Jose Manuel Ossorio
Juan David Aguirre
Marco Antonio Pérez

Engineering Method

Phase 1: identifying the problem

- Description of the problem context:
    - A bookstore requires a software that can simulate how the process of purchasing a book will be when a group of clients visit the store to buy books
- Description of the problem:
    - The bookstore requires the development of a software that can output the departure order of every client, the total value of each purchase and the order in which each client's books were packed, all this taking a set of inputs describing the situation.
- Functional Requirements:
    - The software must:
        - R1: indicate the order of departure of the clients based on each client's order of arrival and time spent at the payment process stage
        - R2: indicate the total amount that each client payed
        - R3: indicate the order in which the books were packed for each client
        - R4: calculate the time it will take for each client to pay and receive their books packed
        - R5: determine the order in which the clients will enter the cash register

- Non-functional requirements:
    - The software must:
        - R1: Have a GUI
        - R2: Be able to find the books quickly using the book's ISBN

Phase 2: Compilation of necessary information:

After analyzing the requirements of the software, we have come to a set of data structures that will make our task possible. We think that we will need to use stacks, queues or heaps and hash tables as the main data structures in order to solve the problem.

That said, we will use the knowledge acquired in our classes about these data structures along with other resources that we will research online if needed.

Phase 3: Creative solutions

We came across the question of whether a heap data structure would be a better approach to the line of clients because heap gives us a good field if the priority of Attention of clients change.

We thought about how and when we could use data structures to solve the problem:

- Stacks: we could use stacks in two scenarios. The first one is the list of books that the client picks. This is a stack because it has LIFO behavior, meaning that the last book that was put on the list will be the first one to be registered. This registration happens to involve another stack, on the cash register's side. So in summary, we will need at least two stacks to work our solution. P.S: in the stack we can only access the last book added, we can t search, delete books out of the top in the stack.

- Queues: when the clients approach the cash register they will need to form in a line. This line will have FIFO behavior, which is why we could use a queue to represent this scenario. The clients will enter the queue in the order they enter the store. P.S.: in the queue we can only get the first book added, we can t search, delete books out of the first position of the queue.

- Hash Tables: we could implement a hash table to store the bookstore's books because the problem tells us that the search for each book must be quick and so we came to the conclusion that a Hash Table will be the most efficient data structure to implement in this case. P.S.: if the library has too many books and the hash table isn't long enough, searching and adding would be close to O(n).

- ArrayList: an ArrayList could be used to store the list of books and it would suffice the time condition since it would return each element in O(1) time if we know its index. If we don't know the index in which the book is we would need to compare the books key, its ISBN with every book in the ArrayList, using O(n) time in the worst case.

- Double Linked List: we could use it to get the attention order or to save books in the shelves. P.S : when we are searching in it it could be O(n) and when we add or delete we have to rename the root, last of the main class, and for the object we have to modify next and last.

- Priority Queue: we could use it to get the attention order of clients when they are arriving at the cash register, the client is going to obtain his or her score by the arriving order, if this score can be modified by another factor, heap will give us the opportunity to do it with very little changes. P.S : every time we add or delete a client depending if the structure is a max-heap or a min-heap we have to run the methods heapy fire up and down to organize the next correct root.

- Threads: we could use threads to implement the cash register when each of them will be a thread that attend the next client in the attendance queue, so when the client end his or her payment the cash register call the next client to be serve.

Phase 4: Transition to preliminary design

Since an ArrayList to store the list of books would cause trouble if the book's index is not known, we have come to the conclusion that a Hash Table would be the better option in this case, considering also that each book has its own key, the ISBN, and it is through its key that we will search for the book.

We have come to the conclusion that a heap will be a much better option to model the scenario of the clients in line than a queue since we need to readjust the priority of the

clients based on the order in which they came into the store and the time they took picking the books.

Therefore, we will be using:

- A Hash table to store the list of books available in the store
- A Stack to store the books the client picks
- A Stack to store the books after they are registered by the cashier
- A PriorityQueue to model the line of clients waiting to pay
- A Thread for each cashier

We suspect that our program may not be consistent due to the use of threads for the cashiers. After a lot of thinking, we decided to stick to this solution since it is the best one we could come up with. We have noticed that there are some errors in the sample output given on the Lab's enunciation, hence we have not been able to fully compare our results with those of the enunciation. However, we believe our program solves the problem successfully.

