**Project’s Design**

1. **Problem Identification**

The problem (that has been identified) is the need to develop a task and reminder management system that allows users to efficiently add, organize, and manage their to-dos and reminders. This problem stems from the lack of an effective tool for organizing tasks and reminders that are easy to use and provide users with the ability to prioritize their activities.

1. **Gathering Necessary Information**

Understanding user needs and expectations is critical to the effective design and development of the task and reminder management system. Knowing the several types of tasks and reminders that users will manage is essential to designing an appropriate data structure.

Understanding how users will interact with the system in their daily workflow is essential to designing an intuitive user interface. Gathering information on how users will assign priorities and categorize tasks is crucial to implementing priority management. The choice of technical aspects, such as programming language, databases, and development tools, should be based on the project's needs. Data security and privacy are key considerations, and it is important to know the expectations of users in this regard.

If the system is subject to specific regulations, gathering information on the corresponding compliance requirements is necessary. Planning for ongoing maintenance and support is essential to ensure optimal system performance over time. User interface design, usability, and navigation should be informed by user feedback to ensure a positive experience. Ongoing feedback and user testing are necessary to adjust and improve the system.

1. **Search For Creative Solutions**

**To solve this problem, we can use structures such as:**

**Array List:**  
Utility: An Array List is a dynamic data structure that provides a resizable list. It can be used to store tasks and reminders in a flexible way.  
Advantages: It allows quick access to items via indexes and is easy to use and manage. It is useful when the size of the list is dynamic and may change over time.  
**Hash table:**  
Utility: A hash table provides efficient lookup and fast insertion and deletion times. It can be used to store tasks and reminders, using a unique identifier as a key.  
Advantages: Provides quick access to data, especially when you need to search for tasks based on a unique identifier, such as a task code.  
**Priority Queue (Heap):**  
Benefit: A heap (min-heap) can be used to organize tasks according to their level of importance or deadline. Priority tasks are handled first.  
Advantages: Ensures that the most important tasks are handled first. Efficient for inserting new tasks and extracting the highest priority task.

**Linked List (FIFO):**  
Utility: A linked list can be useful for managing non-priority tasks on a first-come, first-served basis.  
Advantages: Allows easy insertion and deletion of items in the middle of the list, while maintaining a specific order for non-priority tasks.  
**Stack (****LIFO <Last In, First Out>):**

Utility: A stack can be used to implement the "undo actions" functionality. Each action performed by the user can be stacked for later reversal.

Advantages: Allows to undo the last action in an efficient way, following the Last-In-First-Out (LIFO) principle.

**Search Tree (BST):**

Utility: A binary search tree can be useful for organizing tasks by deadline or priority, facilitating efficient search and retrieval.

Benefit: Ensures quick access to tasks based on a specific criterion, such as deadlines.

**Graph:**

Utility: A graph can be useful if tasks have complex relationships with each other, such as dependencies. Tasks could be nodes and dependencies could be represented by arcs in the network.

Advantages: Allows modeling and managing complex relationships between tasks, such as dependencies and workflows.

1. **Transition From Idea Formulation to Preliminary Designs**

1st Alternative: Hash Table

* Due to the way hash table stores data, it is likely to exist collisions between elements (tasks).
* Depending on the hash function, the hash table could waste a lot of memory.
* Hash tables are useless if it is needed to work with sorted data.

2nd Alternative: Stack (LIFO):

* Stacks are not suitable for complex operations like searching for an element in the middle of the stack.
* Stacks provide access to only one item at a time. In case an item at the end of the stack is required, it will be necessary to go through all the stack to access that specific item.
* Stacks does not work for sorting and searching data.
* Stacks are not useful when working with substantial amounts of data.

3rd Alternative: Queue:

* Queues provide access to only one item at a time.
* It is a fixed data structure.
* Queues are not designed for sorting and searching data.

4th Alternative: Priority queue:

* Its functionality is limited to elements based on a defined priority.
* Priority queues are not well-suited for scenarios where the priority of elements frequently changes.
* Priority queue implementations may have higher memory overhead due to the need to store both elements and their associated priorities.

