In this assignment, you will be building two very popular data structures in C.

Part 1 -- Build a Queue

The queue, in computer science, is defined as #4 in this definition. It is what is known as a first-in, first-out (FIFO) data structure. Thus, queues have a strict policy about how information is stored and accessed. In this part of this assignment, you are going to implement a very specific implementation of a queue that uses an array called a circular queue (or more specifically a ring buffer when the maximum storage size is fixed).

Queue implementation as an array (ring buffer)

A circular buffer has a fixed size or maximum number of elements that it can hold. This is advantageous since knowing how many elements can be stored means that we can use an array in the implementation. (If we do not know, then a queue with a linked list implementation is needed). You can learn more about the circular buffer by reading this Wikipedia page(https://en.wikipedia.org/wiki/Circular\_buffer) on the topic. (This page also has a very nice animation that provides an idea about how the data structure "wraps" around.)

A picture containing circle, diagram, line

Description automatically generated

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!). Your task will be to implement the functions in the myqueue.h to have a working implementation of the queue.

What is a file that ends in .h?

A file that ends in .h is known as a header file. A header file typically is a file that contains an “interface" for a set of functions to perform some task. The actual implementation (i.e. the loops, if-statements, arrays, and tools that do work) is usually found in a .c file. For this assignment, our 'queue' library is quite small, so we will implement the entire functionality into a header file. There are some pros and cons to this approach. One particular "pro" is it makes your code very easy to test. We will simply use our own "queue\_main.c' file to test your implementation (that is why it is important you do not change the names of any functions).

More information on header files

https://gcc.gnu.org/onlinedocs/cpp/Header-Files.html

(<https://gcc.gnu.org/onlinedocs/cpp/Header-Files.html>)

Unit Tests

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. A sample unit tests is given:

A picture containing text, font, screenshot, line

Description automatically generated

You should consider writing some unit tests to test your implementation (In fact, I would strongly encourage you to do so). Note that you can include your unit tests in your submission, and we will have our own test suite. Some example tests we might come up with include:

* Fill a queue, empty the queue, and fill the queue again.
* Create an empty queue and attempt to add something to it
* Make sure your queue does not overwrite any data.
* etc.

Provided Tests

You are provided a file called queue\_test.c which you can compile separately and test the implementation of your queue functions from your header file. Doing so does not guarantee a perfect assignment, but it will give you some confidence your implementation is working.

Compile: gcc queue\_test.c -o queue\_test

Run: ./queue\_test

(An optional task to extend this assignment--not for bonus, but just for fun)

Add additional functions like:

* queue\_peek - Returns the first value in the queue
* queue\_back - Returns the last value in the queue
* queue\_equals - Checks if two queues are equal

Part 2 -- Build a Stack

A stack is what is known as a last-in, first-out (LIFO) data structure. Thus, stacks also have a strict policy about how information is stored and accessed. In this part of this assignment, you are going to implement a very specific implementation of a stack using a linked list.

Stack implementation as a linked list

Our stack has two fundamental operations that it can perform:

1. push - Add elements to the top of the stack
2. .2. pop - Remove elements from the top of the stack.

You will notice, both of these operations happen at the top of the stack as shown in the image below. A picture containing text, diagram, sketch, plan

Description automatically generated

Our stack that we are implementing can expand dynamically, thus a linked list is a good foundation data structure to use in our implementation

ur stack can push elements onto it, until MAX\_DEPTH is reached. MAX\_DEPTH is defined as 32. This is an artificial limit for our implementation. Remember however, we only have a finite amount of memory on our systems, so if we had very large stacks, we could get a stack overflow (https://en.wikipedia.org/wiki/Stack\_overflow).

More information on the stack, can be seen here: https://en.wikipedia.org/wiki/Stack\_(abstract\_data\_type(https://en.wikipedia.org/wiki/Stack\_(abstract\_data\_type) )

Your task will be to implement the functions in the stack.h to have a working implementation of stack.

Unit Tests

Again, unit tests are a great way to ensure that we have a working stack implementation. Sample unit tests are given:

A picture containing text, font, screenshot, line

Description automatically generated

You should also consider writing some unit tests to test your implementation (In fact, I would strongly encourage you to do so). Note that you can include your unit tests in your submission, and we will have our own test suite. Some example tests we might come up with include:

* Fill a stack, empty the stack, and fill the stack again.
* Create an empty stack and pop an item to add something to it.
* Etc

Going Further

(An optional task to extend this assignment--not for a bonus, but just for fun)

Add additional functions like:

* stack\_peek - Returns the top value of the stack (without removing it)
* stack\_equals - Checks if two stacks are equal

Part 3 -- Written Exercises

Use your favorite word processing program to create a PDF called "hw02-exercises.pdf" with answers to the following questions. Each question should be answered starting on a new page of your document. You should add/commit/push this file into the folder for this assignment. When you use outside sources, you should cite them. In addition, you should follow good citation practices. I recommend reading this Purdue University website, their Online Writing Lab, on good citation practices and avoiding plagiarism:

<https://owl.purdue.edu/owl/avoiding_plagiarism/plagiarism_faq.html(https://owl.purdue.edu/owl/avoiding>

\_plagiarism/plagiarism\_faq.html) . The entire "Avoiding Plagiarism" section is very useful and you should read it. If you're going to ask me what citation style you should use, I recommend IEEE, as it's one of the most common in CS.

https://owl.purdue.edu/owl/research\_and\_citation/ieee\_style/ieee\_overview.html(https://owl.purdue.edu/owl/research\_and\_citation/ieee\_style/ieee\_overview.htm)

I don’t-specifically care which citation style you use - if you already know another one like MLA or Chicago, feel free to use that. But if you're learning a new style, pick IEEE, it's the most relevant for CS. How you include a bibliography or works cited page depends on the citation style you choose, just be consistent with the style. You'll be writing research papers or article during your degree here, so this is a great time to start learning the conventions of academic writing if you're not familiar with them - but this is still just a paragraph or two in a homework assignment, so don't spend too much time on this. 'Close enough' for citation will be fine for now.

1. Circular queues are used quite a bit in operating systems and high performance systems, especially when performance matters. Do a little outside research, and edit this section of the readme answering specifically: Why is a ring buffer useful and/or when should it be used?

2. We are going to talk about stacks quite a lot in this course, so it will be important to understand them. Do a little outside research, and edit this section of the README answering specifically: Why is a stack useful and/or when should it be used?

For each of these questions, we are expecting your answer to be in the range of 2-3 paragraphs (can include complexity, example usage, pros/cons, etc).