AVL TREE DATA STRUCTURE

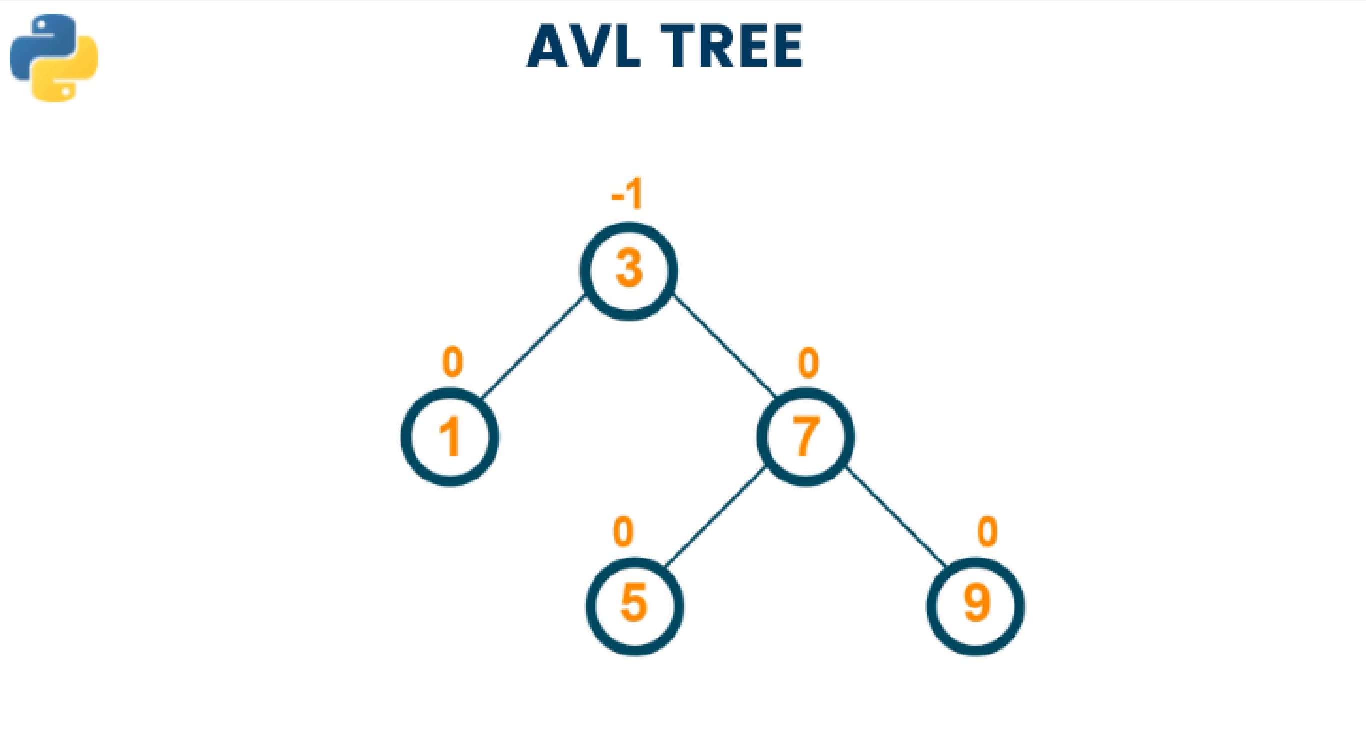
Presented By : Aditi Naik
Siddhesh More

OVERVIEW

- Introduction
- Problems
- Solution to problem
- Insertion & Deletion Operation
- Rotations
- Time complexity

