AVL Trees

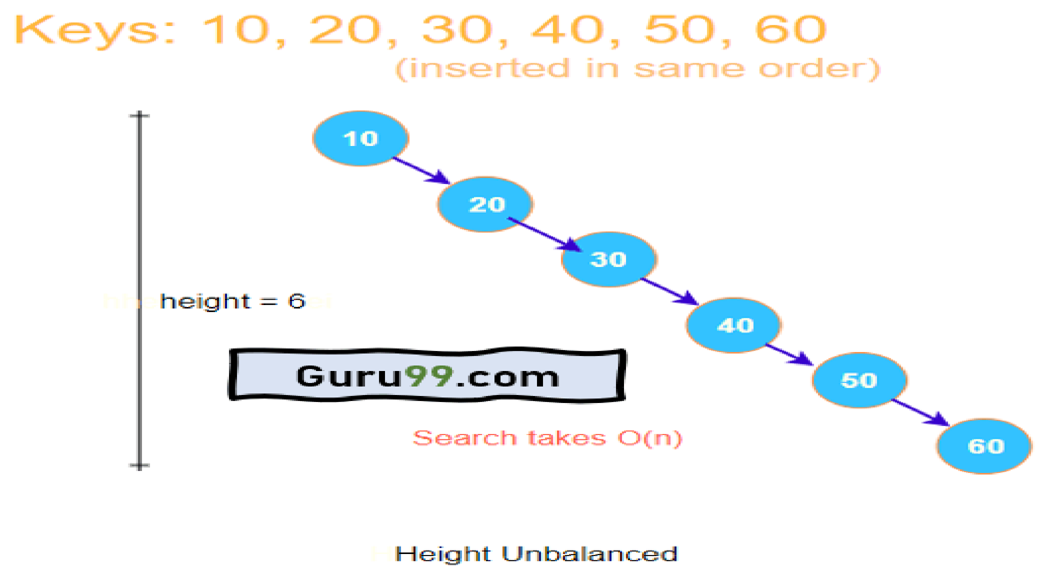
AVL trees are binary search trees in which the difference between the height of the left and right subtree is either -1, 0, or +1.

AVL trees are also called a self-balancing binary search tree. These trees help to maintain the logarithmic search time. It is named after its inventors (AVL) Adelson, Velsky, and Landis.

How does AVL Tree work?

To better understand the need for AVL trees, let us look at some disadvantages of simple binary search trees.

Consider the following keys inserted in the given order in the binary search tree.

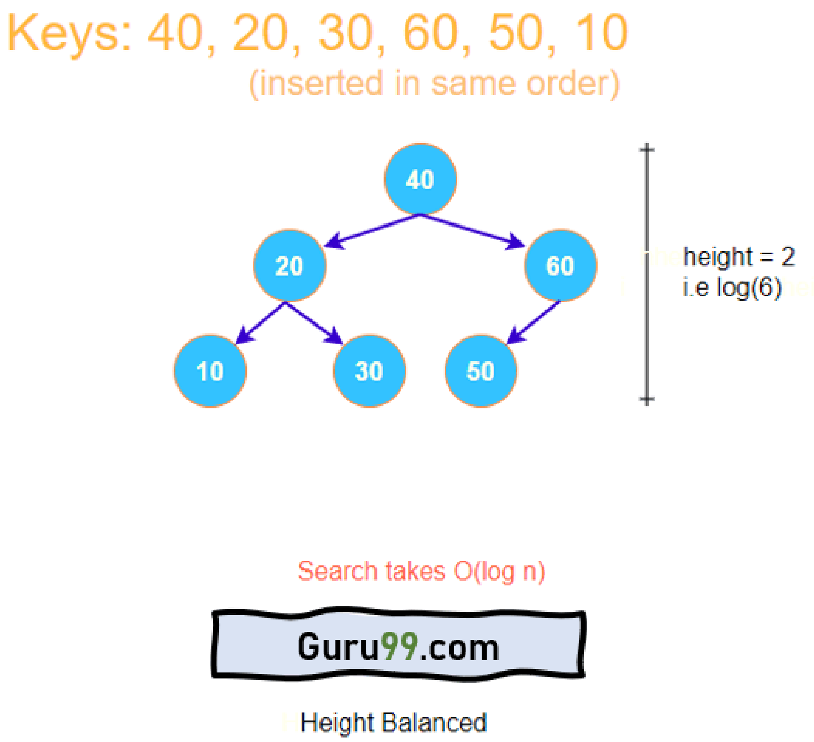


AVL tree visualization

The height of the tree grows linearly in size when we insert the keys in increasing order of their value. Thus, the search operation, at worst, takes O(n).

