Engineering Method - Library

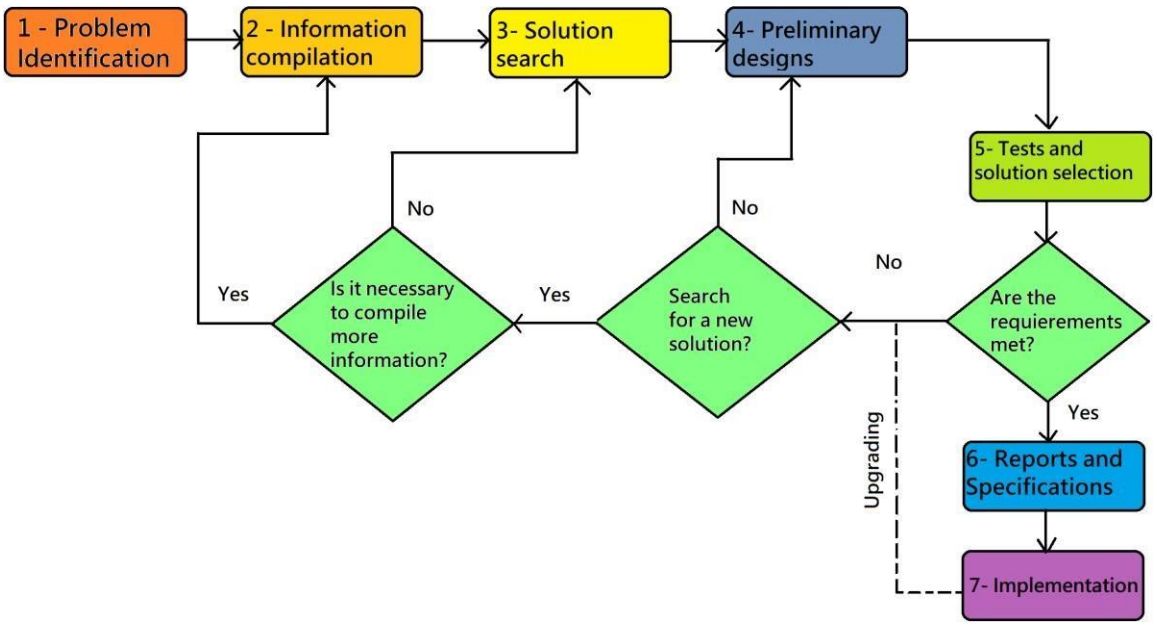
Problem Context:

The company in question, a very important bookstore in the city of Cali that has been willing to open its doors in Sultana del Valle, is dedicated to the sale of books of multiple genres and different languages, and is characterized by a very particular, innovative, and efficient style when it comes to serving its users. This library requires a software solution that allows people of Cali to know a little bit about how their new attraction would work in the city.

Development of the Solution:

To resolve the above situation, the Engineering Method was chosen to develop the solution following a systematic approach and in line with the problematic situation established.

Based on the description of Paul Wright's “Introduction to Engineering”, the following flowchart was defined, the steps of which we will follow in the development of the solution.



Step 1. Problem Identification

The specific needs of the problematic situation are recognized as well as their symptoms and conditions under which it must be resolved.

Identification of needs and symptoms:

- Users of the library's services require purchasing books of multiple genres and different languages.
- There is no program in the library that simulates the process of buying your books.
- The solution to the problem must ensure complete accuracy in such simulation.
- The solution to the problem must be efficient so that it can serve as many users as possible with minimal resource consumption.

Problem Definition:

The library requires the development of a software application that allows you to simulate the process of purchasing your books.

Step 2. Information Collection

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 established. It is important to perform this search with recognized and reliable sources to know which elements are part of the problem and which are not.

Definitions

- Isbn:
  International book numbering system for easy and correct identification. Identification number of a book accordingto the ISBN.

- Algorithmic Analysis:
  Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to [1].

- Asymptotic notation:
  The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N '0, 1, 2, ...'. Such notations are convenient for describing the worst-case running-time function T (n), which usually is defined only on integer input sizes. Frequently, asymptotic notation is used to describe the running times of algorithms.

- Θ-notation:

The **Θ**-notation asymptotically bounds a function from above and below. For a given function g(n), we denote by **Θ**(g(n)) the set of functions:
Θ(g(n)) = {f(n): there exist positive constants c_1, c_2 and n_0
such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$}

- O-notation:
  The O-notation provides an asymptotic upper bound function. For a given function g(n), we denote by O(g(n)) the set of functions:
  O(g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$}
- Ω-notation:
  The Ω-notation provides an asymptotic lower bound function. For a given function g(n), we denote by O(g(n)) the set of functions:
  Ω(g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$} [1]

Step 3. Finding Creative Solutions

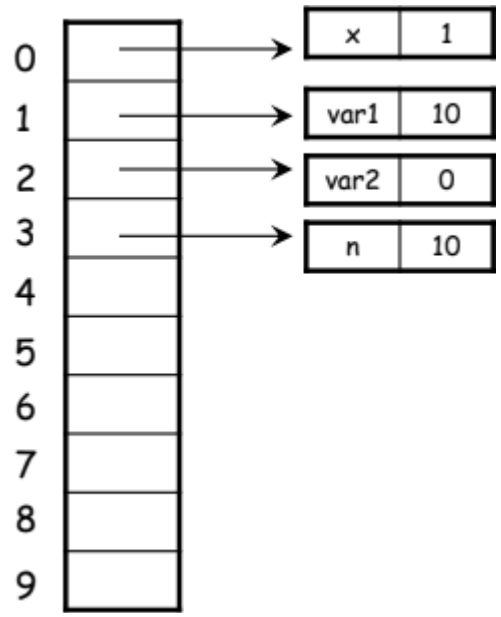
For this step, even if we can think of our own solutions, we look at specialized texts various ways to implement the different abstract data structures/types (ADTs) necessary and that best suit the different cases of each stage of the process of buying books in general. The methods considered as alternatives are as follows:

<u>Alternative 1 - Use Hash Table, Stack, and Queue:</u>

The hash table would fit the bookshelves, the stack would fit the basket with the books chosen by each customer, and the queue would fit the pay row at the cashiers.

The ADT Hash Table:[2]
The ADT hash table is a data structure that associates keys with values. The main operation that it efficiently supports is search:

- Allows access to stored items from a generated key.
- It works by transforming the key with a hash function into a hash, a number that the hash table uses to locate the desired value.



Access Procedures:

Constructor operations to construct a Hash Table:

i. createTable() → creates an empty table.
ii. tableInsert(newItem) → inserts newItem into a table in its proper sorted order according to the item's search key.

Predicate operations to test Hash Tables:

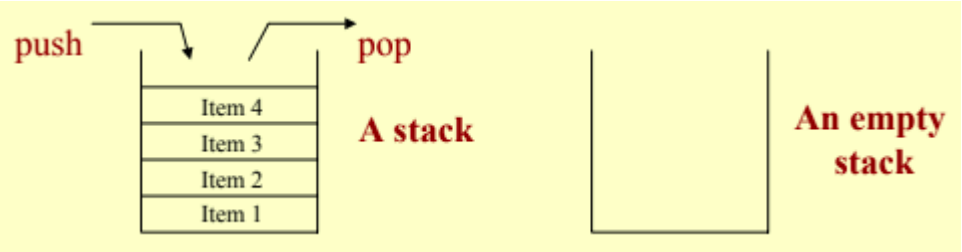
i. isEmpty( ) → determines whether a table is empty.

Selector operations to select items of a Hash Table:

i. tableLength( ) → determines the number of items in the table.
ii. tableDelete(searchKey) → deletes an item with a given search key from the table.
iii. tableRetrieve(searchKey) → retrieves an item with a given search key from a table.

The ADT Stack: [3]

The ADT Stack is a linear sequence of an arbitrary number of items, together with access procedures. The access procedures permit insertions and deletion of items only at one end of the sequence (the "top"). The stack is a list structure, sometimes called a last-in-first-out (or LIFO) list. A stack is either empty, or it consists of a sequence of items. Access is limited to the "top" item on the stack at all times.



