Trees - for Programming Contests

Quick Intro

A general tree is a connected graph with n vertices and n-1 edges, with no cycles. The latter condition can be proven to be redundant. A rooted tree is one where a particular vertex is designated as a "root". A binary search tree is a rooted tree. (Note that if we choose a different "root" in a BST, the BST property no longer holds. Also, in a BST, the neighbors of a vertex have "directions" associated with them, while in a general tree, this isn't the case.)

Binary Trees

Binary Trees are typically taught in a standard data structures class. In class, you are introduced how to store data in a binary search tree, showing how to do insert and delete operations. Some other standard problems you learn to solve are:

1) Pre-order, In-order and Post-order traversals
2) Calculating the sum of all the values in the tree
3) Counting the number of leaf nodes in the tree
4) Calculating the height of a tree.
5) Determining if each node in the tree is balanced (AVL tree property)

Each of these tasks take O(n) time, where n is the number of nodes in the tree.

The insert and delete operations take O(log n), on average, as long as the tree is reasonably balanced. But, in a programming contest, unless the input restricts the structure of a tree otherwise, you can't assume the tree will be well-balanced.

Rooted Tree Exploration

Typically, in a rooted tree where each node can have an arbitrary number of children, we explore our nodes either in a pre-order traversal or post-order traversal. A pre-order traversal visits the root first, then recursively visits each subtree. A post-order traversal first recursively visits each subtree and then visits the root. For binary trees, an in-order traversal is defined where the root is visited after the left subtree, but before the right subtree. In most problems involving trees in programming contests, we'll typically use one of these traversal techniques. Many times, a post-order visiting of the nodes occurs. We first solve the problem at hand for each subtree, recursively. Then we take all of those answers and combine them to solve the given problem at hand. We can think of a recursive function that finds the height of a tree to fit this paradigm:

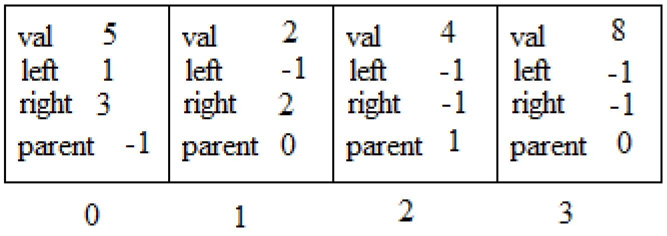
```
int height(tree* root) {
    if (root == NULL) return -1;

    int res = 0, i;
    for (i=0; i<root->numChildren; i++) {
        int cur = height(root->child[i]);
        if (cur+1 > res) res = cur+1;
    }
    return res;
}
```

For this specific problem, we simply want to find the height of all the subtrees of our root, take the maximum of these values and add 1 (for the extra link from the root to the maximal subtree).

Binary Tree Storage

A node for a binary tree typically contains some data, a pointer to a left node and a pointer to a right node. However, many times in programming contests, due to how the data is expressed, we can store each node in an array and our left and right pointers are just integers which tell us which indexes store the left and right child nodes, respectively. Also, sometimes it's useful to store a link to the parent of a node. Here is a picture of how a binary tree with four nodes (root 5, left child 2, right child 8, and left-right child 4) might be stored, with the root stored at index 0:



val	5	val	2	val	4	val	8
left	1	left	-1	left	-1	left	-1
right	3	right	2	right	-1	right	-1
parent	-1	parent	0	parent	1	parent	0
0		1		2		3	

Thus, if we store the data in this manner, a null link is represented with -1 and left and right aren't references (in Java) or pointers (in C++). Also, we can still keep the array representation but still have references/pointers for left and right if we want. Which way you do it is largely personal preference. Finally, notice that the nodes in the array may not be in any particular order. It's the left and right indexes which tell you where to go and these can go anywhere in the array. If you had this representation, but didn't know where the root was, just find the node with the parent link set to -1.

