CMSC351 - Fall 2014, Homework #2

Due: October 8th at the start of class

Name:
Section:

- Grades depend on neatness and clarity.
- Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. You should ALWAYS prove the correctness of your algorithms either directly or by referring to a proof in the book.
- Write your answers in the spaces provided. If needed, attach other pages.
- The grades would be out of 100. Four problems would be selected and everyones grade would be based only on those problems. You will also get 25 bonus points for trying to solve all problems.

Problem 1 Suppose you are given an expression with n paranthesis. Design an algorithm with running time $O(n)$ to check whether it is valid. Examples of valid expressions are (()())()() and ()()((((()))))). Examples of invalid expressions are ())()(and (((()()))(().

Solution: Use a stack. For each *open paranthesis* push it into the stack. For each *close paranthesis* pop stack. If at some point the stack should be popped but it is empty the expression is invalid. If at the end of expression the stack is nonempty, the expression is invalid. Otherwise it is valid.

*N*ote: The whole process can also be done by incrementing and decrementing a counter as well.

Problem 2 Function $T(n)$ is defined by the following recurrence relation.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor \sqrt{n} \rfloor) + 1 & \text{if } n > 1 \end{cases}$$

Prove $T(n) = O(\log n)$.

Solution: We prove this by Substitution Method. We want to show $T(n) \leq c \log n - b$ for constants $c = 4$ and $b = 1$. Note that $T(2) = 3 \leq 4log(2) - 1$ and $T(3) = 3 \leq 4log(3) - 1$, and thus we have $T(n) \leq c \log n - b$ for $n = 2, 3$. Suppose $T(m) \leq c \log m - b$ for any integer $m < n$.

$$\begin{aligned} T(n) &= 2T(\lfloor \sqrt{n} \rfloor) + 1 \\ (I.H) &\leq 2c \log \lfloor \sqrt{n} \rfloor - 2b + 1 \\ &\leq 2c \log \sqrt{n} - 2b + 1 \\ &= 2c(\frac{1}{2} \log n) - 2b + 1 = c \log n - 2b + 1 \leq c \log n - b \end{aligned}$$

Problem 3 Function $T(n)$ is defined by the following recurrence relation.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor \frac{n}{2} \rfloor) + n \log n & \text{if } n > 1 \end{cases}$$

Prove $T(n) = O(n(\log n)^2)$ by the substitution method.

Solution: We want to prove $T(n) \leq cn(\log n)^2$. This holds for $n = 2, 3$ if $c \geq 3$. Suppose $T(m) \leq cm(\log m)^2$ for $m < n$ and for some constant $c \geq 3$. (c can be determined later).

$$\begin{aligned} T(n) &= 2T(\lfloor \frac{n}{2} \rfloor) + n \log n \\ (I.H) &\leq 2c(\lfloor \frac{n}{2} \rfloor)(\log(\lfloor \frac{n}{2} \rfloor))^2 + n \log n \\ &\leq 2c(\frac{n}{2})(\log(\frac{n}{2}))^2 + n \log n \\ &\leq 2c(\frac{n}{2})((\log n)^2 + (\log 2)^2 - 2 \log n \log 2) + n \log n \\ &= cn((\log n)^2 + 1 - 2 \log n) + n \log n \\ &= cn(\log n)^2 + n(c - 2c \log n + \log n) \\ &\leq cn(\log n)^2 \end{aligned}$$

The last inequality comes from the fact that $c - 2c \log n + \log n \leq 0$ for $c \geq 3$ and $n \geq 2$.

Problem 4 Solve the following recurrence relations using the Master Theorem, or just state that the Master Theorem does not apply.

(a) $T(n) = 8T(\frac{n}{2}) + 10n^3 \log^2(n)$

Note that the growth of $10n^3 \log^2(n)$ is not the same as the growth of any polynomial function. So, Master Theorem does not apply here.

*H*int: There is a general version of the Master Theorem which is beyond the scope of this class. By the general version of the Master Theorem if $f(n) = \Theta(n^c \log^k(n))$ for $c = \log_b a$ then $T(n) = \Theta(n^c \log^{k+1}(n))$. Here, $\log_2 8 = 3$ and $f(n) = \Theta(n^3 \log^2(n)$. Therefore, $T(n) = \Theta(n^3 \log^3(n))$.
Both solutions are acceptable!

(b) $T(n) = 5T(\frac{n}{3}) + \Theta(n^{2.3})$

Here $2.3 > \log_3(5)$. Therefore, $T(n) = \Theta(f(n)) = \Theta(n^{2.3})$.

(c) $T(n) = 4T(\sqrt{n}) + O(n^2)$

Master Theorem does not apply here.

(d) $T(n) = 9T(\frac{n}{3}) + 100n^2$
Here $c = \log_b a = \log_3 9 = 2$ and $f(n) = \Theta(n^c)$ so $T(n) = \Theta(n^c log(n)) = \Theta(n^2 \log n)$.

Problem 5 Implement a queue by using two stacks.

