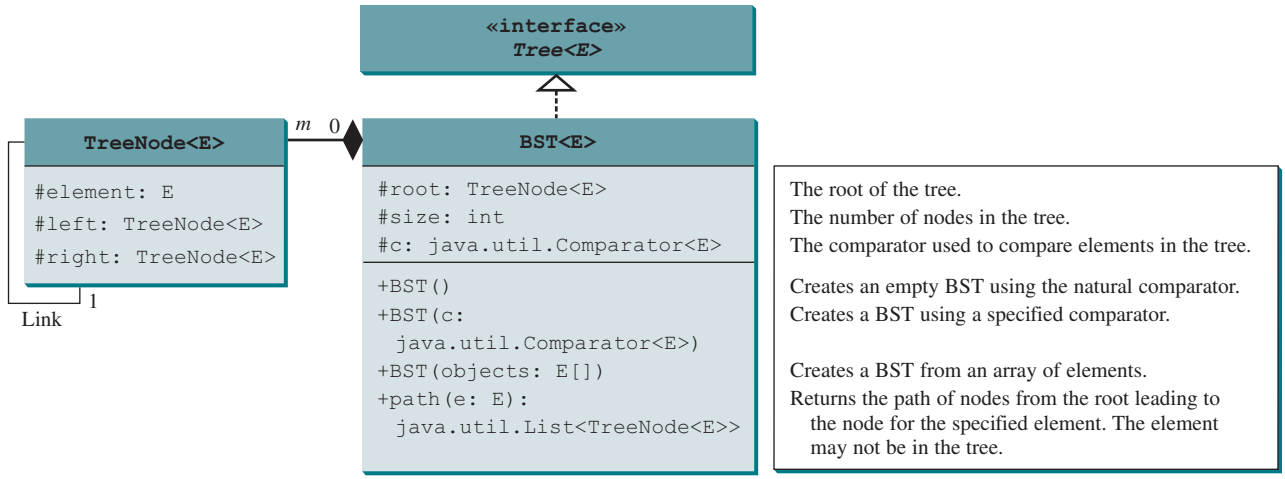


FIGURE 25.7 The **BST** class defines a concrete BST.

Listing 25.3 gives the implementation for **Tree**. It provides default implementations for the **add**, **isEmpty**, **remove**, **containsAll**, **addAll**, **removeAll**, **retainAll**, **toArray()**, and **toArray(T[])** methods inherited from the **Collection** interface as well as the **inorder()**, **preorder()**, and **postorder()** defined in the **Tree** interface.

LISTING 25.3 Tree.java

```
1  import java.util.Collection;
2
3  public interface Tree<E> extends Collection<E> {
4    /** Return true if the element is in the tree */
5    public boolean search(E e);
6
7    /** Insert element e into the binary tree
8     * Return true if the element is inserted successfully */
9    public boolean insert(E e);
10
11   /** Delete the specified element from the tree
12    * Return true if the element is deleted successfully */
13   public boolean delete(E e);
14
15   /** Get the number of elements in the tree */
16   public int getSize();
17
18   /** Inorder traversal from the root*/
19   public default void inorder() {
20   }
21
22   /** Postorder traversal from the root */
23   public default void postorder() {
24   }
25
26   /** Preorder traversal from the root */
27   public default void preorder() {
28   }
29
30   @Override /** Return true if the tree is empty */
31   public default boolean isEmpty() {
32     return size() == 0;
33   }
```

interface
search
insert
delete
getSize
inorder
postorder
preorder
default isEmpty

```
34
35    @Override
36    public default boolean contains(Object e) {                    default contains
37      return search((E)e);
38    }
39
40    @Override
41    public default boolean add(E e) {                              default add
42      return insert(e);
43    }
44
45    @Override
46    public default boolean remove(Object e) {                      default remove
47      return delete((E)e);
48    }
49
50    @Override
51    public default int size() {                                    default size
52      return getSize();
53    }
54
55    @Override
56    public default boolean containsAll(Collection<?> c) {          default containsAll
57      // Left as an exercise
58      return false;
59    }
60
61    @Override
62    public default boolean addAll(Collection<? extends E> c) {     default addAll
63      // Left as an exercise
64      return false;
65    }
66
67    @Override
68    public default boolean removeAll(Collection<?> c) {            default removeAll
69      // Left as an exercise
70      return false;
71    }
72
73    @Override
74    public default boolean retainAll(Collection<?> c) {            default retainAll
75      // Left as an exercise
76      return false;
77    }
78
79    @Override
80    public default Object[] toArray() {                            default toArray()
81      // Left as an exercise
82      return null;
83    }
84
85    @Override
86    public default <T> T[] toArray(T[] array) {                    default toArray(T[])
87      // Left as an exercise
88      return null;
89    }
90  }
```

Listing 25.4 gives the implementations for the **BST** class.