One key item to note is that many standard solutions to contest questions will get Time Limit Exceeded if they require exploring a full path in the tree for every query. There are complicated ways to get around this, but sometimes there are easier ways to make your code much faster. In particular, the easiest way to speed up binary tree code tends to be to store extra information in each node. For example, if we store the height of each node in the node, then we don't have to run a height function each time on a particular node to obtain its height. Instead we have a node's height in O(1) time and just have to maintain all heights after each insert or delete to the tree. (This isn't as hard as it sounds, most of the time…) Also, in addition, depending on the problem, it may be useful to store even more information in each node, so that you can look that information up immediately instead of recalculating it each time you need it. In general, when you choose to store more data in a node, then you have to maintain that data each time it changes. Typically, the extra time it takes to maintain this data amortizes to be just fine.

Typical Structure of a Solution to a Binary Tree Problem
The typical structure to a binary tree problem is as follows:

```
public static int solve(node* root, ...) {

    if (root == null) return ...;

    int left = solve(root.left, ...);
    int right = solve(root.right, ...);

    // Combine answers from left and right somehow.
    // return answer
}
```

In short, the easiest way to conceive of these problems is to solve them recursively on each subtree, then figure out how to use those solutions to the subtrees to build a solution to the greater problem.

Let's look at two sample problems where we utilize this sort of strategy. In particular, pay attention to how we have to customize the general strategy to fit our specific problem. (The customizations for trees can look quite different, from problem to problem, but the same over-arching idea generally exists.)

Retrieving a Post-order Traversal
Problem: Given both a preorder and inorder traversal of a binary tree (not necessarily a search tree), produce the corresponding postoder traversal.

Let's consider a sample binary tree:

```
              10
           /       \
         3           14
       /  \         /   \
      8    7        1     9
          / \     / \   / \
         6   12  47 13 18  22
                          /
                         5
```

Its pre-order traversal begins with 10, follows with the left sub-tree of 10, and then follows with the right sub-tree of 10.

Its in-order traversal begins with the left sub-tree of 10, followed by 10, followed by the right sub-tree of 10.

So, if we are given both of these traversals, we should be able to pick out the root of the tree by finding the first item in the pre-order traversal. Then, we should find that value in the in-order traversal. All of the values that come before this value (10 for the tree above) are precisely the values in the left sub-tree.

In a nutshell, if the problem we are solving takes in both lists (preorder, inorder) and produces the list postorder, here is what we have:

```
producePost({10, 3, 8, 7, 6, 12, 14, 1, 47, 13, 9, 18, 22, 5},
            {8, 3, 6, 7, 12, 10, 47, 1, 13, 14, 18, 9, 5, 22}) =

producePost({3, 8, 7, 6, 12}, {8, 3, 6, 7, 12}) +
producePost({14, 1, 47, 13, 9, 18, 22, 5}, {47, 1, 13, 14, 18, 9, 5, 22}) +
{10}
```

In this pseudocode, the plus sign represents list concatenation. The two green pieces highlight the pre-order and in-order traversals of the left sub-tree of 10 while the two orange pieces highlight the pre-order and in-order traversals of the right sub-tree of 10. Since we can identify each of these lists right after locating 10 (root) in the inorder list, we have enough information to make the appropriate recursive calls to get the post-order traversals of each subtree. Once we have that information, we can put together our final answer using both post-order traversals and our root.

In code, we obviously won't be highlighting lists like this. Instead, in code, we realize that the length of both traversals will be the same, but the starting position of each traversal may be different. (Notice that the green traversals start at different indexes within their larger lists.) So, our recursive function will take in this extra information: starting index in the pre-order traversal, starting index in the post-order traversal and the length of the traversal. Thus, if the input to the function were the two lists above with a starting index of 1 for the pre-order traversal, a starting index of 0 for the in-order traversal, and a length of 5, then our function should return the list: 8, 6, 12, 7 and 3, the post order traversal of the left of 10. The accompanying C code is in the posted file PrintPostOrder.c

Maximum Cash Problem

Consider a problem where you are given a binary tree of integers and are asked to retrieve the maximum sum of any subset of nodes such that no node in the subset is an ancestor of another node. For example, in the tree below, a valid subset is highlighted:

```
            10
          /      \
        3          14
       / \        /   \
     8    7       1     9
         / \     /\    / \
        6   12  47 13 18  22
                          /
                         5
```

Since we chose 8, automatically 3 and 10, the ancestors of 8 are not allowed in our choice. Similarly, if we choose 47, we can not choose 1, 14 or 10.

