Programming Assignment 4: A Priority Queue Module

Due Date: Wed, Nov 15 by 11:59PM

Resources:

Lecture Notes on Priority Queues (mostly week 10).
Section 5.9 of Aho/Ullman (starting at page 271):
http://i.stanford.edu/~ullman/focs/ch05.pdf

Note: this is a "mini" programming assignment worth approximately ⅔ the points of other programming assignments.

In this assignment you will implement a generalized priority queue module using binary heaps.

> **BIG-PICTURE**: The basic priority-queue ADT and its heap implementation we have studied so far keep track of only priorities.
>
> A useful generalization is to associate a unique "ID" with each entry in addition to its priority queue. For example, an ID might represent a student and the associated priority might be that student's GPA.
>
> This assignment: You will such a generalized priority queue.
>
> Preview: a priority queue with these features will be useful in implementation of Dijkstra's shortest paths algorithm.

Header File: The interface (ADT) is specified in the given file **pq.h** (which you will not be able to change). The header file contains banner comments describing the behavior and requirements of each function you must implement. The header file is reproduced at the end of this document for reference.

Nothing Pre-Implemented: You will write the complete implementation in a file **pq.c** almost ***completely from scratch*** -- the given pq.c only has a placeholder for the C struct you must design to encapsulate a priority queue.

Overview:

- Again, the most fundamental changes relate to **"IDs"**:
  - **IDs**: each entry in the priority queue will have both a priority (a double) and a *unique* integer ID (up to now we have only had priorities).
  - **ID values**: each ID is a non-negative integer in the range {0..N-1} where N is the capacity of the priority queue (the capacity is fixed upon creation -- see notes on capacity below). You are already familiar with this idea of a unique integer ID -- think "buzzer numbers."
- Additional features:
  - **Support of both Min and Max-Heaps**: The user can specify if they want a min-heap or a max-heap via the min_heap flag passed to **pq_create().**

    As a result, instead of a **pq_delete_min** or **pq_delete_max** function, we have a more generic name which makes sense in both cases: **pq_delete_top()**.
  - See comments for descriptions and requirements of these additional functions (runtime requirements given here for reference):
    - pq_change_priority: O(log N)
    - pq_remove_by_id: O(1)
    - pq_get_priority: O(1)

Complete List of Functions (see pq.h for details):

