**Memoria proyecto EPED 2025**

1. **Preguntas teóricas 2.1**

Precondiciones y postcondiciones de algunas operaciones:

Add (text, date)

* Precondición: text no es NULL ni vacío. date sigue el formato YYYYMMDD y no coincide con ninguna otra fecha planificada.
* Postcondición: Se añade una nueva tarea con text y date a las tareas futuras sin alterar otras tareas ni el histórico.

Delete(date)

* Precondición: Existe una tarea futura con fecha date.
* Postcondición: La tarea con fecha date se elimina de las tareas futuras sin cambiar otras tareas ni el histórico.

Move (origDate, newDate)

* Precondición: Existe una tarea futura con fecha origDate y no existe ninguna con fecha newDate.
* Postcondición: La tarea con fecha origDate se cambia a newDate y las otras tareas se reajustan si es necesario, sin modificar el histórico.

Execute ()

* Precondición: Hay al menos una tarea en las tareas futuras.
* Postcondición: Se saca la tarea con la fecha más próxima y se añade al histórico como completada. Otras tareas se reordenan si es necesario.

Discard ()

* Precondición: Hay al menos una tarea en las tareas futuras.
* Postcondición: Igual que Execute, pero la tarea se marca como no completada al añadirse al histórico.

Estructura histórica de tareas: La mejor forma de almacenar tareas completadas o descartadas es mediante una lista enlazada o un array dinámico ordenado ascendentemente. Solo se requiere insertar la tarea procesada al final o en la posición correspondiente, sin necesidad de búsquedas ni eliminaciones.

1. **Preguntas teóricas 2.3**

Tipo de secuencia

* Es adecuado utilizar listas enlazadas o un array dinámico siempre y cuando se permita insertar elementos en cualquier posición y poder recorrerlos en orden. Esto es porque necesitamos mantener siempre las tareas ordenadas por su fecha. Usando otras estructuras como una cola o una pila no podríamos extraer siempre la tarea de fecha más próxima sin recorrer toda la secuencia.

Orden de almacenamiento

* Lo mejor es ordenar de manera ascendente por fecha, porque así ayudamos a que las operaciones Execute y Discard accedan a la primera tarea en orden.

Impacto en eficiencia

* Add: O(n) para encontrar posiciones, desplazar elementos o correr hasta posición.
* Delete, Move: O(n) localizar tarea por fecha más O(n) para reajustar la secuencia.
* Execute, Discard: O (1) para extraer la cabeza.
* Recorrido (iteratorFuture): O(n) secuencial.
* Sin orden, execute / Discard: Serian O(n) cada vez con orden O (1)

1. **Preguntas teóricas 2.4**
   1. Árbol BST
      * Permite la inserción, búsqueda y eliminación con O(h) donde h es la altura del árbol.
      * Si está bien equilibrado h = log n por lo que puede hacerse en O (log n).
      * Provee iterador en orden creciente de fecha.
   2. Ventajas frente a frecuencia.
      * Operaciones más eficientes, extracción de la fecha mínima y sin costes lineales de desplazamiento o modificación.
2. **Preguntas teóricas sobre el estudio del coste.**
   1. El tamaño del problema viene determinado por el número total de áreas futuras almacenadas en el planificador (n).

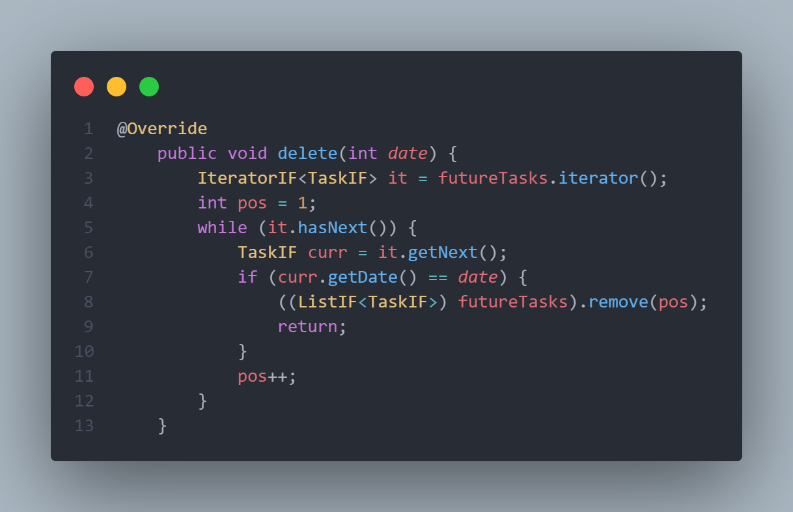
**Caso de SEQUENCE**

* Add (text, date): Recorre la secuencia buscando la posición correcta según la fecha O (n) e inserta nueva tarea O (n) si hay que desplazar elementos, por lo que en el peor de los casos el coste es O (n).

