CIRCULAR AND LINKED QUEUE STRUCTURES

Fordham University

Department of Computer and Information Science

CISC 2200: Data Structures

Instructor: Prof. H. M. Ammari

Student: Destin Piagentini

Fall 2018

Table of Contents

Introduction

This program provides a queue data structure through three separate implementation forms: a circular array, a linked list, and a circular linked list. The purpose of implementing a circular array is to bypass the need to run at O(n) calculations, and, instead, runs at O(1) calculations. This is achieved as the queue itself does not read from index 0 to index MaxLength – 1, but, instead, is able to maintain a queue anywhere in the array by keeping track of a starting and finishing index. In this sense, the array is "circular" because it has no defined start and finish, but is determined by the index variables keeping track of the array. For instance, if we wanted to enqueue an element, we would push the LastIndex variable over one, and add that element in that index, and vice-versa if we wanted to dequeue. In this sense, elements get overwritten, and there is no need to push all elements down by one every time we want to dequeue, as we would be forced to do in a traditional array. The linked lists are a little bit different. Because we can not access individual indexes, we have to shuffle through the list to find the rear of the list if we want to dequeue, or, in the case of the circular linked list, to adjust the pointer circling back to the head. These implementations run at O(n).

In summary, the circular array has a wildly different (and, perhaps, more complex) implementation from the linked lists, but is proven to be a more efficient data structure for queue management, running at O(1) as opposed to O(n).

Design and Solution

Approach and Evolution of the Design

My initial approach on starting the design of this program was to think of how a circular array would work in theory, and how that would translate over to the program. The circular array was, by far, the most difficult portion of this project as I was already familiar with the implementation of linked lists and circular linked lists due to experimentation with them via homework problems and personal experimentation. To better understand the circular array, however, I had to draw out the array on a piece of paper. I quickly realized, since the circle had no beginning or end, there was need for markers (index variables) to keep track of the theoretical start and end of the array. Once the need for index variables was clear, I was able to see how they moved around the circle as elements were enqueued/dequeued. I recognized that there was no need to push any elements over, as we could simply leave them as they were, and if the case arose where we would need to use a space that once held an element, we could simply overwrite that space (however, for user clarity in Developer Mode, the program changes dequeued elements to 0). Once the concept was clear, the implementation was very simple.

I started working on the class interface, and I tried to imagine what types of functions would be needed for the queue implementation. There was, of course, the constructor(s) (one default, the other taking in length and Developer Mode setting) and the destructor. Next, I knew there needed to be an Enqueue ad Dequeue function, of course. Finally, I knew I needed visualization functions to draw the queue for the user (there are two visualization functions – one in a queue-based image, and the other as part of the developer mode). A little bit later, I

understood the need for functions to check whether the queue was full or empty so that many other function preconditions could be checked. The Linked List Implementations required the same functions and one extra, MakeEmpty(), to be called in the destructor to deallocate the entire list (in a dynamic array it is as simple as using the delete keyword).

I had decided early on to include a setting called “Developer Mode” that would show the data structure as it was, to help with debugging and to prove that the program worked correctly.. This helped me see which area of the array was currently part of the circle, and which areas were not, as well as how the “circle” moved through the array.

The implementation started with the constructors, which created the dynamic array, initialized all variables, and initialized all elements of the array to 0 (for readability in Developer Mode, so those viewing in Developer Mode could keep track of what is used and what is not). My next step was to write the visualization functions so I could test my Enqueue and Dequeue functions as they were being written. This was as simple as a couple of print statements (and loops to cycle through the array). Next, I wrote the enqueue and dequeue functions. Since I had already grasped the basic concept on paper with the circular diagram, the implementation was actually very easy. A little trial and error made sure the proper adjustments were made, and the functions were complete relatively quickly. Finally, I created the IsFull and IsEmpty functions. IsEmpty, is a simple function, as it uses a variable keeping track of the amount of queued elements to determine if the array is empty (if the variable is 0). We could not use the indexes because the two indexes remain on top of each other whether the list is empty or only one element is queued. For the IsFull function, it would be easy to check if the amount of queued elements was equal to MaxLength, however, I really wanted to practice my understanding of the implementation so I chose to use a method that checked if the LastIndex was one before the first

index, in other words, there was no extra space (if the two indexes were at opposite ends of the array, the function could also account for this).

The process was the same for the linked list implementations. Since I was already more familiar with those implementations, I had an easier time constructing them. In these implementations, the newly enqueued element became the new head of the list, with the oldest element being in the back of the linked list. This means for the visualization, we could easily cycle through and print. However, when we wanted to dequeue (or, in the case of the circular linked list, dequeue or enqueue), we would have to cycle to the rear. This is where the O(n) time complexity stems from. However, these implementations were much easier thanks to my prior knowledge.

When The program was all set and finished, however, it became apparent that my visualization of the queue (placing the new element in the front) was an incorrect representation of the circular array. When I realized this, I had to rework how each of the data structures operated so that a queue structure was correctly displayed. For the circular queue, where it originally printed the newest element in the front and the oldest in the back, I chose instead to display the array as it was, but to block out empty spaces with the "*" character. However, since the Dequeue function (and the initialization of the array) sets empty spaces to 0, there may be confusion if the user decides to enqueue the number 0. So, in our Visualization function, we have to case handle. If there are one or fewer elements, the function checks if the array is empty, and if it is not, prints that index's values and empty spaces for the other elements. When the two index variables are different, the function checks to see if that index is within the range of indexes active in the circular array. If it is out of that range, it is printed as an empty space. If it is within that range, the index's value is printed. The function checks for both cases, whether the

array is arranged in a way where the LastIndex is at an index lower than FirstIndex (when the array circles around), or if LastIndex is at an index higher than FirstIndex.

The linked list structures were also updated to accommodate this change. Instead of placing the new node at the head, the oldest node remained the head and the newest node was placed at the end of the linked list or circular linked list each time. To dequeue, the ehad was removed and the next element became the new head. In this sense, the queue could correctly print just by running through the linked lists.

Brief Summary of Each Function

Note: Each header will explain the process for each implementation to avoid rewriting the samer functions three times.

Constructors

In each implementation, the role of the constructor is to initialize the list with the correct variables. The constructor is called when a class object is created. For the circular array, the indexes and element counter are all initialized to 0, the length variable is set equal to the requested input, and the Developer Mode variable is set appropriately (in the default constructor, inputs are given generic values). This function also initializes all dynamic array elements to 0, to ensure uniform values are presented in Developer Mode to the user. For the linked lists, the constructors are as simple as initializing the head pointer to null, and initializing the Developer Mode setting correctly. The constructors here use a default parameter for the Developer Mode setting, so they fill the qualification as a default constructor as well. The linked list constructors run at a time complexity of O(1), while the circular array runs at O(n) where n is the max-length to initialize each index to a value of 0.

Destructors

The job of the destructor in all three implementations is to deallocate any stored memory on the heap. The destructor is called when the class object goes out of scope. For the circular array, this is as simple as deleting the dynamic array variable. For the linked list, the MakeEmpty() function is called (described in detail further down). All destructors run at a time complexity of O(1).

```
void Enqueue(int Value);
```

The primary job of the Enqueue function is to add a new value to the theoretical "end" of the queue. The precondition for each implementation is that the list is initialized and is not full. The postcondition is that the queue will gain a new member (the value of which is determined by the user). In the circular array, this function performs by moving the LastIndex over one to the right (or circling back around if at the edge), adding the new element in that position, and updating the QueuedElements counter. In the linked list implementation, the function simply adds a new node to the end of the list, and readjusts the list's previous rear to connect with the new rear. The function uses a pointer to traverse the list to find the rear node so that it can be connected to the new node. If the linked list is empty, however, a new node will just be created and Head will point to it. In a circular linked list, the idea is the same, and the new rear node now points to the Head node, or, if the list is empty, a new node will be created and made to point to itself. The function runs at a time complexity of O(1) for the circular array implementation, but O(n) for the linked list and circular linked list implementations, where n is the length of the linked list, because we need to shuffle to the rear node to place the new node after it.

```
void Dequeue();
```

In the dequeue function for each implementation, the function's job is to remove the oldest member. The precondition for this function is that the list is initialized and not empty. The

post condition is that the oldest member of the queue will be removed, and the proper adjustments will be made to ensure the queue still performs smoothly. In the circular array, the element at FirstIndex is the one to be removed, so it will be set to 0 and the FirstIndex variable will be incremented to no longer include it in the list (or circled back around to the front if FirstIndex is at the end of the dynamic array). The QueuedElements counter will also be decremented to reflect this change.

In the linked list structure, the Head pointer will simply be deleted, and Head will be made to point to the next element. If the Head was the last element, it will be made to be null (which is already the last element's Next pointer). In a circular linked list structure, the function must shuffle to the rear node in order to ensure the rear node will point to the new Head after the current Head is deleted. Once the rear node is found, the Head pointer will then readjust to be the next node after the original Head. The rear node will then connect to this new Head, maintaining the correct circular structure. If there is only one node, however, the node will simply be deleted and the Head pointer will be made to point to null. The time complexity of the circular array implementation and normal linked list implementation is O(1), while the circular linked list implementation run at O(n), where n is the number of elements in the list (due to having to cycle to the end of the list).

```
bool IsFull() const;
```

The purpose of the IsFull function is to determine whether or not there is still sufficient capacity in the queue to add elements. The precondition is that the queue is initialized, and the postcondition is that the function will determine whether the list is full or not, but the queue will remain unchanged. In the circular array, the function checks to see if the LastIndex is one index before the FirstIndex (the indexes span the whole array), or if the LastIndex is in the last index of

the actual array while the FirstIndex variable is at the front index. In other words, the index variables are on opposite ends, but spanning the entire array still. For the linked lists, the procedure is the same. The program attempts to create a new node, and, if there is no compiler error, there is still memory to be allocated. Otherwise, if there's an error, there is no more space t be allocated. All implementations run at O(1).

```
bool IsEmpty() const;
```

The purpose of the IsEmpty function is to check whether the queue is empty or not. The precondition is that the queue is initialized, and the postcondition is that the queue is determined to be empty or not, but the queue is unchanged. In the circular array, the QueuedElements counter is checked. If that counter is equal to 0, the list is empty. Otherwise, in the linked list implementations, if the Head pointer points to null, the list is empty. All implementations run at a time complexity of O(1).

```
void Visualize() const;
```

The purpose of this function is to display the queue elements in a clearly readable queue form. The precondition is that the queue is initialized, and the postcondition is that the queue's elements are printed, but the queue is unchanged. In the circular array structure, the array is printed as it is. If the space is empty, however, there will be an asterisk character in that spot. An empty space in the array holds a value 0, however we can not simply print that space as empty if the value is 0 because a user may enqueue any given index with a value of 0. When the function sees that an index's value is 0, the function determines whether that spot is empty or if it is truly of value 0 (entered by the user) by checking the indexes. If the index is within the active index range for the circular array, it is not empty. Because the indexes may circle around the array, we have to case handle for the case of LastIndex being of a lower index than FirstIndex (meaning

the array circles around), or LastIndex being of a higher index than FirstIndex. In either case, if the index is out of range, it is printed as an empty space. If the two index variables have the same value, however, it can mean that the array is either empty or contains only one element. If the array is empty (checked by the IsEmpty function) or not the same index as the two index variables (meaning it is out of range since an element can only be at the spot indicated by the variables), it is determined truly empty and printed with a "*". However, if the array is not empty and the index in question is that of the two variables, that space is truly filled with a value of 0, and printed as such. For the linked list, the function starts at the head node, and cycles until the rear node (determined by if the node has a null pointer or it circles back to the Head respectively). These functions run at a time complexity of O(n), where n is the number of members in the queue.

```
void DeveloperVisualize() const;
```

The purpose of this function is to provide a visualization that helps clearly represent what is going inside theoretically. The precondition of this function is that the queue is initialized, and the postcondition is that a diagram based on the queue is printed (along with some data), and the queue is unchanged. In the circular array, the array is printed as it appears in memory without indicating what space is truly empty or not, and the FirstIndex and LastIndex values are printed. For the linked lists, a diagram of the links showing node values and pointer visualization is printed. For the circular linked list, several memory addresses concerning the rear and Head nodes are printed to prove that the list is circularly interlocked, and a linked list diagram is printed to show a circular connection. This function runs at a time complexity of O(n), where n is the length of the queue, across all implementations.

```
void MakeEmpty();
```

This function's purpose is to deallocate memory in a linked list. This function is exclusive to linked list implementations. In a normal linked list implementation, this function will delete the Head via a temp pointer, set Head to the next node, and so on until Head finally is null. In a circular linked list, the process is basically the same, but the queue is checked to ensure that it is not empty first. Additionally, in the circular implementation, the rear node is first found and made to point to null instead of Head, so the following while loop can run until Head is equal to null, just as in the normal linked list implementation. This function runs at a time complexity of O(n) where n is the number of elements in the list.

A Brief Overview of the General Program Loop

The program starts by welcoming the user and explaining the data structures and what they do, as well as the Developer Mode. Then, the program takes the user to the main menu after a key press. From the main menu, the user can enter to any of the three implementations, or go to the settings menu to toggle Developer Mode, or quit. If the user chooses the circular array, the user can decide how long they would like the circular array to be. From any implementation menu, the program gives the user the option to enqueue, dequeue, or quit to the main menu. If the user decides to enqueue, the user will be prompted to enter a numerical value to enqueue. The queue will update in real time every option the user choices, and clears the screen each update to ensure the screen doesn't become cluttered. If the Developer Mode setting is on, the Developer Mode visualization will display.

How I Know the Program is Implemented Correctly

I'm confident that the program is implementing these implementations properly because of the Developer Mode. For the circular array, Developer Mode prints the actual array as is, along with the index variables, so one could get a good sense of what is going on with the array

itself. The circular array correctly implements a theoretical circular structure through the ways in which the index variables change and load the array, as shown by the real-time updates in Developer Mode. The linked list implementations build a full linked list diagram in Developer Mode, and it is easy to see each node's value and where it points. For the circular node, additional information such as certain references in memory are displayed to show that the list is truly circular.

Implementation

Note: some comments or code may be edited from its original format to be more readable in a word processor (namely, Microsoft Word).

CircularQueueStructures.cpp (Main file)

```
// CircularStructures.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

/*

	Destin Piagentini
	Professor Ammari
	Data Structures

	References: https://mathbits.com/MathBits/CompSci/Introduction/clear.htm

	References: https://stackoverflow.com/questions/14669457/initialization-of-element-is-skipped-by-case-label


*/

// CircularStructures.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

/*

	Destin Piagentini
	Professor Ammari
	Data Structures

	References: https://mathbits.com/MathBits/CompSci/Introduction/clear.htm

	References: https://stackoverflow.com/questions/14669457/initialization-of-element-is-skipped-by-case-label


*/

#include "CircularArray.h"
#include "LinkedQueue.h"
#include "CircularLinkedQueue.h"
#include <iostream>
#include <stdlib.h>

int main() {

	//Initialize DeveloperMode to true.
```

```
bool DeveloperMode = true;

//Variable which will record the user input in menus.
char UserChoice;

//Instructions
std::cout << "Welcome to the Queue Creator!\n\n";
std::cout << "A queue is a type of datat implementation in which the first
elements put in are the first to be removed.\n\n";
std::cout << "To better understand a queue, picture a line at the grocery
store.\nThe customer who's waited the longest gets served first,\nand the line
keeps moving until all customers are finished.\n\n";
std::cout << "This program will use three different data structures (of your
choosing) as implementation of a queue: \n\n";
std::cout << "\t1) Circular Array - An Array (list) where the list of
elements\n\tis treated as a circle,and elements dont need to be rearranged\n\teach
time, we can just rewrite the old spaces.\n\n";
std::cout << "\t2) Linked List - A list of seperate numbers connected to
eachother.\n\tThink of numbers taped to a piece of yarn! Each number
is\n\tseperate, but they are all on the same line.\n\n";
std::cout << "\t3) Circular Linked List - Same type of structure, except\n\tthe
end of that yarn connects back to the first number!\n\n";
std::cout << "Additionally, this program offers a 'Developer Mode' to help
show\nhow each of the data structures work behind the scenes.\nThis can be turned
on from the settings menu.\n\n";

std::cout << "Press any key to start: ";
std::cin >> UserChoice;

system("CLS"); //Clear screen

do { //Main Menu

	//Instructions.
	std::cout << "Main Menu (Please type a letter based on which queue
	structure you'd like):\n\n\t[1] Circular Array\n\t[2] Linked List\n\t[3]
	Circular Linked List\n\t[4] Settings\n\t[5] Quit\n\n";
	std::cout << "Enter here: ";
	std::cin >> UserChoice;


	switch (UserChoice) { //Main menu switch

		default: {

			//If any invalid characters are entered.
			system("CLS"); //Clear Screen
			std::cout << "Invalid option! Please Try Again!\n\n";

			break;

		}

		//All three main cases follow similar structures (except custom user
		length).
		//All notes here (except custom user length) apply to others.

		case '1': { //Case 1 - Circular Array
```

```
system("CLS");

int LengthInput; //Length for user generated length

do { //Get queue length.

	std::cout << "Please enter the length of the desired
	circular array: ";
	std::cin >> LengthInput;
	std::cout << "\n";

	if (LengthInput < 1) { //If invalid length, we must
	catch it beforehand.

		std::cout << "ERROR! Invalid input! Try
		again!\n\n";

	}

} while (LengthInput < 1);

CircularArray QueueObject(LengthInput, DeveloperMode);

do { //Each Turn Loop

	//Visualize again each time through. Screen will have
	been cleared prior each time.
	QueueObject.Visualize();

	std::cout << "Options:\n\n[E/e] Enque\n[D/d]
	Deque\n[Q/q] Quit to Main menu\n\n";

	std::cout << "Choice: ";
	std::cin >> UserChoice;

	if (UserChoice == 'd' || UserChoice == 'D') {

		//Make sure queue is not empty before dequeing

		if (!QueueObject.IsEmpty()) {

			QueueObject.Dequeue();
			system("CLS"); //Clears screen

		} else {

			//If queue is empty, we can not dequeue.
			system("CLS");
			std::cout << "ERROR: Queue is empty! Can
			not deque empty queue!\n\n";

		}

	} else if (UserChoice == 'e' || UserChoice == 'E') {

		//Make sure queue is not full before enqueing
		if (!QueueObject.IsFull()) {
```

```
                        int ValueInput;

                        //Get value to enqueue, and then enqueue
                        to the object.
                        std::cout << "Value to Queue (Integer):
                        ";
                        std::cin >> ValueInput;
                        QueueObject.Enqueue(ValueInput);

                        system("CLS");

                    } else {

                        //If queue is full, we can not enqueue.
                        system("CLS");
                        std::cout << "ERROR: Queue is full! Can
                        not enque a full queue!\n\n";

                    }

                } else if (UserChoice != 'q' && UserChoice != 'Q') {

                    //If input is invalid.
                    system("CLS");
                    std::cout << "ERROR: Input not an option! Try
                    again!\n\n";

                }

                if (UserChoice == 'q' || UserChoice == 'Q') {

                    //Clear Screen before leaving.
                    system("CLS");

                }

            } while (UserChoice != 'q' && UserChoice != 'Q');

            break;

        } //Circular Array - Case '1' - Needed because no redefinition of
        class object error

        case '2': { //Case 2 - Linked List

            system("CLS");

            LinkedQueue QueueObject(DeveloperMode);

            do {

                QueueObject.Visualize();

                std::cout << "Options:\n\n[E/e] Enque\n[D/d]
                Deque\n[Q/q] Quit to Main menu\n\n";

                std::cout << "Choice: ";
```

```
		std::cin >> UserChoice;

		if (UserChoice == 'd' || UserChoice == 'D') {

			if (!QueueObject.IsEmpty()) {

				QueueObject.Dequeue();
				system("CLS");

			} else {

				system("CLS");
				std::cout << "ERROR: Queue is empty! Can
				not deque empty queue!\n\n";

			}

		} else if (UserChoice == 'e' || UserChoice == 'E') {

			if (!QueueObject.IsFull()) {

				int ValueInput;

				std::cout << "Value to Queue (Integer):
				";
				std::cin >> ValueInput;
				QueueObject.Enqueue(ValueInput);

				system("CLS");

			} else {

				system("CLS");
				std::cout << "ERROR: Queue is full! Can
				not enque a full queue!\n\n";

			}

		} else if (UserChoice != 'q' && UserChoice != 'Q') {

			system("CLS");
			std::cout << "ERROR: Input not an option! Try
			again!\n\n";

		}

		if (UserChoice == 'q' || UserChoice == 'Q') {

			//Clear Screen before leaving
			system("CLS");

		}

	} while (UserChoice != 'q' && UserChoice != 'Q');


	break;
```

```
} //Linked List - Case '2'

case '3':  { //Case 3 - Circular Linked List

	system("CLS");

	CircularLinkedQueue QueueObject(DeveloperMode);

	do {

		QueueObject.Visualize();

		std::cout << "Options:\n\n[E/e] Enque\n[D/d]
		Deque\n[Q/q] Quit to Main menu\n\n";

		std::cout << "Choice: ";
		std::cin >> UserChoice;

		if (UserChoice == 'd' || UserChoice == 'D') {

			if (!QueueObject.IsEmpty()) {

				QueueObject.Dequeue();
				system("CLS");

			} else {

				system("CLS");
				std::cout << "ERROR: Queue is empty! Can
				not deque empty queue!\n\n";

			}

		} else if (UserChoice == 'e' || UserChoice == 'E') {

			if (!QueueObject.IsFull()) {

				int ValueInput;

				std::cout << "Value to Queue (Integer):
				";
				std::cin >> ValueInput;
				QueueObject.Enqueue(ValueInput);

				system("CLS");

			} else {

				system("CLS");
				std::cout << "ERROR: Queue is full! Can
				not enque a full queue!\n\n";

			}

		} else if (UserChoice != 'q' && UserChoice != 'Q') {

			system("CLS");
			std::cout << "ERROR: Input not an option! Try
```

```
                        again!\n\n";

                }

                if (UserChoice == 'q' || UserChoice == 'Q') {

                        //Clear Screen before leaving
                        system("CLS");

                }

        } while (UserChoice != 'q' && UserChoice != 'Q');


        break;

} //Linked List - Case '3'

case '4': { //Case 4 - Settings Menu

        system("CLS");

        do {

                std::cout << "Settings:\n\n";

                if (DeveloperMode) {

                        std::cout << "Developer Mode is on. Would you
                        like to turn it off?\n";
                        std::cout << "Enter (Y or y) to turn off, or (Q
                        or q) to go back to the main menu: ";

                } else {

                        std::cout << "Developer Mode is off. Would you
                        like to turn it on?\n";
                        std::cout << "Enter (Y or y) to turn on, or (Q
                        or q) to go back to the main menu: ";

                }

                std::cin >> UserChoice;

                if (UserChoice == 'y' || UserChoice == 'Y') {

                        system("CLS");
                        DeveloperMode = !DeveloperMode;

                } else if (UserChoice != 'q' && UserChoice != 'Q') {

                        system("CLS");
                        std::cout << "ERROR: Invalid input! Try
                        again!\n\n";

                }

                if (UserChoice == 'q' || UserChoice == 'Q') {
```

```
                                        //Clear Screen before leaving
                                        system("CLS");

                                }

                        } while (UserChoice != 'q' && UserChoice != 'Q');

                        break;

                } //Settings - Case '4'

                case '5': { //Case 5 - Exit

                        //Needed so error message doesn't output, but nothing needs to
                        be done.
                        break;

                } //Exit - Case '5'

        }

    } while (UserChoice != '5');

    return 0;

}
```

