CSCI 262 Data Structures

Spring 2019

Lab 9: Queue, part 1

(Back to Labs)

Goals

- Implement a simple data structure (a queue)

Overview

This lab will walk you through the implementation of a simple data structure - a queue. In this lab, you will create a not-very-capable queue; in a future lab, you will update your code to make a fully general queue.

As usual, we ask that you work with a partner or a team. If you end up solo, please raise your hand and we'll pair you with someone else (or make a three-person team if necessary). You are welcome to change partners throughout the semester!

Before you begin

Download lab09.zip, which contains some starter code for your Queue class along with some testing code. Go ahead and build and run the code - **without making any changes to the starter code, you should have a few passing tests, and a lot of failing tests.** This is because you are given a skeleton `Queue` class declaration with *stub* implementations of all of the public methods (which is what we are testing). A *stub* is just an implementation of a method which does nothing useful, but returns the proper type for the method (e.g., a `bool` method might simply contain "`return true;`").

Simple queue implementation

As mentioned above, you are provided with a skeleton Queue class (named `Queue`, and in a file `Queue.h`, capitalized to prevent accidental confusion with the STL `queue` template class). Your job is to finish the implementation of the `Queue` class.

The class you will be writing can only hold elements of type `char`, has a fixed capacity, and has only the

essential `Queue` methods:

- `bool enqueue(char c);`
- `bool dequeue();`
- `char front();`
- `bool is_empty();`

Refer back to the slides for reminders on what these do!

For testing convenience (but a bit more work for you), the `enqueue()` and `dequeue()` methods return `bool` values indicating whether or not the operation was successful. Specifically, the `enqueue()` method will return `false` if the `Queue` is already full (remember, it has a fixed capacity), and true otherwise, while the `dequeue()` method will return false if the `Queue` is already empty, and true otherwise. Note that there is no good way to signal to the user of the `Queue` that there is a problem if they call `front()` on an empty `Queue` - they can just expect to get garbage values in that case (a better implementation might throw an *exception*, something you will study in later courses).

The methods `front()` and `is_empty()` work exactly as described in our lecture on queues.