Access Procedures:

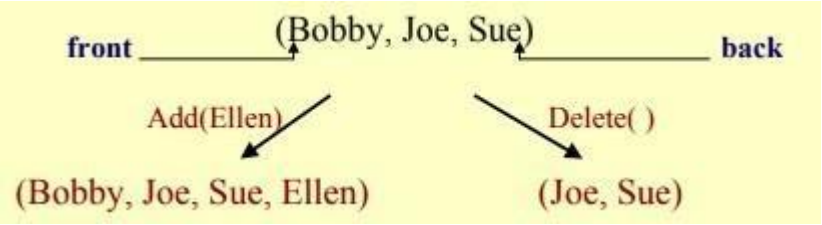
Operations to construct a stack constructor:
i. createStack( ) → creates an empty Stack.
ii. push(item) → adds an item at the top of the stack.

Preach operations to test Stacks:
i. isEmpty( ) → determine whether a stack is empty.

Selector operations to select items of a Stack:
i. top( ) → returns the top item of the stack. It does not change the stack.
ii. pop( ) → changes the stack by removing the top item.

The ADT Queue:

The ADT Queue is a linear sequence of an arbitrary number of items, together with access procedures. The access procedures permit addition only at the back of the queue and deletion of items only at the front of the queue. The queue is a list structure sometimes called a first-in-first-out (or FIFO) list. A queue is either empty, or it consists of a sequence of items. Manipulations or accesses to these items are only permitted at the two ends of the queue.



Access Procedures:

Constructor operations to construct a queue:
i. createQueue( ) → creates an empty queue.
ii. enqueue(item) → adds an item at the end of the queue.

Preach operations to test Queues:

i. isEmpty( ) → determine whether a queue is empty.

Selector operations to select items of a queue:
i. front( ) → returns the item at the front of the queue. It does not change the queue.
ii. dequeue( ) → retrieves and removes the item at the front of the queue.

Alternative 2 - Use Fix, List, Matrix:

The arrangement would be adapted to the bookshelves, the list to the basket with the books chosen by each customer and the matrix to the payment row at the ATMs.

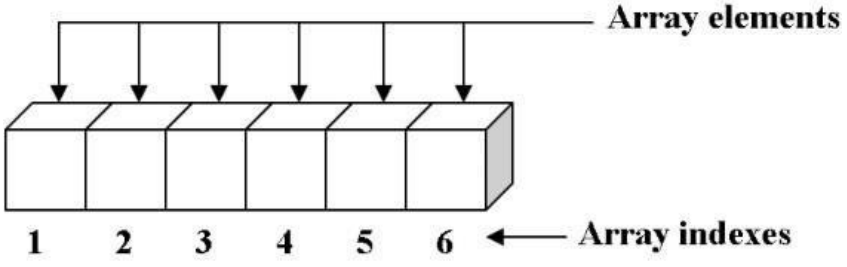
The ADT Array: [4]

The array is a basic abstract data typethat holds an ordered collection of items accessible by an integer index. These items can be anything from primitive types such as integers to more complex types like instances of classes. Since it's an ADT, it doesn't specify an implementation, but it is almost always implemented by an array (data structure) or dynamic array.

Operations to construct Arrays constructor:
i. createArray( ) → creates an empty array.
ii. add(index, item) → inserts item at position index of an array if 1 ≤ index ≤ size() + 1. If index ≤ size(), items at position index onwards are shifted one position to the right.

Preach operations to test Arrays:
i. isEmpty( ) → determine if an array is empty.

Selector operations to select items of a Array:
i. remove(index) → removes item at position index of an array if 1 ≤ index ≤ size(). Items at position index + 1 onwards are shifted one position to the left.
ii. get(index) → returns item at position index of an array if 1 ≤ index ≤ size().
iii. set(index, newValue) → sets the value of an specified index to a new value.
iv. length( ) → returns number of items in an array.



One-dimensional array with six elements

The arrangement would be adapted to the bookshelves, the list to the basket with the books chosen by each customer and the matrix to the payment row at the ATMs.

The ADT List:

The ADT List is a linear sequence of an arbitrary number of items that must be of the same type, together with the following access procedures that can be grouped into three broad categories:

Operations to construct Arrays constructor:
i. createList( ) → creates an empty list.
ii. add(index, item) → inserts item at position index of a list if 1 ≤ index ≤ size() + 1. If index ≤ size(), items at position index onwards are shifted one position to the right.

Preach operations to test Arrays:
i. isEmpty( ) → determine if an array is empty.

Selector operations to select items of a Array:
i. remove(index) → removes item at position index of a list if 1 ≤ index ≤ size(). Items at position index + 1 onwards are shifted one position to the left.
ii. get(index) → returns item at position index of a list if 1 ≤ index ≤ size().
iii. set(index, newValue) → sets the value of an specified index to a new value.
iv. size( ) → returns number of items in a list.

Alternative 3 - Use Binary Search Trees: [5]

The binary search tree would suit both the bookshelves and the baskets with the books chosen by each customer, as well as for the payment row at the ATMs.

A Binary Search Tree (BST) is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

All the nodes follow the below mentioned properties:

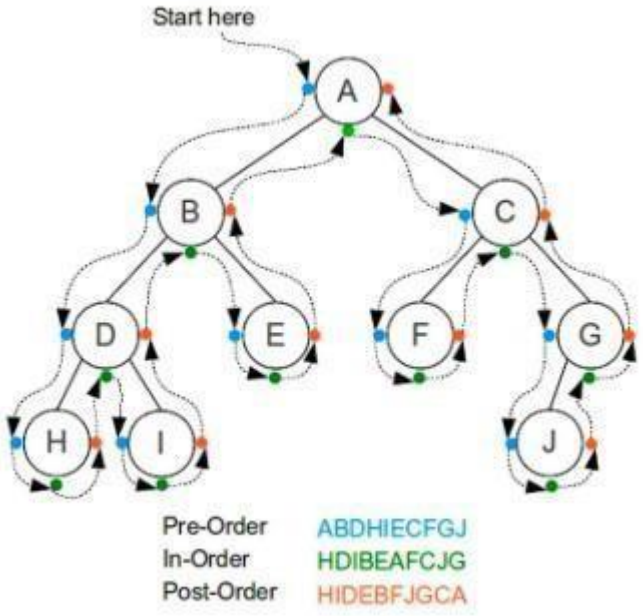
- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments.

Operations to construct BST constructor:
i. createTree( ) → creates an empty tree.
ii. insert(key, value) → inserts value with a key in a BST following the mentioned properties previously.

Preach operations to test BST:
i. isEmpty( ) → determine if a BST is empty by checking the existence of its root.

Selector operations to select items of a BST:
i. remove(index) → removes an element from the BST through its value, adjusting then its elements according to the mentioned properties previously.
ii. search(value) → searches an element in a BST through its value.
iii. preOrderTraversal() → traverses a tree in a pre-order manner.
iv. inOrderTraversal() → traverses a tree in an in-order manner.
v. postOrderTraversal() → traverses a tree in a post-order manner.



[6]

Step 4. Transitioning Ideas to Preliminary Designs

The first thing we do in this step is to dismiss ideas that are not feasible. In this sense we ruled *out Alternative 3 - Use Binary Search Trees* because the implementation in this case of a binary tree would lead it to behave specifically as a linear structure, thus wasting the potential of this data structure.

Careful review of the other alternatives leads to the following:

Alternative 1 - Use Hash Table, Stack and Tail

- ➢ The structures mentioned have a similar behavior to the elements to be modeled in the simulation. The stack with its LIFO (Last In, First Out) behavior is similar to the behavior of baskets; the queue, being FIFO (First In, First Out),is similar to the behavior of the payment queues and the HashTable, allowing to save any element given a key, would serve to simulate the shelves that store copies of books knowing their ISBN.

Alternative 2 - Use Array, List, Matrix:

- ➢ The above-mentioned structures would allow the implementation of the simulation, but its behavior is not approximate to the elements that are to be modeled, which leads to the need for more time to implement the methods necessary for the structures to behave as desired.

