AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

Department Of Computer Engineering

Data Structure And Algorithms Lab

Group-C

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AISSMS IOIT

SE COMPTER ENGINEERING

SUBMITTED BY

Kaustubh S Kabra
ERP No.- 34
Teams No.-20

2020 -2021

Experiment Number:-6

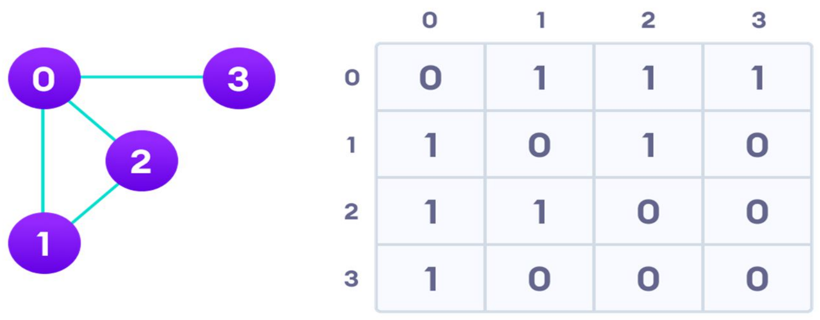
- ***Experiment Name:-***_Depth First Search And Breadth First Search_
- ***Aim:-***_Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and DFS & BFS on that._
- ***Objective:-***
    1) *To understand DFS and BFS.*
    2) *To implement program to represent graph using adjacency matrix and list.*
- ***Theory:-***

Adjacency Matrix:

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

Adjacency matrix representation:

The size of the matrix is VxV where V is the number of vertices in the graph and the value of an entry Aij is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.



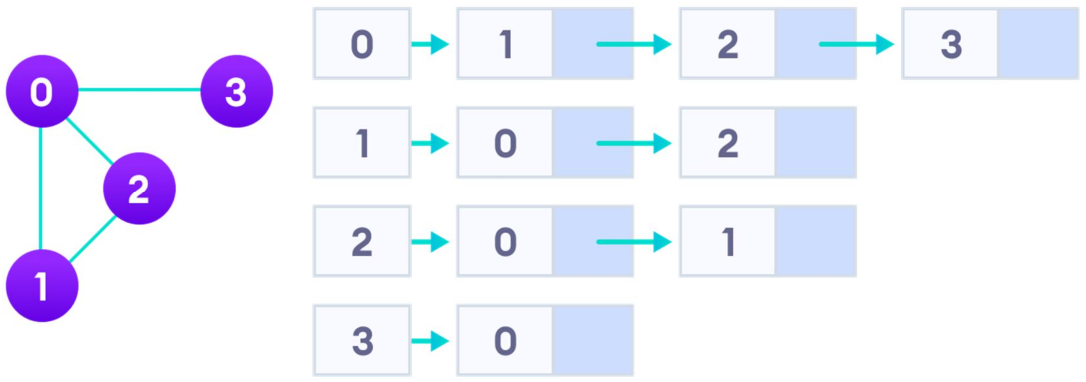
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Adjacency List:

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

Adjacency List representation:



Depth First Search (DFS):

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure.

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. *Visited*
2. *Not Visited*

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

Application of DFS :

1. *For finding the path.*
2. *To test if the graph is bipartite.*

3. For finding the strongly connected components of a graph.
4. For detecting cycles in a graph.

Breadth First Search (BFS):

Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

- **Algorithm:-**

1) Start
2) Create a matrix of size n*n where every element is 0 representing there is no edge in the graph.
3) Now, for every edge of the graph between the vertices i and j set mat[i][j] = 1.
    a. //DFS using adjacency matrix
4) After the adjacency matrix has been created and filled, call the recursive function for the source i.e. vertex 0 that will recursively call the same function for all the vertices adjacent to it.
5) Also, keep an array to keep track of the visited vertices i.e. visited[i] = true represents that vertex i has been been visited before and the DFS function for some already visited node need not be called.
    a. //BFS using adjacency matrix x

