CODIGO

```python
#Funcion que calcula la altura de un arbol
def h(node):
   if node is None:
       return 0

   return 1 + max(h(node.leftnode),h(node.rightnode))

#Funcion que calcula el balance factor
def bf(node):
   if node is None:
       return
   return h(node.leftnode)-h(node.rightnode)


#Funcion que realiza una rotacion a la izquierda de los nodos
def rotateLeft(T, old_root):
   new_root = old_root.rightnode

   #Asignamos el hijo izquierdo de new_root a la derecha de old_root
   #Si ese hijo != None, le asignamos a old_root como padre
   old_root.rightnode = new_root.leftnode
   if new_root.leftnode:
       new_root.leftnode.parent = old_root

   #Ponemos old_root a la izquierda de new_root
   new_root.leftnode = old_root
   new_root.parent = old_root.parent
   old_root.parent = new_root

   #Actualizamos el padre de new_root y validamos si debe ser la raiz del arbol o no
   if new_root.parent is None:
       T.root = new_root
   elif new_root.parent.leftnode == old_root:
       new_root.parent.leftnode = new_root
   else:
       new_root.parent.rightnode = new_root

   return new_root

#Funcion que realiza una rotacion a la derecha de los nodos
def rotateRight(T, old_root):
   new_root = old_root.leftnode

   #Asignamos el hijo derecho de new_root a la izquierda de old_root
   #Si ese hijo != None, le asignamos a old_root como padre
   old_root.leftnode = new_root.rightnode
   if new_root.rightnode:
       new_root.rightnode.parent = old_root

   #Ponemos old_root a la derecha de new_root
   new_root.rightnode = old_root
   new_root.parent = old_root.parent
   old_root.parent = new_root

   #Actualizamos el padre de new_root y validamos si debe ser la raiz del arbol o no
```

```
   if new_root.parent is None:
       T.root = new_root
   else:
       if new_root.parent.leftnode == old_root:
           new_root.parent.leftnode = new_root
       else:
           new_root.parent.rightnode = new_root

   return new_root
#Calcula el balance factor de cada uno de los nodos
def calculateBalance(T):

   def bf_recursive(node):
       if node is None:
           return 0
       node.bf = bf(node)
       bf_recursive(node.leftnode)
       bf_recursive(node.rightnode)

   bf_recursive(T.root)
   return

#reBalance para cada nodo
def reB_node(T, node):
       if node is None:
           return
       if node.bf > 1:
           rotateRight(T,node)
       if node.bf < -1:
           rotateLeft(T,node)
       calculateBalance(T)

#Funcion reBalance para un arbol
def reBalance(T):
   #Actualizamos los bf del arbol y de ser necesario hacemos rotaciones
   calculateBalance(T)
   reB_node(T, T.root)
   return

#Actualiza el bf del nodo actual, se verifica que no haya que hacer una rotacion
#finalmente llamamos al parent del nodo
#Esta funcion se va  usar principalmente cuando hagamos un delete o un insert
def updateBf(T,node):
   node.bf = bf(node)
   reB_node(T,node)
   if node.parent != None:
       updateBf(T,node.parent)
   return

#Insertamos un elemento en el AVL
def insert(T,element,key):
   new_node = AVLNode()
   new_node.value = element
   new_node.key = key

   #Caso con root vacio
   if T.root is None:
```

```
        T.root = new_node
        return key

    #Funcion recursiva que explora el arbol y devuelve verdadero si se pudo insertar el nodo
    def insert_node(node):
        #Verifica nodo izquierdo
        if new_node.key < node.key:
            if node.leftnode is None:
               node.leftnode = new_node
               new_node.parent = node
               return True
            else:
                return insert_node(node.leftnode)
        #Verifica nodo derecho
        elif (new_node.key > node.key):
            if node.rightnode is None:
                node.rightnode = new_node
                new_node.parent = node
                return True
            else:
                return insert_node(node.rightnode)
        else:
            return None

    #Llamamos a la funcion recursiva
    if insert_node(T.root):
        updateBf(T, new_node)
        return key
    else:
        return None


#Buscamos un elemento (.value) en un arbol dado
def search(T, element):
    #Compara el elemento ingresado con el .value de un nodo
    #Funcion recursiva
    def search_node(node, element):
        if node is None:
            return None
        if node.value == element:
            return node.key

        # Buscar en el subárbol izquierdo
        left_result = search_node(node.leftnode, element)
        if left_result is not None:
            return left_result

        # Buscar en el subárbol derecho
        return search_node(node.rightnode, element)

    return search_node(T.root,element)


#Funcion para borrar/desvincular un nodo con una key determinada
def deleteKey(T,key):
```

```
   #Funcion que hace el reemplazo de un nuevo nodo en el lugar del nodo que queremos
reemplazar
   def replace(parent, replace_node, new_child):
       if parent is None:  #Eliminamos la raiz
           T.root = new_child
       elif parent.leftnode == replace_node:
           parent.leftnode = new_child             #Cambiamos el child node
       elif parent.rightnode == replace_node:
           parent.rightnode = new_child            #Cambiamos el child node
       if new_child is not None:
           new_child.parent = parent               #Cambiamos el parent del child node

   def delete_node_key(node):
       #Caso sin child nodes
       if node.leftnode is None and node.rightnode is None:
           replace(node.parent,node,None)
       #Casos 1 childnode
       elif node.leftnode is None:
           replace(node.parent, node, node.rightnode)
       elif node.rightnode is None:
           replace(node.parent, node, node.leftnode)
       #Caso 2 childnodes
       else:
           max_node = node.rightnode
           while max_node.leftnode is not None:
               max_node = max_node.leftnode

           node.key = max_node.key
           node.value = max_node.value
           delete_node_key(max_node)
           #En esta parte hacemos lo siguiente:
           #1. Ubicamos "el menor de los mayores"
           #2. Pasamos su key y value al nodo actual
           #3. Eliminamos el nodo que ocupa la posicion de "El menor de los mayores"

   #Buscamos el elemento que queremos borrar por su key
   def find_node_key(node,key):
       if node is None:
           return None
       if key < node.key:
           return find_node_key(node.leftnode, key)
       elif key > node.key:
           return find_node_key(node.rightnode, key)
       else:
           return node

   #Buscamos el elemento a borrar y si logramos borrarlo devolvemos su key
   target = find_node_key(T.root,key)
   if target is not None:
       bf_check = target.parent
       delete_node_key(target)
       updateBf(T, bf_check)
       return key
   else:
       return None

#Funcion para borrar/desvincular un nodo con .value == element
```

