|  |  |
| --- | --- |
| **CL2001**  **Data Structures Lab**  [DS] | **Lab 09**  **Balance in Binary Search Trees, AVL Trees with all operations** |

**National University of Computer and Emerging Sciences**

**Fall 2025**

**Lab Content**

[Balance in Binary Search Tree:](#_Toc1456374273)

[AVL Tree:](#_Toc1912042948)

[Example of AVL Tree:](#_Toc1665661278)

[Example of a Tree that is NOT an AVL Tree:](#_Toc2036036692)

[Why AVL Trees?](#_Toc952906289)

[Rotations in AVL Trees:](#_Toc1098861418)

[Right Rotation (RR)](#_Toc1885848984)

[Left Rotation (LL)](#_Toc1559447771)

[Left-Right Rotation (LR)](#_Toc458691085)

[Right-Left Rotation (RL)](#_Toc1986746372)

[Insertion in AVL Tree:](#_Toc351972746)

[Illustration of Insertion at AVL Tree](#_Toc1920136265)

[Deletion in AVL Tree:](#_Toc1306242220)

[Searching in AVL Tree:](#_Toc1924647165)

[LAB TASKS](#_Toc206866053)

# **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.

In a balanced binary tree, each node has up to two children. The tree is arranged so that the height difference between the left and right sides of any node is no more than one. This means the tree stays short and wide rather than tall and skinny, which helps in quick searching, adding, and removing of data.

# 

# **AVL Tree:**

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference

between heights of left and right subtrees for any node cannot be more than one. The

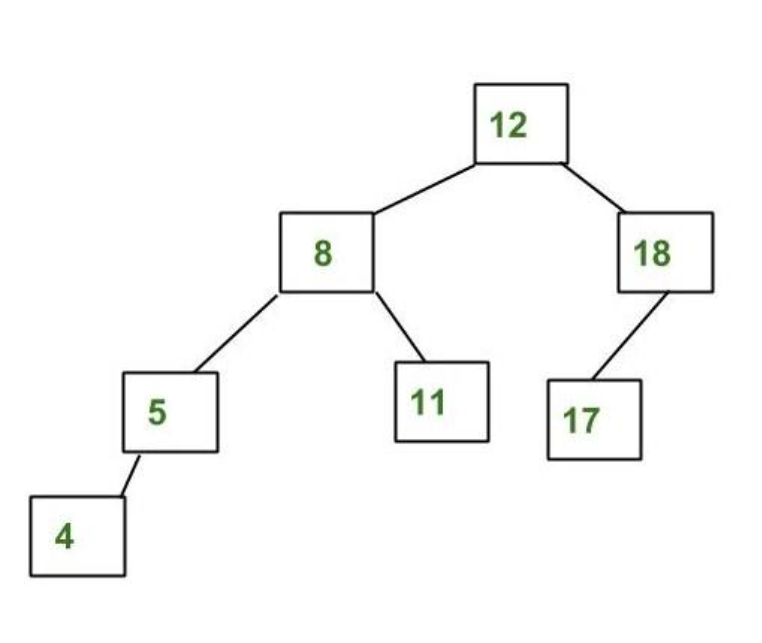
difference between the heights of the left subtree and the right subtree for any node is

known as the balance factor of the node. The AVL tree is named after its inventors, Georgy

Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the

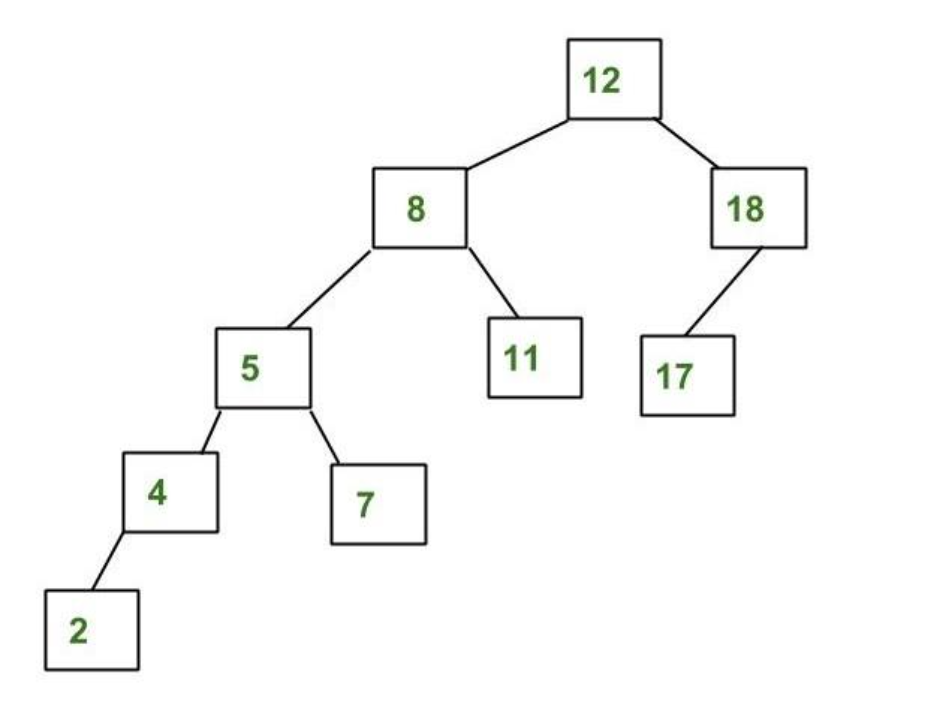
organization of information”.

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.*

**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:**

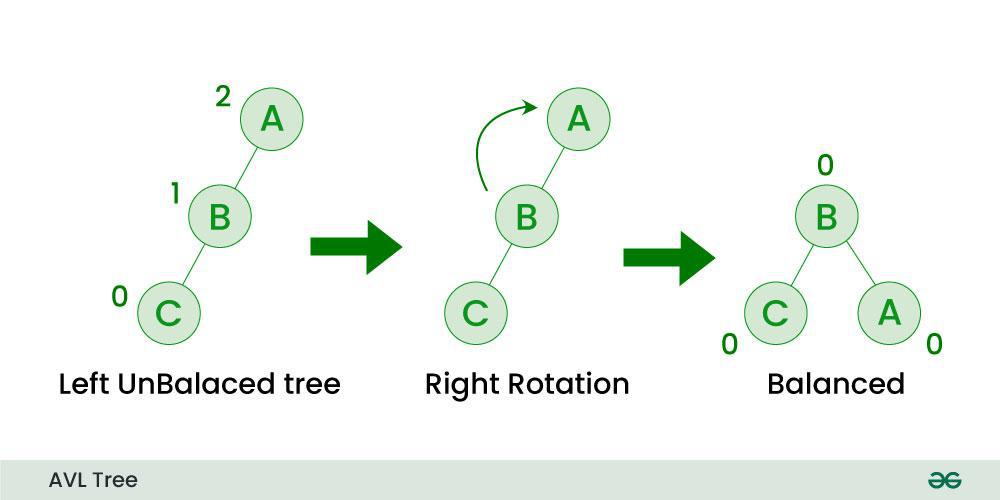
### Rotations in AVL Trees:

*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 (R Rotation)*
2. *Left Rotation (LR)*
3. *Left-Right Rotation (LR)*
4. *Right-Left Rotation (RL)*

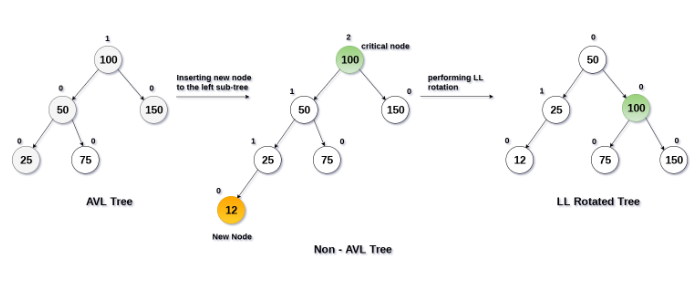
#### Right Rotation (RR)

