Comparison of dynamic data structures

Sebastian Chwilczyński

Karol Cyganik

Theoretical analysis of the structures – Linked List (LL), Binary Search Tree (BST) and Balanced Binary Search Tree (BBST).

In our experiment, we analyzed three data structures - Linked List (LL), Binary Search Tree (BST), and Balanced Binary Search Tree (BBST). They are all pretty similar, but they differ noticeably in a few things.

A **Linked List** is a data structure that just includes a series of connected nodes. Each node stores its data and a pointer to the next node. Every LL starts with a head (starting point of the structure) and ends with a tail (NULL pointer), that gives us information that the list is finished and there are no more data in it. As it stores node after node, this data structure is not very efficient. To insert a node, we either need to go to the end of the list or if the list is ordered to the correct place to preserve order. So in average it takes linear time (*O(n)*) to add a new node (where n is the length of the list). Similarly, to search an element we would need also O(n) time in a worst-case, as we must traverse throughout the list until we find element that we look for, so in the worst-case from the beginning to the end.

A **Binary Search Tree** is a data structure that keeps our data obeying some rules. Namely left sub-tree of considered node contains only elements less than the node's key and right sub-tree elements greater than the key of the node. Furthermore sub trees also must be BST. It is most common to use integers as a keys. Each node has at most 2 children (0 and 1 are also possible), which makes it a binary tree. It's called the Search tree because it allows us to find elements very quickly - in an O(log n) time (in most cases)!

As it's written above, the best and average time complexity of searching is O(log n). That's because when we go left or right from a node, we reject the opposite subtree, so in the best case, we reduce the tree for searching by half. The other operation to measure is an insertion, which also runs in O(log n) time in best and average cases. Why? We can think about insertion as searching for the only suitable place for a new node, so that's why the complexity of searching and inserting is the same. Similarly, with deletion, we need to search the node to delete and then perform another searching for element that can replace deleted node so that the tree still fulfill the BST principles (It can be either minimum of the left sub-tree or maximum of the right-sub-tree).

What should be pointed is that for Linked List to implement less memory is needed than in case of tree structure, as we store only one pointer in each node not two.

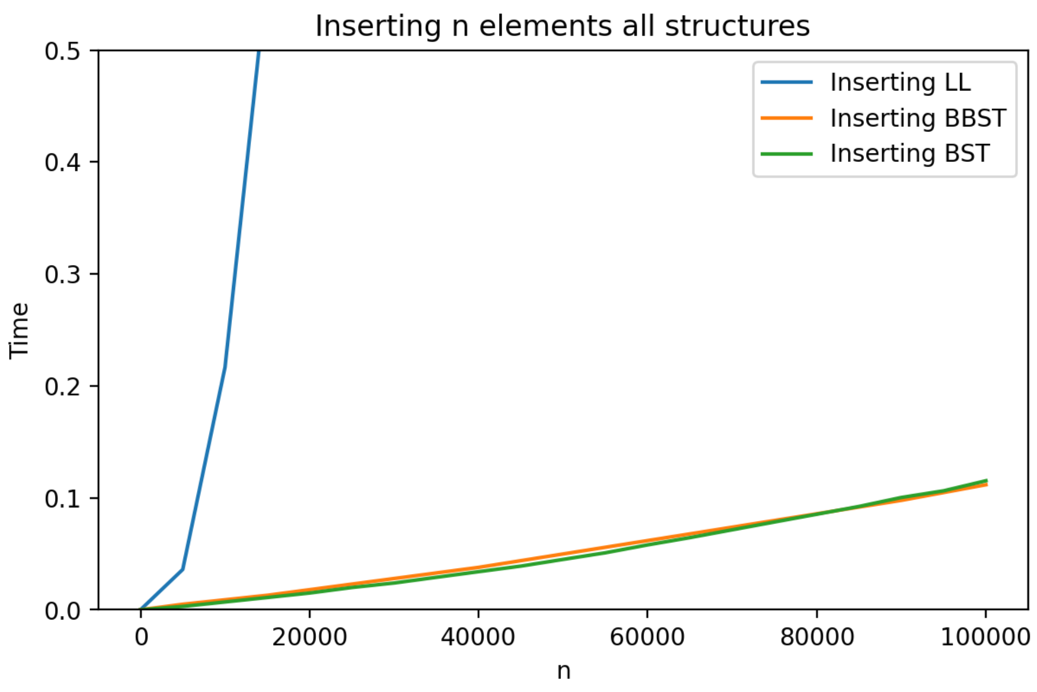
So what are the worst-case complexities? We can think about the worst case, as inserting a sorted list into BST element by element. Then, every node will contain only the right child, so in general it will be just a Linked List with additional left pointer that is always NULL, not really a BST. That's why the complexity of all three operations will be the same as in LL - O(n).

Basically a **Balanced Binary Search Tree** is just a BST but with one more rule to be followed. The absolute value of the difference between heights (balance factor) of left and right sub tress cannot be greater than some positive constant and when this constant is 1 the BBST is called **AVL** tree. In case of implementation it is very similar to previously concerned BST but after insertion and deletion we need to constantly verify whether the balance factor doesn't exceed allowed value, if it does we must rebalance the tree. We do this in terms of tree rotations which in practice just means changing some pointers. This is done in constant time *O(1)* and does not influence overall complexities. Thanks to that we never hit the worst case like it was in BST so then best average and worst case of all operations is *O(log n).* In our experiment we used implementation when difference in height can be at most 1.

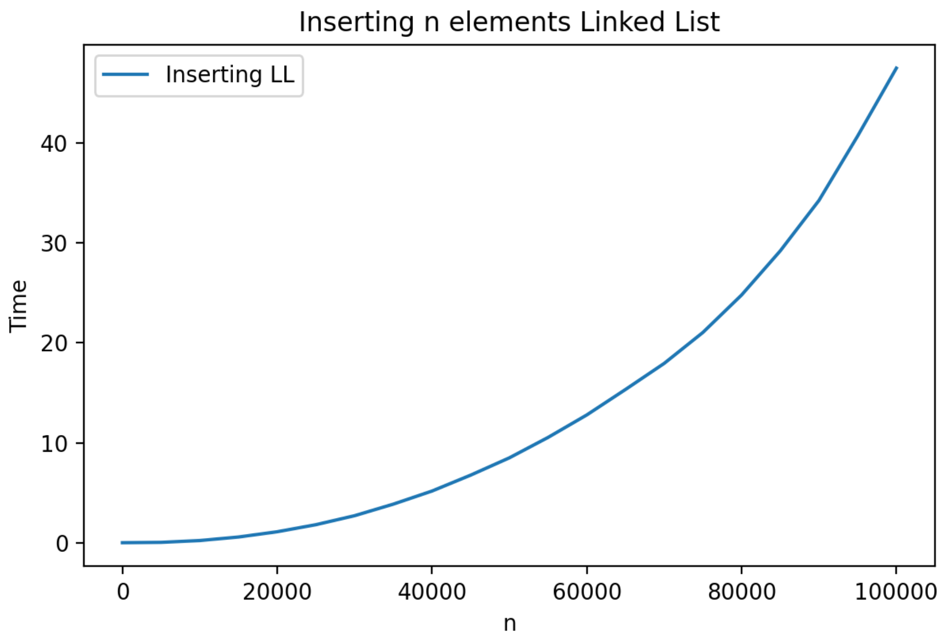
Experiments - measuring times of adding, searching and deleting elements in LL, BST, BBST

In first part of every experiment at first we measured time needed to add n elements to the list so complexities should be as follows: *$O(n^2)$* for linked list, as we add n elements all in linear time, *O(nlogn)* for BST and balanced version of it, since we add n elements all in logarithmic time. Of course in case of BST and the worst case of it we expected to obtain *$O(n^2)$* and then in second part we divided outcomes by n to get time needed only to perform individual operations

