

# Arboles Binarios

PROGRAMACIÓN II

Ing. Tomás Montero Swinnen

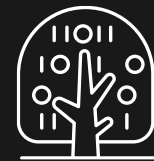
UADE

# Agenda



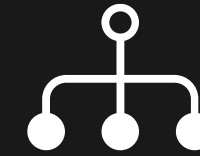
## Repaso

Revisión rápida de conceptos previos para refrescar conocimientos.



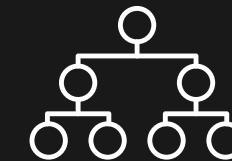
## Arboles Binarios

Árboles en los que cada nodo puede tener como máximo dos hijos.



## Arboles

- Estructuras jerárquicas que consisten en nodos interconectados.



## ABBTDA

Tipo de dato abstracto que representa un árbol binario con una propiedad de orden



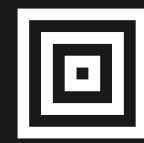
## TDA

Provee una interfaz con la cuál es posible realizar las operaciones permitidas



## Estructuras Dinámicas

Conjunto de nodos con datos y un apuntador al siguiente nodo.



## Recursividad

Funciones que se llaman a sí mismas, evitando el uso de bucles y otros iteradores.

# Repaso

# TDA

## Tipos de datos abstractos

### Importancia de repasar TDA

**Especificación:** Define las características, el contrato que establece cómo se puede interactuar con el TDA.

**Implementación:** Forma de llevar a cabo lo definido por un TDA.

- Puede haber diferentes implementaciones para el mismo TDA.
- Determina cómo se estructuran y manipulan los datos en la implementación específica.

**Clase que utiliza el TDA:** (usuario)

- Utiliza el TDA como una herramienta para resolver problemas.
- Realiza operaciones y utiliza las propiedades definidas en el TDA para alcanzar los objetivos.

# Importancia de repasar Recursividad

Es una técnica fundamental para resolver problemas en Árboles Binarios.

La recursividad se adapta naturalmente a la estructura de un Árbol Binario.

- Simplifica el código y lo hace más legible y mantenible.
- Permite realizar recorridos y búsquedas de manera más elegante y concisa.

## Caso típico de uso: Factorial

$$n! = \begin{cases} \text{si } n = 0 & \Rightarrow 1 \\ \text{si } n \geq 1 & \Rightarrow n (n - 1)! \end{cases}$$

```
public int factorial (double numero) {  
    if (numero==0)  
        return 1;  
    else  
        return numero * factorial(numero-1);  
}
```

# Recursividad



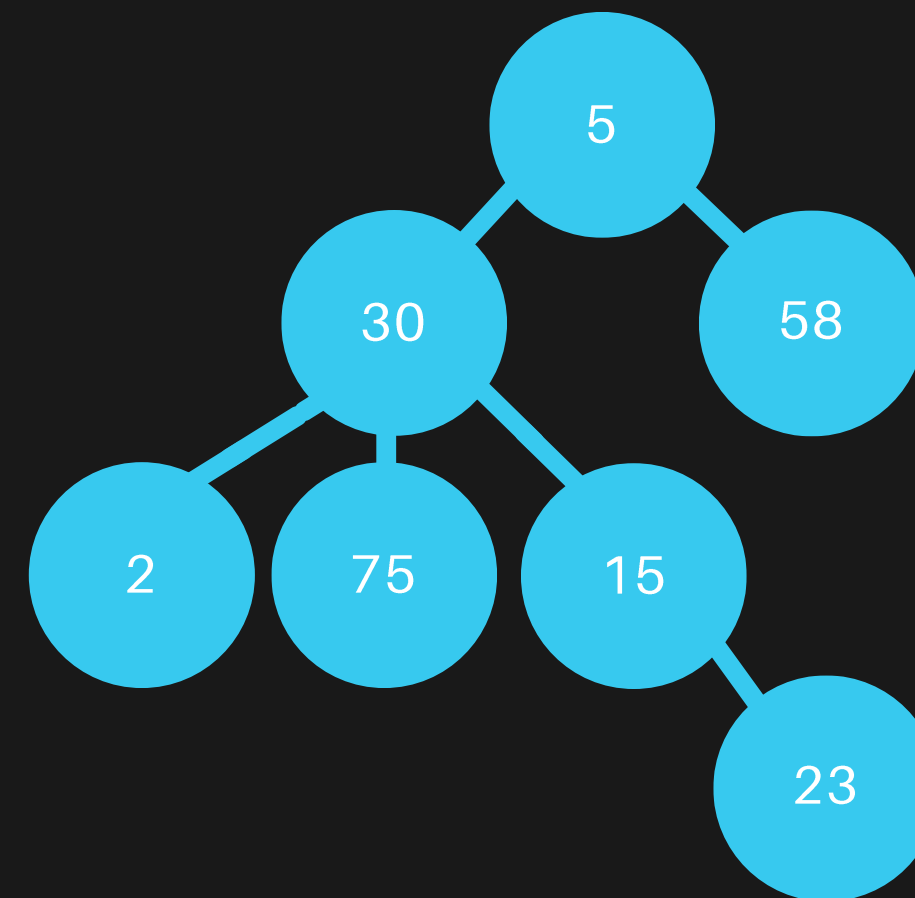


# Árboles

## Árbol:

Estructura jerárquica que se compone de nodos interconectados.

- Conjunto de **nodos** y **enlaces** (aristas) que los conectan de forma jerárquica.
- Cada nodo se encuentra en un **nivel** (profundidad).

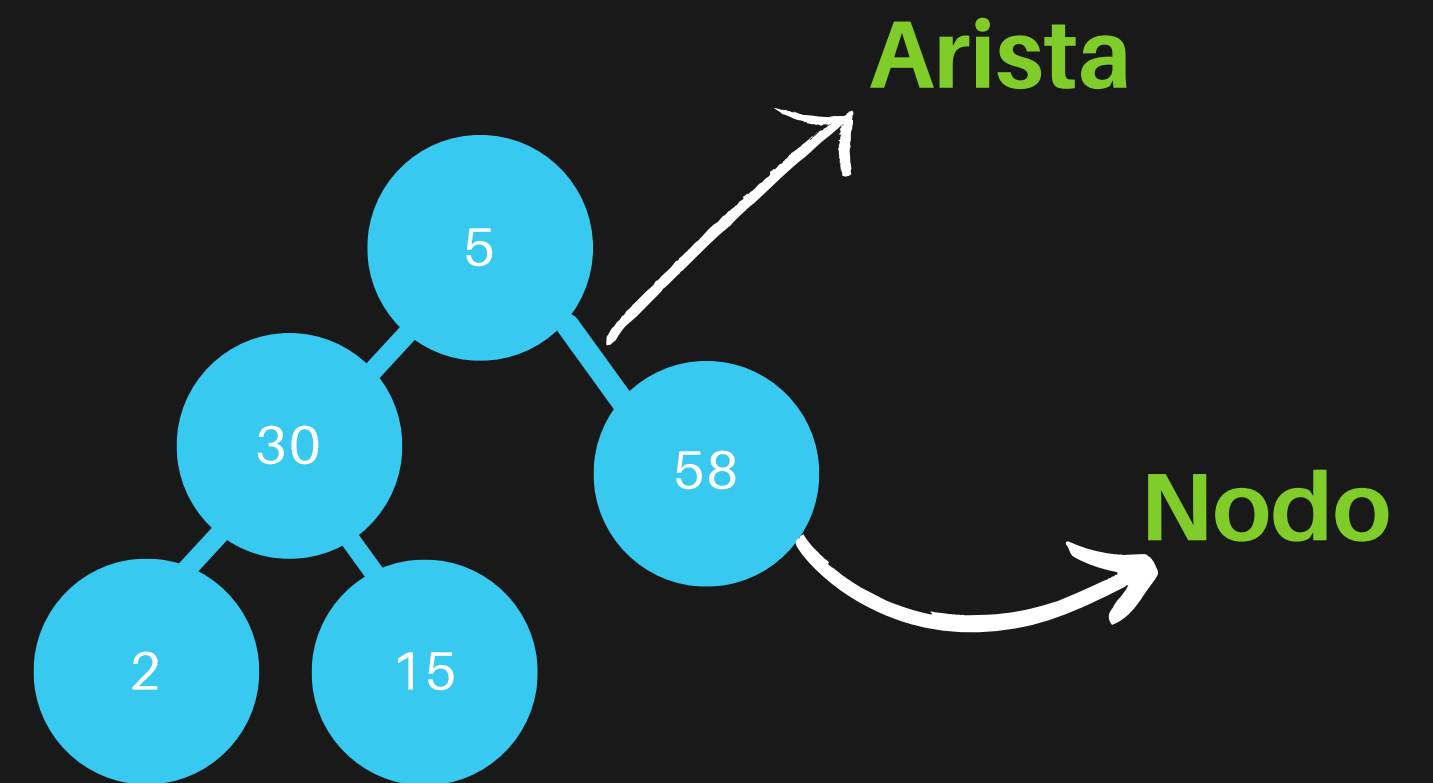




# Árboles

## Nodos y Aristas

- **Nodo**: Almacena datos
- **Arista**: Conecta nodos

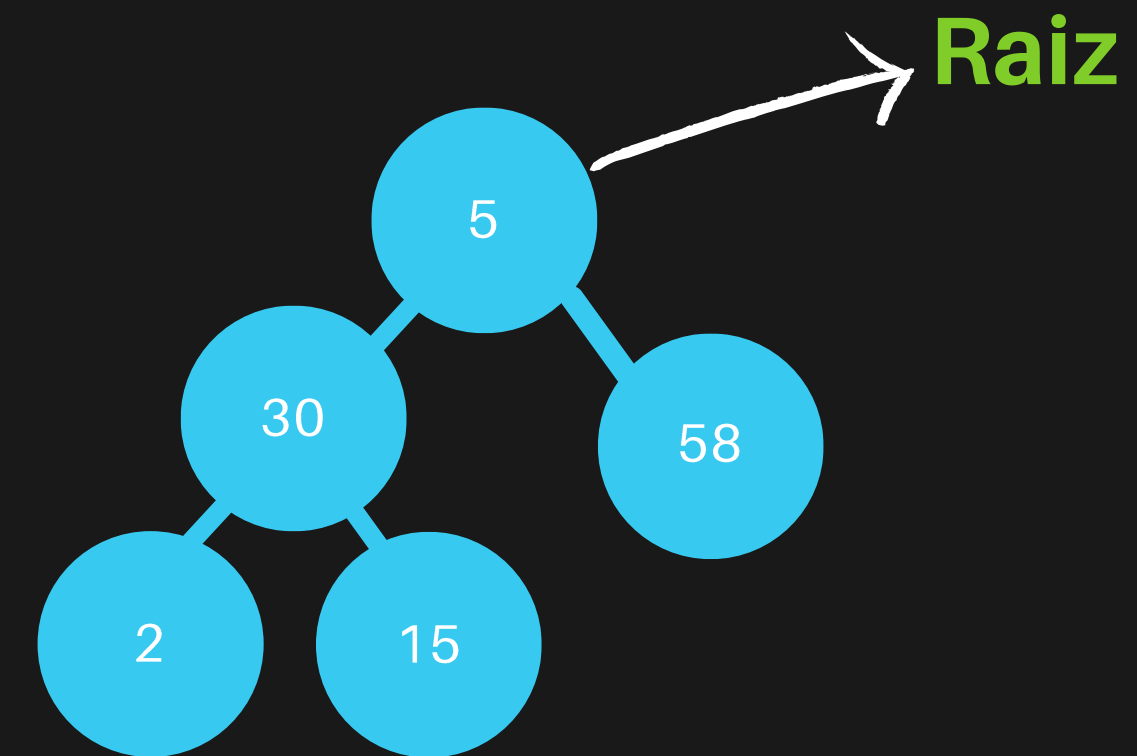




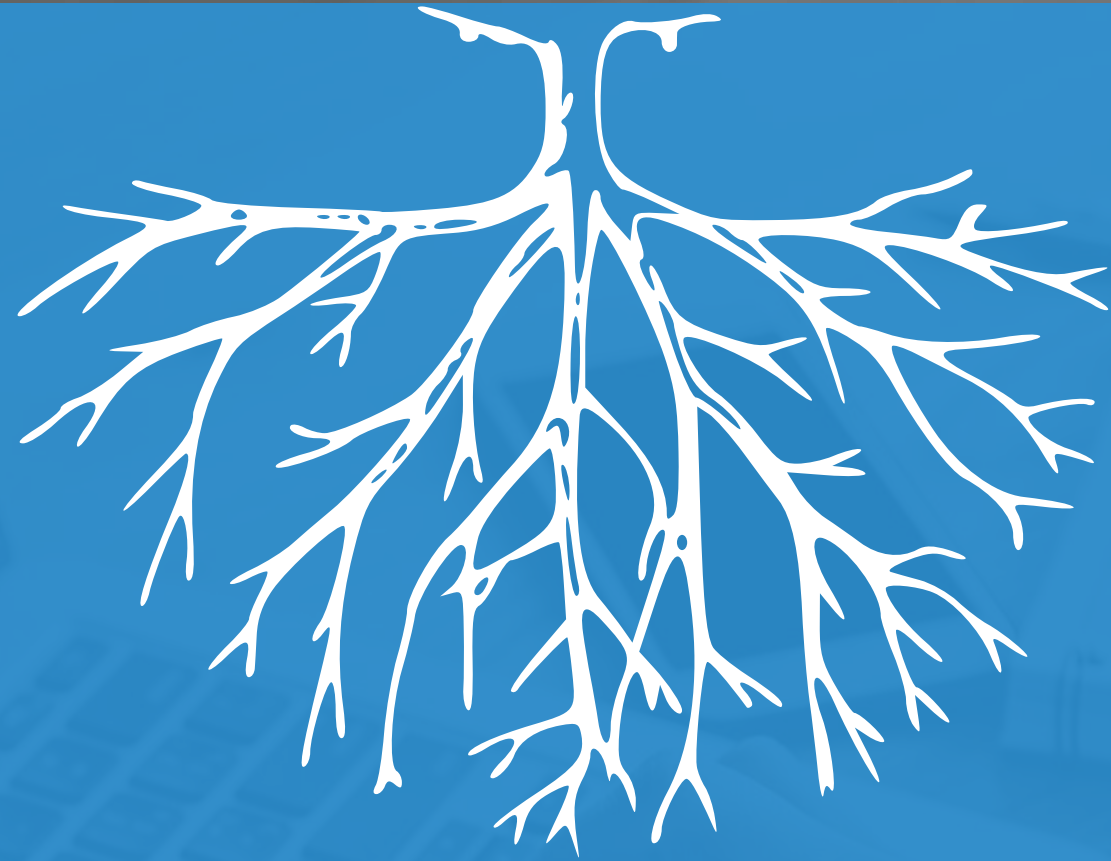
# Árboles

## Raiz

- **Nodo raíz:**
  - Nodo superior en el árbol, que no tiene un padre.
  - Es el punto de partida para recorrer o acceder a otros nodos.
  - Un nodo cualquiera es el origen de un subárbol, y a su vez raíz del mismo.




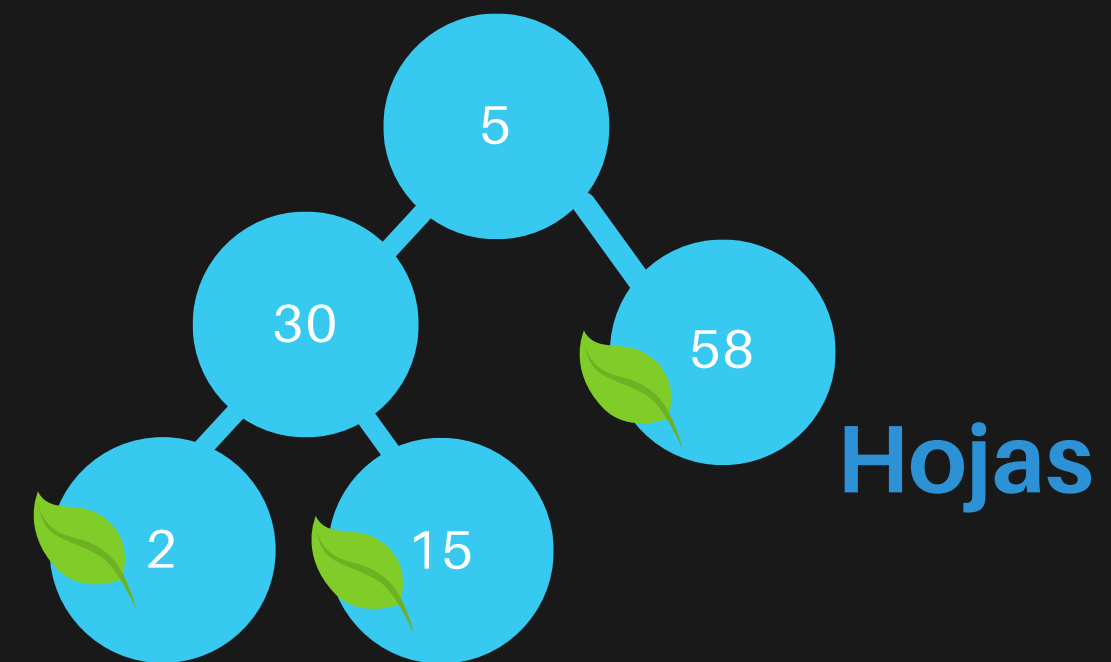




# Árboles

## Hijos y hojas

- **Nodos hijos:**
  - Nodos conectados directamente a un nodo padre.
  - Se ubican en niveles inferiores en relación con su padre.
  - Un nodo sin hijos es una **hoja** 

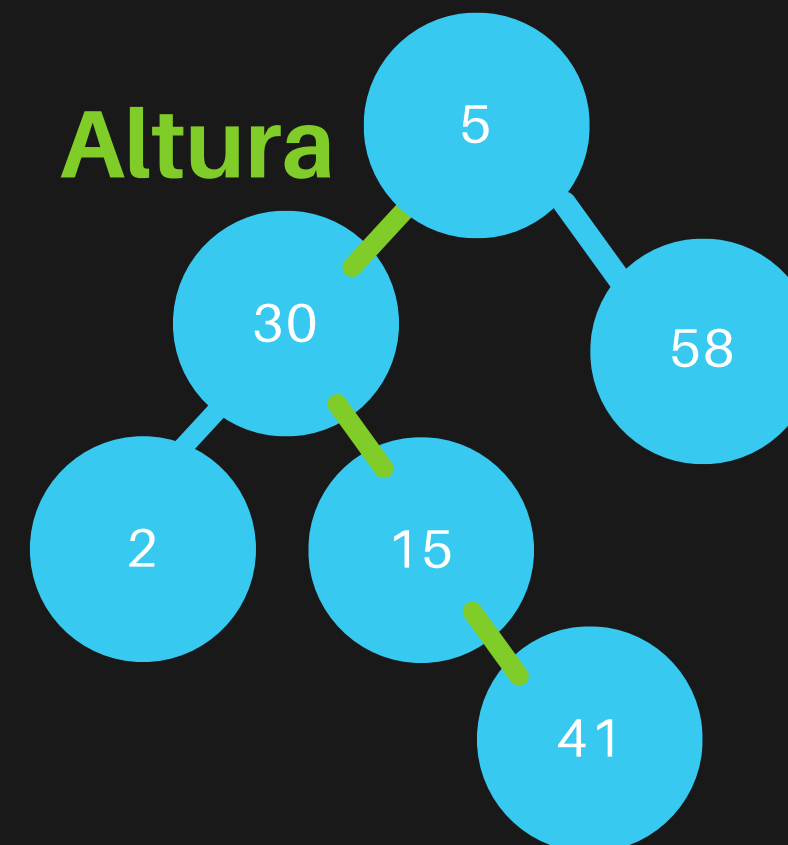




# Árboles

## Altura y Nivel

- **Altura de un árbol:**
  - La altura de un árbol se refiere a la longitud del camino más largo desde la raíz hasta una hoja.
  - Se mide contando el **número de aristas** (enlaces) en ese camino más largo.
- **Nivel de un nodo:**
  - El nivel de un nodo se refiere a la distancia entre ese nodo y la raíz del árbol.
  - La raíz se considera en el nivel 0, los hijos directos de la raíz están en el nivel 1, y así sucesivamente.

