Reporte de Tarea 2

Liqing Yosery Zheng Lu, C38680, liqing.zheng@ucr.ac.cr

***Resumen*—El presente trabajo aborda la comparación entre los tiempos de ejecución en la búsqueda y eliminación en estructuras de datos con elementos ordenados y no ordenados. Con el objetivo de evaluar cómo se ajustan las predicciones teóricas a mediciones prácticas, se implementaron en C++ cinco estructuras: lista simplemente enlazada, lista doblemente enlazada, árbol de búsqueda binaria, árbol rojinegro, y una tabla de dispersión. Para ello se generaron arreglos aleatorios, uno de 1 000 000 con elementos no repetidos, otro también de 1 000 000 pero con elementos repetidos, y dos 10 000, uno para buscar y otro para eliminar. Se midió el tiempo de ejecución promedio en tres corridas mediante la biblioteca `chrono`. Los resultados evidencian que el rendimiento práctico de las estructuras depende no solo de su diseño teórico, sino también del patrón de los datos con que se alimentan. En particular, el árbol binario de búsqueda mostró una caída significativa en su eficiencia al insertar elementos ordenados, debido a la degeneración estructural hacia una forma lineal. El árbol rojinegro, por su parte, se mantuvo eficiente en ambos escenarios, aunque se observaron ligeras diferencias atribuibles a la complejidad de los casos internos del balanceo. En la tabla de dispersión, el orden de los datos también impactó el rendimiento debido a la función hash utilizada, la cual generó agrupamientos no deseados cuando los datos seguían una progresión regular. En cambio, en las listas enlazadas, las diferencias entre los casos ordenados y aleatorios fueron mínimas, aunque se identificaron patrones que podrían ser optimizados. Se concluye que factores como el orden de inserción, la función hash elegida y la implementación concreta pueden tener un impacto sustancial en el comportamiento real de las estructuras de datos.**

***Palabras clave*—Estructuras de datos, búsqueda, eliminación, lista simplemente enlazada, árbol de búsqueda binaria, árbol rojinegro, tabla de dispersión**

I. Introducción

El presente trabajo tiene como propósito analizar y comparar el tiempo de ejecución de las operaciones de búsqueda y eliminación en diversas estructuras de datos bajo dos escenarios distintos: cuando los datos están ordenados y cuando se encuentran en un orden aleatorio. Esta comparación es de interés para determinar en qué situaciones resulta beneficioso ordenar previamente los datos en función del tipo de estructura utilizada.

Las estructuras seleccionadas para este estudio son: la lista simplemente enlazada, el árbol binario de búsqueda, el árbol rojinegro y la tabla de dispersión con encadenamiento. Cada una de estas estructuras presenta distintas complejidades teóricas para las operaciones consideradas, dependiendo de factores como el orden de inserción y el balanceo de los nodos.

En términos generales, se sabe que las operaciones de búsqueda y eliminación en listas simplemente enlazadas presentan una complejidad temporal de $O(n)$ en el peor caso. En el árbol binario de búsqueda, estas operaciones también pueden alcanzar $O(n)$ cuando el árbol está desbalanceado, aunque en el caso promedio se espera una complejidad de $O(\log n)$. El árbol rojinegro, al ser un árbol balanceado por diseño, garantiza una complejidad de $O(\log n)$ en el peor caso para dichas operaciones. Por su parte, las tablas de dispersión con encadenamiento ofrecen un rendimiento promedio constante, $O(1)$, aunque pueden degradarse hasta $O(n^2)$ en el peor caso si se presentan colisiones excesivas y listas de encadenamiento largas.

Este análisis observar en la práctica qué tan sensibles son estas estructuras a la naturaleza de los datos insertados.

II. Metodología

Con el fin de evaluar de manera objetiva el desempeño de las estructuras mencionadas, se generan distintos arreglos que servirán como base para las operaciones de inserción, búsqueda y eliminación. En particular, se emplean los siguientes tipos de arreglos de entrada:

- Un arreglo aleatorio de un millón de elementos que permite duplicados, utilizado para la lista simplemente enlazada.
- Un arreglo aleatorio de un millón de elementos únicos, utilizado para el árbol binario de búsqueda, el árbol rojinegro y la tabla de dispersión, ya que estas estructuras no admiten elementos repetidos.

