Paula Martínez 13866

Algoritmos y Estructuras de Datos II

Trabajo Práctico: Árboles Balanceados: AVL

Parte 1:

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

search():

```
121  # -------- Operaciones de binarytree adaptadas a AVL Trees --------
122  def search(AVLTree,element):
123    if AVLTree.root==None:
124      return None
125    else:
126      key=None
127      key=searchValue(AVLTree.root,element,key)
128      return key
129  def searchValue(current,element,key):
130    #Busca un elemento en el árbol binario
131    if element==current.value:
132      key=current.key
133      return key
134    else:
135      if current.leftnode!=None:
136        key=searchValue(current.leftnode,element,key)
137      if current.rightnode!=None:
138        key=searchValue(current.rightnode,element,key)
139    return key
```

access():

```
266  def access(AVLTree,key):
267    #Permite acceder a un elemento del árbol AVL con su clave
268    current=AVLTree.root
269    element=None
270    element= accessValue(current,key,element)
271    return element
272  def accessValue(current,key,element):
273    if current.key==key:
274      element=current.value
275      return element
276    if key>current.key:
277      if current.rightnode!=None:
278        element=accessValue(current.rightnode,key,element)
279    elif key<current.key:
280      if current.leftnode!=None:
281        element=accessValue(current.leftnode,key,element)
282    return element
```

update():

```
284 def update(AVLTree,element,key):
285     #Permite cambiar el valor de un elemento del árbol AVL con su clave
286     newNode=AVLNode()
287     newNode.value=element
288     newNode.key=key
289     if AVLTree.root==None:
290         return None
291     else:
292         keyUp=None
293         keyUp=updateNode(newNode,AVLTree.root,keyUp)
294         return keyUp
295 def updateNode(newNode,current,keyUp):
296     if current.key==newNode.key:
297         current.value=newNode.value
298         keyUp=newNode.key
299         return keyUp
300     elif newNode.key>current.key:
301         if current.rightnode!=None:
302             keyUp=updateNode(newNode,current.rightnode,keyUp)
303     elif newNode.key<current.key:
304         if current.leftnode!=None:
305             keyUp=updateNode(newNode,current.leftnode,keyUp)
306     return keyUp
```

accessBF(): (extra)

```
308 def accessBF(AVLTree,key):
309     #Permite acceder al balance factor del árbol AVL con su clave
310     current=AVLTree.root
311     bf=None
312     bf= accessBFofNode(current,key,bf)
313     return bf
314 def accessBFofNode(current,key,bf):
315     if current.key==key:
316         bf=current.bf
317         return bf
318     if key>current.key:
319         if current.rightnode!=None:
320             bf=accessValue(current.rightnode,key,bf)
321     elif key<current.key:
322         if current.leftnode!=None:
323             bf=accessValue(current.leftnode,key,bf)
324     return bf
```

insert() y delete() en ejercicios 4 y 5.

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

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 rotateLeft(Tree,avlNode):
    #Implementa la operación rotación a la izquierda
    newNode= avlNode.rightnode
    avlNode.rightnode= newNode.leftnode
    if newNode.leftnode!=None:
        newNode.leftnode.parent=avlNode
    newNode.parent=avlNode.parent
    if avlNode.parent==None: #if avlNode == Tree.root
        Tree.root=newNode
    else:
        if avlNode.parent.leftnode==avlNode:
            avlNode.parent.leftnode=newNode
        else:
            avlNode.parent.rightnode=newNode
    newNode.leftnode=avlNode
    avlNode.parent=newNode
    return newNode
```

