# Programming Assignment 2

Gurpreet Singh (150259)
Nikita Awasthi (150453)

## Algorithm for Balanced BST Creation

The basic idea for the insertion function involves inserting a new node as the leaf node by comparing the values of the nodes in the already perfectly balanced BST using an idea similar to Binary Search. After the insertion, the condition for near balance or perfect balance is checked and the nodes are rebalanced in case of an imbalance.
To rebalance the nodes, a sorted array is created from the elements of BST and then a balanced BST is created from the sorted array.

**Analysis of *BSTFromSortedArray* Function**: The function assigns the value of the middle element to the root node and recursively calls for the left and the right subarrays. It involves visiting each element of the array once and is therefore $O(n)$ where n is the number of elements in the array.

**Analysis of *SortedArrayFromBST* Function**: Again, the function just visits each node of tree once, starting from the leftmost node and appends the elements into an array. Since all the nodes are visited just once, the order is again $O(n)$, where n is the number of nodes in the tree.

## Analysis of *Insert* Function

1. **Perfect Balance**

   The order of inserting an element in a perfectly balanced tree is $O(h)$ where h is the height of the BST, which itself is $O(logn)$, therefore inserting is $O(logn)$.

   Let us consider that the BST at present consists of (k-1) nodes and is perfectly balanced, but the addition of the kth node increases the size such that the BST at some node needs to be rebalanced. Now the creation of a sorted array from Balanced BST also requires visiting each element and is O(n).
   $\Rightarrow SortedArrayToBST = O(n)$
   $\Rightarrow BSTFromSortedArray = O(n)$
   As a result, balancing of BST at a node with size k would be $O(k)$.

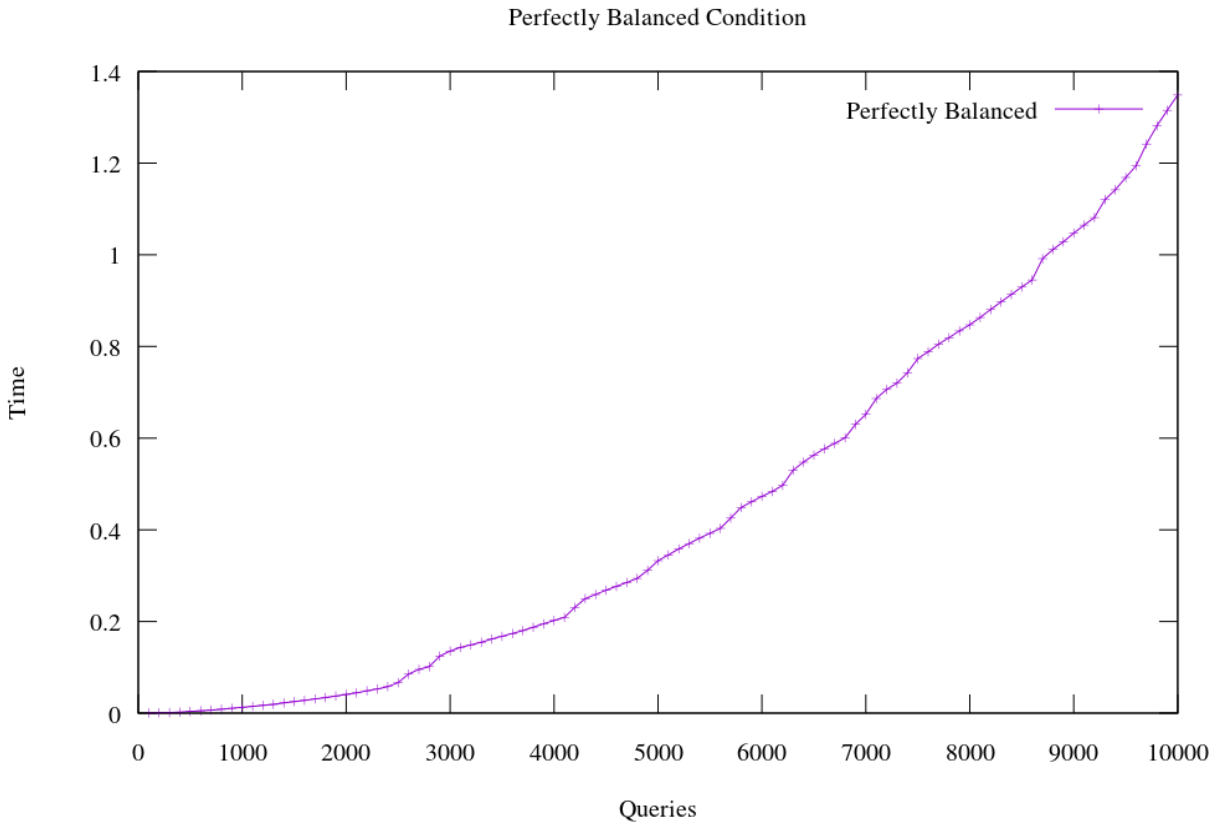
   Now, we need to consider the number of times we would require to rebalance the nodes. For the perfect balanced case,