6) *After the adjacency matrix has been created and filled,*

7) *Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.*

8) *If no adjacent vertex is found, remove the first vertex from the queue.*

9) *Repeat Rule 1 and Rule 2 until the queue is empty.*

10) *Stop*

- ***Program:-***

```
#include <iostream>
#include <unordered_map>
#include <deque>
#include <stack>
#include <queue>
#include <list>
using namespace std;

class AdjacencyMatrix
{
    int vertices, edges;
    int **matrix = NULL;

public:
    AdjacencyMatrix(int vertices, int edges)
    {
        this->vertices = vertices;
        this->edges = edges;

        matrix = new (nothrow) int *[vertices];
        if (!matrix)
            throw bad_alloc();

        for (int i = 0; i < vertices; i++)
        {
            matrix[i] = new (nothrow) int[vertices];
            if (!matrix[i])
                throw bad_alloc();
        }
    }
    ~AdjacencyMatrix()
    {
        delete[] matrix;
    }
    void create();
    void display();
    void DepthFirstSearch(deque<bool> &);
};

void AdjacencyMatrix::create()
{
    for (int i = 0; i < vertices; i++)
    {
        for (int j = 0; j < vertices; j++)
        {
            matrix[i][j] = 0;
        }
    }

    int i, j;
    for (int k = 0; k < edges; k++)
    {
        cout << "\nEnter The Pair of Vertices For Edge " << k + 1 << " : ";
        cin >> i >> j;
        matrix[i][j] = 1;
        matrix[j][i] = 1;
    }
}

void AdjacencyMatrix::display()
```

```
{
    for (int i = 0; i < vertices; i++)
    {
        for (int j = 0; j < vertices; j++)
        {
            cout << " " << matrix[i][j];
        }
        cout << endl;
    }
}

void AdjacencyMatrix::DepthFirstSearch(deque<bool> &visited)
{
    stack<int> st;
    st.push(0);

    while (!st.empty())
    {
        int vertex = st.top();
        st.pop();

        if (!visited[vertex])
        {
            cout << vertex << " ";
            visited[vertex] = 1;
            for (int i = 0; i < vertices; i++)
            {
                if (matrix[vertex][i] == 1 and !visited[i])
                {
                    st.push(i);
                }
            }
        }
    }
    cout << endl;
}
```

```
class AdjacencyList
{
    int vertices, edges;
    unordered_map<int, list<int>> *adjacencylist;

public:
    AdjacencyList(int vertices, int edges)
    {
        this->vertices = vertices;
        this->edges = edges;
        adjacencylist = new (nothrow) unordered_map<int, list<int>>;
        if (!adjacencylist)
            throw bad_alloc();
    }
    ~AdjacencyList()
    {
        delete[] adjacencylist;
    }
    void create();
    void BreadthFirstSearch(deque<bool> &);
};

void AdjacencyList::create()
{
    int i, j;
    for (int k = 0; k < edges; k++)
    {
        cout << "\nEnter The Pair of Vertices For Edge " << k + 1 << " : ";
        cin >> i >> j;
        (*adjacencylist)[i].push_back(j);
    }
}

void AdjacencyList::BreadthFirstSearch(deque<bool> &visited)
{
    queue<int> que;
    que.push(0);

    while (!que.empty())
    {
        int vertex = que.front();
        cout << vertex << " ";
        visited[vertex] = true;
        que.pop();

        list<int>::iterator iter;
        for (iter = (*adjacencylist)[vertex].begin(); iter != (*adjacencylist)[vertex].end();
iter++)
        {
            if (!visited[*iter])
            {
                visited[*iter] = true;
                que.push(*iter);
            }
        }
    }
    cout << endl;
}

int main()
{
    int vertices, edges;
    int userInput;
    AdjacencyMatrix *adjm = NULL;
    AdjacencyList *adjl = NULL;

    while (true)
    {
        cout << "\n1. Create Graph Using Adjacency Matrix\n2. Display Adjacency
Matrix\n3. Create Graph Using Adjacency List\n4. Depth First Traversal\n5.
Breadth First Traversal\n6. Exit\n>>>> ";
```

```
cin >> userInput;

switch (userInput)
{
case 1:
    cout << "\nEnter No. of Vertices: ";
    cin >> vertices;
    cout << "\nEnter No. of Edges: ";
    cin >> edges;
    adjm = new (nothrow) AdjacencyMatrix(vertices, edges);
    adjm->create();
    break;

case 2:
    if (adjm == NULL)
    {
        cout << "\nGraph is Empty!\n";
        break;
    }
    cout << "\nAdjacency Matrix: "
         << "\n";
    adjm->display();
    break;

case 3:
    cout << "\nEnter No. of Vertices: ";
    cin >> vertices;
    cout << "\nEnter No. of Edges: ";
    cin >> edges;
    adjl = new (nothrow) AdjacencyList(vertices, edges);
    adjl->create();
    break;

case 4:
{
    if (adjm == NULL)
```

```
            {
                cout << "\nGraph is Empty!\n";
                break;
            }
            deque<bool> visited(vertices, false);
            cout << "\nDepth First Search Traversal: ";
            adjm->DepthFirstSearch(visited);
            break;
        }
        case 5:
        {
            if (adjl == NULL)
            {
                cout << "\nGraph is Empty!\n";
                break;
            }
            deque<bool> visited(vertices, false);
            cout << "\nBreadth First Search Traversal: ";
            adjl->BreadthFirstSearch(visited);
            break;
        }
        case 6:
            exit(0);
            break;

        default:
            cout << "\nPlease Enter Correct Input!\n";
            break;
        }
    }
    delete adjm;
    delete adjl;
    return 0;
}
```

