

Lab 4

November 15, 2007

Due Date: Tuesday December 4 (early date: November 27)

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 provided start time to a desired destination airport in the shortest amount of time.

The user will give (1) a departure airport, (2) a start time when he/she can be at the departure airport, (3) 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. Furthermore, the flight out of the departure airport must be at least one hour after the stated start time.* Here's some sample output:

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

Your output should include all of the information given above, but can format it differently if you desire as long as it is at least as easy to read. To complete this lab you will need to implement a tagged binary heap that supports trackers, construct a graph representing the structure of the flights among the airports, and implement a variant of Dijkstra's shortest path algorithm to find the best route. Unless explicitly stated otherwise, you must do everything "from scratch."

The files `airport-data.txt` and `flight-data.txt` contain airline schedules (with over 3500 flights) from 1992 collected by Roberto Tamassia from EasySABRE. The file `airport-data.txt` starts with 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 Greenwich mean time (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.

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.

Part 1: Tagged Binary Heap (40 pts)

For the algorithm that will find the best airline route, each airport is associated with a tag which corresponds to the elapsed time in the best route discovered so far from the starting airport to that airport. It will be important to be able to find, and extract, the airport with the smallest tag. You could achieve this goal using a “standard” tagged binary heap that supports `max` and `extractMax` by providing a comparator that defines the priority so that e_1 is greater than e_2 when the tag for e_1 is smaller than the tag for e_2 . (You can find an implementation of such a “ReverseComparator” at the bottom of page 53 of the text.)

To help ensure you understand the binary heap data structure, instead you must directly implement a “min-oriented” tagged binary heap in which the representation property is that for an element, its tag, is no larger than the tag of any of its descendants. Also, you will directly implement a tagged binary heap, as opposed to creating a tagged binary heap by wrapping a binary heap to insert tagged elements. Finally, the application will need to maintain a tracker for each airport through which it will change (in particular, lower) the priority.

The advantage of the approach used in the text is that it is very general and extensible. However, there is a small amount of time overhead that occurs when one class wraps another, versus a direct implementation. One of the goals of this lab is to be sure that you really understand both the mechanism to support trackers, and also the mechanism to create a tagged collection. Thus you will directly support both of these functions.

In your implementation of a “min-oriented” `TaggedBinaryHeap` Class you can only use Java’s built-in array. (This means you cannot use the `TrackedArray` class from the book, but you are welcome to look at it.) So as part of your tagged binary heap you must directly provide the support for the trackers. My recommendation is that your tagged binary heap implementation have an inner class for the tracker, and that each tracker instance has a single instance variable that is an integer holding the index in the array where the element is held. Each element itself should have a tag (of some comparable type `T`) and some associated data (of some arbitrary type `E`).

I strongly recommend that you write an internal `swap` method that swaps the elements at two specified indices of the array (and only move elements using this method). By following this suggestion, all of the maintenance required to maintain the trackers can be done within this swap method.

You are to implement the following methods for your tagged binary heap. The constructors, `size`, `isEmpty`, and `min` methods should run in constant time. The `ensureCapacity`, and `toString` methods will take linear time. All other methods should run in logarithmic time.

`TaggedBinaryHeap()`: Creates a new empty tagged binary heap with a default capacity of 8.

`TaggedBinaryHeap(int capacity)`: Creates a new empty tagged binary heap with the provided capacity.

`void ensureCapacity(int capacity)`: Increases the capacity of this collection to the given value.

`int getSize()`: Returns the number of elements in this collection.

`boolean isEmpty()`: Returns `true` if this collection contains no elements, and otherwise returns `false`.

`String toString()`: Returns a string that describes each element in the collection. To help us grade Part 1, please show the elements in the order that they appear in the underlying array in the form `tag -> data`. 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.

`void put(T tag, E element)`: This method creates a new element with the given tag and associated value, and inserts this element into this collection. A `RuntimeException` is thrown if the collection is already at capacity.

`TaggedBinaryHeap.Tracker putTracked(T tag, E element)`: This method creates a new element with the given tag and associated value, and inserts it into this collection. A tagged binary heap tracker that tracks the new element is returned. A `RuntimeException` is thrown if the collection is already at capacity.

`E min()`: Returns a lowest priority element. It throws a `NoSuchElementException` when this collection is empty.

`E extractMin()`: Removes and returns a lowest priority element. It throws a `NoSuchElementException` when this collection is empty.

For the tracker inner class you must support the following methods where `inCollection` and `get` run in constant time, and the others run in logarithmic time.

`boolean inCollection()`: Returns `true` if and only if the tracker is at an element that is still in the collection.

`E get()`: Returns the element associated with this tracker. Note that the element associated with the locator might no longer be in the collection. If desired, the `inCollection` method can be used to determine if a tracked element is currently in the collection.

`void remove()`: Removes the element associated with this tracker. If the element is no longer in the collection, a `NoSuchElementException` is thrown.

`void update(T newTag)`: Replaces the tag for this element with the given tag (which might be less than or greater than the current tag value).

Part 2: Airport and Flight Classes (15 pts)

In this part you will create the `Airport` and `Flight` classes (and thus implicitly set up an adjacency list representation of the directed weighted multigraph defined by the flights among the airports). Think about what you would like to use for the weight associated with each flight. In your implementation of the `Flight` class, you may use whatever positional collection data structure (from the book or Java's libraries) you would like to use for the list of flights that originate in a given city.

The `toString` method for the `Airport` class should use the three letter acronym, and the `toString` method for the `Flight` class should produce output like: `TW 53 (BOS 1203 PM --> STL 223 PM)`.

To test this class, you should check that you can iterate over all the flights originating from each city. You will probably want to create a subset of the flights to test your implementation. If you are going to submit Part 2 and Part 3 together there is no need for you to provide any output. Otherwise, convince us (with no more than 2 pages of output) that your implementation is correct.

Part 3: Fastest Route Application (45 pts)

In this part you will implement and apply a variation of Dijkstra's shortest path algorithm for the application described on the first page. (Dijkstra's algorithm will be covered in class on Tuesday 11/13.) You may use the `Mapping` class (or Java's `HashMap`) for this part (e.g., to map from a three letter acronym to the associated airport object).

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 the start time. Using these you should compute the best route between that airport and all others. The second option should allow the user to provide a destination airport, and provide 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 are provided on the course home page under labs. 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, is a `GMTtime` class with the following methods:

`GMTtime(int localTime, int gmtOffset, boolean am)`: Creates a new `GMTtime` object with the specified values.

`int minutesSince(GMTtime time)`: Returns the number of minutes (which could be greater than 60) that have elapsed from `time` until `this`. As an example if `t1` is 8:15am and `t2` is 8:50am (in the same time zone), then `t2.minutesSince(t1)` would return 35. This method will properly deal with times crossing day boundaries.

`String toString()`: Returns a string with this time.

For your convenience the constructor computes the value of several instance variables. 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 finally the GMT time converted completely to minutes. For example 2:30AM would be 150 minutes into the day. If desired, you can add public accessors for any of these variables.

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.

- A completed and signed cover sheet with your name written legibly on the top.
- If there were any problems with your implementation (e.g., wrong output was obtained, a runtime 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.
- 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.
- Part 2 when submitted should include your `Airport` and `Flight` classes. If you are not also submitting Part 3, provide a short test program (using a small data set) and its output to demonstrate Part 2 is working properly.
- Part 3 should include your driver and all the code you wrote for the flight scheduling application. You should provide the output from the following queries:
 - 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 forget to account for different time zones. You can tell from the GMT offset, the time zone each airport is in.