Introduction and opening remarks:

In this project the objective was to examine the performance, features and general behavior of the three following data structures: The Ordered List(being a type of a dynamic list with one-directional search mechanism), The Binary Search Tree, and The AVL tree – a self-rebalancing type of a Binary Search Tree – in the above table, one can find time measurements taken whilst the afore mentioned structures were performing operations such as inserting, searching for a particular record or removing a record from the list.

Records are of type:

```
3135343   rgxlynfweogk   vsvgjsztaktr
3074282  ldarzwxyrhcy    hohxocipmxlr
5800291  ndbywaehmsla  gcxadwllrjjm
9830830  otwuxgsbcvfr    tblajxkumgfi
```

Having a unique ID-key and 2 randomly generated strings(name and surname of the student)

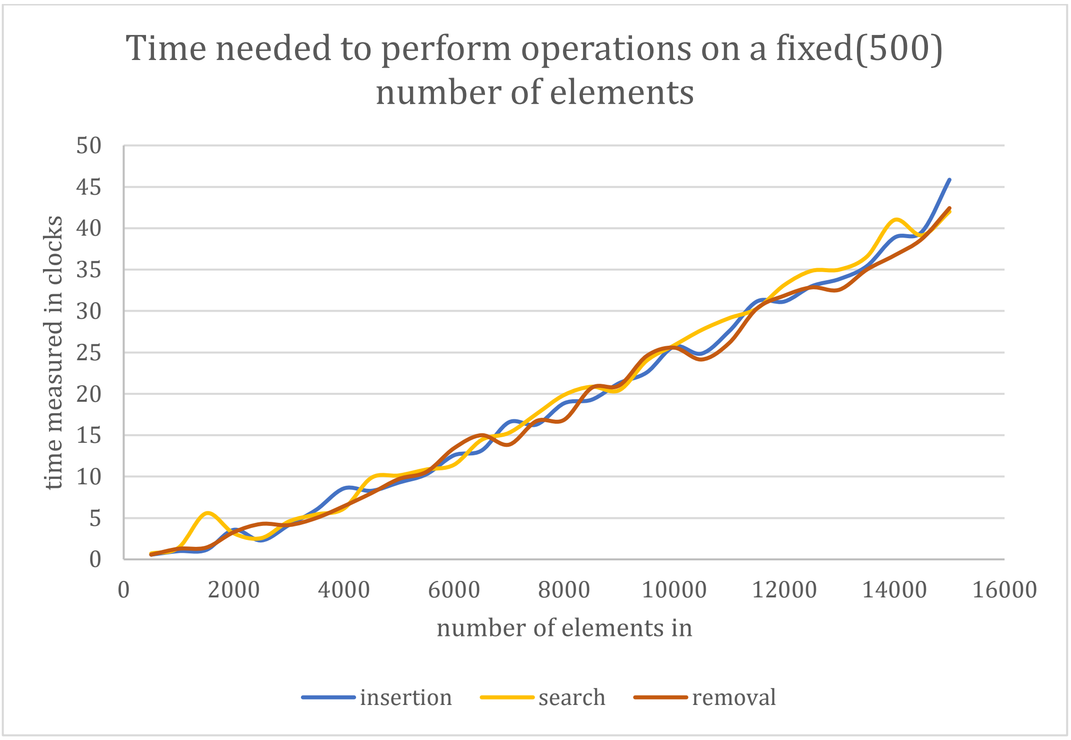
No. of el.	AVL insert	AVL search	AVL remove	BST insert	BST search	BST remove	O. L insert	O. L search	O. L removal
2000	0	0	2.28	2.28	0	0	7.8	24	9.4
4000	3.14	2.57	1	6.57	0	0	37.4	83	48.2
8000	5.57	2	4.85	6.57	0	5.57	188.3	395	205.1
16000	12.28	5.42	10.14	7.14	4.71	8.85	826.2	1721.5	879.1
32000	24.85	11.14	22.14	18.42	12.28	16.85	3544.6	7558.4	3824.7

