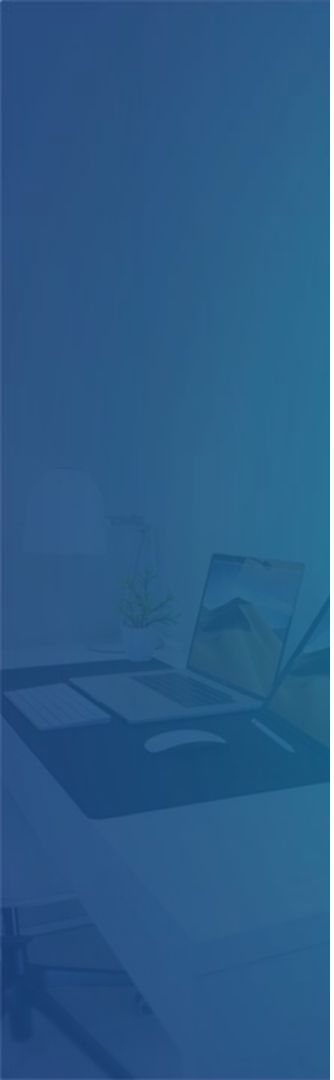
Árbol rojo-negro

Árbol binario de búsqueda autobalanceado

Ing. Miguel Matul Calderón



Introducción

Los árboles de búsqueda binaria son estructuras de datos fundamentales, pero su rendimiento puede verse afectado si el árbol se desequilibra.

Los árboles rojo-negros son un tipo de **árbol binario de búsqueda balanceado** que utiliza un conjunto de reglas para mantener el equilibrio, lo que garantiza una complejidad temporal logarítmica para operaciones como inserción, eliminación y búsqueda.

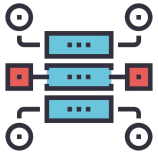
Árbol rojo-negro

Es un tipo de árbol binario, cuya esencia principal es la capacidad de **autobalancearse**. El equilibrio se logra introduciendo un atributo adicional del nodo del árbol: el "color".

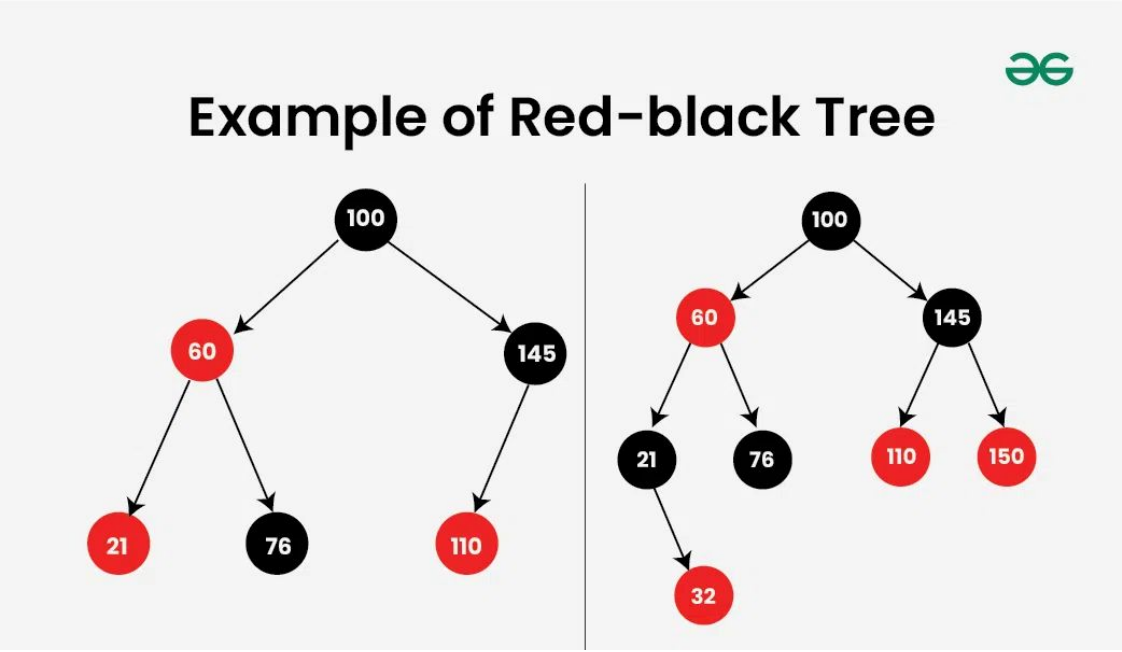
Cada nodo del árbol, además del elemento, almacena 1 bit de información sobre si el nodo es rojo o negro, y esta puede ser no sólo el color, sino también cualquier otra información que permita distinguir un tipo de nodo de otro.

