Engineering Method
BookStore

Alejandro Coronel Sánchez / A00365049

Julián Andrés Riascos / A00365548

Kevin Alejandro Mera / A00364415

Problematic Context

The reading culture was reborn in Cali; therefore, a bookstore is planning to open a headquarter here. Cali has a high population, so the bookstore needs to be quick in the attention to all the clients.

Solution Development

To solve the problem of the bookstore properly, we will use the engineering method.

Step 1. Identification of the problem

- The percentage of people who are reading in Cali is high.
- The bookstore has a very wide diversity and quantity of books.
- They have not any method to find books and attend the clients fast.
- They need a fast attention to them clients.
- The solution needs to be efficient and quick.

Functional requirements

FR1: Add a book catalog with the ISBN codes, quantity of exemplars and the shelve where it is located.

FR2: Add the quantity of cashiers that will work in the day.

FR3: Add the identification of the client who entry to the store.

FR4: Add the buy list of the client with the ISBN codes.

FR5: Return the clients identification in leaving order, the purchase value, and the books in the order they were packed.

FR6: Organize the books in shelves and give a unique code to any shelve in the store.

FR7: Add books from shelves to clients books list.

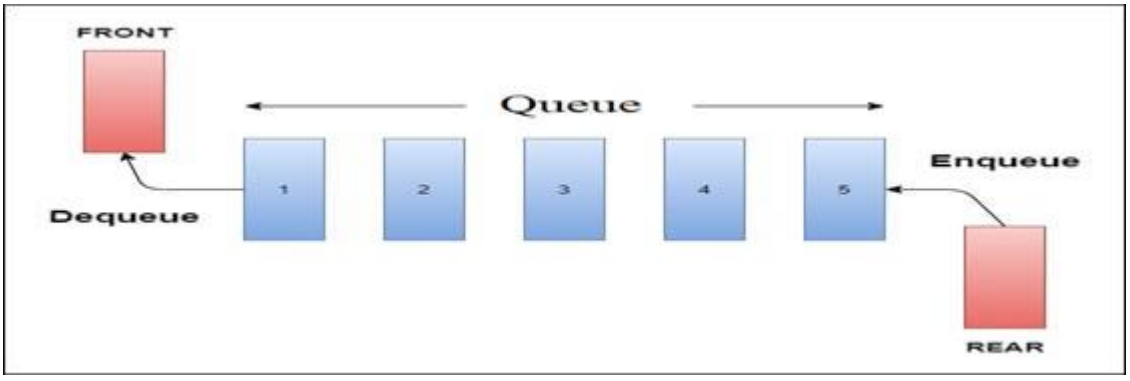
FR8: Make the payment process of clients books.

FR9: Show on the screen the final report about clients payment and books bought.

Step 2. Information Compilation

Queue

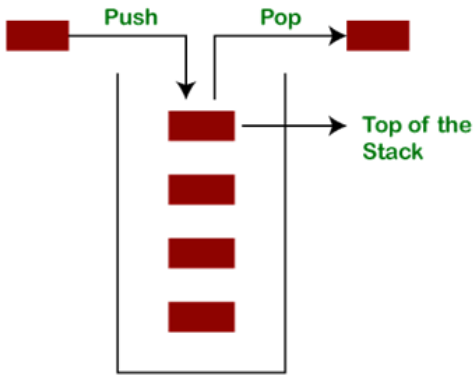
A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Stack

The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO).

The stack data structure has the two most important operations that are push and pop. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack.



What is a Hash Function?

A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as the index in the hash table.

Choosing a good hashing function, **h(k)**, is essential for hash-table based searching. **h** should distribute the elements of our collection as uniformly as possible to the "slots" of the hash table. The key criterion is that there should be a minimum number of collisions.

If the probability that a key, **k**, occurs in our collection is **P(k)**, then if there are **m** slots in our hash table, a *uniform hashing function*, **h(k)**, would ensure:

$$\sum_{k|h(k)=0} P(k) = \sum_{k|h(k)=1} P(k) = \ldots = \sum_{k|h(k)=m-1} P(k) = \frac{1}{m}$$

Sometimes, this is easy to ensure. For example, if the keys are randomly distributed in (0,**r**], then,

h(k) = floor((mk)/r)

will provide uniform hashing.

Most hashing functions will first map the keys to some set of natural numbers.

Having mapped the keys to a set of natural numbers, we then have several possibilities.

Use a **mod** function:

h(k) = k mod m.

When using this method, we usually avoid certain values of **m**. Powers of 2 are usually avoided, for **k mod 2^b** simply selects the **b** low order bits of **k**.

Use the multiplication method:

- Multiply the key by a constant **A**, 0 < **A** < 1,
- Extract the fractional part of the product,
- Multiply this value by **m**.

Thus, the hash function is:

h(k) = floor(m * (kA - floor(kA)))

In this case, the value of **m** is not critical

Use universal hashing:

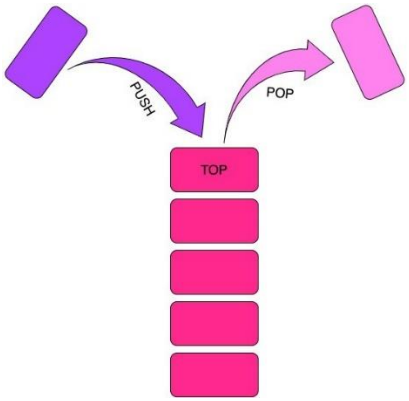
A malicious adversary can always chose the keys so that they all hash to the same slot, leading to an average **O(n)** retrieval time. Universal hashing seeks to avoid this by choosing the hashing function randomly from a collection of hash functions.

Step 3. Search of creative solutions

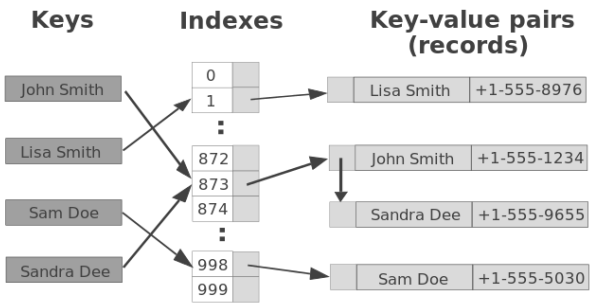
Since we already investigated and inquired about the operation of data structures such as stacks, queues, and hash tables, we can consider different possible solutions to our problem. These can be:

Option 1. Using stacks, queues, hash tables and LinkedList

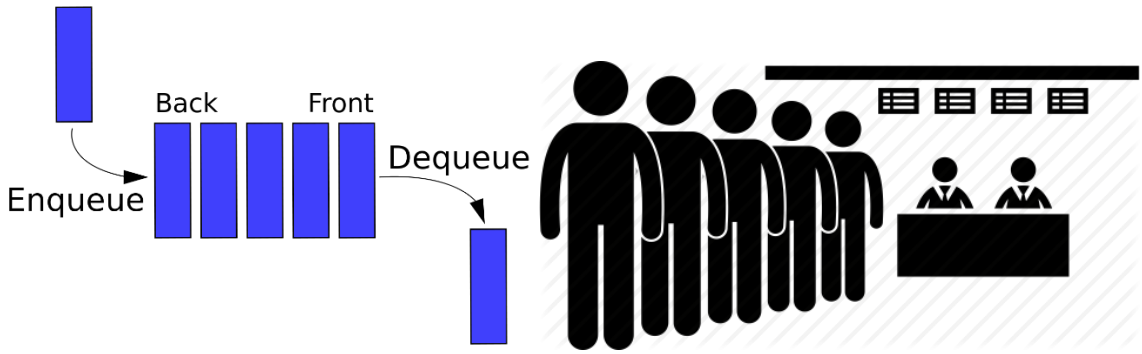
For the shopping bag we can use stacks as data structure because the client will literally have a stack of books on his hands; the first one book it took will be last and the last one it put will be the first o be paid.



Regarding to the bookshelves we can use hash tables since this data structure will help us to optimize the work about organize, find, and catch de book that clients request by ISBN code as key to access to the book.

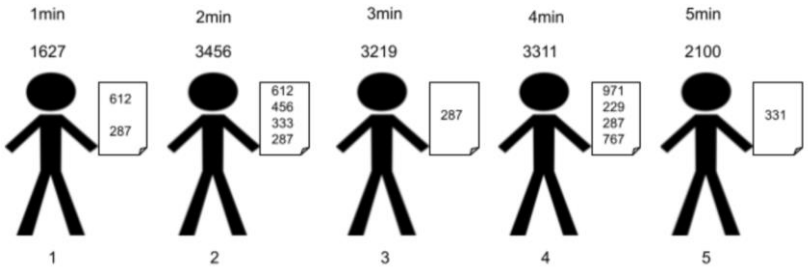


For the queue of clients, we can make use of queue data structure, this will be very close about the real client organization inside the books shop on pay stage.

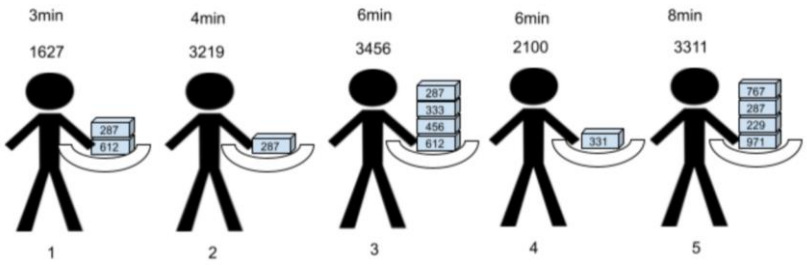


About client's organization on the stages 1, 2 and 3 will be necessary a double linked list data structure since they will be sort by time. This time will be determinate by, in the first place, the order of arrival at the store and books selection (on stage 1 and 2) and finally add time depends in the book's quantity of each client after of pickup it and then go to make the pay.

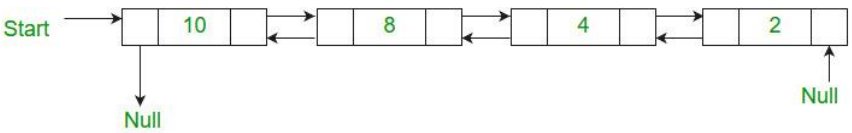
After books selection



After picking up the books

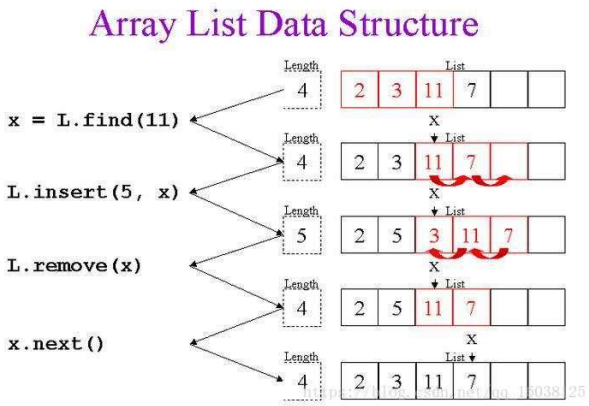


Double linked list function



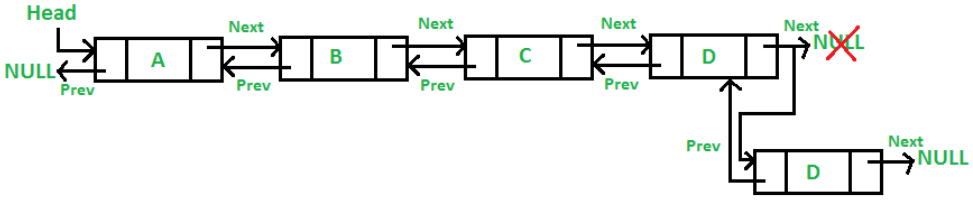
Option 2. Using only ArrayList

For this option is possible to use the ArrayList data structure from the Java API as the unique for all the requirements mentioned, like a container and sorter of the queue of clients, the shelves, and the shopping bag.



