Red-Black Tree Algorithm

*Informe Académico

Irán Alvarez
Computer Science Dept.
Universidad de Ingeniería y Tecnología
Lima, Perú
iran.alvarez@utec.edu.pe

Adrián Auqui
Computer Science Dept.
Universidad de Ingeniería y Tecnología
Lima, Perú
adrian.auqui@utec.edu.pe

Brigitte Rojas
Computer Science Dept.
Universidad de Ingeniería y Tecnología
Lima, Perú
brigitte.rojas@utec.edu.pe

***Abstract*—El presente informe se basa en una investigación sobre la estructura de datos del árbol rojo-negro, sus propiedades, operaciones, limitaciones, así como sus usos en la industria.**

***Index Terms*—Algoritmos de árboles, árboles autobalanceados, estructura de datos, árbol binario de búsqueda, Árbol Rojo y Negro, Rotaciones.**

I. Introducción

Los árboles rojo - negro son un esquema de árbol de búsqueda que garantiza el peor tiempo de ejecución de operaciones básicas basada en conjuntos dinámicos de O(log(n)). "A historically popular alternative to the AVL tree is the red-black tree" (Data Structures and Algorithms, 2000, p. 145).

II. Propiedades

1) **Propiedad de la raíz:** La raíz es negra.
2) **Propiedad externa:** Cada hoja es negra en el árbol rojo-negro.
3) **Propiedad interna:** Los hijos de un nodo rojo son negros.
4) **Propiedad de ruta:** Cada ruta simple desde la raíz hasta el nodo hoja descendiente contiene la misma cantidad de nodos negros.

A. Comparación con el árbol AVL

Los árboles AVL están más equilibrados en comparación con los árboles rojo-negro, pero pueden provocar más rotaciones durante la inserción y eliminación. Entonces, si tenemos una aplicación que implique inserciones y eliminaciones frecuentes, se deben preferir los árboles rojo-negro. Y si las inserciones y eliminaciones son menos frecuentes y la búsqueda es una operación más frecuente, entonces se debe preferir el árbol AVL al árbol rojo - negro.

B. Ventajas

- Los árboles rojos y negros tienen una complejidad temporal garantizada de $O(log(n))$ para operaciones básicas como inserción, eliminación y búsqueda.
- Los árboles rojos y negros se pueden utilizar en una amplia gama de aplicaciones debido a su rendimiento eficiente y versatilidad.
- El mecanismo utilizado para mantener el equilibrio es relativamente simple y fácil de entender.

C. Desventajas

- Los árboles rojos y negros requieren un bit adicional de almacenamiento para cada nodo para almacenar el color del nodo(rojo o negro). Por ende, requiere más espacio de memoria.
- Complejidad en la implementación debido al cambio de color de los nodos y sus rotaciones. Se necesita verificar si cumplen las propiedades no solo verificando el mismo nodo a insertar o eliminar sino también a sus parientes como abuelo, padre, tío.

III. Conceptos importantes

Cada árbol rojo negro con n nodos tiene una altura

$$<= 2Log2(n+1) \tag{1}$$

Esto se puede demostrar utilizando los siguientes hechos:

Para un árbol binario general, sea k el número mínimo de nodos en todas las rutas de raíz a NULL, entonces:

$$n >= 2k–1 \tag{2}$$

(por ejemplo, si k es 3, entonces n es al menos 7). Esta expresión también se puede escribir como:

$$k <= Log2(n+1). \tag{3}$$

De la propiedad 4 de los árboles Rojo-Negro y la afirmación anterior, podemos decir que en un árbol Rojo-Negro con n nodos, hay un camino de raíz a hoja con como máximo

$$Log2(n+1) \tag{4}$$

nodos negros

De las propiedades 3 y 5 de los árboles rojo-negro, podemos afirmar que el número de nodos negros en un árbol rojo - negro es al menos

$$n/2 \tag{5}$$

donde n es el número total de nodos.

Un árbol rojo - negro que contiene n nodos internos tiene una altura de

$$O(log(n)) \tag{6}$$

- bh(v) = número de nodos negros (sin contar v si es negro)

Lema:

$$2^(bh)(v) - 1 \tag{7}$$

A. Altura

La altura del negro es la cantidad de nodos negros en un camino desde la raíz hasta la hoja.

$$h/2 \tag{8}$$

IV. DESARROLLO DE LA ESTRUCTURA DE DATOS

Definiciones de la estructura

El árbol rojo y negro se implementa a partir de una estructura genérica que guarda el dato del nodo, los punteros a sus hijos **(izquierda y derecha)** y el puntero del padre del nodo. Ello con el fin de poder acceder a los tíos, hermanos, abuelos, padres e hijos de un solo nodo, lo cual contribuye eficazmente en las operaciones de inserción, eliminación y búsqueda.

