Arboles

Definición puede ser recursiva o no recursiva.

- **No recursiva**: Un árbol está compuesto por un conjunto de nodos y un conjunto de aristas dirigidas que conectan parejas de nodos.

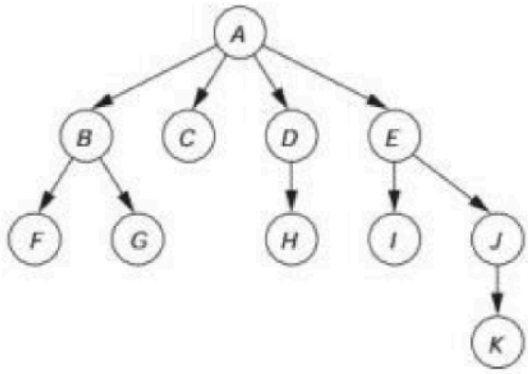
Un árbol con raíz tiene las siguientes propiedades:

1. Uno de los nodos se distingue de los demás por estar designado como raíz.
2. Todo nodo *c*, excepto la raíz, está conectado mediante una arista a exactamente un único otro nodo *p*. El nodo *p* es el padre de *c* y *c* es uno de los hijos de *p*.
3. Existe un camino único que recorre el árbol desde la raíz hasta cada nodo. El número de aristas que hay que recorrer se denomina longitud de camino.

Conceptos Clave

1. **Nodos y Relaciones**:
   - **Raíz**: Nodo superior (sin padre).
   - **Hojas**: Nodos sin hijos.
   - **Hermanos**: Nodos con el mismo padre (ej: `B, C, D, E` en la Figura 18.1).
   - **Camino**: Secuencia de aristas entre dos nodos (ej: camino `A` → `B` → `F` tiene longitud 2).
2. **Métricas**:
   - **Profundidad de un nodo**: Longitud del camino desde la raíz hasta él.
     - *Ejemplo*: Profundidad de `A` = 0, de `B` = 1, de `K` = 3.
   - **Altura de un nodo**: Longitud del camino más largo desde él hasta una hoja.
     - *Ejemplo*: Altura de `E` = 2 (camino `E` → `H` → `K`).
   - **Tamaño de un nodo**: Número de descendientes + 1 (incluyéndolo).
     - *Ejemplo*: Tamaño de `B` = 3 (`B`, `F`, `G`), de `A` = 11 (todo el árbol).
3. **Propiedades**:

- Un árbol con `N` **nodos** tiene `N-1` **aristas** (cada nodo, excepto la raíz, tiene exactamente un padre).
- **Definición recursiva**: Un árbol es:
    - Vacío, o
    - Una raíz + subárboles (cada uno es un árbol general).

