**GRAPHS**

**EXPT NO: 8**  **DATE: 22/12/21**

**AIM**

**1)** Write a program to implement graph using adjacent matrix

1. Insert a vertex
2. Insert an edge
3. Delete a vertex
4. Delete an edge
5. Display Graph

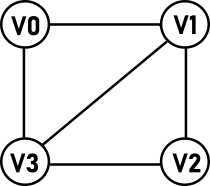
**2)** Write a program to implement Breadth First Search and Depth First search in a directed graph.

**THEORY**

A graph G = (V,E) is a collection of sets V and E where V is the collection of vertices and E is the collection of edges. An edge is a line or arc connecting two vertices and it is denoted by a pair (i,j) where I, j belong to the set of vertices V. A graph can be of two types – Undirected graph or Directed Graph.

**Undirected Graph**

A graph which has unordered pair of vertices is called undirected graph. If there is an edge between vertices u and v then it can be represented as either (u,v) or (v,u).

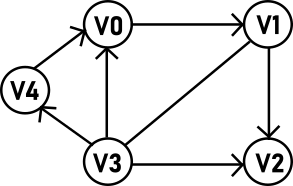


**V(G)** = {V0, V1, V2, V3}

**E(G)** = {(V0,V1), (V0,V3), (V1,V2), (V1,V3), (V2,V3) }

**Directed Graph**

A directed graph or digraph which has ordered pair of vertices (u,v) where u is the tail and v is the head of the edge.in this type of graph, a direction is associated with edge i.e. (u,v) and (v,u) represent different edges.



**V(G)** = {V0, V1, V2, V3, V4}

**E(G)** = { (V0,V1), (V1,V2), (V3,V2), (V3,V0), (V3,V4), (V4,V0) }

|  |  |
| --- | --- |
| **TERMINOLOGY** | **EXPLANATION** |
| Path | A path from vertex u1 to vertex un is a sequence of vertices u1, u2, u3,..un-1,un such that u2 is adjacent to u1, u3 is adjacent to u2,………..un is adjacent to un-1. |
| Simple path | Simple path is a path in which all the vertices are distinct. |
| Closed Path | A path will be called as closed path if the initial node is same as terminal node. A path is called as closed if V0 = Vn |
| Length of a path | The length of a path is the total number of edges included in the path. |
| Weighted graph | A graph is weighted if its edges have been assigned some non-negative value as weight. A weighted graph is also known as network. |
| Degree | In an undirected graph, the degree of a vertex is the number of edges incident on it. |
| Adjacency | Adjacency is a relation between two vertices of a graph. A vertex is adjacent to another vertex u if there is an edge from vertex u to vertex v i.e. edge (u,v)€E |
| Cycle | In a digraph, a path u1, u2, u3….., un-1,un is called a cycle if it has atleast two vertices and the first and last vertices are same i.e. u1=un |
| Complete graphs | A graph is complete if any vertex in the graph is adjacent to all the vertices of the graph or we can say that there is an edge between any pair of vertices in the graph. |
| Connected Graph | A connected graph is the one in which every node is connected with all other nodes. A complete graph contains n(n-1)/2 edges where n is the number of nodes in the graph. |
| Multiple edges | If there is more than one edges between a pair of vertices then the edges are known as multiple edges or parallel edges |
| Subgraph | A graph H is said to be a subgraph of another graph G, iff the vertex set of H is subset of vertex set of G and edge set of H is subset of edge set of G. |
| Isolated Vertex | If the degree of a vertex is 0, then it is called an isolated vertex. |
| Forest | A forest is a disjoint union of trees. In a forest there is at most one path between any two vertices, this means that there is ither no path or a single path between anu two vertices. |
| Spanning tree | A spanning tree of s graph G is a subgraph that includes all vertices of G and some (or all) edges of G, such that all the vertices the vertices are connected and there are no cycles. Spanning tree of a graph is not unique. |
| Spanning forest | A spanning forest is a subgraph that consists of a spanning tree for each connected component of a graph. |

**Representation of graphs**

There are two ways of representing a graph, one is the sequential representation (adjacency matric) and the other is the linked representation (Adjacency List)

**Adjacency Matrix**

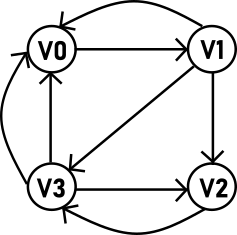
Adjacency matric is a matrix that maintains the information of adjacent vertices. In other words, we can say that this matrix tells us whether a vertex is adjacent to any other vertex or not.

