**Trees**

- Binary Trees
- Traversing Binary Trees
- Red-Black Trees
- Red-Black Tree Insertions
- 2-3-4 Trees
- Implementing 2-3-4 Trees

We will dwell on Binary trees, binary search trees and Binary heaps.

**Binary Trees**

Binary trees are one of the fundamental data structures used in programming. They provide advantages that the data structures we've seen so far (arrays and lists) cannot. In this tutorial we'll learn

- Why you would want to use trees
- Some terminology for describing trees
- What binary trees and binary search trees are
- How to go about creating trees
- How to find and insert data in a tree

We'll also present C++ code fragments for these activities. Next we'll find how to visit all the nodes in a tree, and examine a complete C++ program that incorporates the various tree operations.

**Why Use Binary Trees?**

Why might you want to use a tree? Usually because it combines the advantages of two other structures: an ordered array and a linked list. You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list. Let's explore these topics a bit before delving into the details of trees.

**Slow Insertion in an Ordered Array**

Imagine an array in which all the elements are arranged in order; that is, an ordered array, such as "Ordered Arrays." As we learned, it's quick to search such an array for a particular value, using a binary search. You check in the center of the array. If the object you're looking for is greater than what you find there, you narrow your search to the top half of the array; if it's less, you narrow your search to the bottom half. Applying this process repeatedly finds the object in O(log N) time. It's also quick to iterate through an ordered array, visiting each object in sorted order.

On the other hand, if you want to insert a new object into an ordered array, you first must find where the object will go, and then move all the objects with greater keys up one space in the array to make room for it. These multiple moves are time-consuming because they require, on average, moving half the items (N/2 moves). Deletion involves the same multimove operation, and is thus equally slow. If you're going to be doing a lot of insertions and deletions, an ordered array is a bad choice.

**Slow Searching in a Linked List**

On the other hand, as we saw in "Linked Lists," insertions and deletions are quick to perform on a linked list. They are accomplished simply by changing a few pointers. These operations require O(1) time (the fastest Big O time).

Unfortunately, however, finding a specified element in a linked list is not so easy. You must start at the beginning of the list and visit each element until you find the one you're looking for. Thus you will need to visit an average of N/2 objects, comparing each one's key with the desired value. This is slow, requiring O(N) time. (Notice that times considered fast for a sort are slow for data structure operations.)

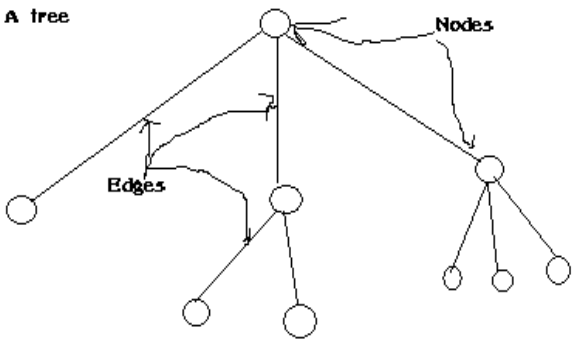
You might think you could speed things up by using an ordered linked list, in which the elements were arranged in order, but this doesn't help. You still must start at the beginning and visit the elements in order because there's no way to access a given element without following the chain of pointers to it. (Of course, in an ordered list it's much quicker to visit the nodes in order than it is in a non-ordered list, but that doesn't help to find an arbitrary object.)

**Trees to the Rescue**

It would be nice if there were a data structure with the quick insertion and deletion of a linked list, and also the quick searching of an ordered array. Trees provide both these characteristics, and are also one of the most interesting data structures.

**What Is a Tree?**

We'll be mostly interested in a particular kind of tree called a binary tree, but let's start by discussing trees in general before moving on to the specifics of binary trees. A tree consists of nodes connected by edges. Figure 15.1 shows a tree. In such a picture of a tree (or in our Workshop applet) the nodes are represented as circles, and the edges as lines connecting the circles.




A tree is actually an instance of a more general category called a *graph*

In computer programs, *nodes* often represent data items such as people, car parts, airline reservations, and so on; in other words, the typical items we store in any kind of data structure. In an OOP language like C++ these real-world entities are represented by objects. We've seen such data items stored in arrays and lists; now we'll see them stored in the nodes of trees.

The *lines* (edges) between the nodes represent the way the nodes are related. Roughly speaking, the lines represent convenience: It's easy (and fast) for a program to get from one node to another if there is a line connecting them. In fact, the only way to get from node to node is to follow a path along the lines. Often you are restricted to going in one direction along edges: from the root downward. Edges are likely to be represented in a program by pointers, if the program is written in C++.

Typically there is one node in the top row of a tree, with lines connecting to more nodes on the second row, even more on the third, and so on. Thus trees are small on the top and large on the bottom. This might seem upside-down compared with real trees, but generally a program starts an operation at the small end of the tree, and it's (arguably) more natural to think about going from top to bottom, as in reading text. There are different kinds of trees. The tree shown in Figure above has more than two children per node. (We'll see what *children* means in a moment.) However, in this hour we'll be discussing a specialized form of tree called a *binary* tree. Each node in a binary tree has a maximum of two children. More general trees, in which nodes can have more than two children, are called *multiway* trees.

Now let's look at terms for various parts of trees.