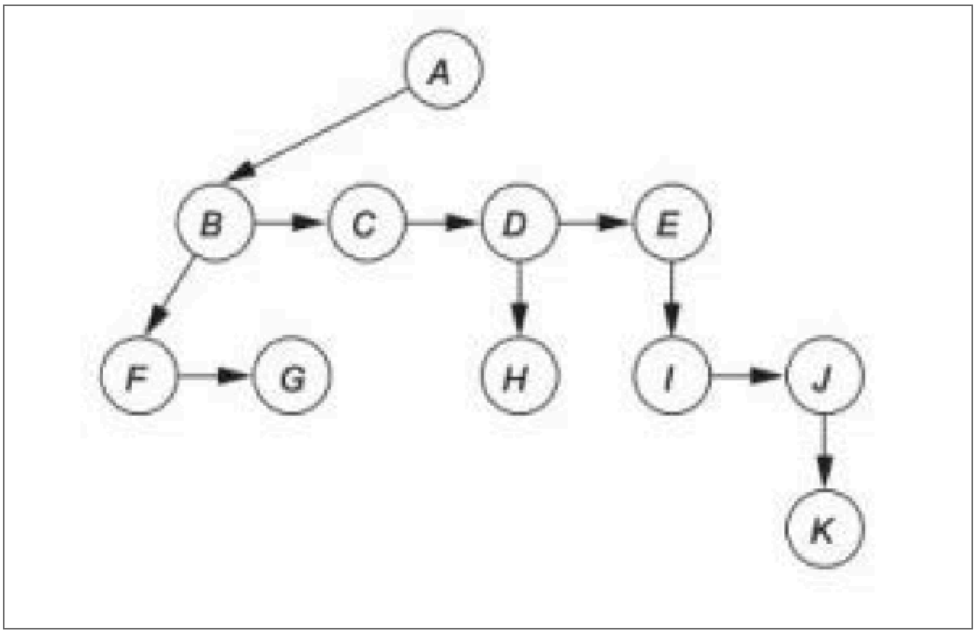


Nodo	Altura	Profundidad
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

Figura 18.1 Un árbol con la información de altura y profundidad.

Implementación de Árboles Generales

- **Método "Primer Hijo/Siguiente Hermano"**:
    - Cada nodo almacena:
        1. Un enlace a su **primer hijo** (más a la izquierda).
        2. Un enlace a su **siguiente hermano** (derecha).
    - **Ventaja**:
        - Ahorra espacio (solo 2 punteros por nodo, sin importar el número de hijos).
        - Permite representar árboles con nodos de grado variable.
- **Ejemplo visual**:



- `B` es primer hijo de `A`.
- `C` y `D` son hermanos de `B`.
- `E` es primer hijo de `B`; `F` es su hermano.

Árboles en Sistemas de Archivos

- **Estructura jerárquica**:
  - **Raíz**: Directorio principal (ej: `mark/` en Unix).
  - **Hijos**: Subdirectorios o archivos (ej: `books/`, `courses/`, `.login`).
  - **Rutas**: Nombres como `mark/books/dsaa/ch1` representan caminos en el árbol.
- **Recorridos**:
  - **Preorden**: Procesar el directorio actual antes que sus hijos (útil para listar rutas).
  - **Postorden**: Procesar hijos antes que el directorio actual (útil para cálculos como el tamaño total de bloques).
    - *Ejemplo*: Calcular el tamaño de `mark/` suma bloques de hijos (`41 + 8 + 2`) + raíz (`1`) = `52`.
- **Aplicación en Java**:

```
// Ejemplo: Recorrido postorden para calcular tamaño
int size(File file) {
    if (!file.isDirectory()) return file.getBlockSize();
    int total = 0;
    for (File child : file.listFiles()) {
        total += size(child); // Recursión en hijos
    }
    return total + file.getBlockSize(); // Suma raíz
}
```

Arboles Binarios

Un árbol binario es un árbol en el que ningún nodo puede tener más de dos hijos.

- **Hijo izquierdo** (`left`).
- **Hijo derecho** (`right`).

Definición y Estructura

- **Nodos**:
  - **Raíz**: Nodo superior sin padre.
  - **Hojas**: Nodos sin hijos.
  - **Recursividad**: Un árbol binario es:
    - Vacío, o
    - Una **raíz** + un **subárbol izquierdo** + un **subárbol derecho**.
- **Implementación**:

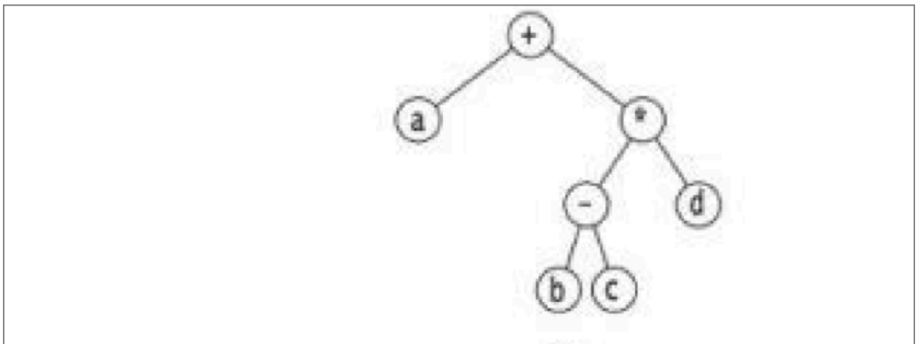
  ```
  class BinaryNode<AnyType> {
      AnyType element;           // Datos del nodo
      BinaryNode<AnyType> left;  // Hijo izquierdo
      BinaryNode<AnyType> right; // Hijo derecho
  }
  ```

