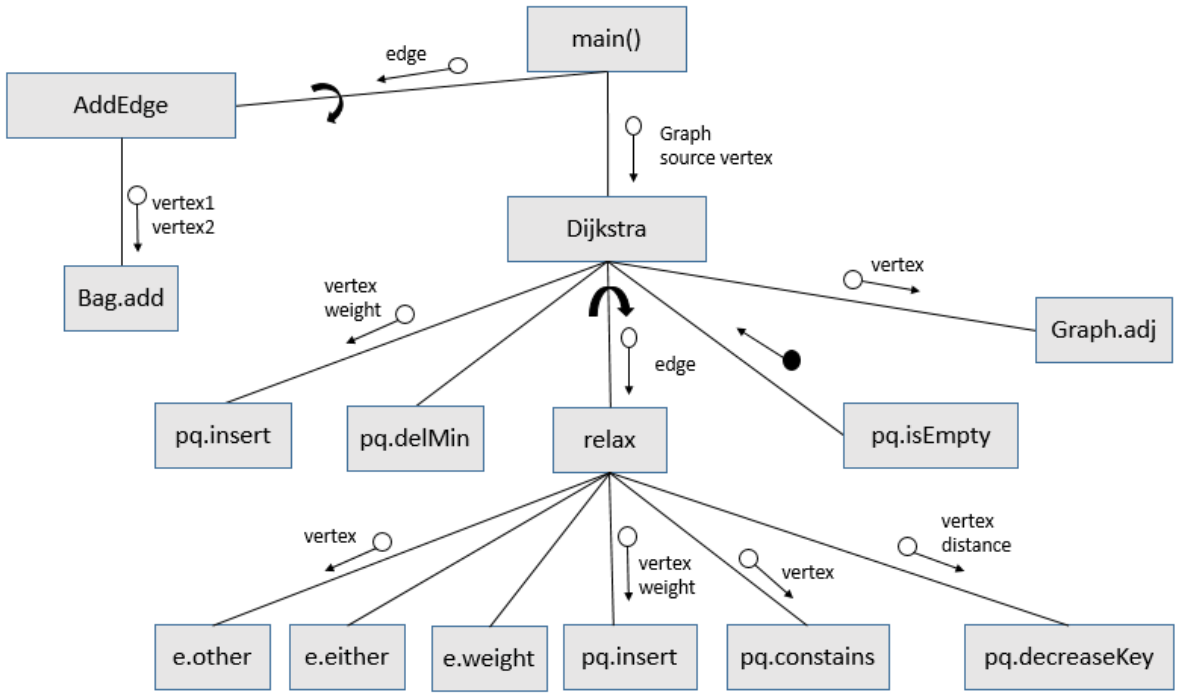
Report

Question 1

➔ Problem Statement and Code Design



➔ Implementation, Functionality

```
DijkstraSP(Graph G, int sourceVertex)
{
   initialize the edgeTo array
   initialize the distances array
   initialize the minimum priority queue

  for (int vertex = 0 , vertex < Graph vertices , vertex++)
     {
       let the distance to each vertex be Positive Infinity
     }
   let the distance to the sourceVertex be 0
   insert the sourceVertex to the min priority queue

   while (the min priority queue is not empty)
    {
      int v = get the minimum value from the priority queue
      for each ( Edge e adjacent to vertex v)
           {
              relax (e)
           }
    }
}
```

```
relax (Edge e)
{
     int v = the vertex on one side of Edge e
     int w = the vertex on the other side of Edge e

 if (distance to w > distance to v + Edge e's weight)
   {
     distance to w = distance to v + Edge e's weight
     edgeTo w = Edge e

     if (w is in the minimum priority queue)
      {
         decrease w and the distance to it to the minimum
          priority queue
      }
     else
      {
         insert w and the distance to it to the minimum
         priority queue
       }
    }
}
```

Main: After reading and storing the input, creating the Graph accordingly, and calling *Dijkstra's* shortest path algorithm, in a for-loop, we park the car in the parking slot with the least cost by finding the shortest path from the first vertex to the parking slot. It performs the mentioned task as follows: If there is no more capacity for cars, it prints out "-1" to the console, else, it finds the parking slot with the minimum cost and available capacity as follows: In a while loop, for each vertex in the graph it checks if it has available capacity, then check if its distance is less than the minimum distance or if the capacity of the current minimum value is 0, then it updates the minimum distance and stores its index value for further comparisons and calculations.

