CHAPTER

20 Binary Trees

TOPICS

20.1 Definition and Applications of Binary Trees

20.2 Binary Search Tree Operations

20.3 Template Considerations for Binary Search Trees

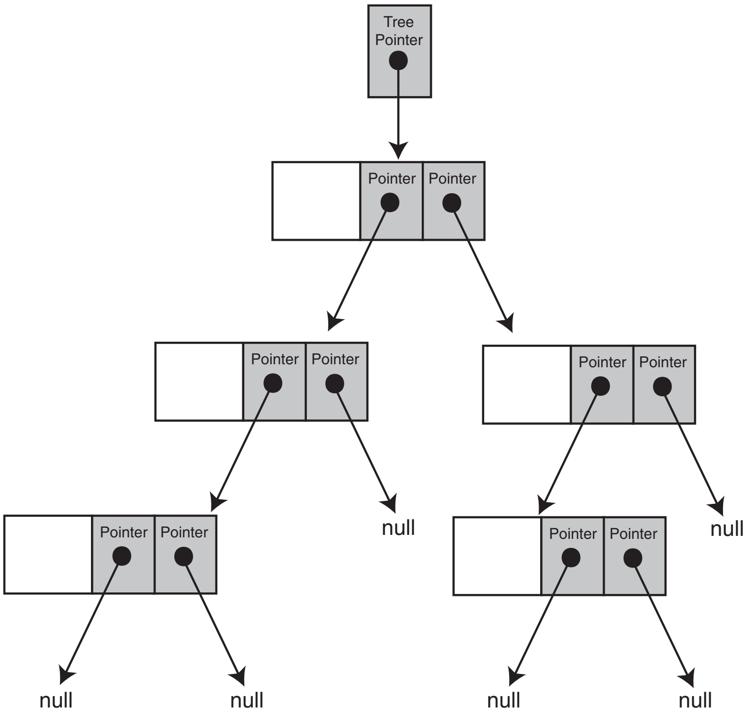
20.1 Definition and Applications of Binary Trees

CONCEPT: A binary tree is a nonlinear linked structure in which each node may point to two other nodes, and every node but the root node has a single predecessor. Binary trees expedite the process of searching large sets of data.

A standard linked list is a linear data structure in which one node is linked to the next. A *binary tree* is a nonlinear linked structure. It is nonlinear because each node can point to two other nodes. Figure 20-1 illustrates the organization of a binary tree.

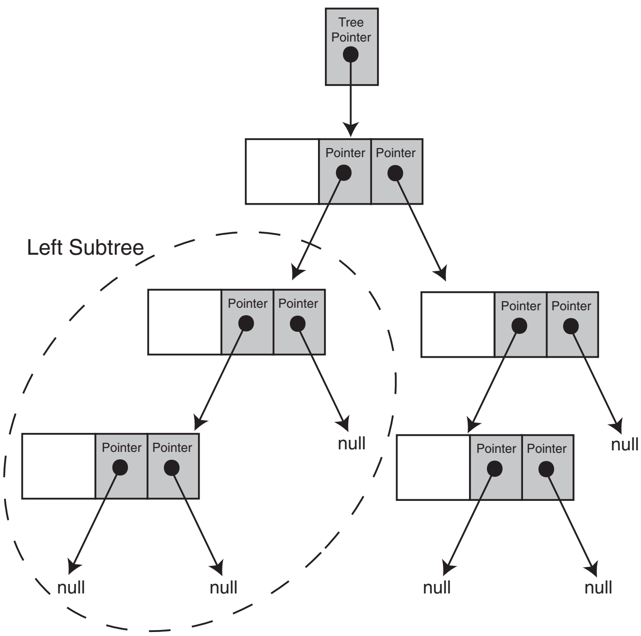
The data structure is called a tree because it resembles an upside-down tree. It is anchored at the top by a *tree pointer*, which is like the head pointer in a standard linked list. The first node in the list is called the *root node*. The root node has pointers to two other nodes, which are called *children*, or *child nodes*. Each of the children has its own set of two pointers and can have its own children. Notice that not all nodes have two children. Some point to only one node, and some point to no other nodes. A node that has no children is called a *leaf node*. All pointers that do not point to a node are set to `nullptr`.

Figure 20-1



Binary trees can be divided into *subtrees*. A subtree is an entire branch of the tree, from one particular node down. For example, Figure 20-2 shows the left subtree from the root node of the tree shown in Figure 20-1.

Figure 20-2

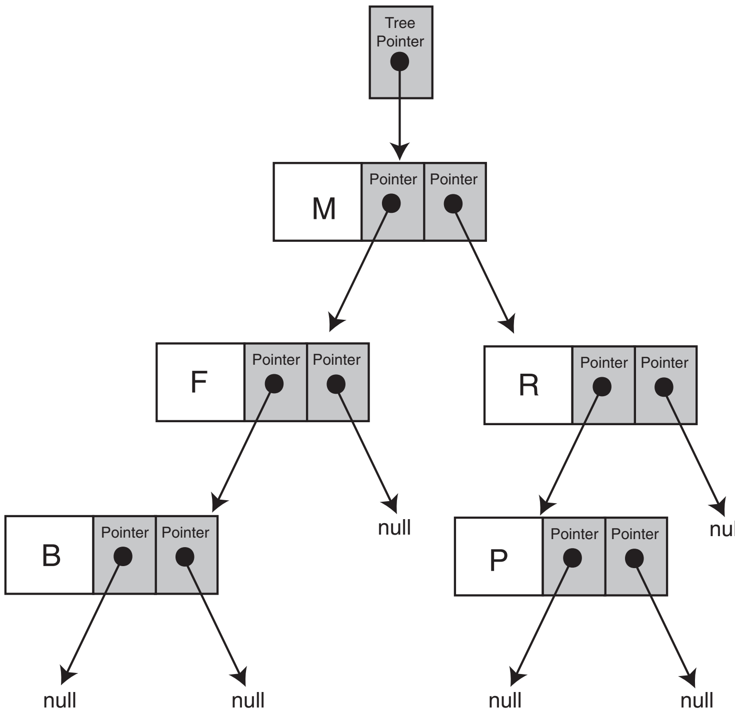


Applications of Binary Trees

Searching any linear data structure, such as an array or a standard linked list, is slow when the structure holds a large amount of data. This is because of the sequential nature of linear data structures. Binary trees are excellent data structures for searching large amounts of data. They are commonly used in database applications to organize key values that index database records. When used to facilitate searches, a binary tree is called a *binary search tree*. Binary search trees are the primary focus of this chapter.

Data are stored in binary search trees in a way that makes a binary search simple. For example, look at Figure 20-3.

