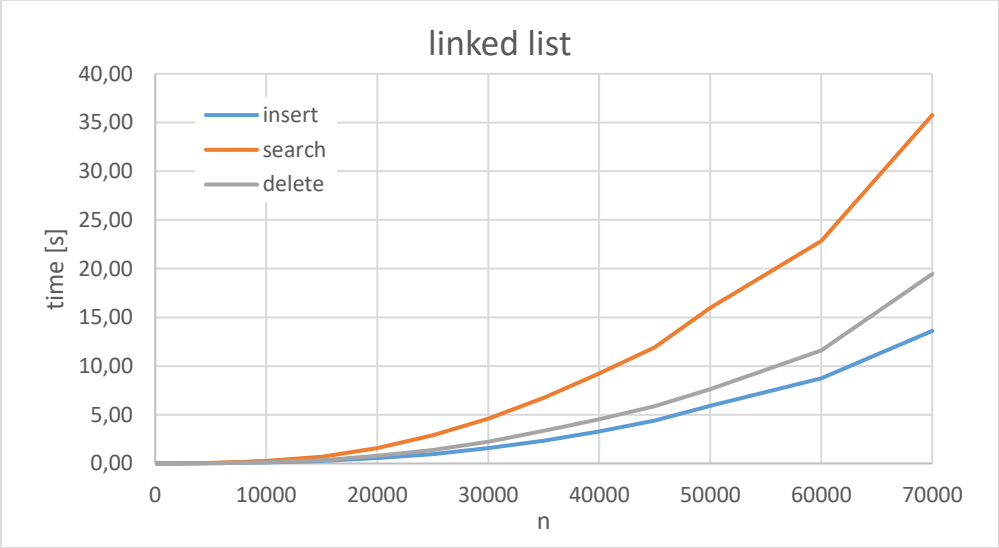
Zuzanna Gawrysiak ID: 148255
Agata Żywot ID: 148258

Report 2 Dynamic Data Structures

Introduction

In the experiment time measurements were taken on arrays of the size n, starting with 100 elements 100 up to 70 000. The structures itself consist of students' indices, while the complete data (with names and surnames of students) is stored in an array of structures. Every outcome concerning linked-list is averaged over 10 runs, while the results for trees (BST and BBST) are based on 100 repetitions. Times of operations are presented in seconds. The order of elements is randomized for every trial and each time probe concerns performing operations for n elements.

1. One directional ordered linked-list



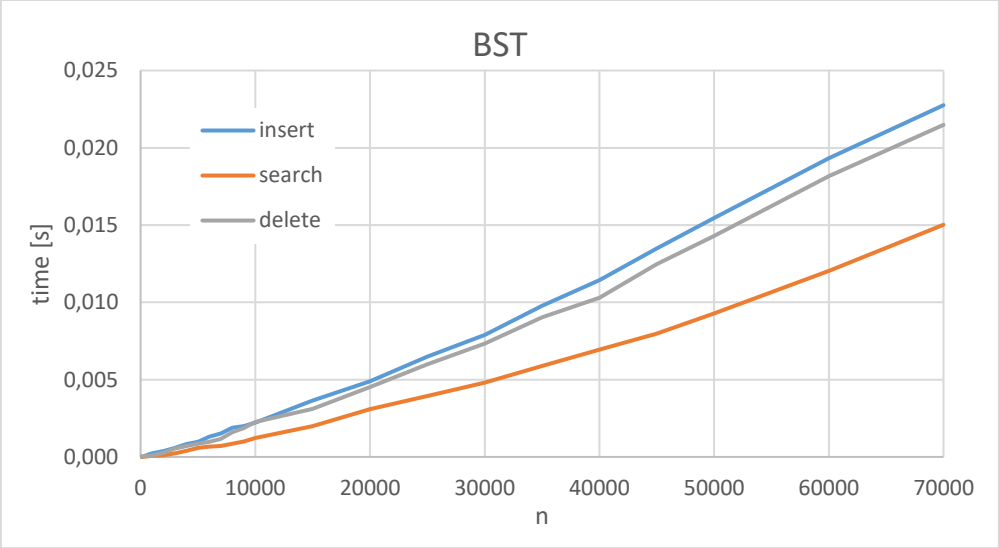
Linked list			
n	**insert**	**search**	**delete**
100	0,0000	0,0000	0,0002
1000	0,0008	0,0020	0,0010
2000	0,0042	0,0086	0,0045
3000	0,0079	0,0191	0,0098
4000	0,0150	0,0391	0,0193
5000	0,0239	0,0618	0,0306
6000	0,0366	0,0912	0,0456
7000	0,0510	0,1277	0,0638
8000	0,0682	0,1695	0,0847
9000	0,0871	0,2171	0,1084
10000	0,1115	0,2763	0,1414
15000	0,2657	0,6803	0,3383
20000	0,5459	1,5736	0,7783
25000	0,9589	2,8816	1,3872
30000	1,5726	4,6168	2,2502
35000	2,3183	6,7403	3,3611
40000	3,2810	9,2352	4,5401
45000	4,4090	11,9249	5,8841
50000	5,9280	15,9689	7,6172
60000	8,7296	22,8262	11,6181
70000	13,6135	35,7752	19,4479