O.L = ordered list

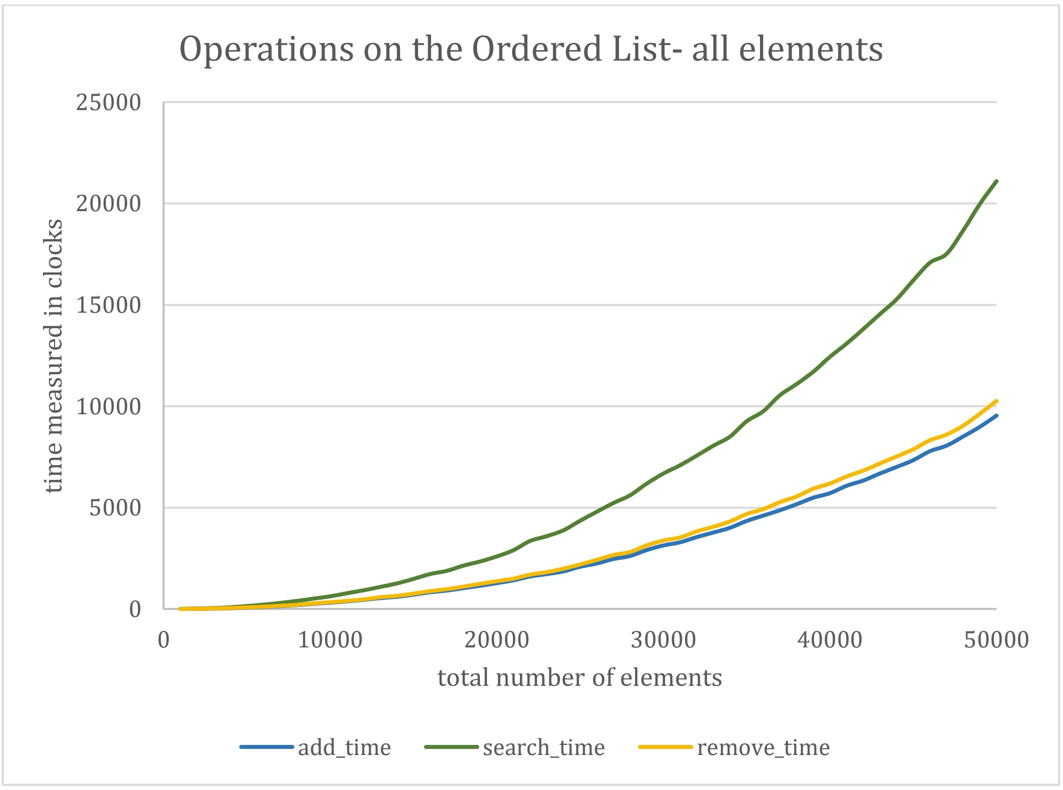
I Ordered List

Looking at the table provided, one can observe that the time needed for the Ordered List to perform the listed operations roughly quadruples with the elements doubling in number. This is a clear indicator that the functions(measuring the time needed to insert, search for and remove **all** of the items) is O(n^2), this is particularly visible for the searching function. Searching for an item in an Ordered list is of complexity O(n) – in the worst case, the algorithm would go through the whole list, only for the searched element to be found at the end. The same hold true for the insertion and deletion of a single element(although additional operations of reassigning the pointers have to be carried out- however those are of O(1) complexity, so the deletion and insertion of elements are both O(n)- this is evident from looking at line chart1.

In the case of insertion and deletion all of the elements in an ordered list(line chart 2), less time is taken up as opposed to the searching of all of the elements, because of the non-constant size of the ordered list, while all of the elements are either added or removed from it. In this respect the maximum number of operations(passing other elements in search of the record to be deleted / proper place for an element to be added) is $\frac{k\times(k+1)}{2} = \sum_{k=0}^{n} k$, which explains why the time for these processes to complete is twice as low as for the searching of n elements to complete(since the input is randomly generated).