$$|size(left\ subtree) - size(right\ subtree)| <= 1$$

So, the insertion of one additional node at any point can cause the imbalance. In the worst case, the number of imbalances would be of $O(n)$, where n is the total number of elements to be inserted.

Therefore, the worst case complexity can be obtained as $T(n) = \sum_{k=3}^{n} ck$ as every balancing operation is order k where is the total number of elements inserted. This leads us to the result that the run time complexity of the inserting n elements is $O(n^2)$.

Considering the plot for the perfectly balanced case, we can also guess that the complexity for performing the operation for the perfectly balanced case is $O(n^2)$.



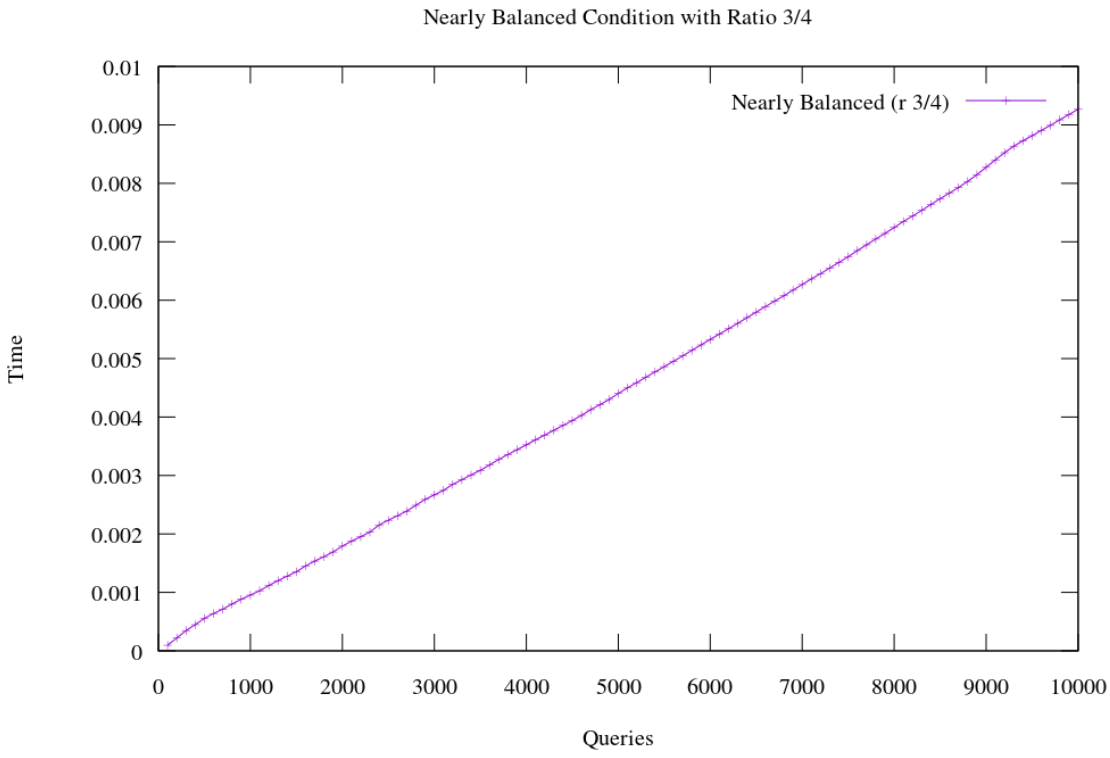

*The jumps in the plot are because of addition rebalancing at certain points*

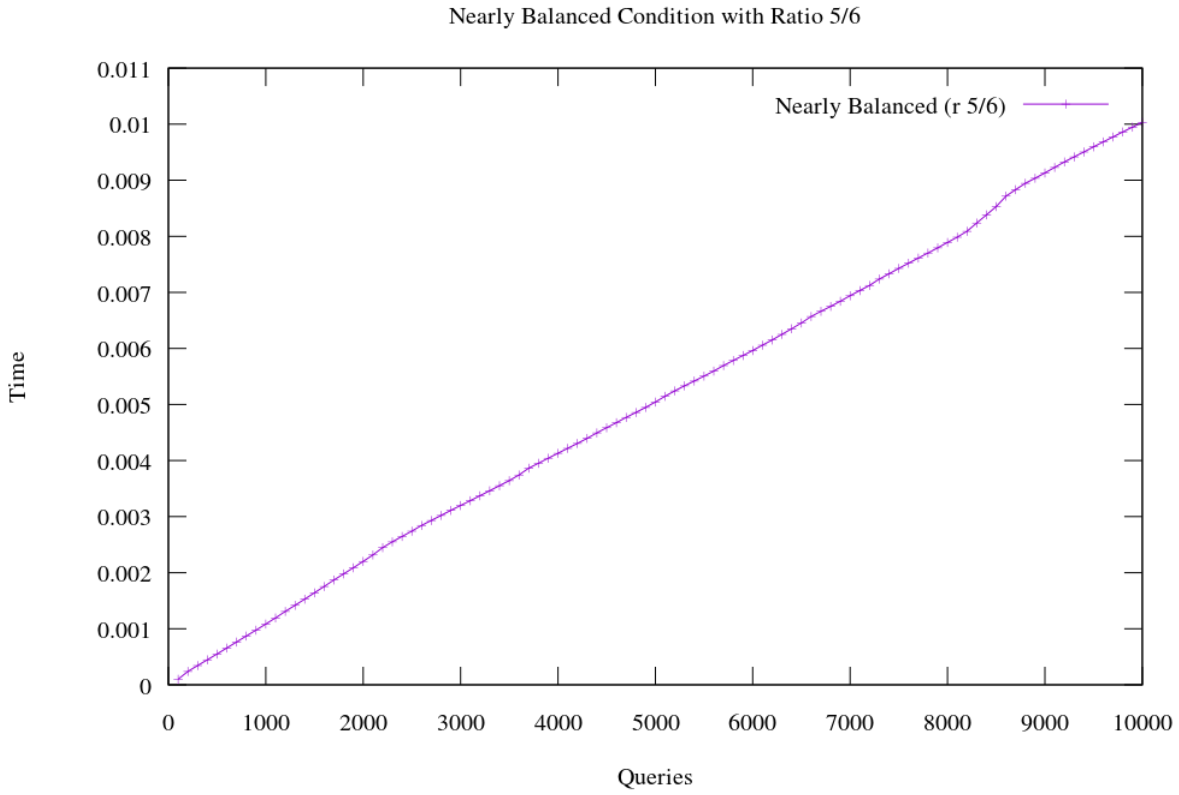
**2. Near Balance**

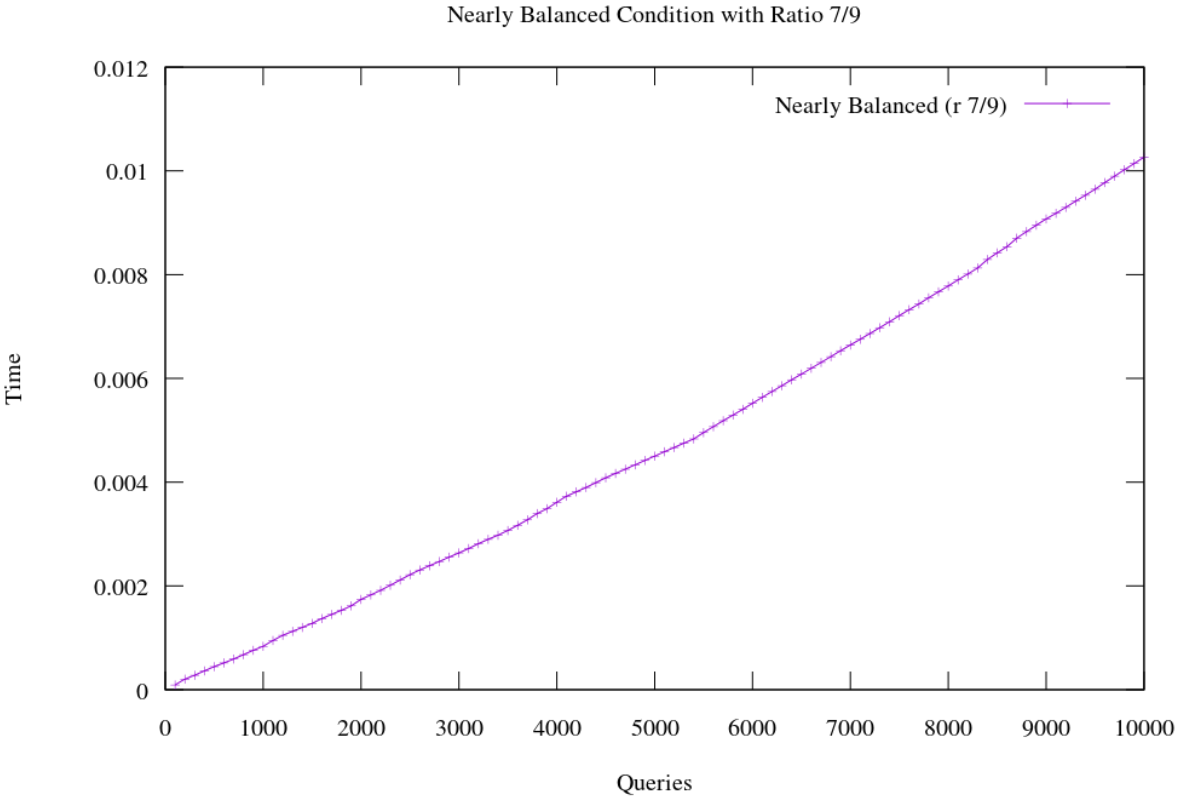
Considering that height of a nearly balanced BST is $O(log_{1/a}n)$ where **a** is the *ratio for the nearly balanced case*, we can say that a normal insert function runs in $O(log_{1/a}n)$ .
And total number of inserts is n, therefore order for inserting will be $O(nlog_{1/a}n)$ . But there will be some cases where we will need to rebalance the tree.

