Data Structures

Course: Data Structures (CS2001) Semester: Fall 2023 Instructor: Syeda Mahnoor Javed

| **Binary Search Tree** |
| --- |

Task-1: A Binary Search Tree (BST) is a binary tree with the following properties:

→ The left subtree of a particular node will always contain nodes whose keys are less than

that node’s key.

→The right subtree of a particular node will always contain nodes with keys greater than

that node’s key. The left and right subtree of a particular node will also, in turn, be binary

search trees

* Insert a node
* Delete a node
* Search a node by key

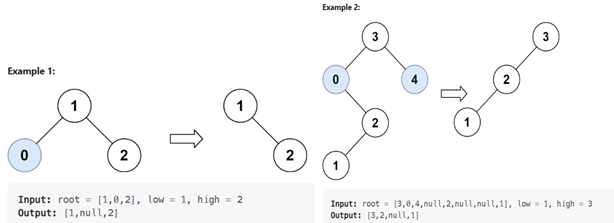
Task-2:

Use the above binary tree and perform traversal(Traverse(in-order,pre-order,post-order)).

Task-3:

Write recursive functions that perform preorder and inorder tree walks.

Task 04: In a sprawling orchard game, a grand tree stands burdened by tangled branches, each heavy with fruit. But amid this abundance, a solitary node bears the mark of zero. This lone zero inspire the orchard keepers to action, prompting them to rearrange the tree's branches? By ensuring every parent has exactly two children, can they breathe new life into the tree, revitalizing the orchard's promise of bountiful harvests. And also find out if the given tree is a binary search tree?

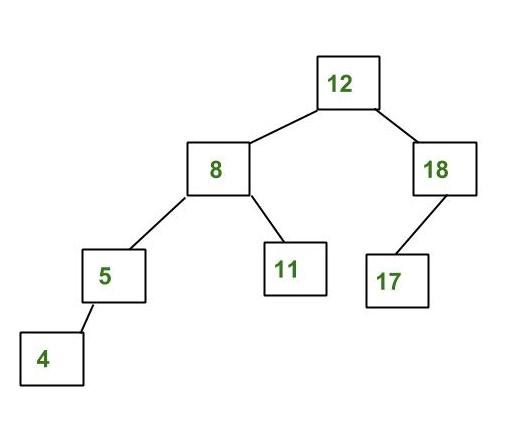


| **AVL Tree** |
| --- |

**AVL Tree:**

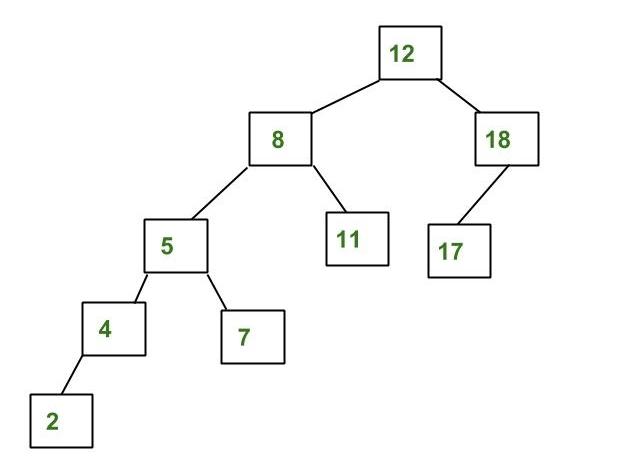
AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

**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.*

## **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)).

* Left Rotation
* Right Rotation

T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

y x

/ \ Right Rotation / \

x T3 - - - - - - - > T1 y

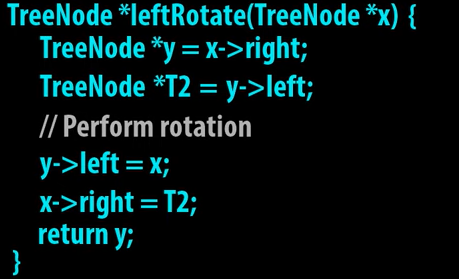
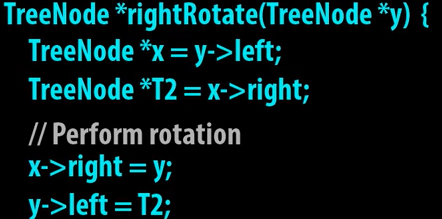
/ \ < - - - - - - - / \

