Lab 05: Efficient Queue with Dynamically-Allocated Array

Overview

This lab will practice efficient circular queues using a dynamically-allocated array.

Circular Queue

Queues are data structures that follow the FIFO (first in, first out) rule to control its enqueue and dequeue behavior.

In the last lab, you implemented a queue on top of a dynamic array. It was able to efficiently insert at the back of the queue, but when dequeuing, the entire list had to be shifted. Ideally, we can avoid this shifting behavior to make the dequeue operation more efficient. For today's lab, you will build on the dynamic array queue from last time, but you are going to create a much better dequeuing operation.

Recall that in lecture, we gave the example of a queue based on a static array, one which had front and rear pointers to track where to enqueue and dequeue in the list. This structure, which can be thought of as a loop or "circular" queue, can also be built on top of a dynamic array.

Hints: You are given a functioning dynamic array queue, the solution to the last lab, in Mimir's **starter code**. The notes for **Lecture 5, Abstract Data Types and Queues** provide pseudocode for a circular queue based on a static array.

Submission

You will need to implement four standard functions of the `Queue` data structure:

```
1. bool empty() const;
2. void enqueue(type elem);
3. void dequeue();
4. const type* front() const;
```

The `empty` function returns `true` if the queue is empty or `false` otherwise.`Enqueue` takes an element as input parameter and adds such an element to the back of the queue. `Dequeue` removes the element at the front of the queue. The `front` function returns a pointer to the element at the front of the queue.

You should also implement public getter methods to expose some of the private variables you may use for internal tracking. This is used to help our automatic test scripts to see the underlying logic of your code. The following methods should be available:

```
5. const T* getData() const;
6. unsigned getCapacity() const;
7. unsigned getSize() const;
8. unsigned getBackIndex() const;
9. unsigned getFrontIndex() const;
```

The `getData` function returns a constant pointer to the underlying dynamic array used to carry data stored in the queue. `getCapacity` should return the allocated size of the underlying dynamic array. getSize exposes the actual size used at a time. `getBackIndex` and `getFrontIndex` should return the index of the back and front elements of the queue. Note that for empty queues the back and front indices are not tested, since it can vary depending on implementation choices.

Also for testing purposes, **you should reallocate your internal array by 1 every time it's needed**. Having a constant allocation offset helps automated test scripts to verify your solution more quickly. When reallocating, make sure to rearrange the new array so that the front is at the beginning and the back is at the end of the array. Remember to properly deallocate unused dynamic arrays to avoid memory leaks. Memory leak is one of the required tests on Mimir.

Hints: You need to reallocate the underlying array when enqueueing new values if the current capacity is not enough to hold all the elements. You should also be mindful of edge cases such as not allowing dequeue to throw an exception with an empty list.

Complete the "Lab 05. Efficient Queue with Dynamically-Allocated Array" activity on Mimir. Differently from Lab 04, you should include both the header (Queue.hpp) and the source file (Queue.cpp) for your efficient queue class**.**

You will implement four standard functions for the efficient circular `Queue` data structure, and five getter functions. A battery of automated tests will be run to ensure that the errors have been fixed and the results can be properly computed.

A grade of "complete" on this lab work requires a score of 100%.