Estructura binaria de búsqueda:

Es útil para mantener el equilibrio del árbol, ya que los nodos internos y la raíz tienen a lo mucho dos hijos. Asimismo, está estructurado de tal manera que los subárboles del lado izquierdo son menores en valor que los subárboles derechos; esto se debe a que sigue la misma estructura del árbol binario de búsqueda balanceada con algunas variantes.

Equilibrio:

La altura desde cualquier rama partiendo de la raíz hasta cualquier hoja o nodos sin hijos no puede ser más de dos veces la altura de cualquier otra rama en el árbol. Esto es una característica que resalta para que las operaciones que se realicen dentro del árbol tengan una complejidad logarítmica.

Colores de los nodos:

Cada nodo del árbol tiene un atributo cuyo valor es rojo o negro. Todo nodo a insertar en el árbol es de color rojo, a excepción de la raíz que siempre es de color negro. A medida que se inserten o eliminen los nodos, estos cambiarán de color debido a las rotaciones; así pues, el árbol rojo y negro mantendrá sus propiedades.

Propiedades del color de los nodos:

- La raíz es siempre de color negro.
- Todos los nodos rojos tienen dos hijos de color negro.
- Cualquier camino que parte desde la raíz hasta las hojas contiene la misma cantidad de nodos de color negro.

Rotaciones:

Se utilizan rotaciones (izquierda y derecha) para mantener el balance durante las operaciones de inserción y eliminación.

Rotación a la Izquierda: La rotación a la izquierda se realiza tomando en cuenta la relación entre un nodo y su hijo derecho.

Rotación a la Derecha: La rotación a la derecha se realiza tomando en cuenta la relación entre un nodo y su hijo izquierdo.

V. OPERACIONES

Búsqueda

La operación de búsqueda en un árbol Rojo-Negro es una de las operaciones mas eficientes que aporta significativamente al algoritmo y su complejidad. Es por ello que se mantiene en un orden logarítmico, especialmente útil cuando se maneja una gran cantidad de datos.

```
Red-Black Tree Search
function SEARCHRBTREE(root, value)
  if root is null or root.key is equal to value then
    return root
  end if
  if value is less than root.key then
    return SEARCHRBTREE(root.left, value)
  else
    return SEARCHRBTREE(root.right, value)
  end if
end function=0
```

Fig. 1. Algoritmo de búsqueda

Se realiza una búsqueda de un valor en un árbol y comienza desde la raíz y se desplaza hacia abajo comparando el dato a buscar con el dato del nodo. Si el dato es menor, se mueve al subárbol izquierdo; si es mayor, al subárbol derecho. El proceso continúa hasta encontrar el dato o llegar a un nodo nulo, indicando que el dato no está en el árbol. Finalmente, el algoritmo devuelve el nodo encontrado o un valor nulo si el dato no existe. La complejidad es

$$O(log(n)) \tag{9}$$

aprovechando la estructura balanceada del árbol.

Inserción

El árbol Rojo-Negro es un algoritmo perteneciente a la familia de árboles binarios de búsqueda balanceados como el AVL, presenta algunas diferencias claves en su enfoque. Por ejemplo, al añadir un nuevo nodo al árbol, el nodo se colorea inicialmente en rojo, a excepción de la raíz que siempre es de color negro. El procedimiento de inserción sigue algunas reglas básicas para realizar esta operación como las rotaciones heredadas del AVL, sin embargo; también se emplea la estrategia de cambio de color, por lo que se apoya de una función auxiliar que modifica el árbol luego de las rotaciones para que todas las propiedades se cumplan y este balanceado.

