

Engineering Method

1. Identification of the problem: A new bookstore needs to organize the purchase method, the idea is that the clients can purchase and search different books more easily. To achieve this the books will be organized making access to them more faster and efficient. Also the time to purchase will be the most efficient possible, the queue should take a little minutes.

2. The collection of the necessary information:

- The list of books is ordered according to the location of the shelves
- A basket is assigned to place the books that the client finds, one on top of the other, following the order supplied in the previous stage.
- The time in which it has entered the store, plus what has been taken collecting the books from each shelf is the order factor with which the row of boxes is entered.
- The strategy in the box section is:
Customers make a single queue even if there are several ATMs or service points.
When one or more service points are available, as many clients as there are available points, to be attended one by one in each of them.
- Once the client enters the bookstore he goes to section #1 of digital catalogs and successively goes to another 3 sections.
- Each client takes 1 unit of time to collect a book, therefore, the total time when leaving section 3 will be the previous value from section 2 plus the amount taken to collect the copies
- If two users take the same time, the one that was before the exit of the previous section comes out first.
- The order in which the customers' books are packed is given taking into account the pile that the cashier has at his side. Whoever is at the top will obviously be the first to pack up and so on.

After having the information from the problem, we need the information from the complexity analysis for the different ordering methods to be able to choose 3 that are beneficial at the time of execution and time:

Source:

<https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/>

Bubble (Bubble Sort): Complexity $O(n^2)$

This algorithm performs the ordering or reordering of a list a of n values, in this case of n terms numbered from 0 to $n-1$; It consists of two nested loops, one with the index i , which gives a smaller size to the path of the bubble in the reverse direction of $2n$, and a second loop with the index j , with a path from 0 to n_i , for each iteration of the first loop, indicating the location of the bubble.

The bubble is two terms in the list in a row, j and $j + 1$, which are compared: if the first is greater than the second, their values are exchanged.

This comparison is repeated in the center of the two loops, resulting in an ordered list. It can be seen that the number of repetitions only depends on n and not on the order of the terms, that is, if we pass an already ordered list to the algorithm, it will perform all the comparisons exactly the same as for an unordered list.

Counting (Counting Sort): Complexity $O(n + k)$

It is a stable algorithm whose computational complexity is $O(n + k)$, where n is the number of elements to be ordered and k is the size of the auxiliary vector (maximum - minimum).

The efficiency of the algorithm is independent of how nearly ordered it was previously. In other words, there is no best and worst case, all cases are treated the same. The counting algorithm is not ordered in situ, but requires additional memory.

Heaps (Heapsort): Complexity $O(n \log n)$

This algorithm consists of storing all the elements of the vector to be ordered in a heap (heap), and then extracting the node that remains as the root node of the heap (top) in successive iterations obtaining the ordered set. It bases its operation on a property of mounds, whereby the top always contains the smallest element (or the largest, depending on how the heap has been defined) of all those stored in it. The algorithm, after each extraction, relocates the last leaf to the right of the last level at the root or top node. Which destroys the heap property of the tree. But, then it carries out a process of "lowering" the number inserted so that the older of its two children is chosen at each movement, with whom it is exchanged. This exchange, carried out successively, "sinks" the node into the tree, restoring the tree's heap property and giving way to the next extraction of the root node.

Insertion (Insertion Order): Complexity $O(n^2)$

Initially you have only one element, which is obviously an ordered set. Then, when there are k elements ordered from least to greatest, the element $k + 1$ is taken and compared with all the elements already ordered, stopping when a minor element (all major elements have been shifted one position to the right) or

when no elements are found anymore (all elements were shifted and this is the smallest). At this point the element $\{k + 1\}$ is inserted and the other elements must be moved.

Mixtures (Merge Sort): Complexity $O(n \log n)$.

If the length of the list is 0 or 1, then it is already sorted. In another case:

Divide the unordered list into two sublists about half the size.

Sort each sublist recursively applying the sort by mix.

Mix the two sublists into a single ordered list.

Merge sort incorporates two main ideas to improve your runtime:

A small list will take fewer steps to sort than a large list.

It takes fewer steps to build an ordered list from two ordered lists than from two unordered lists. For example, each list only needs to be interleaved once they are sorted.

Fast (Quicksort): Complexity $O(n \log n)$.

Choose an element from the set of elements to be ordered, which we will call a pivot.

Relocate the other elements of the list on each side of the pivot, so that on one side are all those less than it, and on the other the greater ones. The elements equal to the pivot can be placed both to its right and to its left, depending on the desired implementation. At this point, the pivot occupies exactly its rightful place in the ordered list.

The list is separated into two sub-lists, one formed by the elements to the left of the pivot, and the other by the elements to its right.

Repeat this process recursively for each sublist as long as they contain more than one element. Once this process is finished, all the elements will be in order.