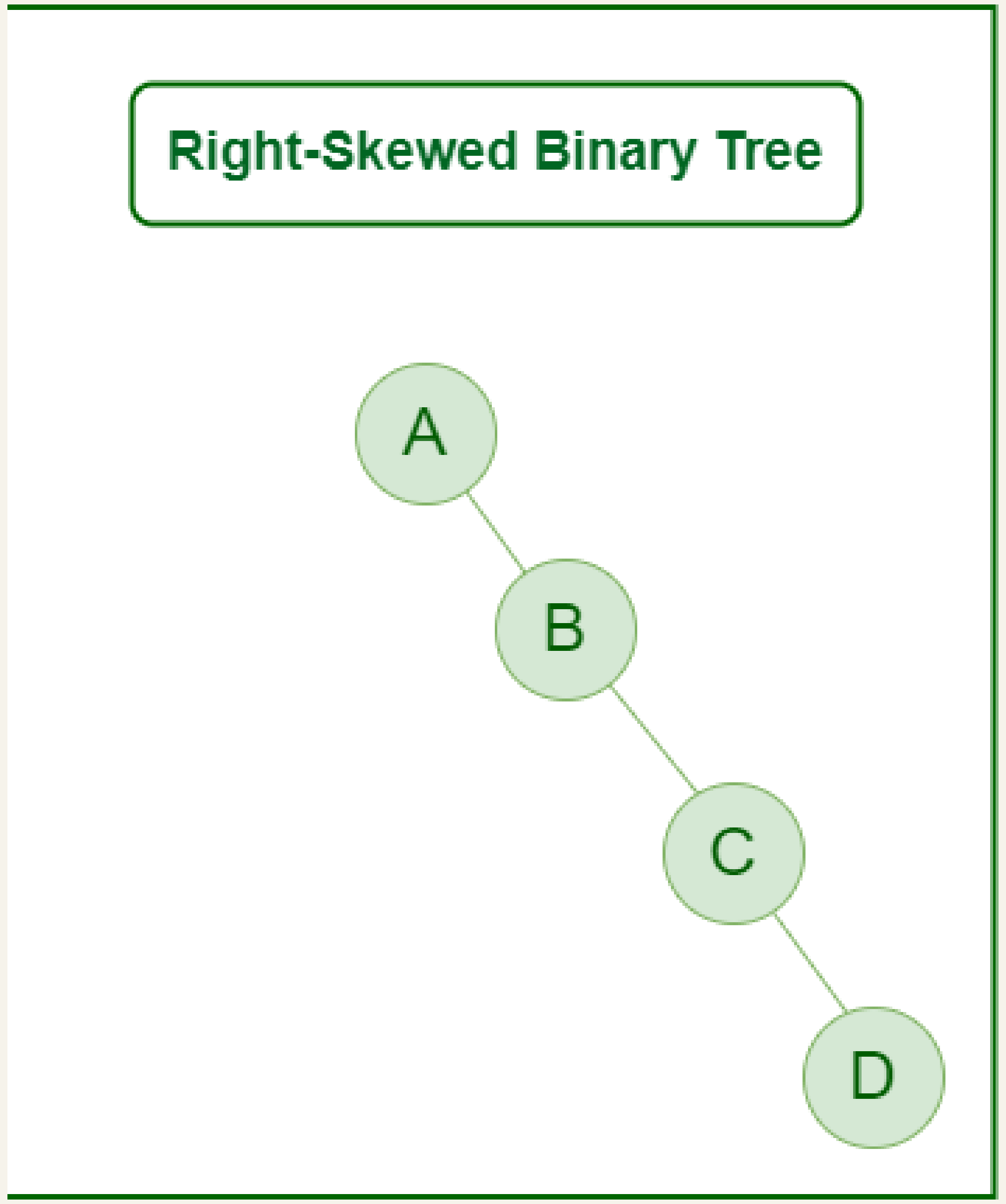
INTRODUCTION

- AVL Tree is a self-balancing binary search tree where the difference between the heights of the left and right subtrees of any node is at most one.
- Every node in an AVL Tree has a balance factor of -1, 0, or +1.
- To maintain the height balance of a BST to ensure efficient searching, insertion, and deletion.
- Used in database indexing, file systems, and memory management.



