Chapter 6

Trees

Chapter Objectives

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the difference between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays
- To learn how to use Java 8 Lambda Expressions and Functional Interfaces to simplify coding
- To learn how to use a binary search tree to store information so that it can be retrieved in an efficient manner
- To learn how to use a Huffman tree to encode characters using fewer bits than ASCII or Unicode, resulting in smaller files and reduced storage requirements

The data organizations you have studied so far are linear in that each element has only one predecessor or successor. Accessing all the elements in sequence is an $\mathbf{O}(n)$ process. In this chapter, we begin our discussion of a data organization that is nonlinear or hierarchical: the tree. Instead of having just one successor, a node in a tree can have multiple successors, but it has just one predecessor. A tree in computer science is like a natural tree, which has a single trunk that may split off into two or more main branches. The predecessor of each main branch is the trunk. Each main branch may spawn several secondary branches (successors of the main branches). The predecessor of each secondary branch is a main branch. In computer science, we draw a tree from the top down, so the root of the tree is at the top of the diagram instead of the bottom.

Because trees have a hierarchical structure, we use them to represent hierarchical organizations of information, such as a class hierarchy, a disk directory and its subdirectories (see Figure 6.1), or a family tree. You will see that trees are recursive data structures because they can be defined recursively. For this reason, many of the methods used to process trees are written as recursive methods.

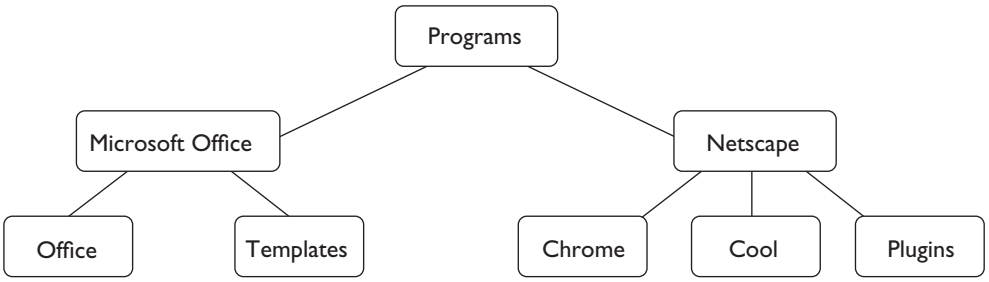


FIGURE 6.1
Part of the Programs Directory

This chapter will focus on a restricted tree structure, a binary tree, in which each element has, at most, two successors. You will learn how to use linked data structures and arrays to represent binary trees. You will also learn how to use a special kind of binary tree called a binary search tree to store information (e.g., the words in a dictionary) in an ordered way. Because each element of a binary tree can have two successors, you will see that searching for an item stored in a binary search tree is much more efficient than searching for an item in a linear data structure: (generally **O**(log *n*) for a binary tree versus **O**(*n*) for a list).

You also will learn about other kinds of binary trees. Expression trees are used to represent arithmetic expressions. The heap is an ordered tree structure that is used as the basis for a very efficient sorting algorithm and for a special kind of queue called the priority queue. The Huffman tree is used for encoding information and compressing files.

