**LAB SESSION 10: GRAPHS**

**AIM:** To implement Graph data structure and perform the listed operations on such graphs.

**PROBLEM DEFINITION:**

Develop a C program to implement a directed graph G = (V, E) having |V|=8 and |E|=12.

Perform the following operations on the graph using an adjacency matrix:

1. Display Breadth First Traversal
2. Display Depth First Traversal

Perform the following operations on the graph using an adjacency list:

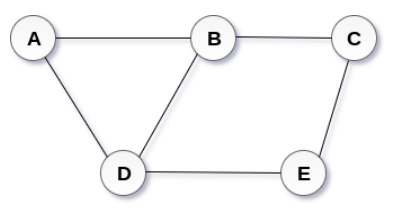
1. Add a new vertex
2. Delete a vertex
3. Add a new edge
4. Delete an edge

**THEORY:**

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labelled or unlabelled.

Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.



**Types of Graphs**

* **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
* **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
* **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

**Graph Representation**

* **Adjacency list**

Adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

* **Adjacency Matrix**

The graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.

**Breadth-First Search**

Breadth-First Search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

* BFS can be used to find the neighbouring locations from a given source location.
* In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighbouring nodes.
* BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.

**Depth -First Search**

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm

Applications of DFS algorithm

* DFS algorithm can be used to implement the topological sorting.
* It can be used to find the paths between two vertices.
* It can also be used to detect cycles in the graph.
* DFS is used to determine if a graph is bipartite or not.

**ALGORITHMS**

Breadth-First Search (BFS) Traversal:

1. Initialize an array 'state' to keep track of visited vertices.

2. Create a queue and enqueue the starting vertex.

3. Mark the starting vertex as 'visited' in the 'state' array.

4. While the queue is not empty:

 a. Dequeue a vertex from the queue and process it.

 b. For each adjacent vertex that is not visited:

 i. Enqueue the adjacent vertex.

 ii. Mark the adjacent vertex as 'visited' in the 'state' array.

5. Repeat steps 4 until the queue is empty.

Depth-First Search (DFS) Algorithm (Non-Recursive):

1. Initialize an empty stack and a 'state' array to track visited vertices.

2. Push the starting vertex onto the stack.

3. While the stack is not empty:

 a. Pop a vertex from the stack and process it.

 b. Mark the popped vertex as 'visited' in the 'state' array.

 c. Push all unvisited adjacent vertices onto the stack.

4. Repeat steps 3 until the stack is empty.

Depth-First Search (DFS) Algorithm (Recursive):

1. Initialize an array 'state' to track visited vertices.

2. Initialize a global time variable and two arrays 'd' and 'f' to store discovery and finished

times.

3. For each vertex 'v' in the graph:

 a. If 'v' is in 'initial' state in the 'state' array, call the DFS function for 'v'.

4. In the DFS function:

 a. Increment the time variable and set 'd[v]' to the current time (discovery time).

 b. Mark 'v' as 'visited' in the 'state' array.

 c. Process 'v'.

 d. For each unvisited adjacent vertex 'i':

 i. Call the DFS function for 'i'.

 e. Increment the time variable and set 'f[v]' to the current time (finished time).

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

#define initial 1

#define waiting 2

#define visited 3

int adj[MAX][MAX];

int n;

int state[MAX];

int queue[MAX], front = -1, rear = -1;

int stack[MAX], top = -1;

void create\_graph()

{

    int max\_edges, i, origin, destin;

    printf("Enter number of vertices: ");

    scanf("%d",&n);

    max\_edges = n\*(n-1);

    for(i = 1; i<=max\_edges; i++){

        printf("Enter edge %d(-1 -1 to quit): ",i);

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

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

            break;

        if(origin >= n || destin >= n || origin < 0 || destin < 0){

            printf("invalid vertex!\n");

            i--;

        }

        else

            adj[origin][destin] = 1;

    }

}

void insert\_edge(int origin, int destin)

{

    if(origin <0 || origin>=n){

        printf("origin vertex does not exist!\n");

        return;

    }

    if(destin<0 || destin>=n){

        printf("destination vertex does not exist!\n");

        return;

    }

    adj[origin][destin] = 1;

}

void display()

{

    int i,j;

    for(i = 0; i<n; i++){

        for(j = 0; j<n; j++)

            printf("%4d",adj[i][j]);

        printf("\n");

    }

}

//Breadth First Search

void insert\_queue(int vertex)

{

    if(rear == MAX -1)

        printf("Queue Overflow!\n");

    else{

        if(front == -1)

            front = 0;

        rear = rear + 1;

        queue[rear] = vertex;

