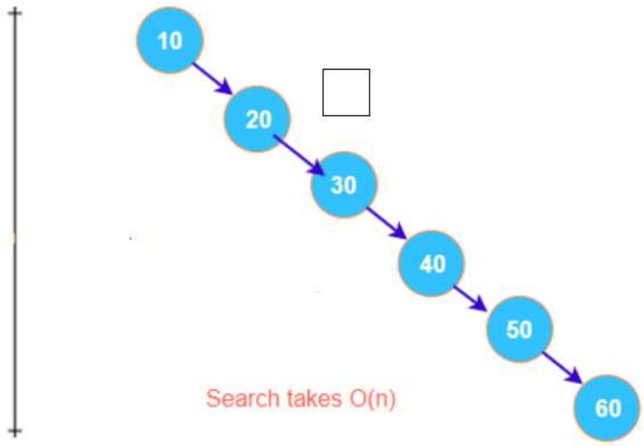
AVL TREES

AVL tree is a self – balancing binary search tree where the difference between the height of left subtree and right sub tree is -1,0 or 1.

In other words, AVL tree is defined as a balanced binary search tree whose balancing factor is -1,0 or 1. Balancing factor is defined by the difference in height of left sub tree and right sub tree. The tree is named after the investors Adelson, Velski and Landis.

NEED FOR AVL TREE

Consider the following AVL tree

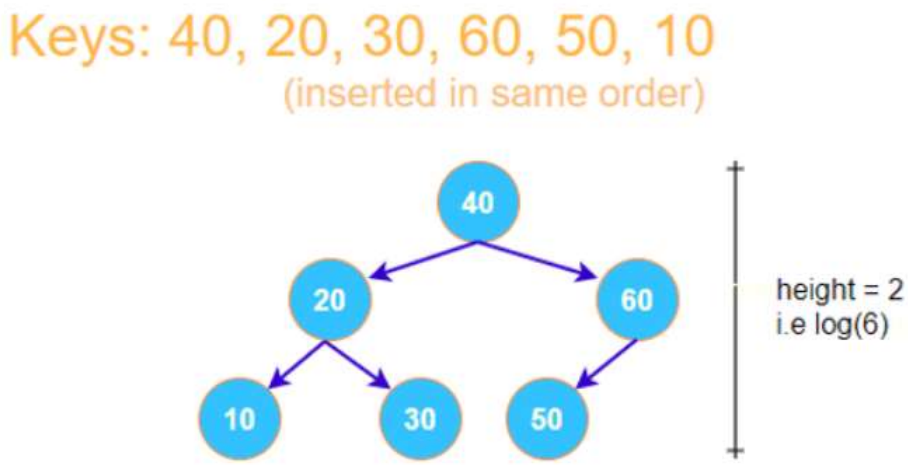


The height of the tree grows linearly in size when we insert the keys in increasing order of their value. Thus, the search operation, at worst, takes O(n).

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.

AVL tree – example

On the other hand, the following is an AVL tree.



Search takes O(log n)

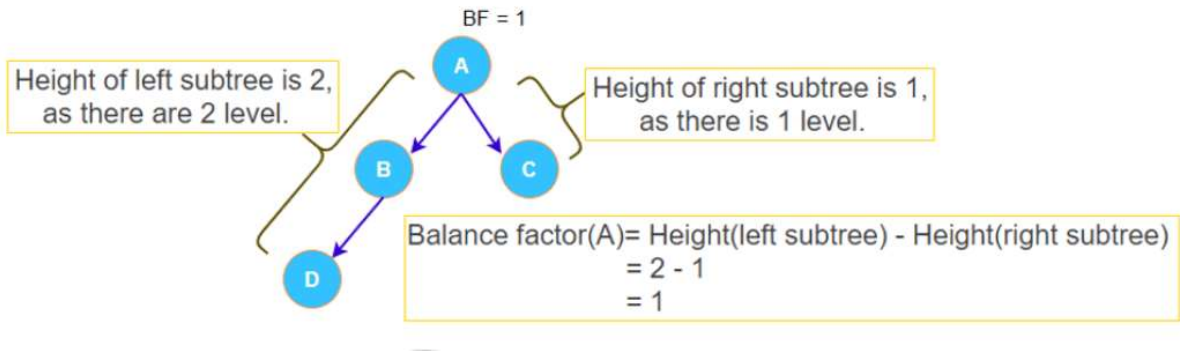
Here, the keys are the same, but since they are inserted in a different order, they take different positions, and the height of the tree remains balanced. Hence search will not take more than O(log n) for any element of the tree. It is now evident that if insertion is done correctly, the tree's height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

BALANCING FACTOR IN AVL TREES

```
BalanceFactor = height(left-sutree) − height(right-sutree)
```

Properties of balancing factor



- 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 right sub-tree contains one extra, i.e., the tree is right 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

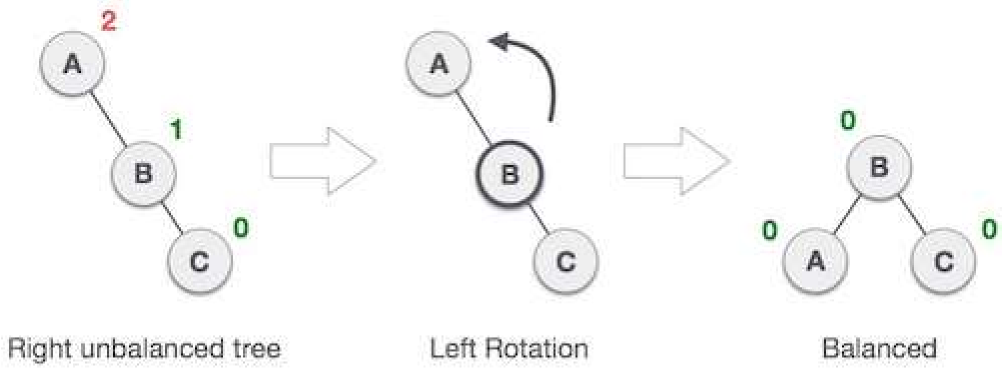
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