**Tree Terminology**

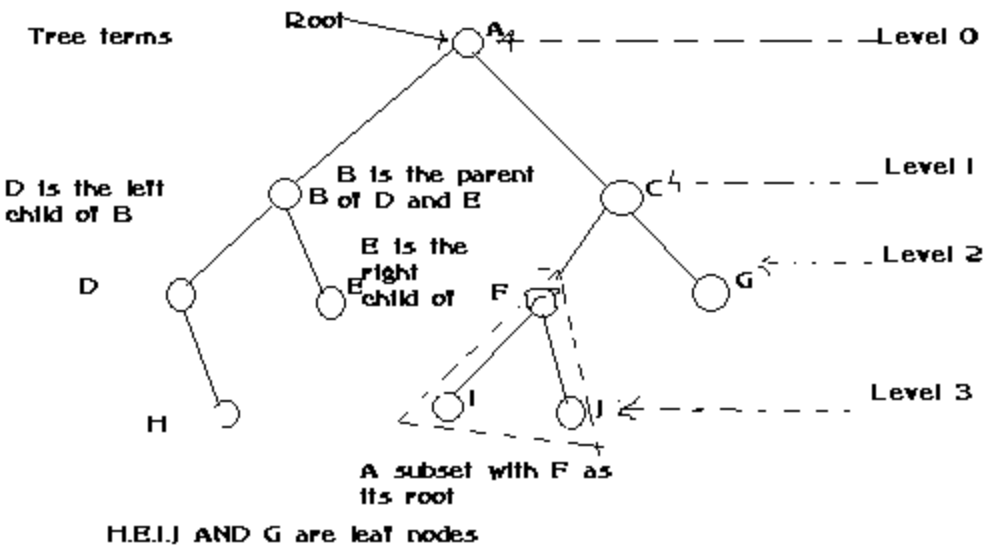
Many terms are used to describe particular aspects of trees. You need to know a few of them so our discussion will be comprehensible. Fortunately, most of these terms are related to real-world trees or to family relationships (as in parents and children), so they're not hard to remember. Figure 15.2 shows many of these terms applied to a binary tree.

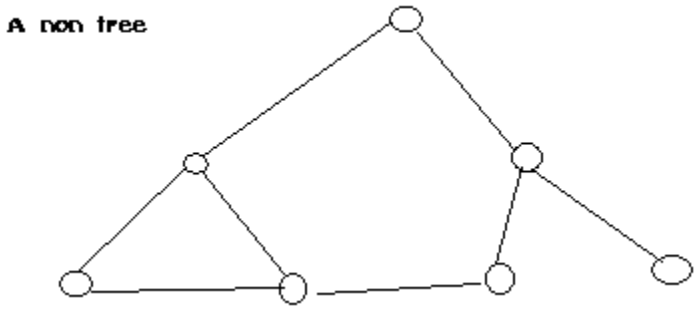
**Path**

Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a *path*.

**Root**

The node at the top of the tree is called the *root*. There is only one root in a tree. For a collection of nodes and edges to be defined as a tree, there must be one (and only one!) path from the root to any other node. Figure three shows a non-tree. You can see that it violates this rule.




**Parent**

Any node (except the root) has exactly one edge running upward to another node. The node above it is called the *parent* of the node.

**Child**

Any node can have one or more lines running downward to other nodes. These nodes below a given node are called its *children*.

**Leaf**

A node that has no children is called a *leaf node* or simply a *leaf*. There can be only one root in a tree, but there can be many leaves.

**Subtree**

Any node can be considered to be the root of a *subtree*, which consists of its children, and its children's children, and so on. If you think in terms of families, a node's subtree contains all its descendants.

**Visiting**

A node is *visited* when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data members, or displaying it. Merely passing over a node on the path from one node to another is not considered to be visiting the node.

**Traversing**

To *traverse* a tree means to visit all the nodes in some specified order. For example, you might visit all the nodes in order of ascending key value. There are other ways to traverse a tree, as we'll see in the next hour.

**Levels**

The *level* of a particular node refers to how many generations the node is from the root. If we assume the root is Level 0, its children will be Level 1, its grandchildren will be Level 2, and so on.

**Keys**

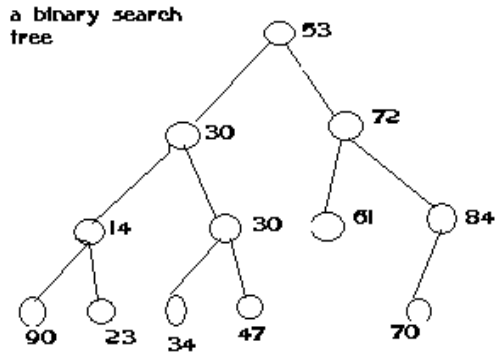
We've seen that one data item in an object is usually designated a *key value*. This value is used to search for the item or perform other operations on it. In tree diagrams, when a circle represents a node holding a data item, the key value of the item is typically shown in the circle. (We'll see many figures later on that show how this looks.)

**Binary Trees**

If every node in a tree can have at most two children, the tree is called a *binary tree*. In this hour we'll focus on binary trees because they are the simplest, the most common, and in many situations the most frequently used.

The two children of each node in a binary tree are called the *left child* and the *right child*, corresponding to their positions when you draw a picture of a tree, as shown in Figure 15.2. A node in a binary tree doesn't necessarily have the maximum of two children; it might have only a left child, or only a right child, or it can have no children at all (which means it's a leaf).

