Harvey Mudd College
Computer Science 42
Fall 2009

Assignment 7
Priority Queue Interface, Implementations, and Factory
Due. 11:59 p.m., Wed., 11 November 2009

The concept of a *priority queue* is useful in many computational problems. For example, it can be used to speed up Dijkstra's algorithm. It can also be used in a variety of simulation problems and as a means of implementing an efficient in-place sorting algorithm.

In this assignment, we want to use Java first to define the *interface* for a priority queue, then provide a couple of *implementations* of priority queue, and, finally, a priority queue factory, to illustrate the *factory-method* pattern from software engineering.

Priority Queue Concept

A priority queue is a variation on the queue concept. Whereas a queue is *first-in*, *first-out*, a priority queue provides instead a different functionality, one based on the type of items being stored in an essential way: *minimum-out*. The assumption is that the data items to be stored in the priority queue have a linear ordering, in the sense that any two items can be compared to see if one is *less than* the other. The ordering criterion is obvious for classes such as numbers and strings, but can also be defined for lists and other classes in many cases.

1. Define an Interface

Define a Java interface for priority queues. Here are the required methods for a PriorityQueue containing elements of class *Type*:

`boolean nonEmpty();`	Return true if the PriorityQueue is non-empty, otherwise false.
`void insert(Type item);`	Insert an item into the PriorityQueue.
`Type removeMin();`	Remove the smallest item, as defined by a Comparator.
`int size();`	Return the current number of elements on the PriorityQueue.

A *Comparator* is a standard Java interface. An object of this class must implement the method compare, of the type shown in the sample implementation below. You may copy this definition and use it, as it will be required with the tests that are going to be provided. (An implementation of Comparator can also provide method *equals*, but we shall not require it.)

```
import java.util.Comparator;

public class IntegerComparator implements Comparator
{

public final int compare(Object a, Object b)
  {
    Integer aa = (Integer)a;
    Integer bb = (Integer)b;

    return aa < bb ? -1 : aa > bb ? 1 : 0;
  }
}
```

2. Implement SimplePriorityQueue class

By *simple*, we mean an unsophisticated, not necessarily very fast, priority queue. It should be *growable*. It is recommended that you use a `Vector` for this purpose. Insert elements into the `PriorityQueue` by adding them to the `Vector` using `addElement`. Find the minimum by scanning through the elements. Use the method `removeElementAt` to remove the minimum element.

3. Test the SimplePriorityQueue class

I will provide unit test code on the course web page. You should get your simple queue to pass the tests before proceeding to the next implementation.

4. Implement the HeapPriorityQueue class

The *heap* data structure will be described in class on Monday. Using it, your enqueue and removeMin operations should run in logarithmic time. This is a significant savings over the simple queue when a large number of items need to be stored. Note: We want to implement this *from scratch*, using class Vector to store the elements.

5. Test the HeapPriorityQueue class

Virtually the same test code will be used as for the simple queue.

6. Implement a PriorityQueueFactory

This is a separate class having one method. When called with either “simple” or “heap”, it should return a PriorityQueue of the named type:

```
PriorityQueueFactory factory = new PriorityQueueFactory<Integer>();
factory.createPriorityQueue("simple", size, new IntegerComparator());
```

5. Test the PriorityQueueFactory class