- ***Output:-***

```
// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 2

// Graph is Empty!

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 4

// Graph is Empty!

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 5

// Graph is Empty!

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 1

// Enter No. of Vertices: 4

// Enter No. of Edges: 6

// Enter The Pair of Vertices For Edge 1 : 0 1

// Enter The Pair of Vertices For Edge 2 : 0 2

// Enter The Pair of Vertices For Edge 3 : 0 3

// Enter The Pair of Vertices For Edge 4 : 1 2

// Enter The Pair of Vertices For Edge 5 : 1 3

// Enter The Pair of Vertices For Edge 6 : 2 3

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 2

// Adjacency Matrix:
//  0 1 1 1
//  1 0 1 1
//  1 1 0 1
//  1 1 1 0

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 4

// Depth First Search Traversal: 0 3 2 1

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 5

// Graph is Empty!

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 3

// Enter No. of Vertices: 4

// Enter No. of Edges: 6

// Enter The Pair of Vertices For Edge 1 : 0 1

// Enter The Pair of Vertices For Edge 2 : 0 2

// Enter The Pair of Vertices For Edge 3 : 0 3
// Enter The Pair of Vertices For Edge 4 : 1 2

// Enter The Pair of Vertices For Edge 5 : 1 3

// Enter The Pair of Vertices For Edge 6 : 2 3

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 5

// Breadth First Search Traversal: 0 1 2 3

// 1. Create Graph Using Adjacency Matrix
// 2. Display Adjacency Matrix
// 3. Create Graph Using Adjacency List
// 4. Depth First Traversal
// 5. Breadth First Traversal
// 6. Exit
// >>>> 6
```

- ***Analysis:-***

Operations	Adjacency Matrix	Adjacency List
Storage Space	This representation makes use of VxV matrix, so space required in worst case is $O(\|V\|^2)$.	In this representation, for every vertex we store its neighbours. In the worst case, if a graph is connected O(V) is required for a vertex and O(E) is required for storing neighbours

Operations	Adjacency Matrix	Adjacency List
		corresponding to every vertex .Thus, overall space complexity is O(|V|+|E|).
Adding a vertex	*In order to add a new vertex to VxV matrix the storage must be increases to (|V|+1)^2. To achieve this we need to copy the whole matrix. Therefore the complexity is* ***O(|V|²).***	*There are two pointers in adjacency list first points to the front node and the other one points to the rear node.Thus insertion of a vertex can be done directly in* ***O(1) time.***
Adding an edge	*To add an edge say from i to j, matrix[i][j] = 1 which requires* ***O(1)*** *time.*	*Similar to insertion of vertex here also two pointers are used pointing to the rear and front of the list. Thus, an edge can be inserted in* ***O(1)****time.*
Removing a vertex	*In order to remove a vertex from V*V matrix the storage must be decreased to |V|^2 from (|V|+1)^2. To achieve this we need to copy the whole matrix. Therefore the complexity is* ***O(|V|²).***	*In order to remove a vertex, we need to search for the vertex which will require O(|V|) time in worst case, after this we need to traverse the edges and in worst case it will require O(|E|) time.Hence, total time complexity is* ***O(|V|+|E|).***
Removing an edge	*To remove an edge say from i to j, matrix[i][j] = 0 which requires* ***O(1)*** *time.*	*To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges.Thus, the time complexity is* ***O(|E|).***
Querying	*In order to find for an existing edge the content of matrix needs to be checked. Given two vertices say i and j matrix[i][j] can be checked in* ***O(1)*** *time.*	*In an adjacency list every vertex is associated with a list of adjacent vertices. For a given graph, in order to check for an edge we need to check for vertices adjacent to given vertex. A vertex can have at most O(|V|)*

Operations	Adjacency Matrix	Adjacency List
		neighbours and in worst can we would have to check for every adjacent vertex. Therefore, time complexity is $O(\|V\|)$.