Phase 5: Evaluating and selecting the best solution

Criteria:

- Criterion A: satisfactory solution of the problem
  [1] The method does not solve the problem satisfactorily
  [2] The method partially solves the problem
  [3] The method solves the problem satisfactorily

- Criterion B: complexity of the implementation of the data structure
  [1] Very complex
  [2] Complex
  [3] Not very complex
  [4] Not complex

- Criterion C: complexity of the understanding of the data structure
  [1] Very complex
  [2] Complex
  [3] Not very complex
  [4] Not complex

- Criterion D: time efficiency
  [1] Not efficient
  [2] Not very efficient
  [3] Efficient
  [4] Very efficient

- Criterion E: space efficiency
  [1] Not efficient
  [2] Not very efficient
  [3] Efficient
  [4] Very efficient

Algorithm	Criterion A	Criterion B	Criterion C	Criterion D	Criterion E	Total
Stack	3	3	4	4	3	17
Queue	3	3	4	4	3	17
Hash Table	4	3	4	4	2	17
ArrayList	3	4	4	2	3	16
Double Linked List	3	3	4	1	3	14
PriorityQueue	3	3	3	4	3	16

In terms of selecting the best data structure to store the list of books, we will use a Hash table since we will need to search the books by their ISBN and not by the index they have in the list. Therefore, an Arraylist would be troublesome when searching for a book.

Queue vs PriorityQueue: we decided that a min priority queue with a min heap would be more accurate in our case since we will need to adjust the priority of the objects in the queue, hence we will use a priority queue.

In summary, the data structures we will be using are:

- Stack

- PriorityQueue
- Hash Table

Phase 6:

ADTs:

MinHeap		
$Array\ a : [a_1,\ a_2,\ a_3, ...,\ a_n]$, where $a_i \rightarrow$ father $a_{2i} \rightarrow$ left child $a_{2i+1} \rightarrow$ right child a_i can be any object therefore it shall be referred to as type T or a generic value.		
$a_i <= a_{2i}$ && $a_i <= a_{2i+1}$		
MinHeap	int	→MinHeap
MinHeap	int x T[]	→MinHeap
Parent	MinHeap x int	→int
LeftChild	MinHeap x int	→int
RightChild	MinHeap x int	→int
GetSize	MinHeap	→int
GetNode	MinHeap x int	→T
Exchange	MinHeap x int x int	→MinHeap
MinHeapify	MinHeap x int	→MinHeap
BuildMinHeap	MinHeap	→MinHeap

Insert	MinHeap x T	→MinHeap
Set	MinHeap x int x T	→MinHeap
Remove	MinHeap x int	→T

Stack		
$ArrayList\ a : [a_1,\ a_2,\ a_3, ...,\ a_n]$, where $a_n \rightarrow$ top From a_{n-1} to $a_0 \rightarrow$Stack.		
Let b be an element $a.push(b) \rightarrow a[n+1] = b$ $a.peek() = b \rightarrow a[0] = b$ $a.pop() = b \rightarrow a[0] = a_1$		
Stack		→Stack
GetStack	Stack	→ArrayList
SetStack	Stack x ArrayList	→Stack
Peek	Stack	→T
Pop	Stack	→T
Push	Stack x T	→Stack
Search	Stack x T	→int
IsEmpty	Stack	→boolean

MinPriorityQueue
$Array\ a : [a_1,\ a_2,\ a_3, ...,\ a_n]$, Where a is a minHeap $a_1 \rightarrow minimum$
$a_i <= a_{2i}$ && $a_i <= a_{2i+1}$

MinPriorityQueue	int	→MinPriorityQueue
IsEmpty	MinPriorityQueue	→boolean
Peek	MinPriorityQueue	→T
Poll	MinPriorityQueue	→T
Offer	MinPriorityQueue x T	→MinPriorityQueue

HashTable		
$Array\ a : [a_1,\ a_2,\ a_3, ...,\ a_n]$, Where a is an int array		
T is an object that can implement hashCode		
HashTable		→HashTable
IsEmpty	HashTable	→boolean
Get	HashTable	→T
Remove	HashTable	→T
Put	HashTable	→HashTable

Class Diagram: the class diagram can be found in a folder in the repository. We decided not to paste it here because it is very large and it requires zooming to be read clearly

Test Cases:

For the Hash Table:

Test Number	1
Method	put(), from class GenericHashTable
Input	The object's key, which is the book's isbn in this case, and the object itself, which is the book
Output	The object is inserted in the hashtable

TestCase	1.1
Objective	Test the put method on an empty hashTable
Scene	An empty hashTable is initialized
Input	new Book("12",1234,2);
Output	The assertNull returns false

TestCase	1.2
Objective	Test the put method on a non-empty hashTable
Scene	An empty hashTable is initialized and some books are added
Input	new Book("45",23423,2);
Output	The assertNull returns false

Test Number	2
Method	get(), from the GenericHashTable class
Input	The key of a book in the hash table

Output	The book with that key

TestCase	2.1
Objective	Test if the hash table properly returns the desired book given the key of such book using the method get()
Scene	An empty hashTable is initialized
Input	new Book("12",1234,2);
Output	assertEquals(book, hashTable.get(book.getIsbn())); it returns true

