Comparison between Binary search tree and AVL search tree

Nijat Sadikhov

February 17, 2021

1 Introduction

In this paper, we compare two comparison-based data structures: Binary Search Tree and AVL Search Tree.

1.1 Binary Search Tree

In order to implement BST structure, first, we need to construct a tree by inserting elements from an array one by one. The rule for this structure is that the first inserted elements becomes a value of a root node. The root may have at most 2 children nodes: left and right. The way to choose where the next element should be inserted is comparing for the value. If value is less than the node's value, it should become node's left child. Otherwise, becoming node's right child. On the other hand, for searching an element, almost the same path is followed. To make it clear, we search for an element by starting from root and going down 1-by-1 according to the comparison rule mentioned above. When the requested element is found, the search method returns back true.

The best time complexity case for BST insertion and search is O(log(n)) and the worst one is O(h) = O(n). The h means the height. In other words, an ordered array is the worst case which will turn the tree into a linked list.

1.2 AVL Search Tree

When it comes to AVL search tree structure, it is similar to BST structure in terms of each node having at most 2 children and the same comparison method when inserting a new element. But, additionally, for AVL, each time a new element is inserted, there should be a check-up from inserted point to make sure that the parent node's balance factor is not equal to either 2 or -2. The balance factor is the difference of the left node's height and the right node's height. Node's height can be obtained by finding the max of its children's height and adding 1. In case of balance factor being 2 or -2, updating of the node's place is expected by implementing rotations.

There are 4 rotation cases: Left-Left, Left-Right, Right-Left and Right-Right.

BF -> Balance Factor
If node’s BF is -2 and its left child’s BF is -1, then left-left case is required.
If node’s BF is -2 and its left child’s BF is 1, then left-right case is required.
If node’s BF is 2 and its right child’s BF is -1, then right-left case is required.
If node’s BF is 2 and its right child’s BF is 1, then right-right case is required.

Left-Left: Node becomes child of its right child
Right-Right: Node becomes child of its left child
Left-Right: Node’s left child becomes child of its right child and again Left-Left rotation happens
Right-Left: Node’s right child becomes child of its left child and again Right-Right rotation happens

After rotation process ends, a check-up occurs until the node’s balance factor is 0.

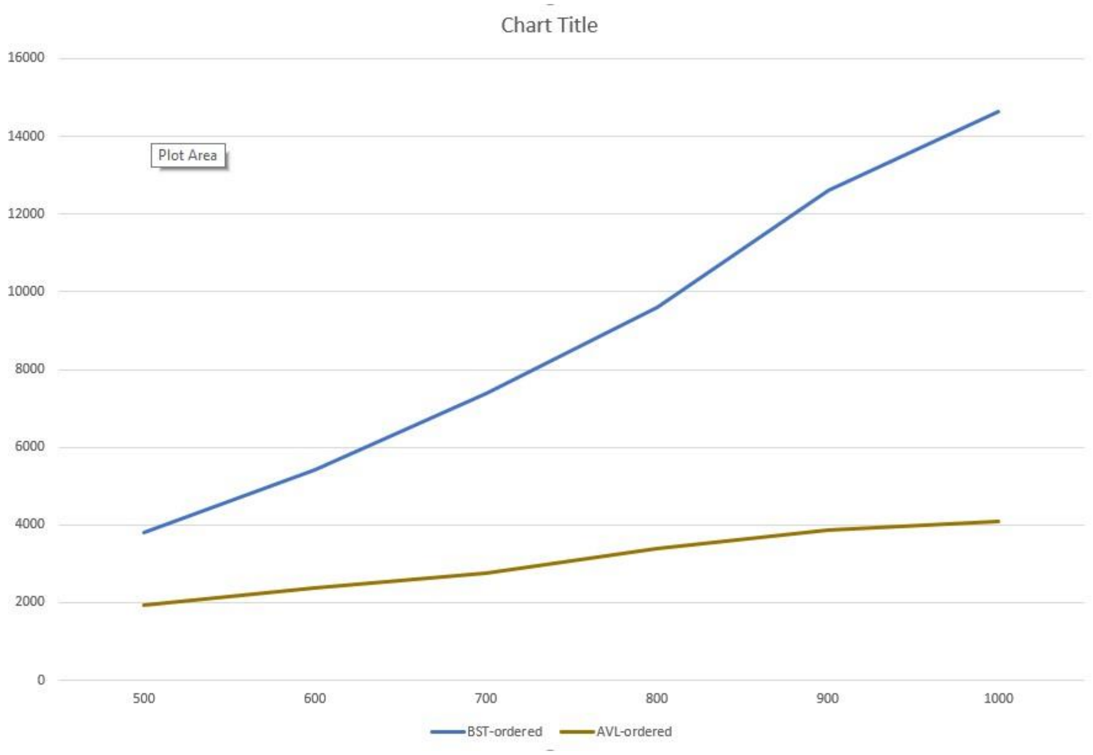
The time complexity of AVL insertion and search is O(log(n)). In comparison with BST, The search process is guaranteed to be O(log(n)), since each insertion updates the tree and turns it into the best case.