La lógica de RB-Insert-Fixup ajusta el árbol rojo-negro después de la inserción de un nodo en el árbol. Es decir, utiliza rotaciones y cambios de color para mantener las propiedades del árbol, garantizando que no haya violaciones de las reglas del árbol. El bucle while recorre el árbol hasta que se corrigen todas las violaciones, y al final, asegura que la raíz del árbol sea de color negro.

```
Red-Black Tree Insertion
procedure RB-INSERT(T, z) {T is the Red-Black Tree, z is the new node to be inserted}
    BST-Insert(T, z) {Perform standard Binary Search Tree insertion}
    z.color ← RED {Newly inserted node is always colored RED}
    RB-Insert-Fixup(T, z) {Fix any violations introduced during the insertion}
end procedure=0
```

Fig. 2. Algoritmo de inserción

```
RB-Insert-Fixup
procedure RB-INSERT-FIXUP(T, z)
    while z.parent.color = RED do
        if z.parent = z.parent.parent.left then
            y ← z.parent.parent.right {Uncle of z}
            if y.color = RED then
                z.parent.color ← BLACK {Case 1: Recolor parent and uncle}
                y.color ← BLACK
                z.parent.parent.color ← RED
                z ← z.parent.parent {Move up the tree}
            else
                if z = z.parent.right then
                    z ← z.parent {Case 2: Left-rotate to make it Case 3}
                    Left-Rotate(T, z)
                end if
                z.parent.color ← BLACK {Case 3: Recolor and right-rotate}
                z.parent.parent.color ← RED
                Right-Rotate(T, z.parent.parent)
            end if
        else{Symmetric cases for the right side}
            y ← z.parent.parent.left
            if y.color = RED then
                z.parent.color ← BLACK
                y.color ← BLACK
                z.parent.parent.color ← RED
                z ← z.parent.parent
            else
                if z = z.parent.left then
                    z ← z.parent
                    Right-Rotate(T, z)
                end if
                z.parent.color ← BLACK
                z.parent.parent.color ← RED
                Left-Rotate(T, z.parent.parent)
            end if
        end if
    end whileT.root.color ← BLACK {Ensure root is always black}
end procedure=0
```

Fig. 3. Pseudocódigo que modifica el árbol después de una inserción

```
Red-Black Tree Deletion
function DELETERBTREE(root, value)
    nodeToDelete ← SEARCHRBTREE(root, value)
    if nodeToDelete is null then
        return root {Value not found, nothing to delete}
    end if
    replaceNode ← GETSUCCESSOR(nodeToDelete)
    COPYNODEDATA(nodeToDelete, replaceNode)
    fixNode ← GETFIXUPNODE(replaceNode)
    if COLOR(nodeToDelete) is black then
        if COLOR(replaceNode) is red then
 SETCOLOR(replaceNode, black)
        else
            RBDELETEFIXUP(root, fixNode)
        end if
    end if
    REMOVENODE(nodeToDelete)
    return root
end function=0
```

Fig. 4. Algoritmo de eliminación

```
Red-Black Tree Delete Fixup
procedure RBDELETEFIXUP(root, fixNode)
    while fixNode ≠ root and COLOR(fixNode) is black do
        if fixNode = fixNode.parent.left then
            sibling ← fixNode.parent.right
            if COLOR(sibling) is red then
                SETCOLOR(sibling, black)
                SETCOLOR(fixNode.parent, red)
                LEFTROTATE(root, fixNode.parent)
                sibling ← fixNode.parent.right
            end if
            if COLOR(sibling.left) is black and COLOR(sibling.right) is black then
                SETCOLOR(sibling, red)
                fixNode ← fixNode.parent
            else
                if COLOR(sibling.right) is black then
                    SETCOLOR(sibling.left, black)
                    SETCOLOR(sibling, red)
                    RIGHTROTATE(root, sibling)
                    sibling ← fixNode.parent.right
                end if
                SETCOLOR(sibling, Color(fixNode.parent))
                SETCOLOR(fixNode.parent, black)
                SETCOLOR(sibling.right, black)
                LEFTROTATE(root, fixNode.parent)
                fixNode ← root
            end if
        else{Symmetric cases for the right subtree}
        end if
    end while
    SETCOLOR(fixNode, black)
end procedure=0
```

Fig. 5. Pseudocódigo que modifica el árbol después de una eliminación

Eliminación

El proceso de eliminar un nodo del árbol rojo y negro es complejo debido a la cantidad de condicionales y funciones empleadas para mantener organizado y con las propiedades que se requieren.

