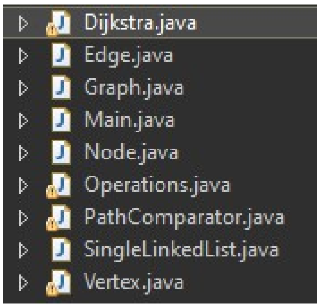
CME 2201 – ASSIGNMENT 2

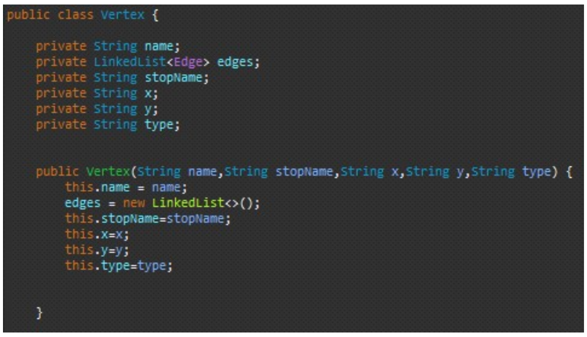
Name: Nilay
Surname: Yücel
Number: 2017510082

A total of **8 classes** were used. The Edge and Vertex classes were used to create the graph class. The **'Linked List'** structure was used to hold the other vertexes to which each vertex was connected. In this way, all the edges of the vertex were placed in a **'Linked List'**. In order to create graph in Graph class, all vertexs were kept in **'Hash Map'** structure. This graph was used in the dijkstra class to find the shortest path. In the Dijkstra class, a hash map was used to keep path and distance. **' Priority Queue'** was used to find the shortest path. A comparator class was created to sort the data sent into **' Priority Queue'**. In the operations class, file reading operations were performed and the datas was added to the relevant data structure.

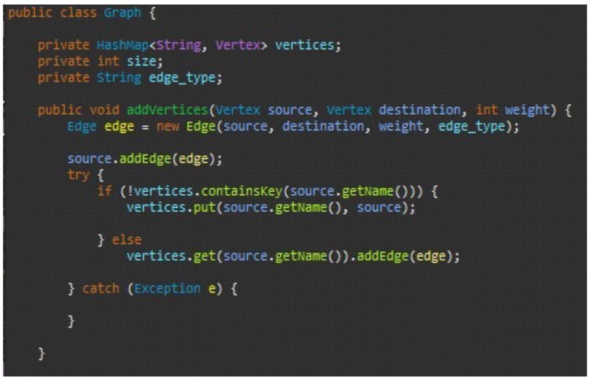
CLASSES



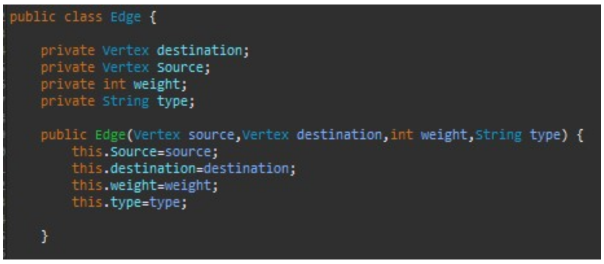
```
Dijkstra.java
Edge.java
Graph.java
Main.java
Node.java
Operations.java
PathComparator.java
SingleLinkedList.java
Vertex.java
```



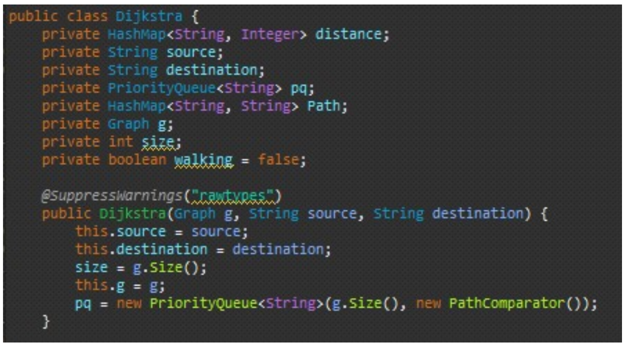
```
public class Vertex {

    private String name;
    private LinkedList<Edge> edges;
    private String stopName;
    private String x;
    private String y;
    private String type;


    public Vertex(String name,String stopName,String x,String y,String type) {
        this.name = name;
        edges = new LinkedList<>();
        this.stopName=stopName;
        this.x=x;
        this.y=y;
        this.type=type;


    }
```