Option 3. Using only LinkedList

For this option we can think about to implement a lot of double linked lists for satisfy with the needs that the bookstore needs. Since it is its own implementation, these lists could be built in a better way (more complex methods) to try to resemble this structure to our real functionality of the store.



Step 4. Ideas transitions to preliminary design

At this point, **we discard the option 2. Using only ArrayList** since we have some data structures that are made for store, organize and access in an easiest way. In this context, these structures will be more useful. Assuming that the Arraylists will be the solution for everything we need to do could be a big mistake when carrying out the code process and run into problems that we could solve using the correct data structures that are in option 1.

The Careful review of the other alternative leads us to the following:

Option 1. Using stacks, queues, hashtables and linkedList.

- Appropriate data structures are used.
- Better organization according to the problem that arises.

- Search algorithms are implemented that are more effective.
- Less ambiguity when using different data structures and easier to understand the code for future updates.
- Arraylist or List from java API could be implemented instead using double linked list at the last part about sort clients by time.

Option 3. Using only LinkedList.

- Moldable data structure.
- May not be better organized within the context of the store in terms of shelving.
- It can have recursive methods to facilitate complex tasks by executing multiple times at a time.

Step 5. Evaluation and best solution selection

The criteria we chose in this case are the ones we list below. Next to each one a numerical value has been established with the aim of establishing a value that indicates which of the possible values of each criterion have the most tendency to be a better candidate (i.e., they are more desirable).

Criteria:

1. Criterion A. Organization. The option is organized and understandable for future improvements and data structures provide a logic organization when executing the program:

   [2] Use various data structures.
   [1] Uses few data structures.

2. Criterion B. Efficiency. The maximum complexity that may exist at the time of sorting anything data.
   [6] Logarithmic [$O(\log n)$]
   [5] Lineal [$O(n)$]
   [4] Quadratic [$O(n^2)$]
   [3] Cubic [$O(n^3)$]
   [2] Exponential [$O(x^n)$]
   [1] Factorial [$O(n!)$]

3. Criterion C. Ease of implementation.
   [1] Easy

[0] Hard

Evaluation

Evaluating the criteria of the last two alternatives, we have the following table:

	Criterion A	Criterion B	Criterion C	Total
Option 1. Using stacks, queues, hash tables and LinkedList	[2] Use various data structures.	[4] Quadratic $[O(n^2)]$	[1] Easy	7
Option 3. Using only LinkedList	[1] Uses few data structures.	[3] Cubic $[O(n^3)]$	[0] Hard	4

Selection

According to the previous analysis of criteria, it is concluded that option 1 should be chosen since it accumulates a higher score compared to option 3.

Step 6. Preparation of Reports and Specifications

Problem specification (input / output)

Problem: an efficient book purchasing and consultation system

Inputs: n, m, ISBN, b, c, are positive integers that such that n is the quantity of cashiers, m shelves quantity with their ISBN identifiers and price. Likewise, the b books quantity for each ISBN code. Later, c number of clients with their ID.

Outputs: A resume of clients purchase about total cost, and ISBN codes. All the above in order that they were packed.

Considerations:

The following cases should be considered to sort clients:

1. At the first time, the clients are sorted in order of arrival.
2. Each client takes a 1 unit of time on takes up a book, so, the time now to exit from section 3 will be the previous value from the section 2 plus the time of take up the books. At this time, this will the queue at the pay section.

Step 7. Design implementation

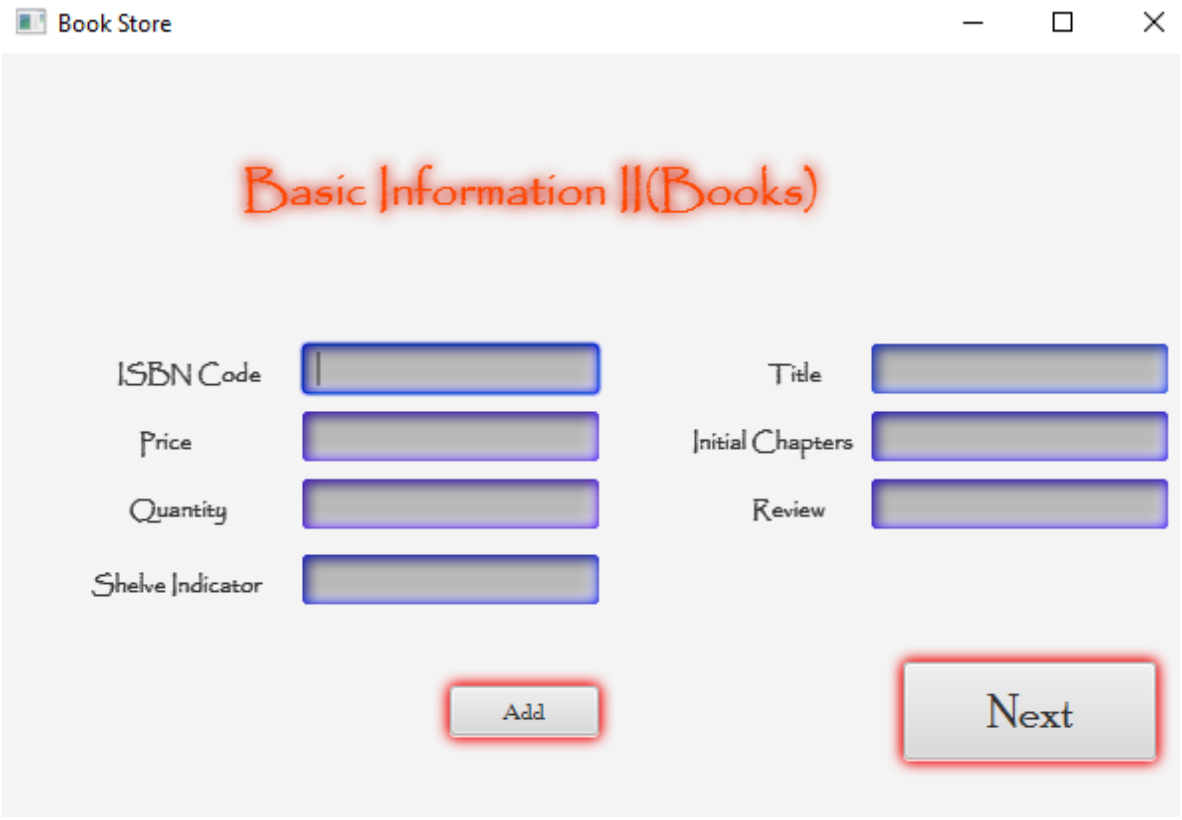
Implementation on Java programming language.

a) List of things to implement:
b) Add the quantity of cashiers.
c) Add a book catalog with the ISBN codes, quantity of exemplars and the shelve where it is located.
d) Organize the books in shelves and give a unique code to any shelve in the store.
e) Add the clients identification.
f) Add the buy list of the client with the ISBN codes.
g) Return the clients identification in leaving order, the purchase value, and the books in the order they were packed.

