CS 15: Circular deBuggers Lab

```
                  ,
     _,-'\   /|    .    .    /`.
 _,-'     \_/_|_  |\   |`. /   `._,--===--.__
^       _/"/  " \ : \__|_ /.   ,'    :.  :. .`-._
       // ^   /7 t'""    "`-._/ ,'\   :   :  :  .`.
       Y      L/ )\         ]],'   \  :   :  :   : `.
       |        /  `.n_n_n,','\_    \ ;   ;  ;   ;  _>
       |__    ,'     |  \`-'    `-.__\_______.==---'
      //  `""\\      |   \            \
      \|     |/      /    \            \
                    /     |             `.
                   /      |               ^
                  ^       |
```

From: `https://www.asciiart.eu/animals/insects/ants`

Introduction

This week, we will be working on **debugging** code by introducing a new data structure: a *circular buffer!* We will also be reviewing *stacks* and *queues.*

Key Data Structures

In this lab, we will be fixing a buggy implementation of a circular buffer, and you'll see how to use a circular buffer to implement higher-level abstractions like stacks and queues.

Stacks

Stacks are a *Last-In/First-Out (LIFO)* data structure. Imagine a stack of books. The book at the top of the stack would be the book that was most recently added to the stack. Note that if we tried to take a book directly out of any part of this stack besides the top, it would fall over! The only book that we can pull from the stack without creating an avalanche is the one at the top. We can add to the top and remove from the top.

Queues

Queues are a *First-In/First-Out (FIFO)* data structure. This time, imagine a line at the grocery store. The first person who entered the line will be the first person to leave. Again, note that if the cashier tried to take the second person in line before the first, the other customers wouldn't be happy. So, like stacks, we can only directly access elements at the front of a queue. We add elements at the back and remove from the front.

Implementing stacks and queues

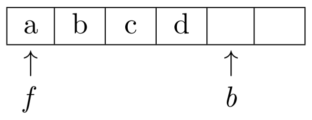
How should we implement these data structures? As we can see, both stacks and queues are essentially just lists with certain restrictions. This is a great hint that we should use either an ArrayList or LinkedList. But which one would be better? Can both be used without taking any major hits to runtime? *Consider the computational complexity of adding/removing from the front or back of a list when you think about this question!*

Circular Buffers

A *buffer* is a place to store data temporarily. Circular Buffers combine the best traits of LinkedLists and ArrayLists into one structure. Through some careful maintenance of indexing variables, circular buffers offer `O(1)` insertion and deletion to the front and back of the list, as well as `O(1)` access to any element.

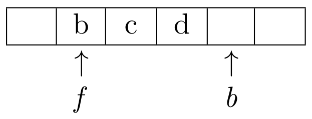
Like ArrayLists, circular buffers store data in standard C++ arrays located on the heap. They also have member variables that track the number of items inserted into the array and capacity of the array. Where things get funky, though, is with the front and back member variables. These will be integers that track the index of the first element in the array and the index of the last element `+ 1` respectively.

For example, consider a buffer with capacity 6 that has already had 4 characters inserted into it:

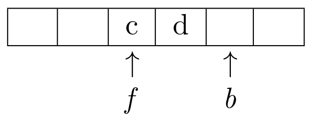


Here, we get that `f = 0`, and `b = 3 + 1 = 4`.

Okay! What if we remove '`a`'?

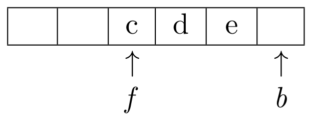


Boom! All we had to do was increment `f` from `f = 0` to `f = 1`! What if we remove '`b`'?



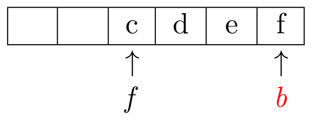
Now, `f = 2`, `b = 4`.

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



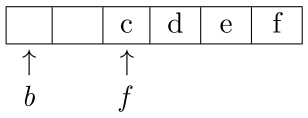
We put '`e`' in the first available slot, given by `b`, and then incremented `b`.

What if we want to add '`f`' to the back?

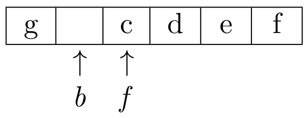


Problem! What is `b` now? It can't point past the end of the array. Let's think *circular* buffers! How about we point `b` to the front?

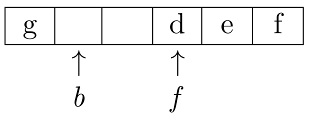
Also think about how you would go about moving `b` from the last element in the list to being the front—can any of the 5 operations (+, -, x, ÷, %) help you?



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



Let's remove the element at `array[front]` ('c'), then update `front`.



Notice that we still need to expand the array if we need to store more elements than we have capacity. Otherwise, you'll risk adding elements and having them overwrite existing data. Expansion is similar to ArrayLists, except now we need to be mindful of our `front` and `back` member variables. Talk about how you might go about expanding a circular buffer with the person sitting next to you!

The Lab

Now that you have a better understanding of how circular buffers work, we're going to have you debug our implementation of a circular buffer!

Getting Started

Get the lab files in the usual way—the lab is called `lab_debug_circular_buffers`. (It is important that you master the Unix commands!)

To-Do Items

Your tasks are:

1. Find and fix the bugs in our implementation:
   (a) Start with the **compilation** errors, then run our code to debug the **runtime** errors
   (b) `main.cpp` calls and tests each of the circular buffer functions. If you need hints on what could be creating issues, that's a great first place to look!
2. Implement each of the unimplemented functions in `Stack.cpp`
3. Implement each of the unimplemented functions in `Queue.cpp`

Running Our Code

To compile our code, use the command `make pokedex`. To execute, run `./pokedex` in terminal. Happy debugging!

Debugging Tips and Tricks

There are a few steps that we want you to follow when debugging. These will help you to diagnose, then fix whatever is broken in your code. Use the following strategies to fix our buggy implementation of the circular buffer!

1. Add print statements to narrow down the location of the bug to one specific line or function.
2. Perform thorough tests. Note that `assert()` statements that fail will lead to a big Valgrind dump—to avoid these, comment out assert statements and replace them with print statements that check the same thing.
3. Run Valgrind. See our guide to fixing Valgrind errors (also accessible from the course website).
4. Read the spec, provided documentation, and style guide. Sometimes the answer you want is easily accessible!

5. Check Piazza, Stack Overflow, and other online resources. **These should not be used to write programs/functions in their entirety**, but for help with specific compilation errors, details on C++ libraries, etc.

Submitting

You will need to submit the following files:

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

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

```
     _  _
     | )/ )
\\   |//,' __
  (")(_)-"()))=-
     (\\
```

From: `https://www.asciiart.eu/animals/insects/ants`