DATA STRUCTURES

FIRST ASSIGNMENT

Isabel Martínez Gómez

Javier García Jiménez

**-----------------------------------INDEX-------------------------------------**



UNIVERSIDAD DE ALCALÁ

GRADO EN INGENIERÍA INFORMÁTICA

2018-2019

**ANALYSIS**

1. Data flow from user to system and from system to user

2. ADT specifications:

1. Explanation of the adaptations made to the standard specification of the ADT fit requirements.

2. Reasoning on convenience or need of use of Dynamic DS in every case, comparing with Static DS characteristics.

3. Definition of operations of the ADT

1. Name, arguments and return values

**DESIGN**

1. Diagram of the representation of the ADT in the memory of the computer and explanation of every operation.

2. UML diagram and class diagram.

3. Explanation of the classes.

3.1 Explanation of ADT methods. Including those created for operations and any other needed

4. Explanation of the behaviour of the program.

**IMPLEMENTATIONS**

1. Diagram with source files and their relations

2. Explanation of every difficult section of the program

**REVIEW**

1. Running time of the solution (using big-Oh notation)

2. Possible enhancements for the solution (in terms of efficiency and others)

**REFERENCES (BIBLIOGRAPHY)**

**------------------------------------ANALYSIS-------------------------------**

**1.1. DATA FLOW FROM USER TO SYSTEM AND FROM SYSTEM TO USER**

This exercise has a class Restaurant which will interact with the user. First, the system welcomes the user and ask him what he wants to do. Depending on the user’s reply, the system will do one thing or another.

The user has the choice to choose anytime:

* Execute a program phase
* End the program

The system gives the possibility to execute all phases one by one, being these phases:

* Clients and first queue creation
* Plates preparation
* Entrance of the clients to the restaurant
* Meal and exit

**1.2. ADT SPECIFICATIONS**

**1.2.1 EXPLANATION OF THE ADAPTATIONS MADE TO THE STANDARD SPECIFICATION OF THE ADT FIT REQUIREMENT**

The standard specifications of the ADTs of the program have been adapted in order to fit the requirements of the assignment.

In StackElement (the Stack´s cell) two functions have been added:

* getLetter
* getNextLetter

These functions allow us to work with the StackElement items in an easier way.

Also, in the Stack a new extra function has been added:

* pushLetters

The objective of this function is filling the Stack with the 150 letters (consonants or vowels).

On the other hand, four functions have been added to QueueElement (the Queue´s cell):

* setClient
* getClient
* setNextClient
* getNextClient

These four functions are added so as to work in an easier way with the elements of the Queue.

Finally, in the Queue two functions have been added:

* getFront
* getRear

Just like the other functions this two ones allow us working in an easier way with the front and the rear of the Queue.

**1.2.2 REASONING ON CONVENIENCE OR NEED OF USE OF DYNAMIC DATA STRUCTURES IN EVERY CASE, COMPARING WITH STATIC DATA STRUCTURES CHARACTERISTICS**

As we already know, dynamic data structures offer us many advantages when we are going to program.

Throughout the exercise, we have used two essential dynamic data structures: stacks and queues.

In contrast to the static data structures, dynamic DS allows modifying their size while running, which is quite useful if the programmer doesn't know the size of the data to be stored. In addition, dynamic data structures makes a better efficient use of memory than static, this is because it only uses what memory is needed.

However, static data structures have been used on this program too. In the program there are three arrays: an array of consonants, an array of vowels and an array of tables. These types of structures are mostly used when the size is known by the programmer. When a program is started, the compiler allocates space for the array, for this reason sometimes memory is wasted.

The advantages of the static data structures compare to dynamics are that static ones are easier to program and check overflow, also allows arrays random access while dynamic DS only allows serial access, due to the serial access, dynamic DS searching is slower.

**1.2.2.3 DEFINITIONS OF THE OPERATIONS OF THE ADT**

**1.2.2.3.1 NAME, ARGUMENTS AND RETURN VALUES**

In this part, a brief explanation of the operations of queue and stack is provided. Moreover, a specification of the stack and queue is included to sum up how are the ADTs modified to achieve the objective of the program.

