**EL MÉTODO DE DISEÑO EN INGENIERÍA**

**Contexto Problemático**

En la vida cotidiana las personas se enfrentan a un entorno casa vez mas agitado y exigente, caracterizado por una abrumadora cantidad de responsabilidades, tareas y recordatorios que deben gestionar. La multiplicidad de roles y compromiso laborales, así como la contante interconexión digital, da lugar a una creciente complejidad en la organización de las actividades diarias, lo que genera una gran dificultad en las personas de mantenerse al tanto de sus actividades y compromisos.

Para abordar esta problemática se ha concedido un sistema de gestión de tareas y recordatorios, este se debe integrar como una solución integral diseñada para proporcionar una solución eficaz para que el usuario pueda administrar y organizar sus responsabilidades. En este contexto actual, el Sistema de Gestión de Tareas y Recordatorios se presenta como una herramienta tecnológica muy valiosa para lograr un equilibrio entre la vida laboral y personal. En la era digital, la gestión de tareas y recordatorios se ha vuelto cada vez más desafiante debido a la gran cantidad de información y distracciones que enfrentamos constantemente. Por lo tanto, este sistema es un gran aliado para ayudarnos a mantenernos organizados y recordar nuestros compromisos importantes.

**FASE 1: IDENTIFICACIÓN DEL PROBLEMA**

Problema: La falta de un sistema eficiente para que los usuarios gestionen sus tareas pendientes y recordatorios.

Necesidades:

* Organización de tareas y recordatorios.
* Priorización de tareas.
* Capacidad de deshacer acciones.
* Diseñar una interfaz de usuario.
* Acceso rápido a la información.
* Registro de acciones.

**FASE 2: RECOPILACIÓN DE LA INFORMACIÓN NECESARIA**

**Pila**

Una pila está formada por datos y un tope, lo datos son conjunto de elementos, en general del mismo tipo, ordenados implícitamente y accesibles desde un único punto, el tope es el indicador de la posición del último elemento insertado; da lugar a una ordenación LIFO. (Cantador, s.f.)

**LIFO: Last-In, Last-Out**

El patrón de gestión de colas LIFO o algoritmo LIFO se basa en el formato que indica que la última solicitud en entrar debe ser la primera que se atienda y salga. Esto es porque se asume que las peticiones que llevan en cola más tiempo se han ido, por lo que atenderlas ya no es una prioridad. En cambio, las solicitudes nuevas son entendidas por el algoritmo como opciones con mayores probabilidades de ser atendidas por el emisor. (keepcoding., s.f.)

**FIFO: First-In, First-Out**

El patrón FIFO o programa FIFO, por su parte, toma como base el principio de que la solicitud que llegue primero debe ser la que se atienda y salga primero, mientras que la petición más nueva es atendida al final de la cola. (keepcoding., s.f.)

**Cola**

El tipo de Dato COLA (queue) es una estructura de datos que organiza los datos de la siguiente manera: A partir de una dirección de memoria, los datos se almacenan sucesivamente como si fueran una colección ordenada de elementos. (sedici.unlp.edu.ar, s.f.)

**Tabla hash**

Una tabla hash o mapa hash es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada usando el nombre, número de cuenta o id. Funciona transformando la clave con una función hash en un hash, un número que la tabla hash utiliza para localizar el valor deseado. (Ingeneriaudb, s.f.)

**Cola de prioridad**

Es una de las clases que implementa esta interfaz y ordena los elementos en base a su orden natural, según lo especificado por el método comparó los elementos **Comparable**, o mediante un objeto **Comparator** que se suministra a través del constructor. Esta clase proporciona una funcionalidad que permite inserciones en orden y eliminaciones de la parte frontal (desencolar normalmente). (pcresumen, s.f.)

**TAD**

Un tipo abstracto de datos (TAD) es una colección de propiedades y de operaciones que se definen mediante una especificación que es independiente de cualquier representación. La abstracción se centra en la independencia de la representación. Esto permite al programador modificar la representación del TAD sin que esto afecte a su utilización. (openaccess, s.f.)

**Interfaz de usuario**

La interfaz de usuario es el medio por el cual una persona controla una aplicación de software o dispositivo de hardware. Es decir, el programa incluye controles gráficos que optimizan la experiencia de usuario al emplear un mouse o teclado, lo que posibilita la interacción con los procesadores para realizar un trabajo. (Lenis, s.f.)

**Tarea**

La palabra tarea es utilizada para referir una actividad o práctica que es demandada por diversos motivos como académicos, laborales, domésticos, etc. Generalmente estas actividades o prácticas se encuentran enmarcadas en un tiempo determinado o situación específica. En otras palabras, las tareas son obras o trabajos a desarrollar que se encuentran delimitadas por ciertas reglas (tiempo, espacio, formas, medios, etc.). (conceptodefinicion, s.f.)