Operaciones Básicas

- **Inserción y Eliminación**: Dependen del tipo de árbol (ej: árbol de búsqueda binaria).
- **Recorridos**:
  - **Preorden**: Raíz → Izquierdo → Derecho (útil para copiar árboles).
  - **Inorden**: Izquierdo → Raíz → Derecho (devuelve elementos ordenados en BST).
  - **Postorden**: Izquierdo → Derecho → Raíz (útil para liberar memoria).
  - **Por niveles** (BFS): Nivel por nivel (usando colas).
- **Métodos auxiliares**:
  - `size()`: Número de nodos (recursivo).
  - `height()`: Longitud del camino más largo a una hoja.

Aplicaciones Clave

1. **Árboles de Expresión**:
   - **Hojas**: Operandos (ej: variables o constantes).
   - **Nodos internos**: Operadores (ej: `+`,).
   - **Uso**: Evaluación de expresiones aritméticas.



(a)

   Ejemplo: Representa `(d * (b - c)) + a`.

2. **Árboles de Huffman**:
   - **Hojas**: Símbolos de un alfabeto.
   - **Códigos**: `0` para enlace izquierdo, `1` para derecho (ej: `b = 100`).

- **Uso**: Compresión de datos sin pérdida.

3. **Árboles de Búsqueda Binaria (BST)**:
    - **Propiedad:** `left < root < right`.
    - **Uso**: Búsqueda eficiente (`O(log n)` en casos balanceados).
4. **Colas de Prioridad**:
    - Implementadas con **montículos binarios** (heap).

Operación `merge`:

- **Objetivo**: Unir dos árboles bajo una nueva raíz.
- **Complicaciones**:
    - Evitar que nodos estén en múltiples árboles (aliasing).
    - Manejar casos como `t1.merge(x, t1, t2)` (autorreferencias).
- **Solución**:

```
void merge(AnyType rootItem, BinaryTree<AnyType> t1, BinaryTree
<AnyType> t2) {
    if (t1.root == this.root || t2.root == this.root) {
        // Manejar aliasing para evitar ciclos
    }
    root = new BinaryNode<>(rootItem, t1.root, t2.root);
    t1.root = t2.root = null; // Evitar compartir nodos
}
```

Implementación en Java

- **Clase `BinaryTree`:**
    - Contiene una referencia a la `raíz` (`root`).
    - Métodos: `isEmpty()`, `makeEmpty()`, recorridos (`printPreOrder`, etc.).
- **Clase `BinaryNode`:**
    - Almacena `element`, `left`, `right`.
    - Métodos estáticos: `size()`, `height()`.

Los árboles binarios son **versátiles** y **eficientes** para:

- Representar expresiones y jerarquías.
- Optimizar búsquedas y compresiones.
- Implementar estructuras avanzadas (BST, heaps).

Dificultades:

- Balanceo (evitar degradación a O(n)).
- Manejo de memoria en operaciones como `merge`.

Recursión en árboles binarios

Los árboles binarios son estructuras **recursivas por naturaleza.**

1. `size()`: Calcular el Tamaño del Árbol

- **Objetivo**: Contar el número total de nodos en el subárbol con raíz en `t`.
- **Lógica recursiva**:
  - **Caso base**: Si el árbol es vacío (`t == null`), retorna `0`.
- **Código**:

```
public static <AnyType> int size(BinaryNode<AnyType> t) {
    if (t == null) return 0;
    return 1 + size(t.left) + size(t.right);
}
```

