Unidad 7: Árboles

Estructura de datos

Cosijopii García

Marzo 2026

7.3 Tipos de árboles

7.3.1 Árboles Generales (N-arios)

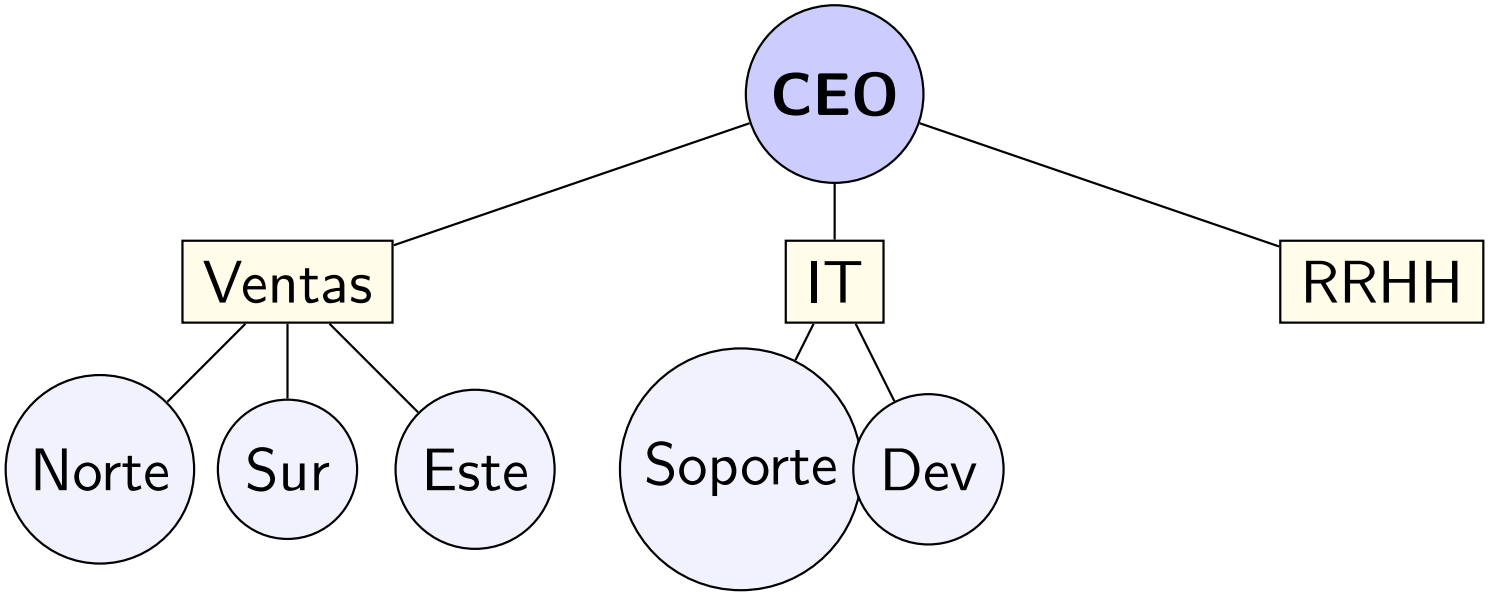
Un **Árbol General** es la forma más abstracta de esta estructura.

- **Grado del nodo:** El número de hijos de un nodo no está restringido.
- **Estructura:** Representa jerarquías donde una entidad posee o contiene múltiples sub-entidades de forma desigual.
- **Aplicación:** Sistemas de archivos de SO, organigramas corporativos, procesamiento de documentos XML/HTML.

Analogía

Una carpeta en tu computadora que contiene 10 subcarpetas, y una de esas contiene solo un archivo. No hay simetría obligatoria.

Visualización: Estructura de Árbol General



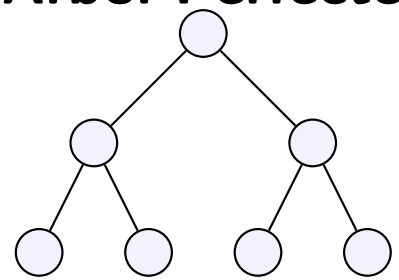
7.3.2 Árboles Binarios

Es un árbol enraizado donde cada nodo tiene, como máximo, un **grado de salida de 2**.

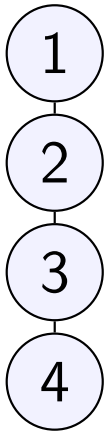
- **Definición Formal:** Un conjunto finito de nodos que está vacío, o consiste en una raíz y dos subárboles binarios disjuntos (izquierdo y derecho).

Árboles Binarios

Árbol Perfecto



Árbol Degenerado (Skewed)



El árbol degenerado pierde la eficiencia logarítmica, comportándose como una lista enlazada.

7.3.3 Árboles Binarios de Búsqueda (ABB)

Añaden una **propiedad de orden** a la estructura binaria para optimizar el acceso a datos.

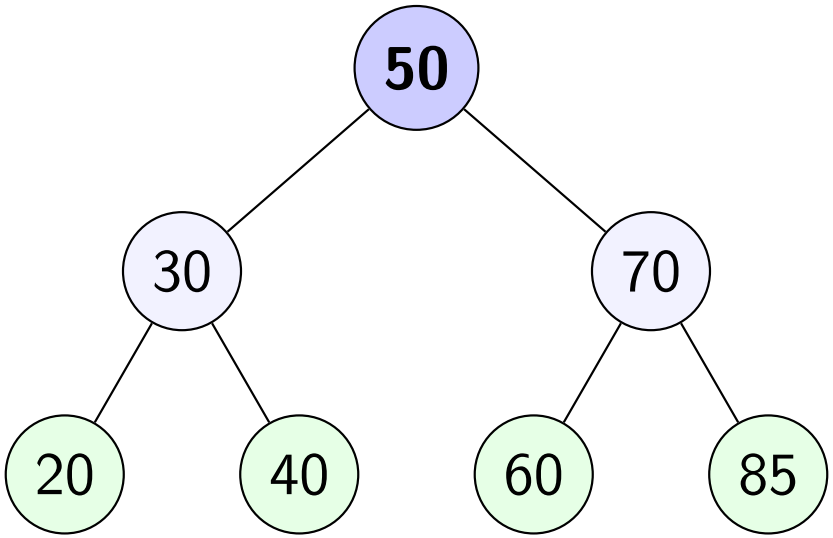
La Regla de Ordenamiento

Para cualquier nodo k:

- Todos los nodos en su subárbol izquierdo $< k$.
- Todos los nodos en su subárbol derecho $> k$.

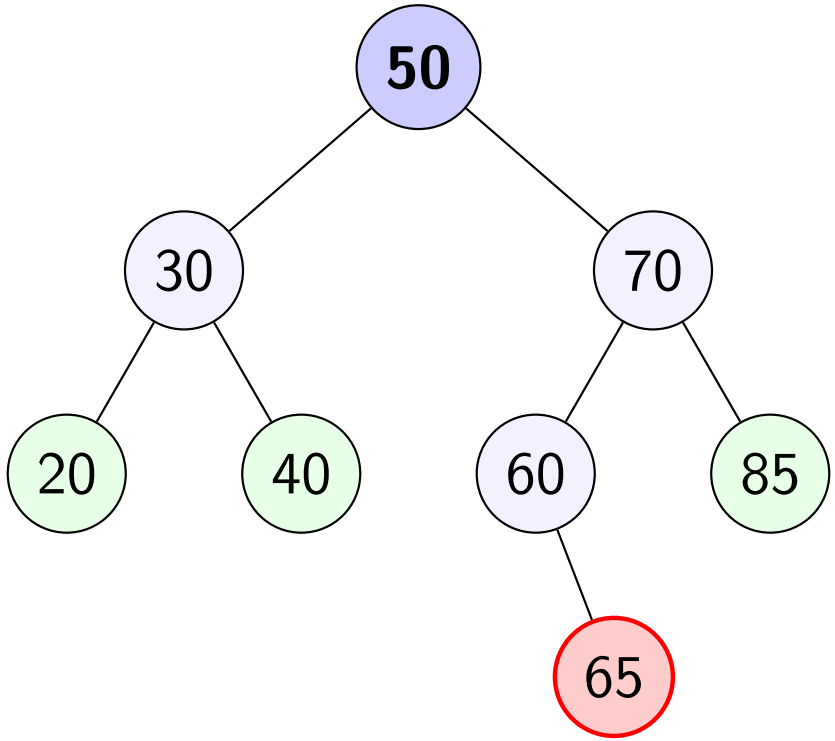
- **Costo Computacional:** En un árbol balanceado, la búsqueda es $O(\log n)$.
- **Uso:** Implementación de diccionarios, conjuntos y mapas.

Propiedad de Orden del ABB



Reto didáctico: Si quisiéramos insertar el valor **65**, ¿dónde quedaría? (Derecha de 50 $\rightarrow$ Izquierda de 70 $\rightarrow$ Derecha de 60).

Solución: Inserción en el ABB



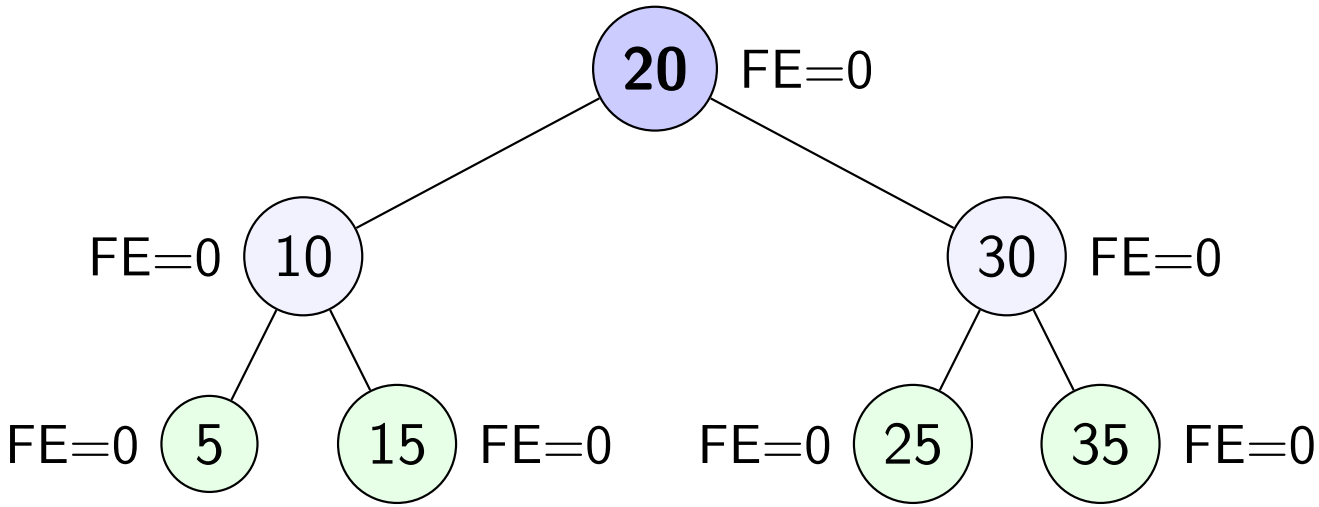
Resultado del reto: El valor **65** recorrió el siguiente camino:
$65 > 50$ (Derecha) $\rightarrow 65 < 70$ (Izquierda) $\rightarrow 65 > 60$ (Derecha).