It takes linear time to search for an element; hence there is no use of using the Binary Search Tree structure. On the other hand, if the height of the tree is balanced, we get better searching time.

Let us now look at the same keys but inserted in a different order.



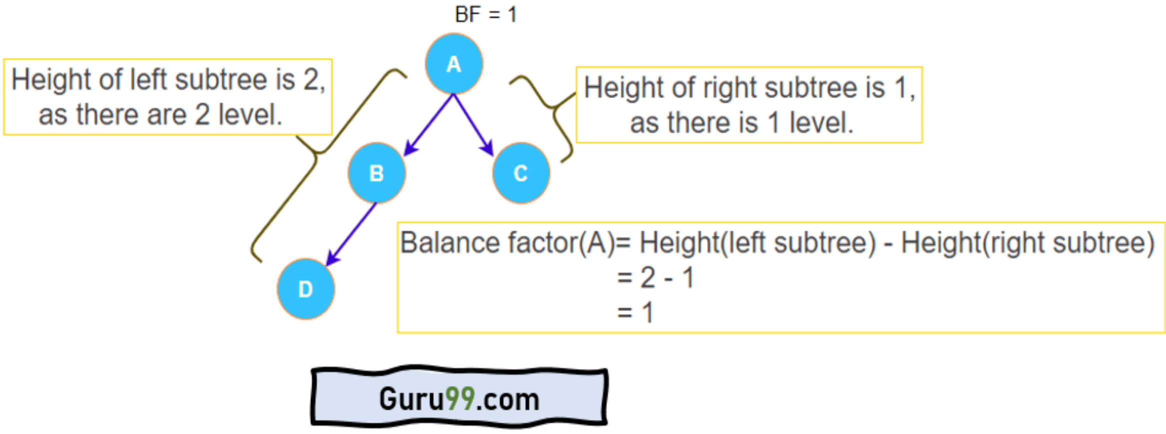
Here, the keys are the same, but since they are inserted in a different order, they take different positions, and the height of the tree remains balanced. Hence search will not take more than O(log n) for any element of the tree. It is now evident that if insertion is done correctly, the tree's height can be kept balanced.

In AVL trees, we keep a check on the height of the tree during insertion operation. Modifications are made to maintain the balanced height without violating the fundamental properties of Binary Search Tree.

Balance Factor in AVL Trees

Balance factor (BF) is a fundamental attribute of every node in AVL trees that helps to monitor the tree's height.

Properties of Balance Factor



Balance factor AVL tree

- The balance factor is known as the difference between the height of the left subtree and the right subtree.
- Balance factor(node) = height(node->left) - height(node->right)
- Allowed values of BF are –1, 0, and +1.
- The value –1 indicates that the left sub-tree contains one extra, i.e., the tree is left heavy.
- The value +1 indicates that the left sub-tree contains one extra, i.e., the tree is left heavy.
- The value 0 shows that the tree includes equal nodes on each side, i.e., the tree is perfectly balanced.

