| Computación y estructuras discretas I |
| --- |
| Engineering Method |
| Ti I |
| David Dulce, Nicolas Cardona, Juan Esteban Eraso  16-10-2023 |

# Step 1. Problem identification

Concrete recognition is given to the specific needs of the problematic situation, as well as its symptoms and the conditions under which it must be resolved.

*Problem Definition*

The problem is to develop a task and reminder management system that allows users to efficiently organize and manage their pending tasks.

*Identification of needs and symptoms*

-The user needs to store tasks and reminders.

-A user interface is required to enable users to add, edit, and delete tasks and reminders. Users can view a list of all tasks and reminders, sorted by deadline or priority.

-Two categories for tasks are required: "Priority" and "Non-priority."

-It is necessary to implement a method that allows undoing user actions in the system.

To accomplish all of this, we need to Filter tasks by priority, Filter tasks by date, Sort tasks by date, Sort tasks by priority, Sort tasks by status.

In summary, the following needs to be allowed:

-Create tasks

-Edit tasks

-Delete tasks

-Show a list of tasks in the order of arrival.

-Show a list of tasks by priority

-Mark tasks as priority

-Undo actions

*Requirements*

| Client | Marlon |
| --- | --- |
| User | Users of the task and reminder management system. |
| Functional requirements | RF0 - Add Task  RF1 - Show Task List  RF1.1 - By Arrival Order  RF1.2 - By Priority Order  RF2 - Modify Task  RF3 - Delete Task  RF4 - Undo Actions |
| Problem context | A system for task and reminder management needs to be designed, allowing users to add, organize, and manage their pending tasks and reminders. These tasks will be stored based on their category: Priority or Non-Priority. Users will be able to view their saved tasks in order of arrival and by priority. Additionally, the system will include an undo function for the most recent action. |
| Non-functional requirements | * Graphical User Interface * Quick Response |

| Identifier and Name | *[RF0-add task]* | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system allows the user to add a task or reminder.* | | | |
| Inputs | **Name** | **Data type** | | **Valid values condition** |
| title | String | |  |
|  | description | String | |  |
|  | dueDate | Date | |  |
|  | priority | int or char | | 1. Non-priority  2. Priority.  2.1 high  2.2 medium  2.3 low |
| Result or Postcondition | The program stores the task or reminder. | | | |
| Outputs | **Output Name** | | **Data type** | **Format** |
| msg | | String | A message is displayed confirming that the task has been added. |

| Identifier and Name | *[RF1-mostrar lista de tareas]* | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system will display a list of all tasks and reminders, sorted by due date or priority.* | | | |
| Inputs | **Name** | **Data Type** | | **Valid values condition** |
|  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
| Result or Postcondition | The program displays the list of all tasks and reminders, sorted by due date or priority. | | | |
| Outputs | **Output Name** | | **Data Type** | **Format** |
|  | |  |  |

| Identifier and Name | *[RF1.1-Show the list of tasks in order of arrival]* | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system displays non-priority tasks to the user in the order they were added.* | | | |
| Inputs | **Name** | **Data Type** | | **Valid values condition** |
|  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
| Result or Postcondition | Display the list of tasks in the order they were added. | | | |
| Outputs | **Name** | | **Data Type** | **Format** |
|  | |  |  |

| Identifier and Name | *[RF1.2-Display the list of tasks in order of priority]* | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system displays the prioritized tasks to the user in order of priority.* | | | |
| Inputs | **Name** | **Date Type** | | **Valid values condition** |
|  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
| Result or Postcondition | Display the list of tasks in order of priority | | | |
| Outputs | **Name** | | **Date Type** | **Format** |
|  | |  |  |

| Identifier and Name | *[RF2-Modify task]* | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system will allow the user to modify a previously added task according to the criteria they choose.* | | | |
| Inputs | **Name** | **Date Type** | | **Valid values condition** |
| title | String | |  |
|  | description | String | |  |
|  | priority | Enum | | 1. Non-priority  2. Priority.  2.1 high  2.2 medium  2.3 low |
|  |  |  | |  |
| Result or Postcondition | The task will be modified. | | | |
| Outputs | **Name** | | **Data Type** | **Format** |
| msg | | String | Message confirming the task modification. |

