Algoritmo y Estructura de Datos 2

Trabajo Práctico Número 2: Árboles AVL

Parte 1:

Ejercicio 1:

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

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

        avlnode.rightnode = newroot.leftnode
        avlnode.parent = newroot
        newroot.leftnode = avlnode
        if avlnode.rightnode != None:
                avlnode.rightnode.parent = avlnode
```

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(Tree,avlnode):
        newroot = avlnode.leftnode
        newroot.parent = avlnode.parent
        if avlnode.parent == None:
                Tree.root = newroot
        else:
                if avlnode.parent.leftnode == avlnode:
                        avlnode.parent.leftnode = newroot
                else:
                        avlnode.parent.rightnode = newroot

        avlnode.leftnode = newroot.rightnode
        avlnode.parent = newroot
        newroot.rightnode = avlnode
        if avlnode.leftnode != None:
                avlnode.leftnode.parent = avlnode
```

Ejercicio 2:

calculateBalance(AVLTree)

Descripción: Calcula el factor de balanceo de un árbol binario de búsqueda.

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

```python
def calculateBalance(AVLTree):
        if AVLTree.root != None:
                if AVLTree.root.leftnode == None and AVLTree.root.rightnode == None:
                        AVLTree.root.bf = 0
                        return
                else:
                        calculateBalanceR(AVLTree.root)
                        return
        else:
                return
```

```python
def calculateBalanceR(currentnode):
        #Caso Base
        if currentnode == None:
                return 0

        #Caso general
        leftH = calculateBalanceR(currentnode.leftnode)
        rightH = calculateBalanceR(currentnode.rightnode)

        currentnode.bf = leftH - rightH

        if leftH >= rightH:
                return leftH + 1
        else:
                return rightH + 1
```

Ejercicio 3:

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 difiere a lo sumo en una unidad.

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

```
def rebalanceR(currentnode,AVLTree):
    #Caso Base
    if currentnode == None:
        return

    #Caso general
    rebalanceR(currentnode.leftnode,AVLTree)
    rebalanceR(currentnode.rightnode,AVLTree)

    if currentnode.bf < -1:
        if currentnode.rightnode.bf > 0:
            rotateRight(AVLTree,currentnode.rightnode)
            rotateLeft(AVLTree,currentnode)
        else:
            rotateLeft(AVLTree,currentnode)
    elif currentnode.bf > 1:
        if currentnode.leftnode.bf < 0:
            rotateLeft(AVLTree,currentnode.leftnode)
            rotateRight(AVLTree,currentnode.rightnode)
        else:
            rotateRight(AVLTree,currentnode)
    else:
        return

    calculateBalance(AVLTree)
```

Ejercicio 4:

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

```python
def insertAVL(AVL, element, key):
  newnode = AVLNode
  newnode.value = element
  newnode.key = key
  newnode.bf = 0
  if AVL.root != None:
    insertR(newnode, AVL.root)
    return rebalanceR(AVL)
  else:
    AVL.root = newnode
    return key


def insertR(newnode, currentnode):
  if newnode.key != currentnode.key:
    if newnode.key > currentnode.key:
      if currentnode.rigthnode == None:
        currentnode.rigthnode = newnode
        newnode.parent = currentnode
        return newnode.key
      else:
        return insertR(newnode, currentnode.rigthnode)
    else:
      if currentnode.leftnode == None:
        currentnode.leftnode = newnode
        newnode.parent = currentnode
        return newnode.key
      else:
        return insertR(newnode, currentnode.leftnode)
  else:
    return
```

Ejercicio 5:

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

```python
def deleteAVL(AVL, element):
  deletenode = accessnodeE(AVL.root, element)
  if deletenode != None:
    deleteNodeCases(AVL, deletenode)
    return reBalance(AVL)
  else:
    return
```

```python
def deleteNodeCases(AVL, deletenode):
  key = deletenode.key

  #Caso 1: El nodo a eliminar es una hoja
  if deletenode.leftnode == None and deletenode.rigthnode == None:
    if deletenode.parent != None:
      if deletenode.parent.leftnode != None and deletenode.parent.leftnode == deletenode:
        deletenode.parent.leftnode = None
        return key
      else:
        deletenode.parent.rigthnode = None
        return key
    else:
      AVL.root = None
      return key
```

```python
  #Caso 2: El nodo a eliminar tiene un hijo

  if deletenode.leftnode != None and deletenode.rigthnode == None:
    if deletenode.parent != None:
      if deletenode.parent.leftnode != None and deletenode.parent.leftnode == deletenode:
        deletenode.parent.leftnode = deletenode.leftnode
      else:
        deletenode.parent.rigthnode = deletenode.leftnode

      deletenode.leftnode.parent = deletenode.parent
      return key

    else:
      AVL.root = deletenode.leftnode
      deletenode.leftnode.parent = None
      return key

  if deletenode.rigthnode != None and deletenode.leftnode == None:
    if deletenode.parent != None:
      if deletenode.parent.leftnode != None and deletenode.parent.leftnode == deletenode:
        deletenode.parent.leftnode = deletenode.rigthnode
      else:
        deletenode.parent.rigthnode = deletenode.rigthnode

      deletenode.rigthnode.parent = deletenode.parent
      return key

    else:
      AVL.root = deletenode.rigthnode
      deletenode.rigthnode.parent = None
      return key
```

```python
    changenode = smaller(deletenode.rigthnode)
    deletenode.value = changenode.value
    deletenode.key = changenode.key

    if changenode.parent.rigthnode == changenode:
      changenode.parent.rigthnode = None
    else:
      changenode.parent.leftnode = None

    return key
```

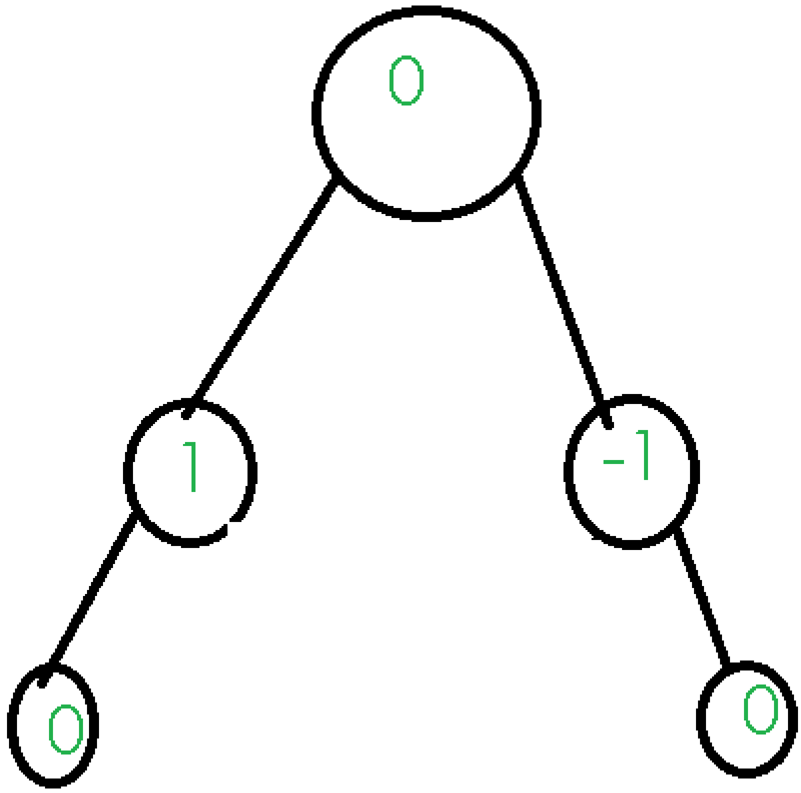
```python
def bigger(node):
    if node.rigthnode != None:
        return bigger(node.rigthnode)
    else:
        return node


def smaller(node):
    if node.leftnode != None:
        return smaller(node.leftnode)
    else:
        return node


def accessnodeE(node, element):
    if node != None:
        if node.value == element:
            return node
        else:
            rigthnode = accessnodeE(node.rigthnode, element)
            if rigthnode != None:
                return rigthnode

            leftnode = accessnodeE(node.leftnode, element)
            if leftnode != None:
                return leftnode
    else:
        return
```

Parte 2:

Ejercicio 6

a. **F** En un AVL el penúltimo nivel tiene que estar completo

Para demostrar que esta proposición es falsa, mostraremos un contraejemplo en donde el penúltimo nivel está incompleto y el árbol sigue siendo de estructura AVL.



