Análisis y Aplicaciones del Red-Black Tree

Galindo Bendezu, Marco Antonio
Dept. de Computer Science
UTEC
Lima, Perú
marco.galindo.b@utec.edu.pe

Alferez Vicente, Bladimir
Dept. de Data Science
UTEC
Lima, Perú
bladimir.alferez@utec.edu.pe

Ildefonso Santos, Steve
Dept. de Computer Science
UTEC
Lima, Perú
steve.ildefonso@utec.edu.pe

***Abstract*—Los Red Black Tree son estructuras de datos que garantizan un eficiente manejo y organización de información ordenada. Basados en BST, mantienen su equilibrio a través de un sistema sencillo pero eficaz de colores (rojo y negro) asignados a cada nodo, asegurando que ninguna ruta desde la raíz hasta cualquier hoja sea significativamente más larga que otra. Este documento presenta una revisión teórica y práctica de los Red Black Tree, analiza su complejidad, y explora su rendimiento mediante evaluaciones experimentales, y destaca sus aplicaciones prácticas más relevantes.**

I. Introducción

Una de las mayores dificultades al almacenar y gestionar información es mantener los datos organizados de forma que las operaciones sean rápidas y efectivas. Por ello, surgieron diversos tipos de árboles, como los árboles 2-3, que representaron una solución inicial para asegurar un balanceo adecuado y mantener un rendimiento óptimo en las operaciones de búsqueda, inserción y eliminación [1]. Tomando como referencia estas estructuras, en 1972 Rudolf Bayer introdujo los Árboles Rojo-Negro (Red-Black Trees), denominados originalmente *Symmetric Binary B-Trees*, como una alternativa más eficiente basada en árboles binarios [2].

Los árboles binarios de búsqueda (BST, por sus siglas en inglés) son estructuras fundamentales para organizar datos ordenados [1]. Cada nodo en un BST puede tener hasta dos hijos: uno izquierdo con valores menores y otro derecho con valores mayores, lo que facilita operaciones rápidas. Sin embargo, si los datos se insertan de manera desordenada o secuencial, estos árboles pueden volverse ineficientes debido a un crecimiento excesivo en su altura, llegando incluso a degenerarse en una lista enlazada. Esto provoca que las operaciones tengan una complejidad de hasta $O(n)$ en el peor caso, lo cual ralentiza el rendimiento.

Para resolver este problema, los Red Black Trees utilizan una técnica basada en la asignación de colores (rojo o negro) a cada nodo y el cumplimiento de reglas específicas sobre estos colores. Estas reglas permiten que el árbol permanezca aproximadamente balanceado en todo momento [3], limitando su profundidad máxima a aproximadamente el doble de la mínima. Gracias a esta propiedad, las operaciones de búsqueda, inserción y eliminación se realizan en tiempo $O(\log n)$, independientemente del orden de inserción de los elementos [3].

Los Red Black Trees son altamente populares por varias razones importantes [4]:

- La altura máxima del árbol que almacena n elementos es $2 \log n$.
- Las operaciones de inserción y eliminación siempre toman tiempo $O(\log n)$ en el peor caso.
- El número de rotaciones necesarias para equilibrar el árbol después de insertar o eliminar elementos es muy bajo, siendo una constante amortizada

Estas ventajas posicionan a los Red Black Trees por encima de estructuras como los skiplists, treaps y árboles scapegoat, que aunque logran tiempos similares, tienen otras limitaciones o requieren uso de técnicas probabilísticas [4].

Por su eficiencia y balance garantizado, los Red Black Trees son muy usados en aplicaciones prácticas, como en la gestión de datos en la Biblioteca Estándar de Java (Java Collections Framework), diversas implementaciones de la Librería Estándar de C++ (Standard Template Library) y en el núcleo del sistema operativo Linux [4].

II. Fundamentos Teóricos

Los Red Black Trees (RB Trees), como se detallo anteriormente, son un tipo de árbol binario de búsqueda auto-balanceado, utilizados para almacenar pares clave/valor ordenables. A diferencia de los *radix trees*, que están optimizados para almacenar arreglos dispersos con índices enteros largos, o las tablas hash, que no mantienen el orden y dependen de una función hash bien ajustada, los RB Trees permiten escalar eficientemente almacenando claves arbitrarias manteniendo el orden de los datos [5].