LISTING 25.4 BST.java

BST class
root
size
comparator
no-arg constructor
natural order comparator
constructor
constructor
search
compare objects
insert
new root
compare objects

```
1  public class BST<E> implements Tree<E> {
2    protected TreeNode<E> root;
3    protected int size = 0;
4    protected java.util.Comparator<E> c;
5
6    /** Create a default BST with a natural order comparator */
7    public BST() {
8      this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
9    }
10
11   /** Create a BST with a specified comparator */
12   public BST(java.util.Comparator<E> c) {
13     this.c = c;
14   }
15
16   /** Create a binary tree from an array of objects */
17   public BST(E[] objects) {
18     this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);
19     for (int i = 0; i < objects.length; i++)
20       add(objects[i]);
21   }
22
23   @Override /** Return true if the element is in the tree */
24   public boolean search(E e) {
25     TreeNode<E> current = root; // Start from the root
26
27     while (current != null) {
28       if (c.compare(e, current.element) < 0) {
29         current = current.left;
30       }
31       else if (c.compare(e, current.element) > 0) {
32         current = current.right;
33       }
34       else // element matches current.element
35         return true; // Element is found
36     }
37
38     return false;
39   }
40
41   @Override /** Insert element e into the binary tree
42    * Return true if the element is inserted successfully */
43   public boolean insert(E e) {
44     if (root == null)
45       root = createNewNode(e); // Create a new root
46     else {
47       // Locate the parent node
48       TreeNode<E> parent = null;
49       TreeNode<E> current = root;
50       while (current != null)
51         if (c.compare(e, current.element) < 0) {
52           parent = current;
53           current = current.left;
54         }
55         else if (c.compare(e, current.element) > 0) {
56           parent = current;
57           current = current.right;
58         }
59         else
```

```
60            return false; // Duplicate node not inserted
61
62          // Create the new node and attach it to the parent node
63          if (c.compare(e, parent.element) < 0)                    link to parent
64            parent.left = createNewNode(e);
65          else
66            parent.right = createNewNode(e);
67        }
68
69        size++;                                                    increase size
70        return true; // Element inserted successfully
71      }
72
73      protected TreeNode<E> createNewNode(E e) {                   create new node
74        return new TreeNode<>(e);
75      }
76
77      @Override /** Inorder traversal from the root */
78      public void inorder() {                                      inorder
79        inorder(root);
80      }
81
82      /** Inorder traversal from a subtree */
83      protected void inorder(TreeNode<E> root) {                   recursive helper method
84        if (root == null) return;
85        inorder(root.left);
86        System.out.print(root.element + " ");
87        inorder(root.right);
88      }
89
90      @Override /** Postorder traversal from the root */
91      public void postorder() {                                    postorder
92        postorder(root);
93      }
94
95      /** Postorder traversal from a subtree */
96      protected void postorder(TreeNode<E> root) {                 recursive helper method
97        if (root == null) return;
98        postorder(root.left);
99        postorder(root.right);
100       System.out.print(root.element + " ");
101     }
102
103     @Override /** Preorder traversal from the root */
104     public void preorder() {                                     preorder
105       preorder(root);
106     }
107
108     /** Preorder traversal from a subtree */
109     protected void preorder(TreeNode<E> root) {                  recursive helper method
110       if (root == null) return;
111       System.out.print(root.element + " ");
112       preorder(root.left);
113       preorder(root.right);
114     }
115
116     /** This inner class is static, because it does not access
117         any instance members defined in its outer class */
118     public static class TreeNode<E> {                            inner class
119       protected E element;
120       protected TreeNode<E> left;
```

```
121       protected TreeNode<E> right;
122
123       public TreeNode(E e) {
124         element = e;
125       }
126     }
127
128     @Override /** Get the number of nodes in the tree */
getSize 129     public int getSize() {
130       return size;
131     }
132
133     /** Returns the root of the tree */
getRoot 134     public TreeNode<E> getRoot() {
135       return root;
136     }
137
138    /** Returns a path from the root leading to the specified element */
path    139     public java.util.ArrayList<TreeNode<E>> path(E e) {
140       java.util.ArrayList<TreeNode<E>> list =
141         new java.util.ArrayList<>();
142       TreeNode<E> current = root; // Start from the root
143
144       while (current != null) {
145         list.add(current); // Add the node to the list
146         if (c.compare(e, current.element) < 0) {
147           current = current.left;
148         }
149         else if (c.compare(e, current.element) > 0) {
150           current = current.right;
151         }
152         else
153           break;
154       }
155
156       return list; // Return an array list of nodes
157     }
158
159     @Override /** Delete an element from the binary tree.
160      * Return true if the element is deleted successfully
161      * Return false if the element is not in the tree */
delete  162     public boolean delete(E e) {
163      // Locate the node to be deleted and also locate its parent node
locate parent 164       TreeNode<E> parent = null;
locate current 165       TreeNode<E> current = root;
166       while (current != null) {
167         if (c.compare(e, current.element) < 0) {
168           parent = current;
169           current = current.left;
170         }
171         else if (c.compare(e, current.element) > 0) {
172           parent = current;
173           current = current.right;
174         }
175         else
current found 176           break; // Element is in the tree pointed at by current
177       }
178
not found 179       if (current == null)
```

```
180       return false; // Element is not in the tree
181
182     // Case 1: current has no left child
183     if (current.left == null) {                                  Case 1
184       // Connect the parent with the right child of the current node
185       if (parent == null) {
186         root = current.right;
187       }
188       else {
189         if (c.compare(e, parent.element) < 0)
190           parent.left = current.right;                           reconnect parent
191         else
192           parent.right = current.right;                          reconnect parent
193       }
194     }
195     else {
196       // Case 2: The current node has a left child                Case 2
197       // Locate the rightmost node in the left subtree of
198       // the current node and also its parent
199       TreeNode<E> parentOfRightMost = current;                   locate parentOfRightMost
200       TreeNode<E> rightMost = current.left;                      locate rightMost
201
202       while (rightMost.right != null) {
203         parentOfRightMost = rightMost;
204         rightMost = rightMost.right; // Keep going to the right
205       }
206
207       // Replace the element in current by the element in rightMost
208       current.element = rightMost.element;                       replace current
209
210       // Eliminate rightmost node
211       if (parentOfRightMost.right == rightMost)
212         parentOfRightMost.right = rightMost.left;
213       else                                                       reconnect parentOfRightMost
214         // Special case: parentOfRightMost == current
215         parentOfRightMost.left = rightMost.left;
216     }
217
218     size--;                                                      reduce size
219     return true; // Element deleted successfully                 successful deletion
220   }
221
222   @Override /** Obtain an iterator. Use inorder. */
223   public java.util.Iterator<E> iterator() {                      iterator
224     return new InorderIterator();
225   }
226
227   // Inner class InorderIterator
228   private class InorderIterator implements java.util.Iterator<E> {  iterator class
229     // Store the elements in a list
230     private java.util.ArrayList<E> list =                        internal list
231       new java.util.ArrayList<>();
232    private int current = 0; // Point to the current element in list  current position
233
234     public InorderIterator() {
235       inorder(); // Traverse binary tree and store elements in list
236     }
237
238     /** Inorder traversal from the root*/
239     private void inorder() {
240       inorder(root);                                             obtain inorder list
```

```
241       }
242
243       /** Inorder traversal from a subtree */
244       private void inorder(TreeNode<E> root) {
245         if (root == null) return;
246         inorder(root.left);
247         list.add(root.element);
248         inorder(root.right);
249       }
250
251       @Override /** More elements for traversing? */
252       public boolean hasNext() {
253         if (current < list.size())
254           return true;
255
256         return false;
257       }
258
259       @Override /** Get the current element and move to the next */
260       public E next() {
261         return list.get(current++);
262       }
263
264       @Override // Remove the element returned by the last next()
265       public void remove() {
266          if (current == 0) // next() has not been called yet
267           throw new IllegalStateException();
268
269          delete(list.get(--current));
270         list.clear(); // Clear the list
271         inorder(); // Rebuild the list
272       }
273     }
274
275     @Override /** Remove all elements from the tree */
276     public void clear() {
277       root = null;
278       size = 0;
279     }
280   }
```

hasNext in iterator?

get next element

remove the current

clear list
refresh list

tree clear method

The **insert(E e)** method (lines 43–71) 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 78–88) 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 91–101) and the **preorder()** method (lines 104–114) are implemented similarly using recursion.