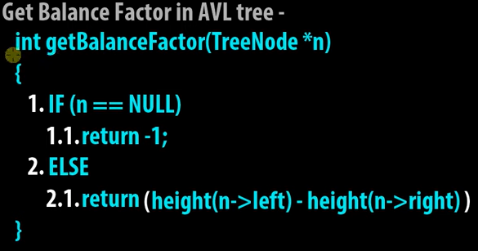
T1 T2 Left Rotation T2 T3

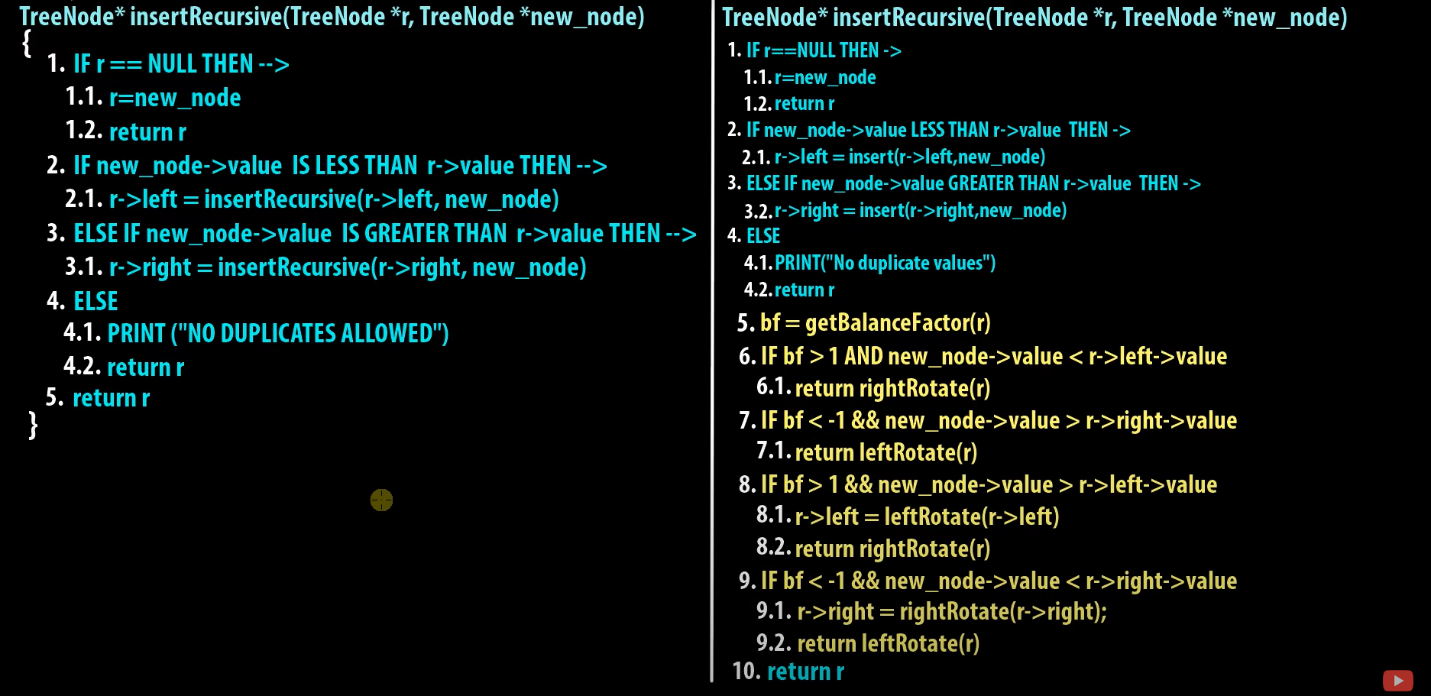
Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.





