Representación y Tipos de Árboles

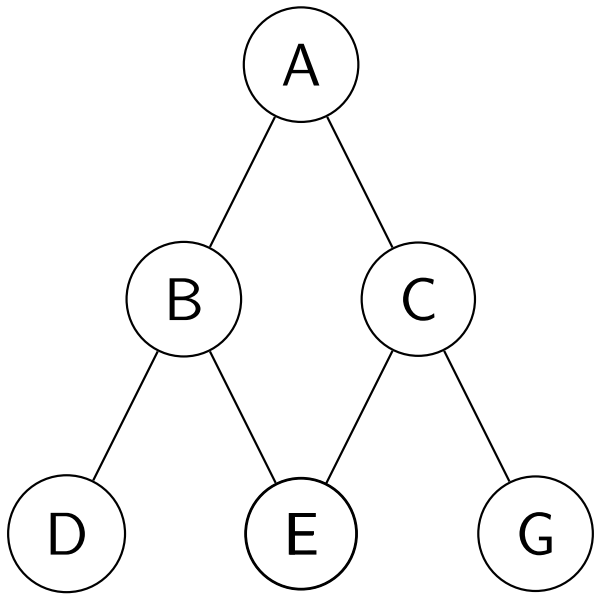
Cosijopii

Contenido

1. Representación de Árboles
2. Tipos de Árboles
   - Árbol Binario
   - Árbol Binario de Búsqueda
3. Ejemplo de BST
4. Implementación en C
5. Ejercicios Prácticos
6. Balanceo en Árboles Binarios
7. Introducción a los Árboles Rojo-Negro
8. Algoritmos Fundamentales
9. Ejercicios
   - Árbol B
10. Ejercicios
11. Conclusión y Recomendaciones

Árboles: Definición y Conceptos Básicos

- Un **árbol** es una estructura de datos jerárquica formada por nodos.
- Características principales:
  - Cada nodo tiene un valor y puede tener "hijos".
  - Existe un nodo especial llamado **raíz**.
  - Los nodos sin hijos se denominan **hojas**.
- Ejemplo de un **árbol**:



Representación de Árboles en Computadoras

- **Representación como nodos enlazados:**

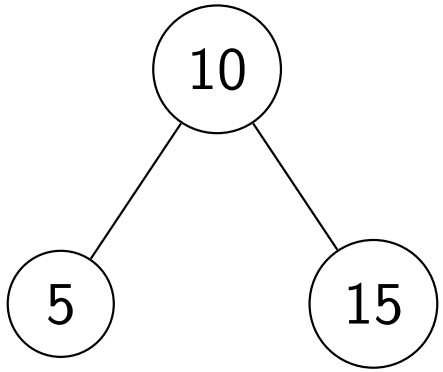
```
1 struct Nodo {
2     int valor;
3     struct Nodo *izquierdo;
4     struct Nodo *derecho;
5 };
```

Listing 1: Estructura de un nodo de árbol

- **Representación como arreglos:**
  - Los árboles binarios pueden representarse en un arreglo con **hijos en posiciones relacionadas**.
    Ejemplo: Nodo en posición i:
    - Hijo izquierdo: $2i + 1$
    - Hijo derecho: $2i + 2$

Árbol Binario

- Cada nodo tiene a lo sumo dos hijos.
- Representación visual:



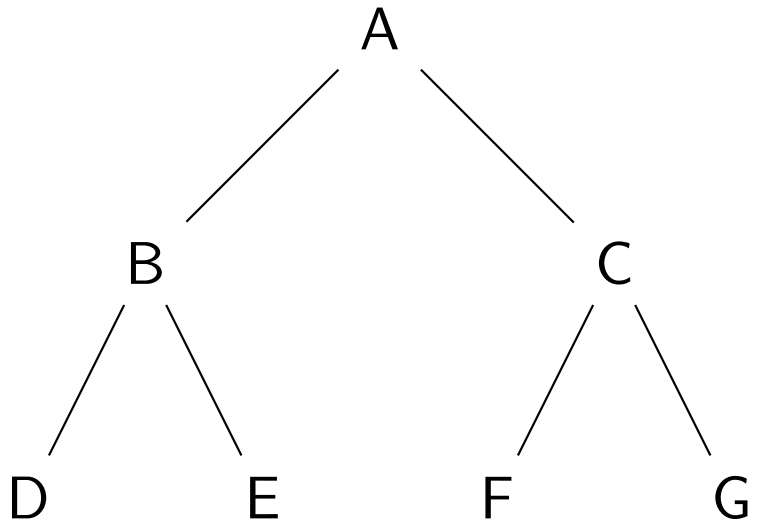
Propiedades:

- Altura del árbol es el número de niveles.
- Árboles completos y completos balanceados tienen aplicación en sistemas.

