Parte 1

Importante: Los ejercicios de esta primera parte tienen como objetivo codificar las diferentes funciones básicas necesarias para la implementar un árbol AVL.

A partir de estructuras definidas como :

```
class AVLTree:
     root = None

class AVLNode:
        parent = None
        leftnode = None
        rightnode = None
        key = None
        value = None
        bf = None
```

Copiar y adaptar todas las operaciones del **binarytree.py** (i.e insert(), delete(), search(),etc) al nuevo módulo **avltree.py.** Notar que estos luego deberán ser implementados para cumplir que la propiedad de un árbol AVL

Ejercicio 1

Crear un modulo de nombre **avltree.py** Implementar las siguientes funciones:

rotateLeft(Tree,avlnode)

Descripción: Implementa la operación rotación a la izquierda
Entrada: Un Tree junto a un AVLnode sobre el cual se va a operar la rotación a la izquierda
Salida: retorna la nueva raíz

```python
def rotateLeft(B,node):
    currentNode = node.rightnode
    parent = node.parent

    node.rightnode = currentNode.leftnode
    currentNode.leftnode = node
    node.parent = currentNode

    if parent == None:
        B.root = currentNode
    else:
        currentNode.parent = parent
        if parent.leftnode == node:
            parent.leftnode = currentNode
        else:
            parent.rightnode = currentNode

    return B
```

rotateRight(Tree,avlnode)

Descripción: Implementa la operación rotación a la derecha
Entrada: Un Tree junto a un AVLnode sobre el cual se va a operar la rotación a la derecha
Salida: retorna la nueva raíz

```python
def rotateRight(B, node):
    currentNode = node.leftnode
    parent = node.parent

    node.leftnode = currentNode.rightnode
    currentNode.rightnode = node
    node.parent = currentNode

    if parent == None:
        B.root = currentNode
    else:
        currentNode.parent = parent
        if parent.leftnode == node:
            parent.leftnode = currentNode
        else:
            parent.rightnode = currentNode

    return B
```

Ejercicio 2

Implementar una función recursiva que calcule el elemento `balanceFactor` de cada subárbol siguiendo la siguiente especificación:

calculateBalance(AVLTree)

Descripción: Calcula el factor de balanceo de un árbol binario de búsqueda.
Entrada: El árbol AVL sobre el cual se quiere operar.
Salida: El árbol AVL con el valor de balanceFactor para cada subarbol

```python
def calculateBalance(B):
    calculateBalanceR(B.root)


def calculateBalanceR(node):
    if node == None:
        return
    else:
        leftHeight = heigthTree(node.leftnode) - 1
        rightHeight = heigthTree(node.rightnode) - 1
        node.bf = leftHeight - rightHeight

        if node.leftnode != None:
            calculateBalanceR(node.leftnode)
        if node.rightnode != None:
            calculateBalanceR(node.rightnode)
```

Ejercicio 3

Implementar una funcion en el modulo avltree.py de acuerdo a las siguientes especifcaciones:

reBalance(AVLTree)

Descripción: balancea un árbol binario de búsqueda. Para esto se deberá primero calcular el **balanceFactor** del árbol y luego en función de esto aplicar la estrategia de rotación que corresponda.

Entrada: El árbol binario de tipo AVL sobre el cual se quiere operar.

Salida: Un árbol binario de búsqueda balanceado. Es decir luego de esta operación se cumple que la altura (h) de su subárbol derecho e izquierdo difieren a lo sumo en una unidad.

```python
def reBalance(B):
    calculateBalance(B)
    rebalanceR(B,B.root)


def rebalanceR(B,node):
    if node.bf > 1 or node.bf < -1:
        if node.bf > 1 and node.leftnode.bf >= 0:
            rotateRight(B, node)
        else:
            if node.bf > 1 and node.leftnode.bf < 0:
                rotateRight(B,node)
                rotateLeft(B, node)

        if node.bf < -1 and node.rightnode.bf <= 0:
            rotateLeft(B, node)
        else:
            if node.bf < -1 and node.rightnode.bf > 0:
                rotateLeft(B, node)
                rotateRight(B, node)
    else:
        if node.leftnode != None:
            return rebalanceR(node.leftnode)
        if node.rightnode != None:
            return rebalanceR(node.rightnode)

    return B
```