As can be assumed, the efficiency of the algorithm depends on the position in which the chosen pivot ends.

In the best case, the pivot ends up in the center of the list, dividing it into two sublists of equal size. In this case, the order of complexity of the algorithm is $\Omega(n \cdot \log n)$.

In the worst case, the pivot ends up at one end of the list. The order of complexity of the algorithm is then $O(n^2)$. The worst case will depend on the implementation of the algorithm, although it usually occurs in lists that are ordered, or almost ordered. But it mainly depends on the pivot, if for example the implemented algorithm always takes the first element of the array as a pivot, and the array that we pass to it is ordered, it will always generate an empty array on its left, which is inefficient.

Selection (Selection order): Complexity $O(n^2)$.

Find the smallest item in the list

Trade it with the first

Find the next minimum in the rest of the list

Swap it with the second

And in general:

Find the minimum element between an i position and the end of the list

Swap the minimum with the element at position

Source:

<https://medium.com/techwomenc/estructuras-de-datos-a29062de5483>

LinkedList:

A list is that structure that represents a countable number of ordered values where the same value can be repeated and considered a different value from an existing one. The main characteristics of a list would be:

Each node in our list contains one or more fields that contain the value we want to store.

Each node contains at least one pointer field pointing to another node, which means that we can have 2 pointers, one that points to a consequent node and another that points to a previous node, thus forming a double linked list.

It is necessary to have a pointer that points to the head of the structure in order to know where to start.

Stack:

A stack supports the ordered recovery of data last-in, first-out (LIFO) or: the last data to enter, the first data to leave. If we use a stack it is because probably the order of data retrieval does not matter so much to us, we simply want to stack and unstack them, so the fundamental operations in a stack are push and pop to put and get data from the stack.

Queue:

A queue or queue supports the orderly retrieval of data first-in, first-out (FIFO) or: the first data to enter is the first data to leave.

In this type of structure, order does matter, since the first in the queue must always be the first to be attended and the rest wait their turn in an orderly manner. The operations of this structure are: enqueue and dequeue to queue and un-queue (put and get from the queue).

HashTable:

A Java Hashtable is a data structure that uses a hash function to identify data by means of a key or key . The hash function transforms a key to an index value of an array of elements.

3.The search for creative solutions.

- Implement a queue when organizing each client's basket of books, in addition to implementing a stack to order the clients' queue when making payments and a HashTable to organize the shelves where the books will go

For the ordering methods of the list of books we will use the following algorithms:

- Bubble sort
- Quick sort
- Heap sort

- Implement the three data structures as follows:

HashTable for the different shelves, where books will be kept and in case they find collisions, they are solved with direct addressing, forming a linked list. Implement queues for customers when paying for their purchases, in addition to stacks when each customer picks up the book they are looking for

For the ordering methods of the list of books we will use the following algorithms:

- Bubble sort
- Merge sort
- Heap sort

- Implement queues for customers when paying for their purchases, in addition to stacks when each customer picks up the book they are looking for, and finally and for the shelves use HashTable with open addressing to fix collisions

For the ordering methods of the list of books we will use the following algorithms:

- Selection sort
- Merge sort
- Quick sort

- Implement the three data structures as follows:

HashTable for the different shelves, where books will be kept and in case they find collisions, they are solved with direct addressing, forming a linked list. Implement queues for customers when paying for their purchases, in addition to stacks when each customer picks up the book they are looking for

For the ordering methods of the list of books we will use the following algorithms:

- Selection sort
- Merge sort
- Heap sort

- Implement a binary tree to organize the basket of books for each client, also implement a linked list to order the queue of customers when making payments and a stack to order the shelf

For the ordering methods of the list of books we will use the following algorithms:

- Insertion sort
- Counting sort
- Quick sort

Implementing the basket of books as trees could, however, we can only access the last added element (LIFO) so the most suitable for this basket of books would be the stack structure, as well as a list for customers can reach be inefficient since we do not need to access each client, but we must access the first client to enter the queue (FIFO), which is why the queue structure is more suitable for this function. As for the shelves, as we are going to save books by their code, the implementation of the hashtable helps us, since we can place the books according to their code in an array and solve the collisions with a link between the books.

Therefore, the ideal structures are the following:

STACK = Basket of books

HashTable = Bookshelf

Queue = Queue of clients.

The other options are discarded, because although they may well solve the problem, they are not the most appropriate.

Sorting Algorithms:

- Bubble sort
- Merge sort
- Heap sort

Regarding the ordering algorithms, we decided to use these algorithms to organize the books, since in them we have efficient algorithms such as mergeSort and HeapSort () both $O(n \lg n)$. The bubble was also implemented. It is a good algorithm that works very well with this book entry, because generally the book entry will not be excessively large, the order of the bubble is $O(n^2)$.