The kind of binary tree we'll be dealing with in this discussion is technically called a *binary search tree*. The defining characteristic of a binary search tree is this: A node's left child must have a key less than its parent, and a node's right child must have a key greater than or equal to its parent. Figure 15.4 shows a binary search tree.




Now that we've learned how to describe the parts of a tree, let's look at a tree structure that you're probably already familiar with.

**A Tree Analogy in Your Computer**

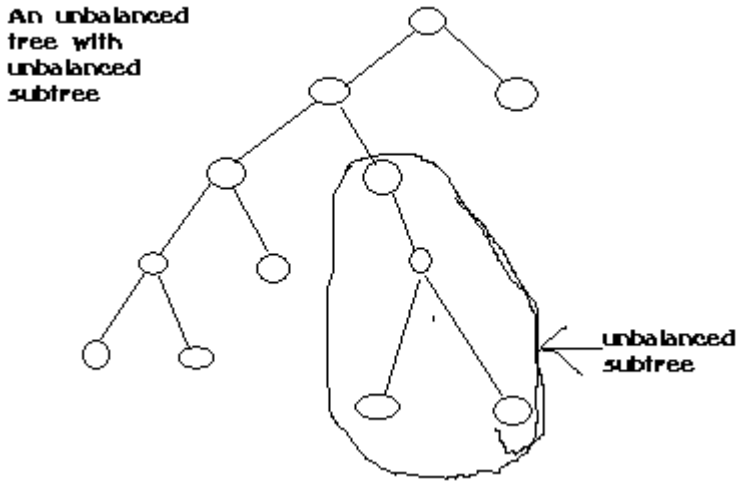
One commonly encountered tree is the hierarchical file structure in a computer system. The root directory of a given device (designated with the backslash, as in C:\, on many systems) is the tree's root. The directories one level below the root directory is its children. There may be many levels of subdirectories. Files are leaves; they have no children of their own.

Clearly a hierarchical file structure is not a binary tree because a directory can have many children. A complete pathname, such as C:\SALES\EAST\NOVEMBER\SMITH.DAT, corresponds to the path from the root to the SMITH.DAT leaf node. Terms used for file structures, such as root and path, were borrowed from tree theory.

A hierarchical file structure differs in a significant way from the trees we'll be discussing here. In the file structure, subdirectories contain no data; they contain only references to other subdirectories or to files. Only files contain data. In a tree, every node contains data (a personnel record, car part specifications, or whatever). In addition to the data, all nodes (except leaves) contain pointers to other nodes.

**Unbalanced Trees**

Notice that some of thetrees you generate are *unbalanced*, that is, they have most of their nodes on one side of the root or the other, as shown in Figure below. Individual subtrees may also be unbalanced.




Trees become unbalanced because of the order in which the data items are inserted. If these key values are inserted randomly, the tree will be more or less balanced. However, if an ascending sequence (like 11, 18, 33, 42, 65, and so on) or a descending sequence is generated, all the values will be right children (if ascending) or left children (if descending) and the tree will be unbalanced. The key values in the Workshop applet are generated randomly, but of course some short ascending or descending sequences will be created anyway, which will lead to local imbalances. When you learn how to insert items into the tree in the Workshop applet, you can try building up a tree by inserting such an ordered sequence of items and see what happens.

If you ask for a large number of nodes when you use Fill to create a tree, you might not get as many nodes as you requested. Depending on how unbalanced the tree becomes, some branches might not be able to hold a full number of nodes. This is because the depth of the applet's tree is limited to five; the problem would not arise in a real tree.
If a tree is created by data items whose key values arrive in random order, the problem of unbalanced trees might not be too much of a problem for larger trees because the chances of a long run of numbers in sequence are small. But key values can arrive in strict sequence; for example, when a data-entry person arranges a stack of personnel files into order of ascending employee number before entering the data. When this happens, tree efficiency can be seriously degraded.

**Representing the Tree in C++ Code**

Let's see how we might implement a binary tree in C++. As with other data structures, there are several approaches to representing a tree in the computer's memory. The most common is to store the nodes at unrelated locations in memory, and connect them using pointers in each node that point to its children. (It's also possible to represent a tree in memory as an array, but we'll ignore that possibility here.)

As we discuss individual operations we'll show code fragments pertaining to that operation.

The complete program from which these fragments are extracted can be seen later.

**The** Node **Class**

First, we need a class of node objects. These objects contain the data representing the objects being stored (employees in an employee database, for example) and also pointers to each of the node's two children. Here's how that looks.

```
class Node
{
public:
int iData; //data item (key)
double dData; //data item
Node* pLeftChild; //this node's left child
Node* pRightChild; //this node's right child
//--------------------------------------------------------------
//constructor
Node() : iData(0), dData(0.0), pLeftChild(NULL),
pRightChild(NULL)
{ }
//--------------------------------------------------------------
void displayNode() //display ourself: {75, 7.5}
{
cout << '{' << iData << ", " << dData << "} ";
}
}; //end class Node
```

Some programmers also include a pointer to the node's parent. This simplifies some operations but complicates others, so we don't include it. We do include a member function called displayNode() to display the node's data, but its code isn't relevant here.

There are other approaches to designing class Node. Instead of placing the data items directly into the node, you could use a pointer to an object representing the data item:

```
class Node
{
Person* p1; //pointer to Person object
Node* pLeftChild; //pointer to left child
Node* pRightChild; //pointer to right child
};
```

This makes it conceptually clearer that the node and the data item it holds aren't the same thing, but it results in somewhat more complicated code, so we'll stick to the first approach.

