Estructuras de Datos

Ericka Araya-C20553

***Resumen*—En este trabajo, se llevó a cabo una evaluación exhaustiva del rendimiento de las estructuras de datos de lista enlazada simple y árbol de búsqueda binaria mediante la implementación de funciones de inserción y búsqueda. Los resultados revelaron que la lista enlazada simple exhibe un buen desempeño en operaciones de inserción, pero se queda rezagada en términos de eficiencia en búsquedas, especialmente cuando son aleatorias. Por otro lado, el árbol de búsqueda binaria mostró tiempos más altos en inserción, pero destacó significativamente en operaciones de búsqueda, tanto aleatorias como ordenadas. Estos hallazgos resaltan la importancia de comprender las fortalezas y debilidades de cada estructura de datos para seleccionar la más adecuada según los requisitos del problema.**

***Palabras clave*—estructura de datos, lista enlazada simple, árboles de búsqueda binaria, inserción, búsqueda.**

I. Introducción

En este trabajo, se profundiza en el análisis de los experimentos llevados a cabo para observar los tiempos de ejecución de las funciones de inserción y búsqueda de las estructuras de datos de Lista Enlazada Simple y Árboles de Búsqueda Binaria.

La función de insertar de la lista enlazada simple inserta un nodo "x.ªl principio de la lista, justo después del nodo centinela nil. Establece el siguiente nodo de "xçomo el primer nodo actual de la lista (que es el nodo siguiente de nil). Luego, actualiza el nodo centinela para que su siguiente nodo sea "x", lo que convierte a "x.en el nuevo primer nodo de la lista.

La función de buscar de la lista enlazada simple, busca iterativamente un nodo con una clave específica en la lista enlazada simple. Para simplificar la lógica de búsqueda, establece la clave del nodo centinela "nilçomo la clave buscada. Luego, comienza desde el primer nodo de la lista (el nodo siguiente a "nil") y recorre la lista comparando la clave de cada nodo con la clave buscada. Si encuentra un nodo con la clave buscada, devuelve ese nodo; de lo contrario, continúa hasta llegar nuevamente a "nil". Este método devuelve el nodo encontrado o "nil"si la clave no está presente en la lista, lo que facilita la identificación de resultados de búsqueda.

Por otro lado, la función insertar del árbol binario de búsqueda inserta un nodo ’z’ en la posición adecuada dentro del árbol para mantener su propiedad de orden. Comienza desde la raíz y busca iterativamente la posición correcta comparando la clave de ’z’, con las claves de los nodos existentes. Utiliza dos punteros, ’y’ y ’x’, para rastrear el nodo actual y su padre. Si la clave de ’z’ es menor que la del nodo actual ’x’, se mueve al subárbol izquierdo; si es mayor, al subárbol derecho. Una vez que encuentra una posición donde ’z’ es ’nullptr’, establece ’y’ como el padre de ’z’ y coloca a ’z’ como hijo izquierdo o derecho de y, según corresponda. Esto asegura que ’z’ se inserte en el lugar correcto para mantener la estructura de búsqueda del árbol.

Por último, la función de buscar para el árbol de búsqueda binario, busca un nodo con una clave específica en el árbol de búsqueda binario. realiza la búsqueda con un enfoque iterativo: comienza en el nodo "xz recorre el árbol moviéndose a los subárboles izquierdo o derecho según corresponda, hasta encontrar la clave o llegar a "nullptr". Ambos métodos devuelven el nodo que contiene la clave buscada o "nullptr"si no se encuentra.

En las pruebas se realizaron dos experimentos para cada función, donde en una ocasión se insertó 1 000 000 elementos con llaves seleccionados aleatoriamente en el rango [0, 2n) es decir, 0 <= x < 2n, siendo ’n’ el número de elementos a insertar(1 000 000) y se hizo una búsqueda de 10 000 llaves, en la otra prueba se insertó de igual manera 1 000 000 elementos, pero insertando llaves en orden de tal manera que se vaya insertando: 0, 1, ..., n - 1.

El artículo se estructura en dos partes principales: la primera se centra en el análisis individual de cada estructura de datos, evaluando la consistencia de los tiempos de inserción y búsqueda y las variaciones en el promedio de duración para las dos pruebas(Aleatoriamente y Ordenadamente) realizadas de las funciones ya dichas.