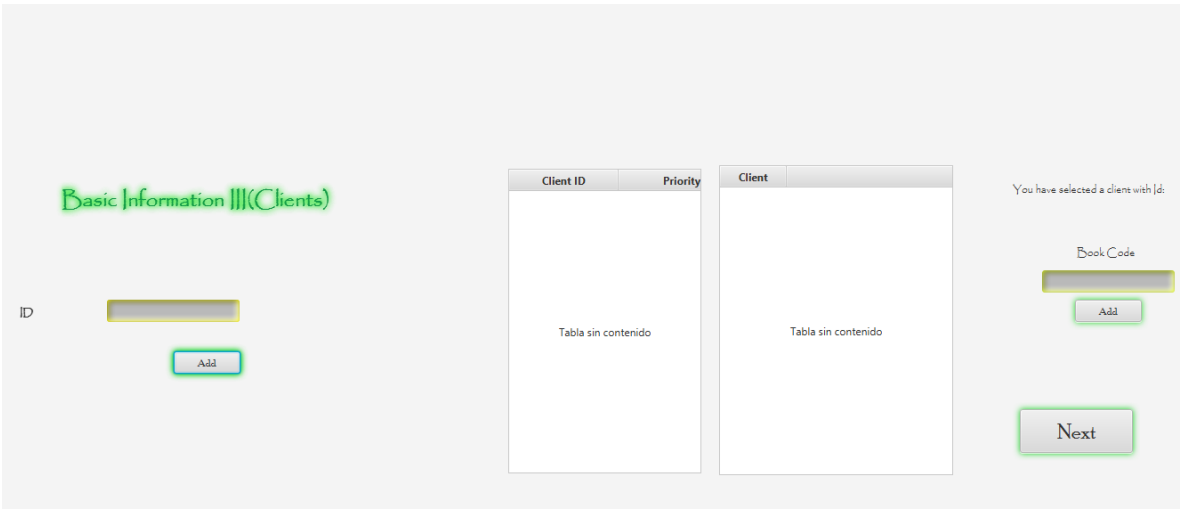
GUI Mockups

First Screen

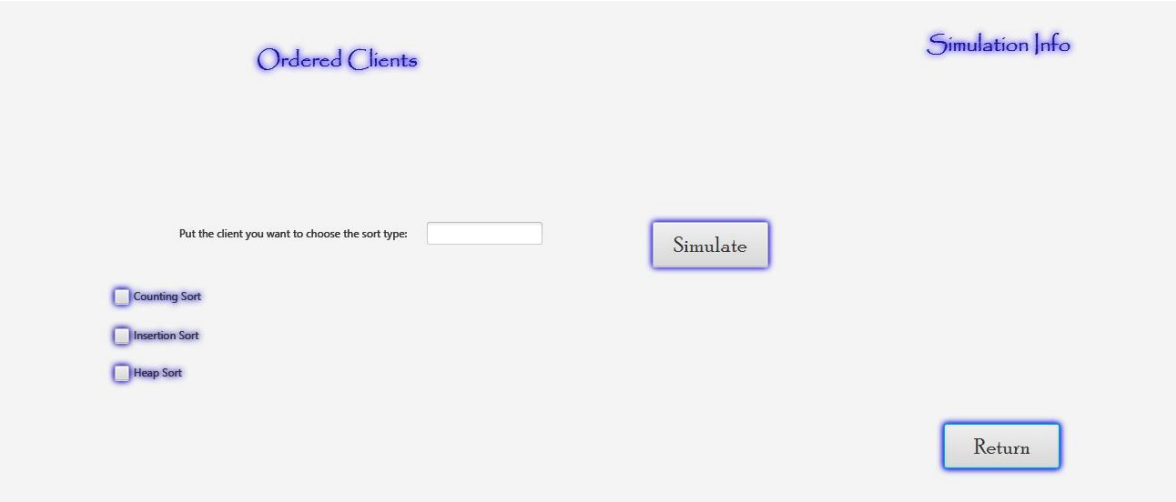
Second Screen



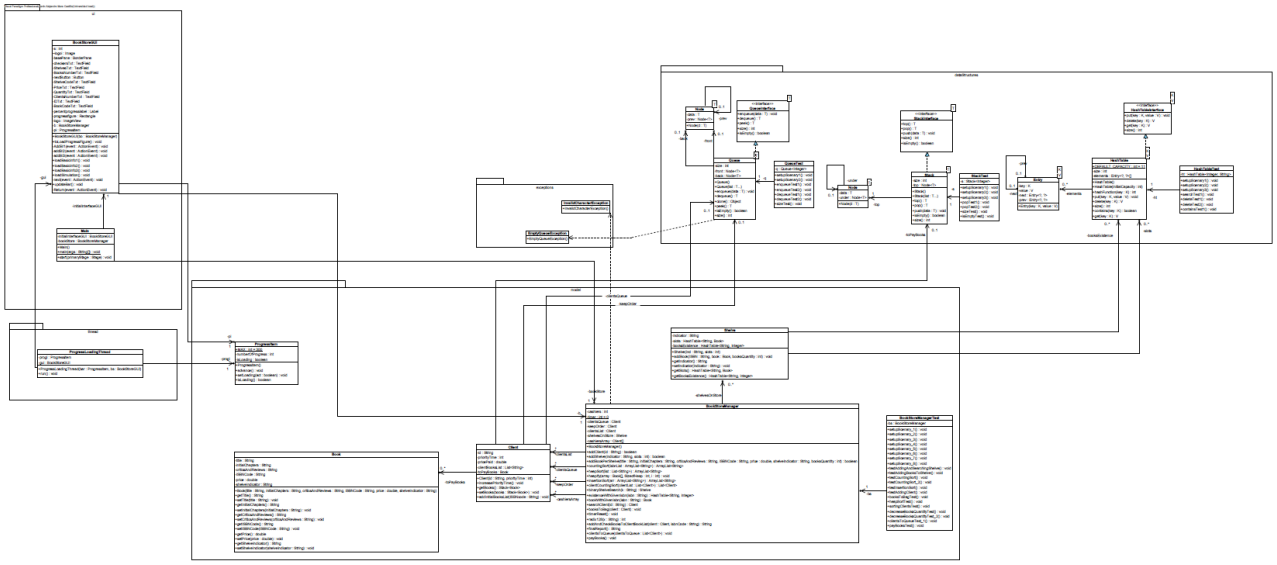
Third Screen



Fourth Screen

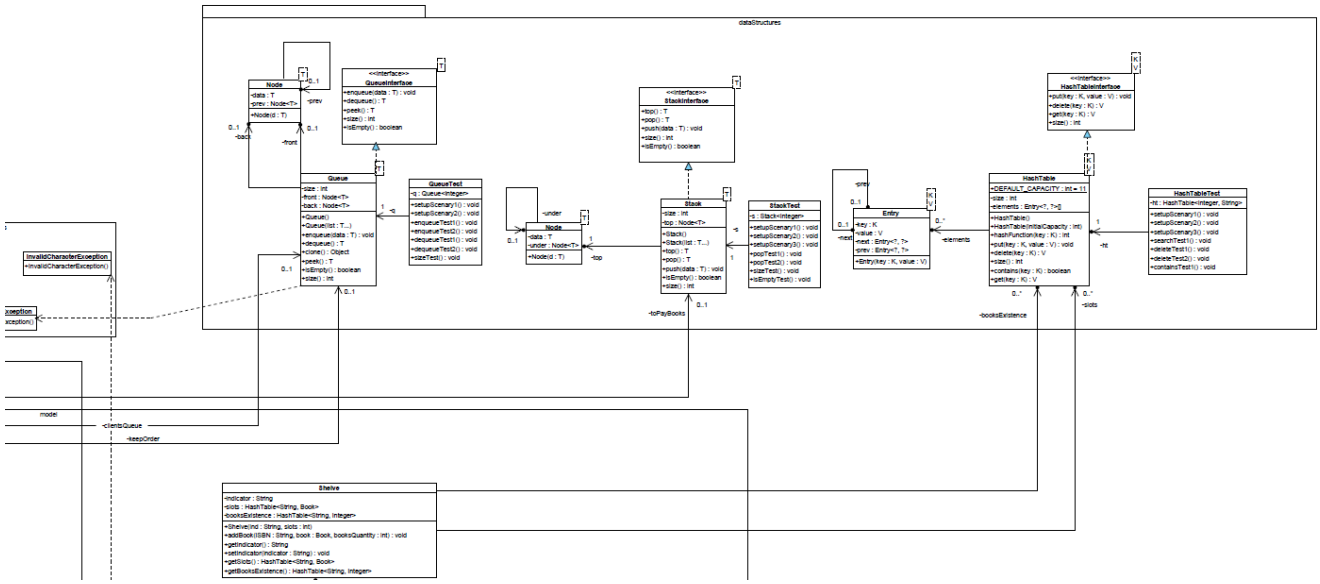


Class Diagram

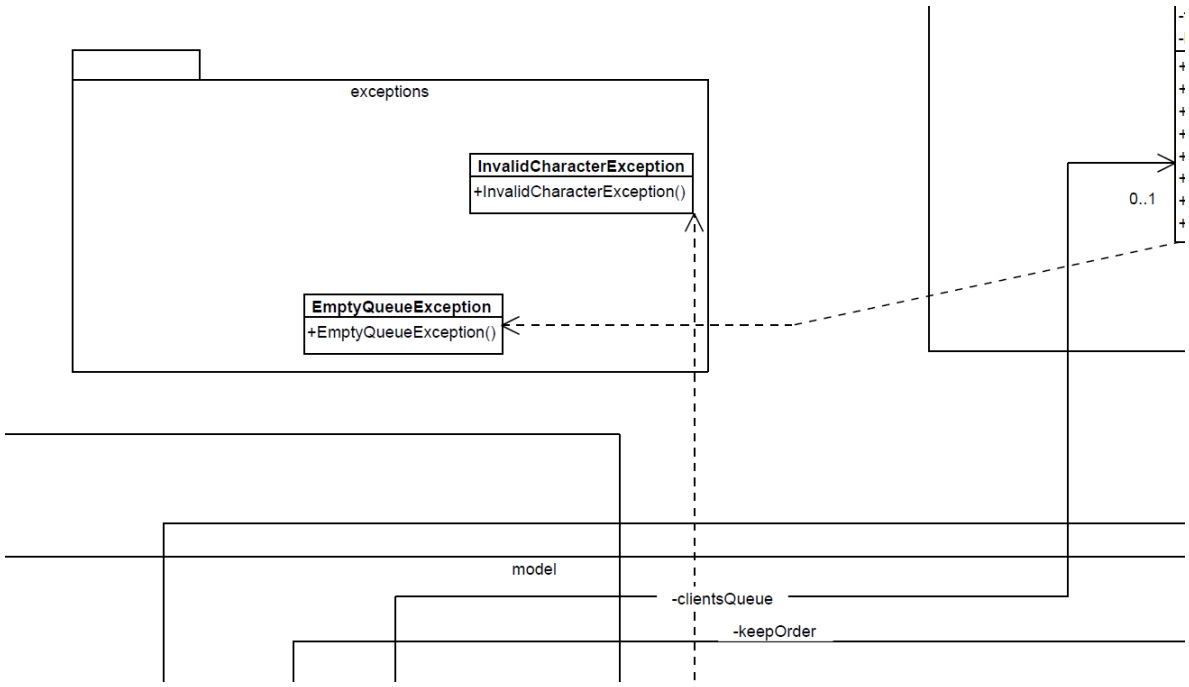


Class Diagram with zoom / pdf archive can found here:
https://github.com/foltrest2/book-store/blob/master/docs/BookStore%20Diagram.pdf