**FASE 3: BÚSQUEDA DE SOLUCIONES CREATIVAS**

En esta problemática se pueden establecer soluciones como:

* Diseñar una interfaz de usuario intuitiva y atractiva que permita a los usuarios agregar, modificar y eliminar tareas y recordatorios de manera eficiente.
* Implementar una función de búsqueda avanzada que permita a los usuarios encontrar rápidamente tareas específicas en su lista.
* Incorporar la opción de asignar tareas a otros usuarios o colaboradores para una gestión compartida.
* Revisar y optimizar la estructura de la tabla hash para garantizar un acceso rápido y eficiente a las tareas y recordatorios.
* Evaluar la experiencia del usuario en la interfaz de usuario y asegurarse de que sea amigable y fácil de navegar.
* Revisar y optimizar la estructura de la tabla hash para garantizar un acceso rápido y eficiente a las tareas y recordatorios.
* Evaluar la experiencia del usuario en la interfaz de usuario y asegurarse de que sea amigable y fácil de navegar.
* Crear una pila de acciones: Implementa una estructura de datos tipo pila (LIFO) para realizar un seguimiento de las acciones del usuario.
* Registrar Acciones: Cada vez que el usuario realice una acción (agregar, modificar o eliminar una tarea), registra la acción en la pila con detalles como el tipo de acción y los detalles de la tarea afectada.
* Método para Deshacer: Implementa un método que permita deshacer la última acción realizada por el usuario. Este método desapilará la última acción de la pila y revertirá la acción correspondiente en función de la información almacenada en la pila.
* Uso del Método de Deshacer: En la interfaz de usuario, proporciona a los usuarios la opción de "Deshacer". Cuando seleccionen esta opción, llama al método deshacer(), que restaurará la última acción realizada por el usuario.

|  |
| --- |
| TAD <Max priority queue > |
| Max priority queue={size,comparator} |
| Inv: {comparator(a,b)= True} |
| Primitive Operations:  Createpriorityqueue(size): ->priorityqueue  enqueue(data) : priorityqueuexnodo -> void  Dequeue(): priorityqueue->Node  peek (): -> node  size(): -> data  Clear (): -> void |

**Max Priority Queque**

|  |
| --- |
| Createpriorityqueue(Size)  “Creates a new priority queque”  {pre: Size}  {pos: print priorityqueque data} |

|  |
| --- |
| enqueue(data)  “Adds data to queque”  {pre: data}  {pos: print priorityqueque data with enquque data } |

|  |
| --- |
| Dequeue()  “Removes data from queque ”  {pre: !=null}  {pos: print priorityqueque data without Dequeque data } |

|  |
| --- |
| peek ()  “print queque ”  {pre: !=null}  {pos: prints queque} |

|  |
| --- |
| size()  “Determines queque size”  {pre: !=null}  {pos: queque size} |

|  |
| --- |
| TAD <Stack > |
| Stack = {size, comparator} |
| Inv: {comparator(pila, elemento1, elemento2) = True} |
| Primitive Operations:  **CreateStack(size)**: ->stack  **push(data)**: Agrega un elemento a la pila.  **pop()**: Elimina y devuelve el elemento en la parte superior de la pila.  **peek()**: -> node  **size()**: -> data  **clear()**: -> void |

**Stack**

|  |
| --- |
| **CreateStack(size)**  "Create a new stack."  {pre: Size}  {pos: "Print the empty stack structure.} |

|  |
| --- |
| **push(data)**  “"Add data to the stack."”  {pre: data}  {pos "Print the stack structure with data added to the top."} |

|  |
| --- |
| **pop()**  “Removes and returns the element at the top of the stack”  {pre: !=null}  {pos: "Print the stack structure without the removed element.} |

|  |
| --- |
| peek ()  “Returns the element at the top of the stack without removing it.”  {pre: !=null}  {pos: print the element at the top of the stack} |

|  |
| --- |
| size()  “Determines stack size”  {pre: N/A}  {pos: “Print the current stack size”} |

|  |
| --- |
| **clear()**  “empty the stack”  {pre: !=null  { pos: “Print the empty stack structure.” } |

|  |
| --- |
| TAD <Queue > |
| Queue={add,poll,size} |
| Inv: {comparator(a,b)= True ^ (Q={x1,x2,x3,x4,x5…xn} ^ Q.poll()=x1)} |
| Primitive Operations:  CreateQueue(): ->Queue  add(Node) : add.Node -> void  poll(): Queue->Node  size(): -> data  isEmpty (): -> boolean |

**Queue:**

|  |
| --- |
| CreateHashTable(add,poll,size)  “Creates a new queue”  {pre add,poll,size }  {pos: String “queue created”} |

|  |
| --- |
| add(Node)  “adds a node to the queue”  {pre: Node!=null y queue !=null}  {pos: String “added”} |

|  |
| --- |
| poll ()  “takes the first element out of the queue”  {pre: queue ¡= null}  {pos: Node} |

|  |
| --- |
| size ()  “find and return the queue size”  {pre: queue !=null }  {pos: prints size} |

|  |
| --- |
| isEmpty()  “determines if the queue is empty”  {pre: queue! = null}  {pos: boolean} |