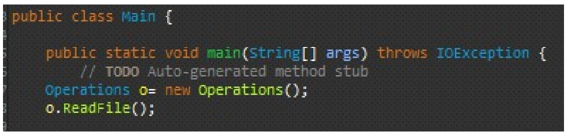
```
public class Graph {

    private HashMap<String, Vertex> vertices;
    private int size;
    private String edge_type;

    public void addVertices(Vertex source, Vertex destination, int weight) {
        Edge edge = new Edge(source, destination, weight, edge_type);

        source.addEdge(edge);
        try {
            if (!vertices.containsKey(source.getName())) {
                vertices.put(source.getName(), source);

            } else
                vertices.get(source.getName()).addEdge(edge);

        } catch (Exception e) {

        }

    }
```



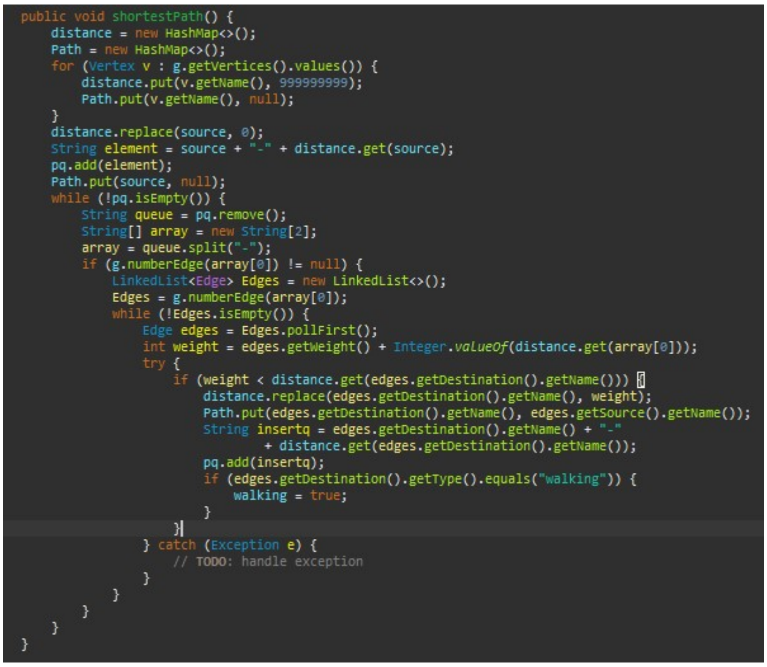
```
public class Edge {

    private Vertex destination;
    private Vertex Source;
    private int weight;
    private String type;

    public Edge(Vertex source,Vertex destination,int weight,String type) {
        this.Source=source;
        this.destination=destination;
        this.weight=weight;
        this.type=type;

    }
```



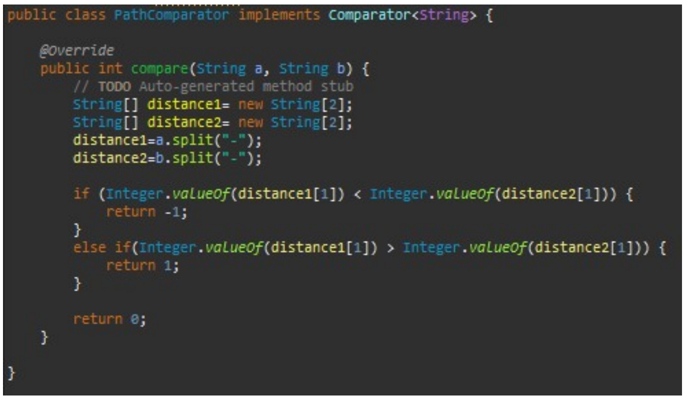
```
public class Dijkstra {
    private HashMap<String, Integer> distance;
    private String source;
    private String destination;
    private PriorityQueue<String> pq;
    private HashMap<String, String> Path;
    private Graph g;
    private int size;
    private boolean walking = false;

    @SuppressWarnings("rawtypes")
    public Dijkstra(Graph g, String source, String destination) {
        this.source = source;
        this.destination = destination;
        size = g.Size();
        this.g = g;
        pq = new PriorityQueue<String>(g.Size(), new PathComparator());
    }
```



