OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 5 Árboles
5.2. Árboles Binarios de Búsqueda (ABB)

Profesores EDA

Índice

- **Introducción (conceptos básicos)**
- TAD Árbol Binario
  - Recorridos
  - Implementación
- **TAD Árbol Binario de Búsqueda**
- Equilibrado de árboles

Árbol Binario de Búsqueda (Binary Search Tree (BST)) - Introducción

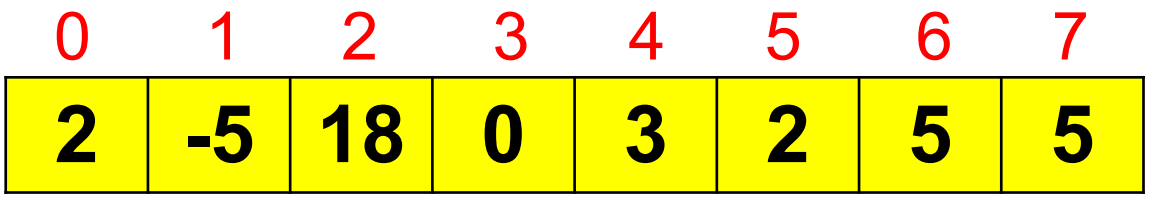
- ¿Qué estructura de datos usarías para almacenar una colección de datos sobre la que realizar las siguientes operaciones?

- *Search(x)*
- *Insert(x)*
- *Remove(x)*

Fuente: Bogdan Rosu Creative,CC BY 4.0, Iconfinder.

Introducción

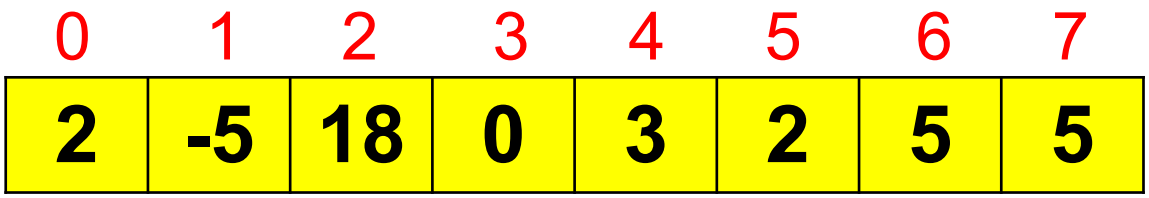
Python List



Operación	**Complejidad temporal**
search(x)	
insert(x)	
remove(x)	

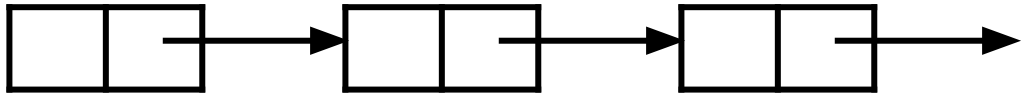
Introducción

Python List



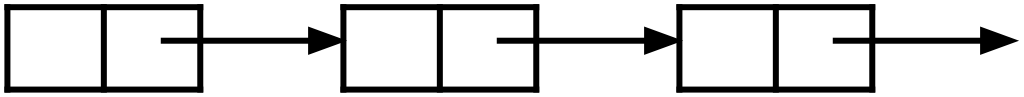
Operación	**Complejidad temporal**
search(x)	O(n)
insert(x)	O(1) / O(n)
remove(x)	O(1) / O(n)

Introducción

Lista enlazada  None

Operación	**Complejidad temporal**
search(x)	
insert(x)	
remove(x)	

Introducción

Lista enlazada  None

Operación	**Complejidad temporal**
search(x)	O(n)
insert(x)	O(1) / O(n)
remove(x)	O(1) / O(n)

Introducción

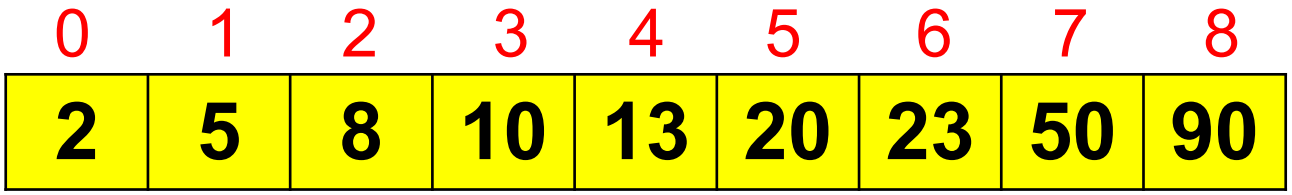
- Supongamos que el tiempo de ejecución necesario para realizar una comparación de un elemento en la lista = 10^{-6} segundos.

- Si la búsqueda se realizará sobre una lista almacenando los usuarios de Facebook (con más de 1 billón de usuarios):

  - 1 billón (10^8) x 10^{-6} segundos = 100 segundos!!!

Introducción

Python List (ordenada)



n

n/2

n/4

n/8

…

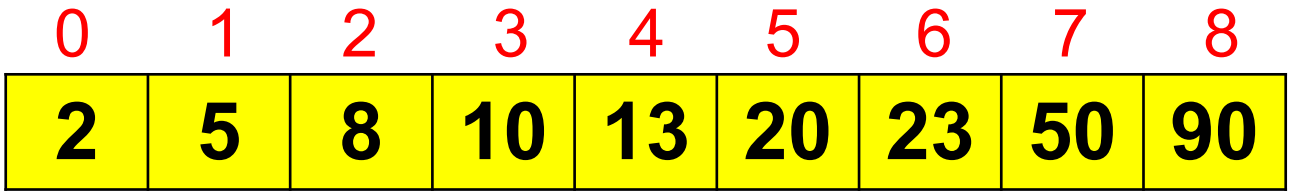
$n/2^k = 1$

$n = 2^k$

$k = \log_2 n$

Introducción

Python List (ordenada)



operation	Time complexity
search(x)	$O(\log_2 n)$

n
n/2
n/4
n/8
…
$n/2^k = 1$
$n = 2^k$
$k = \log_2 n$

Introducción

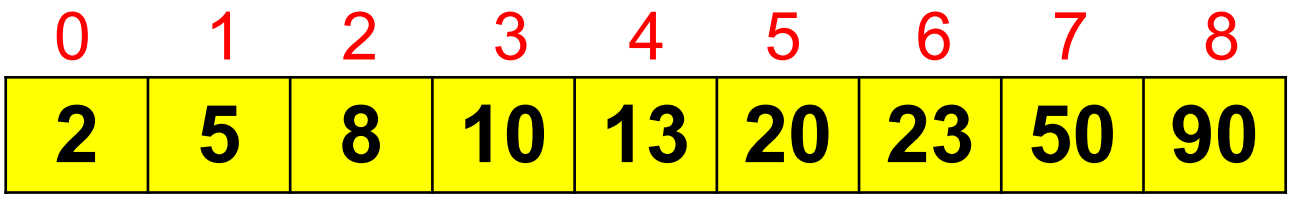
- En este caso, el número máximo de comparaciones para buscar un determinado usuario de Facebook sería:

- n=1 billion (10^8) => k=$\log^2(10^8)$ = 18.42 comparaciones

  - 18.42 x 10^{-6} segundos = 0.00001 segundos << 100 segundos

Introducción

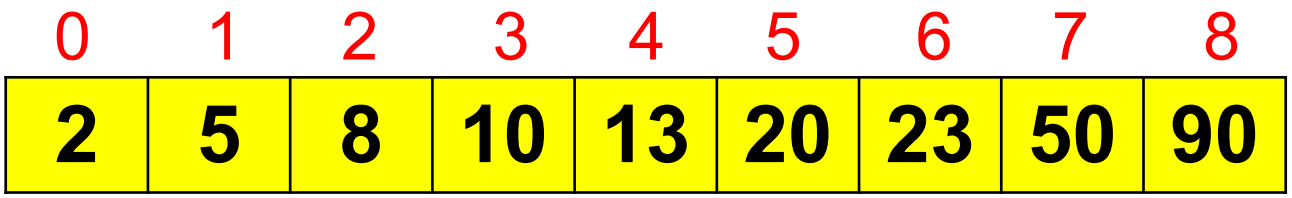
Python List (sorted)



operation	Time complexity
search(x)	$O(\log_2 n)$
insert(x)	
remove(x)	

Introducción

Python List (sorted)



operation	Time complexity
search(x)	$O(\log_2 n)$
insert(x)	O(n)
remove(x)	O(n)

Introducción

- Objetivo: proponer una estructura con complejidad logarítmica en las tres operaciones.

operation	**Time complexity**
search(x)	$O(\log_2 n)$
insert(x)	$O(\log_2 n)$
remove(x)	$O(\log_2 n)$

Fuente: Bogdan Rosu Creative,CC BY 4.0, Iconfinder.

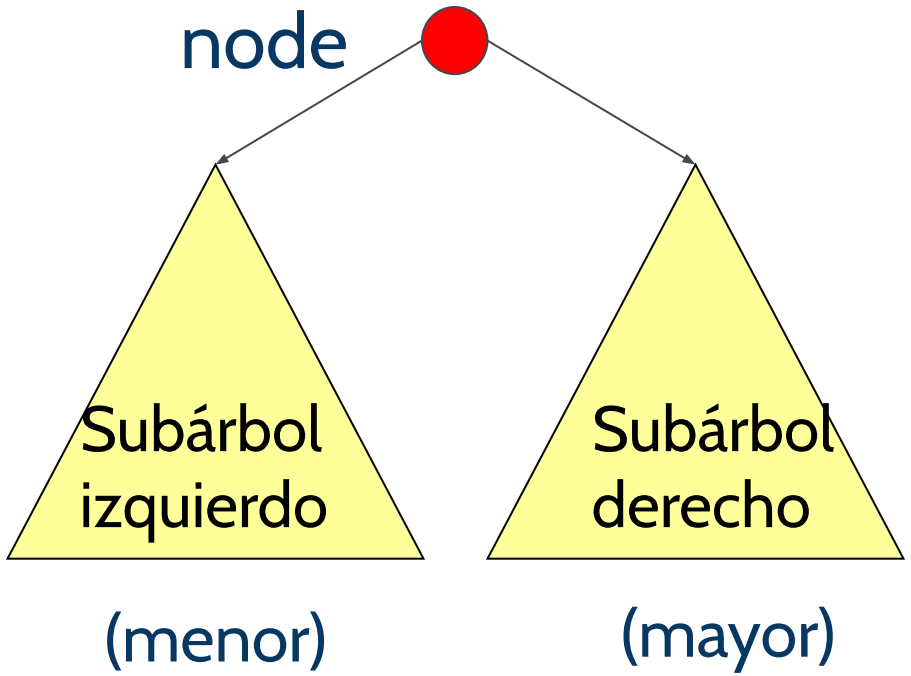
Árboles Binarios de Búsqueda

- Un **ABB** es un **árbol binario** que cumple la siguiente condición:
  - *Para todo nodo, todos los elementos de su subárbol izquierdo son menores que el elemento del nodo, y a su vez el elemento del nodo es menor que todos los elementos del subárbol derecho.*

Árboles Binarios de Búsqueda

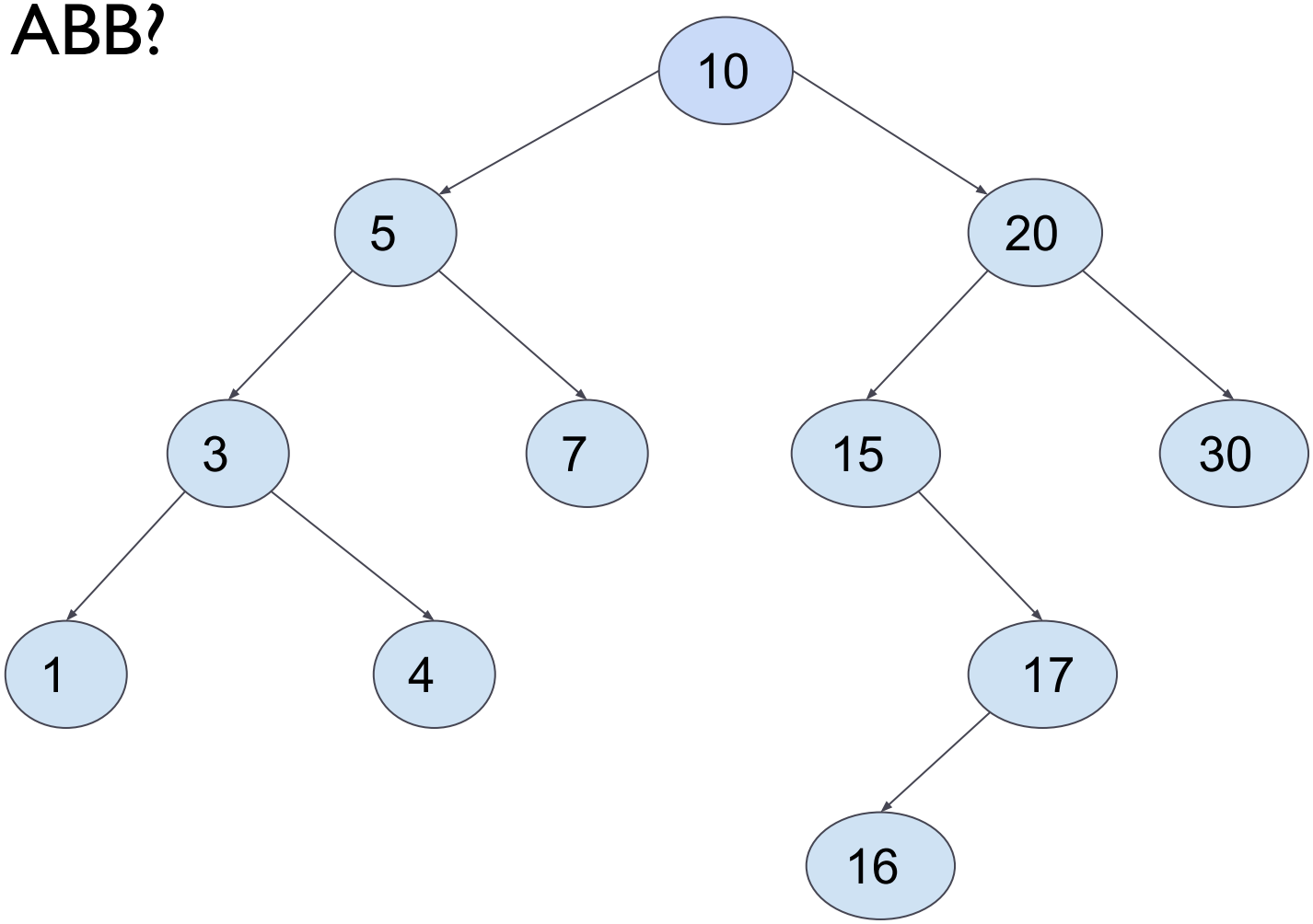
Para todo nodo se debe cumplir:

elementos de node.left < node.elem, y
node.elem < elementos de node.right



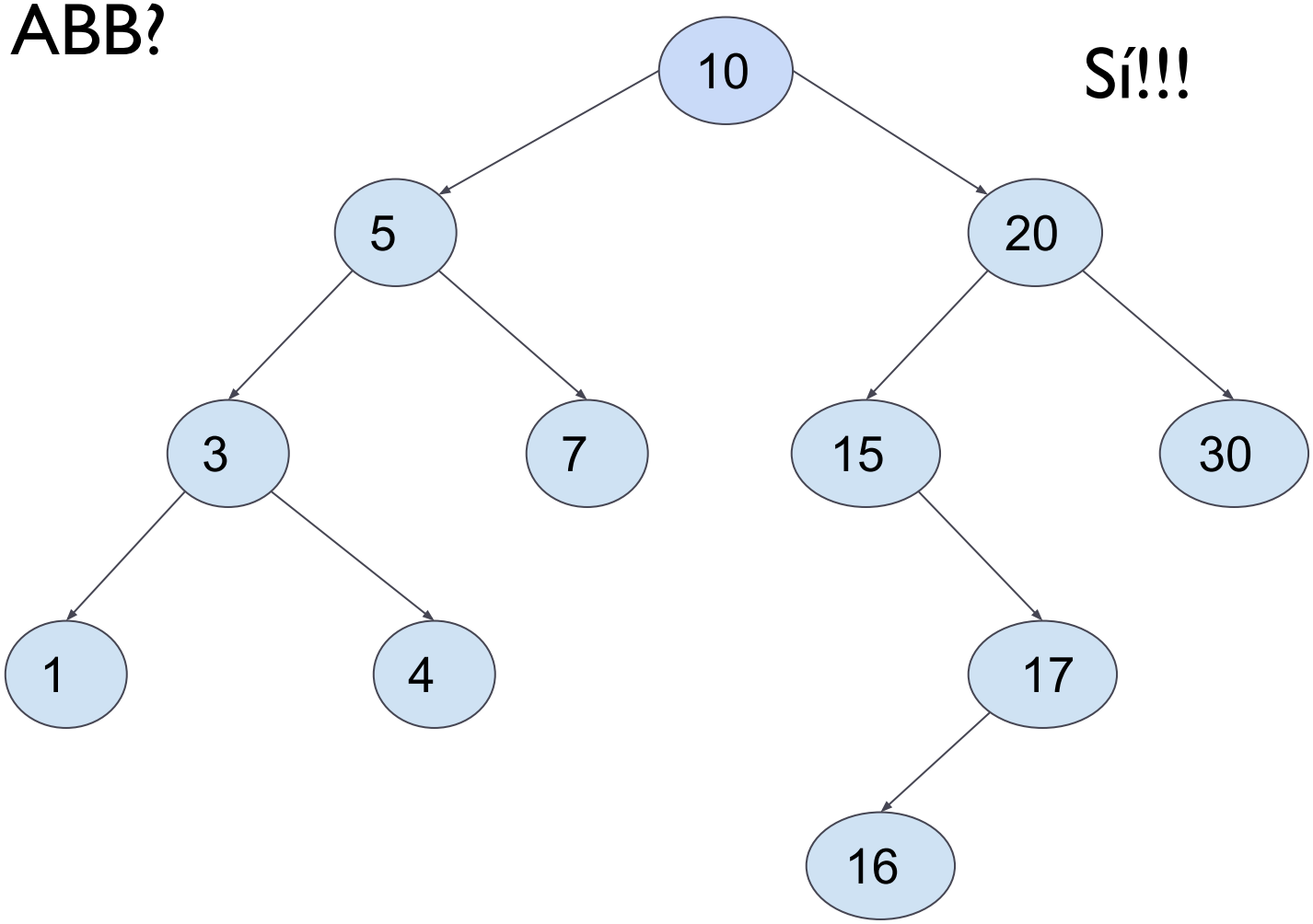
Árboles Binarios de Búsqueda

Es un ABB?



Árboles Binarios de Búsqueda

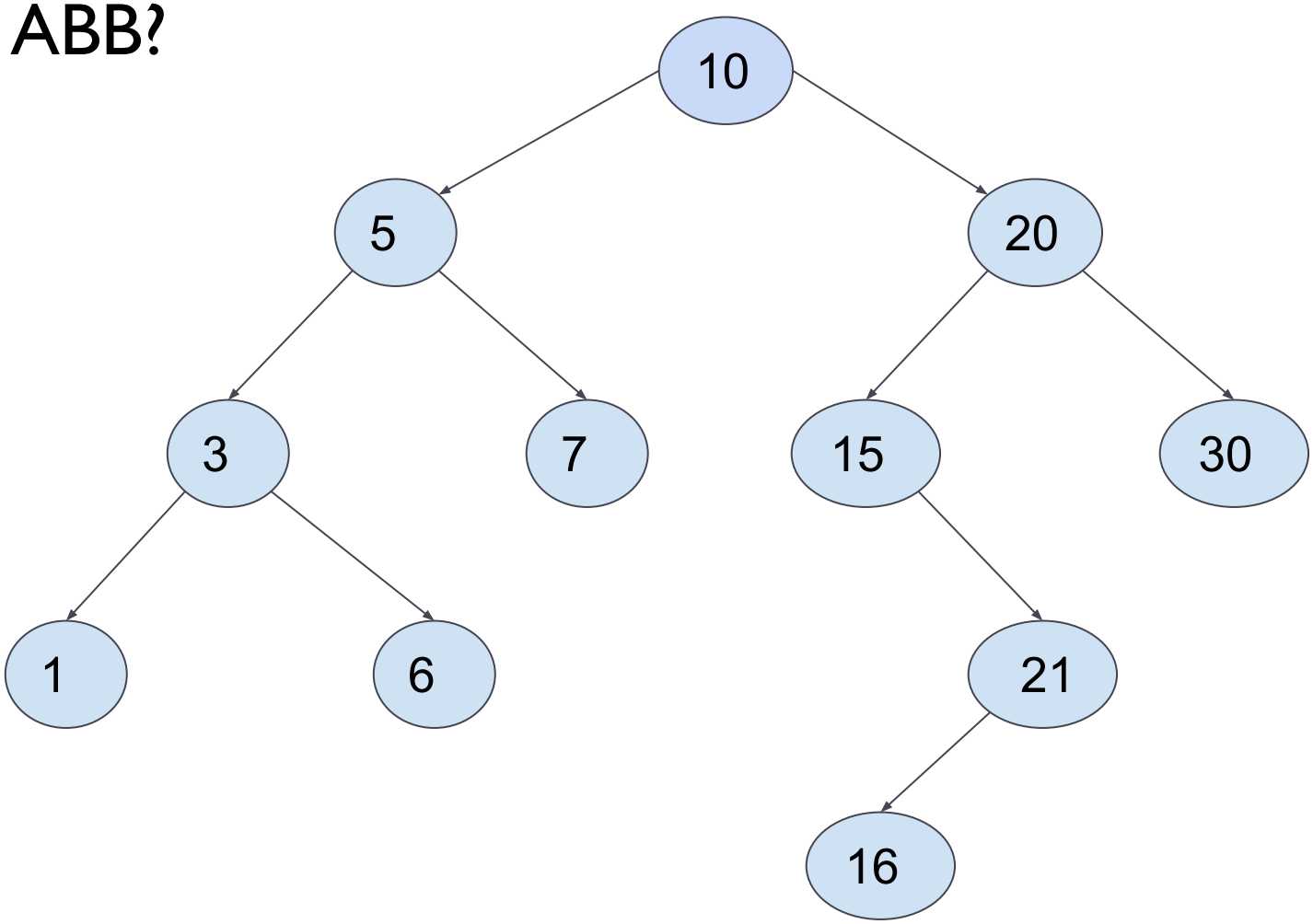
Es un ABB?