4.Moving from ideas to preliminary designs:

We consider that solutions 2 and 3 are the ones that most closely approximate the central idea of the problem and those that meet the initially proposed requirements

Criteria:

Complexity:

Each design and the methods that come with it present an appropriate difficulty for the development of it.

1. Difficult
2. Medium
3. Easy

Solution precision:

The solution is viable and meets the effective development of all the requirements of the problem

1. Imprecise
2. Almost Accurate
3. Accurate

Effectiveness:

Regardless of the complexity, the solution complies with a perfect effectiveness lacking any logical error

1. No Effective
2. Effective Medium
3. Effective

Usability:

The solution can be used because it meets all the proposed objectives

1. Not usable at all
2. Complex
3. Usable

Evaluating the previous criteria in the alternatives that remain, we obtain the following table:

	Complexity	Solution precision	Effectiveness	Usability
--	-------------------	-------------------------------	----------------------	------------------

Alternative 1	3	3	3	3
Alternative 2	3	2	2	3

Alternative 1: 12 points

Alternative 2: 10 points

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

5.The evaluation and selection of the preferred solution:

- Because several data structures need to be implemented for the different problems and as mentioned above, queues for customers, stacks for book storage and hash tables for bookshelves will be implemented. in order to have fast and efficient data structures as well as the ordering of the structures.

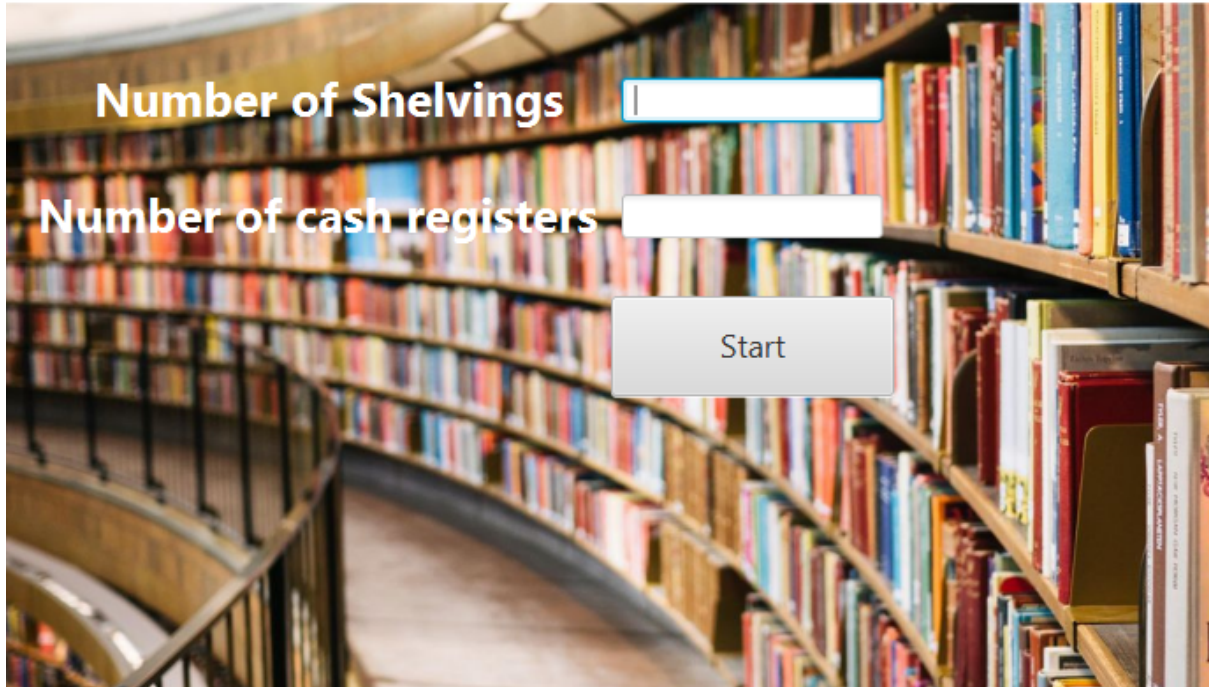
6.Design implementation:

img #1(section 1):

Screen of load(section 1) where is entered number of shelvings and cash registers



