Bryan Kline
Project 10 Description
11/18/15

Project 10 is the continuation of a series of projects on data structures which is focuses on queues. There are three types of queues that are the focus of this project, two array-based queues, front stay and front wrap, and one node-based queue. Queues are structured, as the name implies, in such a way that items are queued one after the and those added or enqueued first are removed or dequeued first and those enqueued last are dequeued last. Array-based queues are arrays and as such have a predetermined maximum length and the elements in the array can move around in the array while node-based queues are queues of nodes where nodes are like containers that hold the data and a pointer to the next node and so have no maximum size. Queues, unlike stacks with their top, have both a front and a rear. This project consists of only the implementation files (no header, main driver, or makefile) for the different types of queues which were written in C++ using gedit and complied and run in Ubuntu/Xubuntu with Valgrind and are free of memory leaks. The documentation for these files will be broken into three distinct parts corresponding to the different files, queue1.cpp, queue2.cpp, and queue3.cpp, the array-based front stay, array-based front wrap, and Node-based queue implementations, respectively.

Array-based front stay (queue1.cpp):

The array-based Queue class header contains the data members front, rear, and max which are ints that correspond to the front of the queue, the rear of the queue, and the max size of the int array, as well as an int pointer that points to the array of ints that make up the queue. The array-based Queue class has a default constructor, a copy constructor and a destructor and its functions are enqueue(), dequeue(), empty(), full(), and clear() and the overloaded operators for assignment and equality which are member functions and an overloaded insertion operator which is a friend function. The constructor takes in an int as a parameter and, if it's greater than zero allocates memory of that size for the int array “data” and sets data members to default values. The copy constructor takes in another Queue class object, allocates memory of that size, and copies in its values into the new Queue. The destructor frees memory allocated for “data” and sets data members back to default values.

The function enqueue() takes in an int as a parameter and first checks that the Queue isn't full with a call to full() and if it's not then “rear” is incremented and the element in “data” at that spot and sets it equal to the int passed in and returns true if it was successful and false if it wasn't. The function dequeue() takes in an int by reference and if the Queue isn't empty then the int passed in as a parameter is set equal to the element at the front of the Queue, a counter controlled loop moves all elements toward “front” and “rear” is incremented. Finally, true is returned if it was successful and false if it wasn't. The function empty() checks if “data” is NULL or if “rear” is -1 and returns true if either are true and returns false otherwise. The function full() checks if “rear” is equal to “max” minus one and returns if it is and false otherwise. The function clear() checks if the array is empty and if it isn't then sets “rear” to -1 and returns true, otherwise it returns false.

The overloaded assignment operator works in much the same as the copy constructor, with the exception that if first checks that the Queue passed in as a parameter isn't “this” dereferenced and that it returns “this” dereferenced, otherwise they are very similar in that the calling Queue has its int array memory freed, then new memory of the size of the other Queue allocated to “data” and then a counter controlled loop goes from “front” to “rear” and copies in the values from one to the other. The overloaded equality operator first compares “front”, “rear” and “max” of each Queue and returns false if either doesn't match, then uses a counter controlled loop to go through each array and compares each value, returning false if any aren't the same and finally returning true if all are equal. Finally, the overloaded insertion operator first checks that the Queue isn't empty, if it is it just prints out a new line, and then uses a counter controlled loop to go from “front” to “rear” and prints each int in the array to the screen, putting brackets around the values at “front” and “rear”.

Array-based front wrap (queue2.cpp):

The default and copy constructors, the destructor, member functions, and overloaded operators are much the same for front wrap as in front stay, the main difference being that “front” and “rear” can move to the end of the int array and wrap around to the beginning of the array. The way in which “front” and “rear” wrap around to the beginning of the array is by setting them equal to their current value plus one modded by the size of the array, “max”. This is effectively the same as setting their value to zero if their current value is “max” minus one, but as the project specification indicates that if statements aren't to be used and that the modulo operator should be used instead it is done this way. There is an if statement in the enqueue() and dequeue() member functions but this is only to check that the Queue isn't empty or full first. In the case of dequeue() there is an additional if statement checking if “front” and “rear” are equal, if so then there is only one element in the array and then both can be set to -1.

