Quiz 2 CS213M 2014

15 Marks, Closed Notes, 1 Hour. I have made every effort to ensure that all required assumptions have been stated. If absolutely necessary, do make more assumptions and state them very clearly. Whenever possible, describe your algorithm using pseudo code.

Present the choice of

1. data structure
2. along with the algorithm (in step-wise, systemmatic pseudocode) and
3. its running time

that are the most time-efficient (space is not a concern), for each of the following tasks:

1. You have n distinct numeric keys in an array and you need to report all the keys that are larger than or equal to a given query key x (which is not necesssarily amongst the n keys you have). Note that the keys do not need to be reported in sorted order.

   (4 Marks)

   SOLUTION: The same answer holds for *smaller* and *larger*; you could use the negative values of numbers to change from one to the other.

   You cannot do better than to scan the array once, comparing each element against x. This will be $O(n)$! No data structure could improve on this. However, if x were to keep changing, you might want to consider using a heap to be more efficient!

   Using a heap T, your algorithm can run in $O(n + k)$ time, where k is the number of keys reported and n is the time required construct the heap-bottom up.

   Note that if v is a node of T that is larger then x, then all the keys stored in its subtrees are larger then x. Therefore it is enough for us to scan from the root the nodes of T that are smaller then x. There are two basic ways to do this: a recursive one and an iterative one that uses an external data structure (either queue or stack).

Recursive algorithm has to be run as RecurFindSmaller(T, $T.root()$,x) to find all keys in T that are smaller then or equal to x.

RecurFindSmaller(T, v, x)
Input: a heap T; a node of T, v; a query key x
Output: keys of in the subtree of T rooted in v that are smaller or equal to x
if (T.isInternal(v)) **then**
if (v.key() $\leq x$) **then**
report key v.key()
RecurFindSmaller(T, T.leftChild(v), x)
RecurFindSmaller(T, T.rightChild(v), x)

Iterative algorithm using a queue (or alternatively a stack if we replace enqueue and dequeue methods by push and pop methods correspondingly.

IterFindSmaller(T, x)
Input: a heap T; a query key x
Output: keys of in the subtree of T rooted in v that are smaller or equal to x
Q is an empty queue
Q.enqueue(T.root())
while (! Q.isEmpty()) **do**
current $\leftarrow$ Q.dequeue()
if (T.isInternal(*current*) and *current*.key() $\leq x$) **then**
report key *current*.key()
Q.enqueue(T.leftChild(*current*))
Q.enqueue(T.rightChild(*current*))

This algorithm runs in $O(k)$ for the following reason. As we traverse down the tree, each node with a key smaller or equal to x gets scanned exactly once. The only other nodes that get scanned are their children. Given k returned keys, their k corresponding nodes in the heap have $2k$ children. Despite the fact that some of their children are the nodes whose key is $\leq x$ also, this still gives us the desired upper bound: we end up scanning $3k$ nodes (k nodes corresponding to the reported keys, and $2k$ of their children), which is $O(k)$.

2. You have n distinct numeric keys in an array and you need to report the k^{th} largest element.

   (4 Marks)

   SOLUTION: The same answer holds for *smaller* and *larger*; you could use the negative values of numbers to change from one to the other.

   Construct a heap, which takes $O(n)$ time. Then call method *removeMin* k times, which takes $O(k \log n)$ time.

3. You have n distinct numeric keys in an array and for some other purpose, the keys are stored in an AVL tree, in a Heap as well as in a HashMap

(you can assume any hashing function). Which data structure will you choose to implement the method *countAllInRange(k_1,k_2)* that computes and returns the number of entries with key k such that $k_1 \leq k \leq k_2$. Since the three data structures are already provided, you can ignore the time complexity of populating that data structure.

(7 Marks)

SOLUTION: Use an AVL tree. Complexity will be $O(logn)$. For each node of the tree, maintain the size of the corresponding subtree, defined as the number of internal nodes in that subtree. While performing the search operation in both the insertion and deletion, the subtree sizes can be either incremented or decremented. During the rebalancing, care must be taken to update the subtree sizes of the three nodes involved (labeled a, b, and c by the restructure algorithm).

To calculate the number of nodes in a range (k_1, k_2), search for both k_1 and k_2, and let P_1 and P_2 be the associated search paths. Call v the last node common to the two paths. Traverse path P_1 from v to k_1. For each internal node $w \neq v$ encountered, if the right child of w is in not in P_1, add one plus the size of the subtree of the child to the current sum. Similarly, traverse path P_2 from v to k_2. For each internal node $w \neq v$ encountered, if the left child of w is in not in P_2, add one plus the size of the subtree of the left to the current sum. Finally, add one to the current sum (for the key stored at node v).