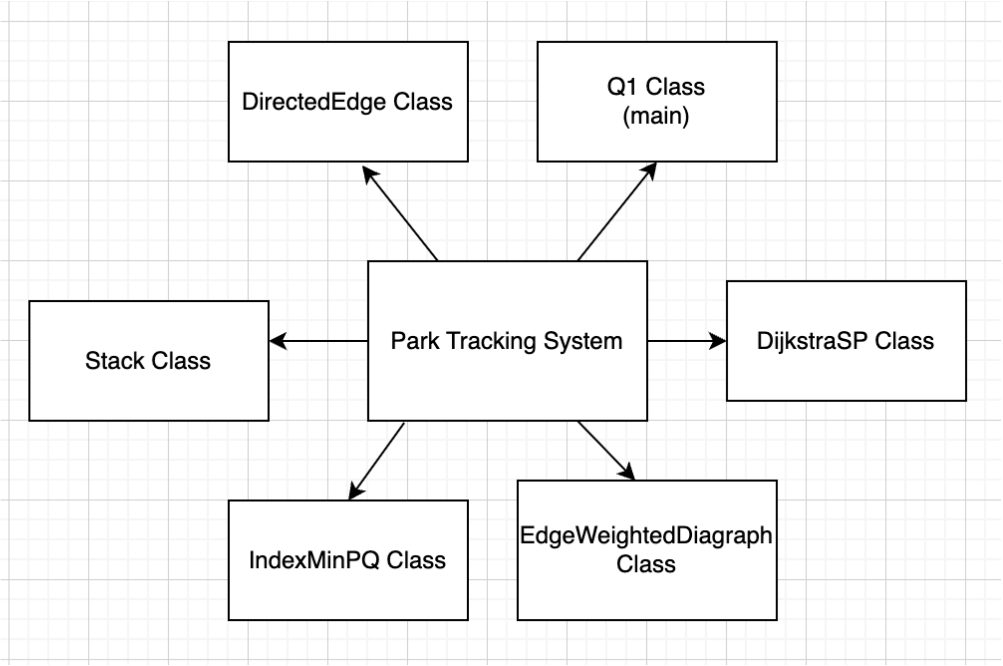
CMPE343 HW -3
Burak SAĞLAM : 13760307838 - Section 2
Sarp ARSLAN : 11458145526 - Section 3

QUESTION 1

Problem Statement and Code Design

In this report, we are going to implement Park Tracking System for Esenboga Airport. We are going to implement this by using graph structure. We have implemented class to solve this park problem in Esenboga Airport. These sub-module shows to structure that used in the system.



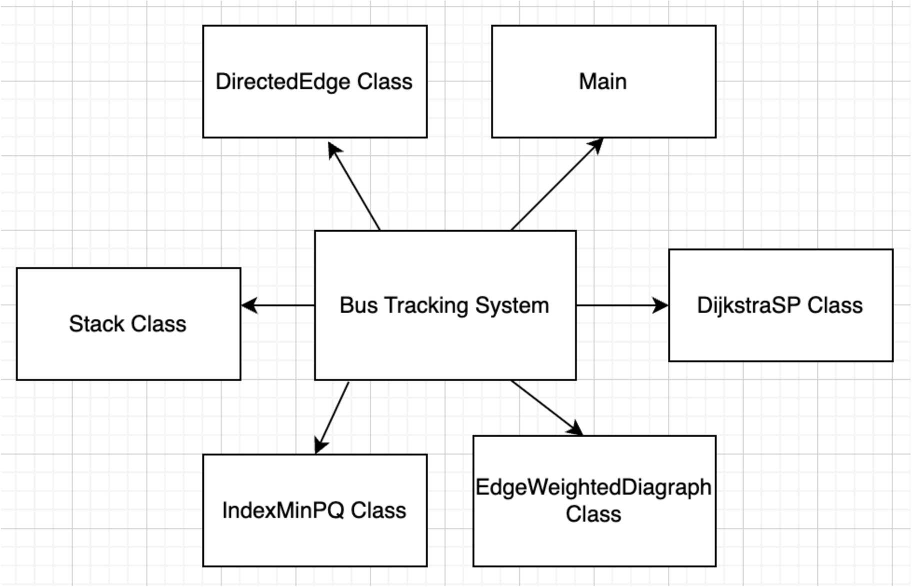
TESTING

In testing my program can work with any graph data structures. We tested our code with given inputs and our inputs. At first, we did not able to check -1 condition in output, in addition to that there were some errors caused by wrong impelemtation of the algorithm and ArrayOutOfBounds as an effect of wrongly adding edges to the graph. After a research we handled the problems.

QUESTION 2

Problem Statement and Code Design

In this report, we are going to implement Bus Tracking System for buses in Ankara. We are going to implement this by using graph structure. We have implemented 5 class to solve to track buses in Ankara. These sub-module shows to structure that used in the system.



TESTING

At first the common problem with our code was that the edge weights for the graph were not calculated correctly. In the early stages of the code, our current weight calculations only took into account the length of the route, but we also took into account the time it would take for another bus to arrive when we were at a stop, and thus the edge weights were calculated.

