Assignment Queues: Circular Array Buffers and Priority Queues

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Implement Queue API functions.
- Introduction to the concept of managing a fixed sized buffer as a circular buffer for an array based queue.
- Learn about priority queues and how to use them.
- More examples and practice of using inheritance and class abstractions / ADT in C++.

Description

In this assignment, we will be looking at array and linked list implementations of the Queue API/abstraction you should have been learning about in this unit. We will switch back and concentrate more on the array based version of the Queue `AQueue` this week, though you will be implementing one of the methods for the linked list version as well.

As with the previous assignment, both our textbook sources give examples of a `Queue` ADT Malik 6th edition pg. 1201, and Shaffer 3.2 C++ edition, pg. 129. The `Queue` ADT we give and complete in this assignment is similar in structure to the `List` and `Stack` ADT from the previous assignments. An array based (`AQueue`) and linked list based (`LQueue`) implementations of the `Queue` abstraction are again given and used for this assignment.

A Queue is a is a first-in-first-out (FIFO) data structure. The first value put into a queue will be at the front, and values enqueued after this one line up behind it.

The array based implementation of a Queue has some interesting considerations in terms of performance. As you should be familiar with by now, we have an issue if we try and use an array for our Queue data type. If we enqueue new items onto the end of the array, this is typically a constant $\mathcal{O}(1)$ time operation (unless the array has to be grown and copied). But taking an item off of the front (index 0) of the array means we have to shift all items down by 1 index, turning the `dequeue()` operation into a $\mathcal{O}(n)$ operation.

And we can't get around this by trying to reverse which ends we use for enqueuing and dequeueing. If we use the back (last) index of the array as the front of the queue for dequeueing, then that operation becomes constant time. But to insert new values into the array at index 0 means we will have to shift values every time we `enqueue()` onto the queue. So at least one of the operations will have to be $\mathcal{O}(n)$ if we use the array based implementation of the queue in the normal way.

However, we can use an array in such a way to allow for constant time $\mathcal{O}(1)$ operations for `enqueue()` `dequeue()` and `front()` operations of our `Queue`. However, to do this we have to treat the array of values as a circular buffer instead of as a regular array. A circular buffer is a common data structure used in programs to allow a fixed sized array to work as an efficient queue. We still have the problem that if we need to add more values than the array will hold, we will have to grow the array. So `enqueue()` operations can always be $\mathcal{O}(n)$ in the worst case if we have to allocate more memory and copy the existing values to the new memory. To implement a circular buffer, we have to keep track of the current front and back index. Initially when we enqueue the first item onto the queue, the front and back index will both be index 0, the location where we place the first item. When we add more items, we add them into index 1, 2, 3, etc. So the front would still be index 0, but back will point to an item at a higher index, and we increase back index for each new item we enqueue onto the queue. But if we dequeue an item to remove it from the queue, instead of shifting values, we just increment the front index by one. So for example, lets say we enqueue items `5, 7, 3, 2` onto the queue in that order (so 5 is at the front of the queue) and then dequeue the front item `5`. We would end up with an array that looks like the following:

```
index       0      1          2          3          4   ...
values =           7          3          2
             frontIndex             backIndex
```

The array is a circular buffer, because we will wrap around the end of the array as needed. For example, lets assume that the allocation size of the array is `allocationSize = 5`, so that the valid indexes are 0 to 4 as shown in the previous example. If we enqueue two more items, `8, 1`, the array of `values` would now look like the following:

```
index        0             1          2          3          4   ...
values =     1             7          3          2          8
         backIndex     frontIndex
```

When we enqueued 8 it just ended up at index 4, the last valid index for this array. But when we enqueue 1 the array is not yet full. So we wrap back around to index 0 and this value is placed there. At this point the array is full, the `size` of the queue is 5, because it is holding 5 values. If we want to enqueue another value on this queue, it will have to be dynamically grown so we can fit more values. But if we end up dequeueing 1 or more values before we need to enqueue some more, this would make some room, and we might be able to keep going for a bit before we have to dynamically grow the queue.

So notice that, again ignoring the case where `enqueue()` needs to first grow the array allocation, both `dequeue()` and `enqueue()` operations are constant time $\mathcal{O}(1)$ operations. In both cases we simply have to increment the front or back index. Though we do have to be careful that we wrap around the end of the buffer when we increment either of these indexes. A common method to do this when incrementing the front or back index is to use modulo arithmetic. For example, in your `enqueue()` method which adds a value onto the end of the array, you might first increment the `backIndex` to the next location of the array where the new value will be inserted. You can do the following:

```
backIndex = (backIndex + 1) % size;
```

This has the same effect as explicitly testing if we have gone past the end of the array

```
backIndex++;
if (backIndex >= size)
{
  backIndex = 0;
}
```

The second example may be clearer, but they both perform the same task. In the first case, if the new value of `(backIndex + 1)` is equal to the array `size`, the remainder of dividing by `size` will be 0, which will wrap the `backIndex` back around to 0 as desired. Using modulo (remainder) arithemetic for circular buffer indexing is common and you will see it done this way often if you look at other code using an array as a circular buffer.

In this assignment you will be required to implement the `enqueue()` `dequeue()` and `front()` methods of the array based `AQueue` class. For both `enqueue()` and `dequeue()` you will need to correctly increment the back and/or front index, and wrap it around the circular buffer correctly as just discussed.