Recorridos en Árboles

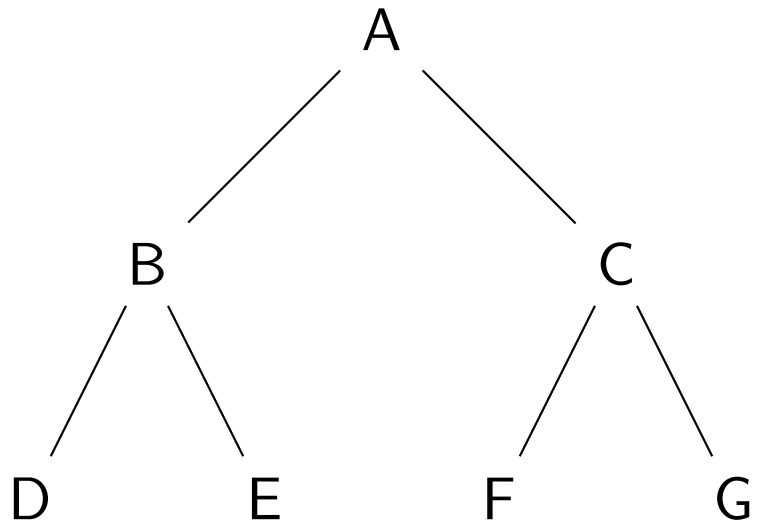
Un recorrido en un árbol es una forma sistemática de visitar todos los nodos del árbol.

- **Preorden:** Raíz, izquierda, derecha.
- **Inorden:** Izquierda, raíz, derecha.
- **Postorden:** Izquierda, derecha, raíz.



Recorrido en Preorden I

En preorden visitamos primero la raíz, luego el subárbol izquierdo y



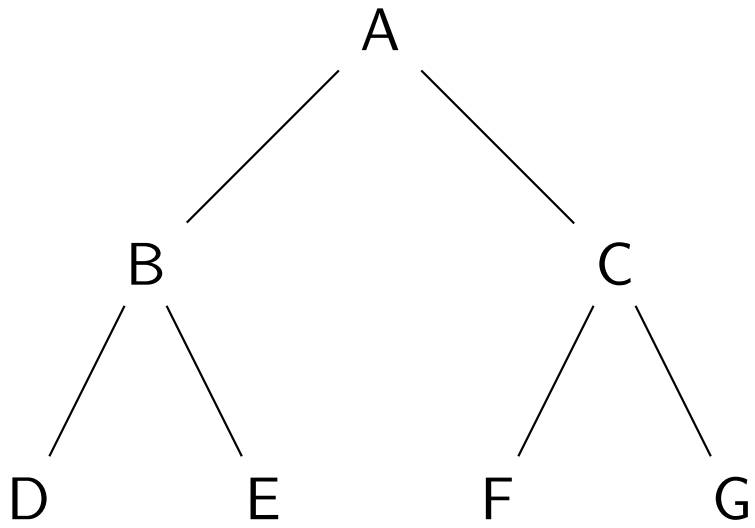
finalmente el derecho. Preorden: A, B, D, E, C, F, G

Recorrido en Preorden II

```
void preorden(struct Nodo* nodo) {
    if (nodo != NULL) {
        printf("%d-", nodo->valor);
        preorden(nodo->izquierdo);
        preorden(nodo->derecho);
    }
}
```

Recorrido en Inorden I

En inorden visitamos primero el subárbol izquierdo, luego la raíz y



finalmente el derecho.