Implementation, Functionality

Q1

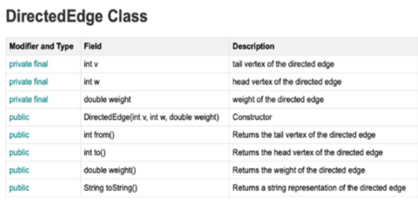
To be able to explain the functionality of this program we need to explain how this code works. This program includes a function that reads user inputs and assigns these stands and their weights to a structure. Parking lot program is a system that helps the user to manage parking spaces using a graph structure. The user makes entries that include starting points, target points, and weights (distance, cost, etc.). These entries are converted into a graph structure and the shortest route or the most suitable route is found according to certain criteria such as short distance or minimum cost. The program finds the shortest path on the graph using the Dijkstra algorithm. The results show the user the nodes of the shortest path and the total cost (with an additional charge if any). It also sorts the most suitable roads according to the number of cars received from the user. Thus, the user can see the routes where the cars are optimally filled in a certain order. In the it prints the results for chosen car number.

Q2

To explain the functionality of this program I need to explain how this code works.
Finding the shortest time to get to each station in Ankara Kızılay while taking into account a number of bus stations and their schedules is the issue at hand. In Ankara Kızılay, there are N

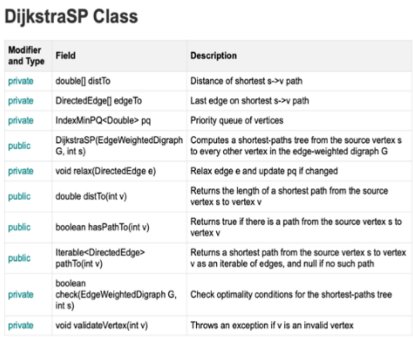
bus stops and M buses that operate. Every bus has a schedule with t stations on it. The buses continuously move from one station to the next in accordance with their schedules, and when they reach the end, they turn around and restart at the beginning. If a person is at the same station as a bus at any particular time, they can board the bus and ride along with it. Any station can be chosen as the place to get off the bus. The program gets inputs from user and produce the graph structure to work with. After that we get the shorthest path with Dijkstra Algorithm. In the end program calculates the times and prints them.

1. **DirectedEdge:** Initializes a directed edge from vertex to vertex with given weight .
2. **DijkstraSP**: Computes a shortest path from the source vertex to every other vertex in the edge weighted diagraph.
3. **Main**: Main Class fort he program.
4. **IndexMinPQ :** Initializes an empty indexed priority queue with indices between 0 and-1.
5. **EdgeWeightedDigraph :** Initializes an empty edge-weighted digraph.



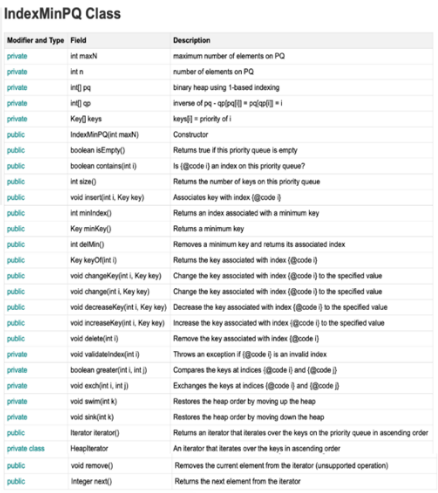
DirectedEdge Class

Modifier and Type	Field	Description
private final	int v	tail vertex of the directed edge
private final	int w	head vertex of the directed edge
private final	double weight	weight of the directed edge
public	DirectedEdge(int v, int w, double weight)	Constructor
public	int from()	Returns the tail vertex of the directed edge
public	int to()	Returns the head vertex of the directed edge
public	double weight()	Returns the weight of the directed edge
public	String toString()	Returns a string representation of the directed edge



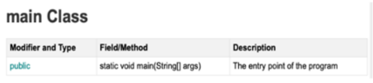
DijkstraSP Class

Modifier and Type	Field	Description
private	double[] distTo	Distance of shortest s->v path
private	DirectedEdge[] edgeTo	Last edge on shortest s->v path
private	IndexMinPQ<Double> pq	Priority queue of vertices
public	DijkstraSP(EdgeWeightedDigraph G, int s)	Computes a shortest-paths tree from the source vertex s to every other vertex in the edge-weighted digraph G
private	void relax(DirectedEdge e)	Relax edge e and update pq if changed
public	double distTo(int v)	Returns the length of a shortest path from the source vertex s to vertex v
public	boolean hasPathTo(int v)	Returns true if there is a path from the source vertex s to vertex v
public	Iterable<DirectedEdge> pathTo(int v)	Returns a shortest path from the source vertex s to vertex v as an iterable of edges, and null if no such path
private	boolean check(EdgeWeightedDigraph G, int s)	Check optimality conditions for the shortest-paths tree
private	void validateVertex(int v)	Throws an exception if v is an invalid vertex



