```
//Graphs

import java.util.*;
class BFS{

      private int V;
      private LinkedList<Integer> adj[];

      public BFS(int v){
            V = v;
            adj = new LinkedList[v];
        for (int i=0; i<v; ++i){
            adj[i] = new LinkedList();
        }
      }

      public void addEdge(int v, int w){
            adj[v].add(w);
      }

      public void breadthFirstSearch(int s){
            boolean visited[] = new boolean[V];
            LinkedList<Integer> queue = new LinkedList<Integer>();
            queue.add(s);
            visited[s] = true;

            while(queue.size() != 0){
                  s = queue.poll();
                  System.out.print(s + " ");

                  Iterator<Integer> itr = adj[s].listIterator();
                  while(itr.hasNext()){
                        int n = itr.next();
                        if(!visited[n]){
                              visited[n] = true;
                              queue.add(n);
                        }
                  }
            }
      }

      public static void main(String args[]){
            BFS bfs = new BFS(6);
            bfs.addEdge(0, 1);
            bfs.addEdge(0, 2);
            bfs.addEdge(1, 0);
            bfs.addEdge(1, 3);
            bfs.addEdge(1, 4);
            bfs.addEdge(2, 0);
            bfs.addEdge(2, 4);
            bfs.addEdge(3, 1);
            bfs.addEdge(3, 4);
            bfs.addEdge(3, 5);
            bfs.addEdge(4, 1);
            bfs.addEdge(4, 2);
            bfs.addEdge(4, 3);
            bfs.addEdge(4, 5);
            bfs.addEdge(5, 3);
            bfs.addEdge(5, 4);

            System.out.println("Following is Breadth First Traversal "+
                           "(starting from vertex 2)");

            bfs.breadthFirstSearch(0);
      }
}
```

```
import java.util.*;

class DFS{
      private int V;
      private LinkedList<Integer> adj[];

      public DFS(int v){
            V = v;
            adj = new LinkedList[v];
        for (int i=0; i<v; ++i){
            adj[i] = new LinkedList();
        }
      }

      public void addEdge(int v, int w){
            adj[v].add(w);
      }

      public void depthFirstSearch(int s){
            boolean visited[] = new boolean[V];

            for(int i=0;i<V;i++){
                  if(!visited[i]){
                        DFSUtil(i,visited);
                  }
            }
      }

      public void DFSUtil(int v, boolean visited[]){

            visited[v] = true;
            System.out.print(v + " ");

            Iterator<Integer> itr = adj[v].listIterator();
            while(itr.hasNext()){
                  int n = itr.next();
                  if(!visited[n]){
                        DFSUtil(n,visited);
                  }
            }
      }
      public static void main(String args[]){
            DFS dfs = new DFS(6);
            dfs.addEdge(0, 1);
        dfs.addEdge(0, 2);
        dfs.addEdge(1, 3);
        dfs.addEdge(2, 1);
        dfs.addEdge(3, 2);
        dfs.addEdge(4, 3);
        dfs.addEdge(4, 5);
        dfs.addEdge(5, 5);

        System.out.println("Following is Depth First Traversal "+
                           "(starting from vertex 0)");

        dfs.depthFirstSearch(0);
      }

}
```

```
import java.util.*;

class Vertex {
    String id;
    String name;

    Vertex(String id, String name){
        this.id = id;
        this.name = name;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Vertex other = (Vertex) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return name;
    }
}

class Edge{
    String id;
    Vertex source;
    Vertex destination;
    int weight;

    Edge(String id, Vertex source, Vertex destination, int weight){
        this.id = id;
        this.source = source;
        this.destination = destination;
        this.weight = weight;
    }

    public String getId() {
        return id;
    }

    public Vertex getSource() {
        return source;
    }

    public Vertex getDestination() {
        return destination;
    }

    public int getWeight() {
        return weight;
    }

    @Override
    public String toString() {
        return source + " " + destination;
    }
}

class Graph {

    List<Vertex> vertices;
    List<Edge> edges;