Trees

6.1 Tree Terminology and Applications

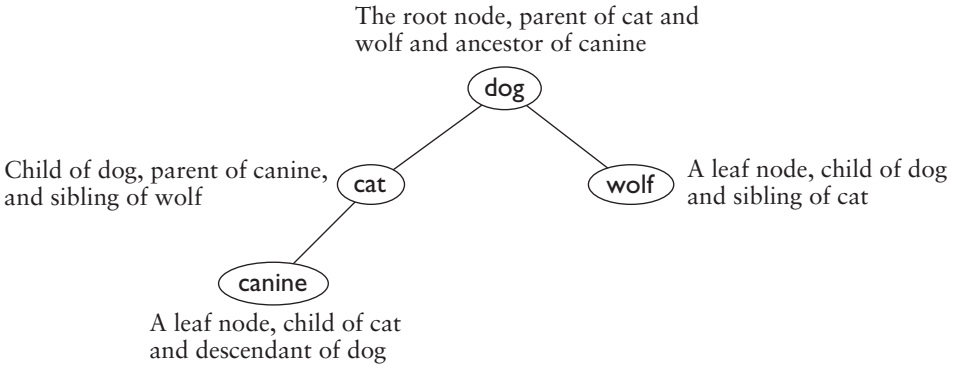
Tree Terminology

We use the same terminology to describe trees in computer science as we do trees in nature. A computer science tree consists of a collection of elements or nodes, with each node linked to its successors. The node at the top of a tree is called its *root* because computer science trees grow from the top down. The links from a node to its successors are called *branches*. The successors of a node are called its *children*. The predecessor of a node is called its *parent*. Each node in a tree has exactly one parent except for the root node, which has no parent. Nodes that have the same parent are *siblings*. A node that has no children is a *leaf node*. Leaf nodes are also known as *external* nodes, and nonleaf nodes are known as *internal* nodes.

A generalization of the parent–child relationship is the *ancestor–descendant relationship*. If node A is the parent of node B, which is the parent of node C, node A is node C's *ancestor,* and node C is node A's *descendant*. Sometimes we say that node A and node C are a grandparent and grandchild, respectively. The root node is an ancestor of every other node in a tree, and every other node in a tree is a descendant of the root node.

Figure 6.2 illustrates these features in a tree that stores a collection of words. The branches are the lines connecting a parent to its children. In discussing this tree, we will refer to a node by the string that it stores. For example, we will refer to the node that stores the string `"dog"` as node *dog*.

FIGURE 6.2
A Tree of Words



A *subtree of a node* is a tree whose root is a child of that node. For example, the nodes *cat* and *canine* and the branch connecting them are a subtree of node *dog*. The other subtree of node *dog* is the tree consisting of the single node *wolf*. The subtree consisting of the single node *canine* is a subtree of node *cat*.

The *level of a node* is a measure of its distance from the root. It is defined recursively as follows:

- If node *n* is the root of tree T, its level is 1.
- If node *n* is not the root of tree T, its level is 1 + the level of its parent.

For the tree in Figure 6.2, node *dog* is at level 1, nodes *cat* and *wolf* are at level 2, and node *canine* is at level 3. Since nodes are below the root, we sometimes use the term *depth* as an alternative term for level. The two have the same meaning.

The *height of a tree* is the number of nodes in the longest path from the root node to a leaf node. The height of the tree in Figure 6.2 is 3 (the longest path goes through the nodes *dog, cat,* and *canine*). Another way of saying this is as follows:

- If T is empty, its height is 0.
- If T is not empty, its height is the maximum depth of its nodes.

An alternate definition of the height of a tree is the number of branches in the longest path from the root node to a leaf node +1.

Binary Trees

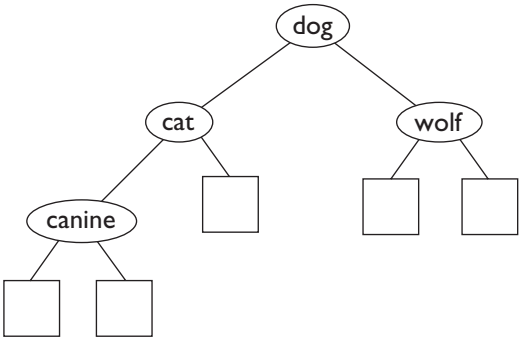
The tree in Figure 6.2 is a *binary tree*. Informally, this is a binary tree because each node has at most two subtrees. A more formal definition for a binary tree follows.

A set of nodes T is a binary tree if either of the following is true:

- T is empty.
- If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary trees.

We refer to T_L as the left subtree and T_R as the right subtree. For the tree in Figure 6.2, the right subtree of node *cat* is empty. The leaf nodes (*wolf* and *canine*) have empty left and right subtrees. This is illustrated in Figure 6.3, where the empty subtrees are indicated by the squares. Generally, the empty subtrees are represented by **`null`** references, but another value may be chosen. From now on, we will consistently use a **`null`** reference and will not draw the squares for the empty subtrees.