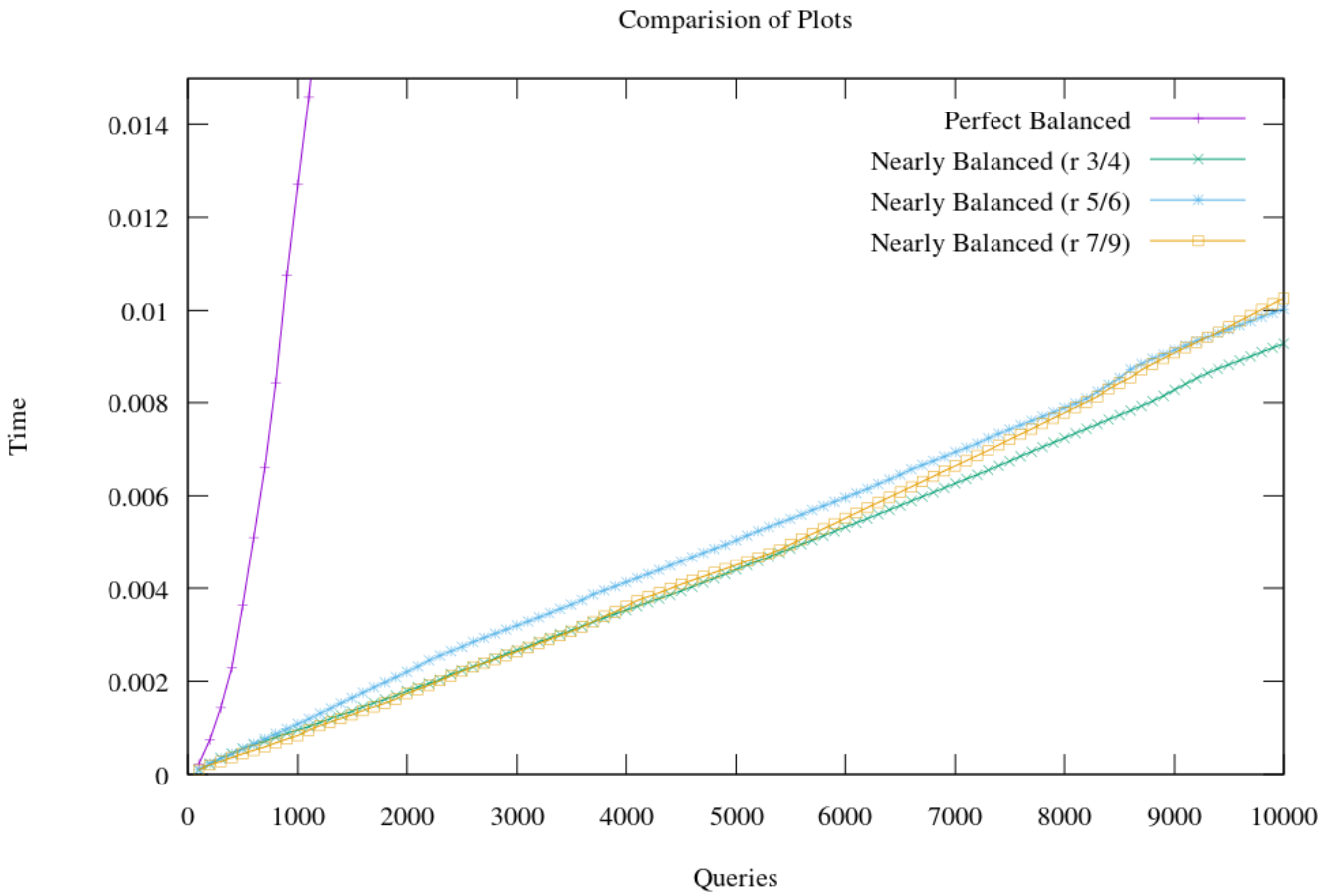
Assume we have reached a point where after inserting a node, we need to rebalance the binary tree. As observed in the earlier analysis of perfect balance condition, we know the order of balancing the nodes is $O(n)$.
To observe the number of times we need to rebalance, we can see that we will need to rebalance if the ratio of nodes in a subtree is greater than a * size of the parent node of this subtree. Assume if we are starting with a perfectly balanced subtree with 'k' number of nodes, then we can say that at least $\frac{1-a}{a} * k + 1$ further insertions on one side of a tree are required to make the tree imbalanced again. Now the size of the complete tree is 'k / a + 1'. Therefore, we can say that the maximum number of times this can happen is $O(log_{1/a}n)$. Hence the total order still remains $O(nlog_{1/a}n)$

Therefore, the complexity of the operation for the nearly balanced case is $O(n\ log\ n)$.
The plots for the varying values of the ratio are as shown in the following plots:




Nearly Balanced Condition with Ratio 5/6
Nearly Balanced (r 5/6)
Time
Queries
0.011
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000


Nearly Balanced Condition with Ratio 7/9
Nearly Balanced (r 7/9)
Time
Queries
0.012
0.01
0.008
0.006
0.004
0.002
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000




The above plot shows the comparison of the running time of the various algorithms. In the analysis for the *perfect balance,* it was shown that the worst case time complexity of the approach is $O(n^2)$. This is further established through the graph comparing the various algorithms. The running time for the perfectly balanced case has a relatively very high order of asymptotic growth as compared to the nearly balanced case for all the three different ratios.
A look at the scale of the graph reveals that the run-time complexity of the *near balanced* is very small as compared to the *perfect balance* which is in accordance to the complexities calculated for the two cases in the analysis presented earlier. The asymptotic growth of $O(n \log n)$ is very small as compared to that of $O(n^2)$.

Considering the three ratios of the *near balance* case, it can be seen from the plots that the as the value of n increases, the time taken is minimum for a=¾. This can be seen from the analysis that the number of times this is required depends on $log_{1/a}n$. The value is minimum for a=¾ and is consistent with our analysis. For the other two values of a, the results are almost identical and that is also seen for the plots.

We can therefore conclude that the nearly balanced case has a better run-time complexity than the perfectly balanced case. For the nearly balanced case, the run-time complexities are almost

similar for the values of a within the same range and dependent on the value of 1/a.