VIDEO GAME STORE

DIEGO FERNANDO HIDALGO LÓPEZ
LAURA DANIELA MARTÍNEZ ORTIZ

28 DE SEPTIEMBRE DE 2021

ALGORITHMS AND DATA STRUCTURES

ICESI UNIVERSITY

Engineering method

Problematic context

A foreign millionaire decided to invest and start in the city of Cali by creating a video game store that provides its services in an innovative way. For this reason, it requires a program that allows locals to learn a little about how this new store would work in the city through the simulation of the process of buying video games from the exit of section 1.

Developing the solution

In order to solve the problem previously presented will be made use of the indicated in the Method of the Engineering to develop a solution following a systematic approach and according to the proposed problematic situation.

Based on the description of the Engineering Method in the book "Introduction to Engineering" by Paul Wright, the steps to be followed in the development of the solution are presented.

1. Problem identification
2. Compilation of necessary information
3. Search of creative solutions
4. Transition from the formulation of ideas to preliminary designs
5. Evaluation and selection of the best solution
6. Preparation of reports and specifications
7. Design implementation

1. Problem identification

At this stage, the needs of the problematic situation are recognized, as well as its symptoms and conditions under which it must be resolved.

<u>Identification of needs and conditions</u>

- The owner wants to make known the innovative way in which he provides the service in the video game store.
- In order for the customer to receive an adequate service, he must go through different sections. The solution to the problem must ensure that:
    - Section 2 allows the customer to do the search, here is indicated the blocks or shelves where they should look for the game of their interest.
    - Section 3 of the physical copies of the games allows the customer to collect those present on their list in the order provided at the previous stage.
    - Section 4 allows the customer to perform the payment process.
- As a result, the application should report:
    - The order of departure of the clients.
    - The value of each purchase.
    - The order in which the games were packed.

<u>Problem definition</u>

The new video game store requires the development of a program that, through a simulation, allows it to make its innovative service known to the citizens of Cali.

2. Compilation of necessary information

In order to have complete clarity in the concepts involved, a search is made for the definitions of the terms most closely related to the problem presented.

The information found on the elements to be implemented is listed below:

Data structures

- Hash tables

  A hash table is a data structure that implements an associative array (dictionary). In an associative array, data is stored as a collection of key-value pairs. The position of the data within the array is determined by applying a hashing algorithm to the key - a process called hashing. The hashing algorithm is called a hash function.

  Source: Hash table. (s. f.). isaac computer science. Recovered from: https://isaaccomputerscience.org/concepts/dsa_datastruct_hash_table?examBoard=all&stage=all

- Queue

  The Java Queue interface, represents a data structure designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue.

  Source: Jenkov, J. (2020). Java Queue. Jenkov.Com. Recovered from: http://tutorials.jenkov.com/java-collections/queue.html

- Stack

  The Java Stack class is a classical stack data structure. You can push elements to the top of a Java Stack and pop them again, meaning read and remove the elements from the top of the stack.

  Source: Jenkov, J. (2020). Java Stack. Jenkov.Com. Recovered from: http://tutorials.jenkov.com/java-collections/stack.html

Sorting algorithm

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Source: Geeks for Geeks. (2021). Sorting Algorithms. Geeks for Geeks. Recovered from: https://www.geeksforgeeks.org/sorting-algorithms/

Generics

Generics mean parameterized types. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

Source: Geeks for Geeks. (2021). Sorting Algorithms. Geeks for Geeks. Recovered from: https://www.geeksforgeeks.org/generics-in-java/

Abstract Data Type (ADT)

The abstract data type is a special kind of data type, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working is totally hidden from the user. The ADT is made of primitive data types, but operation logics are hidden.

Source: Chakraborty, A. (2019). Abstract Data Type in Data Structures. Tutorials Point. Recovered from: https://www.tutorialspoint.com/abstract-data-type-in-data-structures

3. **Search of creative solutions**

At the end of the brainstorming, the following solutions are proposed:

- *Alternative 1:* Implement an application whose user is a customer, in which they take care of passing through the service sections.
- *Alternative 2:* Implement an application with an administrator user who coordinates the purchase process of each client.
- *Alternative 3:* Implement an application whose users are the customers, in which each one of them, from their session, can enter and perform the purchase process.

4. **Transition from the formulation of ideas to preliminary designs**

First, ideas that are not feasible are discarded. We decided to discard alternative 3 because we do not have the knowledge to make an application in which several users can be connected at the same time. In addition, to simulate the process properly, a group of people would have to be coordinated to log in at the same time.

