CL2001 Data Structures Lab	Lab 09 Balance in Binary Search Trees, AVL Trees with all operations

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

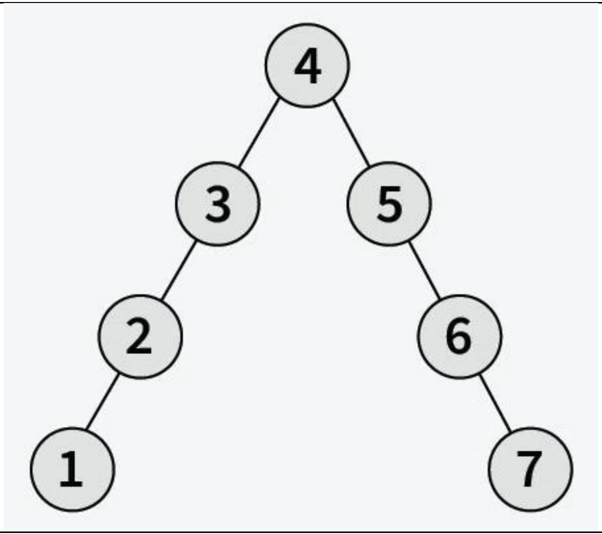
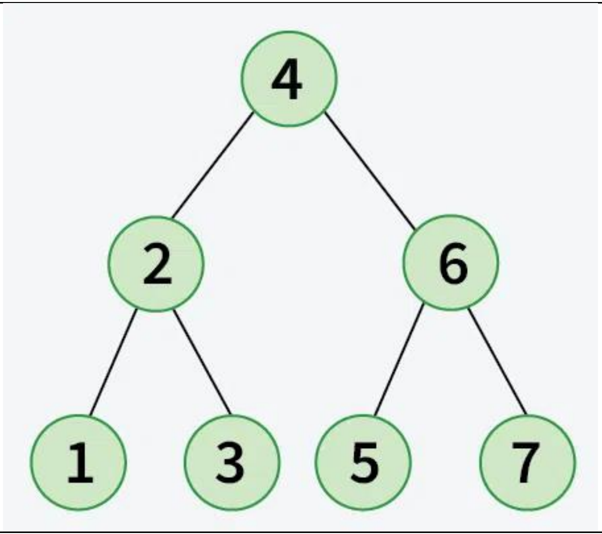
Fall 2025

Lab Content

1. Balance BST
2. AVL Trees
3. Rotation in AVL Trees
4. Insertion, Deletion and Searching in AVL Trees

Balance in Binary Search Tree:

A balanced binary tree is a type of tree in data structure used to keep data sorted and easy to search. In this tree, the left and right sides are kept at nearly the same height. This balance helps to make sure that no side of the tree is too deep, which can slow down data operations.

	
Unbalanced Binary Search Tree	**Balanced Binary Search Tree**

Approach:

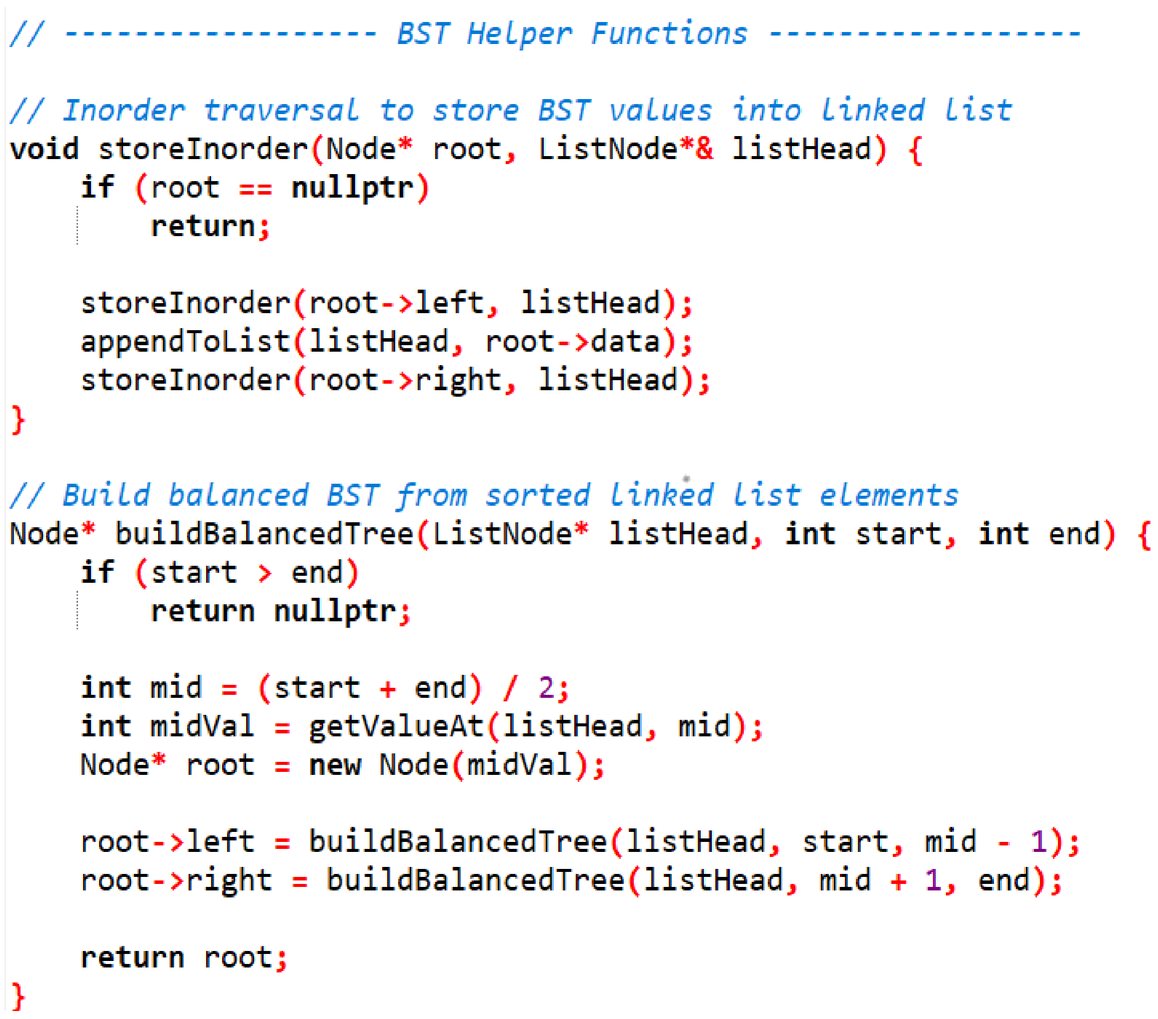
The idea is to store the elements of the tree in an array using inorder traversal. Inorder traversal of a BST produces a sorted array. Once we have a sorted array, recursively construct a balanced BST by picking the middle element of the array as the root for each subtree.

