Traversing Binary Trees; Non-Recursive Traversals; Deletions in Binary Trees and General Trees

Print Sequences and Binary Trees
As we've seen, a binary tree produces specific preorder, inorder, and postorder print sequences (lists of node-labels). What about the other way around? Does a particular print sequence specify a particular binary tree?

Suppose that ABC is the preorder sequence for some binary tree. We can see that A must be the root, but the sequence doesn't tell us if B is the left child of the root, or the right child. If ABC were the inorder sequence for some tree, we don't even have a clue as to what label is at the root node. And if ABC were a postorder sequence, aside from seeing that C must be at the root, we don't know if A is the left child or the right child of the root.

Thus, no single sequence (of the 3 standard sequences) can uniquely specify a binary tree. What if 2 sequences were given? Suppose CAB is the preorder sequence, and BAC is the postorder sequence. Aside from both sequences "telling us" that C is at the root, we still don't know if A is the left child of the root, or the right child - and whether B is the left child of A or the right child, so this pair of sequences won't specify a unique binary tree.

At this point, it should be clear that we need the inorder sequence and either preorder or postorder. The inorder sequence will resolve the left-right problem, and the other sequence (pre or post) will tell us the roots of the various subtrees. For example, if inorder is CBA, and preorder is CAB, then we know that C is at the root, and both B and A are in the right subtree - the left subtree of the root is empty. Since A comes next in the preorder sequence, it is the root of the right subtree, and - looking at the inorder sequence - B is its left child, and we have a unique tree. You should try this (recursive) process on several examples. Sketch a binary tree with 8-10 nodes and (unique) labels, determine the inorder sequence and - say - the preorder sequence. Now try to reconstruct the tree using the method outlined above. Then repeat the process with the inorder sequence and the postorder sequence of the tree: see if you can "regenerate" the original tree.

General Trees and Forests We define a general tree as the root of a forest, and a forest is an (ordered) set of zero or more general trees. This mutually recursive definition of general trees and forests allows us to talk about trees where nodes might have more than two children. The children of a node (the trees of the forest that it roots) are sequenced: the first, the second, etc. There is no sense of left and right in
general trees, except we typically draw the tree with the sequential ordering from left to right.

Notice that a general tree must have a root (in contrast to a binary tree), and that a forest may be empty (it is a set, and sets can be empty).

We can extend our notion of preorder and postorder traversal to general trees: the preorder traversal of a general tree visits the root, and then traverses the first subtree (if there is one) in the forest (that the root is the parent of) in preorder, then traverses the second tree (if there is one) of the forest in preorder, etc. And similarly for postorder. Try some examples of these traversals on a general tree where say - the root has 3 children (the forest below the root contains 3 general trees), and each of the children is the root of a general tree - of your choosing. (Make sure to have several nodes with more than 2 children, and notice that nodes with one child should be drawn with the child directly below the parent - there is no left and right child as in a binary tree.)

A binary tree is easily and uniformly represented in a programming language as a record with a data field (for the label, say), and pointer fields for left and right pointers. Every node record is the same; the data and pointer values capture the labels on the nodes and the structure of the tree. What about trees where nodes can have more than two children? Can we provide say, records with 1000 pointer fields in order to be able to represent general trees where nodes might have up to 1000 children? In any given case, most of the fields of all the node records will be unused, so a great deal of memory is required - and wasted. And what if we want to introduce a node with 1001 children? We have to reconfigure the implementation to accommodate this one “anomalous” node. Surely, this is not satisfactory.

Remarkably, a forest (and therefore, a general tree = a one-tree forest) can be represented by a binary tree. Obviously, an empty forest can be represented by the empty binary tree. Now think of a non-empty forest as comprised of 3 parts: the root of the first general tree in the forest, the first child of that root, and - the remaining trees in the forest (if any). Then we can use a “standard” binary node record to hold the label of the root of the first general tree, use the left pointer of the binary node to point to the first child of that root, and the right pointer of the binary node to point to - the root of the first general tree in the rest of the forest (the 2nd general tree in the forest). And we continue this process recursively. Observe how the general tree (the one-tree forest) below is represented by a binary tree:
Since A is the only tree in this forest, the right pointer in the binary tree representative (BTR) is NIL. A’s first child is B, so the left pointer of A in the BTR points to B. So we’ve used up the fields of the node record for A in the BTR. Now what about B (in the forest)? Its first (only) child is E - so we use the left pointer of B in the BTR and make E its left child - while the root of the first general tree in the rest of the forest (the two trees which have C and D as roots) is C, so we use the right pointer of B in the BTR to point to C. And we continue this recursively.

Is it clear to you why C (in the BTR) has no left child? And why D (in the BTR) has no right child? And can you reverse this representation process by recovering the general tree (or forest) from the BTR? Try several examples to be sure that the process is familiar to you.

This representation method is valuable already because it permits us to build binary trees (implemented in the standard, uniform way) which capture all the information about quite arbitrary forests of general trees. But there is even more value to the BTR (the standard representation, so-called). It happens to be true that: the preorder traversal sequence of the BTR is the same as the preorder traversal sequence of the forest it represents, and the inorder traversal of the BTR is the same as the postorder traversal of the forest it represents! Notice, for example, that the preorder sequence for the (one-tree) forest above is ABECDFGH, the same as the BTR, and the postorder sequence for the forest is EBCFGHDA - the same as the inorder sequence for the BTR. (Test this out on some of your own examples. It’s also a good exercise in gaining competence with the traversal methods.)

