Binary Tree Overview

What to expect in this Overview

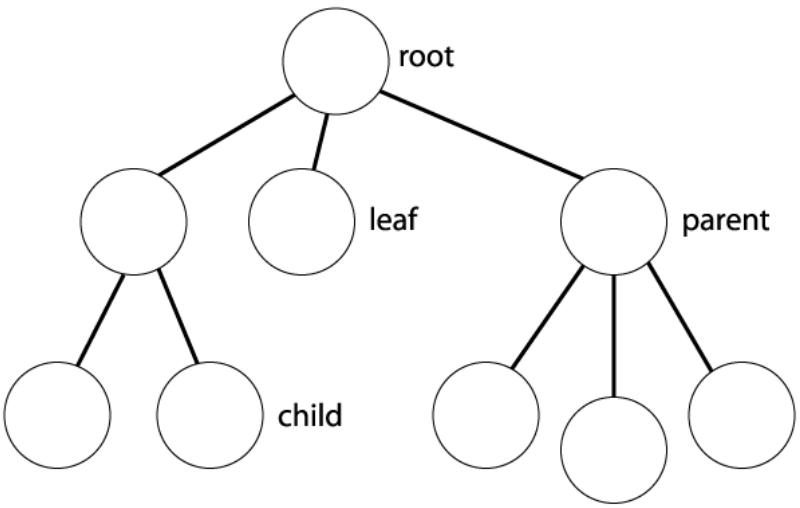
Over the next 2 weeks we will study a complex non-linear ADT called a Binary Tree. It differs from the Stack and Queue ADTs in that traversal and structure are non-linear. Simply put, whereas the elements of a stack and queue have a simple one-way path from one element to the next, a tree has a structure where each element has a concept of a parent and from 0 to N children. If a tree is a binary tree, N is up to 2 and the children are known as left and right. In this overview, the following topics will be covered

1. **The Characteristics of a Binary Tree**
2. **Binary Tree Terminology**
3. **Traversal of a Binary Tree, including pre-order, in-order and post order traversals**
4. **Representing a Binary Tree**
5. **A Binary Search Tree (BST)**
6. **Building a Binary Search Tree**

The discussion is meant to provide a foundation for you to tackle the readings in your text. Note that these discussions will provide the basis for us to discuss efficient tree structure and traversals in the week that follows this one.

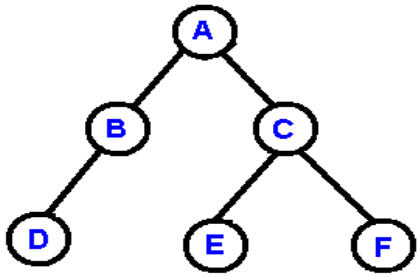
The Characteristics of a Tree

As previously stated, a tree is a series of nodes that are connected. The picture below depicts a tree that has a root (the top node). The root has 3 children (subtrees), two of which have children and one that has no children (called a leaf node).



Generic Tree with Root, Parent, Child & Leaf Depiction

A binary tree is a tree that constrains the number of children associated with a parent to a maximum of two. The graphic below shows such a depiction:



Generic Binary Tree

In this depiction:

A is the root
B and C are children (subtrees of A)
D is a left child of B
C is a right child of A and has both a left child (E) and a right child (F)
D, E and F are leaf nodes

Binary Tree Terminology

There are a number of terms that are significant in understanding tree structure. These are:

Root – In the depiction, node A is the root (top of the tree)
Degree of a node – This is the number of subtrees for a given node. In the depiction the degrees are A(2), B(1), C(2), D, E, and F(0)
Parent – A is the parent of B and C. C is the parent of E and F. B is the parent of D
Child – B and C are the children (siblings) of A. D is the child of B…
Terminal (leaf) – These are nodes that have no children. D, E, and F are leaf nodes
Moment – The moment of a tree is the number of nodes in the tree (6 in this case)
Weight – The weight of a tree is the number of leaf nodes (3 in this case)
Level – The number of branches that get traversed to get from the root to the node. In this case, B and C have a level of 1 and D, E and F have a level of 2
Height – The number of levels in the tree (one more than its highest level). In the depiction above, the height of the tree is 3

There are several other terms used to express relationships of the nodes. If the order of the subtrees is important, the tree is known as **ordered**. If trees are not connected, they are known as **disjointed**. A **forest** is simply the number of disjointed trees.

When we discuss traversal of a binary tree or the characteristics of a binary search tree, these terms will be especially important.

Traversal of a Binary Tree

Our goal will be to devise algorithms that will efficiently traverse nodes in a binary tree. We may do so for a variety of reasons such as retrieving a node and its children, searching for specific data, deleting a node or moving a node. There are three forms of traversal each of which entails visiting a node, traversing to its left (if any) child and traversing to its right (if any) child. Of course, when we traverse to a child we are now at a node, so the process simply repeats itself. This can be done both in a recursive or non-recursive manner.

Pre-order Traversal

Pre-order traversal is done in the following order:

1. Visit of the root node
2. Traverse the node's left subtree using pre-order steps
3. Traverse to the node's right subtree using pre-order steps

Of course, steps 2 and 3 really just recursively return to step 1, where the child node replaces the root node in the traversal step. You exit at step 1 if the node is NULL.

In-order Traversal

In-Order Traversal is done in the following order:

1. Traverse the left subtree using in-order steps
2. Visit the root node
3. Traverse the right subtree using in-order steps

Of course, steps 1 and 3 really just recursively return to step 1, where the child node replaces the root node in the traversal step. You exit at step 1 if the node is NULL.

