CSE 241 Algorithms and Data Structures **Lab 3a**

A Priority Queue

Assigned: 10/31/2012 *Due Date: 11/28/2012*

Important Note: Lab 3 is divided into two parts. You are now reading Part (a), which is worth 75% of the total credit for the lab. Part (b), which is worth another 75%, comes with a separate handout and code repository. So, Lab 3 will weight 1.5 times higher than Lab 1 or Lab 2. It also offers an opportunity for earning extra credits (see Part (b) of Lab 3).

Both parts of Lab 3 are formally due at the same time. However, because this lab requires two significant pieces of implementation (heaps plus Dijkstra's algorithm), we have also set up the autograder so that you may test a *preliminary* version of Part (a). We will not record a grade for this preliminary code; your grade will be assigned based on what you turn in on Wedesday, 11/28. Note that **you will need a working heap to complete Part (b)**.

Finally, please note that this is the boring part of the lab, but it may be your first implemented sorting algorithm (with a small addition of code) as a programmer; the more interesting bit comes in Part (b).

1 Implement a Min-First Priority Queue (75%)

Implement a template/generic class `PriorityQueue<T>` based on the binary heap data structure described in class. The queue maintains a collection of records of type `T`, each associated with an integer-valued key. Your queue should be *min-first*; that is, the smallest element goes on top of the heap. Your priority queue must hold not only keys but the associated records as well. This shouldn't require any massive re-engineering: when you would ordinarily move a key, just move the associated record along with it.

Your class must support the usual priority queue methods, plus **decreaseKey**. To make use of **decreaseKey**, you'll need to implement a `Handle` data type (See Section 1.3 below) that you can use to find an element in the priority queue at any time.

1.1 Interface Specification

Your `PriorityQueue<T>` class should support the following methods:

- a constructor that creates an empty priority queue
- `boolean isEmpty()` – return true iff the queue contains no elements.
- `Handle insert(int key, T value)` – insert the key *key* into the queue, along with the associated object *value*. Return a `Handle` object that refers to the pair (*key*, *value*); the handle will be used by **decreaseKey** to find the pair later.
- `int min()` – return the smallest key in the queue.
- `T extractMin()` – remove the (*key*, *value*) pair with the smallest key from the queue and return its *value* component.
- `boolean decreaseKey(Handle h, int newkey)` – attempt to decrease the key of the pair referenced by the `Handle` *h* to the value "newkey." If *h* refers to a (*key*, *value*) pair that has been removed from the priority queue, or if newkey is $\geq$ than the pair's current key, do not modify the queue, and return `false` to indicate that nothing has changed. Otherwise, replace the pair's key with newkey, fix up the heap, and return `true`.

- `int handleGetKey(Handle h)` – return the *key* component of the pair pointed to by the handle h. The result is undefined if h refers to a pair no longer in the queue.
- `T handleGetValue(Handle h)` – return the *value* component of the pair pointed to by the handle h. The result is undefined if h refers to a pair no longer in the queue.
- `String toString()` – print a list of pairs of the form "(*key*, *value*)" for every element in your priority queue. Please print the pairs starting with cell 1 of the array and going up to the highest-numbered cell. Use the `toString` method of each *value* object to print it.

C++ users will need to modify the above specification slightly. The most important change is that records are passed explicitly by reference (Java does this implicity); hence, you should substitute `T*` wherever you see `T` above. Similarly, you'll want to use `Handle *`'s instead of `Handle`s. The other change is that you should implement an output operator `<<` for your heap instead of a `toString` method. Because C++ objects don't have a default `operator<<` function, you'll need to implement such a function for any type you use to parameterize your priority queue.

Your `insert`, `extractMin`, and `decreaseKey` methods should all run in worst-case time $O(\log n)$, where n is the heap size. The `isEmpty`, `min`, `handleGetKey`, and `handleGetValue` methods should all run in constant time. Your heap should be able to hold arbitrarily many elements; you should implement size doubling for the heap (either explicitly or via a language-supplied resizable array type) to meet this requirement.

Note: the above design is not very clean because `Handle` should really be an opaque type that can return its key and value, rather than just asking the heap to do it. The reason I'm not asking for the nicer design is that the internals of the `Handle` would have to be exposed to your `PriorityQueue` but hidden from the rest of the world. You can do this using, e.g., the `friend` declaration in C++ or the package facility in Java, but it's a detail that you shouldn't have to worry about right now.

1.2 What to Turn In for a Not-Graded Test

We have provided a "unit test" program in the Lab3a driver file to help you test your heap before you actually need to use it for something important. This file contains three little procedures designed to exercise your `PriorityQueue` class. The autograder will check your output against the correct result, though the first two procedures create small enough heaps that you should be able to validate the correctness of the output by hand. For example, if you build your heap for a series of n random numbers, you should be able to use the heap to sort the numbers by consecutively deleting n elements from the heap. Note that your heap may not look exactly the same as ours, but the order of the extracted keys and the contents associated with each handle should still be the same.

As usual, if you check in your code, a correctly modified control file, and a README describing anything interesting you noticed in implementing your queue, and a brief summary of how you implemented Handles.

1.3 Advice on Implementing Heaps with Keys, Values, and Handles

A `Handle` is just a piece of information that lets you find where a given key currently appears in the heap. For an array-based heap, it suffices to maintain an integer index that points to the cell containing the key. You'll need to modify the index stored in a `Handle` from inside your `PriorityQueue` class whenever it changes, that is, whenever the key referenced by the `Handle` moves up or down the heap. You should also *invalidate* a handle when its key is deleted from the heap, so that it can be recognized as invalid if used in a later call to `decreaseKey`.

Any of your methods that modify the heap must be able to move a (*key*, *value*) pair from one heap node to another and update the associated `Handle`. You'll probably want to keep track of key, value, and handle together in each heap node. To make sure that all three items always get moved together, I suggest implementing a **swap** method that takes two heap nodes, exchanges all their data at once, and updates their `Handles`. If you use *only* this method to move data around in the heap, you won't have to worry about copying all the right fields. You can implement all the heap rearrangements we talked about in class in terms of your **swap** operation.