7.3.4 Árboles Auto-balanceados (AVL)

Nombrados así en honor a sus inventores (**A**delson-**V**elsky y **L**andis, 1962), surgen para evitar la degradación de los ABB. Un árbol AVL es un ABB con una estricta restricción de altura.

- **Factor de Equilibrio (FE):** $FE = altura(Subder) - altura(Subizq)$.
- **Condición AVL:** Para todo nodo, $|FE| \leq 1$.
- **Mecanismo:** Si una inserción rompe esta regla, se aplican **rotaciones** (Simple o Doble).

Visualización: Balanceo AVL

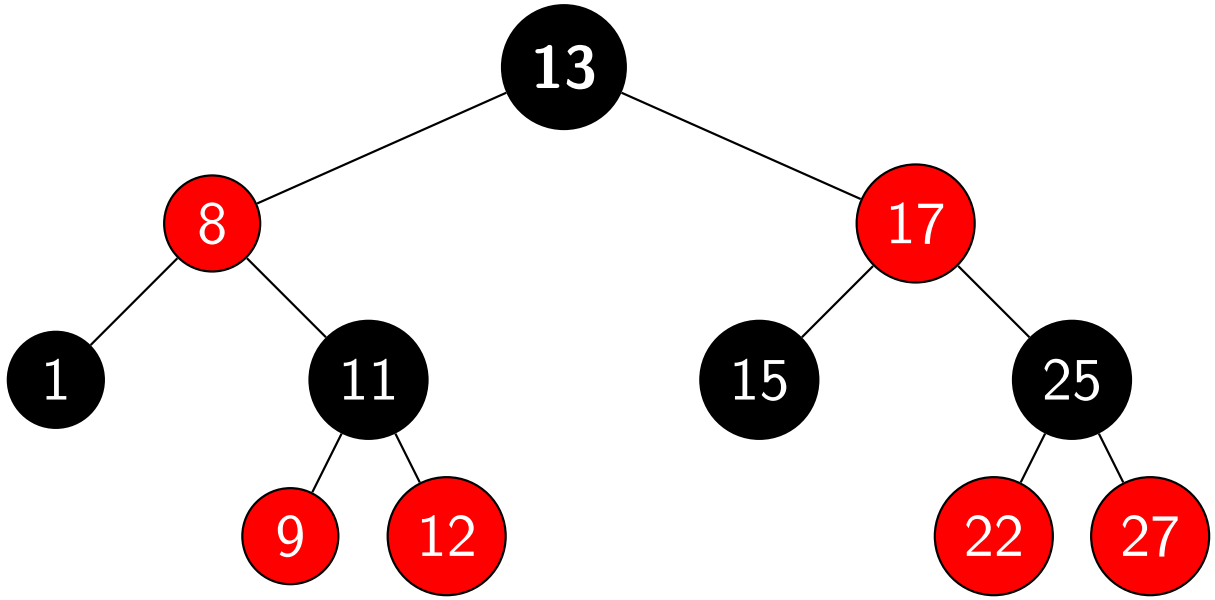


7.3.5 Árboles Rojo y Negro

Un **Árbol Rojo-Negro** es un ABB auto-balanceado que añade un bit extra de información por nodo: su color. Para garantizar el auto-balanceo y mantener las operaciones en $O(\log n)$, el árbol debe cumplir estrictamente estas propiedades:

1. **Propiedad de Color:** Cada nodo es estrictamente **Rojo** o **Negro**.
2. **Raíz Segura:** La raíz del árbol **siempre** es Negra.
3. **Sin Rojos Adyacentes:** Un nodo Rojo no puede tener hijos Rojos. (Es decir, si un nodo es rojo, sus dos hijos deben ser obligatoriamente negros).
4. **Altura Negra Perfecta:** Cualquier camino desde un nodo dado hasta sus hojas descendientes contiene **exactamente la misma cantidad** de nodos Negros.

Visualización: Árbol Rojo y Negro



Ventaja sobre AVL: Requiere menos rotaciones durante la inserción/eliminación, haciéndolo ideal para sistemas con muchas modificaciones.

Rojo-Negro vs. AVL: ¿Cuál elegir?

Ambos garantizan operaciones en $O(\log n)$, pero tienen un enfoque distinto en su balanceo:

La gran ventaja del Árbol Rojo-Negro

Menos rotaciones. Al tener un balanceo menos estricto, requiere muchas menos reestructuraciones al insertar o eliminar nodos. Es más rápido para escribir datos.

- **Usar Rojo-Negro cuando:** Tu sistema hace **muchas modificaciones** (inserciones/eliminaciones).
  Ejemplo: `std::map` en C++, `TreeMap` en Java.
- **Usar AVL cuando:** Tu sistema hace **muchas lecturas** (búsquedas frecuentes y pocos cambios). Al ser más plano y estricto, encontrar un dato es ligeramente más rápido.
  Ejemplo: Bases de datos en memoria para consultas de solo lectura.

7.3.6 Árboles B (B-Trees)

Fueron diseñados específicamente para optimizar la lectura/escritura en **discos magnéticos y bases de datos**. A diferencia de los anteriores, no son binarios; son árboles de búsqueda **multicamino** (anchos y poco profundos).

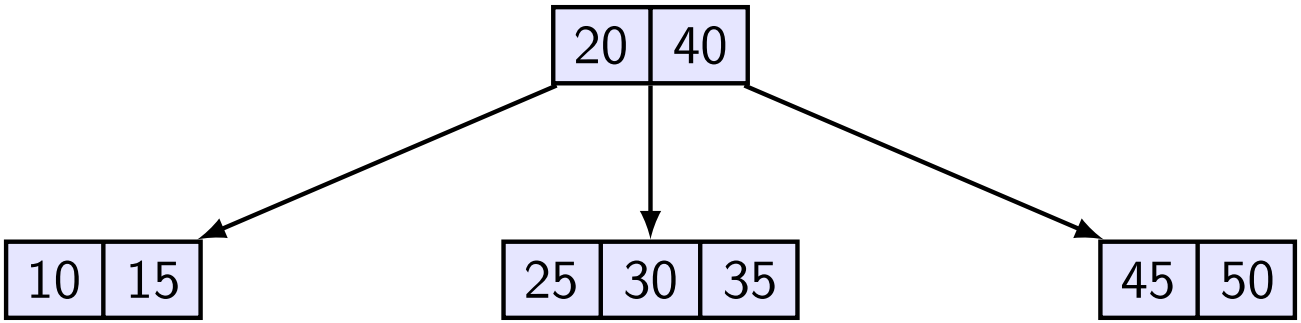
Características Principales (Orden m)

- **Múltiples datos por nodo:** Un nodo puede contener hasta $m - 1$ claves (datos) ordenadas.
- **Múltiples caminos:** Un nodo interno puede tener hasta m hijos.
- **Balanceo Perfecto: Todos los nodos hoja están exactamente en el mismo nivel**. El árbol crece hacia arriba, no hacia abajo.

Nota: Al meter muchos datos en un solo nodo, logramos leer un bloque entero del disco duro de un solo golpe, minimizando el costoso I/O (Entrada/Salida).

Visualización: Árbol B de Orden 4

En un árbol de Orden $m = 4$, cada nodo tiene máximo **3 claves** y **4 hijos**. Observa cómo los datos dividen los caminos:



- ¿Buscas el **30**? $\rightarrow$ $30 > 20$ y $30 < 40$ $\rightarrow$ Tomas el camino del centro.
- Todas las hojas están en el **nivel 2**.

7.4 Operaciones con árboles binarios

Representación en C

Antes de operar un árbol, debemos definir cómo nace en la memoria. En C, utilizamos `structs` y **punteros** genéricos para enlazar los nodos, aprovechando la memoria dinámica (Heap).

Anatomía de un Nodo

- Un campo para el **dato** (ej. entero).
- Un puntero al **hijo izquierdo**.
- Un puntero al **hijo derecho**.

Un árbol vacío es simplemente un puntero inicializado en `NULL`.

```c
// Definicion del Nodo en C
typedef struct Nodo {
    int dato;
    struct Nodo* izq;
    struct Nodo* der;
} Nodo;

// Creacion de un nuevo nodo
Nodo* crearNodo(int valor) {
    Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
    nuevo->dato = valor;
    nuevo->izq = NULL;
    nuevo->der = NULL;
    return nuevo;
}
```

7.4.1 Operaciones Fundamentales

Con nuestra estructura `Nodo` definida, podemos manipular el árbol. Todo se basa en **recursividad**.

- **Inserción:** Navegar hasta encontrar un puntero `NULL` y enlazar ahí un nuevo `malloc`.
- **Búsqueda:** Recorrer los punteros buscando coincidencias.
- **Eliminación:** Liberar memoria (`free()`) y reconectar los punteros huérfanos.

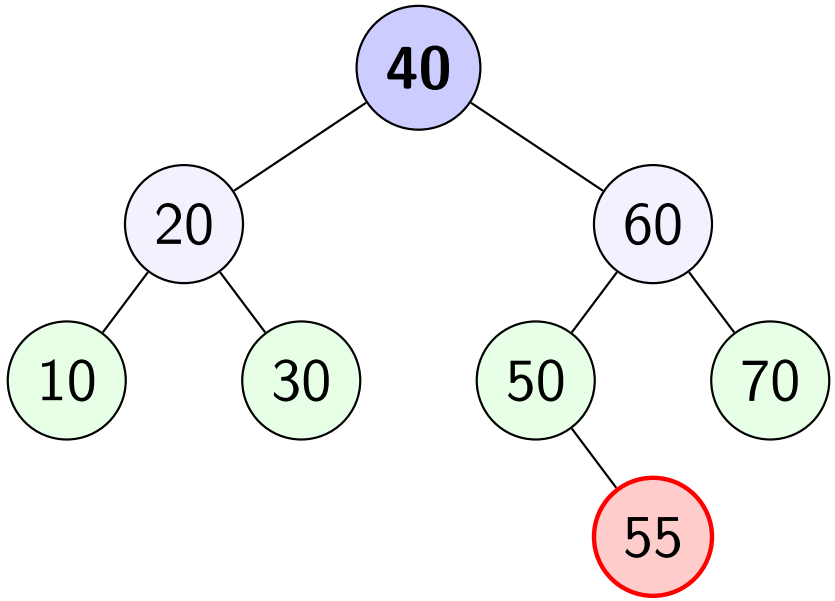
Recursividad

Dado que cada hijo de un nodo es, por definición, la raíz de su propio subárbol, las funciones recursivas son la forma más natural y elegante de programar estas operaciones.

Inserción en ABB: Teoría y Reglas

Reglas de Inserción

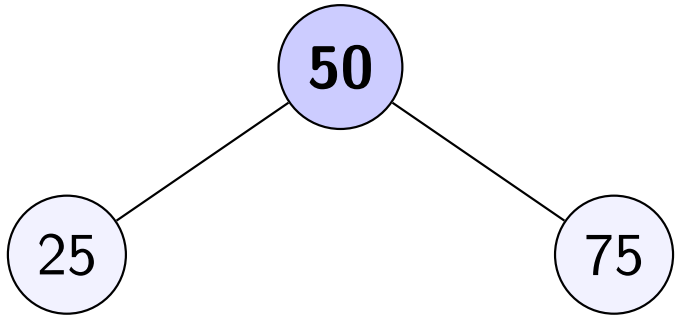
1. Comenzamos a comparar desde la **Raíz**.
2. Si el valor es **menor**, bajamos por la izquierda.
3. Si el valor es **mayor**, bajamos por la derecha.
4. Repetimos hasta encontrar un espacio vacío y enlazamos el nodo.



