**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.

**Paso 4: Transición de las Ideas a los Diseños Preliminares**

* **Almacenamiento de tareas y recordatorios**
* **Arreglos:** Los arreglos tienen un tamaño fijo, lo que significa que hay que conocer con precisión la cantidad máxima de elementos que se necesita almacenar. Además, los arreglos pueden llevar al desperdicio de espacio si se elige uno lo suficientemente grande para acomodar todas las tareas posibles. Por otro lado, las operaciones de inserción y eliminación pueden ser ineficientes, especialmente cuando no se realizan al final del arreglo. La búsqueda de elementos dentro de un arreglo no ordenado nos limita ya que la búsqueda se realiza por medio de índices.
* **Pilas:** Las pilas siguen el principio de 'último en entrar, primero en salir' (LIFO), lo que limita el acceso a los elementos solo desde la parte superior. Cuando se requiere un acceso más flexible a tareas y recordatorios, cómo buscar elementos basados en criterios específicos o eliminar elementos intermedios, las pilas pueden resultar ineficientes y complicadas de manejar.
* **Colas:** Las colas siguen el principio 'primero en entrar, primero en salir' (FIFO), lo que significa que sólo puedes acceder al elemento que ha estado en la cola durante más tiempo. Esto puede ser una desventaja cuando se necesita acceder a tareas o recordatorios específicos basados en criterios como prioridad o fecha de vencimiento. Además, los elementos en medio de la cola no son fácilmente visibles ni accesibles, lo que puede dificultar la gestión simultánea de múltiples tareas. La eliminación de elementos específicos que no están en la parte frontal de la cola también puede resultar compleja e ineficiente.
* **Gestión de prioridades**
  + **Stack:** Las pilas, al seguir el principio 'último en entrar, primero en salir' (LIFO), no permiten la categorización natural de tareas en 'Prioritaria' y 'No prioritaria', lo que dificulta la organización de tareas por nivel de importancia. Además, las pilas no se adaptan de manera eficiente a la gestión de tareas prioritarias, ya que no son inherentemente capaces de mantener una jerarquía de prioridades. Además, si se desea utilizar FIFO para las tareas no prioritarias, implementarlo en una pila resultaría ineficiente y complicado. La limitación de acceso a solo el elemento superior también dificulta la gestión de múltiples tareas de diferentes categorías de prioridad al mismo tiempo.
* **Deshacer acciones:**
* **Queue:** Es una estructura de datos que sigue el principio FIFO (First In, First Out), que sigue un estricto orden de ejecución. Esto podría limitar la flexibilidad en la gestión de acciones, ya que las acciones se desharían en el mismo orden en el que se realizaron, lo que puede no ser óptimo para aplicaciones que requieren priorización de acciones o la capacidad de "rehacer" acciones en un orden específico.

**Paso 5: Criterios de evaluación**

Para evaluar y seleccionar entre las diversas opciones de estructuras de datos, aplicaremos los siguientes criterios: eficiencia, facilidad de uso, mantenibilidad y escalabilidad.

* **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** |

Teniendo en cuenta los anteriores resultados podemos concluir lo siguiente:

* **Almacenamiento de tareas y recordatorios:** Basado en los resultados de la evaluación las tablas hash obtienen la puntuación más alta en cada uno de ellos, lo que las convierte en la opción preferida para almacenar tareas y recordatorios. Tienen una alta eficiencia, son fáciles de usar, mantener y son altamente escalables. Por lo tanto, las tablas hash son la elección ganadora para este caso de uso.
* **Gestión de prioridades:** Teniendo en cuenta la evaluación, las colas de prioridades obtienen una puntuación alta en eficiencia y escalabilidad debido a su capacidad para manejar tareas prioritarias de manera eficaz. Las colas simples también son eficientes, pero su priorización es limitada. Los árboles de búsqueda binaria y las listas enlazadas obtienen puntuaciones más bajas en términos de eficiencia y escalabilidad debido a su estructura general.
* **Deshacer acciones:** Con base en la evaluación, las pilas obtienen una puntuación favorable en eficiencia, facilidad de uso, mantenibilidad y escalabilidad debido a su naturaleza LIFO, que las hace adecuadas para el seguimiento y deshacer acciones de manera eficiente. Las listas enlazadas (Linked Lists) también son una opción viable, pero obtienen una puntuación ligeramente más baja debido a su estructura general.