Step 5. Evaluation and Selection of the Best Solution

Criteria

The criteria for evaluating solution alternatives should be defined and based on this outcome, the solution that best meets the needs of the problem stated is chosen. The criteria we chose in this case are the ones listed below. A numerical value has been established next to each one in order to establish a weight indicating which of the possible values of each criterion have the most weight (i.e., they are mostdesirable).

Criterion A. Spatial complexity. The most efficient solution would be prefered.
Efficiency can be:

- [1] O(n!)
- [2] O(2))
- [3] O(n-2)
- [4] O(n log n)
- [5] O(n)
- [6] O(log n), O(1)

Criterion B. Implementation behavior. A solution that behaves similarly to the elements of the model is preferred. The behavior can be:

- [3] Similar
- [2] Partially similar
- [1] Not similar

Evaluation

By evaluating the above criteria in the alternatives that are maintained, we get the following table:

	Criterion A	**Criterion B**	**Total**
Alternative 1	5 O(n)	3 Similar	8
Alternative 2	5 O(n)	2 Partially similar	7

Selection

According to the previous evaluation Alternative 1 should be selected, as it obtained the highest score according to the defined criteria.

Step 6. Preparation of Reports and Specifications

Problem Specification:

Problem: Simulate the purchase of books in the library.

Input: The number of cashiers available during the day, the number of shelves with their identifier and an L number of books on that shelf. Then the data of the L books on the shelf (ISBN, price and number of copies). Subsequently, the necessary information of the customers who enter the bookstore during the day (customer ID and the ISBNs of the books to be purchased).

Output: The results of testing the different (4) sections are presented: 1: Customers with the ISBNs of the books they want to buy; 2: customers with the ISBNs of the books they want to buy, organized in order of proximity to the shelf; 3:Customers with the book baskets in the order in which they picked them up and 4: Customers with the price they paid and the books packed.

Considerations
The following cases should be taken into account for testing this process:

1. Customers who do not purchase a book may be submitted.
2. Customers who search for books that are not available (do not exist or are out of stock) may be presented.
3. Simulation of sections have an asynchronous, non-concurrent time passage. Time is measured in discrete, non-continuous units; and each step takes a unit of time.
4. In the fourth section, when paying customers pass from a single row, so it can be said that they are all served on the same first come, first served basis. The order of departure can vary, as this will depend on how long each one's attention takes at the cashier (number of items to be purchased).

Functional requirements

The system must be able to:

FR1: Enable the simulation of the library by receiving the necessary parameters. Through the graphical interface, the following parameters are entered: number of shelves, the book catalog (BOOK ISBN, number of copies, shelf where it is located, price), the number of cashiers to be used during the day, the series of codes or cards representing customers (in the order in which they entered the store) and the list of books per buyer (ISBN codes). As a result, the library data is set with the given parameters.

FR2: Indicate the blocks or shelves where each customer should look for the volume of interest. For this, the system first sorts each received list of ISBN codes according to the location of the shelves, so that the buyer follows the best route; the client specifies the sorting algorithm to use (if the book is out of stock, its ISBN will not appear in this final sorted list). When a customer's list of codes is sorted, the number of available copies of that book decreases.

FR3: Keep track of the time it takes for each customer in sections 2 and 3, (ordering the list of codes for each customer and having collected the books of interest in each shelf). Picking up a book and sorting a customer's code list takes a unit of time.

FR4: Simulate the basket. The program keeps track of the order of the books that each customer finds and stores in their respective basket, one on top of the other, following the order provided in section 2.

FR5: Order customers according to the time registered in the **FR3** at the time of entering section 4, in ascending order (the one that has taken the least time in sections 2 and 3 will be the first in the payment queue).

FR6: Simulate the attention process. When one or more cashiers are available, they attend as many customers as there are available cashiers. Since customers pass from the single queue, it is affirmed that every customer is served in that same arrival order. The order of departure does vary, since this depends on the time that each customer's attention takes (according to the number of books to be purchased).

FR7: Calculate the total payment of each customer. For this, the prices of the books entered at the cash register are added up and the total cost is presented to the customer. The packing order of the books is obtained by inverting the order in which the customer had them in their basket.

FR8: Inform in detail about customers when leaving. A report is presented on the order of departure of customers, the order in which their books are packed and how much they paid for them.

ADT DESIGN

TAD Generic Stack			
Stack = { e1, e2, … , en } ei = element ∈ Stack Top = en ∈ Stack T(e_i) = type of ei Size = |Stack|			
{ inv: ∀ ei , ej ∈ Stack ⟹ (T(e_i) = T(e_j) | i ≠ j) Size ≥ 0 Size = 0 ⟹ ∄ Top ⋀ Stack = ∅ Top ⟹ accessibility ⋀ manipulation }			
Primitive operations			**Operation type**
▪ CreateStack		→ Stack	Constructor
▪ PushElement	Element T x Top x Stack	→ Stack	Modifier
▪ PopElement	Element T x Top x Stack	→ Top	Modifier
▪ PeekTop	Element T x Top x Stack	→ Top	Analyzer
▪ IsEmpty	Stack	→ Boolean	Analyzer

CreateStack()

"Creates an empty Stack"

{ pre: *TRUE* }

{ post: *Stack* = ∅ }

PushElement(element, top, Stack)

"Adds a new element at the top of the Stack"

{ pre: ∃ Stack, element ∉ Stack }

{ post: |Stack| + 1, element ∈ Stack, top = element }

PopElement(element, top, Stack)

"Removes and retrieves the element on the top of the Stack"

{ pre: ∃ Stack, Stack ≠ ∅, top = element ∈ Stack }

{ post: element, |Stack| - 1, element ∉ Stack, top = $element_{n-1}$}

PeekElement(element, top, Stack)

"Retrieves the element on the top of the Stack"

{ pre: ∃ Stack, Stack ≠ ∅, top = element ∈ Stack }

{ post: top }

IsEmpty(Stack)

"Determines whether the Stack is empty"

{ pre: ∃ Stack }

{ post: True if Stack = ∅, otherwise False }

TAD Generic Queue

Queue = { e_1, e_2, … , e_n }
e_i = element ∈ *Queue*
Front = e_1 ∈ *Queue*
Back = e_n ∈ *Queue*
T(e_j) = type of e_j
Size = |*Queue*|

{ inv: ∀ e_i, e_j ∈ *Queue* ⟹ (T(e_i) = T(e_j) | i ≠ j)

Size ≥ 0

Size = 0 ⟹ ∄ Front ⋀ ∄ Back ⋀ Queue = ∅ Front

⋀ Back ⟹ accessibility ⋀ manipulation

+: Back

-: Front }

Primitive operations				Operation type
▪ CreateQueue		→	Queue	Constructor
▪ EnqueueElement	Element T x Back x Queue	→	Queue	Modifier
▪ DequeueElement	Element T x Front x Queue	→	Front	Modifier
▪ FrontElement	Element T x Front x Queue	→	Front	Analyzer
▪ IsEmpty	Queue	→	Boolean	Analyzer

CreateQueue()

"Creates an empty Queue"

{ pre: *TRUE* }

{ post: *Queue* = ∅ }

EnqueueElement(element, back, Queue)

"Adds a new element at the back of the Queue"

{ pre: ∃ Queue, element ∉ Queue }

{ post: |Queue| + 1, element ∈ Queue, back = element }

DequeueElement(element, front, Queue)

"Removes and retrieves the element at the front of the Queue"

{ pre: ∃ Queue, Queue ≠ ∅, front = element ∈ Queue }

{ post: element, |Queue| - 1, element ∉ Queue, front = $element_{i+1}$ }

FrontElement(element, front, Queue)

"Retrieves the element at the front of the Queue"

{ pre: ∃ Queue, Queue ≠ ∅, front = element ∈ Queue }

{ post: front }

IsEmpty(Queue)

"Determines whether the Queue is empty"