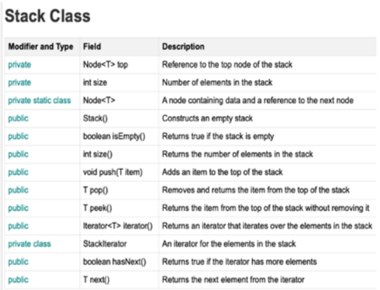
IndexMinPQ Class

Modifier and Type	Field	Description
private	int maxN	maximum number of elements on PQ
private	int n	number of elements on PQ
private	int[] pq	binary heap using 1-based indexing
private	int[] qp	inverse of pq - qp[pq[i]] = pq[qp[i]] = i
private	Key[] keys	keys[i] = priority of i
public	IndexMinPQ(int maxN)	Constructor
public	boolean isEmpty()	Returns true if this priority queue is empty
public	boolean contains(int i)	Is {@code i} an index on this priority queue?
public	int size()	Returns the number of keys on this priority queue
public	void insert(int i, Key key)	Associates key with index {@code i}
public	int minIndex()	Returns an index associated with a minimum key
public	Key minKey()	Returns a minimum key
public	int delMin()	Removes a minimum key and returns its associated index
public	Key keyOf(int i)	Returns the key associated with index {@code i}
public	void changeKey(int i, Key key)	Change the key associated with index {@code i} to the specified value
public	void change(int i, Key key)	Change the key associated with index {@code i} to the specified value
public	void decreaseKey(int i, Key key)	Decrease the key associated with index {@code i} to the specified value
public	void increaseKey(int i, Key key)	Increase the key associated with index {@code i} to the specified value
public	void delete(int i)	Remove the key associated with index {@code i}
private	void validateIndex(int i)	Throws an exception if {@code i} is an invalid index
private	boolean greater(int i, int j)	Compares the keys at indices {@code i} and {@code j}
private	void exch(int i, int j)	Exchanges the keys at indices {@code i} and {@code j}
private	void swim(int k)	Restores the heap order by moving up the heap
private	void sink(int k)	Restores the heap order by moving down the heap
public	Iterator iterator()	Returns an iterator that iterates over the keys on the priority queue in ascending order
private class	HeapIterator	An iterator that iterates over the keys in ascending order
public	void remove()	Removes the current element from the iterator (unsupported operation)
public	Integer next()	Returns the next element from the iterator



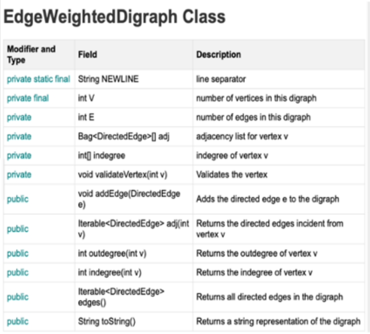
main Class

Modifier and Type	Field/Method	Description
public	static void main(String[] args)	The entry point of the program



Stack Class

Modifier and Type	Field	Description
private	Node<T> top	Reference to the top node of the stack
private	int size	Number of elements in the stack
private static class	Node<T>	A node containing data and a reference to the next node
public	Stack()	Constructs an empty stack
public	boolean isEmpty()	Returns true if the stack is empty
public	int size()	Returns the number of elements in the stack
public	void push(T item)	Adds an item to the top of the stack
public	T pop()	Removes and returns the item from the top of the stack
public	T peek()	Returns the item from the top of the stack without removing it
public	Iterator<T> iterator()	Returns an iterator that iterates over the elements in the stack
private class	StackIterator	An iterator for the elements in the stack
public	boolean hasNext()	Returns true if the iterator has more elements
public	T next()	Returns the next element from the iterator



EdgeWeightedDigraph Class

Modifier and Type	Field	Description
private static final	String NEWLINE	line separator
private final	int V	number of vertices in this digraph
private	int E	number of edges in this digraph
private	Bag<DirectedEdge>[] adj	adjacency list for vertex v
private	int[] indegree	indegree of vertex v
private	void validateVertex(int v)	Validates the vertex
public	void addEdge(DirectedEdge e)	Adds the directed edge e to the digraph
public	Iterable<DirectedEdge> adj(int v)	Returns the directed edges incident from vertex v
public	int outdegree(int v)	Returns the outdegree of vertex v
public	int indegree(int v)	Returns the indegree of vertex v
public	Iterable<DirectedEdge> edges()	Returns all directed edges in the digraph
public	String toString()	Returns a string representation of the digraph

FINAL ASSESSMENTS

We learned about the implementation of the dijsktra algorithm in real life, and we had the chance to use the knowledge we learned in theory in practice. In addition, making the necessary implementations to use this algorithm allowed us to better understand the working principle of this algorithm. At the beggining of homework,choosing the graph strucuture to work with was hard. After understanding the problem we choose the best fit for both problems. Working with Dijkstra Algorithm was hard but we managed to work with and we get the correct outputs. We think that our favorite part of the assignment was researching for the best structures to work with because we learned so many different structures.