Propiedades de los árboles rojo-negro

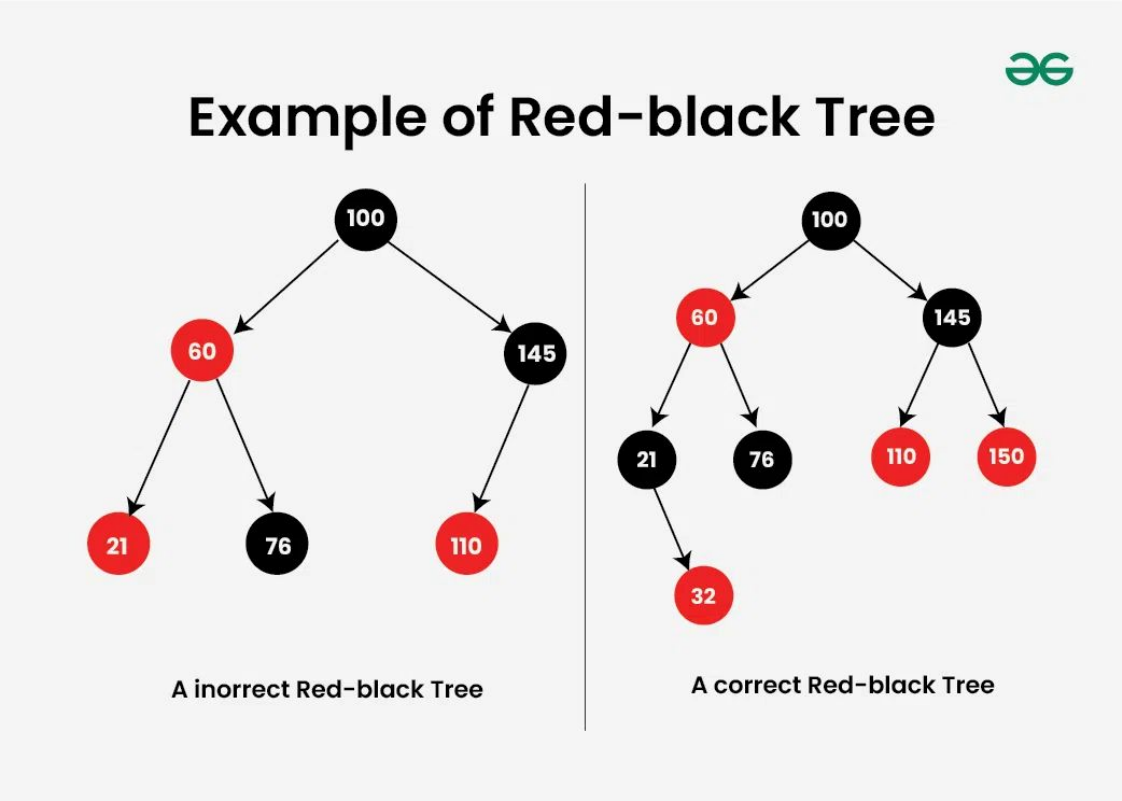
1. **Color del nodo:** cada nodo es rojo o negro.
2. **Propiedad de la raíz:** la raíz del árbol siempre es negra.
3. **Propiedad rojo:** los nodos rojos no pueden tener hijos rojos (no puede haber dos nodos rojos consecutivos en ningún camino).
4. **Propiedad negro:** cada camino desde un nodo hasta sus nodos nulos descendientes (hojas) tiene la misma cantidad de nodos negros.
5. **Propiedad de la hoja:** todas las hojas (nodos NIL) son negras.