Ejemplo: Insertar el **55**. *Camino:* $55 > 40$ *(Der)* $\rightarrow 55 < 60$ *(Izq)* $\rightarrow 55 > 50$ *(Der).*

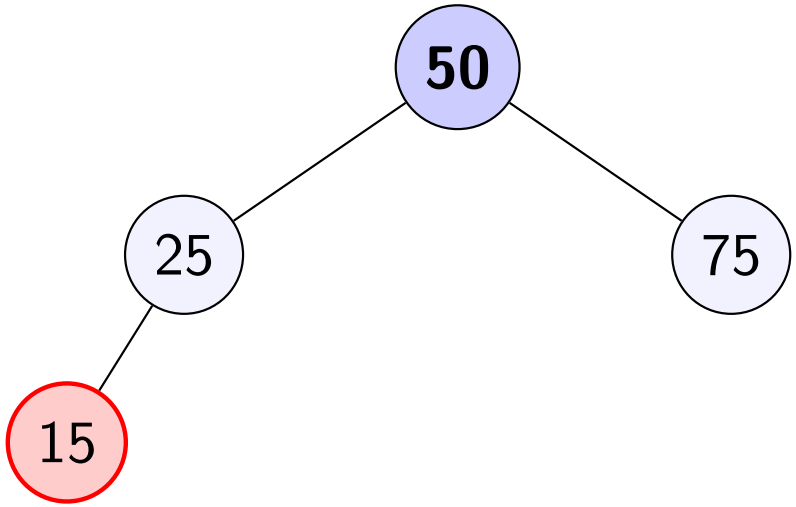
Ejercicio:

Instrucciones: Dado el siguiente árbol base, determina dónde quedarían enlazados los valores **15**, **35** y **65**, asumiendo que los insertas uno por uno en ese orden.



Ejercicio:

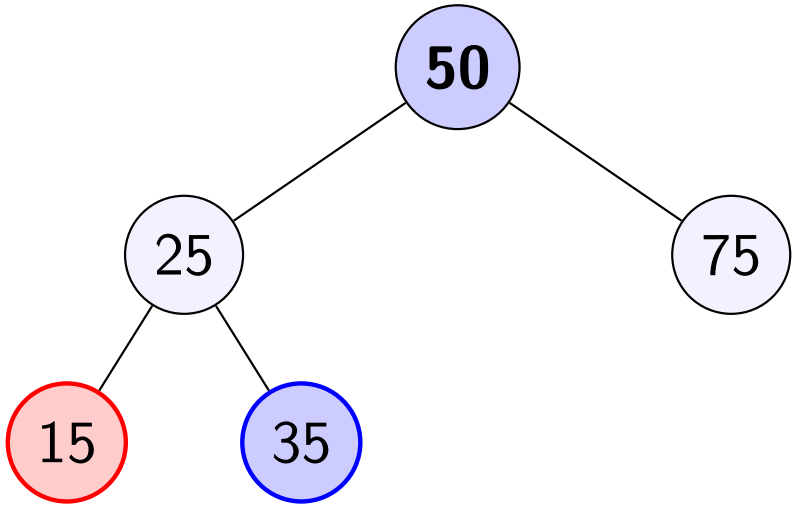
Instrucciones: Dado el siguiente árbol base, determina dónde quedarían enlazados los valores **15**, **35** y **65**, asumiendo que los insertas uno por uno en ese orden.



- **Valor 15:** Queda a la **izquierda** del 25.
-
-

Ejercicio:

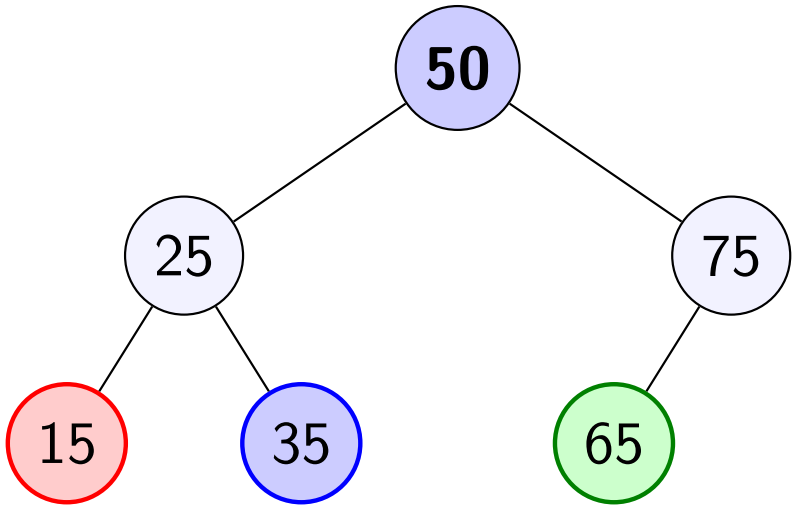
Instrucciones: Dado el siguiente árbol base, determina dónde quedarían enlazados los valores **15**, **35** y **65**, asumiendo que los insertas uno por uno en ese orden.



- **Valor 15:** Queda a la **izquierda** del 25.
- **Valor 35:** Queda a la **derecha** del 25.
-

Ejercicio:

Instrucciones: Dado el siguiente árbol base, determina dónde quedarían enlazados los valores **15**, **35** y **65**, asumiendo que los insertas uno por uno en ese orden.



- **Valor 15:** Queda a la **izquierda** del 25.
- **Valor 35:** Queda a la **derecha** del 25.
- **Valor 65:** Queda a la **izquierda** del 75.

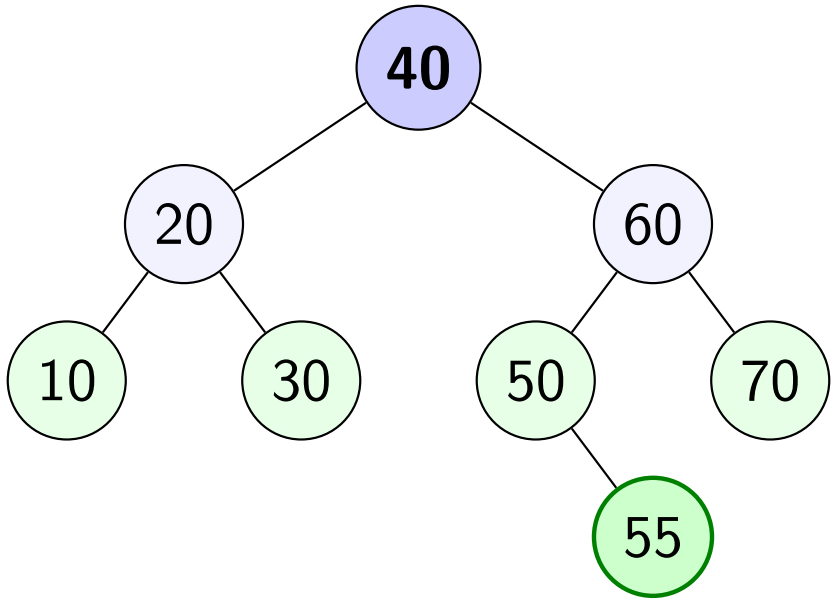
Implementación: Inserción en C

```c
Nodo* insertar(Nodo* raiz, int valor) {
    // 1. Caso base: encontramos un espacio vacio
    if (raiz == NULL) {
        return crearNodo(valor);
    }

    // 2. Si es menor, construimos hacia la izquierda
    if (valor < raiz->dato) {
        raiz->izq = insertar(raiz->izq, valor);
    }
    // 3. Si es mayor, construimos hacia la derecha
    else if (valor > raiz->dato) {
        raiz->der = insertar(raiz->der, valor);
    }

    // 4. Retornamos la raiz actualizada
    return raiz;
}
```

Búsqueda en ABB: Teoría y Reglas

Reglas de Búsqueda

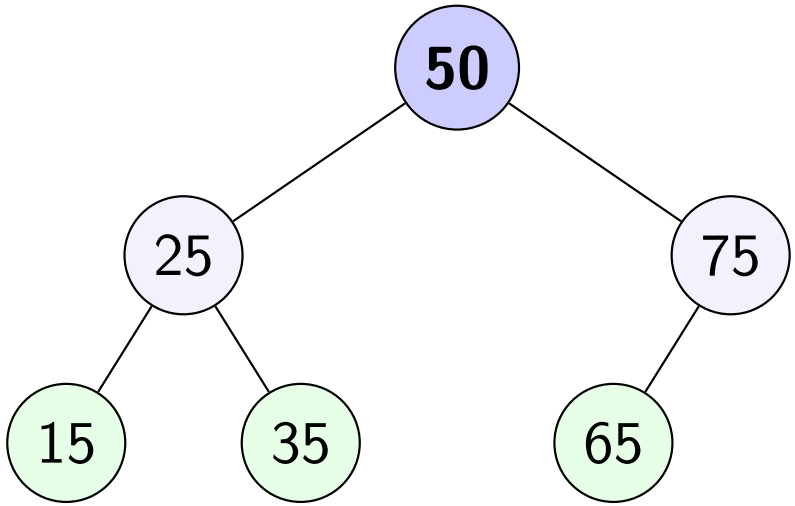
1. Comenzamos a comparar desde la **Raíz**.
2. Si el valor es **igual**, ¡Lo encontramos!
3. Si el valor es **menor**, buscamos por la izquierda.
4. Si el valor es **mayor**, buscamos por la derecha.



Ejemplo: Buscar el **55**. *Camino:* $55 > 40$ *(Der)* $\rightarrow 55 < 60$ *(Izq)* $\rightarrow 55 > 50$ *(Der)* $\rightarrow$ **¡Encontrado!**

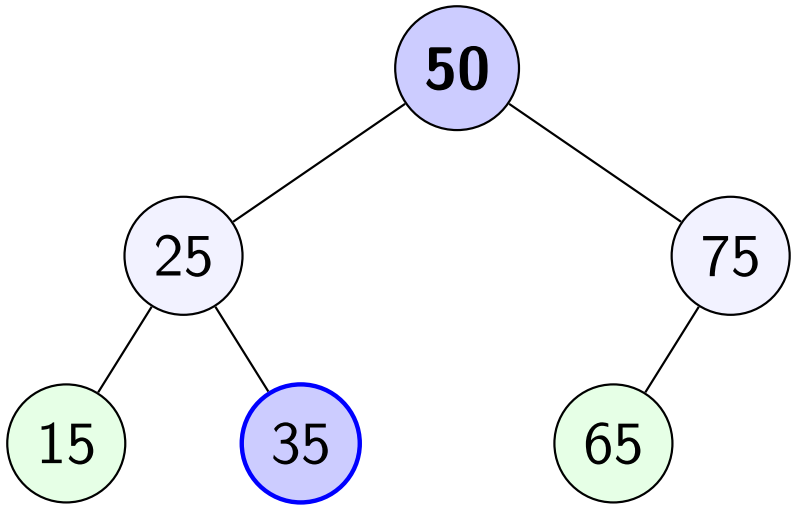
Ejercicio:

Instrucciones: Dado el siguiente árbol, determina qué camino tomarías y dónde terminaría la búsqueda para los valores **35**, **65** y **20**.



