**Paso 1. Identificación del Problema**

***Identificación de necesidades y síntomas***

- Los usuarios necesitan un sistema de gestión de tareas y recordatorios.

- El sistema debe permitir a los usuarios agregar, modificar y eliminar tareas y recordatorios.

- El sistema debe permitir a los usuarios ingresar en el sistema la información de sus tareas y recordatorios: título, descripción, fecha límite, prioridad, etc.

- Los usuarios deben tener la posibilidad de ver una lista de todas las tareas y recordatorios, ordenados por fecha límite o prioridad

- El sistema debe permitir a los usuarios gestionar sus tareas de acuerdo a un orden de prioridad. Las tareas más importantes se deben atender primero.

- Los usuarios pueden también agregar tareas sin una prioridad definida, las cuales se deben atender según su orden de llegada.

- El sistema debe permitir a los usuarios deshacer acciones que hayan realizado dentro del sistema, dándoles la posibilidad de revertir cambios.

**Identificación y definición concreta sin ambigüedad del proble**ma

Los usuarios requieren de un sistema de gestión de tareas y recordatorios que les permita agregar, modificar y eliminar tareas y recordatorios.

**Requerimientos funcionales**

**TABLA DE ESPECIFICACIÓN DEL PROBLEMA DE INGENIERÍA DE SOFTWARE, identificando los siguientes elementos**

\*En este caso, no se especificó cuál sería la empresa o la persona que mandó a hacer el programa ni el público objetivo que va a hacer uso de las funcionalidades del sistema. Pero podríamos decir por ejemplo que el cliente es la universidad y los usuarios serán los estudiantes, por lo que sería un sistema diseñado por la universidad para ayudar a los estudiantes a gestionar mejor su tiempo.

|  |  |
| --- | --- |
| CLIENTE | \* |
| USUARIO | \* |
| REQUERIMIENTOS FUNCIONALES | RF1: Agregar tarea  RF2: Modificar tarea  RF3: eliminar tarea  RF4: Agregar un recordatorio  RF5: Modificar un recordatorio  RF6: eliminar un recordatorio  RF7: Organizar tareas según prioridad  RF8: deshacer acciones. |
| CONTEXTO DEL PROBLEMA | Los usuarios requieren de un sistema de gestión de tareas y recordatorios que les permita agregar, modificar y eliminar tareas y recordatorios. |
| REQUERIMIENTOS NO FUNCIONALES |  |

**Tabla de análisis de requerimientos funcionales (Nota: Una tabla por cada requerimiento funcional)**

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF1: Agregar tarea | | |
| Resumen | El usuario ingresa la información de una nueva tarea que va a agregar a su sistema de gestión. Se recibe el título de la tarea, su descripción, la fecha límite y, si aplica, su prioridad. Como grupo hemos establecido que la prioridad a manejar será de 1 a 4. Siendo 1 lo menos prioritario y 4 lo más prioritario. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Titulo | String | Debe tener 3 caracteres mínimo |
| Descripción | String | Debe tener 3 caracteres mínimo |
| Fecha Limite | Int | Debe tener 10 caractres. . No puede ser menor a la fecha actual. |
| prioridad | Boolean | Debe seleccionar si es prioritario o no |
|  | Valor de prioridad | int | Valor entre 1 y 4 |
| Actividades generales necesarias para obtener los resultados | 1. Ingresar Titulo 2. Ingresar Descripción 3. Ingresa fecha limite 4. Indica si es prioritaria 5. Indica nivel de prioridad, si aplica. | | |
| Resultado o postcondición | Se guardan todos los datos registrados y se crea una nueva tarea en el sistema. | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| confirmacionCreacion deTarea. | Boolean | Ingresar todos los datos de acuerdo con los tipos de variable |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF2: Modificar tarea | | |
| Resumen | El usuario modifica la información de una tarea. Para ello selecciona una tarea de la lista de tareas, posteriormente indica qué información de la tarea desea cambiar e ingresa los datos por los cuales quiere reemplazar la anterior información. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| selección | int | El valor ingresado debe estar en el rango de las posibles opciones. |
| Selección de la Información a modificar | int | El valor ingresado debe estar en el rango de las posibles opciones. |
| Valor de la información a reemplazar | String |  |
| Actividades generales necesarias para obtener los resultados | 1. El usuario selecciona una tarea de la lista de tareas. 2. Indica qué quiere modificar. 3. Ingresa la información. 4. Se guarda el cambio. | | |
| Resultado o postcondición | Guardar todos los cambios registrados. | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| confirmacionCambioRealizado. | Boolean |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF3: Eliminar una tarea | | |
| Resumen | Para que el usuario pueda eliminar una tarea el sistema le muestra la lista de tareas, el usuario debe indicar la tarea que va a eliminar de acuerdo a la lista que se le muestra. Una vez hecho esto el sistema elimina la tarea de la lista de tareas. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| selección | int | El valor ingresado debe estar en el rango de las posibles opciones. |
| Actividades generales necesarias para obtener los resultados | 1. Ingresar selección de tarea 2. Confirmar la eliminación 3. Eliminar tarea | | |
| Resultado o postcondición | Se elimina la tarea de la lista de tareas. | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| confirmacionEliminacionTareal. | Boolean | Ingresar todos los datos de acuerdo con los tipos de variable |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF4: Agregar un recordatorio | | |
| Resumen | El usuario ingresa la información de un nuevo recordatorio que va a agregar a su sistema de gestión. Se recibe el título, descripción, y la fecha límite. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Titulo | String | Debe tener 3 caracteres mínimo |
| Descripción | String | Debe tener 3 características mínimo |
| Fecha Limite | Int | Debe tener 10 caractres. No puede ser menor a la fecha actual. |
| Actividades generales necesarias para obtener los resultados | 1. Ingresar Titulo 2. Ingresar Descripción 3. Ingresa fecha limite | | |
| Resultado o postcondición | Se guardan todos los datos registrados y se crea un nuevo recordatorio en el sistema. | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| confirmacionAgregarRecordatorio. | Boolean | Ingresar todos los datos de acuerdo con los tipos de variable |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF6: Modificar Recordatorio | | |
| Resumen | El usuario modifica la información de un recordatorio. Para ello selecciona uno de la lista, posteriormente indica qué información desea cambiar e ingresa los datos por los cuales quiere reemplazar la anterior información. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| selección | int | El valor ingresado debe estar en el rango de las posibles opciones. |
| Selección de la Información a modificar | int | El valor ingresado debe estar en el rango de las posibles opciones. |
| Valor de la información a reemplazar | String |  |
| Actividades generales necesarias para obtener los resultados | 1. El usuario selecciona un recordatorio. 2. Indica qué quiere modificar. 3. Ingresa la información. 4. Se guarda el cambio. | | |
| Resultado o postcondición | Guardar todos los cambios registrados. | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| confirmacionModificarRecordatorio. | Boolean | Ingresar todos los datos de acuerdo con los tipos de variable |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF6: Eliminar un recordatorio | | |
| Resumen | Para que el usuario pueda eliminar un recordatorio el sistema le muestra la lista, el usuario debe indicar el recordatorio que va a eliminar. Una vez hecho esto el sistema elimina el recordatorio de la lista. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| selección | int | El valor ingresado debe estar en el rango de las posibles opciones. |
| Actividades generales necesarias para obtener los resultados | 1.Ingresar selección  2.Confirmar la eliminación  3.Eliminar tarea | | |
| Resultado o postcondición | Se elimina el recordatorio de la lista. | | |
| Salidas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| confirmacionEliminacionrRecordatoriol. | Boolean | Ingresar todos los datos de acuerdo con los tipos de variable |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF7: Organizar tareas según prioridad | | |
| Resumen | Cuando se va a mostrar la lista de tareas y recordatorios en pantalla, las tareas deben estar ordenadas con respecto a su prioridad, mostrando primero las tareas más importantes. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
|  | Imprimir lista | int | El valor ingresado debe estar en el rango de las posibles opciones. |
| Actividades generales necesarias para obtener los resultados | 1. El usuario indica que quiere ver su lista de tareas y recordatorios. | | |
| Resultado o postcondición | Se muestra en pantalla la lista de tareas y recordatorios. | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
|  | ConfirmacionOrganizarTareas | Boolean | Ingresar los datos con las anteriores condiciones |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF8: Deshacer acciones | | |
| Resumen | El sistema debe tener la posibilidad de deshacer acciones. Igual que en un documento de Word, el usuario debe poder volver al estado anterior del sistema hasta que se devuelva al inicio. Por lo tanto, las acciones que el usuario haya realizado se deben revertir hasta el punto que el usuario indique. Para esto el usuario simplemente selecciona la opción *deshacer* del programa. El programa revierte una acción cada vez que el usuario selecciona la opción. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
|  | selección | int | El valor ingresado debe estar en el rango de las posibles opciones. (en este caso, la opción de deshacer) |
| Actividades generales necesarias para obtener los resultados | 1. El usuario indica que quiere deshacer acciones. | | |
| Resultado o postcondición | El programa se devuelve a un estado anterior, es decir, se revierte una acción realizada. Si el usuario había agregado una tarea, se elimina, si había eliminado una tarea, esta se vuelve a ingresar, y así con cada caso. | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
|  | ConfirmacionDeshacerAccciones | Boolean |  |