Ejercicio 4:

Implementar la operación **`insert()`** en el módulo **`avltree.py`** garantizando que el árbol binario resultante sea un árbol AVL.

```python
def insert(B, element, key):
    node = AVLNode()
    node.key = key
    node.value = element
    node.bf = 0
    if B.root == None:
        B.root = node
        return key
    newNode = insertR(B.root, node)
    update_bf(newNode.parent)
    nodo = searchBalance(newNode.parent)
    if nodo != None:
        rebalanceR(B, nodo)
    return newNode.key


def insertR(currentNode, node):
    a = None
    if currentNode.key > node.key:
        if currentNode.leftnode != None:
            a = insertR(currentNode.leftnode, node)
        else:
            node.parent = currentNode
            currentNode.leftnode = node
            return node
    else:
        if currentNode.rightnode != None:
            a = insertR(currentNode.rightnode, node)
        else:
            node.parent = currentNode
            currentNode.rightnode = node
            return node
    return a
```

```python
def update_bf(node):
    if node == None:
        return
    else:
        leftHeight = heigthTree(node.leftnode) - 1
        rightHeight = heigthTree(node.rightnode) - 1
        node.bf = leftHeight - rightHeight
        update_bf(node.parent)

def heigthTree(currentNode):
    return 0 if currentNode == None else 1 + max(heigthTree(currentNode.leftnode), heigthTree(currentNode.rightnode))

def max(heigthLeftNode, heigthRightNode):
    return heigthLeftNode if heigthLeftNode > heigthRightNode else heigthRightNode


def searchBalance(node):
    if node == None:
        return
    else:
        if node.bf < -1 or node.bf > 1:
            return node
        else:
            return searchBalance(node.parent)
```

Ejercicio 5:

Implementar la operación **`delete()`** en el módulo **`avltree.py`** garantizando que el árbol binario resultante sea un árbol AVL.

```python
def delete(B,element):
    if B.root.value == element:
        if B.root.leftnode.rightnode != None:
            nodeReplace = B.root.leftnode.rightnode
            nodeReplace.leftnode = B.root.leftnode
            nodeReplace.rightnode = B.root.rightnode
            B.root = nodeReplace
        else:
            nodeReplace = B.root.rightnode.leftnode
            nodeReplace.leftnode = B.root.leftnode
            nodeReplace.rightnode = B.root.rightnode
            B.root = nodeReplace
    else:
        parent = deleteR(B.root,element)

    calculateBalance(B)
    nodo = searchBalance(parent)

    if nodo != None:
        rebalanceR(B, nodo)
```