Presented table and chart are based on the time measurements of inserting, searching and deleting elements in **linked-list**. Each measuring point corresponds to the time of performing an operation for n elements, hence searching of the list is more time consuming – it always operates on the whole list of n elements, while deleting has less elements to handle with every repetition (and insertion the other way round).

Time complexity of each operation is O(n), since in the worst case one has to iterate through the whole list to find a suitable key (index). When it comes to searching for an element not present in the list, the program can be terminated if the checked key is greater than the searched one (since the list is sorted), which will reduce the number of operations.

Zuzanna Gawrysiak ID: 148255
Agata Żywot ID: 148258

2. Binary Search Tree



Binary Search Tree			
n	**insert**	**search**	**delete**
100	0,00000	0,00000	0,00003
1000	0,00023	0,00008	0,00009
2000	0,00038	0,00011	0,00028
3000	0,00058	0,00022	0,00054
4000	0,00083	0,00041	0,00072
5000	0,00097	0,00059	0,00085
6000	0,00132	0,00066	0,00098
7000	0,00152	0,00071	0,00117
8000	0,00189	0,00085	0,00160
9000	0,00198	0,00100	0,00187
10000	0,00222	0,00123	0,00227
15000	0,00364	0,00199	0,00311
20000	0,00488	0,00309	0,00451
25000	0,00648	0,00394	0,00598
30000	0,00790	0,00482	0,00734
35000	0,00978	0,00589	0,00903
40000	0,01143	0,00693	0,01031
45000	0,01350	0,00797	0,01247
50000	0,01545	0,00928	0,01430
60000	0,01932	0,01204	0,01818
70000	0,02275	0,01502	0,02148

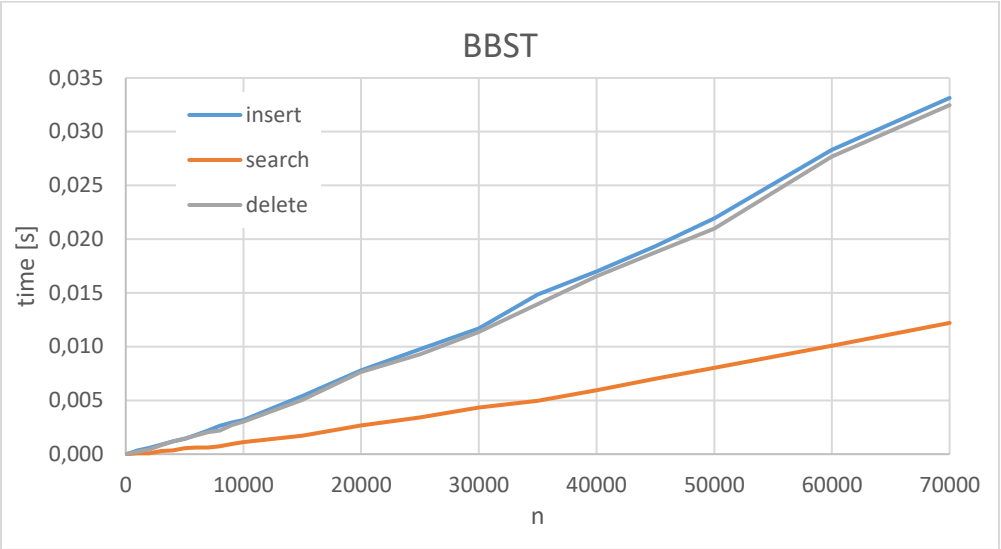
Another type of examined structures is **Binary Search Tree (BST)**. As in the previous case, provided chart and table present the time needed to perform insertion, searching and deletion for n elements.

