



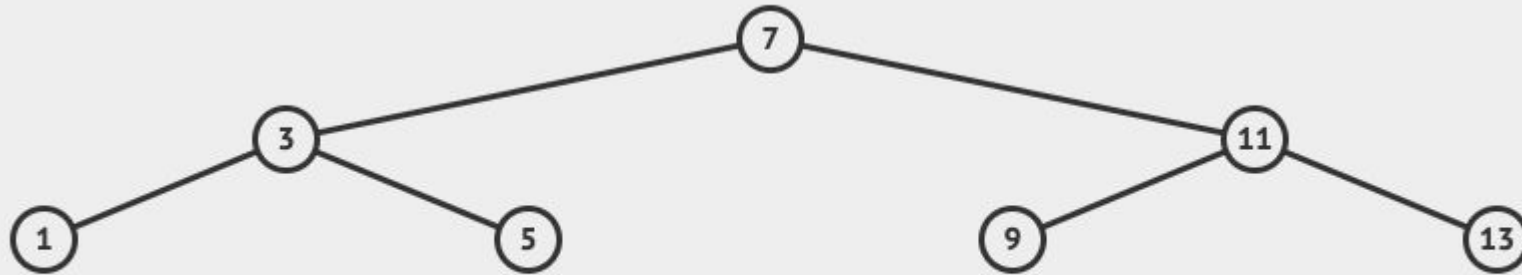
# ÁRBOLES AVL

(versión online)



# Árboles Binarios de búsqueda

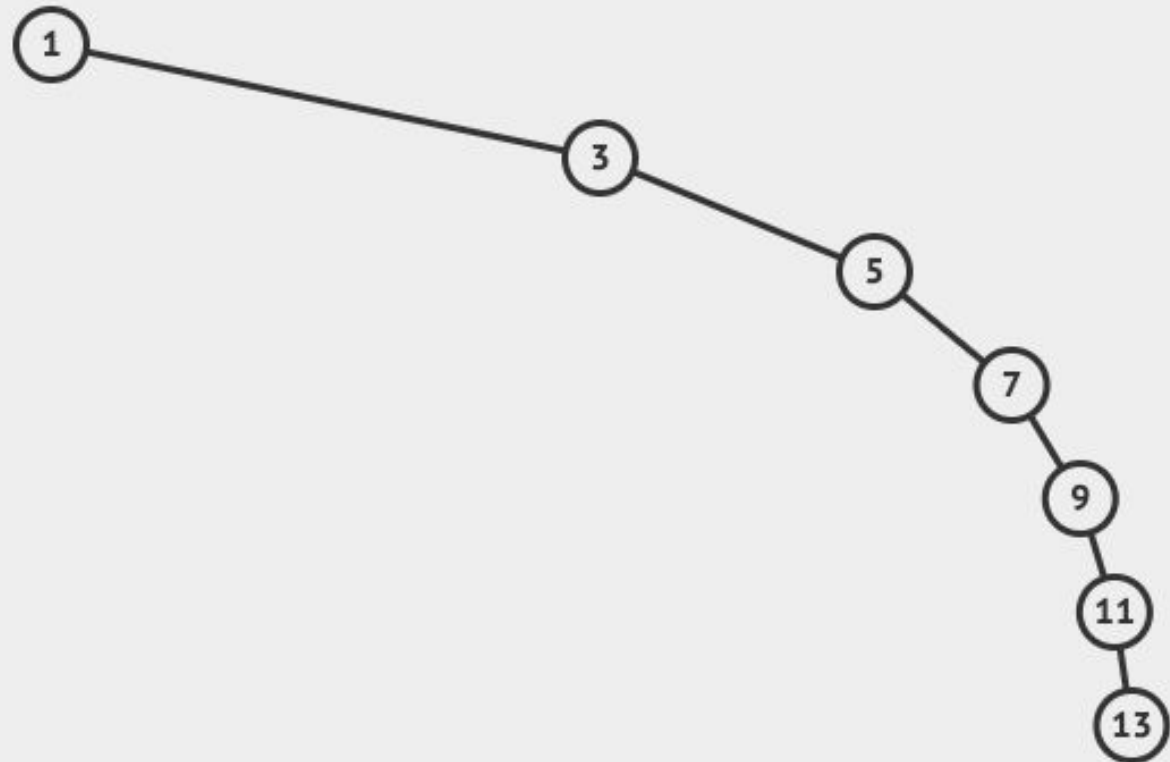
- **Tiempos de inserción, borrado y búsqueda:**
  - **Caso general:**  $O(\log n) \rightarrow n$  es la cantidad de elementos



**Claves: 7,3,11,1,5,9,13**

# Árboles Binarios de búsqueda

- Tiempos de inserción, borrado y búsqueda:
  - Peor caso:  $O(n)$  → caso “lista”



**Claves: 1, 3, 5, 7, 9, 11, 13**

# Propósito de este PPT

- Reconocer si un AB se trata o no de un AVL.
- Saber cuándo hay que balancear (diferencia de altura).
- Saber cómo se calcula la altura de un ab.
- Saber cómo seleccionar qué rotación usar.
- Saber cómo elegir las variables de la rotación.
- Ejemplos.

# AVL (**A**delson-**V**elsky & **L**andis)

- Un árbol AVL es un tipo especial de árbol binario ideado por los matemáticos rusos Georgy **A**delson-**V**elsky y Yevgueni **L**andis.
- Fue el primer árbol de búsqueda binario auto-balanceable que se ideó.
- Un árbol AVL nos garantiza que para cada nodo, la diferencia entre la altura de sus hijos es igual o menor a 1.

# ¿Cuándo balanceamos?

Cuando la diferencia de la altura (h) de sus hijos es mayor o igual a 2:

$|h(a \rightarrow \text{izq}) - h(a \rightarrow \text{der})| = 2 \rightarrow \text{HAY QUE BALANCEAR}$

$|h(a \rightarrow \text{izq}) - h(a \rightarrow \text{der})| \leq 1 \rightarrow \text{ESTA BALANCEADO}$

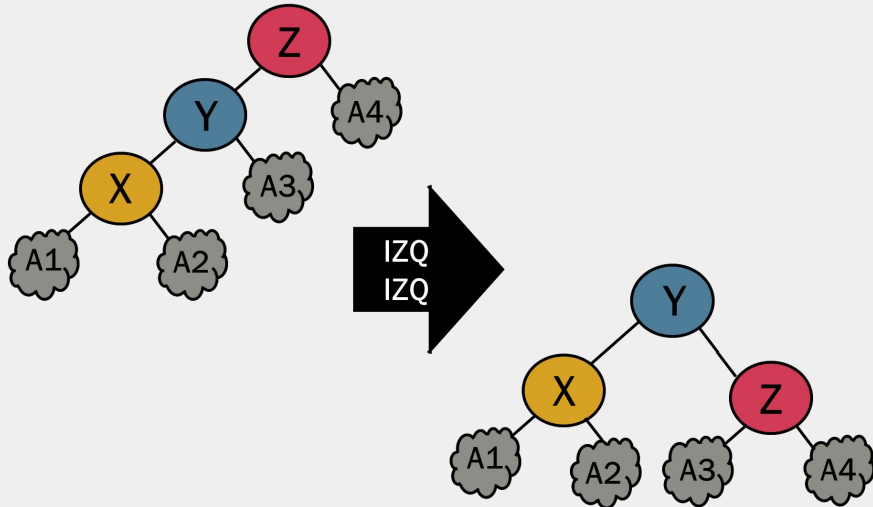
# ¿Cómo calculamos la altura?

La altura es el máximo entre la altura de sus hijos más si mismo:

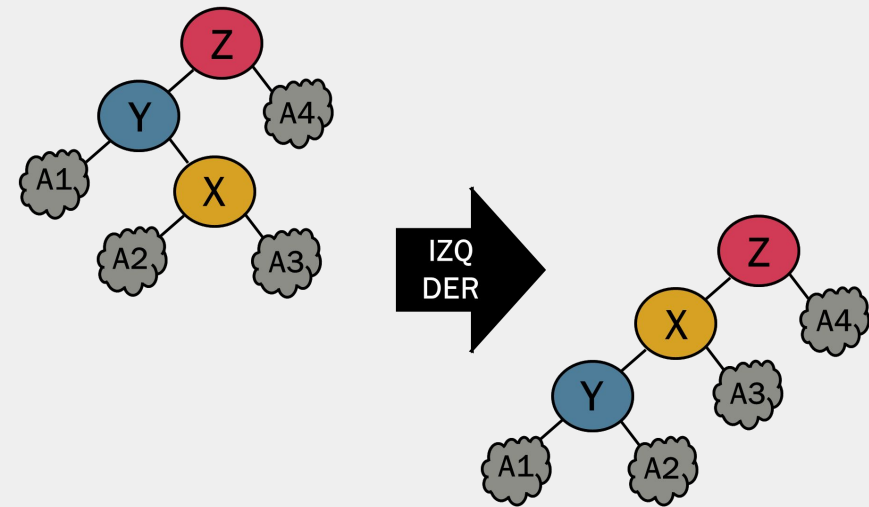
$$h(a) = \max(h(a \rightarrow \text{izq}), h(a \rightarrow \text{der})) + 1$$

# ROTACIONES

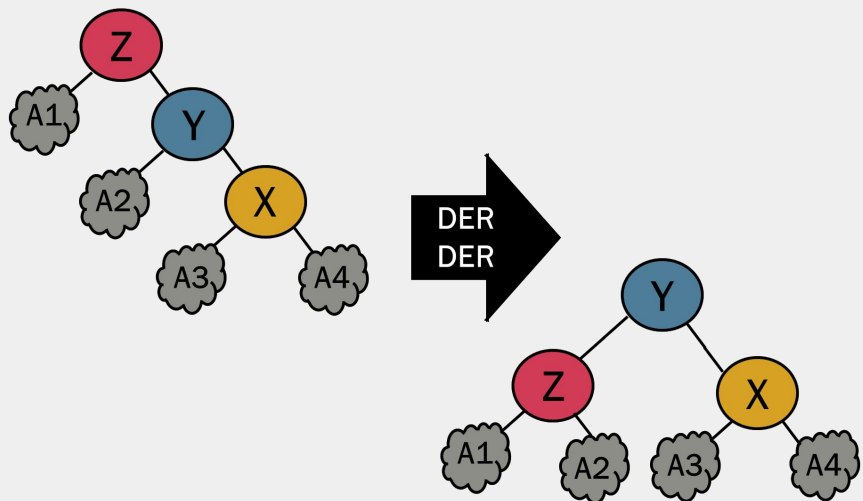
Izquierda-Izquierda (I-I)



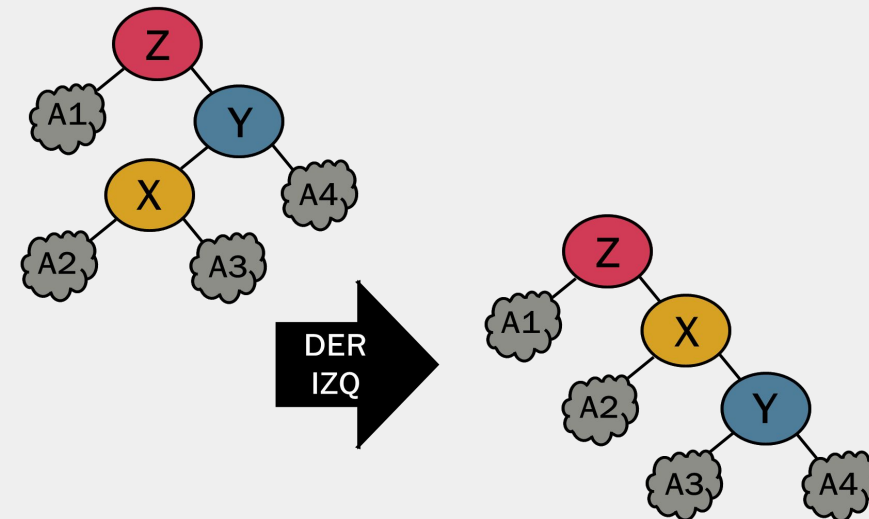
Izquierda-Derecha (I-D)

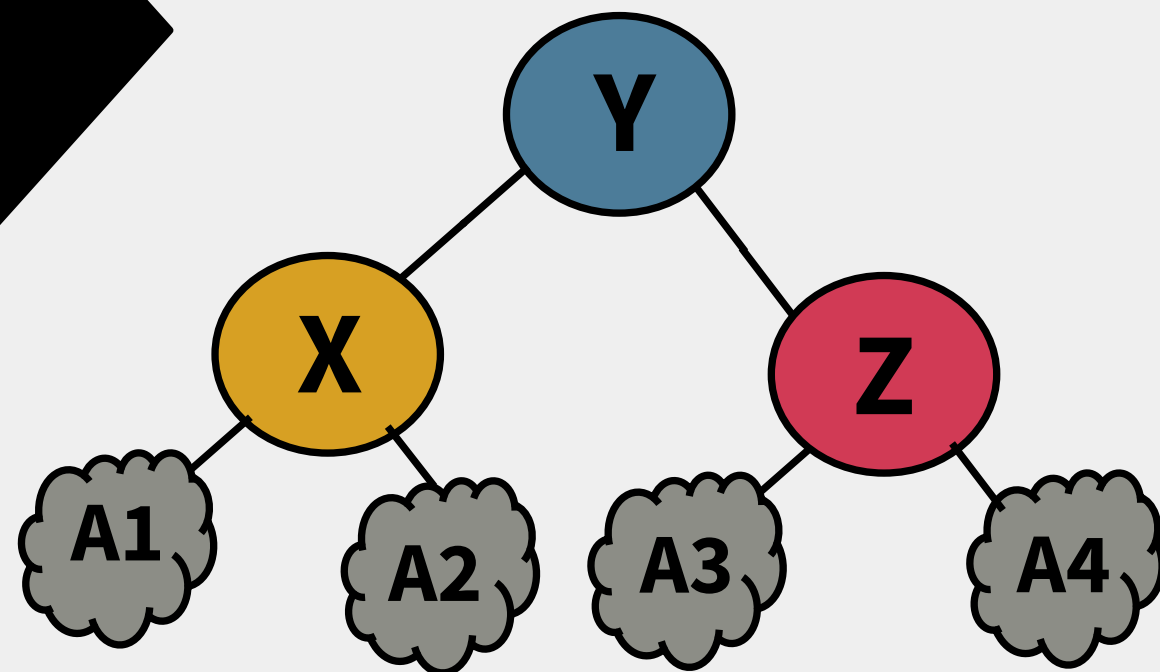
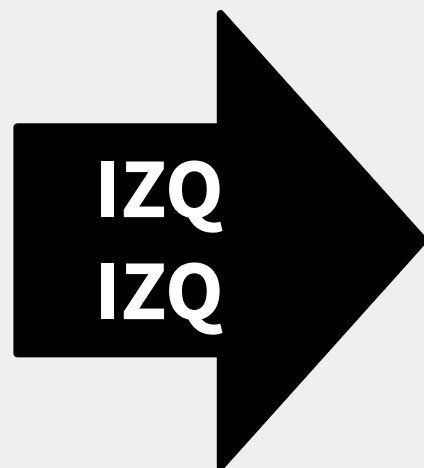
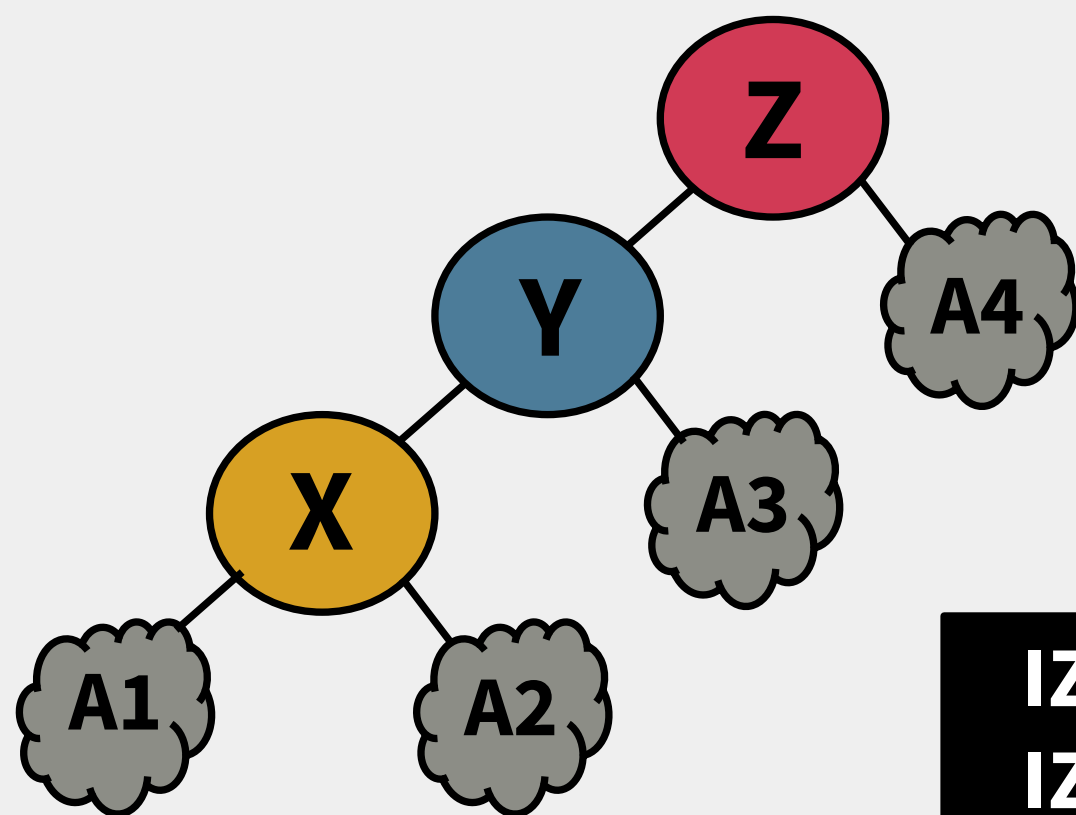


Derecha-Derecha (D-D)

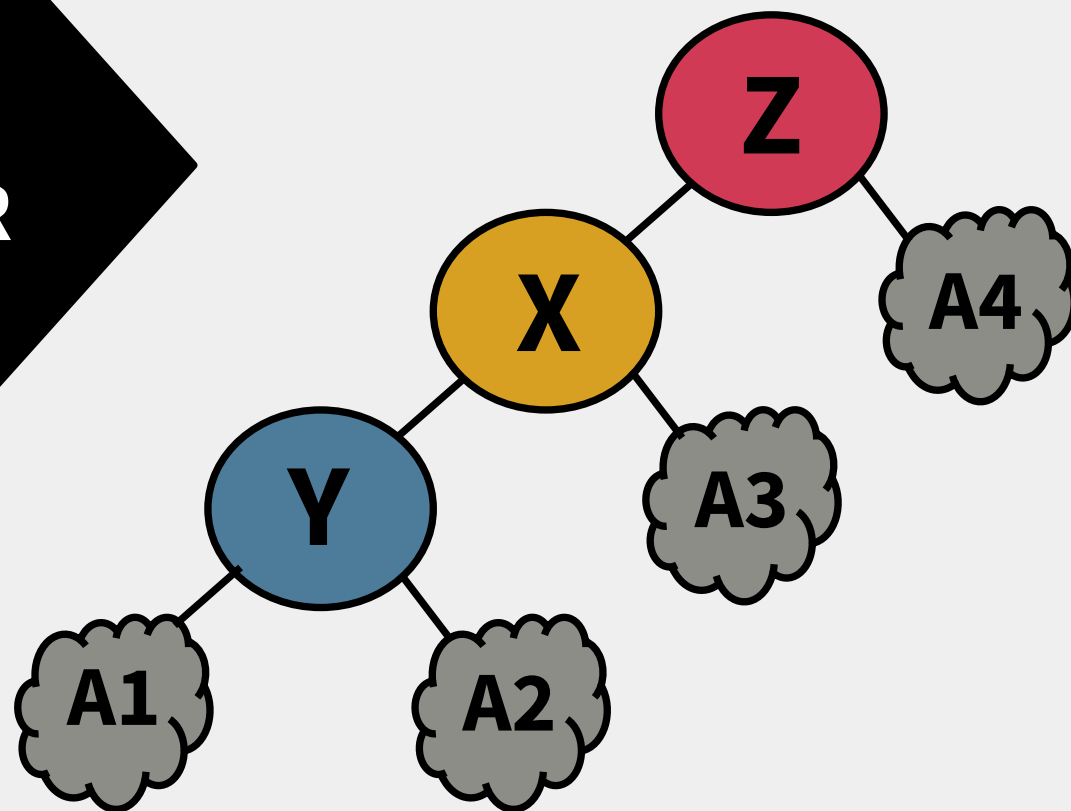
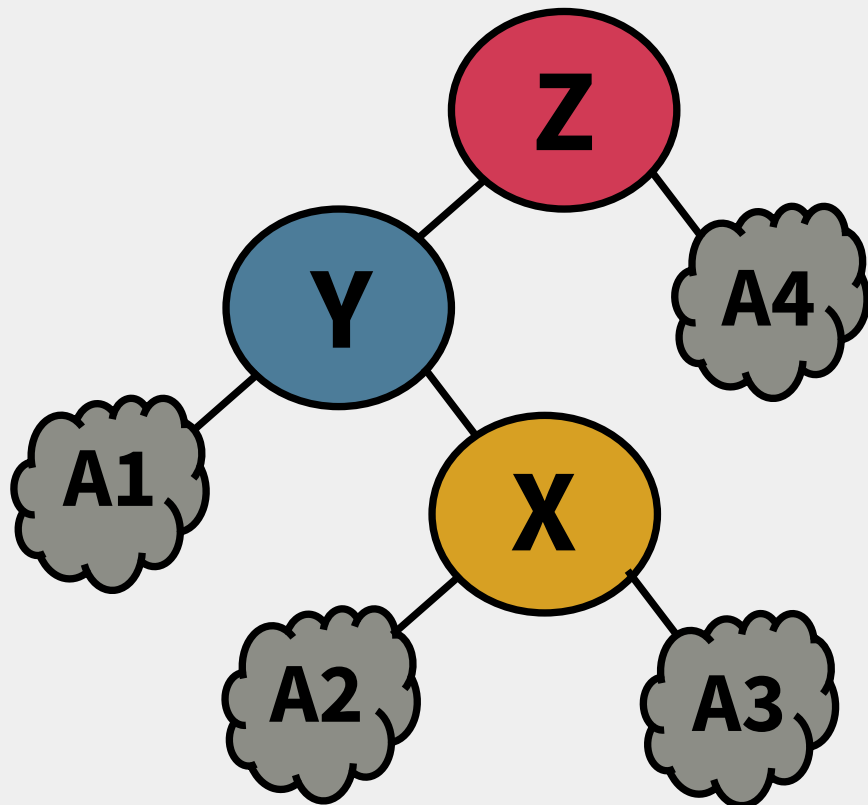


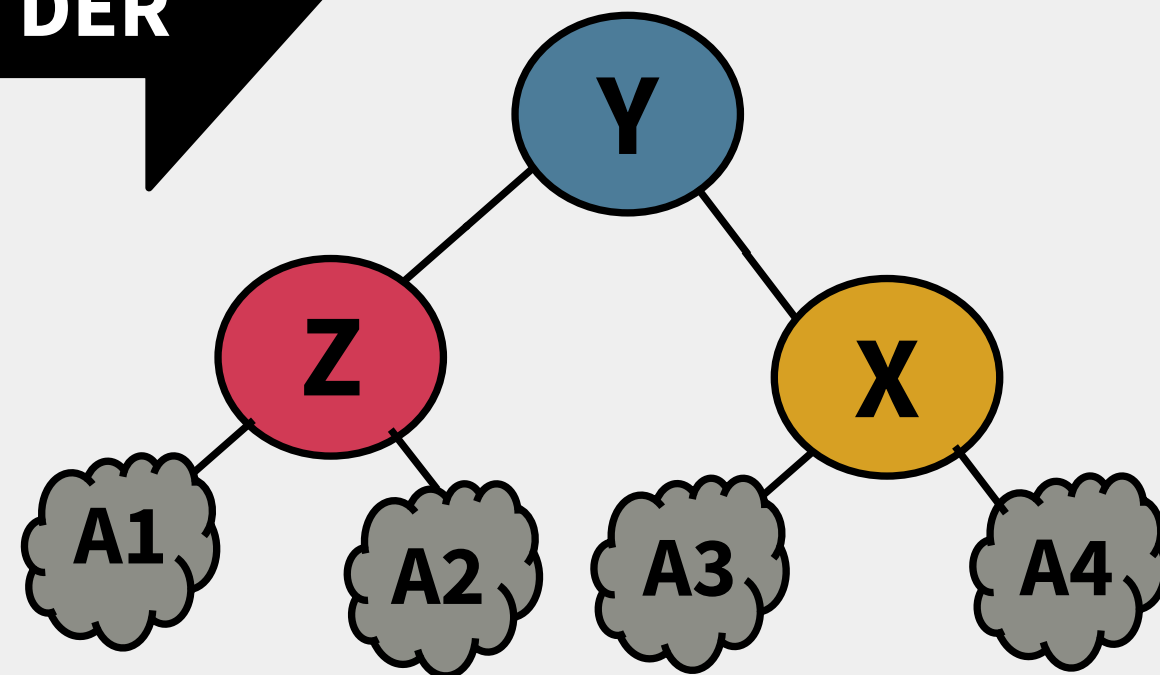
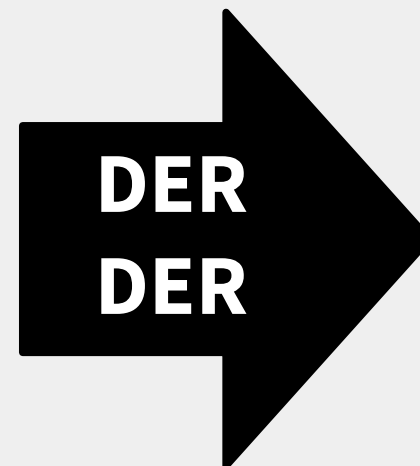
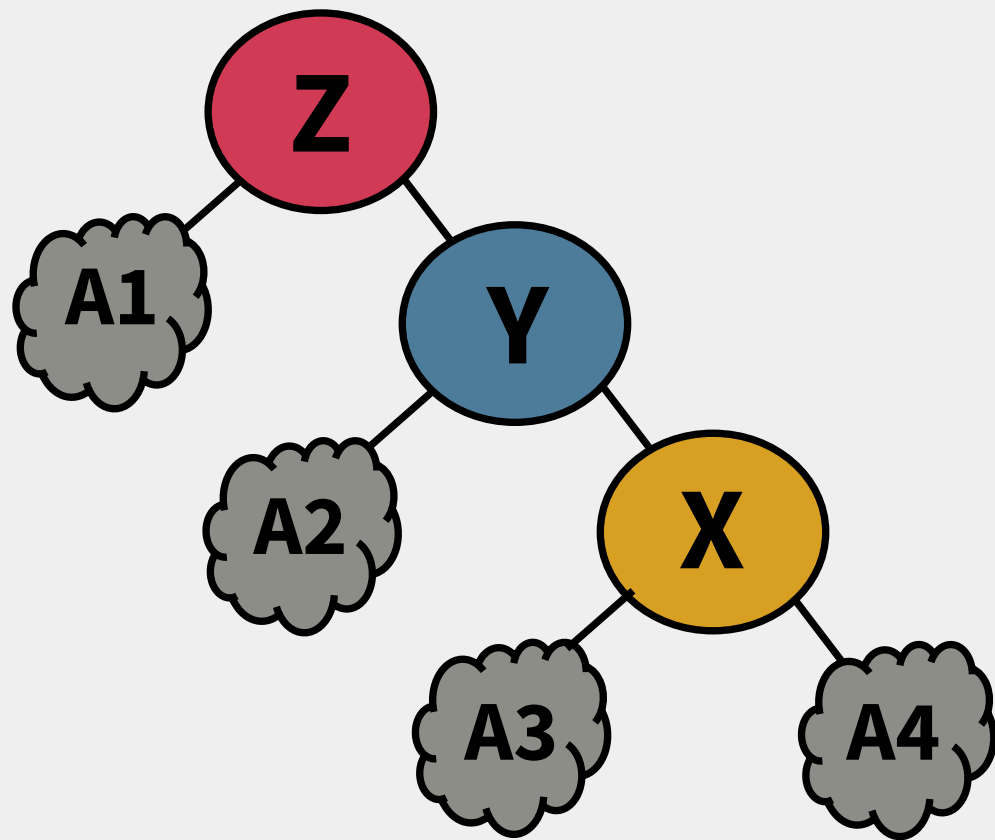
Derecha-Izquierda (D-I)

