ESO-207 Programming Assignment-2

Anmol Porwal(150108) - Varun Khare(150793)

23 Feburary 2017

1 Introduction

Here we experimentally calculated and analysed the running time on inserting random sequences of length 10000 in a Binary Search Tree while preserving the following conditions of the tree:

1. Perfectly Balanced Binary Search Tree.
2. Nearly Balanced Binary Search Tree with values of size ratio α at each node to be at most:
   - 3/5
   - 3/4
   - 5/6
   - 7/9
   - 99/100

   where size ratio α at a node ν is defined as :

$$\alpha = max\{\|subtree(left(v))\|, \|subtree(right(v))\|\}/\|subtree(v)\| \tag{1}$$

Here we have assumed the following definition of the two trees:

1. Perfectly Balanced BST : A perfectly balanced BST is a BST where for every node ν the difference between the size of the left and right subtree of ν is at most 1.
2. Nearly Balanced BST with ratio α : A nearly balanced BST with ratio α is a BST such that for every node ν in the tree,

$$max\{\|subtree(left(v))\|, \|subtree(right(v))\|\} \leq \alpha * \|subtree(v)\| \tag{2}$$

2 Theoretical Analysis

As seen from the graphs, the time of insertion for the Perfectly Balanced BST is much larger than the Nearly Balanced counter parts. This is because of continuous re-balancing after every two insertions (worst case) in Perfectly Balanced BST. Here we calculate the time complexity for arbitrary n insertions for each of the above cases and hence justify the observations.

But, First we prove the following lemma:
Lemma1: the time complexity of rebalancing a tree in O(n).
proof:

- Time Complexity analysis for BSTFromSortedArray:
  refer Figure:2. Let The time complexity be S(n).

```
200  if(NotBalancedNode != NULL)
201  {
202      SortedArrayFromBST (NotBalancedNode ,out,0  );
203      if( UnBalancedPapa == NULL )
204          curr =  BSTFromSortedArray (out , NotBalancedNode -> size );
205      else{       if( UnBalancedPapa -> left == NotBalancedNode   )
206          UnBalancedPapa -> left = BSTFromSortedArray (out , NotBalancedNode -> size );
207          else
208              UnBalancedPapa -> right = BSTFromSortedArray (out , NotBalancedNode -> size );
209      }
210  }
```

Figure 1: part of main code in insert to re-balance tree

```
56 Node *BSTFromSortedArray(int input[], int len){
57     if( len == 0)
58     {   Node *curr = NULL;
59         return curr;
60     }
61     else if( len == 1  )
62     { Node *curr = init( input[0]  ) ;
63         return curr;
64     }
65 
66     else
67     {
68         Node *curr = init( input[len/2]  ) ;
69         curr -> size = len ;
70 
71         curr -> left = BSTFromSortedArray( input , len/2    );
72         if ( len != 2 )
73             curr -> right = BSTFromSortedArray( (input+len/2+1) , (len - len/2 - 1)  );
74         return curr;
75     }
```

Figure 2: code for BST from sorted array

```
40  SortedArrayFromBST(Node *curr,int output[],int fst){
41  int temp= fst;
42  if(curr == NULL )
43      return 0;
44  fst+=SortedArrayFromBST( curr -> left , output , fst   );
45  output[fst] = curr -> val;
46  fst++;
47  fst+=SortedArrayFromBST( curr -> right , output , fst );
48  return fst-temp;
```

Figure 3: code for sorted array from BST

- The first If block and the else if block has O(1)
- lines 68-69 has O(1)
- at line 71 since size of array half time complexity is S(n/2).
- lines 72-74 are O(1).
- similarly at line 73 time complexity S(n/2).

Therefore equation becomes:

$$S(n) = 2 * S(n/2) + O(1) \tag{3}$$

By master theorem S(n) = O(n).

- Time Complexity analysis for SortedArrayFromBST:
  refer Figure:3. Let the time complexity be G(n).
  - line 41-43 is O(1).
  - line 45-46 is O(1).
  - line 48 is O(1)
  - line 44 and 47 we take it to be G(n/2) each. (Here one can argue that exact time complexity would be G(βn) and G((1-β)n) where $\beta \in$ (0,1). this is indeed the case but we are considering the worst case time which comes out to be for $\beta = 0.5$. We need to maximize G(βn) + G((1-β)n) hence we differentiate it wrt to β and equate to 0 which gives us $\beta = 0.5$. Also when we calculate this sum for $\beta = 1/4$ it comes out to be O(log(n)) hence this comfirms that $\beta = 0.5$ is indeed the optimum case.)

  Therefore equation becomes:

$$G(n) = 2 * G(n/2) + O(1) \tag{4}$$

  By master theorem G(n) = O(n).

Now we get back to the main code.refer to Figure:1 Let time complexity of This part of code be T(n).

- at line 202 time complexity = O(size of NotBalancedNode)
- now any one of the three conditions at lines 203,205,207 are executed with each of complexity = O(size of NotBalancedNode)

Hence final complexity of balancing subtree of a node ν of size n = O(n).

$$T(n) = G(n) + S(n) + c = O(n) \tag{5}$$

Hence proved.

