CSE222 / CSE505 SPRING 2022

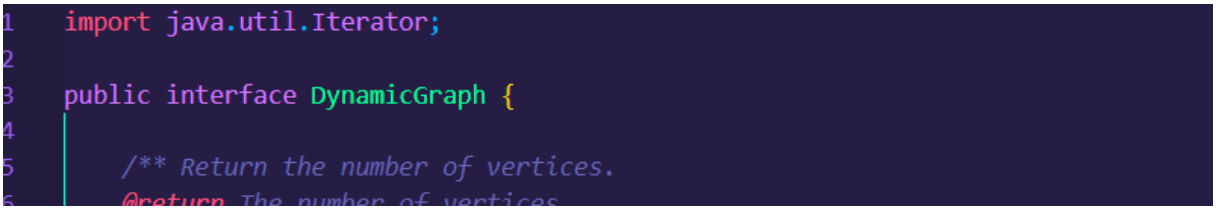
HOMEWORK 8 REPORT

Burcu Sultan Orhan

1901042667

1- Detailed System Requirements

First there needs to be DynamicGraph interface to be implemented

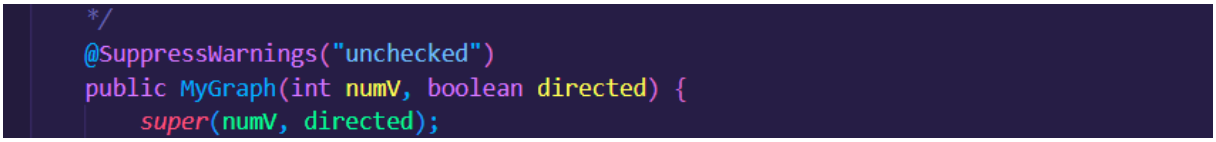


```java
import java.util.Iterator;

public interface DynamicGraph {

    /** Return the number of vertices.
     @return The number of vertices
```

Then there needs to be AbstractGraph class to be extended

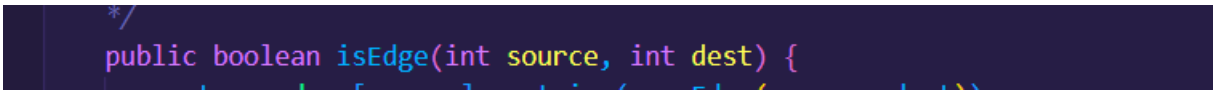
```java
import java.io.BufferedReader;
import java.io.IOException;

public abstract class AbstractGraph {
```

And in MyGraph class, there needs to be a consructor like this:



```java
     */
    @SuppressWarnings("unchecked")
    public MyGraph(int numV, boolean directed) {
        super(numV, directed);
```

There needs to be isEdge method to check whether an edge exist between two vertices



```java
     */
    public boolean isEdge(int source, int dest) {
```

Then there needs to be methods such as:
-insert, which takes an edge as parameter and adds the edge to the graph
-Iterator, which iterates through edges
-getEdge, which takes two vertices' IDs as parameter and returns the edge between these vertices

```
public void insert(Edge edge) {
    edges[edge.getSource()].add(edge);
    if (!isDirected()) {
    edges[edge.getDest()].add(new Edge(edge.getDest(),
                                       edge.getSource(),
                                       edge.getWeight()));
    }
}

public Iterator <Edge> edgeIterator(int source) {
    return edges[source].iterator();
}

/** Get the edge between two vertices
    @param source The source
    @param dest The destination
    @return the edge between these two vertices
*/
public Edge getEdge(int source, int dest) {
    Edge target = new Edge(source, dest, Double.POSITIVE_INFINITY);
    for (Edge edge : edges[source]) {
    if (edge.equals(target))
        return edge;
```

And there needs to be two different methods to get certain vertex, one takes a key as parameter, other takes a label as a parameter

```
public Vertex getVertex(String key){
    for(Vertex thisVertex : vertices){
        if(thisVertex.getData().get(key) != null){
            return thisVertex;
        }
    }
    return null;
}

public Vertex getVertexLabel(String label){
    for(Vertex thisVertex : vertices){
        if(thisVertex.getLabel().compareTo(label) == 0){
            return thisVertex;
        }
    }
    return null;
}
```