pq_create
pq_free
pq_insert
pq_change_priority
pq_remove_by_id
pq_get_priority
pq_delete_top
pq_peek_top
pq_size
pq_capacity

```
/**
* General description:  priority queue which stores pairs
*   <id, priority>.  Top of queue is determined by priority
*   (min or max depending on configuration).
*
*   There can be only one (or zero) entry for a particular id.
*
*   Capacity is fixed on creation.
*
*   IDs are integers in the range [0..N-1] where N is the capacity
*   of the priority queue set on creation.  Any values outside this
*   range are not valid IDs.
**/

// "Opaque type" -- definition of pq_struct hidden in pq.c
typedef struct pq_struct PQ;

/**
* Function: pq_create
* Parameters: capacity - self-explanatory
*             min_heap - if 1 (really non-zero), then it is a
min-heap
*                        if 0, then a max-heap
*
* Returns:  Pointer to empty priority queue with given capacity and
*           min/max behavior as specified.
*
*/
extern PQ * pq_create(int capacity, int min_heap);

/**
* Function: pq_free
* Parameters: PQ * pq
* Returns: --
* Desc: deallocates all memory associated with passed priority
*       queue.
*
*/
extern void pq_free(PQ * pq);
```

```c
/**
* Function: pq_insert
* Parameters: priority queue pq
*             id of entry to insert
*             priority of entry to insert
* Returns: 1 on success; 0 on failure.
*          fails if id is out of range or
*            there is already an entry for id
*          succeeds otherwise.
*
* Desc: self-explanatory
*
* Runtime:  O(log n)
*
*/
extern int pq_insert(PQ * pq, int id, double priority);

/**
* Function: pq_change_priority
* Parameters: priority queue ptr pq
*             element id
*             new_priority
* Returns: 1 on success; 0 on failure
* Desc: If there is an entry for the given id, its associated
*       priority is changed to new_priority and the data
*       structure is modified accordingly.
*       Otherwise, it is a failure (id not in pq or out of range)
* Runtime:  O(log n)
*
*/
extern int pq_change_priority(PQ * pq, int id, double new_priority);

/**
* Function: pq_remove_by_id
* Parameters: priority queue pq,
*             element id
* Returns: 1 on success; 0 on failure
* Desc: if there is an entry associated with the given id, it is
*       removed from the priority queue.
*       Otherwise the data structure is unchanged and 0 is returned.
* Runtime:  O(log n)
*
*/
extern int pq_remove_by_id(PQ * pq, int id);
```

```c
/**
* Function: pq_get_priority
* Parameters: priority queue pq
*             element id
*             double pointer priority ("out" param)
* Returns: 1 on success; 0 on failure
* Desc: if there is an entry for given id, *priority is assigned
*       the associated priority and 1 is returned.
*       Otherwise 0 is returned and *priority has no meaning.
* Runtime:  O(1)
*/
extern int pq_get_priority(PQ * pq, int id, double *priority);

/**
* Function: pq_delete_top
* Parameters: priority queue pq
*             int pointers id and priority ("out" parameters)
* Returns: 1 on success; 0 on failure (empty priority queue)
* Desc: if queue is non-empty the "top" element is deleted and
*       its id and priority are stored in *id and *priority;
*       The "top" element will be either min or max (wrt priority)
*       depending on how the priority queue was configured.
*
*       If queue is empty, 0 is returned.
*
* Runtime:  O(log n)
*
*/
extern int pq_delete_top(PQ * pq, int *id, double *priority);

/**
* Function: pq_peek_top
* Parameters: priority queue pq
*             int pointers id and priority ("out" parameters)
* Returns: 1 on success; 0 on failure (empty priority queue)
* Desc: if queue is non-empty information about the "top"
*       element (id and priority) is stored in *id and *priority;
*       The "top" element will be either min or max (wrt priority)
*       depending on how the priority queue was configured.
*
*       The priority queue itself is unchanged (contrast with
*       pq_delete_top).!
*
*       If queue is empty, 0 is returned.
*
* Runtime:  O(1)
*/
extern int pq_peek_top(PQ * pq, int *id, double *priority);
```

```c
/**
* Function:  pq_capacity
* Parameters: priority queue pq
* Returns: capacity of priority queue (as set on creation)
* Desc: see "returns"
*
* Runtime:   O(1)
*
*/
extern int pq_capacity(PQ * pq);

/**
* Function: pq_size
* Parameters: priority queue pq
* Returns: number of elements currently in queue
* Desc: see above
*
* Runtime:  O(1)
*/
extern int pq_size(PQ * pq);
```

Tip:

Think about the relevant information associated with an entry:

Of course, you have:

its ID and
its priority

What else is important about an entry? Remember, you are implementing a binary heap solution. So... the current position/index of the entry in the heap array (equivalently, its position in the tree) seems important...

Readme File:

To make grading more straightforward (and to make you explain how you achieved the assignment goals), you must also submit a Readme file.

The directory containing this handout also contains a template Readme file which you should complete (it is organized as a sequence of questions for you to answer).

Submission:

You will submit an archive file containing both your source code file pq.c and your readme file. Using the given pq.h file, we should be able to compile your submission into a .o by doing this:

```
gcc -c pq.c
```

Note: pq.c should not contain a main function. You will certainly have developed one or more main drivers to test your implementation of the priority queue. But these should be in other “client” files (which you do not have to submit) or, if you wrote a tester/driver in pq.c, just be sure to comment it out or disable it in some way.

Aside: a handy trick in C is to use the preprocessor to enable/disable chunks of code -- like a main function. Here is an example:

```
// file:  util.c
/** lots of functions ***/



/** END OF UTILITY FUNCTIONS **/

#ifdef DEV

// driver program exercising above functions
int main(){

  // blah blah


}

#endif
```

`$ gcc -DDEV util.c`	This will behave the same as if we inserted `#define DEV` At the top of the source file and produce an executable.
`$ gcc -c util.c`	This on the other hand would compile as if the main function was not there at all -- producing a .o file to which other programs can link.

This is kind of nice because you get two behaviors without changing the source code.

(You could even have multiple main functions each enabled by a different preprocessor symbol. By the way, there is nothing magic about “DEV” -- I just chose it).

With respect to this assignment, you would be “safe” if you employed

```
this strategy -- we will be able to compile your code into a .o; link
our own driver/test to that .o and run our tests.
```