Problem 1

```
/**
 * A Node is an immutable representation of a node by itself. A node has a label
 * and can be used in combination with other nodes to create edges and graphs.
 */
public class Node {
    private final String label;

    // Abstraction function:
    // Node represents a node with the label label

    // Representation invariant:
    // label != null

    /**
     * Constructs a node with an empty string label
     * @requires none
     * @modifies none
     * @effects none
     * @throws none
     * @returns Node with an empty string label
     */
    public Node() {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Constructs a node with the given label
     * @param label Label to be given to the node
     * @requires none
     * @modifies none
     * @effects none
     * @throws NullPointerException if label == null
     * @returns Node with given label
     */
    public Node(String label) {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Constructs a copy of a node
     * @param n Node to be copied
     * @requires none
     * @modifies none
     * @effects none
     * @throws NullPointerException if n == null
     * @returns Node with the same data as n
     */
    public Node(Node n) {
        throw new RuntimeException("Not implemented");
    }
    /**
     * Get node’s label
     * @requires none
     * @modifies none
     * @effects none
     * @throws none
     * @returns Node’s label
     */
    public String label() {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Check if nodes are equal
     * @requires none
     * @modifies none
     * @effects none
     * @throws none
     * @returns True iff obj is a Node and obj.label == this.label
     */
    public boolean equals(Object obj) {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Standard hashCode function
     * @requires none
     * @modifies none
     * @effects none
     * @throws none
     * @returns An int that all objects equal to this will also return
     */
    public int hashCode() {
        throw new RuntimeException("Not implemented");
     }
}
```

```
/**
 * An Edge is an immutable representation of an edge between two nodes. An edge has a direction
 * defined by the node on the outgoing side and the node on the incoming side and a label.
 * For example, in an edge <A,B>, A is the outgoing node and B is the incoming node.
 */
public class Edge {
    private final Node outgoingNode;
    private final Node incomingNode;
    private final String label;

    // Abstraction function:
    // Edge represents an edge from outgoingNode to incomingNode with the label label

    // Representation invariant:
    // outgoingNode != null && incomingNode != null && label != null

    /**
     * Constructs an edge from outNode to inNode with an empty string label
     * @param outNode Node that will have this edge be an outgoing edge
     * @param inNode Node that will have this edge be an incoming edge
     * @requires none
     * @modifies none
     * @effects none
     * @throws NullPointerException if outNode == null || inNode == null
     * @returns An edge from outNode to inNode with an empty string label
     */
    public Edge(Node outNode, Node inNode) {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Constructs an edge from outNode to inNode with the given label
     * @param outNode Node that will have this edge be an outgoing edge
     * @param inNode Node that will have this edge be an incoming edge
     * @param label Edge’s label
     * @requires none
     * @modifies none
     * @effects none
     * @throws NullPointerException if outNode == null || inNode == null || label == null
     * @returns An edge from outNode to inNode with given label
     */
    public Edge(Node outNode, Node inNode, String label) {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Constructs a copy of an edge
     * @param e Edge to be copied
     * @requires none
     * @modifies none
     * @effects none
 * @throws NullPointerException if e == null
 * @returns An edge with the same data as e
 */
public Edge(Edge e) {
    throw new RuntimeException("Not implemented");
}

/**
 * Get edge's outgoing node
 * @requires none
 * @modifies none
 * @effects none
 * @throws none
 * @returns Edge's outgoing node
 */
public Node outgoingNode() {
    throw new RuntimeException("Not implemented");
}

/**
 * Get edge's incoming node
 * @requires none
 * @modifies none
 * @effects none
 * @throws none
 * @returns Edge's incoming node
 */
public Node incomingNode() {
    throw new RuntimeException("Not implemented");
}

/**
 * Get edge's label
 * @requires none
 * @modifies none
 * @effects none
 * @throws none
 * @returns Edge's label
 */
public String label() {
    throw new RuntimeException("Not implemented");
}

/**
 * Reverse an edge (ex: an edge <A,B> when reversed is <B,A> with the same label)
 * @requires none
 * @modifies none
 * @effects none
 * @throws none
 * @returns A reversed edge
 */
```

```
    public Edge reverse() {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Reverse an edge and give it a new label
     * @param newLabel New label to be given to reversed edge
     * @requires none
     * @modifies none
     * @effects none
     * @throws NullPointerException if newLabel == null
     * @returns A reversed edge with the label newLabel
     */
    public Edge reverse(String newLabel) {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Check if edges are equal
     * @requires none
     * @modifies none
     * @effects none
     * @throws none
     * @returns True iff obj is an Edge, obj.outgoingNode == this.outgoingNode,
     *          obj.incomingNode == this.incomingNode, and obj.label == this.label
     */
    public boolean equals(Object obj) {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Standard hashCode function
     * @requires none
     * @modifies none
     * @effects none
     * @throws none
     * @returns An int that all objects equal to this will also return
     */
    public int hashCode() {
        throw new RuntimeException("Not implemented");
     }
}
```

```
/**
 * A Graph is a mutable representation of a directed, labeled multigraph. This graph
 * can include empty graphs (no nodes and edges), graphs without edges, graphs with any
 * number of edges between a pair of nodes, and graphs with reflexive edges. All nodes and
 * edges in the graph are labeled.