Circularrray.h

```
#pragma once

//CircularArray - Class Interface
class CircularArray {
    public:

    //Constructors/Destructor
    CircularArray(); //Default Constructor
    CircularArray(int InputLength, bool DeveloperSetting = true);
    ~CircularArray();

    //Functions that will directly alter the list
    //Add an element (Enqueue) or remove an element (Dequeue).
    void Enqueue(int Value);
    void Dequeue();

    //Checks if the list is Full/Empty respectively.
    bool IsFull() const;
    bool IsEmpty() const;

    //Visualization for the user.
    void Visualize() const;

    private:

    //Data Members
```

```
        //Queue is the dynamic array to hold the queue.
        //The index variables used to enforce ciruclar structure,
        //Keeping track of where the queue "starts" and ends based on user action.
        int * Queue;
        int FirstIndex;
        int LastIndex;

        //QueuedElements maintained to determine if list is empty (0 elements).
        //MaxLength used to help determine if full. MaxLength determined by the user.
        int QueuedElements;
        int MaxLength;

        //Determines whether to run DeveloperVisualize() or not
        bool DeveloperMode;

        //Helper Functions

        //Visualizes the array itself (not in queue visual structure) as part of Developer
        Mode.
        void DeveloperVisualize() const;

};
```

CircularArray.cpp

```
#include "CircularArray.h"
#include <iostream>

//Precondition: Queue is not initialized, CircularArray object created from "main"
without parameters. Default constructor.
//Postcondition: Queue is initialized with generic data values.
CircularArray::CircularArray() {

        //Initialize data members
        MaxLength = 9;
        QueuedElements = 0;

        //Dynamic array allocated based on user's requested length.
        Queue = new int[MaxLength];

        FirstIndex = 0;
        LastIndex = 0;

        for (int i = 0; i < MaxLength; i++) {

                Queue[i] = 0; //Initialize Queue values to 0 (needed for readability in
                              developer mode).

        }

        DeveloperMode = true;

}

//Precondtion: Queue is not initialized, CircularArray object created from "main" with
length of user request.
```

```
//Postcondition: Queue is initialized with data values based on user request.
CircularArray::CircularArray(int InputLength, bool DeveloperSetting) {

	//Initialize data members
	MaxLength = InputLength;
	QueuedElements = 0;

	//Dynamic array allocated based on user's requested length.
	Queue = new int[MaxLength];

	FirstIndex = 0;
	LastIndex = 0;

	for (int i = 0; i < MaxLength; i++) {

		Queue[i] = 0; //Initialize Queue values to 0 (needed for readability in
			      developer mode).

	}

	DeveloperMode = DeveloperSetting;

}

//Precondition: Queue is initialized, goes out of scope.
//Postcondition: Dynamic array is deallocated to avoid memory leaks.
CircularArray::~CircularArray() {

	delete [] Queue;

}

//Precondition: Queue is initialized.
//Postcondition: Queue is displayed in a readable queue form to the user. Queue is
unchanged.
void CircularArray::Visualize() const {

	std::cout << "This is the current queue: \n\n";

	std::cout << "Queue:\t| ";

	for (int i = 0; i < MaxLength; i++) {


		//Since we initialize to 0, we need to check if the 0 was entered by the
		user.
		//(meaning it is a memebr of the queue) or it is representing an empty
		space.
		if (Queue[i] == 0) {

			if (FirstIndex == LastIndex) {

				//If the indexes are on top of each other, there is either one
				member or no members.

				if (IsEmpty() || i != LastIndex) {

					//If empty, this is an empty element.
```

```
                        //Or if the current index is not the
                        FirstIndex/LastIndex, we know it must be empty
                        //because if the two indexes are the same only that
                        index could possible hold an element.
                        std::cout << '*';

                } else if (i == LastIndex) {

                        //If i is the LastIndex/FirstIndex space, only the
                        first element is full.
                        std::cout << Queue[i];

                }

        } else if (FirstIndex < LastIndex) {

                //If the array does not wrap around

                //If the 0 is not between the two indexes it is not an
                intended member of the queue
                if (i < FirstIndex || i > LastIndex) {

                        std::cout << '*';

                } else {

                        std::cout << Queue[i];

                }

        } else if (LastIndex < FirstIndex) {

                //If the indexes wrap around the queue

                //If i is out of the wrapped range
                if (i > LastIndex && i < FirstIndex) {

                        std::cout << '*';

                } else {

                        std::cout << Queue[i];

                }

        }

    } else { //If any other number

            std::cout << Queue[i]; //Print as normal

    }

    std::cout << " | ";

}

std::cout << "\n\n";
```

```
	if (DeveloperMode) {

		//Wont display if empty or DeveloperMode is off.
		DeveloperVisualize();

	}

	std::cout << "The maximum length of the queue is " << MaxLength << ".\n\n";


}

//Precondition: Queue is initialized.
//Postcondition: Queue displayed as it is. Queue unchanged.
void CircularArray::DeveloperVisualize() const {

	for (int i = 0; i < MaxLength; i++) {

		std::cout << Queue[i] << "[" << i << "] | ";

	}

	std::cout << "\n\nFirstIndex: " << FirstIndex << "\tLastIndex: " << LastIndex <<
"\n\n";

}


//Precondition: Queue is initialized and is not full.
//Postcondition: Value is enqueued in the "back" of the queue.
void CircularArray::Enqueue(int Value) {

	if (!IsEmpty()) {

		//If the queue is empty, the index's will be on top of each other.
		//If they are, we can just add the new element in that place.
		//If not, we add it to the next spot (we need to alter LastIndex).

		if (LastIndex != MaxLength - 1) {

			//If we dont need the index counter to circle back.

			LastIndex++;

		} else {

			LastIndex = 0;

		}

	}

	//Add the new value tot he proper index plcace, update counter variable.
	Queue[LastIndex] = Value;
	QueuedElements++;
```

```
}

//Precondition: Queue is initialized and is not Empty.
//Postcondition: Least recent element is removed from the queue.
void CircularArray::Dequeue() {

	Queue[FirstIndex] = 0; //Set removed element back to 0.

	if (LastIndex != FirstIndex) {

		//If there was more than one element left before dequeing
		//We need to update FIrstIndex.
		//(If there was only one element, the indexes would be
		//on top of eachother).

		if (FirstIndex != MaxLength - 1) {

			//If we dont need the index counter to circle back.

			FirstIndex++;

		} else {

			//If we need FirstIndex to circle back to the front.
			FirstIndex = 0;

		}

	}

	//Update counter.
	QueuedElements--;

}

//Precondition: Queue is initialized.
//Postcondition: Queue is determined to be empty/not empty. Queue unchanged.
bool CircularArray::IsEmpty() const {

	return (QueuedElements == 0);

}

//Precondition: Queue is initialized.
//Postcondition: Queue is determined to be full/not full. Queue unchanged.
bool CircularArray::IsFull() const {

	if (LastIndex == MaxLength - 1 && FirstIndex == 0) {

		//If the two index trackers are at opposite ends but circularly next to
		eachother.
		return true;

	} else {

		//Else, if full, Last will be one behind first.
		return (LastIndex == FirstIndex - 1);
```

```
	}

}
```

LinkedQueue.h

```
#pragma once

//Header guards so QueueNode isn't defined twice between LinkedQueue.h and
CircularLinkedQueue.h
#ifndef STRUCTNODE
#define STRUCTNODE

//The struct interface each node in the Linked List uses
struct QueueNode {

	int NodeValue;
	QueueNode * Next;

};

#endif

//LinkedQueue - Class Interface
class LinkedQueue {

	public:

	//Constructor/Destructor
	LinkedQueue(bool DeveloperSetting = true); //Default parameter, so this is default
	constructor.
	~LinkedQueue();

	//Functions

	//Functions that will directly alter the list
	//Add an element (Enqueue) or remove an element (Dequeue).
	void Enqueue(int Value);
	void Dequeue();

	//Checks if the list is Full/Empty respectively.
	bool IsFull() const;
	bool IsEmpty() const;

	//Visualization for the user.
	void Visualize() const;

	private:


	//Points to Head of linked list
	QueueNode * Head;
	//Determines whether to run DeveloperVisualize() or not
	bool DeveloperMode;

	//Helper Functions

	//Special diagram for DeveloperMode.
```

```
        void DeveloperVisualize() const;
        //Makes the list empty upon deconstruction.
        void MakeEmpty();
```

LinkedQueue.cpp

```
#include "LinkedQueue.h"
#include <iostream>

//Precondition: List is not initialized, object created in "main".
//Postcondition: List object initialized with correct settings.
LinkedQueue::LinkedQueue(bool DeveloperSetting) {

        //Head starts as NULL (no elements).
        Head = 0;

        //Determines whether to display DeveloperMode visualization or not.
        DeveloperMode = DeveloperSetting;

}

//Precondition: List is initialized, goes out of scope.
//Postcondition: All list memory is deallocated, list is freed up.
LinkedQueue::~LinkedQueue() {

        MakeEmpty();

}

//Precondition: List is initialized and not full.
//Post condition: New node will become the head of the list.
void LinkedQueue::Enqueue(int Value) {

        //Put at front - easier for visualize function

        QueueNode * NewNode = new QueueNode;

        NewNode->NodeValue = Value;
        NewNode->Next = 0;

        if (IsEmpty()) {

                //If Head is null, the NewNode is the head.
                Head = NewNode;

        } else {

                //If there is already an element in the linked list.

                QueueNode * TraversingNode = Head; //Used for looping through while loop.

                while (TraversingNode->Next != 0) {

                        //Cycle until we reach last node.
                        TraversingNode = TraversingNode->Next;

                }
```

```
                TraversingNode->Next = NewNode; //Connect NewNode so that NewNode is the
                                                end of the queue.

        }

}

//Precondition: List is initilaized and is not empty.
//Postcondition: Oldest element in linked list is dequeued, deallocated, and removed.
void LinkedQueue::Dequeue() {

        QueueNode * ToDelete = Head; //Will be deleting head (oldest element).

        Head = Head->Next; //Reallign the head to be the next node.

        delete ToDelete; //Delete old head.

}

//Precondition: Linked List is initialized.
//Postcondition: List unchanged, checked whether maximum capacity has been reached.
bool LinkedQueue::IsFull() const {

        QueueNode * TempNode;

        try {

                //Try to create new node. If no error, we still have storage space.
                TempNode = new QueueNode;
                delete TempNode;

                return false;

        }

        catch (std::bad_alloc exception) {

                //If error, catch error and record that the list has reached maximum
                capacity.
                return true;

        }

}

//Precondition: List is initialized.
//Postcondition: List is determined to be empty or not empty. List unchanged.
bool LinkedQueue::IsEmpty() const {

        //If the "Head" pointer isn't pointing to anything (pointing to NULL), the list is
        empty.
        return (Head == 0);

}

//Precondition: List is initialized.
//Postcondition: List is displayed in a queue form to the user. List is unchanged.
```

```
void LinkedQueue::Visualize() const {

	if (IsEmpty()) {

		//If empty, tell the user.
		std::cout << "The queue is currently empty.\n\n";

	} else {

		//If not empty, we visualize.
		std::cout << "This is the current queue: \n\n";

		std::cout << "\t| ";

		QueueNode * Location;

		Location = Head;

		while (Location != 0) {

			std::cout << Location->NodeValue << " | ";

			Location = Location->Next;

		}

		std::cout << "\t\n\n";

		if (DeveloperMode) {

			//Wont display if empty or DeveloperMode is off.
			DeveloperVisualize();

		}

	}

}

//Precondition: List is initialized.
//Postcondition: List displayed in a more clearly "Linked List" structure. List
unchanged.
void LinkedQueue::DeveloperVisualize() const {

	std::cout << "\t";

	QueueNode * Location;

	Location = Head;

	while (Location != 0) {

		//Cycle through each node, starting with head, to display values.
		std::cout << "| " << Location->NodeValue << " |";

		Location = Location->Next;

		if (Location != 0) {
```

```
                        std::cout << " --> ";

                } else {

                        //End of list has "x" to mark pointer as NULL.
                        std::cout << " x |";

                }

        }

        std::cout << "\n\n";

        std::cout << "\n\nHead Node's Value = " << Head->NodeValue << "\n\n";

}

//Precondition: Linked List is initialized and not empty.
//Postcondition: All memory deallocated, list is empty.
void LinkedQueue::MakeEmpty() {

        QueueNode * TempPointer;

        while (Head != 0) {

                //Delete Head each time until we're done
                //And move head one step further until it hits NULL

                TempPointer = Head;

                Head = Head->Next;

                delete TempPointer;

        }
}
```

CircularLinkedQueue.h

```
#pragma once

//Header guards so QueueNode isn't defined twice between LinkedQueue.h and
CircularLinkedQueue.h
#ifndef STRUCTNODE
#define STRUCTNODE

//The struct interface each node in the Linked List uses
struct QueueNode {

        int NodeValue;
        QueueNode * Next;

};

#endif
```

```
//CircularLinkedQueue - Class Interface
class CircularLinkedQueue {
	public:

	//Constructor/Destructor
	CircularLinkedQueue(bool DeveloperSetting = true); //Default parameter, so this is
	default constructor.
	~CircularLinkedQueue();

	//Functions

	//Functions that will directly alter the list
	//Add an element (Enqueue) or remove an element (Dequeue).
	void Enqueue(int Value);
	void Dequeue();

	//Checks if the list is Full/Empty respectively.
	bool IsFull() const;
	bool IsEmpty() const;

	//Visualization for the user.
	void Visualize() const;

	private:

	//Data Members

	//Points to Head of linked list
	QueueNode * Head;
	//Determines whether to run DeveloperVisualize() or not
	bool DeveloperMode;

	//Helper Functions
	void DeveloperVisualize() const; //Special diagram for DeveloperMode
	//Makes the list empty upon deconstruction.
	void MakeEmpty();

};
```

CirculuarLinkedQueue.cpp

```
#include "CircularLinkedQueue.h"
#include <iostream>

//Precondition: List is not initialized, object created in "main".
//Postcondition: List object initialized with correct settings.
CircularLinkedQueue::CircularLinkedQueue(bool DeveloperSetting) {

	//Head starts as NULL (no elements).
	Head = 0;

	//Determines whether to display DeveloperMode visualization or not.
	DeveloperMode = DeveloperSetting;

}

//Precondition: List is initialized, goes out of scope.
```

```
//Postcondition: All list memory is deallocated, list is freed up.
CircularLinkedQueue::~CircularLinkedQueue() {

	MakeEmpty();

}

//Precondition: List is initialized and not full.
//Post condition: New node will become the head of the list.
void CircularLinkedQueue::Enqueue(int Value) {

	//"Main" ensures list is not full before calling.

	QueueNode * NewNode;

	NewNode = new QueueNode;

	//Load the new node with the user's value.
	NewNode->NodeValue = Value;

	if (IsEmpty()) {

		//If empty, create node pointing to self.
		NewNode->Next = NewNode;

		Head = NewNode;

	} else {

		//Since NewNode will be at the end, NewNode circles back to head.
		NewNode->Next = Head;

		QueueNode * TempLocater;

		TempLocater = Head;

		//Loop won't run if one element, but that means we've already found our
		last node.
		while (TempLocater->Next != Head) {

			//Go until we find the node that circles back to head, or the "last"
			node in the linked list
			TempLocater = TempLocater->Next;

		}

		//Circle back to new head.
		TempLocater->Next = NewNode;

	}

}

//Precondition: List is initilaized and is not empty.
//Postcondition: Oldest element in linked list is dequeued, deallocated, and removed.
void CircularLinkedQueue::Dequeue() {

	QueueNode * ToDelete = Head; //We will be deleting the head.
```

```
	//We need to find rear to make it circle to the node after head.
	QueueNode * RearLocater = Head;

	while (RearLocater->Next != Head) {

		//Cycle until we find the node before head (the linked list rear).
		RearLocater = RearLocater->Next;

	}

	if (Head->Next == Head) {

		//If circles back to itself (only one element)

		delete ToDelete; //Delete the head.

		Head = 0; //Set Head to null.

	} else {

		Head = Head->Next; //Update new Head

		RearLocater->Next = Head; //Circle back to new Head.

		delete ToDelete; //Delete old Head.

	}

}

//Precondition: List is initialized.
//Postcondition: List is determined to be full or not full. List unchanged.
bool CircularLinkedQueue::IsFull() const {

	QueueNode * TempNode;

	try {

		//Try to create new node. If no error, we still have storage space.
		TempNode = new QueueNode;
		delete TempNode;

		return false;

	}

	catch (std::bad_alloc exception) {

		//If error, catch error and record that the list has reached maximum
		capacity.
		return true;

	}

}

//Precondition: List is initialized.
```

```
//Postcondition: List is determined to be empty or not empty. List unchanged.
bool CircularLinkedQueue::IsEmpty() const {

	//If head is NULL, we know there are no elements.
	return (Head == 0);

}

//Precondition: List is initialized.
//Postcondition: List is displayed in a queue form to the user. List is unchanged.
void CircularLinkedQueue::Visualize() const {

	//If empty, tell the user.
	if (IsEmpty()) {

		std::cout << "The queue is currently empty.\n\n";

	} else {

		//If not empty, we visualize.
		std::cout << "This is the current queue: \n\n";

		std::cout << "\t| ";

		QueueNode * Location;

		Location = Head;

		do {

			//Run once and go until We encoutner the Head again.
			std::cout << Location->NodeValue << " | ";

			Location = Location->Next;

		} while (Location != Head);

		std::cout << "\t\n\n";

		if (DeveloperMode) {

			//Wont display if empty or DeveloperMode is off.
			DeveloperVisualize();

		}

	}

}

//Precondition: List is initialized.
//Postcondition: List displayed in a more clearly "Linked List" structure. List
unchanged.
void CircularLinkedQueue::DeveloperVisualize() const{

	std::cout << "\t";

	QueueNode * Location;
```

```
	Location = Head;

	std::cout << " --> ";

	if (Location->Next == Head) {

		//If there is only one element (head points to itself), the loop will not
		//run, so we make sure to still print the singular node value.
		std::cout << "| " << Location->NodeValue << " |";

	}

	while (Location->Next != Head) {

		//Print value at node
		std::cout << "| " << Location->NodeValue << " |";

		//Cycle through to next node
		Location = Location->Next;

		if (Location->Next != Head) {

			//If we
			std::cout << " --> ";

		} else {

			//Loop will exit after this, so make sure to print last result.
			//(This is the node that circles back to Head).
			std::cout << " --> | " << Location->NodeValue << " |";

		}

	}


	std::cout << " --> \n\n";

	//Extra data to prove linked list is circular.
	std::cout << "\n\nHead Node's Value = " << Head->NodeValue << "\n";
	std::cout << "Head Node Location = " << Head << "\nNode Before Head Location = "
	<< Location << "\nNode Before Head's ->Next Location = " << Location->Next <<
	"\n\n";


}

//Precondition: Linked List is initialized and not empty.
//Postcondition: All memory deallocated, list is empty.
void CircularLinkedQueue::MakeEmpty() {

	if (IsEmpty()) {

		//Check to make sure it isn't already empty.
		return;

	} else {
```

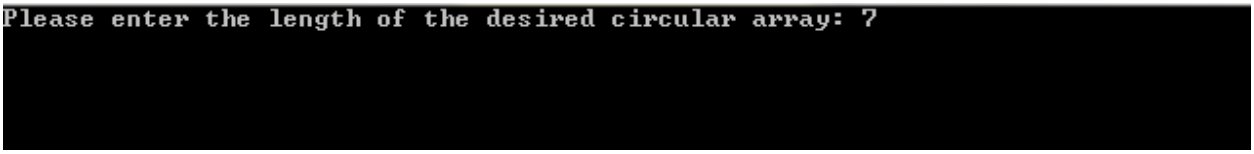
```
		//Temp pointer will cycle through, move Head forward, and delete what's at
		the spot.
		//Will also be used to make the last element point to NULL instead of head,
		//so we know where to stop when it's time to delete (instead of readjusting
		//the end pointer, which is suppsed to point to head, of the linked list
		each time).
		QueueNode * TempPointer;

		TempPointer = Head;

		while (TempPointer->Next != Head) {

			TempPointer = TempPointer->Next;

		}

		TempPointer->Next = 0; //Seal up the last node so we know when to stop.

		while (Head != 0) {

			//Delete Head each time until we're done
			//And move head one step further until it hits NULL
			TempPointer = Head;

			Head = Head->Next;

			delete TempPointer;

		}

	}

}
```

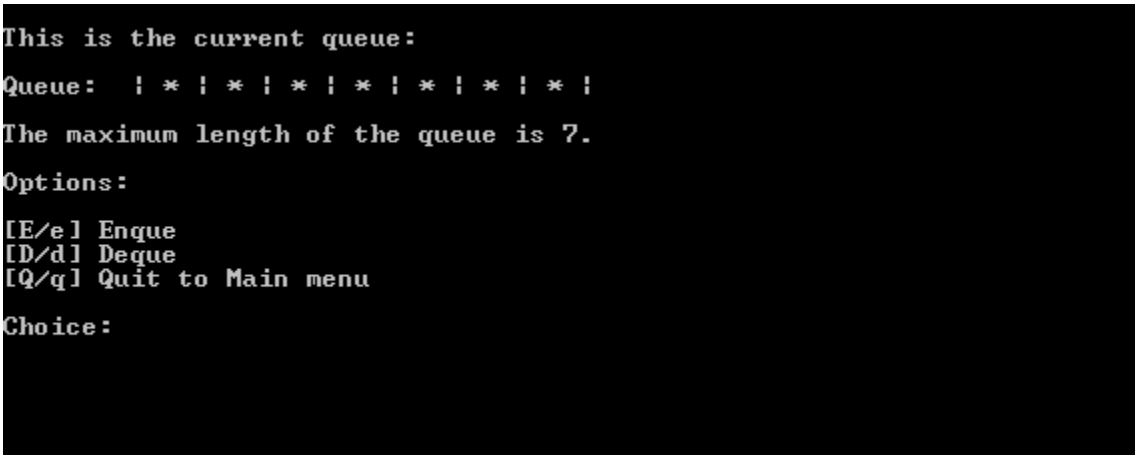
Tests

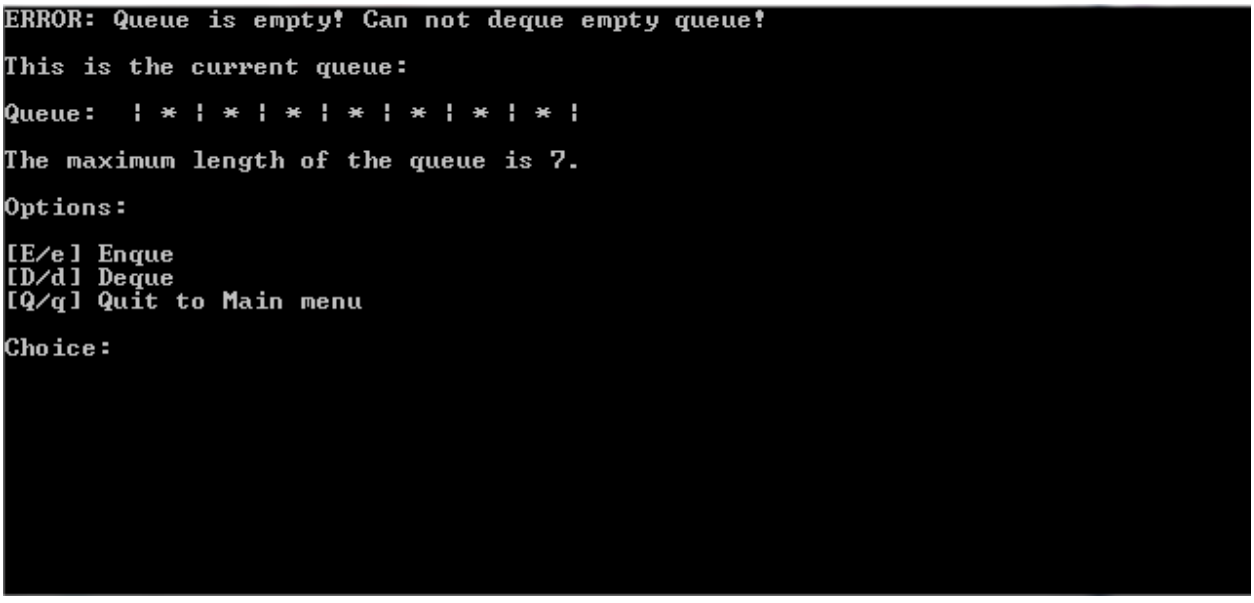
```
Welcome to the Queue Creator!

A queue is a type of datat implementation in which the first elements put in are
 the first to be removed.

To better understand a queue, picture a line at the grocery store.
The customer who's waited the longest gets served first,
and the line keeps moving until all customers are finished.

This program will use three different data structures (of your choosing) as impl
ementation of a queue:

        1) Circular Array - An Array (list) where the list of elements
        is treated as a circle,and elements dont need to be rearranged
        each time, we can just rewrite the old spaces.

        2) Linked List - A list of seperate numbers connected to eachother.
        Think of numbers taped to a piece of yarn! Each number is
        seperate, but they are all on the same line.

        3) Circular Linked List - Same type of structure, except
        the end of that yarn connects back to the first number!

Additionally, this program offers a 'Developer Mode' to help show
how each of the data structures work behind the scenes.
This can be turned on from the settings menu.

Press any key to start:
```

```
Invalid option! Please Try Again!

Main Menu (Please type a letter based on which queue structure you'd like):

        [1] Circular Array
        [2] Linked List
        [3] Circular Linked List
        [4] Settings
        [5] Quit

Enter here:
```

These pictures show the introduction screen and basic main menu screen that the user sees.

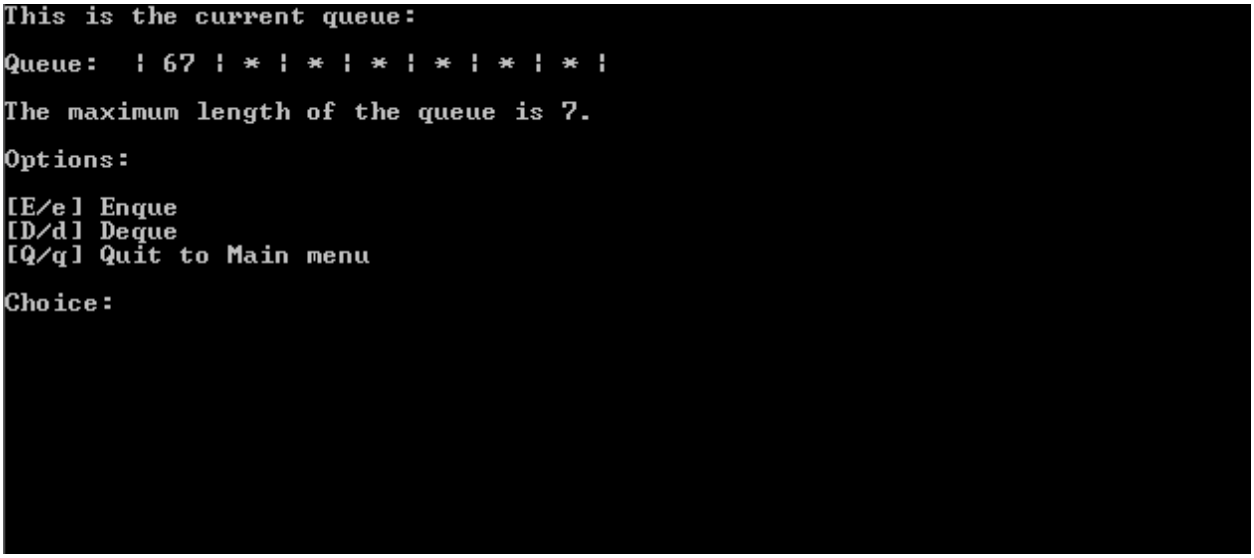


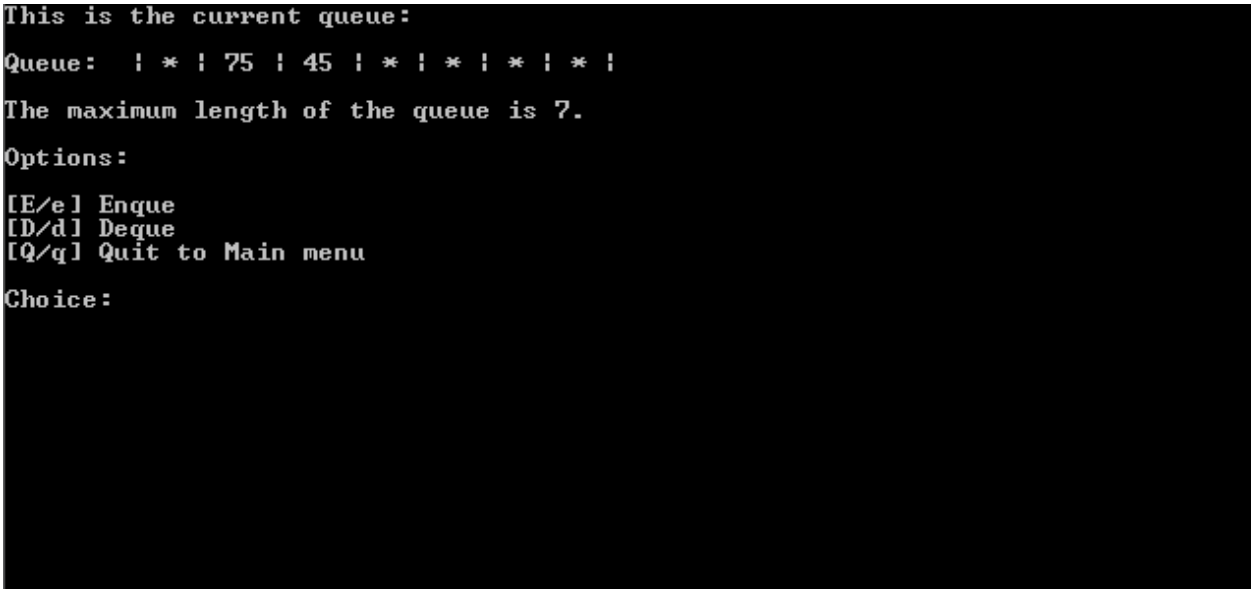


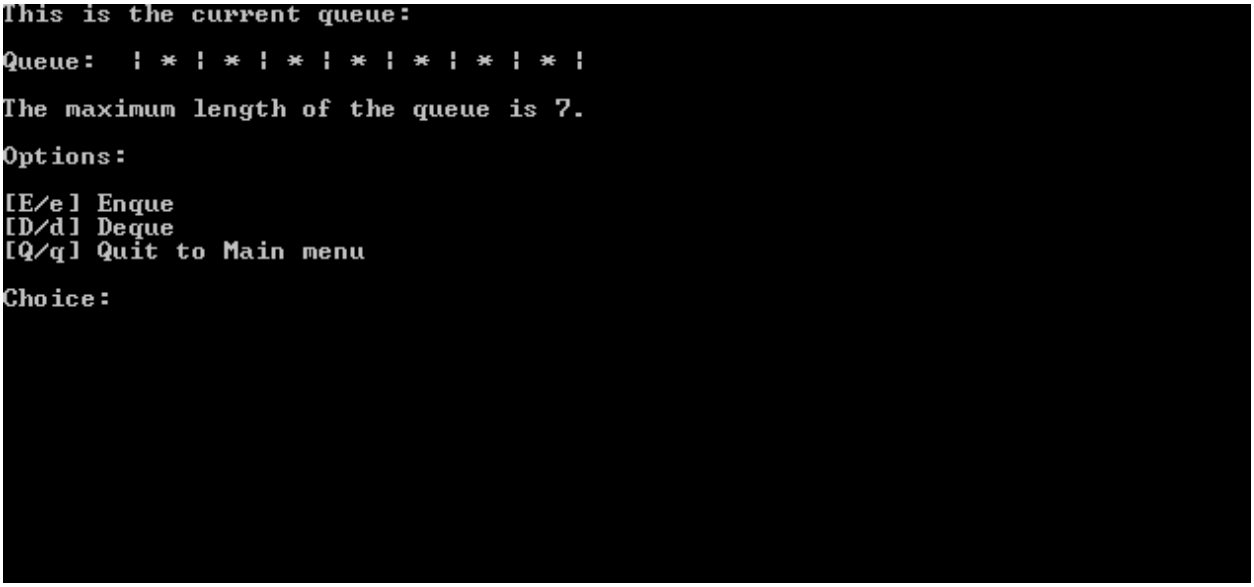


These pictures show what the user will see when they choose the circular array option. They will be prompted to enter a certain length to make the array. If the user tries to dequeue when the array is empty, they will get an error message (and if they try to enqueue when the

queue is full). The bottom picture shows the user choosing to enqueue, and a prompt for the value of the enqueue.



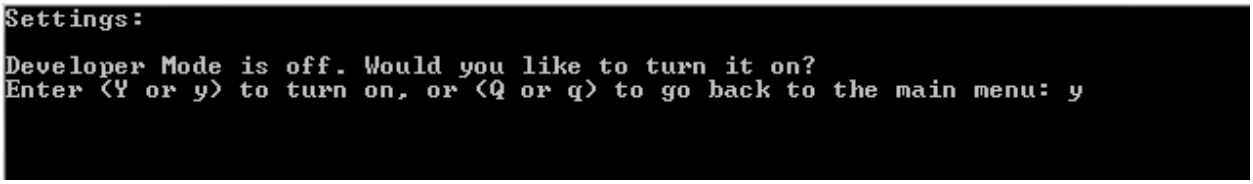


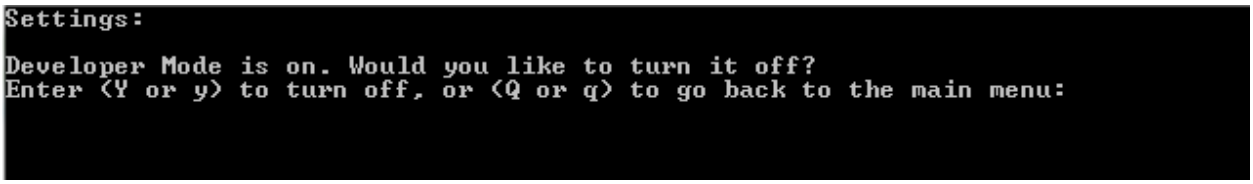


These pictures show a normal flow of options for the circular array. A user enqueues, maybe enqueues some more, and then dequeues one. Finally, the user dequeues until the array is empty once more.

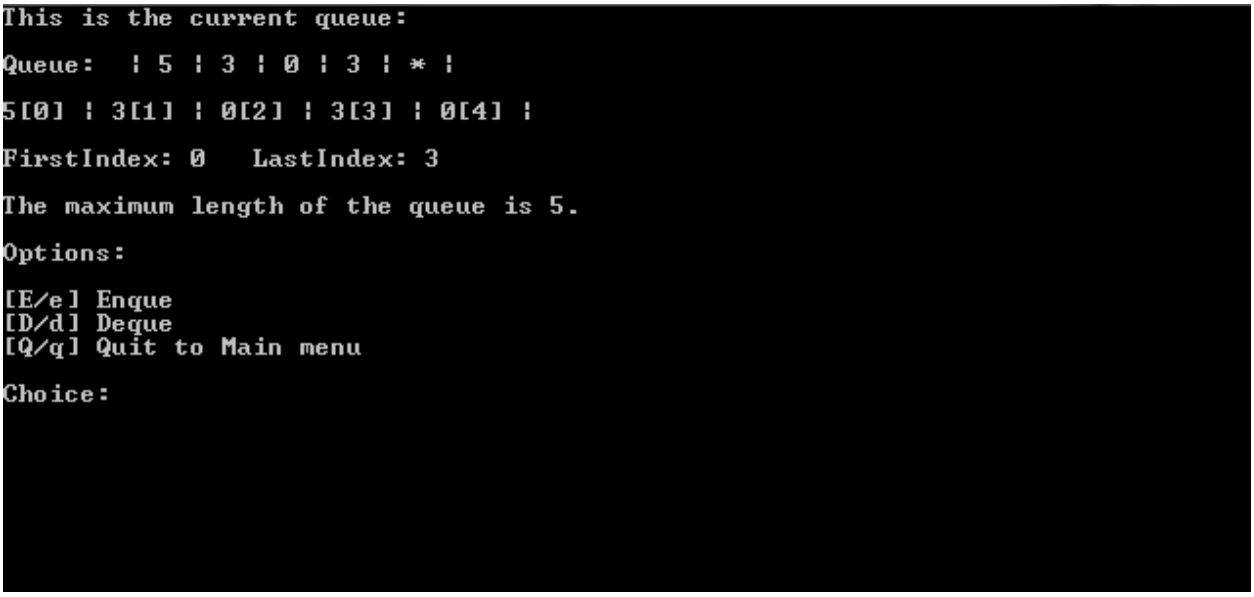
These pictures show enqueuing and dequeuing with a linked list structure.

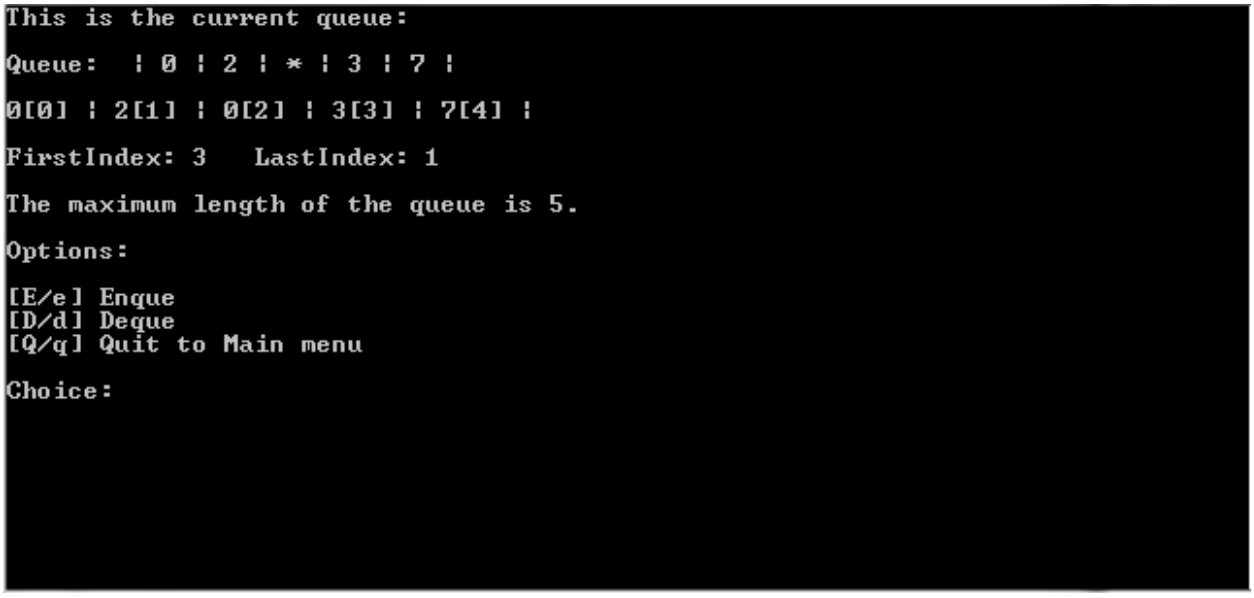
These pictures show enqueuing and dequeuing again with a circular linked structure.



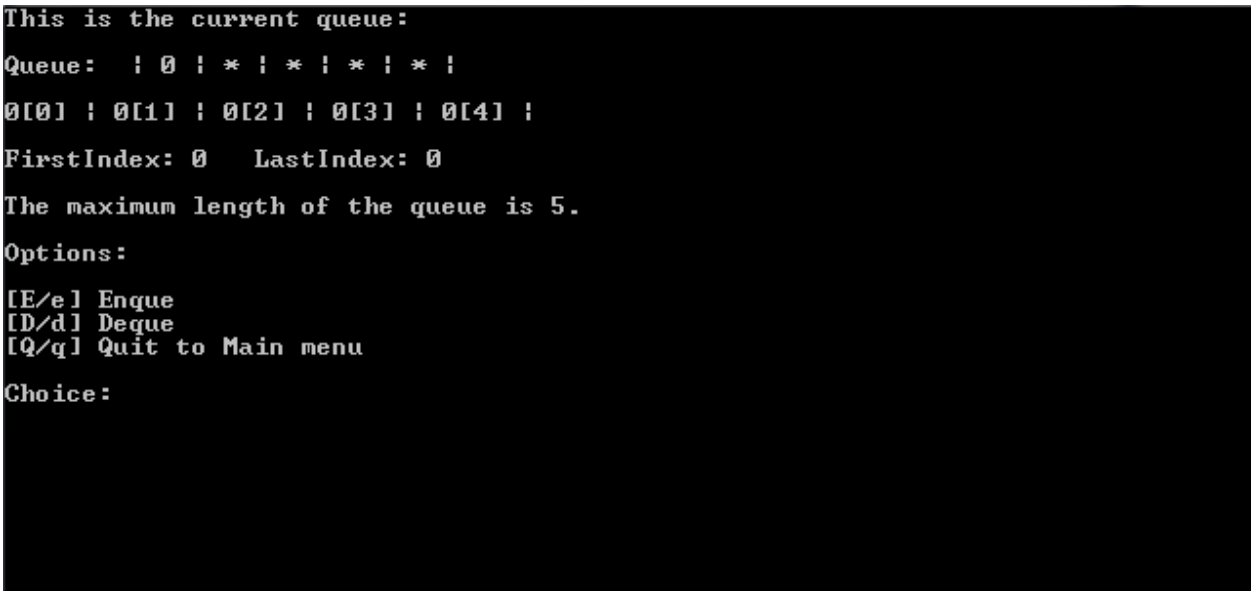


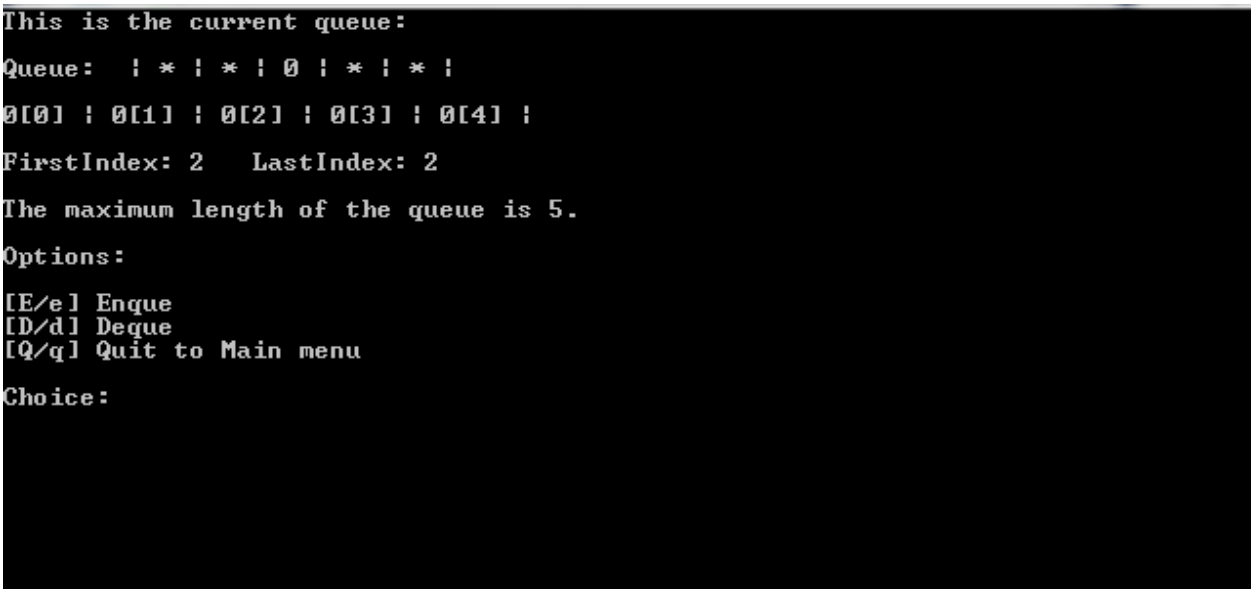
These pictures show the settings menu, and the user toggling Developer Mode on.



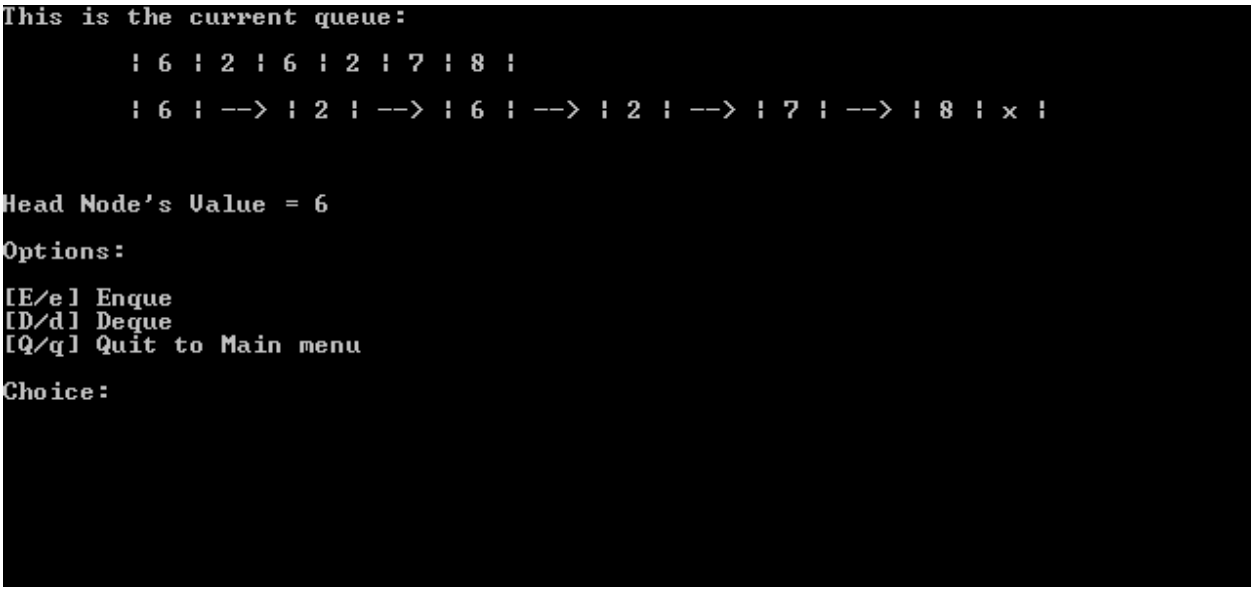


These pictures show the developer mode with the circular array. In the first picture, the array has been enqueued from the start, but in the second picture the queue has been altered in a way so that it wraps around the array. This is the circular queue in action, and proof that it works in a circular implementation.

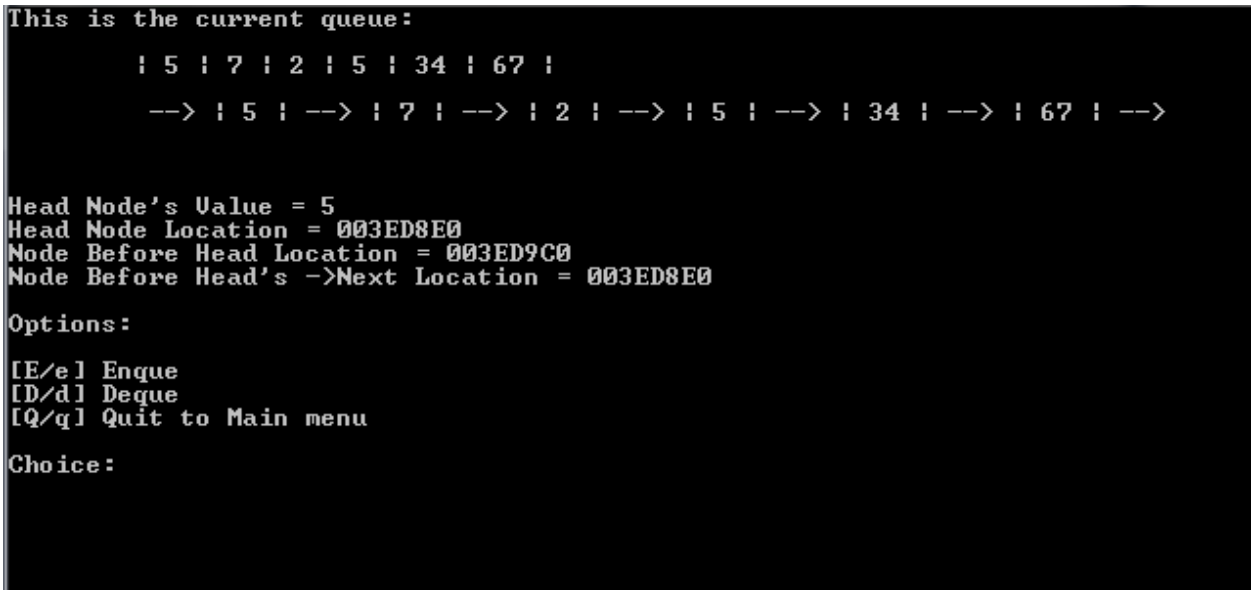


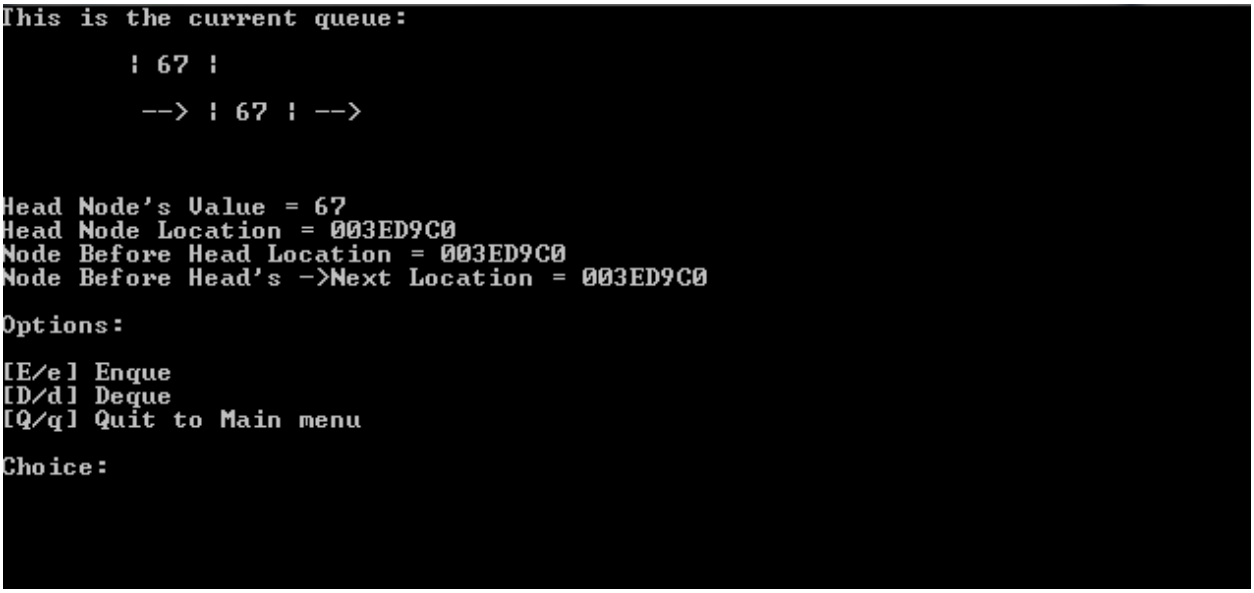


These pictures with a circular array prove that the Visualize function can correctly tell the difference between an empty space and an intended 0 value when the two index variables are on top of each other.



These pictures show the normal linked list developer mode, and shows a user experimenting with enqueuing and dequeuing.





These pictures show the circular linked list Developer Mode. Note that the data shows key facts that prove the list is circularly linked. Note that the rear node's memory location changes as we dequeue, but the Head's does not, proving what we know in theory of a circularly linked list.

Lessons Learned

In this project, my understanding of different data implementations was cemented as I experimented and created them. In addition, I learned how to implement a new structure, the circular array, which opened my eyes to a new way of implementing certain structures. I learned more about time complexity and efficiency between the different implementations, and how multiple implementations can perform the task several different, more efficient ways.

In conclusion, I am extremely happy at the outcome of this product and feel that I have a strong knowledge on linked lists and circular linked lists/arrays. This project was both challenging and rewarding, but I am extremely happy at the outcome.

References

I wanted to make sure the screen didn't become too cluttered as the program progressed, so I needed a method to clear the screen each time a user progresses from one menu or queue update to the next. Luckily, I found a method for doing that on www.mathbits.com.

Full link: https://mathbits.com/MathBits/CompSci/Introduction/clear.htm

In the main menu switch statement of the program, I was encountering an unusual error when I introduced class objects to the mix. The error read that the "initialization of 'element' is skipped by 'case label'. Luckily, I found someone with a similar problem on www.stackoverflow.com. This discussion gave me the solution to cover my switch statements with brackets to avoid this unusual error.

Full link: https://stackoverflow.com/questions/14669457/initialization-of-element-is-skipped-by-case-label