Árboles de Búsqueda

Jorge Mario Londoño Peláez & Varias AI

May 7, 2025

1 Árboles Binarios de Búsqueda

1.1 Definición

Un **árbol de búsqueda binaria** (Binary Search Tree (BST)) es una estructura de datos arbórea en la que cada nodo contiene un par llave-valor y se cumplen las siguientes propiedades fundamentales:

1. Cada nodo tiene como máximo dos hijos: un hijo izquierdo y un hijo derecho.
2. Una llave solo puede aparecer una vez en un árbol. Las llaves no pueden ser nulas.
3. Para cualquier nodo, el valor de su llave es *mayor* que el valor de la llave de cualquier nodo en su subárbol **izquierdo**.
4. Para cualquier nodo, el valor de su llave es *menor* que el valor de la llave de cualquier nodo en su subárbol **derecho**.

Los BST son una implementación eficiente de la **tabla de símbolos ordenada**. En esencia, un BST permite mantener un conjunto de llaves ordenadas, facilitando búsquedas, inserciones y eliminaciones eficientes.

1.1.1 Aplicaciones Prácticas

Los BST tienen numerosas aplicaciones en el mundo real:

- Implementación de diccionarios y mapas
- Sistemas de caché
- Compiladores (tablas de símbolos)
- Bases de datos (índices)
- Algoritmos de compresión
- Sistemas de archivos

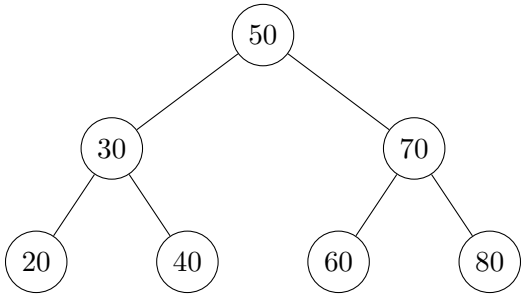
1.1.2 Construcción del BST

Desde una perspectiva de programación orientada a objetos, un BST se construye a partir de nodos. Cada nodo contiene:

- `Llave:` El valor que se utiliza para ordenar los nodos en el árbol.
- `Valor:` La información asociada a la llave.
- `Hijo Izquierdo:` Un puntero al subárbol izquierdo, que contiene nodos con llaves *menores* a la del nodo actual.
- `Hijo Derecho:` Un puntero al subárbol derecho, que contiene nodos con llaves *mayores* a la del nodo actual.

1.1.3 Ejemplo Visual

Consideremos un BST con las siguientes llaves: 50, 30, 70, 20, 40, 60, 80.



En este ejemplo:

- La raíz es 50
- Todos los nodos en el subárbol izquierdo son menores que 50
- Todos los nodos en el subárbol derecho son mayores que 50
- Cada subárbol también mantiene la propiedad de BST

1.2 Operaciones en el BST

Los BST soportan una variedad de operaciones esenciales. A continuación, se describen algunas de las más importantes, junto con su pseudocódigo y análisis de complejidad.

1.2.1 Get (Búsqueda)

La operación `get` busca un nodo con una llave específica en el árbol.

Algorithm 1 Pseudocódigo de la operación get

```
1:  function GET(node, llave)
2:      if node = null then
3:          return null
4:      end if
5:      if llave < node.llave then
6:          return GET(node.izquierdo, llave)
7:      else if llave > node.llave then
8:          return GET(node.derecho, llave)
9:      else
10:         return node.valor
11:     end if
12: end function
```

Análisis de Complejidad: En el mejor de los casos, la complejidad es $O(1)$ (si la llave está en la raíz). En el peor de los casos, la complejidad es $O(h)$, donde h es la altura del árbol. En un árbol balanceado, h es $O(\log n)$, mientras que en un árbol no balanceado, h puede ser $O(n)$.

1.2.2 Put (Inserción)

La operación `put` inserta un nuevo nodo con una llave y valor dados en el árbol. Es esencial para construir y modificar el BST.

Algorithm 2 Pseudocódigo de la operación put

```
1:  function PUT(node, llave, valor)
2:      if node = null then
3:          return NUEVONODO(llave, valor)
4:      end if
5:      if llave < node.llave then
6:          node.izquierdo ← PUT(node.izquierdo, llave, valor)
7:      else if llave > node.llave then
8:          node.derecho ← PUT(node.derecho, llave, valor)
9:      else
10:         node.valor ← valor
11:     end if
12:     return node
13: end function
```

Análisis de Complejidad: Similar a `get`, la complejidad es $O(h)$, donde h es la altura del árbol.

1.2.3 Delete (Eliminación)

La operación `delete` elimina un nodo con una llave específica del árbol. Esta operación es más compleja que `get` y `put`, ya que requiere reorganizar el árbol para mantener las propiedades del BST. Para su implementación se utiliza una regla sencilla como reemplazar el nodo a eliminar por su sucesor o predecesor.