Follow the steps below to solve the problem:

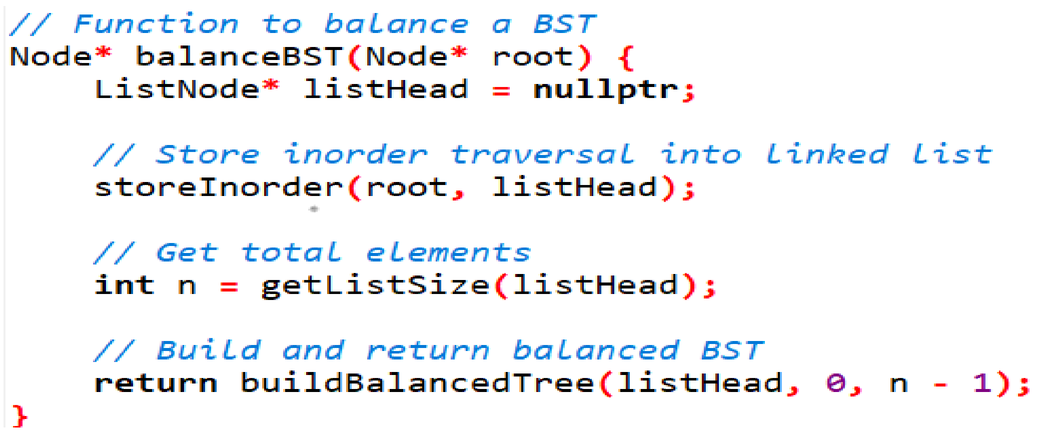
1. Traverse given BST in inorder and store result in an array. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.
2. Build a balanced BST from the above created sorted array using the recursive approach discussed in Sorted Array to Balanced BST.

```
// ----------------- Node Definition -----------------
class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// ----------------- Linked List Definition -----------------
class ListNode {
public:
    int data;
    ListNode* next;

    ListNode(int value) {
        data = value;
        next = nullptr;
    }
};
```

```
// Append node to Linked List
void appendToList(ListNode*& head, int value) {
    if (head == nullptr) {
        head = new ListNode(value);
        return;
    }
    ListNode* curr = head;
    while (curr->next != nullptr)
        curr = curr->next;
    curr->next = new ListNode(value);
}

// Get size of Linked List
int getListSize(ListNode* head) {
    int count = 0;
    while (head != nullptr) {
        count++;
        head = head->next;
    }
    return count;
}
```



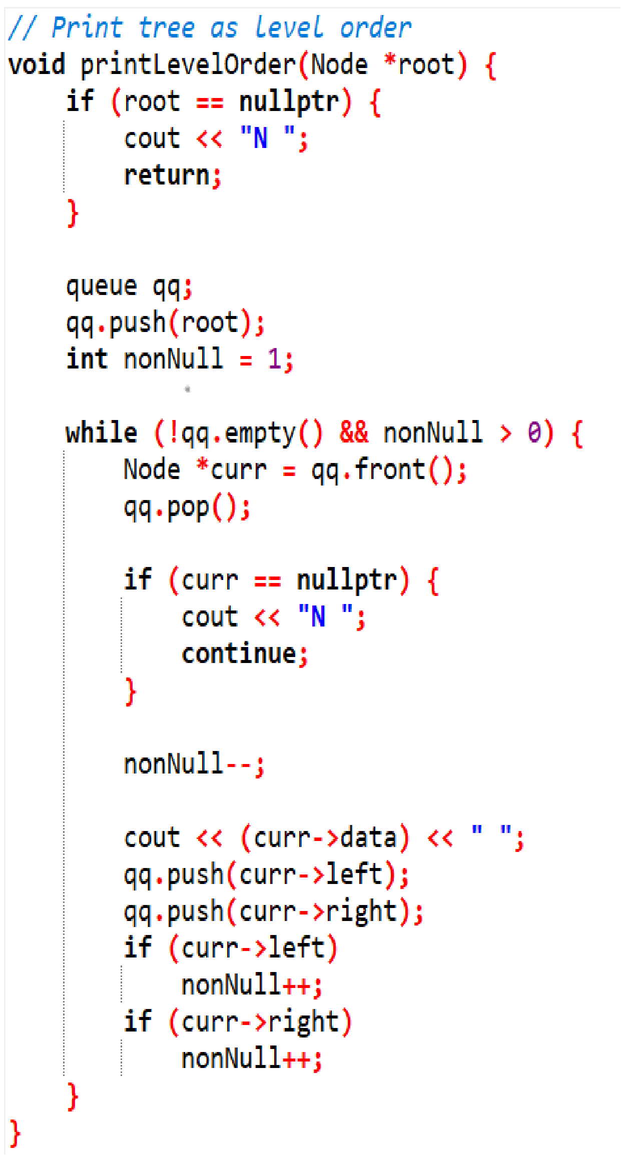
```
// ------------------ BST Helper Functions ------------------

// Inorder traversal to store BST values into linked list
void storeInorder(Node* root, ListNode*& listHead) {
    if (root == nullptr)
        return;

    storeInorder(root->left, listHead);
    appendToList(listHead, root->data);
    storeInorder(root->right, listHead);
}

// Build balanced BST from sorted linked list elements
Node* buildBalancedTree(ListNode* listHead, int start, int end) {
    if (start > end)
        return nullptr;

    int mid = (start + end) / 2;
    int midVal = getValueAt(listHead, mid);
    Node* root = new Node(midVal);

    root->left = buildBalancedTree(listHead, start, mid - 1);
    root->right = buildBalancedTree(listHead, mid + 1, end);

    return root;
}
```