The other functions are much the same, but one notable difference is full() in which if statements are used to determine the position of “front” and “rear” and whether or not the Queue is full. Because the constraint of using the modulo operator was only specified for the enqueue() and dequeue() functions I wanted to use if statements here merely for practice with a different way of determining where the “front” and “rear” were after having potentially wrapped around to the beginning. This is done by first checking if “front” is less than “rear” and if so the only time that this is the case and the array is full is if “front” is at the first index and “rear” is at the last, so true is returned if this is the case and false otherwise. If “front” is greater than “rear” then the array is full when the difference of the two is one, and in that case true is returned otherwise false is returned. Finally, if the size of the array is one and “front” and “rear” are equal and not both -1 then the array is full and true is returned. For all other cases false is returned.

The implementation for the rest of Queue front wrap is, as stated previous, much the same for all functions with the exception to counter controlled loops that go through the array. Instead of starting at “front” and ending at “rear” and incrementing by one they start at “front” and end at “rear”, but are incremented in the same way as “front” and “rear” wrap, by adding one to the counter and modding it by max.

Node-based (queue3.cpp):

Just as in Project 9, the Node based Queue has a separate header file from the array-based Queues which contains a class specification for the Queue class as well as the Node class. The Queue class has the same constructors, destructor, member functions and overloaded operators, but instead of ints for “front”, “rear”, “max” and an int pointer for “data” it just has two Node pointers, “front” and “rear”. The Node class has only a copy constructor and two data members, and int “data” and a Node pointer “next”. The implementation for the Node-based Queue class is much like that for Stacks, the default constructor sets both pointers to NULL, the copy constructor takes in a Queue and while it's not pointing to the “rear” a temp pointer moves through the Queue and makes new Nodes and links them together, starting at “front” and ending at “rear”, while the destructor goes through the Queue with a temp pointer while it's not at “rear” and deletes Nodes and then sets “front” and “rear” to NULL.

The member function enqueue() takes in an int “number” and first checks that the Queue is not full with a call to full() then with a temp pointer moves through the Queue until it reaches the end, creates a new Node with “number” and NULL passed to the Node's copy constructor, then makes “next” of the “rear” Node point to it, then sets “rear” to that Node. True is returned if it was successful and false if it wasn't. The function dequeue() takes in an int by reference and if the Queue isn't empty then a temp points to “front”, “front” is moved to the next Node and the the Node at temp is deleted. If the Node was the only one in the Queue then both “front” and “rear” are set to NULL. The function empty() checks if both “front” and “rear” are NULL and if so true is returned, otherwise false is returned and the function full() simply returns false as a Node-based Queue can't be full, assuming the is sufficient memory on the machine. The function clear() works the same way as the destructor, it goes through the array and, until a pointer gets to the rear, the Nodes are deleted and “front” and “rear” are set to NULL.

The overloaded assignment operator works nearly the same as the copy constructor, except that it checks that both Queues aren't the same, calls clear() to delete the memory of the calling Queue, and “this” dereferenced is returned. It goes through the Queues with temp pointers and copies in one into the other, creating new Nodes as it goes and passing the values into the Nodes' copy constructors. The overloaded equality operator first checks if both Queues are empty and returns true if they are, if they're not then using temp pointers it goes through the Queues and compares the “data” values in each Node, returning false if at any point they are not equal. Finally, it checks that both are at the end of the Queue, if one isn't then false is returned, and then if both are true is returned. The overloaded insertion operator first checks that the Queue isn't empty, if it is it prints a new line, and if not then it uses a temp pointer to go through the Queue Node by Node and prints out their “data” values to the screen, putting brackets around “data” of the “front” and “rear” Nodes.