I chose to implement a private "*Node*" class that uses a "*destinationToWeightMap*" HashMap within each node. Since each node is connected to another destination node via a weighted edge, the **key** for each HashMap was the integer for the destination node, and the corresponding **value** for the HashMap was the weighted edge. This structure seemed to be more intuitive than using an ArrayList or LinkedList since the nodes connected by edges are already given in the input file. Each node contains a HashMap to associate its destination with an edge, but another HashMap is also used again to structure all the nodes together into one graph. I implemented a "*nodeMap*" HashMap that takes a node's integer as the **key** and maps it to that node in the graph as its **value**. I used a HashMap for the main execution of the algorithm because it is very efficient to grab a node via its integer key, and then access any of its attributes (one of which is the *destinationToWeightMap* HashMap).
HashMap operations are O(1) on average and O(n) for worst case or for iteration. Thus, my program should have an average runtime of O(1) when it comes to inserting nodes or searching for a specific node because each key will only map to one node value (so there won't be many entries with the same key). However, when it comes to searching a shortest path within the Dijkstra's algorithm, my program should have a run time of O(n) since adjacent node value comparisons will be made using basic arrays (n being the number of nodes in the adjacent array). The *nodeMap* HashMap is sorted and will insert a new node into the graph in order. Thus, while searching all of a node's adjacent vertices, each node will be checked in its value order and then the one with the smallest weight will be the one used to build the shortest path. A better way to implement this could be to insert the adjacent nodes in an array by order of lowest to highest weights since each node already has a HashMap of all the weights via its destination node. For searching purposes, that will improve the way of looking for the shortest path.
After doing some research and given that my program uses a HashMap data structure, I would assume that my program uses an average memory space of **32 * SIZE + 4 * CAPACITY** number of bytes. Since a pointer to the next entry is contained within a HashMap along with a key and a value, each entry is considered to typically occupy 32 bytes. Thus, my program with size being the number of vertices must be multiplied by 32 for entry storage alone. In addition, 4 times the amount that the map can hold will be used for the array of entries. Even though a HashMap is very space efficient compared to other data structures such as LinkedList or TreeMaps, a HashMap can still be improved. A THashMap can replace an implementation for a HashMap to save memory. It contains one array for keys, and another for values, so it uses an average memory space of **8 * CAPACITY** number of bytes for storage instead.

Online Sources Used:

https://java-performance.info/memory-consumption-of-java-data-types-2/#:~:text=The%20total%20memory%20consumption%20of,by%20keys%20or%20values!).

https://seniorjava.wordpress.com/2013/09/01/java-objects-memory-size-reference/

… and lecture slides