Estructuras de datos

Gael Alpízar Alfaro C20270

***Resumen*— El trabajo consiste en implementar varias estructuras de datos de ordenamiento en C++ y elaborar un análisis acerca de sus tiempos de duración. Entre estas estructuras se encuentran la lista enlazada, el árbol de búsqueda binaria, el árbol rojinegro y la tabla de dispersión. Se realizarán pruebas para recolectar la recolección de datos sobre el rendimiento de estas estructuras, ejecutándolas varias veces con diferentes cantidades de elementos y capturando el tiempo de ejecución en segundos. Se analizará la variación de los tiempos en tres ejecuciones y se generarán gráficos para comparar los tiempos promedio contra la cantidad de elementos. Además, se graficarán los tiempos promedio en un mismo par de ejes para facilitar la comparación entre sus respectivos resultados.**

Palabras clave—estructura de datos, árbol de búsqueda binaria, lista enlazada, árbol rojinegro, tabla de dispersión.

I. Introducción

Las estructuras de datos son esenciales tanto en la informática y software como la resolución de necesidades o problemas en nuestras vidas diarias, optimizando la organización y gestión de datos en aplicaciones modernas. En la actualidad, con la explosión de datos generados, su eficiencia y rendimiento son cruciales. Dicho lo anterior es crucial el análisis de estas estructuras, siendo algunas destacadas como lo son la lista enlazada, el árbol de búsqueda binaria, el árbol rojinegro y la tabla de dispersión, proporcionando así bases sólidas para la manipulación eficiente de datos, destacando su relevancia en la optimización de recursos y tiempo de procesamiento.

Entre estas estructuras de datos se encuentra el árbol de búsqueda binaria, el cual está organizado en un árbol binario, donde cada nodo contiene un elemento clave y datos, además de referencias a sus hijos izquierdo y derecho, y a su padre. Los nodos cumplen con la propiedad del árbol de búsqueda binaria, donde los nodos en el subárbol izquierdo de un nodo tienen claves menores que la clave del nodo, y los nodos en el subárbol derecho tienen claves mayores.

El árbol de búsqueda binaria admite varias operaciones dinámicas de conjuntos, como búsqueda, mínimo, máximo, predecesor, sucesor, inserción y eliminación.

Las operaciones básicas en el árbol de búsqueda binaria poseen un tiempo proporcional a la altura del árbol. Además, sea un árbol binario completo con n nodos, estas operaciones se ejecutan con un tiempo, siendo el peor caso, de $\Theta(\lg n)$.

La complejidad de las operaciones de inserción y búsqueda en un árbol de búsqueda binaria depende de la altura del árbol. Estos procedimientos comienzan en la raíz del árbol y siguen un camino descendente, comparando las claves de los nodos para determinar la posición adecuada. En el peor de los casos, ambos procedimientos tienen una complejidad de tiempo de O(h), donde h es la altura del árbol.

Este peor caso ocurre cuando el árbol está desbalanceado, como cuando los elementos se insertan en orden ascendente o descendente, formando una cadena lineal. En este escenario, la altura del árbol sería igual al número de nodos, lo que resultaría en una complejidad de O(n), donde n es el número de nodos en el árbol.

Por otro lado, cuando el árbol está balanceado, como en el caso de una inserción aleatoria, la altura del árbol tiende a ser logarítmica en relación con el número de nodos, lo que resulta en una complejidad de O(lg n) para la inserción y búsqueda. Esto se debe a que, en promedio, tanto la inserción como la búsqueda dividen el espacio de búsqueda a la mitad en cada paso descendente del árbol, lo que lleva a una altura logarítmica.

Otra estructura de datos fundamental es una lista enlazada, la cual es una estructura de datos lineal donde los elementos están dispuestos en un orden determinado por punteros en cada objeto. Cada elemento de una lista enlazada contiene un atributo de clave y un puntero al siguiente elemento en la secuencia. Estas listas proporcionan una representación simple y flexible para conjuntos dinámicos, admitiendo todas las operaciones básicas, como búsqueda e inserción.

Cuando se busca un elemento en una lista enlazada, se realiza una búsqueda lineal simple a lo largo de la lista. En el peor de los casos, esta operación tiene un tiempo de ejecución de $\Theta(n)$, ya que puede requerir recorrer toda la lista para encontrar el elemento deseado. Sin embargo, en el mejor de los casos, cuando el elemento buscado está al principio de la lista, la búsqueda tiene una complejidad de tiempo de O(1), lo que significa que es constante.

Para insertar un nuevo elemento en una lista enlazada, se realiza un procedimiento que inserta el elemento al frente de la lista. Este proceso tiene un tiempo de ejecución constante de O(1), independientemente del tamaño de la lista. En el caso de la inserción ordenada, en el peor de los casos, puede requerirse recorrer toda la lista para encontrar la posición adecuada para insertar el nuevo elemento, lo que resulta en una complejidad de tiempo de $\Theta(n)$. Sin embargo, en el mejor de los casos, cuando se inserta al principio de la lista, la complejidad sigue siendo O(1).