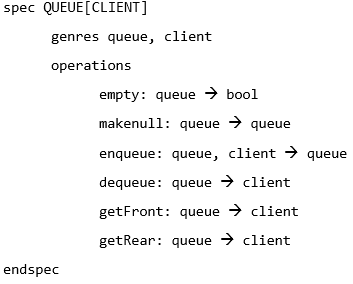
**Operations of stack:**

* **Empty:** returns true if the queue or stack is empty and false otherwise
* **Pop:** returns and delete the top element of the stack
* **Push:** insert a string at the top of the stack
* **pushLetters**: insert 150 random consonants or vowels into a stack

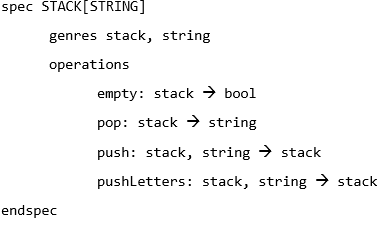
**Operations of queue:**

* **Empty:** returns true if the queue or stack is empty and false otherwise
* **Makenull:** makes the queue to be an empty queue
* **Enqueue**: inserts a queue element client at the end of the queue
* **Dequeue:** returns the first queue element client from the queue and delete it
* **getRear**: returns the last client of the queue
* **getFront:** returns the first client of the queue

**SPECIFICATION OF QUEUE**



**SPECIFICATION OF STACK**



**---------------------------------DESIGN-------------------------------------**

**2.1 DIAGRAM OF THE REPRESENTATION OF THE ADT IN THE MEMORY OF THE COMPUTER AND EXPLANATION OF EVERY OPERATION**

As a small introduction to the representation of the ADT in the memory of the computer, you will see below a representation of stacks and queues according to a pointer implementation.

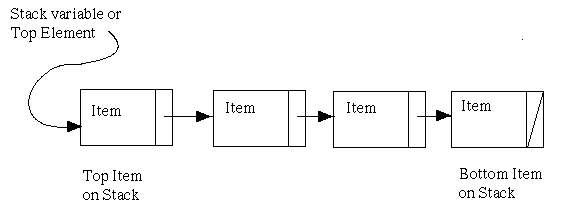
A pointer implementation is based on linked cells which have an item and a pointer indicating next cell.

The stack ADT has a LIFO behaviour (Last In First Out). It means, the last item introduced in the stack, is the first item retrieved from the stack.

A cursor called top indicates the current position of the first stack element. All the operations of the stack are going to occur on the top.

It is important to note that a stack grows towards decreasing memory positions being the bottom the maximum length of the stack.

The representation of a stack in the memory of the computer is similar to Fig 2.1

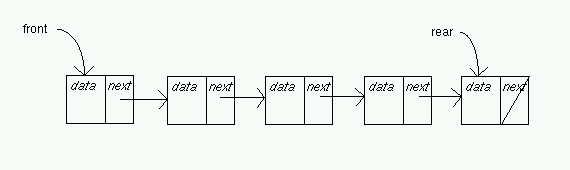


**Fig 2.1 Pointer Implementation of a stack**

A cursor called front indicates the current position of the first element in the queue whereas the rear indicates the last element of the queue.

Compared to the stack, the queue has a FIFO behaviour (First In First Out). It means, the first item in, is the first item out. In this way, we say that insertions (enqueue) are done in the rear and deletions (dequeue) are done in the front.

The representation of a queue in the memory of the computer is similar to Fig 2.2



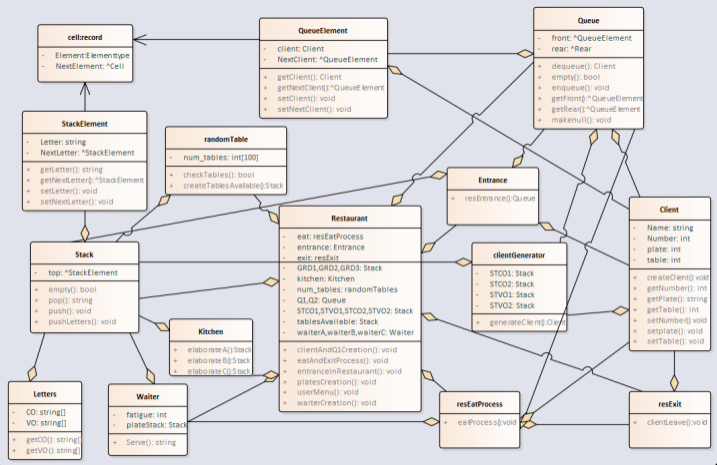
**Fig 2.2 Pointer implementation of queues**

