Stacks and Queues – Stack odds, queue evens

Laboratory Assignment 1

Beatriz Cortés Sánchez - 09098798K

Héctor Campos Romero - PUT YO DNI

Tabla de contenido

[1. Analysis 2](#_Toc434331697)

[ADT specifications 2](#_Toc434331698)

[Brief definition of operations 3](#_Toc434331699)

[2. Design 3](#_Toc434331700)

[Representation of ADTs 3](#_Toc434331701)

[UML and class diagram 5](#_Toc434331702)

[Explanation of the classes 5](#_Toc434331703)

[Behaviour of the program 6](#_Toc434331704)

[3. Implementation 7](#_Toc434331705)

[Difficult sections 7](#_Toc434331706)

[4. Review 8](#_Toc434331707)

[Big Oh notation 8](#_Toc434331708)

[5. References 8](#_Toc434331709)

# Analysis

## ADT specifications

Since we are expected to solve the problem in the assignment this way, the ADTs in this assignment are a stack and a queue. Two queues will be used but their implementation is the same, so one ADT is enough for both. Therefore we’ll be defining a stack and a queue.

-Stack specification:

Spec STACK[ITEM]

genre stack, item

operations

push: stack item -> stack

pop: stack -> item

giveTop: stack -> item

empty: stack -> boolean

endspec

-Queue specification:

Spec QUEUE[ITEM]

genre queue, item

operations

enqueue: queue item -> queue

dequeue: queue -> item

giveFront: queue -> item

empty: queue -> boolean

listQueue: void -> void

endspec

## Brief definition of operations

The ADT operations we will use are the basic ones (*pop*, *push*, *dequeue*, *enqueue*, etc) but in printing all elements in the queue at that moment. In both stacks and queues, we have decided the queue ADT we have added an operation called *listQueue* that will be used as output for what’s currently in the queue, and that we won’t be using the *makeNull* operation, so we won’t be listing or implementing it.

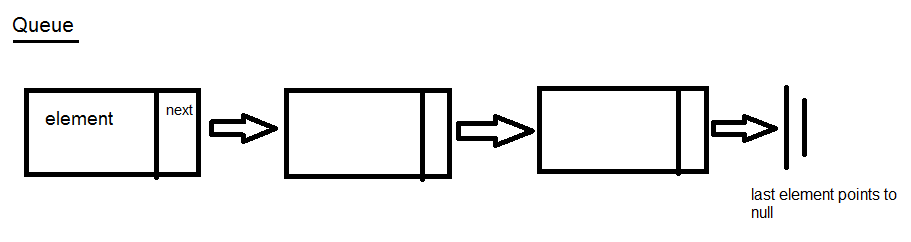
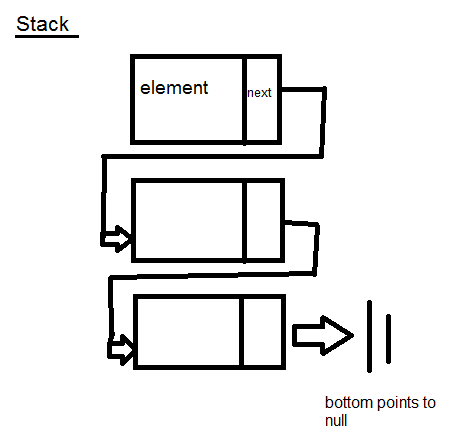
The operations giveFront and giveTop returns front and top attributes. They should be called front and top, but they are named this way because CodeLite does not allow you to call and attribute and another method in the same project with the same name.

# Design

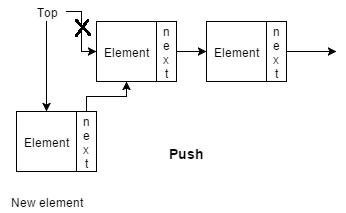
## Representation of ADTs

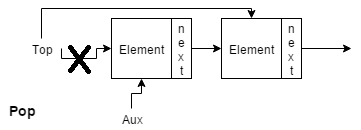
We decided to implement both ADTs using the pointer implementation due to the ability to share the Cell class in both of them. Cell is defines as an element and a pointer to the next cell.