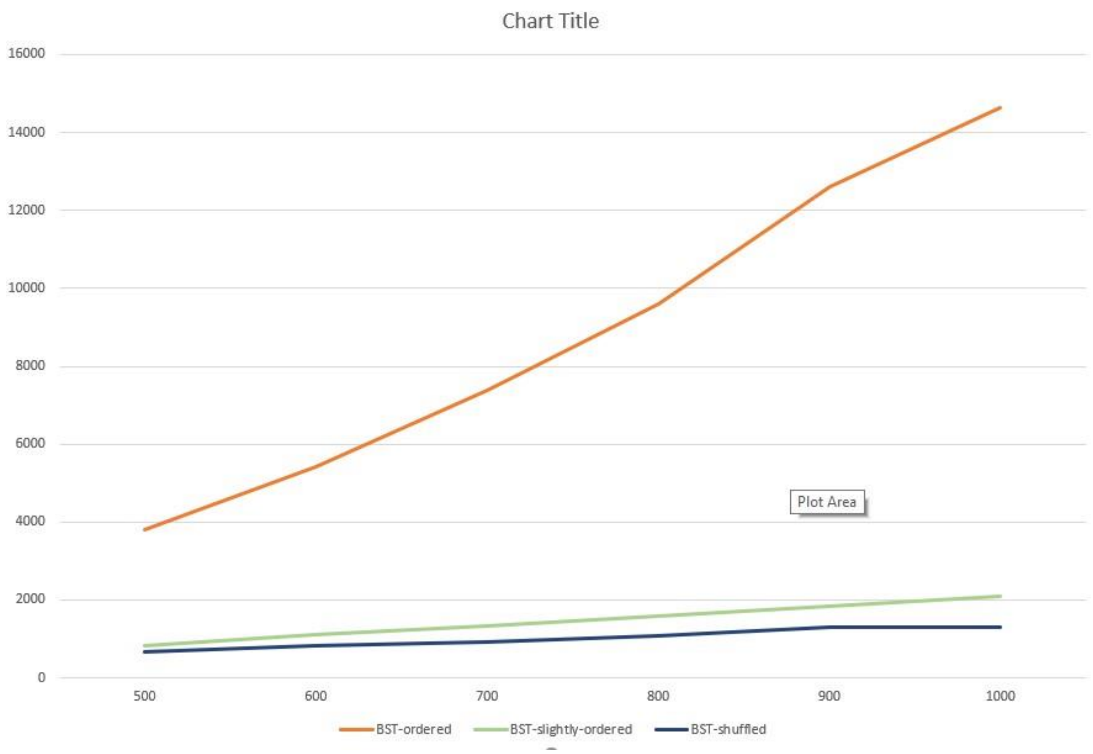
2 Methodology

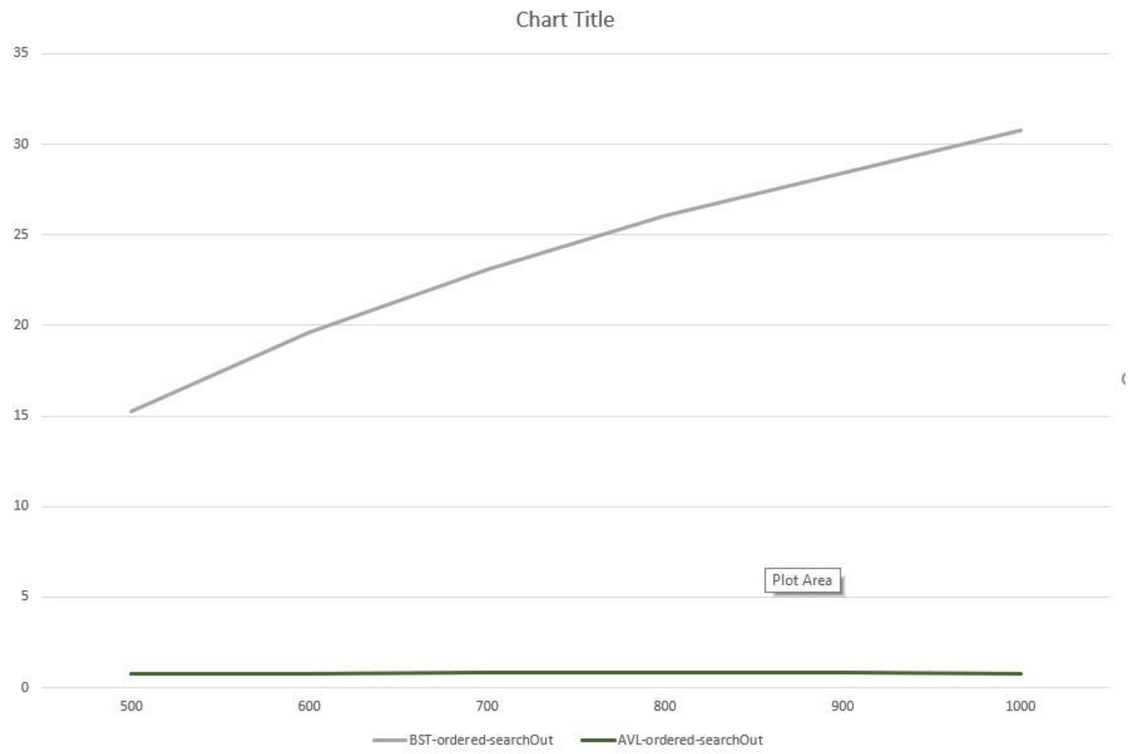
The algorithms were implemented in C++. The results were generated for three categories: randomly shuffled arrays of values 0, 1, . . . , n – 1, ordered arrays and slightly ordered arrays. The algorithms were tested on arrays of sizes from 100, 200, 300, . . . , 10000. In forementioned categories, each algorithm was given the same input data. Each array size was tested 100 times. The result for each size is the average time it took for each algorithm to insert the input array and search for an element which is included and for the one that is not included.

3 Results

In average case, for ordered arrays, BTS insertion takes more time than the all array cases of AVL insertion. In the following arrays: ordered, slightly ordered and shuffled - AVL insertion time is almost the same: ordered faster than shuffled and shuffled faster slightly ordered. When it comes to searching an element inside of a Tree, for AVL structure, it takes less than BST structure in all array cases. BST's searching an out-element is the slowest for an ordered case. AVL's ordered search for an in-element is faster than BST's one.