Ejercicio:

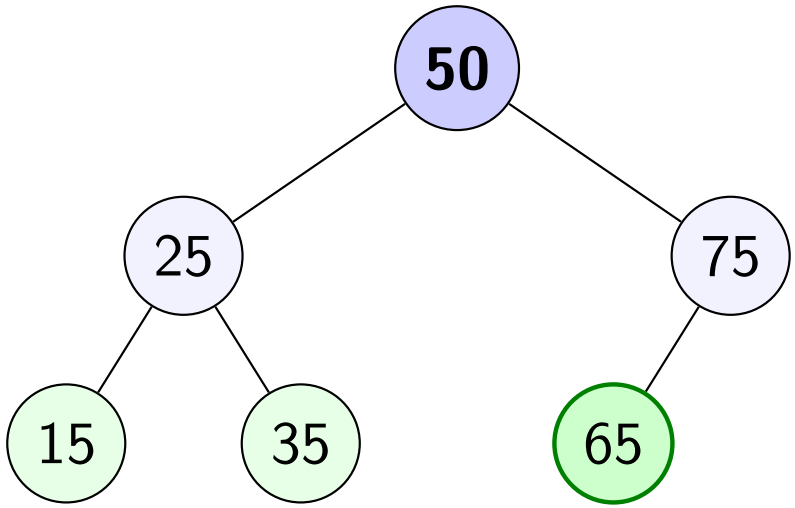
Instrucciones: Dado el siguiente árbol, determina qué camino tomarías y dónde terminaría la búsqueda para los valores **35**, **65** y **20**.



- **Valor 35:** 50 (Izq) $\rightarrow$ 25 (Der) $\rightarrow$ **¡Encontrado!**
-
-

Ejercicio:

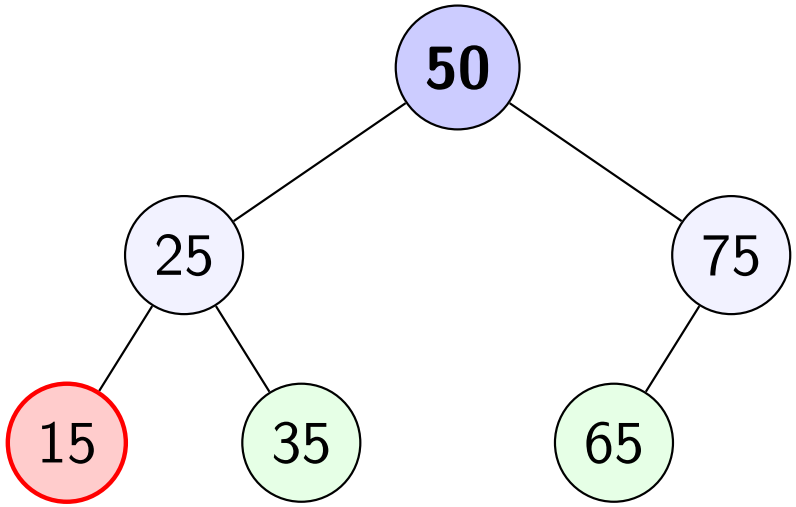
Instrucciones: Dado el siguiente árbol, determina qué camino tomarías y dónde terminaría la búsqueda para los valores **35**, **65** y **20**.



- **Valor 35:** 50 (Izq) $\rightarrow$ 25 (Der) $\rightarrow$ **¡Encontrado!**
- **Valor 65:** 50 (Der) $\rightarrow$ 75 (Izq) $\rightarrow$ **¡Encontrado!**
-

Ejercicio:

Instrucciones: Dado el siguiente árbol, determina qué camino tomarías y dónde terminaría la búsqueda para los valores **35**, **65** y **20**.



- **Valor 35:** 50 (Izq) → 25 (Der) → **¡Encontrado!**
- **Valor 65:** 50 (Der) → 75 (Izq) → **¡Encontrado!**
- **Valor 20:** 50 (Izq) → 25 (Izq) → 15 (Der) → **NULL (No existe).**

Implementación: Búsqueda en C

```
Nodo* buscar(Nodo* raiz, int valor) {
    // 1. Caso base: vacio (no existe) o lo encontramos
    if (raiz == NULL || raiz->dato == valor) {
        return raiz;
    }

    // 2. Si el valor buscado es menor, iterar por la iz
    if (valor < raiz->dato) {
        return buscar(raiz->izq, valor);
    }

    // 3. Si el valor buscado es mayor, iterar por la de
    return buscar(raiz->der, valor);
}
```

Eliminación en ABB: los 3 Casos

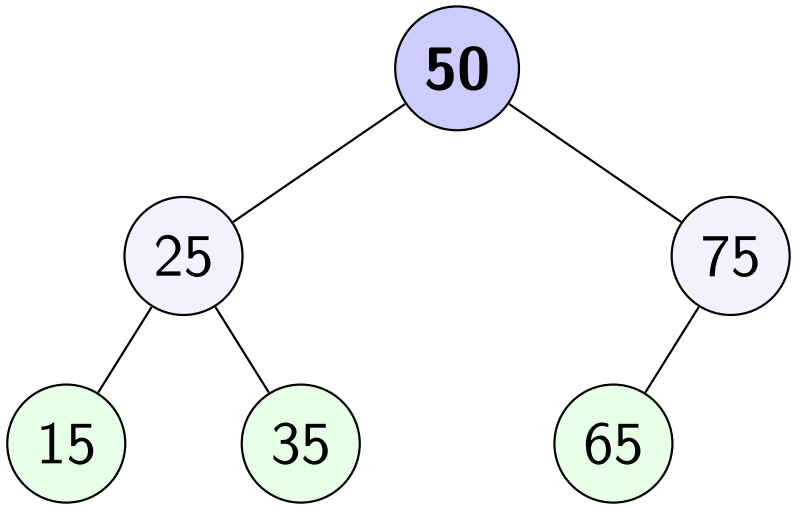
Eliminar es la operación más compleja porque, al quitar un nodo en medio del árbol, debemos reconectar las ramas sin romper la propiedad de orden.

Los 3 Escenarios Posibles

1. **Nodo Hoja (0 hijos):** Es el más fácil. Simplemente se borra y su lugar lo ocupa un `NULL`.
2. **Nodo con 1 hijo:** El nodo se borra y su único hijo "sube"directamente a ocupar su lugar (hace un bypass).
3. **Nodo con 2 hijos:** ¡El caso difícil! Buscamos a su **Sucesor** (el nodo más pequeño del lado derecho), copiamos su valor para reemplazar al nodo actual, y luego eliminamos al sucesor original.

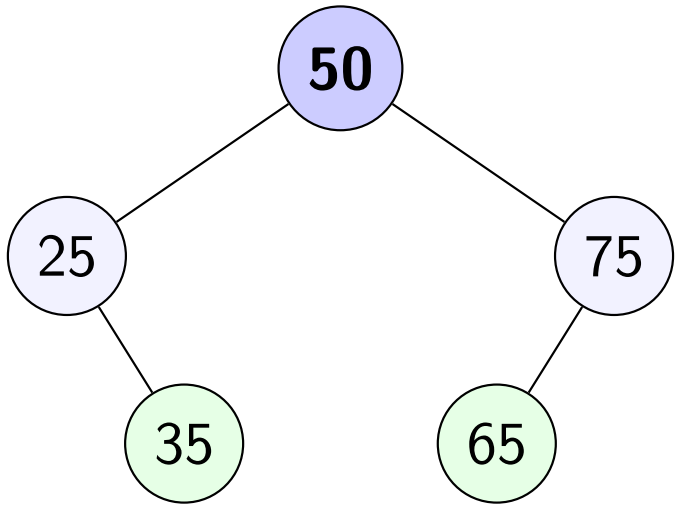
Ejercicio: Reconectando el Árbol

Instrucciones: Partiendo siempre del árbol base, analiza cómo quedaría estructurado si eliminamos el **15**, o el **75**, o el **25**.



Ejercicio: Reconectando el Árbol

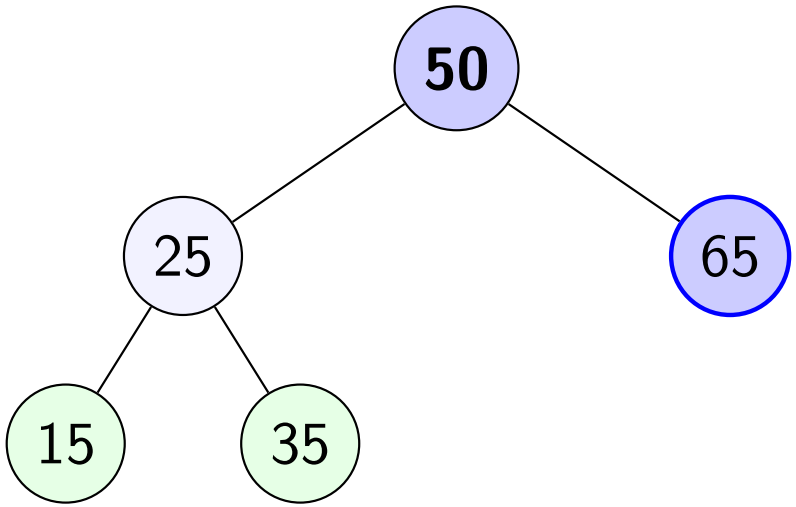
Instrucciones: Partiendo siempre del árbol base, analiza cómo quedaría estructurado si eliminamos el **15**, o el **75**, o el **25**.



- **Caso 1 (Borrar 15):** Al ser hoja, simplemente desaparece.
-
-

Ejercicio: Reconectando el Árbol

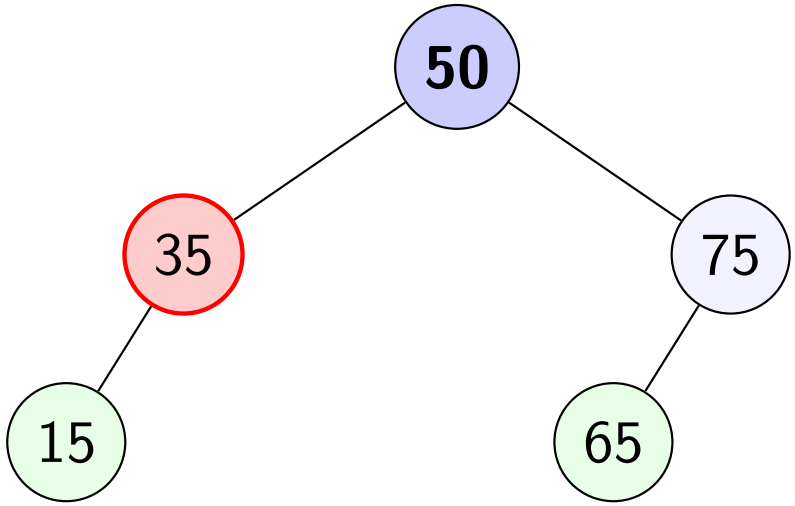
Instrucciones: Partiendo siempre del árbol base, analiza cómo quedaría estructurado si eliminamos el **15**, o el **75**, o el **25**.