    }

}

int delete\_queue()

{

    int del\_item;

    if(front == -1 || front >rear){

        printf("Queue Underflow!\n");

        exit(1);

    }

    del\_item = queue[front];

    front = front + 1;

    return del\_item;

}

int isEmpty\_queue()

{

    if(front == -1 || front > rear){

        return 1;

    }

    else{

        return 0;

    }

}

void BFS(int v)

{

    int i;

    insert\_queue(v);

    state[v] = waiting;

    while(!isEmpty\_queue()){

        v = delete\_queue();

        printf("%d\t",v);

        state[v] = visited;

        for(i = 0; i<n; i++){

            if(adj[v][i] == 1 && state[i] == initial){

                insert\_queue(i);

                state[i] = waiting;

            }

        }

    }

    printf("\n");

}

void BF\_traversal(){

    int v;

    for(v = 0; v<n; v++)

        state[v] = initial;

    printf("Enter starting vertex for breadth first traversal: ");

    scanf("%d",&v);

    BFS(v);

    for(v =0; v<n; v++)

        if(state[v] == initial)

            BFS(v);

}

int isEmpty\_stack()

{

    if(top == -1)

        return 1;

    else return 0;

}

int pop()

{

    int v;

    if(top == -1){

        printf("Stack Underflow\n");

        exit(1);

    }

    v = stack[top];

    top = top-1;

    return v;

}

void push(int v)

{

    if(top == MAX-1){

        printf("Stack Overflow! \n");

    }

    else{

        top = top +1;

        stack[top] = v;

    }

}

void DFS(int v)

{

    int i;

    push(v);

    while(!isEmpty\_stack()){

        v = pop();

        if(state[v] == initial){

            printf("%d\t",v);

            state[v] = visited;

        }

        for(i = n; i >= 0; i--){

            if(adj[v][i] == 1 && state[i] == initial)

                push(i);

        }

    }

    printf("\n");

}

void DF\_traversal()

{

    int v;

    for(v = 0; v < n; v++)

        state[v] = initial;

    printf("Enter the starting vertex: ");

    scanf("%d",&v);

    DFS(v);

    for(v = 0; v < n; v++)

        if(state[v] == initial)

            DFS(v);

}

int main()

{

    int ch, origin, destin;

    create\_graph();

    while(1){

        printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*");

        printf("\n1.Insert an edge\n2.Delete an edge\n3.Display\n4.Breadth First Traversal\n5.Depth First Traversal \n6.Exit\n");

        printf("Enter your choice: ");

        scanf("%d",&ch);

        switch(ch){

        case 1:

            printf("Enter the edge to be inserted: \n");

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

            insert\_edge(origin,destin);

            break;

        case 2:

            break;

        case 3:

            display();

            break;

        case 4:

            BF\_traversal();

            break;

        case 5:

            DF\_traversal();

            break;

        case 6:

            return 0;