FIGURE 6.3
A Tree of Words with Null Subtrees Indicated



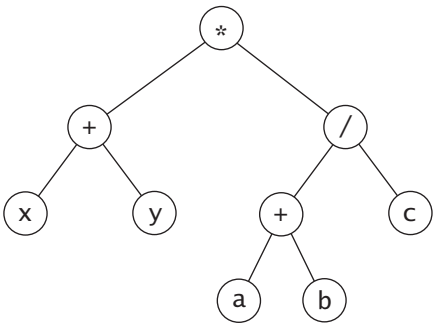
Some Types of Binary Trees

Next, we discuss three different types of binary trees that are common in computer science.

An Expression Tree

Figure 6.4 shows a binary tree that stores an expression. Each node contains an operator (+, -, *, /, %) or an operand. The expression in Figure 6.4 corresponds to `(x + y) * ((a + b) / c)`. Operands are stored in leaf nodes. Parentheses are not stored in the tree because the tree structure dictates the order of operator evaluation. Operators in nodes at higher levels are evaluated after operators in nodes at lower levels, so the operator * in the root node is evaluated last. If a node contains a binary operator, its left subtree represents the operator's left operand and its right subtree represents the operator's right operand. The left subtree of the root represents the expression `x + y`, and the right subtree of the root represents the expression `(a + b) / c`.

FIGURE 6.4
Expression Tree



A Huffman Tree

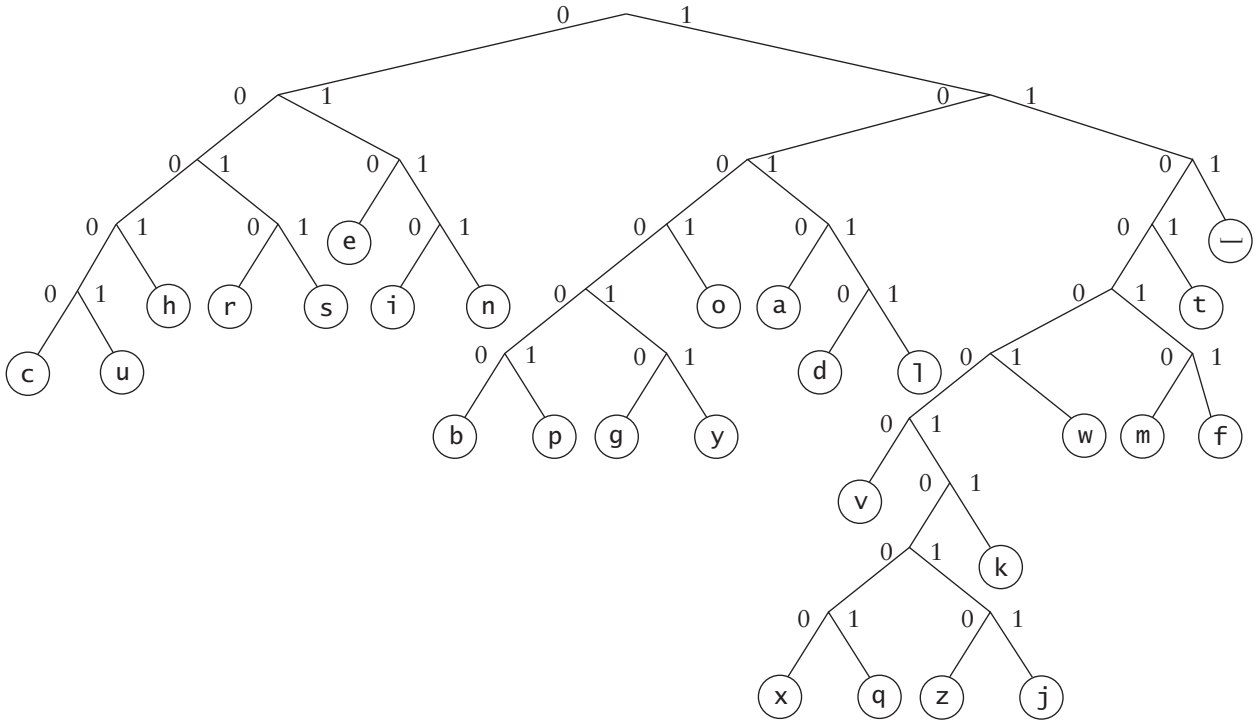
Another use of a binary tree is to represent *Huffman codes* for characters that might appear in a text file. Unlike ASCII or Unicode encoding, which use the same number of bits to encode each character, a Huffman code uses different numbers of bits to encode the letters. It uses fewer bits for the more common letters (e.g., space, *e*, *a*, and *t*) and more bits for the less common letters (e.g., *q*, *x*, and *z*). On average, using Huffman codes to encode text files should give you files with fewer bits than you would get using other codes. Many programs that compress files use Huffman encoding to generate smaller files in order to save disk space or to reduce the time spent sending the files over the Internet.