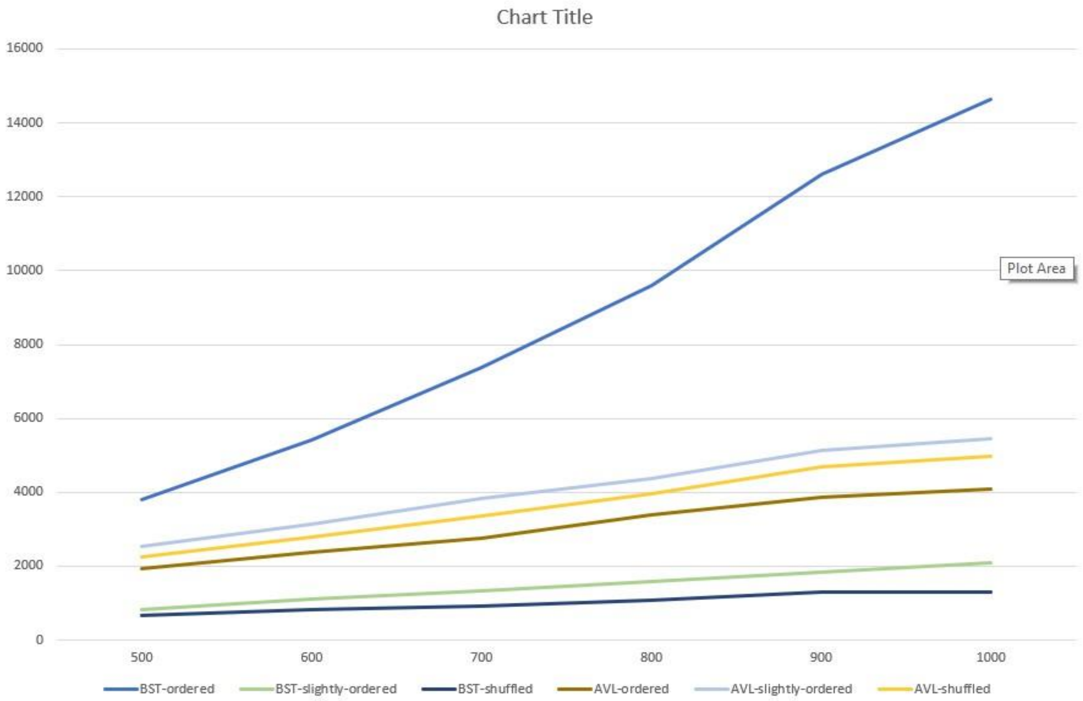






4 Conclusions

We can conclude that AVL tree structure is the best for insertion and search on average. Insertion of slightly and shuffled arrays into BST might be faster than AVL but the ordered of it is the slowest.



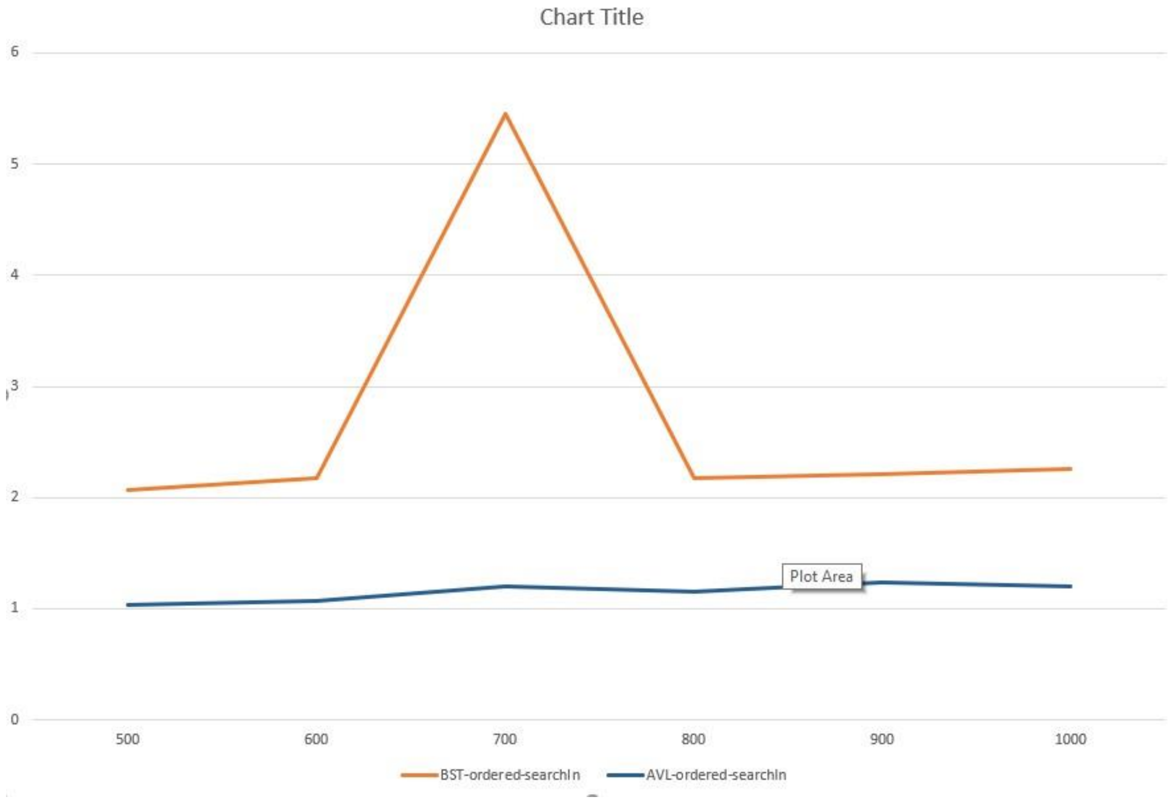


Chart Title
6
5
4
3
2
1
0
500
600
700
800
900
1000
Plot Area
BST-ordered-searchIn
AVL-ordered-searchIn

