Estructuras de Datos

Andres Murillo Murillo C15424

***Resumen* — Las estructuras de datos son fundamentales para la optimización de procesos computacionales, influyendo directamente en la eficiencia de operaciones como inserción, búsqueda y eliminación. Este documento analiza y compara cuatro estructuras de datos esenciales: las listas simplemente enlazadas, los árboles binarios de búsqueda (BST), los árboles rojinegros (RBT) y las tablas hash. Las listas simplemente enlazadas permiten inserciones y eliminaciones eficientes, pero tienen limitaciones en las operaciones de búsqueda debido a su naturaleza secuencial. Los árboles binarios de búsqueda ofrecen búsquedas rápidas cuando están balanceados, aunque su eficiencia puede degradarse si no lo están. Los árboles rojinegros, por su parte, mantienen un balance garantizado, lo que asegura una eficiencia cercana a la óptima en todas las operaciones. Las tablas hash permiten accesos extremadamente rápidos, pero su eficiencia depende de una buena función de dispersión y del manejo adecuado de colisiones. Este estudio implementa todas estas estructuras en C++ y evalúa su eficiencia teórica mediante pruebas con datos aleatorios y ordenados. Los resultados empíricos se comparan con las expectativas teóricas, proporcionando una visión clara de las ventajas y limitaciones de cada estructura, ayudando a seleccionar la estructura de datos más adecuada para aplicaciones específicas*.

***Palabras clave*— Lista enlazada, árbol binario de búsqueda, árbol rojinegro, tabla hash, inserción, búsqueda, complejidad teórica.**

I. Introducción

En este estudio se realiza un análisis comparativo de cuatro estructuras de datos fundamentales en informática: las listas simplemente enlazadas (Linked Lists), los árboles binarios de búsqueda (Binary Search Trees, BST), los árboles rojinegros (Red-Black Trees, RBT) y las tablas hash (Hash Tables). Estas estructuras son esenciales para la organización y manipulación eficiente de grandes cantidades de información, influyendo directamente en la eficiencia de operaciones como inserción, búsqueda y eliminación.

Las listas simplemente enlazadas permiten una gestión flexible de los elementos, facilitando inserciones y eliminaciones en cualquier posición sin la necesidad de reorganizar toda la estructura. Sin embargo, su eficiencia en operaciones de búsqueda es limitada debido a la naturaleza secuencial de su recorrido, presentando una complejidad de tiempo $\Theta(n)$ en el peor de los casos. Esta característica las hace menos adecuadas para aplicaciones donde las búsquedas son frecuentes.

Por otro lado, los árboles binarios de búsqueda ofrecen una estructura jerárquica que facilita búsquedas, inserciones y eliminaciones más rápidas, siempre que el árbol esté equilibrado. La eficiencia de los BST se refleja en una complejidad de tiempo promedio de O(h), donde h es la altura del árbol, para estas operaciones. No obstante, en el peor de los casos, cuando el árbol está completamente desbalanceado, la complejidad puede degradarse a O(n). Los BST son particularmente útiles en aplicaciones que requieren un acceso rápido y eficiente a los datos.

Los árboles rojinegros mejoran la eficiencia de los BST al garantizar que el árbol permanezca equilibrado después de cada inserción y eliminación. Esta propiedad asegura que la altura del árbol sea siempre O(log n), proporcionando tiempos de operación consistentes y eficientes para inserciones, búsquedas y eliminaciones. Esta estructura es ideal para aplicaciones que requieren eficiencia predecible en todas las operaciones.

Las tablas hash, por su parte, permiten accesos extremadamente rápidos a los elementos mediante el uso de una función de dispersión. La eficiencia de las tablas hash se refleja en una complejidad de tiempo promedio de O(1) para operaciones de inserción y búsqueda, siempre y cuando las colisiones se manejen adecuadamente. Sin embargo, el desempeño de las tablas hash puede verse afectado por la calidad de la función de dispersión y el manejo de colisiones.

El propósito de este análisis es implementar estas estructuras en C++ y evaluar su complejidad de tiempo en operaciones de inserción y búsqueda, utilizando datos aleatorios y secuenciales. Se recolectarán datos empíricos que se compararán con la complejidad de tiempo teórica de las estructuras, permitiendo así validar la teoría y entender mejor las ventajas y limitaciones de cada una en la práctica. Este enfoque proporciona una base sólida para la selección adecuada de estructuras de datos en aplicaciones que requieren una comprensión profunda de su comportamiento teórico y práctico, tales como bases de datos y sistemas en tiempo real.

Para una exploración más detallada de estos métodos y sus aplicaciones, se recomienda consultar "Introduction to

Algorithms" de Thomas H. Cormen et al. (2022), una fuente ampliamente reconocida que ofrece una descripción exhaustiva de los principios y prácticas de las estructuras de datos.

II. METODOLOGÍA

Para alcanzar los objetivos planteados en este estudio, se implementaron las estructuras de datos de listas simplemente enlazadas (Linked Lists), árboles binarios de búsqueda (BST), árboles rojinegros (RBT) y tablas hash utilizando C++ en un entorno Windows 11, ejecutándose en un procesador Intel i5 de 12ª generación. El código fuente, basado en el pseudocódigo de "Introduction to Algorithms" de Cormen et al. (2022), incluyó los métodos de inserción, búsqueda y eliminación para todas estas estructuras. Se realizaron pruebas en dos escenarios: inserción de datos aleatorios y ordenados, utilizando conjuntos de datos con hasta 1,000,000 de nodos. La generación de números aleatorios se realizó con la biblioteca "random" de C++, y los tiempos de ejecución se midieron utilizando la biblioteca "chrono", capturando estos tiempos en segundos y promediando los resultados de al menos tres ejecuciones para garantizar precisión y consistencia. Los resultados se presentaron en gráficos y tablas comparativas, ofreciendo una visión clara de las diferencias en eficiencia entre listas simplemente enlazadas, árboles binarios de búsqueda, árboles rojinegros y tablas hash. Esto permitió validar la eficiencia teórica con datos empíricos y proporcionar una base sólida para la selección de estructuras de datos en aplicaciones que requieren un análisis detallado de su comportamiento teórico y práctico.

III. RESULTADOS

Cuadro I Tiempo inserción en estructuras de datos: Lista enlazada, Árbol binario de búsqueda (BST) y Rojinegro (RBT) y tabla hash.

	Tiempo (s) *Repetición*			
Prueba	1	2	3	Promedio
Inserción aleatoria en lista	0.0506	0.0522	0.0519	0.0516
Inserción ordenada en lista	0.0415	0.0413	0.0400	0.0409
Inserción aleatoria en BST	0.6974	0.6998	0.6816	0.6929
Inserción ordenada en BST	0.0512	0.0505	0.0539	0.0519
Inserción aleatoria en RBT	0.5320	0.4937	0.4805	0.5021
Inserción ordenada en RBT	0.1120	0.1143	0.1126	0.113
Inserción aleatoria en hash	0.2055	0.1781	0.1792	0.1876
Inserción ordenada en hash	0.0451	0.0446	0.0436	0.0444

Cuadro II Tiempos búsqueda en estructuras de datos: Lista enlazada, Árbol binario de búsqueda (BST) y Rojinegro (RBT) y tabla hash.