Sí!!!

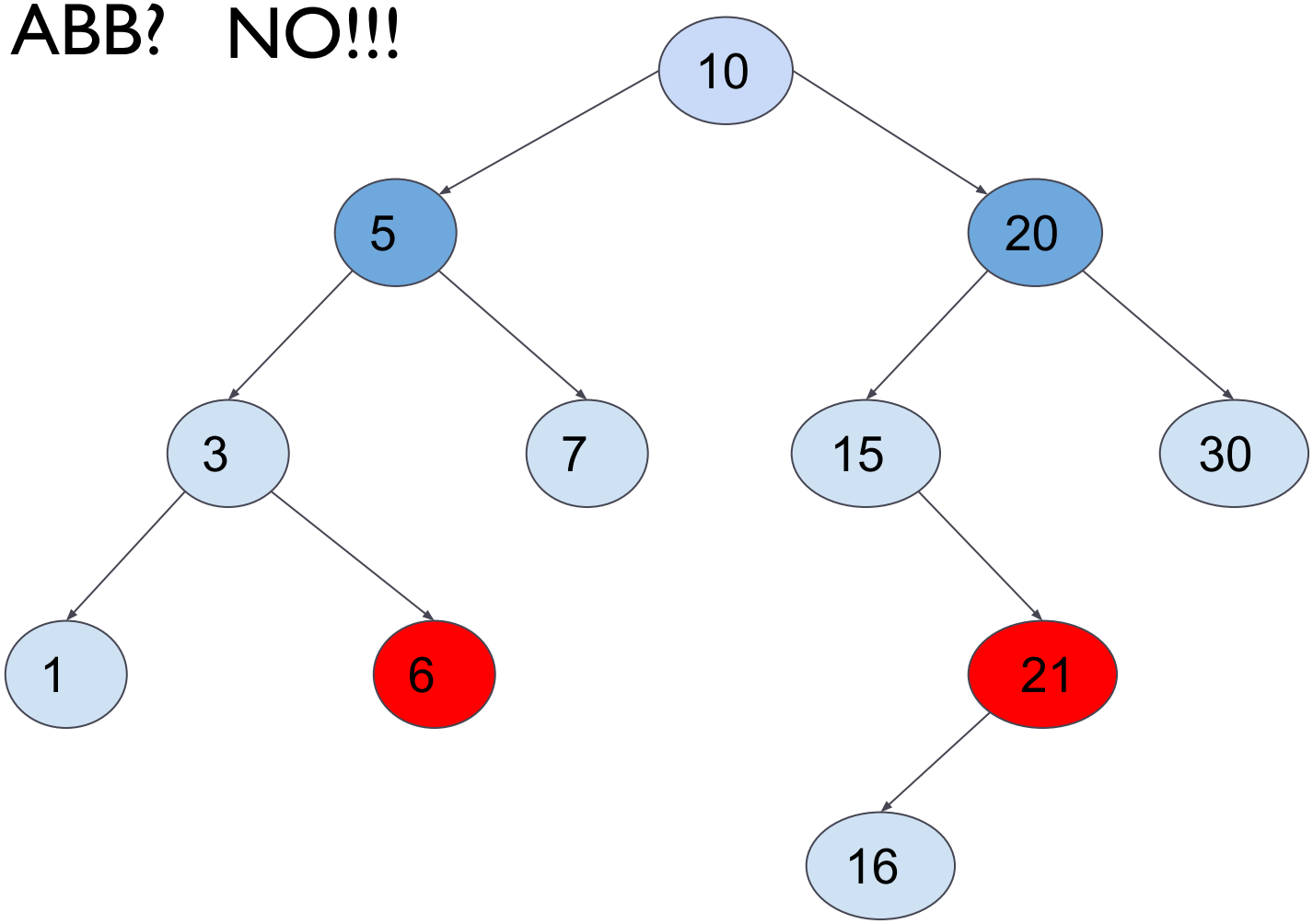
Árboles Binarios de Búsqueda

Es un ABB?



Árboles Binarios de Búsqueda

Es un ABB? NO!!!



TAD Árbol Binario de Búsqueda

Definición: Árbol Binario que cumple la siguiente propiedad:
Para cualquier nodo, *node*, todos los elementos de su subárbol izquierdo (*node.left*) son más pequeños que su elemento (*node.elem*), y a su vez, este elemento (*node.elem*) es más pequeño que todos los elementos de su subárbol derecho (*node.right*).

TAD Árbol Binario de Búsqueda

Operaciones del TAD ABB:

- **search**(elem): recibe el valor de un elemento y devuelve el nodo que contiene el elemento. Si el elemento no existe en el árbol, se muestra un mensaje y devuelve None.
- **insert**(elem): recibe el valor de un elemento e inserta un nuevo nodo en el árbol (por tanto, el árbol es modificado). Si el elemento ya existía, simplemente se muestra un mensaje informando que elementos duplicados no están permitidos.

TAD Árbol Binario de Búsqueda

Operaciones del TAD ABB:

- **remove**(elem): recibe el valor de un elemento a buscar. Una vez encontrado dicho nodo es eliminado, y el árbol es modificado. Si el elemento no existe, simplemente se muestra un mensaje informando que no existe.

- Como un ABB es un AB, también tendrá todas las operaciones de un AB: size, height, in-order, etc.

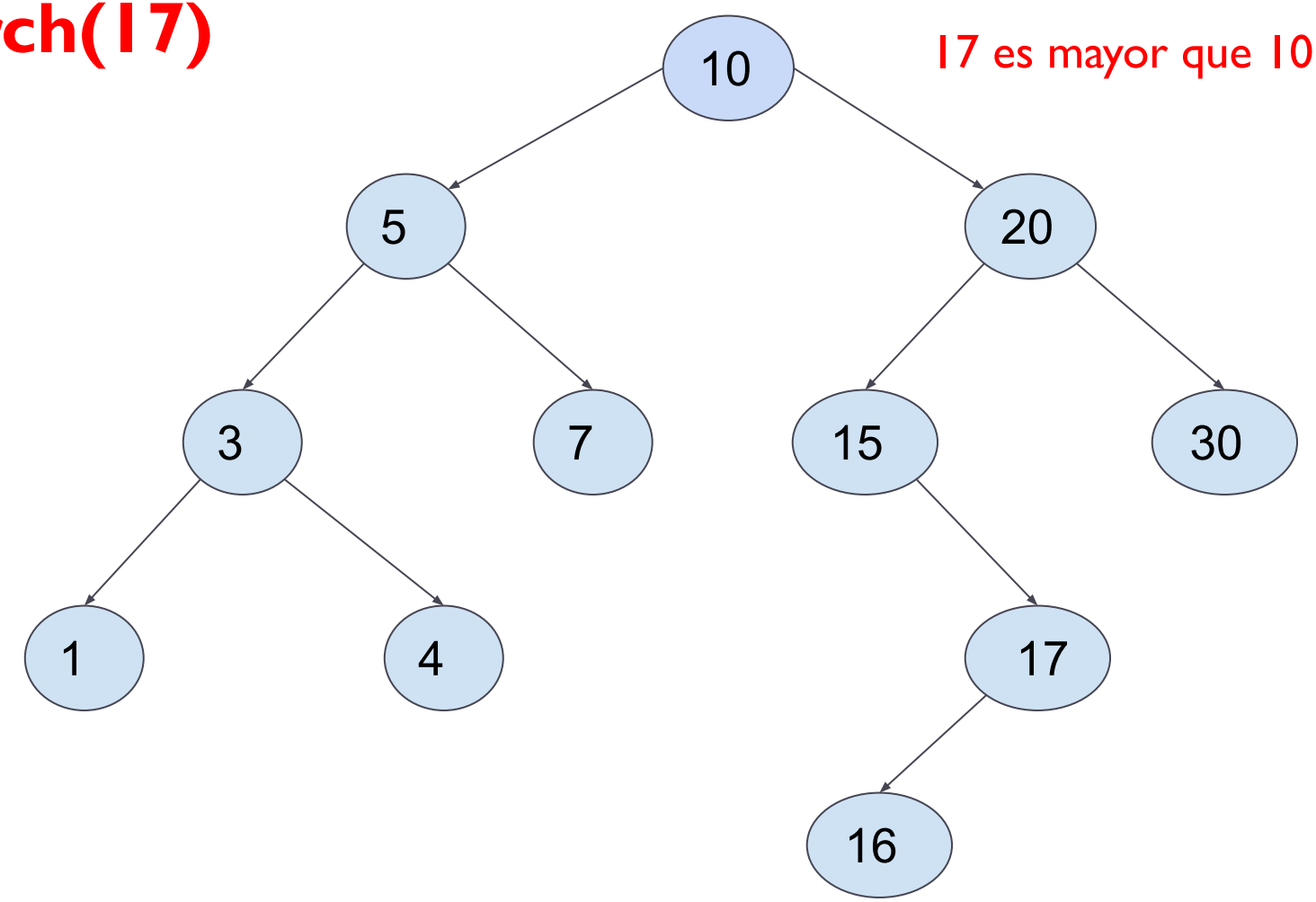
Un herramienta muy útil para ABB!!!

- Puedes practicar creando tus propios árboles binarios de búsqueda las operaciones de búsqueda, inserción y borrado en https://visualgo.net/en/bst

search

Árboles Binarios de Búsqueda-search

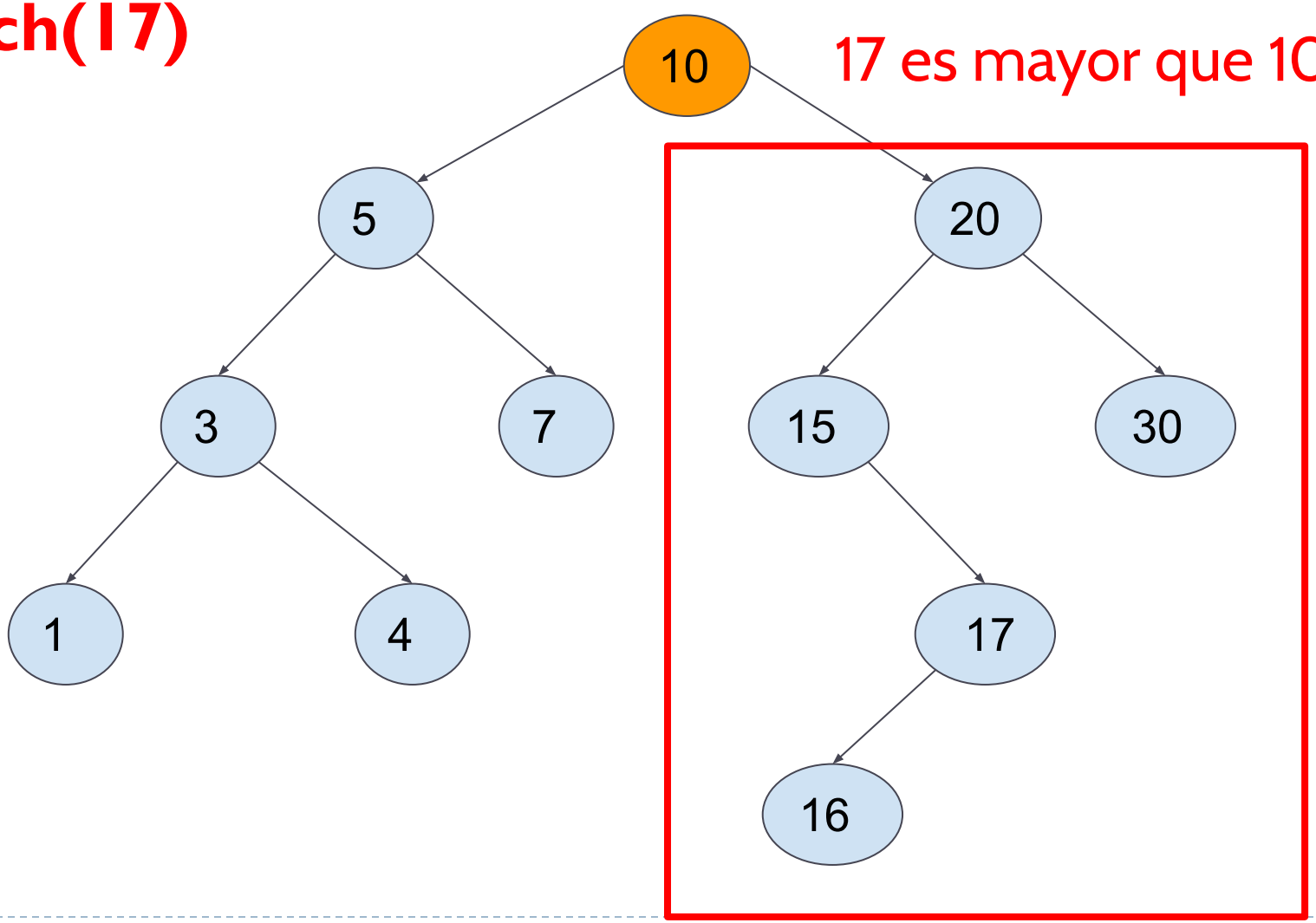
search(17)



17 es mayor que 10

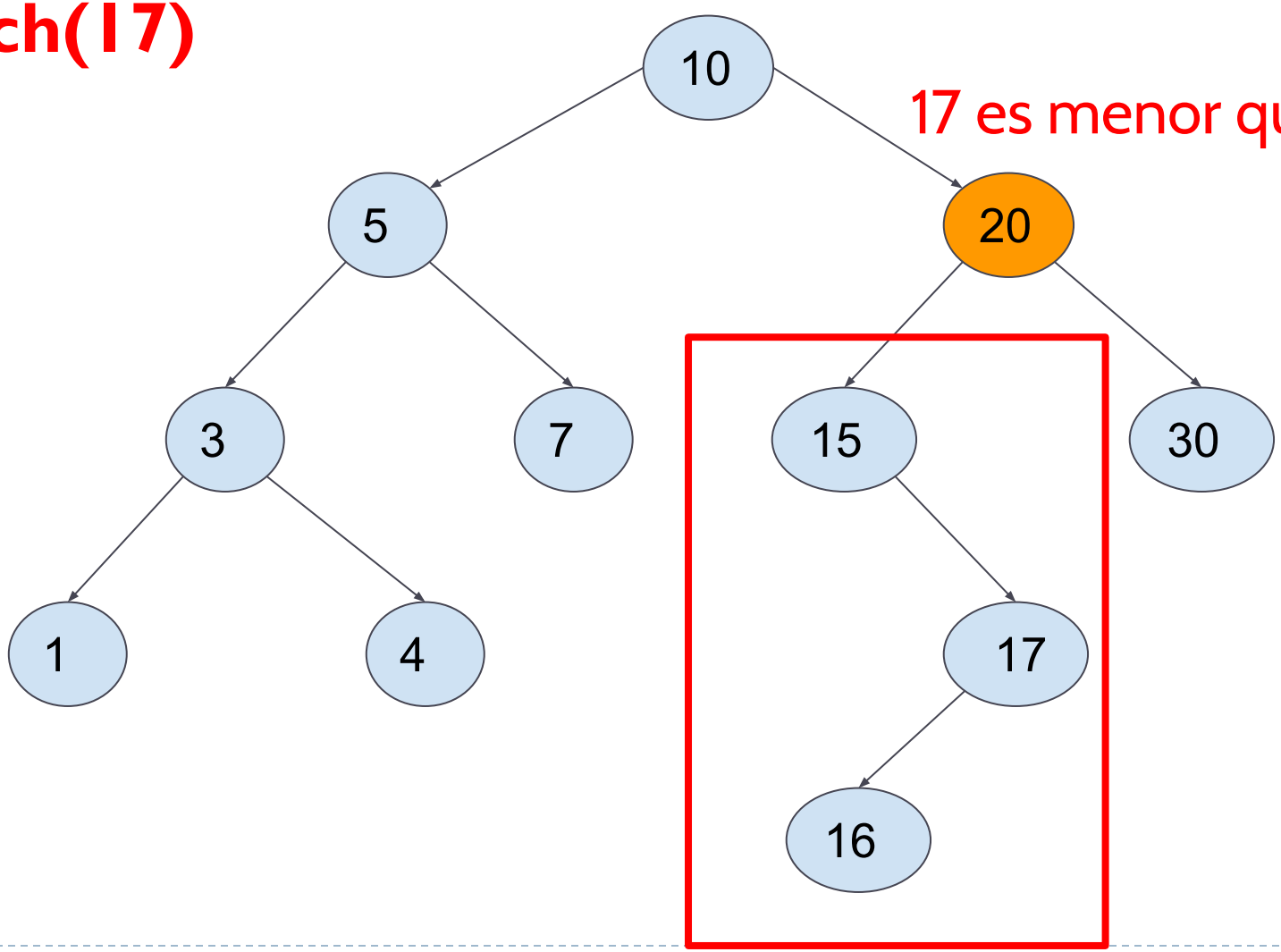
Árboles Binarios de Búsqueda-search

search(17)



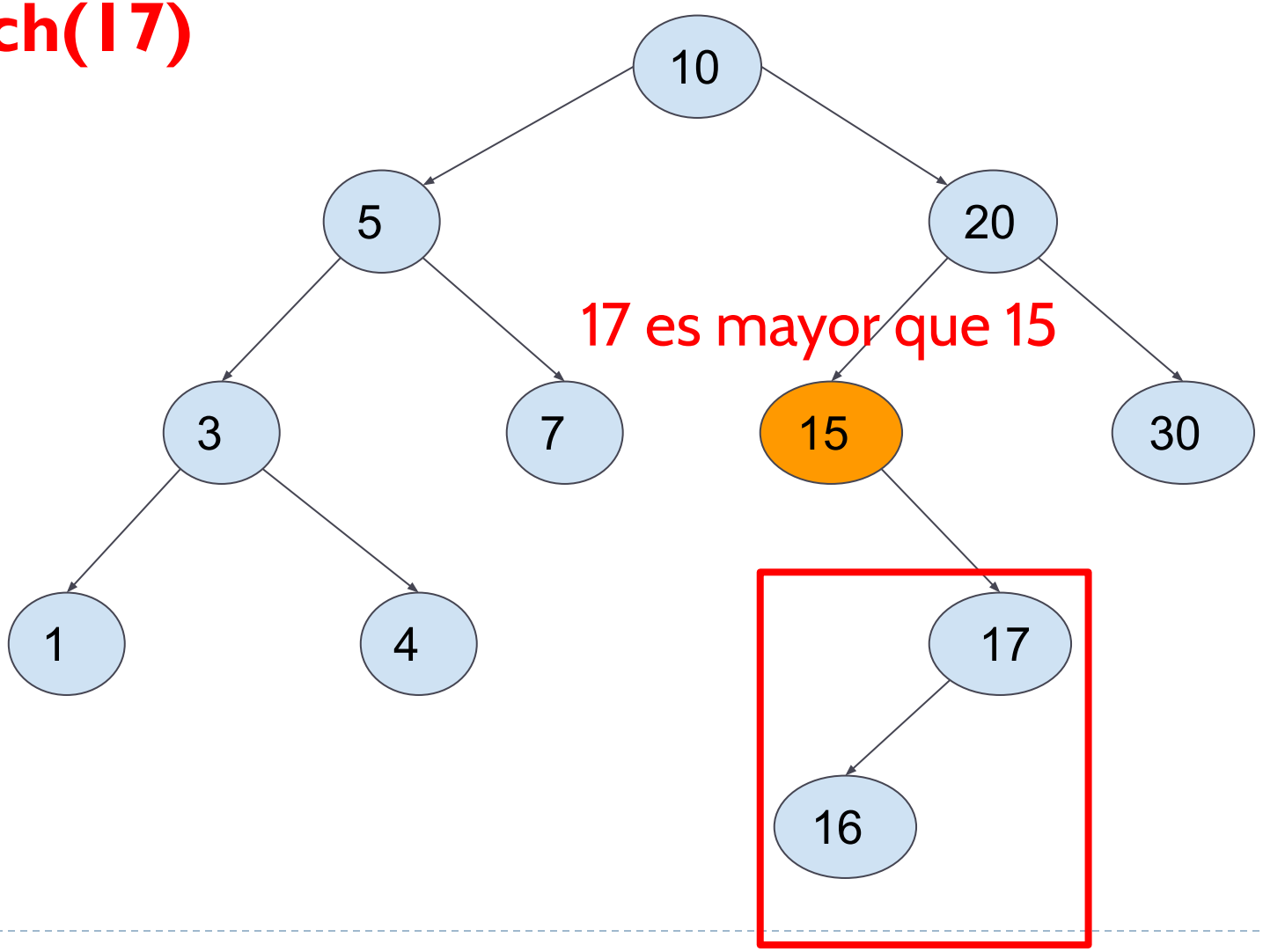
Árboles Binarios de Búsqueda-search

search(17)