 */
public class Graph {
    private Set<Node> nodes;
    private Map<Node, Set<Edge>> outgoingEdges;
    private Map<Node, Set<Edge>> incomingEdges;

    // Abstraction function:
    // A Graph represents a directed, labeled multigraph with the nodes nodes and edges
    // outgoingEdges / incomingEdges (both maps contain the same edges, just stored
    // differently).

    // Representation invariant: (no null nodes/edges, nodes in edges are in graph,
    //                            edges are in the sets they should be in)
    // forall n in nodes: n != null && n in incomingEdges && n in outgoingEdges &&
    // forall e in outgoingEdges[n]: e != null &&
    //                               e.incomingNode() in nodes && e.outgoingNode() in nodes &&
    //                               e in incomingEdges[e.incomingNode()] && n == e.outgoingNode() &&
    // forall e in incomingEdges[n]: e != null &&
    //                               e.incomingNode() in nodes && e.outgoingNode() in nodes &&
    //                               e in outgoingEdges[e.outgoingNode()] && n == e.incomingNode()

    /**
     * Constructs an empty graph (no nodes and no edges)
     * @requires none
     * @modifies none
     * @effects none
     * @throws none
     * @returns A graph with no nodes and no edges
     */
    public Graph() {
        throw new RuntimeException("Not implemented");
    }

    /**
     * Constructs a graph with the given list of nodes and no edges. Only unique nodes
     * will be placed in the graph
     * Ex: nodes = [A,A,B] -> graph only contains [A,B]
     * @param nodes Nodes to be put in graph
     * @requires none
     * @modifies none
     * @effects none
     * @throws NullPointerException if nodes == null || a node is null
     * @returns A graph with unique nodes (if given a duplicate node, it isn’t added)
     */
    public Graph(List<Node> nodes) {
        throw new RuntimeException("Not implemented");
}

/**
 * Constructs a graph with the given list of nodes and edges. Only unique nodes and
 * edges will be placed in the graph
 * Ex: edges = [(<A,B>, label1), (<A,B>, label1), (<A,B>, label2), (<B,C>, label1)]
 *     -> graph only contains [(<A,B>, label1), (<A,B>, label2), (<B,C>, label1)]
 * @param nodes Nodes to be put in graph
 * @param edges Edges to be put in graph
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if nodes == null || edges == null || a node/edge is null
 *         IllegalArgumentException if an edge has an incoming/outgoing node not in nodes
 * @returns A graph with unique nodes and unique edges (if a duplicate node/edge is given,
            it isn’t added)
 */
public Graph(List<Node> nodes, List<Edge> edges) {
    throw new RuntimeException("Not implemented");
}

/**
 * Constructs a copy of a graph
 * @param g Graph to be copied
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if g == null
 * @returns A graph with the same data as g
 */
public Graph(Graph g) {
    throw new RuntimeException("Not implemented");
}

/**
 * Get all nodes in the graph
 * @requires none
 * @modifies none
 * @effects none
 * @throws none
 * @returns List of nodes in the graph
 */
public List<Node> nodes() {
    throw new RuntimeException("Not implemeneted");
}

/**
 * Check if a node is in the graph
 * @param n Node to be checked
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if n == null
 * @returns True if there exists a node in the graph s.t. node.equals(n). False otherwise
 */
public boolean containsNode(Node n) {
    throw new RuntimeException("Not implemented");
}

/**
 * Get all edges in the graph
 * @requires none
 * @modifies none
 * @effects none
 * @throws none
 * @returns List of edges in the graph
 */
public List<Edge> edges() {
    throw new RuntimeException("Not implemeneted");
}

/**
 * Get all edges from one node to another node
 * @param out Node on the outgoing side of the edge
 * @param in Node on the incoming side of the edge
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if out == null || in == null
 * @returns If in or out isn't in the graph, an empty list is returned.
