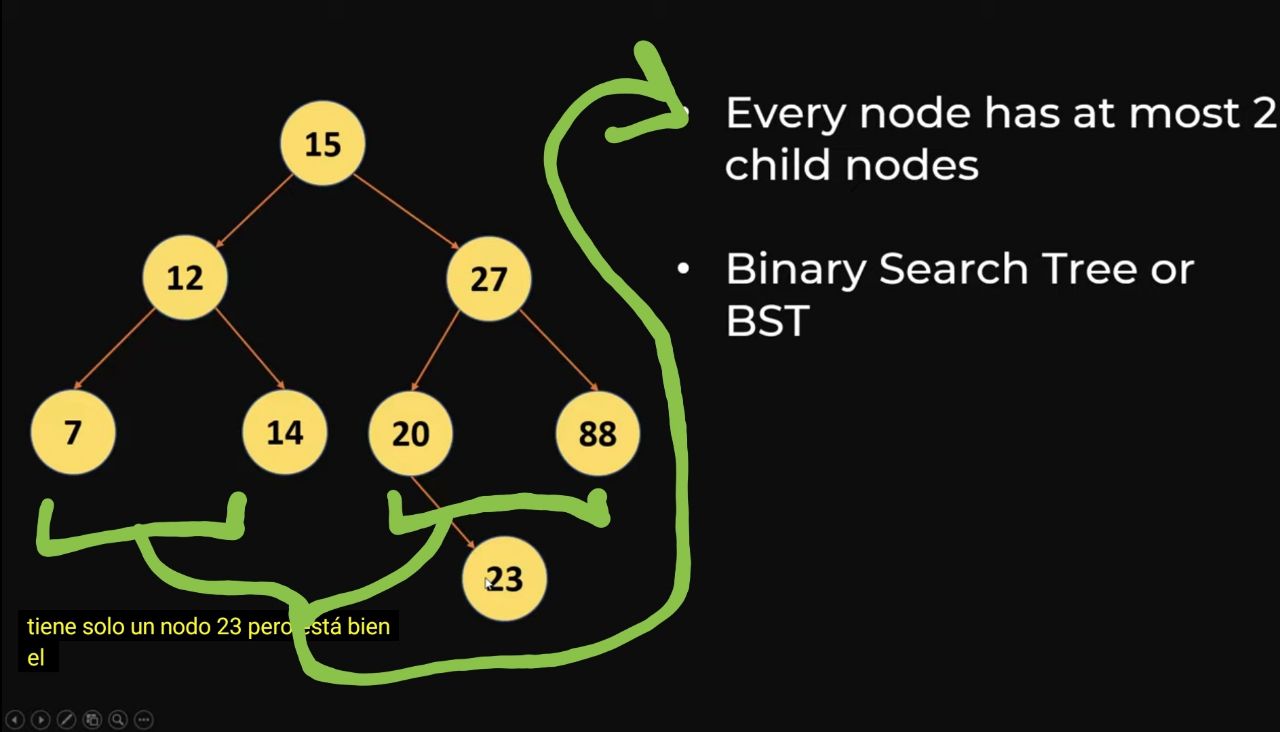
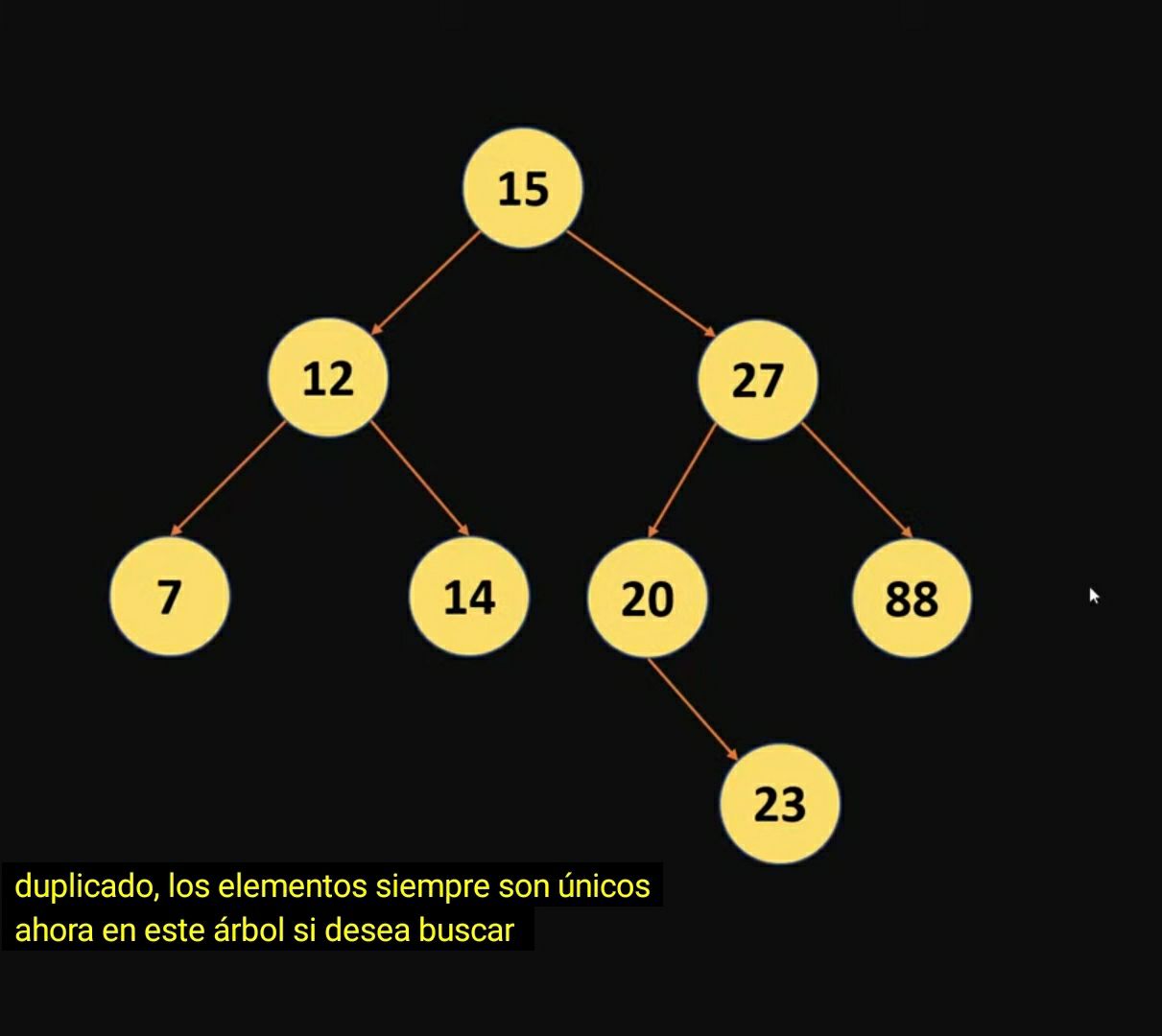
Es un árbol normal, pero con la diferencia de que cada nodo tiene como mínimo dos hijos.

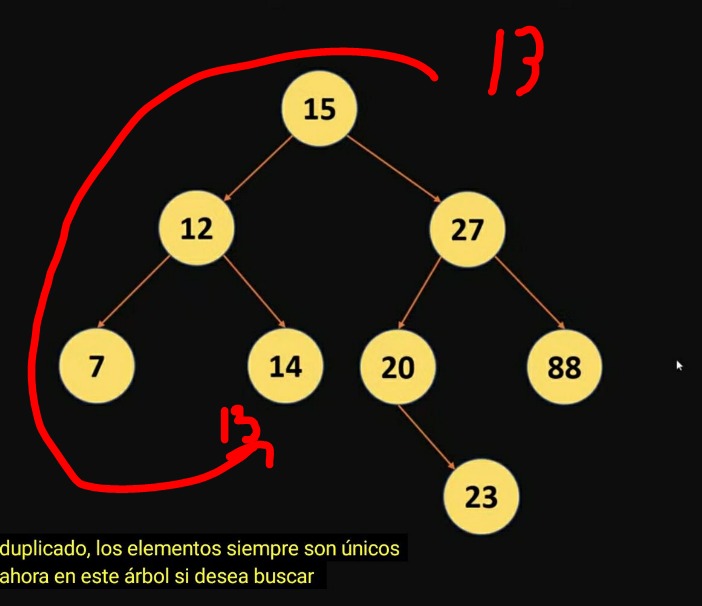


También el árbol tiene un orden, los de la <<<<izquierda<<<< son menores que el HEAD, y los de la >>>>derecha>>>> son mayores que el HEAD, Otra cosa es que los elementos no son duplicados.