The entries of the adjacency matrix are filled using this definition.

**1**  If there is an edge from vertex i to vertex j

A(i,j) =

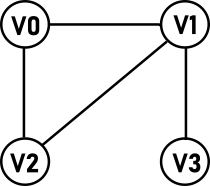
**0** If there is no edge from vertex i to vertex j



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **V0** | **V1** | **V2** | **V3** |
| **V0** | 0 | 1 | 0 | 1 |
| **V1** | 1 | 0 | 1 | 1 |
| **V2** | 0 | 0 | 0 | 1 |
| **V3** | 1 | 0 | 1 | 0 |

Directed Graph

Adjacency Matric



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **V0** | **V1** | **V2** | **V3** |
| **V0** | 0 | 1 | 0 | 1 |
| **V1** | 1 | 0 | 1 | 1 |
| **V2** | 0 | 0 | 0 | 1 |
| **V3** | 1 | 0 | 1 | 0 |

Adjacency Matric

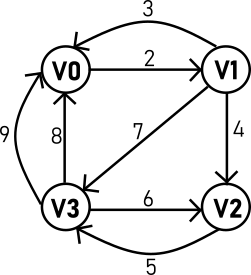
Undirected Graph

If a graph has some weights on its edges, then the elements of adjacency matric can be defined as

**Weight on edge**  If there is an edge from vertex i to vertex j

A(i,j) =

**0** If there is no edge from vertex i to vertex j



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **V0** | **V1** | **V2** | **V3** |
| **V0** | 0 | 2 | 0 | 8 |
| **V1** | 3 | 0 | 4 | 7 |
| **V2** | 0 | 0 | 0 | 5 |
| **V3** | 9 | 0 | 6 | 0 |

Weighted Adjacency Matrix

Weighted Undirected Graph

**Adjacency List**

If the graph is not dense i.e., the number of edges is less, then it is efficient to represent the graph through adjacency list.

In adjacency list representation if graph, we maintain two linked lists. The first linked list is the vertex list that keeps track of all the vertices in the graph and second linked list is the edge that maintains a list of adjacent vertices for each vertex. Suppose there are n vertices then we will create one list which will keep information of all n vertices in the graph and after that we will create n lists, where each list will keep information of all adjacent vertices of that particular vertex.

The structure of the nodes of these two lists would be

struct vertex

{

int info;

struct vertex \*nextVertex;

struct Edge \*firstEdge;

}\*start=NULL;

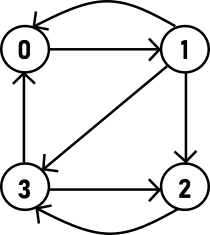
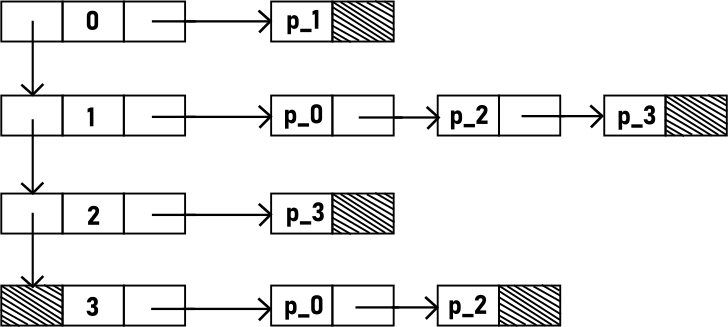
struct Edge

{

struct vertex \*destVertex;

