## **Engineering Method on Toy Store System**

By: Juan David Calderón, Juan Jose Vidarte and Juan Pablo Bello

**Phase 1: Elicitation of the Problem**

First, in this project, we need to understand the client's needs. This problem is about a toy store that requires a system to automate its orders and record information. This process is crucial because, nowadays, purchasing methods have evolved, with new branches of sales emerging. Therefore, for a toy store, where the main reason for purchasing is often for gifts, it is important to provide a complete and efficient user experience to prevent customers from turning to competitors.

The first requirement we identified is customer registration. It's essential to have accurate and sufficient information about the user to facilitate effective purchases and provide a record in case of potential complaints. On the other hand, regarding the product registration by the administrator, it is important to establish a system that helps keep track of inventory and organize unit sales for each product. This makes it easier to filter and display product types to customers, different from how it would be done in a physical store. Moreover, allowing customers to view available products online helps them know if they can purchase these items in person at the store. In this product registration, it's notable how prices can be constantly updated due to the flexibility of changing information in the system, such as adjusting prices based on the time of year, discounts, and other factors. Additionally, the system allows for the removal of products that are out of stock or no longer sold, providing a detailed filter of items that are actually moving in the store.

Finally, one of the crucial parts is selling the products. For this to work, the previous sections must be complete, as orders cannot exist without customers and products. Thus, the orders will be linked to the customers. In this case, the client requested that the system be organized according to the order of requests. If an order is made for a user, that user will receive all the orders according to a predetermined priority, with the option to prioritize which orders they want to receive first. Therefore, the organization will first be by user order and then by priority of all that user's orders. This approach ensures efficient dispatch of orders to customers and better encapsulation of the toy store's orders.

