



UNIVERSIDAD NACIONAL DE ASUNCIÓN  
FACULTAD POLITÉCNICA  
*Construyendo el futuro*

# Árboles binarios (de búsqueda)

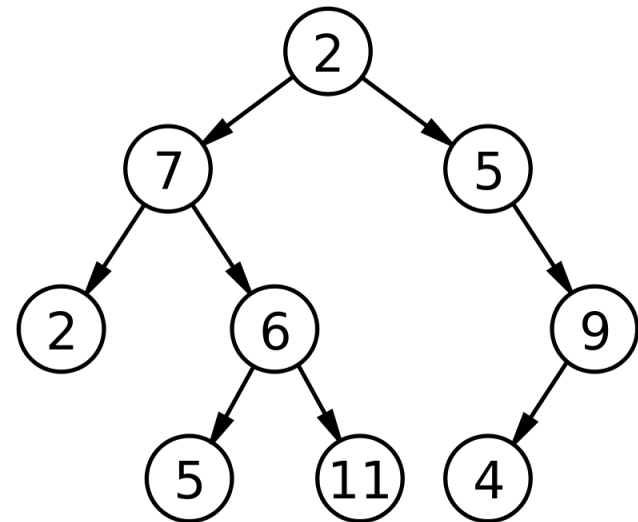
Algoritmos y Estructuras de Datos II

Departamento de Informática



# ¿Qué veremos?

- Árboles
  - Definición
  - Representación
  - Árbol binario de búsqueda
    - Inserción
    - Búsqueda
    - Borrado
    - Ejercicios



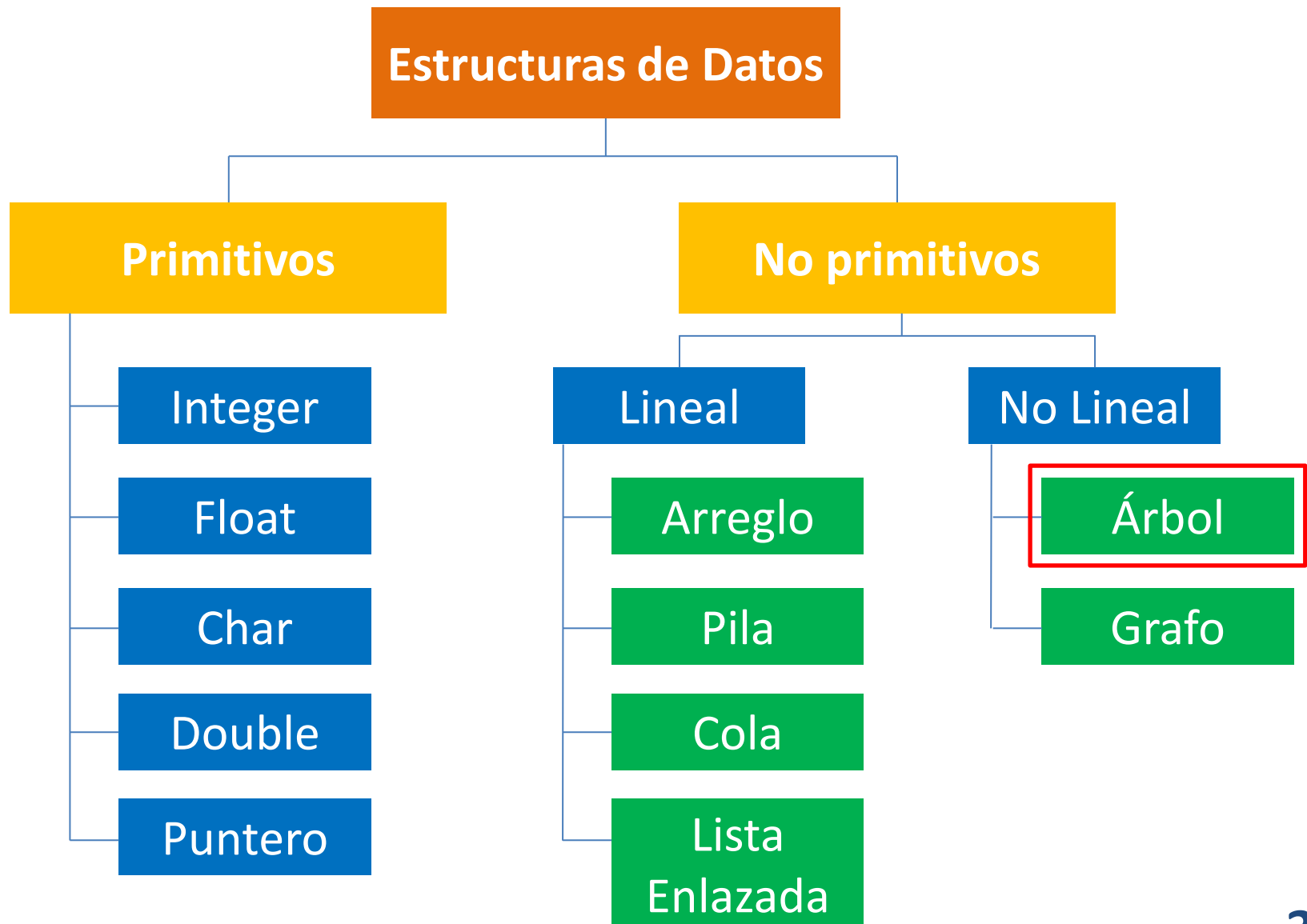
Referencias:

Capítulo 13 – “Fundamentos de la Programación. Algoritmos, estructuras de datos y objetos” – Luis Joyanes Aguilar;

Capítulo 12 – “Introduction to Algorithms”, Cormen *et al.*

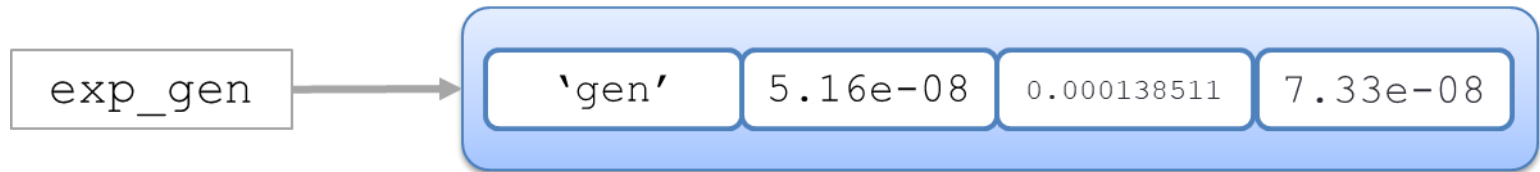
\*Agradecimientos a los profesores Carlos Filippi y Cristian Aceval por los materiales de referencia

# Estructuras de datos

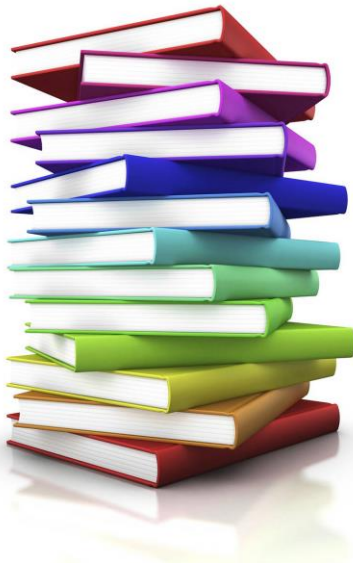


# Estructuras de datos lineales

Las estructuras de datos que han sido examinadas hasta ahora son lineales. A cada elemento le correspondía siempre un “siguiente” elemento. La linealidad es típica de cadenas, de elementos de arrays o listas, de campos en estructuras, entradas en pilas o colas.

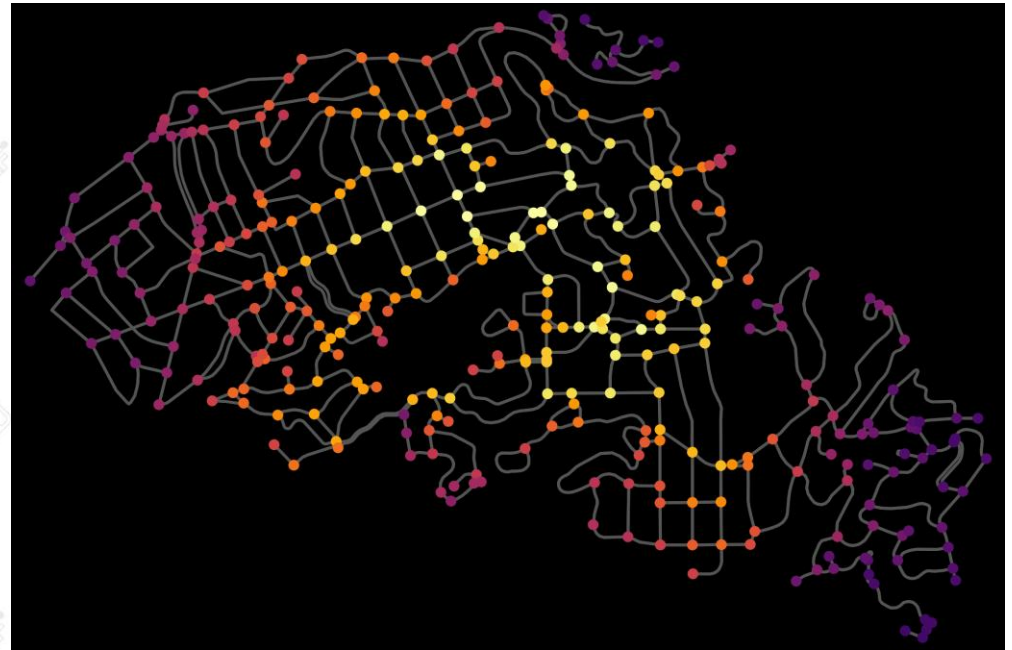
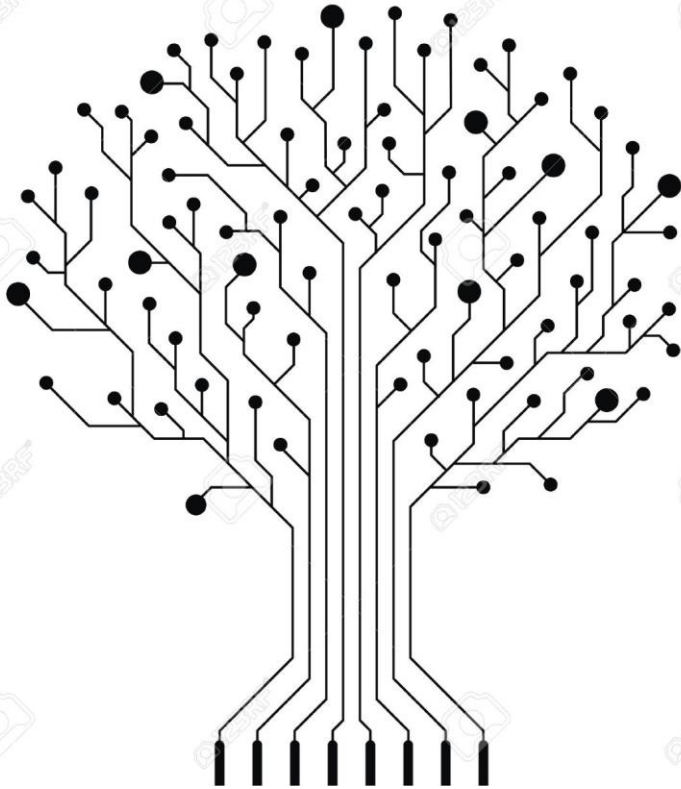


Vector



# Estructuras de datos no lineales

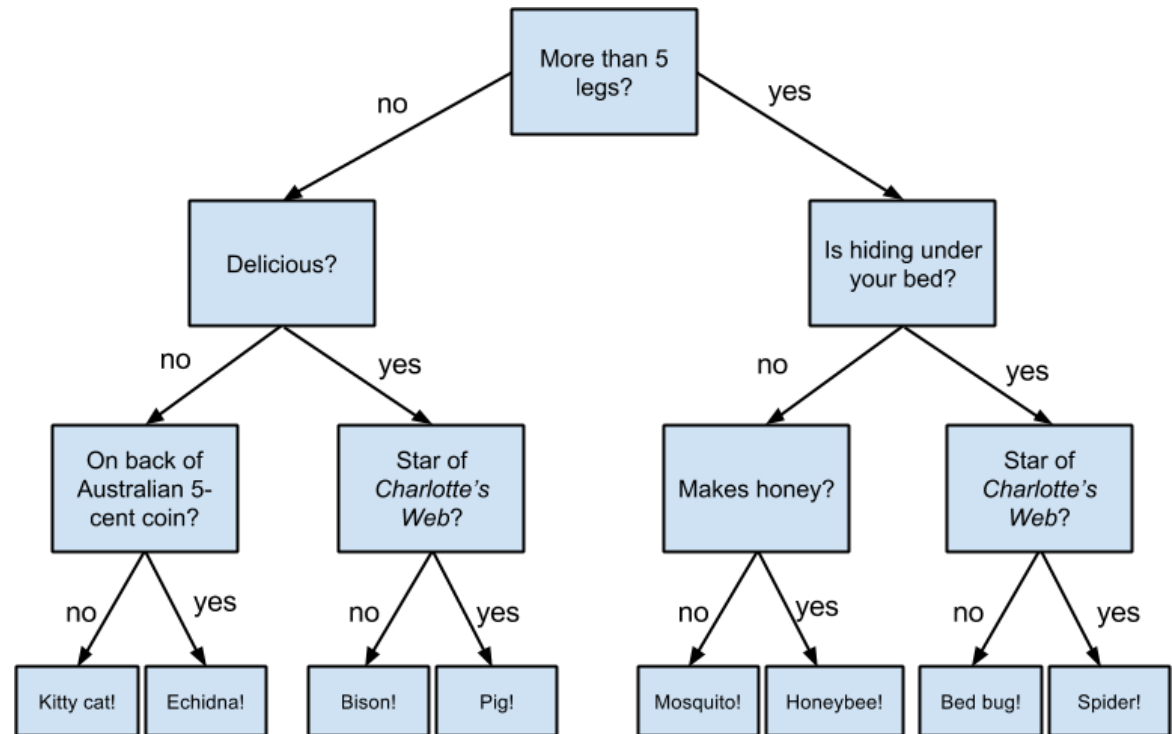
En estas estructuras cada elemento puede tener diferentes “siguientes” elementos, que introduce el concepto de estructuras de bifurcación. Las estructuras de datos no lineales son **árboles** y **grafos**.



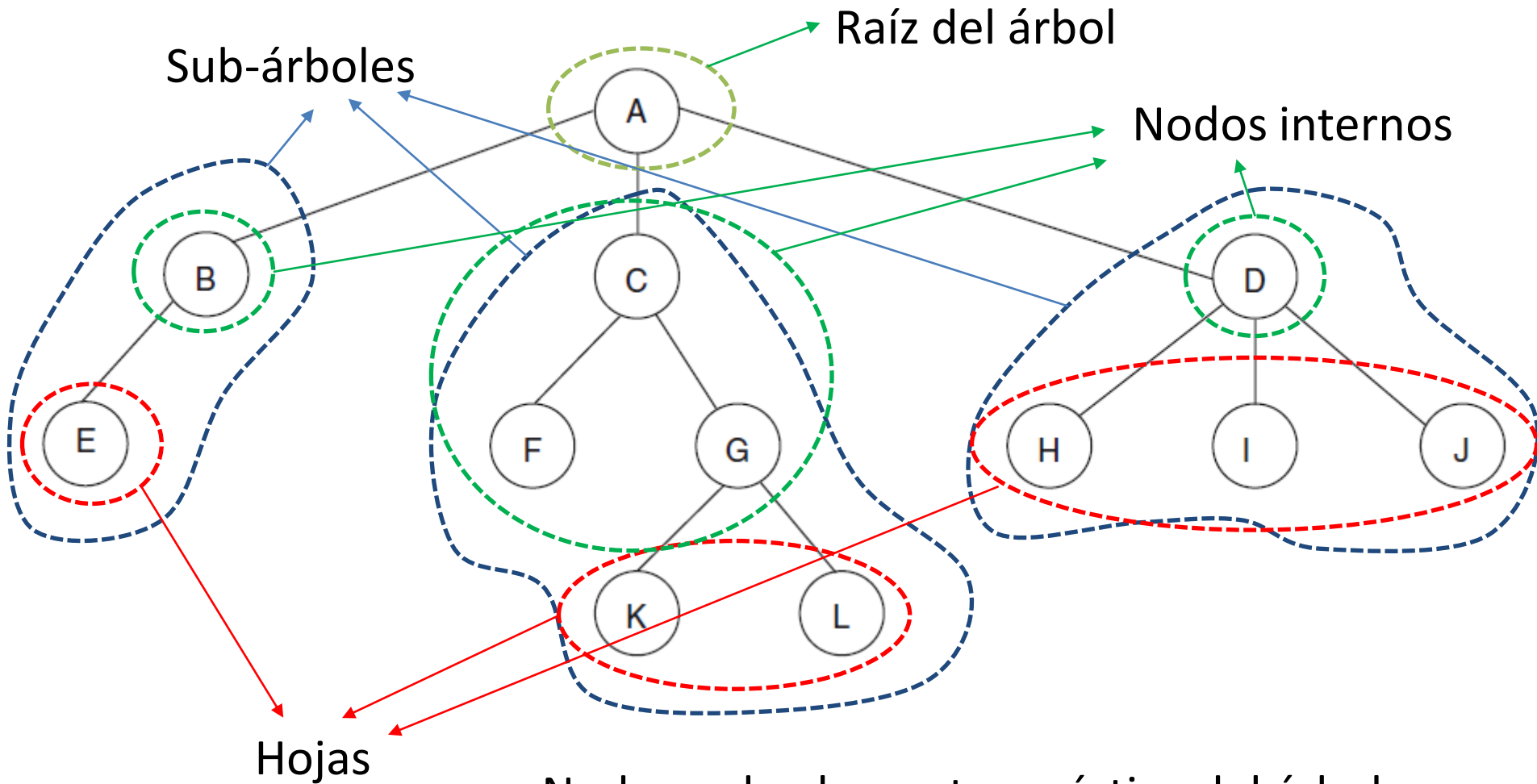
# Árboles

El árbol es una estructura de datos fundamental en ciencias de la computación, y se encuentra desde el análisis teórico de algoritmos (ej: métodos de ordenamiento y búsqueda) hasta la compilación (árboles sintácticos para representar las expresiones de un lenguaje); e incluso en la inteligencia artificial (ej: árboles de decisión).

