Árboles de Búsqueda

Jorge Mario Londoño Peláez & Varias AI

April 25, 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:
Definición:

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 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.

1.1.2 Ejemplo

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.
Pseudocódigo:

```
1: function GET(node, llave)
2:     if node = null then
3:         return null
4:     end if
```

```
    if llave < node.llave then
        return GET(node.izquierdo, llave)
    else if llave > node.llave then
        return GET(node.derecho, llave)
    else
        return node.valor
    end if
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.

Pseudocódigo:

```
function PUT(node, llave, valor)
    if node = null then
        return NUEVONODO(llave, valor)
    end if
    if llave < node.llave then
        node.izquierdo ← PUT(node.izquierdo, llave, valor)
    else if llave > node.llave then
        node.derecho ← PUT(node.derecho, llave, valor)
    else
        node.valor ← valor
    end if
    return node
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.

Pseudocódigo:

```
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
```

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.
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.

Pseudocódigo:

```
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
```

```
        if node.derecho = null then
            return node.izquierdo
        end if
        if node.izquierdo = null then
            return node.derecho
        end if
        t ← node
        node ← Min(t.derecho)
        node.derecho ← DeleteMin(t.derecho)
        node.izquierdo ← t.izquierdo
    end if
    return node
end function
```

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 $O(h)$.

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

En el recorrido en preorden, primero se visita el nodo raíz, luego se recorre el subárbol izquierdo y, finalmente, se recorre el subárbol derecho.

1.3.2 Inorden

En el recorrido inorden, primero se recorre el subárbol izquierdo, luego se visita el nodo raíz y, finalmente, se recorre el subárbol derecho. Este tipo de recorrido produce una lista ordenada de las llaves del árbol.

Pseudocódigo:

```
1: function INORDER(node)
2:     if node ≠ null then
3:         INORDER(node.left)
4:         Visitar node
5:         INORDER(node.right)
6:     end if
7: end function
```

Análisis de Complejidad: El recorrido inorden visita cada nodo del árbol una vez, por lo que su complejidad es $O(n)$, donde n es el número de nodos en el árbol.

1.3.3 Postorden

En el recorrido postorden, primero se recorre el subárbol izquierdo, luego se recorre el subárbol derecho y, finalmente, se visita el nodo raíz.

1.3.4 Búsquedas por Rangos de Llaves

Búsquedas por Rangos: El recorrido inorden es especialmente útil para realizar búsquedas por rangos de llaves en un BST. Dado un rango de llaves $[a, b]$, se puede recorrer el árbol en inorden y seleccionar solo los nodos cuyas llaves estén dentro de este rango. Si durante el recorrido se encuentra un hijo izquierdo con llave menor que a, se puede omitir el recorrido de ese subárbol, y si se encuentra un hijo derecho con llave mayor que b, se puede omitir el recorrido de ese subárbol también. Esto optimiza la búsqueda y reduce el tiempo de ejecución. Este proceso de omitir subárboles se conoce como **poda** y es una técnica común en algoritmos de árboles.

Pseudocódigo del algoritmo de búsqueda por rangos:

```
1: function RANGESEARCH(node, a, b)
2:     if node = null then
3:         return
4:     end if
5:     if a < node.llave then
6:         RANGESEARCH(node.izquierdo, a, b)
7:     end if
8:     if a ≤ node.llave ≤ b then
9:         Visitar node
10:    end if
11:    if node.llave < b then
12:        RANGESEARCH(node.derecho, a, b)
13:    end if
14: end function
```

Análisis de Complejidad: En el peor de los casos, la complejidad de esta operación es $O(n)$, donde n es el número de nodos en el árbol (si el rango incluye todas las llaves del árbol). En el

mejor de los casos, la complejidad puede ser $O(\log n + k)$, donde k es el número de nodos en el rango $[a, b]$. La poda de subárboles puede mejorar significativamente el rendimiento en la práctica.

2 Árboles de búsqueda 2-3

2.1 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.

2.2 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.

2.3 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.

2.4 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 Árboles Rojo-Negros

3.1 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.

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 $O(\lg n)$.

3.2 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.

3.3 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.

3.3.1 Representación del Color

El color de un nodo se representa generalmente con un valor booleano: `True` para rojo y `False` para negro.

3.3.2 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.

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
```

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
```

3.3.3 Inserción

La inserción de un nuevo nodo en un árbol rojo-negro comienza como una inserción estándar en un árbol binario de búsqueda. El nuevo nodo se colorea inicialmente de rojo. Luego, se realizan recoloreos y rotaciones para mantener las propiedades del árbol rojo-negro.

Pseudocódigo de Inserción:

```
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);
```

```
    else if cmp > 0 then
        h.right ← PUT(h.right, llave, valor);
    else
        h.val ← valor;
    end if
    if isRed(h.right) and !isRed(h.left) then
        h = ROTATELEFT(h)
    end if
    if isRed(h.left) and isRed(h.left.left) then
        h = ROTATERIGHT(h)
    end if
    if isRed(h.left) and isRed(h.right) then
        FLIPCOLORS(h)
    end if
    h.N = size(h.left) + size(h.right) + 1
    return h
end function
```

3.3.4 Eliminación

La eliminación en un árbol rojo-negro es aún más compleja que la inserción. Similar a la inserción, la eliminación comienza como una eliminación estándar en un árbol binario de búsqueda. Luego, se realizan recoloreos y rotaciones para mantener las propiedades del árbol rojo-negro.

3.4 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).

En comparación con los árboles binarios de búsqueda no balanceados, que pueden tener una complejidad de $O(n)$ en el peor caso, los árboles rojo-negros ofrecen un rendimiento mucho más consistente y eficiente para grandes conjuntos de datos.

4 Otros tipos de árboles de búsqueda

4.1 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

4.2 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

4.3 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