Conclusion:- *Hence, we have studied and implemented graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS.*

Experiment Number:-7

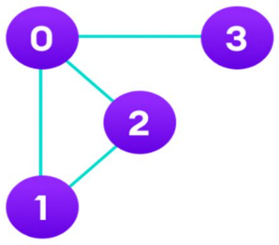
- **Experiment Name:-** *Adjacency list representation of the graph or use Adjacency matrix representation of the graph.*
- **Aim:-** *There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.*
- **Objective:-**
    1) *To understand graph using Adjacency matrix.*
    2) *To understand graph using Adjacency list representation*
- **Theory:-**

Graph Data Structure:

A graph data structure is a collection of nodes that have data and are connected to other nodes.

More precisely, a graph is a data structure (V, E) that consists of

1. *A collection of vertices V.*
2. *A collection of edges E, represented as ordered pairs of vertices (u,v).*



In the graph,

```
V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}
```

Graph Terminology:

1. *Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.*
2. *Path: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.*
3. *Directed Graph: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.*

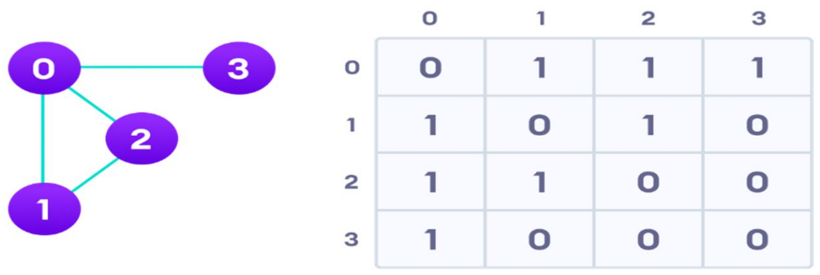
Graph Representation:

Graphs are commonly represented in two ways:

• ***Adjacency Matrix:***

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

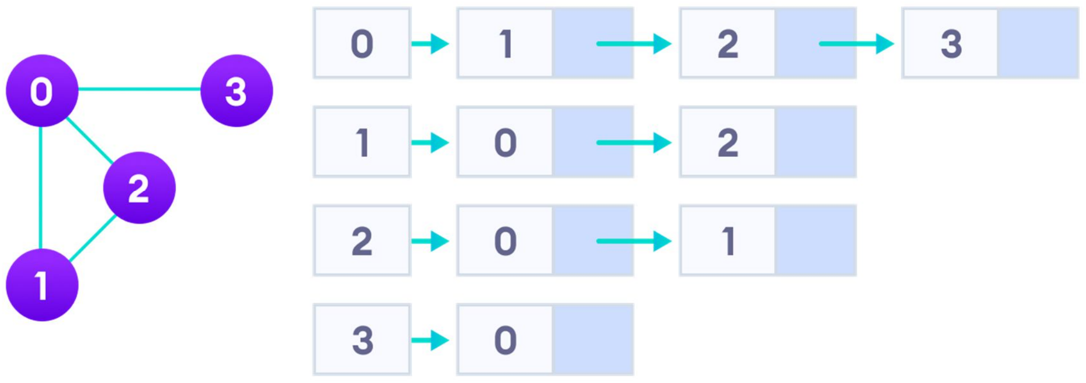


	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

• ***Adjacency List:***

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.



Graph Operations:

The most common graph operations are:

1. *Check if the element is present in the graph*
2. *Graph Traversal*
3. *Add elements(vertex, edges) to graph*

4. *Finding the path from one vertex to another*

- ***Algorithm:-***

1) *Start*
2) *Create a 2D array (size NxN)*
3) *Initialize all value of this matrix to zero.*
4) *For each edge(flight hour) between two cities in arr[][](say* ***X and Y****), Update value at* ***Adj[X][Y]*** *and* ***Adj[Y][X]*** *to 1, denotes that there is an edge between X and Y.*
    a. *//DFS using adjacency matrix*
5) *Now, for every edge of the graph between the vertices i and j set mat[i][j] = number of hour .*
6) *After the adjacency matrix has been created and filled, call the recursive function for the source i.e. vertex 0 that will recursively call the same function for all the vertices adjacent to it.*
7) *Also, keep an array to keep track of the visited vertices i.e. visited[i] = true represents that vertex i has been been visited before and the DFS function for some already visited node need not be called.*
    a. *//BFS using adjacency matrix*
8) *Create a matrix of size n*n where every element is 0 representing there is no edge in the graph.*
9) *Now, for every edge of the graph between the vertices i and j set mat[i][j] = 1.*
10) *After the adjacency matrix has been created and filled,*
11) *Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.*

12)If no adjacent vertex is found, remove the first vertex from the queue.