1. **Evaluation and Selection of the Best Solution**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Temporal E.** | **Space E.** | **Usability** | **Scalability** | **Total** |
| **Hash Table** | **2** | **2** | **2** | **2** | **8** |
| **Stack** | **1** | **2** | **2** | **1** | **4** |
| **Queue** | **1** | **2** | **2** | **1** | **4** |
| **Priority Queue** | **2** | **2** | **1** | **2** | **7** |

With this table, we can notice the best structure is the Hash Table. In fact, it will be used to storage, print, search, modify and delete any Task, functioning as a “Data Base” or persistence system throughout the program execution.

With the necessity of using a structure to manage the Tasks that are “priority" or important, our best chance would be to select the Priority Queue. This structure will be used to see the most relevant Task in our system and to have the possibility to edit or finish it.

A characteristic (that is mandatory) is to have the possibility to undo an action. Given that the action that must be undid is the last one that has been done, it is necessary to implement a Stack, because this structure manages itself like this.

A requirement with less importance is to manage non-priority tasks. In fact, we just need to “manage” or see in each iteration is the first that was added. So, the queue is the most appropriate for this work.

1. **Preparation of Reports and Specifications**

Problem specification: storage and prioritization of tasks

Inputs: the new task and its prioritization (a number between 1 and 5).

Outputs: a Boolean indicating that task was correctly created and added to the priority queue.

Diagrama de flujo

Pseudocódigo:

**Aqui se especifica el problema, se escriben algunas consideraciones y después se realiza el**

1. **Design Implementation**

**Hash table methods**

|  |  |
| --- | --- |
| HashTable() |  |
| Create a hash node with a predefined size |

|  |  |
| --- | --- |
| isEmpty() |  |
| The function checks if a data structure is empty by comparing its size to zero. |
| return: The method is returning a boolean value, specifically whether the size variable is equal to 0. |

|  |  |
| --- | --- |
| hashFunction(K key) |  |
| The hashFunction takes a key and returns a hash value within a predefined size. |
| param: key The parameter "key" is of type K, which represents the type of the key that will be used to calculate the hash value. |
| return: The method is returning the hash value of the given key. |

|  |  |
| --- | --- |
| V (K key, V value) |  |
| The function puts a key-value pair into a hash table, using a hash function to determine the index, and handles collisions by chaining. |
| param: key The key is the unique identifier for the value being stored in the hash table. It is used to determine the index at which the value will be stored in the table. |
| return: value The value parameter represents the value that is being associated with the specified key in the hash table. |

|  |  |
| --- | --- |
| remove (K key) |  |
| The remove() function removes a key-value pair from the hash table based on the given key. |
| param: key The key is the value that is used to identify and locate the element in the hash table that needs to be removed. |
| return: void |

|  |  |
| --- | --- |
| Int size() |  |
| The function returns the size of a data structure. |
| param: none |
| return: The method is returning the value of the variable "size". |

|  |  |
| --- | --- |
| V getValue (K key) |  |
| The function retrieves the value associated with a given key in a hash table. |
| param: Key The key parameter is the key of the element we want to retrieve the value for. |
| return: The method is returning the value associated with the given key. |

|  |  |
| --- | --- |
| PrintHashTable() |  |
| The function iterates through a hash table and prints the key-value pairs of each node. |
| param: none |
| return: void |