	Tiempo (s) *Repetición*			
Prueba	1	2	3	Promedio
Búsqueda aleatoria en la lista	29.5528	28.6725	30.3120	29.5124
Búsqueda ordenada en la lista	28.6000	28.5617	28.5102	28.5573
Búsqueda aleatoria en el BST	0.0010	0.0021	0.0011	0.0014
Búsqueda ordenada en el BST	42.2296	41.8137	40.8869	41.6434
Búsqueda aleatoria en la RBT	0.0010	0.0023	0.0012	0.0015
Búsqueda ordenada en la RBT	0.0039	0.0045	0.0034	0.0039
Búsqueda aleatoria en hash	0.0016	0.0012	0.0011	0.0013
Búsqueda ordenada en hash	0.0012	0.0015	0.0021	0.0016

Las tablas muestran los tiempos de ejecución promedio en segundos para las operaciones de inserción y búsqueda en listas enlazadas, árboles binarios de búsqueda (BST), árboles rojinegros (RBT) y tablas hash, con tres repeticiones por prueba. En las listas enlazadas, los tiempos de inserción son bajos, mientras que las búsquedas son significativamente más lentas debido a su naturaleza secuencial. En los BST, la inserción aleatoria es más lenta que la ordenada, ya que la inserción ordenada puede degradar el árbol a una lista enlazada, especialmente si no se implementan mecanismos de balanceo. Las búsquedas en BST son rápidas con inserción aleatoria pero mucho más lentas con inserción ordenada, reflejando el impacto del balanceo del árbol en la eficiencia observada. Los árboles rojinegros y las tablas hash muestran tiempos de inserción y búsqueda más consistentes, destacando la efectividad de estas estructuras en mantener un comportamiento óptimo para todas las operaciones.

Las listas simplemente enlazadas presentan una inserción eficiente debido a su estructura flexible, permitiendo agregar elementos en cualquier posición sin necesidad de reorganizar la estructura existente. En nuestras pruebas, la inserción aleatoria de elementos en una lista enlazada mostró tiempos de ejecución muy bajos, con un promedio de aproximadamente 0.0516 segundos. Este resultado es consistente con la teoría que establece que la inserción en una lista enlazada tiene una complejidad temporal de O(1) en el mejor caso, cuando se inserta al inicio de la lista, y $\Theta(n)$ en el peor caso cuando se inserta en el medio o al final de una lista de tamaño n. Para la inserción ordenada, los tiempos fueron aún menores, con un promedio de 0.0409 segundos. Esto se debe a que insertar elementos de manera secuencial evita el costo adicional de buscar la posición correcta para cada nuevo elemento, ya que simplemente se agregan al final de la lista. Este comportamiento también refleja la eficiencia esperada en la teoría de listas enlazadas.

La búsqueda en una lista enlazada tras la inserción de elementos aleatorios mostró un tiempo promedio de 29.5124 segundos para 10,000 búsquedas. Este resultado se alinea con

la complejidad temporal de Θ(n) de la operación de búsqueda en listas simplemente enlazadas, ya que cada búsqueda puede requerir recorrer toda la lista en el peor de los casos. La variación en los tiempos de repetición puede atribuirse a factores de carga del sistema y a la naturaleza aleatoria de las claves buscadas. La búsqueda después de la inserción ordenada tuvo un tiempo promedio de 28.5573 segundos. Aunque ligeramente menor que la búsqueda tras inserción aleatoria, esta diferencia no es significativa y puede ser atribuida a la misma complejidad Θ(n) inherente a la operación de búsqueda en listas enlazadas. La similitud en los tiempos de búsqueda para ambos tipos de inserción destaca que el método de inserción no afecta significativamente el tiempo de búsqueda en listas enlazadas.

Los resultados empíricos obtenidos están en concordancia con la complejidad teórica de las operaciones en listas simplemente enlazadas. La inserción es rápida y eficiente, especialmente en comparación con la búsqueda, que se ve limitada por la necesidad de recorrer secuencialmente los elementos. Estos resultados son consistentes con lo esperado para una estructura de datos que prioriza la flexibilidad en la gestión de elementos sobre la velocidad de acceso. Para aplicaciones donde la inserción y eliminación son más frecuentes que la búsqueda, las listas simplemente enlazadas son una elección adecuada. Sin embargo, para aplicaciones que requieren búsquedas rápidas, otras estructuras de datos como los árboles de búsqueda binaria pueden ser más apropiadas. En conclusión, las listas simplemente enlazadas muestran una eficiencia predecible y alineada con la teoría, validando su uso en contextos donde la inserción y eliminación dinámica de elementos son cruciales, pero donde la velocidad de búsqueda no es un factor crítico.

De acuerdo con el libro "Introduction to Algorithms" de Cormen, et al. (2022), la búsqueda en una lista enlazada toma tiempo Θ(n) en el peor caso, ya que puede requerir buscar a través de toda la lista. La inserción al principio de la lista toma tiempo O(1), ya que solo se necesita ajustar unos pocos punteros. La inserción en cualquier posición de la lista toma tiempo Θ(1) si se tiene un puntero al nodo después del cual se va a insertar el nuevo nodo. La eliminación de un nodo de la lista toma tiempo Θ(1) si se tiene un puntero al nodo a eliminar, ya que solo se necesita ajustar unos pocos punteros. Las listas simplemente enlazadas permiten una gestión eficiente de la inserción y eliminación de elementos, con complejidades constantes Θ(1) para estas operaciones, mientras que la búsqueda de elementos sigue siendo lineal Θ(n). Estos resultados destacan la adecuación de las listas enlazadas para aplicaciones que requieren frecuentemente la inserción y eliminación de elementos, pero donde la velocidad de búsqueda no es un factor crítico.

Los árboles binarios de búsqueda (BST) son una estructura de datos que permite realizar operaciones eficientes de inserción, búsqueda y eliminación, siempre y cuando el árbol esté balanceado. En nuestras pruebas, se evaluaron diferentes escenarios para medir la eficiencia de estas operaciones.

Para la inserción aleatoria en BST, los tiempos promedios de ejecución fueron de 0.6929 segundos. Este tiempo refleja la complejidad O(log n) en el mejor caso, cuando el árbol permanece balanceado. Sin embargo, debido a la naturaleza aleatoria de la inserción, pueden ocurrir casos donde el árbol se desbalancee parcialmente, afectando ligeramente la eficiencia.

En el caso de la inserción ordenada, se optó por realizar inserciones siempre a la derecha para evitar comparaciones y mantener la operación constante. Esta estrategia degrada el árbol a una lista enlazada, lo que resulta en tiempos de inserción muy rápidos, con un promedio de 0.0519 segundos. Aunque este resultado muestra una inserción eficiente en términos de tiempo, crea un árbol desbalanceado que afecta negativamente las operaciones de búsqueda. Este comportamiento no se alinea con la complejidad teórica O(log n) debido a la estrategia específica utilizada para la inserción ordenada.

