**Storing in trees: comparison between Binary Search Tree and AVL Tree**

Siarhei Miachkou

January 06, 2023

1. **Introduction**

In this paper we approach the problem of storing large amounts of data. There is more to it than just finding enough space, inserting new entries, or accessing/removing the old ones can appear to be quite time consuming if we happen to choose an inappropriate data structure. For this reason, it is found practical to use trees for storing a lot of data, since they allow to optimize all the actions mentioned above.

But there are different types of trees, which one should we go with? In this paper we will investigate two of the more popular versions: Binary Search Tree and AVL tree and try to decide when to choose which considering their strengths and weaknesses.

**1.1** **Binary Search Tree (BST)**

Let us start with the description of the Binary Search Tree since it is a base for AVL tree implementation. Binary Search Tree consists of nodes, each node stores a single value and pointers to two other nodes (that is why it is called Binary) left and right. The very first node of the tree is called the root, we put our first value there. Then when it is time to insert the next one, we make an important decision: where to store it? We want to be able to access data as quickly as possible, so we need a set of strict rules describing how we store our values.

These rules go like so: store the smaller values on the left and the bigger values on the right (there are several ways of solving tiebreaks, in our implementation we are going to store similar values to the left of their copies). It might not seem like a lot of help but this way of storing entries is extremely efficient. Every node we check gives us much more information than just found/not found, since we know where the value, we are looking for can be relatively to this node, thus getting rid of half of the remaining nodes every time. And so, we go left/right until we find our value or reach a null pointer indicating the absence of it.

The complexity of this process is O(h), where h is a height (the number of levels) of the tree. But is it good or bad? How big is height of tree comparing to the number of elements? Well, in case of Binary Search Tree it depends on the order in which we insert the values. The best case is h = log(n), but it is hard to achieve (we need to insert the middle value first and then go to the max and min values at the same time interchangeably one by one). The worst case is h = n and it is much easier to get, just insert values in the sorted order, so that it goes always to the left or always to the right (basically a linked list with an extra pointer for every node).

So, the good thing is that the Binary Search Tree is quite easy to implement and maintain, on the other hand we might end up with a giant linked list that will take a tremendous amount of time to insert new elements or find the old ones.

**1.2 AVL Tree**

As we have just mentioned the problem with the Binary Search Tree is that inserting a sorted data results in a terrible linked-list-like structure. And that is why AVL Tree (named after the two of its creators Adelson-Velsky and Evgenii Landis) exists. Its implementation is far more complex, but it allows to solve the problem of h close or equal to n cases. The idea is simple: make sure that the equality h = log(n) always holds. The way we achieve this is by constantly monitoring where our nodes place themselves and rearranging them whenever it is needed. Now let’s address this problem step by step, the first question is how do we know I the latest insertion violates the h = log(n) condition?

There are not one but two ways of controlling this by storing balance factors or heights of each node. Yes, the nodes we use for the Binary Search Tree and the AVL Tree are not the same. The AVL Tree’s node contains one more field (balance factor or height). We will go with the height option since it is much easier to understand, describe and implement. Now each of our nodes stores its individual height (height of the subtree in which this node is a root). This gives us an ability to control how our tree is forming. Every time we insert a new node, we go recursively retrieving by the path it has come checking heights of every node’s children. If the difference between their heights (the balance factor of a parent node) is more than one we have a problem to solve, this brunch is too long comparing to the others, and it ruins out h = log(n) equality that we want to be true so badly. So how do we fix this? It depends on where the problem has occurred.

There are 4 cases: to the left of the left child (left left), to the right of the left child (left right), to the right of the right child (right right) or to the left of the right child (right left). The simpler cases are left left and right right ones because they require just one rotation. And rotating the tree is how we are going to keep its height as low as possible.