Having analysed the concept of BST, one could have noticed that, for a single element, the number of nodes needed to be visited before performing any of the aforementioned operations is in the worst case is equal to h, where the value of h represents the height of a BST. Consequently, all processes in BST can be performed in time O(h). Since the elements are chosen randomly, the average complexity for each operation is equal to O(logn). However, one has to be mindful of the worst case scenario where the data is already sorted by key and the tree will be exceptionally skewed. Consequently, time complexity would change to O(n).

Despite all the operations in BST being of the same time complexity class O(logn) , the results have shown that searching performs better compared to insertion and deletion. It might be associated with the fact that the latter operations require some additional procedures. During insertion, a value is initially appended to the appropriate leaf-node which involves creating a new node every time. Taking deleting into consideration, there are three cases that can happen, namely: no subtrees, one or two subtrees. The last two cases require an additional replacement of a node to be deleted with its proper inorder successor. In the case of searching there is no need to perform any additional operation apart from determining whether a node containing the searched value exists or not.

Zuzanna Gawrysiak ID: 148255
Agata Żywot ID: 148258

3. Balanced Binary Search Tree (AVL)



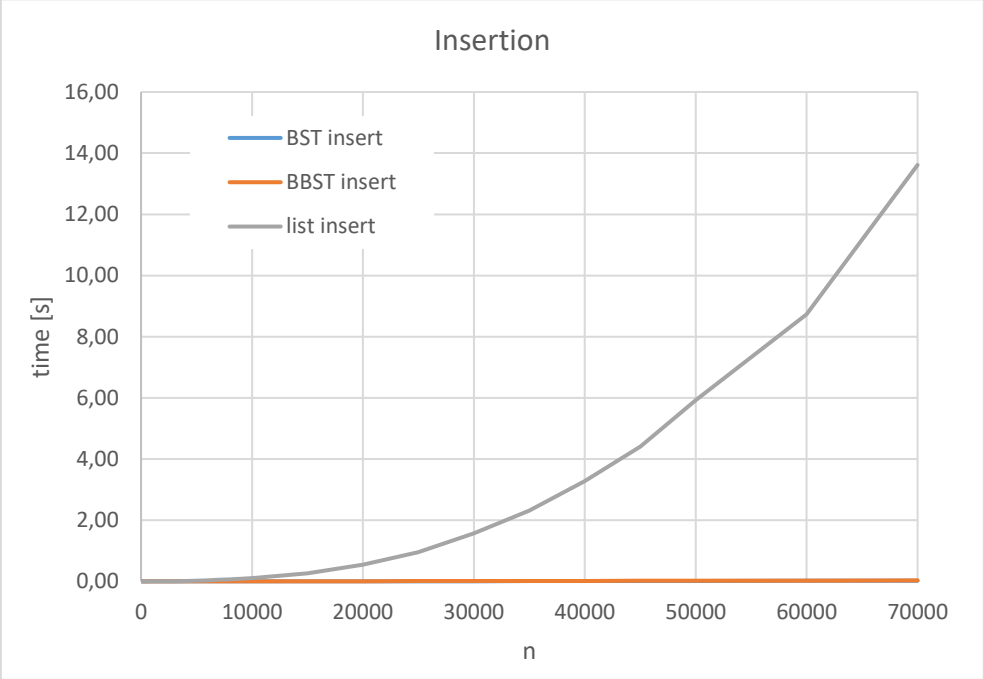
Balanced Binary Search Tree			
n	**insert**	**search**	**delete**
100	0,00000	0,00000	0,00003
1000	0,00034	0,00008	0,00024
2000	0,00060	0,00009	0,00045
3000	0,00088	0,00028	0,00083
4000	0,00117	0,00035	0,00119
5000	0,00143	0,00056	0,00144
6000	0,00181	0,00061	0,00174
7000	0,00220	0,00063	0,00204
8000	0,00264	0,00075	0,00220
9000	0,00294	0,00094	0,00271
10000	0,00319	0,00112	0,00302
15000	0,00541	0,00171	0,00507
20000	0,00781	0,00266	0,00764
25000	0,00975	0,00340	0,00928
30000	0,01169	0,00433	0,01138
35000	0,01486	0,00496	0,01395
40000	0,01698	0,00593	0,01654
45000	0,01934	0,00702	0,01877
50000	0,02195	0,00803	0,02100
60000	0,02831	0,01008	0,02769
70000	0,03314	0,01221	0,03247