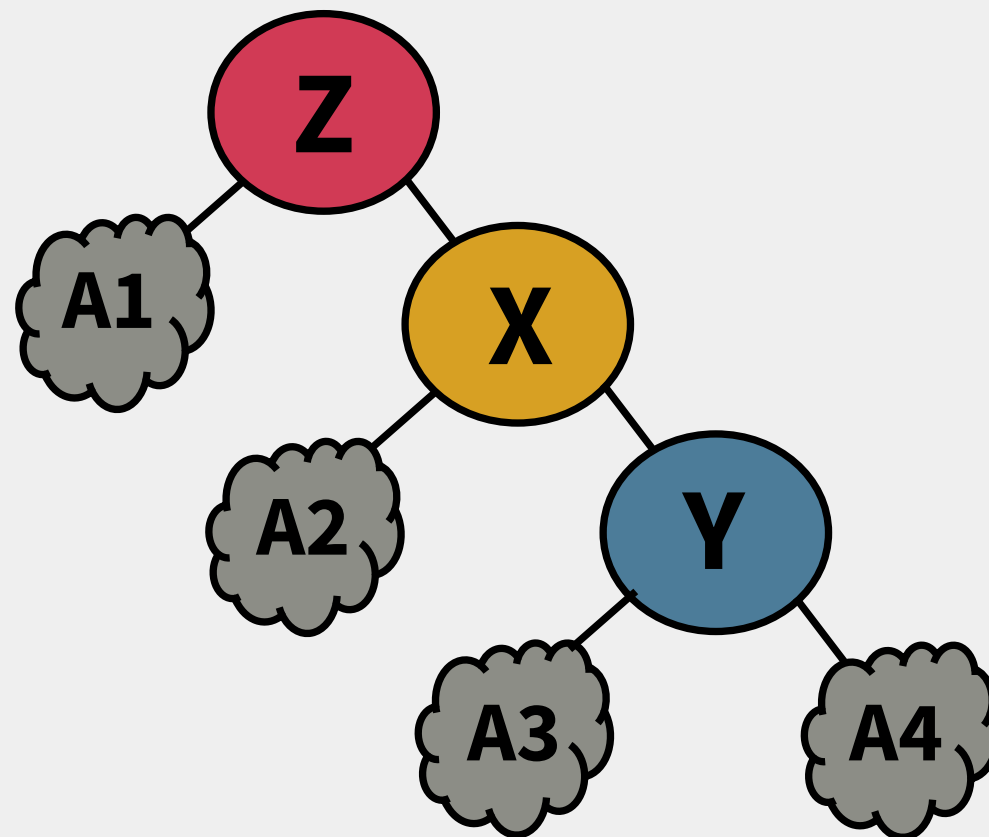
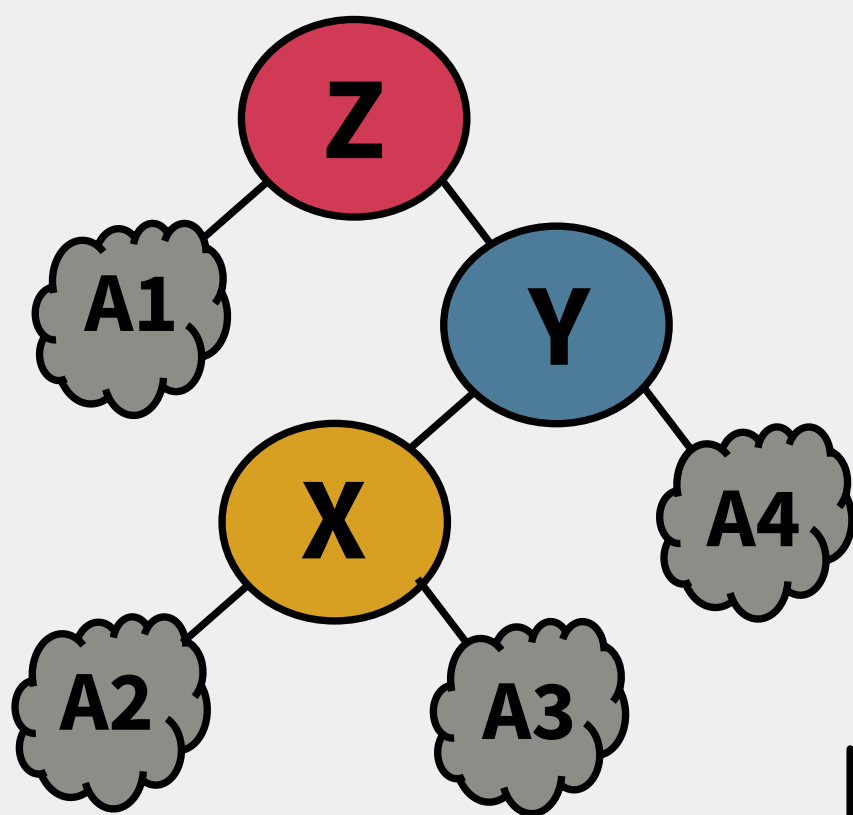




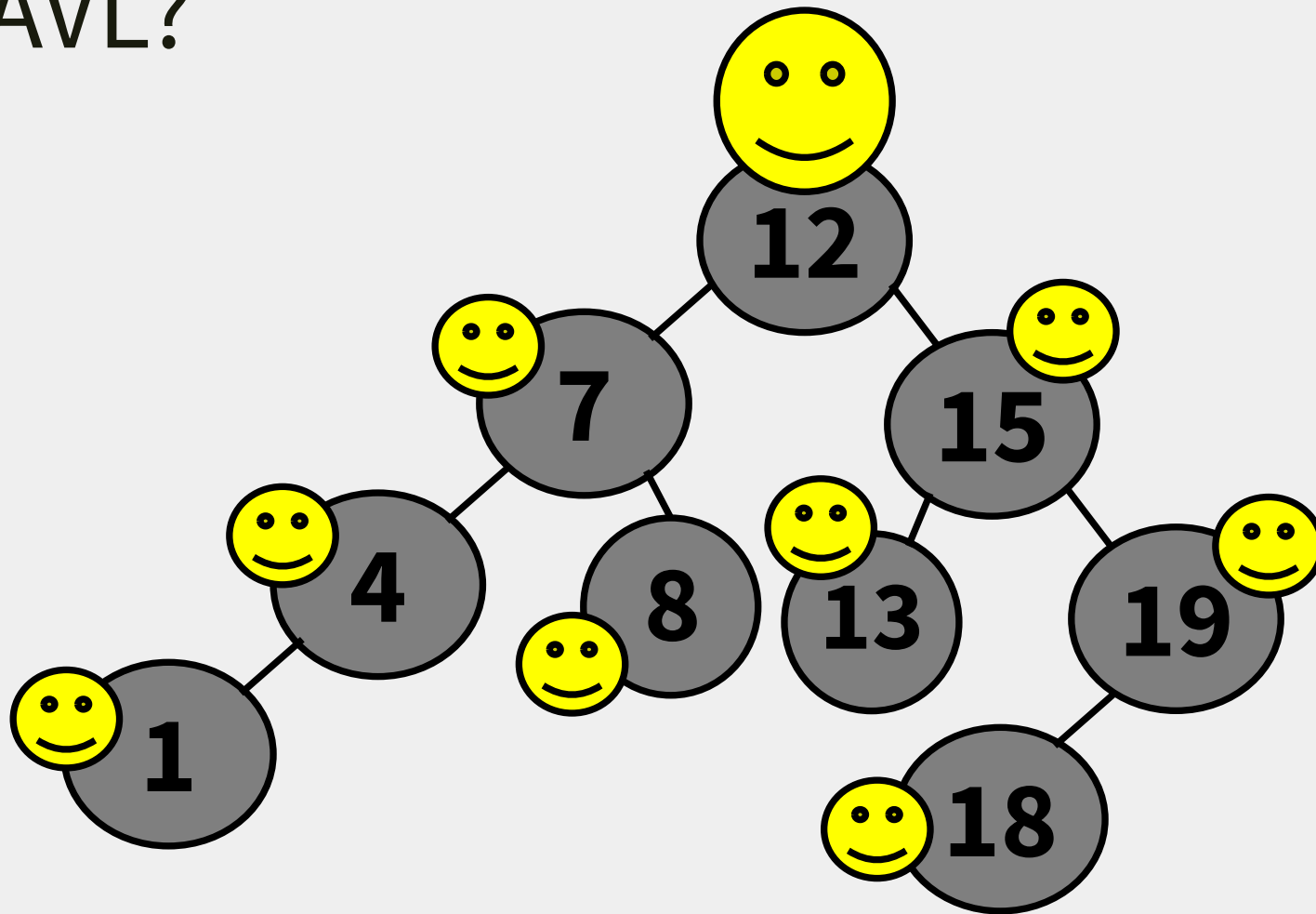








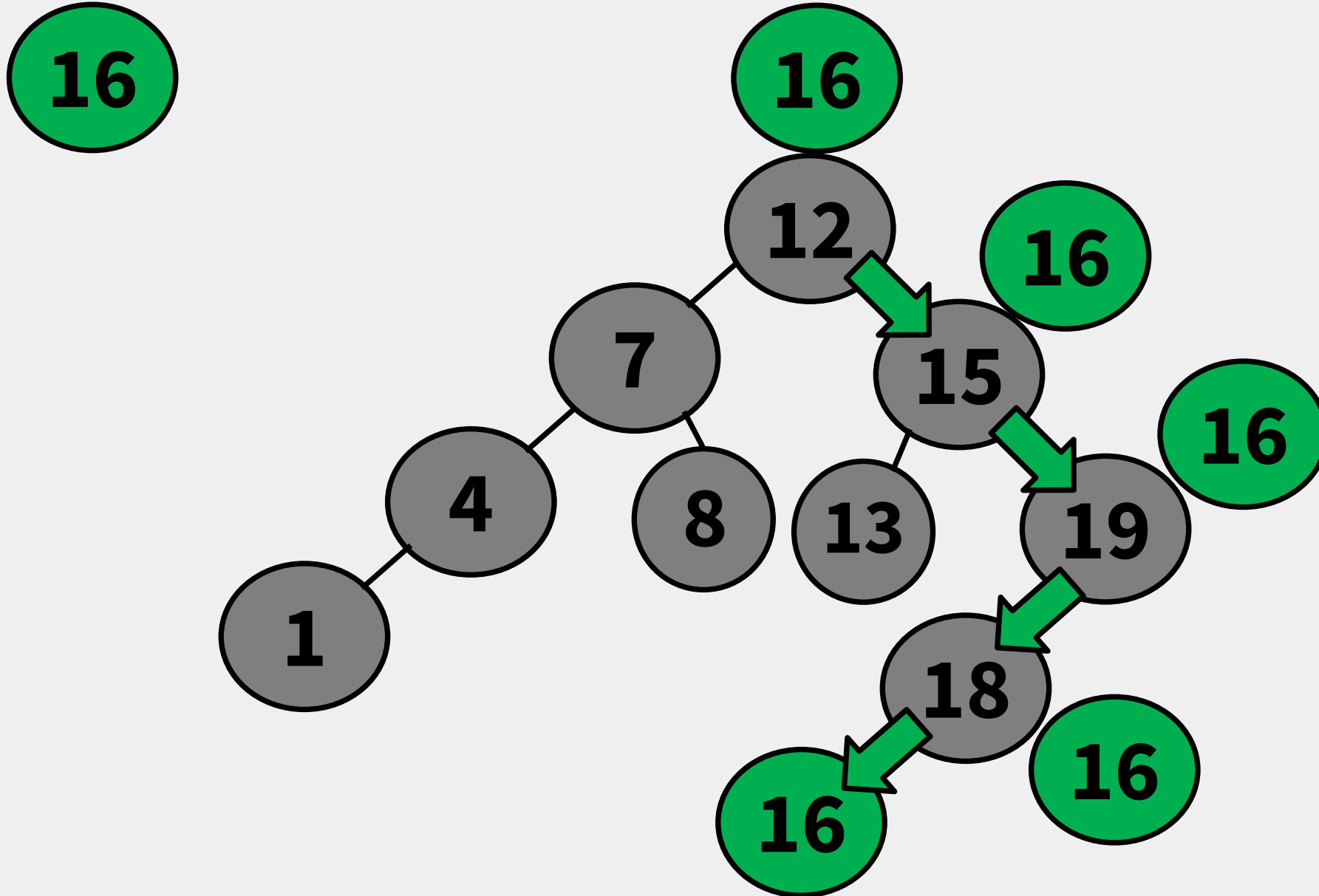
Tenemos el siguiente árbol, ¿se trata de un AVL?



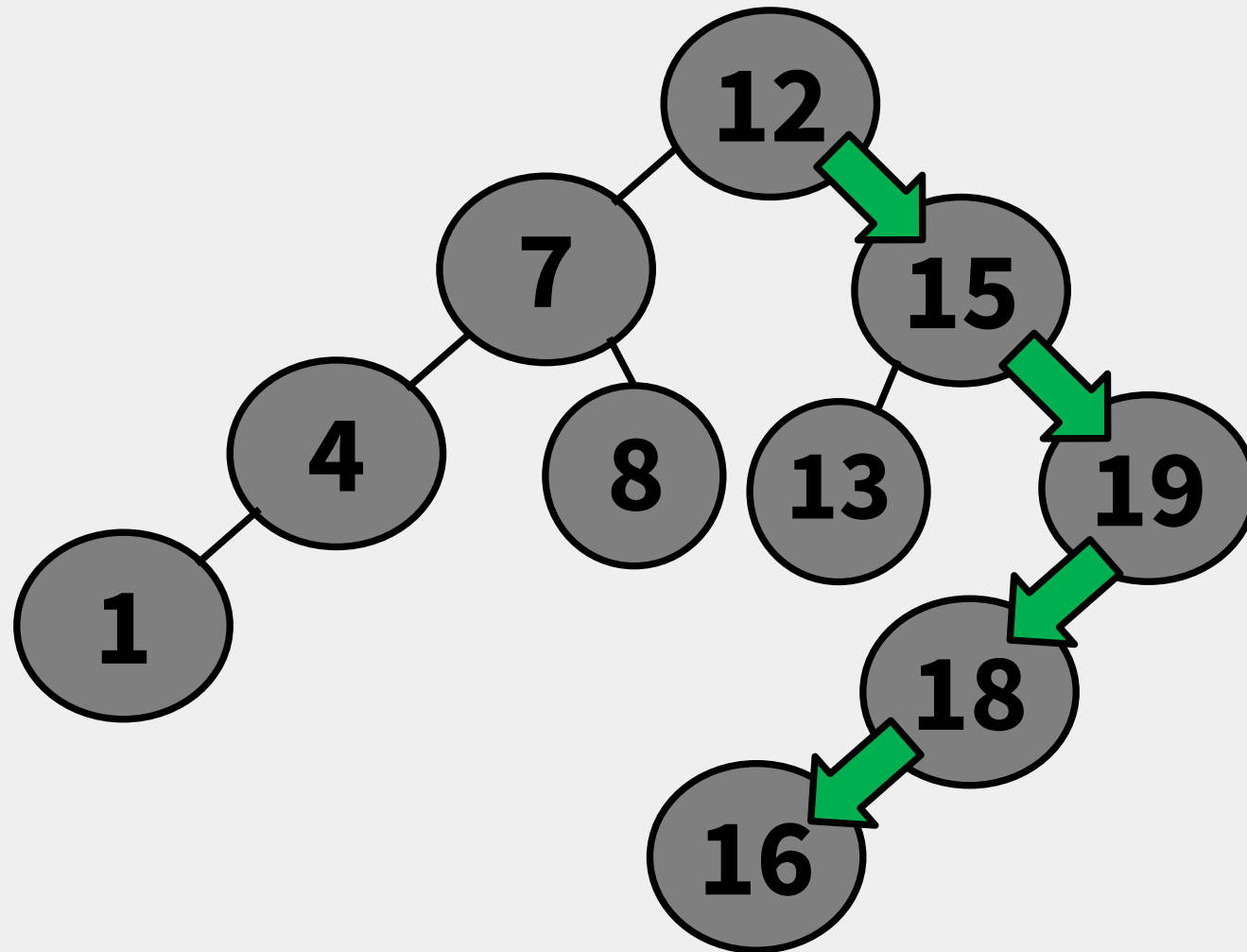
Al insertar, a lo sumo, sólo puede romperse el balanceado en algún nodo que se encuentre sobre el camino de inserción.

Veamos un ejemplo ..

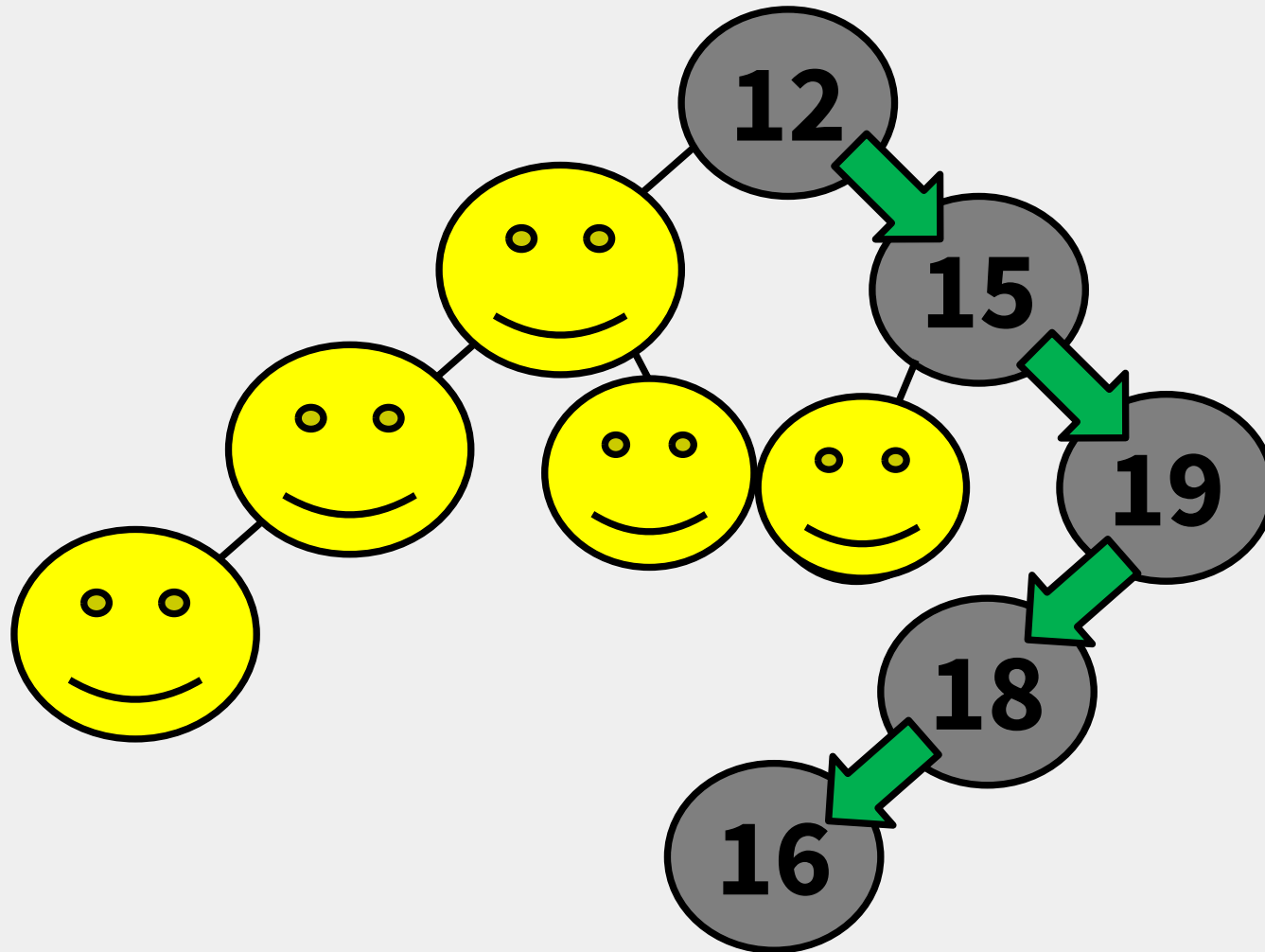
Ahora queremos insertar al 16:



Hay que volver a verificar que este balanceado, pero no sobre todo el árbol, sino sólo sobre el camino de inserción.

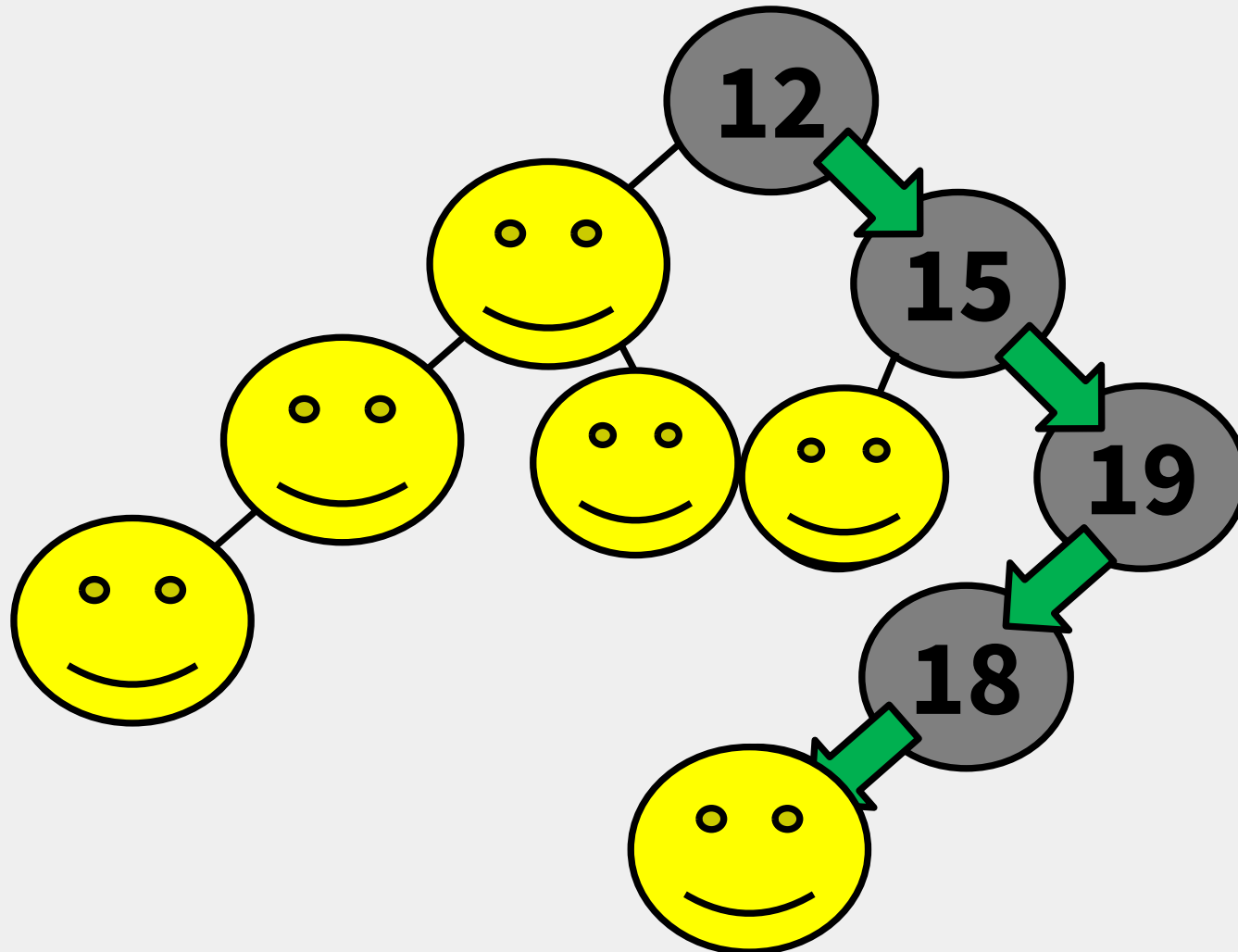


Hay que volver a verificar que este balanceado, pero no sobre todo el árbol, sino sólo sobre el camino de inserción.

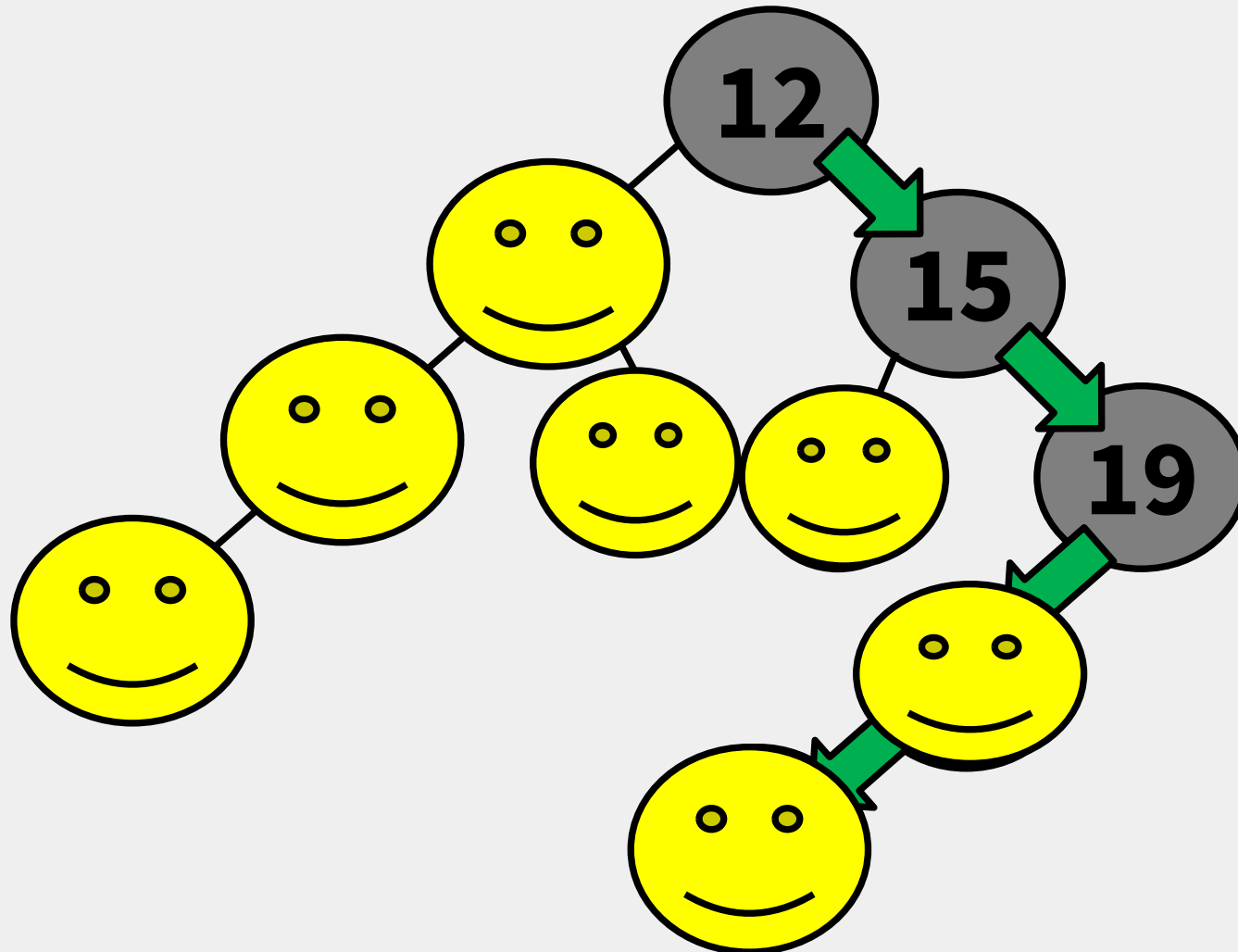




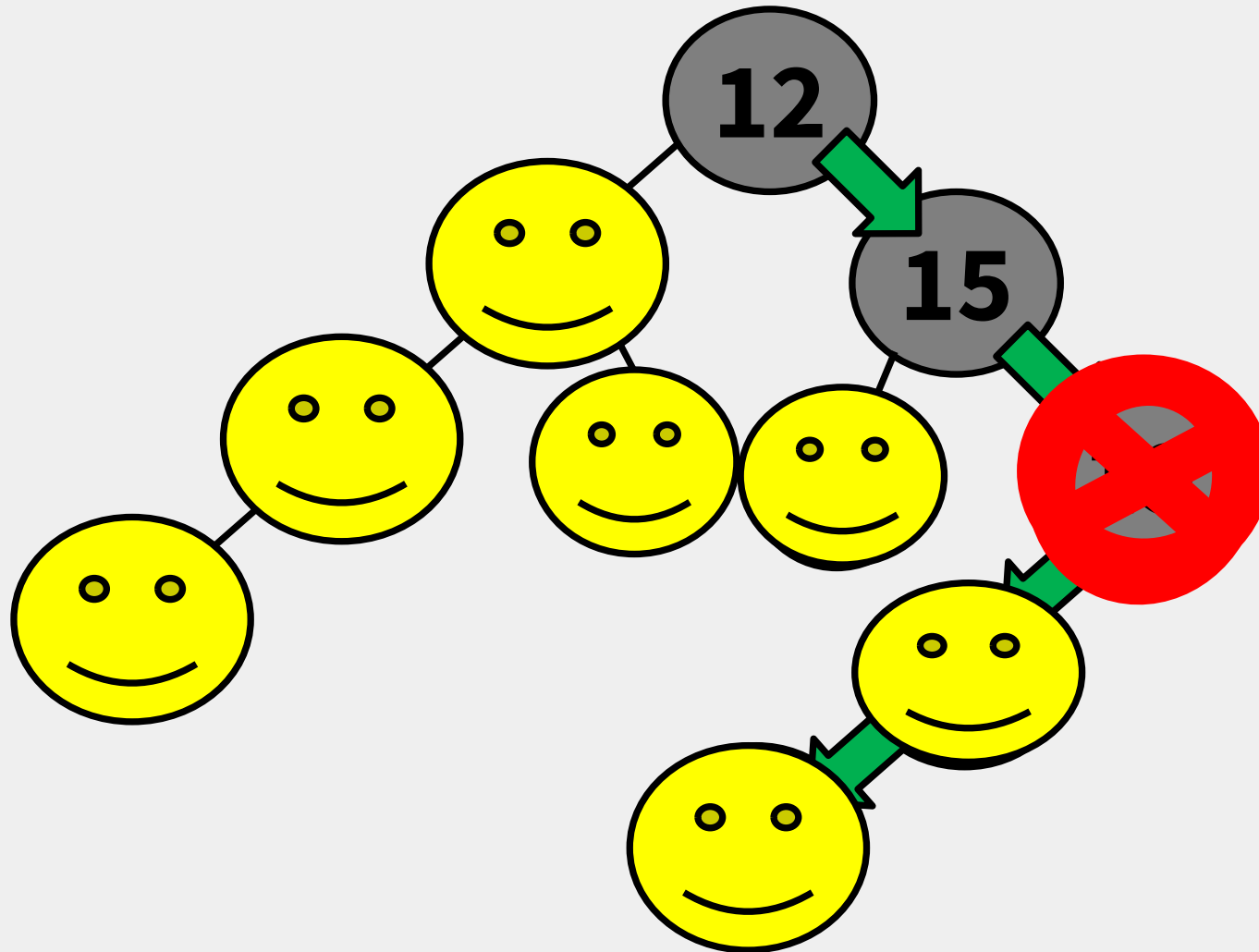
Hay que volver a verificar que este  
balanceado, pero no sobre todo el árbol,  
sino sólo sobre el camino de inserción.