AVL Rotations

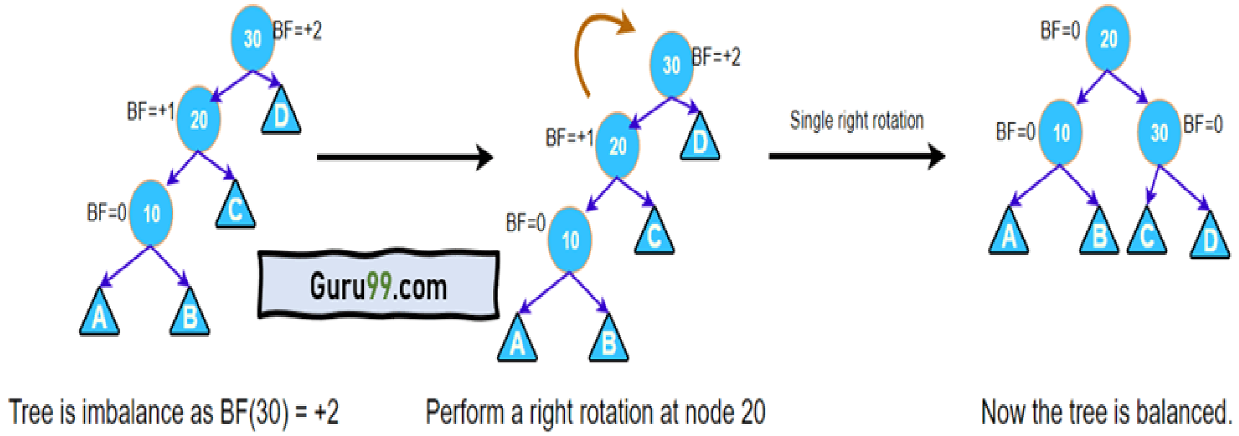
To make the AVL Tree balance itself, when inserting or deleting a node from the tree, rotations are performed.

We perform the following LL rotation, RR rotation, LR rotation, and RL rotation.

- Left – Left Rotation
- Right – Right Rotation
- Right – Left Rotation
- Left – Right Rotation

Left – Left Rotation

This rotation is performed when a new node is inserted at the left child of the left subtree.

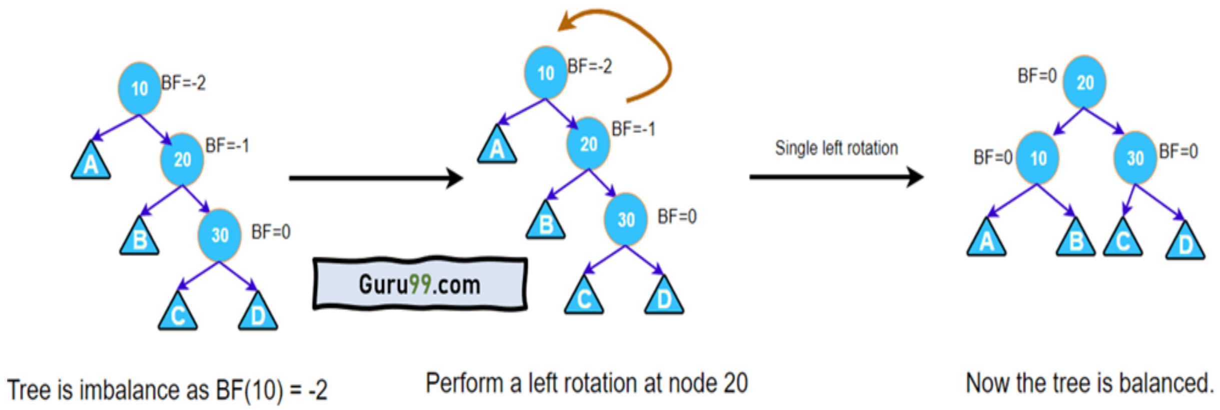


AVL Tree Left – Left Rotation

A single right rotation is performed. This type of rotation is identified when a node has a balanced factor as +2, and its left-child has a balance factor as +1.