- **Caso 1 (Borrar 15):** Al ser hoja, simplemente desaparece.
- **Caso 2 (Borrar 75):** Tiene 1 hijo. El **65** sube a ocupar su lugar.
-

Ejercicio: Reconectando el Árbol

Instrucciones: Partiendo siempre del árbol base, analiza cómo quedaría estructurado si eliminamos el **15**, o el **75**, o el **25**.



- **Caso 1 (Borrar 15):** Al ser hoja, simplemente desaparece.
- **Caso 2 (Borrar 75):** Tiene 1 hijo. El **65** sube a ocupar su lugar.
- **Caso 3 (Borrar 25):** Tiene 2 hijos. Buscamos su sucesor (35). El **35** toma su lugar y desaparece de abajo.

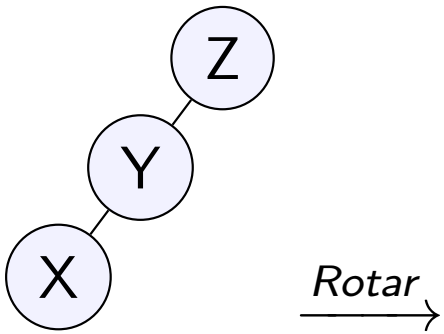
Implementación: Eliminación en C

```c
Nodo* eliminar(Nodo* raiz, int valor) {
    if (raiz == NULL) return raiz; // Caso base

    // Buscar el nodo
    if (valor < raiz->dato) raiz->izq = eliminar(raiz->izq, valor);
    else if (valor > raiz->dato) raiz->der = eliminar(raiz->der, valor);
    else {
        //  Nodo encontrado!
        // Casos 1 y 2: Cero o un hijo
        if (raiz->izq == NULL) {
            Nodo* temp = raiz->der; free(raiz); return temp;
        } else if (raiz->der == NULL) {
            Nodo* temp = raiz->izq; free(raiz); return temp;
        }

        // Caso 3: Dos hijos. Buscamos sucesor (minimo del subarbol derecho)
        Nodo* temp = minimoValorNodo(raiz->der);

        raiz->dato = temp->dato; // Copiamos valor del sucesor

        // Eliminamos al sucesor original
        raiz->der = eliminar(raiz->der, temp->dato);
    }
    return raiz;
}
```

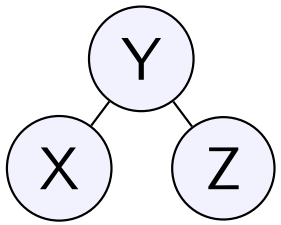
7.4.4 Operación de Balanceo (Rotaciones)

Cuando una inserción o eliminación rompe el equilibrio de un árbol AVL o Rojo-Negro, el árbol reestructura sus punteros usando **Rotaciones** en tiempo $O(1)$.

Desbalance Izquierdo
(Rotación Simple a la Derecha)



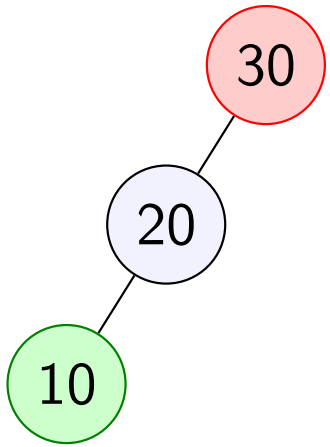
Árbol Balanceado



7.4.4 Balanceo: Rotación a la Derecha

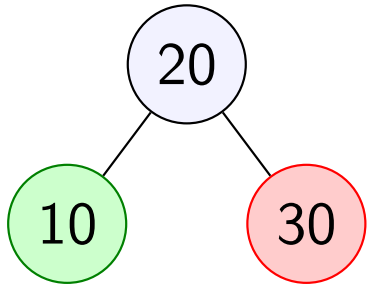
Veámoslo con un ejemplo: insertamos **30**, **20** y finalmente **10**. El árbol se degenera en una lista (desbalanceo izquierdo), por lo que aplicamos una **Rotación Simple a la Derecha** en el nodo 30 para restaurar el orden $O(\log n)$.

Desbalance Izquierdo



Rotar

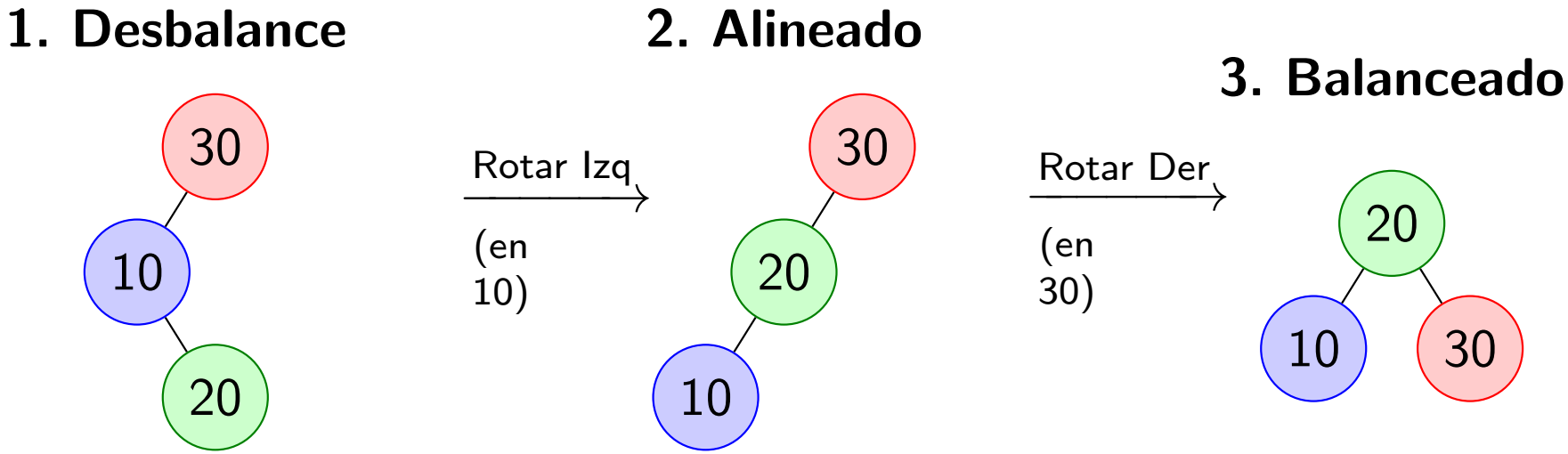
Árbol Balanceado



El nodo 20 sube a la raíz, empujando al 30 hacia abajo a su derecha.

7.4.5 Balanceo: Rotación Doble (Izq-Der)

Ocurre al insertar en **"Zig-Zag"** (ej. **30**, luego **10**, luego **20**). Una sola rotación no basta porque el 20 quedaría atrapado. ¡Solución: Aplicar dos rotaciones simples!



Paso 1: Rotación Izquierda en el hijo (10) para hacer una línea. Paso 2: Rotación Derecha en la raíz (30).

Implementación: Rotaciones Simples en C

Ambas operaciones se ejecutan en tiempo $O(1)$.

```
// --- ROTACION A LA DERECHA (Para desbalanceo Izquierdo) ---
// 'y' es la raiz desbalanceada original
Nodo* rotacionDerecha(Nodo* y) {
    Nodo* x = y->izq;       // 'x' sera la nueva raiz
    Nodo* T2 = x->der;      // Rescatamos el subarbol derecho de 'x'

    x->der = y;             // El nodo original baja a la derecha
    y->izq = T2;            // El subarbol huerfano se enlaza a la izquierda

    // (Aqui se actualizarian las alturas de 'y' y luego de 'x')
    return x;               // Retornamos la nueva raiz
}

// --- ROTACION A LA IZQUIERDA (Para desbalanceo Derecho) ---
// 'x' es la raiz desbalanceada original
Nodo* rotacionIzquierda(Nodo* x) {
    Nodo* y = x->der;       // 'y' sera la nueva raiz
    Nodo* T2 = y->izq;      // Rescatamos el subarbol izquierdo de 'y'

    y->izq = x;             // El nodo original baja a la izquierda
    x->der = T2;            // El subarbol huerfano se enlaza a la derecha

    // (Aqui se actualizarian las alturas de 'x' y luego de 'y')
    return y;               // Retornamos la nueva raiz
}
```

7.5 Recorrido en un árbol binario

7.5 Recorrido en un Árbol Binario

Recorrer un árbol significa visitar **todos sus nodos** exactamente una vez en un orden específico. Dado que no es una estructura lineal, existen múltiples caminos sistemáticos.

1. **Preorden:** Raíz $\rightarrow$ Izquierda $\rightarrow$ Derecha.
2. **Inorden:** Izquierda $\rightarrow$ Raíz $\rightarrow$ Derecha.
3. **Posorden:** Izquierda $\rightarrow$ Derecha $\rightarrow$ Raíz.

7.5.1 Preorden (Raíz, Izq, Der)

Aplicación Práctica:
Se utiliza para representar y recorrer **Estructuras Jerárquicas** como sistemas de archivos (directorios); permite procesar una carpeta (el nodo raíz) antes de listar su contenido interno (los nodos hijos).

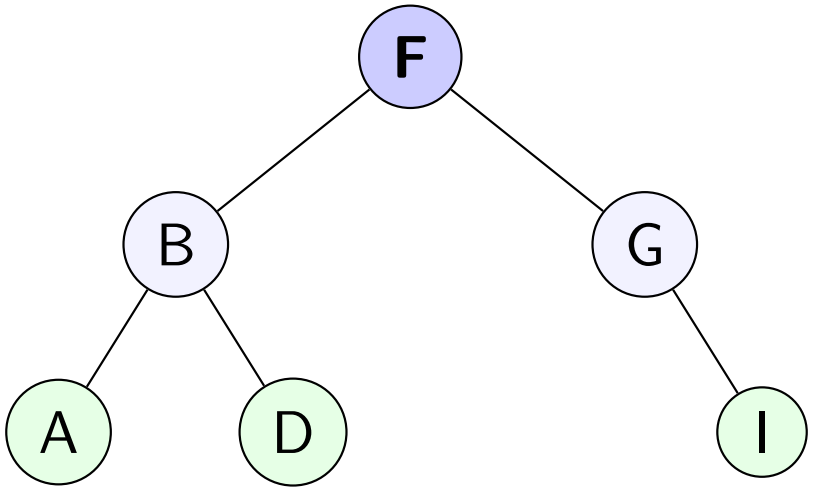
Pseudocódigo

```
FUNCION Preorden(raiz):
  SI raiz NO ES NULO:
    Imprimir(raiz.dato)
    Preorden(raiz.izq)
    Preorden(raiz.der)
```