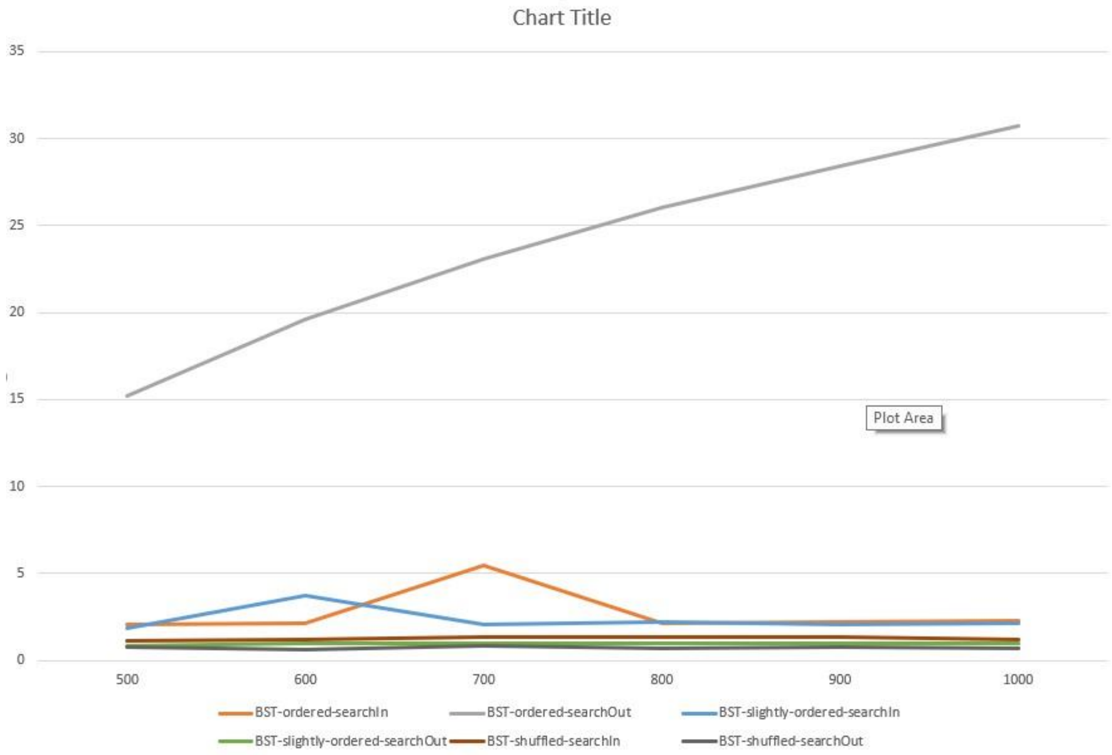


Chart Title
35
30
25
20
15
10
5
0
500
600
700
800
900
1000
Plot Area
BST-ordered-searchIn
BST-ordered-searchOut
BST-slightly-ordered-searchIn
BST-slightly-ordered-searchOut
BST-shuffled-searchIn
BST-shuffled-searchOut

