**Task and Reminders System Management**

**Phase 1: Issue’s Context:**

Design a system for the management of tasks and reminders, that allows the user to add, organize, and administer his or her pending tasks and reminders.

**Identifying the problem:**

* The users need a tool for the management of their tasks and reminders effectively. This necessity arises from the hardships they come across while trying to maintain a registry of their daily responsibilities, due dates and priorities.

**Síntomas y necesidades**

* The users need an efficient way of storing and organizing their tasks and reminders. This includes the capacity to add details such as the title, description, deadline and priority.
* The users need a way of assigning priorities to their tasks and reminders. This could include being able to categorize tasks as a “Priority” or “Non-priority” and organize them as such.
* The users need to be able to undo actions that they’ve done in the system in order to correct errors or non desirable changes . This also includes a list of action and the possibility to revert them.
* The users need an efficient way of managing their tasks and reminders in a way that will save them time and prevent mistakes.
* The users might need to be able to receive reminders and notifications about the tasks that have a deadline.

**Fase Information Compilation:**

* **Hash Tables:** It's a data structure that is used for the efficient storage and recovery of data. It's based off of a hash function that maps out the key of a certain value. This hash function calculates the location or index where the value will be stored in the hash table.
* **LIFO:** "Last In, First Out", refers to a behavior in which the last element that is added or placed in the data structure, is the first one to be extracted or used. This means that the most recent element has the priority over the rest of the elements that were added previously.
* **Stack:** It's a linear data structure that follows the LIFO (Last in, First Out) principle, which means that the last element that is inserted in the Stack, is the first one to be pointed at, retrieved or deleted. In other words, the elements are added and retrieved of the Stack by a single extremity called “the top”.
* **Queue:** It’s a linear data structure that follows the principle of FIFO (First in, First Out), which means that the first element that is inserted in the Queue is the first one to be pointed at, retrieved or deleted. It’s often represented as a Queue of elements, where the elements are added at the end (back) and are eliminated at its start (front).
* **Analysis of temporal complexity:** An analysis of temporal complexity, also known as an analysis of execution runtime, is an evaluation of the time consumption that an algorithm or program has in function of the size of its entry. The objective that an analysis of temporal complexity has is to comprehend how the execution runtime increases whilst its entry becomes larger, which gives us valuable information about the efficiency and performance of the algorithm at hand.
* **Analysis of spatial complexity:** The analysis of spatial complexity, also known as analysis of memory consumption, refers to the evaluation of how much memory (storage space) an algorithm or program requires in function of the size of its entry. Just as the analysis of temporal complexity is centered around runtime, the analysis of spatial complexity is centered around the amount of memory that an algorithm uses up.
* **Iterative Algorithms:** An iterative algorithm is a type of algorithm that solves a problem through a repetition of a set of instructions or steps in a loop, where each iteration of the loop refines the solution progressively until a certain end criteria is met. Iterative algorithms are known for using loops or cycles in order to repeat a series of operations until a desired result is achieved or a termination condition is met.
* **API:** An API, or “Application Programming Interface”, is a set of rules and protocols that permits different software components to communicate to one and other. In essence, an API defines the ways in which software programs or modules can interact and access functions and data provided by other programs, libraries or services.
* **Big O:** It’s used to describe the efficiency or complexity of an algorithm in terms of the runtime or the use of resources in relation to the size of its input. In other words, Big O is used to analyze the behavior of an algorithm when its input increases in size and tends towards infinity.
* **Unit Tests:** Its a work method that implements testing each code unit individually and/or separately. This in order to verify that it all works according to what is desired. A code unit refers to the smallest and most manageable part of a program, as a function, a method, or even a class.
* **Heapsort:** Heapsort is an efficient sorting algorithm that is used to sort a set of data in a specified order (ascending or descending.) The Heapsort algorithm is based on a data structure called a “mound” or “heap”, which is a special type of complete binary tree that meets the properties of a “mound”. In a “mound”, the value of each node is bigger or equal to the values of it’s children, (in the case of a maximum mound) and less or equal to its children (in the case of a minimum mound.) The root of the tree is the maximum (in a maximum mound) or the minimum (in a minimum mound.)
* **Arrays:** It’s a data structure that is used to store a collection of elements of the same type. In order to access the data that is stored, the user must enter the number of the position in the array where the data is being stored, but for this to work, the array’s length must have been defined previously. The length of an array is fixed, which means that this is a static data structure.
* **Arraylist:** It’s a data structure that represents a dynamic list or array. A difference with the traditional static arrays, is that an ArrayList can change its size dynamically during the execution of the program. Like traditional arrays, an ArrayList’s elements can be accessed with its position in the list.
* **Linked List:** A linked list is a dynamic data structure that stores and organizes a collection of elements in a sequential way. In contrast to an array for example, where the elements are stored in continuous locations of memory, in a linked list, the elements are stored in such a way that they are dispersed, and each element contains a reference (or link) to the next element in the list.