The review of the other 2 alternatives leads us to the following:

- *Alternative 1.*

  - It begins with the creation of an administrator user who is responsible for coordinating the registration of shelves, games and the number of cashiers available during the day.
  - Each customer is allowed to register in the system.
  - The client accesses the system and records their list of games. (final section 1).
  - The customer goes to section 2 and chooses the algorithm that he wants to use to sort his game list.
  - Then the customer goes to section 3 where they can see their shopping cart and how long it took them to collect each game. At this point they should leave the system and wait for other users to get to this section in order to move to the last section.
  - Once all customers perform the above process, the row is organized to pay based on their recorded time up to section 3.
  - Finally it shows the way in which the bag of the purchase of the customers remains and their order of departure.

- *Alternative 2.*

  - It is managed by a single user (the administrator).
  - It is divided into the sections defined by the statement.
  - In the first part, the basic data is requested in order to simulate the process (number of cashiers available, games, shelves, customer identification).
  - In the first section, the manager adds the games to each of the previously registered customers.
  - In the second section, each client is given two different sorting algorithms to choose from in order to sort the list of video games.

- The third section shows the times that customers take after collecting the games and how their shopping basket looks, in addition to showing them in the order in which they will pay in the fourth section.
- Finally, in the fourth section, the way in which each client's games are packed and their order of departure is shown.

5. Evaluation and selection of the best solution

As the engineering design process evolves, the engineer can evaluate alternative ways to solve the problem in question. Commonly, the engineer abandons the design possibilities that are not promising, thus obtaining a set progressively smaller of options. Feedback, modification and evaluation can occur repeatedly as the device or system evolves from concept to final design.

Criteria

The criteria we chose in this case are those listed below. Next to each criteria, a numerical value has been established with the aim of establishing a weight that indicates which of the possible values of each criteria have more weight and, therefore, are more desirable.

- *Criteria A.* Time taken to complete the simulation under the same conditions.
  - [3] Little time.
  - [2] Normal time.
  - [1] Long time.
- *Criteria B.* Results.
  - [2] Correct order.
  - [1] Wrong order.
- *Criteria C.* Ease of administration.
  - [3]Easy.
  - [2] Normal.
  - [1] Difficult.
- *Criteria D.* Description of each section.
  - [3] Complete.
  - [2] Partially complete.
  - [1] Incomplete.

Evaluation

Evaluating the above criteria in the alternatives that are maintained, we obtain the following table:

	Criteria A	**Criteria B**	**Criteria C**	**Criteria D**	**Total**
Alternative 1: Several users (customers)	1 Long time	2 Correct order	1 Difficult	2 Partially complete	6
Alternative 2: One user (administrator)	2 Normal time	2 Correct order	3 Easy	3 Complete	10

Selection

According to the previous evaluation, the Alternative 2 must be selected because it obtained the highest score according to the defined criteria.

6. **Preparation of reports and specifications**

   Review the functional and non-functional requirements section and the class diagram.

7. **Design implementation**

   Review the implemented project.

Specification of requirements

Functional requirements

In order to correctly meet with the needs and functionalities required for this project, the system to be developed must be able to:

FR1: **Manage the video game shelves of the store** with a letter that identifies them, an amount of video games, and the video games that each shelf contains.

- **FR1.1 -** It must be possible to register a new shelf.
- **FR1.2 -** It must be possible to add a new game to a specific shelf.

FR2: **Manage the video game catalog of the store** with a shelf where the video game it's placed, a code, an amount of available copies and a price.

- **FR2.1 -** It must be possible to register a new video game.

FR3: **Manage the cashiers to be used during the day.**

- **FR3.1 -** It must be possible to register the amount of cashiers available during the day.
- **FR3.2 -** The cashier must indicate if it's being used by a customer.

FR4: **Manage the customers that enter the store** with an identification number and a list of video games.

- **FR4.1 -** It must be possible to register a new customer.
- **FR4.2 -** The customers must be registered in the same order they entered the store.

Non-functional requirements

In order to guarantee the correct operation of the system and assure the quality of the software, the system must have the next validations:

NFR1: **Tests.** Implement tests to make sure that the methods work correctly.
NFR2: **Sorting algorithms.** Sort the list of games of the clients.
NFR2: **Generics.**

Time complexity analysis and
Spatial complexity analysis

Insertion Sort

- **Spatial complexity analysis**

Type	**Variable**	**Atomic values**
Input	n	1
Auxiliary	i j	1 1
Output	-	-
Total = O(1)		

- **Time complexity analysis**

private <E extends Comparable<E>> void insertionSort (LinkedList<E> list)			
#	**Instruction**	**Number of times it is repeated**	
1	int n = list.size()	1	C_1
2	for(int i = 1; i < n; i ++)	n + 1	C_2
3	E key = list.get(i)	n	C_3
4	int j = i - 1	n	C_4
5	while(j >= 0 && list.get(j).compareTo(key) > 0)	$\frac{n(n+1)}{2} + 1$	C_5
6	list.setElement(j + 1, list.get(j))	$\frac{n(n+1)}{2}$	C_6
7	j = j - 1	$\frac{n(n+1)}{2}$	C_7
8	list.setElement(j + 1, key)	n	C8