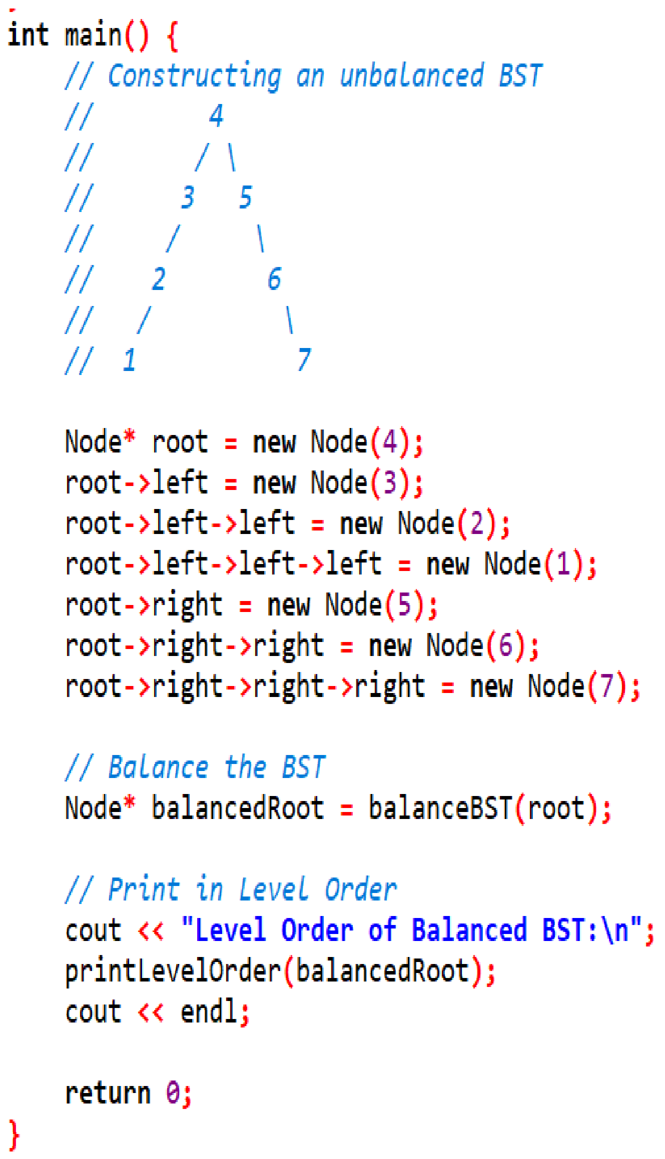
```
// Function to balance a BST
Node* balanceBST(Node* root) {
    ListNode* listHead = nullptr;

    // Store inorder traversal into linked list
    storeInorder(root, listHead);

    // Get total elements
    int n = getListSize(listHead);

    // Build and return balanced BST
    return buildBalancedTree(listHead, 0, n - 1);
}
```

Level Order Traversal using Queue:

When you traverse a tree breadth-first, the breadth gets broader. You need a queue to keep track of the roots of subtrees to visit more deeply next, like a "to do" list. Hence, we are doing Level Order Traversal through a queue.



```cpp
// Print tree as level order
void printLevelOrder(Node *root) {
    if (root == nullptr) {
        cout << "N ";
        return;
    }

    queue qq;
    qq.push(root);
    int nonNull = 1;

    while (!qq.empty() && nonNull > 0) {
        Node *curr = qq.front();
        qq.pop();

        if (curr == nullptr) {
            cout << "N ";
            continue;
        }

        nonNull--;

        cout << (curr->data) << " ";
        qq.push(curr->left);
        qq.push(curr->right);
        if (curr->left)
            nonNull++;
        if (curr->right)
            nonNull++;
    }
}
```