#### 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.*

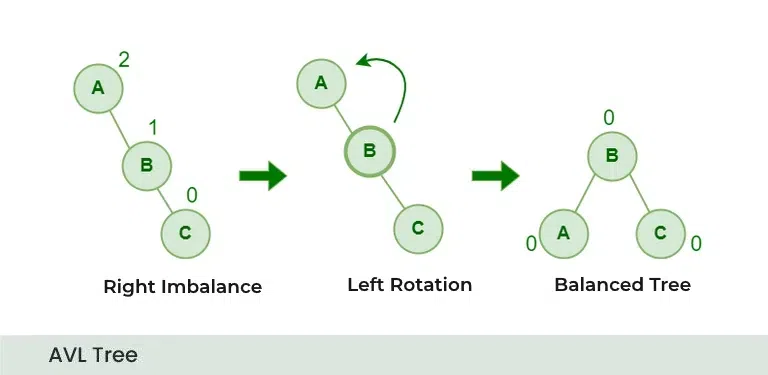
|  |
| --- |
|  |

#### **Example:**



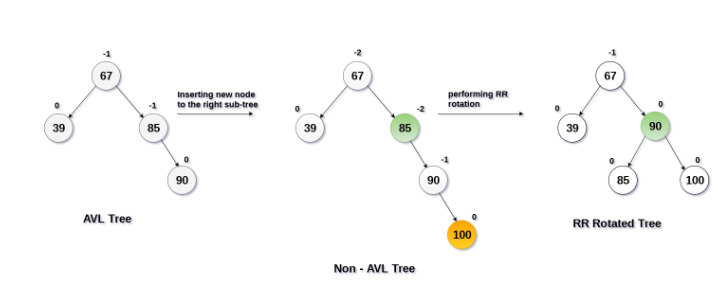
#### Left Rotation (LL)