Entre otras estructuras de datos se encuentra el árbol rojinegro, una variante de los árboles binarios de búsqueda que asegura que el árbol se mantenga balanceado durante todas las operaciones. Un árbol rojinegro es un árbol binario de búsqueda con una propiedad adicional: cada nodo tiene un color (rojo o negro) que, junto con otras propiedades, garantiza que el árbol se mantenga aproximadamente balanceado. Las propiedades

que debe cumplir un árbol rojinegro incluyen que cada nodo sea rojo o negro, la raíz sea negra, todas las hojas (NIL) sean negras, un nodo rojo tenga ambos hijos negros, y todos los caminos simples desde un nodo hasta sus hojas descendientes contengan el mismo número de nodos negros. Estas propiedades aseguran que cualquier ruta desde la raíz hasta una hoja no sea más del doble de larga que cualquier otra, manteniendo el árbol aproximadamente balanceado y garantizando que su altura sea O(lg n), donde n es el número de nodos.

Las operaciones de búsqueda en un árbol rojinegro tienen una complejidad de tiempo de O(lg n), debido a que la altura del árbol se mantiene logarítmica. Esta eficiencia se mantiene tanto en el caso de inserciones aleatorias como en el caso de inserciones ordenadas. En inserciones aleatorias, los árboles rojinegros tienden a mantenerse balanceados de forma natural, lo que asegura que la búsqueda siga siendo eficiente. En cambio, en inserciones ordenadas, que en un árbol binario de búsqueda simple podrían llevar a una altura lineal y a una búsqueda ineficiente con complejidad O(n), los árboles rojinegros utilizan rotaciones y recoloreos para mantener el balanceo del árbol, asegurando que la búsqueda tenga siempre una complejidad de O(lg n). Así, los árboles rojinegros proporcionan una estructura robusta para operaciones de búsqueda eficientes, independientemente del orden de inserción de los elementos.

Las tablas de dispersión (hash tables) son una estructura de datos eficiente para implementar diccionarios, soportando operaciones básicas como insertar, buscar y borrar con un tiempo promedio constante de O(1). Este rendimiento se debe a la función hash que distribuye las claves uniformemente entre las posiciones del array, minimizando la longitud de las listas enlazadas (o cadenas) en cada posición. Bajo la suposición de hash uniforme simple, donde cada clave es igualmente probable de hash a cualquier posición del array, la carga (load factor) a = n/m se mantiene baja, asegurando que las operaciones de búsqueda requieren, en promedio, inspeccionar solo unos pocos elementos.

Cuando las claves se insertan aleatoriamente, la búsqueda sigue teniendo un tiempo promedio de O(1) gracias a la buena distribución proporcionada por la función hash. Incluso cuando las claves se insertan en orden, una función hash bien diseñada dispersará estas claves de manera pseudoaleatoria por el array, evitando largas cadenas en cualquier posición específica. Sin embargo, si la función hash no está bien diseñada y produce muchas colisiones para claves ordenadas, la longitud de las cadenas puede aumentar, incrementando el tiempo de búsqueda. A pesar de esto, en la práctica, con una función hash adecuada, la búsqueda en una tabla de dispersión sigue siendo altamente eficiente, independientemente del orden de inserción de las claves.

En sí, el objetivo de este reporte es analizar y comparar las características, ventajas y limitaciones de estas estructuras, centrándose específicamente en las operaciones de búsqueda e inserción. Al comprender las complejidades de estas operaciones en diferentes contextos, tanto ordenados como aleatorios, se espera proporcionar información útil para un análisis claro de estas estructuras de datos y saber como manejar las estructuras de datos más eficientemente para aprovecharlas en situaciones de nuestras vidas diarias.

II. Metodología

Para llevar a cabo este estudio, se implementaron las estructuras de datos de lista enlazada, árbol de búsqueda binaria, árbol rojinegro y la tabla de dispersión utilizando el lenguaje de programación C++. Estas implementaciones siguieron las directrices y algoritmos presentados en el libro "Introduction to Algorithms", Cormen et al. (2022). El código fuente se estructuró en archivos de cabecera (`llist.h`, `bstree.h`, chasht.h` y rbtree.h`), con las pruebas ejecutadas desde un archivo principal (`main.cpp`). Las pruebas se realizaron en una computadora con procesador Intel i7 de 13.ª generación.

La implementación del árbol de búsqueda binaria y el árbol rojinegro incluyó métodos esenciales como el constructor, destructor, inserción, eliminación, búsqueda recursiva e iterativa, búsqueda del máximo y mínimo, obtención del sucesor de un nodo y recorrido en orden, con la excepción que en el árbol rojinegro se implementaron métodos y atributos distintivos de este. Para la lista enlazada y la tabla de dispersión, se desarrollaron métodos clave como constructor, destructor, inserción, búsqueda y eliminación. Toda la implementación se documentó detalladamente para asegurar claridad y facilidad de comprensión.

En cada estructura de datos se insertaron 1,000,000 de nodos, cada uno con una clave seleccionada aleatoriamente en el rango [0, 2n[, donde n es el número total de nodos. Posteriormente, se llevaron a cabo 10,000 búsquedas de elementos aleatorios, registrando el tiempo total de las búsquedas, tanto si los elementos estaban presentes como si no lo estaban.

De igual manera, se realizaron pruebas de inserción ordenada insertando secuencialmente las claves de 0 a n-1 en ambas estructuras de datos. Nuevamente, se efectuaron 10,000 búsquedas de elementos, registrando el tiempo de ejecución.

