IN THIS CHAPTER, YOU WILL:

- Learn about binary trees
- Explore various binary tree traversal algorithms
- Learn how to organize data in a binary search tree
- Learn how to insert and delete items in a binary search tree
- Explore nonrecursive binary tree traversal algorithms
When data is being organized, a programmer’s highest priority is to organize it in such a way that item insertion, deletion, and lookups (searches) are fast. You have already seen how to store and process data in an array. Because an array is a random-access data structure, if the data is properly organized (say, sorted), then we can use a search algorithm, such as a binary search, to effectively find and retrieve an item from the list. However, we know that storing data in an array has its limitations. For example, item insertion (especially if the array is sorted) and item deletion can be very time consuming, especially if the list size is very large, because each of these operations requires data movement. To speed up item insertion and deletion, we used linked lists. Item insertion and deletion in a linked list do not require any data movement; we simply adjust some of the links in the list. However, one of the drawbacks of linked lists is that they must be processed sequentially. That is, to insert or delete an item, or simply to search the list for a particular item, we must begin our search at the first node in the list. As you know, a sequential search is good only for very small lists because the average search length of a sequential search is half the size of the list.

Binary Trees

This chapter discusses how to organize data dynamically so that item insertion, deletion, and lookups are more efficient.

We first introduce some definitions to facilitate our discussion.

**Definition:** A binary tree, $T$, is either empty or such that:

i. $T$ has a special node called the **root** node;

ii. $T$ has two sets of nodes, $L_T$ and $R_T$, called the **left subtree** and **right subtree** of $T$, respectively; and

iii. $L_T$ and $R_T$ are binary trees.

Suppose that $T$ is a binary tree with the root node $A$. Let $L_A$ denote the left subtree of $A$ and $R_A$ denote the right subtree of $A$. Now $L_A$ and $R_A$ are binary trees. Suppose that $B$ is the root node of $L_A$ and $C$ is the root node of $R_A$. $B$ is called the **left child** of $A$; $C$ is called the **right child** of $A$. Moreover, $A$ is called the **parent** of $B$ and $C$.

A binary tree can be shown pictorially. In the diagram of a binary tree, each node of the binary tree is represented as a circle, and the circle is labeled by the node. The root node of the binary tree is drawn at the top. The left child of the root node (if any) is drawn below and to the left of the root node. Similarly, the right child of the root node (if any) is drawn below and to the right of the root node. Children are connected to the parent by an arrow from the parent to the child. An arrow is usually called a **directed edge** or a **directed branch** (or simply a **branch**) (see Figure 19-1). Because the root node, $B$, of $L_A$ is already drawn, we apply the same (recursive) procedure to draw the remaining parts of $L_A$. $R_A$ is drawn similarly. If a node has no left child, for example, we draw an arrow from the node to the left, ending with three stacked lines. That is, three lines at the end of an arrow indicate that the subtree is empty.
In Figure 19-1, the root node of this binary tree is A. The left subtree of the root node, which we denote by $L_A$, is the set $L_A = \{B, D, E, G\}$, and the right subtree of the root node, which we denote by $R_A$, is the set $R_A = \{C, F, H\}$. The root node of the left subtree of $A$—that is, the root node of $L_A$—is node $B$. The root node of $R_A$ is $C$, and so on. Clearly, $L_A$ and $R_A$ are binary trees. Because three lines at the end of an arrow mean that the subtree is empty, it follows that the left subtree of $D$ is empty. Also, note that for node $F$, the left child is $H$ and node $F$ has no right child.

Example 19-1 shows nonempty binary trees.

**EXAMPLE 19-1**

Figure 19-2 shows binary trees with one, two, or three nodes.

In the binary tree of Figure 19-2(a), the root node is $A$, $L_A =$ empty, and $R_A =$ empty.
In the binary tree of Figure 19-2(b), the root node is $A$, $L_A = \{B\}$, and $R_A = \text{empty}$. The root node of $L_A = B$, $L_B = \text{empty}$, and $R_B = \text{empty}$.

In the binary tree of Figure 19-2(c), the root node is $A$, $L_A = \text{empty}$, and $R_A = \{C\}$. The root node of $R_A = C$, $L_C = \text{empty}$, and $R_C = \text{empty}$.

In the binary tree of Figure 19-2(d), the root node is $A$, $L_A = \{B\}$, and $R_A = \{C\}$. The root node of $L_A = B$, $L_B = \text{empty}$, and $R_B = \text{empty}$. The root node of $R_A = C$, $L_C = \text{empty}$, and $R_C = \text{empty}$.

**EXAMPLE 19-2**

Figure 19-3 shows other cases of nonempty binary trees with three nodes.

As you can see from the preceding examples, every node in a binary tree has, at most, two children. Thus, every node, other than storing its own information, must keep track of its left subtree and right subtree. This implies that every node has two pointers, say, $l\text{link}$ and $r\text{link}$. The pointer $l\text{link}$ points to the root node of the left subtree of the node; the pointer $r\text{link}$ points to the root node of the right subtree of the node.

The following *struct* defines the node of a binary tree:

```cpp
template <class elemType>
struct nodeType
{
    elemType info;
    nodeType<elemType> *lLink;
    nodeType<elemType> *rLink;
};
```
From the definition of the node, it is clear that for each node:

1. The data is stored in `info`.
2. A pointer to the left child is stored in `lLink`.
3. A pointer to the right child is stored in `rLink`.

Furthermore, a pointer to the root node of the binary tree is stored outside of the binary tree in a pointer variable, usually called the `root`, of type `nodeType`. Thus, in general, a binary tree looks like the diagram in Figure 19-4.

For simplicity, we will continue to draw binary trees as before. That is, we will use circles to represent nodes and left and right arrows to represent links. As before, three lines at the end of an arrow mean that the subtree is empty.

Before we leave this section, let us define a few terms.

A node in a binary tree is called a **leaf** if it has no left and right children. Let $U$ and $V$ be two nodes in the binary tree $T$. $U$ is called the **parent** of $V$ if there is a branch from $U$ to $V$. A **path** from a node $X$ to a node $Y$ in a binary tree is a sequence of nodes $X_0, X_1, \ldots, X_n$ such that:

i. $X = X_0, X_n = Y$

ii. $X_{i-1}$ is the parent of $X_i$ for all $i = 1, 2, \ldots, n$. That is, there is a branch from $X_0$ to $X_1$, $X_1$ to $X_2$, \ldots, $X_{i-1}$ to $X_i$, \ldots, $X_{n-1}$ to $X_n$.

If $X_0, X_1, \ldots, X_n$ is a path from node $X$ to node $Y$, sometimes we denote it by $X = X_0 - X_1 - \cdots - X_{n-1} - X_n = Y$ or simply $X - X_1 - \cdots - X_{n-1} - Y$.

Because the branches only go from a parent to its children, from the previous discussion it is clear that in a binary tree, there is a unique path from the root to every node in the binary tree.

**Definition:** The length of a path in a binary tree is the number of branches on that path.
**Definition**: The level of a node in a binary tree is the number of branches on the path from the root to the node.

Clearly, the level of the root node of a binary tree is 0, and the level of the children of the root node is 1.

**Definition**: The height of a binary tree is the number of nodes on the longest path from the root to a leaf.

---

**EXAMPLE 19-3**

Consider the binary tree of Figure 19-5. In this example, the terms such as node A and (node with info A) mean the same thing.

![Binary Tree Diagram]

FIGURE 19-5 Binary tree

In this binary tree, the nodes I, E, and H have no left and right children. So, the nodes I, E, and H are leaves.

There is a branch from node A to node B. So, node A is the parent of node B. Similarly, node A is the parent of node C, node B is the parent of nodes D and E, node C is the parent of node F, node D is the parent of node G, and so on.

A–B–D–G is a path from node A to node G. Because there are three branches on this path, the length of this path is 3. Similarly, B–D–G–I is a path from node B to node I.

There are three leaves in this binary tree, which are I, E, and H. Also, the paths from root to these leaves are: A–B–D–G–I, A–B–E, and A–C–F–H. Clearly, the longest path from
root to a leaf is A–B–D–G–I. The number of nodes on this path is 5. Hence, the height of
the binary tree is 5.

Suppose that a pointer, \( p \), to the root node of a binary tree is given. We next describe a
C++ function, \texttt{height}, to find the height of the binary tree. The pointer to the root
node is passed as a parameter to the function \texttt{height}.

If the binary tree is empty, then the height is 0. Suppose that the binary tree is nonempty. To find
the height of the binary tree, we first find the height of the left subtree and the height of the right
subtree. We then take the maximum of these two heights and add 1 to find the height of the
binary tree. To find the height of the left (right) subtree, we apply the same procedure because the
left (right) subtree is a binary tree. Therefore, the general algorithm to find the height of a binary
tree is as follows. Suppose \( \texttt{height}(p) \) denotes the height of the binary tree with root \( p \).

\[
\begin{align*}
\text{if} \ (p \ \text{is NULL}) & \quad \text{height}(p) = 0 \\
\text{else} & \quad \text{height}(p) = 1 + \max(\text{height}(p->lLink), \text{height}(p->rLink))
\end{align*}
\]

Clearly, this is a recursive algorithm. The following function implements this algorithm:

\[
\begin{align*}
\text{template} \ <\text{class elemType}> & \quad \text{int} \ \text{height}(\text{nodeType<elemType> } *p) \\
& \quad \{
\quad \text{if} \ (p == \text{NULL}) & \quad \text{return} \ 0;
\quad \text{else} & \quad \text{return} \ 1 + \max(\text{height}(p->lLink), \text{height}(p->rLink));
\quad \}
\end{align*}
\]

The definition of the function \texttt{height} uses the function \texttt{max} to determine the larger of
two integers. The function \texttt{max} can be easily implemented.

Similarly, we can implement algorithms to find the number of nodes and number of
leaves in a binary tree.

\section*{Copy Tree}

One useful operation on binary trees is to make an identical copy of a binary tree. A
binary tree is a dynamic data structure; that is, memory for the nodes of a binary tree is
allocated and deallocated during program execution. Therefore, if we use just the value of
the pointer of the root node to make a copy of a binary tree, we get a shallow copy of the
data. To make an identical copy of a binary tree, we need to create as many nodes as there
are in the binary tree to be copied. Moreover, in the copied tree, these nodes must appear
in the same order as they are in the original binary tree.

