**Lab Task 13**



**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** |

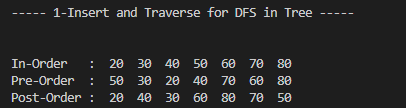
# **Lab 13: DFS and BFS**

**1- Insert and Traverse for DFS in tree**

This C++ program defines a basic **Binary Search Tree (BST)** and demonstrates **Depth-First Search (DFS)** traversals: **in-order**, **pre-order**, and **post-order**. The Node structure represents each tree node with integer data and pointers to left and right children. The **insertTree** function inserts a new node into the BST by recursively placing smaller values to the left and larger values to the right, ensuring sorted tree structure. The three traversal functions—**inorder**, **preorder**, and **postorder**—are recursive implementations of DFS patterns: in-order visits left, root, then right; pre-order visits root first; post-order visits root last.

The **main** function constructs the tree by inserting specific values, then prints the nodes in all three traversal orders. This is done to demonstrate how DFS methods explore tree structures differently.

**Outputs:**

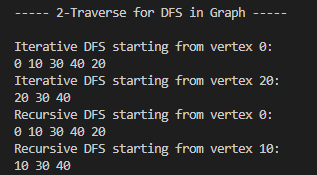


**2- Traverse for DFS in graph**

This C++ program implements **Depth-First Search (DFS)** for a **directed graph** using both **iterative** and **recursive** approaches. The Graph class represents the graph using an adjacency list (list<int>\* adj) to store connections for each vertex. The constructor initializes the number of vertices, and **addEdge** adds directed edges between vertices. The recursive method uses a helper function **DFSUtil** to explore nodes by marking them visited and recursively traversing unvisited adjacent nodes. The iterative method simulates recursion using a **stack**: it pushes the start vertex and explores connected nodes in reverse order (to maintain correct visit sequence), marking each node once visited.

In **main**, a sample graph is created with vertices and edges, and both DFS methods are demonstrated from different starting points to show how DFS explores as deep as possible before backtracking. This highlights the flexibility and depth-first nature of traversal.

**Outputs:**



**3- Insert and Traverse for BFS in tree**

This C++ program demonstrates **Breadth-First Search (BFS)** traversal in a **binary tree**, along with level-order insertion. The Node structure defines each tree node with integer data and pointers to left and right children. The **insertTree** function inserts nodes level-by-level using a queue: it finds the first available spot (left or right) in the tree to maintain a complete binary tree structure. This ensures nodes are added in breadth-first order. The traverse function performs BFS by using a queue to visit nodes level-by-level—starting from the root, it prints each node's data and **enqueues** its children for later processing.

In **main**, several nodes are inserted, and BFS traversal is **displayed**, illustrating how the algorithm explores all nodes at the current depth before moving to the next.

**Outputs:**



**4- Traverse for BFS in graph**

This C++ program performs **Breadth-First Search (BFS)** traversal on a **directed graph** using an adjacency list representation. The Graph class dynamically allocates an array of lists (adj) to store neighbors for each vertex. The **addEdge** method adds a directed edge from vertex v to vertex w, and a commented line indicates how it can be adapted for undirected graphs. The BFS function uses a **queue** to explore vertices level-by-level, beginning at a specified starting vertex. It marks each visited node to avoid repetitions and enqueues unvisited adjacent nodes for future exploration.

In **main**, a graph is built with several directed edges, and BFS is executed from different starting vertices to show how it visits nodes breadth-wise—processing all neighbors of a node before moving deeper.

**Outputs:**

