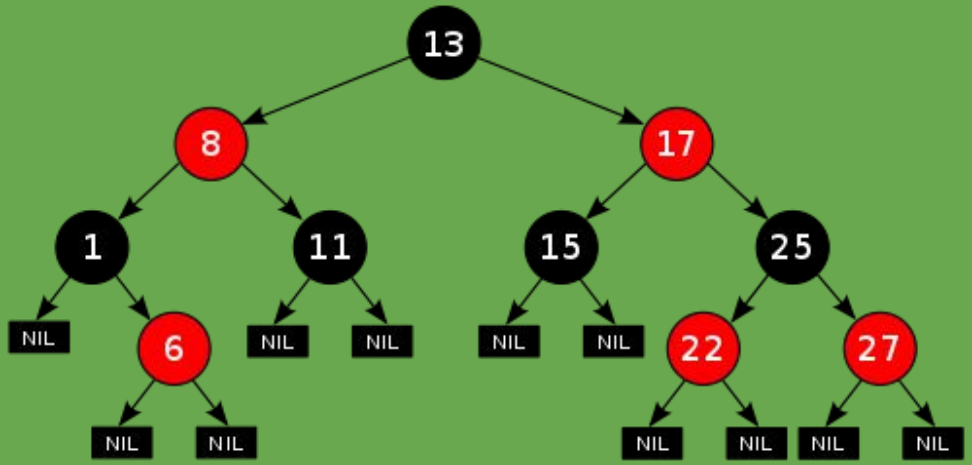


ÁRBOLES BINARIOS

Yaneth Mejía Rendón

¿QUÉ ES UN ÁRBOL?

- Estructura de datos NO LINEAL
- Es una estructura recursiva que utiliza colecciones de nodos (las pilas, colas, listas y colas dobles no son recursivas), que se ordenan dependiendo de cómo se agregan o eliminan.
- Inicia por un **nodo raíz,** donde **cada nodo** contiene **un valor**, y opcionalmente una lista de referencias a otros nodos (sus hijos).

Limitación: ninguna referencia puede estar duplicada, o apuntar al nodo raíz.

- Los árboles se usan para representar orden, jerarquía.
- No puede tener ciclos

¿QUÉ ES UN ÁRBOL?

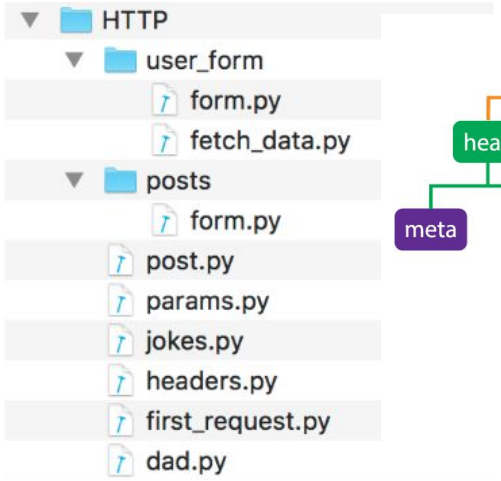
Como desarrolladores web, la forma más fácil de visualizar esta estructura de datos es imaginándonos el árbol del DOM (Document Object Model).

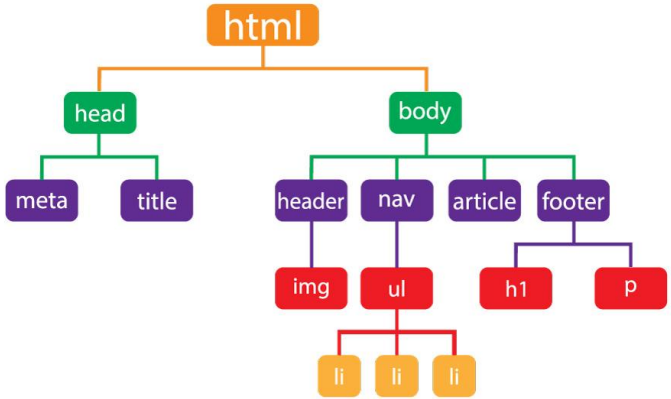
En esta estructura, partimos de un nodo raíz (<html>), y cada nodo hijo (elemento del DOM) tiene un padre, y puede o no contener uno o más hijos.

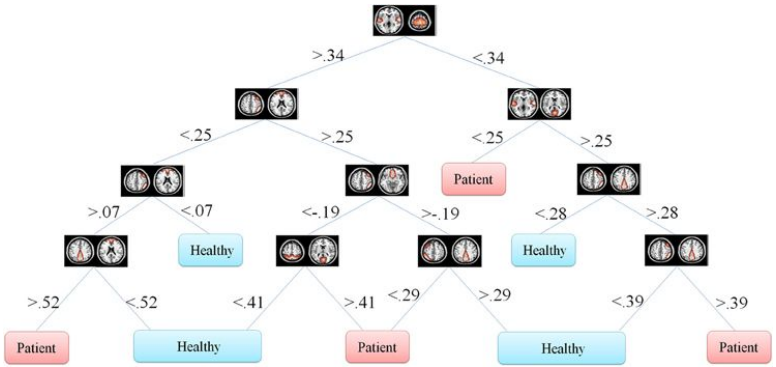
Un nodo sólo puede tener un padre, ya que no puede ser hijo directo de más de un nodo.

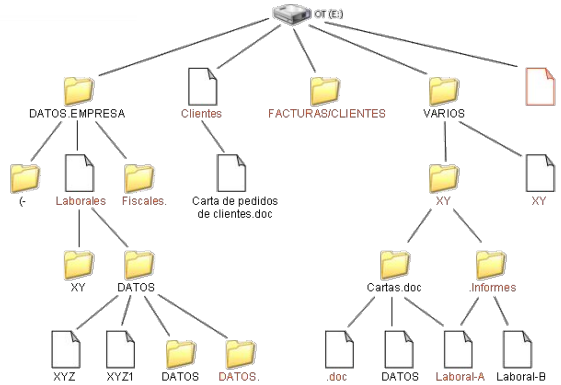
EJEMPLOS DE ESTRUCTURAS DE ÁRBOL

HTML DOM
Enrutamiento de red
Árbol de sintaxis abstracta
Árboles de decisión en inteligencia artificial
Directorio de carpetas de nuestro equipo







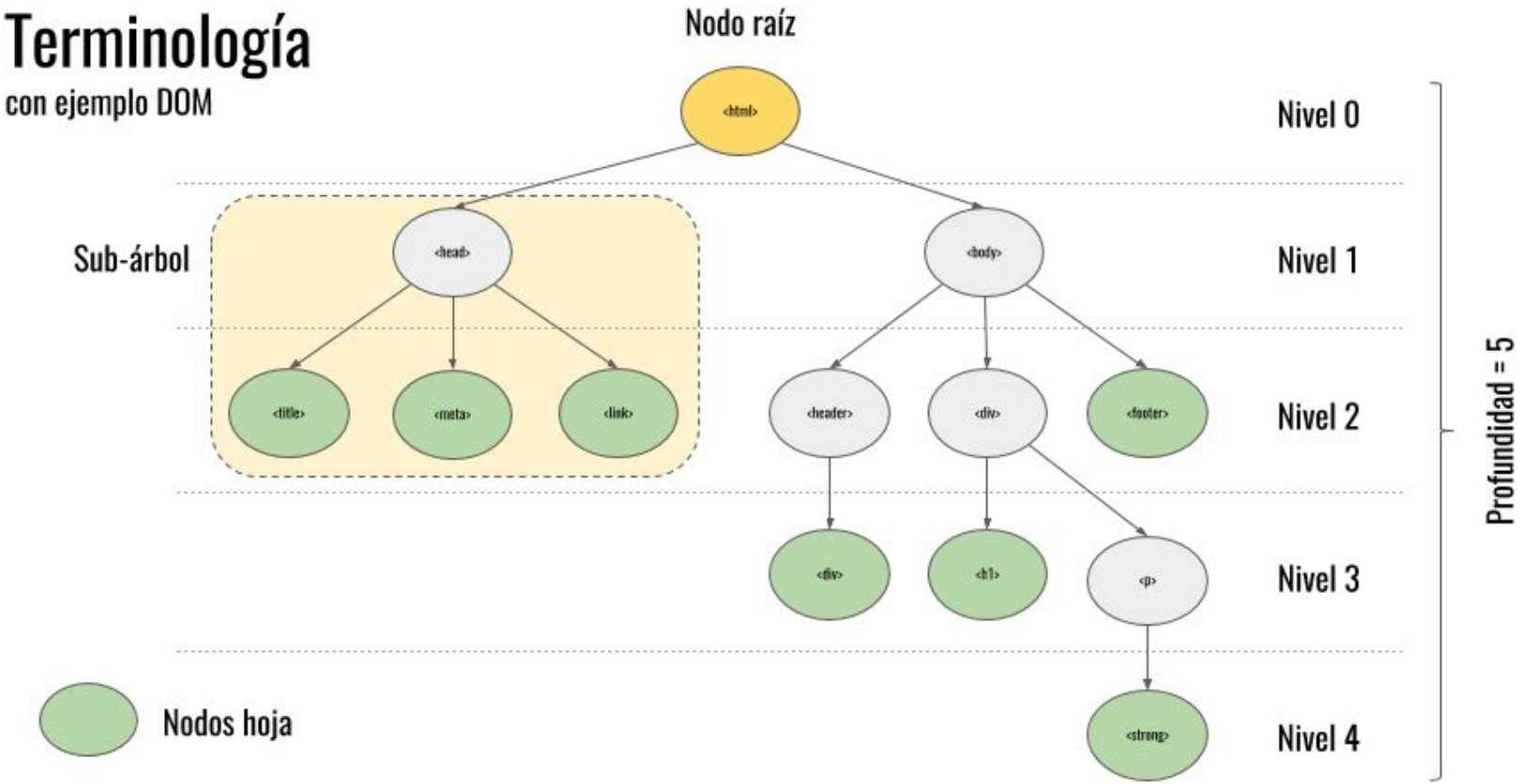


TERMINOLOGÍA DEL ÁRBOL

- **Raíz:** el nodo superior de un árbol.
- **Hijo: un** nodo conectado directamente a otro nodo cuando se aleja de la raíz.
- **Padre** - La noción inversa de un hijo.
- **Hermanos** -Un grupo de nodos con el mismo padre.
- **Hoja** - Un nodo sin hijos.
- **Borde** : la conexión entre un nodo y otro.

Terminología

con ejemplo DOM



RECORRER UN ÁRBOL

Navegar en el Árbol

El árbol tiene una propiedad root o también conocida RAÍZ donde se coloca el primer valor.

Los nodos tienen propiedades left y right para ir para la izquierda o derecha del árbol por medio de punteros.

```
function ArbolBinario(){
    this.root = null
    this.current = this.root
}

function Node(value){
    this.value=value
    this.left=null
    this.right=null
    this.height=0
}
```

Auto Balanceo

Cada vez que elimines o agregues un elemento el árbol va a validar que se encuentre balanceado, de lo contrario, se va a balancear por sí mismo.