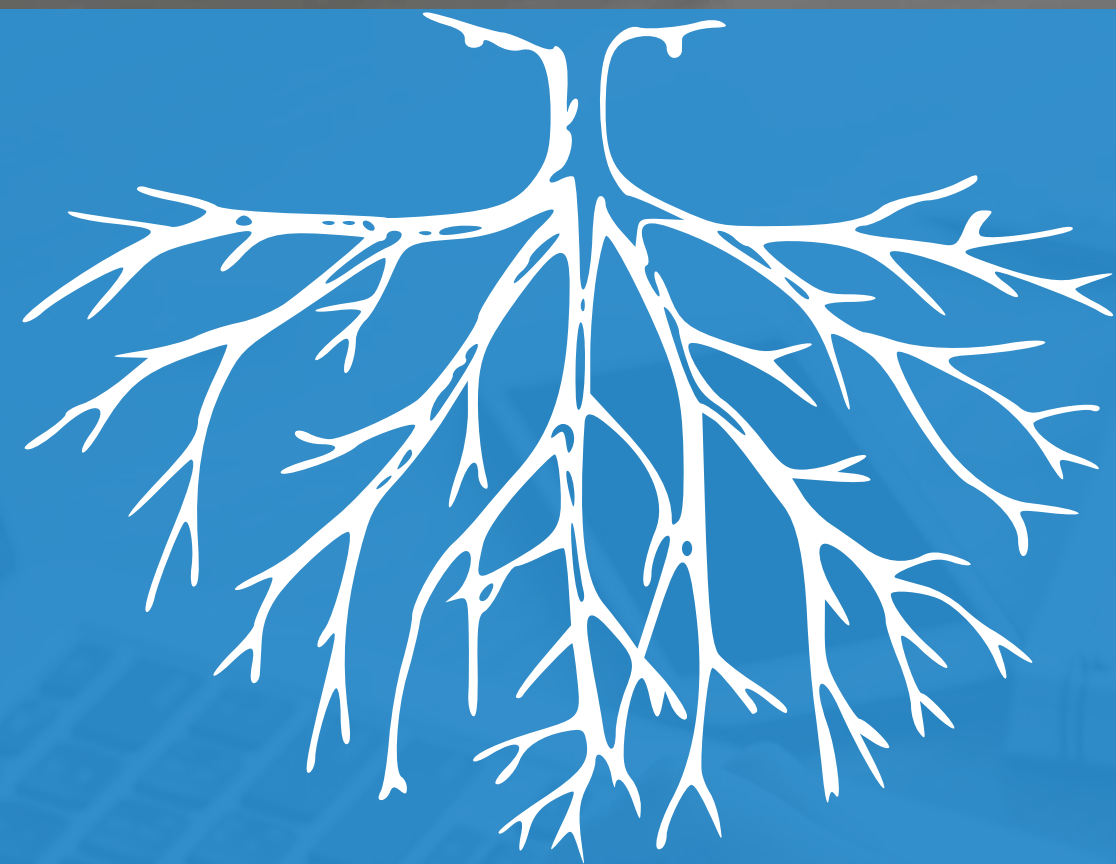


Nivel 0

Nivel 1

Nivel 2

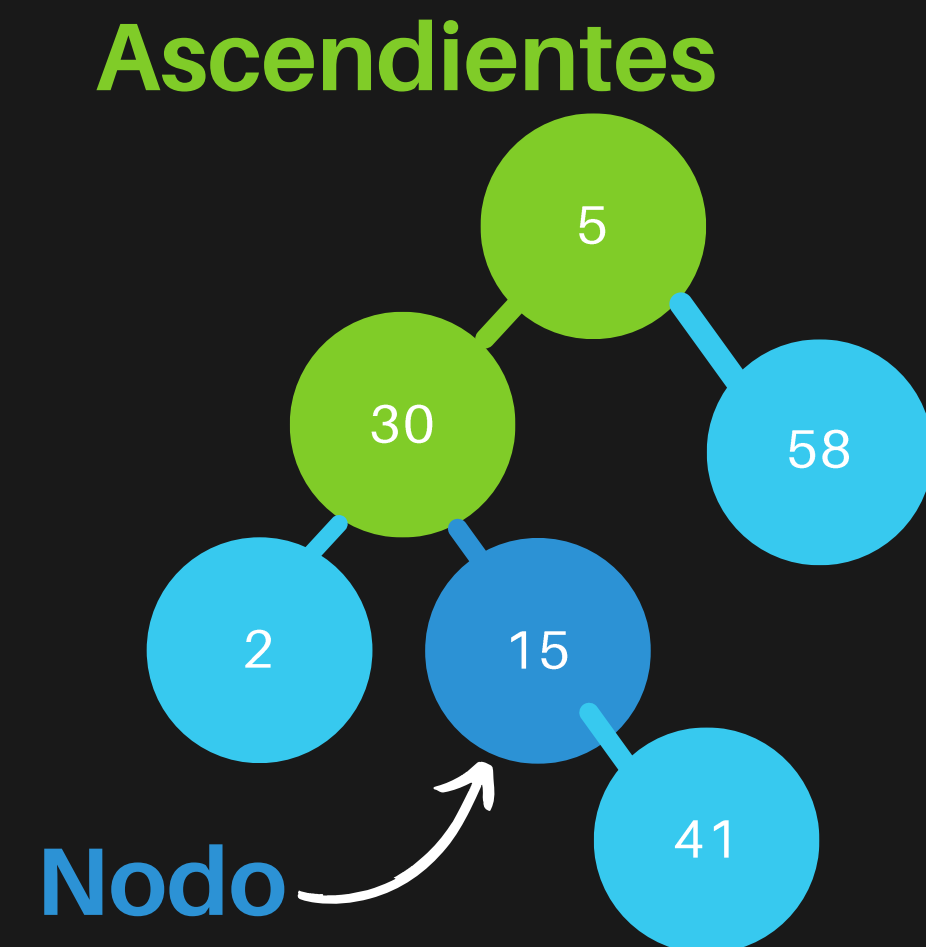
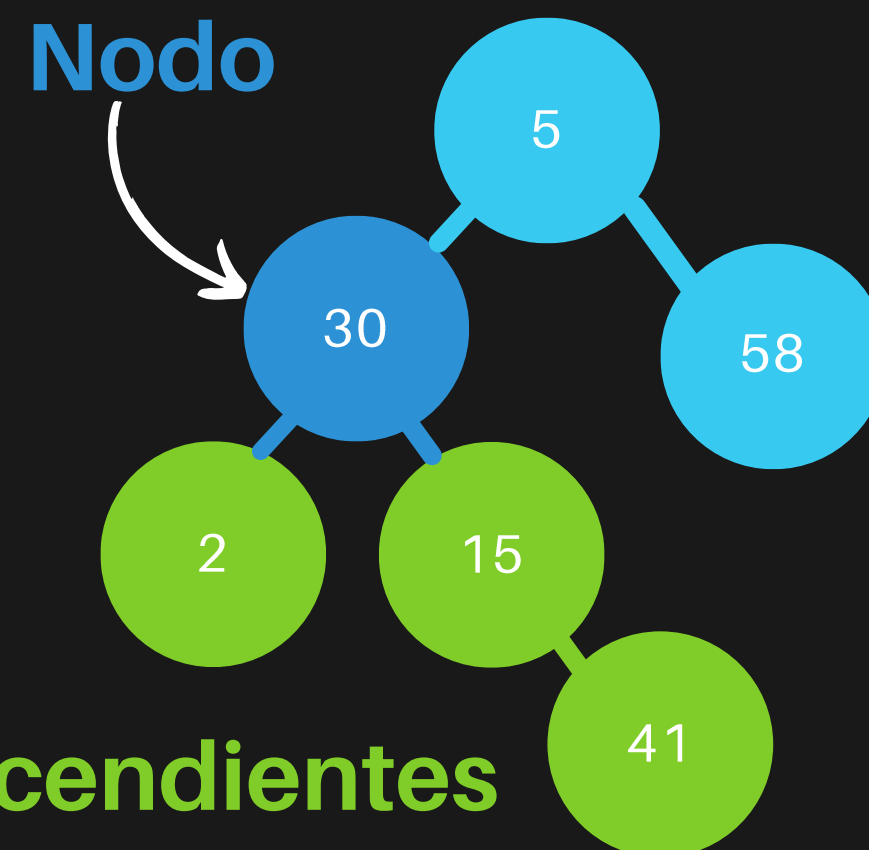
Nivel 3



# Árboles

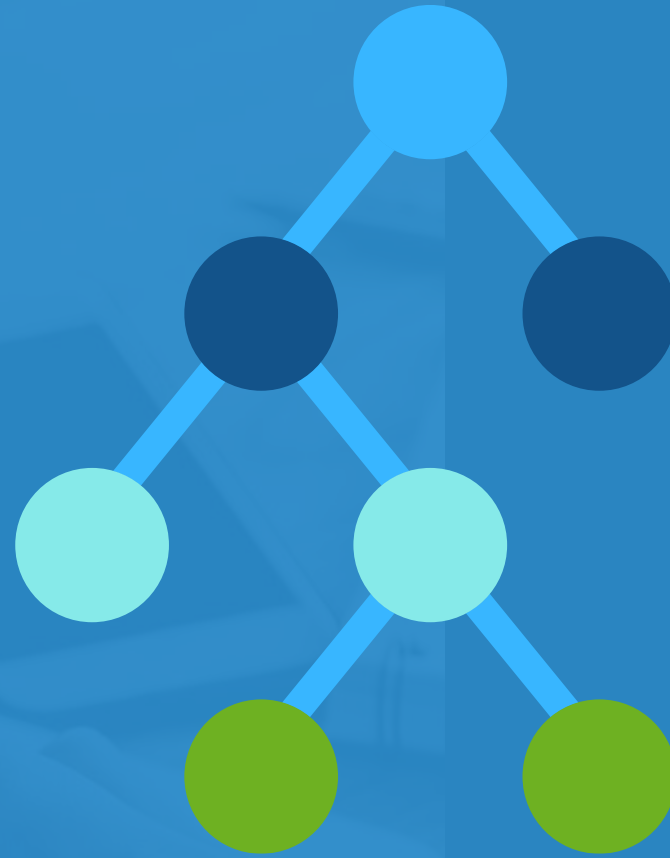
## Ascendientes y Descendientes

- **Ascendientes:**
  - Los ascendientes de un nodo son los nodos ubicados por encima de él en el camino hacia la raíz.
- **Descendientes:**
  - Los descendientes de un nodo son todos los nodos ubicados por debajo de él en el subárbol cuya raíz es ese nodo.



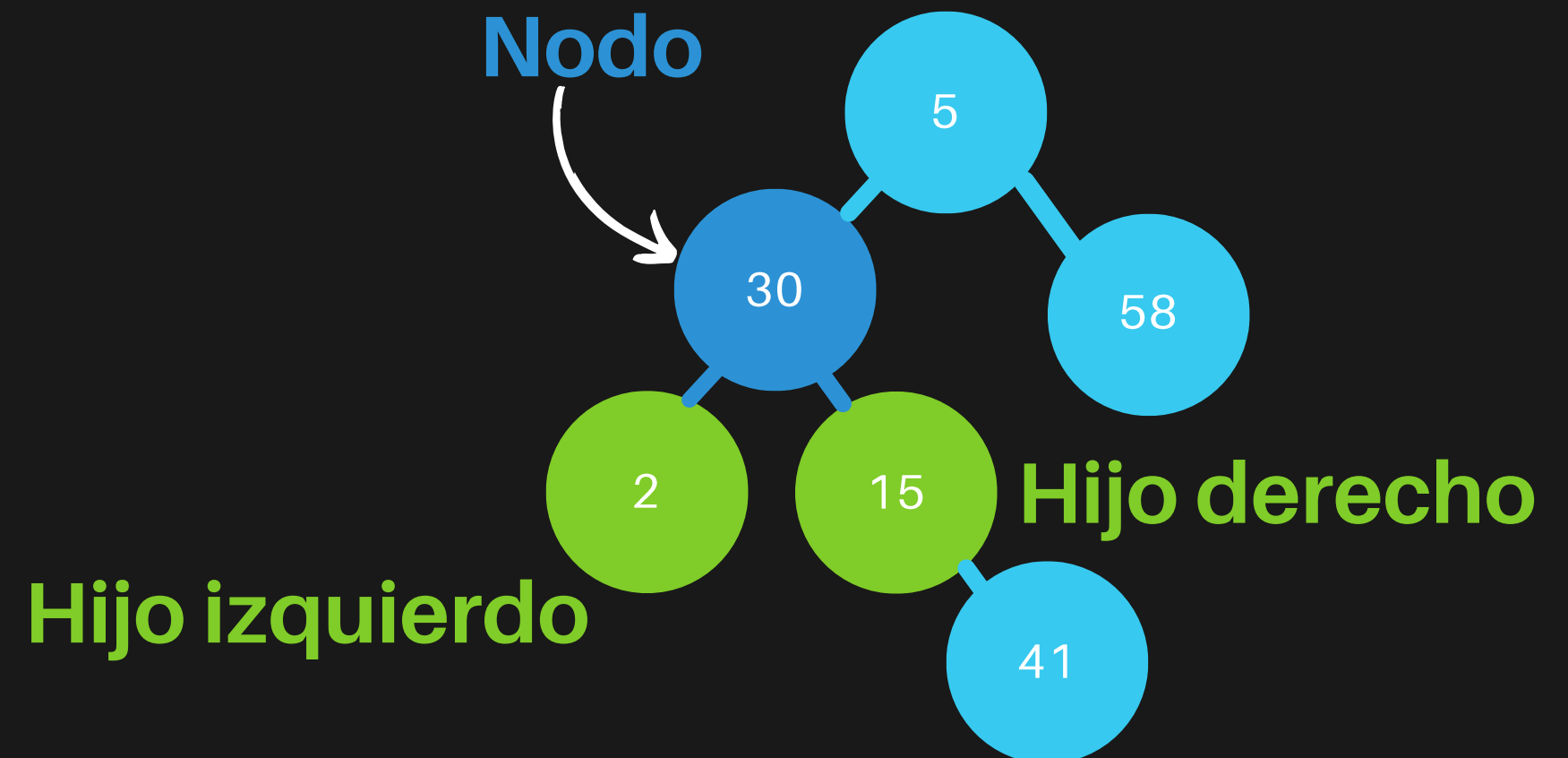


# Árboles binarios



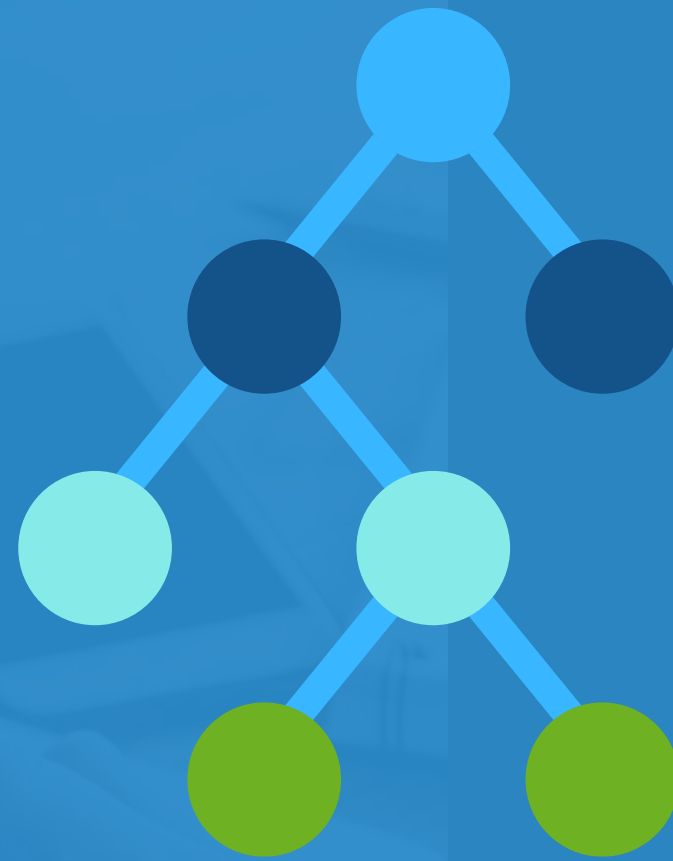
## ¿Qué es un Árbol Binario?

- Un **Árbol Binario** es una estructura jerárquica compuesta por nodos, donde cada nodo tiene un valor y puede tener hasta dos hijos: **un hijo izquierdo y un hijo derecho**.
- Cada nodo puede ser visto como la raíz de su propio subárbol, que a su vez puede contener más nodos y así sucesivamente.



