```
import java.util.ArrayList;
import java.util.Set;

/**
 * Project Dijkstra Algorithm
 * This class is used to define the halforder
 *
 * @author Hakan Tanis
 * @author Kevin Adamczewski
 * @author Jonas Litmeyer
 * Date 30.05.2018
 * @version 3.0
 *
 * Last Change:
 * by: Kevin Adamczewski
 * date: 04.06.2018
 */

public class HalfOrder
{
    /**
     * initialisation of methods
     * <img src="halforder.jpg" alt="illustration" >
     */

        public ArrayList<Node> nodes;
        public ArrayList<Edge> edges;

        public Object nodelist;

        /**
     * initialisation of methods
     */
    public void init()
    {

        ArrayList<Node> nodelist = createNodelistExampleOne();
        //      printNodes(nodelist);

        System.out.println();

        ArrayList<Edge> edgelist = createEdgelistExampleOne(nodelist);
        //      printEdges(edgelist);

        ArrayList<ArrayList<Node>> listOfLists =
                computeHalfOrder(getFirstNode(nodelist), nodelist, edgelist);

    }

    /**
     * @param nodelist get the first node
     * @return first node
```

```
 */
public Node getFirstNode(ArrayList<Node> nodelist)
{
    return nodelist.get(0);

}

/**
 * @param edgelist prints edges of given edgelist
 * @param description of edge set to be printed as prefix
 */
private void printEdges(ArrayList<Edge> edgelist, String descr)
{
    System.out.println("Edges: " + descr);

    for (int i = 0; i < edgelist.size(); i++)
    {
        System.out.println(edgelist.get(i));
    }

    System.out.println("--------");
}

public Edge returnEdge (int i)
{
    if ( i < 0 || i >= edges.size())
    {
            return null;
    }
    return edges.get(i);
}

public Node returnNode (int i)
{
    if ( i < 0 || i >= nodes.size())
    {
            return null;
    }
    return nodes.get(i);
}

/**
 * @param nodelist create and add list of edges (example one)
 * @return edgelist with sources, destinations and weights
 */
private ArrayList<Edge> createEdgelistExampleOne(ArrayList<Node> nodelist)
{
    ArrayList<Edge> edgelist = new ArrayList<Edge>();

    edges = edgelist;

    edgelist.add(new Edge(8.0));
```

```
    edgelist.add(new Edge(10.0));
    edgelist.add(new Edge(5.0));
    edgelist.add(new Edge(3.0));
    edgelist.add(new Edge(2.2));
    edgelist.add(new Edge(9.0));
    edgelist.add(new Edge(20.5));

    // setting destination and source
    edgelist.get(0).setDestination(nodelist.get(1));
    edgelist.get(0).setSource(nodelist.get(0));

    edgelist.get(1).setDestination(nodelist.get(3));
    edgelist.get(1).setSource(nodelist.get(1));

    edgelist.get(2).setDestination(nodelist.get(4));
    edgelist.get(2).setSource(nodelist.get(2));

    edgelist.get(3).setDestination(nodelist.get(5));
    edgelist.get(3).setSource(nodelist.get(3));

    edgelist.get(4).setDestination(nodelist.get(5));
    edgelist.get(4).setSource(nodelist.get(4));

    edgelist.get(5).setDestination(nodelist.get(2));
    edgelist.get(5).setSource(nodelist.get(0));

    edgelist.get(6).setDestination(nodelist.get(5));
    edgelist.get(6).setSource(nodelist.get(0));

    return edgelist;
}

/**
 * @param nodelist prints nodes
 * @param description of node set to be printed as prefix
 */
private void printNodes(ArrayList<Node> nodelist, String descr)
{
    System.out.println("Nodelist : " + descr);
    System.out.println();

    for (int i = 0; i < nodelist.size(); i++)
    {
        System.out.println(nodelist.get(i));
    }

    System.out.println("--------");
}

/**
 * create list of nodes (example one)
 * @return list of named nodes (example one)
```

```
     */
    private ArrayList<Node> createNodelistExampleOne()
    {
        ArrayList<Node> nodelist = new ArrayList<Node>();

        nodes = nodelist;

        nodelist.add(new Node("Knoten 1 ",1,1));
        nodelist.add(new Node("Knoten 2 ",2,2));
        nodelist.add(new Node("Knoten 3 ",3,3));
        nodelist.add(new Node("Knoten 4 ",4,4));
        nodelist.add(new Node("Knoten 5 ",5,5));
        nodelist.add(new Node("Knoten 6 ",6,6));

        return nodelist;
    }

    /**
     * checks if double edges exists