Given a pointer to the root node of a binary tree, we next describe a function that makes
a copy of a given binary tree. This function is also quite useful in implementing the copy
constructor and overloading the assignment operator, as described later in this chapter
(see “Implementing Binary Trees”).
template <class elemType>
void copyTree(nodeType<elemType>* &copiedTreeRoot, 
    nodeType<elemType>* otherTreeRoot)
{
    if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL;
    else
    {
        copiedTreeRoot = new nodeType<elemType>;
        copiedTreeRoot->info = otherTreeRoot->info;
        copyTree(copiedTreeRoot->lLink, otherTreeRoot->lLink);
        copyTree(copiedTreeRoot->rLink, otherTreeRoot->rLink);
    }
} //end copyTree

We will use the function copyTree when we overload the assignment operator and implement the copy constructor.

**Binary Tree Traversal**

The item insertion, deletion, and lookup operations require that the binary tree be traversed. Thus, the most common operation performed on a binary tree is to traverse the binary tree, or visit each node of the binary tree. As you can see from the diagram of a binary tree, the traversal must start at the root node because there is a pointer to the root node of the binary tree. For each node, we have two choices:

- Visit the node first.
- Visit the subtrees first.

These choices lead to three commonly used traversals of a binary tree:

- Inorder traversal
- Preorder traversal
- Postorder traversal

**INORDER TRAVERSAL**

In an inorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.
2. Visit the node.
3. Traverse the right subtree.

**PREORDER TRAVERSAL**

In a preorder traversal, the binary tree is traversed as follows:

1. Visit the node.
2. Traverse the left subtree.
3. Traverse the right subtree.
**POSTORDER TRAVERSAL**

In a postorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the node.

Clearly, each of these traversal algorithms is recursive.

The listing of the nodes produced by the inorder traversal of a binary tree is called the **inorder sequence**. The listing of the nodes produced by the preorder traversal is called the **preorder sequence**, and the listing of the nodes produced by the postorder traversal is called the **postorder sequence**.

**EXAMPLE 19-4**

Consider the binary tree in Figure 19-6. Let $T$ be a binary tree. Suppose that $T$ is nonempty and the root node of $T$ is $A$. Then inorder($T$) or inorder($A$) denotes the listing of nodes of $T$ in the inorder sequence and root($T$) denotes the root node of $T$. For simplicity, we assume that visiting a node means to output the data stored in the node. In the section “Binary Tree Traversal and Functions as Parameters,” we will explain how to modify the binary tree traversal algorithms so that by using a function, the user can specify the action to be performed on a node when the node is visited.

Let $T$ denote the binary tree in Figure 19-6. Then, root($T$) = $A$. Therefore, we start the traversal at $A$. That is, determine inorder($A$). Because the binary tree is nonempty, to determine inorder($A$), we do the following:
1. Determine inorder($L_A$), where $L_A$ is the left subtree of $A$. Note that $L_A = \{B, D, F\}$.
2. Visit $A$.
3. Determine inorder($R_A$), where $R_A$ is the right subtree of $A$. Note that $R_A = \{C, E, G\}$.

Now we cannot do Step 2 until we have finished Step 1.

1. inorder($L_A$): Now $L_A$ is a binary tree, and root($L_A$) = $B$. So to determine inorder($L_A$), we do the following:
   1.1. Determine inorder($L_B$), where $L_B = \{D, F\}$.
   1.2. Visit $B$.
   1.3. Determine inorder($R_B$), where $R_B$ = empty.

As before, first we complete Step 1.1 before proceeding to Step 1.2.

1.1. inorder($L_B$): Now $L_B$ is a binary tree, and root($L_B$) = $D$. So to determine inorder($L_B$), we do the following:
   1.1.1. Determine inorder($L_D$), where $L_D = \emptyset$.
   1.1.2. Visit $D$.
   1.1.3. Determine inorder($R_D$), where $R_D = \{F\}$. Because $L_D = \emptyset$, Step 1.1.1 is completed, so we proceed to Step 1.1.2, which outputs $D$. Because Step 1.1.2 is also completed, we proceed to Step 1.1.3.

1.1.3. Determine inorder($R_D$), where $R_D = \{F\}$. Now $R_D$ is a binary tree, and root($R_D$) = $F$. So to determine inorder($R_D$), we do the following:
   1.1.3.1. Determine inorder($L_F$), where $L_F = \emptyset$.
   1.1.3.2. Visit $F$.
   1.1.3.3. Determine inorder($R_F$), where $R_F = \emptyset$. Because $L_F = \emptyset$, Step 1.1.3.1 is completed, so we proceed to Step 1.1.3.2, which outputs $F$. Because Step 1.1.3.2 is also completed, we proceed to Step 1.1.3.3. Because $R_F = \emptyset$, this step is also completed. Thus, Step 1.1.3 is completed, which in turn completes Step 1.

Next, we proceed to Step 1.2, which outputs $B$. After completing Step 1.2, we proceed to Step 1.3. Now Step 1.3 requires us to determine inorder($R_B$). However, $R_B$ = empty, so Step 1.3 is completed, which in turn completes Step 1.

2. Next, we proceed to Step 2, which outputs $A$. At this point we have completed inorder($L_A$) and visited $A$. 

1314  |  Chapter 19: Binary Trees
3. Now, we proceed to Step 3, that is, determine inorder($R_A$), where $R_A = \{C, E, G\}$. Now $R_A$ is a nonempty binary tree and root($R_A$) = $C$, so to determine inorder($R_A$), we need to do the following:

3.1. Determine inorder($L_C$), where $L_C = \emptyset$.
3.2. Visit $C$.
3.3. Determine inorder($R_C$), where $R_C = \{E, G\}$.

Now $L_C = \emptyset$, so Step 3.1 is completed. Next, in Step 3.2, we output $C$, which completes this step. After completing Step 3.2, we proceed to Step 3.3.

3.3. Determine inorder($R_C$), where $R_C = \{E, G\}$. Now $R_C$ is a nonempty binary tree with root($R_C$) = $E$. Thus, inorder($R_C$) requires us to complete the following steps:

3.3.1. Determine inorder($L_E$), where $L_E = \{G\}$.
3.3.2. Visit $E$.
3.3.3. Determine inorder($R_E$), where $R_E = \emptyset$.

To complete Step 3.3.1, we must determine inorder($L_E$), where $L_E = \{G\}$. Now $L_E$ is a binary tree with root($L_E$) = $\{G\}$. Therefore, to determine inorder($L_E$), we must complete the following steps:

3.3.1.1. Determine inorder($L_G$), where $L_G = \emptyset$.
3.3.1.2. Visit $G$.
3.3.1.3. Determine inorder($R_G$), where $R_G = \emptyset$.

Now $L_G = \emptyset$, so Step 3.3.1.1 is completed. Next Step 3.3.1.2 outputs $G$, which completes this step. Because $R_G = \emptyset$, Step 3.3.1.3 is also completed. This in turn completes Step 3.3.1.

After completing Step 3.3.1, to complete Step 3.3.2, we output $E$. Next because $R_E = \emptyset$, Step 3.3.3 is also completed, which in turn completes Step 3.3.

Because Step 3.3 is completed, Step 3 is also completed, that is we have determined inorder($R_A$). It now follows that:

inorder($A$) = $DFBACGE$

In a similar manner, the preorder and postorder traversals output the nodes in the following order:

preorder($A$) = $ABDFCEF$
postorder($A$) = $FDBGEC$A

As you can see from the walk-through of the inorder traversal, after visiting the left subtree of a node, we must come back to the node itself. The links are only in one direction; that is, the parent node points to the left and right children, but there is no pointer from each child to the parent. Therefore, before going to a child, we must somehow save a pointer to the
parent node. A convenient way to do this is to write a recursive inorder function because in
a recursive call, after completing a particular call, the control goes back to the caller. (Later,
we will discuss how to write nonrecursive traversal functions.) The recursive definition of
the function to implement the inorder traversal algorithms is:

```cpp
template <class elemType>
void inorder(nodeType<elemType> *p) const
{
    if (p != NULL)
    {
        inorder(p->lLink);
        cout << p->info << " ";
        inorder(p->rLink);
    }
}
```

To do the inorder traversal of a binary tree, the root node of the binary tree is passed as a
parameter to the function `inorder`. For example, if `root` points to the root node of the
binary tree, a call to the function `inorder` is:

```cpp
inorder(root);
```

Similarly, we can write the functions to implement the preorder and postorder traversals.
The definitions of these functions are given next.

```cpp
template <class elemType>
void preorder(nodeType<elemType> *p) const
{
    if (p != NULL)
    {
        cout << p->info << " ";
        preorder(p->lLink);
        preorder(p->rLink);
    }
}
```

```cpp
template <class elemType>
void postorder(nodeType<elemType> *p) const
{
    if (p != NULL)
    {
        postorder(p->lLink);
        postorder(p->rLink);
        cout << p->info << " ";
    }
}
```

This section described the binary tree traversal algorithms inorder, preorder, and post-
order. If you want to make a copy of a binary tree while preserving the structure of the
binary tree, you can use preorder traversal. To delete all of the nodes of a binary tree, you
can use the postorder traversal. Later in this chapter, we will discuss binary search trees.
The inorder traversal of a binary search tree visits the nodes in sorted order.
In addition to the inorder, preorder, and postorder traversals, a binary tree can also be traversed **level-by-level**, also known as **breadth-first traversal**. In Chapter 20, we discuss graphs. A binary tree is also a graph. We discuss how to implement breadth-first traversal algorithms for graphs. You can modify that algorithm to do a breadth-first traversal of binary trees.

### Implementing Binary Trees

The preceding sections described various operations that can be performed on a binary tree, as well as the functions to implement these operations. This section describes binary trees as an abstract data type (ADT). Before designing the class to implement a binary tree as an ADT, let us list the various operations that are typically performed on a binary tree.

1. Determine whether the binary tree is empty.
2. Search the binary tree for a particular item.
3. Insert an item in the binary tree.
4. Delete an item from the binary tree.
5. Find the height of the binary tree.
6. Find the number of nodes in the binary tree.
7. Find the number of leaves in the binary tree.
8. Traverse the binary tree.
9. Copy the binary tree.