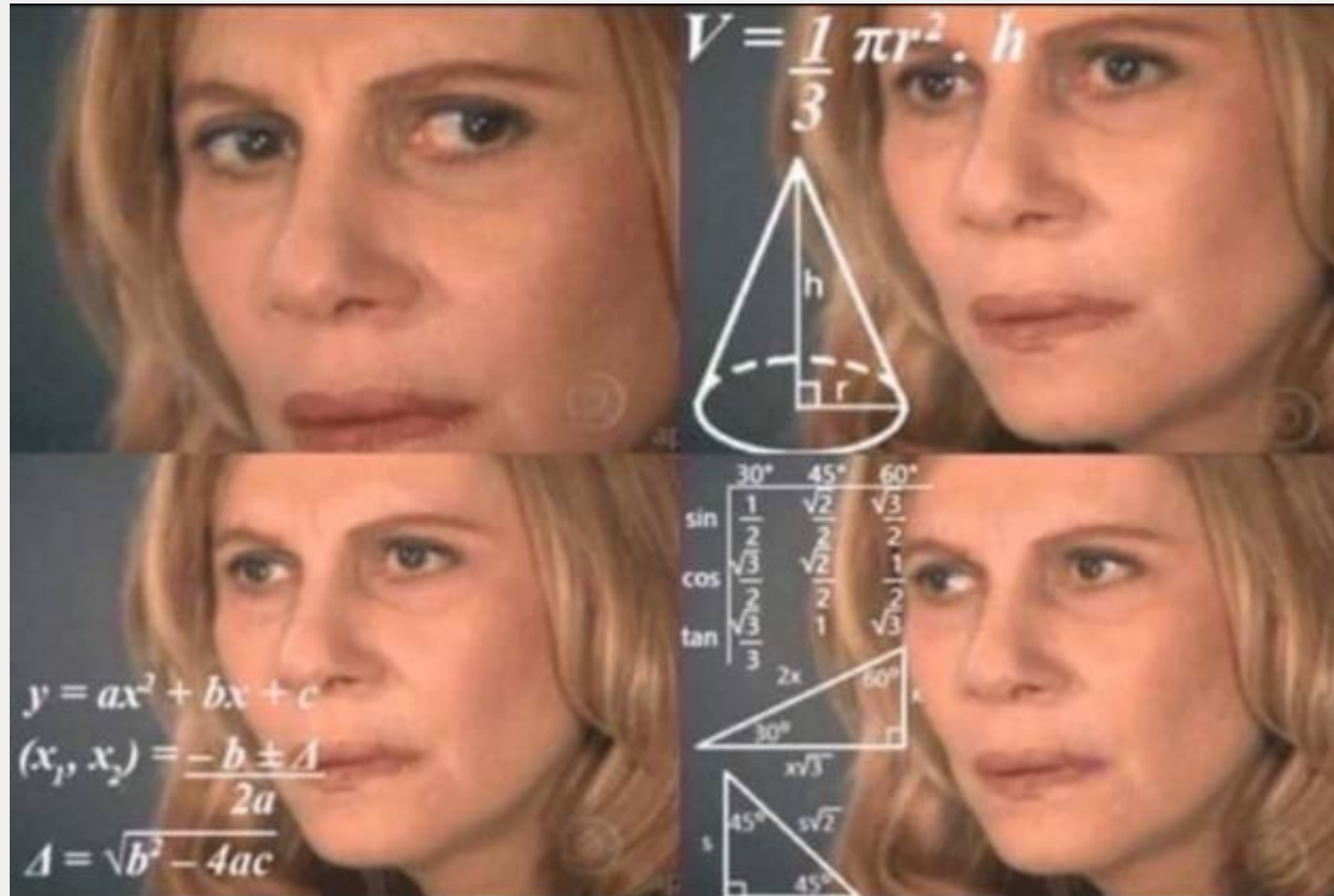
Hay que volver a verificar que este balanceado, pero no sobre todo el árbol, sino sólo sobre el camino de inserción.



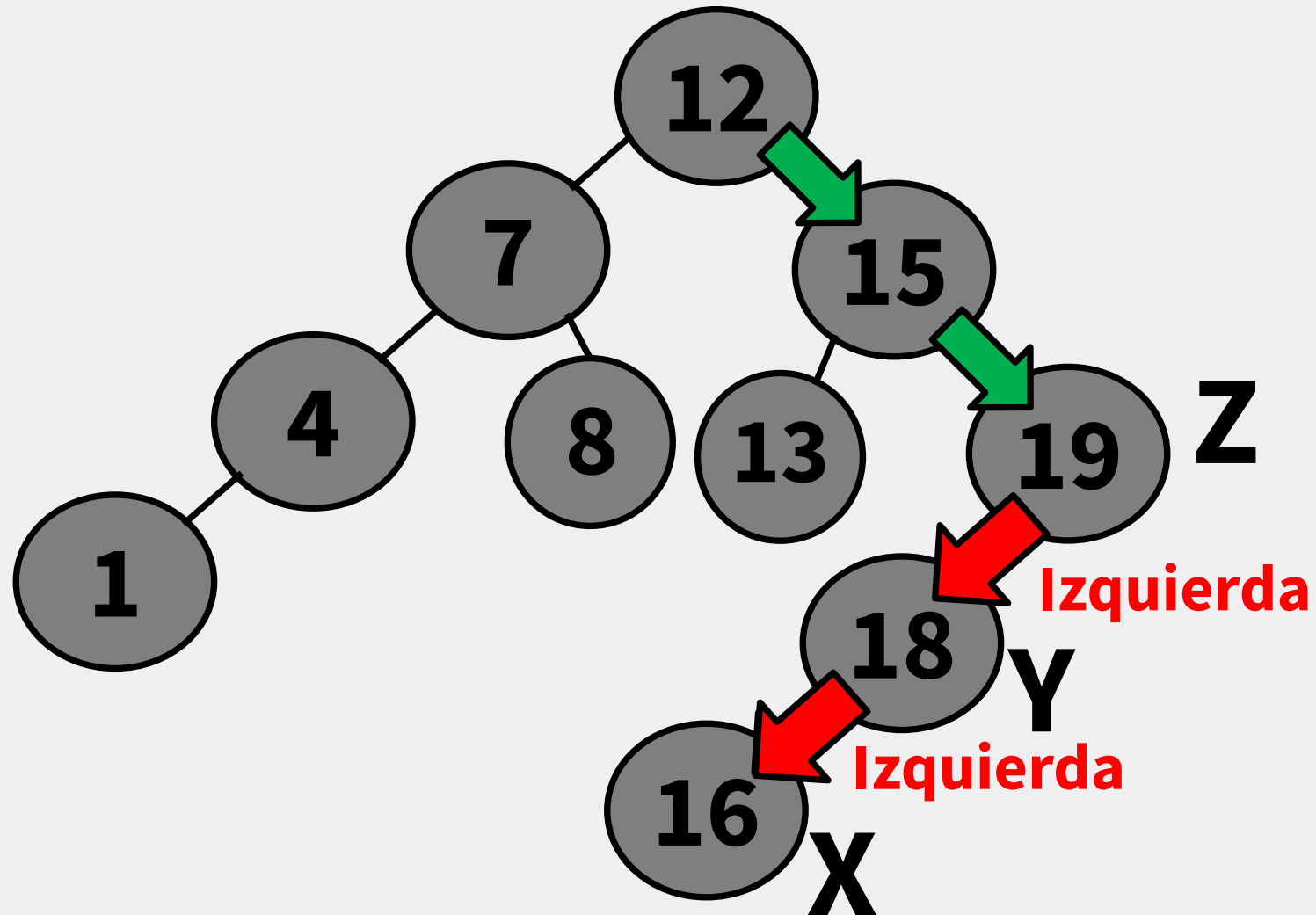
Hay que volver a verificar que este balanceado, pero no sobre todo el árbol, sino sólo sobre el camino de inserción.



¿Cómo sabemos que rotación elegir?  
¿Cómo elegimos 'z', 'y', 'x'?



Nos paramos sobre el nodo que rompió la invariante del AVL (ESE ES 'Z') y continuamos sobre el camino de inserción hacia el nodo insertado para elegir consecutivamente a 'Y' y luego a 'X'.

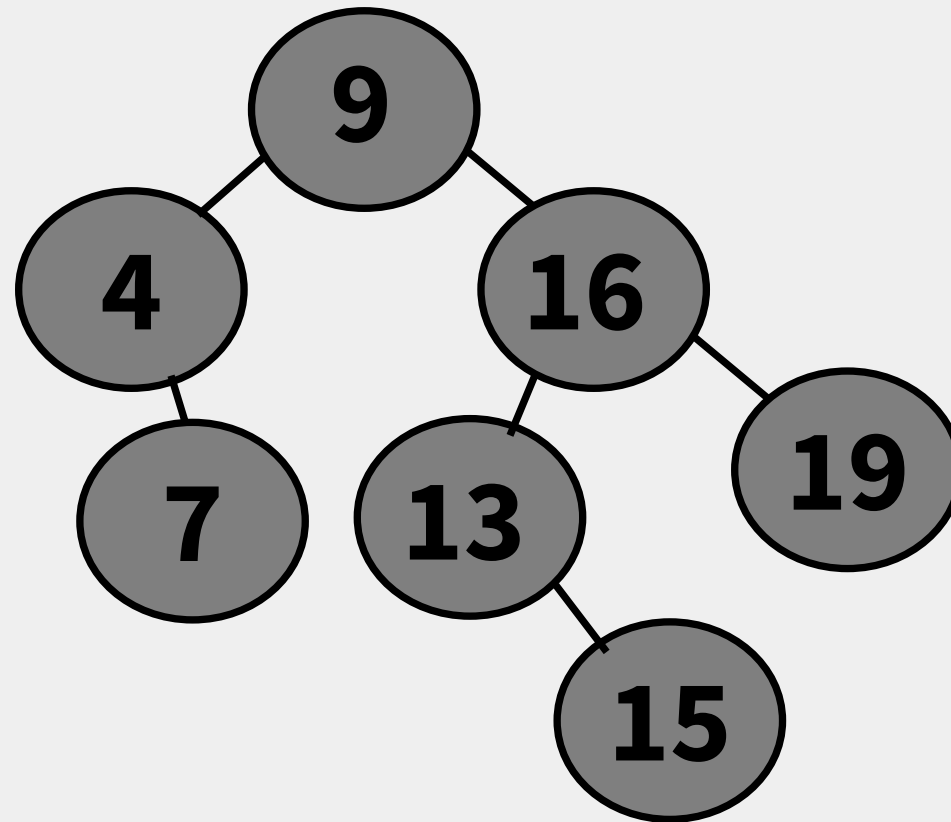


Y AHORA HAGAMOS UN SEGUIMIENTO ..



Sobre el siguiente AVL realizar,  
mostrando pasos intermedios, las  
siguientes inserciones:

10, 14, 11 (en ese orden)

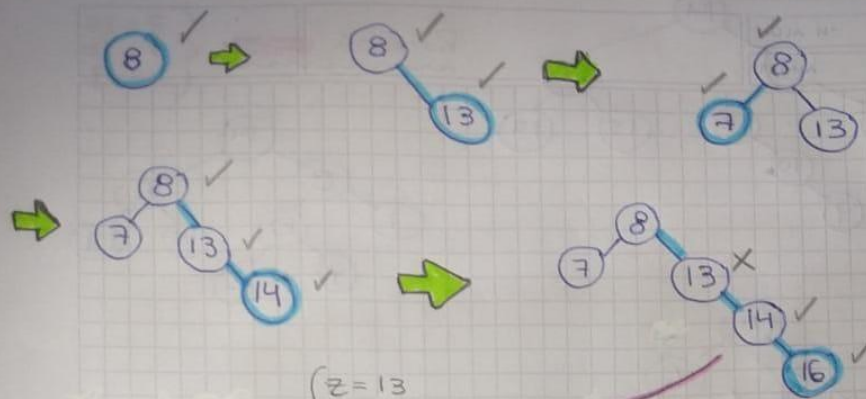




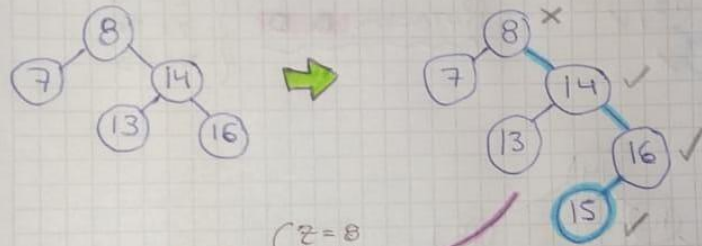
Mostrar los cambios en la estructura de un árbol AVL inicialmente vacío **(incluyendo los pasos intermedios)** al realizar las siguientes inserciones:

**8, 13, 7, 14, 16, 15, 17, 9, 10, 11.**

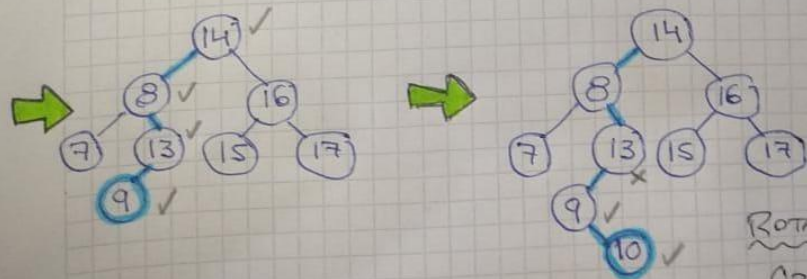
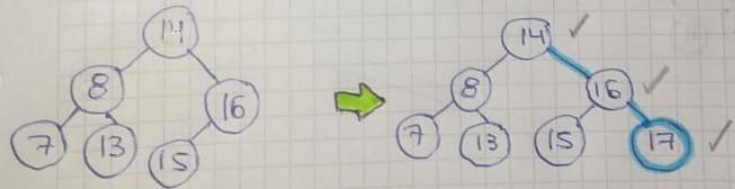




ROTACIÓN D-D:  $\begin{cases} Z=13 \\ Y=14 \\ X=16 \end{cases}$

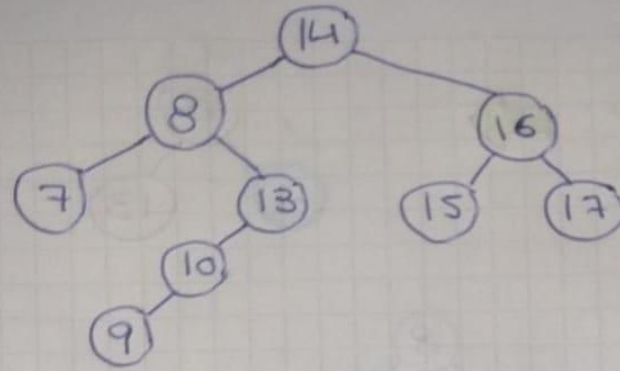


ROTACIÓN D-D:  $\begin{cases} Z=8 \\ Y=14 \\ X=16 \end{cases}$

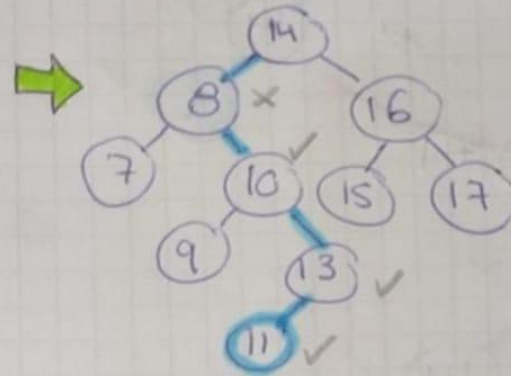
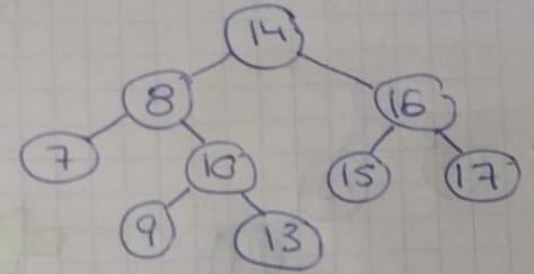


ROTACIÓN I-D

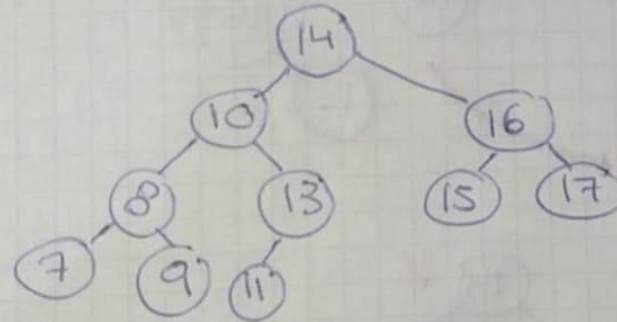
$\begin{cases} Z=13 \\ Y=9 \\ X=10 \end{cases}$



ROTACIÓN I-I:  $\begin{cases} Z=13 \\ Y=10 \\ X=9 \end{cases}$



ROTACIÓN D-D:  $\begin{cases} Z=8 \\ Y=10 \\ X=13 \end{cases}$



VISUALIZADOR online:

<https://visualgo.net/es/bst>

- IMPORTANTE: SELECCIONAR OPCIÓN AVL





# EJEMPLO DE ROTACIÓN DE RAÍZ



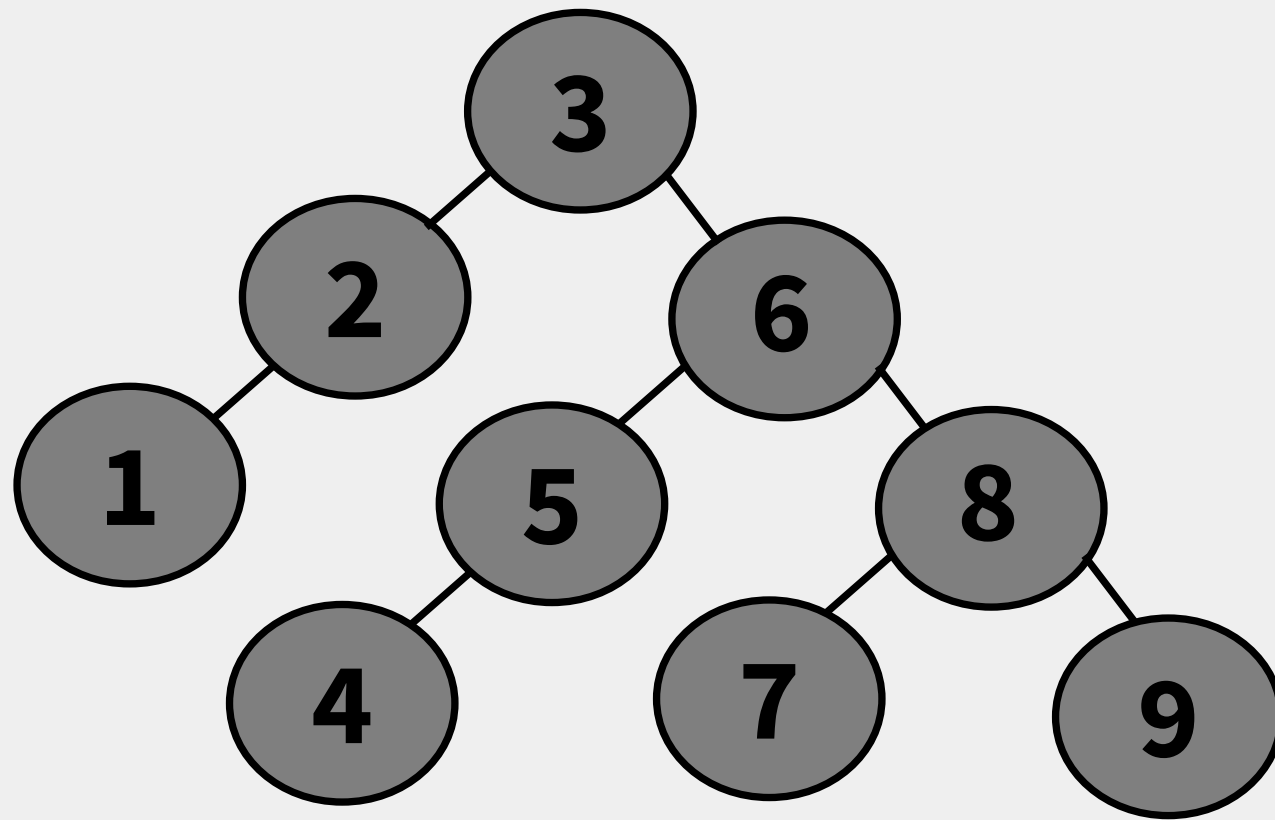
# ROTACIÓN DE RAÍZ (ejemplo)

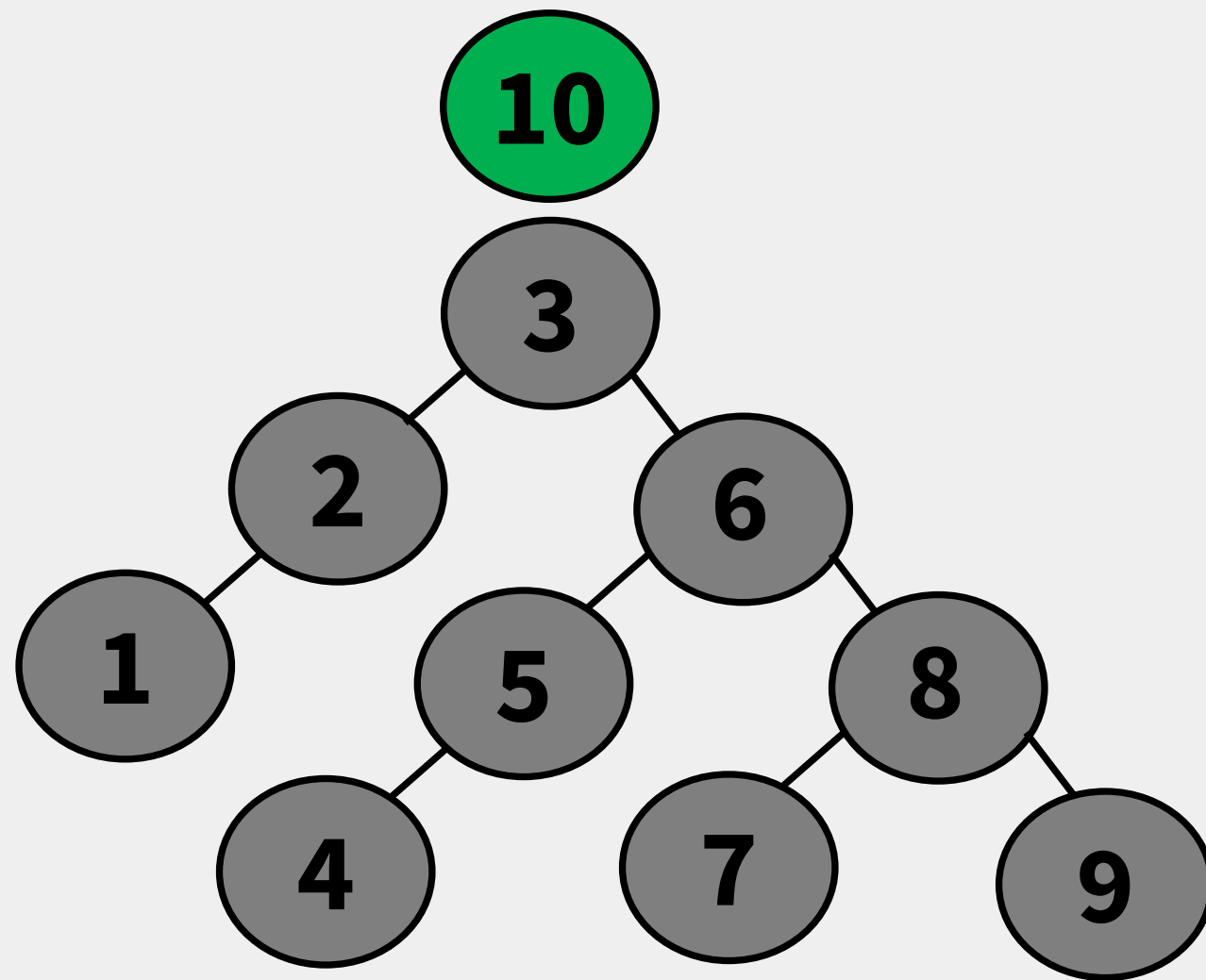
Dado el siguiente AVL, cuyo recorrido **PRE ORDER** es:

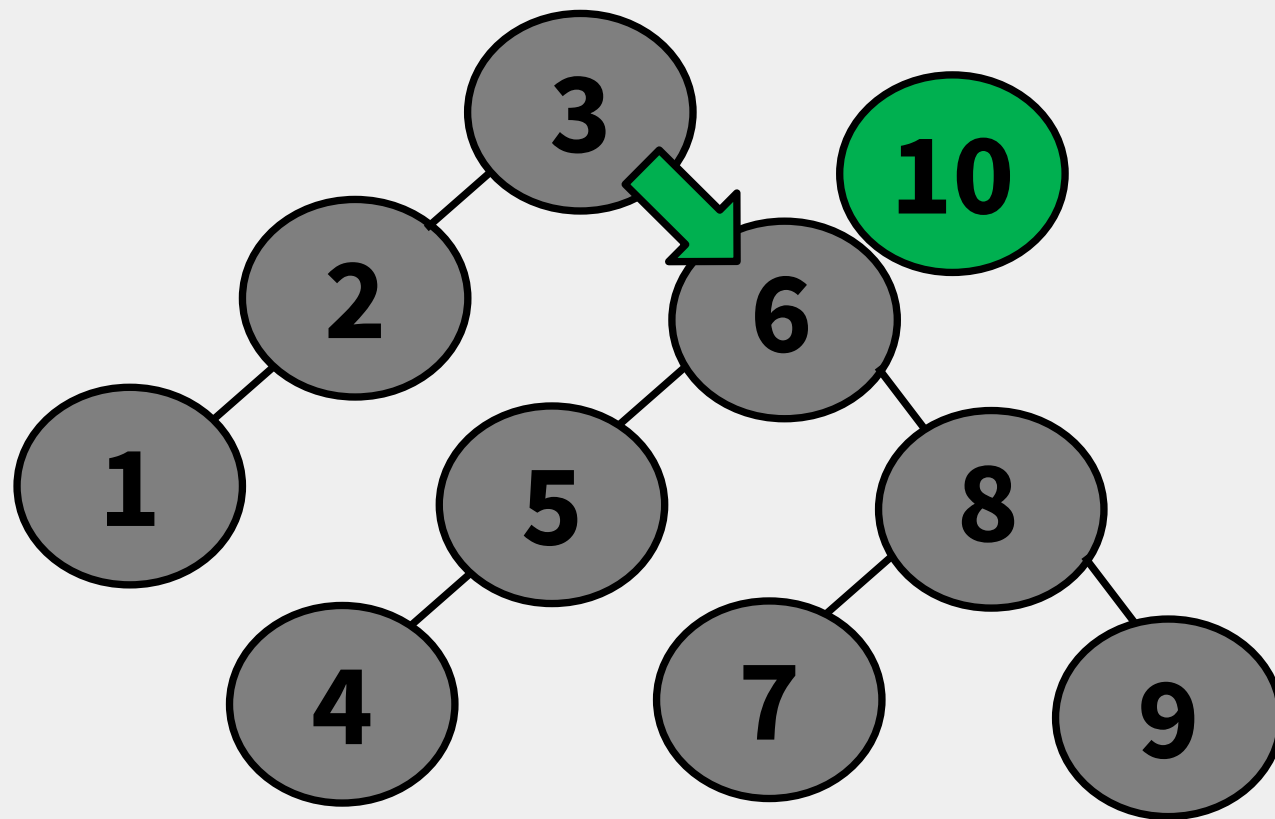
**3 → 2 → 1 → 6 → 5 → 4 → 8 → 7 → 9**

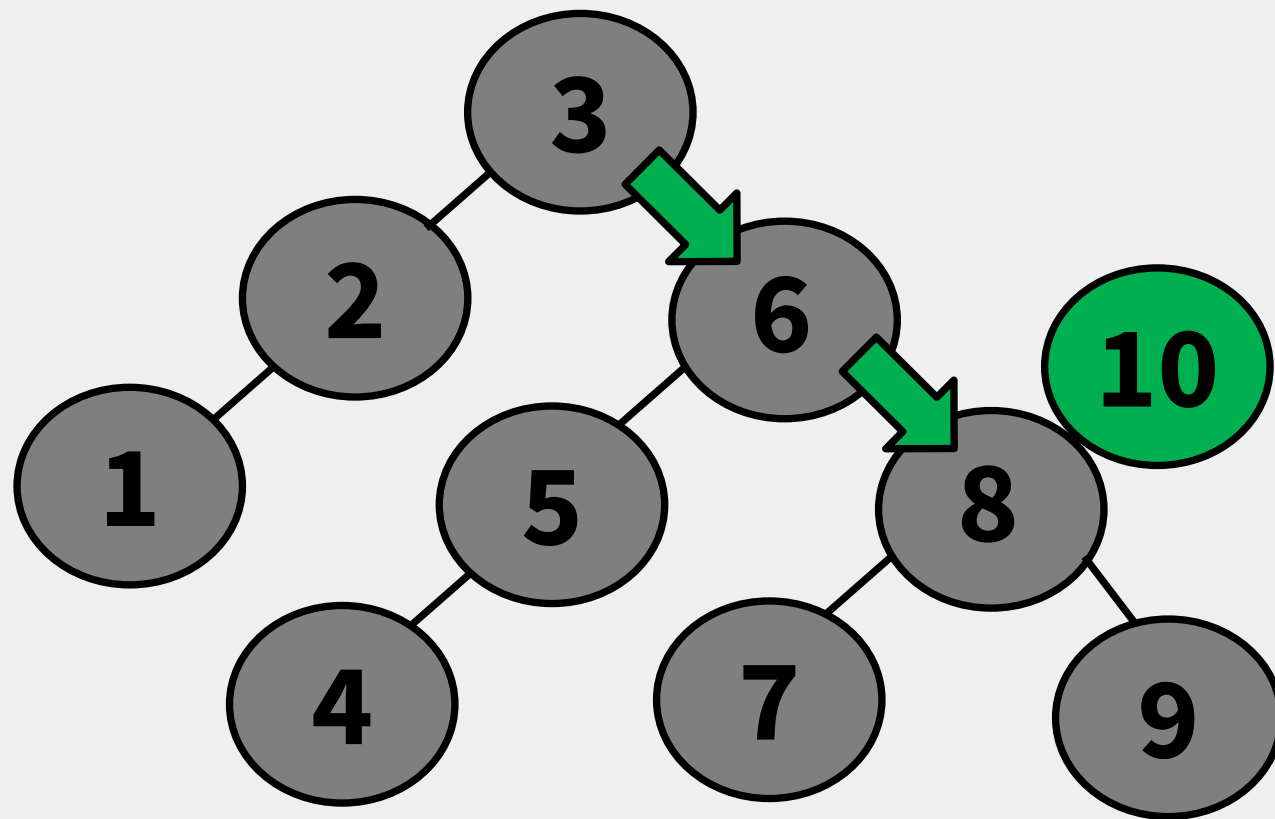
Insertar el elemento **10**.