 *          Else, a list of edges <out, in> is returned
 */
public List<Edge> edges(Node out, Node in) {
    throw new RuntimeException("Not implemented");
}

/**
 * Get all outgoing edges of a node
 * @param n Node whose outgoing edges you want
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if n == null
 * @returns If n is not in the graph, an empty list is returned.
 *          Else, a list of outgoing edges of n is returned
 *          (forall outgoingEdges e: e == <n, _>)
 */
public List<Edge> outgoingEdges(Node n) {
    throw new RuntimeException("Not implemeneted");
}

/**
```

```
 * Get all incoming edges of a node
 * @param n Node whose incoming edges you want
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if n == null
 * @returns If n is not in the graph, an empty list is returned.
 *          Else, a list of incoming edges of n is returned
 *          (forall incomingEdges e: e == <_, n>)
 */
public List<Edge> incomingEdges(Node n) {
    throw new RuntimeException("Not implemeneted");
}

/**
 * Get out-degree of a node (number of outgoing edges of a node)
 * @param n Node whose out-degree you want
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if n == null
 * @returns If n is not in the graph, -1 is returned.
 *          Else, out-degree of n is returned.
 *          out-degree >= 0
 */
public int outDegree(Node n) {
    throw new RuntimeException("Not implemented");
}

/**
 * Get in-degree of a node (number of incoming edges of a node)
 * @param n Node whose in-degree you want
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if n == null
 * @returns If n is not in the graph, -1 is returned.
 *          Else, in-degree of n is returned.
 *          in-degree >= 0
 */
public int inDegree(Node n) {
    throw new RuntimeException("Not implemented");
}

/**
 * Check if an edge is in the graph
 * @param e Edge to be checked
 * @requires none
 * @modifies none
 * @effects none
 * @throws NullPointerException if e == null
 * @returns True if there exists an edge in the graph s.t. edge.equals(e). False otherwise
 */
public boolean containsEdge(Edge e) {
    throw new RuntimeException("Not implemented");
}

/**
 * Add a node to the graph
 * @param n Node to be added
 * @requires none
 * @modifies this
 * @effects Adds n to the graph
 * @throws NullPointerException if n == null
 * @returns True if n isn’t in the graph already. False otherwise
 */
public boolean addNode(Node n) {
    throw new RuntimeException("Not implemented");
}

/**
 * Remove a node from the graph
 * @param n Node to be removed
 * @requires none
 * @modifies this
 * @effects Removes n and any connected edges from the graph
 * @throws NullPointerException if n == null
 * @returns True if n was in the graph. False otherwise
 */
public boolean removeNode(Node n) {
    throw new RuntimeException("Not implemented");
}

/**
 * Add an edge to the graph
 * @param e Edge to be added
 * @requires none
 * @modifies this
 * @effects Adds e to the graph
 * @throws NullPointerException if e == null
 * @returns True if e’s incoming node and outgoing node is in the graph and e isn’t in
 *          the graph already. False otherwise
 */
public boolean addEdge(Edge e) {
    throw new RuntimeException("Not implemented");
}

/**
 * Remove an edge from the graph
 * @param e Edge to be removed
 * @requires none
 * @modifies this
```

```
     * @effects Removes e from the graph
     * @throws NullPointerException if e == null
     * @returns True if e was in the graph. False otherwise
     */
    public boolean removeEdge(Edge e) {
        throw new RuntimeException("Not implemented");
    }
}
```