Erick Gonzalez Parada #178145

Andre Francois Duhamel Gutiérrez #177315

Alejandro González Díaz #178645

# Activity #5 Circular queues

## Project objective

The objective of this activity is to identify the advantages, disadvantages & security aspects of the circular queues and the differences in its implementation in Java and C (C language was not asked but we want to do it for fun)

[[1]](#_Bibliography)

A circular queue is an extended version of a normal queue. The circular queue differs in the connection between the last element and the first element of the queue, forming a type of circle. This data structure follows the FIFO principle o also known as ring buffer.

## Solution design

### UML diagram for Java’s circular queue implementation.

Texto

Descripción generada automáticamente con confianza media

### Approach

For the java implementation we as a team decided to take the approach of creating an array of type Object in order to declare the circular queue, although we could have used a linked list in order to make it dynamic and altering the memory size as the program runs. We also implemented a circularqEnqueue() and circularqDequeue() functions in order to manage the data wanted to push or pop from the circular queue.

For the C implementation we decided to use headers and 2 files in order to implement the circular queue. We also decided to use a enqueue() and dequeue() functions

## Main steps

We first needed to create 3 attributes of the class Queue, which are used to declare the array, the size of the array and the front and rear positions. The we first need to create an object of the Class Queue which in its constructor sets the size of the array, and the front and rear values to -1. After that we had to develop the circularqEnqueue() and circularqDequeue(). The handling of exceptions was used with a try-catch, which handled when the queue is full or empty.

## Security aspects

The security aspects we took in count for both programs (Java and C):

Buffer Overflow/Underflow: When you enqueue an element into a full queue or dequeue an element from an empty queue, it can lead to buffer overflow and underflow errors respectively. In the provided code, these cases are handled by throwing exceptions when such operations are attempted by using try-catch.

Data Integrity: Since the queue is implemented using an array, the data is stored in contiguous memory locations. This makes it efficient but also susceptible to data corruption if not handled properly. However, in Java, this risk is mitigated by the JVM’s memory management.

Thread Safety: If our queue is being accessed by multiple threads, we need to ensure that the enqueue and dequeue operations are thread-safe. This can be achieved by synchronizing these methods or using a lock.

Memory Management: In the provided code, the size of the queue (array) is fixed at the time of creation. If we’re dealing with a large amount of data or if the required size of the queue can change dynamically, we might need a dynamic data structure or implement a mechanism to resize the array. As said, in the introduction we could have used something like a linked list, but the objective of this task is to understand the functioning of a circular queue in a static data structure.

## Analysis

### Code (Java Circular queue)

class Queue {

//attributes

    private  Object array[];

    private int front, rear;

    private static int n;

    Queue(int *n*) {

//setting the values of the attributes in order to setup correctly the circular queue

        Queue.n = *n*;

        array = new Object[*n*];

        front = rear = -1;

    }

    public void circularqEnqueue(Object *Element*) throws Exception {

        try { //this try statement helps us to throw an exception without stopping the execution of the program

            if ((rear + 1) % n == front) {

                throw new Exception(); //if the rear is one position before the front value it will throw the exception

            }

//moving the rear value depending on the quantity of queued values

            rear = (rear + 1) % n;

//assigning the value to its position

            array[rear] = *Element*;

            if (front == -1) {

                front = 0; //advancing the front to its initial position

            }

        } catch (Exception *e*) {

            System.out.println("The queue was full when inserting the value of " + *Element*); //this handler helps to continue the program even when the queue is full

        }

    }

    public Object circularqDequeue() throws Exception {

        try {//this try statement helps us to throw an exception without stopping the execution of the program

            if (front == -1) {

                throw new Exception();//if the front value hasn’t been changed, no value has been pushed into the queue

            }

//getting the value that is being popped off

            Object element = array[front];

//

            if (front == rear) {

                front = rear = -1;

            } else {

                front = (front + 1) % n; //moving the front value to the next value.