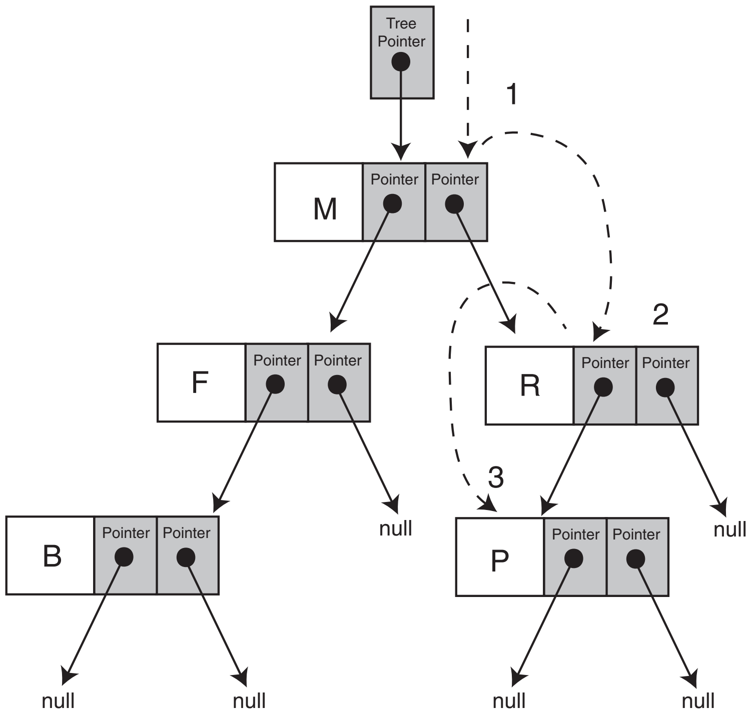
Figure 20-3



The figure depicts a binary search tree where each node stores a letter of the alphabet. Notice that the root node holds the letter M. The left child of the root node holds the letter F, and the right child holds R. Values are stored in a binary search tree so that a node's left child holds data whose value is less than the node's data, and the node's right child holds data whose value is greater than the node's data. This is true for all nodes in the tree that have children.

It is also true that *all* the nodes to the left of a node hold values less than the node's value. Likewise, all the nodes to the right of a node hold values that are greater than the node's data. When an application is searching a binary tree, it starts at the root node. If the root node does not hold the search value, the application branches either to the left or right child, depending on whether the search value is less than or greater than the value at the root node. This process continues until the value is found. Figure 20-4 illustrates the search pattern for finding the letter P in the binary tree shown.

Figure 20-4



Checkpoint

20.1 Describe the difference between a binary tree and a standard linked list.

20.2 What is a root node?

20.3 What is a child node?

20.4 What is a leaf node?

20.5 What is a subtree?

20.6 Why are binary trees suitable for algorithms that must search large amounts of data?

20.2 Binary Search Tree Operations

CONCEPT: There are many operations that may be performed on a binary search tree. In this section we will discuss creating a binary search tree and inserting, finding, and deleting nodes.

In this section you will learn some basic operations that may be performed on a binary search tree. We will study a simple class that implements a binary tree for storing integer values.

Creating a Binary Tree

We will demonstrate the fundamental binary tree operations using a simple ADT: the `IntBinaryTree` class. The basis of our binary tree node is the following `struct` declaration:

```
struct TreeNode
{
    int value;
    TreeNode *left;
    TreeNode *right;
};
```

Each node has a `value` member for storing its integer data, as well as `left` and `right` pointers. The `struct` is implemented in the class declaration shown here:

Contents of `IntBinaryTree.h`

```
// Specification file for the IntBinaryTree class
#ifndef INTBINARYTREE_H
#define INTBINARYTREE_H

class IntBinaryTree
{
private:
    struct TreeNode
    {
        int value;          // The value in the node
        TreeNode *left;     // Pointer to left child node
        TreeNode *right;    // Pointer to right child node
    };

    TreeNode *root;         // Pointer to the root node

    // Private member functions
    void insert(TreeNode *&, TreeNode *&);
    void destroySubTree(TreeNode *);
    void deleteNode(int, TreeNode *&);
    void makeDeletion(TreeNode *&);
    void displayInOrder(TreeNode *) const;
    void displayPreOrder(TreeNode *) const;
    void displayPostOrder(TreeNode *) const;

public:
    // Constructor
    IntBinaryTree()
        { root = nullptr; }

    // Destructor
    ~IntBinaryTree()
        { destroySubTree(root); }

    // Binary tree operations
    void insertNode(int);
    bool searchNode(int);
    void remove(int);

```

```
        void displayInOrder() const
            { displayInOrder(root); }

        void displayPreOrder() const
            { displayPreOrder(root); }

        void displayPostOrder() const
            { displayPostOrder(root); }
};
#endif
```

The `root` pointer will be used as the tree pointer. Similar to the head pointer in a linked list, root will point to the first node in the tree, or to `nullptr` if the tree is empty. It is initialized in the constructor, which is declared inline. The destructor calls `destroySubTree`, a private member function that recursively deletes all the nodes in the tree.

Inserting a Node

The code to insert a node into the tree is fairly straightforward. The public member function `insertNode` is called with the number to be inserted passed as an argument. The code for the function, which is in `IntBinaryTree.cpp`, is shown here:

VideoNote
Inserting a Node in a Binary Tree

```
27 void IntBinaryTree::insertNode(int num)
28 {
29     TreeNode *newNode = nullptr;     // Pointer to a new node.
30 
31     // Create a new node and store num in it.
32     newNode = new TreeNode;
33     newNode->value = num;
34     newNode->left = newNode->right = nullptr;
35 
36     // Insert the node.
37     insert(root, newNode);
38 }
```

First, a new node is allocated in line 32 and its address stored in the local pointer variable `newNode`. The value passed as an argument is stored in the node's `value` member in line 33. The node's `left` and `right` child pointers are set to `nullptr` in line 34 because all nodes must be inserted as leaf nodes. Next, the private member function `insert` is called in line 37. Notice that the `root` pointer and the `newNode` pointer are passed as arguments. The code for the `insert` function is shown here:

```
void IntBinaryTree::insert(TreeNode *&nodePtr, TreeNode *&newNode)
{
   if (nodePtr == nullptr)
      nodePtr = newNode;                // Insert the node.
   else if (newNode->value < nodePtr->value)
      insert(nodePtr->left, newNode); // Search the left branch.
   else
      insert(nodePtr->right, newNode); // Search the right branch.
}
```

In general, this function takes a pointer to a subtree and a pointer to a new node as arguments. It searches for the appropriate location in the subtree to insert the node, and then makes the insertion. Notice the declaration of the first parameter, `nodePtr`:

```
TreeNode *&nodePtr
```

The `nodePtr` parameter is not simply a pointer to a `TreeNode` structure, but a *reference* to a pointer to a `TreeNode` structure. This means that any action performed on `nodePtr` is actually performed on the argument that was passed into `nodePtr`. The reason for this will be explained momentarily.