---

2. `height()`: Calcular la Altura del Árbol

- **Objetivo**: Determinar la longitud del camino más largo desde la raíz hasta una hoja.
- **Lógica recursiva**:
  - **Caso base**: Árbol vacío (`t == null`) retorna `1` (para que una hoja tenga altura `0`).
- **Código**:

```
public static <AnyType> int height(BinaryNode<AnyType> t) {
    if (t == null) return -1;
```

```
    return 1 + Math.max(height(t.left), height(t.right));
}
```

---

3. `duplicate()`: Copiar un Árbol

- **Objetivo**: Crear una copia exacta del subárbol con raíz en el nodo actual.
- **Lógica recursiva**:
  1. Crear un nuevo nodo con el mismo `element`.
  2. Copiar recursivamente los subárboles izquierdo y derecho (si existen).
- **Código**:

```
public BinaryNode<AnyType> duplicate() {
    BinaryNode<AnyType> root = new BinaryNode<>(element, null, nul
l);
    if (left != null)  root.left = left.duplicate();  // Copia izquierda
    if (right != null) root.right = right.duplicate(); // Copia derecha
    return root;
}
```

Claves

- **Caso base**: Siempre manejar el árbol vacío (`null`).
- **División del problema**:
  - Tratar el nodo actual.
  - Delegar el resto a los subárboles izquierdo/derecho.
- **Eficiencia**:
  - **Tiempo**: *O(n)* (cada nodo se visita una vez).
  - **Espacio**: *O(h)* (altura del árbol para la pila de llamadas recursivas).

¿Por qué usar recursión?

- **Simplicidad**: El código refleja la definición recursiva del árbol.
- **Legibilidad**: Más claro que versiones iterativas con pilas/colas.

Dificultades:

- **Desbordamiento de pila**: En árboles muy desbalanceados (usar iterativos si es crítico).

Recorrido del árbol: clases iteradoras

Recorridos en:

- **Preorden:** Primero se procesa el nodo y luego recursivamente sus hijos. `duplicate()` es un ejemplo porque primero crea la raíz y luego se copia recursivamente en un subárbol izquierdo, para terminar copiando el subárbol derecho.
- **Postorden:** Los nodos se procesan después de haber procesado recursivamente a sus hijos. La información de un nodo se obtiene luego de obtener la de sus hijos. → Ej: `size()` y `height()`
- **En orden:** El nodo actual se procesa entre ambas llamadas recursivas. Se procesa recursivamente el hijo izquierdo, luego el actual y luego el hijo derecho. Se usa, por ejemplo en árboles de expresión.

→ Todas estas rutinas de recorrido son **O(n).**

(snippets de codigo con rutinas)

```
public void printPreOrder(){
    System.out.println(element); // NODO
    if (left != null)
        left.printPreOrder(); // IZQUIERDA
    if (right != null)
        right.printPreOrder(); // DERECHA
}

public void printPostOrder(){
    if (left != null)
        left.printPreOrder(); // IZQUIERDA
    if (right != null)
        right.printPreOrder(); // DERECHA
    System.out.println(element); // NODO
}

public void printInOrder(){
    if (left != null)
```

```
    left.printPreOrder(); // IZQUIERDA
  System.out.println(element); // NODO
  if (right != null)
    right.printPreOrder(); // DERECHA
  }
```

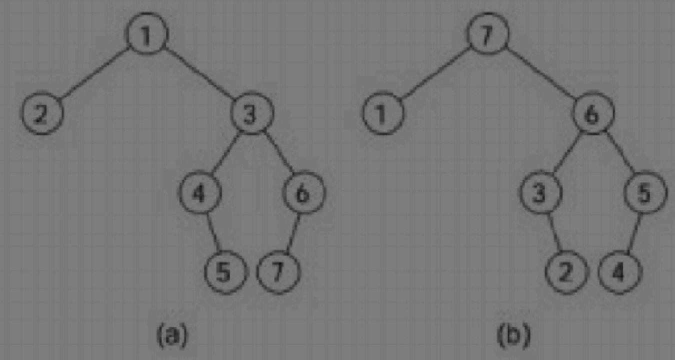


Figura 18.23 Rutas de visita en (a) preorden, (b) postorden y (c) orden.