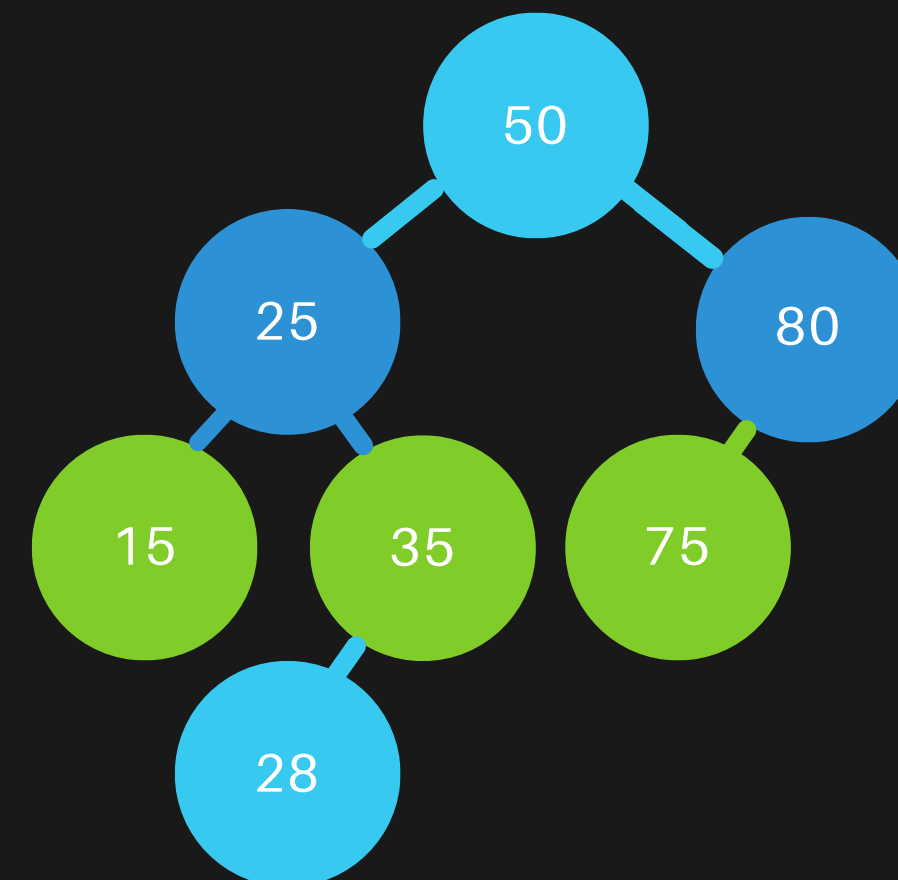


# Árboles binarios de búsqueda [ABB]

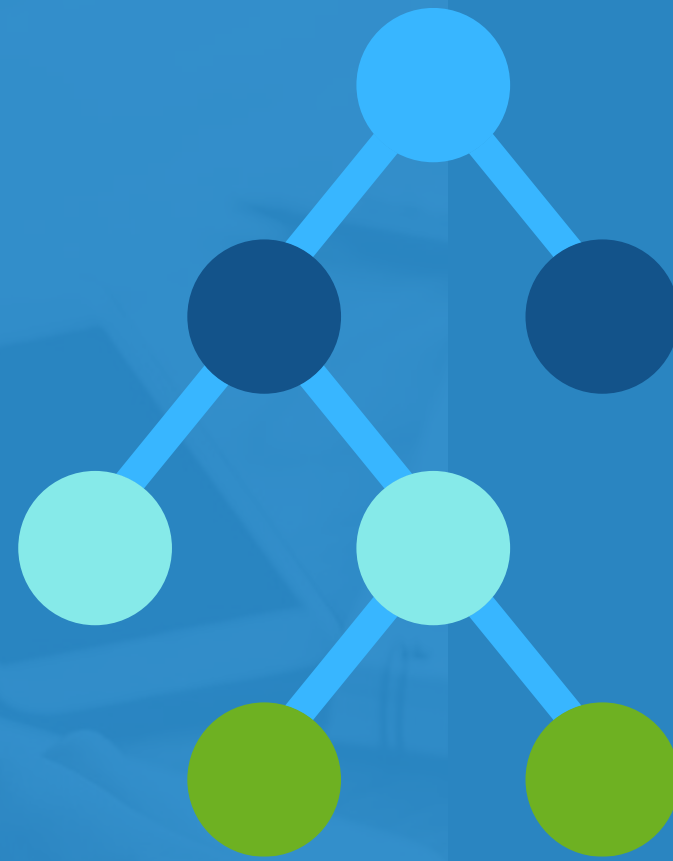


## Árboles Binarios de Búsqueda (ABB)

- Un **ABB** es un tipo especial de árbol binario en el que cada nodo cumple la propiedad de que todos los nodos en su subárbol izquierdo tienen valores **menores** que el nodo y todos los nodos en su subárbol derecho tienen valores **mayores**.
- Cada nodo tiene un **valor único** y puede tener hasta dos hijos

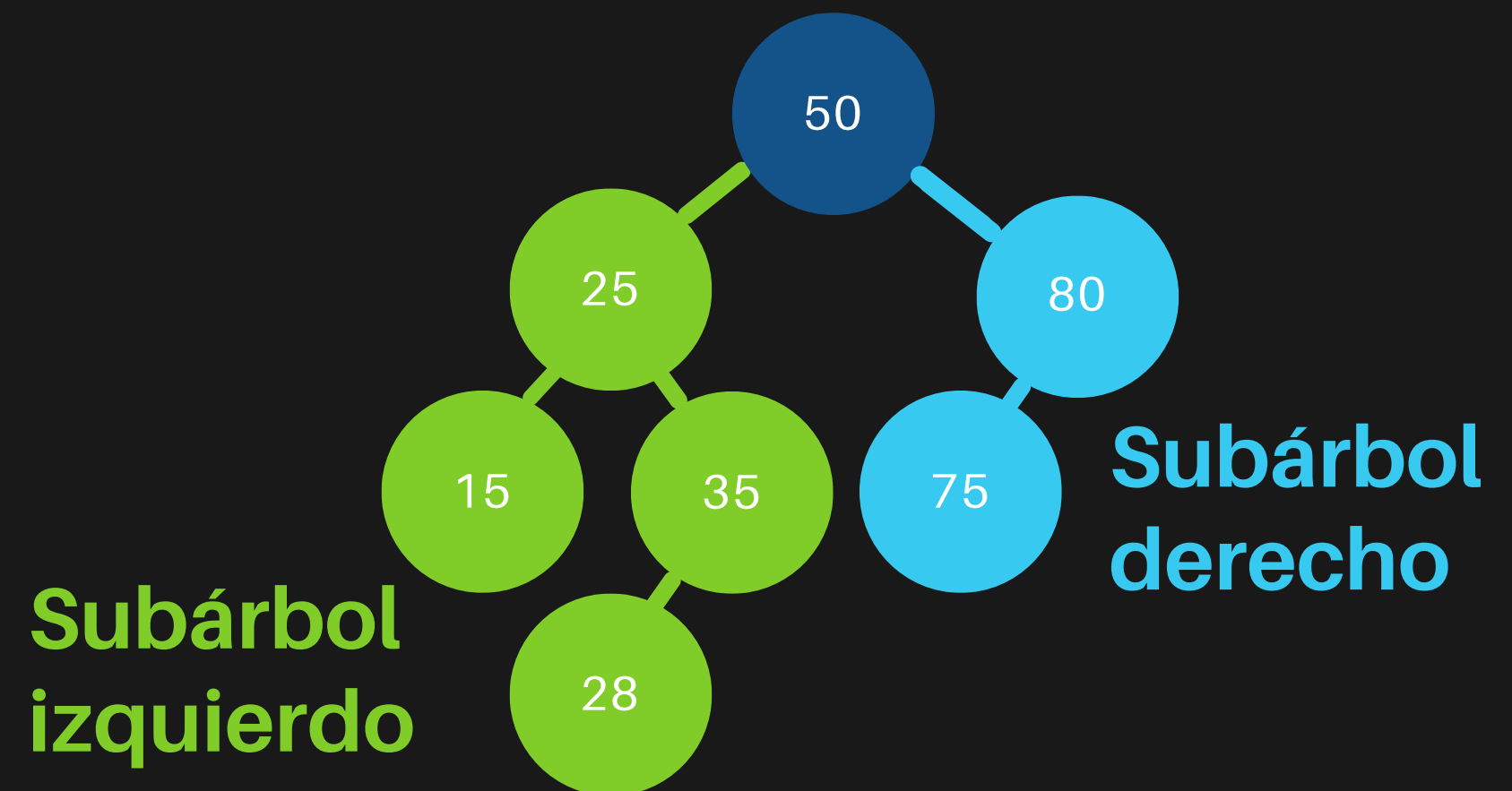


# Árboles binarios de búsqueda [ABB]

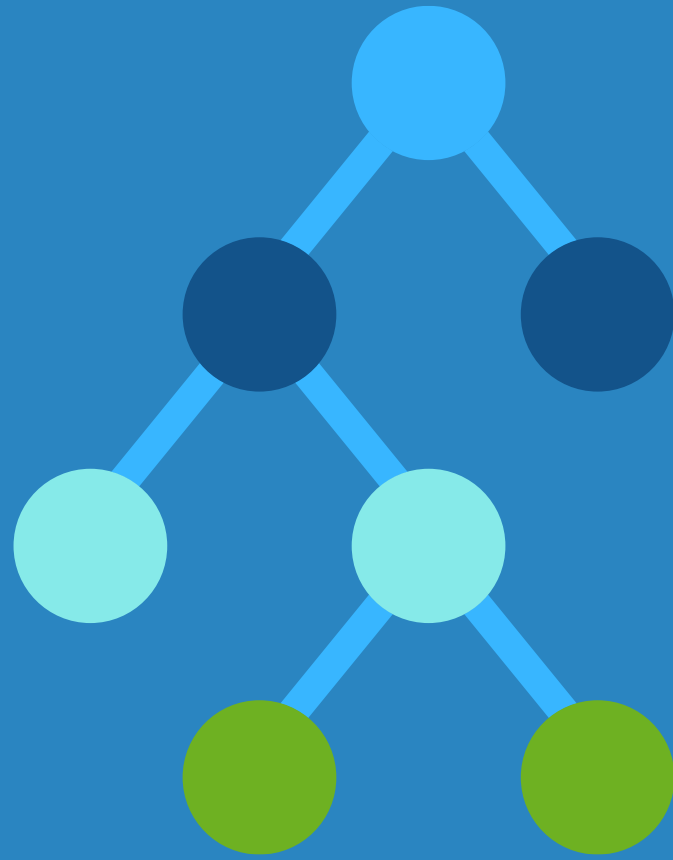


## Árboles Binarios de Búsqueda (ABB)

- **Búsqueda eficiente:** La estructura ordenada permite realizar búsquedas en tiempo logarítmico ( $O(\log n)$ ).
- **Inserción y eliminación eficientes:** Mantienen la estructura ordenada al insertar o eliminar nodos.
- Algoritmos de **ordenamiento:** Pueden utilizarse para implementar algoritmos de ordenamiento eficientes, como el ordenamiento inorden.



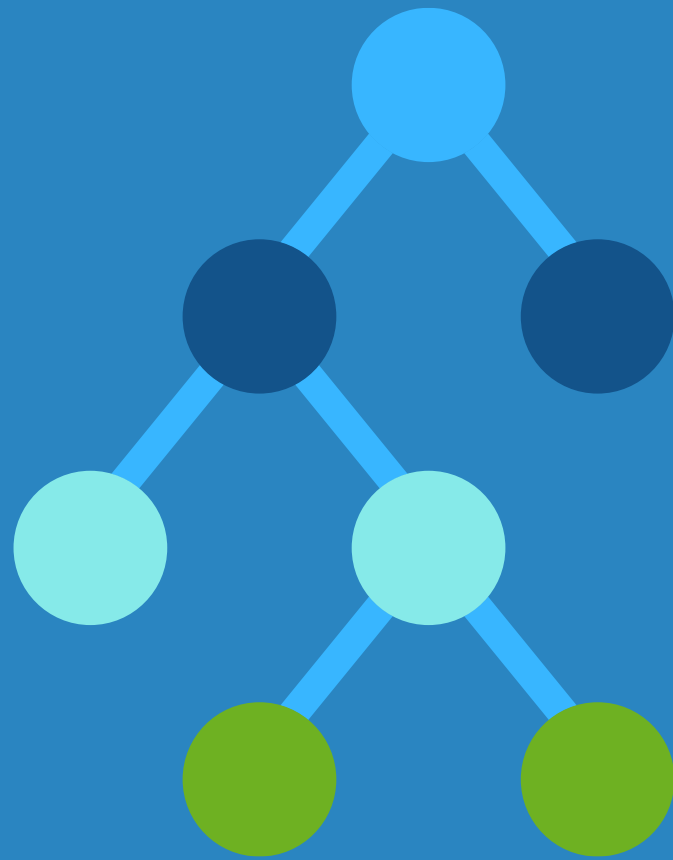
# Especificación[ABB]



## Árboles binarios de búsqueda [ABB]

### ABB TDA

- **inicializarABB**: Establece el árbol binario de búsqueda en su estado inicial.
- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (*se asume que el árbol está inicializado*).
- **eliminarElem**: Remueve el elemento dado manteniendo la propiedad de ABB (*se asume que el árbol está inicializado*).
- **raíz**: Recupera el valor de la raíz de un ABB (*se asume que el árbol está inicializado y no está vacío*).
- **hijoIzq**: Devuelve el subárbol izquierdo (*se asume que el árbol está inicializado y no está vacío*).
- **hijoDer**: Devuelve el subárbol derecho (*se asume que el árbol está inicializado y no está vacío*).
- **arbolVacio**: Indica si el árbol está vacío o no (*se asume que el árbol está inicializado*).



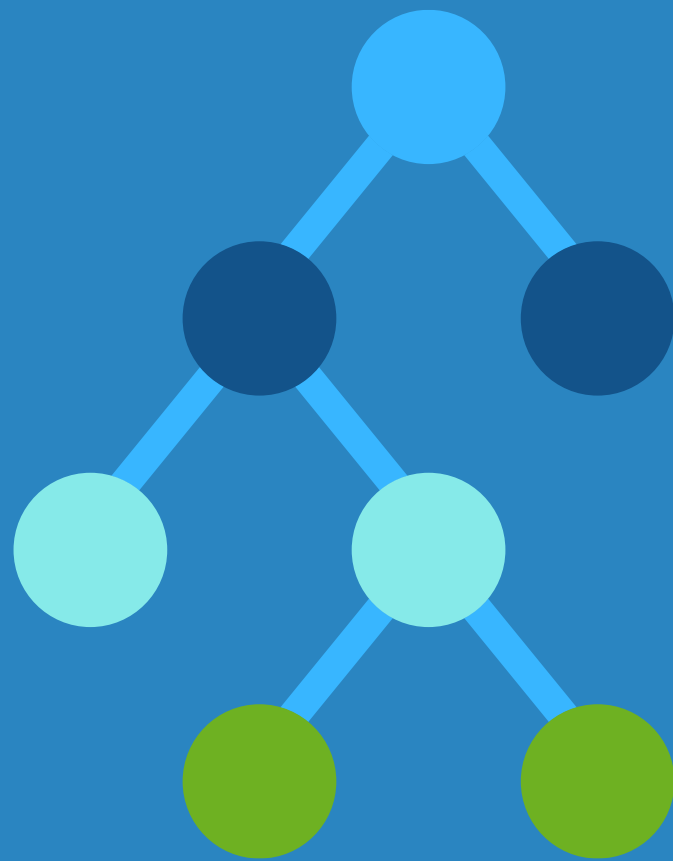
# Árboles binarios de búsqueda [ABB]

## Recorridos

- **Recorrido en preOrder:**
  - Explora los nodos en el siguiente orden: raíz, subárbol izquierdo, subárbol derecho.
  - Primero se visita la raíz, luego el subárbol izquierdo y finalmente el subárbol derecho.
- **Recorrido en inOrder:**
  - Explora los nodos en el siguiente orden: subárbol izquierdo, raíz, subárbol derecho.
  - Primero se visita el subárbol izquierdo, luego la raíz y finalmente el subárbol derecho. Recorrido **menor a mayor**
- **Recorrido en postOrder:**
  - Explora los nodos en el siguiente orden: subárbol izquierdo, subárbol derecho, raíz.
  - Primero se visita el subárbol izquierdo, luego el subárbol derecho y finalmente la raíz.



