LISTING 8.1 Continued

```
      {
      String s = getString();
      return Integer.parseInt(s);
      }
// -------------------------------------------------------------
   }  // end class TreeApp
////////////////////////////////////////////////////////////////
```

You can use the control+C key combination to exit from this program; in the interest of simplicity there's no single-letter key for this action.

The Huffman Code

You shouldn't get the idea that binary trees are always search trees. Many binary trees are used in other ways. We saw an example in Figure 8.11, where a binary tree represents an algebraic expression.

In this section we'll discuss an algorithm that uses a binary tree in a surprising way to compress data. It's called the Huffman code, after David Huffman who discovered it in 1952. Data compression is important in many situations. An example is sending data over the Internet, where, especially over a dial-up connection, transmission can take a long time. An implementation of this scheme is somewhat lengthy, so we won't show a complete program. Instead, we'll focus on the concepts and leave the implementation as an exercise.

Character Codes

Each character in a normal uncompressed text file is represented in the computer by one byte (for the venerable ASCII code) or by two bytes (for the newer Unicode, which is designed to work for all languages.) In these schemes, every character requires the same number of bits. Table 8.3 shows how some characters are represented in binary using the ASCII code. As you can see, every character takes 8 bits.

TABLE 8.3 Some ASCII Codes

Character	Decimal	Binary
A	65	01000000
B	66	01000001
C	67	01000010
…	…	…
X	88	01011000
Y	89	01011001
Z	90	01011010

There are several approaches to compressing data. For text, the most common approach is to reduce the number of bits that represent the most-used characters. In English, E is often the most common letter, so it seems reasonable to use as few bits as possible to encode it. On the other hand, Z is seldom used, so using a large number of bits is not so bad.

Suppose we use just two bits for E, say 01. We can't encode every letter of the alphabet in two bits because there are only four 2-bit combinations: 00, 01, 10, and 11. Can we use these four combinations for the four most-used characters? Unfortunately not. We must be careful that no character is represented by the same bit combination that appears at the beginning of a longer code used for some other character. For example, if E is 01, and X is 01011000, then anyone decoding 01011000 wouldn't know if the initial 01 represented an E or the beginning of an X. This leads to a rule: *No code can be the prefix of any other code.*

Something else to consider is that in some messages E might not be the most-used character. If the text is a Java source file, for example, the ; (semicolon) character might appear more often than E. Here's the solution to that problem: For each message, we make up a new code tailored to that particular message. Suppose we want to send the message `SUSIE SAYS IT IS EASY`. The letter S appears a lot, and so does the space character. We might want to make up a table showing how many times each letter appears. This is called a frequency table, as shown in Table 8.4.

TABLE 8.4 Frequency Table

Character	Count
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Space	4
Linefeed	1

The characters with the highest counts should be coded with a small number of bits. Table 8.5 shows how we might encode the characters in the Susie message.

TABLE 8.5 Huffman Code

Character	Code
A	010
E	1111
I	110

TABLE 8.5 Continued

Character	Code
S	10
T	0110
U	01111
Y	1110
Space	00
Linefeed	01110

We use 10 for S and 00 for the space. We can't use 01 or 11 because they are prefixes for other characters. What about 3-bit combinations? There are eight possibilities: 000, 001, 010, 011, 100, 101, 110, and 111. A is 010 and I is 110. Why aren't any other combinations used? We already know we can't use anything starting with 10 or 00; that eliminates four possibilities. Also, 011 is used at the beginning of U and the linefeed, and 111 is used at the beginning of E and Y. Only two 3-bit codes remain, which we use for A and I. In a similar way we can see why only three 4-bit codes are available.

Thus, the entire message is coded as

```
10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 00 110 10 00
➥ 1111 010 10 1110 01110
```

For sanity reasons we show this message broken into the codes for individual characters. Of course, in reality all the bits would run together; there is no space character in a binary message, only 0s and 1s.

Decoding with the Huffman Tree

We'll see later how to create Huffman codes. First, we'll examine the somewhat easier process of decoding. Suppose we received the string of bits shown in the preceding section. How would we transform it back into characters? We can use a kind of binary tree called a *Huffman tree*. Figure 8.23 shows the Huffman tree for the code just discussed.