## **Study Case:**

|  |  |
| --- | --- |
| Client | **The toy store** |
| User | **Customers**, the ones who would make the purchases on the online store, **Operators** are the ones that are responsible for preparing and dispatching orders and the management of products and users. |
| Functional Requirements | *FR1 = Register Customer*  *FR2 - Register Products*  *FR3 = Set products*  *FR4 = Eliminate Products*  *FR5 = Undo Last Action*  *FR6 = Register Orders*  *FR7 = Order preparation* |
| Context | *The small toy store in the city, because the last result on selling of the product shows the necessity for an automatization of the store. This by contacting the* |
| Non Functional Requirements | *FRN1 = Order Management Users*  *FRN2 = Order Management Products of User* |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID and Name | *[RF1-Register customers]* | | | |
| Summary | *[the costumer must be able to register their information, adding the customer's name and their respective data. ]* | | | |
| Input | **Nombre entrada** | **Tipo de dato** | | **Condición valores válidos** |
| name | String | | *[String with the name of the costumer to be registered]* |
| address | String | | *[String with the address to be registered]* |
| email | String | | *[String with the email to be registered]* |
| password | String | | [String with the password to be registered] |
| Post condition | Customer is registered successfully or the information could not be registered. | | | |
| Output | **Nombre salida** | | **Tipo de dato** | **Formato** |
| SuccesfullRegistration | | String | *[String text with the message:*  *“Your customer information was successfully added.”]* |
| UnsuccesfullRegistration | | String | [String text with the message:  “Sorry, there was a mistake and your information could not be registered”] |
|  | |  |  |
|  | |  |  |
|  | |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Id and Name | *RF2- Register Products* | | | | |
| Summary | *[This operation is for the operation to register new products in the system. This information of the Product, that has a code, name, description and price. Finally, this information has to be saved.]* | | | | |
| Input | **Nombre entrada** | **Tipo de dato** | | **Condición valores válidos** | |
| idProduct | String | | *“This value has to generate a code, at least one String, must be unique .*  *ProductXXXX ”* | |
| nameProduct | String | | “This String it has to have at least one char or it's going to be null” | |
| Description | String | | “This String it has to have at least one char or it's going to be null” | |
| price | double | | “The prices it has to be number, if it's a different type of value it would return null ”  “Must be the value greater than zero” | |
| Result | The product is added to the system’s product catalog. In case of duplicate product codes, an error message is displayed. | | | | |
| Output | **Nombre salida** | | **Tipo de dato** | | **Formato** |
| SuccessRegistrationProduct | | String | | *[String text with the message:*  *“Your Product information was successfully added.”]* |
| UnsuccessfulRegistration | | String | | [String text with the message:  “Sorry, there was a mistake and your information could not be registered”] |
| Product | | Object | | “NEW PRODUCT” |
|  | |  | |  |
|  | |  | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Identificador y nombre | *[RF3-Set products]* | | | |
| Resumen | *[the administrator must be able to change a product’s information, making it so that the last information is erased and replaced by the new one. ]* | | | |
| Entradas | **Nombre entrada** | **Tipo de dato** | | **Condición valores válidos** |
| nameProduct | String | | *[String with the name of the product to be modified]* |
| newProductName | String | | *[String with the new name to be modified]* |
| newDescription | String | | *[String with the new descriptionl to be modified]* |
| newPrice | double | | [String with the new price to be modified] |
| Resultado o Postcondición | The chosen product’s information is successfully modified, erasing its old related data and replacing it by the new one. | | | |
| Salidas | **Nombre salida** | | **Tipo de dato** | **Formato** |
| SuccesfullModification | | String | *[String text with the message:*  *“The product’s information was successfully modified”]* |
| UnsuccesfullModification | | String | [String text with the message:  “Sorry, there was a mistake and the product’s information could not be modified”] |
|  | |  |  |
|  | |  |  |
|  | |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Identificador y nombre | *RF4- Eliminate Products* | | | | |
| Resumen | *[This is the method that allows the operators to delete a product from the catalog. Once it is deleted, the product will no longer appear in the catalog for customers. The condition for searching for the elimination of a product is the idProduct.]* | | | | |
| Entradas | **Nombre entrada** | **Tipo de dato** | | **Condición valores válidos** | |
| idProduct | String | | *“This product code must exist or it would return a null ”* | |
| Resultado o Postcondición | The product is successfully removed from the system. If the product code does not exist, an error message is shown. | | | | |
| Salidas | **Nombre salida** | | **Tipo de dato** | | **Formato** |
| SuccessRemovingOfProduct | | String | | *[String text with the message:*  *“Your Product has being removed from the Catalog.”]* |
| UnsuccessfulRemovingOfProduct | | String | | [String text with the message:  “Sorry, there was a mistake, you code must be typed erroning or the product must not exist”] |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Identificador y nombre | *[RF5-undo last action]* | | | | |
| Resumen | *[the administrator must be able to undo the last action performed by the system (either register product, register client, or create order), utilizing this feature will erase the last action performed by the user by taking out the last item of a pile data structure. ]* | | | | |
| Entradas | **Nombre entrada** |  | **Tipo de dato** | | **Condición valores válidos** |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
| Resultado o Postcondición | The last action performed by the user is deleted, in case of it being by exemple, an order creation, all of the data related to that created order will be deleted. | | | | |
| Salidas | **Nombre salida** | | | **Tipo de dato** | **Formato** |
|  | | |  |  |
|  | | |  |  |
|  | | |  |  |
|  | | |  |  |
|  | | |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Identificador y nombre | *RF6- Register Order* | | | | |
| Resumen | *[This operation is for the operation to register new orders on the system. This information of the Order is : numberOfOrder, Customer, Date and Total of order. Finally, this information has to be saved.]* | | | | |
| Entradas | **Nombre entrada** | **Tipo de dato** | | **Condición valores válidos** | |
| List <Product> | Product | | *“The order must take correct products for the order, if the product doesn't exist it would return null.”* | |
| Customer | Customer | | “The object of the Customer must be registered for assign an order” | |
| date | Date | | “The value of the date it has to be on this format *dd-mm-yyyy*” | |
| Resultado o Postcondición | The order is successfully registered, an order number is generated, and the total and date are recorded. | | | | |
| Salidas | **Nombre salida** | | **Tipo de dato** | | **Formato** |
| SuccessRegistrationOrder | | String | | *[String text with the message:*  *“Your Order information was successfully added.”]* |
| UnsuccessfulOrder | | String | | [String text with the message:  “Sorry, there was a mistake and you order information couldnt be registered ”] |
| idNumberOrder | | String | | “This must not be empty”  “The id number for the order is” + idNumberOrder |
| totalOrder | | double | | “The numerical value has to be > 0”  “The total value for the order is” + totalOrder |
| Order | | Order | | NEW ORDER |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Identificador y nombre | *[RF7-order preparation]* | | | |
| Resumen | *[the administrator must be able to prepare the orders meant to be sent, following a set of established parameters. ]* | | | |
| Entradas | **Nombre entrada** | **Tipo de dato** | | **Condición valores válidos** |
|  |  | |  |
|  |  | |  |
|  |  | |  |
|  |  | |  |
| Resultado o Postcondición | The chosen product’s information is successfully modified, erasing its old related data and replacing it by the new one. | | | |
| Salidas | **Nombre salida** | | **Tipo de dato** | **Formato** |
| orderOfShipments | | String | *[String text with the order of the products to be shipped.]* |
| productPriorityByUser | | String | [String text with the priority of products of each user”] |
|  | |  |  |
|  | |  |  |
|  | |  |  |