Agregar:

- *Clase abstracta iteradora en Java

Recorrido en postorden (izquierda → derecha → raíz).

> Mantiene una pila que almacena los nodos que han sido visitados, pero cuyas llamadas recursivas no han sido todavía completadas

Cada nodo se **apila 3 veces**, marcando su estado de procesamiento:

1. **Primera vez**: Antes de procesar el subárbol izquierdo.
2. **Segunda vez**: Antes de procesar el subárbol derecho.
3. **Tercera vez**: Listo para ser "visitado" (impreso, evaluado, etc.).

IMPLEMENTACION CON PILA

- **Clase** `StNode`: Almacena un nodo y un contador (`timesPopped`).

```
protected static class StNode<AnyType> {
    BinaryNode<AnyType> node;
    int timesPopped; // 0, 1, 2 (para 1ª, 2ª, 3ª extracción)
}
```

- **Algoritmo (** `advance` **)**
  1. **Inicialización**: Apilar la raíz con `timesPopped = 0`.
  2. **Bucle principal**:
     - **Si** `timesPopped == 2`: El nodo se visita (tercera extracción).
     - **Si** `timesPopped == 0`: Apilar el hijo izquierdo (si existe).
     - **Si** `timesPopped == 1`: Apilar el hijo derecho (si existe).
  3. **Fin**: Cuando la pila esté vacía.

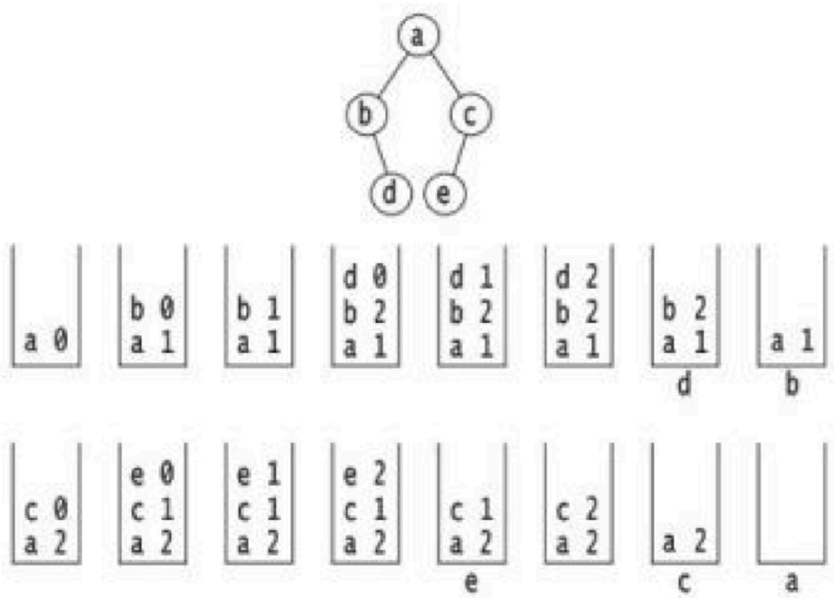


Figura 18.25 Estados de la pila durante el recorrido en postorden.

Recorrido en Preorden (Raíz → Izquierda → Derecha)

- **Orden**: Visita la raíz **antes** que los hijos.
- **Uso típico**: Copiar la estructura del árbol, expresiones prefijas (`+ A B`).

- **Implementación iterativa** (con pila):
  1. Apilar la raíz.
  2. Mientras la pila no esté vacía:
     - Extraer un nodo y visitarlo.
     - Apilar su hijo **derecho** (si existe).
     - Apilar su hijo **izquierdo** (si existe).
- **Ejemplo**:**Salida**: `1 → 2 → 3`.

```
  1
 / \\
2   3
```

Recorrido en Inorden (Izquierda → Raíz → Derecha)

- **Orden**: Visita la raíz **entre** los hijos.
- **Uso típico**: Obtener elementos de un BST **ordenados**.
- **Implementación iterativa** (con pila y contador `timesPopped`):
  1. Apilar nodos con `timesPopped = 0`.
  2. Cuando un nodo se extrae por **segunda vez**, se visita.
  3. Antes de visitarlo, se apila su hijo derecho (si existe).
- **Ejemplo**:**Salida**: `1 → 2 → 3`.

```
  2
 / \\
1   3
```

Recorrido por Niveles (BFS)

- **Orden**: Visita nodos nivel por nivel (de arriba a abajo, izquierda a derecha).
- **Uso típico**: Búsqueda en anchura, calcular la altura del árbol.
- **Implementación iterativa** (con cola):
  1. Encolar la raíz.