The `if` statement in line 14 determines whether `nodePtr` is set to `nullptr`:

```
if (nodePtr == nullptr)
    nodePtr = newNode;  // Insert the node.
```

If `nodePtr` is set to `nullptr`, it is at the end of a branch and the insertion point has been found. `nodePtr` is then made to point to `newNode`, which inserts `newNode` into the tree. This is why the `nodePtr` parameter is a reference. If it weren't a reference, this function would be making a copy of a node point to the new node, not the actual node in the tree.

If `nodePtr` is not set to `nullptr`, the following `else if` statement in line 16 executes:

```
else if (newNode->value < nodePtr->value)
    insert(nodePtr->left, newNode);  // Search the left branch.
```

If the new node's value is less than the value pointed to by `nodePtr`, the insertion point is somewhere in the left subtree. If this is the case, the `insert` function is recursively called in line 17 with `nodePtr->left` passed as the subtree argument.

If the new node's value is not less than the value pointed to by `nodePtr`, the `else` statement in line 18 executes:

```
else
    insert(nodePtr->right, newNode);  // Search the right branch.
```

The `else` statement recursively calls the `insert` function called with `nodePtr->right` passed as the subtree argument.

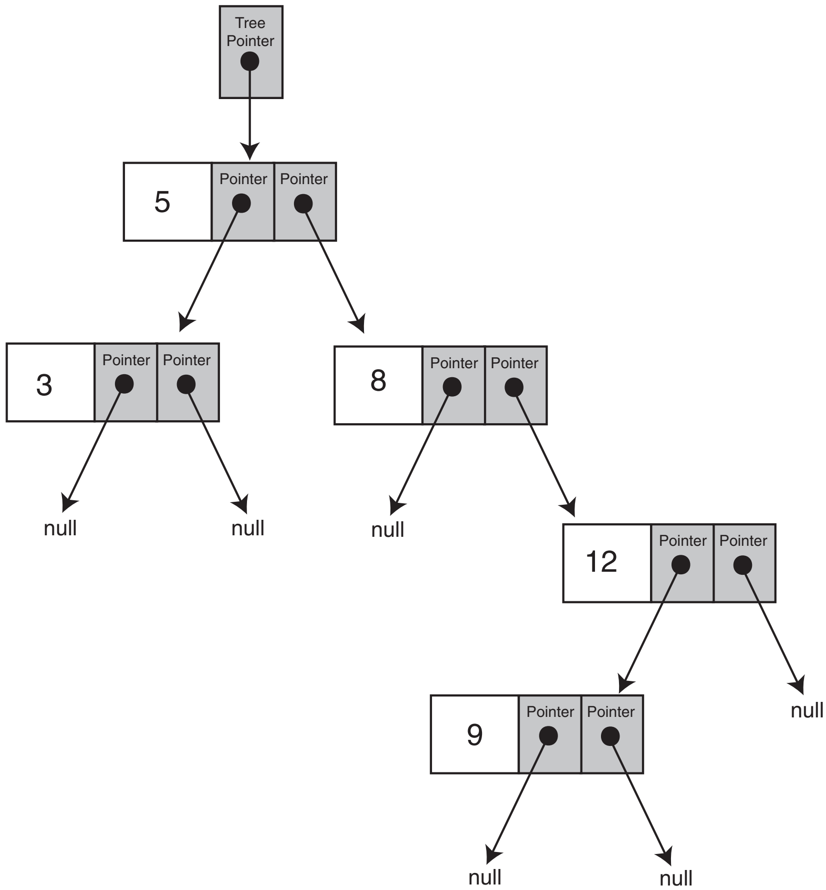
Program 20-1 demonstrates these functions.

Program 20-1

```
// This program builds a binary tree with 5 nodes.
#include <iostream>
#include "IntBinaryTree.h"
using namespace std;

int main()
{
    IntBinaryTree tree;

    cout << "Inserting nodes. ";
    tree.insertNode(5);
    tree.insertNode(8);
    tree.insertNode(3);
    tree.insertNode(12);
    tree.insertNode(9);
    cout << "Done.\n";

    return 0;
}
```

Figure 20-5 shows the structure of the binary tree built by Program 20-1.

Figure 20-5



> **NOTE:** The shape of the tree is determined by the order in which the values are inserted. The root node in the diagram above holds the value 5 because that was the first value inserted. By stepping through the function, you can see how the other nodes came to appear in their depicted positions.

> **NOTE:** If the new value being inserted into the tree is equal to an existing value, the insertion algorithm inserts it to the right of the existing value.

Traversing the Tree

There are three common methods for traversing a binary tree and processing the value of each node: *inorder*, *preorder*, and *postorder*. Each of these methods is best implemented as a recursive function. The algorithms are described as follows:

- ***Inorder traversal***
  1. The current node's left subtree is traversed.
  2. The current node's data is processed.
  3. The current node's right subtree is traversed.
- ***Preorder traversal***
  1. The current node's data is processed
  2. The current node's left subtree is traversed.
  3. The current node's right subtree is traversed.
- ***Postorder traversal***
  1. The current node's left subtree is traversed.
  2. The current node's right subtree is traversed.
  3. The current node's data is processed.

The `IntBinaryTree` class can display all the values in the tree using all three of these algorithms. The algorithms are initiated by the following inline public member functions:

```
void displayInOrder() const
   { displayInOrder(root); }

void displayPreOrder() const
   { displayPreOrder(root); }

void displayPostOrder() const
   { displayPostOrder(root); }
```

Each of the public member functions calls an overloaded recursive private member function and passes the root pointer as an argument. The recursive functions, which are very simple and straightforward, are shown here:

```
//**************************************************************
// The displayInOrder member function displays the values     *
// in the subtree pointed to by nodePtr, via inorder traversal. *
//**************************************************************

void IntBinaryTree::displayInOrder(TreeNode *nodePtr) const
{
     if (nodePtr)
     {
          displayInOrder(nodePtr->left);
          cout << nodePtr->value << endl;
          displayInOrder(nodePtr->right);
     }
}

//**************************************************************
// The displayPreOrder member function displays the values    *
// in the subtree pointed to by nodePtr, via preorder traversal. *
//**************************************************************

void IntBinaryTree::displayPreOrder(TreeNode *nodePtr) const
{
     if (nodePtr)
     {
          cout << nodePtr->value << endl;
          displayPreOrder(nodePtr->left);
```

```
            displayPreOrder(nodePtr->right);
        }
    }

    //*********************************************************
    // The displayPostOrder member function displays the values   *
    // in the subtree pointed to by nodePtr, via postorder traversal. *
    //*********************************************************

    void IntBinaryTree::displayPostOrder(TreeNode *nodePtr) const
    {
        if (nodePtr)
        {
            displayPostOrder(nodePtr->left);
            displayPostOrder(nodePtr->right);
            cout << nodePtr->value << endl;
        }
    }
```

Program 20-2, which is a modification of Program 20-1, demonstrates each of the traversal methods.

