DOCUMENTO TÉCNICO ÁRBOLES

Integrantes:

- **Diego Cárdenas**
- **Felipe Calvache**
- **Zayra Gutiérrez**

Especificación:

Se requiere realizar las operaciones de actualización y eliminación en una clase Binary Tree, siguiendo las reglas de BST (sin balance), incorporando el atributo de factor de balance AVL a la clase y ajustando las operaciones existentes para mantener su consistencia. Además, se debe optimizar el CRUD considerando el factor de balance AVL por nodo, con el objetivo de lograr una transición suave a la versión balanceada BST. Finalmente, se debe generar un código Main y un informe que incluya un ejemplo de caso de uso para demostrar la corrección de las operaciones.

Diseño:

Estrategia

Antes de implementar las operaciones de Update y Delete en una lista enlazada, se necesita definir la función de Delete, la cual consiste en eliminar el nodo que se quiere actualizar y reemplazarlo con el nuevo valor. Para la función de Delete, se deben considerar diferentes escenarios, incluyendo la búsqueda del valor que se quiere eliminar, la identificación del sucesor y la actualización del árbol. Una vez definida la función de Delete, se puede implementar la función de Update.

Para agregar el factor de balance AVL, se deben considerar diferentes casos, incluyendo asignar el valor 0 si el nodo no tiene hijos, analizar la altura del árbol por la izquierda y por la derecha de manera recursiva y restar las alturas para obtener el factor de balance. En las funciones CRUD, se debe implementar el balance AVL y asegurarse de que el factor de balance se mantenga menor o igual a 1. En caso de que no se cumpla esta condición, se deben realizar rotaciones para mantener el balance del árbol.

Codigo

```python
def delete(self, value):
    wrench = self.search(value)
    left ,right, root = wrench.getLeft(), wrench.getRight(), wrench.getRoot()
    if wrench is not None:
        has_two = wrench.getLeft() is not None and wrench.getRight() is not None
        if not has_two:
            parent = wrench.getParent()
            wrench.clear()
            if left is not None:
                parent.setLeft(left)
            else:
                parent.setRight(right)
        else:
            left_right_most = left.maximum()
            wrench.setRoot(left_right_most.getRoot())
            left_right_most.setRoot(root)
            parent = left_right_most.getParent()
            parent.setRight(None)
            left_right_most.clear()
            parent.setHeight()
            parent.setAVL()
    else:
        raise Exception("El valor no se encunetra")
    self.setHeight()
    self.setAVL()
```

```python
def update(self,old_key,new_key):
    self.delete(old_key)
    self.insert(new_key)
```

```python
def setHeight(self):
    if self.isEmpty():
        self.height = 0
    else:
        left, right = self.left, self.right
        if left:
            left.setHeight()
        if right:
            right.setHeight()
        leftHeight , rightHeight = 0 if left is None else left.getHeight(), 0 if right is None else right.getHeiht()
        self.height = 1 + max(leftHeight,rightHeight)
```

```python
def setAVL(self):
    if self.isEmpty():
        self.avl = 0
    else:
        left, right = self.left, self.right
        if left:
            left.setAVL()
        if right:
            right.setAVL()
        leftHeight, rightHeight = 0 if left is None else left.getHeight(), 0 if right is None else right.getHeiht()
        self.avl = rightHeight - leftHeight
```

```python
def balance(self, lastValue):
    root,left,right,avl = self.getRoot, self.left, self.right,self.avl
    if avl > 1 and right.getRoot() <= lastValue:
        self.rotateLeft()
    if avl > 1 and right.getRoot() > lastValue:
        self.rotateRightLeft()
    if avl < -1 and left.getRoot() > lastValue:
        self.rotateRight()
    if avl < -1 and left.getRoot() <= lastValue:
        self.rotateLeftRight()
    self.setHeight()
    self.setAVL()
```

```
def rotateLeft(self):
    root, left, right, avl, rl = self.getRoot(), self.left, self.rig
ht, self.avl, self.right.getLeft()
    self.setRoot(right.getRoot())
    self.setRight(right.getRight())
    self.setLeft(BinaryTree(root))
    self.left.setRight(rl)
    self.left.setLeft(left)
    self.setHeight()
    self.setAVL()

def rotateRight(self):
    root, left, right, avl, rl = self.getRoot(), self.left, self.rig
ht, self.avl, self.left.getRight()
    self.setRoot(left.getRoot())
    self.setRight(left.getRight())
    self.setLeft(BinaryTree(root))
    self.Right.setLeft(rl)
    self.Right.setRight(right)
    self.setHeight()
    self.setAVL()
```

```
def rotateLeftRight(self):
    root, left, right, avl = self.getRoot(), self.left, self.right, self.
avl
    lRoot, lvr, lvl, ll = left.getRoot(), left.right.right, left.right.le
ft, left.left
    left.setRoot(left.right.getRoot())
    left.setLeft(BinaryTree(lRoot))
    left.setRight(lvr)
    left.left.setLeft(ll)
    left.left.setRight(lvl)
    self.rotateRight()

def rotateRightLeft(self):
    root, left, right, avl = self.getRoot(), self.left, self.right, self.
avl
    rRoot, rr, lvr, lvl = right.getRoot(), right.right, right.left.right,
right.left.left
    right.setRoot(right.left.getRoot())
    right.setRight(BinaryTree(rRoot))
    right.setLeft(lvl)
    right.right.setLeft(lvr)
    right.right.setRight(rr)
    self.rotateLeft()
```

Casos de prueba.

Entrada	Justificación	Salida
(['3', '1', '0', '2', '7', '5', '4', '6','8','9'], ['0', '1', '2', '3', '4', '5', '6', '7','8', '9']) tree.update(3, 29)	Probar el funcionamiento de la función Update	['2', '1', '0', '7', '4', '9', '8', '29'], ['0', '1', '2', '4', '7', '8', '9', '29']
(['3', '1', '0', '2', '7', '5', '4', '6', '8','9'], ['0', '1', '2', '3', '4', '5', '6', '7','8', '9']) tree.delete(10)	Verificar que la búsqueda funciona correctamente al tratar de eliminar un valor que no existe en el árbol.	"No se encuentra el valor"