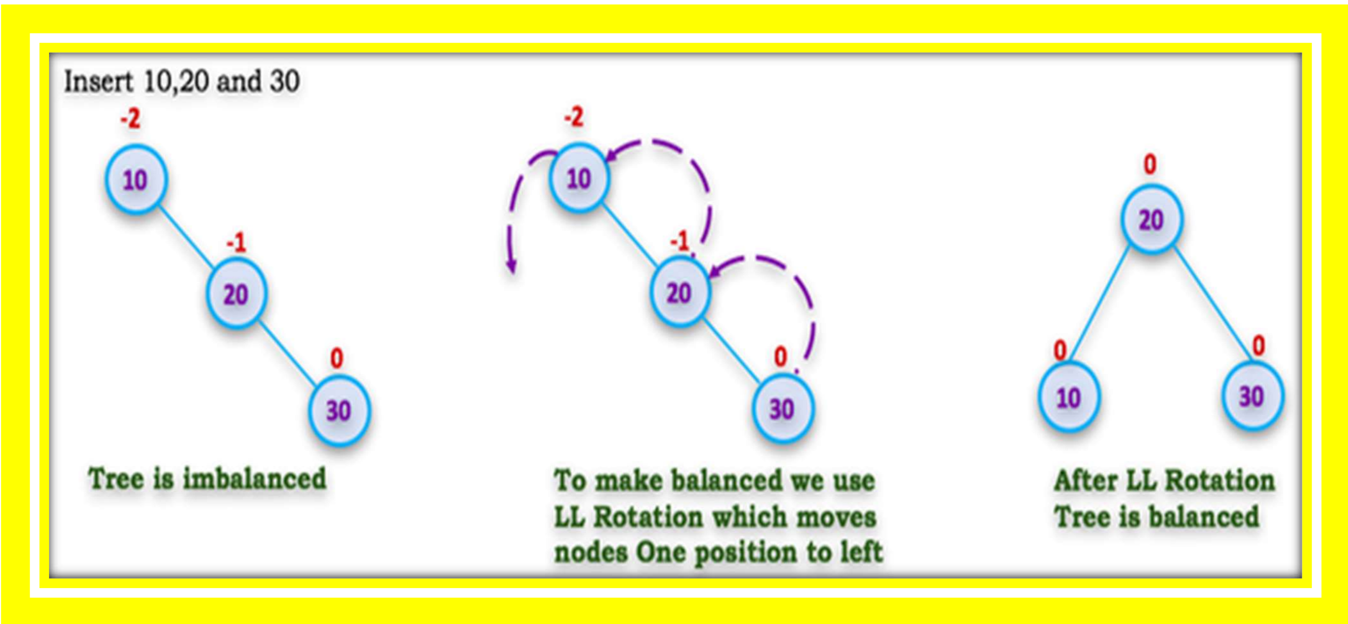
The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −

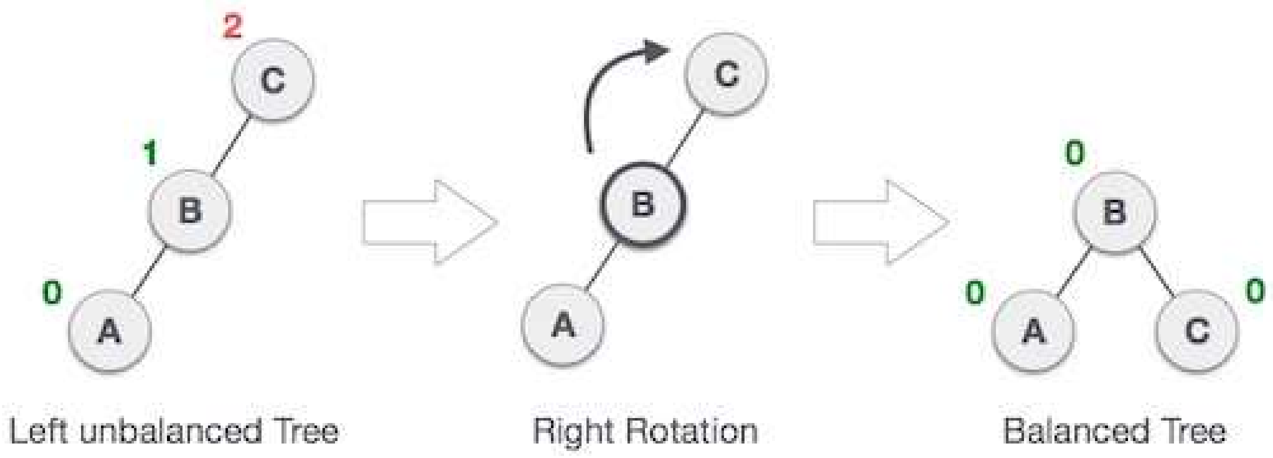


In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

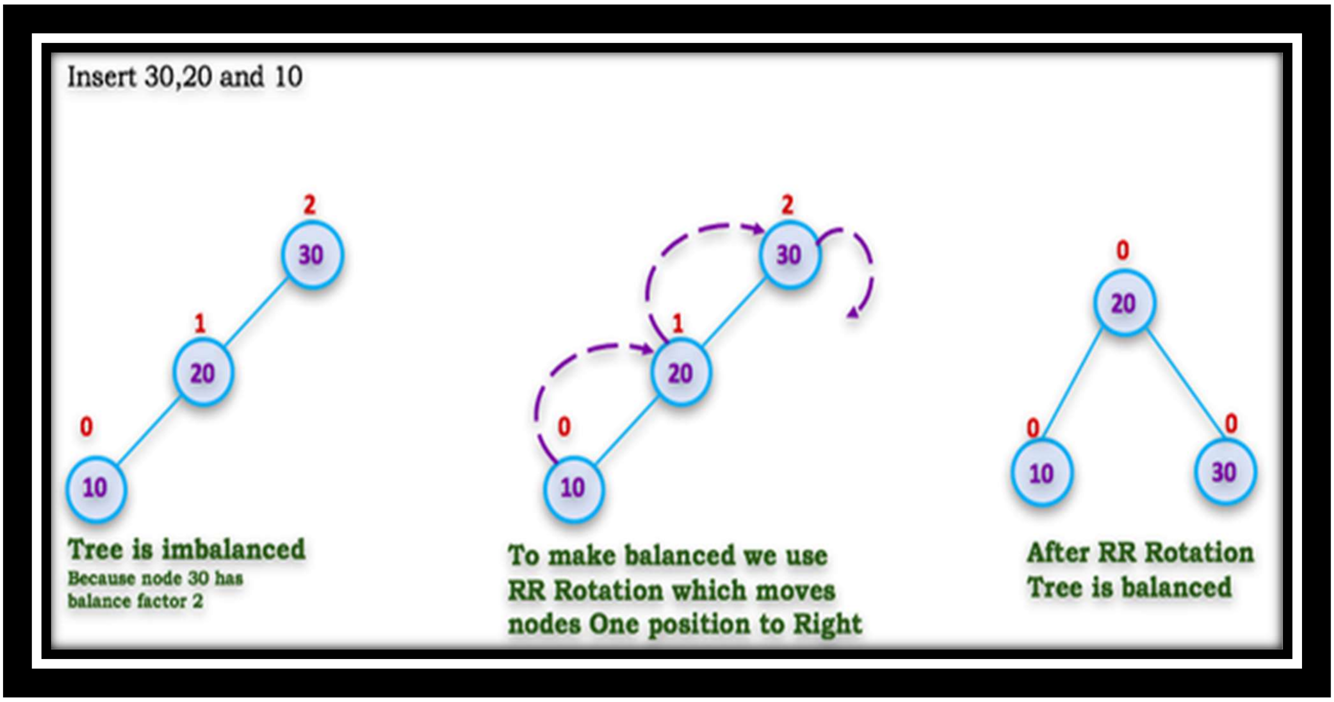


Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

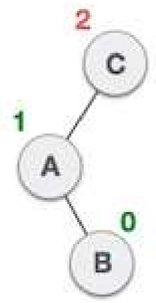
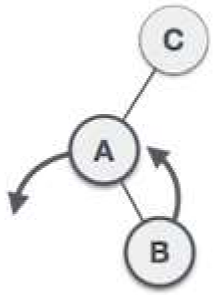
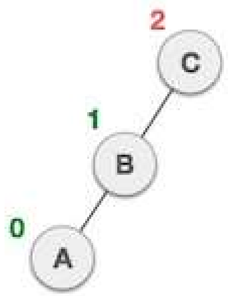
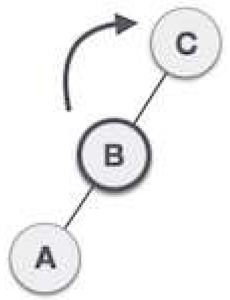


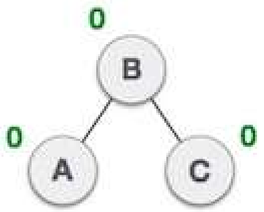
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.