The **path(E e)** method (lines 139–157) 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 162–273) will be discussed in Sections 25.8 and 25.10.

Design Pattern Note

The design for the `createNewNode()` method applies the factory method pattern, which creates an object returned from the method rather than directly using the constructor in the code to create the object. Suppose the factory method returns an object of the type `A`. This design enables you to override the method to create an object of the subtype of `A`.The `createNewNode()` method in the `BST` class returns a `TreeNode` object. Later in later chapters, we will override this method to return an object of the subtype of `TreeNode`.

Listing 25.5 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 25.5 TestBST.java

```
1  public class TestBST {
2    public static void main(String[] args) {
3      // Create a BST
4      BST<String> tree = new BST<>();                                  create tree
5      tree.insert("George");                                           insert
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();                                                  inorder
16     System.out.print("\nPostorder: ");
17     tree.postorder();                                                postorder
18     System.out.print("\nPreorder: ");
19     tree.preorder();                                                 preorder
20     System.out.print("\nThe number of nodes is " + tree.getSize());  getSize
21
22     // Search for an element
23     System.out.print("\nIs Peter in the tree? " +
24       tree.search("Peter"));                                         search
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 }
```

```
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 25.8.

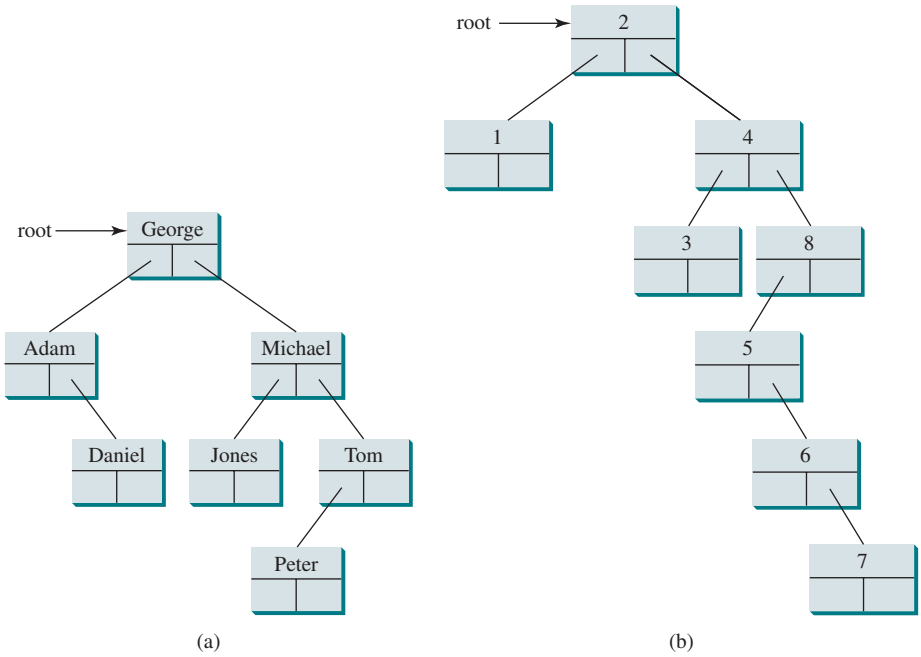


FIGURE 25.8 The BSTs in Listing 25.5 are pictured here after they are created.

If the elements are inserted in a different order (e.g., Daniel, Adam, Jones, Peter, Tom, Michael, and 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.

25.7.1 Write the code that creates a `BST` for integers, inserts numbers `9`, `3`, `12`, `30` to the tree, and deletes `9`, display the tree in order, preorder, and postorder, searches for `4`, and displays the number of elements in the tree.