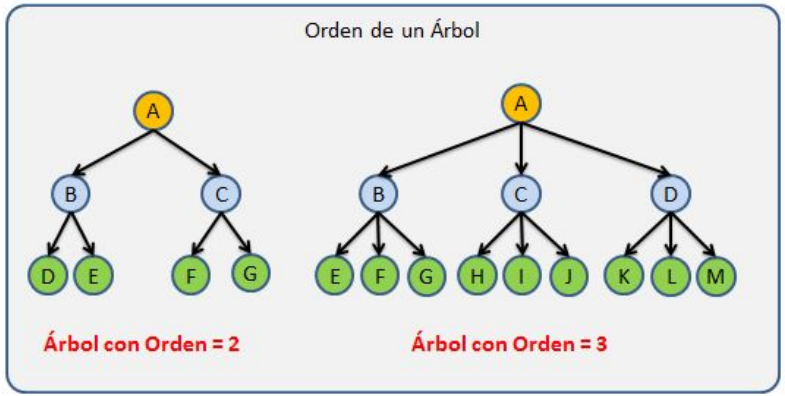
❒ La altura de un nodo es el largo de la trayectoria más larga de ese nodo a una hoja.

❒ La profundidad de un nodo es el largo de la trayectoria desde la raíz a ese nodo.

ORDEN O GRADO DE LOS ÁRBOLES

Determina cuántos hijos puede tener un nodo. Por ejemplo:

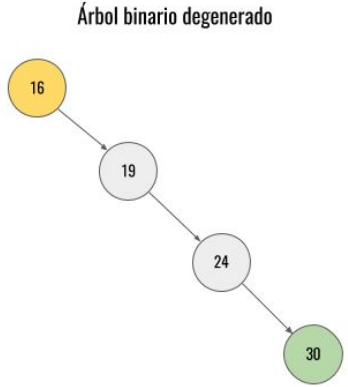
- Un árbol de orden 2 sería un **árbol binario**, donde cada nodo puede tener como máximo dos hijos.
- Un árbol de orden 3 o **ternario** permitiría que cada nodo tenga un máximo de tres hijos.
- Un árbol no está obligado a determinar un orden o grado.



TIPOS DE ÁRBOLES BINARIOS

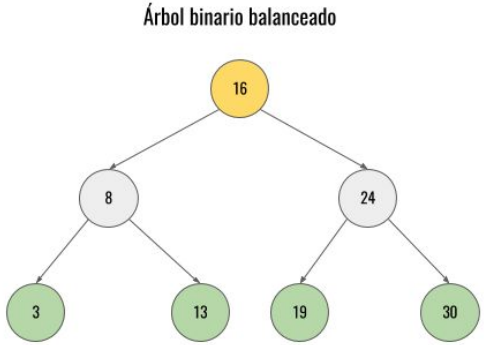
Árbol degenerado

Cuando un árbol contiene 1 sólo hijo por nodo. Este tipo de árboles se comportan como listas.



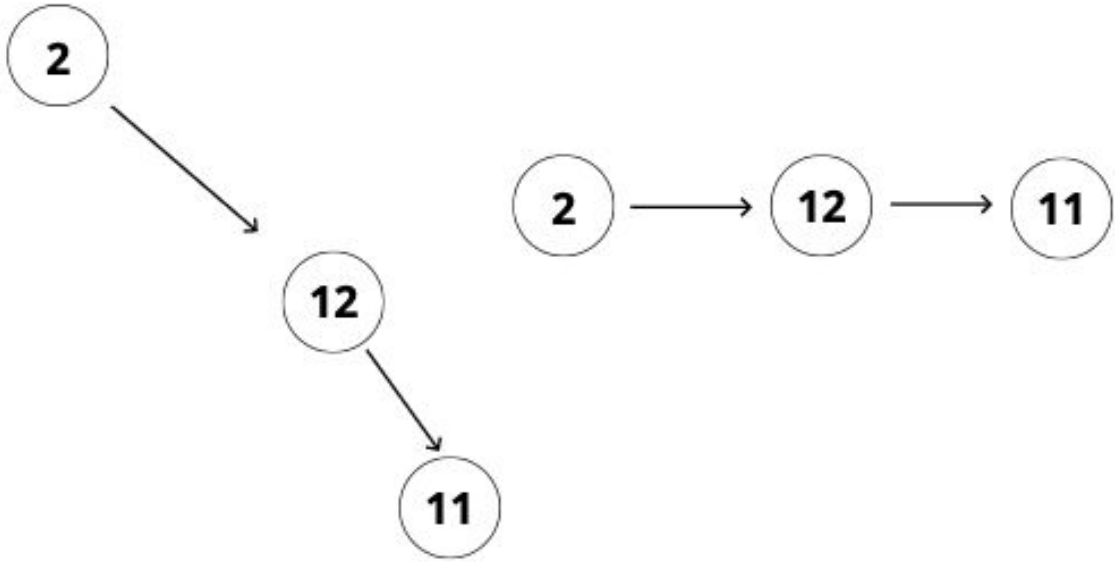
Árbol balanceado

Lo opuesto a un árbol degenerado, donde el árbol tiene la profundidad mínima posible dado un número de elementos.

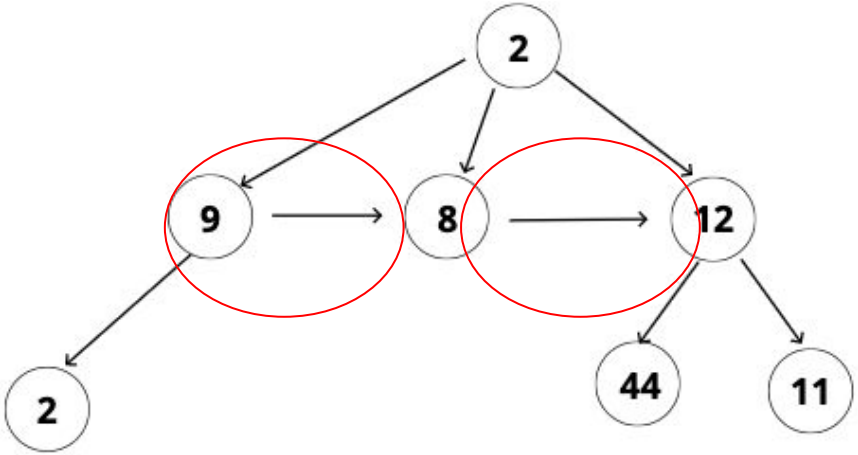


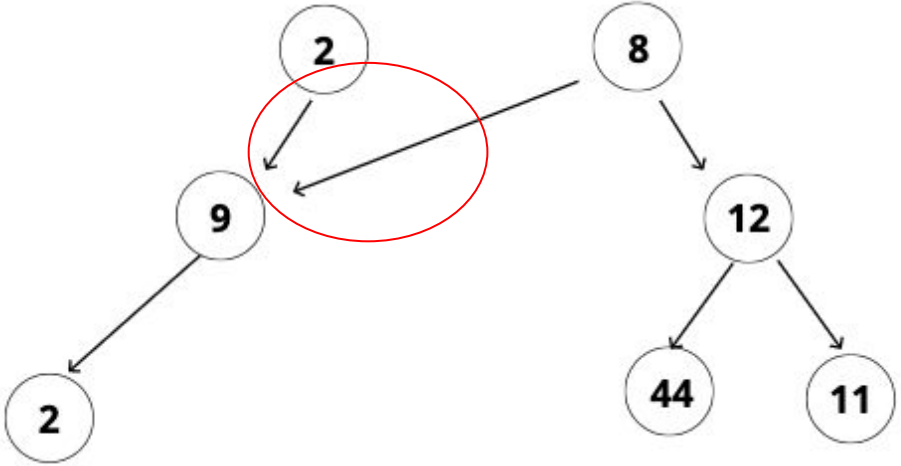
Una lista enlazada individualmente

(una especie de caso especial de un árbol)