Left-Right Rotation

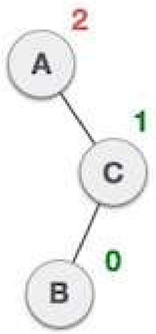
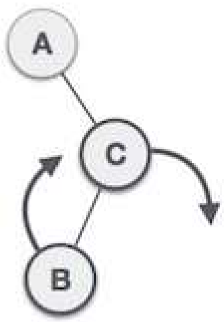
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

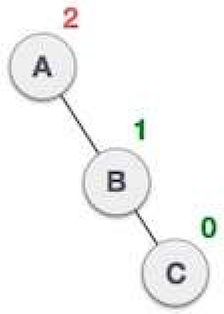
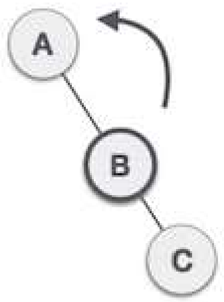
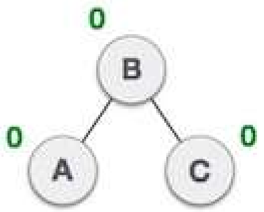
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.

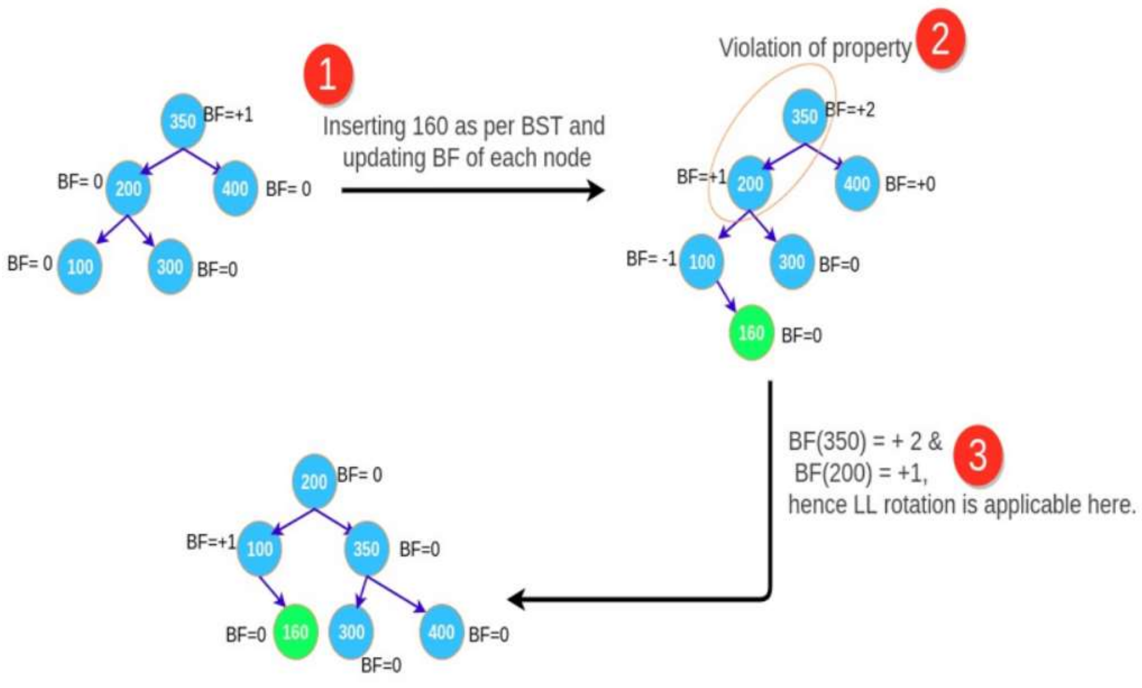
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

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.

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.



AVL tree insertion implementation

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.

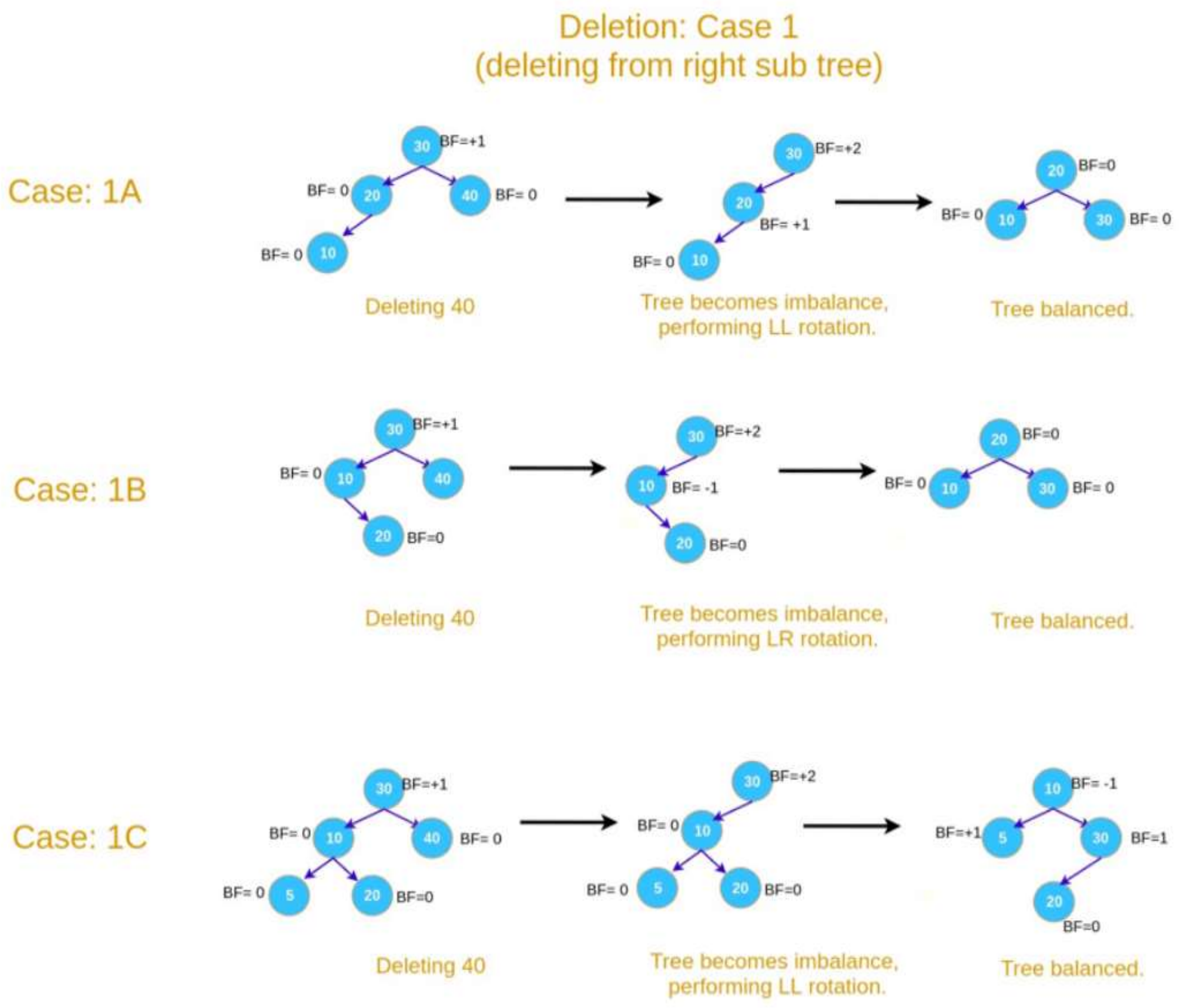
Step 1: Find the element in the tree.

