**Método de la Ingeniería**

Members:

* Juan José De La Pava - A00381213 + Marlon Gómez Victoria
* David Artunduaga Penagos - A00396342 + Marlon Gomez Victoria
* Rony Farid Ordoñez - A00397968 + Jeison Mejía Trujillo

**PHASE 1: PROBLEM IDENTIFICATION**

**Problem**: The problem consists of the need to design and develop a task and reminder management system that allows users to efficiently add, organize, and manage their to-do's and personal reminders. This system must address several key issues, such as storing detailed task information, managing priorities, undoing actions, and providing a friendly user interface.

**Problem Definition:** The Need to develop a task and reminder management system that allows users to manage their to-dos, assign priorities, and undo actions, with an intuitive user interface.

**Identification of needs and symptoms**

* Users need an effective way to manage and remember their daily tasks and reminders.
* Users often forget their tasks and deadlines due to the lack of a proper management tool.
* Users feel overwhelmed by the number of tasks and cannot identify which ones are the most important.
* Users want to be able to track actions performed in the system and revert changes if necessary.
* Users have difficulty finding specific tasks in their list due to a lack of sorting or classification.
* Users want to be able to differentiate between priority and non-priority tasks.

**Requirements:**

***Note:*** *The requirements specification is in a file with the same name in the DOCS folder of the project.*

**RF0: Add Tasks and Reminders**

Users must be able to add new tasks and reminders to the system.

Each task or reminder should include information such as a title, description, deadline, and priority. They are subsequently organized into two data structures, one by priority and another by order of arrival.

**RF1: Modify Tasks and Reminders**

Users should have the ability to modify the information of existing tasks and reminders.

This includes the capability to change the title, description, deadline, and priority.

**RF2: Delete Tasks and Reminders**

Users should be able to delete tasks and reminders from the system when they are no longer needed.

**RF3: Task Priority Management**

The system should have categorization for tasks, dividing them into two categories: "Priority" and "Non-priority."

**RF4: List Tasks and Reminders**

Users must be able to view a list of all their tasks and reminders.

They should have the option to sort this list by deadline or priority.

**RF5: Undo Function**

Implement a function that allows users to undo the last action performed in the system.

This includes the ability to undo adding, modifying, or deleting tasks and reminders.

**Non-Functional Requirements (NFRs):**

**NFR 1: Temporal Complexity Analysis**

Analyze the temporal complexity of at least two algorithms implemented in the project, such as heapsort and priority queue insertion.

**NFR 2: Spatial Complexity Analysis**

Perform an analysis of the spatial complexity of at least two algorithms implemented in the project, such as storage in the hash table and the priority queue.

**NFR 3: Abstract Data Type (ADT) Design**

Design and document the ADT of the data structures used in the project, such as the hash table and the priority queue.

**NFR 4: Class Diagram Design**

Create a comprehensive class diagram that includes data structures, the model, the user interface, and testing.

Use generics to make the design generic and reusable.

**NFR 5: Test Case Design**

Design test cases that cover different scenarios to validate the data structures and the system.

**NFR 6: Intuitive User Interface**

Design a user-friendly interface that allows users to interact with the system intuitively.

It should provide clearly labeled buttons or actions for adding, editing, and deleting tasks and reminders.

**PHASE 2: GATHERING THE NECESSARY INFORMATION**

To gain a comprehensive understanding of the concepts involved in the development of the task and reminder management system, a search for definitions and key elements related to the stated problem was conducted.

Definitions:

**Hash Table:** A data structure that enables efficient storage and retrieval of data through a hash function. Each entry in the hash table consists of a unique key and an associated value. The hash function is used to convert the key into a specific location in the table, allowing swift access to stored data.

(https://en.wikipedia.org/wiki/Hash\_table)

**User Interface**: The user interface (UI) is the point of human-computer interaction and communication in a device. This can include display screens, keyboards, a mouse, and the appearance of a desktop. It is also the way through which a user interacts with an application or a website.

