# Relative Analysis of AVL Trees and Binary Search Trees

1. Short Description of the Task

The task involves enforcing two types of tone- balancing double hunt trees an AVL tree and a  BST

The performance of these two data structures is also compared by fitting and searching for rudiments.

2. Short Description of the Data Structures and Algorithms

AVL Tree An AVL tree is a tone- balancing double hunt tree where the difference between the heights of the left and right subtrees of any knot is lower than or equal to one.

still, rebalancing is done to restore this property, If at any time they differ by further than one.

Rebalancing is achieved through reels, specifically, right and left reels.

BST is a tree data structure in which each knot has at most two children, appertained to as the left child and the right child.

For each knot, all rudiments in the left subtree are lower than the knot, and all rudiments in the right subtree are lesser.

3. Methodology Used to Compare the Data Structures and Algorithms

The comparison is done by measuring the time taken to fit and search for rudiments in both data structures.

Three types of data are used for the test arbitrary data, ordered data, and nearly ordered data.

The chrono library is used to measure the time taken for these operations

4. Results

The results are the time taken by AVL and BST to fit and search for rudiments.

These results are displayed in the press using cout.

5. Conclusions

The conclusions are grounded on the results of the time taken by both data structures.

Generally, AVL trees are briskly in searching rudiments than BSTs because they're more balanced, but take a bit further time in insertion due to the reels done for balancing the tree.

still, the exact performance can vary depending on the nature of the input data.

# Detailed Explanation of the Code

The law is divided into several sections, each performing a specific task in the comparison of AVL trees and BSTs.

• knot Structures The law begins by defining the structures for the bumps of the AVL tree and the BST. Each knot contains a key( or data), pointers to the left and right child bumps, and in the case of the AVL tree, a height trait.

• Insertion Functions : The insert function for both AVL and BST checks if the knot is null( in which case a new knot is created), or if the key to be fitted is lower than or lesser than the current knot's key. Depending on the comparison, the function is called recursively on the left or right child.

• Balancing in AVL Tree In the AVL tree, after insertion, the balance of the knot difference in height between the left and right subtrees) is checked. However, reels are performed to rebalance the tree, If the balance factor is further than 1 or lower than-1.

• Gyration Functions : The rightRotate and leftRotate functions perform reels on the bumps of the AVL tree. A right gyration is done when the left child's height is

lesser than the right child's, and a left gyration is done when the right child's height is lesser than the left child's.

• Data Generation Functions : The generateRandomData, generateOrderedData, and generateNearlyOrderedData functions induce different types of data for testing the performance of the AVL tree and BST. The arbitrary data is generated by opting arbitrary words from a predefined list, the ordered data is a sorted list of words, and the nearly ordered data is a list that's substantially sorted but has some rudiments shifted.

• Performance Testing Function : The testPerformance function measures the time taken to fit and search for rudiments in both the AVL tree and BST. It uses the chrono library to get high- resolution time ahead and after the insert and search operations, and also calculates the duration. This function is called for each type of data( arbitrary, ordered, and nearly ordered).

• Main Function : The main function initializes the data, generates the different types of data, and calls the testPerformance function to perform the tests. It also prints out the results of the performance tests.