Lab 3

Leo Man
Jesper Persson
Gustav Hammarström
Department of Engineering Sciences and Mathematics

January 9, 2025

1 Project layout

In the project you will find the following. This report. The datastructure **D** from the assignment implemented in python, under the name BalancedBST.py. As well as a standard binary search tree named StandardBST.py. An some scripts that tests the binary search trees.

2 Implementation of data structure

The insertions part of the method is the same as the standard binary search tree. Checking if the data point to be inserted is smaller or larger then the current node. If smaller go to the left node and check if there is a node/tree there if not recursively insert the data point on this node. Other wise (larger or equal) do the same but for the right node/tree.

The data structure as described in the assignment requires that the tree contains its own size. And since no other operation on the data structure other than insertion is to be implemented all of the size logic is handled in the insertions method.

The size of each node is updated during the post-recursion phase. This happens as the recursion "unwinds," ensuring that the sizes of all nodes on the insertion path are recalculated from the leaf node back up to the root. Specifically, the size of a node is calculated as:

$$\text{size} = 1 + (\text{size of left subtree, if it exists, otherwise } 0) + (\text{size of the right subtree if it exists, otherwise } 0) \quad (1)$$

Directly after the size is calculated, the balancing condition is checked. If

$$\text{child tree} > \text{parent tree } \cdot c \quad (2)$$

then subtree rooted at the current node is restructured into a perfectly balanced binary search tree containing the same keys.

The re-balancing is done in three steps. Firstly getting the current tree keys as an sorted array with in order traversal. Then building an perfectly balanced tree from that array. And lastly replacing the current sub tree with this tree.

3 Run time of insertion

The insertion method for the data structure has two different cases one where only insertion is made and when insertion and rebalancing is made.

When no rebalancing is done at all in combination with the worse case, data points are inserted in sorted ascending or descending order. The tree will be skewed so far that the tree will be an linked list. Meaning that the time complexity for one insertion

$$O(n) \quad (3)$$

And if a tree perfectly balanced the time complexity of one insertion is

$$O(log\, n). \quad (4)$$

meaning that dependent on how well the tree is balanced the the insertion time complexity for one insertion is in the range $O(log\, n)$ to $O(n)$.

The case when re-balancing and insertion is done. The time complexity of the re balancing also has to be taken to account. The decision comparisons and the size calculations are all constant. And the re-balancing time complexity is $O(n)$ since all nodes of the subtree has to be visited in the in order traversal. And the something goes for building the balanced tree.

Therefore when re balancing has to be made we get

$$O(n\,) + O(log\, n) = O(n) \quad (5)$$

And for for n insertions in series we get the time complexity ranging from

$$O(n\, log\, n) \longrightarrow O(n^2) \quad (6)$$

dependent of the value c.

3.1 Simulation experiments

And upon running a the BalancedBST datastructure with values of c ranging from 0.51 to 0.99, when inserting 10 000 datapoints we can clearly see how it takes longer time to insert the elements when c is close to 0.5. And