# Árboles binarios de búsqueda [ABB]



## Recorridos

- Recorrido en preOrder:

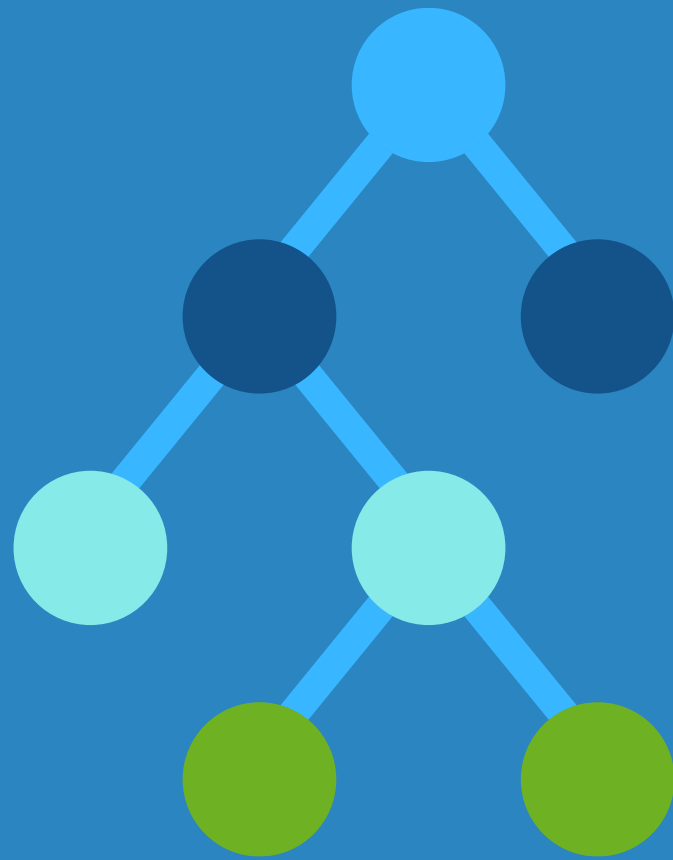
```
public void preOrder (ABBTDA a){  
    if (!a.arbolVacio()) {  
        System.out.println(a.raiz());  
        preOrder(a.hijoIzq());  
        preOrder(a.hijoDer());  
    }  
}
```

- Recorrido en inOrder:

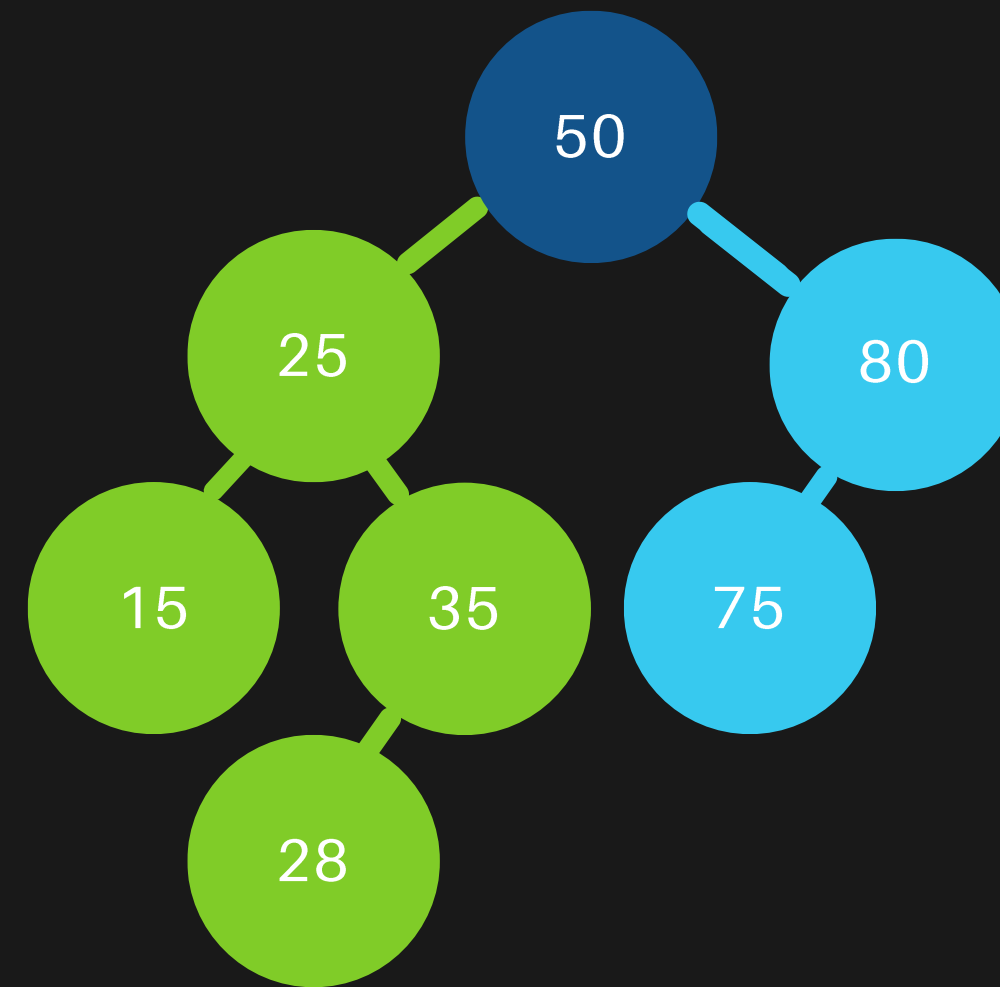
```
public void inOrder (ABBTDA a) {  
    if (!a.arbolVacio( )) {  
        inOrder(a.hijoIzq( ));  
        System.out.println(a.raiz( ));  
        inOrder(a.hijoDer( ));  
    }  
}
```

- Recorrido en postOrder:

```
public void postOrder (ABBTDA a) {  
    if (!a.arbolVacio( )) {  
        postOrder(a.hijoIzq( ));  
        postOrder(a.hijoDer( ));  
        System.out.println(a.raiz( ));  
    }  
}
```



# Árboles binarios de búsqueda [ABB]

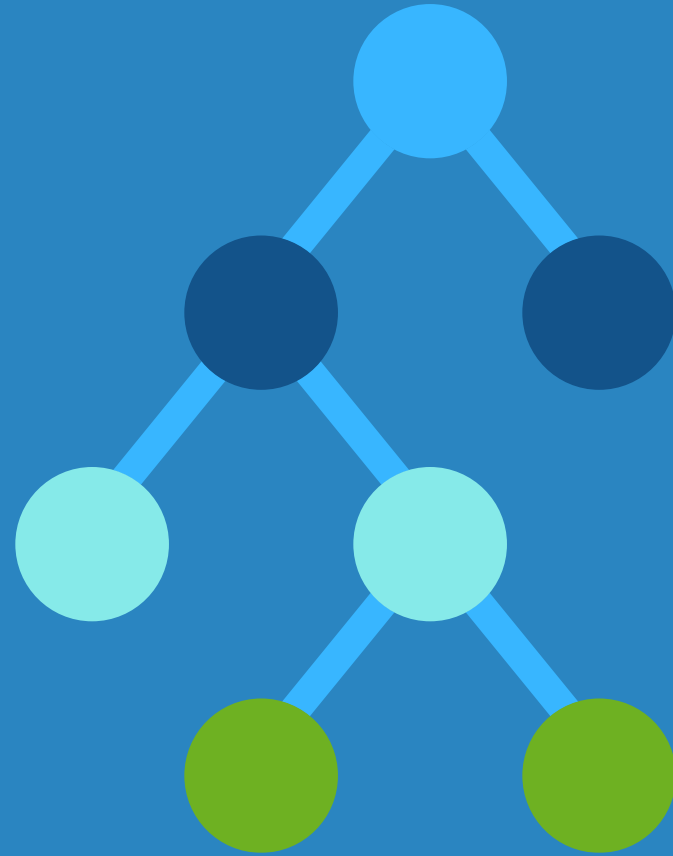


## Ejemplo de recorridos

- **Recorrido en preOrder:** 50-25-15-35-28-80-75
- **Recorrido en inOrder:** 15-25-28-35-50-75-80
- **Recorrido en postOrder:** 15-28-35-25-75-80-50

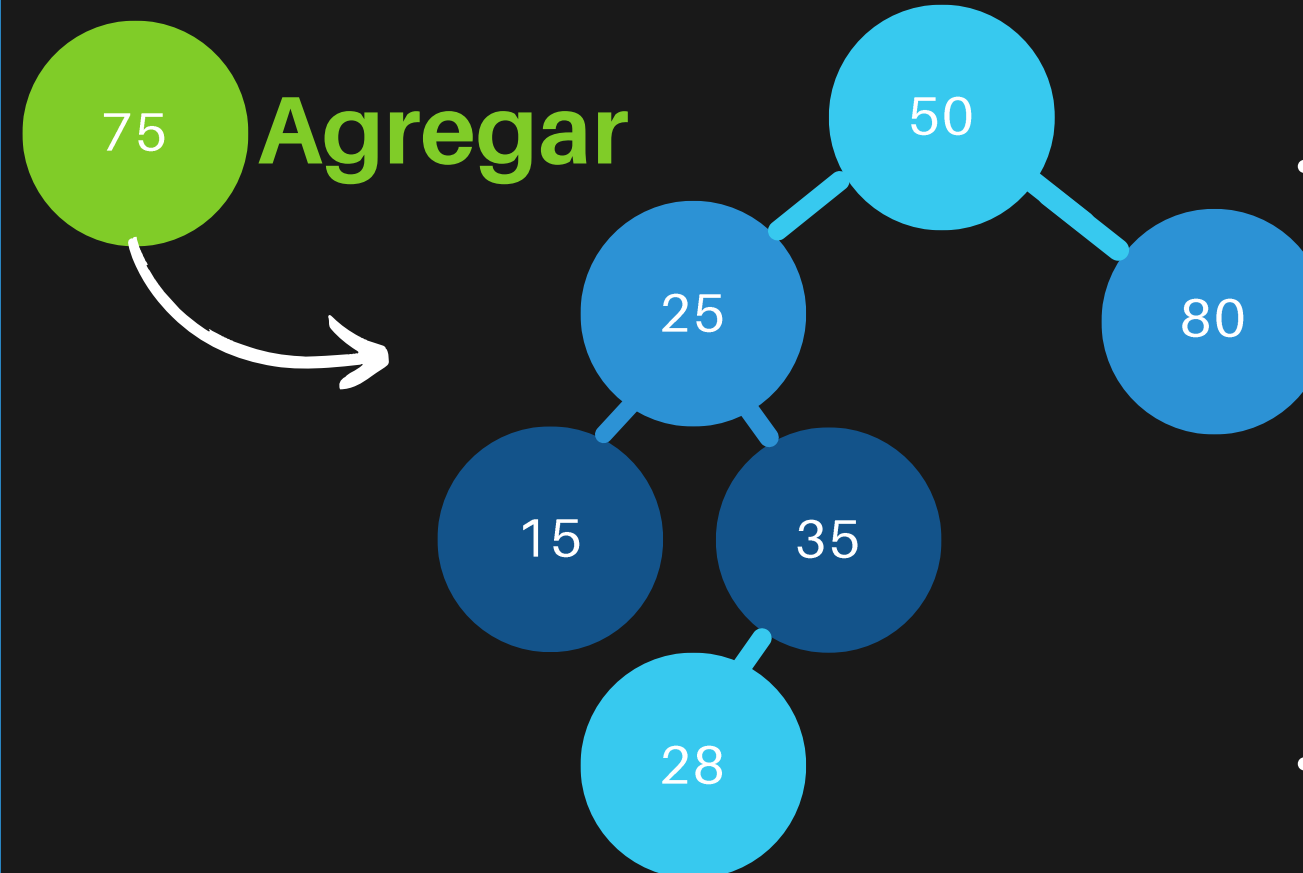
# agregarElem()

## Árboles binarios de búsqueda [ABB]



## ABB TDA

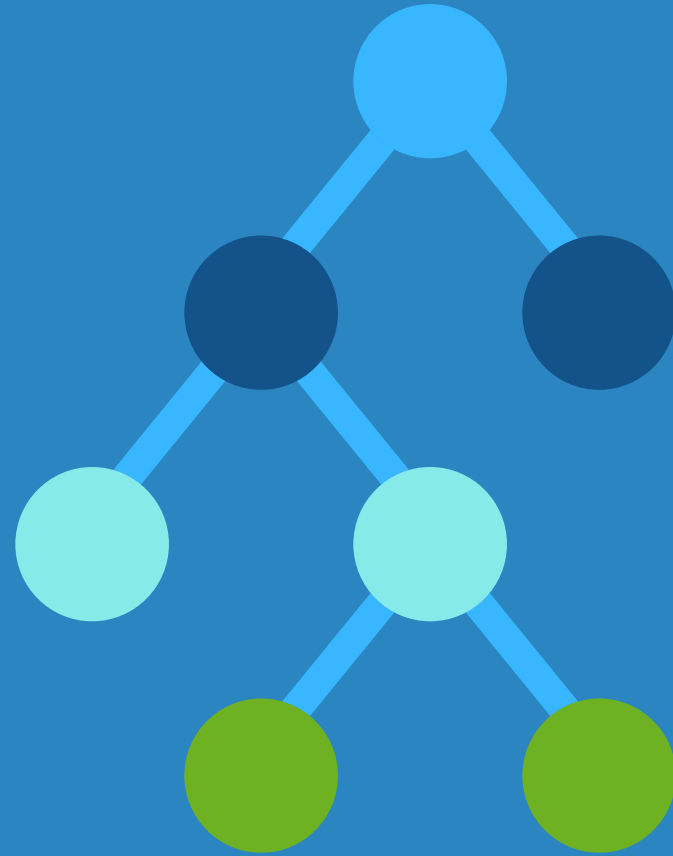
- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (se *asume* que el árbol está inicializado).



### Estrategia de inserción

- **Paso 1:** Comenzamos en la raíz del árbol.
  - Si el **árbol está vacío**, el nuevo elemento se convierte en la raíz.
- **Paso 2:** Comparamos el nuevo elemento con el valor del nodo actual.
  - Si el **nuevo elemento es menor**, nos movemos hacia el subárbol izquierdo.
  - Si el nuevo elemento es mayor, nos movemos hacia el subárbol derecho.
  - Continuamos este proceso hasta encontrar una posición adecuada.
- **Paso 3:** Llegamos a un nodo vacío o nulo.
  - Insertamos el **nuevo elemento en ese nodo como una hoja**.
- **Paso 4:** **Mantenemos la propiedad de orden del ABB.**
- Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

# agregarElem()



## Árboles binarios de búsqueda [ABB]

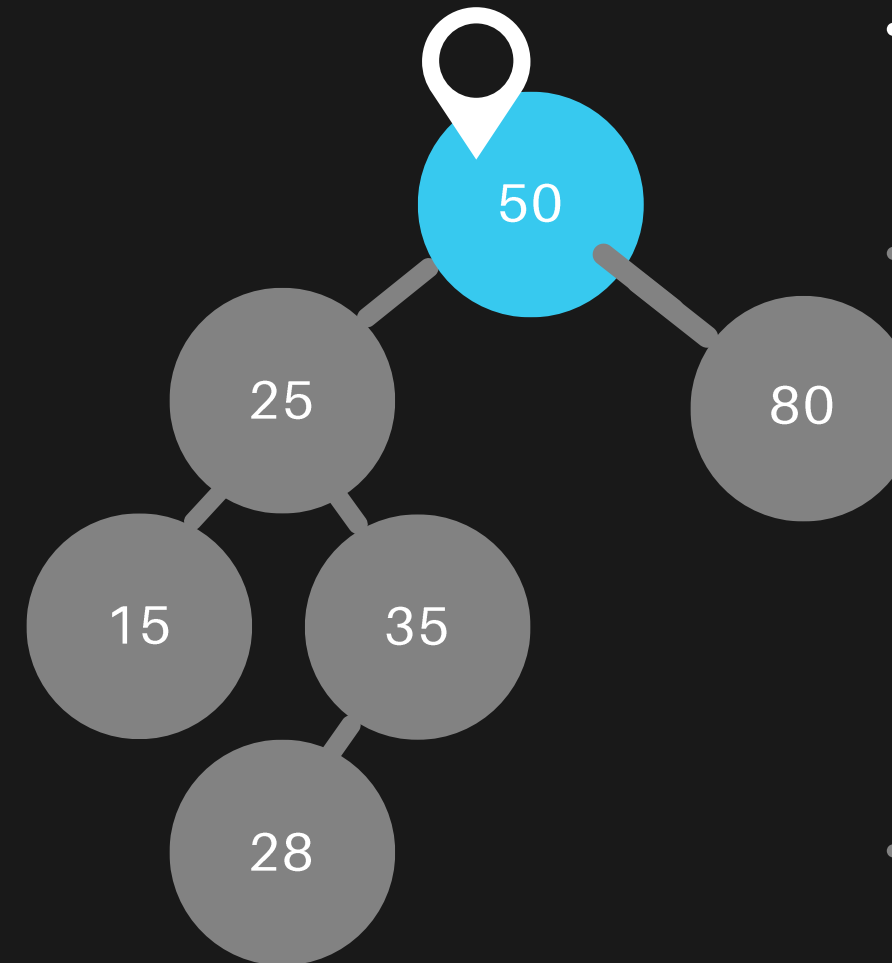
### ABB TDA

- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (se *asume* que el árbol está inicializado).

**Agregar**

75

**No está vacío**



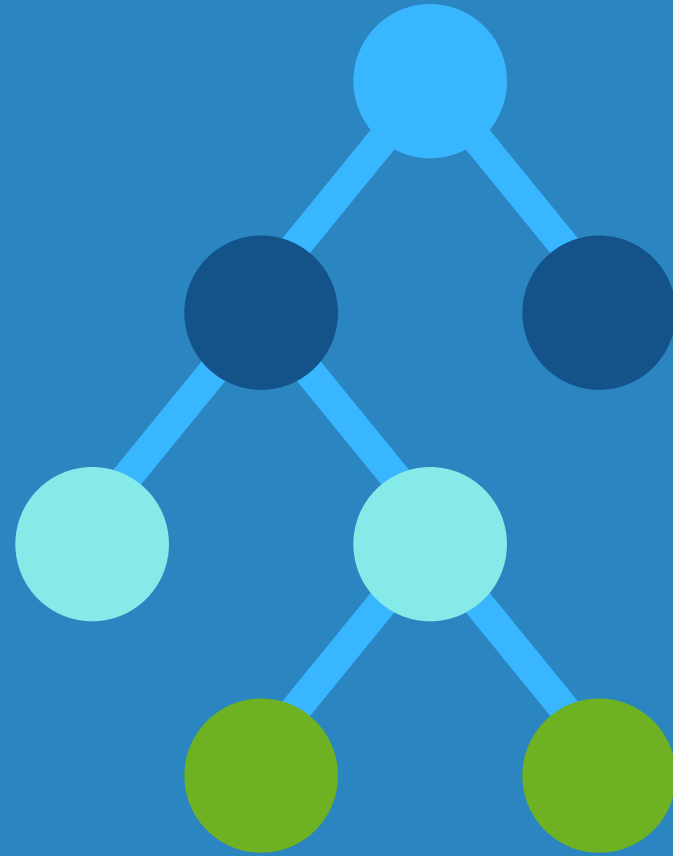
**Estrategia de inserción**

- **Paso 1:** Comenzamos en la raíz del árbol.
  - Si el **árbol está vacío**, el nuevo elemento se convierte en la raíz.
- **Paso 2:** Comparamos el nuevo elemento con el valor del nodo actual.
  - Si el nuevo elemento es menor, nos movemos hacia el subárbol izquierdo.
  - Si el nuevo elemento es mayor, nos movemos hacia el subárbol derecho.
  - Continuamos este proceso hasta encontrar una posición adecuada.
- **Paso 3:** Llegamos a un nodo vacío o nulo.
  - Insertamos el nuevo elemento en ese nodo como una hoja.
- **Paso 4:** Mantenemos la propiedad de orden del ABB.
- Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