line chart 1

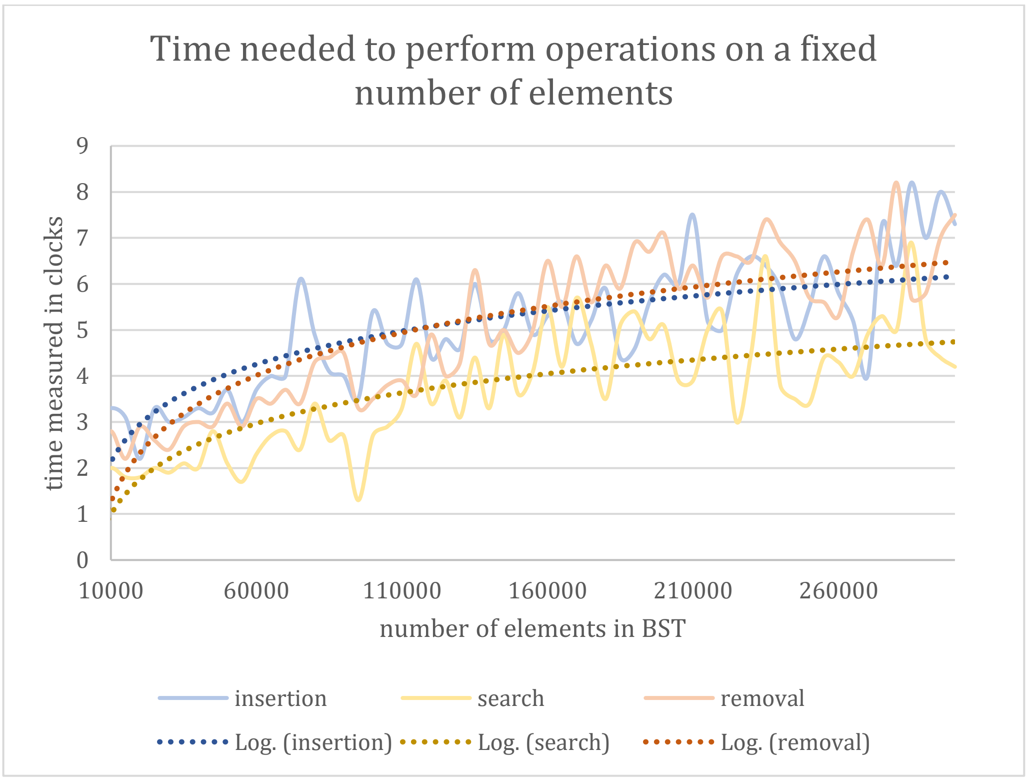


line chart 2

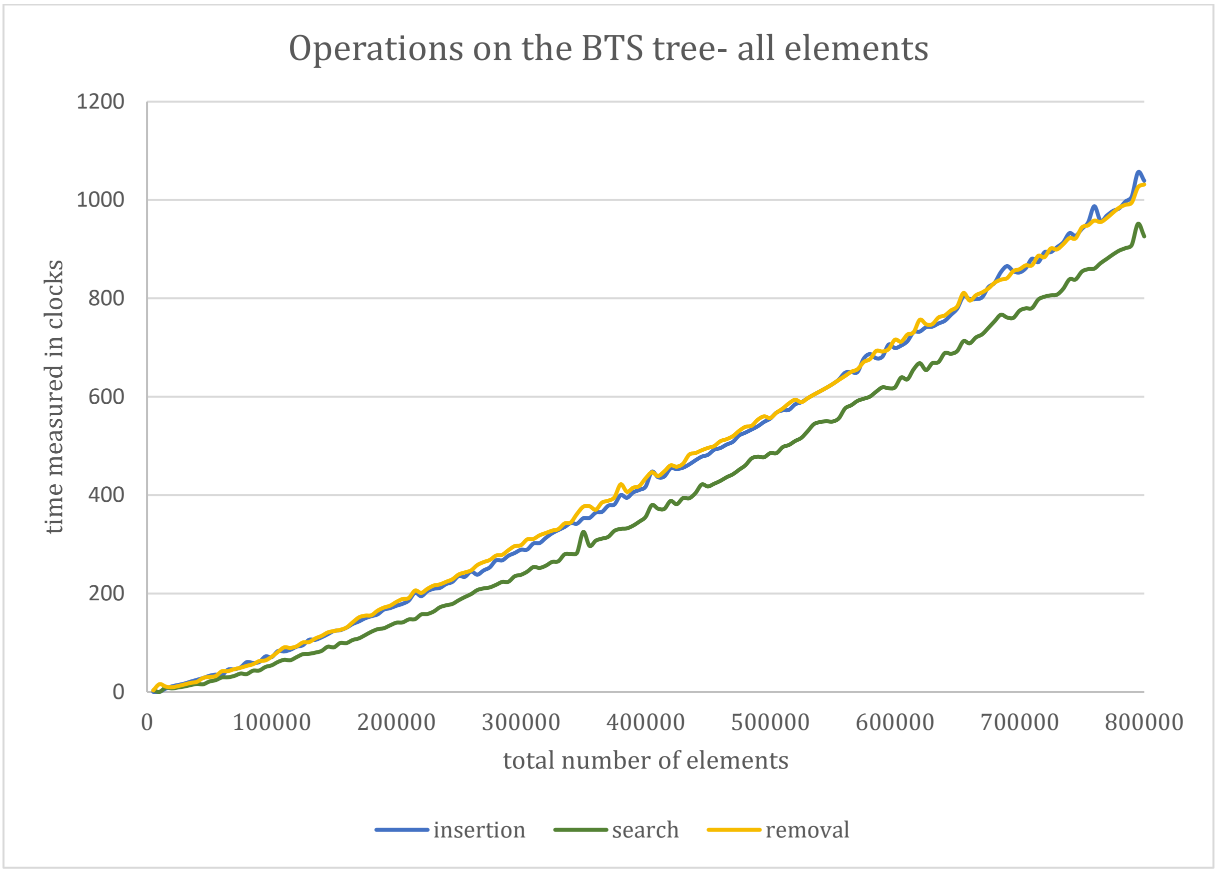
II BST

In contrast to the ordered list, even form looking at the chart illustrating the time needed to perform operations on the BST(**all** of the nodes), one can state that it is definitely not O(n^2), rather O(nlog(n)), this was tested by performing the operations on each element of the list. However a test measuring how much time it takes for the processes to complete, when one only performs a fixed number of operations (i.e. in our case 5000) was also conducted . In the latter – although the data is randomized, so it is not apparent, that the worst cases(inserting/searching/removing) are O(h), h- being the height of the binary search tree (at the end of the text, a table with sample heights of BSTs is provided), which in the worst case can be even n, this happens when a sorted array is fed into a BST(it essentially becomes an Ordered Linked List), but on randomized input data this is highly unlikely to happen. From the table provided, one can conclude that the average height of a BST would be approximately 3*log(n) – basing this estimation on the small data set – so the tested operations are therefore θ(log(n)). Knowing that the time may vary due to not only diverse heights but also positions of elements in the BST, one can nonetheless conclude that the trend line (line chart 3) fits the data nicely.

As for the different operations carried out, one can observe that all of them perform similarly both in terms of their mechanism as well as in terms of their time complexity, particularly the deletion and insertion are close to each other on both charts – below them, the searching operation, which executes faster, this is due to the fact that it does not need to perform any “additional actions” such as switching/substituting nodes like the two other ones, although (line chart 4) it operates on a larger number of elements (number of elements vary while removing/ inserting nodes – but contrary to the list – this does have a huge impact since number of comparisons while going down the tree is correlated to the height which is ~3*log(n).