    public Graph(List<Vertex> vertices, List<Edge> edges) {
        this.vertices = vertices;
        this.edges = edges;
    }

    public List<Vertex> getVertices() {
        return vertices;
    }

    public List<Edge> getEdges() {
        return edges;
    }
}

public class Dijikstra {

    List<Vertex> nodes;
    List<Edge> edges;
    Set<Vertex> unvisited;
    Set<Vertex> visited;
    Map<Vertex,Vertex> parents;
    Map<Vertex,Integer> distance;

    public Dijikstra(Graph graph) {
        this.nodes = new ArrayList<Vertex>(graph.getVertices());
        this.edges = new ArrayList<Edge>(graph.getEdges());
    }


    public void execute(Vertex source){
        unvisited = new HashSet<>();
        visited = new HashSet<>();
        parents = new HashMap<>();
        distance = new HashMap<>();
        distance.put(source,0);
        unvisited.add(source);
        while(unvisited.size()!=0){
            Vertex node = getMinimum(unvisited);
            visited.add(node);
            unvisited.remove(node);
            findMinimalDistances(node);
        }
    }

    private void findMinimalDistances(Vertex node) {
        List<Vertex> adjacentNodes = getNeighbors(node);
        for (Vertex target : adjacentNodes) {
            if (getShortestDistance(target) > getShortestDistance(node)
                    + getDistance(node, target)) {
                distance.put(target, getShortestDistance(node)
                        + getDistance(node, target));
                parents.put(target, node);
                unvisited.add(target);
            }
        }
    }

    private int getDistance(Vertex node, Vertex target) {
        for (Edge edge : edges) {
            if (edge.getSource().equals(node)
                    && edge.getDestination().equals(target)) {
                return edge.getWeight();
            }
        }
        throw new RuntimeException("Should not happen");
    }

    private List<Vertex> getNeighbors(Vertex node) {
        List<Vertex> neighbors = new ArrayList<Vertex>();
        for (Edge edge : edges) {
            if (edge.getSource().equals(node)
                    && !isVisited(edge.getDestination())) {
                neighbors.add(edge.getDestination());
            }
        }
        return neighbors;
    }

    private Vertex getMinimum(Set<Vertex> vertexes) {
        Vertex minimum = null;
        for (Vertex vertex : vertexes) {
            if (minimum == null) {
                minimum = vertex;
            } else {
                if (getShortestDistance(vertex) <
getShortestDistance(minimum)) {
                    minimum = vertex;
                }
            }
        }
        return minimum;
    }

    private boolean isVisited(Vertex vertex) {
        return visited.contains(vertex);
    }

    private int getShortestDistance(Vertex destination) {
        Integer d = distance.get(destination);
        if (d == null) {
            return Integer.MAX_VALUE;
        } else {
            return d;
        }
    }

    public LinkedList<Vertex> getPath(Vertex target) {
        LinkedList<Vertex> path = new LinkedList<Vertex>();
        Vertex step = target;
        // check if a path exists
        if (parents.get(step) == null) {
            return null;
        }
        path.add(step);
        while (parents.get(step) != null) {
            step = parents.get(step);
            path.add(step);
        }
        // Put it into the correct order
        Collections.reverse(path);
        return path;
    }

    public static void main(String[] args) {
        List<Vertex> nodes = new ArrayList<>();
        for (int i = 0; i < 11; i++) {
            Vertex location = new Vertex("Node_" + i, "Node_" + i);
            nodes.add(location);
        }
        List<Edge> edges = new ArrayList<>();
        Edge edge0 = new Edge("Edge_0",nodes.get(0), nodes.get(1), 85);
        Edge edge1 = new Edge("Edge_1",nodes.get(0), nodes.get(2), 217);
        Edge edge2 = new Edge("Edge_2",nodes.get(0), nodes.get(4), 173);
        Edge edge3 = new Edge("Edge_3",nodes.get(2), nodes.get(6), 186);
        Edge edge4 = new Edge("Edge_4",nodes.get(2), nodes.get(7), 103);
        Edge edge5 = new Edge("Edge_5",nodes.get(3), nodes.get(7), 183);
        Edge edge6 = new Edge("Edge_6",nodes.get(5), nodes.get(8), 250);