Para evitar ineficiencias y posibles errores al insertar claves ordenadas en un árbol de búsqueda binaria, se implementó un método alternativo para crear un árbol balanceado a partir de un arreglo. Esto se hace, principalmente, ya que al insertar claves en un orden secuencial en un árbol de búsqueda binaria puede ser ineficiente ya que produce un árbol altamente desbalanceado, donde todos los nodos tienen solo un hijo.

En el caso de la tabla de dispersión, para su implementación se optó a utilizar una versión modificada de la lista enlazada para adaptarla y transformarla en una lista doblemente enlazada. De igual forma, en este mismo, para las mediciones se trabajó con un factor de carga $\alpha = 1$, donde m = n.

Cada conjunto de pruebas se repitió tres veces para garantizar la precisión y consistencia de los resultados. Los tiempos de ejecución se midieron utilizando la biblioteca `chrono` de C++ y se calcularon promedios para obtener datos precisos. Se generaron gráficos para visualizar los tiempos promedio de cada escenario de inserción y búsqueda, permitiendo una comparación clara entre las estructuras de datos estudiadas.

Cuadro I Tiempo de ejecución en segundos para estructuras de datos Árbol de Búsqueda Binaria, Lista Enlazada, Árbol Rojinegro y Tabla de Dispersión, en inserción y búsqueda.

Estructura	*Ejecución* 1	2	3	Prom.
Lista Enlazada				
Inserción Aleatoria	0.1472	0.0843	0.0832	0.1049
Inserción Ordenada	0.0331	0.0343	0.0320	0.0331
Búsqueda post Inserción Aleatoria	60.0212	81.1963	65.4571	68.8915
Búsqueda post Inserción Ordenada	84.3942	63.3713	78.8151	75.5269
Árbol de búsqueda binaria				
Inserción Aleatoria	1.6620	2.1852	1.5385	1.7952
Inserción Ordenada	0.1163	0.1597	0.1324	0.1361
Búsqueda post Inserción Aleatoria	0.0188	0.0219	0.0162	0.0190
Búsqueda post Inserción Ordenada	52.5012	53.6471	52.2766	52.8083
Árbol rojinegro				
Inserción Aleatoria	0.5625	0.6813	0.6476	0.6305
Inserción Ordenada	0.1037	0.1058	0.1032	0.1042
Búsqueda post Inserción Aleatoria	0.0013	0.0015	0.0022	0.0017
Búsqueda post Inserción Ordenada	0.0047	0.0032	0.0041	0.0040
Tabla de dispersión				
Inserción Aleatoria	0.1935	0.1594	0.1623	0.1717
Inserción Ordenada	0.0412	0.0397	0.0391	0.0400
Búsqueda post Inserción Aleatoria	0.0015	0.0013	0.0011	0.0013
Búsqueda post Inserción Ordenada	0.0013	0.0014	0.0016	0.0014

Este enfoque metodológico permitió evaluar de manera exhaustiva el comportamiento de estas estructuras de datos en términos de inserción y búsqueda, tanto en escenarios aleatorios como ordenados. Los resultados empíricos se compararon con las expectativas teóricas, proporcionando una visión clara de la eficiencia y ventajas de cada estructura de datos en diferentes contextos de uso.

III. Resultados

Los resultados obtenidos de las pruebas realizadas brindan una visión detallada sobre el rendimiento de las estructuras de datos de lista enlazada y árbol de búsqueda binaria en diferentes escenarios de inserción y búsqueda.

Primeramente, en la lista enlazada, el tiempo promedio en la inserción aleatoria fue de aproximadamente 0.1049 segundos. Desde una perspectiva teórica, la complejidad esperada para la inserción aleatoria en una lista enlazada es lineal, es decir, O(n), donde n es el número de elementos en la lista. Este tiempo lineal se debe al hecho de que, en promedio, se necesita recorrer la mitad de la lista para insertar un nuevo elemento aleatoriamente. Sin embargo, en términos prácticos, este tiempo fue bastante bajo, lo que sugiere una eficiencia aceptable en la inserción aleatoria en una lista enlazada.

Por otro lado, la búsqueda después de la inserción aleatoria mostró un tiempo promedio de alrededor de 68.8915 segundos. La complejidad teórica esperada para la búsqueda en una lista enlazada es lineal O(n), ya que, en el peor de los casos, puede ser necesario recorrer toda la lista para encontrar el elemento deseado. Este tiempo relativamente alto para la búsqueda puede atribuirse a la naturaleza secuencial de la lista enlazada, que requiere recorrerla desde el principio hasta el elemento buscado.

En contraste, la inserción ordenada en la lista enlazada mostró tiempos promedios más bajos, alrededor de 0.0331 segundos. Sin embargo, la búsqueda después de la inserción ordenada resultó en tiempos promedio significativamente más altos, alrededor de 75.5269 segundos. La complejidad teórica esperada para la búsqueda en una lista enlazada ordenada es también lineal O(n), lo que implica recorrer secuencialmente la lista desde el principio hasta el elemento deseado. Esta complejidad se refleja en los tiempos de búsqueda observados.

Al tomar en cuenta estas complejidades, se puede apreciar cómo afectan los tiempos de ejecución de las operaciones en la lista enlazada, tanto después de inserciones aleatorias como ordenadas. Ahora, al introducir la Figura 1, que muestra el "Gráfico de búsqueda tras inserción ordenada e inserción aleatoria en una lista enlazada", se proporcionará una representación visual de estos tiempos y cómo difieren entre las dos estrategias de inserción. de las operaciones de inserción aleatoria y ordenada.

