ECE 150: Fundamentals of Programming (Fall 2016)

Section 001: Project 2

Due on Monday, November 21, 11:59pm ET

The course project is divided over three parts, leading to a simple computer simulator. The overall project will therefore give you better insight into the operation of a computer, as well as teaching you the basics of discrete-event simulation. In this seconds part of the part of the project you are required to design and code a priority queue that will be used to manage the simulated events.

Learning Outcomes

After completing this part of the project, students should be able to:

1. Design and build a simple linked data structure, ordered by priority
2. Develop basic test cases for your code

Requirements

Submission: You should submit the result of this exercise to the project `150-Project-EventQueue`. Your submission file must be named `eventQueue.cpp` (case sensitive).

Important Note #1: We will not be opening the Marmoset submission for any of the project submission requirements until one week before the submission is due. In the case of this part of the project, this means the submission option will not be made available on Marmoset until November 14th. This is intended to encourage students to develop their own test cases and to do coding and testing on their local machine, getting the basic functionality correct, before submitting it to Marmoset for formal testing.

Important Note #2: In this part of the **you will develop a source code file with only the required C++ functions**. Within your source code file (`eventQueue.cpp`), it should have the statement

```
#include "eventQueue.h"
```

which we will provide. `eventQueue.cpp` **should not include a** `main()` **function**.

Important Note #3: We will also provide a separate file, `eventQueueTester.cpp`, which will contain a sample `main()` for how your two functions will be used. It is expected that you will change `eventQueueTester.cpp` to create various tests of your own to test your eventQueue functions.

Important Note #4: Do not submit `eventQueue.h` nor `eventQueueTester.cpp`. You should only submit `eventQueue.cpp` to marmoset for testing.

Task Description

In a simple linked list, items can be appended to the linked list, in which case they are added to the end of the list, or prepended, in which case they are added to the front of the list. For example, suppose we have the linked list:

```
FrontOfList → (item A) → (item B) → (item C) → EndOfList
```

and we append item D to the list. Then the list will be:

```
FrontOfList → (item A) → (item B) → (item C) → (item D) → EndOfList
```

If we then prepend item E to the list, we then have

```
FrontOfList → (item E) → (item A) → (item B) → (item C) → (item D) → EndOfList
```

We may be able to find and remove items from the list. For example, we may want to remove item C from the list, which will result in:

```
FrontOfList → (item E) → (item A) → (item B) → (item D) → EndOfList
```

Instead of finding a specific item in a linked list and removing it, we might simply want to remove the first item from the list. In that case, we have:

```
FrontOfList → (item A) → (item B) → (item D) → EndOfList
```

When we restrict a linked list such that the only way we can add items to the list is by appending the item to the end of the list, and the only way we can remove an item from the list is by removing it from the front of the list, we give the linked list a special name: a queue.[1] The operations on such a linked list are then:

```
enqueue(item)
```

and

```
item = dequeue()
```

`enqueue()` will append the item to the list, while `dequeue()` will take the item off the list (and return it to the calling code). With such a system, we would not be able to prepend item E to the front of the list, nor could we remove item B from the middle of the list. So instead we might have our initial queue:

```
FrontOfQueue → (item A) → (item B) → (item C) → EndOfQueue
```

If we `enqueue(item D)` and `enqueue(item E)`, we have:

```
FrontOfQueue → (item A) → (item B) → (item C) → → (item D) → (item E) → EndOfQueue
```

And if we perform a `dequeue()` operation, we will give `item A` to the calling code, and the queue will now contain:

```
FrontOfQueue → (item B) → (item C) → → (item D) → (item E) → EndOfQueue
```

It is often the case that we do not want the items in a queue to be ordered simply according to the order in which they were added to the queue. For example, imagine having homework tasks to complete. If you had a simple queue managing them, you would add the items to your homework queue as they were given to you. Whenever you finished an assignment, the next assignment you would work on would be the one at the front of the queue. In other words, you would do the assignments in the order in which they were assigned. This is probably a bad strategy. A better strategy is to do the assignments in the order of the due date of the assignment. So you still put each new assignment into a queue, but the queue is ordered according to the due date of the assignment. The earlier the due date of an assignment, the closer it is to the front of the queue. For example, if we have a homework queue like this:

```
FrontOfQueue → (ECE 106; Nov 14) → (CHE 103; Nov 16) → EndOfQueue
```

Now if you are handed an assignment in MATH 117 that is due on November 11th and you `enqueue(MATH 117; Nov 11)`, the queue would now be:

```
FrontOfQueue → (MATH 117; Nov 11) → (ECE 106; Nov 14) → (CHE 103; Nov 16) → EndOfQueue
```

Now when doing a `dequeue()` operation to get the next piece of assigned homework on which to work, we will dequeue the Math assignment, due on the 11th rather than delaying that work until after both the ECE and CHE assignments have been completed, as would be the case with a simple queue.

In the same way as we want to do our homework in the order in which is it due, in discrete-event simulation we want to process simulated events in the order in which they will occur in our simulation. To describe how this is done, we must first understand what an event is within a discrete-event simulation. Very simply, an event is something that occurs at a given time, and as a result of occurring, the state of the system may change and other events may occur in the future. The nature of the change in the system and the new events that occur will be the subject of part 3 of the project. For this part of the project it is sufficient to know that events therefore have a "time" at which they occur and other information, including the type of the event. For the purpose of this part of

[1] Some people use the term "FIFO" which is an acronym for First-In-First-Out, especially in computer hardware, but the term "Queue" is the common term in software.

the project, we have strictly limited the definition of an event to just those two pieces of information, and we encode them in a C struct as follows:

```
struct Event {
     float     time;          // Simulated time of event
     EventType type;          // Event type
};
```

where the `EventType` is defined as an enumerated data type:

```
enum EventType { LOADSTORE, ALU, COMPAREJUMP };
```

For this part of the project, you are required to implement the functions

```
bool enqueue(const Event* e);
```

and

```
const Event* dequeue();
```

as a priority queue, where the queue is ordered by the value of "time" in the events being stored in the queue. Note that the event stored is "const" because the queue is to store the event, not change it. Space for the event will be created and deleted by the calling program (`eventQueueTester.cpp` is provided as a sample), but any storage space you need to store the event will be your responsibility.

The `enqueue()` function returns a `bool`, which should be `true` if the event was enqueued and `false` otherwise. The kinds of conditions under which you would ***not*** be able to enqueue an event are unlikely to occur during program testing (*e.g.,* the system has insufficient memory).

The `dequeue()` function returns a pointer to the event at the front of the queue, removing it in the process. If there are no events left in the queue, the `dequeue()` function returns 0 (*i.e.,* the NULL pointer).

These functions are to be implemented in a file `eventQueue.cpp` with any supporting functions you choose to implement as part of that. The header file and a sample testing program are available on Learn with this project description.