Alvarez Lucía, TP Árboles AVL

# 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
# Salida: retorna la nueva raiz
def rotateLeft(avl, avlnode):
  newparent=avlnode.rightnode #avlnode es la raíz
  if avlnode.parent.leftnode==avlnode: #me fijo si es hijo izquierdo o derecho
    left=True
  else:
    left=False
  if avlnode.parent==None: #avlnode es la raiz
    #cambio referencias y queda como nuevo padre el hijo izq y como raiz del avl
    avlnode.parent=newparent
    avlnode.rightnode=newparent.leftnode #le asigno como hijo izq el ex hijo der de la nueva raiz
    avlnode.rightnode.parent=avlnode
    newparent.parent=None #elimino el padre de la nueva raiz
    newparent.leftnode=avlnode
  else: #no es la raíz
    exparent=avlnode.parent #gurdamos referencias
    if newparent.leftnode!=None: #cambiamos hijos si tienen
      avlnode.rightnode=newparent.leftnode
      avlnode.rightnode.parent=avlnode
    else: #sino solo eliminamos el viejo hijo
      avlnode.rightnode=None
    avlnode.parent=newparent #rotamos
    newparent.leftnode=avlnode
    newparent.parent=exparent  #actualizamos padre del nuevo nodo rotado
    if left:
      exparent.leftnode=newparent
    else:
      exparent.rightnode=newparent
  avlnode=reHeightBF(avlnode) #actualizar alturas y bf
  newparent=reHeightBF(newparent)
  return
```

**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**

```
def rotateRight(avl, avlnode):
    newparent=avlnode.leftnode #guardo el nuevo padre (hijo izq)
    if avlnode.parent.leftnode==avlnode:  #me fijo si es hijo izquierdo o derecho
        left=True
    else:
        left=False
    if avlnode.parent==None: #avlnode es la raiz
        #cambio referencias y queda como nuevo padre el hijo izq y como raiz del avl
        avlnode.parent=newparent
        avlnode.leftnode=newparent.rightnode #le asigno como hijo izq el ex hijo der de la nueva raiz
        avlnode.leftnode.parent=avlnode
        newparent.parent=None #elimino el padre de la nueva raiz
        newparent.rightnode=avlnode
    else: #no es la raíz
        exparent=avlnode.parent #gurdamos referencias
        if newparent.rightnode!=None: #cambiamos hijos si tienen
            avlnode.leftnode=newparent.rightnode
            avlnode.leftnode.parent=avlnode
        else: #sino solo eliminamos el viejo hijo
            avlnode.leftnode=None
        #rotamos
        avlnode.parent=newparent
        newparent.rightnode=avlnode
        newparent.parent=exparent #actualizamos padre del nuevo nodo rotado
        if left:  #colocamos al nuevo nodo al lado correspondiente
            exparent.leftnode=newparent
        else:
            exparent.rightnode=newparent
    avlnode=reHeightBF(avlnode)  #actualizar alturas y bf
    newparent=reHeightBF(newparent)
    return
```

```
#recalcula la altura y el balance factor de un nodo fijandose en las alturas de los hijos
def reHeightBF(node):
    if node.leftnode!=None: #tengo hijo a la izq
        if node.rightnode!=None: #tengo hijo a la derecha
            node.bf=node.leftnode.height - node.rightnode.height
            node.height=max(node.leftnode.height, node.rightnode.height) +1
        else: #no tengo hijo a la derecha
            node.bf=node.leftnode.height
            node.height=node.leftnode.height +1
    else: #no tengo hijo izq
        if node.rightnode!=None: #tengo hijo derecho
            node.bf=- node.rightnode.height
            node.height=node.rightnode.height +1
        else: #no tengo hijo a la derecha
            node.bf=0
            node.height=1
    return node
```

# 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 subárbol