**The** Tree **Class**

We'll also need a class from which to create the tree itself; the object that holds all the nodes. We'll call this class Tree. It has only one data member: a Node* variable that holds a pointer to the root. It doesn't need data members for the other nodes because they are all accessed from the root.

The Tree class has a number of member functions: for finding and inserting, several for different kinds of traverses, and one to display the tree. Here's a skeleton version:

```
class Tree
{
private:
Node* pRoot; //first node of tree
public:
//--------------------------------------------------------------
Tree() : pRoot(NULL) //constructor
{ }
//--------------------------------------------------------------
Node* find(int key) //find node with given key
{ /*body not shown*/ }
//--------------------------------------------------------------
void insert(int id, double dd) //insert new node
{ /*body not shown*/ }
//--------------------------------------------------------------
void traverse(int traverseType)
{ /*body not shown*/ }
//--------------------------------------------------------------
void displayTree()
{ /*body not shown*/ }
//--------------------------------------------------------------
}; //end class Tree
```

**The** main() **Function**

Finally, we need a way to perform operations on the tree. Here's how you might write a main() routine to create a tree, insert three nodes into it, and then search for one of them:

```
int main()
{
Tree theTree; //make a tree
theTree.insert(50, 1.5); //insert 3 nodes
theTree.insert(25, 1.7);
theTree.insert(75, 1.9);
Node* found = theTree.find(25); //find node with key 25
if(found != NULL)
```

```
cout << “Found the node with key 25” << endl;
else
cout << “Could not find node with key 25” << endl;
return 0;
} // end main()
```

In Listing 16.1 in the next hour the main() routine provides a primitive user interface so you can use the keyboard to insert, find, or perform other operations.

Next we’ll look at individual tree operations: finding a node and inserting a node. We’ll also briefly mention the problem of deleting a node.

**Finding a Node**

Finding a node with a specific key is the simplest of the major tree operations. Remember that the nodes in a binary search tree correspond to objects containing information. They could be objects representing people, with an employee number as the key and also perhaps name, address, telephone number, salary, and other data members. Or they could represent car parts, with a part number as the key value and data members for quantity on hand, price, and so on. However, the only characteristics of each node that we can see in the Workshop applet are a number and a color. A node is created with these two characteristics, and keeps them throughout its life.

**Using Example below Find a Node**

Look at the Workshop applet, and pick a node, preferably one near the bottom of the tree (as far from the root as possible). The number shown in this node is its key value. We’re going to demonstrate how the Workshop applet finds the node, given the key value. For purposes of this discussion we’ll assume you’ve decided to find the node representing the item with key value 57, as shown in Figure 15.7. Of course, when you run the Workshop applet you’ll get a different tree and will need to pick a different key value.

In Figure 15.7 the arrow starts at the root. The program compares the key value 57 with the value at the root, which is 63. The key is less, so the program knows the desired node must be on the left side of the tree; either the root’s left child or one of this child’s descendants. The left child of the root has the value 27, so the comparison of 57 and 27 will show that the desired node is in the right subtree of 27. The arrow will go to 51, the root of this subtree. Here, 57 is again greater than the 51 node, so we go to the right, to 58, and then to the left, to 57. This time the comparison shows 57 equals the node’s key value, so we’ve found the node we want.

**C++ Code for Finding a Node**

Here’s the code for the find() routine, which is a member function of the Tree class.

```
Node* find(int key) //find node with given key
{ //(assumes non-empty tree)
Node* pCurrent = pRoot; //start at root
while(pCurrent->iData != key) //while no match,
{
```

```
if(key < pCurrent->iData) //go left?
pCurrent = pCurrent->pLeftChild;
else //or go right?
pCurrent = pCurrent->pRightChild;
if(pCurrent == NULL) //if no child,
return NULL; //didn't find it
}
return pCurrent; //found it
} //end find()
```

This routine uses the variable pCurrent to hold a pointer to the node it is currently examining. The argument key is the value to be found. The routine starts at the root. (It has to; this is the only node it can access directly.) That is, it sets pCurrent to the root.

Then, in the while loop, it compares the value to be found, key, with the value of the iData member (the key) in the current node. If key is less than this data member, pCurrent is set to the node's left child. If key is greater than (or equal) to the node's iData data member, then pCurrent is set to the node's right child.

**Can't Find the Node**

If pCurrent becomes equal to NULL, we couldn't find the next child node in the sequence; we've reached the end of the line without finding the node we were looking for, so it can't exist. We return NULL to indicate this fact.

**Found the Node**

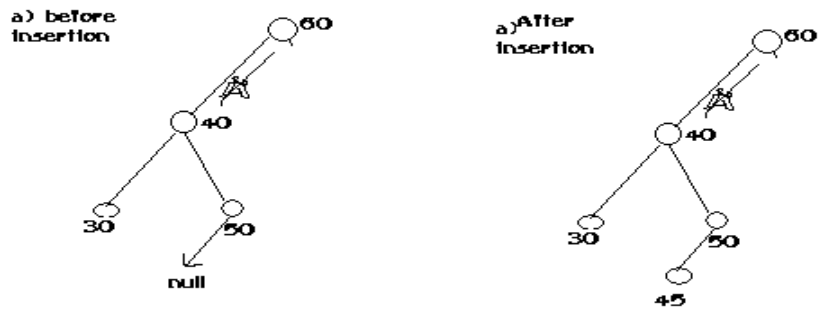
If the condition of the while loop is not satisfied, so that we exit from the bottom of the loop, the iData data member of pCurrent is equal to key; that is, we've found the node we want. We return the node, so that the routine that called find() can access any of the node's data.

