

Universidad Nacional de Cuyo  
Facultad de Ingeniería  
Licenciatura en Ciencias de la Computación

---

# TRABAJO PRÁCTICO N<sup>o</sup> 2

ALGORITMOS Y ESTRUCTURAS DE DATOS

ÁRBOLES BALANCEADOS: AVL

2024

---

Gonzalo Padilla

Marzo 2024



## Parte 1

**Importante:** Los ejercicios de esta primera parte tienen como objetivo codificar las diferentes funciones básicas necesarias para implementar un árbol AVL.

A partir de estructuras definidas como:

```
class AVLTree:
    root = None

class AVLNode:
    parent = None
    leftnode = None
    rightnode = None
    key = None
    value = None
    bf = None
```

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

### Solución

Ver 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

## Solución

```
0 def rotateLeft(t, node):
1     # Caso especial donde debe realizarse una rotación previa
2     if node.rightrightnode.bf > 0:
3         rotateRight(t, node.rightrightnode)
4
5     a = node # raíz actual
6     b = node.rightrightnode # nueva raíz
7
8     # Convertir a 'b' en la nueva raíz
9     a.rightrightnode = None
10    b.parent = a.parent
11    if b.parent:
12        if b.key > b.parent.key:
13            b.parent.rightrightnode = b
14        else:
15            b.parent.leftnode = b
16    else:
17        t.root = b
18
19    # Si 'b' tenía hijo izquierdo, pasa a ser el
20    # hijo derecho de 'a'
21    if b.leftnode:
22        a.rightrightnode = b.leftnode
23        b.leftnode.parent = a
24        b.leftnode = None
25
26    # Nodo 'a' para a ser hijo izquierdo de 'b'
27    b.leftnode = a
28    a.parent = b
29
30    return b # Retornamos la nueva raíz
```

## 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

## Solución

```
0 def calculateBalance(AVLTree):
1     if not AVLTree:
2         return None
3
4     # Calcula el balance factor de cada nodo, calculando
5     # previamente la altura de cada sub árbol. Orden O(n)
6     def calculateNodeBalance(node):
7         if not node:
8             return 0
9
10        leftHeight = calculateNodeBalance(node.leftnode)
11        rightHeight = calculateNodeBalance(node.rightrightnode)
12        node.bf = leftHeight - rightHeight
13        return max(leftHeight, rightHeight) + 1
14
15    calculateNodeBalance(AVLTree.root)
16    return AVLTree # Retornamos el árbol
```

## 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.

## Solución

```
0  # Balancea un nodo y sus hijos
1  # Retorna True si ocurrió un balanceo de forma exitosa
2  # Si no ocurrieron, o fallaron, retorna False
3  def reBalanceNode(t, node):
4      if not node:
5          return False
6
7      # Balanceamos los subárboles del nodo
8      # Si un nodo inferior fue balanceado, por propiedades
9      # de los AVL ya no es necesario balancear este nodo
10     wasReBalanced = reBalanceNode(t, node.leftnode) or reBalanceNode(t, node.rightnode)
11
12     if wasReBalanced:
13         return True
14
15     if abs(node.bf) < 2: # Nodo balanceado
16         return False
17
18     if node.bf == 2:
19         return rotateRight(t, node) # Balanceamos
20     elif node.bf == -2:
21         return rotateLeft(t, node) # Balanceamos
22     elif abs(node.bf) > 2:
23         print(f'Node con key = {node.key} tiene un balance factor = {node.bg}
24               ↳ incorregible!')
25         return False # Algo salió muy mal
26
27 def reBalance(t):
28     if not t:
29         return None
30
31     calculateBalance(t) # Calculamos los balances por si no existen o están
32                         ↳ desactualizados
33     reBalanceNode(t, t.root) # Balanceamos
34     calculateBalance(t) # Recalculamos los balances
35     return t
```

