```
package for_DSA;
// using adjacency matrix
import java.util.Scanner;

class myGraph{
    int adj[][];
    int n;
    public myGraph(int i)
    {
        n = i;
        adj = new int[n][n];
    }
    public void create_graph()
    {
        int i,max_edges,origin,destin;

        Scanner sc=new Scanner(System.in);
        max_edges = n*(n-1); /*directed graph*/

        for(i=1; i<=max_edges; i++)
        {
            System.out.println("Enter edge (-1 -1) to quit : ");
            origin = sc.nextInt();
            destin = sc.nextInt();
            if((origin==-1) && (destin==-1))
                break;
            if(origin>=n || destin>=n || origin<0 || destin<0)
            {
                System.out.println("Invalid edge!\n");
                i--;
            }
            else
                adj[origin][destin] = 1;
        }/*End of for*/
```

```
    //sc.close();
}/*End of create_graph()*/

public void insert_edge(int origin,int destin)
{
    if(origin<0 || origin>=n)
    {
        System.out.print("Origin vertex does not exist\n");
        return;
    }
    if(destin<0 || destin>=n)
    {
        System.out.print("Destination vertex does not exist\n");
        return;
    }
    adj[origin][destin] = 1;
}/*End of insert_edge()*/

public void del_edge(int origin, int destin)
{
    if(origin<0 || origin>=n || destin<0 || destin>=n || adj[origin][destin]==0)
    {
        System.out.print("This edge does not exist\n");
        return;
    }
    adj[origin][destin] = 0;
}/*End of del_edge()*/


public void display()
{
    int i,j;
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
            System.out.print("   "+adj[i][j]);
```

```
            System.out.print("\n");
        }
    }/*End of display()*/


}
public class GraphDemo {

    public static void main(String[] args) {
        int choice,origin,destin;
        myGraph g=new myGraph(5);
        g.create_graph();
        Scanner sc=new Scanner(System.in);
        do
        {
            System.out.print("1.Insert an edge\n");
            System.out.print("2.Delete an edge\n");
            System.out.print("3.Display\n");
            System.out.print("4.Exit\n");
            System.out.print("Enter your choice : ");
            choice = sc.nextInt();

            switch(choice)
            {
             case 1:
                System.out.print("Enter an edge to be inserted : ");
                origin = sc.nextInt();
                destin = sc.nextInt();
                g.insert_edge(origin,destin);
                break;
             case 2:
                System.out.print("Enter an edge to be deleted : ");
                origin = sc.nextInt();
                destin = sc.nextInt();
                g.del_edge(origin,destin);
                break;
```

```
            case 3:
               g.display();
               break;
            case 4:
               break;
            default:
               System.out.print("Wrong choice\n");
               break;
            }/*End of switch*/
         }while(choice!=4);/*End of while*/
       sc.close();
    }

}
```

```
package for_DSA;
/* ************************************************************
   Graph using adjacency list
*/
import java.util.Scanner;

//class Edge;

class Vertex
{
    int info;
    Vertex nextVertex;  /*next vertex in the linked list of vertices*/
    Edge firstEdge;  /*first Edge of the adjacency list of this vertex*/
};

class Edge
{
    Vertex destVertex;  /*Destination vertex of the Edge*/
    Edge nextEdge; /*next Edge of the adjacency list*/
};
```

```
class GraphLL {
  Vertex start;

    GraphLL()
    {
        start = null;
    }

    void insertVertex(int u)
    {
        Vertex tmp,ptr;
        tmp = new 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;
    }/*End of insertVertex()*/

    void deleteVertex(int u)
    {
        Vertex tmp,q;
        Edge p,temporary;
        if(start == null)
        {
            System.out.print("No vertices to be deleted\n");