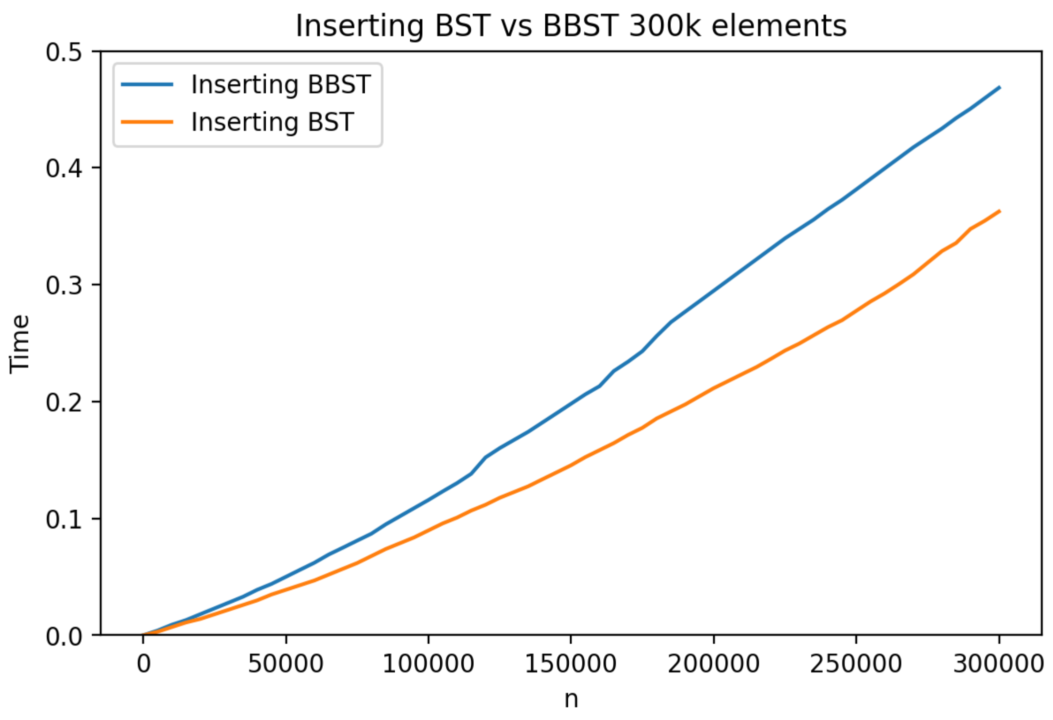
Insertion



Here outcomes are in line with our expectations. We can easily observe that plot of the linked list looks like a parabola and as a whole it represents like this:

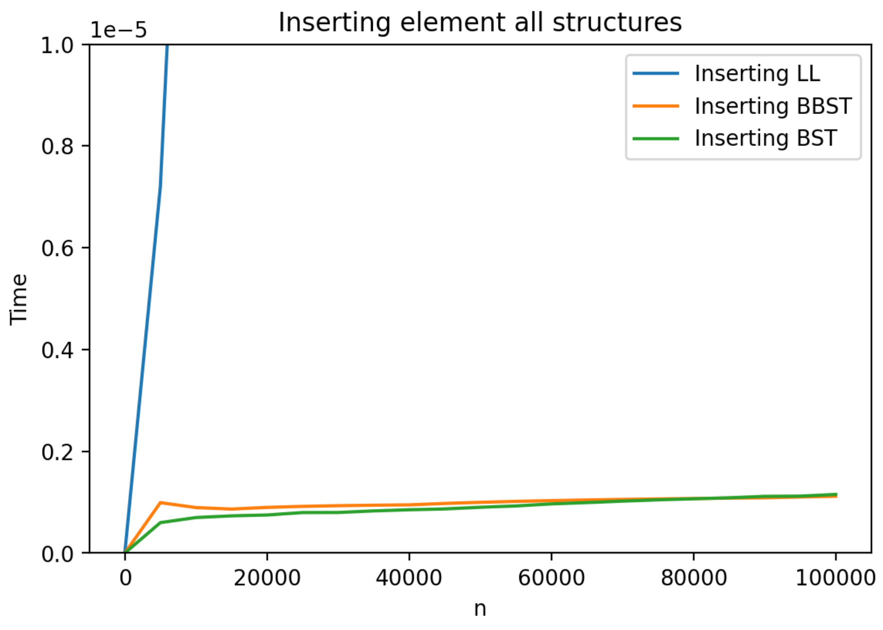


For both trees complexities are *O(nlogn)*. What is interesting here is that we expected that BBST will be faster than unbalanced variant, but actually they go shoulder to shoulder. For us the reason is that we constantly check if the tree is balanced and that takes extra time in every insert operation. But taking into account that we measured only 100k element we created a comparison for 300k:

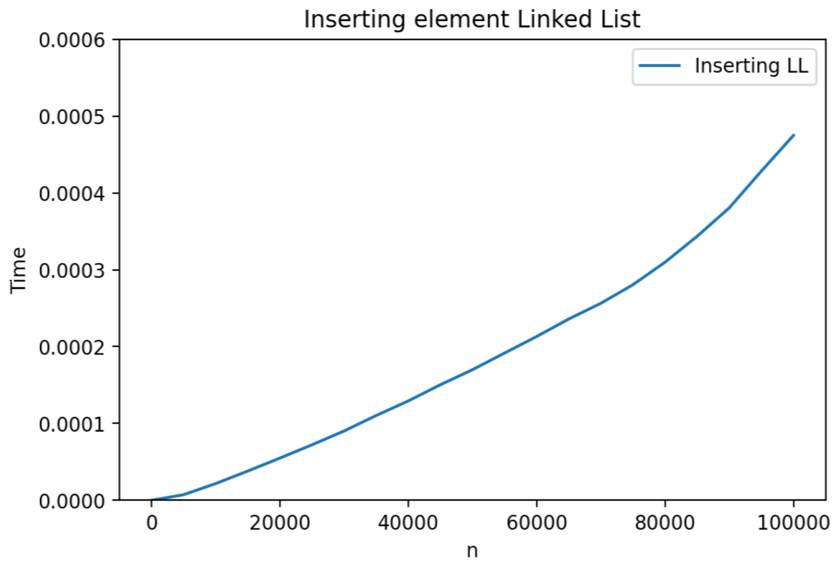


Unexpectedly the gap has got even bigger when we increased amount of inserted elements. The only difference in implementation is that we added 4 additional if statements after successful insert to check whether the tree is balanced and then if not, did appropriate rotation. Even though if statements and rotations take in theory constant time to execute and does not influence on overall characteristic of the complexity function, it is enough to deteriorate efficiency that it works a little bit slower than its unbalanced version.

