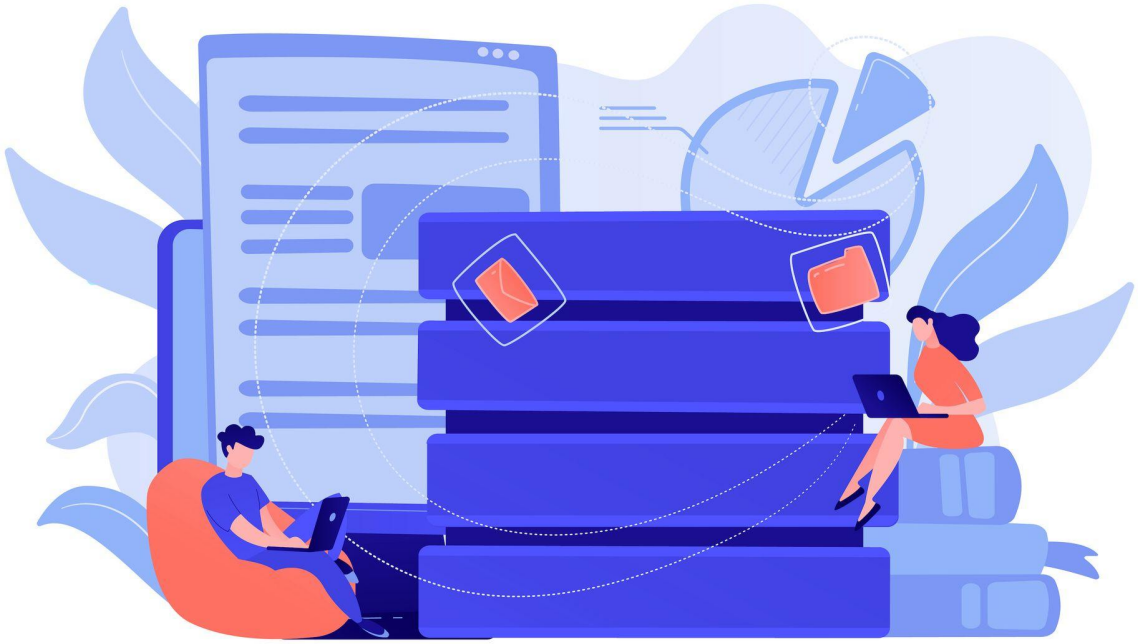


Binary search tree vs AVL tree

Insertion and search time comparison on dictionary of words

Nurbek Shukan

18.12.23

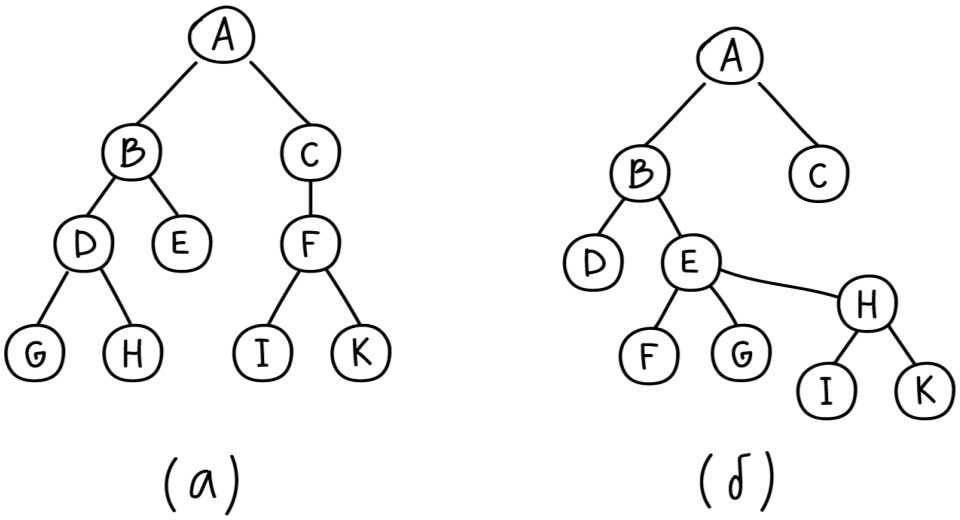
Introduction

Organizing data storage in the form of a tree allows you to bypass the limitations of a linear data structure. For example, in the latter it is impossible to organize quick search and insertion of elements at the same time. At the same time, large amounts of data can be efficiently selected and updated in a hierarchical data structure.