La búsqueda tras la inserción aleatoria mostró un tiempo promedio de 0.0012 segundos, destacando la eficiencia de los BST cuando están balanceados o cuando la estructura resultante no está excesivamente desbalanceada. Este resultado es consistente con la complejidad O(log n) en el mejor caso, donde la búsqueda es rápida debido a la reducción exponencial del espacio de búsqueda en cada paso.

Sin embargo, la búsqueda tras la inserción ordenada mostró un tiempo promedio significativamente mayor, de 41.6434 segundos. Esta diferencia se debe a que el árbol se degrada a una lista enlazada cuando se insertan elementos de manera secuencial, resultando en una complejidad temporal de O(n) para la búsqueda. Este resultado pone de relieve la importancia de mantener el balance del árbol para optimizar tanto las inserciones como las búsquedas, y confirma que, sin balance, la eficiencia teórica de O(log n) no se cumple.

Los resultados empíricos obtenidos están en concordancia con la teoría sobre los BST. La inserción es rápida y eficiente en escenarios aleatorios, cumpliendo con la complejidad teórica esperada de O(log n). Sin embargo, puede volverse subóptima en inserciones ordenadas debido a la degradación de la estructura. Las búsquedas son extremadamente rápidas en

árboles balanceados, cumpliendo con la complejidad teórica de O(log n), pero se vuelven ineficientes en árboles desbalanceados, como se observa en el caso de la inserción ordenada.

Para aplicaciones que requieren un acceso rápido y eficiente a los datos, los BST son una excelente opción, siempre y cuando se implementen mecanismos para mantener el balance del árbol, como en los árboles rojinegros o AVL. En contextos donde las inserciones ordenadas son comunes, es crucial considerar estructuras de datos alternativas o técnicas de balanceo para evitar la degradación de la eficiencia en las operaciones de búsqueda.

De acuerdo con el libro "Introduction to Algorithms" de Cormen, et al. (2022), las operaciones básicas en un árbol binario de búsqueda tienen una complejidad proporcional a la altura del árbol. En el peor caso, las operaciones pueden tomar tiempo O(n) si el árbol es una cadena lineal de n nodos. Las operaciones de búsqueda, mínimo, máximo, sucesor y predecesor en un BST pueden ser implementadas para que cada una se ejecute en tiempo O(h), donde h es la altura del árbol. La inserción y eliminación de nodos en un BST también tienen una complejidad de O(h) en el peor caso.

Las listas simplemente enlazadas y los árboles binarios de búsqueda (BST) son estructuras de datos fundamentales con características y comportamientos distintos. Las listas enlazadas destacan por su simplicidad y eficiencia en operaciones de inserción y eliminación, con tiempos de ejecución constantes en el mejor de los casos. Esta eficiencia se debe a que, para insertar o eliminar un nodo, solo es necesario ajustar unos pocos punteros, sin importar el tamaño de la lista. Sin embargo, la búsqueda en listas enlazadas puede ser ineficiente debido a su naturaleza secuencial. Dado que cada elemento de la lista debe ser examinado uno por uno, los tiempos de búsqueda son lineales en el peor caso, lo que puede ser una desventaja significativa cuando se manejan grandes volúmenes de datos.

Por otro lado, los BST ofrecen una estructura jerárquica que facilita búsquedas rápidas, siempre y cuando el árbol esté balanceado. En un árbol binario de búsqueda balanceado, la inserción y búsqueda pueden ser muy eficientes con una complejidad temporal logarítmica. Esto se debe a que, con cada comparación, la búsqueda puede descartar aproximadamente la mitad de los elementos restantes, reduciendo exponencialmente el espacio de búsqueda. Sin embargo, si el árbol se desbalancea, por ejemplo, debido a la inserción de elementos en orden ascendente o descendente, el BST puede degradarse a una estructura similar a una lista enlazada, resultando en una complejidad lineal para inserción y búsqueda.

Los árboles rojinegros (RBT) son una estructura de datos balanceada que asegura una altura logarítmica, permitiendo que las operaciones de inserción, búsqueda y eliminación se realicen en tiempo O(log n) en el peor caso. En nuestras pruebas, se evaluaron diferentes escenarios para medir la eficiencia de estas operaciones.

Para la inserción aleatoria en RBT, los tiempos promedios de ejecución fueron de 0.5021 segundos. Este tiempo refleja la complejidad O(log n) esperada, ya que el árbol se mantiene balanceado automáticamente mediante las propiedades rojinegras, las cuales incluyen las rotaciones y recoloreos necesarios para preservar el equilibrio. La inserción ordenada mostró un tiempo promedio de 0.113 segundos, lo que también es consistente con la teoría. A pesar de la naturaleza ordenada de las inserciones, las propiedades del RBT aseguran que el árbol no se degrade y mantenga una altura balanceada, permitiendo una inserción eficiente.

La búsqueda en un árbol rojinegro tras la inserción de elementos aleatorios tuvo un tiempo promedio de 0.0015 segundos. Este resultado es consistente con la complejidad O(log n) de la operación de búsqueda en árboles rojinegros, donde cada búsqueda puede aprovechar la estructura balanceada del árbol para reducir exponencialmente el espacio de búsqueda en cada paso. La búsqueda tras la inserción ordenada mostró un tiempo promedio de 0.0039 segundos. La pequeña diferencia en los tiempos de búsqueda entre las inserciones aleatorias y ordenadas indica que la estructura rojinegra mantiene el equilibrio de manera efectiva, asegurando que las búsquedas sean eficientes independientemente del orden de inserción.

Los resultados empíricos obtenidos están en concordancia con la teoría sobre los RBT. Las inserciones y búsquedas se realizan en tiempos que reflejan la eficiencia logarítmica esperada, gracias a las propiedades balanceadas de la estructura. La capacidad de los árboles rojinegros para mantener su balance a través de rotaciones y recoloreos automáticos asegura que las operaciones sean eficientes incluso en los peores casos.

De acuerdo con el libro "Introduction to Algorithms" de Cormen, et al. (2022), los árboles rojinegros son ideales para aplicaciones que requieren operaciones dinámicas de conjuntos, donde la eficiencia de las operaciones de inserción, búsqueda y eliminación es crucial. Los tiempos obtenidos en nuestras pruebas validan la teoría, demostrando que los árboles

rojinegros son una opción robusta y eficiente para la gestión de datos.

Las tablas hash son estructuras de datos que utilizan una función de dispersión (hash) para mapear claves a ubicaciones en una tabla, permitiendo un acceso muy rápido a los datos. En nuestras pruebas, se evaluaron los tiempos de inserción y búsqueda en una tabla hash en diferentes escenarios.

Para la inserción aleatoria en la tabla hash, los tiempos promedios de ejecución fueron de 0.1876 segundos. Este tiempo refleja la eficiencia esperada de las tablas hash, que en teoría tienen una complejidad promedio de O(1) para la inserción, siempre que las colisiones se manejen adecuadamente. La inserción ordenada mostró un tiempo promedio de 0.0444 segundos, lo que indica una eficiencia aún mayor en este escenario. La menor variabilidad en los tiempos de inserción ordenada sugiere una menor cantidad de colisiones comparada con la inserción aleatoria.