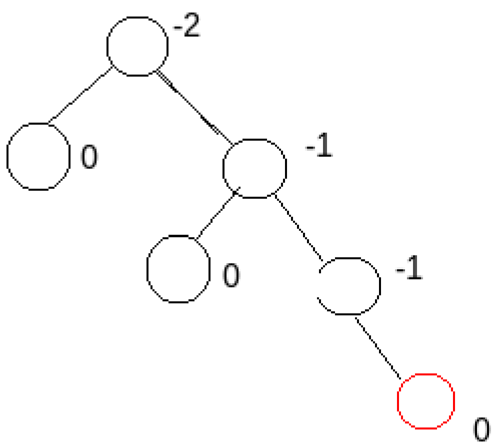
```python
def delete(T,element):
   key_to_find = search(T,element)
   if key_to_find is None:
       return None

   #Usamos la funcion deleteKey para borrar ese elemento
   return deleteKey(T,key_to_find)
```

PARTE 2 - AVL

Ejercicio 6:

1. **Responder V o F y justificar su respuesta:**
    a. **___ En un AVL el penúltimo nivel tiene que estar completo**
    b. **___ Un AVL donde todos los nodos tengan factor de balance 0 es completo**
    c. **___ 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.**
    d. **___ En todo *AVL* existe al menos un nodo con factor de balance 0.**

a. F. No es necesario que el penúltimo nivel esté completo. Podemos encontrar ejemplos de AVLs en los que esa condición no se cumpla.
b. V. Podemos probar hacer un árbol nosotros, y para mantener el bf=0 en todos los nodos, vamos a ver que es necesario completar el árbol, si no estuviese completo encontraríamos algún nodo que no cumpla con esa propiedad.
c. F. Si bien el padre del nodo que agregamos puede no estar desbalanceado, al agregar un nodo podemos desbalancear la estructura "mayor" del árbol. Ej:

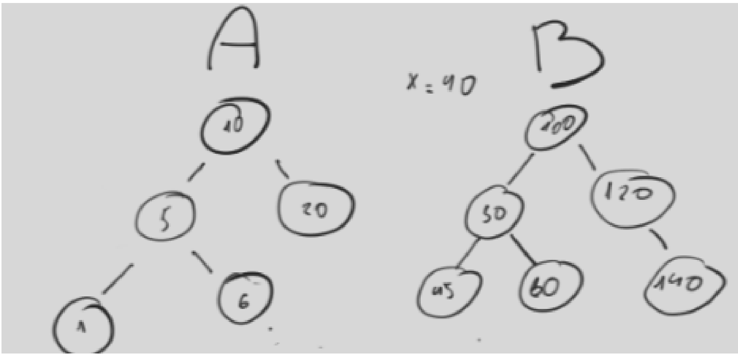


Luego de agregar el nodo rojo, su padre no está desbalanceado y tampoco el padre del padre. Sin embargo, vemos que la raíz árbol ahora sí se encuentra desbalanceada.

d.V. Las hojas de nuestro AVL van a tener un bf=0. También tendremos nodos con bf=0 en el caso de que sus dos subárboles laterales tengan la misma altura.