2. Mientras la cola no esté vacía:
    - Extraer un nodo y visitarlo.
    - Encolar su hijo **izquierdo** (si existe).
    - Encolar su hijo **derecho** (si existe).

- **Ejemplo**:**Salida**: `1 → 2 → 3 → 4 → 5`.

```
   1
  / \\
 2   3
    / \\
   4   5
```

Ejemplo de Código (Recorrido por Niveles)

```
void levelOrder(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        System.out.print(node.val + " ");
        if (node.left != null) queue.add(node.left);
        if (node.right != null) queue.add(node.right);
    }
}
```

Conclusión

- **Preorden/Postorden/Inorden**: Usan **pilas** y lógica recursiva simulada.
- **Por niveles**: Usa **colas** para procesar nodos en orden FIFO.
- **Elección del recorrido**: Depende de la aplicación (ej: inorden para BST ordenados, postorden para liberar memoria).

Árboles de búsqueda binaria

Un **BST** es un árbol binario con la siguiente propiedad para cada nodo X:

- **Subárbol izquierdo**: Todos los valores < X.
- **Subárbol derecho**: Todos los valores > X.
- **No se permiten duplicados** (o se manejan con estructuras auxiliares).
- Cuando el árbol está balanceado y tiene n nodos, su altura es de $O(\log N)$

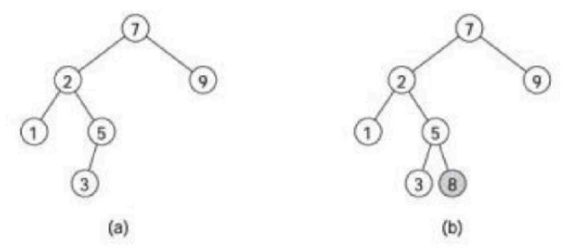


Figura 19.1 Dos árboles binarios: (a) un árbol de búsqueda; (b) no es un árbol de búsqueda.

Operaciones Detalladas

Búsqueda (`find`)

- **Algoritmo**:
  1. Comienza en la raíz.
  2. Compara el valor buscado con el nodo actual:
     - **Si es menor**: Ve al hijo izquierdo.
     - **Si es mayor**: Ve al hijo derecho.
     - **Si es igual**: Retorna el nodo.
  3. Si se llega a `null`, el valor no existe.
- **Complejidad**:
  - **Mejor caso**: O(1) (raíz).
  - **Promedio**: O(log N) (árbol balanceado).
  - **Peor caso**: O(N) (árbol degenerado, como una lista).

Inserción (`insert`)

- **Pasos**:

1. Realiza una búsqueda hasta encontrar un `null` donde debería estar el valor.
2. Inserta el nuevo nodo en esa posición.

- **Ejemplo**:
  - Insertar `6` en el árbol anterior:
    - 10 → 5 → 7 → (izquierdo de 7 es `null`).
    - Se inserta `6` como hijo izquierdo de `7`.

Eliminación (`remove`)

- **Casos específicos**:
  1. **Nodo hoja** (ej: `3`):
     - Simplemente se elimina.
  2. **Nodo con un hijo** (ej: `15` si `12` no existiera):
     - El padre del nodo eliminao apunta al hijo (único) del nodo.
  3. **Nodo con dos hijos** (ej: `5`):
     - **Paso 1**: Encuentra el **sucesor inorden** (mínimo del subárbol derecho) o predecesor (máximo del izquierdo).
       - Para `5`, el sucesor es `6` (si existiera) o `7`.
     - **Paso 2**: Copia el valor del sucesor en el nodo a eliminar.
     - **Paso 3**: Elimina el sucesor (que será un caso simple de 0 o 1 hijo).
- **Ejemplo Completo**:
  - Eliminar `5`:
    - Sucesor: `7`.
    - Copia `7` en lugar de `5`.
    - Elimina el nodo `7` original (hoja).