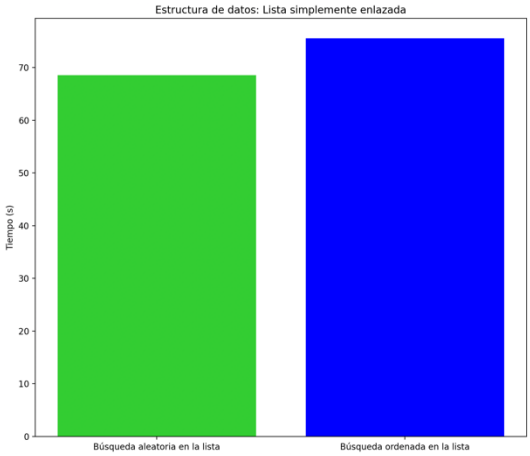

Figure 1. Gráfica de búsqueda por inserción aleatoria y ordenada de la lista enlazada

La Figura 1 proporciona una representación visual de los tiempos de búsqueda tras la inserción ordenada e inserción aleatoria en una lista enlazada. Se observa claramente que los tiempos de búsqueda después de la inserción aleatoria son generalmente más bajos en comparación con la inserción ordenada. Esto sugiere que, aunque la inserción aleatoria puede requerir más tiempo inicialmente debido a la naturaleza de las operaciones de enlace, resulta en una estructura de lista que facilita búsquedas más eficientes en comparación con la inserción ordenada, donde la lista está secuencialmente organizada.

Al analizar el comportamiento del árbol de búsqueda binaria, se observa una clara diferencia en los tiempos promedio de inserción y búsqueda según el orden de inserción de los elementos. En el caso de la inserción aleatoria, con un tiempo promedio de inserción de aproximadamente 1.7952 segundos, se evidencia una distribución equilibrada de los elementos en el árbol. Esto se traduce en tiempos de búsqueda más eficientes, con un promedio de aproximadamente 0.0190 segundos. La complejidad logarítmica O(lg n) en la búsqueda es consistente con la naturaleza balanceada del árbol generado por la inserción aleatoria.

Por otro lado, cuando los elementos se insertan en orden ascendente, el árbol tiende a degenerarse en una estructura lineal, lo que afecta negativamente la eficiencia de las operaciones. El tiempo promedio de inserción en este caso es de alrededor de 0.1361 segundos, mientras que el tiempo promedio de búsqueda aumenta significativamente a aproximadamente 52.8083 segundos. Esta diferencia en los tiempos refleja una complejidad lineal O(n) en la búsqueda, ya que se requiere recorrer secuencialmente la estructura del árbol para encontrar el elemento deseado.

Es importante destacar que, para contrarrestar los efectos adversos de la inserción ordenada, se implementó un método alternativo para construir un árbol balanceado a partir de un arreglo. Aunque este enfoque puede resultar en un tiempo de inserción ligeramente mayor en comparación con la inserción aleatoria, con un promedio de aproximadamente 0.1361 segundos, garantiza una búsqueda eficiente con una complejidad logarítmica O(lg n) en promedio, similar a la inserción aleatoria.

Al observar los tiempos promedio registrados para las operaciones de inserción y búsqueda en el árbol de búsqueda binaria, se confirma la relación entre el orden de inserción y la eficiencia de las operaciones. Los resultados obtenidos respaldan las expectativas teóricas en términos de complejidad temporal y destacan la importancia de considerar el patrón de inserción al diseñar y utilizar estructuras de datos como los árboles de búsqueda binaria.

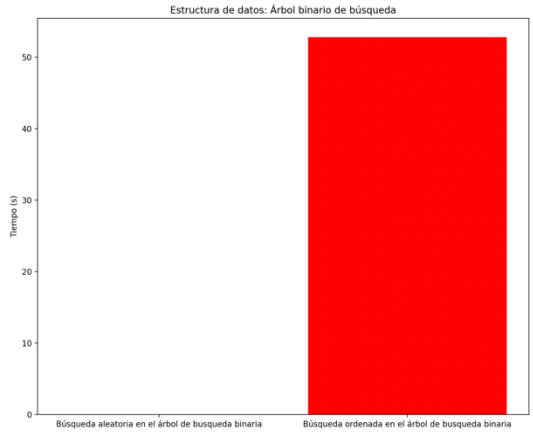

Figure 2. Gráfica de búsqueda por inserción aleatoria y ordenada del árbol de búsqueda binaria

En la Figura 2 se puede observar claramente la diferencia en los tiempos de búsqueda tras la inserción aleatoria y la inserción ordenada en el árbol de búsqueda binaria. Mientras que la búsqueda tras la inserción aleatoria muestra tiempos significativamente mucho más bajos, indicativos de una estructura balanceada, la búsqueda tras la inserción ordenada exhibe tiempos considerablemente más altos debido a la degeneración del árbol en una estructura lineal. Esta disparidad subraya la importancia del balance del árbol en la eficiencia de las operaciones de búsqueda.

El árbol rojinegro, conocido por su eficiencia en mantener el balance durante las inserciones y eliminaciones, mostró un rendimiento consistente en las pruebas. Para la inserción aleatoria, el tiempo promedio fue de aproximadamente 0.6305 segundos. La naturaleza balanceada del árbol rojinegro asegura que las operaciones de inserción y búsqueda se realicen en tiempo logarítmico O(logn), independientemente del orden de inserción.

