

C++ Priority Queue

Due Sunday April 13, 2014

In this assignment, you will create a priority queue. You should construct a correct, robust, and elegant data structure. You may want to write a small program to test it, but the product will be just the header file and the implementation file for `priorityqueue62` named, naturally enough, `priorityqueue62.h` and `priorityqueue62.cpp`. This will also give you some practice in reading and using documentation.

Getting started

Begin by reading about assertions in C++. Their purpose is the same as assertions in Java, and the notation is not very different. See pages 159–160 in Weiss’s book or look in the C library pages on the C++ reference site (i.e. “`cassert`”).

Then read about the STL classes `vector` and `map` and the `pair` class (which you can find by doing a search in the reference pages).

Priority queue

We want to build a priority queue that stores pairs of integers. Each pair consists of a unique key and a priority. The priority should be a non-negative integer. The priority queue will allow the user to insert pairs, to remove the pair with the lowest priority, and to reduce the priority of an existing pair in the priority queue. Please use `pair` from the Standard Template Library (STL) to store the pair of integers in the priority queue. Also you must use the `assert` statement to check all preconditions.

The public members, described below, are required:

`priorityqueue62()` The default constructor creates an empty priority queue.

`~priorityqueue62()` The destructor must match the implementation and recycle all dynamically-created objects. In our case, we don’t create any dynamically-created objects, so this should be very simple. Just put

```
priorityqueue62::~~priorityqueue62(){};
```

in your `.cpp` file and it should work.

`void push(int key, int priority)` We adopt the push-pop language of the STL’s `priority_queue`. This function inserts a new element with a given key and priority. The key must not already appear in the queue, and the priority must be non-negative.

`void pop()` This function removes the element with smallest priority. No value is returned. It is an error to pop from an empty priority queue.

`int top_priority() const` In the STL `priority_queue`, the function `top()` returns the element of least priority. Instead of returning a pair, we have two “top” functions that return integers. This one gives the least *priority* value.

`int top_key() const` This “top” function gives the *key* of the element of least priority.

`void reduce_priority(int key, int newpriority)` This function lowers the priority of a specified element. It is an error if the key is not present in the priority queue, or if the new priority is negative or is greater than the current priority.

`int get_priority(int key)` This function returns the priority of the element with the specified key. It returns `-1` if the key is not present in the priority queue. Technically, we should make this method `const`, since it does not modify the priority queue, however, this will make life easier for you.

`bool is_empty() const` This function tells us if the priority queue is empty.

`bool is_present(int key) const` This function returns true if there is an element in the queue with the specified key.

`void clear()` This is a straightforward method that empties the priority queue.

`int size() const` This is a function that returns the number of elements in the queue.

The supplied header file suggests an implementation that uses a heap-ordered vector like that in the class `VectorHeap` from Bailey’s structure library (see the source code on the handouts page). The only additional complication is that this implementation must be able to locate a particular *key* in the heap. An additional data structure, a `map`, from keys to heap indices, will do the trick. Some private member variables are included and some private methods headers that may be useful to define. If you don’t want to use these, that’s fine, just change the header file.

You may choose a different implementation. If you do, you will have to change the private part of the header.

Whatever your implementation, **all of the functions must work correctly in logarithmic time**. In other words, linear search for a key across a vector is not permitted. As stated in the STL documentation, the `map` operations are guaranteed to run in logarithmic time.

Submission

Submit the files `priorityqueue62.h` and `priorityqueue62.cpp`. Since we are no longer exporting from Eclipse and creating jar files, just make a directory called “Assignment10_LastNameFirstName” and put your files in there and copy the directory to the dropbox as before. This will be the approach we will take for all of our C++ submissions.

Even though we’re writing in C++, you must continue to comment your code as before using the *Javadoc* style of commenting.

Grading

criterion	points
Functionality of priority queue	10
<code>assert</code> error checking	4
Appropriate comments (including JavaDoc)	2
Style and formatting	2
General correctness	2
submitted correctly	1

Some hints

- Try and follow the outline from our existing binary heap.
- As always, try and code incrementally, testing and compiling as you go.
- Use `cout` to help you debug (but remove these when you submit your final version).
- The one challenge with this assignment beyond the C++ part, is keeping the `map` (i.e. hashtable) up to date when you change the heap. Think about exactly what you are storing in the `map` to make methods efficient.
- If you have a `pair` object, you can simply access the two elements of that object using `.first` and `.second`. Look at the examples in the documentation.