CSE 241 Algorithms and Data Structures	Spring Semester, 2006

Lab 4

April 4, 2006 *Due Date: Tuesday April 18 (early date: April 11)*

This lab will bring together several of the topics we have studied this semester to find the route (i.e. sequence of flights) from a given airport at a desired arrival time to a desired destination airport in the shortest amount of time. Here's Some sample output from Part 3.

```
Options: Compute Shortest Routes(1), Show Route (2), Exit(3): 1
Source Airport Code: BOS
Arrival Time at Airport:  1100
AM or PM (A or P):A

Options: Compute Shortest Routes(1), Show Route (2), Exit(3): 2
Destination Airport Code: HOU
Elapsed time since BOS airport arrival to HOU arrival is 7 hours and 45 minutes.
Itinerary:
        TW 53 (BOS 1203 PM --> STL 223 PM)
        WN 759 (STL 400 PM --> HOU 545 PM)

Options: Compute Shortest Routes(1), Show Route (2), Exit(3): 2
Destination Airport Code: SFO
Elapsed time since BOS airport arrival to SFO arrival is 9 hours and 36 minutes.
Itinerary:
        TW 53 (BOS 1203 PM --> STL 223 PM)
        TW 183 (STL 325 PM --> SFO 536 PM)

Options: Compute Shortest Routes(1), Show Route (2), Exit(3): 2
Destination Airport Code: JFK
Elapsed time since BOS airport arrival to JFK arrival is 5 hours and 13 minutes.
Itinerary:
        DL 1865 (BOS 245 PM --> JFK 413 PM)
```

To keep on track, everyone should complete Part 1 by the early due date. I'd recommend that you try to complete Parts 1 and 2 by the early due date. The material needed for Part 3 will not be covered in full depth until April 11th. The expectation is you will wait until then to start Part 3.

Part 1: Binary Heap (40 pts)

Design and implement a BinaryHeap Class. Your binary heap must have a `Tracker` as an inner class. My recommendation is that each `Tracker` object has a single instance variable that is an integer holding the index in the array where the tracked element is held. I also strongly recommend that you write a `private swap` method that swaps two elements in the binary heap (and only move elements using this method). By doing this all of the maintenance required to keep the instance variable within the `Tracker` correct can be done within this method.

Here are the methods that you are to implement. The methods `size` and `isEmpty` should run in constant time. All other methods should run in $O(\log n)$ time where n is the number of elements currently stored in the heap.

- There should be two constructors provided. One constructor should takes no parameters and initializes the array storing the binary heap to some default size. The second constructor should take a single parameter (an integer) which indicates the number of elements that are to be held in the binary heap.

- `BinaryHeap.Tracker put(double key, Object data)` takes as input a key (of type `double`) and a reference to the associated data (of type `Object` in Java or using a `Template` in C++). It should insert the given item in the priority queue returning a `Tracker` to it. (If you prefer you are welcome to make the key an object that implements comparable and then use the comparator. As another option you could instead support an `add` method that takes as a single parameter a comparable object. If you make either of these changes then you'll need to make appropriate changes in the constructor and some of the later methods. If you have any questions about this option, just ask.)

- `int size()` returns the number of items in the priority queue
- `boolean isEmpty()` returns true iff the priority queue is holding no items.
- `double minimumKey()` returns the value of the minimum key.
- `Object extractMin()` removes the item with the minimum key and returns its associated data.
- `toString` (or in C++ overload the << operator) which takes no parameters and returns a string (or `ostream` in C++). So that we can tell the structure of the binary heap, please output the key and associated data for the array elements of your binary heap representation in the order they appear in the array. You do not need your program to show the binary heap as a tree but you may find this helpful to do by hand so you can check your output.

The `BinaryHeap.Tracker` method should support the following methods. The `inHeap` method should run in constant time and the `decreaseKey` method should run in logarithmic time. If desired, you can also include any other methods that you'd like to have.

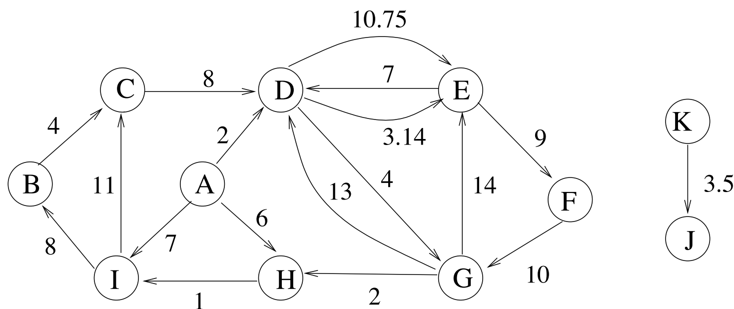
- `boolean inHeap()` should return true iff the tracked element is still in the priority queue.
- `boolean decreaseKey(double newKey)` takes as input a new key. Let x be the binary heap currently tracked by the tracker on which this method is called. If x's key is smaller than `newKey` then it does nothing and returns `false` (indicating the key was not changed). Otherwise, it decreases the value of x's key to `newKey` (fixing the binary heap and tracker as appropriate) and returns `true`.