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

Step 3: Two cases are possible:-

Case 1: Deleting from the right subtree.

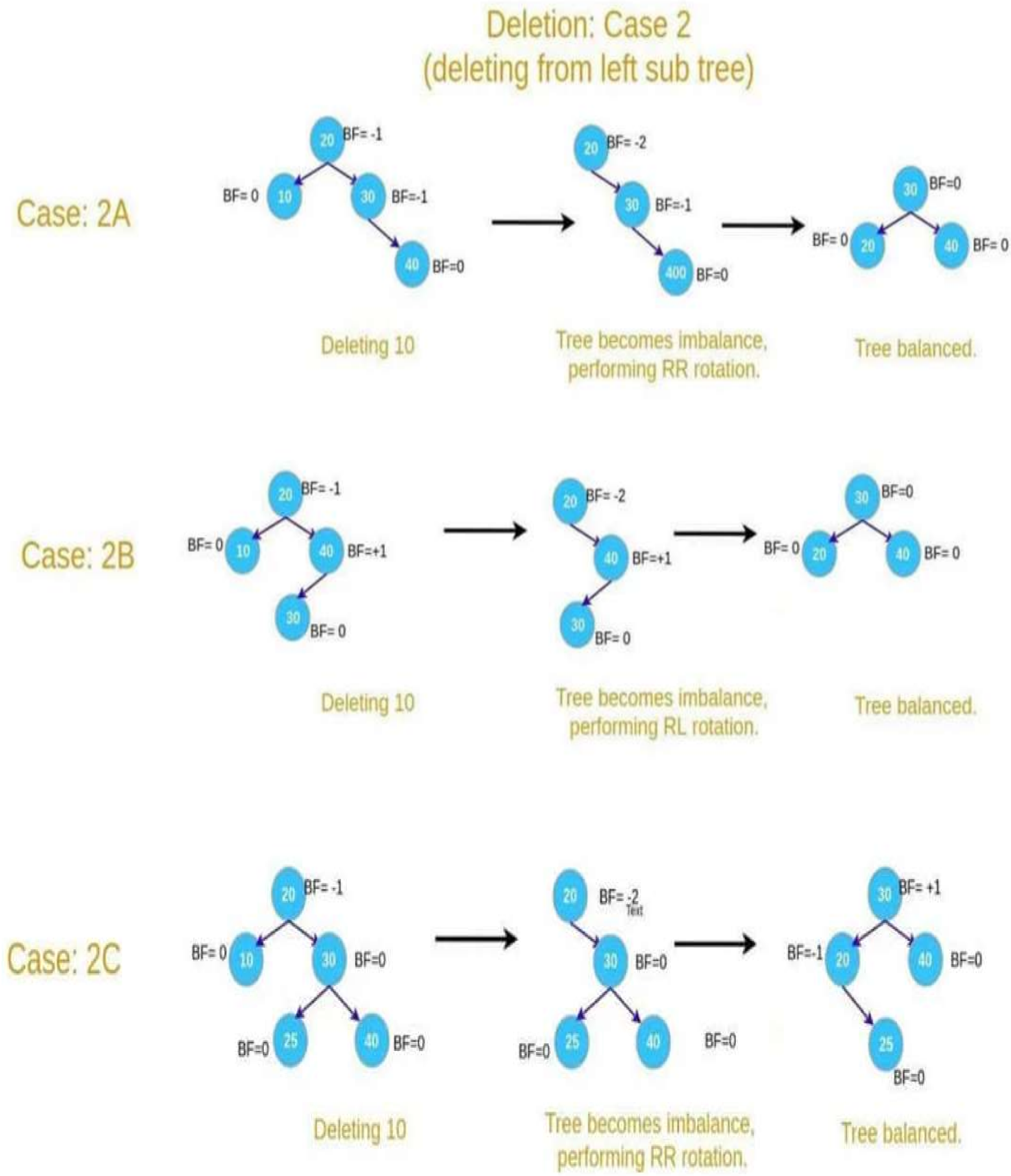
- 1A. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
- 1B. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.
- 1C. If BF(node) = +2 and BF(node -> left-child) = 0, perform LL rotation.



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.



PSEUDOCODE FOR AVL ROTATIONS

Left rotation

```
struct node * llrotation(struct node *n){
        struct node *p;
        struct node *tp;
        p = n;
        tp = p->left;
        p->left = tp->right;
        tp->right = p;
        return tp;
    }
```

Right rotation

```
    struct node * rrotation(struct node *n){
        struct node *p;
        struct node *tp;
        p = n;
        tp = p->right;
        p->right = tp->left;
        tp->left = p;
        return tp;
    }
```

Right – left rotation

```
struct node * rlrotation(struct node *n){
        struct node *p;
        struct node *tp;
        struct node *tp2;
```

```
    p = n;
    tp = p->right;
    tp2 =p->right->left;
    p -> right = tp2->left;
    tp ->left = tp2->right;
    tp2 ->left = p;
    tp2->right = tp;
    return tp2;
  }
```

Left – right rotation

```
  struct node * lrrotation(struct node *n){
    struct node *p;
    struct node *tp;
    struct node *tp2;
    p = n;
    tp = p->left;
    tp2 =p->left->right;
    p -> left = tp2->right;
    tp ->right = tp2->left;
    tp2 ->right = p;
    tp2->left = tp;
    return tp2;
  }
```

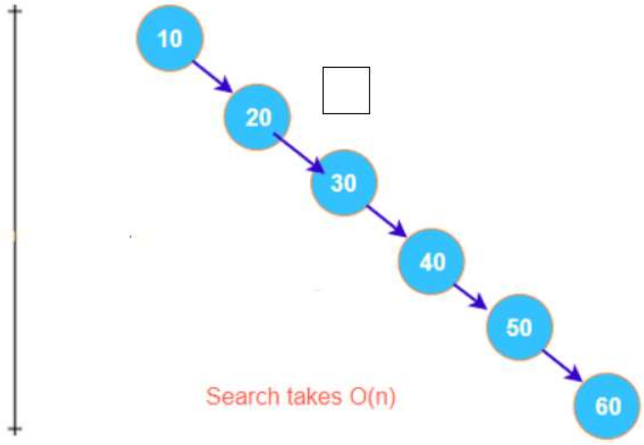
AVL TREES

AVL tree is a self – balancing binary search tree where the difference between the height of left subtree and right sub tree is -1,0 or 1.

