**Binary Trees in C++**

WE HAVE SEEN how objects can be linked into lists. When an object contains two pointers to objects of the same type, structures can be created that are much more complicated than linked lists. In this section, we'll look at one of the most basic and useful structures of this type: binary trees. Each of the objects in a binary tree contains two pointers, typically called left and right. In addition to these pointers, of course, the nodes can contain other types of data. For example, a binary tree of integers could be made up of objects of the following type:

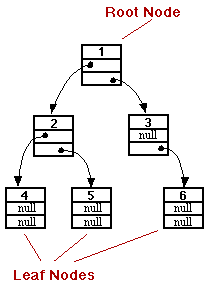
struct TreeNode {

int item; // The data in this node.

TreeNode \*left; // Pointer to the left subtree.

TreeNode \*right; // Pointer to the right subtree.

}

The left and right pointers in a TreeNode can be NULL or can point to other objects of type TreeNode. A node that points to another node is said to be theparent of that node, and the node it points to is called a child. In the picture at the right, for example, node 3 is the parent of node 6, and nodes 4 and 5 are children of node 2. Not every linked structure made up of tree nodes is a binary tree. A binary tree must have the following properties: There is exactly one node in the tree which has no parent. This node is called the root of the tree. Every other node in the tree has exactly one parent. Finally, there can be no loops in a binary tree. That is, it is not possible to follow a chain of pointers starting at some node and arriving back at the same node.

A node that has no children is called a leaf. A leaf node can be recognized by the fact that both the left and right pointers in the node are NULL. In the standard picture of a binary tree, the root node is shown at the top and the leaf nodes at the bottom -- which doesn't show much respect with the analogy to real trees. But at least you can see the branching, tree-like structure that gives a binary tree its name.

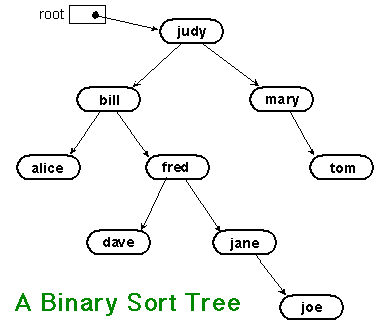
Consider any node in a binary tree. Look at that node together with all its descendents (that is, its children, the children of its children, and so on). This set of nodes forms a binary tree, which is called a subtree of the original tree. For example, in the picture, nodes 2, 4, and 5 form a subtree. This subtree is called the left subtree of the root. Similarly, nodes 3 and 6 make up the right subtree of the root. We can consider any non-empty binary tree to be made up of a root node, a left subtree, and a right subtree. Either or both of the subtrees can be empty. This is a **recursive** definition, matching the recursive definition of the TreeNode class. So it should not be a surprise that recursive functions are often used to process trees.

Consider the problem of counting the nodes in a binary tree. As an exercise, you might try to come up with a non-recursive algorithm to do the counting. The heart of problem is keeping track of which nodes remain to be counted. It's not so easy to do this, and in fact it's not even possible without using an auxiliary data structure. With recursion, however, the algorithm is almost trivial. Either the tree is empty or it consists of a root and two subtrees. If the tree is empty, the number of nodes is zero. (This is the base case of the recursion.) Otherwise, use recursion to count the nodes in each subtree. Add the results from the subtrees together, and add one to count the root. This gives the total number of nodes in the tree.

**Binary Sort Trees**

One of our examples using a list was a linked list of strings, in which the strings were kept in increasing order. While a linked list works well for a small number of strings, it becomes inefficient for a large number of items. When inserting an item into the list, searching for that item's position requires looking at, on average, half the items in the list. Finding an item in the list requires a similar amount of time. If the strings are stored in a sorted array instead of in a linked list, then searching becomes more efficient because binary search can be used. However, inserting a new item into the array is still inefficient since it means moving, on average, half of the items in the array to make a space for the new item. A binary tree can be used to store an ordered list of strings, or other items, in a way that makes both searching and insertion efficient. A binary tree used in this way is called a binary sort tree.

A binary sort tree is a binary tree with the following property: For every node in the tree, the item in that node is greater than every item in the left subtree of that node, and it is less than or equal to all the items in the right subtree of that node. Here for example is a binary sort tree containing items of type string. (In this picture, I haven't bothered to draw all the pointer variables. Non-NULL pointers are shown as arrows.)



Binary sort trees have this useful property: An inorder traversal of the tree will process the items in increasing order. In fact, this is really just another way of expressing the definition. For example, if an inorder traversal is used to print the items in the tree shown above, then the items will be in alphabetical order. The definition of an inorder traversal guarantees that all the items in the left subtree of "judy" are printed before "judy", and all the items in the right subtree of "judy" are printed after "judy". But the binary sort tree property guarantees that the items in the left subtree of "judy" are precisely those that precede "judy" in alphabetical order, and all the items in the right subtree follow "judy" in alphabetical order. So, we know that "judy" is output in its proper alphabetical position. But the same argument applies to the subtrees. "Bill" will be output after "alice" and before "fred" and its descendents. "Fred" will be output after "dave" and before "jane" and "joe". And so on.

Suppose that we want to search for a given item in a binary search tree. Compare that item to the root item of the tree. If they are equal, we're done. If the item we are looking for is less than the root item, then we need to search the left subtree of the root -- the right subtree can be eliminated because it only contains items that are greater than or equal to the root. Similarly, if the item we are looking for is greater than the item in the root, then we only need to look in the right subtree. In either case, the same procedure can then be applied to search the subtree. Inserting a new item is similar: Start by searching the tree for the position where the new item belongs. When that position is found, create a new node and attach it to the tree at that position.

Searching and inserting are efficient operations on a binary search tree, provided that the tree is close to being balanced. A binary tree is balanced if for each node, the left subtree of that node contains approximately the same number of nodes as the right subtree. In a perfectly balanced tree, the two numbers differ by at most one. Not all binary trees are balanced, but if the tree is created randomly, there is a high probability that the tree is approximately balanced. During a search of any binary sort tree, every comparison eliminates one of two subtrees from further consideration. If the tree is balanced, that means cutting the number of items still under consideration in half. This is exactly the same as the binary search algorithm, and the result is a similarly efficient algorithm.

The sample program [SortTreeDemo.cc](http://math.hws.edu/eck/cs225/s03/binary_trees/SortTreeDemo.cc.txt) is a demonstration of binary sort trees. The program includes functions that implement inorder traversal, searching, and insertion. We'll look at the latter two functions below. The main() routine tests the functions by letting you type in strings to be inserted into the tree. Here is an applet that simulates this program:

In this program, nodes in the binary tree are represented using the following class, including a simple constructor that makes creating nodes easier:

struct TreeNode {

// An object of type TreeNode represents one node

// in a binary tree of strings.

string item; // The data in this node.

TreeNode left; // Pointer to left subtree.

TreeNode right; // Pointer to right subtree.

TreeNode(string str) {

// Constructor. Make a node containing str.

item = str;

left = NULL;

right = NULL;

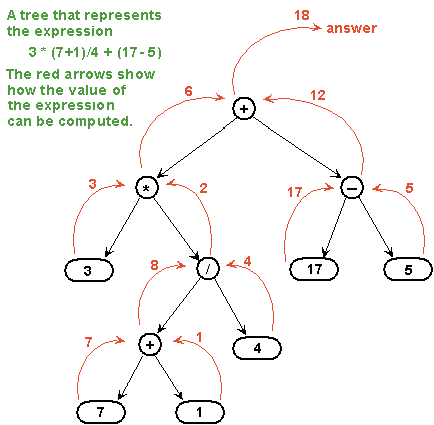
}

}; // end struct Treenode

**Expression Trees**

Another application of trees is to store mathematical expressions such as 15\*(x+y) or sqrt(42)+7 in a convenient form. Let's stick for the moment to expressions made up of numbers and the operators +, -, \*, and /. Consider the expression 3\*((7+1)/4)+(17-5). This expression is made up of two subexpressions, 3\*((7+1)/4) and (17-5), combined with the operator "+". When the expression is represented as a binary tree, the root node holds the operator +, while the subtrees of the root node represent the subexpressions 3\*((7+1)/4) and (17-5). Every node in the tree holds either a number or an operator. A node that holds a number is a leaf node of the tree. A node that holds an operator has two subtrees representing the operands to which the operator applies. The tree is shown in the illustration below. I will refer to a tree of this type as an expression tree.

Given an expression tree, it's easy to find the value of the expression that it represents. Each node in the tree has an associated value. If the node is a leaf node, then its value is simply the number that the node contains. If the node contains an operator, then the associated value is computed by first finding the values of its child nodes and then applying the operator to those values. The process is shown by the red arrows in the illustration. The value computed for the root node is the value of the expression as a whole. There are other uses for expression trees. For example, a postorder traversal of the tree will output the postfix form of the expression.



An expression tree contains two types of nodes: nodes that contain numbers and nodes that contain operators. Furthermore, we might want to add other types of nodes to make the trees more useful, such as nodes that contain variables. If we want to work with expression trees in C++, how can we deal with this variety of nodes? One way -- which will be frowned upon by object-oriented purists -- is to include an instance variable in each node object to record which type of node it is:

const int NUMBER = 0, // Values representing two kinds of nodes.

OPERATOR = 1;

struct ExpNode { // A node in an expression tree.

int kind; // Which type of node is this?

// (Value is NUMBER or OPERATOR.)

double number; // The value in a node of type NUMBER.

char op; // The operator in a node of type OPERATOR.

ExpNode \*left; // Pointers to subtrees,

ExpNode \*right; // in a node of type OPERATOR.

ExpNode( double val ) {

// Constructor for making a node of type NUMBER.

kind = NUMBER;

number = val;

}

ExpNode( char op, ExpNode \*left, ExpNode \*right ) {

// Constructor for making a node of type OPERATOR.

kind = OPERATOR;

this->op = op;

this->left = left;

this->right = right;

}

}; // end ExpNode

Given this definition, the following recursive function will find the value of an expression tree:

double getValue( ExpNode \*node ) {

// Return the value of the expression represented by

// the tree to which node refers. Node must be non-NULL.

if ( node->kind == NUMBER ) {

// The value of a NUMBER node is the number it holds.

return node->number;

}

else { // The kind must be OPERATOR.

// Get the values of the operands and combine them

// using the operator.

double leftVal = getValue( node->left );

double rightVal = getValue( node->right );

switch ( node->op ) {

case '+': return leftVal + rightVal;

case '-': return leftVal - rightVal;

case '\*': return leftVal \* rightVal;

case '/': return leftVal / rightVal;

}

}

} // end getValue()

Although this approach works, a more object-oriented approach is to note that since there are two types of nodes, there should be two classes to represent them, ConstNode and BinOpNode. To represent the general idea of a node in an expression tree, we need another class, ExpNode. Both ConstNode and BinOpNode will be subclasses of ExpNode. Since any actual node will be either aConstNode or a BinOpNode, ExpNode should be an abstract class. (We have not covered abstact classes and inheritence yet, so this will be a preview, which you can skip if you like.) Since one of the things we want to do with nodes is find their values, each class should have an instance method for finding the value:

class ExpNode {

// Represents a node of any type in an expression tree.

// This is an "abstract" class, since it contains an undefined

// function, value(), that must be defined in subclasses.

// The word "virtual" says that the defintion can change

// in a subclass. The "= 0" says that this function has

// no definition in this class.

public:

virtual double value() = 0; // Return the value of this node.

}; // end class ExpNode

class ConstNode : public ExpNode {

// Represents a node that holds a number. (The

// ": public ExpNode" says that this class is

// a subclass of ExpNode.)

double number; // The number in the node.

public:

ConstNode( double val ) {

// Constructor. Create a node to hold val.

number = val;

}

double value() {

// The value is just the number that the node holds.

return number;

}

}; // end class ConstNode

class BinOpNode : public ExpNode {

// Represents a node that holds an operator.

char op; // The operator.

ExpNode \*left; // The left operand.

ExpNode \*right; // The right operand.

public:

BinOpNode( char op, ExpNode \*left, ExpNode \*right ) {

// Constructor. Create a node to hold the given data.

this->op = op;

this->left = left;

this->right = right;

}

double value() {

// To get the value, compute the value of the left and

// right operands, and combine them with the operator.

double leftVal = left->value();

double rightVal = right->value();

switch ( op ) {

case '+': return leftVal + rightVal;

case '-': return leftVal - rightVal;

case '\*': return leftVal \* rightVal;

case '/': return leftVal / rightVal;

}

}

}; // end class BinOpNode

Note that the left and right operands of a BinOpNode are of type ExpNode\*, not BinOpNode\*. This allows the operand to be either a ConstNode or another BinOpNode -- or any other type of ExpNode that we might eventually create. Since every ExpNode has a value() method, we can call left->value() to compute the value of the left operand. If left is in fact a ConstNode, this will call the value()method in the ConstNode class. If it is in fact a BinOpNode, then left->value() will call the value() method in the BinOpNode class. Each node knows how to compute its own value.

Although it might seem more complicated at first, the object-oriented approach has some advantages. For one thing, it doesn't waste memory. In the original ExpNode class, only some of the instance variables in each node were actually used, and we needed an extra instance variable to keep track of the type of node. More important, though, is the fact that new types of nodes can be added more cleanly, since it can be done by creating a new subclass of ExpNode rather than by modifying an existing class.