Figure 6.5 shows the Huffman encoding tree for an alphabet consisting of the lowercase letters and the space character. All the characters are at leaf nodes. The data stored at nonleaf nodes is not shown. To determine the code for a letter, you form a binary string by tracing the path from the root node to that letter. Each time you go left, append a 0, and each time you go right, append a 1. To reach the space character, you go right three times, so the code is 111. The code for the letter *d* is 10110 (right, left, right, right, left).

The two characters shown at level 4 of the tree (space, *e*) are the most common and, therefore, have the shortest codes (111, 010). The next most common characters (*a, o, i,* etc.) are at level 5 of the tree.

You can store the code for each letter in an array. For example, the code for the space `' '` would be at position 0, the letter `'a'` would be at position 1, and the code for letter `'z'` would be at position 26. You can *encode* each letter in a file by looking up its code in the array.

FIGURE 6.5
Huffman Code Tree



However, to *decode* a file of letters and spaces, you walk down the Huffman tree, starting at the root, until you reach a letter and then append that letter to the output text. Once you have reached a letter, go back to the root. Here is an example. The substrings that represent the individual letters are shown in alternate shades of black to help you follow the process. The underscore in the second line represents a space character (code is 111).

```
10001001001111010101010001010111010011
  g   o _ e a g   l e s
```

Huffman trees are discussed further in Section 6.7.

A Binary Search Tree

The tree in Figure 6.2 is a *binary search tree* because, for each node, all words in its left subtree precede the word in that node, and all words in its right subtree follow the word in that node. For example, for the root node *dog*, all words in its left subtree (*cat, canine*) precede dog in the dictionary, and all words in its right subtree (*wolf*) follow *dog*. Similarly, for the node *cat,* the word in its left subtree (*canine*) precedes it. There are no duplicate entries in a binary search tree.

More formally, we define a binary search tree as follows:

A set of nodes T is a binary search tree if either of the following is true:

- T is empty.
- If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of T is greater than all values in T_L and is less than all values in T_R.

The order relations in a binary search tree expedite searching the tree. A recursive algorithm for searching a binary search tree follows:

1. `if` the tree is empty
2. Return `null` *(target is not found)*.

   `else if` the target matches the root node's data
3. Return the data stored at the root node.

   `else if` the target is less than the root node's data
4. Return the result of searching the left subtree of the root.

   `else`
5. Return the result of searching the right subtree of the root.

The first two cases are base cases and self-explanatory. In the first recursive case, if the target is less than the root node's data, we search only the left subtree (T_L) because all data items in T_R are larger than the root node's data and, therefore, larger than the target. Likewise, we execute the second recursive step (search the right subtree) if the target is greater than the root node's data.

Just as with a binary search of an array, each probe into the binary search tree has the potential of eliminating half the elements in the tree. If the binary search tree is relatively balanced (i.e., the depths of the leaves are approximately the same), searching a binary search tree is an **O**(log *n*) process, just like a binary search of an ordered array.

What is the advantage of using a binary search tree instead of just storing elements in an array and then sorting it? A binary search tree never has to be sorted because its elements always satisfy the required order relations. When new elements are inserted (or removed), the binary search tree property can be maintained. In contrast, an array must be expanded whenever new elements are added, and it must be compacted whenever elements are removed. Both expanding and contracting involve shifting items and are thus **O**(*n*) operations.

Full, Perfect, and Complete Binary Trees

The tree on the left in Figure. 6.6 is called a *full binary tree* because all nodes have either 2 children or 0 children (the leaf nodes). The tree in the middle is a *perfect binary tree,* which is defined as a full binary tree of height n(n is 3) with exactly $2^n - 1$ (7) nodes. The tree on the right is a *complete binary tree,* which is a perfect binary tree through level $n - 1$ with some extra leaf nodes at level n (the tree height), all toward the left.