Program 20-2

```
// This program builds a binary tree with 5 nodes.
// The nodes are displayed with inorder, preorder,
// and postorder algorithms.
#include <iostream>
#include "IntBinaryTree.h"
using namespace std;

int main()
{
    IntBinaryTree tree;

    // Insert some nodes.
    cout << "Inserting nodes.\n";
    tree.insertNode(5);
    tree.insertNode(8);
    tree.insertNode(3);
    tree.insertNode(12);
    tree.insertNode(9);

    // Display inorder.
    cout << "Inorder traversal:\n";
    tree.displayInOrder();

    // Display preorder.
    cout << "\nPreorder traversal:\n";
    tree.displayPreOrder();

    // Display postorder.
    cout << "\nPostorder traversal:\n";
    tree.displayPostOrder();
    return 0;
}
```

Program Output

```
Inserting nodes.
Inorder traversal:
3
5
8
9
12

Preorder traversal:
5
3
8
12
9

Postorder traversal:
3
9
12
8
5
```

Searching the Tree

The `IntBinaryTree` class has a public member function, `searchNode`, that returns `true` if a value is found in the tree, or `false` otherwise. The function simply starts at the `root` node and traverses the tree until it finds the search value or runs out of nodes. The code is shown here.

```
bool IntBinaryTree::searchNode(int num)
{
    TreeNode *nodePtr = root;

    while (nodePtr)
    {
        if (nodePtr->value == num)
            return true;
        else if (num < nodePtr->value)
            nodePtr = nodePtr->left;
        else
            nodePtr = nodePtr->right;
    }
    return false;
}
```

Program 20-3 demonstrates this function.

Program 20-3

```
// This program builds a binary tree with 5 nodes.
// The SearchNode function is demonstrated.
#include <iostream>
#include "IntBinaryTree.h"
using namespace std;

int main()
{
    IntBinaryTree tree;

    // Insert some nodes in the tree.
    cout << "Inserting nodes.\n";
    tree.insertNode(5);
    tree.insertNode(8);
    tree.insertNode(3);
    tree.insertNode(12);
    tree.insertNode(9);

    // Search for the value 3.
    if (tree.searchNode(3))
        cout << "3 is found in the tree.\n";
    else
        cout << "3 was not found in the tree.\n";

    // Search for the value 100.
    if (tree.searchNode(100))
        cout << "100 is found in the tree.\n";
    else
        cout << "100 was not found in the tree.\n";
    return 0;
}
```

Program Output

```
Inserting nodes.
3 is found in the tree.
100 was not found in the tree.
```

Deleting a Node

VideoNote
Deleting a Node from a Binary Tree

Deleting a leaf node is not difficult. We simply find its parent and set the child pointer that links to it to `nullptr`, then free the node's memory. But what if we want to delete a node that has child nodes? We must delete the node while at the same time preserving the subtrees that the node links to.

There are two possible situations to face when deleting a nonleaf node: the node has one child, or the node has two children. Figure 20-6 illustrates a tree in which we are about to delete a node with one subtree.

Figure 20-6

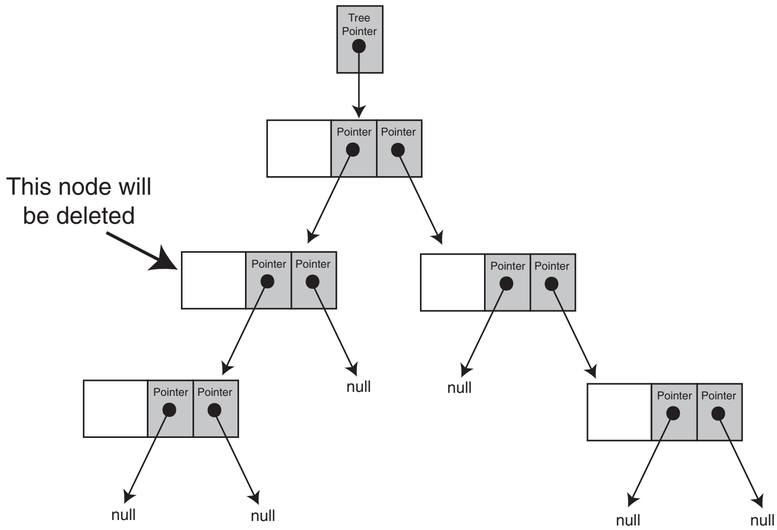
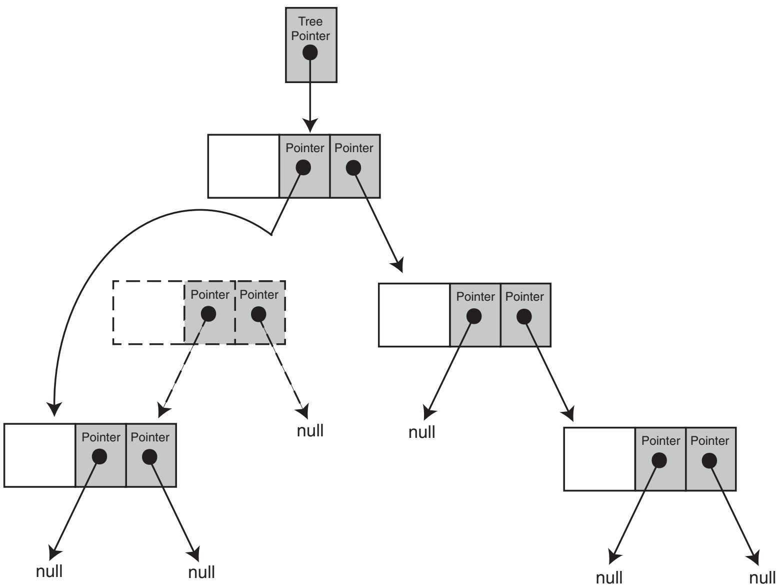