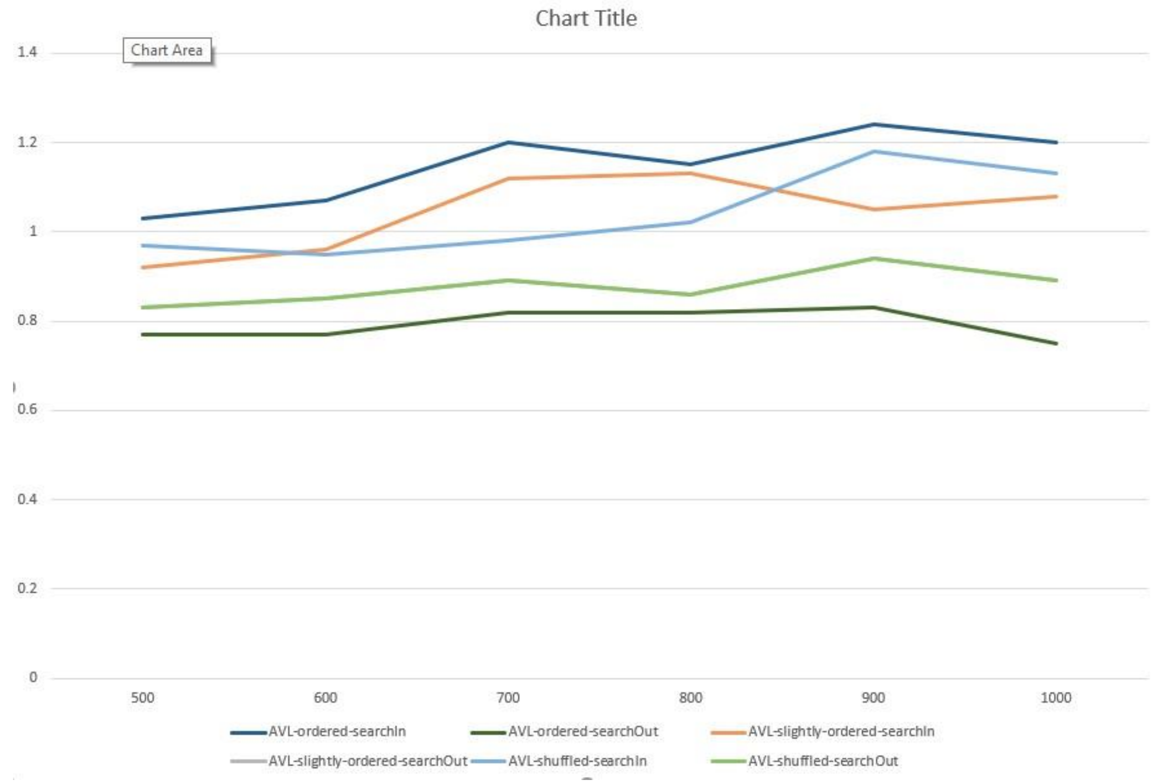


Chart Title
Chart Area
1.4
1.2
1
0.8
0.6
0.4
0.2
0
500
600
700
800
900
1000
AVL-ordered-searchIn
AVL-ordered-searchOut
AVL-slightly-ordered-searchIn
AVL-slightly-ordered-searchOut
AVL-shuffled-searchIn
AVL-shuffled-searchOut

