Árbol binario

Es una estructura de datos jerárquica (que va por niveles) que está compuesta por nodos, cada nodo tiene 2 hijos (el hijo izquierdo y el hijo derecho). Hay un nodo principal (El que contiene a todos los nodos y es el punto de acceso) en este libro lo llamaremos raíz. Y cada hijo del nodo principal puede tener otros hijos (1 a la derecha y uno a la izquierda).

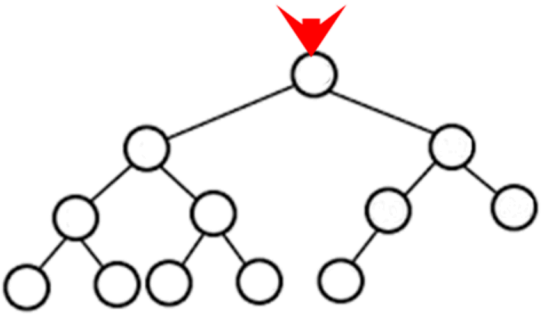


FIG 00: representación de la estructura de un árbol binario

ADVERTENCIA
LOS ÁRBOLES SE RIGEN POR LA SIGUIENTE REGLA:
- SI ES MÁS GRANDE VA A LA DERECHA.
- SI ES MENOR VA A LA IZQUIERDA

Advertencia

Esto solo funciona con recursividad
En caso de no entender recursividad es imposible entender los árboles

Por qué existen los árboles:

- Organización y eficiencia: Permiten organizar datos de manera que las operaciones de búsqueda/inserción/eliminación puedan realizarse eficientemente.
- Tienen relaciones jerárquicas.
- Implementan otras estructuras: Árboles binarios de búsqueda, montículos, Árboles AVL.
- Facilidad de recorridos: Permiten diferentes formas de recorrer los nodos: en orden, pre orden, post orden.

Un árbol no es más que una colección de nodos, los cuales contienen una izquierda y una derecha y almacenan un dato. Por ello cada nodo contiene 3 elementos:

- Izquierda.
- dato.
- Derecha.

FIG 01: representación de un nodo al lado de su respectivo código.

Para poder utilizar esto necesitamos crear algo que se llama "Árbol"

Una Árbol es una colección de nodos que nos va a permitir guardar la información, dicha clase contiene un apuntador (el principal es la raíz)… si observa la siguiente figura usted podrá notar cómo funciona:

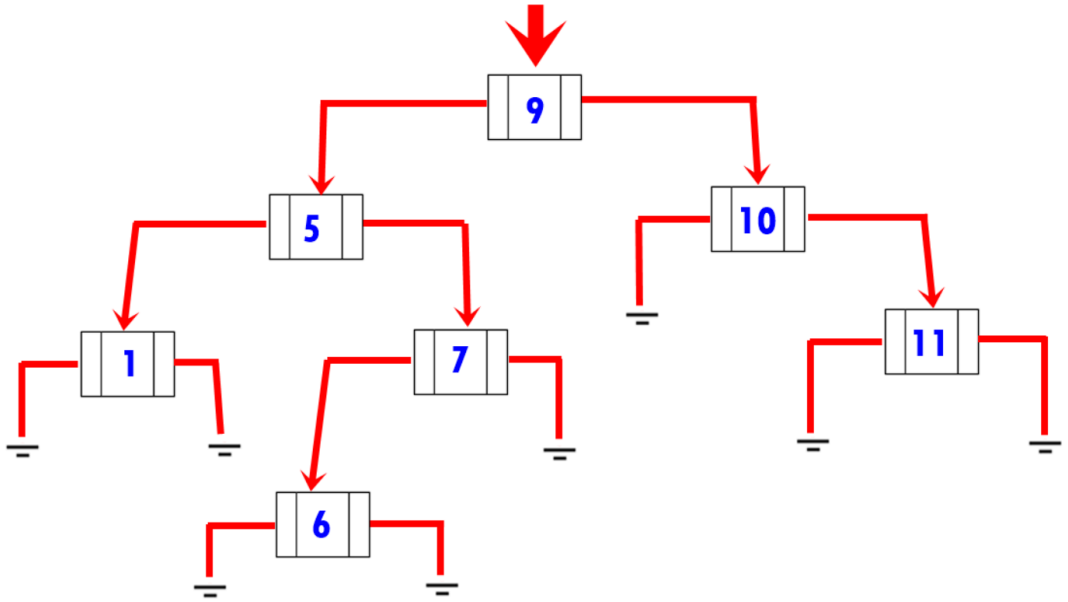


FIG 02: ejemplo visual de un árbol.

Los árboles necesitan de los siguientes atributos y métodos para poder funcionar:

- Se necesita un pivote el cual es el encargado de moverse para leer/escribir la información.
- Se necesita un método para insertar un valor.
- Se necesita un método para buscar un valor.
- Se necesita un método para eliminar un valor.
- Se necesitan 3 métodos para recorrer el árbol: (In order, Pre Order y Post Orden).
- Se necesita un método para calcular la altura de un árbol.
- Se necesita un método para contar los nodos de un árbol.
- Se necesita de un método para buscar los valores mínimo y máximo.

Que es el pivote

El pivote no es más que un nodo el cual vamos a utilizar para poder movernos entre los Nodos de ese árbol (Moverse entre las ramas) con este movimiento nosotros vamos a poder:

- Agregar, Eliminar, Buscar y Modificar los datos de un nodo.

El pivote es el encargado de poder ser el punto de acceso para movernos en el árbol, sin él no tendríamos una forma de recorrer el árbol.

```python
class Tree:
    def __init__(self) -> None:
        self.root = None
```

Advertencia: El nodo principal inicia en None debido a que no existe DATA para construir el primer nodo.

Como se agregan los elementos a un árbol

La única manera de lograr la inserción de datos en un árbol binario es mediante la recursividad debido a que tenemos que hacer una PILA de métodos para poder agregar pero acá lo veremos paso a paso, pero antes vamos a ver las dos posibilidades:

- **Caso 1:** el árbol está vacío: entonces es muy simple “solamente se declara la raíz como un nuevo nodo”

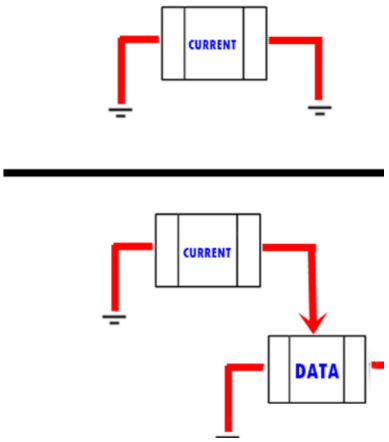
```python
def insert(self, data):
    if self.root == None:
        self.root = Node(data)
```

- **Caso 2:** el árbol ya contiene información: Pues entonces tenemos que hacer uso de “El 100% de nuestra capacidad cerebral” y usar la recursividad para solucionar los siguientes subcasos:

  - Sub caso 2.1: el dato que queremos agregar es más grande que el dato que contiene el nodo actual y el lado derecho está disponible para almacenar la información:

CURRENNT < DATA

```python
def _insert(self, pivot, data):
    if data > pivot.data:
        if pivot.right == None:
            pivot.right = Node(data)
```

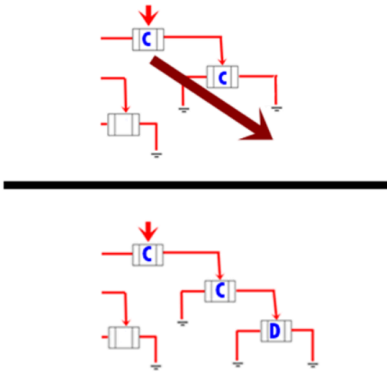


  - Sub caso 2.2: el dato que queremos agregar es más grande que el dato que contiene el nodo actual pero al lado derecho no hay espacio disponible:

  “Pues entonces en ese caso volvemos a llamar a la recursión a la derecha”

CURRENNT < DATA

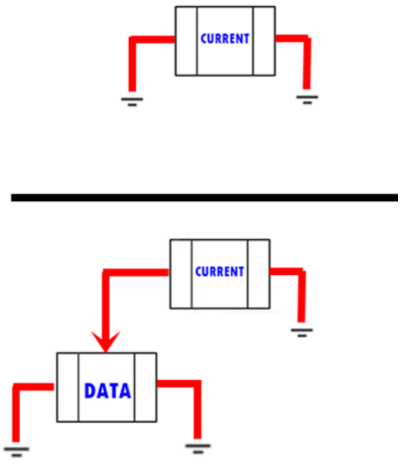
```
def _insert(self, pivot, data):
    if data > pivot.data:
        if pivot.right == None:
            pivot.right = Node(data)
        else:
            self._insert(pivot.right, data)
```



- Caso 2.3: el dato que queremos agregar es más pequeño que el dato que contiene el nodo actual pero al lado izquierdo está desocupado:

CURRENT > DATA

```
def _insert(self, pivot, data):
    if data > pivot.data: ···
    else:
        if pivot.left == None:
            pivot.left = Node(data)
```

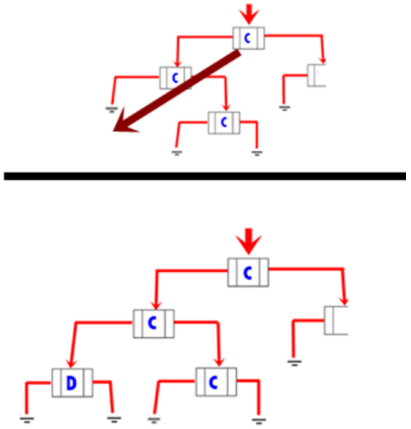


- Caso 2.4: el dato que queremos insertar es más pequeño que el dato que contiene el nodo actual pero el lado izquierdo se encuentra ocupado:

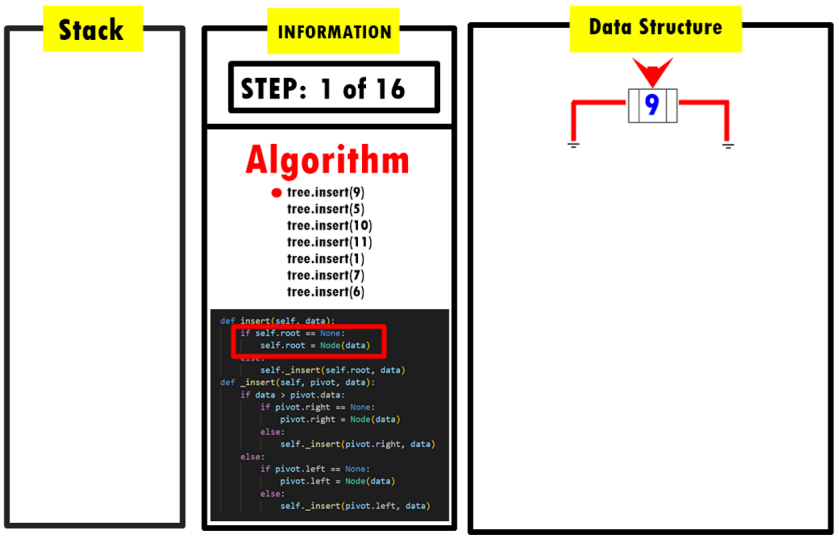
“Pues entonces en ese caso volvemos a llamar a la recursión a la izquierda”

CURRENT > DATA

```
def _insert(self, pivot, data):
    if data > pivot.data: ···
    else:
        if pivot.left == None:
            pivot.left = Node(data)
        else:
            self._insert(pivot.left, data)
```

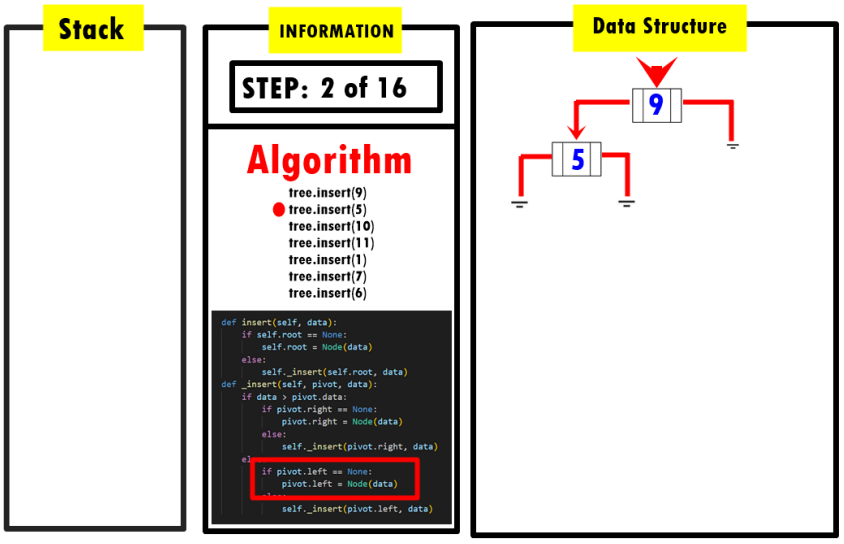


El lector notará que es complejo, pero la verdad solo hay una forma de entender esto y es con lápiz y papel hacer el seguimiento de inserción de los algoritmos. A continuación vamos a ver como se agregan los elementos 9,5,10,11,1,7,6 a un árbol binario.



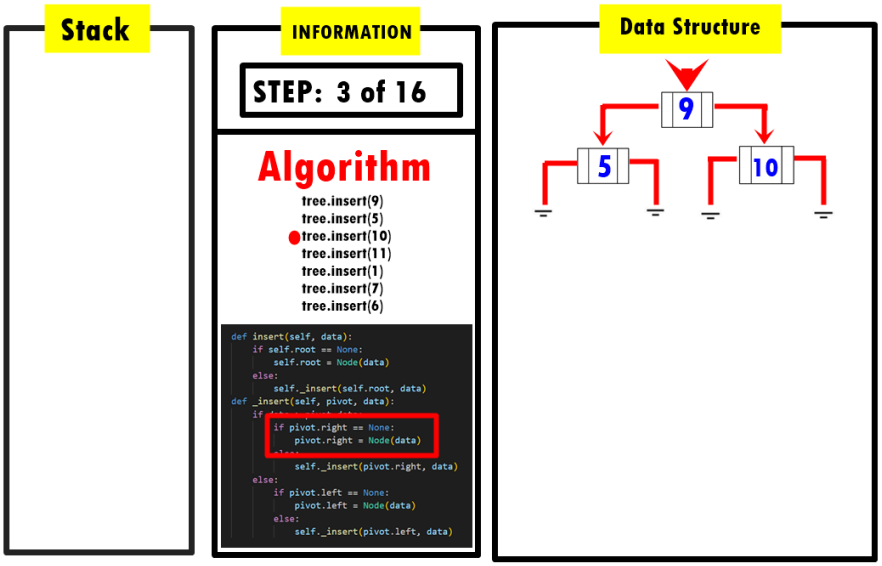
Agregar elemento 9 de [9,5,10,11,1,7,6]

El primer elemento es el más fácil de agregar puesto que solo basta con crear un nodo y establecerlo como la raíz.



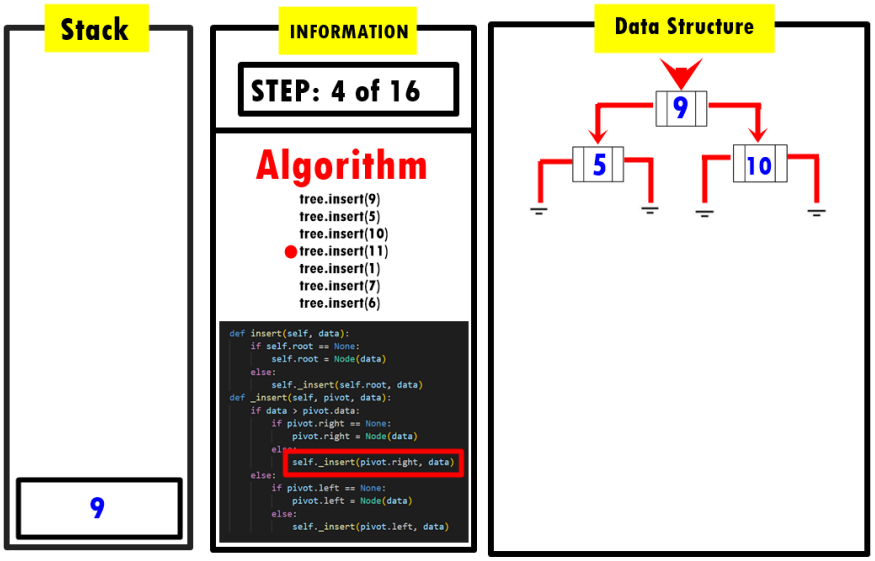
Agregar elemento 5 de [9,5,10,11,1,7,6]

Para agregar un elemento cuando ya existe una raíz se tiene que hacer uso de la recursividad y eso se hace por que tenemos que empezar a buscar por la derecha o la izquierda cuál será la posición correcta de nuestro dato. En este caso como el 5 es menor que el nueve va a la izquierda.



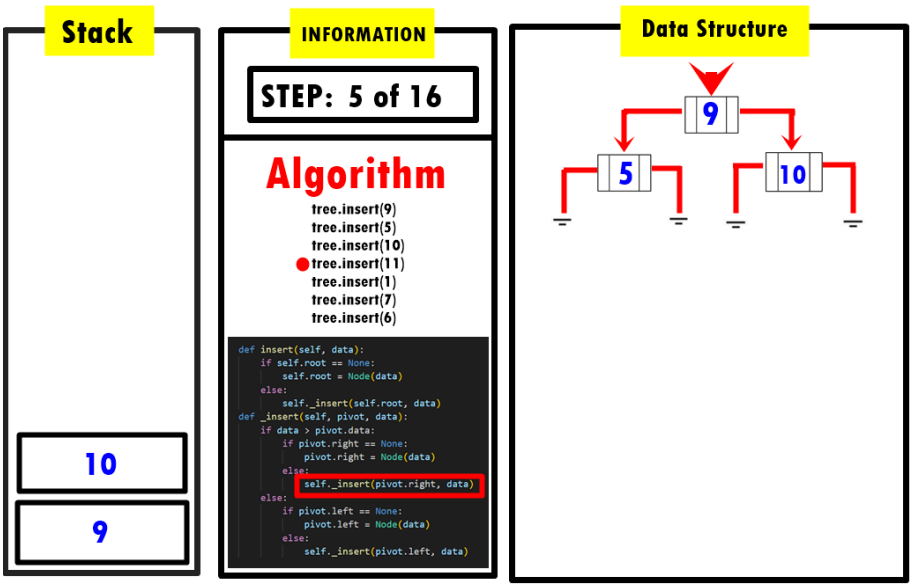
Agregar elemento 10 de [9,5,10,11,1,7,6]

Cuando se desea agregar un elemento a un árbol se hace uso de la recursividad y siempre empezamos por la raíz, el árbol empieza en 9 y basados en las reglas del árbol el 10 es mayor que el 9 por lo tanto va a la derecha.

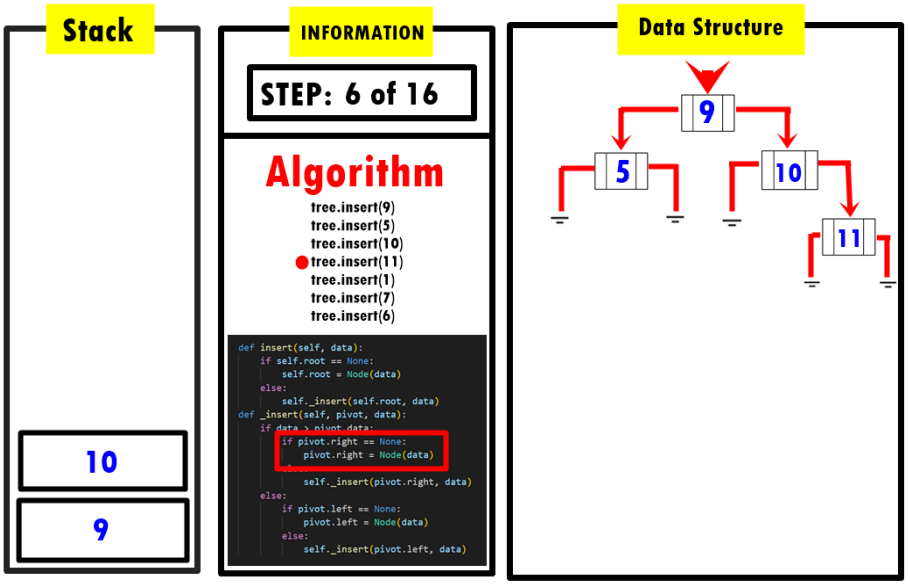


Agregar elemento 11 de [9,5,10,11,1,7,6]

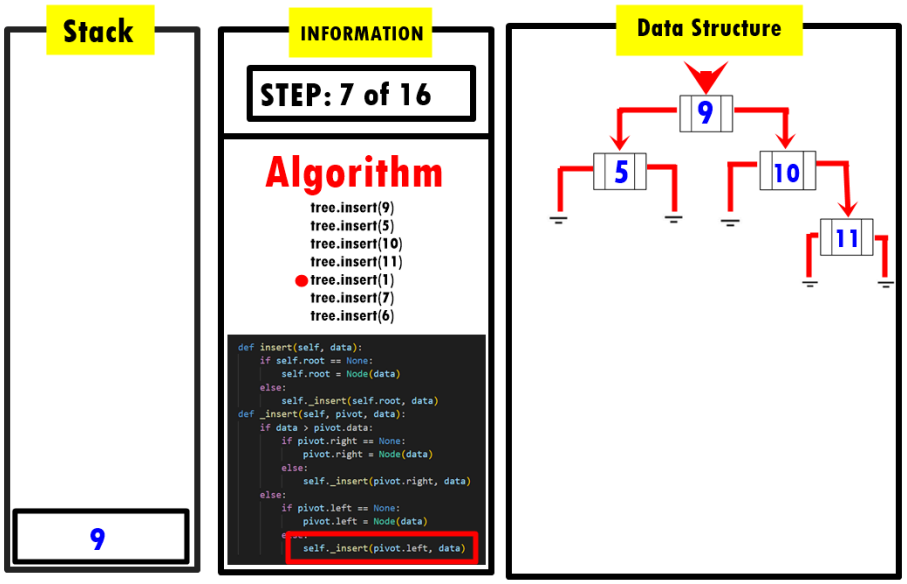
Tenemos que usar la recursión a la derecha debido a que al lado derecho del 9 se encuentra el 10, por ello tenemos que movernos hacia la derecha para buscar el espacio disponible.



Usando la recursión estamos parados en el 10.

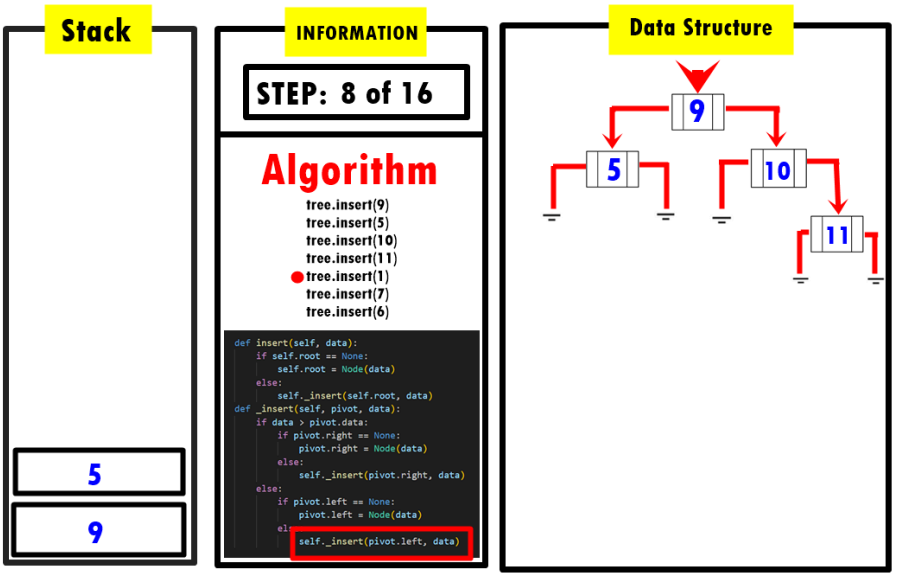


Tuvimos que recorrer 2 nodos para poder tener una posición libre para almacenar nuestro número 11, como este número es más grande procederemos a almacenarlo a la derecha.

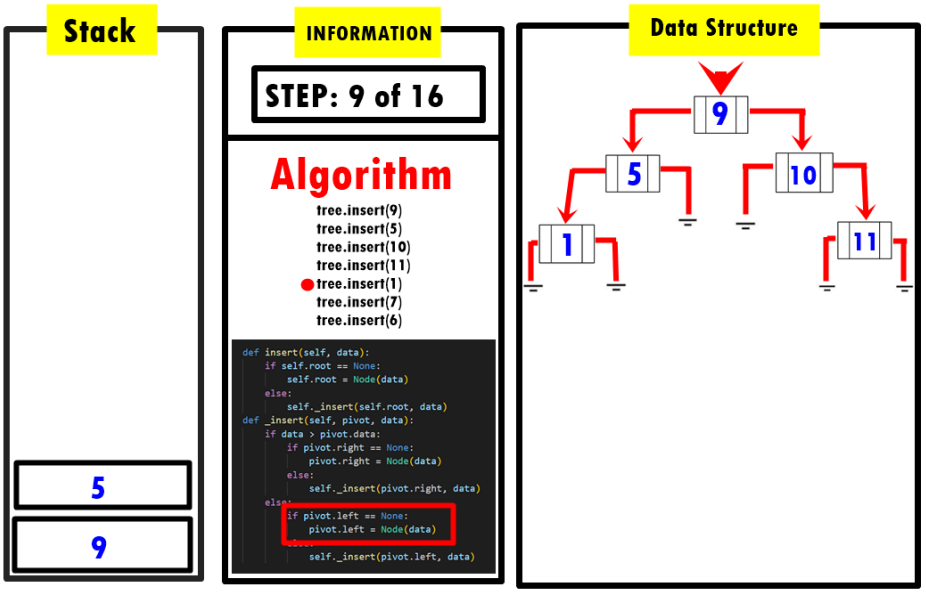


Agregar elemento 1 de [9,5,10,11,1,7,6]

El elemento que queremos agregar es menor que nueve por ello tenemos que movernos a la izquierda.

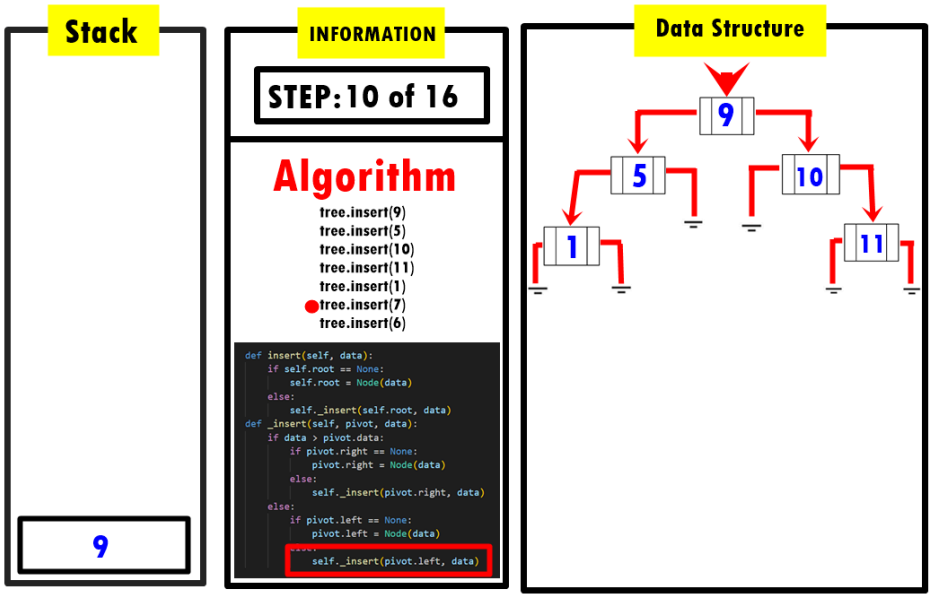


El espacio a la izquierda del 9 ya se encuentra ocupado por el 5, además el valor que se quiere insertar el uno es menor que el 5 por ello procederemos a movernos otra vez a la izquierda.

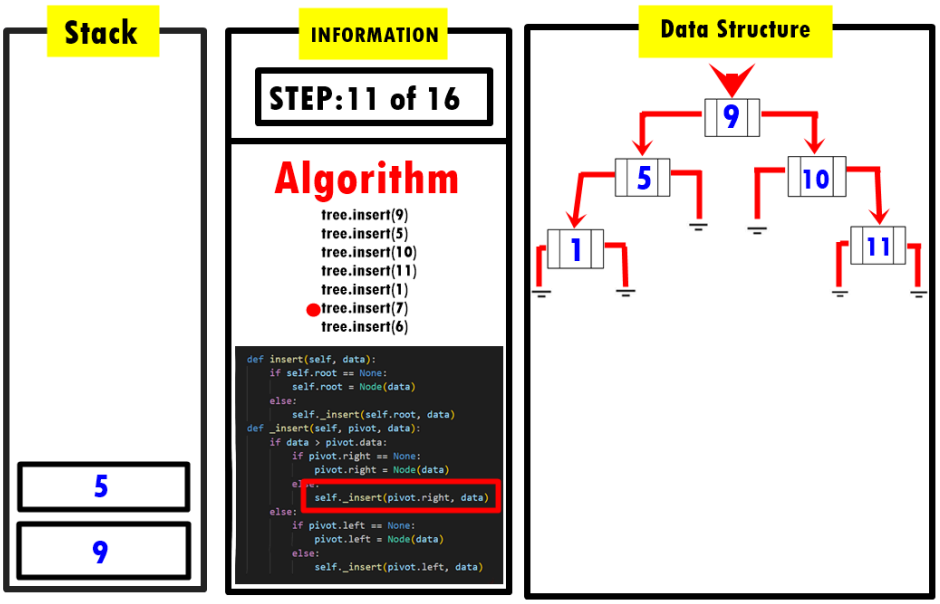


Agregar elemento 7 de [9,5,10,11,1,7,6]

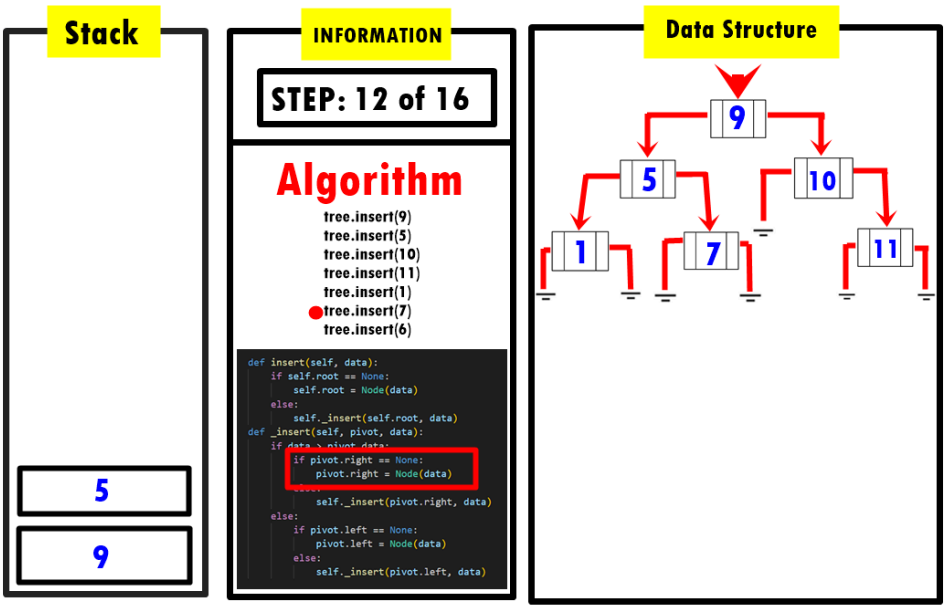
Cuando nos movimos para el cinco existe un espacio disponible a la izquierda.



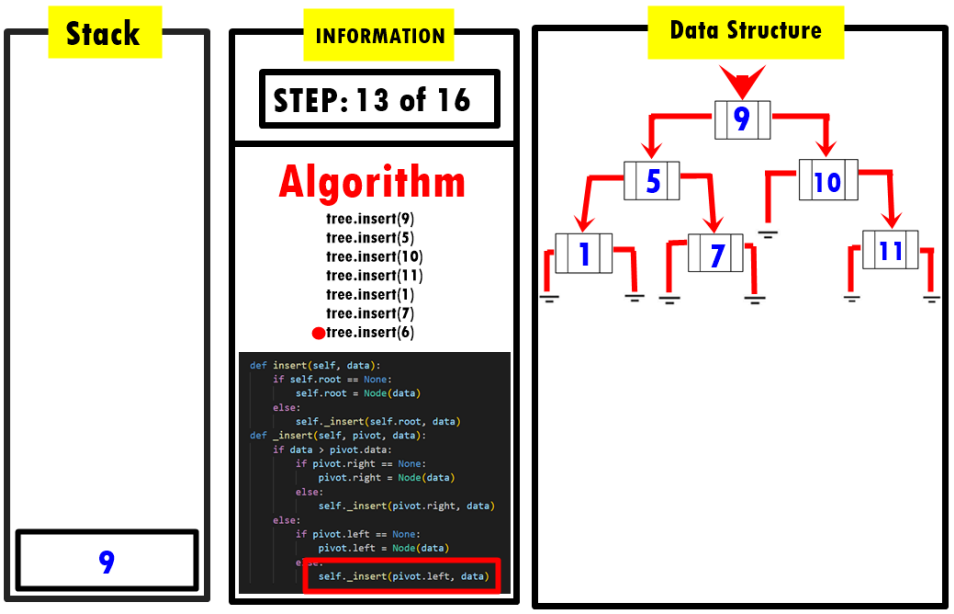
Ahora vamos a insertar el 7 en el árbol, el nodo raíz es el primero que se agrega a la pila, como el número que hay en la raíz es mayor que el que se desea agregar entonces lo que hacemos es movernos hacia la izquierda.



Ahora estamos parados en el 5 y deseamos agregar el número 7 entonces lo que tenemos que hacer es agregarlo a la derecha debido a que es más grande que el dato del nodo actual.

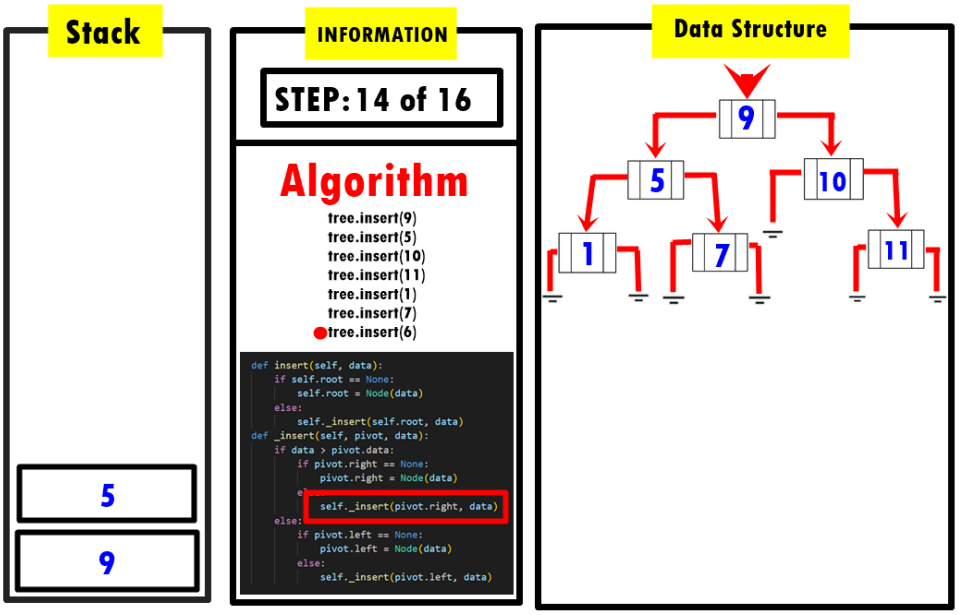


Se agrega el 7 a la derecha del 5.

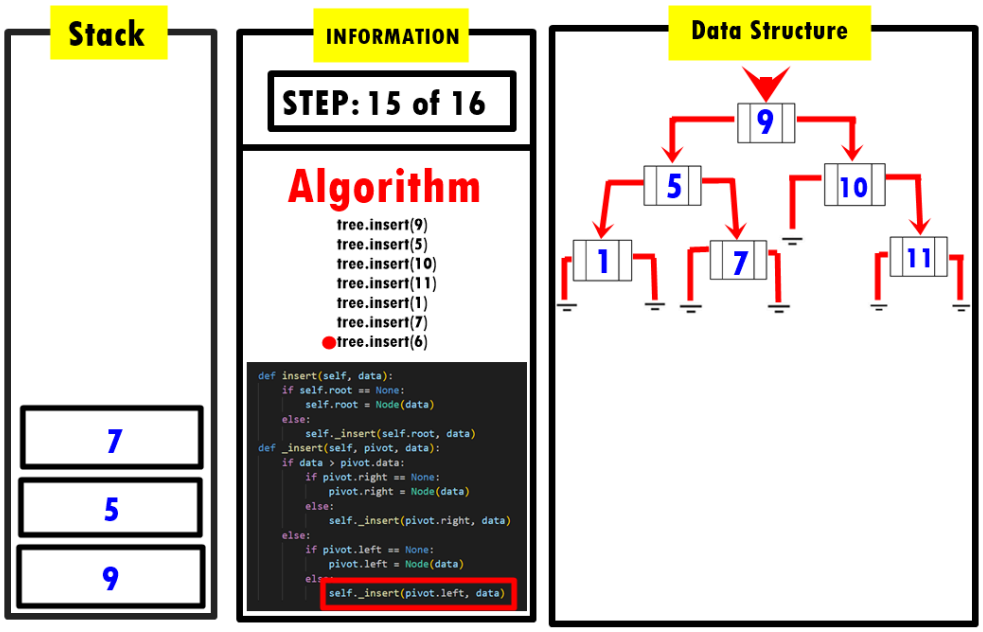


Agregar elemento 6 de [9,5,10,11,1,7,6]

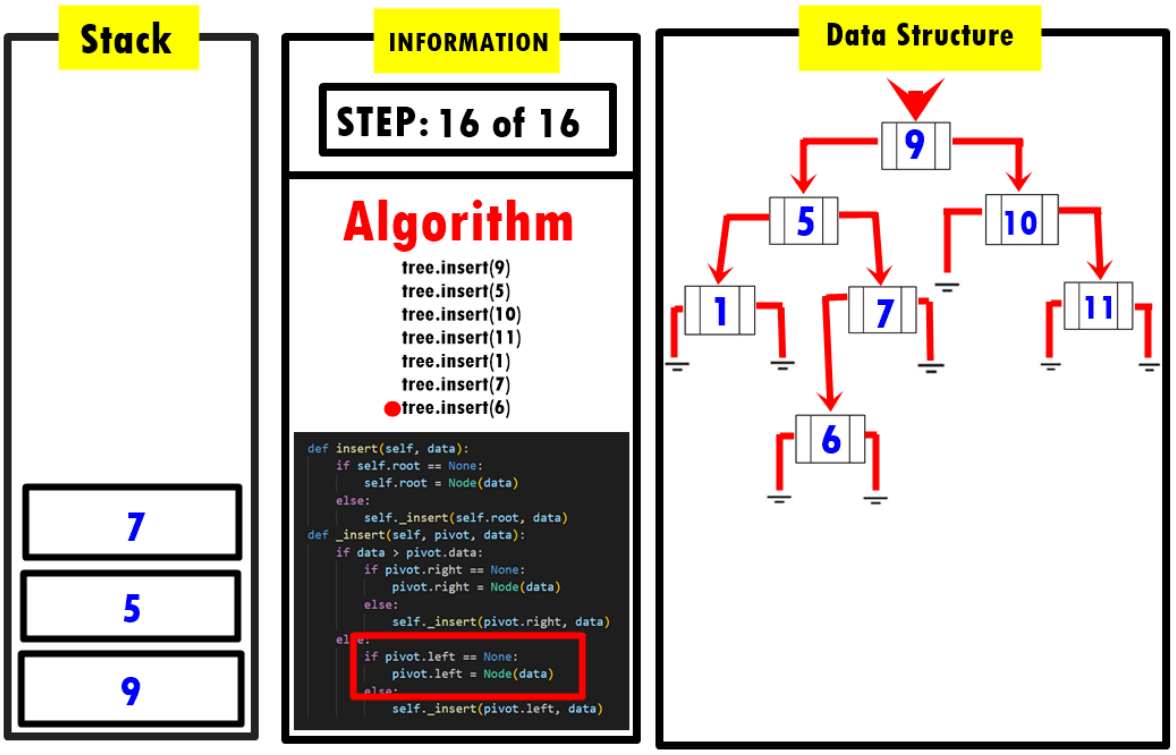
Estamos en este momento en la raíz del árbol, lo agregamos a la pila y nos movemos a la izquierda debido a que el elemento es menor.



Ahora estamos en el nodo 5 y no hay un espacio disponible a la izquierda por ello tenemos que volver a movernos hacia la derecha.



Ahora estamos parados en el 7 y la buena noticia es que tenemos un espacio disponible a la izquierda por ello vamos a agregarlo.



El elemento ha sido agregado a la izquierda y con ello agregamos los 7 nodos que planteamos al principio. [9,5,10,11,1,7,6]

Buscar

En realidad lo que vamos a hacer es retornar un nodo según su valor de datos, y para esto tenemos que hacer uso de la recursividad con la siguiente lógica:

- Si el dato es igual al valor del pivote.dato entonces retornaremos el pivote en caso contrario solo hay 2 posibles opciones: “El valor es mayor ó El valor es menor”
- En caso de que el valor que tenemos es mayor que el pivote pues entonces procederemos a enviar recursivamente a buscarlo en el nodo de la derecha.
- En caso de que el valor que tenemos es menor que el pivote pues entonces procederemos a enviar recursivamente a buscarlo en el nodo de la izquierda.

```
def searchByData(self, data):
    return self._searchByData(self.root, data)
def _searchByData(self, pivot,data):
    if pivot != None:
        if pivot.data == data:
            return pivot
        else:
            if pivot.data < data:
                return self._searchByData(pivot.right, data)
            else:
                return self._searchByData(pivot.left, data)

    return None
```

Eliminar

Este es un proceso más complicado, un proceso largo y complejo el cual enfrentaremos. En el mundo de los árboles y las estructuras de datos, para eliminar un nodo hay que tener en cuenta múltiples casos base además de hacer búsquedas para intercambiar nodos de posición.

Caso 1
El nodo a eliminar no tiene hijos “es una hoja”

Este es el caso más simple de todos lo único que se hace es:

“Hacer que el padre apunte su próximo (Entiéndase por próximo la dirección donde se encuentra el hijo, ya sea que su hijo se encuentre a la derecha ó izquierda) se reemplazará por None.

```
def _del_node(self, node):
    if self.isTheNodeALeaf(node):
        return None
```