There exist two rotation options: left and right. To perform a left rotation, we assign the left subtree of the right child as the right subtree of our imbalanced node, then we assign the imbalanced node subtree to the left subtree of this right child and finally we swap their places. In simpler words we put our node to the left of its right child pulling the whole subtree up and reducing its height. The process of the right rotation is just mirroring the left one, so we are not going to describe it separately.

Now back to solving our problem: in left left case scenario we need a single right rotation performed on the imbalanced node, in right right case scenario – a single left rotation performed on the imbalanced node, in left right case scenario we first need to perform a left rotation on the right child and then the right rotation on the imbalanced node itself, in right left case scenario – a right rotation on the left child and then a left rotation on the imbalanced node.

But how all these rotations affect our insertion’s time complexity (searching in the BST and in the AVL Tree is an identical process)? In fact, even though it does obviously take more time the complexity remains the same: O(h) because performing rotations is guaranteed to be constant time. Thus, our AVL Tree ensures that the h = log(n) and still has the same complexity as the Binary Search Tree.

1. **Methodology**

Both data structures were implemented in C++. The scenario of storing a dictionary was imitated using the .txt file containing 394400 English word sorted alphabetically. Three cases were tested for both trees: random, partly sorted and sorted. For the partly sorted and random cases were used arrays of 300.000 words. To achieve a partly sorted array 1.500 random swaps were made, to achieve a random array 150.000 random swaps were made. For the sorted case were used an array of 20.000 words, since the bigger number of words resulted in overflow. The insertion times were acquired by measuring the time it takes to enter 100th, 200th, 300th, …, 20.000th(300.000th) element, and the search times were acquired by measuring the time it takes to find the 100th, 200th, 300th, ..., 20.000th(300.000th) element and an element absent in the tree. Each measurement was performed 100 times and the average time was recorded as the result.

1. **Results**

As it was expected when the data comes sorted the Binary Search Tree is slower by a huge margin compared to the AVL tree in both insertion (Figure 1) and searching (Figure 2). With the partly sorted data the situation improves a little but still the AVL tree is much faster than Binary Search Tree (Figures 3, 4). But when it comes to the randomly ordered data things become more interesting: now insertion into the Binary Search Tree is faster because the height is likely to be far from being equal to the number of elements and no rotations are performed saving some time (Figure 5). However, when it comes to insertion AVL Tree is still faster, no rotations are performed in this case and the height of AVL Tree is likely to be lower (Figure 6).

1. **Conclusions**

By running the tests, we concluded that it is more efficient to use the AVL Tree if the data comes sorted or even partly sorted. When the data is organized randomly then the Binary Search Tree is most likely the way to go, unless there will be much more searching than insertion, than it is might be better to use the AVL Tree since its searching time is still mostly better even of the data was inserted in the random order.

