Parte 1: ALGO 2 Estudiante: Joaquin Villegas

1-

```
def rotateLeft(Tree, avlnode):
    if Tree.root == avlnode:
        newRoot = avlnode.rightnode
        leftSon = newRoot.leftnode
        newRoot.parent = None
        avlnode.rightnode = None
        avlnode.parent = newRoot
        Tree.root = newRoot
        newRoot.leftnode = avlnode
        avlnode.rightnode = leftSon
        if leftSon != None:
            leftSon.parent = avlnode
    else:
        newRoot = avlnode.rightnode
        leftSon = newRoot.leftnode
        newRoot.parent = avlnode.parent
        if avlnode.parent.leftnode == avlnode:
            avlnode.parent.leftnode = newRoot
        else:
            avlnode.parent.rightnode = newRoot
        avlnode.parent = newRoot
        avlnode.rightnode = leftSon
        newRoot.leftnode = avlnode
        if leftSon != None:
            leftSon.parent = avlnode
```

```
def rotateRight(Tree, avlnode):
    if Tree.root == avlnode:
        newRoot = avlnode.leftnode
        rightSon = newRoot.rightnode
        newRoot.parent = None
        Tree.root = newRoot
        avlnode.parent = newRoot
        newRoot.rightnode = avlnode
        avlnode.leftnode = rightSon
        if rightSon != None:
            rightSon.parent = avlnode
    else:
        newRoot = avlnode.leftnode
        rightSon = newRoot.rightnode
        newRoot.parent = avlnode.parent
        if avlnode.parent.leftnode == avlnode:
            avlnode.parent.leftnode = newRoot
        else:
            avlnode.parent.rightnode = newRoot
        avlnode.parent = newRoot
        avlnode.leftnode = rightSon
        newRoot.rightnode = avlnode
        if rightSon != None:
            rightSon.parent = avlnode
```

2-

```
def calculateBalance(AVLTree):
    return calculateBalanceR(AVLTree, AVLTree.root)

def heightTree(currentNode):
  if currentNode == None:
    return 0

  hLeft = heightTree(currentNode.leftnode)
  hRight = heightTree(currentNode.rightnode)

  if hLeft >= hRight:
    hTree = hLeft + 1
  else:
    hTree = hRight + 1

  return hTree

def calculateBalanceR(AVLTree, currentNode):
  bfValue = heightTree(currentNode.leftnode) - heightTree(currentNode.rightnode)
  currentNode.bf = bfValue

  if currentNode.leftnode != None:
    calculateBalanceR(AVLTree, currentNode.leftnode)
  if currentNode.rightnode != None:
    calculateBalanceR(AVLTree, currentNode.rightnode)

  if AVLTree.root == currentNode:
    return AVLTree
```

3-

```
def reBalance(AVLTree, currentNode):
    if currentNode.bf < 0:
        if currentNode.rightnode.bf > 0:
            rotateRight(AVLTree, currentNode.rightnode)
            rotateLeft(AVLTree, currentNode)
        else:
            rotateLeft(AVLTree, currentNode)
    elif currentNode.bf > 0:
        if currentNode.leftnode.bf < 0:
            rotateLeft(AVLTree, currentNode.leftnode)
            rotateRight(AVLTree, currentNode)
        else:
            rotateRight(AVLTree, currentNode)
    calculateBalance(AVLTree)
# def reBalance(AVLTree):
#     calculateBalance(AVLTree)

#     while searchDesbalance(AVLTree, AVLTree.root) == False:
#         searchDesbalance(AVLTree, AVLTree.root)
```