La función se encarga de corregir posibles violaciones a las propiedades del árbol después de realizar una eliminación. Después de quitar un nodo del árbol, el $DeleteFixup$ ajusta la estructura y los colores de los nodos para garantizar que el árbol siga siendo un árbol válido. Durante este proceso, se pueden realizar rotaciones y cambios de color en el árbol para restablecer las propiedades esenciales, como la propiedad de color de los nodos y la propiedad de distancia negra constante. El pseudocódigo esta basado de una manera recursiva, subi-

endo desde el nodo eliminado hasta la raíz del árbol, lo cual nos asegura que todas las propiedades del árbol rojo-negro se mantengan intactas.

RESTRICCIONES

1) Un nodo posee un color, ya sea rojo o negro.
2) La raíz y las hojas son siempre de color negro.
3) Los nodos nuevos insertados son siempre de color rojo, a excepción de la raíz.
4) Si un nodo es rojo, entonces sus hijos son de color negro.
5) Todos los caminos desde un nodo hasta sus descendientes contienen el mismo número de nodos de color negro.

VI. EJEMPLOS DEL ALGORITMO

A continuación se presentan ejemplos concretos que mostraran como funciona las operaciones menciones del algoritmo.

A. Inserción

La operación de inserción se divide por los siguientes casos:

1) El nodo a insertar es la raíz

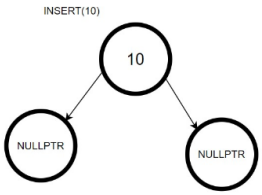

Fig. 6. CASO I: Insertar 10

Se puede ver que la raíz tiene la propiedad de color negro por ser el primer nodo en ser insertado, así como dos hijos que apuntan a nullptr y son de color negro también.

2) El nodo insertado tiene como padre a un nodo rojo

Fig. 7. Caso II: Insertando 24

Se puede ver que se hace un cambio de color al nodo padre del nodo a insertar. Como se puede ver el nodo padre del nodo a insertar (24) es el nodo con valor 17, ahora paso de ser rojo a negro debido a que no se admite un padre e hijo de color rojo a la vez.

3) El nodo insertado tiene nodos de color rojo a su padre y tío
El nodo a insertar es el número 40 que se debería ubicar a la derecha del nodo 34, sin embargo, como todo nodo que se inserta toma el color rojo y tanto el padre como el tío son del mismo color, a diferencia del caso anterior, ahora se necesita verificar el abuelo del nodo a insertar y como cada nodo tiene un puntero de su padre, entonces se puede acceder también al abuelo. Lo que se realiza es un cambio de color del abuelo, es decir de negro a rojo, lo que ocasiona que tanto el tío y el padre del nodo insertado también cambien de color, es decir de color rojo a negro; y finalmente se obtiene un árbol que cumple con la propiedad.

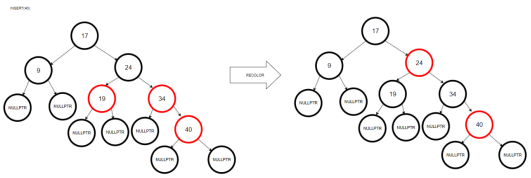

Fig. 8. Caso III: Insertando 40

4) El nodo insertado tiene como padre a un nodo rojo y a su tío de color negro (Se forma una forma triángula entre el abuelo, padre e hijo)

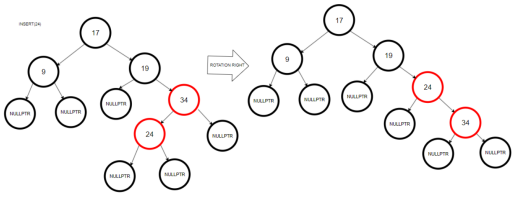

Fig. 9. Caso IV: Rotación a la derecha luego de insertar 24

En este ejemplo, se puede analizar que al insertar el nodo 24 en el árbol, existe un conflicto, pues hay dos nodos de color rojo, es decir 24 y 34 asimismo notamos que el tío del nodo es de color negro. Entonces, para balancear el árbol, se realiza una rotación a la derecha.

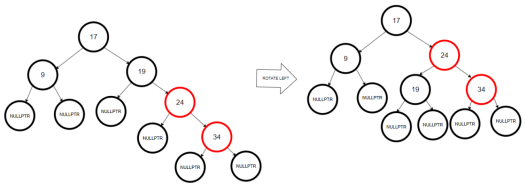

Fig. 10. Caso IV: Rotación a la izquierda luego de insertar 24

Sin embargo, el árbol aún no se encuentra balanceado así que se necesita una rotación, pero hacia la izquierda donde el abuelo del nodo insertado 19 pasa a ser el hijo izquierdo de 24 y hermano de 34.
Por último, notamos que el árbol esta incumpliendo la propiedad de color, ya que 24 y 34 son de color rojo, así que cambiamos el color del padre del nodo insertado, es decir 24 y el hermano del nodo insertado también cambia de color, solo porque 19 y 34 tienen distinto color.