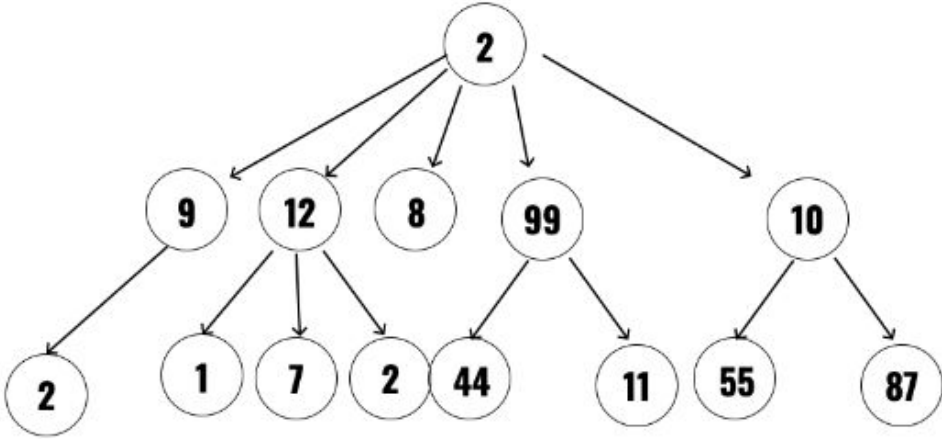


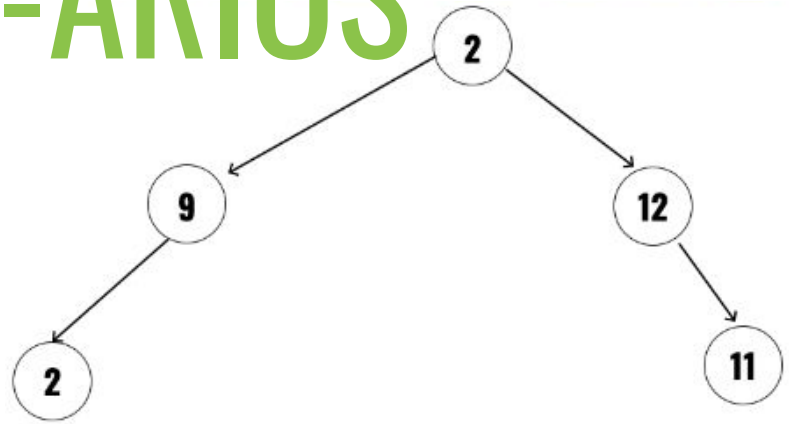
NO SON ÁRBOLES

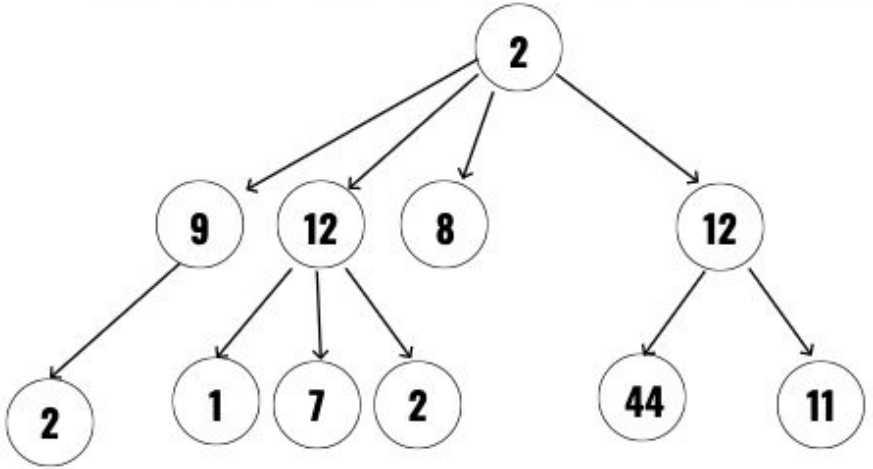




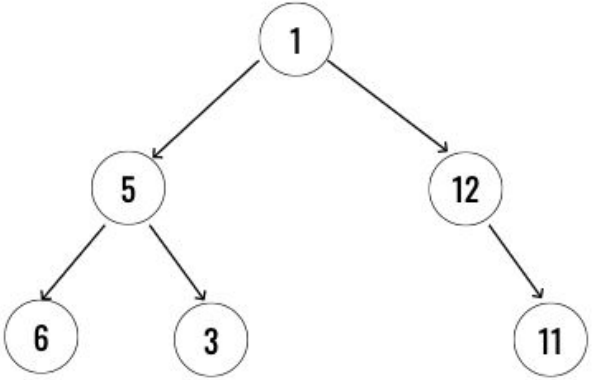
ÁRBOLES N-ARIOS



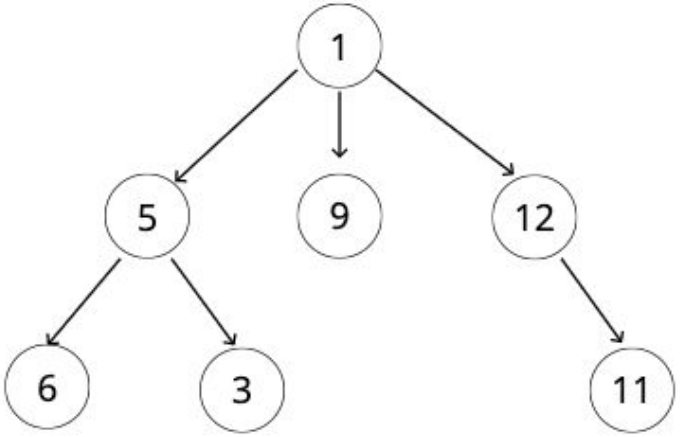




ÁRBOLES BINARIOS



NO ES UN ÁRBOL BINARIO



ÁRBOLES BINARIOS

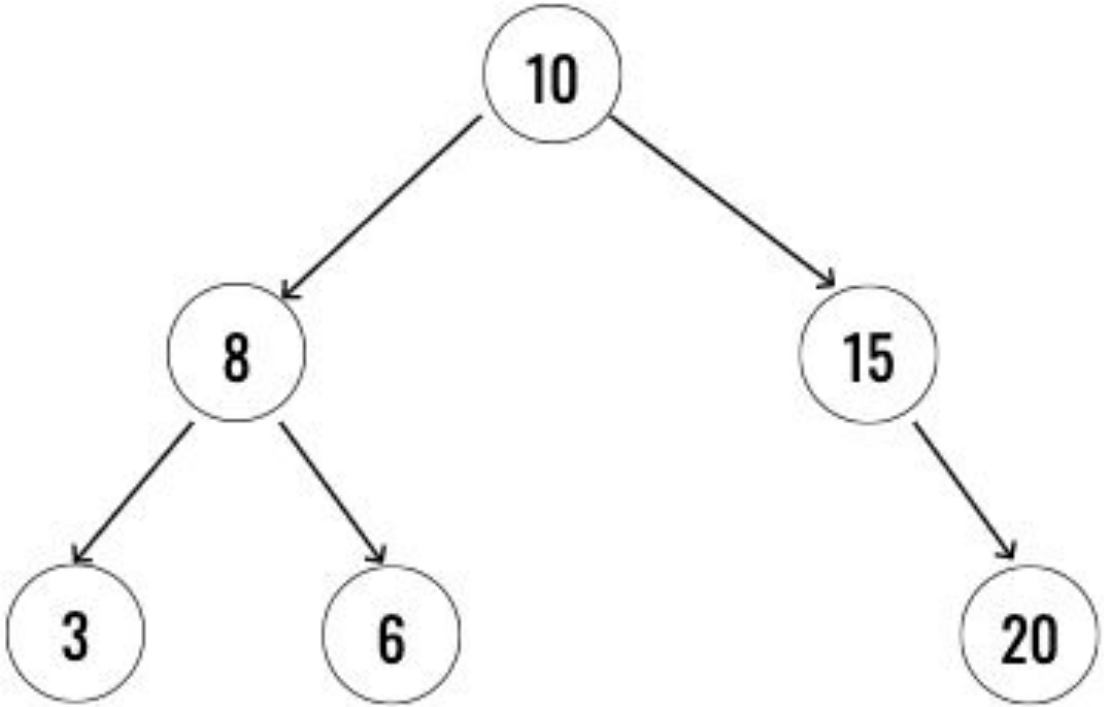
Muchas aplicaciones diferentes también!

- Árboles de decisión (verdadero / falso)
- Indicaciones de la base de datos
- Clasificación de los algoritmos

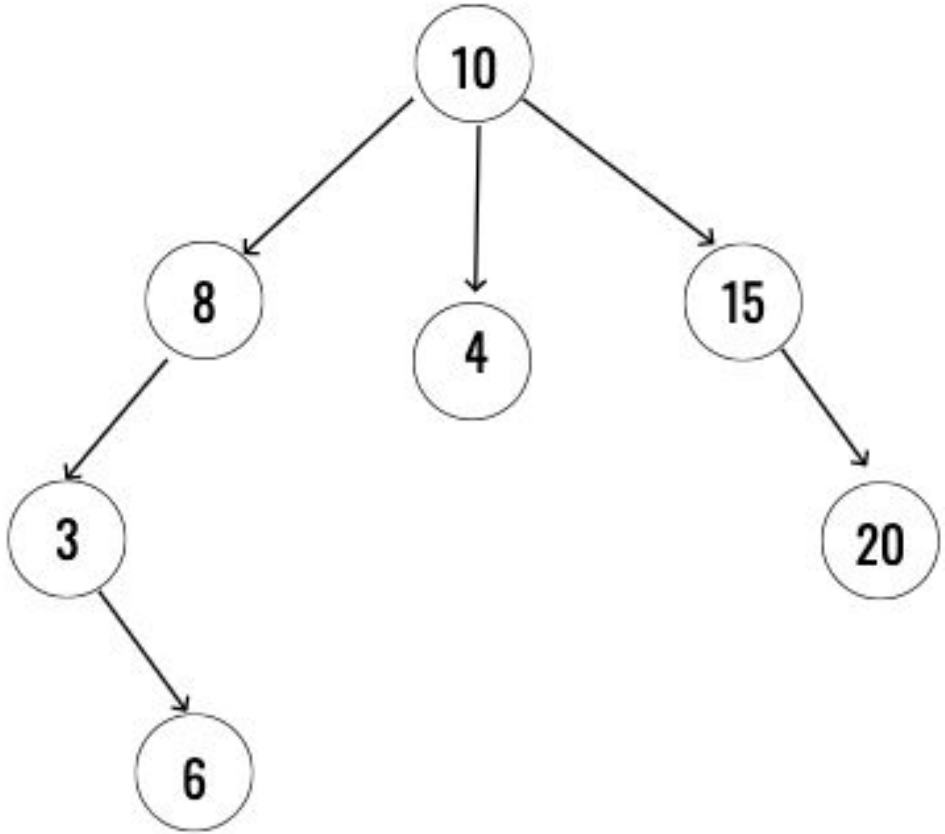
CÓMO FUNCIONA BSTS

- Cada nodo padre tiene como máximo **dos** hijos
- Cada nodo a la izquierda de un nodo principal **siempre** es **menor** que el principal
- Cada nodo a la derecha de un nodo principal **siempre** es **mayor** que el principal

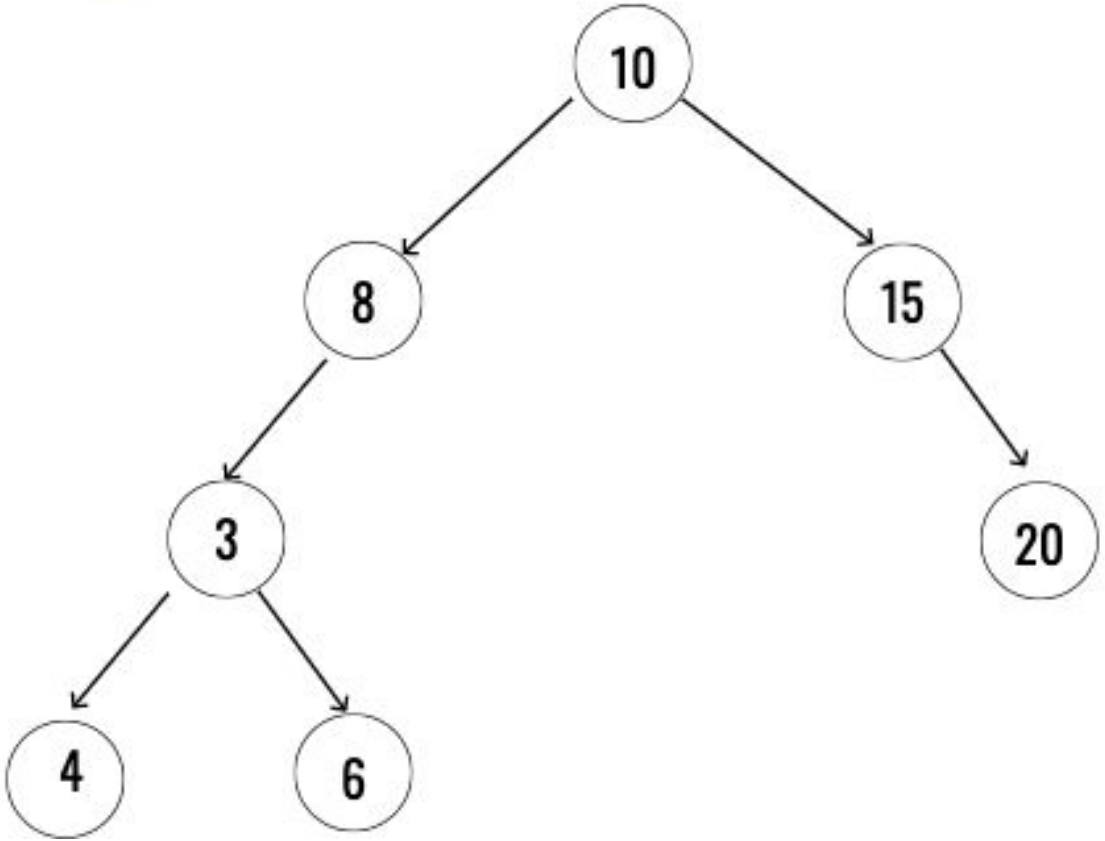
¿Es este un BST válido?