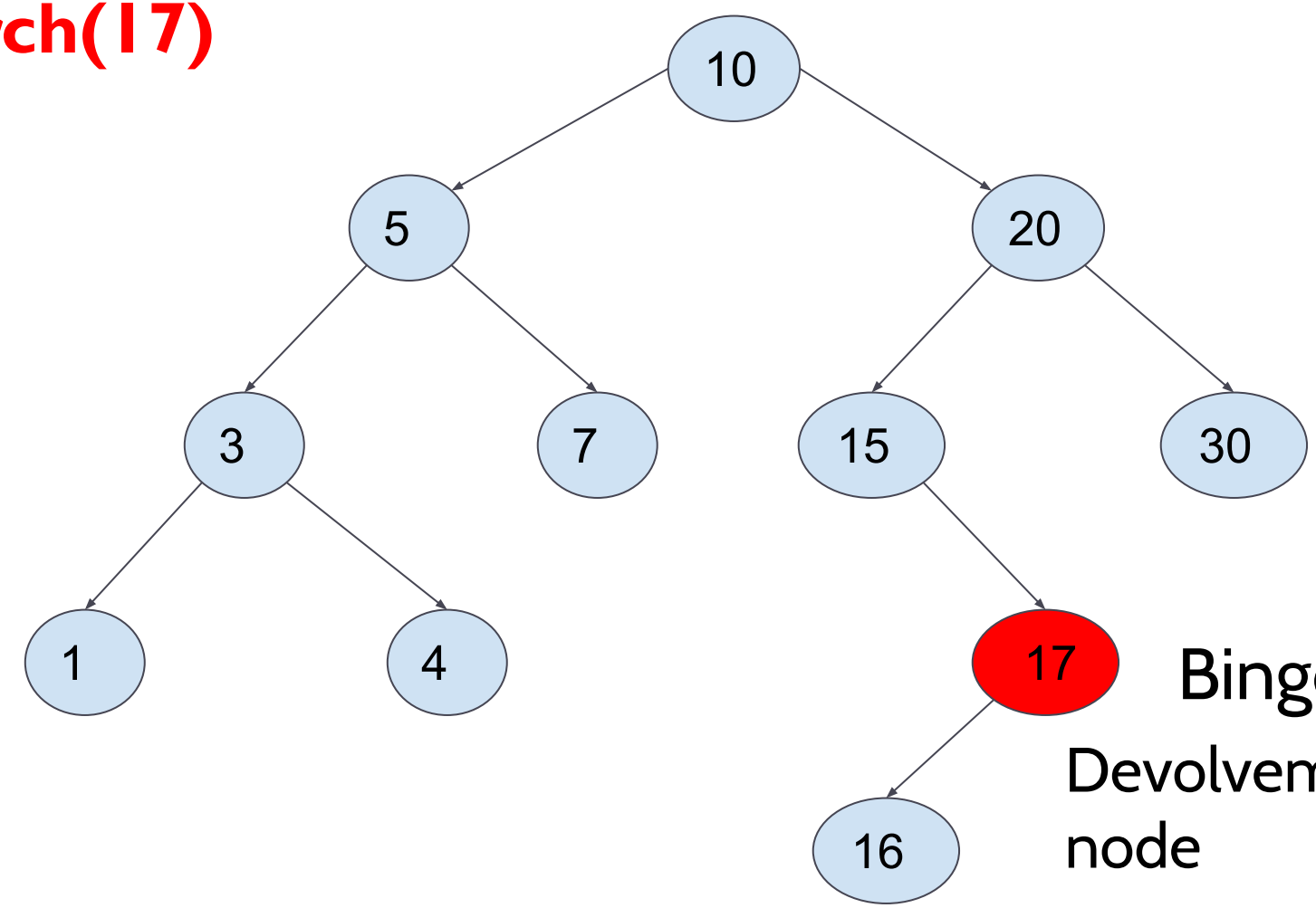
Árboles Binarios de Búsqueda-search

search(17)



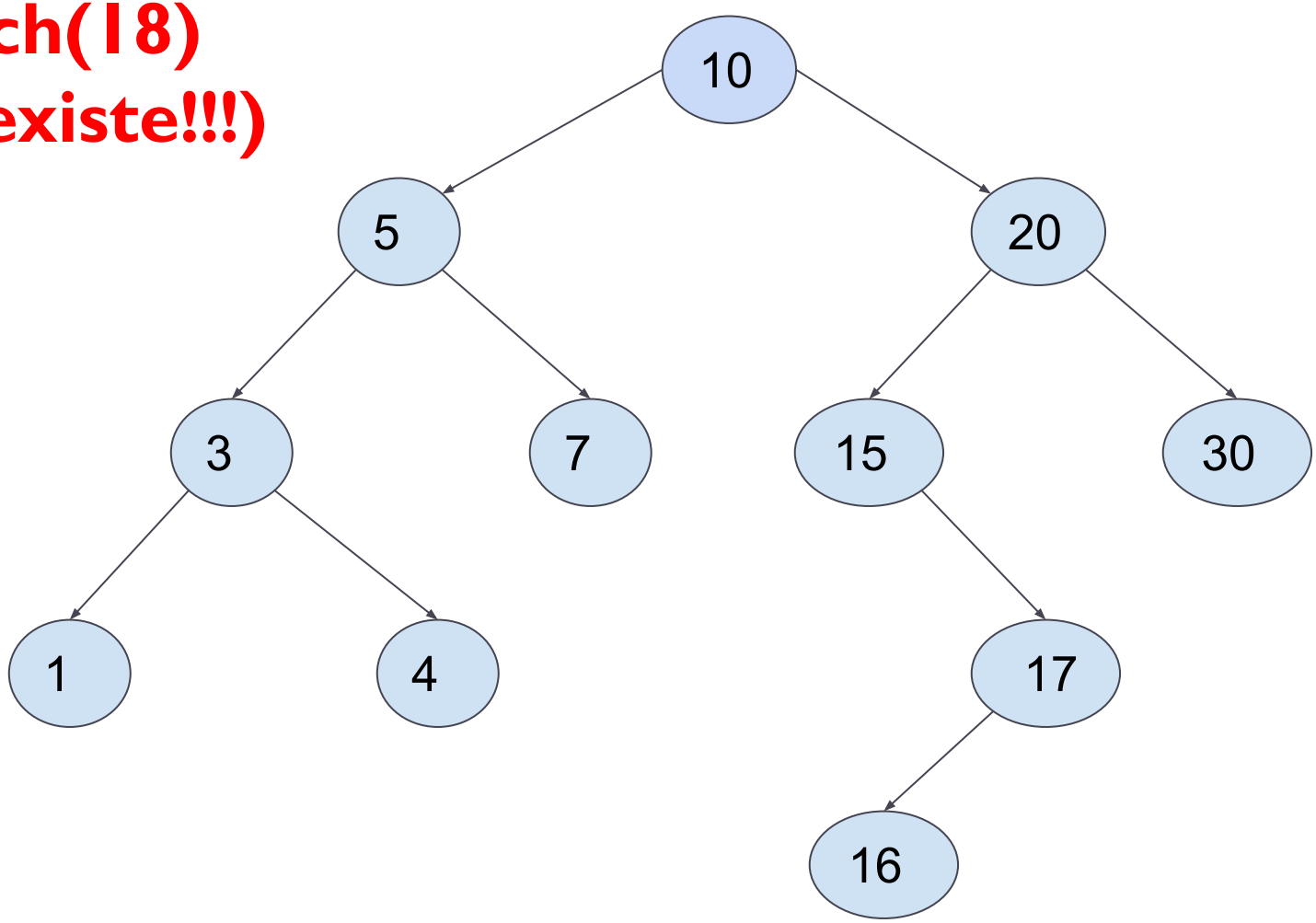
Árboles Binarios de Búsqueda-search

search(17)



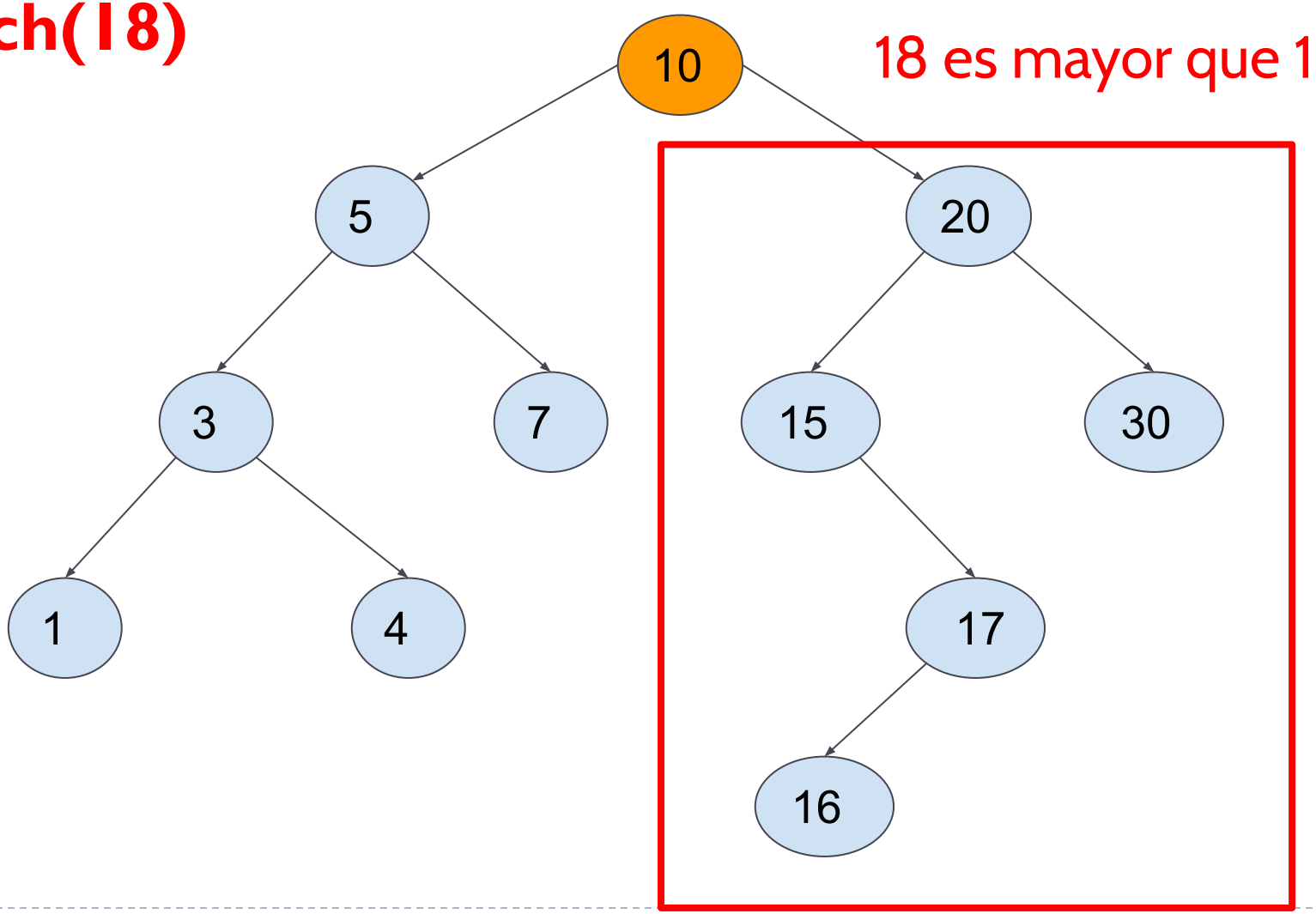
Árboles Binarios de Búsqueda-search

search(18)
(no existe!!!)



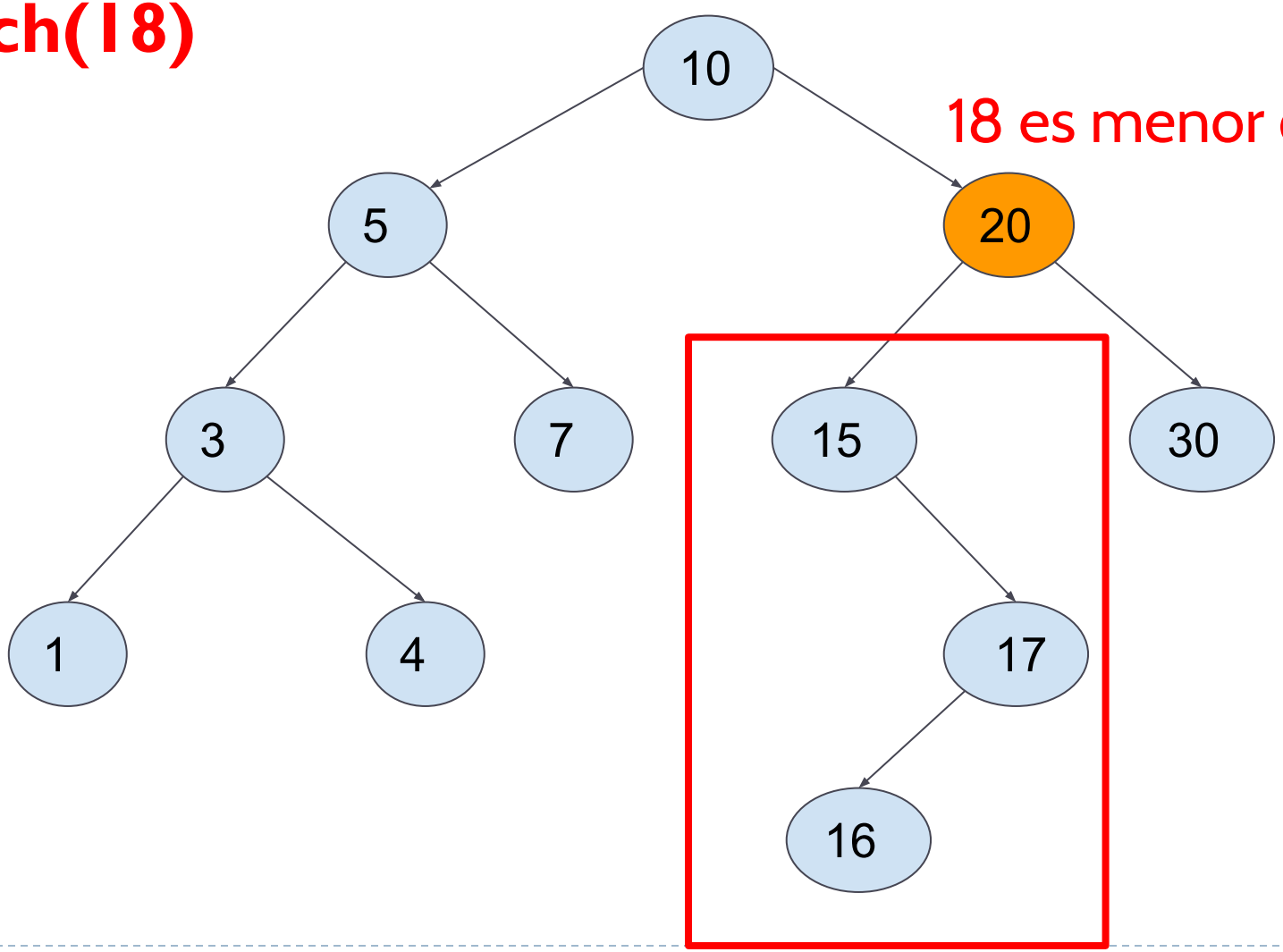
Árboles Binarios de Búsqueda-search

search(18)



Árboles Binarios de Búsqueda-search

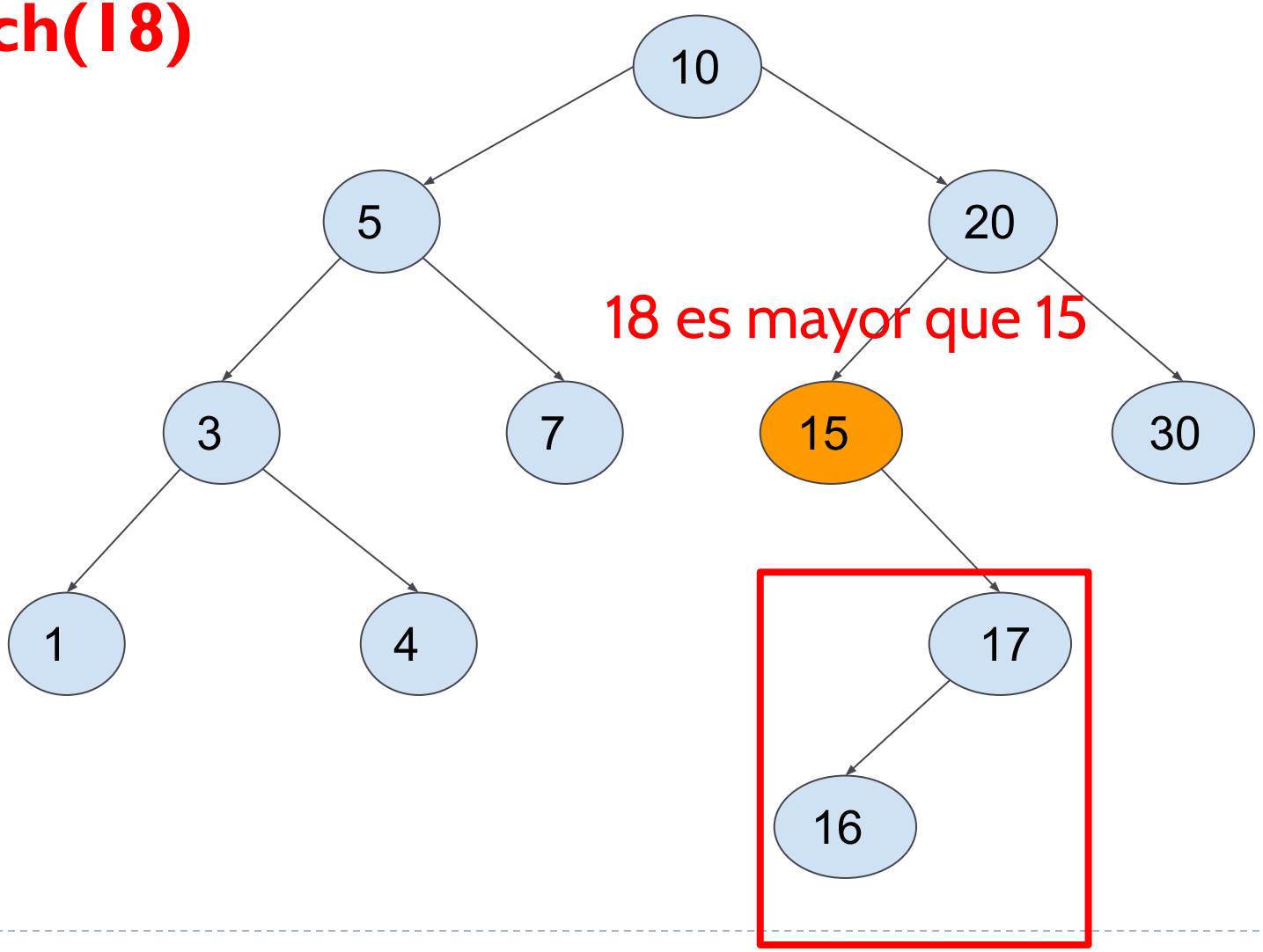
search(18)