Operación de borrado del mínimo La operación de borrado del mínimo en un BST elimina el nodo con la llave más pequeña del árbol. Dado que el nodo con la llave más pequeña se encuentra siempre en el extremo izquierdo del árbol (o subárbol), la operación consiste en descender recursivamente por el subárbol izquierdo hasta encontrar el nodo mínimo. Una vez encontrado, este nodo se elimina y se reemplaza por su hijo derecho (que puede ser nulo). Este proceso mantiene la propiedad de BST, ya que todos los nodos en el subárbol derecho del nodo eliminado son mayores que cualquier otro nodo en el árbol restante.

Análisis de Complejidad: La complejidad de esta operación es $O(h)$, donde h es la altura del árbol, ya que en el peor de los casos debe recorrerse la altura completa del árbol para encontrar el mínimo.

Operación de borrado de nodos en general El borrado de un nodo en general es una operación más compleja que el borrado del mínimo, ya que requiere considerar varios casos para mantener las propiedades del BST. El proceso general consta de los siguientes pasos:

1. **Buscar el nodo a borrar:** Se busca el nodo con la llave que se desea eliminar. Sea t la referencia a este nodo.

Algorithm 3 Pseudocódigo de la operación deleteMin

```
1: function DeleteMin(node)
2:     if node = null then
3:         return null
4:     end if
5:     if node.izquierdo = null then
6:         return node.derecho
7:     end if
8:     node.izquierdo ← DeleteMin(node.izquierdo)
9:     return node
10: end function
```

2. **Caso 1: El nodo a borrar no tiene hijos:** Simplemente se elimina el nodo. Esto se reduce a establecer el puntero del padre a nulo.
3. **Caso 2: El nodo a borrar tiene un solo hijo:** Se reemplaza el nodo por su único hijo. El hijo puede ser izquierdo o derecho.
4. **Caso 3: El nodo a borrar tiene dos hijos:** Se encuentra el sucesor de t, que es el nodo con la llave más pequeña en el subárbol derecho de t. Sea x el sucesor de t, es decir, $x = \text{Min}(t.derecho)$. El nodo x reemplazará a t en el árbol. Luego, se actualizan los hijos de x:
   - El hijo derecho de x se obtiene eliminando el mínimo del subárbol derecho de t: $x.derecho = \text{DeleteMin}(t.derecho)$.
   - El hijo izquierdo de x es el hijo izquierdo de t: $x.izquierdo = t.izquierdo$.
5. **Actualizar el tamaño:** Se actualiza el tamaño de los nodos en el camino de retorno de las llamadas recursivas.

Análisis de Complejidad: La complejidad de esta operación es $O(h)$, donde h es la altura del árbol. En el peor de los casos, la búsqueda del nodo a eliminar y la búsqueda de su sucesor (o predecesor) pueden requerir recorrer la altura completa del árbol.

1.2.4 Otras operaciones

Además de las operaciones básicas, los BST, como tablas de símbolos ordenadas soportan otras operaciones:

- **Mínimo:** Encuentra el nodo con la llave mínima.
- **Máximo:** Encuentra el nodo con la llave máxima.
- **Piso (Floor):** Encuentra el nodo con la llave más grande menor o igual a una llave dada.
- **Techo (Ceiling):** Encuentra el nodo con la llave más pequeña mayor o igual a una llave dada.
- **Rango (Rank):** Determina cuántas llaves en el árbol son menores que una llave dada.
- **Seleccionar (Select):** Encuentra la llave que tiene un rango específico.

La complejidad de estas operaciones también es $O(h)$.

Algorithm 4 Pseudocódigo de la operación delete

```
1: function DELETE(node, llave)
2:     if node = null then
3:         return null
4:     end if
5:     if llave < node.llave then
6:         node.izquierdo ← DELETE(node.izquierdo, llave)
7:     else if llave > node.llave then
8:         node.derecho ← DELETE(node.derecho, llave)
9:     else
10:        if node.derecho = null then
11:            return node.izquierdo
12:        end if
13:        if node.izquierdo = null then
14:            return node.derecho
15:        end if
16:        t ← node
17:        node ← MIN(t.derecho)
18:        node.derecho ← DELETEMIN(t.derecho)
19:        node.izquierdo ← t.izquierdo
20:    end if
21:    return node
22: end function
```

1.3 Recorrido de Árboles

El recorrido de un árbol implica visitar cada nodo del árbol exactamente una vez. Hay varias formas de recorrer un BST, cada una con sus propias características y aplicaciones.