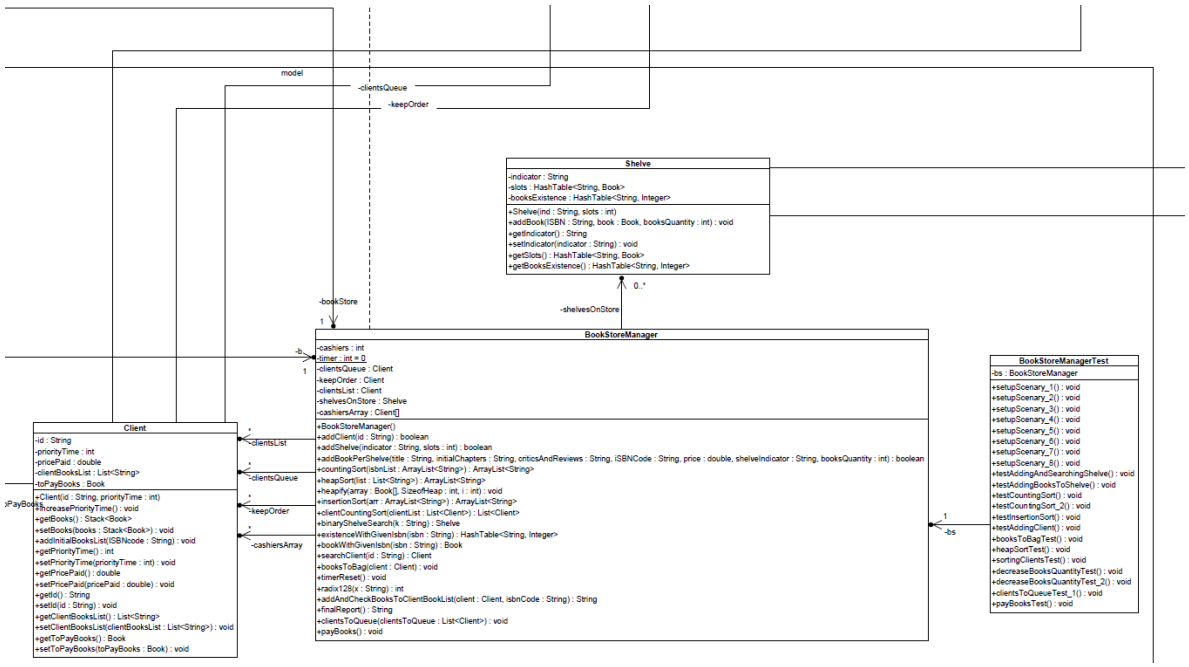
dataStrcutures package

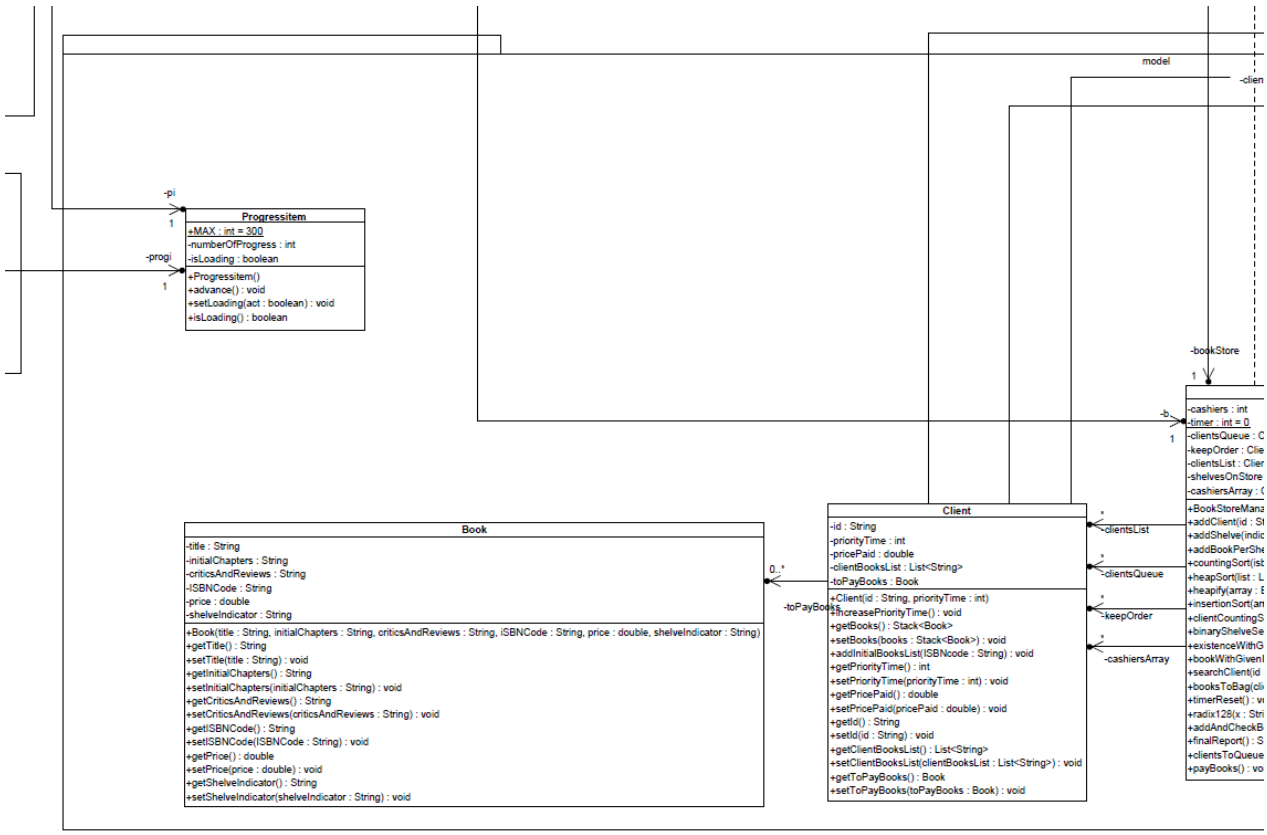


Exceptions

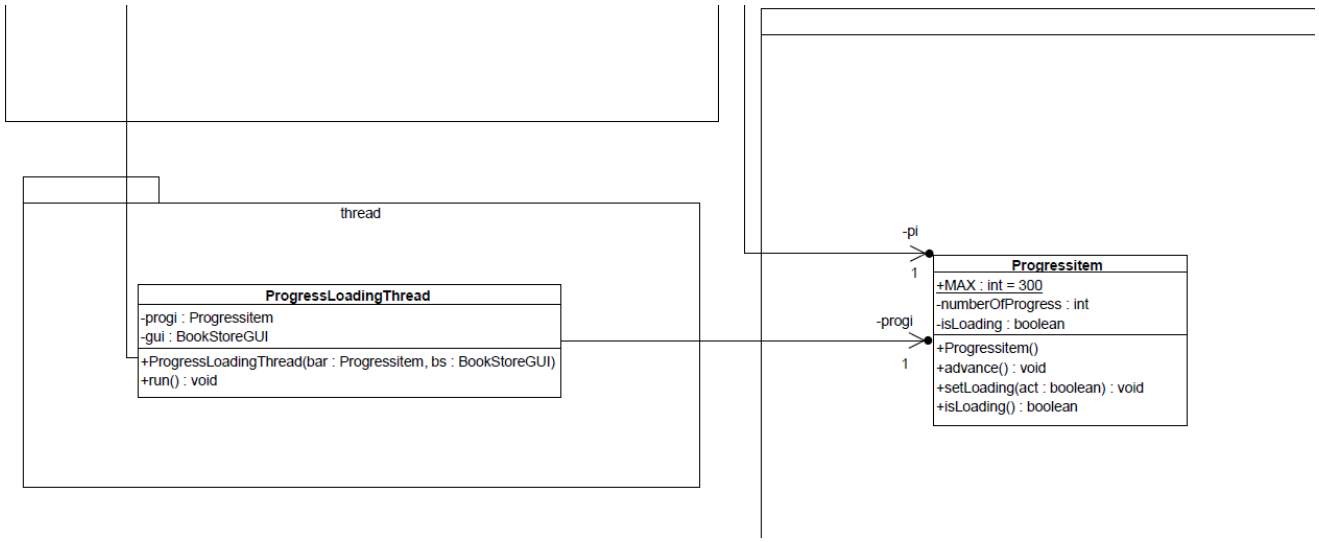


Model

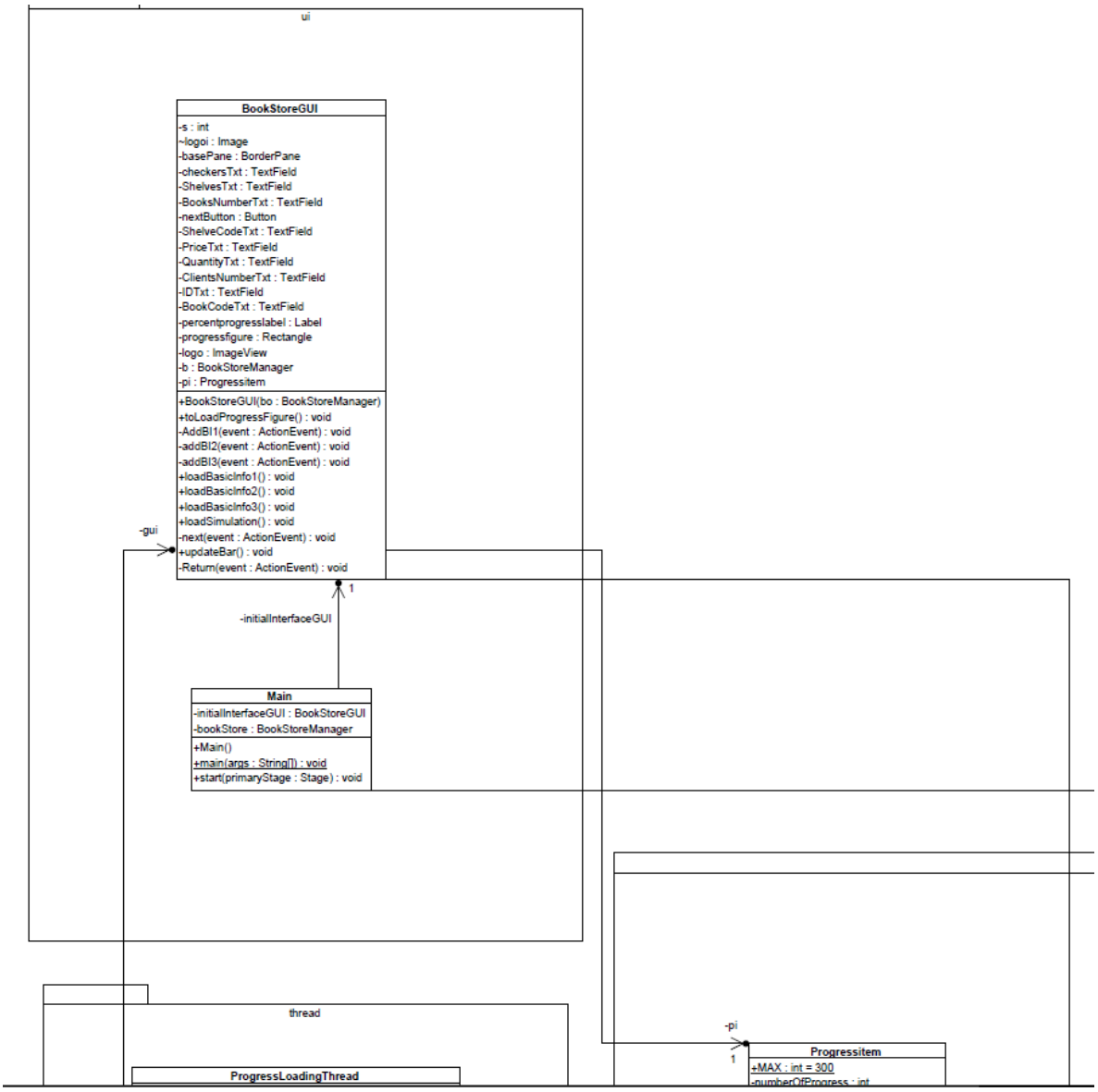

model
-clientsQueue
-keepOrder
Shelve
-indicator : String
-slots : HashTable<String, Book>
-booksExistence : HashTable<String, Integer>
+Shelve(ind : String, slots : int)
+addBook(ISBN : String, book : Book, booksQuantity : int) : void
+getIndicator() : String
+setIndicator(indicator : String) : void
+getSlots() : HashTable<String, Book>
+getBooksExistence() : HashTable<String, Integer>
0..*
-shelvesOnStore
-bookStore
1
BookStoreManager
-cashiers : int
-timer : int = 0
-clientsQueue : Client
-keepOrder : Client
-clientsList : Client
-shelvesOnStore : Shelve
-cashiersArray : Client[]
+BookStoreManager()
+addClient(id : String) : boolean
+addShelve(indicator : String, slots : int) : boolean
+addBookPerShelve(title : String, initialChapters : String, criticsAndReviews : String, ISBNCode : String, price : double, shelveIndicator : String, booksQuantity : int) : boolean
+countingSort(isbnList : ArrayList<String>) : ArrayList<String>
+heapSort(list : List<String>) : ArrayList<String>
+heapify(array : Book[], SizeofHeap : int, i : int) : void
+insertionSort(arr : ArrayList<String>) : ArrayList<String>
+clientCountingSort(clientList : List<Client>) : List<Client>
+binaryShelveSearch(k : String) : Shelve
+existenceWithGivenIsbn(isbn : String) : HashTable<String, Integer>
+bookWithGivenIsbn(isbn : String) : Book
+searchClient(id : String) : Client
+booksToBag(client : Client) : void
+timerReset() : void
+radix128(x : String) : int
+addAndCheckBooksToClientBookList(client : Client, isbnCode : String) : String
+finalReport() : String
+clientsToQueue(clientsToQueue : List<Client>) : void
+payBooks() : void
-b
1
BookStoreManagerTest
-bs : BookStoreManager
+setupScenary_1() : void
+setupScenary_2() : void
+setupScenary_3() : void
+setupScenary_4() : void
+setupScenary_5() : void
+setupScenary_6() : void
+setupScenary_7() : void
+setupScenary_8() : void
+testAddingAndSearchingShelve() : void
+testAddingBooksToShelve() : void
+testCountingSort() : void
+testCountingSort_2() : void
+testInsertionSort() : void
+testAddingClient() : void
+booksToBagTest() : void
+heapSortTest() : void
+sortingClientsTest() : void
+decreaseBooksQuantityTest() : void
+decreaseBooksQuantityTest_2() : void
+clientsToQueueTest_1() : void
+payBooksTest() : void
1
-bs
Client
-id : String
-priorityTime : int
-pricePaid : double
-clientBooksList : List<String>
-toPayBooks : Book
+Client(id : String, priorityTime : int)
+increasePriorityTime() : void
+getBooks() : Stack<Book>
+setBooks(books : Stack<Book>) : void
+addInitialBooksList(ISBNcode : String) : void
+getPriorityTime() : int
+setPriorityTime(priorityTime : int) : void
+getPricePaid() : double
+setPricePaid(pricePaid : double) : void
+getId() : String
+setId(id : String) : void
+getClientBooksList() : List<String>
+setClientBooksList(clientBooksList : List<String>) : void
+getToPayBooks() : Book
+setToPayBooks(toPayBooks : Book) : void
*
-clientsList
*
-clientsQueue
*
-keepOrder
*
-cashiersArray


model
-pi
1
Progressitem
+MAX : int = 300
-numberOfProgress : int
-isLoading : boolean
+Progressitem()
+advance() : void
+setLoading(act : boolean) : void
+isLoading() : boolean
-progi
1
-bookStore
1
-b
1
-cashiers : int
-timer : int = 0
Book
-title : String
-initialChapters : String
-criticsAndReviews : String
-ISBNCode : String
-price : double
-shelveIndicator : String
+Book(title : String, initialChapters : String, criticsAndReviews : String, ISBNCode : String, price : double, shelveIndicator : String)
+getTitle() : String
+setTitle(title : String) : void
+getInitialChapters() : String
+setInitialChapters(initialChapters : String) : void
+getCriticsAndReviews() : String
+setCriticsAndReviews(criticsAndReviews : String) : void
+getISBNCode() : String
+setISBNCode(ISBNCode : String) : void
+getPrice() : double
+setPrice(price : double) : void
+getShelveIndicator() : String
+setShelveIndicator(shelveIndicator : String) : void
0..*
-toPayBooks
Client
-id : String
-priorityTime : int
-pricePaid : double
-clientBooksList : List<String>
-toPayBooks : Book
+Client(id : String, priorityTime : int)
+increasePriorityTime() : void
+getBooks() : Stack<Book>
+setBooks(books : Stack<Book>) : void
+addInitialBooksList(ISBNcode : String) : void
+getPriorityTime() : int
+setPriorityTime(priorityTime : int) : void
+getPricePaid() : double
+setPricePaid(pricePaid : double) : void
+getId() : String
+setId(id : String) : void
+getClientBooksList() : List<String>
+setClientBooksList(clientBooksList : List<String>) : void
+getToPayBooks() : Book
+setToPayBooks(toPayBooks : Book) : void
*
-clientsList
*
-clientsQueue
*
-keepOrder
*
-cashiersArray

Thread


thread
ProgressLoadingThread
-progi : Progressitem
-gui : BookStoreGUI
+ProgressLoadingThread(bar : Progressitem, bs : BookStoreGUI)
+run() : void
-pi
1
-progi
1
Progressitem
+MAX : int = 300
-numberOfProgress : int
-isLoading : boolean
+Progressitem()
+advance() : void
+setLoading(act : boolean) : void
+isLoading() : boolean

Ui



Test cases design

StackTest

Sceneries configuration

Name	**Class**	**Stage**
setupScenary1	StackTest	A stack with 4 elements
setupScenary2	StackTest	A empty stack
setupScenary3	StackTest	A stack of integer by a defined integer Array input = {1, 2, 3, 4}

Test cases design

Test objective: check if stack is popping and making the top well				
Class	**Method**	**Setup**	**Input**	**Result**
Queue	pop	setupScenary1	None	Pulls 4 out the stack since last integer added to stack
Queue	top	setupScenary1	None	See 3 at the top of the stack

Queue	pop	setupScenary1	None	Pulls 3 out the stack since the last integer added to stack
Stack	top	setupScenary1	None	See 2 at the top of the stack

Test objective: check if stack is popping and making the top well creating stack by an array of integers				
Class	**Method**	**Setup**	**Input**	**Result**
Stack	pop	setupScenary3	None	Pulls 4 out the stack since last integer added to stack
Stack	top	setupScenary3	None	See 3 at the top of the stack
Stack	pop	setupScenary3	None	Pulls 3 out the stack since the last integer added to stack