**2.2 UML DIAGRAM AND CLASS DIAGRAM**

**CLASS DIAGRAM**

The below image Fig 2.3 represents the class diagram of the program in which the relationships of the classes are shown. This program consists of 14 classes and 1 record that will serve both ADTs.

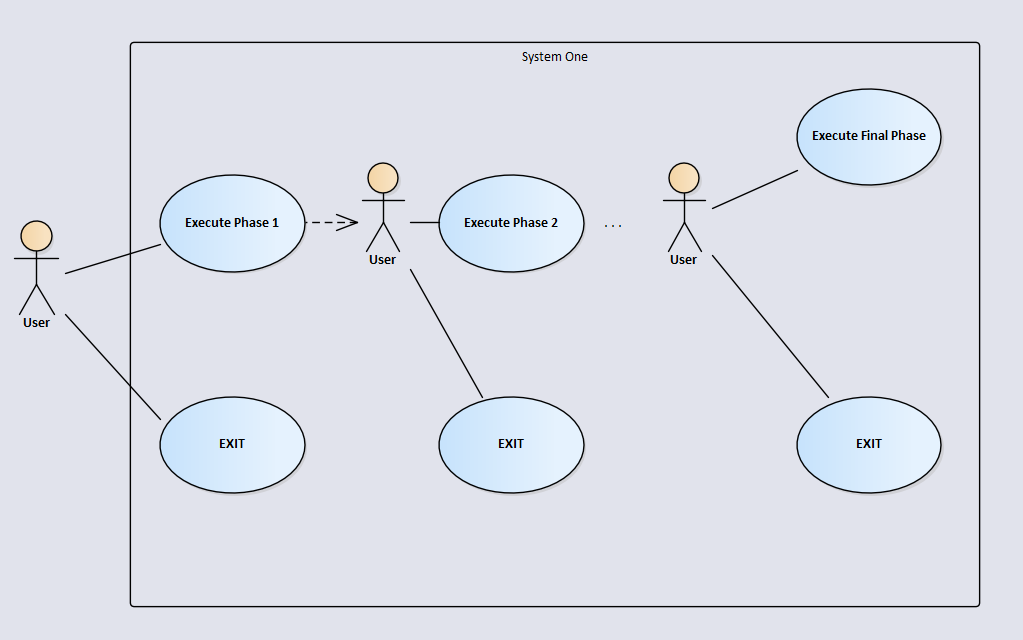
Each of the classes have a clear function of the program for example, the class Entrance manage the entrance of clients in the restaurant, Waiter manage how the waiters serve the plates, etc.



**Fig 2.3 Class diagram**

**UML CASE-USE DIAGRAM**

The attached image Fig 2.4 represents an UML diagram which represents how the user can interact with the system.



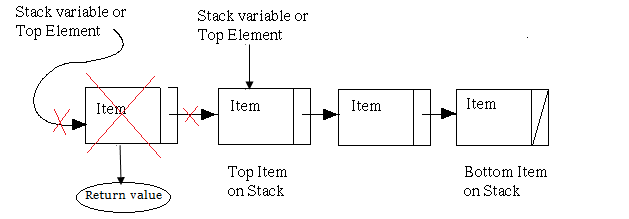
**Fig 2.4 Use-Case diagram**

**2.3 EXPLANATION OF CLASSES**

**2.3.1 EXPLANATION OF ADT METHODS. INCLUDING THOSE CREATED FOR OPERATIONS AND ANY OTHERS NEEDED**

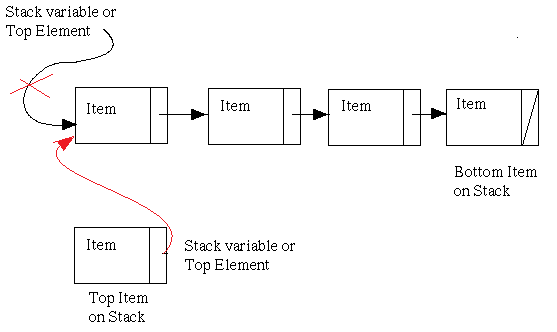
**STACK METHODS**

* Empty(): this method returns a boolean, this boolean will be true if the stack is empty and false if the stack is not empty. The way for checking if a stack is empty, is to see where the top is pointing, if is pointing to null the stack is empty.
* Pop(): returns and delete the top element of the stack. It first check if the stack is empty and if the stack is empty it occur an error, if it’s not empty it returns the element at the top of the stack and delete it, the top now is going to be the next element of the previous top. Fig 2.5



**Fig 2.5 POP()**

* Push(): inserts a string at the top of the stack. This method allocates a new cell and puts the top to point the new cell, this new cell is going to point the previous top cell. Fig 2.6



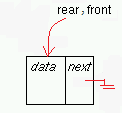
**Fig 2.6 PUSH()**

* PushLetters(): inserts 150 random consonants or vowels into a stack. This method pushes into a stack 150 random strings all vowels or consonants depending of the given parameter.

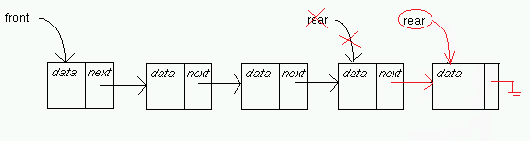
**QUEUE METHODS**

* Empty(): this method returns a boolean, this boolean will be true if the queue is empty and false if the queue is not empty. The way for checking if a queue is empty, is to see where the front is pointing, if the front is pointing to null the queue is empty.
* Makenull(): makes the queue to be an empty queue. It is achieved making the front and the rear point to null.
* Enqueue(): inserts a queue element at the end of the queue. It checks if the queue is empty, if the queue is empty it allocates a new cell which is going to be pointed by the front and by the rear, the new cell is going to point to null. Fig 2.7

Otherwise, if the queue is not empty we allocate a new cell which is going to be the new rear with the queue element introduced and will point to null. The previous rear is going to point the new rear. Fig 2.8

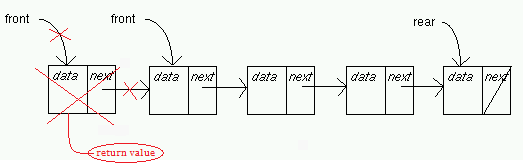


**Fig 2.7 ENQUEUE() if empty**



**Fig 2.8 ENQUEUE() if not empty**

* Dequeue(): deletes the element at the front of the queue and return it. It checks if the queue is empty and if it is empty an error occurs. If the queue is not empty, the element at the front is returned and deleted. Also, the new front will be the next cell pointed by the previous one. Fig 2.9



**Fig 2.9 DEQUEUE()**

* getFront(): returns the element of the front
* getRear(): returns the element of the rear

**2.4 EXPLANATION OF THE BEHAVIOUR OF THE PROGRAM**

This program is an object-oriented program that simulates the behaviour of a Restaurant. Thanks to the user, the program executes the phases of a Restaurant one by one.

First, a class Restaurant is created, and it initialized all the components that the Restaurant needs:

* Four stacks that will have letters
* Two queues that will have clients
* Another three stacks that will contain plates
* A stack that will be filled with numbers of tables
* A client generator that will be used to create clients with a random name and a number
* A class Entrance that will manage the entrance of the clients in the restaurant
* Three waiters that will serve the type of plates to the clients
* A class RandomTables that will manage the number of tables so as to don’t repeat any table that has been assigned to a client
* A class ResExit that will manage the exit of the clients showing their information
* A class EatProcess which is made to control the plates the clients are going to eat and how the waiters serve those plates

The class Restaurant will open a menu that gives two options to the user:

* Execute a program phase
* Exit the program

After each phase is executed, the program displays again the previous menu with the two options, so the user can execute the program phase by phase if he wants.

The phases of the program are:

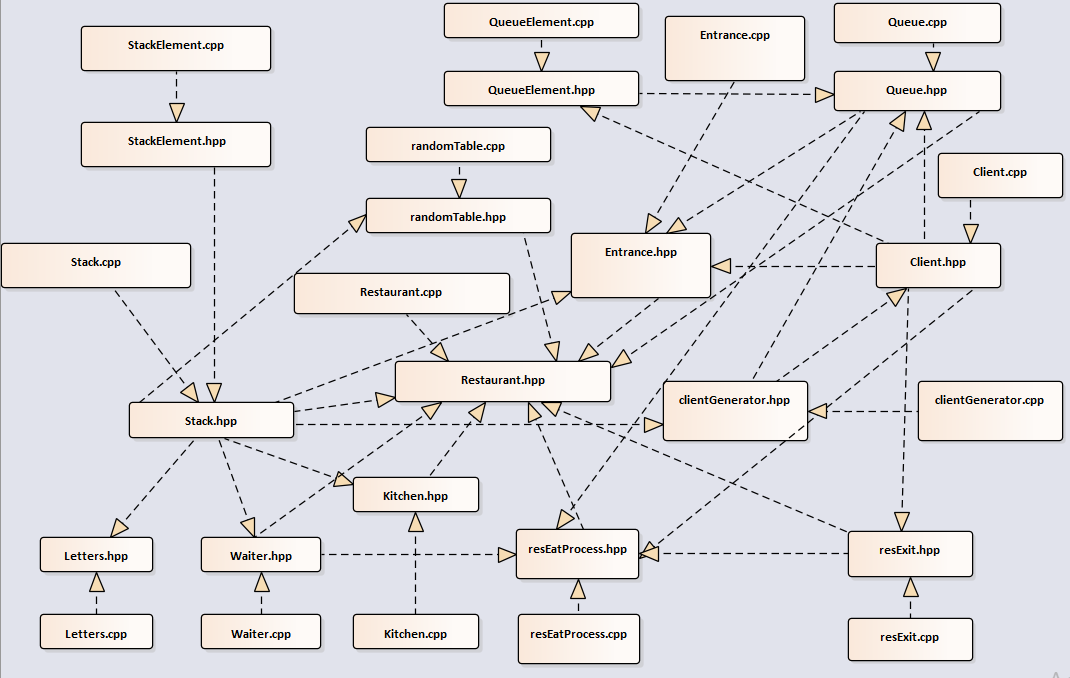
* **PHASE 1:** the program creates a random number of clients between 120-150 (the client generator creates a random name using the four letter stacks and after that, gives the client a number) and enqueues them in the first queue.
* **PHASE 2:** the program creates the plates. The three stacks of plates are filled with types A, B and C respectively.
* **PHASE 3:** the program dequeues the first queue, add a table to the clients dequeued and then, enqueues the clients in the second queue.
* **PHASE 4:** the program gives the waiters their correspondent stack. The clients of the second queue start to request their plates, the waiters serve the plates and finally the clients leave the restaurant showing their exit status (name, number, table and plate consumed).

**--------------------------IMPLEMENTATIONS-------------------------**

**3.1 DIAGRAM WITH SOURCE FILES AND THEIR RELATIONS**

In the image below, the relations of the files included on the program are shown (Fig 3.1). Each source file .hpp has an include of its correspondent .cpp and besides, includes every .hpp that the source file needs to execute its functions properly.

**Fig 3.1 Source files relations Diagram**



**3.2 EXPLANATION OF EVERY DIFFICULT SECTION OF THE PROGRAM**

In this section we find some of the difficulties of the exercise. Therefore, a screen capture of the code is attached with some explications in order to understand the purpose of the methods.

A complicated method throughout the program is createTablesAvailable. This method, returns a stack with tables numerated from 1 to 100 with no numbers repeated and without any order.

First, we have a static data structure num\_tables which contains the number of tables from 1 to 100. Also, we have a method checkTables which is the responsible for going throughout the array num\_tables looking for tables available.

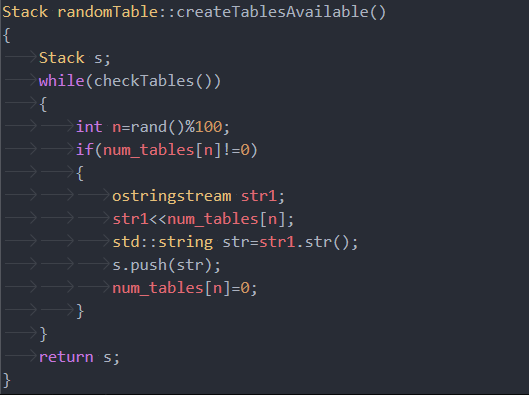
How can I look for tables available?

Every time that a number of table is pushed into the stack, this element in the array is changed to 0. Thanks to this, the method differentiates the available tables from the non-available ones.

Once we have understood it, method createTablesAvailable is going to push into the stack all the number of tables available in a random way and as we said previously, changing in the array the number of table to 0.

Finally, it must be emphasized that integers are not being introduced to the stack. Before doing the push, the integer is converted to string by using *ostringstream* and then pushing to a stack. This is done because the stacks in the program has strings inside them and it would not be effective to create a new type of stack with integers inside.

Look the attached Fig 3.2



**Fig 3.2 createTableAvailable()**

**---------------------------------REVIEW-------------------------------------**

**4.1. RUNNING TIMES OF OPERATIONS**

Last but not least, it is important to highlight how the operations of the program works in terms of efficiency. For this, in this section the running times of the program’s operations will be analysed.

* **Class Client:**
  + getPlate: O(1)
  + setPlate: O(1)
  + createclient: O(1)
  + getNumber: O(1)
  + setName: O(1)
  + getName: O(1)
  + getNumber: O(1)
* **Class clientGenerator:**
  + generateclient: O(n)
* **Class Entrance:**
  + resEntrance: O(1)
* **Class Kitchen:**
  + elaborateA: O(n)
  + elaborateB: O(n)
  + elaborateC: O(n)
* **Class Letters:**
  + getVO: O(1)
  + getCO: O(1)
* **Class Queue:**
  + enqueue: O(1)
  + makenull: O(1)
  + dequeue: O(1)
  + getFront: O(1)
  + getRear: O(1)
  + empty: O(1)
* **Class QueueElement:**
  + setNextClient: O(1)
  + setClient: O(1)
  + getNextClient: O(1)
  + getClient: O(1)
* **Class randomTable:**
  + createTablesAvailable: O(n2)
  + checkTables: O(n)
* **Class resEatProcess:**
  + eatProcess: O(1)
* **Class resExit:**
  + clientLeave: O(1)
* **Class Restaurant:**
  + clientsAndQ1Creation: O(n)
  + platesCreation: O(1)
  + entranceInRestaurant: O(n2)
  + waiterCreation: O(1)
  + eatAndExitProcess: O(1)
  + userMenu: O(n2)
* **Class Stack:**
  + push: O(1)
  + pushLetters: O(1)
  + pop: O(1)
  + empty: O(1)
* **Class StackElement:**
  + getLetter: O(1)
  + getNextLetter: O(1)
  + setNextLetter: O(1)
  + setLetter: O(1)
* **Class Waiter:**
  + Serve: O(1)

**4.2 POSSIBLE ENHANCEMENTS TO THE SOLUTION**

In conclusion, it must be mentioned that some parts of the code can be improved.

The function userMenu has a running time a little bit high, this can be improved by making a menu that is more organized and that interacts with the user in a more efficient way.

Even though this program is quite efficient because most of its functions are ran in a constant time O(1).

**-------------------REFERENCES (BIBLIOGRAPHY)-----------------**

In order to make this program the best way possible, we have searched information and we have looked into websites that helped us to do this work in terms of C++ code, issues… etc. Furthermore, in terms of explaining how the program works in a proper way we checked for some explanations of internet with the aim to have the ideas clear before starting to explain our code.

<https://stackoverflow.com/>

<http://www.cplusplus.com/>

<https://www.geeksforgeeks.org/static-data-structure-vs-dynamic-data-structure/>

Apart from looking websites, we have also have checked and used a lot of our class notes and the blackboard site of Data structures, that has helped us a lot in our project.