Dictionary, Comparison AVL vs BST Trees

Faisal Kassem

January 22, 2023

1. Introduction

In this paper, we investigate a problem: Create a dictionary using AVL and BST trees as the data store. We aim to investigate and compare the effectiveness of AVL and BST trees.

Trees may have a bunch of various methods and operations (especially depending on type and ways of implementation), but in the analysis, we will focus on insertion and searching.

1.1. BST Tree

A tree in computer science is a data structure where data is stored in hierarchically ordered Nodes. For example, the key of each internal node is greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. In a binary tree, each node doesn't have more than two children.

BST trees allow binary search for fast lookup and addition of data items. Since the nodes in a BST are placed so that each comparison skips about half of the tree.

The performance of a binary search tree depends on the order of insertion of the nodes into the tree, since the arbitrary insertions mainly lead to the degeneracy of the tree structure.

The complexity analysis shows: On average insert and search take
$O(\log n)$ for n nodes. In the worst case: $O(n)$

Searching in a binary search tree for a specific key can be implemented in a recursive or iterative way.

In the case of this analysis searching, and clearing of the tree (besides we are not going to compare the speed of this operation) for the BST tree is done iteratively.

1.1.1. Searching

Searching begins by examining the root node. If the tree is *nullptr*, the key being searched for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and the node is returned. If the key is less than that of the root, the search proceeds by examining the left subtree. Similarly, if the key is greater than that of the root, the search proceeds by examining the right subtree. This process is repeated until the key is found or the remaining subtree is *nullptr*

1.1.2. Insertion

The insertion causes the BST representation to change dynamically. The procedure goes in this way: we start from the root, and if the root is *nullptr* the newly inserted value becomes root, otherwise basing if the value is $<$ than the current Node value we proceed to the left, or to the right if the value is $>$ than current Node value. New nodes are inserted as leaf nodes in the BST.

1.2. AVL Tree

AVL Tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree. AVL Trees can be expressed as the extension of BSTs. Similarly like in BST, the value stored in a node is greater than any node in the left subtree, and values in the right subtree are greater or equal to the value in the node. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

Lookup, and insertion all take $O(\log n)$ time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

As was mentioned above, AVL trees maintain balance by using rotations to fix tree balance during an insertion

The first part of an insertion proceeds normally, just as in BST. However, after the insertion, an additional operation is made upward of the insertion path, updating the balance information and balancing using rotations.

To be able to do so, we have to store additional information in a node. Either it would be necessary to maintain the height of a subtree or the balance factor of a node.

1.2.1. Balance Factor

In a binary tree, the balance factor of node X is defined to be the height difference.

$$BF(X) := Height(RightSubtree(X)) - Height(LeftSubtree(X))$$

A binary tree is defined to be an AVL Tree if:
$BF(X) \in \{-1, 0, 1\}$

If $BF(X) < 0$ a node is "left-heavy", one with $BF(X) > 0$ is called "right-heavy", and $BF(X) = 0$ is "balanced".

Read-only operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree. Still, modifications have to observe and restore the height balance of the sub-trees. This in total $O(h) + O(h) + 2 \cdot O(1) = O(h)$.

1.2.2. Insertion

Inserting a node into an AVL Tree initially follows the same process as inserting it into a Binary Search Tree. If Tree is empty, the node is inserted as the root of the tree. If the tree is not empty, then we go down the root, and recursively go down the tree searching for the location to insert the new node. The comparison function guides this traversal. In this case, the node always replaces a NULL reference (left or right) of an external node in the tree i.e., the node is either made a left-child or a right-child of the external node. Secondly proceed backward along the insertion path, updating balance factors. We also check if the AVL tree property is not violated. A violation occurs when an insertion changes the heights of subtrees. So if after insertion we notice a node that would have +2 or -2, we perform node rotations. A correctly applied rotation will balance the tree and

maintain the AVL tree property. The update checks and rotate procedure finishes if we reach the root, or we reach a balance factor of 0 after a rotation.

The general idea is that we start at the parent of the leaf node that we have inserted and we proceed upward the tree (in the direction of the root). If the height of the left subtree was increased after insertion, then decrement the balance factor. If the height of the right subtree was increased after insertion, then increment the balance factor. If the balance factor became 0, then we can stop the updating process.