{ pre: ∃ Queue }

{ post: True if Queue = ∅, otherwise False }

TAD Generic Hash Table

HashTable = { (i_1(k_1 ; v_1)), (i_2(k_2 ; v_2)), … , (i_n(k_n ; v_n)) }

h : hash function related to *HashTable*

k_i = key hashed at i_i

i_i = h(k_i) = index of a subset ∈ *HashTable*

v_i = associated value to k_i

T(k_i) = type of k_i

T(v_i) = type of v_i

Length = |*HashTable*|

{ inv: ∀ k_i, k_j ∈ *HashTable* ⟹ (T(k_i) = T(k_j) | i ≠ j)

∀ v_i, v_j ∈ *HashTable* ⟹ (T(v_i) = T(v_j) | i ≠ j)

∃! k_i ∈ *HashTable*

Length ≥ 0

Length = 0 ⟹ *HashTable* = ∅

h(k_i) ⟹ i_i ∈ $\mathbb{Z}^+$ ∧ 0}

Primitive operations	Operation type
▪ CreateTable → HashTable ▪ TableInsert Element(Key x Value) x index x h x HashTable → HashTable ▪ TableDelete Key x index x h x HashTable → HashTable ▪ TableRetrieve Key x index x h x HashTable → Value T ▪ TableLength HashTable → Integer ▪ IsEmpty HashTable → Boolean	Constructor Modifier Modifier Analyzer Analyzer

CreateTable()

"Creates an empty Hash Table"

{ pre: *TRUE* }

{ post: HashTable = ∅ }

TableInsert(element(key, value), index, hash, HashTable)

"Inserts a new element with key and value at an index of the Hash Table in its proper sorted order (hash) according to the element's search key"

{ pre: ∃ HashTable, element(key, value) ∉ HashTable, index(element) = hash(key) }

{ post: |HashTable| + 1, element(key, value) ∈ HashTable, HashTable[index] = element(key, value) }

TableDelete(key, index, hash, HashTable)

"Deletes the value of an element with a given search key from the Hash Table"

{ pre: ∃ HashTable, HashTable ≠ ∅, element(key, value) ∈ HashTable, index(element) = hash(key) }

{ post: |HashTable| - 1, element(key, value) ∉ HashTable, HashTable[index] = NIL

TableRetrieve(key, index, hash, HashTable)

"Searches and retrieves the value of an element with a given search key from the Hash Table"

{ pre: ∃ HashTable, HashTable ≠ ∅, element(key, value) ∈ HashTable, index(element) = hash(key) }

{ post: value ∈ element) }

TableLength(HashTable)

"Determines the number of elements in the table"

{ pre: ∃ HashTable }

{ post: |HashTable| ∈ Integer}

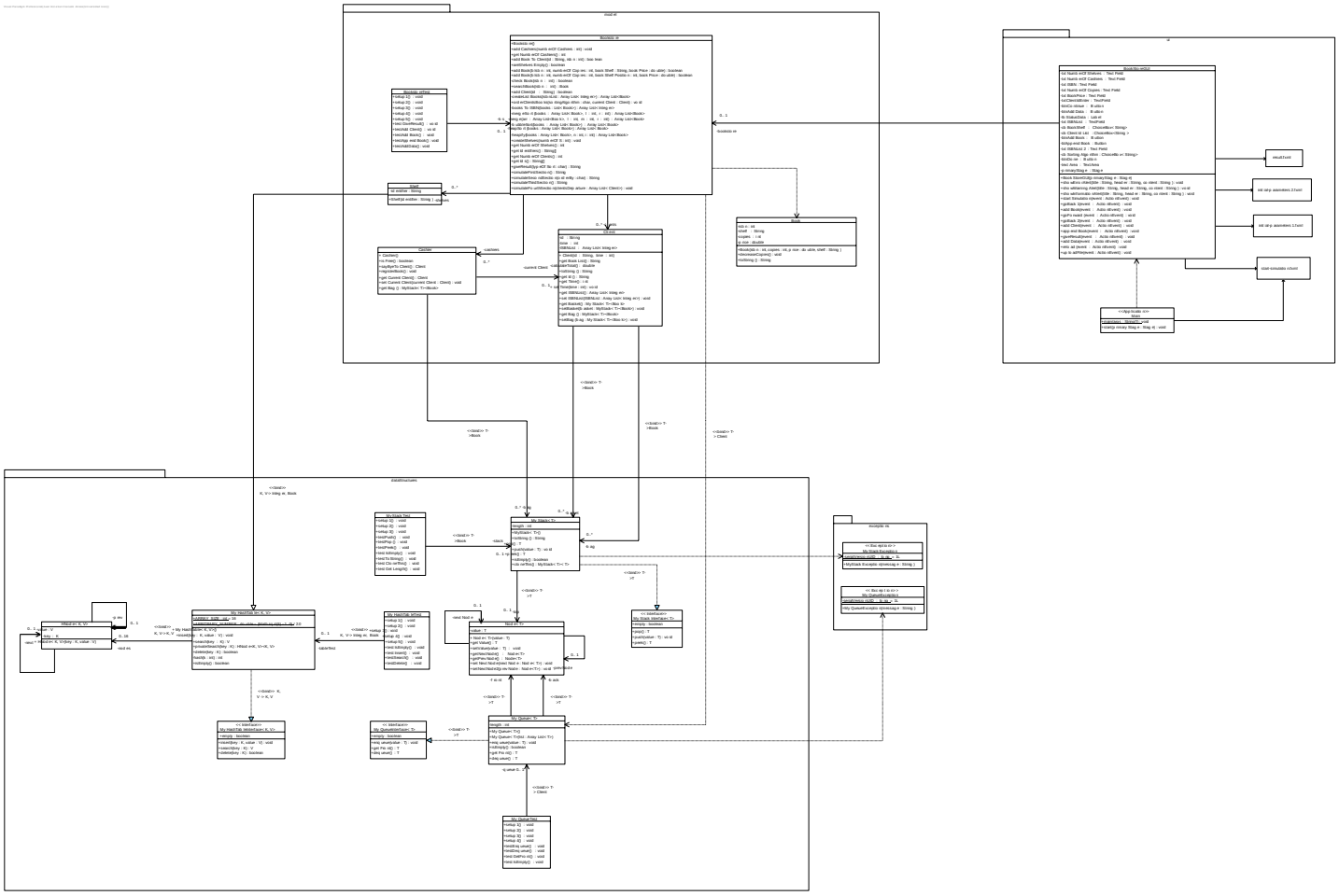
IsEmpty(HashTable)

"Determines whether the Hash Table is empty"

