ECE 2574 ♦ Supplementary Notes for Homework 3

This project involves completing the following tasks:

1. Read the input file and gather a list of cities, a list of all known flights and their costs.
2. Find a route from a given origin to a given destination by systematically exploring the flight-map of cities, beginning at the origin. This exploration is done using a depth first search of the flight-map, and can be implemented using the Stack ADT, as explained in this document.
3. Write the route found, along with total cost, to the output file.

This document discusses the following topics:

1. A Stack ADT implementation
2. A Mapclass implementation which uses the Stack ADT

1. The Stack ADT

Conceptually, a stack has a simple structure—a data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack; the only element that can be removed is the element that was at the top of the stack. Consequently, a stack is said to have "first in last out" behavior (or "last in, first out"). The first item added to a stack will be the last item removed from a stack.

So where do stacks come into play? As you've already seen, stacks are a useful way to organize our thoughts about how functions are called. In fact, the "call stack" is the term used for the list of functions either executing or waiting for other functions to return.

In this project, you need to design your own Stack ADT class. Let us say we use the struct below to store each of the flights we retrieve from the input file:

```
#pragma once

struct flight {
     char City; //destination
     char parentCity; //origin
     int price;
};
```

For the purposes of the graph search, we need to push nodes onto a stack, and then search the flights going out of `City` (stored as the struct above).

As a side-note, the preprocessor directive "`#pragma once`" causes a header to be included only once in a compilation. This prevents the same variables, functions, or classes from being defined more than once. For example, if a header file is included both in a source file (a cpp file) as well as in some other header file, this directive will retain only one copy.

The header file for the Stack ADT that you need to implement may look similar to the example given below. This is for your reference only; your implementation may look *different* from this:

```
typedef char StackItemType;

class stackP
{
public:
      stackP(void);
      stackP( const stackP &toCopy );

      ~stackP(void);

      bool isEmpty() const;

      void push(const StackItemType& newItem);

      void pop();

      void pop(StackItemType& stackTop);

      // peek() returns the contents of the top-most element
      StackItemType peek();

      int getCount();

      // If you need to keep track of the size of the stack,
      // use getCount()

      // May need more methods to support stack operations.
      ...
      ...


private:

      struct stackNode {
            StackItemType item;
            stackNode *next;
      };

      stackNode *top;
      int count;
};
```

For example, the stack class above may be used as follows:

```
int main(void) {
      ...
      stackP stack1;
      stack1.push('A');
      ...
}
```

A stack makes it easy to implement the idea of a first-come first-serve system. For example, for the graph search problem with an origin node A and a destination D, one way to proceed with the search is as follows. If A has 2 children B and C – we would like to search all nodes connected downstream from B, and then move to C. In effect, we want to serve B and all its children, grandchildren, etc. before moving on to C. To achieve this, we push B and C onto a stack. We then pop the topmost element in the stack, say B, and add each of B's downstream nodes onto the stack one-by-one. If we do this repeatedly, we will have a node from a deeper level in the graph at the top of the stack in each iteration. If no outgoing edges are found for an element, we simply pop the element. For each node that is pushed, we check if it is directly connected to the destination D by waiting for D to be pushed onto the stack

2. Mapclass

The `Mapclass` is used to encapsulate all the information associated with a flightmap, including which cities are involved and the flights between the cities. The Mapclass class also contains the methods and data structures for performing the depth-first search mentioned above.

Mapclass retrieves flights from the file and stores them in a linked list, where they may be accessed easily. It also holds a list of all cities occurring in the input file. The example Mapclass header file given below uses a linked list of cities (pointed to by `cityHead`) and flights (pointed to by `fpHead`). Cities and flights may be added into these linked lists using public methods. Mapclass may also contain a `CurrentFlightPath` stack which is a stack containing cities which are included in the path from `currentCity`. Other methods, such as the search method to find a route from an origin city to a destination city, are also part of the `Mapclass`.

The following example header called `mapclass.h` is for your reference only. Your class may be more complex or simpler. You will need to develop your own algorithm for determining a flight route to each city that can be visited from the origin city and the associated cost.

```
#pragma once
#include "stackP.h"
#include <iostream>
#include <iomanip>

using namespace std;

struct cityNode {
          char cityName;
          bool visited;
          bool isOrigin;
          cityNode *next;
     };

struct fpNode {
          flight nodeFlight;
          fpNode *next;
};

class mapclass
{
```

```
public:

      //linked list of all cities in flightmap
      cityNode *cityHead;

      //linked list of all flights in flightmap
      fpNode *fpHead;

      //stack which holds flights during flightmap search
      stackP currentFlightPath;

      mapclass(void);

      ~mapclass(void);

      int getRoutePrice();

      // Add a Flight to the list of all flights
      void addFlight( char start, char end, int price );

      // Add a City to the list of all cities
      void addCity( char cityName, bool isOrigin );

      // Determine whether a city has been already visited
      bool visitedCity( char cityName );

      // Marks city as visited
      void markVisited( char cityName, bool visited );

      void backTrack();

      void setCurrentCity( char cityName )

      // Prepares the class to be used for another search.
      void clear();

      // Find a flight path to a destination from currentCity
      bool search( char destination );

private:

      char currentCity;
      int currentPrice;
      fpNode *fpCurrent;
      cityNode *cityCurrent;
};
```

The two methods, `search()` and `getRoutePrice()`, use algorithms that may be more complex compared to those of other methods. Pseudo-code for the `search()` and the `getRoutePrice()` methods are given below. These are examples; your implementation may use different algorithms.

```
/* The city being considered in a particular iteration is called
CCity. The list of all Flights is called FlightList. Let desired
destination be Dest. */
```

search(Dest)

```
//search() searches the flightmap iteratively to find a path to Dest
from CurrentCity

CCity = CurrentCity; //Initialization

Stack.push(CCity);

while stack.getCount() > 0:
      topcity = stack.peek();
      CCity = topcity;
      Mark CCity as visited;
      If CCity is Dest, terminate;
      For each flight in FlightList:
            If flight.origin is CCity:
                  If flight.destination is unmarked:
                        stack.push(flight.destination);
                        break;
      If no push occurs in this iteration:
            stack.pop();

//search Ends
```

getRoutePrice()

```
TotalCost = 0;
topcity = stack.peek(); //Initialization
stack.pop();
top2city = stack.peek(); //Initialization

while stack.getCount() > 1:
      topcity = stack.peek();
      stack.pop();
      top2city = stack.peek();
      For each flight in FlightList:
            if flight.destination == topcity && flight.origin ==
            topcity:
                  CCity = flight.origin;
                  TotalCost = TotalCost + flight.price;
                  break;


return TotalCost;
```

3. Correct Output for testfile1.txt

For your reference, the correct output for testfile1.txt (which is posted in Scholar) is given below. The GTA will use testfile1, testfile2, testfile3 and some other test files to check the correctness of your code. It is suggested that you create your own test files to verify your code.

Destination	Flight Route from A	Total Cost
B	A,B	$100
C	A,B,C	$300
G	A,B,C,G	$550
D	A,B,C,G,D	$750
H	A,B,C,G,H	$850

Again, be sure to follow the output formats given, as well as the naming convention specified in the homework specification document.