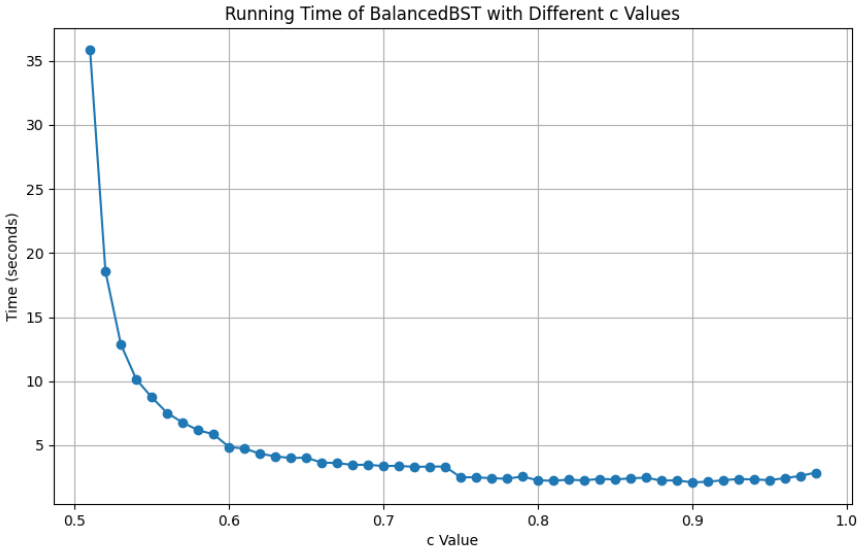

Figure 1: Number of operations on the worst case

running the simulation on the range of c = 0.6 to 0.99 when inserting 100 000 data points. we can see that

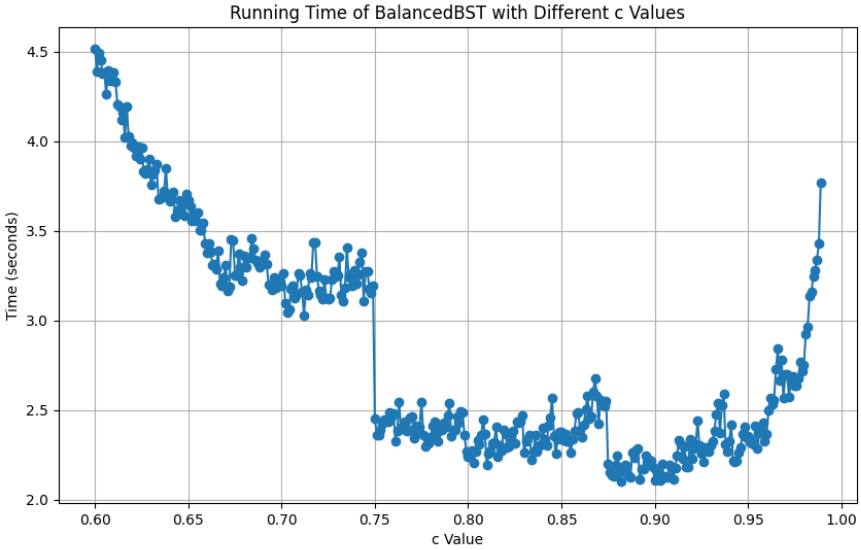

Figure 2: Number of operations on the worst case

when there are very few re-balancing c is higher it preforms somewhat worse then when more re-balancing is done, since the tree has deeper depth and the insertion part of the algorithm is more computational heavy as described before.

And when running a simulation while inserting values in sorted descending order we get this. Where for the

Type	**Insert Size**	**Execution Time (s)**	**Constant c**
BalancedBST	1,000	0.0091	0.750
BalancedBST	10,000	0.1301	0.750
BalancedBST	100,000	1.8338	0.750
BalancedBST	1,000,000	30.5197	0.750
BalancedBST	10,000,000	425.6526	0.750
BalancedBST	100,000,000	5092.4707	0.750
StandardBST	1,000	0.0215	-
StandardBST	10,000	1.5517	-
StandardBST	100,000	160.0259	-
StandardBST	1,000,000	Too long	-
StandardBST	10,000,000	Too long	-
StandardBST	100,000,000	Too long	-

Table 1: Execution Time Comparison for BalancedBST and StandardBST sorted data insorted

balancedBST we get an time complexity that is roughly proportional to $O(n\,log\,n)$ and for the standardBST we can clearly see that the time complexity is $O(n^2)$ just as expected.