        Edge edge7 = new Edge("Edge_7",nodes.get(8), nodes.get(9), 84);
        Edge edge8 = new Edge("Edge_8",nodes.get(7), nodes.get(9), 167);
        Edge edge9 = new Edge("Edge_9",nodes.get(4), nodes.get(9), 502);
        Edge edge10 = new Edge("Edge_10",nodes.get(9), nodes.get(10), 40);
        Edge edge11 = new Edge("Edge_11",nodes.get(1), nodes.get(10), 600);
        edges.add(edge0);
        edges.add(edge1);
        edges.add(edge2);
        edges.add(edge3);
        edges.add(edge4);
        edges.add(edge5);
        edges.add(edge6);
        edges.add(edge7);
        edges.add(edge8);
        edges.add(edge9);
        edges.add(edge10);
        edges.add(edge11);

        Graph graph = new Graph(nodes, edges);
        Dijikstra dijkstra = new Dijikstra(graph);
        dijkstra.execute(nodes.get(0));
        LinkedList<Vertex> path = dijkstra.getPath(nodes.get(10));

        for (Vertex vertex : path) {
            System.out.println(vertex);
        }

    }
```

```
}


import java.util.*;

class TopologicalSort{

      ytt4

      public void TS(){

            boolean visited[] = new boolean[V];
            Stack stack = new Stack();

            for(int i=0; i<V; i++){
                  visited[i] = false;
            }

            for(int i=0; i<V; i++){
                  if(!visited[i]){
                        TSUtil(i,visited,stack);
                  }
            }

            while(!stack.empty()){
                  System.out.print(stack.pop() + " ");
            }
      }

      public void TSUtil(int v, boolean visited[], Stack stack){

            visited[v] = true;
            Integer i;

            Iterator<Integer> itr = adj[v].listIterator();
            while(itr.hasNext()){
                  i = itr.next();
                  if(!visited[i]){
                        TSUtil(i,visited,stack);
                  }
            }

            stack.push(new Integer(v));
      }

      public static void main(String args[]){
            TopologicalSort  ts = new TopologicalSort(10);
            ts.addEdge(0, 1);
        ts.addEdge(0, 5);
        ts.addEdge(1, 7);
        ts.addEdge(3, 2);
        ts.addEdge(3, 4);
        ts.addEdge(3, 7);
        ts.addEdge(3, 8);
        ts.addEdge(4, 8);
        ts.addEdge(6, 0);
        ts.addEdge(6, 1);
        ts.addEdge(6, 2);
        ts.addEdge(8, 2);
        ts.addEdge(9, 4);

        System.out.println("Following is TopologicalSort Traversal "+
                           "(starting from vertex 0)");

        ts.TS();
      }
}
```

```
import java.util.*;

class TopologicalSort_KahnAlgo{

      private int V;
      private LinkedList<Integer> adj[];

      public TopologicalSort_KahnAlgo(int v){
            V = v;
            adj = new LinkedList[v];
        for (int i=0; i<v; ++i){
            adj[i] = new LinkedList();
        }
      }

      public void addEdge(int v, int w){
            adj[v].add(w);
      }

      public void TS_KahnAlgo(){
            int inDegree[] = new int[V];
            int visited = 0;

            for(int i=0; i<V; i++){
                  LinkedList<Integer> temp = adj[i];
                  for(int j=0; j<temp.size(); j++){
                        inDegree[temp.get(j)]++;
                  }
            }
            Queue<Integer> queue = new LinkedList<Integer>();
            for(int i=0;i<V;i++){
                  if(inDegree[i] == 0){
                        queue.add(i);
                  }
            }
            Vector<Integer> tsSort = new Vector<Integer>();

            while(queue.size() != 0){
                  int n = queue.poll();
                  tsSort.add(n);
                  Iterator<Integer> itr = adj[n].listIterator();
                  while(itr.hasNext()){
                        int m = itr.next();
                        --inDegree[m];
                        if(inDegree[m] == 0){
                              queue.add(m);
                        }
                  }
                  visited++;
            }

            if(visited != V)
                  System.out.println("Cycle present. No TopologicalSort 
possible.....");
            else{
                  for(int i=0;i<tsSort.size();i++){
                        System.out.print(tsSort.get(i) + " ");
