**Part-1: Breadth First Search (BFS) of an Undirected Graph**

**Pseudocode:**

This pseudocode assumes that the graph has a "node" class with a "value" attribute and a list of "neighbors", and a "visited" boolean flag. It also assumes that the graph has an adjacency list representation, and that there is a queue data structure available.

The BFS algorithm starts at the given start node, marks it as visited, and adds it to the queue. It then dequeues the first node from the queue and visits its neighbors. If a neighbor has not been visited, it marks it as visited and adds it to the queue. This process continues until the queue is empty, at which point all reachable nodes in the graph have been visited.

The nodes are visited in a breadth-first order, which means that all the neighbors of a node are visited before any of their neighbors. This can be useful for finding the shortest path between two nodes, or for discovering the connected components of an undirected graph.

PROCEDURE BFS(graph: Graph, start\_node: Node)

queue = new Queue()

start\_node.visited = True

queue.enqueue(start\_node)

WHILE queue is not empty

current\_node = queue.dequeue()

print current\_node.value

FOR each neighbor in current\_node.neighbors

IF neighbor.visited is False

neighbor.visited = True

queue.enqueue(neighbor)

END WHILE

**Code:**

// Program to print BFS traversal from a given source vertex.

#include <bits/stdc++.h>

using namespace std;

class Graph {

int V;

vector<list<int> > adj;

public:

Graph(int V);

void addEdge(int v, int w);

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj.resize(V);

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

}

void Graph::BFS(int s)

{

vector<bool> visited;

visited.resize(V, false);

list<int> queue;

visited[s] = true;

queue.push\_back(s);

while (!queue.empty()) {

s = queue.front();

cout << s << " ";

queue.pop\_front();

for (auto adjecent : adj[s]) {

if (!visited[adjecent]) {

visited[adjecent] = true;

queue.push\_back(adjecent);

}

}

}

}

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);

cout << "Breadth First Traversal "

<< "(starting from vertex 1) \n";

g.BFS(1);

return 0;

}

**Output Screenshot:**



**Part-2: Depth First Search (DFS) of an Undirected Graph Pseudocode:**

This pseudocode assumes that the graph has a "node" class with a "value" attribute and a list of "neighbors", and a "visited" boolean flag. It also assumes that the graph has an adjacency list representation.

The DFS algorithm starts at the given node, marks it as visited, and prints its value. It then visits each of the unvisited neighbors of the node by calling the DFS function recursively. This process continues until all reachable nodes in the graph have been visited.

The nodes are visited in a depth-first order, which means that a node is completely explored before any of its neighbors. This can be useful for discovering cycles in a graph, or for topological sorting.

PROCEDURE DFS(graph: Graph, node: Node)

node.visited = True

print node.value

FOR each neighbor in node.neighbors

IF neighbor.visited is False

DFS(graph, neighbor)

**Code:**

// DFS traversal from a given vertex in a given graph

#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);

void DFS(int v);

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

}

void Graph::DFS(int v)

{

visited[v] = true;

cout << v << " ";

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

int main()

{

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal"

" (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

**Output Screenshot:**

****