Phase 1: limited-use queue

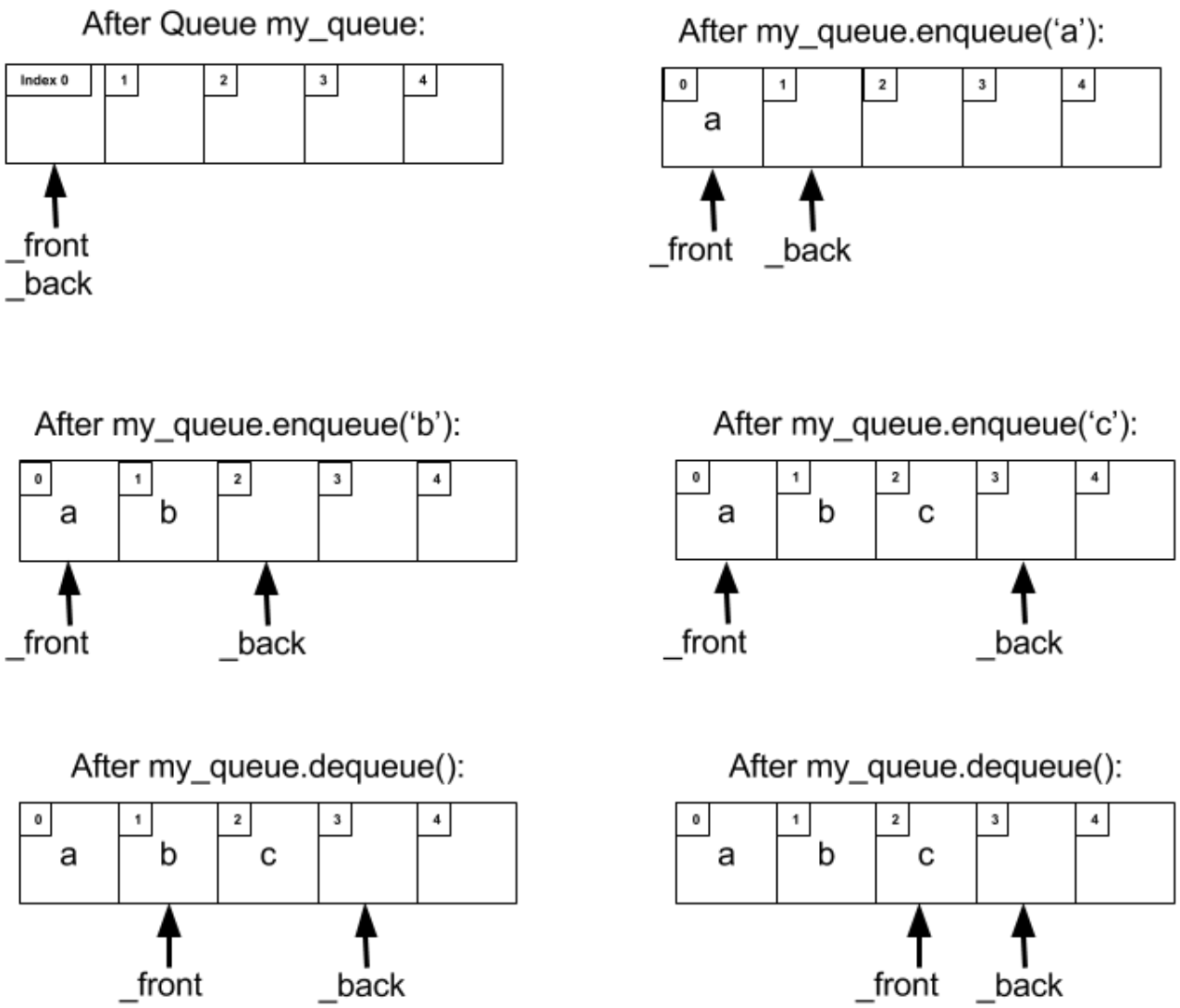
You will be building your `Queue` on top of a simple array of `char`. You don't need to use dynamic array allocation yet - just use the array `_data` as declared in the code already provided. For convenience as we go through the exercises, you can set the constant `ARRAY_SZ` to the size of the array (you will need this constant in a few places, so better to just define it once).

The simplest way you might think to design the queue is to simply start at the front of the array and fill to the back of the array; once the array is filled, no more elements can be enqueued, ever. You just need a couple of variables to keep track of where your active data begins and ends. Go ahead and add a couple of variables to your `Queue` class declaration in `Queue.h` - one to mark the first element in the queue, and one to mark the next empty location in the queue. For simplicity in the explanations that follow, I will refer to these as `_front` and `_back` respectively, but you can name them whatever you want.

Both `_front` and `_back` should start at zero, to give you the most use of your short-lived queue. The implementations of the four methods are pretty straightforward:

- `bool enqueue(char c)` - return `false` if already at capacity, otherwise put the passed value in the array at index `_back`, then increment `_back` and return true.
- `bool dequeue()` - return `false` if empty, otherwise add one to `_front` and return true.
- `char front()` - return the value at index `_front` in your array.
- `bool is_empty()` - return `true` if `_front == _back`.

Here's a diagram of enqueue and dequeue in action, using a queue named my_queue:



Go ahead and implement the methods as described above, setting `ARRAY_SZ` to 5, then build and run the `tests` program. You should be passing *some* of the tests, but since the queue simply runs out of space after a few enqueue operations, not all of the tests will pass.

Interlude: an expensive queue

So what could we do to fix the above queue so that it can be re-used indefinitely (as long as we never need more than 5 elements in the queue at one time, that is)?

One very simple approach would be to simply move all of the elements in the array down one index whenever dequeuing, rather than moving the `_front` index. That way, your first element is always in index 0, and you always have the full capacity of the queue available. (No need to code up any of this part, just some thought experiments!)

Suppose your queue has a capacity of *n*, where *n* is some largish number; and furthermore, suppose your queue is, on average, about half full. While enqueuing is very cheap (set one array value, change an int index value), dequeuing has suddenly become relatively expensive; now you have to run a loop that changes the values of (on average) *n/2* array locations! This means a simple dequeue operation has an algorithmic complexity of O(n), which is not optimal to say the least. In the next section, we'll see how we can improve on this to make the `dequeue()` operation have *constant* complexity - O(1).