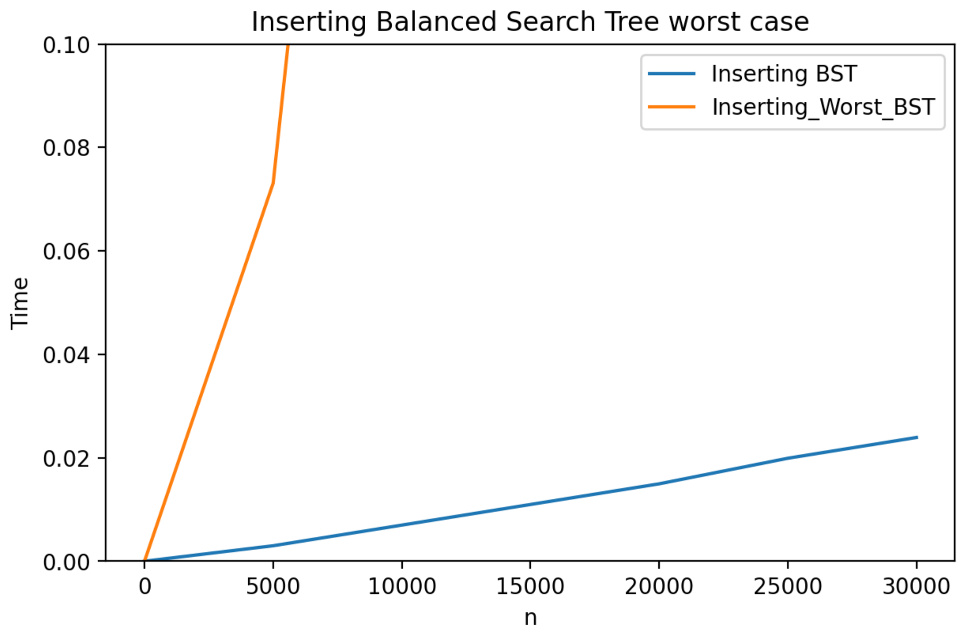
Then we measured an averaged time needed to add one element, when in the structure were n elements:



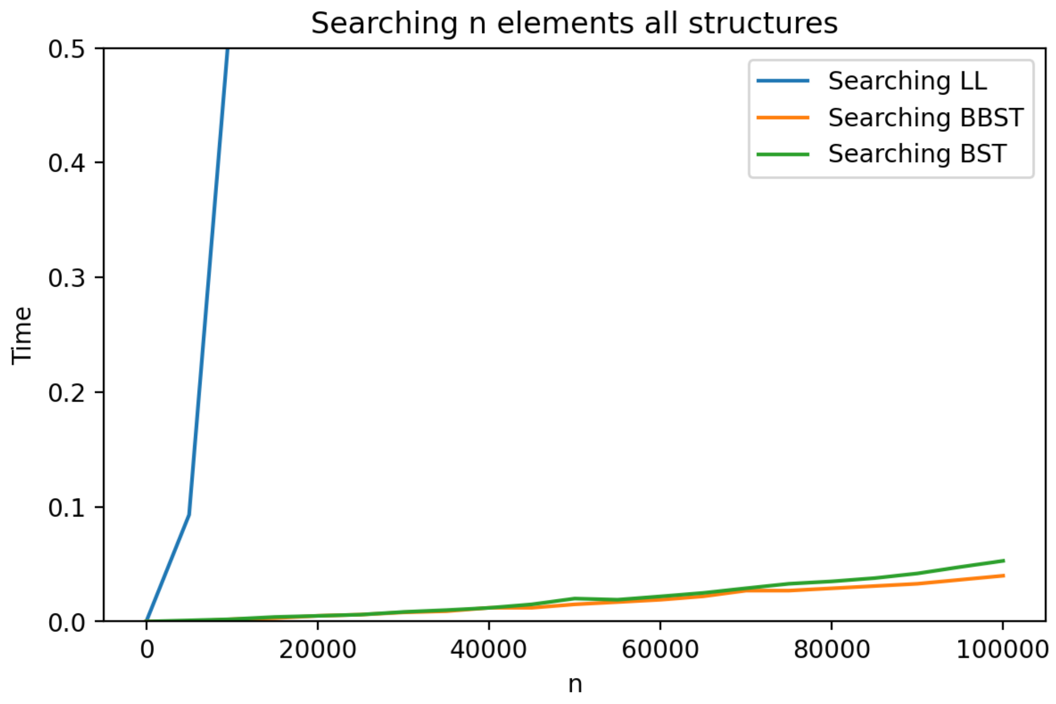
As we can see, the time grows as a logarithmic function in BST and BBST, so the function is similar to *O(log n)*. Linked list is much slower, but the insertion time for one element is equal to *O(n)*, which is a lot worse than *O(log n)*. The interesting thing is that BBST was slightly worse than BST at the beginning, but later they were both comparable. That peak around 5 thousand elements in the AVL tree characteristic may be caused by some measurements error, because it seems quite strange that adding 5 thousand elements take more time than adding 100 thousand.



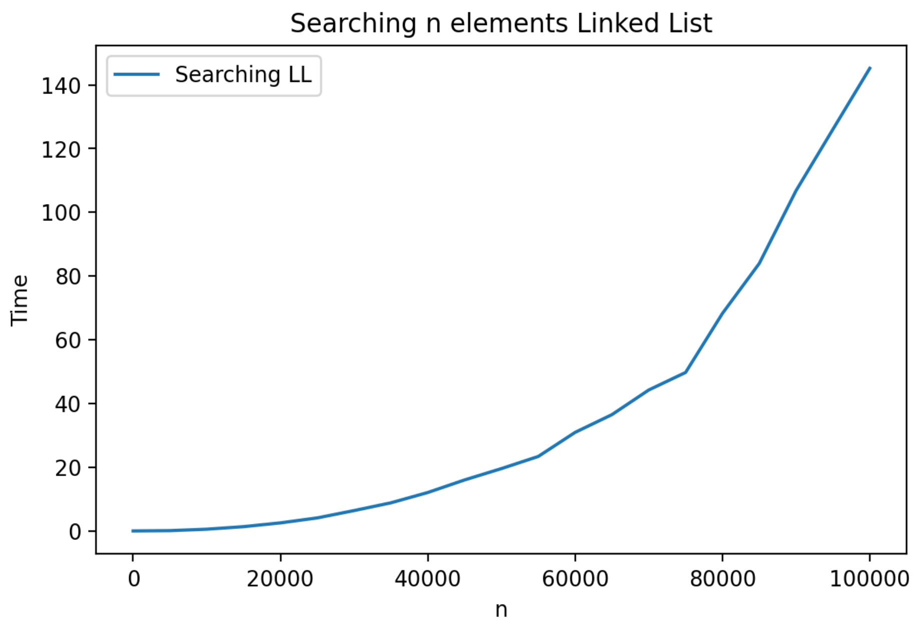
Interesting thing happens, when we add elements from a sorted list to the BST. Then, every node has only right child, what makes it a Linked List with additional memory consumption since every node has a pointer to the left child that is always NULL. The plot below shows the complexity is very similar to the Linked List and deterioration in comparison of random distribution of indexes is substantial. We measure time for only 300k elements because when we try to expand the amount we encounter the stack overflow error due to going recursively too deep.