4-

```
def crearNodo(element, key, currentNode):
  nodoAInsertar = AVLNode()
  nodoAInsertar.key = key
  nodoAInsertar.value = element
  nodoAInsertar.parent = currentNode
  return nodoAInsertar

def buscarPosicion(B, element, key, currentNode):
  if key == currentNode.key:
    return None

  if key > currentNode.key:
    if currentNode.rightnode != None:
      nodoInsert = buscarPosicion(B, element, key, currentNode.rightnode)
    else:
      nodoInsert = crearNodo(element, key, currentNode)
      currentNode.rightnode = nodoInsert
  else:
    if currentNode.leftnode != None:
      nodoInsert = buscarPosicion(B, element, key, currentNode.leftnode)
    else:
      nodoInsert = crearNodo(element, key, currentNode)
      currentNode.leftnode = nodoInsert

  return nodoInsert


def insert(B, element, key):
  if B.root == None:
    nodoRaiz = AVLNode()
    nodoRaiz.key = key
    nodoRaiz.value = element
    nodoRaiz.parent = None
    nodoRaiz.bf = 0
    B.root = nodoRaiz
    return nodoRaiz.key
  else:
    nodo = buscarPosicion(B, element, key, B.root)
    update_bf(B, nodo)
    return  nodo.key
```

```
def update_bf(AVLTree, currentNode):
    auxNode = currentNode
    while auxNode != None:
        bfValue = heightTree(auxNode.leftnode) - heightTree(auxNode.rightnode)
        auxNode.bf = bfValue
        # print("/////")
        # print(auxNode.key)
        # print(auxNode.bf)
        # print("/////")
        auxNode = auxNode.parent

    while currentNode != None:
        if currentNode.bf > 1 or currentNode.bf < -1:

            reBalance(AVLTree, currentNode)
        currentNode = currentNode.parent
    return
```

5-

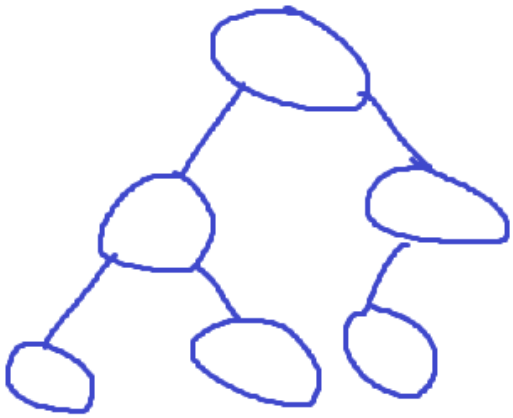
```
def DeleteElement(currentNode):
  #Primera situacion: el nodo a eliminar es una hoja.
  if currentNode.rightnode == None and currentNode.leftnode == None:
    if currentNode.parent.leftnode == currentNode:
      currentNode.parent.leftnode = None
    else:
      currentNode.parent.rightnode = None
    currentNode.parent = None
    return currentNode.key

  #Tercera Situacion: el nodo a eliminar tiene dos hijos.
  if currentNode.rightnode != None and currentNode.leftnode != None:
    #Buscamos el mayor elemento de los menores SI ES QUE TIENE, En el caso contrario, deberemos buscar el menor elemento de los mayores.
    nodeAux = currentNode.leftnode

    while nodeAux.rightnode != None:
      nodeAux = nodeAux.rightnode
    valueAux = nodeAux.value
    keyAux = nodeAux.key
    keyEliminada = currentNode.key
    DeleteElement(nodeAux)
    currentNode.value = valueAux
    currentNode.key = keyAux
    return keyEliminada

  #Segunda Situacion: el nodo a eliminar tiene un hijo.
  #¿Debo decir que su parent sea None?
  if currentNode.rightnode == None and currentNode.leftnode != None:
    if currentNode.parent.leftnode == currentNode:
      currentNode.leftnode.parent = currentNode.parent
      currentNode.parent.leftnode = currentNode.leftnode
    else:
      currentNode.leftnode.parent = currentNode.parent
      currentNode.parent.rightnode = currentNode.leftnode
    currentNode.parent = None
    return currentNode.key
  else:
    if currentNode.rightnode != None and currentNode.leftnode == None:
      if currentNode.parent.leftnode == currentNode:
        currentNode.rightnode.parent = currentNode.parent
        currentNode.parent.leftnode = currentNode.rightnode
      else:
        currentNode.rightnode.parent = currentNode.parent
        currentNode.parent.rightnode = currentNode.rightnode
    currentNode.parent = None
    return currentNode.key
```

```
def busquedaDeleteKey(B, key, currentNode):
  keyEliminated = None
  if currentNode.key == key:
    keyEliminated = DeleteElement(currentNode)
  else:
    if currentNode.leftnode != None and keyEliminated == None:
      keyEliminated = busquedaDeleteKey(B, key, currentNode.leftnode)
    if currentNode.rightnode != None and keyEliminated == None:
      keyEliminated = busquedaDeleteKey(B, key, currentNode.rightnode)

  if keyEliminated != None:
    return keyEliminated
  else:
    return None

def deleteKey(B, key):
  if B.root == None:
    return None
  else:
    keyEliminated = busquedaDeleteKey(B, key, B.root)

    calculateBalance(B)
    while searchDesbalance(B, B.root) == False:
      searchDesbalance(B, B.root)

def searchDesbalance(AVLTree, currentNode):
  boolDesbalance = True
  if currentNode.bf > 1 or currentNode.bf < -1:
    reBalance(AVLTree, currentNode)
    boolDesbalance = False
  else:
    if currentNode.leftnode != None:
      searchDesbalance(AVLTree, currentNode.leftnode)
    if currentNode.rightnode != None:
      searchDesbalance(AVLTree, currentNode.rightnode)
  return boolDesbalance
```

