Lab 6: Queues & Unit tests

The purpose of this lab is to give you experience working queues. In this lab, the queue is implemented with a dynamic array using a vector like struct.

This lab has a total of 100 possible points. You are encouraged to work as a pair.

Objectives

Upon completion of the laboratory exercise, you will be able to do the following:

- Read implementations for queues.
- Implementation of enqueue function
- Implementation of print function
- Use queue data structures to add and remove items.
- Write unit test functions to test your queue functions
- Create makefiles that allows you to compile easily.
- Use Valgrind facility to check for memory leaks

Part 0: Starter Code

Pull the starter code from the class repo in Github. Once you have the folder in your local repo (either in the Khoury server or your machine, you should see a folder named Lab6. Once you open the folder, you should be able to see the following files:
main_queue.c , queue.c and queue. h and also unit_tests.c, unit_tests.h and main_test.c

Part 1: Testing the Code

1. The queue data structure for this lab is implemented as an array. Open `queue.h` to see queue the blueprint (or api) for the queue data.

2. Open `main_queue.c`. Look at the code and predict what the queue has in it before you test this function. What does the queue contain (in order such that the left-most item will be the next item dequeued) after 14 is enqueued?

front of queue_____back of queue

3. Compile and run the code as follows:

```
gcc -Wall main_queue.c queue.c queue.h
./a.out
```

4. Is your prediction in #2 correct? YES NO

5. Open `queue.c` to see how the functions are implemented. The `frontIndex` is the index in the array where the first item in the queue is held. This is the index from which items shall be removed

from. The `backIndex` is the index where a new last item is inserted. You will notice that the actual last item in the queue is at `backIndex` -1 position. Note that by keeping track of the `frontIndex` and `backIndex`, we do not need to physically move items in the data array (copying/shifting them to the left as items are removed from the queue).

a. What is the value of `queues frontIndex` just after 4 in enqueued in main? _____
(You can print this value if to confirm your prediction.)

b. What is the value of `queues backIndex` just after 4 is enqueued in main? _____
(You can print this value if to confirm your prediction.)

c. What is the value of `queues frontIndex` just after 20 is enqueued in main? _____

d. What is the value of `queues backIndex` just after 20 is enqueued in main? _____

9. Open `queue.h`. At the top of the file, you will see two preprocessor directives:

```
#ifndef QUEUE_H
#define QUEUE_H
```

What do these preprocessor directives do _____

Part 2: MakeFile

Using examples from earlier classes, create a makefile so that you can easily be able to run your code. Make sure that your code compiles and runs with no memory leak. You will need to demonstrate your makefile as well. Notice that this part just requires creating targets and variables for these files: main_queue.c , queue.c and queue. h

Part 3: Print Function

You may have noticed the print function is just but a stub. Please complete this function. Note that the function should be able to print all the data starting with the item at the form of queue to last item in the queue.

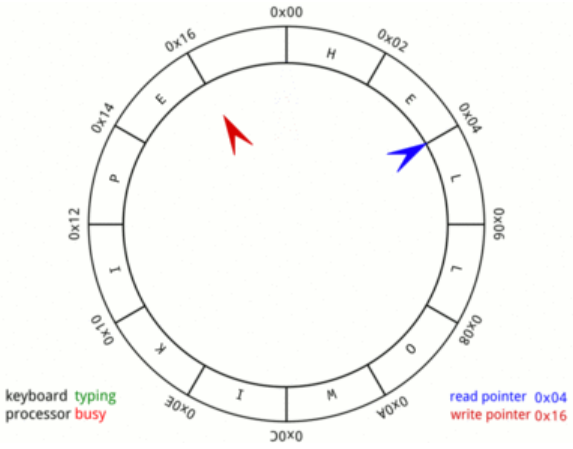
Checkpoint 1 [30 points]: Show your lab instructor/assistant the result of your program running with Show your answers to the questions above.

Part 4: Enqueue function

The current implementation does not take care of when the array is full. In this section, you need to work on creating code fragment that allows for the queue to double its capacity when it is full. Note that

doubling the capacity should mean that we would copy items from old queue to a new queue. The idea is that items should not lose their place in the queue. Further, notice that you can reset the position in the index when you do the copying such that item at front Index starts again at zero.

Since we are able to dequeue items, you might notice that the backIndex (or last item in the queue) does not correspond to the total items in the queue, so you need to cleverly design your function that allows the system to check whether array is full (by comparing total items with capacity) and then allows items to be added in the open spots. Essentially this means that your array is wrapping around.



Think about what mathematical operator has a *wrap around behavior* that could be useful in your implementation when enqueuing and dequeuing items in your queue (*hint* this operator will save you from writing many if-statements in your code!).

Notice that changing this enqueue function has other multiple parts from different functions that may need to be modified for your data structure to work correctly:

a) Ensuring the wrap around behavior for when the queue is not full
b) Ensure that when print Front item, the correct front item is picked
c) Ensure that when printing Tail/Back item that the correct back/tail is picked
d) Ensure that dequeue removes the correct Front item.

Checkpoint 2 [20 points]: Show your lab instructor/assistant the result of your program running and your code. Show your answers to the questions above.

Part 5: Unit Testing

A unit test is a standalone test that checks for the correctness of a specific use case in your code. In our case, we are testing if we have a working queue implementation.

Modify your Makefile so that you add a new target called unit_test that will build a new executable that we can use to test our queue data structure. Feel free to use other variables in your macro and remember to modify your clean target so that it can clean up the space for these new files. You will also need to add target lines for the new .o files: unit_tests.o and main_test.o

```
unit_test: queue.o main_test.o unit_tests.o
     $(CC) $(CFLAGS) -o unit_test main_test.o unit_tests.o queue.o
```

We begin by building a framework for testing our code. We are going to create a main program in file called main_test.c that will initialize an array of function pointers where every function pointer will hold the address of a test function which has the following signature:

```
Status long_function_name(char* buffer, int length);
```

The idea is to write a program that will automatically run all of our test functions and report on their success or failure. As we write more functions to test our code this unit will become more and more useful to us.

The provided starter code already has two test functuions provided in the unit_tests.c and the header file unit_tests.h. In each test notice that they return the same type and take the same two types of parameters. The name for a test should be descriptive of what it is testing. Do not shy away from long function names. In the above test we are trying to create a queue object using the queue_init_default function. We then simply check to see if we get something that is not NULL back. This function does not verify that the object that we get back is in fact a queue and does not attempt to do any further testing on it. Think about these tests as tiny tests that are to test one thing at a time.

The second test tries to verify that the queue we create using the default init function has a total number of items set at zero. We expect a default queue to be “empty” and so it makes sense for it to not have anything stored in it after the init. Your tests should allocate the memory they need for their test and free it up before they return to the caller.

Checkpoint 3 [20 points]: Demonstrate to your TA that you can build unit_test with your Makefile and can run the sample code against the above given tests. You must demonstrate that your code does not have any memory leaks using valgrind.

The final part of this lab is about learning to test your own code. You must write a total of 13 more tests (so that you have a total of 15 tests. Your tests must satisfy the following. They must have your loginid in the name of the function so my functions will now be called test_jmwaura_blah_blah_blah.

Checkpoint 4 [30 points]: Demonstrate to your TA all your 15-unit tests and demonstrate how they work. You must demonstrate that your code does not have any memory leaks using valgrind.

If any components/ checkpoints have not been seen at the end of the lab; carry out the work during the week and show it next lab hour.

For submission: Make sure your code works fine in the Khoury Server. Then push this work to your github repo.