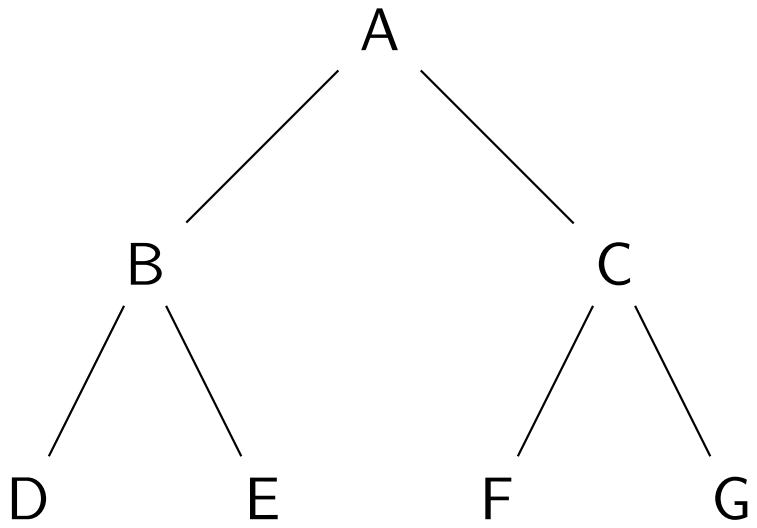
Inorden: D, B, E, A, F, C, G

Recorrido en Inorden II

```
void inorden(struct Nodo* nodo) {
    if (nodo != NULL) {
        inorden(nodo->izquierdo);
        printf("%d-", nodo->valor);
        inorden(nodo->derecho);
    }
}
```

Recorrido en Postorden I

En postorden visitamos primero el subárbol izquierdo, luego el derecho y



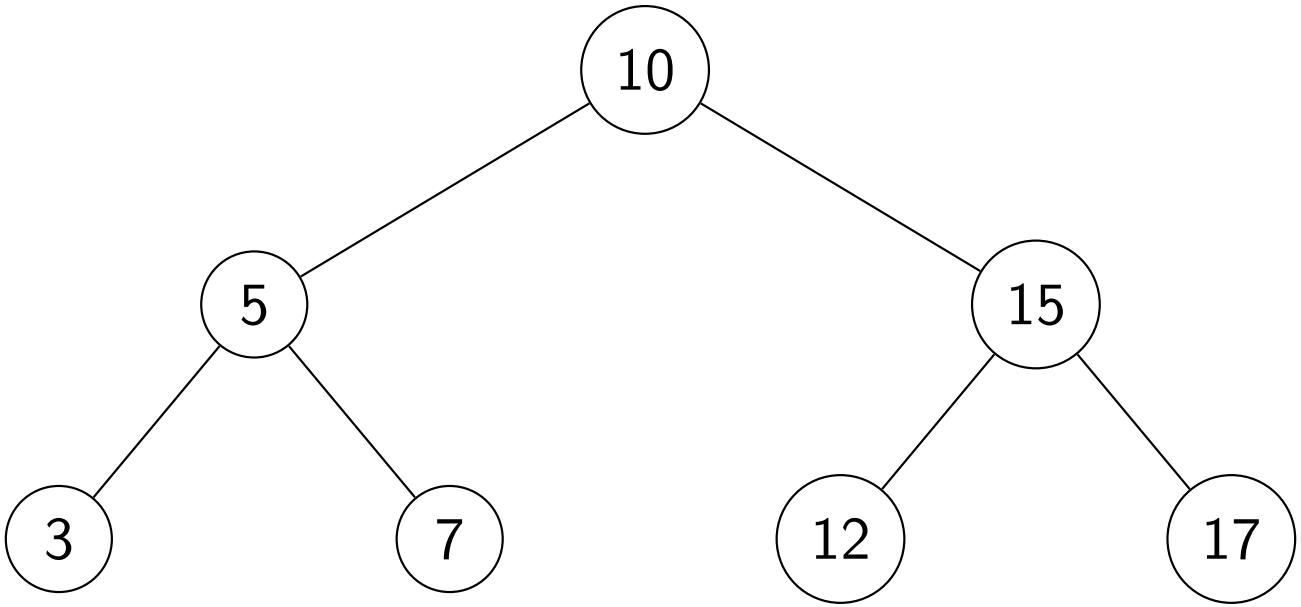
finalmente la raíz. Postorden: D, E, B, F, G, C, A

Recorrido en Postorden II

```
void postorden(struct Nodo* nodo) {
    if (nodo != NULL) {
        postorden(nodo->izquierdo);
        postorden(nodo->derecho);
        printf("%d-", nodo->valor);
    }
}
```