Thus, a forest can be described by its preorder and postorder traversal sequences, and from that information, we can build a binary tree to represent the forest! (How?) The BTR can then be processed to answer questions about the structure of the forest.

Non-recursive Binary Tree Traversals
Are the (preorder, inorder, and postorder) traversals of binary trees actually different processes? Consider the non-recursive algorithm below which uses an explicit stack S to direct the traversal of a binary tree. The stack items are objects containing two fields: a reference node to (the root of) a binary tree, and an integer phase. Initially, the stack is empty. The value of the integer phase will determine which of three operations are performed at the node that is currently being processed by the algorithm. Assume that a binary (search) tree has been built, and that bt is a reference to it.

```java
class StackItem {
    protected BtNode node;
    protected int phase;

    public StackItem(BtNode n, int i) {
        node = n;
        phase = i;
    }

    // setters and getters for the instance variables
}

public void universalTraverser() {
```
// invoked by BinTree object bt

        Stack S = new Stack();
        S.push(new StackItem(this.root, 1));

        while(S.isEmpty() != true) {
            StackItem si = (StackItem) S.pop(); // pop returns Object
            if (si.getNode() != NULL)
                switch(si.getPhase()) {
                case 1:
                    S.push(new StackItem(si.getNode(), 2));
                    // DO FIRST OPERATION
                    break;
                case 2:
                    S.push(new StackItem(si.getNode(), 3));
                    // DO SECOND OPERATION
                    break;
                case 3:
                    // DO THIRD OPERATION
                    break;
                }
        }

        If "first operation" is System.out.print(si.getNode().getKey()), "second operation" is S.push(new StackItem(si.getNode().getLeft(), 1)), and "third operation" is S.push(new StackItem(si.getNode().getRight(), 1)), then we generate the preorder print sequence. What rearrangements are necessary to generate either an inorder sequence or a postorder sequence? Note that each binary tree node is pushed - and popped - exactly three times corresponding to the values of phase, and that the nodes are explored in exactly the same sequence for each traversal. In fact, the "background" sequence in which the nodes are visited is preorder. How could it be otherwise if - in each case - we do left before right, and binary trees have pointers only to roots of subtrees?

Suppose we modify the method above so that it has a parameter String s whose value is passed by the user when the method is invoked. That is, s can have one of the values "pre", "in", or "post". Can you see how to write some elementary decision logic to enable the code to process any of the three traversal sequences, according to the user specification?

**Deletions in Binary Trees** The principle for deletion is to preserve the inorder sequence of the node labels. That is, the inorder sequence after the deletion should be the same as it was before - minus the label for the deleted node. There are three cases to examine:

- The node to be deleted is a leaf (no children): we simply "snip" it off from its parent, that is, make the reference to the node a null reference. Of course, this requires that we have a reference to the parent node. Inorder is preserved, since the deleted node had neither left subtree or right subtree, and thus contributed only its label to the inorder sequence.

- The node to be deleted has one child (which may be the root of an elaborate subtree): Since the subtree is processed inorder before/after (depending on whether it's the left or right subtree) the node to be deleted, we simply "hoist" the subtree into the position of the node that is to be deleted, thereby preserving inorder. Once again, we'll need a reference to the parent of the deleted node so we can alter one of its member fields. Suppose that d is a
reference to the node marked for deletion, that \( p \) is its parent, and that \( c \) is the (single) child of the "doomed" node. Then we would write:

```java
if(p.getLeft() == d) p.setLeft(c);
else p.setRight(c);
```

- The node to be deleted has two children: We can "replace" this node with either its inorder predecessor (the node just previous to it in the inorder sequence) or its inorder successor (the node just after it in the inorder sequence). Where do we find the predecessor(successor) node? It is the rightmost(leftmost) node in the left(right) subtree of the node to be deleted. And since it is the leftmost(rightmost) node in its subtree, it has at most - one child! Thus, we swap the information at the node to be deleted (key, datapointer) with that at its predecessor (or successor), and then resolve the deletion from that point with the methods above.

Examples: The tree below has inorder sequence DBHGIEACF. To delete H, we "snip" its pointer from G, and the resulting tree has inorder sequence DBGIEACF. To delete E (from the tree below), since it has one child we arrange for B.R to point to G, producing the tree on the right, with inorder sequence DBHGIACF, as desired.

```
A
|-- B
     |-- D
     |    |-- G
     |     |-- H
     |        |-- I

A
|-- B
     |-- D
     |    |-- G
     |     |-- H
     |        |-- I
```

Finally, to delete A, we can replace it with its inorder predecessor (E) or with its successor (C). In the former case, we have

```
E
|-- B
     |-- D
     |    |-- G
     |     |-- H
     |        |-- I

E
|-- B
     |-- D
     |    |-- G
     |     |-- H
     |        |-- I
```

with inorder sequence DBHGIACF (A is gone). In the latter case we have

```
C
|-- B
     |-- D
     |    |-- G
     |     |-- H
     |        |-- I

C
|-- B
     |-- D
     |    |-- G
     |     |-- H
     |        |-- I
```
with inorder DBHIGECF, as desired.