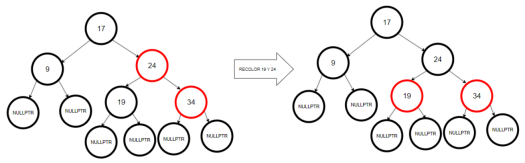

Fig. 11. Caso IV: Cambio de color a los nodos con valor 19 y 24

5) El padre del nodo insertado es rojo, el tío es negro. (Se forma una forma diagonal entre el abuelo, padre e hijo)

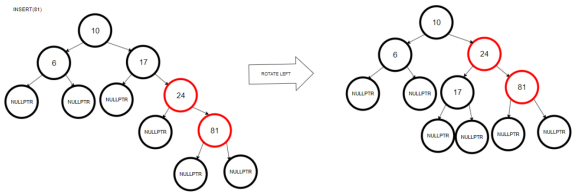

Fig. 12. Caso V: Insertando 81

Cuando se tiene dos nodos de color rojo juntos y formando una diagonal o línea, entonces se hace una rotación al lado contrario donde se ubican los nodos, es decir, a la izquierda en el ejemplo.

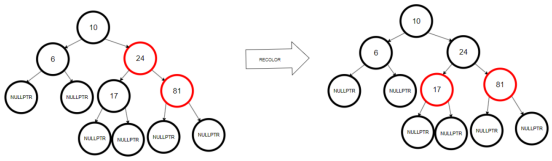

Fig. 13. Caso V: Aplicando cambio de color a los nodos

Una vez realizada la rotación simple, pues notamos que las propiedades aún no cumplen porque el nodo 24 y 81 son rojos aún. Es por ello que luego se emplea la técnica del cambio de color al padre del nodo, es decir cambiamos el color de 24 y también a su hijo izquierdo para mantener las propiedades.

B. *Eliminación*

La operación de eliminación se divide por los siguientes casos:

1) El nodo a eliminar es una hoja o tiene un solo hijo

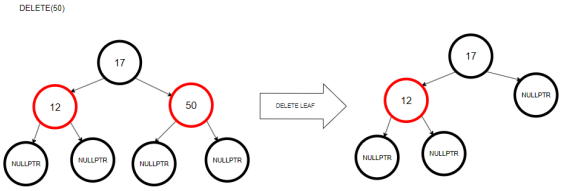

Fig. 14. Caso I: Eliminando 50

Si el nodo es una hoja, el nodo se elimina sin modificar la estructura del árbol tal como sucede con el árbol binario de búsqueda balanceado. Ahora el padre del nodo eliminado tiene como hijo a $nullptr$.

2) El nodo a eliminar o su padre tiene color rojo

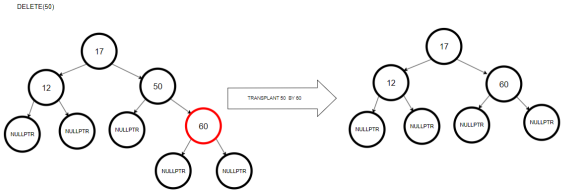

Fig. 15. Caso II: Eliminando 50

El nodo con valor 50 solo tiene un hijo el cual es 60. Entonces cuando se elimine 50, el nodo hijo toma su lugar heredando el color del padre que se eliminó, el cual es negro. Es por ello que se aprecia que 60 pasa a ser hijo derecho de 17 y es negro.

3) El nodo a eliminar y su padre es de color negro
   a) El nodo pasa a ser ”double black”

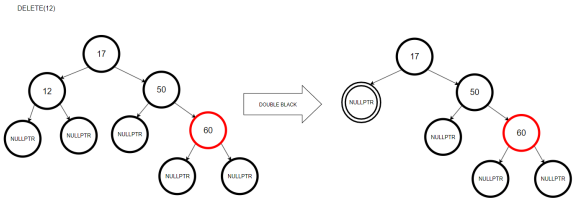

Fig. 16. Caso 3.1: Eliminando 12

Cuando eliminamos un nodo negro que es una hoja en un árbol rojo-negro, se introduce un $excesodenegro$ o $doubleblack$ en el camino desde la raíz hasta el nodo eliminado. Este exceso de negro se representa como un nodo ”doble negro” en el nodo hermano del nodo eliminado. Ahora, el objetivo es corregir esta

situación para mantener las propiedades del árbol rojo-negro. Para ello, mostraremos los subcasos para controlar este exceso.

i) Se realiza lo siguiente mientras el nodo actual u sea doble negro y no sea la raíz. Sea s el hermano del nodo.