**10**

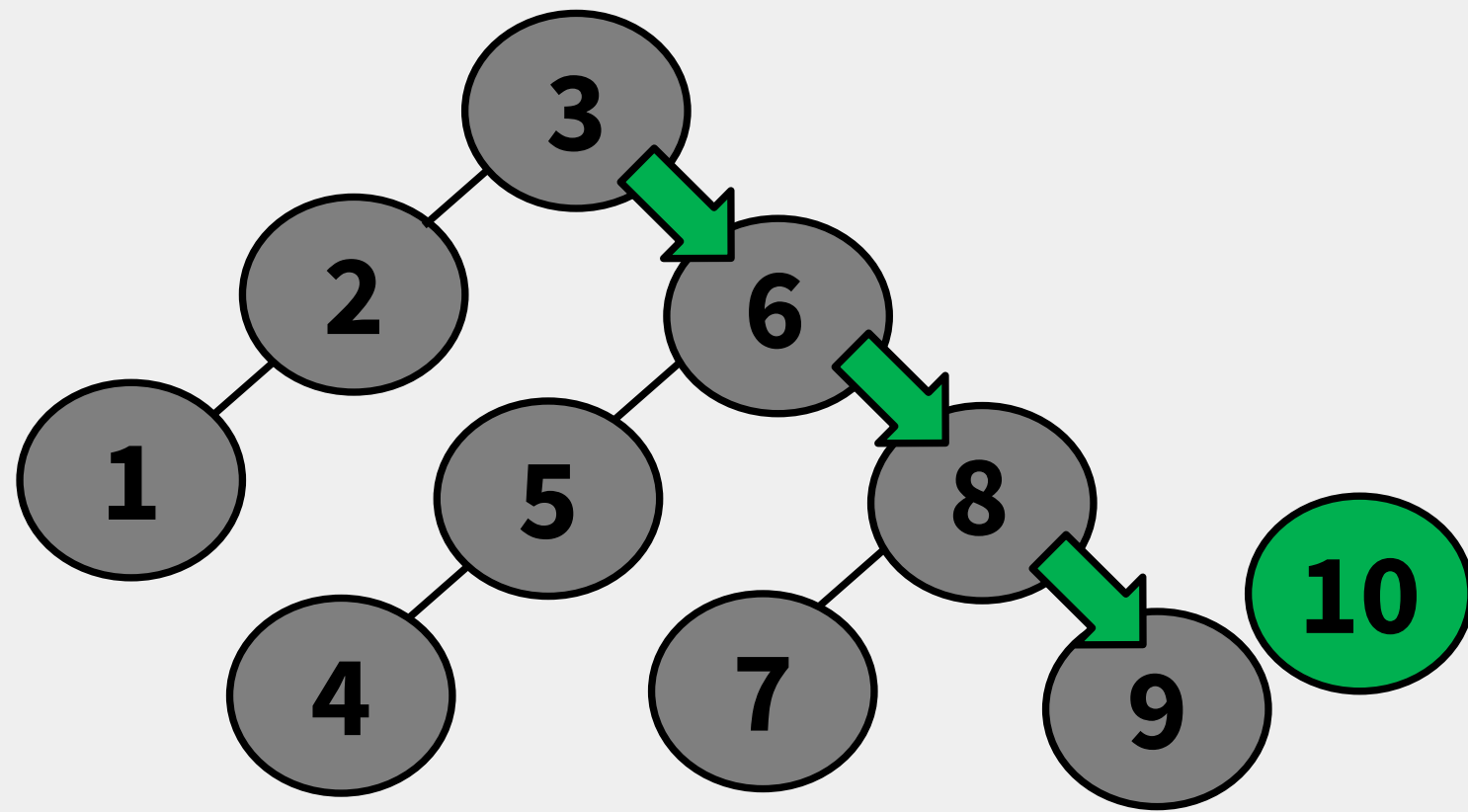




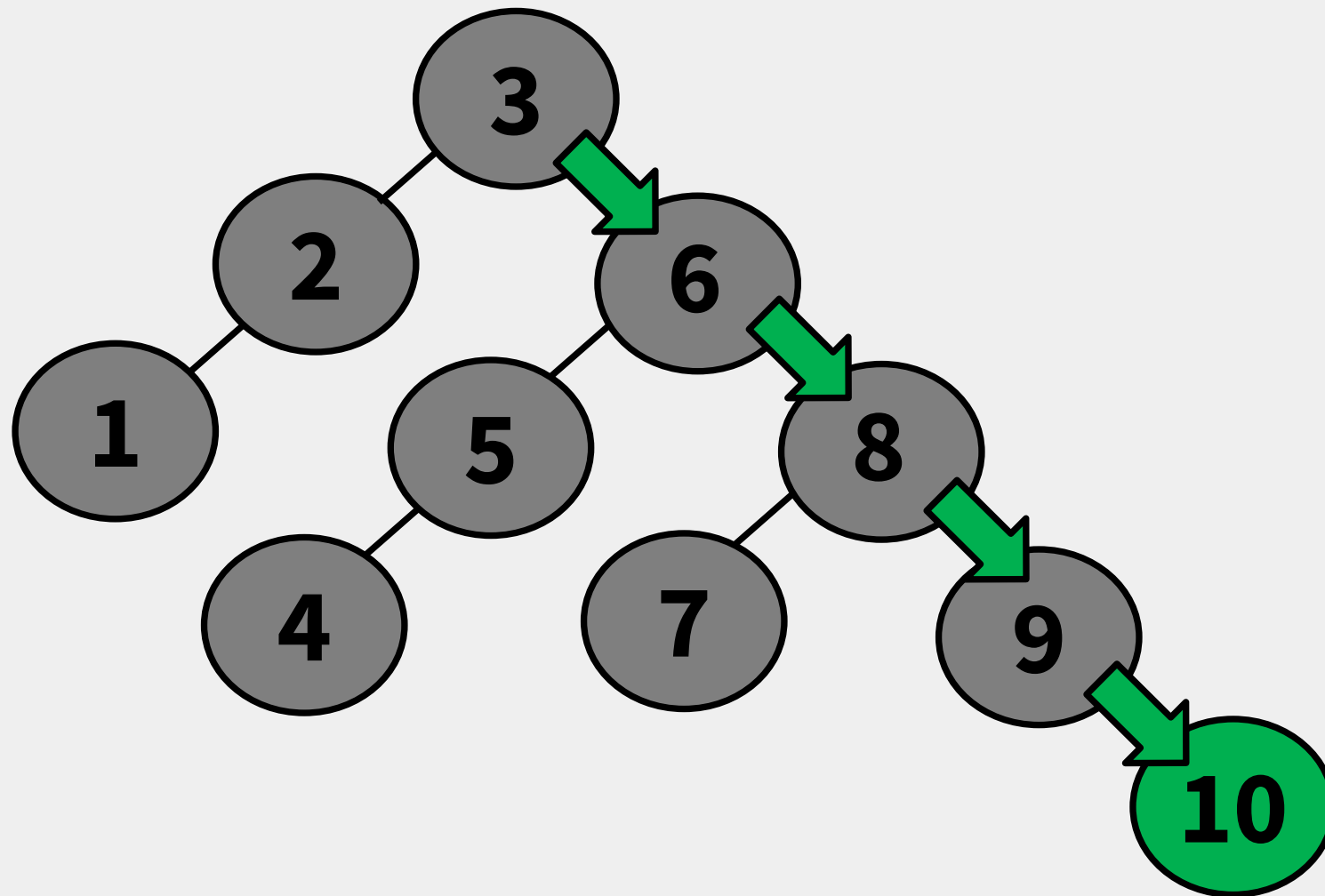




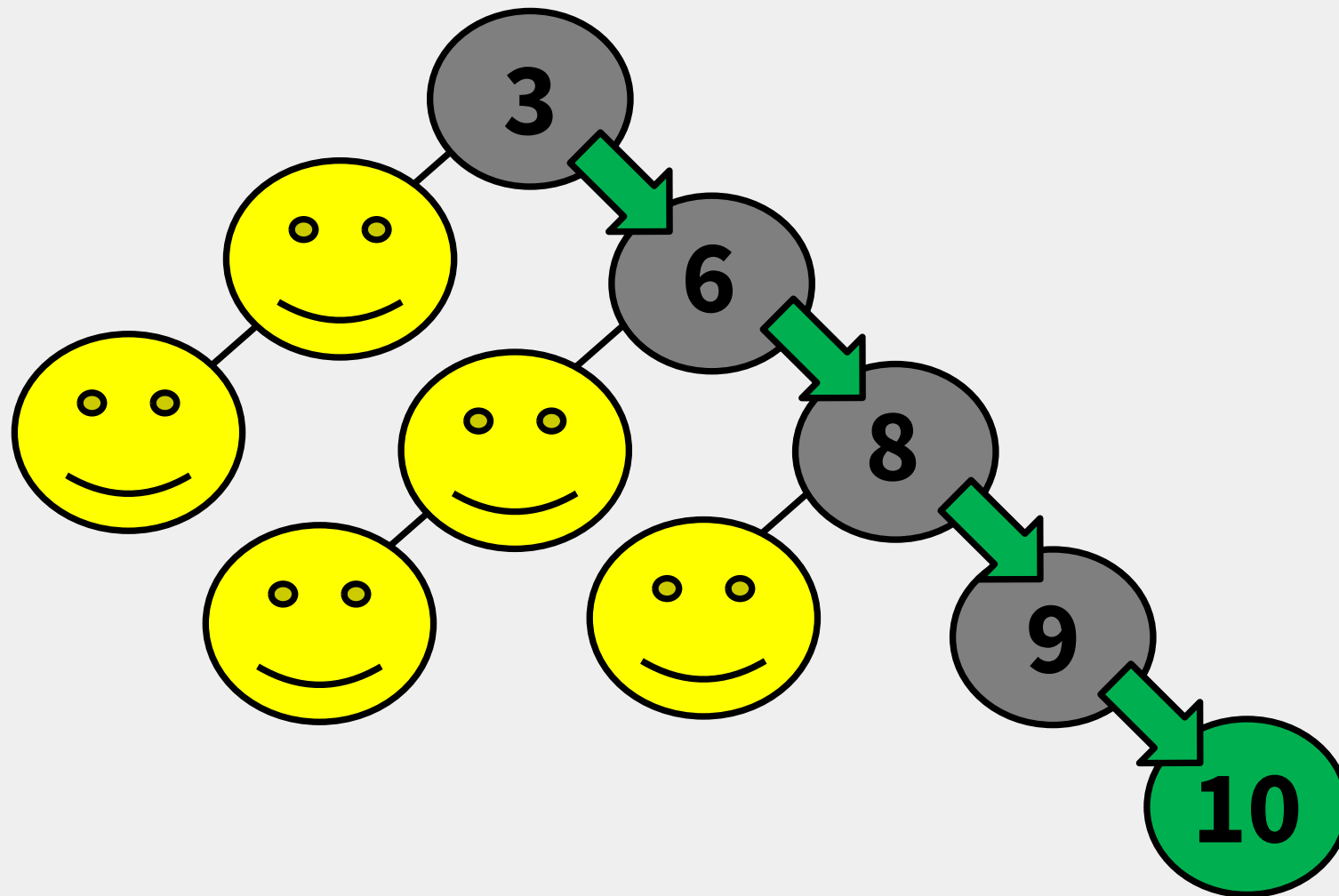




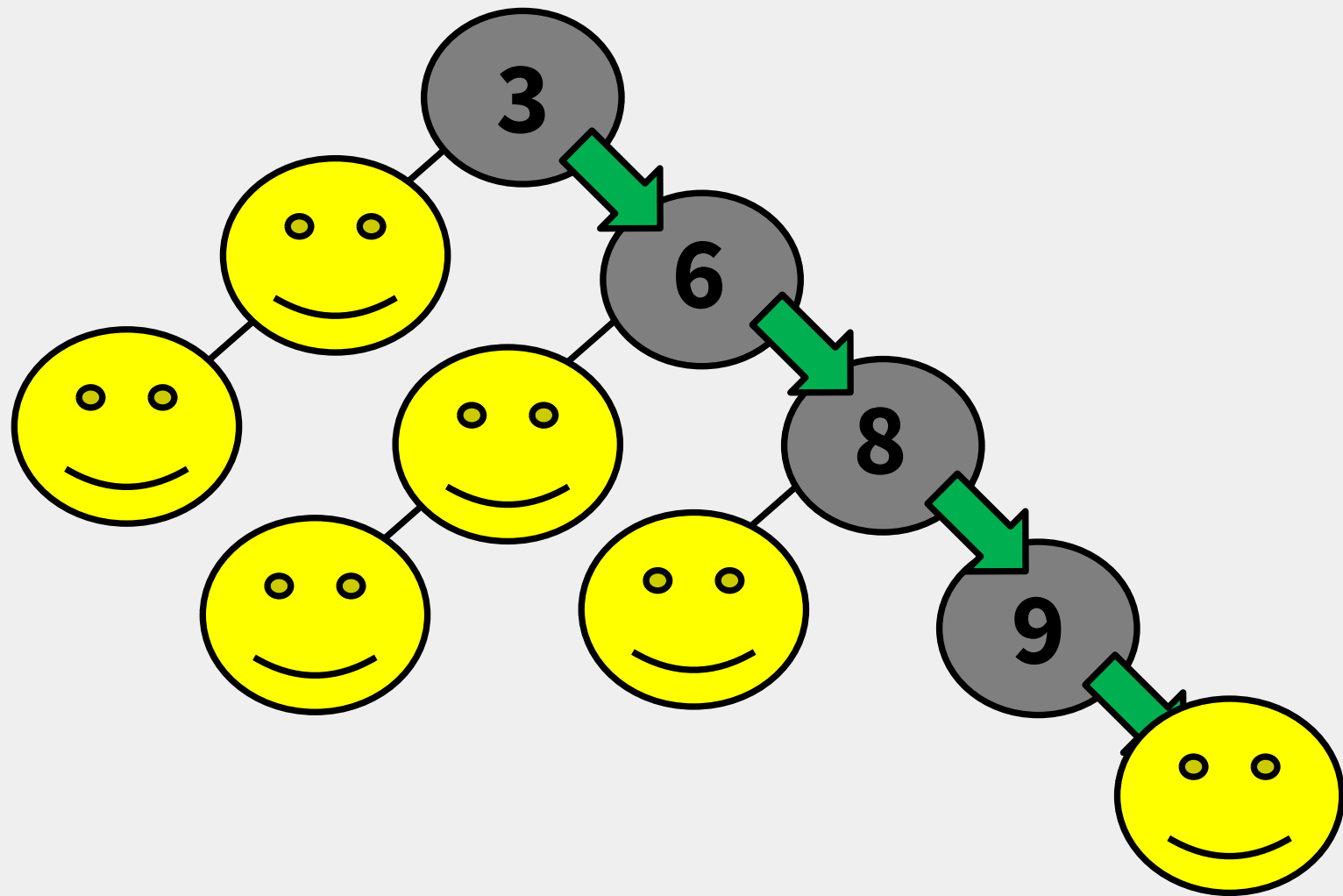
# ¿Está balanceado?



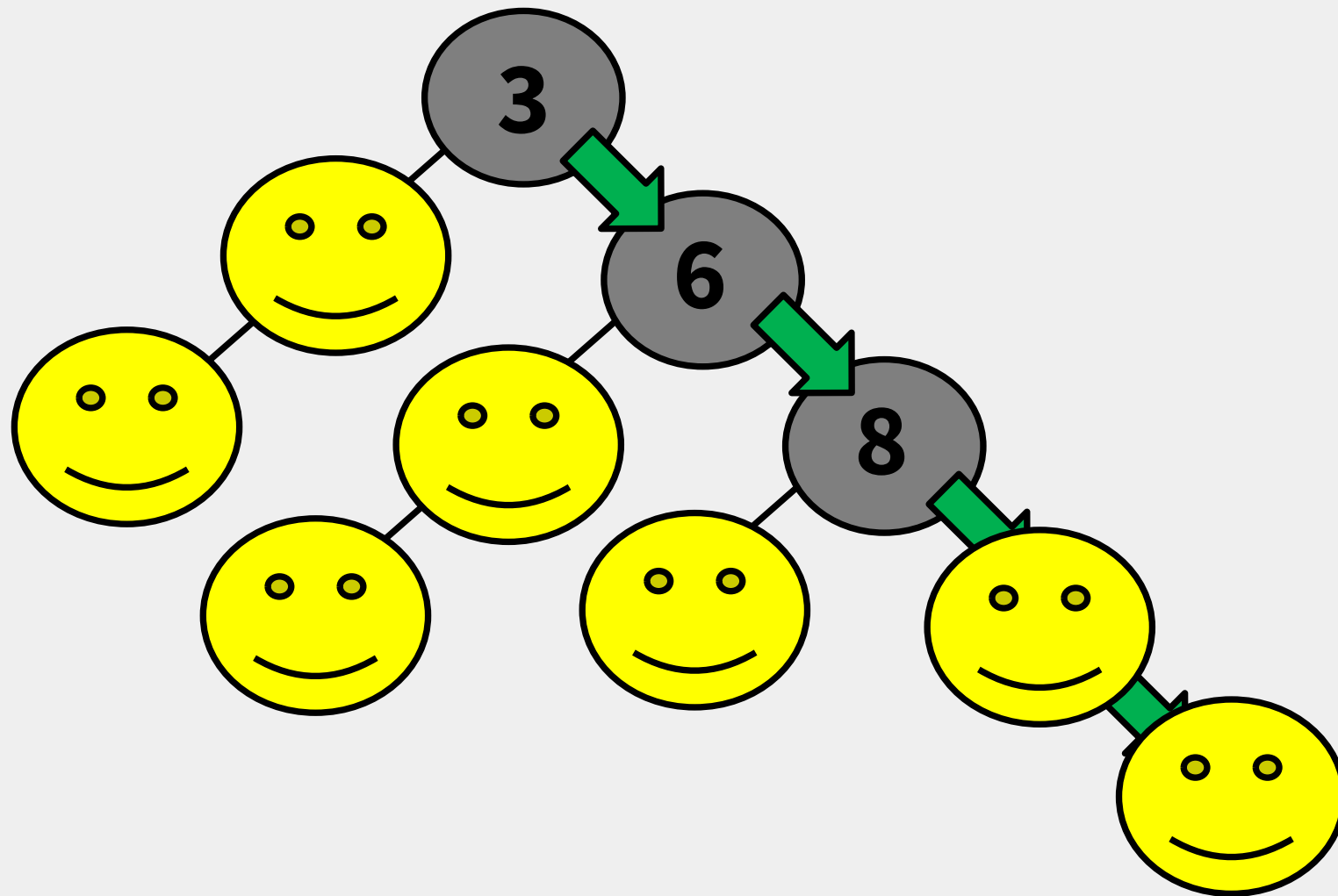
# ¿Está balanceado?



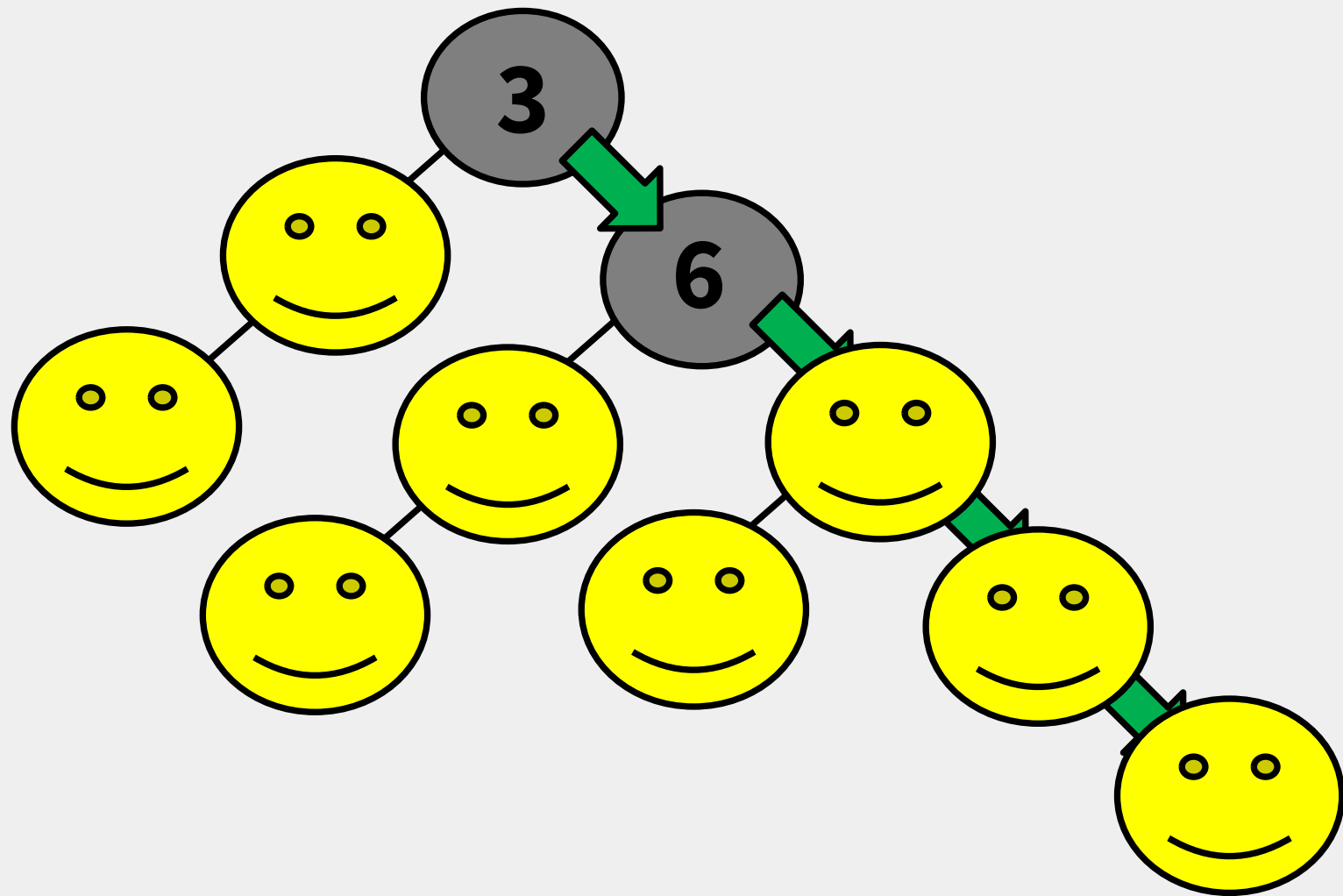
# ¿Está balanceado?



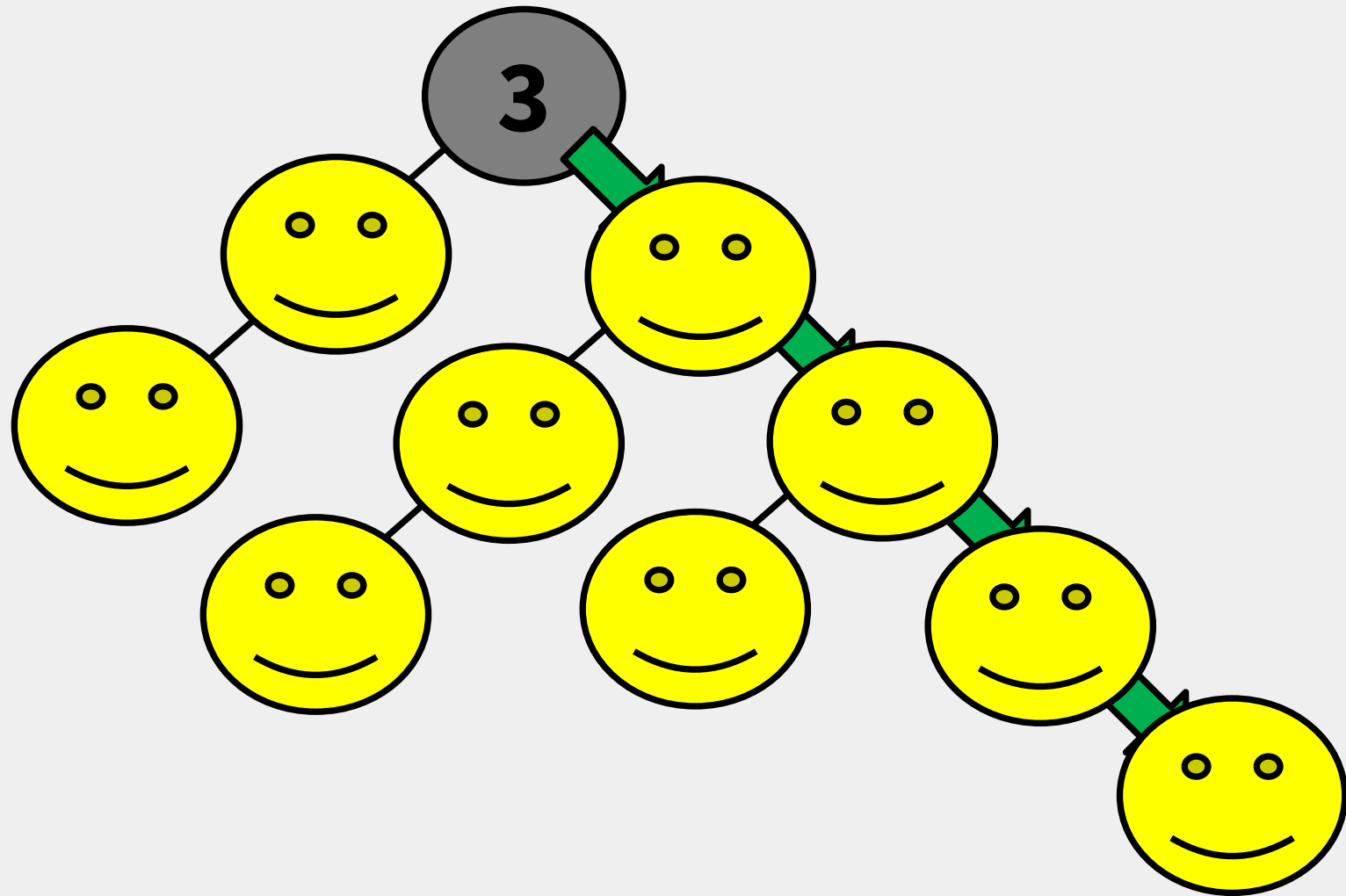
# ¿Está balanceado?



