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.

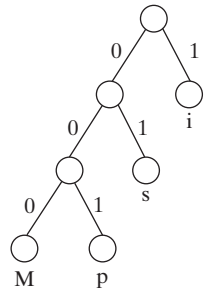
Compressing data is a common task. There are many utilities available for compressing files. This section introduces Huffman coding, invented by David Huffman in 1952.

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	========> 00010101101011001011	========> 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.

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`.

decoding

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

construct coding tree

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.

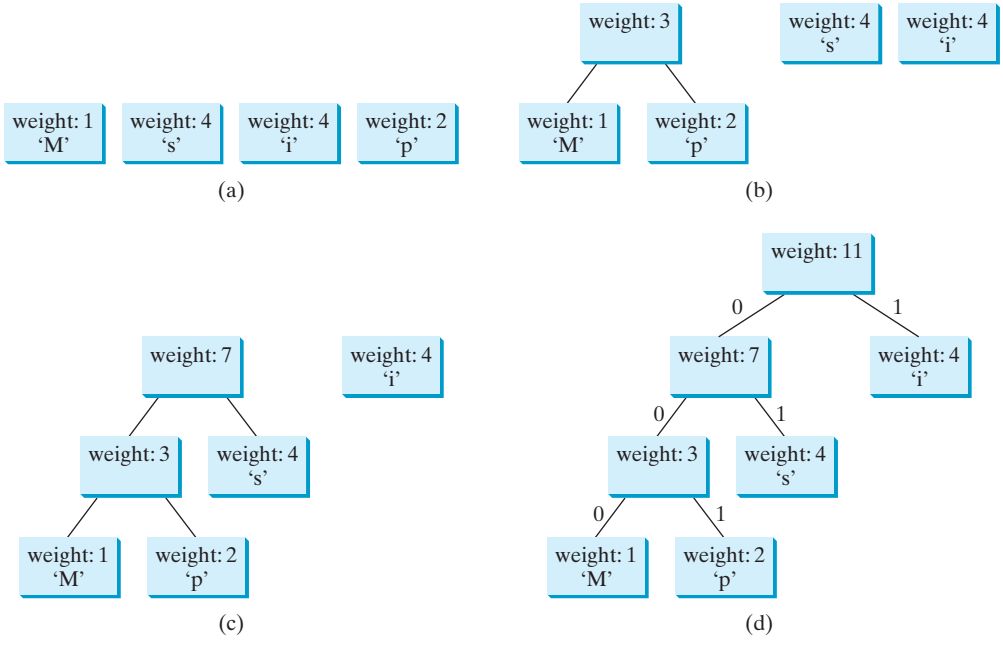


FIGURE 25.21 The coding tree is built by repeatedly combining the two smallest-weighted trees.

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

prefix property

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

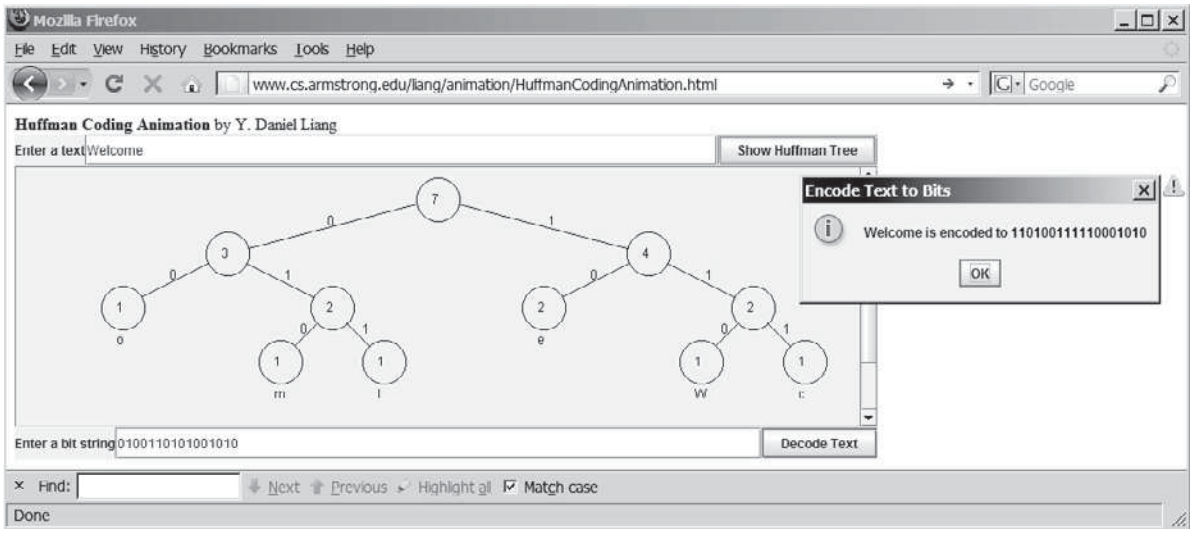


FIGURE 25.22 The animation tool enables you to create and view a Huffman tree, and it performs encoding and decoding using the tree.

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.