## Ejercicio 4

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

## Solución

```
0 def _insertNode(currentNode, newNode):
1     if newNode.key < currentNode.key: # Insertamos a la izquierda
2         if not currentNode.leftnode: # Si no hay hijo, pasa a ser el nuevo hijo
3             currentNode.leftnode = newNode
4             newNode.parent = currentNode
5             return newNode.key
6         else: # Si lo hay, llamamos a la recursión
7             return _insertNode(currentNode.leftnode, newNode)
8     elif newNode.key > currentNode.key: # Insertamos a la derecha
9         if not currentNode.rightnode: # Si no hay hijo, pasa a ser el nuevo hijo
10            currentNode.rightnode = newNode
11            newNode.parent = currentNode
12            return newNode.key
13        else: # Si lo hay, llamamos a la recursión
14            return _insertNode(currentNode.rightnode, newNode)
15    else:
16        print("Error! Ya existe un elemento para la key indicada!")
17        return None # Si ya existe la key en el árbol, devolvemos None
18
19 def insert(t, element, key):
20     if not t:
21         return None
22
23     # Creamos el nuevo nodo
24     newNode = AVLNode()
25     newNode.key = key
26     newNode.value = element
27     newNode.bf = 0
28
29     # Si no hay raíz, pasa a ser la raíz
30     if not t.root:
31         t.root = newNode
32         return key
33
34     key = _insertNode(t.root, newNode) # Si no, insertamos
35     reBalance(t) # Rebalanceamos
36     return key # Retornamos la key si fue exitoso (puede ser None)
```

## Ejercicio 5

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

## Solución

```
0 def _deleteNode(B, node):
1     if not node:
2         return None
3
4     # El nodo a eliminar es reemplazado por el más pequeño de su subárbol derecho
5     # o el más grande de su subárbol izquierdo para mantener el orden
6     # _deleteNode lo elimina de su posición y nos lo devuelve
7     newNode = _deleteNode(B, _findSmallest(node.rightrightnode) or
8         ↪ _findLargest(node.leftnode))
9
10    # Si existe un reemplazo, reemplazamos
11    if newNode:
12        newNode.leftnode = node.leftnode
13        newNode.rightrightnode = node.rightrightnode
14        if newNode.leftnode:
15            newNode.leftnode.parent = newNode
16        if newNode.rightrightnode:
17            newNode.rightrightnode.parent = newNode
18        newNode.parent = node.parent
19
20    # Apuntamos el padre al nuevo nodo
21    if not node.parent:
22        B.root = newNode
23    elif node.parent.leftnode == node:
24        node.parent.leftnode = newNode
25    else:
26        node.parent.rightrightnode = newNode
27
28    return node # Retornamos el nodo eliminado
29
30 def delete(B, element):
31     node = _deleteNode(B, _findNodeByValue(B.root, element)) # Buscamos el nodo y lo
32     ↪ eliminamos
33     reBalance(B) # Rebalanceamos el árbol
34     return node.key if node else None # Retornamos la key
```

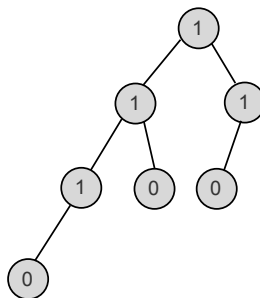
## Parte 2

### Ejercicio 6

1. Responder V o F y justificar su respuesta:

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

El siguiente contraejemplo demuestra que la proposición es falsa.



Como puede verse, es un AVL ya que el balance factor de cada nodo es -1, 0 o 1, pero su penúltimo nivel está incompleto.

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

Un árbol es completo si todos los nodos de todos sus niveles, excepto el último, tienen 2 hijos. Como todos los nodos del árbol tienen balance factor 0, entonces cada uno debe tener 0 hijos, o tener 2 hijos que formen subárboles de la misma altura.

Sea  $h$  la altura del árbol, analicemos cada nivel  $l$ . En el primer nivel ( $l = 0$ ), si  $h = l = 0$ , entonces la raíz deberá tener necesariamente 0 hijos. En tal caso, se tiene un árbol completo, ya que los nodos de su último (y único) nivel tienen 0 hijos. En cambio, si  $l \neq h$ , entonces necesariamente deberá tener algún hijo. Pero como ya vimos, solo puede tener 0 o 2 hijos, por lo que debe tener 2. Además, los subárboles que formen sus hijos deberán tener ambos la misma altura  $h - l - 1$ .