| Identifier and Name | *[RF3-Delete Task]* | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system allows the user to delete a task.* | | | |
| Inputs | **Name** | **Date Type** | | **Valid values condition** |
| key | int | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
| Result or Postcondition | The task will be deleted.  . | | | |
| Outputs | **Name** | | **Data Type** | **Format** |
| msg | | String | Message confirming that the task has been deleted |

| Identifier and Name | *[RF4-Undo actions]* | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system allows the user to undo actions* | | | |
| Inputs | **Name** | **Data Type** | | **Valid values condition** |
| option | int | |  |
|  |  |  | |  |
|  |  |  | |  |
|  |  |  | |  |
| Result or Postcondition | The most recent action is reversed. | | | |
| Outputs | **Name** | | **Data Type** | **Format** |
| msg | | String | Confirmation message: the undo action was executed |

## 

## **Step 2. Gathering Information**

*Information about what is required*

**To store tasks and reminders**: a hash table is used. The key could be a unique identifier, and the value could be the task/reminder information. Each entry in the hash table could have the following information: title, description, due date, priority, etc.

Users can view a list of all tasks and reminders, sorted by due date or priority. Since the tasks and reminders will be in a hash table, heapsort will be used for sorting.

**Priority Management**: The system will have two task categories: "Priority" and "Non-priority." A priority queue will be used to organize tasks based on their level of importance. Additionally, a specific category will be created for non-priority tasks, allowing these tasks to be managed in the order they were received, following the "first in, first out" (FIFO) principle.

**Undo Method Implementation:** Implement a method that allows users to undo the last action they performed. To undo actions performed by a user in the system, a stack (LIFO) can be used to keep track of the actions carried out.

Recommendation:

First, create a stack to track user actions. Each time the user performs an action, record the action in the stack. Each stack entry should contain information about the action taken and details of the task.

Undo Method: This method should pop the last action from the stack and reverse the corresponding action based on the information stored in the stack.

Using the Undo Method: In your user interface, provide users with the option to undo the last action performed. When the user selects the 'Undo' option, call the undo() method. This will reverse the last action taken.

*Definitions*

**HashTable:** Hash Tables are a data structure that allow you to create a list of paired values. You can then retrieve a certain value by using the key for that value, which you put into the table beforehand. A Hash Table transforms a key into an integer index using a hash function, and the index will decide where to store the key/value pair in memory.It is used to efficiently store and retrieve data, making it very useful when you need to map keys to values and perform efficient searches by key.

HASH TABLE TIME COMPLEXITY IN BIG O NOTATION

Algorithm Average Worst case

Space O(n) O(n)

Search O(1) O(n)

Insert O(1) O(n)

Delete O(1) O(n)

Taken from: <https://www.freecodecamp.org/news/javascript-hash-table-associative-array-hashing-in-js/>

**Hash Function:**A hash function is a mathematical algorithm that transforms input data into a unique code.

Taken From: <https://keepcoding.io/blog/que-es-una-funcion-hash/>

**Heapsort**: is a comparison-based sorting technique that relies on the Binary Heap data structure. It operates as follows:

The array is transformed into a heap data structure using heapify.

Then, one by one, the root node of the Max-heap is removed and replaced with the last node of the heap.

Subsequently, the heap is adjusted (heapify) to maintain the heap property.

This process is repeated until the size of the heap is greater than 1.

It's efficient and has a guaranteed runtime of O(n log n), making it suitable for large data sets.

Taken From: <https://www.geeksforgeeks.org/heap-sort/>

**Stack:** Is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last. When implementing a Stack, you can have the push method to add elements to it, peek to view the element at the top of the stack without removing it, and pop to remove the element at the top of the stack. You can also check the size of the stack with size and determine if it is empty with isEmpty.

Taken From: <https://www.geeksforgeeks.org/stack-data-structure/?ref=lbp>

**Queue**:A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order. Allows you to store objects and then retrieve them in the order they were inserted.It implements methods to add elements to it, 'offer' to add elements, 'peek' to see the element at the front of the queue without removing it, and 'poll' to remove the element at the front of the queue. You can also check the size of the queue with 'size' and check if it's empty with 'isEmpty'

Taken From: <https://www.geeksforgeeks.org/queue-data-structure/?ref=lbp>

**PriorityQueue:** The PriorityQueue operates by priority, as its name suggests, instead of following the FIFO (First-In-First-Out) model of a regular queue. In this data structure, elements are sorted based on their natural order or through a custom comparator.