line chart 3



line chart 4

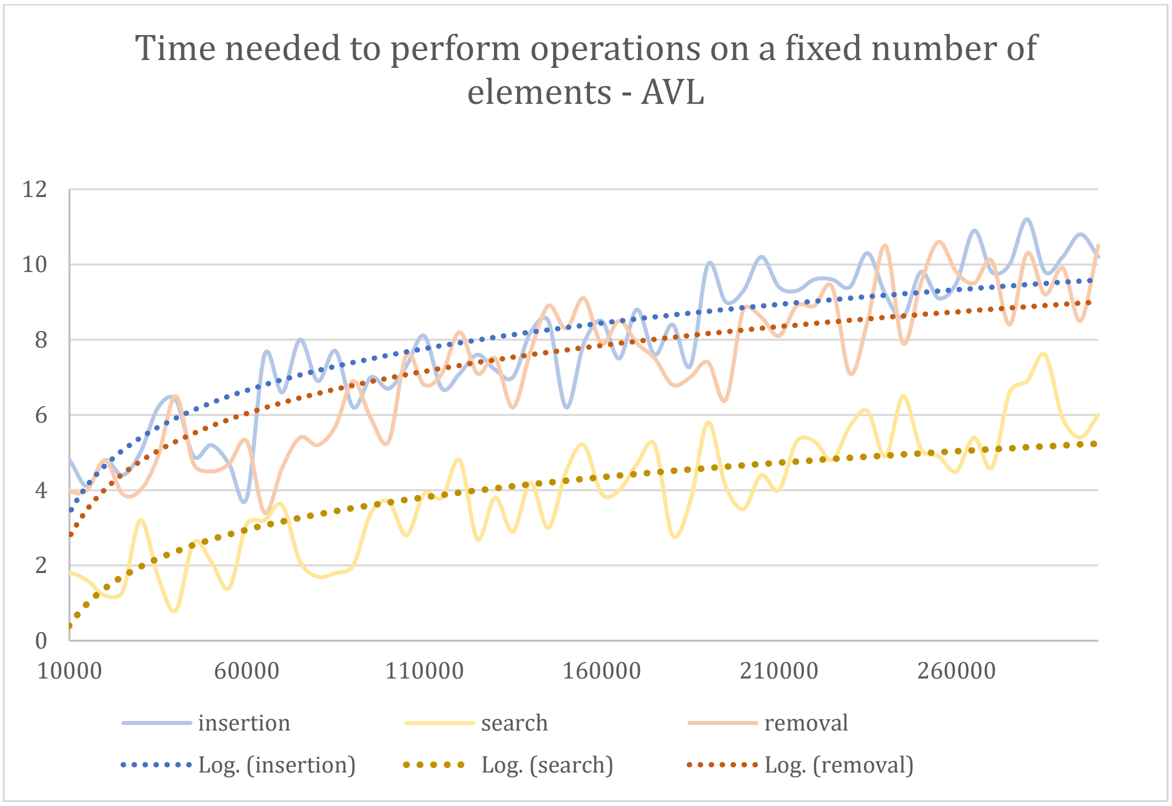
III AVL binary search tree

The AVL tree is a subtype of the binary tree in which for each node, the difference in heights of subtrees doesn't exceed 1. For this to be possible a number of actions have to be carried out both when deleting and adding a record to the tree. Due to this balancing, one can see on the charts below, that the above named operations take up much more time to be executed, even taking into account that when removing/ inserting a node the tree ends up/ starts out as a single node and throughout the whole process is varying in size(the latter is true also for the usual non-self-balancing binary search tree).

The insertion operation does not differ greatly from the usual BST insertion algorithm, with the distinction that at most two rotations have to be performed, for the tree to keep its balance. Because of the fact that the height of a AVL tree is at most 1,44*log(n), the insertion algorithm has a complexity of O(logn), since the rotations are O(1), assuming the height of nodes is computed non-recursively, and stored as an attribute(which is the case in our project).