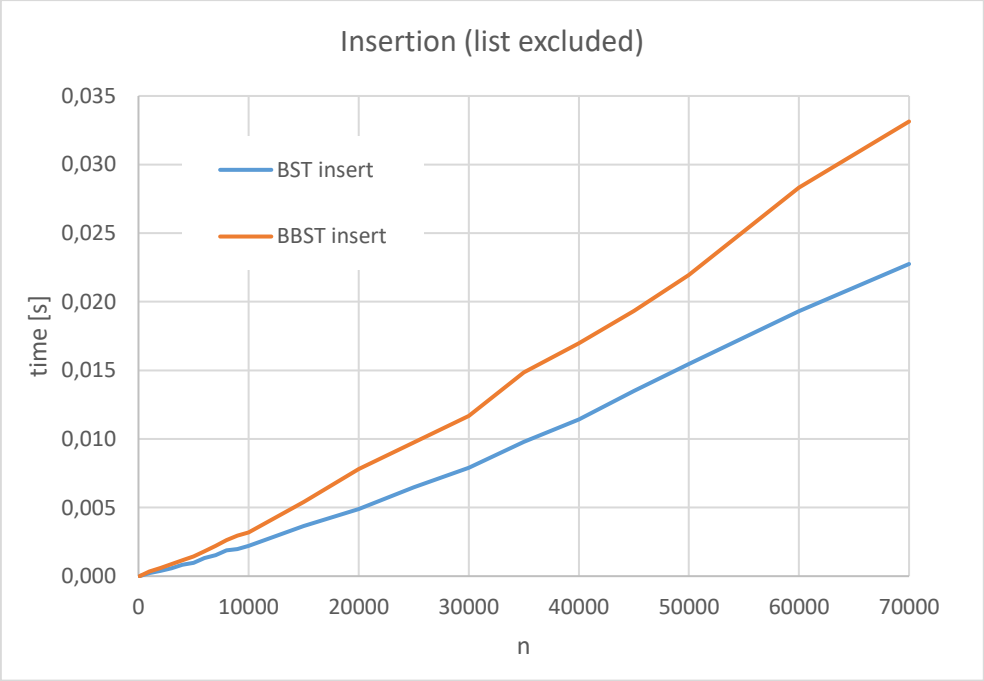
One of the types of tree structures is **Balanced Binary Search Tree (BBST)** which is in fact BST with the additional feature that the difference between the height of the left and right subtree of any node cannot be more than one, hence the height of this type of tree is equal to O(logn). Consequently, the complexity of insertion, searching and deletion in BBST is equal to O(logn) in both the average and the worst cases.

However, the results obtained picture even more significant difference in time of performance between searching and the remaining operations than it has been observed in BST. This is the result of implementing AVL - the type of BBST which checks the difference in the heights of the left and right subtrees in every node during these two operations and self-balances if necessary. This additional process contributes to slower performance of insertion and deletion.

