Lab 12 - Implementing a Queue Using a Circular Linked List

A recent topic presented four designs for a FIFO queue container that uses a singly-linked list as the underlying data structure. One of the designs permits the `enqueue`, `dequeue` and `front` functions to have efficient implementations; that is, the `enqueue`, `dequeue` and `front` functions all run in constant time.

In this lab, you are going to reimplement the queue to use a *circular* singly-linked list as the underlying data structure.

General Requirements

None of the functions you write should call `calloc` or `realloc`.

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your functions should produce console output; i.e., contain `printf` statements.

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). Instructions were given in the VS Code installation instructions.

Finish each exercise (i.e., write the function and verify that it passes all of its tests) before you move on to the next one. Do not leave testing until after you have written all your functions.

Instructions

Step 1: Depending on your operating system, download (**lab12_win.zip** or **lab12_mac.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Double click on lab12.code-workspace

Step 3: lab12_xxx folder contains the following files:

- circular_queue.c contains an incomplete implementation of a queue module. Several functions are fully implemented:
    - `alloc_queue` allocates and initializes a new, empty queue;
    - `queue_is_empty` returns `true` is a queue is empty; otherwise it returns `false`;
    - `queue_size` returns the number of elements stored in a queue;
    - `queue_print` outputs the contents of a queue on the console;

  This file also has incomplete implementations of functions `enqueue`, `front` and `dequeue`.

- circular_queue.h contains the declarations for the queue data structure and the nodes in a queue's circular-linked list (see the `typedef`s for `node_t` and `queue_t`), followed by the prototypes for functions that operate on queues. **Do not modify circular_queue.h.**
- main.c contains a simple *test harness* that exercises the functions in circular_queue.c. Unlike the test harnesses provided in several labs, this one does not use the sput framework. The harness doesn't compare the actual and expected results of each test and keep track of the number of tests that pass and fail. Instead, the expected and actual results are displayed on the console, and you have to review this output to determine if the functions are correct. **Do not modify `main()` or any of the test functions.**

Step 4: Run the project. It should be built without any compilation or linking errors.

Step 5: Execute the project. The test harness will show that functions `enqueue`, `front` and `enqueue` do not produce correct results (look at the output printed in the console window and, for each test case, compare the expected and actual results). This is what we'd expect, because you haven't started working on the functions that the test harness tests.

Step 6: Do Exercises 0 through 3. If you become "stuck" while working on the exercises, consider using C Tutor to help you discover the problems in your solutions.

Exercise 0

Open circular_queue.c and circular_queue.h. Read the declarations for structs `node_t` and `queue_t` in circular_queue.h. Read the code for `alloc_queue`. Make sure you understand why the `rear` and `size` members of the `malloc`'d `queue_t` struct are initialized to `NULL` and `0`, respectively.

The queue's data structure is a circular linked list. Every node in this linked list points to another node, and the tail node points to the head node.

Read the code for `queue_print`. The function's first parameter, `queue`, is a pointer to a `queue_t` struct. Notice how `queue->rear` points to the node at the tail of the linked list, which corresponds to the rear of the queue. This node points to the node at the head of the linked list, which corresponds to the front of the queue. In other words, `queue->rear->next` points to the head node.

Exercise 1

File circular_queue.c contains an incomplete definition of a function named `enqueue`. The function prototype is:

```
void enqueue(queue_t *queue, int value);
```

Parameter `queue` points to a queue. The function will terminate (via `assert`) if `queue` is `NULL`.

This function will store the specified value at the rear of the queue.

Design and implement `enqueue` (but read the following paragraphs before you do this.)

There are two cases to consider:

- The queue is empty.
- The queue is not empty (its linked list has one or more nodes).

Hint: in order to maintain the circular property of the queue's linked list, when the queue has only one node, that node must point to itself. In other words, the `next` member of the only node in the linked list must point to that node.

We recommend that you sketch some "before and after" diagrams of the queue for each case before you write any code. (One diagram should show the queue - the `queue_t` struct and its circular linked list - before the function is called, the other diagram should show the queue after the function returns.) Use these diagrams as a guide while you code the function. Feel free to "borrow" and adapt code from the queue implementation posted on Brightspace.

We also recommend that you use an iterative, incremental approach when implementing this function. For example, during the first iteration, write just enough code to handle the "queue is empty" case. Use the console output to help you identify and correct any flaws. When your function passes the tests for this case, write the code for the "non-empty queue" case and retest your function.

Verify that `enqueue` passes all of its tests before you start Exercise 2.

Exercise 2

File circular_queue.c contains an incomplete definition of a function named `front`. The function prototype is:

```
_Bool front(const queue_t *queue, int *element);
```

Parameter `queue` points to a queue. The function will terminate (via `assert`) if `queue` is `NULL`.

This function copies the value stored at the front of a queue to the variable pointed to by parameter `element`, and returns `true`. The function returns `false` if the queue is empty. The function does not modify the queue.

Design and implement `front`. Use the console output to help you identify and correct any flaws. Verify that `front` passes all of its tests before you start Exercise 3.

Exercise 3

File circular_queue.c contains an incomplete definition of a function named `dequeue`. The function prototype is:

```
_Bool dequeue(queue_t *queue, int *element)
```

Parameter `queue` points to a queue. The function will terminate (via `assert`) if `queue` is `NULL`.

This function copies the value stored at the front of a queue to the variable pointed to by

parameter `element`, removes that value from the queue, and returns `true`. The function returns `false` if the queue is empty.

Design and implement `dequeue`.

There are three cases to consider:

- The queue is empty.
- The queue has one element (its linked list has exactly one node).
- The queue has two or more elements (its linked list has two or more nodes).

We recommend that you follow the same approach that suggested for Exercise 1; that is, before you write any code, sketch some "before and after" diagrams of the queue for each of the cases, then use the incremental, iterative technique to code and test the function.

Use the console output to help you identify and correct any flaws. Verify that `front` passes all of its tests.

Wrap-up

Submit circular_queue.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace* (lab3 file). Remember: submit early, submit often.
- Make sure that circular_queue.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you are submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

Homework Exercise - Visualizing Program Execution

You will be expected to be able to understand and draw diagrams that depict the execution of short C functions that manipulate linked lists, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills.

If you didn't use C Tutor to help you implement the solutions to the exercises, use the tool to visualize the execution of your `enqueue`, `front` and `dequeue` functions.

1. Click on this link to open C Tutor.
2. Copy/paste the `node_t` declaration from circular_linked_list.h into C Tutor. Copy `alloc_queue` and your solutions to Exercises 1 through 3 from circular_queue.c into C Tutor. You can comment out the `assert` calls.
3. Write a short `main` function that exercises your function. Feel free to borrow code from this lab's test harness.
4. *Without using C Tutor*, trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before each of the functions returns. Use the same notation as C Tutor.
5. Use C Tutor to trace your program one statement at a time, stopping just before your function returns. Compare your diagrams to the visualizations displayed by C Tutor. If C Tutor complains that your program is too long, delete the comments above the function definitions.
6. Do not submit your solutions for this exercise.