Operaciones Adicionales

`findMin` y `findMax`

- `findMin`:

    - Baja siempre por la izquierda hasta encontrar un `null`.
    - Ejemplo: En el árbol anterior, `findMin` retorna `3`.
- **`findMax`**:
    - Baja siempre por la derecha.
    - Ejemplo: `findMax` retorna `20`.

Recorridos

- **Inorden** (izquierda → raíz → derecha):
    - Retorna valores ordenados: `3, 5, 7, 10, 12, 15, 20`.
- **Preorden** y **Postorden**: Útiles para copiar árboles o evaluar expresiones.

Problemas y Optimizaciones

Problema del Desbalanceo

NO TENEMOS CONTROL DE LA ALTURA → Puede volverse lista enlazada. La forma del árbol y su altura depende del orden en el que se insertan los elementos. Lo mejor sería que el árbol tenga la altura mínima, es decir, que esté balanceado.

- **Causa**: Inserción/eliminación en orden secuencial (ej: `1, 2, 3, 4`).
- **Resultado**: El árbol degenera en una lista ($(O(N))$ en operaciones).

Aplicaciones Prácticas

- **Java**: `TreeSet` y `TreeMap` usan BST (implementación Rojo-Negro).
- **Bases de datos**: Índices para búsquedas rápidas.
- **Compiladores**: Árboles de sintaxis abstracta (AST).

Ejemplo de Código (Eliminación en BST)

```
public TreeNode deleteNode(TreeNode root, int key) {
    if (root == null) return null;

    if (key < root.val) {
        root.left = deleteNode(root.left, key);
```

```
    } else if (key > root.val) {
        root.right = deleteNode(root.right, key);
    } else {
        // Caso 1: Nodo hoja o con un hijo
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;

        // Caso 2: Dos hijos
        TreeNode successor = findMin(root.right);
        root.val = successor.val;
        root.right = deleteNode(root.right, successor.val);
    }
    return root;
}

private TreeNode findMin(TreeNode node) {
    while (node.left != null) node = node.left;
    return node;
}
```

Árboles AVL

Un **árbol AVL** es un **árbol de búsqueda binaria (BST)** con una **condición de equilibrio adicional**:

- Para **todo nodo**, la diferencia de alturas entre sus subárboles izquierdo y derecho (**factor de equilibrio**) debe ser $(-1), (0), \text{ o } (1)$.
- **Altura de un subárbol vacío**:(-1).

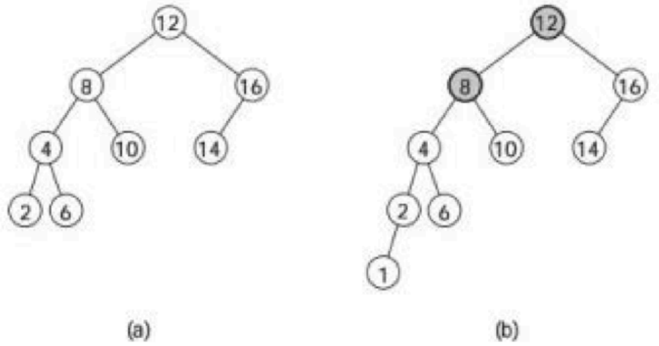


Figura 19.21 Dos árboles de búsqueda binaria: (a) un árbol AVL; (b) un árbol no AVL (los nodos no equilibrados se muestran más oscuros).