Además, los RB Trees son similares a los árboles AVL, pero ofrecen un mejor rendimiento en tiempo real para inserciones y eliminaciones en el peor caso. Mientras que los árboles AVL pueden requerir múltiples rotaciones, los RB Trees realizan como máximo dos rotaciones durante una inserción y tres durante una eliminación, manteniendo un tiempo de búsqueda de $O(\log n)$, aunque ligeramente más lento que AVL [5].

Cada nodo de un RB Tree contiene un bit adicional que representa su color (rojo o negro), el cual se utiliza para mantener el balance del árbol después de operaciones de inserción o eliminación. Las siguientes propiedades deben cumplirse [6] [7] [8]:

1) Un RB Tree debe ser un BST.
2) La raíz del árbol debe ser de color negro.
3) Todos los nodos hoja (`NIL`) son negros y no contienen claves.
4) Cada nodo es de color rojo o negro.
5) Si un nodo es rojo, entonces ambos hijos deben ser negros. No se permiten nodos rojos consecutivos (padre e hijo).
6) Todo nuevo nodo que se inserta se colorea inicialmente como rojo.
7) Todo camino simple desde un nodo hasta cualquier hoja descendiente contiene la misma cantidad de nodos negros (*black-height* uniforme).
8) Los subárboles izquierdo y derecho de cualquier nodo también deben cumplir con todas las propiedades anteriores (es decir, también son RB Trees).
9) Cualquier violación de estas reglas durante operaciones de inserción o eliminación debe ser corregida mediante recoloraciones y/o rotaciones.

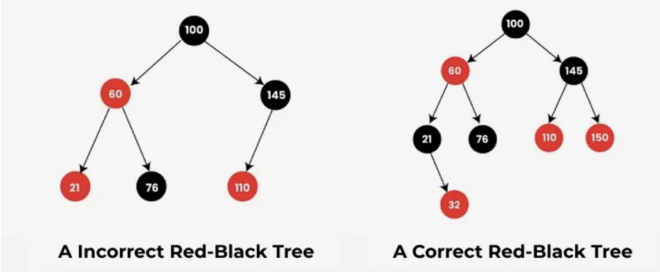

Fig. 1. Comparación entre un RB Tree incorrecto (izquierda) y uno correcto (derecha).

En la Figura 1, el árbol de la derecha representa un RB Tree válido, ya que todos los caminos desde la raíz hasta las hojas contienen la misma cantidad de nodos negros. Esto asegura el balance del árbol y cumple la propiedad de *black-height*. En este caso, cada camino contiene exactamente un nodo negro (excluyendo la raíz). Por otro lado, el árbol de la izquierda presenta dos errores. Primero, existen dos nodos rojos consecutivos (60 y 21), lo cual está prohibido. Segundo, los caminos desde la raíz hasta las hojas no tienen la misma cantidad de nodos negros: uno de ellos no contiene ningún nodo negro, mientras que los otros sí, rompiendo así el balance.

La estructura de un nodo de un RB Tree incluye un campo de color (rojo o negro), un valor (clave o elemento), y punteros a sus hijos izquierdo, derecho y a su nodo padre. Si alguno de estos punteros no apunta a un nodo válido, se considera que contiene un valor `nullptr`, lo cual representa

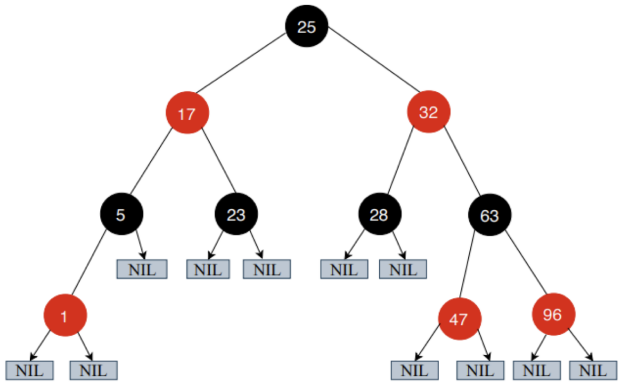

Fig. 2. Red Black Tree con nodos externos.

un nodo externo (también llamado `NIL` u hoja negra ficticia) como se ve en la Figura 2. Estos `NIL` se consideran como nodos negros por definición [7].