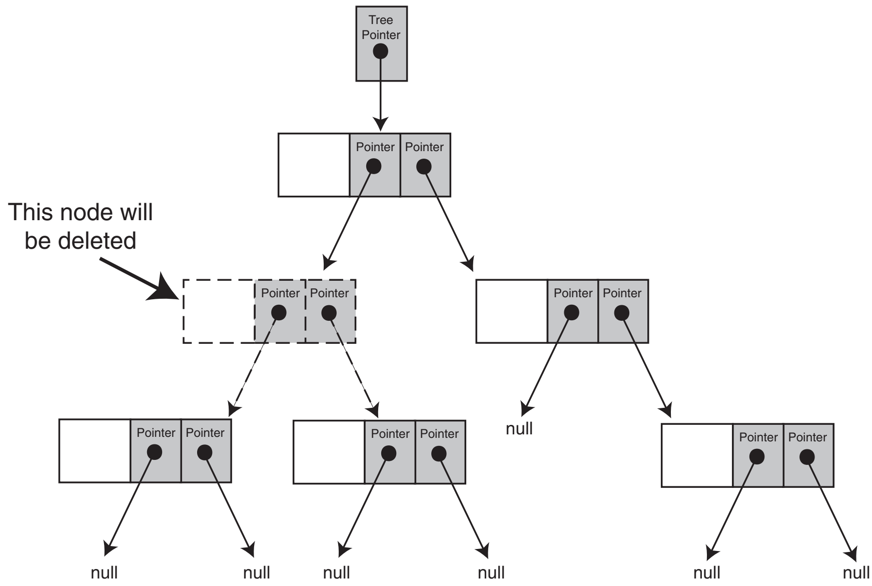
Figure 20-7 shows how we will link the node's subtree with its parent:

Figure 20-7



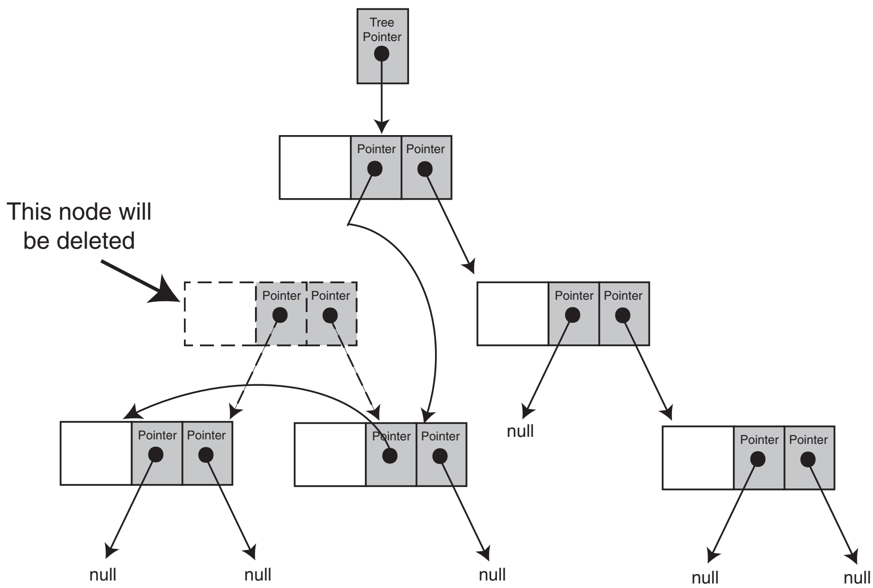
The problem is not as easily solved, however, when the node we are about to delete has two subtrees. For example, look at Figure 20-8:

Figure 20-8



Obviously, we cannot attach both of the node's subtrees to its parent, so there must be an alternative solution. One way of addressing this problem is to attach the node's right subtree to the parent, then find a position in the right subtree to attach the left subtree. The result is shown in Figure 20-9.

Figure 20-9



Now we will see how this action is implemented in code. To delete a node from the `IntBinaryTree`, call the public member `remove`. The argument passed to the function is the value of the node you wish to delete. The `remove` member function is shown here:

```
void IntBinaryTree::remove(int num)
{
    deleteNode(num, root);
}
```

The `remove` member function calls the `deleteNode` member function. It passes the value of the node to delete and the `root` pointer. The `deleteNode` member function is shown here:

```
 95  void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
 96  {
 97      if (num < nodePtr->value)
 98          deleteNode(num, nodePtr->left);
 99      else if (num > nodePtr->value)
100          deleteNode(num, nodePtr->right);
101      else
102          makeDeletion(nodePtr);
103  }
```

Notice that this function's arguments are references to pointers. Like the `insert` function, the `deleteNode` function must have access to an actual pointer in the tree. You will see why momentarily.

The `deleteNode` function uses an `if/else if` statement. The first part of the statement is in lines 97 and 98:

```
if (num < nodePtr->value)
    deleteNode(num, nodePtr->left);
```

This code compares the parameter `num` with the `value` member of the node that `nodePtr` points to. If `num` is less, then the value being searched for will appear somewhere in `nodePtr`'s left subtree (if it appears in the tree at all). In this case, the `deleteNode` function is recursively called with `num` as the first argument and `nodePtr->left` as the second argument.

If `num` is not less than `nodePtr->value`, the `else if` in lines 99 and 100 statement is executed:

```
else if (num > nodePtr->value)
    deleteNode(num, nodePtr->right);
```

If `num` is greater than `nodePtr->value`, then the value being searched for will appear somewhere in `nodePtr`'s right subtree (if it appears in the tree at all). So, the `deleteNode` function is recursively called with `num` as the first argument and `nodePtr->right` as the second argument.

If `num` is equal to `nodePtr->value`, then neither of the `if` statements will find a true condition. In this case, `nodePtr` points to the node that is to be deleted, and the trailing `else` in lines 101 and 102 will execute:

```
else
    makeDeletion(nodePtr);
```

The trailing `else` statement calls the `makeDeletion` function and passes `nodePtr` as its argument. The `makeDeletion` function actually deletes the node from the tree and must reattach the deleted node's subtrees as shown in Figure 20-9. Therefore, it must have access to the actual pointer, in the binary tree, to the node that is being deleted (not just a copy of the pointer). This is why the `nodePtr` parameter in the `deleteNode` function is a reference. It must pass to `makeDeletion` the actual pointer, in the binary tree, to the node that is to be deleted. The `makeDeletion` function's code is as follows:

```
void IntBinaryTree::makeDeletion(TreeNode *&nodePtr)
{
    // Define a temporary pointer to use in reattaching
    // the left subtree.
    TreeNode *tempNodePtr = nullptr;

    if (nodePtr == nullptr)
        cout << "Cannot delete empty node.\n";
    else if (nodePtr->right == nullptr)
    {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->left;  // Reattach the left child.
        delete tempNodePtr;
    }
    else if (nodePtr->left == nullptr)
    {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->right;  // Reattach the right child.
        delete tempNodePtr;
    }
    // If the node has two children.
    else
    {
        // Move one node to the right.
        tempNodePtr = nodePtr->right;
        // Go to the end left node.
        while (tempNodePtr->left)
            tempNodePtr = tempNodePtr->left;
        // Reattach the left subtree.
        tempNodePtr->left = nodePtr->left;
        tempNodePtr = nodePtr;
        // Reattach the right subtree.
        nodePtr = nodePtr->right;
        delete tempNodePtr;
    }
}
```

Program 20-4 demonstrates these functions.