Right – Right Rotation

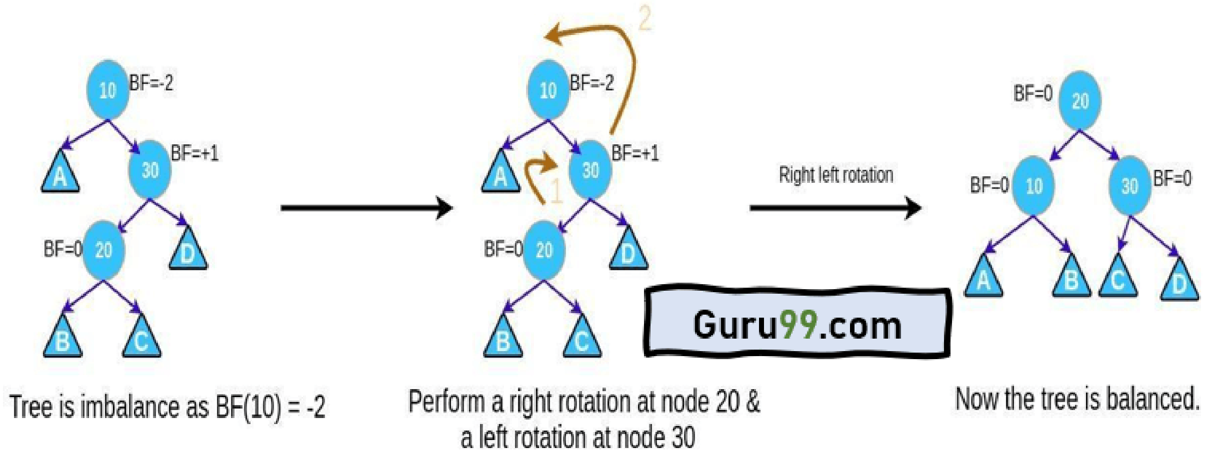
This rotation is performed when a new node is inserted at the right child of the right subtree.



A single left rotation is performed. This type of rotation is identified when a node has a balanced factor as -2, and its right-child has a balance factor as -1.

Right – Left Rotation

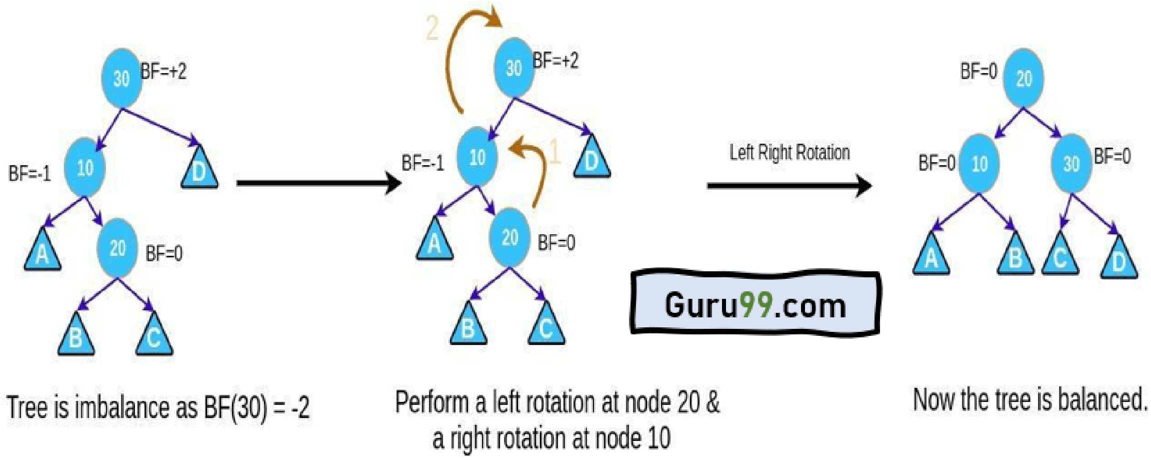
This rotation is performed when a new node is inserted at the right child of the left subtree.



This rotation is performed when a node has a balance factor as –2, and its right-child has a balance factor as +1.