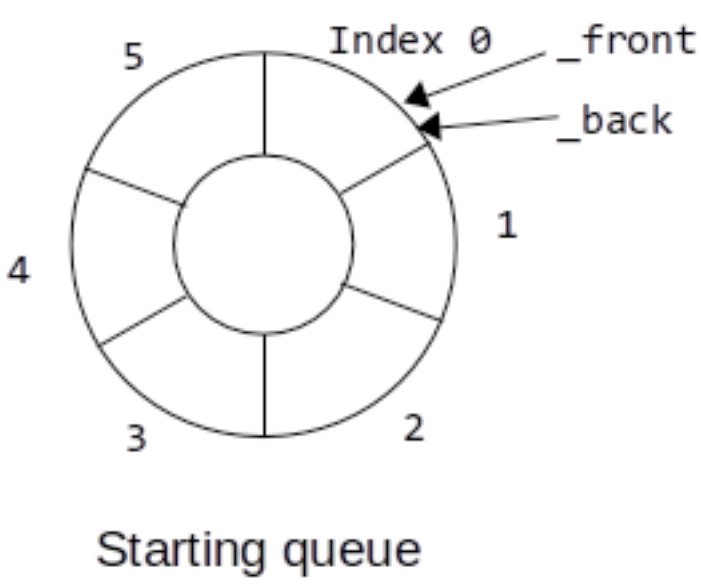
Phase 2: circular array

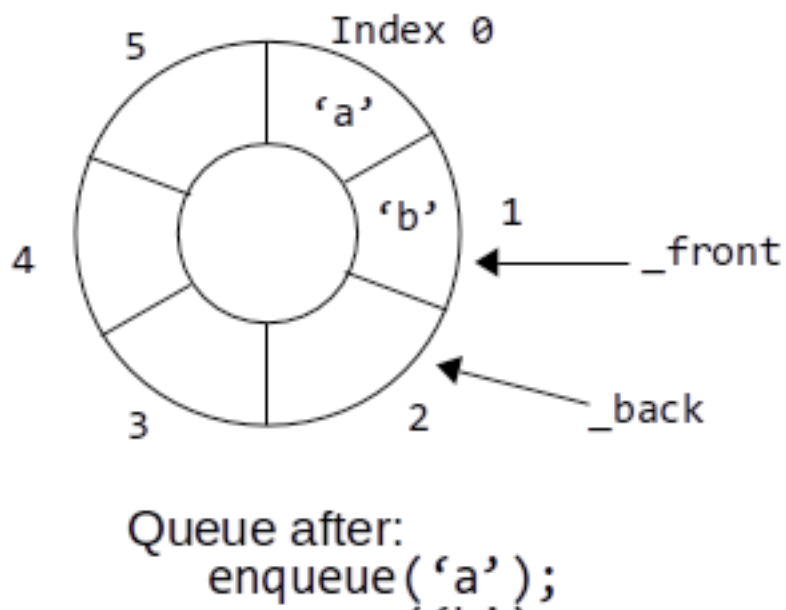
To avoid moving elements around the array while still being able to use the free space in the array, we will employ a simple trick: treating the array as if it were circular. That is, we will act as if the next index after the last index of the array is, in fact, index 0. That way, if we are at the end of the array, but there is room at the front (after some series of enqueues and dequeues), we can start re-using the space at the front of the array.

The way to make this work is to use the modulo operator (`%`) to wrap indices back around once they go off the end of the array. There are some other tricks we need, as well, detailed below.

