15-122: Principles of Imperative Computation — Spring 2018

Lab 11: PQ puns are too hard — 20 March

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with your neighbors as you work through this lab!

Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab-pq .
% cd lab-pq
```

You should add your code to the existing files `pqsort.c1` and `pqmedian.c1` in the directory `lab-pq`.

Grading: Finish at least task (2.a) or (2.b) for credit, and additionally finish (3.a) for extra credit. We want you to think about the last problem. Once your code passes the tests, show it to a TA. Make sure you have reasonable contracts!

Generic priority queues

In this lab, we'll use an implementation of generic priority queues. We implemented priority queues as heaps in class.

```
/************************/
/*** Client interface ***/
/************************/
typedef void* elem;

// f(x,y) returns true if e1 is STRICTLY higher priority than e2
typedef bool has_higher_priority_fn(elem e1, elem e2);

/************************/
/*** Library interface ***/
/************************/
// typedef ______* pq_t;

bool pq_empty(pq_t P)
  /*@requires P != NULL; @*/ ;

pq_t pq_new(has_higher_priority_fn* prior)
  /*@requires prior != NULL; @*/
  /*@ensures \result != NULL && pq_empty(\result); @*/ ;

void pq_add(pq_t P, elem e)
  /*@requires P != NULL; @*/ ;

elem pq_rem(pq_t P)
  /*@requires P != NULL && !pq_empty(P); @*/ ;

int pq_size(pq_t H)
  /*@requires H != NULL; @*/ ;
```

Unlike the priority queues from class, these priority queues are unbounded (in particular, there is no `pq_full` function). Note that the interface provides the function `pq_size(H)`, which returns the number of elements in priority queue `H`. The code you see here is from `pq.c1`, which you'll need to compile along with your code for this assignment. While the priority queue interface we give you is, in fact, implemented with the heap data structure (with the unbounded array trick to be used so that they don't get full), you should respect the interface and not rely on this assumption for anything except efficiency.

1pt

Sorting using priority queues

In this part, you'll use the priority queue interface to sort an array. Your solution should work with any implementation of priority queues. If priority queues are implemented as heaps the way we did in class, sorting the array should have worst-case complexity in $O(n \log n)$.

Hint: *Most of the work you have to do is in correctly instantiating the client interface.* Don't re-implement the heap data structure, and don't re-implement a sort like mergesort or quicksort. Do respect the interface of priority queues.

(2.a) In the file `pqsort.c1`, use priority queues to finish the implementation of `sort_by_word`, which takes an array of frequency information structs and sorts them by ASCII-betically by the words.

(2.b) In the file `pqsort.c1`, use priority queues to finish the implementation of `sort_by_count`, which takes an array of frequency information structs and sorts them by frequency.

Compile and test your code by running this command:

```
% cc0 -d -x pq.c1 pqsort.c1
```

The first set of outputs should be sorted by word, and the second set of outputs should be sorted by frequency.

2pt

Finding the median

We know how to return the element with highest priority out of a priority queue. Now, let's find the element with the k-th highest priority (if k is 1, it returns the element of **highest priority**).

(3.a) In file `pqmedian.c1`, complete the implementation of the function `k_priority(H,k)` that returns the k-th priority element in priority queue `H`. On return, `H` should contain the same elements as when the function was called.

The *median* of a collection H of elements is the element m in H so that half of the other elements of H are larger than or equal to m and the other half is smaller or equal to m. If H contains an even number of elements, this definition is ambiguous since it asks us to take "half" of an odd number of elements. In this case, we will (inaccurately) let the "smaller half" have one element more than the "larger half".[1]

(3.b) Also in file `pqmedian.c1`, complete the implementation of the function `median(H)` that returns the median element in priority queue `H`. On return, `H` should contain the same elements as when the function was called. Hint: the function `pq_size` may come handy.

Compile and test your code by running this command:

```
% cc0 -d -x pq.c1 pqmedian.c1
```

3pt

[1] You can find the actual definition of median online. Why can't we use it in this exercise?