

Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2017

Lab 11 - Implementing a Queue Using a Circular Linked List

Objective

Implement a queue collection that uses a circular linked list as the underlying data structure.

Attendance/Demo

To receive credit for this lab, you must demonstrate your solutions to the exercises. When you have finished all the exercises, call a TA, who will who will review your code, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will ask you to demonstrate whatever code 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

- Two sets of lecture slides that deal with queues are posted on cuLearn. Download and review these slides.
- Open the C Tutor example that contains a partial implementation of a queue based on a singly-linked list. Review the code for the enqueue and dequeue functions. Use C Tutor to trace the execution of these functions.

The last two slides in file SYSC_2006_W16_Queues_Part_2.pdf illustrate how a circular singly linked list can be used to implement a queue. In this lab, you are going to implement a queue module that uses that data structure.

General Requirements

You have been provided with three files:

- `circular_intqueue.c` contains an incomplete implementation of a queue module. Several functions are fully implemented:
 - `intnode_construct` is the same function that we've used previously to allocate and initialize nodes in a singly-linked list;
 - `intqueue_construct` allocates and initializes a new, empty queue;
 - `intqueue_print` outputs the contents of a queue on the console;

This file also has incomplete implementations of functions `intqueue_enqueue`, `intqueue_front` and `intqueue_dequeue`.

- `circular_intqueue.h` contains the declarations for the queue data structure and the nodes in a queue's circular-linked list (see the typedefs for `intqueue_t` and `intnode_t`), along with the prototypes for functions that operate on queues. **Do not modify `circular_intqueue.h`.**
- `main.c` contains a simple *test harness* that exercises the functions in `circular_intqueue.c`. Unlike the test harnesses provided in some 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). Pelles C makes it easy to do this - instructions were provided in Labs 1 and 2.

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. Don't leave testing until after you've written all your functions.

Instructions

1. Launch Pelles C and create a new Pelles C project named `circular_queue`.
 - If you're using the 64-bit edition of Pelles C, the project type should be Win 64 Console program (EXE). (Although the 64-bit edition of Pelles C can build 32-bit programs, you may run into difficulties if you attempt to use the debugger to debug 32-bit programs.)
 - If you're using the 32-bit edition of Pelles C, the project type should be Win32 Console program (EXE).

When you finish this step, Pelles C will create a folder named `circular_queue`.

2. Download file `main.c`, `circular_intqueue.c` and `circular_intqueue.h` from cuLearn. Move these files into your `circular_queue` folder.
3. You must add `main.c` and `circular_intqueue.c` to your project. To do this:
 - select Project > Add files to project... from the menu bar.
 - in the dialogue box, select `main.c`, then click Open. An icon labelled `main.c` will appear in the Pelles C project window.

- repeat this for `circular_intqueue.c`

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

4. Build the project. It should build without any compilation or linking errors.
5. Execute the project. The test harness will show that functions `intqueue_enqueue`, `intqueue_front` and `intqueue_dequeue` 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_intqueue.c` and `circular_intqueue.h` in the Pelles C editor. Carefully read the structure declarations in `circular_intqueue.h` and the code for `intqueue_construct` and `intqueue_print` in `circular_intqueue.c`. Notice how `queue->rear` points to the node at the rear of the queue (the tail of the linked list). This node points to the node at the front of the queue (the head of the linked list). In other words, `queue->rear->next` is a pointer to the node at the front of the queue (the head of the linked list). Every node in the linked list points to another node; that's why this data structure is called a circular linked list.

Exercise 1

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

```
void intqueue_enqueue(intqueue_t *queue, int value);
```

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

This function will enqueue the specified value; i.e., append it to the rear of the queue.

Design and implement `intqueue_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 contain a pointer to that node.

We recommend that you sketch some "before and after" diagrams of the queue for each of the cases before you write any code. (One diagram shows the circular linked list before the function is called, the other diagram shows the linked list 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, instead of writing the entire function before you start testing. For example, during the first iteration, write just enough code to handle the "queue is empty" case. Run the test harness and fix 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 your `intqueue_enqueue` function passes all the tests before you start Exercise 2.

Exercise 2

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

```
_Bool intqueue_front(const intqueue_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.

Finish the implementation of `intqueue_front`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `intqueue_front` function passes all the tests before you start Exercise 3.

Exercise 3

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

```
_Bool intqueue_dequeue(intqueue_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`, remove that value from the queue by deallocating its node in the linked list, and returns `true`. The function returns `false` if the queue is empty.

Design and implement `intqueue_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, sketch some "before and after" diagrams of the queue for each of the cases, before you write any code, 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 back up 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.

Submitting your Lab Work

For accreditation purposes, we archive samples of student work from one of the labs. You must submit your solutions for this lab to cuLearn. Here are the instructions:

1. First, you'll do is package the project in a ZIP file (compressed folder) named `circular_queue.zip`. To do this:
 - 1.1. From the menu bar, select **Project > ZIP Files...** A **Save As** dialog box will appear. If you named your Pelles C project `circular_queue`, the zip file will have this name by default; otherwise, you'll have to edit the **File name:** field and rename the file to `circular_queue` before you save it. **Do not use any other name for your zip file** (e.g., `lab11.zip`, `my_project.zip`, etc.).
 - 1.2. Click **Save**. Pelles C will create a compressed (zipped) folder, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder `circular_queue`).
2. Log in to cuLearn and submit `circular_queue.zip`. To do this:
 - 2.1. Click the **Submit Lab 11** link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the **Add submission** button. A page containing a **File submissions** box will appear. Drag `circular_queue.zip` to the **File submissions** box. **Do not submit another type of file** (e.g., a Pelles C `.ppj` file, a **RAR** file, a `.txt` file, etc.)
 - 2.2. After the icon for the file appears in the box, click the **Save changes** button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 2.4. If you aren't ready to do this; for example, you want to do some more work on the project and resubmit it later, you can leave the file with "draft" submission status. When you're ready to submit the final version, you can replace or delete your "draft" file submission by

following the instructions in Step 2.3, then finish the submission process by following the instructions in Step 2.4.

- 2.3. You can replace or delete the file by clicking the **Edit my submission** button. The page containing the **File submissions** box will appear.
 - 2.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the **File submissions** box, then click the **Overwrite** button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the **Save changes** button.
 - 2.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the **Delete** button., then click the **OK** button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the **Save changes** button.
- 2.4. Once you're sure that you don't want to make any changes, click the **Submit assignment** button. A **Submit assignment** page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the **Continue** button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".

Your grade for this lab will be the grade you receive when you demonstrate your solutions during the lab (S or M = 2/2, U = 0/2). The files you submit to cuLearn will not be regraded; however, 1 mark will be deducted if you don't submit the ZIP file by the deadline (the mark for an S or M grade will be reduced to 1/2).

Deadline: 11:55 p.m. on the last day of classes. Late submissions will not be accepted.

Homework Exercise - Visualizing Program Execution

In the final exam, you will be expected to be able to draw diagrams that depict the execution of functions that manipulate queues, using the same notation as C Tutor.

1. Launch C Tutor (the *Labs* section on cuLearn has a link to the website).
2. Copy the `intnode_t` and `intqueue_t` declaration from `circular_intqueue.h` into C Tutor. Copy `intnode_construct`, `intqueue_construct` and your solutions to Exercises 1 through 3 from `circular_intqueue.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 the `return` statements in each of your

list functions are executed. Use the same notation as C Tutor.

5. Use C Tutor to trace your program one statement at a time, stopping just before each `return` statement is executed. Compare your diagrams to the visualization displayed by C Tutor.