Data Structures & Applications
Assignment 02

Sayanta Adhikari
AI22MTECH12005

October 18, 2023

1 Comparision of BST and AVL Tree operations

1.1 Binary Search Tree (BST)

A Binary Search Tree is a binary tree in which each node has at most two child nodes, usually referred to as the left child and the right child. It has the following key properties:

1. **Ordering Property:** For any given node, all elements in its left subtree are smaller than the node's value, and all elements in its right subtree are larger.
2. **Unique Values:** Each node in a BST contains a unique element.
3. **Efficient Searching:** Due to its ordering property, searching for an element in a BST can be done efficiently in $O(\log n)$ time on average, where n is the number of elements. In the worst case, when the tree is highly unbalanced, searching can take $O(n)$ time.
4. **Inefficient Insertion/Deletion:** While searching is efficient, insertion and deletion operations can lead to tree imbalance in the worst case, making them inefficient.

1.2 AVL Tree

An AVL (Adelson-Velsky and Landis) Tree is a self-balancing binary search tree. It's a specific type of binary tree that maintains its balance using the concept of "heights" or "balances" at each node. It has the following key properties:

1. **Height-Balanced:** In an AVL tree, the height of any node's left and right subtrees (called the balance factor) can differ by at most one. In other words, the tree is "balanced" in terms of its height.
2. **Guaranteed Logarithmic Height:** Due to its self-balancing property, the height of an AVL tree is guaranteed to be $O(\log n)$ in the worst case, where n is the number of elements. This ensures that search, insertion, and deletion operations have consistent, efficient time complexity.
3. **Balancing Operations:** To maintain balance, AVL trees employ balancing operations, such as rotations, during insertion and deletion. These operations adjust the structure of the tree to ensure that it remains height-balanced.
4. **Efficient Operations:** AVL trees provide consistent $O(\log n)$ time complexity for searching, insertion, and deletion operations in both average and worst-case scenarios.

We are going to experimentally show the difference between heights and computations done in BST and AVL Tree.

1.3 Setup

We have a 100 such sequence of 1000 unique random numbers. We sequentially insert them in BST or AVL Tree as per our requirements. Then we choose at random 100 keys without replacement and delete them sequentially. Throughout this process, we compute the number of comparisons and pointer operations done. Along with that, we also compare the heights of these trees.

1.4 Results & Remarks

1.4.1 Heights

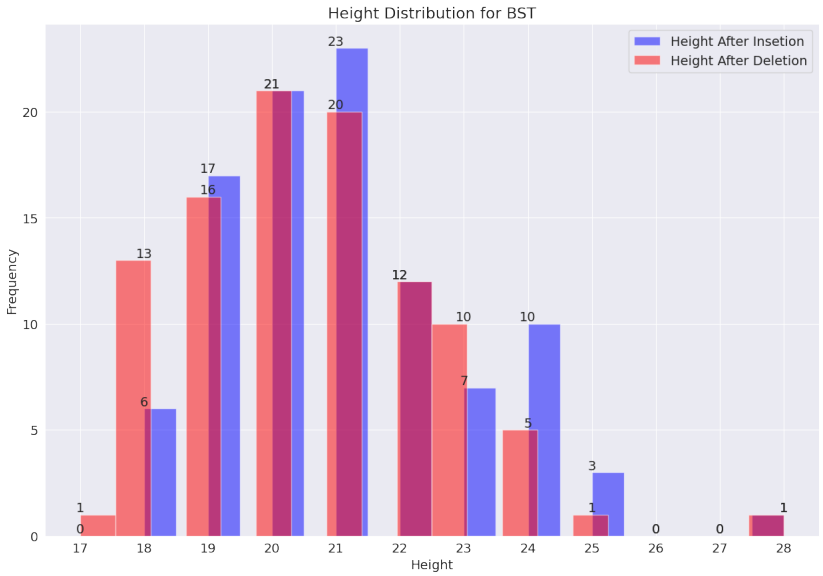


Figure 1: Height Distribution of BST. Blue: Distribution after inserting all 1000 keys Red: Distribution after inserting all 1000 keys and then deleting 100 keys.