```python
def rotateRight(Tree,avlNode):
    #Implementa la operación rotación a la derecha
    newNode= avlNode.leftnode
    avlNode.leftnode= newNode.rightnode
    if newNode.rightnode!=None:
        newNode.rightnode.parent=avlNode
    newNode.parent=avlNode.parent
    if avlNode.parent==None: #if avlNode == Tree.root
        Tree.root=newNode
    else:
        if avlNode.parent.leftnode==avlNode:
            avlNode.parent.leftnode=newNode
        else:
            avlNode.parent.rightnode=newNode
    newNode.rightnode=avlNode
    avlNode.parent=newNode
    return newNode
```

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(AVLTree):
  #Calcula el balance factor de un árbol binario de búsqueda
  calculateBFPerNode(AVLTree.root)

def calculateBFPerNode(current):
  current.bf=(calculateHeight(current.leftnode) - calculateHeight(current.rightnode))
  if current.leftnode!=None:
    calculateBFPerNode(current.leftnode)
  if current.rightnode!=None:
    calculateBFPerNode(current.rightnode)

def calculateHeight(current):
  #Saca la altura de un nodo
  if current==None:
    return 0
  elif current.leftnode==None and current.rightnode==None:
    return 1
  elif current.leftnode!=None and current.rightnode==None:
    return 1 + calculateHeight(current.leftnode)
  elif current.leftnode==None and current.rightnode!=None:
    return 1 + calculateHeight(current.rightnode)
  else:
    #compara las alturas de los hijos izquierdo y derecho y devuelve la más grande + el mismo nodo
    return 1 + max(calculateHeight(current.leftnode),calculateHeight(current.rightnode))

def calculateBFforOneNode(current):
  #Calcula el balance factor de solo un nodo
  if current!=None:
    current.bf=(calculateHeight(current.leftnode) - calculateHeight(current.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.

```
 98 def reBalance(AVLTree):
 99     calculateBalance(AVLTree)
100     BalanceTree(AVLTree,AVLTree.root)
101 
102 def BalanceTree(AVLTree,current):
103     if current.leftnode!=None:
104         BalanceTree(AVLTree,current.leftnode)
105     if current.rightnode!=None:
106         BalanceTree(AVLTree,current.rightnode)
107     if current.bf>1 or current.bf<-1:
108         if current.bf>1: #rotación a la derecha
109             if current.leftnode!=None and current.leftnode.bf==-1:
110                 rotateLeft(AVLTree,current.leftnode)
111                 rotateRight(AVLTree,current)
112             else:
113                 rotateRight(AVLTree,current)
114         if current.bf<-1: #rotación a la izquierda
115             if current.rightnode!=None and current.rightnode.bf==1:
116                 rotateRight(AVLTree,current.rightnode)
117                 rotateLeft(AVLTree,current)
118             else:
119                 rotateLeft(AVLTree,current)
```

Ejercicio 4:

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

```
141 def insert(AVLTree,element,key):
142   #Inserta un elemento con una clave determinada en un AVLTree y rebalancea luego de
    insertar si es necesario
143   newNode=AVLNode( )
144   newNode.value=element
145   newNode.key=key
146   if AVLTree.root==None:
147     AVLTree.root=newNode
148   else:
149     keyInsert=None
150     keyInsert=insertNode(newNode,AVLTree.root,keyInsert)
151     reBalance(AVLTree)
152     return keyInsert
153 def insertNode(newNode,current,keyInsert):
154   if newNode.key>current.key:
155     if current.rightnode!=None:
156       insertNode(newNode,current.rightnode,keyInsert)
157     else:
158       current.rightnode=newNode
159       newNode.parent=current
160   elif newNode.key<current.key:
161     if current.leftnode!=None:
162       insertNode(newNode,current.leftnode,keyInsert)
163     else:
164       current.leftnode=newNode
165       newNode.parent=current
166   else:
167     return None
168   return newNode.key
```

Ejercicio 5:

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

```
170 def delete(AVLTree,element):
171   #Elimina un elemento de un árbol AVL
172   if AVLTree.root==None:
173     return None
174   elif AVLTree.root.rightnode==None and AVLTree.root.leftnode==None:
175     if AVLTree.root.value==element:
176       key=AVLTree.root.key
177       AVLTree.root=None
178       return key
179     else:
180       return None
181   else:
182     key=search(AVLTree,element)
183     if key!=None:
184       deletedKey=deleteKey(AVLTree,key)
185       return deletedKey
186     else:
187       return None
```