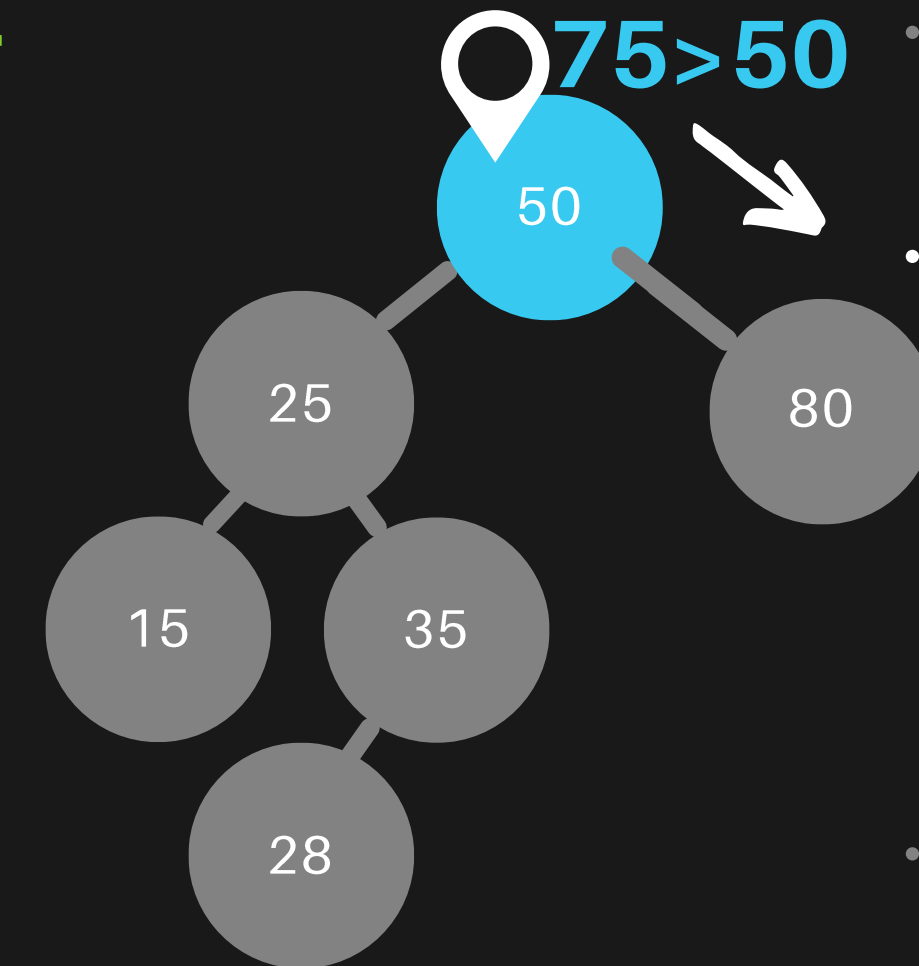


# agregarElem()

## Árboles binarios de búsqueda [ABB]



Agregar



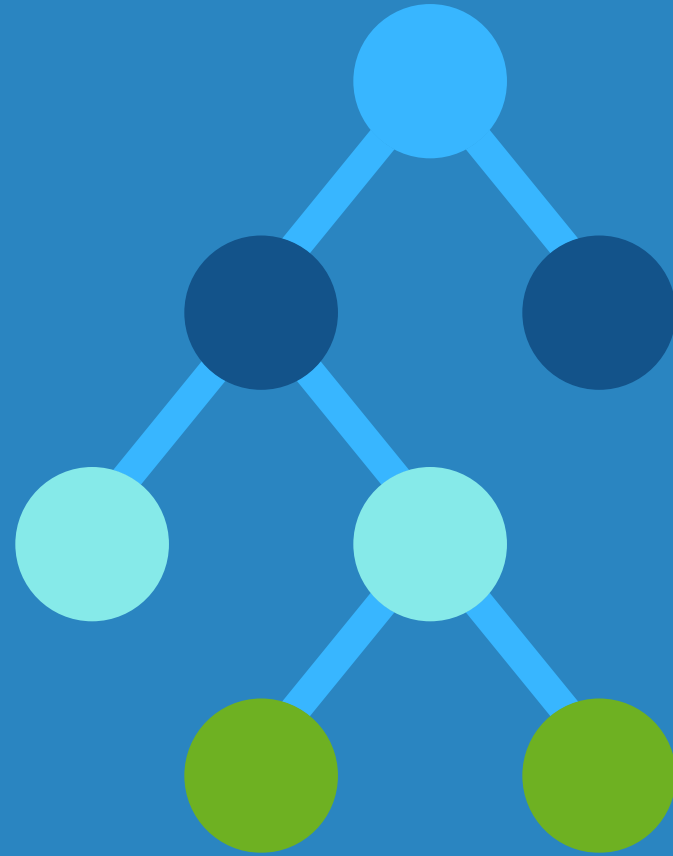
## ABB TDA

- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (se *asume* que el árbol está inicializado).

### Estrategia de inserción

- **Paso 1**: Comenzamos en la raíz del árbol.
  - Si el árbol está vacío, el nuevo elemento se convierte en la raíz.
- **Paso 2**: Comparamos el nuevo elemento con el valor del nodo actual.
  - Si el nuevo elemento es menor, nos movemos hacia el subárbol izquierdo.
  - Si el nuevo elemento es mayor, nos movemos hacia el subárbol derecho.
  - Continuamos este proceso hasta encontrar una posición adecuada.
- **Paso 3**: Llegamos a un nodo vacío o nulo.
  - Insertamos el nuevo elemento en ese nodo como una hoja.
- **Paso 4**: Mantenemos la propiedad de orden del ABB.
- Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

# agregarElem()



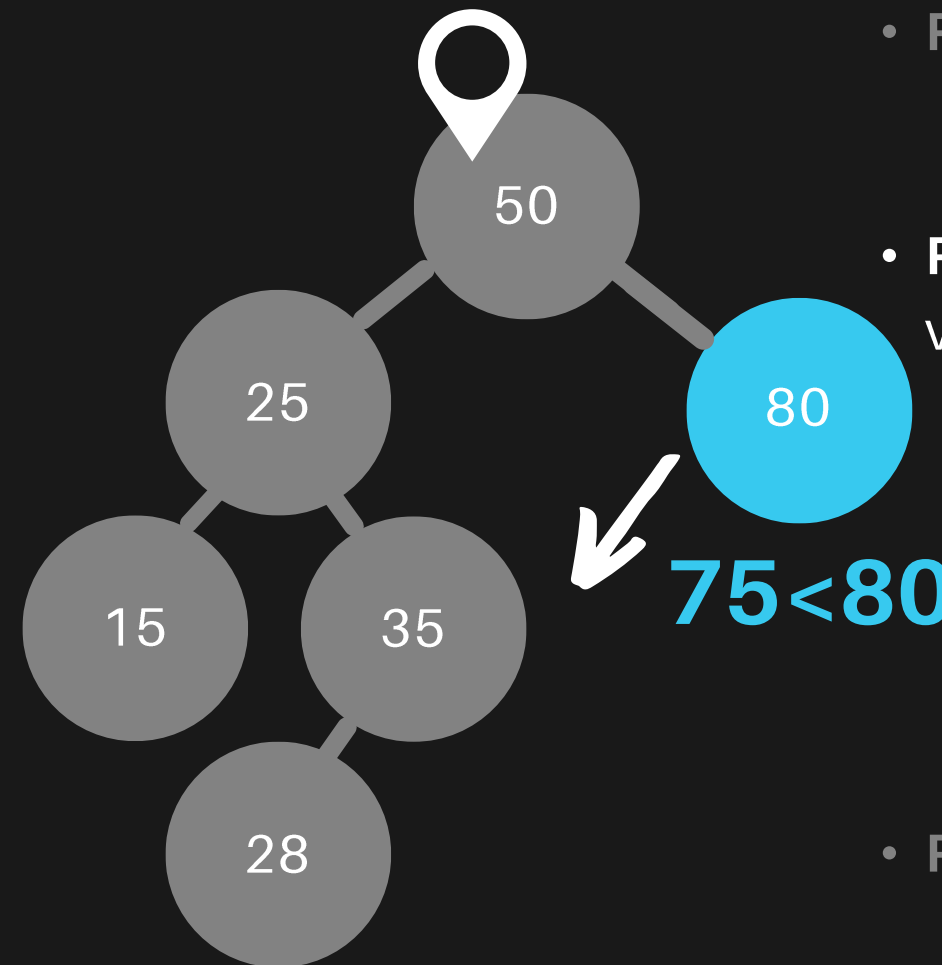
## Árboles binarios de búsqueda [ABB]

### ABB TDA

- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (se *asume* que el árbol está inicializado).

**Agregar**

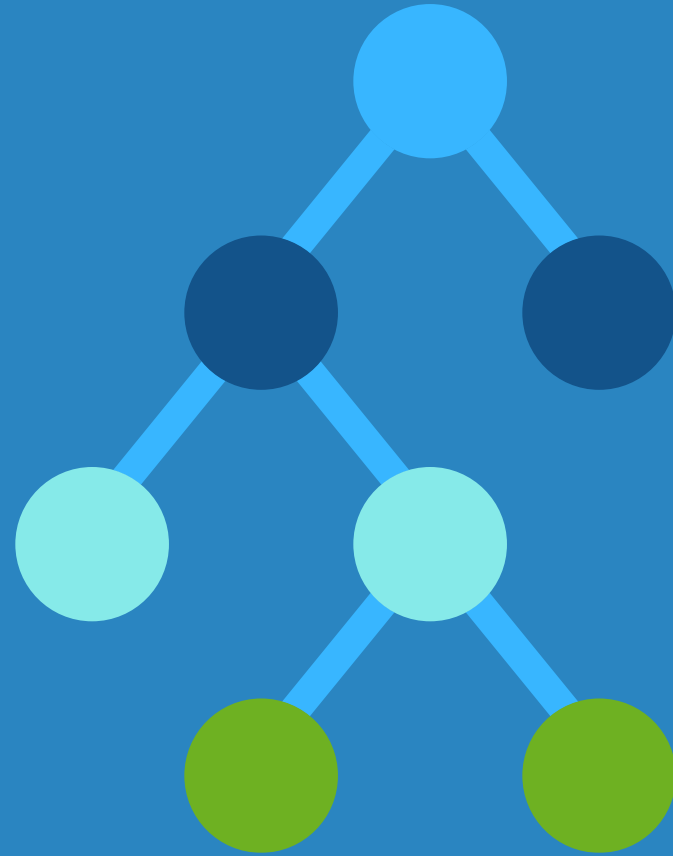
75



#### Estrategia de inserción

- **Paso 1:** Comenzamos en la raíz del árbol.
  - Si el árbol está vacío, el nuevo elemento se convierte en la raíz.
- **Paso 2:** Comparamos el nuevo elemento con el valor del nodo actual.
  - Si el nuevo elemento es menor, nos movemos hacia el subárbol izquierdo.
  - Si el nuevo elemento es mayor, nos movemos hacia el subárbol derecho.
  - Continuamos este proceso hasta encontrar una posición adecuada.
- **Paso 3:** Llegamos a un nodo vacío o nulo.
  - Insertamos el nuevo elemento en ese nodo como una hoja.
- **Paso 4:** Mantenemos la propiedad de orden del ABB.
- Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

# agregarElem()

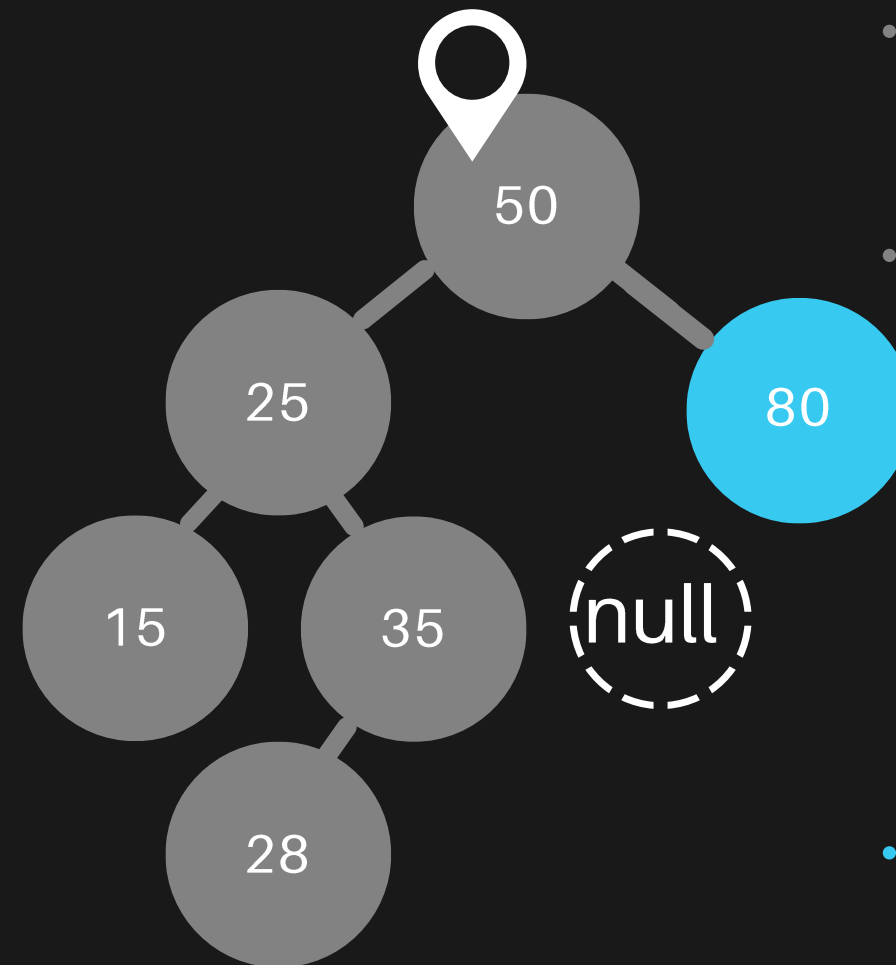


## Árboles binarios de búsqueda [ABB]

### ABB TDA

- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (se *asume* que el árbol está inicializado).

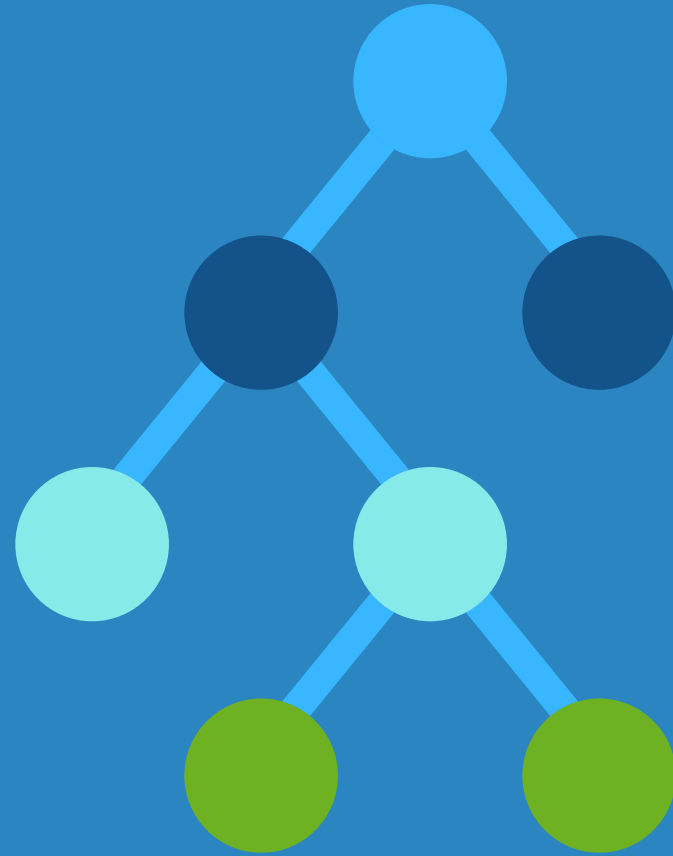
**Agregar**



#### Estrategia de inserción

- **Paso 1:** Comenzamos en la raíz del árbol.
  - Si el árbol está vacío, el nuevo elemento se convierte en la raíz.
- **Paso 2:** Comparamos el nuevo elemento con el valor del nodo actual.
  - Si el nuevo elemento es menor, nos movemos hacia el subárbol izquierdo.
  - Si el nuevo elemento es mayor, nos movemos hacia el subárbol derecho.
  - Continuamos este proceso hasta encontrar una posición adecuada.
- **Paso 3:** Llegamos a un nodo vacío o nulo.
  - Insertamos el nuevo elemento en ese nodo como una hoja.
- **Paso 4:** Mantenemos la propiedad de orden del ABB.
- Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

# agregarElem()



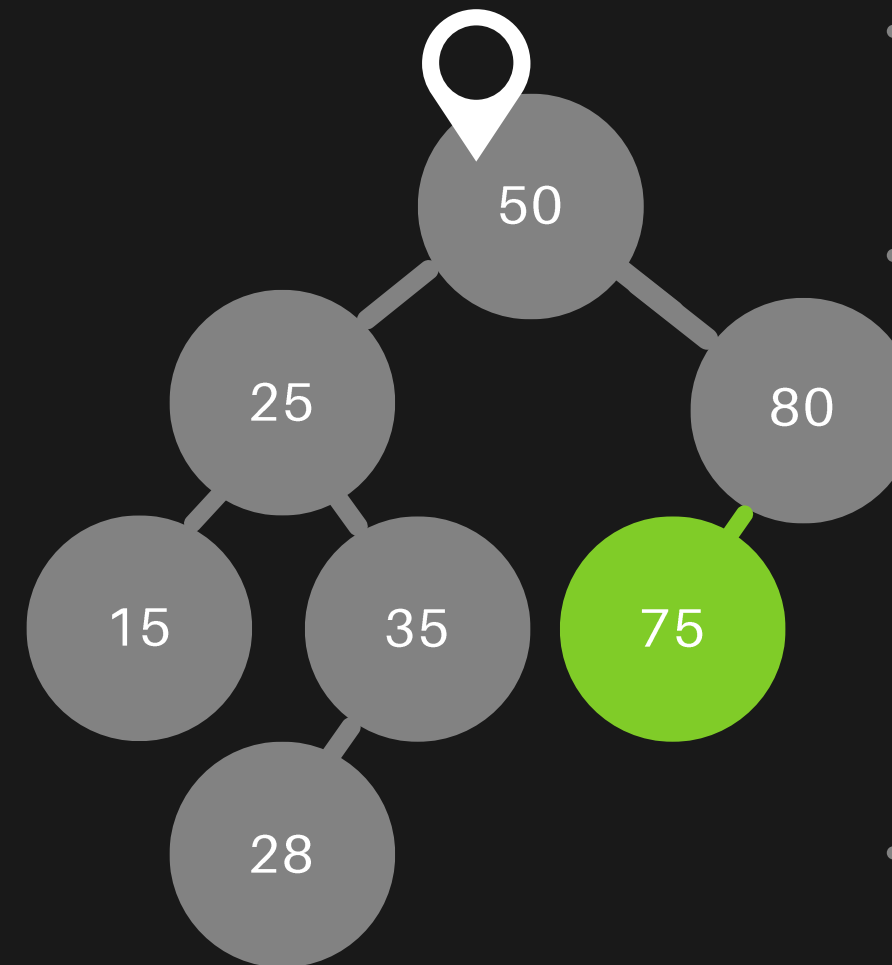
## Árboles binarios de búsqueda [ABB]

### ABB TDA

- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (se *asume* que el árbol está inicializado).