Árbol rojo-negro



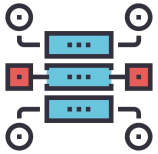
Árbol rojo-negro



Comparación con árboles AVL

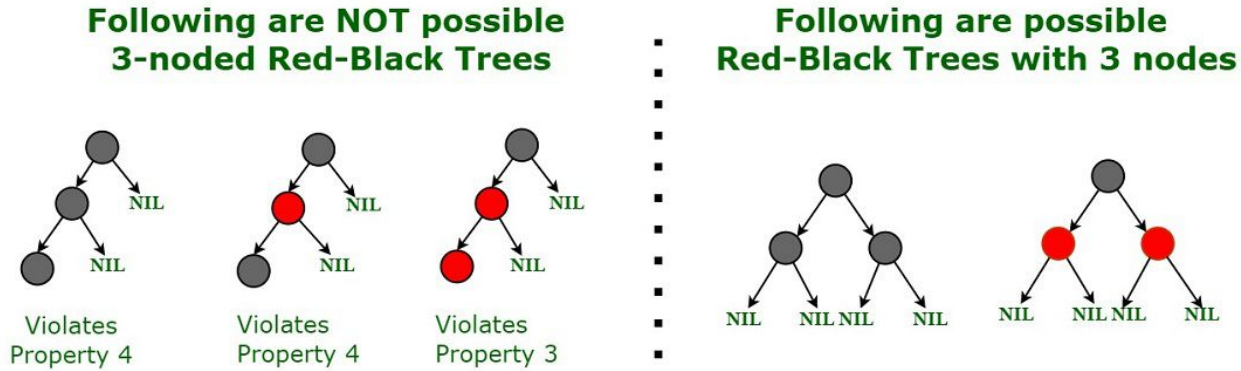
Los árboles AVL están más balanceados en comparación con los árboles rojo-negro, pero pueden provocar más rotaciones durante la inserción y la eliminación.

Por lo tanto, si su aplicación implica **inserciones y eliminaciones frecuentes**, entonces se deben preferir los árboles rojo-negros. Y si las inserciones y eliminaciones son menos frecuentes y la búsqueda es una operación más frecuente, entonces se debe preferir el árbol AVL sobre el árbol rojo-negro.



¿Cómo se asegura el balance?

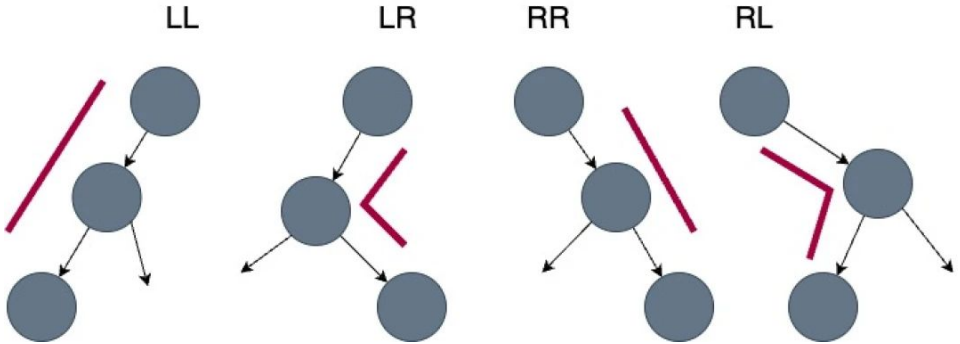
Un ejemplo sencillo para entender el balanceo es que, una cadena de 3 nodos no es posible en el árbol rojo-negro. Podemos probar cualquier combinación de colores y ver si todos ellos violan la propiedad del árbol rojo-negro.