**Efficiency of the Find Operation**

As you can see, how long it takes to find a node depends on how many levels down it is situated. In the Workshop applet there can be up to 31 nodes, but no more than 5 levels. Thus you can find any node using a maximum of only 5 comparisons. This is O(log N) time, or more specifically O(log2 N) time; the logarithm to the base 2. We'll discuss this further toward the end of the next hour.

**Inserting a Node**

To insert a node we must first find the place to insert it. This is much the same process as trying to find a node which turns out not to exist, as described in the section on find. We follow the path from the root to the appropriate node, which will be the parent of the new node.After this parent is found, the new node is connected as its left or right child, depending on whether the new node's key is less or greater than that of the parent.




The value 45 is less than 60 but greater than 40, so we arrive at node 50. Now we want to go left because 45 is less than 50, but 50 has no left child; its pLeftChild data member is NULL. When it sees this NULL, the insertion routine has found the place to attach the new node. The Workshop applet does this by creating a new node with the value 45 (and a randomly generated color) and connecting it as the left child of 50, as shown in Figure below

**C++ Code for Inserting a Node**

The insert() function starts by creating the new node, using its arguments to supply the data.

Next, insert() must determine where to insert the new node. This is done using roughly the same code as finding a node, described in the section on find(). The difference is that when you're simply trying to *find* a node and you encounter a NULL (non-existent) node, you know the node you're looking for doesn't exist, so you return immediately.

When you're trying to *insert* a node you insert it (creating it first, if necessary) before returning.

The value to be searched for is the data item passed in the argument id. The while loop uses true as its condition because it doesn't care if it encounters a node with the same value as id; it treats another node with the same key value as if it were simply greater than the key value. (We'll return to the subject of duplicate nodes in the next hour.)

In a real tree (as opposed to the Workshop applet) a place to insert a new node will always be found (unless you run out of memory); when it is, and the new node is attached, the while loop exits with a return statement.

Here's the code for the insert() function:

```
void insert(int id, double dd) //insert new node
{
Node* pNewNode = new Node; //make new node
pNewNode->iData = id; //insert data
pNewNode->dData = dd;
if(pRoot==NULL) //no node in root
pRoot = pNewNode;
else //root occupied
{
Node* pCurrent = pRoot; //start at root
Node* pParent;
while(true) //(exits internally)
{
pParent = pCurrent;
```

```
if(id < pCurrent->iData) //go left?
{
pCurrent = pCurrent->pLeftChild;
if(pCurrent == NULL) //if end of the line,
{ //insert on left
pParent->pLeftChild = pNewNode;
return;
}
} //end if go left
else //or go right?
{
pCurrent = pCurrent->pRightChild;
if(pCurrent == NULL) //if end of the line
{ //insert on right
pParent->pRightChild = pNewNode;
return;
}
} //end else go right
} //end while
} //end else not root
} //end insert()
```

We use a new variable, pParent (a pointer to the parent of pCurrent), to remember the last non-NULL node we encountered (50 in the figure). This is necessary because pCurrent is set to NULL in the process of discovering that its previous value did not have an appropriate child. If we didn't save pParent, we would lose track of where we were.
To insert the new node, change the appropriate child pointer in pParent (the last non-NULL node you encountered) to point to the new node. If you were looking unsuccessfully for pParent's left child, you attach the new node as pParent's left child; if you were looking for its right child, you attach the new node as its right child. In Figure 15.8, 45 are attached as the left child of 50.

**Deleting a Node**

Research on the same

**Summary**

In this hour, you've learned the following:

- Trees consist of nodes (circles) connected by edges (lines).
- The root is the topmost node in a tree; it has no parent.
- In a binary tree, a node has at most two children.
- In a binary search tree, all the nodes that are left descendants of node A have key values less than A; all the nodes that are A's right descendants have key values greater than (or equal to) A.
- Trees perform searches, insertions, and deletions in O(log N) time.
- Nodes represent the data-objects being stored in the tree.

● Edges are most commonly represented in a program by pointers to a node's children (and sometimes to its parent).
● An unbalanced tree is one whose root has many more left descendents than right descendants, or vice versa.
● Searching for a node involves comparing the value to be found with the key value of a node, and going to that node's left child if the key search value is less, or to the node's right child if the search value is greater.
● Insertion involves finding the place to insert the new node, and then changing a child data member in its new parent to refer to it.

**Q&A**

**Q Trees seems much more complicated than arrays or linked lists. Are they really useful?**

**A** Trees are probably the single most useful data structure. They have comparatively fast searching, insertion, and deletion, which is not the case with simpler structures. For storing large amounts of data, a tree is usually the first thing you should consider.

**Q Don't you sometimes need to rearrange the nodes in a tree when you insert a new node?**

**A** Never. The new node is always attached to a leaf node, or as the missing child of a node with one child. However, when deleting a node rearrangement may be necessary.

**Q Can we use the tree we've seen in this hour as a general-purpose data storage structure?**

**A** Only in some circumstances. As we'll discuss in Hour 17, simple trees work poorly when the order of data insertion creates an unbalanced tree.