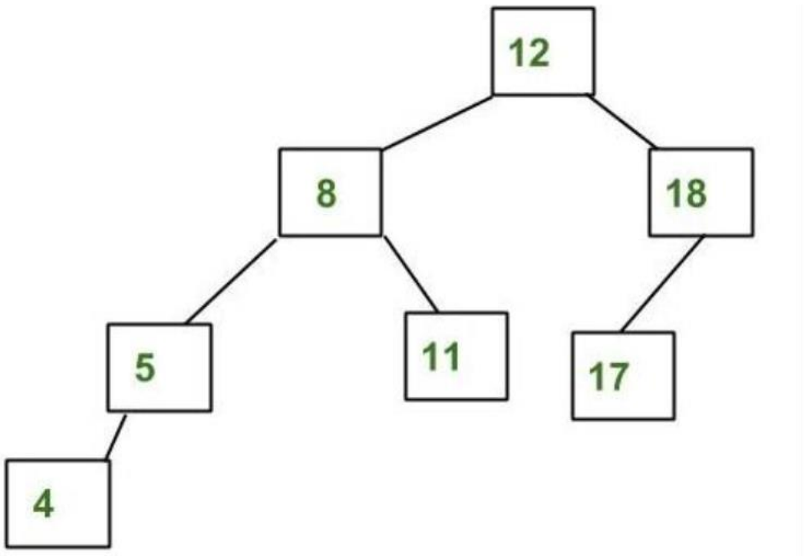


```cpp
int main() {
    // Constructing an unbalanced BST
    //         4
    //        / \
    //       3   5
    //      /     \
    //     2       6
    //    /         \
    //   1           7

    Node* root = new Node(4);
    root->left = new Node(3);
    root->left->left = new Node(2);
    root->left->left->left = new Node(1);
    root->right = new Node(5);
    root->right->right = new Node(6);
    root->right->right->right = new Node(7);

    // Balance the BST
    Node* balancedRoot = balanceBST(root);

    // Print in Level Order
    cout << "Level Order of Balanced BST:\n";
    printLevelOrder(balancedRoot);
    cout << endl;

    return 0;
}
```

AVL Tree:

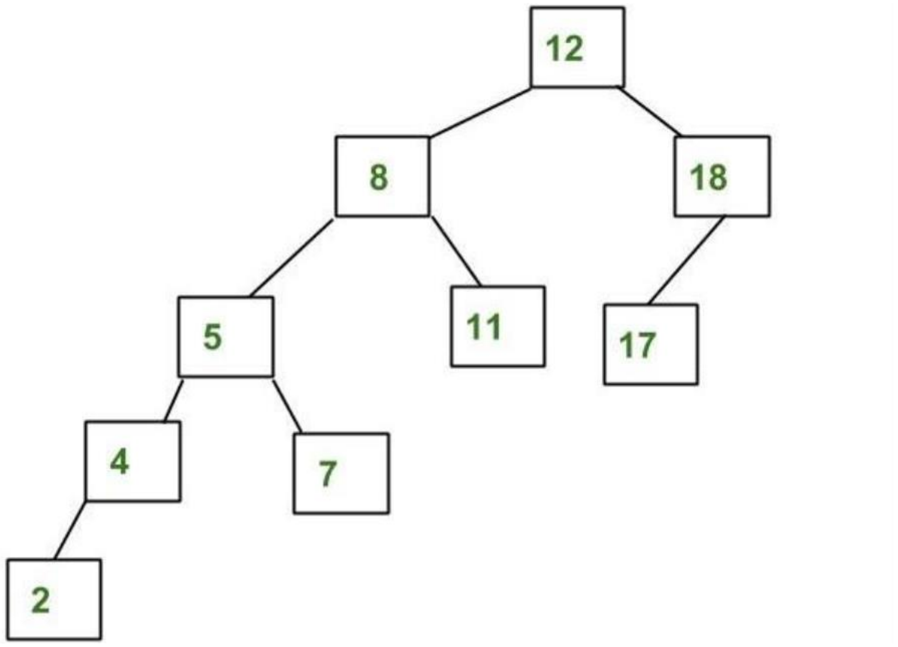
AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Example of AVL Tree:



The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take **O(h)** time where **h** is the height of the BST. The cost of these operations may become **O(n)** for a **skewed Binary tree**. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.

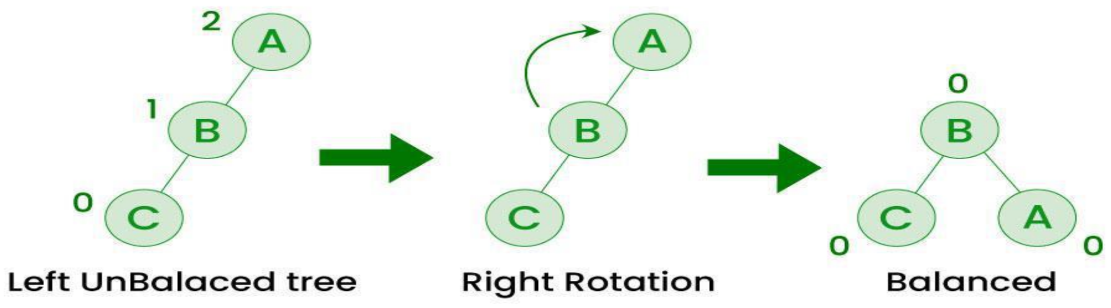
Rotations in AVL Trees:

Rotations are the most important part of the working of the AVL tree. They are responsible for maintaining the balance in the AVL tree. There are 4 types of rotations based on the 4 possible cases:

1. Right Rotation (RR)
2. Left Rotation (LL)
3. Left-Right Rotation (LR)
4. Right-Left Rotation (RL)

Right Rotation (RR)

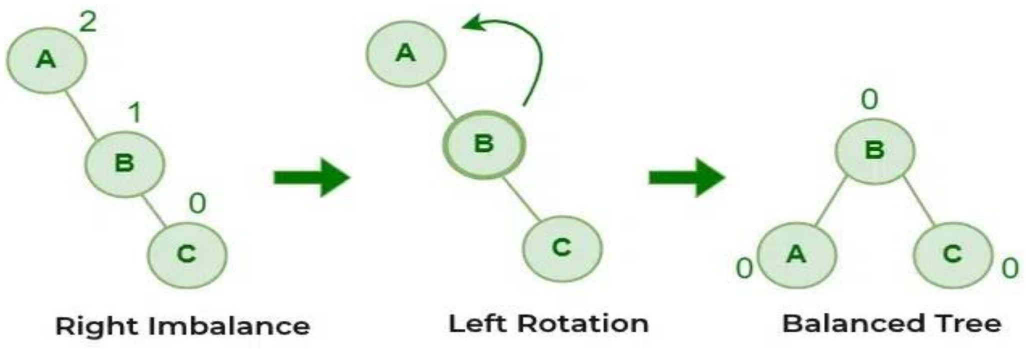
The Right Rotation (RR) is applied in an AVL tree when a node becomes unbalanced due to an insertion into the right subtree of its right child, leading to a Left Imbalance. To correct this imbalance, the unbalanced node is rotated 90° to the right (clockwise) along the top edge connected to its parent.



```
// Right rotation
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x; // new root
}
```