The characters in the message appear in the tree as leaf nodes. The higher their frequency in the message, the higher up they appear in the tree. The number outside each circle is the frequency. The numbers outside non-leaf nodes are the sums of the frequencies of their children. We'll see later why this is important.

How do we use this tree to decode the message? For each character you start at the root. If you see a 0 bit, you go left to the next node, and if you see a 1 bit, you go right. Try it with the code for A, which is 010. You go left, then right, then left again, and, *mirabile dictu*, you find yourself on the A node. This is shown by the arrows in Figure 8.23.

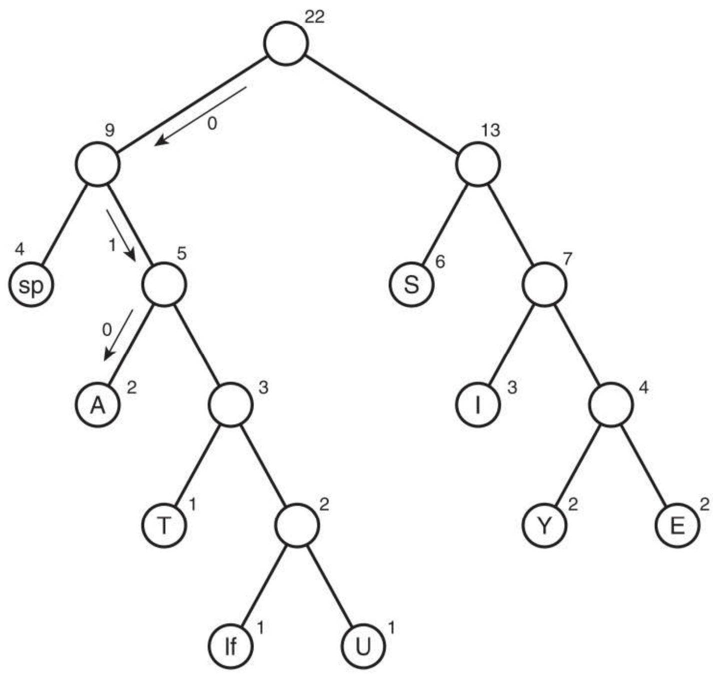

FIGURE 8.23 Huffman tree.

You'll see you can do the same with the other characters. If you have the patience, you can decode the entire bit string this way.

Creating the Huffman Tree

We've seen how to use the Huffman tree for decoding, but how do we create this tree? There are many ways to handle this problem. We'll base our approach on the `Node` and `Tree` classes in the `tree.java` program in Listing 8.1 (although routines that are specific to search trees, like `find()`, `insert()`, and `delete()` are no longer relevant). Here is the algorithm for constructing the tree:

1. Make a `Node` object (as seen in `tree.java`) for each character used in the message. For our Susie example that would be nine nodes. Each node has two data items: the character and that character's frequency in the message. Table 8.4 provides this information for the Susie message.

2. Make a tree object for each of these nodes. The node becomes the root of the tree.

3. Insert these trees in a priority queue (as described in Chapter 4). They are ordered by frequency, with the smallest frequency having the highest priority. That is, when you remove a tree, it's always the one with the least-used character.

Now do the following:

1. Remove two trees from the priority queue, and make them into children of a new node. The new node has a frequency that is the sum of the children's frequencies; its character field can be left blank.
2. Insert this new three-node tree back into the priority queue.
3. Keep repeating steps 1 and 2. The trees will get larger and larger, and there will be fewer and fewer of them. When there is only one tree left in the queue, it is the Huffman tree and you're done.

Figures 8.24 and 8.25 show how the Huffman tree is constructed for the Susie message.