Paso 2. Recopilación de Información

Fuentes:

<https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/las-mejores-aplicaciones-de-gestion-de-proyectos/>

(IONOS es una empresa de software con más de 30 años de trayectoria en la industria)

<https://www.xataka.com/basics/14-mejores-aplicaciones-para-administrar-tus-tareas-tus-proyectos>

<https://www.ionos.es/startupguide/productividad/gestion-del-tiempo/>

<https://www.freecodecamp.org/news/hash-tables/>

<https://es.wikipedia.org/wiki/Lluvia_de_ideas>

Para la recopilación de información debemos buscar los conceptos más relevantes para nuestro problema. En este caso, vamos a empezar buscando cuáles son los sistemas o aplicaciones más usadas para la gestión del tiempo y tareas tanto en la vida cotidiana como para proyectos laborales. Las principales 3 que encontramos fueron:

1. **Todoist:** Una de sus características más importantes es que está disponible para casi todo tipo de dispositivo: dispositivos móviles, computadores y relojes inteligentes. En cuanto a su funcionalidad, se pueden crear tareas con descripciones y subtareas, eligiendo el día y la hora a la que deben ser terminadas. Tiene un sistema de etiquetas para catalogar estas tareas, y un sistema de prioridades en cuatro niveles para decidir cuáles de ellas son más urgentes. También se puede trabajar en equipo y crear tareas que se repiten con el tiempo.
2. **Trello:** Junto a Asana, Trello es uno de los servicios más populares para gestionar las tareas a la hora de trabajar en equipo o a nivel empresarial, aunque se puede usar también para proyectos personales. Destaca sobre todo por su interfaz, con una estructura de tableros y tarjetas con los que se pueden gestionar las tareas de una manera mucho más visual. Cada tablero es un proyecto y cada tarjeta dentro de un tablero representa una tarea a realizar. Dentro de cada tarjeta se pueden añadir descripciones, subtareas o adjuntar archivos.
3. **Microsoft To Do:** Después de haber comprado Wunderlist, una de las mejores aplicaciones para gestión de tareas, Microsoft diseñó este gestor de tareas. Tiene una interfaz limpia y sencilla, pudiendo crear varias tareas en ella, ponerles fechas y horas límite para su cumplimiento. Una de sus mejores características es una página principal que se llama *Mi Día*, que muestra todo lo que una persona tiene que hacer en el día presente.

No solamente es importante buscar aplicaciones de gestión de tareas, sino entender el contexto y el por qué es importante la tarea que debemos llevar a cabo, para entender el contexto y tener una mejor claridad en la lógica de negocio, investigamos acerca de la gestión del tiempo y su importancia:

La gestión del tiempo sirve para usar el tiempo disponible de la manera más eficiente posible. Dominar la gestión del tiempo implica varias ventajas:

* Mayor productividad
* Reducción del estrés
* Más tiempo para uno mismo y la familia
* Mayor equilibrio entre el trabajo y la vida privada

Estas ventajas son, al final de cuentas, lo que esperamos lograr con el programa que queremos diseñar.

Para la resolución del problema planteado, es necesario también tener claridad sobre las estructuras que vamos a implementar para el desarrollo de la solución, pasamos a dar una información breve pero detallada de las estructuras más importantes que vamos a usar para la solución del problema:

# Hash Table

Una tabla hash, también conocida como mapa hash, es una estructura de datos que mapea claves a valores. Funciona en el concepto de hashing, donde cada clave es traducida por una función hash en un índice distinto en un arreglo. El índice funciona como un lugar de almacenamiento para el valor correspondiente.

puntos clave a tener en cuenta:

Función Hash: Una función que traduce las claves a índices de un arreglo se conoce como una función hash. Las claves deben ser distribuidas uniformemente a través del arreglo por una buena función hash para reducir las colisiones y asegurar velocidades rápidas de búsqueda.

Factor de Carga: El factor de carga de una tabla hash se determina por cuántos elementos se guardan allí en relación con cuán grande es la tabla. Si el factor de carga es alto, la tabla puede estar desordenada y tener tiempos de búsqueda más largos y colisiones.

Resolución de Colisiones: Las colisiones ocurren cuando dos o más claves apuntan al mismo índice del arreglo. Existen varias técnicas para resolver colisiones, como el encadenamiento y la dirección abierta.

