Assignment 1: A C Queue Module

CS 325

Due Feb. 3, 2012

Introduction

For this project you will write a simple module that does queue manipulation. While this program is perhaps not very interesting in its own right, it will allow you to "brush up" on some of the finer points of C — namely `struct`s, pointers, and dynamic memory allocation.

In coding, be sure to follow good programming practices and consistent conventions — I don't care that your conventions are not my conventions, but your conventions must be both reasonable and consistent. *In particular*, remember to carefully document each module's interface, data structures, and specification (*what* the module does, but not *how* it is implemented).

The files `queue.h` and `queue.c` on the class home page comprise the queue module. `queue.h` is complete. `queue.c` contains stub functions that you will complete. See "Testing" below for what to do with `test.c`.

The Queue Module

This module implements and maintains a number of queues that can be maintained in either FIFO (first-in-first-out) or key order. The queue elements are of type `qnode`:

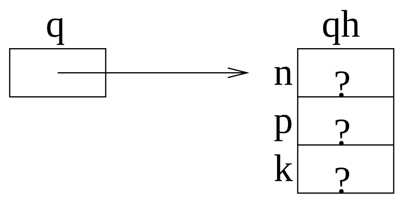
```
typedef struct qnode {
   struct qnode *next;
   struct qnode *prev;
   int key; \\
   /* other things would go here */
} qnode;
```

`qnode` is defined in the header file `queue.h`. A queue is implemented as a doubly-linked list using the `next` and `prev` fields to point to the succeeding and previous elements, respectively. The queue itself is represented by a pointer to a *queue header*, a structure of type `qnode` which is not considered to be an element of the queue. If the queue is empty, the `next` and `prev` fields of the queue header point back to the header itself.

By way of illustration, one would create the queue `q` this way:

```
qnode qh;   /* qh is a qnode. */
qnode *q = &qh;   /* q is a POINTER to a qnode, initialized to point to qh. */
```

Following the initialization of `q`, our memory model looks like this:



Note that the `next`, `prev`, and `key` elements of `qh` are uninitialized. In this context, we typically use `q` to access the members of `qh` rather than `qh` itself. To directly initialize the `key` field to `5` using `qh` we would write:

```
qh.key = 5;
```

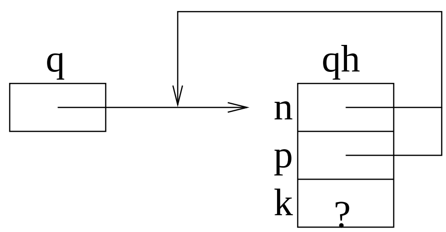
To accomplish this with the pointer `q` we would have to first dereference `q` using the pointer dereference operator (`*`) and then apply the member selection operator:

```
(*q).key = 5;
```

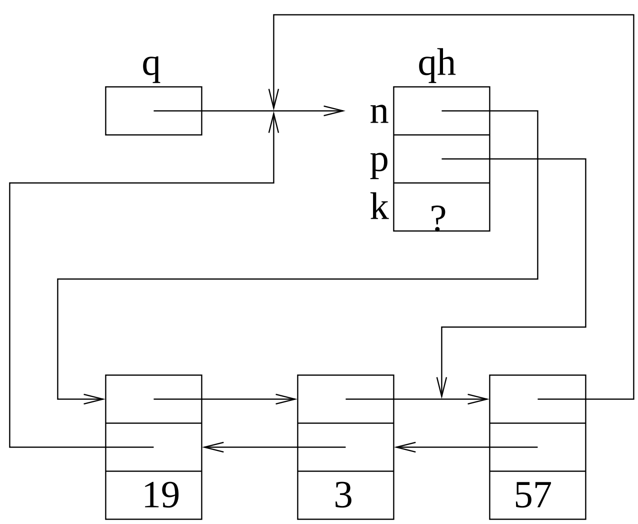
This operator pair occurs so frequently that there is a single operator that handles both of them at once:

```
q->key = 5;
```

An empty queue looks like this:



A queue with three items would look like this:



Recall that a queue is ordered. What is the key value of the first item in the queue? What pointer expression could be used to change the key value `3` in the queue to `33`?

Module Function Descriptions

The module should have the following externally visible functions:

`void initqu(qnode *queue)`

Initialize the queue header pointed to by `queue` to be the empty queue. The header node must already exist.

`int emptyqu(qnode *queue)`

Return 1 if the queue is empty, 0 if it is not.

`void insqu(qnode *queue, qnode *element)`

Insert the element pointed to by `element` into the queue of which `queue` is a member, immediately after the element pointed to by `queue`. This function may be used to insert at either the beginning or the end of a queue. Suppose the queue header is pointed to by a variable named `q`. Then to insert an element `element` at the end of the queue use `insqu(q->prev, element)` and to insert at the beginning of the queue, use `insqu(q, element)`.

`qnode *remqu(qnode *element)`

Remove the queue element pointed to by `element` from whatever queue it is in, returning that element itself. Exception: If the element is the only thing in the queue (i.e., the header of an empty queue), then leave the queue unchanged and return `NULL`. This function may be used to remove from either the beginning or the end of a queue. Suppose the queue header is pointed to by a variable named `q`. Then to remove the first element in the queue use `remqu(q->next)` and to remove the last element, use `remqu(q->prev)`. If you are not careful, you can remove the queue header, effectively destroying the queue.

`void ordinsqu(qnode *queue, qnode *element)`

Insert the element pointed to by `element` into the queue whose header is pointed at by `queue` based on the value in the key field. The queue should be maintained in increasing order by key value; if one or more elements already exist with the key value of the given element, insert the new element after the existing elements.

`qnode *ordremqu(qnode *queue, qnode *val)`

Remove the first element in the queue whose key value is `val` and return it. If the queue is empty or a element with the given value does not exist, return `NULL` and leave the queue unchanged.

Implementation and Testing

Create a new C Project in Eclipse with the CDT perspective or use your favorite C compiler. Add the three files `queue.h`, `queue.c`, and `test.c` to your project. Build the project. It should compile and link with a few warnings (remember — `queue.c` contains stubs). Running it will generate a memory access exception due to a dangling pointer. Complete the stubs in `queue.c`. `test.c` is a test driver module. The test module reports on its progress by posting messages to `stdout`. It will attempt to diagnose errors, but its analysis is primitive.

Hints: None of the functions should require more than a dozen lines of code. `ordinsqu()` and `ordremqu()` may make use of `insqu()` and `remqu()`, respectively.

Project Turn-In

Email `queue.c` as an attachment to `kelliher[at]goucher.edu` by the beginning of class on the 3rd. Late assignments will only be accepted by prior agreement.