CS 15 Stacks, Queues, and Circular Buffers Lab

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/15/files/lab_stacks_queues_circular_buffers/`. At this point in the semester, you should be familiar with the process of set-

ting 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.

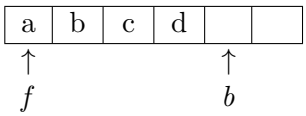
Conversely, a queue is a data structure with a First-In/First-Out (FIFO) property. Imagine standing in a line (or a queue) at a grocery store. The person who gets in line first is first to be helped.

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 sp 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!

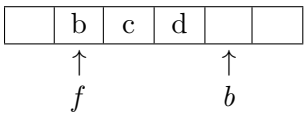
With circular buffer, all the data will still be 'together', but, to avoid the unnecessary copying our data, we'll keep track of the used and unused slots in the array. If someone removes an item from the front or the back, we'll note that the slot in the array for the removed item is available for reuse.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



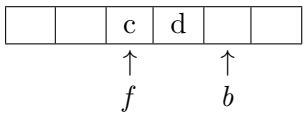
Here, f tracks the 'front' of the list - the first data item, and b tracks the 'back' of the list - the first available space in the array. In the lab, `front`

and `back` are actually the integer indices in the array of the first used slot and first free slot, respectively. [1]

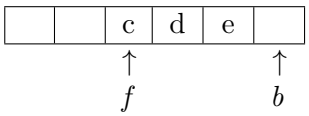
Okay! What if we remove 'a'?



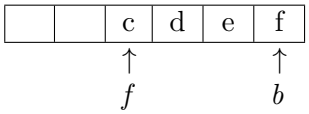
Ka-ching! All we had to do was increment f! What if we remove 'b'?



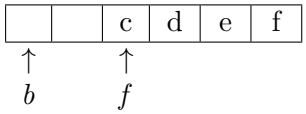
Is this too easy? What if we want to add 'e' to the back?



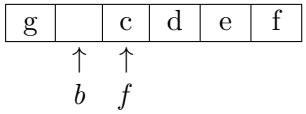
We put 'e' in the first available slot, indexed by b, and then incremented b. What if we want to add 'f' to the back?



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?

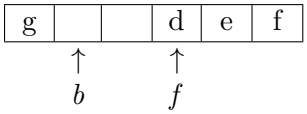


Now, what happens when we insert 'g' at array[`back`], then update `back`?



Let's remove the element at array[`front`], then update `front`

[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!



Notice that we still need to expand the array if we need to store more elements than we have capacity. Therefore, you must still track the current size and capacity. Otherwise, you'll risk adding elements and having them overwrite existing data. Also, expansion requires a little care.

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 working, and then come back to expand.

Also, it would to your advantage to write tests **as you're writing your functions!** Lab is a good opportunity to begin developing this habit!

See the `testing` section for details.

Tasks for Cicular 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::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!

3. Write the `void CircularBuffer::addAtFront(ElementType elem)` method and test it. This should add the given elem to the "front" of the sequence. Note: This should not shift elements!

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

5. Write `ElementType CircularBuffer::removeFromFront()` method and test it. This should remove the first element in the sequence.

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

Note!

A function to add an element to the back of the array has already been given to you. **Be sure you understand it.** You may use this function as a template for the other functions you need to write.

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, you have **two** options - either way you'd like to test is fine! Whichever way you go, please add more tests to the respective file!

1. You may use the provided `main.cpp` driver file, or
2. You may use the provided `unit_tests.h` testing file.

Both files have the same kinds of tests!

Driver file testing

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!

unit_test testing

To test with the unit testing framework, run the command `unit_test` (or, use the `VSCode` extension as discussed in lab1.)

Providing your code

When you are confident that your program is correct, turn it in.

You will need to submit the following files:

```
unit_tests.h
CircularBuffer.cpp
CircularBuffer.h
Queue.cpp
Queue.h
Stack.cpp
Stack.h
Makefile
ElementType.h
README
main.cpp
```

You must submit them using Gradescope to the assignment `lab_stacks_queues_circular_buffers`
Submit these files directly, do not try to submit them in a folder.

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?