{ pre: ∃ HashTable }

{ post: True if HashTable = ∅, otherwise False }

Class diagram


mod ui
Bookstore ui
Bookstore mQ
+add Cashier(cashId: int)Cashiers : int): void
+add Book To Shelf(): String, isbn: int): bool
+add Book To Client(id : String, isbn: int): bool
+setAllShelves Empty(): boolean
+add Book(b:Isb n : int, numS:int)Copies : int, bookShelf : String, bookPrice : db.utils): boolean
+add Book(b:Isb n : int, numS:int)Copies : int, bookShelf:Positio n : int, bookPrice : db.utils): boolean
+check Book(isbn n : int): boolean
+search Book(isbn n : int): Book
+check Book(isbn n : int): boolean
+delete Book(isbn:Isbn): Array Lstn: String arr): Array Lstn<Book>
+sort mCheckBook by(s:String): char, current Client : Client): vo id
+books To SB(Books : Lstn Book>): Array Lstn< String arr>
+reg info (t:Books : Array Lstn Book>, i : int, r : int) : Array Lstn<Book>
+search Book(isbn:Isbn): Array Lstn: String arr): Array Lstn<Book>
+Available(Books : Array Lstn Book>, r : Array Lstn<Book>
+by Pr (t:Books : Array Lstn Book>): Array Lstn<Book>
+map(Books : Array Lstn: Book>, n : int, i : int): Array Lstn<Book>
+create Shelves(numS:int)Of S: int): void
+create Shelf(shelfId : String): nil
+get id: int(String): String
+get Numb mOf Client(s) : int
+get id: s): String()
+get Result(typ:mOf Se d : char) : String
+simulate Fee(fee:s): String
+simulate Issu ref(books: s)): id entity : char) : String
+simulate Issu ref(books: s)): id entity : char) : String
+simulate Fu ull(books: s)):Books(typ: allusre : Array Lstn: Client): void
Bookstore mQ
Bookstore mShelf
+setup 1(): void
+setup 2(): void
+setup 3(): void
+setup 4(): void
+setup 5(): void
+test Client Result(): int id
+test Add Client(): vo id
+test Add Book(): void
+test App end Book(): void
+test Add Cli ent(): void
Shelf
+id: int(the : String
+Shelf(id: int)the : String): void
Cashier
- Cashier()
+is Free(): boolean
+req Myself's Client(): Client
+req Info Book(): void
+get Current Client(): Client
+set Current Client(current Client : Client): void
+get Reg 1: MyStack< T>>Books>
Client
- id : String
- Wallet : int
- Books : Array Lstn: String arr>
+ Client(id : String, time : int)
+get Book List(): String
+get Total(): double
+ toString (): String
+get Id(): String
+get Time(): String
+set Time(time : int): void
+get Wallet(s): Array Lstn: String arr>
+set Wallet(s)(Wallet : Array Lstn: String arr): void
+get Books(): My Stack< T>>Bks b>
+ update(s): stdnt: MyStack< T>>Books>: void
+ add Book(b : My Stack< T>>Bks b>): void
+ toString (): My Stack< T>>Bks b>): void
Book
isbn: int
shelf : String
price : int
is sold : double
+ Book(isbn n : int, copies : int, p rice : db.utils, shelf : String)
+ toString (): String
ui
BookStu ui(Gui)
txt Numb mOf Students : Text Field
txt Numb mOf Cashiers : Text Field
txt ISBN : Text Field
txt Numb mOf Copies : Text Field
txt BookPrice : Text Field
txt ClientStdEmail : TextField
txt Client Name : TextField
btnAdd Data : Button
lb StatusData : Lab el
txt ISBNtoU : TextField
cb BookShelf : ChooseBox< String>
cb Book Name : ChooseBox<String >
btnAdd Book : Button
btnApp end Book : Button
txt ISBNtoU 2 : Text Field
cb Sorting Algo ritihm : ChooseBo x< String>
btnStr ow : Butto n
text Area : TextArea
btnGetReg Fe : Button
+Book Store(GUI):void(String a : String a)
+add Book(isbn:Isbn, copies : int, price : String, ss shelf : String): void
+chs withsorting Alet(title : String, head er : String, ss stand : String): void
+chs withformats shelf(title : String, head er : String, ss stand : String): void
+load Simulate r(event : Actio nEvent): void
+getBack 2(event : Actio nEvent): void
+get Book Name(event : Actio nEvent): void
+getTr ransit Event(: Actio nEvent): void
+getBack 2(event : Actio nEvent): void
+add Client(event : Actio nEvent): void
+app end Book(event : Actio nEvent): void
+get Book Name(event : Actio nEvent): void
+add Data(event : Actio nEvent): void
+chs ad (event : Actio nEvent): void
+up to act#(event : Actio nEvent): void
result Fund
btl ary assmblers 2.fuel
btl ary assmblers 1.fuel
start simulatio n.fund
<<App Guitto >>
Main
+Main(args : String[]): void
+ start(p rimary Stag e : Stag e): void
dataStructures
<<class>>
K, V> bting st, Book
My Stack Test
+setup 1(): void
+setup 2(): void
+setup 3(): void
+setup 4(): void
+testPop(): void
+testPeek(): void
+test IsEmpty(): void
+test To String(): void
+test Clo ne():void : void
+test Get Length(): void
My Stack< T>
-length : int
+MyStack< T>()
+ toString (): String
+ clear(): void
+ push(value : T): vo id
+ isEmpty(): boolean
+ pop(): T
+ peek(): T
+ size(): int
+ clone(): MyStack< T>
<<class>>
>T
<<class>>
>Stack
<<class>>
>Stack
<<class>>
>Stack
<<class>>
> Client
Invisible ds
<< this api is o >>
My Stack Exceptio n
+MyStack Exceptio n(mesag e : String)
<< this api i s in >>
My Queue Exceptio n
+My QueueExceptio n(mesag e : String)
My Node< K, V>
- key : K
- value : V
- next : My Node< K, value : V>
+ My Node(key : K, value : V)
+ setNext(next : My Node< K, value : V>)
+ getNext(): My Node< K, value : V>
+ getKey(): K
+ getValue(): V
+ setValue(value : V): void
g: ste
0.. 1
0.. 1
0.. 1
0.. 1
next: n
next ex
My HashTable Sc h, V>
- buckets : int[] = 10
- hashFunction(key : K): int
+ put(key : K, value : V): void
+ get(key : K): V
+ remove(key : K): void
+ containsKey(key : K): boolean
+ isEmpty(): boolean
<<class>>
K, V> bting ex, Book
<<class>>
K, V>>K, V
My HashTable Iterator
+ hasNext(): boolean
+ next(): K
<<class>>
K, V> bting ex, Book
tableNext
My Node< T>
- value : T
+ Node(ex: T)(value : T)
+ getValue(): T
+ setValue(value : T): void
+ getNext Node(): Node<T>
+ getPrev Node(): Node<T>
+ setNext Node(nextN ode : Node< T>): void
+ setPrevNode(p rev Node : Node< T>): void
next Node
prevNode
My Queue< T>
-length : int
+ My Queue< T>()
+ My Queue< T>(list : Array Lstn< T>)
+ enq ueue(value : T): void
+ isEmpty(): boolean
+ get Fro nt(): T
+ deq ueue(): T
<<class>>
>T
<<class>>
>T
<<class>>
>T
<<class>>
>T
front
rear
q: ueue
My Queue Iterator< T>
+ empty : boolean
+ next(): T
+ hasN ext(value : T): vo id
+ peek(): T
My Client Test
+setup 1(): void
+setup 2(): void
+setup 3(): void
+setup 4(): void
+testEnq ueue(): void
+test Deq ueue(): void
+test FindFro nt(): void
+test IsEmpty(): void
<<class>>
> Client

Design of the scenarios and test cases

Scenario Configuration

Name	Class	Scenario
setup1	MyStackTest	A stack of type Book is initialized as MyStack<Book>.
setup2	MyStackTest	A stack of type Book is initialized as MyStack<Book> and an object from the Book class is created and added to this stack, with isbn = 333, copies = 2, price = 20000 and shelf .
setup3	MyStackTest	A stack of type Book is initialized as MyStack<Book> and two objects from the Book class are created and added to this stack: The first Book has: isbn = 333, copies = 2, price = 20000 and shelf = "A". The second Book has: isbn = 111, copies = 6, price = 30000 and shelf = "A".
setup1	MyQueueTest	A queue of type Client is initialized as MyQueue<Client>.
setup2	MyQueueTest	A queue of type Client is initialized as MyQueue<Client> and an object from the Client class is created and added to this queue, with id = "111", time = 3.
setup3	MyQueueTest	A queue of type Client is initialized as MyQueue<Client> and two objects from the Client class are created and added to this queue: The first Client has: id = "111", time = 3. The second Client has: id = "333", time = 4.
setup4	MyQueueTest	A queue of type Client is initialized as MyQueue<Client> and three objects from the Client class are created and added to this queue: The first Client has: id = "111", time = 3. The second Client has: id = "333", time = 4. The third Client has: id = "555", time = 5.
setup1	MyHashTableTest	A hash table of key type Integer and value type Book is initialized as MyHashTable<Integer, Book>.
setup2	MyHashTableTest	A hash table of key type Integer and value type Book is initialized as MyHashTable<Integer, Book> and an object from the Book class is created and added to this table, with isbn = 441, copies = 3, price = 20000 and shelf = "A".
setup3	MyHashTableTest	A hash table of key type Integer and value type Book is initialized as MyHashTable<Integer, Book> and two objects from the Book class are created and added to this table (no collision) :