```python
#calcula bf y altura desde el nodo recibido hasta la raiz, no todo el arbol
def calculateBalanceRecur(node):
  if node.leftnode!=None:
    #si tengo hijo a la izq sin calcular su altura, la calculo
    if node.leftnode.height==None:
      return calculateBalanceRecur(node.leftnode)
    else:
      #si ya está calculada y no tengo un hijo a la derecha, calculo bf
      if node.rightnode==None:
        node.height=node.leftnode.height + 1
        node.bf= node.leftnode.height
        return node
      elif node.rightnode.height==None:
        #si tengo hijo a la derecha pero su altura no está, la calculo
        return calculateBalanceRecur(node.rightnode)
      else:
        #tengo los dos hijos con sus alturas, calculo bf y altura del nodo actual
        node.height= max(node.leftnode.height,node.rightnode.height) +1
        node.bf= node.leftnode.height - node.rightnode.height
        if node.parent==None:
          return node
        else:
          return calculateBalanceRecur(node.parent) #calculo altura y bf del padre
```

```
elif node.rightnode!=None:
  #tengo hijo a la derecha sin altura, la calculamos
  if node.rightnode.height==None:
    return calculateBalanceRecur(node.rightnode)
  else:
    #no tengo hijo a la izquierda y el derecho tiene altura, calcular bf
    if node.leftnode==None:
      node.height=node.rightnode.height+1
      node.bf = -node.rightnode.height
      if node.parent==None:
        return node
      else:
        return calculateBalanceRecur(node.parent) #calculo altura y bf del padre
    elif node.leftnode.height==None:
      #tengo hijo izq sin altura, calcular altura
      return calculateBalanceRecur(node.leftnode)
    else:
      #tengo dos hijos y alturas, calculo bf y altura
      node.height=max(node.leftnode.height,node.rightnode.height) +1
      node.bf = node.leftnode.height - node.rightnode.height
      return node
```

```
else:
  #estoy en una hoja, bf y altura son 0
  node.bf=0
  node.height=1 #usamos altura uno como si no tuviera hijos asi se puede calcular bf sin problema
return calculateBalanceRecur(node.parent) #calculo altura y bf del padre
```

# 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
#recibe el avl y el nodo a balancear, balancea desde el nodo hasta la raiz
def reBalanceRecu(avl,node):
  if node==None:
    return avl, node
  reHeightBF(node) #recalcula la altura y bf del nodo para contemplar casos recursivos
  if node.bf==0 or node.bf==1 or node.bf==-1: #revisa si el nodo está balanceado
    if node.parent==None:
      return node
    else:
      return reBalanceRecu(avl,node.parent) #si está balanceado balancea el padre
  else:
    if node.bf==-2: #nodo desbalanceado
      if node.rightnode.bf==1: #caso especial
        rotateRight(avl,node.rightnode) #roto primero a la derecha el hijo
      rotateLeft(avl,node)
    elif node.bf==2:
      if node.leftnode.bf==-1: #caso especial
        rotateLeft(avl,node.leftnode) #roto primero a la izquierda el hijo
      rotateRight(avl,node)
  if node.parent==None:
    return node #si es la raiz devuelvo el nodo
  else:
    return reBalanceRecu(avl,node.parent) #recalculo el bf y altura del padre
```

# Ejercicio 4:

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

```python
def insertT(AVL,element,key):
  if key==None: #caso donde no se pasa bien la key
    return None
  else: #creo el nodo con todos sus atibutos
    newNode =AVLNode()
    newNode.key = key
    newNode.value = element
    newNode.bf=0
    newNode.height=1
    if AVL.root==None: #si no existe el arbol lo creo con el nodo como raiz
      AVL.root=newNode
      return key
    else: #inserto el elemento si ya habia raiz
      insertElement(newNode, AVL.root)
      reBalanceRecu(AVL,newNode) #rebalanceo desde el padre del nodo insertado hasta la raiz
      return key
```

```python
#Recibe el nuevo nodo y el actual para recursivamente buscar el lugar correspondiente según la key del nuevo nodo
def insertElement(newNode, currentNode):
  if newNode.key > currentNode.key: #comparo keys para saber si va a la izq o derecha
    if currentNode.rightnode == None:
      currentNode.rightnode= newNode
      newNode.parent=currentNode
    else:
      insertElement(newNode, currentNode.rightnode) #si no hay espacio continuo comparando
  else:
    if currentNode.leftnode == None:
      currentNode.leftnode = newNode
      newNode.parent=currentNode
    else:
      insertElement(newNode, currentNode.leftnode)
  return
```