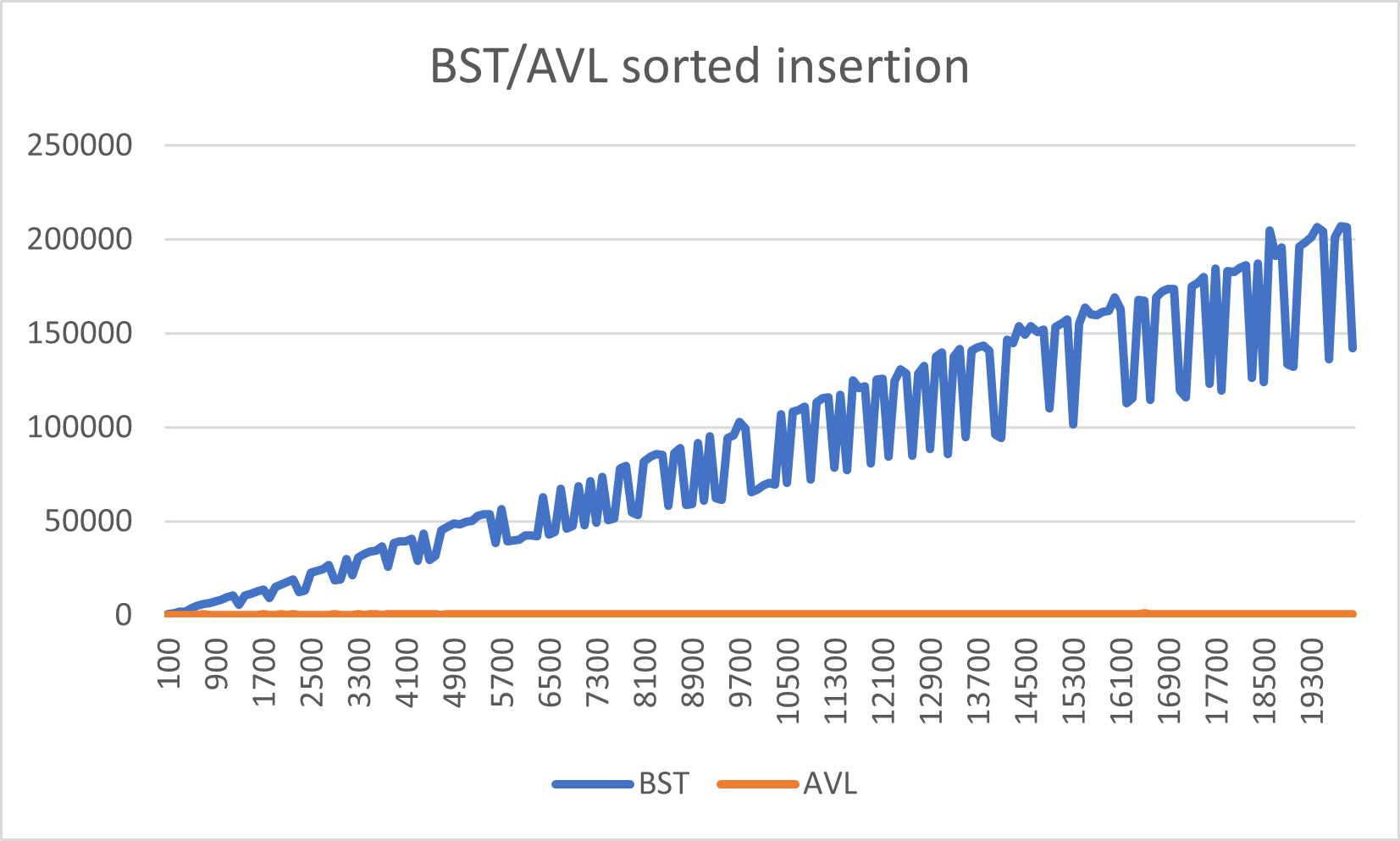


Figure 1: Inserting sorted data into the BST and AVL Tree

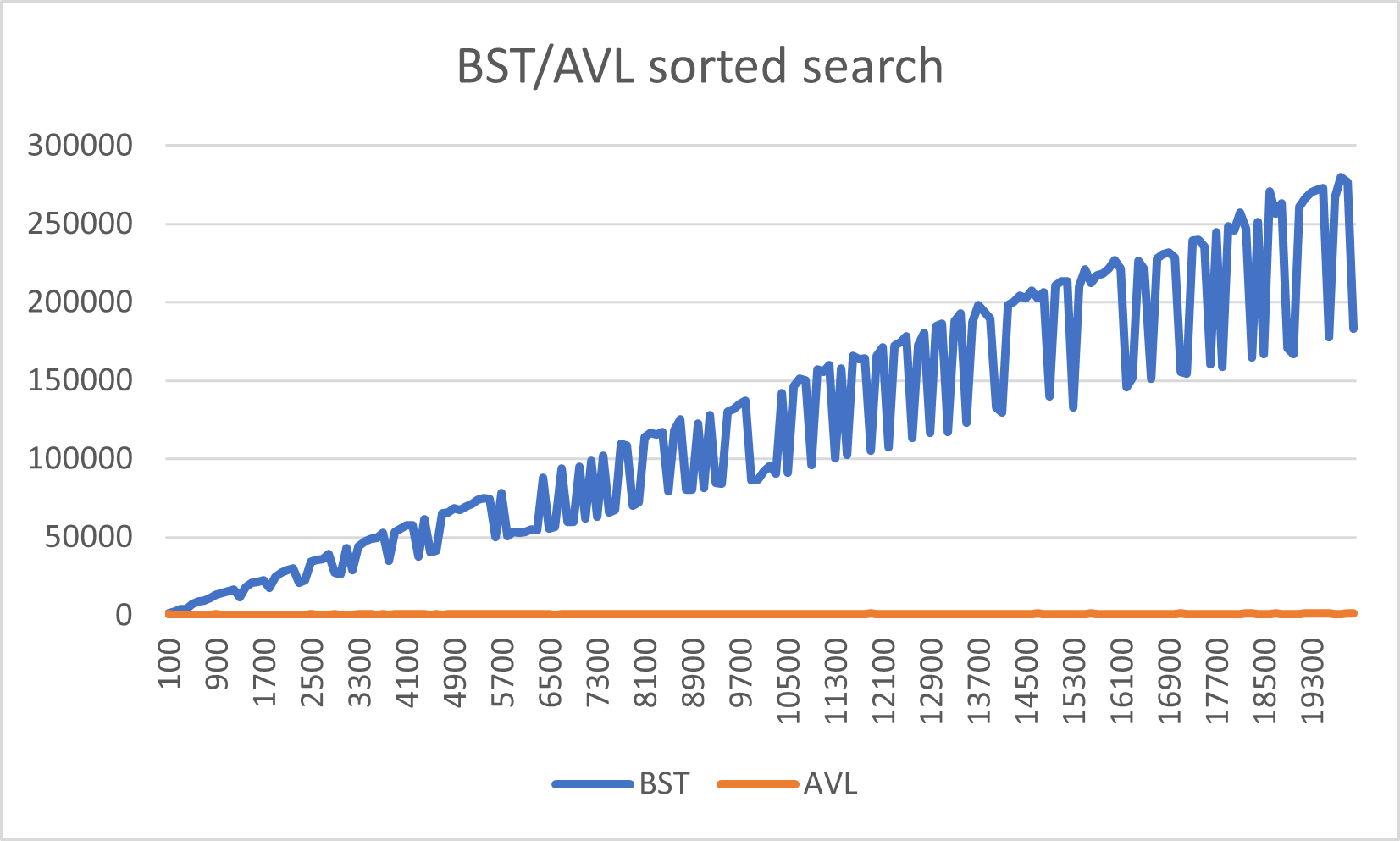