Priority Queue

Una cola de prioridad es un tipo de estructura de datos, similar a una cola regular o una pila. Cada elemento en una cola de prioridad tiene una prioridad asociada. Los elementos con alta prioridad se atienden antes que los elementos con baja prioridad.

En una cola de prioridad, cada elemento tiene un valor de prioridad asociado. Cuando se agrega un elemento a la cola, se inserta en una posición basada en su valor de prioridad. Por ejemplo, si se agrega un elemento con un valor de prioridad alto a una cola de prioridad, es insertado cerca del frente de la cola, mientras que un elemento con un valor de prioridad bajo es insertado cerca del final.

Las operaciones típicas que soporta una cola de prioridad son:

Inserción: Cuando se va a insertar un nuevo elemento en una cola de prioridad, se evalúa desde arriba hacia abajo y de izquierda a derecha (haciendo la abstracción desde el arreglo a un árbol). Sin embargo, si el elemento no está en el lugar correcto, entonces se compara con el nodo padre. Si el elemento no está en el orden correcto, los elementos se intercambian.

Eliminación: Se eliminará el elemento que tenga la máxima prioridad primero. Por lo tanto, se elimina el nodo raíz de la cola. También se puede implementar de manera que el elemento con la menor prioridad se elimine primero. Todo depende de la implementación y el objetivo que se busca. En nuestro caso queremos que las tareas con mayor prioridad se atiendan primero.

Las colas de prioridad son a menudo utilizadas en sistemas en tiempo real, donde el orden en que los elementos son procesados puede tener consecuencias significativas.

# Pila (Stack)

Es una estructura de datos lineal que sigue el principio LIFO (Last In, First Out), lo que significa que el último elemento en entrar es el primero en salir.

Las pilas permiten básicamente dos operaciones: apilar (push) y desapilar (pop). Al apilar un elemento, se agrega al tope de la pila. Al desapilar, se elimina el elemento que está en el tope de la pila.

Las pilas se utilizan en el manejo de las llamadas a funciones, donde cada vez que se llama a una función se apila en la pila y cuando termina la función se desapila. También se utilizan para revertir una operación (por ejemplo, la función de deshacer en un editor de texto) y en la evaluación de expresiones matemáticas en notación polaca inversa.

Las pilas son simples y fáciles de implementar. Las operaciones de inserción y eliminación son rápidas y eficientes. Además, las pilas son útiles para resolver problemas que requieren una búsqueda exhaustiva y un seguimiento de un camino o ruta, como en el backtracking.

En una pila, solo es posible acceder al elemento en la cima de la pila, lo que puede representar una desventaja si se necesita acceder a elementos intermedios. Además, las pilas no están diseñadas para la búsqueda eficiente de elementos específicos.

Una pila es una estructura de datos esencial para la programación y se utiliza en una variedad de problemas y aplicaciones.

# Cola (Queue)

Es una estructura de datos lineal que sigue el principio FIFO (First In, First Out), lo que significa que el primer elemento en entrar es el primero en salir.

Las operaciones básicas que se pueden realizar en una cola son: encolar (enqueue) y desencolar (dequeue). Al encolar un elemento, se agrega al final de la cola, y al desencolar se elimina el elemento que está en el frente de la cola.

La cola se utiliza en diversas aplicaciones como:

* En sistemas operativos, la cola se utiliza para almacenar procesos en espera de ser ejecutados.
* En redes de computadoras, las colas se utilizan para almacenar paquetes en espera de ser transmitidos.
* En sistemas de impresión, las colas se utilizan para almacenar documentos en espera de ser impresos.

Las ventajas de las colas incluyen su orden de procesamiento específico, su escalabilidad y su utilidad en la simulación y planificación de eventos.

Sin embargo, también tienen algunas desventajas. Por ejemplo, solo es posible acceder al primer elemento en la cola, lo que puede ser una limitación si se necesita acceder a elementos intermedios. Además, las colas no están diseñadas para la eliminación eficiente de elementos intermedios.

**Paso 3. Búsqueda de Soluciones Creativas**

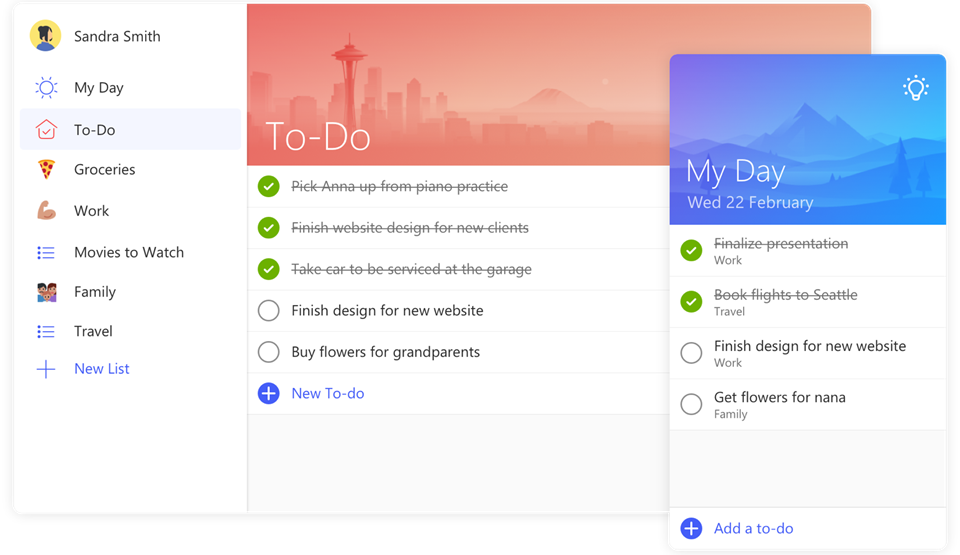
Para la búsqueda de soluciones creativas empleamos la lluvia de ideas, la cual es una herramienta de trabajo grupal que facilita el surgimiento de nuevas ideas sobre un tema o problema determinado. La lluvia de ideas es una técnica de grupo para generar ideas originales en un ambiente creativo.

Alternativa 1. Agenda física y google calendar



Lo primero que se nos ocurrió como grupo fue hacer una agenda física para la gestión de las tareas. El estudiante puede llevar un registro en uno de sus cuadernos de sus tareas, asignándoles a las tareas prioritarias un criterio definido por él mismo. Para los recordatorios puede utilizar la aplicación de google calendar o cualquier otra aplicación similar.

Alternativa 2. Uso de cualquiera de las aplicaciones mencionadas



En la fase de recopilación de información mencionamos algunas aplicaciones existentes que ayudan a gestionar el tiempo y las tareas. El estudiante o usuario puede hacer uso de cualquiera de estas herramientas, con seguridad le será de ayuda pues estas aplicaciones son conocidas por su ayuda en la gestión de tareas.