In other words, AVL tree is defined as a balanced binary search tree whose balancing factor is -1,0 or 1. Balancing factor is defined by the difference in height of left sub tree and right sub tree. The tree is named after the investors Adelson, Velski and Landis.

NEED FOR AVL TREE

Consider the following AVL tree

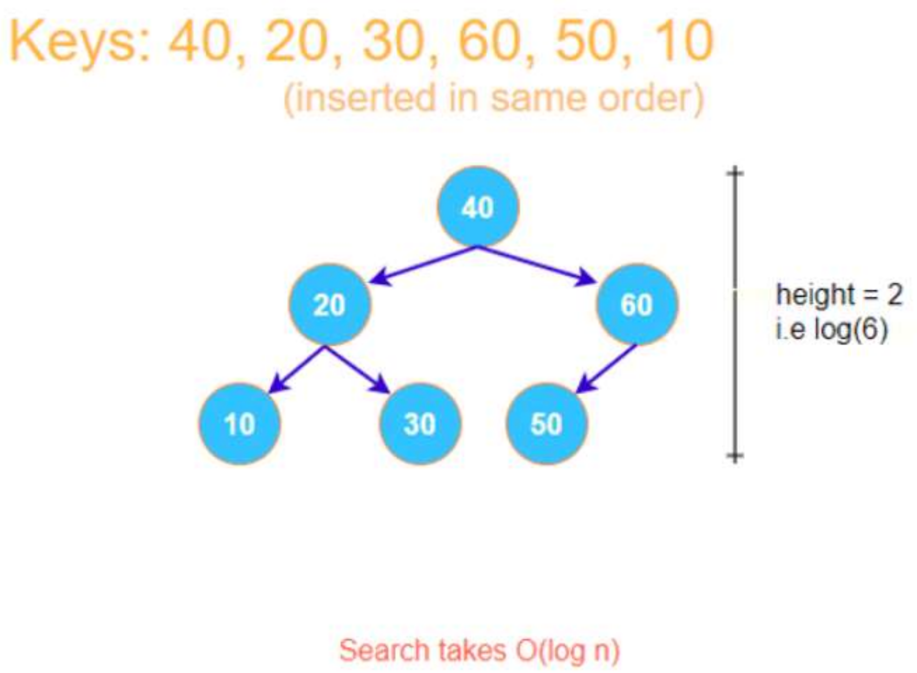


The height of the tree grows linearly in size when we insert the keys in increasing order of their value. Thus, the search operation, at worst, takes O(n).

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.

AVL tree – example

On the other hand, the following is an AVL tree.



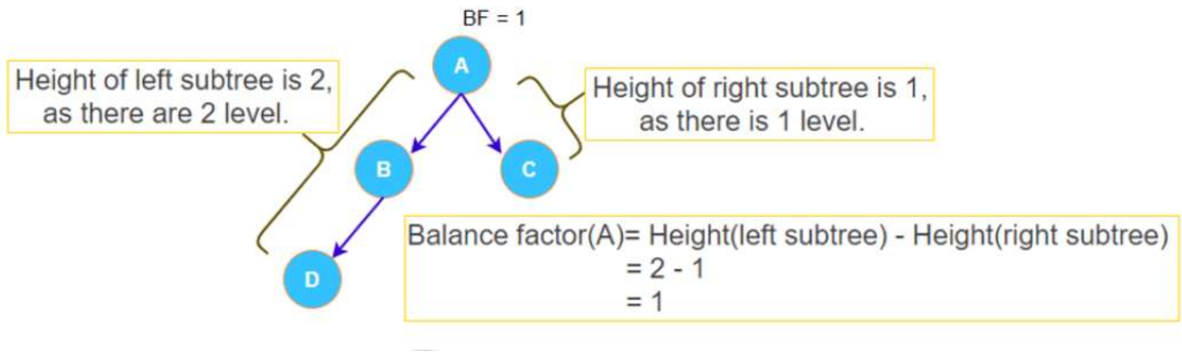
Here, the keys are the same, but since they are inserted in a different order, they take different positions, and the height of the tree remains balanced. Hence search will not take more than O(log n) for any element of the tree. It is now evident that if insertion is done correctly, the tree's height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

BALANCING FACTOR IN AVL TREES

```
BalanceFactor = height(left-sutree) − height(right-sutree)
```

Properties of balancing factor



- 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 right sub-tree contains one extra, i.e., the tree is right 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

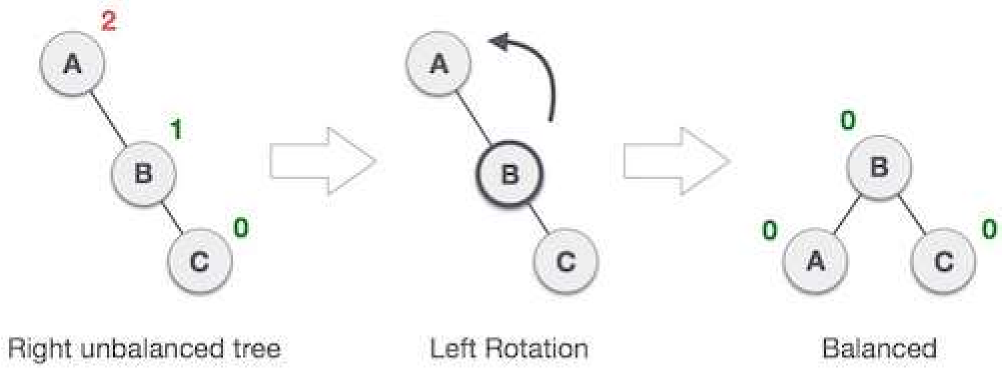
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