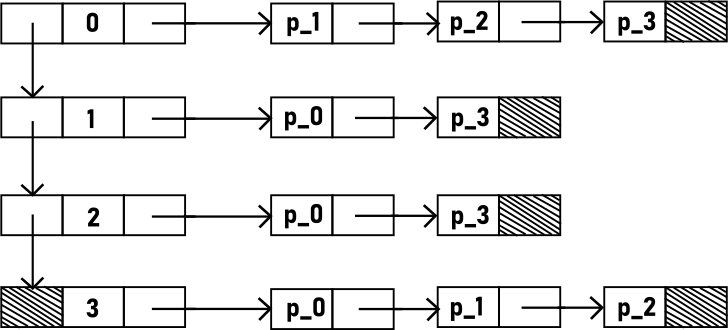
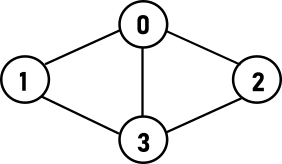
struct Edge \*nextEdge;

};



Adjacency List

Directed Graph



Adjacency List

Undirected Graph

**Traversal**

**Breadth First Search**

In this Technique, first we visit the starting vertex and then visit all the vertices adjacent to the starting vertex. After this we pick these adjacent vertices one by one and visit their adjacent vertices and this process goes on.

Note that these traversals are not unique, there can be different traversals depending on the order in which we visit the successors. BFS is implemented using a queue data structure.

During the algorithm, any vertex will be in one of the three states – initial, waiting, visited. At the start of the algorithm all vertices will be initial state, when a vertex is inserted in the queue its state will change from initial to waiting. When a vertex will be deleted from the queue and visited, its state will change from waiting to visited.

Procedure: -

Initially queue is empty and all vertices are in initial state.

1. Insert the starting vertex, and all vertices are in initial state.
2. Delete front element from the queue and visit it, change its state to waiting.
3. Look for the adjacent vertices of the deleted element, and from these, insert only those vertices into the queue which are in the initial state. Change the state of all these inserted vertices from initial to waiting.
4. Repeat steps 2, 3 until the queue is empty.

**Depth First Search**

Traversing using DFS is like traversing a maze. We travel along a path in the graph and when a dead end comes, we backtrack. This technique is named so because search proceeds deeper in the graph i.e., we traverse along a path as deep as we can. The dead end means that we reach a vertex which do not have adjacent vertex or all it adjacent vertices are visited. Depth first search is implemented using a stack or recursively.

During the algorithm any vertex will be in one of these two states – initial or visited. At the start of the adjacent all vertices will be initial state, and when a vertex will be popped from the stack and its state will change to visited.

Procedure: -

1. Push starting vertex on the stack
2. Pop vertex on the stack
3. If popped vertex is in initial state, visit it and change its state to visited. Push all unvisited vertices adjacent to the popped vertex.
4. Repeat steps 2and 3 until the stack is empty

**ALGORITHM**

**1)**

**Void insertEdge(int origin, int destin)**

1. if GCHECK(origin)

1. Origin vertex does not exists

2. return

2. if GCHECK(destin)

1. Destin vertex does not exists

2. return

3. adj[origin][destin]=1

**#define GCHECK(x)**

(x<0||x>=n)

Returns 1 if true else returns 0

**Void createGraph()**

1. Declare I, maxEdge,origin,destin

2. Input Number of Vertices

3.maxEdge=n\*(n-1)

3. for i=0; i<maxEdge; i++

1. input origin and destin

2. if origin ==-1 and destin == -1

1. break

4. if GCHECK(origin)||GCHECK(destin)

1. Invalid Edge

2. i=i-1

5. else

1. adj[origin][destin]=1

**Void delVertex(int vert)**

1. Declare int i

2. if n==0

1. Graph empty

2. return

3. if GCHECK(vert)

1. Invalid Vertex

2. return

4. while vet<n

1. for i=0 ; i<n; i++

1. adj[i][vert]=adj[i][vert+1]

2. for i=0; i<n; i++

1.adj[vert]i]=adj[vert+1][i]