Left Rotation (LL)

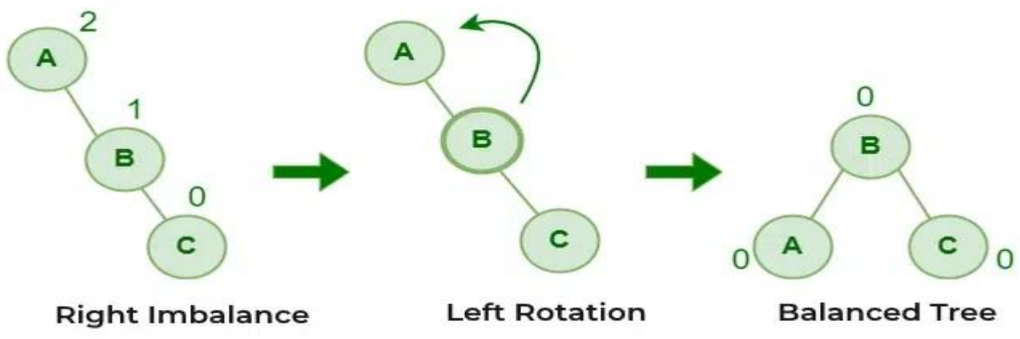
The Left Rotation (LL) is used to balance a node that becomes unbalanced due to an insertion into the left subtree of its left child, also resulting in a Left Imbalance. The solution is to rotate the unbalanced node 90° to the left (anti-clockwise) along the top edge connected to its parent.



```
// Left rotation
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y; // new root
}
```

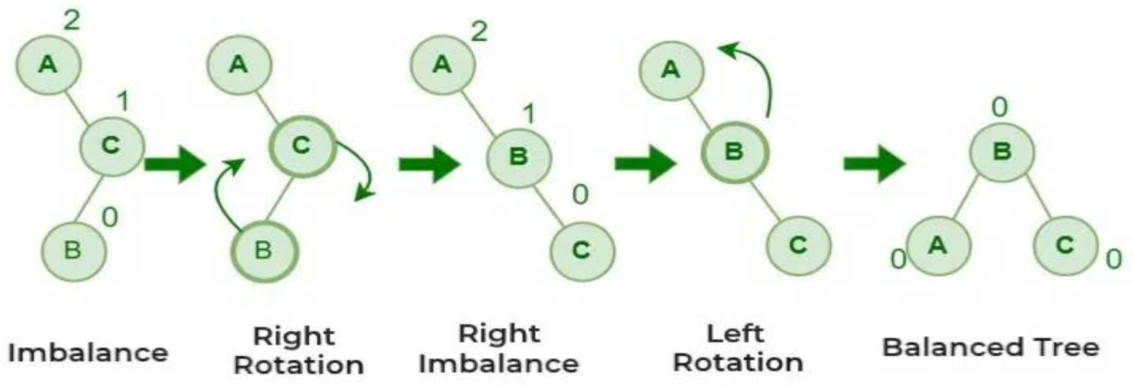
Left-Right Rotation (LR)

The Left-Right Rotation (LR) is necessary when the left child of a node is right-heavy, creating a double imbalance. This situation is resolved by performing a left rotation on the left child, followed by a right rotation on the original node.



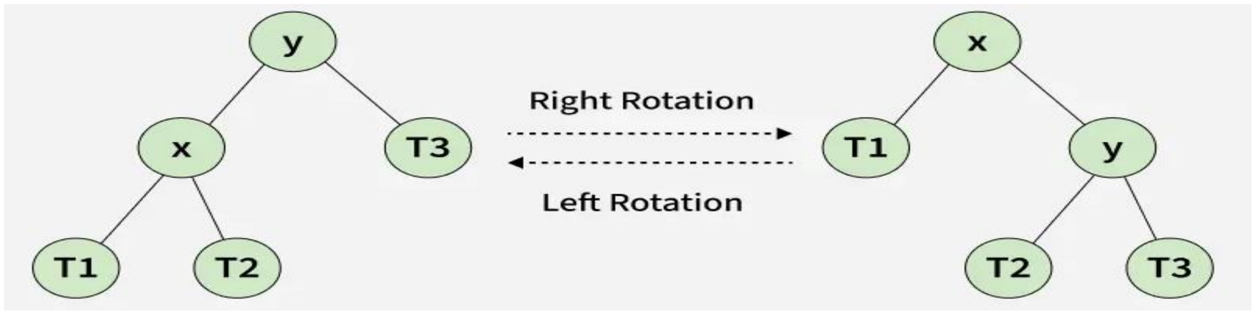
Right-Left Rotation (RL)

The Right-Left Rotation (RL) is used when the right child of a node is left-heavy. This imbalance is corrected by performing a right rotation on the right child, followed by a left rotation on the original node.