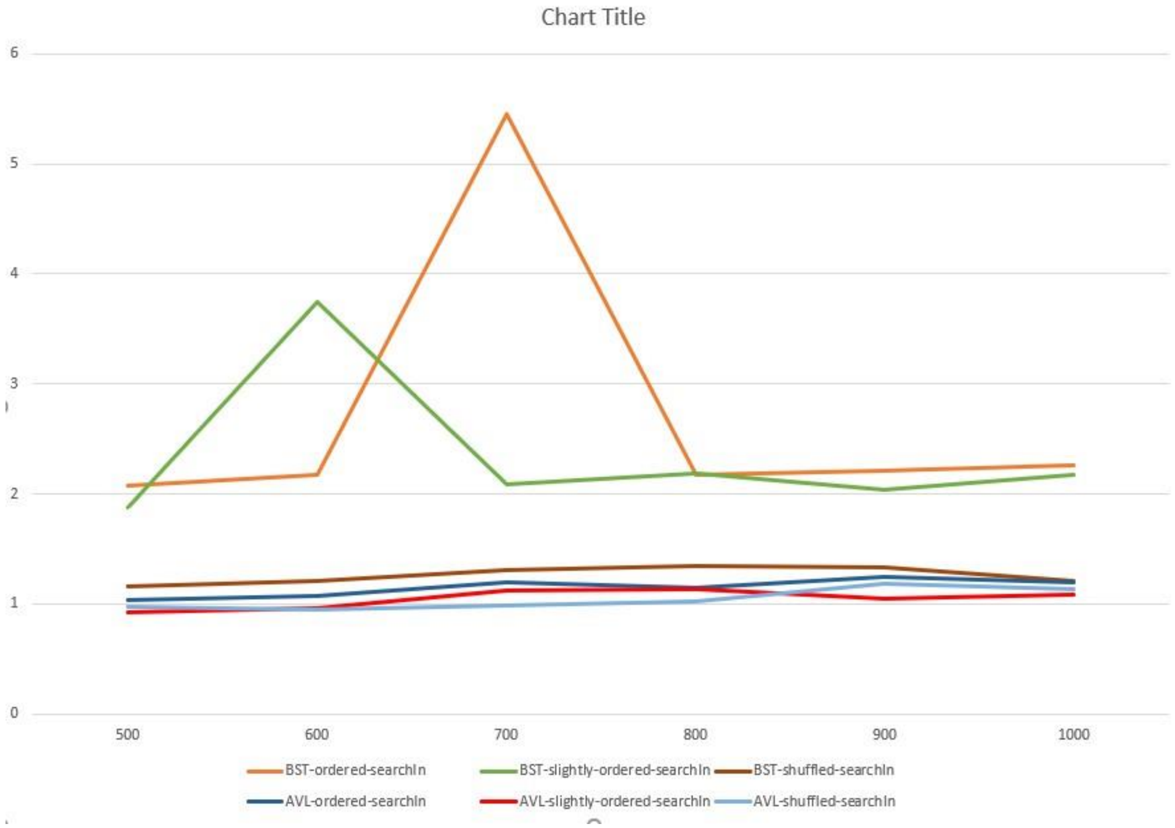


Chart Title
6
5
4
3
2
1
0
500
600
700
800
900
1000
BST-ordered-searchIn
BST-slightly-ordered-searchIn
BST-shuffled-searchIn
AVL-ordered-searchIn
AVL-slightly-ordered-searchIn
AVL-shuffled-searchIn

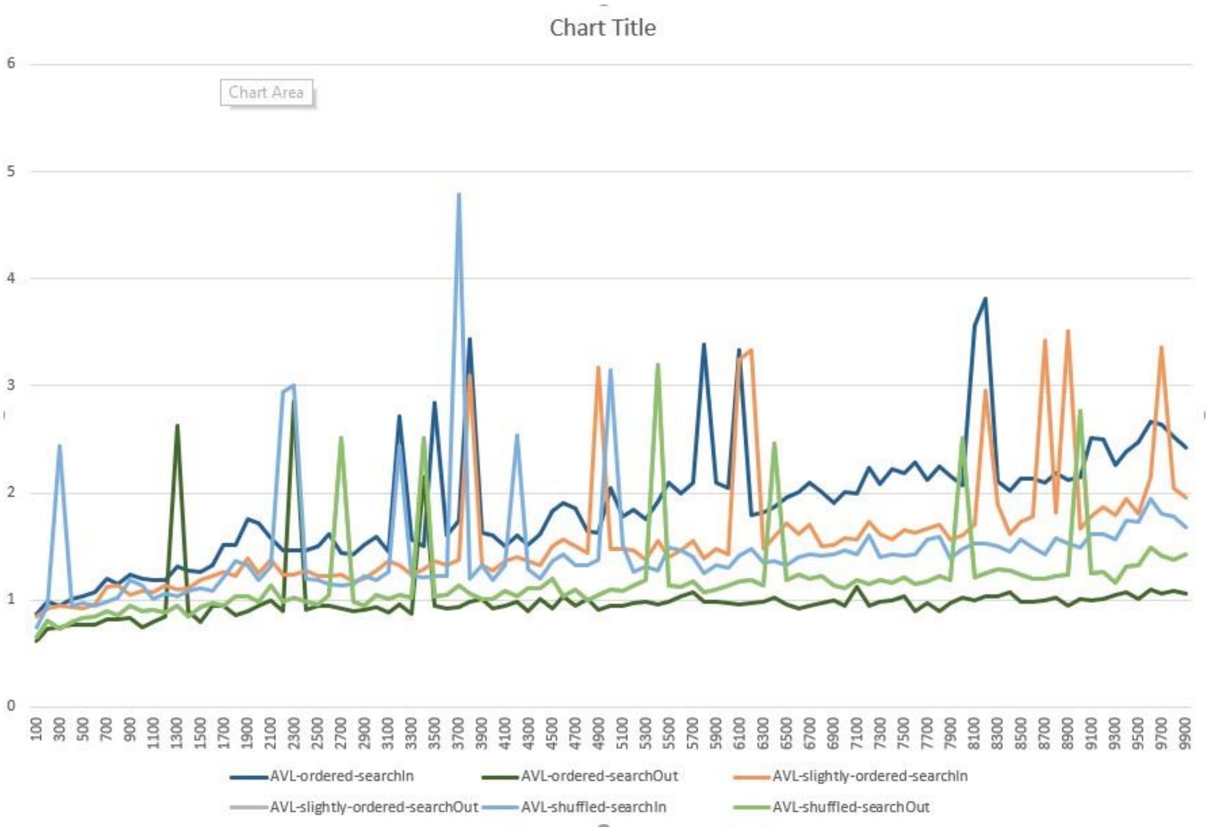


Chart Title
Chart Area
6
5
4
3
2
1
0
100 300 500 700 900 1100 1300 1500 1700 1900 2100 2300 2500 2700 2900 3100 3300 3500 3700 3900 4100 4300 4500 4700 4900 5100 5300 5500 5700 5900 6100 6300 6500 6700 6900 7100 7300 7500 7700 7900 8100 8300 8500 8700 8900 9100 9300 9500 9700 9900
AVL-ordered-searchIn
AVL-ordered-searchOut
AVL-slightly-ordered-searchIn
AVL-slightly-ordered-searchOut
AVL-shuffled-searchIn
AVL-shuffled-searchOut