Program 20-4

```
// This program builds a binary tree with 5 nodes.
// The deleteNode function is used to remove two of them.
#include <iostream>
#include "IntBinaryTree.h"
using namespace std;

int main()
{
    IntBinaryTree tree;

    // Insert some values into the tree.
    cout << "Inserting nodes.\n";
    tree.insertNode(5);
    tree.insertNode(8);
    tree.insertNode(3);
    tree.insertNode(12);
    tree.insertNode(9);

    // Display the values.
    cout << "Here are the values in the tree:\n";
    tree.displayInOrder();

    // Delete the value 8.
    cout << "Deleting 8...\n";
    tree.remove(8);

    // Delete the value 12.
    cout << "Deleting 12...\n";
    tree.remove(12);

    // Display the values.
    cout << "Now, here are the nodes:\n";
    tree.displayInOrder();
    return 0;
}
```

Program Output

```
Inserting nodes.
Here are the values in the tree:
3
5
8
9
12
Deleting 8...
Deleting 12...
Now, here are the nodes:
3
5
9
```

For your reference, the entire contents of `IntBinaryTree` file are shown below.

Contents of `IntBinaryTree.cpp`

```
// Implementation file for the IntBinaryTree class
#include <iostream>
#include "IntBinaryTree.h"
using namespace std;

//*************************************************************
// insert accepts a TreeNode pointer and a pointer to a node. *
// The function inserts the node into the tree pointed to by  *
// the TreeNode pointer. This function is called recursively. *
//*************************************************************

void IntBinaryTree::insert(TreeNode *&nodePtr, TreeNode *&newNode)
{
    if (nodePtr == nullptr)
        nodePtr = newNode;             // Insert the node.
    else if (newNode->value < nodePtr->value)
        insert(nodePtr->left, newNode); // Search the left branch.
    else
        insert(nodePtr->right, newNode); // Search the right branch.
}

//**********************************************************
// insertNode creates a new node to hold num as its value, *
// and passes it to the insert function.                   *
//**********************************************************

void IntBinaryTree::insertNode(int num)
{
    TreeNode *newNode = nullptr;   // Pointer to a new node.

    // Create a new node and store num in it.
    newNode = new TreeNode;
    newNode->value = num;
    newNode->left = newNode->right = nullptr;

    // Insert the node.
    insert(root, newNode);
}

//***************************************************
// destroySubTree is called by the destructor. It   *
// deletes all nodes in the tree.                   *
//***************************************************

void IntBinaryTree::destroySubTree(TreeNode *nodePtr)
{
    if (nodePtr)
    {
        if (nodePtr->left)
            destroySubTree(nodePtr->left);
```

```
        if (nodePtr->right)
            destroySubTree(nodePtr->right);
        delete nodePtr;
    }
}

//*****************************************************
// searchNode determines whether a value is present in *
// the tree. If so, the function returns true.         *
// Otherwise, it returns false.                        *
//*****************************************************

bool IntBinaryTree::searchNode(int num)
{
    TreeNode *nodePtr = root;

    while (nodePtr)
    {
        if (nodePtr->value == num)
            return true;
        else if (num < nodePtr->value)
            nodePtr = nodePtr->left;
        else
            nodePtr = nodePtr->right;
    }
    return false;
}

//*********************************************
// remove calls deleteNode to delete the       *
// node whose value member is the same as num. *
//*********************************************

void IntBinaryTree::remove(int num)
{
    deleteNode(num, root);
}


//********************************************
// deleteNode deletes the node whose value   *
// member is the same as num.                *
//********************************************

void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
{
    if (num < nodePtr->value)
        deleteNode(num, nodePtr->left);
    else if (num > nodePtr->value)
        deleteNode(num, nodePtr->right);
    else
        makeDeletion(nodePtr);
}


```

```
//**********************************************************
// makeDeletion takes a reference to a pointer to the node *
// that is to be deleted. The node is removed and the      *
// branches of the tree below the node are reattached.     *
//**********************************************************

void IntBinaryTree::makeDeletion(TreeNode *&nodePtr)
{
     // Define a temporary pointer to use in reattaching
     // the left subtree.
     TreeNode *tempNodePtr = nullptr;

     if (nodePtr == nullptr)
          cout << "Cannot delete empty node.\n";
     else if (nodePtr->right == nullptr)
     {
          tempNodePtr = nodePtr;
          nodePtr = nodePtr->left;  // Reattach the left child.
          delete tempNodePtr;
     }
     else if (nodePtr->left == nullptr)
     {
          tempNodePtr = nodePtr;
          nodePtr = nodePtr->right;  // Reattach the right child.
          delete tempNodePtr;
     }
     // If the node has two children.
     else
     {
          // Move one node to the right.
          tempNodePtr = nodePtr->right;
          // Go to the end left node.
          while (tempNodePtr->left)
             tempNodePtr = tempNodePtr->left;
          // Reattach the left subtree.
          tempNodePtr->left = nodePtr->left;
          tempNodePtr = nodePtr;
          // Reattach the right subtree.
          nodePtr = nodePtr->right;
          delete tempNodePtr;
     }
}

//************************************************************
// The displayInOrder member function displays the values     *
// in the subtree pointed to by nodePtr, via inorder traversal. *
//************************************************************

void IntBinaryTree::displayInOrder(TreeNode *nodePtr) const
{
     if (nodePtr)
     {
          displayInOrder(nodePtr->left);
          cout << nodePtr->value << endl;
          displayInOrder(nodePtr->right);
     }
}
```

```

//**************************************************************
// The displayPreOrder member function displays the values    *
// in the subtree pointed to by nodePtr, via preorder traversal. *
//**************************************************************

void IntBinaryTree::displayPreOrder(TreeNode *nodePtr) const
{
    if (nodePtr)
    {
        cout << nodePtr->value << endl;
        displayPreOrder(nodePtr->left);
        displayPreOrder(nodePtr->right);
    }
}

//***************************************************************
// The displayPostOrder member function displays the values    *
// in the subtree pointed to by nodePtr, via postorder traversal. *
//***************************************************************

void IntBinaryTree::displayPostOrder(TreeNode *nodePtr) const
{
    if (nodePtr)
    {
        displayPostOrder(nodePtr->left);
        displayPostOrder(nodePtr->right);
        cout << nodePtr->value << endl;
    }
}
```

Checkpoint

20.7 Describe the sequence of events in an inorder traversal.

20.8 Describe the sequence of events in a preorder traversal.

20.9 Describe the sequence of events in a postorder traversal.

20.10 Describe the steps taken in deleting a leaf node.

20.11 Describe the steps taken in deleting a node with one child.

20.12 Describe the steps taken in deleting a node with two children.

20.3 Template Considerations for Binary Search Trees

CONCEPT: Binary search trees may be implemented as templates, but any data types used with them must support the <, >, and == operators.