General Trees

A general tree is a tree that does not have the restriction that each node of a tree has at most two subtrees. So nodes in a general tree can have any number of subtrees. Figure 6.7 shows a general tree that represents a family tree showing the descendants of King William I (the Conqueror) of England.

FIGURE 6.6
Full, Perfect, and Complete Binary Trees

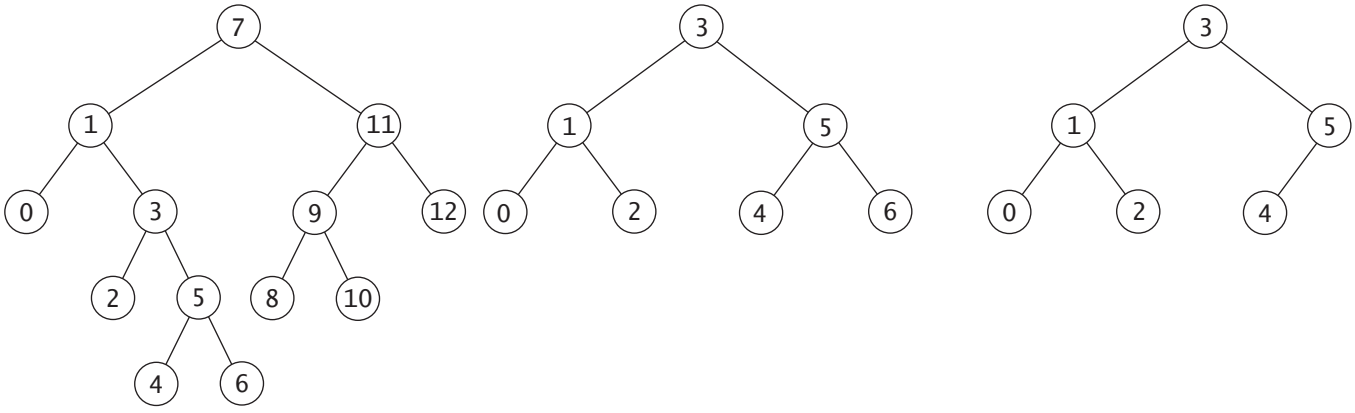
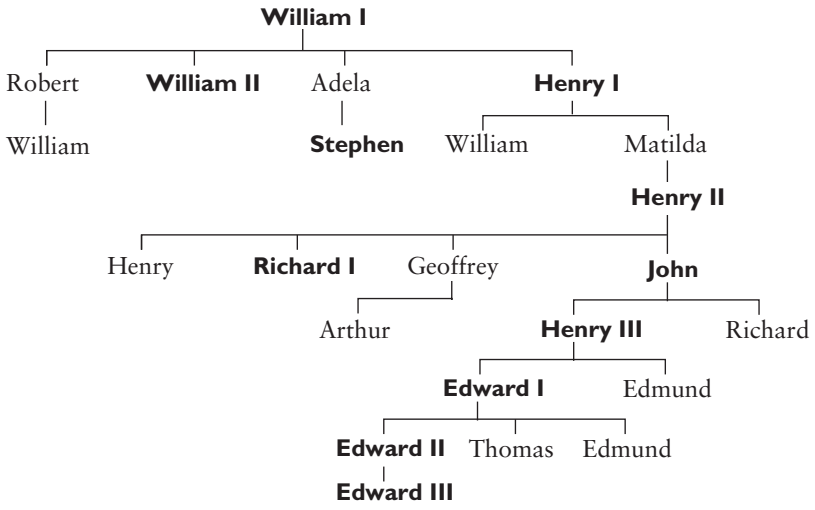