The item search, insertion, and deletion operations all require the binary tree to be traversed. However, because the nodes of a binary tree are in no particular order, these algorithms are not very efficient on arbitrary binary trees. That is, no criteria exist to guide the search on these binary trees, as we will see in the next section. Therefore, we will discuss these algorithms when we discuss special binary trees.

The following class defines binary trees as an ADT. The definition of the node is the same as before. However, for the sake of completeness and easy reference, we give the definition of the node followed by the definition of the class.

```cpp
//Definition of the Node
template <class elemType>
struct nodeType
{
    elemType info;
    nodeType<elemType> *lLink;
    nodeType<elemType> *rLink;
};
```
/Definition of the class
template <class elemType>
class binaryTreeType
{
public:
    const binaryTreeType<elemType>& operator=
        (const binaryTreeType<elemType>&);
        //Overload the assignment operator.

    bool isEmpty() const;
        //Function to determine whether the binary tree is empty.
        //Postcondition: Returns true if the binary tree is empty;
        //otherwise, returns false.

    void inorderTraversal() const;
        //Function to do an inorder traversal of the binary tree.
        //Postcondition: Nodes are printed in inorder sequence.

    void preorderTraversal() const;
        //Function to do a preorder traversal of the binary tree.
        //Postcondition: Nodes are printed in preorder sequence.

    void postorderTraversal() const;
        //Function to do a postorder traversal of the binary tree.
        //Postcondition: Nodes are printed in postorder sequence.

    int treeHeight() const;
        //Function to determine the height of a binary tree.
        //Postcondition: Returns the height of the binary tree.

    int treeNodeCount() const;
        //Function to determine the number of nodes in a
        //binary tree.
        //Postcondition: Returns the number of nodes in the
        //binary tree.

    int treeLeavesCount() const;
        //Function to determine the number of leaves in a
        //binary tree.
        //Postcondition: Returns the number of leaves in the
        //binary tree.

    void destroyTree();
        //Function to destroy the binary tree.
        //Postcondition: Memory space occupied by each node
        //is deallocated.
        // root = NULL;

    virtual bool search(const elemType& searchItem) const = 0;
        //Function to determine if searchItem is in the binary
        //tree.
        //Postcondition: Returns true if searchItem is found in
        //the binary tree; otherwise, returns false.
virtual void insert(const elemType& insertItem) = 0;
//Function to insert insertItem in the binary tree.
//Postcondition: If there is no node in the binary tree
//    that has the same info as insertItem, a
//    node with the info insertItem is created
//    and inserted in the binary search tree.

virtual void deleteNode(const elemType& deleteItem) = 0;
//Function to delete deleteItem from the binary tree.
//Postcondition: If a node with the same info as
//    deleteItem is found, it is deleted from
//    the binary tree.
//    If the binary tree is empty or
//    deleteItem is not in the binary tree,
//    an appropriate message is printed.

binaryTreeType(const binaryTreeType<elemType>& otherTree);
//Copy constructor

binaryTreeType();
//Default constructor

~binaryTreeType();
//Destructor

protected:
    nodeType<elemType> *root;

private:
    void copyTree(nodeType<elemType>* &copiedTreeRoot,
                  nodeType<elemType>* otherTreeRoot);
    //Makes a copy of the binary tree to which
    //otherTreeRoot points.
    //Postcondition: The pointer copiedTreeRoot points to
    //    the root of the copied binary tree.

    void destroy(nodeType<elemType>* &p);
    //Function to destroy the binary tree to which p points.
    //Postcondition: Memory space occupied by each node, in
    //    the binary tree to which p points, is
    //    deallocated.
    //    p = NULL;

    void inorder(nodeType<elemType> *p) const;
    //Function to do an inorder traversal of the binary
    //tree to which p points.
    //Postcondition: Nodes of the binary tree, to which p
    //points, are printed in inorder sequence.

    void preorder(nodeType<elemType> *p) const;
    //Function to do a preorder traversal of the binary
    //tree to which p points.
    //Postcondition: Nodes of the binary tree, to which p
    //points, are printed in preorder
    //sequence.
void postorder(nodeType<elemType> *p) const;
  //Function to do a postorder traversal of the binary
  //tree to which p points.
  //Postcondition: Nodes of the binary tree, to which p
  //  points, are printed in postorder
  //  sequence.
int height(nodeType<elemType> *p) const;
  //Function to determine the height of the binary tree
  //to which p points.
  //Postcondition: Height of the binary tree to which
  //  p points is returned.
int max(int x, int y) const;
  //Function to determine the larger of x and y.
  //Postcondition: Returns the larger of x and y.
int nodeCount(nodeType<elemType> *p) const;
  //Function to determine the number of nodes in
  //the binary tree to which p points.
  //Postcondition: The number of nodes in the binary
  //  tree to which p points is returned.
int leavesCount(nodeType<elemType> *p) const;
  //Function to determine the number of leaves in
  //the binary tree to which p points.
  //Postcondition: The number of leaves in the binary
  //  tree to which p points is returned.
};

We leave the UML class diagram of the class binaryTreeType as an exercise for you. See Exercise 31 at the end of this chapter.

The functions search, insert, and deleteNode are declared as abstract in the definition of the class binaryTreeType. This is because, in this section, we are discussing arbitrary binary trees. Implementing these operations for arbitrary binary trees is inefficient, if not impossible, as we will discuss in the section “Binary Search Trees.” Because the class binaryTreeType contains abstract functions, this class is an abstract class. So, you cannot create objects of this class. In the section “Binary Search Tree,” we will derive a class from the class binaryTreeType and provide the definitions of these functions.

Note that the definition of the class binaryTreeType contains the statement to overload the assignment operator, copy constructor, and destructor. This is because the class binaryTreeType contains pointer member variables. Recall that for classes with pointer member variables, we must explicitly overload the assignment operator, include the copy constructor, and include the destructor.

The definition of the class binaryTreeType contains several member functions that are private members of the class. These functions are used to implement the public member functions of the class. For example, to do an inorder traversal, the function inorderTraversal calls the function inorder and passes the pointer root as a parameter to this function. Moreover, the pointer root is declared as a protected member so that we can later derive special binary trees.
Next, we give the definitions of the nonabstract member functions of the class `binaryTreeType`.

The binary tree is empty if `root` is `NULL`. So the definition of the function `isEmpty` is:

```cpp
template <class elemType>
bool binaryTreeType<elemType>::isEmpty() const
{
    return (root == NULL);
}
```

The default constructor initializes the binary tree to an empty state; that is, it sets the pointer `root` to `NULL`. Therefore, the definition of the default constructor is:

```cpp
template <class elemType>
binaryTreeType<elemType>::binaryTreeType()
{
    root = NULL;
}
```

The definitions of the other functions are:

```cpp
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal() const
{
    inorder(root);
}
```

```cpp
template <class elemType>
void binaryTreeType<elemType>::preorderTraversal() const
{
    preorder(root);
}
```

```cpp
template <class elemType>
void binaryTreeType<elemType>::postorderTraversal() const
{
    postorder(root);
}
```

```cpp
template <class elemType>
int binaryTreeType<elemType>::treeHeight() const
{
    return height(root);
}
```

```cpp
template <class elemType>
int binaryTreeType<elemType>::treeNodeCount() const
{
    return nodeCount(root);
}
```


```cpp
template <class elemType>
int binaryTreeType<elemType>::treeLeavesCount() const
{
    return leavesCount(root);
}

template <class elemType>
void binaryTreeType<elemType>::inorder
(nodeType<elemType> *p) const
{
    if (p != NULL)
    {
        inorder(p->lLink);
        cout << p->info << " ";
        inorder(p->rLink);
    }
}

template <class elemType>
void binaryTreeType<elemType>::preorder
(nodeType<elemType> *p) const
{
    if (p != NULL)
    {
        cout << p->info << " ";
        preorder(p->lLink);
        preorder(p->rLink);
    }
}

template <class elemType>
void binaryTreeType<elemType>::postorder
(nodeType<elemType> *p) const
{
    if (p != NULL)
    {
        postorder(p->lLink);
        postorder(p->rLink);
        cout << p->info << " ";
    }
}

template<class elemType>
int binaryTreeType<elemType>::height
(nodeType<elemType> *p) const
{
    if (p == NULL)
        return 0;
    else
        return 1 + max(height(p->lLink), height(p->rLink));
}
```
template <class elemType>
int binaryTreeType<elemType>::max(int x, int y) const
{
    if (x >= y)
        return x;
    else
        return y;
}

The definitions of the functions nodeCount and leavesCount are left as exercises for you. See Programming Exercises 1 and 2 at the end of this chapter.

Next, we give the definitions of the functions copyTree, destroy, destroyTree; the copy constructor; and the destructor. We also overload the assignment operator.

The definition of the function copyTree is the same as before; here, this function is a member of the class binaryTreeType.

template <class elemType>
void binaryTreeType<elemType>::copyTree
(nodeType<elemType>* &copiedTreeRoot, nodeType<elemType>* otherTreeRoot)
{
    if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL;
    else
    {
        copiedTreeRoot = new nodeType<elemType>;
        copiedTreeRoot->info = otherTreeRoot->info;
        copyTree(copiedTreeRoot->lLink, otherTreeRoot->lLink);
        copyTree(copiedTreeRoot->rLink, otherTreeRoot->rLink);
    }
} //end copyTree

To destroy a binary tree, for each node, first we destroy its left subtree, then its right subtree, and then the node itself. We must use the operator delete to deallocate the memory occupied by the node. The definition of the function destroy is:

template <class elemType>
void binaryTreeType<elemType>::destroy(nodeType<elemType>* &p)
{
    if (p != NULL)
    {
        destroy(p->lLink);
        destroy(p->rLink);
        delete p;
        p = NULL;
    }
}

To implement the function destroyTree, we use the function destroy and pass the root node of the binary tree to the function destroy. The definition of the function destroyTree is:
template <class elemType>
void binaryTreeType<elemType>::destroyTree()
{
    destroy(root);
}

Recall that when a class object is passed by value, the copy constructor copies the value of the actual parameters into the formal parameters. Because the class binaryTreeType has pointer member variables and a pointer is used to create dynamic memory, we must provide the definition of the copy constructor to avoid the shallow copying of data. The definition of the copy constructor, given next, uses the function copyTree to make an identical copy of the binary tree that is passed as a parameter.