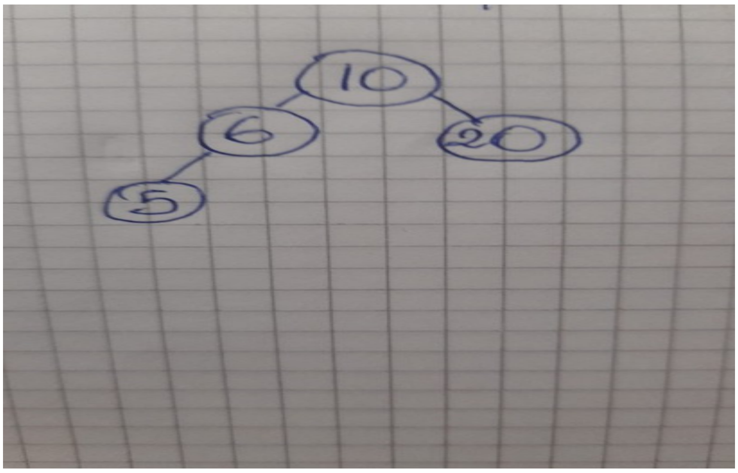
```python
def deleteR(currentNode, element):
    a = None
    if currentNode.value == element:
        key = currentNode.key
        if currentNode.leftnode == None and currentNode.rightnode == None:
            nodo = currentNode
            currentNode = currentNode.parent
            nodo.parent = None
            if currentNode.leftnode.key == key:
                currentNode.leftnode = None
            else:
                currentNode.rightnode = None
            return currentNode
        elif (currentNode.leftnode != None and currentNode.rightnode == None) or (currentNode.rightnode != None and currentNode.leftnode == None):
            nodo = currentNode.leftnode if currentNode.leftnode else currentNode.rightnode
            currentNode = currentNode.parent
            if currentNode.leftnode.key == key:
                nodo.parent = currentNode
                currentNode.leftnode = nodo
            else:
                nodo.parent = currentNode
                currentNode.rightnode = nodo
            return currentNode
        else:
            nodoleft = currentNode.leftnode
            nodoright = currentNode.rightnode
            nodo = currentNode
            currentNode = currentNode.parent
            if nodoleft.leftnode == None and nodoleft.rightnode == None and nodoright.leftnode == None and nodoright.rightnode == None:
                nodoleft.parent = currentNode
                if currentNode.leftnode.key == key:
                    currentNode.leftnode = nodoleft
                else:
                    currentNode.rightnode = nodoleft
                nodoleft.rightnode = nodoright
            else:
                nodo_replace = nodoleft.rightnode if nodoleft.rightnode else nodoright.leftnode
                nodo_replace.parent = currentNode
                if currentNode.leftnode.key == key:
                    currentNode.leftnode = nodo_replace
                else:
                    currentNode.rightnode = nodo_replace
            return currentNode

    if currentNode.leftnode != None:
        a = deleteR(currentNode.leftnode, element)
    if currentNode.rightnode != None and a == None:
        a = deleteR(currentNode.rightnode, element)
    return a
```

Parte 2

Ejercicio 6:

1. Responder V o F y justificar su respuesta:
    a. _F__ En un AVL el penúltimo nivel tiene que estar completo

    contraejemplo:

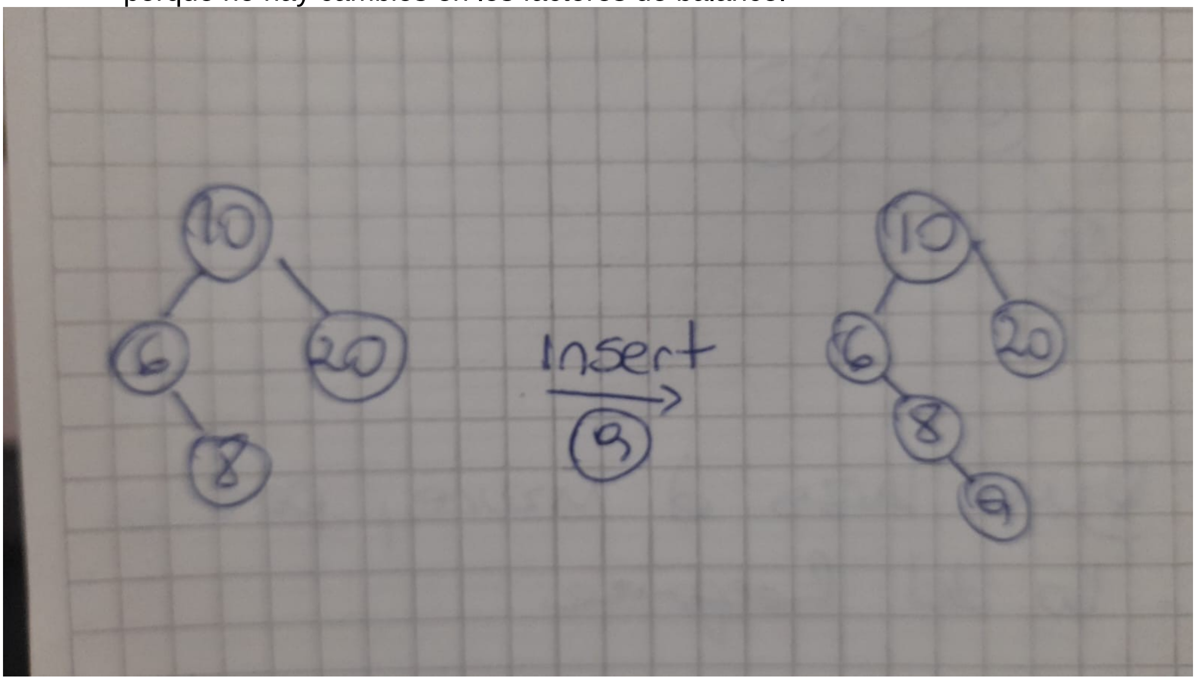


b. _V__ Un AVL donde todos los nodos tengan factor de balance 0 es completo

El balanceFactor es = h(leftnode) – h(rightnode)
Un arbol es un arbol que tiene en todo sus niveles 2 hijos