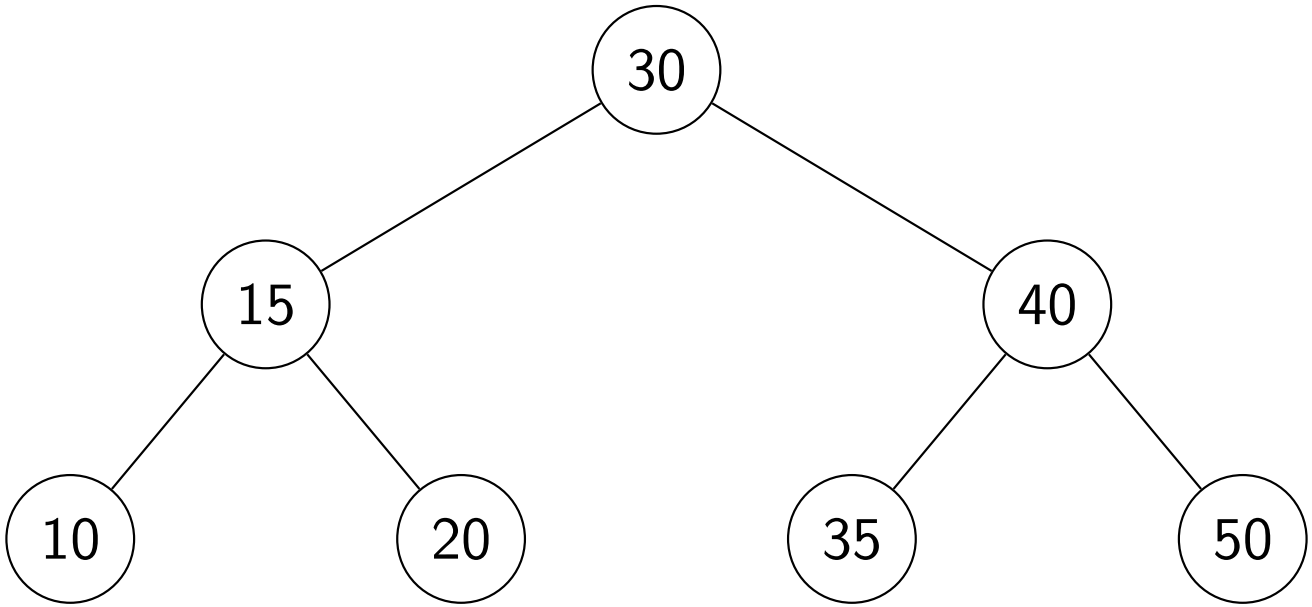
Árbol Binario de Búsqueda (BST)

- Es un **árbol binario** con las siguientes propiedades:
  - Todos los nodos en el subárbol izquierdo son menores que la raíz.
  - Todos los nodos en el subárbol derecho son mayores que la raíz.
- Operaciones fundamentales:
  - Inserción
  - Búsqueda
  - Eliminación
- Ejemplo:



Representación Gráfica de un BST

- Dado el siguiente conjunto de datos: $\{30, 15, 40, 10, 20, 35, 50\}$
- Construimos el **árbol binario de búsqueda**:



Estructura de un Nodo en C I

Los nodos en un **árbol binario de búsqueda** se definen como estructuras.

Listing 2: Estructura de Nodo

```c
#include <stdio.h>
#include <stdlib.h>

struct Nodo {
        int valor;
        struct Nodo* izquierdo;
        struct Nodo* derecho;
};

// Crear un nuevo nodo
struct Nodo* nuevoNodo(int valor) {
```

Estructura de un Nodo en C II

```
    struct Nodo* nodo = (struct Nodo*)
    malloc(sizeof(struct Nodo));
    nodo->valor = valor;
    nodo->izquierdo = NULL;
    nodo->derecho = NULL;
    return nodo;
}
```

Inserción en un BST

La función de inserción respeta las reglas del **árbol binario de búsqueda**.

Listing 3: Función de Inserción

```
struct Nodo* insertar(struct Nodo* nodo, int valor) {
        if (nodo == NULL) {
                return nuevoNodo(valor);
        }
        if (valor < nodo->valor) {
                nodo->izquierdo =
          insertar(nodo->izquierdo, valor);
        } else if (valor > nodo->valor) {
                nodo->derecho =
          insertar(nodo->derecho, valor);
        }
        return nodo;
}
```

Búsqueda en un BST

La búsqueda sigue un camino desde la raíz hasta un nodo hoja.