Zuzanna Gawrysiak ID: 148255
Agata Żywot ID: 148258

4. Dependence of insertion time of the number n for given structures





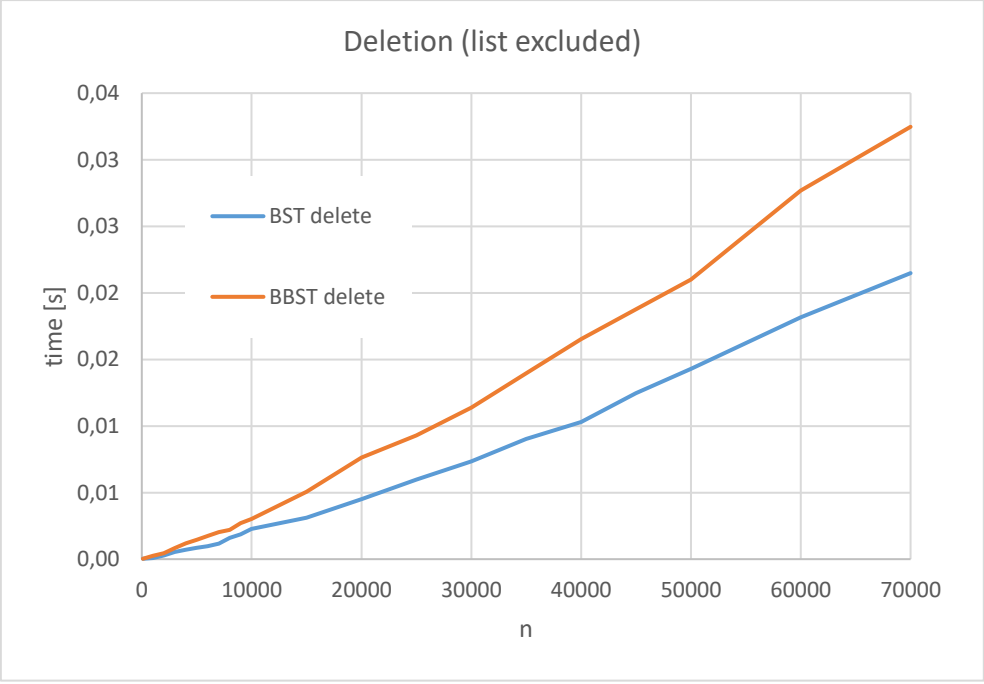
n	BST insert	BBST insert	list insert
100	0,00000	0,00000	0,00000
1000	0,00023	0,00034	0,00080
2000	0,00038	0,00060	0,00420
3000	0,00058	0,00088	0,00790
4000	0,00083	0,00117	0,01500
5000	0,00097	0,00143	0,02390
6000	0,00132	0,00181	0,03660
7000	0,00152	0,00220	0,05100
8000	0,00189	0,00264	0,06820
9000	0,00198	0,00294	0,08710
10000	0,00222	0,00319	0,11150
15000	0,00364	0,00541	0,26570
20000	0,00488	0,00781	0,54590
25000	0,00648	0,00975	0,95890
30000	0,00790	0,01169	1,57260
35000	0,00978	0,01486	2,31830
40000	0,01143	0,01698	3,28100
45000	0,01350	0,01934	4,40900
50000	0,01545	0,02195	5,92800
60000	0,01932	0,02831	8,72960
70000	0,02275	0,03314	13,61350

The operation of adding n elements has time complexity O(n) for the linked list, which was explained in point 1. For BST it is O(h), where h is the height of the tree. The worst case is O(n), since then we need to traverse all elements to insert a given number. Considering BBST, its general time complexity is O(logn), however our time measurements included also balancing a new tree after insertion. Hence its performance is slightly worse than BST's.

The advantage of single linked-list is its simplicity and lack of need to perform any additional operations. On the other hand, it is significantly slower than the other presented structures. BST seems to be the best choice considering insertion, however it imposes a threat of obtaining a highly imbalanced tree, which results in poor performance. In the long run, the best choice might be BBST, since with every insertion it preserves its balanced structure (at least in our implementation) and ensures decent performance with every repetition. Time spent on balancing the tree pays off with a stable, well-organized data structure.