Si digamos que tenemos un arbol con todos sus nodos con balance factor = 0 y nodo en el penultimo nivel, que tiene un solo hijo.
Si calculamos su balance factor es -1 o 1. Entonces se cumple que no es completo y no es igual a cero su balance factor

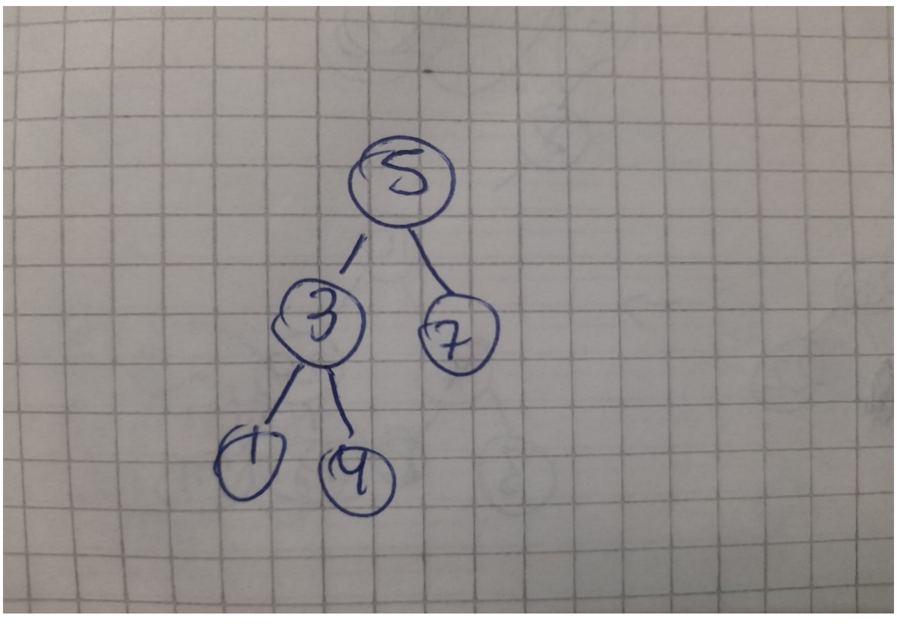
c. _F__ En la inserción en un AVL, si al actualizarle el factor de balance al padre del nodo insertado éste no se desbalanceó, entonces no hay que seguir verificando hacia arriba porque no hay cambios en los factores de balance.



Si calculamos el bf de 8, es -1, pero el balance factor de 6 es -2

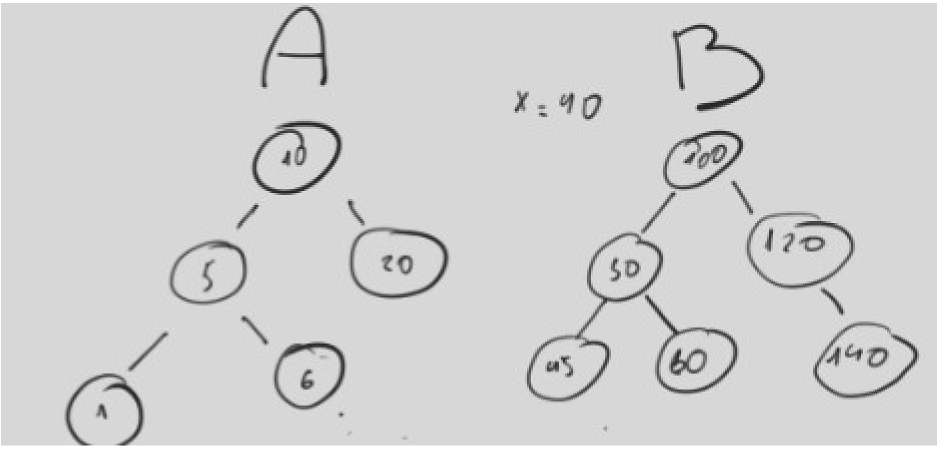
d. _F_ En todo *AVL* existe al menos un nodo con factor de balance 0.