            }

            return element; //returning the dequeued value

        } catch (Exception *e*) {

            return ("The queue is empty"); //handling the exception

        }

    }

    public static void main(String[] *args*) throws Exception {

        Queue q = new Queue(5); //creation of q object

//pushing values into the queue

        q.circularqEnqueue("One");

        q.circularqEnqueue("Two");

        q.circularqEnqueue("Three");

        q.circularqEnqueue("Four");

        q.circularqEnqueue("Five");

//popping the values following the FIFO principle

        System.out.println("The dequeued value is " + q.circularqDequeue());

        System.out.println("The dequeued value is " + q.circularqDequeue());

//pushing more values into the queue, exception thrown when passing “Eight”

        q.circularqEnqueue("Six");

        q.circularqEnqueue("Seven");

        q.circularqEnqueue("Eight");

//popping all the values in the queue, exception thrown at the last dequeue

        System.out.println("\n" + q.circularqDequeue());

        System.out.println(q.circularqDequeue());

        System.out.println(q.circularqDequeue());

        System.out.println(q.circularqDequeue());

        System.out.println(q.circularqDequeue());

        System.out.println(q.circularqDequeue());

    }

}

### Output

### The following screenshots are the output of the program Java circular queue

Texto

Descripción generada automáticamente

This output shows correctly the functioning of a circular in Java, and handling the exceptions when the queue was empty and full, showing which value was not inserted.

### Code (C circular queues)

Note that the classes are separated in different files:

STRINGQUEUEH.h

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX 5

*//global vars*

extern char\* queue[MAX];

extern int front, rear;

void enqueue(const char\* *str*);

char\* dequeue();

void display();

MAIN.C

#include "../headers/stringQueueh.h"

#include <stdio.h>

#include <stdlib.h>

int main() {

//Queuing of values

    enqueue("One");

    enqueue("Two");

    enqueue("Three");

    enqueue("Four");

    enqueue("Five");

    display(); //display of the circular queue

//Dequeuing of values

    printf("Dequeued: *%s*\n", dequeue());

    printf("Dequeued: *%s*\n", dequeue());

    display();

//Dequeuing of all the circular list

    printf("Dequeued: *%s*\n", dequeue());

    printf("Dequeued: *%s*\n", dequeue());

    printf("Dequeued: *%s*\n", dequeue());

    display();

*// Clean up memory*

    while (front != -1) {

        free(dequeue());

    }

    return 0;

    }

QUEUE.C

*// Include a user-defined header file that presumably contains necessary declarations.*

#include "../headers/stringQueueh.h"

*// Define the maximum capacity of the queue as 'MAX'.*

char\* queue[MAX];

*// Initialize front and rear pointers to -1, indicating an empty queue.*

int front = -1, rear = -1;

*// Function to enqueue (add) a string to the queue.*

void enqueue(const char\* *str*) {

*// Check if the queue is full. If front is at the beginning and rear is at the end,*

*// or if rear is right behind front (circular queue logic), it's considered full.*

    if ((front == 0 && rear == MAX - 1) || (rear == (front - 1) % (MAX - 1))) {

        printf("Queue is full. Cannot enqueue.\n");

        return;

    }

*// If the queue is empty, set front and rear to 0.*

    if (front == -1) {

        front = rear = 0;

    } else {

*// Increment rear in a circular manner to add an element.*

        rear = (rear + 1) % MAX;

    }

*// Duplicate the input string and store it in the queue.*

    queue[rear] = strdup(*str*);

    printf("*%s* enqueued to the queue.\n", *str*);

}

*// Function to dequeue (remove) a string from the queue.*

char\* dequeue() {

*// Check if the queue is empty.*

    if (front == -1) {

        printf("Queue is empty. Cannot dequeue.\n");

        return NULL;

    }

*// Get the string at the front of the queue.*

    char\* str = queue[front];

*// If there was only one element in the queue, reset front and rear to -1 (empty).*

    if (front == rear) {

        front = rear = -1;

    } else {

*// Increment front in a circular manner to remove the element.*

        front = (front + 1) % MAX;

    }

*// Return the dequeued string.*

    return str;

}

*// Function to display the contents of the queue.*

void display() {

*// Check if the queue is empty.*

    if (front == -1) {

        printf("Queue is empty.\n");

        return;

    }

    int i = front;

    printf("Queue elements: ");

    while (1) {

*// Print the strings in the queue from front to rear.*

        printf("*%s* ", queue[i]);

        if (i == rear)

            break;

        i = (i + 1) % MAX;

    }

    printf("\n");

}

### Output

Texto

Descripción generada automáticamente

The output is the execution of the C implementation of a circular queue. The circular queue functions correctly as the first two elements are not appearing in the final dequeuing.

## Conclusions.

As we can see regularly in the report, the objective was successful, through the newest updates to java we as a team agreed that the java team is making an improvement in user (the dev) experience while using java, however we decided to not change the essence of java to determine the key difference between java and C which are that although C is faster in java we can maintain a very nice order among classes and code in general so in cases were we have a huge program java can be preferred over C language, nevertheless C is overall better.

## Bibliography

GeeksforGeeks. (2023). Introduction to circular queue. *GeeksforGeeks*. https://www.geeksforgeeks.org/introduction-to-circular-queue/