Since with a single insertion the height of an AVL subtree cannot increase by more than one, the temporary balance factor of a node after insertion will be in the range $[-2, +2]$. For each node checked, if the temporary balance factor remains in the range from −1 to +1 then only an update of the balance factor and no rotation is necessary. However, if the temporary balance factor is ± 2, the subtree rooted at this node is AVL unbalanced, and a rotation is needed.

1.2.3. Rotations

There are four cases: left-left, left-right, right-left, and right-right.

Which one is picked depends on the balance factors of the node in which the AVL tree property was violated and in the balance factor of the subtree into which the node was inserted.

In the left-left case: the balance factor in some node X is -2, and the balance factor in the left node is -1. To correct the tree, we perform a right rotation at node X.

A mirror image of the left-left case is the right-right case: the balance factor in node X is +2, and the balance factor in the right node is +1. To correct the tree, we perform a left rotation at node X.

A left-right case occurs when the balance factor in node X is -2, and the balance factor of the left subtree is +1. In that situation, we first left-rotate the left child of X, and then right-rotate X.

Similarly, a right-left case is the mirror image of the left-right case: the balance factor in node X is +2, and the balance factor in the right subtree is -1. To correct the tree, right rotate the right child of X, and then left rotate X.

1.2.4. Searching

Searching for a specific key in an AVL tree can be done the same way as that of any balanced or unbalanced binary search tree. The number of comparisons required for a successful search is limited by the height h and for an unsuccessful search is very close to h, so both are in O(log n).

2. Methodology

The data structures were implemented in C++. The results were generated for the following:

Functionality Tested:

1. Insertion (How long on average it takes to insert a new element in a non-empty tree)
   1.1. Insertion Time for AVL
   2.1. Insertion Time for BST
2. Search Time (How long on average it takes to search for an element) element stored in a tree
   2.1 Search Time for AVL
   2.2 Search Time for BST
3. Search Time (How long on average it takes to search for an element) element is not stored in a tree

3.1 Search Time for AVL
3.2 Search Time for BST

The input is divided into three groups:

1. Random Data (data taken from the list in random order)
2. Ordered Data (data taken from the sorted list)
3. Nearly ordered data (data which is sorted partially, and the coefficient for the unsorted part was chosen to be 10%)

The algorithms were tested on data of sizes from 100, 200, 300, ... 10000. The data type used in the analysis is c++ ***string*** data type. The data for points 1 and 2 (mentioned above) can be accessed here: https://www.mit.edu/~ecprice/wordlist.10000 .
The data for point 3 is randomly made up of words not stored in the link above.
Each dataset's
size was tested 100 times.
The result for each size is the average time it took for each algorithm to perform above mentioned functionality.

3. Results

According to the conducted analysis, the results in Figure 1 depicted that in the case when random data (randomly picked word from the wordlist) is inserted into the AVL or BST tree, AVL performance is slightly worse compared to BST

