

Assignment #5—Priority Queues

Due: Monday, February 23

In this assignment, you will have the chance to implement a collection class called a *priority queue*, which stores its elements in an order determined by a numeric priority value. As in conventional English usage, lower numeric values correspond to higher levels of urgency, so that a task with priority 1 comes before a task with priority 2. This interpretation, unfortunately, can cause confusion with a phrase like “A has higher priority than B” because A will in that case have a *lower* numeric priority. To avoid ambiguity, this handout uses the terminology “A is more urgent than B” to indicate that A should come earlier in the priority queue.

As in the various lecture discussions of how to implement the collection classes from Chapter 5, this assignment asks you to implement the `PriorityQueue` class using three different underlying representations:

1. A dynamic array that stores the elements in priority order
2. A linked list that stores the elements in priority order
3. A data structure called a *heap* that stores the elements in a way that guarantees that the fundamental queue operations run in $O(\log N)$ time

The Priority Queue assignment is designed to accomplish the following objectives:

- To give you the opportunity to write your own collection class
- To allow you to practice working with linked lists and pointers
- To let you experiment with multiple implementations of a particular abstraction

The methods in the `PriorityQueue` class

For each of the implementations, the `PriorityQueue` class is exported through the `pqueue.h` interface. The operations you need to implement are always the same; what changes is the internal representation, which is of little concern to the client. The methods exported by the `PriorityQueue` class appear in Figure 1 on the next page. As you can see from the table, the methods in the `PriorityQueue` class are similar to those in the `Queue` class from Chapter 5. The only differences are (1) the `enqueue` method takes an extra argument indicating the priority of the item and (2) there is a new `peekPriority` method, which returns the priority associated with the most urgent item in the queue.

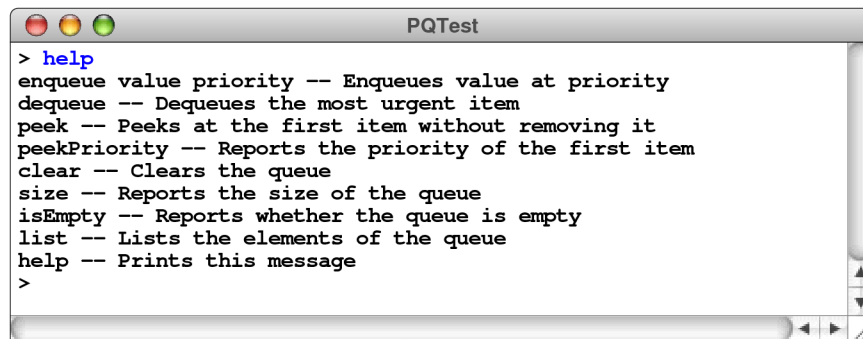
The `PQTest` program

In addition to writing the three versions of the `pqueue.h` interface, you are also responsible for creating an interactive test program by adding the necessary code to the `PQTest.cpp` file. This program should initialize a priority queue object and then manipulate it by reading commands from the user. Each of those commands begins with a command keyword that usually appears alone on the input line. In the case of the

Figure 1. Exported methods in the PriorityQueue class

enqueue (<i>value</i> , <i>priority</i>)	Adds <i>value</i> at the appropriate position in the queue as determined by the numeric value <i>priority</i> . As in conventional English usage, lower numeric values correspond to higher levels of urgency, so that a task with priority 1 comes before a task with priority 2. If the <i>priority</i> parameter is missing, it defaults to priority 1.
dequeue ()	Removes the value at the head of the queue and returns it to the caller. Calling dequeue on an empty queue causes a runtime error.
peek ()	Returns the value of the most urgent item in the queue without removing it. Calling peek on an empty queue causes a runtime error.
peekPriority ()	Returns the priority of the most urgent item in the queue without removing it. Calling peekPriority on an empty queue causes a runtime error.
clear ()	Removes all the elements from a queue.
size ()	Returns the number of elements currently on the queue.
isEmpty ()	Returns true if the queue is empty.

enqueue command, the keyword is followed by two additional values: the value to be added to the priority queue (a **string**) and its associated priority value (a **double**). One of the commands that you need to implement is the **help** command, which produces a list of the available commands, as follows:



```

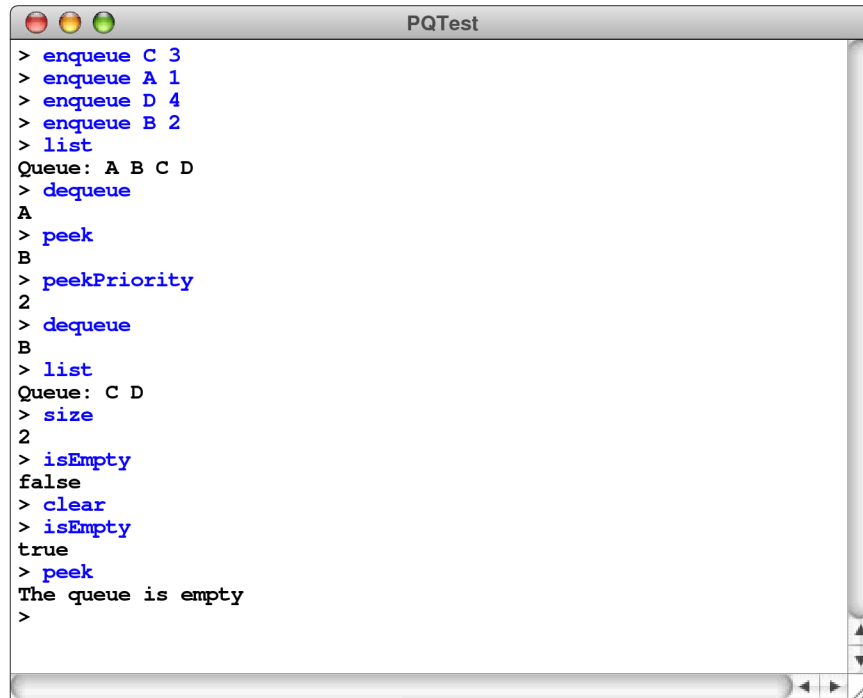
> help
enqueue value priority -- Enqueues value at priority
dequeue -- Dequeues the most urgent item
peek -- Peeks at the first item without removing it
peekPriority -- Reports the priority of the first item
clear -- Clears the queue
size -- Reports the size of the queue
isEmpty -- Reports whether the queue is empty
list -- Lists the elements of the queue
help -- Prints this message
>