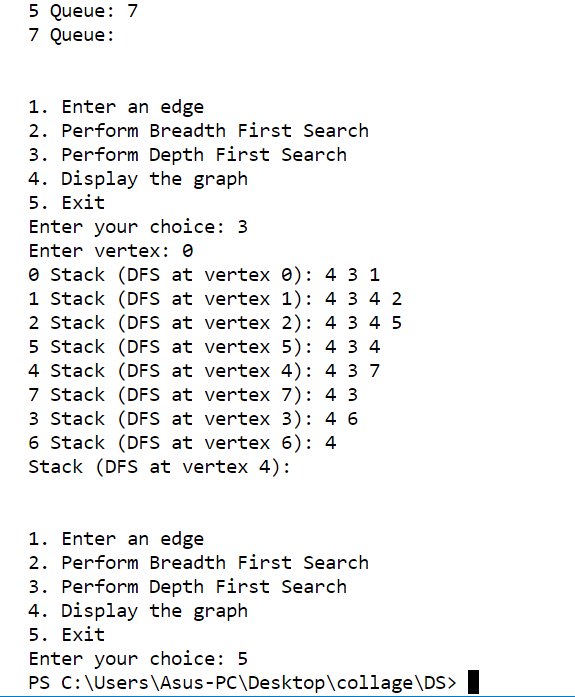
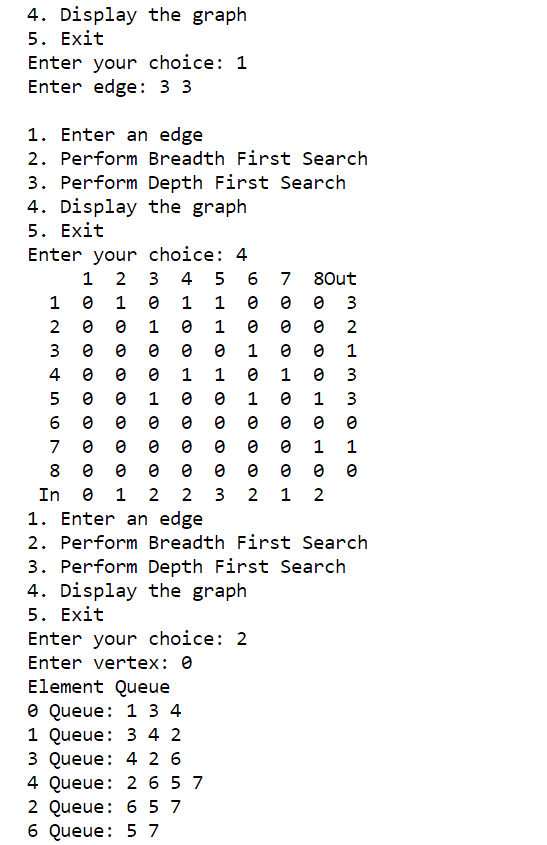
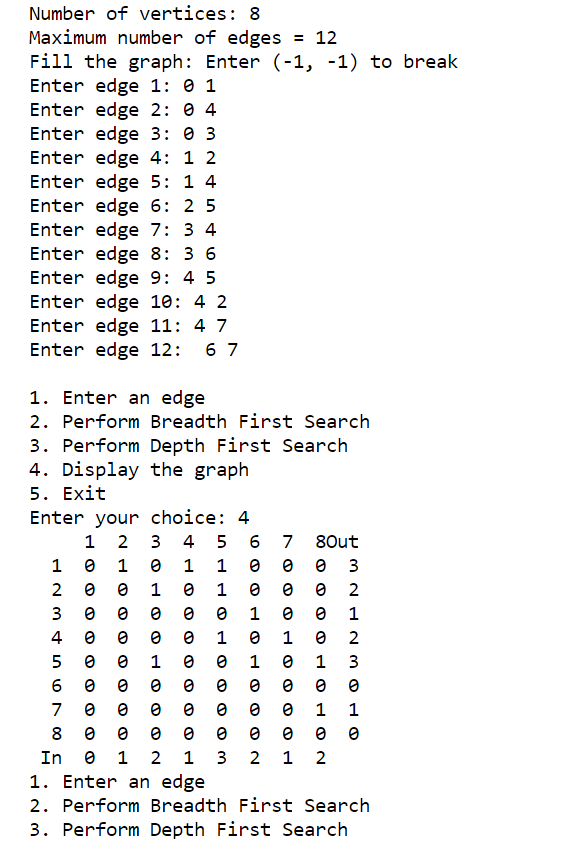
        default:

            printf("Invalid input!\n");

        }

    }

}



#include <stdio.h>

#include <stdlib.h>

int max\_edges = 12;

int n = 8;

struct vertex;

struct edge{

    struct vertex \*destvertex;

    struct edge \*nextedge;

};

struct vertex{

    struct vertex \*nextvertex;

    int info;

    struct edge \*firstedge;

} \*start = NULL;

void insertvertex(int u){

    struct vertex \*tmp, \*ptr;

    tmp = (struct vertex \*)malloc(sizeof(struct vertex));

    tmp->info = u;

    tmp->nextvertex = NULL;

    tmp->firstedge = NULL;

    if (start == NULL){

        start = tmp;

        return;

    }

    ptr = start;

    while (ptr->nextvertex != NULL)

        ptr = ptr->nextvertex;

    ptr->nextvertex = tmp;

}

struct vertex \*findvertex(int u){

    struct vertex \*ptr, \*loc;

    ptr = start;

    while (ptr != NULL){

        if (ptr->info == u){

            loc = ptr;

            return loc;

        }

        else

            ptr = ptr->nextvertex;

    }

    loc = NULL;

    return loc;

}

void insertedge(int u, int v){

    struct vertex \*locu, \*locv;

    struct edge \*ptr, \*tmp;

    locu = findvertex(u);

    locv = findvertex(v);

    if (locu == NULL || locv == NULL){

        printf("Origin or destination vertex not present\n");

        return;

    }

    tmp = (struct edge \*)malloc(sizeof(struct edge));

    tmp->destvertex = locv;

    tmp->nextedge = NULL;

    if (locu->firstedge == NULL){

        locu->firstedge = tmp;

        return;

    }

    ptr = locu->firstedge;

    while (ptr->nextedge != NULL)

        ptr = ptr->nextedge;

    ptr->nextedge = tmp;