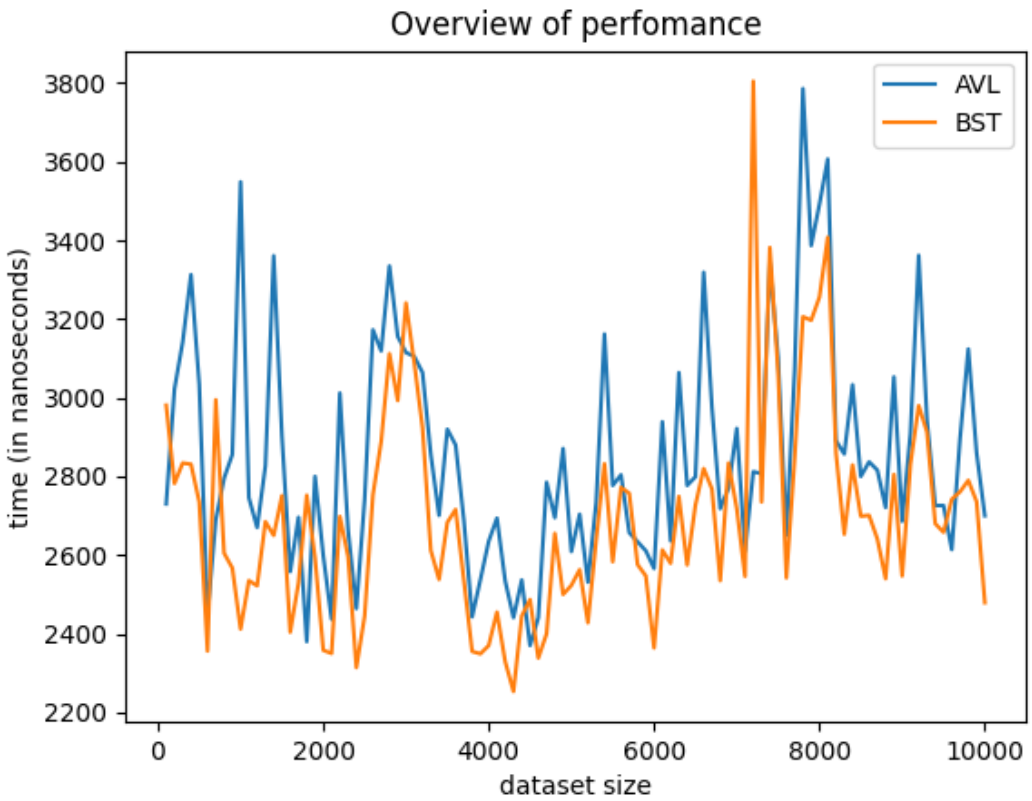

Figure 1. Average time BST and AVL trees took to insert nth element in the range from 0...10000

Nonetheless, when it comes to the insertion of the data (see Figure 2) which is sorted (in our cases ordered words) the Insertion time into BST has grown up incredibly and is increased linearly corresponding to the size of the tree

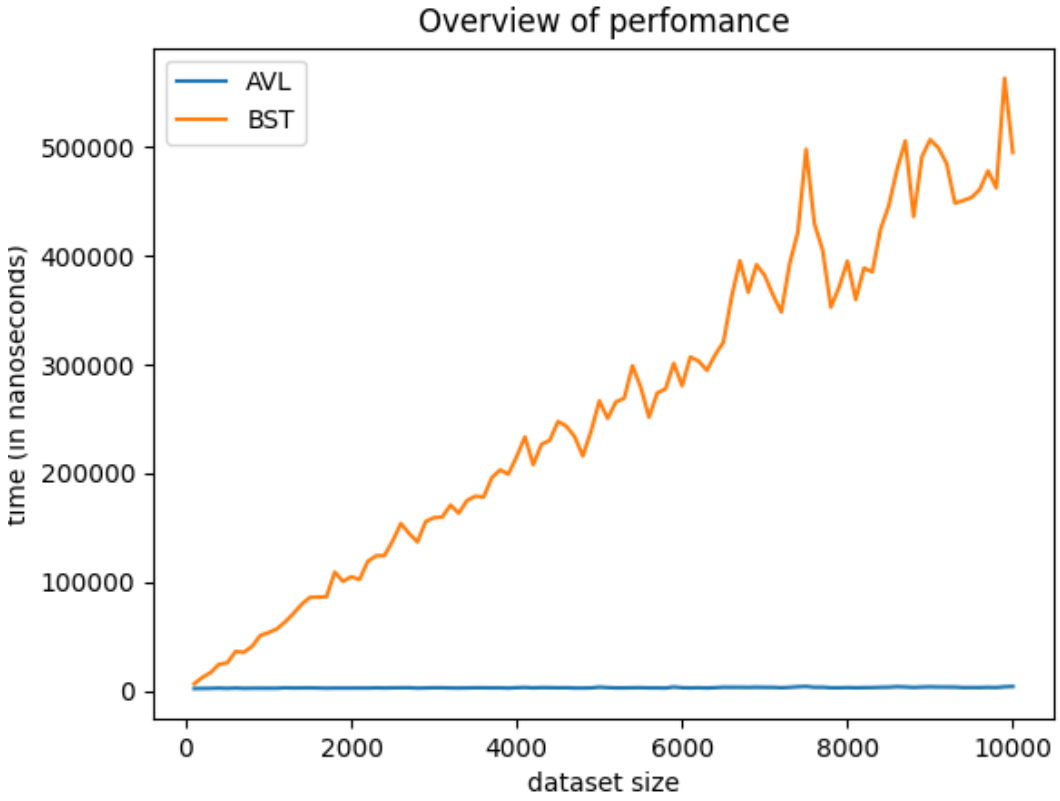

Figure 2. Average time BST and AVL trees took to insert nth element in the range from 0...10000 for ordered data

On the other side, when the data is nearly ordered, in general, AVL and BST performed the insertion with the same time tendency, but BST in most of the cases has peaks higher than AVL, and the lower the coefficient of elements which are not in order in data set, the worse BST performs eventually.