            return;
```

```
    }
    if(start.info == u)/* Vertex to be deleted is first vertex of list*/
    {
        tmp = start;
        start = start.nextVertex;
    }
    else /* Vertex to be deleted is in between or at last */
    {
      q = start;
        while(q.nextVertex != null)
        {
            if(q.nextVertex.info == u)
                break;
            q = q.nextVertex;
        }
        if(q.nextVertex==null)
        {
            System.out.print("Vertex not found\n");
            return;
        }
        else
        {
            tmp = q.nextVertex;
            q.nextVertex = tmp.nextVertex;
        }
    }
    /*Before freeing the node tmp, free all edges going from this vertex */
    p = tmp.firstEdge;
    while(p!=null)
    {
        temporary = p;
        p = p.nextEdge;
        temporary=null;
    }
    tmp=null;
}/*End of deleteVertex()*/
```

```
void deleteIncomingEdges(int u)
{
    Vertex ptr;
    Edge q,tmp;

    ptr = start;
    while(ptr!=null)
    {
        if(ptr.firstEdge == null)   /*Edge list for vertex ptr is empty*/
        {
            ptr = ptr.nextVertex;
            continue; /* continue searching in other Edge lists */
        }

        if(ptr.firstEdge.destVertex.info == u)
        {
            tmp = ptr.firstEdge;
            ptr.firstEdge = ptr.firstEdge.nextEdge;
            tmp=null;
            continue; /* continue searching in other Edge lists */
        }
        q = ptr.firstEdge;
        while(q.nextEdge!= null)
        {
            if(q.nextEdge.destVertex.info == u)
            {
                tmp = q.nextEdge;
                q.nextEdge = tmp.nextEdge;
                tmp=null;
                continue;
            }
            q = q.nextEdge;
        }
        ptr = ptr.nextVertex;
    }/*End of while*/
```

```
}/*End of deleteIncomingEdges()*/


Vertex findVertex(int u)
{
    Vertex ptr,loc;
    ptr = start;
    while(ptr!=null)
    {
        if(ptr.info == u )
        {
            loc = ptr;
            return loc;
        }
        else
            ptr = ptr.nextVertex;
    }
    loc = null;
    return loc;
}/*End of findVertex()*/

void insertEdge(int u,int v)
{
    Vertex locu,locv;
    Edge ptr,tmp;

    locu = findVertex(u);
    locv = findVertex(v);

    if(locu == null )
    {
        System.out.print("Start vertex not present, first insert vertex \n");
        return;
    }
    if(locv == null )
```

```
    {
        System.out.print("End vertex not present, first insert vertex %d\n");
        return;
    }
    tmp = new 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;

}/*End of insertEdge()*/

void deleteEdge(int u,int v)
{
    Vertex locu;
    Edge tmp,q;

    locu = findVertex(u);

    if(locu == null )
    {
        System.out.print("Start vertex not present\n");
        return;
    }
    if(locu.firstEdge == null)
    {
        System.out.print("Edge not present\n");
        return;
```

```
    }

    if(locu.firstEdge.destVertex.info == v)
    {
        tmp = locu.firstEdge;
        locu.firstEdge = locu.firstEdge.nextEdge;
        tmp=null;
        return;
    }
    q = locu.firstEdge;
    while(q.nextEdge != null)
    {
        if(q.nextEdge.destVertex.info == v)
        {
            tmp = q.nextEdge;
            q.nextEdge = tmp.nextEdge;
            tmp = null;
            return;
        }
        q = q.nextEdge;
    }/*End of while*/

    System.out.print("This Edge not present in the graph\n");
}/*End of deleteEdge()*/

void display()
{
    Vertex ptr;
    Edge q;

    ptr = start;
    while(ptr!=null)
    {
        System.out.print("     "+ptr.info);
        q = ptr.firstEdge;
        while(q!=null)
```

```
            {
                System.out.print("  "+q.destVertex.info);
                q = q.nextEdge;
            }
            System.out.print("\n");
            ptr = ptr.nextVertex;
         }
    }/*End of display()*/

}

public class GraphDemoLinkedList {

