UNIVERSIDAD AUTÓNOMA DE BENI
"JOSÉ BALLIVIAN"

ESTRUCTURA DE DATOS ll

ARBOLES BINARIOS

Universitario (a): Osias Peña Chaurara
Einar Rodrigo Valdivia Jare
Zamira Villar Castellanos
Carrera: INGENIERIA DE SISTEMAS

Docente: ing, Karina Lopez Machicao

Materia: ESTRUCTURA DE DATOS ll

Indice

Introducción 4

1 Implementación del árbol binario 5

1.1. Árbol binario: 5

1.2. Recorrido del árbol: 5

1.2.1. Pre-Orden: 5

1.2.2. In-Orden: 5

1.2.3. Post-Orden: 5

2. Creación del programa principal 6

2.1. Diseño de la estructura. 6

3.1. Declaración de la clase: 6

3.2. Método insertar: 7

3.2.1 Diagrama de flujo: 7

3.3. Método Recorrido en pre-orden: 8

3.4 Método recorrido in-Orden: 9

3.5 Metodo recorrido Post-Orden: 9

3.6 Metodo función Buscar: ..... **¡Error! Marcador no definido.**

3.7 Metodo función Encontrar-Minimo: 10

3.8 Metodo Eliminar: 11

3.9 Metodos Contar Elementos: 12

3.10 Metodos Altura: 12

3.11 Metodos Recorrido: 13

3.12 Metodos de Ordenar: ..... **¡Error! Marcador no definido.**

3.13 Método Encontrar Padre: 14

3.14 Método Insertar_Nodo: 14

3.15 Método Dibujar_nodo: ..... **¡Error! Marcador no definido.**

3.16 Método Dibujar_Arbol: ..... **¡Error! Marcador no definido.**

Introducción

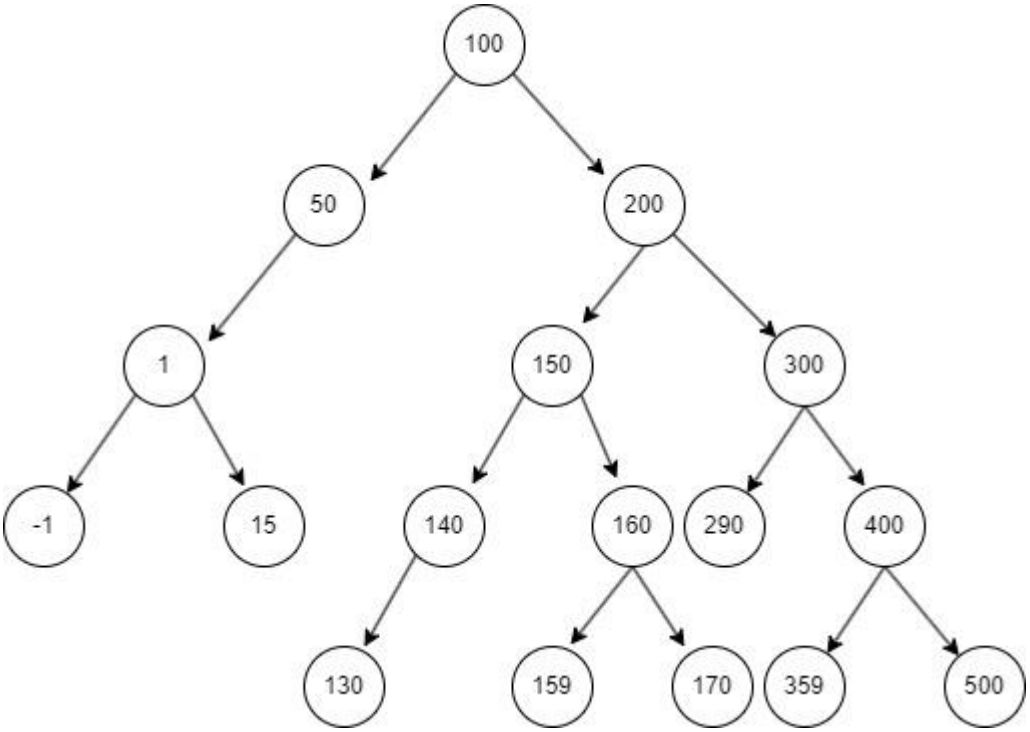
El presente proyecto tiene como objetivo la implementación de una estructura de datos conocida como árbol binario de búsqueda. El árbol binario es una estructura jerárquica en la cual cada nodo puede tener hasta dos hijos: uno izquierdo y uno derecho. Además, cumple con una propiedad fundamental que permite organizar los elementos de manera eficiente: los valores menores se ubican en el subárbol izquierdo y los valores mayores en el subárbol derecho.