La búsqueda después de la inserción aleatoria en el árbol rojinegro presentó un tiempo promedio de alrededor de 0.0017 segundos. Este rendimiento eficiente se debe a la propiedad del árbol rojinegro de mantener su altura logarítmica, lo que facilita búsquedas rápidas. De manera similar, la inserción ordenada en el árbol rojinegro demostró ser eficiente, con un tiempo promedio de 0.1042 segundos. La búsqueda post inserción ordenada tuvo un tiempo promedio de aproximadamente 0.0040 segundos, lo que confirma la eficiencia del árbol rojinegro en mantener un rendimiento óptimo incluso en casos de inserciones ordenadas.

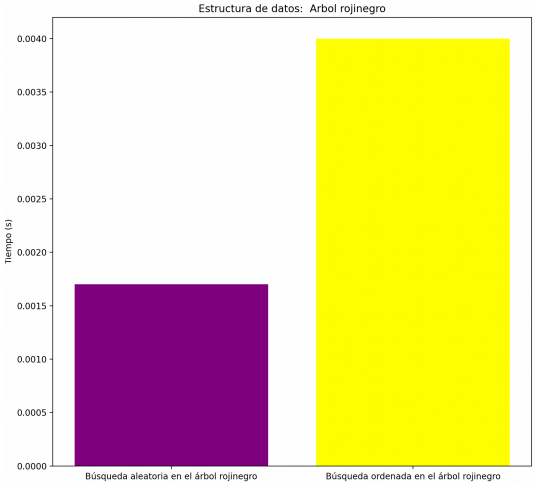

Figure 3. Gráfica de búsqueda por inserción aleatoria y ordenada del árbol rojinegro

La Figura 3 ilustra los tiempos de búsqueda tras la inserción aleatoria y ordenada en el árbol rojinegro. Los resultados muestran claramente que tanto la inserción como la búsqueda se benefician de la estructura balanceada del árbol rojinegro, manteniendo tiempos de operación consistentemente bajos.

Las tablas de dispersión (hash tables) se destacaron por su excepcional rendimiento en las operaciones de inserción y búsqueda. La inserción aleatoria en una tabla de dispersión tuvo un tiempo promedio de 0.1717 segundos. Debido a la eficiencia de las funciones hash bien diseñadas, la complejidad promedio de la inserción y búsqueda en una tabla de dispersión es constante O(1).

La búsqueda después de la inserción aleatoria en la tabla de dispersión presentó un tiempo promedio de aproximadamente 0.0013 segundos. Este rendimiento sobresaliente se debe a la capacidad de las tablas de dispersión para distribuir uniformemente los elementos en sus ranuras, minimizando las colisiones y manteniendo el tiempo de búsqueda constante. De manera similar, la inserción ordenada en una tabla de dispersión fue muy eficiente, con un tiempo promedio de 0.0400 segundos,

y la búsqueda post inserción ordenada tuvo un tiempo promedio de 0.0014 segundos.

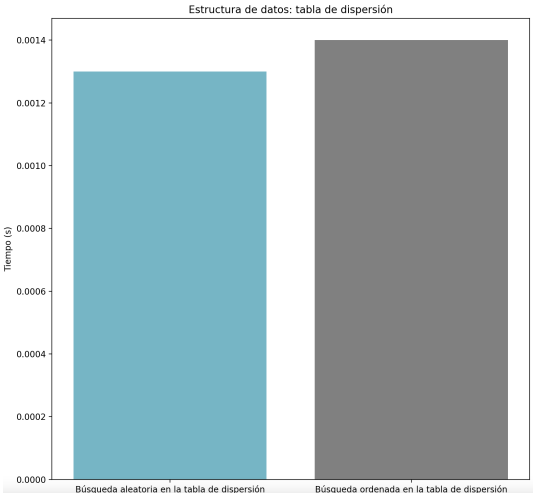

Figure 4. Gráfica de búsqueda por inserción aleatoria y ordenada de la tabla de dispersión

La Figura 4 proporciona una representación visual de los tiempos de búsqueda tras la inserción aleatoria y ordenada en una tabla de dispersión. Los resultados destacan la eficiencia de las tablas de dispersión en todas las operaciones, con tiempos de búsqueda prácticamente constantes, independientemente del orden de inserción de los elementos.

Ahora a continuación se hará una comparación de búsqueda tras inserción aleatoria, luego con inserción ordenada: Lista Enlazada vs. Árbol de Búsqueda Binaria vs. Árbol Rojinegro vs. Tabla de Disperción.

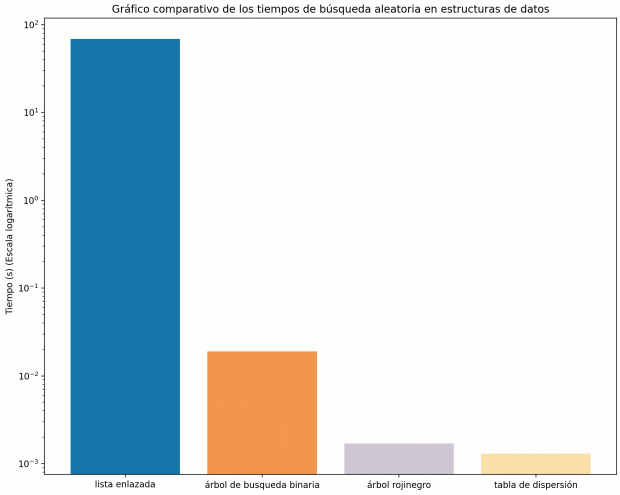

Figure 5. Gráfica comparativa de búsqueda por inserción aleatoria de las estructuras de datos, con escala logarítmica.