#### 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.*

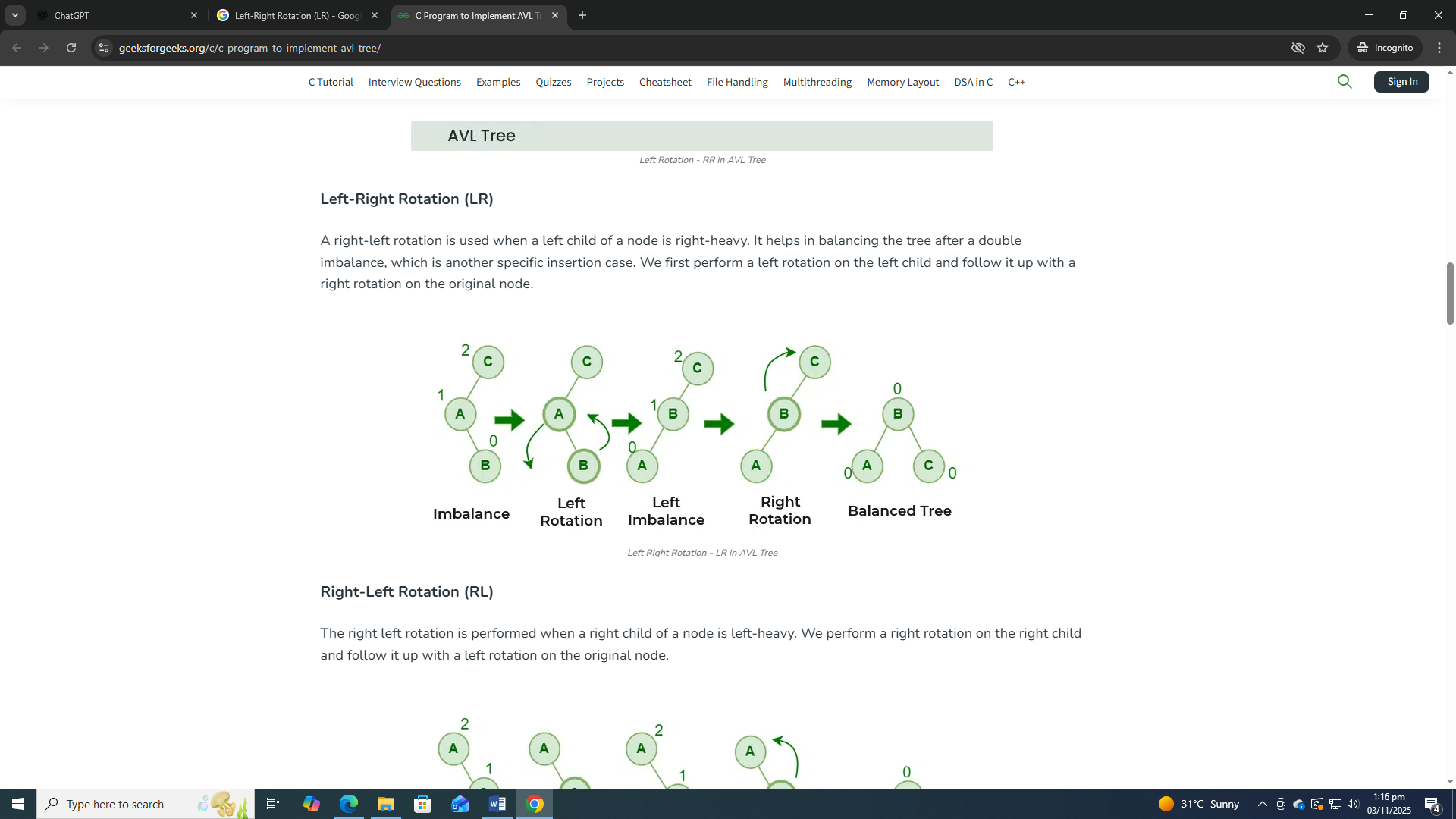
|  |
| --- |
|  |

#### Example:

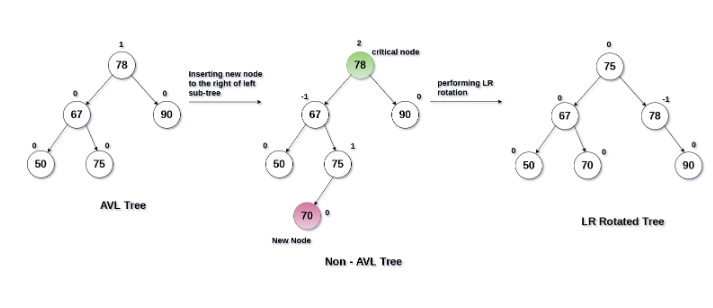


#### Left-Right Rotation (LR)

*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.*

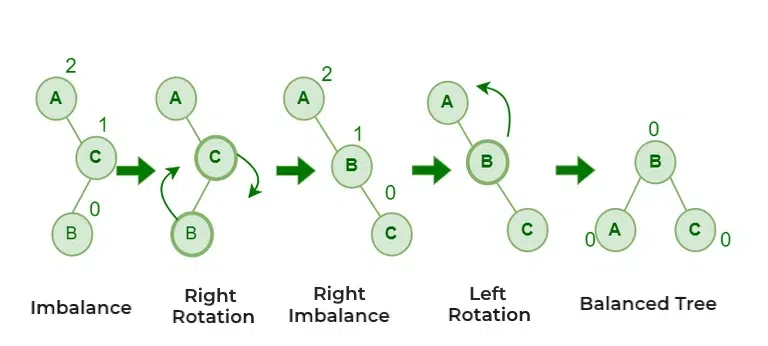


#### **Example:**



#### Right-Left Rotation (RL)

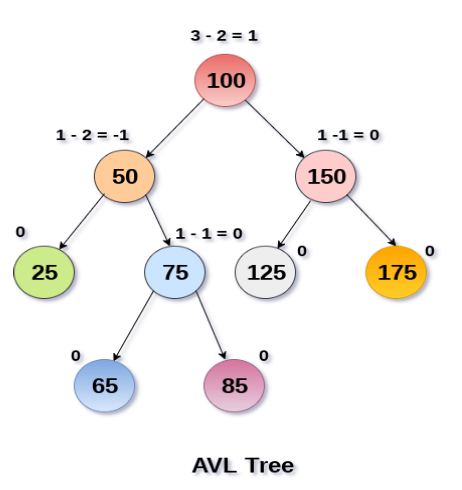
*The Right-Left Rotation (RL) is used when the right child of a node is left-heavy. This imbalance is corrected by