La segunda parte se encarga de comparar los tiempos promedio de las búsquedas, tanto con inserción aleatoria como con inserción ordenada, de ambas estructuras de datos, con el objetivo de comprender cómo varían los tiempos promedio de búsqueda utilizando diferentes estructuras de datos. Más específicamente, esta comparación tiene como objetivo identificar fortalezas y debilidades de cada algoritmo en términos de tiempo de ejecución.

Además, al comparar estructuras de datos también se pueden identificar casos de uso específicos para cada uno. Por ejemplo, la lista enlazada puede ser más eficiente en escenarios donde las inserciones frecuentes son primordiales, mientras que el árbol binario de búsqueda podría ser más adecuado para aplicaciones que requieren búsquedas rápidas y eficientes en conjuntos de datos más grandes y ordenados. La identificación de estos casos de uso ayudará a entender mejor cuándo y por qué elegir una estructura de datos sobre la otra, optimizando el rendimiento de los algoritmos en distintas aplicaciones prácticas.

Para la inserción en el árbol de búsqueda binaria, se implementó un método que garantiza la inserción de elementos manteniendo la propiedad de orden del árbol. El experimento de inserción ordenada para el caso del árbol se realizó mediante esta función, donde se construyó un árbol balanceado a partir de un arreglo de elementos. Este proceso implicó la implementación de un algoritmo que construye un árbol balanceado a partir de un vector de claves, utilizando un enfoque recursivo para dividir el arreglo en subárboles izquierdo y derecho de manera equilibrada. Este experimento tenía como objetivo

evaluar el rendimiento del árbol de búsqueda binaria en un escenario en el que los elementos están distribuidos de manera uniforme, lo que puede resultar en un árbol más equilibrado y, potencialmente, en mejores tiempos de búsqueda. El método de construcción del árbol balanceado se diseñó para garantizar una distribución uniforme de los elementos en el árbol, lo que podría contribuir a un mejor rendimiento en las operaciones de búsqueda.

II. Metodología

Para llevar a cabo la evaluación del rendimiento de los algoritmos de ordenamiento, se siguió un enfoque metodológico estructurado que implicaba varias etapas bien definidas.

1. Implementación de las funciones: Se implementaron las funciones de inserción y búsqueda de las estructuras de datos de Lista Enlaza Simple y Árbol de Búsqueda Binaria(estas hechas como clases), en un entorno de desarrollo en C++. con el objetivo de facilitar su reutilización y mantenimiento. Cada Función se desarrolló específicamente para hacer inserción y búsqueda de nodos con llaves en listas enlazadas simples y en árboles de búsqueda binaria.

2. Implementación del código de los algoritmos: El código de las funciones realizadas se basó en el pseudocódigo proporcionado en los libros "Data Structures and Algorithms" de Aho, Hopcroft y Ullman [1], así como en "Introduction to Algorithms" de Cormen, Leiserson, Rivest y Stein [2]. Se garantizó que la implementación de las funciones fuera fiel a los conceptos y técnicas descritas en estos libros.

3. Diseño del procedimiento experimental: Se diseñó un procedimiento experimental detallado para evaluar el tiempo de ejecución de las funciones de inserción y búsqueda, en ambas estructuras de datos. Este procedimiento incluyó la selección de conjuntos de datos que consistían en llaves aleatorias y llaves ordenadas. En cada experimento, se insertaron 1.000.000 de elementos en la estructura de datos y posteriormente se realizaron 10.000 búsquedas. Se aseguraron condiciones controladas para minimizar variaciones en los tiempos de ejecución debido a factores externos.

4. Registro y análisis de tiempos de ejecución: Se registraron los tiempos de cada ejecución de las funciones de inserción y búsqueda para los conjuntos de datos aleatorios y ordenados. Se calculó el tiempo promedio de ejecución para cada conjunto de datos y se organizó la información en tablas para facilitar su análisis. Además, se utilizaron técnicas de visualización de datos para crear gráficos que representen los tiempos promedio, lo que permitió analizar con mayor claridad las diferencias entre los tiempos promedio de inserción y búsqueda para las dos estructuras de datos.

5. Análisis de resultados: Se llevó a cabo un análisis detallado de los resultados obtenidos, centrándose en identificar patrones y diferencias significativas en los tiempos de ejecución de las funciones de inserción y búsqueda. Se examinaron las barras y curvas producidas por las gráficas para comprender cómo variaban los tiempos de ejecución en función de la naturaleza de los datos insertados (aleatorios y ordenados). Este análisis permitió evaluar las fortalezas y debilidades de cada estructura de datos en distintos escenarios de inserción y búsqueda, proporcionando una comprensión más profunda de su rendimiento y eficiencia.