Las estructuras tipo árbol se usan principalmente para representar datos con una relación jerárquica entre sus elementos, como son árboles genealógicos, tablas, etc.

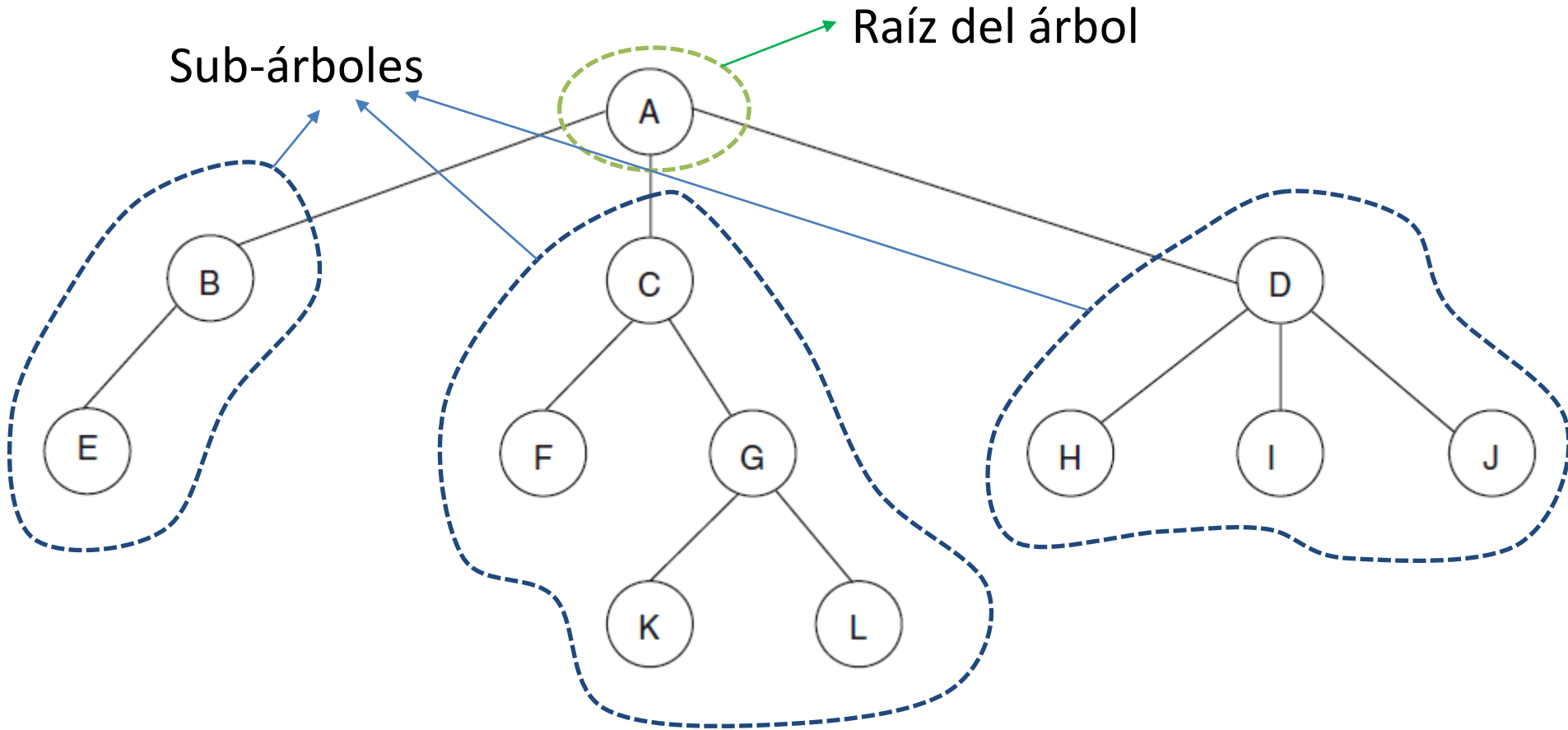


# Árboles – Composición



Nodo: cada elemento o vértice del árbol  
Arista: conexión entre un par de nodos

# Árboles – Definición recursiva

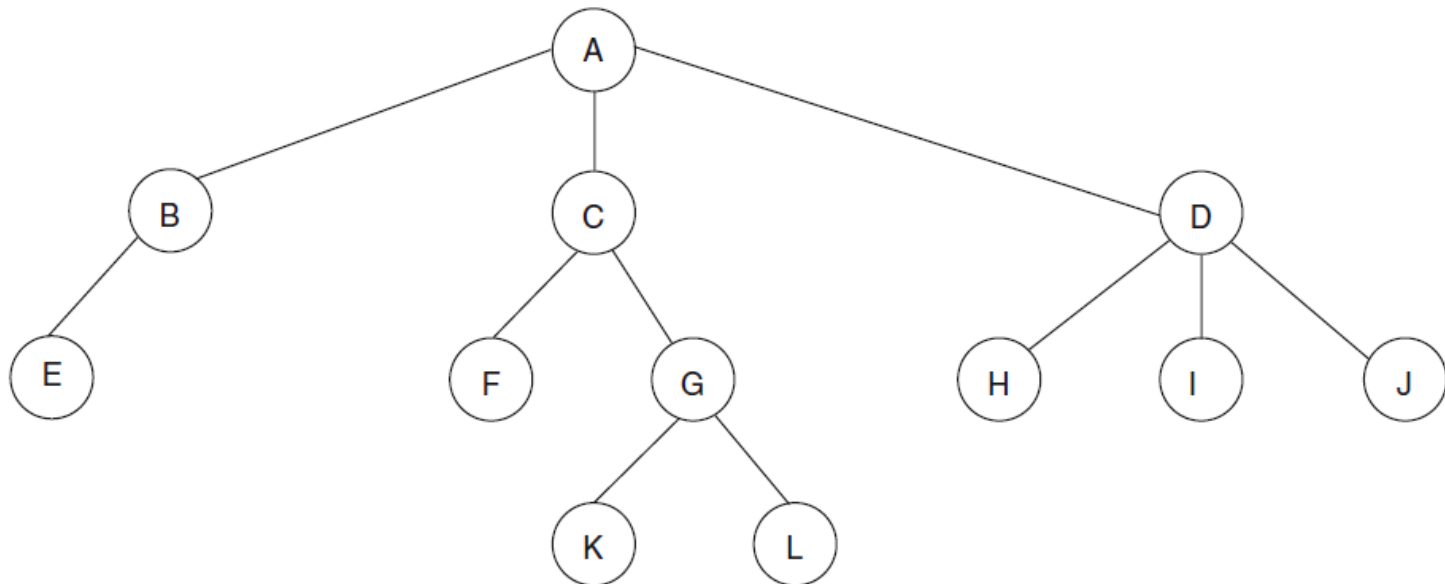


Colección de elementos que es vacío o consiste en un elemento raíz y cero o más subárboles no vacíos  $T_1, T_2, \dots, T_n$ ; con cada una de sus respectivas raíces conectada por medio de una arista a la raíz.



# Árboles – Definiciones

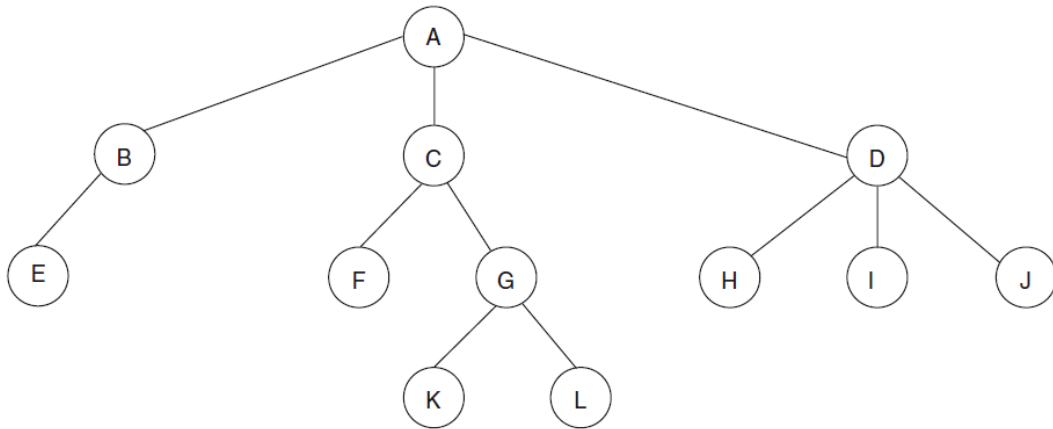
- **Nivel o profundidad de un nodo:** longitud del camino desde la raíz al nodo.
- **Altura de un nodo:** longitud del camino del nodo hasta la hoja más profunda. La altura del árbol es la altura de la raíz.
- **Tamaño de un nodo:** número de descendientes (incluyéndose él). El tamaño del árbol es el tamaño de la raíz.



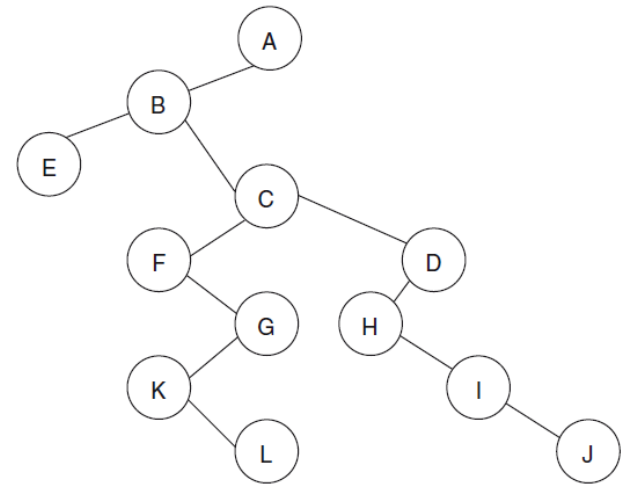
# Árboles binarios

Un árbol binario es un conjunto finito de nodos, tales que:

- Existe un nodo denominado raíz del árbol.
- Cada nodo puede tener 0, 1 o 2 subárboles (o hijos), conocidos como subárbol (o hijo) izquierdo y subárbol (o hijo) derecho.



Árbol general



Árbol binario  
equivalente

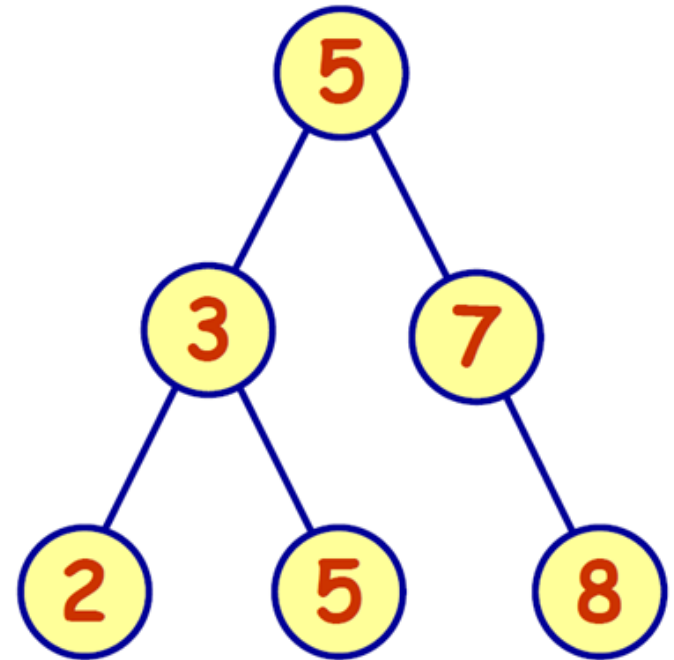
# Árboles binarios - Representación

Los árboles pueden representarse a través de **listas**. Se representa cada nodo del árbol mediante un objeto “nodo”. Se supone que cada nodo contiene una clave, y los demás atributos dependen de la representación empleada.

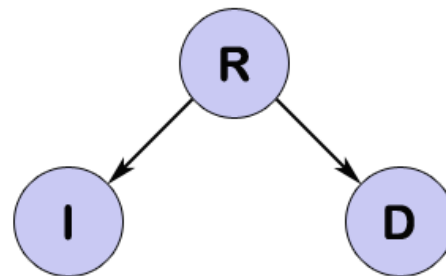
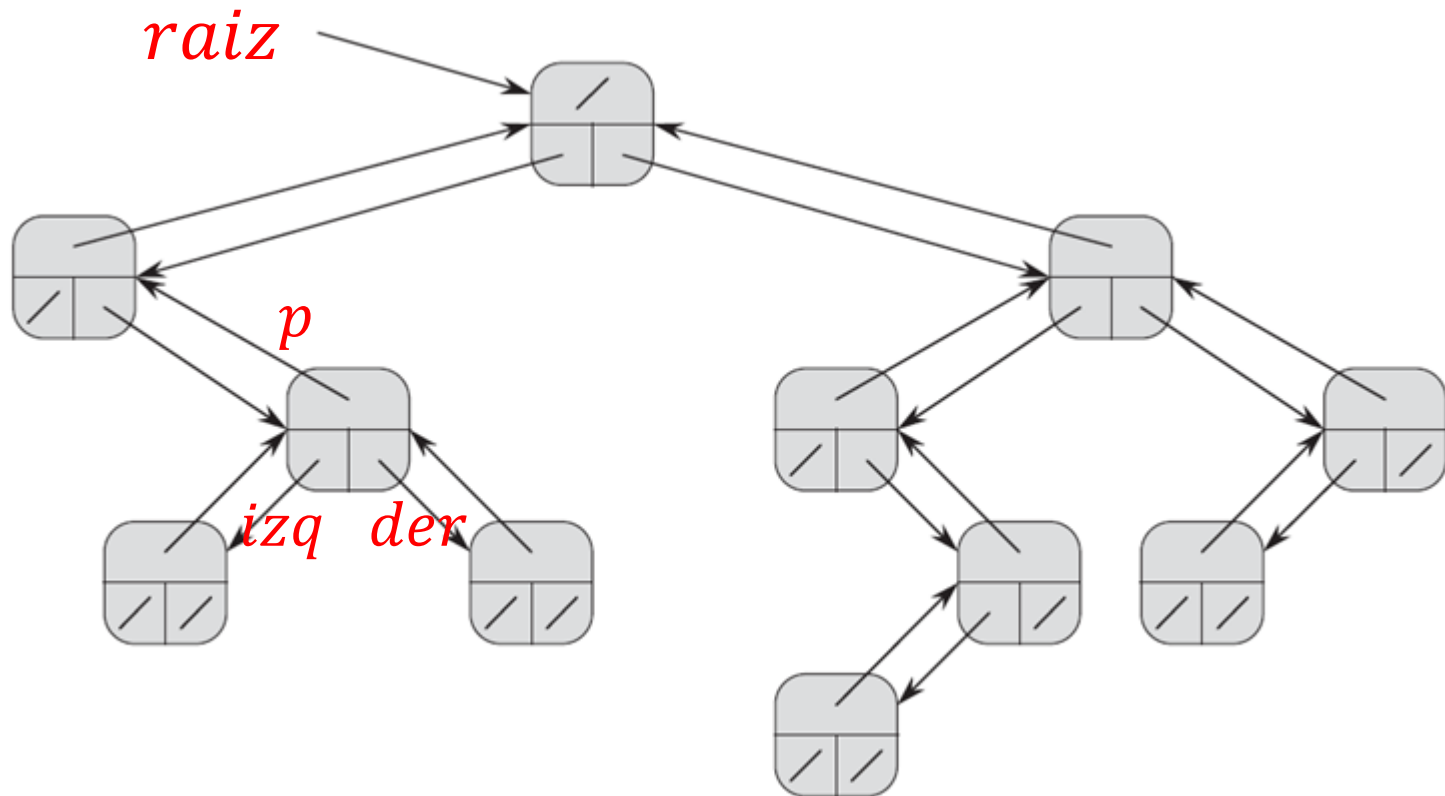
Se usan los siguientes atributos:

- ***p***: indica el padre de un nodo
- ***izq***: indica el hijo izquierdo de un nodo
- ***der***: indica el hijo derecho de un nodo

La raíz del árbol T se referencia (o apunta) mediante el objeto ***raiz***.