   A) Si el hermano s es negro y al menos uno de los hijos del hermano es rojo, se realizan rotaciones. El hijo rojo de s sea r. Este caso se puede dividir en cuatro subcasos según las posiciones de s y r:
   B) Caso I: izquierda-izquierda: Sea s es el hijo izquierdo de su padre y r es el hijo izquierdo de s o ambos hijos de s son rojos.
   C) Caso II: izquierda-derecha: Sea s es el hijo izquierdo de su padre y r es el hijo derecho de s.
   D) Caso III: derecha-derecha: Sea s es el hijo derecho de su padre y r es el hijo derecho de s, o ambos hijos de s son rojos.
   E) Caso IV: derecha-izquierda: Sea s es el hijo derecho de su padre y r es el hijo izquierdo rojo de s.

   F) El hermano es de color negro y ambos de sus hijos son negros, se realiza un cambio de color y se vuelve a llamar recursivamente al padre si el padre es negro.
   En este caso, si el padre era rojo, entonces no necesitaríamos recurrir para el padre, simplemente podemos hacerlo negro (rojo + doble negro = negro simple). Además, después de hacerlo negro, la propiedad de doble negro se propaga hacia arriba y necesitamos continuar con el proceso de ajuste para garantizar que el árbol rojo-negro mantenga sus propiedades.
   G) El hermano es de color rojo Se realiza rotación para mover al antiguo hermano hacia arriba; además, se cambia el color del antiguo hermano y su padre. El nuevo hermano siempre es de color negro. Esto convierte principalmente el árbol al caso de hermano negro (por rotación) y se trata verificando dos casos. A continuación detallamos los subcasos para resolver el conflicto:
   H) Caso: Izquierda Ahora s es el hijo izquierdo de su padre.
   I) Caso: Derecha Ahora s es el hijo derecho de su padre.

ii) Si u es la raíz, entonces se cambia a negro simple (single black). Ello ocasiona que la altura negra del árbol completo se reduzca en 1.

C. *Búsqueda*

La operación de búsqueda es tal cual se implementa en el árbol AVL, ya que se inicia comparando el nodo a buscar con la raíz, dependiendo del valor del dato que se quiere encontrar, se realizan comparaciones. Por ejemplo, si se quiere buscar en el siguiente árbol (ver figura 17) entonces hacemos la pregunta si ese valor a buscar es menor o mayor que la raíz; si fuese menor pasamos al segundo nivel del árbol y seguimos comparando hasta encontrar el valor o recorrer el subárbol izquierdo y no encontrarlo. Lo mismo ocurre si el dato es mayor que la raíz porque se realizan las mismas comparaciones hasta encontrar el valor o hasta que se haya comparado todo el subárbol derecho.

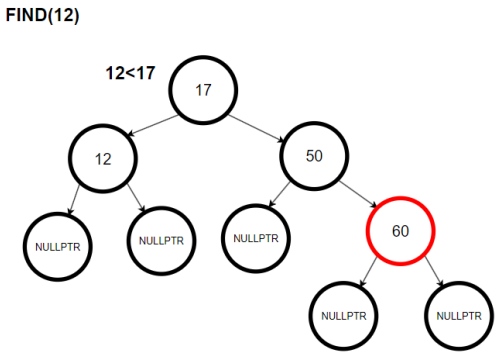

Fig. 17. Buscar 12: Se compara 17 con el dato

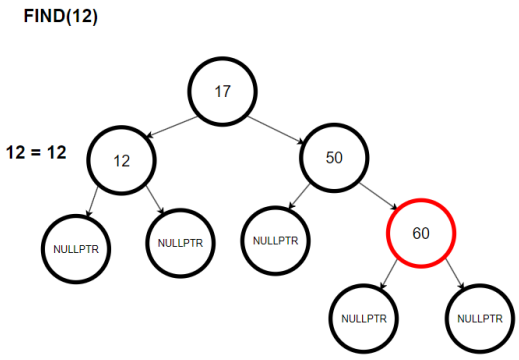

Fig. 18. Buscar 12: Se posiciona a la izquierda de 17 y se compara el dato con 12