Ejercicio 7:

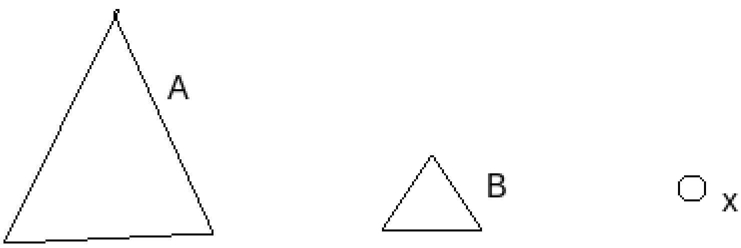
Sean *A* y *B* dos *AVL* de *m* y *n* nodos respectivamente y sea *x* un key cualquiera de forma tal que para todo key *a* ∈ *A* y para todo key *b* ∈ *B* se cumple que *a* < *x* < *b*. Plantear un algoritmo *O*(log *n* + log *m*) que devuelva un *AVL* que contenga los key de *A*, el key *x* y los key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

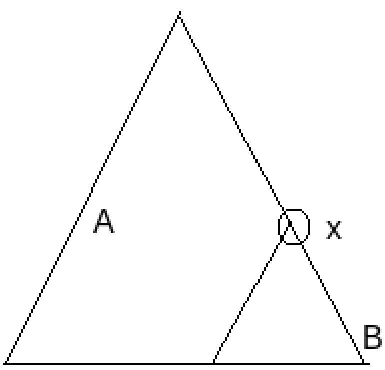


Por el orden de complejidad que se nos pide, sabemos que no vamos a poder usar una solución en la que tengamos que balancear el árbol múltiples veces (Ej: insertar los nodos uno por uno tomando a x como la raíz).

Sabemos que recorrer un AVL, es una tarea que se realiza en O(log n). Guiándonos por esto, llegamos al siguiente algoritmo:

1. Evaluamos las alturas de nuestros AVLs A y B (h(A)-h(B)). Esto nos debería dar como resultado: un número positivo si A es más alto que B, 0 si son igual de altos y un número negativo si B es más alto que A.
2. Tomamos el resultado anterior para decidir sobre qué árbol operaremos. (siempre operaremos sobre el árbol más alto y si son iguales podremos elegir cualquiera)
3. Sabemos que **a<x<b**, y podremos usar esto a nuestro favor. La idea general consiste en compensar la altura del árbol más pequeño, con los niveles más altos del árbol más grande.





4. Antes de hacer una inserción, debemos calcular el balance factor de la raíz del árbol que vamos a trabajar; hacemos esto para evitar cualquier clase de desequilibrio al final de la inserción. Casos:

   Si vamos a hacer una inserción en A, cuya raíz tiene un **bf = -1**, quiere decir que el subárbol derecho (el que nos interesa) es más alto que el subárbol izquierdo. **Solución = hacer una rotación hacia la izquierda en la raíz de A.**

   Si vamos a hacer una inserción en B, cuya raíz tiene un **bf = 1**, quiere decir que el subárbol izquierdo (el que nos interesa) es más alto que el subárbol derecho. **Solución = hacer una rotación hacia la derecha en la raíz de B.**

5. Una vez hecho esto, ya podemos unir todos los nodos. Usamos el cálculo hecho en el paso **1.** ((h(A)-h(B)) para saber en que nivel debemos insertar **x**. Se vería algo así:

```
#Agrupamos en A

current = A.head

nivel_x = h(A)-h(B)

count = 0

while count < nivel_x:

        current = current.rightnode

#”Hacemos espacio” para insertar x y movemos el nodo de ese lugar como hijo izquierdo de x

xNode.parent = current.parent

current.parent = xNode

xNode.leftnode = current

#Conectamos B a la derecha de x

xNode.rightnode = B.root

B.root.parent = xNode.
```

Para insertar en el árbol B el proceso se repetiría, pero tendríamos que bajar por la izquierda del árbol, insertar el nodo “placeholder” de **x** a la derecha de este e insertar A a la izquierda 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.

Para comenzar tomamos un caso general de un árbol balanceado de altura h, para que el árbol esté balanceado los hijos tienen que diferenciarse en su altura en como mucho una unidad y deben ser menor en una o dos unidades a la altura de su padre para poder seguir siendo balanceado.

Consideremos el caso en el que un hijo se diferencia en una unidad de la altura del padre, y el otro se diferencia en dos, y así sucesivamente.

Para calcular la mínima longitud de una rama truncada, sabemos que va a ser igual al nivel del primer nodo que tiene una referencia a none; y recordando el planteamiento anterior, vamos a tener una rama cuyos hijos “bajan” de a **h-1,** otra cuyos hijos “bajen” de a **h-2**(por la diferencia de altura que tienen con su padre).

Cuando la altura del nodo sea igual a 0, estaremos en un nodo hoja. Y también sabemos que si bajamos por la rama en la que los hijos tienen 2 de diferencia de altura con su padre (**h-2**), obtendremos: h-2,h-4…**h-2k.** Y podremos plantear lo siguiente:

h-2k=0 —--> **h=2k** —----> **h/2=k**

Por lo tanto, el menor nivel con rama truncada (o mínima longitud de una rama truncada) es h/2.