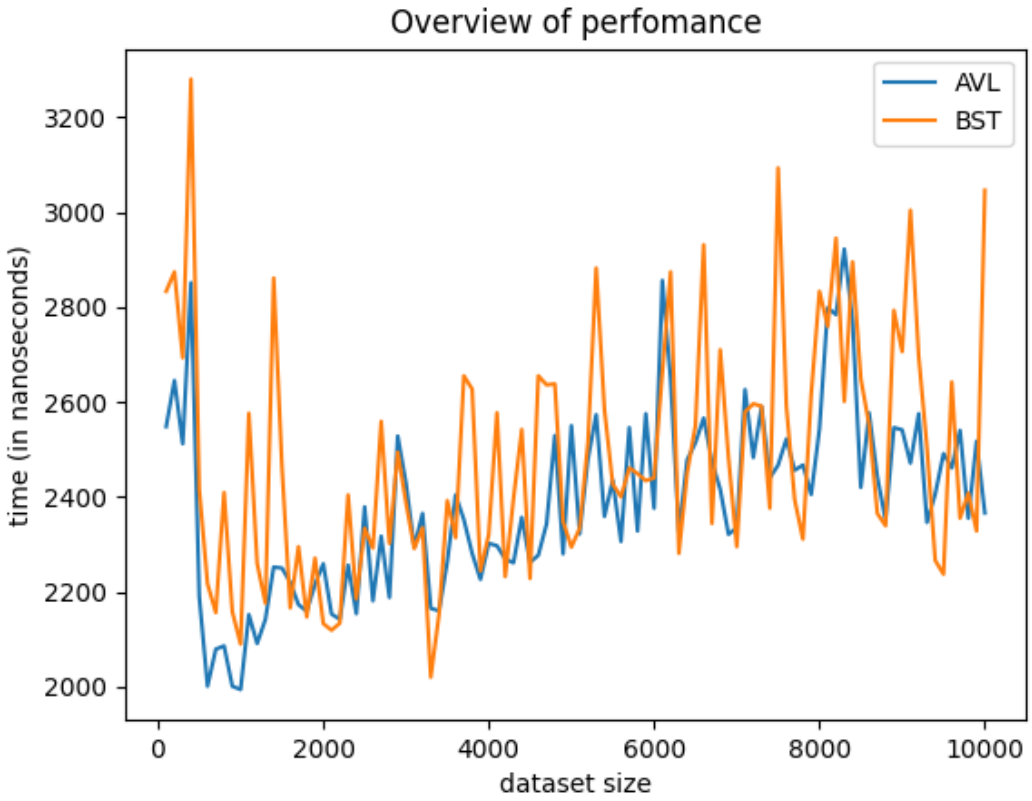

Figure 3. Average time BST and AVL trees took to insert nth element in the range from 0...10000 for nearly ordered data

Considering the case when we need to search for a particular piece of data (word) in trees, to investigate we can look at Figure 4, which depicts the input in random order. Besides, there was a peak in time searching between the 2000 and 4000 dataset sizes for AVL, the tendency of time searching is similar, but BST performed a little better compared to AVL.

