CS 201, Fall 2020
Homework Assignment 4

Due: 23:59, December 28, 2020

In this homework, you will work on an extended version of the *HPAir* problem that we studied in the class, which is also explained in Section 6.4 in your text book (Carrano's book, 6'th edition). In this assignment, we will provide you with a directed flight graph that shows the connections between cities. A connection between two cities consists of a "flight id" and a "cost". By using this flight graph, your program must find all possible paths (sequences of flights) between the given departure and destination cities. It also should find the least cost path among all paths. A sample graph is shown in Figure 1.

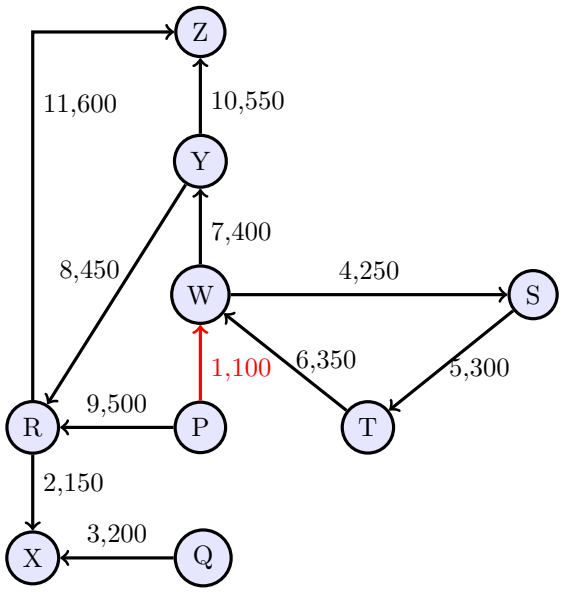

Figure 1: Sample flight graph.

In Figure 1, each vertex corresponds to a city and each edge represents a flight between two cities. Each edge has a flight id and a cost. For example, in Figure 1, the red colored edge's id is "1" and its cost is "100". The graph can be stored by using adjacency lists as described in the Carrano book or by using an adjacency matrix where the rows and columns correspond to individual cities. Given this graph, when the departure city is selected as "`P`" and the destination city is selected as "`Z`", your program must find the paths "`P` → `R` → `Z`", "`P` → `W` → `Y` → `Z`", and "`P` → `W` → `Y` → `R` → `Z`", and since the second path has a cost of 1050, it is selected as the least cost path.
We will provide you with three files: `cityFile`, `flightFile`, and `requestFile`. The "`cityFile`" includes the names of all cities in the flight graph. Each line has exactly one city name. The file for the example graph is:

```
P
Q
R
S
T
W
X
Y
Z
```

City names can have any number of letters and can include spaces (e.g., Los Angeles). They can be given in an arbitrary order (i.e., not necessarily in alphabetical order).

The "flightFile" includes the edge list of the flight graph where each line describes an edge with its origin city, destination city, flight id, and cost, separated by commas. The file for the example graph is:

```
P,W,1,100
R,X,2,150
Q,X,3,200
W,S,4,250
S,T,5,300
T,W,6,350
W,Y,7,400
Y,R,8,450
P,R,9,500
Y,Z,10,550
R,Z,11,600
```

The flights can be given in an arbitrary order (i.e., not necessarily sorted according to flight id or cost). Both flight ids and costs can be assumed to be positive integers.
The "requestFile" includes a pair of city names in each line. First city name is assumed to be the departure city and the second city name is assumed to be the destination city. Each line has exactly one pair of cities. An example is:

```
P,Z
Q,X
```

Your solution must be implemented in a class called FlightMap. Below is the required public part of the FlightMap class. The interface for the class must be written in a file called FlightMap.h and its implementation must be written in a file called FlightMap.cpp. You can define additional public and private member functions and data members in this class. You can also define additional classes in your solution.

```
class FlightMap {

public:
    FlightMap( const string cityFile, const string flightFile );
    ~FlightMap();

    void displayAllCities() const;
    void displayAdjacentCities( const string cityName ) const;
    void displayFlightMap() const;

    void findFlights( const string deptCity, const string destCity ) const;
    void findLeastCostFlight( const string deptCity, const string destCity )
        const;

    void runFlightRequests( const string requestFile ) const;
};
```

The member functions are defined as follows:

FlightMap: Constructor. Reads the flight graph information from the files cityFile and flightFile given as parameters and stores the flight graph. The graph can be stored by using adjacency lists as described in the Carrano book or by using an adjacency matrix where the rows and columns correspond to individual cities.

displayAllCities: Displays all cities in the flight graph.