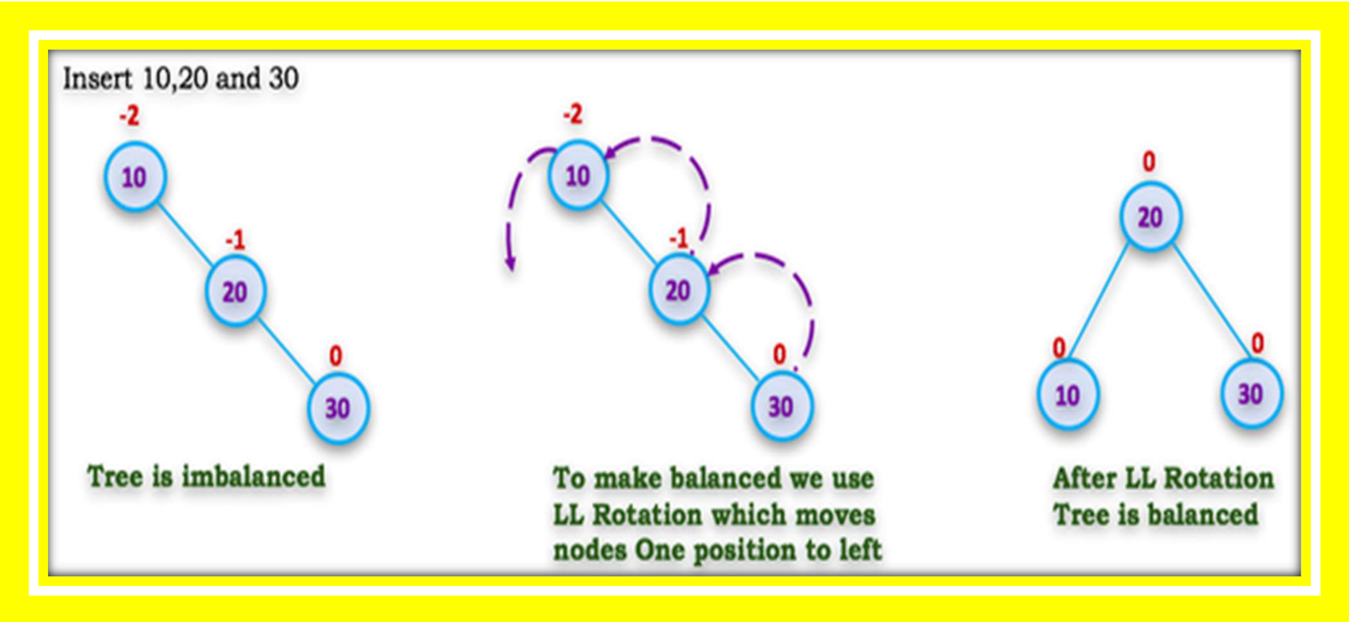
The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −

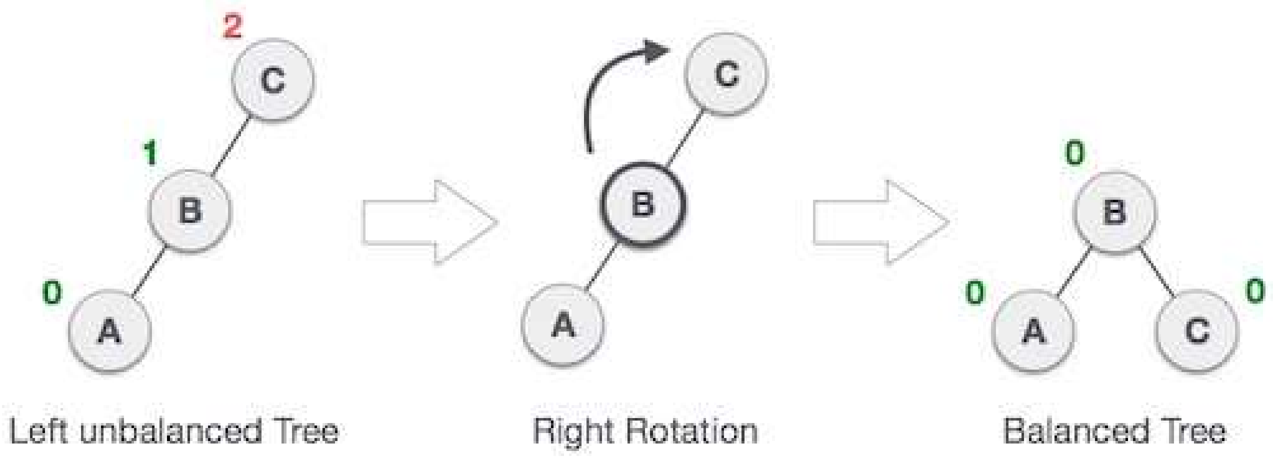


In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

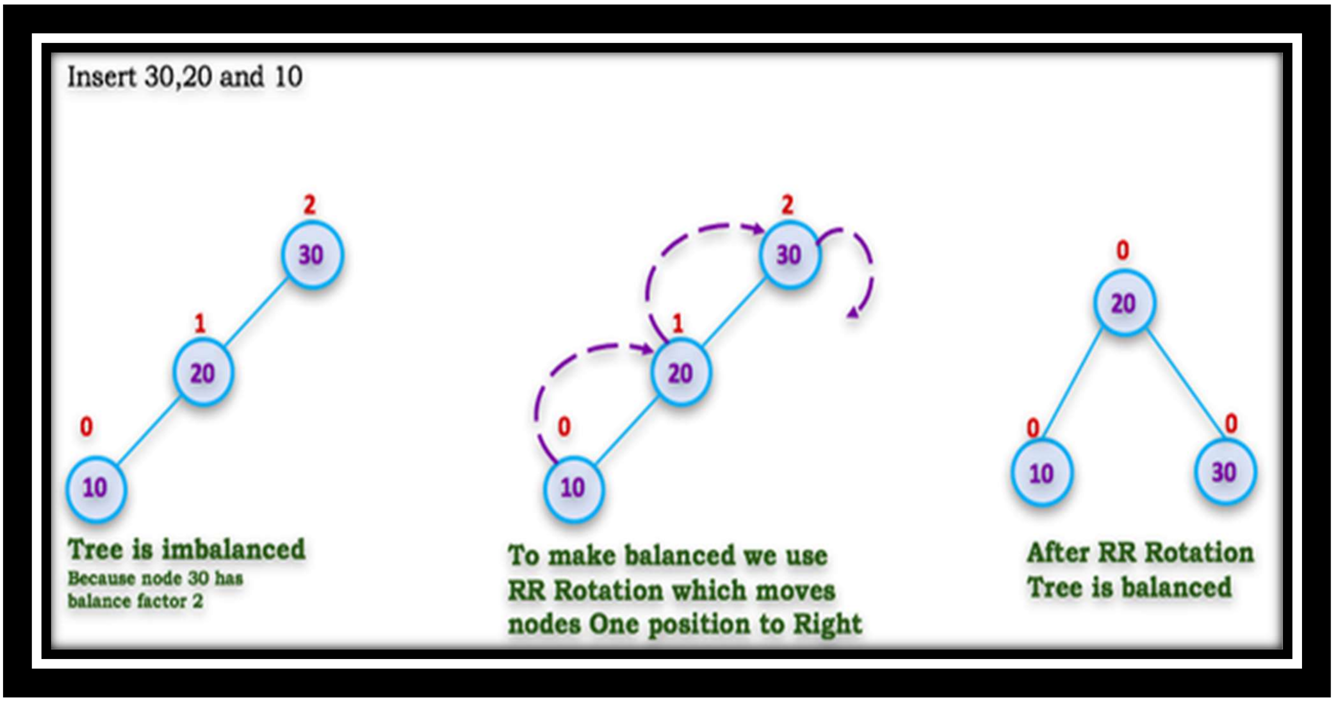


Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

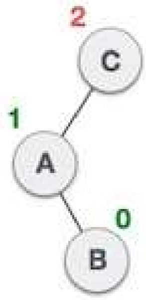
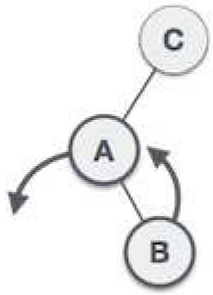
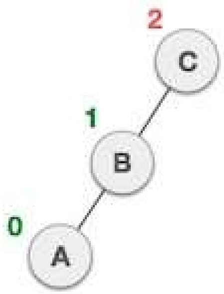
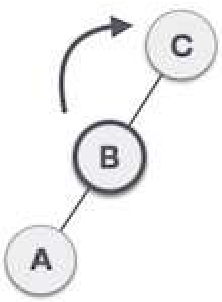