Characterization:

- n: input size.

$$T(n) = \left(c_1\right) + c_2(n+1) + c_3 n + c_4 n + c_5\left(\frac{n(n+1)}{2} + 1\right) + c_6\left(\frac{n(n+1)}{2}\right) + c_7\left(\frac{n(n+1)}{2}\right) + c_8 n$$

$$T(n) = c_1 + c_2 n + c_2 + c_3 n + c_4 n + c_8 n + c_5 + c_5\frac{n^2}{2} + c_5\frac{n}{2} + c_6\frac{n^2}{2} + c_6\frac{n}{2} + c_7\frac{n^2}{2} + c_7\frac{n}{2}$$

$$T(n) = n^2\left(\frac{1}{2}\left(c_5 + c_6 + c_7\right)\right) + n\left(c_2 + c_3 + c_4 + c_8 + \frac{1}{2}\left(c_5 + c_6 + c_7\right)\right) + c_1 + c_2 + c_5$$

$$a = \frac{1}{2}\left(c_5 + c_6 + c_7\right)$$

$$b = c_2 + c_3 + c_4 + c_8 + \frac{1}{2}\left(c_5 + c_6 + c_7\right)$$

$$c = c_1 + c_2 + c_5$$

$$T(n) = an^2 + bn + c$$

$$T(n) = O(n^2)$$

Bubble Sort

- **Spatial complexity analysis**

Type	**Variable**	**Atomic values**
Input	n	1
Auxiliary	i j	1 1
Output	-	-
Total = O(1)		

- **Time complexity analysis**

private <E extends Comparable<E>> void bubbleSort (LinkedList<E> list)			
#	Instruction	Number of times it is repeated	
1	int n = list.size()	1	C_1
2	for(int i = 0; i < n; i ++)	n + 1	C_2
3	for(int j = 1; j < (n - i); j ++)	$\frac{n(n+1)}{2} + 1$	C_3
4	if(list.get(j - 1).compareTo(list.get(j)) > 0)	$\frac{n(n+1)}{2}$	C_4
5	E aux = list.get(j - 1)	$\frac{n(n+1)}{2}$	C_5
6	list.setElement(j - 1, list.get(j))	$\frac{n(n+1)}{2}$	C_6
7	list.setElement(j, aux)	$\frac{n(n+1)}{2}$	C_7

Characterization:

- n: input size

$$T(n) = (c_1 * 1) + (c_2 * (n + 1)) + (c_3 * \frac{n(n+1)}{2} + 1) + (c_4 * \frac{n(n+1)}{2}) + (c_5 * \frac{n(n+1)}{2}) + (c_6 * \frac{n(n+1)}{2}) + (c_7$$

$$T(n) = c_1 + c_2 n + c_2 + c_3 + c_3 \frac{n^2}{2} + c_3 \frac{n}{2} + c_4 \frac{n^2}{2} + c_4 \frac{n}{2} + c_5 \frac{n^2}{2} + c_5 \frac{n}{2} + c_6 \frac{n^2}{2} + c_6 \frac{n}{2} + c_7 \frac{n^2}{2} + c_7 \frac{n}{2}$$

$$T(n) = c_3 \frac{n^2}{2} + c_4 \frac{n^2}{2} + c_5 \frac{n^2}{2} + c_6 \frac{n^2}{2} + c_7 \frac{n^2}{2} + c_2 n + c_3 \frac{n}{2} + c_4 \frac{n}{2} + c_5 \frac{n}{2} + c_6 \frac{n}{2} + c_7 \frac{n}{2} + c_2 + c_1$$

$$T(n) = n^2\left(\frac{1}{2}\left(c_3 + c_4 + c_5 + c_6 + c_7\right)\right) + n\left(c_2 + \frac{1}{2}\left(c_3 + c_4 + c_5 + c_6 + c_7\right)\right) + c_1 + c_2$$

$$a = \frac{1}{2}\left(c_3 + c_4 + c_5 + c_6 + c_7\right)$$

$$b = c_2 + \frac{1}{2}\left(c_3 + c_4 + c_5 + c_6 + c_7\right)$$

$$c = c_1 + c_2$$

$$T(n) = an^2 + bn + c$$

Abstract Data Type (ADT)

HashTable

ADT HashTable		
HashTable = {<k_1, k_2, k_3, … ,k_n>, <v_1, v_2, v_3, … ,v_n>, Count = <count>, Capacity = <capacity>}		
{inv: Each key k is associated with a value v: $k_1 = v_1$, $k_2 = v_2$, $k_3 = v_3$, … , $k_n = v_n$}		
Primitive operations:		
HashTable:		→HashTable
Size:	HashTable	→Integer
GetCapacity:	HashTable	→Integer
HashCode:	HashTable x Key	→Integer
Insert:	HashTable x Key x Value	→HashTable
Search:	HashTable x Key	→Value
Remove:	HashTable x Key	→ Boolean

