Project 7 Documentation

Purpose of the Program

==================

The core objectives of this project are to test our ability to create and make use of queue-based dynamic data structures, and to generalize our code using templates. A review of knowledge on working with Queue structures along with manipulation of dynamic memory, pointers, iostream to all extents, and classes is also a central part of the implementations of the project in its source code. We accomplished this by implementing two types of queues, one which is array-based while the other is node-based.

Program Design

=============

The program works to implement the functions and member variables of a unique templated Queue class with an Array-based and Node-based variant. As a First In First Out (FIFO) structure, the Queue inserts data at the back (push) and removes data from the front (pop). The Queue’s front data member points to the first inserted element (the front one), while the back data member points to the last one inserted (the rear one). These Queues work with DataType class objects. The ArrayQueue source code consists of a header file called ArrayQueue.h, which stores all of the specifications and actual implementations that are required for the ArrayQueue, including function prototypes and member variables. NodeQueue’s source code consists of a header file called NodeQueue.h, which stores all specifications and implementations that are required for the NodeQueue, including all function prototypes and initialized member variables.

The ArrayQueue class contains the private data members m\_array, a DataType-type pointer which always points to the dynamically allocated data for any ArrayQueue object, m\_front, an int containing the m\_array index of the front (first) element of the Queue, m\_back, an int containing the m\_array index of the back (last) element of the Queue, and m\_size, an int that counts how many elements are currently stored in the Queue. The NodeQueue class contains the data member m\_front, a templated Node-type pointer that always points to the first Node element of the NodeQueue data structure as well as the data member m\_back, a templated Node-type Pointer that points to the last Node element of the NodeQueue.

The ArrayQueue class has the following public member functions: default constructor, parameterized constructor, copy constructor, destructor, insertAfter, insertBefore, removeForward, removeBackward, find(), front(), back(), size(), empty, full, and clear. The default constructor will instantiate new ArrayQueue objects with no valid data, while the parameterized constructor instantiates new ArrayQueue objects that are initialized to hold a Queue with certain number of elements with a default value. The ArrayQueue copy constructor will instantiate a new object that acts as a separate copy of the data of the ArrayQueue object being copied, while the destructor destroys the instance of the m\_array. Size() will return the size of m\_array, front() returns the first element of the ArrayQueue object, and back() returns the last element of the m\_array. Empty() will check to see if the calling Queue object is an empty Queue object, clear() clears the m\_array of an ArrayQueue object, while full() checks to see if the calling Queue object is full (by default it always returns false). Push will insert an element of a given value at the back of the Queue, while pop removes from the first element of the Queue with each call. Finally, ArrayQueue has a friend function operator<< that outputs the data for a ArrayQueue object to the file or terminal depending on the type of output stream variable passed in as its parameter.

The ArrayQueue.h file is solely responsible for the implementation of the ArrayQueue class member functions that are first specified in the same file. In addition to these source code files, the proj7.cpp source code file acts as a test driver that demonstrates the functionalities of the ArrayQueue class. The test driver does this by using a variety of Queues and calls to each of the public member functions in the ArrayQueue class one-by-one in code sections tagged by the number corresponding to each function. The NodeQueue class essentially has the same functionality as the ArrayQueue class, with identical functions that are responsible for identical tasks. The key difference is that all these functions have been rewritten to perform the same operations on linked Queue data structures that use Nodes instead of allocated arrays. The NodeQueue also has pointers to both the front and back elements of the Queue.

Problems of Program

================

During the process of coding for the program, there were some issues that turned up concerning exact details of implementation for the copy constructors of the ArrayQueue and NodeQueue class that had been responsible for creation of object copies. In addition to this, there were some problems and issues about the implementation of the push function for the NodeQueue class as well as other functions in the ArrayQueue and NodeQueue class such as the pop function responsible for dequeueing the first element of a Queue object with each call along with its incorporation into the implementation of the overloaded output operator for both ArrayQueue and NodeQueue since there was an initial mismatch in the function parameters being passed from the output operator to the pop function. However, with time these issues with implementations of the class functions were resolved in due time with diligence and persistence in the debugging process, especially with proper initialization of pointers and the use of temporary variables.

During the conversion process of the source code from its base version to its template format, there were some other issues that came up with regards to template class and function syntax, most notably with the output operator and the NodeQueue functions that were required to access the m\_next member variable of any Node class object. The format of the template function declarations and implementations was quite confusing at times to understand, due to the specificity of template syntax needed to properly generalize the functions of a class for use by multiple data types. With the proper formatting though, as well as some additional declarations of the NodeQueue class to grant access to the m\_next member variables of Node class objects, these issues were also resolved in due time, leading to successful source code execution.

Changes to Program

================

I would have liked to double check the parameterized constructor for the NodeQueue class once more, just to make sure that it could have worked and been tested properly using the test driver code, despite the fact that the logic behind the parameterized constructor for NodeQueue is sound and works in theory. With the repairs to the code of the parameterized constructor for NodeQueue, I would also liked to have made some adjustments to the functions of the templated classes of ArrayQueue and NodeQueue that could have improved their efficiency and usefulness in function for multiple types of scenarios if possible.