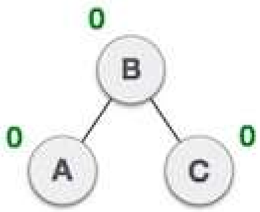
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.



Left-Right Rotation

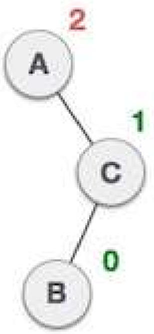
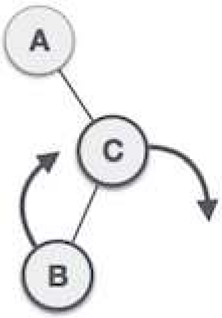
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

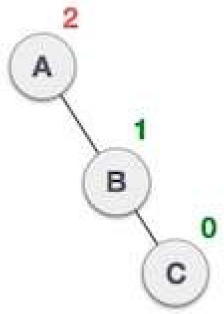
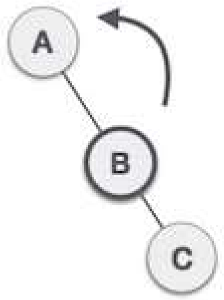
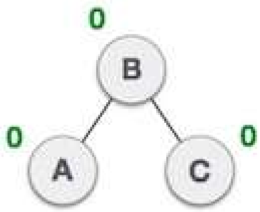
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.

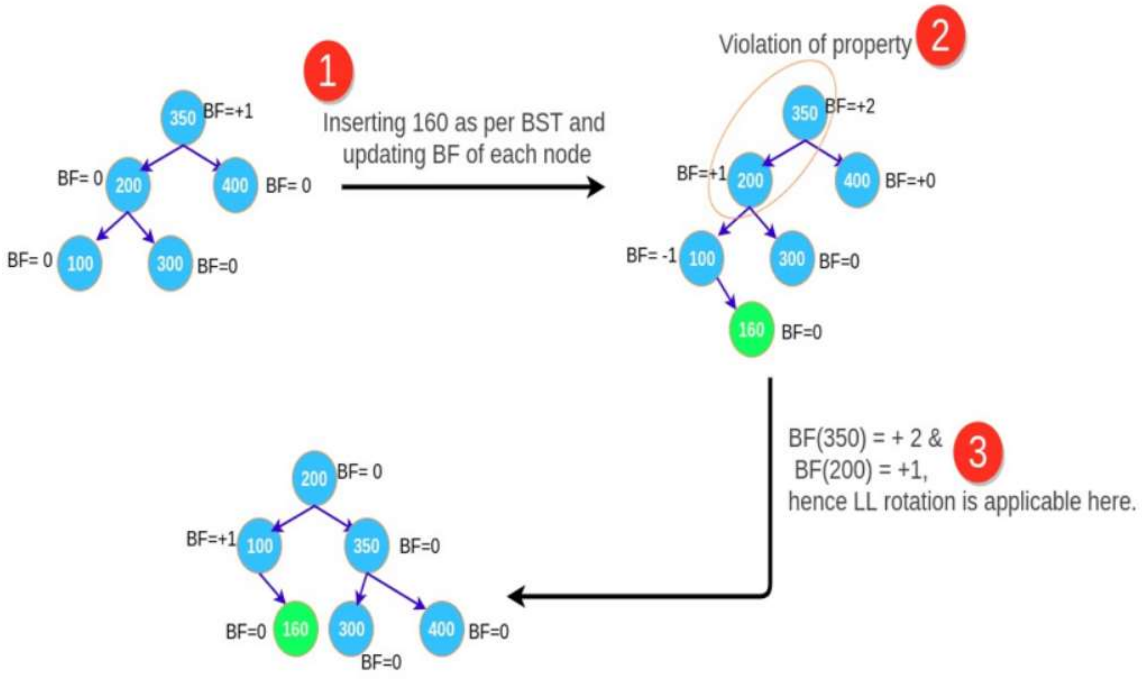
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

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.

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.



AVL tree insertion implementation

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.

5. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
6. If BF(node) = -2 and BF(node -> right-child) = 1, perform RR rotation.
7. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation.
8. 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.

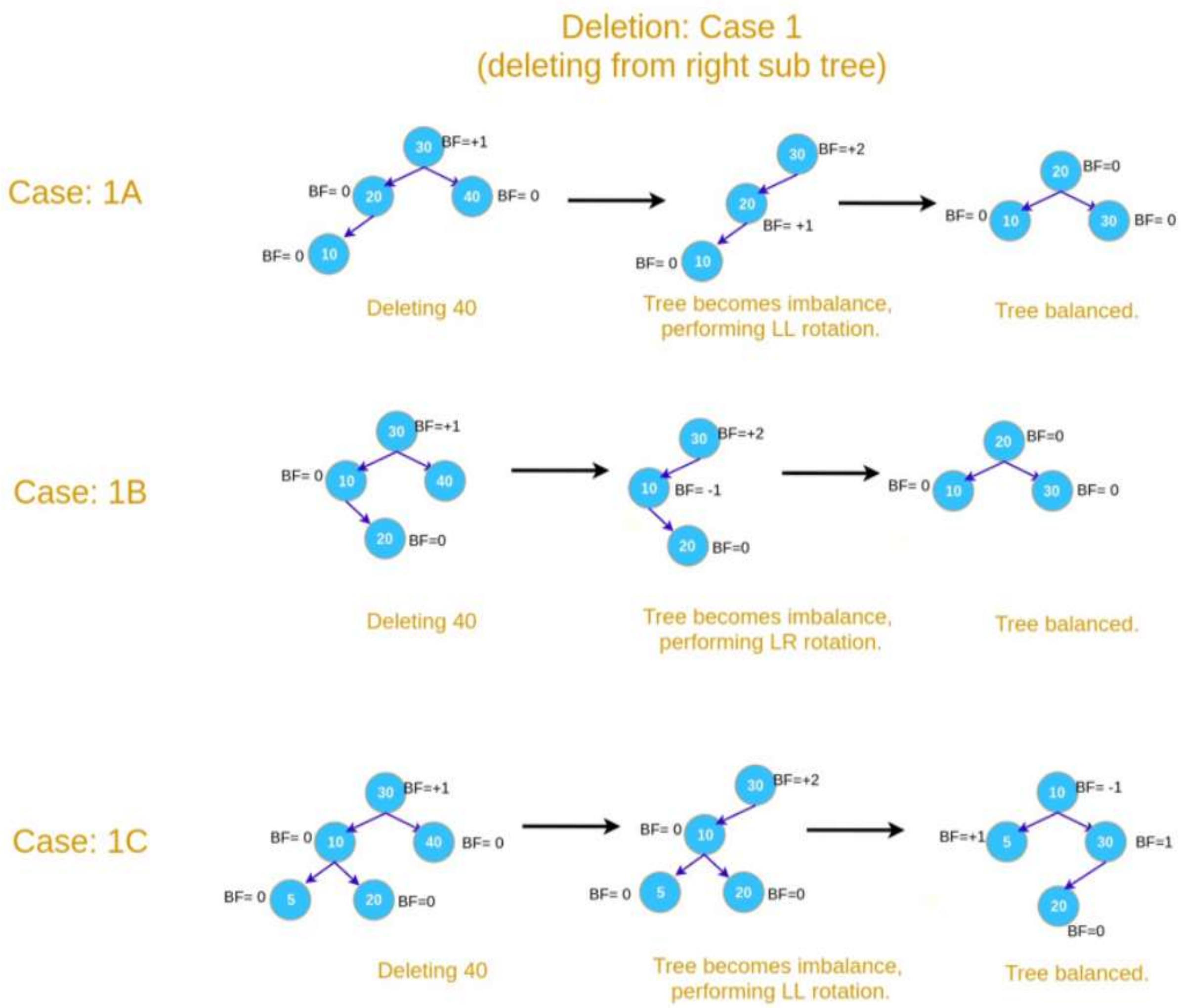
Step 1: Find the element in the tree.