                  }
            }
      }

      public static void main(String args[]){
            TopologicalSort_KahnAlgo ts = new TopologicalSort_KahnAlgo(10);
            ts.addEdge(0, 1);
        ts.addEdge(0, 5);
        ts.addEdge(1, 7);
        ts.addEdge(3, 2);
        ts.addEdge(3, 4);
        ts.addEdge(3, 7);
        ts.addEdge(3, 8);
        ts.addEdge(4, 8);
        ts.addEdge(6, 0);
        ts.addEdge(6, 1);
        ts.addEdge(6, 2);
        ts.addEdge(8, 2);
        ts.addEdge(9, 4);

        System.out.println("Following is TopologicalSort Traversal "+
                           "(starting from vertex 0)");

        ts.TS_KahnAlgo();
      }
}


import java.util.*;

class CyclicInDirected{

    private int V;
    private LinkedList<Integer> adj[];

    public CyclicInDirected(int v){
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i){
            adj[i] = new LinkedList();
        }
    }

    public void addEdge(int v, int w){
        adj[v].add(w);
    }

    public boolean isCyclic(){
        boolean visited[] = new boolean[V];
        boolean restack[] = new boolean[V];
        for(int i=0;i<V;i++){
            visited[i] = false;
            restack[i] = false;
        }

        for(int i=0;i<V;i++){
            if(isCyclicUtil(i,visited,restack)){
                return true;
            }
        }
        return false;
    }

    public boolean isCyclicUtil(int v, boolean visited[], boolean restack[]){

        if(restack[v])
            return true;
        if(visited[v])
            return false;

        visited[v] = true;
        restack[v] = true;
        Integer c;

        Iterator<Integer> itr = adj[v].listIterator();
        while(itr.hasNext()){
            c = itr.next();
            if(isCyclicUtil(c,visited,restack)){
                return true;
            }
        }
        restack[v] = false;
        return false;
    }

    public static void main(String[] args)
    {
        CyclicInDirected graph = new CyclicInDirected(2);
        graph.addEdge(0, 1);
        //graph.addEdge(0, 2);
        //graph.addEdge(1, 2);
        //graph.addEdge(2, 0);
        //graph.addEdge(2, 3);
        //graph.addEdge(3, 3);
        if(graph.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't " + "contain cycle");
    }
}


import java.util.*;

class CyclicInUnDirected{

      private int V;
      private LinkedList<Integer> adj[];

      public CyclicInUnDirected(int v){
            V = v;
            adj = new LinkedList[v];
        for (int i=0; i<v; ++i){
            adj[i] = new LinkedList();
        }
      }

      public void addEdge(int v, int w){
            adj[v].add(w);
            adj[w].add(v);
      }

      public boolean isCyclic(){
            boolean visited[] = new boolean[V];
            for(int i=0;i<V;i++){
                  visited[i] = false;
            }

            for(int i=0;i<V;i++){
                  if(!visited[i]){
                        if(isCyclicUtil(i,visited,-1)){
                              return true;
                        }
                  }
            }
            return false;
      }

      public boolean isCyclicUtil(int v, boolean visited[], int parent){
            visited[v] = true;
            Integer i;

            Iterator<Integer> itr = adj[v].listIterator();
            while(itr.hasNext()){
                  i = itr.next();
                  if(!visited[i]){
                        if(isCyclicUtil(i,visited,v)){
                              return true;
                        }
                  }
                  else if(i!=parent){
                        return true;
                  }
            }
            return false;
      }

      public static void main(String args[])
    {
        // Create a graph given in the above diagram
        CyclicInUnDirected g1 = new CyclicInUnDirected(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 0);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        if (g1.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contains cycle");

        CyclicInUnDirected g2 = new CyclicInUnDirected(3);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        if (g2.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contains cycle");
    }
}
```