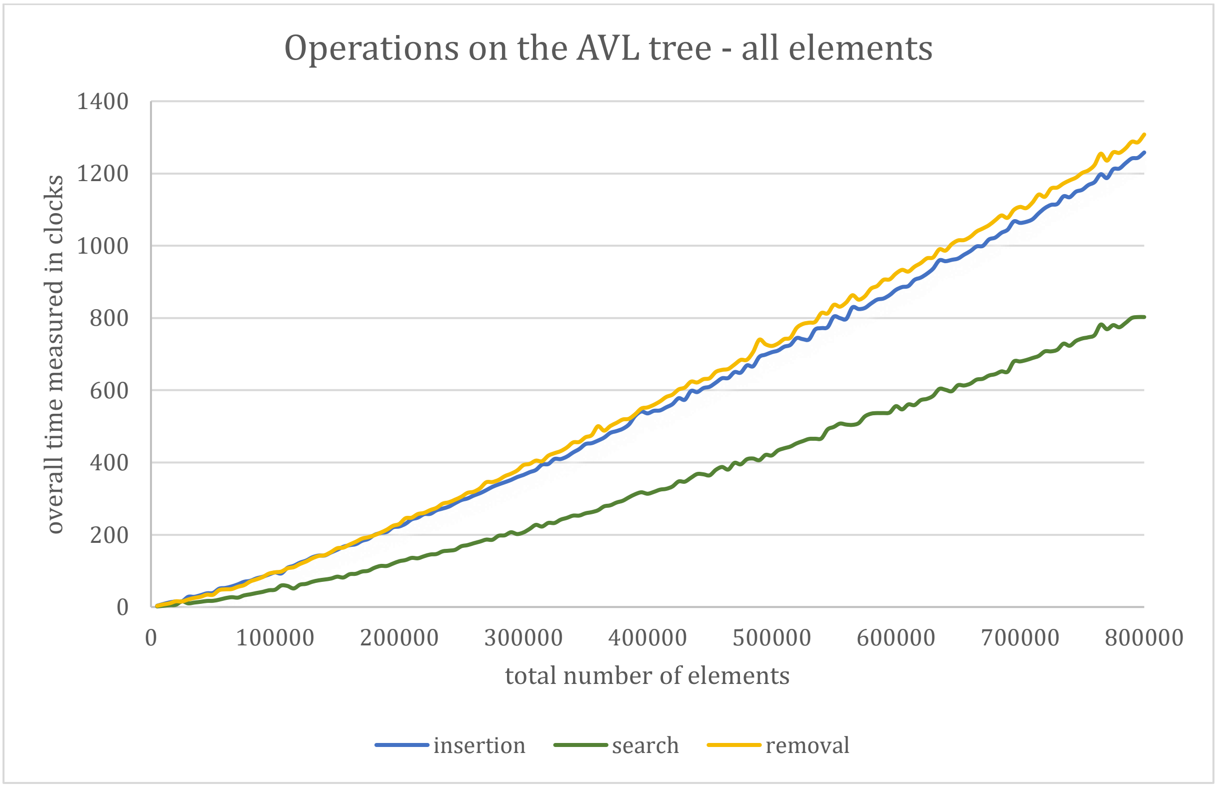
As for the removal algorithm it does not deviate much in its logic from the insertion algorithm also being akin to the usual deletion algorithm of a BST, but contrary to the insertion(AVL) it may well perform multiple rotations, because of the fact that several predecessor-nodes might get imbalanced as a result of a single node deletion, this explains why the removing of nodes starts taking up more time as the tree grows bigger(Chart ...), but since the rotations are (O(1), removing a single node still is of complexity O(logn).

Regarding the search algorithm, it performs in the exact same way as usual in a BST tree, but because of the upper bound on the height, the worst case is not allowed, which renders it O(logn).



line chart 5

On the above chart one can notice that the gap between the searching algorithm and the other two has widened as compared to line chart 2 – this is due to the aforementioned rotations.



