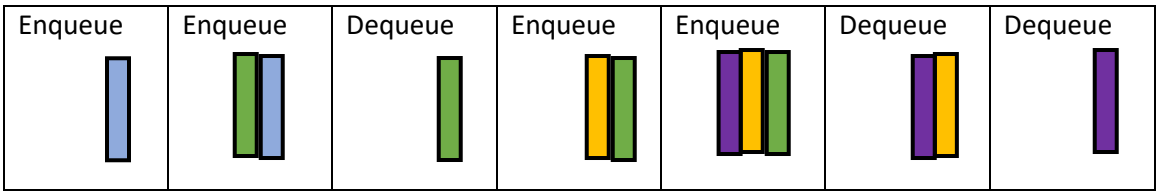
Assignment 1 Report - QUEUE

Code: Minh Pham

Report: Joseph Levesque

1. A queue is a data structure that holds information using functions that operate on a FIFO (first in, first out) model. This means that as data is entered into a queue, it will be removed in the same order that it is put in. The function that inserts elements into the queue is called “enqueue”, and it places those elements at the end. The function that takes elements out of the queue is called “dequeue”, and it takes them from the start of the queue, the elements that were added first. For example, if elements 3, 62, 7, 23, and 4 are added to the queue (enqueue) in that order, they would be removed (dequeue) in this order: 3, 62, 7, 23, and 4. The queue data structure is important for when the order that data was inserted is significant to the order that the data needs to be deleted. An example of a queue in everyday life is the line to checkout at the grocery store. The first customers to get into the line will be the first customers that make it to the checkout, and the order that people get into the line will be the order that people checkout in (first in, first out).
   Diagram (Starting with an empty queue):
   For the purpose of the diagram, the leftmost blocks are the end of the queue and the right is the start. Read the table from left to right.



2. An array is a basic computer programming tool in which different elements of data are stored together in memory. Unlike a linked list where data can be stored in different places in memory, an array and the data in it is stored all in 1 big block of memory in one place. Thus, you do not need to store the data in a node with a pointer to the next element. You already know that the next element will be stored in the next block of memory. You can access the various elements of an array by referencing their index, usually relating it to the first element of the array, then going that distance forwards in memory (multiplied by the size of each element).
   Diagram:

Index: 0 Data: 3 Address: 0x0000	Index: 1 Data: 6 Address: 0x0004	Index: 2 Data: 7 Address: 0x0008	Index: 3 Data: 2 Address: 0x000C	Index: 4 Data: 4 Address: 0x0010

3. Our program mainly consists of a class definition called Queue, which begins by storing index variables for capacity, size, front, and end (to be used in an array). It then defines a pointer variable for an array ‘a’. There are then 7 function definitions: “empty()” returns a Boolean value

for if the queue is empty or not, “size()” returns the current size of the queue, “front()” returns the element at the front of the queue, “back()” returns the element at the back of the queue, “push()” inserts a new element at the back of the array (enqueue), “p0p()” removes the element from the start of the array (dequeue), and “print()” prints out all the current elements in the queue.
To test these functions, we declared a Queue ‘q’ in our main function and printed what each return function returned (front, back, size, empty) after performing the various functions of the void functions (pop, push). As expected, when we used the push function, the program (after making sure the queue wasn’t full) inserted the element into the position of the array indexed by the “end” variable, or at the end of the queue. When we used the pop function, we simply moved the index of the front index variable to the next element in the array. In terms of efficiency, this pop function doesn’t really do anything about the elements in memory that have been popped (dequeue), and this could be fixed by deleting the element after moving the front index variable to the next element.

Example:	**Execution Time (Custom Code):**	**Execution Time (STL library):**
Check if queue is empty	.000154 ms	.000186 ms
Enqueue (push(5))	3e-7 ms	6e -7ms
Enqueue (push(8))	3e-7ms	5e-7 ms
Enqueue (push(4))	1e-7 ms	2e-7 ms
Print Front	.0001195 ms	.0001108 ms
Print Back	.0001116 ms	.0001107 ms
Print Size	.0001114 ms	.0001117 ms
Dequeue (pop())	1e-7 ms	2e-7 ms
Enqueue (push(5)) (2^{nd} test)	1e-7 ms	2e-7 ms
Enqueue (push(8)) (2^{nd} test)	1e-7 ms	2e-7 ms
Enqueue (push(4)) (2^{nd} test)	1e-7 ms	1e-7 ms
Print Size	.000192 ms	.0001051 ms
Enqueue (push(4)) (3^{rd} test)	1e-7 ms	1e-7 ms
Check if queue is empty	1.7e-6 ms	1.37e-6 ms
Print size	1.94e-6 ms	1.52e-6 ms

(last 2 test cases were on a different computer)

As shown by these results, the custom code did pretty well in comparison to the STL library. In many cases, it actually worked faster. It may be because we used smaller test cases and our custom code works better for that, whereas it would’ve been much slower with larger test cases.