COMP 3711H – Honors Design and Analysis of Algorithms
2014 Fall Semester – Written Assignment # 1
Distributed: Sept, 25 2014 – Due: October 7, 2014
Revised October 6, 2014

Your solutions should contain (i) your name, (ii) your student ID #, and (iii) your email address
Some Notes:

- Please write clearly and briefly.
- Please follow the guidelines on doing your own work and avoiding plagiarism given on the class home page. Don't forget to *acknowledge* individuals who assisted you, or sources where you found solutions.
- Please make a *copy* of your assignment before submitting it. If we can't find your answers, we will ask you to resubmit the copy.
- In what follows the phrase *Give an* $O(f(n))$ *algorithm* means to describe the algorithm, prove its correctness and correctly analyze its running time to show that it's $O(f(n))$.
- Each problem is worth 25 points.
- You will be asked to submit a PDF. This can be generated by Latex, from Word or a scan of a (legible) handwritten solution. The solution submission method will be announced on September 30.
- Revision of October 6, 2014.
  "$\log n$" in question 1(b) was changed into a "h".

Problem 1: Locally Minimal Elements

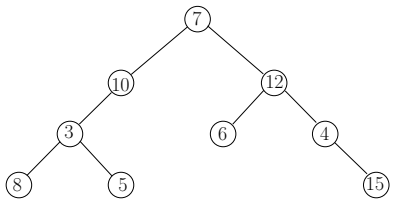
(a) The input to this problem is an n element array $A[1, \ldots, n]$. Element $A[i]$ is *locally minimal* if it is not larger than all of its neighbors (either the neighbors to both sides, or the neighbor to one side if it's the first or last item). For example, 7, 6 and both 8's are the locally minimal items in the array below

7	13	15	12	6	9	11	14	16	23	25	24	8	8

Finding the absolute globally minimal item requires $O(n)$ time to inspect every item. Finding a locally minimal one can be done faster.

Give an $O(\log n)$ algorithm for finding a locally minimal element in the array. If there are many such items, you only need to return one of them.

(b) A node is locally minimal in a binary tree if it is not larger than all of its (1,2 or 3 depending upon the case) neighbors. For example, in the tree below, 7, 3, 6 and 4 are the locally minimal elements



The input here is a pointer to the root of an n-node binary tree. The tree is described using a key value and left and right (possibly empty) pointers for each node. No parent pointers are provided.

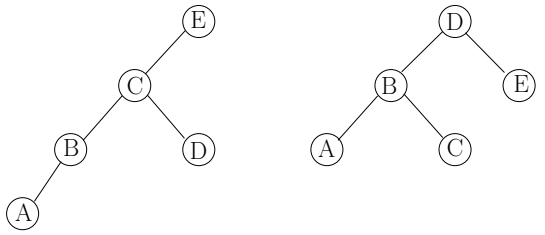
Give an $O(h)$ algorithm for finding a locally minimal element in the tree, where h is the height of the tree. If there are many such items, you only need to return one of them.

Problem 2: Average Construction of Binary Search Trees

Let v be a node in tree T.
The *depth*, $d(v)$, of v in T is the length of the path from the root of T to v.
Note that the depth of the root is 0.
The *Path Length of* T, $PL(T)$, is the sum of the depths of all of the nodes of T; $PL(T) = \sum_{v \in T} d(v)$.

Note that $\frac{1}{n} PL(T)$ is the average depth of a node in the tree.

There are many different possible binary search trees that can store a given set of keys. But, if the tree is built by inserting the nodes one at a time using the algorithm given in class, the insertion order fixes the structure of the tree. The figure gives two construction orders and the path lengths of the resulting Binary Search Trees.



Created by ECBDA Path length is $1+2+2+3=8$

Created by DBACE Path length is $1+1+2+2=6$

Note that the cost of inserting a node into the tree is the depth at which the node is inserted. The total cost of building a tree is the sum of the costs of inserting each of the nodes.

Suppose the input is a random permutation of the numbers $1, 2, \ldots n$. That is, each of the $n!$ possible input orders is equally likely.

(a) Prove that if T is built from such a random ordering, then the average value of $PL(T)$ is $O(n \log n)$.
This average is taken over all possible input orderings.

Note that this implies that for a randomly built tree, the average depth of a node is $O(\log n)$

(b) Prove that the average cost of building the tree is $O(n \log n)$.

Hint: Try using the average case analysis of quicksort as one piece of the solution of this problem.

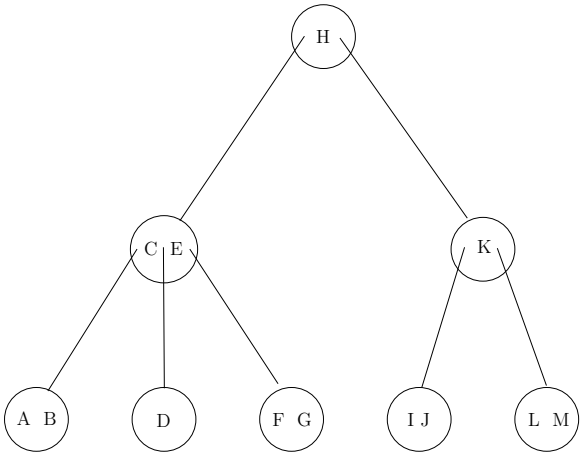
Problem 3: A Different Type of Balanced Tree.
Consider the following type of search tree.

- All leaves are at the same depth
- All nodes have pointers to their parents (parent of the root is itself)
- All nodes contain 1 or 2 keys.
  If a node contains one key it has two children;
  If a node contains two keys, it has three children.

  In what follows p's are pointers to subtrees, K's are key values. The phrase "key in p" is shorthand for "key in the subtree pointed to by p.

  - If an internal node has two children it is of the form p_0, K_1, p_1 where all keys in p_0 are $< K_1$ and all keys in p_1 are $\geq K_1$.
  - If an internal node had three children it is of the form p_0, K_1, p_1, K_2, p_2 where all keys in p_0 are $< K_1$, all keys in p_1 are $\geq K_1$ and $< K_2$ and all keys in p_2 are $\geq K_2$.

The diagram provides an example of such a tree.



(a) Prove that the height of the tree is $O(\log n)$

(b) Give an $O(\log n)$ algorithm for finding an item in the tree

(c) Give an $O(\log n)$ algorithm for inserting a new item into the tree

Problem 4: Heaps

In class we learned how to implicitly maintain a binary min-heap in an array and perform *Insert* and *Extract-Min* in $O(\log n)$ time.

(a) Using the same data structure, give an $O(\log n)$ algorithm for *Delete*. The input to the algorithm is a location in the array and the output is a min-heap in the array with that item removed.

(b) In class we saw how to merge two sorted lists using $n-1$ comparisons where n is the total number of items in the two lists. Extending that same idea to merging k sorted lists would require $(k-1)(n-1)$ comparisons in the worst case. Using the data structure for part (a) show how to merge k sorted lists in $O(n \log k)$ time.