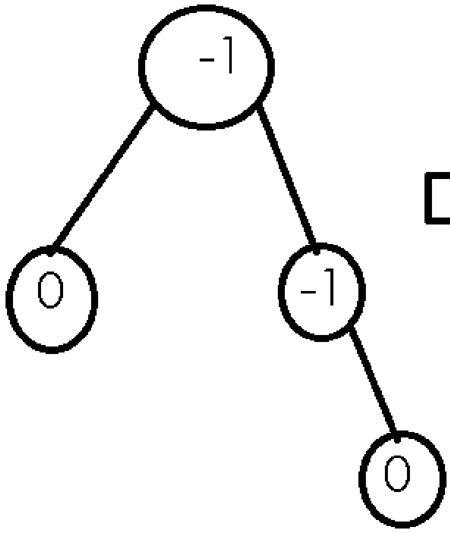
Podemos observar que los nodos que tienen factor de balance 1 y -1 están balanceados aunque no estén completos. Entonces la estructura sigue siendo AVL.

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

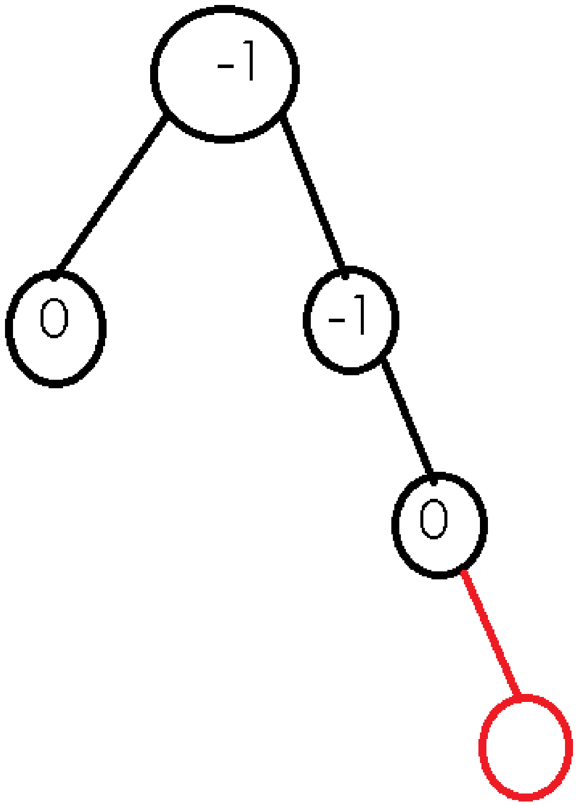
En el caso que nosotros agreguemos un nodo más o lo eliminemos a un AVL donde todos los balance factor sean igual a 0, el balance factor del padre en donde se insertó o en donde se eliminó pasaría a ser distinto de 0.

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.

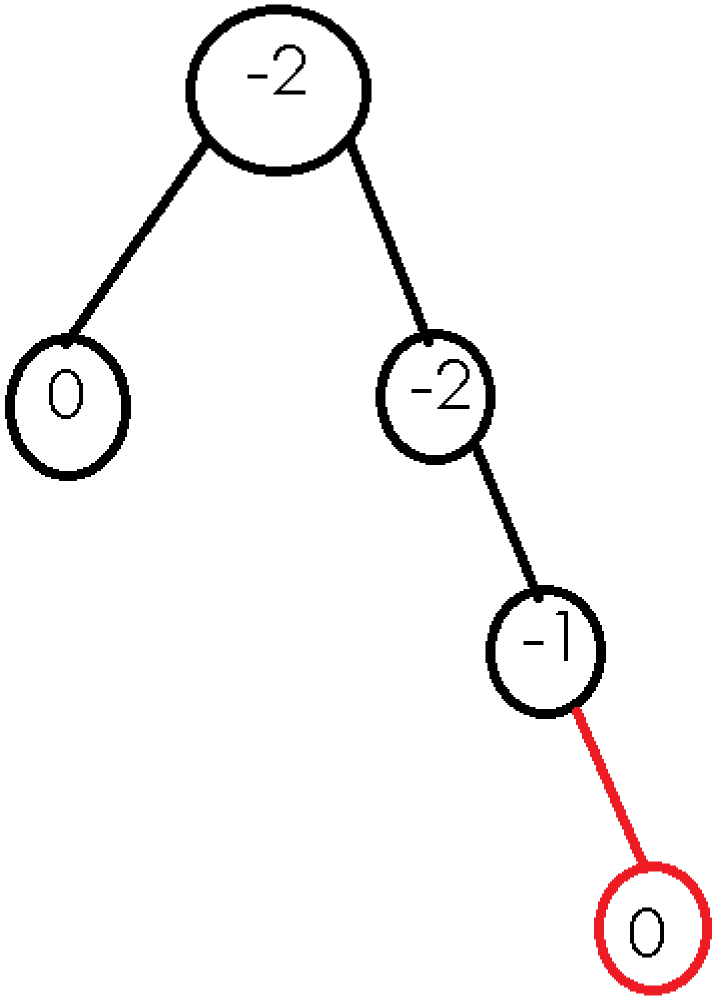
Para demostrar que es Falsa la proposición mostraremos un contraejemplo.