    //copy constructor
template <class elemType>
binaryTreeType<elemType>::binaryTreeType(const binaryTreeType<elemType>& otherTree)
{
    if (otherTree.root == NULL) //otherTree is empty
        root = NULL;
    else
        copyTree(root, otherTree.root);
}

The definition of the destructor is quite straightforward. When the object of type binaryTreeType goes out of scope, the destructor deallocates the memory occupied by the nodes of the binary tree. The definition of the destructor uses the function destroy to accomplish this task.

    //Destructor
template <class elemType>
binaryTreeType<elemType>::~binaryTreeType()
{
    destroy(root);
}

Next, we discuss the definition of the function to overload the assignment operator. To assign the value of one binary tree to another binary tree, we make an identical copy of the binary tree to be assigned by using the function copyTree. The definition of the function to overload the assignment operator is:

    //Overload the assignment operator
template <class elemType>
const binaryTreeType<elemType>& binaryTreeType<elemType>::operator=(const binaryTreeType<elemType>& otherTree)
{
    if (this != &otherTree) //avoid self-copy
    {
        if (root != NULL) //if the binary tree is not empty,
            //destroy the binary tree
            destroy(root);
    
    return *this;
}
if (otherTree.root == NULL) //otherTree is empty
    root = NULL;
else
    copyTree(root, otherTree.root);
}//end else
return *this;

Binary Search Trees

Now that you know the basic operations on a binary tree, this section discusses a special type of binary tree called the binary search tree.

Consider the binary tree in Figure 19-7.

Suppose that we want to determine whether 53 is in the binary tree. To do so, we can use any of the previous traversal algorithms to visit each node and compare the search item with the data stored in the node. However, this could require us to traverse a large part of the binary tree, so the search will be slow. The reason that we need to visit each node in the binary tree until either the item is found or we have traversed the entire binary tree is that no criteria exist to guide our search. This case is like an arbitrary linked list, in which we must start our search at the first node and continue looking at each node until either the item is found or the entire list is searched.

On the other hand, consider the binary tree in Figure 19-8.
In the binary tree in Figure 19-8, the data in each node is:

- Larger than the data in its left child
- Smaller than the data in its right child

The binary tree in Figure 19-8 has some order to its nodes. Suppose that we want to determine whether 58 is in this binary tree. As before, we must start our search at the root node. We compare 58 with the data in the root node; that is, we compare 58 with 60. Because 58 \( \neq \) 60 and 58 \( < \) 60, it is guaranteed that 58 will not be in the right subtree of the root node. Therefore, if 58 is in the binary tree, then it must be in the left subtree of the root node. We follow the left pointer of the root node and go to the node with info 50. We now apply the same criteria at this node. Because 58 \( > \) 50, we must follow the right pointer of this node and go to the node with info 58. At this node, we find 58.

This example shows that every time we move down to a child, we eliminate one of the subtrees of the node from our search. If the binary tree is nicely constructed, then the search is very similar to the binary search on arrays.

The binary tree given in Figure 19-8 is a special type of binary tree called a binary search tree. (In the following definition, by the term key of the node, we mean the key of the data item that uniquely identifies the item.)

**Definition:** A **binary search tree**, \( T \), is either empty or:

i. \( T \) has a special node called the **root** node;

ii. \( T \) has two sets of nodes, \( L_T \) and \( R_T \), called the left subtree and right subtree of \( T \), respectively;
iii. The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree; and
iv. $L_T$ and $R_T$ are binary search trees.

The following operations are typically performed on a binary search tree.

1. Determine whether the binary search tree is empty.
2. Search the binary search tree for a particular item.
3. Insert an item in the binary search tree.
4. Delete an item from the binary search tree.
5. Find the height of the binary search tree.
6. Find the number of nodes in the binary search tree.
7. Find the number of leaves in the binary search tree.
8. Traverse the binary search tree.
9. Copy the binary search tree.

Clearly, every binary search tree is a binary tree. The height of a binary search tree is determined in the same way as the height of a binary tree. Similarly, the operations to find the number of nodes, to find the number of leaves, and to do inorder, preorder, and postorder traversals of a binary search tree are the same as those for a binary tree. Therefore, we can inherit all of these operations from the binary tree. That is, we can extend the definition of the binary tree by using the principle of inheritance and hence define the binary search tree.

The following class defines a binary search tree as an ADT by extending the definition of the binary tree:

```cpp
template <class elemType>
class bSearchTreeType: public binaryTreeType<elemType> {
    public:
        bool search(const elemType& searchItem) const;
        //Function to determine if searchItem is in the binary search tree.
        //Postcondition: Returns true if searchItem is found in the binary search tree; otherwise, returns false.

        void insert(const elemType& insertItem);
        //Function to insert insertItem in the binary search tree.
        //Postcondition: If there is no node in the binary search tree that has the same info as insertItem, a node with the info insertItem is created and inserted in the binary search tree.
```
void deleteNode(const elemType& deleteItem);
//Function to delete deleteItem from the binary search tree.
//Postcondition: If a node with the same info as deleteItem
//is found, it is deleted from the binary
//search tree.
//If the binary tree is empty or deleteItem
//is not in the binary tree, an appropriate
//message is printed.

private:
void deleteFromTree(nodeType<elemType>* &p);
//Function to delete the node to which p points is
//deleted from the binary search tree.
//Postcondition: The node to which p points is deleted
//from the binary search tree.

};

We leave it as an exercise for you to draw the UML class diagram of the class bSearchTreeType and the inheritance hierarchy. See Exercise 32 at the end of this chapter.

Next, we describe each of these operations.

SEARCH
The function search searches the binary search tree for a given item. If the item is found in the binary search tree, it returns true; otherwise, it returns false. Because the pointer root points to the root node of the binary search tree, we must begin our search at the root node. Furthermore, because root must always point to the root node, we need a pointer—say, current—to traverse the binary search tree. The pointer current is initialized to root.

If the binary search tree is nonempty, we first compare the search item with the info in the root node. If they are the same, we stop the search and return true. Otherwise, if the search item is smaller than the info in the node, we follow lLink to go to the left subtree; otherwise, we follow rLink to go to the right subtree. We repeat this process for the next node. If the search item is in the binary search tree, our search ends at the node containing the search item; otherwise, the search ends at an empty subtree. Thus, the general algorithm is:

if root is NULL
    Cannot search an empty tree, returns false.
else
    { current = root;
      while (current is not NULL and not found)
         if (current->info is the same as the search item)
             set found to true;
         else
             if (current->info is greater than the search item)
                 follow the lLink of current
             else
                 follow the rLink of current
    }
This pseudocode algorithm translates into the following C++ function:

```cpp
template <class elemType>
bool bSearchTreeType<elemType>::search(
    const elemType& searchItem) const
{
    nodeType<elemType> *current;
    bool found = false;

    if (root == NULL)
        cout << "Cannot search an empty tree." << endl;
    else
    {
        current = root;
        while (current != NULL && !found)
        {
            if (current->info == searchItem)
                found = true;
            else if (current->info > searchItem)
                current = current->lLink;
            else
                current = current->rLink;
        } // end while
    } // end else

    return found;
} // end search
```

**INSERT**

After inserting an item in a binary search tree, the resulting binary tree must be a binary search tree. To insert a new item, first we search the binary search tree and find the place where the new item is to be inserted. The search algorithm is similar to the search algorithm of the function search. Here, we traverse the binary search tree with two pointers—a pointer, say, `current`, to check the current node and a pointer, say, `trailCurrent`, pointing to the parent of `current`. Because duplicate items are not allowed, our search must end at an empty subtree. We can then use the pointer `trailCurrent` to insert the new item at the proper place. The item to be inserted, `insertItem`, is passed as a parameter to the function `insert`. The general algorithm is:

a. Create a new node and copy `insertItem` into the new node. Also set `lLink` and `rLink` of the new node to `NULL`.

b. If the root is `NULL`, the tree is empty, so make root point to the new node.
   ```cpp
   else
   {
       current = root;
       while (current is not NULL) // search the binary tree
   ```


```c++
template <class elemType>
void bSearchTreeType<elemType>::insert(
    const elemType& insertItem)
{
    nodeType<elemType> *current; //pointer to traverse the tree
    nodeType<elemType> *trailCurrent; //pointer behind current
    nodeType<elemType> *newNode; //pointer to create the node

    newNode = new nodeType<elemType>;
    newNode->info = insertItem;
    newNode->lLink = NULL;
    newNode->rLink = NULL;

    if (root == NULL)
        root = newNode;
    else
    {
        current = root;
        while (current != NULL)
        {
            trailCurrent = current;
            if (current->info == insertItem)
            {
                cout << "The item to be inserted is already ";
                cout << "in the tree -- duplicates are not " << endl;
                return;
            }
            else if (current->info > insertItem)
                current = current->lLink;
            else
                current = current->rLink;
        } //end while
    }
}
```
if (trailCurrent->info > insertItem)
    trailCurrent->lLink = newNode;
else
    trailCurrent->rLink = newNode;
}
} // end insert

DELETE

As before, first we search the binary search tree to find the node to be deleted. To help you better understand the delete operation, before describing the function to delete an item from the binary search tree, let us consider the binary search tree in Figure 19-9.

![Binary search tree before deleting a node]

After deleting the desired item (if it exists in the binary search tree), the resulting tree must be a binary search tree. The delete operation has four cases, as follows:

Case 1: The node to be deleted has no left and right subtrees; that is, the node to be deleted is a leaf. For example, the node with info 45 is a leaf.

Case 2: The node to be deleted has no left subtree; that is, the left subtree is empty, but it has a nonempty right subtree. For example, the left subtree of node with info 40 is empty, and its right subtree is nonempty.
**Case 3:** The node to be deleted has no right subtree; that is, the right subtree is empty, but it has a nonempty left subtree. For example, the left subtree of node with info 80 is empty, and its right subtree is nonempty.

**Case 4:** The node to be deleted has nonempty left and right subtrees. For example, the left and the right subtrees of node with info 50 are nonempty.

