CS 202
Fundamental Structures of Computer Science II
Fall 2017
Assignment 2 Solutions

Section 1
Selim Firat Yilmaz
21502736
firat.yilmaz@ug.bilkent.edu.tr

November 6, 2017

Contents

1 Question 1 **3**
1.1 (a) BST Traversals 3
1.2 (b) BST Insert/Remove 3
1.3 (c) Construction of a Binary Tree 4

2 Question 3 **4**
2.1 Performance Analysis 4
2.2 Height Analysis 5

3 Appendix **6**
3.1 Performance Analysis Output 6
3.2 Height Analysis Output 7

1 Question 1

1.1 (a) BST Traversals

Preorder Traversal: R, G, L, A, O, H, I, T, M
Inorder Traversal: A, L, G, O, R, I, T, H, M
Postorder Traversal: A, L, O, G, T, I, M, H, R

1.2 (b) BST Insert/Remove

Figure 1: After desired elements inserted to an empty BST

Figure 2: After desired elements removed from the BST above

1.3 (c) Construction of a Binary Tree

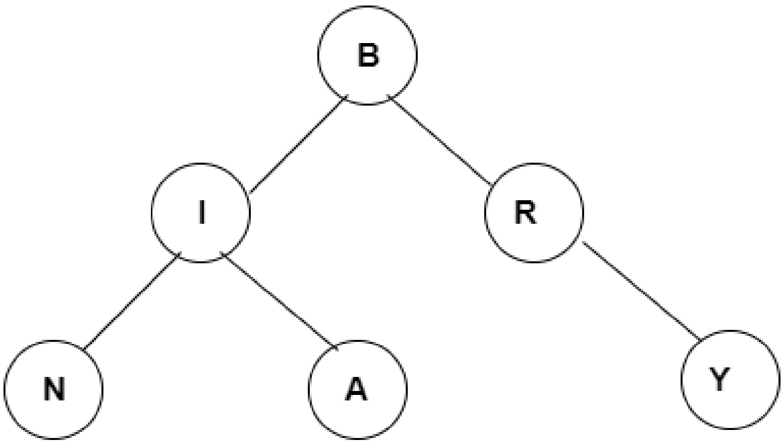

Figure 3: Constructed Binary Tree

Inorder Traversal: N, I, A, B, R, Y

2 Question 3

2.1 Performance Analysis

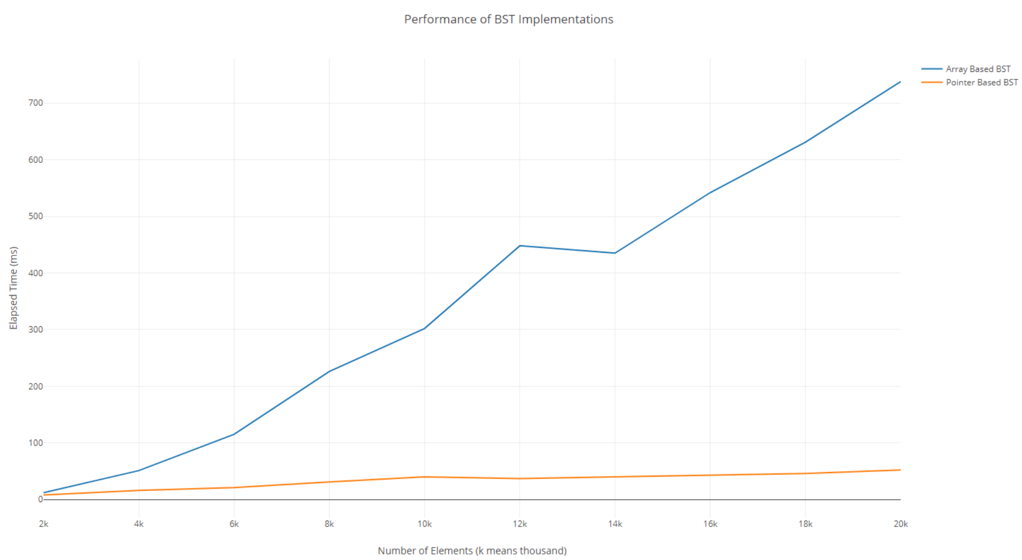

Figure 4: Performance Analysis of BST Implementations

The only requirement in pointer based BST for insertion of an element is to find the place for it. Thus, it is $O(log_2 N)$ where N is the number of elements in the BST. As I expected, pointer based BST works faster on # inputs in range of [2k, 20k] because of that array based implementation requires resizing of array on each iteration in addition to finding place to insert new items.

It should be remembered that in each resizing, all elements needed to be copied to new array. When N is the number of elements in the array, array will be resized $\lceil log_2 N \rceil$ times. Assume N is power of 2 for simplicity. Then, in each resizing there will be 2^i elements to copy where $i = 2, 3, ..., log_2 \frac{N}{2}, log_2 N$. Thus, in total there will be $2N - 2$ copy operation in total, which is $O(N)$. It roughly it matches well with the graph.

If we need to do insertion operations often, the pointer based BST is better because it requires much less time than array based BST.

