**Project Description: Djikstra’s Algorithm**

In this program, you will implement Dijkstra's algorithm for shortest paths using an adjacency list representation.

**Specifications**

Your program will read in a graph from a file, compute the shortest path between every pair of vertices (where they exist), output a shortest path table, and output individual paths when requested.

The file format will be as follows. The first line will contain the number of vertices n. Following are text descriptions of the vertices 1 through n (50 chars max length). After that, each line consists of 3 ints representing an edge. If there is a directed edge from vertex 1 to vertex 2 with a weight of 10, the line will be: 1 2 10. A zero for the first integer signals the end of the data. This is a valid example:

4

Home office  
Living room  
Dining room  
Kitchen

1 2 10

1 3 5

2 4 10

2 1 15

3 1 5

3 4 20

0 0 0

You will have a findShortestPath method that computes all shortest paths. A subsequent call to displayAll will output a table formatted as follows:

Description From To Dist Path

Home office

1 2 10 1 2

1 3 5 1 3

1 4 20 1 2 4

Living room

2 1 15 2 1

2 3 20 2 1 3

2 4 10 2 4

Dining room

3 1 5 3 1

3 2 15 3 1 2

3 4 20 3 4

Kitchen

4 1 --

4 2 --

4 3 --

The output for a single detailed path should have a similar format, but it should also include the location descriptions on additional lines. For a call to G.display(2, 3);, the output for this graph should be:

2 3 20 2 1 3

Living room

Home office

Dining room

**More details**

* As mentioned above, you can assume that the input is properly formatted. Here is a data file with the graph above: [HW3.txtPreview the document](https://canvas.uw.edu/courses/1387455/files/64353863/download?wrap=1) (the Canvas preview of this file may make it look like the data is wrong and there are periods in the file - don't trust it).
* Use an adjacency list to store the edges. This will require dynamic memory. Do not use vectors to store the edge lists.
* The graph will have no more than 100 vertices.
* Your class should have the following *public* methods: constructor, copy constructor, destructor, operator=, buildgraph(ifstream &), insertEdge(int, int, int), removeEdge(int, int), findShortestPath(), displayAll(), display(int, int).
* For insertEdge, replace any previous edge that existed between the two vertices.
* Use recursion, not a container (including a string) to display a path. Remember that you work backwards from the destination to the source to recover the path.
* Your code should compile with this driver: [HW3.cppPreview the document](https://canvas.uw.edu/courses/1387455/files/64353862/download?wrap=1)
* I've provided a start to your Graph class here: [Graph.h](https://canvas.uw.edu/courses/1387455/files/64353860/download?wrap=1" \o "Graph.h)[Preview the document](https://canvas.uw.edu/courses/1387455/files/64353860/download?wrap=1) [Graph.cppPreview the document](https://canvas.uw.edu/courses/1387455/files/64353864/download?wrap=1)
* The header file provides you with a Table struct like in the notes and a 2D array of Table elements, which is necessary to record all shortest paths (not just a single source).
* At each step of Dijkstra's algorithm, you will need to determine which vertex to visit next. The best way to accomplish this is actually to use a priority queue (heap). However, you may use a less efficient technique that scans the entire array of vertices for the unvisited vertex with the lowest weight path. This results in an O(n2) algorithm for a single source (if done correctly). You will run the algorithm using each vertex as the source. If you are adventurous, you can use the STL priority\_queue class to improve this.