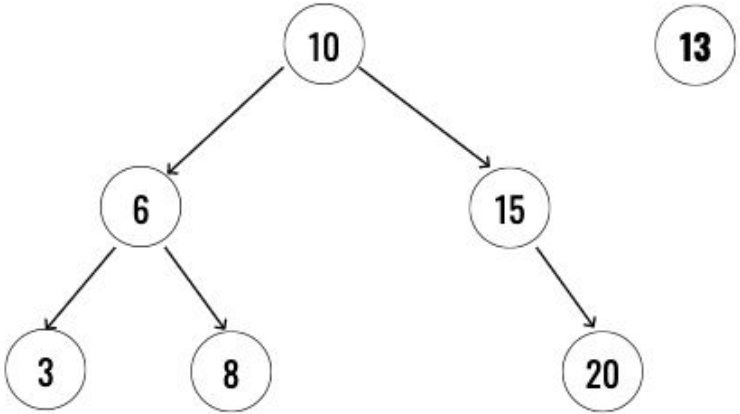
¿Es este un BST válido?



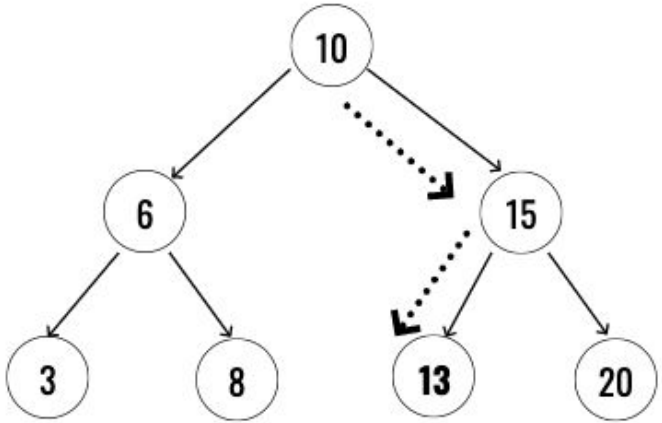
¿Es este un BST válido?



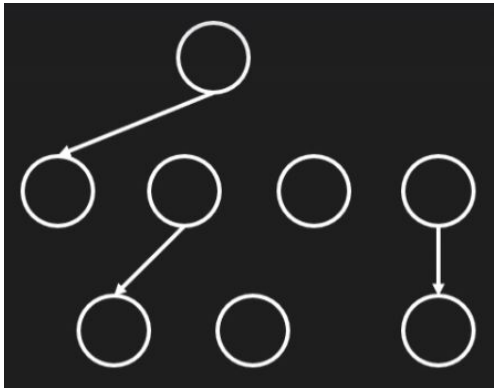
INSERTANDO

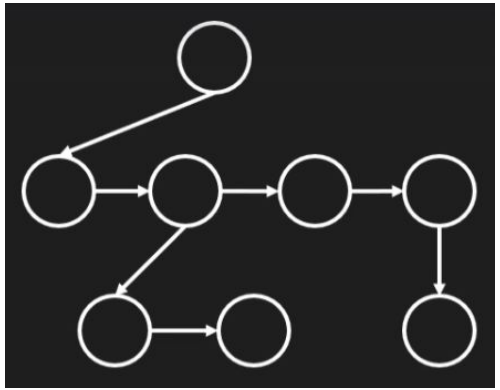


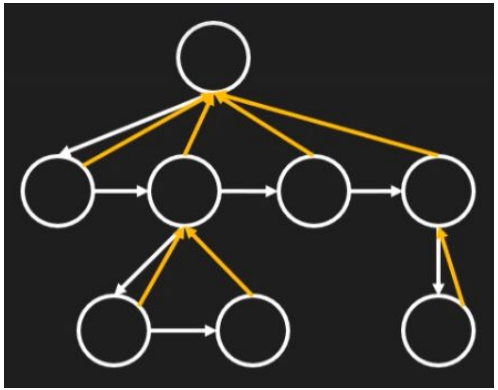
INSERTANDO



1. Lista circular que apuntara a la cabeza
2. Cada elemento de la lista tendra una referencia de quien es el padre







1. **Propiedad nodo**: valor y nodo siguiente

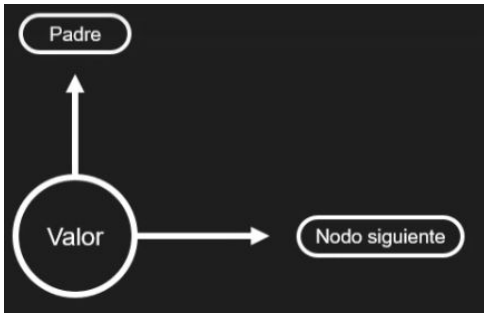
2. **Propiedad padre**

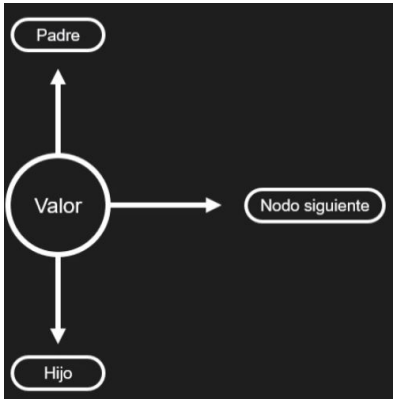
3. **Propiedad hijo**, el cual no se debe declarar en el constructor, debido a que no existe hasta que se añade el nuevo elemento, con la función append()

4. Haga de cuenta que el nodo siguiente apunta a un nodo hermano de ese nivel, para facilitarnos el recorrido

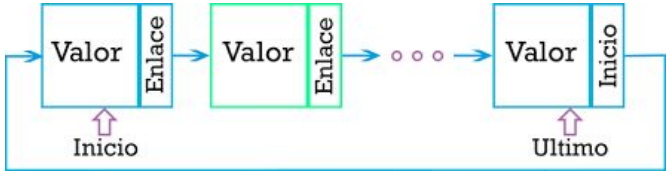
GeneralTree > binaryTree.py > Tree > sll > __init__

```
1   class Tree:
2       class sll:
3           class Node:
4               #Creamos el método inicializador de la clase nodo
5               def __init__(self, value, father):
6                   self.value = value
7                   self.father = father
8                   self.son = None
9                   self.next = None
10          #Ingreso a la clase sll y creamos el método inicializador
11          def __init__(self):
12              self.head = None
13              self.tail = None
14              self.length = 0
```





AÑADIR NODO RAÍZ EN LISTA VACÍA CON MÉTODO INSERT:

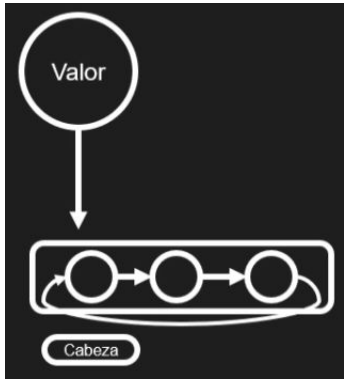


A partir de la cabeza se tendrá acceso a los demás elementos de la lista circular.

Cada uno de los nodos debe tener referencia explícita de quién es el padre.

Cuando se crea la lista se debe especificar la cabeza, cola y el tamaño de la lista.

Cuando se añade un nuevo nodo se debe incluir la referencia de quién es el padre.



- Si solo existe un elemento se enlazara la cabeza con la cola para hacer la lista circular.
- Si la raíz es igual a None quiere decir que no hay **nada**.
- Por ende, si no hay nada, la raíz será igual a una nueva lista circular y en el append sólo se le envía el valor sin el padre, puesto que es el nodo raíz.

```
def insert(self, value, father):
    if self.root == None:
        self.root = self.sll()
        self.root.append(value, None)
```

AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

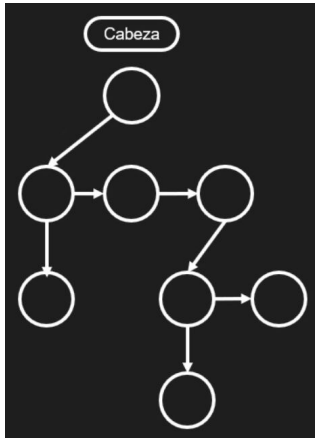
Si ya existe el nodo raíz:

- El nodo actual lo ubicamos en la cabeza de la lista circular por niveles

- Declaro una función interna secundaria en **insert** para recorrer el árbol.

- El parámetro **comparision_node_value** de la función recorrer, es para cuando lleguemos al nivel más profundo del árbol, debemos de volver a subir, así evitar un bucle infinito, donde volvemos a visitar nodos previamente visitados.

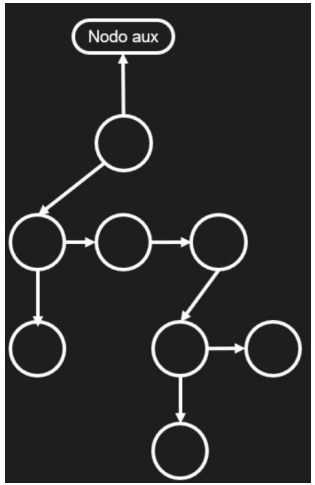
```
def insert(self, value, father):
    if self.root == None:
        self.root = self.sll()
        self.root.append(value, None)
    else:
        current_node = self.root.head
        def tree_route(node, comparision_node_value = None):
            auxiliar_node = node.father
```

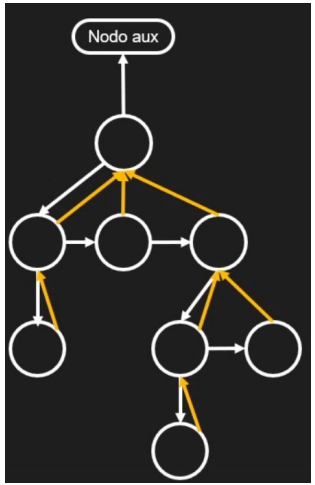


AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

- La variable **auxiliar_node** va almacenar el padre del nodo visitado, teniendo en cuenta que si es la raíz, almacenaría None, porque no tiene padre.

```
def insert(self, value, father):
    if self.root == None:
        self.root = self.sll()
        self.root.append(value, None)
    else:
        current_node = self.root.head
        def tree_route(node, comparision_node_value = None):
            auxiliar_node = node.father
```



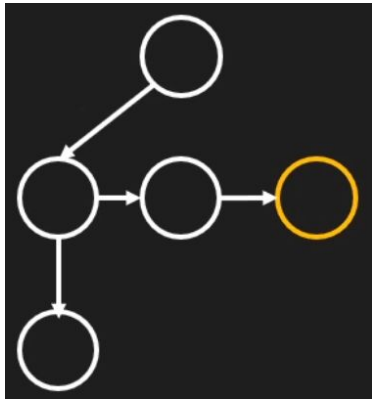


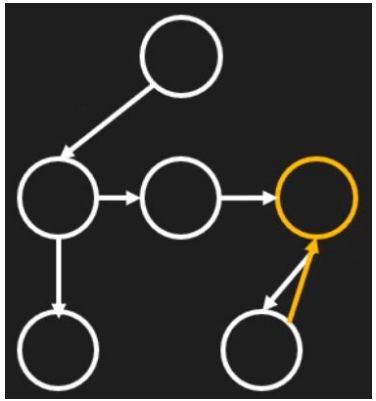
AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

- El primer if valida si el value que se quiere añadir es el mismo valor que tiene uno de los nodos padres. No se puede añadir, porque no pueden existir duplicados en el árbol.