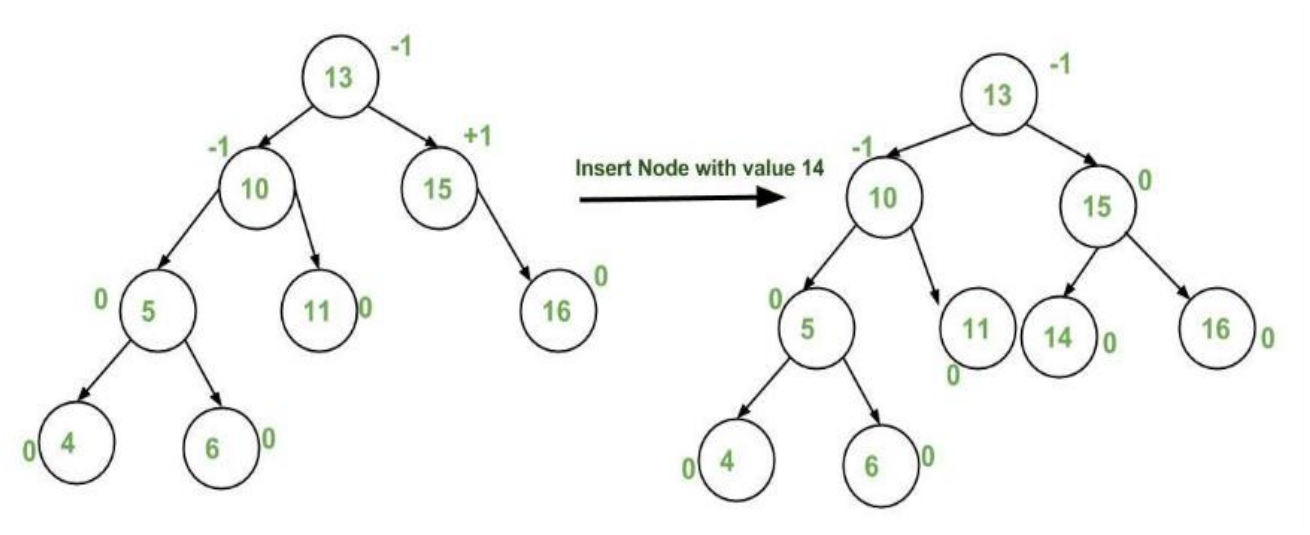
Insertion in AVL Tree:

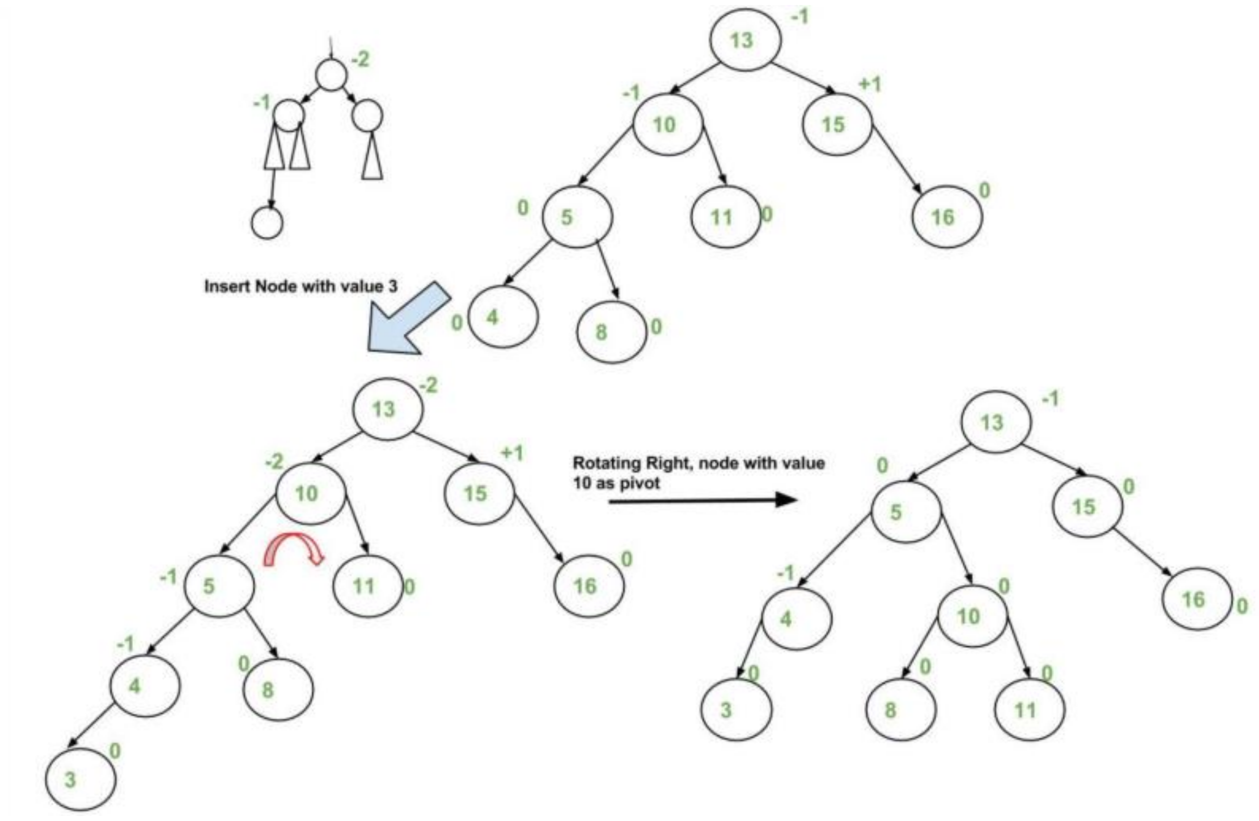
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