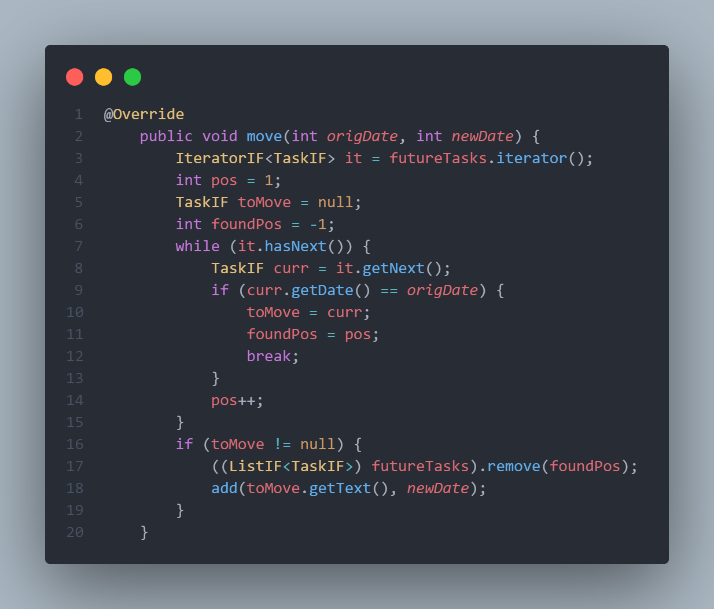
Texto

El contenido generado por IA puede ser incorrecto.

* Delete (date): Cuando busca la tarea por fecha O (n) eliminar una tarea y reordena si es necesario O (n) así que igual que el apartado anterior en el peor de los casos es O (n).



* Move (origDate, newDate): Es igual que un Delete + Add por lo que el coste es O (n) en notación asintótica.



**Caso de TREE**

* Add (text, date): Al insertar una tarea usando el método Add del árbol O (log n) si está el árbol equilibrado eso sí. así que en el peor de los casos será O (log n) o en su defecto O (n) si el árbol no se encuentra en equilibrio.

Texto

El contenido generado por IA puede ser incorrecto.

* Delete (date): Busca la tarea recorriendo el árbol seria O (n), llamar a remove del BSTree que puede variar entre O (log n) o en el peor de los casos O (n) y con esto el coste en el peor de los casos quedaría O (n).



* Move (origDate, newDate): Combinando los dos apartados anteriores nos queda un coste en el peor caso de O (n) pero si se encuentra el árbol equilibrado seria O (log n).



* 1. Los costes empíricos observados coinciden con el análisis teórico. Se observa en las pruebas y los datos recogidos que par números bajos de tareas ambas implementaciones son bastante eficientes en cuanto a tiempos. Es a partir de las 100000 tareas cuando se empieza a ver que SEQUENCE tiene una eficiencia más estable. En pruebas realizadas con inserciones, eliminaciones y movimientos SEQUENCE

