CS 201, Fall 2022
Homework Assignment 4

Due: 23:59, December 23, 2022

In this homework, you will work on the *HPAir* problem that we studied in the class, which is also explained in Section 6.4 in your text book. 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. A sample graph is shown in Figure 1. Some ideas about the implementation and some detailed pseudocodes are provided in Section 6.4 and on pages 222–225 in your text book.

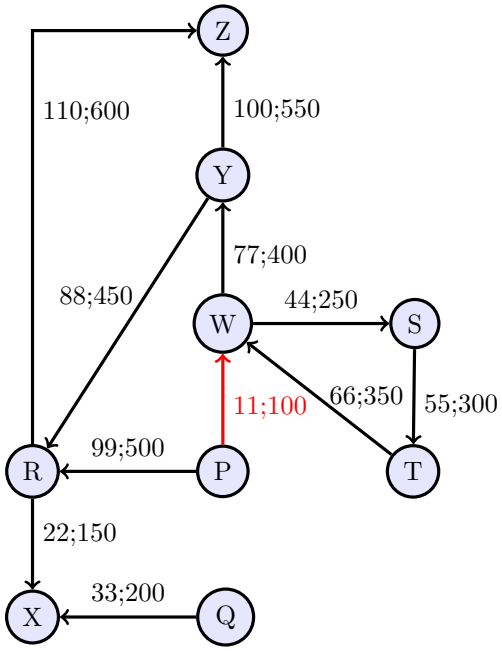

Figure 1: Sample flight graph.

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 "11" and its cost is "100". The graph must be stored by using adjacency lists as described in the Carrano book. 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` → `R` → `Z`", and "`P` → `W` → `Y` → `Z`", together with their corresponding costs as 1100, 1550, and 1050, respectively.

We will provide you with two files: `cityFile` and `flightFile`. 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
T
Z
W
Y
X
S
```

City names can have any number of letters and can include spaces (e.g., New York). They can be given in an arbitrary order (i.e., not necessarily in alphabetical order). You can assume that the city names are unique.

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 semi-column. The file for the example graph is:

```
R;X;22;150
Q;X;33;200
T;W;66;350
W;S;44;250
Y;R;88;450
P;W;11;100
S;T;55;300
Y;Z;100;550
P;R;99;500
W;Y;77;400
R;Z;110;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. You can assume that there is at most a single edge between a particular pair of cities and can also assume that the flight ids are unique.

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;
};
```

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 edges must be stored by using adjacency lists as described in the Carrano book. The flight graph will not change once the `FlightMap` object is constructed by using the information provided in the given files. You can also assume that there will be some cities and some flights in the input files (i.e., the files will not be empty) and the file contents are correct (e.g., the cities in the flight file exist in the city file).

`displayAllCities:` Displays all cities in the flight graph. The city names must be displayed in ascending alphabetical order.

`displayAdjacentCities:` Displays the cities adjacent to the given city (i.e., all cities that have a flight originating from the given city). The adjacent city names must be displayed in ascending alphabetical order.

`displayFlightMap:` Displays all adjacent cities for all cities in the graph. In other words, this function displays the whole flight graph. All city names (origin cities and their adjacent cities) must be displayed in ascending alphabetical order.

`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 message. If there are multiple paths, they must be displayed in ascending order of total costs. You can ignore the scenario where there are multiple paths with the same total cost.

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. Your implementation should use the format given in the example output to display the messages expected as the result of the defined functions.

Example test code:

```
#include "FlightMap.h"

int main() {

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

    fm.displayAllCities();
    cout << endl;

    fm.displayAdjacentCities( "W" );
    cout << endl;

    fm.displayAdjacentCities( "X" );
    cout << endl;

    fm.displayFlightMap();
    cout << endl;

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

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

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

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

    return 0;
}
```

Output of the example test code:

```
9 cities and 11 flights have been loaded

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 -> S, Y,

The cities adjacent to X are:
X ->

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

Request is to fly from W to Z:
Flight #77 from W to Y, Cost: 400 TL
Flight #100 from Y to Z, Cost: 550 TL
Total Cost: 950 TL
Flight #77 from W to Y, Cost: 400 TL
Flight #88 from Y to R, Cost: 450 TL
Flight #110 from R to Z, Cost: 600 TL
Total Cost: 1450 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 #100 from Y to Z, Cost: 550 TL
Total Cost: 550 TL
Flight #88 from Y to R, Cost: 450 TL
Flight #110 from R to Z, Cost: 600 TL
Total Cost: 1050 TL

Request is to fly from P to Z:
Flight #11 from P to W, Cost: 100 TL
Flight #77 from W to Y, Cost: 400 TL
Flight #100 from Y to Z, Cost: 550 TL
Total Cost: 1050 TL
Flight #99 from P to R, Cost: 500 TL
Flight #110 from R to Z, Cost: 600 TL
Total Cost: 1100 TL
Flight #11 from P to W, Cost: 100 TL
Flight #77 from W to Y, Cost: 400 TL
Flight #88 from Y to R, Cost: 450 TL
Flight #110 from R to Z, Cost: 600 TL
Total Cost: 1550 TL
```

IMPORTANT NOTES:
Do not start your homework before reading these notes!!!

NOTES ABOUT IMPLEMENTATION:

1. You ARE NOT ALLOWED to modify the given parts of the header file. 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. In this assignment, you MUST use the data structures and related functions in the C++ standard template library (STL). In particular, you must use `stack` as described above. You can use additional containers such as `vector`, `list`, `map`, etc., as well. Some suggestions are as follows:

   - If you have an internal integer id for each city, you may think about storing the collection of cities as `vector< string >` so that the name of the city with a given id can be obtained from the vector.
   - If you want to obtain the id of a city given its name, you may think about storing the city and id pairs as `map< string, int >` so that the id of the city with a given name can be obtained from the map.
   - The flight graph must be stored using adjacency lists. For a given city, you may think about storing the list of adjacent cities in a list object and store all of these list objects in a vector as `vector< list< T > >` where `T` is the data type that contains information about a particular flight.
   - STL provides `sort` functions as part of some containers (e.g., `list`) and as part of the `algorithm` library.

3. You ARE NOT ALLOWED to use any global variables or any global functions (except the functions in STL).

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

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. To detect memory leaks, you may want to use Valgrind which is available at `http://valgrind.org`.

6. Otherwise stated in the description, you may assume that the inputs for the functions and the file contents are always valid (e.g., city names are unique, there is at most one flight between a pair of cities, etc.).

NOTES ABOUT SUBMISSION:

1. In this assignment, you must have separate interface and implementation files (i.e., separate `.h` and `.cpp` files) for your class. Your class name MUST BE `FlightMap` and your file names MUST BE `FlightMap.h` and `FlightMap.cpp`. Note that you may write additional class(es) in your solution.

2. The code (`main` function) given above is just an example. We will test your implementation using different scenarios, which will contain different function calls. Thus, do not test your implementation only by using this example code. We recommend you to write your own driver files to make extra tests. However, you MUST NOT submit these test codes (we will use our own test code). In other words, do not submit a file that contains a function called `main`.

3. You should put all of your `.h` and `.cpp` files into a folder and zip the folder (in this zip file, there should not be any file containing a `main` function). The name of this zip file should conform to the following name convention: secX-Firstname-Lastname-StudentID.zip where X is your section number. The submissions that do not obey these rules will not be graded.

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

5. You are free to write your programs in any environment (you may use Linux, Windows, MacOS, etc.). On the other hand, 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. Thus, we recommend you to make sure that your program compiles and properly works on dijkstra.ug.bcc.bilkent.edu.tr before submitting your assignment.

6. This assignment is due by 23:59 on Friday, December 23, 2022. You should upload your work to Moodle before the deadline. No hardcopy submission is needed. The standard rules about late homework submissions apply. Please see the course home page for further discussion of the late homework policy.

7. We use an automated tool as well as manual inspection to check your submissions against plagiarism. Please see the course home page for further discussion of academic integrity and the honor code for programming courses in our department.

8. This homework will be graded by your TA Utku Gülgeç (utku.gulgec at bilkent.edu.tr). Thus, you may ask your homework related questions directly to him. There will also be a forum on Moodle for questions.