		The first Book has: isbn = 441, copies = 3, price = 20000 and shelf = "A". The second Book has: isbn = 159, copies = 10, price = 100000 and shelf = "A".
setup4	MyHashTableTest	A hash table of key type Integer and value type Book is initialized as MyHashTable<Integer, Book> and two objects from the Book class are created and added to this table (collision) : The first Book has: isbn = 441, copies = 3, price = 20000 and shelf = "A". The second Book has: isbn = 229, copies = 5, price, 30000 and shelf , "A".
setup5	MyHashTableTest	A hash table of key type Integer and value type Book is initialized as MyHashTable<Integer, Book> and three objects from the Book class are created and added to this table (one collision) : The first Book has: isbn = 441, copies = 3, price = 20000 and shelf = "A". The second Book has: isbn = 159, copies = 10, price = 100000 and shelf = "A". The third Book has: isbn = 229, copies = 5, price, 30000 and shelf , "A".
setup1	BookstoreTest	An object of the Bookstore class is initialized and 3 cashiers, 3 shelves, 11 books and 5 clients are added to the system: 2 books to a first client. 4 books to a second client. 1 book to a third client. 4 books to a fourth client. 1 book to a fifth client.
setup2	BookstoreTest	An object of the Bookstore class is initialized.
setup3	BookstoreTest	An object of the Bookstore class is initialized and 3 cashiers and 3 shelves are added to the system.
setup4	BookstoreTest	An object of the Bookstore class is initialized and 1 client is added.
setup5	BookstoreTest	An object of the Bookstore class is initialized and 3 cashiers, 3 shelves, 1 book, and 1 client are added to the system.

Test cases design

Test Objective: Verify that a new item is successfully added at the top of an item stack. In this case it is a stack of books.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Book>	Push	setup1	Book type object with: isbn = 333 copies = 2 price = 20000 shelf = "A"	A new Book item was successfully added to a book stack. This stack is now size equal to 1, it is false that it is empty, and the book at its top is from ISBN equal to 333.
MyStack <Book>	Push	setup2	Book type object with: isbn = 111 copies = 6 price = 30000 shelf = "A"	A new Book item was successfully added to the workbook stack. This stack now has size equal to 2, and the book at its top is ISBN equal to 111.
MyStack <Book>	Push	setup3	Book type object with: isbn = 555 copies = 8 price = 50000 shelf = "A"	A new Book item was successfully added to the workbook stack. This stack now has size equal to 3, and the book at its top is ISBN equal to 555.

Explanation of the third case in this push test (setup3):

We have firstly a stack with 2 books. The first one has isbn = 333, the second one with isbn = 111. This last one is at the top of the stack. So, we begin the pushing process of a new book with isbn = 555. In this case, we just push it above the book with isbn = 111. So, the book with isbn = 555 is now at the top of the stack.

Test Objective: Verify that the item is successfully deleted and returned from the top of an item stack. In this case it is a stack of books.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Book>	Pop	setup1	None	The MyStackException comes out by communicating the error of performing the delete a workbook action on an empty workbook stack.
MyStack <Book>	Pop	setup2	None	Book object with: isbn = 333 copies = 2 price = 20000 shelf = "A"

				A Book-type item at the top of the book stack was successfully deleted and returned. This stack is now size equal to 0, empty, and therefore no book is at its

				top.
MyStack <Book>	Pop	setup3	None	Book object with: isbn = 111 copies = 6 price = 30000 shelf = "A" A Book-type item at the top of the book stack was successfully deleted and returned. This stack is now size equal to 1, it is still not empty, and at its top is now an ISBN book equal to 333.

Explanation of the third case in this pop test (setup3):

We have firstly a stack with 2 books. The first one has isbn = 333, and the second one with isbn = 111 (at the top). So, we begin the poping process of a book. In this case, the books can just be poped by the top. So, we just pop the book with isbn = 111 and therefore, the book with isbn = 333 is now at the top of the stack.

Test Objective: Verify that the item is returned correctly at the top of an item stack. In this case it is a stack of books.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Book>	Peek	setup1	None	The MyStackException comes out, communicating the error of performing the return workbook action at the top of a book stack, when this stack is empty.
MyStack <Book>	Peek	setup2	None	Book object with: isbn = 333 copies = 2 price = 20000 shelf = "A" A Book-type item at the top of the book stack was successfully returned. This stack is still equal to 1 in size, it is not empty, and therefore its top remains the returned book.
MyStack <Book>	Peek	setup3	None	Book object with: isbn = 111 copies = 6 price = 30000

				shelf = "A" A Book-type item at the top of the book stack was successfully returned. This stack is still size equal to 2, and its top remains the returned book.

Test Objective: Verify that an item stack is empty. In this case it is a stack of books.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Book>	isEmpty	setup1	None	True. The book stack is size equal to 0, so it has no book.
MyStack <Book>	isEmpty	setup2	None	False. The book stack is size equal to 1, so it is not empty.

Test Objective: Verify that the desired value(s) of a stack item(s) are displayed correctly from the top down. In this case it is a stack of books.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Book>	toString	setup3	None	"111 333" In a text string, ISBNs, separated by a space, are returned from the books present in the book stack.

Test Objective: Verify that an item stack is successfully cloned. In this case it is a stack of books.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Book>	cloneThis	setup2	None	Clone of the book stack. The book stack was successfully cloned. The size of your clone is also equal to 1, and the book at its top is the same as well. However, both the stack and its

				clone have different object identifiers.

Test Objective: Verify that the size of an item stack is returned correctly. In this case it is a stack of books.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Book>	getLength	setup1	None	0 The size of the book stack was successfully returned.
MyStack <Book>	getLength	setup2	None	1 The size of the book stack was successfully returned.
MyStack <Book>	getLength	setup3	None	2 The size of the book stack was successfully returned.

Test Objective: Verify that a new item is successfully added to the back of an item queue. In this case it is a client queue.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyQueue <Client>	enqueue	setup1	Client type object with: id = "111" time = 3	A new Client type item was successfully added to the back of a client queue. This queue is now size equal to 1, it is false that it is empty, and the client on its front (and back) is ID equal to "111".
MyQueue <Client>	enqueue	setup2	Client type object with: id = "333" time = 4	A new Client type item was successfully added to the back of a client queue. This queue is now size equal to 2, the client on its front is ID equal to "111", and the client on its back is ID equal to "333".
MyQueue <Client>	enqueue	setup3	Client type object with: id = "555" time = 5	A new Client type item was successfully added to the back of a client queue. This queue is now size equal to 3, the client on its front is ID equal to "111", and the client on its back is ID equal to "555".

Explanation of the third case in this enqueue test (setup3):

We have firstly a queue with 2 clients. The first one has id = “111”, and the second one has id

= “333”. This first one is at the front of the queue and the second one at the back. So, we begin the enqueuing process of a new client with id = “555”. In this case, we just enqueue it behind the client with id = “333”. So, the client with id = “555” is now at the back of the queue.

Test Objective: Verify that the item is successfully deleted and returned from the front of an item queue. In this case it is a client queue.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyQueue <Client>	dequeue	setup1	None	The MyQueueException comes out reporting the error of performing the delete a client on an empty client queue.
MyQueue <Client>	dequeue	setup2	None	Client object with: id = "111" time = 3 A Client-type item on the front of the client queue was successfully deleted and returned. This queue is now size equal to 0, empty, and therefore no clients are on its front and back.
MyQueue <Client>	dequeue	setup3	None	Client object with: id = "111" time = 3 A Client-type item on the front of the client queue was successfully deleted and returned. This queue is now size equal to 1, and both its front and back are assigned the client id = "333".
MyQueue <Client>	dequeue	setup4	None	Client object with: id = "111" time = 3 A Client-type item on the front of the client queue was successfully deleted and returned. This queue is now size equal to 2, on its front is now the client of id = "333", and on its back there is still the client id = "555".

