Trabajo Práctico Nº 1

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

```python
def rotateLeft(t, node):
    # (raíz) A -> B -> C

    # Caso especial:
    if node.rightnode.bf > 0:
        rotateRight(t, node.rightnode)

    a = node
    b = node.rightnode

    # Nodo B es la nueva raíz
    a.rightnode = None
    b.parent = a.parent
    if b.parent:
        if b.key > b.parent.key:
            b.parent.rightnode = b
        else:
            b.parent.leftnode = b
    else:
        t.root = b

    # Si B tenía hijo izquierdo, pasa a ser el
    # hijo derecho de A
    if b.leftnode:
        a.rightnode = b.leftnode
        b.leftnode.parent = a
        b.leftnode = None

    # Nodo A es el hijo izquierdo de B
    b.leftnode = a
    a.parent = b

    return b
```

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

```python
def calculateBalance(AVLTree):
    if not AVLTree:
        return None

    def calculateNodeBalance(node):
        if not node:
            return 0

        leftHeight = calculateNodeBalance(node.leftnode)
        rightHeight = calculateNodeBalance(node.rightnode)
        node.bf = leftHeight - rightHeight
        return max(leftHeight, rightHeight) + 1

    calculateNodeBalance(AVLTree.root)
    return AVLTree
```

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.

Solución

```
0  def reBalanceNode(t, node):
1      if not node:
2          return False
3
4      wasReBalanced = reBalanceNode(t, node.leftnode) or reBalanceNode(t, node.rightnode)
5
6      if wasReBalanced:
7          return True
8
9      if abs(node.bf) < 2:
10         return False
11
12     if node.bf == 2:
13         rotateRight(t, node)
14     elif node.bf == -2:
15         rotateLeft(t, node)
16     elif abs(node.bf) > 2:
17         print(f'Node con key = {node.key} tiene un balance factor = {node.bg}
           ↪  incorregible!')
18
19     return True
20
21 def reBalance(t):
22     if not t:
23         return None
24
25     calculateBalance(t)
26     reBalanceNode(t, t.root)
27     calculateBalance(t)
28     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:
2          if not currentNode.leftnode:
3              currentNode.leftnode = newNode
4              newNode.parent = currentNode
5              return newNode.key
6          else:
7              return _insertNode(currentNode.leftnode, newNode)
8      elif newNode.key > currentNode.key:
9          if not currentNode.rightnode:
10             currentNode.rightnode = newNode
11             newNode.parent = currentNode
12             return newNode.key
13         else:
14             return _insertNode(currentNode.rightnode, newNode)
15     else:
16         print("Error! Ya existe un elemento para la key indicada!")
17         return None
18
19 def insert(t, element, key):
20     if not t:
21         return None
22
23     newNode = AVLNode()
24     newNode.key = key
25     newNode.value = element
26     newNode.bf = 0
27
28     if not t.root:
29         t.root = newNode
30         return key
31
32     result = _insertNode(t.root, newNode)
33     reBalance(t)
34     return result
```

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      newNode = _deleteNode(B, _findSmallest(node.rightnode) or
       ↪  _findLargest(node.leftnode))
5
6      if newNode:
7          newNode.leftnode = node.leftnode
8          newNode.rightnode = node.rightnode
9          if newNode.leftnode:
10             newNode.leftnode.parent = newNode
11         if newNode.rightnode:
12             newNode.rightnode.parent = newNode
13         newNode.parent = node.parent
14
15     if not node.parent:
16         B.root = newNode
17     elif node.parent.leftnode == node:
18         node.parent.leftnode = newNode
19     else:
20         node.parent.rightnode = newNode
21
22     return node
```

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) _V_ 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.

Solución

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

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

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

```python
def height(t):
    if t is None:
        return None

    def nodeHeight(node):
        if node is None:
            return 0

        longest = node.rightnode if node.bf < 0 else node.leftnode
        return nodeHeight(longest) + 1

    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.

   Solución