Este es el árbol antes de insertar un nodo, vemos que por ahora cumple la propiedad de ser un AVL



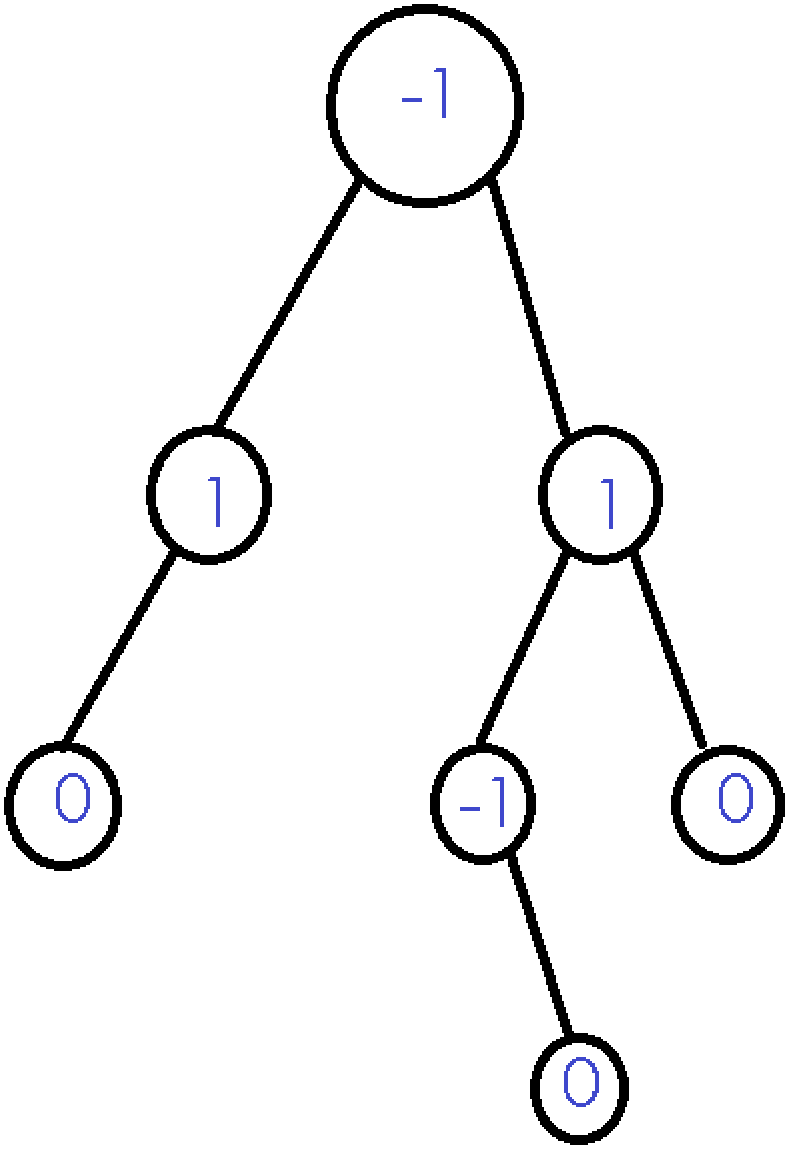
Insertamos el nodo y volvemos a calcular el balance factor desde ese nodo hasta la raíz del árbol



Entonces podemos observar que el padre del nodo insertado su balance factor es -1 (está balanceado), pero esto no quiere decir que no haya que revisitar y volver a calcular el balance factor de los nodos padres hasta la raíz ya que observamos que quedaron desbalanceados (-2 y -2 respectivamente). Por lo tanto la proposición es Falsa.