Implementación en C

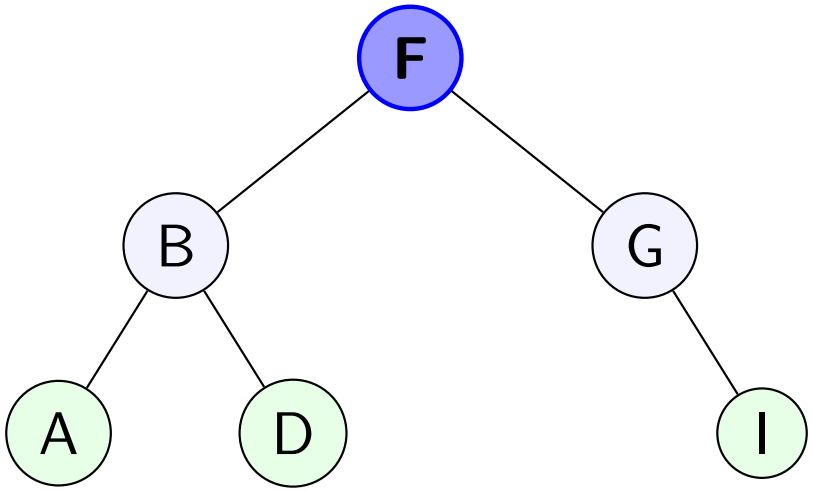
```c
void preorden(Nodo* raiz) {
    if (raiz != NULL) {
        // 1. Visita Raiz
        printf("%d ", raiz->dato);

        // 2. Recorre Izquierda
        preorden(raiz->izq);

        // 3. Recorre Derecha
        preorden(raiz->der);
    }
}
```

Animación: Recorrido Preorden (Raíz, Izq, Der)



Secuencia Preorden:

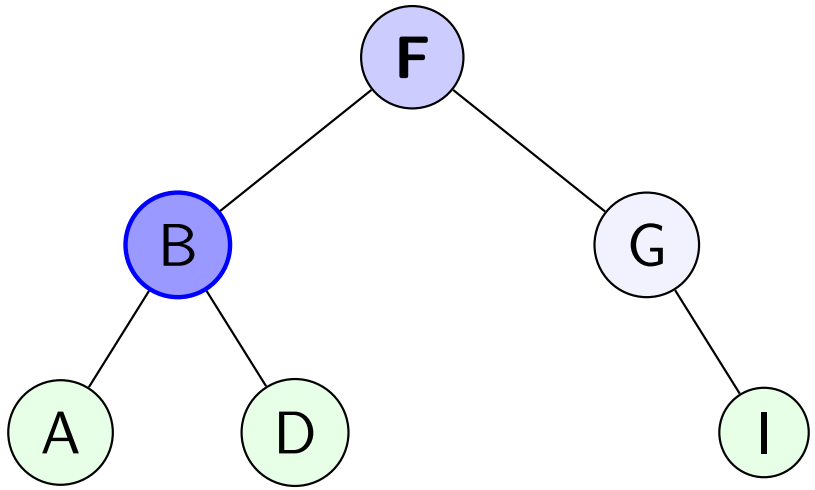
Animación: Recorrido Preorden (Raíz, Izq, Der)



Secuencia Preorden:

F

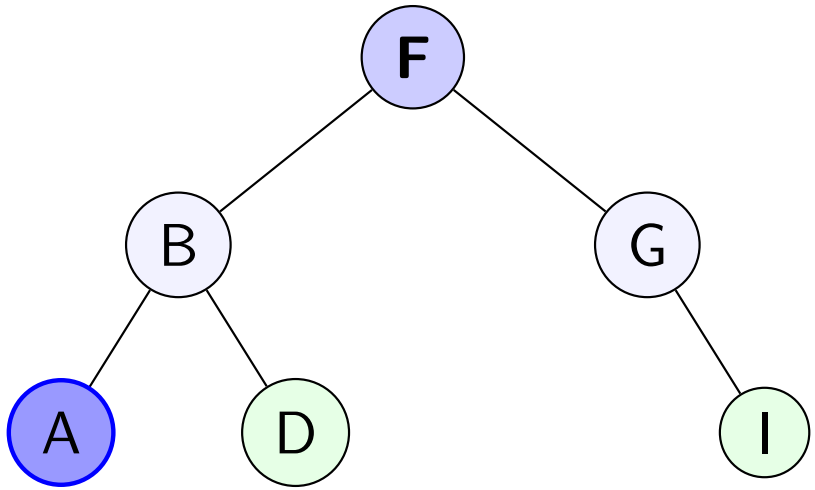
Animación: Recorrido Preorden (Raíz, Izq, Der)



Secuencia Preorden:

F B

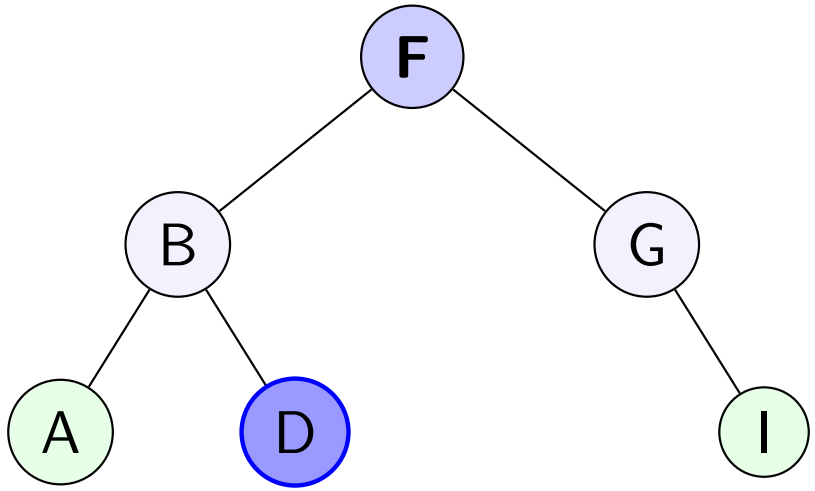
Animación: Recorrido Preorden (Raíz, Izq, Der)



Secuencia Preorden:

F B A

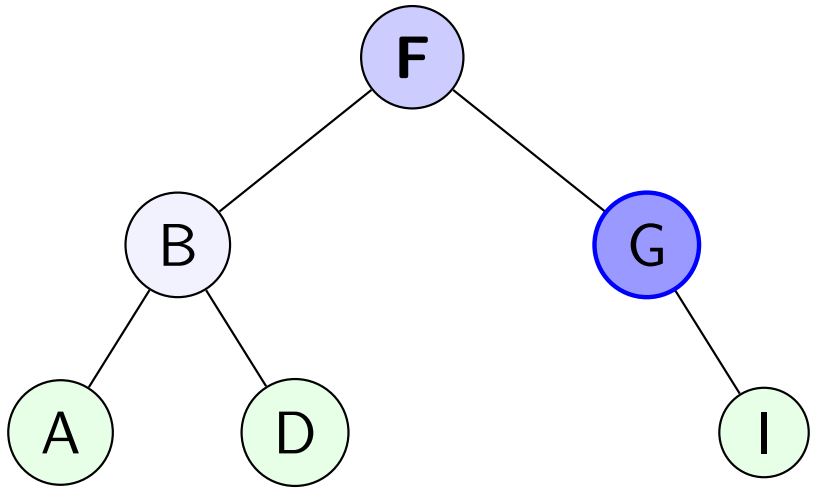
Animación: Recorrido Preorden (Raíz, Izq, Der)



Secuencia Preorden:

F B A D

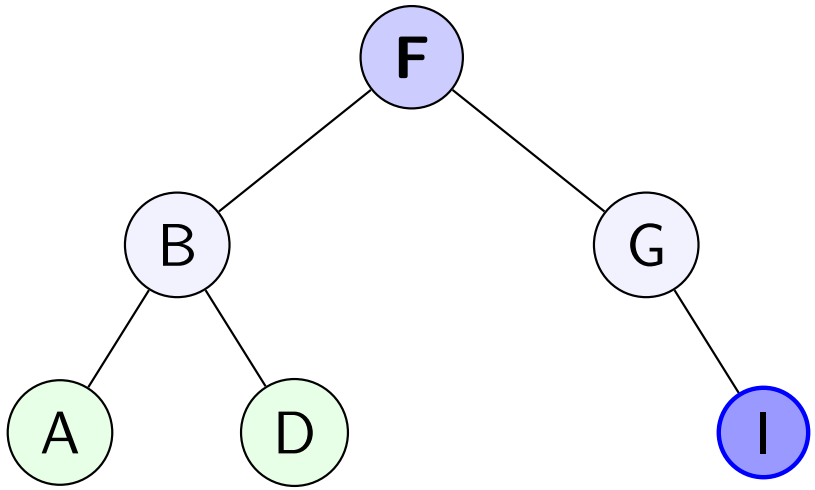
Animación: Recorrido Preorden (Raíz, Izq, Der)



Secuencia Preorden:

F B A D G

Animación: Recorrido Preorden (Raíz, Izq, Der)



Secuencia Preorden:

F B A D G I

7.5.2 Inorden (Izq, Raíz, Der)

Aplicación Práctica:
En un Árbol Binario de Búsqueda (ABB), este recorrido devuelve los valores **ordenados de menor a mayor**.

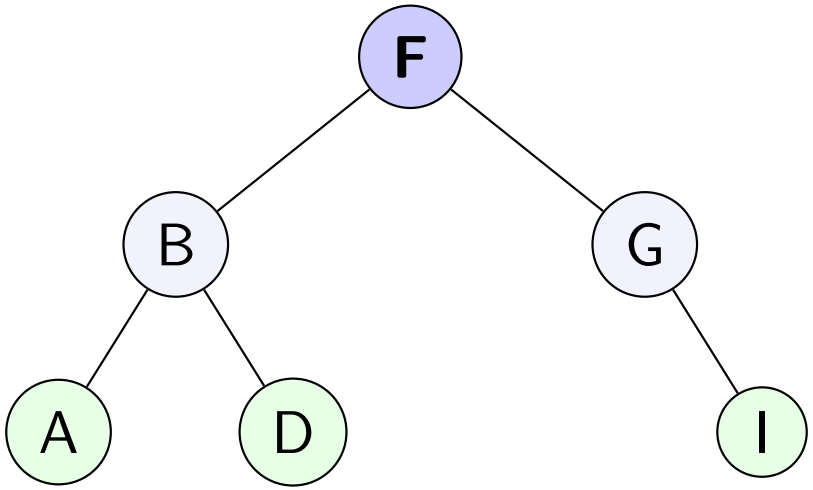
Pseudocódigo

```
FUNCION Inorden(raiz):
  SI raiz NO ES NULO:
    Inorden(raiz.izq)
    Imprimir(raiz.dato)
    Inorden(raiz.der)
```

Implementación en C

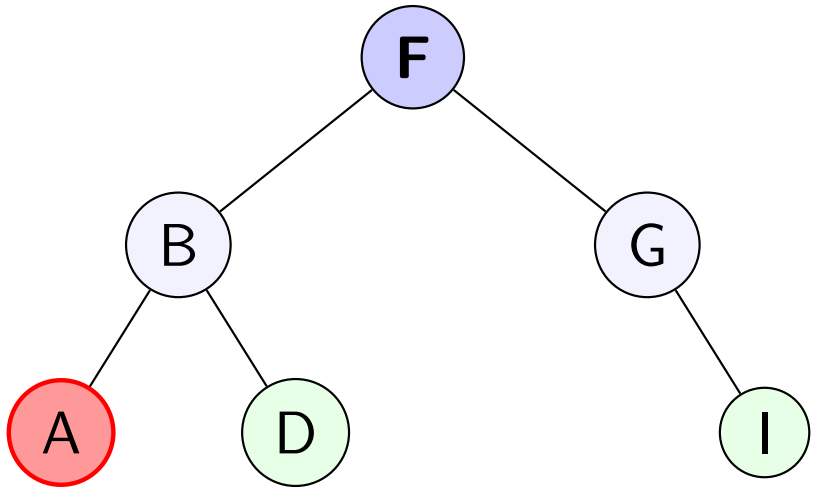
```c
void inorden(Nodo* raiz) {
    if (raiz != NULL) {
        // 1. Recorre Izquierda
        inorden(raiz->izq);

        // 2. Visita Raiz
        printf("%d ", raiz->dato);

        // 3. Recorre Derecha
        inorden(raiz->der);
    }
}
```