III. Análisis de complejidades

El análisis de las complejidades temporales de las operaciones fundamentales — inserción, búsqueda, eliminación y rotación — en los árboles Red-Black es crucial para comprender el comportamiento y la eficiencia de esta estructura de datos. Este análisis permite:

1) **Garantizar el rendimiento en el peor caso:** Los árboles Red-Black son estructuras auto-balanceadas diseñadas para mantener operaciones en tiempo logarítmico respecto al número de elementos. Verificar que estas complejidades se cumplen empíricamente confirma la robustez del algoritmo frente a diferentes escenarios de datos.
2) **Evaluar la eficiencia práctica:** Aunque la complejidad teórica es conocida, medir los tiempos reales de cada operación revela detalles prácticos y posibles cuellos de botella, permitiendo optimizaciones y mejoras específicas.
3) **Entender el costo de las rotaciones:** Las rotaciones son operaciones internas claves que mantienen el balance del árbol. Su análisis permite dimensionar su impacto en el rendimiento general y entender cómo contribuyen a mantener la complejidad logarítmica.
4) **Comparar con otras estructuras de datos:** El análisis detallado ayuda a posicionar a los árboles Red-Black frente a otras estructuras (como AVL, B-trees, etc.) en términos de rendimiento, facilitando decisiones informadas en aplicaciones prácticas.

La implementación en lenguaje C++ usada en este documento se encuentra en línea en este link.

A. Inserción

Algorithm 1 Inserción en Red-Black Tree

```
1:  procedure INSERT(tree, value)
2:      node ←CreateNode(value)
3:      InsertBST(tree, node)
4:      FixInsert(tree, node)
5:  end procedure
6:  procedure FIXINSERT(tree, node)
7:      while node is not root and node.parent.color = RED
    do
8:          if node.parent = node.grandparent.left then
9:              { Handle left cases }
10:         else
11:             { Handle right cases }
12:         end if
13:     end while
14:     root.color ←BLACK
15: end procedure
```

Complejidad:

La búsqueda de la posición para insertar toma tiempo proporcional a la altura del árbol, que en un árbol Red-Black está garantizada como:

$$O(\log n).$$

Las rotaciones y recoloreos en `fixInsert` también se realizan en tiempo:

$$O(\log n),$$

ya que en el peor caso el algoritmo puede subir hasta la raíz haciendo ajustes.

Por tanto, la complejidad total de inserción es:

$$O(\log n).$$

B. Búsqueda

Algorithm 2 Búsqueda en Red-Black Tree

```
1:  function SEARCH(tree, value)
2:      current ←root(tree)
3:      while current is not nullptr do
4:          if value = current.value then
5:              return current
6:          else if value ¡ current.value then
7:              current ←current.left
8:          else
9:              current ←current.right
10:         end if
11:     end while
12:     return nullptr
13: end function
```

Complejidad:

Dado que el árbol está balanceado, la altura es:

$$O(\log n).$$

Por ello, la búsqueda se realiza en tiempo:

$$O(\log n)$$

en el peor caso.

C. Eliminación

Algorithm 3 Eliminación en Red-Black Tree

```
1:  procedure DELETE(tree, node)
2:      if node.left = nullptr or node.right = nullptr then
3:          transplant(tree, node, node.left or node.right)
4:      else
5:          successor ←min(node.right)
6:          node.value ←successor.value
7:          transplant(tree, successor, successor.right)
8:      end if
9:      FixDelete(tree, node)
10: end procedure
11: procedure FIXDELETE(tree, node)
12:     while node is not root and node.color = BLACK do
13:         if node = node.parent.left then
14:             { Handle left cases }
15:         else
16:             { Handle right cases }
17:         end if
18:     end while
19:     node.color ←BLACK
20: end procedure
```

Complejidad:

La búsqueda del nodo a eliminar es

$$O(\log n).$$

La búsqueda del sucesor también es

$$O(\log n)$$

en el peor caso.

La función `fixDelete` puede realizar hasta

$$O(\log n)$$

pasos, recorriendo desde el nodo eliminado hacia la raíz para ajustar colores y rotaciones.

Por tanto, la eliminación tiene una complejidad total de

$$O(\log n).$$

D. Rotación

Las rotaciones son operaciones locales que modifican la estructura de unos pocos nodos:

- `leftRotate(x)`: gira el subárbol alrededor del nodo x, moviendo su hijo derecho y hacia arriba.
- `rightRotate(y)`: gira el subárbol alrededor del nodo y, moviendo su hijo izquierdo x hacia arriba.