1.3.1 Preorden (Raíz-Izquierda-Derecha)

En el recorrido en preorden:

1. Se visita el nodo raíz
2. Se recorre el subárbol izquierdo en preorden
3. Se recorre el subárbol derecho en preorden

Aplicaciones:

- Creación de una copia del árbol
- Serialización del árbol
- Evaluación de expresiones prefijas

1.3.2 Inorden (Izquierda-Raíz-Derecha)

En el recorrido inorden:

1. Se recorre el subárbol izquierdo en inorden
2. Se visita el nodo raíz
3. Se recorre el subárbol derecho en inorden

Aplicaciones:

- Obtener elementos en orden
- Validación de BST
- Búsquedas por rango

1.3.3 Postorden (Izquierda-Derecha-Raíz)

En el recorrido postorden:

1. Se recorre el subárbol izquierdo en postorden
2. Se recorre el subárbol derecho en postorden
3. Se visita el nodo raíz

Aplicaciones:

- Eliminación de árboles
- Evaluación de expresiones postfijas
- Cálculo de alturas y tamaños

1.4 Consideraciones de Implementación

1.4.1 Mejores Prácticas

- Mantener el árbol balanceado para optimizar el rendimiento
- Implementar validación de propiedades del BST
- Manejar casos especiales (árbol vacío, nodos nulos)
- Implementar iteradores para recorridos eficientes

1.4.2 Pitfalls Comunes

- No verificar propiedades del BST durante inserciones
- Olvidar actualizar punteros durante eliminaciones
- No manejar casos de árbol degenerado
- Ignorar el balance del árbol

1.5 Optimizaciones

1.5.1 Árboles Balanceados

Para mantener un rendimiento óptimo, es crucial mantener el árbol balanceado. Algunas implementaciones comunes incluyen:

- Árboles AVL
- Árboles Rojo-Negro
- Árboles B

1.5.2 Casos de Uso y Selección

La elección entre diferentes tipos de árboles balanceados depende de:

- Frecuencia de operaciones de lectura vs. escritura
- Requisitos de memoria
- Necesidad de operaciones concurrentes
- Patrones de acceso a los datos

2 Ejercicios y Práctica

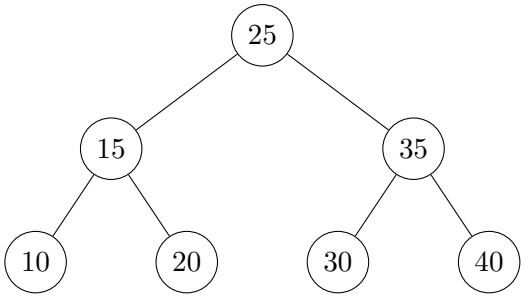
2.1 Ejercicios Teóricos

1. **Propiedades del BST**
   (a) Demuestra que en un BST, el recorrido inorden siempre produce una secuencia ordenada de llaves.
   (b) ¿Por qué es importante que las llaves en un BST sean únicas?
   (c) Explica por qué un árbol binario completo no necesariamente es un BST.
2. **Análisis de Complejidad**
   (a) ¿Cuál es la complejidad de buscar un elemento en un BST degenerado (que se asemeja a una lista enlazada)?
   (b) Explica por qué la complejidad de las operaciones básicas en un BST balanceado es $O(\log n)$.
   (c) ¿Cuál es la complejidad de recorrer todo el árbol en cualquiera de los tres órdenes (preorden, inorden, postorden)?
3. **Operaciones**
   (a) Describe el proceso de eliminar un nodo con dos hijos en un BST.
   (b) ¿Por qué es importante mantener el árbol balanceado?
   (c) Explica la diferencia entre el sucesor y el predecesor de un nodo en un BST.

2.2 Ejercicios Prácticos

2.2.1 Construcción de BST

1. Construye un BST insertando los siguientes números en orden: 45, 30, 60, 20, 40, 50, 70.
   (a) Dibuja el árbol resultante
   (b) Realiza un recorrido inorden y escribe la secuencia resultante
   (c) Encuentra el sucesor de 45
   (d) Encuentra el predecesor de 60
2. Dado el siguiente BST:



   (a) Elimina el nodo 15 y dibuja el árbol resultante
   (b) Elimina el nodo 25 y dibuja el árbol resultante
   (c) Inserta el nodo 22 y dibuja el árbol resultante

2.2.2 Implementación

1. Implementa las siguientes operaciones en tu lenguaje de programación preferido:
   (a) Inserción de un nodo
   (b) Búsqueda de un nodo
   (c) Eliminación de un nodo
   (d) Recorrido inorden
2. Extiende la implementación anterior para incluir:
   (a) Cálculo de la altura del árbol
   (b) Verificación de si el árbol está balanceado
   (c) Búsqueda por rango
   (d) Encontrar el k-ésimo elemento más pequeño

2.3 Problemas de Desafío

1. **Validación de BST**
   (a) Implementa una función que verifique si un árbol binario dado es un BST válido.
   (b) Considera los casos especiales: árbol vacío, árbol con un solo nodo, árbol degenerado.

(c) Optimiza tu solución para que sea eficiente en términos de tiempo y espacio.

2. **Balanceo de BST**

   (a) Implementa una función que balancee un BST dado.