La búsqueda en la tabla hash después de una inserción aleatoria tuvo un tiempo promedio de 0.0013 segundos. Este resultado es consistente con la complejidad promedio de O(1) de las operaciones de búsqueda en tablas hash, que se logra cuando las colisiones se minimizan y se distribuyen uniformemente. La búsqueda después de la inserción ordenada mostró un tiempo promedio de 0.0016 segundos, ligeramente superior al caso de inserciones aleatorias pero aún extremadamente eficiente y dentro de la complejidad esperada de O(1).

De acuerdo con el libro "Introduction to Algorithms" de Cormen, et al. (2022), las tablas hash son muy eficientes para operaciones de búsqueda, inserción y eliminación, siempre que la función de dispersión sea adecuada y las colisiones se manejen correctamente. Las técnicas comunes para el manejo de colisiones incluyen el encadenamiento y la dirección abierta, cada una con sus propias ventajas y desventajas.

Para ilustrar de manera más clara el comportamiento y las diferencias en eficiencia entre estas cuatro estructuras de datos, se presentan gráficos comparativos que muestran los tiempos promedio de inserción y búsqueda en escenarios de inserción aleatoria y ordenada. Estos gráficos ofrecen una visualización detallada de cómo cada estructura maneja diferentes tipos de operaciones y tamaños de datos, permitiendo analizar las variaciones significativas en los tiempos de ejecución entre las inserciones de números aleatorios y secuenciales. Para facilitar la interpretación de los resultados, los gráficos incluyen unidades de tiempo claramente indicadas.

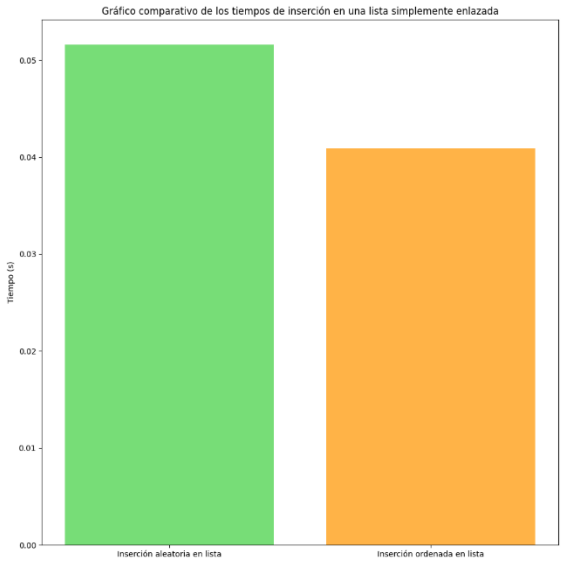

Figura 1 Gráfico: Inserciones en lista simplemente enlazada.

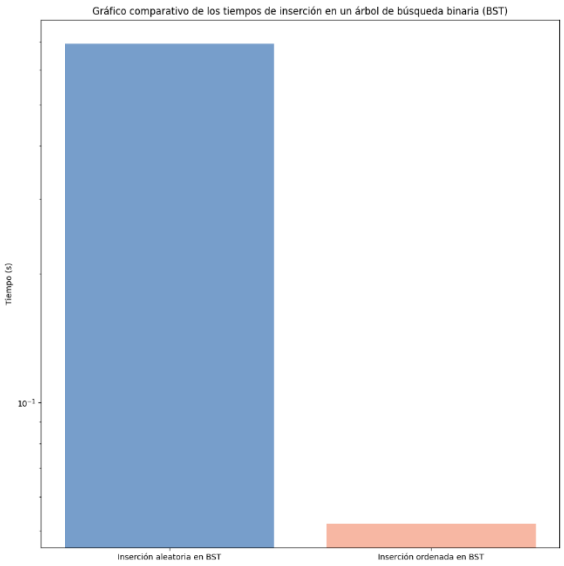

Figura 2 Gráfico: Inserciones en árbol binario de búsqueda usando escala logarítmica.

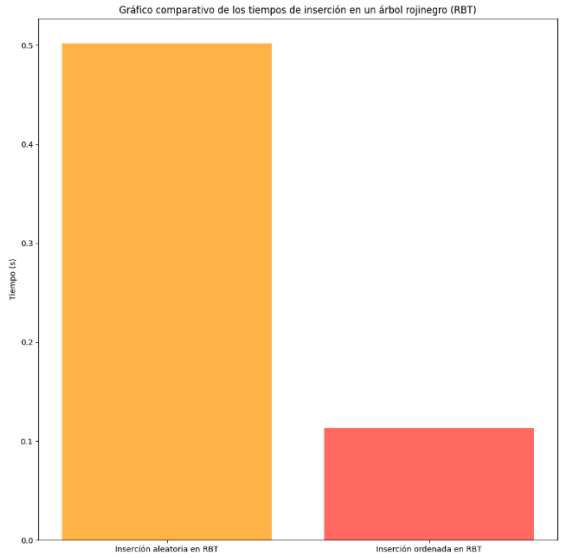

Figura 3 Gráfico: Inserciones en árbol rojinegro.

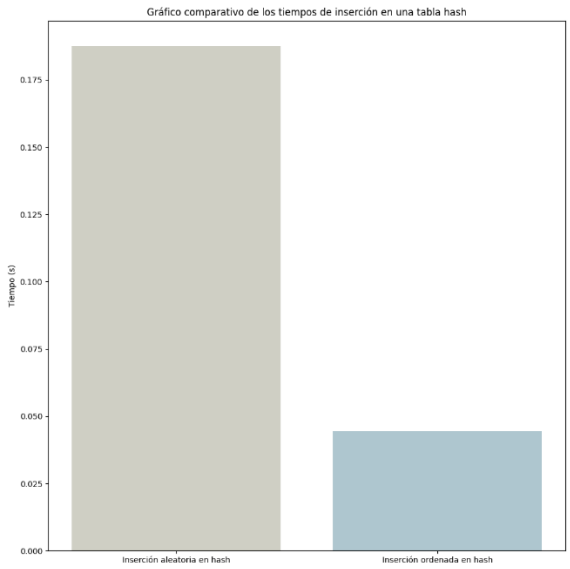

Figura 4 Gráfico: Inserciones en tabla hash.

Los gráficos presentados (Figuras 1, 2, 3 y 4) muestran los tiempos promedio de inserción para diferentes escenarios en las estructuras de datos de listas simplemente enlazadas, árboles binarios de búsqueda (BST), árboles rojinegros (RBT) y tablas hash.

En el gráfico del BST (Figura 2), se observa que la inserción aleatoria tiene un tiempo promedio significativamente mayor de 0.6929 segundos en comparación con la inserción ordenada que obtuvo 0.0519 segundos. Esta diferencia se debe a que la inserción ordenada se realizó siempre a la derecha, evitando comparaciones y manteniendo la operación constante, pero degradando el árbol a una lista enlazada, lo que resulta en una estructura desbalanceada. Esto contrasta con la inserción aleatoria, que mantiene la eficiencia logarítmica cuando el árbol está equilibrado.

