**AIM:**

To create a graph and display the ordering of vertices.

**TOPOLOGICAL SORTING**

**ALGORITHM:**

**CREATE NODE:-**

1. Allocate memory for a new node.
2. Set the vertex of the node to the given vertex v.
3. Initialize the next pointer of the node to NULL.
4. Return the created node.

**CREATE GRAPH:-**

1. Allocate memory for a new graph.
2. Set the numVertices of the graph to the given number of vertices.
3. Allocate memory for an array of adjacency lists (adjLists).
4. Allocate memory for an array to track visited vertices (visited).
5. Initialize each adjacency list to NULL.
6. Initialize each element of the visited array to 0 (not visited).
7. Return the created graph.

**ADDEDGE:-**

1. Create a new node for the destination vertex dest.
2. Add the node to the adjacency list of the source vertex src by setting the new node's next pointer to the current head of the list and then updating the head to the new node.
3. Create a new node for the source vertex src.
4. Add the node to the adjacency list of the destination vertex dest in the same manner as above (since the graph is undirected).

**PRINTGRAPH:-**

1. For each vertex in the graph:
2. Print the vertex number.
3. Traverse the adjacency list of the vertex and print each connected vertex followed by " -> ".
4. Print "NULL" at the end of each adjacency list.

**DFS:-**

1. Mark the current vertex as visited by setting the corresponding element in the visited array to 1.
2. Record the current vertex in the order array and increment the index.
3. For each adjacent vertex of the current vertex:
4. If the adjacent vertex is not visited, recursively perform DFS on the adjacent vertex.

**BFS:-**

1. Initialize an empty queue.
2. Mark the start vertex as visited and enqueue it.
3. While the queue is not empty:
4. Dequeue a vertex from the queue.
5. Record the vertex in the order array.
6. For each adjacent vertex of the dequeued vertex:
7. If the adjacent vertex is not visited, mark it as visited and enqueue it.

**MAIN PROGRAM:-**

1. Define the number of vertices.
2. Create the graph by calling createGraph with the number of vertices.
3. Add edges to the graph by calling addEdge with appropriate source and destination vertices.
4. Print the graph representation by calling printGraph.
5. Initialize an order array and an index for DFS.
6. Perform DFS starting from vertex 0 by calling DFS and passing the graph, starting vertex, order array, and index.
7. Print the order of vertices visited by DFS.
8. Reset the visited array to 0 for BFS.
9. Initialize an order array for BFS.
10. Perform BFS starting from vertex 0 by calling BFS and passing the graph, starting vertex, and order array.
11. Print the order of vertices visited by BFS.

**PROGRAM**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

typedef struct Graph {

int numVertices;

Node\*\* adjLists;

int\* visited;

} Graph;

// Function to create a node

Node\* createNode(int v) {

Node\* newNode = malloc(sizeof(Node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Function to create a graph

Graph\* createGraph(int vertices) {

Graph\* graph = malloc(sizeof(Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(Node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Function to add an edge to the graph

void addEdge(Graph\* graph, int src, int dest) {

Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Function to print the graph

void printGraph(Graph\* graph) {

for (int v = 0; v < graph->numVertices; v++) {

Node\* temp = graph->adjLists[v];

printf("\n Vertex %d\n: ", v);

while (temp) {

printf("%d -> ", temp->vertex);

temp = temp->next;

}

printf("NULL\n");

}

}

// Function to perform DFS

void DFS(Graph\* graph, int vertex, int\* order, int\* index) {

Node\* adjList = graph->adjLists[vertex];

Node\* temp = adjList;

graph->visited[vertex] = 1;

order[(\*index)++] = vertex;

while (temp != NULL) {

int connectedVertex = temp->vertex;

if (graph->visited[connectedVertex] == 0) {

DFS(graph, connectedVertex, order, index);

}

temp = temp->next;

}

}

// Function to perform BFS

void BFS(Graph\* graph, int startVertex, int\* order) {

int queue[graph->numVertices];

int front = 0, rear = 0;

int index = 0;

graph->visited[startVertex] = 1;

queue[rear++] = startVertex;

while (front != rear) {

int currentVertex = queue[front++];

order[index++] = currentVertex;

Node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

queue[rear++] = adjVertex;

graph->visited[adjVertex] = 1;

}

temp = temp->next;

}

}

}

// Main function

int main() {

int vertices = 6;

Graph\* graph = createGraph(vertices);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

addEdge(graph, 3, 5);

printf("Graph representation:\n");

printGraph(graph);

int order[vertices];

int index = 0;

printf("\nDepth First Search (starting from vertex 0):\n");

DFS(graph, 0, order, &index);

for (int i = 0; i < index; i++) {

printf("%d ", order[i]);

}

printf("\n");

// Reset visited array for BFS

for (int i = 0; i < vertices; i++) {

graph->visited[i] = 0;

}

printf("\nBreadth First Search (starting from vertex 0):\n");

BFS(graph, 0, order);

for (int i = 0; i < vertices; i++) {

printf("%d ", order[i]);

}

printf("\n");

return 0;

}

**OUTPUT:**

Graph representation:

Vertex 0

: 2 -> 1 -> NULL

Vertex 1

: 3 -> 2 -> 0 -> NULL

Vertex 2

: 4 -> 1 -> 0 -> NULL

Vertex 3

: 5 -> 4 -> 1 -> NULL

Vertex 4

: 3 -> 2 -> NULL

Vertex 5

: 3 -> NULL

Depth First Search (starting from vertex 0):

0 1 3 5 4 2

Breadth First Search (starting from vertex 0):

0 1 2 3 4 5