FIGURE 6.7
Family Tree for the Descendants of William I of England



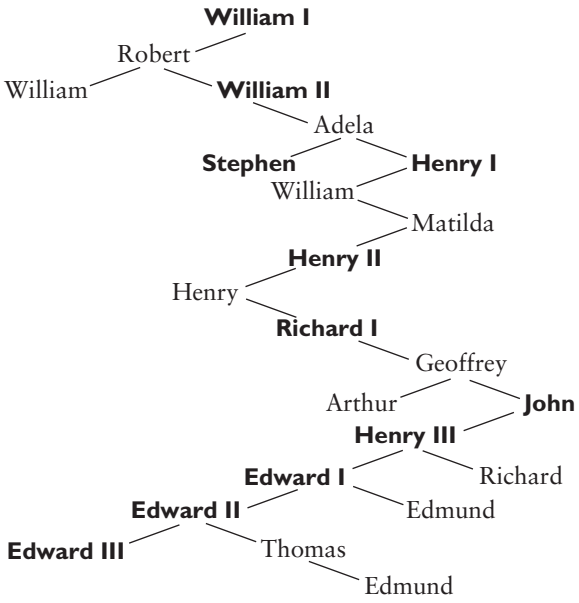


FIGURE 6.8
Binary Tree Equivalent of King William's Family Tree

We will not discuss general trees in this chapter. However, it is worth mentioning that a general tree can be represented using a binary tree. Figure 6.8 shows a binary tree representation of the family tree in Figure 6.7. We obtained it by connecting the left branch from a node to the oldest child (if any). Each right branch from a node is connected to the next younger sibling (if any).

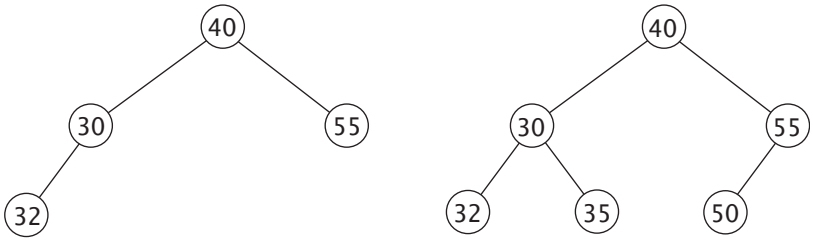
The names of the men who became kings are in boldface type. You would expect the eldest son to succeed his father as king; however, this would not be the case if the eldest male died before his father. For example, Robert died before William I, so William II became king instead. Starting with King John (near the bottom of the tree), the eldest son of each king did become the King of England.

EXERCISES FOR SECTION 6.1

SELF-CHECK

1. Draw binary expression trees for the following infix expressions. Your trees should enforce the Java rules for operator evaluation (higher-precedence operators before lower-precedence operators and left associativity).
   a. `x / y + a - b * c`
   b. `(x * a) - y / b * (c + d)`
   c. `(x + (a * (b - c))/d`
2. Using the Huffman tree in Figure 6.5,
   a. Write the binary string for the message "scissors cuts paper".
   b. Decode the following binary string using the tree in Figure 6.5:
   `11000100010100010010111011000111111100011010101111101101001`

3. For each tree shown below, answer these questions. What is its height? Is it a full tree? Is it a complete tree? Is it a binary search tree? If not, make it a binary search tree.



4. For the binary trees in Figures 6.2–6.5, indicate whether each tree is full, perfect, complete, or none of the above.

5. Represent the general tree in Figure 6.1 as a binary tree.

6.2 Tree Traversals

Often we want to determine the nodes of a tree and their relationship. We can do this by walking through the tree in a prescribed order and visiting the nodes (processing the information in the nodes) as they are encountered. This process is known as *tree traversal*. We will discuss three kinds of traversal in this section: inorder, preorder, and postorder. These three methods are characterized by when they visit a node in relation to the nodes in its subtrees (T_L and T_R).

- Preorder: Visit root node, traverse T_L, and traverse T_R.
- Inorder: Traverse T_L, visit root node, and traverse T_R.
- Postorder Traverse T_L, traverse T_R, and visit root node.

Because trees are recursive data structures, we can write similar recursive algorithms for all three techniques. The difference in the algorithms is whether the root is visited before the children are traversed (pre), in between traversing the left and right children (in), or after the children are traversed (post).

Algorithm for Preorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.

Algorithm for Inorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

Algorithm for Postorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

Visualizing Tree Traversals

You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree. If the mouse always keeps the tree to the left (from the mouse's point of view), it will trace the route shown in gray around the tree shown in Figure 6.9. This is known as an *Euler tour*.