NATIONAL LIBRARY

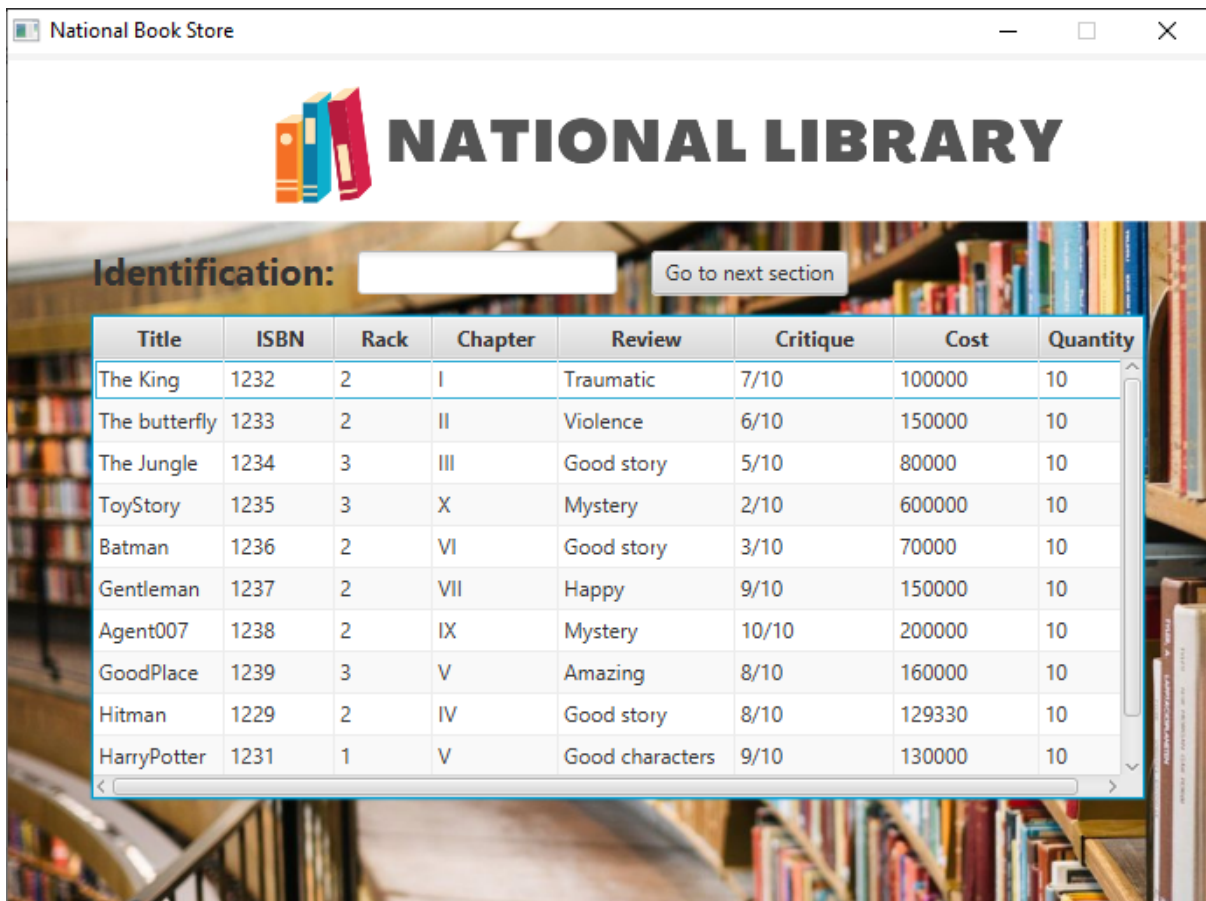


Number of Shelvings

Number of cash registers

img #2:

Screen of books list and info and enter a number of client identification.



img #3(section 2):

Screen to search and add to client basket books(section 2) also the list of books searched will be sorted by merge, heap or bubble and add other clients.



img #4(section 3):

This Screen shows the arrival order of clients(section 3) to stacks to pay in cash registers



NATIONAL LIBRARY

Identification	Quantity Of Books
123	0
123	0
123	0
123	0

Identification	ISBN bought books	Price
No content in table		

Finish and exit

Functional Requirements

1. Must allow adding establishment
2. Model section 1 where are the digital catalogs, available titles present within the store, initial chapters, reviews, reviews of consulted books. etc.
3. Create the ISBN code list for the item the client wants
4. Should be able to very fast searches for books by their ISBN code(in section 2, the search gives the shelving of the book).
5. Order the list of books according to the location of the shelves.
6. Collect the books searched and place them in a basket one on top of the other.
7. Make the payment once the basket is full..
8. Take the time since the client enters the store, picks up the books and leaves the establishment.
9. Basic information will be requested such as: book catalog (ISBN of the book, number of copies, shelf where it is located), number of ATMs to be used during the day, series of codes or cards representing customers (in the order in which they entered the store).
10. Displayed the clients departure order
11. the value of each purchase will be displayed.
12. The order in which your books were packed will be displayed.

Functional No-requirements

1. Use 3 different sorting algorithms. Only one of these algorithms can have average time complexity of (n^2)
2. The program will have a graphical interface.
3. Use data structures (queues, stacks, hash tables).