COMP 3711H – Fall 2016
Tutorial 3 - Solution Sketch

Note. The answers given below might not be complete. Some are sketches or hints, sometimes missing details. Also answers to somne of thee simplest questions might not be given.

1. **Heapify**
   In class, we learned how to maintain a min-heap implicitly in an array.
   Given that $A[i \dots (j-1)]$ represents an implicit min-heap, we saw how, in $O(\log j)$ time to add $A[j]$ to the min-heap. This led to an $O(n \log n)$ algorithm for constructing a min-heap from array $A[1, \dots, n]$.

   For this problem show how to construct a min-heap from an array $A[1 \dots n]$ in $O(n)$ time.

   It might help to visualize the min-heap as a binary tree and not an array.

   For simplification, you may assume that $n = 2^k + 1$ for some n, i.e., the tree is complete.

   Hint: Consider heapifying the nodes in the order from bottom to top.

   "Heapify" the nodes row by row, moving from $h = 1$ to $h = k$ (h being the height of the node, with leaves having $h = 0$ and root having $h = k$). By "heapify' a node" we mean make the tree rooted at the node have the min-heap property. Note that at the time we process a node, its two subtrees have already been heapified. Thus, we can heapify by bubbling the node down as many levels as appropriate (as shown in class when performing Extract-Min).

   The cost of heapifying a node at height h is $O(h)$ and there are 2^{k-h} nodes at height h. Thus, the total cost of heapifying all nodes is

$$O\left(\sum_{h=1}^{k} h 2^{k-h}\right) = O\left(2^k \sum_{h=1}^{k} d 2^{-d}\right) = O(2^k) = O(n).$$

2. **Randomized Binary Search Trees**

   - Consider a Binary search tree T on n keys.
     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. This is also the average time to search for a randomly chosen node in the tree.
   - Suppose that every key K_i in a set of n keys has real weight w_i associated with it, with the weights being unique.
   - There is a unique binary search tree that can be built on the n keys that also satisfies min-heap order by the weights (Why?).
   - Suppose n weights $w_1, w_2, \dots, w_n$ are chosen independently at random from the unit interval $[0, 1]$ and then sorted. The resulting order is a random permutation of the n items.

A *Treap* or *Randomized Binary Search Tree* on n keys K_i is constructed by choosing n weights w_i independently at random from the unit interval $[0, 1]$ and associating w_i with K_i. The Treap is the unique BST built on the n keys that also satisfies min-heap order on the weights.

(a) If T is the Treap built, prove that the average value of $PL(T)$ is $O(n \log n)$
Hint: consider quicksort

Choosing the random weights fixes a random permutation. Note the following fact.

Let $\pi = \pi_1, \pi_2, \ldots, \pi_n$ be a random permutation on n items. Let π^1 be the items with value less than π_1 writen in the same order as in π and let π^2 be the items with value greater than π_1 writen in the same order as in π. Then π^1 is a random permutation on its items (i.e., each one of its $|\pi^1|!$ orderings is equally likely to occur) and π^2 is a random permutation on its items.

The process described essentially builds the tree from a permutation π as follows:

- *Choose the first item in the permutation as the root of the tree.*
- *If π^1 is not empty, recursively build the tree off of π^1 and make it the left subtree of π_1.*
- *If π^2 is not empty, recursively build the tree off of π^2 and make it the right subtree of π_1.*

Let C_n be the average value of $PLT(T)$ where T is built from a random π on n items. From the construction described above, if π_1 is the kth key, then the left subtree is a random Treap on $k-1$ items so it has average path length C_{k-1} and the right subtree is a random Treap on the $n-k$ items to the right so it has average path length C_{n-k}. Since every item in the final tree is one level deeper than it is in the left or right subtree we find that the average path length, CONDITIONED ON THE ROOT being the kth key is

$$n - 1 + C_{k-1} + C_{n-k}.$$

Since every one of the n keys is equally likely to be the root

$$C_n = \frac{1}{n}\sum_{i=1}^{n}(n - 1 + C_{k-1} + C_{n-k}) = n - 1 + \sum_{i=1}^{n}(C_{k-1} + C_{n-k})$$

with initial condition $c_0 = 0$. This is EXACTLY the quicksort recurrence, so $c_n = O(n \log n)$.

(b) Describe how to build T in time $O(n \log n + PL(T))$

After picking the random weights sort the keys by their weights, from largest to smallest. Then insert the keys into the tree in that largest to smallest weight order. Notice that this exactly builds the associated Treap (this can be proven by induction) and the cost of inserting a node is exactly its depth. Thus, the cost of building the tree is $O(n \log n)$ for the sort plus $PLT(T)$, which is what we wanted to prove.