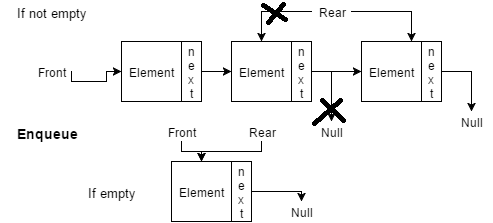
The diagrams of the ADTs in memory should be as the following, noticing each box represents a Cell:

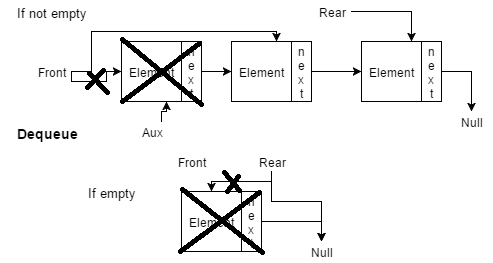


As for the ADT operations, we’ll be describing some of the basic ones in the following diagrams based upon the ones we’ve just seen to explain how they work:





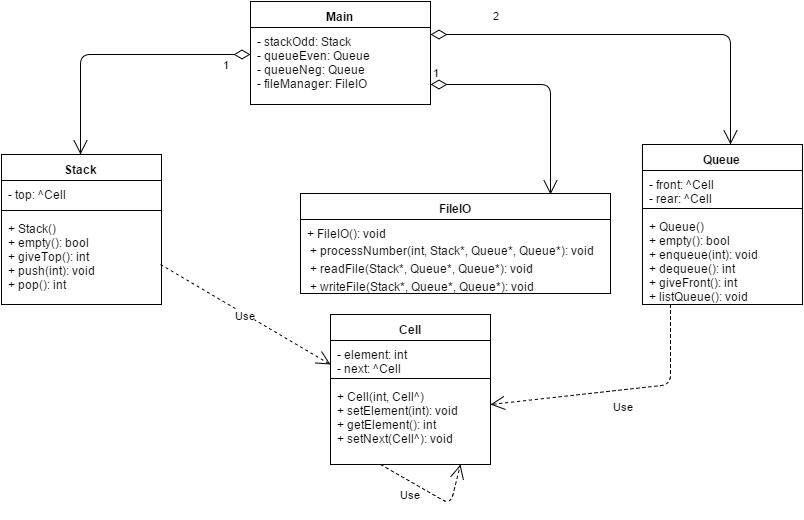




It’s also noted that listQueue would work by making a copy of the current queue and giving the output by simply dequeuing everything in it and printing it. While doing this would destroy the queue, we don’t need to worry as long as we’re doing it with a copy.

As for the classes themselves, we will be using the following class diagram. It lists the classes themselves as well as the attributes and the methods used by them.

## UML and class diagram



## Explanation of the classes

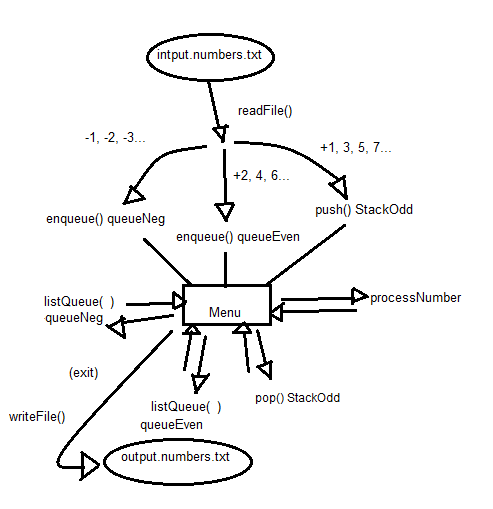
In this section, we explain the roles of each class:

* Cell class: Represents the items of the stack and the queue and because of that they will be a separated class, but common between them. It contains a Cell pointer (next) and an integer (element) as attributes, since we’ll be storing integer numbers and the pointer will lead to the next cell. As they are private, we use a method to set and to get the value for both for operations in the queue and stack.
* Queue class: This class is going to use the pointer implementation of the ADT because is easier to implement and to understand. The operations in the ADT will be the methods of the class and *front* and *rear* pointers to Cells will be private attributes. Since we’re not using these outside their own class, they don’t need get and set methods.
* Stack class: In this class we’ll be using the pointer implementation of the ADT as seen in the Queue class. The attribute will be the *top* Cell pointer and the ADT operations will be the methods.
* Main: The main file will contain the menu related part, which has a while loop to print the menu and wait for input option. This option is used in a switch to make the corresponding operations and when one is requested, main will simply call the corresponding operation.
* FileIO class: Input and output class to manage the files I/O. Has methods to read and write a file with the classification specified for both the input and output files. It consists only of *readFile* and *writeFile*. In the readFile method the file is read and each number is classified and stored in one of the three ADTs and the writeFile one writes each ADT in different lines of the file. It also holds the procedure to classify numbers both when reading from a file and when the user requests to classify one.