Importancia del Equilibrio

- **Objetivo**: Garantizar que la **altura del árbol sea** $(O(\log N))$, evitando degradación a $(O(N))$ (como en BST desbalanceados).
- **Teorema 19.3**:
  - Un árbol AVL de altura \(H\) tiene al menos ($F_{H+3} - 1$) nodos (donde (F_i) es el (i)-ésimo número de Fibonacci).
  - Esto implica: $(H < 1.441 \log(N+2) - 1.328)$

Operaciones y Mantenimiento del Equilibrio

Inserción y Reequilibrio

1. **Inserción estándar**: Como en un BST, pero **verificando el equilibrio** desde el nodo insertado hasta la raíz.
2. **Casos de desequilibrio** (Figura 19.21):
   - **Caso 1**: Inserción en el subárbol **izquierdo-izquierdo** (LL).
   - **Caso 2**: Inserción en el subárbol **izquierdo-derecho** (LR).
   - **Caso 3**: Inserción en el subárbol **derecho-izquierdo** (RL).
   - **Caso 4**: Inserción en el subárbol **derecho-derecho** (RR).

Rotaciones para Reequilibrar

- **Rotación Simple** (Figuras 19.24, 19.26):
  - **Caso 1 (LL)**: Rotación a la **derecha** sobre el nodo desbalanceado.

```
static BinaryNode rotateWithLeftChild(BinaryNode k2) {
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}
```

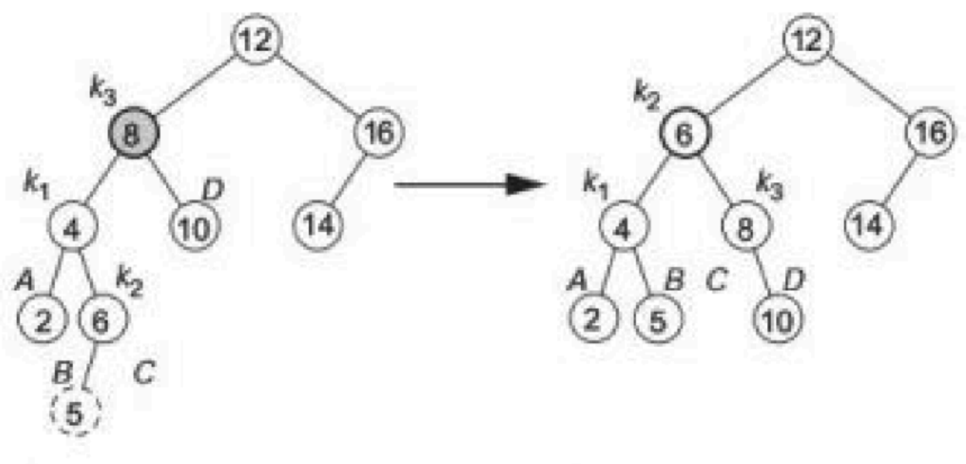
- **Caso 4 (RR)**: Rotación a la **izquierda** (simétrica a LL).

- **Rotación Doble** (Figuras 19.29, 19.31):
  - **Caso 2 (LR)**:
    1. Rotación a la **izquierda** sobre el hijo izquierdo.
    2. Rotación a la **derecha** sobre el nodo desbalanceado.

```
static BinaryNode doubleRotateWithLeftChild(BinaryNode k3) {
    k3.left = rotateWithRightChild(k3.left); // Paso 1
    return rotateWithLeftChild(k3);          // Paso 2
}
```

  - **Caso 3 (RL)**: Simétrico a LR (rotación derecha-izquierda).

Ejemplo de Rotación Doble (Figura 19.30):



(a) Antes de la rotación. (b) Después de la rotación.

Figura 19.30 Una rotación doble corrige el árbol AVL después de la inserción de 5.

Complejidad y Ventajas

- **Operaciones**:
  - **Búsqueda/Inserción/Eliminación**: $(O(\log N))$ en el peor caso.
- **Ventajas**:
  - Garantiza **altura logarítmica** incluso en inserciones/eliminaciones arbitrarias.
  - Ideal para aplicaciones donde el **peor caso** debe ser controlado (ej: bases de datos).
- **Desventajas**:
  - **Overhead**: Cálculo constante de factores de equilibrio y rotaciones.
  - **Alternativas más eficientes**: Árboles Rojo-Negro o AA (menos rotaciones).