# Ejercicio 5:

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

```python
#Devuelve None si el elemento a eliminar no se encuentra.
def deleteT(AVL, element):
    key=searchT(AVL,element) #busca la key del elemento
    #si no existe el elemento retorna none
    if key==None:
        return None
    else:
        #uso accessReturnNode para recibir el nodo que hay que eliminar directamente y no tener que buscarlo
        node=accessReturnNode(AVL.root, key)
        parent=node.parent
        #el nodo no tiene hijos
        if node.rightnode==None and node.leftnode == None:
            if node.key>parent.key:
                parent.rightnode=None
                reBalanceRecu(AVL, parent) #rebalanceo el padre del nodo
                return key
            elif node.key<parent.key:
                parent.leftnode=None
                reBalanceRecu(AVL, parent)  #rebalanceo el padre del nodo
                return key
            else:
                return None
```

```python
        elif node.rightnode!=None:
            #tiene un solo hijo (el de la derecha)
            if node.leftnode==None:
                if node.key<parent.key:
                    current=node.rightnode
                    parent.leftnode=current
                    current.parent=parent
                    reBalanceRecu(AVL, parent)  #rebalanceo el padre del nodo
                    return key
                else:
                    current=node.rightnode
                    parent.rightnode=current
                    current.parent=parent
                    reBalanceRecu(AVL, parent) #rebalanceo el padre del nodo
                    return key
```

```
    else:
    #tiene dos hijos
    #menor de los mayores
        current=node.rightnode
        while current.leftnode!=None:
            current=current.leftnode
        exleft=current.leftnode
        current.leftnode=node.leftnode
        if current.rightnode!=None:
            rightnode=current.rightnode
            if current.rightnode.value!=node.rightnode.value :
                current.rightnode=node.rightnode
                rightnode.parent=current.rightnode
                current.rightnode.leftnode=rightnode
            else:
                current.rightnode=None
        else:
            current.rightnode=node.rightnode
            current.rightnode.leftnode=exleft
            node.rightnode.parent=current
        if key<parent.key:
            parent.leftnode=current
        else:
            parent.rightnode=current
        current.parent=parent
        reBalanceRecu(AVL, current.rightnode)  #rebalanceo el hijo derecho ya que le cambia el bf
        return key
else:
```

```
else:
#right no existe y left si
    if node.key<parent.key:
        current=node.leftnode
        parent.leftnode=current
        current.parent=parent
        reBalanceRecu(AVL, parent)
        return key
    else:
        current=node.leftnode
        parent.rightnode = current
        current.parent=parent
        reBalanceRecu(AVL, parent)
        return key
```

