# IT DS 201 LAB

SUBMITTED BY ADITYA SINGH 2K19/EP/005

## Program 20 : Write a program to implement Breadth First Search.

**CODE**

```
void addEdge(int src, int dest) {
    adjLists[src].push_back(dest);
    adjLists[dest].push_back(src);
}

void BFS(int startVertex) {
    visited = new bool[numVertices];
    for (int i = 0; i < numVertices; i++)
        visited[i] = false;

    list<int> queue;

    visited[startVertex] = true;
    queue.push_back(startVertex);

    list<int>::iterator i;

    while (!queue.empty()) {
        int curr = queue.front();
        cout << curr << " ";
        queue.pop_front();

        for (i = adjLists[curr].begin(); i != adjLists[curr].end(); ++i) {
            int adjVertex = *i;
            if (!visited[adjVertex]) {
                visited[adjVertex] = true;
                queue.push_back(adjVertex);
            }
        }
    }
}
```

```
#include <iostream>
#include <list>

using namespace std;

class Graph {
    int numVertices;
    list<int> *adjLists;
    bool* visited;

public:
    Graph(int vertices) {
        numVertices = vertices;
        adjLists = new list<int>[vertices];
    }
```

## ALGORITHM

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing the same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

**INPUT/OUTPUT**

```
int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    g.BFS(0);

    return 0;
}
```

```
0 1 2 3 [Finished in 709ms]
```

# Program 21 : Write a program to implement Depth First Search.

**CODE**

```
#include <bits/stdc++.h>
using namespace std;

class Graph {

public:
    map<int, bool> visited;
    map<int, list<int>> adj;

    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    void DFSUtil(int v) {
        visited[v] = true;
        cout << v << " ";

        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i) {
            if (!visited[*i])
                DFSUtil(*i);
        }
    }

    void DFS() {
        for (auto i : adj) {
            if (visited[i.first] == false)
                DFSUtil(i.first);
        }
    }
};
```

## ALGORITHM

1. Create a recursive function that takes the index of the node and a visited array.
2. Mark the current node as visited and print the node.
3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
4. Run a loop from 0 to the number of vertices and check if the node is unvisited in the previous DFS, call the recursive function with the current node.

## INPUT/OUTPUT

```
int main() {

    Graph g;
    g.addEdge(0, 1);
    g.addEdge(0, 9);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(9, 3);

    cout << "Depth First Traversal \n";
    g.DFS();

    return 0;
}
```

```
Depth First Traversal
0 1 2 3 9 [Finished in 3.7s]
```

**END**