Para generar el arreglo de un millón de elementos únicos de forma aleatoria, se construye primero un arreglo de tamaño tres millones con los enteros del rango $[0, 2\,999\,999]$, que son todos únicos por definición. Luego, se aplica un algoritmo de mezcla aleatoria (`shuffle`) sobre ese arreglo. Finalmente, se toman los primeros un millón de elementos resultantes como el arreglo aleatorio sin repetición.

Además, se crean dos arreglos adicionales de diez mil elementos con posibles duplicados, que serán empleados para las operaciones de búsqueda y eliminación en todas las estructuras, asegurando así condiciones equitativas durante las pruebas.

Todos los elementos generados pertenecen al rango de enteros $[0, 2\,999\,999]$ y fueron producidos mediante una función de generación pseudoaleatoria. Para medir el rendimiento, se utiliza la biblioteca estándar de C++ `chrono`, registrando los tiempos en microsegundos.

La inserción ordenada se simula utilizando un ciclo for. No se modifica el método de inserción de cada estructura, excepto en el caso del árbol binario de búsqueda, donde se implementa un método alternativo *fast insert*. Este método inserta primero el valor central de un arreglo ordenado, seguido recursivamente por los elementos más cercanos al centro, lo cual promueve un árbol más balanceado y mejora el rendimiento de búsqueda y eliminación.

La inserción ordenada se simula utilizando un ciclo `for`, en el que se insertan los elementos en orden creciente, lo cual

hace a los datos de la estructura ordenados. No se modifica el método de inserción de ninguna estructura, excepto en el caso del árbol binario de búsqueda, donde también se implementa un método alternativo llamado `fastInsert`. El método `fastInsert` no realiza búsquedas para encontrar la posición correcta de cada nodo. En su lugar, crea directamente una cadena de nodos conectados todos hacia la derecha. Para ello, simplemente recorre un ciclo desde 0 hasta $n-1$ e inserta los valores uno tras otro como hijos derechos del nodo anterior. Como cada inserción se realiza en tiempo constante $O(1)$, la complejidad total del proceso es $O(n)$. Esto contrasta con el método de inserción tradicional del árbol binario de búsqueda, en el que insertar n elementos ordenados hace que cada inserción tome tiempo proporcional a la altura actual del árbol. En ese caso, la complejidad total es:

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

Por lo tanto, `fastInsert` es una alternativa mucho más eficiente para los fines de este proyecto, ya que genera exactamente la misma estructura desbalanceada que se obtendría al insertar los elementos uno por uno mediante el método tradicional, pero con un costo computacional significativamente menor.

Cada prueba de búsqueda y eliminación se repite tres veces por estructura y por tipo de dato (ordenado y aleatorio), con el fin de obtener resultados más representativos. Los tiempos de ejecución se registran en tablas específicas por estructura, las cuales incluyen los resultados de las tres corridas para cada operación (búsqueda y eliminación) en ambos escenarios (datos ordenados y aleatorios). En dichas tablas también se calcula el promedio correspondiente para cada combinación de operación y tipo de dato.

El tiempo se mide utilizando la biblioteca `chrono` de C++. Dado que algunas estructuras son teóricamente más lentas, se usan diferentes unidades para mejorar la legibilidad de los resultados. Se utiliza milisegundos para la lista simplemente enlazada y el árbol binario de búsqueda, y microsegundos para el árbol rojinegro y la tabla de dispersión con encadenamiento.

Con base en los promedios obtenidos, se generarán gráficos de barras para cada estructura, comparando el desempeño entre datos ordenados y aleatorios, tanto en las operaciones de búsqueda como de eliminación. Además, se elaborarán gráficos comparativos entre todas las estructuras, agrupados por tipo de operación y tipo de inserción, de acuerdo con los siguientes criterios:

1. Comparación del tiempo de búsqueda con llaves insertadas aleatoriamente.
2. Comparación del tiempo de búsqueda con llaves insertadas de forma ordenada.
3. Comparación del tiempo de eliminación con llaves insertadas aleatoriamente.
4. Comparación del tiempo de eliminación con llaves insertadas de forma ordenada.