**Phase 2:**

The following document has the objective of informing the technical aspects of our project, to properly administrate a toy store, we must utilize different complex algorithms and data structures. Managing the product inventory, the orders and delivery logic, managing the priority of orders, and more must be done by creating and executing algorithms and storing information in different types of data structures. This is because this way well be able to efficiently create our project. Below we list all the information we collected to create this project:

## **Generics**

Definition:

Generic programming is a style of programming in which algorithms are written in generic terms, to then be instantiated when needed for specific types of parameters. It makes it so that you can reutilize a particular part of code and make programming more efficient.

What we’re using generics for:

-all FR

Generics will be used to create different interfaces and classes that will put out generic methods, this way we’ll be able to call these methods with different types of data, letting us implement different methods that will let us operate the logic behind our system.

## Hashtable

Definition:

A diagram of a key value pair

Description automatically generated

A hash table serves multiple purposes, it generalizes the simpler notion of an ordinary array, it allows us to store keys and values that are linked to each other, being useful for example to implement dictionaries, there are multiple concepts important to know about hash tables:

-Direct addressing, it is simple it just lets you access information directly, to access information that is in the 50th slot, you just put the key directly to that slot.

-hashing,

-hash collisions: when two or more keys guide you to the same position it is called a hash collision.

-chaining, it is used to not encounter hash collisions, it guarantees the security of data because it works in a way where it will try to store the data in the next slot that’s available.

What we’re using hashtable for:

-FR1

We're using this data structure to store and quickly find information about our customers, products, and orders. Think of it as an incredibly efficient filing system that lets us access the data we need in a snap, which is super helpful for example when trying to look up information about a costumer.

Customers: We keep basic information: names, addresses, emails, and passwords. This way, we can quickly get to a customer's information when they log in or want to update their details.

Products: Our hash table holds product information such as product codes, names, descriptions, and prices. This makes it easy for us to find any product quickly when needed.

Orders: We store order details like the order number, the customer who placed it, the date, and the total amount. Using a hash table means we can access this information swiftly when processing orders or checking their status.

**Stack**

Definition:

The pile data structure is a data structure where you can only perform two actions, adding an element to the last slot or taking out the element from the last slot. Imagine a pile of dirty plates in a restaurant, to access one of them you will have to take out every plate that’s above it.

