# Homework 4: Problem 1 - Graph ADT Specification

## 1 Overview

This document contains the specification for a Graph ADT that represents a directed labeled multigraph. The Graph ADT is designed to store nodes with String data and edges with String labels. The graph allows multiple edges between the same pair of nodes (multigraph property) and reflexive edges (from a node to itself).

## 2 Graph ADT

```java
/**
 * A mutable directed labeled multigraph. A multigraph is a graph which can have multiple
 * edges between the same pair of nodes. Each edge has a label of type String.
 * The graph stores String data for nodes.
 * Nodes are uniquely identified by their data.
 */
public class Graph {

    // Abstraction Function:
    // Represents a directed labeled multigraph where:
    // - Nodes are represented by String data values
    // - Directed edges exist from one node to another, potentially with multiple edges
    //   between the same pair of nodes
    // - Each edge has a String label
    // - Nodes are uniquely identified by their data values (no two nodes have the same data)
    //
    // Representation Invariant:
    // - nodeEdges != null
    // - No key in nodeEdges is null
    // - No value in nodeEdges is null
    // - For every node key in nodeEdges, the corresponding map value is not null
    // - Every node that appears as a destination in an edge
         list is also a key in nodeEdges
    // - For every edge, both source and destination nodes
        exist in the graph
    // - For every edge, the edge label is not null

    /**
     * @spec.effects Constructs a new empty graph
     */
    public Graph();

    /**
     * Adds a node to this graph.
     *
     * @param nodeData the data of the node to add
     * @spec.requires nodeData != null
     * @spec.modifies this
     * @spec.effects If no node with nodeData exists in this
          graph, adds a node with nodeData to this.
     *               If a node with nodeData already exists,
          the graph remains unchanged.
     */
    public void addNode(String nodeData);

    /**
     * Adds an edge from the node with parentData to the
         node with childData.
     *
     * @param parentData the data of the parent node
     * @param childData the data of the child node
     * @param edgeLabel the label of the edge
     * @spec.requires parentData != null && childData !=
         null && edgeLabel != null
     * @spec.requires nodes with parentData and childData
         exist in this graph
     * @spec.modifies this
     * @spec.effects Adds an edge from the node with
         parentData to the node with childData with label
         edgeLabel.
     *               If an identical edge already exists,
         this method may or may not add a duplicate edge.
     */
    public void addEdge(String parentData, String childData,
         String edgeLabel);

    /**
     * Returns whether a node exists in this graph.
     *
     * @param nodeData the data of the node to check
     * @spec.requires nodeData != null
     * @return true if a node with nodeData exists in this graph, false otherwise
     */
    public boolean containsNode(String nodeData);

    /**
     * Returns whether an edge exists from the node with parentData to the node with childData with the given label.
     *
     * @param parentData the data of the parent node
     * @param childData the data of the child node
     * @param edgeLabel the label of the edge
     * @spec.requires parentData != null && childData != null && edgeLabel != null
     * @return true if an edge from the node with parentData to the node with childData with label edgeLabel exists,
     *         false otherwise
     */
    public boolean containsEdge(String parentData, String childData, String edgeLabel);

    /**
     * Returns a set of all nodes in this graph.
     *
     * @return a set containing all nodes in this graph
     * @spec.effects The returned set is a copy; changes to it will not affect this graph.
     */
    public Set<String> getNodes();

    /**
     * Returns a map of child nodes and their corresponding edge labels from a given parent node.
     *
     * @param parentData the data of the parent node
     * @spec.requires parentData != null
     * @spec.requires a node with parentData exists in this graph
     * @return a map where each key is a child node data and each value is a list of edge labels from the parent to that child
     * @spec.effects The returned map is a copy; changes to it will not affect this graph.
     */
    public Map<String, List<String>> getChildrenWithLabels(String parentData);

    /**
```

```java
     * Returns a list of all child nodes of a given parent
       node.
     *
     * @param parentData the data of the parent node
     * @spec.requires parentData != null
     * @spec.requires a node with parentData exists in this
       graph
     * @return a list containing all child nodes of the node
        with parentData
     * @spec.effects The returned list is a copy; changes to
        it will not affect this graph.
     *              If a node has multiple edges to the same
        child, that child will appear multiple times in the
        list.
     */
    public List<String> getChildren(String parentData);

    /**
     * Returns a list of all edge labels from the node with
       parentData to the node with childData.
     *
     * @param parentData the data of the parent node
     * @param childData the data of the child node
     * @spec.requires parentData != null && childData !=
       null
     * @spec.requires nodes with parentData and childData
       exist in this graph
     * @return a list containing all edge labels from the
       node with parentData to the node with childData
     * @spec.effects The returned list is a copy; changes to
        it will not affect this graph.
     */
    public List<String> getEdgeLabels(String parentData,
       String childData);

    /**
     * Returns a list of all parent nodes of a given child
       node.
     *
     * @param childData the data of the child node
     * @spec.requires childData != null
     * @spec.requires a node with childData exists in this
       graph
     * @return a list containing all parent nodes of the
       node with childData
     * @spec.effects The returned list is a copy; changes to
        it will not affect this graph.
     */
    public List<String> getParents(String childData);

    /**
     * Returns the number of nodes in this graph.
     *
     * @return the number of nodes in this graph
     */
    public int numNodes();

    /**
     * Returns the number of edges in this graph.
     *
     * @return the number of edges in this graph
     */
    public int numEdges();
}
```

## 3 Design Decisions

1. **Mutability**: The Graph ADT is designed to be mutable, allowing nodes and edges to be added after construction. This provides flexibility for clients to build and modify graphs as needed.

2. **Node Representation**: Nodes are uniquely identified by their String data. This simplifies the design and makes it easy for clients to reference nodes without needing to keep track of separate node objects.

3. **Edge Representation**: Edges are represented by their source node, destination node, and String label. The design allows multiple edges between the same pair of nodes, potentially with the same label.

4. **Duplicate Edges**: The Graph ADT leaves it to the implementation to decide whether to allow duplicate edges (same source, destination, and label). The specification indicates that `addEdge` may or may not add a duplicate edge.

5. **Defensive Copying**: The specification ensures that clients cannot modify the internal representation of the graph through returned collections. All collections returned by methods are copies of the internal data.

6. **Error Handling**: The specification uses `@spec.requires` to indicate preconditions that clients must satisfy. This is a design choice to keep the interface simple and avoid exception handling complexity.

7. **Reflexive Edges**: The Graph ADT allows reflexive edges (edges from a node to itself) as specified in the problem statement.

## 4 GraphWrapper Interface

The GraphWrapper class provides a specific interface for testing the Graph ADT. It implements the required methods while delegating the actual graph operations to the Graph ADT:

- `public GraphWrapper()`: Constructs a new GraphWrapper with an empty Graph.
- `public void addNode(String nodeData)`: Adds a node to the graph.
- `public void addEdge(String parentNode, String childNode, String edgeLabel)`: Adds an edge from the parent node to the child node with the given label.
- `public Iterator<String> listNodes()`: Returns an iterator over all nodes in lexicographical order.
- `public Iterator<String> listChildren(String parentNode)`: Returns an iterator over all children of the given parent node in the format "childNode(edgeLabel)" in lexicographical order.
- `public String listChildrenXML(String parentNode)`: Returns an XML representation of all children of the given parent node in lexicographical order.