   (b) Considera diferentes estrategias de balanceo (AVL, Rojo-Negro).

   (c) Analiza la complejidad de tu solución.

3. **Operaciones Avanzadas**

   (a) Implementa una operación para encontrar el ancestro común más bajo (LCA) de dos nodos.

   (b) Implementa una operación para encontrar la suma de todos los nodos en un rango dado.

   (c) Implementa una operación para convertir un BST en una lista doblemente enlazada ordenada.

2.4 Soluciones y Pistas

2.4.1 Pistas para los Ejercicios Teóricos

1. Para demostrar que el recorrido inorden produce una secuencia ordenada, considera la propiedad de BST que establece que todos los elementos en el subárbol izquierdo son menores que la raíz.

2. La complejidad en un árbol degenerado es $O(n)$ porque el árbol se convierte esencialmente en una lista enlazada.

3. Al eliminar un nodo con dos hijos, considera usar el sucesor o predecesor para mantener las propiedades del BST.

2.4.2 Pistas para los Ejercicios Prácticos

1. Al construir el BST, recuerda que cada nuevo nodo debe mantener la propiedad de que todos los elementos a su izquierda son menores y todos los elementos a su derecha son mayores.

2. Para la implementación, considera usar una estructura de nodo con punteros a los hijos izquierdo y derecho.

3. Al implementar la eliminación, considera los tres casos: nodo sin hijos, nodo con un hijo, y nodo con dos hijos.

2.4.3 Pistas para los Problemas de Desafío

1. Para la validación de BST, considera usar un recorrido inorden y verificar que cada elemento sea mayor que el anterior.

2. Para el balanceo, considera usar rotaciones y mantener información de balance en cada nodo.

3. Para el LCA, considera que el ancestro común más bajo debe estar entre los dos nodos en el recorrido inorden.

3 Árboles de búsqueda 2-3

3.1 Definición

Los árboles de búsqueda 2-3 son una estructura de datos arbórea diseñada para mantener el árbol balanceado, garantizando un rendimiento de búsqueda en el peor de los casos de $O(\log n)$, donde n es el número de elementos en el árbol. A diferencia de los árboles binarios de búsqueda, los árboles 2-3 permiten que cada nodo tenga dos o tres hijos.

Definición formal:
Un árbol 2-3 es un árbol que satisface las siguientes propiedades:

- Se compone de dos tipos de nodos: 2-nodes y 3-nodes.
- Los 2-nodes tienen 2 hijos y un par llave-valor.
- Los 3-nodes tienen 3 hijos y dos pares llave-valor.
- Todos los nodos hoja están al mismo nivel, lo que garantiza que el árbol esté balanceado.
- Los valores en los nodos se mantienen en orden de búsqueda, similar a los árboles binarios de búsqueda.

3.2 Estructura Visual

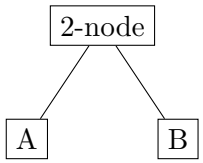


Figure 1: Ejemplo de un 2-node con dos hijos

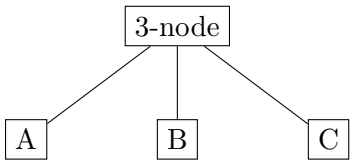


Figure 2: Ejemplo de un 3-node con tres hijos

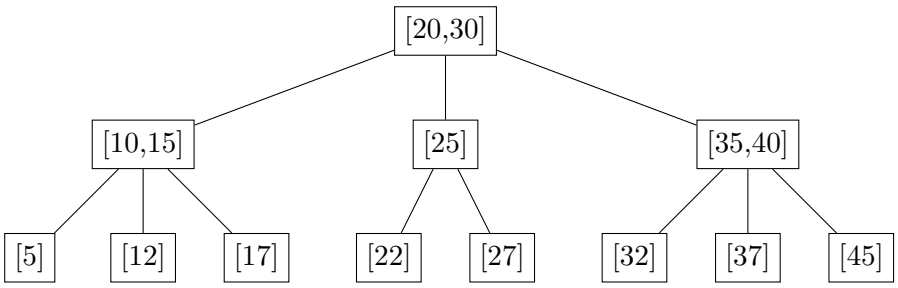


Figure 3: Ejemplo de un árbol 2-3 completo

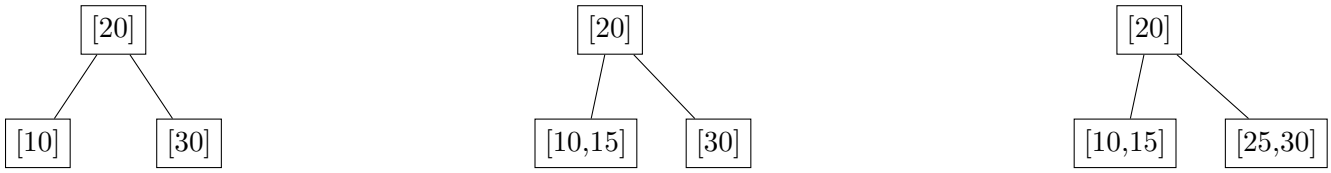


Figure 4: Pasos de inserción en un árbol 2-3

3.3 Búsqueda en árboles 2-3

El algoritmo de búsqueda en un árbol 2-3 es una extensión de la búsqueda en un árbol binario de búsqueda. Comienza en la raíz y compara el valor buscado con los valores almacenados en el nodo actual.

