I Tarea Programada:

Implementación de estructuras de datos en el juego “Combate Naval”

Estudiantes: Melissa Garita Chacón C23186, Paula Sofía Grell Araya C22977, Melanny Hernández Rivera B83808, Emanuel Ramírez Rojas C06378.

Resumen. En el presente trabajo, se implementarán diversas estructuras de datos, previamente estudiadas, para comprender sus funcionalidades y aplicaciones en situaciones del mundo real. Estas estructuras incluyen listas enlazadas, árboles binarios y algoritmos de búsqueda, como la búsqueda binaria.

Se simulará una “batalla naval” para implementar las estructuras representadas como barcos, lo cual permitirá realizar operaciones básicas de búsquedas, inserciones y eliminaciones. Además, se medirán las iteraciones y tiempos de ejecución, comparando así los resultados obtenidos en la práctica con la teoría.

Palabras clave: *estructuras de datos, batalla naval, algoritmos, complejidad temporal.*

I. INTRODUCCIÓN

En el presente trabajo, se llevará a cabo la implementación de distintos algoritmos y estructuras de datos, con el objetivo de evidenciar en la práctica los conceptos teóricos. Se sabe que las estructuras de datos tienen aplicaciones muy diversas, dependiendo de la necesidad que se presente y del problema que se pretenda resolver. Hasta ahora, se reconoce que los árboles binarios son algunas de las estructuras más estables, ya que su aplicación en contextos como la gestión de datos o los sistemas de atención de pacientes, respaldan esta afirmación.

Una de las implementaciones de los árboles binarios también se refleja en el desarrollo de videojuegos, como es el caso de las batallas navales. En este tipo de juegos (y como se presentará en el presente proyecto), los árboles pueden ser utilizados para representar posiciones y estados de los barcos, facilitando operaciones de búsqueda de elementos, simulaciones de ataques y de toma de decisiones, optimizando así los métodos en cuanto a temas de eficiencia y tiempo.

De este modo, el presente proyecto busca no sólo reforzar los conocimientos teóricos que se han adquirido sobre algoritmos y estructuras de datos, sino también, la demostración de su aplicabilidad en entornos simulados sobre problemas reales. A través de una simulación de una “batalla naval”, se buscará el ofrecimiento de una perspectiva práctica y analítica sobre el comportamiento de las estructuras utilizadas, evaluando sus tiempos de ejecución y desempeño en casos prácticos.

II. METODOLOGÍA

Para llevar a cabo el desarrollo del trabajo, se implementará el código utilizando el lenguaje de programación C++ y sus respectivas librerías, haciendo especial uso de la librería chrono, la cual permitirá medir el tiempo de ejecución en milisegundos. El trabajo consistirá en la simulación de una batalla naval, desarrollado mediante la implementación de distintas clases que representarán los

elementos del juego. Estas clases incorporarán internamente diversos algoritmos asociados a estructuras de datos como listas enlazadas, sets, árboles binarios y la búsqueda binaria de elementos.

Además, se incluirán contadores de iteraciones en las funciones de búsqueda con el propósito de ilustrar con mayor claridad la cantidad de veces que se ejecutan estas acciones dentro de cada estructura. Estas métricas permitirán comparar el rendimiento entre diferentes estructuras y evaluar su eficiencia bajo distintos escenarios de la simulación.

III. RESULTADOS

A partir de la realización de los experimentos durante la simulación de la "batalla naval", se obtuvieron los siguientes resultados:

Algoritmo de Búsqueda Lineal sobre Lista Enlazada

A continuación, se muestran los resultados obtenidos para el algoritmo de Búsqueda Lineal sobre Lista Enlazada:

Daño	Iteraciones	Tiempo (ms)
1	903	0.003435
1	903	0.003832
1	903	0.003628
1	913	0.011159
1	913	0.003067
1	913	0.007532

Fuente: Elaboración propia

Algoritmo de Búsqueda Binaria

A continuación, se muestran los resultados obtenidos para el algoritmo de Búsqueda Binaria:

Daño	Iteraciones	Tiempo (ms)
1	506	0.002173
1	808	0.004037
1	674	0.002668
1	827	0.004607
1	827	0.002641
5	187	0.001284

Fuente: Elaboración propia

Algoritmo Set

A continuación, se muestran los resultados obtenidos para el algoritmo de Set:

Iteraciones	Tiempo (ms)
915	0.024120
915	0.024264
915	0.033534

Fuente: Elaboración propia

Algoritmo RedBlack Tree

A continuación, se muestran los resultados obtenidos para el algoritmo de RedBlack Tree:

Daño	Iteraciones	Tiempo (ms)
100	10	0.002094
111	9	0.001722

100	10	0.001608
111	9	0.001164
111	9	0.002391
100	10	0.001528

Fuente: Elaboración propia

Algoritmo B-Tree

A continuación, se muestran los resultados obtenidos para el algoritmo del B-Tree:

Daño	Iteraciones	Tiempo (ms)
333	3	0.001709
166	6	0.001378
166	6	0.001344
200	5	0.001153
111	9	0.003213
142	7	0.001850

Fuente: Elaboración propia

Algoritmo Splay-Tree

A continuación, se muestran los resultados obtenidos para el algoritmo del Splay Tree:

Daño	Iteraciones	Tiempo (ms)
1	811	0.00718
7	133	0.001857
3	299	0.002486
10	95	0.006688
76	13	0.000638

Fuente: Elaboración propia

IV. DISCUSIÓN

Después de obtener los tiempos de ejecución de los algoritmos, resulta necesario realizar una comparación con la teoría existente sobre el tema.

Barco Linked (Búsqueda Lineal sobre Lista Enlazada)

Las listas enlazadas, tanto simples como dobles, presentan un comportamiento muy similar. Para realizar una operación de búsqueda, es necesario recorrer nodo por nodo, comparando el contenido del nodo actual con el elemento buscado. Esto implica que, en el peor de los casos, la búsqueda tendrá una complejidad temporal de O(n), ya que se debe recorrer toda la lista desde el inicio -o desde el final, en el caso de listas dobles-, para determinar si el elemento se encuentra o no en la estructura. En el mejor de los casos, si el elemento se encuentra en la primera posición o en la última, dependiendo del punto de inicio, la búsqueda puede tener una complejidad de O(1).

En el caso del barco Linked, la lista enlazada que se implementó para los casos de estudio, es coherente debido a que sus iteraciones coinciden con sus nodos revisados. Su tiempo de búsqueda son bajos pero de forma de creciente con el valor encontrado y el daño en este caso no es relevante para el análisis de complejidad. Las iteraciones que se obtuvieron son altas, lo cual es un buen indicador que la teoría se cumple, más en casos donde la lista enlazada posee muchos elementos. Los tiempos de búsqueda son bajos pero crecientes. Cuando no se encuentra el dato, el tiempo oscila entre 0.003 y 0.0038 ms, y cuando sí se encuentra, el tiempo sube a 0.007 o

incluso 0.011 ms. Esto tiene sentido: al encontrar el valor se termina la búsqueda, pero el nodo puede estar al final de la lista (iteración 913), lo que genera un tiempo un poco más alto.

Barco Vanguardia (Búsqueda Binaria)

Los árboles de búsqueda binaria permiten realizar operaciones de inserción, búsqueda y eliminación con una complejidad promedio de O(log n), donde n representa la cantidad de nodos en el árbol. Esta eficiencia se logra gracias a la forma en que se divide la estructura en cada paso, reduciendo a la mitad el espacio de búsqueda.

Sin embargo, en el peor de los casos, si el árbol se desbalancea -por ejemplo, si todas las inserciones se realizan en un solo lado-, puede degenerar en una estructura lineal, lo que incrementa la complejidad a O(n). Las operaciones de búsqueda e inserción suelen mantener la complejidad de O(log n) en estructuras balanceadas, al igual que las eliminaciones, que también requieren un número logarítmico de pasos en promedio.

De acuerdo con los resultados obtenidos para este algoritmo, puede evidenciarse que el número de iteraciones es elevado, lo cual tiene sentido si se observa desde la teoría. Las iteraciones aumentan considerablemente cuando el árbol binario no está balanceado, ya que existe la posibilidad de que el nodo buscado se encuentre en una posición muy profunda dentro de la estructura o que el nodo no sea encontrado del todo. Los resultados muestran que para esta implementación, algunas búsquedas llegaron a requerir más de 800 iteraciones, lo cual es indicativo de un árbol desbalanceado. En teoría, un árbol binario de búsqueda tiene una complejidad de O(log n), pero esto solo se cumple cuando el árbol está balanceado. Si los elementos se insertan en orden o casi en orden, el árbol se degenera en una estructura lineal, lo cual lleva la búsqueda a comportarse como O(n).

Si se considera que esta implementación no cuenta con un mecanismo de balanceo, entonces es completamente esperable que los tiempos y las iteraciones se vean afectados negativamente. Tal es el caso de búsquedas con más de 800 iteraciones, lo cual no solo representa un esfuerzo computacional mayor, sino también una ineficiencia en términos de tiempo, a pesar de que el número de nodos no sea tan alto.