In this paper, we will make comparison between two popular version of trees, **Binary Searh Tree** and **AVL tree**. The problem is to arrange a dictionary of *n* words according to some total order which is computed in O(1) and compare their insertion and search time. We use < as the total order.

Binary trees

Before getting into BST and AVL trees, let's first make some introduction to binary trees. One of the most commonly used and easiest to implement subtypes of trees is binary trees. In addition to organizing searches, binary trees are used when parsing mathematical expressions and computer programs. They are also used to store data for compression algorithms, and they also underlie other data structures, such as priority queues, heaps, and dictionaries. A **binary tree** is a tree in which each of its nodes has at most two child nodes. Moreover, each child node is also a binary tree. Consider the example treesin the following figure:



Tree (a) is binary. Each of its nodes has at most two child nodes, each of which also has at most two children. For example, nodes E, G, H, I, and K are **leaf nodes**, meaning they have zero child nodes. Node C has only one child node, while nodes A, B, D and F each have two children.

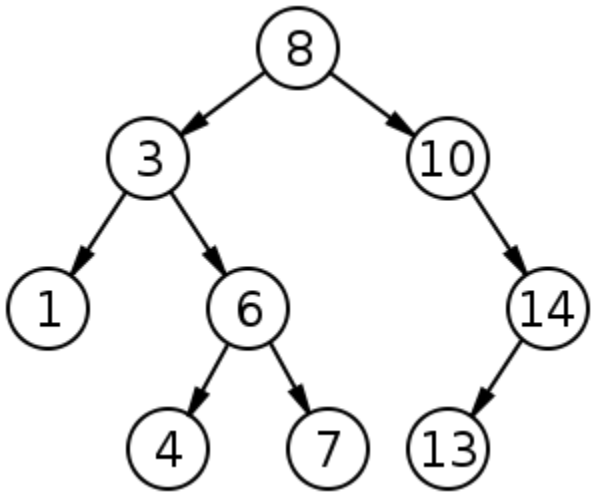
As soon as the rule of two children is violated, the tree ceases to belong to the class of binary. Thus, tree (b) is not binary, since node E has three child nodes.

Due to the fact that there are never more than two child nodes, they are called **right and left child nodes.**

Binary Search Tree

Binary search trees differ from regular binary trees in that they store data in sorted form. Storing values inside a binary search tree is organized as follows:

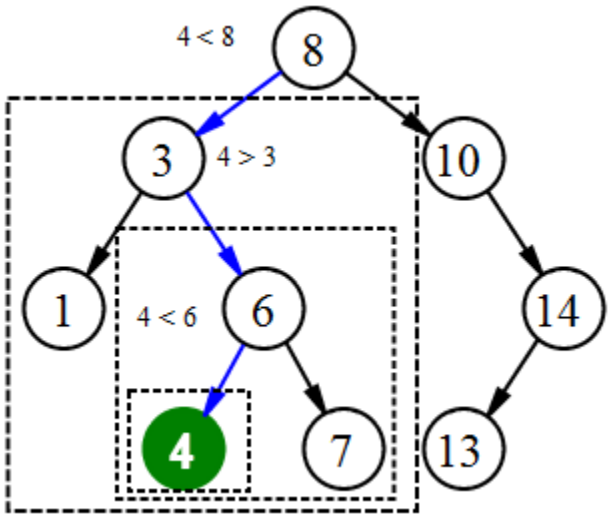
- All values in the nodes of the left child subtree are less than the value of the parent node
- All values in the nodes of the right child subtree are greater than the value of the parent node
- Each child node is also a binary search tree



Thanks to this data storage structure, searching for a node in a binary search tree takes **average case** $O(logn)$. This is significantly less if you store values in lists - $O(n)$.

