Lab: Queues
CSC 161, “Imperative Problem Solving and Data Structures”
Department of Computer Science · Grinnell College
April 26, 2016

The goal for today’s lab is to write an abstract data type (interface and implementation) for queues—data structures of variable size that follow a “first-in, first-out” access discipline (in contrast to the “last-in, first-out” discipline of stacks). Once in the queue, items are “dequeued” one at a time from the front, just as they are “popped” from the top of a stack, but new items are “enqueued” at the rear of a queue, and cannot be accessed again until all of the previously enqueued items have been removed (by dequeuing).

Pairs
- Eli Salm and Anna Blindermann
- Faizaan Ali and Colin Greenman
- Sophie Gaschott and Erhaan Ahmad
- Lex Martin and Mattori Birnbaum
- Cory McCartan and Yuyin Sun
- Jong Hoon Bae and Ella Nicolson
- Dennis Chan and Gemma Nash
- Lilly Webster and Zachary Susag
- Tanner Tufto and Josh Lavin
- Saung Thuya and Ying Zhang
- Jae Hong Shin and Tyler Williams

Background
Like stacks, queues can be implemented in several ways. In the lab on buffers later this week, we’ll look at an implementation that uses an array. Today, we’ll work on an implementation that uses an enclosed linked list.

Most of the functions have straightforward and familiar implementations using linked lists, essentially identical to the corresponding functions for stacks in King’s stackADT3.c implementation.

Two of them are a little more difficult: the function that determines the size of the queue, and the function that adds an item at the rear of the queue.

To find the size of a queue, we could compute the length of the enclosed list of items. It would be faster, though, to have an extra field of the enclosing struct that is initialized to 0 by the constructor, incremented every time an item is added to the queue, and decremented every time an item is removed. In other words, it would be an invariant of the queue type that this length field would always contain the number of items in the queue. Then the function that asks for the size of the queue could simply return the value of this field instead of having to traverse the enclosed list.

To add an item at the end of the queue, we could run a cursor down the enclosed list until cursor->next became NULL. Then we could overwrite cursor->next with a pointer to a newly allocated node containing the item to be added. Once again, the problem with this strategy is that it’s slow because we have to traverse the whole list in order to find the last node.

A better idea would be to have a third field in which we can keep an extra pointer, one that always points to the last node in the list (unless the list is empty, in which case this extra pointer should probably be set to NULL). Again, we could initialize this field in
the constructor and update it as we add items to and delete items from the queue, always maintaining the invariant that it points to the last node.

Exercises

The interface to a Queue type is similar to the interface to a Stack type: a constructor that returns a freshly allocated, empty queue; a destructor that frees a queue when one is finished with it; a function for adding an item at the end of a queue, provided that the queue is not full; a function for removing an item from the front of a queue, provided that it is not empty; a predicate that determines whether a given queue is empty, and another that determines whether a given queue is full.

We'll also want a function that inspects a queue and returns the element at the front without removing it from the queue. (This is essential in a way that the corresponding function for stacks is not, since it cannot be easily simulated by the other operations.)

Finally, it will be useful to have a function that reports the number of items waiting in the queue.

Exercise 000: Write a queues.h file that includes signatures for these functions, along with an “opaque” type definition of the Queue type, defining it as a pointer to a struct type that will be defined only in the implementation file queues.c. Also include a “transparent” type definition for Item, making it a synonym for int.

Exercise 001: Write a queues.c file that defines the struct type to which Queues point with the three fields described above: one for a pointer to the front of the list, one for a pointer to the last node in the list, and one for the size of the queue.

Exercise 010: Implement the constructor and destructor functions for the Queue type. In addition to freeing the storage allocated for the queue, the destructor should have a side effect on the caller: It should replace the caller’s pointer to the deallocated with a null pointer so as not to leave a dangling pointer behind. (Note that this may require an adjustment in the type of the parameter.)

Exercise 011: Implement the emptiness and fullness predicates and the function that determines the number of items in the queue.

Exercise 100: Implement the function that adds an item to the queue, the function that removes an item from a non-empty queue (and returns it) and the function that returns the item at the front of a non-empty queue (without removing it from the queue).

Exercise 101: The file /home/stone/c/code/test-queues.c contains a test program for queues. Copy it into the directory in which you have placed queues.c and queues.h. Edit your copy so that the names of the functions and the order of the arguments in the function calls correspond to the function signatures in queues.h.

Exercise 110: Compile queues.c and test-queues.c and link the resulting .o files into an executable. Run the test program and account for any reported failures. Remedy them by making corrections either in the test program or in your implementation.