Hashtable

|  |  |  |
| --- | --- | --- |
| **TablaHash = [L0, L1, L2, ..., LN-1], donde Li es una lista enlazada que contiene todas las tuplas (ki, vi) tales que h(ki) = i, para todo i = 0, 1, 2, ..., N-1.** | | |
| ∀ i j ∈ [0, N-1], i ≠ j ⇒ (ki ≠ kj) ∨ (h(ki) ≠ h(kj)) ∨ (ki = kj ∧ nodes[h(ki)] contiene (ki, vi) ∧ nodes[h(kj)] contiene (kj, vj)) | | |
| **Operaciones primitivas** | | |
| **Método** | **Entradas** | **Salidas** |
| HashTable |  | HashTable |
| insert | HashTable x K key x V value | HashTable |
| searchValue | HashTable x Node<K, V> node x K key | V value |
| delete | HashTable x Node<K, V> node x K key | HashTable |

|  |
| --- |
| **Nombre: HashTable()** |
| Descripción: Crea un objeto de tipo HashTable |
| Pre: True |
| Post: HashTable |

|  |
| --- |
| **Nombre: insert()** |
| Descripción: Añade un valor a la tabla hash con una key establecida. |
| Pre: True |
| Post: HashTable |

|  |
| --- |
| **Nombre: searchValue()** |
| Descripción: Devuelve el valor de un objeto con la clave buscada |
| Pre: HashTable != Ø |
| Post: Node<K, V> goal |

|  |
| --- |
| **Nombre: delete()** |
| Descripción: Elimina un valor asociado con una clave específica de la tabla |
| Pre: HashTable.size() > 0 |
| Post: HashTable |

Heap

|  |  |  |
| --- | --- | --- |
| **Objeto abstracto: Heap = {A[0], A[1], ..., A[n]}, donde A[i] es el valor del nodo en la posición i del árbol.** | | |
| A[0, n-1], donde el elemento en A[i] tiene dos hijos en A[2i+1] y A[2i+2] | | |
| **Método** | **Entradas** | **Salidas** |
| Heap() | **-** | Heap |
| maxHeapify() | Heap x arr | Heap |
| getHeap() | - | Heap arr |
| heapSort() | Heap x arr | Heap |
| getDad() | Heap x arr x int i | Node<K, V> |
| buildMaxHeap() | Heap x arr | Heap |
| getRight() | Heap x arr x int i | Node<K, V> |
| getLeft() | Heap x arr x int i | Node<K, V> |

|  |
| --- |
| **Nombre: Heap()** |
| **Descripción: Crea un objeto de tipo Heap** |
| **Pre: -** |
| **Post: Heap** |
|  |
| **Nombre: maxHeapify()** |
| **Descripción: Organiza el heap de tal forma que el nodo actual sea mayor o igual que sus hijos** |
| **Pre: Heap** |
| **Post: Heap tal que A[2i] < arr[i] y arr[2i + 1] > arr[i]** |
|  |
| **Nombre: heapSort()** |
| **Descripción: Ordena un arreglo de elementos (heap) en orden ascendente** |
| **Pre: Heap** |
| **Post: arr[i] ≤ arr[j] para todo i<j** |

|  |
| --- |
| **Nombre: getDad()** |
| **Descripción: Esto nos ayuda a encontrar el padre de un valor** |
| **Pre: i pertence a el arreglo del heap** |
| **Post: Node<K, V>** |
|  |
| **Nombre: getLeft()** |
| **Descripción: Retorna la posición del hijo izquierdo del nodo actual** |
| **Pre: i pertence a el arreglo del heap** |
| **Post: Node<K, V>** |
|  |
| **Nombre: getRight()** |
| **Descripción: Retorna la posición del hijo derecho del nodo actual** |
| **Pre: i pertence a el arreglo del heap** |
| **Post: Node<K, V>** |
|  |
| **Nombre: getHeap()** |
| **Descripción: Retorna el arreglo que representa el heap** |
| **Pre: i <= Size(arr)** |
| **Post: Heap arr** |

# Referencias

Cantador, I. (s.f.). *Escuela Politécnica Superior*. Obtenido de https://www.cartagena99.com/recursos/alumnos/apuntes/prog2-tema2-Pilas-1617.pdf

Ingeneriaudb. (s.f.). Obtenido de https://www.udb.edu.sv/udb\_files/recursos\_guias/informatica-ingenieria/programacion-con-estructuras-de-datos/2020/i/guia-8.pdf

keepcoding. (s.f.). *keepcoding.* Obtenido de https://keepcoding.io/blog/tecnicas-lifo-y-fifo-gestion-de-colas/

Lenis, A. (s.f.). Obtenido de https://blog.hubspot.es/website/interfaz-usuario#que-es

openaccess. (s.f.). Obtenido de https://openaccess.uoc.edu/bitstream/10609/9122/1/Intro\_TADs.pdf

pcresumen. (s.f.). *pcresumen*. Obtenido de https://www.pcresumen.com/menu-software/35-lenguajes-de-programacion/java/98-la-clase-priorityqueue-y-la-interfaz-queue#:~:text=PriorityQueue%20es%20una%20de%20las,suministra%20a%20trav%C3%A9s%20del%20constructor

sedici.unlp.edu.ar. (s.f.). Obtenido de http://sedici.unlp.edu.ar/bitstream/handle/10915/33386/Documento\_completo.pdf?sequence=3