# Árboles binarios - Representación

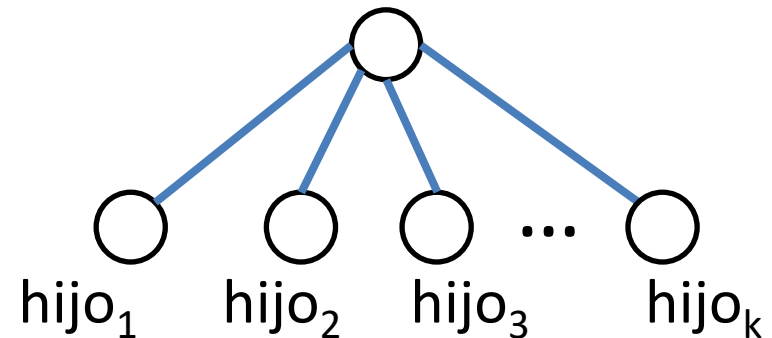
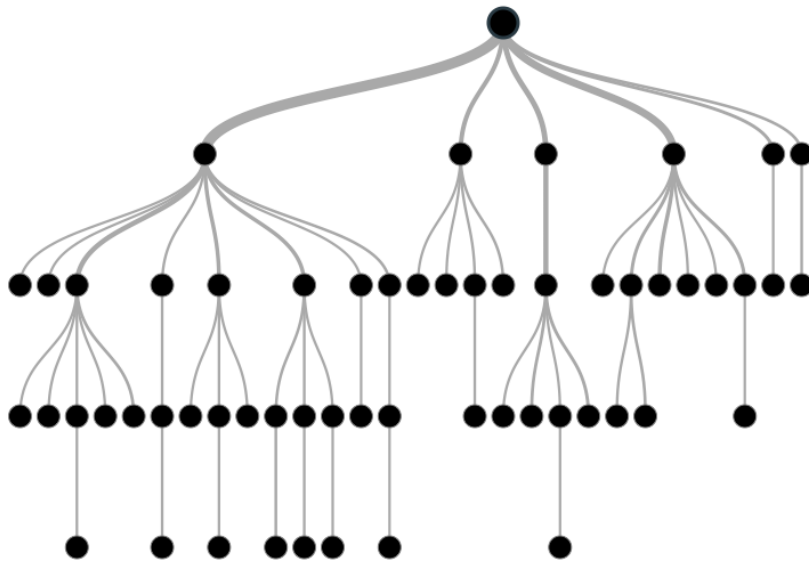


**R: Raíz.**  
**I: Subárbol izquierdo**  
**D: Subárbol derecho**

# Árbol general - Representación

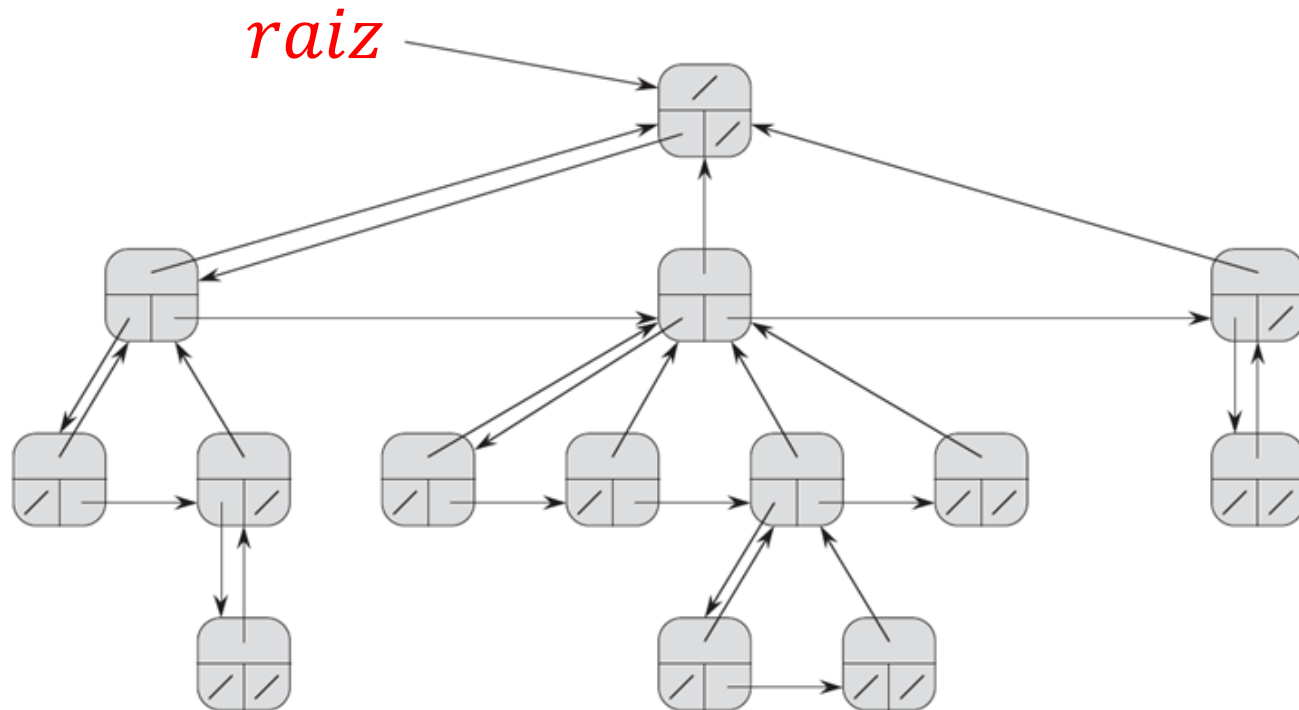
Si la cantidad de hijos de un nodo cualquiera es a lo más una constante  $k$ , se puede usar un esquema de representación similar al empleado para un árbol binario.

Se rempazan *izq* y *der* por  $hijo_1, hijo_2, \dots, hijo_k$ .



# Árbol general - Representación

Existe una forma más inteligente de representar árboles con un número arbitrario de hijos, que se denomina **representación hijo-izquierdo, hermano-derecho**. Para cada nodo, sus hijos se almacenan en una lista enlazada. En lugar de tener una referencia por cada hijo, se tienen sólo dos referencias .

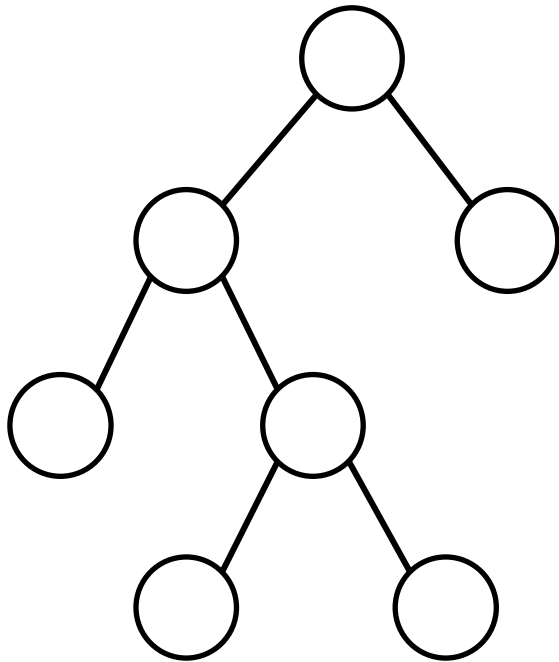


*x. izq. hijo*  $\Rightarrow$  hijo de  $x$  más a la izquierda.

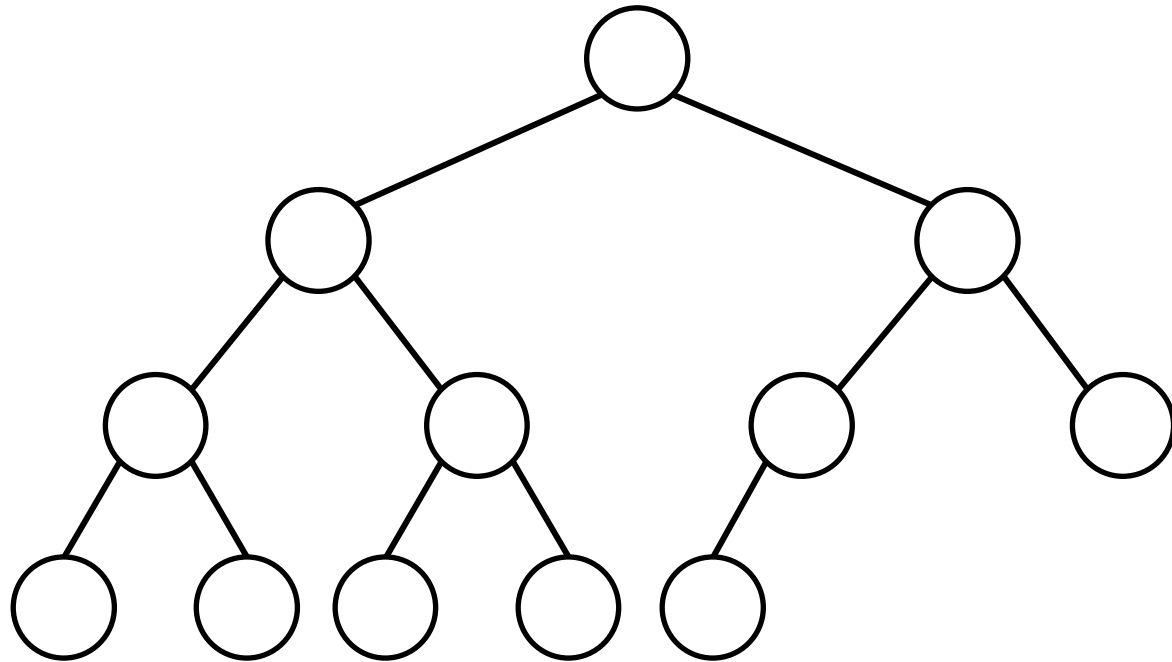
*x. der. hermano*  $\Rightarrow$  hermano de  $x$  más cercano al lado derecho. **14**

# Tipos de árboles binarios

- **Lleno:** todos los nodos tienen exactamente 0 o 2 hijos.
- **Completo:** todos los niveles están completos a excepción del último (que se completa de izquierda a derecha).



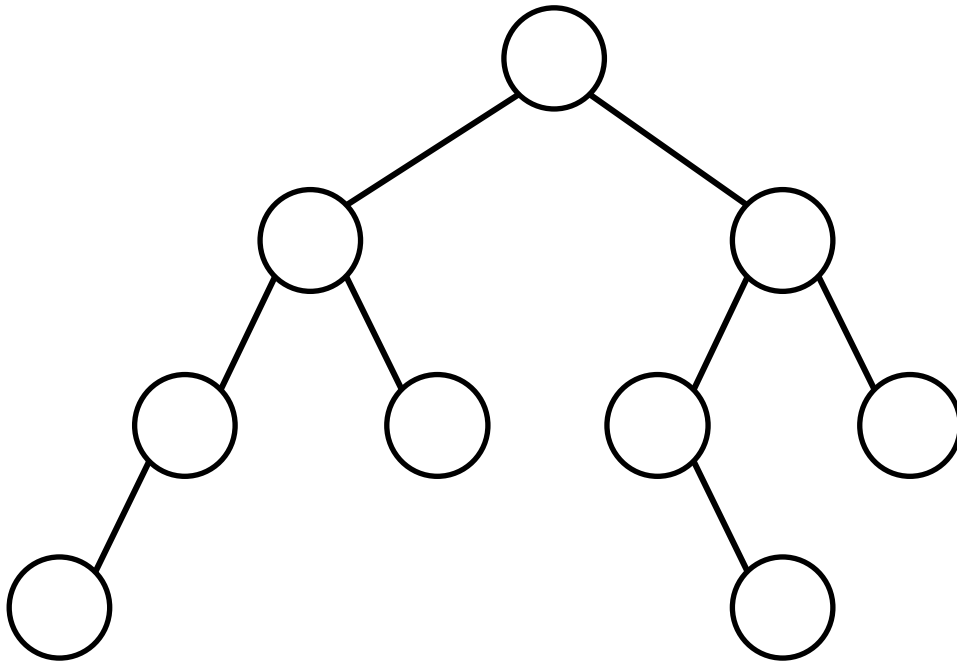
Árbol Binario Lleno



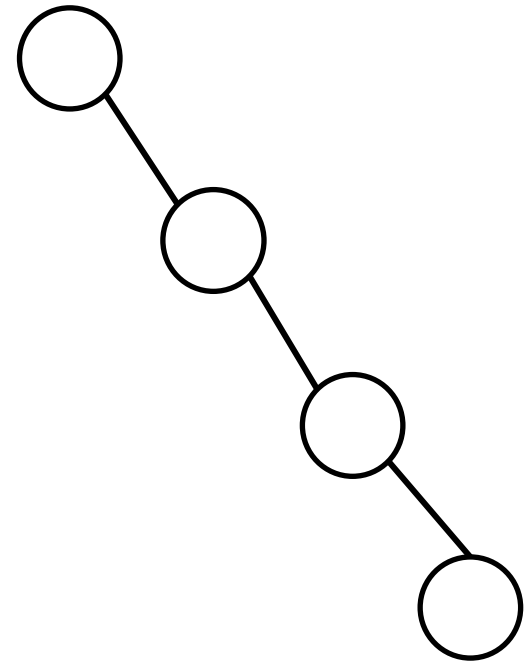
Árbol Binario Completo

# Tipos de árboles binarios

- **Equilibrado:** las alturas de los dos subárboles de cada nodo del árbol se diferencian en una unidad como máximo.
- **Degenerado:** todos sus nodos tienen solamente un subárbol, excepto el último.



Árbol Binario Equilibrado

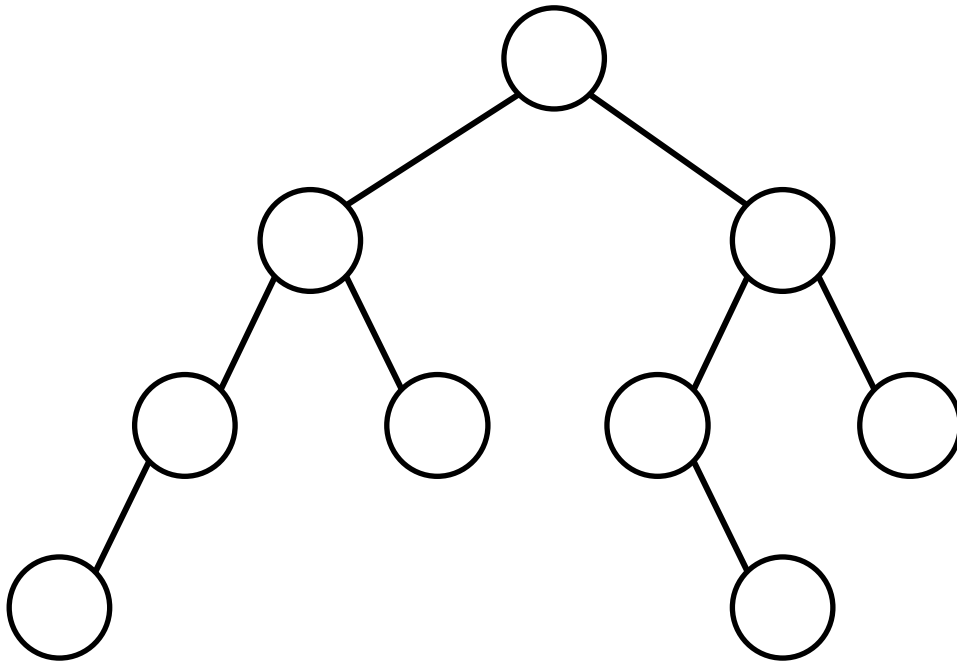


Árbol Binario Degenerado

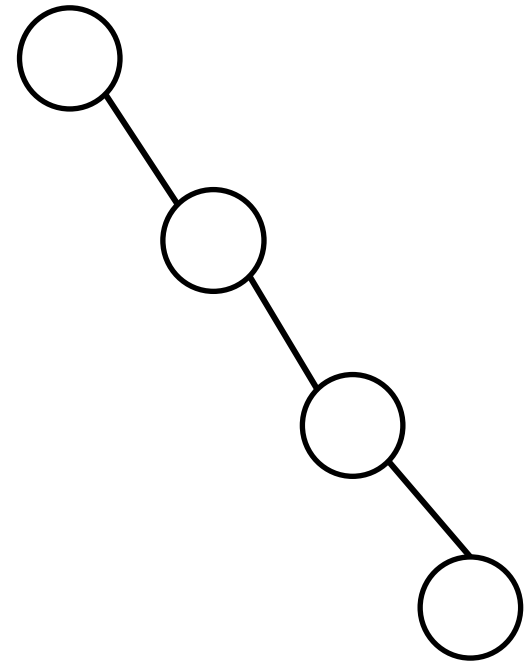


# Tipos de árboles binarios

- **Equilibrado:** las alturas de los dos subárboles de cada nodo del árbol se diferencian en una unidad como máximo.
- **Degenerado:** todos sus nodos tienen solamente un subárbol, excepto el último.



