CS 15 Lab: Stacks, Queues, and Circular Buffers

Introduction

This week, we are working with Stacks, Queues, and Cicular buffers. In particular, you will have to implement a few functions for all three! You will also implement tests along the way.

Getting Started

The files for the lab are located at `/comp/15m1/files/lab_stacks_queues_cb/`. 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, we will be implementing both a stack and a queue class. For both classes, we will implement a single underlying data structure - a circular buffer! Circular buffers are commonly used in things like network hardware as well as in video and audio systems - details are below.

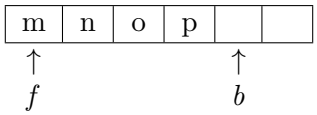
Stacks and Queues

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

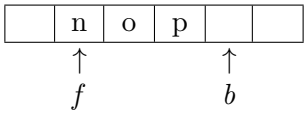
What is a circular buffer? A *buffer* is a place to store data temporarily. We're typically going to want to keep items in order, so an `ArrayList` like the ones you've built so far is a logical choice to implement a *buffer*. Remember, though, that adding and removing items at one end of an `ArrayList` is 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 circular buffer, all the data will still be 'together', but, to avoid the unnecessary copying our data, we'll keep track of the used and unused slots in the 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 elements inserted into it:

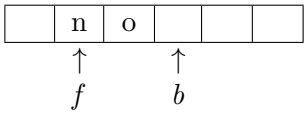


Here, f tracks the 'front' of the list - the first data item, and b tracks the 'back' of the list - the first available space in the array. In the lab, `front` and `back` are actually the integer indices in the array of the first used slot and first free slot, respectively. [1]

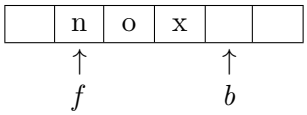
Okay! What if we remove the front element, 'm', at the 'f' index?



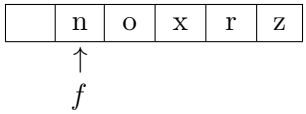
Ka-ching! All we had to do was increment f! What if we remove the element at the back of the buffer, 'p'?



What if we want to add 'x' to the back?

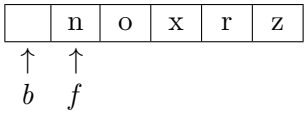


Let's add 'r' to the back, and then add 'z' to the back.

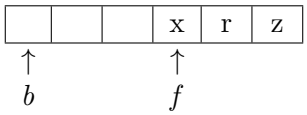


[1] Note: f and b are terrible names for data members. We're using short names to make our diagrams easy to read. front and back would be much better in a program!

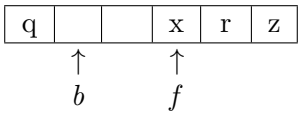
Problem! What is b now? It can't point past the end of the array. Let's think *circular* buffers! How about we point b to the front of the actual array?



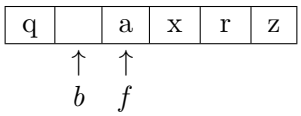
Let's let's make two calls to the function that removes the front element



Now what happens when we insert 'q' at the back?



Now what happens when we insert 'a' at the front?



Depending on implementation of the buffer, you might decide to simply overwrite elements if you run out of room; for our lab, we will assume you allocate enough memory for the elements to start - expansion, however, is a JFFE.

The Lab

For the lab there are bunch of functions for you to write! Work through as many as you can, and test your functions as you go.

Notes

- The `CircularBuffer`, `Stack`, and `Queue` classes are all templated with the element type, so they are to be implemented in their respective '.h' files.
- `CircularBuffer::addAtBack` and `CircularBuffer::removeFromFront` have already been given to you. **Be sure you understand these functions!** You may use them as templates for the other functions you will need to write.
- You are welcome to add additional private methods to the `CircularBuffer` class, if you wish.

Tasks for Cicular Buffer

1. Write the `nextIndex` helper function and test it [note that `addAtBack` uses it!]. Given an index in the buffer, this function returns the next index. Note that in this case, if the input is the 'back' index, the 'next' index will be the next logical position where you would insert an element (not the 'front' index). Remember to wrap around to the start of the array when you get to the end!

2. Write the `addAtFront` method and test it. This should add the given elem to the "front" of the circular array as illustrated in the examples above. Note: This should not shift elements! You may find it helpful to write a `prevIndex` helper function that complements the `nextIndex` function you wrote above.

3. Write the `removeFromBack` method and test it. This should remove the last element in the sequence. See the provided `removeFromFront` for a template of a similar function.

4. JFFE: (skip this for now and come back if you have time later): Write the `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!

Tasks for Stack and Queue

Once you have written the circular buffer methods, it should be easy to implement the following methods in the Stack and Queue classes.

1. Write the `void Stack::push(ElementType element)` method and test it.

2. Write the `ElementType Stack::pop()` method and test it.

3. Write the `void Queue::enqueue(ElementType element)` method and test it.

4. Write the `ElementType Queue::dequeue()` method and test it.

Tips

- `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 array, 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 the course staff. Recall the previous lab in which we went over incremental development and debugging output, and use these strategies as you write your functions.

Testing your code

To test your work, we have provided you with a `main.cpp` file. To test with the driver file: run the command `make`, then run the command `./partyPlaylist`. Does the output make sense? Don't forget to run `valgrind ./partyPlaylist` to check for memory leaks/errors!

Submitting your code

See Canvas for instructions on how to submit your lab code. The files you will need to submit are:

```
CircularBuffer.h Queue.h Stack.h Makefile main.cpp
```

Wrap-up Questions

- In your opinion, which data structure is better for a DJ table?
- Which one is better for a Top 10 Countdown on the Radio?
- How do stacks and queues interact with each other?
- What happens when you dequeue into a stack and then pop everything off?