Según la figura 5, se observa una clara disparidad entre las estructuras de datos analizadas. La lista enlazada muestra un rendimiento notablemente inferior. Este resultado se refleja como un pico pronunciado en la gráfica, indicando que cada búsqueda requiere un recorrido secuencial completo de los nodos, lo cual es inherentemente menos eficiente a medida que aumenta el tamaño de los datos. En contraste, tanto el árbol de búsqueda binaria como el árbol rojinegro y la tabla de dispersión exhiben tiempos de búsqueda significativamente más bajos y consistentes. Estas estructuras mantienen tiempos de acceso rápidos y predecibles, representados como líneas casi planas en la gráfica, lo que subraya su capacidad para manejar eficientemente operaciones de búsqueda en conjuntos de datos grandes y variados.

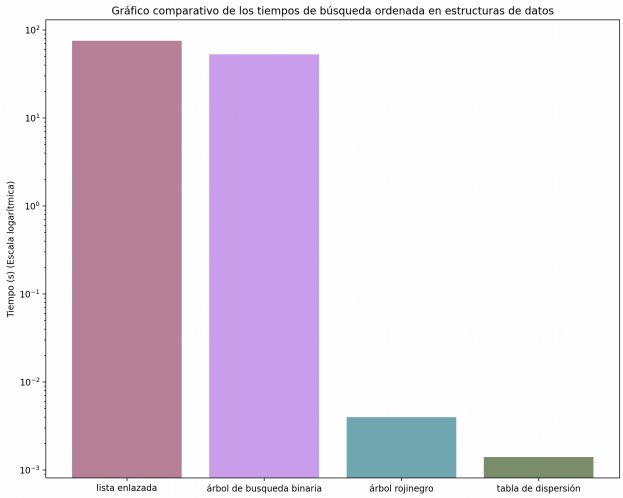

Figure 6. Gráfica comparativa de búsqueda por inserción ordenada de las estructuras de datos, con escala logarítmica.

En la Figura 6, se observa la comparación de los tiempos de búsqueda tras la inserción ordenada entre la lista enlazada, el árbol de búsqueda binaria, el árbol rojinegro y la tabla de dispersión. Se destaca una diferencia aún más marcada entre las estructuras de datos evaluadas. La lista enlazada muestra tiempos de búsqueda promedio de 75.5269 segundos, continuando con su desempeño menos eficiente debido a su naturaleza secuencial. Cada búsqueda implica recorrer todos los nodos en orden, resultando en un aumento lineal en los tiempos de búsqueda conforme crece el tamaño de los datos.

En contraste, el árbol de búsqueda binaria mantiene tiempos de búsqueda más bajos, con un promedio de 52.8083 segundos. Aunque puede experimentar un incremento respecto a la inserción aleatoria debido a la posible desbalanceación del árbol, sigue siendo considerablemente más eficiente que la lista enlazada gracias a su estructura balanceada que permite operaciones logarítmicas. El árbol rojinegro, con un tiempo promedio de búsqueda de 0.0040 segundos, también muestra una eficiencia destacable, mayor que la del Árbol de Búsqueda Binaria en este contexto particular. Por último, la la tabla de dispersión continúa demostrando su eficacia con tiempos de búsqueda promedio de 0.0014 segundos, los más bajos entre todas las estructuras evaluadas.

IV. Conclusiones

El estudio comparativo entre listas enlazadas, árboles de búsqueda binaria, árboles rojinegros y tablas de dispersión en diferentes escenarios de inserción y búsqueda ha mostrado diferencias claras en el comportamiento de estas estructuras de datos en la práctica observada. Las listas enlazadas demostraron ser eficientes en términos de inserción debido a su estructura simple y lineal, especialmente en el caso de inserciones aleatorias. Sin embargo, las búsquedas en listas enlazadas resultaron ser considerablemente más lentas, ya que es necesario recorrer la lista de manera secuencial para localizar un elemento.

Por otro lado, los árboles de búsqueda binaria exhibieron un comportamiento variable dependiendo del orden de inserción. La inserción aleatoria tendió a mantener el árbol equilibrado, resultando en tiempos de búsqueda significativamente más cortos en comparación con las listas enlazadas. Este balance permite que las operaciones de búsqueda se realicen de manera logarítmica, mostrando una clara ventaja de los árboles de búsqueda binaria para búsquedas eficientes.

El árbol rojinegro, aunque similar en concepto al árbol de búsqueda binaria, también mostró tiempos de búsqueda excepcionalmente bajos, destacando su capacidad para manejar eficientemente operaciones post inserción, tanto aleatoria como ordenada. Esto evidencia su robustez y eficacia en aplicaciones que requieren tiempos de respuesta rápidos y predecibles.

Por último, la tabla de dispersión continuó demostrando su eficiencia con tiempos de búsqueda mínimos y consistentes, independientemente del patrón de inserción. Su capacidad para realizar búsquedas directas mediante funciones de dispersión permite un acceso rápido a los datos almacenados.