Figure 2: Searching for sorted data in the BST and AVL Tree

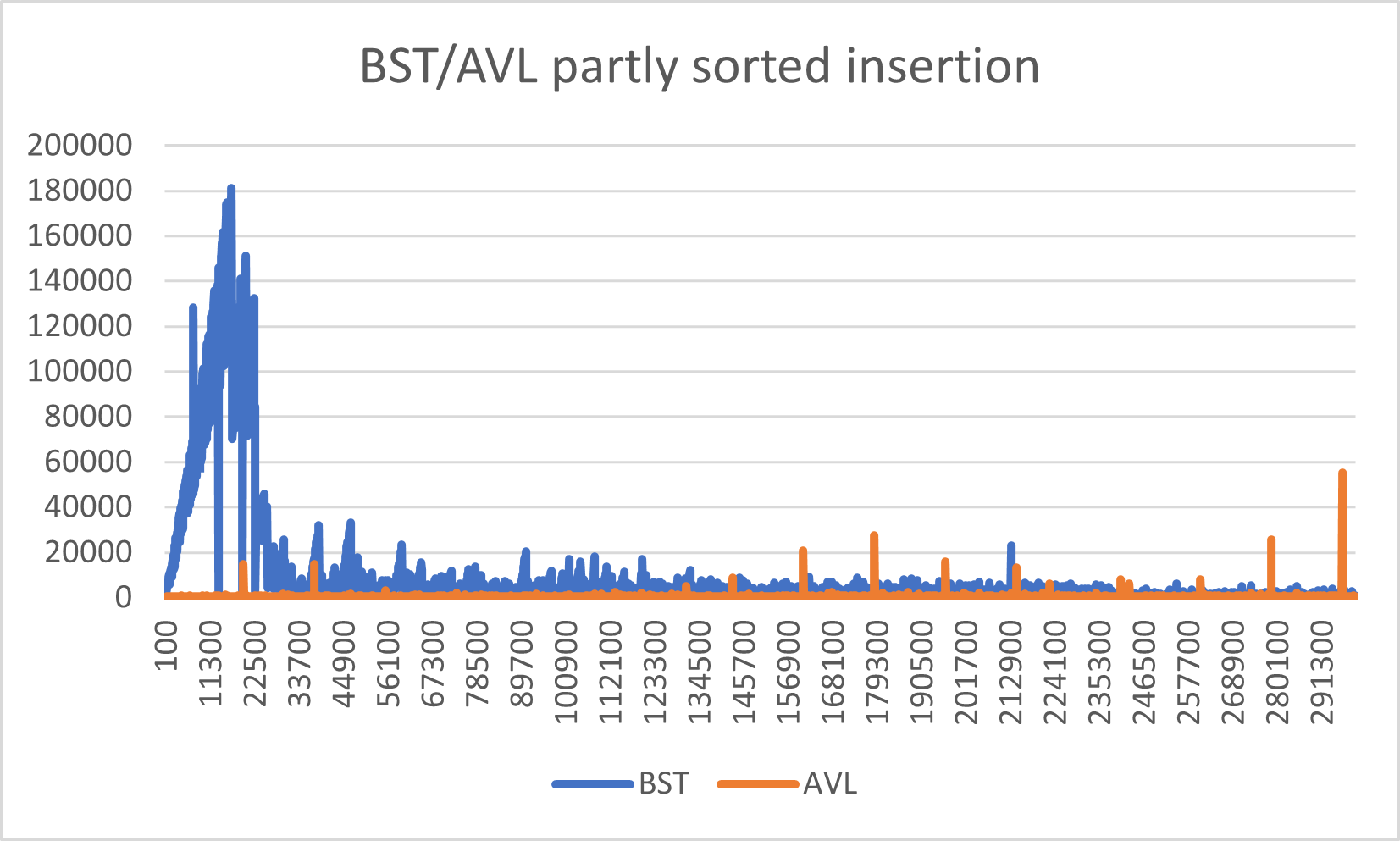