- **ESCENARIO I: Si el valor del nodo es un padre, se debe validar lo siguiente:**

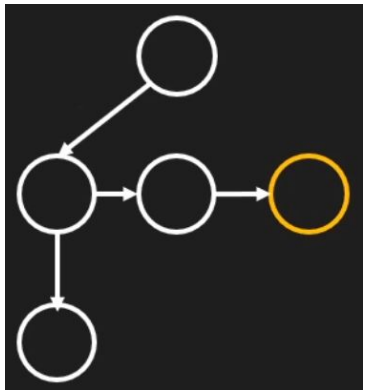
    - **Si el nodo hijo es None**, se declara que esa propiedad hijo es igual a una nueva lista circular y se añade, pasandole a append el value y como padre el nodo en el que estamos actualmente (el amarillo) que recibe insert
    - Con la referencia circular podemos dejar clara la relación entre padre e hijo.

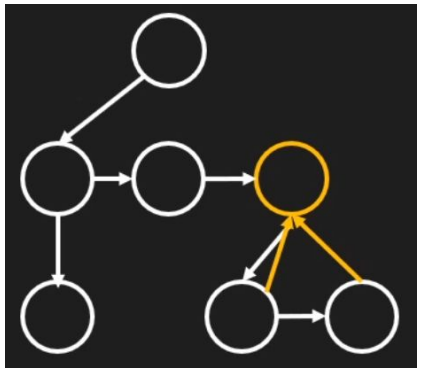




AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

- **Si el nodo hijo NO es None**, Se añade el nuevo hijo con append a ese padre.

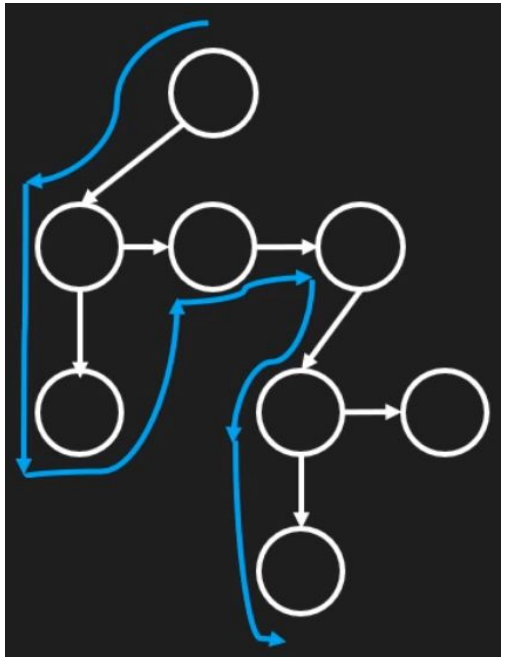




AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

ESCENARIO III: Si el valor del nodo NO es un padre, es decir es un nodo hoja, se debe validar lo siguiente:

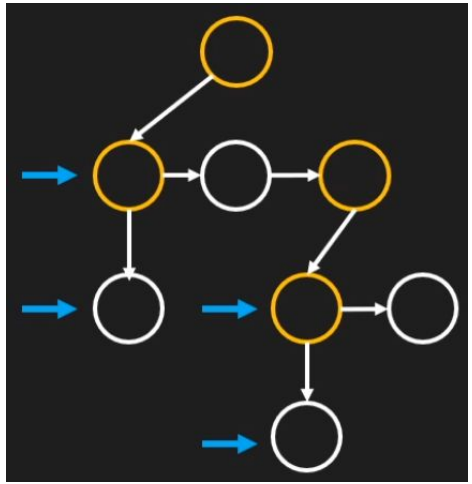
Se debe hacer un recorrido en el sentido que muestra la imagen desde el subárbol más a la izquierda a la derecha. De arriba hacia abajo.



Si el padre existe:

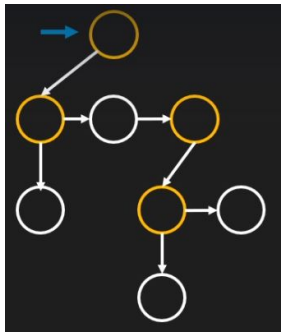
- Si tiene hijos, como los nodos señalados en amarillo.
- Si el nodo hijo en su propiedad cabeza valor es igual al nodo **comparision_node_value**

Si el hijo del padre en el que estamos, su cabeza es el nodo **comparision_node_value**
Ya visitamos ese padre



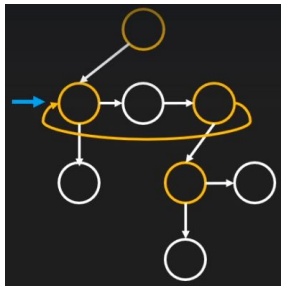
AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

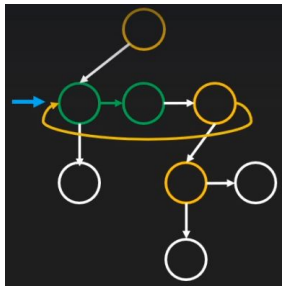
ESCENARIO III: Si el valor del nodo NO es un padre, es decir es un nodo hoja, se debe validar lo siguiente:

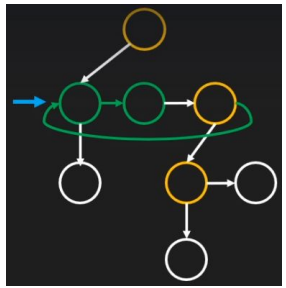


- Si el nodo valor es igual a la raíz, ya terminamos de recorrer el árbol y no encontramos el padre para insertar el nuevo nodo.

- De lo contrario: Si el nodo en el que nos encontramos, el valor del nodo siguiente no es igual al nodo auxiliar (el padre del nodo en el que nos encontramos) que es la referencia a la cabeza de la lista.
    - En la imagen dos no se cumple que el nodo siguiente sea la cabeza, es decir, no estamos visitando el último nodo de esa lista.



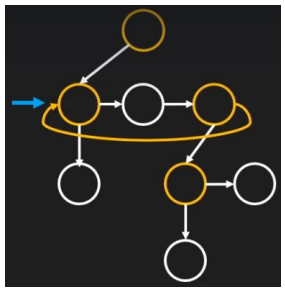


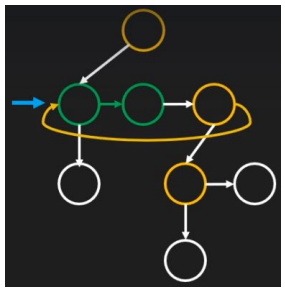


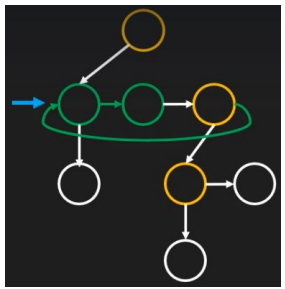
AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

ESCENARIO III: Si el valor del nodo NO es un padre, es decir es un nodo hoja, se debe validar lo siguiente:

```
    else:
        if node.son != None:
            #Ya visitamos todo el árbol en produnfidad, nos devolvemos
            if node.son.head.value == comparision_node_value:
                if node.value == self.root.head.value:
                    return False
                elif node.next.value != auxiliar_node.son.head.value:
                    return tree_route(node.nex, node.value)
    print(auxiliar_node)
```





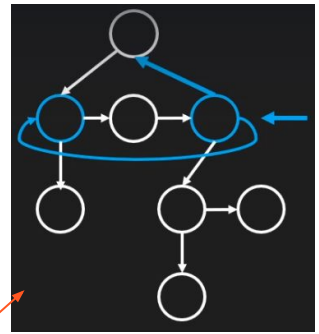


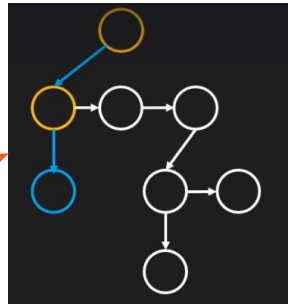
AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

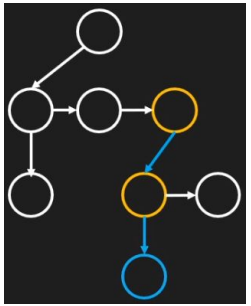
ESCENARIO III: Si el valor del nodo NO es un padre, es decir es un nodo hoja, se debe validar lo siguiente:

IMPORTANTE: Siempre bajar a lo más profundo del árbol llamando la función de forma recursiva.

```
if node.son != None:
    #Ya visitamos todo el árbol en produnfidad, nos devolvemos
    if node.son.head.value == comparision_node_value:
        if node.value == self.root.head.value:
            return False
        elif node.next.value != auxiliar_node.son.head.value:
            return tree_route(node.next, node.value)
        else:
            return tree_route(node.father, node.next.value)
    else:
        return tree_route(node.son.head, node.son.head.value)
```







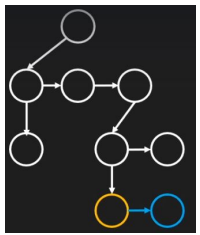
AÑADIR NODO DESPUÉS DE LA RAÍZ EN LISTA CON EL MÉTODO INSERT:

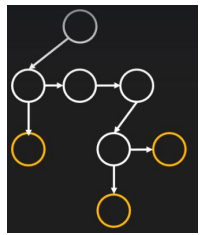
ESCENARIO III: Si el valor del nodo NO es un padre, es decir es un nodo hoja, se debe validar lo siguiente:
IMPORTANTE: Siempre bajar a lo más profundo del árbol llamando la función de forma recursiva.

```
if node.son != None:
    #Ya visitamos todo el árbol en produnfidad, nos devolvemos
    if node.son.head.value == comparision_node_value:
        if node.value == self.root.head.value:
            return False
        elif node.next.value != auxiliar_node.son.head.value:
            return tree_route(node.next, node.value)
        else:
            return tree_route(node.father, node.next.value)
    else:
        return tree_route(node.son.head, node.son.head.value)
elif node.next.value != auxiliar_node.son.head.value:
    return tree_route(node.next, node.value)
```