Listing 4: Función de Búsqueda

```
int buscar(struct Nodo* nodo, int valor) {
    if (nodo == NULL) {
        return 0; // No encontrado
    }
    if (valor == nodo->valor) {
        return 1; // Encontrado
    }
    if (valor < nodo->valor) {
        return buscar(nodo->izquierdo, valor);
    } else {
        return buscar(nodo->derecho, valor);
    }
}
```

Ejercicio 1: Construcción de un BST

- Dado el conjunto de valores: $\{25, 15, 50, 10, 22, 35, 70\}$
- Construya un **árbol binario de búsqueda**.

Preguntas:

1. ¿Cuál es la altura del árbol?
2. ¿Cuáles son los nodos hoja?

Ejercicio 2: Inserción y Búsqueda

- Usando el conjunto de valores del ejercicio anterior:
  1. Inserte el valor **27** en el BST.
  2. Realice una búsqueda del valor **35**.

Ejercicio 3: Árbol Binario de Búsqueda

Construye un **árbol binario de búsqueda (BST)** con los siguientes valores:

$$30,\ 15,\ 40,\ 10,\ 20,\ 35,\ 50$$

Preguntas:

1. ¿Cómo se ve el BST después de insertar todos los valores?
2. Realiza una búsqueda en el BST para encontrar el valor 35. ¿Cuántos pasos necesitas?
3. Elimina el nodo con el valor 15. ¿Cómo queda el BST?

¿Qué es el Balanceo de un Árbol Binario?

- El **balanceo de un árbol binario** asegura que las operaciones de búsqueda, inserción y eliminación sean eficientes ($O(\log n)$).
- En un árbol **no balanceado**, su estructura puede degenerar en una lista vinculada, con tiempo de búsqueda $O(n)$.

Definición de Árbol Balanceado:

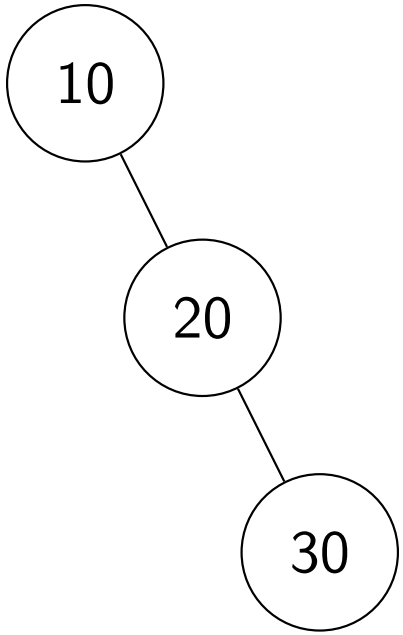
- Un árbol está balanceado si la diferencia de alturas entre los subárboles izquierdo y derecho de cada nodo es menor o igual a 1.

Técnicas de Balanceo

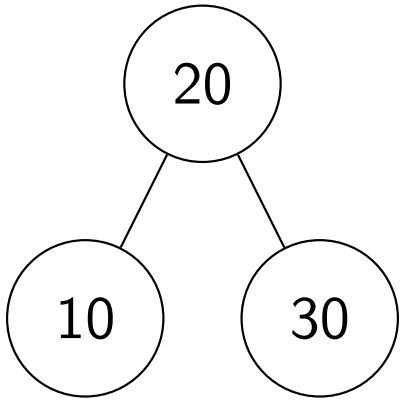
- **Rotaciones:** Ajustan la estructura del árbol para redistribuir los nodos.
  - **Rotación Izquierda:** Se aplica cuando un subárbol derecho es más alto.
  - **Rotación Derecha:** Se aplica cuando un subárbol izquierdo es más alto.
- **Re-equilibrio:** Después de insertar o eliminar nodos, se recalculan alturas y se aplican rotaciones si es necesario.
- **Árbol AVL:** Es un árbol binario de búsqueda donde la diferencia de alturas entre subárboles no supera 1.