However, If you use a sorted array to store data, the speed of searching for elements will be equal, which is **worst case** $O(n)$. But when evaluating the insertion time, storing in an array is significantly inferior to working with trees - versus, respectively.

Insertion. All values less than the current node value should be placed in the left subtree, and larger values in the right subtree. To insert a new node, you need to check that the current node is not empty. Then there can be two ways: If so, compare the value with the inserted one. Based on the comparison result, we check for the right or left subtrees, If the node is empty, create a new one and fill in the link to the current node as a parent. In my case, The implementation of the insertion function involves the use of two pointers, providing an efficient means to navigate the tree and determine the optimal position for a new node. The function begins by initializing a double pointer, which points to the root of the tree. Through a loop, the algorithm traverses the tree, comparing the value to be inserted with the values of each node. Based on the comparisons, the double pointer is adjusted to either the left or right child of the current node. This iterative process continues until an empty position is found, at which point a new node with the specified value is created and linked to the tree. The **time complexity** is $O(h)$, where **h is the height of the tree**.



Search. If the search value of a binary search tree is less than the value of a node, then it can only be in the left subtree. The search value that is greater than the node value can only be in the right subtree. The time complexity is the same as in Insertion, is **linearly dependent** on the **height** of the tree.

AVL tree

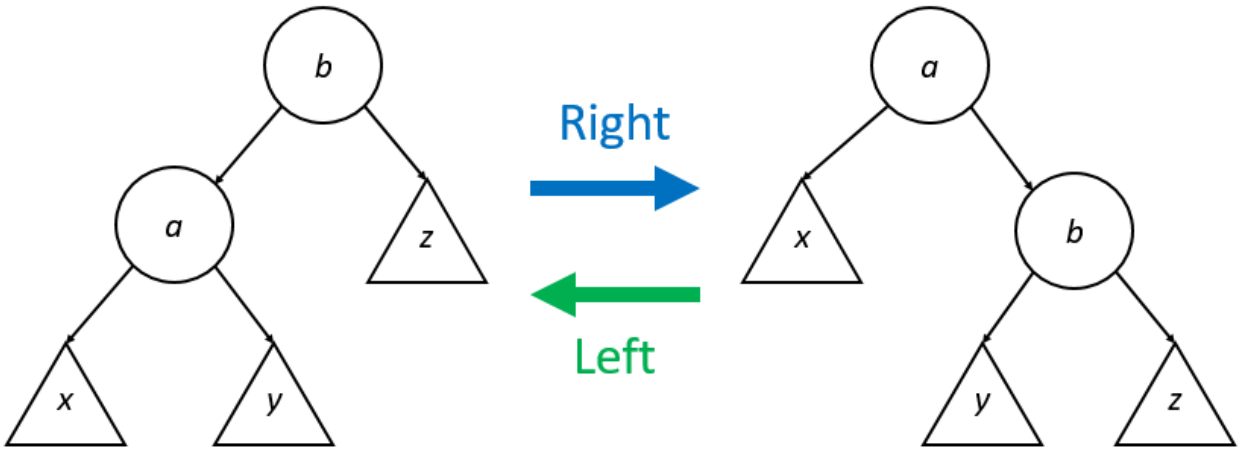
a modified class of trees that have all the advantages of binary search trees and never degenerate. These are called balanced or AVL trees. By balance we mean that for each node of a tree the heights of both its subtrees differ by no more than 1. Strictly speaking, this criterion should be called AVL-balanced in contrast to ideal balance, when for each node of the tree the number of nodes in the left and right subtrees differ by no more than 1. New insertion and deletion methods in the AVL tree class ensure that all nodes remain balanced in height.

Theorem. An AVL tree with n keys has height $h = O(logn)$.