Alternativa 3. Desarrollo de una aplicación web

En la recopilación de información nos dimos cuenta de que una de las características que las personas más aprecian de una aplicación que les ayude en su gestión de tareas, es poder acceder a dicha aplicación desde gran variedad de dispositivos y desde cualquier lugar, es por esto que pensamos en el desarrollo de una aplicación web; una persona podría acceder a ella desde su teléfono o computador e incluso podría acceder desde diferentes lugares como una oficina, una biblioteca, entre otros.

Alternativa 4. Desarrollo de un sistema sencillo en Java

Desarrollo de un sistema que cumpla con los requisitos enunciados, usando los conocimientos que hemos adquirido en la universidad.

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

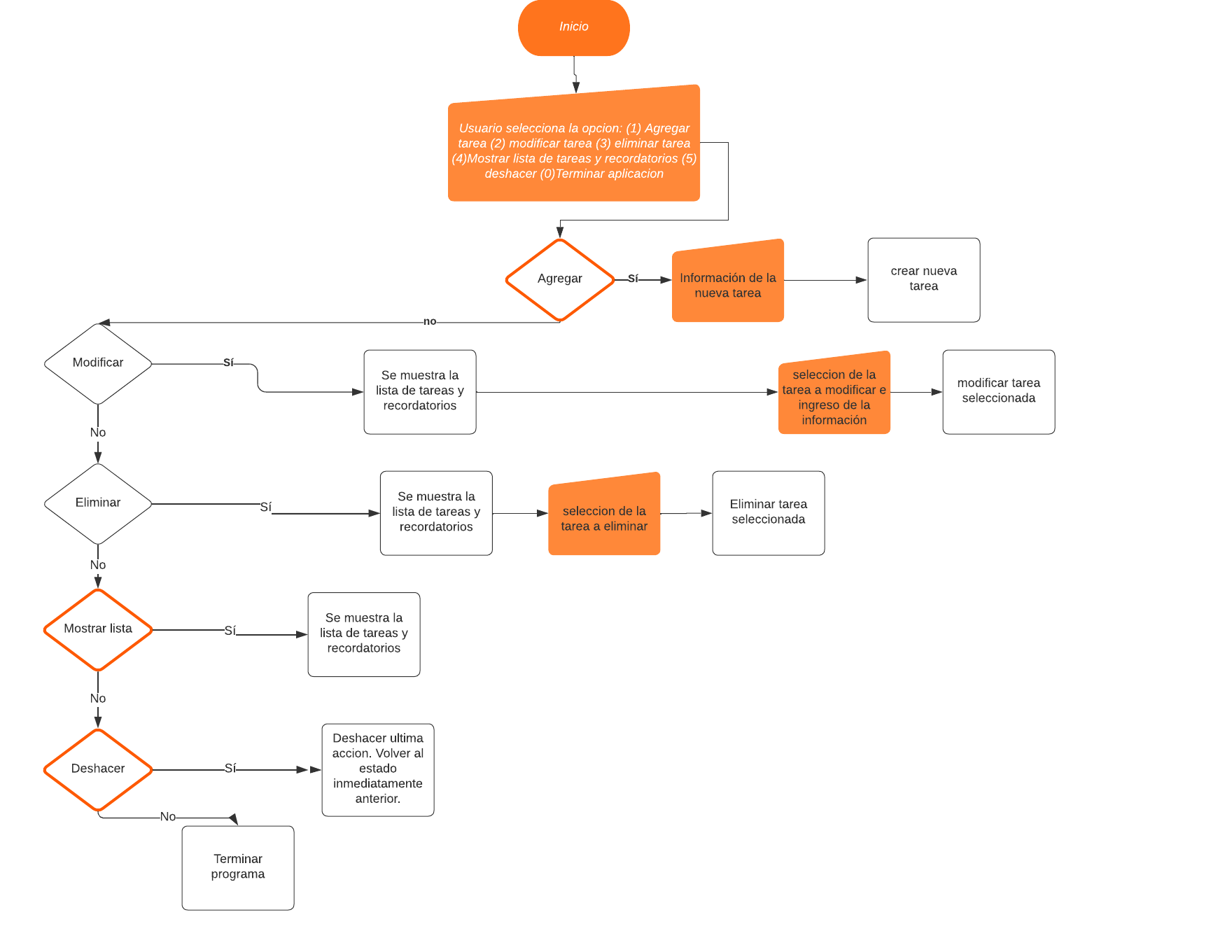
Descartamos las alternativas 1 y 2, pues a pesar de ser prácticas, sencillas y poderosas, no requieren realmente ningún esfuerzo por parte de nosotros los estudiantes, y puesto que el objetivo de este trabajo es poner a prueba nuestros conocimientos, no tendría cabida usarlas como solución para este caso. Descartamos también la alternativa 3 puesto que no contamos con los conocimientos necesarios para desarrollar una aplicación web con todo lo requiere: diseño y lógica del front-end, manejo de los datos desde el back-end, deployment, entre muchos otros. Por lo tanto, vamos a revisar cuidadosamente la alternativa 4:

Alternativa 4. Desarrollo de un sistema sencillo en Java

* Para desarrollar un programa en Java necesitamos saber qué estructuras de datos vamos a implementar para poder dar solución a lo que nos pide el enunciado. Es por esto que vamos a utilizar las estructuras que investigamos en la fase de recopilación de información, pues estas nos garantizan un uso eficiente de los recursos.
* En la tabla hash se almacenarán las tareas, la pila servirá para el método de deshacer, la cola de prioridad para gestionar las tareas que tengan prioridad, y la cola para aquellas que no.
* Se debe mostrar al usuario la lista de tareas (según su prioridad) y recordatorios.