Asimismo, en el caso de que no se encuentre el valor se muestra a continuación:

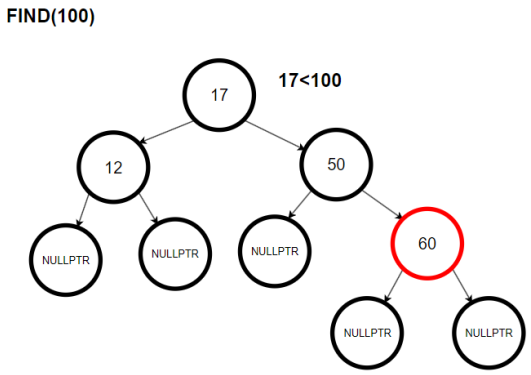

Fig. 19. Buscar 100: Se compara 17 con el dato

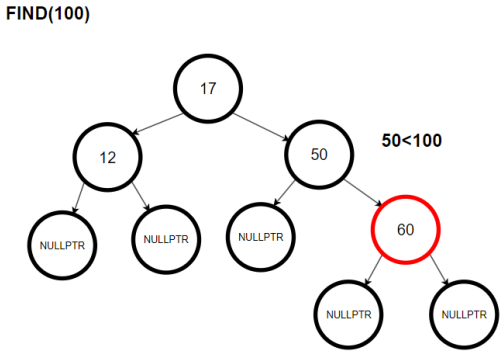

Fig. 20. Buscar 100: Se compara 50 con el dato

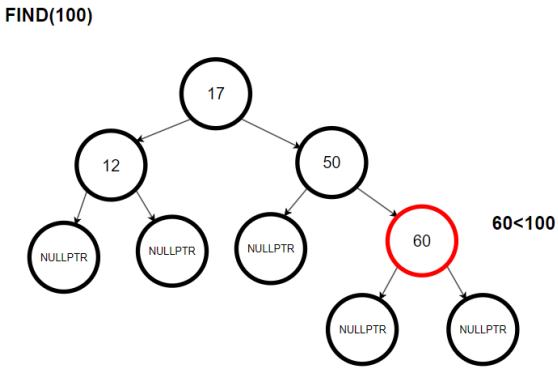

Fig. 21. Buscar 100: Se compara 60 con el dato

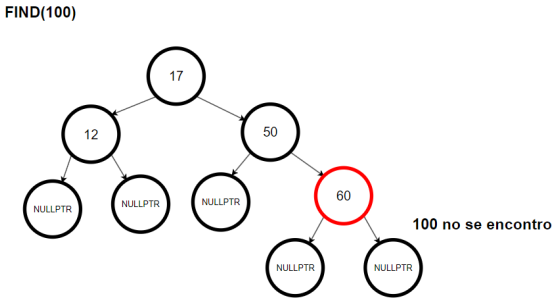

Fig. 22. Buscar 100: El valor a buscar no se encontró y se alcanzó el final del subárbol derecho

VII. Complejidad del algoritmo

i) Complejidad Computacional
La operación de búsqueda, inserción y eliminación en un árbol rojo-negro tiene una complejidad temporal de

$$O(logn). \tag{10}$$

Esto se debe a que el árbol rojo-negro siempre está equilibrado, por ende la altura del árbol es logarítmica en función del número de nodos. La propiedad de equilibrio de los árboles rojo-negro asegura que la altura del árbol nunca sea mayor que

$$2 * log(n + 1) \tag{11}$$

, donde n es el número de nodos en el árbol. Por lo tanto, las operaciones tienen una complejidad temporal de

$$O(logn) \tag{12}$$

ii) Complejidad Espacial
La complejidad espacial de un árbol rojo-negro es

$$O(n) \tag{13}$$

porque cada nodo del árbol requiere una cantidad de espacio para almacenar información adicional como el color, punteros a los hijos derecho e izquierdo, el dato y un puntero al padre. Es por ello que, es proporcional al número de nodos en el árbol.