- Si la llave buscada es igual a alguno de los valores en el nodo, la búsqueda se completa.
- Si la llave buscada es menor que el valor más pequeño en el nodo, la búsqueda continúa en el hijo más a la izquierda.
- Si la llave buscada está entre las llaves en el nodo, la búsqueda continúa en el hijo central.
- Si la llave buscada es mayor que la llave mayor, la búsqueda continúa en el hijo más a la derecha.

La búsqueda continúa hasta que se encuentra el valor o se llega a un nodo hoja.

3.4 Inserción en árboles 2-3

La inserción en un árbol 2-3 también debe mantener la propiedad de equilibrio del árbol. El proceso general es el siguiente:

1. **Buscar la posición de inserción:** Se realiza una búsqueda para encontrar el nodo hoja donde se debe insertar el nuevo valor.
2. **Insertar en el nodo hoja:**
   - Si el nodo hoja tiene espacio (es decir, es un nodo 2), el nuevo valor se inserta en el nodo, manteniendo el orden.
   - Si el nodo hoja está lleno (es decir, es un nodo 3), se divide el nodo en dos nodos, y el valor medio se promueve al nodo padre.
3. **Propagar la división (si es necesario):** Si la promoción del valor medio hace que el nodo padre se llene, el proceso de división se repite en el nodo padre. Esto puede propagarse hasta la raíz del árbol. Si la raíz se divide, se crea una nueva raíz, y la altura del árbol aumenta en uno.

3.5 Eliminación en árboles 2-3

La eliminación en árboles 2-3 es más compleja que la inserción y requiere varios casos especiales:

1. **Eliminación en un nodo hoja:**
   - Si el nodo es un 3-node, simplemente se elimina la llave.
   - Si el nodo es un 2-node, se debe realizar una redistribución o fusión con nodos hermanos.
2. **Eliminación en un nodo interno:**
   - Se reemplaza la llave a eliminar con su predecesor o sucesor.
   - Se elimina el predecesor o sucesor de su posición original.
3. **Redistribución y fusión:**
   - Si un nodo se queda sin llaves, se intenta redistribuir llaves desde nodos hermanos.
   - Si la redistribución no es posible, se fusiona con un nodo hermano.

3.6 Ventajas y Desventajas

Ventajas:

- Garantiza un rendimiento de $O(\log n)$ para todas las operaciones.
- Mantiene el árbol balanceado automáticamente.
- No requiere rotaciones como los árboles rojo-negros.

Desventajas:

- Implementación más compleja que los árboles binarios de búsqueda.
- Mayor overhead de memoria por nodo.
- Operaciones de inserción y eliminación más complejas.

3.7 Pseudocódigo de Operaciones Básicas

3.8 Complejidad Algorítmica

Las operaciones de búsqueda, inserción y eliminación en árboles 2-3 tienen una complejidad algorítmica de $O(\log n)$ en el peor de los casos, debido a la propiedad de equilibrio del árbol. Esto los convierte en una opción eficiente para mantener conjuntos de datos ordenados donde el rendimiento de la búsqueda es crítico.

3.9 Relación con Árboles Rojo-Negros

Los árboles 2-3 son conceptualmente más simples que los árboles rojo-negros, pero su implementación directa es más compleja. Por esta razón, en la práctica se suelen implementar árboles rojo-negros, que son una representación binaria de los árboles 2-3. Los árboles rojo-negros mantienen las mismas propiedades de equilibrio pero con una implementación más eficiente en términos de memoria y operaciones.

Algorithm 5 Búsqueda en Árbol 2-3

```
1: function BUSCAR(nodo, llave)
2:     if nodo es nulo then
3:         return No encontrado
4:     end if
5:     if llave está en nodo then
6:         return Encontrado
7:     else if nodo es 2-node then
8:         if llave ¡ nodo.llave then
9:             return Buscar(nodo.izquierdo, llave)
10:        else
11:            return Buscar(nodo.derecho, llave)
12:        end if
13:    else
14:        if llave ¡ nodo.llave1 then
15:            return Buscar(nodo.izquierdo, llave)
16:        else if llave ¡ nodo.llave2 then
17:            return Buscar(nodo.central, llave)
18:        else
19:            return Buscar(nodo.derecho, llave)
20:        end if
21:    end if
22: end function
```

4 Árboles Rojo-Negros

4.1 Definición

Los árboles rojo-negros son una clase de árboles de búsqueda auto-balanceables. Además de los atributos típicos de un árbol binario de búsqueda, cada nodo en un árbol rojo-negro tiene un atributo de color, que puede ser rojo o negro. Estos colores se utilizan para asegurar que el árbol permanezca balanceado durante las inserciones y eliminaciones.