If we record each node as the mouse first encounters it (indicated by the arrows pointing down in Figure 6.9), we get the following sequence:

a b d g e h c f i j

This is a preorder traversal because the mouse visits each node before traversing its subtrees. The mouse also walks down the left branch (if it exists) of each node before going down the right branch, so the mouse visits a node, traverses its left subtree, and traverses its right subtree.

If we record each node as the mouse returns from traversing its left subtree (indicated by the arrows pointing to the right in Figure 6.9), we get the following sequence:

d g b h e a i f j c

This is an inorder traversal. The mouse traverses the left subtree, visits the root, and then traverses the right subtree. Node *d* is visited first because it has no left subtree.

If we record each node as the mouse last encounters it (indicated by the arrows pointing up in Figure 6.9), we get the following sequence:

g d h e b i j f c a

This is a postorder traversal because we visit the node after traversing both its subtrees. The mouse traverses the left subtree, traverses the right subtree, and then visits the node.

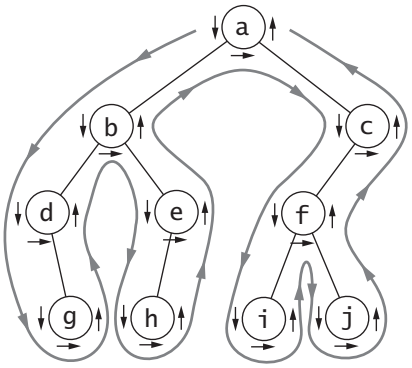


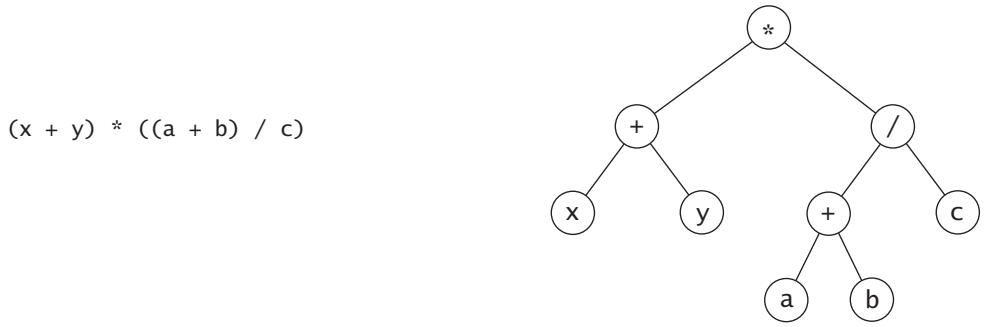
FIGURE 6.9
Traversal of a Binary Tree

Traversals of Binary Search Trees and Expression Trees

An inorder traversal of a binary search tree results in the nodes being visited in sequence by increasing data value. For example, for the binary search tree shown earlier in Figure 6.2, the inorder traversal would visit the nodes in the sequence:

canine, cat, dog, wolf

Traversals of expression trees give interesting results. If we perform an inorder traversal of the expression tree first shown in Figure 6.4 and repeated here, we visit the nodes in the sequence `x + y* a + b / c`. If we insert parentheses where they belong, we get the infix expression



The postorder traversal of this tree would visit the nodes in the sequence

x y + a b + c / *

which is the postfix form of the expression. To illustrate this, we show the *operand–operand–operator* groupings under the expression.

The preorder traversal visits the nodes in the sequence

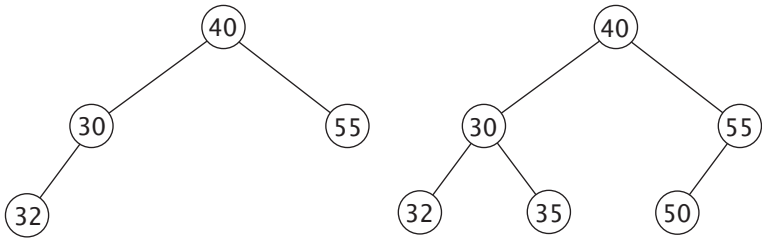
* + x y / + a b c

which is the prefix form of the expression. To illustrate this, we show the *operator–operand–operand* groupings under the expression.