Solution: Use two stacks S_1 and S_2. Whenever you need to push an element in the queue, push it in S_1. The first time that you need to pop out of queue you need to access the last element of stack. Pop elements of S_1 one by one and push them into S_2. Now the bottom element in S_1 is the top element in S_2. So pop out one element of S_2. From now on whenever you have to dequeue, you only need to pop one element out of S_2 and whenever you have to push into queue, you push the element in S_1. If at some point you have to dequeue and S_2 is empty, again pop elements of S_1 and push them into S_2 one by one.

Also note that for each element, it is pushed once to S_1 and once out of S_1 and also once into S_2 and once out of S_2 (once an element is popped out of a stack it never returns to the stack). So the running time of inserting and removing n elements would be $O(n)$.

Problem 6 Suppose you have a linked list of size n. Give an $O(n)$ algorithm to find out whether the linked list has a loop. Note that you cannot store the elements of the list in another array or anything like that. In other words, you can use extra $O(1)$ space.

Note: We do not know the size of the linked list. The input is the pointer to the first element of the linked list.

Hint: Define two pointers and move one with speed 1 and another with speed 2!

Solution : First of all, if a pointer is moving though a link list with a loop, once it falls inside the loop it always moves through the elements of the loop and never comes out. Now assume both pointers are in the loop. At each point the distance between the two pointers is decremented by one. Therefore, in at most m steps the two pointers would be on the same node where m is the size of the loop.

If no loop exists, the second pointer would hit the end of the linked list sooner than the first pointer.

Problem 7 Design an algorithm to find the lowest common ancestor of two given nodes in a balanced binary search tree. Your algorithm should runs in $O(\log n)$ where n is the number of nodes.

Solution: Let N_1 and N_2 denote the values of the two given nodes. Start from the root of the tree. Whenever you are at a node, compare N_1 and N_2 against the value of this node and decide for each of them whether you should move to the right child or the left child. If for both of them you should move to the same child then move to that node. For example, if both N_1 and N_2 are bigger than the value of the current node, move to the right child. However, if at some node, N_1 is bigger than the value of the node and N_2 is smaller (or the other way), the current node must be the lowest common ancestor. At each step if N_1 (or N_2) is equal to the value of the current node, the current is the lowest common ancestor. The order of this algorithm is height of the binary search tree which is $O(log(n))$.

Problem 8 Design an algorithm to find the **next** (e.g. in-order successor) node of a given node in a binary search tree.

Solution: You need to output the element in tree that is bigger than the value of the given node. The first observation is that the answer is either in the nodes right subtree or is one of the ancestors. You can check that no other node can be an answer to this problem. Do the following

1) If right subtree of node is not NULL, then successor lies in right subtree. Go to the right subtree and return the node with minimum key value in right subtree. To find the node with min key value, all you need to do is to move to the left child while possible.

2) If right subtree of node is NULL, then successor is one of the ancestors. In this case travel up using the parent pointer until you see a node which is left child of its parent. The parent of such a node is the successor.

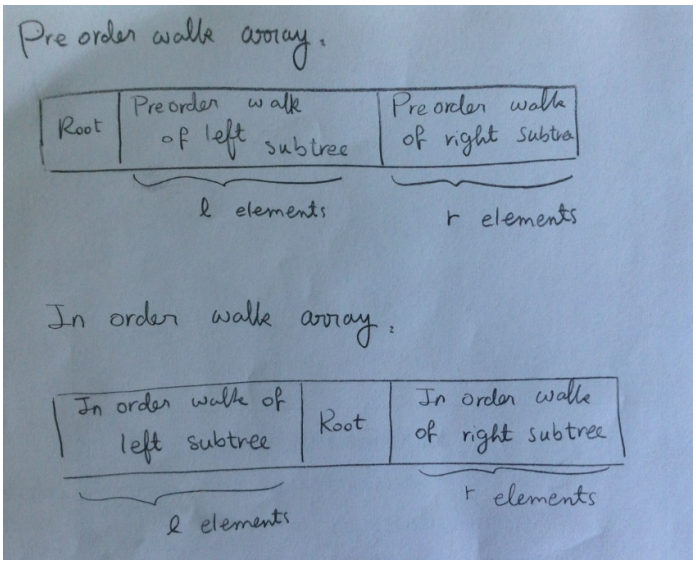

Figure 1: In-order and Pre-order arrays

Problem 9 Design an algorithm to construct a binary tree from given in-order and pre-order tree walks. (Note: This is not necessarily a binary search tree!)

Solution: The inorder and preorder tree walks can be given in an array as shown in figure 1.

You can find the root simply from the preorder array(First element). Root is $l + 1$-th element of the in-order array as well. Once you find root in the in-order array, you know its location (or the number of elements in the left subtree denoted by l). Also you can find the pr-order and in-order representations of both left and right subtrees. This means you can find the left and right subtree recursively. Also you have the root itself. All you need to do now is to set the left subtree's root as the left child of current root and the right subtree's root as the right child. The following function builds the tree for given arrays of preorder and inorder recursively. The output is the root of the tree. The location of root in the in-order array is denoted by $k = l + 1$.

```
function BuildTree(preorder, inorder, n)
    if n ≤ 0 then
        return null
    else
        k ← location of root in inorder array
        root.key ← preorder(1)
        root.left ← BuildTree(preorder[2:k],inorder[1,k-1], k-1)
        root.right ← BuildTree(preorder[k+1:n],inorder[k+1:n], n-k)
        return root
    end if
end function
```