**Paso 5. Evaluación y Selección de la Mejor Solución**

Diagrama de flujo del sistema:



Criterios

* Criterio A. Estética. La solución es agradable a la vista del usuario:
  + [2] Tiene un buen diseño.
  + [1] No es agradable.
* Criterio B. Completitud. Cumple con las funcionalidades requeridas.
  + [3] cumple con todas las funcionalidades que los usuarios necesitan.
  + [2] cumple con algunas, pero no con todas las funcionalidades requeridas.
  + [1] No cumple con las funcionalidades que se piden.
* Criterio C. Exactitud. La solución cumple con las funcionalidades empleando las estructuras de datos pertinentes.
  + [3] emplea todas las estructuras de datos pertinentes.
  + [2] emplea algunas, pero no todas.
  + [1] no emplea las estructuras pertinentes.
* Criterio D. Portabilidad. El sistema se puede usar en una gran variedad de dispositivos.
  + [2] se puede acceder desde muchos dispositivos.
  + [1] Es muy limitado y se puede acceder desde pocos dispositivos.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Criterio A | Criterio B | Criterio C | Criterio D | Total |
| Alternativa 4. Desarrollo de un sistema sencillo en Java | 1 | 3 | 3 | 1 | 8 |

La selección que se decidió tomar fue la Alternativa 4

Definición de todos los TAD de las estructuras a utilizar

|  |
| --- |
| TAD Hash Table |
| Una tabla hash es una estructura de datos que implementa una función de mapeo donde es el conjunto de claves y es el conjunto de valores. Para hacer esto, que es el comportamiento de un diccionario, se realizan dos operaciones: primero una función hash donde es un conjunto finito de enteros positivos. Es decir, la función hash convierte la clave en un número entero. En segundo lugar, para poder obtener los valores que se han insertado en la tabla hash, se hace una búsqueda mediante la clave asociada a dicho valor. La búsqueda se hace ingresando la clave a la función hash y utilizando el valor que retorna la función para.  ---  Una tabla hash es una estructura de datos que almacena valores en un arreglo en forma de diccionario. Es decir, cuando se ingresa un objeto a la tabla hash, se hace mediante una llave y un valor, la llave se utiliza para determinar la posición que el objeto tendrá en el arreglo mediante una función hash, y lo que se guardará en dicha posición será el valor del objeto. |
| El invariante de una tabla hash es que para una llave k, siempre se obtiene el mismo valor del código hash, independientemente de cuántas veces se realice la operación.  Teniendo k una clave, h la función hash, i una posición en el arreglo:  {inv:{ para todo k se cumple que cuando la ingreso a la función hash obtengo siempre la misma posición }}  ¿podemos considerar al factor de carga como un invariante?  El factor de carga se define como el número de elementos en la tabla hash dividido por el número total de espacios o “buckets” en la tabla. En otras palabras, es el número promedio de elementos por bucket. Si el factor de carga es demasiado alto, significa que hay demasiados elementos en la tabla hash en relación con el número de buckets, lo que puede llevar a un mayor número de colisiones y, por lo tanto, a un aumento en el tiempo de búsqueda.  Por otro lado, si el factor de carga es demasiado bajo, significa que hay muchos buckets vacíos, lo que puede resultar en un uso ineficiente del espacio.  Por lo tanto, mantener un factor de carga óptimo es crucial para garantizar un equilibrio entre el tiempo (velocidad de las operaciones de búsqueda, inserción y eliminación) y el espacio (memoria utilizada por la tabla hash). |
|  |

|  |
| --- |
| TAD Priority Queue |
| ⟨ob𝑗𝑒𝑡𝑜 𝑎𝑏𝑠𝑡𝑟𝑎𝑐𝑡𝑜⟩: 𝑃𝑟𝑖𝑜𝑟𝑖𝑡𝑦𝑄𝑢𝑒𝑢𝑒=⟨𝑆 (𝑣, 𝑝), 𝑡𝑜𝑝, 𝑠𝑖𝑧𝑒⟩ |
| {Inv.: 0 ≤ 𝑛 ∧ 𝑠𝑖𝑧𝑒(𝑃𝑟𝑖𝑜𝑟𝑖𝑡𝑦𝑄𝑢𝑒𝑢𝑒) = 𝑛 ∧ 𝑡𝑜𝑝 = 𝑛 ∧ ∀ (𝑣\_1, 𝑝\_1), (𝑣\_2, 𝑝\_2) ∈ 𝑆: 𝑝\_1 ≥ 𝑝\_2} |
| Constructor operaciones  PriorityQueue  **Modifier opearations:**  Enque void  Dequeue  Front  **Analizing operations**  isEmpty boolean |

|  |
| --- |
| PriorityQueue  Builds an empty priorityQueue.  {pre: none}  {post: Queue q = Ø} |

|  |
| --- |
| Enqueue  An element is added to the queue with a priority (a numeric value).  {pre: 𝑃𝑟𝑖𝑜𝑟𝑖𝑡𝑦𝑄𝑢𝑒𝑢𝑒 pq =⟨e\_("(v\_1, p\_1" )) e\_("(v\_2, p\_" 2))…e\_("(v\_n, p\_" n))⟩ ∧element e or pq="Ø" ⟩ ∧element e}  {post: PriorityQueue pq = =⟨e\_("(v\_1, p\_1" )) e\_("(v\_2, p\_" 2))…e\_("(v\_n, p\_" n)), e⟩ ∨q= ⟨e⟩} |

|  |
| --- |
| **Dequeue:**  Extracts the element in Queue q’s front.  {pre: 𝑃𝑟𝑖𝑜𝑟𝑖𝑡𝑦𝑄𝑢𝑒𝑢𝑒 pq i.e pq = ⟨ ⟩  {post: PriorityQueue pq = =⟨ ⟩ } |

|  |
| --- |
| **isEmpty:**  **Determines if the PriorityQueue pq is empty or not.**  **{pre: 𝑃𝑟𝑖𝑜𝑟𝑖𝑡𝑦𝑄𝑢𝑒𝑢𝑒 pq}**  **{post: true if s = Ø or false if s ≠ Ø}** |

|  |
| --- |
| TAD Stack |
| ⟨objeto abstracto⟩:Stack=⟨⟨e\_(1, ) e\_(2, ) e\_(3…,) e\_n ⟩, top⟩ |
| inv: Invariant of STACK: 0≤n ∧size(Stack)=n∧top=en } |
| **Constructor de operación**  Stack  **Modificaciones de operaciones**  Push void  Pop T  **Análisis de operaciones**  Size int  Top T  IsEmpty Boolean  Peek T |

|  |
| --- |
| Stack ()  Builds an empty stack.  {pre:}  {post: Stack s = Ø} |

|  |
| --- |
| push ()  Adds the new element e to stack s.  {pre: Stack s = [Ecuación]}  {pos: Stack s = [Ecuación]} |

|  |
| --- |
| Pop  Extracts from the stack s, the most recently inserted element.  {pre: Stack s ≠ Ø}  {post: Stack s = [Ecuación]} |

|  |
| --- |
| Top  Recovers the value of the element on the top of the stack.  {pre: Stack s ≠ Ø}  {post: Element en } |

|  |
| --- |
| isEmpty  Determines if the stack s is empty or not.  {pre: Stack s}  {post: true if s = Ø or false if s ≠ Ø} |