We are implementing this number of classes because we believe is better for the portability of the code, some of it can be used in future assignments to avoid wasting time implementing the same code.

## Behaviour of the program

The program works reading a .txt file, then its content is classified in the stack and queues depending on the type of the number. Next, the menu will be shown to the user, letting them choose between the options we were specified to add. If the user presses the exit option the program will finish after saving the queues and the list in a new .txt file, then closing. The following diagram gives an idea of the flow of the program itself:



# Implementation

## Difficult sections

The hardest parts to implement were notably the IO methods. Since we didn’t know how to work with files in C++, it took a good deal of research and tests to figure out how to correctly use the input file and create the output file. We ended up figuring out how to do it thanks to the fstream library. This class in particular took a good deal of work due to the problem of updating the storage classes (queues and stack) and passing them over with pointers. It ended up being fixed as well. A problem we had that wouldn’t let us compile and we didn’t understand why was simply a misspelling in the WriteFile method, where we spelled the declaration of the written file as “ifstream”, it was actually “ofstream”.

Another problem we ran into was displaying that the stack was empty when attempting to pop an empty stack. Since the operation itself is defined so it returns a number, we couldn’t just return without giving back anything, so we ended up setting it so it would return a number that wouldn’t be in the stack in any way since the stack we’re using will only hold odd positive numbers, so by retuning a 0 when it’s empty and taking that value as a signal that it’s empty, we can set in the menu a message to display it’s empty when pop returns a 0.

When reading and storing the numbers, we decided to set a simple format for them. For input.numbers.txt we simply give a list of numbers separated by commas and ending the line with a dot. The program will read until it reaches the dot. For output.numbers.txt, we decided to set three lines following the same pattern separating them by commas, but the first line holds odds numbers, then puts a dot, jumps a line and lists even numbers, and lastly a line for negative numbers. Example files have been provided in the debug folder.

We didn’t encounter any more problems when coding but the code itself is commented and can be checked. We tried to keep it as easy to read as we could.

# Review

## Big Oh notation

The program itself can have a different running time according to what the user selects, so let’s break it down in parts:

When the program opens, before the menu is shown, *readFile* is called. This method has a running time of O(m) as its running time depends of the length of the file introduced due to a while loop. Inside *readFile*, we call *processNumber*, which has a running time of O(1). This can call *enqueue* and *push*, that also have a running time of O(1) for its simple statements, so the final running time of *readFile* is O(m).

Once we reach the menu, it’s in a loop that will repeat as many times as the user wants. Inside, there are only O(1) statements. Let’s analyse the running time of the chosen options.

The first option uses *processNumber*, and we already know its running time is O(1). The second option simply pops the stack and prints the result. *Pop* uses simple statements, so its running time is O(1). The third and fourth options call *listQueue*. Its running time is O(m) as it will take longer to iterate through the whole Queue the longer it is. When we choose the final option, it breaks the loop to exit the program, but it first will call writeFile.

*WriteFile* is the last step of the program and it consists of three loops, but not nested. It simply does the same thing with three different objects: empty them and print them to a file. As it needs to iterate over all three of them, the initial running time would be O(3m). However, inside these loops, *dequeue*, *empty* and *pop* are called. We already know *pop*’s running time is O(1). *Dequeue* only uses simple statements even if it uses an if, so it’s also O(1). Empty is used both in Stack and Queue, but they both have a running time of O(1). Knowing all this, *writeFile*’s running time remains O(3m).

Taking all this into account, we can estimate the running time of a session by knowing what the user will select, but overall it’s not a bad running time for any of the operations.

(fill enhancements. Ideas??)

# References

<http://www.cplusplus.com/doc/tutorial/files/>

<http://stackoverflow.com/questions/7868936/read-file-line-by-line>

<http://www.cprogramming.com/tutorial/lesson10.html>

<https://www.draw.io/>