Animación: Recorrido Inorden (Izq, Raíz, Der)



Secuencia Inorden:

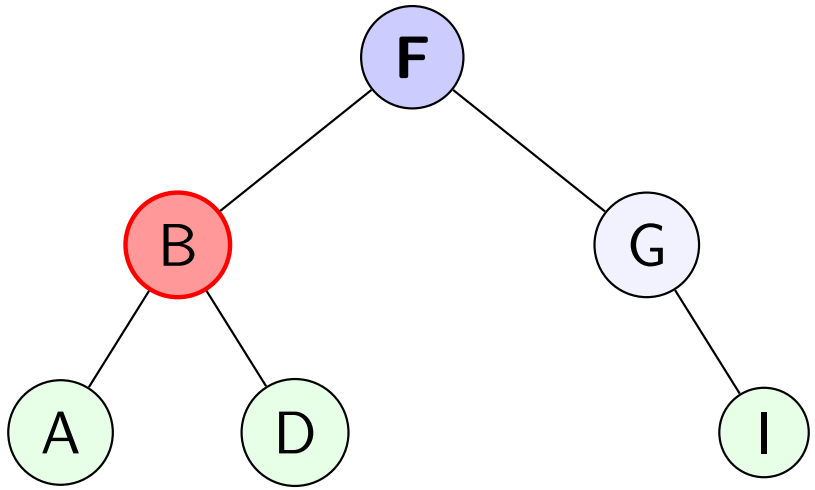
Animación: Recorrido Inorden (Izq, Raíz, Der)



Secuencia Inorden:

A

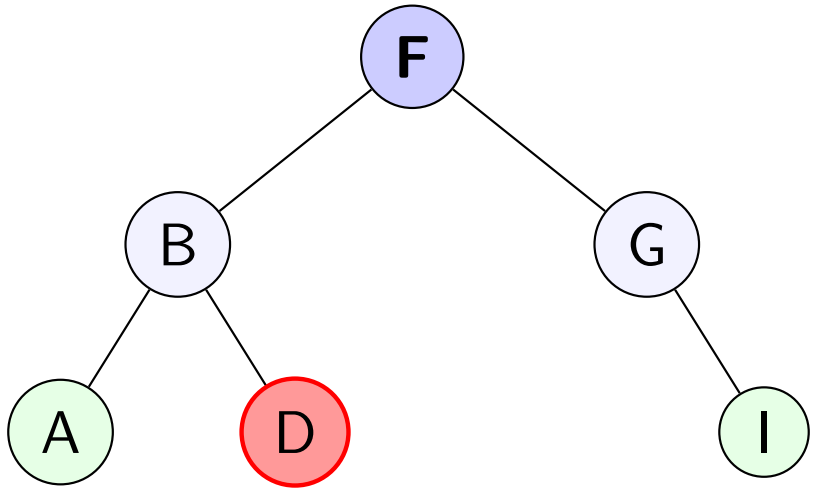
Animación: Recorrido Inorden (Izq, Raíz, Der)



Secuencia Inorden:

A B

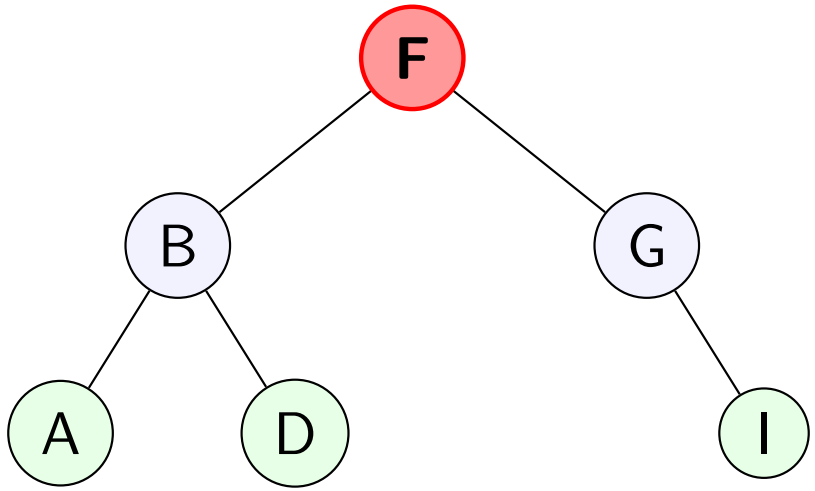
Animación: Recorrido Inorden (Izq, Raíz, Der)



Secuencia Inorden:

A B D

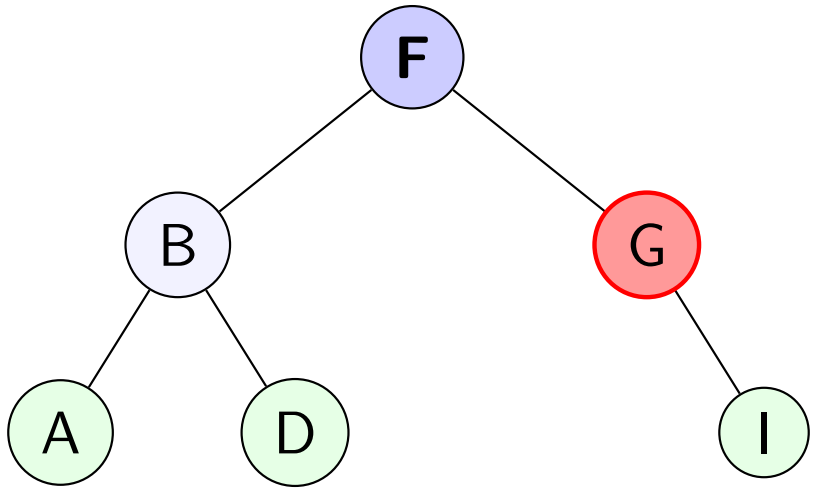
Animación: Recorrido Inorden (Izq, Raíz, Der)



Secuencia Inorden:

A B D F

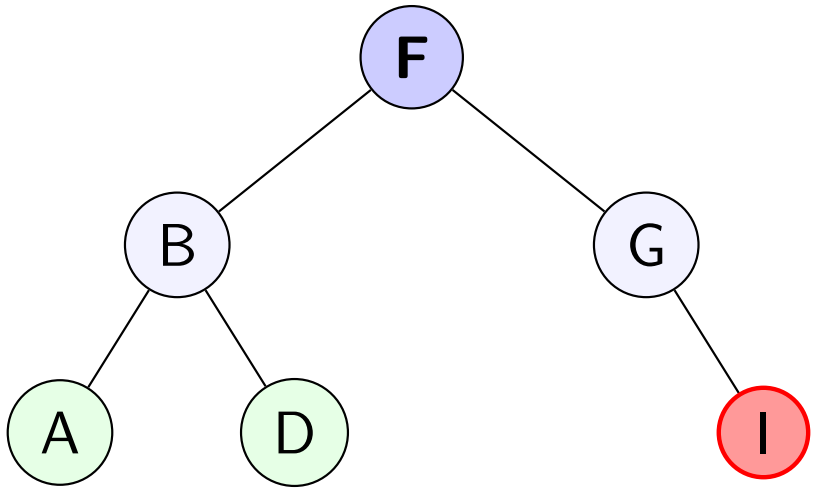
Animación: Recorrido Inorden (Izq, Raíz, Der)



Secuencia Inorden:

A B D F G

Animación: Recorrido Inorden (Izq, Raíz, Der)



Secuencia Inorden:

A B D F G I

7.5.3 Posorden (Izq, Der, Raíz)

Aplicación Práctica:
Se usa para **liberar la memoria** de un árbol (destruir nodos hijos antes que el padre) o calcular el **tamaño/altura** de subárboles.

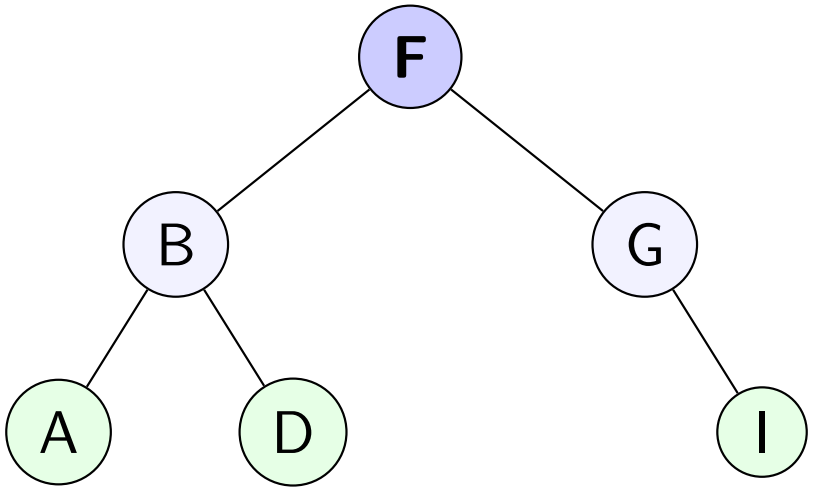
Pseudocódigo

```
FUNCION Posorden(raiz):
  SI raiz NO ES NULO:
    Posorden(raiz.izq)
    Posorden(raiz.der)
    Imprimir(raiz.dato)
```

Implementación en C

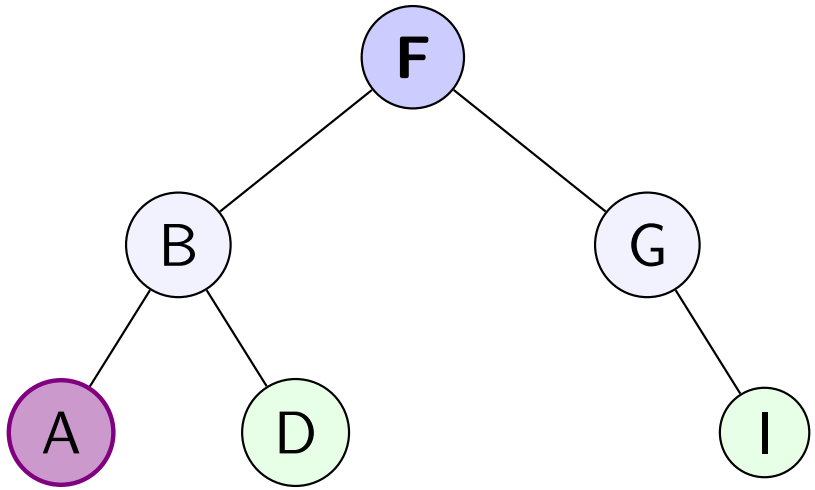
```c
void posorden(Nodo* raiz) {
    if (raiz != NULL) {
        // 1. Recorre Izquierda
        posorden(raiz->izq);

        // 2. Recorre Derecha
        posorden(raiz->der);

        // 3. Visita Raiz
        printf("%d ", raiz->dato);
    }
}
```