Running the standardBST on random data

Type	**Insert Size**	**Execution Time (s)**	**Constant c**
BalancedBST	1,000	0.0051	0.750
BalancedBST	10,000	0.0459	0.750
BalancedBST	100,000	0.6180	0.750
BalancedBST	1,000,000	11.6541	0.750
StandardBST	1,000	0.0008	-
StandardBST	10,000	0.0105	-
StandardBST	100,000	0.1717	-
StandardBST	1,000,000	4.2787	-

Table 2: Execution Time Comparison for BalancedBST and StandardBST on randomly inserted data

And upon testing different c values and seeing how it grows, we get. These graphs and as we could already tell from figure 1 and 2. The algorithm preforms the best with a c value around 0.9

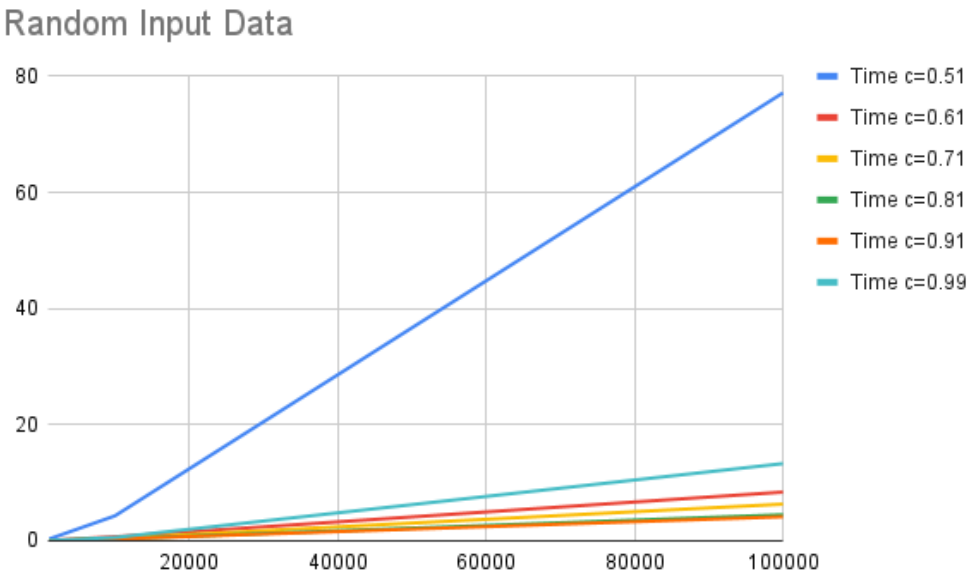

Figure 3: Running time growth for random input data

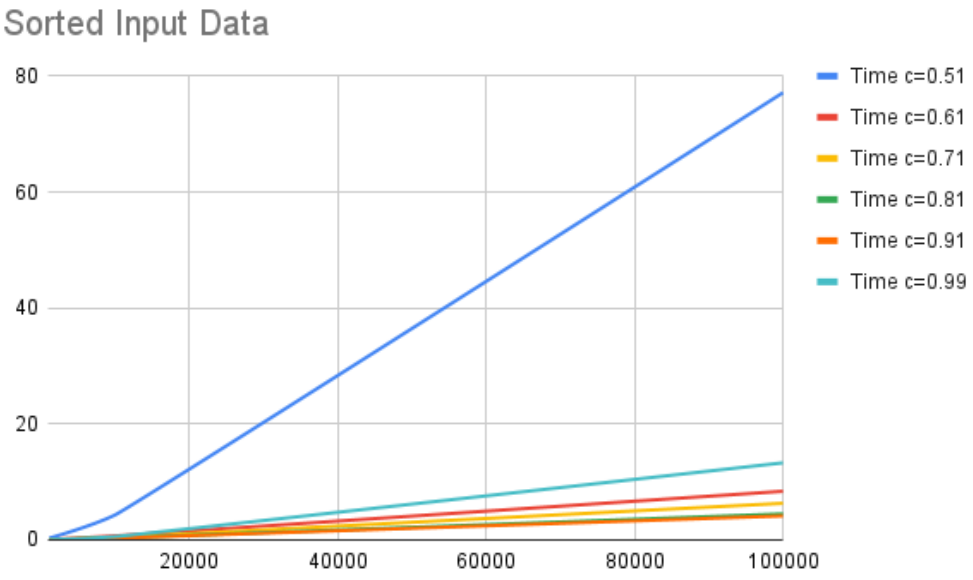

Figure 4: Running time growth for sorted input data