There needs to be methods such as:
-newVertex, which creates a new vertex with given label and weight
-addVertex, which adds the given vertex to the graph
-addEdge, which creates and adds an edge whose source, destination and weight information are given

```java
    */
    public Vertex newVertex(String label, double weight) {
        Vertex thisVertex = new Vertex(label, weight);
        return thisVertex;
    }

    /**
     * This method adds given vertex to graph
     * @param new_vertex vertex to-be-added
     * @return added vertex
     */
    public Vertex addVertex(Vertex new_vertex) {
        try{vertices.add(new_vertex);
        new_vertex.setIndex(vertices.size());}catch(NullPointerException e){}
        return new_vertex;
    }

    /**
     * This method adds an edge with given parameters to the graph
     * @param vertexID1 ID of the source vertex
     * @param vertexID2 ID of the destination vertex
     * @param weight weight of the edge
     * @return added edge
     */
    public Edge addEdge(int vertexID1, int vertexID2, double weight) {
        Edge thisEdge = new Edge(vertexID1, vertexID2, weight);
        edges[thisEdge.getSource()].add(thisEdge);
        return thisEdge;
    }
```

And there needs to be methods such as:
-removeEdge, which removes the edge between two given vertices
-removeVertex, which removes the vertex whose ID is given
-removeVertex, which removes the vertex whose label is given

```java
public Edge removeEdge(int vertexID1, int vertexID2) {
    edges[vertexID1].remove(getEdge(vertexID1, vertexID2));
    return getEdge(vertexID1, vertexID2);
}

/**
 * This method removes vertex with the given ID
 * @param vertexID ID of the vertex to be removed
 * @return removed vertex
 */
public Vertex removeVertex(int vertexID) {
    Vertex temp = new Vertex(" ", 0);
    for(Vertex thisVertex : vertices){
        if(thisVertex.getIndex() == vertexID){
            temp = thisVertex;
            vertices.remove(thisVertex);
        }
    }
    return temp;
}

/**
 * This method removes vertex with the given label
 * @param label label of the vertex to be removed
 * @return removed vertex
 */
public Vertex removeVertex(String label) {
    Vertex temp = new Vertex(" ", 0);
    for(Vertex thisVertex : vertices){
        int bools = thisVertex.getLabel().compareTo(label);
        if(bools == 0){
```

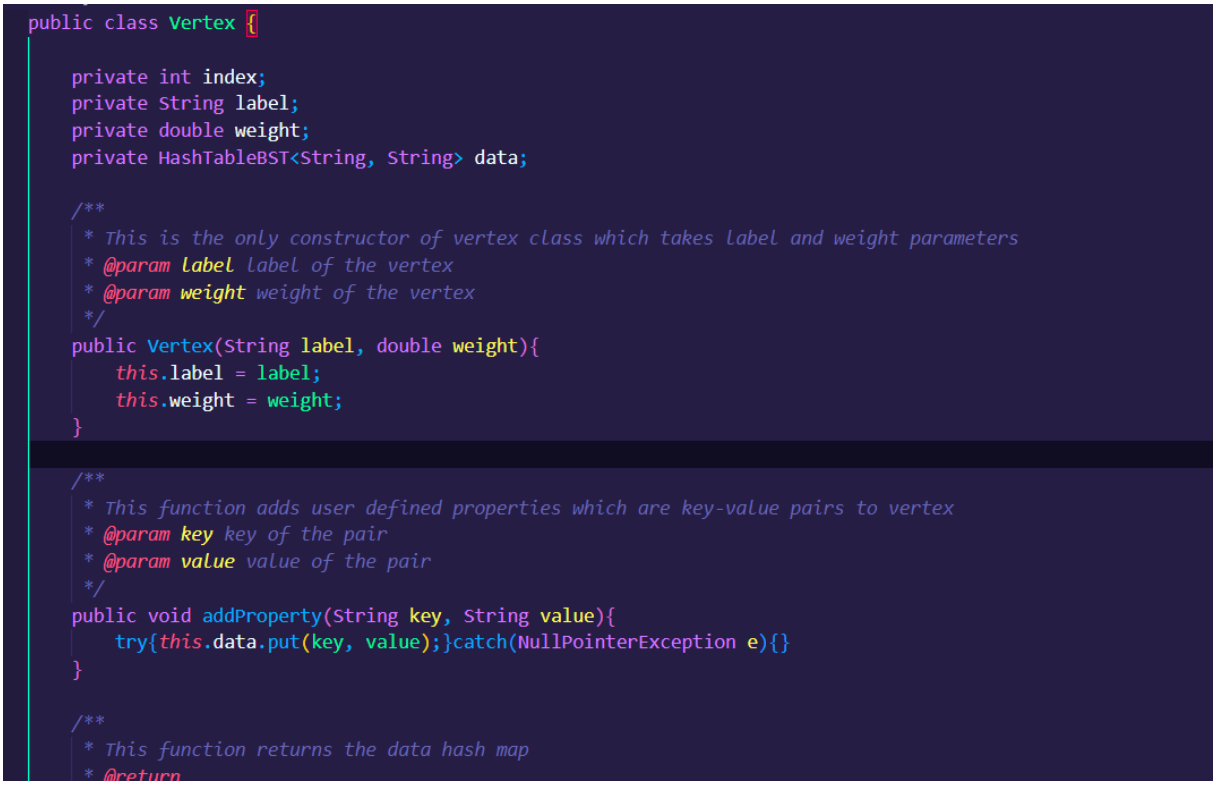
And there needs to be methods such as:
-filterVertices, which filters the vertices
-exportMatrix, which converts our adjacency list graph to an adjacency matrix graph

```
 */
public Vertex filterVertices(String key, String filter) {
    if(getVertex(key) != null){
        MyGraph newGraph = new MyGraph(10, true);
        Vertex localVertex = new Vertex("burcu", 1.0);
        newGraph.addVertex(localVertex);
        localVertex.addProperty(key, filter);
    }
    return null;
}

/**
 * This method exports adjacency list graph to an adjacency matrix graph
 * @return matrix
 */
public double[][] exportMatrix() {
    double[][] data = new double[100][3];
    for(int i=0; i<100; i++){
        for(Edge localEdge : edges[i]){
```

