Worth: 11% **Due:** Friday February 12

For each question, please write up detailed answers carefully. Make sure that you use notation and terminology correctly, and that you explain and justify what you are doing. Marks may be deducted for incorrect/ambiguous use of notation and terminology, and for making incorrect, unjustified or vague claims in your solutions.

1. Let D be an ordered Dictionary with n items implemented with an AVL tree. Show how to implement the following method for D in time $O(\log n)$:

countAllinRange(k_1, k_2): return the number of items in D with key k such that $k_1 \leqslant k \leqslant k_2$.

note: you may assume that the AVL tree supports a data field at each node x which stores the number of nodes in the subtree rooted at x.

Solution : We will show how to implement a method AllinRangefromNode(v, k_1, k_2) where v is a node in D. This is enough of course since AllinRange(k_1, k_2) = AllinRangefromNode($root, k_1, k_2$). In addition we will implement "countAllBiggerThan(v,k)" and "countAllsmallerThan(v,k)" that return the number of items in the subtree of D rooted at node v, with keys bigger-than or smaller-than k respectively.

We denote by "numnods(v)" the data field supported by the AVL that stores the number of nodes in the subtree rooted at v.

AllinRangefromNode(v, k_1, k_2) {
if $v = NIL$ return 0
if ($v.key < k_1$) return AllinRangefromNode($rightchild(v), k_1, k_2$)
if ($v.key > k_1$) return AllinRangefromNode($leftchild(v), k_1, k_2$)
else return 1+ countAllBiggerThan($leftchild(v), k_1$) + countAllSmallerThan($rightchild(v), k_2$)
}

Why does this work? If $v.key < k_1$ then we can safely ignore the elements in the left subtree of v and continue with the same query to the right child of v. A similar argument justifies the third line. If, however,$k_1 \leqslant v.key \leqslant k_2$ then v should be counted, and for the two subtrees of v we only need to count the elements that are bigger than k_1 in the left subtree, and smaller than k_2 in the right subtree.

How long does it take? We can bound the running time of this method by the height of the tree, that is $O(logn)$ plus the running time of the other two methods. This is since every time the conditions in line 2 or line 3 are satisfied in the recursion tree, we move to a deeper node. Hence there can be as many recursive calls to AllinRangefromNode as the height of the tree.

Now we define 'countAllBiggerThan(v, k) and 'countAllSmallerThan(v, k).

countAllBiggerThan(v, k) {
if $v = NIL$ return 0
if ($v.key < k$) return countAllBiggerThan($rightchild(v), k$)
else return 1 + countAllBiggerThan($leftchild(v), k$) + numnodes($rightchild(v)$)
}

countAllSamllerThan(v, k) {
if $v = NIL$ return 0
if ($v.key > k$) return countAllSmallerThan($lefttchild(v), k$)
else return countAllBiggerThan($rightchild(v), k$) + numnodes($leftchild(v)$)
}

Why do they work? We analyze the first of the two methods (the other is proved similarly). If the condition in line 2 holds then we can avoid looking at the left subtree as all elelemtns there are clearly too small. Otherwise, *all* elements of the right subtree must be counted, the root must be counted and in addition all sufficiently large elements in the right subtree should, hence the recursion.

How long do they take? As before its clear that the recursion can be no longer than the height of the tree, as regardless of whether the condition in line 2 holds, we move to a child node of the original node.

2. Given two AVL trees T_1 and T_2, where the largest key in T_1 is less than the smallest key in T_2, Join(T_1, T_2) returns an AVL tree containing the union of the elements in T_1 and T_2.
Give an algorithm (in pseudocode) for Join that runs in time $O(\log n)$, where n is the size of the resulting AVL tree. Justify the correctness and efficiency of your algorithm.

Solution:

Begin by computing the heights h_1 of T_1 and h_2 of T_2. This takes time $O(h_1 + h_2)$: You simply traverse a path from the root, going to left child if the balance factor is -1, to the right child if it is positive, and to any of the children if the balance factor is 0, until you reach a leaf. Assume that $h_1 \geqslant h_2$; the other case is symmetric.

Next, DELETE the smallest element x from T_2, leaving T_2' of height h. This takes $O(h_2)$ time.

Find a node v on the rightmost path from the root of T_1, whose height is either h or $h + 1$, as follows:

$v \leftarrow root(T_1)$
$h' \leftarrow h_1$
while $h' > h + 1$ do
    if balance factor (v) = -1
    then $h' \leftarrow h' - 2$
    else $h' \leftarrow h' - 1$
    $v \leftarrow rightchild(v)$

This takes $O(h_1)$ time.