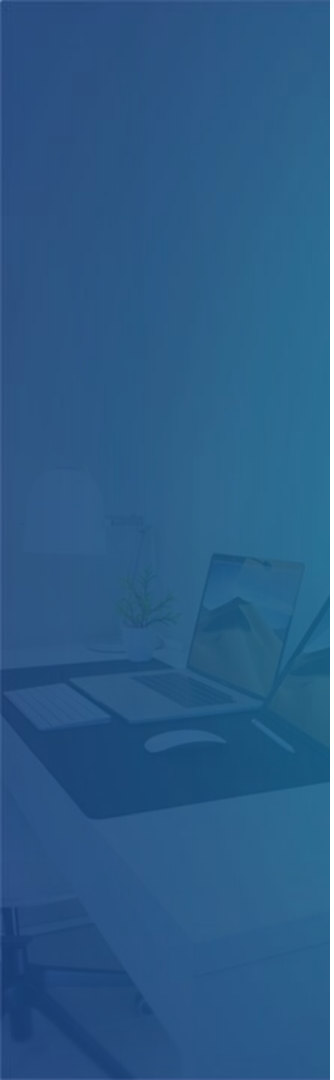


Operaciones principales

1. Buscar un elemento
2. Insertar un elemento
3. Eliminar un elemento

Las dos últimas operaciones conducen a un cambio en la estructura del árbol y, por lo tanto, su resultado puede ser un desequilibrio en el árbol.





1. Inserción

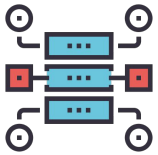
Insertar un nuevo nodo en un árbol rojo-negro involucra un **proceso de 2 pasos**:

1. Inserción estándar de un árbol binario de búsqueda
2. Corregir cualquier infracción a las propiedades de un árbol rojo-negro:
    a. Si el padre del nuevo nodo es negro, no se infringe ninguna propiedad
    b. Si el padre es rojo, podría haber una infracción en las propiedades del árbol rojo-negro.

Correcciones en la inserción

Después de insertar el nuevo nodo como nodo rojo, podríamos encontrarnos con diferentes casos dependiendo de los colores del padre y del tío del nodo (el hermano del padre):

Caso 1: El tío es rojo: cambia el color del padre y del tío a negro, y del abuelo a rojo. Luego, sube por el árbol para comprobar si hay más infracciones.

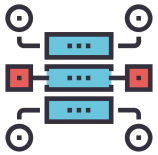


Correcciones en la inserción

Caso 2: El tío es negro:

Subcaso 2.1: El nodo es un hijo derecho: realiza una rotación a la izquierda en el padre.

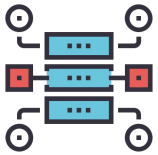
Subcaso 2.2: El nodo es un hijo izquierdo: realiza una rotación a la derecha en el abuelo y cambia el color según corresponda.

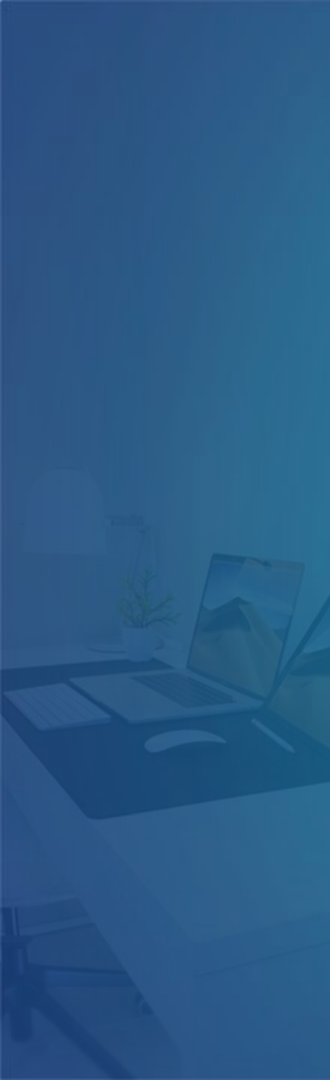


2. Búsqueda

La búsqueda de un nodo en un árbol rojo-negro es similar a la búsqueda en un árbol binario de búsqueda estándar.

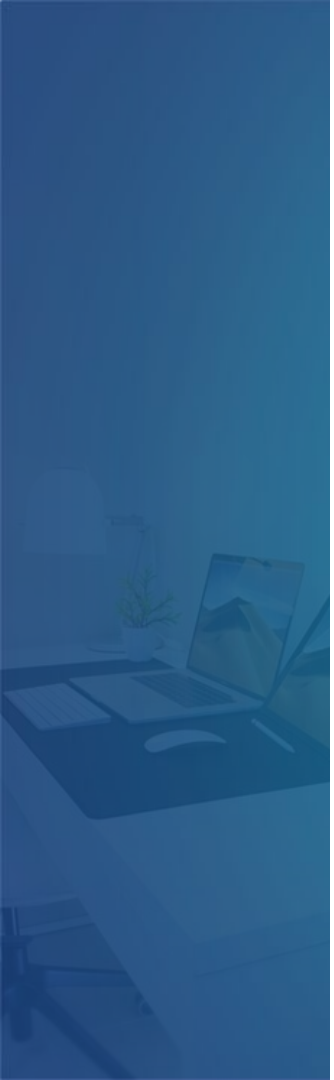
La operación de búsqueda sigue una **ruta sencilla desde la raíz hasta una hoja**, comparando el valor de destino con el valor del nodo actual y moviéndose hacia la izquierda o la derecha según corresponda.