## Traversing Binary Trees

In this hour we'll continue our discussion of binary search trees. You'll learn
● What it means to traverse a tree
● Three different kinds of traversals
● How to write C++ code to traverse a tree
● About the efficiency of binary trees
We'll also present the complete C++ listing that ties together the various binary-tree member functions we've seen so far.

**Traversing the Tree**

Traversing a tree means visiting each node in a specified order. This process is not as commonly used as finding, inserting, and deleting nodes. One reason for this is that traversal is not particularly fast. But traversing a tree has some surprisingly useful applications and is theoretically interesting. There are three simple ways to traverse a tree. They're called *preorder*, *inorder*, and *postorder*. The order most commonly used for binary search trees is in order, so let's look at that first, and then return briefly to the other two.

**Inorder Traversal**

An *inorder traversal* of a binary search tree will cause all the nodes to be visited in ascending order, based on their key values. If you want to create a sorted list of the data in a binary tree, this is one way to do it.
The simplest way to carry out a traversal is the use of recursion

Here's how it works. A recursive function to traverse the tree is called with a node as an argument. Initially, this node is the root. The function must perform only three tasks.
1. Call itself to traverse the node's left subtree.
2. Visit the node.
3. Call itself to traverse the node's right subtree.
Remember that visiting a node means doing something to it: displaying it, writing it to a file, or whatever.
Traversals work with any binary tree, not just with binary search trees. The traversal mechanism doesn't pay any attention to the key values of the nodes; it only concerns itself with whether a node has children.

**C++ Code for Traversing**

The actual code for inorder traversal is so simple we show it before seeing how traversal looks in the Workshop applet. The routine, inOrder(), performs the three steps already described. The visit to the node consists of displaying the contents of the node. Like any recursive function, there must be a base case: the condition that causes the routine to return immediately, without calling itself. In inOrder() this happens when the node passed as an argument is NULL. Here's the code for the inOrder() member function:

```
void inOrder(Node* pLocalRoot)
{
if(pLocalRoot != NULL)
{
inOrder(pLocalRoot->pLeftChild); //left child
cout << pLocalRoot->iData << " "; //display node
inOrder(pLocalRoot->pRightChild); //right child
}
}
```

This member function is initially called with the root as an argument:

```
inOrder(root);
```

After that, the function is on its own, calling itself recursively until there are no more nodes to visit.

**Traversing a 3-Node Tree**

Let's look at a simple example to get an idea of how this recursive traversal routine works. Imagine traversing a tree with only three nodes: a root (A), with a left child (B), and a right child (C), as shown below

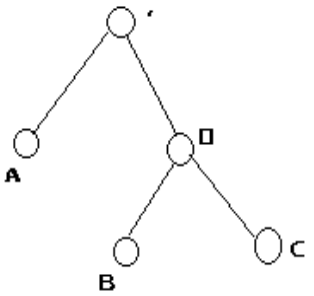
**To Do: Follow the Steps of an InOrder Traversal**

1. Start by calling inOrder() with the root A as an argument. This incarnation of inOrder() we'll call inOrder(A).
2. inOrder(A) first calls inOrder() with its left child, B, as an argument. This second incarnation of inOrder() we'll call inOrder(B).
3. inOrder(B) now calls itself with its left child as an argument. However, it has no left child, so this argument is NULL. This creates an invocation of inorder() we could call inOrder(NULL).
4. There are now three instances of inOrder() in existence: inOrder(A), inOrder(B), and inOrder(NULL). However, inOrder(NULL) returns immediately when it finds its argument is NULL. (We all have days like that.)
5. Now inOrder(B) goes on to visit B; we'll assume this means to display it.
6. Then inOrder(B) calls inOrder() again, with its right child as an argument. Again this argument is NULL, so the second inorder(NULL) returns immediately.
7. Now inOrder(B) has carried out tasks 1, 2, and 3, so it returns (and thereby ceases to exist).
8. Now we're back to inOrder(A), just returning from traversing A's left child.
9. We visit A, and then call inOrder() again with C as an argument, creating inOrder(C). Like inOrder(B), inOrder(C) has no children, so task 1 returns with no action, task 2 visits C, and task 3 returns with no action.
10. inOrder(B) now returns to inOrder(A).
11. However, inOrder(A) is now done, so it returns and the entire traversal is complete.

The order in which the nodes were visited is A, B, C; they have been visited *inorder*. In a binary search tree this would be the order of ascending keys. More complex trees are handled similarly. The inOrder() function calls itself for each node, until it has worked its way through the entire tree.

**Preorder and Post order Traversals**

You can traverse the tree in two ways besides inorder; they're called *preorder* and *postorder*. It's fairly clear why you might want to traverse a tree inorder, but the motivation for preorder and postorder traversals is more obscure. However, these traversals are indeed useful if you're writing programs that parse or analyze algebraic expressions. Let's see why that should be true. A binary tree (not a binary search tree) can be used to represent an algebraic expression that involves the binary arithmetic operators +, -, /, and *. The root node holds an operator, and the other nodes represent either a variable name (like A, B, or C), or another operator. Each subtree is an algebraic expression. For example, the binary tree shown in Figure 16.3 represents the algebraic expression A*(B+C). This is called *infix* notation; it's the notation normally used in algebra. Traversing the tree inorder will generate the correct inorder sequence A*B+C, but you'll need to insert the parentheses yourself



What's all this got to do with preorder and postorder traversals? Let's see what's involved. For these other traversals the same three tasks are used as for inorder, but in a different sequence. Here's the sequence for a preorder() member function:

1. Visit the node.
2. Call itself to traverse the node's left subtree.
3. Call itself to traverse the node's right subtree.