Searching

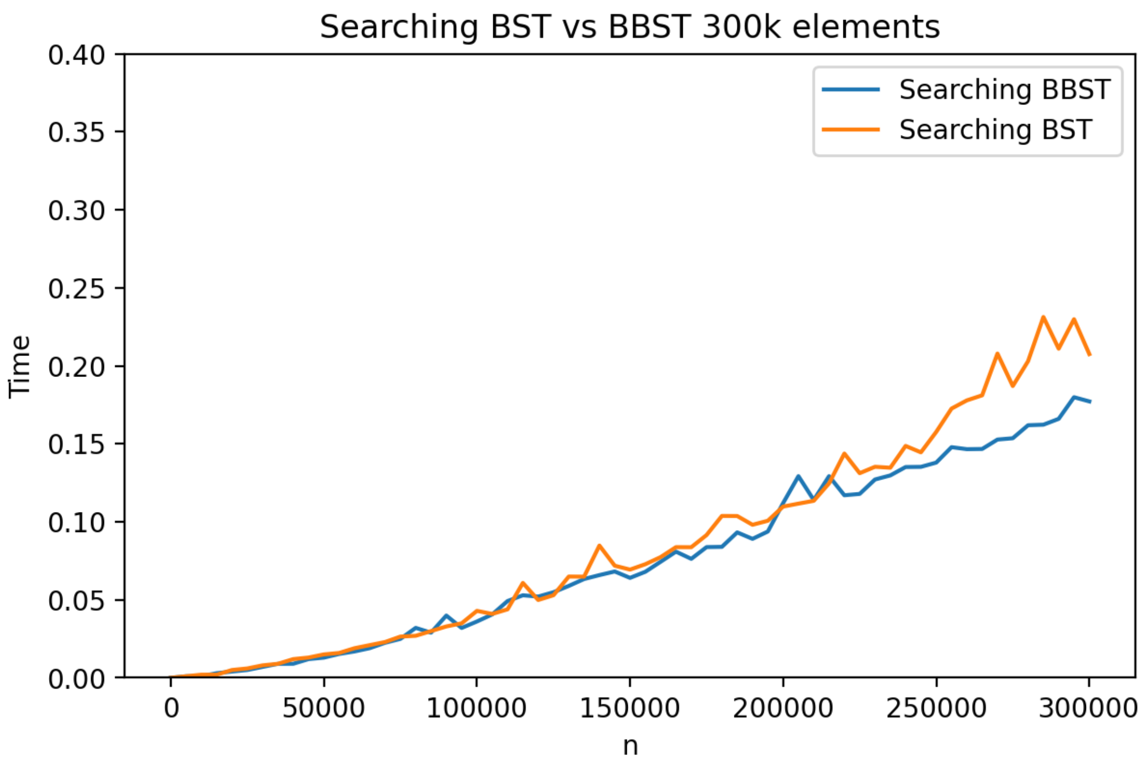


Here results in terms of shape of the plot are almost identical. We can observe that here AVL tree is doing slightly better job when we increase the amount of elements. For Ordered List time needed to search n elements is much worse than time needed to insert elements



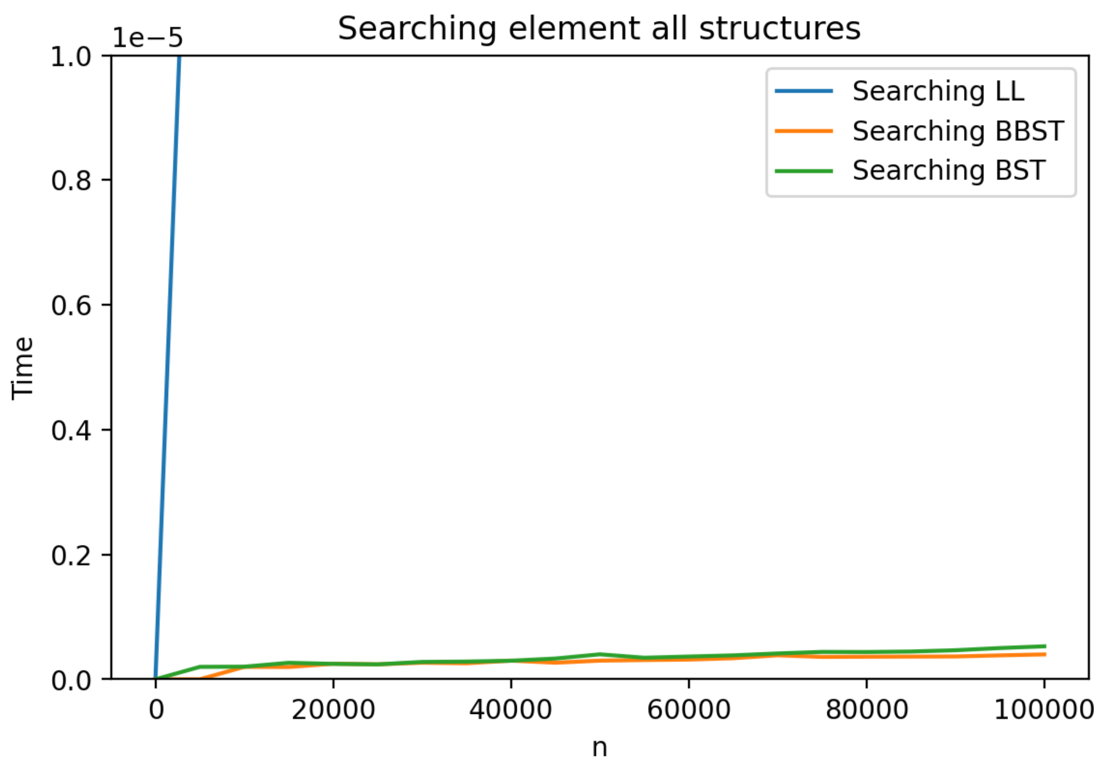
In comparison to searching it seems like big deterioration but have in mind that when we search we already store n elements and when we insert we start from 0 and only at the end store n.

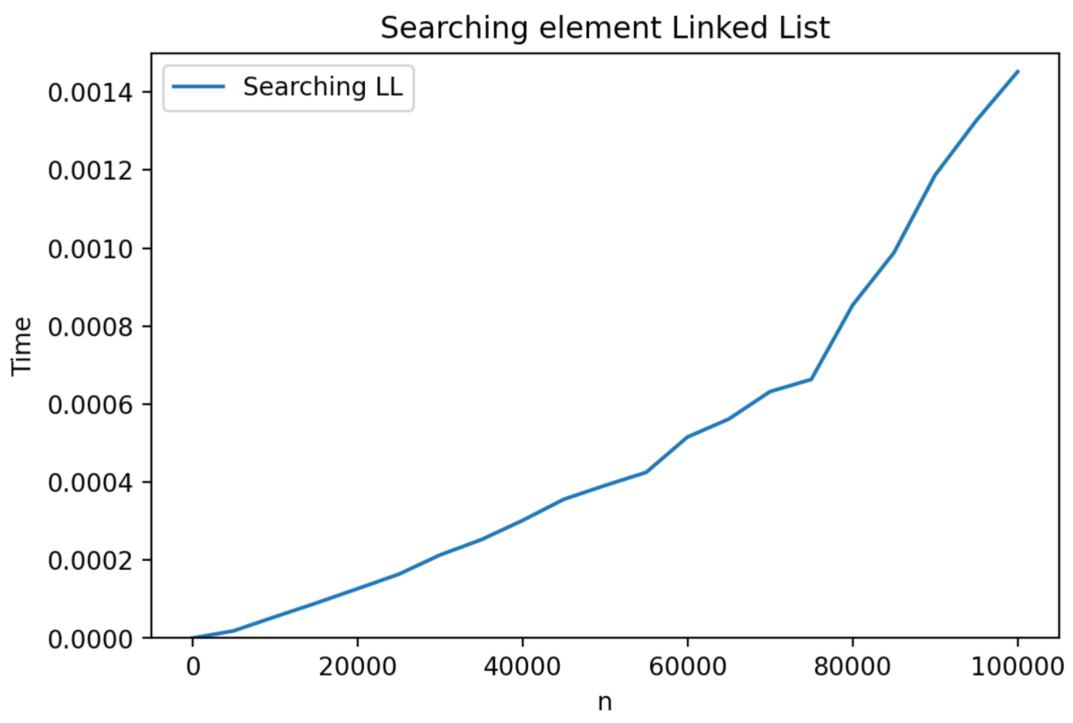
Now let's take a closer look on our tree structures:



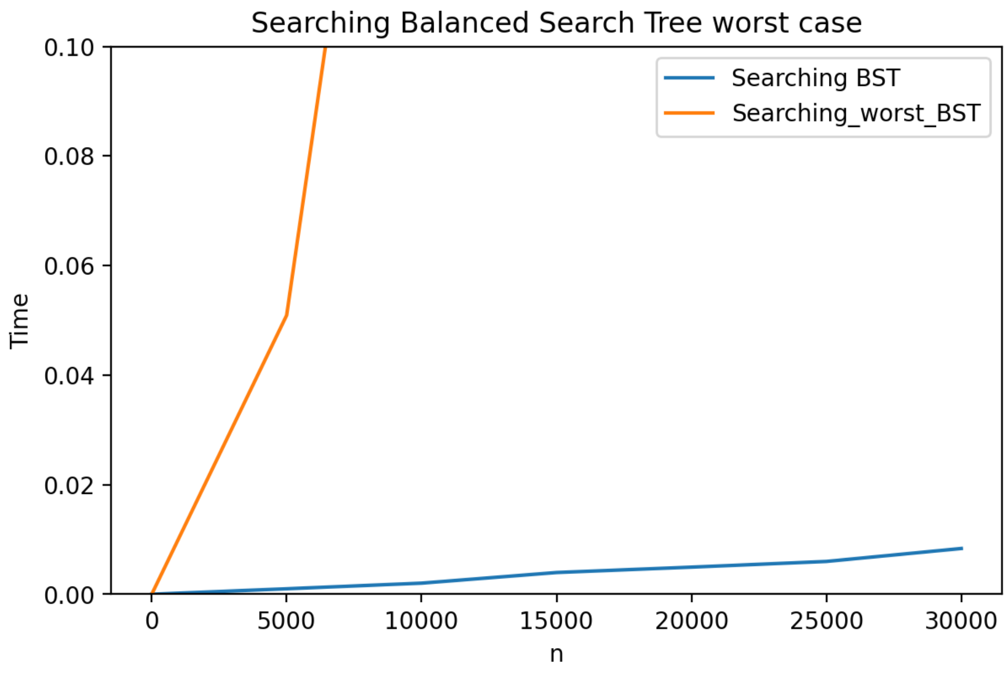
For 300k elements we can now be 100% sure the winner is Balanced version of BST. We think that searching is most common and most important operation performed on data structures in case of efficiency, so a little bit more complex implementation and longer time to insert elements proved worthwhile.

Then we again divided the total searching time by n to obtain an average time needed for searching for only one element in a structure which has n elements.



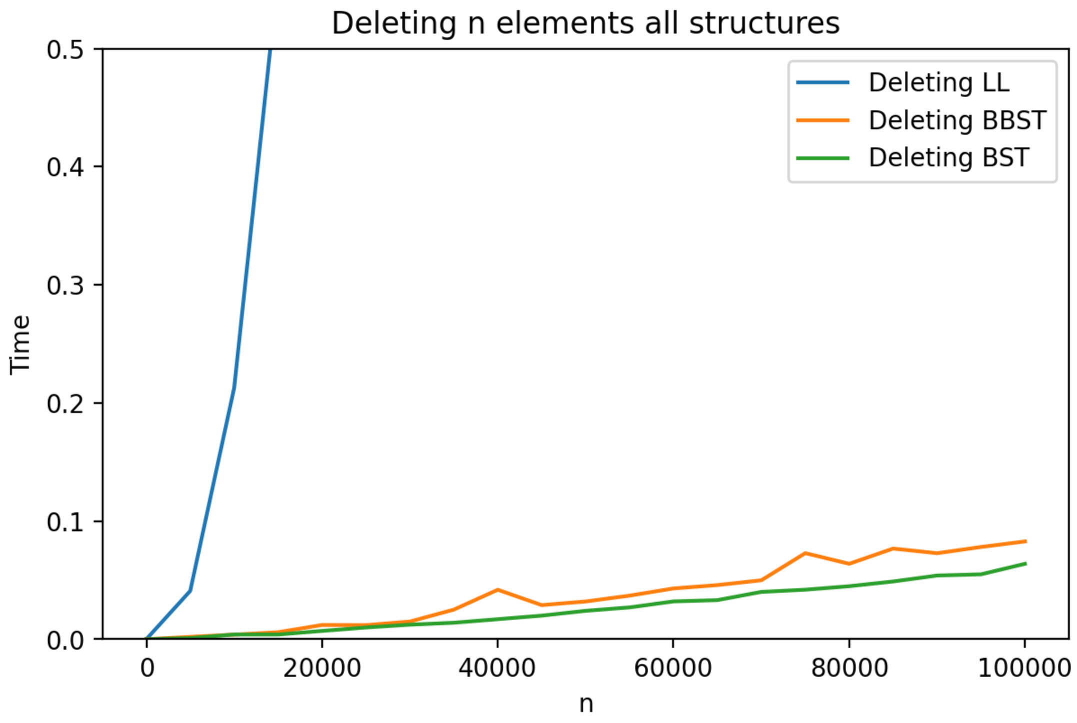


In BST worst case for searching occurs when we add elements from a sorted list and we obtain just a Linked List with additional useless pointers. As the plot shows, noticeably more time is needed to search all elements in the list.