18 es menor que 20

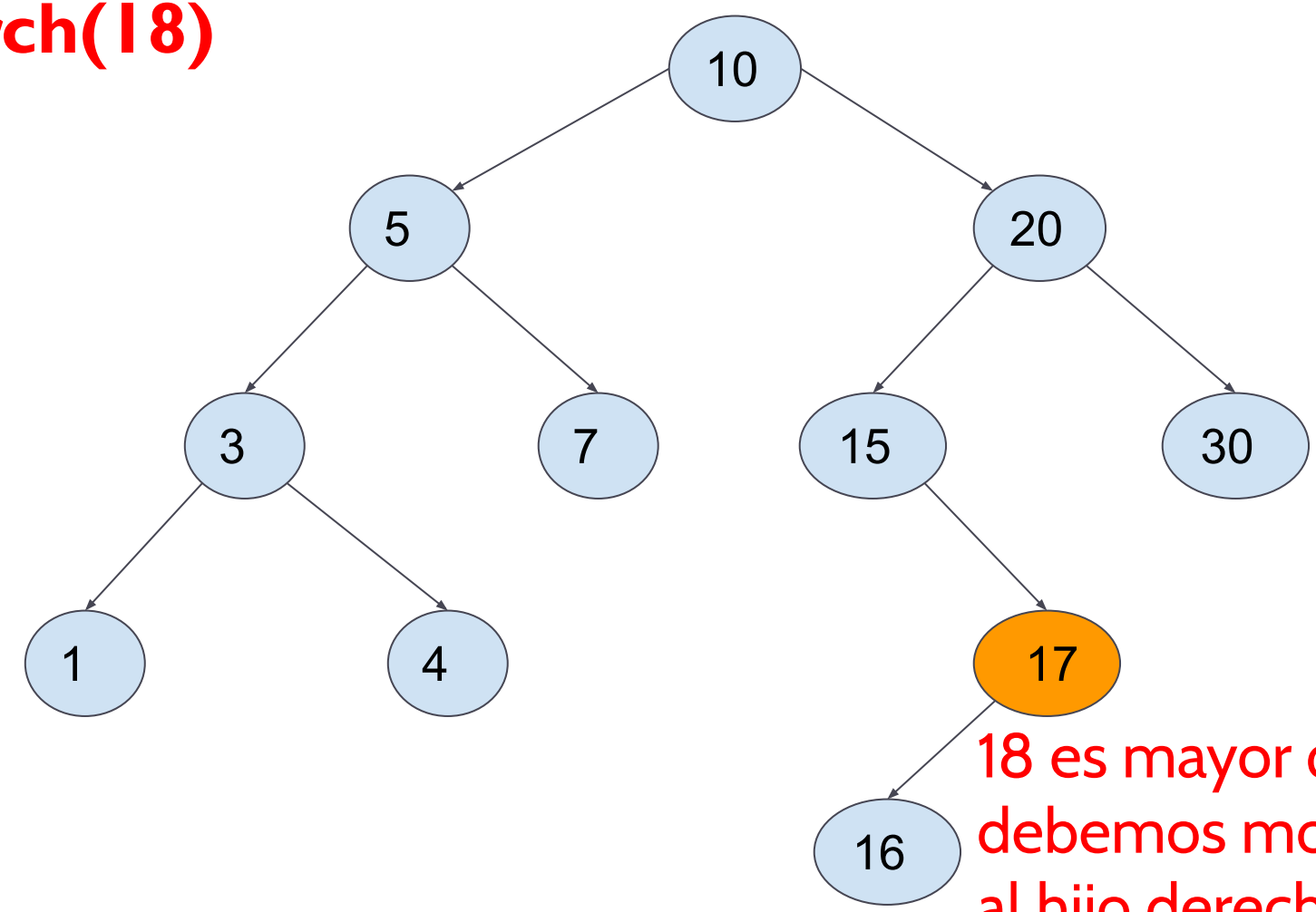
Árboles Binarios de Búsqueda-search

search(18)



Árboles Binarios de Búsqueda-search

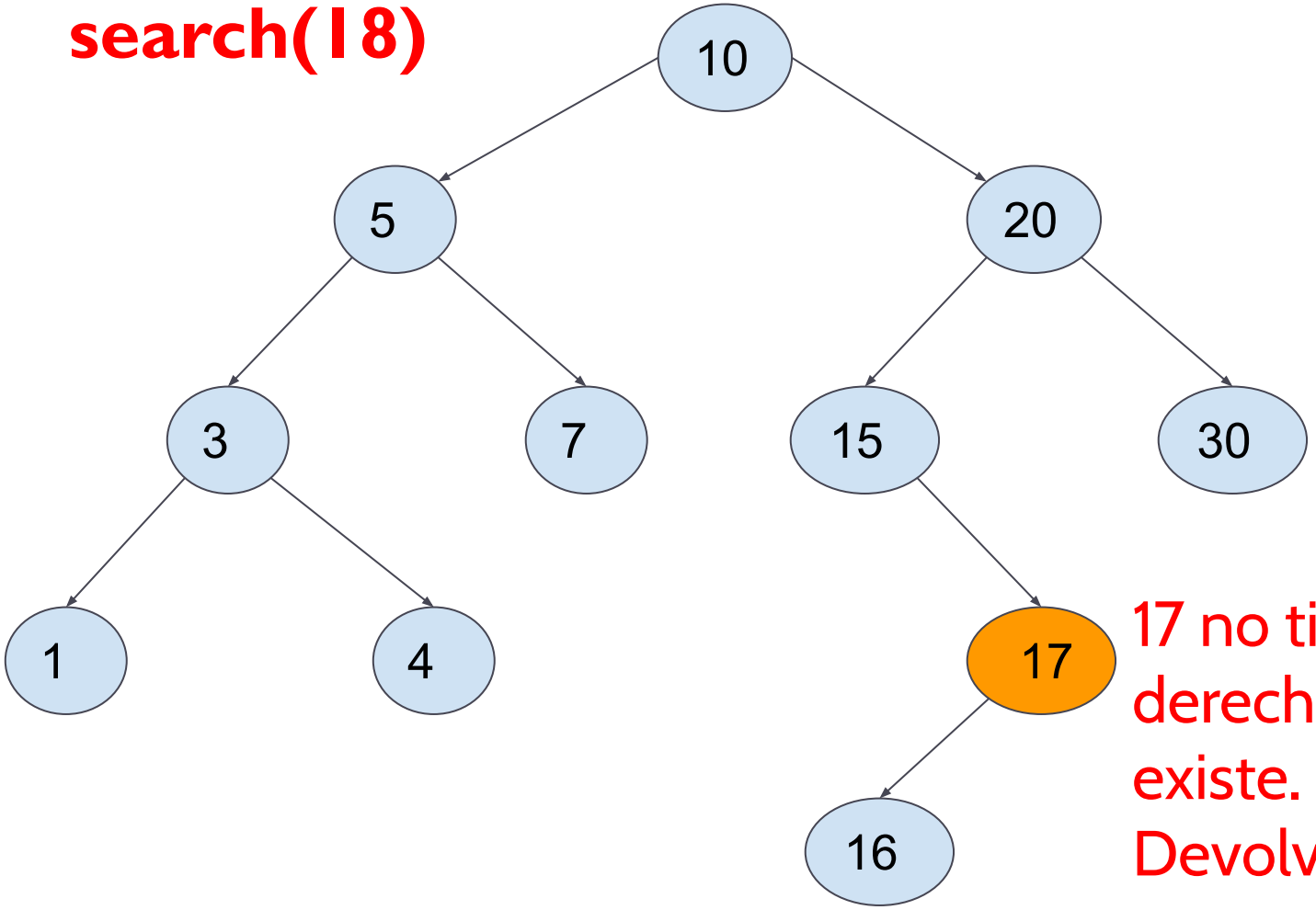
search(18)



18 es mayor que 17
debemos movernos
al hijo derecho de 17

Árboles Binarios de Búsqueda-search

search(18)



17 no tiene hijo derecho!!!. 18 no existe.
Devolvemos None

Implementación search

```
def search(self, elem: object) -> BinaryNode:
    """Returns the node whose elem is elem"""
    return self._search(self._root, elem)

def _search(self, node: BinaryNode, elem: object) -> BinaryNode:
    """Recursive function"""
    ...
```

Implementación search

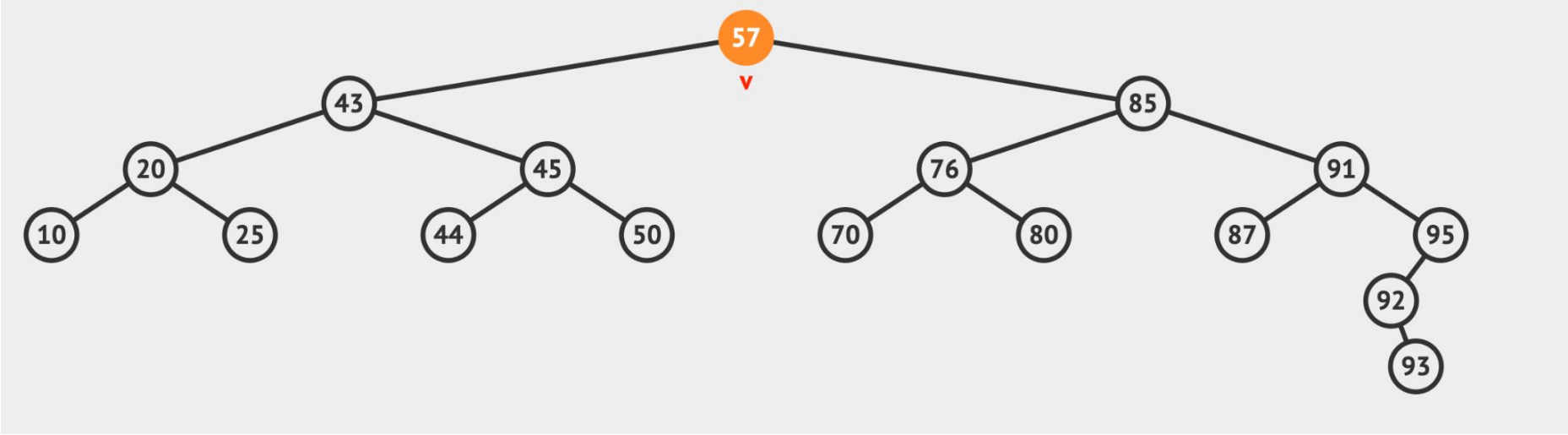
```
def search(self, elem: object) -> BinaryNode:
    """Returns the node whose elem is elem"""
    return self._search(self._root, elem)

def _search(self, node: BinaryNode, elem: object) -> BinaryNode:
    """Recursive function"""
    if node is None or node.elem == elem:
        return node
    elif elem < node.elem:
        return self._search(node.left, elem)
    elif elem > node.elem:
        return self._search(node.right, elem)
```

Implementación search (iterativo)

```
def searchit(self, elem: object) -> BinaryNode:
    """iterative function"""
    node = self._root
    while node:
        if node.elem == elem:
            # we have found it!!! we can return it and leave the function
            return node

        if elem < node.elem:
            node = node.left
        else:
            node = node.right
    return node
```

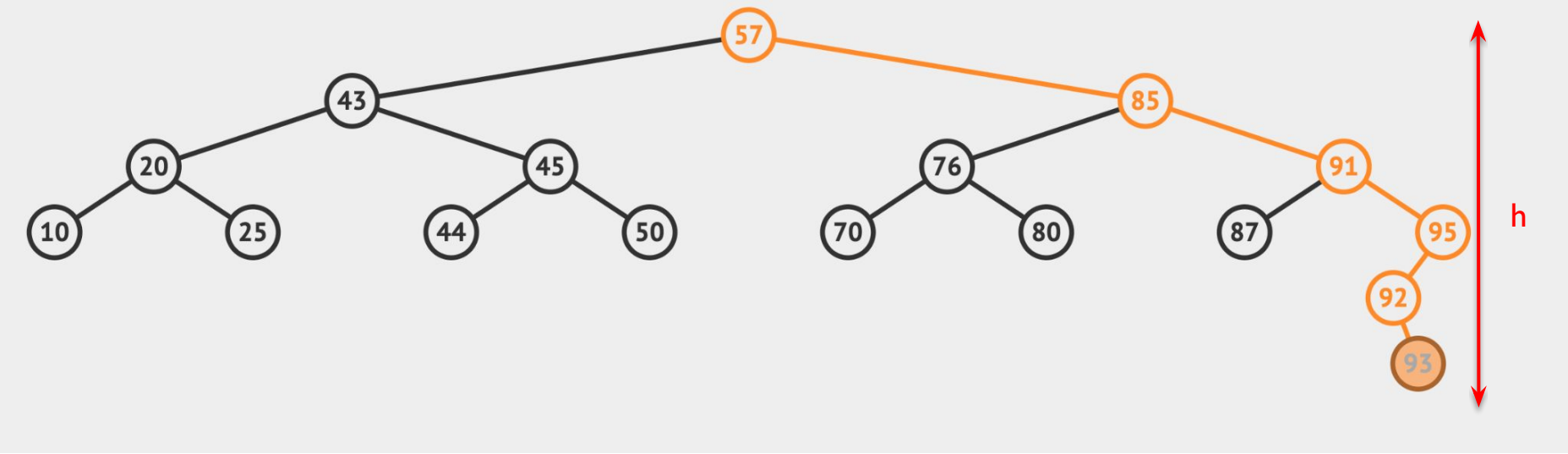
Complejidad temporal search

En la función search(), ¿cuál es el peor caso?



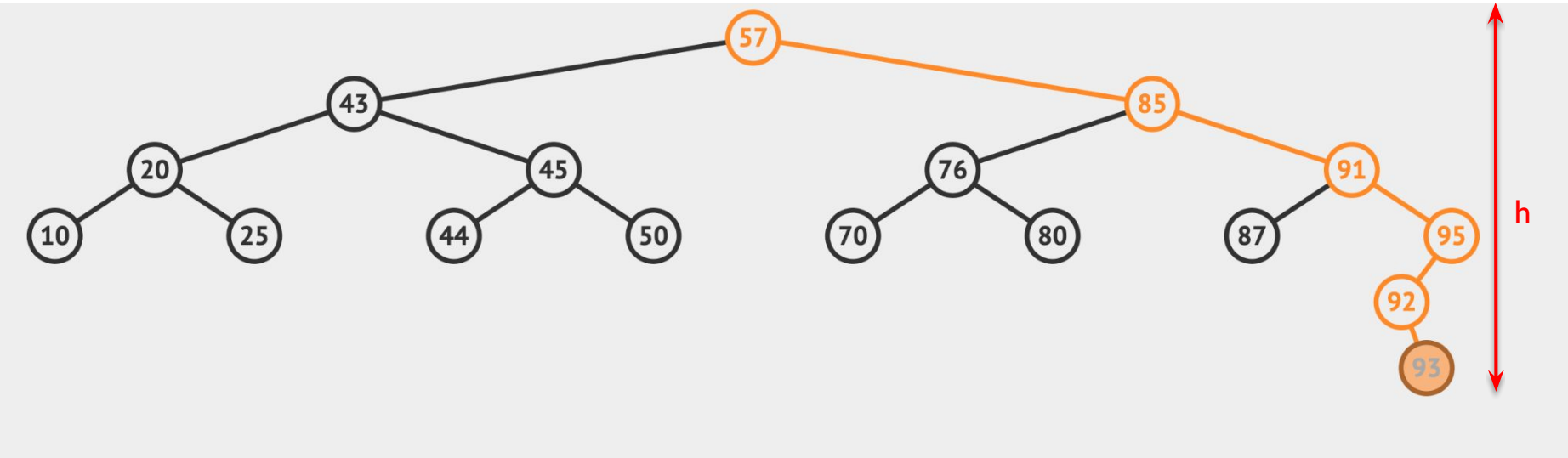
Complejidad temporal search

El peor caso es cuando buscamos 93, porque está en la hoja de la rama más larga del árbol. La búsqueda de 94 también es un caso peor, porque tendríamos que llegar hasta 93 para saber que no podemos continuar la búsqueda y que el elemento no existe.



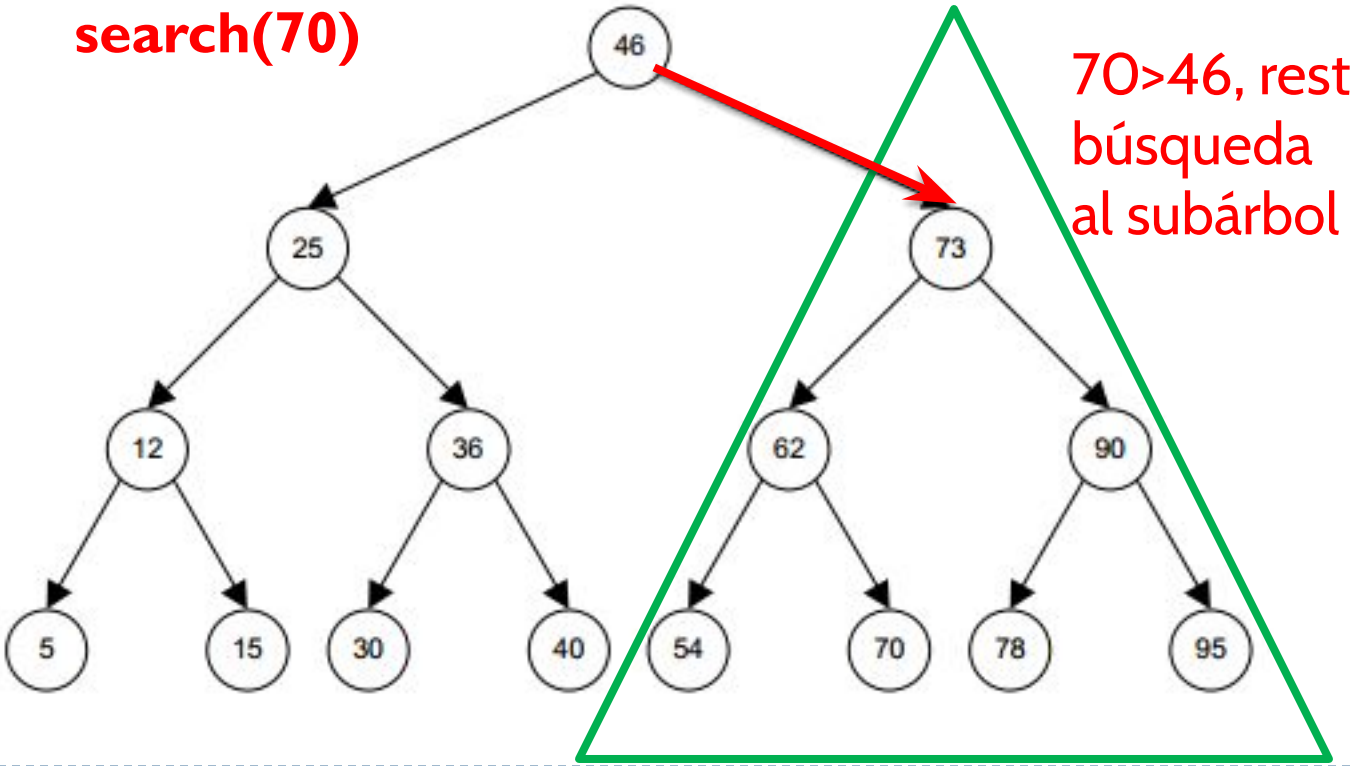
Complejidad temporal search

La longitud de la rama más larga de un árbol, ¿qué es? Es la altura, por tanto, en el peor de los casos, la complejidad de search será O(h), donde h es la altura del árbol.



Complejidad temporal search

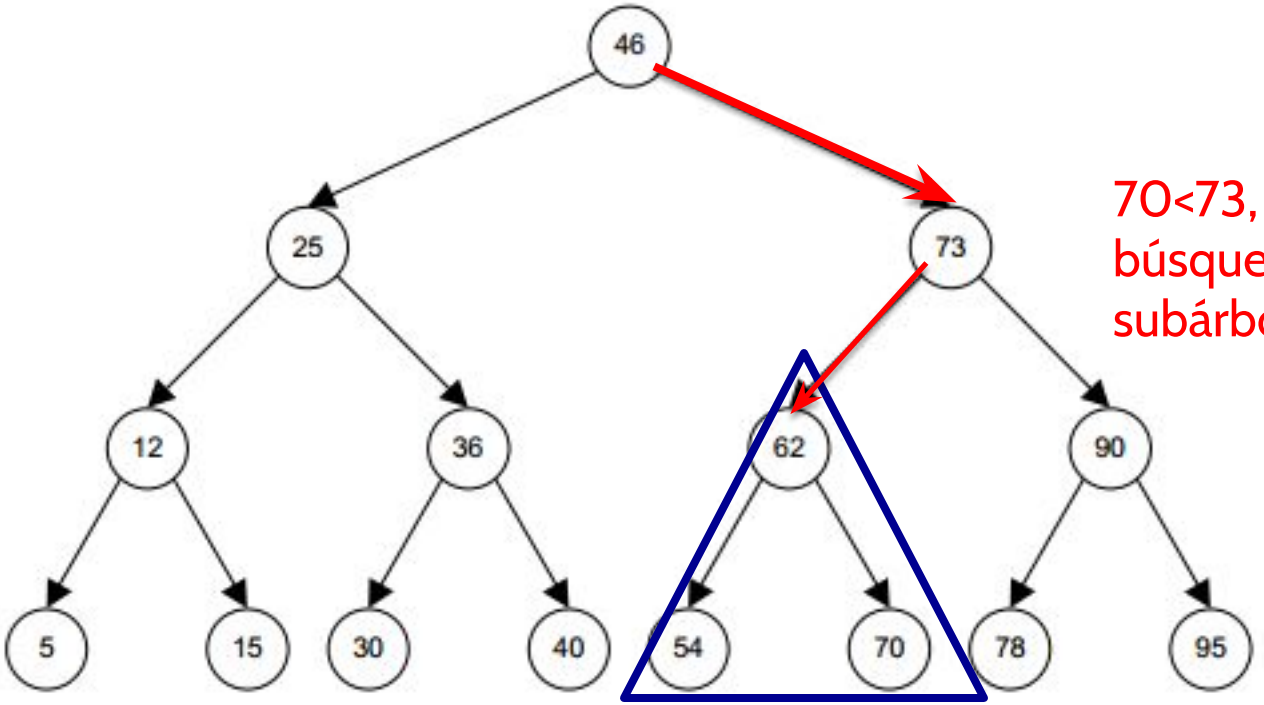
En un ABB, ¿qué ocurre con el espacio de búsqueda cada vez que hacemos una comparación para buscar un nodo?