EXERCISES FOR SECTION 6.2

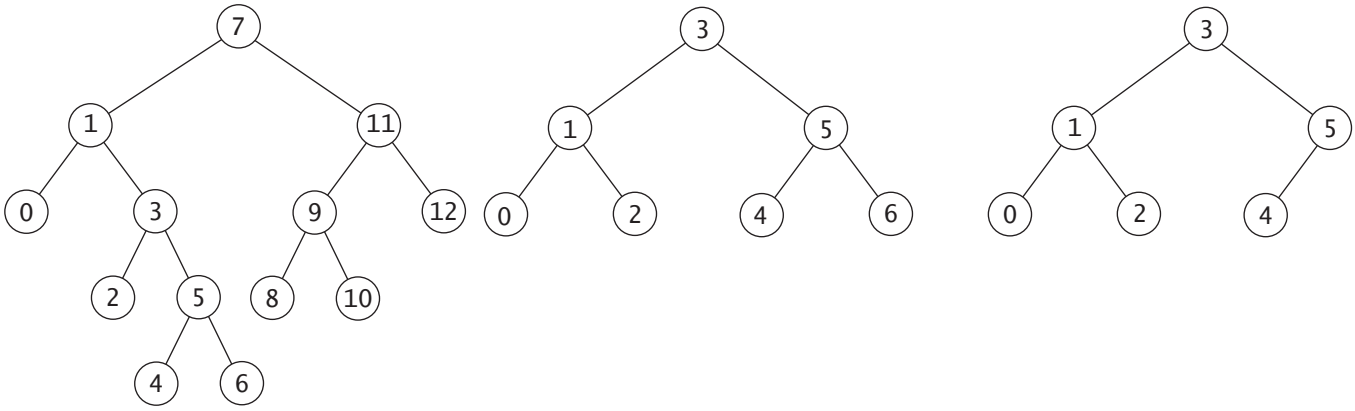
SELF-CHECK

1. For the following trees:



If visiting a node displays the integer value stored, show the inorder, preorder, and postorder traversal of each tree.

2. Repeat Exercise 1 above for the trees in Figure 6.6, redrawn below.



3. Draw an expression tree corresponding to each of the following:
   a. Inorder traversal is `x / y + 3 * b / c` (Your tree should represent the Java meaning of the expression.)
   b. Postorder traversal is `x y z + a b - c * / -`
   c. Preorder traversal is `* + a - x y / c d`
4. Explain why the statement "Your tree should represent the Java meaning of the expression" was not needed for parts b and c of Exercise 3 above.

6.3 Implementing a `BinaryTree` Class

In this section, we show how to use linked data structures to represent binary trees and binary tree nodes. We begin by focusing on the structure of a binary tree node.

The `Node<E>` Class

Just as for a linked list, a node consists of a data part and links (references) to successor nodes. So that we can store any kind of data in a tree node, we will make the data part a reference of type `E`. Instead of having a single link (reference) to a successor node as in a list, a binary tree node must have links (references) to both its left and right subtrees. Figure 6.10 shows the structure of a binary tree node; Listing 6.1 shows its implementation.

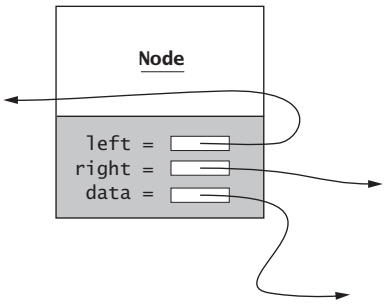


FIGURE 6.10
Linked Structure to Represent a Node

LISTING 6.1
Nested Class `Node`

```
/** Class to encapsulate a tree node. */
protected static class Node<E> implements Serializable {
    // Data Fields
    /** The information stored in this node. */
    protected E data;
    /** Reference to the left child. */
    protected Node<E> left;
    /** Reference to the right child. */
    protected Node<E> right;

    // Constructors
    /** Construct a node with given data and no children.
        @param data The data to store in this node
     */
    public Node(E data) {
        this.data = data;
        left = null;
        right = null;
    }
```