Figure 19-10 illustrates these four cases.
Case 1: Suppose that we want to delete 45 from the binary search tree in Figure 19-9. We search the binary tree and arrive at the node containing 45. Because this node is a leaf and is the left child of its parent, we can simply set the lLink of the parent node to NULL and deallocate the memory occupied by this node. After deleting this node, Figure 19-10(a) shows the resulting binary search tree.

Case 2: Suppose that we want to delete 30 from the binary search tree in Figure 19-9. In this case, the node to be deleted has no left subtree. Because 30 is the left child of its parent node, we make the lLink of the parent node point to the right child of 30 and then deallocate the memory occupied by 30. Figure 19-10(b) shows the resulting binary tree.

Case 3: Suppose that we want to delete 80 from the binary search tree of Figure 19-9. The node containing 80 has no right child and is the right child of its parent. Thus, we make the rLink of the parent of 80—that is, 70—point to the left child of 80. Figure 19-10(c) shows the resulting binary tree.

Case 4: Suppose that we want to delete 50 from the binary search tree in Figure 19-9. The node with info 50 has a nonempty left subtree and a nonempty right subtree. Here, we first reduce this case to either case 2 or case 3 as follows. To be specific, suppose that we reduce it to case 3—that is, the node to be deleted has no right subtree. For this case, we find the immediate predecessor of 50 in this binary tree, which is 48. This is done by first going to the left child of 50 and then locating the rightmost node of the left subtree of 50. To do so, we follow the rLink of the nodes. Because the binary search tree is finite, we eventually arrive at a node that has no right subtree. Next, we swap the info in the node to be deleted with the info of its immediate predecessor. In this case, we swap 48 with 50. This reduces to the case wherein the node to be deleted has no right subtree. We now apply case 3 to delete the node. (Note: Because we will delete the immediate predecessor from the binary tree, we, in fact, copy only the info of the immediate predecessor into the node to be deleted.) After deleting 50 from the binary search tree in Figure 19-9, the resulting binary tree is as shown in Figure 19-10(d).

In each case, we clearly see that the resulting binary tree is again a binary search tree. From this discussion, it follows that to delete an item from the binary search tree, we must do the following:

1. Find the node containing the item (if any) to be deleted.
2. Delete the node.

We accomplish the second step by a separate function, which we will call deleteFromTree. Given a pointer to the node to be deleted, this function deletes the node by taking into account the previous four cases.

From the preceding examples, it is clear that whenever we delete a node from the binary tree, we adjust one of the pointers of the parent node. Because the adjustment has to be made in the parent node, we must call the function deleteFromTree by using an appropriate pointer of the parent node. For example, suppose that the node to be deleted is 35, which is the right child of its parent node. Suppose that trailCurrent points to the node containing 30, the parent node of 35. A call to the function deleteFromTree is:
deleteFromTree(trailCurrent->rLink);
Of course, if the node to be deleted is the root node, then the call to the function deleteFromTree is:
deleteFromTree(root);

We now define the C++ function deleteFromTree.

template <class elemType>
void bSearchTreeType<elemType>::deleteFromTree
(nodeType<elemType>* &p)
{
    nodeType<elemType> *current; //pointer to traverse the tree
    nodeType<elemType> *trailCurrent; //pointer behind current
    nodeType<elemType> *temp; //pointer to delete the node

    if (p == NULL)
        cout << "Error: The node to be deleted is NULL."
        << endl;
    else if (p->lLink == NULL && p->rLink == NULL)
    {                                 //current did not move;
        temp = p;
        p = NULL;
        delete temp;
    }
    else if (p->lLink == NULL)
    {                                 //current == p->lLink; adjust p
        temp = p;
        p = temp->rLink;
        delete temp;
    }
    else if (p->rLink == NULL)
    {                                 //current == p->rLink; adjust p
        temp = p;
        p = temp->lLink;
        delete temp;
    }
    else
    {                                 //current did not move;
        current = p->lLink;
        trailCurrent = NULL;

        while (current->rLink != NULL)
        {                                 //current did not move;
            trailCurrent = current;
            current = current->rLink;
        }//end while

        p->info = current->info;

        if (trailCurrent == NULL)      //current did not move;
            p->lLink = current->lLink;
    }
}
else
    trailCurrent->rLink = current->lLink;

delete current;
} //end else
} //end deleteFromTree

Next, we describe the function deleteNode. The function deleteNode first searches the binary search tree to find the node containing the item to be deleted. The item to be deleted, deleteItem, is passed as a parameter to the function. If the node containing deleteItem is found in the binary search tree, the function deleteNode calls the function deleteFromTree to delete the node. The definition of the function deleteNode is given next.

template <class elemType>
void bSearchTreeType<elemType>::deleteNode
(const elemType& deleteItem)
{
    nodeType<elemType> *current; //pointer to traverse the tree
    nodeType<elemType> *trailCurrent; //pointer behind current
    bool found = false;

    if (root == NULL)
        cout << "Cannot delete from an empty tree."
        << endl;
    else
    {
        current = root;
        trailCurrent = root;

        while (current != NULL && !found)
        {
            if (current->info == deleteItem)
                found = true;
            else
            {
                trailCurrent = current;

                if (current->info > deleteItem)
                    current = current->lLink;
                else
                    current = current->rLink;
            }
        } //end while

        if (current == NULL)
            cout << "The item to be deleted is not in the tree."
            << endl;
        else if (found)
        {
            if (current == root)
                deleteFromTree(root);
            else
                trailCurrent->rLink = current->lLink;
        } //end else
    } //end deleteFromTree

else if (trailCurrent->info > deleteItem)
    deleteFromTree(trailCurrent->lLink);
else
    deleteFromTree(trailCurrent->rLink);
}
else
    cout << "The item to be deleted is not in the tree."
    << endl;
} // end deleteNode

**Binary Search Tree: Analysis**

Let $T$ be a binary search tree with $n$ nodes, in which $n > 0$. Suppose that we want to determine whether an item, $x$, is in $T$. The performance of the search algorithm depends on the shape of $T$. Let us first consider the worst case. In the worst case, $T$ is linear. That is, the $T$ is one of the forms shown in Figure 19-11.

![Figure 19-11 Linear binary search trees](image)

Because $T$ is linear, the performance of the search algorithm on $T$ is the same as its performance on a linear list. Therefore, in the successful case, on average, the search algorithm makes $\frac{n + 1}{2} = O(n)$ key comparisons. In the unsuccessful case, it makes $n$ comparisons.

Let us now consider the average-case behavior. In the successful case, the search would end at a node. Because there are $n$ items, there are $n!$ possible orderings of the keys. We assume that all $n!$ orderings of the keys are possible. Let $S(n)$ denote the number of comparisons in the average successful case, and let $U(n)$ denote the number of comparisons in the average unsuccessful case.

The number of comparisons required to determine whether $x$ is in $T$ is one more than the number of comparisons required to insert $x$ in $T$. Furthermore, the number of
comparisons required to insert \( x \) in \( T \) is the same as the number of comparisons made in the unsuccessful search, reflecting that \( x \) is not in \( T \). From this, it follows that:

\[
S(n) = 1 + \frac{U(0) + U(1) + \ldots + U(n-1)}{n}
\]  
(19-1)

It is also known that:

\[
S(n) = \left(1 + \frac{1}{n}\right) U(n) - 3
\]  
(19-2)

Solving equations (19-1) and (19-2), it can be shown that:

\[
U(n) \approx 2.77\log_2 n = O(\log_2 n)
\]

and:

\[
S(n) \approx 2.77\log_2 n = O(\log_2 n)
\]

We can now formulate the following result.

**Theorem:** Let \( T \) be a binary search tree with \( n \) nodes, in which \( n > 0 \). The average number of nodes visited in a search of \( T \) is approximately \( 1.39\log_2 n = O(\log_2 n) \), and the number of key comparisons is approximately \( 2.77\log_2 n = O(\log_2 n) \).

**Nonrecursive Binary Tree Traversal Algorithms**

The previous sections described how to do the following:

- Traverse a binary tree using the inorder, preorder, and postorder methods.
- Construct a binary tree.
- Insert an item in the binary tree.
- Delete an item from the binary tree.

The traversal algorithms—inorder, preorder, and postorder—discussed earlier are recursive. Because traversing a binary tree is a fundamental operation, this section discusses the nonrecursive inorder, preorder, and postorder traversal algorithms.

**Nonrecursive Inorder Traversal**

In the inorder traversal of a binary tree, for each node, the left subtree is visited first, then the node, and then the right subtree. It follows that in an inorder traversal, the first node visited is the leftmost node of the binary tree. For example, in the binary tree in Figure 19-12, the leftmost node is the node with info 28.
To get to the leftmost node of the binary tree, we start by traversing the binary tree at the root node and then follow the left link of each node until the left link of a node becomes null. From this point, we back up to the parent node, visit the node, and then move to the right node. Because links go in only one direction, to get back to a node, we must save a pointer to the node before moving to the child node. Moreover, the nodes must be backtracked in the order they were traversed. It follows that while backtracking, the nodes must be visited in a last-in first-out manner. This can be done by using a stack. We, therefore, save a pointer to a node in a stack. The general algorithm is as follows:

1. \( \text{current} = \text{root}; \) //Start traversing the binary tree at the root node

2. \( \text{while (current is not NULL or stack is nonempty)} \)
   \( \text{if (current is not NULL)} \)
   \( \{ \)
   \( \text{push current onto stack;} \)
   \( \text{current} = \text{current->lLink}; \)
   \( \} \)
   \( \text{else} \)
   \( \{ \)
   \( \text{current} = \text{stack.top();} \)
   \( \text{pop stack;} \)
   \( \text{visit current;} \) //visit the node
   \( \text{current} = \text{current->rLink;} \) //move to the right child
   \( \} \)