3. vert=vert+1

5. n=n-1

**void display()**

1. Declare I,j
2. For i=0; i<n; i++
3. For j=0; j< n j++

1. print adj[i][j]

**Void delEdge(int origin, int destin)**

1. if GCHECK(origin)|| GCHECK(destin)||adj[origin][destin]

1. No Edge exists

2. return

adj[origin][destin]=0

**2)**

**void BFS\_traversal()**

1. Declare int u,i
2. for(i=0;i<n;i++)
3. state [i] = INITIAL
4. Input starting vertex u
5. Call BFS (u)

**void DFS\_traversal()**

1. Declare int u,i
2. For(i=0;i<n;i++)
3. State [i]=INITIAL
4. Input starting vertex u
5. Call DFS(u)

**void BFS(int u)**

1. Declare i
2. Call insert (u)
3. While(!isEmpty())
4. Call del() and initialize the returned value to u
5. Set state [u]=VISITED
6. for(i=0;i<n;i++)
7. if adj [uu] [i]==1 &&state [i] == INITIAL
8. Call insert(i)
9. Set state [i]=waiting

**void DFS (int u)**

1. Declare int i
2. If state [u] == INITIAL
3. Output u
4. State [u] = VISITED
5. for(i=0;i<n;i++)

1. if adj [u] [i] ==1 && state [i] == INITIAL

1. call DFS(i)

**CODES**

**1)**

#include<stdio.h>

#include<string.h>

#define MAX 50

#define LINE printf("~");

#define GCHECK(x)(x<0||x>=n)

int n;

int adj[MAX][MAX];

void DESIGN(int n)

{int i; for(i=0;i<n;i++)LINE;}

void DESIGN2(int n)

{

int i;

printf(" ");

for(i=0;i<5;i++)

printf(" ");

DESIGN(n-5);

}

void createGraph()

{

int i, maxEdges,origin,destin;

printf("ENTER NUMBER OF VERTICES: ");

scanf("%d",&n);

maxEdges=n\*(n-1);

for(i=1;i<maxEdges;i++)

{

printf("ENTER EDGE %d (-1,-1 to exit): ",i);

scanf("%d,%d",&origin,&destin);

if(origin==-1&&destin==-1)

break;

if(GCHECK(origin)||GCHECK(destin))

{

printf("INVALID EDGE\n");

i--;

}

else

adj[origin][destin]=1;

}

}

void insertEdge(int origin,int destin)

{

if(GCHECK(origin))

{printf("ORIGIN VERTEXT DOES NOT EXIST\n");return;}

if(GCHECK(destin))

{printf("DESTINATION VERTEXT DOES NOT EXIST\n");return;}

adj[origin][destin]=1;

}

void delEdge(int origin,int destin)

{

if(GCHECK(origin)||GCHECK(destin)||adj[origin][destin]==0)

{

printf("NO EDGE EXISTS\n");

return;

}

adj[origin][destin]=0;

}

void delVertex(int vert)

{

int i,TEMP=vert;

if(n==0)

{

printf("GRAPH EMPTY\n");

return;

}

if(GCHECK(vert))

{

printf("INVALID VERTEX\n");

return;

}

while(vert<n)

{

for(i=0;i<n;i++)

adj[i][vert]=adj[i][vert+1];

for(i=0;i<n;i++)

adj[vert][i]=adj[vert+1][i];

vert++;

}

printf("THE VERTEX DELETED WAS: %d\n",TEMP);

n--;

}

void display()

{

int i,j;

printf(" ");

for(i=0;i<n;i++)

printf("%6d",i);

printf("\n");

DESIGN2(n\*6);

printf("\n");

for(i=0;i<n;i++)

{

printf(" %2d|",i);

for(j=0;j<n;j++)

printf("%6d",adj[i][j]);

printf("\n");

}

}

int main()

{

int c, origin, destin,d;

createGraph();

while(1)

{

printf("\n\n");