Definición formal:
Un árbol rojo-negro es un árbol binario de búsqueda que satisface las siguientes propiedades:

1. Cada nodo es rojo o negro.
2. La raíz es negra.
3. Si un nodo es rojo, entonces sus dos hijos son negros.
4. Todos los caminos desde cualquier nodo a todas sus hojas descendientes contienen el mismo número de nodos negros.

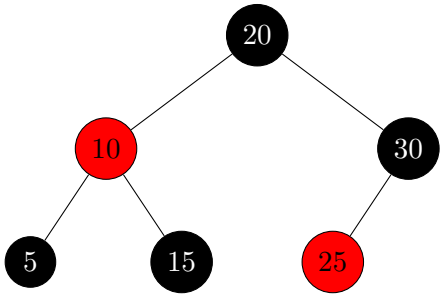


Figure 5: Ejemplo de un árbol rojo-negro válido. Los nodos negros se muestran con fondo negro y texto blanco, los rojos con fondo rojo.

La propiedad clave de los árboles rojo-negros es que garantizan que la altura del árbol sea a lo sumo $2\lg(n+1)$, donde n es el número de nodos internos (no hoja) en el árbol. Esto asegura que las operaciones de búsqueda, inserción y eliminación puedan realizarse en tiempo $O(\lg n)$.

4.2 Relación con los Árboles 2-3

Existe una estrecha relación entre los árboles rojo-negros y los árboles 2-3. Un árbol rojo-negro puede ser visto como una representación binaria de un árbol 2-3. En un árbol 2-3:

- Un nodo 2 tiene un hijo izquierdo y un hijo derecho. En el árbol rojo-negro, esto se representa como un nodo negro con dos hijos negros.
- Un nodo 3 tiene un hijo izquierdo, un hijo central y un hijo derecho. En el árbol rojo-negro, esto se representa como un nodo negro con un hijo rojo (ya sea a la izquierda o a la derecha) y un hijo negro. El nodo rojo y su padre negro representan el nodo 3.

Esta correspondencia asegura que la altura de un árbol rojo-negro sea logarítmica, ya que simula la estructura balanceada de un árbol 2-3.

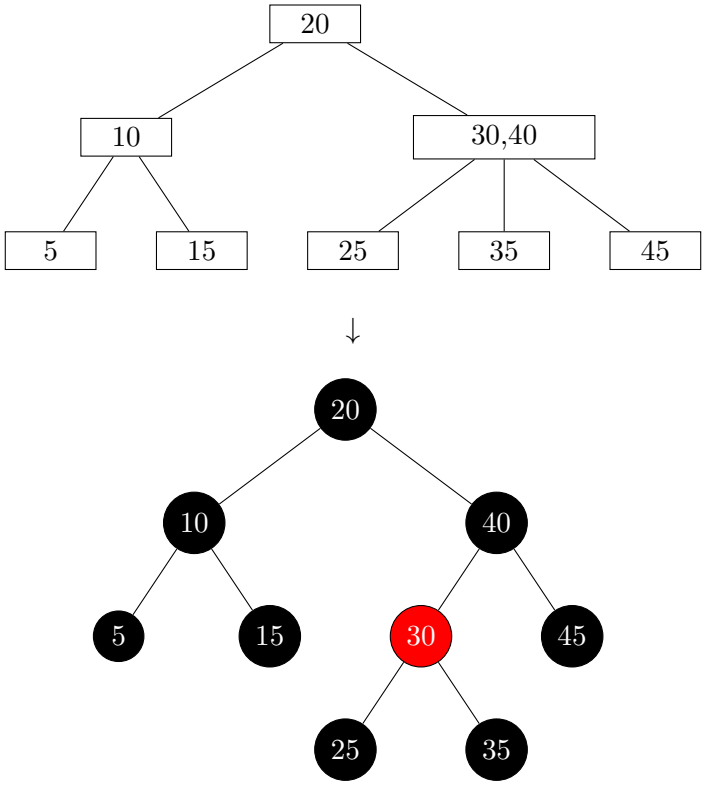


Figure 6: Transformación de un árbol 2-3 a su representación en árbol rojo-negro. El nodo 3-nodo (30,40) se representa como un nodo negro (30) con un hijo negro (35) y un nodo rojo (40).

4.3 Implementación del Árbol Rojo-Negro

La implementación de un árbol rojo-negro implica mantener la estructura del árbol binario de búsqueda y las propiedades de coloración. Las operaciones clave son la inserción y la eliminación, que deben preservar las propiedades del árbol rojo-negro. Esto se logra mediante rotaciones y recoloreos.