    public static void main(String[] args) {
        int choice,u,origin,destin;
        GraphLL glst = new GraphLL();
        Scanner sc=new Scanner(System.in);
        do
        {
            System.out.print("1.Insert a Vertex\n");
            System.out.print("2.Insert an Edge\n");
            System.out.print("3.Delete a Vertex\n");
            System.out.print("4.Delete an Edge\n");
            System.out.print("5.Display\n");
            System.out.print("6.Exit\n");
            System.out.print("Enter your choice : ");
            choice = sc.nextInt();

            switch(choice)
            {
             case 1:
                System.out.print("Enter a vertex to be inserted : ");
                u= sc.nextInt();
                glst.insertVertex(u);
                break;
             case 2:
```

```
            System.out.print("Enter an Edge to be inserted : ");
            origin=sc.nextInt();
            destin = sc.nextInt();
            glst.insertEdge(origin,destin);
            break;
         case 3:
            System.out.print("Enter a vertex to be deleted : ");
            u=sc.nextInt();
            /*This function deletes all edges coming to this vertex*/
            glst.deleteIncomingEdges(u);
            /*This function deletes the vertex from the vertex list*/
            glst.deleteVertex(u);
            break;
         case 4:
            System.out.print("Enter an edge to be deleted : ");
            origin = sc.nextInt();
            destin = sc.nextInt();
            glst.deleteEdge(origin,destin);
            break;
         case 5:
            glst.display();

            break;
         case 6:
            break;
         default:
            System.out.print("Wrong choice\n");
            break;
         }/*End of switch*/
      }while(choice!=6);/*End of while*/

   }
}
```

****

DFS _ Traversal

```
package for_DSA;

import java.util.Scanner;

class myGraphforDFS{
    int adj[][];
    int n;
    int state[];
    public myGraphforDFS(int d)
    {
      n = d;
        adj = new int[n][n];
        state = new int[n];
        for(int i=0;i<n;i++)
            state[i]=0;
    }

    public void create_graph()
    {
        int i,max_edges,origin,destin;

        Scanner sc=new Scanner(System.in);
        max_edges = n*(n-1); /*directed graph*/

        for(i=1; i<=max_edges; i++)
        {
            System.out.println("Enter edge (-1 -1) to quit : ");
            origin = sc.nextInt();
            destin = sc.nextInt();
            if((origin==-1) && (destin==-1))
                break;
            if(origin>=n || destin>=n || origin<0 || destin<0)
            {
                System.out.println("Invalid edge!\n");
                i--;
```

```
        }
        else
            adj[origin][destin] = 1;
    }/*End of for*/
    //sc.close();
}/*End of create_graph()*/

public void insert_edge(int origin,int destin)
{
    if(origin<0 || origin>=n)
    {
        System.out.print("Origin vertex does not exist\n");
        return;
    }
    if(destin<0 || destin>=n)
    {
        System.out.print("Destination vertex does not exist\n");
        return;
    }
    adj[origin][destin] = 1;
}/*End of insert_edge()*/

public void del_edge(int origin, int destin)
{
    if(origin<0 || origin>=n || destin<0 || destin>=n || adj[origin][destin]==0)
    {
        System.out.print("This edge does not exist\n");
        return;
    }
    adj[origin][destin] = 0;
}/*End of del_edge()*/


public void display()
{
    int i,j;
```

```
        for(i=0; i<n; i++)
        {
            for(j=0; j<n; j++)
                System.out.print("   "+adj[i][j]);
            System.out.print("\n");
        }
    }/*End of display()*/


    public void DF_Traversal(int v)
    {
        intStack st = new intStack(50);
        int i;
        st.push(v);
        while(!st.isEmpty())
        {
            v = st.pop();
            if(state[v]==0)
            {
                System.out.print("  "+v);