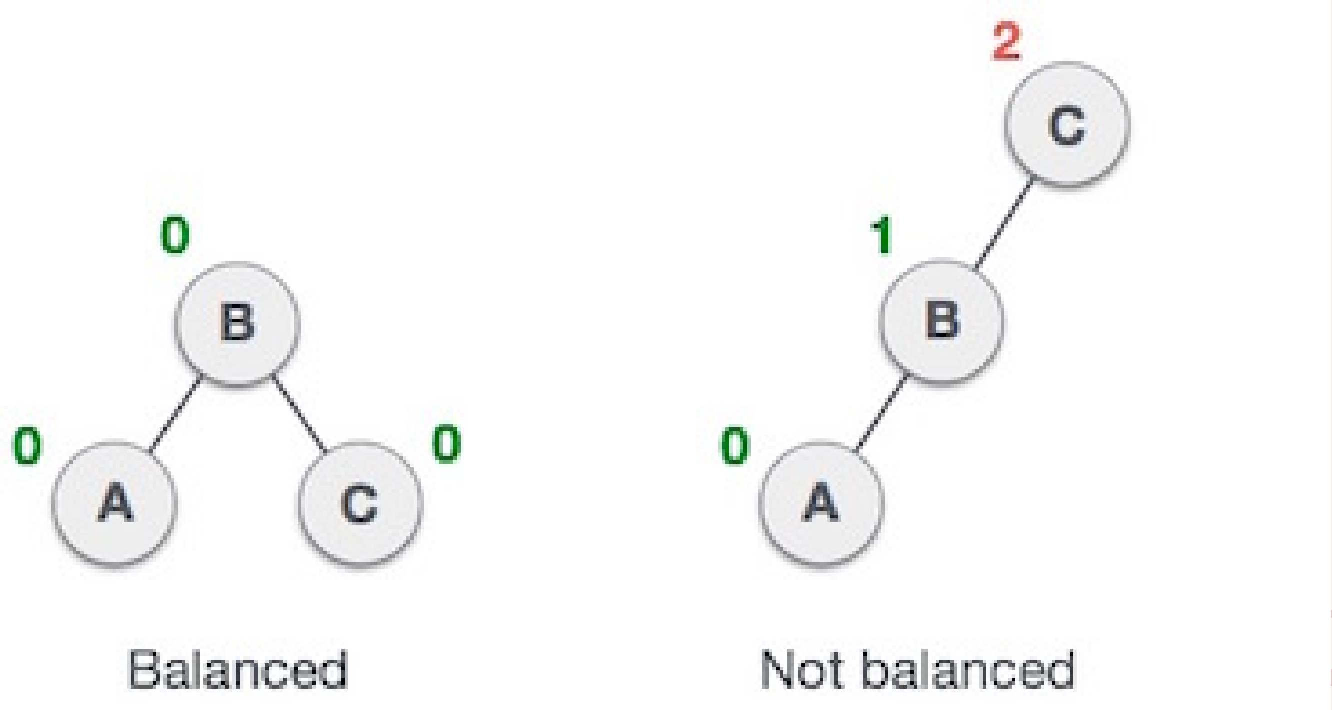
PROBLEMS OF BINARY SEARCH TREES (BST)

- Unbalanced Growth:
- In an unbalanced BST, nodes may not be evenly distributed across levels.
- Example: inserting sorted data results in a skewed tree (degenerates into a linked list).
- Increased Search Time:
- Depth of the tree can increase to O(n)in worst-case scenarios.
- Searching, inserting, and deleting operations become inefficient when the tree is unbalanced.



HOW AVL TREES SOLVE BST PROBLEMS

- Automatic Balancing:
- After each insertion or deletion, AVL trees adjust themselves to maintain balance.
- Ensures that the tree's height is always O(logn)).
- Efficient Search, Insertion, and Deletion:
- Operations take O(logn) time due to balanced height.
- Improves efficiency over unbalanced BSTs.