Left – Right Rotation

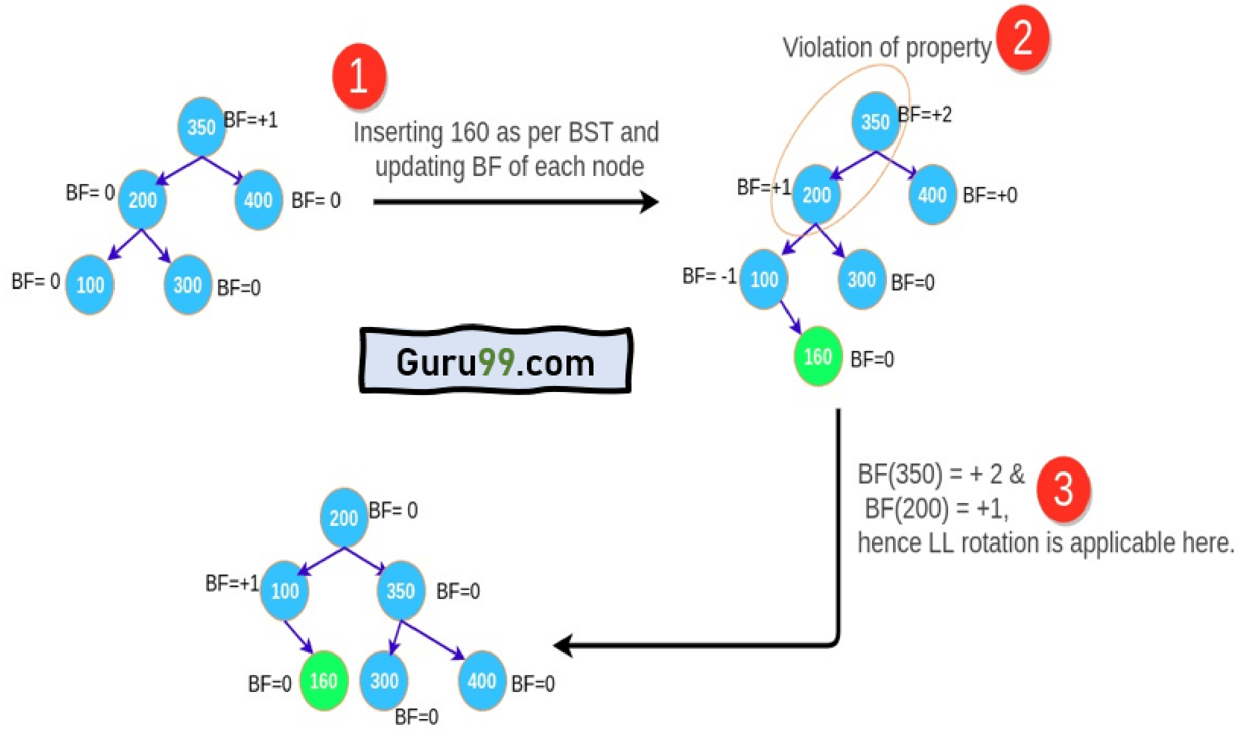
This rotation is performed when a new node is inserted at the left child of the right subtree.



This rotation is performed when a node has a balance factor as +2, and its right-child has a balance factor as -1.

Insertion in AVL Trees

Insert operation is almost the same as in simple binary search trees. After every insertion, we balance the height of the tree. Insert operation takes O(log n) worst time complexity.



AVL tree insertion implementation

Step 1:Insert the node in the AVL tree using the same insertion algorithm of BST. In the above example, insert 160.

Step 2:Once the node is added, the balance factor of each node is updated. After 160 is inserted, the balance factor of every node is updated.

Step 3:Now check if any node violates the range of the balance factor if the balance factor is violated, then perform rotations using the below case. In the above example, the balance factor of 350 is violated and case 1 becomes applicable there, we perform LL rotation and the tree is balanced again.

1. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
2. If BF(node) = -2 and BF(node -> right-child) = 1, perform RR rotation.
3. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation.
4. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.

Deletion in AVL Trees

Deletion is also very straight forward. We delete using the same logic as in simple binary search trees. After deletion, we restructure the tree, if needed, to maintain its balanced height.