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

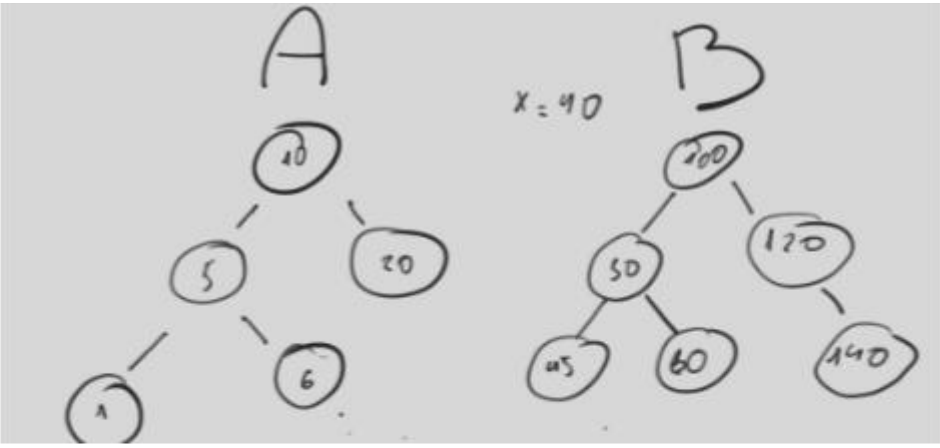
Si no consideramos a los nodos hojas en esta proposición, podemos decir que es Falsa. Para justificarlo mostraremos un árbol AVL que sigue cumpliendo la propiedad aunque todos sus nodos (no hojas) el factor de balanceo sea distinto de 0:



Observamos que el Árbol cumple propiedad de AVL aunque sus nodos internos (nodos no hojas), su factor de balanceo son distintos de 0.

Ejercicio 7:

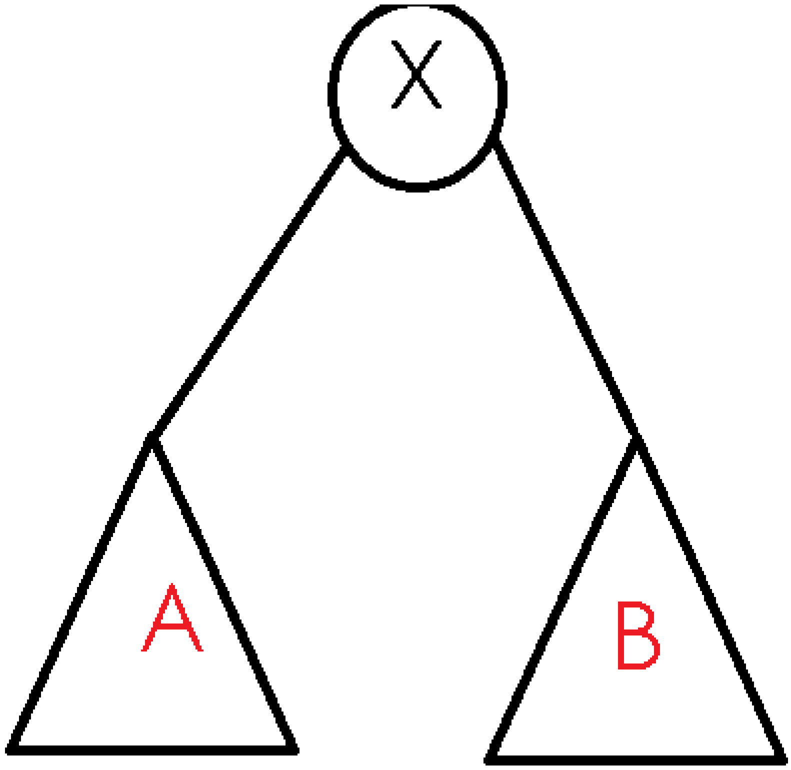
Sean A y B dos AVL de m y n nodos respectivamente y sea x un key cualquiera de forma tal que para todo key a ∈ A y para todo key b ∈ B se cumple que a < x < b. Plantear un algoritmo O(log n + log m) que devuelva un AVL que contenga los key de A, el key x y los key de B.



Para empezar a plantear el Algoritmo lo primero que hacemos es calcular la altura del árbol A y del árbol B. Una vez hecho esto que nos queda por cierto log n y log m (n y m la altura de cada árbol correspondiente) planteamos el algoritmo en cuestión.

Si observamos bien se nos pueden presentar varios casos que difieren a la hora de armar el árbol con la key x.