INSERTION/DELETION:

- Perform a standard BST insertion/Deletion
- Calculate the balancing factor of each node
- Perform rotations (if unbalanced):

Balancing Factor

BF(N)=Height of left subtree - Height of right subtree

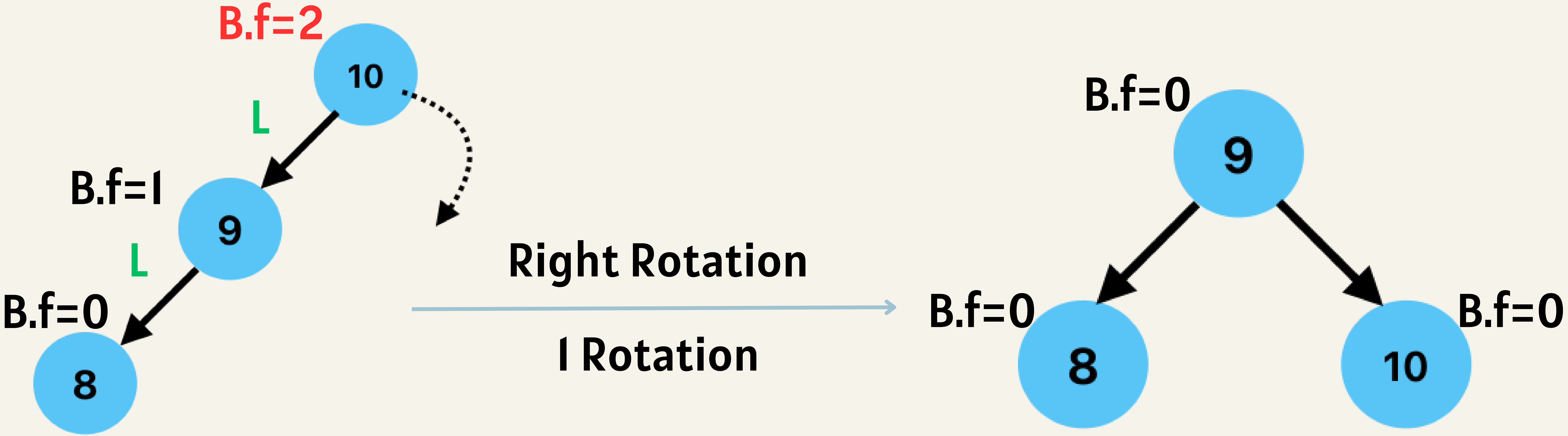
Rotation

Left-left (LL)
Right-right (RR)
Left-right (LR)
Right-left (RL)

ROTATION

Left-Left Case (LL)

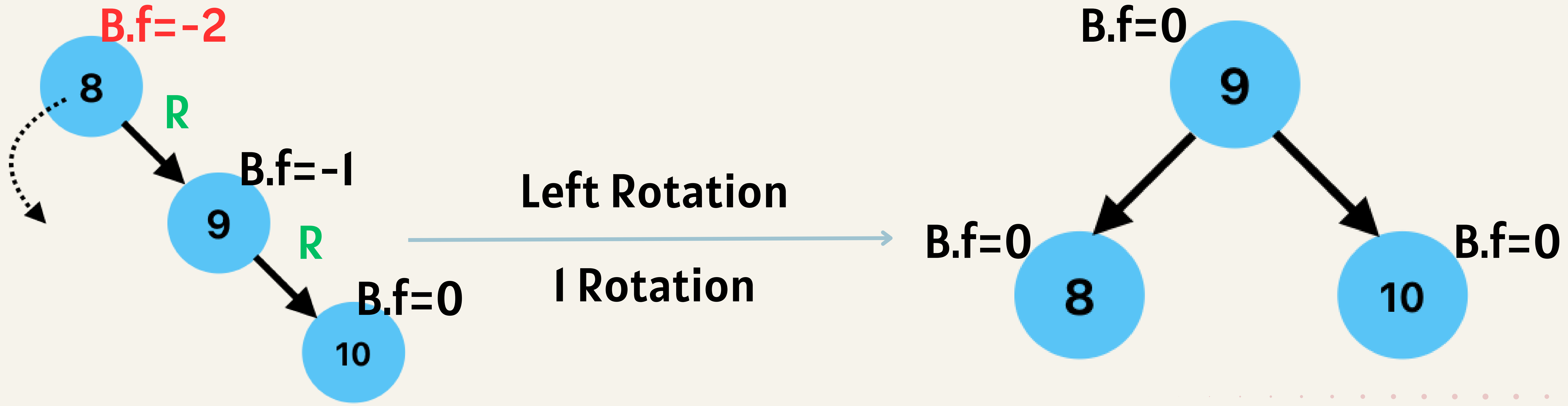
Array: [10,9,8]