En conclusión, los resultados obtenidos para este caso sí son coherentes con la teoría, siempre que se tenga en cuenta que el árbol binario utilizado no cuenta con balanceo automático.

Barco Centinela (Set)

En los conjuntos conocidos como sets, la teoría indica que las operaciones de búsqueda, inserción y eliminación presentan una complejidad promedio de O(log n), dado que estas estructuras deben mantenerse ordenadas. Esto implica que el tiempo requerido para dichas operaciones es proporcional al logaritmo del número de elementos almacenados (n).

En el peor de los casos, la complejidad puede aumentar hasta O(n). No obstante, si se consideran los conjuntos no ordenados o unordered sets, estas operaciones pueden realizarse en tiempo constante, es decir, O(1), ya que no es necesario recorrer toda la estructura para acceder a un elemento en específico.

En el caso práctico de la implementación de los sets, puede observarse que, para este estudio, no se cumple la teoría antes mencionada. Las iteraciones comprenden valores muy elevados, y los tiempos de

ejecución son los mayores obtenidos durante el experimento (entre 0.024120 y 0.033534). Las 915 iteraciones sugieren que el comportamiento de este algoritmo se acerca más a una búsqueda lineal, pues podría partirse del hecho de estar ante un "arreglo" sin ordenar, o un árbol completamente desbalanceado.

Barco Resolución (RedBlack Tree)

Los árboles rojo-negros o Red Black Trees, son una variante de los árboles binarios de búsqueda, los cuales incluyen reglas y casos adicionales con el fin de mantener su equilibro de manera automática luego de la inserción o eliminación de un nodo. Gracias a dichas propiedades, se garantiza correctamente que el árbol mantenga una altura logarítmica con respecto al número de nodos, manteniendo así, eficiencia en sus operaciones principales (inserciones, búsqueda, eliminaciones).

Las operaciones de inserción, búsqueda y eliminaciones presentan una complejidad temporal de O(log n), tanto en el peor de los casos como en el caso promedio. Lo anterior, sucede debido a que, por la estructura del algoritmo, se asegura que no se desbalancee cuando se realizan modificaciones en los nodos.

En el caso práctico de la implementación de esta estructura dentro de la batalla naval, se puede observar que, después de las corridas realizadas, se obtuvieron resultados conformes con la teoría, ya que las iteraciones encaja perfectamente con lo esperado, sus tiempos de búsqueda son consistentes y va de acuerdo con su rango de una media baja (1.1 a 2.3 ms). Se encuentra una estabilidad con el algoritmo, como otros árboles anteriores podemos ver que tiene una métrica personalizada y el hecho de que sea tan constante sugiere que el árbol está manteniendo bien su estructura interna.

Barco Invicto (B-Tree)

Los B-Trees o árboles B, son estructuras de datos que se diseñaron para optimizar operaciones en sistemas como las bases de datos. Estos poseen la posibilidad de tener múltiples hijos por nodo, almacenando una clave específica en cada uno, lo que reduce de forma significativa la altura del árbol, y, en consecuencia, el número de accesos necesarios para llegar a un dato en específico.

En las operaciones de búsqueda, inserción y eliminación de nodos, se conoce que estas estructuras poseen una complejidad promedio igual que en el peor de los casos, siendo O(log n), gracias a que el árbol se mantiene balanceado automáticamente producto de las divisiones o fusiones de nodos de acuerdo con la necesidad del caso.

De acuerdo con los resultados obtenidos para este algoritmo, se puede evidenciar que el número de iteraciones tiene sentido ya que se observa desde la teoría, que si se cumple. El número de iteraciones es bajo lo cuál representa una cantidad de niveles (altura) del árbol es pequeña en relación con la cantidad de elementos que contiene, su tiempo de búsqueda es mínimo. Los tiempos de altura son muy estables, ya que se observa que varían entre 0.0011 y 0.0032 ms, lo cual es una diferencia pequeña. En conclusión, los resultados sugieren que esta estructura se mantiene equilibrada, como debe estarlo. El número de iteraciones es muy bajo, lo cual refleja correctamente la baja altura que caracteriza a estos algoritmos, y además el tiempo de ejecución de los resultados, refleja la rapidez del acceso jerárquico que caracteriza a los B-Trees.