Nuestro espacio de búsqueda, ha pasado a ser la mitad

Complejidad temporal search

search(70)



70<73, restringir búsqueda al subárbol izquierdo

n

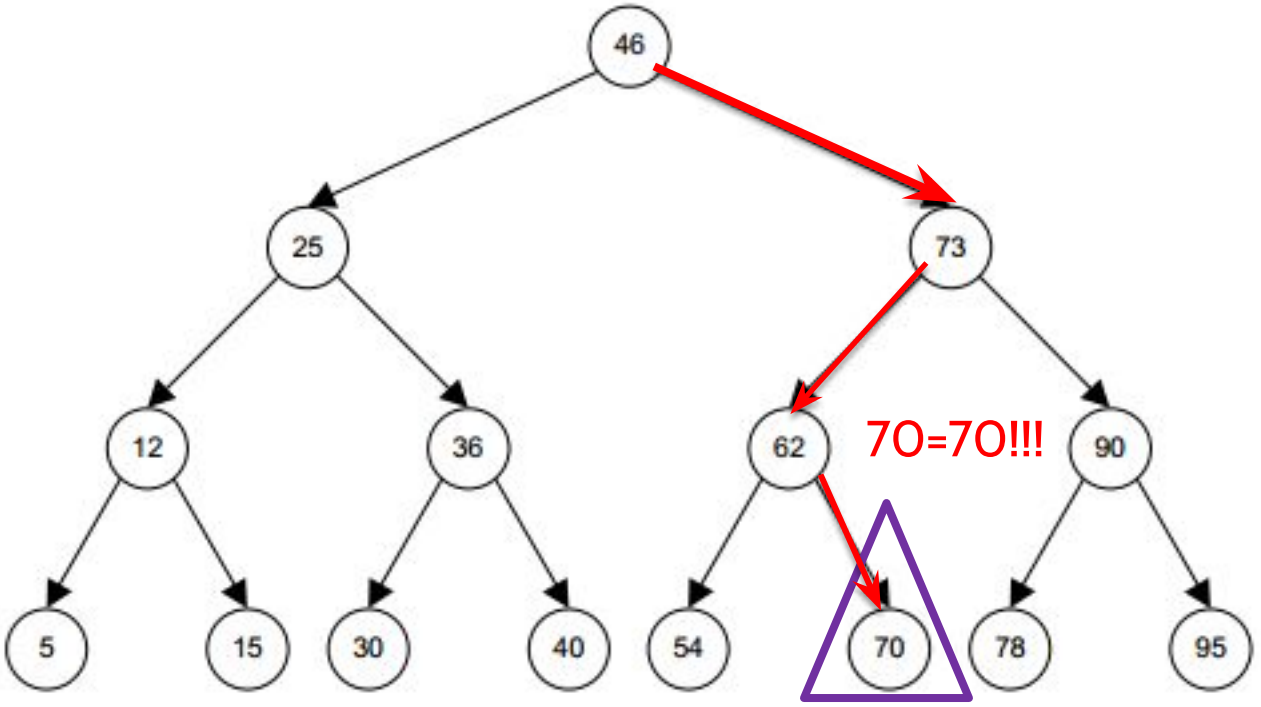
n/2

$n/2^2$

El espacio de búsqueda se vuelve a dividir por la mitad.

Complejidad temporal search

search(70)



Después de 3 pasos, hemos encontrado el nodo buscado

Complejidad temporal search

En cada paso, el espacio de búsqueda se divide por la mitad

$Step_1 = n / 2^1$

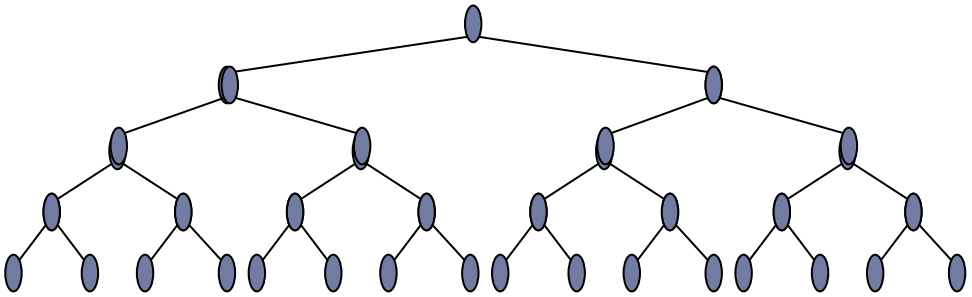
$Step_2 = n / 2^2$

$Step_3 = n / 2^3$

.

.

$Step_k = n / 2^k$ Después de k pasos, encontraremos el nodo o bien llegaremos a None (no existe el nodo). En ambos casos, el espacio de búsqueda ha quedado reducido a 1. Por tanto, podemos considerar la ecuación $n / 2^k = 1$



Complejidad temporal search

$Step_1 = n / 2^1$

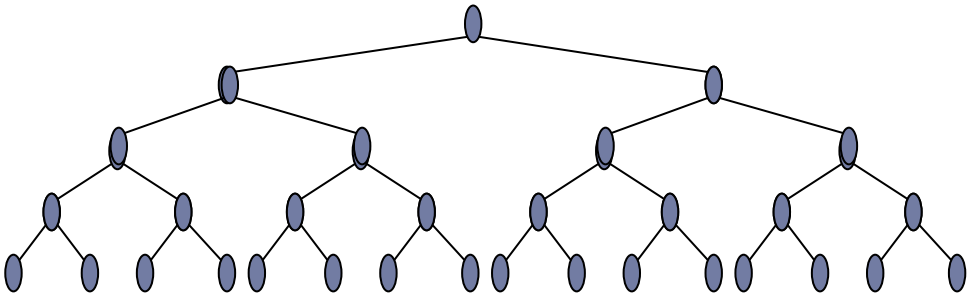
$Step_2 = n / 2^2$

$Step_3 = n / 2^3$

.

.

$Step_k = n / 2^k$



En el peor de los casos, estaremos buscando en la rama más larga, y k=h

$n / 2^h = 1 \Rightarrow n = 2^h$

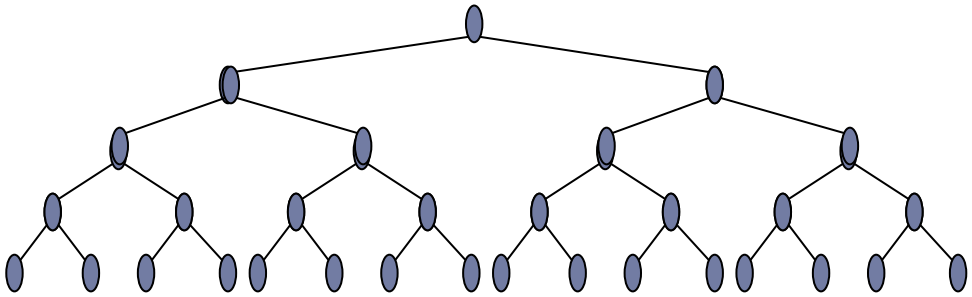
$\mathbf{h = \log_2(n)}$

Complejidad temporal search

Hemos visto que la **complejidad temporal de search es O(h)**, donde **h** es la **altura** del árbol.

También acabamos de demostrar que $h = \log_2(n)$.

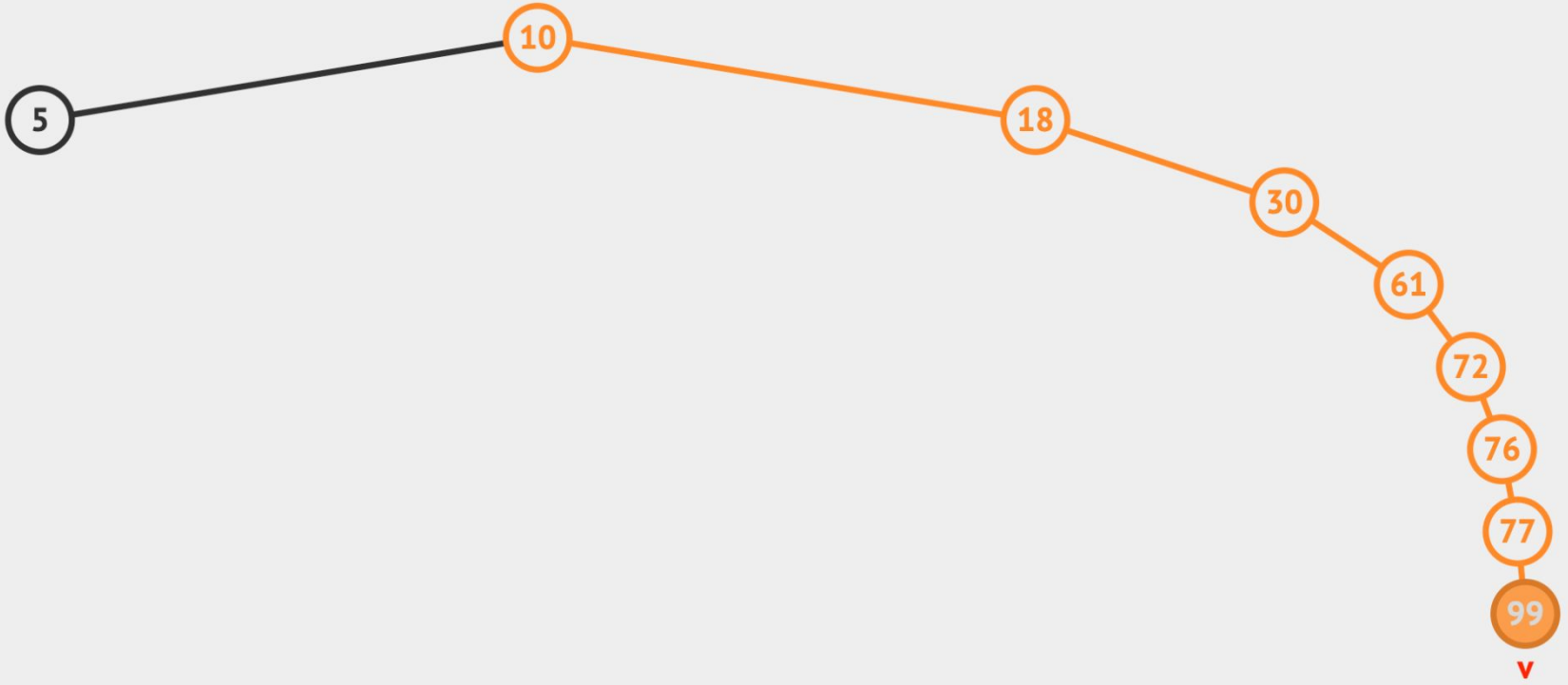
Por tanto, la c**omplejidad temporal de search** es **$O(\log_2(n))$.**



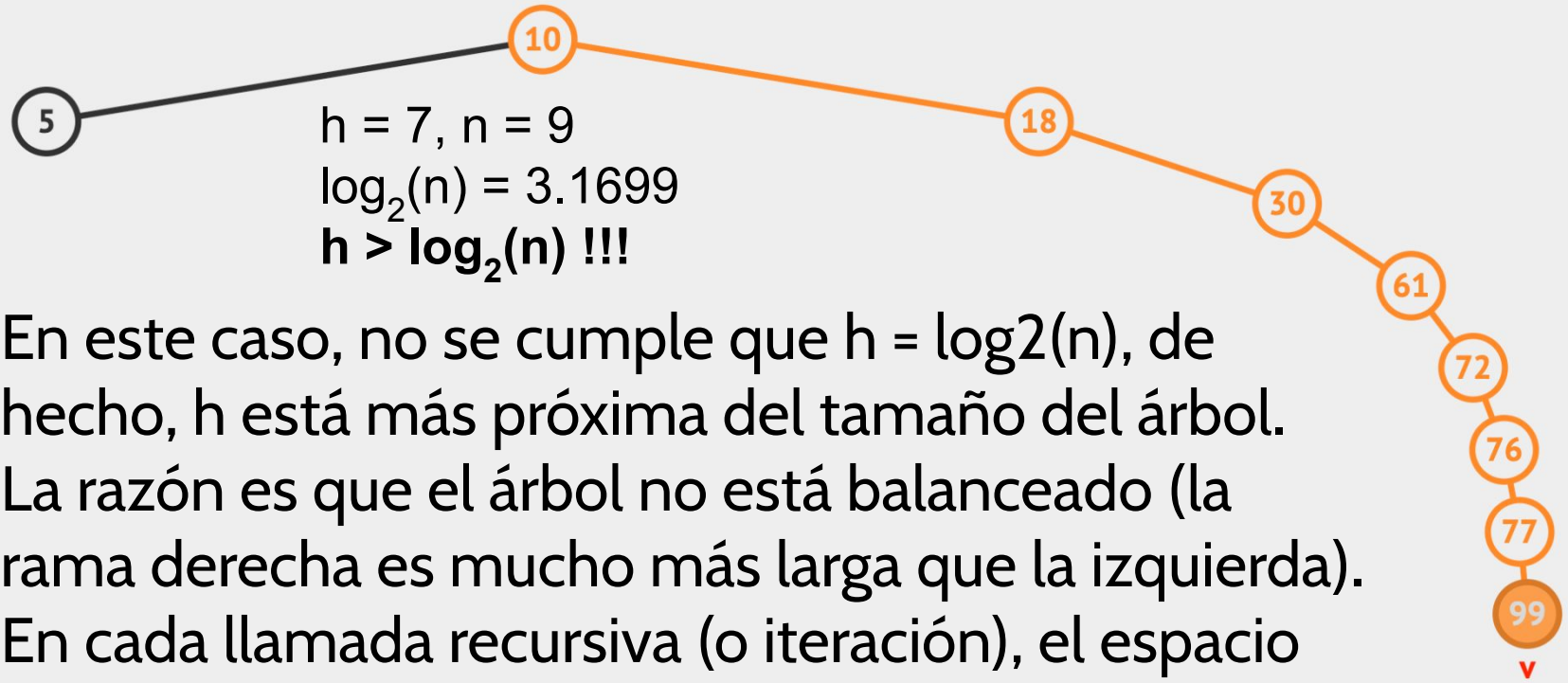
¿Va a ser cierto para cualquier tipo de ABB?

Complejidad temporal search

¿Cuál es la complejidad temporal de search(99) en el siguiente ABB?



Complejidad temporal search



En este caso, no se cumple que h = log2(n), de hecho, h está más próxima del tamaño del árbol. La razón es que el árbol no está balanceado (la rama derecha es mucho más larga que la izquierda). En cada llamada recursiva (o iteración), el espacio de búsqueda no se divide por la mitad.

Complejidad temporal search

En los árboles no balanceados, h no es $\log_2(n)$, en su lugar, h->n
(h se aproxima al tamaño del árbol). Por tanto, la **complejidad temporal de los árboles no balanceados** será:

O(h) = O(n)

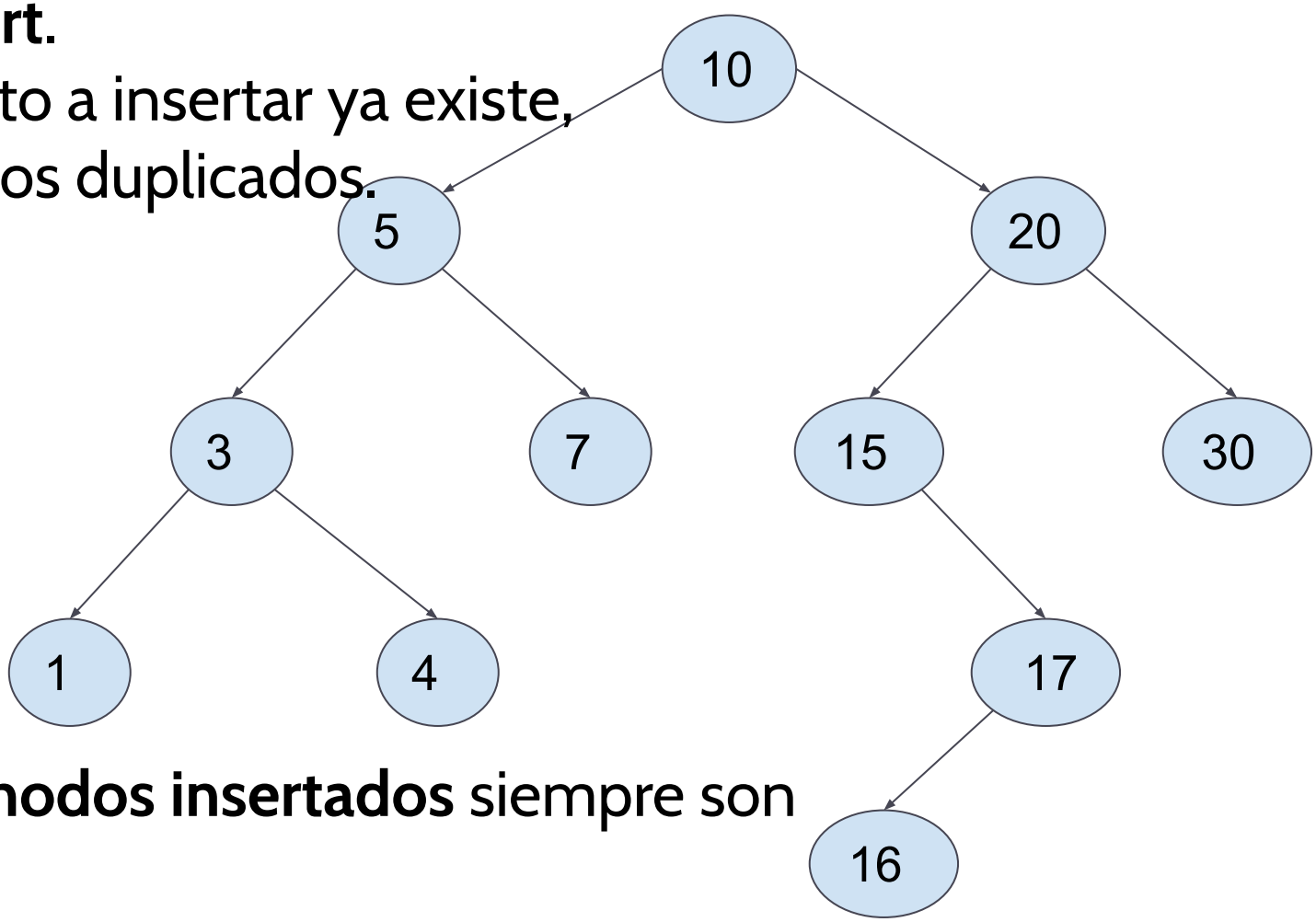
Lo ideal es tener una complejidad logarítmica.

Para ello deberemos mantener los árboles balanceados (para cualquier nodo, se cumple que sus ramas izquierda y derecha tiene una longitud similar ±1)

insert

Árboles Binarios de Búsqueda - insert

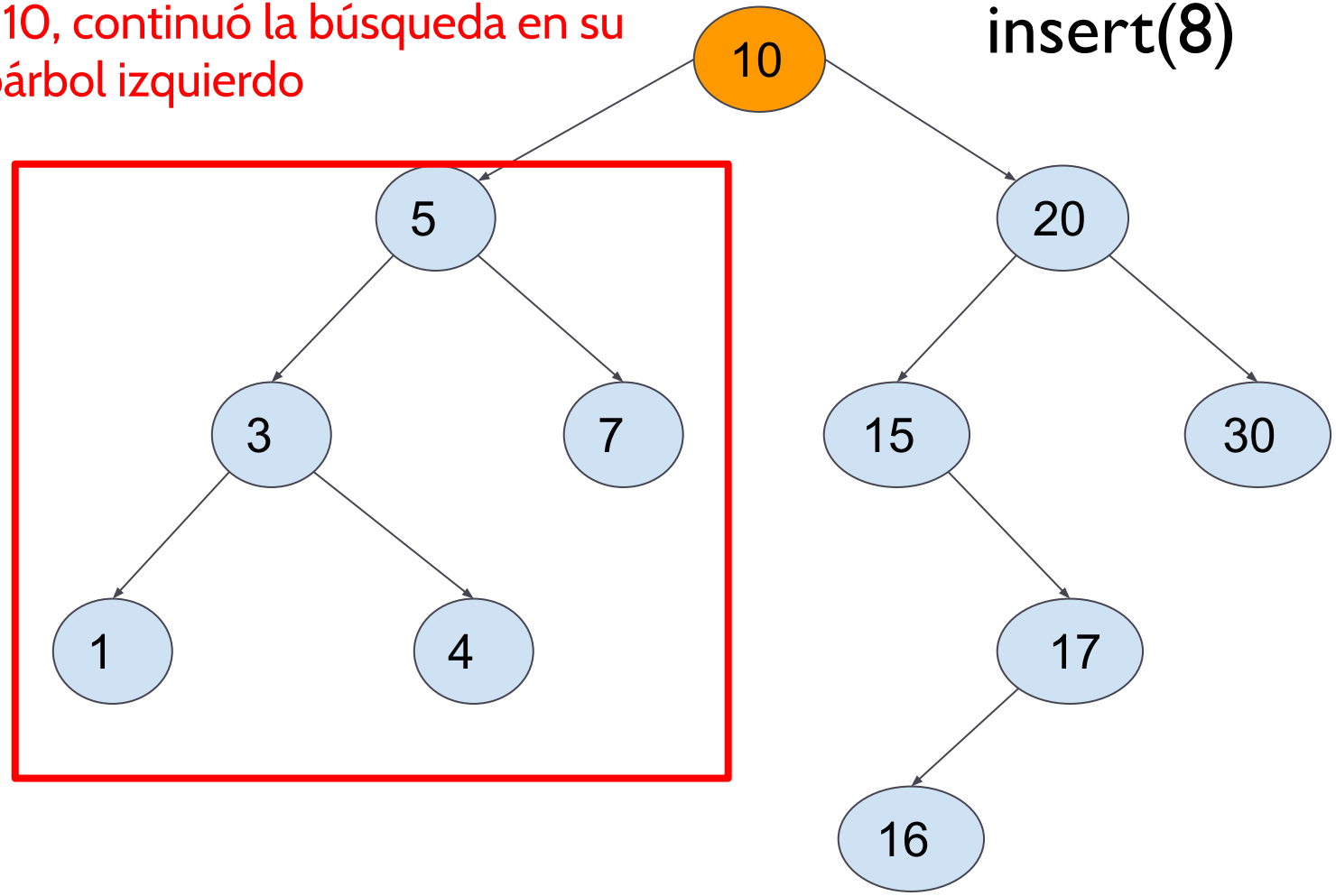
Método **insert**.
Si el elemento a insertar ya existe, no permitimos duplicados.