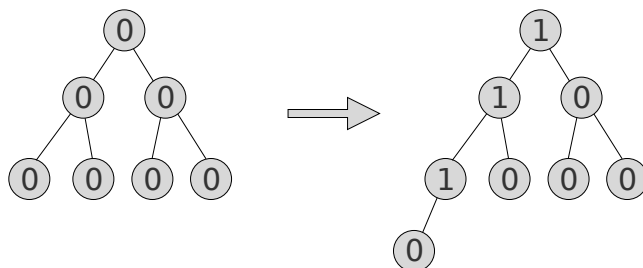
Ahora analizamos el siguiente nivel ( $l = 1$ ). Se tienen dos posibles casos.

- 1) ( $l = h$ ) En tal caso, estamos en el último nivel, y por lo tanto cada nodo del mismo debe tener 0 hijos.
- 2) ( $l < h$ ) En este caso, algún nodo debe tener 2 hijos. Pero como dijimos que cada subárbol de este nivel debe tener la misma altura ( $h - l$ ), entonces *todos* los nodos de este nivel tienen 2 hijos. Como cada nodo tiene balance factor 0, entonces sus hijos deben también formar subárboles de la misma altura  $h - l - 1$ .

Y así para cada  $l = 1, 2, \dots, h$ . Entonces, puede verse que para todos los niveles, excepto el último, cada nodo tiene 2 hijos, entonces se tiene un árbol 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.

Se demuestra mediante el siguiente contraejemplo



Al insertar un nodo a la izquierda del árbol, su padre no se desbalanceó (es -1, 0 o 1). Aún así, cambiaron los balance factor de todos los nodos superiores.

d) V En todo AVL existe al menos un nodo con factor de balance 0.

Todas las hojas tienen balance factor 0, por lo que es verdadero. Si no se toman en cuenta las mismas, entonces la proposición es falsa. Basta tomar un árbol que solo tiene dos nodos: la raíz y un solo hijo. En este caso el hijo es un nodo hoja con balance factor 0, y la raíz, que es el único nodo interior, tiene balance factor 1, entonces el árbol no tiene ningún nodo interior con balance factor 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  $B$ .

### Solución

Se calculan las alturas  $h_A$  y  $h_B$  de ambos árboles (operaciones de orden  $O(\log m)$  y  $O(\log n)$ , respectivamente). Si  $|h_A - h_B| < 2$  entonces simplemente basta conectar el árbol  $A$  como hijo izquierdo de  $x$ , y a  $B$  como su hijo derecho, y se tendrá un árbol AVL con  $x$  como su raíz. En cambio, si  $|h_A - h_B| \geq 2$ , entonces insertaremos el subárbol de menor altura en el de mayor altura de la siguiente manera:

1. Recorremos el árbol de mayor altura por su derecha si se trata de  $A$ , o por su izquierda si es  $B$ , hasta haber recorrido una distancia igual a la diferencia entre la altura del árbol más alto y la altura del más corto. Es decir, una vez que nos detenemos, estaremos en la raíz de un subárbol con altura igual a la del árbol más corto.
2. A tal nodo, lo reemplazamos por  $x$ , colocando al mismo como hijo de  $x$ . Si estamos en  $A$ , como hijo izquierdo, si no, como hijo derecho. Esto preserva el orden de todas las llaves, pero ahora  $x$  es un nodo muy desbalanceado.
3. Finalmente, colocamos al árbol más corto como el hijo restante de  $x$ . Esto balancea al árbol, ya que debajo de  $x$ , el subárbol que ahora es hermano del árbol más corto, tenía una altura igual a él, ya que recorrimos el árbol originalmente hasta esa distancia. También se preserva el orden de las keys.