Taken From: <https://www.geeksforgeeks.org/priority-queue-class-in-java/>

### 

### 

### **Step 3. Creative Solutions Search**

**Alternative 1:**

**HashTable, LinkedList, Stack, Queue, PriorityQueue, Heapsort:**

The problem requires that task searching be as fast as possible, so implementing a hash table would be one of the best options because the search would have a complexity of O(1). In case of collisions in the hash table, we would implement a linked list to handle them. Regarding task prioritization, a PriorityQueue is a great option for this problem. PriorityQueue organizes elements so that the ones with the highest priority are at the top of the queue and are the first to be removed or accessed. Priority is generally determined based on a specific criterion, and they will be sorted using a comparison algorithm like Heapsort (N log n). Additionally, we can work with non-prioritized tasks using a queue that processes them in a first-come, first-served (FIFO) order. As for the stack (LIFO), it's implemented for the undo function. To achieve this, we create a stack to track user actions. Each entry in the stack contains information about the action taken and the details of the affected task. Then, we implement a method to undo the user's action by popping the last action from the stack and reversing the corresponding action based on the information stored in the stack.

**Alternative 2:**

**Array, Heapsort, Priority Queue, Stack,queue:**

To store tasks, we use an array, which is a static data structure. Searching for a task in the array would have a complexity of O(n), and with a large amount of data, it's not very effective. For prioritized tasks, a Priority Queue is utilized, which will sort tasks using Heapsort based on their priority levels (Heapsort has a complexity of O(n log n)). For non-prioritized tasks, we employ a queue (FIFO) to manage them. Lastly, we use a stack for the undo feature to reverse actions taken by the user.

**Alternative 3:**

**ArrayList, Stack, Queue, PriorityQueue, and Heapsort.**

ArrayList for Tasks: We use an ArrayList to store all tasks. Each element in the ArrayList contains detailed information about a task, including its title, description, due date, and priority. Heapsort for sorting the priority queue. Stack for Undo: We employ a stack (LIFO) to track user actions. Each time a user takes an action, it's recorded in the stack. Each entry in the stack includes information about the action and the details of the affected task. Queue for Non-Prioritized Tasks: We use a queue (FIFO) to manage non-prioritized tasks in the order they arrive, ensuring straightforward task management.

**Step 4. Transition from Ideas to Preliminary Designs.**

In this step, unfeasible alternatives are discarded. Therefore, Alternative 2 is discarded because a static data structure for managing a large number of tasks is not suitable for building an efficient and functional system. Additionally, task searching would have a complexity of O(n), making it less convenient. Despite implementing Priority Queue with Heapsort, which is a viable sorting method, managing tasks in a static structure isn't practical.

As a result, we consider Alternatives 1 and 3.

**Alternative 3** resolves the issue of using a static data structure for storing tasks by employing a dynamic ArrayList. This allows the system to accommodate all necessary tasks efficiently, although task searching in an ArrayList is O(n), which is not ideal but still functional. For prioritized tasks, Heapsort could be applied within a PriorityQueue to prioritize tasks by their levels of importance. For non-prioritized tasks, a queue is suitable due to its first-come, first-served (FIFO) handling, and a stack is used for the undo feature. In summary, this alternative is a possible implementation.

Therefore, the preliminary design for Alternative 3 is as follows:

**Data Structures:**

**ArrayList**: We will use an ArrayList to store all tasks. Each element of the ArrayList will contain detailed information about a task, including title, description, due date, and priority.

**Heapsort:** The Heapsort algorithm will be used to sort prioritized tasks within the PriorityQueue based on their priority.

**Stack:** A stack (LIFO) will be used to track user actions and enable the undo function.

**Queue:** A queue (FIFO) will be used to manage non-prioritized tasks in the order of arrival.

**Priority Queue:** A priority queue will be used to manage prioritized tasks.

**Functionality:**

-Each task will be stored in the ArrayList.

-Task prioritization will be handled through the priority queue, automatically organizing tasks at the top based on their priority levels.

-Non-prioritized tasks will be managed in a queue (FIFO) to ensure they are completed in the order of arrival.