2.1 Perfectly Balanced BST

Consider the root node(ν) after j^{th} insertion as perfectly balanced with T(left) and T(right) as sub-trees. Now in the worst case we need to maximize the number of re-balancing at ν. This would occur if all the insertions go to only one sub-tree and hence we need to balance the T(ν) after every two insertions.
Hence the time taken for rebalancing after j+2 insertion is O(j+2). Additional O(log j+2) time is taken for traversal along the tree from ν to leaf. Thus time complexity for j+2 insertion is O(log(j+2) + j+2). So if re-balancing occurs at j^{th} insertion then T(j)=O(j+log(j))

Finally for n arbitrary operations in BST:

$$T(n) = \sum_{j=1}^{n} O(log(j) + j) \tag{6}$$

This is because amongst searching (O(log(n)), insertion and deletion (O(1)); insertion is most computationally expensive. Thus for upper bound we take n insertions in the BST. On Further Evaluation

$$T(n) = \sum_{j=1}^{n} (O(log(j)) + \sum_{j=1}^{n} O(j) = O(nlog(n) + n^2) = O(n^2) \tag{7}$$

2.2 Nearly Balanced BST

Let us consider a BST 'T' of size ratio α and size n. Also $T_l(\nu)$ and $T_r(\nu)$ are the left and right sub-trees w.r.t any node ν. By Nearly Balanced condition we get

$$H(n) \leq H(\alpha n) + 1 \leq H(1) + \lceil -log_\alpha n \rceil \tag{8}$$

Where H(n) is height of BST with size n.
Since H(1)=0 therefore

$$H(n) \leq \lceil -log_\alpha n \rceil \tag{9}$$

Starting from a perfectly balanced BST of size k we need at least y (say) more nodes to unbalance the tree. Then,

$$k/2 + y = \alpha(k + y) \tag{10}$$

$$y \geq k(2\alpha - 1)/(2 - 2\alpha) \tag{11}$$

$$y \in O(k) \tag{12}$$

For a node ν, define $I_k(v) =$

$$\begin{cases} 1 & if\ size\ of\ \nu \text{ increases on } k^{th} \text{ insertion} \\ 0 & otherwise \end{cases}$$

We are taking $I_k(\nu) = 0$ if $\nu \notin T$ i.e node ν has not been inserted in T till k^{th} insertion. The height of above Nearly balanced tree is O($log_\alpha k$) and each insertion increases the size of the nodes lying only along its path therefore,

$$\sum_{\nu \in T} I_k(\nu) = O(-log_\alpha k) \tag{13}$$

The above equation is applied to entire tree. Time complexity for rebalancing a specific node ν is

$$Time \in O(k + y) = O(k) \tag{14}$$

Following k^{th} insertion (from start) rebalancing occurs again after x more insertion (say). As total number of nodes after x insertions(following k^{th} insertion) is k+x. The total number of insertions for which $I_k(v) = 1$ can be greater than y due to insertions on different sub tree. Now,

$$\sum_{r=k+1}^{k+x} (I_r(\nu)) \geq y\ (\in O(k)) \tag{15}$$

This gives for n insertions at the node ν

$$TimeComplexity = O(k) \subseteq O(\sum_{r=1}^{n} I_r(\nu)) \tag{16}$$

Therefore for all the vertexes we get,

$$Time(n) = \sum_{\nu \in T} \sum_{r=1}^{n} I_r(\nu) = \sum_{r \in [n]} \sum_{\nu \in T} I_r(\nu) = O(-nlog_\alpha n) \tag{17}$$

3 Comparison Perfectly Balanced vs Nearly Balanced

From the above we have proven that Nearly Balanced trees require much less computation (O(nlog(n)) for n arbitrary operations while Perfectly Balanced takes O(n^2). This justifies the graphs where perfectly balanced tree take much more time for insertions.
Also increase in α decreases the value of $O(-nlog_\alpha n)$. Therefore, as we go on increasing the size ratio, the Time required decreases and graphs' slope decreases though marginally. This validates the plot for $\alpha \in \{0.6, \frac{5}{6}, \frac{7}{9}, \frac{99}{100}, \frac{3}{4}\}$

4 Plots

The graphs were obtained by plotting the time required in inserting k number of nodes while maintaining the desired condition versus k using gnuplot, where the data was obtained by modifying the C++ code provided along with the report to obtain the time for insertions in the tree. This time was calculated for various number(around 1000) of randomly generated input arrays of size 10000 and then the average of these times was used to plot the graphs to get a proper estimate. The graphs are given below:

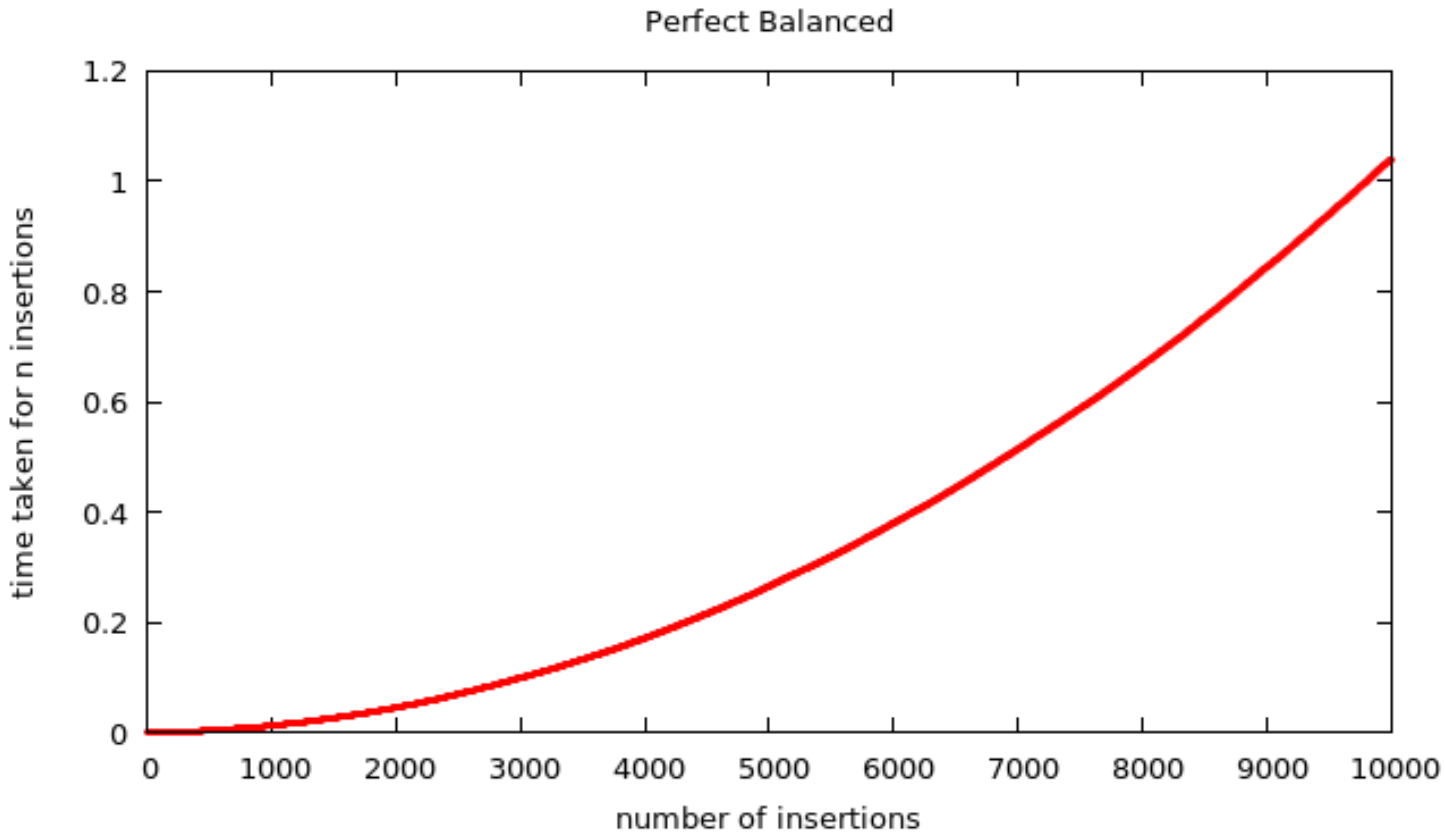

Figure 4: Graph for perfectly balanced condition

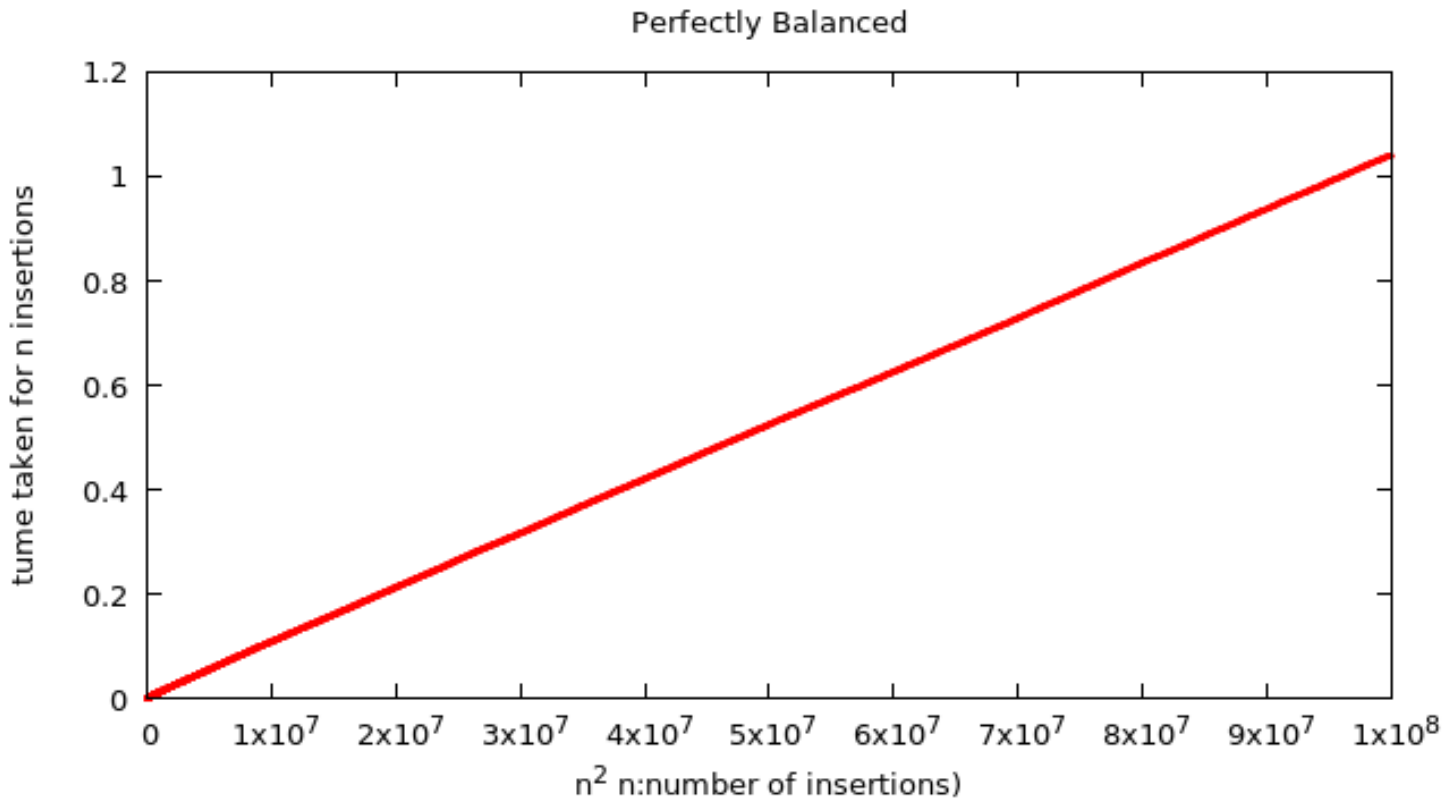

Figure 5: linearity of time with n^2 for perfectly balanced