A driver called `BinaryHeapTester` (`.java` and `.cc`) is provided. *You must turn in your output from this when you submit Part 1.* You may modify `BinaryHeapTester` if your interface is slightly different than above – just briefly describe the modifications made in your write-up. You don't need to provide the code.

Part 2: Weighted Directed Graph (20 pts)

Design and implement a Weighted Directed Graph Class that allows multiple edges between vertices (which in the application in Part 3 will be airline flights between cities). You should implement a `Vertex` class that will hold the information associated with each vertex and an `Edge` class that will hold the information associated with each Edge. Here are the basic methods that I recommend you have for each class – add others as needed. I think their names are self explanatory. If you have any questions, please ask. (For those using C++ replace the `toString` methods by overloading the << operator.)

A driver called `GraphTester` (`.java` and `.cc`) is provided that creates the graph shown below. *You must turn in your output from this when you submit Part 2.* You may modify this driver if your interface is slightly different than above.



For the `Edge` class you need just store the head, tail and the edge weight.

- constructor: `Edge(Vertex head, Vertex tail, double weight)`
- public methods: `Vertex getTail()`, `Vertex getHead()`, `double getWeight()`, `String toString()`

I recommend that you include an instance variable in your `Vertex` class that is a head pointer to the list of outgoing edges. You are welcome to use a list class in the Java or STL libraries in which case this instance variable will just be a list. Observe, that each element in this list is an edge (as opposed to the vertex where the edge goes). A little more will need to be added for Part 3, but you can do that after you get Part 2 working.

- constructor `Vertex(String name)` where `name` is what will be used by the `toString` method.
- `String toString()`

Finally for the graph class you most likely will want instance variables for the number of vertices, number of edges, and a list of vertices in the graph. (The edges will be held in the adjacency lists referenced by the vertex objects). I recommend the following methods for the graph class:

- a constructor that initializes the graph to have no vertices and no edges (i.e. an empty graph).
- public methods: `void addVertex(Vertex v)`, `void addEdge(Edge e)`, `int numberOfVertices()`, `int numberOfEdges()`
- `Iterator outEdges(Vertex v)` - return an iterator that goes through all of the outgoing edges from v (in an arbitrary order). If you use a Java LinkedList (or ArrayList) for your adjacency list then you can just use the provided `iterator()` method.
- `Iterator vertexIterator()` - returns an iterator over all of the vertices in the graph (in an arbitrary order).
- `String toString()` - To be sure that both of your above iterators work. For the `toString` method you should use `vertexIterator` to iterate over all of the vertices. Then for each vertex v use `outEdges(v)` to iterator over v's outgoing edges printing out each one (giving the head, tail, and weight in some easy to read form).

Part 3: Fastest Route Application (40 pts)

Design and Implement a Variant of Dijkstra's Shortest Path Algorithm for the application described on the first page. Believe it or not, you are really almost done – your graph and binary heap classes do all the work. You just need to make the right calls to them :-). I'd like everyone to spend some time thinking about how you modify Dijkstra's algorithm for this problem and so won't give the details here. I will go over it in class on April 11th.

Let me state the problem once again. You are given an list of all flights between a set of airports. The user gives a departure airport and a time when he/she would like to arrive at the airport. The user will also gives a destination airport. The task is to find a sequence of flights that will get the user to the destination airport at the earliest time *with the restriction of a minimum layover of one hour at any intermediate airports and also the flight out of the departure airport must be at least one hour after the stated arrival time*. Your output should include all of the information given in the sample output on page 1 but can format it differently if you desire as long as it is at least as easy to read. There are two data files: `airport-data.txt` and `flight-data.txt` both of which contain actual airline schedules (with over 3500 flights) from 1992 collected by Roberto Tamassia from EasySABRE.

The file `airport-data.txt` first contains the number of cities (an integer). Then for each city there is a line with two items (separated by a tab). The first is a 3 letter airport code. The second is an offset from GMT. The data from which I constructed these two data files also contained the x and y coordinate of the airport and the name of the city/airport but I removed those. If anyone would like to see the original data, it is `full-airplane-data.txt`. However, you don't need to use this for your lab. I recommend that you create an `Airport` class that extends `Vertex`. This can hold the GMT conversion that is provided in the data file and in general could hold the other data associated with an airport.