|  |
| --- |
| TAD MaxHeap |
| Objeto Abstracto de un MaxHeap:  - Elementos: Un MaxHeap contiene una colección de elementos e\_1, e\_2, e\_3, ..., e\_n. Cada elemento es de un tipo genérico T y se almacena en una estructura de datos interna, generalmente un arreglo.  - Top (tope): El "top" o elemento superior del MaxHeap es el elemento de máxima prioridad según la relación de orden definida por la comparación de los elementos T. |
| Invariante del MaxHeap:  - 0 <= n: El número de elementos en el MaxHeap siempre es mayor o igual a cero.  - size(MaxHeap) = n: El tamaño del MaxHeap es igual al número de elementos n almacenados en él.  - top = e\_n: El elemento en la parte superior del MaxHeap es igual al último elemento añadido, lo que garantiza que sea el elemento de máxima prioridad en el MaxHeap.  Esta invariante asegura que el MaxHeap se mantiene correctamente ordenado y que el elemento en la parte superior es el de máxima prioridad en función de la relación de orden definida por T. |
| **Constructores:**  **1. Constructor Stack():**  **Modificaciones de Operaciones:**  **2. Push(T elemento) - void:**  **3. Pop() - T:**  **Análisis de Operaciones:**  **4. Size() - int:**  **5. Top() - T**  **6. IsEmpty() - boolean:**  **7. Peek() - T:** |

|  |
| --- |
| Operación isEmpty en MaxHeap:  - La operación isEmpty determina si el MaxHeap 'heap' está vacío o no.  Precondición:  - Se asume que el MaxHeap 'heap' está definido y existe como entrada.  Método insert (Push):  - Antes de insertar un elemento en el MaxHeap, se verifica si el MaxHeap está vacío usando la operación isEmpty.  - isEmpty(heap) devuelve true si 'heap' no contiene elementos, es decir, si 'heap' está vacío (size(heap) = 0).  - isEmpty(heap) devuelve false si 'heap' contiene uno o más elementos, es decir, si 'heap' no está vacío (size(heap) > 0).  Método extractMax (Pop):  - Antes de extraer el elemento máximo del MaxHeap, se verifica si el MaxHeap está vacío usando la operación isEmpty.  - isEmpty(heap) devuelve true si 'heap' no contiene elementos, es decir, si 'heap' está vacío (size(heap) = 0).  - isEmpty(heap) devuelve false si 'heap' contiene uno o más elementos, es decir, si 'heap' no está vacío (size(heap) > 0).  Método size:  - El método size devuelve el número de elementos en el MaxHeap, pero no verifica si el MaxHeap está vacío o no.  Método heapMaximum (Top):  - El método heapMaximum devuelve el elemento máximo en el MaxHeap, pero no verifica si el MaxHeap está vacío o no.  Método peek:  - El método peek devuelve el elemento en la parte superior del MaxHeap sin eliminarlo, pero no verifica si el MaxHeap está vacío o no. |

PSEUDOCODIGO:

Clase Cola<T> Implementa ICola<T>

Privado Nodo<T> primero

Privado Nodo<T> ultimo

Privado Entero tamaño

Clase Interna Nodo<T>

Privado T elemento

Privado Nodo<T> siguiente

Método Cola()

primero <- Nulo

ultimo <- Nulo

tamaño <- 0

Método Vacía() -> Booleano

Retornar primero == Nulo

Método Tamaño() -> Entero

Retornar tamaño

Método ConsultarFrente() -> T

Si Vacía() Entonces

Lanzar Excepción "Desbordamiento de la cola"

Sino

Retornar primero.elemento

Método Encolar(elemento: T)

NuevoNodo <- Crear un nuevo Nodo<T> con elemento y siguiente como Nulo

Si Vacía() Entonces

primero <- NuevoNodo

Sino

ultimo.siguiente <- NuevoNodo

ultimo <- NuevoNodo

tamaño <- tamaño + 1

Método Desencolar() -> T

Si Vacía() Entonces

Lanzar Excepción "Desbordamiento de la cola"

Sino

elemento <- primero.elemento

primero <- primero.siguiente

tamaño <- tamaño - 1

Si Vacía() Entonces

ultimo <- Nulo

Fin Si

Retornar elemento

Método Imprimir() -> Cadena

Cadena mensaje <- ""

Si primero == Nulo Entonces

mensaje <- "No hay tareas registradas"

Sino

mensaje <- ImprimirNodo(primero)

Fin Si

Retornar mensaje

Método ImprimirNodo(actual: Nodo<T>) -> Cadena

Cadena mensaje <- actual.elemento.ConvertirAString()

Si actual.siguiente != Nulo Entonces

mensaje <- mensaje + " -> " + ImprimirNodo(actual.siguiente)

Fin Si

Retornar mensaje

Class Pair <K, T> :

// Attributes

K key

T value

//Methods

K getKey ():

return key.

T getValue ():

return value.

void setValue (T value) :

value = value

void setKey (K key):

key = key

End.

Stack

Class Node <T>

// Attributes

K key

Node <K> next

// Methods

K getKey ():

return key.

void setKey (K key):

key = key

getNext ()

return Node<K>

void setNext (Node<K> next)

next = next

End.

Class Stack:

Stack<K>:

//Attributes

top: Node<K>

//Constructor

Stack ():

top = null

Push method to add a new item to the top of the stack.

push (key: K):

node = new Node<K>(key)

if (top! = null) then

node. SetNext(top)

top = node

Pop method to delete the top item from the stack and return its key value.

pop (): K

if (isEmpty ()) then

throw EmptyStackException.

pulled = top

top = top.getNext()

return pulled.getKey()

Top method to return the key value of the top item in the stack.

top (): K

if (isEmpty ()) then

throw EmptyStackException

return top.getKey()

isEmpty method to check if the stack is empty.

isEmpty (): boolean

return top == null

Fin Stack<K>.

HASH TABLE:

Class arrayGen <T> // También se usa para la priorityQueue

Attributes

- [] array: arreglo de tipo T

Methods:

- constructor (size: int):

- Create a new size array

- isEmpty (index: int): boolean

- Return true if the item in the index position is null, otherwise return false

- setElement (index : int, element : T) : void

- Assign the element element in the index position of the array

- getElement (index: entero): T

- Return the element to the index position of the array

- getSize (): int

- Return the array size

Clase TablaHash<K, V> Implementa ITablaHash<K, V>

Privado Arreglo de HashNode<K, V> tabla

Privado HashNode<K, V> eliminado

Privado Entero tamaño

Constante Decimal KNUTH <- (Raíz cuadrada de 5 - 1) / 2.0

// Tamaño == m

Método TablaHash(tamaño: Entero)

tamaño <- tamaño

tabla <- Nuevo Arreglo de HashNode<K, V> con tamaño elementos

Método funciónHash(objeto: K) -> Entero

// 1. Obtener la representación en cadena del objeto

// String info <- objeto.ConvertirACadena()

// 2. Calcular el valor hash utilizando el método de hashcode

Entero clave <- objeto.hashCode()

Si clave < 0 Entonces

clave <- clave \* (-1)