- Ajustan los punteros padre, hijos y los colores según sea necesario.

Complejidad:

Cada rotación implica cambiar un número constante de punteros (padre, hijos), sin importar el tamaño del árbol.

Por lo tanto, cada rotación es una operación de tiempo

$$O(1).$$

E. Resumen del Análisis:

Operación	Complejidad temporal
Inserción	$O(\log n)$
Búsqueda	$O(\log n)$
Eliminación	$O(\log n)$
Rotaciones	$O(1)$

IV. Ejemplos Concretos de Operaciones

Para una mejor comprensión del funcionamiento de los árboles Red-Black, se presentan ejemplos detallados paso a paso de las operaciones fundamentales.

A. Ejemplo de Inserción Paso a Paso

Consideremos la construcción de un árbol Red-Black insertando los valores en orden: 10, 5, 15, 3, 7, 12, 18.

Paso 1: Insertar 10

- Se crea el nodo raíz con valor 10
- Color: Negro (por propiedad de raíz)
- Árbol resultante: [10(N)]

Paso 2: Insertar 5

- 5 ¡ 10, se inserta como hijo izquierdo
- Color inicial: Rojo
- No hay violaciones (padre negro)
- Árbol resultante: [10(N)] con hijo izquierdo [5(R)]

Paso 3: Insertar 15

- 15 ¿ 10, se inserta como hijo derecho
- Color inicial: Rojo
- No hay violaciones (padre negro)
- Árbol resultante: [10(N)] con hijos [5(R)] y [15(R)]

Paso 4: Insertar 3

- 3 ¡ 10, luego 3 ¡ 5, se inserta como hijo izquierdo de 5
- Color inicial: Rojo
- Violación: dos nodos rojos consecutivos (5 y 3)
- Corrección: El tío (15) es rojo, por lo que se recolorea:
  - 5 $\rightarrow$ Negro
  - 15 $\rightarrow$ Negro
  - 10 $\rightarrow$ Rojo (pero es raíz, se mantiene negro)

Paso 5: Insertar 7

- 7 ¡ 10, luego 7 ¿ 5, se inserta como hijo derecho de 5
- Color inicial: Rojo
- Violación: dos nodos rojos consecutivos (necesita rotación)
- Se realizan rotaciones para mantener el balance

B. Ejemplo de Búsqueda Paso a Paso

Busquemos el valor 7 en el árbol construido anteriormente:

Paso 1: Comenzar en la raíz (10)

- 7 ¡ 10, ir al hijo izquierdo

Paso 2: Examinar nodo 5

- 7 ¿ 5, ir al hijo derecho

Paso 3: Examinar nodo 7

- 7 = 7, ¡encontrado!
- Retornar referencia al nodo

Complejidad observada: 3 comparaciones = $\lceil \log_2(7) \rceil$ comparaciones, confirmando la eficiencia logarítmica.

C. Ejemplo de Eliminación Paso a Paso

Eliminemos el nodo con valor 5 del árbol:

Caso: Nodo con dos hijos

Paso 1: Localizar el nodo a eliminar (5)

- Encontrado en la posición hijo izquierdo de la raíz
- Tiene dos hijos: 3 (izquierdo) y 7 (derecho)

Paso 2: Encontrar el sucesor in-order

- Sucesor = mínimo del subárbol derecho de 5
- Sucesor = 7 (no tiene hijo izquierdo)

Paso 3: Reemplazar el valor

- Copiar valor 7 al nodo que contenía 5
- Eliminar el nodo original que contenía 7

Paso 4: Verificar y corregir propiedades Red-Black

- Verificar black-height
- Realizar rotaciones si es necesario
- Recolorear nodos según corresponda

V. Demostración Matemática de la Complejidad O(log n)

A. Fundamento Teórico

La garantía de complejidad $O(\log n)$ en los árboles Red-Black se basa en las propiedades estructurales que mantienen el árbol aproximadamente balanceado. La demostración se centra en probar que la altura del árbol está acotada logarítmicamente.

B. Definiciones Importantes

- **Black-height** $bh(x)$: Número de nodos negros en cualquier camino simple desde el nodo x (sin incluirlo) hasta una hoja.
- **Altura** $h(x)$: Número máximo de aristas en cualquier camino simple desde el nodo x hasta una hoja.