ROTATION

Right- Right Case(RR)

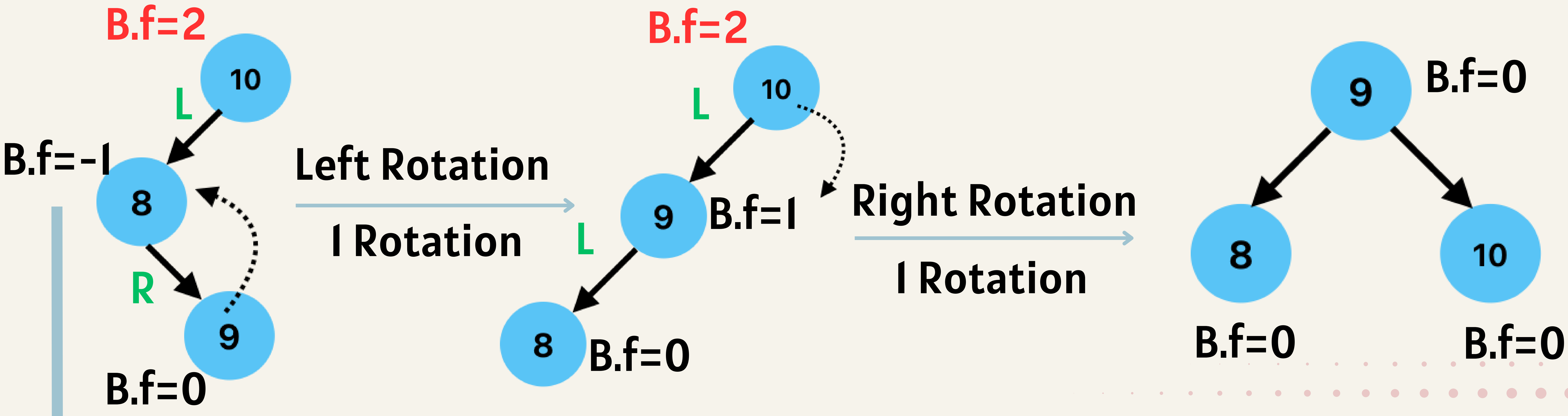
Array: [8,9,10]



ROTATION

Left-right case(LR)

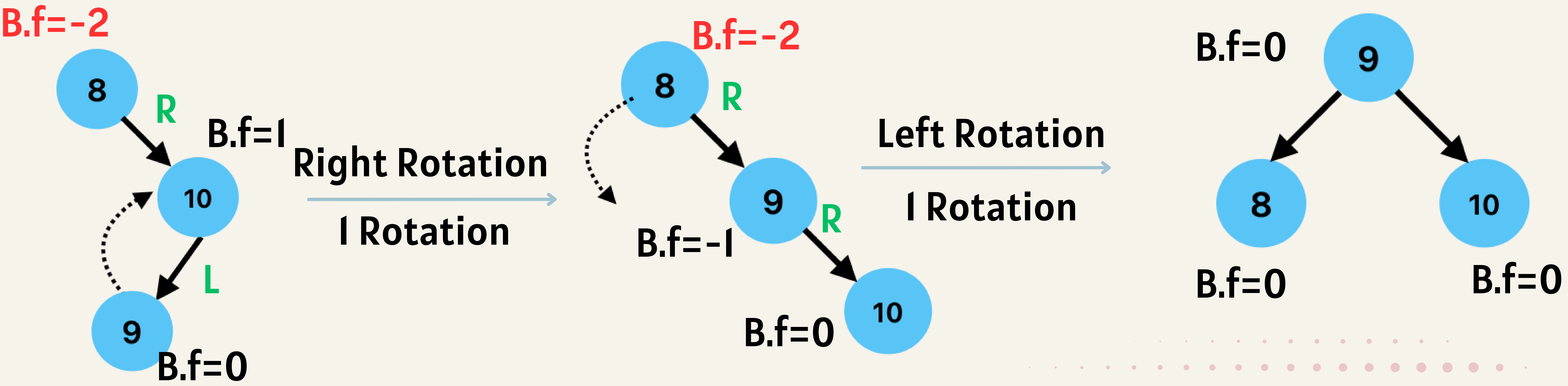
Array: [10,8,9]