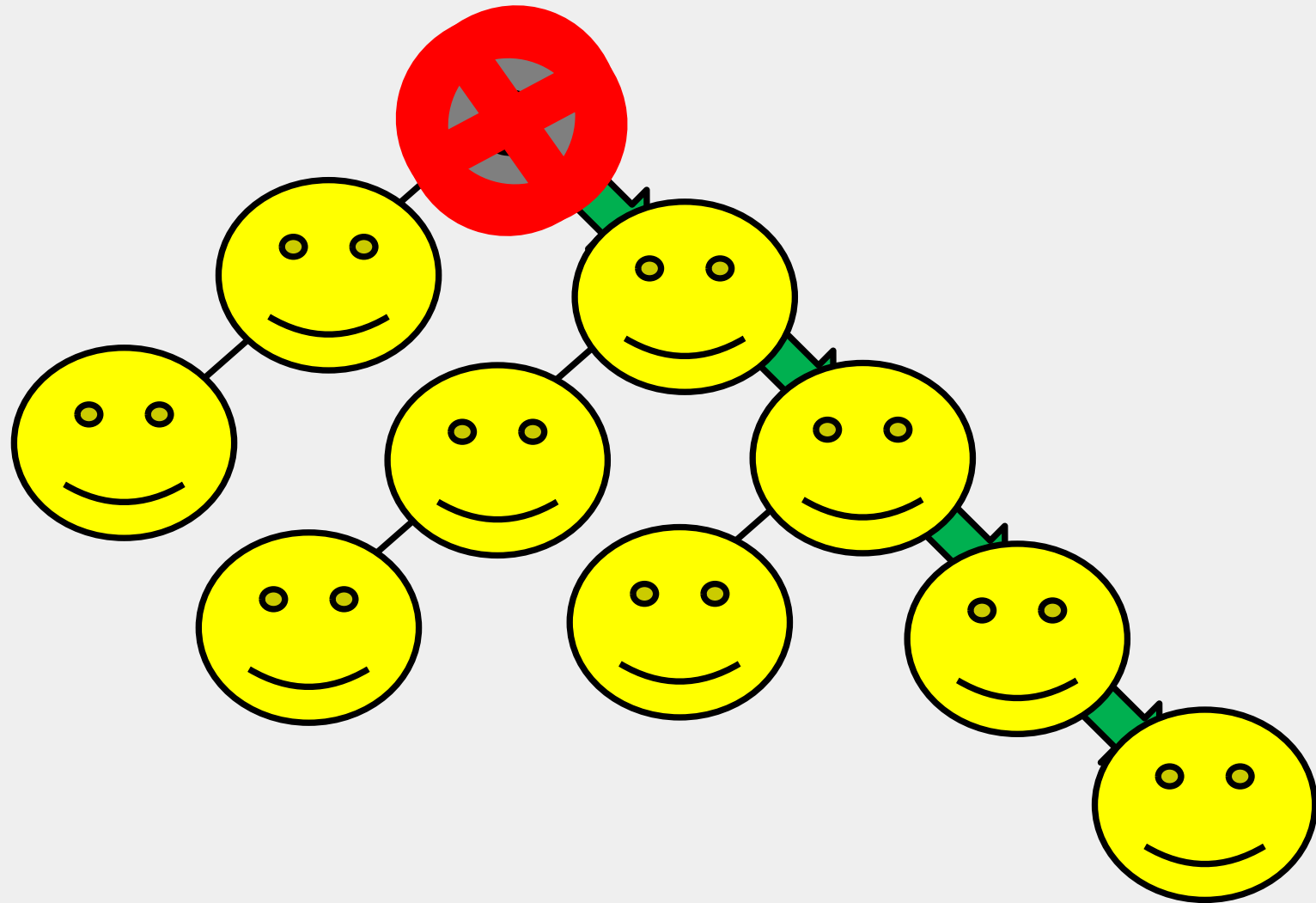
# ¿Está balanceado?



# ¿Está balanceado?

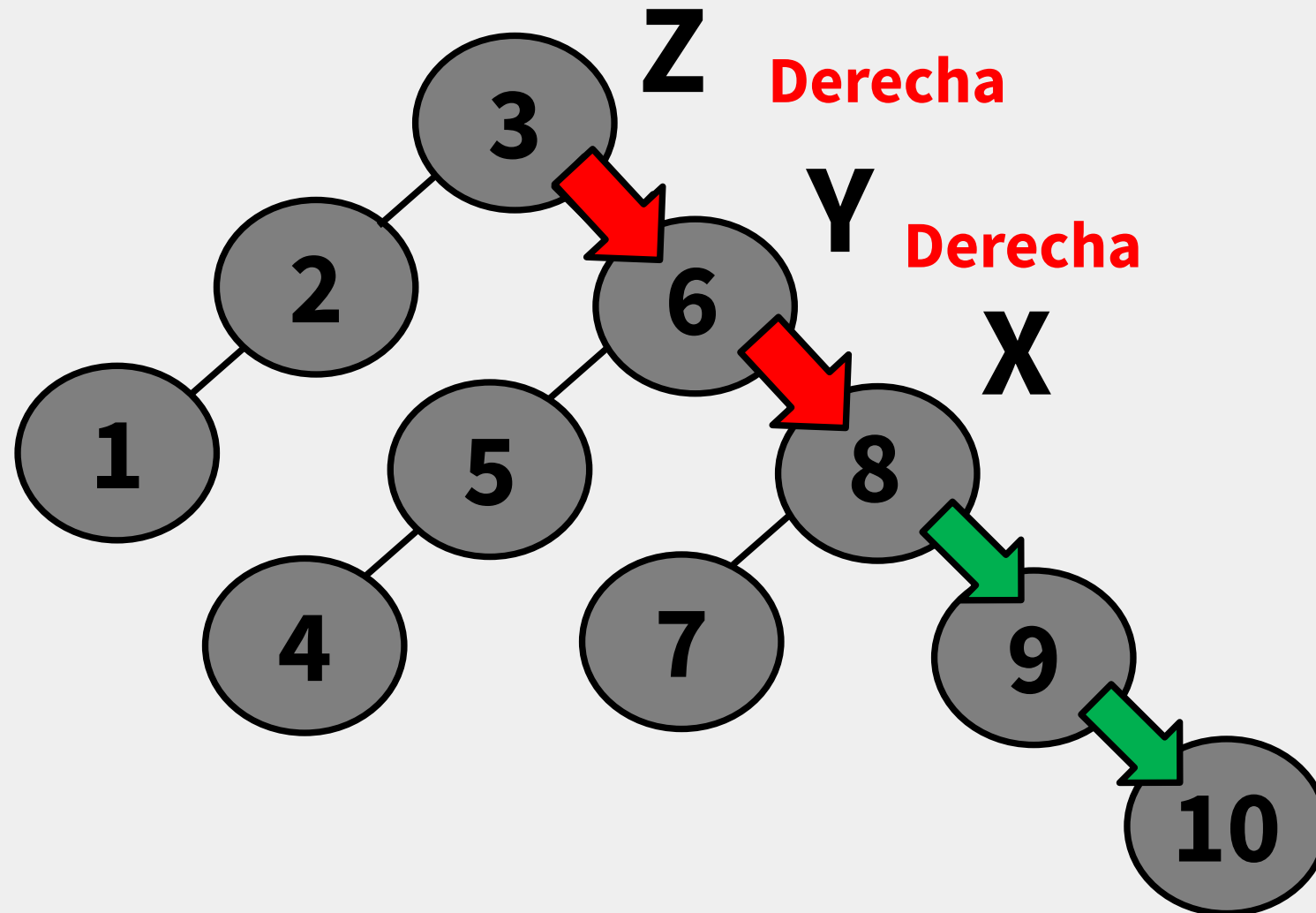


# ¿Está balanceado?

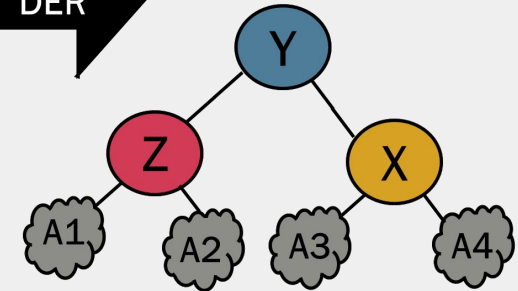
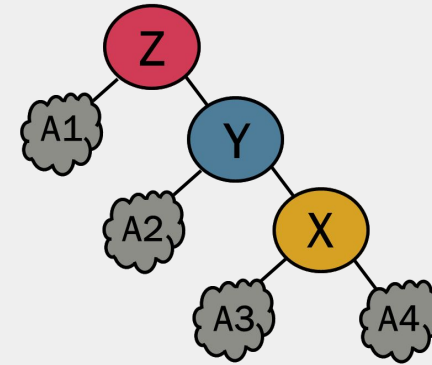
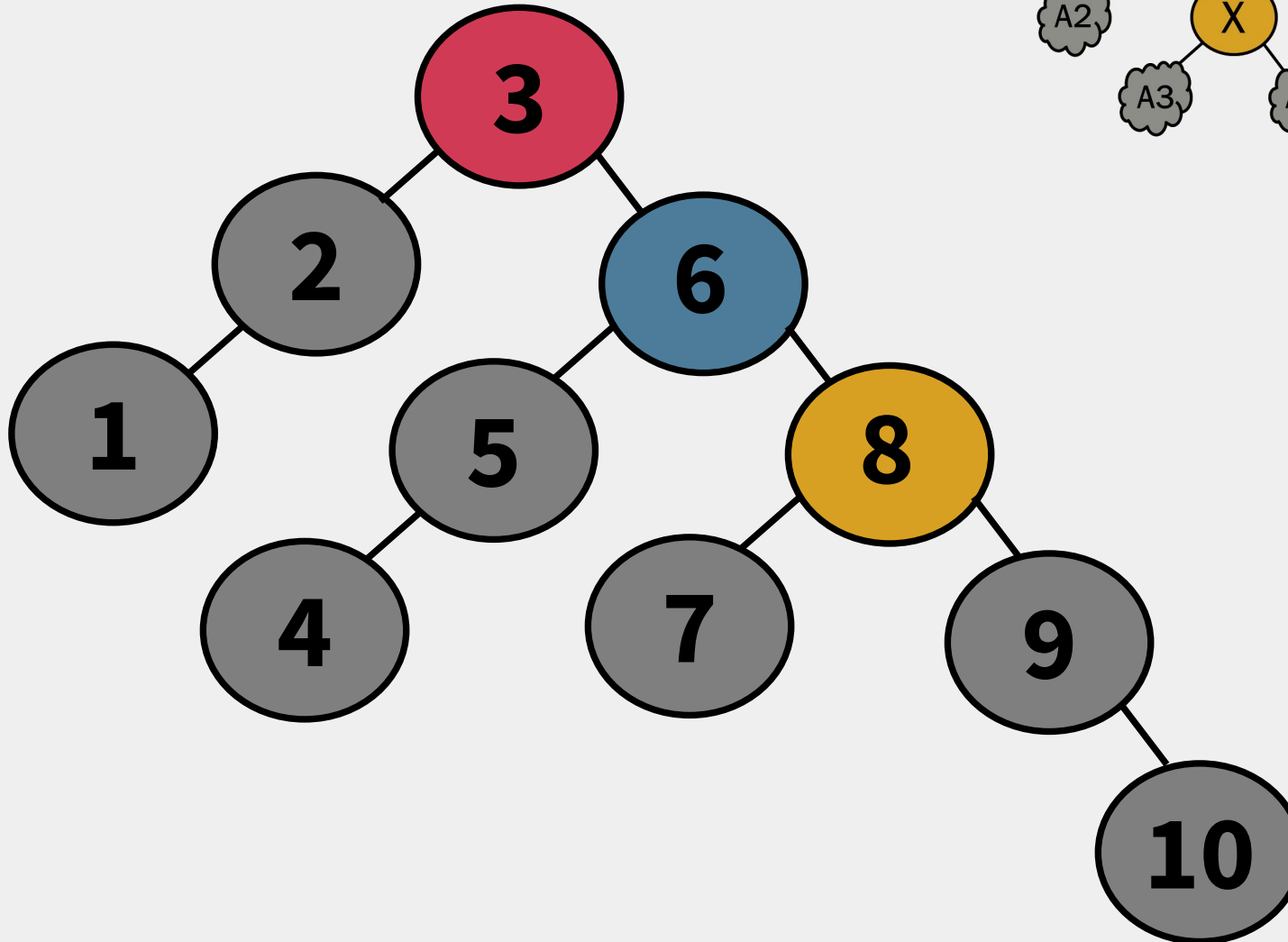




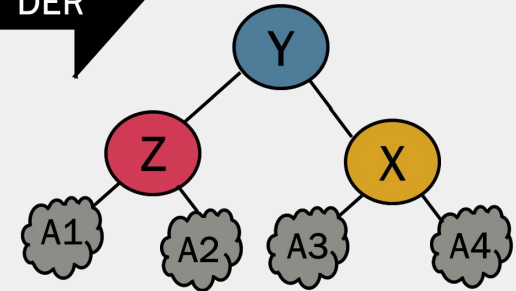
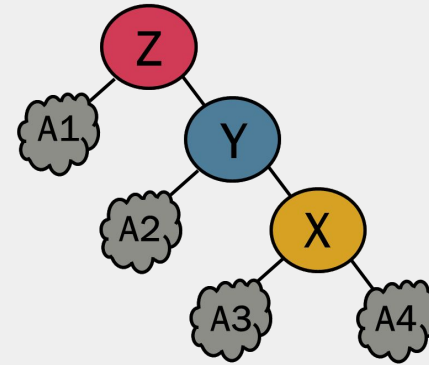
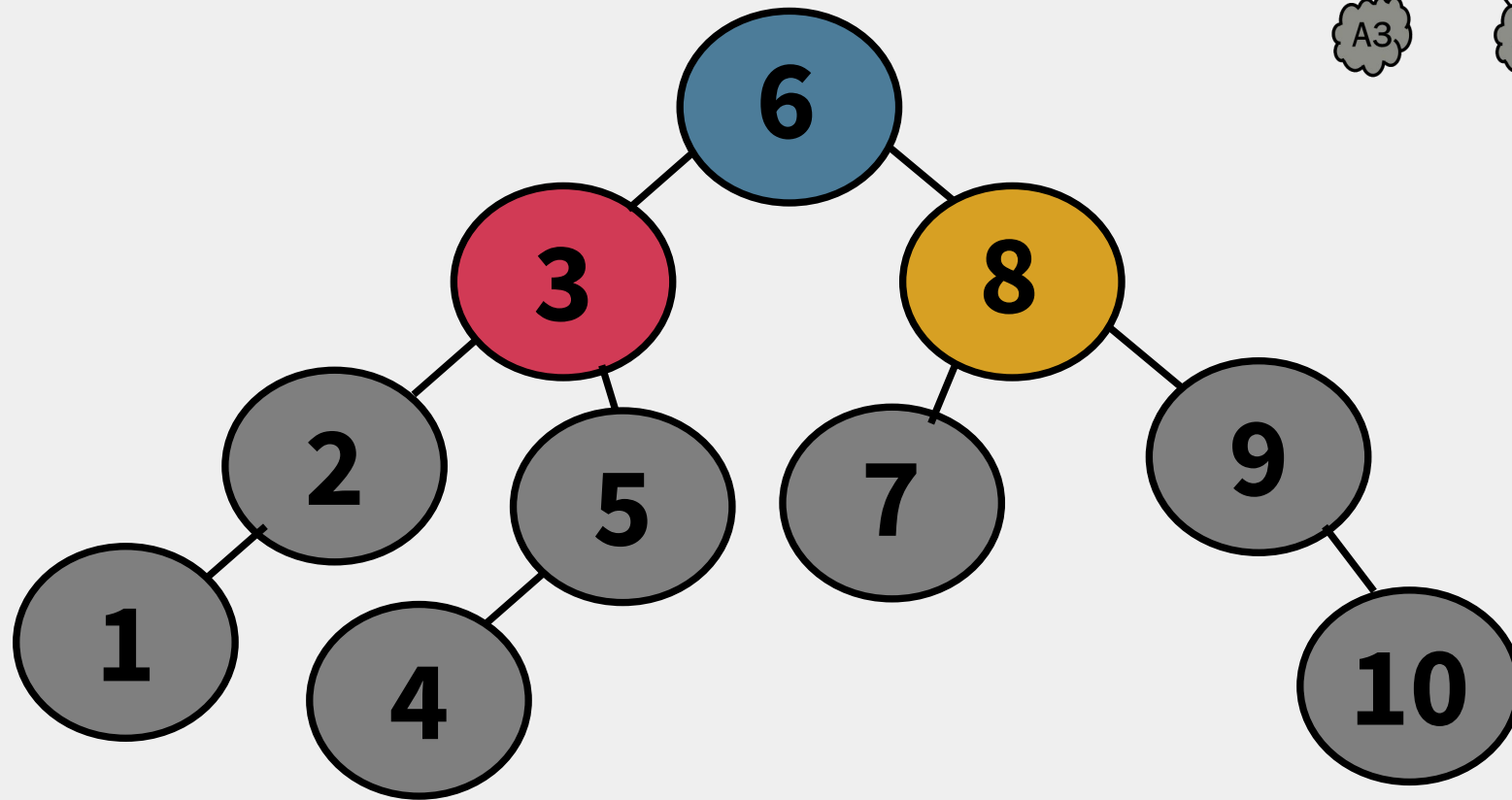
# Elegimos la rotación



# Elegimos la rotación



# Rotación



# Ejercicio Bonus

del final del 20.02.2020

Implementar una primitiva para el árbol binario

```
bool ab_prop_avl(const ab_t*)
```

que reciba dicho árbol y devuelva si el árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ . Justificar el orden de la primitiva. Indicar el tipo de recorrido utilizado. Considerar que la estructura del árbol es:

```
typedef struct ab {  
    struct ab* izq;  
    struct ab* der;  
} ab_t;
```

# Ejercicio Bonus

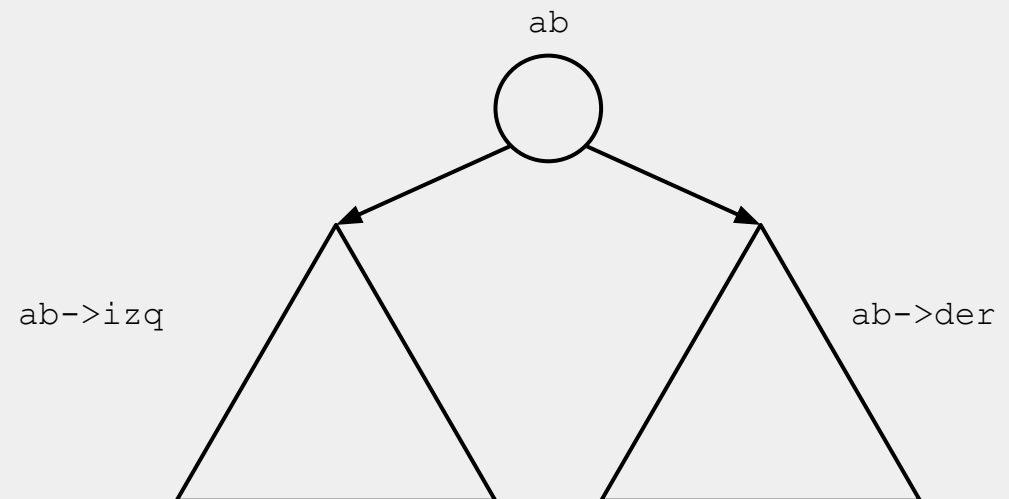
del final del 20.02.2020

Implementar una primitiva para el árbol binario

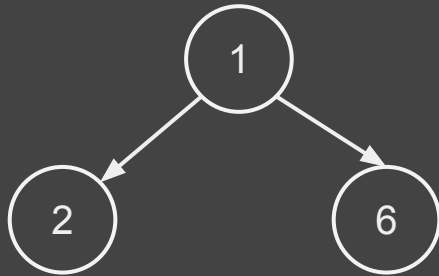
```
bool ab_prop_avl(const ab_t*)
```

que reciba dicho árbol y devuelva si el árbol cumple con la propiedad de AVL.  
La primitiva no debe ejecutar en más que  $O(n)$ . Justificar el orden de la primitiva.  
Indicar el tipo de recorrido utilizado. Considerar que la estructura del árbol es:

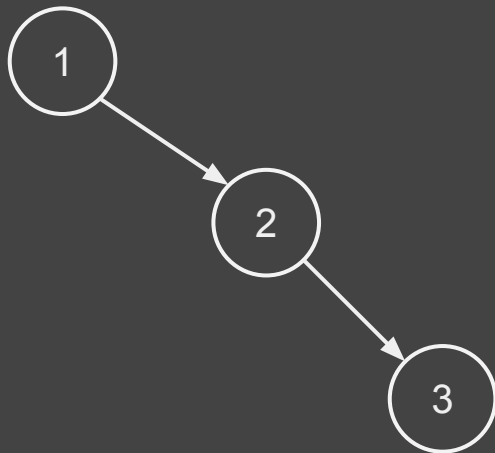
```
typedef struct ab {  
    struct ab* izq;  
    struct ab* der;  
} ab_t;
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .



→ **true**



→ **false**

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

- Propiedad AVL  $\longrightarrow$  Nos garantiza que para cada nodo, la diferencia entre la altura de sus hijos es igual o menor a 1.

$$|h(a \rightarrow \text{izq}) - h(a \rightarrow \text{der})| \leq 1 \rightarrow \text{ESTA BALANCEADO}$$

¿Cómo calculabamos la altura?

$$h(a) = \max(h(a \rightarrow \text{izq}), h(a \rightarrow \text{der})) + 1$$

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

- Caso base? → Si no hay árbol, está balanceado

```
if (!ab) {  
    return true;  
}
```

- Caso recursivo? →
  - Llamo recursivamente para el sub-árbol izquierdo
  - Llamo recursivamente para el sub-árbol derecho
  - Calculo la diferencia de altura para cada nodo

```
if (!ab_prop_avl(ab->izq)) return false;  
if (!ab_prop_avl(ab->der)) return false;  
return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1;
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    if (!ab_prop_avl(ab->izq)) return false;  
    if (!ab_prop_avl(ab->der)) return false;  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1;  
}
```

**Problem solved wohoo!**

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    if (!ab_prop_avl(ab->izq)) return false;  
    if (!ab_prop_avl(ab->der)) return false;  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1;  
}
```

**Problem solved wohoo!**



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    if (!ab_prop_avl(ab->izq)) return false;  
    if (!ab_prop_avl(ab->der)) return false;  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1;  
}
```

## Problema?

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    if (!ab_prop_avl(ab->izq)) return false;  
    if (!ab_prop_avl(ab->der)) return false;  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1;  
}
```

## Problema?

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab){  
    if (!ab_prop_avl(ab->izq)) return false; T(n/2)  
    if (!ab_prop_avl(ab->der)) return false; T(n/2)  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1; O(n)  
}
```

## Complejidad

$$T(n) = 2 \cdot T(n/2) + O(n)$$

Teorema Maestro:

$$\left. \begin{array}{l} A = 2 \\ B = 2 \\ C = 1 \end{array} \right\} \log_B(A) = C \longrightarrow \log_2(2) = 1 = C$$

~~$$T(n) = O(n \log n)$$~~

**No cumple con la consigna :(**

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab){  
    if (!ab_prop_avl(ab->izq)) return false;  $T(n/2)$   
    if (!ab_prop_avl(ab->der)) return false;  $T(n/2)$   
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1;  $O(n)$   
}
```

## Solución?

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

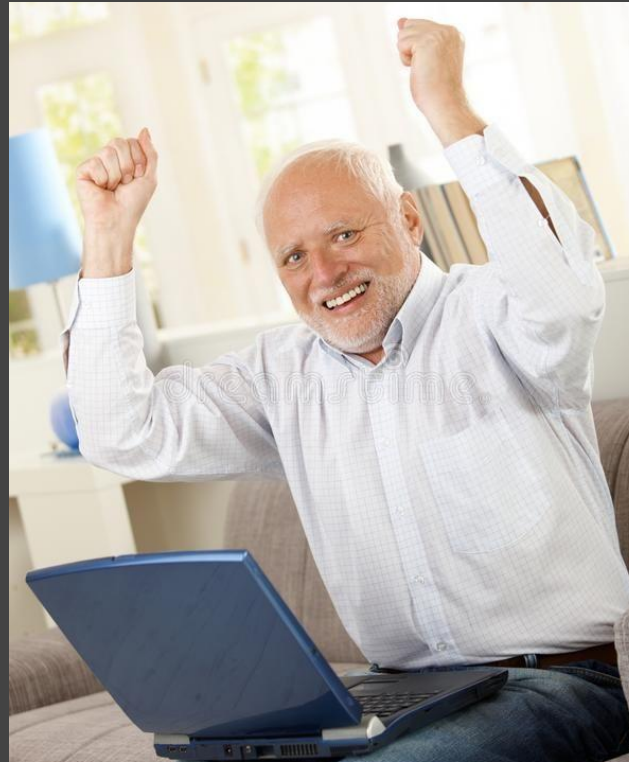
```
bool ab_prop_avl(const ab_t* ab){  
    if (!ab_prop_avl(ab->izq)) return false; T(n/2)  
    if (!ab_prop_avl(ab->der)) return false; T(n/2)  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1; O(n)  
}
```

**Solución? Punteros!**

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab){  
    if (!ab_prop_avl(ab->izq)) return false; T(n/2)  
    if (!ab_prop_avl(ab->der)) return false; T(n/2)  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1; O(n)  
}
```

Solución? **Punteros!**





Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab){  
    if (!ab_prop_avl(ab->izq)) return false; T(n/2)  
    if (!ab_prop_avl(ab->der)) return false; T(n/2)  
    return diff(ab_altura(ab->izq), ab_altura(ab->der)) <= 1; O(n)  
}
```

## Solución: Punteros!

```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}  
  
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura);
```

# Tenemos que ir calculando la altura en la misma función que utilizamos para saber si está balanceado y devolverla

Cómo es que devolvemos dos cosas? —————> uso de punteros.

```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}  
  
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura);
```

Hacemos que devuelva la altura por parámetro, y el valor de retorno indica si está balanceado

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

- Caso base?  $\longrightarrow$  Si no hay árbol, la altura de un árbol vacío es cero. Y está balanceado.

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

- Caso base?  $\longrightarrow$  Si no hay árbol, la altura de un árbol vacío es cero. Y está balanceado.

```
if (!ab) {  
    *altura = 0;  
    return true;  
}
```

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

- Caso base? → Si no hay árbol, la altura de un árbol vacío es cero. Y está balanceado.

```
if (!ab) {  
    *altura = 0;  
    return true;  
}
```

- Caso recursivo? →
  - Chequeo la condición en cada sub-árbol izquierdo
  - Chequeo la condición en cada sub-árbol derecho
  - Condición? La diferencia de altura entre sus hijos no puede ser mayor a 1
  - La altura de cada sub-árbol se calcula como la altura máxima entre los hijos de la derecha y los hijos de la izquierda + 1. Estos chequeos deben hacerse en  $O(1)$

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

- Caso base?  $\longrightarrow$ 

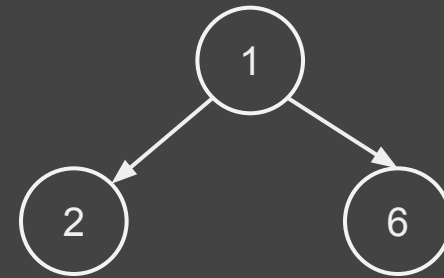
```
if (!ab) {  
    *altura = 0;  
    return true;  
}
```
- Caso recursivo?  $\longrightarrow$ 

```
if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
    return false;  
}  
  
if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
    return false;  
}  
  
if (diff(altura_izq, altura_der) > 1) {  
    return false;  
}  
  
*altura = max(altura_izq, altura_der) + 1;  
return true;
```

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

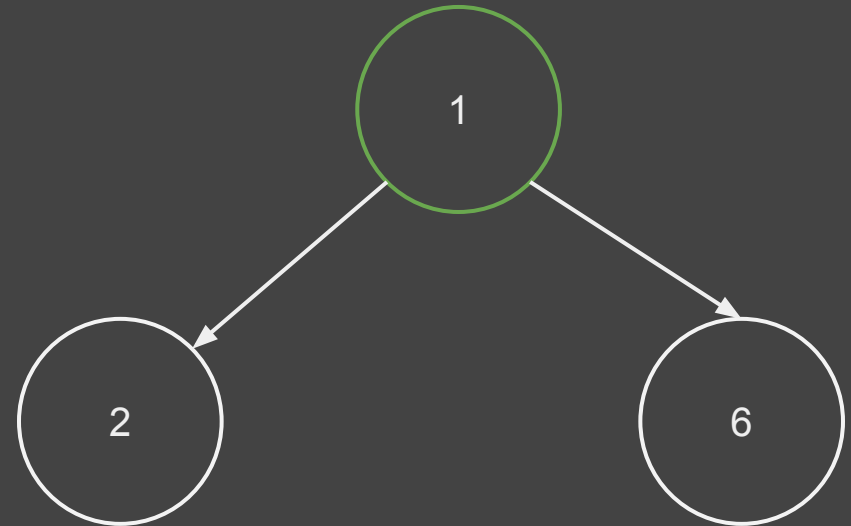
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

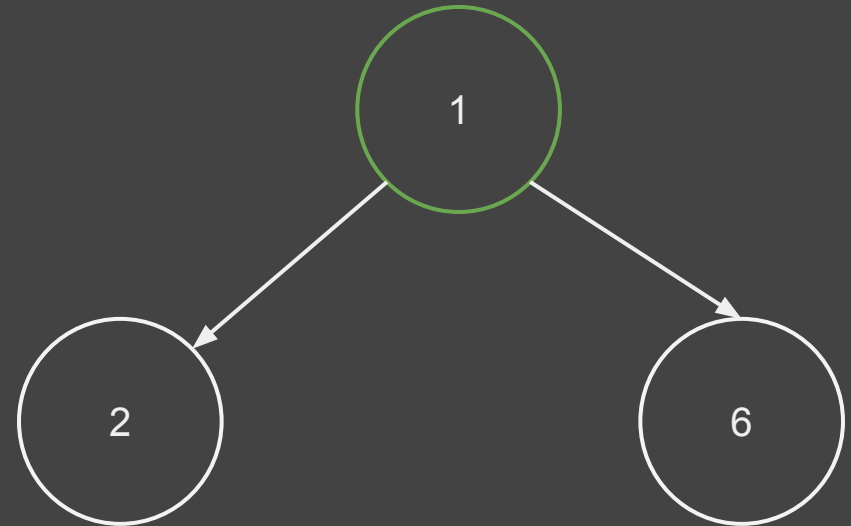




Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

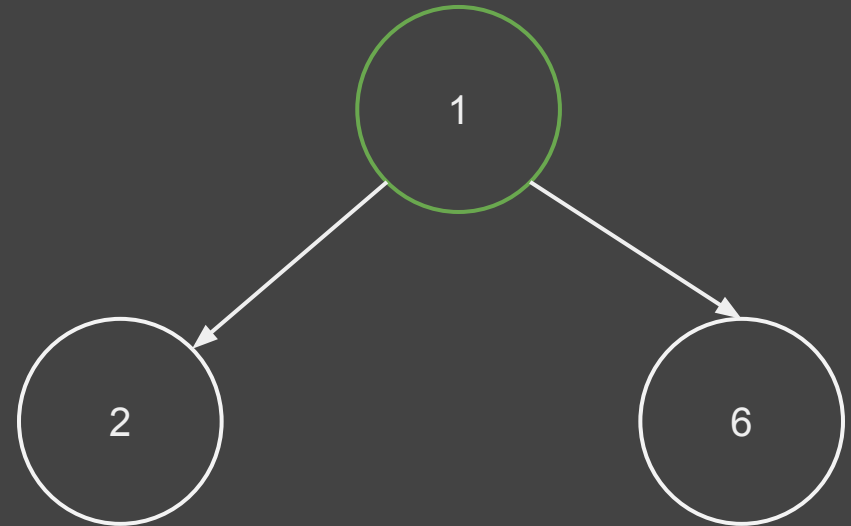
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