2.2 Height Analysis

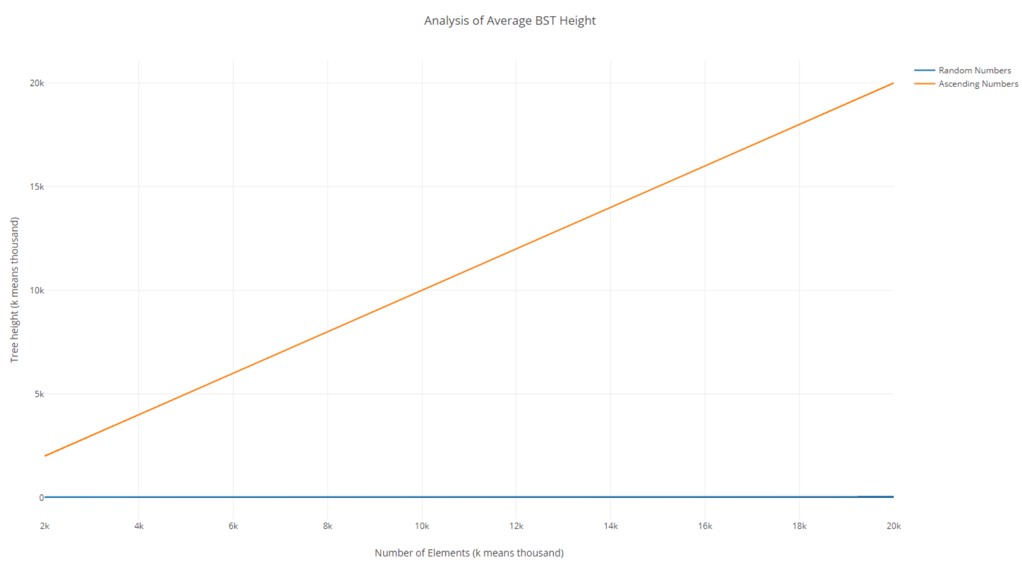

Figure 5: Height Analysis of Different Cases

For clarity, I would like to attach the plot with only ascending numbers test:

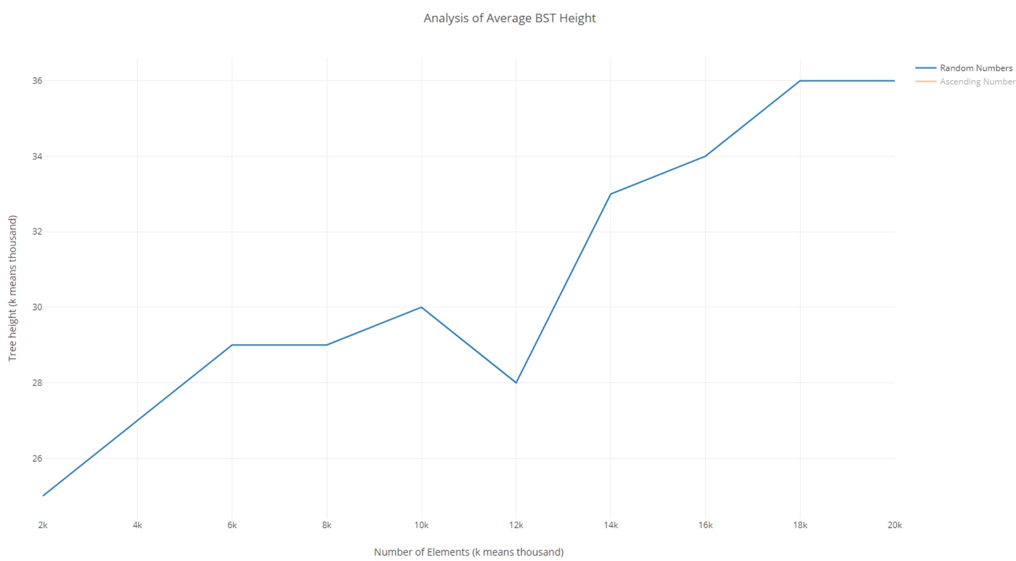

Figure 6: Height Analysis of Different Cases

The worst case in terms of height for a BST is when elements are sorted in ascending or descending order. As is in our experiment, when elements are sorted in ascending order, the height of the BST is exactly equals to the number of elements on the BST.

In theory, the average height of BST is $O(logn)$ where n is the number of elements in the tree. In practice, we expect it to be some constant factor of $log_2 n$ because it is what we have as a result of our experiment. Also, it cannot be $log_2 n$ because it is theoretical optimum value and it is improbable. Thus, it is bigger than $log_2 n$ but still $O(logn)$. The results are in the Appendix below.

In order to prevent worse case from happening while inserting a predefined set of data into BST, we could shuffle the data we have. Since it is unlikely that shuffling would result in a sorted data, we would have achieve our goal to prevent worst case.

3 Appendix

3.1 Performance Analysis Output

```
Part e - Performance analysis of BST implementations
----------------------------------------------------------------
```

Array Size	Array Based	Pointer Based
2000	12 ms	8 ms
4000	51 ms	16 ms
6000	115 ms	21 ms
8000	226 ms	31 ms
10000	302 ms	40 ms
12000	448 ms	37 ms
14000	435 ms	40 ms
16000	542 ms	43 ms
18000	631 ms	46 ms
20000	738 ms	52 ms

3.2 Height Analysis Output

Part f - Analysis of BST height

Array Size	Random Numbers	Ascending Numbers
2000	25	2000
4000	27	4000
6000	29	6000
8000	29	8000
10000	30	10000
12000	28	12000
14000	33	14000
16000	34	16000
18000	36	18000
20000	36	20000