25.7.2 Add a new method named `getSmallest()` in the `BST` class that returns the smallest element in the tree.

25.7.3 Add a new method named `getLargest()` in the `BST` class that returns the largest element in the tree.

25.8 Deleting Elements from a BST

To delete an element from a BST, first locate it in the tree 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 is presented in Section 25.5. 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.

locating element

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.

Case 1: The current node does not have a left child, as shown in Figure 25.9a. In this case, simply connect the parent with the right child of the current node, as shown in Figure 25.9b.

For example, to delete node `10` in Figure 25.10a, you would connect the parent of node `10` with the right child of node `10`, as shown in Figure 25.10b.

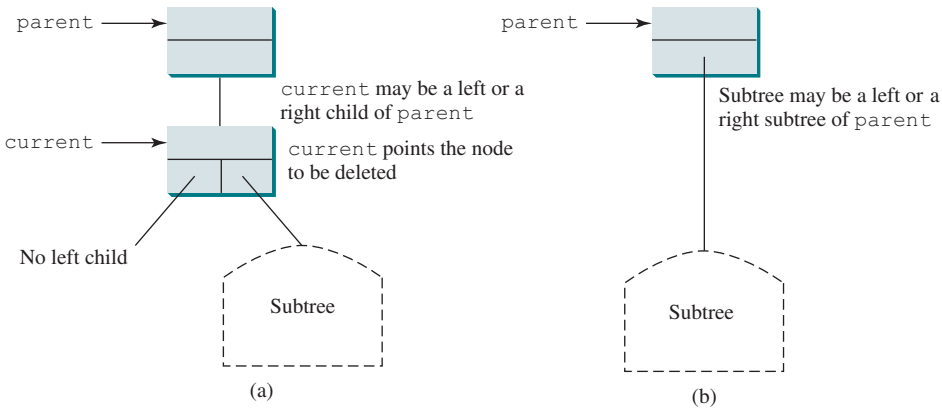


FIGURE 25.9 Case 1: The current node has no left child.

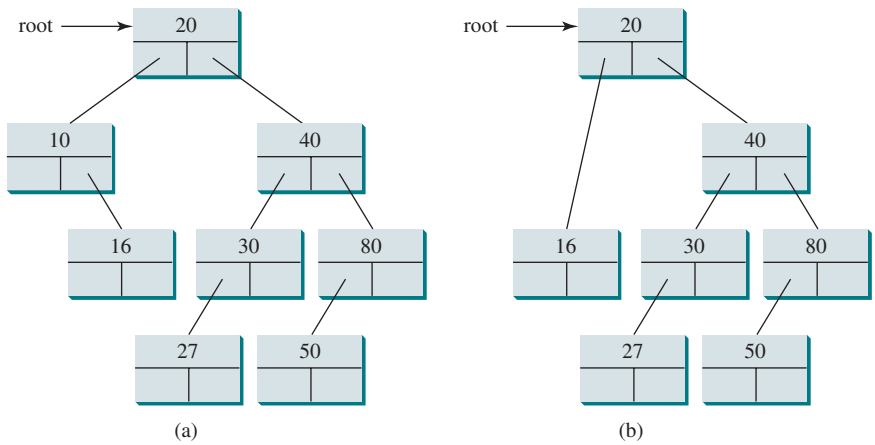


FIGURE 25.10 Case 1: Deleting node `10` from (a) results in (b).

Note

If the current node is a leaf, it falls into Case 1. For example, to delete element `16` in Figure 25.10a, connect its right child (in this case, it is `null`) to the parent of node `16`.

delete a leaf

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 25.11a. Note 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 25.11b.

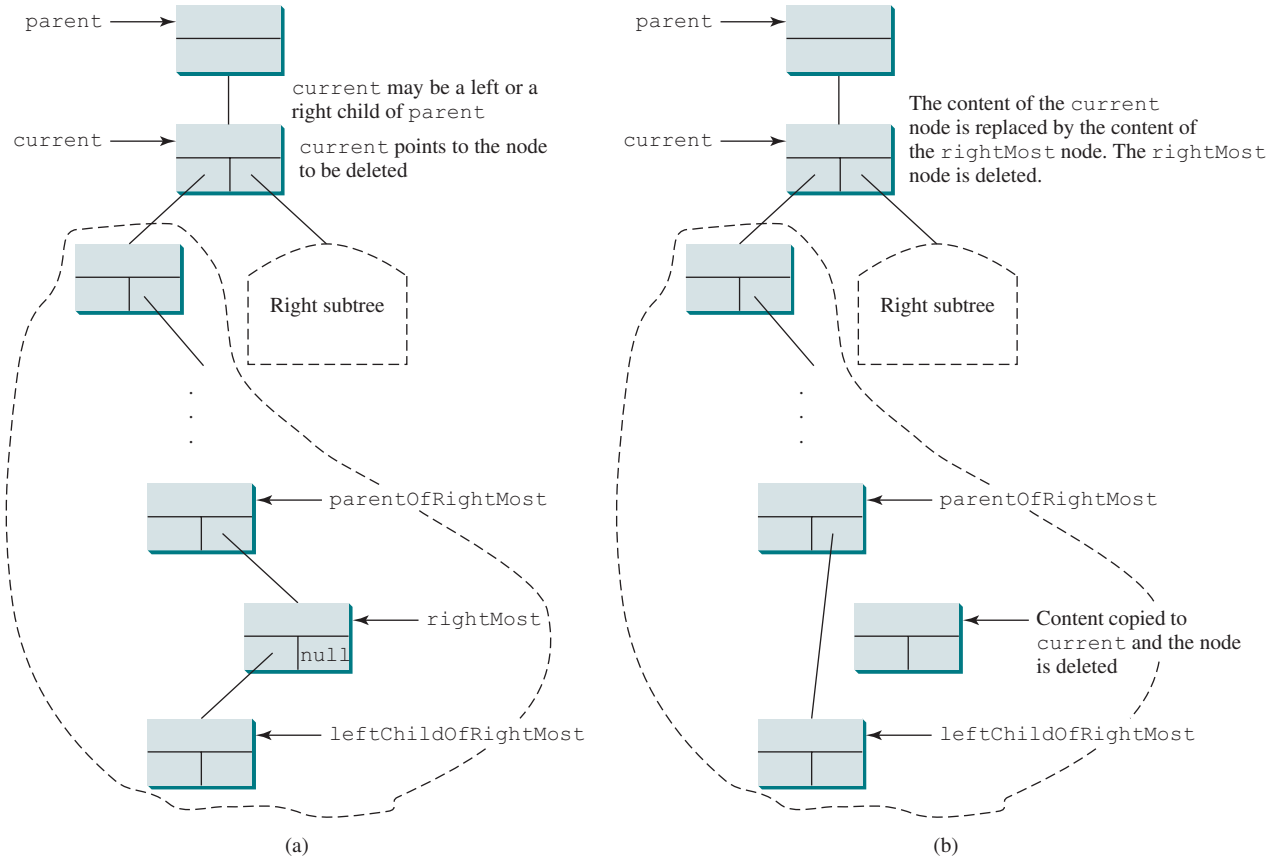


FIGURE 25.11 Case 2: The current node has a left child.

For example, consider deleting node **20** in Figure 25.12a. 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 25.12b.

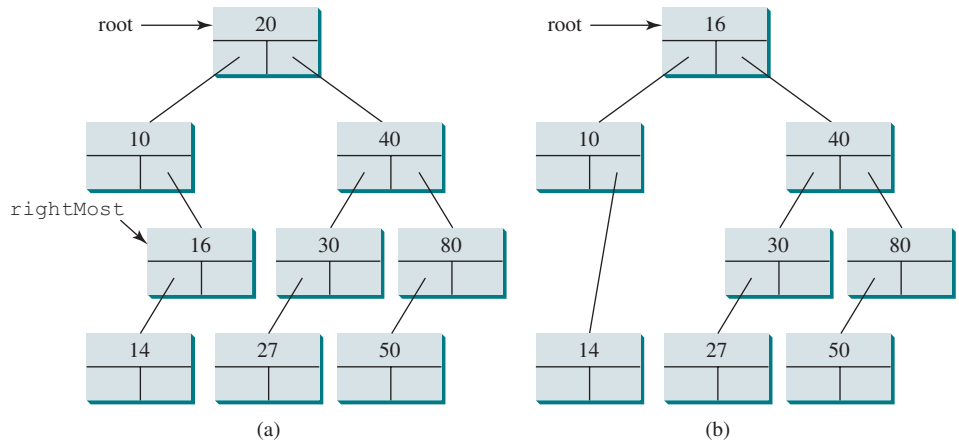


FIGURE 25.12 Case 2: Deleting node **20** from (a) results in (b).

Note

If the left child of **current** does not have a right child, **current.left** points to the largest 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 left child of **rightMost** with **parentOfRightMost**.

special case

The algorithm for deleting an element from a binary search tree can be described in Listing 25.6.

`delete` method

LISTING 25.6 Deleting an Element from a BST