Stack	top	setupScenary3	None	See 2 at the top of the stack

Test objective: check if stack is resizing after a pop				
Class	**Method**	**Setup**	**Input**	**Result**
Stack	size	setupScenary1	None	Size turns from 4 to 3 after making a pop

Test objective: check if stack is empty or no				
Class	**Method**	**Setup**	**Input**	**Result**
Stack	isEmpty	setupScenary2	None	True, stack is empty

QueueTest

Sceneries configuration

Name	**Class**	**Stage**
setupScenary1	QueueTest	A queue with 4 elements
setupScenary2	QueueTest	A queue of integer by a defined integer Array input = {1, 2, 3, 4}

Test cases design

Test objective: check if queue is peeking well				
Class	**Method**	**Setup**	**Input**	**Result**
Queue	peek	setupScenary1	None	See 1 out the queue since first integer added to queue

Test objective: check if queue is peeking well by creating queue due an array				
Class	**Method**	**Setup**	**Input**	**Result**
Queue	peek	setupScenary2	None	See 1 out the queue since first integer added to queue

Test objective: check if queue is dequeuing well				
Class	**Method**	**Setup**	**Input**	**Result**
Queue	dequeue & peek	setupScenary1	None	After dequeue see 2 when peeking.
Queue	dequeue & peek	setupScenary1	None	After dequeue see 3 when peeking.

Queue	dequeue & peek	setupScenary1	None	After dequeue see 4 when peeking.

Test objective: check if queue is dequeuing well by creating queue due an array				
Class	**Method**	**Setup**	**Input**	**Result**
Queue	dequeue & peek	setupScenary1	None	After dequeue see 2 when peeking.
Queue	dequeue & peek	setupScenary1	None	After dequeue see 3 when peeking.
Queue	dequeue & peek	setupScenary1	None	After dequeue see 4 when peeking.

Test objective: check if queue is not empty				
Class	**Method**	**Setup**	**Input**	**Result**
Queue	size	setupScenary1	None	True, queue is not empty and have size = 4

HashTableTest

Sceneries configuration

Name	Class	Stage
setupScenary1	HashTableTest	A hashTable with 5 elements inputs = {(2, value2), (4, value4), (8, value8), (16, value16), (18, value18)}
setupScenary2	HashTableTest	A hashTable with 5 elements inputs = {(2, value2), (4, value4), (8, value8), (8, value8), (10, value10)}
setupScenary3	HashTableTest	A hashTable with 5 elements inputs = {(123, value1), (345, value2), (678, value3), (91011, value4), (111213, value5)}

Test cases design

Test objective: check if hash table is getting the correct values				
Class	**Method**	**Setup**	**Input**	**Result**

HashTable	get	setupScenary1	None	After get value of key 2, returns value2
HashTable	get	setupScenary1	None	After get value of key 4, returns value4
HashTable	get	setupScenary1	None	After get value of key 8, returns value8
HashTable	get	setupScenary1	None	After get value of key 16, returns value16
HashTable	get	setupScenary1	None	After get value of key 18, returns value18

Test objective: check if hash table is deleting and getting well				
Class	**Method**	**Setup**	**Input**	**Result**
HashTable	get	setupScenary1	None	After get value of key 2, returns value2
HashTable	get	setupScenary1	None	After get value of key 4, returns value4
HashTable	get	setupScenary1	None	After get value of key 8, returns value8

HashTable	delete	setupScenary1	None	After delete pair with the key 16, the value deleted is value16 too
HashTable	get	setupScenary1	None	After get value of key 16, returns null
HashTable	delete	setupScenary1	None	After delete pair with the key 2, the value deleted is value2 too
HashTable	get	setupScenary1	None	After get value of key 2, returns null
HashTable	get	setupScenary1	None	After get value of key 4, returns value4
HashTable	get	setupScenary1	None	After get value of key 8, returns value8
HashTable	delete	setupScenary1	None	After delete pair with the key 8, the value deleted is value8 too
HashTable	get	setupScenary1	None	After get value of key 8, returns null
HashTable	delete	setupScenary1	None	After delete pair with the key 4, the value deleted is value4 too

HashTable	get	setupScenary1	None	After get value of key 4, returns null
HashTable	delete	setupScenary1	None	After delete pair with the key 18, the value deleted is value18 too
HashTable	get	setupScenary1	None	After get value of key 18, returns null

Test objective: check if hash table is deleting, getting, and putting well				
Class	**Method**	**Setup**	**Input**	**Result**
HashTable	get	setupScenary2	None	After get value of key 2, returns value2
HashTable	get	setupScenary2	None	After get value of key 4, returns value4
HashTable	get	setupScenary2	None	After get value of key 6, returns value6
HashTable	get	setupScenary2	None	After get value of key 8, returns value8

HashTable	get	setupScenary2	None	After get value of key 10, returns value10
HashTable	put	setupScenary2	K=12, V=value 12	Puts the pair (12, value12) into hash table
HashTable	get	setupScenary2	None	After get value of key 12, returns value12
HashTable	delete	setupScenary2	None	After delete pair with the key 10, the value deleted is value10 too
HashTable	get	setupScenary2	None	After get value of key 10, returns null
HashTable	delete	setupScenary2	None	After delete pair with the key 8, the value deleted is value8 too
HashTable	get	setupScenary2	None	After get value of key 8, returns null
HashTable	delete	setupScenary2	None	After delete pair with the key 12, the value deleted is value12 too
HashTable	get	setupScenary2	None	After get value of key 12, returns null

HashTable	delete	setupScenary2	None	After delete pair with the key 6, the value deleted is value6 too
HashTable	get	setupScenary2	None	After get value of key 6, returns null
HashTable	delete	setupScenary2	None	After delete pair with the key 4, the value deleted is value4 too
HashTable	get	setupScenary2	None	After get value of key 4, returns null
HashTable	delete	setupScenary2	None	After delete pair with the key 2, the value deleted is value2 too
HashTable	get	setupScenary2	None	After get value of key 2, returns null
HashTable	put	setupScenary2	K=14, V=value 14	Puts the pair (14, value14) into hash table
HashTable	get	setupScenary2	None	After get value of key 14, returns value14
HashTable	put	setupScenary2	K=16, V=value 16	Puts the pair (16, value16) into hash table