Let u denote the parent of v.
Form a new tree whose root contains the key x, whose left subtree is the subtree rooted at v and whose right subtree is T_2'.
Note that this is a valid binary search tree, since all the keys in the subtree rooted at v are in T_1 and, hence, smaller than x, and, by construction, x is smaller than or equal to all elements in T_2'. The balance factor of the root of this new tree is $h - h'$, which is either -1 or 0, so this new tree is a valid AVL tree. The height of this new tree is $h' + 1$, which is 1 bigger than v's height. Let the root of this new tree be the right child of u, in place of v. Again, since all keys in this new tree are at least as big as u, this results in a valid binary search tree. This part of the construction takes constant time.

Now, as in the INSERT algorithm, we go up the tree, starting at u, fixing balance factors and perhaps doing a rotation. This takes $O(h_1)$ time. Note that the correctness follows from the condition at u before this process is begun is a condition that can arise during the INSERT algorithm.

Since $h_1, h_2 \in O(\log n)$, the total time taken by this algorithm is in $O(\log n)$.

3. Suppose we are using a priority queue Q to schedule jobs on a CPU. Job requests together with their priorities are inserted into the queue. Whenever the CPU is free, the next job to execute is found by extracting the highest priority job in Q. Using a heap implementation for Q the scheduler can both extract the next job from Q and add a new job to Q in time O(log n) where n is the number of pending jobs in the queue.

However, in practice, we want our scheduler to be able to do more than just insert jobs and extract the highest priority job. In particular, we want our scheduler to be able to remove a job x from the job queue (Delete(x)) as well as change the priority of a job x to some new value k (Change-Priority(x, k)). Show how to add the operations Delete(x) and Change-Priority(x, k) to the heap data structure so that (a) they both run in time O(log n) and (b) the resulting data structure after executing either these operations is still a heap. For both operations, you may assume that parameter x gives you the index corresponding to job x in the array representation of the heap.

Solution:

- For the *delete* operation, we start with taking the last item (let's say it has key k) and placing it in position x. So now the original key of x is gone, and instead we have y there, and in addition the "heap" is of the right size. But of course, it's not necessarily a heap: y may agree or disagree with its parents/children. If y is smaller than the key of the parent, we perform up-heap bubble. Otherwise, we perform down-heap bubble (which, in the case that y is smaller than the keys of the children, will stop right away). Since there is either up-heap or down-heap bubble, this will have ruinning timeof $O(\log n)$. The initial swap is of course $O(1)$.
- The change-priority is similar but even easier: All you have to do is change the key of poistion x to k, and continue as the second phase of the delete(x) method above. In fact, you can even implement delete(x) by performing the swap, and then using change-priority as a black box if you wish! Again, clearly this takes $O(\log n)$.

4. [10 marks]
Consider two Binomial Heap H_1 and H_2. You are required to create the union H of H_1 and H_2 and also to find the minimum key in H. A standard way would be to perform

$H \leftarrow \text{Union}(\text{H}_1, \text{H}_2)$
$m \leftarrow \text{LookupMin}(\text{H})$.

Alternatively, you may perform

$m_1 \leftarrow \text{LookupMin}(\text{H}_1)$
$m_2 \leftarrow \text{LookupMin}(\text{H}_2)$
$m \leftarrow \min(m_1, m_2)$
$H \leftarrow \text{Union}(\text{H}_1, \text{H}_2)$.

Give an example of heaps H_1 and H_2 in which the first sequence of operations is faster, and one in which the second is faster. Justify your answer.

Solution: Notice that the union operation will have the same inputs in both cases (the lookupMin operations do not change the Binomial heaps). So the question boilds down to comparing the cost of the lookupMin operations. For a Binomail heap J, denote by $\ell(J)$ length of the linked list of Binomial trees in H. Recall that the cost of lookupMin(J) is simply $\ell(J)$. Hence, if we denote by u the running time of persorming the union of H_1 and H_2 we get that the first method would run in time

$$u + \ell(H)$$

and the second in time

$$\ell(H_1) + \ell(H_2) + 1 + u.$$

An example in which the first method is faster is one where $\ell(H)$ is bigger than $\ell(H_1) + \ell(H_2) + 1$ and vice versa. If H_1 and H_2 contain list of binomial trees of all heights upto some k, then H will contains only *one* binomial tree (of height $k+1$, and so $\ell(H) = 1$ and $\ell(H_1) + \ell(H_2) + 1 = 2k + 1$. On the other hand, it is always the case that $\ell(H) \leqslant \ell(H_1) + \ell(H_2)$ (do you see why? equality will happen only when the binomial trees of H_1 are all different than those of H_2) and so an example of the opposite kind does not exist.