(https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI)

**Priority Queue:** A data structure that organizes prioritized tasks based on their level of importance. Tasks are inserted into the queue according to their priority, ensuring that the most critical tasks are managed first and completed earlier.  
  
(https://www.geeksforgeeks.org/priority-queue-set-1-introduction/)

**"Undo" Method:** Undo is an interaction technique that is implemented in many computer programs. It erases the last change done to the document, reverting it to an older state. In some more advanced programs, such as graphic processing, undo will negate the last command done to the file being edited.

(https://en.wikipedia.org/wiki/Undo)

**LIFO Stack:** A data structure that stores elements in a manner where the last added element is the first to be removed. In the context of "undo," a stack is used to keep a record of user actions, enabling the efficient reversal of the most recent action.

(https://www.geeksforgeeks.org/stack-data-structure/)

**Heap Sort:** Heap Sort is a sorting algorithm based on heap data structures. It involves transforming the set of elements to be sorted into a heap, a specialized binary data structure where the parent element is always greater or smaller than its child elements, depending on whether ascending or descending order is desired. The algorithm then iteratively extracts the top element from the heap (the maximum or minimum, depending on the desired order) and places it in the correct position in the sorted list. Heap Sort is efficient and guarantees a runtime of O(n log n), making it suitable for sorting tasks based on their priority or deadline in the task and reminder management system.

(https://www.geeksforgeeks.org/heap-sort/)

**PHASE 3: SEARCH FOR CREATIVE SOLUTIONS**

In Phase 3 of the solution exploration process, various alternatives have been identified to address the key aspects of the task and reminder management system:

**Store Tasks and Reminders:**

*Alternative 1. Fixed-size Arrays:*

A fixed-size array offers a solution for storing tasks and reminders. Each element within the array can represent a task or reminder, with the possibility of resizing the array when it reaches capacity. While this approach provides a straightforward means of storage, it may require careful management of size limitations.

*Alternative 2. Dynamic Arrays:*

Dynamic arrays, such as ArrayList in some programming languages, provide flexibility in accommodating tasks and reminders. These arrays can dynamically resize themselves as needed, ensuring that there are no predefined size restrictions. This adaptability can be particularly helpful in managing an ever-changing list of tasks and reminders.

**Priority Management:**

*Alternative 1. Stacks:*

Stacks, operating under the Last-In-First-Out (LIFO) principle, present a viable means of managing tasks with varying priorities. Tasks can be pushed onto the stack based on their priority levels, ensuring that high-priority tasks take precedence. This approach aids in the efficient processing of critical tasks, aligning with the system's priority management requirements.

*Alternative 2. Queue:*

Queues adhere to the First-In-First-Out (FIFO) principle and can be effectively employed for handling non-prioritized tasks in the order of their arrival. This approach ensures that tasks are processed in the sequence they are received, making it suitable for managing tasks without specific priority assignments.

*Alternative 3. Heap:*

The utilization of priority queues implemented through heap data structures provides a robust solution for managing tasks according to their priority levels. The heap structure enables the system to first process tasks with higher priorities, promoting efficient task management. This approach aligns with the system's objective of categorizing and addressing tasks based on their importance and urgency.

*Alternative 4. Binary Search Tree (BST):*

A Binary Search Tree (BST) offers an additional alternative for managing tasks based on priority. In a BST, each task is represented as a node with a priority value, and tasks are organized in a hierarchical structure. The BST property ensures that tasks with higher priorities are placed closer to the root of the tree, allowing for efficient retrieval and processing of high-priority tasks.