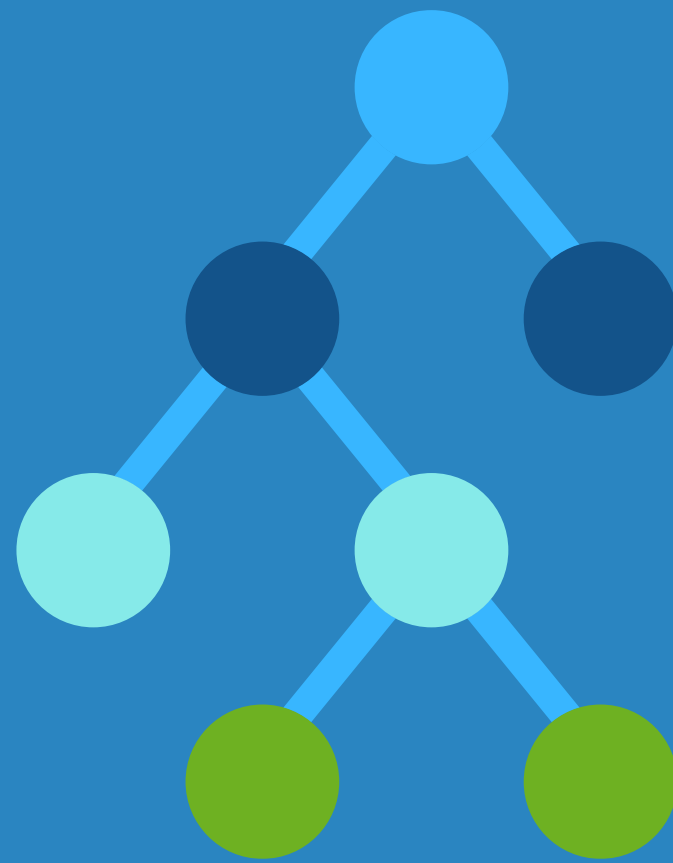
#### Estrategia de inserción

- **Paso 1**: Comenzamos en la raíz del árbol.
  - Si el árbol está vacío, el nuevo elemento se convierte en la raíz.
- **Paso 2**: Comparamos el nuevo elemento con el valor del nodo actual.
  - Si el nuevo elemento es menor, nos movemos hacia el subárbol izquierdo.
  - Si el nuevo elemento es mayor, nos movemos hacia el subárbol derecho.
  - Continuamos este proceso hasta encontrar una posición adecuada.
- **Paso 3**: Llegamos a un nodo vacío o nulo.
  - Insertamos el nuevo elemento en ese nodo como una hoja.
- ✓ **Paso 4**: Mantenemos la propiedad de orden del ABB.
- ✓ Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.





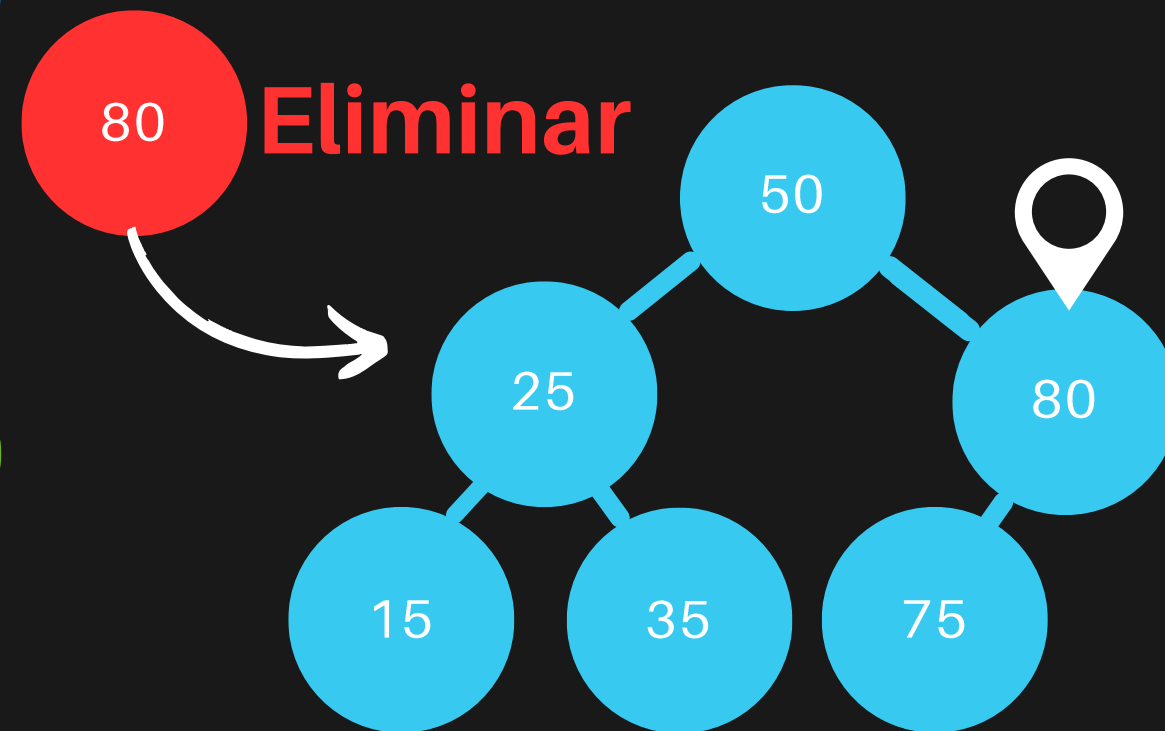
# eliminarElem()



## Árboles binarios de búsqueda [ABB]

### ABB TDA

**eliminarElem:** Remueve el elemento dado manteniendo la propiedad de ABB (se asume que el árbol está inicializado).

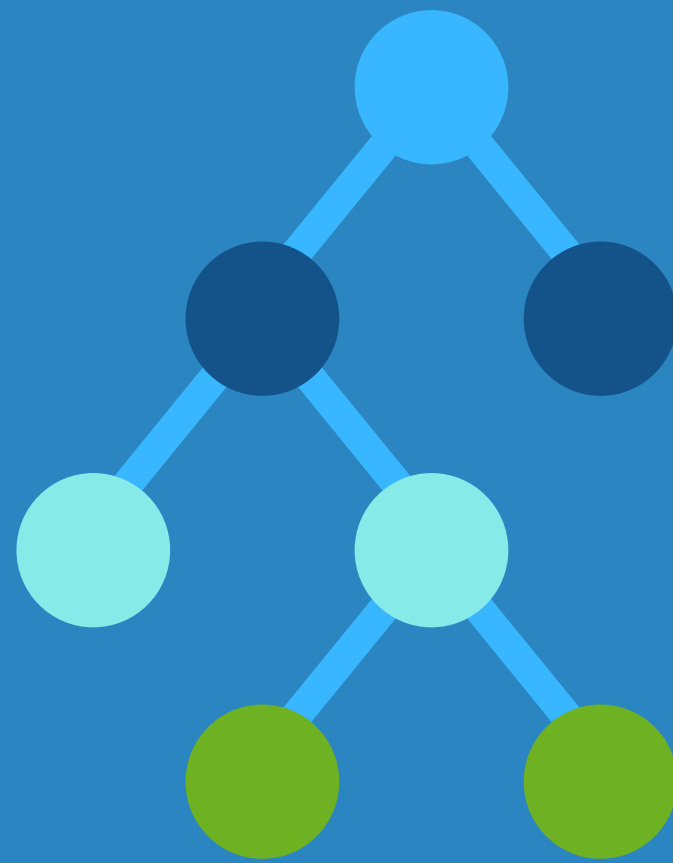


#### Estrategia de eliminación:

Para eliminar un elemento en un ABB, seguimos los siguientes pasos:

- Paso 1: Búsqueda del nodo a eliminar.
  - Comenzamos en la raíz y buscamos el nodo que contiene el elemento a eliminar.
- Paso 2: Casos de eliminación:
  - **Caso 1: Nodo con subárbol izquierdo.**
    - Reemplazamos el nodo con el mayor elemento de su subárbol izquierdo.
  - **Caso 2: Nodo con subárbol derecho.**
    - Reemplazamos el nodo con el menor elemento de su subárbol derecho.
  - **Caso 3: Nodo sin subárboles.**
    - Es una hoja y se reemplaza por null.
- Paso 3: Mantenimiento del orden del ABB.
  - Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

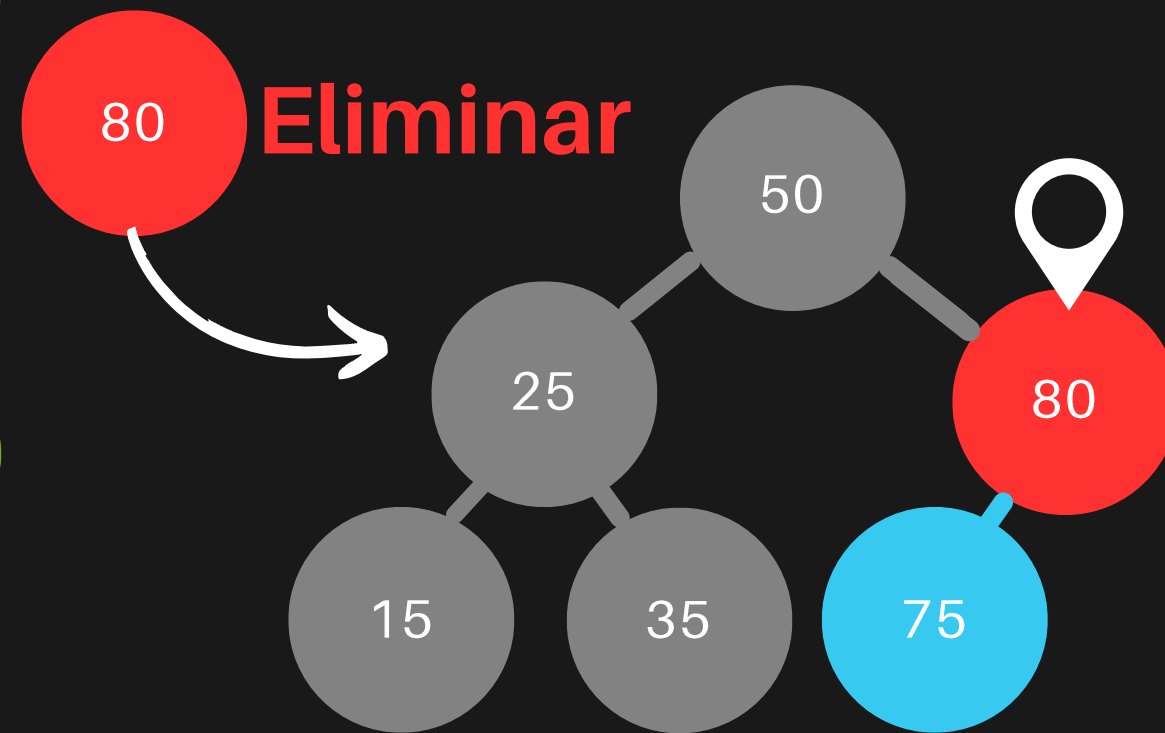
# eliminarElem()



## Árboles binarios de búsqueda [ABB]

### ABB TDA

**eliminarElem:** Remueve el elemento dado manteniendo la propiedad de ABB (se asume que el árbol está inicializado).

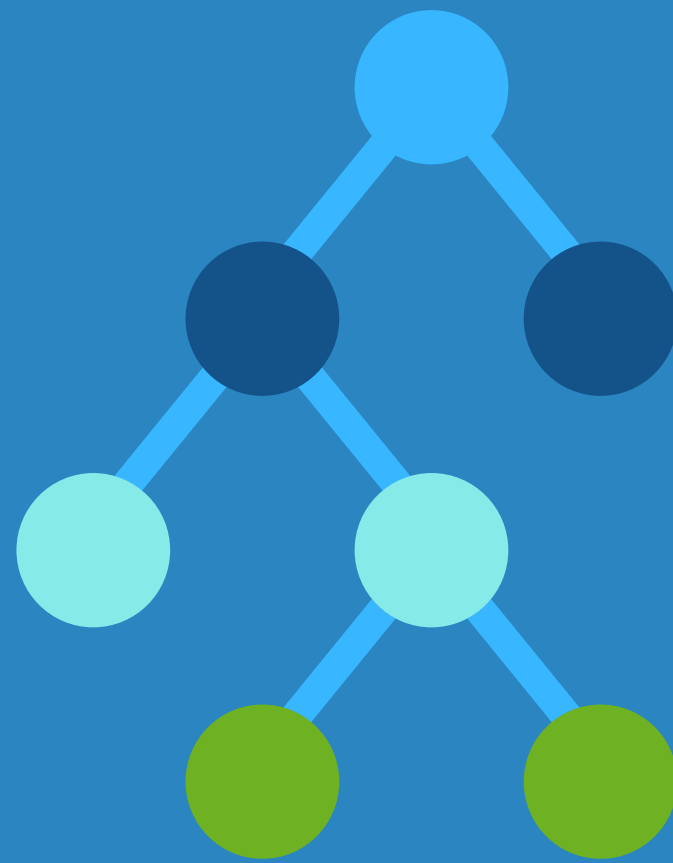


#### Estrategia de eliminación:

Para eliminar un elemento en un ABB, seguimos los siguientes pasos:

- Paso 1: Búsqueda del nodo a eliminar.
  - Comenzamos en la raíz y buscamos el nodo que contiene el elemento a eliminar.
- Paso 2: Casos de eliminación:
  - **Caso 1: Nodo con subárbol izquierdo.**
    - Reemplazamos el nodo con el mayor elemento de su subárbol izquierdo.
  - **Caso 2: Nodo con subárbol derecho.**
    - Reemplazamos el nodo con el menor elemento de su subárbol derecho.
  - **Caso 3: Nodo sin subárboles.**
    - Es una hoja y se reemplaza por null.
- Paso 3: Mantenimiento del orden del ABB.
  - Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

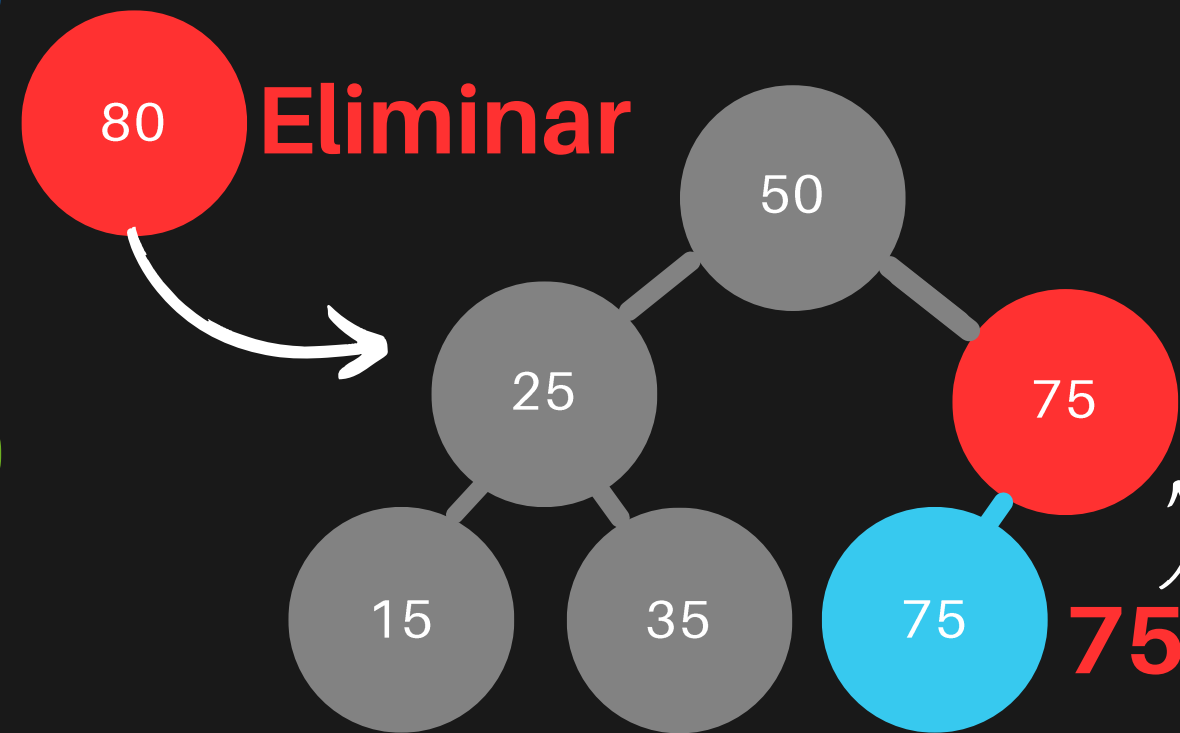
# eliminarElem()



## Árboles binarios de búsqueda [ABB]

### ABB TDA

**eliminarElem:** Remueve el elemento dado manteniendo la propiedad de ABB (se asume que el árbol está inicializado).

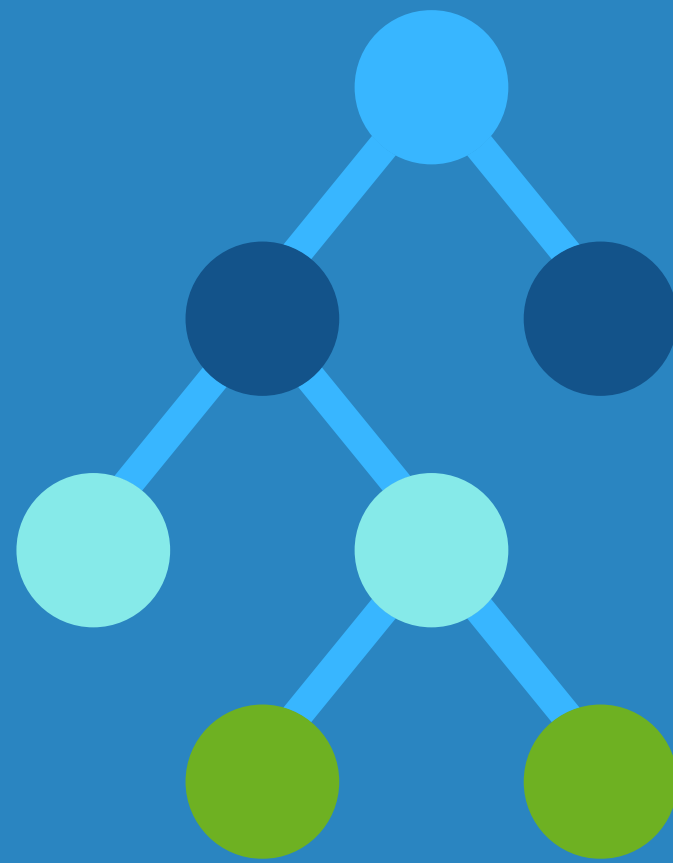


#### Estrategia de eliminación:

Para eliminar un elemento en un ABB, seguimos los siguientes pasos:

- Paso 1: Búsqueda del nodo a eliminar.
  - Comenzamos en la raíz y buscamos el nodo que contiene el elemento a eliminar.
- Paso 2: Casos de eliminación:
  - **Caso 1: Nodo con subárbol izquierdo.**
    - Reemplazamos el nodo con el mayor elemento de su subárbol izquierdo.
  - **Caso 2: Nodo con subárbol derecho.**
    - Reemplazamos el nodo con el menor elemento de su subárbol derecho.
  - **Caso 3: Nodo sin subárboles.**
    - Es una hoja y se reemplaza por null.
- Paso 3: Mantenimiento del orden del ABB.
  - Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

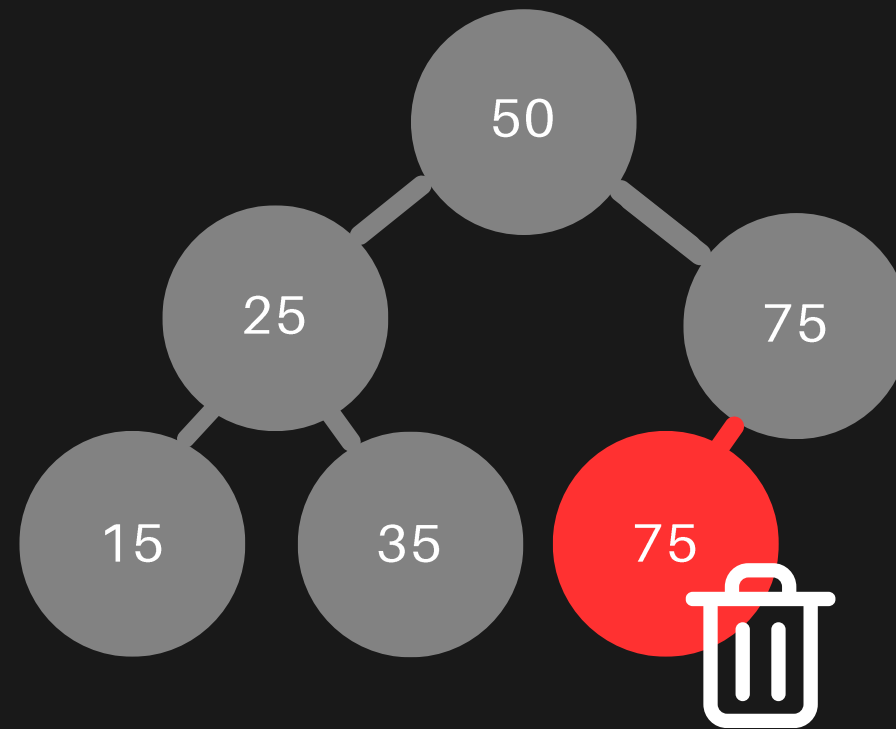
# eliminarElem()



## Árboles binarios de búsqueda [ABB]

### ABB TDA

**eliminarElem:** Remueve el elemento dado manteniendo la propiedad de ABB (se asume que el árbol está inicializado).

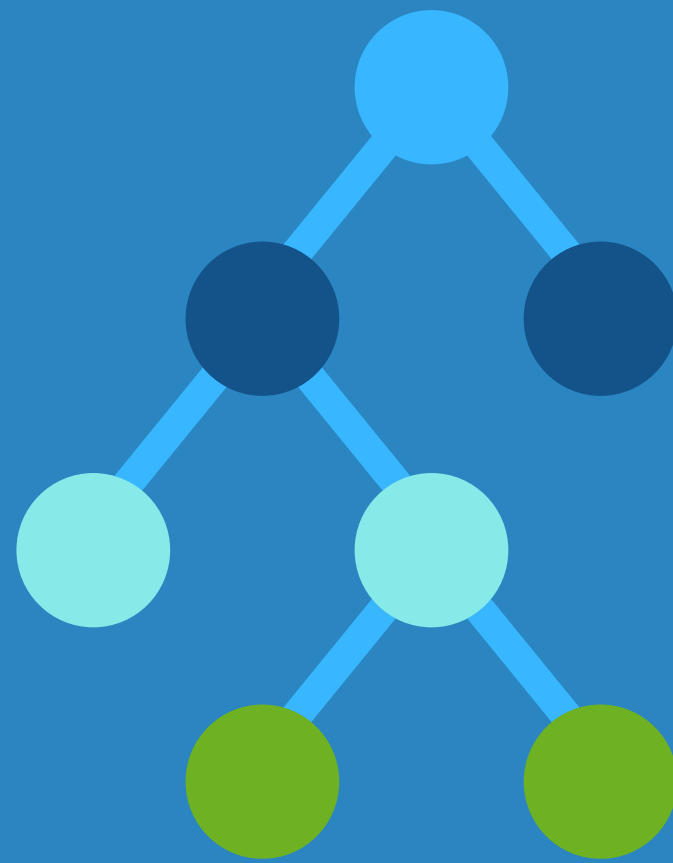


#### Estrategia de eliminación:

Para eliminar un elemento en un ABB, seguimos los siguientes pasos:

- Paso 1: Búsqueda del nodo a eliminar.
  - Comenzamos en la raíz y buscamos el nodo que contiene el elemento a eliminar.
- Paso 2: Casos de eliminación:
  - **Caso 1: Nodo con subárbol izquierdo.**
    - Reemplazamos el nodo con el mayor elemento de su subárbol izquierdo.
  - **Caso 2: Nodo con subárbol derecho.**
    - Reemplazamos el nodo con el menor elemento de su subárbol derecho.
  - **Caso 3: Nodo sin subárboles.**
    - Es una hoja y se reemplaza por null.
- Paso 3: Mantenimiento del orden del ABB.
  - Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

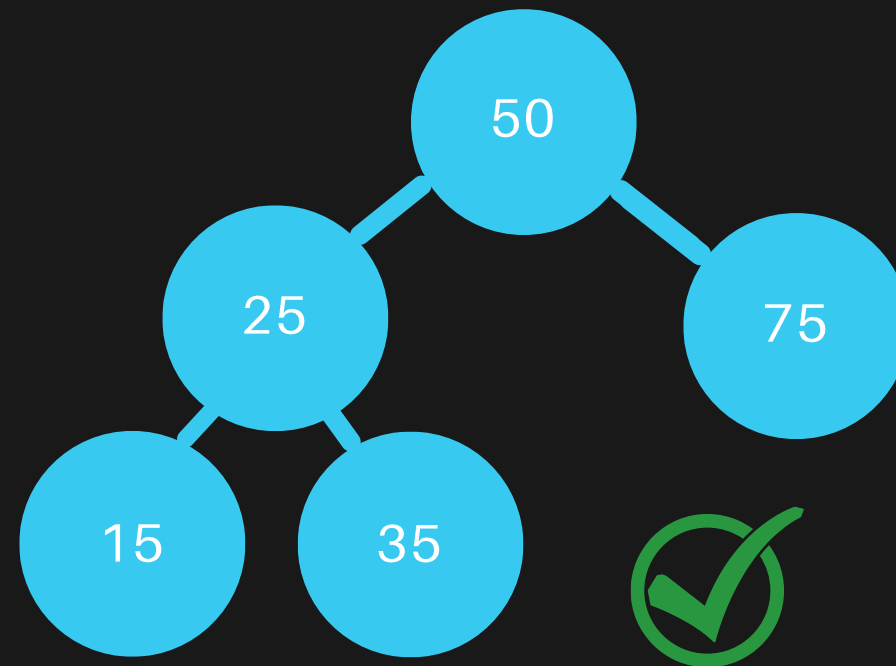
# eliminarElem()



## Árboles binarios de búsqueda [ABB]

### ABB TDA

**eliminarElem:** Remueve el elemento dado manteniendo la propiedad de ABB (se asume que el árbol está inicializado).



#### Estrategia de eliminación:

Para eliminar un elemento en un ABB, seguimos los siguientes pasos:

- Paso 1: Búsqueda del nodo a eliminar.
  - Comenzamos en la raíz y buscamos el nodo que contiene el elemento a eliminar.
- Paso 2: Casos de eliminación:
  - **Caso 1: Nodo con subárbol izquierdo.**
    - Reemplazamos el nodo con el mayor elemento de su subárbol izquierdo.
  - **Caso 2: Nodo con subárbol derecho.**
    - Reemplazamos el nodo con el menor elemento de su subárbol derecho.
  - **Caso 3: Nodo sin subárboles.**
    - Es una hoja y se reemplaza por null.
- Paso 3: Mantenimiento del orden del ABB.
  - Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

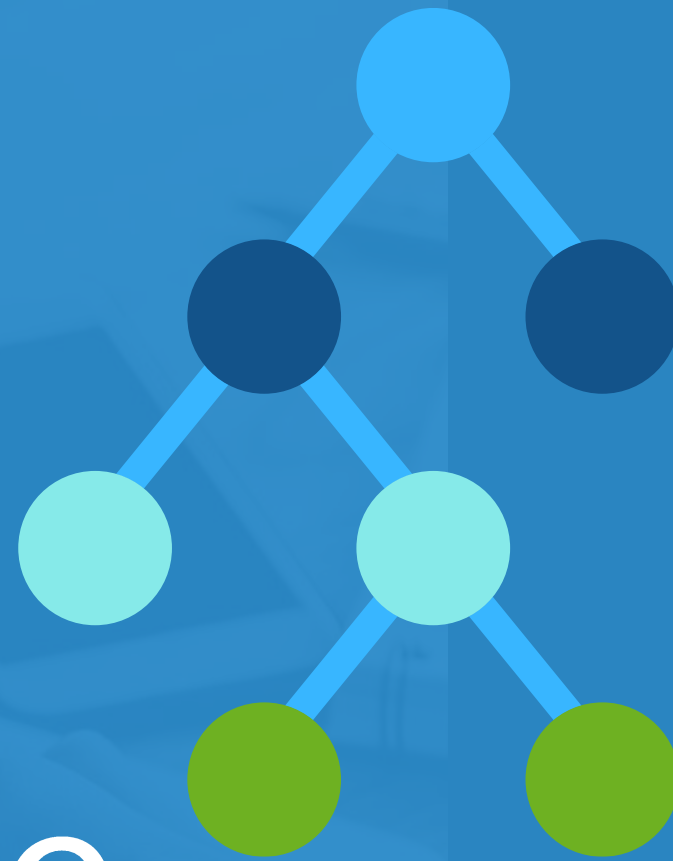


- Para calcular el costo de una búsqueda en un ABB, se analiza la **profundidad del árbol en relación con la cantidad de nodos** (tamaño de entrada).
- Durante la búsqueda, descendemos de nivel en el árbol. En el peor de los casos, se encontrará el valor en el nivel más bajo o no se encontrará.

- El costo de búsqueda en un ABB depende de cómo esté estructurado el árbol y del número de nodos presentes.
- A medida que aumenta la cantidad de nodos, la profundidad del árbol puede influir en el costo de búsqueda.

# ABB

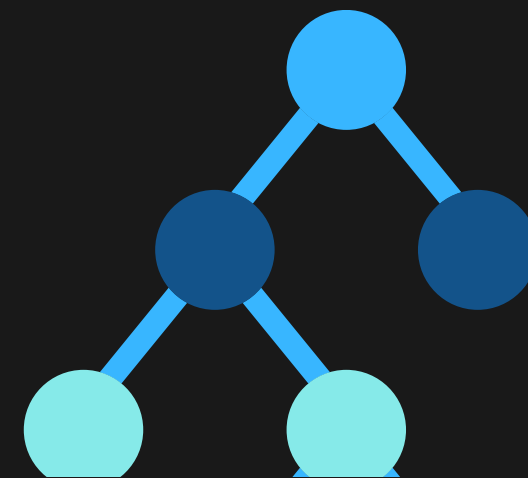
## Costo de búsqueda



## Árbol balanceado vs. Árbol degenerado:

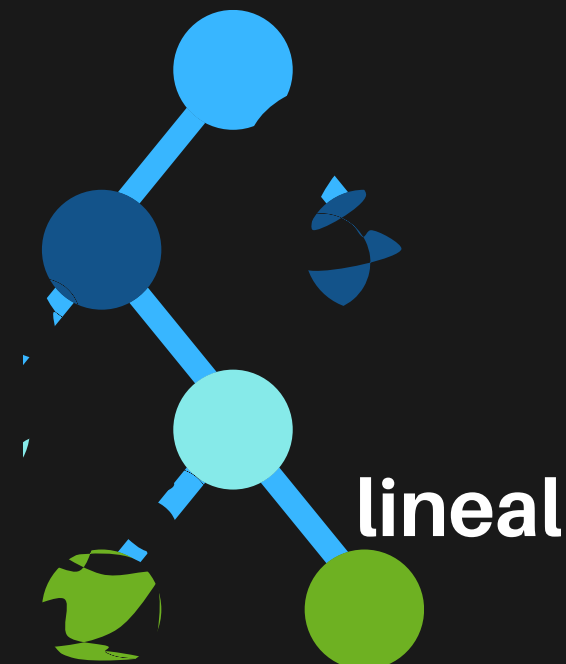
- Un árbol balanceado distribuye los nodos de manera equilibrada, lo que minimiza la profundidad y, por lo tanto, el tiempo de búsqueda.
- Un árbol degenerado, como una lista, tiene todos los nodos en un solo camino, lo que resulta en un costo de búsqueda lineal.
- Es preferible contar con un árbol balanceado al realizar búsquedas, ya que su costo es logarítmico en comparación con un árbol degenerado (lista) que tiene un costo lineal en el peor caso.

### Balanceado



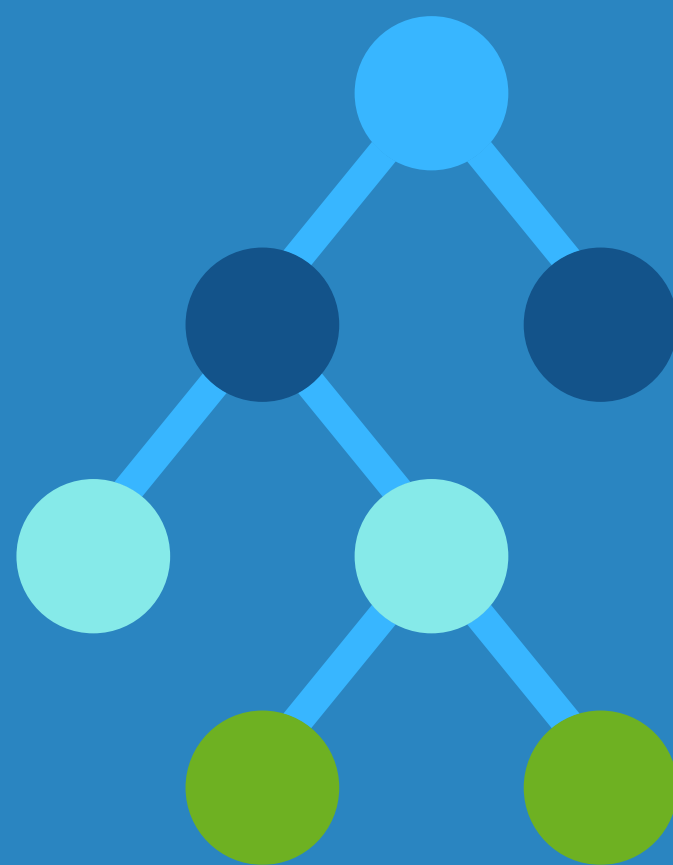
logarítmica

### Degenerado



lineal

# Implementación



Nodo

Árboles binarios  
de búsqueda  
[ABB]

ABB TDA

```
class NodoABB{  
    int info ;  
    ABBTDA hijoIzq;  
    ABBTDA hijoDer;  
}
```

la clase ABB implementa ABBTDA

```
public class ABB implements ABBTDA{  
    NodoABB raiz ;  
  
    public int raiz (){  
        return raiz.info ;  
    }  
  
    public boolean arbolVacio(){  
        return (raiz == null);  
    }  
  
    public void inicializarArbol (){  
        raiz = null ;  
    }  
  
    public ABBTDA hijoDer(){  
        return raiz.hijoDer;  
    }  
  
    public ABBTDA hijoIzq(){  
        return raiz.hijoIzq;  
    }  
}
```

obtengo la raiz del nodo

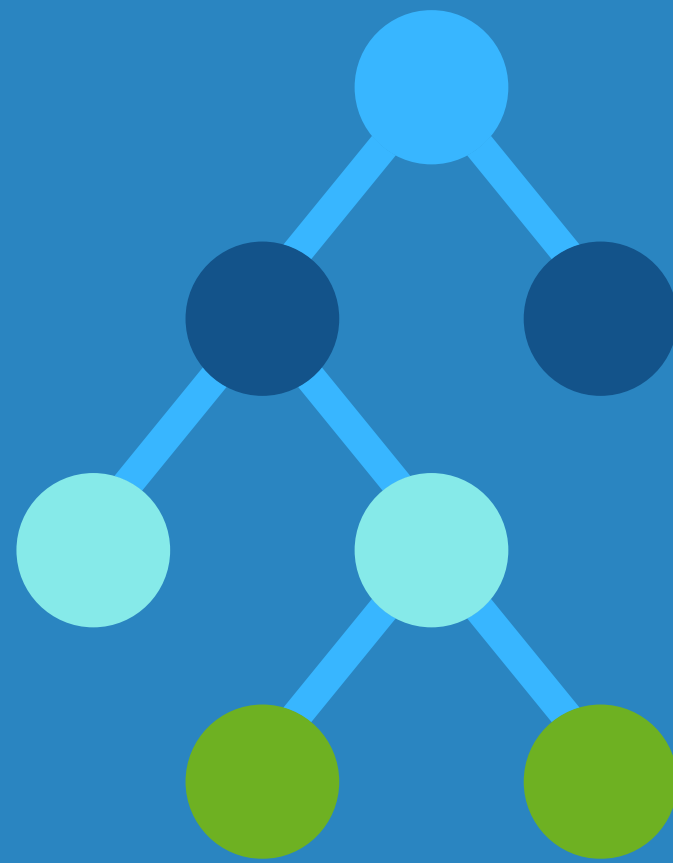
devuelve si el arbol está vacío

Inicializa el árbol

Devuelve el subárbol Derecho

Devuelve el subárbol Izquierdo

# Implementación



## Árboles binarios de búsqueda [ABB]

### ABB TDA

### Agregar

```
public void AgregarElem(int x){  
  
    if(raiz == null ){  
        raiz = new NodoABB();  
        raiz.info = x;  
        raiz.hijoIzq = new ABB();  
        raiz.hijoIzq. InicializarArbol();  
        raiz.hijoDer = new ABB();  
        raiz.hijoDer. InicializarArbol();  
    }  
    else if ( raiz.info > x )  
        raiz.hijoIzq.AgregarElem(x);  
    else if( raiz.info < x )  
        raiz.hijoDer. AgregarElem(x);  
}
```

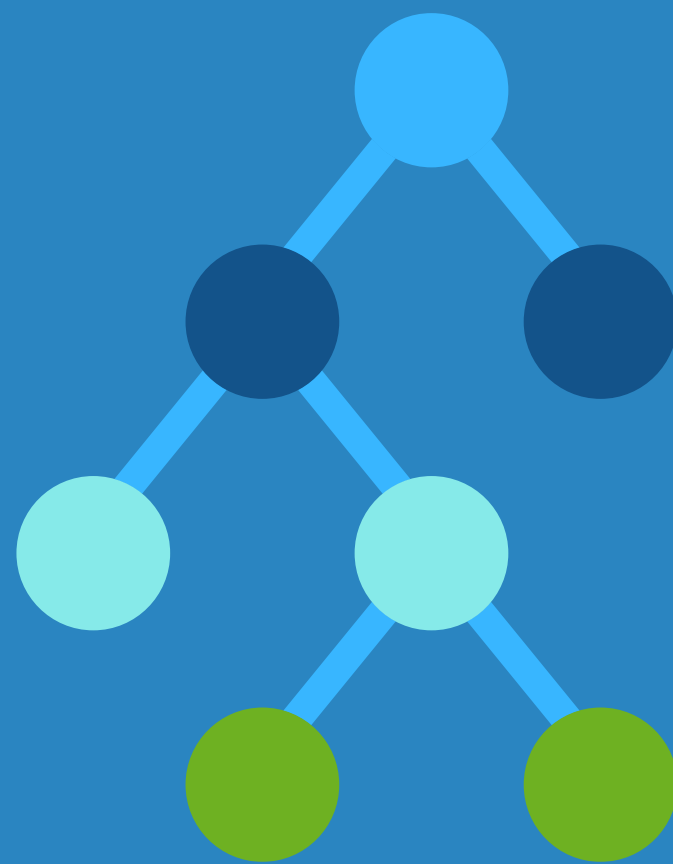
Si la raiz del nodo donde estoy es null, inicializo el árbol en el nodo, y los hijos.