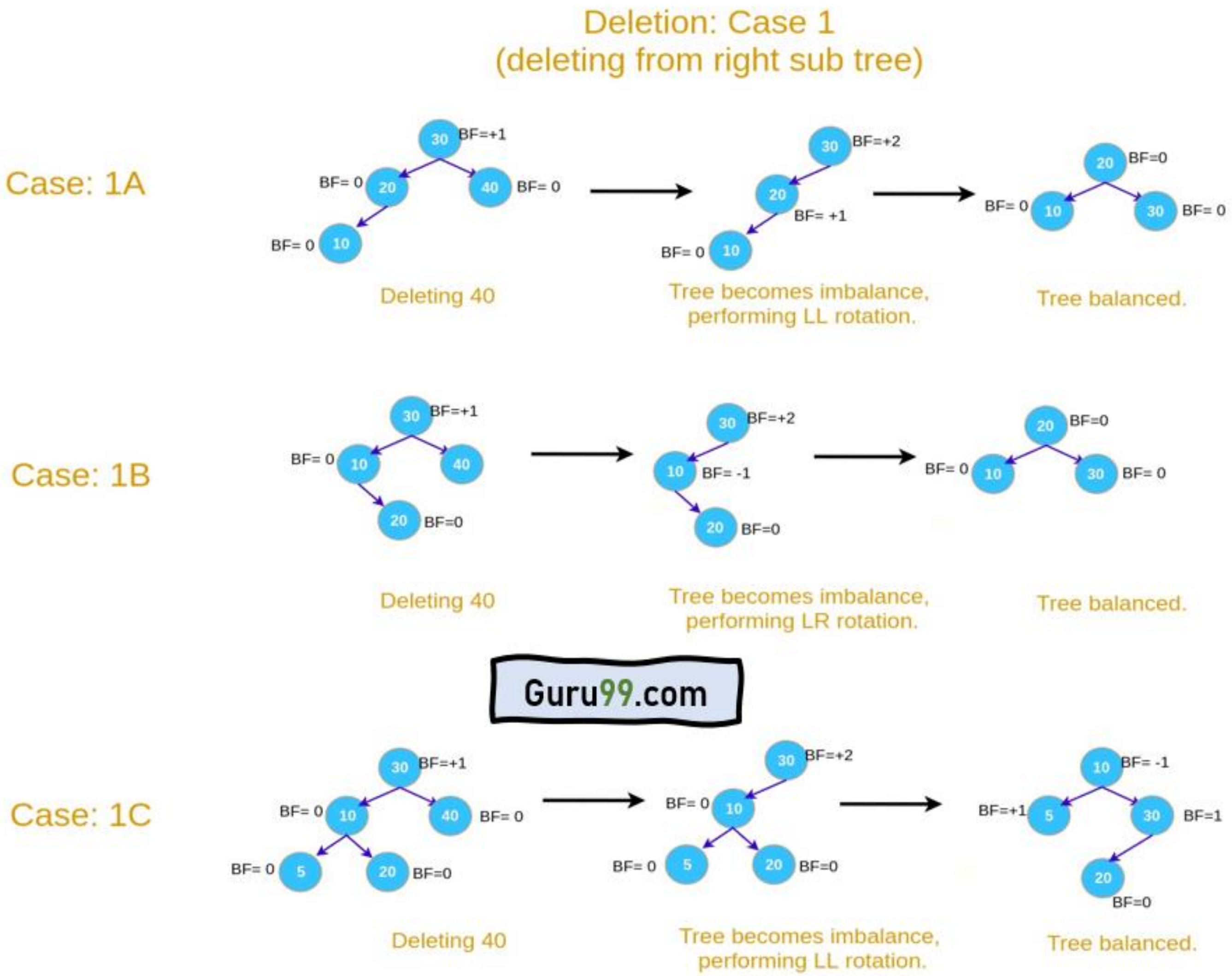
Step 1: Find the element in the tree.

Step 2: Delete the node, as per the BST Deletion.

Step 3: Two cases are possible:-

Case 1: Deleting from the right subtree.

- 1A. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
- 1B. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.
- 1C. If BF(node) = +2 and BF(node -> left-child) = 0, perform LL rotation.



Case 2: Deleting from left subtree.

- 2A. If BF(node) = -2 and BF(node -> right-child) = -1, perform RR rotation.
- 2B. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation.
- 2C. If BF(node) = -2 and BF(node -> right-child) = 0, perform RR rotation.