# CSE222 HW7

## Q1

Creating a binary search tree with given data that has the structure of given binary tree is done by a constructor, below you can see it.



Before structure imitating the structure of given binary tree, the tree is created normally. Whose time function can be written as

For copying the structure, it is done by equalizing number of nodes there is in the left subtrees of trees, below you can see the insides of the function.



Base case of this function is to there’s no suitable tree to work on. It is rather hard to calculate average time complexity of this function so its worst and best cases will be calculated. Best case is having left subtree size as correct for every recursive call and trees are complete, so subtrees have almost half of the total number of items. Then the time complexity would be

Since getting size of a tree is linear and subtrees have (almost) half of all nodes.

For the worst case it would be such case that generated binary search tree has every node on one side and the example binary tree has its on the other, so the difference of the size of subtrees would be equal to their size for every recursive call. Then the time complexity would be

**Note:** Asymptotic equal of the function was found by plotting its graph since I couldn't work it out mathematically.

## Q2

Binary tree balancing function is as follows



It calls a recursive function to generate a balanced AVL version of given tree, which you can see the insides below



Since calculating average time complexity is rather hard its best and worst case scenarios will be explored. Best case scenario would be the given tree is already balanced, so neither one of if blocks would be processed but the getBalance methods for every recursive call. Getting balance depends on difference subtree heights so it takes time for a tree with n nodes. Then time complexity would be

The worst case would be having tree leaning on one of its sides in every recursive call and call the function for the third time. getBalance and rotate methods executed in statement blocks don’t really influence it asymptotically since getBalance method is already executed for a bigger tree (root node) and rotate methods are . Now for the third execution to happen there must be at least height difference of 2 (two) between subtrees, and it is always the smaller one that is rebalanced.

Now it is rather hard to compute worst case scenario while thinking for left and right subtrees with different heights, so we will compute as if subtrees have equal heights and yet third balance is executed **still** with same height as they had before third call. This won’t ever happen in real cases and so this is an upper bound. So, time complexity would be

Where .

## Q3

This implementation’s analyzation will be less mathematical for easiness. The base case of insertion to CustomSkipList is it having no head, which then insertion is constant time.

The best case is having the next node that head node is linked to greater than the new node that is to be inserted. This insertion is since the head’s link array should be updated and level of head depends on the number of level one nodes and they are less than the total number of nodes.

The worst case is traversing a skip list whose every node is a tall node until the very end. So, finding nodes that will link to node that will be inserted would take a linear time, there can be maximum of many nodes that can be linked to this node and linking takes time each time. Therefore, time complexity would be .

Since the ratio between level n and level n+1 nodes are averagely constant, the number of different level nodes have an exponential relation. So, on average insertion has a logarithmic time complexity.