Figure 2: Height Distribution of AVL Tree. Blue: Distribution after inserting all 1000 keys Red: Distribution after inserting all 1000 keys and then deleting 100 keys.

From the Plots 1 & 2, we can see that as AVL trees have the balancing property, the heights for all

the 100 trees are the same, whereas in the case of BST, depending on the order of insertion we have a wide variety of different heights.

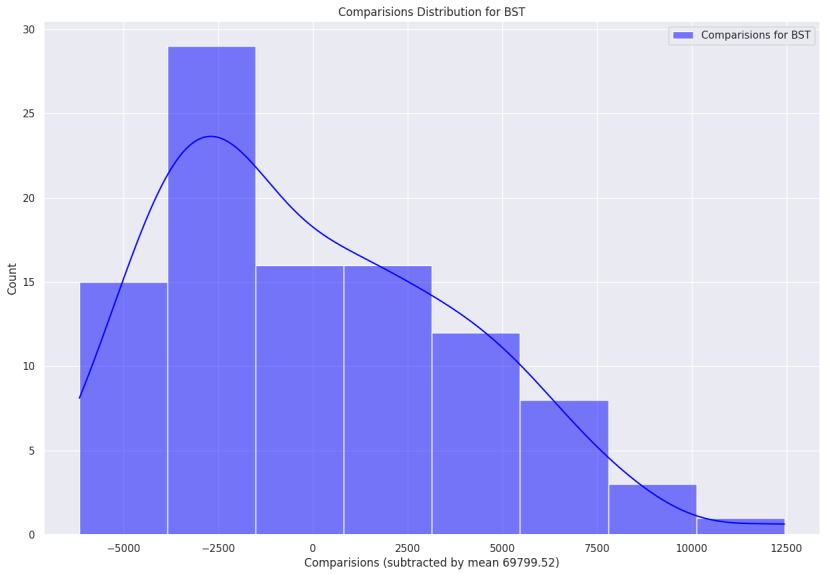
For AVL trees, the height is after inserting 1000 keys:

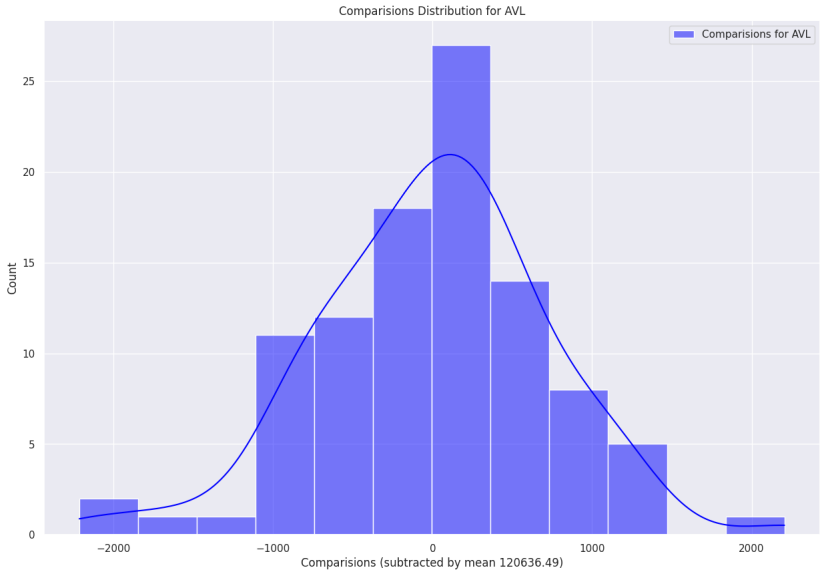
$$h = 1 + \lceil \log_2(1000) \rceil = 11$$

From experiments also, we are getting the same theoretical results.

Another observation is that, as we know, in a tree of 1000 nodes, 500 of them will be on the leaf. Removing 100 keys will, at max, reduce the height by one in the case of balanced trees (AVL), whereas that's not the case in BST. So we see a wide difference in the distribution in the case of BST but not in the case of AVL Tree.