Estos gráficos permitirán contrastar el rendimiento relativo entre estructuras y evaluar qué tan alineados están los resultados experimentales con las expectativas teóricas. Asimismo, servirán como base para el análisis comparativo en la discusión del informe.

Cuadro I
TIEMPOS DE EJECUCIÓN PARA LAS OPERACIONES DE BÚSQUEDA Y ELIMINACIÓN DE LA LISTA SIMPLEMENTE ENLAZADA.

Operación	Buscar		Eliminar	
Corrida	Ordenado	Aleatorio	Ordenado	Aleatorio
1	12546	12301	16569	16298
2	12509	12211	16456	16236
3	12572	12353	16613	16237
Promedio (ms)	12542,3	12288,3	16546	16257

III. RESULTADOS

Los tiempos de ejecución de las tres corridas para cada estructura de datos se presentan en los cuadros I a IV, cada uno correspondiente a una estructura distinta: lista simplemente enlazada, árbol binario de búsqueda, árbol rojinegro y tabla de dispersión con encadenamiento. Todas las estructuras se probaron con las mismas entradas en cada corrida. Lo anterior se hizo para que la comparación entre estructuras fuera justa. Como se usó la misma computadora y los mismos datos en cada ronda, los promedios no varían mucho entre corridas. Además, se verificó con `Valgrind` que no hubiera fugas de memoria durante las pruebas, y todas las estructuras gestionaron correctamente los recursos utilizados.

Lista: La operación buscar es más rápida que eliminar, pero es porque eliminar ocupa buscar primero. ABB: buscar y eliminar tienen tiempos relativamente parecidos, pero lo que cambia mucho es ordenado o aleatorio. Rojinegro: buscar es más rapido que eliminar, también porque ocupa llamar a buscar primero, igual que la lista. Hash: buscar es mas rapdio porque ocupa encontrarlo antes de eliminar

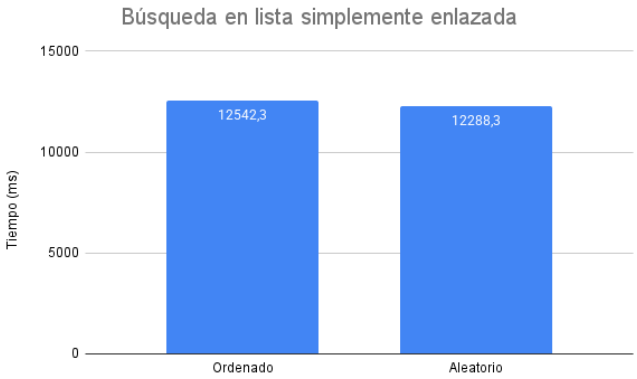

Figura 1. Lista ordenada contra lista aleatoria en operación búsqueda.

La figura 1 no muestra grandes diferencias entre la búsqueda en una lista simplemente enlazada ordenada de una no ordenada. El resultado sí fue el esperado porque se les aplicó el mismo algoritmo.

La figura 2 no muestra grandes diferencias entre la eliminación de elementos en una lista simplemente enlazada

Figura 2. Lista ordenada contra lista aleatoria en operación eliminar.

Cuadro II
TIEMPOS DE EJECUCIÓN PARA LAS OPERACIONES DE BÚSQUEDA Y ELIMINACIÓN DEL ÁRBOL DE BÚSQUEDA BINARIA.

Operación	Buscar		Eliminar	
Corrida	Ordenado	Aleatorio	Ordenado	Aleatorio
1	20233	3	20490	3
2	20007	3	20069	3
3	20192	3	20304	3
Promedio (ms)	20144	3	20287,7	3

ordenada de una no ordenada. El resultado también fue el esperado porque igual que la búsqueda, se les aplicó el mismo algoritmo.