                state[v]=1;
            }
            for(i=n-1; i>=0; i--)
            {
                if(adj[v][i]==1 && state[i]==0)
                    st.push(i);
            }
        }
    }

}
public class GraphDFSTraversal_Main {

    public static void main(String[] args) {
        myGraphforDFS g=new myGraphforDFS(12);
        //g.create_graph();
```

```
        g.insert_edge(0, 1);
        g.insert_edge(0, 3);
        g.insert_edge(1, 2);
        g.insert_edge(1, 5);
        g.insert_edge(1, 4);
        g.insert_edge(2, 3);
        g.insert_edge(2, 5);
        g.insert_edge(3, 6);
        g.insert_edge(4, 7);
        g.insert_edge(5, 7);
        g.insert_edge(5, 6);
        g.insert_edge(5, 8);
        g.insert_edge(6, 9);
        g.insert_edge(8, 9);
        g.insert_edge(7, 8);

      g.DF_Traversal(5);

    }

}
```

****

```
/*
 * BFS traversal
 */
package for_DSA;


import java.util.Scanner;

class myGraphforBFS{
    int adj[][];
```

```
int n;

public myGraphforBFS(int i)
{
    n = i;

    adj = new int[n][n];
}
public void create_graph()
{
    int i,max_edges,origin,destin;

    Scanner sc=new Scanner(System.in);
    max_edges = n*(n-1); /*directed graph*/

    for(i=1; i<=max_edges; i++)
    {
        System.out.println("Enter edge (-1 -1) to quit : ");
        origin = sc.nextInt();
        destin = sc.nextInt();
        if((origin==-1) && (destin==-1))
            break;
        if(origin>=n || destin>=n || origin<0 || destin<0)
        {
            System.out.println("Invalid edge!\n");
            i--;
        }
        else
            adj[origin][destin] = 1;
    }/*End of for*/
    //sc.close();
}/*End of create_graph()*/

public void insert_edge(int origin,int destin)
{
    if(origin<0 || origin>=n)
```

```
    {
        System.out.print("Origin vertex does not exist\n");
        return;
    }
    if(destin<0 || destin>=n)
    {
        System.out.print("Destination vertex does not exist\n");
        return;
    }
    adj[origin][destin] = 1;
}/*End of insert_edge()*/

public void del_edge(int origin, int destin)
{
    if(origin<0 || origin>=n || destin<0 || destin>=n || adj[origin][destin]==0)
    {
        System.out.print("This edge does not exist\n");
        return;
    }
    adj[origin][destin] = 0;
}/*End of del_edge()*/


public void display()
{
    int i,j;
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
            System.out.print("   "+adj[i][j]);
        System.out.print("\n");
    }
}/*End of display()*/

public void BF_Traversal(int v)
{
```

```
        Scanner sc=new Scanner(System.in);
        int i;
        int state[];
        state=new int[n];
        for(i=0; i<n; i++)
            state[i] = 1;  //

        intQueue q=new intQueue(20);
        q.insert(v);
        state[v] = 2;

        while(!q.isEmpty())
        {
            v = q.del( );
            System.out.print("  "+v);
            state[v] = 3;

            for(i=0; i<n; i++)
            {
                /*Check for adjacent unvisited vertices */
                if(adj[v][i] == 1 && state[i] == 1)
                {
                    q.insert(i);
                    state[i] = 2;
                }
            }
        }
        System.out.print("\n");
        sc.close();
    }/*End of BF_Traverse()*/
}
public class GraphTraversal {

    public static void main(String[] args) {
        //int choice;
        myGraphforBFS g=new myGraphforBFS(9);
```

```
        g.create_graph();
       g.BF_Traversal(1);
    }

}
```

****

Dijkstra's Algorithm :

```
package for_DSA;

import java.util.Scanner;

class Graph_for_Dijkstra{
    int adj[][];
    int n;
    int status[];     // 0 for Temp   1  for Perm
    int predecessor[];  /*predecessor of each vertex in shortest path*/
    int pathLength[];
    public Graph_for_Dijkstra(int d)
    {
       n = d;
        adj = new int[n][n];
        status = new int[n];
        predecessor = new int[n];
        pathLength = new int[n];
        for(int i=0;i<n;i++)
            status[i]=0;
    }
    public void insert_edge(int origin,int destin)
    {
        if(origin<0 || origin>=n)
        {
            System.out.print("Origin vertex does not exist\n");
            return;
```

```
    }
    if(destin<0 || destin>=n)
    {
        System.out.print("Destination vertex does not exist\n");
        return;
    }
    adj[origin][destin] = 1;
}/*End of insert_edge()*/
public void Dijkstra(int s)
{
    int i, current;
    /* Make all vertices temporary */
    for(i=0; i<n; i++)
    {
        predecessor[i] =  -1;
        pathLength[i] = 9999;
        status[i] = 0;
    }
    /* Make pathLength of source vertex equal to 0 */
    pathLength[s] = 0;
    while(true)
    {
        /*Search for temporary vertex with minimum pathLength
        and make it current vertex*/
        current = min_temp( );

        if( current == -1 )
            return;

        status[current] = 1;

        for(i=0; i<n; i++)
        {
            /*Checks for adjacent temporary vertices */
            if ( adj[current][i] !=0 && status[i] == 0 )
                if( pathLength[current] + adj[current][i] < pathLength[i] )
```

```
                {
                    predecessor[i] = current;  /*Relabel*/
                    pathLength[i] = pathLength[current] + adj[current][i];
                }
        }
    }
}

    /*Returns the temporary vertex with minimum value of pathLength
      Returns NIL if no temporary vertex left or
      all temporary vertices left have pathLength infinity*/
public int min_temp()
{
        int i;
        int min = 9999;
        int k = -1;
        for(i=0;i<n;i++)
        {
            if(status[i] == 0 && pathLength[i] < min)
            {
                min = pathLength[i];
                k = i;
            }
        }
        return k;
}/*End of min_temp( )*/
void findPath(int s, int v )
{
    int i,u;
    int path[]=new int[50];           /*stores the shortest path*/
    int shortdist = 0;    /*length of shortest path*/
    int count = 0;        /*number of vertices in the shortest path*/

    /*Store the full path in the array path*/
    while( v != s )
    {
```

```
            count++;
            path[count] = v;
            u = predecessor[v];
            shortdist += adj[u][v];
            v = u;
        }
        count++;
        path[count]=s;

        System.out.println("Shortest Path is : ");
        for(i=count; i>=1; i--)
            System.out.print("  "+path[i]);
        System.out.print("\n Shortest distance is : "+ shortdist);
    }/*End of findPath()*/


    }


public class Dijkstra_Algorithm_Main {

    public static void main(String[] args) {
        Graph_for_Dijkstra gd = new Graph_for_Dijkstra(8);
        int pathLength[]=new int[50];
        int s,v;
        gd.adj[0][1] = 8;
        gd.adj[0][2] = 2;
        gd.adj[0][3] = 7;
        gd.adj[1][5] = 16;
        gd.adj[2][0] = 5;
        gd.adj[2][3] = 4;
        gd.adj[2][6] = 3;
        gd.adj[3][4] = 9;
        gd.adj[4][0] = 4;
        gd.adj[4][5] = 5;
        gd.adj[4][7] = 8;
```

```
        gd.adj[6][2] = 6;
        gd.adj[6][3] = 3;
        gd.adj[6][4] = 4;
        gd.adj[7][5] = 2;
        gd.adj[7][6] = 5;
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter source vertex(-1 to quit): ");
        s = sc.nextInt();

        gd.Dijkstra(s);

        while(true)
        {
            System.out.print("Enter destination vertex(-1 to quit): ");
            v = sc.nextInt();
            if(v == -1)
                break;
            if(v < 0 || v >= gd.n )
                System.out.print("This vertex does not exist\n");
            else if(v == s)
                System.out.print("Source and destination vertices are same\n");
            else if( pathLength[v] == 9999 )
                System.out.print("There is no path from source to destination vertex\n");
            else
                gd.findPath(s,v);
        }

    }

}
```