

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

```
334 def rotateLeft(Tree,avlnode):
335     nuevaRaiz = avlnode.righnode
336     #si el nodo a rotar es la raiz
337     if avlnode == Tree.root:
338         #si la nueva raiz no tiene un hijo izquierdo
339         if nuevaRaiz.leftnode == None:
340             avlnode.righnode = None
341
342     else:
343         #si la nueva raiz tiene un hijo derecho
344         avlnode.righnode = nuevaRaiz.leftnode
345         nuevaRaiz.leftnode.parent = avlnode
346
347         nuevaRaiz.leftnode = avlnode
348         avlnode.parent = nuevaRaiz
349         Tree.root = nuevaRaiz
350         return nuevaRaiz
351     else:
352
353         if avlnode.parent.leftnode == avlnode:
354             avlnode.parent.leftnode = nuevaRaiz
355         else:
356             avlnode.parent.righnode = nuevaRaiz
357
358         avlnode.parent = nuevaRaiz
359         nuevaRaiz.parent = avlnode.parent
360
361         if nuevaRaiz.leftnode == None:
362             avlnode.righnode = None
363         else:
364             avlnode.righnode = nuevaRaiz.leftnode
365
366         nuevaRaiz.leftnode = avlnode
367
368     return nuevaRaiz
```

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

```
372 #EJERCICIO 1.2 - TP1 ARBOLES
373 #salida: retorna la nueva raiz del arbol
374
375 def rotateRight(Tree, avlnode):
376     nuevaRaiz = avlnode.leftnode
377     #El nodo a rotar es la raiz
378     if avlnode == Tree.root:
379         #si la nueva raiz tiene un hijo derecho
380         if avlnode.leftnode.rightnode != None:
381             #nuevaRaiz = avlnode.leftnode
382             avlnode.leftnode = nuevaRaiz.rightnode
383             nuevaRaiz.rightnode.parent = avlnode
384         else:
385             #si la nueva raiz no tiene un hijo derecho
386             avlnode.leftnode = None
387
388             nuevaRaiz.rightnode = avlnode
389             avlnode.parent = nuevaRaiz
390             Tree.root = nuevaRaiz
391             return nuevaRaiz
392
393     else:
394         #si el nodo a rotar no es la raiz
395         if avlnode.parent.leftnode == avlnode:
396             #el hijo izquierdo del padre de avlnode (avlnode.parent.leftnode)
397             avlnode.parent.leftnode = nuevaRaiz
398         else:
399             avlnode.parent.rightnode = nuevaRaiz
400             avlnode.parent = avlnode.leftnode
401             #si la nueva raiz tiene un hijo derecho
402             if avlnode.leftnode.rightnode == None:
403                 avlnode.leftnode.rightnode = avlnode
404                 avlnode.leftnode = None #importante, sino no se hace bien el arbol
405                 return nuevaRaiz
406             else:
407                 #si la nueva raiz no tiene un hijo derecho
408                 avlnode.leftnode = avlnode.leftnode.rightnode
409                 avlnode.leftnode.parent.rightnode = avlnode
410                 return nuevaRaiz
```

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

```
411 |
412 | #-----
413 | #EJERCICIO 2 - TP1 ARBOLES
414 | #Implementar una función recursiva que calcule el elemento balanceFactor de cada subárbol
415 | #Salida: AVL con el valor de balanceFactor para cada subarbol
416 |
417 | def altura(node):
418 |     #complejidad: O(n)
419 |     if node == None:
420 |         return 0
421 |     else:
422 |         return 1 + max(altura(node.leftnode), altura(node.rightnode))
423 |
424 | def calculateBalance(AVL):
425 |     #complejidad: O(n^2) porque calculateBalanceR pasa por cada nodo y le calcula el bf (bf = h(hijo
426 |     #izq) - h(hijo der). La función altura (h) tiene una complejidad de O(n) porque tiene que recorrer
427 |     #cada subarbol para ver cuantos nodos tiene a derecha y a izquierda
428 |     if AVL.root == None:
429 |         return
430 |     else:
431 |         calculateBalanceR(AVL.root)
432 |         return AVL
433 |
434 | def calculateBalanceR(node):
435 |     if node == None:
436 |         return
437 |     else:
438 |         node.bf = altura(node.leftnode) - altura(node.rightnode)
439 |         calculateBalanceR(node.leftnode)
440 |         calculateBalanceR(node.rightnode)
441 |         return node.bf
```

Ejercicio 3

Implementar una función en el modulo `avltree.py` de acuerdo a las siguientes especificaciones:

`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.

```
445 def rebalance(AVLTree):
446     calculateBalance(AVLTree)
447     if AVLTree.root == None:
448         return
449     else:
450         recorreAbajoR(AVLTree, AVLTree.root)
451     return AVLTree
452
453 def recorreAbajoR(T, node):
454     if node == None:
455         return
456     else:
457         if node.bf < -1:
458             recorreAbajoR(T, node.rightnode)
459         else:
460             if node.bf > 1:
461                 recorreAbajoR(T, node.leftnode)
462             else:
463                 rebalanceAux(T, node.parent)
464
465 def recorreArribaR(T, node):
466     if node == None:
467         return T
468     else:
469         if node.bf < 1 and node.bf > -1:
470             recorreArribaR(T, node.parent)
471         else:
472             rebalanceAux(T, node)
473
```

```
474 def rebalanceAux(T, node):
475     if node == None:
476         return
477     else:
478         if node.bf < -1:
479             if node.rightnode.leftnode != None:
480                 #si el hijo derecho tiene un hijo izquierdo
481                 #hago rotacion a la derecha del hijoderecho
482                 rotateRight(T, node.rightnode)
483                 #hago rotacin a la izquierda del nodo
484                 rotateLeft(T, node)
485             else:
486                 rotateLeft(T, node)
487             calculateBalance(T)
488         else:
489             if node.bf > 1:
490                 if node.leftnode.rightnode != None:
491                     rotateLeft(T, node.leftnode)
492                     rotateRight(T, node)
493                 else:
494                     rotateRight(T, node)
495                 calculateBalance(T)
496             else:
497                 recorreArribaR(T, node)
498
```

Ejercicio 4:

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

```
501
502 def insertAvlnode(T, element, key):
503     insert(T, element, key)
504     rebalance(T)
505
```

Ejercicio 5:

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

```
508
509 def deleteAvlnode(AVLTree,element):
510     rebalance(AVLTree)
511     delete(AVLTree,element)
512
```

Parte 2

Ejercicio 6:

1. Responder V o F y justificar su respuesta:

- ___ En un AVL el penúltimo nivel tiene que estar completo
- ___ Un AVL donde todos los nodos tengan factor de balance 0 es completo
- ___ 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.
- ___ En todo AVL existe al menos un nodo con factor de balance 0.

a) Falso

Es un AVL y el penúltimo nivel no está completo

Un AVL es completo cuando todos sus niveles tienen 2 hijos

b) Verdadero

todos los nodos tienen bf 0
→ Es un AVL completo
Si cada nodo tiene dos hijos entonces sus bf = 0

c) Falso

Inserto 75

El nodo padre (60) no se desbalanceó → pero hay que seguir verificando hacia arriba porque quedaron los nodos 50 y 70 desbalanceados

d) Verdadero

Los hijos del árbol siempre tienen bf = 0.

Ejercicio 7:

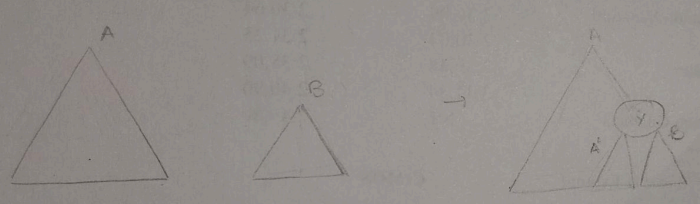
Sean A y B dos AVL de m y n nodos respectivamente y sea x un key cualquiera de forma tal que para todo key $a \in A$ y para todo key $b \in B$ se cumple que $a < x < b$. Plantear un algoritmo $O(\log n + \log m)$ que devuelva un AVL que contenga los key de A , el key x y los key de B .

Ejercicio 7:

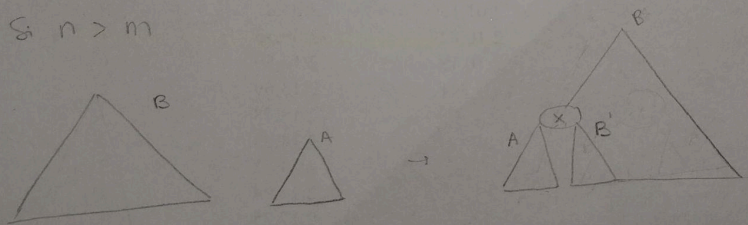
A es un AVL de m nodos, a es un key $\forall a \in A$
 B es un AVL de n nodos, b es un key $\forall b \in B$
 x es un key,
 $a < x < b$

$O(\log n + \log m)$

• Si $m > n$:



• Si $n > m$:

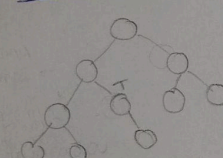


Ejercicio 8:

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.

Ejercicio 8:



La rama truncada es T , la cual máximo puede tener un hijo el cual es una hoja, entonces $T+1$ es una hoja y por ser una hoja nos permite obtener la información sobre la altura del árbol, la cual va a diferir en 1 de la altura del nodo hoja por lo cual al hacer $\frac{h+1}{2} = h_T$