III. Resultados

Los tiempos de ejecución de las corridas de ambas estructuras se muestran en los cuadros I y II.

Cuadro I
TIEMPO DE EJECUCIÓN DE OPERACIONES PARA LA ESTRUCTURA DE DATOS DE LISTA ENLAZADA SIMPLE.

		Tiempo (ms)			
		Corrida			
Operación	Elementos	1	2	3	Prom
Inserción aleatoria	1000000	75	89	72	79
Inserción ordenada	1000000	46	45	45	45
Búsqueda aleatoria	10000	40594	40516	42809	41306
Búsqueda ordenada	10000	38569	38722	38755	38682

Cuadro II
TIEMPO DE EJECUCIÓN DE OPERACIONES PARA LA ESTRUCTURA DE DATOS DE ÁRBOL DE BÚSQUEDA BINARIA.

		Tiempo (ms)			
		Corrida			
Operación	Elementos	1	2	3	Prom
Inserción aleatoria	1000000	1088	1476	1572	1379
Inserción ordenada	1000000	90	89	85	88
Búsqueda aleatoria	10000	12	15	16	14
Búsqueda ordenada	10000	3	3	3	3

Para empezar, al analizar la estructura de datos de lista enlazada simple, observamos que su desempeño es bastante consistente tanto en las operaciones de inserción como en las de búsqueda. En el caso de la inserción, vemos que los tiempos son relativamente bajos, con una ligera diferencia entre inserción aleatoria y ordenada. Sin embargo, la verdadera disparidad se presenta en las operaciones de búsqueda. Aquí es donde la lista enlazada simple muestra una notable debilidad, especialmente en el caso de la búsqueda aleatoria, donde los tiempos son considerablemente más altos en comparación con la búsqueda ordenada. Este comportamiento sugiere que la estructura de lista enlazada simple puede no ser la más adecuada para aplicaciones que requieren un alto rendimiento en operaciones de búsqueda, especialmente cuando se trata de búsquedas no secuenciales. Gráficos 1 y 2.

Por otro lado, al analizar el árbol de búsqueda binaria, observamos un rendimiento más variado dependiendo del tipo de operación. En el caso de la inserción, vemos que los tiempos son generalmente más altos que los de la lista enlazada simple, especialmente para la inserción aleatoria. Sin embargo, para la inserción ordenada, el árbol de búsqueda binaria muestra un rendimiento comparable, e incluso ligeramente mejor en algunas corridas. Donde realmente brilla el árbol de búsqueda binaria es en las operaciones de búsqueda. Tanto

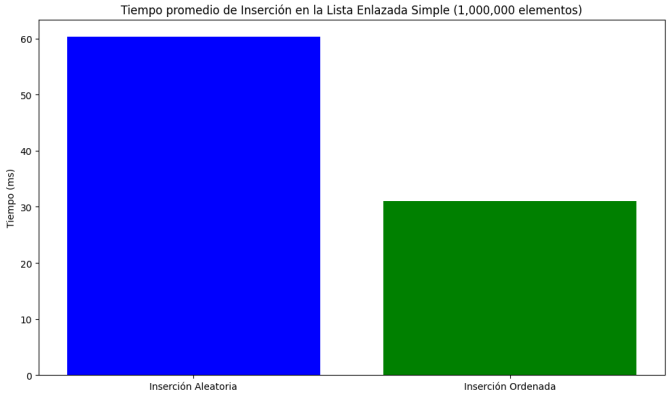

Figura 1. Tiempos promedio de inserción en la Lista Enlazada Simple.