### 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.

### Solución

Encontrar la longitud mínima que puede tener una rama truncada en un AVL de altura  $h$  es equivalente a encontrar la mínima cantidad de niveles completos del mismo. Esto es cierto ya que si el siguiente nivel está completo, entonces todos los nodos del nivel actual tienen 2 hijos (entonces ninguno es el final de una rama truncada), y si el nivel actual no es completo, entonces existe uno con menos de 2 hijos en el nivel anterior (entonces existe una rama truncada de menor longitud al nivel actual).

Podemos encontrar el nivel mínimo completo de un árbol siguiendo los recorridos de menor longitud posible. Si el árbol es de altura  $h$ , entonces existe un recorrido desde la raíz hasta algún nodo del último nivel, que contiene  $h$  aristas y  $h + 1$  nodos. Para mantener el balance de la raíz, debe existir otro recorrido, que pase por el otro hijo de la raíz, y que tenga altura  $h$  o  $h - 1$ . Como nos interesa encontrar el recorrido de menor longitud posible, nos concentramos en el caso en el que tiene longitud  $h - 1$ .

De igual forma, desde la perspectiva del hijo de la raíz contenido en este segundo recorrido (en el nivel  $n = 1$ ), tal recorrido tiene un altura  $h - 2$  (comenzando desde el nivel 1), por lo que también debe ser el comienzo de otro recorrido de  $h - 3$  aristas, que pase por su otro hijo, para mantener su propio balance. En general, en este mínimo recorrido, el nodo del nivel  $l$  forma parte de un recorrido que comienza en él, con altura  $h - 2l$  a partir de sí mismo, y debe a su vez ser el comienzo de un recorrido de altura  $h - 2l - 1$ .



Nótese que ambos números pueden ser cero o negativos. Cuando el primero sea cero, significa que el nodo de ese nivel es el final del recorrido, y no existe recorrido menor, entonces estamos en el mínimo nivel completo, y el nodo no tiene hijos (es una hoja). Si, en cambio, el segundo número es 0, y el primero positivo, quiere decir que para mantener al nodo balanceado, uno de sus hijos puede no existir (pero no ninguno), entonces hemos encontrado también el mínimo nivel completo y, por lo tanto, la longitud mínima de una rama truncada.

Finalmente, tenemos que  $h - 2l = 0$  si  $l = \frac{h}{2}$ , y que  $h - 2l - 1 = 0$  si  $l = \frac{h-1}{2}$ . Si  $h$  es par, solo la primera ecuación tiene solución entera, por lo que utilizamos la fórmula correspondiente. De la misma forma, si  $h$  es impar, solo la segunda tiene solución entera. La fórmula  $l = \lfloor h/2 \rfloor$  logra que alguno de los dos números sea 0, y por lo tanto indica la altura mínima de una rama truncada, que es lo que se quería probar.

## Parte 3

### Ejercicios Opcionales

1. Si  $n$  es la cantidad de nodos en un árbol AVL, implemente la operación **height()** en el módulo **avltree.py** que determine su altura en  $O(\log n)$ . Justifique el por qué de dicho orden.

#### Solución

```
0 def height(t):
1     if t is None:
2         return None
3
4     def nodeHeight(node):
5         if node is None:
6             return 0
7
8         # Utilizando el balance factor, elegimos el subárbol
9         # de mayor altura y seguimos la recursión por ahí.
10        longest = node.rightrightnode if node.bf < 0 else node.leftnode
11
12        # Retornamos la altura del subárbol más alto + 1
13        return nodeHeight(longest) + 1
14
15    return nodeHeight(t.root) - 1 if t.root else 0
```

2. Considere una modificación en el módulo **avltree.py** donde a cada nodo se le ha agregado el campo **count** que almacena el número de nodos que hay en el subárbol en el que él es raíz. Programe un algoritmo  $O(\log n)$  que determine la cantidad de nodos en el árbol cuyo valor del key se encuentra en un intervalo  $[a, b]$  dado como parámetro. Explique brevemente por qué el algoritmo programado por usted tiene dicho orden.