ROTATION

Right-left case(RL)

Array: [8,10,9]



CODE FOR RIGHT ROTATIONS


```java
static Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = Math.max(height(y.left),
                        height(y.right)) + 1;
    x.height = Math.max(height(x.left),
                        height(x.right)) + 1;

    // Return new root
    return x;
}
```

```java
static int height(Node N) {
    if (N == null)
        return 0;
    return N.height;
}
```

CODE FOR LEFT ROTATION


```
static Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = Math.max(height(x.left),
                        height(x.right)) + 1;
    y.height = Math.max(height(y.left),
                        height(y.right)) + 1;

    // Return new root
    return y;
}
```

CHECK BALANCING CODE


```java
// Get Balance factor of node N
static int getBalance(Node N) {
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}
```

```java
// Left Left Case
if (balance > 1 && key < node.left.key)
    return rightRotate(node);

// Right Right Case
if (balance < -1 && key > node.right.key)
    return leftRotate(node);

// Left Right Case
if (balance > 1 && key > node.left.key) {
    node.left = leftRotate(node.left);
    return rightRotate(node);
}

// Right Left Case
if (balance < -1 && key < node.right.key) {
    node.right = rightRotate(node.right);
    return leftRotate(node);
}
```

INSERTION CODE


```java
static Node insert(Node node, int key) {
    // 1. Perform the normal BST insertion
    if (node == null)
        return new Node(key);

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else // Duplicate keys not allowed
        return node;

    // 2. Update height of this ancestor node
    node.height = Math.max(height(node.left),
                           height(node.right)) + 1;
```

```java
    // 3. Get the balance factor of this node
    // to check whether this node became
    // unbalanced
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}
```

DELETION CODE


```java
static Node deleteNode(Node root, int key) {
    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == null)
        return root;

    // If the key to be deleted is smaller
    // than the root's key, then it lies in
    // left subtree
    if (key < root.key)
        root.left = deleteNode(root.left, key);

    // If the key to be deleted is greater
    // than the root's key, then it lies in
    // right subtree
    else if (key > root.key)
        root.right = deleteNode(root.right, key);

    // if key is same as root's key, then
    // this is the node to be deleted
    else {
        // node with only one child or no child
        if ((root.left == null) ||
            (root.right == null)) {
            Node temp = root.left != null ?
                        root.left : root.right;

            // No child case
            if (temp == null) {
                temp = root;
                root = null;
            } else // One child case
                root = temp; // Copy the contents of
                             // the non-empty child
```

```java
        } else {
            // node with two children: Get the
            // inorder successor (smallest in
            // the right subtree)
            Node temp = minValueNode(root.right);

            // Copy the inorder successor's
            // data to this node
            root.key = temp.key;

            // Delete the inorder successor
            root.right = deleteNode(root.right, temp.key);
        }
    }

    // If the tree had only one node then return
    if (root == null)
        return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root.height = Math.max(height(root.left),
                           height(root.right)) + 1;
```

```java
    // 3. Get the balance factor of this node
    // to check whether this node became
    // unbalanced
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}
```

TIME COMPLEXITY

Step 1: BST Insertion

In a balanced AVL tree with n nodes,

the height h is bounded by:

$h \leq c \log(n)$

where c is a constant (approximately 1.44).

The time complexity of BST insertion/deletion is O(h), so:

$T(n) = O(\log n)$

Step 2: Rebalancing

1. Updating balance factors
2. Performing rotations (if needed)

$T(n) = O(\log n)$

Total Time Complexity

$T(n) = T_{insert/delete}(n) + T_{rebalance}(n)$

$T(n) = O(\log n) + O(\log n)$

$T(n) = O(\log n)$