El proyecto consiste en desarrollar una clase llamada "Árbol" que represente un árbol binario de búsqueda. Esta clase proporcionará una serie de métodos para insertar nodos, eliminar nodos, buscar valores, realizar recorridos del árbol, ordenar los elementos, contar elementos, calcular la altura del árbol, encontrar el padre de un nodo y dibujar el árbol. La finalidad de este proyecto es brindar una implementación versátil y completa de una estructura de datos poderosa como el árbol binario de búsqueda, que permite almacenar y organizar elementos de manera eficiente

Un árbol binario es una estructura de datos jerárquica en la que cada nodo puede tener hasta dos hijos, uno izquierdo y uno derecho. El primer nodo del árbol se denomina raíz, y a partir de ahí se ramifica en nodos secundarios. Cada nodo puede contener un valor o una información asociada. La propiedad clave de un árbol binario de búsqueda es que los valores menores se ubican en el subárbol izquierdo, mientras que los valores mayores se ubican en el subárbol derecho. Esto permite una rápida búsqueda y acceso a los elementos del árbol.

1 Implementación del árbol binario

1.1. Árbol binario:



1.2. Recorrido del árbol:

1.2.1. Pre-Orden:

100 50 1 -1 15 200 150 140 130 160 159 170 300 290 400 359 500

1.2.2. In-Orden:

-1 1 15 50 100 130 140 150 159 160 170 200 290 300 359 400 500

1.2.3. Post-Orden:

-1 15 1 50 130 140 159 170 160 150 290 359 500 400 300 200 100

Altura del árbol: 5

Cantidad de elementos en el árbol: 17

Lista ordenada:

[-1, 1, 15, 50, 100, 130, 140, 150, 159, 160, 170, 200, 290, 300, 359, 400, 500]

2. Creación del programa principal

2.1. Diseño de la estructura.

Diagrama de clase representa una visión simplificada de la implementación en la clase Árbol, proporcionando una idea general de la estructura y funcionalidad del código.

3. Código en Python:

3.1. Declaración de la clase:

```python
class Arbol:
    def __init__(self, dato):
        self.derecha = None
        self.izquierda = None
        self.info = dato
```

El programa en pieza con la declaración de la clase árbol. Con el método constructor.

Tiene los atributos “derecha” e “izquierda” que servirán para enganchar nodos a diferentes lados, También cuenta con el atributo “info” para guardar la llave del nodo.