**Fase 3: Possible Solutions**

* **Store tasks and reminders**
  + Arrays
  + Arraylist
  + Linked Lists
  + Stacks
  + Queues
  + Hash Tables
* **Priority Management**
  + Stacks
  + Queues
  + BST
  + Linked Lists
  + Priority Queues
* **Undo Actions**
  + Stacks
  + Queues
  + Linked List

In order to store and manage tasks, priorities and actions to undo, we can use a variety of data structures, each with its own access process.

* Arrays and ArrayLists lend a direct access through the element's position, perfect for when we need references for specific elements.
* Linked Lists require sequential access from beginning to end, since every element is linked from one to the next.
* Stacks work with LIFO (Last-In-First-Out), where only the element at the top can be accessed.
* Queues work with FIFO (First-In-First-Out), where the only element that can be accessed is at the front of the Queue.
* Hash tables use keys in order to access its values, which permits a fast and constant access in function of the key that was input. Selecting the data structure will depend on the nature of the data and the specific requirements for accessing and manipulation.

**Phase 4: Transition from Ideas to Preliminary Designs**

* **Task and Reminder Storage**

**Arrays:** Arrays have a fixed size, which means you need to know the exact maximum number of elements you need to store. Additionally, arrays can lead to wasted space if you choose one large enough to accommodate all possible tasks. On the other hand, insertion and deletion operations can be inefficient, especially when not performed at the end of the array. Searching for elements within an unordered array is limited because the search is conducted using indices

**Stacks:** Stacks follow the 'last in, first out' (LIFO) principle, limiting access to elements only from the top. When more flexible access to tasks and reminders is required, such as searching for elements based on specific criteria or removing intermediate elements, stacks can be inefficient and challenging to manage.

* **Queues:** Queues follow the 'first in, first out' (FIFO) principle, which means you can only access the item that has been in the queue the longest. This can be a disadvantage when you need to access specific tasks or reminders based on criteria like priority or due date. Additionally, items in the middle of the queue are not easily visible or accessible, making it challenging to manage multiple tasks simultaneously. Removing specific items that are not at the front of the queue can also be complex and inefficient.
* **Priority managment**
  + **Stack:** Stacks, following the 'last in, first out' (LIFO) principle, do not naturally allow for the categorization of tasks into 'Priority' and 'Non-priority,' making it challenging to organize tasks by level of importance. Additionally, stacks do not efficiently adapt to the management of priority tasks as they are not inherently capable of maintaining a hierarchy of priorities. Furthermore, if you wish to use FIFO for non-priority tasks, implementing it in a stack would be inefficient and complex. The limitation of access to only the top element also makes it difficult to manage multiple tasks of different priority categories simultaneously.
* **Undo actions:**
  + **Queue:** It is a data structure that follows the FIFO (First In, First Out) principle, maintaining a strict order of execution. This could limit flexibility in action management since actions would be undone in the same order in which they were performed, which may not be optimal for applications that require action prioritization or the ability to 'redo' actions in a specific order.

**Phase 5: Evaluation criteria**

To evaluate and select among the various data structure options, we will apply the following criteria: efficiency, ease of use, maintainability, and scalability.

* **Store tasks and reminders**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Efficiency** | **Usability** | **Maintainable** | **Scalability** | **Total** |
| **Linked list** | **3** | **4** | **4** | **4** | **15** |
| **Stack** | **2** | **3** | **3** | **2** | **10** |
| **Queue** | **2** | **3** | **3** | **3** | **11** |
| **Hash table** | **5** | **4** | **4** | **4** | **17** |
| **Arraylist** | **3** | **4** | **4** | **4** | **15** |

* **Priority Management:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Efficiency** | **Usability** | **Maintainable** | **Scalability** | **Total** |
| **Queues** | **4** | **4** | **4** | **4** | **16** |
| **BST** | **3** | **3** | **3** | **4** | **13** |
| **Linked List** | **3** | **3** | **3** | **3** | **12** |
| **Priority Queues** | **5** | **4** | **4** | **5** | **18** |

* **Undo actions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Efficiency** | **Usability** | **Maintainable** | **Scalability** | **Total** |
| **Stacks** | **4** | **4** | **4** | **4** | **16** |
| **Linked list** | **3** | **3** | **3** | **3** | **12** |

Taking into account the previous results, we can conclude the following:

* **Store tasks and reminders:** Based on the evaluation results, hash tables receive the highest score in each of them, making them the preferred option for storing tasks and reminders. They exhibit high efficiency, ease of use, maintainability, and are highly scalable. Therefore, hash tables are the winning choice for this use case.
* **Priority managment:** Taking the evaluation into account, priority queues receive a high score in efficiency and scalability due to their ability to handle priority tasks effectively. Simple queues are also efficient, but their prioritization is limited. Binary search trees and linked lists receive lower scores in terms of efficiency and scalability due to their overall structure.
* **Undo actions:** Based on the evaluation, stacks receive a favorable score in efficiency, ease of use, maintainability, and scalability due to their LIFO nature, making them suitable for tracking and undoing actions efficiently. Linked lists are also a viable option, but they receive a slightly lower score due to their overall structure.