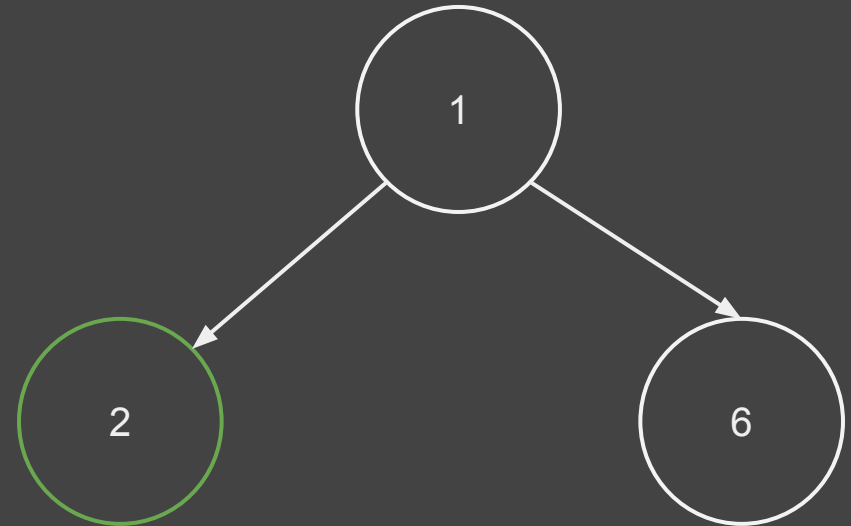


Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

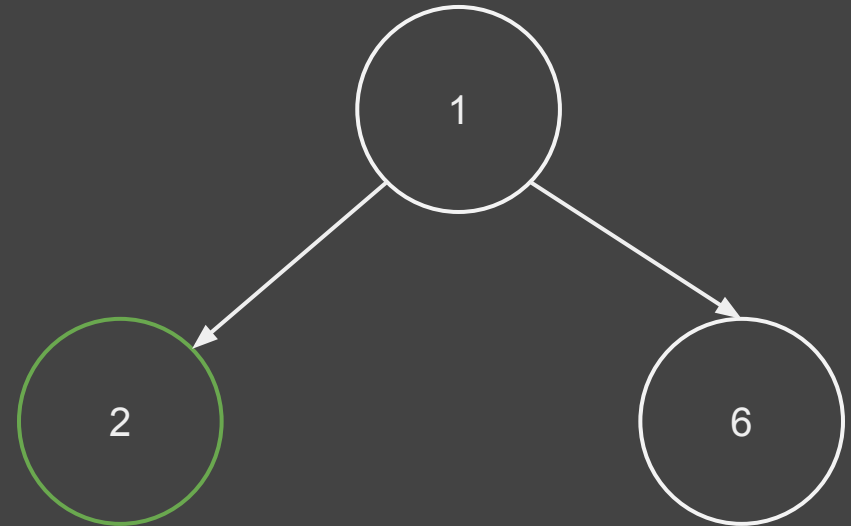
El 2 no es NULL



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

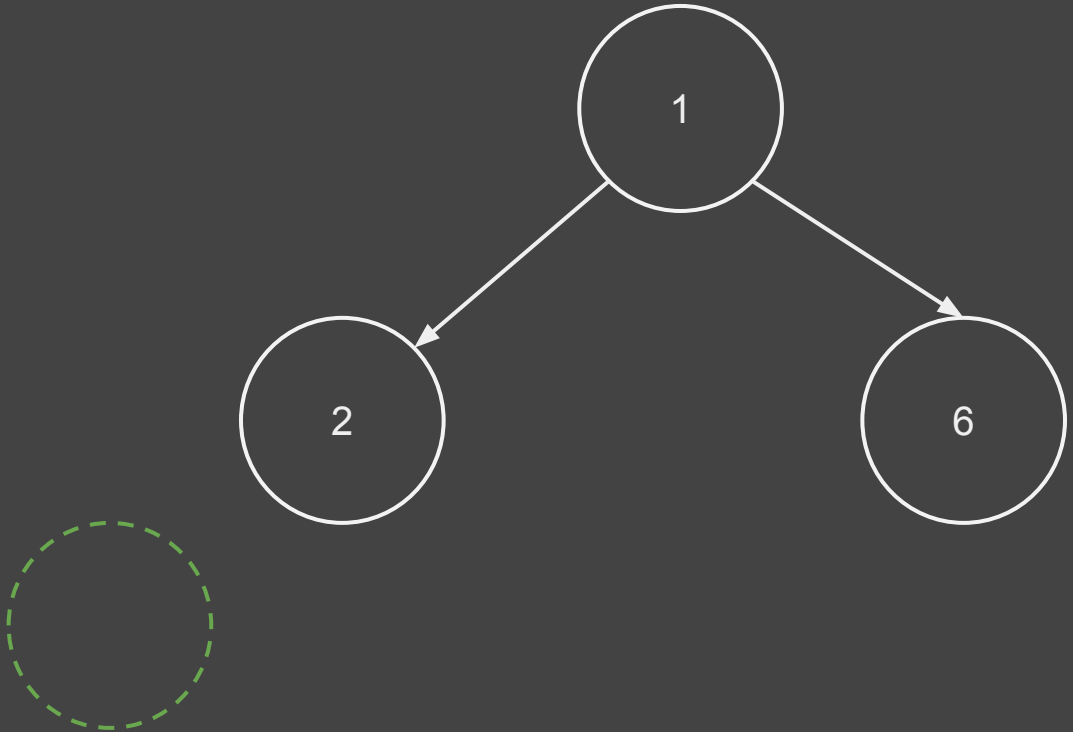
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

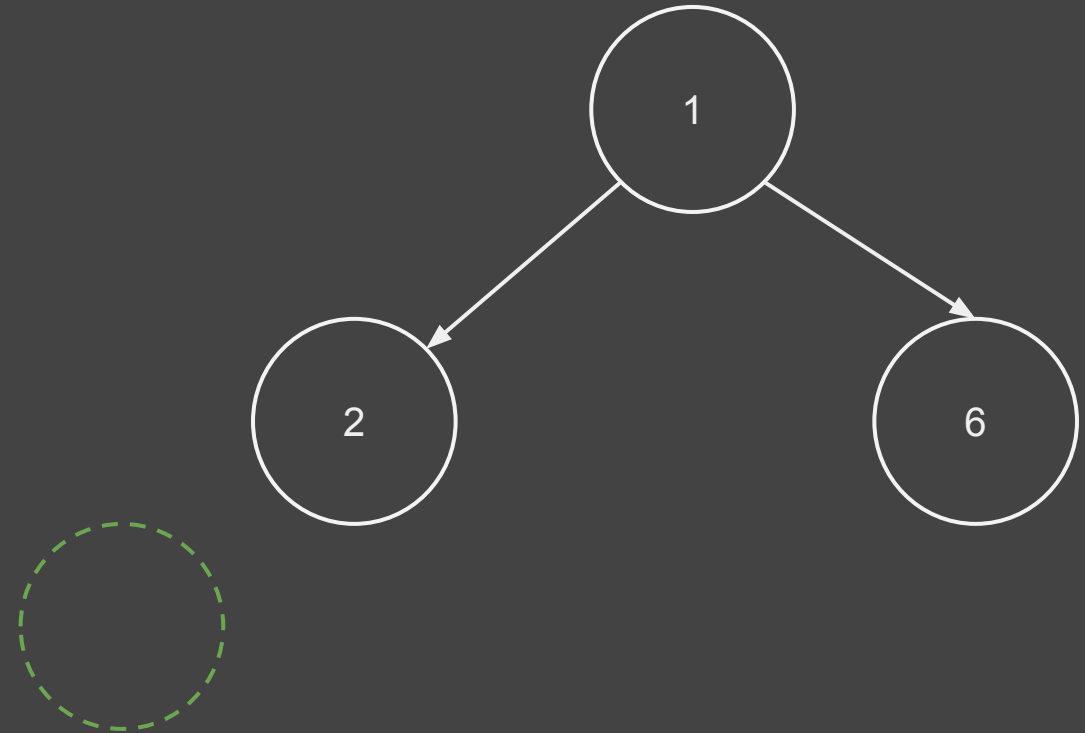
```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

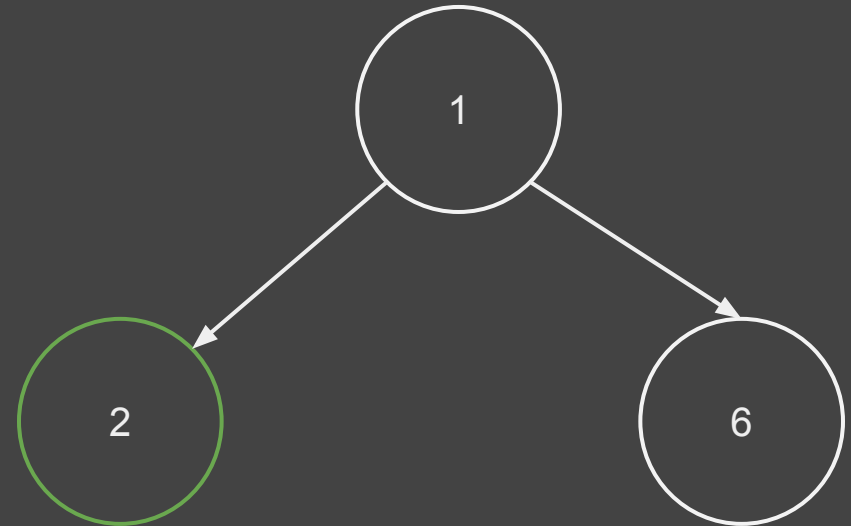
```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}  
  
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {  
    if (!ab) {  
        *altura = 0;  
        return true;  
    }  
    size_t altura_izq;  
    size_t altura_der;  
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
        return false;  
    }  
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
        return false;  
    }  
    if (diff(altura_izq, altura_der) > 1) {  
        return false;  
    }  
    *altura = max(altura_izq, altura_der) + 1;  
    return true;  
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

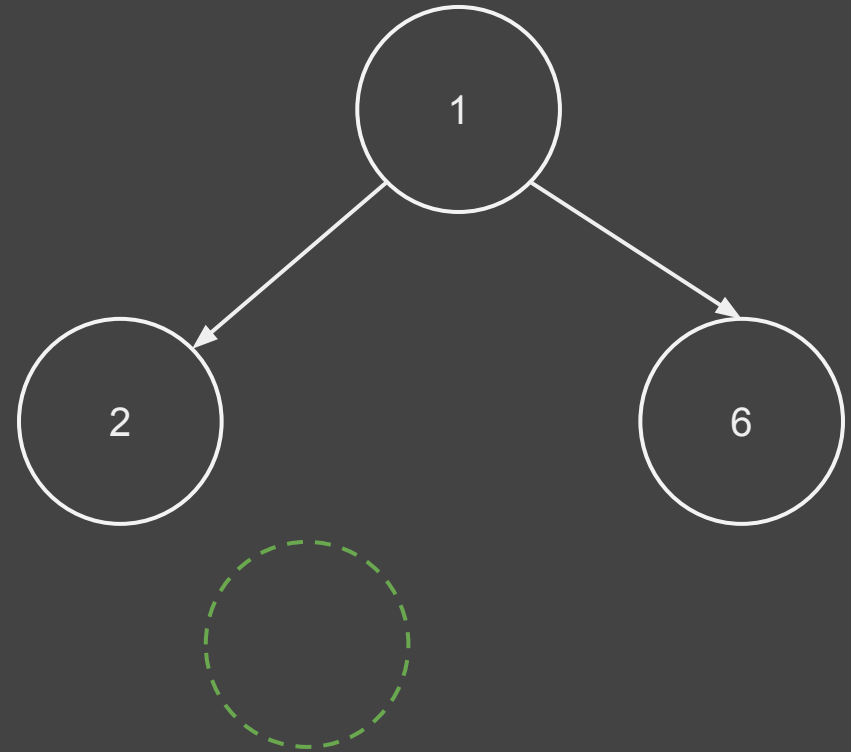
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

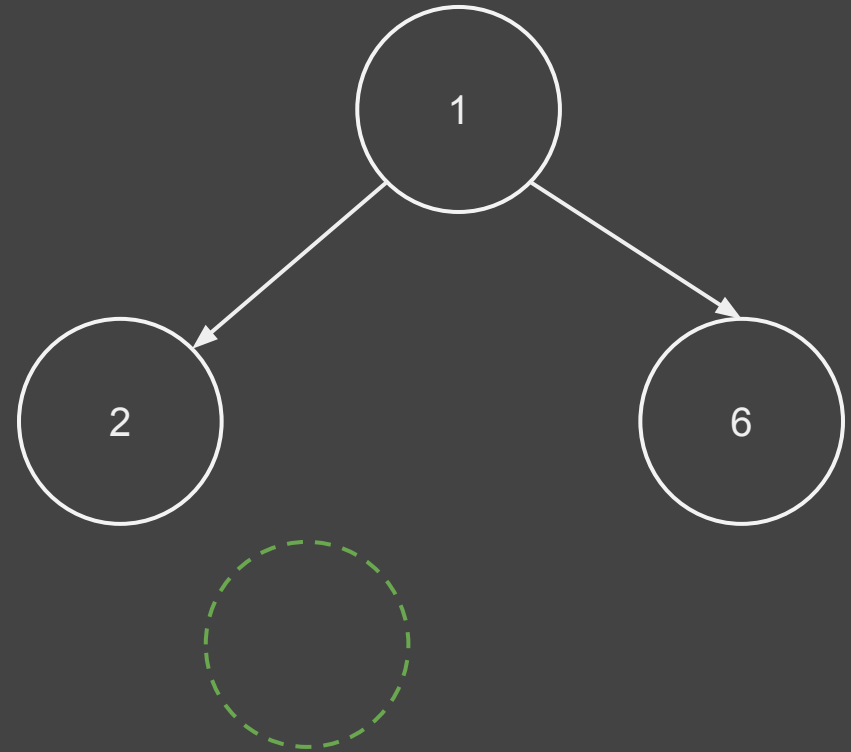




Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

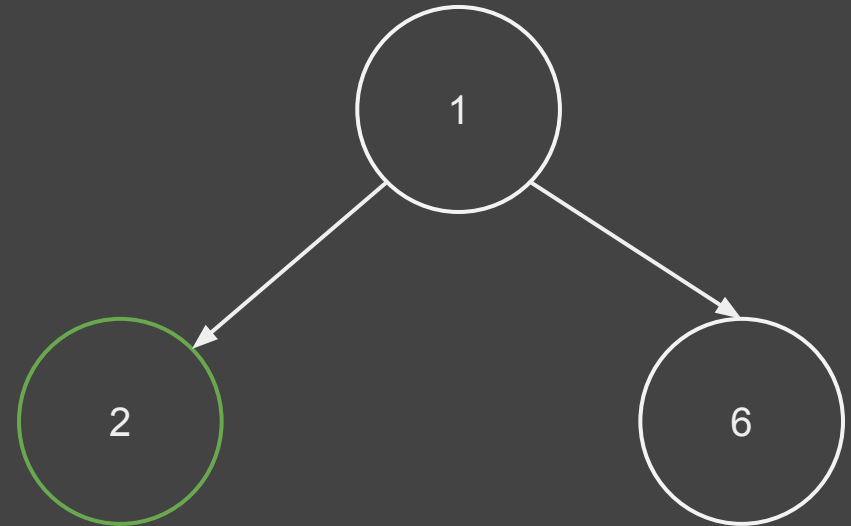
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

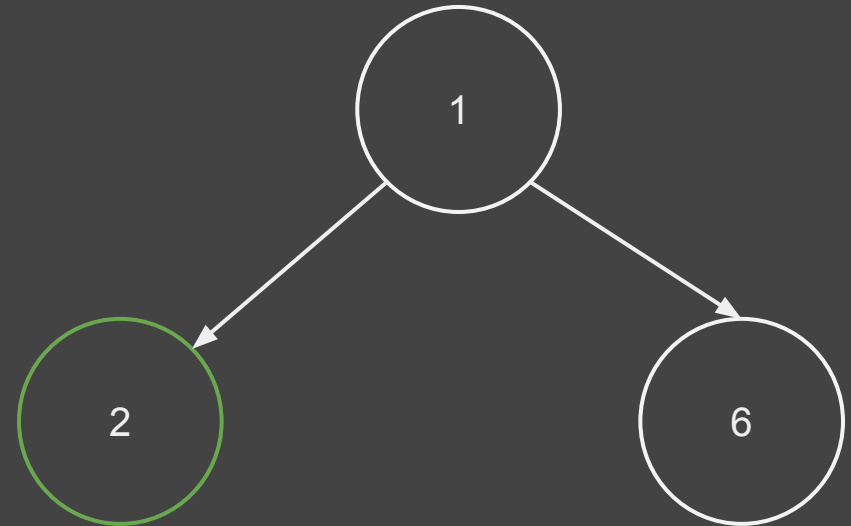
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

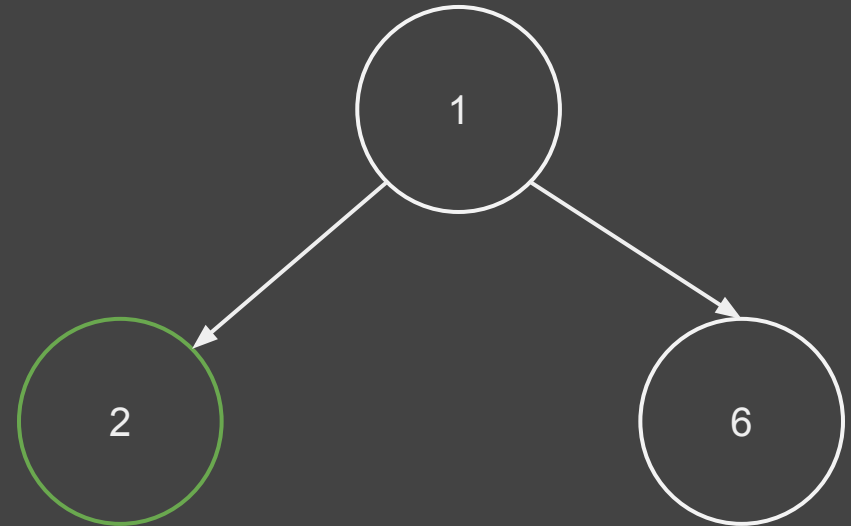
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

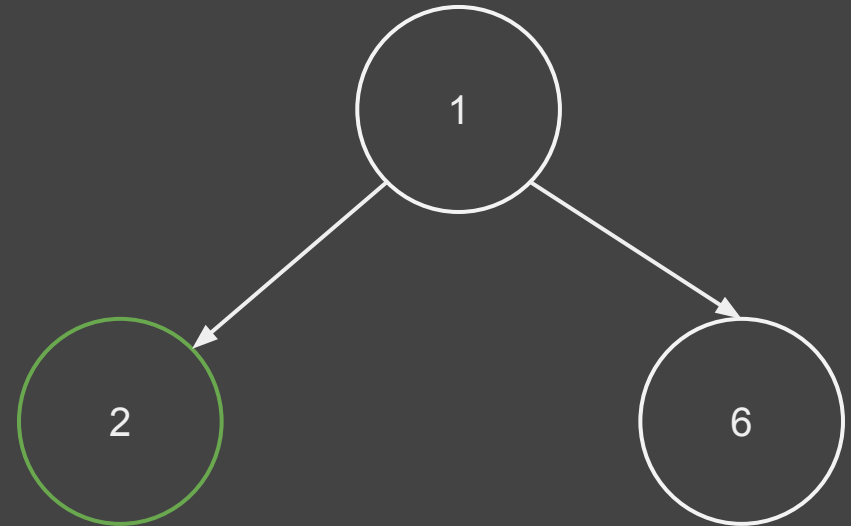
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

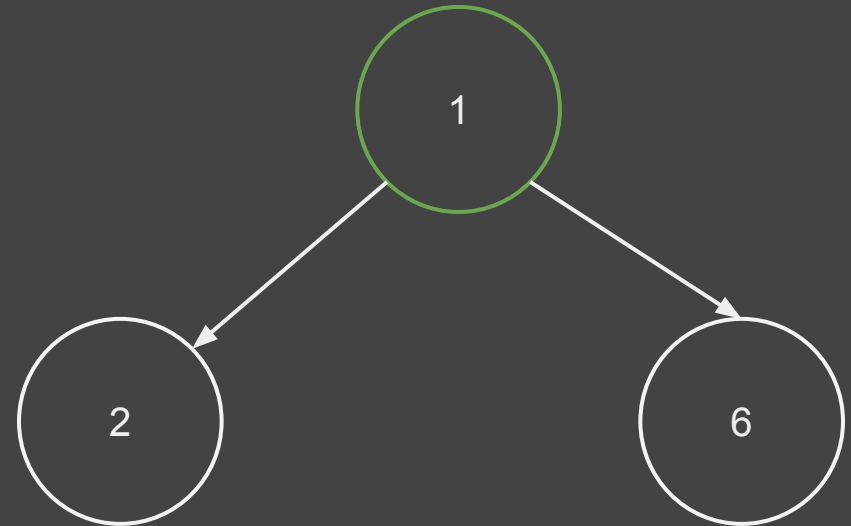


**Este nodo está balanceado**

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

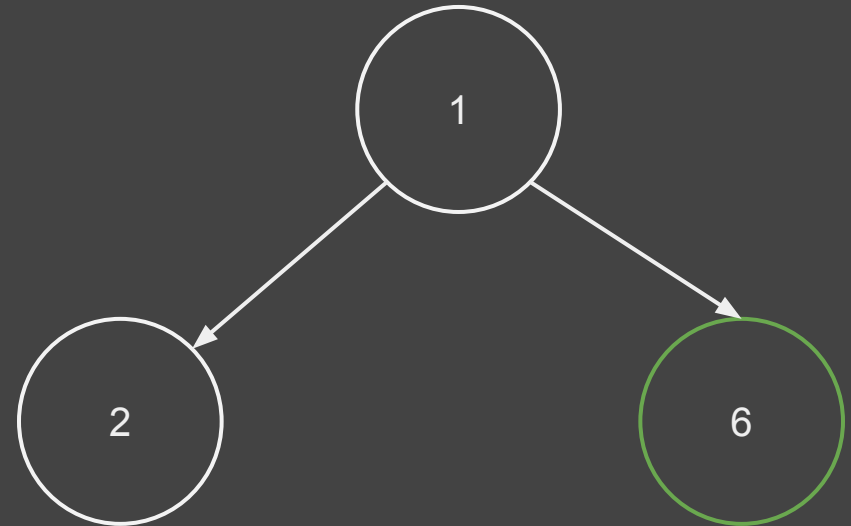
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

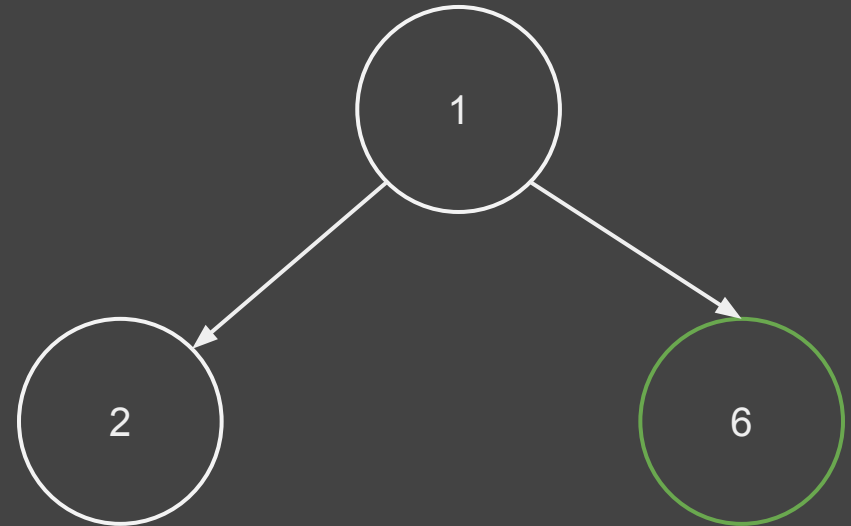
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) { El 6 no es NULL
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

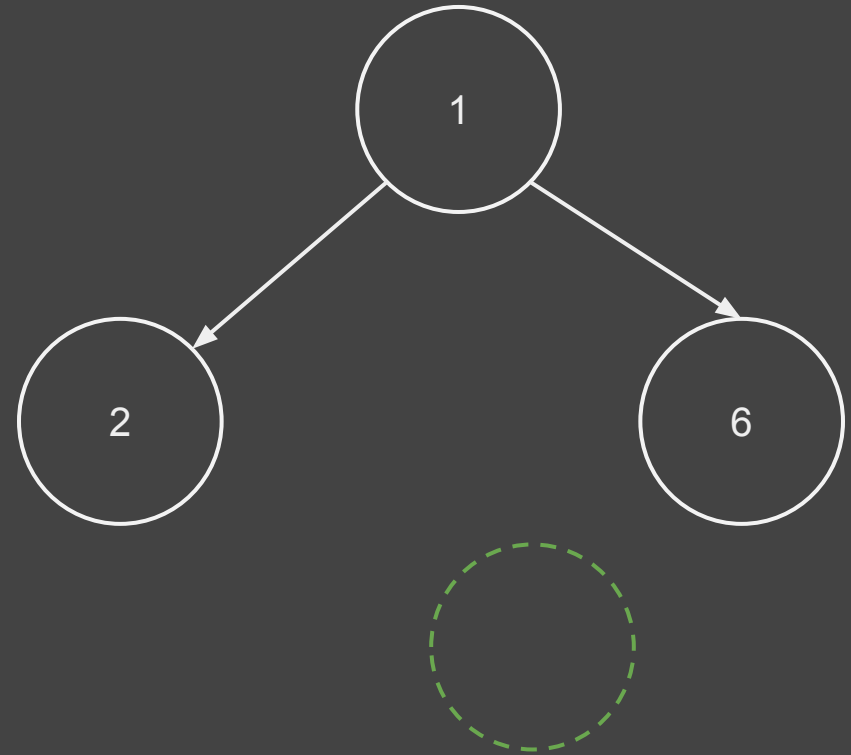




Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

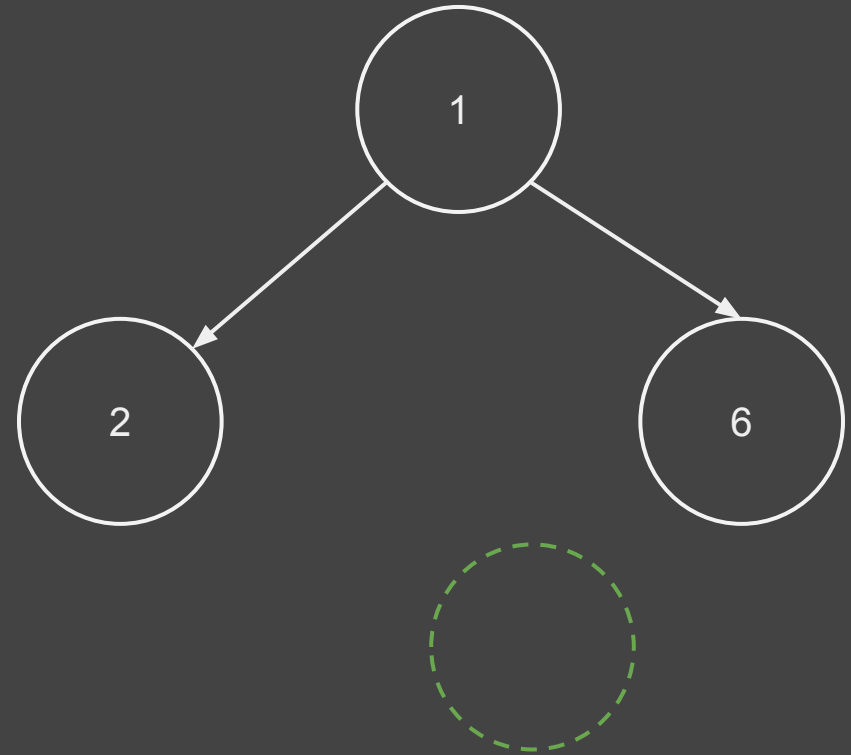
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