Insertion. Inserting a new key into an AVL tree is done in the same way as it is done in simple binary search trees: we go down the tree, choosing the right or left direction of movement depending on the result of comparing the key in the current node and the inserted key. The only difference is that when returning from inserted value (that is, after the key has been inserted into either the right or left subtree and that tree is balanced), the current node is rebalanced. It is strictly proven that the imbalance arising from such an insertion in any node along the path of movement does not exceed two, which means that the application of the above-described balancing function is correct. If we climbed to node i from the left subtree, then balance increases by one, if from the right, then it decreases by one or if with heights, increases by one. If you arrive at a node and its balance becomes zero, this means the height of the subtree has not changed and the ascent stops. If you arrive at a vertex and its balance becomes equal to 1 or −1, then this means the height of the subtree has changed and the ascent continues. If we reach the top and its balance becomes equal to 2 or −2, then we do one of four rotations and, if after the rotation the balance becomes equal to zero, then we stop, otherwise we continue to rise.

Rotation. **AVL Rotations** can be done in two directions: **right** or **left**. Below is a diagram generalizing both right and left **AVL Rotations**. In the diagram, the triangles represent arbitrary subtrees of any shape: they can be empty, small, large, etc. The circles are the

"important" nodes upon which we are performing the rotation.



There are four possible cases in AVL tree rotations: **left-left**, **left-right**, **right-left**, and **right-right**. The choice of which case to apply depends on the balance factors of the node where the AVL tree property was violated and the balance factor of the subtree where the new node was inserted.

In the **left-left** case, the balance factor in a node X is -2, and the balance factor in the left child is -1. To fix the tree, a right rotation is performed at node X.

The **right-right** case is the mirror image of the left-left case: the balance factor in node X is +2, and the balance factor in the right child is +1. To correct the tree, a left rotation is performed at node X.

In a **left-right** case, the balance factor in node X is -2, and the balance factor of the left subtree is +1. In this scenario, the left child of X is first left-rotated, followed by a right rotation at 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 fix the tree, a right rotation is performed on the right child of X, followed by a left rotation at X.

Search. In AVL tree search function is done the same way as in BST.