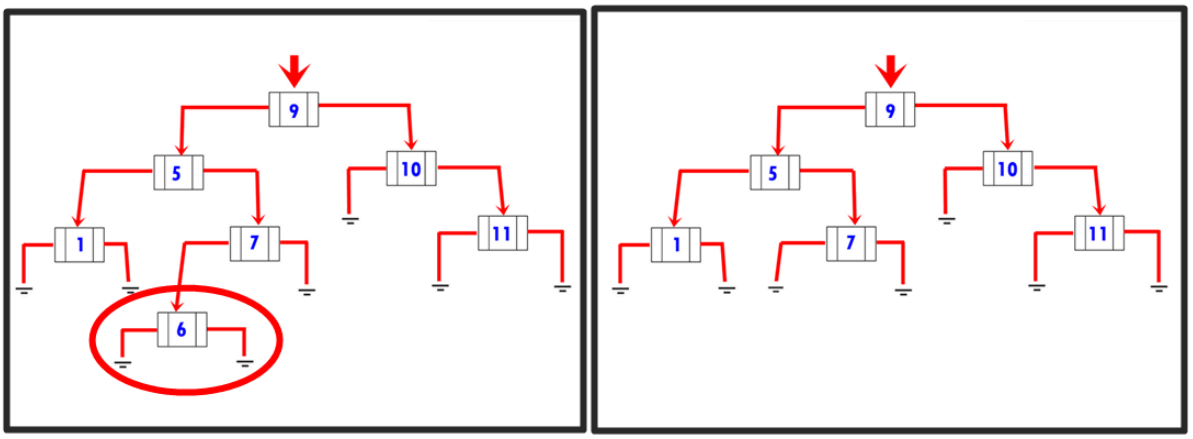


FIG 03: Ejemplo de cómo eliminar una hoja “Nodo con data = 6”

Caso 2
El nodo a eliminar solo tiene 1 hijo

Este caso implica al padre y los hijos del nodo a eliminar, lo que tenemos que hacer es que el padre apunte al hijo del nodo a eliminar “RECUERDA QUE EL NODO A ELIMINAR SOLO TIENE UN HIJO”

```
def _del_node(self, node):
    if self.isTheNodeALeaf(node): ···
    if node.left is None:
        return node.right
    if node.right is None:
        return node.left
```

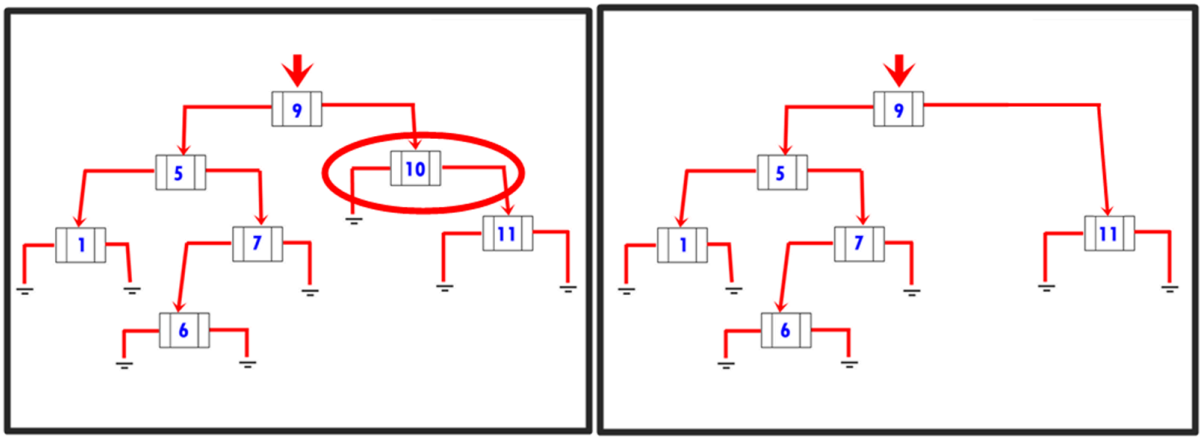


FIG 04: Ejemplo de cómo eliminar un nodo que solo tiene 1 hijo.

Caso 3
El nodo a eliminar tiene 2 hijos

Para este caso, lo que se tiene que hacer es un reemplazo, tenemos que buscar el mínimo valor por la derecha y luego hacer una búsqueda para eliminar ese valor por la derecha.

```
min_right_value = self.get_min_value(node.right)
node.data = min_right_value
node.right = self._del_node_by_data(node.right, min_right_value)
return node
```

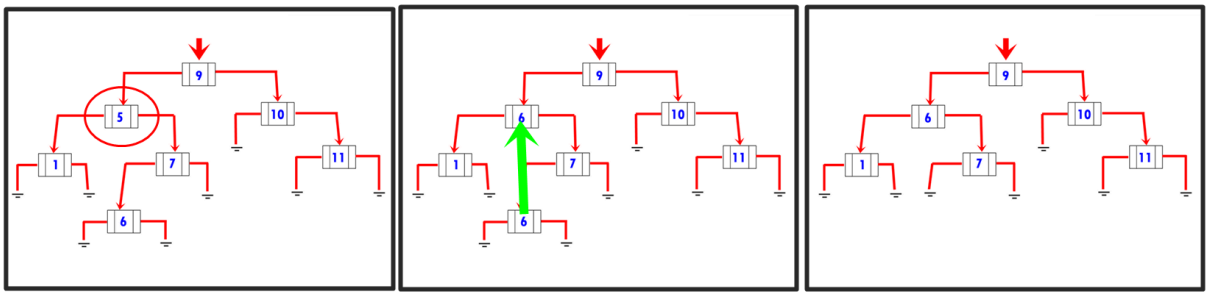


FIG 05: Ejemplo de cómo eliminar un nodo con 2 hijos.

Los 3 casos que acabamos de mencionar son aplicables a cada nodo del árbol incluyendo la raíz.

Recorridos

Cuando hablamos de recorridos estamos generando una estrategia para poder mostrar la información de ciertas maneras existen 3 formas de mostrar la información:

- In order: “En orden” muestra los elementos del árbol de manera ascendente (Osea se empieza por el valor más bajo hasta el más alto).

```
def viewInOrder(self):
    self._viewInOrder(self.root)
def _viewInOrder(self, pivot):
    if pivot != None:
        self._viewInOrder(pivot.left)
        print(pivot.data)
        self._viewInOrder(pivot.right)
```

- Pre order: muestra primero la raíz y luego se muestra lo que hay a la izquierda y por último se muestra la derecha.

```
def viewPreOrder(self):
    self._viewPreOrder(self.root)
def _viewPreOrder(self, pivot):
    if pivot != None:
        print(pivot.data)
        self._viewPreOrder(pivot.left)
        self._viewPreOrder(pivot.right)
```

- Post order: primero se recorre la izquierda, luego la derecha y por último se muestra la raíz.

```
def viewPostOrder(self):
    self._viewPostOrder(self.root)
def _viewPostOrder(self, pivot):
    if pivot != None:
        self._viewPostOrder(pivot.left)
        self._viewPostOrder(pivot.right)
        print(pivot.data)
```