4.3.1 Representación del Color

El color de un nodo se representa generalmente con un valor booleano: `True` para rojo y `False` para negro. En la implementación, es común usar constantes para mayor claridad:

```
private static final boolean RED = true;
private static final boolean BLACK = false;
```

4.3.2 Rotaciones

Las rotaciones son operaciones que modifican la estructura local del árbol sin afectar el orden de las llaves. Hay dos tipos de rotaciones:

- **Rotación a la izquierda:** Permite mover el hijo derecho hacia la posición de su padre.
- **Rotación a la derecha:** Permite mover el hijo izquierdo hacia la posición de su padre.

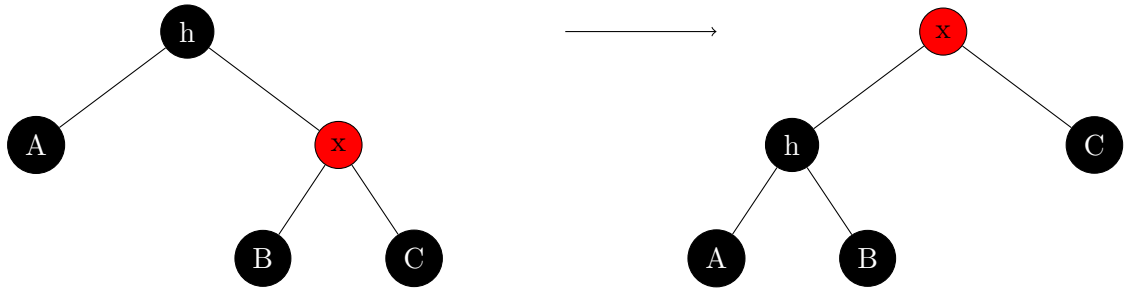


Figure 7: Ejemplo de rotación a la izquierda.

Algorithm 6 Pseudocódigo de Rotación a la Izquierda

```
1: function LeftRotate(Node h)
2:     x ← h.right
3:     h.right ← x.left
4:     x.left ← h
5:     x.color ← h.color
6:     h.color ← RED
7:     x.n ← h.n
8:     h.n ← size(h.left) + size(h.right) + 1
9:     return x
10: end function
```

4.3.3 Inserción

La inserción de un nuevo nodo en un árbol rojo-negro sigue estos pasos:

1. Insertar el nuevo nodo como en un BST normal, coloreándolo de rojo.
2. Verificar y corregir las propiedades del árbol rojo-negro:
   - Si el padre es negro, no se necesita hacer nada.
   - Si el padre es rojo, se necesitan rotaciones y recoloreos.

Ejemplo de inserción:

Algorithm 7 Pseudocódigo de Rotación a la Derecha

```
1: function RightRotate(Node h)
2:     x ← h.left
3:     h.left ← x.right
4:     x.right ← h
5:     x.color ← h.color
6:     h.color ← RED
7:     x.n ← h.n
8:     h.n ← size(h.left) + size(h.right) + 1
9:     return x
10: end function
```

Algorithm 8 Pseudocódigo de la operación put

```
1: function PUT(llave, valor)
2:     root ← PUT(root, llave, valor)
3:     root.color ← BLACK
4: end function
5: function PUT(Node h, llave, valor)
6:     if h == null then
7:         return new NODE(key, val, RED, 1);
8:     end if
9:     cmp ← key.compareTo(h.key)
10:    if cmp < 0 then
11:        h.left ← PUT(h.left, llave, valor);
12:    else if cmp > 0 then
13:        h.right ← PUT(h.right, llave, valor);
14:    else
15:        h.val ← valor;
16:    end if
17:    if isRed(h.right) and !isRed(h.left) then
18:        h = ROTATELEFT(h)
19:    end if
20:    if isRed(h.left) and isRed(h.left.left) then
21:        h = ROTATERIGHT(h)
22:    end if
23:    if isRed(h.left) and isRed(h.right) then
24:        FLIPCOLORS(h)
25:    end if
26:    h.N = size(h.left) + size(h.right) + 1
27:    return h
28: end function
```

4.3.4 Eliminación

La eliminación en un árbol rojo-negro es más compleja que la inserción. El proceso sigue estos pasos:

1. Realizar la eliminación como en un BST normal.
2. Si el nodo eliminado era rojo, no se necesitan correcciones.
3. Si el nodo eliminado era negro, se necesitan correcciones para mantener las propiedades:
    - Si el hermano es rojo, realizar una rotación y recolorear.
    - Si el hermano es negro con dos hijos negros, recolorear.
    - Si el hermano es negro con al menos un hijo rojo, realizar rotaciones y recoloreos.

4.4 Eficiencia de Operaciones en Árboles Rojo-Negros

Las operaciones de búsqueda, inserción y eliminación en árboles rojo-negros tienen una complejidad temporal de $O(\log n)$ en el peor caso. Esto se debe a que la altura del árbol se mantiene logarítmica mediante las propiedades de coloración y las operaciones de reestructuración (rotaciones y recoloreos).