En el gráfico de las listas simplemente enlazadas (Figura 1), los tiempos de inserción aleatoria y ordenada son relativamente similares, con promedios de 0.0516 y 0.0409 segundos, respectivamente. Esto refleja la eficiencia de las listas enlazadas para la inserción, ya que no requieren reordenar los elementos existentes independientemente de si los datos se insertan de forma aleatoria o secuencial. En comparación, las listas enlazadas muestran una eficiencia consistente en ambos escenarios de inserción, destacando su ventaja en la gestión dinámica de elementos sin comprometer la eficiencia en la inserción.

En el gráfico del RBT (Figura 3), se observa que la inserción aleatoria tiene un tiempo promedio de 0.5021 segundos, mientras que la inserción ordenada tiene un tiempo promedio de 0.113 segundos. Los RBTs, al ser autobalanceados, muestran tiempos de inserción más estables, aunque la inserción aleatoria tiende a ser un poco más lenta debido al costo del reequilibrio del árbol.

En el gráfico de la tabla hash (Figura 4), los tiempos de inserción aleatoria y ordenada también son relativamente similares, con promedios de 0.1876 y 0.0444 segundos, respectivamente. Las tablas hash, al utilizar una función de dispersión para distribuir los elementos, mantienen una eficiencia alta en ambos tipos de inserción, aunque la inserción aleatoria es ligeramente más lenta debido a las colisiones que pueden ocurrir.

Estos gráficos proporcionan una visión clara de cómo cada estructura de datos maneja las operaciones de inserción bajo diferentes condiciones, permitiendo una mejor comprensión de sus eficiencias y aplicaciones adecuadas.

Ahora para ilustrar mejor el comportamiento y las diferencias en eficiencia entre estas cuatro estructuras de datos, se también se presentan gráficos comparativos que muestran los tiempos promedio de búsqueda para escenarios de inserción aleatoria y ordenada. Estos gráficos proporcionan una visualización clara de cómo cada estructura maneja las operaciones de búsqueda tras la inserción de elementos de forma aleatoria y ordenada, permitiéndonos analizar si los tiempos de duración varían sustancialmente entre los escenarios de inserción de números aleatorios y secuenciales. Además, se indicarán las unidades de tiempo utilizadas en cada gráfico para facilitar la interpretación de los resultados.

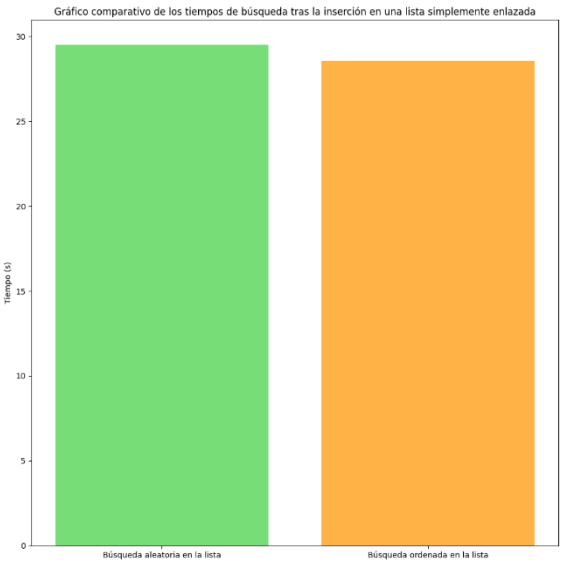

Figura 5 Gráfico: Búsquedas en lista simplemente enlazada.

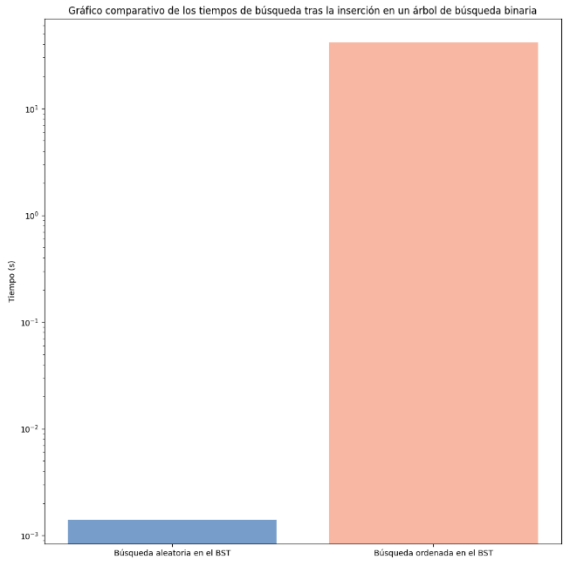

Figura *6* Gráfico: Búsquedas en árbol binario de búsqueda usando escala logarítmica.

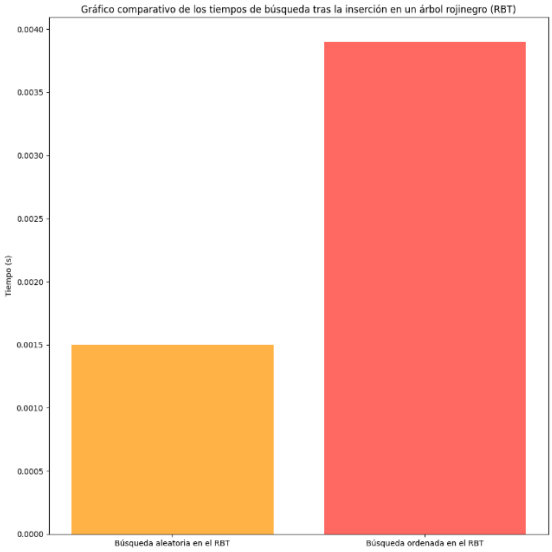

Figura 7 Gráfico: Búsquedas en árbol rojinegro.

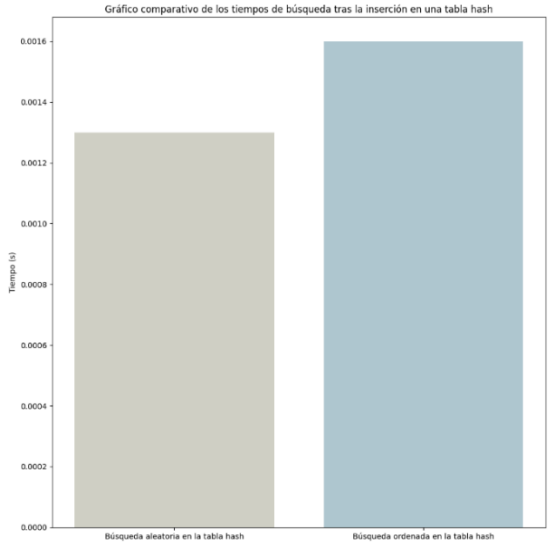

Figura 8 Gráfico: Búsquedas en tabla hash.