Explanation of the fourth case in this dequeue test (setup4):

We have firstly a queue with 3 clients. The first one has id = “111” (at the front), the second

one with id = “333” and the third one with id = “555” (at the back). So, we begin the dequeuing process of a client. In this case, the clients can just be dequeued by the front. So, we just dequeue the client with id = “111” and therefore, the client with id = “333” is now at the front of the queue.

Test Objective: Verify that the item is successfully returned on the front of an item queue. In this case it is a client queue.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyQueue <Client>	getFront	setup1	None	The MyQueueException comes out reporting the error of performing the return client action on the front of a client queue, when this queue is empty.
MyQueue <Client>	getFront	setup2	None	Client object with: id = "111" time = 3 A Client-type item on the front of the client queue was successfully returned. This queue is still equal to 1 in size, and both its front and back are still assigned to the id client "333".
MyQueue <Client>	getFront	setup3	None	Book object with: isbn = 111 copies = 6 price = 30000 shelf = "A" A Client-type item on the front of the client queue was successfully returned. This queue is still size equal to 2, and its front is still the returned client.

Test Objective: Verify that an item queue is empty. In this case it is a client queue.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyStack <Client>	isEmpty	setup1	None	True.

				The client queue is equal to 0 in size, so it has no clients.
MyStack <Client>	isEmpty	setup2	None	False. The client queue is size equal to 1, so it is not empty.

Test Objective: Verify that a new element with Key and Value is successfully added to a chained hash table of size 16, by using a hash function (multiplication method). Arbitrary constant A = $\frac{(\sqrt{5}-1)}{2}$. In this case it is a book table.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyHashTable <Integer,Book>	insert	setup1 2 Book objects are created: Attributes of the first object: isbn1 = 441 copies1 = 3 price1 = 20000 shelf1 = "A" Attributes of the second object: isbn2 = 229 copies2 = 5 price2 = 30000 shelf2 = "A"	Key 441 Value = Book Object 1 Key 229 Value = Book Object 2	At first point, it is successfully added a new element of key = 441 and Value = Book Object 1, to the hash table at index 8. At a second point, it is successfully added another new element of key = 229 and Value = Book Object 2, to the hash table at index 8 as well. The number of items that have been added to the table is now equal to 2. However, they were added in the same index (collision), so the first item in the linked list at index 8 of the table is the key element 441, and is followed by the Key element 229.
MyHashTable <Integer,Book>	insert	setup2 An object of type Book is created with: isbn = 159	Key 159 Value = Book Object	It is successfully added a new element of key = 159 and Value = Book Object, to the hash table at index 4. The number of items that have been added to the table is now equal to 2.

		copies = 10 price = 100000 shelf = "A"		

Explanation of the first case in this insert test (setup1):

We have firstly an empty hash table. Then, we begin the inserting process of two elements. So, we need to use the hash function that uses the multiplication method to find the index where we will can insert each element in the table:

First element: h(441) = ⌊ ArraySize * ((k * A) % 1) ⌋ = ⌊ 16 * ((441 * $\frac{(\sqrt{5}-1)}{2}$) % 1) ⌋ = **8**

Second element: h(229) = ⌊ ArraySize * ((k * A) % 1) ⌋ = ⌊ 16 * ((229 * $\frac{(\sqrt{5}-1)}{2}$) % 1) ⌋ = **8**

As we have the same index for the two elements, we will have to resort to a chaining strategy with a linked list at index 8. So firstly we will put the element of key = 441 in the slot of index 8, then we will create a reference to a next element; this one will be the element of key = 229.

Test Objective: Verify that an item given its Key is searched correctly, in a chained hash table of size 16, by using a hash function (multiplication method). Arbitrary constant A = $\frac{(\sqrt{5}-1)}{2}$. In this case it is a book table.				
Class	**Method**	**Scenario**	**Input values**	**Result**

MyHashTable<Integer,Book>	search	setup1 2 Book objects are created and added: Attributes of the first object: isbn1 = 441 copies1 = 3 price1 = 20000 shelf1 = "A" Attributes of the second object: isbn2 = 229 copies2 = 5 price2 = 30000 shelf2 = "A"	Key 441 Key 229	For both the first element of Key = 441 and the second element of key = 229, are searched correctly at index 8 (collision) of the hash table and then in the linked list located at this index. Their respective values were returned: Attribute Book 1 object: isbn1 = 441 copies1 = 3 price1 = 20000 shelf1 = "A" Attribute Book 2 object: isbn2 = 229 copies2 = 5 price2 = 30000 shelf2 = "A"
MyHashTable<Integer,Book>	search	setup2 An object of type Book is created and added with: isbn = 159 copies = 10 price = 100000 shelf = "A"	Key 159	It is successfully searched for an element of Key = 159 in the hash table at index 4, and its value was returned: Attribute Book Object: isbn = 159 copies = 10 price s10,0000 shelf = "A"

MyHashTable<Integer,Book>	search	setup2	Key = 900	Null The Key element 900 was not found in the hash table.

Test Objective: Verify that an element given its Key is successfully deleted from a chained hash table of size 16, by using a hash function (multiplication method). Arbitrary constant A = $\frac{(\sqrt{5}-1)}{2}$. In this case it is a book table.

Class	Method	Scenario	Input values	Result
MyHashTable<Integer,Book>	delete	setup1	Key 1	False The Key element 1 was not deleted because the hash table is empty, so that element does not exist.
MyHashTable<Integer,Book>	delete	setup2	Key 441	True The Key element 441 was successfully removed from the hash table because it existed on the hash table. Now the table is empty.
MyHashTable<Integer,Book>	delete	setup3	Key 441	True The Key element 441 was successfully removed from the hash table because it existed on the hash table. Now the number of elements present in the hash table is equal to 1.
MyHashTable<Integer,Book>	delete	setup3	Key 159	True The Key element 159 was successfully removed from the hash table because it existed in

				the hash table. Now the number of elements present in the hash table is equal to 1.
MyHashTable<Integer,Book>	delete	setup4	Key 441	True The Key element 441 at index 8 of the hash table was successfully deleted because it existed on the hash table. Now the number of elements present in the hash table is equal to 1. As before there was collision in the mentioned index, now the previous key element is null .
MyHashTable<Integer,Book>	delete	setup4	Key 229	True The Key element 229 was successfully deleted at index 8 of the hash table because it existed on the hash table. Now the number of elements present in the hash table is equal to 1. As before there was collision in the mentioned index, now the next one of the Key element is null .

Explanation of the penultimate case in this delete test (setup4):

We have firstly a hash table with two elements at index 8, linked together. The first element of this linked list has a key = 441 and its next element has a key = 229. Then, we begin the deleting process of the element of key = 441. So, we need to use the hash function that uses the multiplication method to find the index where we will can access to this element in the table:

h(441) = ⌊ ArraySize * ((k * A) % 1) ⌋ = ⌊ 16 * ((441 * $\frac{(\sqrt{5}-1)}{2}$) % 1) ⌋ = **8**

So now, at index 8 of the hash table, we set the references of the elements of the linked list: the next element of the element of key = 441 is now the first element of the linked list. And finally the previous element of the element of key = 229 is pointing to null.

Test Objective: Verify that a hash table of items with Key and Value is empty. In this case it is a book table.				
Class	**Method**	**Scenario**	**Input values**	**Result**
MyHashTable<Integer, Book>	isEmpty	setup1	None	True. The hash table has a size equal to 0, so it has no elements.

MyHashTable<Integer, Book>	isEmpty	setup2	None	False. The hash table is size equal to 1, so it is not empty.

Test Objective: Verify that the quantities of shelves are correctly added to the library.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Bookstore	createShelves	setup2	numberOfShelves = 3	The number of shelves was successfully added to the library. Now this one has 3.

Test Objective: Verify that ATM quantities are correctly added to the library.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Bookstore	addcashiers	setup2	numberOfcashiers 3	The number of cashiers was successfully added to the bookstore. Now this one has 3.