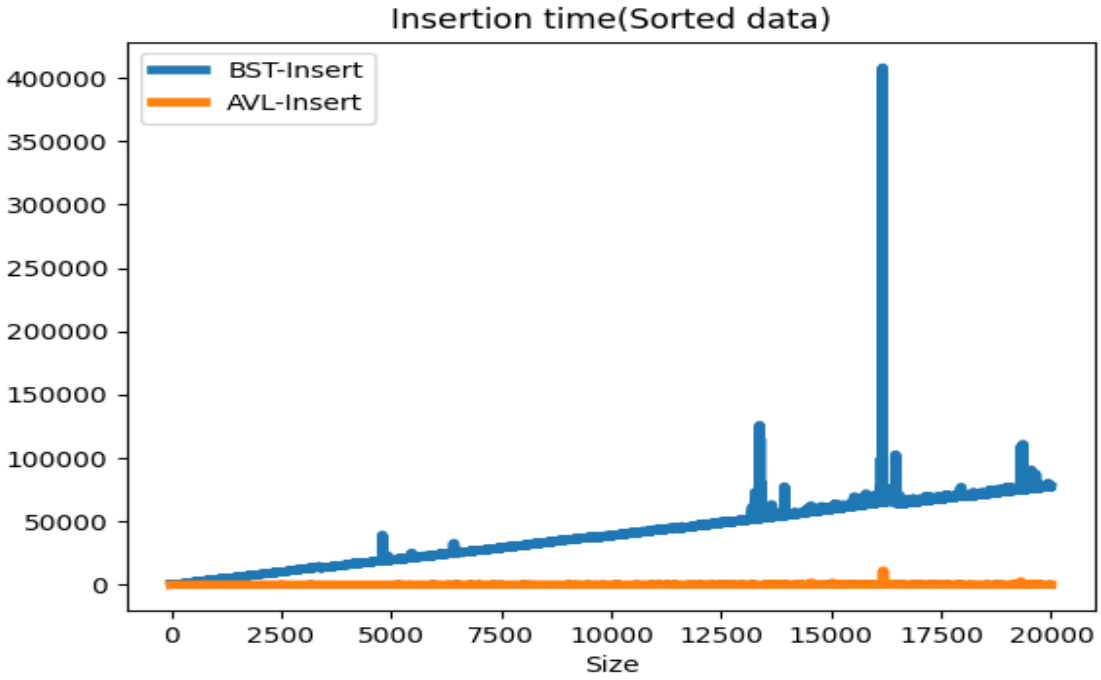
Methodology

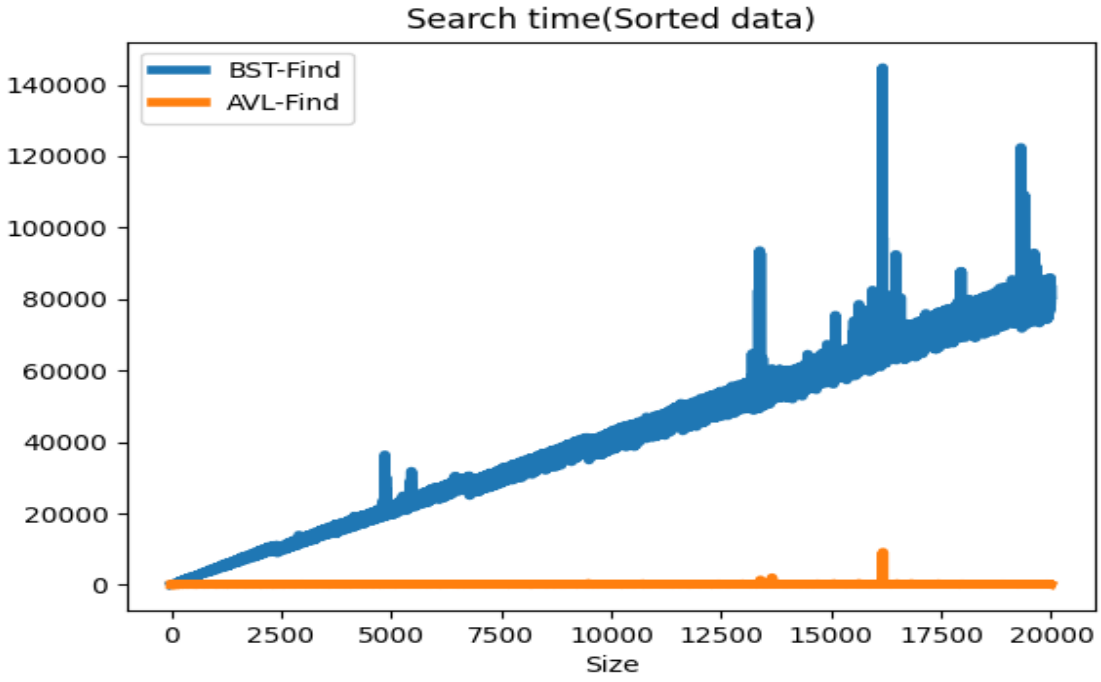
The C++ implementation of the algorithms produced results across three distinct categories:

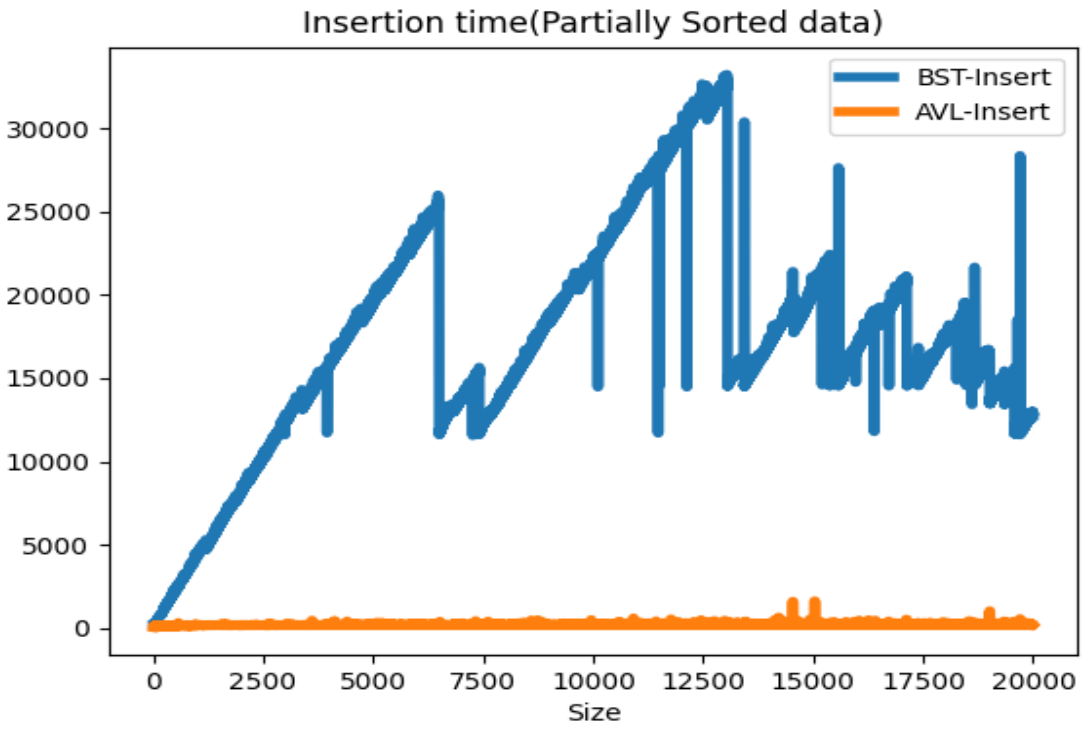
1) Randomly shuffled dictionary words
2) Sorted dictionary
3) Partially sorted dictionary

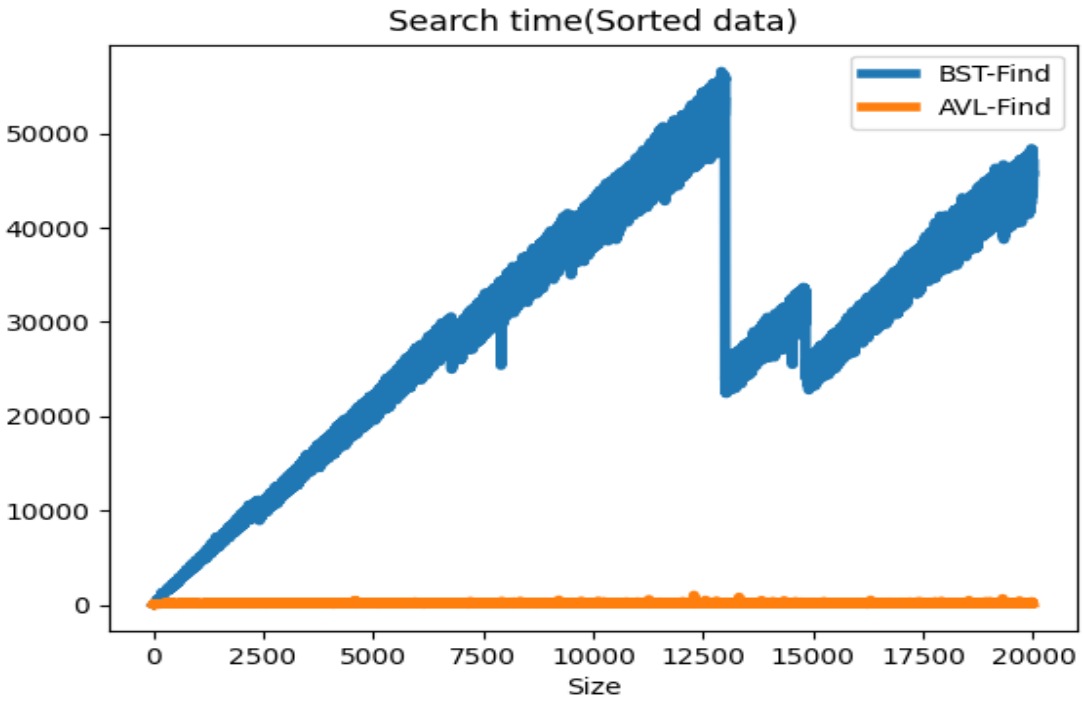
The algorithms underwent testing on arrays of varying sizes: 10, 20, 100, 200, 1000, 2000, ..., 20000. The same dataset was applied to each algorithm across all thre categories. Each array size underwent testing 100 times. The reported result for each size is the average time taken by each tree to insert and search the corresponding input word.