Ejemplo de Rotaciones

Antes de Rotación Izquierda:



Después de Rotación Izquierda:

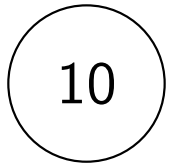


Balanceo en Árboles AVL I

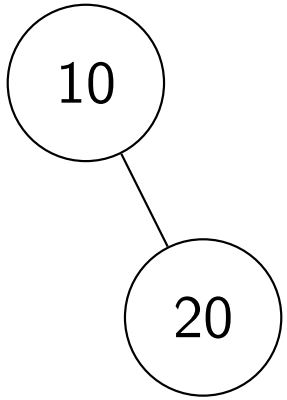
Ejemplo: Inserción Secuencial en AVL

- Insertar $10, 20, 30$:

1. **Insertamos 10:**

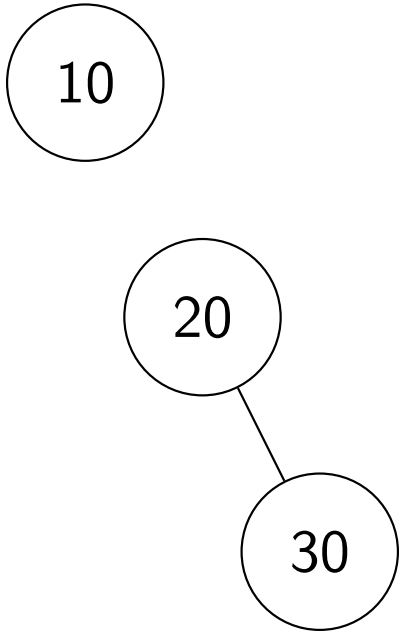


2. **Insertamos 20:**

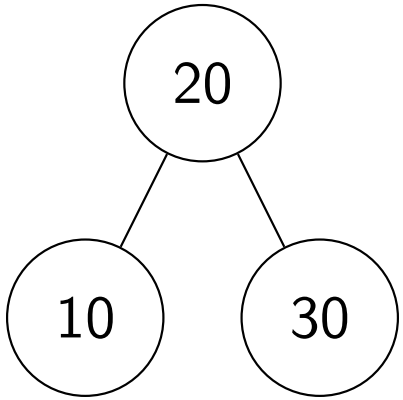


3. **Insertamos 30 (Desbalance):**

Balanceo en Árboles AVL II



4. Después de Rotación Izquierda:

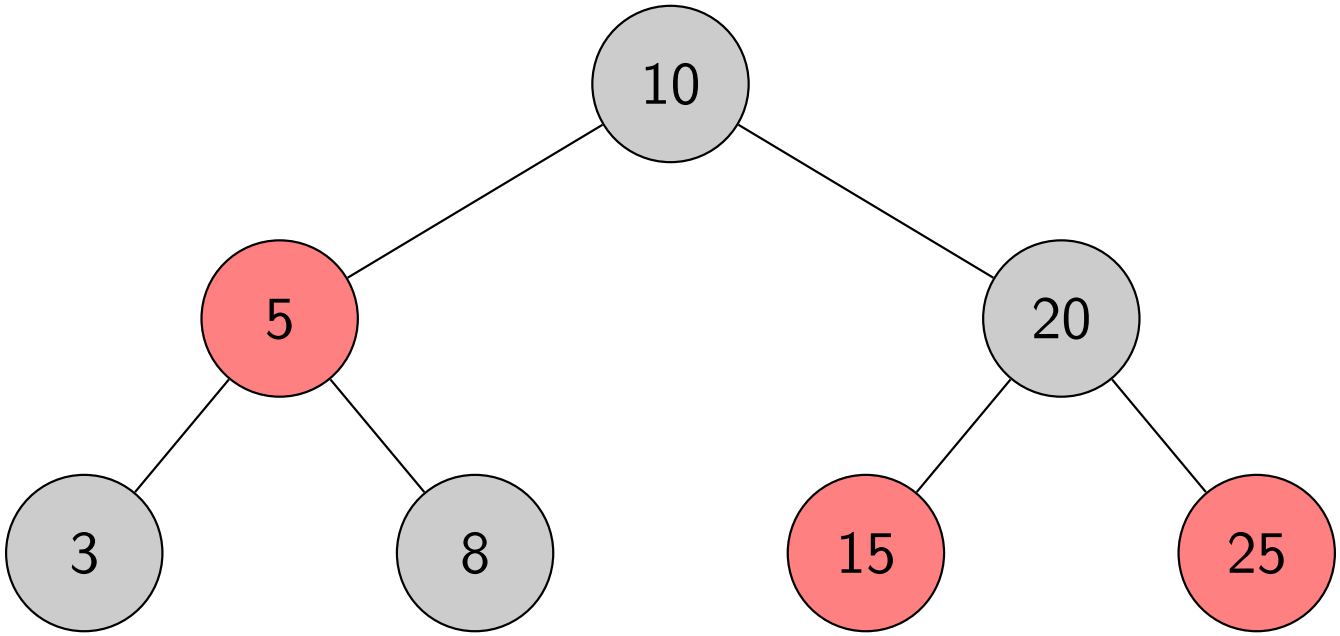


¿Qué es un Árbol Rojo-Negro?