`displayAdjacentCities:` Displays the cities adjacent to the given city (i.e., all cities that have a flight coming from the given city).

`displayFlightMap:` Displays all adjacent cities for all cities in the graph. In other words, this function displays the whole flight graph.

`findFlights:` Finds and displays all paths (sequences of flights) from the departure city to the destination city. If there is no such path, displays a warning message.

`findLeastCostFlight:` Finds and displays all paths and identifies the least cost path from the departure city to the destination city. If there are multiple paths that have the same lowest cost, only one of them will be displayed (any one of them is acceptable).

`runFlightRequests:` For each request given in the `requestFile`, finds and displays all paths from the departure city to the destination city as well as the least cost path.

Here is an example test program that uses this class and the corresponding output. We will use a similar program to test your solution so make sure that the name of the class is `FlightMap`, its interface is in the file called `FlightMap.h`, and the required functions are defined as shown above.

Example test code:

```
#include "FlightMap.h"

int main() {

    FlightMap fm( "files/cityFile.txt", "files/flightFile.txt" );

    cout << "The list of the cities that HPAir serves is given below:" << endl;
    fm.displayAllCities();
    cout << endl;

    cout << "The cities adjacent to W are:" << endl;
    fm.displayAdjacentCities( "W" );
    cout << endl;

    cout << "The whole flight map is shown below:" << endl;
    fm.displayFlightMap();
    cout << endl;

    fm.findFlights( "W", "Z" );
    cout << endl;

    fm.findFlights( "S", "P" );
    cout << endl;

    fm.findLeastCostFlight( "Y", "Z" );
    cout << endl;

    fm.findLeastCostFlight( "P", "X" );
    cout << endl;

    fm.runFlightRequests( "files/requestFile.txt" );

    return 0;
}
```

Output of the example test code:

```
The list of the cities that HPAir serves is given below:
P, Q, R, S, T, W, X, Y, Z,

The cities adjacent to W are:
W -> Y, S,

The whole flight map is shown below:
P -> R, W,
Q -> X,
R -> Z, X,
S -> T,
T -> W,
W -> Y, S,
X ->
Y -> Z, R,
Z ->

Request is to fly from W to Z:
Flight #7 from W to Y    Cost: 400 TL
Flight #8 from Y to R    Cost: 450 TL
Flight #11 from R to Z   Cost: 600 TL
Total Cost .......... 1450 TL

Flight #7 from W to Y    Cost: 400 TL
Flight #10 from Y to Z   Cost: 550 TL
Total Cost .......... 950 TL

Request is to fly from S to P:
Sorry. HPAir does not fly from S to P

Request is to fly from Y to Z:
Flight #8 from Y to R    Cost: 450 TL
Flight #11 from R to Z   Cost: 600 TL
Total Cost .......... 1050 TL

Flight #10 from Y to Z   Cost: 550 TL
Total Cost .......... 550 TL

A least cost path from Y to Z is Y -> Z and its cost is 550 TL

Request is to fly from P to X:
Flight #1 from P to W    Cost: 100 TL
Flight #7 from W to Y    Cost: 400 TL
Flight #8 from Y to R    Cost: 450 TL
Flight #2 from R to X    Cost: 150 TL
Total Cost .......... 1100 TL

Flight #9 from P to R    Cost: 500 TL
Flight #2 from R to X    Cost: 150 TL
Total Cost .......... 650 TL

A least cost path from P to X is P -> R -> X and its cost is 650 TL

Request is to fly from P to Z:
```

```
Flight #1 from P to W    Cost: 100 TL
Flight #7 from W to Y    Cost: 400 TL
Flight #8 from Y to R    Cost: 450 TL
Flight #11 from R to Z   Cost: 600 TL
Total Cost .......... 1550 TL

Flight #1 from P to W    Cost: 100 TL
Flight #7 from W to Y    Cost: 400 TL
Flight #10 from Y to Z   Cost: 550 TL
Total Cost .......... 1050 TL

Flight #9 from P to R    Cost: 500 TL
Flight #11 from R to Z   Cost: 600 TL
Total Cost .......... 1100 TL

A least cost path from P to Z is P -> W -> Y -> Z and its cost is 1050 TL

Request is to fly from Q to X:
Flight #3 from Q to X    Cost: 200 TL
Total Cost .......... 200 TL

A least cost path from Q to X is Q -> X and its cost is 200 TL
```

Below is an example code segment for reading from text files. You can modify it according to the specific format of the text files in this assignment.

```
//Declare variables
string fileName = "test.txt";
string text;
ifstream inputFile;
...
//Open the stream for the input file
inputFile.open( fileName.c_str(), ios_base::in );
...
//Continue until the end of the file
while ( inputFile.eof() == false ) {
    //Read until a comma
    getline( inputFile, text, ',' );
    //Read until the end of the line
    getline( inputFile, text, '\n' );
}
...
//Close the input file stream
inputFile.close();
```

IMPORTANT NOTES:

Do not start your homework before reading these notes!!!

Output message for each operation should match the format shown in the output of the example code.

NOTES ABOUT IMPLEMENTATION:

1. You MUST use the nonrecursive solution using a stack to implement the search algorithm for finding the paths between the cities as discussed in the class. You will get no points if you use any other algorithm for the solution of the search problem. You can use other data structures to help with your implementation but the main search algorithm must be implemented using a stack.

2. You MUST use your own implementation of a stack in this assignment. In other words, you cannot use any existing stack code from any other source. However, you can adapt the stack codes in the Carrano book or in our lecture slides.

3. You are NOT ALLOWED to modify the given parts of the class definition. Note that you can define additional public and private member functions and data members in the given class. You can also define additional classes in your solution.

4. You are NOT ALLOWED to use any global variables or any global functions.

5. Your code MUST NOT have any memory leaks. You will lose points if you have memory leaks in your program even though the outputs of the operations are correct.

6. Make sure that each file that you submit (each and every file in the zip archive) contains your name, section, and student number at the top as comments.

NOTES ABOUT SUBMISSION:

1. This assignment is due by 23:59 on December 28, 2020. This homework will be graded by your TA Aydamir Mirzayev (aydamir.mirzayev at bilkent edu tr). Please direct your homework related questions to him.

2. You MUST have separate interface and implementation files (i.e., separate `.h` and `.cpp` files) for your classes. We will test your implementation by writing our own driver `.cpp` file which will include your header file. For this reason, your class' name MUST be "`FlightMap`" and your files' name MUST be "`FlightMap.h`" and "`FlightMap.cpp`". You should upload these two files (and any additional files if you wrote additional classes in your solution) as a single zip file. In this zip file, there should not be any file containing the main function. The name of this zip file should be `secX_Firstname_Lastname_StudentID.zip` where `X` is your section number. The submissions that do not obey these rules will not be graded. We also recommend you to write your own driver file to test each of your functions. However, you MUST NOT submit this test code (we will use our own test code). In other words, your submitted code should not include any `main` function.

3. No hard copy submission is needed. The standard rules about late homework submissions apply. Please see the course web page for further discussion of the late homework policy as well as academic integrity.

4. You are free to write your programs in any environment (you may use either Linux or Windows). Yet, we will test your programs on "dijkstra.ug.bcc.bilkent.edu.tr" and we will expect your programs to compile and run on the dijkstra machine. If we could not get your program properly work on the dijkstra machine, you would lose a considerable amount of points. Therefore, we recommend you to make sure that your program compiles and properly works on dijkstra.ug.bcc.bilkent.edu.tr before submitting your assignment.