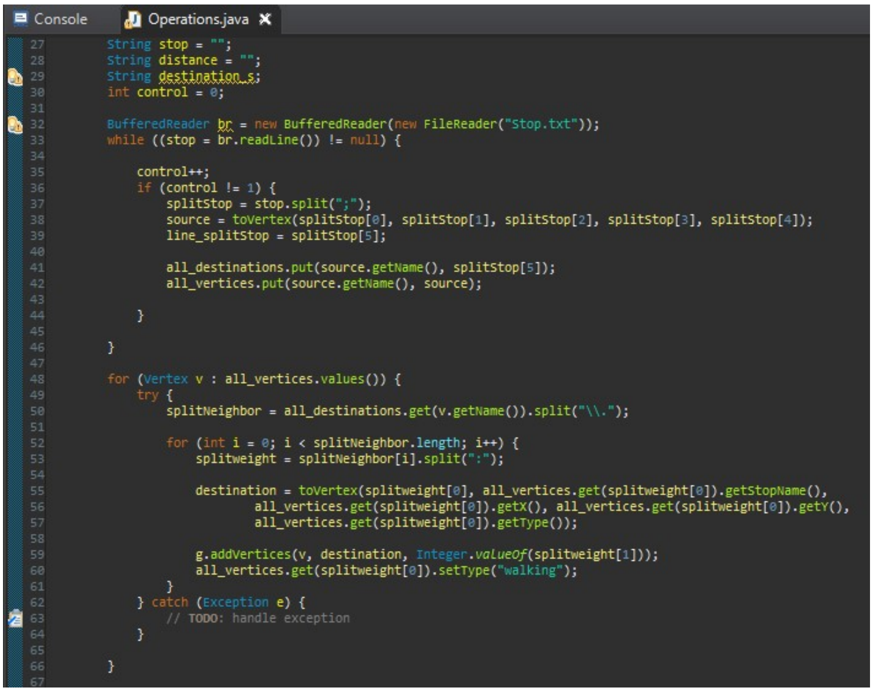
```
public class Main {

    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        Operations o= new Operations();
        o.ReadFile();
```



```java
public void shortestPath() {
    distance = new HashMap<>();
    Path = new HashMap<>();
    for (Vertex v : g.getVertices().values()) {
        distance.put(v.getName(), 999999999);
        Path.put(v.getName(), null);
    }
    distance.replace(source, 0);
    String element = source + "-" + distance.get(source);
    pq.add(element);
    Path.put(source, null);
    while (!pq.isEmpty()) {
        String queue = pq.remove();
        String[] array = new String[2];
        array = queue.split("-");
        if (g.numberEdge(array[0]) != null) {
            LinkedList<Edge> Edges = new LinkedList<>();
            Edges = g.numberEdge(array[0]);
            while (!Edges.isEmpty()) {
                Edge edges = Edges.pollFirst();
                int weight = edges.getWeight() + Integer.valueOf(distance.get(array[0]));
                try {
                    if (weight < distance.get(edges.getDestination().getName())) {
                        distance.replace(edges.getDestination().getName(), weight);
                        Path.put(edges.getDestination().getName(), edges.getSource().getName());
                        String insertq = edges.getDestination().getName() + "-"
                                + distance.get(edges.getDestination().getName());
                        pq.add(insertq);
                        if (edges.getDestination().getType().equals("walking")) {
                            walking = true;
                        }
                    }
                } catch (Exception e) {
                    // TODO: handle exception
                }
            }
        }
    }
}
```

