Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Fall 2019

Lab 10 - Implementing a Queue Using a Circular Linked List

Attendance/Demo

After you finish the exercises, a TA will review your solutions, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will grade the work you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.**

Getting Started

In a recent lecture, several different designs for a queue container were presented and analyzed. A singly-linked list was used as the underlying data structure. One design was selected (the one in which the `enqueue`, `dequeue` and `front` functions all run in constant time) and this design was implemented in C.

- Download and review the lecture slides.
- Open the C Tutor examples and review the `queue_construct`, `enqueue`, `dequeue` and `front` functions. Use C Tutor to trace the execution of these functions.

Another design illustrated how a *circular* singly linked list can be used to implement a queue. In this lab, you are going to implement a queue based on that data structure.

General Requirements

You have been provided with three files:

- circular_queue.c contains an incomplete implementation of a queue module. Several functions are fully implemented:
    - `queue_construct` 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`), along with 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.**

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 for selecting the formatting style and formatting blocks of code are in the Lab 1 handout.

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

Instructions

Step 1: Launch Pelles C and create a new Pelles C project named circular_queue. (Instructions for creating projects are in the handout for Lab 1.) If you're using the 64-bit edition of Pelles C, select Win 64 Console program (EXE) as the project type. If you're using the 32-bit edition of Pelles C, select Win32 Console program (EXE). **Don't click the icons for Console application wizard, Win32 Program (EXE) or Win64 Program (EXE) - these are not correct types for this project.**

Step 2: Download file main.c, circular_queue.c and circular_queue.h from cuLearn. Move these files into your circular_queue folder.

Step 3: Add main.c and circular_queue.c to your project. (Instructions for doing this are in the handout for Lab 1.)

You don't need to add circular_queue.h to the project. Pelles C will do this after you've added main.c.

Step 4: Build the project. It should build 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.

Exercise 0

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

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 (the rear of the queue). This node points to the node at head of the linked list (the front of the queue); that is, `queue->rear->next` points to head node. Every node in this linked list points to another node, and the tail node points to the head node; that's why this data structure is called a *circular* linked list.

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 you need to consider:

- The queue is empty.
- The queue has one or more elements (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.

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. Build the project, correcting any compilation errors, then execute the project. 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 it passes all the tests for both cases. You can then add the `assert` statement to handle the "`NULL queue` parameter" case.

Verify that `enqueue` passes all the 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`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `front` passes all the 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 you need 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.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises, assign a grade

(Satisfactory, Marginal or Unsatisfactory) and have you initial the grading/sign-out sheet.

2. Remember to backup your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service. All files you've created on the hard disk will be deleted when you log out.

Homework Exercise - Visualizing Program Execution

In the final exam, you will be expected to be able to 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 when working with queues based on linked lists.

1. Launch C Tutor (the *Labs* section on cuLearn has a link to the website).

2. Copy the `node_t` and `queue_t` declaration from circular_queue.h into C Tutor. Copy `queue_construct` and your solutions to Exercises 1 through 3 from circular_queue.c into C Tutor.

3. Write a short `main` function that exercises your list functions. 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 your queue 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 each of your queue functions returns. Compare your diagrams to the visualizations displayed by C Tutor.