What we’re using pile for:

FR 5-undo last action

The pile data structure will be essential to create the requirement of being able to undo the last action done in the system, by implementing a pile structure the system will be able to eliminate the last slot of data saved in it, making a sort of going back function.

**Queue**

Definition:

The first element entering the queue is the one that gets rewarded first, in our case will be the first one to be attended of the customer requests.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

A queue works by following the “First in, first out” (FIFO) principle, meaning that the first item added is the first one to be removed. Queues are widely used in many algorithms and applications because they make it simple and efficient to manage the flow of data. There is multiple basic operation in queues:

-enqueue: adds an element to the rear of the queue

-dequeue: removes and returns the element from the front of the queue

-empty

-full

What we’re using queue for:

Queues will be used to manage the order of customer requests, ensuring that they are processed in the order they were received. The store may receive multiple order requests from various users, and queues ensure that these events are handled sequentially (FIFO - First In, First Out), making sure orders are processed in the same order in which they were created. This way, operators can dispatch products in a fair and organized manner.

**Priority Queue**

Definition:

A priority queue is like a queue, but in this case, it lets us associate a priority value to each element, letting us insert and eliminate elements in a specific order. The one with the highest priority will always be at the start of the queue.

What we’re using priority queue for:

In addition to handling the order of requests, a priority queue will be implemented to manage the dispatch of products based on priority. Some products may be in higher demand or more important to specific customers (e.g., products with limited stock or urgent deliveries). This structure will allow orders with higher priority to be processed first

Source:

<https://en.wikipedia.org/wiki/Generic_programming>

<https://bitybyte.github.io/Hashtables/>

<https://medium.com/@lupomontero/pilas-stacks-76f3b8a31f61#:~:text=Las%20pilas%20(stacks)%20son%20una%20estructura%20de%20datos%20donde%20tenemos,%C3%BAltimo%20elemento%20de%20la%20pila>

<https://www.geeksforgeeks.org/queue-data-structure>

**Phase 3: Creative Solutions**

Creative Solutions for Develop a product to manage the toy store and solve the problems that it carries

For these creative solutions we will first separate the product in different problems and then we will show alternative solutions to solve the problems to achieve a good software solution

The first problem is the way to save and store the customer products and orders:

Hash Table for managing customers, products, and orders

Alternative 1: Use a standard hash table to manage each entity (customers, products, and orders) separately. This structure allows for quick and efficient searches of customers and products by their unique identifiers, such as email for customers and product codes for products. Orders would be stored in a separate hash table using the order number as the key.

Alternative 2: Implement a hierarchical hash table, where customers act as the primary key, and each customer has an associated subset of hash tables for their orders. This would improve organization and quick access to information, though it might increase the complexity of accessing each level.

For saving the different types of things in those data structures we will create our own hash table with insertion methods to maximize the efficiency of the actions.

The second is the management of the requests:

Queue for managing order requests

Alternative 1: Use a simple FIFO (First In, First Out) queue to manage order requests in the order they are received from customers. Whenever a customer places an order, it is added to the end of the queue. When processing orders, the system would remove and handle the first order in the queue.

Alternative 2: Use a customer-segmented FIFO queue. In this approach, each customer would have their own sub-queue of orders, and the system would manage these queues concurrently. This would optimize order processing by customer, though it might add complexity when handling many customers.

Priority Queue for dispatching orders by importance

Alternative 1: Use a priority queue based on product price, where higher-priced products have higher priority for dispatch. This could align with the store's business interests, ensuring that more valuable products are dispatched first.

Alternative 2: Create a priority queue based on order creation date. Each order has a priority assigned based on when it was created. Older orders would have higher priority, ensuring that no order is delayed for too long.

Another problem is that the application must have something to revert the last action:

Alternative 1: Implement a stack that stores action objects (e.g., adding a customer, creating a product, creating an order). Each time an action is performed, an object representing that action, along with the necessary data to reverse it, is stored in the stack. The stack allows you to undo the most recent action.