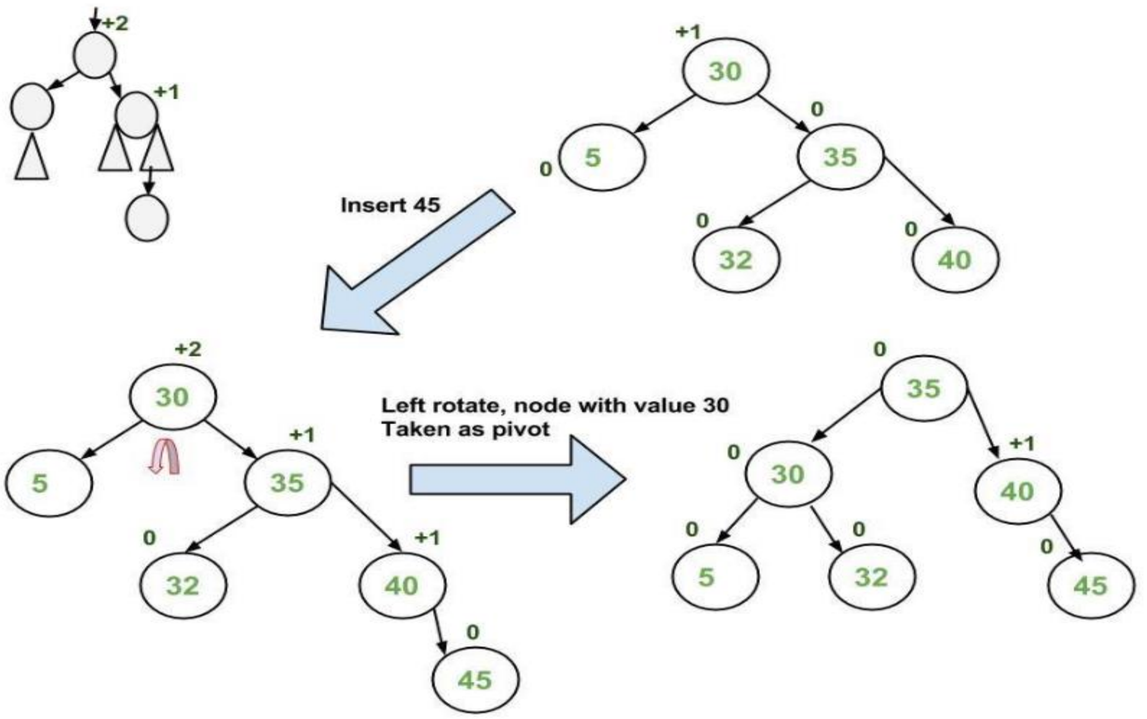


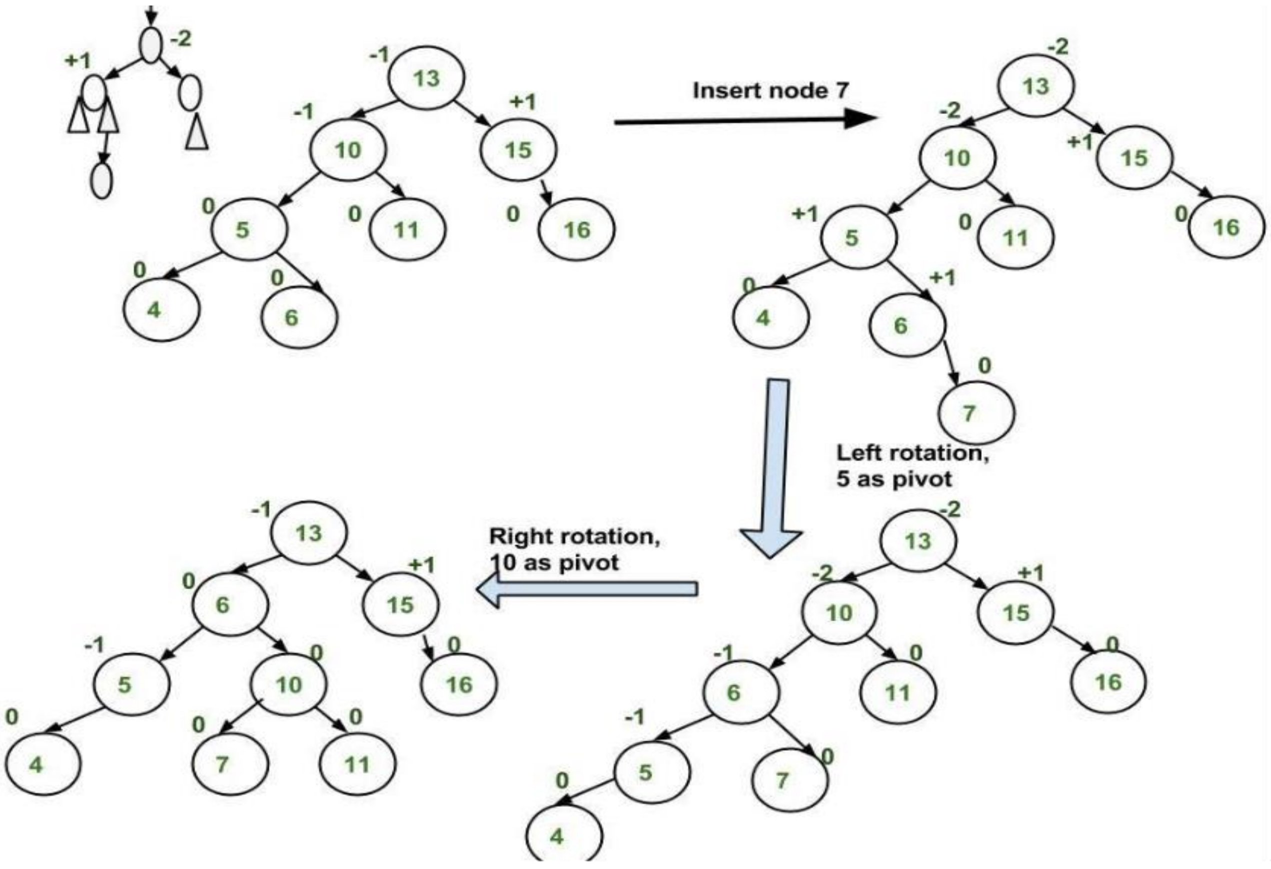
```cpp
// Insert a node
Node* insert(Node* node, int key) {
    // 1. Normal BST insertion
    if (node == nullptr)
        return newNode(key);

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

    // 2. Update height
    node->height = 1 + max(getHeight(node->left), getHeight(node->right));

    // 3. Get balance factor
    int balance = getBalance(node);          // 4. If it is unbalanced then Balance the tree

    // 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; // unchanged
}
```

