1 Problem 1: Specifications and Stubs

```
package hw4;

import java.util.Map;
import java.util.Iterator;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

/**
 * <b>Graph</b> represents a mutable directed multigraph with labeled edges.
 * <p>
 *
 * Examples of Graphs are: social networks, computer networks,
 * airline flights between cities, etc.
 */

public class Graph {
 // Abstraction Function:
    // Represents a graph where nodes are represented by strings and edges are
    // represented as labeled strings.
 // Each node in the graph is mapped to a list of strings,
    // where each string represents an edge with the format "childNode(edgeLabel)".
 // If there is a reflexive edge, parentNode(edgeLabel)
    // should appear in the list of children.
 // If there are no children, the list will be empty.
 // If there are no nodes, the size of the graph will be 0.
    //
 // Representation Invariant for every Graph adjacencyList:
    // Each key in the adjacencyList (representing a node)
    // must be a non-null, non-empty string.
 // Each value in the adjacencyList (representing edges) must be a list of non-null,
    // non-empty strings in the format "childNode(edgeLabel)".
 // No duplicate nodes are allowed in the adjacencyList.

 // In other words:
 // The adjacencyList must not contain null keys or values.
 // Each node must have a valid list of edges associated with it.
 // No two nodes can have the same name.
```

```
// Each edge representation in the list must follow the specified format:
   // "childNode(edgeLabel)".


private Map<String, List<String>> adjacencyList;


/**
    * @effects Constructs a new Graph with no Nodes or edges.
    */
public Graph() {
throw new RuntimeException("not implemented");
}

/**
 *
    * @param adjacencyList, a map of Nodes to their corresponding edges
       and labels in the graph
    * @requires adjacencyList != null && all nodes != ""
    * @effects Constructs a new Graph with the given adjacencyList
    */
public Graph(Map<String, List<String>> adjacencyList) {
throw new RuntimeException("not implemented");
}



   /**
    *
    * @param nodeData, a string representing the node to be added
    * @requires nodeData != "" && nodeData is not already in the graph
    * @modifies adjacencyList
    * @effects adjacencyList.size = adjacencyList.size + 1
    */
   public void addNode(String nodeData) {
    throw new RuntimeException("not implemented");
   }

/**
 *
```

```
 * @param parentNode, the parent node of the edge (the node the edge is coming from)
 * @param childNode, the child node of the edge (the node the edge is pointing to)
 * @param edgeLabel, the label of the edge being added
 * @requires parentNode != "" && childNode != ""
 *           && parentNode is already in the graph
 * @modifies adjacencyList
 * @effects adds a new child node to the parent node with the given label
     *         if the child node doesn't exist,
 *         otherwise updates the label of the edge to the child node

 */
public void addEdge(String parentNode, String childNode, String edgeLabel) {
throw new RuntimeException("not implemented");
}



/**
 *
 * @param nodeData, a string representing the node to be removed
 * @requires nodeData != "" && nodeData is in the graph
 * @modifies adjacencyList
 * @effects removes the node from the graph
 */
public void removeNode(String nodeData) {
throw new RuntimeException("not implemented");
}

/**
 *
 * @param parentNode, the parent node of the edge (the node the edge is coming from)
 * @param childNode,  the child node of the edge (the node the edge is pointing to)
 * @param edgeLabel,  the label of the edge being removed
 * @requires parentNode != "" && childNode != "" && edgeLabel != "" &&
 *           parentNode is in the graph
 * @modifies adjacencyList
 * @effects removes the edge from the parent node with the child node and label
 */
public void removeEdge(String parentNode, String childNode, String edgeLabel) {
throw new RuntimeException("not implemented");
```

```
}



/**
 * Returns an iterator that represents all the nodes this Graph
 *
 * @requires adjencencyList != null
 * @return Iterator<String> that represents all the nodes in the graph
 */
public Iterator<String> listNodes() {
throw new RuntimeException("not implemented");
}

/**
 * Returns an iterator that represents all the children of the given node in
 * lexicographical order. First by the node name and secondarily by the edge label.
 * If there is a reflexive edge, parentNode(edgeLabel) should appear in the
     * list of children.
 *
 * @param parentNode, the node to get the children of
 * @requires parentNode != "" && parentNode is in the graph
 * @return Iterator<String> that represents all the children of the given node
 */
public Iterator<String> listChildren(String parentNode) {
throw new RuntimeException("not implemented");
}



/**
 * Returns a boolean indicating whether the given node is in the graph
 *
 * @param parentNode, the parent node of the edge (the node the edge is coming from)
 * @requires nodeData != ""
 * @return boolean, true if the node is in the graph, false otherwise
 */
public boolean containsNode(String nodeData) {
throw new RuntimeException("not implemented");
}
```

```
 /**
  * Returns the edge label value if the edge is in the graph, nothing otherwise
  *
  * @param parentNode, the parent node of the edge (the node the edge is coming from)
  * @param childNode,  the child node of the edge (the node the edge is pointing to)
  * @requires parentNode != "" && childNode != ""
  * @return edge label value if the edge is in the graph, nothing otherwise
  */
 public String getEdgeLabel(String parentNode, String childNode) {
 throw new RuntimeException("not implemented");
 }

}
```