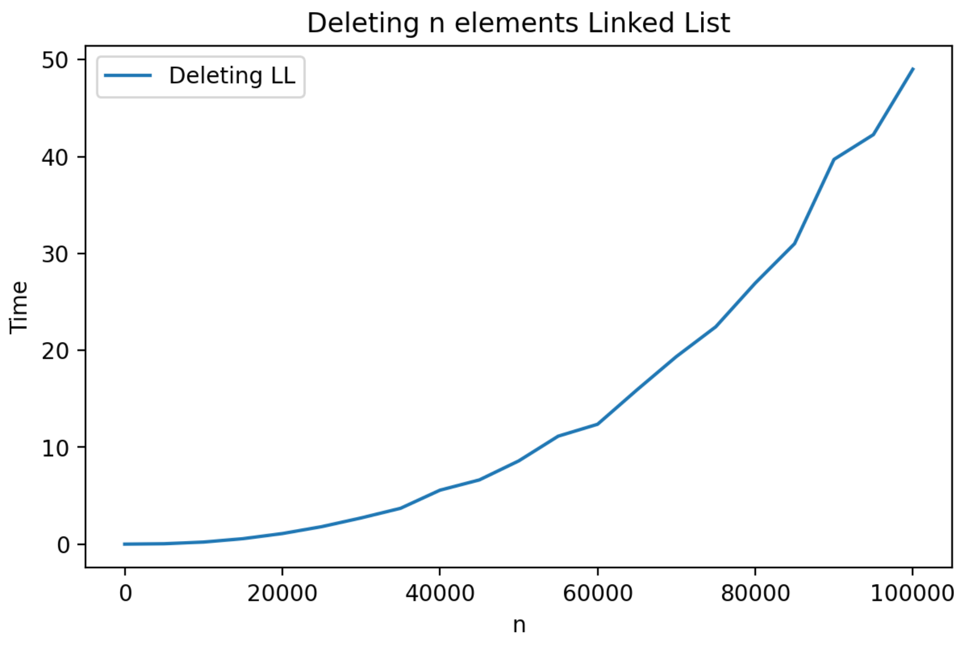


Deletion

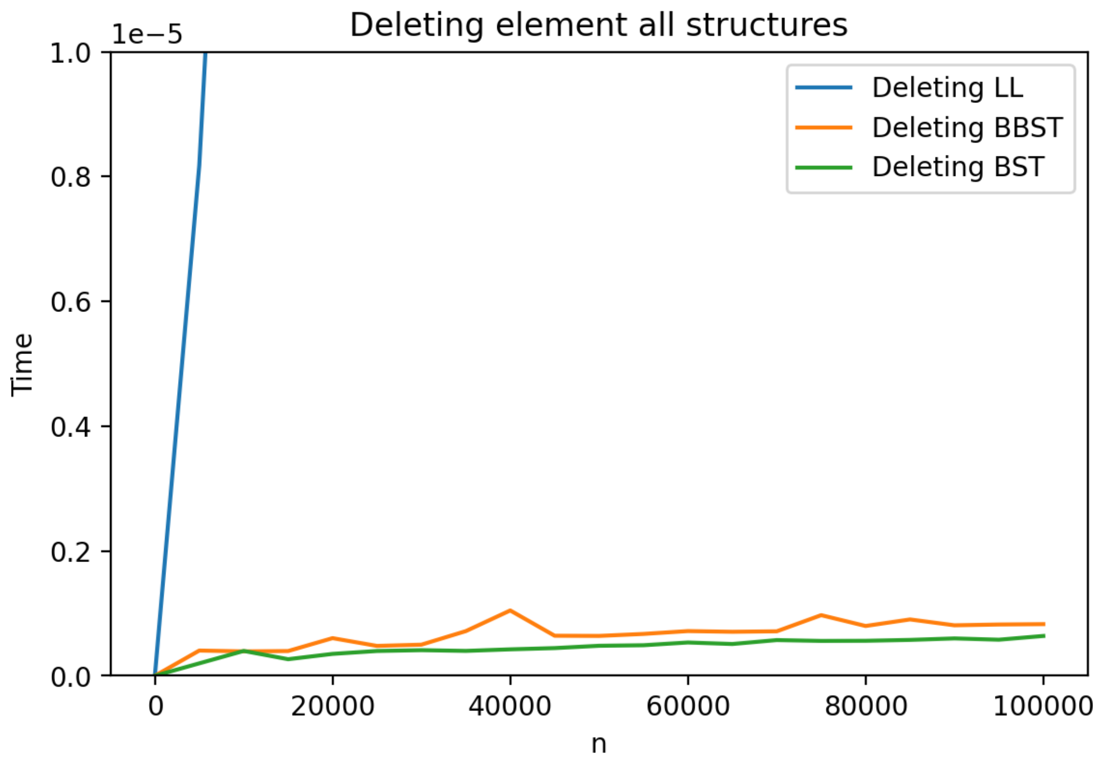
We measured time needed to delete all elements by at first creating structure of size n and then deleting each member of the structure.

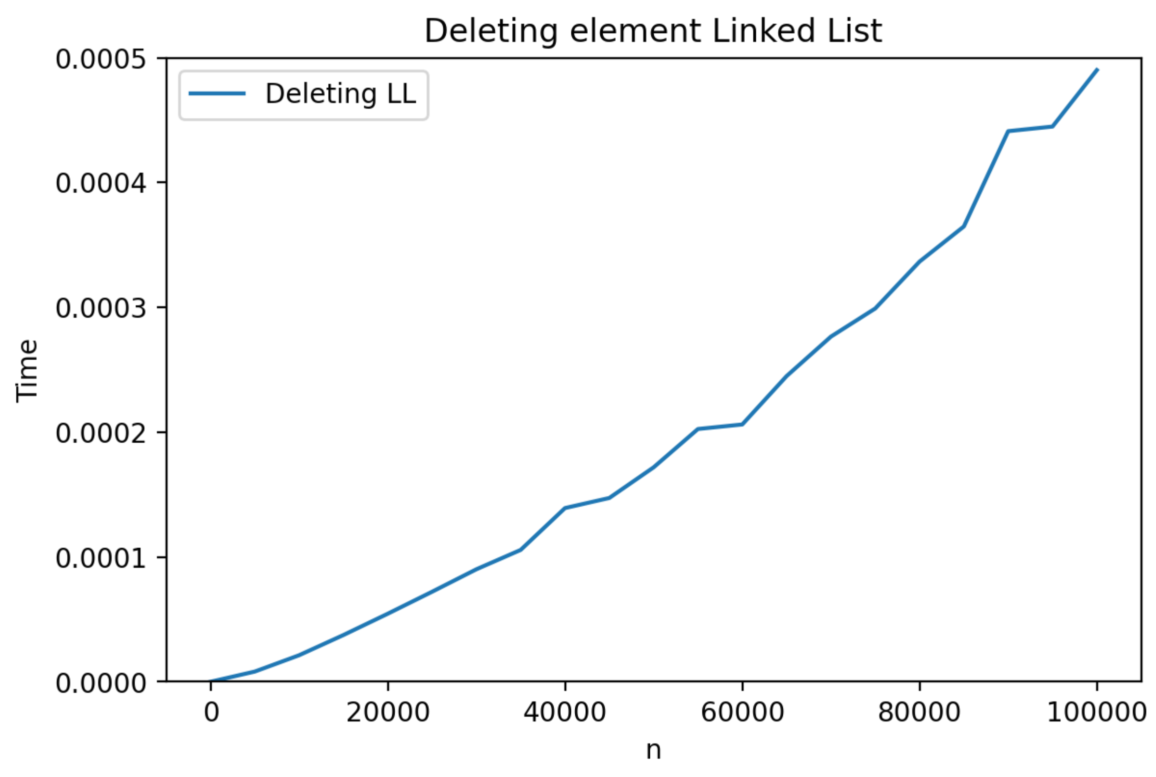


Again nothing that would shock us. What may be interesting is that BBST loses in terms of deleting, but taking into account balance factor every time costs.

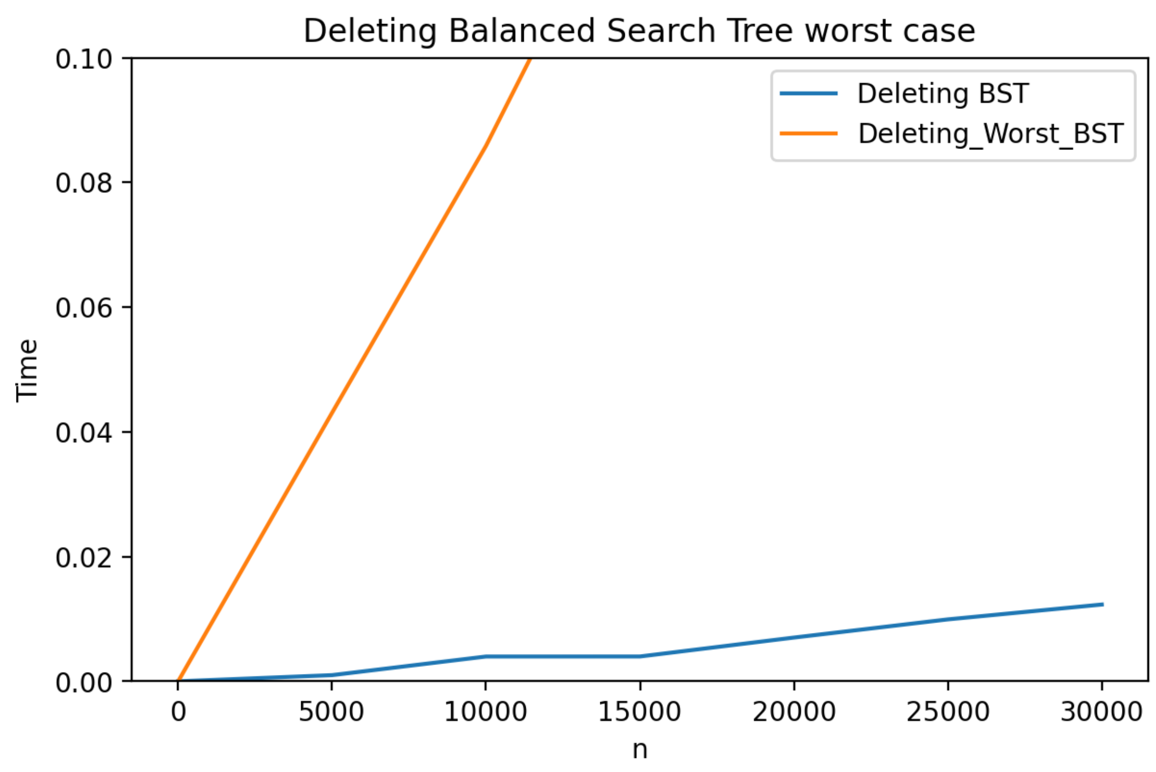


Below plot shows the time needed to delete one element from a structure which has n elements. Similarly to insertion and searching, we have linear time for LL and logarithmic times for BST and BBST.