HashTable	get	setupScenary2	None	After get value of key 16, returns value16
HashTable	delete	setupScenary2	None	After delete pair with the key 14, the value deleted is value14 too
HashTable	get	setupScenary2	None	After get value of key 14, returns null
HashTable	delete	setupScenary2	None	After delete pair with the key 16, the value deleted is value16 too
HashTable	get	setupScenary2	None	After get value of key 16, returns null

Test objective: check if hash table is getting, contains method is fine and putting well				
Class	**Method**	**Setup**	**Input**	**Result**
HashTable	get	setupScenary3	None	After get value of key 123, returns value1
HashTable	get	setupScenary3	None	After get value of key 345, returns value2

HashTable	get	setupScenary3	None	After get value of key 678, returns value3
HashTable	get	setupScenary3	None	After get value of key 91011, returns value4
HashTable	get	setupScenary3	None	After get value of key 111213, returns value5
HashTable	contains	setupScenary3	None	After making a contains about key 123 returns true
HashTable	put	setupScenary3	K=123, V=value xd	Do not Puts the pair (123, valuexd) into hash table since key already exist
HashTable	get	setupScenary3	None	After get value of key 123, returns value1

BookStoreManagerTest

Sceneries configuration

Name	**Class**	**Stage**
setupScenary_1	BookStoreManagerTest	Create 3 different shelves to the arrayList of store shelves by their indicator and slots Shelves inputs = {(A, 4), (B, 5), (C, 5)} And add a

		book to shelve “C”. Book input = ("El dia y la noche", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "767", 50000, "C", 3)
setupScenary_2	BookStoreManagerTest	Create 3 different shelves to the arrayList of store shelves by their indicator and slots Shelves inputs = {(A, 4), (B, 5), (C, 5)} And add 3 books per shelve. Books inputs = {("El dia y la noche1", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "767", 50000, "C", 5), ("El dia y la noche2", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "123", 50000, "A", 4), ("El dia y la noche3", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "456", 50000, "B", 3)}
setupScenary_3	BookStoreManagerTest	Try to add 4 clients, 1 with a same id already added, so finally, are only 3 clients added. Clients inputs = {1234, 1234, 1235, 1236}
setupScenary_4	BookStoreManagerTest	Add 3 clients to store. Clients inputs = {123, 456, 798} Add 3 shelves to store. Shelve inputs = {(A, 4), (B, 5), (C, 5)} And add 3 books per shelve. Books inputs = {("El dia y la noche1", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "767", 50000, "C", 5), ("El dia y la noche2", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "123", 50000, "A", 4), ("El dia y la noche3", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "456", 50000, "B", 3)} Then adds

		books per clients, for client 123 books with isbn codes: {767, 123, 456} for client 456: {123, 456, 767} for the last client 798: {456, 767, 123} Finally, the books are added to each clients bags.
setupScenary_5	BookStoreManagerTest	Add 3 clients to store. Clients inputs = {123, 456, 798} Add 3 shelves to store. Shelve inputs = {(A, 4), (B, 5), (C, 5)} And add 3 books per shelve. Books inputs = {("El atardecer renaciente1", "Capitulo 1", "El amor en los tiempos del cólera...", "6545", 15500.0, "C", 5), ("El atardecer renaciente2", "Capitulo 2", "El amor en los tiempos del cólera...", "9485", 15500.0, "B", 5), ("El atardecer renaciente3", "Capitulo 3", "El amor en los tiempos del cólera...", "1654", 15500.0, "A", 5)} Finally, the books are checked by existence and then added to clients bag respectively
setupScenary_6	BookStoreManagerTest	Add 3 clients to store. Clients inputs = {123, 456, 798} Add 3 shelves to store. Shelve inputs = {(A, 4), (B, 5), (C, 5)} And add 3 books per shelve. Books inputs = {("El dia y la noche1", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "767", 50000, "C", 5), ("El dia y la noche2", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "123", 50000, "A", 4), ("El dia y la noche3", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "456", 50000, "B", 3)} Then adds books per clients. For client 123 books with isbn codes: {767, 123,

		456} for client 456: {456} for the last client 798: {456, 767, 123}. Finally, the books are added to each clients bags.
setupScenary_7	BookStoreManagerTest	Add 5 clients to store. Clients inputs = {123, 456, 798, 534, 239} Add 3 shelves to store. Shelve inputs = {(A, 4), (B, 5), (C, 5)} And add 3 books per shelve. Books inputs = {("El dia y la noche1", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "767", 50000, "C", 5), ("El dia y la noche2", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "123", 50000, "A", 4), ("El dia y la noche3", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "456", 50000, "B", 3)} Then adds books per clients. For client 123 books with isbn codes: {767,767,456} for client 456: {456} for the last client 798: {456, 767, 123} for the client 534: {456} for the client 239: {456} Finally, the books are added to each clients bag.
setupScenary_8	BookStoreManagerTest	Add 5 clients to store. Clients inputs = {123, 456, 798, 534, 239} Add 3 shelves to store. Shelve inputs = {(A, 4), (B, 5), (C, 5)} And add 3 books per shelve. Books inputs = {("El dia y la noche1", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "767", 50000, "C", 5), ("El dia y la noche2", "Capitulo 1: Erase una vez la luna y el sol...", "Buenisimo", "123", 50000, "A", 4), ("El dia y la noche3", "Capitulo 1: Erase una vez la luna y el sol...",

		"Buenisimo", "456", 50000, "B", 3)} Then adds books per clients. For client 123 books with isbn codes: {767,767,456} for client 456: {456} for the last client 798: {456, 767, 123} for the client 534: {456} for the client 239: {456} Finally, the books are added to each clients bag. Clients queue to pay and the books payment.

Test cases design

Test objective: check if is adding and searching shelves well				
Class	**Method**	**Setup**	**Input**	**Result**
BookStore Manager	binaryShelve Search	setupScenar y1	“A” as shelve indicator	After searching the indicator “A” of shelve, returns the shelve with that indicator
BookStore Manager	binaryShelve Search	setupScenar y1	“D” as shelve indicator	After searching the indicator “D” of shelve, returns null.

Test objective: check if a list of book codes is correctly sorted by counting Sort

Class	Method	Setup	Input	Result
BookStore Manager	countingSort	setupScenary_2	“noSortedBooks” as an arraylikst of unordered book codes	After getting the list of book codes, it returns an array list of sorted book codes

Test objective: check if a list of book codes is correctly sorted by Insertion Sort				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	insertionSort	setupScenary_2	“ns” as an arraylist of unordered book codes	After getting the list of book codes, it returns an array list of sorted book codes

Test objective: check if the clients are being added correctly				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	getClientList	setupScenary_3	The id of the first client	Returns the id of the first client

BookStoreManager	getClientList	setupScenary_3	none	Return the size of the current client list
BookStoreManager	getClientList	setupScenary_3	The first position of the client list	Returns the priority time of the client in the first position
BookStoreManager	getClientList	setupScenary_3	The second position of the client list	Returns the priority time of the client in the second position
BookStoreManager	getClientList	setupScenary_3	The third position of the clientlist	Returns the priority time of the client in the third position

Test objective: check if the books are added correctly to a client bag				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	getClientList	SetupScenary_4	The position of the first client	Return the book code of the last book that was put into the client bag

BookStoreManager	getClientList	SetupScenary_4	The position of the second client	Return the book code of the last book that was put into the client bag
BookStoreManager	getClientList	SetupScenary_4	The position of the third client	Return the book code of the last book that was put into the client bag

Test objective: check if a list of book codes are correctly ordered by heap sort				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	heapSort	SetupScenary_5	The book list of the client in the first position of the client list	Return an ordered book code list of the client

Test objective: check if a list of book codes are correctly ordered by heap sort				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	heapSort	SetupScenary_5	The book list of the client in the first position of the client list	Return an ordered book code list of the client

Test objective: check if the clients are sorted correctly				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	clientCountingSort	SetupScenary_6	The current client list	A sortered client list
BookStoreManager	getClientList	SetupScenary_6	The first position in the	Return the id of

			new sortered list	the client in the first position of the sortered list
BookStoreManager	getClientList	SetupScenary_6	The second position in the new sortered list	Return the id of the client in the second position of the sortered list
BookStoreManager	getClientList	SetupScenary_6	The third position in the new sortered list	Return the id of the client in the third position of the sortered list
BookStoreManager	getClientList	SetupScenary_6	The first position in the sortered client list	The priority time of the client in the first position

BookStoreManager	getClientList	SetupScenary_6	The second position in the sortered client list	The priority time of the client in the second position
BookStoreManager	getClientList	SetupScenary_6	The third position in the sortered client list	The priority time of the client in the third position

Test objective: check if the books quantity is decreasing by clients pickup				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	getBooksExistence	SetupScenary_6	None	The book with code 767 now has 3 remaining
BookStoreManager	getBooksExistence	SetupScenary_6	None	The book with code 123 now has 4 remaining
BookStoreManager	getBooksExistence	SetupScenary_6	None	The book with code 456 now

				has 1 remaining
BookStoreManager	getBooksExistence	SetupScenary_7	None	The book with code 767 now has 2 remaining
BookStoreManager	getBooksExistence	SetupScenary_7	None	The book with code 123 now has 5 remaining
BookStoreManager	getBooksExistence	SetupScenary_7	None	The book with code 456 now has 0 remaining
BookStoreManager	getBooksExistence	SetupScenary_7	None	The client 239 returns null since do not have books added

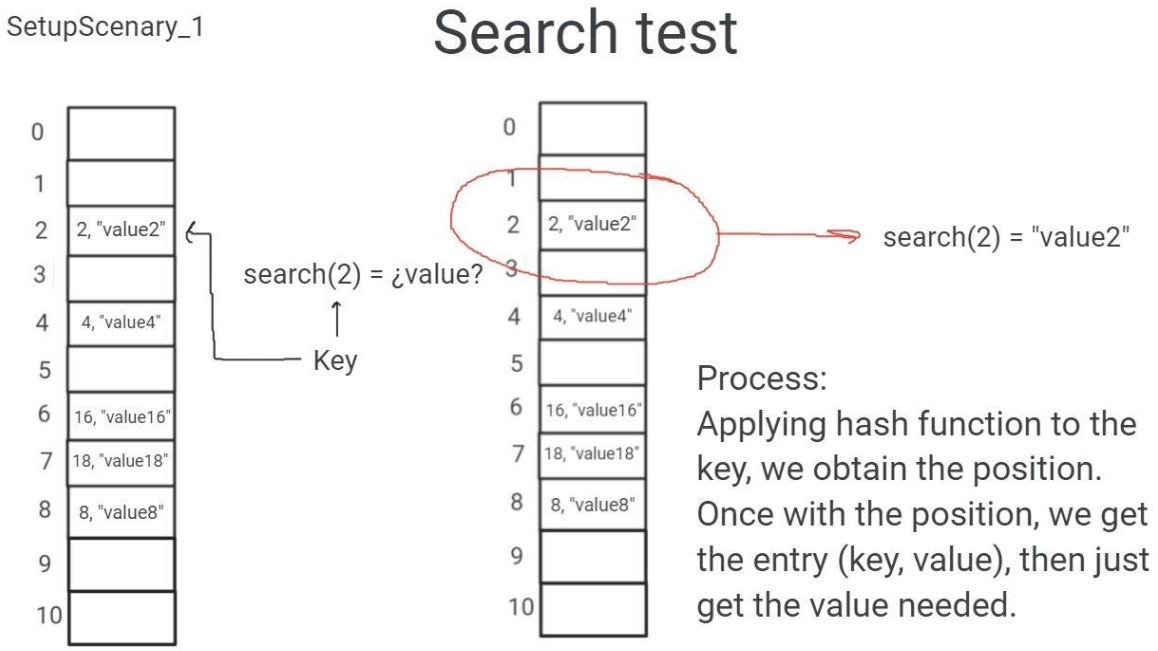
Test objective: check if the clients are getting correctly queued.				
Class	**Method**	**Setup**	**Input**	**Result**