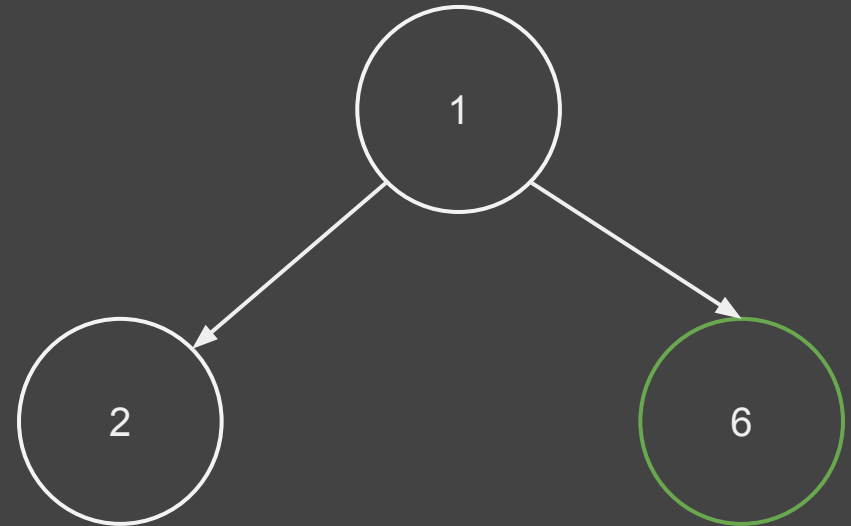
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

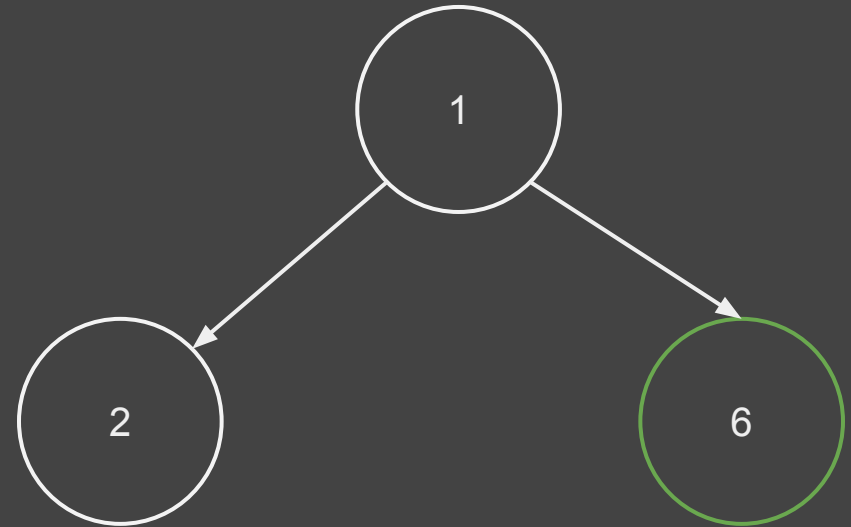
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

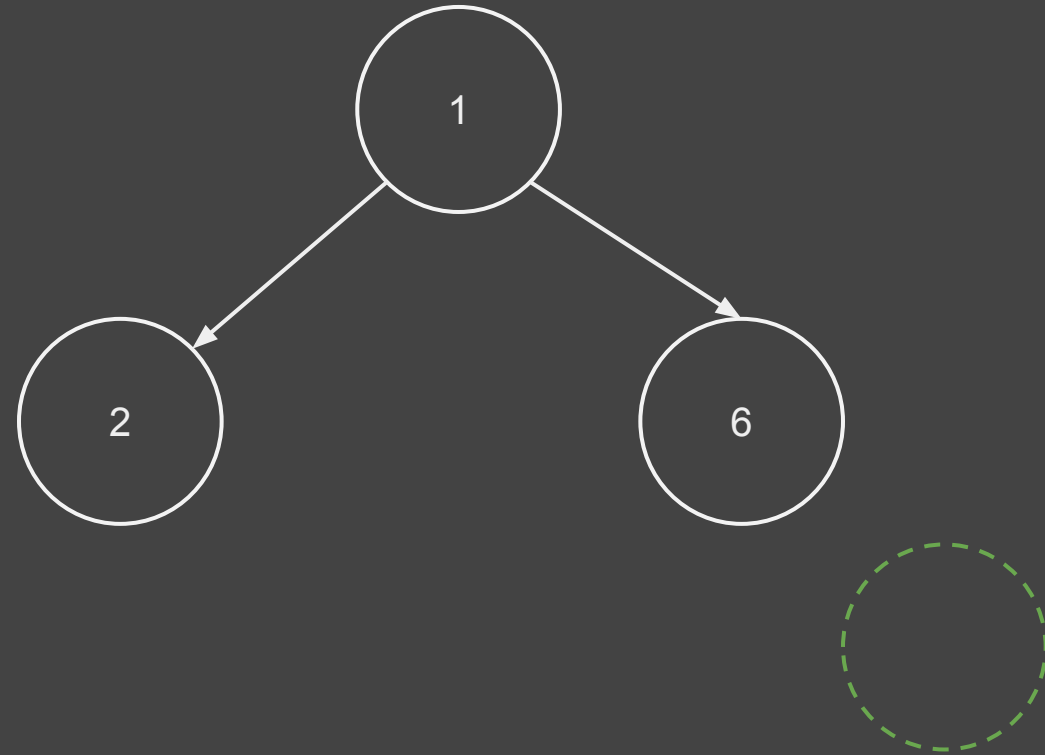
```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

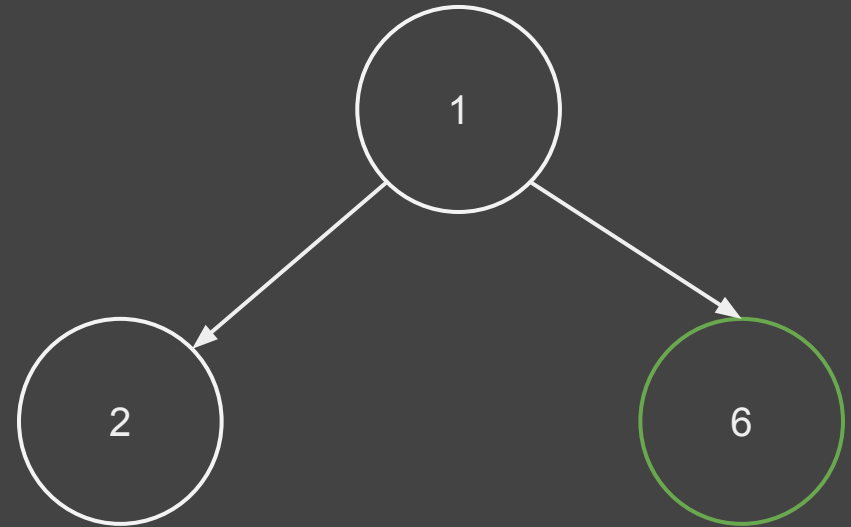
```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}  
  
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {  
    if (!ab) {  
        *altura = 0;  
        return true;  
    }  
    size_t altura_izq;  
    size_t altura_der;  
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
        return false;  
    }  
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
        return false;  
    }  
    if (diff(altura_izq, altura_der) > 1) {  
        return false;  
    }  
    *altura = max(altura_izq, altura_der) + 1;  
    return true;  
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

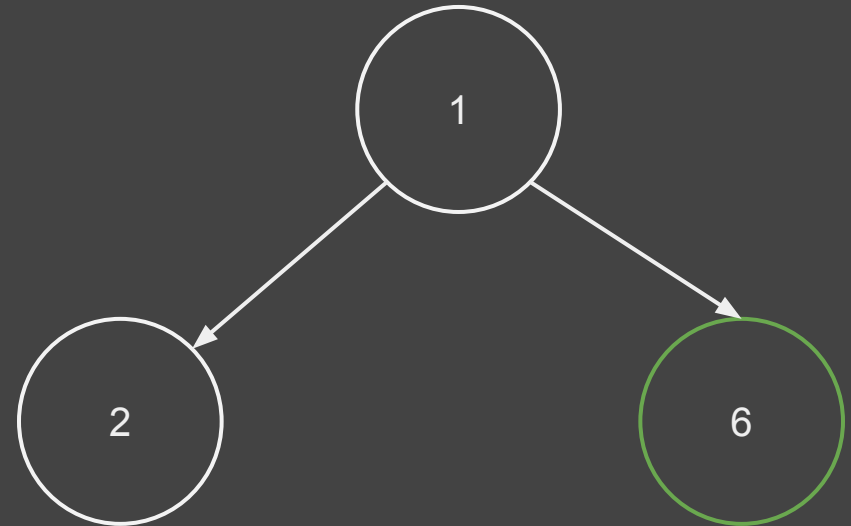
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

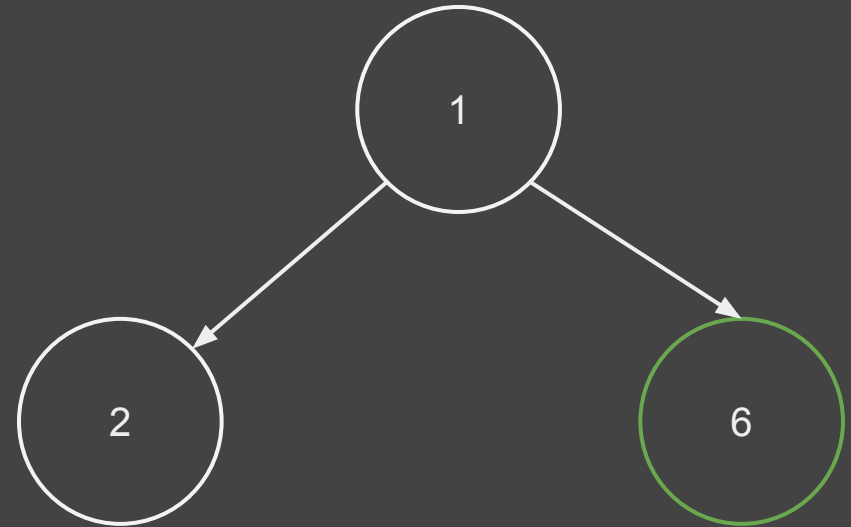
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



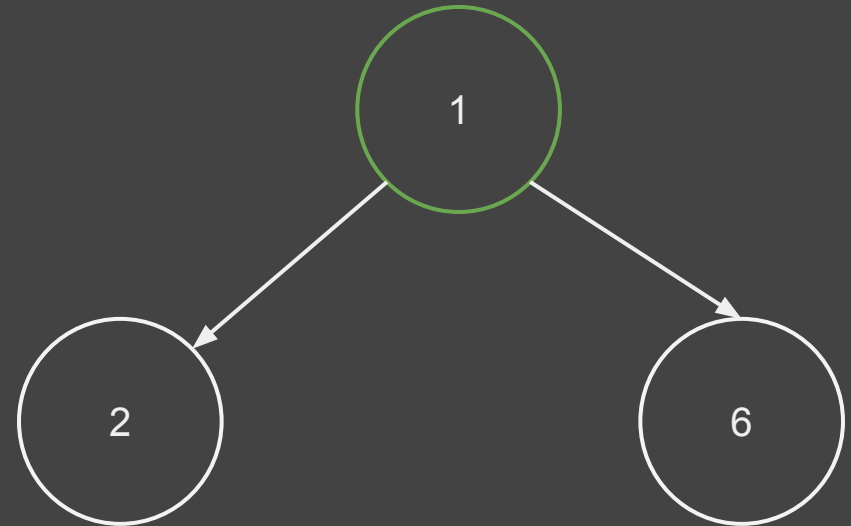
**Este nodo está balanceado**



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

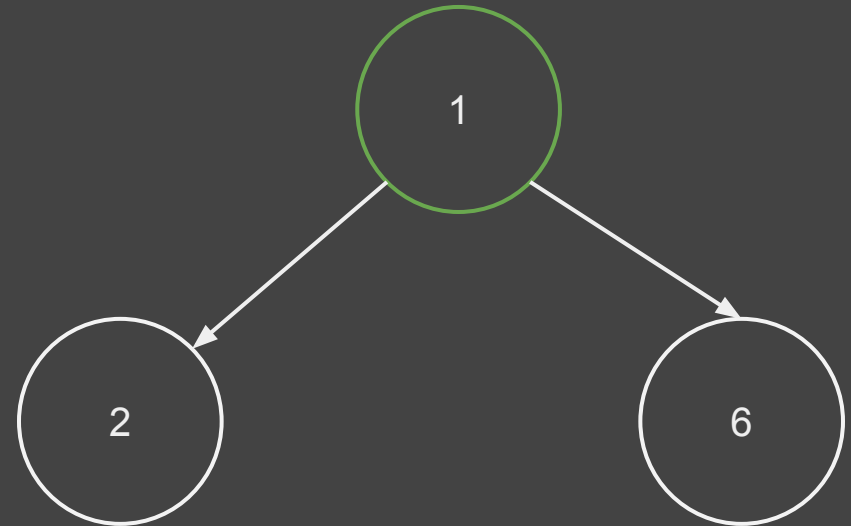


Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

Este nodo está balanceado

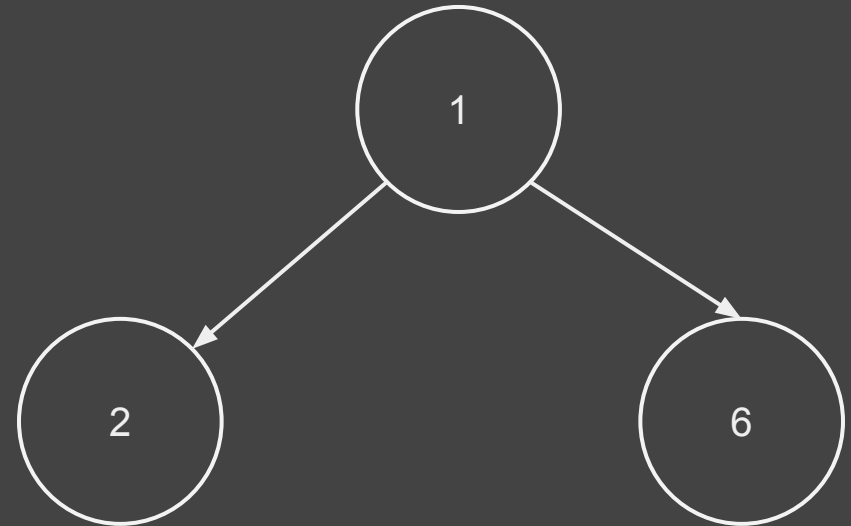


Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

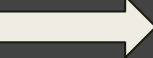
```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}
```

```
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {  
    if (!ab) {  
        *altura = 0;  
        return true;  
    }  
    size_t altura_izq;  
    size_t altura_der;  
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
        return false;  
    }  
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
        return false;  
    }  
    if (diff(altura_izq, altura_der) > 1) {  
        return false;  
    }  
    *altura = max(altura_izq, altura_der) + 1;  
    return true;  
}
```

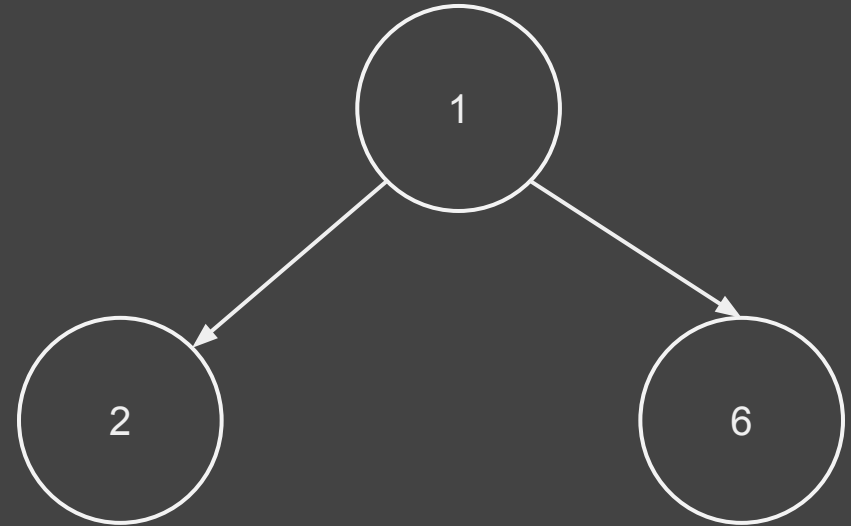
→ true



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  true
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}
```

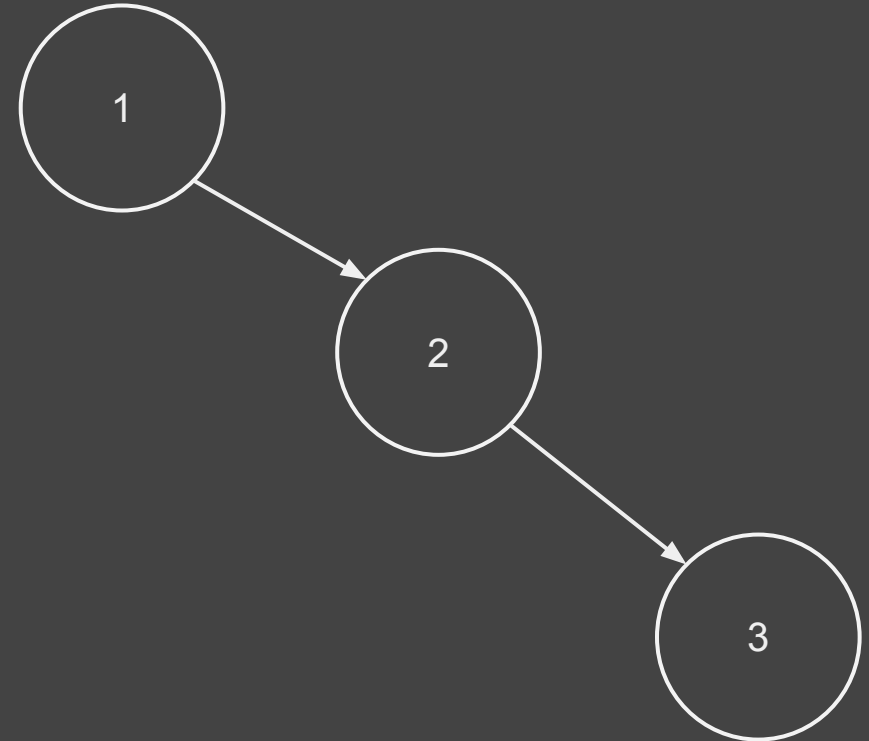
```
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

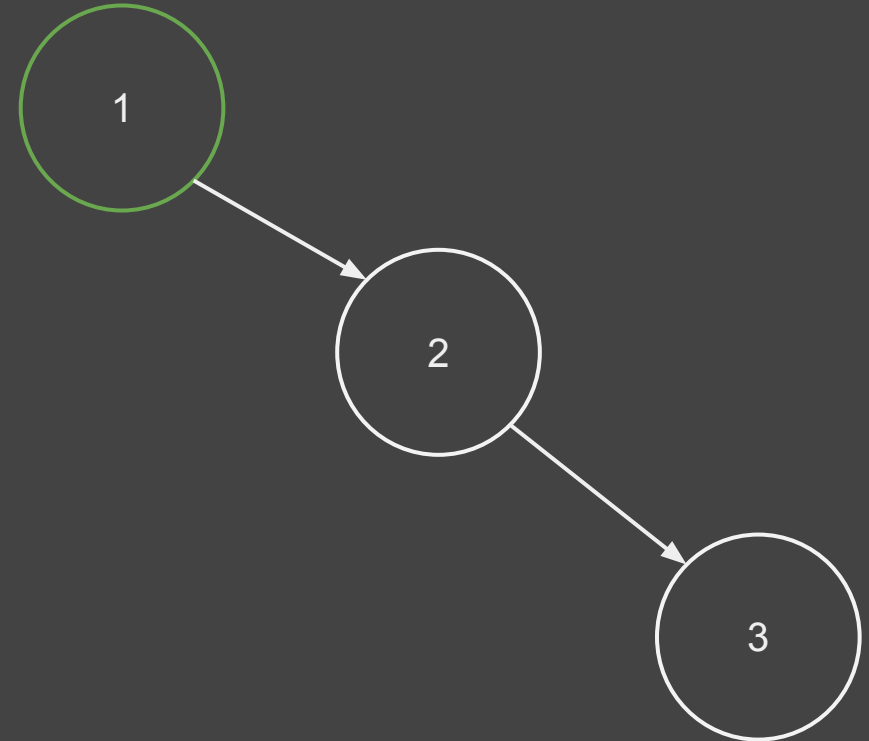
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

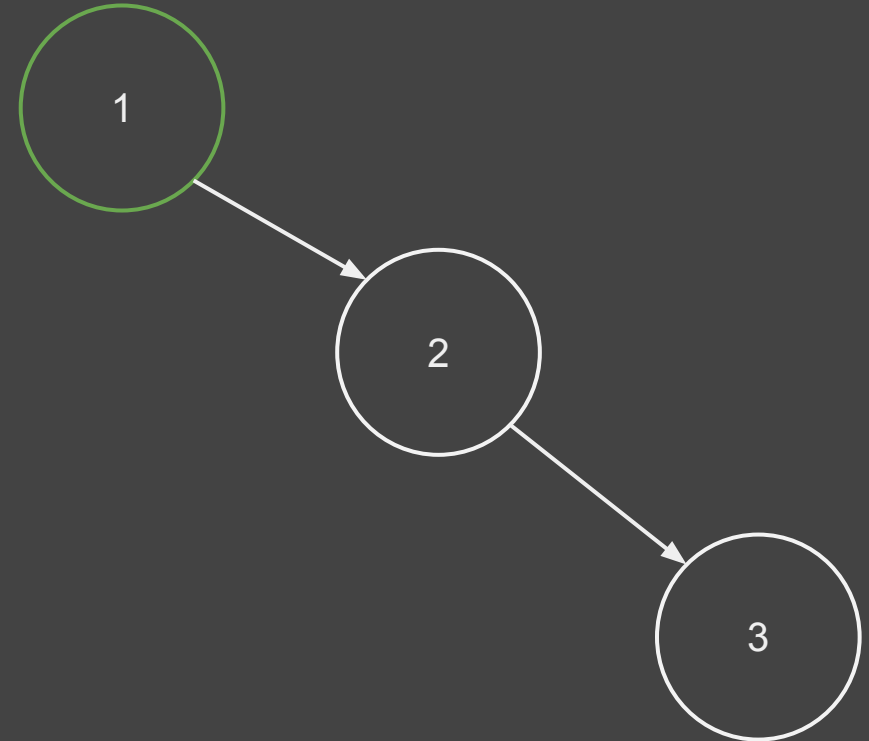
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

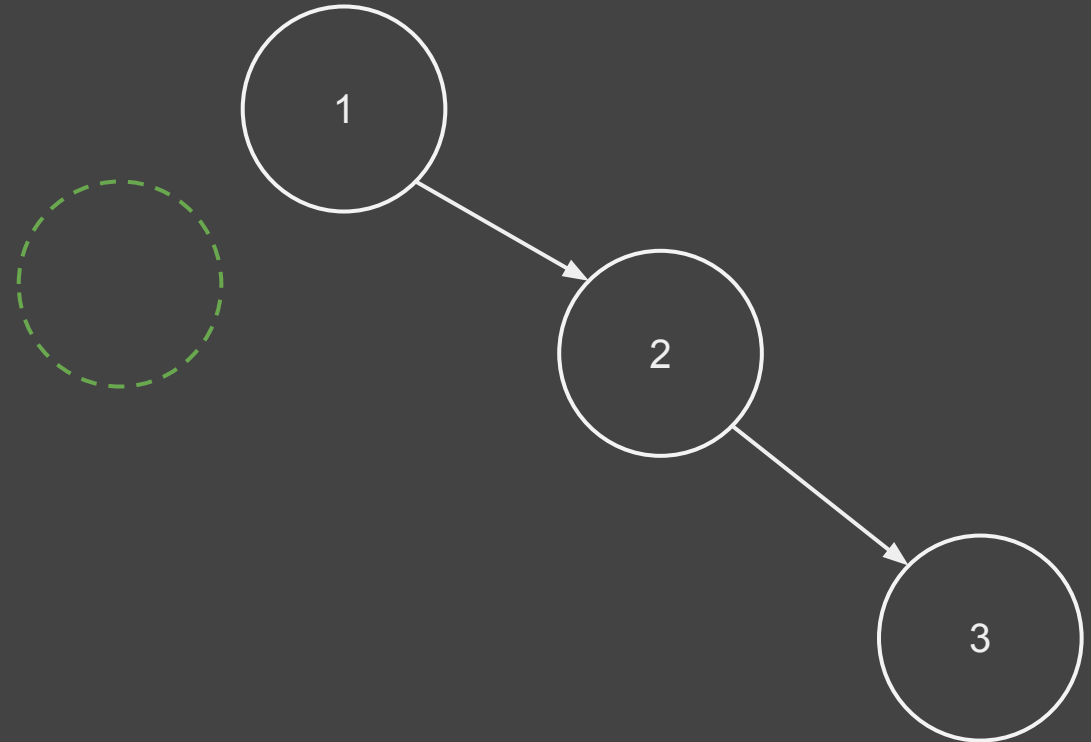
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

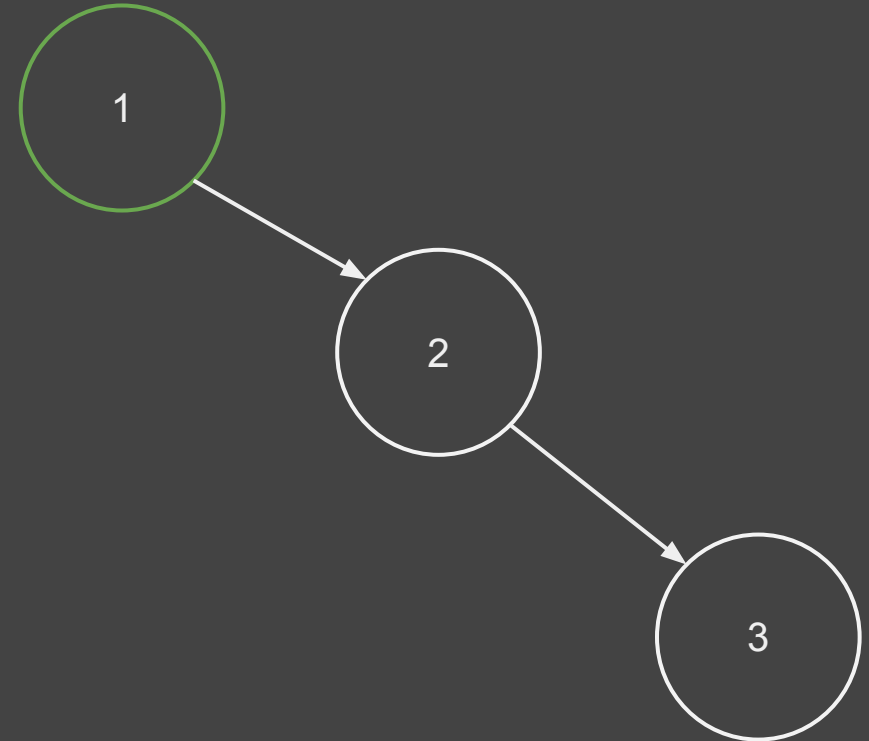




Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

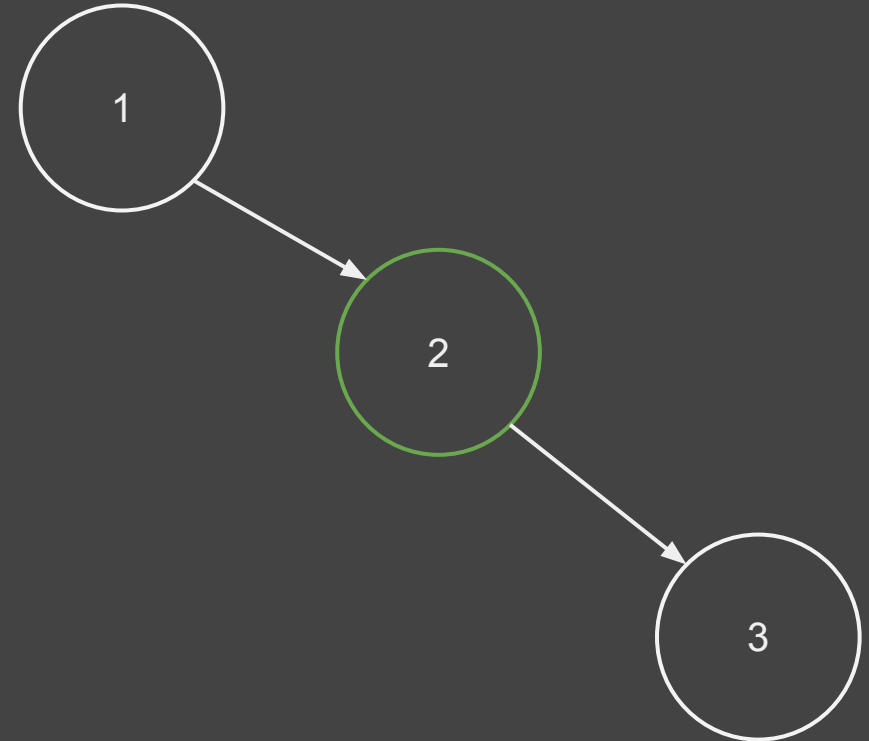
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

