Traversing Binary Trees

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.

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 inorder.

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. The 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.

**Preorder and Postorder 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 analyze algebraic expressions.

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 the following figure which 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.
A tree representing an algebraic expression.

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:

Preorder Traversal
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 previous figure 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 previous figure, 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. We use postorder traversal to delete all the nodes when the tree is destroyed.

**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. Thus about half of all searches or insertions or deletions require finding a node on the lowest level.

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, The following table shows how many levels are necessary to hold a given number of nodes.

| NUMBER OF LEVELS FOR SPECIFIED NUMBER OF NODES |
This situation is very much like the ordered array. 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

\[
N = 2^L - 1
\]

Adding 1 to both sides of the equation, we have

\[
N+1 = 2^L
\]

This is equivalent to

\[
L = \log_2 (N+1)
\]

Thus the time needed to carry out the common tree operations is proportional to the base-2 log of \( N \).

If the tree isn’t full, analysis is difficult. We can say that for a tree with a given number of levels, average search times will be shorter for the non-full tree than the full tree because fewer searches will proceed to lower levels.

Compare the tree to the other data-storage structures. In an unordered array or a linked list containing 1,000,000 items, it would
take you on the average 500,000 comparisons to find the one you wanted. But in a tree of 1,000,000 items, it takes 20 (or fewer) comparisons.

In an ordered array you can find an item equally quickly, but inserting an item requires, on the average, moving 500,000 items. Inserting an item in a tree with 1,000,000 items requires 20 or fewer comparisons, plus a small amount of time to connect the item.

Similarly, deleting an item from a 1,000,000-item array requires moving an average of 500,000 items. We haven’t investigated deletion, but it can be shown that deletion time is also proportional to the log of the number of nodes. Thus a tree provides high efficiency for all the common data-storage operations.

Traversing is not as fast as the other operations. However, traversals are probably not very commonly carried out in a typical large database. They’re more appropriate when a tree is used as an aid to parsing algebraic or similar expressions, which are probably not too long anyway.

**Summary**
- 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.