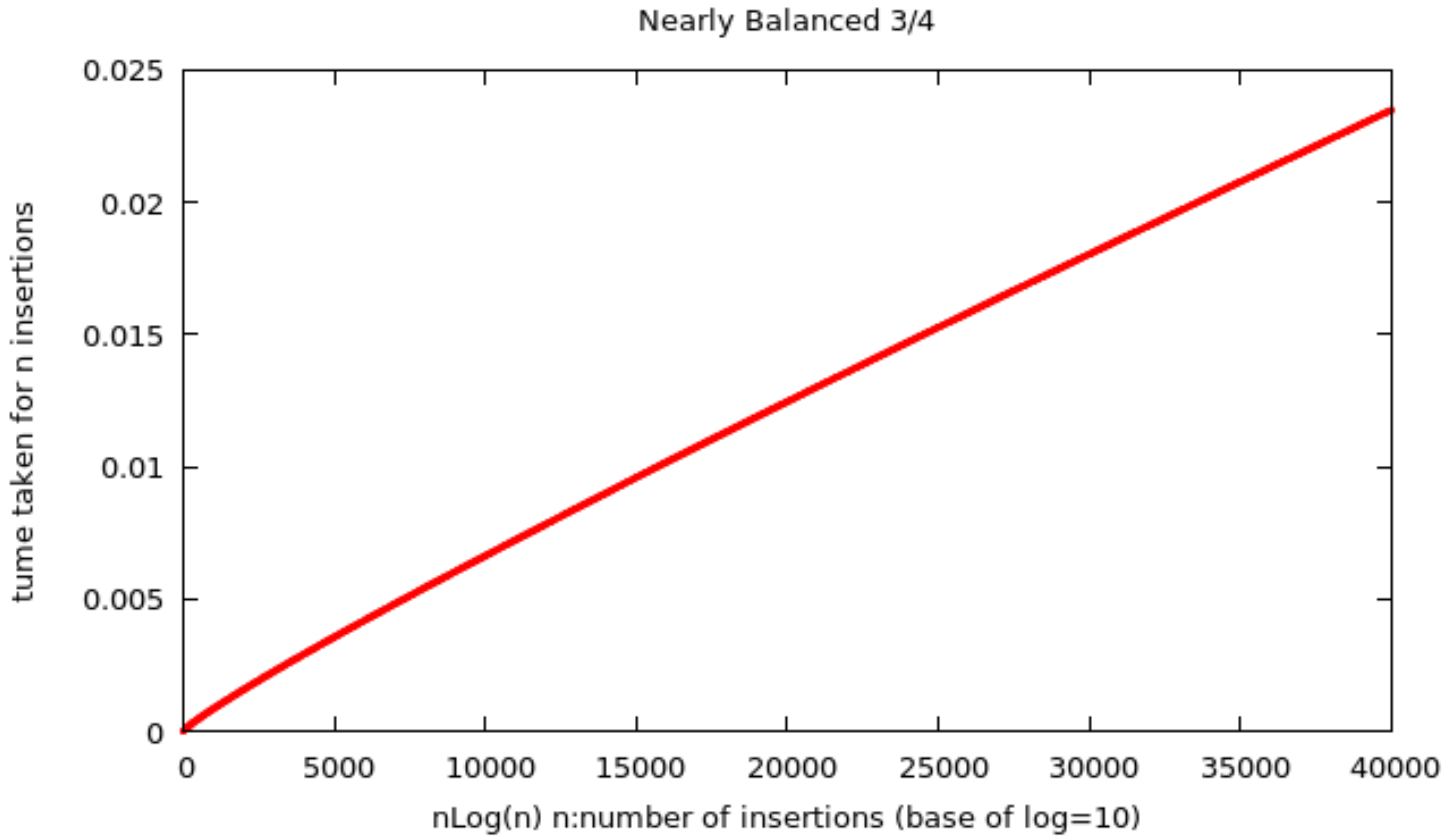

Figure 6: Graph for nearly balanced condition with balance ratio 3/4.This is a linear curve showing that time complexity is nlog(n).