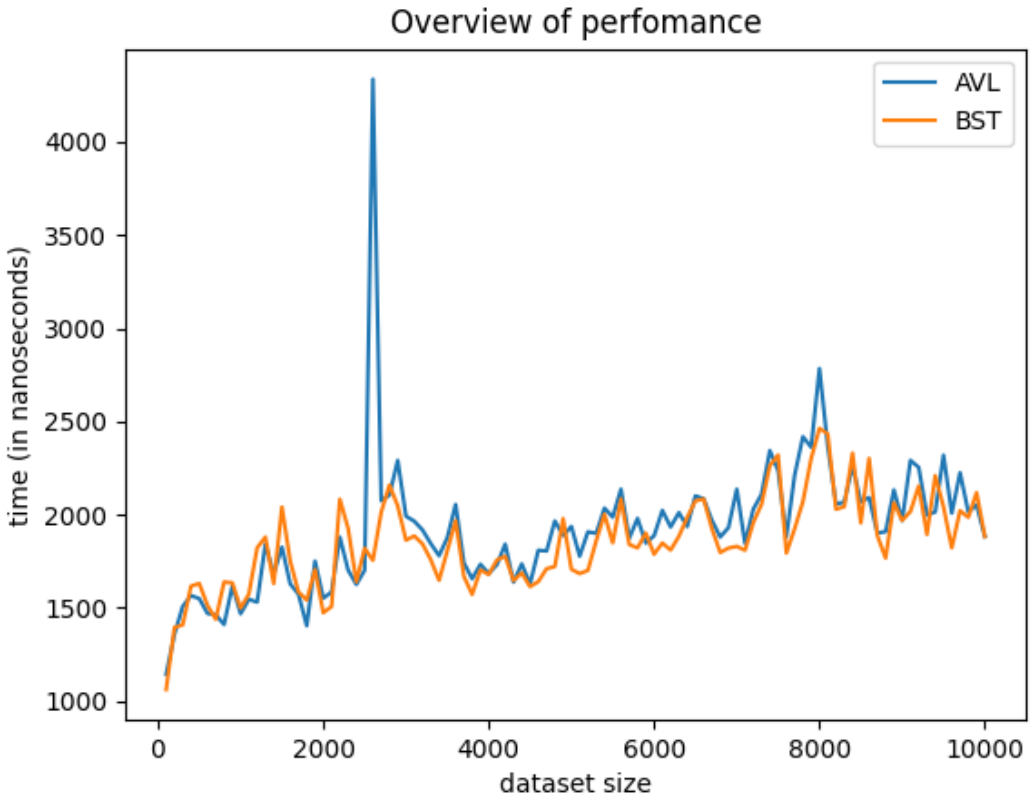

Figure 4. Average time BST and AVL trees took to search for nth element in the range from 0...10000 for random input data

In spite of that, a different pattern between AVL and BST trees can be noticed, when we consider Nearly Ordered / Ordered data input for the trees (Word is stored in the trees). Where searching time in BST grows extremely high in terms of dataset size (see Figure 5,6)

For a dataset that is nearly ordered to search for the element that is stored in trees, AVL is about 2-3 times faster than BST