HashTable() → Constructor
“Creates an empty hash table (with size equal to 0) to add new values with their hash codes.”
{pre: TRUE}
{post: HashTable = {Count = 0, Capacity = 100}}

Size() → Analyzer
“Returns the number of values in the hash table as an integer variable”
{pre: HashTable = {Count = <count>, Capacity = <capacity>}}
{post: <count>}

GetCapacity() → Analyzer
“Returns the capacity of the hash table as an integer variable”
{pre: HashTable = {Count = <count>, Capacity = <capacity>}}
{post: <capacity>}

HashCode(K key) → Analyzer
“Returns the hash code of a key as an integer variable”
{pre: HashTable = {<k_1, k_2, k_3, … ,k_n>, <v_1, v_2, v_3, … ,v_n>, Count = <count>, Capacity = <capacity>}}
{post: hash code}

Insert(K key, V value) → Modifier
“Inserts a new value to the hash table, associating a key with a value”
{pre: HashTable h = <k_1, k_2, k_3, … ,k_n>, <v_1, v_2, v_3, … ,v_n>, Key k, Value v}
{post: HashTable h = <k_1, k_2, k_3, … ,k_n, k >, <v_1, v_2, v_3, … ,v_n, v >, h.size() = h.size() + 1}

Search(K key) → Analyzer
“Returns the value associated to a key”
{pre: HashTable h = <k_1, k_2, k_3, … ,k_n>, <v_1, v_2, v_3, … ,v_n>, Key k, Value v}
{post: Value v associated to the Key k or null if the Key does not exist}

Remove(K key) → Modifier
“Removes a value in the hash table”
{pre: HashTable h = <k_1, k_2, k_3, … ,k_n>, <v_1, v_2, v_3, … ,v_n>, Key k, Value v}
{post: TRUE if hashTable.count = hashTable.count - 1 FALSE if the Key does not exist}

Queue

ADT Queue
Queue = {<e_1, e_2, e_3, … ,e_n>, Node = <front>, LastNode = <rear>, Size = <size>}
{inv: For all Nodes e it is fulfilled that $e_n = last$, $e_1 = first$, and the first one added is the first one out; $0 \leq n \wedge$ Size(Queue) = n $\wedge$ front = e_1 $\wedge$ rear = e_n}
Primitive operations:

Queue:		→Queue
IsEmpty:	Queue	→Boolean
Size:	Queue	→Integer
Enqueue:	Queue x Element	→Queue
Dequeue:	Queue	→Element
Front:	Queue	→Element
Rear:	Queue	→Element
Reverse:	Queue	→Queue

Queue() → Constructor
"Creates an empty queue (with size equal to 0) to add new elements."
{pre: TRUE}
{post: Queue = {Node = null, LastNode = null, Size = 0}}

IsEmpty() → Analyzer
"Checks if the queue is empty "
{pre: Queue = {Node = <front>, LastNode = <rear>, Size = <size>} }
{post: TRUE if queue.size = 0 FALSE if the queue isn't empty}

Size() → Analyzer
"Returns the number of nodes in the queue as an integer variable"
{pre: Queue = {Node = <front>, LastNode = <rear>, Size = <size>}}
{post: <size>}

Enqueue(E e) → Modifier
"Inserts a new element e at the bottom of the queue"
{pre: : Queue q = <e_1, e_2, e_3, … ,e_n> and element e or q = ∅ and element e}
{post: Queue q = <e_1, e_2, e_3, … ,e_n, e > or q = < e >, q.size = q.size + 1}

Dequeue() → Modifier
“Extracts the element in Queue q’s front”
{pre: Queue q = <e_1, e_2, e_3, … ,e_n> }
{post: Queue q = < e_2, e_3, e_4, … ,e_{n-1} > and Element e_1, q.size = q.size - 1}

Front() → Analyzer
“Recovers the value of the element on the front of the queue”
{pre: Queue q = <e_1, e_2, e_3, … ,e_n>}
{post: Element e_1}

Rear() → Analyzer
“Recovers the value of the last element of the queue”
{pre: Queue q = <e_1, e_2, e_3, … ,e_n>}
{post: Element e_n}

Reverse() → Modifier
“Builds a reverse queue from a previous one”
{pre: Queue q = <e_1, e_2, e_3, … ,e_n>}
{post: Queue r = <e_n, … , e_3, e_2, e_1>}