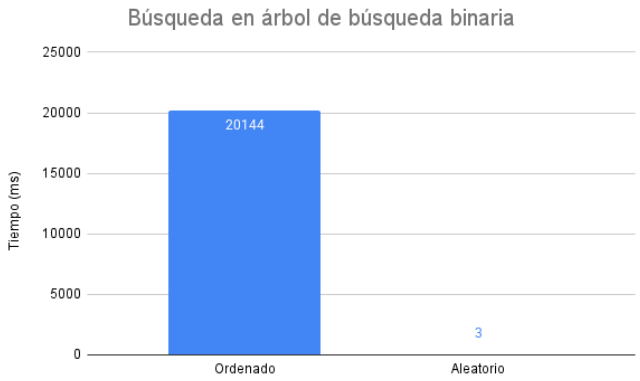

Figura 3. Árbol binario ordenado contra árbol binario aleatorio en operación búsqueda.

La figura 3 muestra una diferencia considerable en la operación de búsqueda. Sin embargo, el resultado sí fue el esperado porque un árbol cuyos nodos se le insertaron en orden es un árbol degenerado, o sea, que tiene una o múltiples ramas mucho más largas. Por ende, se comporta parecido a una lista enlazada.

La figura 4 también muestra una diferencia considerable en la operación de eliminar. Tal como en la operación buscar, el resultado sí fue el esperado por la misma razón. El árbol al

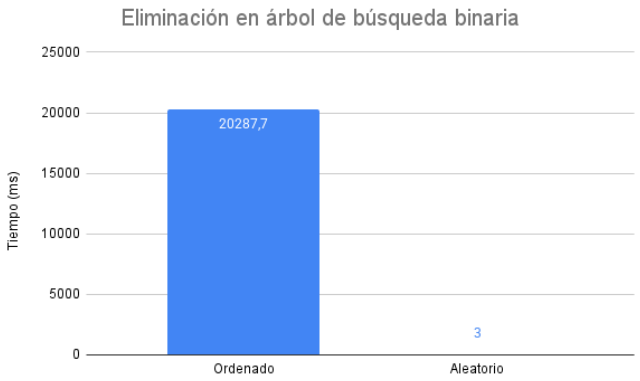

Figura 4. Árbol binario ordenado contra árbol binario aleatorio en operación eliminar.

Cuadro III
TIEMPOS DE EJECUCIÓN PARA LAS OPERACIONES DE BÚSQUEDA Y ELIMINACIÓN DEL ÁRBOL ROJINEGRO.

Operación	Buscar		Eliminar	
Corrida	Ordenado	Aleatorio	Ordenado	Aleatorio
1	1906	3608	2472	4227
2	2411	3166	2810	3546
3	1550	3195	2044	3563
Promedio (μs)	1955,7	3323	2442	3778,7

que se le insertan nodos en orden no está balanceado y eso repercute directamente en su tiempo de ejecución.

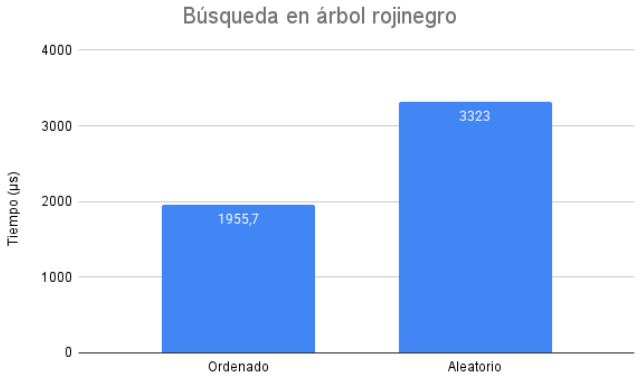

Figura 5. Árbol rojinegro ordenado contra árbol rojinegro aleatorio en operación búsqueda.

La figura 5 muestra algo de diferencia en la operación de búsqueda en la estructura árboles rojinegros. El resultado no fue el esperado porque usando el mismo algoritmo, deberían de tener un tiempo de ejecución parecido.

La figura 6 también muestra algo diferencia en la operación de eliminar. El resultado tampoco fue el esperado porque para eliminar también usan el mismo algoritmo.