Los gráficos y la tabla presentados muestran los tiempos promedio de búsqueda para diferentes escenarios en las estructuras de datos de listas simplemente enlazadas, árboles binarios de búsqueda (BST), árboles rojinegros (RBT) y tablas hash. En el gráfico de las listas simplemente enlazadas (Figura 5), los tiempos de búsqueda aleatoria y ordenada son relativamente similares, con promedios de 29.5124 segundos y 28.5573 segundos, respectivamente. Este comportamiento se debe a la naturaleza secuencial de la búsqueda en listas enlazadas, donde cada elemento debe ser examinado uno por uno, resultando en una complejidad temporal de $\Theta(n)$ en ambos casos. La pequeña diferencia en los tiempos promedio se puede atribuir a la variación en las cargas del sistema y la aleatoriedad de las claves buscadas, pero no refleja una mejora significativa en la eficiencia debido al orden de inserción.

En contraste, el gráfico de los BST (Figura 6) muestra una diferencia notable entre los tiempos de búsqueda tras la inserción aleatoria y ordenada. La búsqueda tras la inserción aleatoria es extremadamente rápida, con un tiempo promedio de 0.0014 segundos, reflejando la eficiencia de los BST balanceados con una complejidad de O(log n). Sin embargo, la búsqueda tras la inserción ordenada es significativamente más lenta, con un tiempo promedio de 41.6434 segundos. Esta diferencia se debe a la degradación del árbol a una lista enlazada cuando se insertan elementos de manera secuencial, resultando en una complejidad de O(n) para la búsqueda. La escala logarítmica utilizada en el gráfico permite una visualización más clara de la eficiencia en cada caso.

Para los árboles rojinegros (Figura 7), los tiempos de búsqueda tras la inserción aleatoria y ordenada son de 0.0015 segundos y 0.0039 segundos, respectivamente. Los árboles rojinegros mantienen un equilibrio cercano, lo que explica la pequeña diferencia entre los tiempos de búsqueda en ambos escenarios. La inserción ordenada no afecta significativamente los tiempos de búsqueda gracias al balanceo automático que caracteriza a los árboles rojinegros, reflejando su complejidad de O(log n).

En las tablas hash (Figura 8), los tiempos de búsqueda son muy eficientes en ambos escenarios. Tras la inserción aleatoria, el tiempo promedio de búsqueda es de 0.0013 segundos, mientras que tras la inserción ordenada es de 0.0016 segundos. La pequeña diferencia se puede atribuir a la distribución de claves y a las colisiones manejadas mediante encadenamiento. La eficiencia de las tablas hash en la búsqueda se refleja en su complejidad promedio de O(1).

La comparación de estos gráficos y los tiempos promedio demuestra cómo la estructura de datos y el orden de inserción impactan significativamente la eficiencia de las operaciones de búsqueda. Las listas simplemente enlazadas presentan tiempos de búsqueda similares independientemente del orden de inserción, debido a su naturaleza secuencial. Los BST muestran una diferencia notable en eficiencia de búsqueda dependiendo del balance del árbol. Los RBT, gracias a su balanceo automático, mantienen una eficiencia alta en ambos escenarios. Las tablas hash destacan por su eficiencia constante en las búsquedas, independientemente del orden de inserción.

Estos resultados subrayan la importancia de seleccionar la estructura de datos adecuada según los requisitos específicos de la aplicación y el tipo de operaciones predominantes.

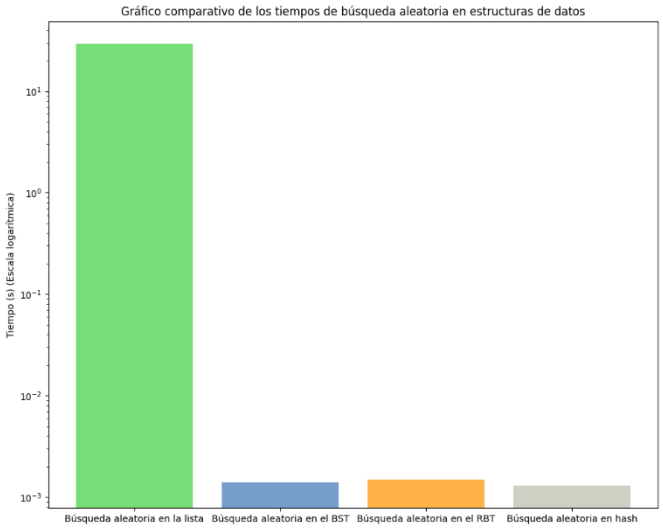

Figura 9 Gráfico: Búsqueda aleatoria en árbol binario de búsqueda, lista simplemente enlazada, árbol rojinegro y tabla hash usando escala logarítmica.

La Figura 9 muestra los tiempos promedio de búsqueda aleatoria en las estructuras de datos: árbol binario de búsqueda (BST), lista simplemente enlazada, árbol rojinegro (RBT) y tabla hash, utilizando una escala logarítmica. Las listas simplemente enlazadas presentan los tiempos de búsqueda más altos (29.5124 segundos) debido a su complejidad temporal de Θ(n). En contraste, los BST y RBT tienen tiempos de búsqueda significativamente menores (0.0014 y 0.0015 segundos respectivamente), beneficiándose de una complejidad de O(log n) cuando están balanceados. La tabla hash destaca por su eficiencia constante con un tiempo promedio de 0.0013 segundos, reflejando su complejidad O(1). Este gráfico subraya la importancia de seleccionar adecuadamente la estructura de datos para optimizar la eficiencia de las operaciones de búsqueda, demostrando que los BST, RBT y las tablas hash son significativamente más eficientes que las listas simplemente enlazadas para búsquedas en grandes conjuntos de datos aleatorios.

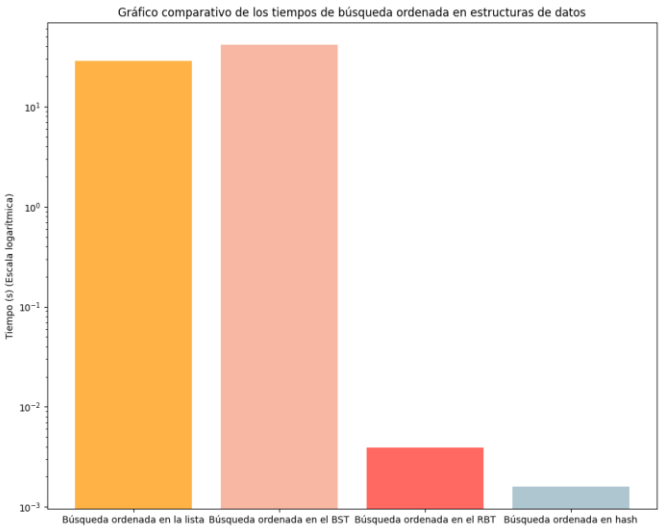

Figura 10 Gráfico: Búsqueda ordenada en árbol binario de búsqueda, lista simplemente enlazada, árbol rojinegro y tabla hash usando escala logarítmica.