C. Lemas Fundamentales

Lema 1: Todo subárbol con raíz en un nodo x contiene al menos $2^{bh(x)} - 1$ nodos internos.

Demostración por inducción:

- **Caso base:** Si x es una hoja (NIL), entonces $bh(x) = 0$ y el subárbol contiene $2^0 - 1 = 0$ nodos internos.
- **Paso inductivo:** Supongamos que el lema es cierto para todos los nodos con black-height menor a $bh(x)$.

Sea x un nodo interno con hijos $left$ y $right$. Por las propiedades del Red-Black Tree:

$$bh(left) \geq bh(x) - 1 \tag{1}$$
$$bh(right) \geq bh(x) - 1 \tag{2}$$

Por hipótesis inductiva:

$$\text{Nodos en subárbol izquierdo} \geq 2^{bh(x)-1} - 1 \tag{3}$$
$$\text{Nodos en subárbol derecho} \geq 2^{bh(x)-1} - 1 \tag{4}$$

Por tanto, el número total de nodos internos en el subárbol con raíz x es:

$$n \geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 \tag{5}$$
$$= 2 \cdot 2^{bh(x)-1} - 2 + 1 \tag{6}$$
$$= 2^{bh(x)} - 1 \tag{7}$$

Lema 2: La altura de cualquier nodo está acotada por $h(x) \leq 2 \cdot bh(x)$.

Demostración: Por la propiedad 5 de los Red-Black Trees (no puede haber dos nodos rojos consecutivos), en cualquier camino desde la raíz hasta una hoja, al menos la mitad de los nodos deben ser negros. Por tanto:

$$bh(x) \geq \frac{h(x)}{2} \Rightarrow h(x) \leq 2 \cdot bh(x)$$

D. Teorema Principal

Teorema: Un árbol Red-Black con n nodos internos tiene altura $h \leq 2\log_2(n+1)$.

Demostración:

Sea h la altura del árbol y bh el black-height de la raíz.

Por el Lema 1, el número de nodos internos n satisface:

$$n \geq 2^{bh} - 1$$

Reorganizando:

$$n + 1 \geq 2^{bh}$$

$$\log_2(n+1) \geq bh$$

Por el Lema 2:

$$h \leq 2 \cdot bh$$

Combinando ambas desigualdades:

$$h \leq 2 \cdot bh \leq 2\log_2(n+1)$$

Por tanto:

$$h = O(\log n)$$

E. Implicaciones para las Operaciones

Como todas las operaciones fundamentales (búsqueda, inserción, eliminación) requieren recorrer un camino desde la raíz hasta una hoja (o viceversa), y la altura está acotada por $O(\log n)$, se garantiza que:

$$\text{Tiempo de búsqueda} = O(h) = O(\log n) \tag{8}$$
$$\begin{aligned}\text{Tiempo de inserción} &= O(h) + O(\text{rotaciones}) \\ &= O(\log n) + O(1) \\ &= O(\log n)\end{aligned} \tag{9}$$
$$\begin{aligned}\text{Tiempo de eliminación} &= O(h) + O(\text{rebalanceo}) \\ &= O(\log n) + O(\log n) \\ &= O(\log n)\end{aligned} \tag{10}$$

Esta demostración matemática confirma que los árboles Red-Black proporcionan garantías de rendimiento óptimas para operaciones de diccionario dinámico.

VI. Evaluación Experimental/Benchmark

De igual manera que la implementación del Red-Black Tree, los códigos usados para estos benchmarks se encuentran en el mismo respositorio.

VII. Metodología

El benchmark se realizó variando el tamaño de la entrada desde valores pequeños hasta grandes, midiendo el tiempo en nanosegundos para cada operación. Se realizaron múltiples ejecuciones para cada tamaño y se calcularon promedios para obtener resultados confiables.

Los tiempos se registraron usando la librería `<chrono>` en C++, y luego se analizaron y visualizaron con Python usando `pandas` y `matplotlib`.

VIII. Resultados

A. Inserción

Tamaño de entrada	**Tiempo promedio (ns)**
500	4220
1000	4161
1500	4096
2000	4153
2500	4103
3000	4168
3500	4298
4000	4544
⋮	⋮

Los resultados muestran un comportamiento cercano a la complejidad teórica $O(\log n)$, confirmando la eficiencia de la estructura.