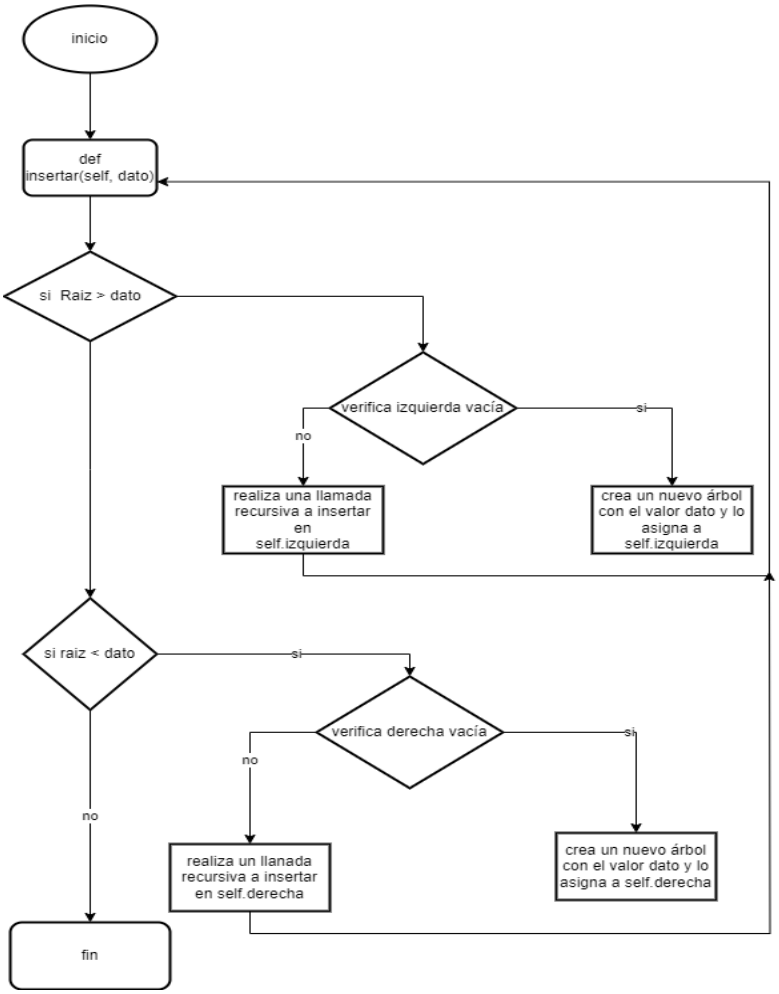
3.2. Método insertar:

```
def insertar(self, dato):
    if self.info > dato:
        if self.izquierda is None:
            self.izquierda = Arbol(dato)
        else:
            self.izquierda.insertar(dato)
    elif self.info < dato:
        if self.derecha is None:
            self.derecha = Arbol(dato)
        else:
            self.derecha.insertar(dato)
```

Para insertar los nodos, con los principios del árbol binario de búsqueda, se cuenta con el método "insertar".

Comienza comparando el valor de self.info con dato. Si self.info es mayor que dato, verifica si self.izquierda es None. Si es None, crea un nuevo árbol con el valor dato y lo asigna a self.izquierda. Si no es None, realiza una llamada recursiva a insertar en self.izquierda. Si self.info es menor que dato, sigue un proceso similar para self.derecha.

3.2.1 Diagrama de flujo:



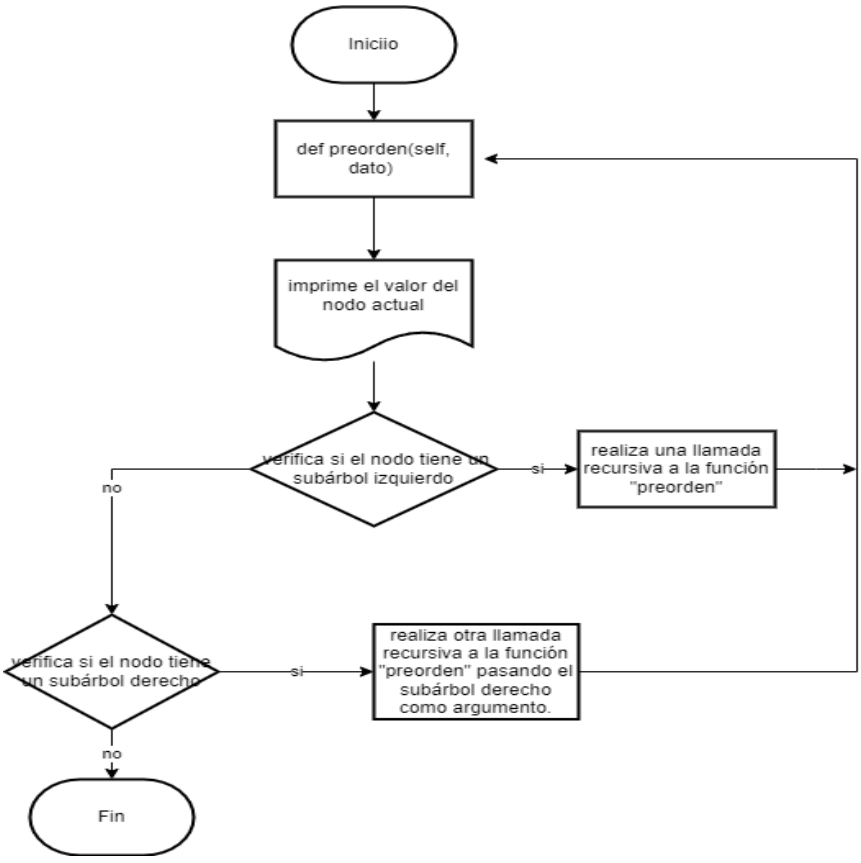
3.3. Método Recorrido en pre-orden:

primero la raíz, luego recursivamente se va por el subárbol izquierdo y subárbol derecho (Raiz-izquierdo-derecho)

```
def preorden(self, nodo):
    print(nodo.info,'')
    if nodo.izquierda !=None:
        self.preorden(nodo.izquierda)
    if nodo.derecha !=None:
        self.preorden(nodo.derecha)
```

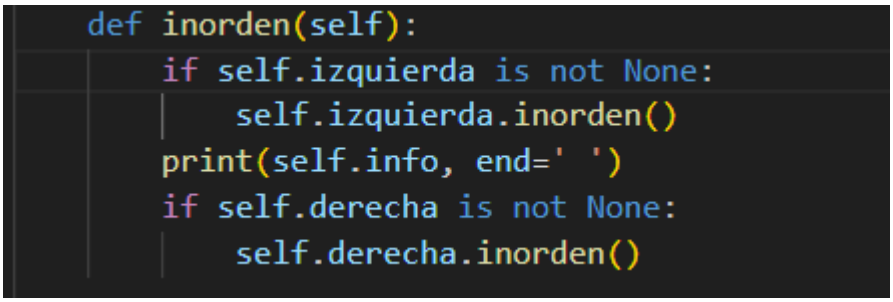
La lógica del código para realizar un recorrido en preorden en un árbol binario. Comienza imprimiendo el valor del nodo actual. Luego, verifica si el nodo tiene un subárbol izquierdo, y si es así, realiza una llamada recursiva a la función "preorden" pasando el subárbol izquierdo como argumento. Después, verifica si el nodo tiene un subárbol derecho, y si es así, realiza otra llamada recursiva a la función "preorden" pasando el subárbol derecho como argumento. El diagrama de flujo continúa hasta que se han visitado todos los nodos del árbol.

Diagrama de flujo:



3.4 Método recorrido in-Orden:

El algoritmo "inorden" se utiliza para recorrer un árbol binario y mostrar sus elementos en un orden específico.



```
def inorden(self):
    if self.izquierda is not None:
        self.izquierda.inorden()
    print(self.info, end=' ')
    if self.derecha is not None:
        self.derecha.inorden()
```

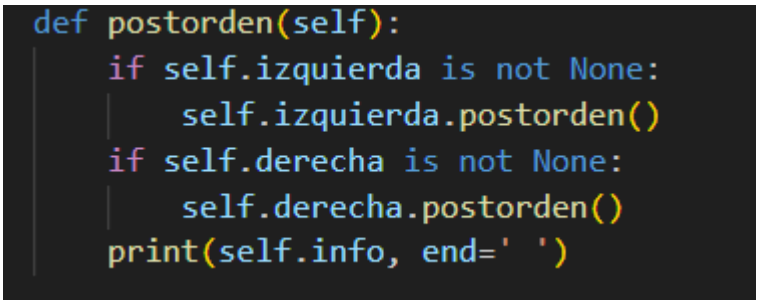
Verifica si el nodo izquierdo del nodo actual no es nulo. Si ese nodo existe, se llama recursivamente a la función "inorden" en ese nodo izquierdo. Esto asegura que se procesen todos los nodos del subárbol izquierdo antes de continuar.

Imprime el valor del nodo actual en la salida. El valor del nodo se obtiene a través de la variable "self.info". Además, se utiliza la opción "end=' '" al imprimir para que el valor del nodo se muestre seguido de un espacio en blanco en lugar de un salto de línea.

Verifica si el nodo derecho del nodo actual no es nulo. Si ese nodo existe, se llama recursivamente a la función "inorden" en ese nodo derecho. Esto garantiza que se procesen todos los nodos del subárbol derecho después de procesar el nodo actual.

3.5 Metodo recorrido Post-Orden:

El recorrido en postorden consiste en visitar los nodos en el siguiente orden: primero el nodo izquierdo, luego el nodo derecho y finalmente el nodo actual.



```
def postorden(self):
    if self.izquierda is not None:
        self.izquierda.postorden()
    if self.derecha is not None:
        self.derecha.postorden()
    print(self.info, end=' ')
```

Verifica si el nodo izquierdo del nodo actual no es nulo. Si ese nodo existe, se llama recursivamente a la función postorden en ese nodo izquierdo. Esto asegura que se procesen todos los nodos del subárbol izquierdo antes de continuar.

Verifica si el nodo derecho del nodo actual no es nulo. Si ese nodo existe, se llama recursivamente a la función postorden en ese nodo derecho. Esto garantiza que se procesen todos los nodos del subárbol derecho después de procesar el subárbol izquierdo.

Imprime el valor del nodo actual en la salida. El valor del nodo se obtiene a través de la variable self.info. Además, se utiliza la opción end=' ' al imprimir para que el valor del nodo se muestre seguido de un espacio en blanco en lugar de un salto de línea.

3.7 Metodo función Encontrar-Minimo:

La función encontrarMinimo encuentra el valor mínimo en un árbol binario de búsqueda.

```
def encontrarMinimo(self):
    if self.izquierda is None:
        return self.info
    return self.izquierda.encontrarMinimo()
```

Comprueba si el nodo actual no tiene un nodo izquierdo, es decir, self.izquierda is None. Si esto es cierto, significa que el nodo actual es el nodo más pequeño en el árbol y, por lo tanto, se devuelve su valor (almacenado en self.info).

Si el nodo actual tiene un nodo izquierdo, se llama recursivamente a la función encontrarMinimo() en el nodo izquierdo (self.izquierda.encontrarMinimo()). Esto se debe a que en un árbol binario de búsqueda, los valores más pequeños se encuentran en el subárbol izquierdo.

La recursión continúa hasta que se encuentre un nodo que no tenga un nodo izquierdo. En ese caso, se devuelve el valor del nodo actual como el valor mínimo.

3.8 Metodo Eliminar:

La función eliminar busca y elimina un nodo específico en un árbol binario de búsqueda.

```
def eliminar(self, dato):
    if self is None:
        return self

    if dato < self.info:
        if self.izquierda is not None:
            self.izquierda = self.izquierda.eliminar(dato)
    elif dato > self.info:
        if self.derecha is not None:
            self.derecha = self.derecha.eliminar(dato)
    else:
        if self.izquierda is None:
            temp = self.derecha
            self = None
            return temp
        elif self.derecha is None:
            temp = self.izquierda
            self = None
            return temp

        temp = self.derecha.encontrarMinimo()
        self.info = temp
        self.derecha = self.derecha.eliminar(temp)

    return self
```

- Si el nodo actual es None, se devuelve el nodo actual.
- Si el nodo actual no es None, se realiza la lógica de eliminación:
  - Si el valor buscado es menor, se llama recursivamente a eliminar en el subárbol izquierdo.
  - Si el valor buscado es mayor, se llama recursivamente a eliminar en el subárbol derecho.
  - Si el valor buscado es igual al valor del nodo actual:
    - Si no tiene hijo izquierdo, se devuelve el hijo derecho.
    - Si no tiene hijo derecho, se devuelve el hijo izquierdo.
    - Si tiene ambos hijos, se encuentra el sucesor inmediato, se reemplaza el valor del nodo actual con el valor del sucesor inmediato y se llama recursivamente a eliminar en el subárbol derecho para eliminar el sucesor inmediato.
- Se devuelve el nodo actual después de las modificaciones.

3.9 Metodos Contar Elementos:

La función contar_elementos se utiliza para contar el número de elementos en un árbol binario.

```
def contar_elementos(self):
    count = 1
    if self.izquierda is not None:
        count += self.izquierda.contar_elementos()
    if self.derecha is not None:
        count += self.derecha.contar_elementos()
    return count
```

Se inicializa una variable count con el valor de 1 para contar el nodo actual.

Se comprueba si el nodo actual tiene un hijo izquierdo (self.izquierda). Si es así, se llama recursivamente a contar_elementos en el hijo izquierdo y se suma el resultado a count. Esto se hace para contar los elementos en el subárbol izquierdo.

Se comprueba si el nodo actual tiene un hijo derecho (self.derecha). Si es así, se llama recursivamente a contar_elementos en el hijo derecho y se suma el resultado a count. Esto se hace para contar los elementos en el subárbol derecho.

Al final de la función, se devuelve count, que representa el número total de elementos en el árbol.

3.10 Metodos Altura:

La función altura se utiliza para calcular la altura de un árbol binario. La altura de un árbol es la longitud máxima de un camino desde la raíz hasta una hoja.

```
def altura(self):
    if self is None:
        return 0

    altura_izq = self.izquierda.altura() if self.izquierda else 0
    altura_der = self.derecha.altura() if self.derecha else 0

    return 1 + max(altura_izq, altura_der)
```

Se verifica si el nodo actual es None. Si es así, se devuelve 0, ya que un árbol vacío no tiene altura.

Si el nodo actual no es None, se procede con el cálculo de la altura:

a. Se calcula la altura del subárbol izquierdo llamando recursivamente a la función altura en el hijo izquierdo (self.izquierda). Si el hijo izquierdo no existe (self.izquierda es None), se considera una altura de 0.

b. Se calcula la altura del subárbol derecho llamando recursivamente a la función altura en el hijo derecho (self.derecha). Si el hijo derecho no existe (self.derecha es None), se considera una altura de 0.

Se devuelve 1 más el valor máximo entre la altura del subárbol izquierdo y la altura del subárbol derecho. Esto se hace para tomar en cuenta el nodo actual en el cálculo de la altura total del árbol.

3.11 Metodo Recorrido:

La función recorrido se utiliza para realizar un recorrido completo de un árbol binario. El recorrido del árbol se divide en tres tipos: preorden, inorden y postorden.

```
def recorrido(self):
    print("\nRecorrido del árbol:")
    print("\n\nPre-Orden:")
    self.preorden()
    print("\n\nIn-Orden:")
    self.inorden()
    print("\n\nPost-Orden:")
    self.postorden()
```

Se imprime un mensaje para indicar que se iniciará el recorrido del árbol.

Se imprime el recorrido preorden llamando a la función preorden. El recorrido preorden se realiza visitando primero el nodo raíz, luego el subárbol izquierdo y finalmente el subárbol derecho.

Se imprime el recorrido inorden llamando a la función inorden. El recorrido inorden se realiza visitando primero el subárbol izquierdo, luego el nodo raíz y finalmente el subárbol derecho.

Se imprime el recorrido postorden llamando a la función postorden. El recorrido postorden se realiza visitando primero el subárbol izquierdo, luego el subárbol derecho y finalmente el nodo raíz.