-User actions will be tracked through a stack, enabling the undo function.

In case of collisions in the hash table, a linked list will be used to handle them and ensure access to all tasks.

-The Heapsort algorithm will be applied to the PriorityQueue to ensure prioritized tasks are sorted by their priority levels.

**Alternative 1** implements a hash table, which is one of the best options for achieving fast task searching with a complexity of O(1) thanks to key management, making it highly optimized. The challenge with hash tables is collision cases, but a solution for this is to implement a linked list to manage them, which has great potential. As for task prioritization, Priority Queue is a strong choice. PriorityQueue organizes elements so that those with the highest priority are at the top of the queue and are the first to be removed or accessed. Prioritization is usually determined based on specific criteria and sorted using a comparison algorithm like Heapsort (N log n).

Additionally, Queue is suitable for handling non-prioritized tasks, ensuring they are processed in the order they arrive (FIFO). In the case of Stack (LIFO), it's implemented to facilitate the undo function. To do this, a stack is created to track user actions. Each time a user takes an action, it's recorded in the stack. Each stack entry contains information about the action taken and the details of the affected task. This design offers functionality and feasibility.

Therefore, the preliminary design for Alternative 1 is as follows:

**Data Structures:**

**HashTable**: A hash table will be used to store all tasks. The key will be a unique identifier for each task, and the value will contain detailed task information.

**LinkedList:** In case of collisions in the hash table, a linked list will be implemented to handle them and allow the storage of multiple elements with the same key.

**PriorityQueue**: A priority queue will be used to manage prioritized tasks. Tasks will be organized and accessed based on their level of priority.

**Queue**: A queue (FIFO) will be employed to manage non-prioritized tasks in the order of arrival.

**Stack**: A stack (LIFO) will be implemented to track user actions and enable the undo function.

**Heapsort**: The Heapsort algorithm will be used to sort prioritized tasks in the PriorityQueue based on their priority level.

**Functionality:**

-Each task will be stored in the hash table with a unique identifier.

-Task prioritization will be done through the priority queue, automatically organizing prioritized tasks at the top.

-Non-prioritized tasks will be managed in a queue (FIFO) to ensure they are processed in the order they arrive.

-User actions will be tracked using a stack, enabling the undo function.

In cases of collisions in the hash table, a linked list will be utilized to resolve them, ensuring access to all tasks.

-The Heapsort algorithm will be applied to the PriorityQueue to ensure prioritized tasks are sorted by their priority level.

So we have two potential alternatives. In the next step, we will evaluate them based on specific criteria and select the best alternative.

**Step 5. Evaluation and Selection of the Best Solution**

*Criteria*

Criteria must be defined to allow the evaluation of solution alternatives and, based on this result, choose the solution that best meets the needs of the problem presented. The criteria chosen in this case are as follows. Next to each one, a numerical value has been assigned in order to establish a weighting that indicates which of the possible values of each criterion carry more weight (i.e., are more desirable).

**Criteria 1:**

The alternative meets the requirements and is optimal.

[2] if it meets the requirements and is optimal.

[1] it meets the requirements but is not optimal.

**Criteria 2: Flexibility and Adaptability:**

[4] High flexibility and adaptability. The solution is highly flexible and can easily adapt to changes in requirements or conditions.

[3] Moderate flexibility and adaptability. The solution is flexible but may require some effort to adapt to changes.

[2] Low flexibility and adaptability. The solution is rigid and difficult to adapt to changes.

[1] Very low flexibility and adaptability. The solution is completely inflexible and cannot adapt to changes.

**Criteria 3: Ease of implementation.**

[2] Simple but it works.

[1] More complex but it works.

**Criteria 4: Problem division: divide and conquer (more divisions are better).**

[2] Uses more than 3 structures.

[1] Uses 3 or fewer structures.

*Evaluation*

|  | Criteria 1 | Criterio 2 | Criterio 3 | Criterio 4 | Total |
| --- | --- | --- | --- | --- | --- |
| Alternativa 1 | 2 | 3 | 1 | 2 | 8 |
| Alternativa 3 | 1 | 2 | 2 | 1 | 6 |

*Selection*

In conclusion, implementing alternative 1 is the best choice according to the criteria.