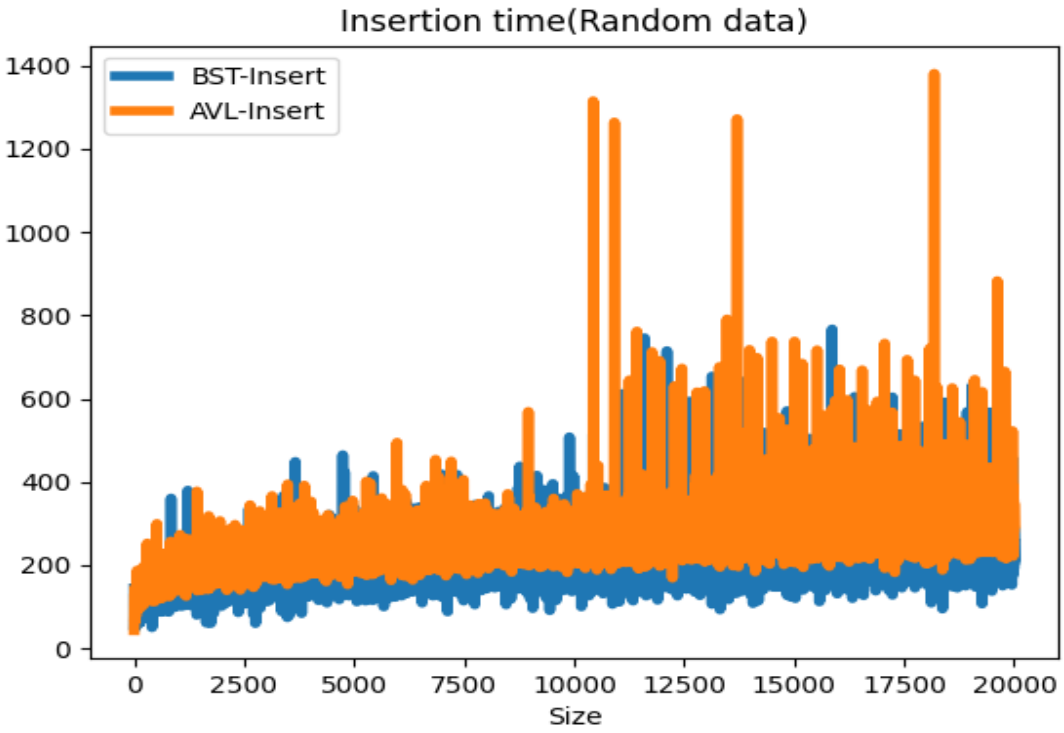
Results

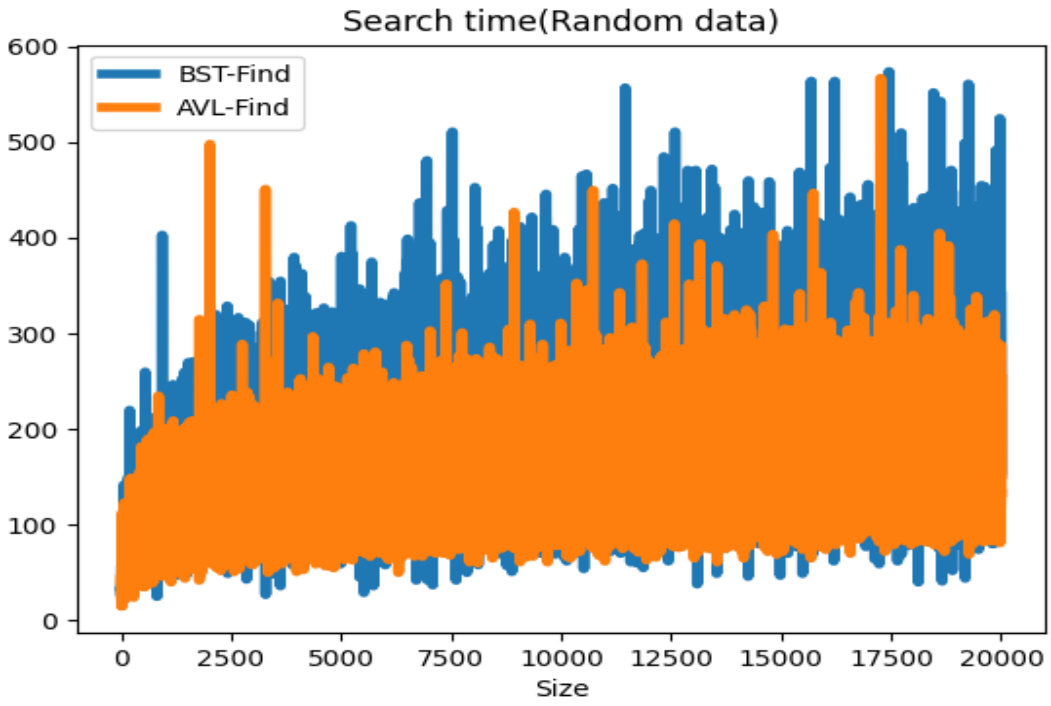

Insertion time(Sorted data)
BST-Insert
AVL-Insert
400000
350000
300000
250000
200000
150000
100000
50000
0
0
2500
5000
7500
10000
12500
15000
17500
20000
Size


Search time(Sorted data)
BST-Find
AVL-Find
140000
120000
100000
80000
60000
40000
20000
0
0
2500
5000
7500
10000
12500
15000
17500
20000
Size


Insertion time(Partially Sorted data)
BST-Insert
AVL-Insert
30000
25000
20000
15000
10000
5000
0
0
2500
5000
7500
10000
12500
15000
17500
20000
Size


Search time(Sorted data)
BST-Find
AVL-Find
50000
40000
30000
20000
10000
0
0
2500
5000
7500
10000
12500
15000
17500
20000
Size


Insertion time(Random data)
BST-Insert
AVL-Insert
1400
1200
1000
800
600
400
200
0
0
2500
5000
7500
10000
12500
15000
17500
20000
Size


Search time(Random data)
BST-Find
AVL-Find
600
500
400
300
200
100
0
0
2500
5000
7500
10000
12500
15000
17500
20000
Size

Conclusions

In our study, we checked how well Binary Search Trees (BST) and AVL Trees handle inserting and finding data. We found that BST works fine for random data, but it struggles with fully sorted or mostly sorted data. However, when the data is random, BST can be okay and even have some advantages.

To make things easier, imagine organizing a list of words. If the words are already in order, like in a dictionary, BST doesn't shine as much. But if the words are mixed up, using BST can be handy. AVL Trees, on the other hand, are better at keeping things balanced, which is helpful for bigger datasets and scenarios where balance matters.

In the real world, whether to use BST or AVL Trees depends on the type of data you're working with. If it's mostly sorted, BST might not be the best choice. But if it's random or a bit messy, BST could have some benefits.

To sum it up, our study gives us a closer look at how BST and AVL Trees handle different situations. Looking ahead, we can use the adaptability of BST and the efficiency of AVL Trees to create smart solutions that fit the different kinds of data we encounter.