```
189 def deleteKey(AVLTree,key):
190   #Elimina una clave de un árbol AVL y rebalancea si es necesario
191   if AVLTree.root==None:
192     return None
193   elif AVLTree.root.rightnode==None and AVLTree.root.leftnode==None:
194     if AVLTree.root.key==key:
195       AVLTree.root=None
196       return key
197     else:
198       return None
199   else:
200     current=AVLTree.root
201     deletedKey=None
202     deletedKey=deleteNodebyKey(AVLTree,current,key,deletedKey)
203     return deletedKey
204 def deleteNodebyKey(AVLTree,current,key,deletedKey):
205   #Busco el nodo a borrar
206   if key>current.key:
207     if current.rightnode!=None:
208       deletedKey=deleteNodebyKey(AVLTree,current.rightnode,key,deletedKey)
209   elif key<current.key:
210     if current.leftnode!=None:
211       deletedKey=deleteNodebyKey(AVLTree,current.leftnode,key,deletedKey)
212   elif key==current.key:
213     deletedKey=current.key
214     #Caso 1: El nodo a borrar es la raiz
215     newNode=AVLNode()
216     newNode=current
217     if key==AVLTree.root.key:
```

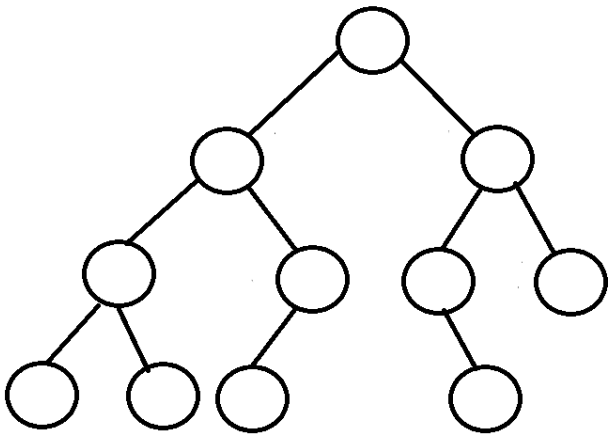
```
218         newNode=buscarMenordeMayores(current.rightnode)
219         newNode.leftnode=current.leftnode
220         newNode.rightnode=current.rightnode
221         AVLTree.root=newNode
222       #Caso 2: El nodo a borrar tiene un hijo a la izquierda
223       elif current.rightnode==None and current.leftnode!=None:
224         if current.parent.rightnode==current:
225           current.parent.rightnode=current.leftnode
226         else:
227           current.parent.leftnode=current.leftnode
228       #Caso 3: El nodo a borrar tiene un hijo a la derecha
229       elif current.rightnode!=None and current.leftnode==None:
230         if current.parent.rightnode==current:
231           current.parent.rightnode=current.rightnode
232         else:
233           current.parent.leftnode=current.rightnode
234       #Caso 4: El nodo a borrar tiene dos hijos
235       elif current.rightnode!=None and current.leftnode!=None:
236         newNode=buscarMenordeMayores(current.rightnode)
237         #Borrar nodo
238         newNode.leftnode=current.leftnode
239         newNode.rightnode=current.rightnode
240         if current.parent.rightnode==current:
241           current.parent.rightnode=newNode
242         else:
243           current.parent.leftnode=newNode
244       #Caso 5: El nodo a borrar no tiene hijos (es una hoja)
245       else:
246         if current.parent.rightnode==current:
247           current.parent.rightnode=None
248         else:
249           current.parent.leftnode=None
250     reBalance(AVLTree)
251     return deletedKey
253 def buscarMenordeMayores(newNode):
254     #Busca el Menor de los Mayores para reemplazar en el nodo a borrar
255     if newNode.leftnode!=None:
256       newNode=buscarMenordeMayores(newNode.leftnode)
257     #Llega al menor de los mayores
258     if newNode.leftnode==None:
259       #Borrar nodo de abajo
260       if newNode.parent.rightnode==newNode:
261         newNode.parent.rightnode=None
262       else:
263         newNode.parent.leftnode=None
264     return newNode
```

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
   b. __V__ Un AVL donde todos los nodos tengan factor de balance 0 es completo
   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.