1.4.2 Comparisions

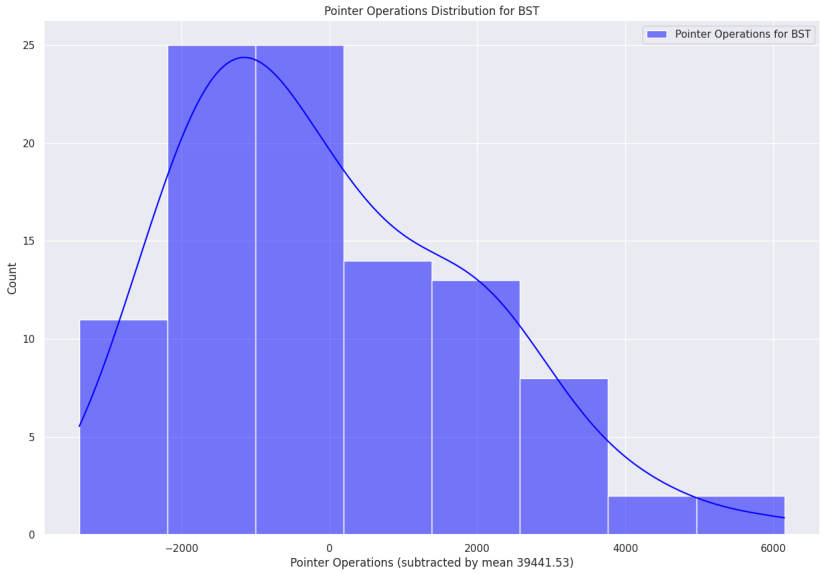


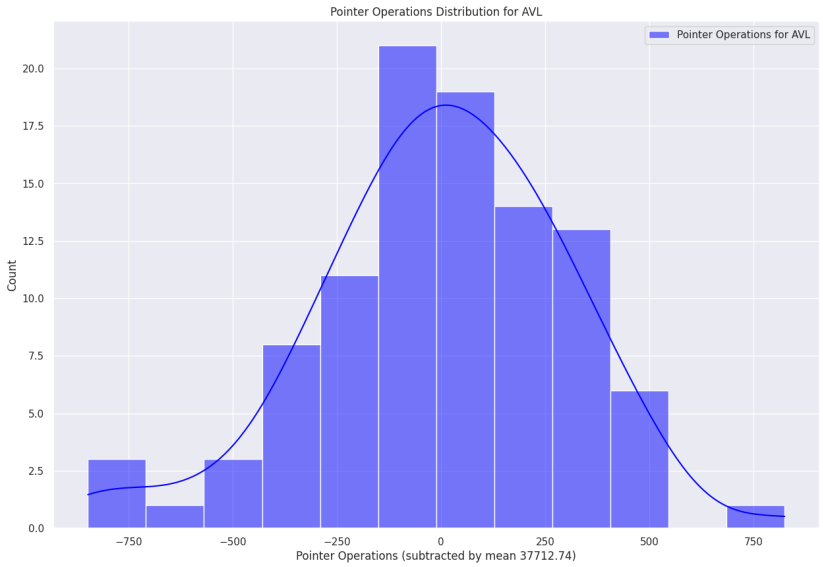


From the figures 1.4.2 & 1.4.2 we can see that the number of comparisons required in AVL Tree is way more than that of a BST. Mean comparisons required in AVL Tree is 120636 and in BST is 39442. The reason for this is AVl Tree does the insertion and deletion the same way as BST does but after inserting or deleting it also does a *Balancing* which requires some extra number of comparisons.

On calculating the variance for comparisons, BST has a standard deviation of 3855, whereas AVL tree has a standard deviation of 716. This also points to the variety of heights that a BST can take whereas AVL tree is always balanced so the number of comparisons required remains closer to the mean in AVL tree compared to BST.

1.4.3 Pointer Operations





From the figure 1.4.3 & 1.4.3 we can see that the number of pointer operations are more or less equal for both cases. Mean Pointer operations for BST is 39442 whereas mean pointer operations for AVL is 37713. So, both BST and AVL tree requires equivalent number of pointer operations, or even BST take more number of pointer operations than AVL tree despite AVL tree having rotaions involved. Standard deviations for both BST and AVL tree are 1954 and 297 respectively. Standard deviation for AVL tree is also less which is also because the balancing keeps the tree balanced at every step so the amount of pointer operations required are less as we are ensuring it at every insertion.