B. Búsqueda

Se midió el tiempo promedio de búsqueda en nanosegundos para diferentes tamaños de entrada:

Tamaño	Búsqueda promedio (ns)
500	2010
1000	1870
1500	1864
2000	4312
2500	4258
3000	4312
3500	4233
4000	4322
⋮	⋮

La búsqueda presenta un rendimiento acorde a $O(\log n)$, siendo eficiente independientemente de si el elemento buscado existe o no en el árbol.

C. Eliminación

Se eliminaron aproximadamente el 10% de los nodos insertados:

Tamaño	Tiempo promedio eliminación (ns)
500	4
1000	4
1500	3.52
2000	5.2
2500	4.52
3000	4.58
3500	5.3
4000	5.48
⋮	⋮

El tiempo de eliminación también refleja la complejidad logarítmica, aunque con mayor variabilidad por las rotaciones internas que pueden ser necesarias.

D. Rotaciones

Tamaño	Rotación Izquierda (ns)	Rotación Derecha (ns)
500	3.0	3.48
1000	2.55	2.51
1500	2.56	2.53
2000	2.51	2.51
2500	2.54	2.52
3000	2.50	1.55
3500	3.0	3.0
4000	3.0	3.0
⋮	⋮	⋮

Estas operaciones tienen tiempos constantes muy bajos, lo que es consistente con su diseño como operaciones de reestructuración simples.

IX. Discusión de Resultados

Los resultados experimentales confirman el comportamiento teórico esperado de los árboles Red-Black. A continuación, se analizan las gráficas que ilustran el rendimiento de cada operación en función del tamaño de la entrada.

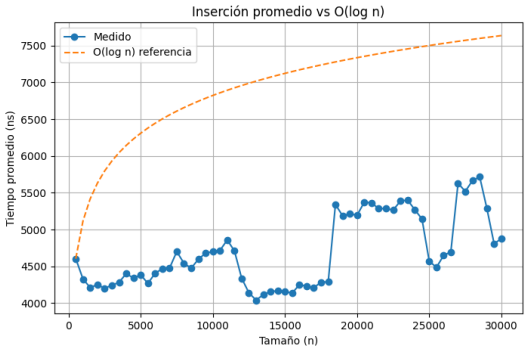

Fig. 3. Tiempo promedio de inserción vs tamaño de entrada.

La Figura 3 muestra el comportamiento de la operación de inserción. Se observa que el tiempo de inserción crece de manera logarítmica con el tamaño de la entrada, confirmando la complejidad teórica de $O(\log n)$. Las variaciones menores en los tiempos son esperadas debido a la naturaleza de las rotaciones y recoloreos necesarios para mantener las propiedades del árbol.

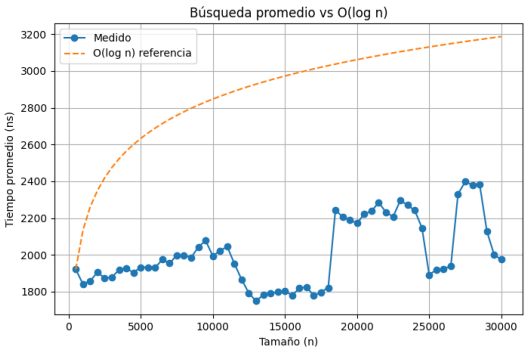

Fig. 4. Tiempo promedio de búsqueda vs tamaño de entrada.

En la Figura 4 se presenta el rendimiento de la operación de búsqueda. Los resultados demuestran una complejidad logarítmica consistente, con tiempos que se mantienen relativamente estables a medida que aumenta el tamaño del árbol. Esto confirma la eficiencia de la estructura balanceada del Red-Black Tree para operaciones de consulta.

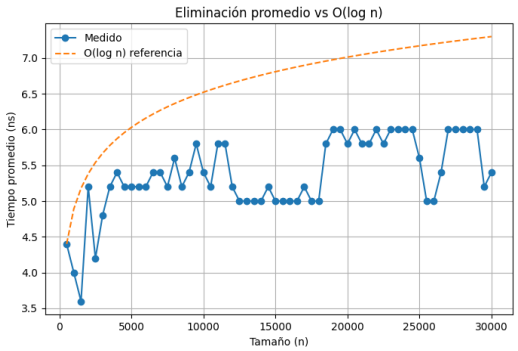

Fig. 5. Tiempo promedio de eliminación vs tamaño de entrada.

