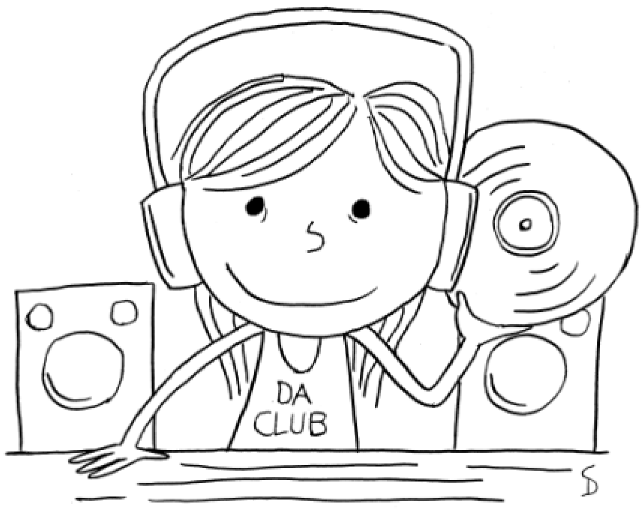
CS 15 Lab 03: Stacks, Queues, and Circular Buffers



Introduction

This week, we are working with Stacks, Queues, and Circular buffers. In particular, you will have to implement a few functions for all three! This week will also have a focus on unit testing.

Getting Started

The files for the lab are located at `/comp/15m1/files/lab03/`. At this point in the semester, you should be familiar with the process of setting up for the lab. If not, refer to any of the previous labs for tips.

Key Data Structures

For this lab, you will be implementing both a stack and a queue class. For both, you will implement a single underlying data structure: a circular buffer. Circular buffers are commonly used in things like network hardware or in audio and video systems. Details follow below.

Stacks and Queues

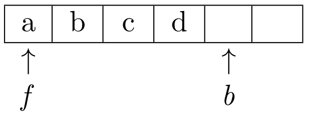
Remember, a stack is a data structure with a Last-In/First-Out (LIFO) property. Imagine stacking a pile of dirty dishes: you add the new plate to the top of the pile, and when you remove a dish to clean it, you will always remove the top plate first.

Conversely, a queue is a data structure with a First-In/First-Out (FIFO) property. Imagine standing in a line (or a queue) at a grocery store. The person who gets in line first is first to be helped.

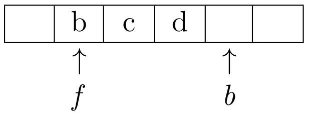
Circular Buffers

What is a circular buffer? A *buffer* is a place to store data temporarily. We're typically going to want to keep items in order, so an `ArrayList` like the ones you've built is a logical choice to implement a buffer. Remember, however, that adding and removing items at one end of an `ArrayList` is always very slow, because you have to copy the contents back and forth to make room or to squeeze items together. Circular buffers don't have this problem!

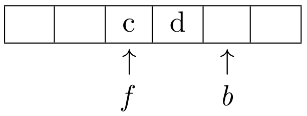
With a circular buffer, all data will still be "together," but, to avoid unnecessary copying, we keep track of the used and unused slots in an array. If someone removes an item from the front or the back, we'll note that the slot in the array for the removed item is available for reuse. For example, consider a buffer with capacity 6 that has already had 4 characters inserted into it:



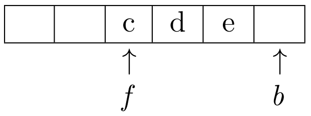
Here, f tracks the 'front' of the buffer—the first *data item*, and b tracks the 'back' of the buffer—the first *available space* in the array. In the lab, `front` and `back` are actually the integer indices in the array of the first used slot and first free slot, respectively.[1] Okay! What if we remove 'a'?



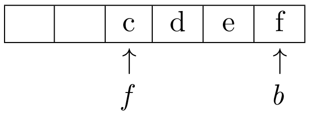
Ka-ching! All we had to do was increment f! While we show 'a' as being removed, we don't actually have to do anything about that array slot—by changing the front-index f, we now have a data-structure that acts *as if* 'a' were removed. What if we now remove 'b'?



Again, we simply increment f forward one space. Is this too easy? What if we want to add 'e' to the back?