Alternative 2: Create a stack for each type of action. For instance, one stack for customer-related actions, another for products, and another for orders. This could provide more control over undoing actions, as each stack specializes in a type of operation, though managing multiple stacks could increase system complexity.

To solve those problems and unify them in a software we will implement a program in java that could be executed in a terminal, knowing that this solution is preliminary and it’s a model of the real application.

**Step 4: Transforming ideas to preliminar versions**

In this case we will study the process of polishing those early ideas, working on them, and then creating early versions of the solutions.

In the previous step we recapped the problems presented to us and we proposed different solutions to such problems, being the first the way to store and save the costumer products and orders.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Here are the implemented data structures and how we organized them.

We implemented the data structure hashtable, discarding the alternative 2 as it would’ve been unnecessarily complicated.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Here you can see how we structured our hashtable data structure, to associate email for costumers, product codes for products and orders with order numbers, we implemented the hashtable class utilizing generics, where every Key K and every Value V can be invocated when needed.

The interface can be implemented, and the controller controls the logic of the methods utilized to accomplish our goal. We implemented the methods Save Products, SaveCostumer, save order, remove Order, etc…

Next problem was how to manage requests, we utilize a simple FIFO queue, we implemented the class queue, which manages the logic of the queue data structure, as well as the corresponding interface and node class, this will let us manage the order requests and its order, but this would only cover the necessity of organizing the orders in a strict manner.

Continuing from the simple FIFO queue implementation, we realized that managing requests solely based on the First In, First Out (FIFO) principle wasn't enough to handle the various priorities that different orders might have. The system needs to have implemented a system where inside the orders of a single user, products need to have a priority level, depending on this level a user will have his items dispatched in order depending on that priority queue. This would follow the system needs. This led us to consider using a priority queue instead of a regular queue for dispatch management inside of the orders.

**Why a Priority Queue?**

A priority queue allows us to assign a priority level to each order. Orders with higher priority will be processed first, regardless of when they were added to the queue. This differs from a regular FIFO queue, where the order of arrival is the sole determining factor for dispatch.

Using a priority queue means that each order is associated with a priority value, and the dispatch logic always selects the order with the highest priority (or lowest priority, depending on how the priority queue is implemented) to be processed next.

Next problem was registering products and costumers, to register them we implemented various hashtables. Every item was linked to a key in hashtables, facilitating their utilization and elimination when needed. The methods saveCostumer,saveOrder, removeProduct, etc. All worked around the utilization of hashtables, the save Customer method checks if a customer with the given ID already exists in the customer hash table, and if not, it creates a new customer, inserts them into the table, pushes the customer onto the action stack, and stores the action type (2) in the typeActionStack so that the action can be undone later. If the customer ID exists, it throws an exception. The saveOrder method first checks if the customer exists; if they don’t, it throws an exception, and if the order number already exists, it throws another one. Then, it looks up or creates the customer’s queue for their orders, creates a new order, puts it in the queue, and adds it to the hash table that tracks orders by number.

Next problem was reverting the last action. Utilizing the stack data structure, we can create methods that let us revert the last action done by the user, we attempted implementing stacks that stores action object in an undo stack, this stack will store all actions and when the user decides to undo it will pop the last object of the stack.

The possible actions that can be reversed are register client, product, and create order. The solution we proposed is making an object string that reunites all these possible actions and reverts them, is like creating a method to delete the info but much fancier. How it works is that when the object gets deleted from the stack, all its information related to it gets also deleted in the hashtable, which contains all the information. To do this we included the undoLastAction method, which basically lets you undo the last thing you did in the system. It keeps track of your actions using two stacks—one for the objects and one for the types of actions you performed. First, it checks if there's anything to undo; if both stacks are empty, it throws an error. Then, it grabs the last object and action type from the stacks. Depending on what that action was, like adding a product or customer, it’ll remove them. If it was placing an order, it'll take that order out of the customer’s queue.