VIII. Descripción de la herramienta visual

En este trabajo, se presenta la integración exitosa de varias tecnologías para la implementación de

la interfaz gráfica de un Red Black Tree. Raylib, una librería de C++ reconocida por su capacidad para trazar figuras geométricas y crear elementos interactivos, sirvió como fundamento principal. Esta librería, conocida por su sintaxis simple y accesible, proporcionó las herramientas esenciales para la creación de interfaces gráficas, incluso para principiantes en programación. Para llevar el proyecto a la web, se empleó WebAssembly, una herramienta que posibilita la compilación de proyectos C++ y sus respectivas librerías al entorno web, incluyendo HTML y JavaScript. Esta decisión facilitó la exportación del proyecto con Raylib a la web, manteniendo su funcionalidad y gráficos sin comprometer su desempeño. Además, se utilizó Vue, un framework de JavaScript, para la construcción del espacio de interacción web. Vue permitió la creación eficiente de componentes interactivos y la gestión fluida de la interfaz de usuario, complementando la potencia de Raylib y la capacidad de exportación de WebAssembly. Esta combinación de tecnologías permitió desarrollar una interfaz gráfica interactiva para visualizar y manipular un Red Black Tree, ofreciendo una experiencia amigable para el usuario y demostrando la viabilidad de integrar estas herramientas para proyectos similares.

IX. Conclusiones

i) Equilibrio y Eficiencia: Red-Black Trees ofrecen un equilibrio automático durante las operaciones de inserción y eliminación, lo que garantiza un tiempo de búsqueda eficiente y operaciones equilibradas.

ii) Uso en Estructuras de Datos: Red-Black Trees son ampliamente utilizados en implementaciones de bibliotecas y sistemas donde la eficiencia en la búsqueda y manipulación de datos es crucial, como en bases de datos y sistemas de archivos.

iii) Garantía de Tiempo Logarítmico:
Las operaciones básicas en un Red-Black Tree, como la búsqueda, inserción y eliminación, garantizan un tiempo logarítmico, lo que significa que el rendimiento no degrada significativamente incluso con un gran volumen de datos.

iv) Comparación con Otras Estructuras:
Comparado con otras estructuras de datos, como AVL Trees, Red-Black Trees ofrecen un rendimiento ligeramente inferior en términos de tiempo de búsqueda, pero su ventaja radica en una implementación más sencilla y menos restricciones en la rotación del árbol.

v) Aplicaciones Prácticas:
En aplicaciones prácticas, la elección entre diferentes estructuras de datos, incluidos los Red-Black Trees, puede depender de los requisitos específicos del problema y de la naturaleza de las operaciones que se realizarán con mayor frecuencia.

vi) Facilidad de Implementación:
La simplicidad relativa de implementar y mantener un Red-Black Tree en comparación con algunas otras estructuras autobalanceadas puede ser una consideración significativa en el diseño de sistemas que requieren eficiencia y flexibilidad.

vii) Consideraciones de Espacio:
Aunque el Red-Black Tree ofrecen un rendimiento eficiente, es importante tener en cuenta que pueden requerir un poco más de espacio en comparación con estructuras de datos no equilibradas debido a los bits adicionales necesarios para almacenar información de color.

Bibliografía

References

[1] Data Structures and Algorithms: Red-Black trees. (s. f.). https://www.eecs.umich.edu/courses/eecs380/ALG/red_black.html#:~:text=Definition%20of%20a%20red%2Dblack,both%20its%20children%20are%20black.

[2] GeeksforGeeks. (2023, 19 septiembre). Deletion in Red-Black Tree. https://www.geeksforgeeks.org/deletion-in-red-black-tree/

[3] Red/Black tree visualization. (s. f.). https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

[4] Besa, J., & Eterovic, Y. (2013). A concurrent red–black tree. *Journal of Parallel and Distributed Computing*, 73(4), 434-449.

[5] Kahrs, S. (2001). Red-black trees with types. *Journal of Functional Programming*, 11(4), 425-432.

[6] Xhakaj, F., & Liew, C. W. (2015, June). A new approach to teaching red-black tree. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 278-283).

[7] Hinze, R. (1999, September). Constructing red-black trees. In *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL* (Vol. 99, pp. 89-99).

[1]

[1]Queremos expresar nuestro sincero agradecimiento al Profesor Heider Sánchez por su apoyo y orientación durante la realización de este informe. Su dedicación y conocimientos han sido una inspiración para nosotros como equipo. Este trabajo está dedicado a él como muestra de nuestro profundo agradecimiento.