Árbol Binario Equilibrado



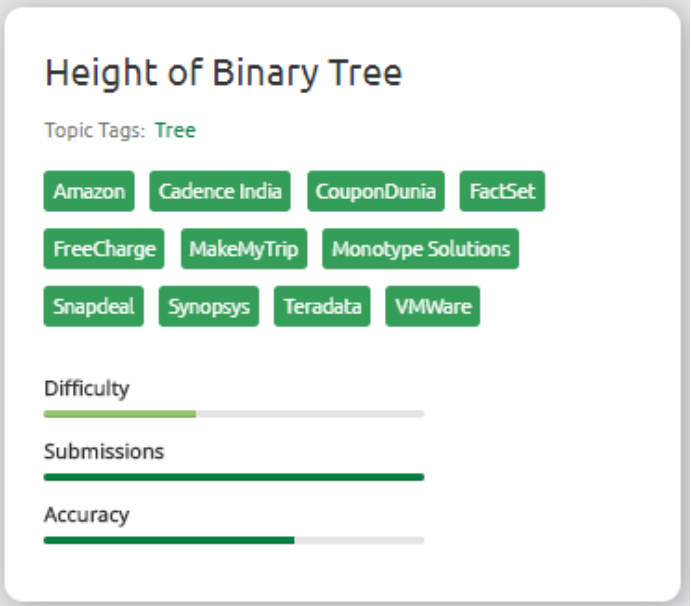
Árbol Binario Degenerado

# Aplicaciones de árboles binarios

- **Árbol binario de búsqueda (BST - *Binary Search Tree*)**: se utiliza en muchas aplicaciones de búsqueda donde los datos entran/salen constantemente, como los objetos map y set en las bibliotecas de muchos idiomas.
- **Árbol de sintaxis**: construido por compiladores y (implícitamente) calculadoras para analizar expresiones
- **Partición de espacio binario**: se utiliza en casi todos los videojuegos 3D para determinar qué objetos deben renderizarse.
- **Montículos (*heaps*)**: se usa para implementar colas de prioridad eficientes, que a su vez se usan para programar procesos en muchos sistemas operativos, calidad de servicio en enrutadores y A\* (algoritmo de búsqueda de ruta usado en aplicaciones de AI incluyendo robótica y videojuegos). También se utiliza en la ordenación del montículo (*heapsort*)
- **Huffman Coding Tree (Chip Uni)**: utilizado en algoritmos de compresión, como los utilizados por los formatos de archivo .jpeg y .mp3.
- **Árboles GGM**: Se usa en aplicaciones criptográficas para generar un árbol de números pseudoaleatorios.

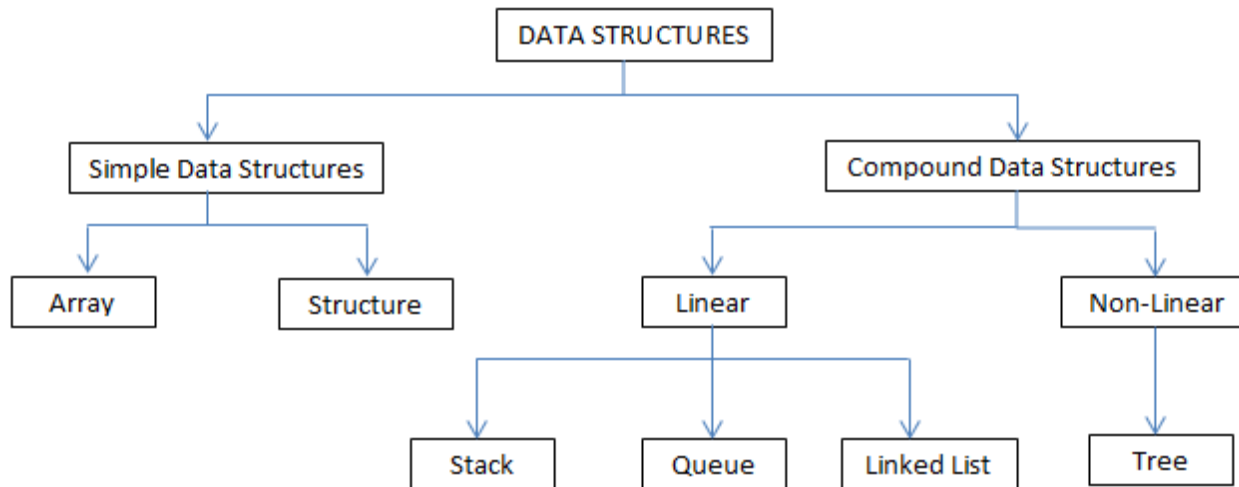
# Ejercicios – Parte 1

- 1- Dado un nodo, calcular su profundidad.
- 2- Encontrar la altura de un árbol binario.
- 3- Obtener la cantidad de elementos de un árbol binario.
- 4- Mostrar los nodos de un árbol binario por niveles.
- 5- Obtener la hoja de mayor valor en un árbol binario.
- 6- Determinar si un árbol binario dado es o no balanceado.
- 7- El diámetro de un árbol está definido como la mayor de las distancias entre todos los pares de vértices del árbol. Dado un árbol binario, encuentre su diámetro.

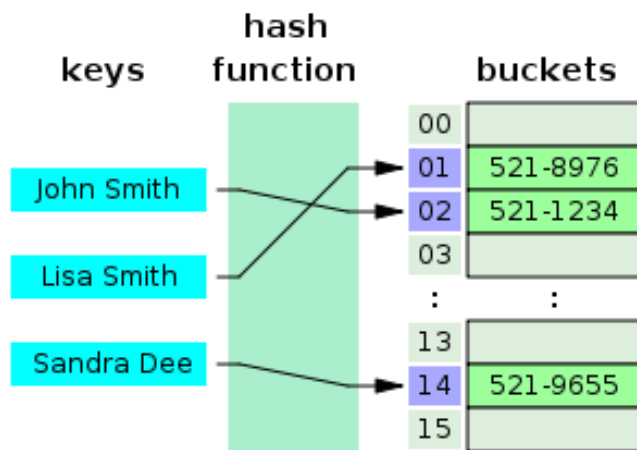


The screenshot shows the LeetCode problem page for 'Height of Binary Tree'. The title is 'Height of Binary Tree'. Below the title, it says 'Topic Tags: Tree'. There are ten tags representing companies: Amazon, Cadence India, CouponDunia, FactSet, FreeCharge, MakeMyTrip, Monotype Solutions, Snapdeal, Synopsys, Teradata, and VMWare. Below the tags, there are three progress bars: 'Difficulty' (approximately 10% green), 'Submissions' (approximately 80% green), and 'Accuracy' (approximately 60% green).

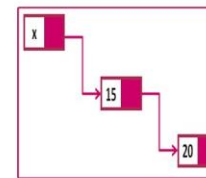
# Recordando: Estructuras de Datos



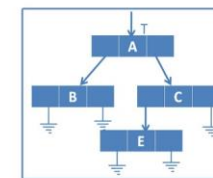
Una estructura de datos es una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen en ella (*acceso, inserción, borrado*).



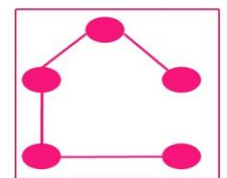
Sorting



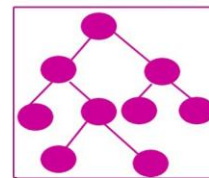
Link list



list



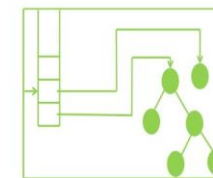
spanning tree



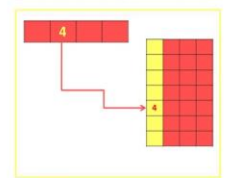
Tree



Graph



Stack



Hashing

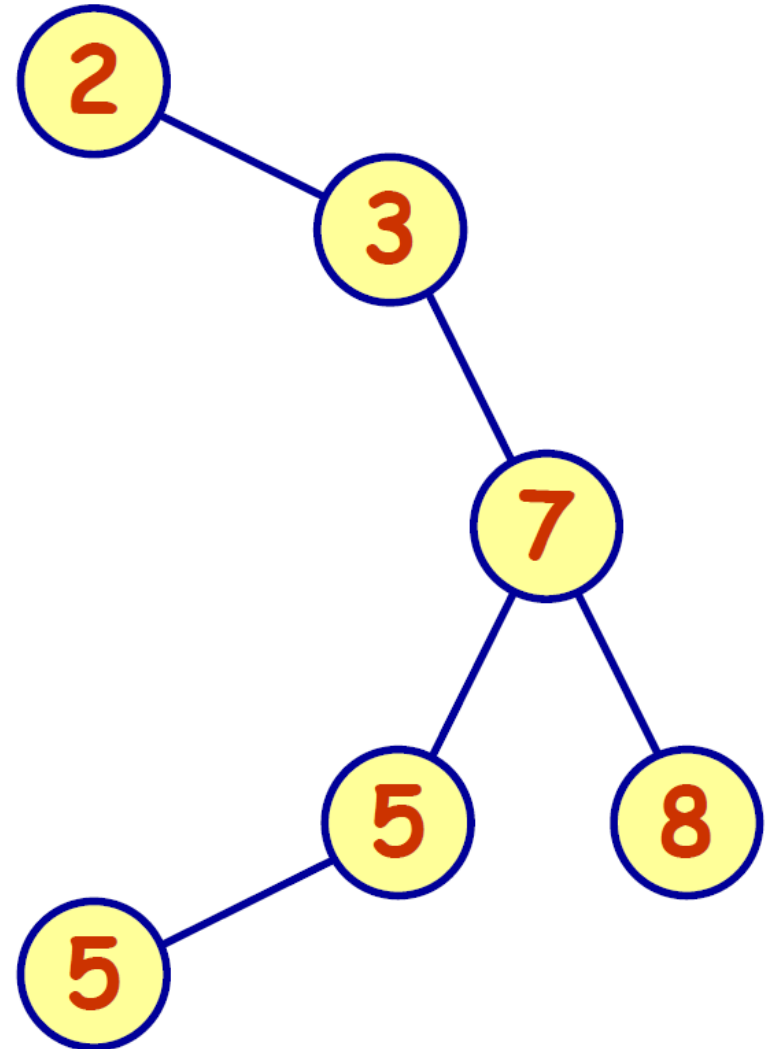
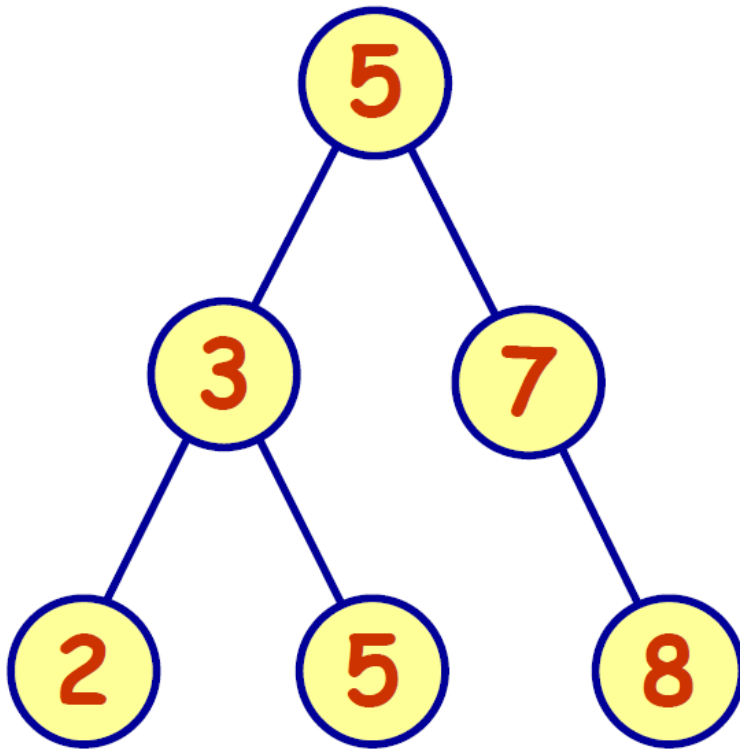
# Aplicación: árbol binario de búsqueda

En un arreglo desordenado, se debe hacer un recorrido lineal para la **búsqueda** de un elemento determinado. Se podría ordenar el arreglo y hacer una búsqueda binaria, pero no se consideran las **inserciones** y **borrados** de elementos (aunque en Python esto no es problema). El **árbol binario de búsqueda** (BST por sus siglas en inglés) es un árbol binario que da soluciones a estos problemas.

Cada nodo  $x$  del BST cumple con la siguiente propiedad:

- Si  $y$  es un nodo en el subárbol izquierdo de  $x$ , entonces  $y.clave \leq x.clave$ .
- Si  $y$  es un nodo en el subárbol derecho de  $x$ , entonces  $y.clave > x.clave$ .

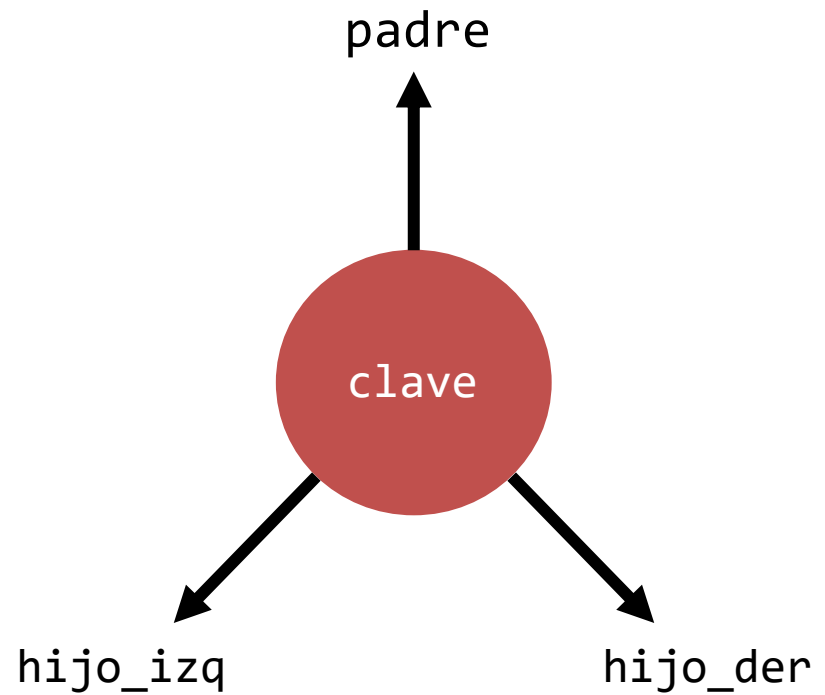
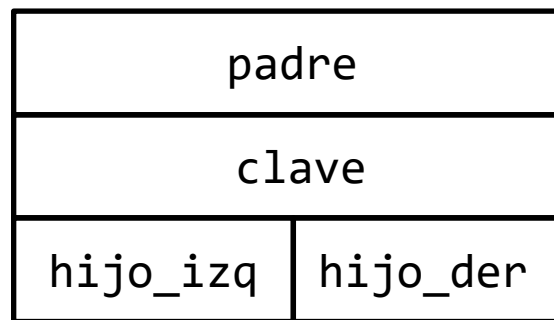
# Ejemplos de BST



# Nodo del BST – Definición del nodo

Cada nodo o vértice contará con su información interna (clave) y con tres objetos (punteros a): el padre, el hijo izquierdo, y el hijo derecho. Inicialmente los objetos tendrán un valor **None** (NULL en C/C++).