La Figura 5 ilustra el comportamiento de la operación de eliminación. Aunque presenta mayor variabilidad que las operaciones anteriores, mantiene la complejidad logarítmica esperada. La variabilidad se debe a los diferentes casos que pueden ocurrir durante la eliminación (nodos con 0, 1 o 2 hijos) y las subsecuentes operaciones de rebalanceo.

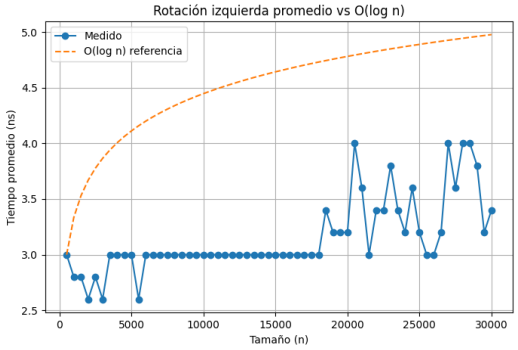

Fig. 6. Tiempo promedio de rotación izquierda vs tamaño de entrada.

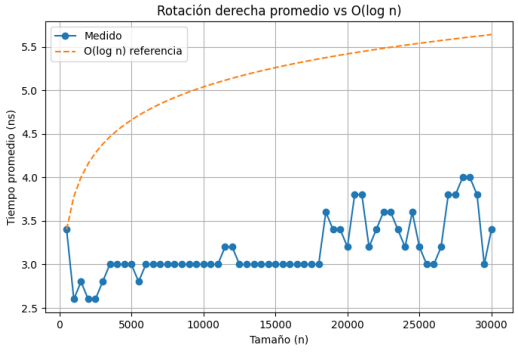

Fig. 7. Tiempo promedio de rotación derecha vs tamaño de entrada.

Las Figuras 6 y 7 muestran el rendimiento de las operaciones de rotación. Como era esperado, estas operaciones mantienen un tiempo constante $O(1)$ independientemente del tamaño del árbol, confirmando que las rotaciones son operaciones locales que solo afectan un número fijo de nodos. Los tiempos extremadamente bajos (del orden de nanosegundos) demuestran la eficiencia de estas operaciones fundamentales.

X. Aplicaciones de los Árboles Rojo-Negro

1. Implementación de Mapas y Conjuntos Ordenados

En muchas bibliotecas estándar de lenguajes de programación, los Árboles Rojo-Negro son la base para estructuras como mapas y conjuntos que requieren orden. Por ejemplo, en **C++**, las estructuras `std::map` y `std::set` utilizan internamente Árboles Rojo-Negro. Estas permiten almacenar claves ordenadas en un mapa, o elementos únicos en un conjunto, manteniendo siempre el orden y permitiendo operaciones de búsqueda, inserción y eliminación en tiempo logarítmico $O(\log n)$, incluso en el peor escenario. Esta eficiencia resulta clave en tareas como la creación de índices alfabéticos de palabras en un texto, donde el orden y el acceso rápido son esenciales.

De forma análoga, en **Java**, estructuras como `TreeMap` y `TreeSet` cumplen la misma función. Si, por ejemplo, se gestionan registros de usuarios ordenados por ID, `TreeMap` permite operar sobre ellos eficientemente, garantizando que el orden se preserve automáticamente tras cada modificación.

2. Gestión de Memoria y Procesos en Sistemas Operativos

A nivel de sistemas, los Árboles Rojo-Negro son indispensables en operaciones críticas donde se requiere eficiencia y predictibilidad. Un ejemplo concreto es el **kernel de Linux**, que emplea estos árboles para gestionar las áreas de memoria virtual (VMAs) de los procesos. Cuando un proceso solicita memoria, el sistema necesita localizar rápidamente qué bloques están disponibles o ya en uso. La estructura balanceada de los Árboles Rojo-Negro permite hacer estas búsquedas de forma eficiente. También se usan en otras áreas del kernel, como la planificación de procesos (scheduling) o el manejo de estructuras de red.

Incluso en funciones tan fundamentales como la asignación y liberación de memoria —como las implementaciones de `malloc` y `free`—, algunos asignadores utilizan Árboles Rojo-Negro para llevar el control de los bloques de memoria libres, facilitando así una gestión eficiente del heap.