When designing a binary tree template, remember that any data types stored in the binary tree must support the <, >, and == operators. If you use the tree to store class objects, these operators must be overridden.

The following code shows an example of a binary tree template. Program 20-5 demonstrates the template. It creates a binary tree that can hold strings and then prompts the user to enter a series of names that are inserted into the tree. The program then displays the contents of the tree using inorder traversal.

Contents of `BinaryTree.h`

```
#ifndef BINARYTREE_H
#define BINARYTREE_H
#include <iostream>
using namespace std;

// BinaryTree template
template <class T>
class BinaryTree
{
private:
    struct TreeNode
    {
        T value;            // The value in the node
        TreeNode *left;     // Pointer to left child node
        TreeNode *right;    // Pointer to right child node
    };

    TreeNode *root; // Pointer to the root node

    // Private member functions
    void insert(TreeNode *&, TreeNode *&);
    void destroySubTree(TreeNode *);
    void deleteNode(T, TreeNode *&);
    void makeDeletion(TreeNode *&);
    void displayInOrder(TreeNode *) const;
    void displayPreOrder(TreeNode *) const;
    void displayPostOrder(TreeNode *) const;

public:
    // Constructor
    BinaryTree()
        { root = nullptr; }

    // Destructor
    ~BinaryTree()
        { destroySubTree(root); }

    // Binary tree operations
    void insertNode(T);
    bool searchNode(T);
    void remove(T);

    void displayInOrder() const
        { displayInOrder(root); }

    void displayPreOrder() const
        { displayPreOrder(root); }

```

```
    void displayPostOrder() const
        { displayPostOrder(root); }
};

//***********************************************************
// insert accepts a TreeNode pointer and a pointer to a node. *
// The function inserts the node into the tree pointed to by  *
// the TreeNode pointer. This function is called recursively. *
//***********************************************************
template <class T>
void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode *&newNode)
{
    if (nodePtr == nullptr)
        nodePtr = newNode;              // Insert the node
    else if (newNode->value < nodePtr->value)
        insert(nodePtr->left, newNode); // Search the left branch
    else
        insert(nodePtr->right, newNode); // Search the right branch
}

//*********************************************************
// insertNode creates a new node to hold num as its value, *
// and passes it to the insert function.                   *
//*********************************************************
template <class T>
void BinaryTree<T>::insertNode(T item)
{
    TreeNode *newNode = nullptr;  // Pointer to a new node

    // Create a new node and store num in it.
    newNode = new TreeNode;
    newNode->value = item;
    newNode->left = newNode->right = nullptr;

    // Insert the node.
    insert(root, newNode);
}

//**************************************************
// destroySubTree is called by the destructor. It   *
// deletes all nodes in the tree.                   *
//**************************************************
template <class T>
void BinaryTree<T>::destroySubTree(TreeNode *nodePtr)
{
    if (nodePtr)
    {
        if (nodePtr->left)
            destroySubTree(nodePtr->left);
        if (nodePtr->right)
            destroySubTree(nodePtr->right);
        delete nodePtr;
    }
}

```

```
//***************************************************
// searchNode determines if a value is present in   *
// the tree. If so, the function returns true.      *
// Otherwise, it returns false.                     *
//***************************************************
template <class T>
bool BinaryTree<T>::searchNode(T item)
{
    TreeNode *nodePtr = root;

    while (nodePtr)
    {
        if (nodePtr->value == item)
            return true;
        else if (item < nodePtr->value)
            nodePtr = nodePtr->left;
        else
            nodePtr = nodePtr->right;
    }
    return false;
}

//***********************************************
// remove calls deleteNode to delete the       *
// node whose value member is the same as num. *
//***********************************************
template <class T>
void BinaryTree<T>::remove(T item)
{
    deleteNode(item, root);
}

//********************************************
// deleteNode deletes the node whose value   *
// member is the same as num.                *
//********************************************
template <class T>
void BinaryTree<T>::deleteNode(T item, TreeNode *&nodePtr)
{
    if (item < nodePtr->value)
        deleteNode(item, nodePtr->left);
    else if (item > nodePtr->value)
        deleteNode(item, nodePtr->right);
    else
        makeDeletion(nodePtr);
}

//***********************************************************
// makeDeletion takes a reference to a pointer to the node *
// that is to be deleted. The node is removed and the      *
// branches of the tree below the node are reattached.     *
//***********************************************************
template <class T>
void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr)
{
```

```
    // Define a temporary pointer to use in reattaching
    // the left subtree.
    TreeNode *tempNodePtr = nullptr;

    if (nodePtr == nullptr)
        cout << "Cannot delete empty node.\n";
    else if (nodePtr->right == nullptr)
    {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->left;  // Reattach the left child
        delete tempNodePtr;
    }
    else if (nodePtr->left == nullptr)
    {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->right;  // Reattach the right child
        delete tempNodePtr;
    }
    // If the node has two children.
    else
    {
        // Move one node to the right.
        tempNodePtr = nodePtr->right;
        // Go to the end left node.
        while (tempNodePtr->left)
            tempNodePtr = tempNodePtr->left;
        // Reattach the left subtree.
        tempNodePtr->left = nodePtr->left;
        tempNodePtr = nodePtr;
        // Reattach the right subtree.
        nodePtr = nodePtr->right;
        delete tempNodePtr;
    }
}

//**************************************************************
// The displayInOrder member function displays the values      *
// in the subtree pointed to by nodePtr, via inorder traversal. *
//**************************************************************
template <class T>
void BinaryTree<T>::displayInOrder(TreeNode *nodePtr) const
{
    if (nodePtr)
    {
        displayInOrder(nodePtr->left);
        cout << nodePtr->value << endl;
        displayInOrder(nodePtr->right);
    }
}

//**************************************************************
// The displayPreOrder member function displays the values      *
// in the subtree pointed to by nodePtr, via preorder traversal. *
//**************************************************************
template <class T>
void BinaryTree<T>::displayPreOrder(TreeNode *nodePtr) const
```

```
{
    if (nodePtr)
    {
        cout << nodePtr->value << endl;
        displayPreOrder(nodePtr->left);
        displayPreOrder(nodePtr->right);
    }
}

//**************************************************************
// The displayPostOrder member function displays the values   *
// in the subtree pointed to by nodePtr, via postorder traversal.*
//**************************************************************
template <class T>
void BinaryTree<T>::displayPostOrder(TreeNode *nodePtr) const
{
    if (nodePtr)
    {
        displayPostOrder(nodePtr->left);
        displayPostOrder(nodePtr->right);
        cout << nodePtr->value << endl;
    }
}
#endif
```

Program 20-5

```
// This program demonstrates the BinaryTree class template.
// It builds a binary tree with 5 nodes.
#include <iostream>
#include "BinaryTree.h"
using namespace std;

const int NUM_NODES = 5;

int main()
{
    string name;

    // Create the binary tree.
    BinaryTree<string> tree;

    // Insert some names.
    for (int count = 0; count < NUM_NODES; count++)
    {
        cout << "Enter a name: ";
        getline(cin, name);
        tree.insertNode(name);
    }

    // Display the values.
    cout << "\nHere are the values in the tree:\n";
    tree.displayInOrder();
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter a name: David [Enter]
Enter a name: Geri [Enter]
Enter a name: Chris [Enter]
Enter a name: Samantha [Enter]
Enter a name: Anthony [Enter]

Here are the values in the tree:
Anthony
Chris
David
Geri
Samantha
```

Review Questions and Exercises

Short Answer

1. Each node in a binary tree may point to how many other nodes?
2. How many predecessors may each node other than the root node have?
3. What is a leaf node?
4. What is a subtree?
5. What initially determines the shape of a binary tree?
6. What are the three methods of traversing a binary tree? What is the difference between these methods?

Fill-in-the-Blank

7. The first node in a binary tree is called the __________.
8. A binary tree node's left and right pointers point to the node's __________.
9. A node with no children is called a(n) __________.
10. A(n) __________ is an entire branch of the tree, from one particular node down.
11. The three common types of traversal with a binary tree are __________, __________, and __________.

Algorithm Workbench

12. Write a pseudocode algorithm for inserting a node in a tree.
13. Write a pseudocode algorithm for the inorder traversal.
14. Write a pseudocode algorithm for the preorder traversal.
15. Write a pseudocode algorithm for the postorder traversal.
16. Write a pseudocode algorithm for searching a tree for a specified value.

17. Suppose the following values are inserted into a binary tree, in the order given:

    12, 7, 9, 10, 22, 24, 30, 18, 3, 14, 20

    Draw a diagram of the resulting binary tree.
18. How would the values in the tree you sketched for Question 17 be displayed in an inorder traversal?
19. How would the values in the tree you sketched for Question 17 be displayed in a preorder traversal?
20. How would the values in the tree you sketched for Question 17 be displayed in a postorder traversal?

True or False

21. T F Each node in a binary tree must have at least two children.
22. T F When a node is inserted into a tree, it must be inserted as a leaf node.
23. T F Values stored in the current node's left subtree are less than the value stored in the current node.
24. T F The shape of a binary tree is determined by the order in which values are inserted.
25. T F In inorder traversal, the node's data is processed first, then the left and right nodes are visited.

Programming Challenges

1. Binary Tree Template

Write your own version of a class template that will create a binary tree that can hold values of any data type. Demonstrate the class with a driver program.

VideoNote
Solving the Node Counter Problem

2. Node Counter

Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that counts and returns the number of nodes in the tree. Demonstrate the function in a driver program.

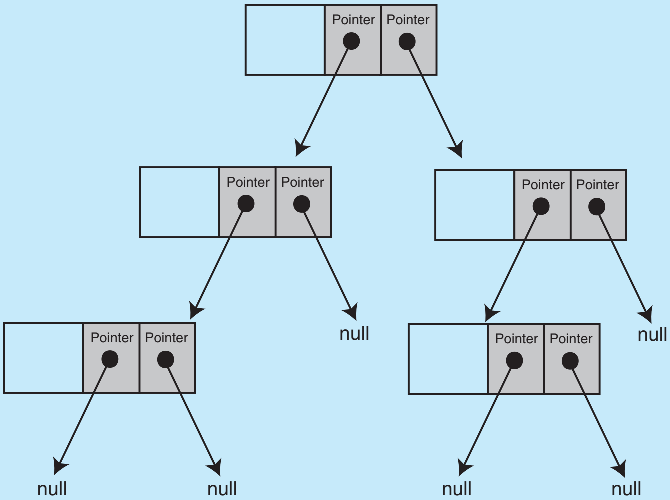
3. Leaf Counter

Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that counts and returns the number of leaf nodes in the tree. Demonstrate the function in a driver program.

4. Tree Height

Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that returns the height of the tree. The height of the tree is the number of levels it contains. For example, the tree shown in Figure 20-10 has three levels.

Figure 20-10



Demonstrate the function in a driver program.

5. **Tree Width**

   Write a member function, for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class, that returns the width of the tree. The width of the tree is the largest number of nodes in the same level. Demonstrate the function in a driver program.

6. **Tree Assignment Operator and Copy Constructor**

   Design an overloaded assignment operator and a copy constructor for either the template you designed in Programming Challenge 1 or the `IntBinaryTree` class. Demonstrate them in a driver program.

7. **Queue Converter**

   Write a program that stores a series of numbers in a binary tree. Then have the program insert the values into a queue in ascending order. Dequeue the values and display them on the screen to confirm that they were stored in the proper order.

8. **Employee Tree**

   Design an `EmployeeInfo` class that holds the following employee information:

Employee ID Number:	an integer
Employee Name:	a string

   Next, use the template you designed in Programming Challenge 1 to implement a binary tree whose nodes hold an instance of the `EmployeeInfo` class. The nodes should be sorted on the Employee ID number.

Test the binary tree by inserting nodes with the following information.

Employee ID Number	**Name**
1021	John Williams
1057	Bill Witherspoon
2487	Jennifer Twain
3769	Sophia Lancaster
1017	Debbie Reece
1275	George McMullen
1899	Ashley Smith
4218	Josh Plemmons

Your program should allow the user to enter an ID number, then search the tree for the number. If the number is found, it should display the employee's name. If the node is not found, it should display a message indicating so.