Viewing this problem recursively, here is a sample C solution:

```
int maxValue(nodeT* root) {

    if (root == NULL) return 0;

    int leftSide = maxValue(root->left);
    int rightSide = maxValue(root->right);

    if (root->cash > leftSide + rightSide)
        return root->cash;
    return leftSide + rightSide;
}
```

Basically, if we were to choose the root node in our subset, we aren't allowed to have ANY nodes below it, so our answer could just be what's stored in the root node. Alternatively, if we don't choose the root node, then we are free to add our recursive solutions from the left and right, since it's impossible for any nodes on the left to have ancestors who are on the right and vice versa.

General Trees

The only difference between a regular tree and binary tree is that a regular tree node is allowed to have any number of children. In code, this means instead of having left and right pointers, we have a list of pointers. In Java, we might have one of the two following options.

```
class node {                        class node {

    public int data;                    public int data;
    public node[] kids;                 public int[] kids;
    public node parent;                 public int parent;
    ...                                 ...
}                                   }
```

A few things to note: In programming contests, we tend to make everything public, which is typically a horrible practice in software engineering. This is so that we have easy access to everything via more succinct code. I use the variable name "kids" because "children" is more typing; I have no other reason than that. These representations assume you know the number of kids a node has when you create it. If you don't, then you would use `ArrayList<node>` or `ArrayList<Integer>`. Then, we could easily add kids dynamically to an existing node.

Typical Structure of a Solution to a General Rooted Tree Problem

The typical structure to a general tree problem is as follows:

```
public static int solve(node* root, ...) {

    if (root == null) return ...;

    int[] tmpAns = new int[root.kids.size()];
    for (int i=0; i<kids.size(); i++)
        tmpAns[i] = solve(root.kids.get(i),...)

    // Combine answers from all subtrees somehow and
    // return answer
}
```

Notice that this structure isn't terribly different than the binary tree code structure. The only difference is that successive recursive calls to the left and right are replaced with a for loop of calls to each kid.

Normal Form Problem - Alternate (Easier) Version

In the Normal Form problem, we are given a boolean expression in disjunctive normal form to evaluate at a fixed truth setting. The boolean expression can be viewed as a tree, with an and/or operator in all non-leaf nodes and a T/F value in each leaf node. The order of operations is naturally dicated by the tree structure. In this original problem statement however, which operators are where in the tree is initially unknown. First, before we tackle that problem, let's do a slightly different an easier version where the operator at each node is known.

Let's assume that the operator at the top level is an and and that the operator below any and node is an or and vice versa. In the problem, the input is represented as a string where parentheses denote a subtree. For example, the string `((F(TF))(TFT))` represents the Boolean expression:

```
((F || (T && F)) && (T || F || T))
```

In code, our recursive function could take in what level of the tree we are in (0 for root), and then it would identify each subtree by looking for pairs of matching parentheses. In this case there are two pairs of matching parentheses at the top level. So, the problem would be recursively solved for the expressions `((F(TF))` and `(TFT)`. Then, we'd just take both results and take the and of them. (If we were on an odd level we'd take the or of our results.) In the base case we just return the primitive value given, or the and/or of the list of primitives given.

Normal Form - Coding Detail

Since the input isn't given in a tree form, our recursive function might have difficulty with kid pointers. Instead, what the recursive function will take in is start and end indexes into the whole input expression. For example,

```
public static boolean solve(String exp, int sI, int eI, int level);
```

would solve the problem for the expression `exp[sI...eI]` where level denotes how far down the whole tree this sub-expression is. (If level is even, we take the and of each subtree, if level is odd, we take the or of each subtree.)

Normal Form Problem

In the actual problem however, you don't know what the root operation is. It could be an and or an or. Rather, you are told that the operator at the deepest level of the tree is an and operation. So, we are forced to go through the tree initially to figure out its depth. Once we know its depth, we know whether or not the top operator is an and or and or, and can process accordingly.

Normal Form - Alternate Solution Idea

In the posted solution, c.java, rather than using recursion at all, I utilize a stack to process the input. Here, whenever a matching close paren comes into play, I just pop off all values and take the and or or of them, appropriately. When we finish processing our expression, the stack is left with the result of the operation.