     * @param edgelist checks if double edges exists
     * @param destination checks if edge e and e2 are the same
     * @return edge if edge e is not the same as edge e2
     * <img src="checkIfDoubleEdges.jpg" alt="illustration" >
     */
    public boolean checkIfDoubleEdges(ArrayList<Edge> edgelist, Node
destination)
    {
        for (int i = 0; i < edgelist.size(); i++)
        {
            for (int j = 1; j < edgelist.size(); j++)
            {
                Edge e = edgelist.get(i);
                Edge e2 = edgelist.get(j);

                if( e == null || e2 == null )
                {
                        return false;
                }

                if (e != e2 && e.getDestination() == e2.getDestination() &&
                        e.getSource() == e2.getSource())
                {
                    return true;
                }

            }

        }
        return false;
    }

    /**
```

```
     * @param source computes next node
     * @param edgelist checks source and destination
     * @return of next node
     */
    public ArrayList<Node> computeNextNodes(Node source, ArrayList<Edge>
edgelist)
    {
        ArrayList<Node> nextNodes = new ArrayList<Node>();

        for (int i = 0; i < edgelist.size(); i++)
        {
            Edge e = edgelist.get(i);

            if (e.getSource() == source)
            {
                nextNodes.add(e.getDestination());
            }

        }

        return nextNodes;
    }

    /**
     * @param nodeToAdd add node if not redundant
     * @param nodelist checks if node is included
     * @return node
     * <img src="addNodeIfNotRedundant.jpg" alt="illustration" >
     */
    public ArrayList<Node> addNodeIfNotRedundant(Node nodeToAdd, ArrayList<Node>
nodelist)
    {
        for (int i = 0; i < nodelist.size(); i++)
        {
            if (nodelist.get(i) == nodeToAdd)
            {
                return nodelist;
            }
        }

        nodelist.add(nodeToAdd);
        return nodelist;
    }

    /**
     * compute HalfOrder
     * @param firstNode get first node to lock first node
     * @param nodelist to calculate the next nodelists in the halforder
     * @param edgelist to put edges between the placement of nodes in the
halforder
     * @return put nodes into the list of lists of the amount of nodes that are
given
```

```
     */
    public ArrayList<ArrayList<Node>> computeHalfOrder(Node firstNode,
ArrayList<Node> nodelist, ArrayList<Edge> edgelist)
    {
        ArrayList<ArrayList<Node>> listOfLists = new
ArrayList<ArrayList<Node>>();
        ArrayList<Node> allNodesProcessed = new ArrayList<Node>();

        ArrayList<Node> stepTarget = computeNextNodes(firstNode, edgelist);
        listOfLists.add(stepTarget);
        allNodesProcessed.addAll(stepTarget);

        int i = 1;

        printNodes(stepTarget, "" + i);
        printNodes(allNodesProcessed, i + " processed");

        while (stepTarget.size() > 0)
        {
            ArrayList<Node> stepSource = stepTarget;
            stepTarget = new ArrayList<Node>();

            for (int j = 0; j < stepSource.size(); j++)
            {
                ArrayList<Node> partStep = computeNextNodes(stepSource.get(j),
edgelist);
                magicalAdd(partStep, stepTarget, allNodesProcessed);
            }

            if (stepTarget.size() > 0)
            {
                listOfLists.add(stepTarget);

                printNodes(stepTarget, "" + ++i);
                printNodes(allNodesProcessed, i + " processed");
            }

        }

        return listOfLists;
    }

    /**
     * @param source of node
     * @param destination of node to node
     * @param alreadyProcessed add already processed node to already processed
list
     * <img src="halforder.jpg" alt="illustration" >
     */
    public void magicalAdd(ArrayList<Node> source, ArrayList<Node> destination,
ArrayList<Node> alreadyProcessed)
    {
```

```
        for (Node node : source)
        {
            if (!alreadyProcessed.contains(node))
            {
                destination.add(node);
                alreadyProcessed.add(node);
            }
        }
    }

}
```