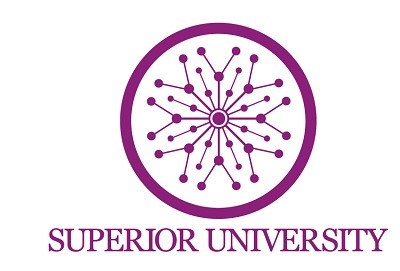
**Lab 13**



Submitted to

:

Sir Rasikh Ali

rR

Submitted by

:

Arooj Fatima

Roll No

:

SU92-BSSEM-S24-091

Subject:

DSA (Lab)

Class:

BS – Software Engineering

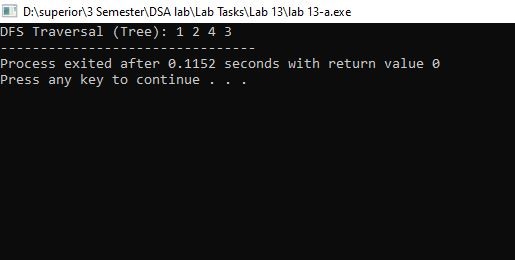
# **Lab # 13**

**DFS and BFS:**

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

This Code demonstrates Depth-First Search (DFS) traversal on a tree. It relies on fixed-size arrays to store child nodes. The **TreeNode** structure contains an integer value (val), a fixed-size array of pointers (children) to hold child nodes, and an integer (childCount) to track how many children are currently attached. The **insertChild** function adds a new child to a node, ensuring the number of children doesn't exceed the defined limit. The **dfsTree** function implements the DFS traversal recursively: it first prints the current node's value, then calls itself on each child node. In the main () function, the tree is constructed with a root node and a few child nodes, and the relationships are established using insertChild. The traversal begins from the root, and the node values are printed in depth-first order. This program avoids dynamic memory containers like vector, making it suitable for educational purposes or environments where only static memory structures are allowed.

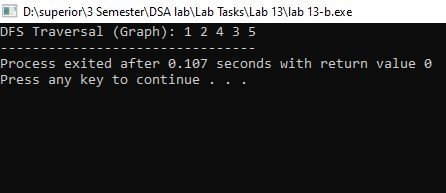
**Output:**



**B-** **Insert and Traverse for DFS in graph**

This Code performs a Depth-First Search (DFS) traversal on an undirected graph. It uses a 2D Boolean array **graph[MAX][MAX]** to represent the graph as an adjacency matrix, where each cell indicates whether there is a connection between two nodes. A visited array is used to keep track of which nodes have already been explored. The **addEdge** function adds an edge between two nodes by setting the corresponding entries in the matrix to true. The **dfsGraph** function implements DFS recursively: it marks the current node as visited, prints it, and then recursively visits all its unvisited neighbors. In the **main** () function, five nodes are connected with edges, and the DFS traversal starts from node 1. The program outputs the nodes in the order they are visited during the DFS. This implementation is simple and effective for small graphs where a static matrix can represent all connections.

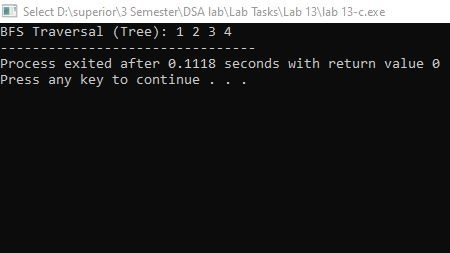
**Output:**



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

This Code implements a simple general tree structure where each node can have multiple children, up to a defined maximum. The **TreeNode** structure stores an integer value, an array of pointers to its children, and a count of how many children it has. A constructor initializes each node’s value and sets all child pointers to nullptr. The **insertChild** function adds a new child to a parent node if the maximum limit hasn't been reached. To traverse the tree, the program uses a Breadth-First Search (BFS) algorithm implemented in the **bfsTree** function. This function uses a fixed-size queue to visit each node level by level, printing their values. In the main function, a small tree is constructed with one root node and three additional child nodes. The tree is then traversed using BFS, and the values are printed in order. This program demonstrates basic concepts of tree creation, insertion, and BFS traversal using arrays instead of dynamic containers like vectors or queues from the STL. It provides a clear illustration of handling tree structures manually in C++.

**Output:**



**D-** **Insert and Traverse for BFS in graph**

This code demonstrates the Breadth-First Search (BFS) algorithm on an undirected graph using an adjacency matrix. The graph is represented by a 2D array where graph[i][j] = 1 indicates a connection between nodes i and j. An auxiliary array visited[] keeps track of visited nodes to avoid revisiting them. The program also manually implements a queue using an array to manage the order of traversal. The **bfsGraph** function starts BFS from a given node, marking it as visited and adding it to the queue. It then repeatedly removes a node from the front of the queue, prints it, and adds all its unvisited adjacent nodes to the rear. In the main function, a graph with 5 nodes and specific connections is defined. The BFS traversal starts from node 1 and outputs the nodes in the order they are visited. This program is a straightforward implementation of BFS without using any STL containers, providing a clear understanding of the traversal mechanism in a graph structure.

**Output:**