Stack

ADT Stack
Stack = {<e_1, e_2, e_3, … ,e_n>, Node = <top>, Size = <size>}
{inv: For all Nodes e it is fulfilled that $e_n = last$, $e_1 = first$, and the last element is the top and last in is the first out, 0 ≤ n ∧ Size(Stack) = n ∧ top = e_n}
Primitive operations: Stack: →Stack

IsEmpty:	Stack	→Boolean
Size:	Stack	→Integer
Push:	Stack x Element	→Stack
Pop:	Stack	→Stack
Top:	Stack	→Element
Peek:	Stack	→Element
Reverse:	Stack	→Stack

Stack() → Constructor
"Creates an empty stack (with size equal to 0) to add new elements."
{pre: TRUE}
{post: Stack = {Node = null, Size = 0}}

IsEmpty() → Analyzer
"Checks if the stack is empty "
{pre: Stack = {Node = <top>, Size = <size>} }
{post: TRUE if stack.size = 0 FALSE if the stack isn't empty}

Size() → Analyzer
"Returns the number of nodes in the stack as an integer variable"
{pre: Stack = {Node = <front>, Size = <size>}}
{post: <size>}

Push(E e) → Modifier
"Adds a new element e to the stack"
{pre: Stack s = <e_1, e_2, e_3, … ,e_n> and element e or s = ∅ and element e}
{post: Stack s = <e_1, e_2, e_3, … ,e_n, e > or s = < e >, s.size = s.size + 1}

Pop() → Modifier

“Extracts the most recently inserted element from the stack”
{pre: Stack s ≠ ∅ o s = <e_1, e_2, e_3, … ,e_n>}
{post: Stack s = <e_1, e_2, e_3, … ,e_{n-1}>, s.size = s.size - 1}

Top() → Analyzer
“Recovers the value of the element on the top of the stack”
{pre: Stack s ≠ ∅ o s = <e_1, e_2, e_3, … ,e_n>}
{post: Element e_n}

Peek() → Analyzer
“Views the first element in the stack but does not remove it.”
{pre: Stack s ≠ ∅ o s = <e_1, e_2, e_3, … ,e_n>}
{post: Element e_n}

Reverse() → Modifier
“Builds a reverse stack from a previous one”
{pre: Stack s = <e_1, e_2, e_3, … ,e_n>}
{post: Stack r = <e_n, … , e_3, e_2, e_1>}

Linked List

ADT LinkedList		
LinkedList = {Node = <first>, Size = <size>}		
{inv: For all Nodes e it is fulfilled that $n_1 \leq n_2 \leq n_3 \leq \ldots \leq n_n$ V $n_1 \geq n_2 \geq n_3 \geq \ldots \geq n_n$}		
Primitive operations: List IsEmpty: Size:	 List List	 →List →Boolean →Integer

Add:	List x Element	→List
Remove:	List x Element	→List
IsInList:	List x Element	→Boolean

List → Constructor
“Creates an empty linked list (with size equal to 0) to add new elements.”
{pre: TRUE}
{post: linkedList = {Node = null, Size = 0}}

IsEmpty() → Analyzer
“Checks if the linked list is empty ”
{pre: linkedList = {Node = <first>, Size = <size>} }
{post: TRUE if linkedList.size = 0 FALSE if the linked list isn’t empty}

Size() → Analyzer
“Returns the number of nodes in the linked list as an integer variable”
{pre: linkedList = {Node = <first>, Size = <size>}}
{post: <size>}

Add(E e) → Modifier
“Adds a new element e to the linked list”
{pre:True}
{post: linkedList.Node = e, linkedList.size = linkedList.size + 1}

Remove(E e) → Modifier
“Removes a node from the linked list”
{pre: e ϵ linkedList}
{post: linkedList.size = linkedList.size - 1}

IsInList(E e) → Analyzer
“Checks if the element is on the linked list”
{pre: TRUE}
{post: TRUE if the element is part of the linked list FALSE if not}

Design of unit tests
Configuration of the scenes

Name	Class	Scenes
setupScenary1	StoreTest	A Client has been added: id = “12345” A Video game has been added: code = 612 name = “GTA” quantity = 3 shelf = “A” price = 17,000 A Shelf has been added: identifier = “A”
setupScenary1	HashTableTest	HashTable<Integer, VideoGame> htable = new HashTable<>(1)
setupScenary1	QueueTest	Queue<Client> queue = new Queue<Client>()
setupScenary1	StackTest	Stack<VideoGame> shoppingCart = new Stack<VideoGame>()
setupScenary2	StoreTest	3 Clients has been added: id = “13579” id = “24680” id = “54321” 4 Video games has been added: code = 612 name = “GTA” quantity = 3 shelf = “A” price = 17,000 code = 331 name = “TS” quantity = 5 shelf = “B” price = 15,000 code = 767 name = “DB” quantity = 2 shelf = “C” price = 10,000 code = 287 name = “Cars” quantity = 6 shelf = “D”