*Undo Method Implementation:*

**Stacks:**

To implement the "undo" functionality, a stack data structure can be employed. With this approach, each user action is pushed onto the stack, creating a chronological record of actions. When a user wishes to undo a specific action, the last action can be efficiently popped from the stack and reversed, restoring the previous state of tasks and reminders. This implementation ensures a straightforward and intuitive user experience by allowing users to easily backtrack and correct their actions within the system

**PHASE 4: TRANSITION FROM BRAINSTORMING TO PRELIMINARY DESIGNS**

We begin the process of discarding ideas that are not viable to solve the problem.

**Task and Reminder Storage:**

**Discarding Fixed-Size Arrays:** At this stage, we decided to discard the idea of using fixed-size arrays for task and reminder storage. The main reason behind this choice is that we do not know in advance the exact number of tasks and reminders that users will want to manage. Fixed-size arrays have a predefined limitation on the number of items they can contain, which could result in data loss if that limit is exceeded. Therefore, to ensure system scalability and to allow users to manage a variable number of tasks and reminders, we believe that fixed-size arrays are not the appropriate choice in this context.

**Task Priority Management:**

**Discarding Binary Search Tree (BST):** We discarded the Binary Search Tree (BST) structure for priority management due to its limitation in organizing items based on priority. Although BSTs can organize items in an ordered manner, they are not designed to prioritize items based on their level of importance. Therefore, they are not suitable for our goal of differentiating and managing tasks based on their priority.

**Discard Stack:** We opted to discard the stack data structure in task priority management. The main reason behind this decision is that stacks operate on the "Last In, First Out" (LIFO) principle, which means that the last task added would have the highest priority. This does not match our need to manage tasks based on their importance or urgency, where priority tasks should be handled before others.

***The remaining ideas (Dynamic Arrangements, Hash Tables, Priority Queuing, and Normal Queuing) are retained for further analysis and design in the next phase.***

**PHASE 5: EVALUATION AND SELECTION OF THE BEST SOLUTION**

**Priority Management:**

Criterion A: Task Processing Efficiency:

* [1] Linear O(n)
* [2] Logarithmic O(n log(n))
* [3] Logarithmic O(log(n))
* [4] Constant O(1)

Criterion B: Ability to Prioritize Tasks:

* [0] No
* [1] Yes

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Queue** | **Priority Queue** |
| Task Processing Efficiency | Linear O(n) | Constant O(1) |
| Ability to Prioritize Tasks | No | Yes |
| Total Score | 1 | 5 |

Based on the evaluation, Priority Queue is the preferred choice due to its superior efficiency in task processing and its ability to prioritize tasks effectively. It achieves a higher total score and aligns better with the project's requirements for task management. Therefore, we select Priority Queue as the optimal solution for managing tasks with different priorities in our system.

**To determine the optimal choice, we will establish a criterion with the following rating scale:**

* 4: Highly efficient
* 3: Efficient
* 2: Not efficient
* 1: Poor

This rating scale will help us objectively assess and compare the alternatives to identify the most suitable option for our project.

**Store Tasks and reminders**

* Criterion A: Storing task and reminder efficiency.
* Criterion B: Accessing task and reminder efficiency.
* Criterion C: Deleting.

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Dynamic Arrays** | **Hash Tables** |
| Criteria A | Poor- 1 | Highly efficient- 4 |
| Criteria B | Poor- 1 | Highly efficient- 4 |
| Criteria C | Poor- 1 | Highly efficient - 4 |
| Total Score | 3 | 12 |

Evaluation:

To store tasks and reminders efficiently, Hash Tables receive the highest score, with a constant O(1) time complexity. Accessing and deleting tasks and reminders also achieve the highest efficiency with Hash Tables, each with a constant O(1) time complexity.

Furthermore, Hash Tables support the management of tasks with different priorities, making them a suitable choice for our project.

Selection:

Based on the evaluation, Alternative 2 (Hash Tables) is the preferred choice due to its consistently high efficiency across all criteria. It meets our requirements for efficient storage, access, and deletion of tasks and reminders while allowing us to manage tasks with different priorities effectively.

**PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS**

*All other documents can be found in the DOCS folder of our project.*