DESIGN(37);printf("\nDIRECTED GRAPH USING ADJACENCY MATRIX\n");DESIGN(37);

printf("\n 1: INSERT A EDGE\n");

printf(" 2: INSERT A NEW VERTEX\n");

printf(" 3: DELETE AN EDGE\n");

printf(" 4: DELETE AN VERTEX\n");

printf(" 5: DISPLAY ADJACENCY MATRIX\n");

printf(" ENTER YOUR CHOICE: ");

scanf("%d",&c);

switch(c)

{

case 1:

printf("ENTER NEW EDGE (Format x,y): ");

scanf("%d,%d",&origin,&destin);

insertEdge(origin,destin);

break;

case 2:

n++;

printf("NEW VERTEX ADDED\n");

break;

case 3:

printf("ENTER EDGE TO BE DELETED (Format x,y): ");

scanf("%d,%d",&origin,&destin);

delEdge(origin,destin);

break;

case 4:

printf("ENTER THE VERTEX TO BE DELETED: ");

scanf("%d",&d);

delVertex(d);

break;

case 5:

display();

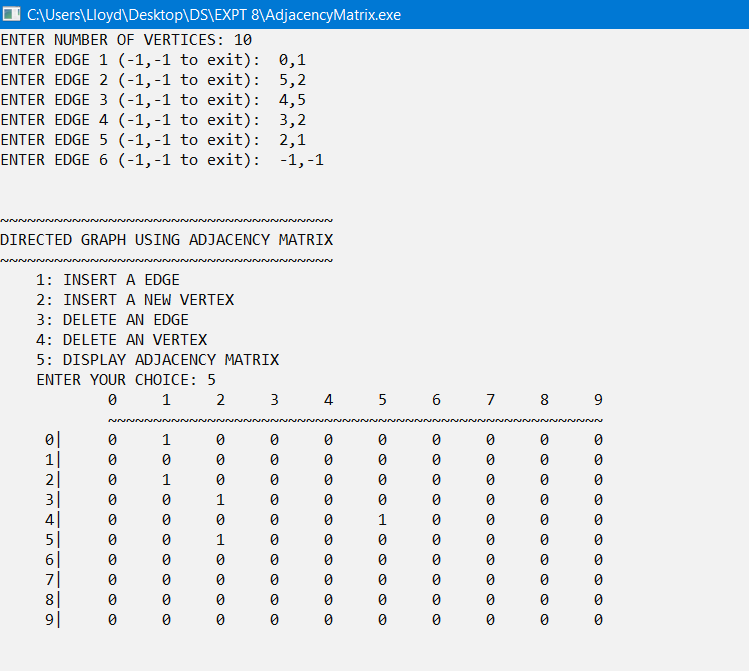
break;

