Massachusetts Institute of Technology — Handout 18
6.042J/18.062J: Mathematics for Computer Science — February 25, 1994
Professors Charles E. Leiserson and Tom Leighton

Solutions to Problem Set 3

Problem W3-8. Tree traversals.

(a) The leaf nodes are processed in the same order, namely left to right, by all three methods of tree traversal. This is because each method processes a left child before a right child.

(b) The idea was to use the stack to automatically keep track of which nodes have been processed and which haven't, without any extra mechanisms. The stack always holds those nodes from which we have explored the left child but not the right. Popping the stack takes you back to the most recently visited node whose right child hasn't been visited yet; this node should now be processed and will not be visited again. The stack is popped when we "run off the end" of the tree by setting x to a left or right child that does not exist (*nil*).

```
INORDER(T)
x ← root(T)
until stack is empty and x = nil
    if x ≠ nil
        PUSH(x)
        x ← left(x)
    if x = nil
        x ← POP
        PROCESS(x)
        x ← right(x)
```

The stack is certainly never larger than the height h of the tree (which is at most the size of the tree), so the asymptotic running time is $\Theta(h)$. Even better, observe that when we're at node x,the stack holds only those nodes on the path from the root to x that have a left child on the path (not those that have a right child on the path), so the stack is never larger than the greatest number of left-children in any path.

(c) (This problem should have asked to prove the statement for $u \neq v$, since it's unpleasant to say that u precedes and follows itself in an ordering.)

We will prove that if u and v are nodes in a binary tree with $u \neq v$, then u is an ancestor of v if and only if u precedes v in preorder and u follows v in postorder. We will say v is a **descendant** of u if u is an ancestor of v. Notice how our proof follows only from the definitions given.

First, suppose u is an ancestor of v. Then from the root, v cannot be reached before u because u is on the unique simple path from the root to v. When u is reached in preorder traversal, it is processed immediately, in particular before v since v has not yet been reached. When u is reached in postorder traversal, all descendants of u are processed first, so u is processed after v.

The other direction is more interesting. Suppose u precedes v in preorder and follows v in postorder. We have four cases based on the possible ancestor relationships of u and v (both, neither, or exactly one is an ancestor of the other). We can immediately rule out one case: u and v cannot be ancestors of each other since $u \neq v$. Since u precedes v in preorder, our argument above says that v cannot be an ancestor of u (the second case). We would like to show that u is an ancestor of v (the third case). The only remaining case is that neither u nor v is an ancestor of the other. We must show this is not possible. Suppose it were true and consider the unique simple paths from the root to u and from the root to v. Let x be the last node these paths have in common; x cannot be u or v since neither is an ancestor of the other. All descendants of $\mathit{left}(x)$ precede all descendants of $\mathit{right}(x)$ in both preorder and postorder. So if u is a descendant of $\mathit{left}(x)$, then v is a descendant of $\mathrm{right}(x)$ and u precedes v in postorder; similarly, if u is a descendant of $\mathrm{right}(x)$ then v precedes u in preorder. Either way, we have a contradiction, and therefore this case is impossible. The only possible case is that u is an ancestor of v.

Problem W3-9. Well-formed string of parentheses and brackets.

(a) Process the string from left to right. When a left parenthesis or bracket is encountered, push it on the stack. When a right parenthesis or bracket is encountered, pop the stack (if the stack was empty already, reject) and compare the types of the two symbols—if they are different, reject, else continue. Accept if the stack is empty after the last symbol of the string is processed.

(b) A correctness proof should show that well-formed strings are accepted and non-well-formed strings are rejected. We will prove (more than a "sketch") the equivalent statement that the algorithm accepts **if and only if** the input string is well-formed.

First, the "if" direction (the algorithm accepts well-formed strings): Proof by induction on the length of the string. There are no well-formed strings of length 1, and it the only well-formed strings of length 2 are ( ) and [], which are both accepted. Suppose w is a well-formed string of length $n > 2$. Then w is either $(\, v\,)$ or $[\, v\,]$ or $u\, v$ where u and v are well-formed. When processing $(\, v\,)$ it will not pop the leading (off the stack while processing v because the algorithm does not try to pop an empty stack when processing v because by induction it accepts v. Since the stack would have been empty on input v, it will hold only (after processing $(\, v$ and the final parenthesis will properly match, emptying the stack. Similarly, the algorithm accepts $[\, v\,]$. Finally, it is easy to see that the algorithm also accepts $u\,v$ since by induction the stack will be empty after processing the well-formed string u.

Now, the "only if" direction (the algorithm accepts only well-formed strings): Again by induction on the length of the string. We will prove that if the algorithm accepts, then the input string is well-formed. Clearly the only strings of length at most 2 that are accepted are ( ) and [], which are both well-formed (let us say that the empty string is also well-formed). Suppose the algorithm accepts a string of length $n > 2$. If the stack is empty after reading some prefix u of w (where u has nonzero length), then we can write $w = u\,v$. Since the stack is empty after u, the algorithm would accept both the input u and the input v. By induction, u and v are therefore well-formed, and by definition therefore so is w. If the stack is never empty until the end of processing w (when it must be empty), the first symbol pushed is not popped until after the last (matching) symbol of w is read. We can therefore write $w = (\,v\,)$ or $w = [\,v\,]$. Since the first symbol of w is not popped during the processing of v, it is easy to see that the algorithm would also accept the input v. By induction, therefore, v is well-formed, and by definition, so is w.

(c) $\Theta(n)$, where n is the length of the string.

Problem W3-10. Box comparisons.

(a) There are many ways to do this. One way is to compare each untested box with the heaviest box seen so far.

(b) Consider the directed graph with an edge from node u to node v if a direct comparison showed that box u was heavier than box v. If there are fewer than $n-1$ edges then the graph is not connected. The nodes of the graph can therefore be partitioned into two disjoint nonempty sets V_1 and V_2 such that there is no edge going from a node in one set to a node in the other. It is therefore possible that the boxes in V_1 are all heavier than those in V_2 or vice versa, and so the heaviest box has not been determined.

(c) Compare the boxes as in a single-elimination playoffs tournament like Wimbledon—create a full binary tree with n leaves, as balanced as possible so the depth is $\lg n$, and compare siblings, working your way from the bottom of the tree to the top. First determine the heaviest (the winner of the tournament, at the top of the tree). The second-heaviest box can lose only to the heaviest, so then determine the heaviest among the $\lg n$ boxes that lost to the heaviest. The total number of comparisons is $n - 1 + \lg n - 1$.

(d) Again, the tournament structure helps. This time, we know that the lightest box must have lost in the first round. So after determining the heaviest, determine the lightest of the $\lceil n/2 \rceil$ boxes that lost in the first round. The total number of comparisons is $n - 1 + \lceil n/2 \rceil - 1$.