Nuestro primer caso ocurrirá cuando las alturas de los árboles sean iguales o difieren a lo sumo una unidad. Es el caso más sencillo ya que simplemente en nuestro nuevo árbol luego de hacer raíz a la key x, insertamos todo el subárbol A al nodo izquierdo y todo el subárbol B al nodo derecho. Por lo tanto quedan balanceado el árbol.

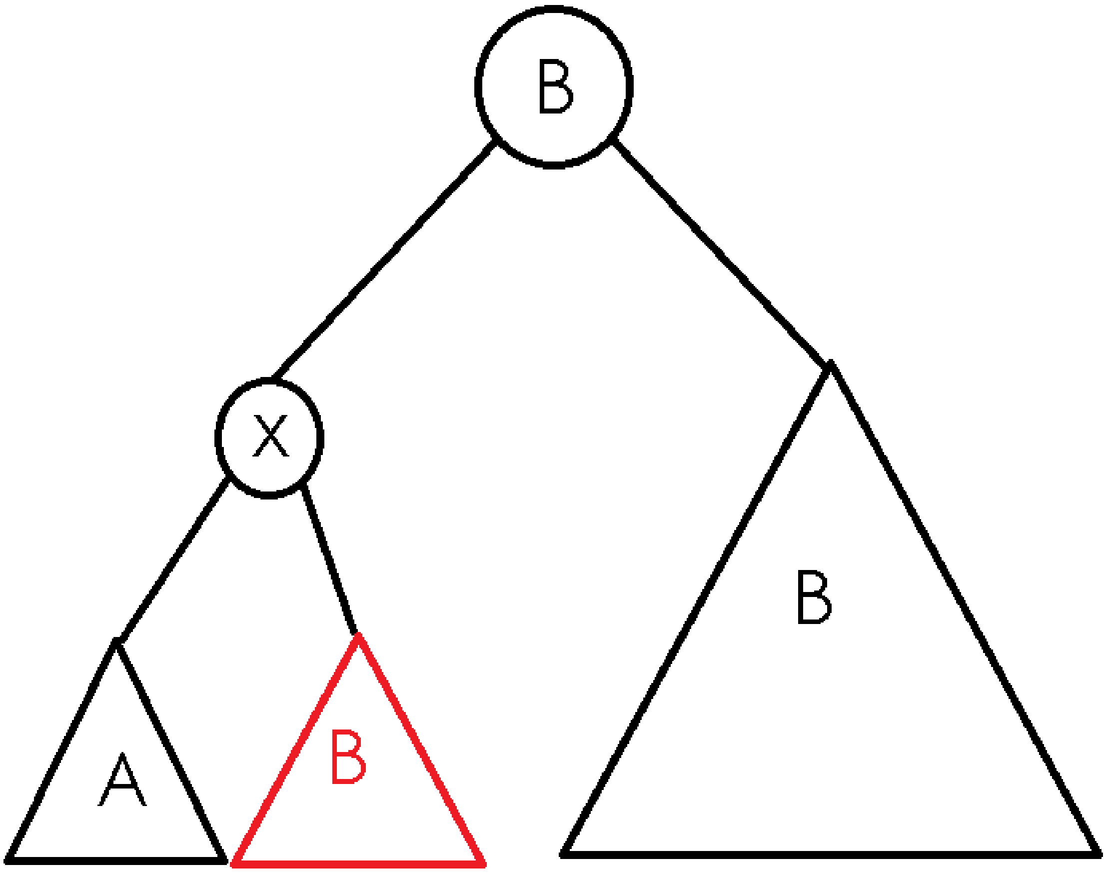


En el Segundo Caso es distinto ya que ocurre que la Altura del árbol B es mucho mayor que la altura del árbol A, es decir que la diferencia es distinto de 0,1 y -1.