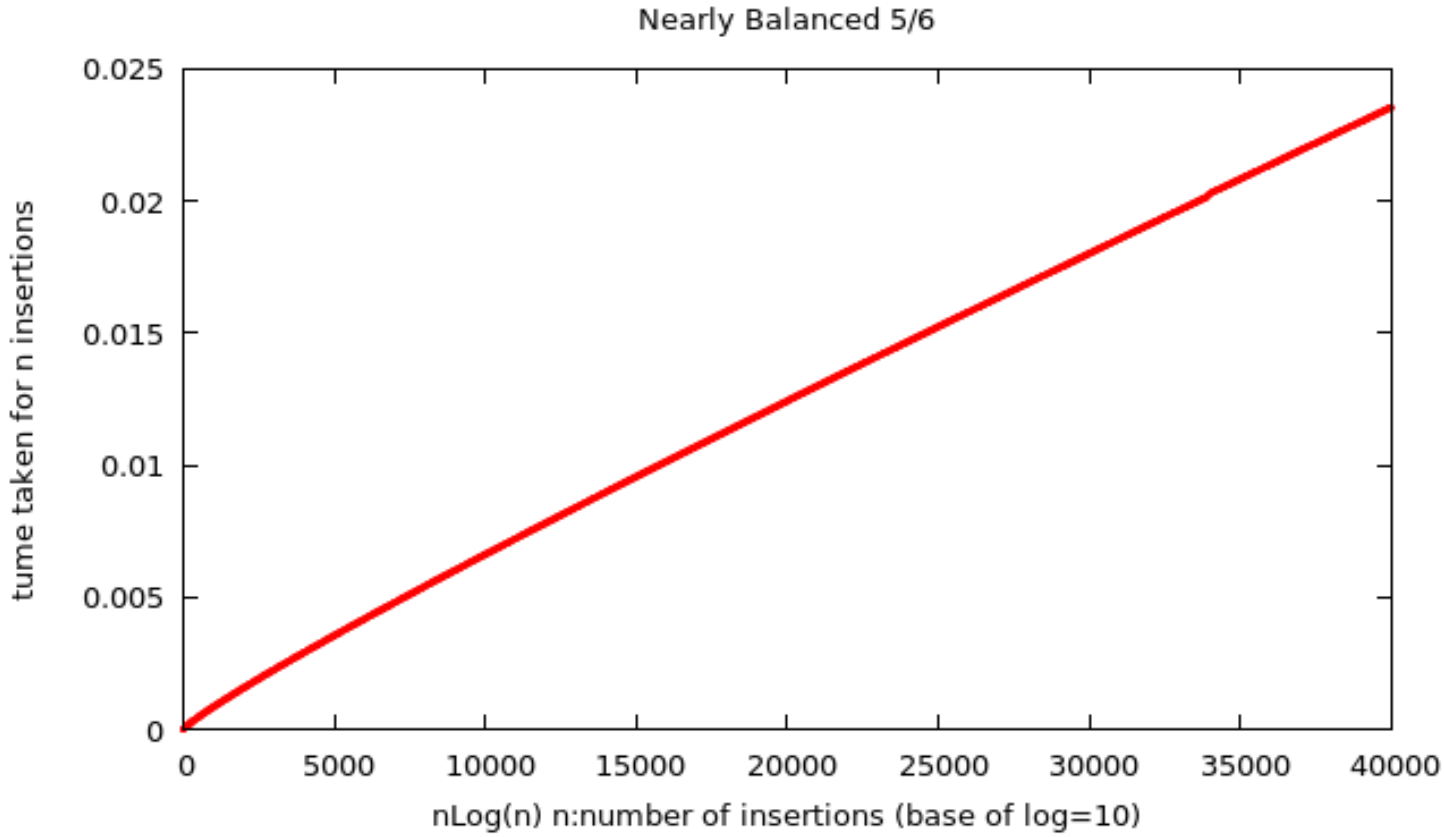

Figure 7: Graph for nearly balanced condition with balance ratio 5/6.This is a linear curve showing that time complexity is nlog(n).