The following function implements the nonrecursive inorder traversal of a binary tree:

\[
\text{template <class elemType>}
\text{void binaryTreeType<elemType>::nonRecursiveInTraversal() const}
\{ \\
\text{stackType<nodeType<elemType>*> stack;} \\
\text{nodeType<elemType> *current;} \\
\text{current = root;} \\
\]
while ((current != NULL) || (!stack.isEmptyStack()))
if (current != NULL)
{
    stack.push(current);
    current = current->lLink;
}
else
{
    current = stack.top();
    stack.pop();
    cout << current->info << " ";
    current = current->rLink;
}

cout << endl;
} //end nonRecursiveInTraversal

Nonrecursive Preorder Traversal

In a preorder traversal of a binary tree, for each node, first the node is visited, then the left subtree is visited, and then the right subtree is visited. As in the case of an inorder traversal, after visiting a node and before moving to the left subtree, we must save a pointer to the node so that after visiting the left subtree, we can visit the right subtree. The general algorithm is as follows:

1. current = root;  //start the traversal at the root node
2. while (current is not NULL or stack is nonempty)
   if (current is not NULL)
   {
       visit current node;
       push current onto stack;
       current = current->lLink;
   }
   else
   {
       current = stack.top();
       pop stack;
       current = current->rLink;  //move to the right child
   }

The following function implements the nonrecursive preorder traversal algorithm:

template <class elemType>
void binaryTreeType<elemType>::nonRecursivePreTraversal() const
{
    stackType<nodeType<elemType>*>* stack;
    nodeType<elemType> *current;

    current = root;
while ((current != NULL) || (!stack.isEmptyStack()))
if (current != NULL)
{
    cout << current->info << " ";
    stack.push(current);
    current = current->lLink;
}
else
{
    current = stack.top();
    stack.pop();
    current = current->rLink;
}

cout << endl;
} //end nonRecursivePreTraversal

Nonrecursive Postorder Traversal

In a postorder traversal of a binary tree, for each node, first the left subtree is visited, then
the right subtree is visited, and then the node is visited. As in the case of an inorder
traversal, in a postorder traversal, the first node visited is the leftmost node of the binary
tree. Because—for each node—the left and right subtrees are visited before visiting the
node, we must indicate to the node whether the left and right subtrees have been visited.
After visiting the left subtree of a node and before visiting the node, we must visit its right
subtree. Therefore, after returning from a left subtree, we must tell the node that the right
subtree needs to be visited, and after visiting the right subtree, we must tell the node that
it can now be visited. To do this, other than saving a pointer to the node (to get back to
the right subtree and to the node itself), we also save an integer value of 1 before moving
to the left subtree and an integer value of 2 before moving to the right subtree. Whenever
the stack is popped, the integer value associated with that pointer is popped as well. This
integer value tells whether the left and right subtrees of a node have been visited.

The general algorithm is:

1. current = root; //start the traversal at the root node
2. v = 0;
3. if current is NULL
   The binary tree is empty
4. if current is not NULL
   a. push current onto stack;
   b. push 1 onto stack;
   c. current = current->lLink;
   d. while (stack is not empty)
      if (current is not NULL and v is 0)
{    push current and 1 onto stack;    current = current->lLink;}
else
{
    assign the top element of stack to current and v;
    pop stack;
    if (v == 1)
    {
        push current and 2 onto stack;
        current = current->rLink;
        v = 0;
    }
else
    visit current;
}

We will use two (parallel) stacks: one to save a pointer to a node and another to save the integer value (1 or 2) associated with this pointer. We leave it as an exercise for you to write the definition of a C++ function to implement the preceding postorder traversal algorithm. See Programming Exercise 6 at the end of this chapter.

Binary Tree Traversal and Functions as Parameters

Suppose that you have stored employee data in a binary search tree, and at the end of the year pay increases or bonuses are to be awarded to each employee. This task requires that each node in the binary search tree be visited and that the salary of each employee be updated. The preceding sections discussed various ways to traverse a binary tree. However, in these traversal algorithms—inorder, preorder, and postorder—whenever we visited a node, for simplicity and for illustration purposes, we output only the data contained in each node. How do we use a traversal algorithm to visit each node and update the data in each node? One way to do so is to first create another binary search tree in which the data in each node is the updated data of the original binary search tree and then destroy the old binary search tree. This would require extra computer time and perhaps extra memory and, therefore, is not efficient. Another solution is to write separate traversal algorithms to update the data. This solution requires you to frequently modify the definition of the class implementing the binary search tree. However, if the user can write an appropriate function to update the data of each employee and then pass the function as a parameter to the traversal algorithms, we can considerably enhance the program’s flexibility. This section describes how to pass functions as parameters to other functions.

In C++, a function name without any parentheses is considered a pointer to the function. To specify a function as a formal parameter to another function, we specify the function type, followed by the function name as a pointer, followed by the parameter types of the function. For example, consider the following statements:

```c++
void fParamFunc1(void (*visit) (int));          //Line 1
void fParamFunc2(void (*visit) (elemType&));    //Line 2
```
The statement in Line 1 declares `fParamFunc1` to be a function that takes as a parameter any `void` function that has one value parameter of type `int`. The statement in Line 2 declares `fParamFunc2` to be a function that takes as a parameter any `void` function that has one reference parameter of type `elemType`.

We can now rewrite, say, the inorder traversal function of the `class binaryTreeType`. Alternately, we can overload the existing inorder traversal functions. To further illustrate function overloading, we will overload the inorder traversal functions. Therefore, we include the following statements in the definition of the `class binaryTreeType`:

```cpp
void inorderTraversal(void (*visit) (elemType&)) const;
//Function to do an inorder traversal of the binary tree.
//The parameter visit, which is a function, specifies
//the action to be taken at each node.
//Postcondition: The action specified by the function
//visit is applied to each node of the
//binary tree.

void inorder(nodeType<elemType> *p, void (*visit) (elemType&)) const;
//Function to do an inorder traversal of the binary tree
//starting at the node specified by the parameter p.
//The parameter visit, which is a function, specifies the
//action to be taken at each node.
//Postcondition: The action specified by the function visit
//is applied to each node of the binary tree
//to which p points.
```

The definitions of these functions are as follows:

```cpp
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal
(void (*visit) (elemType& item)) const
{
   inorder(root, *visit);
}

template <class elemType>
void binaryTreeType<elemType>::inorder(nodeType<elemType>* p, void (*visit) (elemType& item)) const
{
   if (p != NULL)
   {
      inorder(p->lLink, *visit);
      (*visit)(p->info);
      inorder(p->rLink, *visit);
   }
}
```

The statement:

```cpp
(*visit)(p->info);
```

in the definition of the function `inorder` makes a call to the function with one reference parameter of type `elemType` pointed to by the pointer `visit`. 

Example 19-5 further illustrates how functions are passed as parameters to other functions.

**EXAMPLE 19-5**

This example shows how to pass a user-defined function as a parameter to the binary tree traversal algorithms. For illustration purposes, we show how to use only the inorder traversal function.

The following program uses the class `bSearchTreeType`, which is derived from the class `binaryTreeType`, to build the binary tree. The traversal functions are included in the class `binaryTreeType`, which are then inherited by the class `bSearchTreeType`.

```cpp
#include <iostream>
#include "binarySearchTree.h"

using namespace std;

void print(int& x);
void update(int& x);

int main()
{
    bSearchTreeType<int> treeRoot;     //Line 1

    int num;                          //Line 2

    cout << "Line 3: Enter numbers ending "
        << "with -999." << endl;    //Line 3
    cin >> num;                       //Line 4

    while (num != -999)               //Line 5
    {
        treeRoot.insert(num);        //Line 6
        cin >> num;                  //Line 7
    }

    cout << endl
        << "Line 8: Tree nodes in inorder: ";  //Line 8
    treeRoot.inorderTraversal(print); //Line 9
    cout << endl << "Line 10: Tree Height: "
        << treeRoot.treeHeight() 
        << endl << endl;            //Line 10

    cout << "Line 11: ******* Update Nodes "
        << "*******" << endl;       //Line 11
    treeRoot.inorderTraversal(update); //Line 12

    cout << "Line 13: Tree nodes in inorder "
        << "after the update: " << endl
        << " ";                       //Line 13
    treeRoot.inorderTraversal(print); //Line 14
```
cout << endl << "Line 15: Tree Height: " << treeRoot.treeHeight() << endl; //Line 15

return 0; //Line 16
}

void print(int& x) //Line 17
{
    cout << x << " "; //Line 18
}

void update(int& x) //Line 19
{
    x = 2 * x; //Line 20
}

Sample Run: In this sample run, the user input is shaded.

Line 3: Enter numbers ending with -999.
56 87 23 65 34 45 12 90 66 -999

Line 8: Tree nodes in inorder: 12 23 34 45 56 65 66 87 90
Line 10: Tree Height: 4

Line 11: ******* Update Nodes *******
Line 13: Tree nodes in inorder after the update:
    24 46 68 90 112 130 132 174 180
Line 15: Tree Height: 4

This program works as follows. The statement in Line 1 declares treeRoot to be a binary search tree object, in which the data in each node is of type int. The statements in Lines 4 through 7 build the binary search tree. The statement in Line 9 uses the member function inorderTraversal of treeRoot to traverse the binary search tree treeRoot. The parameter to the function inorderTraversal, in Line 9, is the function print (defined at Line 17). Because the function print outputs the value of its argument, the statement in Line 9 outputs the data of the nodes of the binary search tree treeRoot. The statement in Line 10 outputs the height of the binary search tree.

The statement in Line 12 uses the member function inorderTraversal to traverse the binary search tree treeRoot. In Line 12, the actual parameter of the function inorderTraversal is the function update (defined at Line 19). The function update doubles the value of its argument. Therefore, the statement in Line 12 updates the data of each node of the binary search tree by doubling the value. The statements in Lines 14 and 15 output the nodes and the height of the binary search tree.

NOTE (AVL trees) This chapter also discusses AVL trees. The necessary material is in the file AVL Trees.pdf. This file is on the Web site, www.course.com/malik/cpp, accompanying this book.
PROGRAMMING EXAMPLE: DVD Store (Revisited)

In Chapter 16, we designed a program to help a DVD store automate its DVD rental process. That program used an (unordered) linked list to keep track of the DVD inventory in the store. Because the search algorithm on a linked list is sequential and the list is fairly large, the search could be time consuming. In this chapter, you learned how to organize data into a binary tree. If the binary tree is nicely constructed (that is, it is not linear), then the search algorithm can be improved considerably. Moreover, in general, item insertion and deletion in a binary search tree are faster than in a linked list. We will, therefore, redesign the DVD store program so that the DVD inventory can be maintained in a binary tree. As in Chapter 16, we leave the design of the customer list in a binary tree as exercises for you.