La Figura 10 muestra los tiempos promedio de búsqueda ordenada en las estructuras de datos: árbol binario de búsqueda (BST), lista simplemente enlazada, árbol rojinegro (RBT) y tabla hash, utilizando una escala logarítmica. Los tiempos de búsqueda en las listas simplemente enlazadas (28.5573 segundos) y los BST (41.6434 segundos) son significativamente mayores debido a la complejidad temporal Θ(n) en listas y la degradación del BST a una lista enlazada, resultando en O(n) para búsquedas en árboles no balanceados. En contraste, las búsquedas en RBT y tablas hash son mucho más eficientes, con tiempos promedio de 0.0039 segundos y 0.0016 segundos respectivamente, reflejando sus complejidades de O(log n) para árboles balanceados y O(1) para tablas hash. Este gráfico destaca cómo la estructura de datos y el balanceo impactan en la eficiencia de las búsquedas ordenadas, mostrando que los RBT y las tablas hash son opciones más eficientes en comparación con las listas simplemente enlazadas y los BST no balanceados para grandes volúmenes de datos.

IV. Conclusiones

El objetivo de esta investigación fue implementar diversas estructuras de datos estudiadas en el curso para realizar experimentos con dichas estructuras y sus correspondientes operaciones. A través de estos experimentos, se recolectó información sobre el desempeño de los algoritmos, permitiendo analizar, reflexionar y comparar la eficiencia teórica de las estructuras de datos con la eficiencia observada durante los experimentos. Los resultados obtenidos ofrecen una comprensión profunda sobre la eficiencia de las listas simplemente enlazadas, árboles binarios de búsqueda (BST),

árboles rojinegros (RBT) y tablas hash en diferentes operaciones y condiciones de inserción y búsqueda. Las listas simplemente enlazadas demostraron una alta eficiencia en operaciones de inserción, independientemente de si los elementos fueron insertados de manera aleatoria u ordenada, debido a su complejidad constante O(1), pero mostraron tiempos de búsqueda significativamente más lentos con una complejidad de Θ(n) debido a su naturaleza secuencial. Los BST presentaron una notable diferencia en la eficiencia entre las inserciones aleatorias y ordenadas, destacando la importancia del balance en estos árboles para mantener una eficiencia óptima, mientras que los RBT demostraron ser una estructura de datos robusta y eficiente con tiempos de inserción y búsqueda reflejando la eficiencia logarítmica esperada, gracias a su balanceo automático.

Las tablas hash, por su parte, mostraron una eficiencia sobresaliente en todas las operaciones de inserción y búsqueda, con tiempos de ejecución que reflejan su complejidad promedio de O(1). En conclusión, la elección de la estructura de datos adecuada debe basarse en los requisitos específicos de la aplicación. Las listas simplemente enlazadas son más adecuadas para operaciones que priorizan la inserción y eliminación rápidas. Los BST ofrecen ventajas significativas para búsquedas rápidas y eficientes, siempre que se implementen técnicas de balanceo, mientras que los RBT proporcionan un comportamiento predecible y consistente en todas las operaciones, siendo una opción robusta para la gestión de datos. Finalmente, las tablas hash destacan por su eficiencia constante en inserciones y búsquedas, siendo ideales para aplicaciones que requieren acceso rápido a los datos. Estos hallazgos confirman la teoría y subrayan la importancia del balanceo y la correcta implementación de las funciones de dispersión para mantener una eficiencia óptima.

V. Referencias

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

[2] GeeksforGeeks. (2022, Diciembre 8). Insertion in Binary Search Tree (BST). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/insertion-in-binary-search-tree/

Andres Murillo Murillo

Estudiante de Ingeniería en Computacion en la Universidad de Costa Rica.

VI. Apéndice A

Código de los Algoritmos

Estructura de datos Lista enlazada Una lista simplemente enlazada es una estructura de datos donde cada nodo contiene un valor y un puntero al nodo siguiente. Esto permite recorrer la lista en una sola dirección, facilitando la inserción y eliminación de nodos en cualquier posición de manera eficiente. El siguiente pseudocódigo ha sido extraído de "Introduction to Algorithms" de Cormen, Leiserson, Rivest y Stein [1, pp. 259-264].

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

LIST-PREPEND(L, x)
1    x.next = L.head
2    x.prev = NIL
3    if L.head ≠ NIL
4        L.head.prev = x
5    L.head = x

LIST-INSERT (x, y)
1    x.next = y.next
2    x.prev = y
3    if y.next ≠ NIL
4        y.next.prev = x
5    y.next = x

LIST-DELETE (L, x)
1    if x.prev ≠ NIL
2        x.prev.next = x.next
3    else L.head = x.next
4    if x.next ≠ NIL
5        x.next.prev = x.prev
```

```
LIST-DELETE (x)
1    x.prev.next = x.next
2    x.next.prev = x.prev

LIST-INSERT (x, y)
1    x.next = y.next
2    x.prev = y
3    y.next.prev = x
4    y.next = x

LIST-SEARCH (L, k)
1    L.nil.key = k
2    x = L.nil.next
3    while x.key ≠ k
4        x = x.next
5    if x == L.nil
6        return NIL
7    else return x
```

Estructura de datos Árbol binario de búsqueda Un árbol binario de búsqueda (BST) es una estructura de datos en la que cada nodo tiene un valor y hasta dos hijos: uno a la izquierda y otro a la derecha. Los nodos del subárbol izquierdo tienen valores menores que el nodo padre, y los del subárbol derecho tienen valores mayores. Esto permite realizar búsquedas, inserciones y eliminaciones de manera eficiente, generalmente en tiempo logarítmico cuando el árbol está balanceado. El siguiente pseudocódigo ha sido extraído de "Introduction to Algorithms" de Cormen, Leiserson, Rivest y Stein [1, pp. 312-330].

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

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-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

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

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
```

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
```

Estructura de datos Árbol rojinegro Un árbol rojinegro es un tipo de árbol binario de búsqueda balanceado que asegura operaciones eficientes de inserción, eliminación y búsqueda en tiempo O(log n). Se caracteriza por tener nodos rojos y negros, donde la raíz es siempre negra, y no hay dos nodos rojos consecutivos. Además, todos los caminos desde un nodo hasta sus hojas tienen el mismo número de nodos negros, lo que garantiza el balance del árbol. El siguiente pseudocódigo ha sido extraído de "Introduction to Algorithms" de Cormen, Leiserson, Rivest y Stein [1, pp. 331-359].