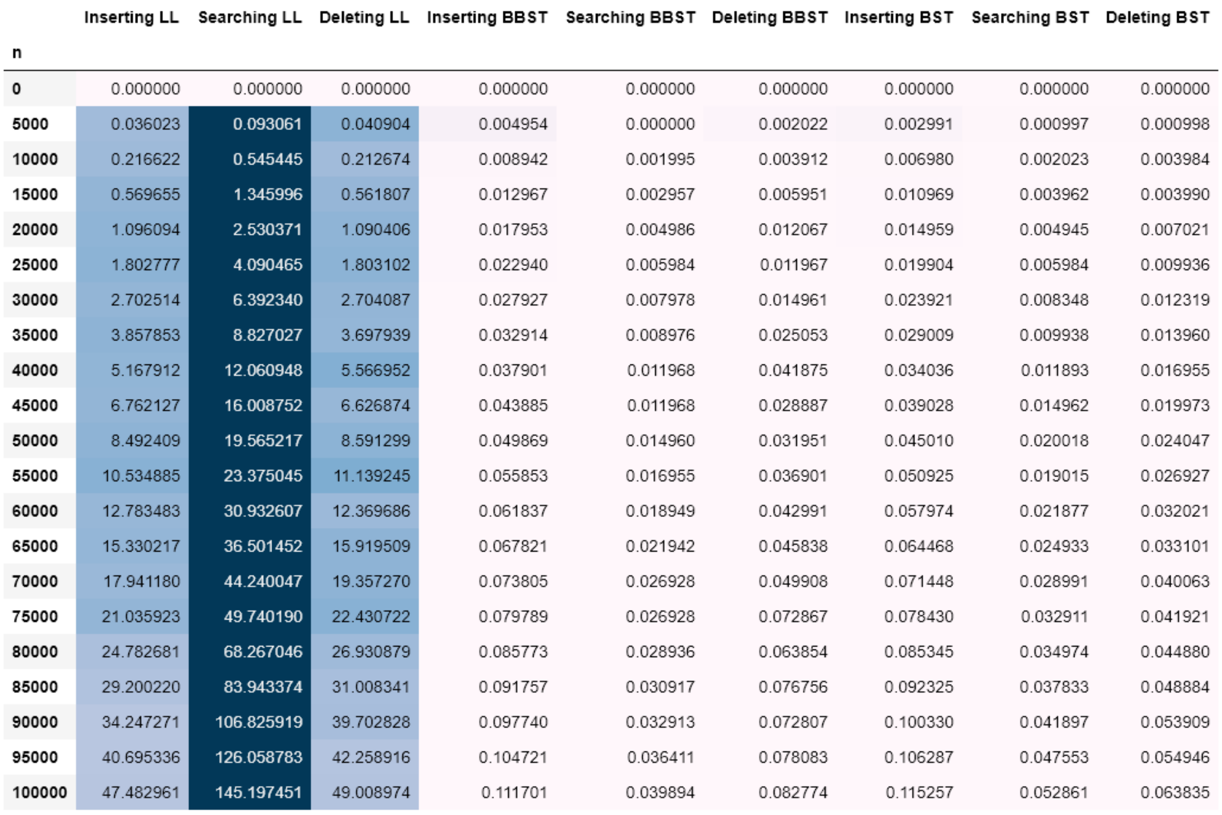




Here, the worst case in BST occurs in the same situation as in insertion and searching. Again, it makes deletion time linear from logarithmic.

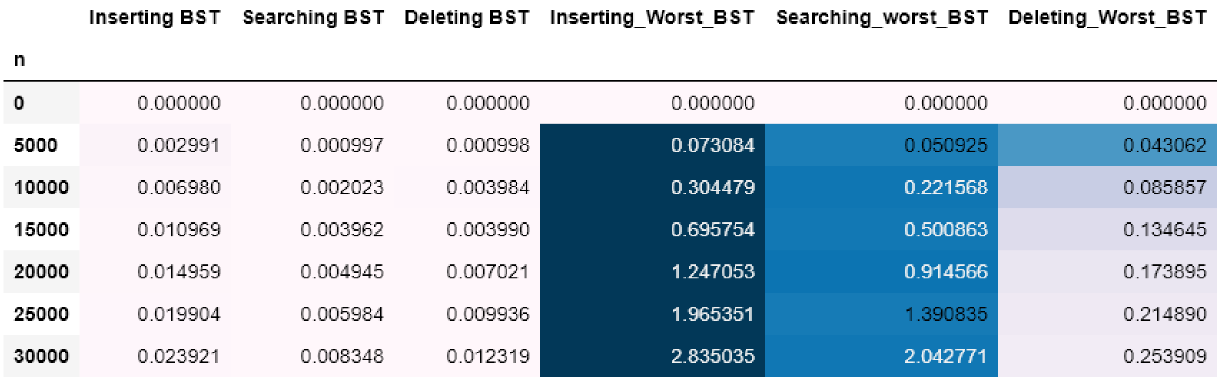


Below we have a table, which shows all the times needed for insertion, searching and deletion in Linked List, Binary Search Tree and Balanced Binary Search Tree. As we can see, LL is definitely the slowest dynamic data structure, the times there are much bigger than in trees.



n	Inserting LL	Searching LL	Deleting LL	Inserting BBST	Searching BBST	Deleting BBST	Inserting BST	Searching BST	Deleting BST
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5000	0.036023	0.093061	0.040904	0.004954	0.000000	0.002022	0.002991	0.000997	0.000998
10000	0.216622	0.545445	0.212674	0.008942	0.001995	0.003912	0.006980	0.002023	0.003984
15000	0.569655	1.345996	0.561807	0.012967	0.002957	0.005951	0.010969	0.003962	0.003990
20000	1.096094	2.530371	1.090406	0.017953	0.004986	0.012067	0.014959	0.004945	0.007021
25000	1.802777	4.090465	1.803102	0.022940	0.005984	0.011967	0.019904	0.005984	0.009936
30000	2.702514	6.392340	2.704087	0.027927	0.007978	0.014961	0.023921	0.008348	0.012319
35000	3.857853	8.827027	3.697939	0.032914	0.008976	0.025053	0.029009	0.009938	0.013960
40000	5.167912	12.060948	5.566952	0.037901	0.011968	0.041875	0.034036	0.011893	0.016955
45000	6.762127	16.008752	6.626874	0.043885	0.011968	0.028887	0.039028	0.014962	0.019973
50000	8.492409	19.565217	8.591299	0.049869	0.014960	0.031951	0.045010	0.020018	0.024047
55000	10.534885	23.375045	11.139245	0.055853	0.016955	0.036901	0.050925	0.019015	0.026927
60000	12.783483	30.932607	12.369686	0.061837	0.018949	0.042991	0.057974	0.021877	0.032021
65000	15.330217	36.501452	15.919509	0.067821	0.021942	0.045838	0.064468	0.024933	0.033101
70000	17.941180	44.240047	19.357270	0.073805	0.026928	0.049908	0.071448	0.028991	0.040063
75000	21.035923	49.740190	22.430722	0.079789	0.026928	0.072867	0.078430	0.032911	0.041921
80000	24.782681	68.267046	26.930879	0.085773	0.028936	0.063854	0.085345	0.034974	0.044880
85000	29.200220	83.943374	31.008341	0.091757	0.030917	0.076756	0.092325	0.037833	0.048884
90000	34.247271	106.825919	39.702828	0.097740	0.032913	0.072807	0.100330	0.041897	0.053909
95000	40.695336	126.058783	42.258916	0.104721	0.036411	0.078083	0.106287	0.047553	0.054946
100000	47.482961	145.197451	49.008974	0.111701	0.039894	0.082774	0.115257	0.052861	0.063835