13)Repeat Rule 1 and Rule 2 until the queue is empty.

14)Stop

- **Program:-**

```
 #include<iostream>
 #include<stdlib.h>
 #include<string.h>
using namespace std;
struct node
{   string vertex;
    int time;
    node *next;
};
class adjmatlist
{    int m[10][10],n,i,j; char ch;  string v[20];   node *head[20];  node *temp=NULL;

     public:
     adjmatlist()
     {     for(i=0;i<20;i++)
           {    head[i]=NULL;  }
     }
     void getgraph();
     void adjlist();

     void displaym();
     void displaya();
};
void adjmatlist::getgraph()
{
   cout<<"\n enter no. of cities(max. 20)";
   cin>>n;
   cout<<"\n enter name of cities";
```

```
  for(i=0;i<n;i++)
     cin>>v[i];
  for(i=0;i<n;i++)
  {
     for(j=0;j<n;j++)
     {  cout<<"\n if path is present between city "<<v[i]<<" and "<<v[j]<<" then press
enter y otherwise n";
         cin>>ch;
         if(ch=='y')
         {
           cout<<"\n enter time required to reach city "<<v[j]<<" from "<<v[i]<<" in
minutes";
           cin>>m[i][j];
         }
         else if(ch=='n')
         {  m[i][j]=0;  }
         else
         { cout<<"\n unknown entry";  }
     }
  }
     adjlist();

}
void adjmatlist::adjlist()
{     cout<<"\n ****";
      for(i=0;i<n;i++)
      {  node *p=new(struct node);
         p->next=NULL;
         p->vertex=v[i];
         head[i]=p;      cout<<"\n"<<head[i]->vertex;
      }

      for(i=0;i<n;i++)
      { for(j=0;j<n;j++)
         {
                 if(m[i][j]!=0)
```

```
                {
                      node *p=new(struct node);
                      p->vertex=v[j];
                      p->time=m[i][j];
                      p->next=NULL;
                      if(head[i]->next==NULL)
                      {  head[i]->next=p;   }
                      else
                      {  temp=head[i];
                      while(temp->next!=NULL)
                      {   temp=temp->next;  }
                          temp->next=p;
                      }

                }

        }
      }

}
void adjmatlist::displaym()
{    cout<<"\n";
    for(j=0;j<n;j++)
    {  cout<<"\t"<<v[j];  }

    for(i=0;i<n;i++)
    {  cout<<"\n "<<v[i];
      for(j=0;j<n;j++)
      {   cout<<"\t"<<m[i][j];
      }
          cout<<"\n";
    }
}
void adjmatlist::displaya()
{
      cout<<"\n adjacency list is";
```

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


            if(head[i]==NULL)
            {   cout<<"\n adjacency list not present";  break;   }
            else
            {
              cout<<"\n"<<head[i]->vertex;
            temp=head[i]->next;
            while(temp!=NULL)
            {  cout<<"-> "<<temp->vertex;
              temp=temp->next;  }

            }





}

  cout<<"\n path and time required to reach cities is";

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


            if(head[i]==NULL)
            {   cout<<"\n adjacency list not present";  break;   }
            else
            {

            temp=head[i]->next;
            while(temp!=NULL)
            {  cout<<"\n"<<head[i]->vertex;
```

```
                        cout<<"-> "<<temp->vertex<<"\n   [time required: "<<temp->time<<"
min ]";
                        temp=temp->next;  }

                     }




      }
}
int main()
{  int m;
   adjmatlist a;

   while(1)
   {
   cout<<"\n\n enter the choice";
   cout<<"\n 1.enter graph";
   cout<<"\n 2.display adjacency matrix for cities";
   cout<<"\n 3.display adjacency list for cities";
   cout<<"\n 4.exit";
   cin>>m;

        switch(m)
      {             case 1: a.getgraph();
                            break;
                   case 2: a.displaym();
                            break;

                        case 3: a.displaya();
                               break;
                         case 4: exit(0);

                         default:  cout<<"\n unknown choice";
       }
```

```
    }
    return 0;
}
```

- ***Output:-***

```
Enter the choice
 1.Enter graph
 2.Display adjacency matrix for cities
 3.Display adjacency list for cities
 4.Exit
1
 Enter no. of cities(max. 20)7