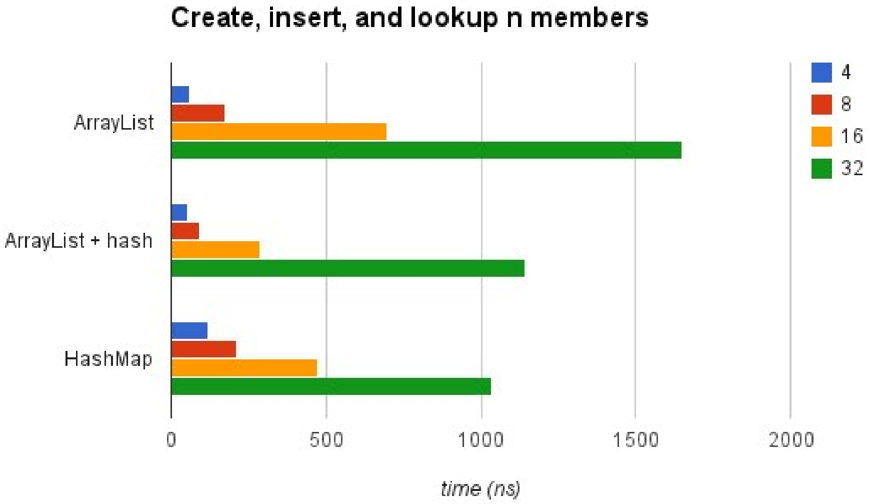


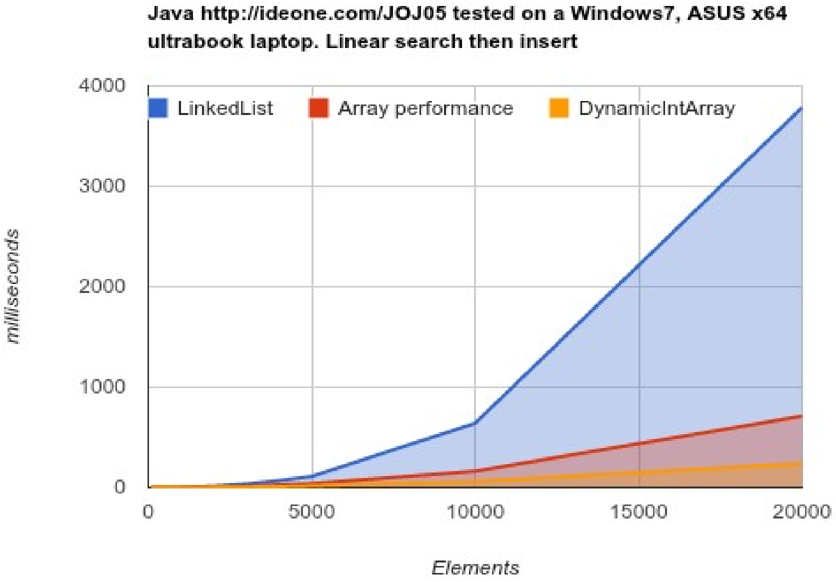
```java
public class PathComparator implements Comparator<String> {

    @Override
    public int compare(String a, String b) {
        // TODO Auto-generated method stub
        String[] distance1= new String[2];
        String[] distance2= new String[2];
        distance1=a.split("-");
        distance2=b.split("-");

        if (Integer.valueOf(distance1[1]) < Integer.valueOf(distance2[1])) {
            return -1;
        }
        else if(Integer.valueOf(distance1[1]) > Integer.valueOf(distance2[1])) {
            return 1;
        }

        return 0;
    }

}
```



```java
        String stop = "";
        String distance = "";
        String destination_s;
        int control = 0;

        BufferedReader br = new BufferedReader(new FileReader("Stop.txt"));
        while ((stop = br.readLine()) != null) {

            control++;
            if (control != 1) {
                splitStop = stop.split(";");
                source = toVertex(splitStop[0], splitStop[1], splitStop[2], splitStop[3], splitStop[4]);
                line_splitStop = splitStop[5];

                all_destinations.put(source.getName(), splitStop[5]);
                all_vertices.put(source.getName(), source);

            }

        }

        for (Vertex v : all_vertices.values()) {
            try {
                splitNeighbor = all_destinations.get(v.getName()).split("\\.");

                for (int i = 0; i < splitNeighbor.length; i++) {
                    splitweight = splitNeighbor[i].split(":");

                    destination = toVertex(splitweight[0], all_vertices.get(splitweight[0]).getStopName(),
                            all_vertices.get(splitweight[0]).getX(), all_vertices.get(splitweight[0]).getY(),
                            all_vertices.get(splitweight[0]).getType());

                    g.addVertices(v, destination, Integer.valueOf(splitweight[1]));
                    all_vertices.get(splitweight[0]).setType("walking");
                }
            } catch (Exception e) {
                // TODO: handle exception
            }

        }
```

PERFORMANCES


Create, insert, and lookup n members
ArrayList
ArrayList + hash
HashMap
time (ns)


Java http://ideone.com/JOJ05 tested on a Windows7, ASUS x64 ultrabook laptop. Linear search then insert
LinkedList
Array performance
DynamicIntArray
milliseconds
Elements