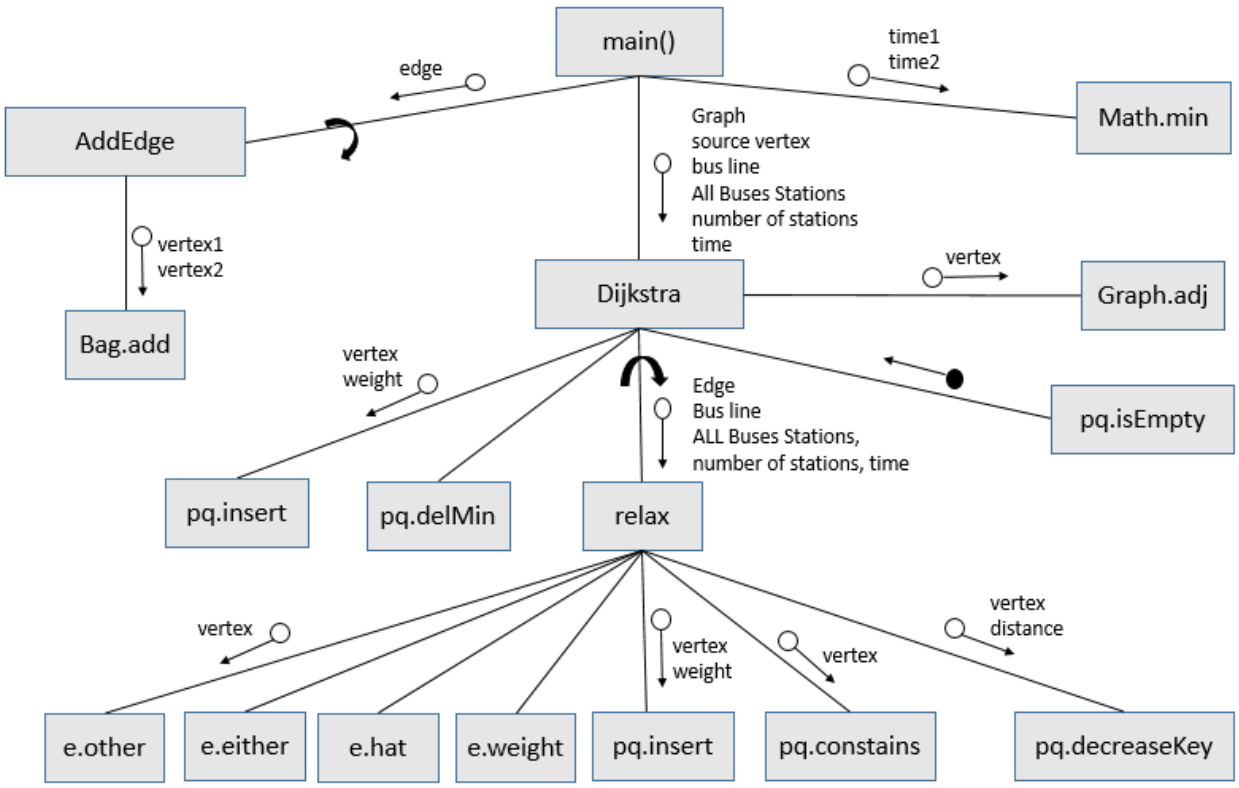
DijkstraSP and relax methods: *DijkstraSP* finds the shortest path from the source vertex to every other vertex in the graph (to every parking slot). It uses the "relax" method as a helper method to find the shortest paths. The *relax* method finds the shortest path from the source vertex to the given edge passed as parameter. It takes an edge and checks the distance to that edge from the source vertex, does the needed distance comparisons and updates the "distTo" array and priority queue when needed.

➔ Testing

For testing this class I tried different graph inputs with different sizes (different number of edges and vertices) to validate that the methods work for different graph sizes. Moreover, I tried changing variables like the parking fee and the weights between the edges to validate that the paths with the minimum cost are calculated effectively and efficiently.