1. **Juegos de pruebas.**

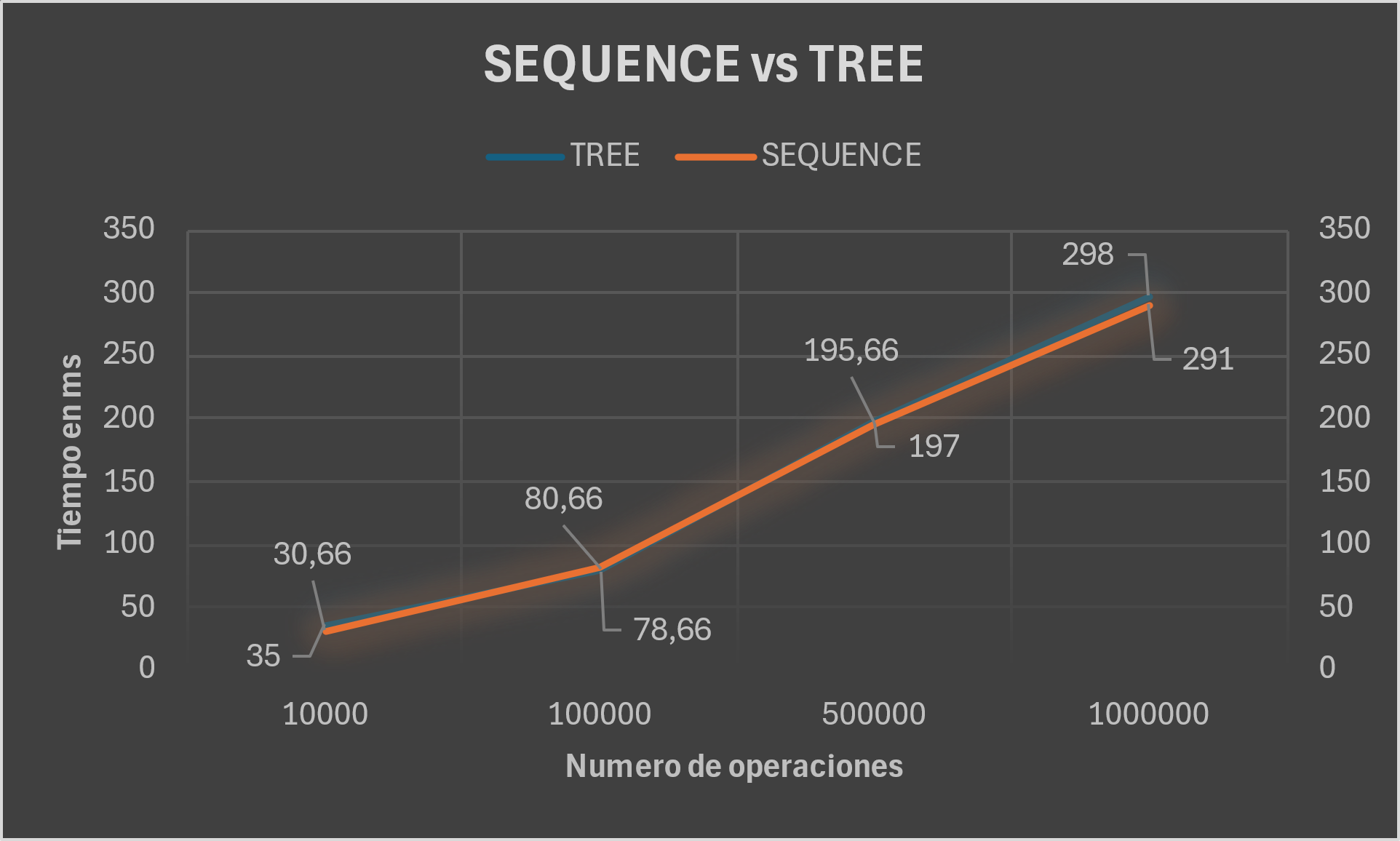
Para evaluar el comportamiento temporal de **SEQUENCE** y **TREE**, se realizaron pruebas con ficheros diseñados especialmente. Estos ficheros contienen una mezcla equilibrada de operaciones Add, Delete, Move, execute y Discard, simulando un uso general del planificador. Aquí están los tiempos de ejecución de cada archivo, medidos en milisegundos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Casos de estudio** | | **Tiempo (ms)** | | **SEQUENCE vs TREE** |
| **SEQUENCE** | **TREE** |
| 1 | Mix 10k | 30.66 | 35 | 14.15 % |
| 2 | Mix 100k | 80.66 | 78.66 | 2.54 % |
| 3 | Mix 500k | 195.66 | 197 | 1.01 % |
| 4 | Mix 1M | 291 | 298 | 2.34 % |

Ambos métodos muestran un rendimiento similar en todos los casos, especialmente con cargas pequeñas y medianas. Aunque teóricamente TREE debería escalar mejor debido a su coste logarítmico promedio, en la práctica, SEQUENCE ha sido ligeramente más eficiente debido a la sobrecarga estructural de TREE y la eficiencia interna de las listas dinámicas usadas en SEQUENCE.

Este resultado indica que, aunque TREE tiene ventajas en búsquedas o inserciones intensivas y desordenadas, en un uso mixto con datos moderadamente ordenados, SEQUENCE es muy competitivo e incluso más rápido. Por lo tanto, para una planificación con muchas operaciones variadas, ambas implementaciones son eficientes, pero SEQUENCE muestra un rendimiento empírico superior en este caso.

Visto en forma de grafico nos quedaría tal que así:



Mis juegos de pruebas los he realizado con un pequeño código de Java que genera 4 archivos .txt con operaciones variadas con un rango de números entre 10K y 1M para ver la diferencia de tiempos en cada rango. Mi código para generarlos es este:

Texto

El contenido generado por IA puede ser incorrecto.

1. Conclusión

En esta práctica, diseñé un sistema de planificación de tareas que permite añadir, eliminar, mover y ejecutar tareas utilizando dos estructuras: una lista ordenada (SEQUENCE) y un árbol de búsqueda (TREE). Para ello, desarrollé una interfaz común que conecta ambas implementaciones y aproveché estructuras de datos proporcionadas por la UNED. Además, verifiqué los resultados a través de pruebas oficiales para garantizar su precisión.

Opté por la estructura SEQUENCE por su eficiencia en operaciones puntuales, al tener un coste computacional de O (1). Sin embargo, implementé TREE debido a su escalabilidad superior cuando se aumenta el volumen de tareas. Esta práctica fue un desafío enriquecedor que me permitió afianzar conceptos esenciales, como el análisis de coste computacional, el diseño modular, el uso de estructuras abstractas y la evaluación de eficiencia en el desarrollo de software. Me siento satisfecho con el resultado, ya que refleja tanto mi compromiso como mi aprendizaje continuo en la materia.