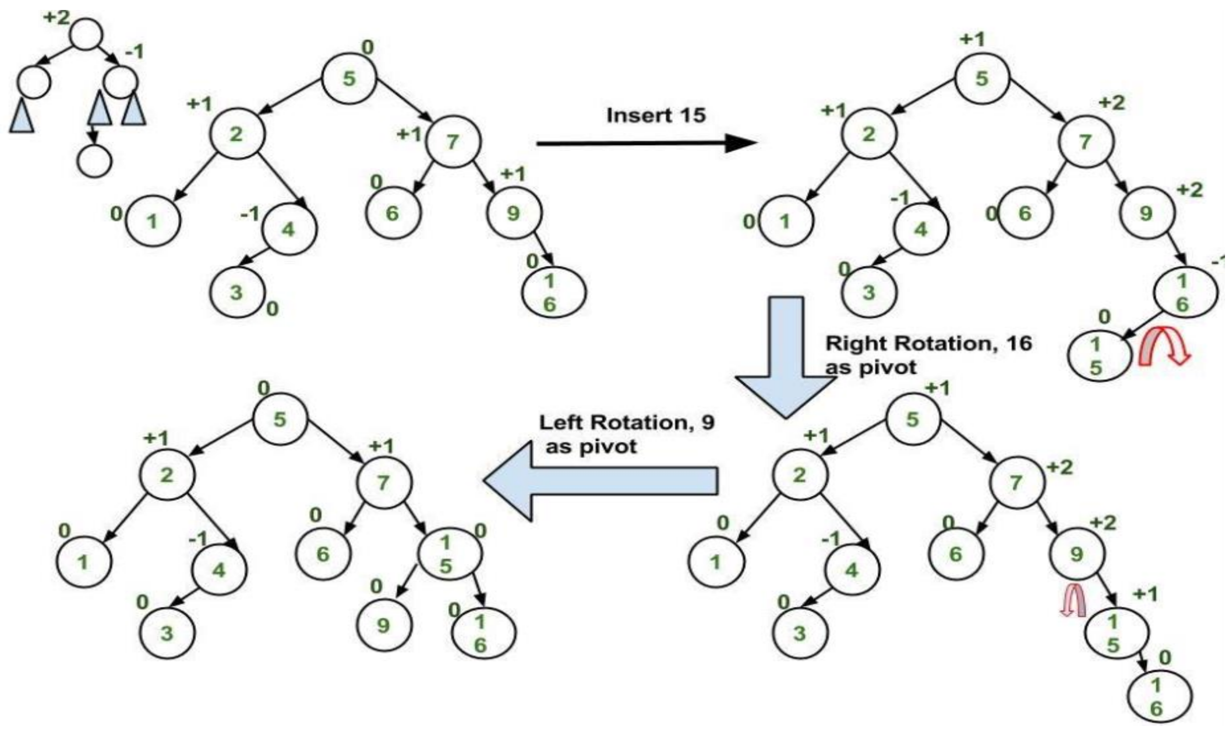
<u>Illustration of Insertion at AVL Tree</u>






Insert 45
Left rotate, node with value 30
Taken as pivot


Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot



Deletion in AVL Tree:

Deletion in an AVL tree involves removing a node and then ensuring the tree remains balanced. After deleting a node, the balance factor of each node is checked, and rotations are performed if necessary to maintain the AVL property.

```
// Delete a node
Node* deleteNode(Node* root, int key) {
    if (root == nullptr)
        return root;

    // Perform standard BST deletion
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // Node with one child or no child
        if ((root->left == nullptr) || (root->right == nullptr)) {
            Node* temp = root->left ? root->left : root->right;

            if (temp == nullptr) { // No child case
                temp = root;
                root = nullptr;
            } else
                *root = *temp; // Copy contents

            delete temp;
        } else {
            // Node with two children
            Node* temp = getMinValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
```

```
        // If tree had only one node
        if (root == nullptr)
            return root;

        // Update height
        root->height = 1 + max(getHeight(root->left), getHeight(root->right));

        // Get balance
        int balance = getBalance(root);

        // Balance the tree

        // Left Left Case
        if (balance > 1 && getBalance(root->left) >= 0)
            return rightRotate(root);

        // Left Right Case
        if (balance > 1 && getBalance(root->left) < 0) {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }

        // Right Right Case
        if (balance < -1 && getBalance(root->right) <= 0)
            return leftRotate(root);

        // Right Left Case
        if (balance < -1 && getBalance(root->right) > 0) {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }

        return root;
    }
```

Searching in AVL Tree:

Searching in an AVL tree is similar to searching in a binary search tree (BST). Since AVL trees are balanced, searching is efficient with a time complexity of O(log n).

```
// Search for a key in AVL Tree
bool AVLsearch(Node* root, int key) {
    if (root == nullptr )
        return false;
    else if(root->key == key)
        return true;
    else if (key < root->key)
        bool val = AVLsearch(root->left, key);
        return val;
    else
        bool val = AVLsearch(root->right, key);
        return val;
}
```