Fin Si

// 3. Aplicar el método de multiplicación para calcular el índice hash

Entero hash <- Entero(Piso(tamaño \* ((clave \* KNUTH) % 1)))

Retornar hash

Método agregar(llave: K, valor: V) -> Booleano

Entero dirección <- funciónHash(llave)

Booleano resultado <- Falso

HashNode<K, V> nuevoNodo <- Nuevo HashNode<K, V>(llave, valor)

// Manejo de colisiones con encadenamiento

Si tabla[dirección] == Nulo O tabla[dirección] == eliminado Entonces

tabla[dirección] <- nuevoNodo

resultado <- Verdadero

Sino

HashNode<K, V> actual <- tabla[dirección]

tabla[dirección] <- nuevoNodo

actual.setAnterior(nuevoNodo)

nuevoNodo.setSiguiente(actual)

resultado <- Verdadero

Fin Si

Retornar resultado

Método buscar(llave: K) -> V

Entero dirección <- funciónHash(llave)

V resultado <- Nulo

// Ubicarse en la posición de la tabla

Si tabla[dirección] != Nulo Entonces

resultado <- tabla[dirección].obtenerValor()

Fin Si

Retornar resultado

Método eliminar(llave: K) -> V

Entero dirección <- funciónHash(llave)

V resultado <- Nulo

// Ubicarse en la posición de la tabla

Si tabla[dirección] != Nulo Entonces

HashNode<K, V> actual <- tabla[dirección]

resultado <- actual.obtenerValor()

Si actual.obtenerSiguiente() != Nulo Entonces

actual <- actual.obtenerSiguiente()

actual.establecerAnterior(Nulo)

tabla[dirección] <- actual

Sino

tabla[dirección] <- eliminado

Fin Si

Fin Si

Retornar resultado

Método imprimir() -> Cadena

Cadena mensaje <- ""

Si tabla == Nulo Entonces

mensaje <- "No hay tareas registradas"

Sino

Para índice de 0 hasta tamaño - 1

Si tabla[índice] != Nulo Entonces

mensaje <- mensaje + imprimirNodo(tabla[índice])

Fin Si

Fin Para

Fin Si

Retornar mensaje

Método imprimirNodo(actual: HashNode<K, V>) -> Cadena

Cadena mensaje <- ""

V temp <- actual.obtenerValor()

mensaje <- temp.ConvertirACadena()

Si actual.obtenerSiguiente() != Nulo Entonces

mensaje <- mensaje + " -> " + imprimirNodo(actual.obtenerSiguiente())

Fin Si

Retornar mensaje

Método obtenerTamaño() -> Entero

Retornar tamaño

Método obtenerTabla() -> Arreglo de HashNode<K, V>

Retornar tablaHeap

Clase Cola<T>

Privado Nodo<T> primero

Privado Nodo<T> ultimo

Privado Entero tamaño

Clase Interna Nodo<T>

Privado T elemento

Privado Nodo<T> siguiente

Método Cola()

primero <- Nulo

ultimo <- Nulo

tamaño <- 0

Método Vacío() -> Booleano

Retornar primero == Nulo

Método Tamaño() -> Entero

Retornar tamaño

Método ConsultarFrente() -> T

Si Vacío() Entonces

Lanzar Excepción "Desbordamiento de la cola"

Sino

Retornar primero.elemento

Método Encolar(elemento: T)

NuevoNodo <- Crear un nuevo Nodo<T> con elemento y siguiente como Nulo

Si Vacío() Entonces

primero <- NuevoNodo

Sino

último.siguiente <- NuevoNodo

último <- NuevoNodo

tamaño <- tamaño + 1

Método Desencolar() -> T

Si Vacío() Entonces

Lanzar Excepción "Desbordamiento de la cola"

Sino

elemento <- primero.elemento

primero <- primero.siguiente

tamaño <- tamaño - 1

Si Vacío() Entonces

último <- Nulo

Retornar elemento

Clase MaxHeap<T extends Comparable<T>> Implementa IHeap<T>

Privado ArrayList<T> heap

Privado ArrayList<T> list

Método MaxHeap(lista: ArrayList<T>)

list <- lista

heap <- lista

Llamar a buildMaxHeap()

Método parent(indice: Entero) -> Entero

Retornar Entero(floor((indice + 1) / 2) - 1)

Método left(indice: Entero) -> Entero

Retornar 2 \* (indice + 1) - 1

Método right(indice: Entero) -> Entero

Retornar 2 \* (indice + 1)

Método swap(i: Entero, j: Entero)

T temp <- heap[i]

heap[i] <- heap[j]

heap[j] <- temp

Método insert(valor: T)

// Agregar el elemento al final

heap.Agregar(valor)

Entero indiceActual <- heap.Tamaño() - 1

// Mantener las propiedades del montículo

heapIncreaseKey(indiceActual)

Método extractMax() -> T

Si heap.Vacío() Entonces

Imprimir "El montículo está vacío"

Retornar Nulo

Fin Si

Si heap.Tamaño() == 1 Entonces

Retornar heap.Quitar(0)

Fin Si

// Extraer el elemento máximo del montículo

T valorMax <- heap[0]

heap[0] <- heap.Quitar(heap.Tamaño() - 1)

// Mantener las propiedades del montículo

maxHeapify(0)

Retornar valorMax

Método maxHeapify(indiceActual: Entero)

Entero indiceIzquierdo <- left(indiceActual)

Entero indiceDerecho <- right(indiceActual)

Entero indiceMayor <- indiceActual

Si indiceIzquierdo < heap.Tamaño() Y heap[indiceIzquierdo].CompararCon(heap[indiceMayor]) > 0 Entonces

indiceMayor <- indiceIzquierdo

Fin Si

Si indiceDerecho < heap.Tamaño() Y heap[indiceDerecho].CompararCon(heap[indiceMayor]) > 0 Entonces

indiceMayor <- indiceDerecho

Fin Si

Si indiceActual != indiceMayor Entonces

swap(indiceActual, indiceMayor)

indiceActual <- indiceMayor

// Descender en el montículo

maxHeapify(indiceActual)

Fin Si

Método buildMaxHeap()

Para i de (list.Tamaño() / 2) hasta 0 con paso -1

maxHeapify(i)

Fin Para

Método heapMaximum() -> T

Retornar heap[0]

Método heapIncreaseKey(indice: Entero)

Mientras indice > 0 Y heap[parent(indice)].CompararCon(heap[indice]) < 0

swap(indice, parent(indice))

indice <- parent(indice)

Fin Mientras

Método print() -> Cadena

Cadena mensaje <- ""

Si heap == Nulo Entonces

mensaje <- "No hay tareas prioritarias almacenadas"

Sino

Para i de 0 hasta heap.Tamaño() - 1

mensaje <- mensaje + heap[i]

Fin Para

Fin Si

Retornar mensaje

**Paso 6. Preparación de Informes y Especificaciones**

\*\*Análisis de Complejidad Temporal para las Operaciones:\*\*

1. \*\*Operación `isEmpty`\*\*:

- La operación `isEmpty` determina si el MaxHeap 'heap' está vacío.