Si el dato del nodo es mayor a mi número a agregar, me muevo para el subárbol izquierdo.

Si el dato del nodo es menor a mi número a agregar, me muevo para el subárbol derecho.



# Implementación



## Árboles binarios de búsqueda [ABB]

### ABB TDA

### Eliminar

```
public void EliminarElem(int x){
    if ( raiz != null ){
        if ( raiz.info == x && raiz.hijoIzq.ArbolVacio() &&
            raiz.hijoDer.ArbolVacio() ){
            raiz = null ;
        }
        else if (raiz.info == x && ! raiz.hijoIzq.ArbolVacio()){
            raiz.info = this.mayor(raiz.hijoIzq);
            raiz.hijoIzq.EliminarElem(raiz.info);
        }
        else if (raiz.info == x && raiz.hijoIzq.ArbolVacio()){
            raiz.info = this.menor(raiz.hijoDer);
            raiz.hijoDer.EliminarElem(raiz.info);
        }
        else if (raiz.info < x){
            raiz.hijoDer.EliminarElem(x);
        }
        else {
            raiz.hijoIzq.EliminarElem(x);
        }
    }
}
```

Aclaración: nunca tengo en cuenta la opción de que no exista el valor pasado por parámetro: es decir, lo ignoro porque no hay nada que hacer

Si tengo un árbol no vacío, lo primero que hago es chequear que el dato de la raíz sea igual al buscado y que tenga ambos hijos vacíos. Si es así, se trata de una hoja. Sólo la elimino.

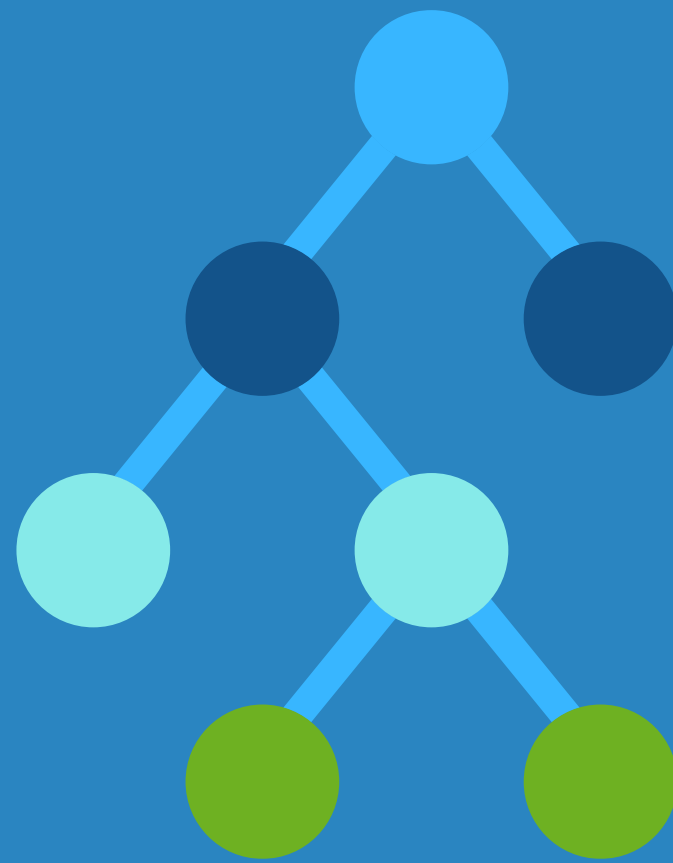
Si el dato del nodo coincide con el buscado, y además tengo un subárbol izquierdo no vacío, me quedo con el mayor dato de mi subárbol izquierdo y ejecuto eliminar para ese nodo descendiente.

Si el dato del nodo coincide con el buscado, y además tengo un subárbol izquierdo vacío, me quedo con el menor dato de mi subárbol derecho y ejecuto eliminar para ese nodo descendiente.

Si el dato del nodo es menor al buscado, sigo buscando por el lado derecho, sino por el lado izquierdo.



# Implementación



Árboles binarios  
de búsqueda  
[ABB]

ABB TDA

Mayor y Menor

```
private int mayor(ABBTDA a){  
    if (a.HijoDer().ArbolVacio ())  
        return a.Raiz();  
    else  
        return mayor(a.HijoDer());  
}
```

```
private int menor(ABBTDA a){  
    if (a.HijoIzq().ArbolVacio())  
        return a.Raiz();  
    else  
        return menor(a.HijoIzq());  
}
```

# Ejercicio de Clase

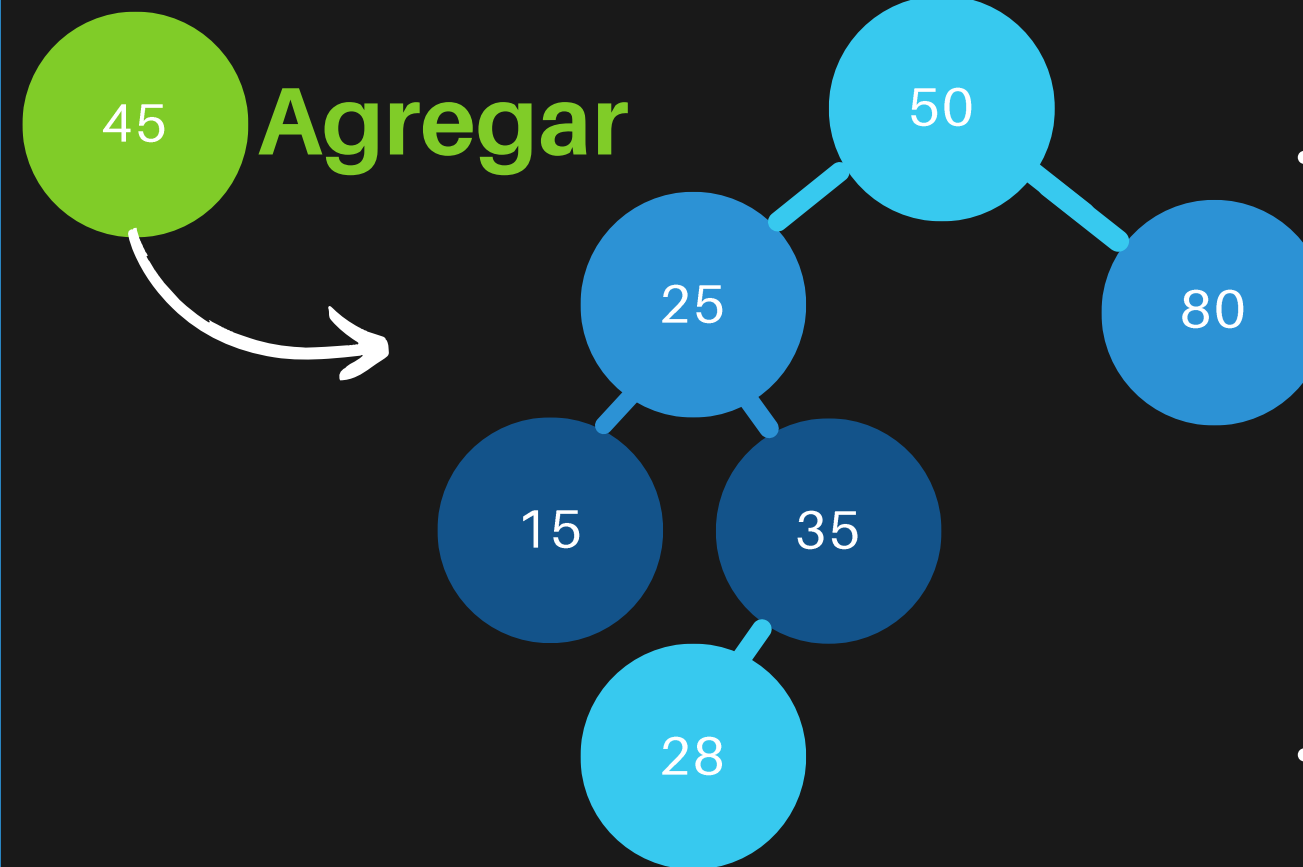
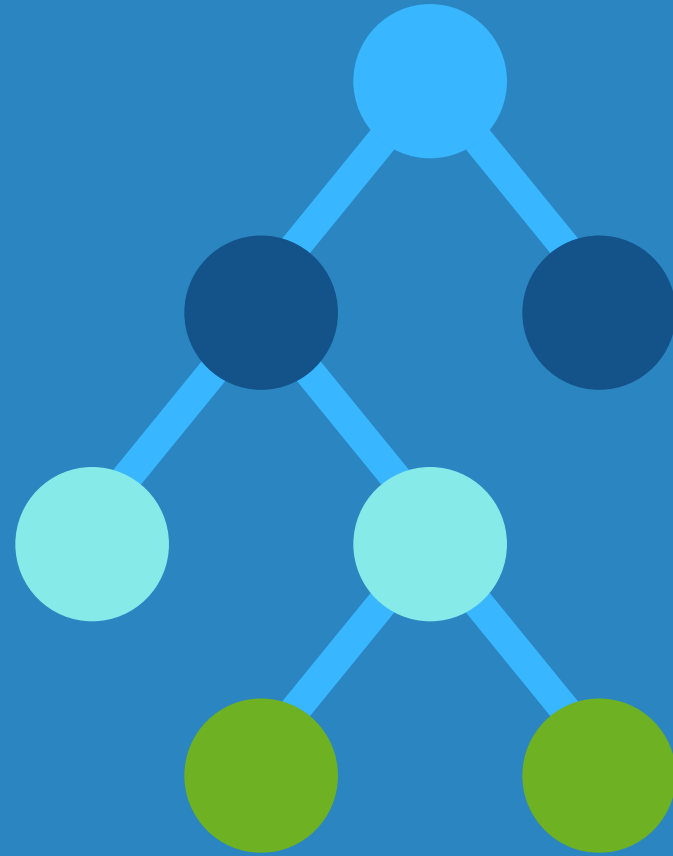
## Paso a paso para agregar

- **agregarElem**: Agrega el elemento dado manteniendo la propiedad de ABB (se *asume que el árbol está inicializado*).

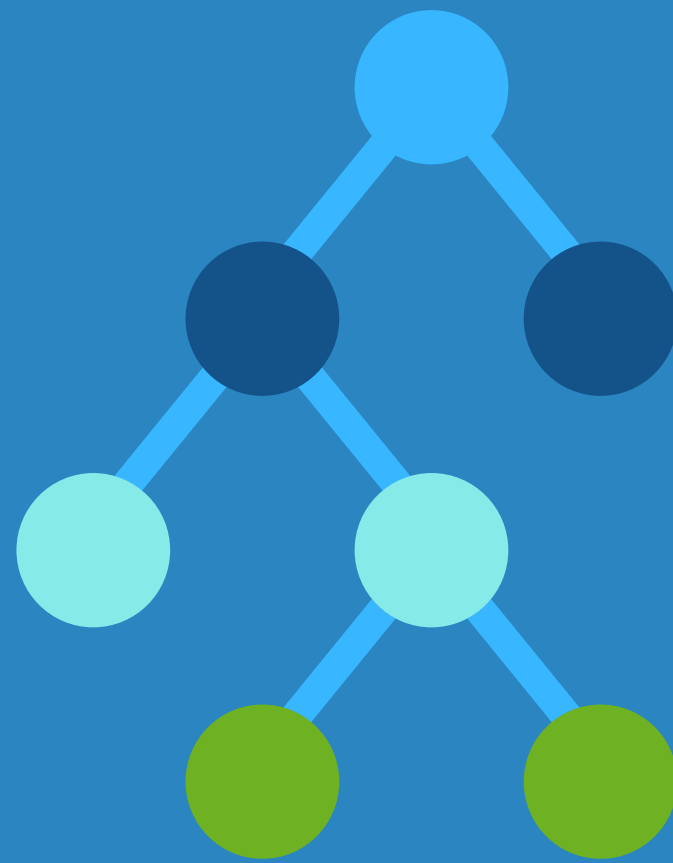
### Estrategia de inserción

- **Paso 1**: Comenzamos en la raíz del árbol.
  - Si el **árbol está vacío**, el nuevo elemento se convierte en la raíz.
- **Paso 2**: Comparamos el nuevo elemento con el valor del nodo actual.
  - Si el **nuevo elemento es menor**, nos movemos hacia el subárbol izquierdo.
  - Si el nuevo elemento es mayor, nos movemos hacia el subárbol derecho.
  - Continuamos este proceso hasta encontrar una posición adecuada.
- **Paso 3**: Llegamos a un nodo vacío o nulo.
  - Insertamos el **nuevo elemento en ese nodo como una hoja**.
- **Paso 4**: **Mantenemos la propiedad de orden del ABB.**
- Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

## Árboles binarios de búsqueda [ABB]

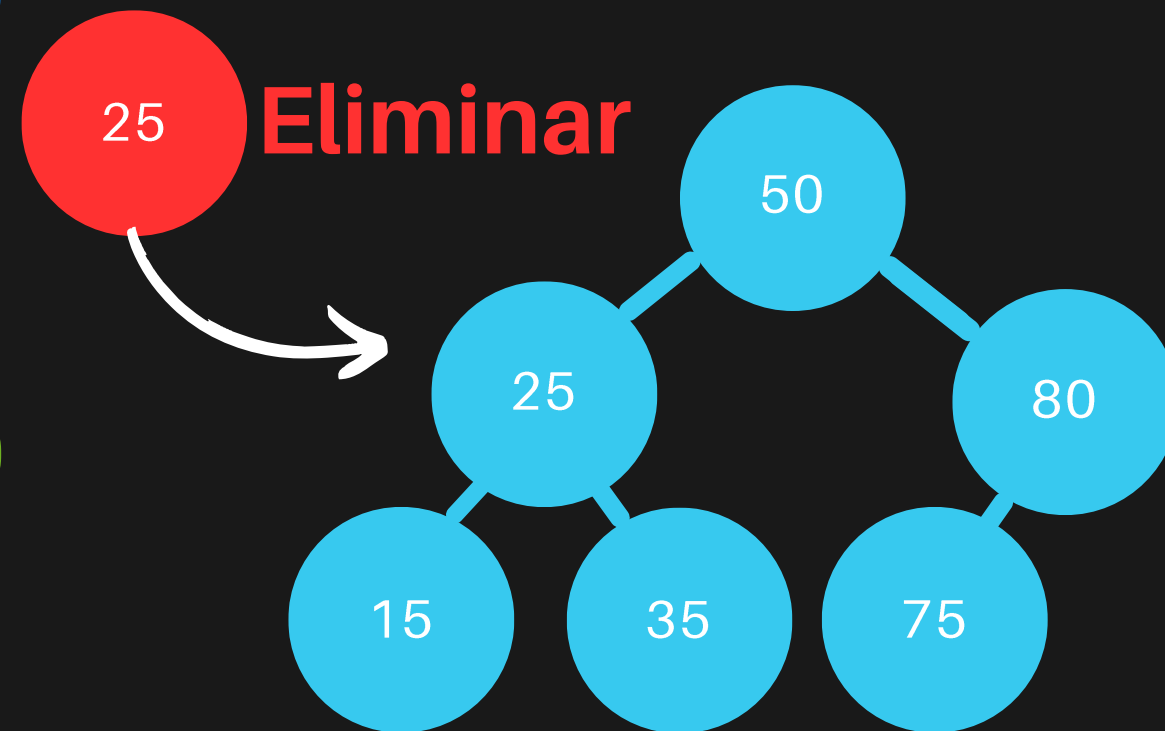


# Ejercicio de Clase **Paso a Paso para Eliminar**



## Árboles binarios de búsqueda [ABB]

**eliminarElem:** Remueve el elemento dado manteniendo la propiedad de ABB (se asume que el árbol está inicializado).



### Estrategia de eliminación:

Para eliminar un elemento en un ABB, seguimos los siguientes pasos:

- Paso 1: Búsqueda del nodo a eliminar.
  - Comenzamos en la raíz y buscamos el nodo que contiene el elemento a eliminar.
- Paso 2: Casos de eliminación:
  - **Caso 1: Nodo con subárbol izquierdo.**
    - Reemplazamos el nodo con el mayor elemento de su subárbol izquierdo.
  - **Caso 2: Nodo con subárbol derecho.**
    - Reemplazamos el nodo con el menor elemento de su subárbol derecho.
  - **Caso 3: Nodo sin subárboles.**
    - Es una hoja y se reemplaza por null.
- Paso 3: Mantenimiento del orden del ABB.
  - Aseguramos que los elementos menores estén en el subárbol izquierdo y los mayores en el subárbol derecho.

# Gracias!

PROGRAMACIÓN II

Ing. Tomás Montero Swinnen

## Bibliografía

- **Programación II - Apuntes de Cátedra**  
Versión 1.3  
Cuadrado Estrebou, María Fernanda  
Trutner, Guillermo Hernán
- **Data Structures & Algorithms in Java**  
Robert Lafore  
Sams, 1998

UADE