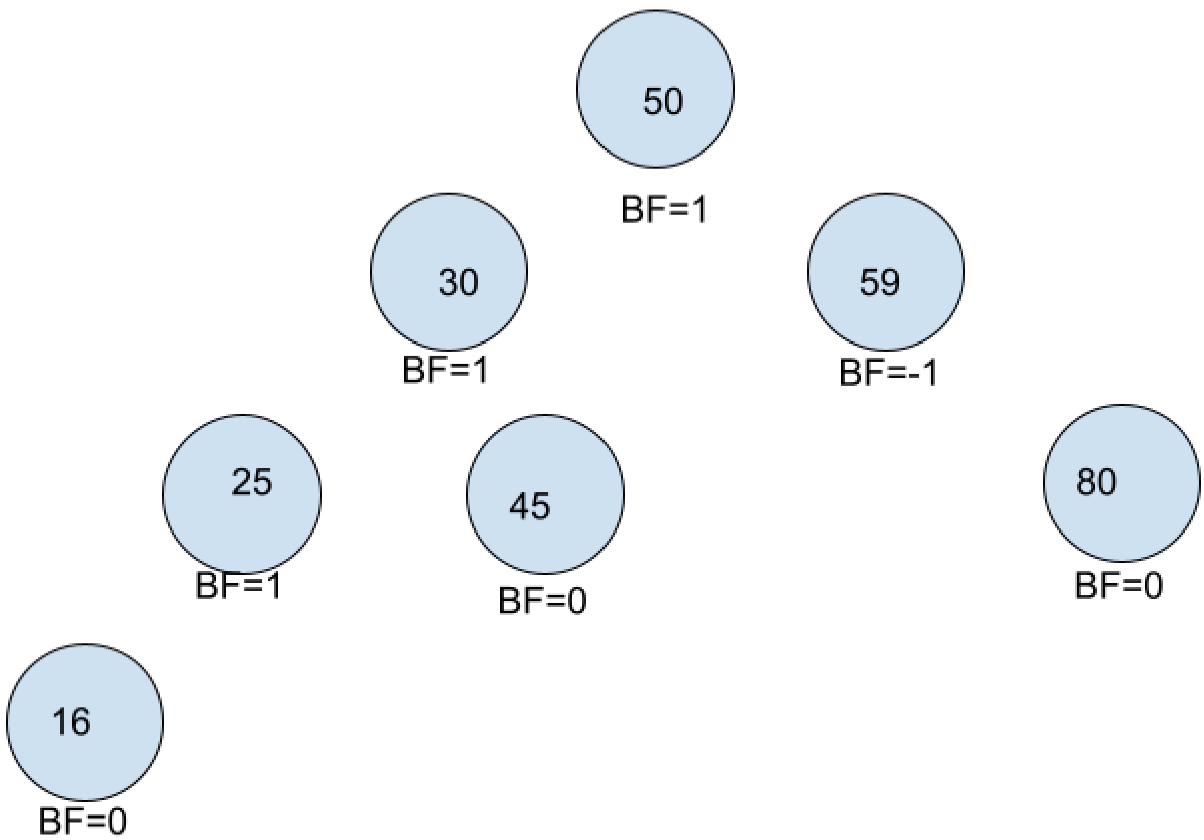
```
def searchT(AVL, element):
  if AVL.root==None:
    return None
  else:
    return searchByValue(AVL.root, element)

#Recibe el nodo actual y el elemento a buscar
#recursivamente se mueve por el arbol hasta encontrar la key correspondiente al valor
def searchByValue(currentNode, element):
  if currentNode!=None:
    if currentNode.value==element:
      return currentNode.key
    else:
      key=searchByValue(currentNode.leftnode, element)
      if key!=None:
        return key
      key=searchByValue(currentNode.rightnode, element)
      if key!=None:
        return key
```

# 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

      Este es un contraejemplo de un árbol avl balanceado sin el penúltimo nivel completo que cumple con todos los requisitos de avl



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

   El tener todos los nodos con BF=0 implica que todos tiene o 0 o 2 hijos y todas las ramas tienen la misma cantidad de nodos.

c. **F** En la inserción en un AVL, si al actualizar 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.

   Al insertar un elemento puede no desbalancear al nodo padre pero si a la raíz ya que si por ejemplo el nodo era BF=0 sin hijos, se le agrega 1 y puede ser que la rama sea ahora la más grande entonces cambiaría el BF de la raíz

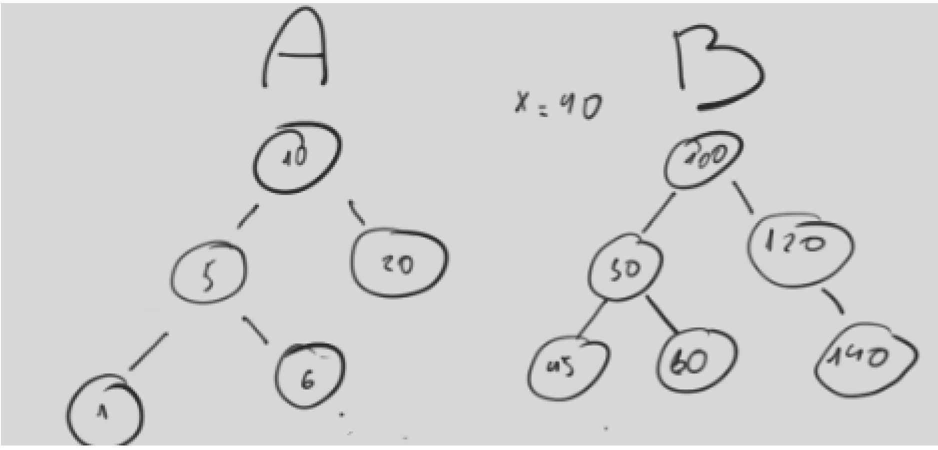
d. **V** En todo *AVL* existe al menos un nodo con factor de balance 0.