**Priority Queue methods**

|  |  |
| --- | --- |
| add(T t) |  |
| The add method adds a new element to a priority queue in ascending order based on the element's natural ordering. |
| param: t The parameter `t` is of type `T`, which is a generic type. It represents the element that needs to be added to the priority queue. |
| return: The method is returning a boolean value, which is always true. |

|  |  |
| --- | --- |
| T remove() |  |
| The remove() function removes and returns the data from the head of a linked list. |
| param: none |
| return: The method is returning the removed data of type T. |

|  |  |
| --- | --- |
| isEmpty() |  |
| The function checks if the head of a linked list is null, indicating that the list is empty. |
| param: none |
| return: The method is returning a boolean value. |

|  |  |
| --- | --- |
| T get() |  |
| The function returns the value at the head of the queue, or throws an exception if the queue is empty. |
| param: none |
| return: The method is returning the value of the head element in the queue. |

|  |  |
| --- | --- |
| size() |  |
| The function returns the size of a priority queue by iterating through its nodes and counting them. |
| param: none |
| return: The size of the NodePriorityQueue. |

|  |  |
| --- | --- |
| remove(T t) |  |
| The remove function removes the first occurrence of a given element from a priority queue. |
| param: t The parameter `t` is of type `T`, which is a generic type. It represents the element that needs to be removed from the priority queue. |
| return: void |

**Queue methods**

|  |  |
| --- | --- |
| isEmpty() |  |
| The function checks if the top element of a stack is null, indicating that the stack is empty. |
| param: none |
| return: The method is returning a boolean value, which indicates whether the stack is empty or not. |

|  |  |
| --- | --- |
| T peek() |  |
| The function returns the first element in the queue, or null if the queue is empty. |
| param: none |
| return: The method is returning the element at the front of the queue, or null if the queue is empty. |

|  |  |
| --- | --- |
| T get() |  |
| The function returns the value at the top of the queue and throws an exception if the queue is empty. |
| param: none |
| return: The method is returning the value of the top element in the queue. |

|  |  |
| --- | --- |
| add() |  |
| The add() function adds a new element to the end of a queue. |
| param: t The parameter `t` is of type `T`, which is a generic type. It represents the element that  \* needs to be added to the queue. |
| return: The method is returning a boolean value indicating whether the element was successfully added to the queue. |

|  |  |
| --- | --- |
| T poll() |  |
| The function `poll()` removes and returns the element at the front of the queue, or returns null if the queue is empty. |
| param: none |
| return: The method is returning an object of type T. |

|  |  |
| --- | --- |
| T remove() |  |
| The function `poll()` removes and returns the element at the front of the queue, or returns null if the queue is empty. |
| param: none |
| return: The remove() function removes and returns the top element from the queue, throwing an exception if the queue is empty. |

|  |  |
| --- | --- |
| remove() |  |
| The remove function removes the first occurrence of a given element from a linked list. |
| param: The parameter "t" is of type T, which is a generic type. It represents the element that needs to be removed from the queue. |
| return: void |

**Stack methods**

|  |  |
| --- | --- |
| T peek() |  |
| The `peek` function returns the top element of a stack without removing it, or null if the stack is empty. |
| param: none |
| return: The method is returning the element at the top of the stack, or null if the stack is empty. |

|  |  |
| --- | --- |
| T get() | U,{3fbc0f0b-b19d-4a7b-95d2-11733261ef0b}{35},3.125,3.125 |
| The function returns the value at the top of the stack, or throws an exception if the stack is empty. |  |
| param: none |  |
| return: The method is returning the value of the top element in the stack. |  |

|  |  |
| --- | --- |
| push() | U,{2de6bd04-8f9d-4f55-a759-2a78f5da5360}{54},3.125,3.125 |
| The function pushes a new element onto a stack and updates its action and value. |
| param: \* @param before The parameter "before" is of type T and represents the element that will be added to  \* the stack before performing the action.  \* @param action The "action" parameter is a string that represents the action being performed.  \* @param after The parameter "after" is of type T, which means it can be any type of object. It  \* represents the object that will be added to the stack after the "before" object. |
| return: The method is returning a boolean value of true. |
| add() |  |
| The add function adds a new element to the top of the stack. |
| param: t The parameter `t` is of type `T`, which represents the element to be added to the stack. |
| return: The method is returning a boolean value of true. |

|  |  |
| --- | --- |
| add() |  |
| The add function adds a new element to the top of the stack. |
| param: t The parameter `t` is of type `T`, which represents the element to be added to the stack. |
| return: The method is returning a boolean value of true. |