   d. __V__ En todo *AVL* existe al menos un nodo con factor de balance 0.

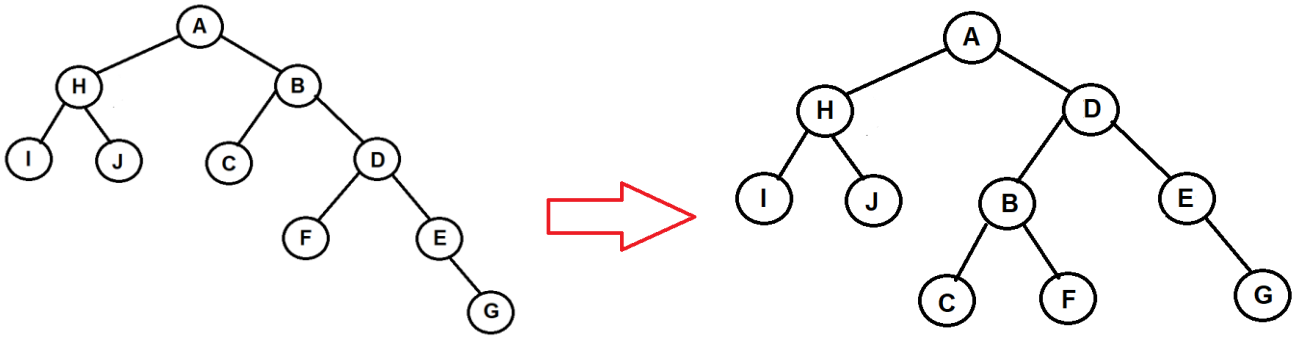
6) a) Contraejemplo:



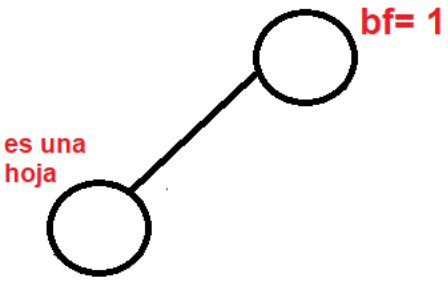
Es un árbol AVL y no está completo

b) Supongamos que existe un AVL que todos sus nodos tienen un balance factor igual a 0 y no es completo.
Si no es completo, existe al menos un nodo que tiene un solo hijo.
Esto es una contradicción, ya que si un nodo tiene solo un hijo, su balance factor va a ser distinto de 0.
Por lo tanto, es verdadero.

c) Contraejemplo:

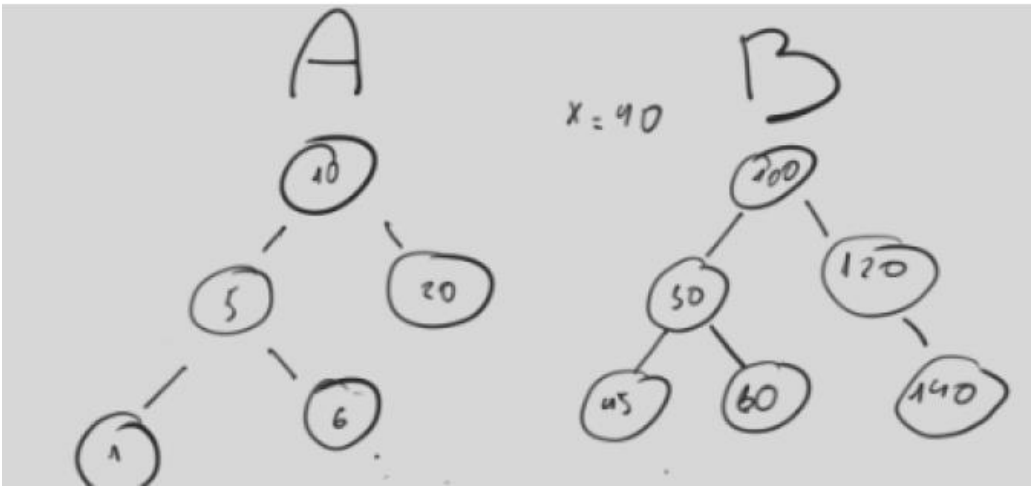


d) Siempre se va a cumplir esto, ya que las hojas tienen balance factor igual a 0.
Si no consideramos las hojas esto es falso, ya que, por ejemplo, tenemos este árbol que no lo cumpliría:

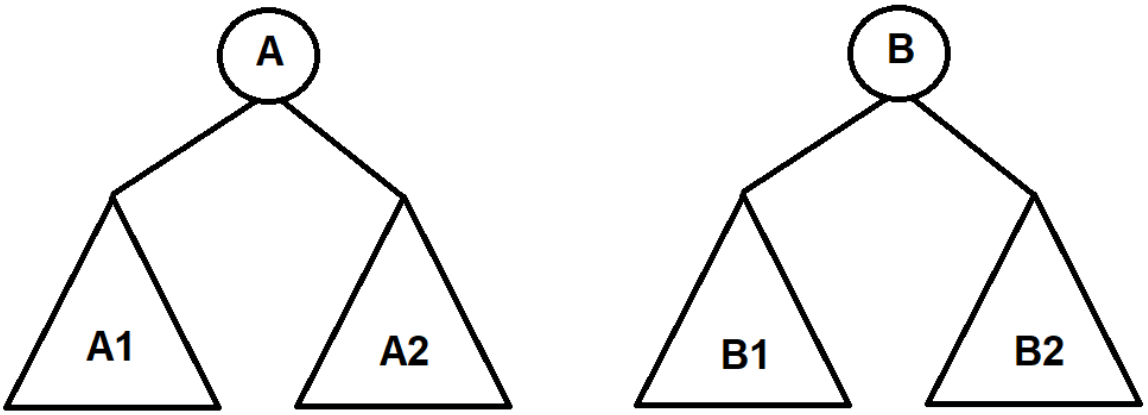


Ejercicio 7:

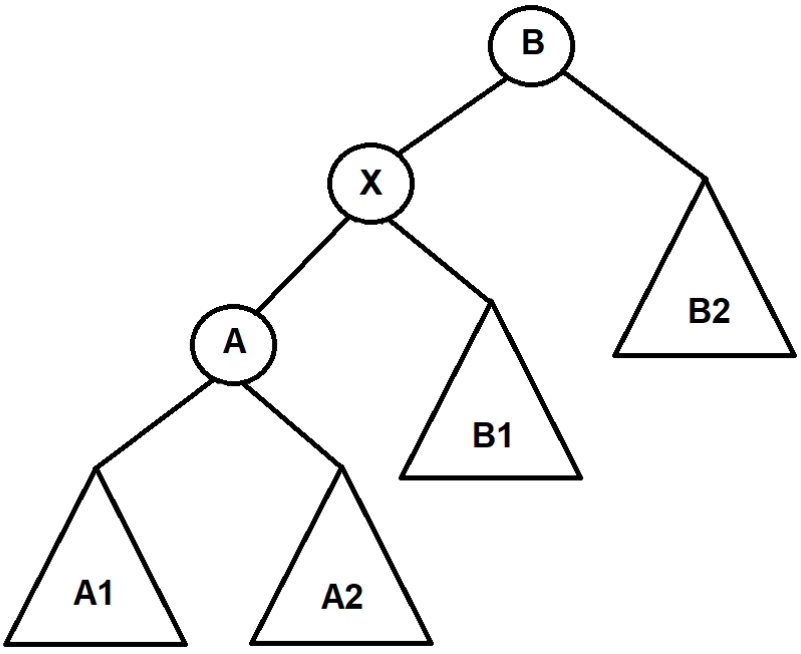
Sean *A* y *B* dos *AVL* de *m* y *n* nodos respectivamente y sea *x* un key cualquiera de forma tal que para todo key $a \in A$ y para todo key $b \in B$ se cumple que $a < x < b$. Plantear un algoritmo $O(\log n + \log m)$ que devuelva un *AVL* que contenga los key de *A*, el key *x* y los key de *B*.



A < X < B **O (log n + log m)**



=



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.

Cuando queremos buscar la rama truncada por el camino más corto, entonces consideremos que la raíz tiene un balance factor de -1, entonces el subárbol derecho tiene mayor altura, por lo tanto el camino más corto es por el subárbol izquierdo y viceversa si el balance factor es igual a 1.
Como solo tiene que recorrer un subárbol va a ser de longitud h/2.