Question 2

➔ Problem Statement and Code Design



➔ **Implementation, Functionality**

```
DijkstraSP(Digraph G, int sourceVertex, int busLine ,
int [] [] All_Buses_Stations, int [] number_of_stations, int time)
{
   initialize the edgeTo array
   initialize the distances array
   initialize the minimum priority queue

  for (int vertex = 0 , vertex < Graph vertices , vertex++)
     {
       let the distance to each vertex be Positive Infinity
     }
   let the distance to the source vertex be 0
   insert the source vertex to the min priority queue

   while (the min priority queue is not empty)
   {
     int v = get the minimum value from the priority queue
     for each ( Edge e adjacent to vertex v)
          {
             relax (e, busLine , All_Buses_Stations,
                    number_of_stations, time)
          }
   }
}
```

```
relax (Edge e, int busLine , int [] [] All_Buses_Stations,
int [] number_of_stations, time) {
     int v = the vertex on one side of Edge e
     int w = the vertex on the other side of Edge e

 if (distance to w > distance to v + Edge e's weight)
  {
     if (the bus line of Edge e == bus line of sourceVertex)
       {
          distance to w = distance to v + Edge e's weight + time
          edgeTo w = Edge e
        }
     else
       {     int waiting_time = 0
             while (the vertex at Edge e's bus line is not v) {waiting_time++}
             distance to w = distance to v + Edge e's weight + waiting_time
             edgeTo w = Edge e
       }

     if (w is in the minimum priority queue)
      {
         decrease w and the distance to it to the minimum priority queue
      }
     else
      {
         insert w and the distance to it to the minimum priority queue } } }
```

Main Class: We may reach the first vertex using several bus lines. Particularly, all the bus lines that include the starting vertex. So we calculate the path to the vertices starting from all the bus lines that include the vertex 1 and compare them with each other to get the minimum among them. This task is performed as follows: for each bus station excluding the 1st station, we check for each bus line available, if the bus line includes the starting vertex 1, we call the DijkstraSP shortest path algorithm that has the Graph, vertex 1 as a source vertex, the bus line number we used to reach vertex 1, an array of all the bus stations of all buses, the number of bus stations in each bus line, and time needed to reach vertex 1 as parameters. And then from DijkstraSP, we get the time needed to reach the vertex we want and we keep comparing all the possible paths to reach a vertex and get the minimum/shortest one among them. We either print the minimum time to reach the vertex or "-1" if the vertex is unreachable.

relax method (modified version): finds the shortest path from the 1st vertex to the given edge considering different bus lines. Along with the Edge, it takes the other parameters passed to the Dijkstra's method as parameters. It does the task as follows: If the bus line of the given edge is the same as the bus line number in the parameter, then it performs the standard calculation to find the distance to the new given vertex while adding the time used to reach vertex 1. Otherwise, if the bus is not at the same station we are at, then we iterate it so it goes to the next station and keep checking until the bus reaches the station we are at. After that we standardly calculate the distance to the vertex while adding the waiting time to it.

➔ Testing

For testing this class I tried the code with different inputs. For example, I tried changing the connections of the bus stations, I tried changing the number of buses giving service and I tried changing the number of stations each bus has. Moreover, I tried different graphs that are connected and disconnected to verify that the code does not only work for the given input, but also gives the right output efficiently based on different given inputs of different values and sizes.

➔ Final Assessments (For All Questions)

The trouble points in completing this assignment were mostly trying to understand the question. For example, I found trouble in understanding the output of the second question in the homework. I could not tell how we got the numbers shown in the output part because in the homework document it was not mentioned that we don't have to consider the first vertex. The most challenging part was trying to figure out how to find the shortest distance in Question 2. It was hard to find a solution method to the given case where there are different bus lines that we have to switch between. However, what I liked about this assignment is that it provided me with hands-on-practice about the new topic of DijkstraSP. I learned from it the algorithms of finding the shortest path from a source vertex to all other vertices in a graph. I also learned the algorithmic thinking behind those methods, by using them and modifying their methods as in the second question (modifying the relax and DijkstraSP methods), which are a very important topic in Data Structures and Algorithms.