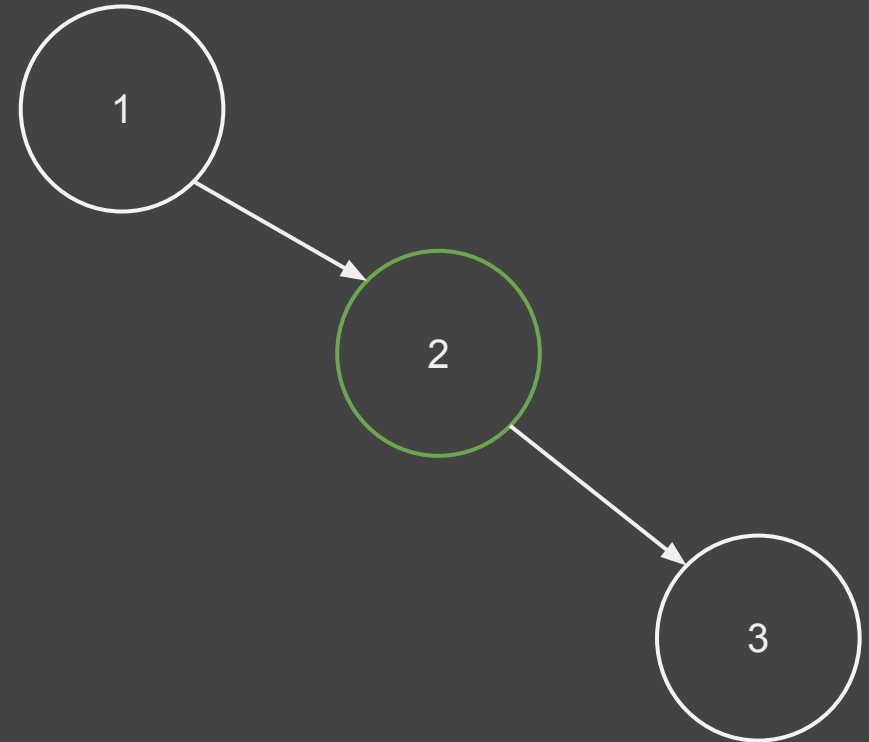
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

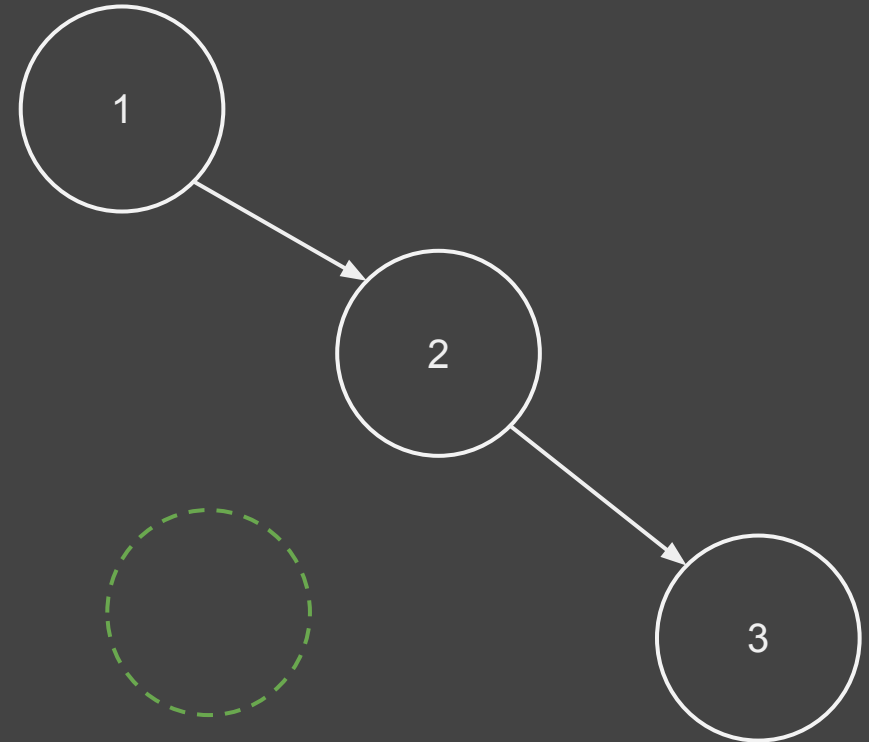
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

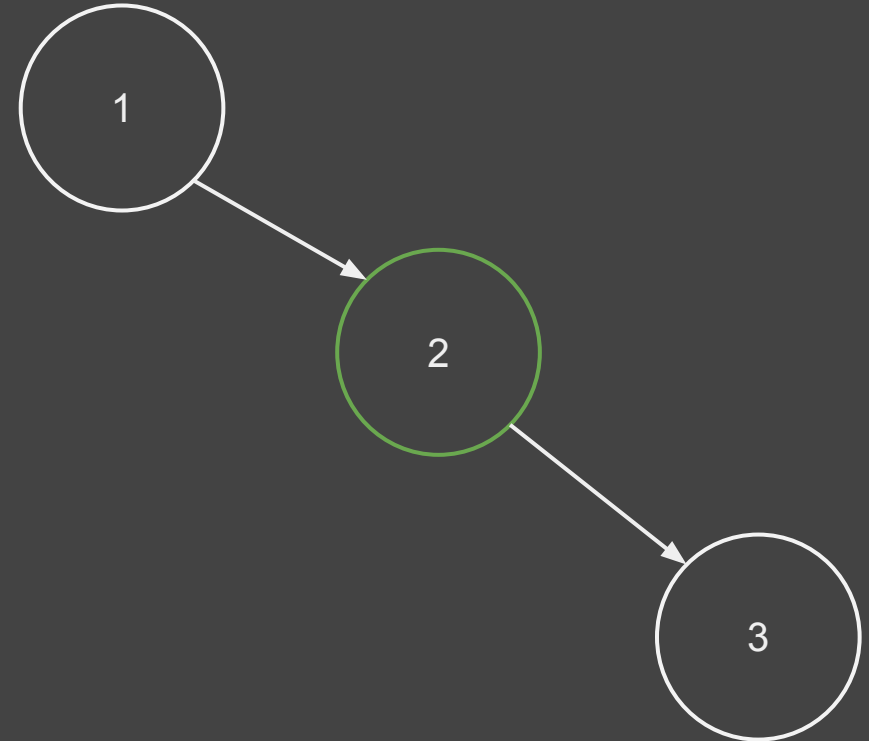
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

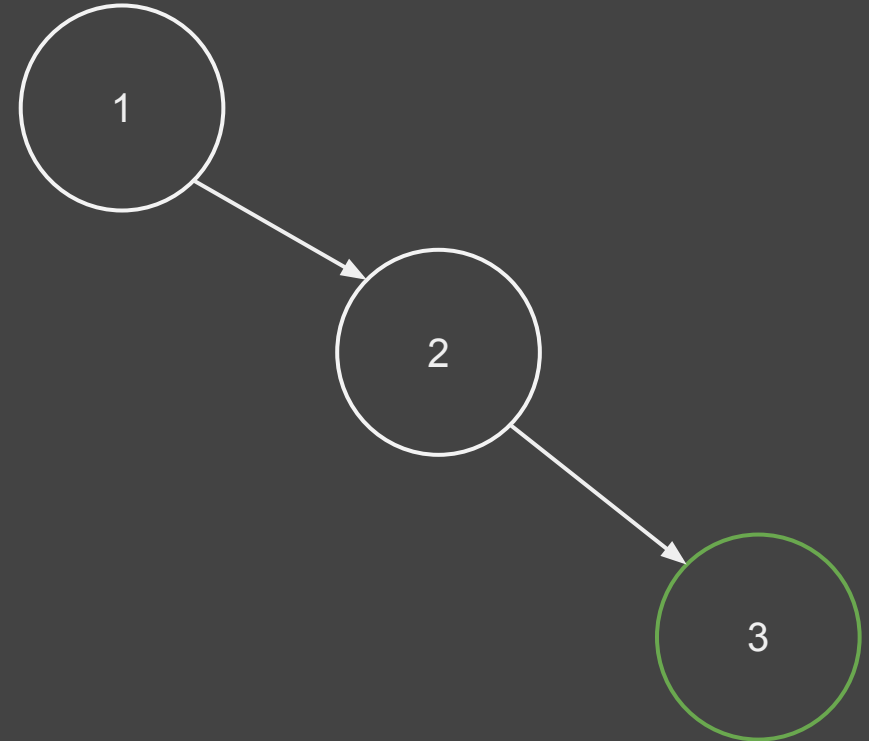
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

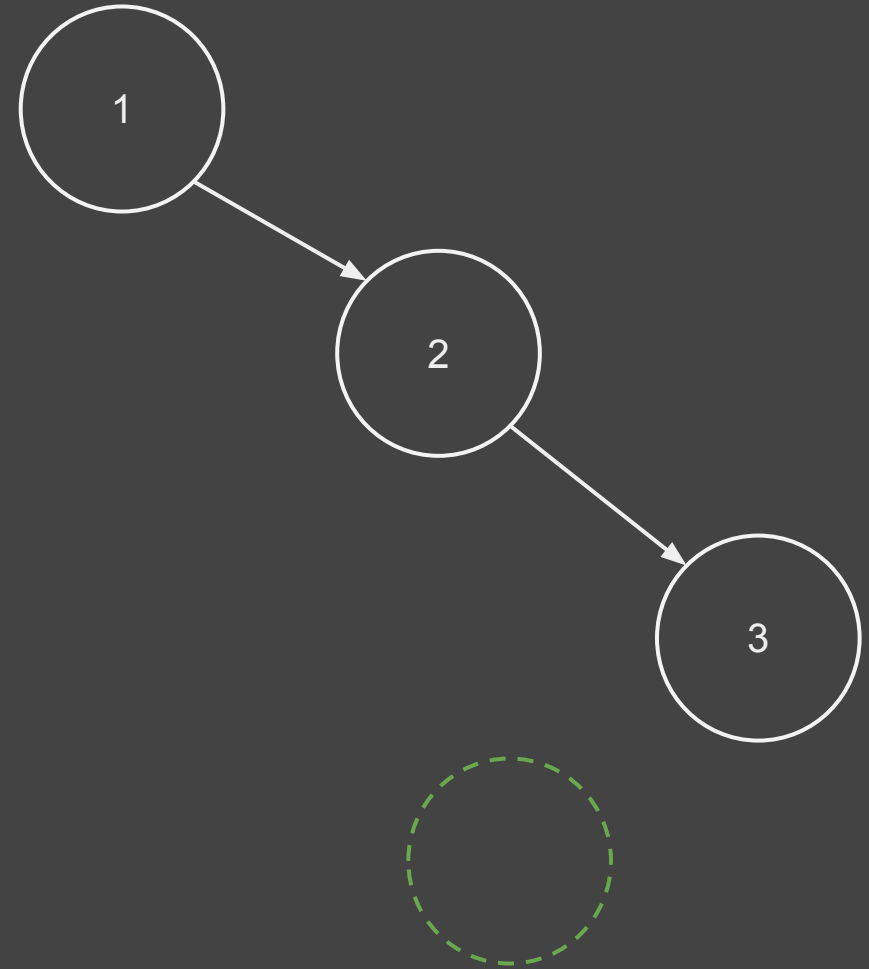
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

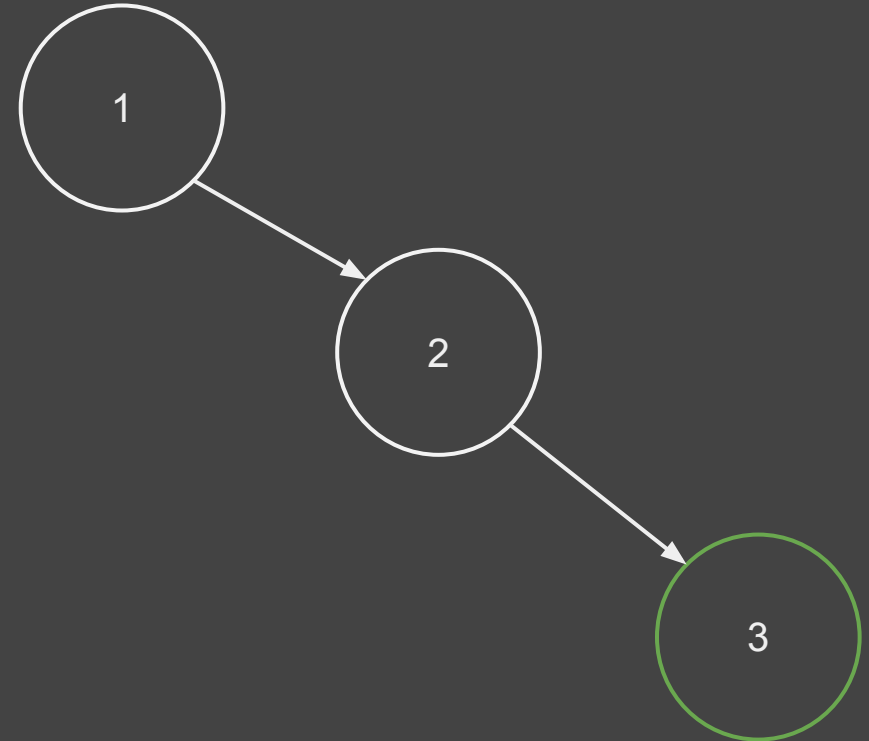
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

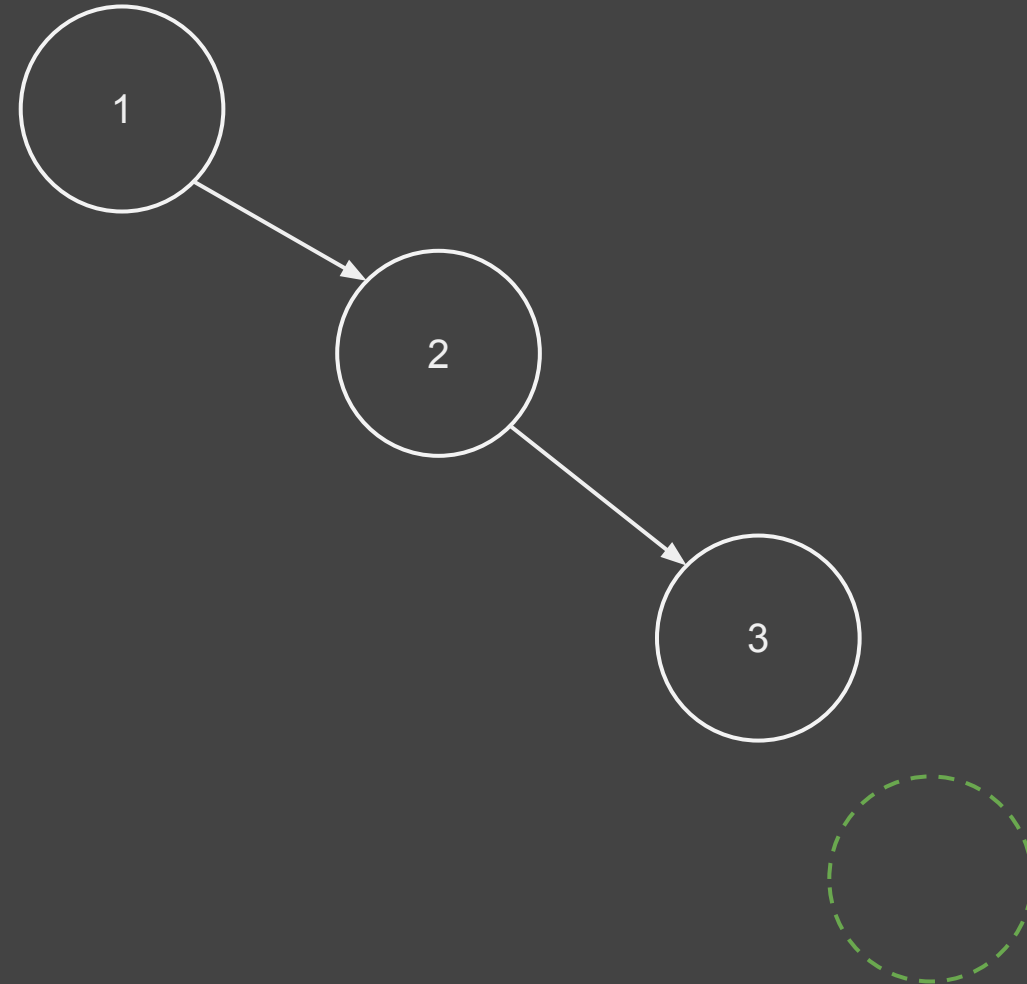




Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

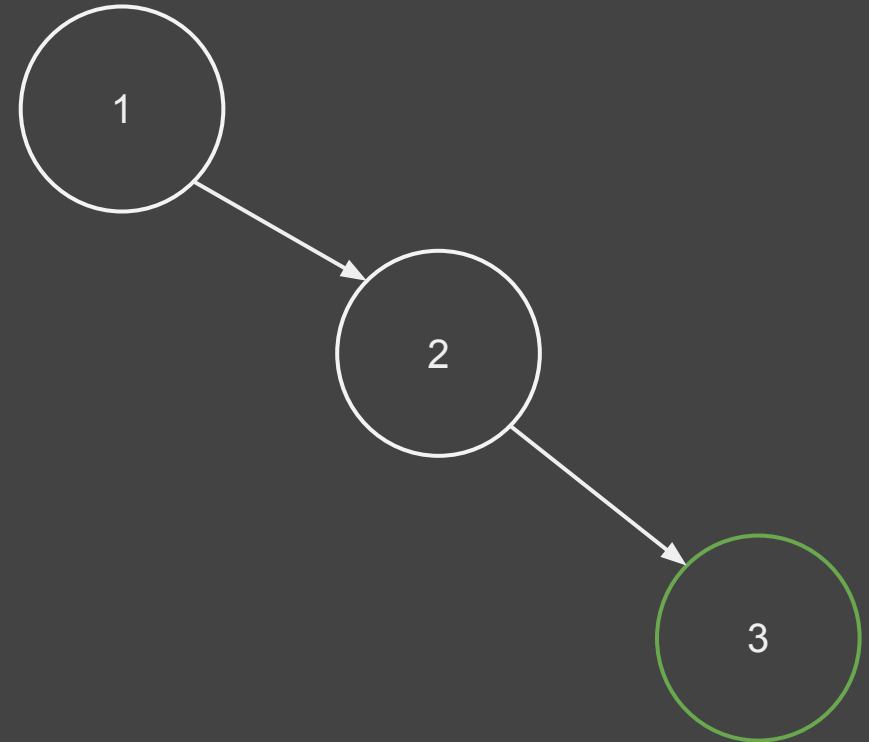
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

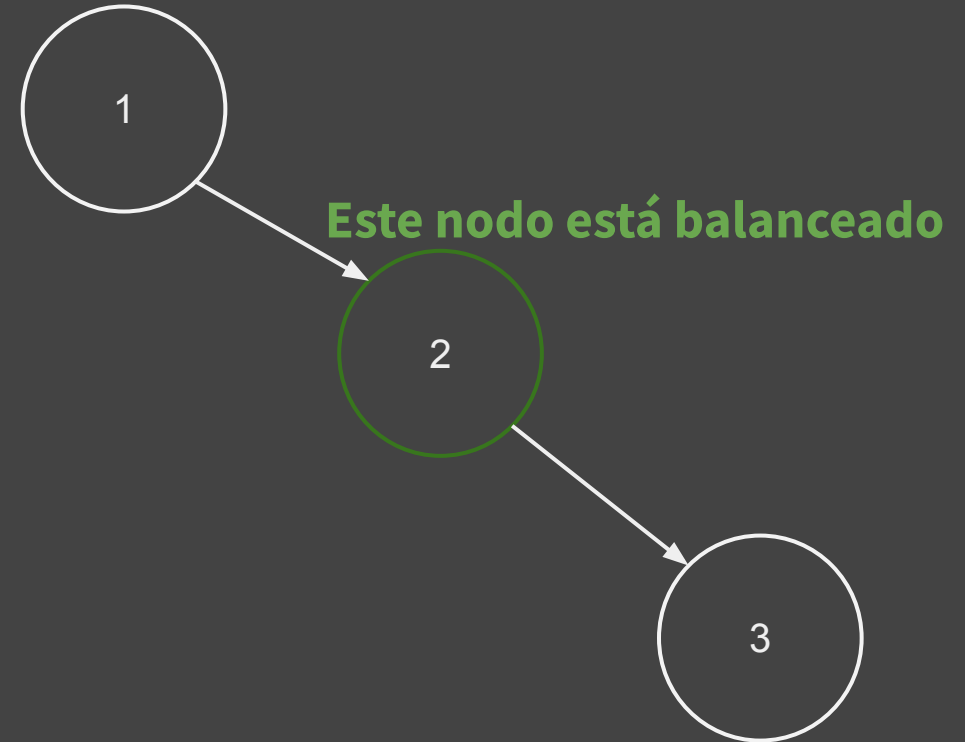


**Este nodo está balanceado**

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

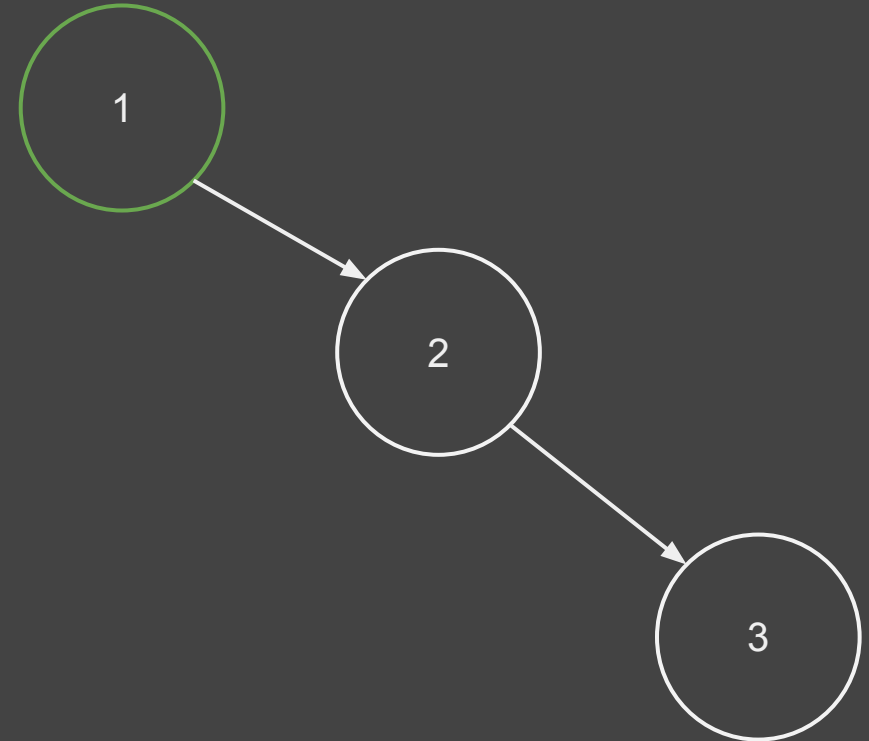
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

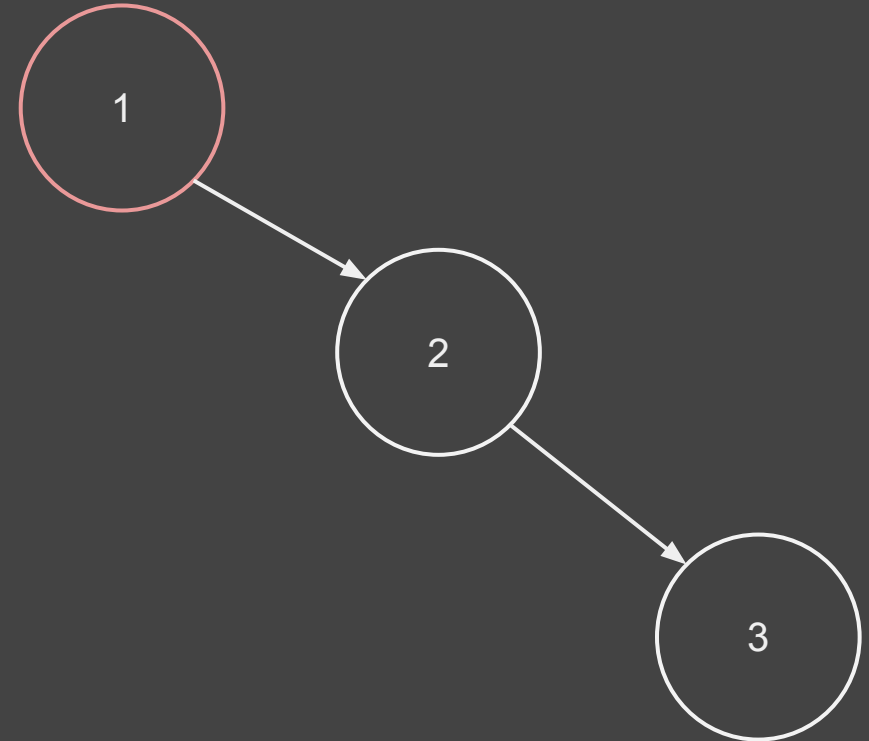


Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

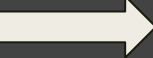
```
bool ab_prop_avl(const ab_t* ab) {
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}

bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

**Este nodo no está balanceado**

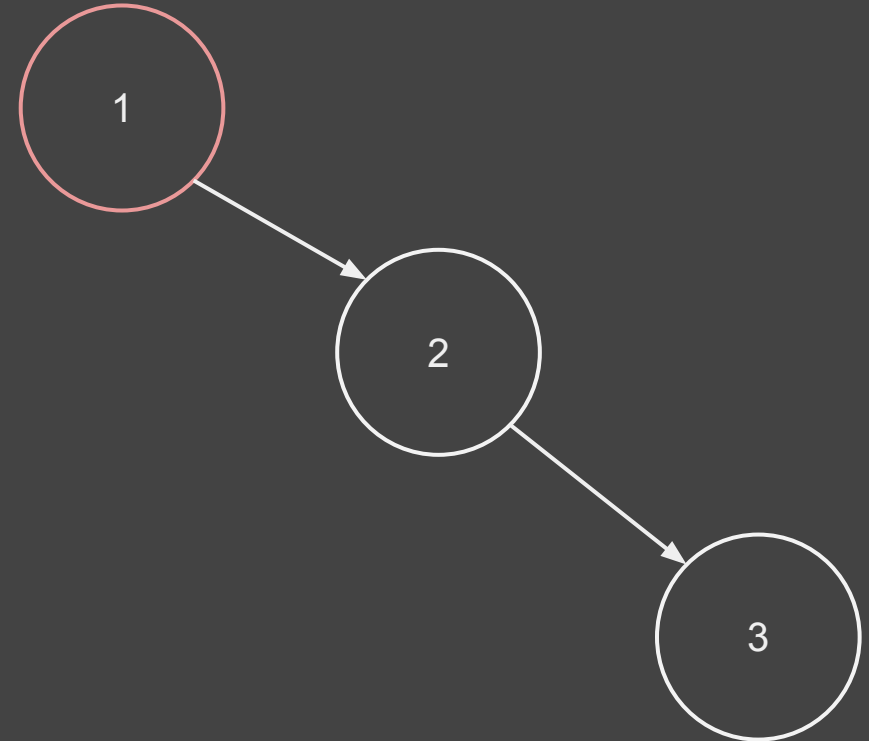


Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  false
    size_t h;
    return ab_prop_avl_rec(ab, &h);
}
```

```
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
    if (!ab) {
        *altura = 0;
        return true;
    }
    size_t altura_izq;
    size_t altura_der;
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {
        return false;
    }
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {
        return false;
    }
    if (diff(altura_izq, altura_der) > 1) {
        return false;
    }
    *altura = max(altura_izq, altura_der) + 1;
    return true;
}
```

**Este nodo no está balanceado**



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}
```

## Complejidad?

$$T(n) = 2 * T(n/2) + O(1)$$

```
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {  
    if (!ab) {  
        *altura = 0;  
        return true;  
    }  
    size_t altura_izq;  
    size_t altura_der;  
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
        return false;  
    }  
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
        return false;  
    }  
    if (diff(altura_izq, altura_der) > 1) {  
        return false;  
    }  
    *altura = max(altura_izq, altura_der) + 1;  
    return true;  
}
```

$O(1)$

$T(n/2)$

$T(n/2)$

$O(1)$

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}
```

```
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {
```

```
    if (!ab) {  
        *altura = 0;  
        return true;  
    }
```

$O(1)$

```
    size_t altura_izq;  
    size_t altura_der;
```

```
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
        return false;  
    }
```

$T(n/2)$

```
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
        return false;  
    }
```

$T(n/2)$

```
    if (diff(altura_izq, altura_der) > 1) {  
        return false;  
    }
```

$O(1)$

```
    *altura = max(altura_izq, altura_der) + 1;  
    return true;  
}
```

## Complejidad?

$$T(n) = 2 \cdot T(n/2) + O(1)$$

Teorema Maestro:

$$\left. \begin{array}{l} A = 2 \\ B = 2 \\ C = 0 \end{array} \right\} \log_B(A) > C$$

$$T(n) = O(n^1)$$



Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}
```

## Recorrido?

```
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {  
    if (!ab) {  
        *altura = 0;  
        return true;  
    }  
    size_t altura_izq;  
    size_t altura_der;  
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
        return false;  
    }  
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
        return false;  
    }  
    if (diff(altura_izq, altura_der) > 1) {  
        return false;  
    }  
    *altura = max(altura_izq, altura_der) + 1;  
    return true;  
}
```

Trabajo sobre sub-árbol izquierdo

Trabajo sobre sub-árbol derecho

visito la raíz (actual)

Implementar una primitiva que dado un árbol binario, devuelve si dicho árbol cumple con la propiedad de AVL. La primitiva no debe ejecutar en más que  $O(n)$ .

```
bool ab_prop_avl(const ab_t* ab) {  
    size_t h;  
    return ab_prop_avl_rec(ab, &h);  
}
```

Recorrido?

```
bool ab_prop_avl_rec(const ab_t* ab, size_t* altura) {  
    if (!ab) {  
        *altura = 0;  
        return true;  
    }  
    size_t altura_izq;  
    size_t altura_der;  
    if (!ab_prop_avl_rec(ab->izq, &altura_izq)) {  
        return false;  
    }  
    if (!ab_prop_avl_rec(ab->der, &altura_der)) {  
        return false;  
    }  
    if (diff(altura_izq, altura_der) > 1) {  
        return false;  
    }  
    *altura = max(altura_izq, altura_der) + 1;  
    return true;  
}
```

Trabajo sobre sub-árbol izquierdo

Trabajo sobre sub-árbol derecho

Visito la raíz

POSTORDER