Nodo

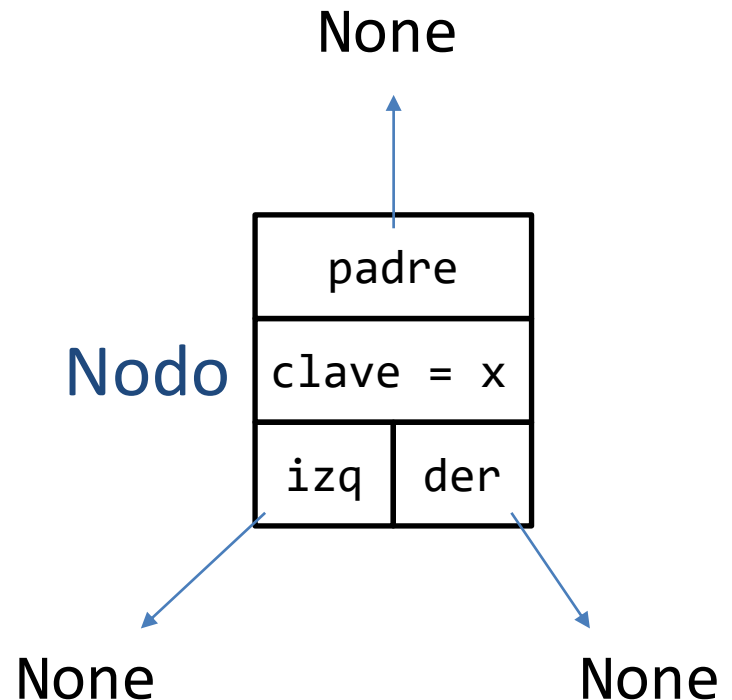


# Nodo del BST – Definición del nodo

Cuando se cree un nodo, se cargará el dato recibido como argumento (x), y los tres punteros tendrán un valor None.

```
class Nodo:
    def __init__(self,x):
        self.padre = None
        self.hijo_izq = None
        self.hijo_der = None
        self.clave = x

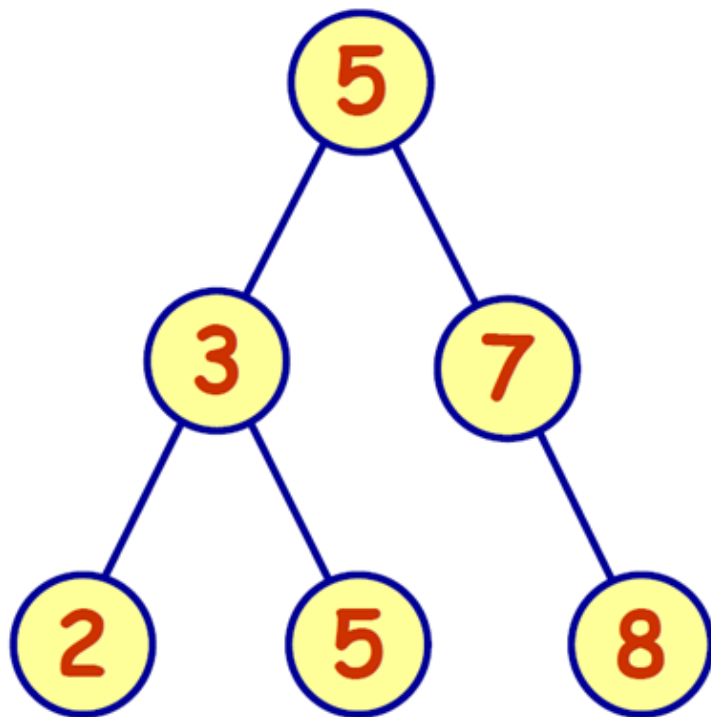
class ArbolBST:
    def __init__(self):
        self.raiz = None
```





# Búsqueda, inserción y borrado

La búsqueda no altera la composición de un BST, simplemente lo recorre para buscar un elemento dado. Pero para la inserción y el borrado, la estructura se debe modificar para que se siga cumpliendo la propiedad del árbol de búsqueda binario. La inserción es relativamente sencilla comparada con el borrado.



**¿Dónde podría insertar el número 6? Y el 4?**

Insert a node in a BST

Topic Tags: Binary Search Tree

Amazon

Microsoft

Difficulty



Submissions

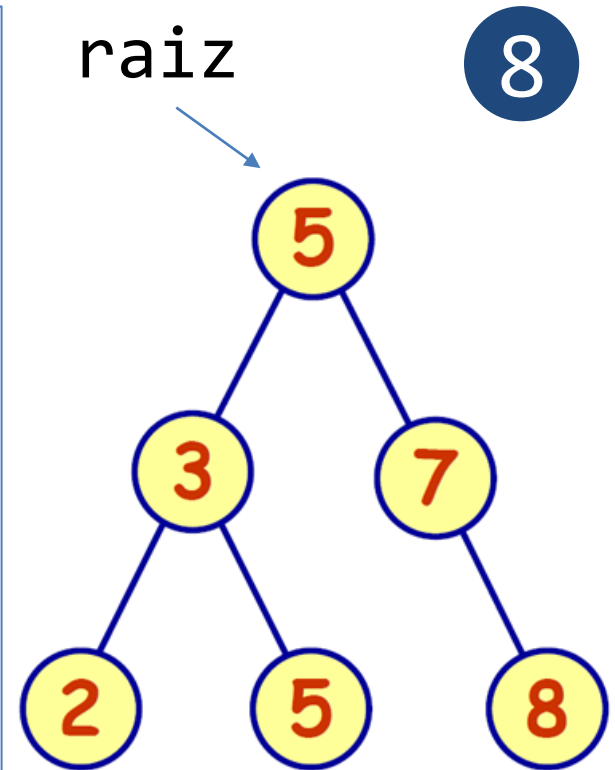


Accuracy

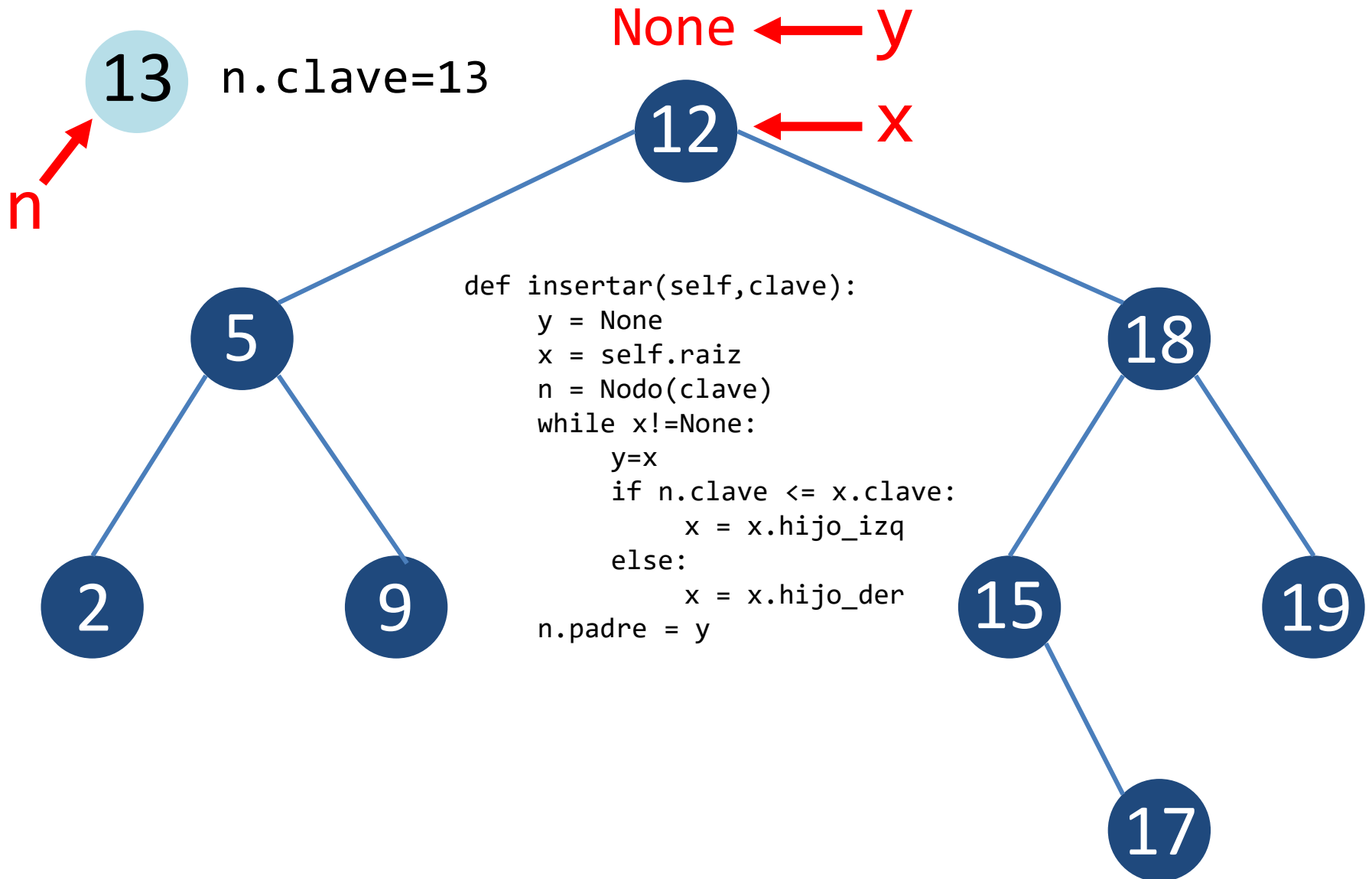


# Algoritmo de inserción

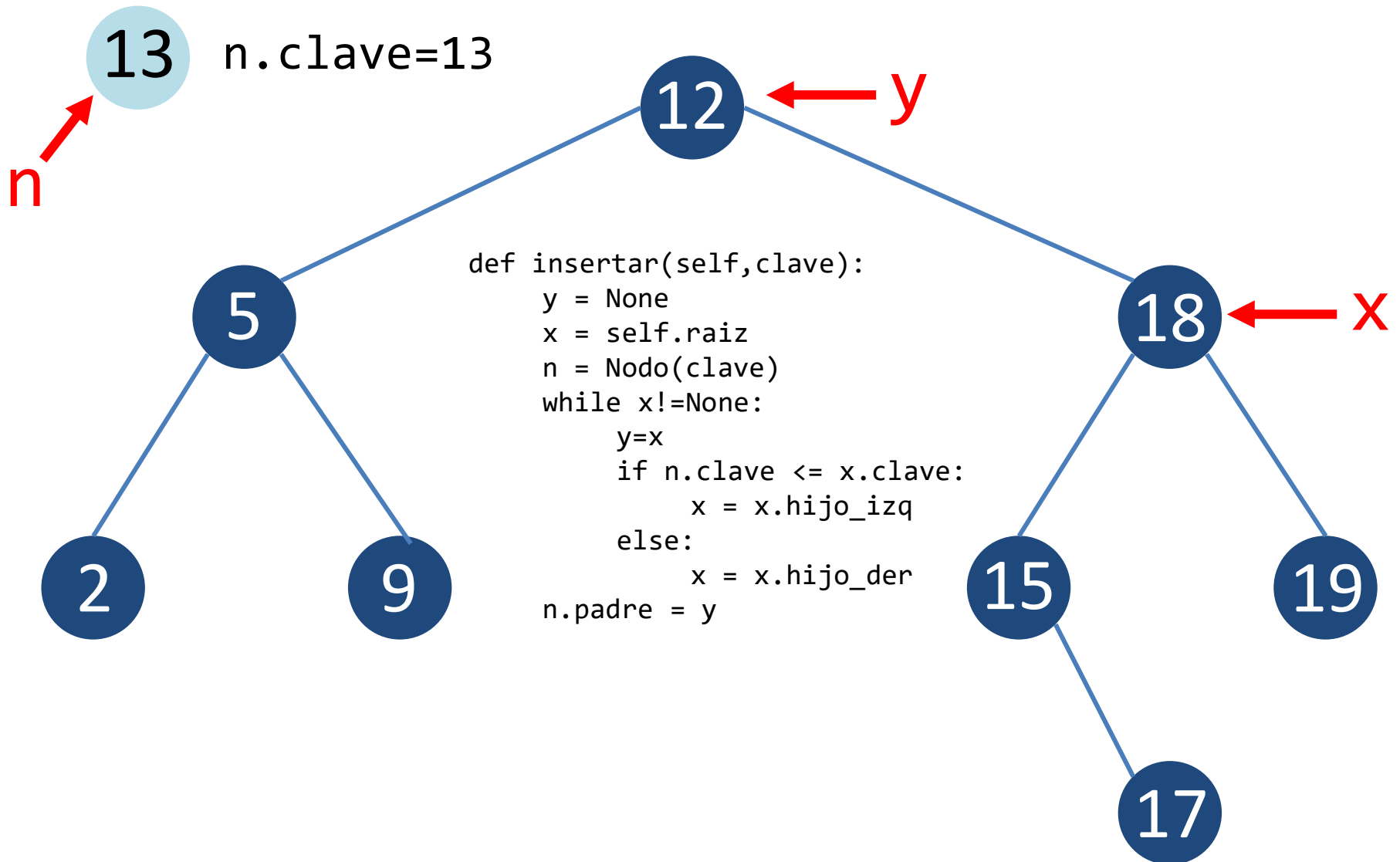
```
def insertar(self,clave):  
    y = None  
    x = self.raiz  
    n = Nodo(clave) #nodo a ser agregado  
    while x!=None:  
        y=x  
        if n.clave <= x.clave:  
            x = x.hijo_izq #va hacia la izq  
        else:  
            x = x.hijo_der #va hacia la der  
    n.padre = y  
    if y==None: #estamos en la raiz  
        self.raiz = n  
    else:  
        if n.clave <= y.clave:  
            y.hijo_izq = n  
        else:  
            y.hijo_der = n
```