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

Step 3: Two cases are possible:-

Case 1: Deleting from the right subtree.

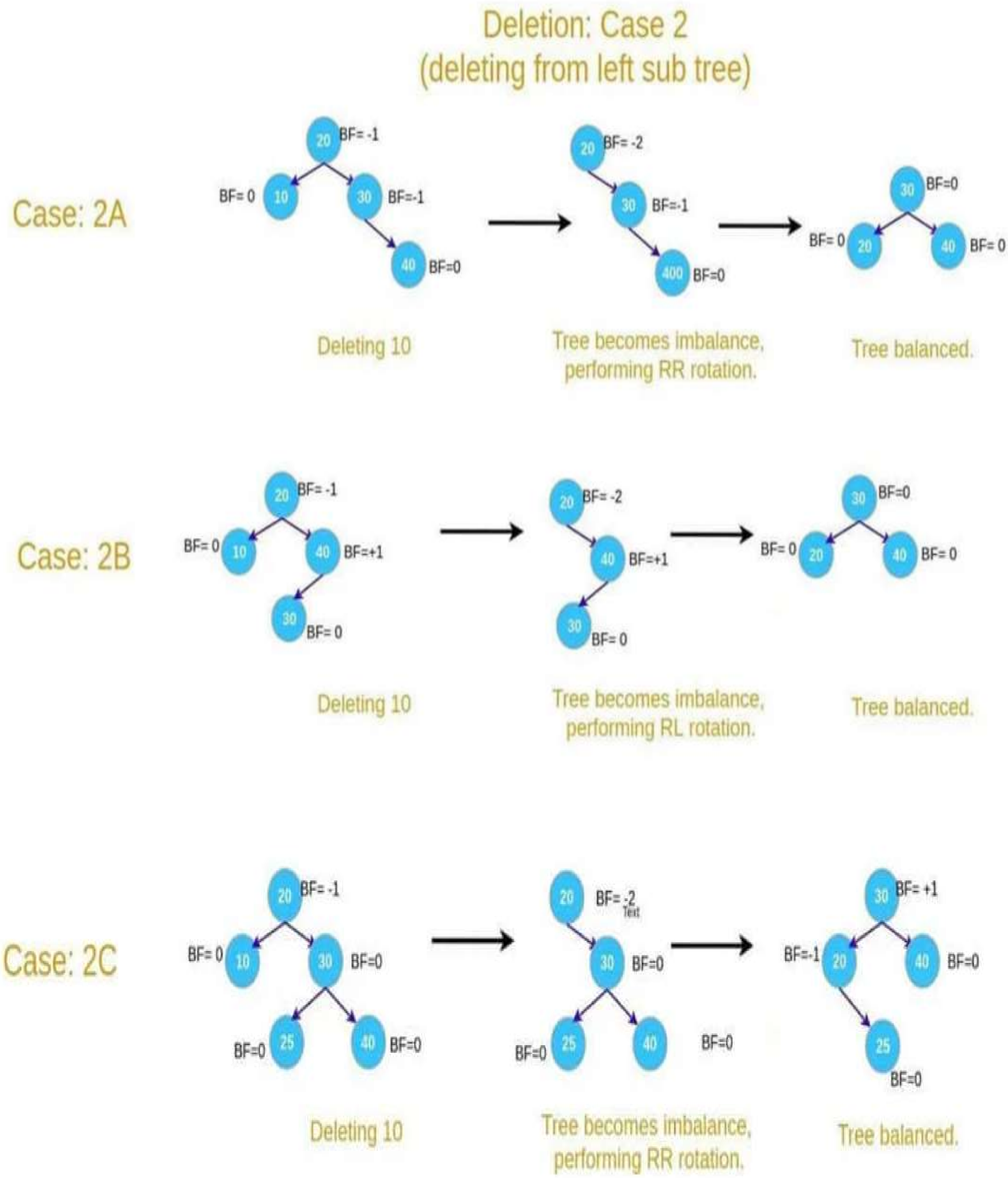
- 1A. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
- 1B. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.
- 1C. If BF(node) = +2 and BF(node -> left-child) = 0, perform LL rotation.



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.



PSEUDOCODE FOR AVL ROTATIONS

Left rotation

```
struct node * llrotation(struct node *n){
        struct node *p;
        struct node *tp;
        p = n;
        tp = p->left;
        p->left = tp->right;
        tp->right = p;
        return tp;
    }
```

Right rotation

```
    struct node * rrrotation(struct node *n){
        struct node *p;
        struct node *tp;
        p = n;
        tp = p->right;
        p->right = tp->left;
        tp->left = p;
        return tp;
    }
```

Right – left rotation

```
struct node * rlrotation(struct node *n){
        struct node *p;
        struct node *tp;
        struct node *tp2;
```

```
    p = n;
    tp = p->right;
    tp2 =p->right->left;
    p -> right = tp2->left;
    tp ->left = tp2->right;
    tp2 ->left = p;
    tp2->right = tp;
    return tp2;
}
```

<u>Left – right rotation</u>

```
struct node * lrrotation(struct node *n){
    struct node *p;
    struct node *tp;
    struct node *tp2;
    p = n;
    tp = p->left;
    tp2 =p->left->right;
    p -> left = tp2->right;
    tp ->right = tp2->left;
    tp2 ->right = p;
    tp2->left = tp;
    return tp2;
}
```