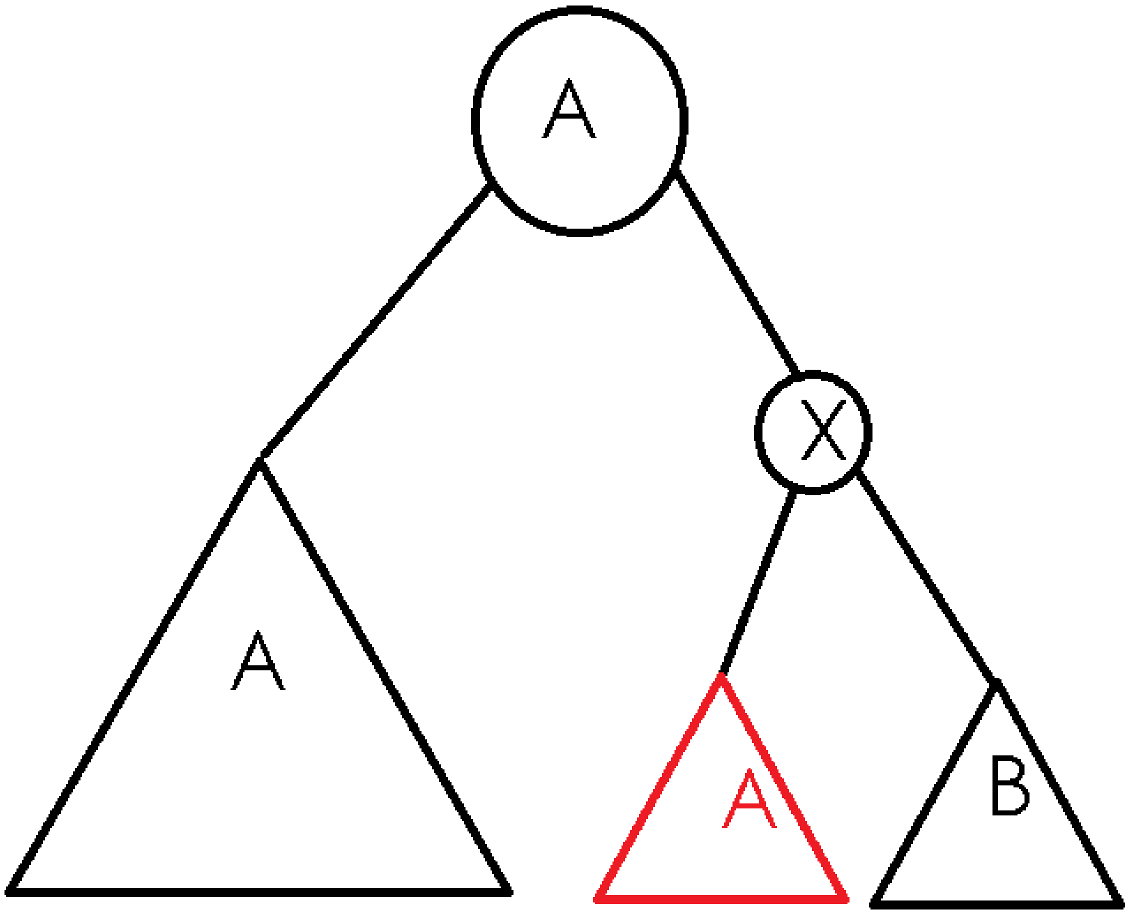
En este caso no podemos simplemente asignar al hijo izquierdo y al hijo derecho como hicimos anteriormente, tenemos que proceder de otra manera tal que nuestro árbol quede balanceado.

Para eso lo que vamos a hacer es buscar en nuestro árbol B un subárbol tal que su altura sea igual a la altura de nuestro árbol A o que difiera en al menos una unidad.

Una vez hecho eso armaremos un subárbol como raíz la key X en donde su nodo izquierdo estará el árbol A y a su derecha el subárbol que encontramos de B. Y para finalizar ubicaremos esto en el nodo izquierdo del árbol B



Y en el Tercer Caso tenemos igual al caso anterior pero la altura A es mucho mayor que la altura del árbol B, por lo tanto procedemos de la misma manera que en el caso anterior modificando en este caso el posicionamiento de cada subárbol, ya que sabemos que todos los keys del Árbol A son menores que los del B entonces el subárbol que contenga a la key con el árbol B y un subárbol con misma altura (o diferencia de uno) de A los colocamos en el nodo derecho de nuestro nuevo Árbol.



En los últimos casos se nos puede presentar que el Árbol que armamos se desbalancee un poco por lo tanto tendremos que aplicar algunas estrategias de rotación para poder balancearlo.