```
boolean delete(E e) {
  Locate element e in the tree;                                  not in the tree
  if element e is not found
    return false;

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

  if (current has no left child)  // Case 1                      Case 1
    Connect the right child of current with parent;
      Now current is not referenced, so it is eliminated;
  else // Case 2                                                 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
}
```

The complete implementation of the **`delete`** method is given in lines 162–220 in Listing 25.4. The method locates the node (named **`current`**) to be deleted and also locates its parent (named **`parent`**) in lines 164–177. If **`current`** is **`null`**, the element is not in the tree. Therefore, the method returns **`false`** (line 180). 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 183–194. 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 185–187). Otherwise, assign **`current.right`** to either **`parent.left`** or **`parent.right`**, depending on whether **`current`** is a left or a right child of **`parent`** (189–192).

Case 2 of the algorithm is covered in lines 195–216. 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 199–205). Replace the element in **`current`** by the element in **`rightMost`** (line 208); assign **`rightMost.left`** to either **`parentOfRightMost.right`** or **`parentOfRightMost.left`** (lines 211–215), depending on whether **`rightMost`** is a right or a left child of **`parentOfRightMost`**.

Listing 25.7 gives a test program that deletes the elements from the binary search tree.

LISTING 25.7 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:");
```

delete an element

```
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");
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  }
```

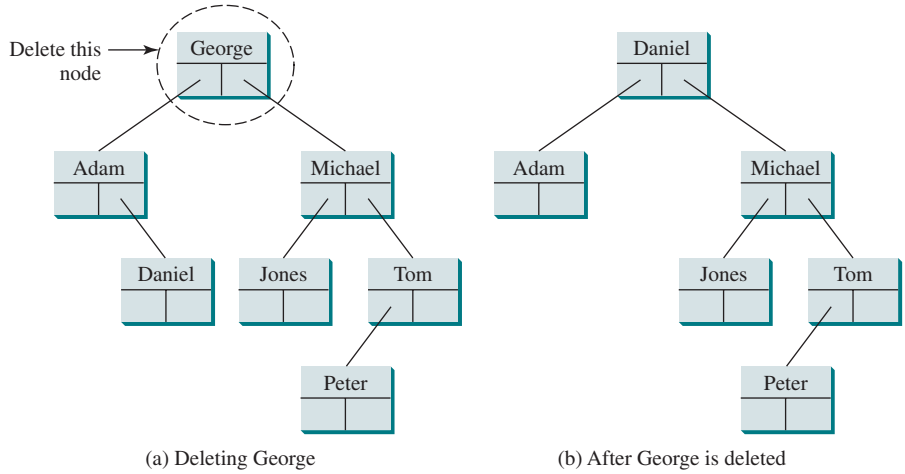
```
Inorder (sorted): Adam Daniel George Jones Michael Peter Tom
Postorder: Daniel Adam Jones Peter Tom Michael George
Preorder: George Adam Daniel Michael Jones Tom Peter
The number of nodes is 7

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 25.12–25.14 show how the tree evolves as the elements are deleted.

BST time complexity

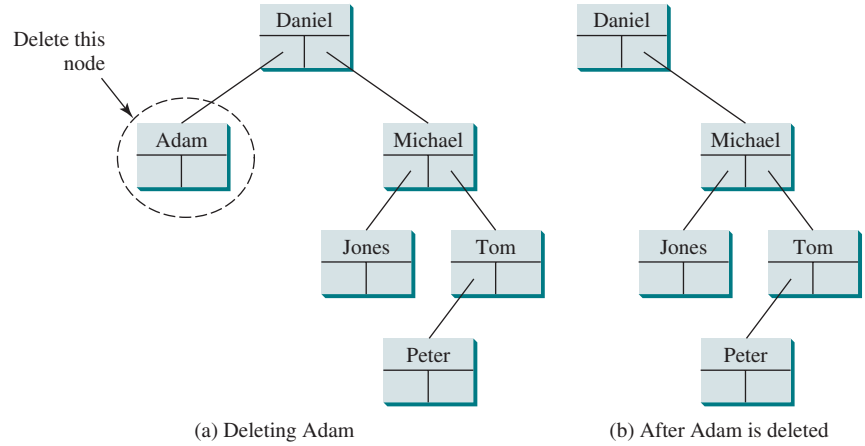
Note

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 $O(n)$. On average, the height of the tree is $O(\log n)$. So, the average time for search, insertion, deletion in a BST is $O(\log n)$.



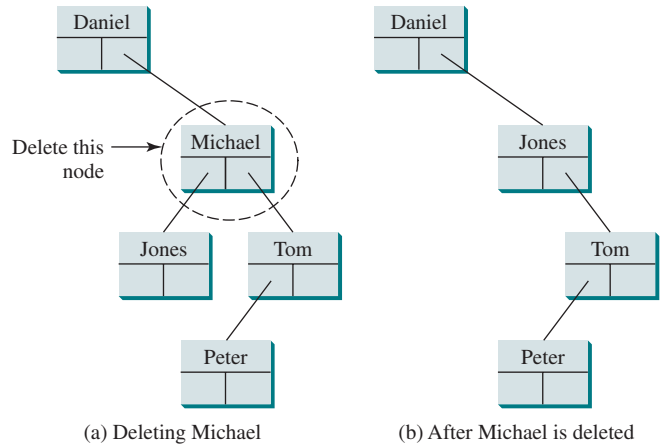
(a) Deleting George (b) After George is deleted

FIGURE 25.13 Deleting George falls into Case 2.



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

FIGURE 25.14 Deleting Adam falls into Case 1.



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

FIGURE 25.15 Deleting Michael falls into Case 2.

Check Point

25.8.1 Show the result of deleting **55** from the tree in Figure 25.4b.

25.8.2 Show the result of deleting **60** from the tree in Figure 25.4b.

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

25.8.4 Is the algorithm correct if lines 203–207 in Listing 25.4 in Case 2 of the `delete()` method are replaced by the following code?

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

25.9 Tree Visualization and MVC

Key Point

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 in Listing 18.9 can be applied to displaying a binary tree. For simplicity, we assume that the keys are positive integers less than **100**. Listing 25.8 and Listing 25.9 give the program, and Figure 25.15 shows some sample runs of the program.

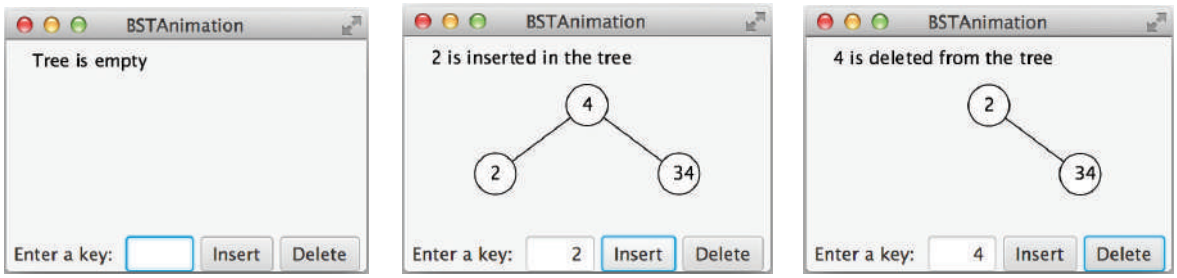


FIGURE 25.16 A binary tree is displayed graphically. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

LISTING 25.8 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
15
16     BorderPane pane = new BorderPane();
17     BTView view = new BTView(tree); // Create a View
```