Los resultados de este estudio observan cómo estas estructuras responden en la práctica, además de hacer una relación con las teorías sobre las complejidades de tiempo esperadas para listas enlazadas, árboles de búsqueda binaria, árboles rojinegros y tablas de dispersión. Mientras que las listas enlazadas pueden ser más adecuadas para operaciones de inserción rápida y simple, los árboles de búsqueda binaria y rojinegros balanceados, junto con las tablas de dispersión, son más efectivos para operaciones de búsqueda rápidas y eficientes. La elección entre estas estructuras debe basarse en las necesidades específicas del contexto de uso, considerando tanto el patrón de inserción como la frecuencia de las operaciones de búsqueda.

Referencias

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. Introduction to Algorithms, IV ed. MIT Press, 2022.

Gael Alpízar Alfaro
Facultad de ingeniería, Computación con Varios Énfasis
Universidad de Costa Rica (UCR).
Carné C20270.

Apéndice A

Pseudocódigo de las Estructuras de Datos

Lista enlazada Una lista enlazada es una estructura de datos lineal compuesta por nodos, donde cada nodo contiene un valor y un puntero que apunta al siguiente nodo en la secuencia. La lista comienza con un nodo cabeza (head), que es el punto de entrada a la lista. A diferencia de los arrays, las listas enlazadas no requieren un bloque contiguo de memoria, lo que permite una inserción y eliminación de elementos más flexible.

```
LIST-INSERT(x)
1. x.next = nil.next
2. nil.next = x


LIST-SEARCH(L, k)
1. x = L.head
2. while x ≠ NIL and x.key ≠ k
3. x = x.next
4. return x


Procedure Delete(x)
1. current = nil
2. while current.next ≠ nil and current.getNext() ≠ x
3. current = current.next
```

```
4. if current.next = x
5. current.next = x.next
```

Árbol de búsqueda binaria Un árbol de búsqueda binaria es una estructura de datos jerárquica compuesta por nodos, donde cada nodo contiene un valor, y tiene hasta dos hijos: un hijo izquierdo y un hijo derecho. En este, para cada nodo, todos los valores en el subárbol izquierdo son menores que el valor del nodo, y todos los valores en el subárbol derecho son mayores.

```
TREE-INSERT (T, z)
1    x = T.root
2    y = NIL
3    while x ≠ NIL
4        y = x
5        if z.key < x.key
6            x = x.left
7        else x = x.right
8    z.p = y
9    if y == NIL
10       T.root = z
11   elseif z.key < y.key
12       y.left = z
13   else y.right = z

TREE-MINIMUM (x)
1 while x.left ≠ NIL
2    x = x.left
3 return x

TREE-MAXIMUM (x)
1 while x.right ≠ NIL
2    x = x.right
3 return x

TRANSPLANT (T, u, v)
1    if u.p == NIL
2        T.root = v
3    elseif u == u.p.left
4        u.p.left = v
5    else u.p.right = v
6    if v ≠ NIL
7        v.p = u.p
```

```
TREE-DELETE (T, z)
1    if z.left == NIL
2        TRANSPLANT (T, z, z.right)
3    elseif z.right == NIL
4        TRANSPLANT (T, z, z.left)
5    else y = TREE-MINIMUM (z.right)
6        if y ≠ z.right
7            TRANSPLANT (T, y, y.right)
8            y.right = z.right
9            y.right.p = y
10       TRANSPLANT (T, z, y)
11       y.left = z.left
12       y.left.p = y

TREE-SEARCH (x, k)
1 if x == NIL or k == x.key
2    return x
3 if k < x.key
4    return TREE-SEARCH (x.left, k)
5 else return TREE-SEARCH (x.right, k)



ITERATIVE-TREE-SEARCH (x, k)
1 while x ≠ NIL and k ≠ x.key
2    if k < x.key
3        x = x.left
4    else x = x.right
5 return x

TREE-SUCCESSOR (x)
1 if x.right ≠ NIL
2    return TREE-MINIMUM (x.right)
3 else
4    y = x.p
5    while y ≠ NIL and x == y.right
6        x = y
7        y = y.p
8    return y
```

```
INORDER-TREE-WALK (x)
1 if x ≠ NIL
2    INORDER-TREE-WALK (x.left)
3    print x.key
4    INORDER-TREE-WALK (x.right)
```

Árbol rojinegro Un árbol rojinegro es una estructura de datos binaria de búsqueda que mantiene balanceadas sus ramas mediante reglas de coloración de nodos. Cada nodo tiene asignado un color rojo o negro, con reglas que garantizan que ninguna rama tenga una profundidad mayor que cualquier otra en más de una unidad. Esto asegura operaciones eficientes de inserción, eliminación y búsqueda, manteniendo la complejidad logarítmica para muchas operaciones clave.

```
RB-INSERT(T, z)
1 y = T.nil
2 x = T.root
3 while x != T.nil
4    y = x
5    if z.key < x.key
6       x = x.left
7    else
8       x = x.right
9 z.p = y
10 if y == T.nil
11    T.root = z
12 elseif z.key < y.key
13    y.left = z
14 else
15    y.right = z
16 z.left = T.nil
17 z.right = T.nil
18 z.color = RED
19 RB-INSERT-FIXUP(T, z)

RB-INSERT-FIXUP(T, z)
1 while z.p.color == RED
2    if z.p == z.p.p.left
3       y = z.p.p.right
4       if y.color == RED
5          z.p.color = BLACK
6          y.color = BLACK
7          z.p.p.color = RED
```

```
8           z = z.p.p
9       else
10          if z == z.p.right
11              z = z.p
12              LEFT-ROTATE(T, z)
13          z.p.color = BLACK
14          z.p.p.color = RED
15          RIGHT-ROTATE(T, z.p.p)
16    else
17        y = z.p.p.left
18        if y.color == RED
19            z.p.color = BLACK
20            y.color = BLACK
21            z.p.p.color = RED
22            z = z.p.p
23        else
24            if z == z.p.left
25                z = z.p
26                RIGHT-ROTATE(T, z)
27            z.p.color = BLACK
28            z.p.p.color = RED
29            LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