4.5 Consideraciones Prácticas

- **Ventajas:**
    - Garantiza operaciones en tiempo $O(\log n)$.
    - Menos rotaciones que los árboles AVL.
    - Buena para implementaciones en memoria.
- **Desventajas:**
    - Mayor complejidad de implementación.
    - Overhead de memoria por el bit de color.
    - Más rotaciones que los árboles B.
- **Casos de Uso:**
    - Implementación de mapas y conjuntos ordenados.
    - Cuando se necesita garantizar tiempo $O(\log n)$.
    - Cuando la memoria no es una restricción crítica.

4.6 Ejercicios Recomendados

1. Implementar las operaciones básicas de un árbol rojo-negro.
2. Demostrar que la altura es a lo sumo $2 \lg(n + 1)$.
3. Analizar los casos de rotación y recoloreo en la inserción.
4. Comparar el rendimiento con otros árboles balanceados.

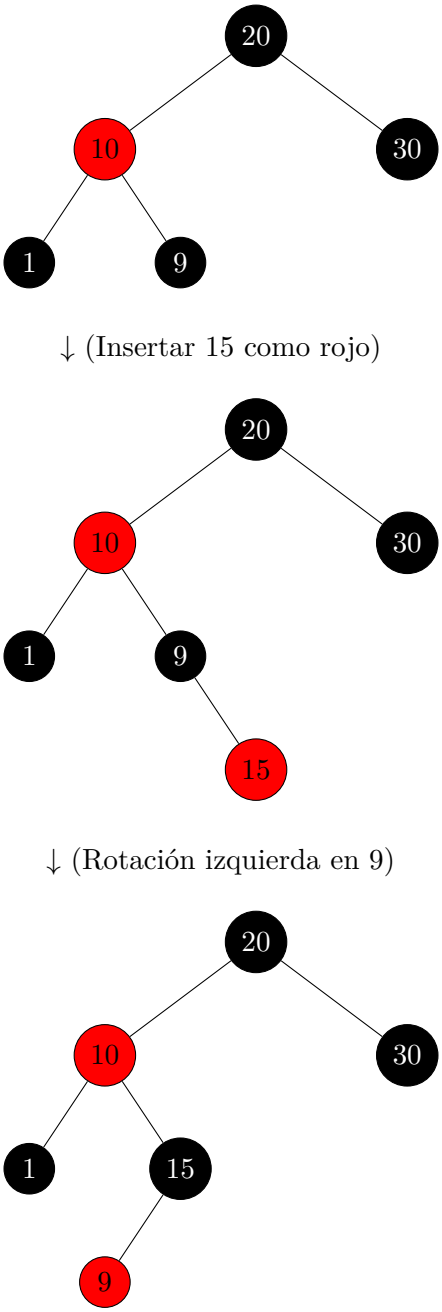


Figure 8: Ejemplo de inserción del valor 15 en un árbol rojo-negro. Se muestra el proceso de inserción y las rotaciones izquierdas necesarias para mantener las propiedades del árbol.

5 Otros tipos de árboles de búsqueda

Esta sección presenta una breve introducción a diferentes tipos de árboles de búsqueda balanceados. Cada tipo de árbol tiene sus propias características y casos de uso específicos. Se incluyen enlaces a recursos externos para un estudio más profundo de cada estructura.

5.1 AVL Tree

Los árboles AVL son árboles de búsqueda binarios auto-balanceables donde las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren como máximo en uno. Esto asegura un árbol balanceado, lo que lleva a una complejidad temporal de $O(\log n)$ para las operaciones de búsqueda, inserción y eliminación.

- AVL Trees @ w3schools
- Wikipedia
- AVL Tree Data Structure @ Geeks for Geeks
- AVL Tree @ Programiz
- AVL Tree Visualization

5.2 B-Tree

Los árboles B son estructuras de datos de árbol auto-balanceables que son particularmente adecuadas para sistemas de almacenamiento que leen y escriben bloques de datos relativamente grandes, como bases de datos y sistemas de archivos. Están optimizados para el acceso al disco y pueden tener un alto factor de ramificación para minimizar la cantidad de accesos al disco necesarios para una operación.

- Introduction of B+ Tree @ Geeks for Geeks
- Wikipedia
- B-Tree Visualization

5.3 B+ Tree

Los árboles B+ son similares a los árboles B, pero con la diferencia clave de que todos los datos se almacenan en los nodos hoja. Los nodos internos solo almacenan claves, que actúan como enrutadores para guiar el proceso de búsqueda. Los árboles B+ se utilizan comúnmente en sistemas de bases de datos para la indexación, ya que proporcionan consultas de rango eficientes y acceso secuencial a los datos.

- Introduction of B+ Tree @ Geeks for Geeks
- The Difference Between B-trees and B+trees @ Baeldung
- Wikipedia
- B+ Tree Visualization