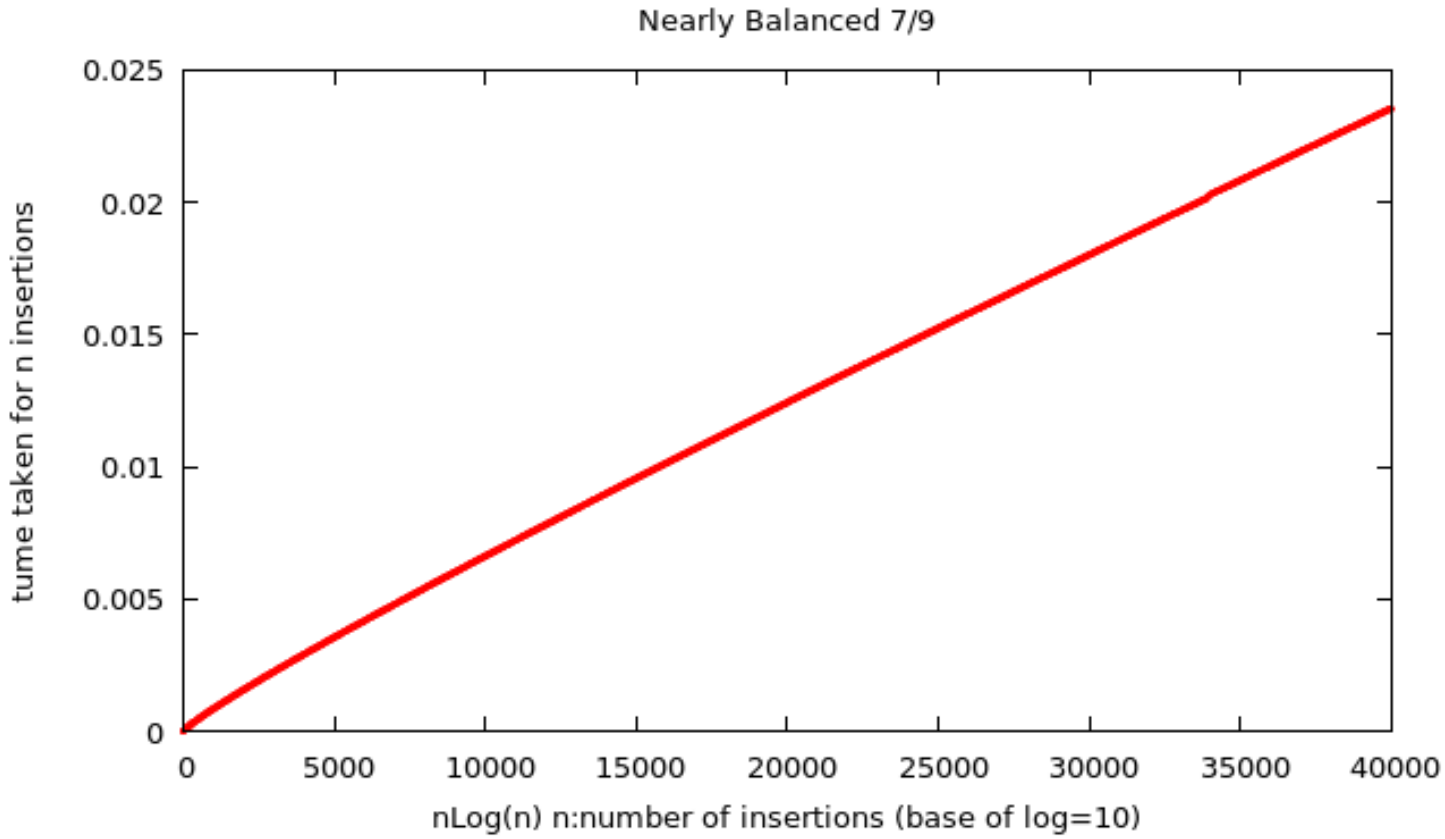

Figure 8: Graph for nearly balanced condition with balance ratio 7/9.This is a linear curve showing that time complexity is nlog(n).

