**Lab Task 12**



**Superior University Gold Campus**

|  |  |
| --- | --- |
| **Submitted to** | ****Mr. Rasikh Ali**** |
| **Submitted by** | **Javaid Ali** |
| **Roll No** | **SU92-BSSEM-S24-029 (Section – 3A)** |
| **Subject** | **Data Structures and Algorithms (Lab)** |
| **Class** | **BS – Software Engineering** |

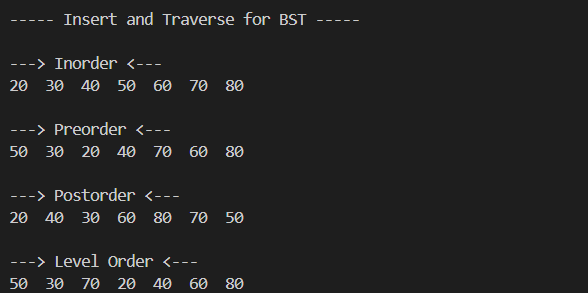
# **Lab 12: BST and AVL**

**1- Insert and Traverse for BST**

This code implements a Binary Search Tree (BST) where each `**TreeNode**` stores an integer value with left/right child pointers, and the `**BST**` class uses recursive helpers to maintain the BST property (left < parent < right) during insertion while offering four traversal methods: **inorder** (left-root-right, producing sorted output), **preorder** (root-left-right, useful for tree copying), **postorder** (left-right-root, helpful for deletion), and **level-order** (breadth-first using a queue, showing nodes by depth).

The `**main**` function demonstrates these operations by inserting values (50, 30, 70, etc.) into the tree and printing each traversal, with **inorder** displaying nodes in ascending order (20, 30, 40...), **preorder** showing the root-first structure (50, 30, 20...), **postorder** revealing leaf nodes early (20, 40, 30...), and **level-order** processing nodes row-by-row (50, 30, 70...), collectively illustrating fundamental BST behaviors for efficient searching and hierarchical data management.

**Outputs:**



**2- Insert and Traverse for AVL**

This code implements a self-balancing **AVL Tree**, an extension of the BST that maintains **O(log n)** search time by automatically balancing itself during insertions. Each `**TreeNode**` now includes a `**height**` attribute to track subtree heights, enabling balance checks. The `**AVL**` class features key balancing operations: `**getHeight()**` and `**getBalance()**` calculate node balance factors, while `**rightRotate()**` and `**leftRotate()**` restructure the tree when imbalances exceed ±1. The `**insertHelper()**` method recursively inserts nodes like a BST but adds balancing logic for four cases—**Left-Left**, **Right-Right**, **Left-Right**, and **Right-Left**—triggering rotations to restore balance. Traversal methods (`**inorder**`, `**preorder**`, `**postorder**`, `**levelOrder**`) remain unchanged from BST but operate on a balanced structure.

In `**main()**`, inserting 10, 20, 30, 40, 50, and 25 demonstrates automatic balancing: the tree reorganizes (e.g., rotating nodes like 20 and 30) to maintain optimal height, ensuring efficient operations. The traversals confirm the tree’s sorted order and structure, with level-order revealing the balanced hierarchy.

**Outputs:**