Figure 3: Inserting partly sorted data into the BST and AVL Tree

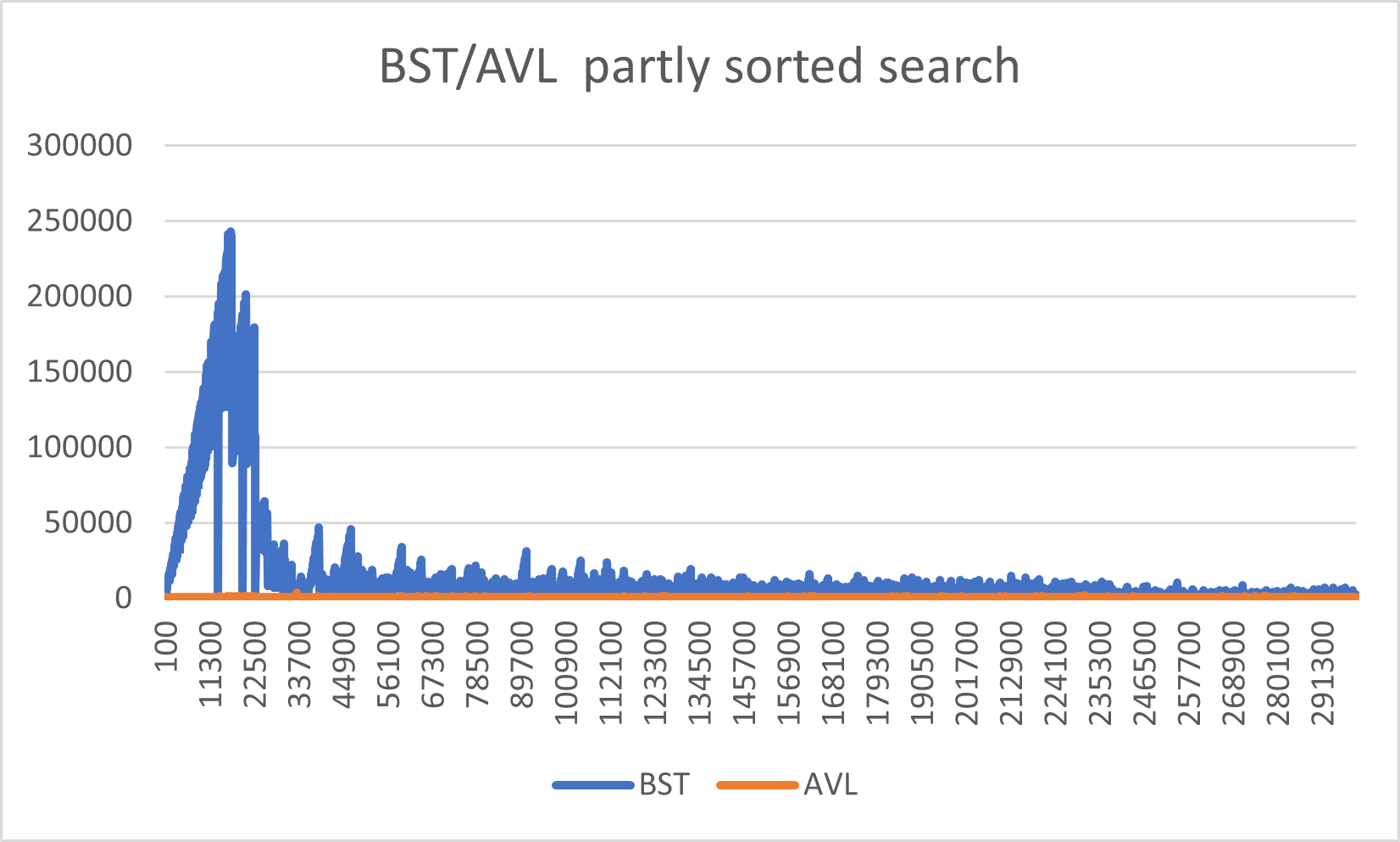


Figure 4: Searching for partly