- Tiempo Costo: O(1)

- Precondición (pre): Se asume que el MaxHeap 'heap' está definido y existe como entrada.

2. \*\*Método `insert` (Push)\*\*:

- Antes de insertar un elemento en el MaxHeap, se verifica si el MaxHeap está vacío usando la operación `isEmpty`.

- Tiempo Costo: O(1)

- Precondición: Se asume que el MaxHeap 'heap' está definido y existe como entrada.

- Postcondición:

- `isEmpty(heap) = true` si 'heap' no contiene elementos, es decir, si 'heap' está vacío (size(heap) = 0).

- `isEmpty(heap) = false` si 'heap' contiene uno o más elementos, es decir, si 'heap' no está vacío (size(heap) > 0).

3. \*\*Método `extractMax` (Pop)\*\*:

- Antes de extraer el elemento máximo del MaxHeap, se verifica si el MaxHeap está vacío usando la operación `isEmpty`.

- Tiempo Costo: O(1)

- Precondición: Se asume que el MaxHeap 'heap' está definido y existe como entrada.

- Postcondición:

- `isEmpty(heap) = true` si 'heap' no contiene elementos, es decir, si 'heap' está vacío (size(heap) = 0).

- `isEmpty(heap) = false` si 'heap' contiene uno o más elementos, es decir, si 'heap' no está vacío (size(heap) > 0).

4. \*\*Método `size`\*\*:

- El método `size` devuelve el número de elementos en el MaxHeap.

- Tiempo Costo: O(1)

- No verifica si el MaxHeap está vacío o no.

5. \*\*Método `heapMaximum` (Top)\*\*:

- El método `heapMaximum` devuelve el elemento máximo en el MaxHeap.

- Tiempo Costo: O(1)

- No verifica si el MaxHeap está vacío o no.

6. \*\*Método `peek`\*\*:

- El método `peek` devuelve el elemento en la parte superior del MaxHeap sin eliminarlo.

- Tiempo Costo: O(1)

- No verifica si el MaxHeap está vacío o no.

\*\*Resumen:\*\*

En resumen, las operaciones `isEmpty`, `insert`, `extractMax`, `size`, `heapMaximum` y `peek` tienen un tiempo constante de O(1), lo que significa que el tiempo necesario para ejecutar estas operaciones no depende del tamaño del MaxHeap. La complejidad temporal total del algoritmo depende de la cantidad de colisiones que ocurren, lo que puede llevar a un peor caso con complejidad O(n^2).

\*\*Análisis de Complejidad Temporal para las Operaciones:\*\*

1. \*\*Operación `isEmpty`\*\*:

- La operación `isEmpty` determina si el MaxHeap 'heap' está vacío.

- Complejidad Temporal: O(1)

- Precondición: Se asume que el MaxHeap 'heap' está definido y existe como entrada.

2. \*\*Método `insert` (Push)\*\*:

- Antes de insertar un elemento en el MaxHeap, se verifica si el MaxHeap está vacío usando la operación `isEmpty`.

- Complejidad Temporal: O(1)

- Precondición: Se asume que el MaxHeap 'heap' está definido y existe como entrada.

- Postcondición:

- `isEmpty(heap) = true` si 'heap' no contiene elementos, es decir, si 'heap' está vacío (size(heap) = 0).

- `isEmpty(heap) = false` si 'heap' contiene uno o más elementos, es decir, si 'heap' no está vacío (size(heap) > 0).

3. \*\*Método `extractMax` (Pop)\*\*:

- Antes de extraer el elemento máximo del MaxHeap, se verifica si el MaxHeap está vacío usando la operación `isEmpty`.

- Complejidad Temporal: O(1)

- Precondición: Se asume que el MaxHeap 'heap' está definido y existe como entrada.

- Postcondición:

- `isEmpty(heap) = true` si 'heap' no contiene elementos, es decir, si 'heap' está vacío (size(heap) = 0).

- `isEmpty(heap) = false` si 'heap' contiene uno o más elementos, es decir, si 'heap' no está vacío (size(heap) > 0).

4. \*\*Método `size`\*\*:

- El método `size` devuelve el número de elementos en el MaxHeap.

- Complejidad Temporal: O(1)

- No verifica si el MaxHeap está vacío o no.

5. \*\*Método `heapMaximum` (Top)\*\*:

- El método `heapMaximum` devuelve el elemento máximo en el MaxHeap.

- Complejidad Temporal: O(1)

- No verifica si el MaxHeap está vacío o no.

6. \*\*Método `peek`\*\*:

- El método `peek` devuelve el elemento en la parte superior del MaxHeap sin eliminarlo.

- Complejidad Temporal: O(1)

- No verifica si el MaxHeap está vacío o no.

\*\*Resumen:\*\*

En resumen, las operaciones `isEmpty`, `insert`, `extractMax`, `size`, `heapMaximum` y `peek` tienen un tiempo constante de O(1), lo que significa que el tiempo necesario para ejecutar estas operaciones no depende del tamaño del MaxHeap. La complejidad temporal total del algoritmo depende de la cantidad de colisiones que ocurren, lo que puede llevar a un peor caso con complejidad O(n^2).

\*\*Complejidad Temporal del Algoritmo:\*\*

La complejidad temporal del algoritmo depende de la cantidad de colisiones que ocurren en el MaxHeap. En el peor caso, la complejidad puede alcanzar O(n^2), debido a la creación de nuevas tablas temporales en caso de colisiones. En el mejor caso, las operaciones tienen una complejidad constante de O(1), independientemente del tamaño del MaxHeap. Por lo tanto, la complejidad temporal del algoritmo puede variar entre O(1) y O(n^2) dependiendo de las colisiones.

\*\*Complejidad Espacial del Algoritmo:\*\*

El análisis de la complejidad espacial muestra que el algoritmo utiliza una cantidad constante de espacio en memoria, O(1), para variables locales y operaciones no recursivas. En el caso de llamadas recursivas, se utiliza espacio adicional en la pila, lo que contribuye a una complejidad espacial de O(n) en el peor caso debido a las llamadas recursivas. El término dominante es O(n), lo que significa que el espacio en memoria crece linealmente con respecto al tamaño del MaxHeap.

Espero que esta versión reformulada en formato de texto sea útil. Si tienes alguna otra solicitud o necesitas más información, no dudes en preguntar.

**Test Case Desing**

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Clase** | **Escenario** |
|  |  |  |

**Diseño de Casos de Prueba**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la Prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de Entrada** | **Resultado** |
|  |  |  |  |  |