3. **AVL Trees**

(a) Construct an AVL tree by inserting the items 134625 in that order.
Next construct another AVL tree on those items by inserting in the order 123456.
Do they have the same height?

You can construct these trees using the web site pointed to by the class lecture note page).

(b) Let T be a tree on n keys that satsifies the AVL balance condition. Is there always an insertion order for the keys that builds T?
If yes, what is it?

Yes there is. Order the keys by their depth in the tree, higest to lowest, breaking ties arbitrarily. Build a tree by inserting the nodes into the tree in that order. The claim is that at every step, after doing the insertion, the AVL condition is satisfied on the tree built so far so no rebalancings are done. Thus, this is like inserting nodes into a STATIC (not AVL) tree in increasing depth order. Such an insertion order will build the original tree.

The tricky part of this proof is to show that, if the original tree is an AVL tree then, after every insertion, no rebalancings need to be done. We will see this through the following intermediate lemma.

Lemma 1: Let T be an AVL tree with height h. Remove any subset of nodes at depth h. What remains is also an AVL tree.

Proof: The proof is by induction on h. The statement is true by observation for $h = 0, 1, 2$. Now suppose the statement is true for all AVL trees of height $< h$.

Let T be an AVL tree of height h with root r and T' the same tree with a given subset of depth h nodes removed. For u a node in T (and T') set T_u to be the subtree in T rooted at u, (T'_u the subtree in T',) $h(u)$ the height of T_u (and $h'(u)$ the height of T'_u).

Let x be the left child of r and y the right child. By definition, T_x, T_y are AVL trees with heights $< h$.

Removing a subset of nodes of depth h from T_x, T_y removes a set of (none, some or all) nodes from depth $h(x)$ in T_x and from depth $h(y)$ in T_y. Thus, by induction, T'_x and T'_y are AVL trees which means that all of their nodes satisfy the AVL balance condition. So, in order to prove the Lemma it only remains to show that node r satisifies the AVL balance condition in T'.

From the AVL condition, one of the following three statements must hold: (i) $h(x) = h(y) = h - 1$ or (ii) $h(y) = h(x) - 1 = h - 2$ or (ii) $h(x) = h(y) = h - 2$.

- *(i) after removing nodes at depth h, $h'(x)$ and $h'(y)$ are either h or $h - 1$ so $|h'(x) - h'(y)| \leq 1$ and r satsifies the AVL condition.*
- *(ii) In (ii) no nodes at depth h are in T_y so $h'(y) = h(y) = h - 2$. On the other hand, $h(x) = h - 1$. The nodes at depth h in the original tree are at depth $h - 1$ in $h'(x)$. After removing (none, some or all) of those nodes to get T', we are left with $h'(x) = h - 1$ OR $h'(x) = h - 2$. In both of those cases $|h'(x) - h'(y)| \leq 1$ and r satisfies the AVL condition.*
- *(iii) Symmetric to (ii)*

The Lemma is therefore proven.

To apply the lemma return to the ordering of the keys that we defined at the top of this solution. Let x be any key. In the original AVL tree, chop off all of the *levels below where x appears, from bottom to top. Applying the lemma iteratively tells us that what remains is an AVL tree. Now remove x and all items after x in the ordering that are still in the tree (they must be on the same level as x). Again the Lemma tells us that what remains will be an AVL tree. Now insert x into the tree using the standard BST insertion algorithm. Because all of its ancestors are in their proper location in the original AVL tree, x is inserted into the same place as it was in the original AVL tree. The Lemma tells us that this tree is also an AVL tree with x and all of the items before it in the ordering in their proper places.*

Inserting every key into the treee using the defined ordering therefore reconstructs the original tree.

(c) What are the minimum and maximum heights for an AVL tree with 88 nodes labelled $1, 2, 3 \ldots, 88$?

The minimum height for ANY binary tree with n nodes is $h = \lfloor \log_2 n \rfloor$ which is the height of a complete tree with n nodes (all levels full except for possibly the bottom one). Such a tree is an AVL tree as well, so this is the minimum for AVL trees too. In our case $h = \lfloor \log_2 88 \rfloor = 6$.

In class we learned that the minimum number of nodes in an AVL tree of height 8 is 88 so an AVL tree of height 8 with 88 nodes exists. The minimum number of nodes in an AVL tree of height 9 is $88 + 54 + 1 > 88$ and larger heights require even more nodes. So, the max height is 8.