**BST vs AVL** 

**1. Left Left Case**

T1, T2, T3 and T4 are subtrees.

z y

/ \ / \

y T4 Right Rotate (z) x z

/ \ - - - - - - - - -> / \ / \

x T3 T1 T2 T3 T4

/ \

T1 T2

**2. Left Right Case**

z z x

/ \ / \ / \

y T4 Left Rotate (y) x T4 Right Rotate(z) y z

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

T1 x y T3 T1 T2 T3 T4

/ \ / \

T2 T3 T1 T2

**3. Right Right Case**

z y

/ \ / \

T1 y Left Rotate(z) z x

/ \ - - - - - - - -> / \ / \

T2 x T1 T2 T3 T4

/ \

T3 T4

**4. Right Left Case**

z z x

/ \ / \ / \

T1 y Right Rotate (y) T1 x Left Rotate(z) z y

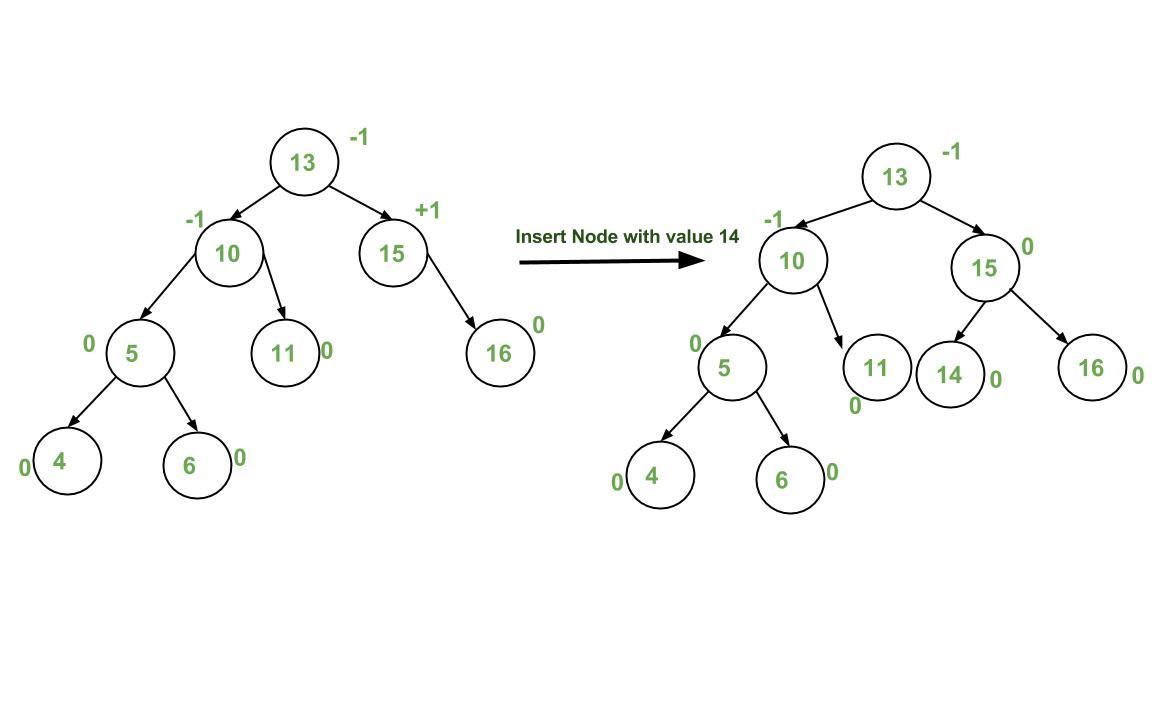
/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