Animación: Recorrido Posorden (Izq, Der, Raíz)



Secuencia Posorden:

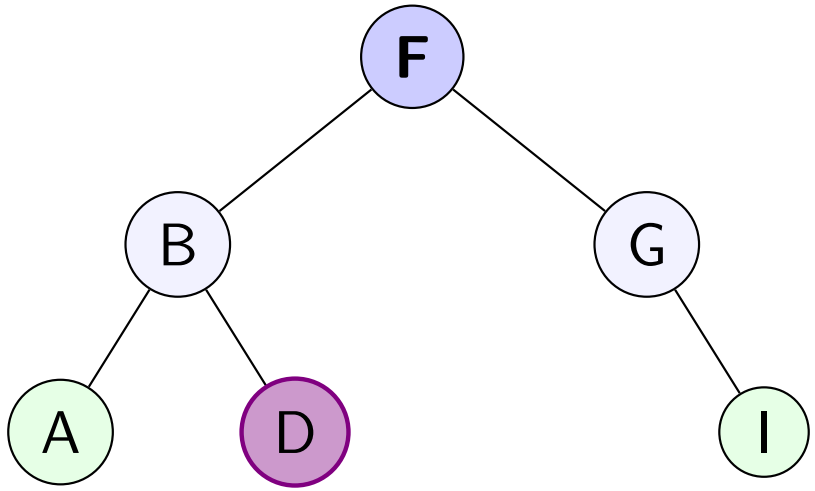
Animación: Recorrido Posorden (Izq, Der, Raíz)



Secuencia Posorden:

A

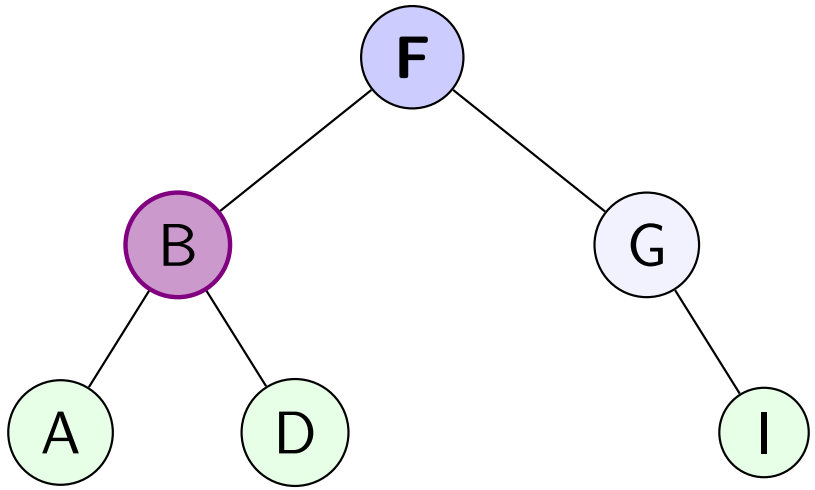
Animación: Recorrido Posorden (Izq, Der, Raíz)



Secuencia Posorden:

A D

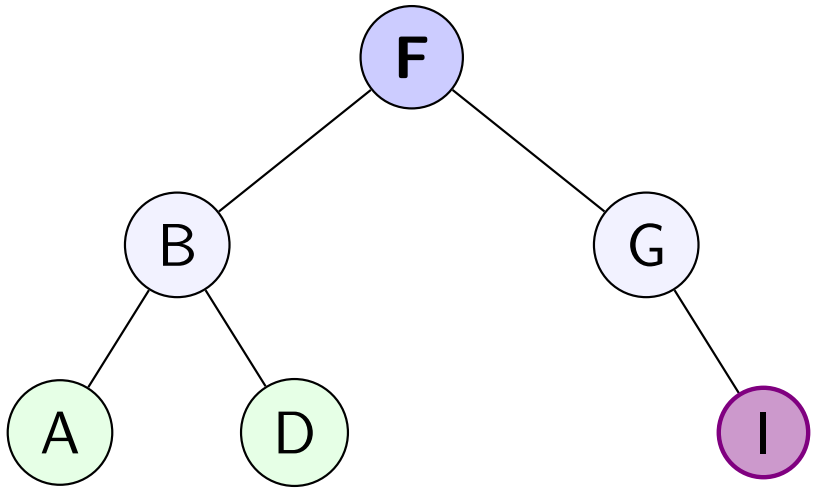
Animación: Recorrido Posorden (Izq, Der, Raíz)



Secuencia Posorden:

A D B

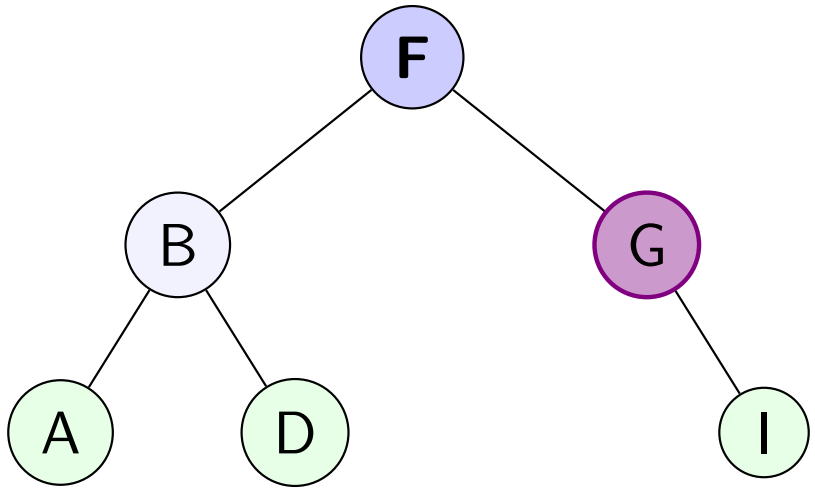
Animación: Recorrido Posorden (Izq, Der, Raíz)



Secuencia Posorden:

A D B I

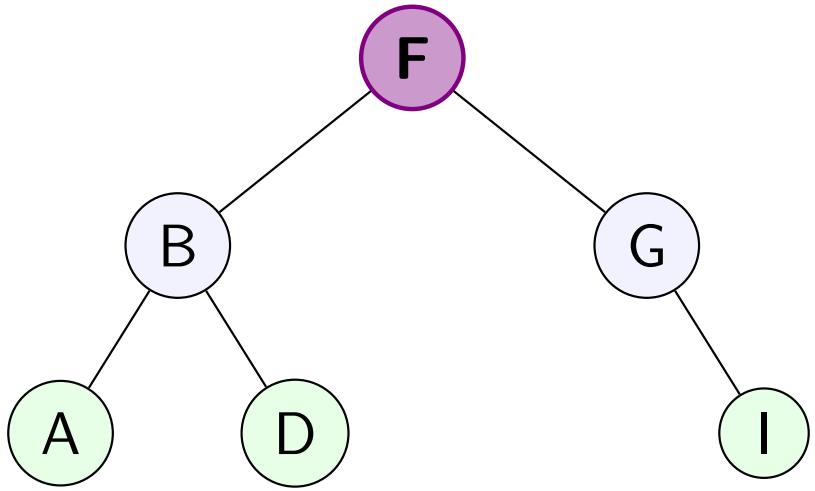
Animación: Recorrido Posorden (Izq, Der, Raíz)



Secuencia Posorden:

A D B I G

Animación: Recorrido Posorden (Izq, Der, Raíz)

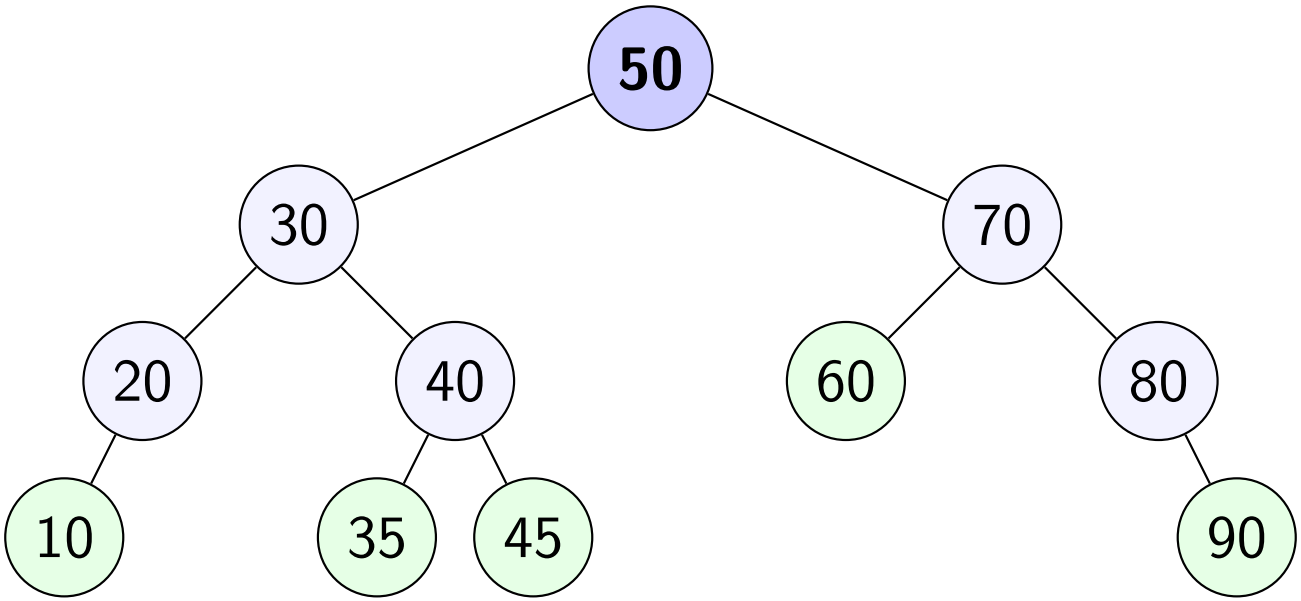


Secuencia Posorden:

A D B I G F

Ejercicio 3: Lectura de Árboles

Aplica los 3 tipos de recorridos al siguiente árbol. Escribe la secuencia de números resultante para cada uno.



Pista vital: Este árbol es un ABB. ¡Uno de los recorridos debe darte los números ordenados de menor a mayor!

Solución Ejercicio 3

- **Preorden (Raíz, Izq, Der):**
  Explora primero el lado izquierdo bajando lo más rápido posible.
  Resultado: `50, 30, 20, 10, 40, 35, 45, 70, 60, 80, 90`

- **Inorden (Izq, Raíz, Der):**
  Dado que es un Árbol Binario de Búsqueda, ¡el resultado queda ordenado de menor a mayor!
  Resultado: `10, 20, 30, 35, 40, 45, 50, 60, 70, 80, 90`

- **Posorden (Izq, Der, Raíz):**
  Procesa las hojas y sube gradualmente hacia la raíz principal.
  Resultado: `10, 20, 35, 45, 40, 30, 60, 90, 80, 70, 50`