Ejercicio 8:

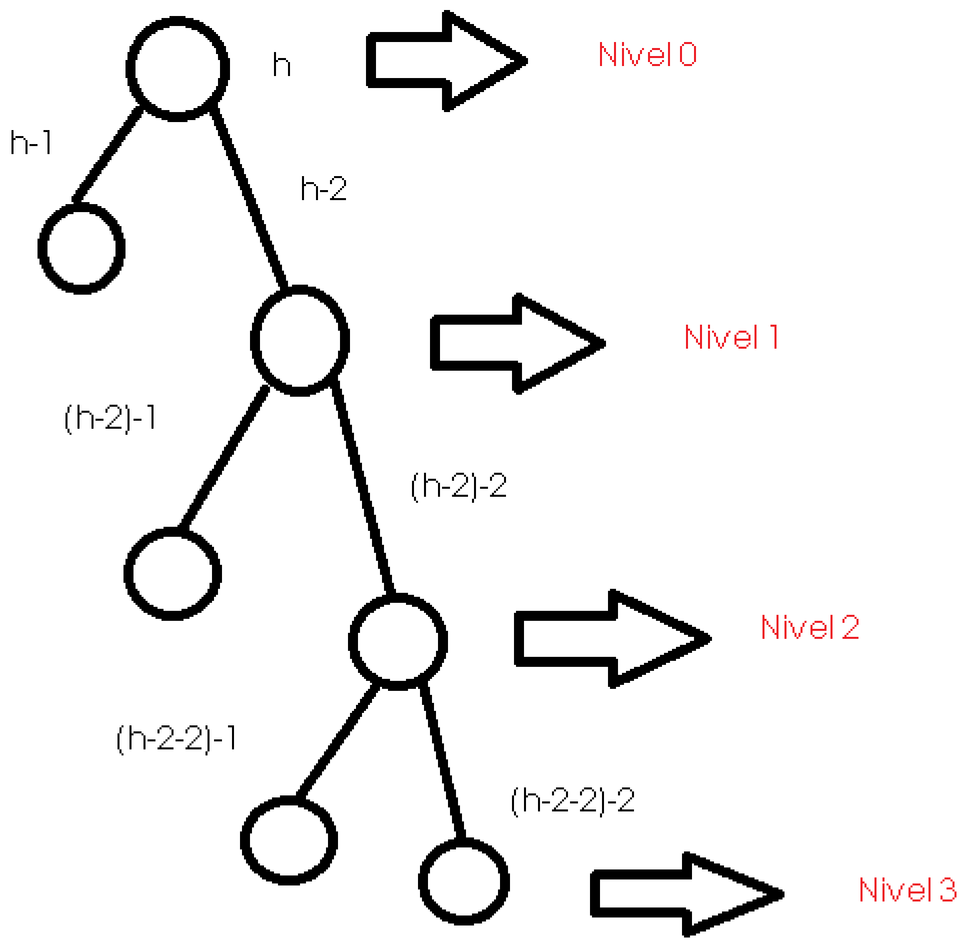
Considere una rama truncada en un AVL como un camino simple desde la raíz hacia un nodo que tenga una referencia None (que le falte algún hijo). Demuestre que la mínima longitud (cantidad de aristas) que puede tener una rama truncada en un AVL de altura h es h/2 (tomando la parte entera por abajo).

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.

En un AVL sabemos que una rama truncada es la cantidad de aristas desde la raíz hacia un nodo con referencia none (es decir que le falte algún hijo o que simplemente sea un nodo hoja).

Nosotros notaremos la mínima longitud desde un nodo raíz a un nodo con referencia none como n, esto quiere decir que al final de la demostración llegaremos a que n = h/2 es decir la altura sobre 2.

En un AVL sabemos que para estar balanceado su BF debe ser 0 o (1 , -1), es decir que difieran a lo menos en una unidad. Pero esto es aplicable para cada nodo del árbol, entonces armaremos un árbol genérico con altura h en donde mostraremos que cada nodo raíz sus alturas difieren a lo menos en una unidad.



En este árbol podemos notar un patrón a medida que vamos descendiendo desde la raíz hasta un nivel n, y es que el número del nivel n multiplicado por 2 es la altura desde el nodo raíz a un nodo de nivel N que tiene dos hijos que su altura difieren a lo menos en uno (por eso denotamos h-1 y h-2, ya que sus diferencias difieren en uno) es decir h-2k

La mínima longitud se va a encontrar desde la raíz del árbol hasta el nivel n, y sabemos que h-2n = 0 ya que es la altura de un nodo hoja.

Entonces:

$h-2n = 0$

$h = 2n$

“Dividimos por 2 a ambos miembros”

$h/2 = n$

Al final quedó demostrado que la altura sobre 2 es igual a la mínima longitud de una rama truncada.