		price = 13,000 3 Shelves has been added: identifier = “A” identifier = “B” identifier = “C” identifier = “D”
setupScenary2	HashTableTest	HashTable<Integer, VideoGame> htable = new HashTable<>(7)
setupScenary2	StackTest	Stack<VideoGame> shoppingCart = new Stack<VideoGame>()
setupScenary3	StoreTest	2 Clients has been added: id = “13579” id = “24680” 2 Video games has been added: code = 612 name = “GTA” quantity = 3 shelf = “A” price = 17,000 code = 331 name = “TS” quantity = 5 shelf = “B” price = 15,000 2 Shelves has been added: identifier = “A” identifier = “B”
setupScenary3	HashTableTest	HashTable<Integer, VideoGame> htable = new HashTable<>(4)
setupScenary3	StackTest	Stack<VideoGame> shoppingCart = new Stack<VideoGame>()

Design of test cases

Methods to add

Purpose of the test: Verify that a video game is added correctly				
Class	**Method**	**Scenes**	**Inputs**	**Results**
Store	registerVideoGame	setupScenary1	A Shelf has been	The game is

			added: identifier = “B” code = 308 name = “FIFA” quantity = 4 shelf = B price = 15,000	added and the number of registered games is now 2.
Store	registerVideoGame	setupScenary2	A Shelf has been added: identifier = “E” code = 308 name = “FIFA” quantity = 4 shelf = “E” price = 15,000	The game is added and the number of registered games is now 5.
Store	registerVideoGame	setupScenary3	A Shelf has been added: identifier = “C” code = 308 name = “FIFA” quantity = 4 shelf = “C” price = 15,000	The game is added and the number of registered games is now 3.

Purpose of the test: Verify that a video game is added correctly to the client list.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
Store	addVideoGameTo Client	setupScenary1	Same as the scenary. The client is already registered.	The game is added to the client list and the number of games is now 1.
Store	addVideoGameTo Client	setupScenary2	Same as the scenary. The client is already registered.	The games are added to the client list and the number of games is now 4.
Store	addVideoGameTo Client	setupScenary3	Same as the scenary. The client is already registered.	The games are added to the client list and the number of games is now 2.

Methods to search

Purpose of the test: Verify that a client can be found.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
Store	searchClient	setupScenary1	id = “56789”	true The value entered corresponds to the registered customer.
Store	searchClient	setupScenary2	id = “45678”	true The value entered corresponds to the registered customer.
Store	searchClient	setupScenary3	id = “10345”	true The value entered corresponds to the registered customer.

Purpose of the test: Verify that the video game’s info is correct according to the code given				
Class	**Method**	**Scenes**	**Inputs**	**Results**
Store	getVideoGameInfo	setupScenary1	code = 612	shelf = “A” code = 612 name = “GTA” price = 17000.0 quantity = 3
Store	getVideoGameInfo	setupScenary2	code = 767	shelf = “C” code = 767 name = “DB” price = 10000.0 quantity = 2
Store	getVideoGameInfo	setupScenary3	code = 331	shelf = “B” code = 331 name = “TS” price = 15000 quantity = 5

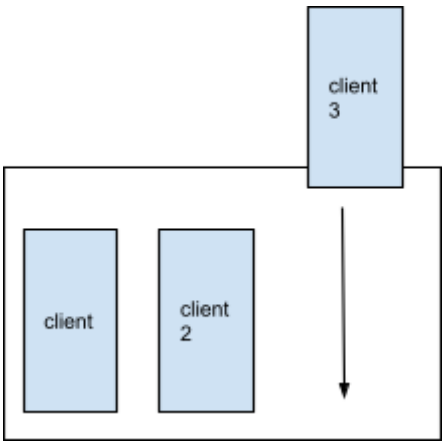
Design of test cases for each data structure

Purpose of the test: Validates the insert method in the Hash table.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
HashTable	insert()	setupScenary1	testGame = new VideoGame(612, "GTA", 3, "A", 17000)	Correctly adds the game. And the method search() is used to verify the information. The size of the hash table is 1.
HashTable	insert()	setupScenary2	VideoGame testGame = new VideoGame(331, "TS", 5, "A", 15000) VideoGame testGame2 = new VideoGame(121, "ZL3", 5, "B", 12000) VideoGame testGame3 = new VideoGame(211, "SD2", 5, "C", 15000) VideoGame testGame4 = new VideoGame(678, "MN", 5, "D", 15000) VideoGame testGame5 = new VideoGame(123, "AGK", 5, "E", 15000) VideoGame testGame6 = new VideoGame(451, "TWD", 5, "F", 15000) VideoGame testGame7 = new VideoGame(301, "GR", 5, "G", 15000)	Correctly adds the games. And the method search() is used to verify the information. The size of the hash table is 7.
HashTable	insert()	setupScenary3	VideoGame testGame = new VideoGame(331, "TS", 5, "A", 15000) VideoGame testGame2 = new VideoGame(121, "ZL3", 5, "B", 12000) VideoGame testGame3 = new VideoGame(211, "SD2", 5, "C", 15000) VideoGame testGame4 = new VideoGame(678, "MN", 5, "D", 15000)	Correctly adds the games. And the method search() is used to verify the information. The size of the hash table is 4.