La figura 7 muestra un poco de diferencia en la operación de búsqueda, siendo el aleatorio más rápido. El resultado sí fue esperado debido a la función hash que se usó. Como se

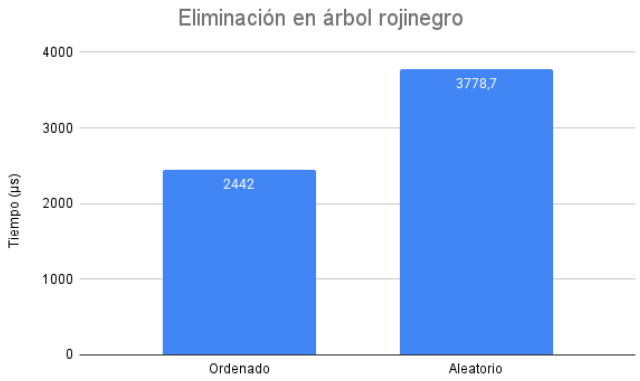

Figura 6. Árbol rojinegro ordenado contra árbol rojinegro aleatorio en operación eliminar.

Cuadro IV
TIEMPOS DE EJECUCIÓN PARA LAS OPERACIONES DE BÚSQUEDA Y ELIMINACIÓN DE LA TABLA DE DISPERSIÓN.

Operación	Buscar		Eliminar	
Corrida	Ordenado	Aleatorio	Ordenado	Aleatorio
1	628	562	790	828
2	691	570	881	802
3	624	563	776	811
Promedio (μs)	647,7	565	815,7	813,7

discutirá más adelante, hubieron más colisiones en la tabla de dispersión ordenada.

La figura 8 casi no muestra diferencias, el cual tiene sentido porque la función hash utilizada fue deficiente y ambas tablas tuvieron que encadenar bastantes valores.

En la figura 9 se puede observar que la estructura más ágil para buscar cuando tiene elementos insertados aleatoriamente, es la tabla de dispersión. Teóricamente, sí es lo esperado porque buscar en una tabla de dispersión es de complejidad $O(1)$. Como la contraparte se tiene a la lista simplemente enlazada, que es la más lenta. Sin embargo, teóricamente también es lo esperado.

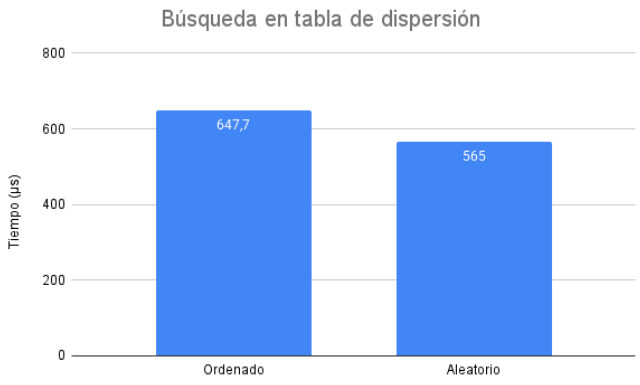

Figura 7. Tabla de dispersión ordenada contra tabla de dispersión aleatoria en operación búsqueda.

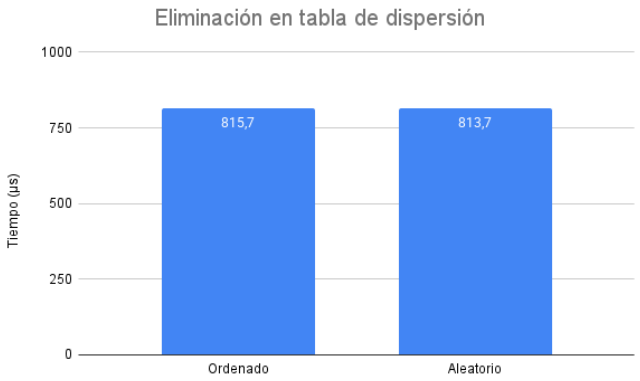

Figura 8. Tabla de dispersión ordenada contra tabla de dispersión aleatoria en operación eliminar.

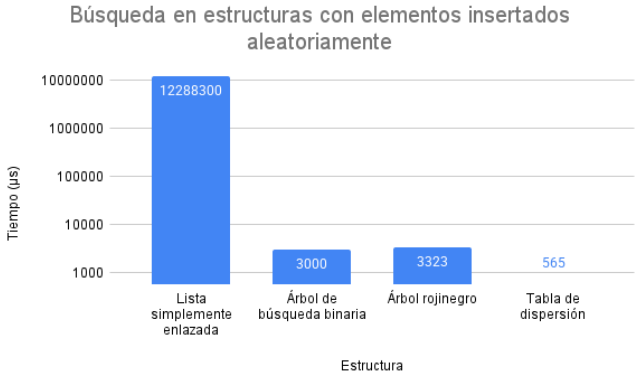

Figura 9. Búsqueda en estructuras con elementos insertados aleatoriamente.