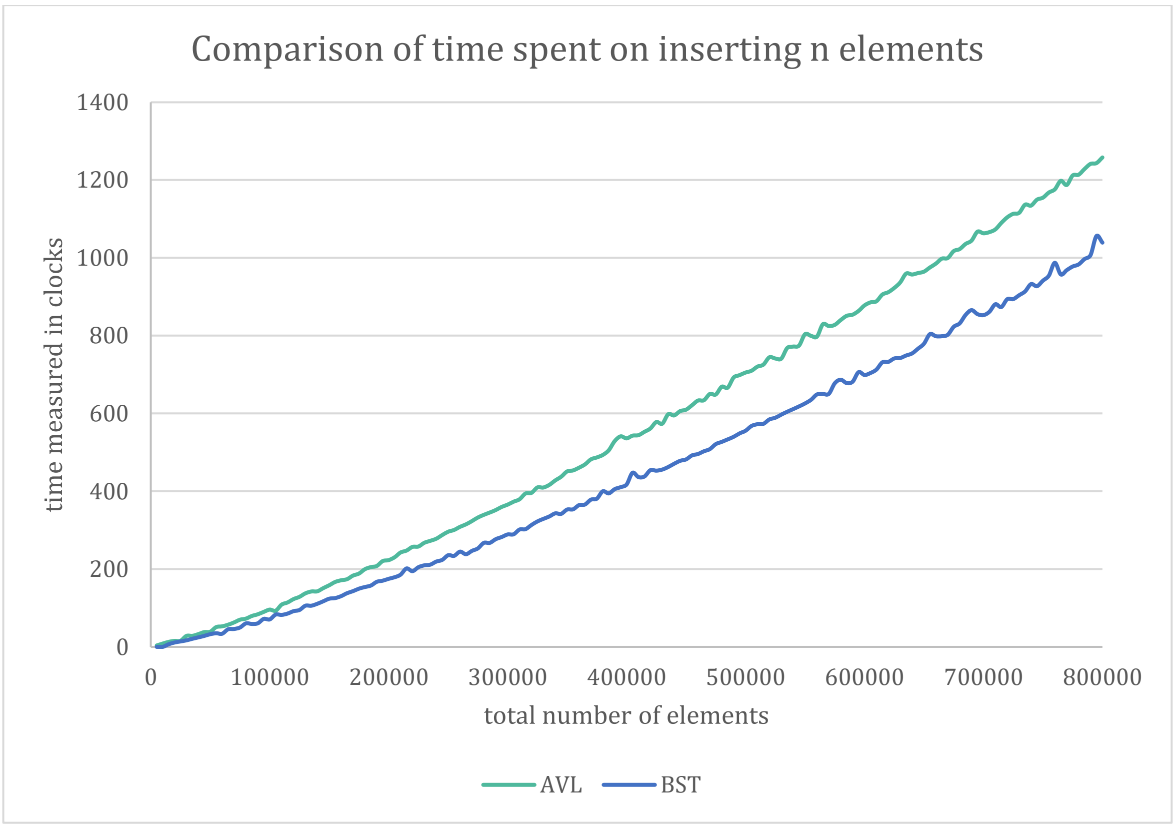
line chart 6

Summary of time complexities:

Single element

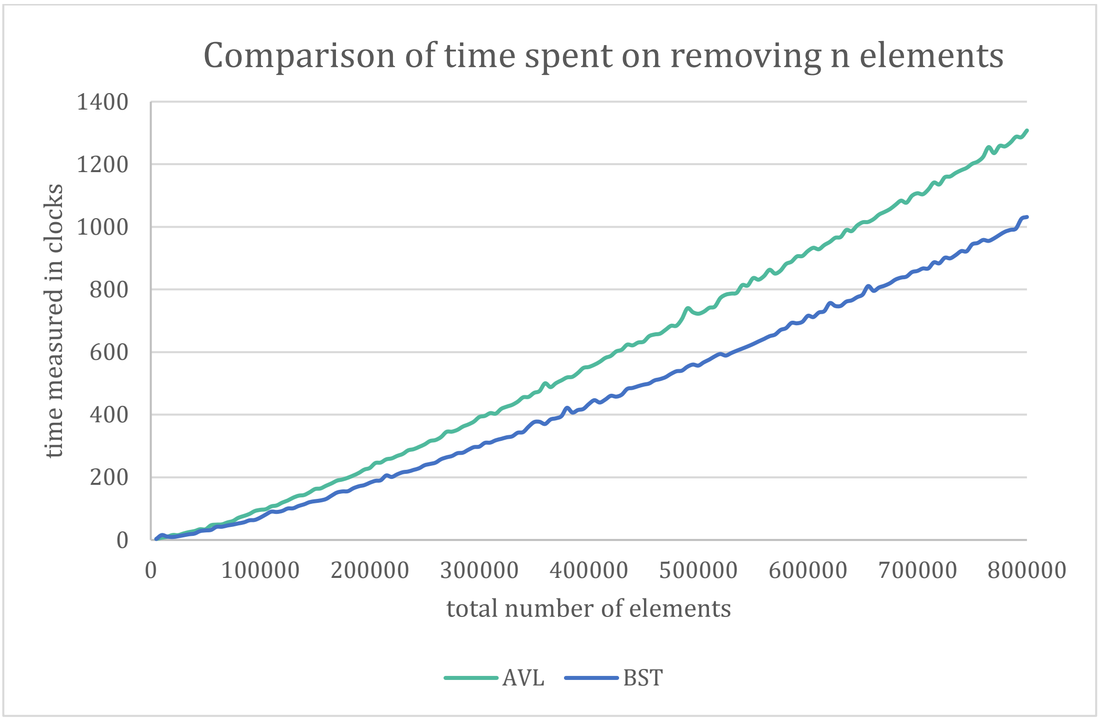
operation\structure	ordered list	BST	AVL
insertion	O(n)	O(h)- (worst case h=n)	O(logn)
search	O(n)	O(h)- (worst case h=n)	O(logn)
removal	O(n)	O(h)- (worst case h=n)	O(logn)