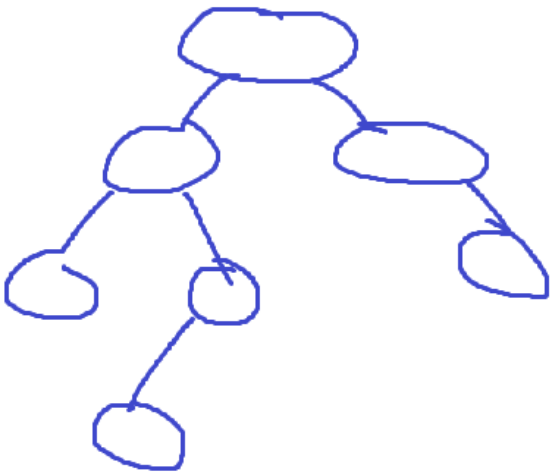
Parte 2:

6-

a. (en este inciso me quedo la duda sobre si se refiere a que este completo significa que todos los nodos de ese nivel tengan dos hijos o que el nivel tenga la cantidad de nodos máximos que se puede tener, asi que responderé a las 2).
   Es falso, porque puedo demostrar un contraejemplo de que no se cumple:

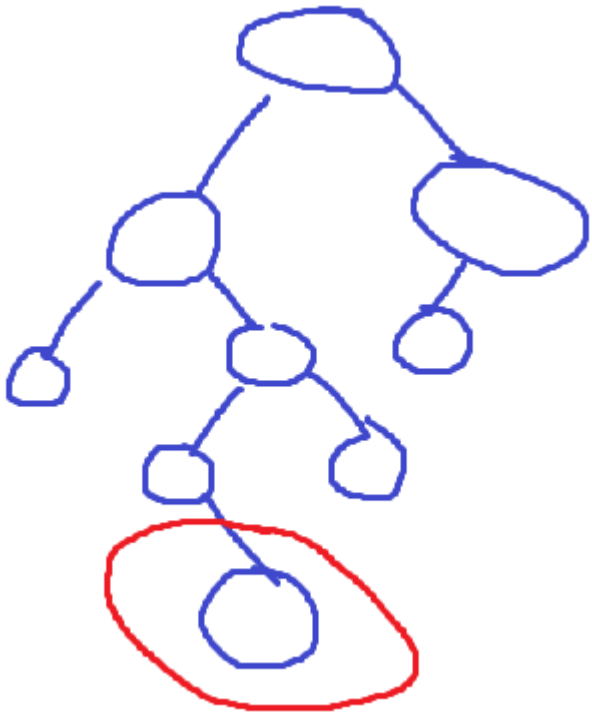


   Este árbol es AVL pero aun asi, hay un nodo del penúltimo nivel que no tiene 2 hijos.
   En la segunda también es falsa porque:



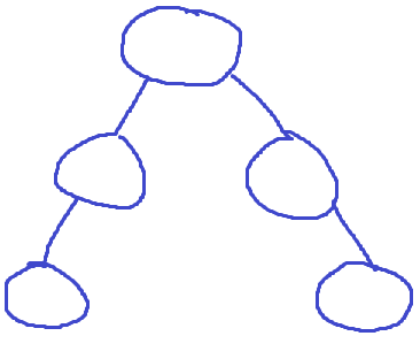
   Vemos que el penúltimo nivel le falta un nodo para que este completo, y aun así es AVL.

b. Es verdadero, ya que si encontramos un nodo que contenga solo un hijo este le modifica el balance factor haciendo que no sea igual a 0.
c. Es falso, lo podemos ver con el siguiente ejemplo.



vemos que no se causa algún problema en el nodo padre (del que esta encirculado en rojo) pero si seguimos subiendo vemos que hay nodos donde si hay problemas con el bf.

d. (en este inciso el profesor Jorge aclaro que en este parece que falto aclarar que se excluye nodo raíz y hojas).
Asi que bien, si es como de verdad esta escrito, es verdadero, porque como sabemos las hojas van a tener bf = 0.
Pero si las excluimos con la raíz, es falso:



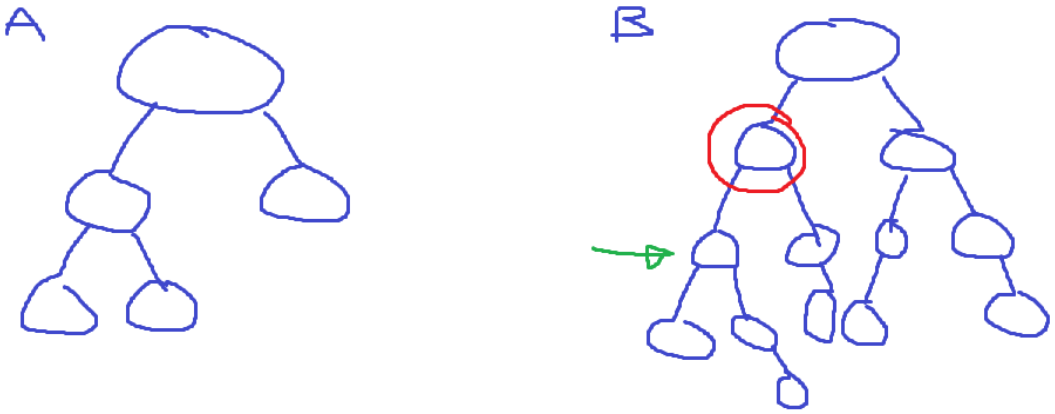
podemos ver aca un ejemplo

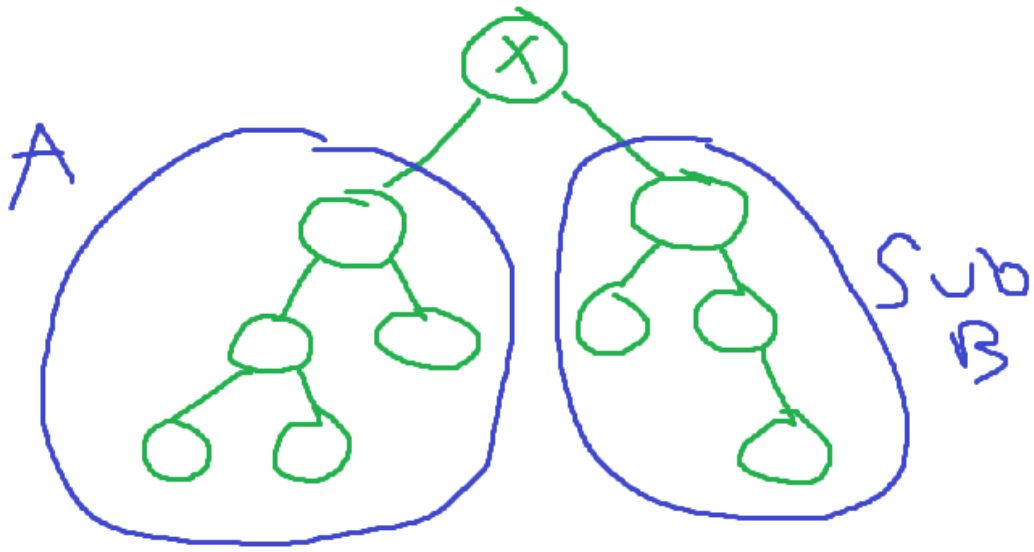
de que los que no son ni hoja ni raíz, su bf es distinto de 0. Asi que no se cumple para todo AVL.