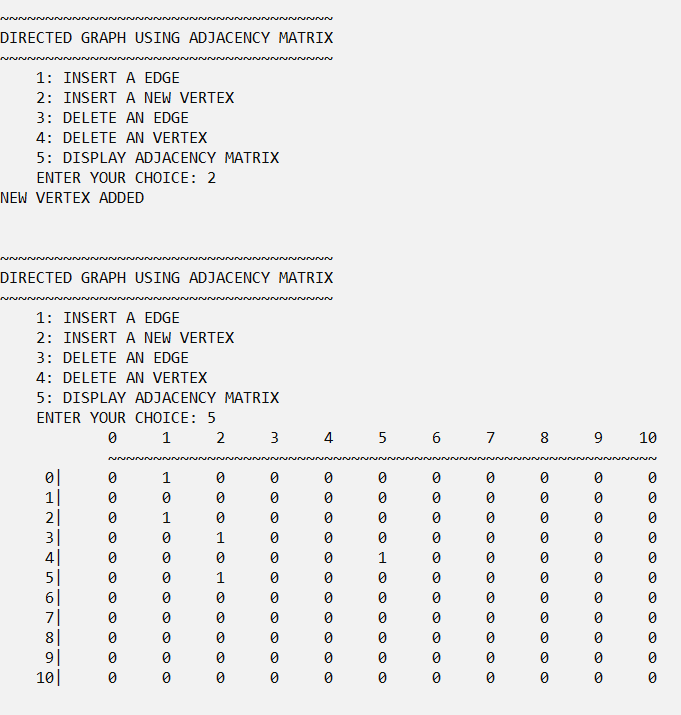
}

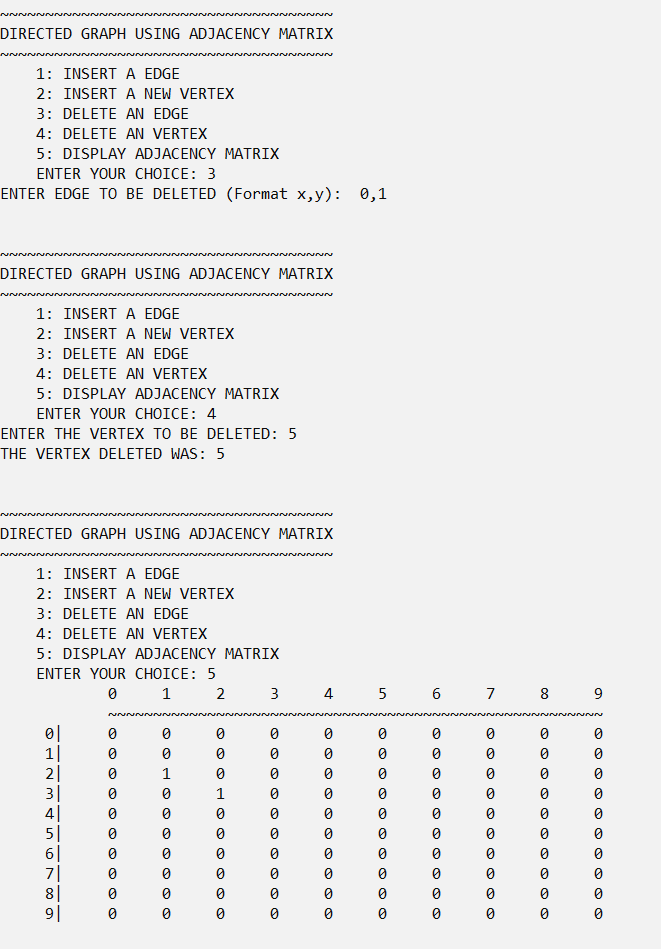
}

}

**OUTPUT**





****

**2)**

#include<stdio.h>

#include<stdlib.h>

#define LINE printf("~");

#define GCHECK(x)(x<0||x>=n)

#define INITIAL 1

#define WAITING 2

#define VISITED 3

#define MAX 30

int adj[MAX][MAX];

int state[MAX];

int n;

int qarr[MAX];

int front=-1, rear=-1;

void insert(int data);

int del();

int isEmpty();

void BFS\_travseral();

void BFS(int u);

void DFS\_travseral();

void DFS(int u);

void display();

void DESIGN(int n)

{int i; for(i=0;i<n;i++)LINE;}

void DESIGN2(int n)

{

int i;

printf(" ");

for(i=0;i<5;i++)

printf(" ");

DESIGN(n-5);

}

void display()

{

int i,j;

printf(" ");

for(i=0;i<n;i++)

printf("%6d",i);

printf("\n");

DESIGN2(n\*6);

printf("\n");

for(i=0;i<n;i++)

{

printf(" %2d|",i);

for(j=0;j<n;j++)

printf("%6d",adj[i][j]);

printf("\n");

}

}

int main()

{

int u, v,i;

printf("ENTER THE NUMBER OF VERTICES: ");

scanf("%d",&n);

int max\_edges=n\*(n-1);

for(i=1;i<=max\_edges;i++)

{

printf("ENTER THE SOURCE AND DESTINATION((-1,-1) TO EXIT): ");

scanf("%d,%d",&u,&v);

if(u==-1 && v==-1)

break;

if(GCHECK(u)|GCHECK(v))

{

printf("INVALID VERTEX\n\n");

i--;

}

else

adj[u][v]=1;

}

display();

printf("\n\n");

DFS\_travseral();

BFS\_travseral();

return 0;