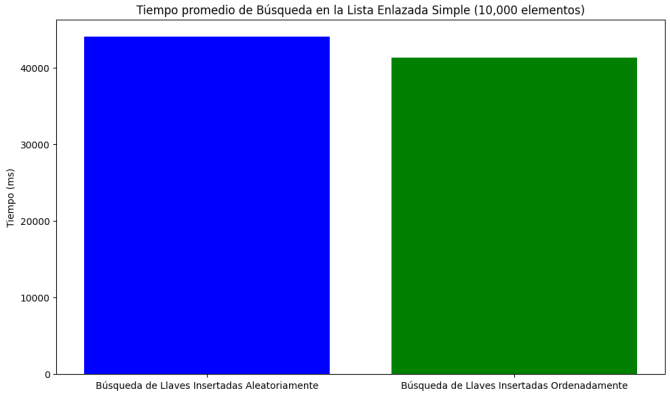

Figura 2. Tiempos promedio de búsqueda en la Lista Enlazada Simple.

para búsquedas aleatorias como ordenadas, el árbol de búsqueda binaria presenta tiempos significativamente más bajos en comparación con la lista enlazada simple. Esto sugiere que el árbol de búsqueda binaria es una opción más eficiente en términos de búsqueda, especialmente para conjuntos de datos grandes o cuando la distribución de las claves no es conocida de antemano, donde hay un incremento gradual en el tiempo de ejecución, en línea con las expectativas del algoritmo. Gráficos 3 y 4.

Al comparar la lista enlazada simple y el árbol de búsqueda

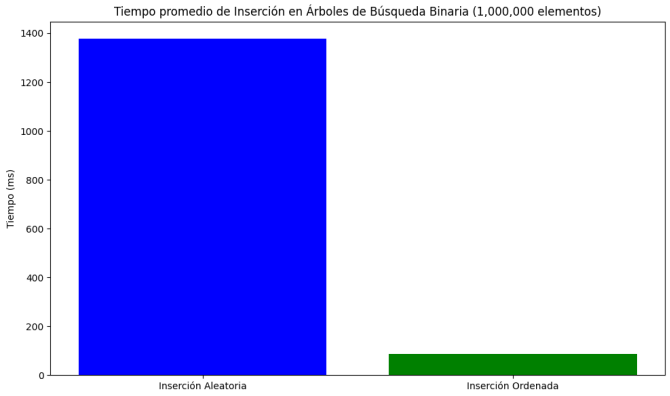

Figura 3. Tiempos promedio de inserción en Árboles de Búsqueda Binaria.

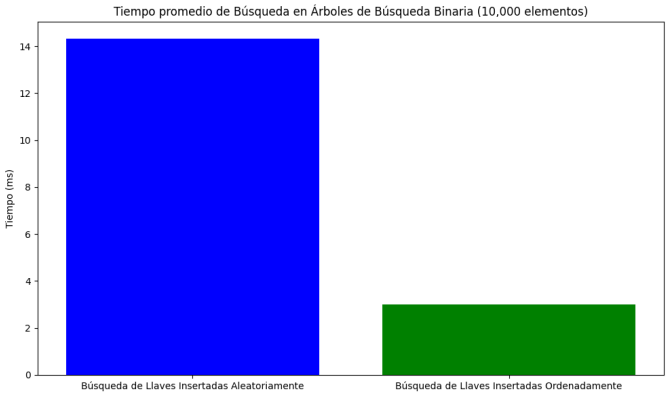

Figura 4. Tiempos promedio de búsqueda en Árboles de Búsqueda Binaria.

binaria en términos de rendimiento y eficiencia, se observan patrones distintivos que resaltan las fortalezas y debilidades de cada estructura de datos. La lista enlazada simple, al priorizar la inserción rápida al principio de la lista, demuestra ser eficaz en escenarios donde las adiciones son frecuentes en esta ubicación. No obstante, su rendimiento disminuye considerablemente para inserciones en posiciones intermedias o finales, y las operaciones de búsqueda lineales pueden resultar ineficientes en conjuntos de datos extensos.

Por otro lado, el árbol de búsqueda binaria ofrece tiempos de búsqueda logarítmicos, lo que lo hace ideal para operaciones de búsqueda en conjuntos de datos grandes y variados. Sin embargo, mantener el equilibrio del árbol puede suponer una sobrecarga adicional en las operaciones de inserción, lo que puede afectar su eficiencia en comparación con la lista enlazada simple en ciertos contextos.