   Los últimos nodos tendrán sí o sí BF = 0 ya que al ser nodos hoja no tienen hijos.

2.

# 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*.



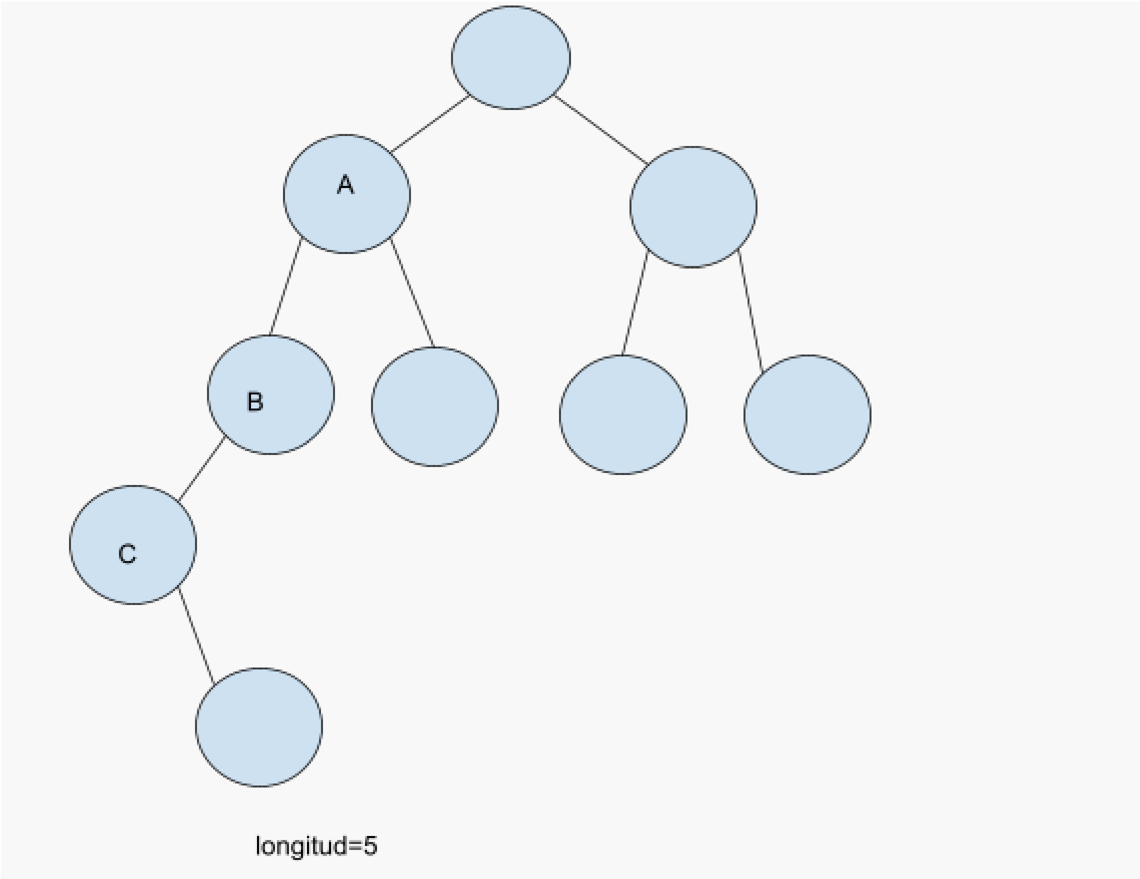

Buscar la altura del árbol A y B, me quedo en el árbol más alto y busco la altura del árbol más chico en el árbol actual
En esa altura insertar la nueva key y como hijo de esa key el árbol más pequeño
Así solo habrá que rebalancear el nuevo nodo como mucho ya que estamos agregando un nodo más a lo sumo a la máxima altura.

# 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.

Al ser un árbol AVL no puede haber una rama truncada muy extensa porque debería balancearse.




En este ejemplo h=4 => h/2=2
Este no podría ser un AVL ya que el nodo B está desbalanceado porque la rama truncada tiene longitud 5 y debe balancearse.

La mínima altura para una rama truncada siempre va a ser una unidad menor a la máxima altura del árbol, porque sino estaría desbalanceado ya que a lo sumo puede tener un nodo hijo y este debe ser una hoja.