Test Objective: Verify that a workbook's ISBNs are successfully added to a client's ISBN list.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Bookstore	addBookToClient	setup5	id = "1111" isbn = 441	Successfully added the isbn = 441 of the workbook to the client's ISBNs list with id = "1111". Now the size of this list is equal to 1.

Test Objective: Verify that a book is correctly added to a bookshelf in the library.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Bookstore	addBook	setup3	isbn = 441 copies = 3 price = 20000 shelf = "A"	The book with isbn 441 was successfully added to the shelf "A" in the library. Now the shelves aren't empty.

Test Objective: Verify that a client is successfully added to the list of clients in the library.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Bookstore	addBook	setup2	id = "1111"	False The client with id = "1111" was successfully added to the library's client list because it did not exist in the library. Now this list has size equal to 1.
Bookstore	addBook	setup2	id = "1111" id = "2222"	True The client with id = "1111" was not added correctly to the library's client list because it already existed in the library. The client list is still equal to 1. False The client with id = "2222" was successfully added to the library's client list because it did not exist in the library. The client list now has a size equal to 2.

Test Objective: By reading a text file with the expected output, verify that the complete result of testing the entire book purchase process in the library correctly matches this output.				
Class	**Method**	**Scenario**	**Input values**	**Result**
Bookstore	giveResult	setup1	typeOfSort = 'B'	The full result of the returned simulation in a String correctly matched the output present in the read text file.

Step 7 - Design implementation

Link to the source folder of the design implementation: https://bit.ly/3tSaFzh

Analysis of the temporal and spatial complexity of the sorting algorithms:

1. Bubble sort

Worst case in the input: when the elements to sort are in a descending order.

a. Time complexity:

Instruction	# number of times the instruction is repeated
Book aux;	1
for (int i = 0; i < books.size() - 1; i++) {	n
for (int j = 0; j < books.size() - i - 1; j++) {	(n(n+1) / 2) - 1
int result = books.get(j).getShelf().compareTo(books.get(j + 1).getShelf());	(n(n+1) / 2) - n
if (result > 0) {	(n(n+1) / 2) - n
aux = books.get(j);	⌈((n(n+1) / 4)⌉
books.set(j, books.get(j + 1));	⌈((n(n+1) / 4)⌉
books.set(j + 1, aux); }	⌈((n(n+1) / 4)⌉
else if (result == 0) {	1
result = books.get(j).getIsbn() - books.get(j + 1).getIsbn();	1
if (result > 0) {	1
aux = books.get(j);	1
books.set(j, books.get(j + 1));	1
books.set(j + 1, aux); }	1

} } }	
return books;	1

T(n) = 1 + n + (n (n + 1) / 2) - 1 + (n (n + 1) / 2) - n + (n (n + 1) / 2) - n + ⌈((n (n + 1) / 4)⌉ + ⌈((n (n + 1) / 4)⌉ + ⌈((n (n + 1) / 4)⌉ + 1 + 1 + 1 + 1 + 1 + 1 + 1 = **9n2 + 5n + 284 = O(n^2)**

b. Space complexity:

Type	Variable	Atomic values
Input/Output	books	n
Auxiliary	aux i j result	1 1 1 1

S(n) = n + 1 + 1 + 1 + 1= **n + 4** = **O(n)**

2. Heap sort

Worst case in the input: worst case occurs when the last row of the tree is exactly half full.

a. Time complexity:

Instruction	# number of times the instruction is repeated
int n = books.size();	1
for (int i = n / 2 - 1; i >= 0; i--)	⌊(n / 2)⌋ + 1
heapify(books, n, i);	⌊(n / 2)⌋ * heapify
for (int i = n - 1; i > 0; i--) {	n

Book temp = books.get(0);	n - 1
books.set(0, books.get(i));	n - 1
books.set(i, temp);	n - 1
heapify(books, i, 0); }	n - 1
return books;	1

T1(n) = 1 + ⌊(n / 2)⌋ + 1 + ⌊(n / 2)⌋ * [16 + T2(2n/3)] + n + n - 1 + n - 1 + n - 1 + n - 1 + 1
= 5n+n2-1+⌊(n / 2)⌋ * [16 + T2(2n/3)] = **O(n log n)**

Instruction	**# number of times the instruction is repeated**
int largest = i;	1
int l = 2 * i + 1;	1
int r = 2 * i + 2;	1
if (l < n) {	1
if(books.get(l).getShelf().compareTo (books.get(largest).getShelf()) > 0)	1
largest = l;	0
else if(books.get(l).getShelf().compareTo (books.get(largest).getShelf()) == 0) {	1
if(books.get(l).getIsbn() - (books.get(largest).getIsbn()) > 0)	1

largest = l;
}
}
	1

if (r < n) {	1
if(books.get(r).getShelf().compareTo (books.get(largest).getShelf()) > 0)	1

largest = r;	0
else if(books.get(r).getShelf().compareTo(books.get(largest).getShelf()) == 0) {	1
if(books.get(r).getIsbn() - (books.get(largest).getIsbn()) > 0)	0

largest = r;
}
}
	0

if (largest != i) {	1
Book swap = books.get(i);	1
books.set(i, books.get(largest));	1
books.set(largest, swap);	1
heapify(books, n, largest); }	T(size of subtree at largest) = T(2n/3)
return books;	1

T2(n) = **16 + T2(2n/3)** = **O(log n)**

b. Space complexity:

Heapsort:

Type	Variable	Atomic values
Input/output	books	n
Auxiliary	n i temp	1 1 1

$S_1(n) = \mathbf{n + 3} = \mathbf{O(n)}$

Heapify:

Type	Variable	Atomic values
Input/Output	books	n
Input	n i	1 1
Auxiliary	largest l r swap	1 1 1 1

$S_2(n) = \mathbf{n + 6 = O(n)}$

3. Merge sort

a. Time complexity:

merge sort

Instruction	# number of times the instruction is repeated
if (l < r) {	1
int m = l + (r - l) / 2;	1
mergeSort(books, l, m);	log2n
mergeSort(books, m + 1, r);	log2n
merge(books, l, m, r); }	1*merge
return books;	1

$T_1(n)$ = 2*log2n + 3 + merge = **2*log2n + 3 + merge** = **O(n log n)**

merge

Instruction	# number of times the instruction is repeated
int n1 = m - l + 1;	1
int n2 = r - m;	1
ArrayList<Book> L = new ArrayList<Book>();	1
ArrayList<Book> R = new ArrayList<Book>();	1
for (int i = 0; i < n1; ++i)	n1 + 1
L.add(i, arr.get(l + i));	n1
for (int j = 0; j < n2; ++j) {	n2 + 1
R.add(j, arr.get(m + 1 + j)) }	n2
int i = 0, j = 0;	1
int k = l;	1
while (i < n1&& j < n2) {	n2 + 1
int result=L.get(i).getShelf().compareTo(R.get(j).getShelf());	n2
if (result < 0) {	n2
arr.set(k, L.get(i));	0
i++; }	0
else if (result == 0) {	n2
result = L.get(i).getIsbn() - R.get(j).getIsbn();	n2
if (result <= 0) {	n2
arr.set(k, L.get(i));	0
i++; }	0

else { arr.set(k,R.get(j));	n2

j++;
}
}
	n2

else { arr.set(k, R.get(j));	0
j++; }	0
k++; }	n2
while (i < n1) {	n1 + 1
arr.set(k, L.get(i));	n1
i++;	n1
k++; }	n1
while (j < n2) {	1
arr.set(k, R.get(j));	0
j++;	0
k++; }	0
return arr;	1

$T_2(n)$ = **6 * n1+ 11 * n2 + 12 = O(n)**

b. Space complexity:

merge sort

Type	Variable	Atomic values
Input/output	books	n

Input	l r	1 1
Auxiliary	m	1

$S_1(n) = \mathbf{n + 3} = \mathbf{O(n)}$

merge

Type	Variable	Atomic values
Input/output	arr	n
Input	l m r	1 1 1
Auxiliary	n1 n2 i j k result	1 1 1 1 1 1

$S_2(n) = \mathbf{n + 9} = \mathbf{O(n)}$