En última instancia, la elección entre estas dos estructuras de datos depende de la naturaleza específica de la aplicación y sus requisitos de rendimiento. Mientras que la lista enlazada simple puede ser preferible para aplicaciones donde las inserciones son predominantemente al principio de la lista y las búsquedas son secuenciales, el árbol de búsqueda binaria sobresale en escenarios que requieren búsquedas eficientes y una estructura de datos más equilibrada. Gráficos 5 y 5

En conclusión, este estudio ha proporcionado una visión detallada del rendimiento de las estructuras de datos de lista enlazada simple y árbol de búsqueda binaria en términos de operaciones de inserción y búsqueda. Se observó que la lista enlazada simple muestra un buen rendimiento en operaciones de inserción, especialmente cuando los elementos se insertan en orden, pero sufre en términos de eficiencia en las operaciones de búsqueda, especialmente para búsquedas no secuenciales. Por otro lado, el árbol de búsqueda binaria ofrece tiempos más bajos en las operaciones de búsqueda, tanto para búsquedas aleatorias como ordenadas, aunque sus tiempos de inserción pueden ser más altos en comparación con la lista enlazada simple. Estos hallazgos destacan la importancia de seleccionar la estructura de datos adecuada según los requisitos específicos de cada aplicación, considerando tanto la distribución de datos como el tipo de operaciones que se realizarán con mayor frecuencia.

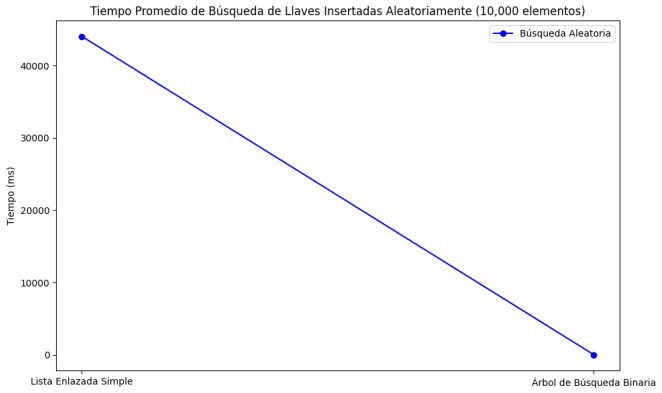

Figura 5. Gráfico comparativo de los tiempos promedio de búsqueda de llaves insertadas aleatoriamente.

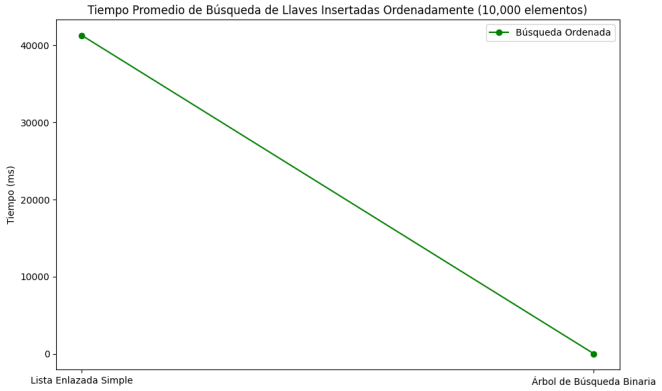

Figura 6. Gráfico comparativo de los tiempos promedio de búsqueda de llaves insertadas ordenadamente.

REFERENCIAS

[1] A. Alfred V, H. John E, and U. Jeffrey D, *Data Structures and Algorithms*, 1st ed. Micheal A, Harrison, 1983.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. The MIT Press, 2009.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

El código se muestra en los algoritmos 1, 2,

Algoritmo 1 El algoritmo LIST-SEARCH busca de manera iterativa una clave k en una lista enlazada simple L. El algoritmo LIST-INSERT inserta un nuevo nodo x después de un nodo y en una lista enlazada simple. Actualiza los punteros del nodo y para enlazar correctamente el nuevo nodo x en la lista.

```
LIST-SEARCH (L, k)
    x = L.head
    while x != NIL and x.key != k
    x = x.next
    return x
LIST-INSERT (x, y)
    x.next = y.next
    if y.next != NIL
        y.next = x
    y.next = x
```

Algoritmo 2 El algoritmo ITERATIVE-TREE-SEARCH busca de manera iterativa una clave k en un árbol binario de búsqueda (BST). El algoritmo TREE-INSERT inserta de manera iterativa un nuevo nodo z en un árbol binario de búsqueda T, manteniendo la propiedad de orden del árbol. :

```
ITERATIVE-TREE-SEARCH (x, k)
    while x != NIL and k != x.key
        if k < x.key
            x = x.left
        else x = x.right
    return x
TREE-INSERT (T, z)
    x = T.root
    y = NIL
    while x != NIL
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == NIL
    T.root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
```

.