- Un **árbol rojo-negro** es un tipo de **árbol binario de búsqueda balanceado** que asegura un tiempo logarítmico para las operaciones de búsqueda, inserción y eliminación.
- Se utiliza ampliamente en estructuras de datos como `std::map` y `std::set` en C++ y en sistemas de bases de datos.
- **Propiedades fundamentales:**
  - Cada nodo es **rojo** o **negro**.
  - La raíz del árbol siempre es **negra**.
  - Los nodos **rojos** no pueden tener hijos rojos (no hay dos nodos rojos consecutivos).
  - Todo camino desde un nodo hasta sus hojas contiene el mismo número de nodos negros.

Ejemplo de Árbol Rojo-Negro

- Representación de un árbol rojo-negro:



Propiedades en Detalle

- **Propiedad 1:** Cada nodo es rojo o negro.
- **Propiedad 2:** La raíz del árbol siempre es negra.
- **Propiedad 3:** Los nodos rojos no pueden tener hijos rojos.
- **Propiedad 4:** Todo camino desde un nodo hasta una hoja nula contiene el mismo número de nodos negros (**balanceo negro**).

Consecuencia: Estas reglas aseguran que el árbol esté aproximadamente balanceado, limitando la altura máxima del árbol a $2\log(n+1)$.

Algoritmo: Inserción en Árbol Rojo-Negro I

- Insertar un nodo en un árbol rojo-negro requiere realizar **rotaciones** y **recoloreos** para preservar las propiedades del árbol.

```
struct Nodo {
    int valor;
    char color; // 'R' para rojo, 'N' para negro
    struct Nodo *izq, *der, *padre;
};

void insertar(struct Nodo** raiz, int valor) {
    // 1. Realizar inserci n est ndar de BST
    struct Nodo* nuevo = crearNodo(valor);
    *raiz = bstInsert(*raiz, nuevo);

    // 2. Arreglar propiedades rojo-negro
    corregirInsercion(raiz, nuevo);
```

Algoritmo: Inserción en Árbol Rojo-Negro II

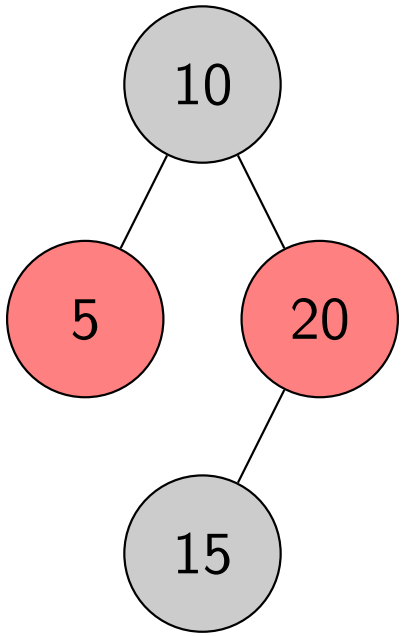
```
}

void corregirInsercion(struct Nodo** raiz, struct N
    // L gica para rotaciones y recoloreos
}
```