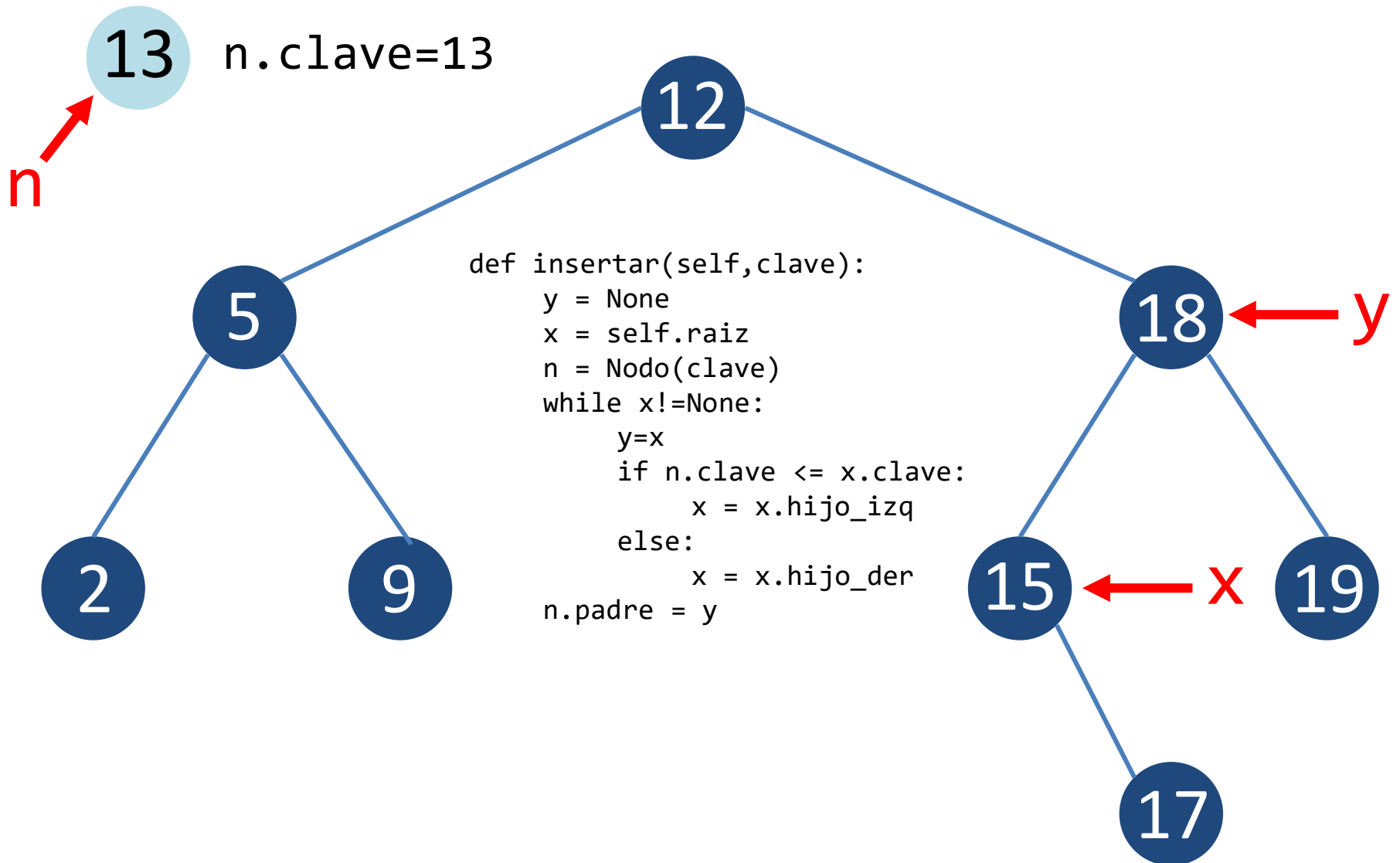
# Algoritmo de inserción - Ejemplo



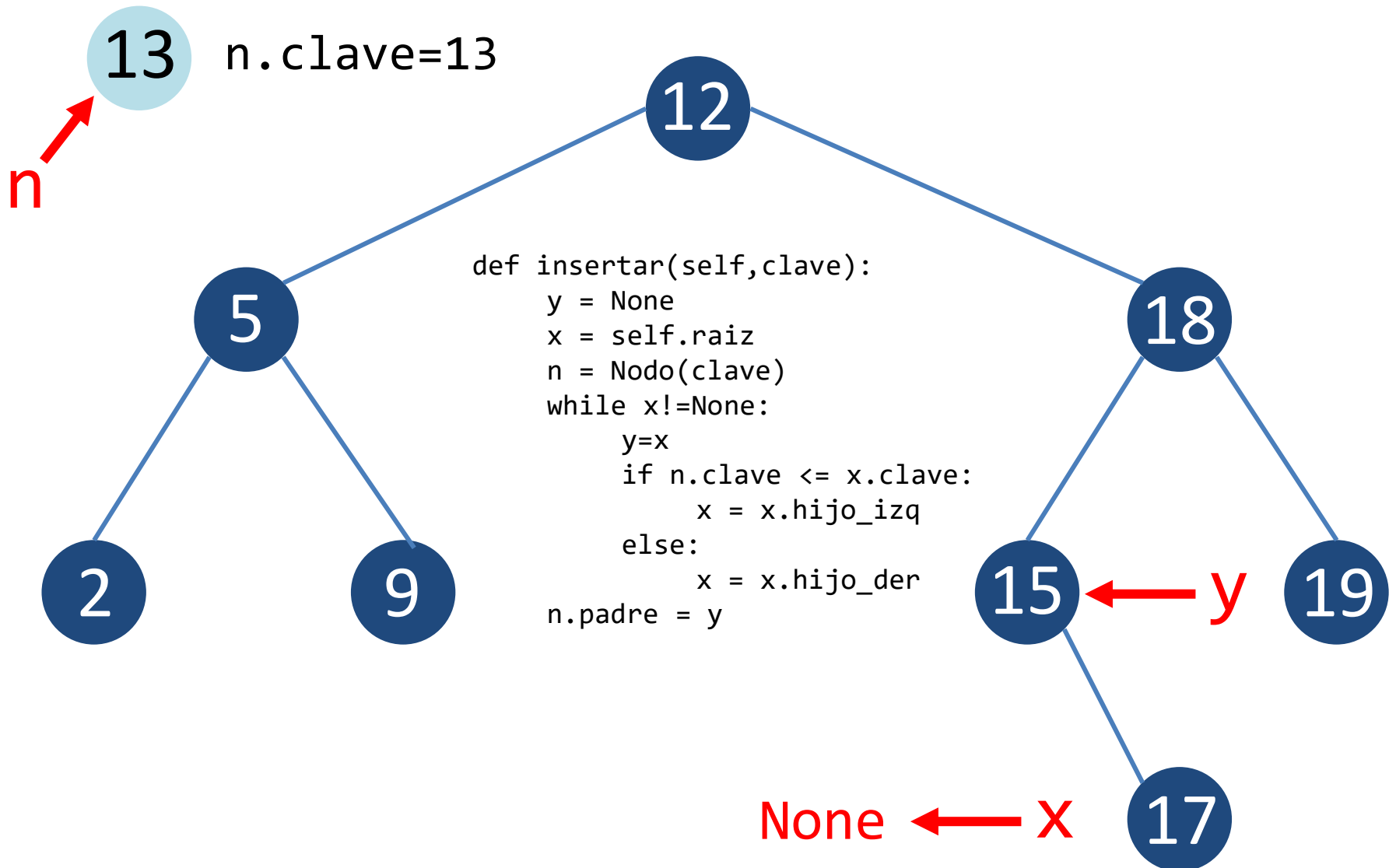
# Algoritmo de inserción - Ejemplo



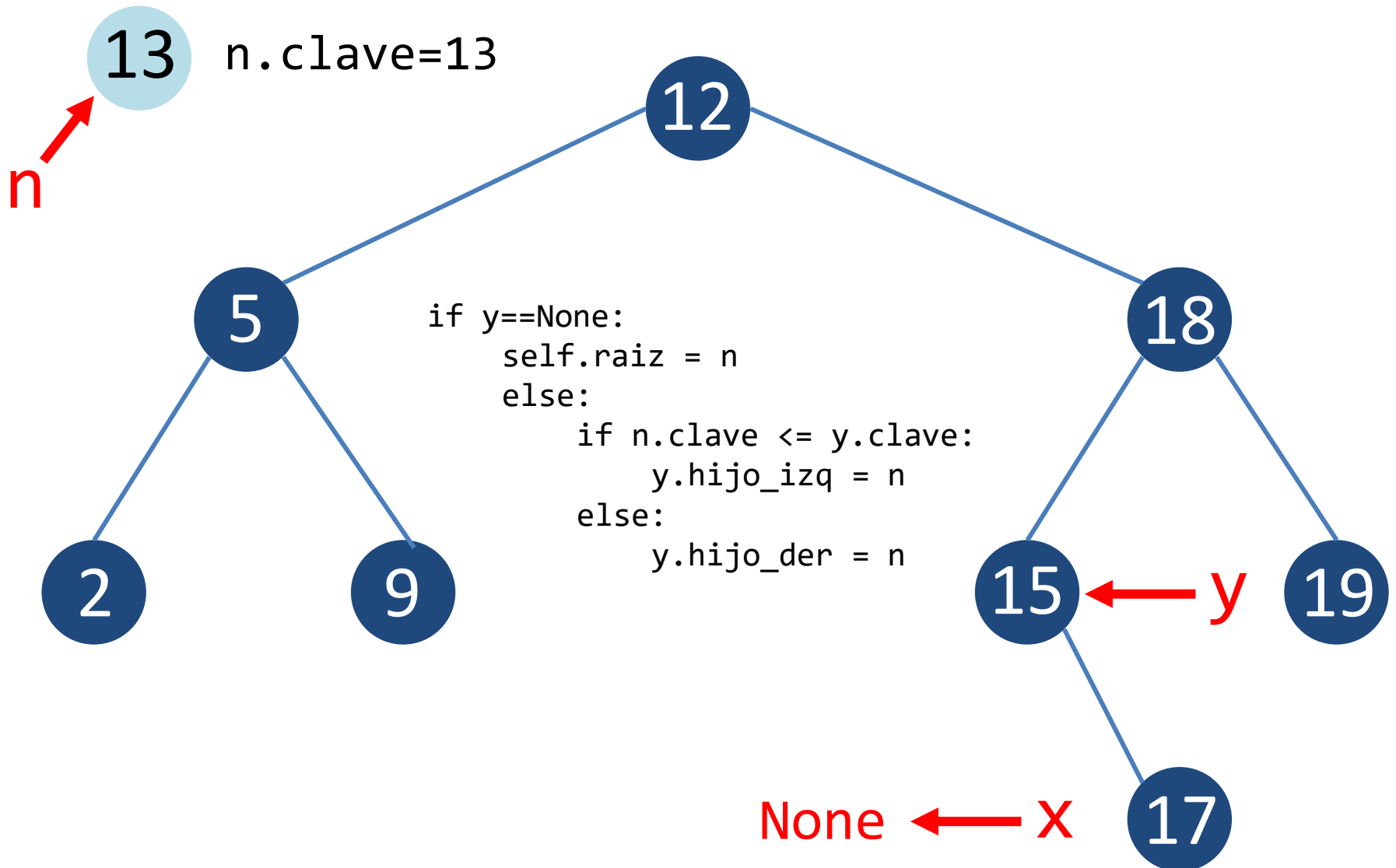
# Algoritmo de inserción - Ejemplo



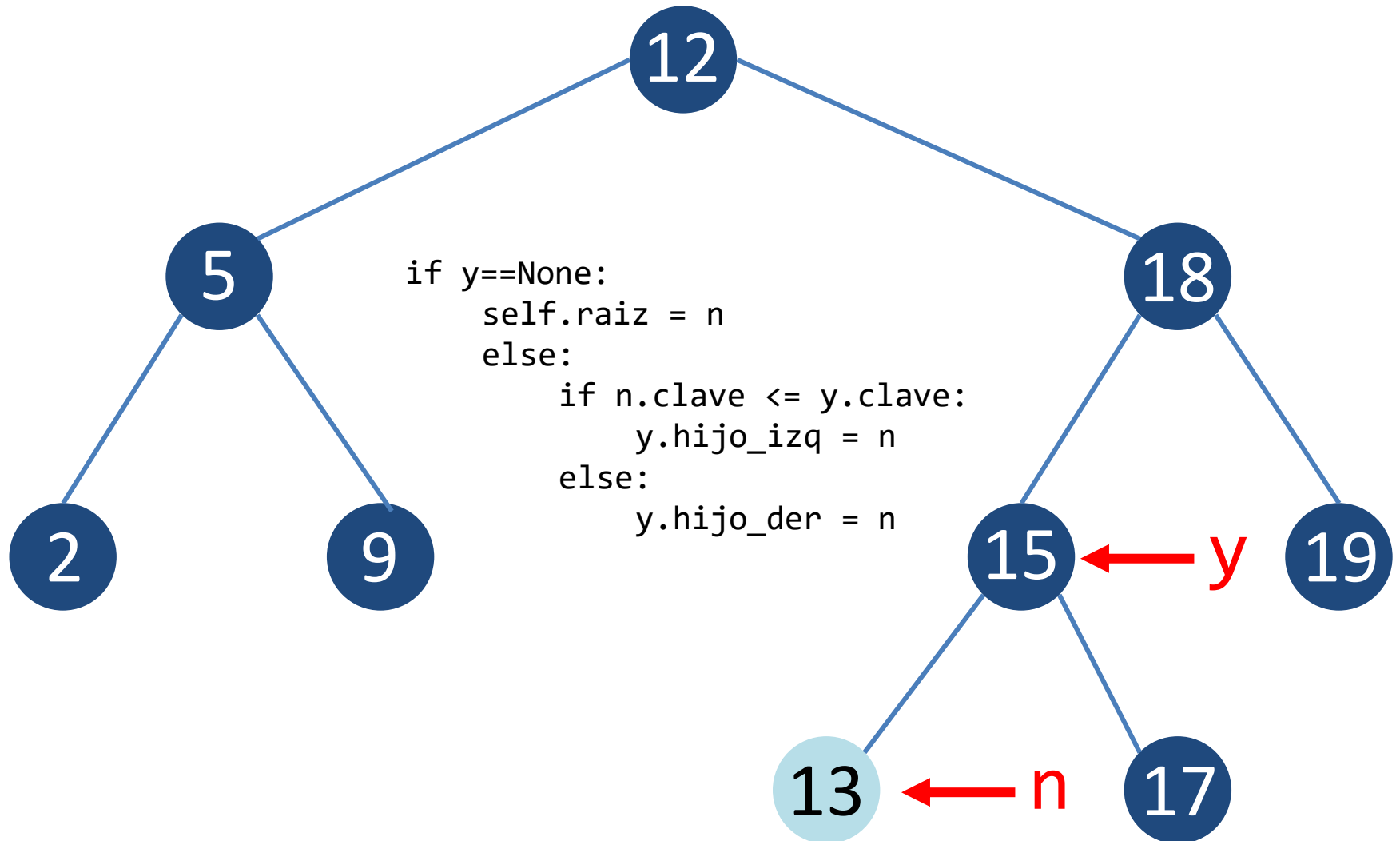
# Algoritmo de inserción - Ejemplo



# Algoritmo de inserción - Ejemplo



# Algoritmo de inserción - Ejemplo





# Ejercicio grupal

Construir el árbol binario de búsqueda (BST) para los siguientes elementos:  $\{5, 1, 4, 7, 8, 2, 9, 0, 3, 6\}$

# Construcción de un árbol – Programa

```
def crearArbol(lista):  
    arb = ArbolBST()  
    for i in lista:  
        arb.insertar(i)  
    print("Se creó un árbol con",len(lista),"elementos.")  
    return arb
```

Nota: igual se pueden agregar más nodos luego de esta operación.

```
► #Test - crearArbol  
lista2 = [5,3,7,2,5,8,6,7,11]  
arbol2 = crearArbol(lista2)  
print(arbol2)
```

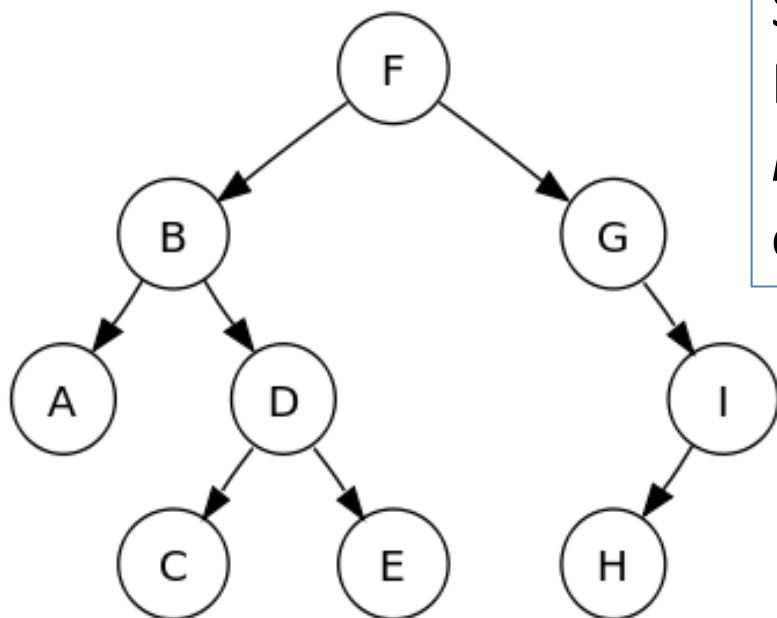
Se creó un árbol con 9 elementos.

Datos del arbol

(0)	Nodo 5 --->	padre: Ninguno	h_izq: 3	h_der: 7
(1)	Nodo 3 --->	padre: 5	h_izq: 2	h_der: 5
(1)	Nodo 7 --->	padre: 5	h_izq: 6	h_der: 8
(2)	Nodo 2 --->	padre: 3	h_izq: Ninguno	h_der: Ninguno
(2)	Nodo 5 --->	padre: 3	h_izq: Ninguno	h_der: Ninguno
(2)	Nodo 6 --->	padre: 7	h_izq: Ninguno	h_der: 7
(2)	Nodo 8 --->	padre: 7	h_izq: Ninguno	h_der: 11
(3)	Nodo 7 --->	padre: 6	h_izq: Ninguno	h_der: Ninguno
(3)	Nodo 11 --->	padre: 8	h_izq: Ninguno	h_der: Ninguno

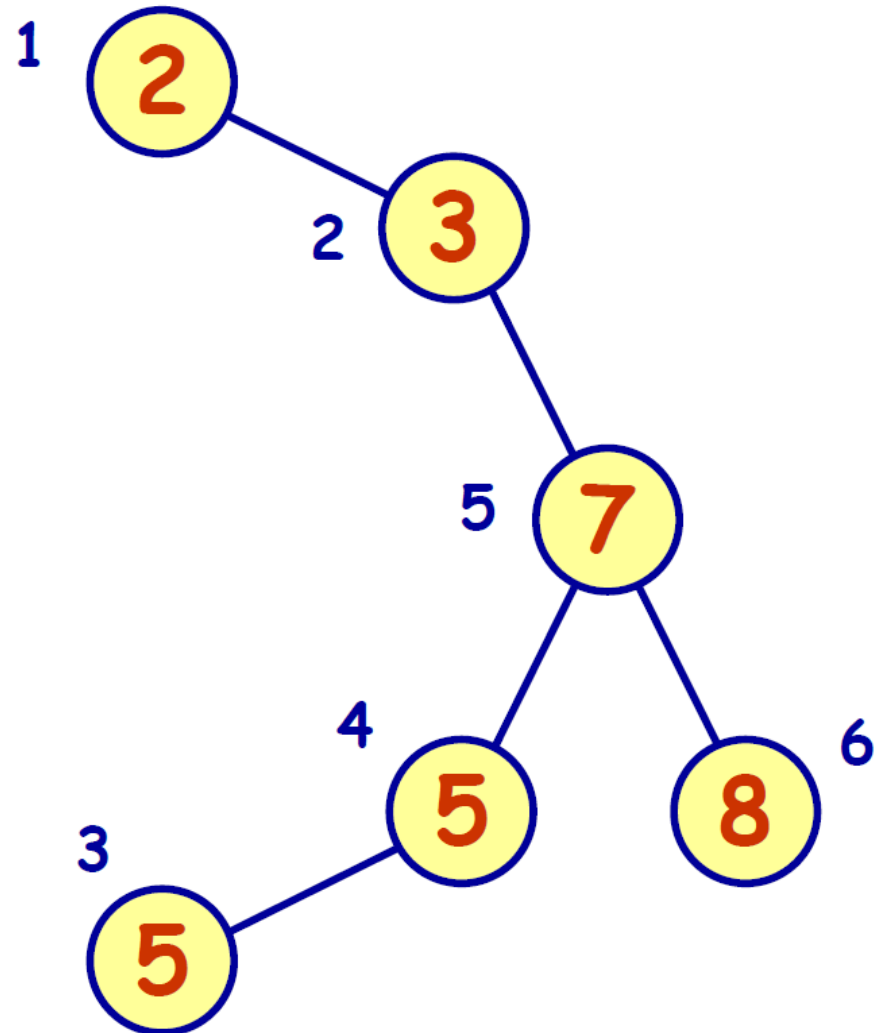
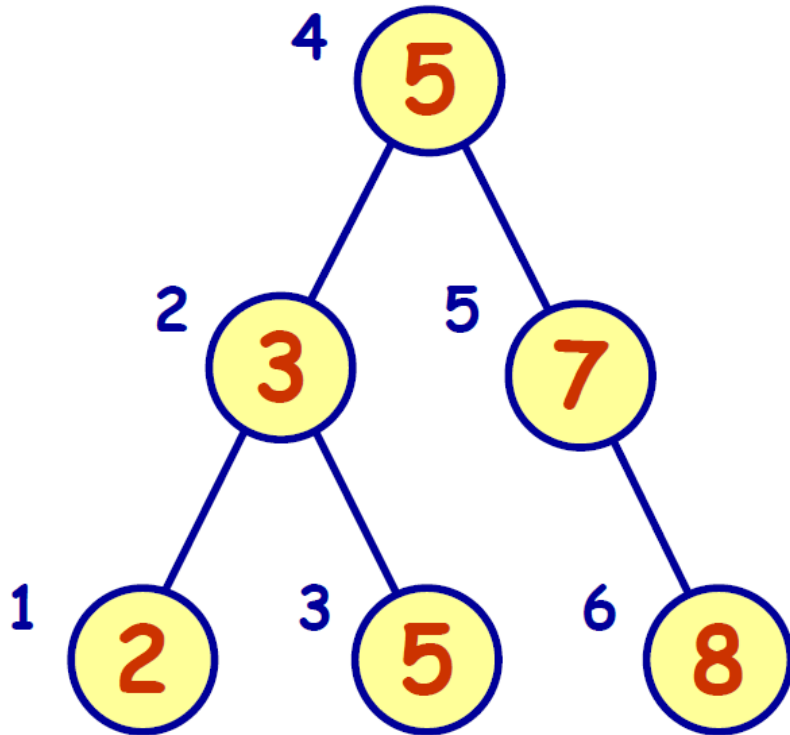
# Recorrido *inorder*

Un **recorrido *inorder*** en un árbol binario se realiza de la siguiente manera: Primero se recorren los vértices del subárbol izquierdo, seguido de la raíz del árbol y finalmente los nodos del subárbol derecho. El recorrido en cada subárbol es el mismo en forma ***recursiva***.