```

With the exception of the **list** and **help** commands, each of these command keywords corresponds to one of the methods in the **PriorityQueue** class. Your test program should implement those commands by calling the corresponding method and then displaying the result, if any.

The **list** command displays the values in priority order, which is extremely useful during testing. It is, however, slightly tricky to write because the code for the test program operates as a client of the **queue.h** interface and therefore has no access to its internal data structures. You must therefore implement the **list** command using only the priority queue functions listed in Figure 1 and still make sure that the queue contains the same values in the same order when the **list** command has finished. In particular, you are not allowed to add additional methods to the **PriorityQueue** class to make it easier to implement **list**.

As an illustration of how the test program works, the following sample run illustrates the use of the various commands:



```

> enqueue C 3
> enqueue A 1
> enqueue D 4
> enqueue B 2
> list
Queue: A B C D
> dequeue
A
> peek
B
> peekPriority
2
> dequeue
B
> list
Queue: C D
> size
2
> isEmpty
false
> clear
> isEmpty
true
> peek
The queue is empty
>

```

The first four lines enqueue the strings "C", "A", "D", and "B", with priority values that cause them to appear in alphabetical order, as indicated by the `list` command on the following line. The last line of the sample run illustrates error checking. In this case, your implementation of the `peek` command must check to make sure that the queue is not empty before it tries to read the first element.

The following sections describe each of the three implementations, but do so very briefly, without all of the diagrams you might expect to be part of this assignment. One factor that makes it possible to save a bit of paper here is that the textbook already contains many diagrams of dynamic arrays, linked lists, and the heap data structure, which means that you can look for the necessary examples are there. More importantly, the assignment demo on the web site includes code to draw diagrams showing the internal structure of the queue for all three implementations. If you want to know how your queue should appear after a particular set of commands, you can simply run the demo.

Implementation 1. Dynamic array

The first version of the priority queue abstraction uses a dynamic array as its underlying representation. This implementation is by far the easiest, partly because we've given you much of the code in the starter project. The supplied code includes the entire class definition, along with the implementations of `expandCapacity`, the copy constructor, and the assignment operator. All you have to do is fill in the implementations of the methods that implement the priority queue behavior in the context of the dynamic array model.

The only methods in the array-based implementation that are at all tricky are **enqueue** and **dequeue**. The **enqueue** method has to search through the array to find the appropriate position to insert an item at the specified priority. Once you find the correct position, you then have to move the remaining elements forward one position to make room for the new arrival. The **dequeue** method returns the value of the initial entry in the queue, but must then shift the elements in the array to fill up that space. Both methods therefore operate in $O(N)$ time. The point of this part of the exercise is to let you develop your test program in a relative simple environment. You should make sure that you get your test program working perfectly with the array-based queue before moving on to the other implementations.

Implementation 2. Linked list

For the second version of the priority queue interface, you need to replace the array from the first implementation with a linked list that stores the values and their priorities. Once again, the **enqueue** method requires linear time to find the correct insertion point, but can then add the new value in constant time; the **dequeue** method runs in $O(1)$ time.

Implementation 3. The heap data structure

The final version of the priority queue implementation uses a data structure called a *heap*, which is discussed in detail in section 16.5 in the text, which begins on page 719 of the textbook. The advantage of the heap implementation is that both **enqueue** and **dequeue** require $O(\log N)$ time.

When you use the demo program with the heap implementation, the application waits for you to click the mouse for each step in the process of adjusting the heap to preserve the ordering property. You should also note that the array at the bottom of the heap-mode display will not necessarily show the array elements in their precise order. The elements will, however, always be stored so that the heap property is maintained. In other words, every child will contain a higher (i.e., less urgent) priority number than its parent, but the two children of a node can appear in either order.

Possible extensions

There are several extensions that you might try if you get done early with the required parts of the assignment, including the following:

- *Make the queue a template class.* The files for the assignment assume that the type stored in a queue is a string. Add templates to allow for any value type.
- *Make the implementation const-correct.* Except for the assignment operator and copy constructor, the `PriorityQueue` class does not use the `const` keyword to show what remains unchanged by each operation. Rewrite the class so that it includes any `const` markers required to make the package obey the rules for const-correctness.
- *Improve the efficiency of the list-based queue.* In practice, many applications tend to enqueue items in order so that new items often belong at the end of the list. Change the linked-list implementation so that adding elements at the end is $O(1)$.
- *Add graphics.* Use the graphics libraries to illustrate the priority queue operation in the way the demo program does.
- *Create an interesting priority-queue application.* Use your imagination.