For a refresher on the modulo operator and why we're using it here, check out the modulo refresher section below

Consider the picture below:



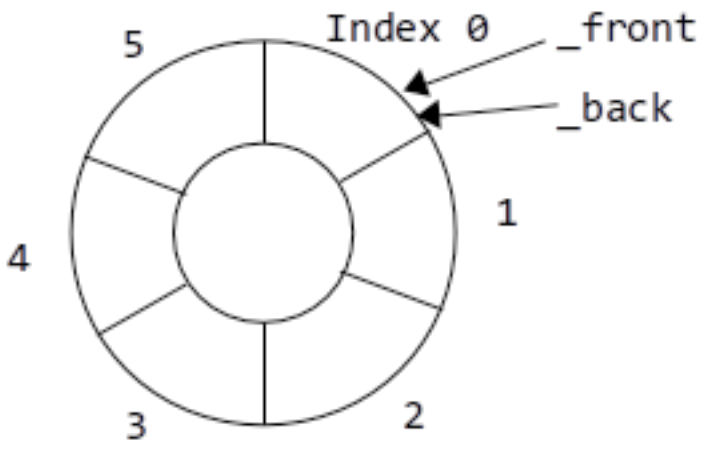


The pictured queue above has an array size of 6 (easier to draw than 5), so when we move our `_front` and `_back` indices, we need to always use `%` to make sure our value stays between 0 and 5. That is, if we update `_back`, afterwards (or as part of the operation) we should do

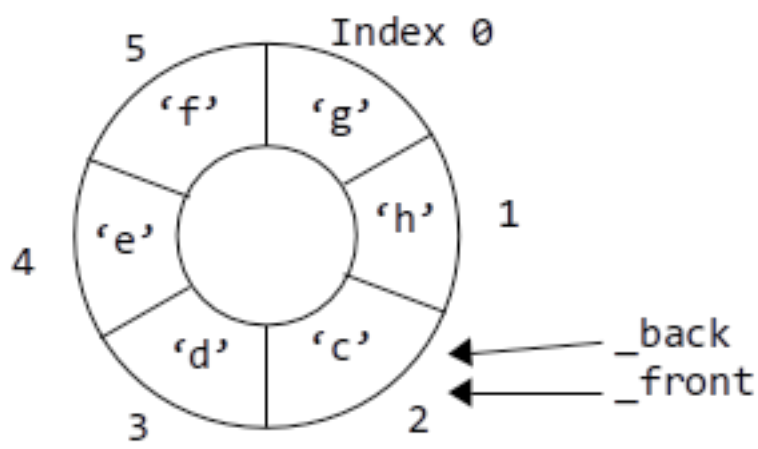
```
_back = _back % 6;
```

and the same with `_front` when we update it. (This is where having a constant like ARRAY_SZ defined is very useful!)

Before you rush out and code this up, though, note that we have one problem we didn't face before. How do we tell when the queue is full? If we aren't careful, we will get in trouble. The obvious choice is to say the queue is full when there is no more room in the array; but consider what this looks like:



Starting queue



Full queue, after many operations.

In this approach, the queue is full if `_front == _back`. But isn't this how we tell if the queue is empty, too? That won't work!

There are a few ways to fix this problem; however, for our purposes, the easiest way is to add an additional private member variable, `_size`, to the Queue class. The _size variable will keep track of how many elements are in the queue. You will need to update the _size variable whenever enqueueing (if not full) or dequeueing (if not empty) occurs.

Now you should be ready to code your final queue. **Be sure to set your array size so that your queue will hold a maximum of 5 elements at one time** - the tests depend on it.

Submission Instructions:

Submit a .zip file to Canvas containing all the source files for this lab: `main.cpp`, `Queue.h`, and `Queue.cpp`. Your zip file should also contain a README. The README should contain your name, the name of your lab partner(s), and any other information you think we should know.

Modulo Refresher:

If you're a little rusty on how the modulo operator works or why we're using it here, note that this works because modulo can be thought of as sort of a "wrap operator". That is, when we mod an int by some value, we'll get back the difference between our int and the next lowest whole multiple of that value. For example:

- `0 % 5 = 0`
- `1 % 5 = 1`
- `2 % 5 = 2`

- `3 % 5 = 3`
- `4 % 5 = 4`
- `5 % 5 = 0`
- `6 % 5 = 1`
- `7 % 5 = 2`
- `8 % 5 = 3`
- `...`
- `10 % 5 = 0`
- `11 % 5 = 1`
- `...`

This behavior causes mod to "wrap" our value back to 0 once we've increased our value to a multiple of the value we're modding by.

The other, more classic way to think of the modulo operator is as the remainder of division, but the "wraparound" behavior is why we're using it in this lab.

Grading:

Percentage of tests passed	9 points
README	1 point
Total:	**10 points**