3.13 Método Encontrar Padre:

La función encontrar_padre se utiliza para buscar el padre de un nodo con un valor específico en un árbol binario de búsqueda.

```
def encontrar_padre(self, dato):
    if (self.izquierda is not None and self.izquierda.info == dato) or
        (self.derecha is not None and self.derecha.info == dato):
        return self.info
    if self.info > dato and self.izquierda is not None:
        return self.izquierda.encontrar_padre(dato)
    if self.info < dato and self.derecha is not None:
        return self.derecha.encontrar_padre(dato)
    return None
```

Se verifica si el nodo actual tiene un hijo izquierdo y si el valor del hijo izquierdo es igual al valor buscado (dato), o si tiene un hijo derecho y si el valor del hijo derecho es igual al valor buscado. En ese caso, se devuelve el valor del nodo actual, ya que es el padre del nodo con el valor buscado.

Si el valor del nodo actual (self.info) es mayor que el valor buscado (dato) y el nodo tiene un hijo izquierdo, se realiza una llamada recursiva a encontrar_padre en el hijo izquierdo (self.izquierda). Esto se hace para buscar el padre en el subárbol izquierdo, ya que el valor buscado es menor que el valor del nodo actual.

Si el valor del nodo actual es menor que el valor buscado y el nodo tiene un hijo derecho, se realiza una llamada recursiva a encontrar_padre en el hijo derecho (self.derecha). Esto se hace para buscar el padre en el subárbol derecho, ya que el valor buscado es mayor que el valor del nodo actual.

Si ninguna de las condiciones anteriores se cumple, significa que el valor buscado no se encuentra en el árbol y se devuelve None, indicando que no se encontró el padre.

3.14 Método Insertar_Nodo:

lógica de la función insertar puede variar dependiendo de cómo esté implementada en el código completo, pero en general, la inserción en un árbol binario de búsqueda sigue los siguientes pasos:

```
def insertar_nodo(self, dato):
    self.insertar(dato)
    print("Nodo insertado con éxito.")
```

Si el árbol está vacío, el nuevo nodo se convierte en la raíz del árbol.

Si el árbol no está vacío, se compara el valor del nuevo nodo con el valor del nodo actual.

Si el valor del nuevo nodo es menor que el valor del nodo actual, se verifica si el nodo actual tiene un hijo izquierdo. Si tiene un hijo izquierdo, se realiza una llamada recursiva a la función insertar en el hijo izquierdo. Si no tiene un hijo izquierdo, se inserta el nuevo nodo como el hijo izquierdo del nodo actual.

Si el valor del nuevo nodo es mayor que el valor del nodo actual, se verifica si el nodo actual tiene un hijo derecho. Si tiene un hijo derecho, se realiza una llamada recursiva a la función insertar en el hijo derecho. Si no tiene un hijo derecho, se inserta el nuevo nodo como el hijo derecho del nodo actual.

Si el valor del nuevo nodo es igual al valor del nodo actual, se puede manejar de diferentes maneras dependiendo de la implementación. Una opción podría ser ignorar la inserción del nodo duplicado, o se podría decidir insertarlo en un lugar específico, como en el subárbol izquierdo o derecho.

Método eliminar_nodo:

El método eliminar_nodo se utiliza para eliminar un nodo con un valor específico de un árbol binario de búsqueda. La lógica del método es la siguiente:

```
def eliminar_nodo(self, dato):
    if self.buscar(dato):
        self.eliminar(dato)
        print("Nodo eliminado con éxito.")
    else:
        print("El nodo no existe en el árbol.")
```

1. Se verifica si el nodo con el valor dato existe en el árbol utilizando el método buscar. Si el nodo se encuentra en el árbol, se procede a eliminarlo. Si no se encuentra, se imprime el mensaje "El nodo no existe en el árbol".
2. Si el nodo existe en el árbol, se llama al método eliminar para eliminar el nodo con el valor dato. La implementación exacta de este método puede variar dependiendo de cómo esté construido el árbol binario de búsqueda.
3. Después de eliminar el nodo, se imprime el mensaje "Nodo eliminado con éxito" para indicar que la operación de eliminación se realizó correctamente.