Remark: The AVL tree is also O(h), but h has a specifically defined upper bound



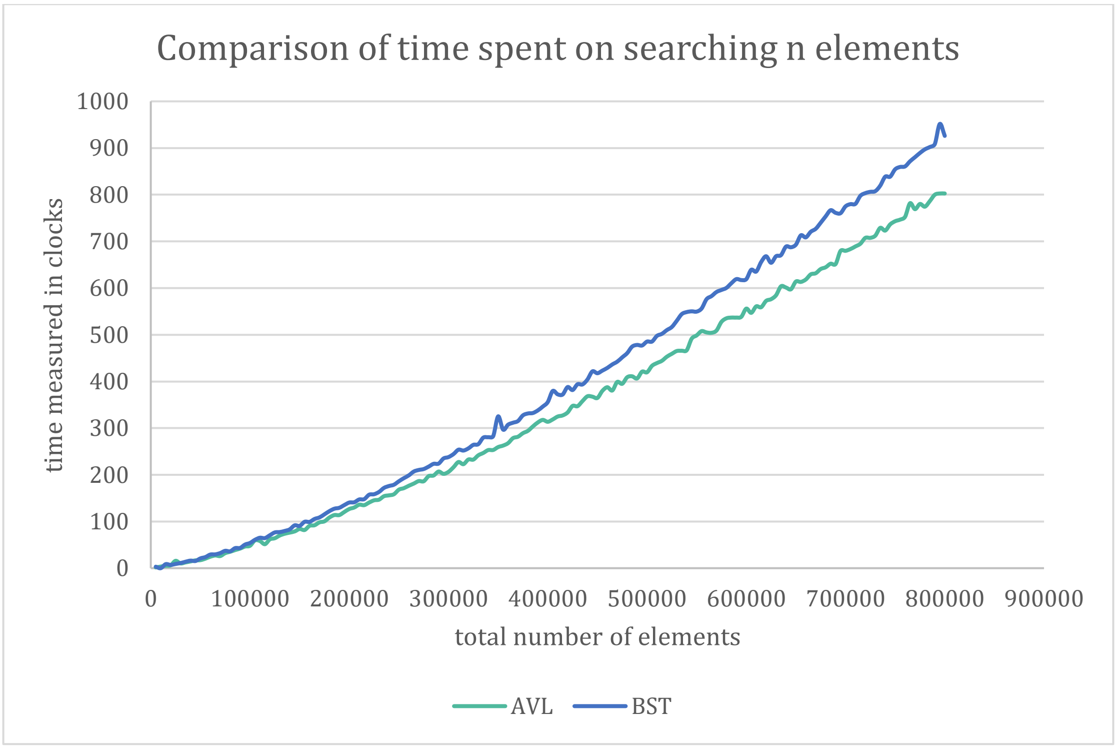
line chart 7

Due to the fact that rotations have o be performed, and the height attribute of nodes has to be constantly updated, the AVLs inserting algorithm takes more time to complete.



line chart 8

The same can be said about the algorithm removing all of the nodes from the AVL tree – the difference is even amplified because of the fact that in large trees several rotations may have to be performed as a result of the deletion of a node.



line chart 9

However, the searching procedure is completed faster as opposed to the BST, because of the upper bound on the AVL Trees height, rendering it favorable if there is great emphasis on searching time, or if sorted data is fed into the tree.

Table of heights:

Number of records	AVL HEIGHT	BST HEIGHT	Maximal number of levels in an AVL tree
5000	15	29	17
10000	16	30	19
15000	16	32	19
20000	17	36	20
25000	18	35	21
30000	18	36	21
35000	18	35	21
40000	19	36	22
45000	18	38	22
50000	19	38	22
55000	19	36	22
60000	19	36	22
65000	19	43	23
70000	19	41	23
75000	20	40	23
80000	20	38	23
85000	19	40	23
90000	20	40	23