Si se imprimen las claves de un BST siguiendo un recorrido *inorder*, las claves aparecen ordenadas en forma ascendente.

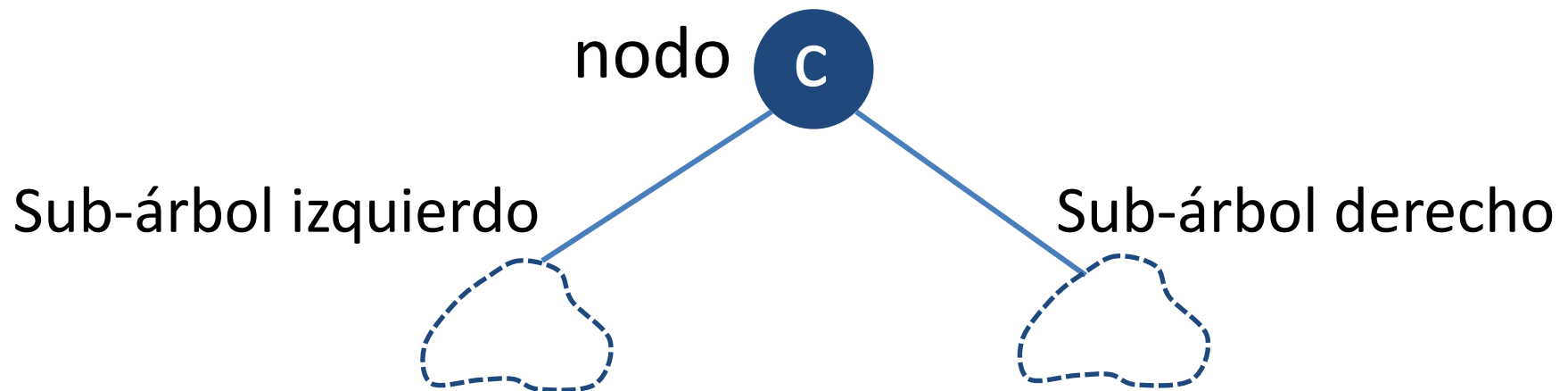
# Recorrido inorder - Ejemplo



# Recorrido **inorder**

El programa recursivo es el siguiente:

```
def recorridoInOrder(nodo):  
    if nodo!=None:  
        recorridoInOrder(nodo.hijo_izq) #sub-árbol izq  
        print(nodo.clave,end=" ") #imprime la clave  
        recorridoInOrder(nodo.hijo_der) #sub-árbol der
```



# Recorridos preorder y postorder

Un **recorrido postorder** en un árbol consiste en recorrer los subárboles de la raíz  $r$  de izquierda a derecha, seguido de  $r$  ; se aplica un procedimiento recursivo para cada subárbol.

### Preorder to Postorder

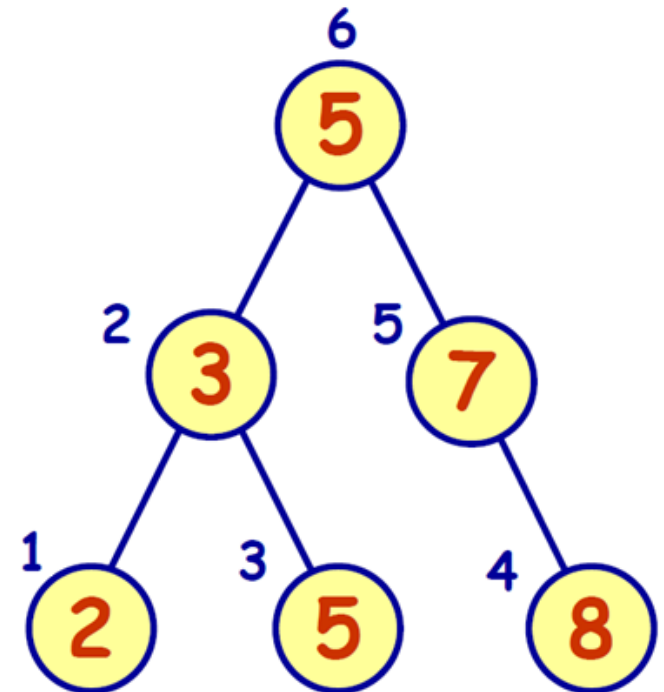
Topic Tags: [Binary Search Tree](#) [Stack](#) [Tree](#)

[Amazon](#)

Difficulty

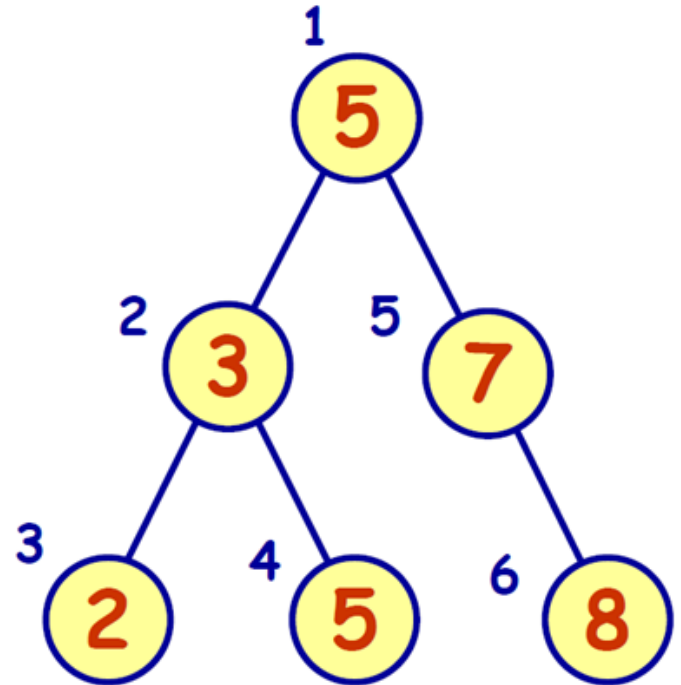
Submissions

Accuracy



# Recorridos preorder y postorder

Un **recorrido preorder** en un árbol consiste en recorrer los vértices uno por uno empezando por la raíz, después por los vértices del subárbol izquierdo, y finalmente por los del subárbol derecho. Para recorrer cada uno de los subárboles se sigue el mismo procedimiento recursivamente.



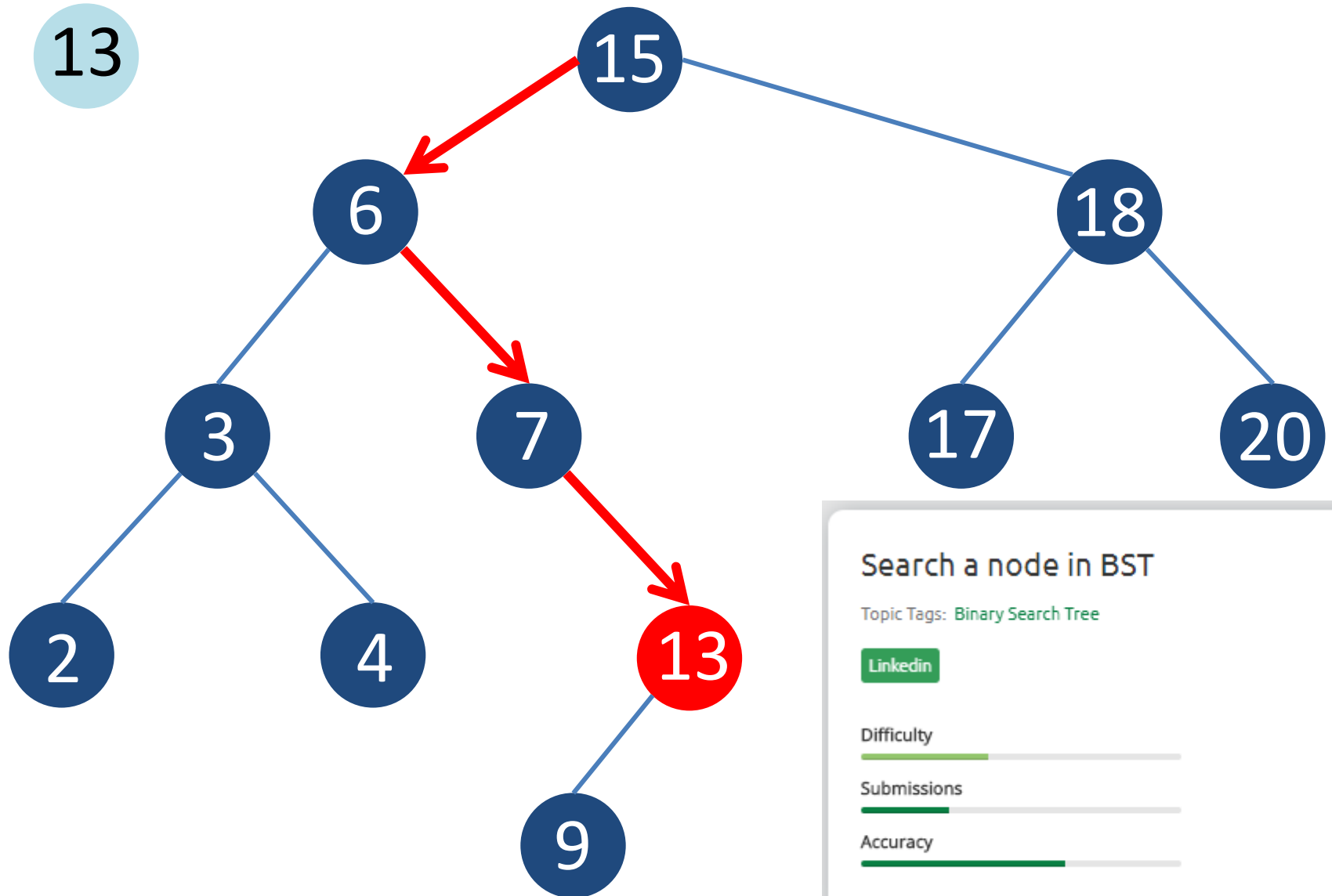
# Búsqueda de una clave

Dado un BST y un ítem con clave  $k$ , la función `buscar(k)` regresa el nodo con la clave  $k$  si el mismo existe; en caso contrario, entrega un **None**.

```
def buscar(self,x):
    nodo = self.raiz #empezamos en la raiz
    while nodo!=None and nodo.clave!=x:
        if x<nodo.clave:
            nodo = nodo.hijo_izq #sub izq
        else:
            nodo = nodo.hijo_der #sub der
    return nodo
```

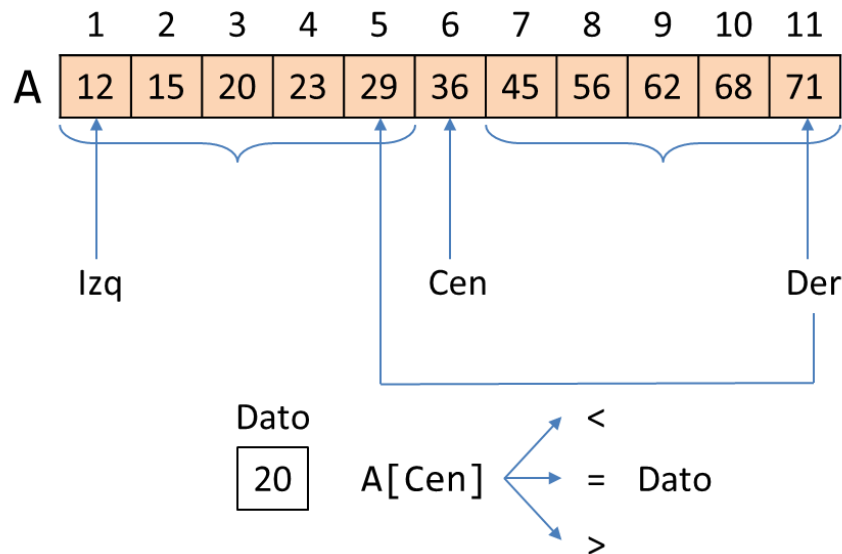


# Búsqueda de una clave - Ejemplo



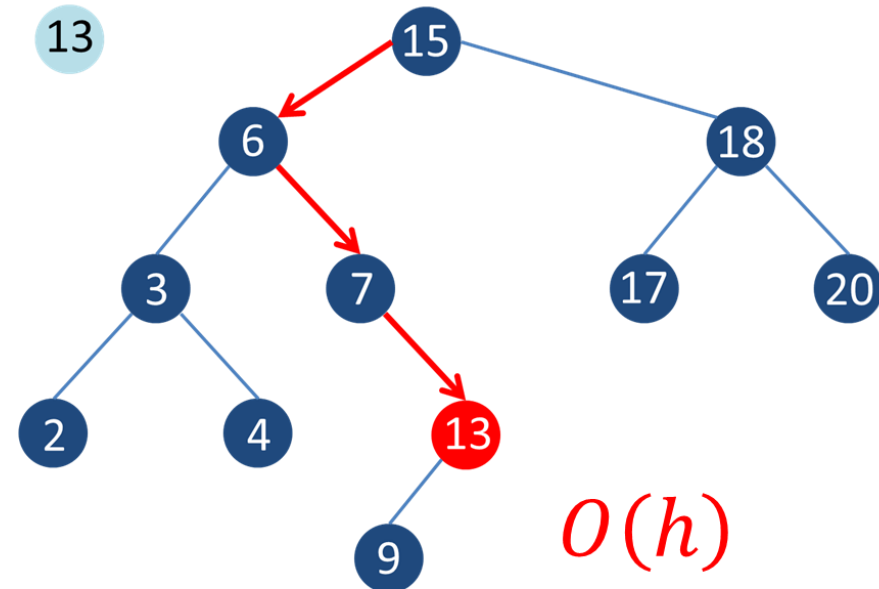
# Comparación

## Búsqueda binaria



$O(\log N)$

## Búsqueda en BST



Si el árbol es balanceado, entonces la altura  $h$  es  $O(\log N)$

Búsqueda secuencial:  $O(N)$

# Comparación

```
In [15]: ► %%time  
          #Busqueda de elementos con recorrido secuencial  
          for x in consultas:  
              busquedaSec(x,lista)
```

Wall time: 23.2 s

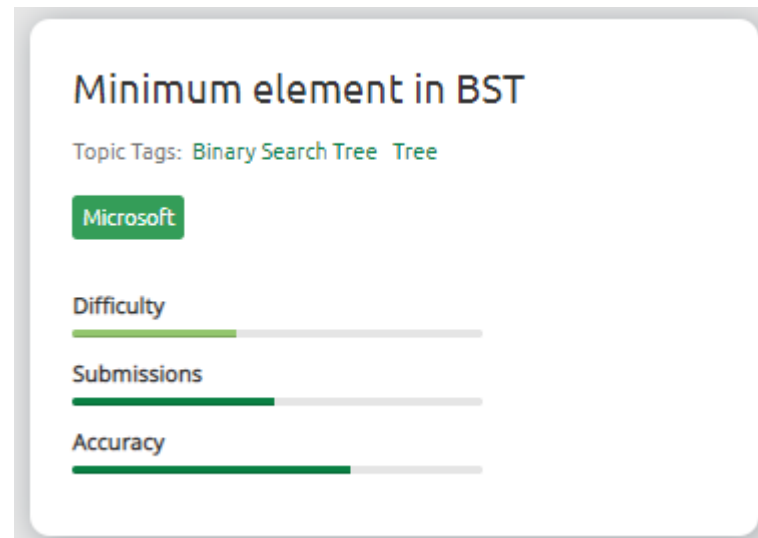
```
In [16]: ► %%time  
          #Busqueda de elementos con un BST  
          for x in consultas:  
              arbolBal.buscar(x)
```

Wall time: 130 ms

\*Realizando  $n = 2^{15} - 1$  consultas en un conjunto de  $n$  números

# Ejercicios – Parte 2

- 1- Encontrar el elemento máximo de un BST. La función debe retornar la referencia al nodo con ese elemento.
- 2- Encontrar el elemento mínimo de un BST. La función debe retornar la referencia al nodo con ese elemento.