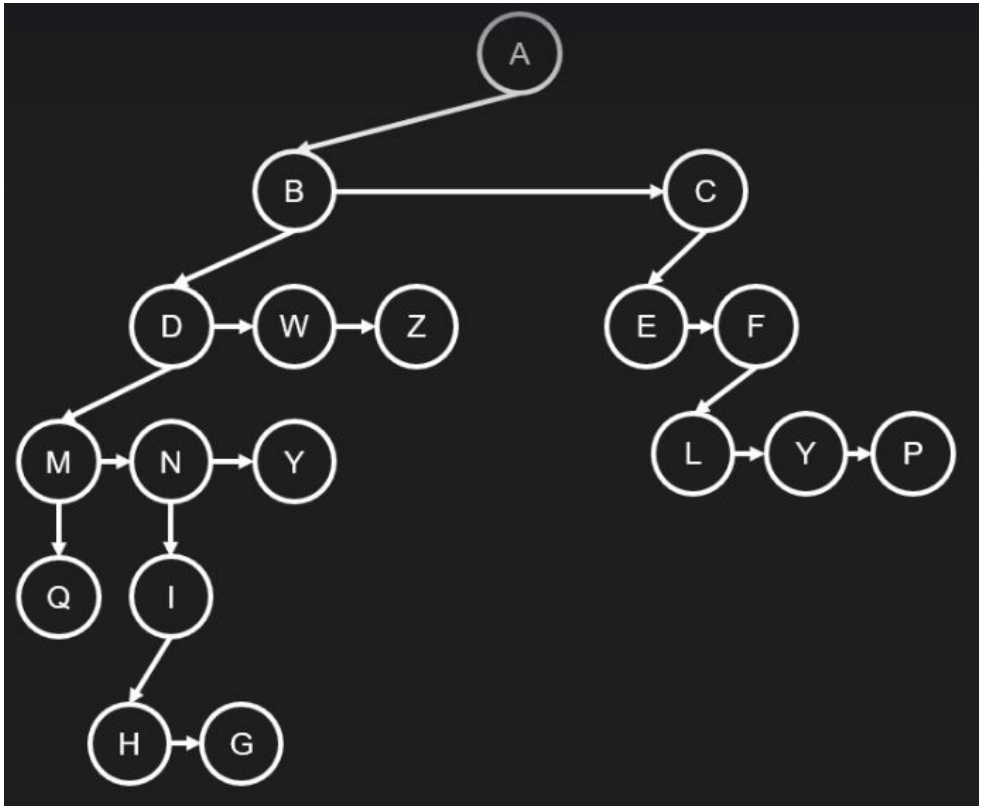
Devolviendonos en el recorrido del árbol

Primera vez que recorremos el árbol. Nodos donde se cumple el caso:



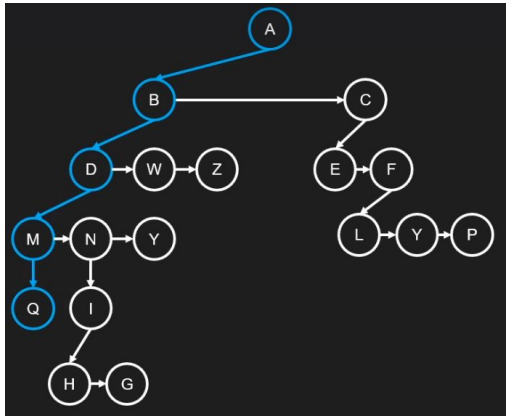


EJEMPLO
AÑADIR UN NUEVO NODO
INSERT(O, Y)

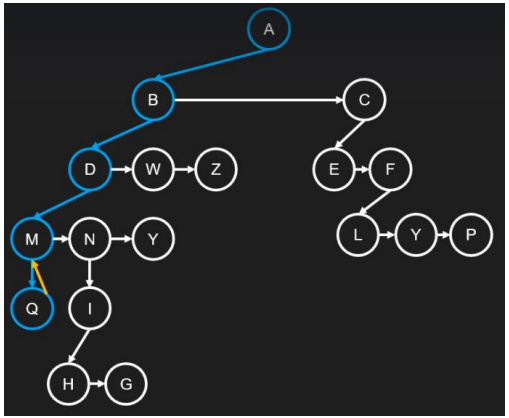


Se tiene este árbol y se quiere insertar el nodo O que tendrá como padre a Y

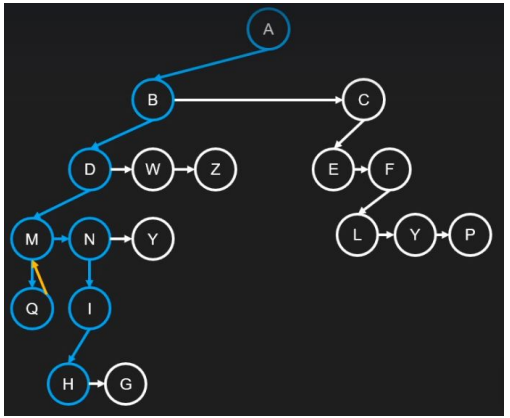
EJEMPLO
AÑADIR UN NUEVO NODO
INSERT(O, Y)



Q llama a la función recorrer, porque no tiene más caminos, ni hijo ni nodo siguiente

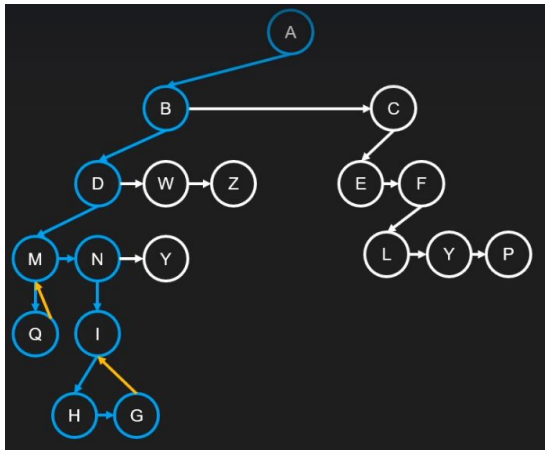


M no es la raíz y si tiene nodo siguiente pasa a N

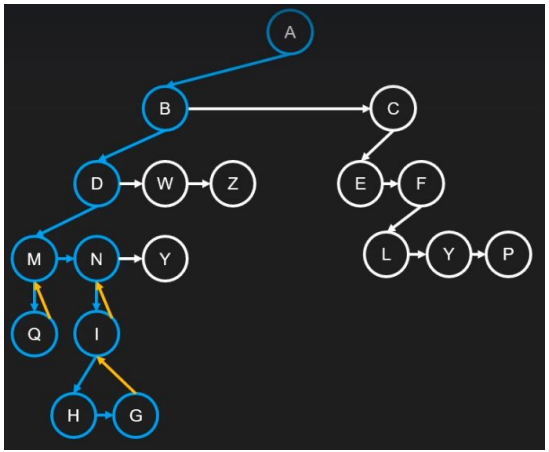


Se evalúa de N hasta H, pasando al nodo siguiente G

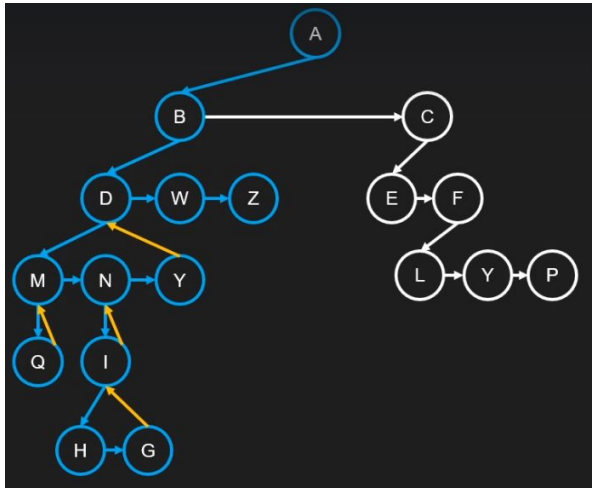
EJEMPLO
AÑADIR UN NUEVO NODO
INSERT(O, Y)



Estando en G evalúa que ya visito H, asi que se devuelve mediante tree_route
Recuerde H es la cabeza del hijo de I que es G también. comparision_node_value es H que sería la cabeza del hijo de I

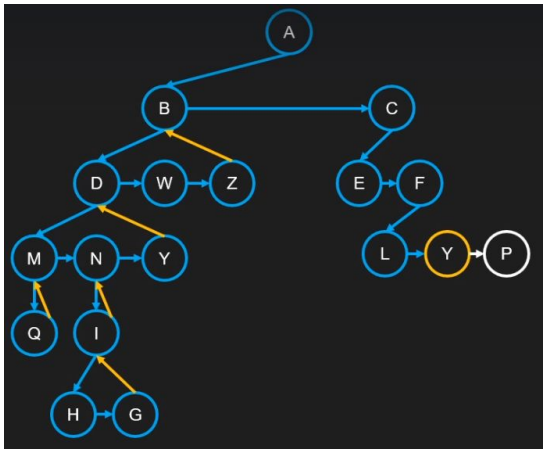


Evalúa para N que su hijo I es el nodo cabeza y que el nodo comparision_node_value prueba es I

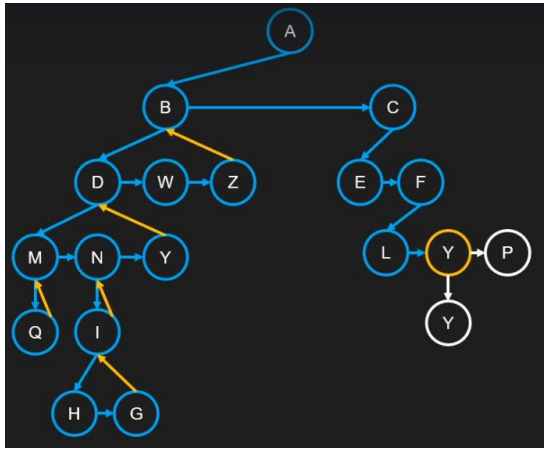


Así continuó toda el recorrido de la lista

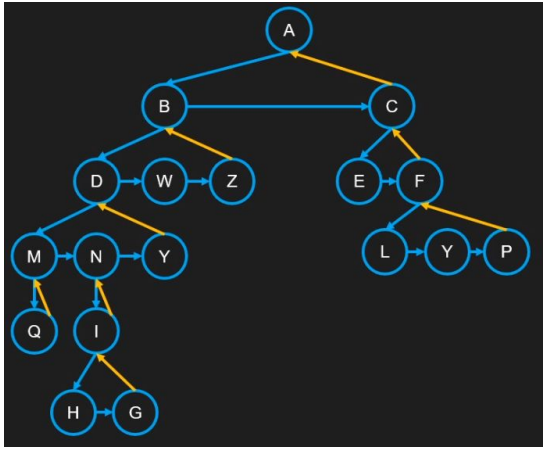
EJEMPLO
AÑADIR UN NUEVO NODO
INSERT(O, Y)



Cuando encuentra el padre, llama el método append()

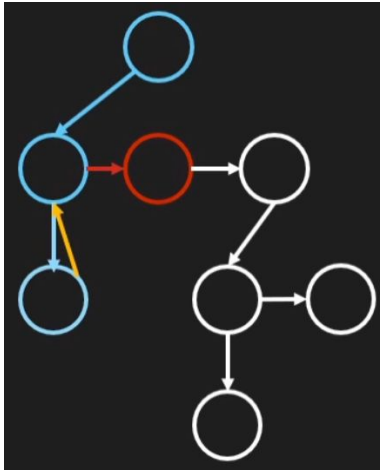


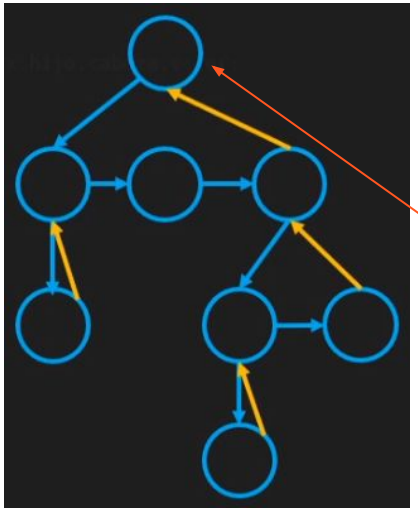
No se podría agregar después de haber hecho todo el recorrido




```
yaneth@yaneth-ROG-Zephyrus-G14-GA401IV-GA401IV:~/Documents/Universidades/UAM/TAD/BinaryTree/GeneralTree$ python3 main.py
b -> a
c -> a
El nodo ya existe
```

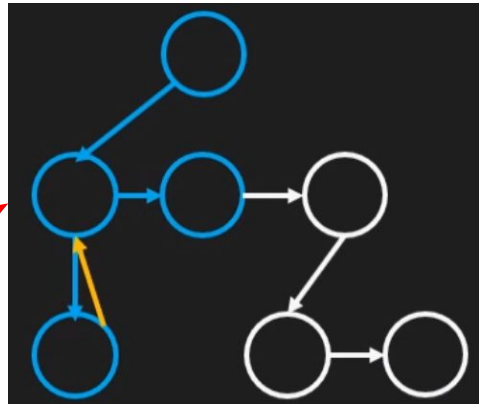
```python
def find(self, value):
    #Iniciamos en la raiz y su propiedad cabeza
    current_node = self.root.head
    def tree_route(node, comparision_node_value=None):
        #Para verificar si lo que se tiene por siguiente es la cabeza de la lista
        auxiliar_node = node.father
        if node.value == value:
            return node.value
```

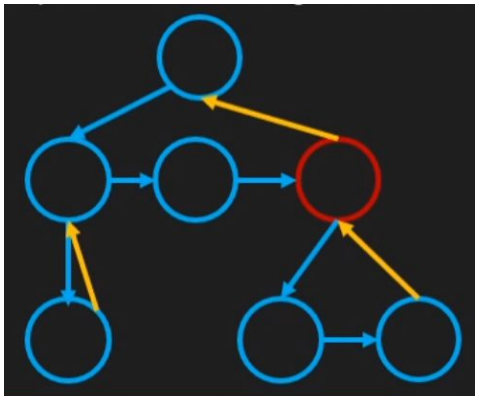




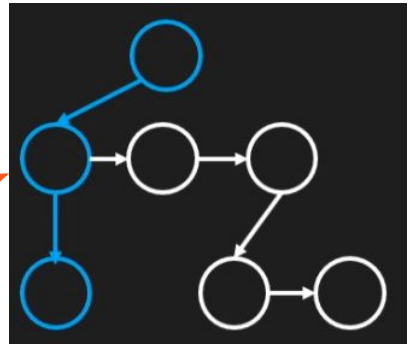
```python
#Debemos recorrer el árbol si no es un padre
#Si el nodo hijo no esta vacío
elif node.son != None:
    #Si el nodo hijo en su propiedad cabeza valor es igual igual a comparision_node_value
    #Es porque ya se visito y debemos devolvernos un nivel
    if node.son.head.value == comparision_node_value:
        #Validar si recorrimos todo el árbol y ya llegamos a la raíz
        if node.value == self.root.head.value:
            return "No existe el node buscado"
        #Si el nodo en el que nos encontramos lo que tiene como nodo sgte no es la cabeza
        #de la lista en la que nos encontramos
        elif node.next.value != auxiliar_node.son.head.value:
            return tree_route(node.next, node.value)
        else:
            return tree_route(node.father, node.next.value)
```

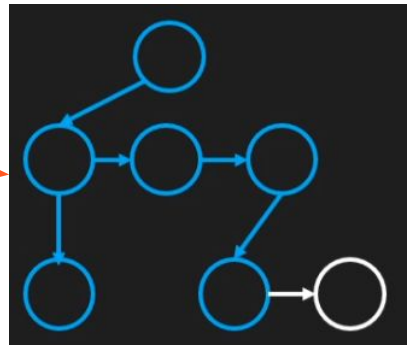
```
elif node.son != None:
    #Si el nodo hijo en su propiedad cabeza valor es igual igual a comparision_node_value
    #Es porque ya se visito y debemos devolvernos un nivel
    if node.son.head.value == comparision_node_value:
        #Validar si recorrimos todo el árbol y ya llegamos a la raíz
        if node.value == self.root.head.value:
            return "No existe el node buscado"
        #Si el nodo en el que nos encontramos lo que tiene como nodo sgte no es la cabeza
        #de la lista pasamos al siguiente nodo de esa lista
        elif node.next.value != auxiliar_node.son.head.value:
            return tree_route(node.next, node.value)
        #De lo contrario, el nodo en el que nos encontramos si es la cabeza de una lista
        else:
            return tree_route(node.father, node.next.value)
```

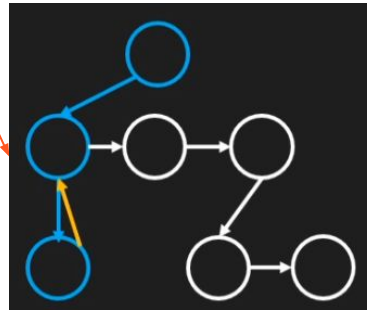


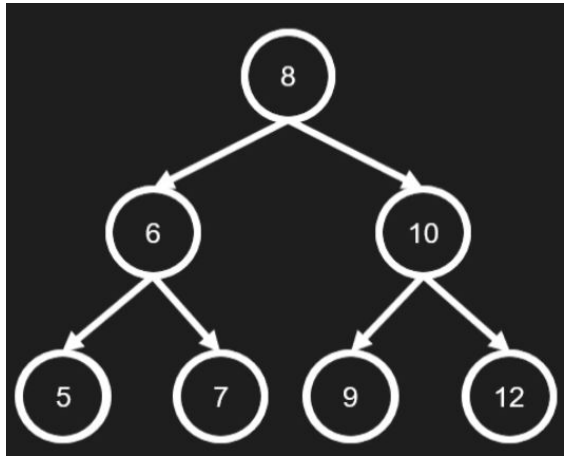


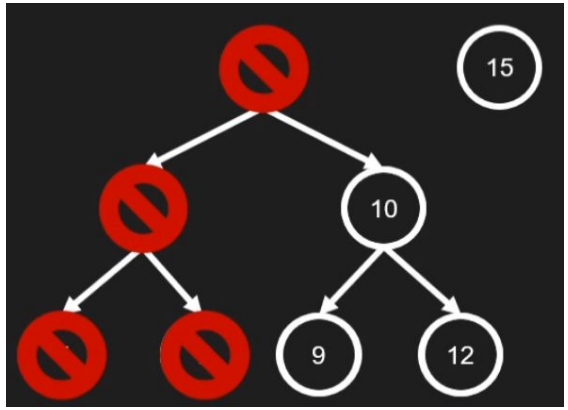
```
elif node.son != None:
    if node.son.head.value == node_puebra:
        if node.value == self.raiz.head.value:
            return " >>>> No existe el nodo buscado <<<<"
        elif node.next.value != node_aux.son.head.value:
            return tree_route(node.next, node.value)
        else:
            return tree_route(node.father, node.next.value)
    else:
        return tree_route(node.son.head, node.son.head.value)
elif node.next.value != node_aux.son.head.value:
    return tree_route(node.next, node.value)
else:
    return tree_route(node.father, node.next.value)
```








8
6
10
5
7
9
12


15
10
9
12

IMPLEMENTACIÓN DEL CÓDIGO

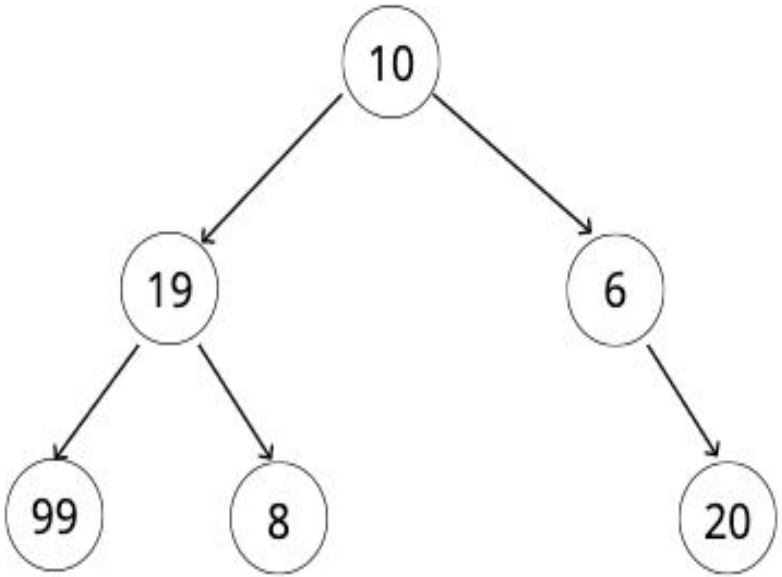
- Adicionar nodo
- Buscar nodo
- Eliminar nodo

Big O de BST

Inserción - **O (log n)**

Buscando - **O (log n)**

NO garantizado!



RECORRIDO DE LOS ÁRBOLES:

VISITA CADA NODO UNA VEZ

RECORRIENDO UN ÁRBOL

Existen 2 formas:

- Búsqueda de amplitud
- Búsqueda en profundidad

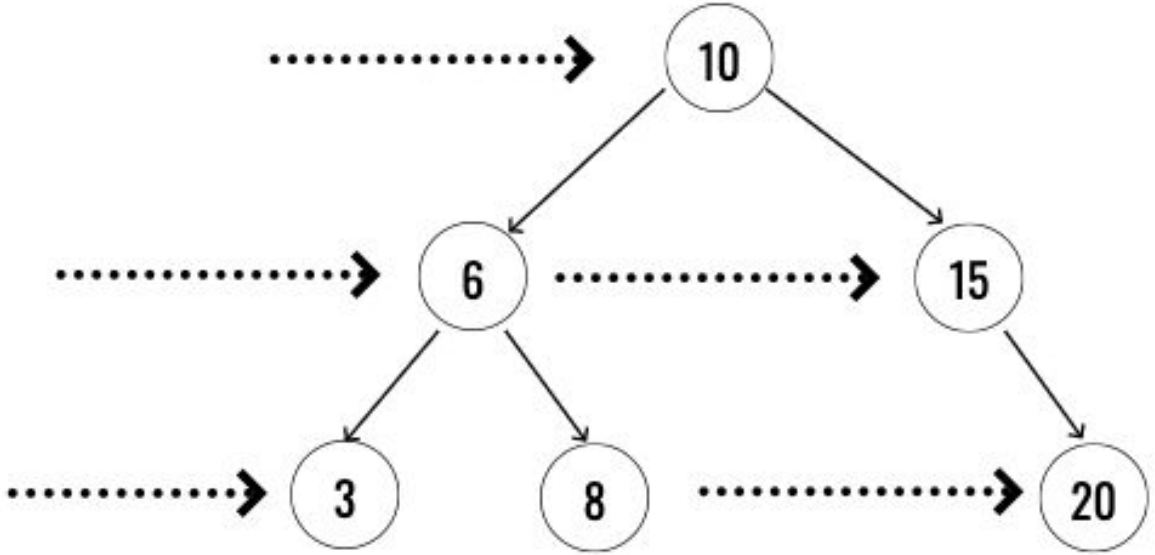
I FORMA: BFS (Breadth - First search)

Pasos - iterativamente

- Cree una cola (esto puede ser una matriz) y una variable para almacenar los valores de los nodos visitados
- Coloque el nodo raíz en la cola
- Bucle siempre que haya algo en la cola
    - Retira un nodo de la cola e inserta el valor del nodo en la variable que almacena los nodos
    - Si hay una propiedad a la izquierda en el nodo retirado de la cola, agréguelo a la cola
    - Si hay una propiedad correcta en el nodo retirado de la cola, agréguelo a la cola
- Devuelve la variable que almacena los valores.

BFS

AMPLITUD Primera busqueda



[10, 6, 15, 3, 8, 20]

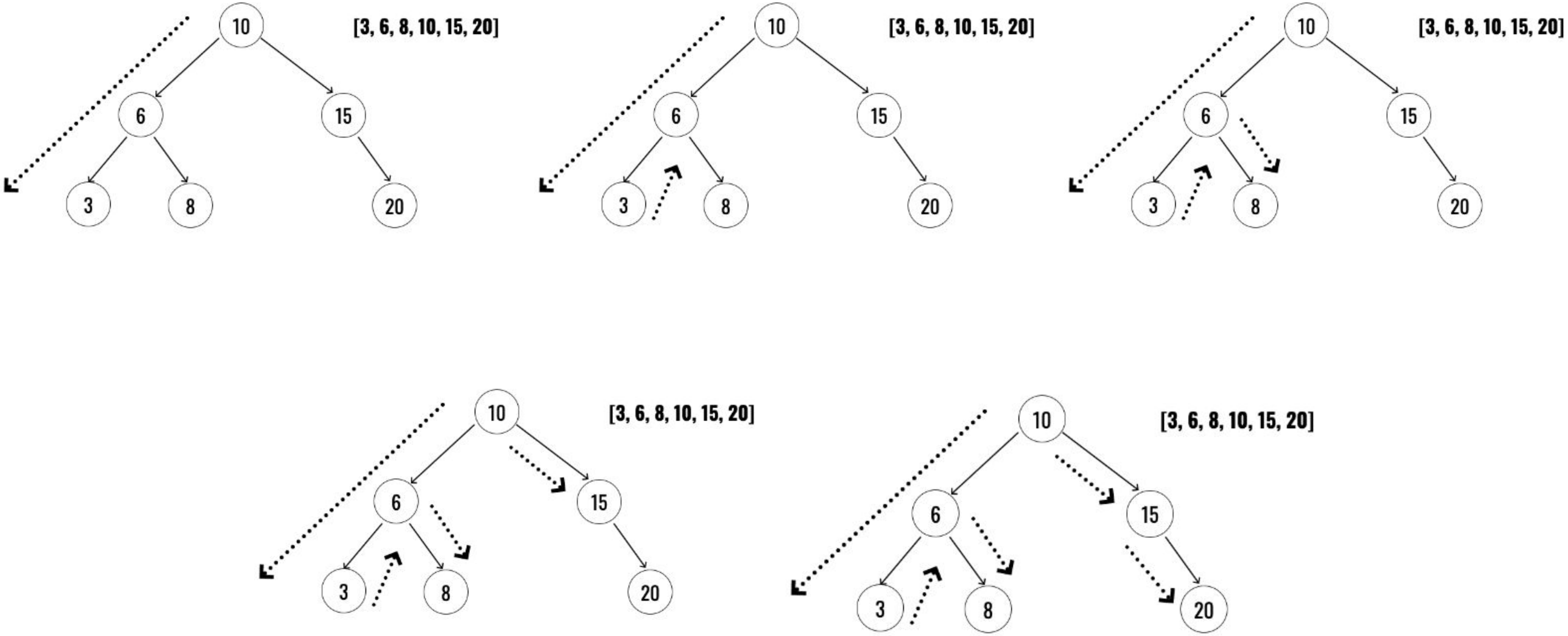
II FORMA DFS (Depth -First Search)

InOrden (Nodo izquierdo, Raiz, Nodo derecho)
PostOrden (Nodo izquierdo, Nodo derecho, Raiz)
PreOrden (Raiz, Nodo izquierdo, Nodo derecho)

DFS - InOrder

- Crea una variable para almacenar los valores de los nodos visitados.
- Almacena la raíz del BST en una variable llamada actual
- Escribe una función auxiliar que acepte un nodo.
    - Si el nodo tiene una propiedad izquierda, llame a la función auxiliar con la propiedad izquierda en el nodo
    - Empuje el valor del nodo a la variable que almacena los valores
    - Si el nodo tiene una propiedad correcta , llame a la función auxiliar con la propiedad correcta en el nodo
- Invoque la función auxiliar con la variable actual.
- Devuelve la matriz de valores

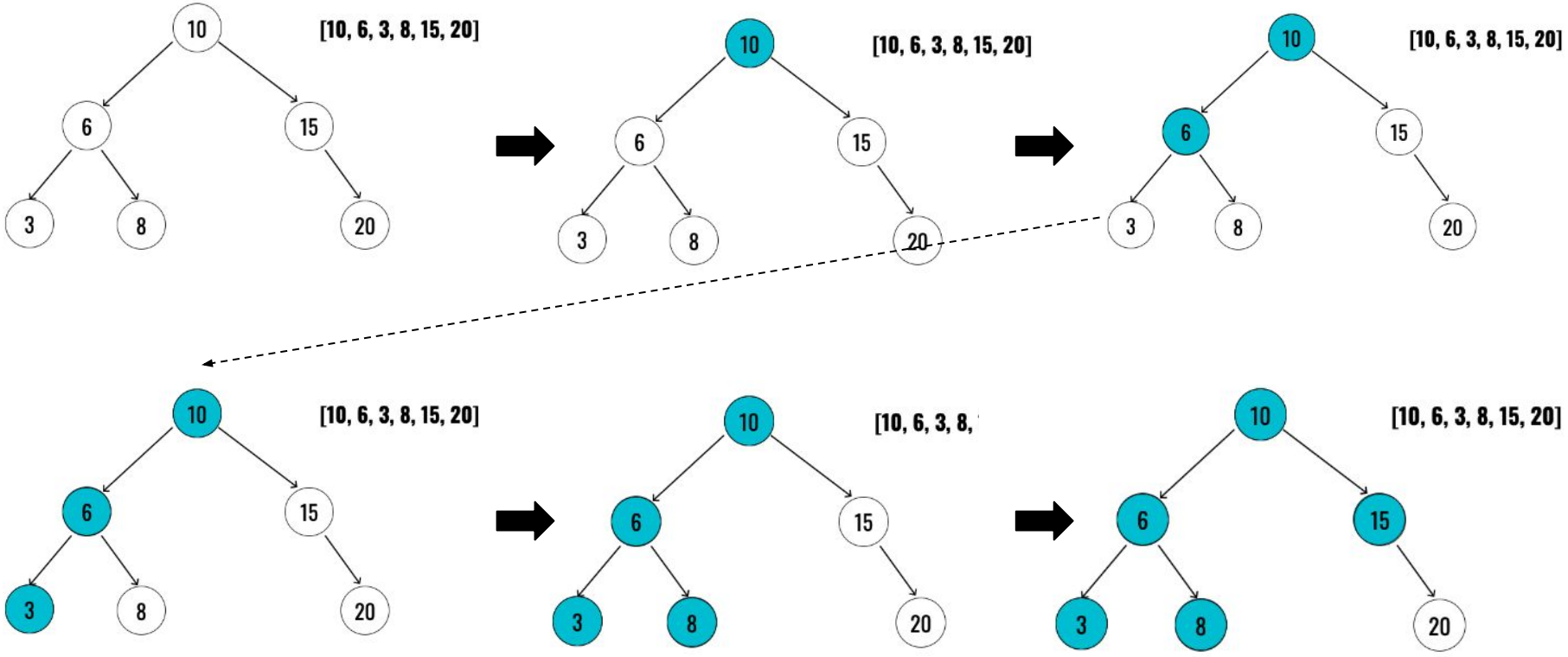
DFS - InOrder


10
6
15
3
8
20
[3, 6, 8, 10, 15, 20]
10
6
15
3
8
20
[3, 6, 8, 10, 15, 20]
10
6
15
3
8
20
[3, 6, 8, 10, 15, 20]
10
6
15
3
8
20
[3, 6, 8, 10, 15, 20]
10
6
15
3
8
20
[3, 6, 8, 10, 15, 20]

DFS - Preorden

- Crea una variable para almacenar los valores de los nodos visitados.
- Almacena la raíz del BST en una variable llamada actual
- Escribe una función auxiliar que acepte un nodo.
    - Empuje el valor del nodo a la variable que almacena los valores
    - Si el nodo tiene una propiedad izquierda, llame a la función auxiliar con la propiedad izquierda en el nodo
    - Si el nodo tiene una propiedad correcta , llame a la función auxiliar con la propiedad correcta en el nodo
- Invoque la función auxiliar con la variable actual.
- Devuelve la matriz de valores

DFS - Preorden



DFS - PostOrder

- Crea una variable para almacenar los valores de los nodos visitados.
- Almacena la raíz del BST en una variable llamada actual
- Escribe una función auxiliar que acepte un nodo.
    - Si el nodo tiene una propiedad izquierda, llame a la función auxiliar con la propiedad izquierda en el nodo
    - Si el nodo tiene una propiedad correcta, llame a la función auxiliar con la propiedad correcta en el nodo
    - Empuje el valor del nodo a la variable que almacena los valores
    - Invoque la función auxiliar con la variable actual.
- Devuelve la matriz de valores

DFS - PostOrder