```
RB-DELETE(T, z)
1 y = z
2 y-original-color = y.color
3 if z.left == T.nil
4     x = z.right
5     RB-TRANSPLANT(T, z, z.right)
6 elseif z.right == T.nil
7     x = z.left
8     RB-TRANSPLANT(T, z, z.left)
9 else
10    y = TREE-MINIMUM(z.right)
11    y-original-color = y.color
12    x = y.right
13    if y.p == z
```

```
14        x.p = y
15    else
16        RB-TRANSPLANT(T, y, y.right)
17        y.right = z.right
18        y.right.p = y
19    RB-TRANSPLANT(T, z, y)
20    y.left = z.left
21    y.left.p = y
22    y.color = z.color
23 if y-original-color == BLACK
24    RB-DELETE-FIXUP(T, x)

RB-DELETE-FIXUP(T, x)
1 while x != T.root and x.color == BLACK
2    if x == x.p.left
3        w = x.p.right
4        if w.color == RED
5            w.color = BLACK
6            x.p.color = RED
7            LEFT-ROTATE(T, x.p)
8            w = x.p.right
9        if w.left.color == BLACK and w.right.color == BLACK
10           w.color = RED
11           x = x.p
12       else
13           if w.right.color == BLACK
14               w.left.color = BLACK
15               w.color = RED
16               RIGHT-ROTATE(T, w)
17               w = x.p.right
18           w.color = x.p.color
19           x.p.color = BLACK
20           w.right.color = BLACK
21           LEFT-ROTATE(T, x.p)
22           x = T.root
23    else
24        w = x.p.left
25        if w.color == RED
26            w.color = BLACK
```

```
27            x.p.color = RED
28            RIGHT-ROTATE(T, x.p)
29            w = x.p.left
30        if w.right.color == BLACK and w.left.color == BLACK
31            w.color = RED
32            x = x.p
33        else
34            if w.left.color == BLACK
35                w.right.color = BLACK
36                w.color = RED
37                LEFT-ROTATE(T, w)
38                w = x.p.left
39            w.color = x.p.color
40            x.p.color = BLACK
41            w.left.color = BLACK
42            RIGHT-ROTATE(T, x.p)
43            x = T.root
44 x.color = BLACK

RB-SEARCH(T, x, k)
1 if x == T.nil or k == x.key
2     return x
3 if k < x.key
4     return RB-SEARCH(T, x.left, k)
5 else
6     return RB-SEARCH(T, x.right, k)

RB-ITERATIVE-SEARCH(T, x, k)
1 while x != T.nil and k != x.key
2     if k < x.key
3         x = x.left
4     else
5         x = x.right
6 return x

RB-MAXIMUM(T, x)
1 while x.right != T.nil
2     x = x.right
3 return x
```

```
RB-MINIMUM(T, x)
1 while x.left != T.nil
2     x = x.left
3 return x

RB-SUCCESSOR(T, x)
1 if x.right != T.nil
2     return RB-MINIMUM(T, x.right)
3 y = x.p
4 while y != T.nil and x == y.right
5     x = y
6     y = y.p
7 return y

RB-INORDER-WALK(T, x)
1 if x != T.nil
2     RB-INORDER-WALK(T, x.left)
3     print x.key
4     RB-INORDER-WALK(T, x.right)
```

Tabla de dispersión (Hash Table) Una tabla de dispersión, también conocida como hash table en inglés, es una estructura de datos que utiliza una función hash para mapear claves a valores, permitiendo un acceso eficiente a los datos. Funciona mediante la aplicación de una función hash que transforma la clave en un índice en la tabla, donde se almacena el valor asociado. Esto permite realizar operaciones como inserción, eliminación y búsqueda en tiempo promedio constante, haciendo que las hash tables sean ideales para aplicaciones que requieren acceso rápido a datos almacenados.

```
constructor(sz)
1. size = sz
2. table = crear vector de tamaño size, cada elemento es una lista enlazada doble vacía

insertar(k)
1. index = hash(k)
2. Crear nuevo nodo newNode con clave k
3. Insertar newNode al inicio de la lista en table[index]

borrar(k)
1. index = hash(k)
2. nodo = buscar nodo con clave k en la lista en table[index]
3. Si nodo != null entonces
4.     Eliminar nodo de la lista en table[index]
```

```
buscar(k)
1. index = hash(k)
2. nodo = buscar nodo con clave k en la lista en table[index]
3. Si nodo == null entonces
4.     retornar null
5. de lo contrario
6.     retornar referencia a la clave en nodo

borrar(k)
1. index = hash(k)
2. nodo = buscar nodo con clave k en la lista en table[index]
3. Si nodo != null entonces
4.     Eliminar nodo de la lista en table[index]

hash(k)
1. return k % size
```