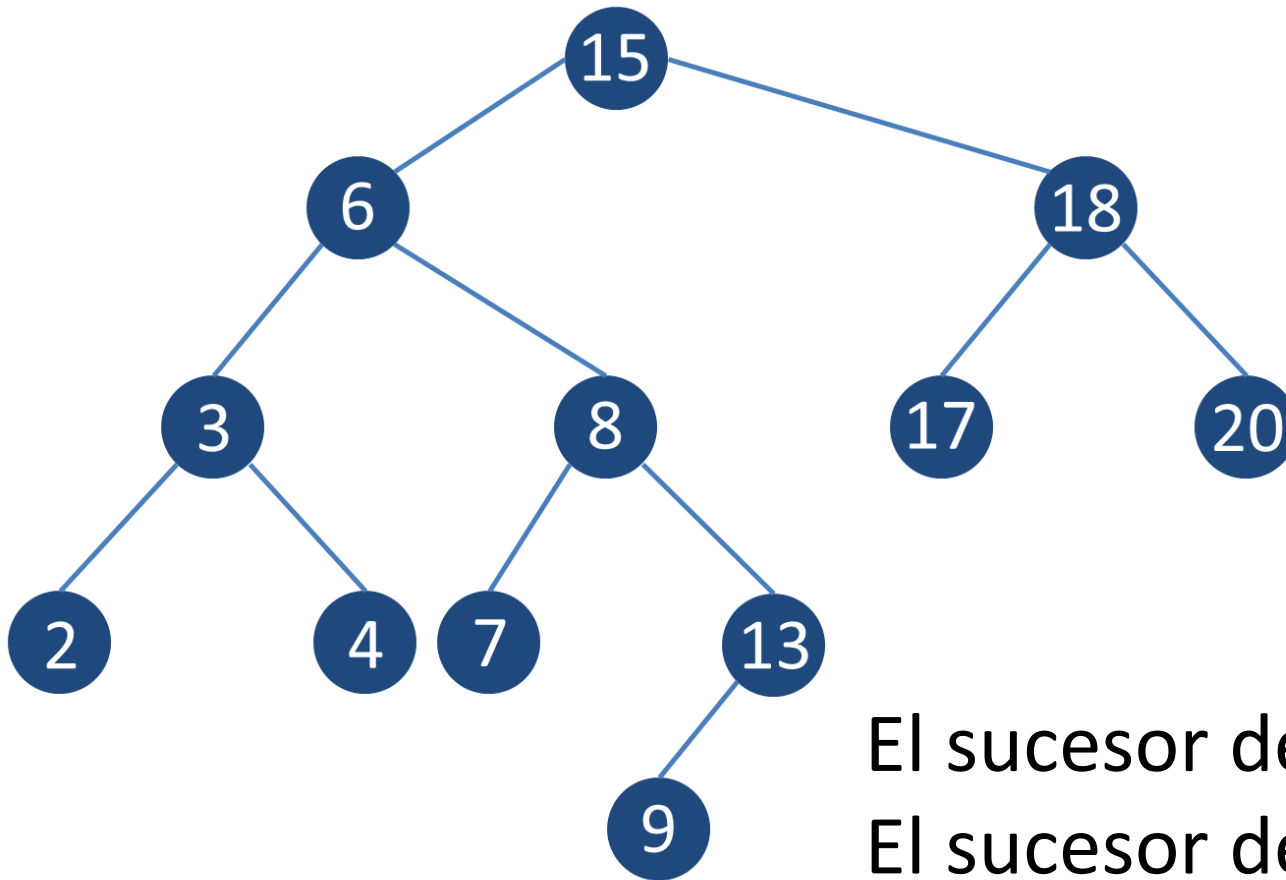
# Sucesor de un nodo

El **sucesor** de un nodo en BST es el siguiente nodo siguiendo un recorrido *inorder*.

- El sucesor de  $x$  (si existe) es el nodo más izquierdo del subárbol derecho de  $x$ ; este nodo es el mínimo en el subárbol enraizado en el hijo derecho de  $x$ .
- Si el subárbol derecho no existe, el sucesor de  $x$  (si existe) es el ancestro más bajo de  $x$  cuyo hijo izquierdo sea también ancestro de  $x$ .

El predecesor de  $x$  se obtiene de una forma parecida.

# Sucesor de un nodo - Ejemplos



El sucesor del 6 es el 7  
El sucesor del 4 es el 6  
El sucesor del 13 es el 15  
El sucesor del 15 es el 17

# Sucesor de un nodo - Programa

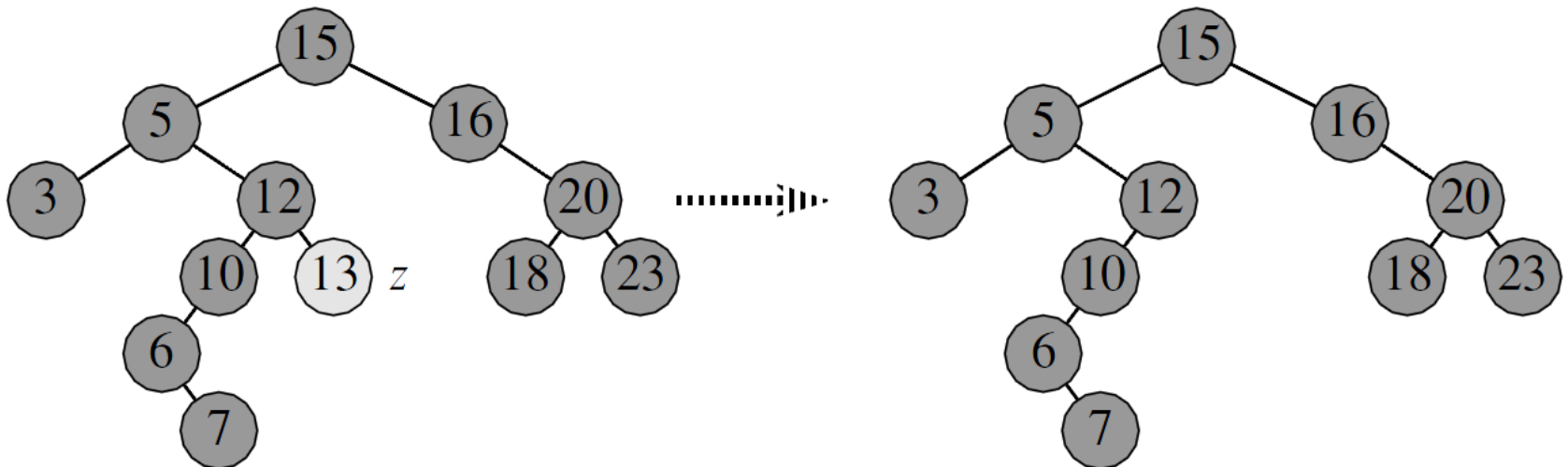
```
def obtenerSucesor(self,nodo):  
    #Si tiene sub-árbol derecho  
    if nodo.hijo_der!=None:  
        return self.obtenerMinimo(nodo.hijo_der)  
  
    #Si no lo tiene  
    y = nodo.padre  
    while y!=None and nodo is y.hijo_der:  
        nodo=y  
        y=y.padre  
    return y
```

# Borrado de un nodo

El procedimiento **borrarNodo** borra un nodo  $z$  del BST. Este procedimiento tiene tres casos:

## Caso 1

Cuando el nodo  $z$  no tiene hijos. En este caso sólo se reemplaza en  $z.padre$  al puntero  $hijo_izq$  o  $hijo_der$  por **None**, dependiendo de donde se encontraba  $z$ .



## Delete a node from BST

Topic Tags: Binary Search Tree

Accolite

Amazon

Qualcomm

Samsung

Difficulty

Submissions

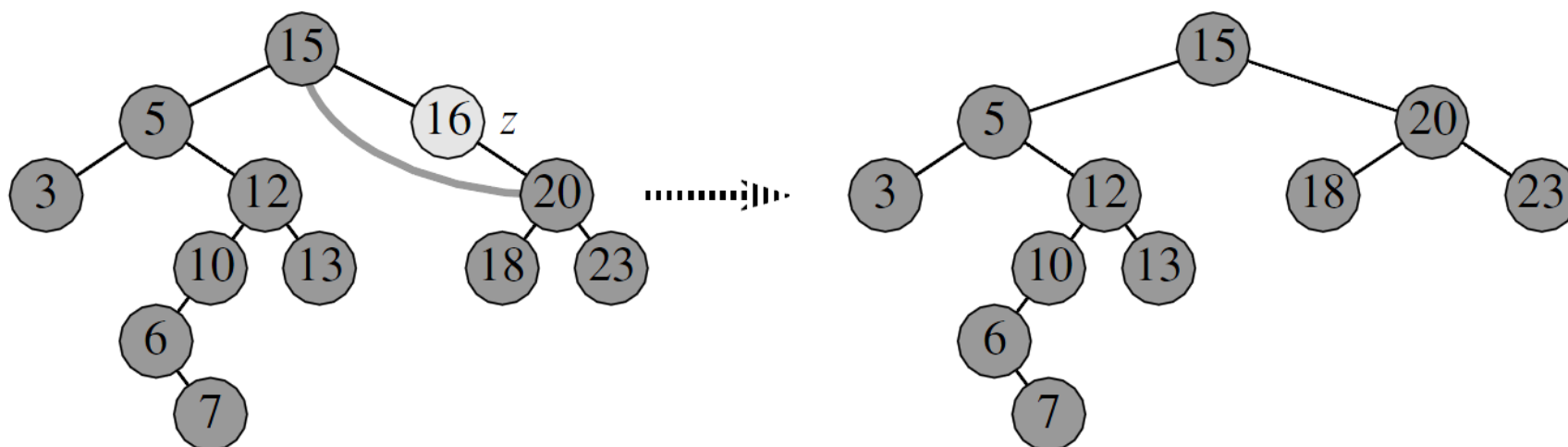
Accuracy



# Borrado de un nodo

## Caso 2

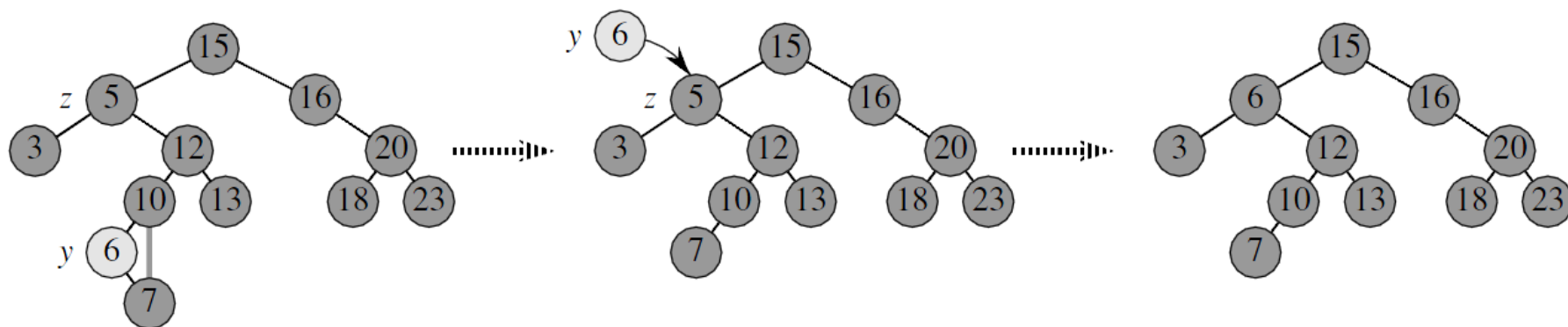
Cuando el nodo  $z$  tiene un hijo. En este caso sólo se actualizan los punteros en  $z.padre$  y en  $z.hijo_izq$  ó  $z.hijo_der$  para que se apunten mutuamente.



# Borrado de un nodo

## Caso 3

Cuando el nodo  $z$  tiene dos hijos. Se corta del BST al sucesor de  $z$  llamado  $y$  que no tenga hijo izquierdo (usando el caso 1 o 2). Se reemplaza la clave e información satelital de  $z$  con la de  $y$ .



# Borrado de un nodo - Programa

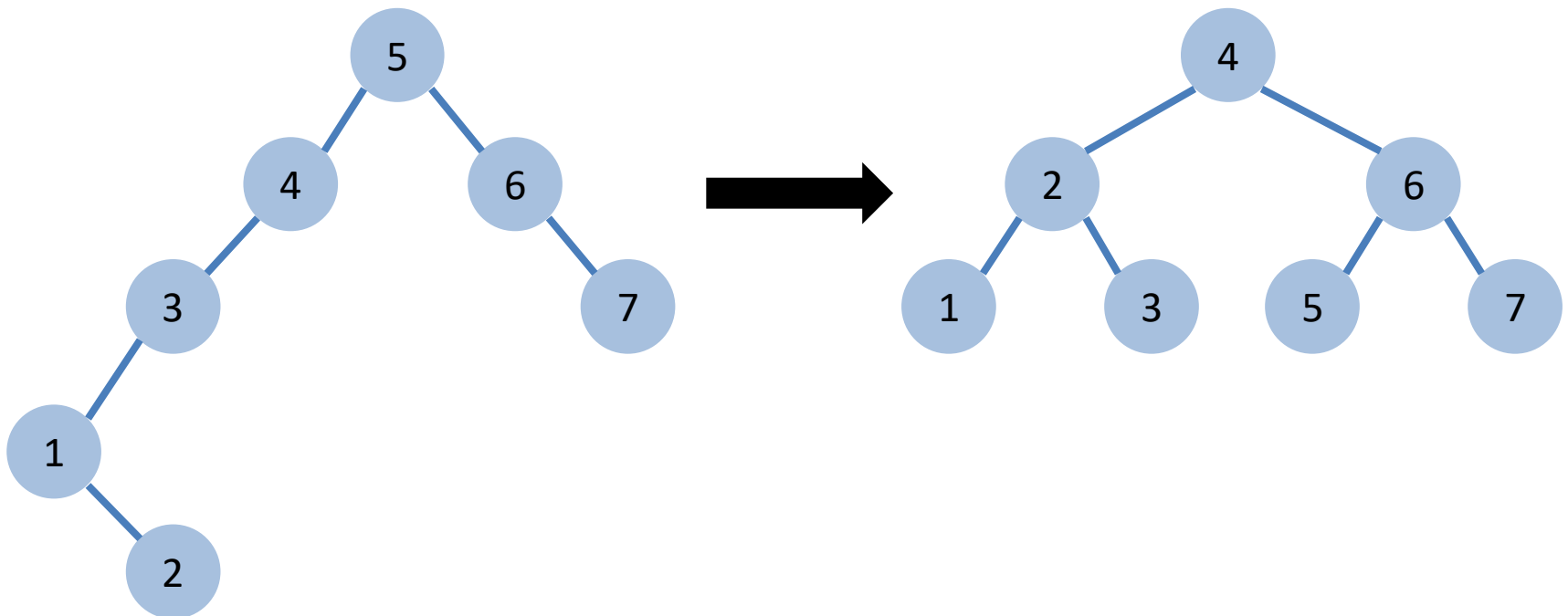
```
def borrar(self,z):
    #y será el nodo de referencia
    if z.hijo_izq==None or z.hijo_der==None: #Casos 1 y 2
        y=z
    else: #Caso 3
        y=self.obtenerSucesor(z)

    #Buscamos un hijo de y (x)
    if y.hijo_izq!=None:
        x = y.hijo_izq
    else:
        x = y.hijo_der
    if x!=None:
        x.padre = y.padre #Caso 2: se cambia el padre de x
    if y.padre==None:
        self.raiz = x #Se cambia la raíz del árbol
    else: #x toma el lugar de y en el padre
        if y is y.padre.hijo_izq:
            y.padre.hijo_izq = x
        else:
            y.padre.hijo_der = x

    #Nos falta considerar algo más para el caso3! (copiar info)
    if not (y is z): #Sólo se da en el caso 3
        z.clave = y.clave
```

# Ejercicios propuestos

- 1- Escriba una función que calcule la suma de todas las claves de las hojas en un BST.
- 2- Escriba una función que devuelva el antecesor de un nodo en un BST (definición análoga al sucesor)
- 3- Escribir una función que reciba un árbol BST, y devuelva un nuevo árbol BST balanceado. Por ejemplo:



# Ejercicios propuestos

4- Escribir una función en Python (llamada `convertirAEspejo`) que convierta un árbol binario a su “espejo”. Por ejemplo:

