```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Random;
import java.util.Scanner;
//Nhan Vo
//CECS 328-Lab#8
public class main {
   public static void main(String[] args){
        Scanner scan=new Scanner(System.in);
        Graph graphG1=new Graph();
        Graph graphG2=new Graph();

        System.out.println("Graph 1: ");
        Vertex a1=new Vertex("a");
        Vertex b1=new Vertex("b");
        Vertex c1=new Vertex("c");
        Vertex d1=new Vertex("d");
        Vertex e1=new Vertex("e");
        Vertex f1=new Vertex("f");
        Vertex g1=new Vertex("g");

        graphG1.addVertex(a1);
        graphG1.addVertex(b1);
        graphG1.addVertex(c1);
        graphG1.addVertex(d1);
        graphG1.addVertex(e1);
        graphG1.addVertex(f1);
        graphG1.addVertex(g1);

        graphG1.addEdge(a1,d1);
        graphG1.addEdge(a1,c1);
        graphG1.addEdge(a1,b1);
        graphG1.addEdge(b1,d1);
        graphG1.addEdge(c1,d1);
        graphG1.addEdge(d1,e1);
        graphG1.addEdge(f1,e1);
        graphG1.addEdge(e1,g1);

        graphG1.display();

        System.out.println("\nGraph 2: ");
        Vertex a2=new Vertex("a");
        Vertex b2=new Vertex("b");
        Vertex c2=new Vertex("c");
        Vertex d2=new Vertex("d");
```

```
        Vertex e2=new Vertex("e");
        Vertex f2=new Vertex("f");

        graphG2.addVertex(a2);
        graphG2.addVertex(b2);
        graphG2.addVertex(c2);
        graphG2.addVertex(d2);
        graphG2.addVertex(e2);
        graphG2.addVertex(f2);

        graphG2.addEdge(a2,b2);
        graphG2.addEdge(a2,c2);
        graphG2.addEdge(b2,c2);
        graphG2.addEdge(c2,e2);
        graphG2.addEdge(b2,e2);
        graphG2.addEdge(e2,b2);
        graphG2.addEdge(e2,d2);
        graphG2.addEdge(b2,d2);
        graphG2.addEdge(f2,e2);
        graphG2.addEdge(d2,f2);

        graphG2.display();
        Graph graph=null;
        //Testing driver
        /*
        Ask the user to select which of the two graph they want to use
to perform DFS and then ask for the vertex
        */
        while(true){
            System.out.println("Please select the graph the you want
to work with (type 1 for graph 1) and (type 2 for graph 2) or -1 to
exist");
            int graphinput=scan.nextInt();
            if(graphinput==-1){
                break;
            }
            while(graphinput!=1 & graphinput!=2){// if user input
incorrect graph number
                System.out.println("Please select the graph the you
want to work with (type 1 for graph 1) and (type 2 for graph 2)");
                graphinput=scan.nextInt();
            }
            if(graphinput==1){
                graph=graphG1;
            }
```

```
            else if(graphinput==2){
                graph=graphG2;
            }
            scan.nextLine();
            System.out.println("Please enter the starting vertex that
you want to perform Depth First Search on");
            String n=scan.nextLine();
            boolean con=true;
            Vertex v=null;
            outerloop:
            while(con) {// check user input vertex
                for (Vertex i : graph.getVertexSet()) {
                    if (i.getKey().equals(n)) {
                        v=i;
                        break outerloop;
                    }
                }

                System.out.println("Please enter the starting vertex
that you want to perform Depth First Search on");
                n=scan.nextLine();

            }
            graph.DFS(v);// perform DFS
        }




    }
}

class Graph{
    private ArrayList<Vertex> vertexSet=new ArrayList<>();
    private int countEdge=0;
    private int time=0;
    private boolean cycle=false;
    private LinkedList<Vertex> topologicalOrder = new LinkedList<>();

    public Graph(){
        //default constructor will create an empty graph
    }

    //Auto generate random graph constructor
    public Graph(int ver){
```

```
        countEdge=(int) ver*((ver-1)/2);// compute maximum number of
edges;
        String alphabet="abcdefgh";
        Random ran=new Random();
        char ch=alphabet.charAt(ran.nextInt(alphabet.length()));
        for(int i=1;i<ver+1;i++){
            this.addVertex(new Vertex(Character.toString(ch)));
            ch+=i;
        }
        for(int i=0;i<countEdge;i++){
            if(i<vertexSet.size()-1){
                this.addEdge(vertexSet.get(i), vertexSet.get(i + 1));
            }
        }
    }

    public void display(){

        String output="";
        for(int i=0;i<vertexSet.size();i++){
            output+="Vertex "+vertexSet.get(i).getKey()+" connect to:
"+vertexSet.get(i).getAdj()+"\n";
        }
        System.out.println(output);
    }

    //add vertex into a graph
    public void addVertex(Vertex v){

        vertexSet.add(v);

    }

    // addEdge method for undirected graph
    public void addEdge(Vertex source, Vertex sink){
        if(source.isNeighbor(sink) || sink.isNeighbor(source)){
            //do nothing since the edge already existed
        }
        else{
            source.addNeighbor(sink);// modify adding edge for
directed graph
            countEdge++;
        }

    }
```

```
    public int getOrder(){

        return vertexSet.size();
    }

    public ArrayList<Vertex> getVertexSet(){
        return vertexSet;
    }

    public int getSize(){

        return countEdge;
    }
    //DFS main run function
    public void DFS (Vertex v){
        time=0;
        for(Vertex i:v.getAdj()){
            if(i.parent==null){
                i.parent=v;
                DFS_visit(i);
            }
        }
        if(cycle!=true){
            for(int i=0;i<topologicalOrder.size();i++){
                System.out.println("Vertex:
"+topologicalOrder.get(i)+"(Start: "+topologicalOrder.get(i).start+
                        "/ End: "+topologicalOrder.get(i).end+") ");
            }
            System.out.println(" ");

        }
    }

    //DFS visit to visit each vertex inside the adj list (neighbors)
    public void DFS_visit(Vertex u){
        time++;
        u.start=time;
        for(Vertex i: u.getAdj()){
            if(i.start==0){
                i.parent=u;
                DFS_visit(i);
            }
            if(i.start!=0 & i.end==0){
                System.out.println("Backward edge found");
```

```
                System.out.println("Cycle detected, topological sort
is impossible");
                cycle=true;
            }
        }
        time++;
        u.end=time;
        topologicalOrder.add(u);
    }

}

class Vertex {
    private String Key;
    private LinkedList<Vertex> adj = new LinkedList<>();
    public int distance;
    public Object parent = null;
    public String color = null;
    public int start;
    public int end;

    // default constructor that take a string as key for the vertex
    public Vertex(String k) {

        Key = k;
    }

    public int getDegree() {

        return adj.size();
    }

    public boolean isNeighbor(Vertex v) {
        for (int i = 0; i < adj.size(); i++) {
            if (adj.get(i).getKey() == v.getKey()) {
                return true;
            }
        }
        return false;
    }

    public String getKey() {

        return Key;
    }
```

```
    public void addNeighbor(Vertex v) {
        adj.add(v);
    }

    public LinkedList<Vertex> getAdj() {

        return adj;
    }

    public String toString() {

        return Key;
    }
}
```