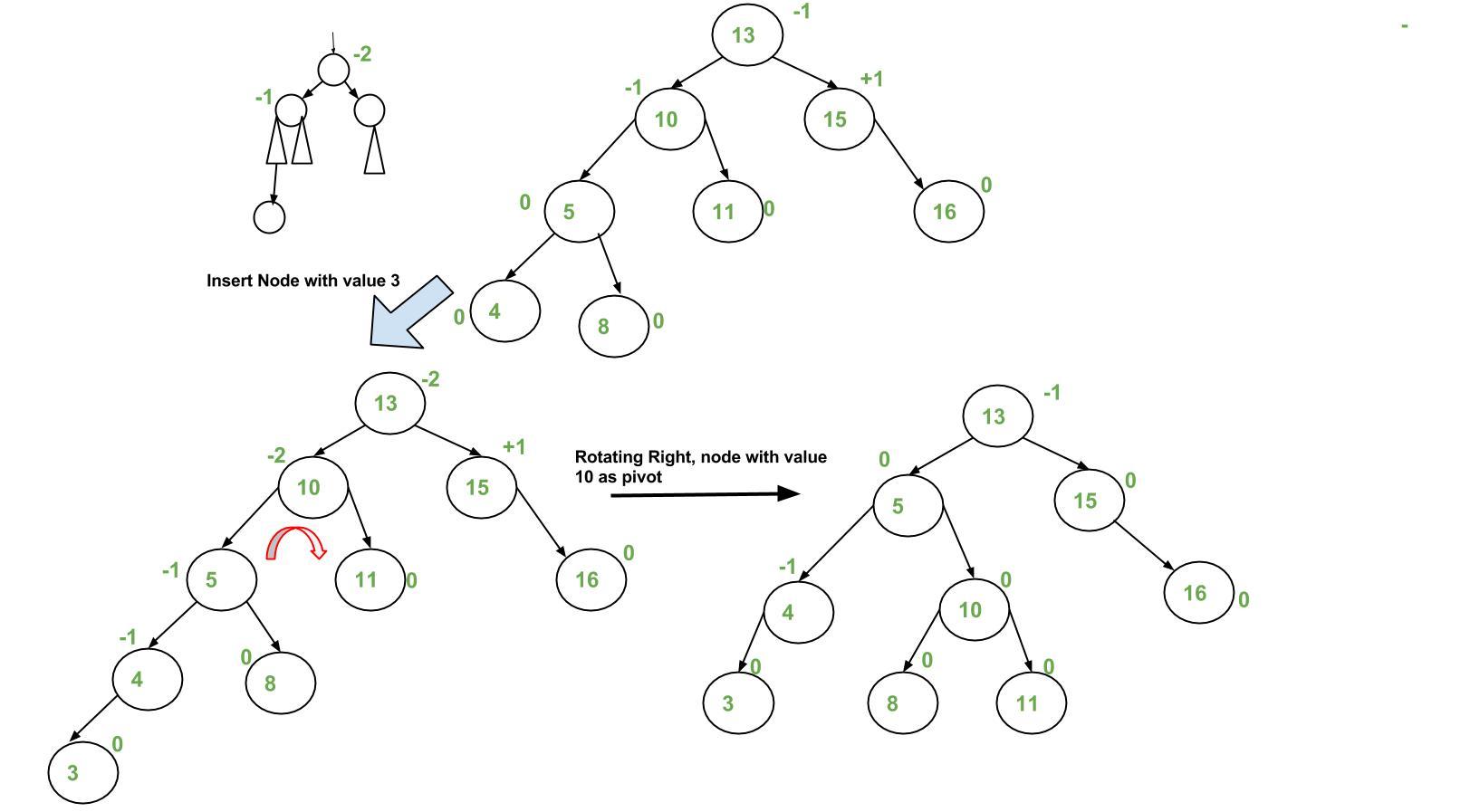
x T4 T2 y T1 T2 T3 T4

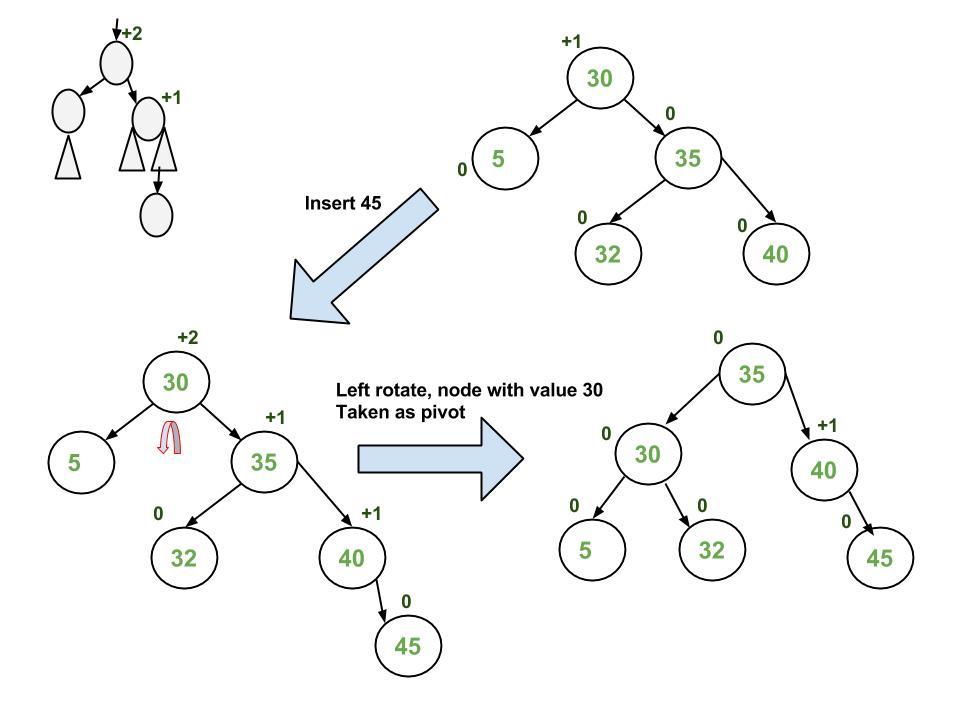
/ \ / \

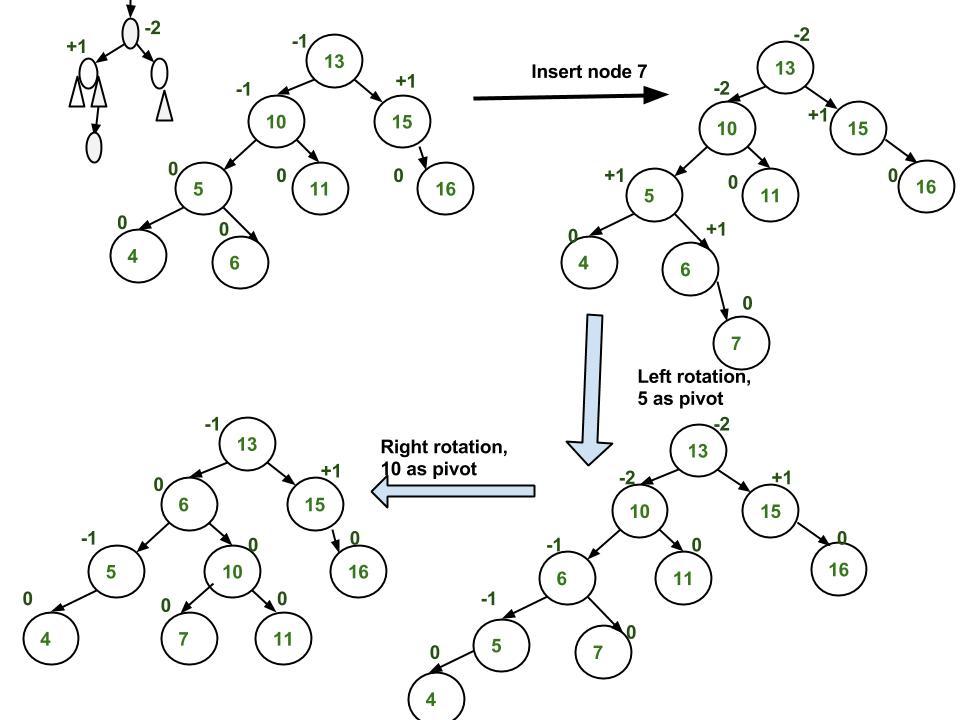
T2 T3 T3 T4

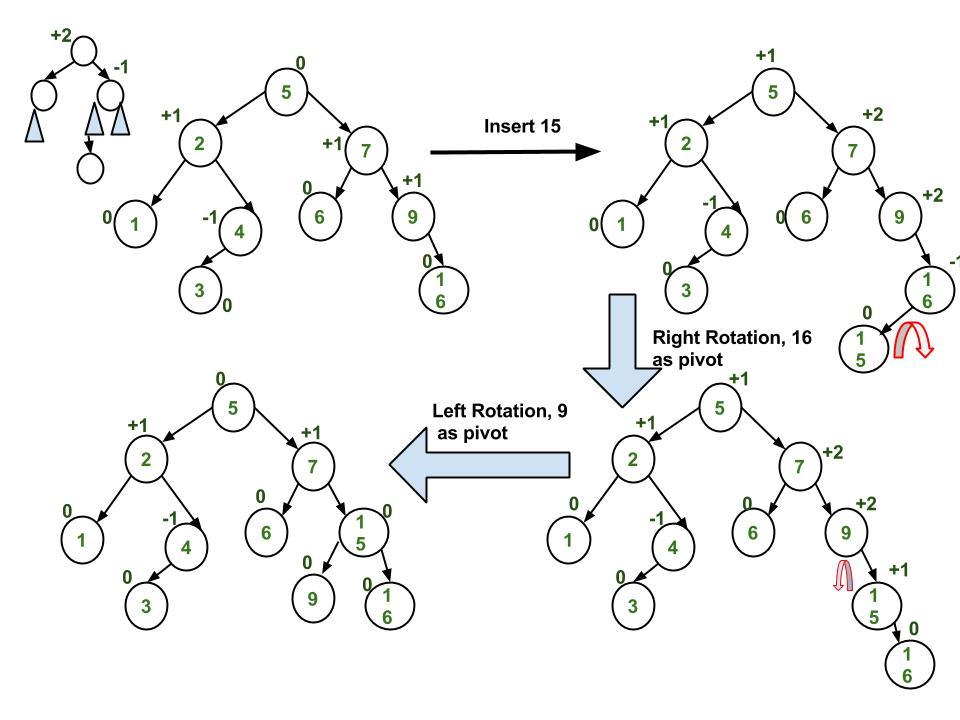
**Illustration of Insertion at AVL Tree**











**Approach:**

*The idea is to use a recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don’t need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.*

Follow the steps mentioned below to implement the idea:

* Perform the normal[BST insertion.](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)
* The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
* Get the balance factor**(left subtree height – right subtree height)** of the current node.
* If the balance factor is greater than **1,** then the current node is unbalanced and we are either in the**Left Left case** or**left Right case**. To check whether it is**left left case**or not, compare the newly inserted key with the key in the**left subtree root**.
* If the balance factor is less than**-1**, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root.

**Task-5:** Suppose you are building an application that stores student records in an AVL tree based on their roll number. You want to insert a new student with roll number 15 into the AVL tree, and you want to ensure that the tree is balanced using left rotation.

Assuming the AVL tree is already populated with the following student records:

Student 1 with roll number 2

Student 2 with roll number 30

Student 3 with roll number 38

Student 4 with roll number 78

Student 5 with roll number 92

* Walk me through the process of inserting the new student record with roll number 15 into the AVL tree.
* After inserting the new student record, what will be the height of the AVL tree?
* Using the left rotation operation, show the resulting AVL tree after inserting the new student record.

**Task-6:** Suppose you have an AVL tree with the following elements: 50, 30, 70, 20, 40, 60, 80. You need to insert a new node with value 55 into the tree and then display the tree after performing a left rotation on the root.

**Task-7:**

Construct a height-balanced BST from a sorted doubly linked list

Input:

Doubly Linked List: 8 10 12 15 18 20 25

Output:

