**Data Structures Lab 09**

**Course:** Data Structures (CL2001) **Semester:** Fall 2024

**Instructor:** Shafique Rehman

**Note:**

* + - * Lab manual cover following below Stack and Queue topics

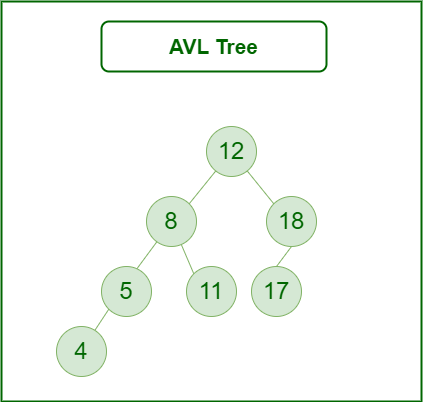
**{Self-balancing Binary Trees, AVL Trees, 2-3 Trees}**

* Maintain discipline during the lab.
* Just raise hand if you have any problem.
* Completing all tasks of each lab is compulsory.
* Get your lab checked at the end of the session.
* Don’t just blatantly copy the same code make changes to it accordingly

# **AVL Tree Data Structure**

An **AVL tree** defined as a self-balancing [**Binary Search Tree**](https://www.geeksforgeeks.org/binary-search-tree-data-structure/) (BST) where the difference between heights of left and right subtrees for any node cannot be more than one. The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node. The AVL tree is named after its inventors, Georgy **A**delson-**V**elsky and Evgenii **L**andis, who published it in their 1962 paper “An algorithm for the organization of information”.

### Example of AVL Trees:



**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. Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again.

**BALANCE FACTOR = HEIGHT(LEFT SUBTREE) – HEIGHT(RIGHT SUBTREE)**

### Operations on an AVL Tree:

* Insertion
* Deletion
* Searching [It is similar to performing a search in BST]

### Rotating the subtrees in an AVL Tree while inserting:

The AVL Tree may rotate in one of the following four ways to keep itself balanced:

**Left Rotation**:

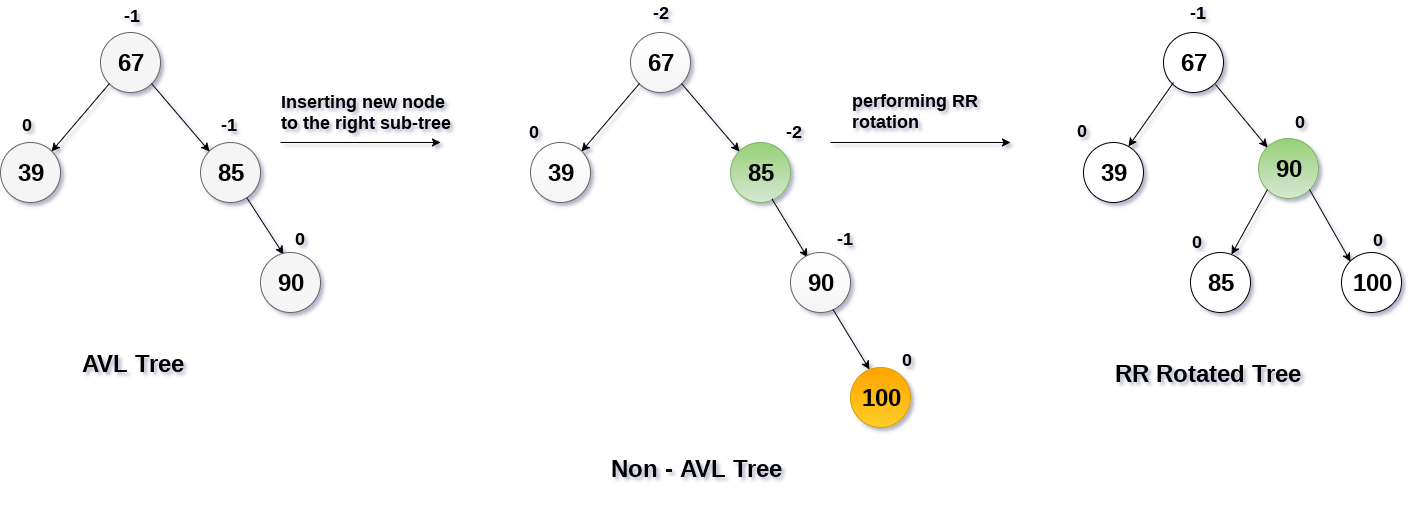
When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.

A diagram of a tree and a tree

Description automatically generated

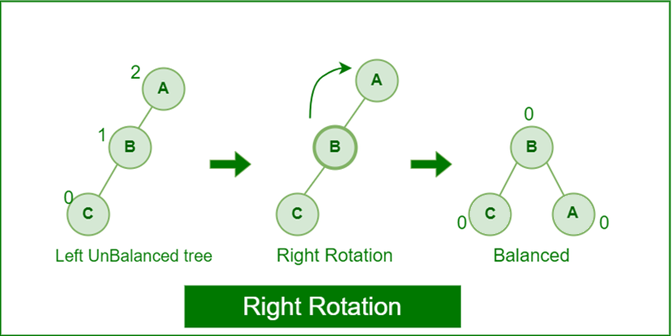
**Left-Rotation in AVL tree**

**Example:**



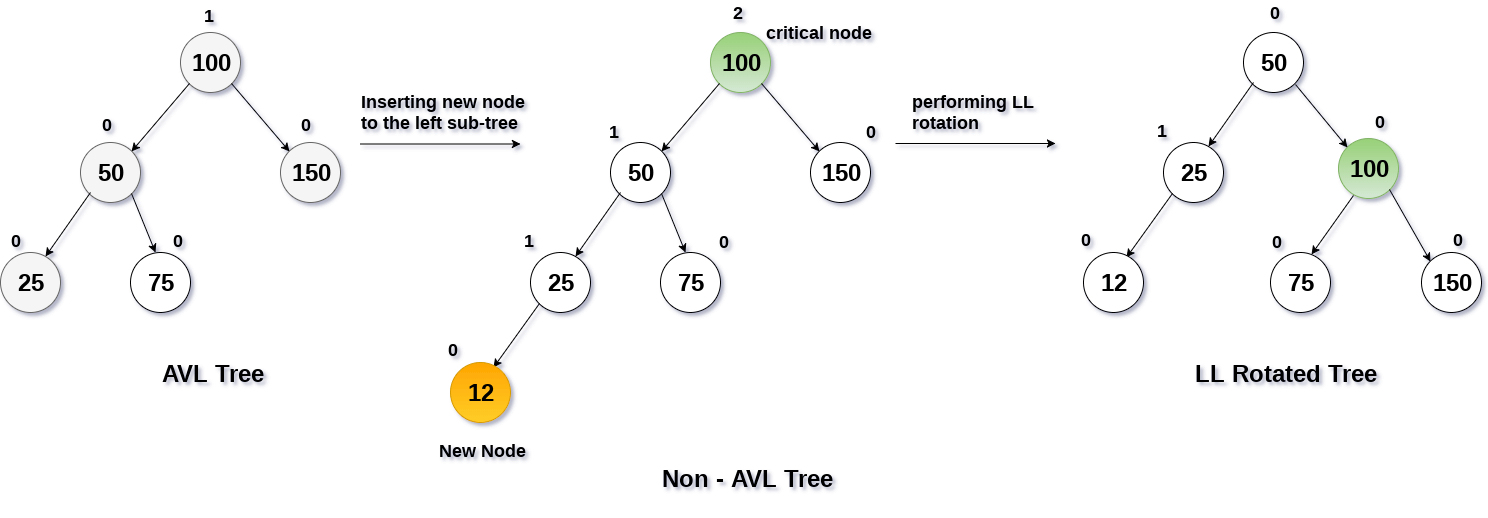
**Right Rotation:**

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



**Right Rotation in AVL tree**

**Example:**



**Left-Right Rotation**:

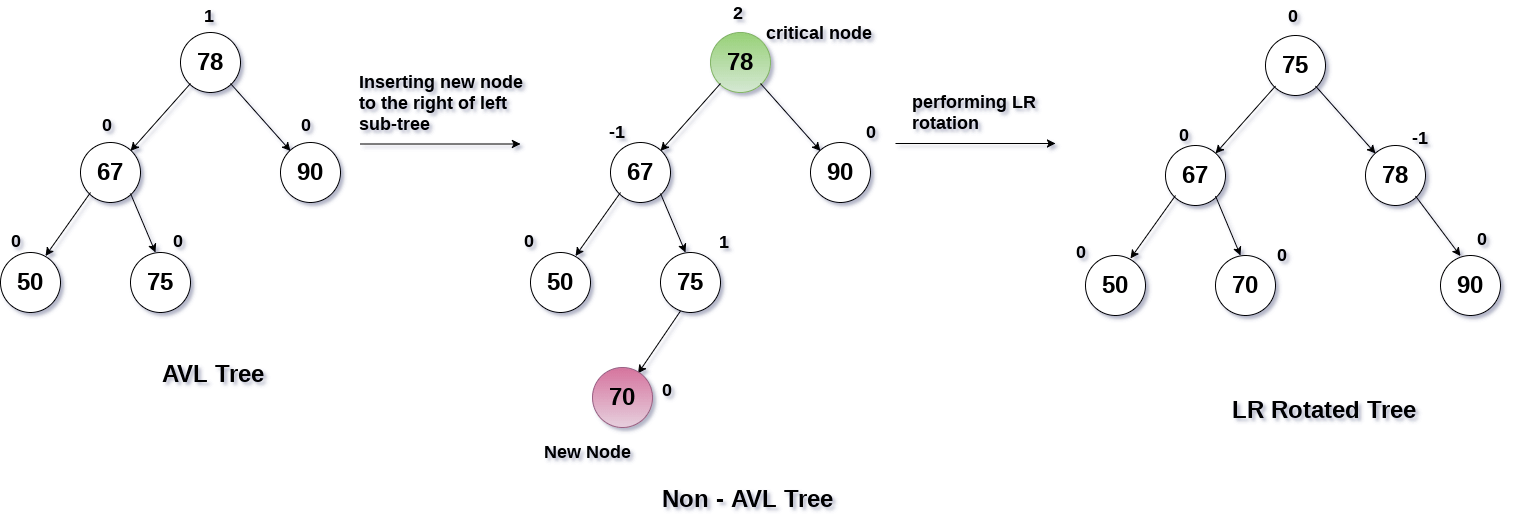
A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.

A diagram of a left-right rotation

Description automatically generated

**Left-Right Rotation in AVL tree**

**Example:**



**Right-Left Rotation:**

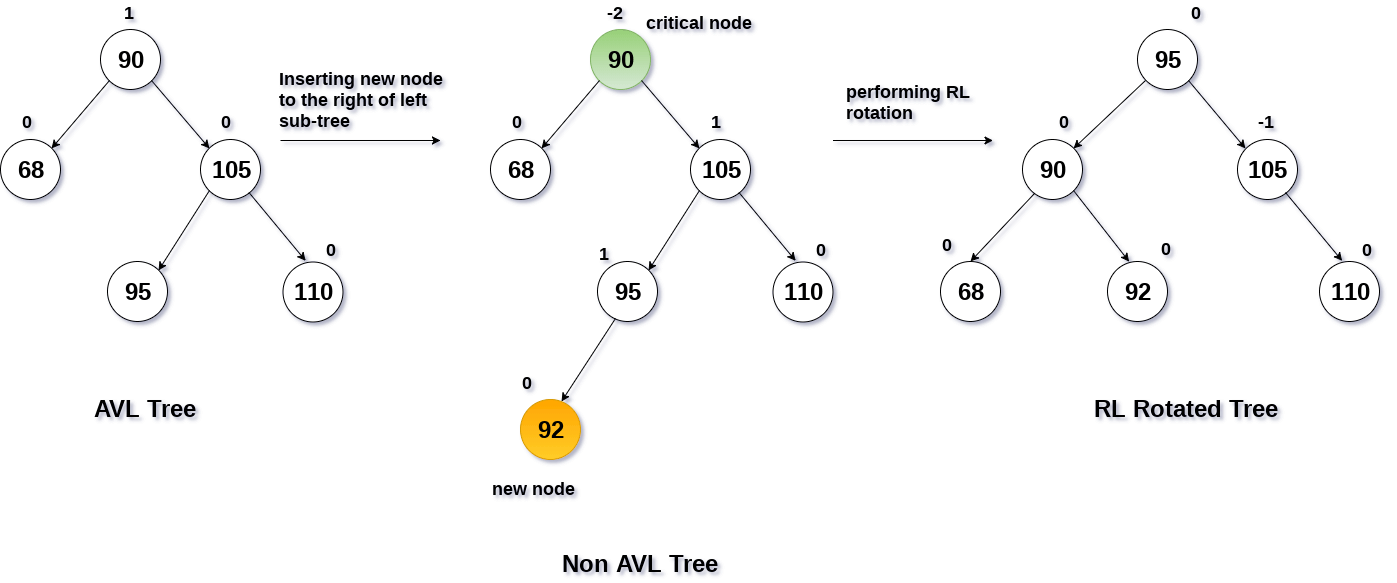
A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.