Purpose of the test: Validates that the method enqueue in the Queue works.

Class	Method	Scenes	Inputs	Results
Queue	Enqueue()	setupScenary1	Client client = new Client("12345")	A new client is added to the queue. To verify it, we used the methods isEmpty() and isInQueue()
Queue	Enqueue()	setupScenary1	Client client = new Client("13579"); Client client2 = new Client("24680"); Client client3 = new Client("54321");	3 new clients are added to the queue. To verify it, we used the methods isEmpty() and isInQueue(). Also we used front() to make sure that client is the first one and rear() to make sure that the client3 is the last. The size is 3.
Queue	Enqueue()	setupScenary1	Client client = new Client("13579") Client client2 = new Client("24680") Client client3 = new Client("54321") Client client4 = new Client("56791") Client client5 = new Client("87634") Client client6 = new Client("23678")	6 new clients are added to the queue. To verify it, we used the methods isEmpty() and isInQueue(). Also we used front() to make sure that client is the first one and rear() to make sure that the client6 is the last. The size is 6.

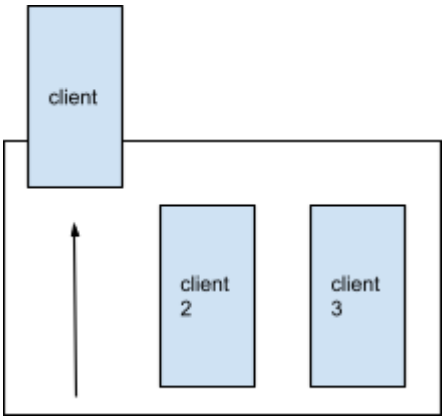
Enqueue:



Purpose of the test: Validates that the method dequeue in the Queue works.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
Queue	Dequeue()	setupScenary1	Client client = new Client("12345")	A new client is added to the queue, then it´s removed. To verify it, we used the methods isEmpty() and isInQueue()
Queue	Dequeue()	setupScenary1	Client client = new Client("13579"); Client client2 = new Client("24680"); Client client3 = new Client("54321");	3 new clients are added to the queue. The first client is removed. To verify it, we used the methods isEmpty() and isInQueue(). Also we used front() to make sure that client2 is the first one and rear() to make sure that the client3 is the last. The size is 2.
Queue	Dequeue()	setupScenary1	Client client = new Client("13579") Client client2 = new Client("24680") Client client3 = new Client("54321")	6 new clients are added to the queue. The first 3 are removed. To verify it, we used the methods isEmpty() and isInQueue(). Also we used front() to make sure that client4 is the first one and rear() to

			Client client4 = new Client("56791") Client client5 = new Client("87634") Client client6 = new Client("23678")	make sure that the client6 is the last. The size is 3.

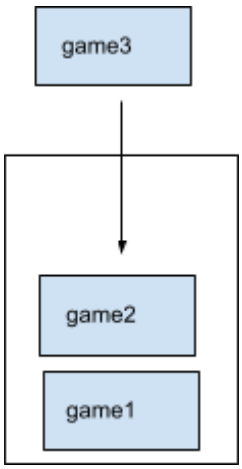
Dequeue:



Purpose of the test: Validates that the method push in the Stack works.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
Stack	Push()	setupScenary1	VideoGame game = new VideoGame(331, "TS", 5, "A", 15000)	One game is pushed to the Stack. It's verified with the methods isEmpty() and isInStack()
Stack	Push()	setupScenary1	VideoGame game = new VideoGame(612, "GTA", 3, "A", 17000) VideoGame game2 = new VideoGame(121, "ZL3", 5, "B", 12000) VideoGame game3 = new VideoGame(211, "SD2", 5, "C",	3 games are pushed to the Stack. It's verified with the methods isEmpty() and isInStack(). Also we used peek() to make sure that game3 is the first one and bottom() to make sure that game is the last. The size is 3.

			15000)	
Stack	Push()	setupScenary1	VideoGame game = new VideoGame(331, "TS", 5, "A", 15000) VideoGame testGame2 = new VideoGame(121, "ZL3", 5, "B", 12000) VideoGame testGame3 = new VideoGame(211, "SD2", 5, "C", 15000) VideoGame testGame4 = new VideoGame(678, "MN", 5, "D", 15000) VideoGame testGame5 = new VideoGame(123, "AGK", 5, "E", 15000) VideoGame testGame6 = new VideoGame(451, "TWD", 5, "F", 15000) VideoGame testGame7 = new VideoGame(301, "GR", 5, "G", 15000)	7 games are pushed to the Stack. It's verified with the methods isEmpty() and isInStack(). Also we used peek() to make sure that game7 is the first one and bottom() to make sure that game is the last. The size is 7.

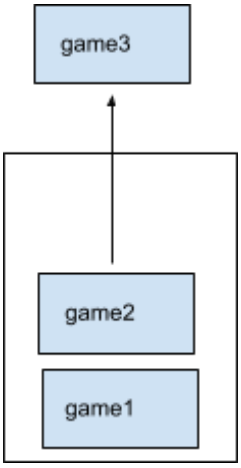
Push:



Purpose of the test: Validates that the method pop in the Stack works.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
Stack	Pop()	setupScenary1	VideoGame game = new VideoGame(331, "TS", 5, "A", 15000)	One game is pushed to the Stack and then it's removed. It's verified with the methods isEmpty() and isInStack()
Stack	Pop()	setupScenary1	VideoGame game = new VideoGame(612, "GTA", 3, "A", 17000) VideoGame game2 = new VideoGame(121, "ZL3", 5, "B", 12000) VideoGame game3 = new VideoGame(211, "SD2", 5, "C", 15000)	3 games are pushed to the Stack. The last game (game3) is removed. It's verified with the methods isEmpty() and isInStack(). Also we used peek() to make sure that game2 is the first one and bottom() to make sure that game is the last. The size is 2.
Stack	Pop()	setupScenary1	VideoGame game = new VideoGame(331, "TS", 5, "A", 15000) VideoGame game2 = new VideoGame(121, "ZL3", 5, "B", 12000) VideoGame game3 = new VideoGame(211, "SD2", 5, "C", 15000) VideoGame game4 = new VideoGame(678, "MN", 5, "D", 15000) VideoGame game5 = new VideoGame(123,	7 games are pushed to the Stack. Then, we removed 4 of them. It's verified with the methods isEmpty() and isInStack(). Also we used peek() to make sure that game3 is the first one and bottom() to make sure that game is the last. The size is 3.

			"AGK", 5, "E", 15000) VideoGame game6 = new VideoGame(451, "TWD", 5, "F", 15000) VideoGame game7 = new VideoGame(301, "GR", 5, "G", 15000)	

Pop:



Purpose of the test: Validates that the method add in the LinkedListworks.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
LinkedList	Add()	setupScenary1	Shelf shelf = new Shelf("A")	One shelf is added to the linked list and. It's verified with the methods isEmpty() and isInList()
LinkedList	Add()	setupScenary1	Shelf shelf = new Shelf("A") Shelf shelf2 = new Shelf("B") Shelf shelf3 = new Shelf("C")	3 shelves are added to the list. It's verified with the methods isEmpty() and isInList(). Also we used getFirst() to make sure that shelf is the first one and getLast() to make sure that

				shelf3 is the last. The size is 3.
LinkedList	Add()	setupScenary1	Shelf shelf = new Shelf("A") Shelf shelf2 = new Shelf("B") Shelf shelf3 = new Shelf("C") Shelf shelf4 = new Shelf("D") Shelf shelf5 = new Shelf("E") Shelf shelf6 = new Shelf("F")	6 shelves are added to the list. It's verified with the methods isEmpty() and isInList(). Also we used getFirst() to make sure that shelf is the first one and getLast() to make sure that shelf6 is the last. The size is 6.

Purpose of the test: Validates that the method remove in the LinkedListworks.				
Class	**Method**	**Scenes**	**Inputs**	**Results**
LinkedList	Remove()	setupScenary1	Shelf shelf = new Shelf("A") Shelf shelf2 = new Shelf("B") Shelf shelf3 = new Shelf("C")	3 shelves are added to the list. The first shelf (shelf) is removed. It's verified with the methods isEmpty() and isInList(). Also we used getFirst() to make sure that shelf2 is the first one and getLast() to make sure that shelf3 is the last. The size is 2.
LinkedList	Remove()	setupScenary1	Shelf shelf = new Shelf("A") Shelf shelf2 = new Shelf("B") Shelf shelf3 = new Shelf("C")	6 shelves are added to the list. Then, we removed 3 of them. It's verified with the methods isEmpty() and isInLis(). Also we used getFirst()

			Shelf shelf4 = new Shelf("D") Shelf shelf5 = new Shelf("E") Shelf shelf6 = new Shelf("F")	to make sure that shelf is the first one and getLast() to make sure that game3 is the last. The size is 3.