Listing 5: Código en C para insertar en un Árbol Rojo-Negro

Ejemplo de Inserción

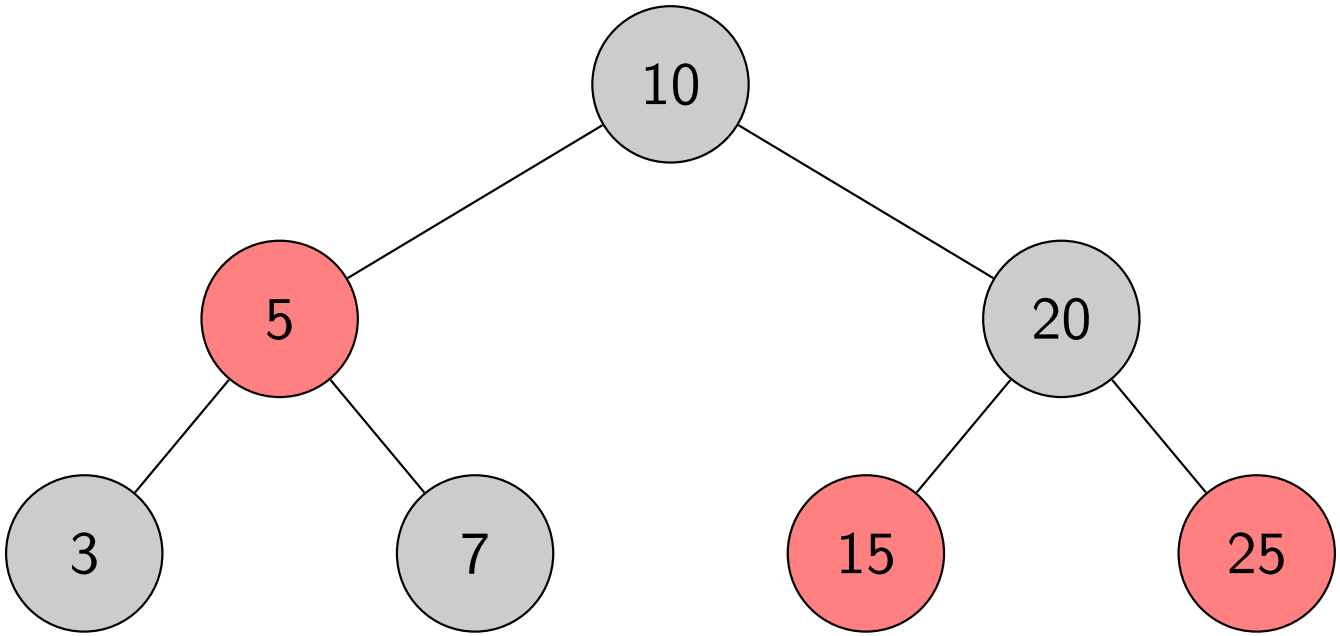
- Supongamos que queremos insertar el valor 12 en el árbol:



Resultado final: El árbol después de balancear las propiedades rojo-negro tendrá 12 como hijo de 10.

Ejercicio 1: Propiedades Rojo-Negro

- Verifica si el siguiente árbol cumple con las propiedades de los árboles rojo-negro:



Pregunta: ¿Cuántos nodos negros hay en cada camino desde la raíz hasta una hoja?

Árbol B

- Estructura de datos auto-balanceada que mantiene datos ordenados.
- Comúnmente utilizada en sistemas de archivos y bases de datos.
- Características:
  - Todos los nodos hoja están al mismo nivel.
  - Cada nodo interno puede tener entre $m/2$ y m hijos, donde m es el orden del árbol.

Ejercicio 3: Árbol Rojo-Negro

Dado un conjunto de valores, construye un **Árbol Rojo-Negro** siguiendo las reglas de balanceo:

$$10,\ 20,\ 30,\ 15,\ 25,\ 5$$

Preguntas:

1. ¿Cuál es el color de la raíz?
2. ¿Qué ocurre cuando insertas 35 en este árbol?
3. Explica cómo se mantiene el balanceo después de las inserciones.

Ejercicio 4: Árbol B

Construye un **Árbol B** de orden 3 con los siguientes valores:

$$5,\ 10,\ 15,\ 20,\ 25,\ 30,\ 35,\ 40$$

Preguntas:

1. Muestra el árbol después de insertar todos los valores.
2. ¿Cuántos niveles tiene el árbol?
3. Inserta el valor 45 y describe los cambios que ocurren en el árbol.

Conclusión

- Los árboles son estructuras de datos fundamentales con múltiples aplicaciones en informática.
- Cada tipo de árbol tiene propiedades únicas que los hacen útiles para problemas específicos.
- Es importante entender las reglas y operaciones básicas de cada tipo para utilizarlos de manera eficiente.

Recomendaciones

- Practica construyendo diferentes tipos de árboles con conjuntos de datos variados.
- Implementa algoritmos de inserción, eliminación y búsqueda en código para comprender mejor su funcionamiento.
- Explora cómo los árboles se utilizan en sistemas reales como bases de datos y sistemas de archivos.