Zuzanna Gawrysiak ID: 148255
Agata Żywot ID: 148258

5. Dependence of deletion time of the number n for given structures



n	BST delete	BBST delete	list delete
100	0,00003	0,00003	0,00020
1000	0,00009	0,00024	0,00100
2000	0,00028	0,00045	0,00450
3000	0,00054	0,00083	0,00980
4000	0,00072	0,00119	0,01930
5000	0,00085	0,00144	0,03060
6000	0,00098	0,00174	0,04560
7000	0,00117	0,00204	0,06380
8000	0,00160	0,00220	0,08470
9000	0,00187	0,00271	0,10840
10000	0,00227	0,00302	0,14140
15000	0,00311	0,00507	0,33830
20000	0,00451	0,00764	0,77830
25000	0,00598	0,00928	1,38720
30000	0,00734	0,01138	2,25020
35000	0,00903	0,01395	3,36110
40000	0,01031	0,01654	4,54010
45000	0,01247	0,01877	5,88410
50000	0,01430	0,02100	7,61720
60000	0,01818	0,02769	11,61810
70000	0,02148	0,03247	19,44790

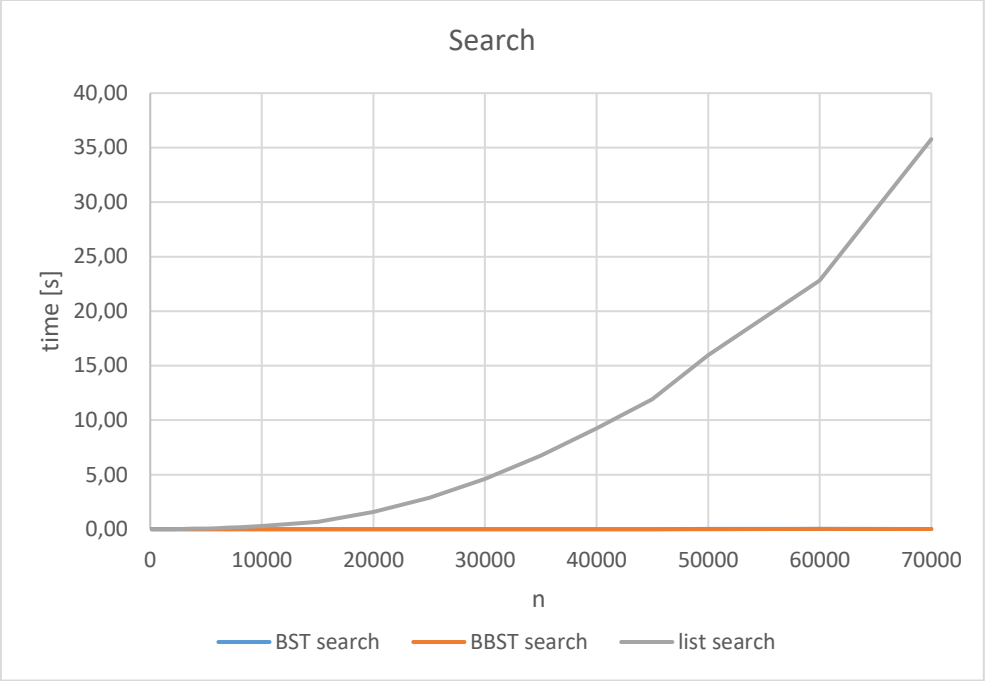
The operation of deleting elements has similar characteristics to the operation of inserting elements. Time complexities for every structure are the same as with insertion, i.e. O(n) for linked-list, O(h) for BST and O(logn) for BBST. Again BBST requires secondary balancing after each deletion, which influences its performance.