7- Se plantea un algoritmo donde primero se busca las alturas del árbol A y B. Si ambas son iguales, no hay problema alguno y les damos de valor parent a las raíces de cada árbol un nodo con la key x, teniendo ese nodo X como subárbol izquierdo a A y de subárbol derecho a B. (Acordarse de que este nodo X será la raíz del nuevo árbol)

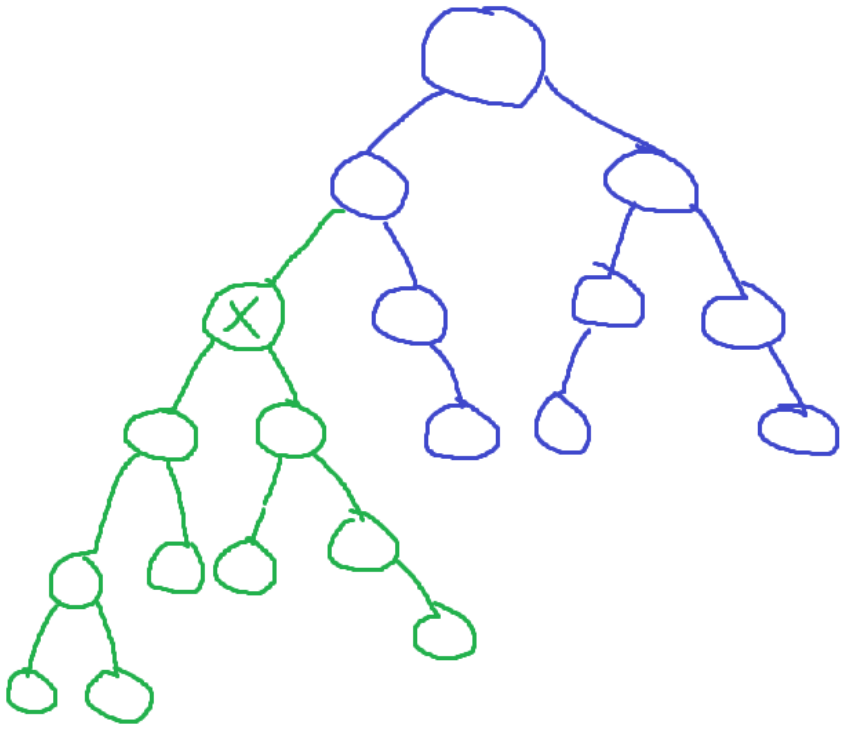
En el caso de alturas distintas, en el que tenga mayor altura debemos de conseguir un subárbol que tenga la misma altura que el de menor (si A es el mas grande buscamos a lo mas derecha posible, y en caso de B a lo mas izquierda), de forma tal que cuando la encontremos, podremos hacer lo mismo en el anterior párrafo. PERO agregando aca, que el nodo X no va a ser la raíz del nuevo árbol, si no de un subárbol. De parent tendrá al nodo del árbol que le cortamos el hijo para que se encontrara la misma altura que el otro.

Por ejemplo: un árbol A de altura 2 y árbol B de altura 4





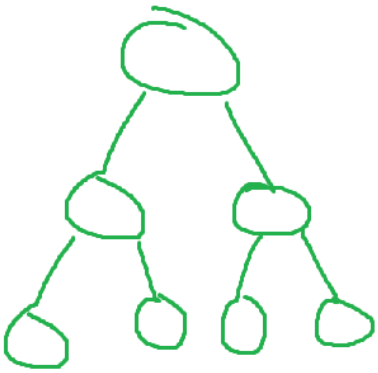
Y esto importante, en este caso el nodo X va a ser el hijo izquierdo del que vemos marcado de rojo. Pero si es en el caso de A, es el hijo derecho.



Nos queda algo asi, como se puede ver esta desbalanceado (pero hay casos en el que queda balanceado), lo que hay que hacer entonces es a partir del nodo X hasta arriba aplicar casos de rotación.

8-

Se demuestra que es falso, veamos este árbol



Vemos que es de h = 2, según el enunciado dice que la minima longitud (cantidad de aristas) es de h / 2. Pero aca da 1, sin embargo viendo la grafica vemos que la cantidad de aristas en realidad es 2.