performing a right rotation on the right child, followed by a left rotation on the original node.*



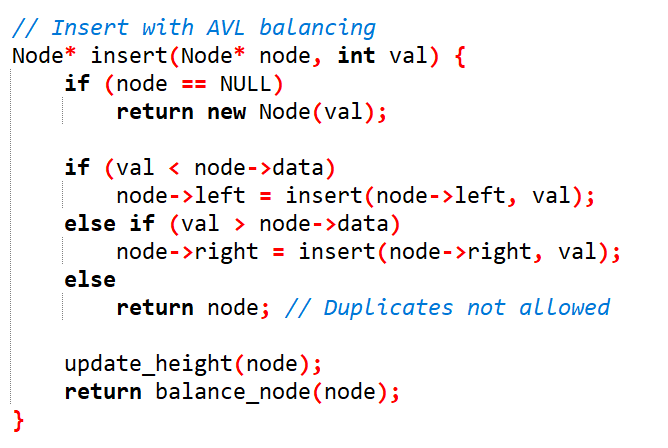
## **Example:**



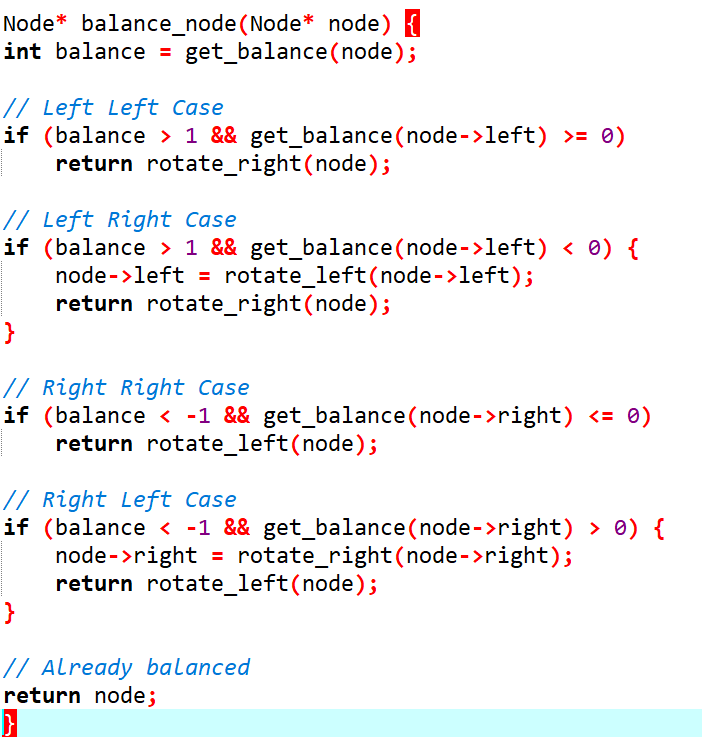
**Example of AVL:**



## Insertion in AVL Tree:



## Balance Node:



## 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.

A screenshot of a computer code

AI-generated content may be incorrect.

## Searching in AVL Tree:

*Same like BST*

**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.

# **LAB TASKS**