**Step 5: evaluation and selection of best solutions.**

In the previous step, we encountered several problems, particularly regarding how to store and manage customers, products, and orders effectively in an online toy store. Multiple solutions were proposed and implemented, leveraging various data structures such as hash tables, queues, priority queues, and stacks. Each was selected to address a specific requirement of the system, such as managing order priorities, undoing actions, and registering customers and products.

**Hash Tables for Efficient Storage**

One of the primary issues was how to store and access customers, products, and orders efficiently. The implementation of a hash table provided an optimal solution. By using generics, the hash table can store a wide range of objects—whether it’s customer information keyed by email, product details keyed by product codes, or orders keyed by their numbers. This ensures that data retrieval is constant time, O(1)O(1)O(1), making hash tables ideal for managing these elements efficiently. The methods such as saveCustomer, saveProduct, and saveOrder serve to hash tables for quick storage and retrieval. Additionally, this structure supports the smooth removal of orders, products, and customers when required.

**FIFO Queue for Basic Order Management**

Initially, a basic FIFO queue was implemented to manage the incoming order requests. This simple queue system helped organize orders in a "first come, first served" manner, ensuring that the earliest requests were fulfilled first. While this system was straightforward, it quickly became evident that a strict FIFO model would not suffice for managing requests that have different levels of urgency. Hence, while useful in certain scenarios, such as managing tasks or processes that must adhere to a strict order, the FIFO queue lacks the flexibility needed for the system.

**Priority Queue for Managing Order Dispatches**

As order management became more complex, the next problem to tackle was the varying priorities of different products within a user’s order. In response, a priority queue was integrated into the system. Unlike a FIFO queue, which only considers the time an order was placed, the priority queue allows us to assign priority levels to different products within an order. This is really important for the system, as it enables high-priority items to be dispatched before others regardless of their arrival time.

The priority queue structure supports this by associating each order or product with a priority level. The queue is structured to always process the highest priority item first, which improves efficiency and ensures customer satisfaction by handling orders with highest priority first.

**Stacks for Undoing Actions**

To handle the challenge of reverting the last action performed by a user, the stack data structure was employed. Stacks follow the Last In, First Out (LIFO) principle, making them ideal for an "undo" functionality. The system implements a stack that tracks user actions, allowing for reversal of operations such as customer registration, product addition, or order creation. We made it so that the system recognizes what was the last action performed Each action is pushed onto the stack as it occurs, and the undoLastAction method pops the most recent action when needed, removing the associated item from the hash table or order queue.

**Conclusion: The Best Solution**

Among the proposed solutions, the combination of hash tables, priority queues, and stacks emerges as the most effective approach for managing the system's complexity. Hash tables offer fast lookups, while priority queues ensure flexible order management based on urgency.

**Phase 6:**

The Diagram class is situated on the Documentation in a pdf and Vpp documents.

**Temporal-Spatial Complexity Analysis:**

**Temporal Complexity**

The temporal complexity analysis is based on the number of operations executed as a function of the input size. In this case, the input is mainly related to searches, insertions, and deletions in the hash table. Most operations in hash tables (search, delete, insert, update) have an expected cost of O(1).

**Temporal Complexity for Method: modifyProduct**

public void modifyProduct(String newCode, String newName, double newPrice, String newDescription, String code) throws InexistentProductException {

// Line 1: Method declaration (O(1))

Product newProduct = productsH.search(code);

// Line 2: Search in the hash table (O(1) on average, O(n) in the worst case)

if (newProduct == null) {

throw new InexistentProductException(code);

}

// Line 3-5: Condition + throw exception (O(1))

if (newCode != null && !newCode.equals(newProduct.getCode())) {

productsH.delete(code);

// Line 6-7: Delete product from hash table (O(1) on average, O(n) in the worst case)

newProduct.setCode(newCode);

productsH.insert(newCode, newProduct);