Proceso de búsqueda

1. Comenzar la búsqueda en el nodo raíz.
2. Recorrer el árbol:
    a. Si el valor buscado es igual al valor del nodo actual, se encuentra el nodo.
    b. Si el valor buscado es menor que el valor del nodo actual, desplácese al hijo izquierdo.
    c. Si el valor buscado es mayor que el valor del nodo actual, desplácese al hijo derecho.
3. Repetir hasta que se encuentre el valor buscado o se alcance un nodo NULO (lo que indica que el valor no está presente en el árbol).



3. Eliminación

Eliminar un nodo de un árbol rojo-negro también implica un **proceso de dos pasos**:

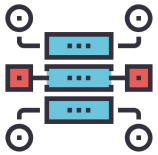
1. Realizar la eliminación estándar de un árbol binario de búsqueda
2. Corregir cualquier infracción que surja a la propiedades del árbol rojo-negro:
    a. Si se elimina un nodo negro, podría surgir una condición de “doble nodo negro”, que requiera soluciones específicas.

Correcciones en la eliminación

Caso 1: El hermano es rojo: rotar el padre y volver a colorear al hermano y al padre.

Caso 2: El hermano es negro:

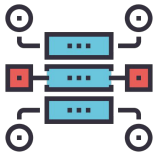
Subcaso 2.1: Los hijos del hermano son negros: volver a colorear al hermano y propagar el doble negro hacia arriba.



Correcciones en la eliminación

Subcaso 2.2: Al menos uno de los hijos del hermano es rojo:

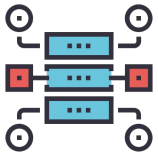
- Si el hijo lejano del hermano es rojo: realizar una rotación en el padre y el hermano, y volver a colorear de forma adecuada.
- Si el hijo cercano del hermano es rojo: rotar al hermano y a su hijo, y luego manejar el caso como se indica anteriormente.



4. Rotación

Las rotaciones son operaciones fundamentales para mantener la estructura equilibrada de un árbol rojo-negro.

Ayudan a preservar las propiedades del árbol, garantizando que el camino más largo desde la raíz hasta cualquier hoja no sea más del doble de largo que el camino más corto. Las rotaciones son de dos tipos: rotaciones hacia la izquierda **(LR)** y rotaciones hacia la derecha **(RR)**.



Left Rotation (LR)

```
procedimiento leftRotate(Nodo* x)
    Nodo* y = x.hijoDerecho
    x.hijoDerecho = y.hijoIzquierdo
    if (y.hijoIzquierdo != nulo)
        y.hijoIzquierdo.padre = x
    y.padre = x.padre
    if (x.padre == nulo)
        raiz = y
    else if (x == x.padre.hijoIzquierdo)
        x.padre.hijoIzquierdo = y
    else
        x.padre.hijoDerecho = y
    y.hijoIzquierdo = x
    x.padre = y
```

Right Rotation (LR)

```
procedimiento rightRotate(Nodo* x)
    Nodo* y = x.hijoIzquierdo
    x.hijoIzquierdo = y.hijoDerecho
    if (y.hijoDerecho != nulo)
        y.hijoDerecho.padre = x
    y.padre = x.padre
    if (x.padre == nulo)
        raiz = y
    else if (x == x.padre.hijoDerecho)
        x.padre.hijoDerecho = y
    else
        x.padre.hijoIzquierdo = y
    y.hijoDerecho = x
    x.padre = y
```

Árbol rojo-negro

Árbol binario de búsqueda autobalanceado

Ing. Miguel Matul Calderón