Los **nuevos nodos insertados** siempre son **nodos hojas**

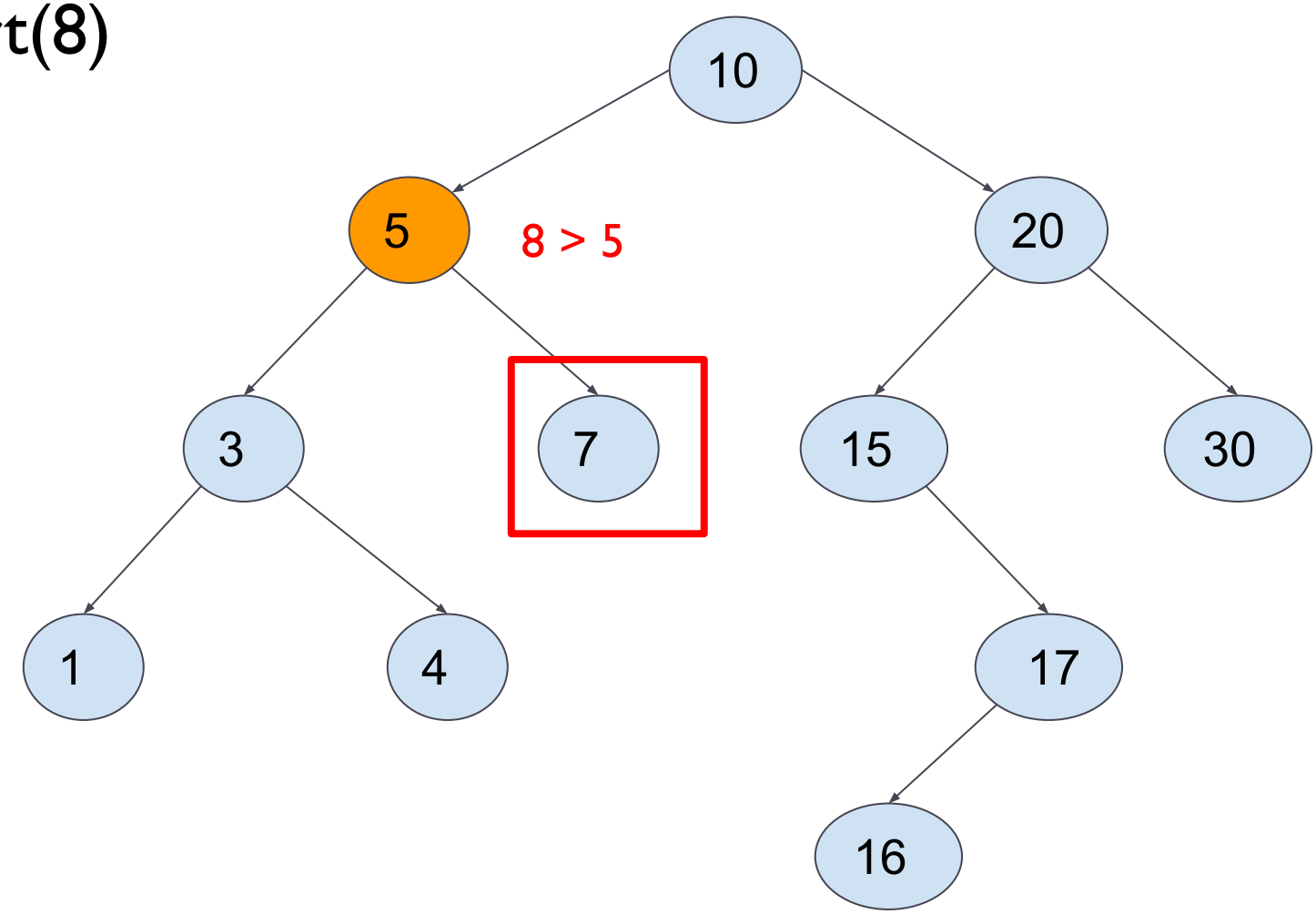
Árboles Binarios de Búsqueda - insert

8 < 10, continuó la búsqueda en su subárbol izquierdo



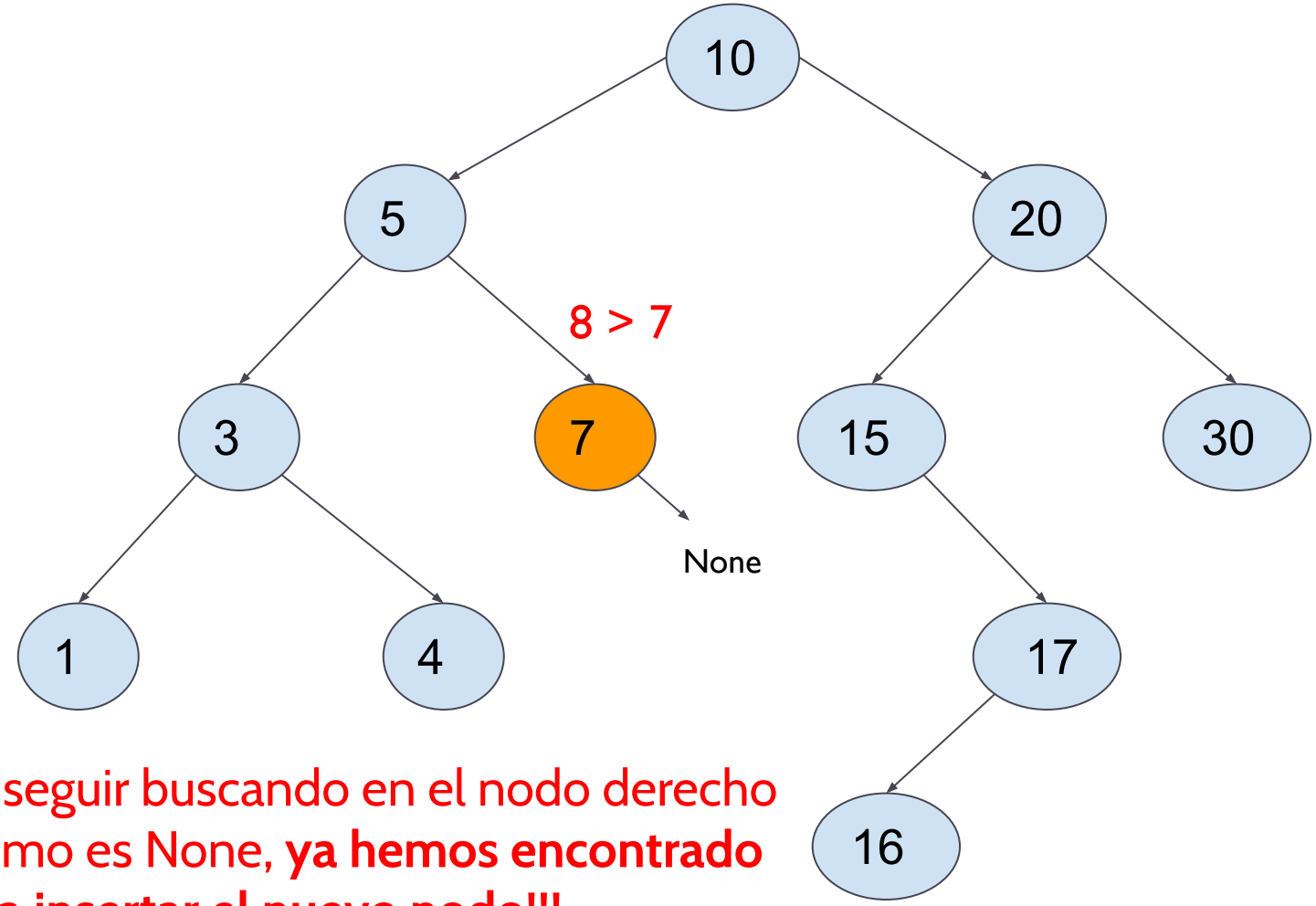
Árboles Binarios de Búsqueda - insert

insert(8)



Árboles Binarios de Búsqueda - insert

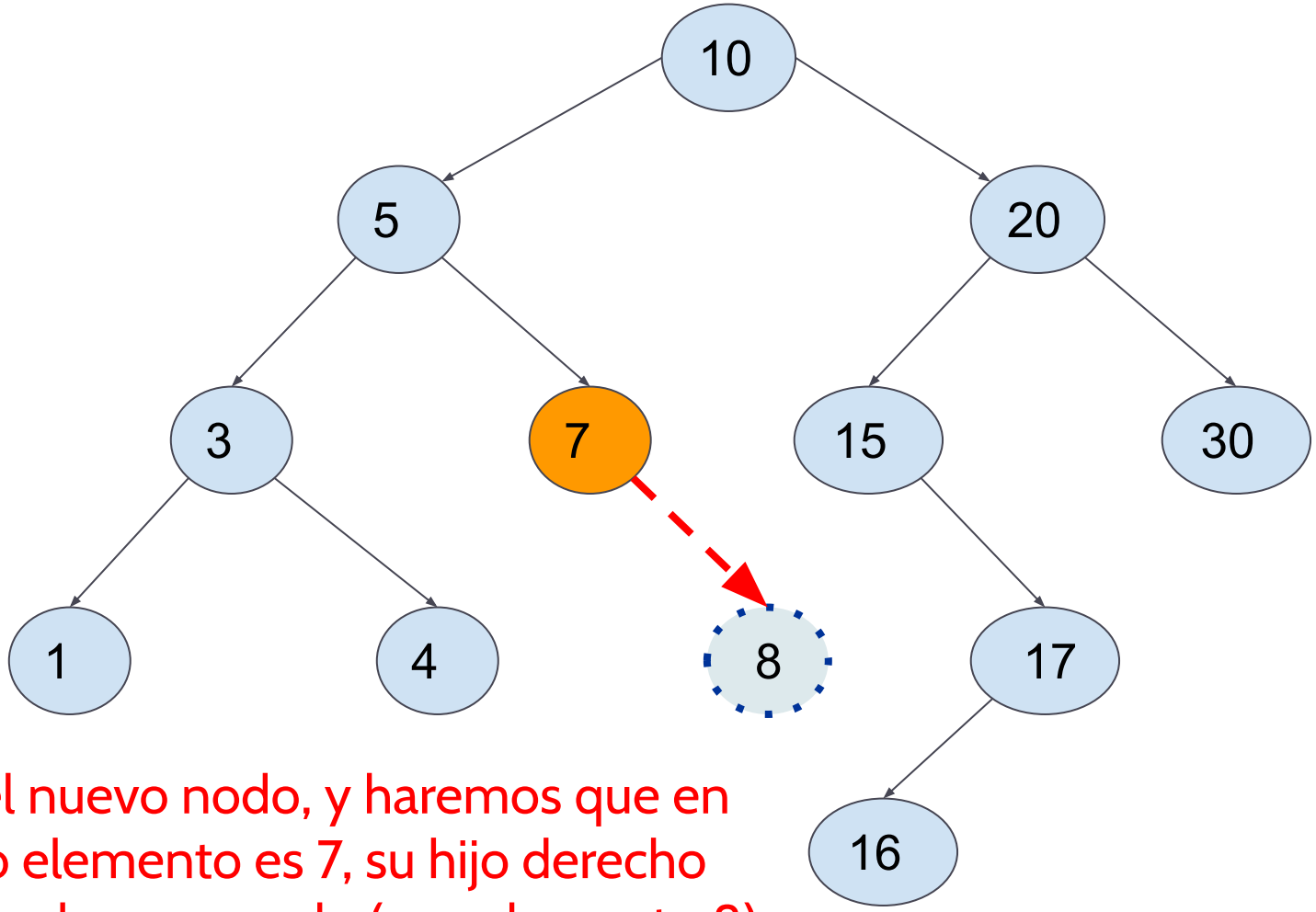
insert(8)



Deberíamos seguir buscando en el nodo derecho de 7, pero como es None, **ya hemos encontrado el hueco para insertar el nuevo nodo!!!**

Árboles Binarios de Búsqueda - insert

insert(8)



Crearemos el nuevo nodo, y haremos que en el nodo cuyo elemento es 7, su hijo derecho debe apuntar al nuevo nodo (con elemento 8)

Implementación insert

```
def insert(self, elem: object) -> None:
    """inserts a new node, with elem is elem. The root must be replaced
    with the new subtree after inserting elem
    in the subtree that hangs down the root"""
    self._root = self._insert(self._root, elem)

def _insert(self, node: BinaryNode, elem: object) -> BinaryNode:
    """recursive function that takes a node and an elem.
    it recursively searches its right location and then inserts it.
    The function returns the new subtree updated with the new node"""
    ...
```

Implementación insert

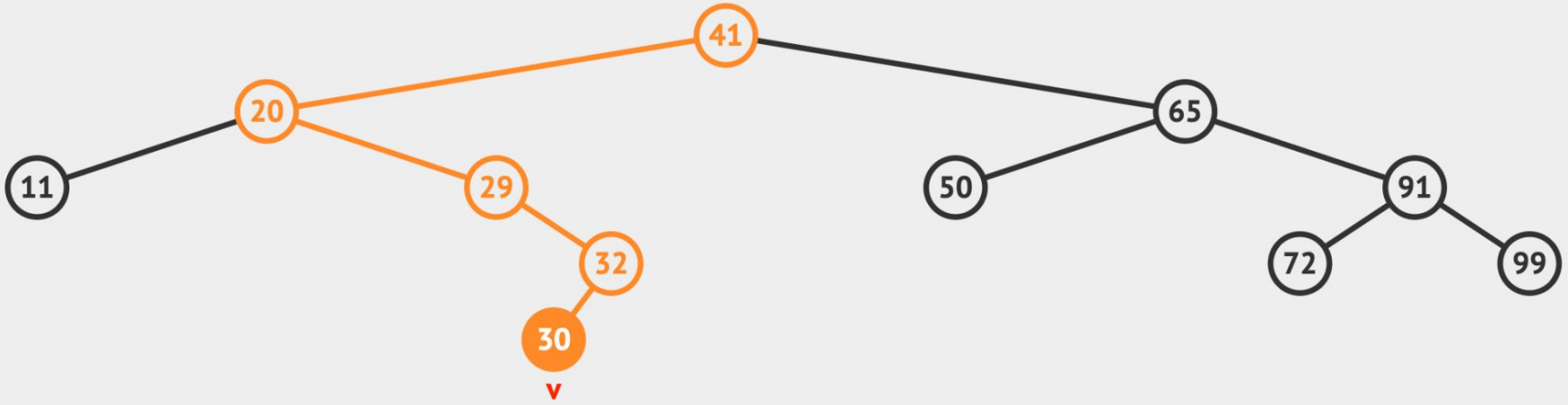
```
def insert(self, elem: object) -> None:
    self._root = self._insert(self._root, elem)

def _insert(self, node: BinaryNode, elem: object) -> BinaryNode:
    if node is None:
        return BinaryNode(elem)

    if node.elem == elem:
        print('Error: elem already exist ', elem)
        return node

    if elem < node.elem:
        node.left = self._insert(node.left, elem)
    else:
        # elem>node.elem
        node.right = self._insert(node.right, elem)

    return node
```

Complejidad Temporal insert

¿Cuál es la complejidad temporal de la función insert?

- Buscar la posición para el nuevo nodo, que en el peor caso, será O(h)
- Crear el nuevo nodo: O(1)
- Añadir la referencia del nodo padre a su nuevo hijo: O(1)

Por tanto, la **complejidad** será O(h). Si el árbol está **balanceado** será **O(log$_2$(n))**, y **O(n) en otro caso**.



remove

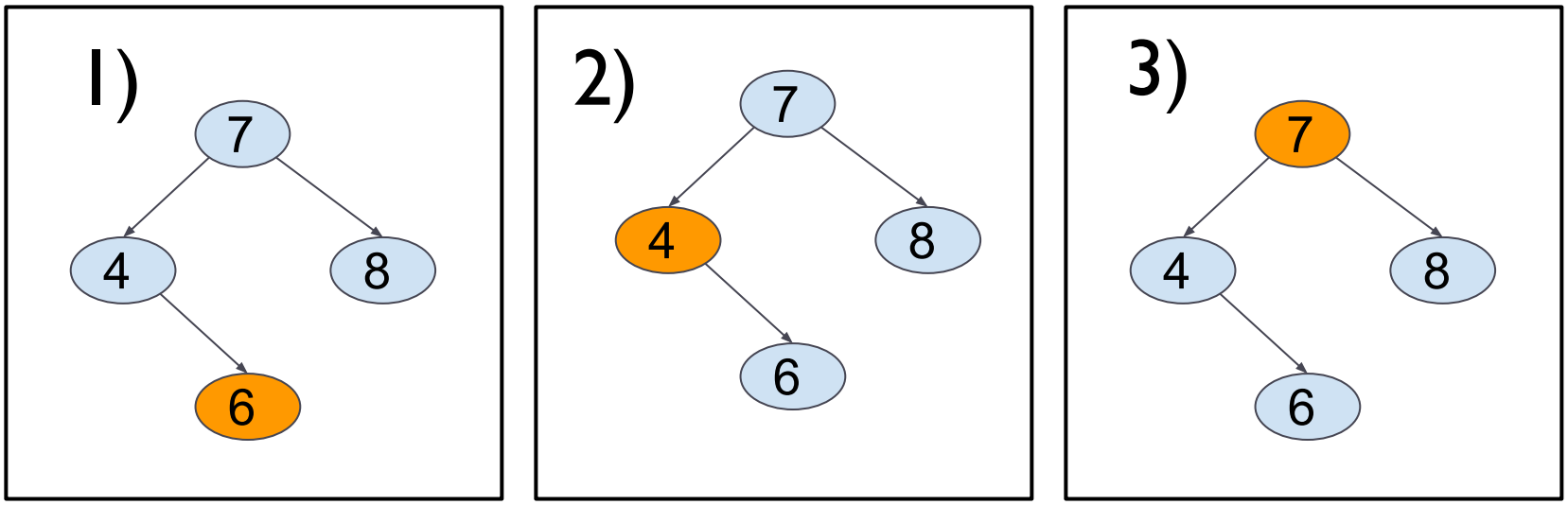
Árboles Binarios de Búsqueda: remove

Si el nodo a borrar existe, vamos a encontrarnos con tres escenarios distintos:

Caso 1: El nodo a borrar es una hoja.

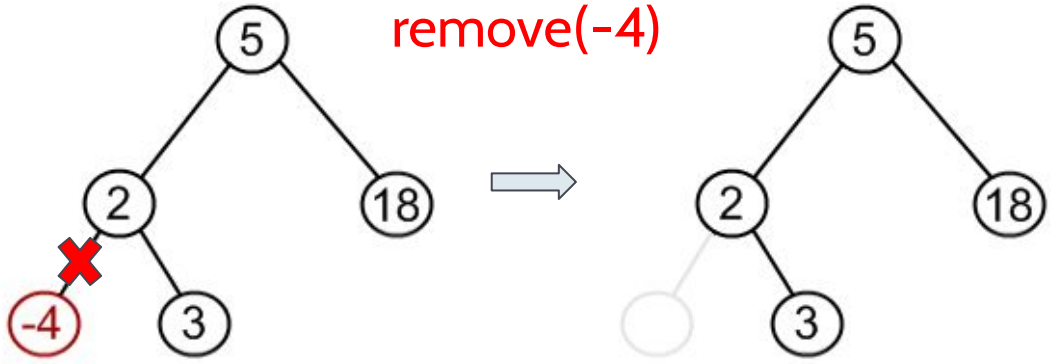
Caso 2: El nodo a borrar sólo tiene un hijo

Caso 3: El nodo a borrar tiene dos hijos



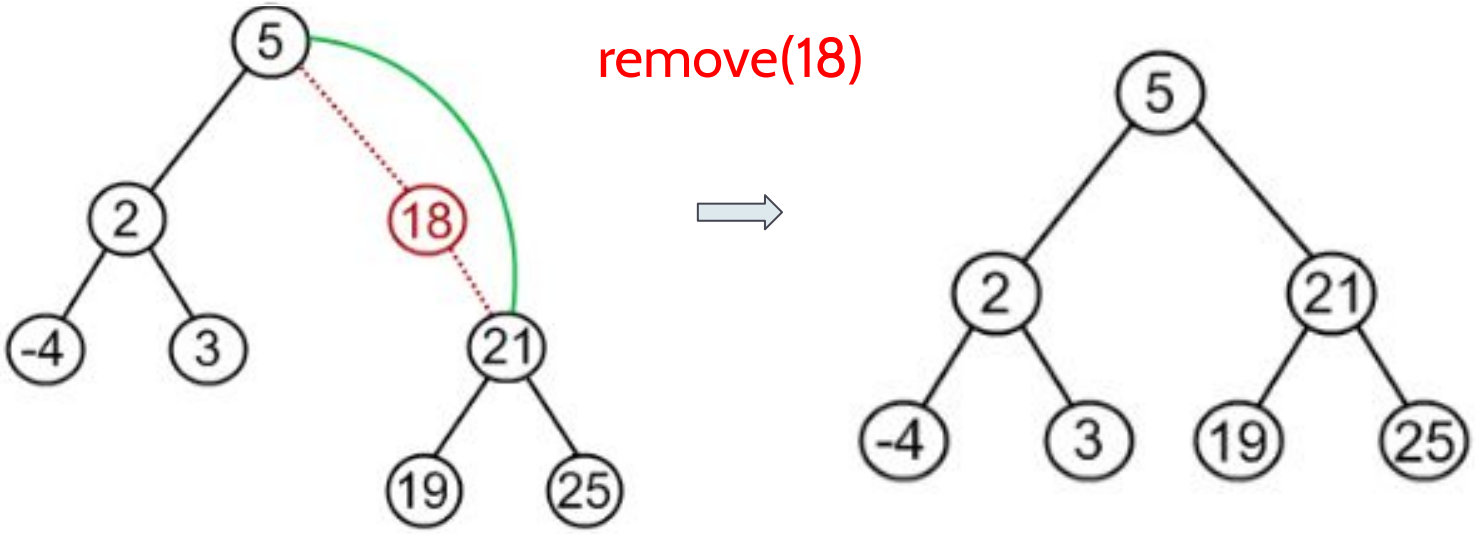
Árboles Binarios de Búsqueda: remove

Caso 1) Borrar un nodo hoja=> Es suficiente con romper la referencia del padre.
En el ejemplo, si queremos borrar el nodo con elemento -4, bastaría con actualizar el atributo .left de su padre (nodo con elemento 2) a None.



Árboles Binarios de Búsqueda: remove

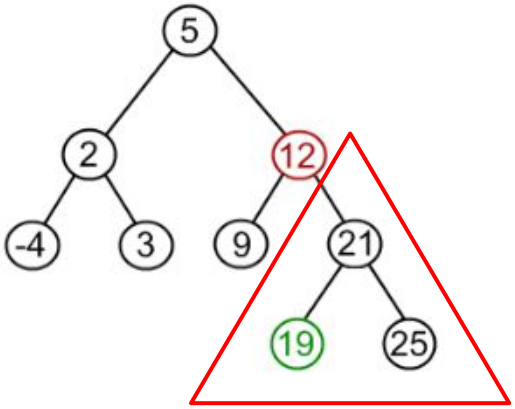
Caso 2) Borrar un nodo con un único hijo: El nodo padre del nodo a borrar deberá apuntar a su nieto (único nodo hijo del nodo a borrar).



Árboles Binarios de Búsqueda: remove

Caso 3: El nodo tiene dos hijos. Debemos buscar su nodo sucesor (en su subárbol derecho, buscar el nodo con menor elemento)

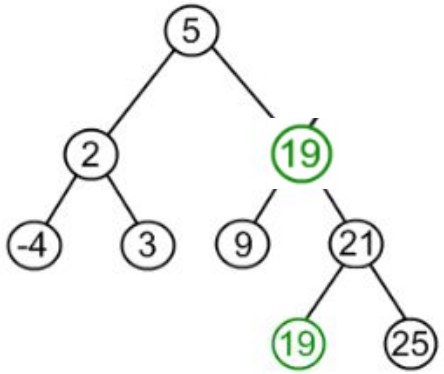
remove(12)



El nodo sucesor al nodo con elemento 12, es el nodo con elemento 19

Árboles Binarios de Búsqueda: remove

Caso 3: El nodo tiene dos hijos. Una vez encontrado el nodo sucesor, vamos a reemplazar el elemento del nodo a borrar, por el elemento del nodo sucesor.



remove(12)

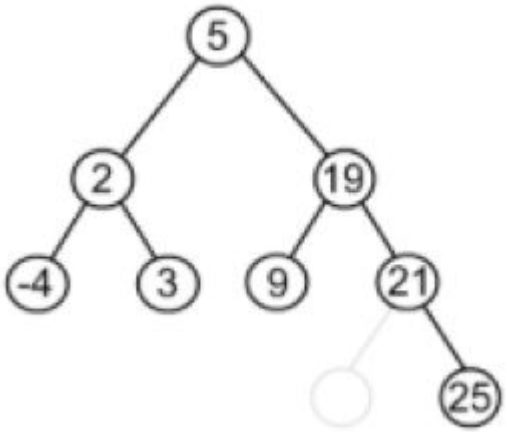
Ahora tendremos dos nodos con el mismo elemento. Debemos borrar el nodo sucesor (que siempre será un caso 1 o caso 2).

Árboles Binarios de Búsqueda: remove

Caso 3: El nodo tiene dos hijos. Borramos el nodo sucesor de su subárbol derecho.

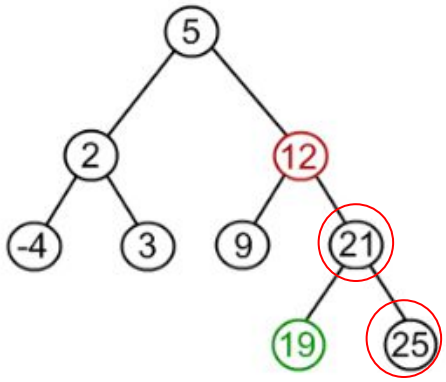
remove(12)

Resultado



Árboles Binarios de Búsqueda: remove

Caso 3: El nodo tiene dos hijos



La estrategia utilizada ha sido reemplazar el nodo a eliminar por uno nodo de su subárbol derecho,

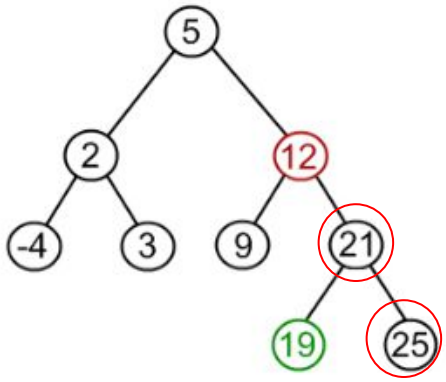
¿por qué hemos elegido el nodo con el elemento más pequeño del subárbol derecho?,

¿Qué pasa si elegimos cualquiera de los otros nodos en el subárbol derecho?

Por ejemplo, ¿qué pasa si reemplazamos el valor de 12, por 21 o por 25?.

Árboles Binarios de Búsqueda: remove

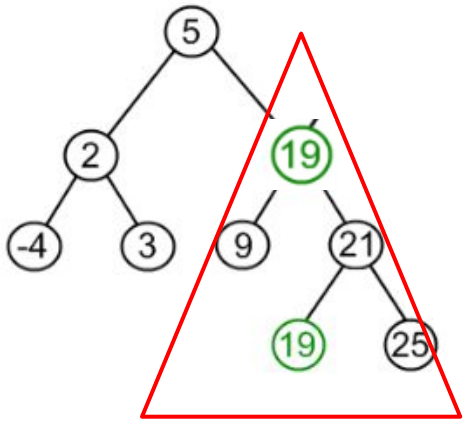
Caso 3: El nodo tiene dos hijos



Debemos elegir siempre el nodo sucesor para que el árbol resultante siga siendo un árbol binario de búsqueda

Árboles Binarios de Búsqueda: remove

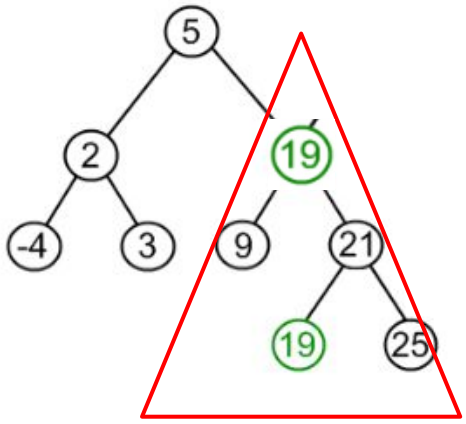
Caso 3: El nodo tiene dos hijos



En este ejemplo, el nodo sucesor que debemos borrar es un nodo hoja.
¿El nodo sucesor podría tener un hijo derecho?
¿El nodo sucesor podría tener un hijo izquierdo?
¿El nodo sucesor podría tener dos hijos?

Árboles Binarios de Búsqueda: remove

Caso 3: El nodo tiene dos hijos



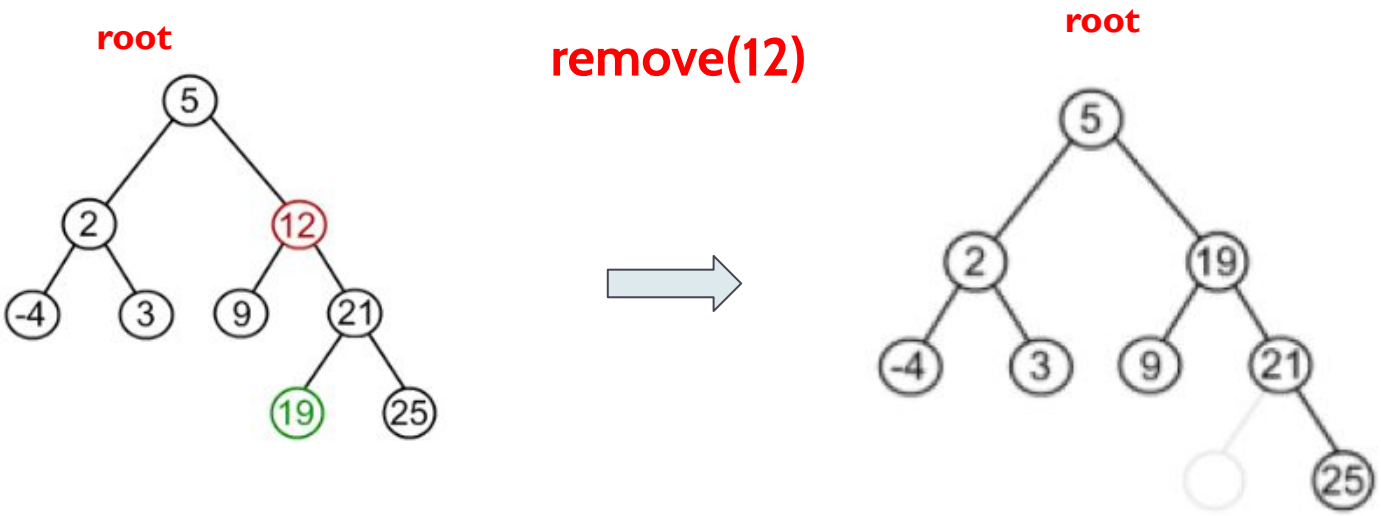
El **nodo sucesor no puede tener hijo izquierdo**, porque de tener hijo izquierdo, ese sería el nodo sucesor (el más pequeño del subárbol derecho).

Por tanto, el sucesor puede ser una hoja o tener un único hijo, el hijo derecho.

Nuestro algoritmo ya está preparado para borrar una hoja o borrar un nodo con un único hijo!!!

Árboles Binarios de Búsqueda: remove

Para borrar un nodo, **primero deberemos buscarlo** en el árbol. Vamos a apoyarnos en una función recursiva **_remove** que nos permita localizar el nodo a borrar. Cada llamada recursiva sobre un determinado nodo (un subárbol), deberá devolver el subárbol actualizado tras borrar el nodo.



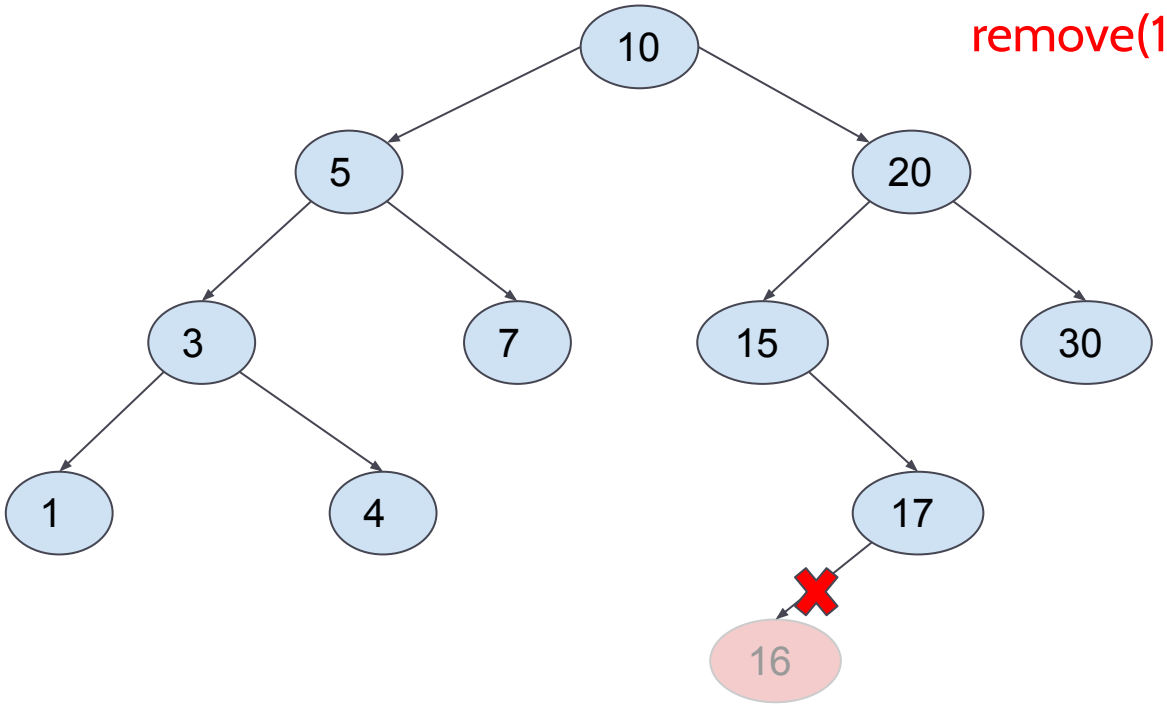
Implementación remove

```
def remove(self, elem: object) -> None:
    # update the root with the new subtree after remove elem
    self._root = self._remove(self._root, elem)

def _remove(self, node: BinaryNode, elem: object) -> BinaryNode:
    """removes from the subtree node, the node whose element is elem.
    returns the subtree node updated after removing elem"""

    # First, we have to search the node
    if elem < node.elem:
        node.left = self._remove(node.left, elem)
    elif elem > node.elem:
        node.right = self._remove(node.right, elem)
    else:  # elem == node.elem
        ...
```

Implementación remove

```
def remove(self, elem: object) -> None:
    self._root = self._remove(self._root, elem)

def _remove(self, node: BinaryNode, elem: object) -> BinaryNode:
    if node is None:
        print(elem, ' not found')
        return node
    if elem < node.elem:
        node.left = self._remove(node.left, elem)
    elif elem > node.elem:
        node.right = self._remove(node.right, elem)
    else:
        if node.left is None and node.right is None:  # Case 1
            return None
        if node.left is None:  # Case 2
            return node.right
        elif node.right is None:  # Case 2
            return node.left
        else:  # Case 3
            successor = self._minimum_node(node.right)
            node.elem = successor.elem
            node.right = self._remove(node.right, successor.elem)
    return node
```

Complejidad Temporal: borrar un hoja

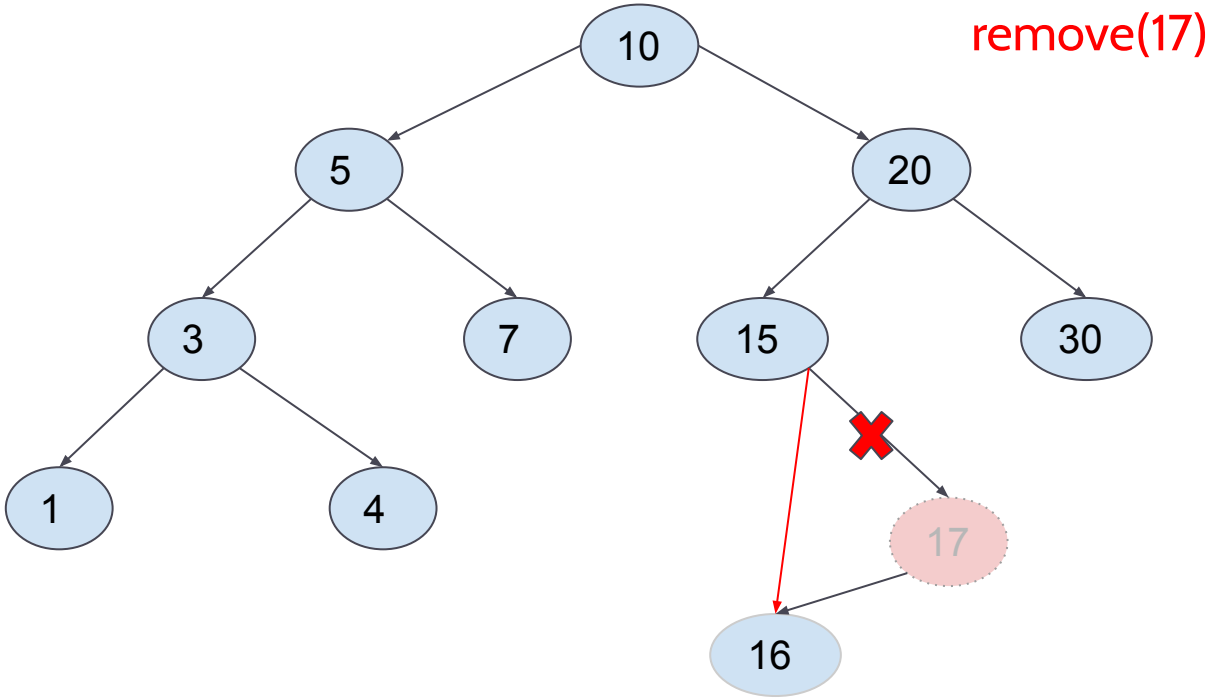


Si el nodo a borrar no tiene hijos, la complejidad temporal será:

- Búsqueda: $O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso.
- Borrar la referencia de su padre: $O(1)$.

Por tanto, la **complejidad** será **$O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso**.

Complejidad Temporal: borrar nodo con un único hijo

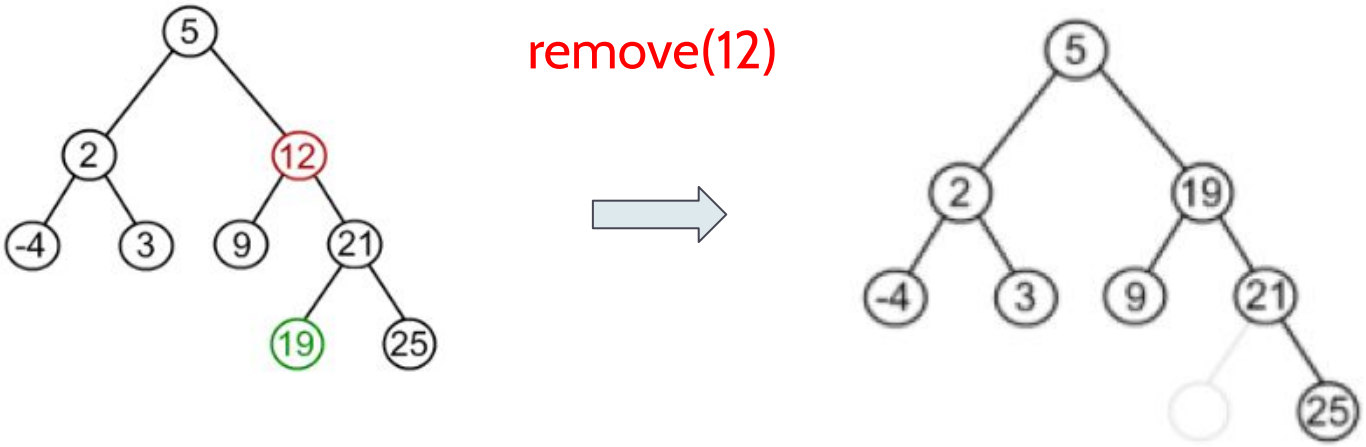


Si el **nodo a borrar únicamente tiene un hijo**, la complejidad temporal será:

- Búsqueda: $O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso.
- Modificar la referencia del nodo padre para que apunte a su nieto: $O(1)$.

Por tanto, la **complejidad** será **$O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso**.