sorted data in the BST and AVL Tree

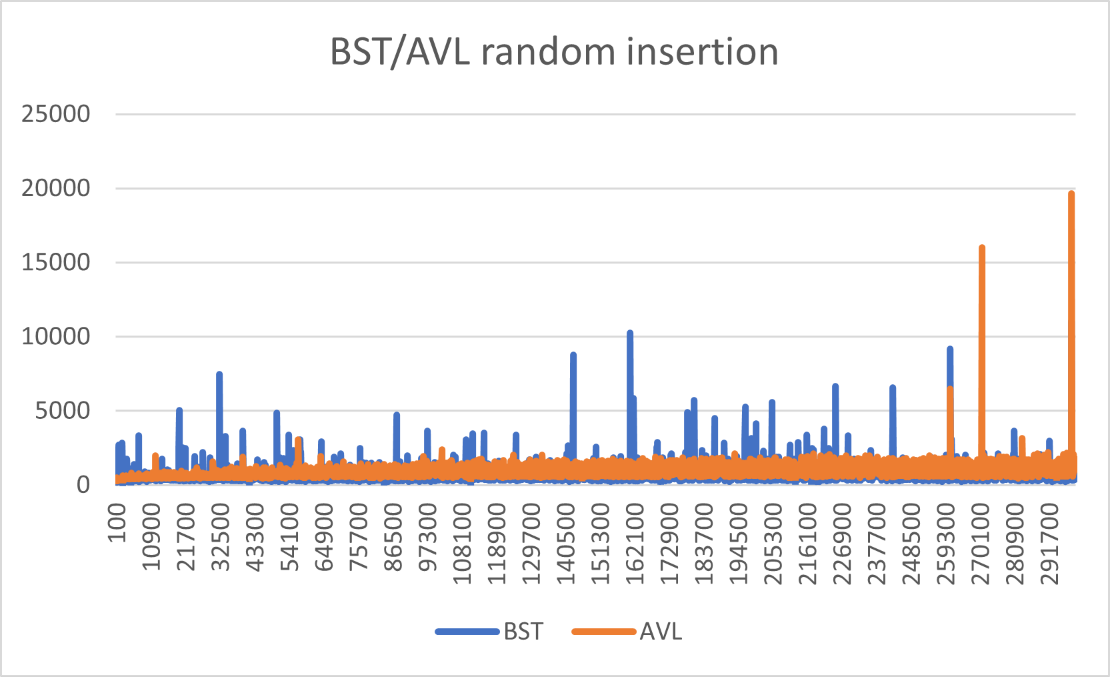


Figure 5: Inserting randomly ordered data into the BST and AVL Tree

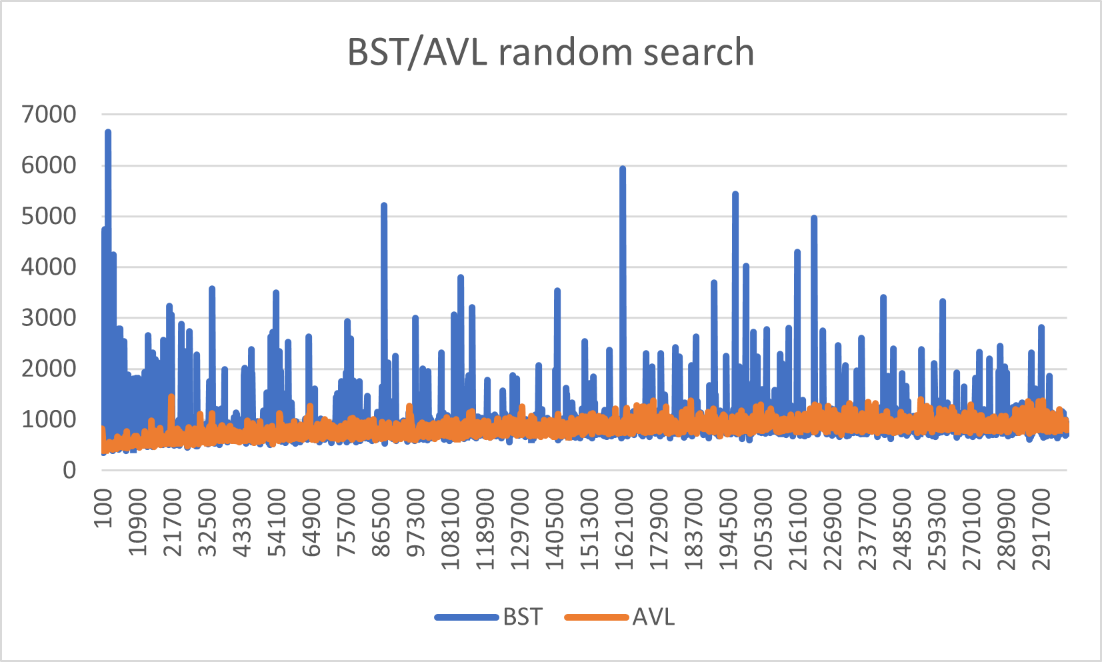


Figure 6: Searching for randomly ordered data in the BST and AVL Tree