sin tener en cuenta las hojas:



Ejercicio 7:

Sean *A* y *B* dos *AVL* de *m* y *n* nodos respectivamente y sea *x* un key cualquiera de forma tal que para todo key $a \in A$ y para todo key $b \in B$ se cumple que $a < x < b$. Plantear un algoritmo $O(\log n + \log m)$ que devuelva un *AVL* que contenga los key de *A*, el key *x* y los key de *B*.



Bueno existen 3 casos:

1. El caso en el que A y B tenga la misma altura

Si ambos arboles tiene la misma altura, log(m) = log(n). Dada la definicion de arriba donde a < x < b

x seria la raiz, A el subarbol izquierdo de x, B seria el subarbol derecho

2. El caso en el que A sea mas grande que B

Buscamos en A un subarbol del tamaño de B, que parta del subarbol derechi de A.

Donde encontremos ese subarbol, colocamos X. Como subarbol izquierdo de X ponemos el subarbol que buscamos. Y B como subarbol derecho de X

3. El caso en el que B sea mas grande que A

Buscamos en B un subarbol del tamaño de A, que parta del subarbol derechi de B.

Donde encontremos ese subarbol, colocamos X. Como subarbol izquierdo de X ponemos el subarbol que buscamos. Y A como subarbol derecho de X

Ejercicio 8:

Considere una rama truncada en un AVL como un camino simple desde la raíz hacia un nodo que tenga una referencia None (que le falte algún hijo). Demuestre que la mínima longitud (cantidad de aristas) que puede tener una rama truncada en un AVL de altura h es h/2 (tomando la parte entera por abajo).

Cualquier camino desde la raíz hasta un nodo que no esté completo puede ser una rama truncada según la definición del ejercicio. Dicho nodo puede no ser necesariamente un nodo hoja.

Dada la definicion de arbol avl y de rama truncada.

Supongamos que hay un camino truncado, de altura h. Si el nodo se encuentra del lado del subarbol izquierdo, suponemos que tiene una altura de h+1 el subarbol izquierdo, analogamente para el subarbol derecho

Podemos decir que la longitud minima del camino es igual al nivel del nodo faltante. entonces supongamos que esta en el nivel k.

Como la altura de un nodo es igual a la diferencia de alturas de sus subarboles. Podemos usar esto para encontrar la altura del nodo faltante

Si la altura es h entonces h-2k es igual a la altura del nodo faltante. Pero como buscamos la longitud minima, debemos encontrar el k.

Cuando la altura del nodo sea igual a 0, estaremos en un nodo hoja. igualamos la altura del nodo faltante a cero (porque si tiene altura cero, es un nodo hoja y no hay rama truncada) y

Entonces h-2k=0 -> h=2k -> h/2=k

Parte 3

Ejercicios Opcionales

1. Si n es la cantidad de nodos en un árbol AVL, implemente la operación **`height()`** en el módulo **`avltree.py`** que determine su altura en O(log n). Justifique el por qué de dicho orden.
2. Considere una modificación en el módulo **`avltree.py`** donde a cada nodo se le ha agregado el campo **`count`** que almacena el número de nodos que hay en el subárbol en el que él es raíz. Programe un algoritmo *O*(log *n*) que determine la cantidad de nodos en el árbol cuyo valor del key se encuentra en un intervalo [*a*, *b*] dado como parámetro. Explique brevemente

por qué el algoritmo programado por usted tiene dicho orden.

A tener en cuenta:

1. **Usen lápiz y papel primero**
2. ~~**No se puede utilizar otra Biblioteca mas alla de algo1.py y las bibliotecas desarrolladas durante Algoritmos y Estructuras de Datos I.**~~

Bibliografia:

[1] Guido Tagliavini Ponce, Balanceo de arboles y arboles AVL (Universidad de Buenos Aires)
[2] Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python.