The table above shows the times in average cases. As we know, BST has the worst case what makes it very similar to Linked List.



n	Inserting BST	Searching BST	Deleting BST	Inserting_Worst_BST	Searching_worst_BST	Deleting_Worst_BST
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5000	0.002991	0.000997	0.000998	0.073084	0.050925	0.043062
10000	0.006980	0.002023	0.003984	0.304479	0.221568	0.085857
15000	0.010969	0.003962	0.003990	0.695754	0.500863	0.134645
20000	0.014959	0.004945	0.007021	1.247053	0.914566	0.173895
25000	0.019904	0.005984	0.009936	1.965351	1.390835	0.214890
30000	0.023921	0.008348	0.012319	2.835035	2.042771	0.253909

Comparison BST and BBST in terms of recursive depth

Both BST and BBST are implemented in recursive way. We tried to measure how deep recursively function goes, using simple counter that increase by 1 each time insert function is called. Then we divided that counter by number of added elements and we obtained such results. What is also important that we make measuring point not to be multiples of 10 but multiples of 2 to check if number of calls increase as we double the number of elements. Results were as we expected. BST looks like logarithm function but actually on average it goes few calls deeper than BBST which is pure logarithm of base 2. We can see, that the more elements we have, the greater the difference is. This is because of the balanced structure of BBST.

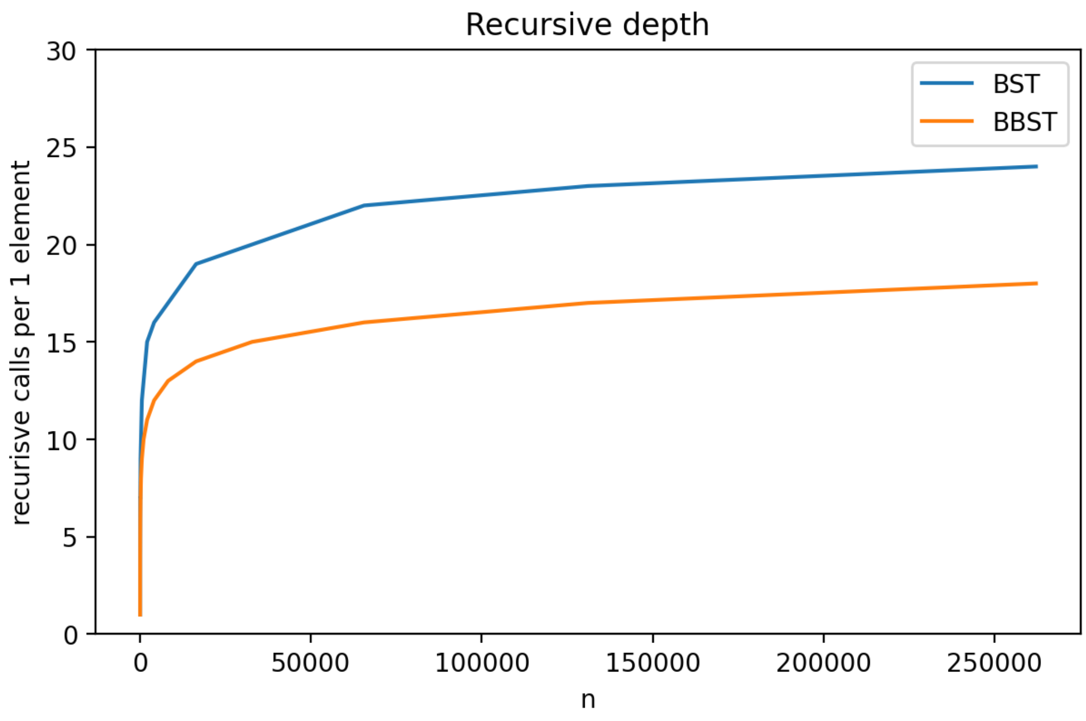
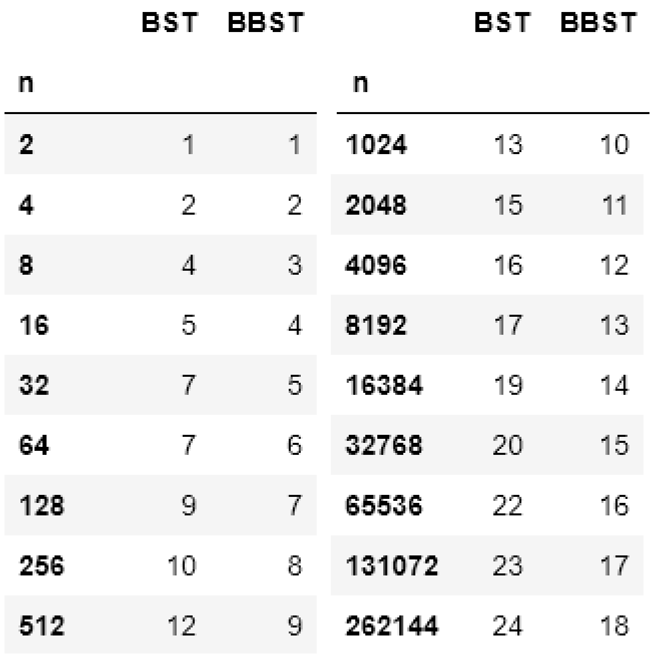


Table of recursive depth



n	BST	BBST
2	1	1
4	2	2
8	4	3
16	5	4
32	7	5
64	7	6
128	9	7
256	10	8
512	12	9
1024	13	10
2048	15	11
4096	16	12
8192	17	13
16384	19	14
32768	20	15
65536	22	16
131072	23	17
262144	24	18

I have to admit that seeing perfect logarithm function in AVL tree is quite satisfying.

Conclusions and remarks

Considering three data structures – Linked List, Binary Search Tree and Balanced Binary Search Tree, each of these has its advantages and disadvantages. Linked List is the simplest in implementation, the memory consumption is the smallest, but it is also the slowest one. Comparably the BST is little harder in implementation, but it gives much better times. The only thing in BST is, that it has a worst case turning BST into a Linked List. Although the worst case when we have random distribution of the data is rather very unlikely, the real life data is sorted or ordered in some way very often. To get rid of the worst case in BST we may try to create it recursively from ordered array always selecting the middle element as a next node and then calling function recursively for the left sub tree giving left part of sorted array and for the right sub tree right part of the sorted array. Then without checking balance factor we are sure that our tree will be balanced. But still after inserting some nodes our tree will fast became unbalanced. All in all it’s not always the best option to choose. Last structure, BBST is just a modification of a BST which solves the problem encountered in the worst case of BST. The tree is always balanced, so it means it needs slightly more operations than BST to add elements and delete them to always make sure balance factor doesn’t exceed allowed value. However searching is much faster. The thing is, that it’s a little bit more complicated to implement, as rotating the tree sometimes is quite tricky.