15-122: Principles of Imperative Computation

Lab A: PQ puns are too hard

Tom Cortina, Rob Simmons

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 `pqstack.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.

```
1  /************************/
2  /*** Client interface ***/
3  /************************/
4  // typedef _______ elem;
5  typedef void* elem;
6
7  // f(x,y) returns true if e1 is STRICTLY higher priority than e2
8  typedef bool higher_priority_fn(elem e1, elem e2);
9
10 /************************/
11 /*** Library interface ***/
12 /************************/
13 // typedef ______* pq_t;
14
15 bool pq_empty(pq_t P)
16   /*@requires P != NULL; @*/ ;
17
18 pq_t pq_new(higher_priority_fn* prior)
19   /*@requires prior != NULL; @*/
20   /*@ensures \result != NULL && pq_empty(\result); @*/ ;
21
22 void pq_add(pq_t P, elem e)
23   /*@requires P != NULL; @*/ ;
24
25 elem pq_rem(pq_t P)
26   /*@requires P != NULL && !pq_empty(P); @*/ ;
```

Unlike the priority queues from class, these priority queues are unbounded. 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.

Sorting using priority queues

In this task, 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, the sort should be 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.

Stacks using priority queues

Stacks can be implemented using a priority queue to hold the data. Each element of the priority queue will then be a stack element along with its "priority".

The stack itself consists of a priority queue (to hold all of the elements) and a `pushcount` field. The `pushcount` field keeps track of how many elements have been pushed on to the stack. In order for your priority queue to act like a stack, you will need to use the `pushcount` field in some way. You will also need to complete the client function that determines which elements have higher priority.

(3.a) Complete the file `pqstack.c1` with your implementation of the stack functions.

Compile and test your code by running this command:

```
% cc0 -d -x heap.c1 pqstack.c1
pop: Hello
pop: there
pop: it
pop: is
pop: good
pop: to
pop: see
pop: you!
```