Barco Tempestad (Splay Tree)

El Splay Tree es una estructura de datos que se ajusta de forma automática, en la que predomina el hecho de que cada vez que se accede a un nodo específico -ya sea por búsquedas, inserciones o eliminaciones-, el nodo se mueve directamente a la raíz por medio de rotaciones llamadas splaying. Esto reduce el tiempo de acceso a los elementos que se requieren, y así se mejora el acceso a elementos que se utilizan frecuentemente pues los acerca a la raíz.

En el Splay Tree, las operaciones de inserción, búsqueda y eliminación es de O(log n), no obstante, aunque el peor de los casos puede costar O(n), tienden a ser muy eficientes en casos donde ciertos elementos se suelen consultar reiteradamente.

De acuerdo con los resultados obtenidos para este algoritmo, puede evidenciarse que el número de iteraciones tiene sentido si se observa desde la teoría. Las iteraciones pueden variar dependiendo de la estructura del árbol, porque existe la posibilidad de que el nodo estuviese muy profundo. Los resultados muestran que para la implementación de este algoritmo, las iteraciones se mantienen en tiempos de ejecución bajos, siendo que algunas iteraciones duraron más de lo esperado. Tal es el caso de las 299 iteraciones, pues siendo una cantidad baja en comparación con las 811 primeras, tomó más tiempo en ejecutarse. Si se analiza desde la teoría anteriormente expuesta, en estas estructuras la complejidad de O(log n) no significa que cada búsqueda vaya a tomar exactamente la cantidad de log n pasos. Algunos accesos pueden costar O(n) si el árbol se encuentra desbalanceado, pero una vez que ya se accede al nodo, se reajusta y los accesos posteriores serán más rápidos en comparación los primeros. Podría asumirse entonces que en este caso, las iteraciones altas como la de 811 son posibles si el nodo está al fondo del árbol y requiere muchas rotaciones y las iteraciones bajas como la de 13 son las esperadas cuando el árbol se ajusta. Asimismo, es importante recordar que el tiempo no siempre escala de forma directa, pues el costo real también dependerá de los recursos del sistema. En conclusión, los resultados obtenidos para este caso sí son coherentes con la teoría.

V. CONCLUSIONES

A partir de los resultados obtenidos en el presente trabajo, se pueden concluir los siguientes aspectos:

Las estructuras de datos balanceadas, como el B-Tree y Red-Black Tree, ofrecieron el mejor rendimiento en términos de búsqueda, con bajos tiempos de ejecución y pocas iteraciones. Esto concuerda con su complejidad teórica de O(log n), demostrando ser altamente eficientes y adecuadas para aplicaciones que requieren búsquedas frecuentes y rápidas.

La estructura Splay Tree, aunque mostró una alta variabilidad en el número de iteraciones, se comportó de acuerdo con su complejidad de O(log n). Esta variabilidad es esperada debido a su naturaleza autoajustable, donde las búsquedas recientes se optimizan a costa de otras menos frecuentes. En contextos donde ciertos elementos se consultan repetidamente, el Splay puede ser muy eficaz.

Por otro lado, estructuras como el Set y Lista Enlazada evidenciaron una eficiencia limitada, mostrando un comportamiento claramente lineal de O(n), con iteraciones elevadas y tiempos mayores. Esto indica que no son apropiadas para búsquedas en conjuntos de datos grandes, al menos sin mejoras o adaptaciones en su implementación.

Finalmente, el Binary Tree presentó un rendimiento irregular, con muchas iteraciones y tiempos comparativamente altos. Esto sugiere que la estructura no estaba balanceada, lo cual provoca una degradación hacia un comportamiento cercano a O(n), a pesar de que teóricamente puede alcanzar O(log n) si se mantiene equilibrada.

En resumen, los resultados empíricos obtenidos validan las expectativas teóricas de cada estructura, y reafirman la importancia de utilizar estructuras balanceadas para mantener una buena eficiencia en operaciones de búsqueda. La selección adecuada de la estructura de datos, basada en el contexto y los requerimientos específicos, es crucial para lograr un rendimiento óptimo en sistemas que manejan grandes volúmenes de información.

VI. REFERENCIAS

[1] T.H. Cormen, C.E. Leiserson, R.L, Rivest, and C. Stein, *Introduction to Algorithms,*3rd ed. The MIT Press, 2009.

[2] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Pearson, 2014.

[3] GeeksforGeeks, “Red-Black Tree | Set 1 (Introduction),” *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction/

[4] GeeksforGeeks, “B-Tree | Set 1 (Introduction),” *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/b-tree-set-1-introduction-2/

[5] GeeksforGeeks, “Splay Tree | Set 1 (Search),” *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/splay-tree-set-1-search/