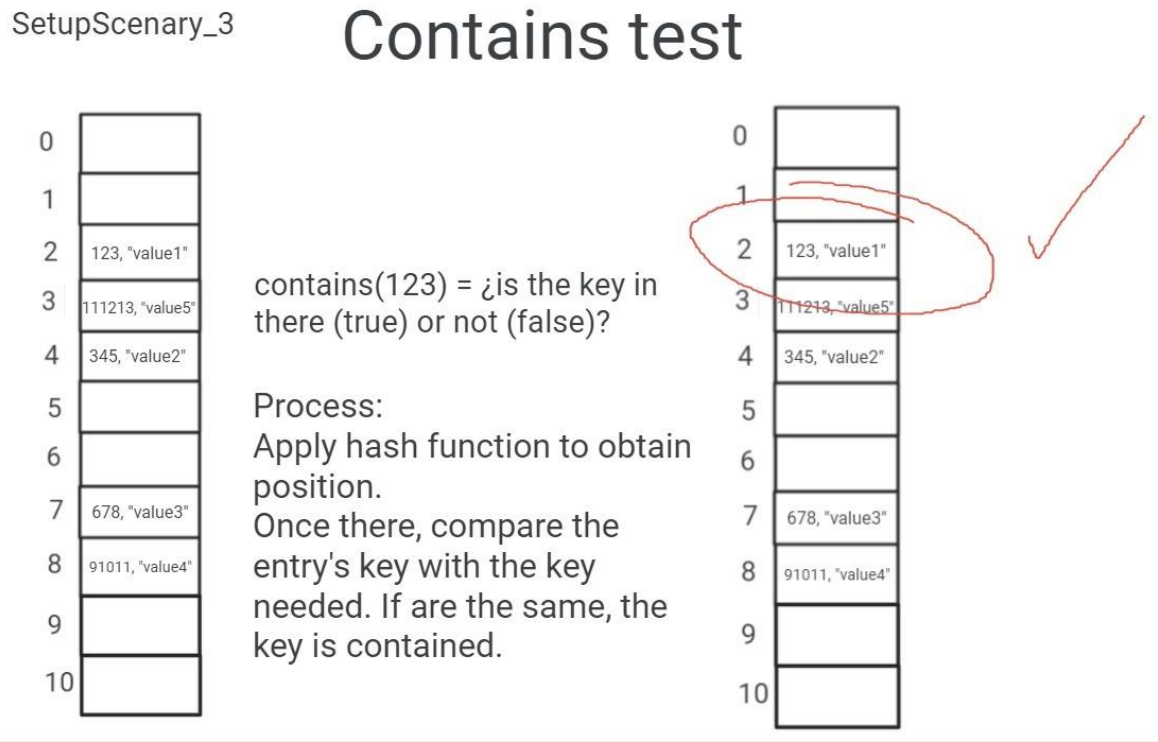
BookStoreManager	clientsToQueue	SetupScenary_7	Sorted clients	Clients are now enqueued
BookStoreManager	getClientsQueue	SetupScenary_7	None	Client with id “456” is the first to dequeue
BookStoreManager	getClientsQueue	SetupScenary_7	None	Client with id “123” is the second to dequeue
BookStoreManager	getClientsQueue	SetupScenary_7	None	Client with id “534” is the third to dequeue
BookStoreManager	getClientsQueue	SetupScenary_7	None	Client with id “798” is the fourth to dequeue

Test objective: check if the clients pay the correct amount of money for their books				
Class	**Method**	**Setup**	**Input**	**Result**
BookStoreManager	clientsToQueue	SetupScenary_7	Sorted clients	Clients are now enqueued
BookStoreManager	payBooks	SetupScenary_7	None	Clients are now dequeued, Client in the position “3” has to pay $ 0, because the book he wanted to buy is not available anymore

Graphic explanations of data structures tests

Hash table tests



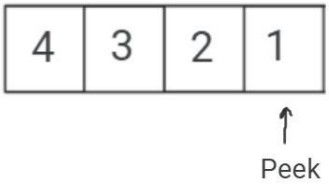


Queue test

Enqueue test

SetupScenary_1

Step 1: empty queue



Step 2: enqueue number 1

Step 3: enqueue number 2

Step 4: enqueue number 3

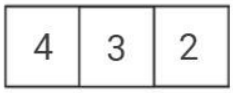
Step 5: enqueue number 4

Process:
If all going well, the peek of the queue must be the number 1. Therefore, I confirm that giving a correct answer.

Dequeue test

Using SetupScenary_1

Step 1: dequeue number 1 → peek = 1



Step 2: dequeue number 2 → peek = 2

Step 3: dequeue number 3 → peek = 3

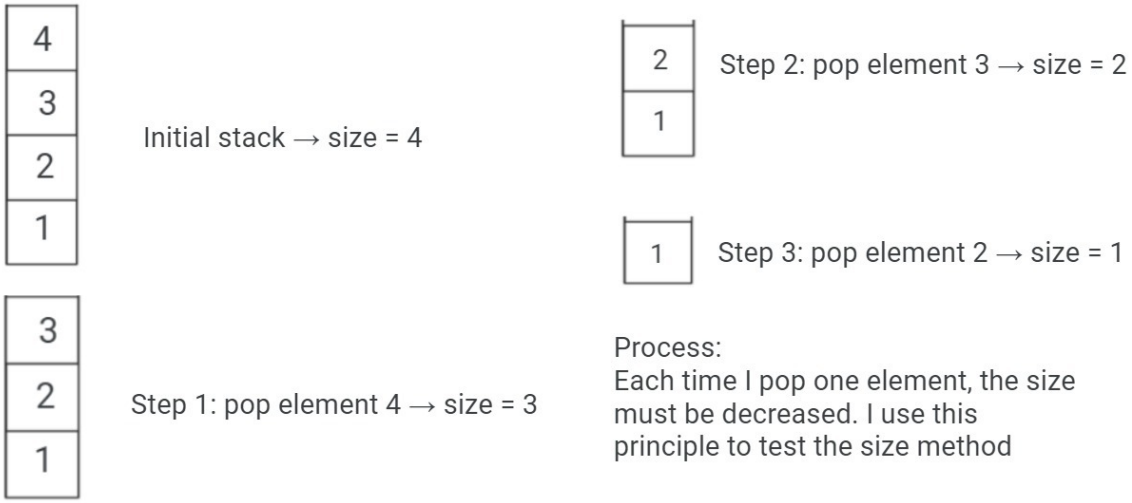
Step 4: dequeue number 4 → peek = 4

Process: Using the same principle of the peek, I verify if the peek when i'm dequeuing is the correct

Stack tests

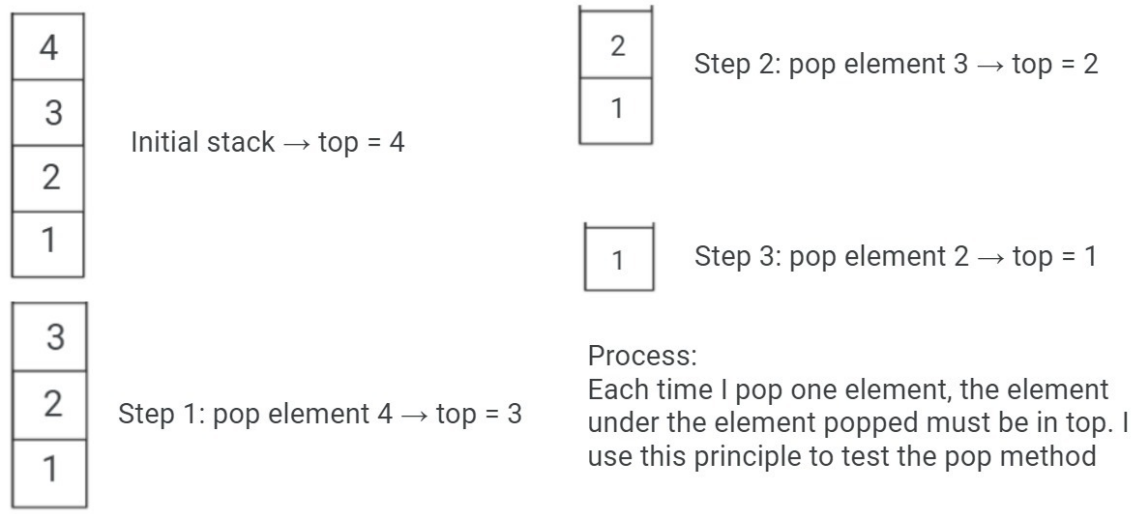
Size test

Using setupScenary_1



Pop test

Using setupScenary_1



Spatial complexity analysis

Counting sort

Type	Variable	Atomic values quantity

Entry	ArrayList<String> isbnList	n
Auxiliar	Book [] books	n
	Int [] counts	127
	Int sumTillLast	1
	Int currentElement	1
	Int positionOfInsert	1
	Book [] outputArray	n
Output	ArrayList<String> sortedBooks	n

$$T(n) = 4n + 130$$

$$T(n) = O(n)$$

Counting sort has a O(n) spatial complexity

Heap sort

Type	Variable	Atomic values quantity
Entry	ArrayList<String> isbnList	n
Auxiliar	Book [] books	n
	Int size	1
Output	ArrayList<String> isbnSorted	n