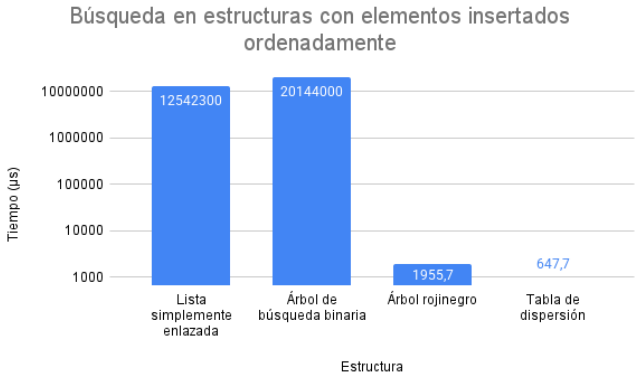

Figura 10. Búsqueda en estructuras con elementos insertados ordenadamente.

En la figura 10 se puede observar que, nuevamente, la tabla de dispersión es la más ágil para buscar. También se puede notar que el árbol de búsqueda binario, que está degenerado debido a la inserción de elementos ordenados, es incluso más lenta que la lista simplemente enlazada. Sí se esperaba que el árbol binario ordenado fuera lento, pero no más que la lista ya que se comporta como una.

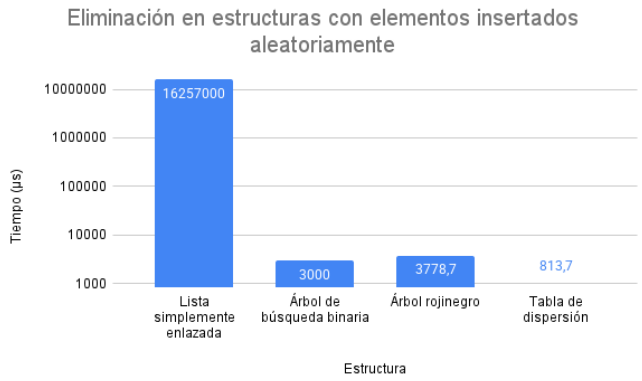

Figura 11. Eliminación en estructuras con elementos insertados aleatoriamente.

Los resultados de la figura 11 sí fueron los esperados. El árbol de búsqueda binaria no dura tanto ejecutando la operación eliminar si sus elementos no fueron insertados ordenadamente. El árbol rojinegro en este caso fue más lento que el árbol de búsqueda binario, probablemente debido a todas las correcciones para mantener sus propiedades. Para efectos del presente trabajo, este es el umbral en el que un árbol de búsqueda binaria es más rápida que un árbol rojinegro, que está balanceado.

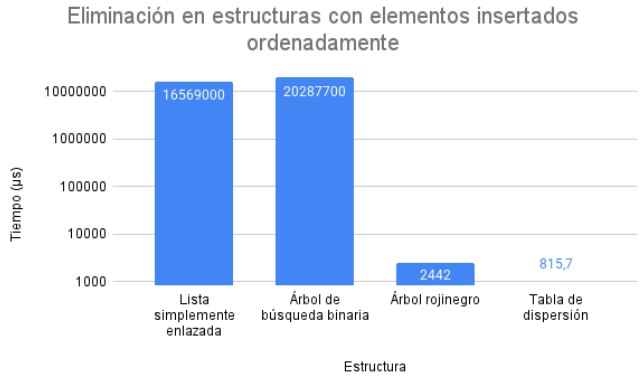

Figura 12. Eliminación en estructuras con elementos insertados ordenadamente.

Los resultados de la figura 12 también fueron los esperados. Siendo el árbol de búsqueda binaria la que más dura debido a su degeneración. Seguidamente de la lista simplemente enlazada que no aprovecha de su condición de ordenado.