create a tree

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);                                          place hBox
30
31      btInsert.setOnAction(e -> {                                    handle insertion
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        }
37        else {
38          tree.insert(key); // Insert a new key                      insert key
39          view.displayTree();                                        display the tree
40          view.setStatus(key + " is inserted in the tree");
41        }
42      });
43
44      btDelete.setOnAction(e -> {                                    handle deletion
45        int key = Integer.parseInt(tfKey.getText());
46        if (!tree.search(key)) {  // key is not in the tree
47          view.displayTree();
48          view.setStatus(key + " is not in the tree");
49        }
50        else {
51          tree.delete(key); // Delete a key                          delete key
52          view.displayTree();                                        display the tree
53          view.setStatus(key + " is deleted from the tree");
54        }
55      });
56
57      // Create a scene and place the pane in the stage
58      Scene scene = new Scene(pane, 450, 250);
59      primaryStage.setTitle("BSTAnimation"); // Set the stage title
60      primaryStage.setScene(scene); // Place the scene in the stage
61      primaryStage.show(); // Display the stage
62    }
63  }                                                                  main method omitted
```

LISTING 25.9 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<>();                         tree to display
9     private double radius = 15; // Tree node radius
10    private double vGap = 50; // Gap between two levels in a tree
11
```

set a tree

clear the display

display tree recursively

connect two nodes

draw left subtree

connect two nodes

draw right subtree

display a node

```
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
21    public void displayTree() {
22      this.getChildren().clear(); // Clear the pane
23      if (tree.getRoot() != null) {
24        // Display tree recursively
25        displayTree(tree.getRoot(), getWidth() / 2, vGap,
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));
36        // Draw the left subtree recursively
37        displayTree(root.left, x - hGap, y + vGap, hGap / 2);
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));
43        // Draw the right subtree recursively
44        displayTree(root.right, x + hGap, y + vGap, hGap / 2);
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,
52        new Text(x - 4, y + 4, root.element + ""));
53    }
54  }
```

In Listing 25.8, 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 38), the tree is repainted (line 39) to reflect the change. After a key is deleted (line 51), the tree is repainted (line 52) to reflect the change.

In Listing 25.9, 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 37 and 44). 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 and 42). Note the method first adds the lines to the pane then adds the circle into the pane (line 52) so the circles will be painted on top of the lines to achieve desired visual effects.

The program assumes 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 25.16.

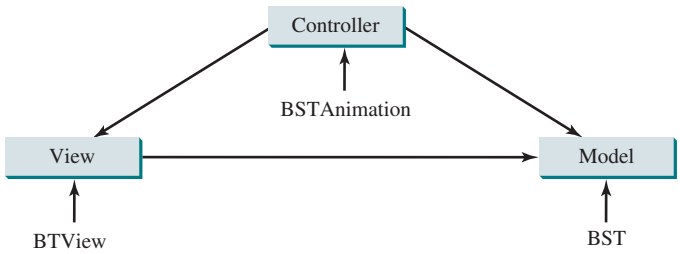


FIGURE 25.17 The controller obtains data and stores it in a model. The view displays the data stored in the model.

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 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 the 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.

25.9.1 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?

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

25.9.3 What would happen if the code in lines 47–52 in Listing 25.9, BTView.java is moved to line 33?

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

25.10 Iterators

`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 25.17) defines the common features of iterators.

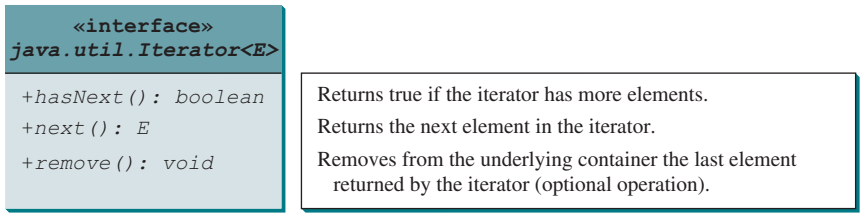


FIGURE 25.18 The `Iterator` interface defines a uniform way of traversing the elements in a container.

The `Tree` interface extends `java.util.Collection`. Since `Collection` extends `java.lang.Iterable`, `BST` is also a subclass 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 25.4 (lines 228–273). The `iterator` method simply returns an instance of `InorderIterator` (line 224).

how to create an iterator

The `InorderIterator` constructor invokes the `inorder` method (line 240). The `inorder(root)` method (lines 239–249) 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 232), 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 261).

The `hasNext()` method checks whether `current` is still in the range of `list` (line 253).

The `remove()` method removes the element returned by the last `next()` (line 269). Afterward, a new list is created (lines 270–271). Note that `current` does not need to be changed.

Listing 25.10 gives a test program that stores the strings in a BST and displays all strings in uppercase.

LISTING 25.10 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)
13       System.out.print(s.toUpperCase() + " ");
14   }
15 }
```

use an iterator
get uppercase letters

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

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

Design Guide

iterator pattern
advantages of iterators

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.

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 (lines 270-271 in Listing 25.4, BST.java). The client should always use the `delete` method in the `BST` 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 $O(\log n)$.

25.10.1 What is an iterator?

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

25.10.3 Suppose you delete `implements Collection<E>` from line 3 in Listing 25.3, Tree.java. Will Listing 25.10 still compile?

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

25.10.5 Write one statement that displays the maximum and minimum element in a `BST` object named `tree` (*Hint*: use the `min` and `max` methods in the `java.util.Collections` class).

25.11 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.

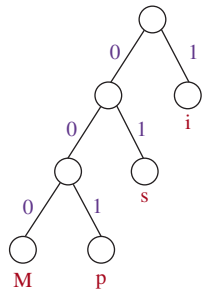
Compressing data is a common task. There are many utilities available for compressing files. This section introduces Huffman coding, invented by David Huffman in 1952.

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 is as shown in Figure 25.18a. The left and right edges of a node are assigned the values `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 25.18b. Since `i` and `s` appear more than `M` and `p` in the text, they are assigned shorter codes.

Huffman coding

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

decoding



(a) Huffman coding tree

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

(b) Character code table

FIGURE 25.19 The codes for characters are constructed based on the occurrence of characters in the text using a coding tree.

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`.

Based on the coding scheme in Figure 25.19,

`is encoded to` `is decoded to`

Mississippi ==========> 000101011010110010011 ==========> Mississippi

construct coding tree

To construct a *Huffman coding tree*, use the following algorithm:

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 25.19b. Initially, the forest contains single-node trees, as shown in Figure 25.20a. The trees are repeatedly combined to form large trees until only one tree is left, as shown in Figures 25.20b–d.

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.

greedy algorithm

Algorithm Design Note

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 of a greedy algorithm, 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 11.19.

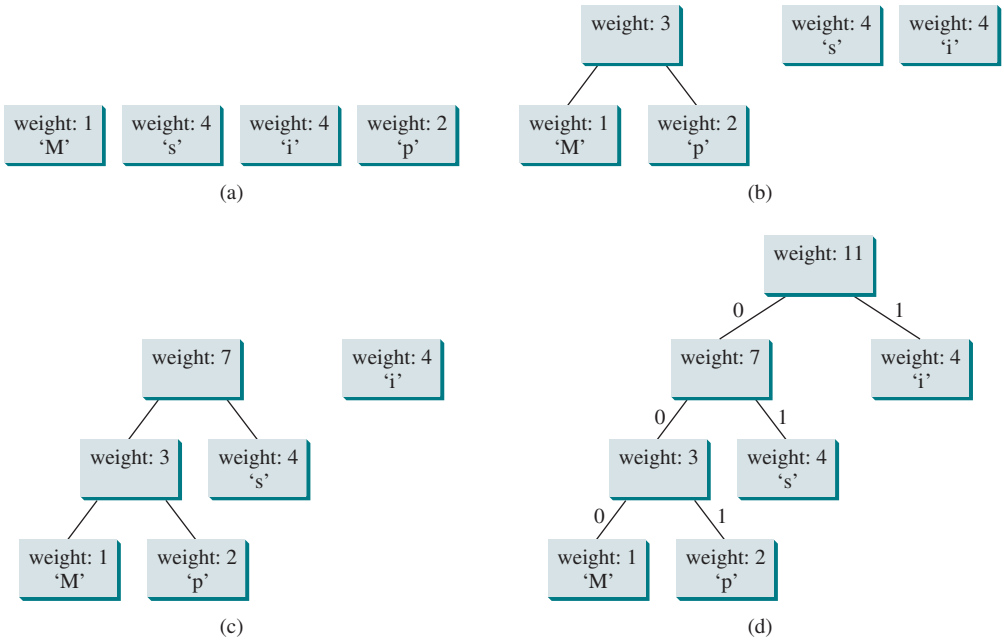


FIGURE 25.20 The coding tree is built by repeatedly combining the two smallest-weighted trees.

Listing 25.11 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 25.11 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      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       get Huffman tree
15     String[] codes = getCode(tree.root); // Get codes                  code for each character
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",
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    */
```

getCode

```
26    public static String[] getCode(Tree.Node root) {
27      if (root == null)  return null;
28      String[] codes = new String[128];
29      assignCode(root, codes);
30      return codes;
31    }
32
33    /* Recursively get codes to the leaf node */
```

assignCode

```
34    private static void assignCode(Tree.Node root, String[] codes) {
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 */
```

getHuffmanTree

```
48    public static Tree getHuffmanTree(int[] counts) {
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 */
```

getCharacterFrequency

```
66    public static int[] getCharacterFrequency(String text) {
67      int[] counts = new int[128]; // 128 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 */
```

Huffman tree

```
76    public static class Tree implements Comparable<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 {                                          tree 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 }
```

```
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

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 128 ASCII characters in the text. If a character appears in the text, its corresponding count is increased by **`1`** (line 70). `getCharacterFrequency`

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 `Node` class

Tree class

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 the smallest-weight tree is removed first from the heap of trees.

getHuffmanTree

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, 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.

assignCode
getCode

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 `127`. Note `codes[i]` is `null` if `(char)i` is not in the text.

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

25.11.2 What is a greedy algorithm? Give an example.

25.11.3 If the `Heap` class in line 50 in Listing 25.9 is replaced by `java.util.PriorityQueue`, will the program still work?

25.11.4 How do you replace lines 94–99 in Listing 25.11 using one line?

Key Terms

binary search tree 960
binary tree 960
breadth-first traversal 964
depth 960
depth-first traversal 964
greedy algorithm 986
height 960
Huffman coding 985
inorder traversal 963
leaf 960
length 960
level 960
postorder traversal 964
preorder traversal 964
sibling 960
tree traversal 963

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 the book Companion Website.

Programming Exercises

MyProgrammingLab™

Sections 25.2–25.6

***25.1** (*Tree height*) Define a new class named `BSTWithHeight` that extends `BST` with the following method:

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

Use https://liveexample.pearsoncmg.com/test/Exercise25_01.txt to test your code.

****25.2** (*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.

***25.3** (*Test perfect binary tree*) A perfect binary tree is a complete binary tree with all levels fully filled. Define a new class named `BSTWithTestPerfect` that extends `BST` with the following methods: (*Hint*: The number of nodes in a perfect binary tree is $2^{\text{height}+1} - 1$).