**DVD Object**

In Chapter 16, a linked list was used to maintain a list of DVDs in the store. Because the linked list was unordered, to see whether a particular DVD was in stock, the sequential search algorithm used the equality operator for comparison. However, in the case of a binary tree, we need other relational operators for the search, insertion, and deletion operations. We will, therefore, overload all of the relational operators. Other than this difference, the class `dvdType` is the same as before. However, we give its definition for the sake of completeness.

```
//*************************************************************
// Author: D.S. Malik
//
// class dvdType
// This class specifies the members to implement a DVD. It
// overloads the stream insertion operator and relational
// operators.
//*************************************************************

class dvdType
{
    friend ostream& operator<<(ostream&, const dvdType&);  

public:
    void setDVDInfo(string title, string star1,
                     string star2, string producer,
                     string director, string productionCo,
                     int setInStock);
    //Function to set the details of a DVD.
    //The member variables are set according to the
    //parameters.
    //Postcondition: dvdTitle = title; movieStar1 = star1;
    //               movieStar2 = star2;
    //               movieProducer = producer;
    //               movieDirector = director;
    //               movieProductionCo = productionCo;
    //               copiesInStock = setInStock;
```
int getNoOfCopiesInStock() const;
    //Function to check the number of copies in stock.
    //Postcondition: The value of copiesInStock is returned.

void checkOut();
    //Function to rent a DVD.
    //The number of copies in stock is decremented by one.
    //Postcondition: copiesInStock--;

void checkIn();
    //Function to check in a DVD.
    //The number of copies in stock is incremented by one.
    //Postcondition: copiesInStock++;

void printTitle() const;
    //Function to print the title of a movie.

void printInfo() const;
    //Function to print the details of a DVD.
    //Postcondition: The title of the movie, stars, director,
    // and so on are output on the screen.

bool checkTitle(string title);
    //Function to check whether the title is the same as the
    //title of the DVD.
    //Postcondition: Returns the value true if the title is
    //the same as the title of the DVD, and
    //false otherwise.

void updateInStock(int num);
    //Function to increment the number of copies in stock by
    //adding the value of the parameter num.
    //Postcondition: copiesInStock = copiesInStock + num;

void setCopiesInStock(int num);
    //Function to set the number of copies in stock.
    //Postcondition: copiesInStock = num;

string getTitle() const;
    //Function to return the title of the DVD.
    //Postcondition: The title of the DVD is returned.

dvdType(string title = "", string star1 = "", string star2 = "", string producer = "", string director = "", string productionCo = ", int setInStock = 0);
    //Constructor
    //The member variables are set according to the incoming
    //parameters. If no values are specified, the default
    //values are assigned.
The definitions of the member functions of the class dvdType are the same as in Chapter 16. Because here we are overloading all of the relational operators, we give only the definitions of these member functions.

```cpp
//Overload the relational operators
bool dvdType::operator==(const dvdType& right) const
{
    return (dvdTitle == right.dvdTitle);
}

bool dvdType::operator!=(const dvdType& right) const
{
    return (dvdTitle != right.dvdTitle);
}

bool dvdType::operator<(const dvdType& right) const
{
    return (dvdTitle < right.dvdTitle);
}

bool dvdType::operator>=(const dvdType& right) const
{
    return (dvdTitle >= right.dvdTitle);
}

bool dvdType::operator<=(const dvdType& right) const
{
    return (dvdTitle <= right.dvdTitle);
}

bool dvdType::operator>(const dvdType& right) const
{
    return (dvdTitle > right.dvdTitle);
}
```
bool dvdType::operator<=(const dvdType& right) const
{
    return (dvdTitle <= right.dvdTitle);
}

bool dvdType::operator>(const dvdType& right) const
{
    return (dvdTitle > right.dvdTitle);
}

bool dvdType::operator>=(const dvdType& right) const
{
    return (dvdTitle >= right.dvdTitle);
}

**DVD List**
The DVD list is maintained in a binary search tree. Therefore, we derive the class dvdBinaryTree from the class bSearchTreeType. The definition of the class dvdBinaryTree is as follows:

```cpp
//***********************************************************
// Author: D.S. Malik
//
// class dvdBinaryTree
// This class extends the class bSearchTreeType to create
// a DVD list.
//***********************************************************

class dvdBinaryTree: public bSearchTreeType<dvdType>
{
    public:
        bool dvdSearch(string title);
        // Function to search the list to see whether a
        // particular title, specified by the parameter title,
        // is in the store.
        // Postcondition: Returns true if the title is found,
        // and false otherwise.

        bool isDVDAvailable(string title);
        // Function to determine whether a copy of a particular
        // DVD is in the store.
        // Postcondition: Returns true if at least one copy of
        // the DVD specified by title is in the
        // store, and false otherwise.

        void dvdCheckIn(string title);
        // Function to check in a DVD returned by a customer.
        // Postcondition: copiesInStock is incremented by one.

        void dvdCheckOut(string title);
        // Function to check out a DVD, that is, rent a DVD.
        // Postcondition: copiesInStock is decremented by one.
```
bool dvdCheckTitle(string title) const;
//Function to determine whether a particular DVD is in
//the store.
//Postcondition: Returns true if the DVD’s title is
//the same as title, and false otherwise.

void dvdUpdateInStock(string title, int num);
//Function to update the number of copies of a DVD
//by adding the value of the parameter num. The
//parameter title specifies the name of the DVD for
//which the number of copies is to be updated.
//Postcondition: copiesInStock = copiesInStock + num;

void dvdSetCopiesInStock(string title, int num);
//Function to reset the number of copies of a DVD.
//The parameter title specifies the name of the DVD
//for which the number of copies is to be reset, and
//the parameter num specifies the number of copies.
//Postcondition: copiesInStock = num;

void dvdPrintTitle() const;
//Function to print the titles of all the DVDs in
//the store.

private:
void searchDVDList(string title, bool & found,
    nodeType<dvdType>* &current) const;
//This function searches the DVD list for a
//particular DVD, specified by the parameter title.
//If the DVD is found, the parameter found is set to
//true, otherwise false; the parameter current points
//to the node containing the DVD.

void inorderTitle(nodeType<dvdType> *p) const;
//This function prints the titles of all the DVDs
//in stock.
};

The definitions of the member functions isDVDAvailable, dvdCheckIn,
dvdCheckOut, dvdCheckTitle, dvdUpdateInStock,
dvdSetCopiesInStock, and dvdSearch of the class dvdBinaryTree are
similar to the definitions of these functions given in Chapter 16. The only difference
is that, here, these are members of the class dvdBinaryTree. You can find the
complete definitions of these functions on the Web site that accompanies this book.
Next, we discuss the definitions of the remaining functions of the
class dvdBinaryTree.

The function searchDVDList uses a search algorithm similar to the search
algorithm for a binary search tree given earlier in this chapter. It returns true if the
search item is found in the list. It also returns a pointer to the node containing the
search item. The definition of this function is as follows:
void dvdBinaryTree::searchDVDList(string title,
    bool& found,
    nodeType<dvdType>* &current) const
{
    found = false;
    dvdType temp;
    temp.setDVDInfo(title, "", "", "", "", "", 0);
    if (root == NULL) //tree is empty
        cout << "Cannot search an empty list. " << endl;
    else
    { //set current point to the root node
        current = root;
        //of the binary tree
        found = false; //set found to false
        while (current != NULL && !found) //search the tree
            if (current->info == temp) //item is found
                found = true;
            else if (current->info > temp)
                current = current->lLink;
            else
                current = current->rLink;
    } //end else
} //end searchDVDList

Given a pointer to the root node of the binary tree containing the DVDs, the
function inorderTitle uses the inorder traversal algorithm to print the titles of the
DVDs. Notice that this function outputs only the DVD titles. The definition of this
function is as follows:

void dvdBinaryTree::inorderTitle
    (nodeType<dvdType> *p) const
{
    if (p != NULL)
    {
        inorderTitle(p->lLink);
        p->info.printTitle();
        inorderTitle(p->rLink);
    }
}

The function dvdPrintTitle uses the function inorderTitle to print the titles of
all DVDs in the store. The definition of this function is:

void dvdBinaryTree::dvdPrintTitle() const
{
    inorderTitle(root);
}
The main program is the same as before. Here, we give only the listing of this program. We assume that the name of the header file containing the definition of the class dvdBinaryTree is dvdBinaryTree.h, and so on.

    /******************************************************************************
    * Author: D.S. Malik
    */
    /******************************************************************************

    #include <iostream>
    #include <fstream>
    #include <string>
    #include "binarySearchTree.h"
    #include "dvdType.h"
    #include "dvdBinaryTree.h"

    using namespace std;

    void createDVDList(ifstream& infile, dvdBinaryTree& dvdList);
    void displayMenu();

    int main()
    {
        dvdBinaryTree dvdList;
        int choice;
        string title;

        ifstream infile;

        infile.open("dvdDat.txt");
        if (!infile)
        {
            cout << "The input file does not exist. "
             << "Program terminates!!" << endl;
            return 1;
        }

        createDVDList(infile, dvdList);
        infile.close();

        displayMenu(); //show the menu
        cout << "Enter your choice: ";
        cin >> choice; //get the request
        cin.ignore(100, '\n'); //ignore the remaining
                        //characters in the line

        cout << endl;

        while (choice != 9)
```cpp
{
    switch (choice)
    {
    case 1:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (dvdList.dvdSearch(title))
            cout << "The store carries " << title << endl;
        else
            cout << "The store does not carry " << title << endl;
        break;

    case 2:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (dvdList.dvdSearch(title))
        {
            if (dvdList.isDVDAvailable(title))
            {
                dvdList.dvdCheckOut(title);
                cout << "Enjoy your movie: " << title << endl;
            }
            else
                cout << "Currently " << title << " is out of stock." << endl;
        }
        else
            cout << "The store does not carry " << title << endl;
        break;

    case 3:
        cout << "Enter the title: ";
        getline(cin, title);
        cout << endl;

        if (dvdList.dvdSearch(title))
        {
            dvdList.dvdCheckIn(title);
            cout << "Thanks for returning " << title << endl;
        }
        else
            cout << "The store does not carry " << title << endl;
        break;
```
case 4:
    cout << "Enter the title: ";
    getline(cin, title);
    cout << endl;

    if (dvdList.dvdSearch(title))
    {
        if (dvdList.isDVDAvailable(title))
            cout << title << " is currently in " << "stock." << endl;
        else
            cout << title << " is currently out " << "of stock." << endl;
    }
    else
        cout << "The store does not carry " << title << endl;
    break;

case 5:
    dvdList.dvdPrintTitle();
    break;

case 6:
    dvdList.inorderTraversal();
    break;

default: cout << "Invalid selection." << endl;
} //end switch

displayMenu(); //display the menu
cout << "Enter your choice: ";
cin >> choice; //get the next request
cin.ignore(100, '\n'); //ignore the remaining
    //characters in the line

    cout << endl;
} //end while

return 0;
}

void createDVDList(ifstream& infile,
    dvdBinaryTree& dvdList)
{
    string title;
    string star1;
    string star2;
    string producer;
    string director;
    string productionCo;
    int inStock;

    string title;
    string star1;
    string star2;
    string producer;
    string director;
    string productionCo;
A binary tree is either empty or it has a special node called the root node. If the tree is nonempty, the root node has two sets of nodes, called the left and right subtrees, such that the left and right subtrees are also binary trees.