La tabla de dispersión fue la más rápida, tanto ordenada como no, tanto para eliminación como búsqueda. Esto es esperado, ya que el caso promedio de las operaciones buscar y eliminar es $O(1)$. La estructura más lenta en promedio es la lista simplemente enlazada porque, aunque el árbol de búsqueda binario es más lenta cuando está ordenada, no es usual encontrar a un árbol binario de esa manera.

IV. Discusión

IV-A. *Listas simplemente enlazadas*

Se puede observar en el Cuadro I que las operaciones de búsqueda y eliminación en la lista simplemente enlazada presentan tiempos similares tanto en el caso ordenado como en el aleatorio. De hecho, el tiempo promedio para la lista con datos ordenados resultó ligeramente mayor que el de la lista con datos aleatorios. Este comportamiento probablemente se debe a que la implementación utilizada no aprovecha el orden de los datos, porque siguiendo la metodología se empleó el mismo algoritmo de recorrido secuencial en ambos casos.

En listas simplemente enlazadas, la búsqueda requiere recorrer nodo por nodo hasta encontrar la llave o llegar al final. Si los datos están ordenados y se implementa una verificación que detenga el recorrido tan pronto se encuentre un valor mayor a la llave buscada, es posible evitar recorrer innecesariamente toda la lista. Este tipo de optimización no fue implementado, por lo que el recorrido completo se realizó en ambos casos. Sin embargo, eso deja a la intrigante del porqué la lista enlazada aleatoria fue levemente más rápida cuando usando el mismo algoritmo debería de dar el mismo tiempo.

Una posible explicación adicional del peor rendimiento en la lista ordenada es que, al estar los datos organizados de menor a mayor, las búsquedas de llaves grandes obligan a recorrer casi toda la lista. En cambio, en la versión aleatoria, algunas llaves pueden aparecer más cerca del inicio, lo que puede reducir el recorrido promedio y, por consiguiente, el tiempo de ejecución.En el caso de la eliminación, también se depende de una búsqueda previa para localizar el nodo a eliminar. Por tanto, cualquier mejora en el proceso de búsqueda repercutiría directamente en el tiempo de eliminación.

IV-B. *Árboles de búsqueda binaria*

Se puede observar en el Cuadro II que el árbol binario de búsqueda presenta un rendimiento significativamente peor cuando las llaves se insertan en orden, en comparación con el caso de inserción aleatoria. Tal como se menciona en *Introducción a los algoritmos* [1], un árbol binario de búsqueda mantiene una altura esperada de $\Theta(\log n)$ únicamente cuando las llaves se insertan en un orden aleatorio. Sin embargo, si se insertan en orden ascendente o descendente, el árbol se degrada en una estructura lineal, similar a una lista enlazada, lo cual explica los tiempos de ejecución considerablemente más altos observados en el escenario ordenado.

En este proyecto se utilizó el método `fastInsert` para generar un árbol ordenado de forma eficiente, insertando los nodos directamente hacia un solo lado sin búsquedas. Aunque esta estrategia reduce el tiempo de construcción (en comparación con inserciones tradicionales), no cambia el hecho de que el árbol resultante esté completamente desbalanceado. Esta gran profundidad impacta negativamente en el tiempo de operaciones como búsqueda y eliminación, que dependen de la altura del árbol.

Cabe destacar que, debido al enfoque metodológico adoptado, no se midieron los tiempos de inserción directamente. No obstante, los efectos del desbalance estructural pueden

observarse claramente en el desempeño de las operaciones de búsqueda y eliminación.

IV-C. Árboles rojinegros

En el Cuadro III se observa que las operaciones de búsqueda y eliminación en el árbol rojinegro son, en promedio, más rápidas cuando los datos fueron insertados en orden. Aunque esta estructura se caracteriza por mantenerse balanceada con una altura del orden de $O(\log n)$ independientemente del orden de inserción, este resultado plantea una interrogante interesante.