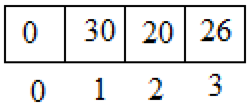
Family Fortune Problem
This problem is similar to the original problem discussed in these notes, but is more general. Instead of the input being a binary tree, it's a general tree. Also, instead of considering ANY subset with no pair of nodes where one is the ancestor of another, we only want to consider subsets of size k exactly.

The general tree issue isn't too difficult to deal with. Rather, we can simply handle this by summing up the results of the recursive calls to each sub-tree and comparing this to the root data. If the root data is greater than the whole sum, we go with it, otherwise, we go with the sum of the kids.

The real difficulty is dealing with the restriction of a subset of size k. Now, instead of each recursive call returning to us a single value representing the maximum wealth (sum of values in nodes without a node and one of its ancestors) period, we actually need the recursive call to report to us the maximum wealth for each possible subset size. Consider the following tree:

```
          10
        /      \
      30         14
    /  \        /   \
   8    7      61     7
       / \    / \   / \
      6  12  47 13 18  5
                       /
                      22
```

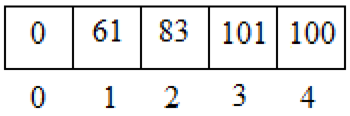
For this tree, if we make the recursive call on node 3, our return value would be the following array:



0	30	20	26
0	1	2	3

We can see that if the tree rooted at 3, if we were to select only 1 node, the maximum we could select is 30, the root. If we were to select any two (such that one wasn't the ancestor of the other), we could choose 8 and 12, which are in different subtrees. (We can't choose 30 anymore.) Finally, if we were to select any 3, the best we could do is select 8, 6 and 12, since 7 can't be selected due to 12's selection.

A recursive call on the tree rooted at 14 would yield the following return value:



0	61	83	101	100
0	1	2	3	4

Now, to solve the problem recursively using these two solutions, we have to combine them into one array, where result[k] is the maximum of array1[i] + array2[j], where i + j = k. Namely, we must find the way to use precisely k nodes such that we maximize the sum of nodes possible.

We do this as follows: we keep pointers i and j into the two arrays, respectively, and see whether it helps us more to add one to i or j. Obviously, when i and j are both 0, it's better to increment j since this gives us an increase of 61. Next, it's best for us to increase i because that leads to an improvement of 30 while increasing j would have lead to an improvement of 22 only. After this, increasing j to 2 is best as that raises our total (instead of decreasing it). Completing this merging of the two arrays yields the following array:

0	61	91	113	131	130	120	126
0	1	2	3	4	5	6	7

Note that this only works due to the structure of the problem. If we had two completely arbitrary arrays, we'd have to run a double for loop structure to do this merge, considering each possible value array1[i] + array2[j] as i ranged from 0 to k, inclusive. In the solution presented above, we assume that if the optimal result for k is taking i items from the first tree and j items from the second tree, then the optimal result for k+1 is either taking i+1 items from the first tree and j items from the second tree OR i items from the first tree and j items from the second tree.

Final Note about Recursion and Trees

A deep tree may result in many nested recursive calls. On a typical Java system, the maximum depth of the call stack is set relatively low. This means that for contest problems, one would get "Run Time Error" for a standard algorithm coded in Java running on a tree of depth 10,000 or so. The exception to this is if the judge changes the default stack size flag for Java. Our judge does, but many other judges (USACO is one example) do not. If you code in C++ this does not become an issue because the call stack is allowed to be quite deep by default in C++. Two ways around this issue if you still want to use Java are:

1) Write your code iteratively, getting rid of the recursion. (This can be messy sometimes, but I've found that many times, you can just run a while loop and keep a stack of where you are in your tree.)

2) There is a "stack trick" that works. You set your program to implement Runnable (for threads) and then you do this in your main:

```
public static void main(String[] args) throws Exception {
     new Thread(null, new item(), "solver", 1L<<29).start();
}
```

The last item indicates the maximum stack size. You might have to play around with its actual value to make sure you don't run out of memory. Finally, just put your actual code in the method run, which a try-catch thrown around it. (I found this doesn't work on all online judges though.)