And lastly for MyGraph class, there needs to be printGraph method to print the graph

```
 * This method prints the graph based on edges
 */
public void printGraph() {
    try{for(int i=0; i<100; i++){
```

Secondly, there needs to be a Vertex class to implement vertices, its constructor which takes a label and a weight as parameters, and an addProperty method which takes key-value pairs as parameters and attach them to that vertex



```java
public class Vertex {

    private int index;
    private String label;
    private double weight;
    private HashTableBST<String, String> data;

    /**
     * This is the only constructor of vertex class which takes label and weight parameters
     * @param label label of the vertex
     * @param weight weight of the vertex
     */
    public Vertex(String label, double weight){
        this.label = label;
        this.weight = weight;
    }

    /**
     * This function adds user defined properties which are key-value pairs to vertex
     * @param key key of the pair
     * @param value value of the pair
     */
    public void addProperty(String key, String value){
        try{this.data.put(key, value);}catch(NullPointerException e){}
    }

    /**
     * This function returns the data hash map
     * @return
```

2- Problem Solving Approach

For this homework, I needed to implement three different abstract data types. I used Binary Search Trees for chaining in Hash Map, and I used Hash Maps for storing key-value pairs in Vertex. I used ArrayLists to collect vertices in Graph and LinkedLists to collect edges in Graph.

3- Test Cases

```
MyGraph ggraph = new MyGraph(10, true);

ggraph.addVertex(ggraph.newVertex("burcu", 6)).addProperty("color", "orange");
ggraph.addVertex(ggraph.newVertex("sultan", 1)).addProperty("color", "purple");
ggraph.addVertex(ggraph.newVertex("orhan", 3)).addProperty("color", "white");

ggraph.addEdge(ggraph.getVertexLabel("burcu").getIndex(), ggraph.getVertexLabel("sultan").getIndex(), 8);
ggraph.addEdge(ggraph.getVertexLabel("burcu").getIndex(), ggraph.getVertexLabel("orhan").getIndex(), 2);
ggraph.addEdge(ggraph.getVertexLabel("orhan").getIndex(), ggraph.getVertexLabel("sultan").getIndex(), 5);

System.out.println("\nThree vertices and three edges added:");
ggraph.printGraph();

ggraph.removeEdge(ggraph.getVertexLabel("burcu").getIndex(), ggraph.getVertexLabel("sultan").getIndex());

System.out.println("\nOne edge removed: ");
ggraph.printGraph();
```

5- Running Command and Results