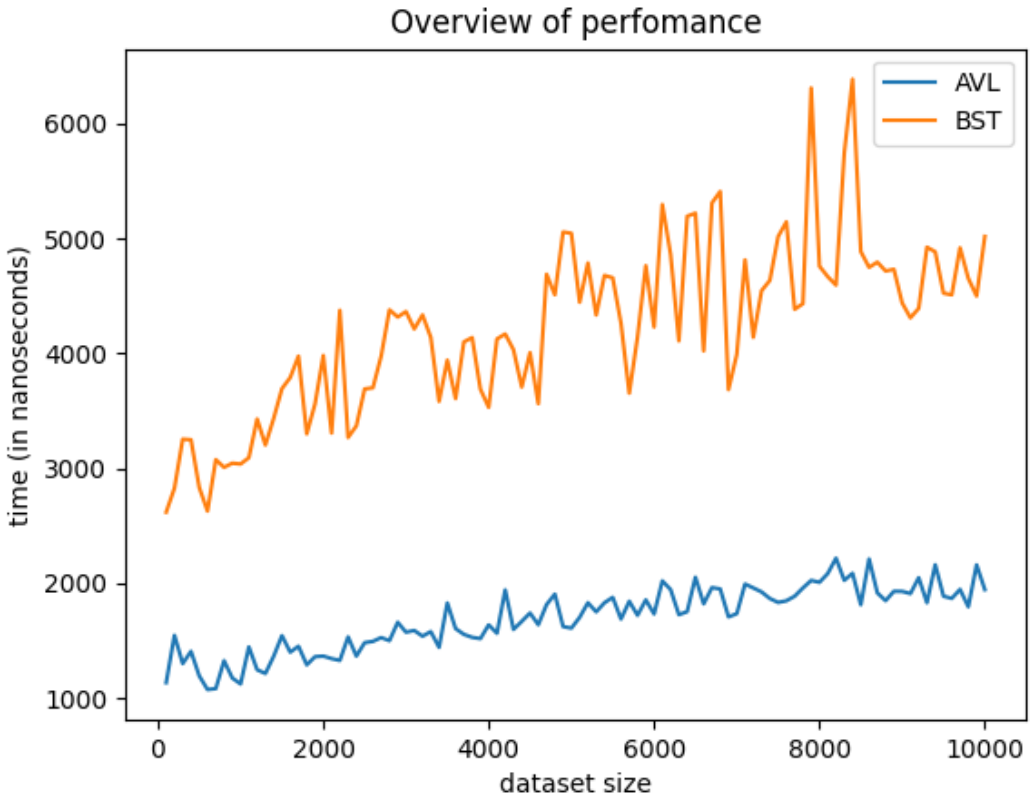

Figure 5. Average time BST and AVL trees took to search for nth element in the range from 0...10000 for nearly ordered data

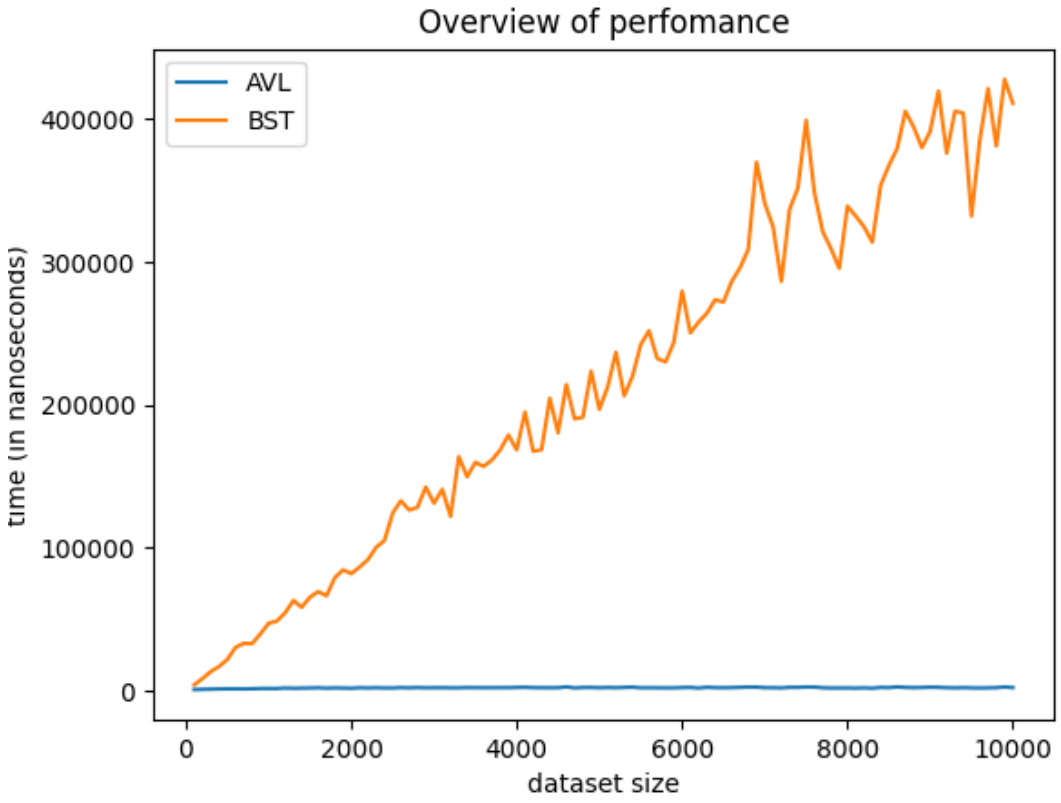

Figure 6. Average time BST and AVL trees took to search for nth element in the range from 0...10000 for ordered data

On the other side, when we investigated the case when we need to search for the word that is not stored in trees, considering the 1-st data type (random data order), in the beginning, the difference of time is small, in the long term BST outgrows in searching time the AVL as illustrated on figure 7.

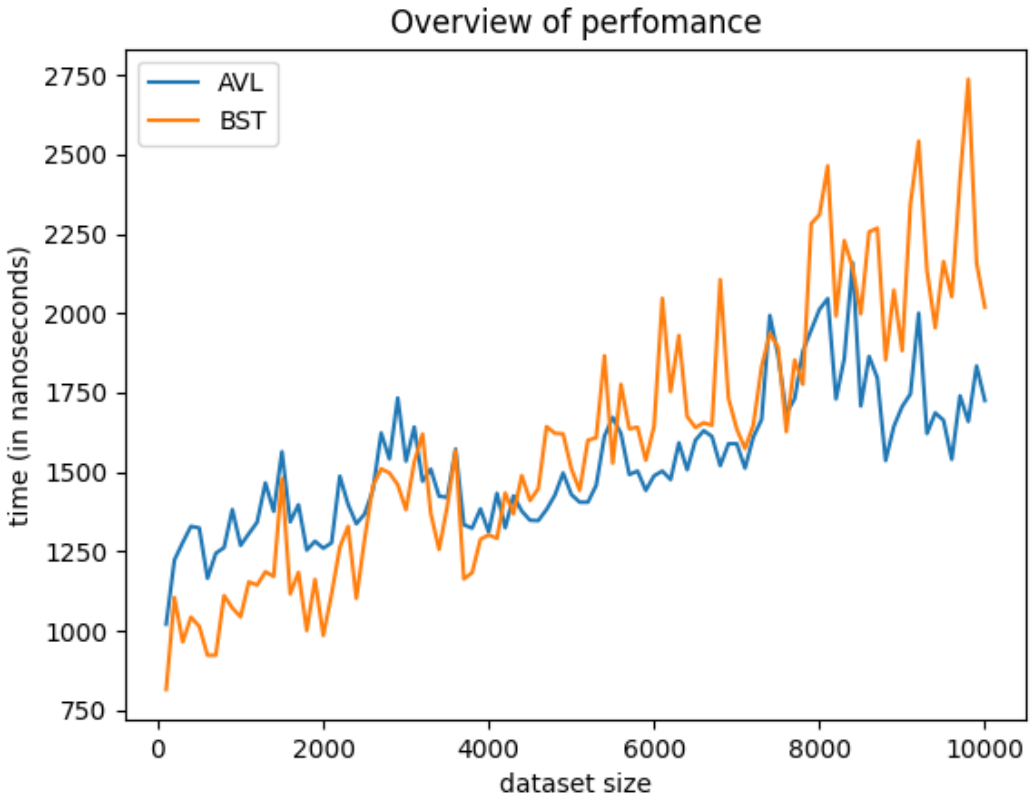

Figure 7. Average time BST and AVL trees took to search for nth element in the range from 0...10000 for random data order. Word is not stored in trees

A similar pattern as in Figures 5 and 6 is revealed in Figure 8,9. Where we see that the time growth is high corresponding to the data set size for BST, whereas AVL is extremely fast in those cases.