}

void delete\_edge(int u, int v){

    struct vertex \*locu;

    struct edge \*q, \*tmp;

    locu = findvertex(u);

    if (locu == NULL || locu->firstedge == NULL){

        printf("Vertex/edge not present\n");

        return;

    }

    if (locu->firstedge->destvertex->info == v){

        tmp = locu->firstedge;

        locu->firstedge = locu->firstedge->nextedge;

        free(tmp);

        return;

    }

    q = locu->firstedge;

    while (q->nextedge != NULL){

        if (q->nextedge->destvertex->info == v){

            tmp = q->nextedge;

            q->nextedge = tmp->nextedge;

            free(tmp);

            return;

        }

        q = q->nextedge;

    }

    printf("Edge not present in the graph\n");

}

void delete\_vertex(int u){

    struct vertex \*prev = NULL;

    struct vertex \*current = start;

    struct vertex \*temp = NULL;

    while (current != NULL){

        if (current->info == u){

            if (prev == NULL)

                start = current->nextvertex;

            else

                prev->nextvertex = current->nextvertex;

            temp = current;

            current = current->nextvertex;

            free(temp);

        }

        else{

            prev = current;

            current = current->nextvertex;

        }

    }

    current = start;

    while (current != NULL){

        delete\_edge(current->info, u);

        current = current->nextvertex;

    }

}

void display(){

    struct vertex \*ptr;

    struct edge \*q;

    ptr = start;

    while (ptr != NULL){

        printf("Vertex %d ->", ptr->info);

        q = ptr->firstedge;

        while (q != NULL){

            if (q->nextedge == NULL)

                printf(" %d", q->destvertex->info);

            else

                printf(" %d->", q->destvertex->info);

            q = q->nextedge;

        }

        printf("\n");

        ptr = ptr->nextvertex;

    }

}

int main(){

    int org, dest;

    printf("Number of vertices: %d\n", n);

    int max\_edges = 12;

    for(int i = 0; i < n; i++)

        insertvertex(i);

    printf("Fill the graph: Enter (-1, -1) to break\n");

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

        printf("Enter edge %d: ", i);

        scanf("%d %d", &org, &dest);

        if(org == -1 && dest == -1)

            break;

        if(org >= n || dest >= n || org < 0 || dest < 0){

            printf("Invalid edge\n");

            i--;

        }

        insertedge(org, dest);

    }

    int s, vertex, edge;

    do{

        printf("\n1. Enter an edge\n");

        printf("2. Delete an edge\n");

        printf("3. Delete a vertex\n");

        printf("4. Display the graph\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &s);

        switch(s){

            case 1: printf("Enter edge: ");

                    scanf("%d %d", &org, &dest);

                    insertedge(org, dest);

                    break;

            case 2: printf("Enter edge: ");

                    scanf("%d %d", &org, &dest);

                    delete\_edge(org, dest);

                    break;

            case 3: printf("Enter vertex: ");

                    scanf("%d", &vertex);

                    delete\_vertex(vertex);

                    break;

            case 4: display();

                    break;

            case 5: break;

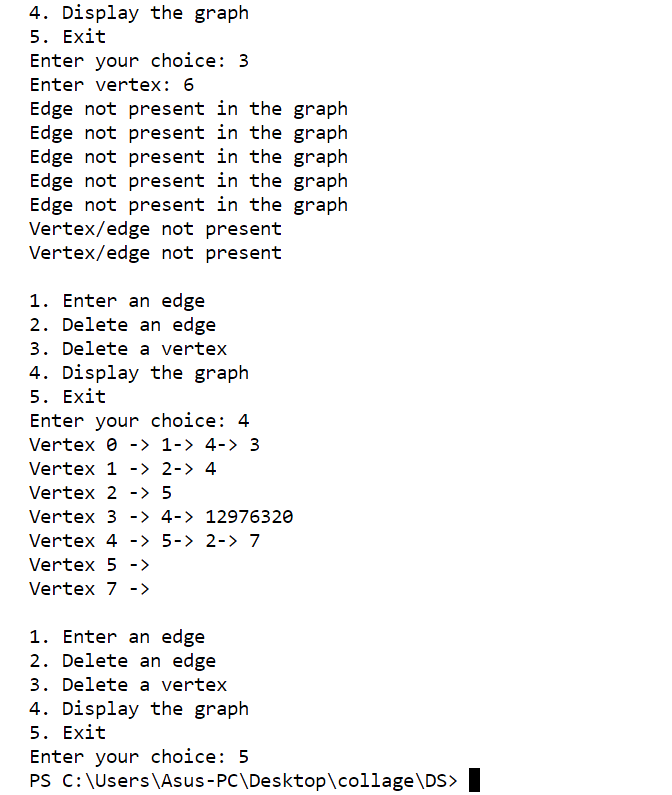
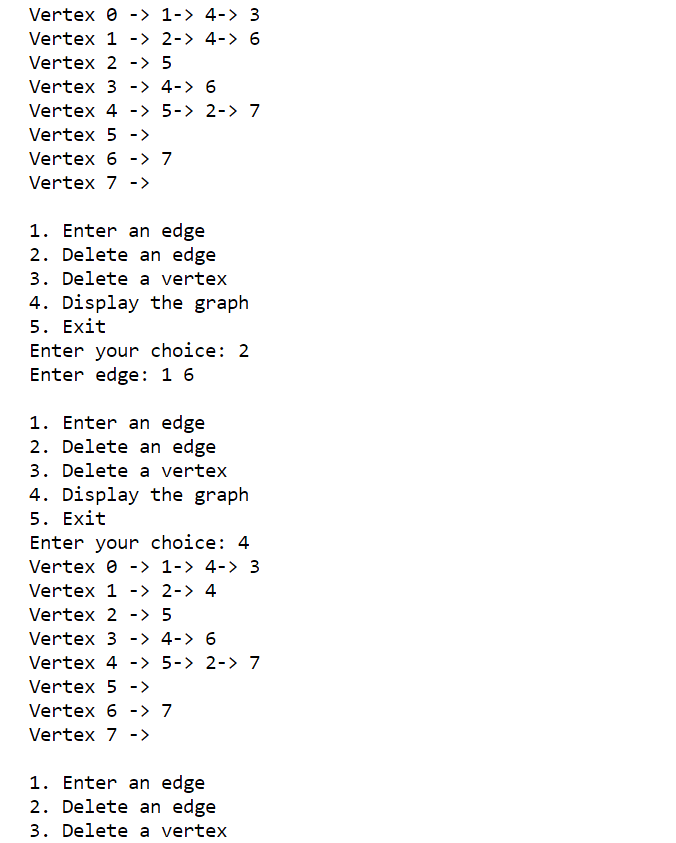
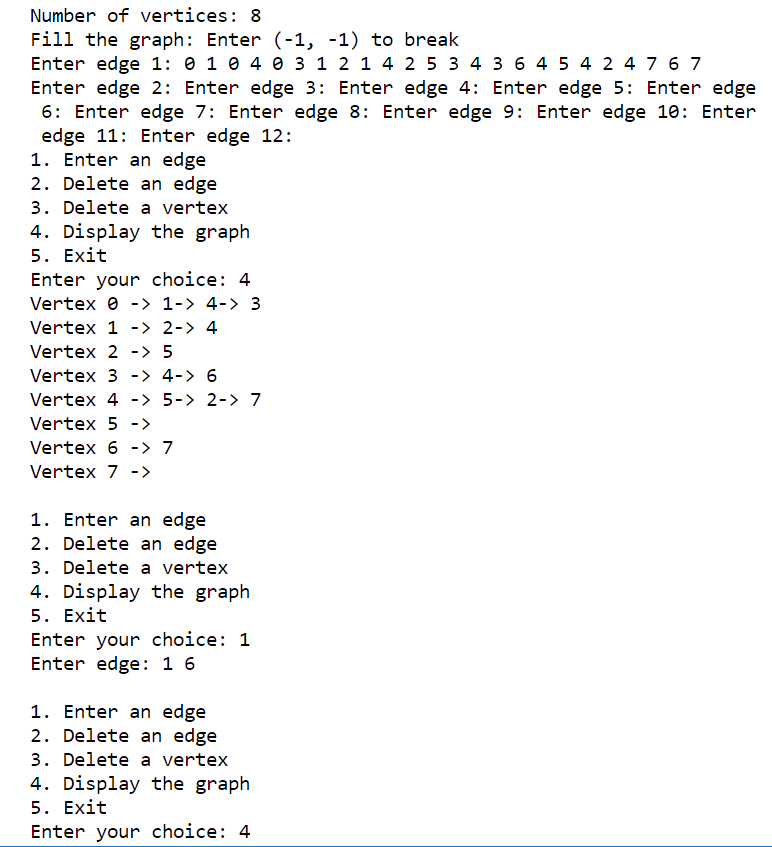
            default: printf("Invalid input\n");

        }

    }while(s != 5);

    return 0;

}



**CONCLUSION:**

Programs implementing graphs using adjacency matrix and adjacency list were written and outputs successfully obtained. BFS, DFS traversals were performed on adjacency matrix graph, while deletion of edge and vertex was done on adjacency list.