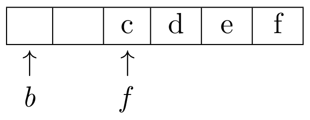


To achieve this, we put 'e' in the first available slot, indexed by b, and then increment b. What if we want to add 'f' to the back?

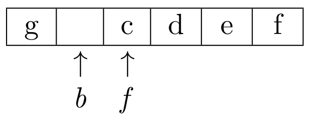


Here, we have a problem! What is b now? It can't point past the end of the array. Instead, let's think *circular* buffers! How about we point b to the front of the array?

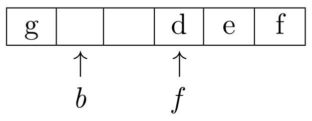
[1]Note: f and b are terrible names for data members! We're using short names to make our diagrams easy to read. `front` and `back` would be much better names to actually use in a program.



After filling the array, the ‘back’ pointer (b) now simply indicates that the next position to insert is really in the front of the array—but no user of the data-structure needs to know this! Instead, they should just be able to choose to insert at front or back of our buffer and have the data-structure take care of the actual locations for them. At this point, if we insert ‘g’ at the back of the buffer, we ill end up with:



Finally, if we remove the element at the front of the buffer, we will end up with:



Note: we still need to expand the array if we ever need to store more elements than we have capacity, and the f and b pointers meet up somewhere in the array. Therefore, we must still track the current size and capacity. Also, expansion requires a little more care, since we have to update our overall data-structure to add extra space, while also appropriately keeping track of the ‘front’ and ‘back’ or our data.

The Lab

There are bunch of functions for you to write. Work through as many as you can in lab. You might want to skip `CircularBuffer::expand()` at first—we suggest getting either the stack or queue set up using the basic public functions of the buffer, and then come back to `expand()` after that.

Tasks for Circular Buffer

1. Write the `int CircularBuffer::nextIndex` helper function that, given an index in the buffer, returns the next index. Remember to wrap around to the start of the array when you get to the end! (Why do we have this function?) Hint: You may find it useful to implement a similar private helper function that returns the previous index. Think about which functions below might benefit from a `prevIndex` function.

2. Write the `void CircularBuffer::addAtFront(ElementType elem)` method and test it. This should add the given elem to the ‘front’ of the sequence. Take a look at the pre-written `addAtBack` function from some direction if you need it. **Note**: unlike in an `ArrayList`, this does *not* need to shift any elements!

3. Write the `ElementType CircularBuffer::removeFromBack()` method and test it. This should remove the last element in the sequence.

4. Write `ElementType CircularBuffer::removeFromFront()` method and test it. This should remove the first element in the sequence. Again, there is no need to shift elements.

5. Write the `void CircularBuffer::expand()` method and test it. This should correctly expand the circular array. Pay special attention to wrapping. This function will be different than other expands you have written!

You are welcome to add additional private methods to the `CircularBuffer` class, if you wish.

Tasks for Stack and Queue

Once you have written (most of) the buffer methods, it should be easy to implement the following methods in the `Stack` and `Queue` classes.

1. Write and test `void Stack::push(ElementType element)`.
2. Write and test `ElementType Stack::pop()`.
3. Write and test `void Queue::enqueue(ElementType element)`.
4. Write and test `ElementType Queue::dequeue()`.

Tips

- Remember that `front` and `back` are integers. They are keeping track of the indices of the front and back of the list within the current array.
- Keep careful track of where `front` and `back` are in the array. If `front` is 0, and then you add an element to the front of the circular buffer, what should `front` be next?
- Make sure you test each function right after you write it!

You are encouraged to discuss test strategies with other students and with course staff.

Compilation and Linking with `make`

To compile and link your code, at the command prompt run the `make` command. `make` will compile your program according to the rules in the `Makefile`. Please look through the `Makefile`. There's a tutorial in there!

Running your code

To run your code, after compilation/linking, at the command prompt, run `./partyPlaylist`. Is the output correct? Does it make sense? Don't forget to run `valgrind ./partyPlaylist` to check for memory leaks and errors.

Submitting your code

When you are confident that your program is correct, run `make clean` to remove compilation products and then turn everything that is left in the folder to Gradescope.