The linked list implementation of the `Queue` abstraction does not have a similar performance issue as the array based implementation. For the linked list `LQueue` implementation, it is natural to use the front of the linked list as the `Queue` front, and the back of the linked list as the back of the `Queue`. Both `enqueue()` and `dequeue()` can be done in constant time $\mathcal{O}(1)$ time when we use the linked list in this way. For example, if we enqueue on the back of the queue, as long as we keep a pointer to the last node of the linked list, it is easy to create a new node and make it the new last node. Likewise, removing the front node of the linked list is constant time if we have a pointer to the front of the linked list, as we have seen before.

You will not be implementing any of the member methods of the `LQueue` class. However, there are two additional tasks to be done once you get the missing `AQueue` member functions added. We will be adding new `PriorityQueue` implementations to the hierarchy of defined queue types. A priority queue is a queue that works similar to the queues that we discussed, but the front item of a priority queue will always be the item in the queue with the highest priority, instead of the item that has been waiting the longest. Priority can be defined as needed for the application. We will assume for our priority queue classes that items of type `T` are ordered by their priority using the normal boolean operations like `operator<()`, `operator>()`, etc. This means that the item with a higher priority will give a boolean result of true if we compare it to a lower priority item using the greater than operator.

```
T value1;
T value2;

if (value1 > value2)
{
  cout << "value1 is higher priority than value2" << endl;
}
```

You will be adding in new `enqueue()` methods for both an array based priority queue `APriorityQueue` and the linked list based priority queue `LPriorityQueue`. In both cases, you will modify the enqueue method to maintain the queue of values in a sorted order, so that the highest priority item is always at the front of the queue, and the lowest is always at the back of the queue. You will do this in both cases by inserting any new values into the queue into the correct position so that the queue is ordered by priority. In both cases this will require you to search the queue, so this means that `enqueue()` will become a $\mathcal{O}(n)$ operation for both the array and linked list priority queues. There are ways to improve on this, some of which we discussed in this units materials, e.g. keeping the items organized as a heap.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/assg-tests-AQueue.cpp`	Unit tests of the array based `AQueue` implementation of the `Queue` API
`src/assg-tests-LQueue.cpp`	Unit tests of the link list based `LQueue` implementation of the `Queue` API
`src/assg-tests-APriorityQueue.cpp`	Unit tests of the array based `APriorityQueue` implementation of the `Queue` API
`src/assg-tests-LPriorityQueue.cpp`	Unit tests of the link list based `LPriorityQueue` implementation of the `Queue` API
`include/Queue.hpp`	Header file of the ADT base `Queue` class that defines the `Queue` interface / abstraction
`include/AQueue.hpp`	Header file of the concrete array based implementation of the `Queue` abstract data type
`include/APriorityQueue.hpp`	Header file of the concrete array based implementation of a priority queue for the `Queue` abstract data type
`include/LQueue.hpp`	Header file of the concrete linked list based implementation of the `Queue` abstract data type
`include/LPriorityQueue.hpp`	Header file of the concrete linked list based implementation of a priority queue for the `Queue` abstract data type
`include/Node.hpp`	Header file of the Node structure used by the linked list implementation
`src/Queue.cpp`	Implementation file of common methods of the `Queue` base class
`src/APriorityQueue.cpp`	Implementation file for the `APriorityQueue` member functions that implement the concrete priority queue array based `Queue`
`src/LQueue.cpp`	Implementation file for the `LQueue` member functions that implement the concrete linked list based `Queue`
`src/LPriorityQueue.cpp`	Implementation file for the `LPriorityQueue` member functions that implement the concrete priority queueu linked list based `Queue`

This week you will mainly be working in the array based `AQueue` and `APriorityQueue` header and implementation files. You will add in the missing `front()`, `enqueue()`, and `dequeue()` methods for the array based `AQueue` class. Then you will implement the modified `enqueue()` methods to insert items into the queue by priority order for both the `APriorityQueue` class, and the linked list `LPriorityQueue` class.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment

Recursion' for our current class semester and section.
2. Clone the repository using the SSH URL to your local class DevBox development environment.
3. Checkout the 'origins/feedback' branch to your local working DevBox repository.
4. Configure the project by running the `configure` script from a terminal.
5. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
6. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also you should close the initial Pull request that should be automatically opened for you, so that you can open your own when committing your work.

Assignment Tasks

Task 1: Implement the `AQueue` `front()` Accessor Method

Task 2: Implement the `AQueue` `dequeue()` Accessor Method

Task 3: Implement the `AQueue` `enqueue()` Accessor Method

Task 4: Implement the `APriorityQueue` `enqueue()` Accessor Method

Task 5: Implement the `LPriorityQueue` `enqueue()` Accessor Method

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 40 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 10 points are awarded for completing each of the 6 tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often

function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.

   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.

3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U10-1 Implementing Queues
- Data Structures - Full Course Using C and C++ The videos 4:32:17 Introduction to Queues through 4:56:33 Linked List implementation of Queue are all excellent introductions to the fundamentals of implementing Queues as arrays and linked lists.
- C++ Classes and Objects
- C++ Inheritance
- C++ Templates