```
/** Returns true if the tree is a perfect binary tree */
public boolean isPerfectBST()
```

Use https://liveexample.pearsoncmg.com/test/Exercise25_03.txt to test your code.

****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*) Define a new class named `BSTWithNumberOfLeaves` that extends `BST` with the following methods:

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

Use https://liveexample.pearsoncmg.com/test/Exercise25_06.txt to test your code.

****25.7** (*Find the nonleaves*) Define a new class named `BSTWithNumberOfNonLeaves` that extends `BST` with the following methods:

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

Use https://liveexample.pearsoncmg.com/test/Exercise25_07.txt to test your code.

*****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.

```
/** Return 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 25.21.

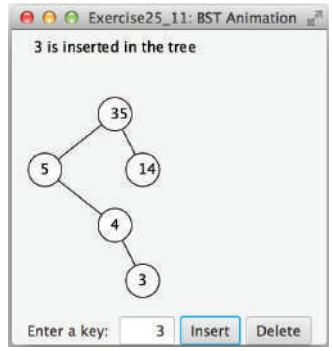


FIGURE 25.21 A binary tree is displayed horizontally.

**25.12 (*Test `BST`*) Design and write a complete test program to test if the `BST` class in Listing 25.4 meets all requirements.

**25.13 (*Add new buttons in `BSTAnimation`*) Modify Listing 25.8, 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 25.22. You need also to modify Listing 25.4, BST.java to implement the `inorderList()`, `preorderList()`, and `postorderList()` methods so 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();
```

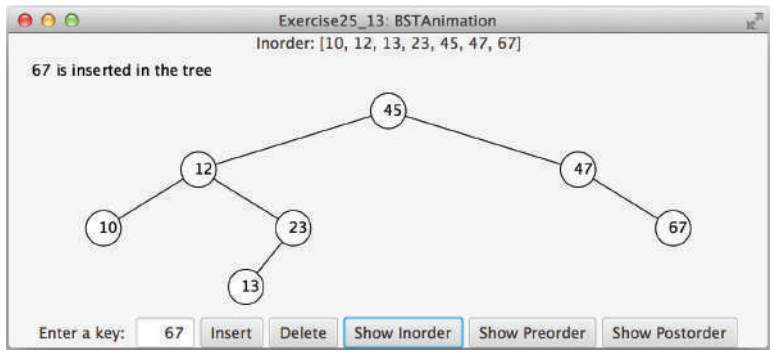


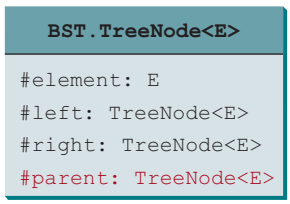
FIGURE 25.22 When you click the Show Inorder, Show Preorder, or Show Postorder button, the elements are displayed in an inorder, preorder, or postorder in a label. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***25.14** (*Breadth-first traversal*) Define a new class named **BSTWithBFT** that extends **BST** with the following method:

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

Use https://liveexample.pearsoncmg.com/test/Exercise25_14.txt to test your code.

*****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**:

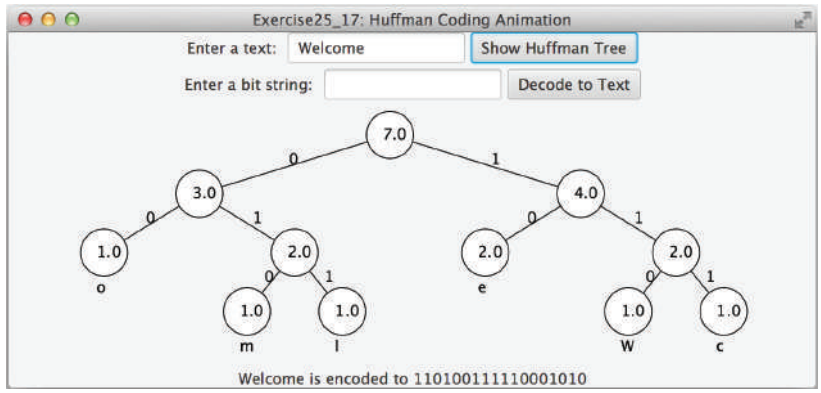
```
/** Return the node for the specified element.
  * Return null if the element is not in the tree. */
private TreeNode<E> getNode(E element)

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

/** Return 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:

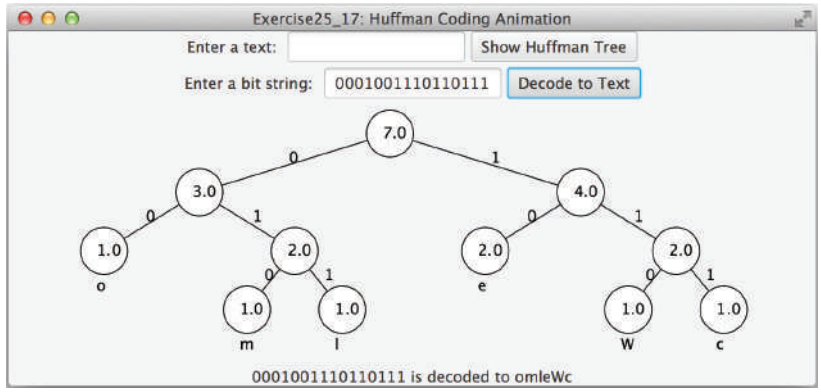
```
Enter 10 integers: 45 54 67 56 50 45 23 59 23 67 Enter
[50, 54, 23]
[59, 56, 67, 54, 23]
```

*****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 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 25.23a. 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 25.23b.



(a)



(b)

FIGURE 25.23 (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. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****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, 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
```