}

void BFS\_travseral()

{

int u,i;

for(i=0;i<n;i++)

state[i]=INITIAL;

printf("\n");DESIGN(10);printf("BREADTH FIRST SEARCH");DESIGN(10);printf("\n");

printf("ENTER THE STARTING VERTEX: ");

scanf("%d",&u);

printf("\nBFS: ");

BFS(u);

}

void BFS(int u)

{

int i;

insert(u);

while(!isEmpty())

{

u=del();

printf("%d ",u);

state[u]=VISITED;

for(i=0;i<n;i++)

{

if(adj[u][i]==1 && state[i]==INITIAL)

{

insert(i);

state[i]=WAITING;

}

}

}

}

void DFS\_travseral()

{

int u,i;

for(i=0;i<n;i++)

state[i]=INITIAL;

printf("\n");DESIGN(10);printf("DEPTH FIRST SEARCH");DESIGN(10);printf("\n");

printf("ENTER THE STARTING VERTEX: ");

scanf("%d",&u);

printf("\nDFS: ");

DFS(u);

}

void DFS(int u)

{

int i;

if(state[u]==INITIAL)

{

printf("%d ",u);

state[u]=VISITED;

}

for(i=0;i<n;i++)

{

if(adj[u][i]==1 && state[i]==INITIAL)

DFS(i);

}

}

void insert(int data)

{

if(rear==MAX-1)

{

printf("Queue Overflow\n");

return;

}

if(front==-1)

{

qarr[++front]=data;

rear=0;

}

else

qarr[++rear]=data;

}

int del()

{

if(isEmpty())

{

printf("Queue Underflow\n");

exit(0);

}

return qarr[front++];

}

int isEmpty()

{

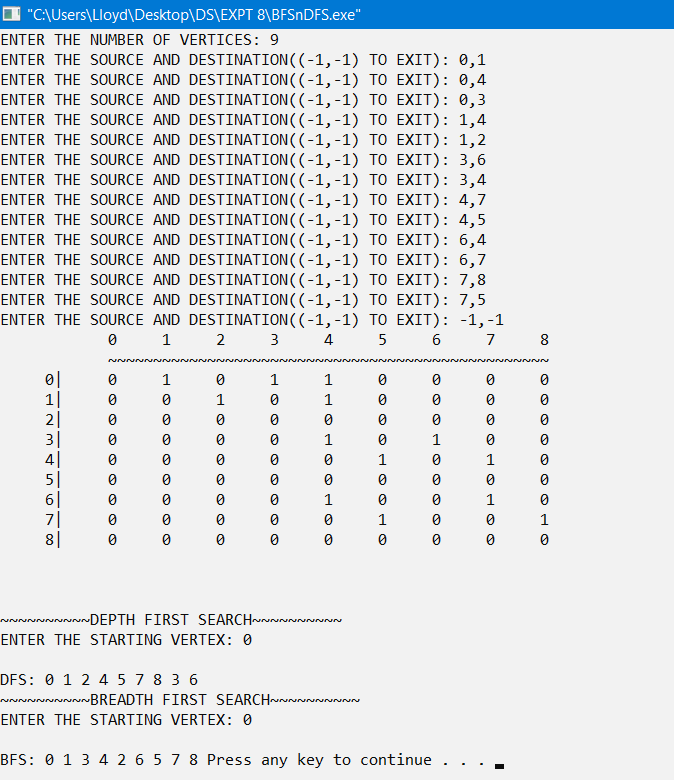
if(front==-1||front>rear)

return 1;

return 0;

}

**OUTPUT**



**CONCLUSION**

The given problem statement was successfully compiled and executed.

**LEARNINGS AND FINDINGS**

This experiment illustrates

1. Concept of Graphs
2. Graph representations
3. Graph traversals

Graphs are a commonly used data structure because they can be used to model many real-world problems. The graphs translate large data into ‘easy to work with’ form.

|  |  |
| --- | --- |
| **SR. NO.** | **COMPILATION TIME** |
| 1 | 0.19 |
| 2 | 0.22 |