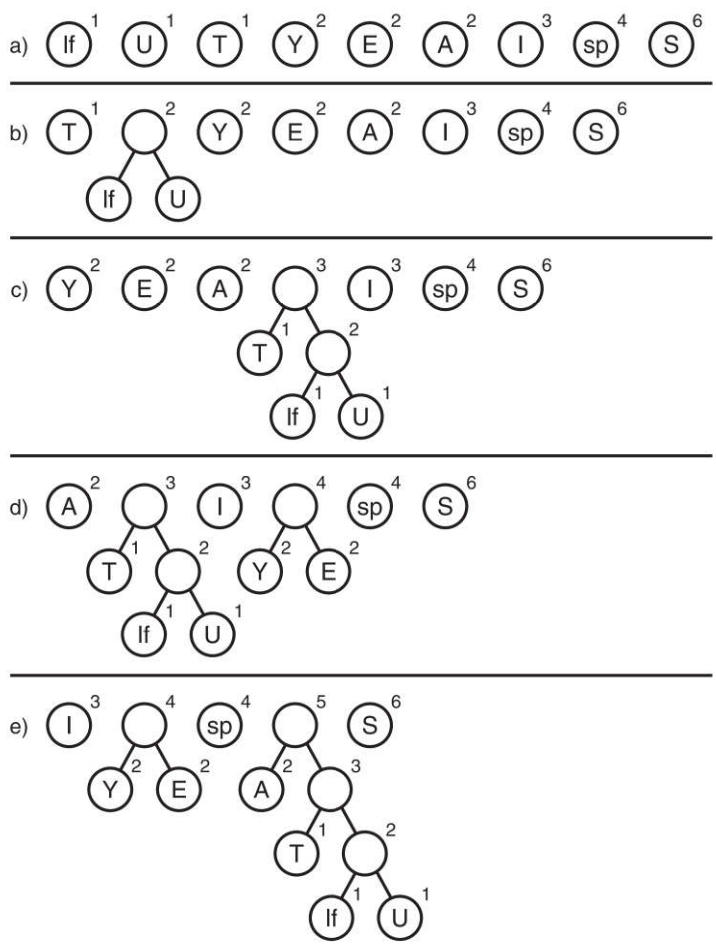

FIGURE 8.24 Growing the Huffman tree, Part 1.

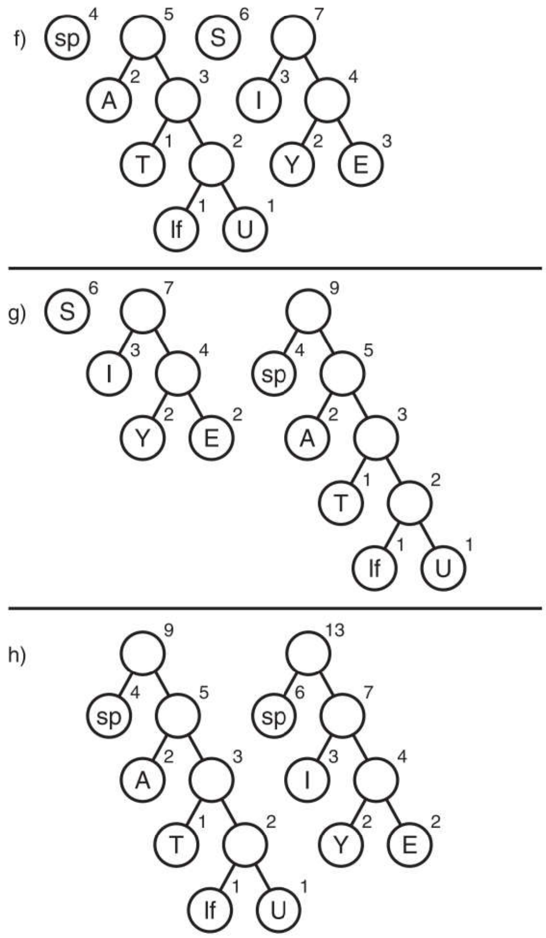

FIGURE 8.25 Growing the Huffman tree, Part 2.

Coding the Message

Now that we have the Huffman tree, how do we code a message? We start by creating a code table, which lists the Huffman code alongside each character. To simplify the discussion, let's assume that, instead of the ASCII code, our computer uses a simplified alphabet that has only uppercase letters with 28 characters. A is 0, B is 1, and so on up to Z, which is 25. A space is 26, and a linefeed is 27. We number these characters so their numerical codes run from 0 to 27. (This is not a compressed code, just a simplification of the ASCII code, the normal way characters are stored in the computer.)

Our code table would be an array of 28 cells. The index of each cell would be the numerical value of the character: 0 for A, 1 for B, and so on. The contents of the cell would be the Huffman code for the corresponding character. Not every cell contains a code; only those that appear in the message. Figure 8.26 shows how this looks for the Susie message.