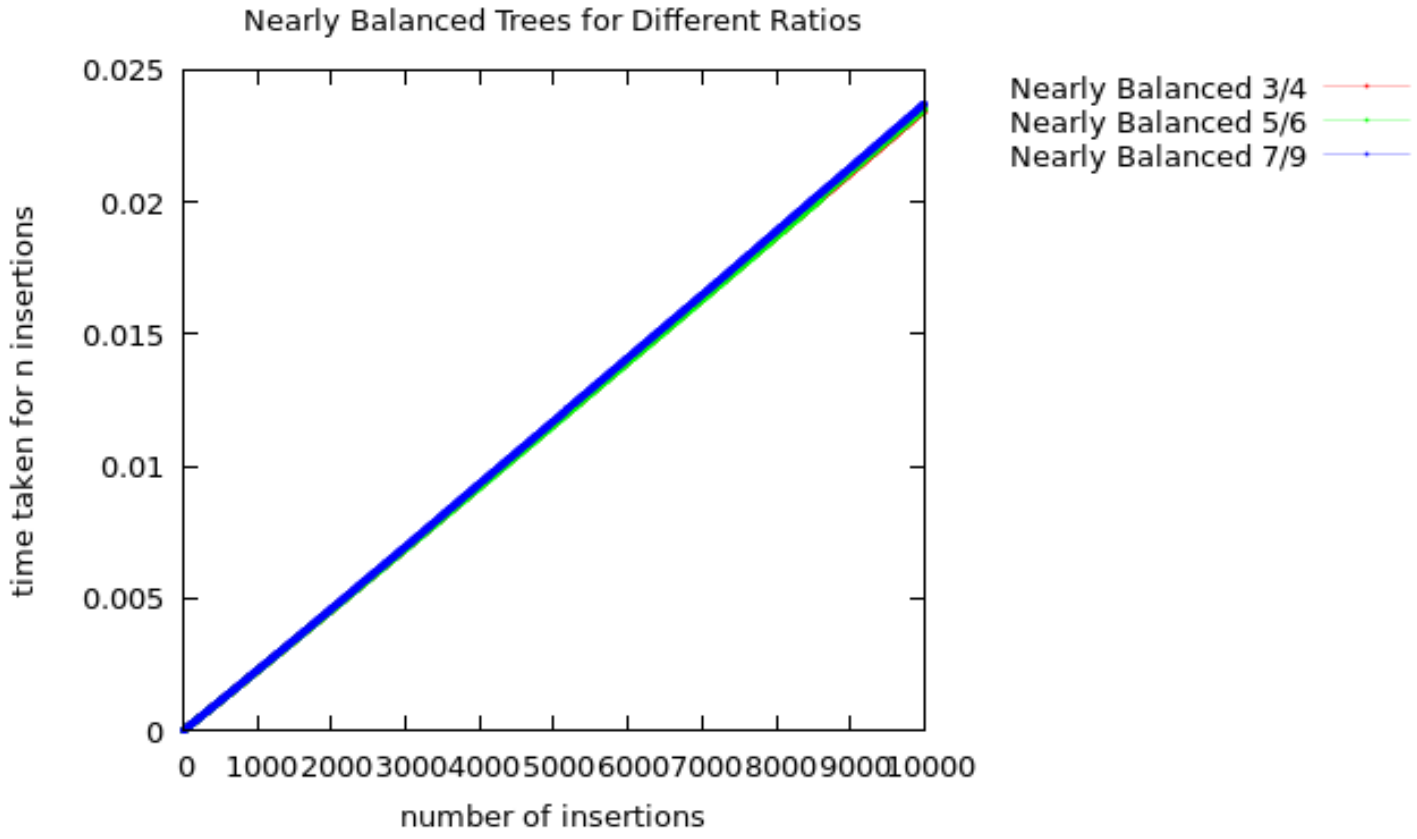

Figure 9: Graph for nearly balanced condition with the three ratios

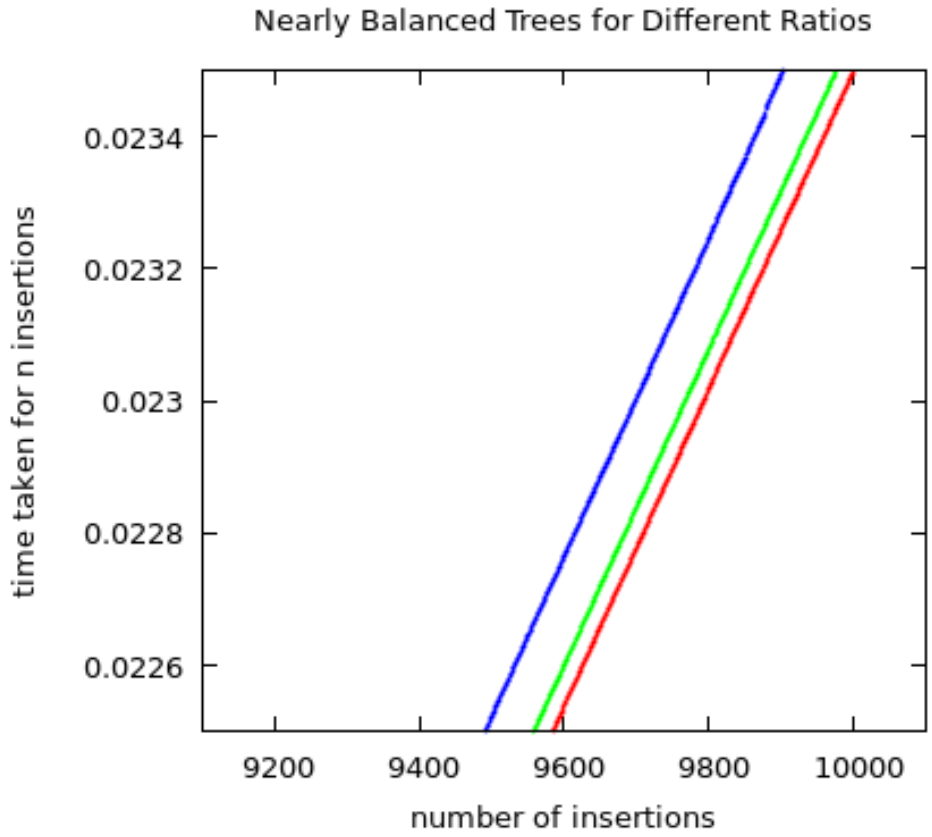

Figure 10: Graph for nearly balanced condition with the three ratios zoomed.Here we can observe that time for 7/9 >5/6 >3/4

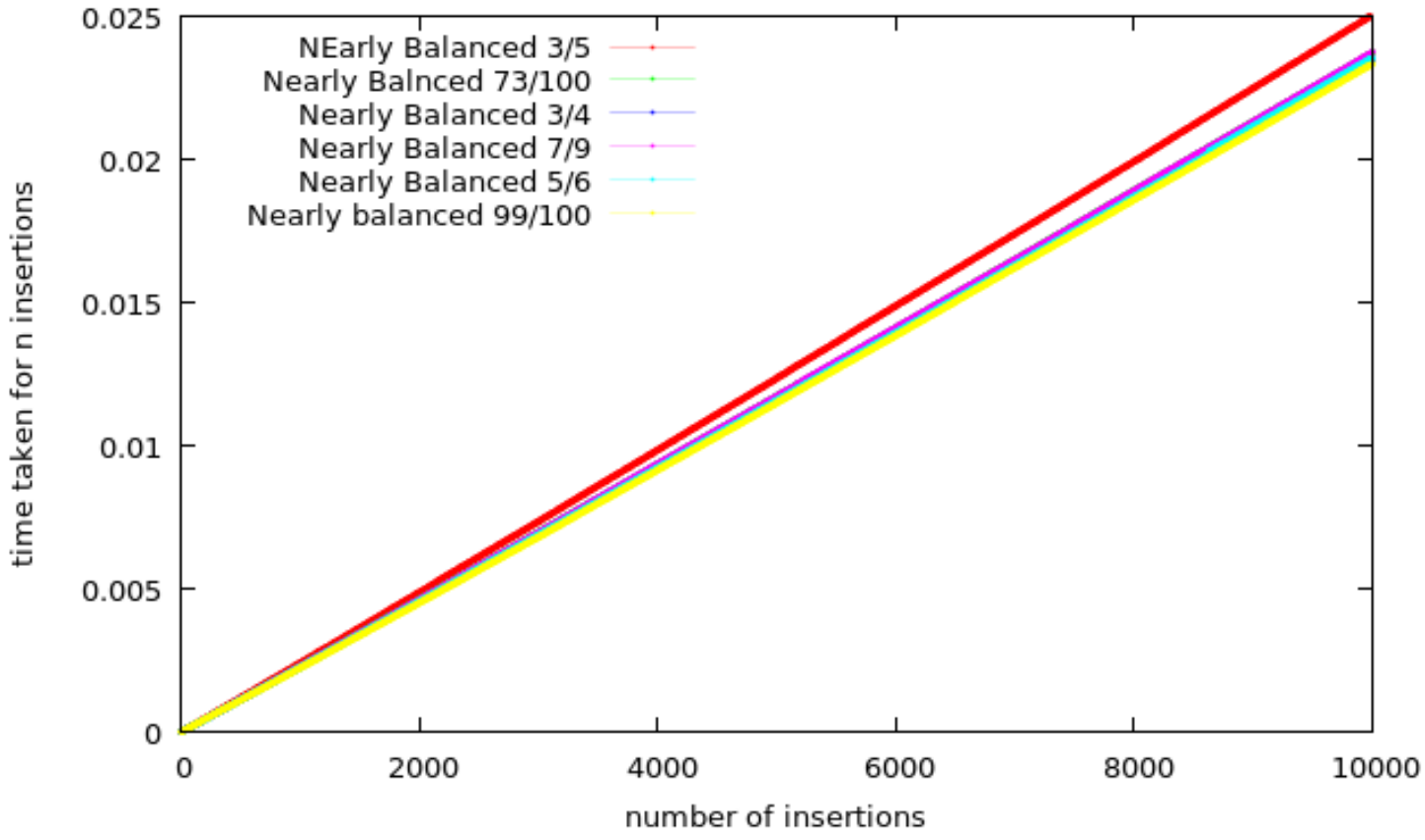

Figure 11: Graph for nearly balanced condition with all 5 ratios. Here we can observe that as the ratio decreases the time increases.

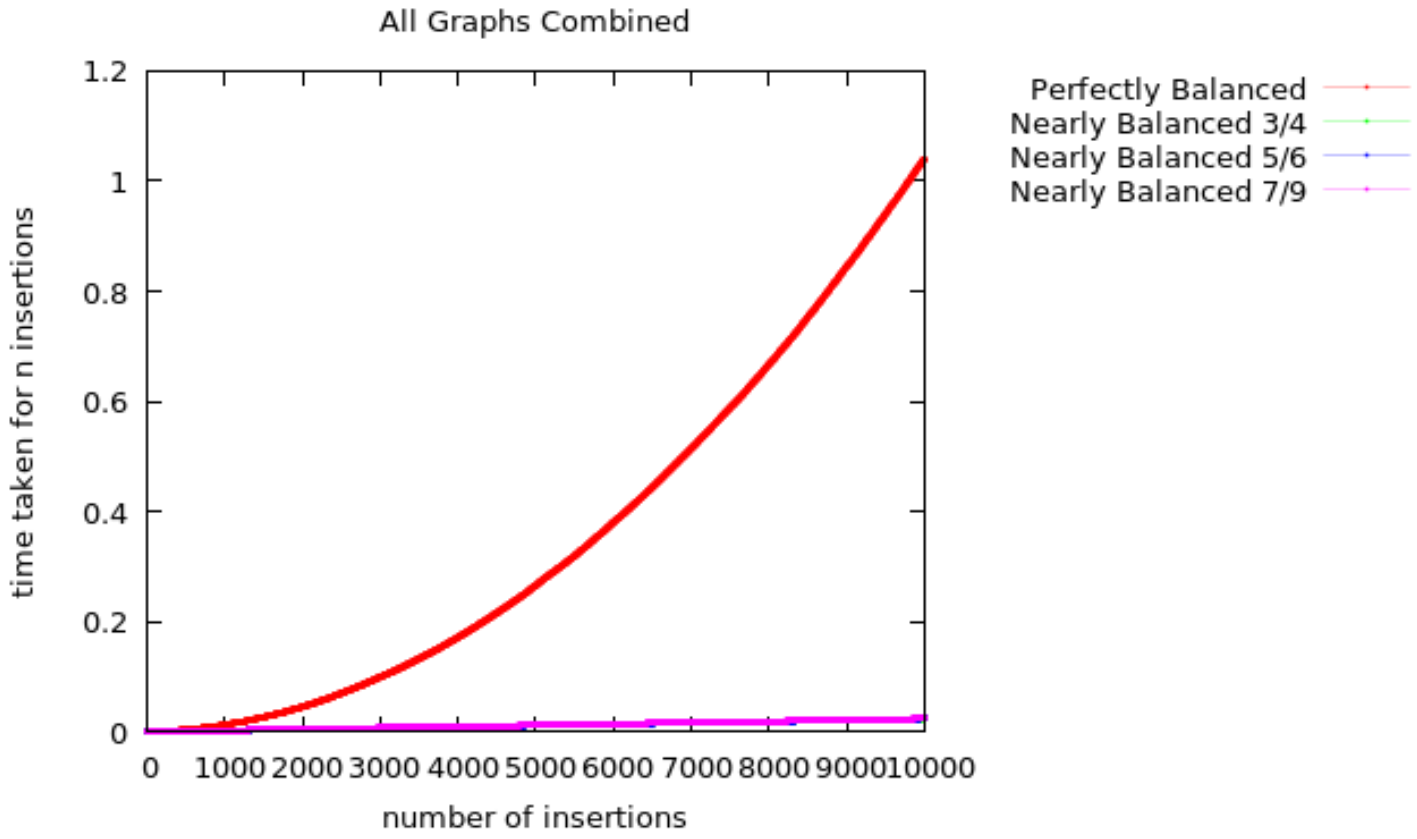

Figure 12: Combined graph for perfectly balanced and nearly balanced