// Line 8-9: Set new code + insert product (O(1) on average, O(n) in the worst case)

}

if (newName != null && !newName.isEmpty()) {

newProduct.setName(newName);

}

// Line 10-12: Update name (O(1))

if (newPrice != newProduct.getPrice() && newPrice > 0) {

newProduct.setPrice(newPrice);

}

// Line 13-15: Update price (O(1))

if (newDescription != null && !newDescription.isEmpty()) {

newProduct.setDescription(newDescription);

}

// Line 16-18: Update description (O(1))

if (newCode == null || newCode.equals(code)) {

productsH.set(code, newProduct);

}

// Line 19-21: Reinsert product if code doesn’t change (O(1) on average, O(n) in the worst case)

}

#### Summary of Temporal Complexity:

* **Line 2**: Product search → **O(1)** (average), **O(n)** (worst case).
* **Line 3-5**: Check if the product exists and throw exception → **O(1)**.
* **Line 6-9**: Delete product, change code, reinsert → **O(1)** (average), **O(n)** (worst case).
* **Line 10-12**: Update name → **O(1)**.
* **Line 13-15**: Update price → **O(1)**.
* **Line 16-18**: Update description → **O(1)**.
* **Line 19-21**: Reinsert product if code doesn’t change → **O(1)** (average), **O(n)** (worst case).

**Temporal Complexity:**

Best case: O(1) (most operations like search, insert, delete, and update are constant in hash tables).

Worst case: O(n) (if collisions occur in the hash table).

**Spatial Complexity:**

O(n) where n is the number of products in the hash table. The space doesn’t increase significantly during execution since only references are updated.

**Caso 2 : Temporal-Spatial Complexity for extractMaximum:**

@Override

public T extractMaximum() throws NoElementFoundException {

// Line 1: Check if the heap is empty (O(1))

if (heap.isEmpty()) {

throw new NoElementFoundException("Priority queue is empty.");

}

// Line 2-3: If the heap is empty, throw exception. This is O(1) as it's just a check.

T maxElement = heap.get(0);

// Line 4: Get the maximum element (the first element). This is O(1) since accessing the first element in a list is constant.

T lastElement = heap.remove(heap.size() - 1);

// Line 5: Remove the last element of the heap. This is O(1) for accessing the last element, but it could be O(n) to remove, depending on the list implementation (if it’s an `ArrayList`), as shifting elements takes O(n).

if (!heap.isEmpty()) {

heap.set(0, lastElement);

// Line 6-7: If the heap is not empty, place the last element at the root. This is O(1) as it's a direct assignment.

heapifyDown(0);

// Line 8: Call `heapifyDown(0)`. This operation adjusts the heap from the root down, so it has a complexity of O(log n), as the heap is a binary tree and only needs to traverse the height of the tree, which is logarithmic in terms of the number of elements.

}

// Line 9: Return the maximum element (O(1)).

return maxElement;

}

**Summary of Temporal Complexities:**

Line 1-3: Check if the heap is empty and throw exception → O(1).

Line 4: Get the maximum element (first element of the heap) → O(1).

Line 5: Remove the last element of the heap. This could be O(n) if the heap is implemented as an ArrayList due to shifting elements. If it’s a more efficient structure like LinkedList, it could be O(1).

Line 6-7: Assign the last element to the root of the heap → O(1).

Line 8: heapifyDown(0) has a complexity of O(log n), as it traverses the height of the heap.

Line 9: Return the maximum element → O(1).

**Temporal Complexity**:

* **Best case**: **O(1)** (for small heaps, most operations are constant).
* **Worst case**: **O(n + log n)** (if the heap is implemented as an ArrayList and elements need to be shifted during removal, otherwise just **O(log n)** for the heapifyDown operation).

**Spatial Complexity**:

* **O(1)**, as there are no significant additional memory requirements beyond a few variables.