Una posible explicación radica en el proceso de balanceo que realiza el árbol durante las inserciones y eliminaciones, específicamente en los procedimientos de `insert-fixup` y `delete-fixup`. Ambos procedimientos tienen una complejidad de $O(\log n)$ en el peor caso. Sin embargo, cuando las llaves se insertan de forma ordenada, es común que el árbol caiga en casos simples del `insert-fixup`, como aquellos que requieren únicamente un recoloreo o una rotación sencilla. En contraste, el patrón aleatorio de inserción puede generar configuraciones más variadas y costosas, provocando que el árbol entre en casos más complejos del `fixup`, afectando ligeramente el tiempo de las operaciones posteriores.

Así, aunque teóricamente ambas versiones del árbol rojinegro mantienen una estructura balanceada, el patrón de inserción puede influir en la cantidad de trabajo que se realiza internamente durante las operaciones de mantenimiento, lo cual se refleja en los tiempos de ejecución medidos.

IV-D. Tablas de dispersión

En el Cuadro IV se observa que las operaciones de búsqueda y eliminación en la tabla de dispersión con encadenamiento son ligeramente más lentas cuando los datos fueron insertados en orden, en comparación con la versión aleatoria. Aunque en teoría el rendimiento de una tabla hash no debería depender del orden de inserción de las llaves, esto solo se cumple si la función hash logra distribuir uniformemente los valores en las posiciones del arreglo.

En este proyecto se utilizó una función hash simple de la forma $h(k) = k \bmod m$. Este tipo de función puede verse afectada por patrones regulares en los datos. Por ejemplo, al insertar valores consecutivos como $0, 1, 2, \ldots, n$ en una tabla con $m = 10$, las llaves se ubican secuencialmente en las posiciones 0 a 9, y luego comienzan a colisionar cíclicamente. Esto genera listas encadenadas de tamaños muy dispares, lo que incrementa el tiempo de recorrido durante la búsqueda y eliminación.

Este efecto es menos probable en el caso aleatorio, donde la dispersión de las llaves tiende a ser más uniforme, incluso con una función hash sencilla. Así, la diferencia observada entre los tiempos del caso ordenado y el aleatorio puede atribuirse a un agrupamiento no deseado causado por la naturaleza regular de los datos y las limitaciones de la función hash utilizadas.

V. Conclusiones

Tal como se planteó en la introducción, se compararon listas simplemente enlazadas, árboles de búsqueda binaria, árboles rojinegros y tablas de dispersión con encadenamiento bajo dos escenarios: datos ordenados y datos aleatorios. Los resultados obtenidos reflejan en gran medida el comportamiento teórico esperado. En estructuras como el árbol de búsqueda binaria, se observó un impacto negativo considerable cuando los datos se insertaron en orden, debido a la degeneración estructural que provoca una mayor altura. En contraste, estructuras balanceadas como el árbol rojinegro mantuvieron tiempos de operación eficientes en ambos casos, aunque se identificaron ligeras mejoras cuando las inserciones seguían un patrón más predecible. En el caso de la lista simplemente enlazada, los resultados fueron similares entre ordenado y aleatorio, con diferencias mínimas atribuibles al posicionamiento de las llaves. Finalmente, en la tabla de dispersión, se observó que una función hash sencilla como k mód m puede producir colisiones más pronunciadas en inserciones ordenadas, lo que sugiere la necesidad de usar funciones de dispersión más robustas en aplicaciones reales.

Una posible mejora para trabajos futuros sería medir también los tiempos de inserción, lo cual permitiría completar el análisis de eficiencia de cada estructura en todos sus aspectos fundamentales. También sería beneficioso implementar versiones optimizadas de las operaciones que aprovechen el orden de los datos cuando esté presente (por ejemplo, salidas tempranas en listas ordenadas). Además, se podrían explorar funciones hash alternativas.

En resumen, los resultados experimentales permiten confirmar que el rendimiento de las estructuras de datos no solo depende de su diseño teórico, sino también de detalles prácticos como el orden de los datos y la implementación de los algoritmos asociados. Este tipo de análisis es fundamental para tomar decisiones informadas al seleccionar estructuras en aplicaciones reales.

Referencias

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

Liqing Yosery Zheng Lu Estudiante de computación en la Universidad de Costa Rica. Lleva el curso de Algoritmos y Estructuras de Datos durante el primer ciclo del año 2025 en la sede Rodrigo Facio.