Post-order Traversal

Post-Order Traversal is done in the following order:

1. Traverse the left subtree using in-order steps
2. Traverse the right subtree using in-order steps
3. Visit the root node

Of course, steps 1 and 2 really just recursively return to step 1, where the child node replaces the root node in the traversal step. You exit at step 1 if the node is NULL.

An alternate explanation of traversals can be found at an excellent Carnegie Mellon posting.

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

- *depth-first traversal*
- *breadth-first traversal*

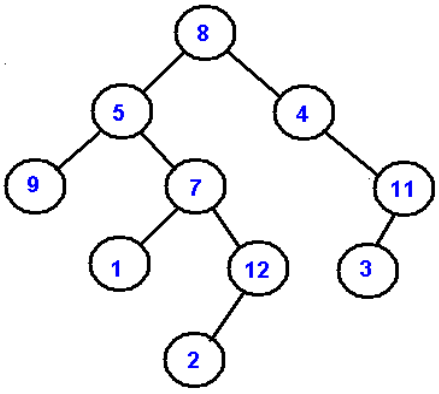
There are three different types of depth-first traversals:

- *PreOrder traversal - visit the parent first and then left and right children;*
- *InOrder traversal - visit the left child, then the parent and the right child;*
- *PostOrder traversal - visit left child, then the right child and then the parent;*

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

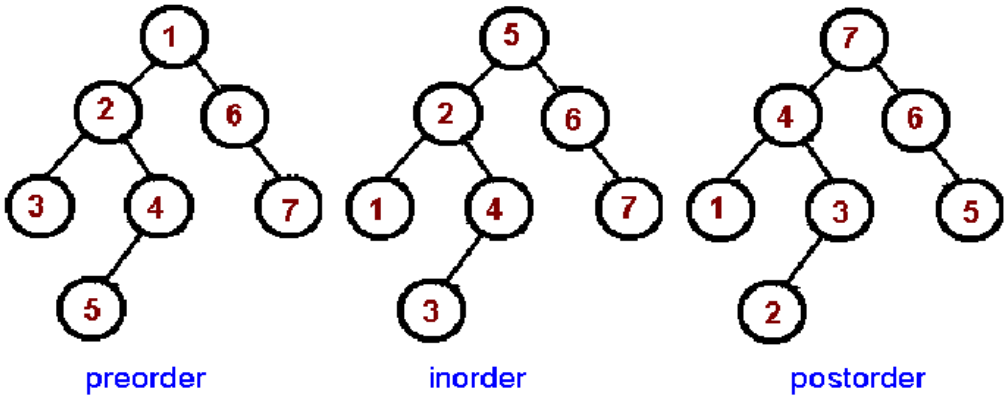
As an example consider the following tree and its four traversals:

PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2



Traversals with Node Numbers

In the next picture we demonstrate the order of node visitation. Number 1 denote the first node in a particular traversal and 7 denote the last node.



Binary Tree Representation

A Binary Tree node consists of 3 parts. They are the node data and linkages to a left side child and a right side child. Based on the underlying structure, the information could be an index into an array or pointers to left and right nodes.

The representation pattern should be familiar by now. The NodeData structure depends on the type of information to be carried within the tree. For a simple integer, it might look like this:

```
typedef struct {
        int num;
        } NodeData;
```

We also need a structure to represent a node:

```
typedef struct treenode {
        NodeData data;
        struct treenode *left, *right;
        } TreeNode, *TreeNodePtr;
```

Note that because C is a single pass compilation system, these declarations would need to be presented in the order shown, since treenode needs the definition of NodeData;

One further structure is needed and it is responsible for defining the structure that contains the root of the tree:

```
typedef struct {
        TreeNodePtr   root;
        } BinaryTree;
```

How you construct the tree using these structures (all left side children, all right side children, alternating left, all right children) are just some of the possibilities. The point to remember is that each node needs to have either an allocated treenode or NULL to indicate the absence of a left or right child.

Your text has an excellent example of building a tree from an input where a tree has data that is an array of characters and an '&' represents a NULL. Once the tree is built, the tree is traversed in pre-order, in-order and post-order format. When you go through the example, pay particular note to the traversals as they demonstrate the ordering previously covered.

The example uses recursive software to traverse the tree. As a reminder, a recursive function needs to call itself while a condition is NOT satisfied. In our case, the exit condition is a NULL at a left or right child.

Binary Search Trees (BST)

In the section just described, a tree was built that had no relationships between a node's value and its children. This would not have been an ordered tree. To construct a tree that optimizes searching, we would like to construct a tree such that we can make decisions on where data can reside by applying a relationship between a node and its children. That is what a Binary Search Tree is all about!

Building a Binary Search Tree

The relationship within a Binary Search Tree is simple:

> *The value of a left child is guaranteed to be less than the node value, which will be less than the value of the right child.*

Note that this constraint implies that no two nodes can have the same value. If you need to have multiple instances of the same node data value, maintaining a count of the occurrences of the node data value will prove a simple and effective way to implement the requirement. In fact, the text example that counts the frequency of words is precisely the technique used.

Where are we at the end of this discussion?

We have covered a sufficient amount of information for one week to absorb! Be sure and look through section 5.7 of your text for an example of finding / building nodes and inserting them into a Binary Search Tree.

Note that so far the tree we have constructed may be far from optimal, but searching it has suddenly become MUCH faster. To find a value we can discount half of the remaining choices on a tree (or subtree) with each node visited because we have gone to the trouble of placing the data such that it maintains the relationship noted above.

Next week, we will cover an alternate traversal mechanism, including linkages that allow us to better traversals. We will learn how to build optimized Binary Trees and how to maintain them.

Enough for another day…