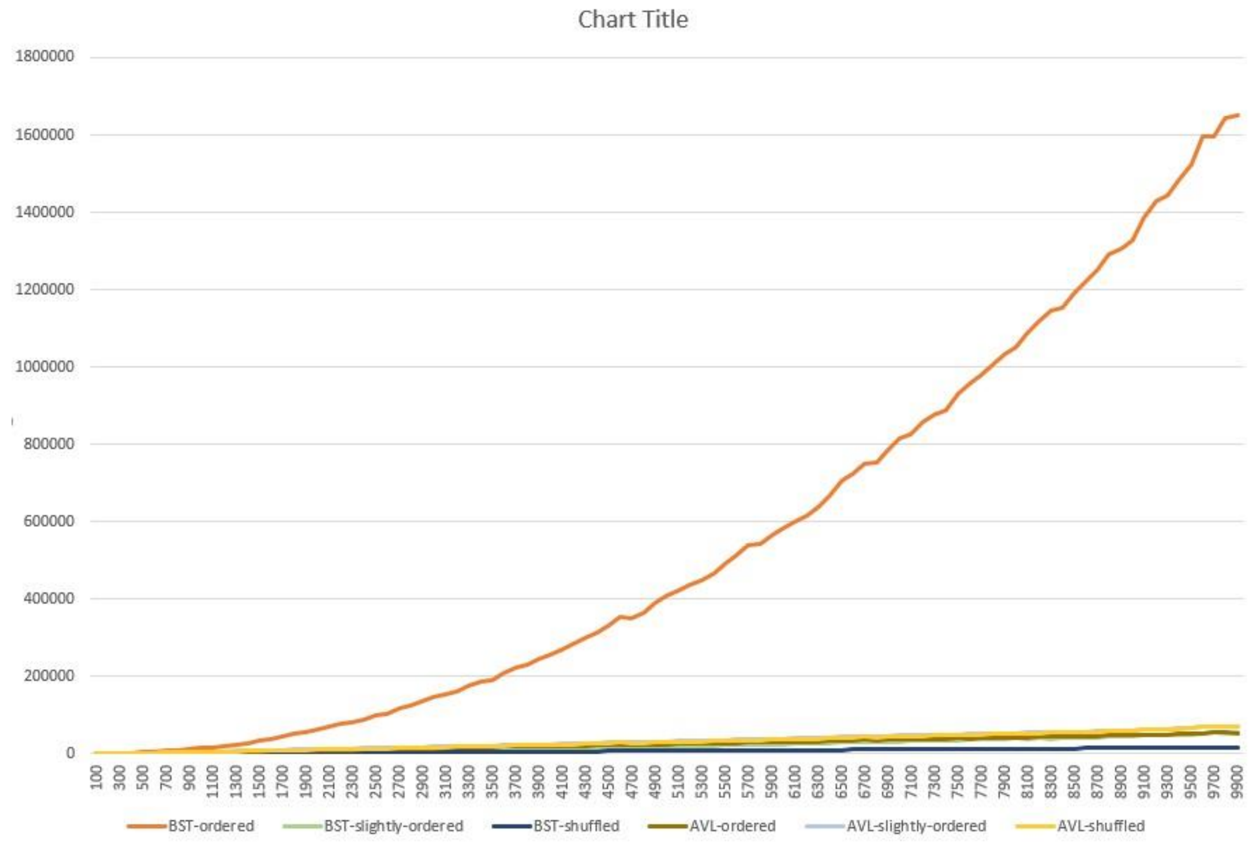


Chart Title
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
100 300 500 700 900 1100 1300 1500 1700 1900 2100 2300 2500 2700 2900 3100 3300 3500 3700 3900 4100 4300 4500 4700 4900 5100 5300 5500 5700 5900 6100 6300 6500 6700 6900 7100 7300 7500 7700 7900 8100 8300 8500 8700 8900 9100 9300 9500 9700 9900
BST-ordered
BST-slightly-ordered
BST-shuffled
AVL-ordered
AVL-slightly-ordered
AVL-shuffled