Complejidad Temporal: borrar nodo con dos hijos



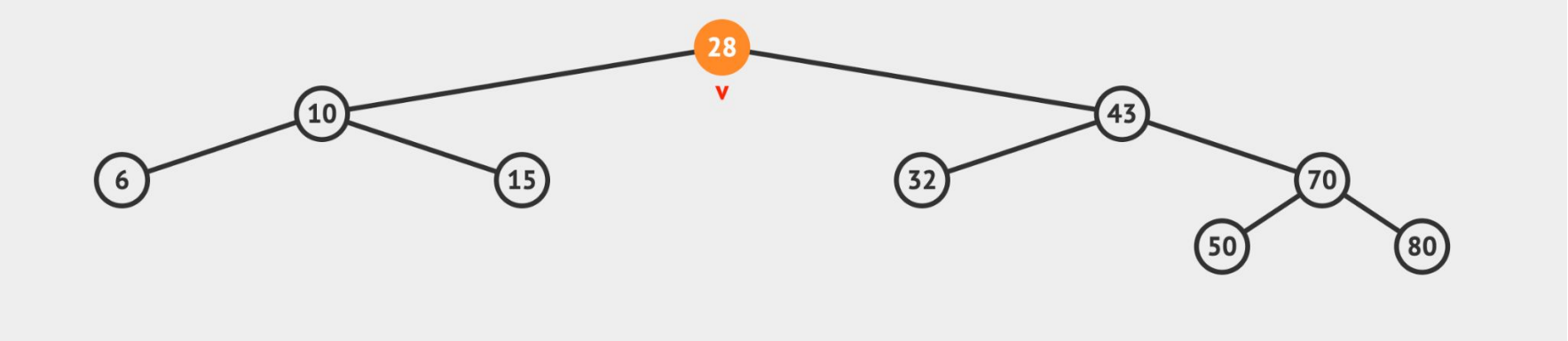
Si el **nodo a borrar tiene dos hijos**, la complejidad temporal será:

- Búsqueda: $O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso.
- Buscar el sucesor: $O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso.
- Reemplazar el elemento del nodo por el elemento del sucesor: $O(1)$
- Borrar el sucesor (caso 1 o caso 2): $O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso.

Por tanto, la complejidad será **$O(\log_2 n)$ si está balanceado, $O(n)$ en otro caso**.

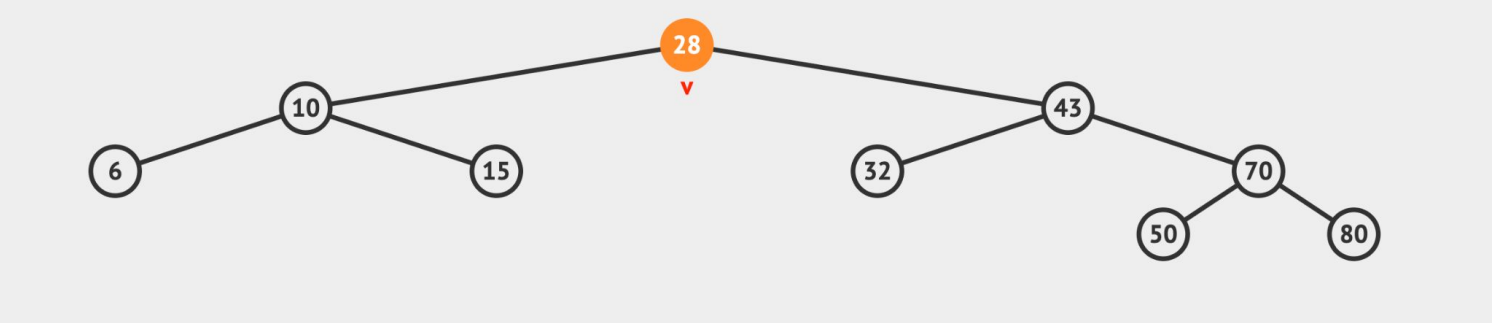
Árboles Binarios de búsqueda: remove

En el siguiente árbol binario de búsqueda, borra esta secuencia: 70, 15, 28, 43, 32

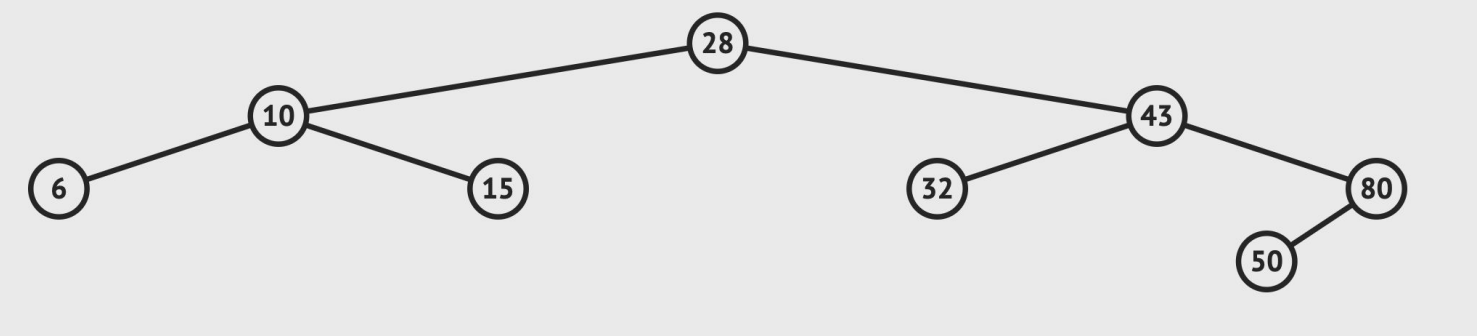


Árboles Binarios de búsqueda: remove

remove(70)

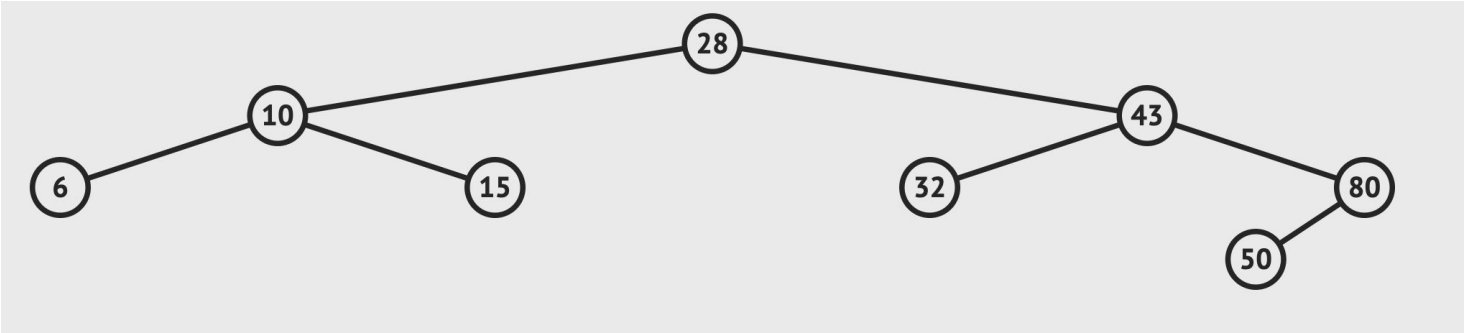


El nodo con elemento 70 ha sido reemplazado por su sucesor (del subárbol derecho).

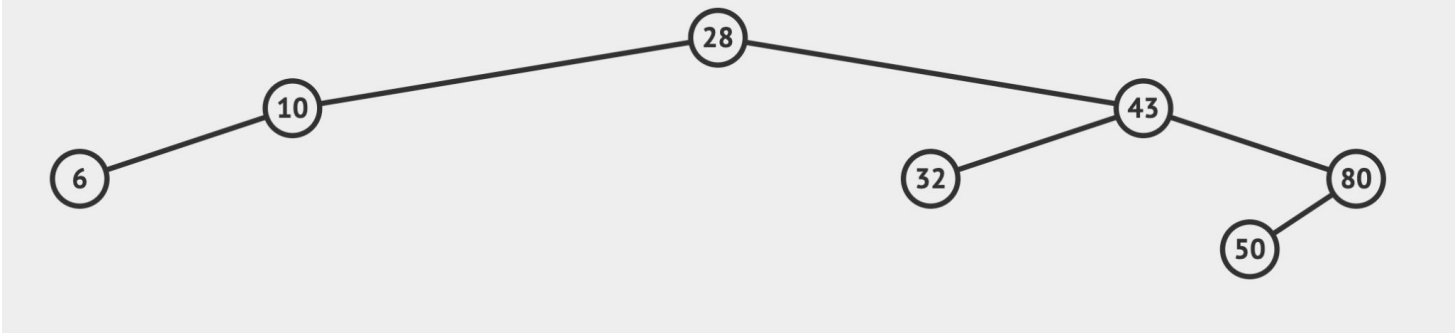


Árboles Binarios de búsqueda: remove

remove(15), es decir, borrar un nodo hoja

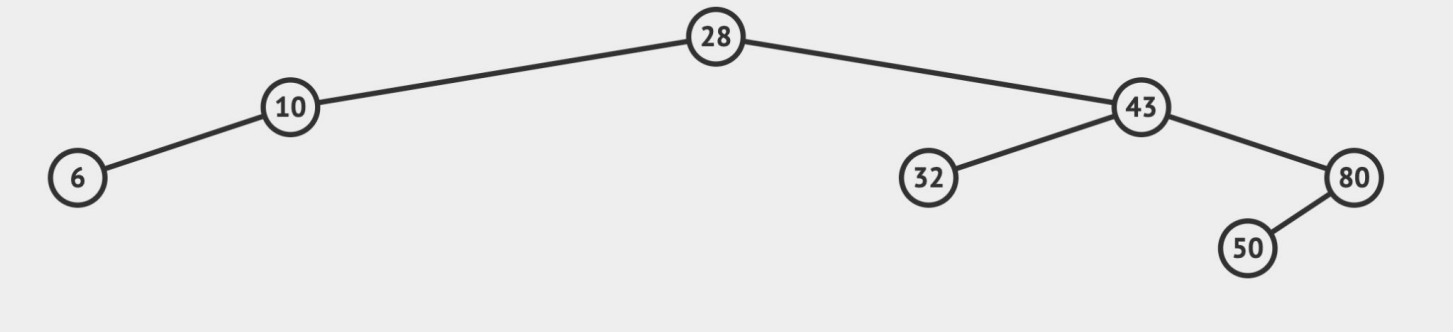


El resultado es:

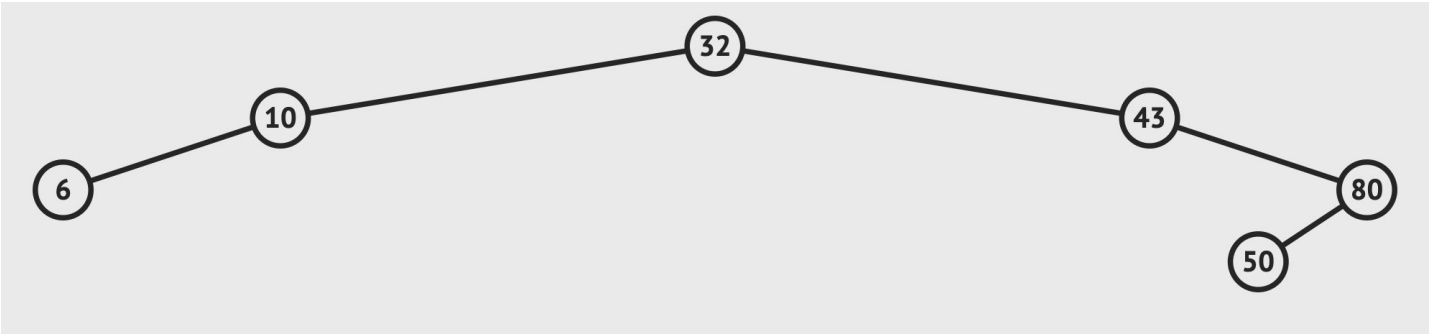


Árboles Binarios de búsqueda: remove

remove(28), vamos a borrar la raíz, que tiene dos hijos

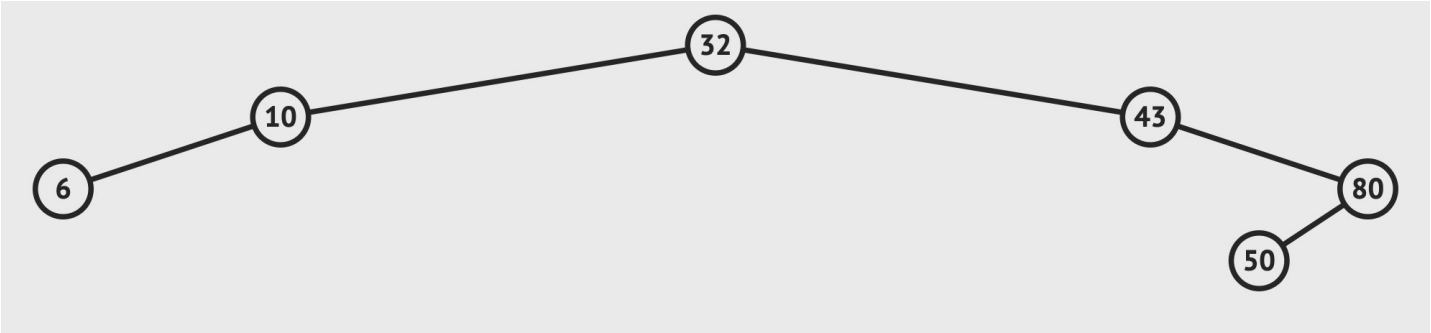


Reemplazamos la raíz por su sucesor (nodo 321) en el subárbol derecho

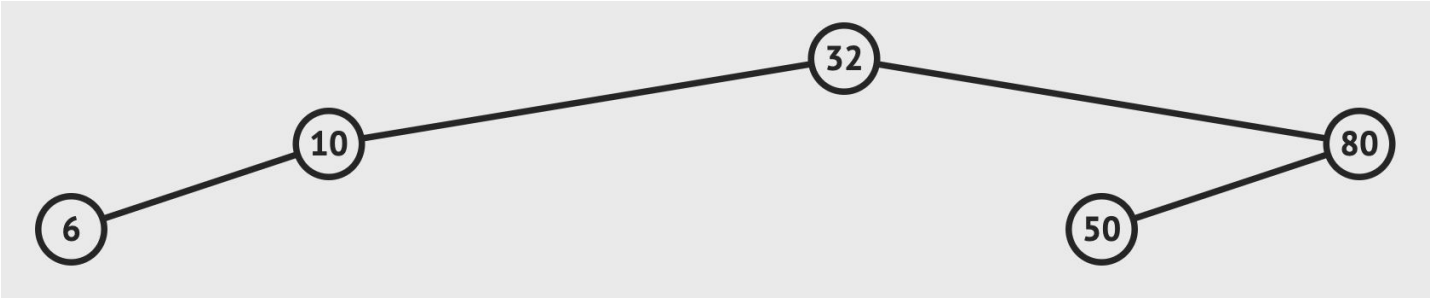


Árboles Binarios de búsqueda: remove

remove(43), es decir, un nodo con un único hijo

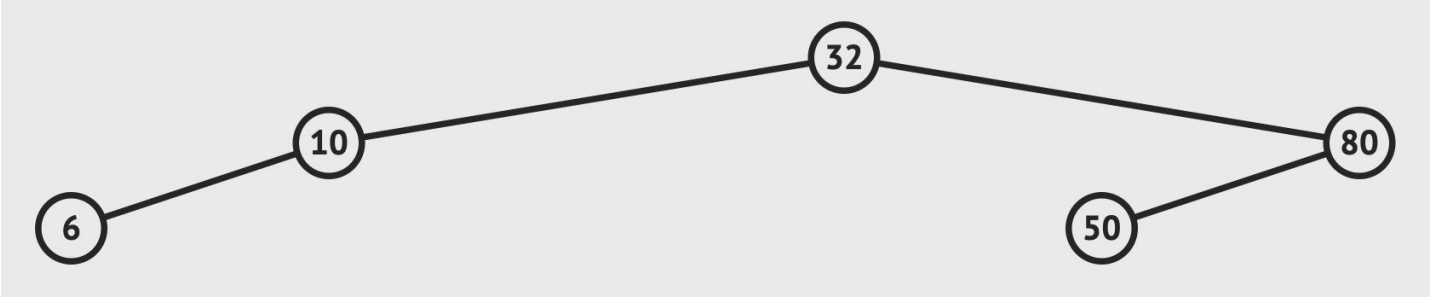


Enlazamos su padre (nodo con elemento 32) a su nieto (nodo con elemento 80)

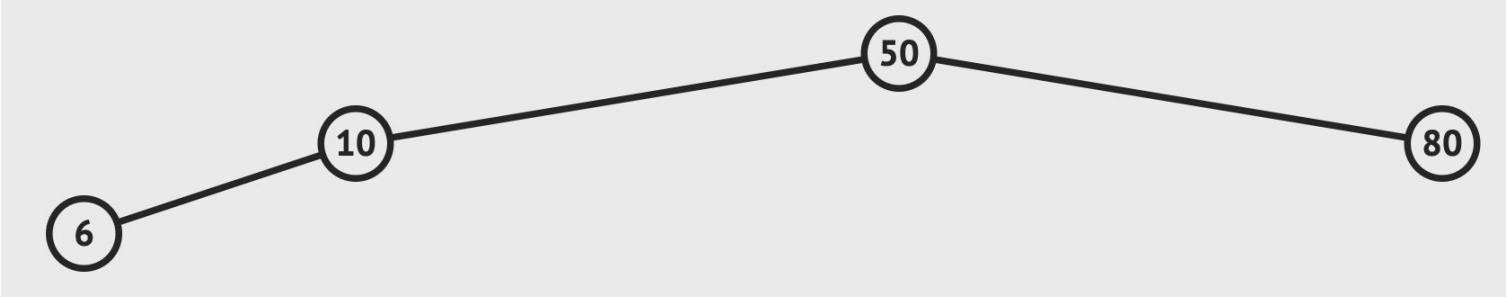


Árboles Binarios de búsqueda: remove

Por último, remove(32), la raíz con dos hijos



Reemplazamos la raíz por su sucesor (nodo más pequeño en el subárbol derecho, es decir, nodo con elemento 50).



Conclusiones

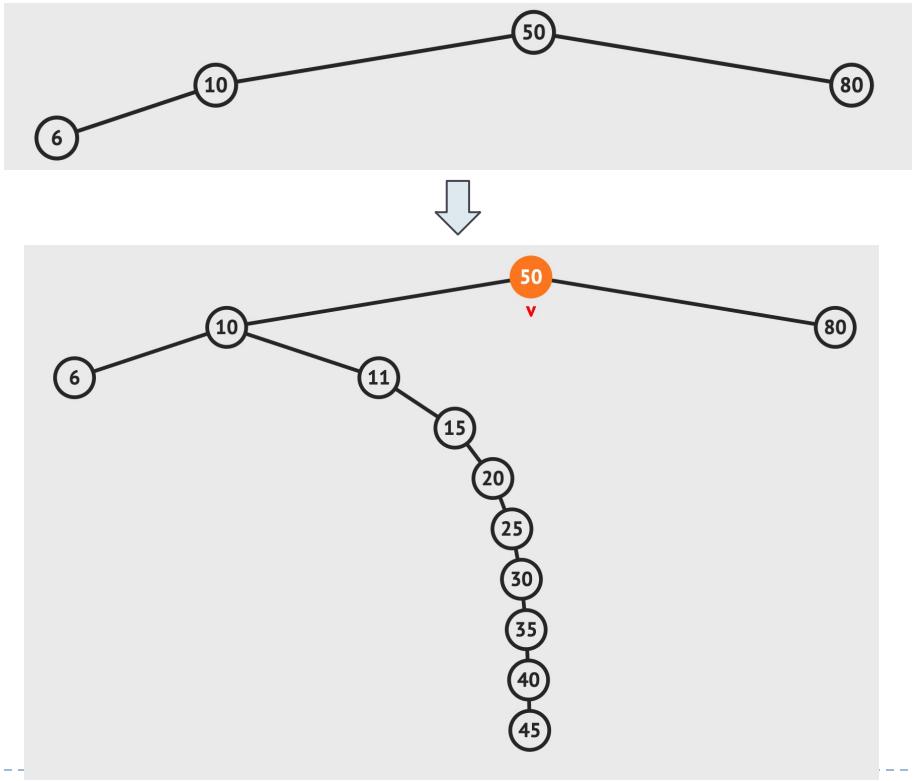
- La información con relaciones jerárquicas es difícilmente representable en una estructura lineal.
- En general, en las estructuras lineales, la complejidad de las operaciones de búsqueda, inserción y borrado es lineal (O(n)).

Conclusiones

- Los árboles binarios de búsqueda (**ABB**), en inglés **Binary Search Tree (BST)**, son estructuras más eficientes que las estructuras lineales, ya que la **complejidad** de sus operaciones es **logarítmica**, siempre y cuando, los árboles estén **balanceados** (para cualquier nodo, la diferencia entre longitud su rama izquierda y derecha es ±1)

Conclusiones

Sin embargo, un ABB podría fácilmente degenerar en una lista, y la complejidad de sus operaciones pasaría de logarítmica a lineal. Por ejemplo, en el siguiente árbol, inserta la secuencia de números: 11,15, 20, 25, 30, 35, 40, 45



Árbol degenerado en una lista, complejidad de las operaciones O(h) → O(n)

Solución: Debemos mantener los árboles balanceados, después de cada inserción o borrado. **Árboles AVL**