The node of a binary tree has two links in it.

A node in the binary tree is called a leaf if it has no left and right children.

A node $U$ is called the parent of a node $V$ if there is a branch from $U$ to $V$. 

```cpp
dvdType newDVD;

getline(infile, title);

while (infile)
{
    getline(infile, star1);
    getline(infile, star2);
   getline(infile, producer);
    getline(infile, director);
    getline(infile, productionCo);
    infile >> inStock;
    infile.ignore(100, '
');
newDVD.setDVDInfo(title, star1, star2, producer, director, productionCo, inStock);
dvdList.insert(newDVD);
}

//end while

void displayMenu()
{
    cout << "Select one of the following:" << endl;
    cout << "1: To check whether the store carries a " << endl;
    cout << "particular DVD." << endl;
    cout << "2: To check out a DVD." << endl;
    cout << "3: To check in a DVD." << endl;
    cout << "4: To check whether a particular DVD is " << endl;
    cout << "in stock." << endl;
    cout << "5: To print only the titles of all the DVDs." << endl;
    cout << "6: To print a list of all the DVDs." << endl;
    cout << "9: To exit" << endl;
}
5. A path from a node $X$ to a node $Y$ in a binary tree is a sequence of nodes $X_0, X_1, \ldots, X_n$ such that (a) $X = X_0, X_n = Y$, and (b) $X_{i-1}$ is the parent of $X_i$ for all $i = 1, 2, \ldots, n$. That is, there is a branch from $X_0$ to $X_1$, $X_1$ to $X_2$, $\ldots$, $X_{i-1}$ to $X_i$, $\ldots$, $X_{n-1}$ to $X_n$.

6. The length of a path in a binary tree is the number of branches on that path.

7. The level of a node in a binary tree is the number of branches on the path from the root to the node.

8. The level of the root node of a binary tree is 0, and the level of the children of the root node is 1.

9. The height of a binary tree is the number of nodes on the longest path from the root to a leaf.

10. In an inorder traversal, the binary tree is traversed as follows:
   a. Traverse the left subtree.
   b. Visit the node.
   c. Traverse the right subtree.

11. In a preorder traversal, the binary tree is traversed as follows:
   a. Visit the node.
   b. Traverse the left subtree.
   c. Traverse the right subtree.

12. In a postorder traversal, the binary tree is traversed as follows:
   a. Traverse the left subtree.
   b. Traverse the right subtree.
   c. Visit the node.

13. A binary search tree $T$ is either empty or:
   i. $T$ has a special node called the root node;
   ii. $T$ has two sets of nodes, $L_T$ and $R_T$, called the left subtree and the right subtree of $T$, respectively;
   iii. The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree; and
   iv. $L_T$ and $R_T$ are binary search trees.

14. To delete a node from a binary search tree that has both left and right nonempty subtrees, first its immediate predecessor is located, then the predecessor’s info is copied into the node, and finally the predecessor is deleted.
EXERCISES

1. Mark the following statements as true or false.
   a. A binary tree must be nonempty.
   b. The level of the root node is 0.
   c. If a tree has only one node, the height of this tree is 0 because the number of levels is 0.
   d. The inorder traversal of a binary tree always outputs the data in ascending order.

2. There are 14 different binary trees with four nodes. Draw all of them.
   The binary tree of Figure 19-13, is to be used for Exercises 3 through 18.

3. Find $L_A$, the node in the left subtree of $A$.
4. Find $R_A$, the node in the right subtree of $A$.
5. Find $L_B$, the node in the left subtree of $B$.
6. Find $R_B$, the node in the right subtree of $B$.
7. Find $L_E$, the node in the left subtree of $E$.
8. Find the height of the tree with root $A$.
9. Find the height of the tree with root $D$.
10. Find the level of the node $H$.
11. Find the level of the node $F$.
12. Find the number of leaves in the binary tree with root $A$. 
13. Find the number of leaves in the binary tree with root C.
14. List the leaves in the binary tree with root E.
15. List the nodes in the path from node A to node L.
16. List the nodes of this binary tree in an inorder sequence.
17. List the nodes of this binary tree in a preorder sequence.
18. List the nodes of this binary tree in a postorder sequence.

The binary tree of Figure 19-14 is to be used for Exercises 19 through 23. (Note: These exercises are independent of each other.)

![Binary Tree](image)

**FIGURE 19-14** Figure for Exercises 19 to 23

19. List the path from the node with info 68 to the node with info 90.
20. A node with info 58 is to be inserted in the tree. List the nodes that are visited by the function `insert` to insert 58. Redraw the tree after inserting 58.
21. Delete node 60 and redraw the binary tree.
22. Delete nodes 50 and 95 in that order. Redraw the binary tree after each deletion.
23. Delete node 75 and redraw the binary tree.
24. Insert 28, 25, 26, 42, 47, 30, 45, 29, 5 into an initially empty binary search tree. Draw the final binary search tree.
25. Prove that a binary tree with \( n \) nodes has exactly \( n + 1 \) empty subtree (null pointers).
26. Suppose that you are given two sequences of elements corresponding to the inorder sequence and the preorder sequence. Prove that it is possible to reconstruct a unique binary tree.

27. The following lists the nodes in a binary tree in two different orders:
   
   \[ \text{preorder: ABCDEFGHIJKLM} \]
   \[ \text{inorder: CEDFBAHJIKGML} \]

   Draw the binary tree.

28. Given the nodes of a binary tree in the preorder sequence and the postorder sequence, show that it may not be possible to reconstruct a unique binary tree.

   The binary tree of Figure 19-15 is to be used for Exercises 29 and 30.

![Binary Tree Diagram](image)

**FIGURE 19-15** Figure for Exercises 29 and 30

29. Recall the nonrecursive inorder traversal algorithm for a binary tree given in this chapter. Do an inorder traversal of the binary tree in Figure 19-15. Show the stack contents after each push and pop operation.

30. Recall the nonrecursive preorder traversal algorithm for a binary tree given in this chapter. Do a preorder traversal of the binary tree in Figure 19-15. Show the stack contents after each push and pop operation.

31. Draw the UML class diagram of the `class binaryTreeType`.

32. Draw the UML class diagram of the `class bSearchTreeType`. Also, show the inheritance hierarchy.

**PROGRAMMING EXERCISES**

1. Write the definition of the function, `nodeCount`, that returns the number of nodes in the binary tree. Add this function to the `class binaryTreeType` and create a program to test this function.
2. Write the definition of the function, **leavesCount**, that takes as a parameter a pointer to the root node of a binary tree and returns the number of leaves in a binary tree. Add this function to the class **binaryTreeType** and create a program to test this function.

3. Write a function, **swapSubtrees**, that swaps all of the left and right subtrees of a binary tree. Add this function to the class **binaryTreeType** and create a program to test this function.

4. Write a function, **singleParent**, that returns the number of nodes in a binary tree that have only one child. Add this function to the class **binaryTreeType** and create a program to test this function. (Note: First create a binary search tree.)

5. Write a program to test various operations on a binary search tree.

6. a. Write the definition of the function to implement the nonrecursive postorder traversal algorithm.

   b. Write a program to test the nonrecursive inorder, preorder, and postorder traversal algorithms. (Note: First create a binary search tree.)

7. Write a version of the preorder traversal algorithm in which a user-defined function can be passed as a parameter to specify the visiting criteria at a node. Also, write a program to test your function.

8. Write a version of the postorder traversal algorithm in which a user-defined function can be passed as a parameter to specify the visiting criteria at a node. Also, write a program to test your function.

9. Write a function that inserts the nodes of a binary tree into an ordered linked list. Also write a program to test your function.

10. Write a program to do the following:

    a. Build a binary search tree, \( T_1 \).

    b. Do a postorder traversal of \( T_1 \) and, while doing the postorder traversal, insert the nodes into a second binary search tree \( T_2 \).

    c. Do a preorder traversal of \( T_2 \) and, while doing the preorder traversal, insert the node into a third binary search tree \( T_3 \).

    d. Do an inorder traversal of \( T_3 \).

    e. Output the heights and the number of leafs in each of the three binary search trees.

11. (DVD Store Program) In Programming Exercise 14 in Chapter 16, you were asked to design and implement a class to maintain customer data in a linked list. Because the search on a linked list is sequential and, therefore, can be time consuming, design and implement the class **customerBTreeType** so that this customer data can be stored in a binary search tree. The class **customerBTreeType** must be derived from the class **bSearchTreeType**, as designed in this chapter. (To output the number of DVDs rented by a customer, write the definition of the function **nodeCount**, as in Programming Exercise 1, of the class **binaryTreeType**.)
12. **(DVD Store Program)** Using classes to implement the DVD data, DVD list data, customer data, and customer list data, as designed in this chapter and in Programming Exercise 11, design and complete the program to put the DVD store into operation. (To output the number of DVDs rented by a customer, write the definition of the function `nodeCount`, as in Programming Exercise 1, of the class `binaryTreeType`.)