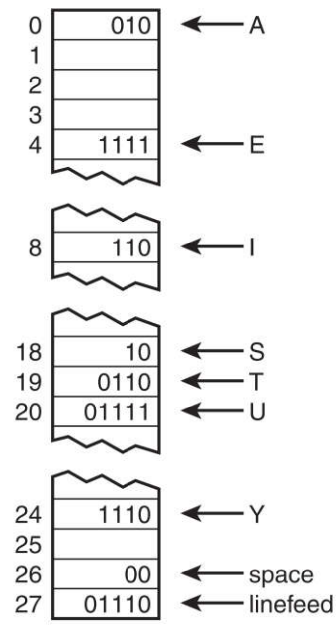

FIGURE 8.26 Code table.

Such a code table makes it easy to generate the coded message: For each character in the original message, we use its code as an index into the code table. We then repeatedly append the Huffman codes to the end of the coded message until it's complete.

Creating the Huffman Code

How do we create the Huffman code to put into the code table? The process is like decoding a message. We start at the root of the Huffman tree and follow every possible path to a leaf node. As we go along the path, we remember the sequence of left and right choices, recording a 0 for a left edge and a 1 for a right edge. When we arrive at the leaf node for a character, the sequence of 0s and 1s is the Huffman code for that character. We put this code into the code table at the appropriate index number.

This process can be handled by calling a method that starts at the root and then calls itself recursively for each child. Eventually, the paths to all the leaf nodes will be explored and the code table will be complete.

Summary

- Trees consist of nodes (circles) connected by edges (lines).
- The root is the topmost node in a tree; it has no parent.
- In a binary tree, a node has at most two children.
- In a binary search tree, all the nodes that are left descendants of node A have key values less than A; all the nodes that are A's right descendants have key values greater than (or equal to) A.
- Trees perform searches, insertions, and deletions in O(log N) time.
- Nodes represent the data objects being stored in the tree.
- Edges are most commonly represented in a program by references to a node's children (and sometimes to its parent).
- Traversing a tree means visiting all its nodes in some order.
- The simplest traversals are preorder, inorder, and postorder.
- An unbalanced tree is one whose root has many more left descendents than right descendants, or vice versa.
- Searching for a node involves comparing the value to be found with the key value of a node, and going to that node's left child if the key search value is less, or to the node's right child if the search value is greater.
- Insertion involves finding the place to insert the new node and then changing a child field in its new parent to refer to it.
- An inorder traversal visits nodes in order of ascending keys.
- Preorder and postorder traversals are useful for parsing algebraic expressions.
- When a node has no children, it can be deleted by setting the child field in its parent to `null`.
- When a node has one child, it can be deleted by setting the child field in its parent to point to its child.
- When a node has two children, it can be deleted by replacing it with its successor.
- The successor to a node A can be found by finding the minimum node in the subtree whose root is A's right child.
- In a deletion of a node with two children, different situations arise, depending on whether the successor is the right child of the node to be deleted or one of the right child's left descendants.

- Nodes with duplicate key values may cause trouble in arrays because only the first one can be found in a search.
- Trees can be represented in the computer's memory as an array, although the reference-based approach is more common.
- A Huffman tree is a binary tree (but not a search tree) used in a data-compression algorithm called Huffman Coding.
- In the Huffman code the characters that appear most frequently are coded with the fewest bits, and those that appear rarely are coded with the most bits.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Insertion and deletion in a tree require what big O time?
2. A binary tree is a search tree if
   a. every non-leaf node has children whose key values are less than (or equal to) the parent.
   b. every left child has a key less than the parent and every right child has a key greater than (or equal to) the parent.
   c. in the path from the root to every leaf node, the key of each node is greater than (or equal to) the key of its parent.
   d. a node can have a maximum of two children.
3. True or False: Not all trees are binary trees.
4. In a complete binary tree with 20 nodes, and the root considered to be at level 0, how many nodes are there at level 4?
5. A subtree of a binary tree always has
   a. a root that is a child of the main tree's root.
   b. a root unconnected to the main tree's root.
   c. fewer nodes than the main tree.
   d. a sibling with the same number of nodes.
6. In the Java code for a tree, the ______ and the _______ are generally separate classes.