Heapify

Type	Variable	Atomic values quantity
Entry	Book [] array	n
	Int sizeOfHeap	1
	Int i	1

$$T(n) = 3n + 1 + (n + 2)$$

$$T(n) = 4n + 3$$

$$T(n) = O(n)$$

Heap sort has a O(n) spatial complexity

Insertion sort

Type	Variable	Atomic values quantity
Entry	ArrayList<String> arr	n
Auxiliar	String current	1
	Int i	1
Output	ArrayList<String> arr	n

$$T(n) = 2n + 2$$

$$T(n) = O(n)$$

Insertion sort has a O(n) spatial complexity

Temporal complexity

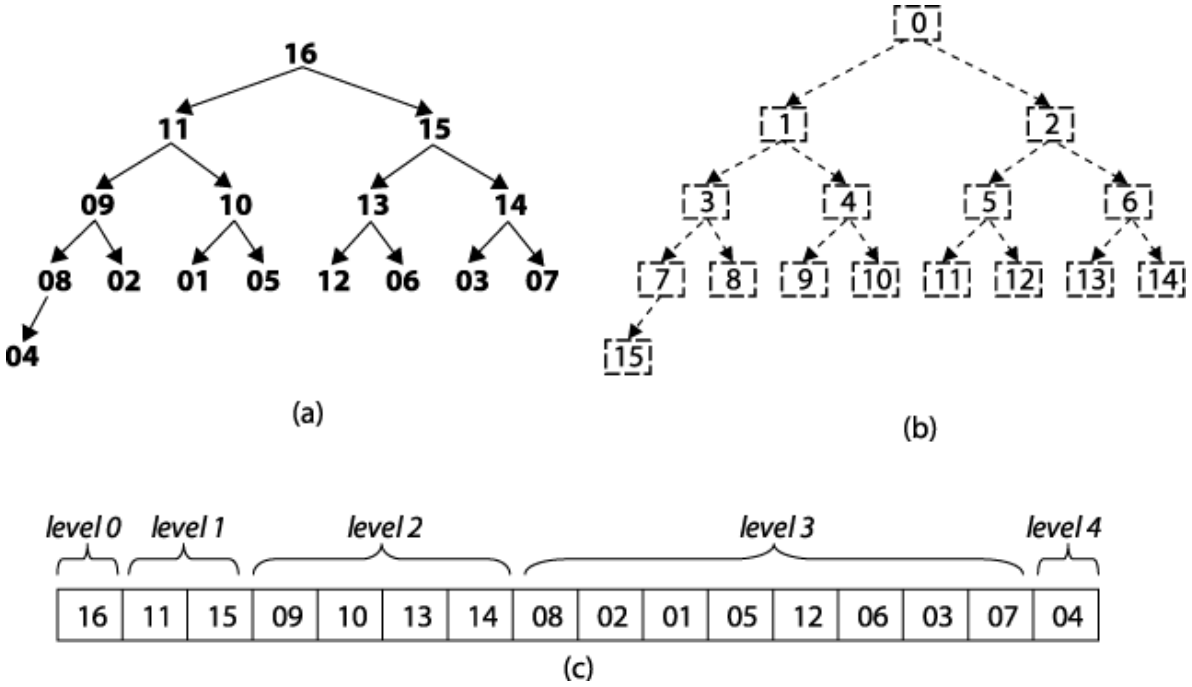
Insertion sort

Line	Number of times that line executes
`for (int j = 1; j < arr.size(); j++) {`	n
`String current = arr.get(i);`	n-1
`int i = j - 1;`	n -1
`while ((i > -1) && (arr.get(i).compareTo(current)>0)) {`	n * (n+1) * (½) - 1
`arr.set(i+1,arr.get(i));`	(n -1)* n * (½)
`i--;`	(n-1) * n * (½)
`}`	-
`arr.set(j + 1, current);`	n-1
`}`	-

$$T(n) = \frac{3n^2}{2} + \frac{7n}{2} - 4$$

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

Heap sort



The height of the tree is log(n), and the cost of building the heap is n, so the overall time complexity is O(n log(n)).

Counting sort

Line	Number of times that line executes
`Book [] books = new Book[isbnList.size()];`	1
`for (int i = 0; i < isbnList.size(); i++) {`	n+1
`books[i] = bookWithGivenIsbn(isbnList.get(i));` `}`	n
`int[] counts = new int[127];`	1
for (int i = 0; i < books.length; i++) {	n+1
`counts[radix128(books[i].getShelveIndicator())]++;` `}`	n
`int sumTillLast = 0;`	1
`for (int i = 0; i < counts.length; i++) {`	n+1
int currentElement = counts[i];	n
counts[i] = sumTillLast;	n

sumTillLast = sumTillLast + currentElement; }	n
Book[] outputArray = new Book[books.length];	1
ArrayList<String> sortedBooks = new ArrayList<>();	1
for (int i = 0; i < books.length; i++) {	n+1
int positionOfInsert = counts[radix128(books[i].getShelveIndicator())];	n
outputArray[positionOfInsert] = books[i];	n
counts[radix128(books[i].getShelveIndicator())]++; }	n
for (int i = 0; i < books.length; i++) {	n+1
sortedBooks.add(outputArray[i].getISBNCode()); }	n
return sortedBooks; }	1

$$T(n) = 14n + 11$$

$$T(n) = O(n)$$

ADT DESIGNS

STACK ADT		
Stack = {{$element_1, element_2, element_3 \dots element_n$}, top}		
{inv: $n \geq 0 \ \wedge n = size(stack) \ \wedge top = element_n$}		
Primitive Operations:		
• CreateStack • IsEmpty • Push • Pop • Top	• Stack • Stack x Element • Stack • Stack	→ Stack → boolean → Stack → Stack → Element

CreateStack()
"Make a empty stack"

{Pre: True}
{post: Stack s_1 = ∅}
Operation type: constructor

IsEmpty()
“Indicates if stack is empty or no”
{Pre: Stack s_1}
{post: True if s_1 = ∅, false if $s_1 \neq \emptyset$}
Operation type: Analyzer

Push (newElement)
“Adds the newElement to stack s_1”
{Pre: (Stack $s_1 = \emptyset \vee s_1 \neq \emptyset$) ∧ newElement}
{post: ($s_1 = \{element_1, element_2, element_3 \dots element_n\}$ Ù top = newElement) Ú($s_1 = \{newElement\}$)}
Operation type: Modifier

Pop ()
“Remove the last item added to the stack s_1”
{Pre: Stack $s_1 \neq \emptyset$ Ù $s_1 = \{element_1, element_2, element_3 \dots element_n\}$ }
{post: $s_1 = \{element_1, element_2, element_3 \dots element_{n-1}\}$}
Operation type: Modifier
Top ()
“Gives the value stored at the top of the stack”
{Pre: Stack $s_1 \neq \emptyset$ Ù $s_1 = \{element_1, element_2, element_3 \dots element_n\}$ }
{post: $element_n$}
Operation type: Analyzer

Queue ADT
Queue = {{$element_1, element_2, element_3 \dots element_n$}, front, back}
{inv: $n \geq 0 \wedge n = size(stack) \wedge front = element_1 \wedge back = element_n$}

Primitive Operations:		
• CreateQueue • IsEmpty • Enqueue • Dequeue • Front	• Queue • Queue x Element • Queue • Queue	→ Queue → boolean → Queue → Element → Element

CreateQueue()
"Make an empty Queue q"
{Pre: True}
{post: Queue q_1 = ∅}
Operation type: constructor

IsEmpty()
"Indicates if queue is empty or no"
{Pre: Queue q_1}
{post: True if q_1 = ∅, false if $q_1 \neq \emptyset$}
Operation type: Analyzer

Enqueue (newElement)
"Adds the newElement to queue q_1"
{Pre: (Queue $q_1 = \emptyset \vee q_1 \neq \emptyset$) ∧ newElement }
{post: ($q_1 = \{element_1, element_2, element_3 \dots element_n\}$ Ù back = newElement) ∨($q_1 = \{newElement\}$)}
Operation type: Modifier

Dequeue ()
"Remove the front element on the queue q_1"
{Pre: (Queue $q_1 \neq \emptyset$) Ù ($q_1 = \{element_1, element_2, element_3 \dots element_n\}$)}
{post: $q_1 = \{element_1, element_2, element_3 \dots element_{n-1}\}$}
Operation type: Modifier

Front ()
"Gives the value stored at the front of the queue"
{Pre: (Queue $q_1 \neq \emptyset$) Ù ($q_1 = \{element_1, element_2, element_3 \dots element_n\}$)}
{post: $element_1$}

Operation type: Analyzer

HashTable ADT		
HastTable = {{$pair_1, pair_2, pair_3 \dots pair_n$} Ù $pair = \{k, v\}$ }		
{inv: $\forall_{Pair}$ (Pair ≠ ∅ Ù $Pair.K \neq \emptyset$) Ù $\forall_{k_1,k_2 \dots k_n}(k_1 \neq k_2)$ }		
Primitive Operations:		
• CreateHashTable • ContainsKey • Put • Delete • Get	• HastTable x K • HastTable x K x V • HastTable x K • HastTable x K	→ HastTable → boolean → HastTable → V → V

CreateHashTable()
"Make an empty HashTable"
{Pre: True}
{post: HashTable t_1 = ∅}
Operation type: constructor

ContainsKey(key)
"Indicates if the HashTable t_1 contains a specific key"
{Pre: HashTable $t_1 \neq \emptyset$ Ù $t_1 = \{pair_1, pair_2, pair_3 \dots pair_n\}$ }
{post: True if pair.k ≠ ∅, false if pair.k = ∅
Operation type: Analyzer

Put(key, value)
"put an element in the HashTable by a K key and V value"
{Pre: HashTable $t_1 \neq \emptyset$ Ù $t_1 = \{pair_1, pair_2, pair_3 \dots pair_n\}$ }
{post: $t_1 = \{pair_1, pair_2, pair_3 \dots pair_n\}$ Ù $\exists_{pair}(pair \in t_1$ Ù $pair.k = key$ Ù $pair.v = value\}$
Operation type: Modifier

Delete(key)
"search an Entry with a respect K key in the HashTable and sets to null"
{Pre: HashTable $t_1 \neq \emptyset$ Ù $t_1 = \{pair_1, pair_2, pair_3 \dots pair_n\}$ }

{post: $\neg\exists_{Pair}(pair \in t_1 \text{ Ù } pair.k = key \text{ Ù } pair \neq \emptyset)$}
Operation type: Modifier

Get(key)
“Gets the value V according to the K key of the entry in the HashTable or null if K does not exist”
{Pre: HashTable $t_1 \neq \emptyset \text{ Ù } t_1 = \{pair_1, pair_2, pair_3 \dots pair_n\}$ }
{post: returns V value if $\exists_{Pair}(pair \in t_1 \text{ Ù } pair.k = key \text{ Ù } pair \neq \emptyset \text{ Ù } pair.v \neq \emptyset)$ returns null if pair.k $\notin t_1$}
Operation type: Analyzer

Post-mortem Report

We consider that the strong work between us was of great need for this project. For the first time the GitHub was used with branches for each member which was practical but at the same time a bit tedious since sometimes we forgot to go to each of our branches. We consider that we had a good job despite the bugs we may have, as they were resolved in a better time than expected. We highlight the opportunity that the AED course gives us to carry out this integrator in a group to strengthen our teamwork and therefore the experience in GitHub.

In general terms, we consider that there was an excellent job on the part of all of us and it was contributed equally.

The aspects to improve could be to have a better planning and the next time to start earlier, however it is complicated since the academic load of the other classes is strong and dedicating, we totally to programming involves a very considerable time.