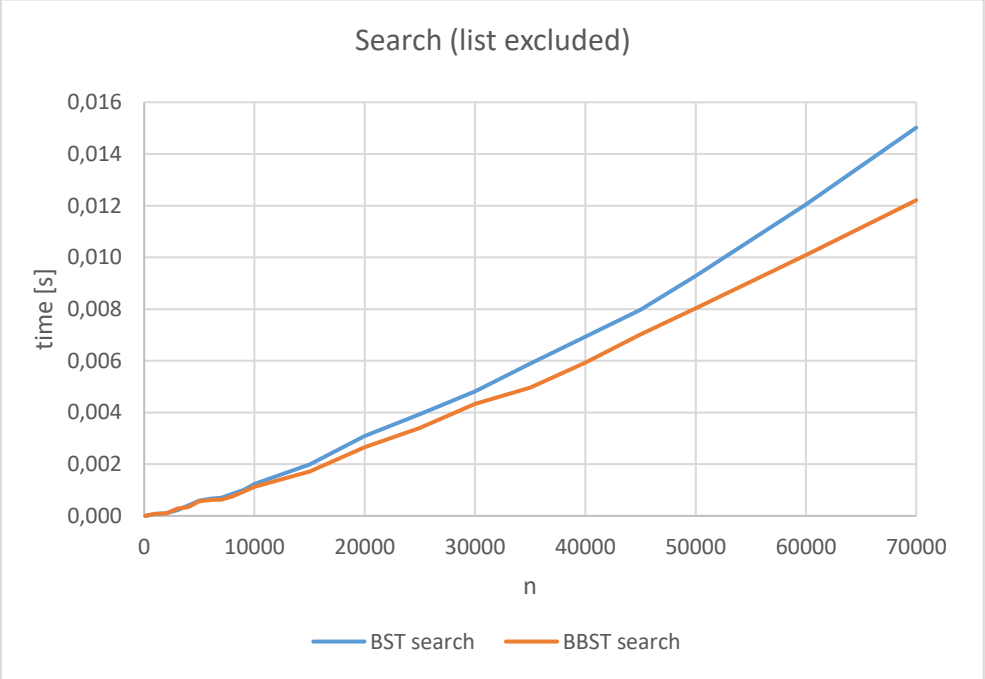
Minor time differences between inserting and deleting elements may be related to the dynamic data allocation and the need to free or assign memory.

Advantages and disadvantages remain the same as for insertion (see point 4).

Zuzanna Gawrysiak ID: 148255
Agata Żywot ID: 148258

6. Searching for the selected element for given structures





n	BST search	BBST search	list search
100	0,00000	0,00000	0,00000
1000	0,00008	0,00008	0,00200
2000	0,00011	0,00009	0,00860
3000	0,00022	0,00028	0,01910
4000	0,00041	0,00035	0,03910
5000	0,00059	0,00056	0,06180
6000	0,00066	0,00061	0,09120
7000	0,00071	0,00063	0,12770
8000	0,00085	0,00075	0,16950
9000	0,00100	0,00094	0,21710
10000	0,00123	0,00112	0,27630
15000	0,00199	0,00171	0,68030
20000	0,00309	0,00266	1,57360
25000	0,00394	0,00340	2,88160
30000	0,00482	0,00433	4,61680
35000	0,00589	0,00496	6,74030
40000	0,00693	0,00593	9,23520
45000	0,00797	0,00702	11,92490
50000	0,00928	0,00803	15,96890
60000	0,01204	0,01008	22,82620
70000	0,01502	0,01221	35,77520

Comparing the structures in terms of the time needed to search n elements plainly shows that the linked list again lags behind BST and BBST with its complexity equal to O(n) (see point 1). Only after excluding the linked list from the chart can one notice that BBST performed better than BST. The result derives from time complexity equal to O(h): in BST the average case is O(logn) and the worst – O(n) whereas in the case of BBST it is O(logn) no matter whether handling the worst or the average case. Due to randomized data order it can be assumed the complexity of searching elements in BST is of O(logn); however, the results are based on 100 repartitions in which the order of elements might not always be as favorable. Therefore, knowing that elements in a structure will be sought many a time, it would be beneficial to implement BBST.

Summary

Analyzing the results of the experiment one can spot both advantages and disadvantages regarding each of the aforementioned structures. Linked list is definitely easy to understand and implement – there are no additional operations that have to be carried out as in the case of trees. However, the time needed to perform insertion, deletion and search is considerably worse compared to the remaining structures. Taking BST into consideration, in the case of handling randomized order of elements it works well, although it is possible to obtain a skewed BST of significantly worse efficiency (in the extreme scenario O(n)). It does not concern BBST in which it is guaranteed to have quick access to every element in the structure. However, the efficiency comes at the cost of the time spent on possible re-balancing after any disturbance of BBST structure. What is more, the implementation of BBST is more difficult compared to linked list or regular BST. Therefore, before choosing the best structure to store data one should first think about the order of elements and the operations that will be performed most often.

Sources:

- "Algorytmy + struktury danych = programy" Niklaus Wirth
- Algorytmy i struktury danych - Studia Informatyczne (mimuw.edu.pl)
- https://www.geeksforgeeks.org/