A diagram of a diagram of a right-left rotation

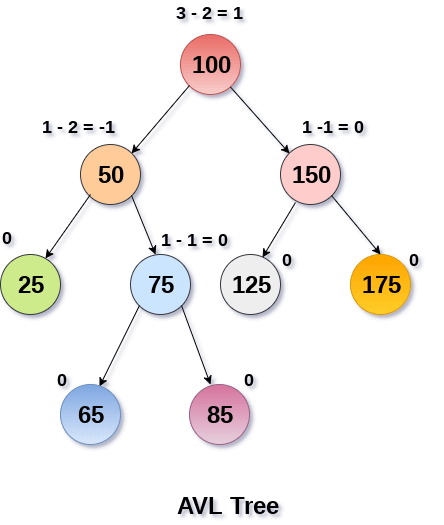
Description automatically generated

**Right-Left Rotation in AVL tree**

**Example:**



An **AVL tree** is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



### Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. Better searching time complexity compared to other trees like binary tree.
4. Height cannot exceed log(N), where, N is the total number of nodes in the tree.

### Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
4. Take more processing for balancing.

**Case for Insertion:**

START

if node == null then:

return new node

if key < node.key then:

node.left = insert (node.left, key)

else if (key > node.key) then:

node.right = insert (node.right, key)

else

return node

node.height = 1 + max (height (node.left), height (node.right))

balance = getBalance (node)

if balance > 1 and key < node.left.key then:

rightRotate

if balance < -1 and key > node.right.key then:

leftRotate

if balance > 1 and key > node.left.key then:

node.left = leftRotate (node.left)

rightRotate

if balance < -1 and key < node.right.key then:

node.right = rightRotate (node.right)

leftRotate (node)

return node

END

### ****Steps to follow for insertion:****

Let the newly inserted node be **w**

* Perform standard**BST** insert for **w**.
* Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y**be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild**of **z** that comes on the path from **w**to **z**.
* Re-balance the tree by performing appropriate rotations on the subtree rooted with**z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
* Following are the possible 4 arrangements:
  + y is the left child of z and x is the left child of y (Left Left Case)
  + y is the left child of z and x is the right child of y (Left Right Case)
  + y is the right child of z and x is the right child of y (Right Right Case)
  + y is the right child of z and x is the left child of y (Right Left Case)

**Case for Deletion**

START

if root == null: return root

if key < root.key:

root.left = delete Node

else if key > root.key:

root.right = delete Node

else:

if root.left == null or root.right == null then:

Node temp = null

if (temp == root.left)

temp = root.right

else

temp = root.left

if temp == null then:

temp = root

root = null

else

root = temp

else:

temp = minimum valued node

root.key = temp.key

root.right = delete Node

if (root == null) then:

return root

root.height = max (height (root.left), height (root.right)) + 1

balance = getBalance

if balance > 1 and getBalance (root.left) >= 0:

rightRotate

if balance > 1 and getBalance (root.left) < 0:

root.left = leftRotate (root.left);

rightRotate

if balance < -1 and getBalance (root.right) <= 0:

leftRotate

if balance < -1 and getBalance (root.right) > 0:

root.right = rightRotate (root.right);

leftRotate

return root

END

### ****Steps to follow for Deletion:****

1. Perform the normal BST deletion.
2. The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3. Get the balance factor (left subtree height – right subtree height) of the current node.
4. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

**Exercise**

**Task -0:** Why AVL? Explain in just two lines.

Given the BST **(A and B)** below, write a function to convert these BST into an AVL tree. The function should take the root of the BST as input and perform the conversion. Write only the conversion function, with no insertion logic needed. You may hard-code the given BST for this task.

|  |  |
| --- | --- |
| **A** | **B** |
| A diagram of a diagram  Description automatically generated | A diagram of a diagram  Description automatically generated |

After converting the above BST to an AVL tree, explain each step of the code: how nodes are linked to form the AVL, the types of temporary pointers used in rotation functions, and how specific nodes (e.g., x, y, t2) correspond to the original tree nodes. Additionally, specify the type of rotation performed (right, left, or double rotation), and explain in 3-4 lines which node is returned at the end of the function and why.

**Task-1:** 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 10

Student 2 with roll number 20

Student 3 with roll number 30

Student 4 with roll number 40

Student 5 with roll number 50

* 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-2:**

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. Write a C++ code to accomplish this task using the AVLTree class and its methods.

**Task-3:**

Suppose you have an AVL tree with the following nodes: 10, 5, 15, 3, and 7 (in this order). Perform the following steps:

1. Insert the value 12 into the tree.
2. Calculate the balance factor for each node.
3. Check the height of the tree.
4. If the tree becomes unbalanced during insertion, implement the rotations required to maintain balance.
5. Provide the final balanced AVL tree.

**Task-4:** Find the kth smallest and largest value in the AVL tree and print its key also print both the leftside and right side height of the tree starting from root.