 Enter name of cities
Jalgaon
Pune
Mumbai
Delhi
Bangalore
Indore
Kolkata

 if path is present between city Jalgaon and Jalgaon then press enter y otherwise n
n

 if path is present between city Jalgaon and Pune then press enter y otherwise n
y

 enter time required to reach city Pune from Jalgaon in minutes
120
 if path is present between city Jalgaon and Mumbai then press enter y otherwise n
y

 enter time required to reach city Mumbai from Jalgaon in minutes
180
```

```
 if path is present between city Jalgaon and Delhi then press enter y otherwise n
y

 enter time required to reach city Delhi from Jalgaon in minutes
360

 if path is present between city Jalgaon and Banglore then press enter y otherwise n
y

 enter time required to reach city Banglore from Jalgaon in minutes
240

 if path is present between city Jalgaon and Indore then press enter y otherwise n
y

 enter time required to reach city Indore from Jalgaon in minutes
60

 if path is present between city Jalgaon and Kolkata then press enter y otherwise n
y

 enter time required to reach city Kolkata from Jalgaon in minutes
420

 if path is present between city Pune and Jalgaon then press enter y otherwise n
y

 enter time required to reach city Jalgaon from Pune in minutes
120

 if path is present between city Pune and Pune then press enter y otherwise n
n

 if path is present between city Pune and Mumbai then press enter y otherwise n
y
```

```
 enter time required to reach city Mumbai from Pune in minutes
60

 if path is present between city Pune and Delhi then press enter y otherwise n
y

 enter time required to reach city Delhi from Pune in minutes
420

 if path is present between city Pune and Banglore then press enter y otherwise n
y

 enter time required to reach city Banglore from Pune in minutes
180

 if path is present between city Pune and Indore then press enter y otherwise n
y

 enter time required to reach city Indore from Pune in minutes
180

 if path is present between city Pune and Kolkata then press enter y otherwise n
y

 enter time required to reach city Kolkata from Pune in minutes
480

 if path is present between city Mumbai and Jalgaon then press enter y otherwise n
y

 enter time required to reach city Jalgaon from Mumbai in minutes
180

 if path is present between city Mumbai and Pune then press enter y otherwise n
y
```

```
 enter time required to reach city Pune from Mumbai in minutes
60

 if path is present between city Mumbai and Mumbai then press enter y otherwise n
n

 if path is present between city Mumbai and Delhi then press enter y otherwise n
y

 enter time required to reach city Delhi from Mumbai in minutes
420

 if path is present between city Mumbai and Banglore then press enter y otherwise n
y

 enter time required to reach city Banglore from Mumbai in minutes
240

 if path is present between city Mumbai and Indore then press enter y otherwise n
y

 enter time required to reach city Indore from Mumbai in minutes
240

 if path is present between city Mumbai and Kolkata then press enter y otherwise n
y

 enter time required to reach city Kolkata from Mumbai in minutes
540

 if path is present between city Delhi and Jalgaon then press enter y otherwise n
y

 enter time required to reach city Jalgaon from Delhi in minutes
360
```

```
 if path is present between city Delhi and Pune then press enter y otherwise n
y

 enter time required to reach city Pune from Delhi in minutes
420

 if path is present between city Delhi and Mumbai then press enter y otherwise n
y

 enter time required to reach city Mumbai from Delhi in minutes
420

 if path is present between city Delhi and Delhi then press enter y otherwise n
n

 if path is present between city Delhi and Banglore then press enter y otherwise n
y

 enter time required to reach city Banglore from Delhi in minutes
540

 if path is present between city Delhi and Indore then press enter y otherwise n
y

 enter time required to reach city Indore from Delhi in minutes
420

 if path is present between city Delhi and Kolkata then press enter y otherwise n
y

 enter time required to reach city Kolkata from Delhi in minutes
300

 if path is present between city Banglore and Jalgaon then press enter y otherwise n
y
```

```
 enter time required to reach city Jalgaon from Banglore in minutes
240

 if path is present between city Banglore and Pune then press enter y otherwise n
y

 enter time required to reach city Pune from Banglore in minutes
180

 if path is present between city Banglore and Mumbai then press enter y otherwise n
y

 enter time required to reach city Mumbai from Banglore in minutes
240

 if path is present between city Banglore and Delhi then press enter y otherwise n
y

 enter time required to reach city Delhi from Banglore in minutes
540

 if path is present between city Banglore and Banglore then press enter y otherwise n
n

 if path is present between city Banglore and Indore then press enter y otherwise n
y

 enter time required to reach city Indore from Banglore in minutes
300

 if path is present between city Banglore and Kolkata then press enter y otherwise n
y

 enter time required to reach city Kolkata from Banglore in minutes
600
```

```
 if path is present between city Indore and Jalgaon then press enter y otherwise n
y

 enter time required to reach city Jalgaon from Indore in minutes
60

 if path is present between city Indore and Pune then press enter y otherwise n
y

 enter time required to reach city Pune from Indore in minutes
180

 if path is present between city Indore and Mumbai then press enter y otherwise n
y

 enter time required to reach city Mumbai from Indore in minutes
240

 if path is present between city Indore and Delhi then press enter y otherwise n
y

 enter time required to reach city Delhi from Indore in minutes
420

 if path is present between city Indore and Banglore then press enter y otherwise n
y

 enter time required to reach city Banglore from Indore in minutes
360

 if path is present between city Indore and Indore then press enter y otherwise n
n

 if path is present between city Indore and Kolkata then press enter y otherwise n
y
```

```
 enter time required to reach city Kolkata from Indore in minutes
420

 if path is present between city Kolkata and Jalgaon then press enter y otherwise n
y

 enter time required to reach city Jalgaon from Kolkata in minutes
420

 if path is present between city Kolkata and Pune then press enter y otherwise n
y

 enter time required to reach city Pune from Kolkata in minutes
480

 if path is present between city Kolkata and Mumbai then press enter y otherwise n
y

 enter time required to reach city Mumbai from Kolkata in minutes
540

 if path is present between city Kolkata and Delhi then press enter y otherwise n
y

 enter time required to reach city Delhi from Kolkata in minutes
300

 if path is present between city Kolkata and Banglore then press enter y otherwise n
y

 enter time required to reach city Banglore from Kolkata in minutes
600

 if path is present between city Kolkata and Indore then press enter y otherwise n
y
```

```
 enter time required to reach city Indore from Kolkata in minutes
420

 if path is present between city Kolkata and Kolkata then press enter y otherwise n
n

 ****
Jalgaon
Pune
Mumbai
Delhi
Bangalore
Indore
Kolkata

 Enter the choice
 1.Enter graph
 2.Display adjacency matrix for cities
 3.Display adjacency list for cities
 4.Exit
2

            Jalgaon   Pune   Mumbai   Delhi   Bangalore   Indore   Kolkata
 Jalgaon       0       120     180      360      240        60       420

 Pune         120       0      60       420      180        180      480

 Mumbai      180       60       0       420      240        240      540

 Delhi       360       420     420       0       540        420      300

 Bangalore   240       180     240      540       0         300      600

 Indore       60       180     240      420      360         0       420

 Kolkata     420      480      540      300      600        420      0
 Enter the choice
 1.Enter graph
 2.Display adjacency matrix for cities
 3.Display adjacency list for cities
 4.Exit3

 Adjacency list is
Jalgaon-> Pune-> Mumbai-> Delhi-> Bangalore-> Indore-> Kolkata
Pune-> Jalgaon-> Mumbai-> Delhi-> Bangalore-> Indore-> Kolkata
Mumbai-> Jalgaon-> Pune-> Delhi-> Bangalore-> Indore-> Kolkata
Delhi-> Jalgaon-> Pune-> Mumbai-> Bangalore-> Indore-> Kolkata
Bangalore-> Jalgaon-> Pune-> Mumbai-> Delhi-> Indore-> Kolkata
Indore-> Jalgaon-> Pune-> Mumbai-> Delhi-> Bangalore-> Kolkata
Kolkata-> Jalgaon-> Pune-> Mumbai-> Delhi-> Bangalore-> Indore
 path and time required to reach cities is
Jalgaon-> Pune
  [time required: 120 min ]
Jalgaon-> Mumbai
  [time required: 180 min ]
Jalgaon-> Delhi
  [time required: 360 min ]
Jalgaon-> Bangalore
  [time required: 240 min ]
Jalgaon-> Indore
  [time required: 60 min ]
Jalgaon-> Kolkata
  [time required: 420 min ]
Pune-> Jalgaon
  [time required: 120 min ]
Pune-> Mumbai
  [time required: 60 min ]
Pune-> Delhi
  [time required: 420 min ]
Pune-> Bangalore
```

```
  [time required: 180 min ]
Pune-> Indore
  [time required: 180 min ]
Pune-> Kolkata
  [time required: 480 min ]
Mumbai-> Jalgaon
  [time required: 180 min ]
Mumbai-> Pune
  [time required: 60 min ]
Mumbai-> Delhi
  [time required: 420 min ]
Mumbai-> Bangalore
  [time required: 240 min ]
Mumbai-> Indore
  [time required: 240 min ]
Mumbai-> Kolkata
  [time required: 540 min ]
Delhi-> Jalgaon
  [time required: 360 min ]
Delhi-> Pune
  [time required: 420 min ]
Delhi-> Mumbai
  [time required: 420 min ]
Delhi-> Bangalore
  [time required: 540 min ]
Delhi-> Indore
  [time required: 420 min ]
Delhi-> Kolkata
  [time required: 300 min ]
Bangalore-> Jalgaon
  [time required: 240 min ]
Bangalore-> Pune
  [time required: 180 min ]
Bangalore-> Mumbai
  [time required: 240 min ]
Bangalore-> Delhi
```

```
  [time required: 540 min ]
Bangalore-> Indore
  [time required: 300 min ]
Bangalore-> Kolkata
  [time required: 600 min ]
Indore-> Jalgaon
  [time required: 60 min ]
Indore-> Pune
  [time required: 180 min ]
Indore-> Mumbai
  [time required: 240 min ]
Indore-> Delhi
  [time required: 420 min ]
Indore-> Banglore
  [time required: 360 min ]
Indore-> Kolkata
  [time required: 420 min ]
Kolkata-> Jalgaon
  [time required: 420 min ]
Kolkata-> Pune
  [time required: 480 min ]
Kolkata-> Mumbai
  [time required: 540 min ]
Kolkata-> Delhi
  [time required: 300 min ]
Kolkata-> Bangalore
  [time required: 600 min ]
Kolkata-> Indore
  [time required: 420 min ]

 Enter the choice
 1.Enter graph
 2.Display adjacency matrix for cities
 3.Display adjacency list for cities
 4.Exit
4
```

- ***Analysis:-***

Operations	*Adjacency Matrix*	*Adjacency List*
Storage Space	*This representation makes use of VxV matrix, so space required in worst case is* **$O(\|V\|^2)$**.	*In this representation, for every vertex we store its neighbours. In the worst case, if a graph is connected O(V) is required for a vertex and O(E) is required for storing neighbours corresponding to every vertex .Thus, overall space complexity is $O(\|V\|+\|E\|)$.*
Adding a vertex	*In order to add a new vertex to VxV matrix the storage must be increases to $(\|V\|+1)^2$. To achieve this we need to copy the whole matrix. Therefore the complexity is* **$O(\|V\|^2)$**.	*There are two pointers in adjacency list first points to the front node and the other one points to the rear node.Thus insertion of a vertex can be done directly in* ***O(1) time.***
Adding an edge	*To add an edge say from i to j, matrix[i][j] = 1 which requires* **O(1)** *time.*	*Similar to insertion of vertex here also two pointers are used pointing to the rear and front of the list. Thus, an edge can be inserted in* **O(1)***time.*
Removing a vertex	*In order to remove a vertex from V*V matrix the storage must be decreased to $\|V\|^2$ from $(\|V\|+1)^2$. To achieve this we need to copy the whole matrix. Therefore the complexity is* **$O(\|V\|^2)$**.	*In order to remove a vertex, we need to search for the vertex which will require $O(\|V\|)$ time in worst case, after this we need to traverse the edges and in worst case it will require $O(\|E\|)$ time.Hence, total time complexity is* **$O(\|V\|+\|E\|)$**.
Removing an edge	*To remove an edge say from i to j, matrix[i][j] = 0 which requires* **O(1)** *time.*	*To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges.Thus, the time complexity is* **$O(\|E\|)$**.

Operations	*Adjacency Matrix*	*Adjacency List*
Querying	*In order to find for an existing edge the content of matrix needs to be checked. Given two vertices say i and j matrix[i][j] can be checked in* **O(1)** *time.*	*In an adjacency list every vertex is associated with a list of adjacent vertices. For a given graph, in order to check for an edge we need to check for vertices adjacent to given vertex. A vertex can have at most O(|V|) neighbours and in worst can we would have to check for every adjacent vertex. Therefore, time complexity is* **O(|V|)** *.*

- ***Conclusion:-*** *Hence, we have studied and implemented graph using Adjacency Matrix and Adjacency List on Flights from one city to other city.*