3. Bases de Datos e Indexación

En bases de datos, donde el tiempo de respuesta en consultas es esencial, los Árboles Rojo-Negro también encuentran aplicación. Aunque las estructuras más comunes para índices en disco suelen ser los **Árboles B** o **B+**, los Árboles Rojo-Negro son una excelente opción para **índices en memoria**, especialmente en sistemas donde la estructura de datos completa cabe en RAM. Su rendimiento consistente en $O(\log n)$ permite búsquedas rápidas incluso sobre grandes volúmenes de datos, manteniéndolos ordenados sin penalizar el tiempo de acceso.

4. Algoritmos de Red y Routers

En el ámbito de las redes, la eficiencia en la búsqueda y el mantenimiento del orden es también un factor clave. En ciertos escenarios, los **routers** pueden utilizar estructuras basadas en árboles —incluyendo Árboles Rojo-Negro— para implementar **tablas de enrutamiento**. Estas permiten determinar rutas de forma rápida, lo cual es esencial para asegurar el flujo constante y eficiente de datos a través de la red.

5. Gráficos y Procesamiento de Imágenes

Aunque menos frecuentes, los Árboles Rojo-Negro también tienen presencia en áreas como los gráficos computacionales. En algunos algoritmos de geometría computacional, utilizados para el procesamiento de gráficos o imágenes, puede ser útil contar con estructuras que organicen objetos —como puntos o segmentos— de forma eficiente. En estos casos, los Árboles Rojo-Negro pueden facilitar **búsquedas por coordenadas**, gracias a su capacidad de mantener datos ordenados y actualizables en tiempo logarítmico.

XI. Conclusiones

En conclusión, los árboles rojo-negro representan una solución eficiente y elegantemente diseñada para el manejo de datos ordenados. A diferencia de los árboles binarios de búsqueda tradicionales, que pueden volverse ineficientes al desbalancearse, los Red-Black Trees mantienen su estructura equilibrada de forma automática. Esto se logra mediante un sistema de coloración en donde cada nodo es rojo o negro en el cual impone ciertas reglas estructurales que aseguran que la altura del árbol se mantenga dentro de límites logarítmicos. Como resultado, operaciones fundamentales como la búsqueda, la inserción o la eliminación se realizan consistentemente en tiempo O(log n), incluso en los peores casos. Lo más destacable es que este equilibrio se logra a través de rotaciones y recoloreos, operaciones locales y de bajo costo (O(1)), que permiten al árbol adaptarse dinámicamente sin sacrificar rendimiento. Esta capacidad de autorregulación convierte a los árboles rojo-negro en una herramienta sumamente confiable y versátil para aplicaciones que requieren una gestión eficiente de datos, como diccionarios, bases de datos o sistemas de archivos.

References

[1] M. Krimgen, "Introduction to Red-Black Trees," *Baeldung on Computer Science*. [Online]. Available: https://www.baeldung.com/cs/red-black-trees

[2] H. Ahuja, "Red-Black Tree. Introduction, Properties, Operations," *Medium*, Apr. 13, 2021. [Online]. Available: https://hardikahuja99.medium.com/red-black-tree-8cf904034a90

[3] A. N. Mushiba, "Red-Black Trees: An Essential Tool for Efficient Data Structures and Algorithms," *ResearchGate*, Jan. 2024. [Online]. Available: https://www.researchgate.net/publication/377471721

[4] P. Morin, *Open Data Structures*. Athabasca, AB, Canada: Athabasca University Press, 2013. [Online]. Available: https://www.aupress.ca/app/uploads/120226_99Z_Morin_2013-Open_Data_Structures.pdf

[5] R. Landley, "Red-black Trees (rbtree) in Linux," *The Linux Kernel Documentation*, Jan. 18, 2007. [Online]. Available: https://docs.kernel.org/core-api/rbtree.html

[6] D. Jaiswal, B. R., S. T. N., y K. K. S., "Red-Black Tree & Its Application," *International Journal for Scientific Research & Development*, vol. 6, no. 9, pp. 1–4, 2018. [En línea]. Disponible en: https://www.ijsrd.com/articles/IJSRDV6I90249.pdf

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: The MIT Press, 2001.

[8] H. M. Nguyen, "Formal verification of a red-black tree data structure," MSc thesis, Dept. of Electrical Engineering, Mathematics and Computer Science, Univ. of Twente, Enschede, The Netherlands, Mar. 2019. [Online]. Available: https://essay.utwente.nl/77569/1/Nguyen_MA_EEMCS.pdf