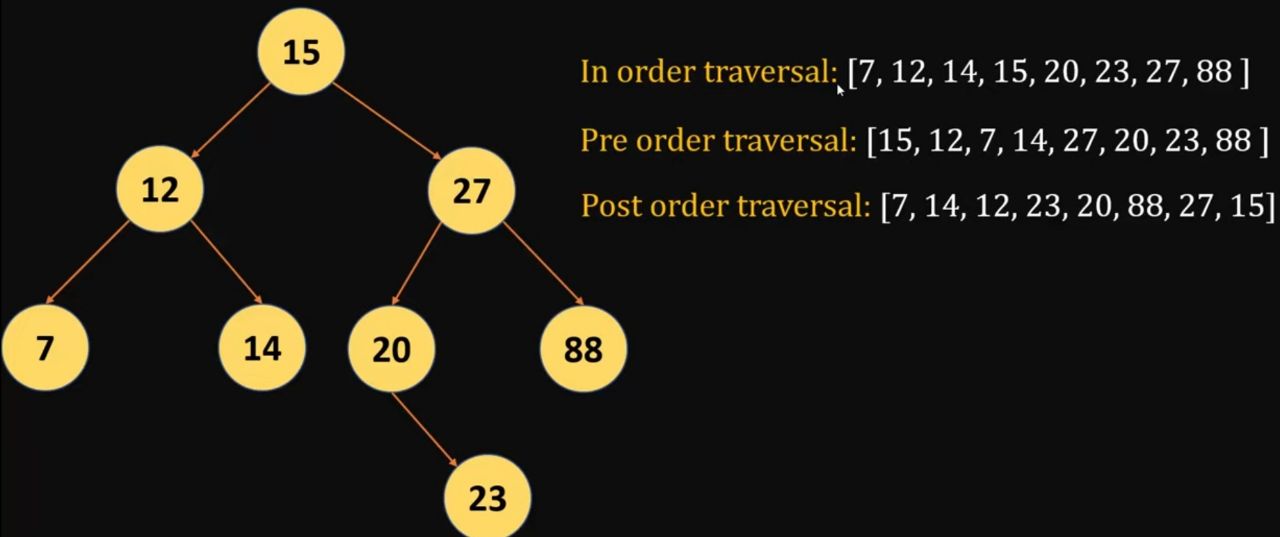


La forma es buscar en un BINARY TREE es la normal, si es menor que el HEAD, buscar en el lado derecho si es mayor buscar el izquierdo.

Nota, si se va a agregar un elemento, el elemento se compara para ver a que extremo pertenece.



Orden de recorrido de un BYNARY TREE



***AL ANALIZAR EL CODIGO, BUSCA EL VIZUALIZADOR Y EL ARCHIVO.PY***

Creación del NODO:

NOTA: por defecto si llamamos al instancia de la clase, se agrega el DATA.

class BinarySearchTreeNode:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

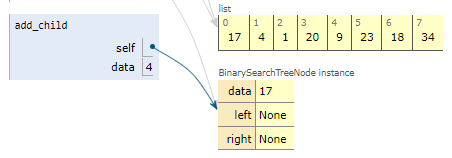
        self.right = None

Función para agregar los NODOS:

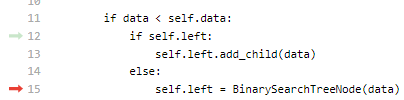
NOTA: Esta DEF se ejecuta tomando en cuenta que ya hay un NODO creado.

Otra cosa es que, al comenzar a iterar los demás elementos, el NODO toma en cuenta que cualquier DATO que estemos pasando después del primer NODO, va a ser vinculado al LEFT.

Observa la img de abajo, donde se está pasando el número 4 y este se vincula al LEFT del nodo 17.



Como el 4 no cumple con los primeros IF, lo que hará es crearse como una instancia, tomando en cuenta que el SELF.LEFT se actualizara:



    def add\_child(self, data):

        if data == self.data:

            return

        if data < self.data:

            if self.left:

                self.left.add\_child(data) # ESTO ES UNA MANERA DE LLAMAR AL LEFT DEL 4

            else:

                self.left = BinarySearchTreeNode(data)

        else:

            if self.right:

                self.right.add\_child(data)

            else:

                self.right = BinarySearchTreeNode(data)

    def in\_order\_traversal(self):

        elements = []

        if self.left:

            elements += self.left.in\_order\_traversal()

        elements.append(self.data)

        if self.right:

            elements += self.right.in\_order\_traversal()

        return elements

def build\_tree(elements):

    root = BinarySearchTreeNode(elements[0])

    for i in range(1, len(elements)):

        root.add\_child(elements[i])

    return root

numbers = [17, 4, 1, 20, 9, 23, 18, 34, 18, 4]

numbers\_tree = build\_tree(numbers)

print(numbers\_tree.in\_order\_traversal())



NOTA: FIJATE COMO SE HAN REMOVIDO LOS DUPLICADOS

    def search(self, val):

        if self.data == val:

            return True

        if val < self.data:

            if self.left:

                self.left.search(val)

        if val > self.data:

            if self.right:

                return self.right.search(val)

            else:

                return False

print(numbers\_tree.search(20))

al igual que en el algoritmo binario, su estructura siempre es la misma

BUSCAR EL MAX Y EL MIN

Nota: EL MAX SIEMPRE ESTARA EN LA PUNTA DERECHA, Y EL MIN EN LA PUNTA IZQUIERDA

El siguiente código, se ejecutará de forma recursiva, hasta encontrar el ultimo nodo de una de las dos puntas del árbol, y lo va a ejecutar hasta que su left o right sea igual a NONE o que no tengan ningún elemento.

    def find\_max(self):

        if self.right is None:

            return self.data

        return self.right.find\_max()

    def find\_min(self):

        if self.left is None:

            return self.data

        return self.left.find\_min()

    def find\_min(self):

        chico = []

        if self.left:

            chico += self.left.find\_min()

        chico.append(self.data)

        if self.right:

            chico += self.right.find\_min()

        return chico

    def find\_max(self):

        chico = []

        if self.left:

            chico += self.left.find\_max()

        chico.append(self.data)

        if self.right:

            chico += self.right.find\_max()

        return chico

    def compute\_sum(self):

        chico = []

        if self.left:

            chico += self.left.compute\_sum()

        chico.append(self.data)

        if self.right:

            chico += self.right.compute\_sum()

        return chico

    def find\_min\_v2(self):

        chico = []

        if self.left:

            chico += self.left.in\_order\_traversal()

        chico.append(self.data)

        if self.right:

            chico += self.right.in\_order\_traversal()

        return min(chico)

    def find\_max\_v2(self):

        chico = []

        if self.left:

            chico += self.left.in\_order\_traversal()

        chico.append(self.data)

        if self.right:

            chico += self.right.in\_order\_traversal()

        return max(chico)

    def sum\_v2(self):

        chico = []

        if self.left:

            chico += self.left.in\_order\_traversal()

        chico.append(self.data)

        if self.right:

            chico += self.right.in\_order\_traversal()

        return sum(chico)

'''

minimo = numbers\_tree.find\_min()

print(min(minimo))

maximo = numbers\_tree.find\_max()

print(max(maximo))

suma = numbers\_tree.compute\_sum()

print(sum(suma))

print(numbers\_tree.find\_min\_v2())

print(numbers\_tree.find\_max\_v2())

print(numbers\_tree.sum\_v2())

'''

BORRAR un elemento de un BINARY TREE

    def delete(self, val):

        if val < self.data:

            if self.left:

                self.left = self.left.delete(val)

        elif val > self.data:

            if self.right:

                self.right = self.right.delete(val)

        else:

            if self.left is None and self.right is None:

                return None

            elif self.left is None:

                return self.right

            elif self.right is None:

                return self.left

            min\_val = self.right.find\_min()

            self.data = min\_val

            self.right = self.right.delete(min\_val)

        return self

GUIA DE EJECUCION:

En este caso estamos borrando el NODO 9, el cual en su última aparición, era RETURN NONE:

    def delete(self, val): # 17, 9| 4, 9| 9, 9|

        if val < self.data: #17 > 9|

            if self.left: # 4

                self.left = self.left.delete(val) # 4, 9|

        elif val > self.data: # 9> 4|

            if self.right: # 9

                self.right = self.right.delete(val) # 9, 9| 9 = none

        else: #

            if self.left is None and self.right is None: # 9 = si

                return None # 9 = none

            elif self.left is None: #

                return self.right #

            elif self.right is None: #

                return self.left #

            min\_val = self.right.find\_min() #

            self.data = min\_val #

            self.right = self.right.delete(min\_val) #

        return self #

En este caso, hemos eliminado un nodo que estaba en el 1er nivel, y hemos subido los valores de los anteriores niveles. Ejemplo, el 20 paso a ser el 23.

    def delete(self, val): #17, 20| 20, 20| 23, 23|

        if val < self.data: #

            if self.left: #

                self.left = self.left.delete(val) #

        elif val > self.data: # 20 > 17|

            if self.right: # 20

                self.right = self.right.delete(val) # 20, 20|

        else: #

            if self.left is None and self.right is None: #

                return None #

            elif self.left is None: # SI

                return self.right # 34

            elif self.right is None: #

                return self.left #

            min\_val = self.right.find\_min() #23 | 23 o NONE

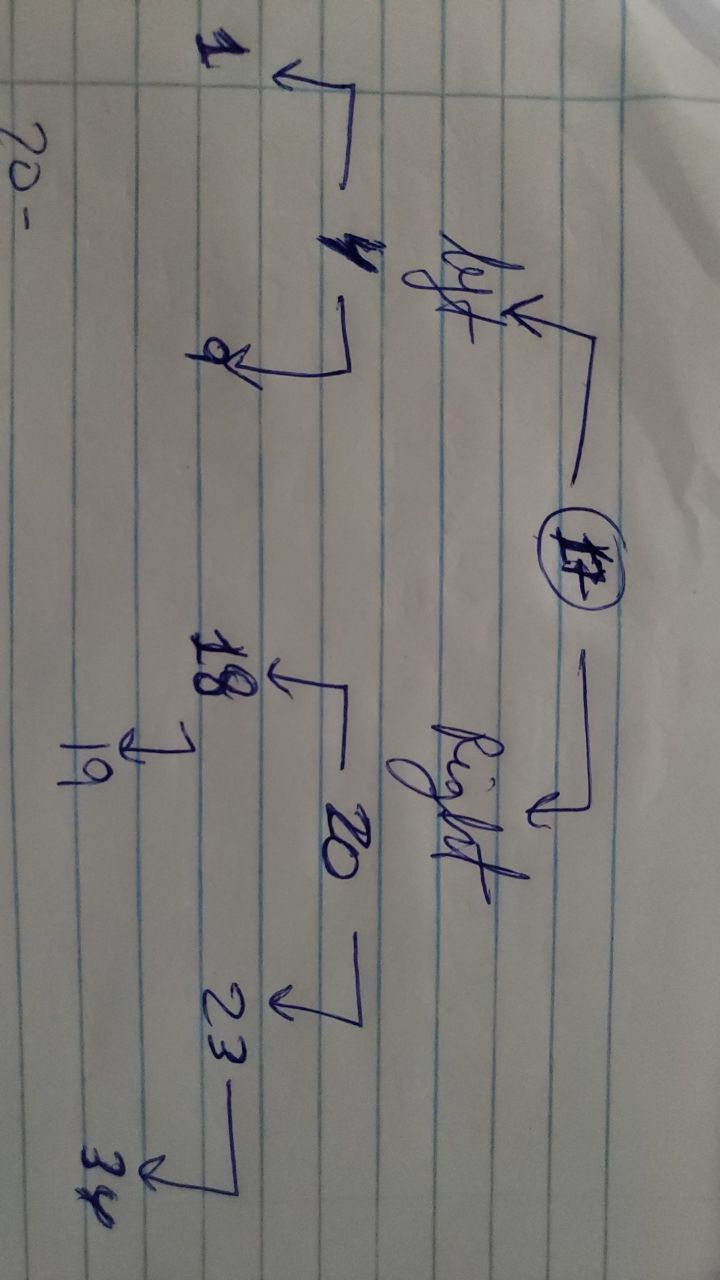
            self.data = min\_val # 20 = 23|

            self.right = self.right.delete(min\_val) #23, 23| 23 + 34

        return self #

HAY OTRA FORMA QUE ES BUSCAR EL ELEMENTO MAS GRANDE, E INTERCAMBIARLO POR EL ELEMENTO A BORRAR.

EN ESTE CASO, HE BORRADO EL ELEMENTO 20 POR EL 19.



    def delete(self, val): #17, 20| 20, 20| 18, 19| # 19, 19

            if val < self.data: #

                if self.left: #

                    self.left = self.left.delete(val) #

            elif val > self.data: # 20 > 17|

                if self.right: # 20 | 19

                    self.right = self.right.delete(val) # 19, 19| 19 = NONE

            else: #

                if self.left is None and self.right is None: #

                    return None # 19 = NONE

                elif self.left is None: #

                    return self.right #

                elif self.right is None: #

                    return self.left #

                min\_val = self.left.find\_max() # 19

                self.data = min\_val # 20 = 19

                self.left = self.left.delete(min\_val) #18, 19| 19 = NONE

            return self #