Traversing the tree shown in Figure above using preorder would generate the expression *A+BC.

This is called *prefix* notation. It's another equally valid way to represent an algebraic expression. One of the nice things about it is that parentheses are never required; the expression is unambiguous without them. Starting on the left, each operator is applied to the next two things in the expression. For the first operator, *, these two things are A and +BC. In turn, the expression +BC means "apply + to the next two things in the expression"—which are B and C—so this last expression is B+C in inorder notation. Inserting that into the original expression *A+BC (preorder) gives us A*(B+C) in inorder.

By simply using different traversals, we've transformed one kind of algebraic notation into another. The postorder traversal member function contains the three tasks arranged in yet another way:

1. Call itself to traverse the node's left subtree.
2. Call itself to traverse the node's right subtree.
3. Visit the node.

For the tree in Figure above, visiting the nodes with a postorder traversal would generate the expression

ABC+*

This is called *postfix* notation. Starting on the right, each operator is applied to the two things on its left. First we apply the * to A and BC+.

Following the rule again for BC+, we apply the + to B and C. This gives us (B+C) in infix. Inserting this in the original expression ABC+* (postfix) gives us A*(B+C) infix.

Besides writing different kinds of algebraic expressions, you might find other clever uses for the different kinds of traversals. As we'll see at the end of this hour, we use postorder traversal to delete all the nodes when the tree is destroyed.

The code in Listing 16.1 later in this hour contains member functions for preorder and postorder traversals, as well as for inorder. Now let's move on from traversals and briefly examine another aspect of binary search trees.

**The Efficiency of Binary Trees**

As you've seen, most operations with trees involve descending the tree from level to level to find a particular node. How long does it take to do this? In a full tree, about half the nodes are on the bottom level. (Actually there's one more node on the bottom row than in the rest of the tree.) Thus about half of all searches or insertions or deletions require finding a node on the lowest level. (An additional quarter of these operations require finding the node on the next-to-lowest level, and so on.)

During a search we need to visit one node on each level. So we can get a good idea how long it takes to carry out these operations by knowing how many levels there are. Assuming a full tree, Table below shows how many levels are necessary to hold a given number of nodes.

This situation is very much like the ordered array discussed in, “Ordered Arrays.” In that case, the number of comparisons for a binary search was approximately equal to the base-2 logarithm of the number of cells in the array. Here, if we call the number of nodes in the first column N, and the number of levels in the second column L, we can say that N is 1 less than 2 raised to the power L, or N = 2L – 1 Adding 1 to both sides of the equation, we have N+1 = 2L. This is equivalent to L = log2(N+1)
Thus the time needed to carry out the common tree operations is proportional to the base-2 log of N. In Big O notation we say such operations take O(log N) time

**Implementing a Binary Search Tree in C++**

Besides implementing a binary search tree, the tree.cpp program also features a primitive user interface. This allows the user to choose an operation (finding, inserting, traversing and displaying the tree) by entering characters. The display routine uses character output to generate a picture of the tree.

THE tree.cpp PROGRAM

```
//tree.cpp
//demonstrates binary tree
#include <iostream>
#include <stack>
using namespace std;
////////////////////////////////////////////////////////////////
class Node
{
public:
int iData; //data item (key)
double dData; //data item
Node* pLeftChild; //this node’s left child
Node* pRightChild; //this node’s right child
//-------------------------------------------------------------
//constructor
Node() : iData(0), dData(0.0), pLeftChild(NULL),
pRightChild(NULL)
{ }
//-------------------------------------------------------------
~Node() //destructor
{ cout << “X-” << iData << “ “; }
//-------------------------------------------------------------
void displayNode() //display ourself: {75, 7.5}
{
cout << ‘{‘ << iData << “, “ << dData << “} “;
}
```

```
}; //end class Node
////////////////////////////////////////////////////////////////
class Tree
{
private:
Node* pRoot; //first node of tree
public:
//--------------------------------------------------------------
Tree() : pRoot(NULL) //constructor
{ }
//--------------------------------------------------------------
Node* find(int key) //find node with given key
{ //(assumes non-empty tree)
Node* pCurrent = pRoot; //start at root
while(pCurrent->iData != key) //while no match,
{
if(key < pCurrent->iData) //go left?
pCurrent = pCurrent->pLeftChild;
else //or go right?
pCurrent = pCurrent->pRightChild;
if(pCurrent == NULL) //if no child,
return NULL; //didn't find it
}
return pCurrent; //found it
} //end find()
//--------------------------------------------------------------
void insert(int id, double dd) //insert new node
{
Node* pNewNode = new Node; //make new node
pNewNode->iData = id; //insert data
pNewNode->dData = dd;
if(pRoot==NULL) //no node in root
pRoot = pNewNode;
else //root occupied
{
Node* pCurrent = pRoot; //start at root
Node* pParent;
while(true) //(exits internally)
{
pParent = pCurrent;
if(id < pCurrent->iData) //go left?
{
```

```
pCurrent = pCurrent->pLeftChild;
if(pCurrent == NULL) //if end of the line,
{ //insert on left
pParent->pLeftChild = pNewNode;
return;
}
} //end if go left
else //or go right?
{
pCurrent = pCurrent->pRightChild;
if(pCurrent == NULL) //if end of the line
{ //insert on right
pParent->pRightChild = pNewNode;
return;
}
} //end else go right
} //end while
} //end else not root
} //end insert()
//--------------------------------------------------------------
void traverse(int traverseType)
{
switch(traverseType)
{
case 1: cout << "\nPreorder traversal: ";
preOrder(pRoot);
break;
case 2: cout << "\nInorder traversal: ";
inOrder(pRoot);
break;
case 3: cout << "\nPostorder traversal: ";
postOrder(pRoot);
break;
}
cout << endl;
}
//--------------------------------------------------------------
void preOrder(Node* pLocalRoot)
{
if(pLocalRoot != NULL)
{
cout << pLocalRoot->iData << " "; //display node
```

```
preOrder(pLocalRoot->pLeftChild); //left child
preOrder(pLocalRoot->pRightChild); //right child
}
}
//--------------------------------------------------------------
void inOrder(Node* pLocalRoot)
{
if(pLocalRoot != NULL)
{
inOrder(pLocalRoot->pLeftChild); //left child
cout << pLocalRoot->iData << " "; //display node
inOrder(pLocalRoot->pRightChild); //right child
}
}
//--------------------------------------------------------------
void postOrder(Node* pLocalRoot)
{
if(pLocalRoot != NULL)
{
postOrder(pLocalRoot->pLeftChild); //left child
postOrder(pLocalRoot->pRightChild); //right child
cout << pLocalRoot->iData << " "; //display node
}
}
//--------------------------------------------------------------
void displayTree()
{
stack<Node*> globalStack;
globalStack.push(pRoot);
int nBlanks = 32;
bool isRowEmpty = false;
cout <<
"..........";
cout << endl;
while(isRowEmpty==false)
{
stack<Node*> localStack;
isRowEmpty = true;
for(int j=0; j<nBlanks; j++)
cout << ' ';
while(globalStack.empty()==false)
{
```

```
Node* temp = globalStack.top();
globalStack.pop();
if(temp != NULL)
{
cout << temp->iData;
localStack.push(temp->pLeftChild);
localStack.push(temp->pRightChild);
if(temp->pLeftChild != NULL ||
temp->pRightChild != NULL)
isRowEmpty = false;
}
else
{
cout << "--";
localStack.push(NULL);
localStack.push(NULL);
}
for(int j=0; j<nBlanks*2-2; j++)
cout << ' ';
} //end while globalStack not empty
cout << endl;
nBlanks /= 2;
while(localStack.empty()==false)
{
globalStack.push( localStack.top() );
localStack.pop();
}
} //end while isRowEmpty is false
cout <<
"...........................................................";
cout << endl;
} //end displayTree()
//--------------------------------------------------------------
void destroy() //deletes all nodes
{ destroyRec(pRoot); } //start at root
//--------------------------------------------------------------
void destroyRec(Node* pLocalRoot) //delete nodes in
{ // this subtree
if(pLocalRoot != NULL)
{ //uses postOrder
destroyRec(pLocalRoot->pLeftChild); //left subtree
destroyRec(pLocalRoot->pRightChild); //right subtree
```

```
delete pLocalRoot; //delete this node
}
}
//--------------------------------------------------------------
}; //end class Tree
////////////////////////////////////////////////////////////////
int main()
{
int value;
char choice;
Node* found;
Tree theTree; //create tree
theTree.insert(50, 5.0); //insert nodes
theTree.insert(25, 2.5);
theTree.insert(75, 7.5);
theTree.insert(12, 1.2);
theTree.insert(37, 3.7);
theTree.insert(43, 4.3);
theTree.insert(30, 3.0);
theTree.insert(33, 3.3);
theTree.insert(87, 8.7);
theTree.insert(93, 9.3);
theTree.insert(97, 9.7);
while(choice != 'q') //interact with user
{ //until user types 'q'
cout << "Enter first letter of ";
cout << "show, insert, find, traverse or quit: ";
cin >> choice;
switch(choice)
{
case 's': //show the tree
theTree.displayTree();
break;
case 'i': //insert a node
cout << "Enter value to insert: ";
cin >> value;
theTree.insert(value, value + 0.9);
break;
case 'f': //find a node
cout << "Enter value to find: ";
cin >> value;
found = theTree.find(value);
```

```
if(found != NULL)
{
cout << "Found: ";
found->displayNode();
cout << endl;
}
else
cout << "Could not find " << value << endl;
break;
case 't': //traverse the tree
cout << "Enter traverse type (1=preorder, "
<< "2=inorder, 3=postorder): ";
cin >> value;
theTree.traverse(value);
break;
case 'q': //quit the program
theTree.destroy();
cout << endl;
break;
default:
cout << "Invalid entry\n";
} //end switch
} //end while
return 0;
} //end main()
```

**Summary**

In this hour, you've learned the following:

- Traversing a tree means visiting all its nodes in some order.
- The simple traversals are preorder, inorder, and postorder.
- An inorder traversal visits nodes in order of ascending keys.
- Preorder and postorder traversals are useful for parsing algebraic expressions, among other things.
- Nodes with duplicate key values might cause trouble because only the first one can be found in a search.
- All the common operations on a binary search tree can be carried out in O(log N) time.