AVL TREES

(Self Balancing Binary Search Trees)

❖ *What are AVL trees?*

- ➢ In computer science, an AVL tree (**Georgy Adelson-Velsky** and **Evgenii Landis'** tree, named after the inventors) is a **self-balancing binary search tree**.
- ➢ It was the first such data structure to be invented.
- ➢ 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.
- ➢ **Search, Insertion,** and **Deletion** 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.

❖ ***Operations***

- ➢ **Search** - Searching a node.
- ➢ **Insertion** - Inserting a node at the right place such that the the tree is balanced.
- ➢ **Deletion -** Deleting a node such that the tree is balanced.

❖ ***Upper Bound of AVL Tree Height***

We can show that an AVL tree with n nodes has O(log n) height. Let N_h represent the minimum number of nodes that can form an AVL tree of height h. If we know N_{h-1} and N_{h-2}, we can determine N_h . Since this N_h -noded tree must have a height h, the root must have a child that has height h − 1. To minimize the total number of nodes in this tree, we would have this subtree contain N_{h-1} nodes. By the property of an AVL tree, if one child has height h − 1, the minimum height of the other child is h − 2. By creating a tree with a root whose left subtree has N_{h-1} nodes and whose right subtree has N_{h-2} nodes, we have constructed the AVL tree of height h with the least nodes possible. This AVL tree has a total of N_{h-1}+ N_{h-2} +1 nodes (N_{h-1} and N_{h-2} coming from the sub-trees at the children of the root, the 1 coming from the root itself).

The base cases are $N_1 = 1$ and $N_2 = 2$. From here, we can iteratively construct N_h by using the fact that $N_h = N_{h-1} + N_{h-2} + 1$ that we figured out above.

Using this formula, we can then reduce as such:

$$\begin{aligned}
N_h &= N_{h-1} + N_{h-2} + 1 \\
N_{h-1} &= N_{h-2} + N_{h-3} + 1 \\
N_h &= (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1 \\
N_h &> 2N_{h-2} \\
N_h &> 2^{\frac{h}{2}} \\
\log N_h &> \log 2^{\frac{h}{2}} \\
2 \log N_h &> h \\
h &= O(\log N_h)
\end{aligned}$$

Insert

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property **(left.node <root.node <right.node).**

1) Left Rotation
2) Right Rotation

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z.

Delete

Steps to follow for deletion.

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property **(left.node <root.node <right.node).**
1) Left Rotation
2) Right Rotation

Let w be the node to be deleted
1) Perform standard BST delete for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z.

Search

Steps to follow for searching

The search operation in an AVL tree is exactly same to a normal BST.
Compare the given key with the root node(let root node be w).

1) If key is equal to w, then w is the desired node.
2) If key is less than w, then go to step 1 with **w = w.left**.
3) If key is greater than w, then go to step 1 with **w = w.right**.

Colour Code

Let k be the key to be inserted, searched or deleted. In an arbitrary step, let it be compared with a node w in existing AVL Tree.

1) If k<w , w is highlighted in green.
2) If k>w , w is highlighted in red.
3) k is highlighted in blue.
4) While checking the balance of the subtree nodes will blink in purple.
5) Arrows are in Darkcyan.
6) Default node colour is black.

Modules Required

1) WxPython
2) Selenium
3) GraphViz

Team Details

1)	Annam Venkata Krishna	CS14B003
2)	Sri Harsha Arangi	CS14B027
3)	Abinash Patra	CS14B032