The file `flight-data.txt` contains one line per flight that is tab delimited with the following 8 items: airline, flight number, code for source airport, local departure time, A or P (for am or pm) for departure time, code for destination airport, local arrival time, A or P for arrival time. The original data set also had fields related to the fare class, number of stops, and more. I recommend that you create a `Flight` class that extends `Edge` that can hold the above information and will have the needed accessors.

The only further change is that you might want to add some methods to your Vertex/Airport class or create a class used for the flight scheduling application that performs some needed computation. The variables you need to store for each vertex/airport are: the tracker for calling `decreaseKey`, the edge from its parent, and the cost to reaching it from the source (which is used as the key in the priority queue). Also, you will need a way to access an `Airport` when given its three letter code since that is what the user will input. Just use a mapping for this (either from Lab 2 or simply use the Java provided HashMap since that is all you need) – the key is the three letter code and the associated data is the `Airport`. You can add any other instance variables that you feel would be useful for your implementation.

For testing this part, you should make a simple driver that provides the following two options (plus whatever others you want for helping debug). First the user should be able to give a source airport and a desired arrival time at the airport and have the best route (i.e. sequence of flights) computed between that airport and all others. The second option should allow the user to provide a destination airport and give the best route computed from the source airport (based on the last

call made to the first option). The driver will also need to read in the data. To help reduce the time you spend here, code fragments to read each of the data files and store the fields in local variables (that you can then decide how to use) are provided on the course home page under labs. You will be given (in the "What to Submit" section) some specific routes to compute. Your output should include all of the information show on page 1. For debugging purposes you might want to make a small version of the data set so that you can trace what is happening.

To help reduce the time you need to spend on converting times into the same time zone and computing flight lengths and layover lengths, on the course page under labs, I'm providing a class `GMTtime` with the following methods:

- constructor: `GMTtime(int localTime, int gmtOffset, boolean am)`
- `int minutesSince(GMTtime time)` which returns the number of minutes (which could be greater than 60) that have elapsed from `time` until `this`. This will properly deal with times going across AM and PM. (The flights have no dates. They are all less than 24 hours.)
- `String toString()` returns a string with the times in the form they were in the data file except it uses "AM" and "PM" versus just "A" and "P".

For your convenience the constructor computes the value of several instance variables that you can use in other methods if needed. These include the local time, the local time in a 24 hour clock (called `localMilitaryTime`), the GMT offset, local hours, minutes (between 0 and 59), the GMT time, and final the GMT time converted completely to minutes. For example 2:30AM would be 150 minutes into the day.

What to Submit (read this carefully)

For this lab you are expected to submit the following. Please include these items in the given order so that the TAs can find everything easily. On the web page under labs are a list of things that we expect (such as commented out code is removed, proper indentation is used, meaningful variable names commented if there would be any doubt as to their purpose, output is labeled and easy to read, a comment is put at the top of each method so it's easy to find, output is checked for correctness and readability, ...) in what you submit. The TAs will deduct if you do not follow these guidelines.

- A completed and signed cover sheet with your name written legibly on the top.
- A write-up that describes your basic design with the focus being on variations/additions from what is described above. **If there were any problems with your implementation (e.g. wrong output was obtained, a run-time error occurred) then clearly indicate that in your write-up and give as much information as you can as to what you think is causing the problem.** If you followed exactly what was described and everything worked then you may not need to put anything here. DO NOT describe anything already described in the lab or in class.
- Part 1 when submitted should be clearly labeled and include: your binary heap code and the output from `BinaryHeapTester`. It is your job to check that the output is correct and if it is not to clearly mark any problems you noticed. You do NOT need to submit `BinaryHeapTester` – if you made any changes to it just summarize those in your write-up.
- Part 2 when submitted should be clearly labeled and include: all code that is used for your graph representation (e.g. the Graph, Vertex, and Edge classes) as well as your output form `GraphTester`. It is your job to check that the output is correct and if it is not to clearly mark any problems you noticed. You do NOT need to submit `GraphTester` – if you made any changes to it just summarize those in your write-up.
- Part 3 when submitted should be clearly labeled and include: any code you wrote that was not included in either Part 1 or Part2. This includes your driver. Finally, you should provide the output from the following:
  - With a source city of STL and desired airport arrival of 7AM show the fastest route to ABQ and PVD.
  - With a source city of PHX and desired airport arrival of 6PM show the fastest route to PVD, DEN and STL.

Although you should not submit the output for the queries shown on page 1 of this assignment, I would recommend you use them to help test that your program is working. If your output does not match that shown on page 1 (and lands in the desired city later than what is shown) then there is a problem. If you cannot fix the problem then submit this output to help illustrate what is happening. Finally, for all of your submitted output you should check that the elapsed time given from the arrival at the source airport to the arrival at the destination airport is correct. Don't forgot to account for different time zones. You can tell from the GMT offset which time zone each airport is in.