Test Number	3
Method	isEmpty(), in the GenericHashTable
Input	
Output	The method will return true if the hash table is empty, or false if it isn't

TestCase	3.1
Objective	Test the isEmpty method on an emtpy hash table
Scene	An empty hashTable is initialized
Input	
Output	Asserttrue returns true when called on the return of the isEmpty method

TestCase	3.2
Objective	Test the isEmpty method on a non-emtpy hash table
Scene	An empty hashTable is initialized and some books are added

Input	
Output	The assertFalse return true when called on the return of the isEmpty method

For the Heap:

TestCase	1.1
Objective	Test the creation of an empty MinHeap
Scene	An empty MinHeap is initialized
Input	Int maxsize = 5
Output	The assertEquals comparing the size of the MinHeap to the integer 0, returning true.

Test Number	1
Method	GenericMinHeap(int), in GenericMinHeap
Input	5
Output	The method will return true if the minHeap is empty, or false if it isn't

TestCase	1.2
Objective	Test the creation of an empty MinHeap and insertion of Clients
Scene	An empty MinHeap is initialized and clients are added
Input	Maxsize = 5 Client c1 = new Client("1", 5); Client c2 = new Client("2", 4); Client c3 = new Client("3", 6);
Output	The assertEquals comparing the size of the MinHeap to the integer 3, returning true.

Test Number	2
Method	GenericMinHeap(int), in GenericMinHeap
Input	5
Output	The method will return true if the minHeap is of the size wanted (3), or false if it isn't

TestCase	2
Objective	Test the creation of a MinHeap from an existing array
	A MinHeap is initialized
Input	Int maxsize, array[]
Output	The assertEquals comparing the size of the MinHeap to the integer 3, returning true.

Test Number	3
Method	BuildMinHeap(), in GenericMinHeap
Input	
Output	The method will return true if the minHeap is empty is properly organized, comparing each position as they were created by the Heap and the expected values. Returns true if the structure of the Heap is correct.

Test Number	4
Method	GenericMinHeap(int, arra[]), in the GenericMinHeap
Input	Int maxsize = 5 Client[] cArray = {c1, c2, c3}
Output	The method will return true if the minHeap is of the correct size, or false if it isn't

Test Number	5
Method	insert(T), in GenericMinHeap
Input	Client c = new Client("4", 1);
Output	The method will return true if the new client is inserted into the correct spot (top one), maintaining the heap property, if not then false.

Test Number	6
Method	insert(T), in GenericMinHeap
Input	Client c = new Client("4", 8);
Output	The method will return true if the new client is inserted into the correct spot (last one), maintaining the heap property, if not then false.

Test Number	7
Method	remove(in), in GenericMinHeap
Input	Client c = new Client("4", 8);
Output	The method will return true if the object removed from the expected position is equals to that which was inserted maintaining the heap property, if not then false.

Test Number	7
Method	remove(in), in GenericMinHeap
Input	Client c = new Client("4", 1);

Output	The method will return true if the object removed from the expected position is equals to that which was inserted maintaining the heap property, if not then false.

For the MinPriorityQueue:

Tests for the Queue are quite scarce for its functionality depends fully on the proper implementation of a MinHeap, therefore, each method shall be tested less rigorously than on the Heap. We can expect if it implements one case using the Heap correctly and the MinHeap was previously tested, the methods should work with no issue.

TestCase	1.1
Objective	Test the creation of an empty MinPriorityQueue
Scene	An empty MinPriorityQueue is initialized
Input	Int maxsize = 5
Output	The assertEquals comparing the size of the PriorityQueue to the integer 0, returning true.

Test Number	1
Method	GenericMinPriorityQueue(int), in GenericMinPriorityQueue
Input	5
Output	The method will return true if the queue was created and is empty, otherwise false.

TestCase	1.2
Objective	Test the creation of an empty MinPriorityQueue and further insertion of Clients
Scene	An empty MinPriorityQueue is initialized then clients are added

Input	Int maxsize = 5 Client c1 = new Client("1", 1); Client c2 = new Client("2", 2); Client c3 = new Client("3", 3);
Output	The assertEquals comparing the size of the PriorityQueue to the integer 3, returning true.

Test Number	2
Method	offer(T), in GenericMinPriorityQueue
Input	c1, c2, c3
Output	The method will return true if the queue was created and isn't empty, otherwise false.

Test Number	3
Method	insert(T), in GenericMinPriorityQueue
Input	c1, c2, c3
Output	The method will return true if the queue was created, isn't empty, and the order of the elements is correct, otherwise false.

Test Number	4
Method	insert(T), in GenericMinPriorityQueue
Input	Client c = new Client("4", 4);
Output	The method will poll continuously until the last object previously inserted is supposed to be poll, if this is consistent, the queue property is valid and returns true, otherwise false.