```
RB-Insert(T, z)
1.  x = T.root  // node being compared with z
2.  y = T.nil  // y will be parent of z
3.  while x != T.nil
4.      y = x  // descend until reaching the sentinel
5.      if z.key < x.key
6.          x = x.left
7.      else
8.          x = x.right
9.  z.p = y  // found the location—insert z with parent y
10. if y == T.nil
11.     T.root = z  // tree T was empty
12. elseif z.key < y.key
13.     y.left = z
14. else
15.     y.right = z
16. z.left = T.nil  // both of z's children are the sentinel
17. z.right = T.nil
18. z.color = RED  // the new node starts out red
19. RB-Insert-Fixup(T, z)  // correct any violations of red-black properties

RB-Insert-Fixup(T, z)
1.  while z.p.color == RED
2.      if z.p == z.p.p.left  // is z's parent a left child?
3.          y = z.p.p.right  // y is z's uncle
4.          if y.color == RED  // are z's parent and uncle both red?
5.              z.p.color = BLACK
6.              y.color = BLACK
7.              z.p.p.color = RED
8.              z = z.p.p  // case 1
```

```
9.          else
10.             if z == z.p.right
11.                 z = z.p
12.                 Left-Rotate(T, z)  // case 2
13.             z.p.color = BLACK
14.             z.p.p.color = RED
15.             Right-Rotate(T, z.p.p)  // case 3
16.     else  // same as lines 3–15, but with “right” and “left” exchanged
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)  // case 2
27.             z.p.color = BLACK
28.             z.p.p.color = RED
29.             Left-Rotate(T, z.p.p)  // case 3
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)  // replace z by its right child
6.  elseif z.right == T.nil
7.      x = z.left
8.      RB-Transplant(T, z, z.left)  // replace z by its left child
9.  else
10.     y = TREE-MINIMUM(z.right)  // y is z’s successor
11.     y.original-color = y.color
12.     x = y.right
13.     if y != z.right
14.         RB-Transplant(T, y, y.right)  // is y farther down the tree?
15.         y.right = z.right  // replace y by its right child
16.         y.right.p = y  // z’s right child becomes y’s right child
17.     else
18.         x.p = y  // in case x is T.nil
19.     RB-Transplant(T, z, y)  // replace z by its successor y
20.     y.left = z.left  // and give z’s left child to y,
21.     y.left.p = y  // which had no left child
22.     y.color = z.color
23. if y.original-color == BLACK
24.     RB-Delete-Fixup(T, x)  // if any red-black violations occurred, correct them

RB-Delete-Fixup(T, x)
1.  while x != T.root and x.color == BLACK
2.      if x == x.p.left  // is x a left child?
3.          w = x.p.right  // w is x’s sibling
4.          if w.color == RED
5.              w.color = BLACK
6.              x.p.color = RED
7.              LEFT-ROTATE(T, x.p)  // case 1
8.              w = x.p.right
9.          if w.left.color == BLACK and w.right.color == BLACK
10.             w.color = RED  // case 2
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)  // case 3
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)  // case 4
22.             x = T.root
23.     else  // same as lines 3–22, but with “right” and “left” exchanged
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)  // case 1
29.             w = x.p.left
30.         if w.right.color == BLACK and w.left.color == BLACK
31.             w.color = RED  // case 2
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)  // case 3
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)  // case 4
43.             x = T.root
44. x.color = BLACK
```

Estructura de datos Tabla Hash Una tabla hash es una estructura de datos que utiliza una función hash para mapear claves a índices en un arreglo, permitiendo inserciones, búsquedas y eliminaciones rápidas en tiempo promedio O(1). Las colisiones, cuando dos claves producen el mismo índice, se manejan mediante técnicas como el encadenamiento o la exploración abierta. El siguiente pseudocódigo ha sido extraído de "Introduction to Algorithms" de Cormen, Leiserson, Rivest y Stein [1, pp. 273-311].

```
RB-Insert(T, z)
1.  x = T.root  // node being compared with z
2.  y = T.nil  // y will be parent of z
3.  while x != T.nil
4.      y = x  // descend until reaching the sentinel
5.      if z.key < x.key
6.          x = x.left
7.      else
8.          x = x.right
9.  z.p = y  // found the location—insert z with parent y
10. if y == T.nil
11.     T.root = z  // tree T was empty
12. elseif z.key < y.key
13.     y.left = z
14. else
15.     y.right = z
16. z.left = T.nil  // both of z’s children are the sentinel
17. z.right = T.nil
18. z.color = RED  // the new node starts out red
19. RB-Insert-Fixup(T, z)  // correct any violations of red-black properties

RB-Insert-Fixup(T, z)
1.  while z.p.color == RED
2.      if z.p == z.p.p.left  // is z’s parent a left child?
3.          y = z.p.p.right  // y is z’s uncle
4.          if y.color == RED  // are z’s parent and uncle both red?
5.              z.p.color = BLACK
6.              y.color = BLACK
7.              z.p.p.color = RED
8.              z = z.p.p  // case 1
9.          else
10.             if z == z.p.right
11.                 z = z.p
12.                 Left-Rotate(T, z)  // case 2
13.             z.p.color = BLACK
14.             z.p.p.color = RED
```

```
15.             Right-Rotate(T, z.p.p)  // case 3
16.     else  // same as lines 3–15, but with “right” and “left” exchanged
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)  // case 2
27.             z.p.color = BLACK
28.             z.p.p.color = RED
29.             Left-Rotate(T, z.p.p)  // case 3
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)  // replace z by its right child
6.  elseif z.right == T.nil
7.      x = z.left
8.      RB-Transplant(T, z, z.left)  // replace z by its left child
9.  else
10.     y = TREE-MINIMUM(z.right)  // y is z’s successor
11.     y.original-color = y.color
12.     x = y.right
13.     if y != z.right
14.         RB-Transplant(T, y, y.right)  // is y farther down the tree?
15.         y.right = z.right  // replace y by its right child
16.         y.right.p = y  // z’s right child becomes y’s right child
17.     else
18.         x.p = y  // in case x is T.nil
19.     RB-Transplant(T, z, y)  // replace z by its successor y
20.     y.left = z.left  // and give z’s left child to y,
21.     y.left.p = y  // which had no left child
22.     y.color = z.color
23. if y.original-color == BLACK
24.     RB-Delete-Fixup(T, x)  // if any red-black violations occurred, correct them

RB-Delete-Fixup(T, x)
1.  while x != T.root and x.color == BLACK
2.      if x == x.p.left  // is x a left child?
3.          w = x.p.right  // w is x’s sibling
4.          if w.color == RED
5.              w.color = BLACK
6.              x.p.color = RED
7.              LEFT-ROTATE(T, x.p)  // case 1
8.              w = x.p.right
9.          if w.left.color == BLACK and w.right.color == BLACK
10.             w.color = RED  // case 2
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)  // case 3
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)  // case 4
22.             x = T.root
23.     else  // same as lines 3–22, but with “right” and “left” exchanged
24.         w = x.p.left
25.         if w.color == RED
```

```
            w.color = BLACK
            x.p.color = RED
            RIGHT-ROTATE(T, x.p)  // case 1
            w = x.p.left
        if w.right.color == BLACK and w.left.color == BLACK
            w.color = RED  // case 2
            x = x.p
        else
            if w.left.color == BLACK
                w.right.color = BLACK
                w.color = RED
                LEFT-ROTATE(T, w)  // case 3
            w = x.p.left
            w.color = x.p.color
            x.p.color = BLACK
            w.left.color = BLACK
            RIGHT-ROTATE(T, x.p)  // case 4
            x = T.root
x.color = BLACK
```