Rishad Rahman (Deepak's Recitation)	March 20, 2015
6.046 Problem 5-1	
Collaborators: *None*	

(a) **Algorithm**: Assume $x.key < k$, otherwise we can use symmetry to modify the algorithm to handle $x.key > k$. We do a upper right traversal from x which is similar to the idea of a reverse search. We walk up if an upper node exists, otherwise we go right. We do this until we reach a node n such that $n.right.key > k$ (or if we ever find $n.key = k$ in which case we just go down to level 0 to return y). Then finish off with the standard bottom-right traversal skip list search starting from n to find k.

Correctness: $n.right.key = k$ is trivial. If $n.right.key < k$ then k must be located to the right of $n.right.key$ hence it is reachable from $n.right$. Else $n.right.key > k$ in which case k is between $n.key$ and $n.right.key$ so it is reachable from n. Upward traversals don't effect the reachability of k from n as long as $n.key < k$.

Runtime: Note we will only be looking at nodes in between x and y. Note that the algorithm is connected by the "Least Common Ancestor" of x and y: the lowest n, which we will denote by $l_{x,y}$, such that $x < n.key < k < n.right.key$ whose height is given by $h(x,y)$. We have $\mathbb{P}[h(x,y) \leq c \lg m] \geq \frac{1}{m^{c-1}}$ using the same analysis as in lecture since there are m nodes between x and y inclusive and the probability that an element gets pushed up more than $c \log m$ times is $\frac{1}{m^c}$ so the Union Bound gives the claim. Now we know the height of our search is $O(\lg m)$ with high probability, now we have to argue why the total number of traversals is $O(\lg m)$ with high probability. This also follows by the same analysis in lecture with the Chernoff bound, on either side of $l_{x,y}$. On traversal from $x \to l_{x,y}$ we have that an up and right traversal occurs with probability $\frac{1}{2}$ so we get that the number of up traversals is at least $c \log m$ with high probability if there are are $8c \log m$ total traversals so it follows that the number of traversals on the left is $O(\lg m)$ with high probability, the number of traversals on the right by symmetry is $O(\lg m)$ with high probability, and so by standard techniques we get that the total number of traversals is $O(\lg m)$ with high probability.

(b) Augment each node n with $n.aug = \text{rank}(n.right) - \text{rank}(n)$. Search obviously doesn't change since it has nothing to do with the augmentation.

Insert: In the initial search keep a counter *count* $= 0$ and initialize a stack S. During the search if we traverse right from a node n we increment *count* by $n.aug$ and if we traverse down we push the current *count* onto the stack. In the upwards propagation step, we do the following for every node n that contains the key we inserted starting at the bottom level. At level 0 we let the augment be 1. Otherwise we go up and let $s = S.pop()$ then $z \leftarrow n.left.aug$, $n.left.aug \leftarrow count - s$, and $n.aug \leftarrow z - n.left.aug$.

We continue this as we do the coin flips and traverse upwards. An exception is when $|S| = 0$ in which case we're increasing the height of the skip list and push up the two ends as well so we use set $n.left.aug = \textit{count}$, $n.aug = N - \textit{count} - 1$, and $n.right.aug = 0$ where N is the number of elements inserted. We can briefly argue correctness by verifying that the augmentation stays correct as we defined it earlier.Note that *count* is the sum of augmentations from consecutive nodes so it sums to $\text{rank}(x)-1$ where x is the node we inserted at level 0. Similarly the count when we went traversed down from a node m during the initial search was $\text{rank}(m) - 1$ which we pushed into S. When we propagate up during the insert to node n I claim $n.left.aug = S.pop()$. This is true since $n.left.key < n.key < n.right.key$ so we must have went downwards from $n.left$ in the initial search and by the nature of a stack we will only pop the value once we come back up to the same level later. We now perform our computation steps from the algorithm:

$z = n.left.aug = \text{rank}(n.right) - \text{rank}(n.left)$ (previous augmentation)

$n.left.aug = \textit{count} - s = (\text{rank}(x) - 1) - (\text{rank}(n.left) - 1) = \text{rank}(x) - \text{rank}(n.left)$

$n.aug = z - n.left.aug = \text{rank}(n.right) - \text{rank}(n.left) - (\text{rank}(x) - \text{rank}(n.left)) = \text{rank}(n.right) - \text{rank}(x)$.

So everything checks in the interior of the list and when the height of the list grows we also check $n.left.aug = \textit{count} = rank(n)-1$ and $n.aug = N-\textit{count}-1 = N-rank(n)$. For runtime notice that there is constant work per traversal to increment *count*, push on to S, and update augmentations. Since number of traversals and stack size is $O(\log N)$ with high probability we have that the runtime doesn't change.

Delete: For every node n to be deleted we do $n.left.aug \leftarrow n.left.aug + n.aug - 1$. This is correct since $n.left.aug+n.aug-1 = \text{rank}(n.right)-\text{rank}(n.left)-1$ where the -1 justifies the fact there is now 1 less element between $n.left$ and $n.right$. This just adds constant time per node deleted hence we still have $O(\log N)$ with high probability.

(c) **Algorithm**: This will be very similar to (a). Initialize $c = r$. Traverse the same way as in (a) except that the check is if $c - n.aug \geq 0$ in order to traverse right (unless you can traverse upwards) in which case $c \leftarrow c - n.aug$. Once you find the "Least Common Ancestor" you do the same thing going downwards except you traverse down when $c - n.aug < 0$, otherwise you go to the right and $c \leftarrow c - n.aug$. Once $c = 0$ we have found the key so we just go down to level 0 to return y.

Correctness: Note that at every right step from a noden c gets decreased by $n.aug$ but the sum of these consecutive augmentations when we're at node n after starting from x is $\text{rank}(n) - \text{rank}(x)$ so $c = r - (\text{rank}(n) - \text{rank}(x))$ at node n. As a result if $c - n.aug \geq 0$ then $r \geq \text{rank}(n.right) - \text{rank}(x)$ so it is safe go right otherwise we are forced to go down as described in the algorithm. Once $c = 0$ that means $r = \text{rank}(n) - \text{rank}(x)$ as desired.

Runtime: There are a total of $m = r + 1$ elements inclusive at level 0 between x and y where $\text{rank}(y) = \text{rank}(x) + r$ so the analysis is exactly the same as in (a) which shows the least common ancestor is of height $O(\log m)$ with high probability hence our total runtime is $O(\log m)$ with high probability.

Rishad Rahman (Deepak's Recitation) March 21, 2015

6.046 Problem 5-2

Collaborators: *None*

(a) Just do a rank-select to find the $n-m$th smallest element then run through the sequence in order to find which ones are larger and return those. This is correct by definition of rank-select so we will definitely output the m largest elements which obviously maximizes $\sum_j s_j$. The space and time complexity are both $O(n)$ because that is the complexity of rank-select and we never have to use anything more than the length of the input.

(b) We will iterate through the list keeping track of S_A, S_B, $S_{A'}$, $S_{B'}$, where we are going to be iterating through the list where — indicates our current location, S_A is our current optimal set of A's (on the left of —), $S_{A'}$ is the set of elements to the left of — not including S_A, similarly defined for S_B and $S_{B'}$ except replace left with right.