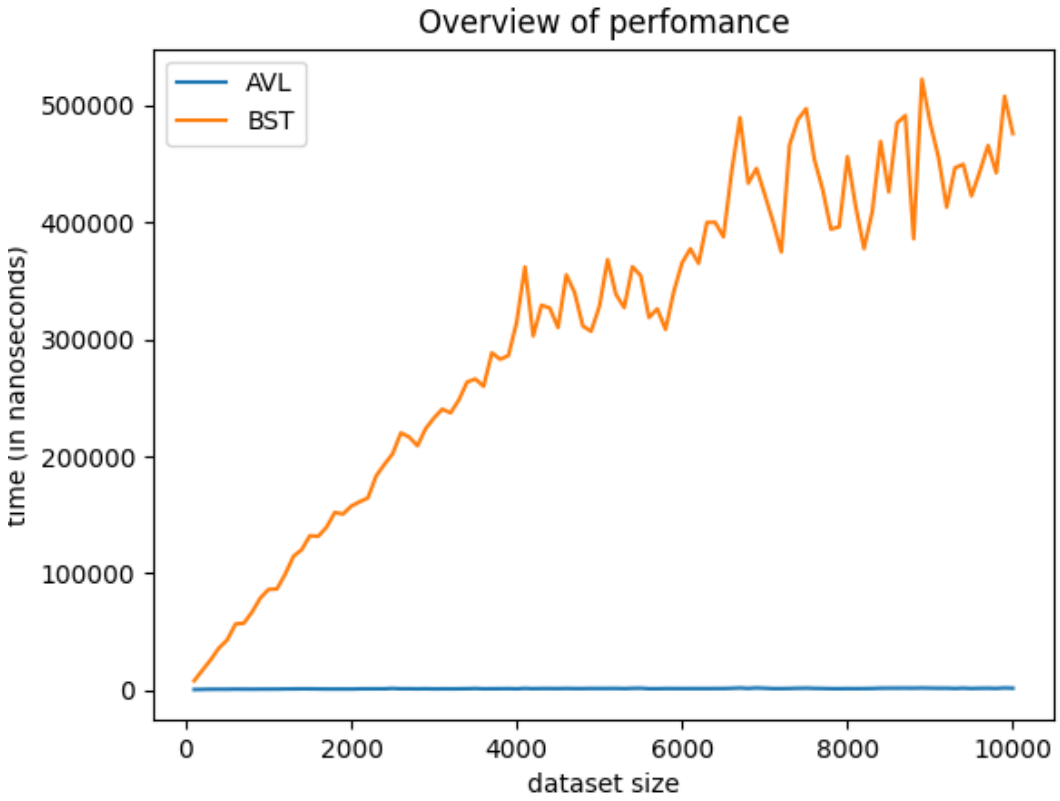

Figure 8. Average time BST and AVL trees took to search for nth element in the range from 0...10000 for ordered data. Word is not stored in trees

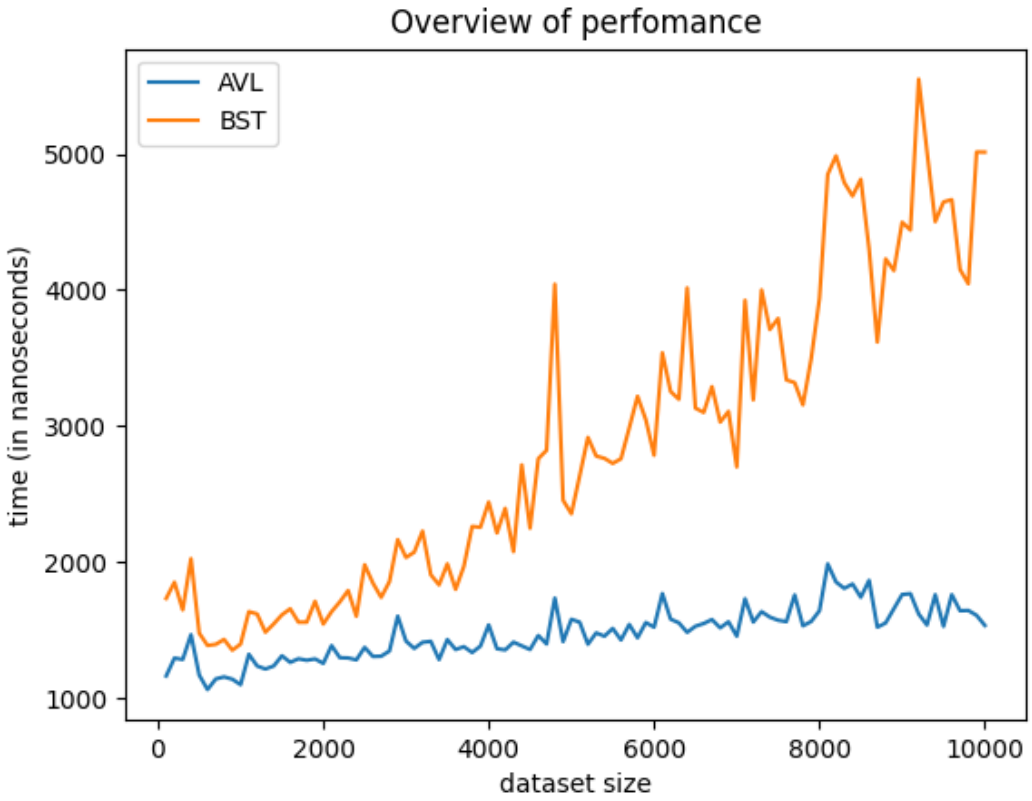

Figure 9. Average time BST and AVL trees took to search for nth element in the range from 0...10000 for nearly ordered data. Word is not stored in trees

4. Conclusions

According to the overview, the AVL tree and BST tree behave in a similar manner and complete operations at the same time when it comes to the random data order in both situations of searching/inserting. However, when the data is nearly (besides insertion, searching is several times slower in BST than in AVL for the nearly sorted datasets) or fully sorted, due to that BST doesn't change its structure dynamically as AVL would perform, BST continuously degrades and slows the process as the dataset grows. Thus, in the case of dictionary data structure, the implementation of the AVL tree suits the best.