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

```python
def reBalanceNode(t, node):
    if not node:
        return False

    wasReBalanced = reBalanceNode(t, node.leftnode) or reBalanceNode(t, node.rightnode)

    if wasReBalanced:
        return True

    if abs(node.bf) < 2:
        return False

    if node.bf == 2:
        rotateRight(t, node)
    elif node.bf == -2:
        rotateLeft(t, node)
    elif abs(node.bf) > 2:
        print(f'Node con key = {node.key} tiene un balance factor = {node.bg}
        ↪  incorregible!')

    return True

def reBalance(t):
    if not t:
        return None

    calculateBalance(t)
    reBalanceNode(t, t.root)
    calculateBalance(t)
    return t
```

Ejercicio 4

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

Solución

```
def _insertNode(currentNode, newNode):
    if newNode.key < currentNode.key:
        if not currentNode.leftnode:
            currentNode.leftnode = newNode
            newNode.parent = currentNode
            return newNode.key
        else:
            return _insertNode(currentNode.leftnode, newNode)
    elif newNode.key > currentNode.key:
        if not currentNode.rightnode:
            currentNode.rightnode = newNode
            newNode.parent = currentNode
            return newNode.key
        else:
            return _insertNode(currentNode.rightnode, newNode)
    else:
        print("Error! Ya existe un elemento para la key indicada!")
        return None

def insert(t, element, key):
    if not t:
        return None

    newNode = AVLNode()
    newNode.key = key
    newNode.value = element
    newNode.bf = 0

    if not t.root:
        t.root = newNode
        return key

    result = _insertNode(t.root, newNode)
    reBalance(t)
    return result
```

Ejercicio 5

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

Solución

```
def _deleteNode(B, node):
    if not node:
        return None

    newNode = _deleteNode(B, _findSmallest(node.rightnode) or
    ↪   _findLargest(node.leftnode))

    if newNode:
        newNode.leftnode = node.leftnode
        newNode.rightnode = node.rightnode
        if newNode.leftnode:
            newNode.leftnode.parent = newNode
        if newNode.rightnode:
            newNode.rightnode.parent = newNode
        newNode.parent = node.parent

    if not node.parent:
        B.root = newNode
    elif node.parent.leftnode == node:
        node.parent.leftnode = newNode
    else:
        node.parent.rightnode = newNode

    return node
```

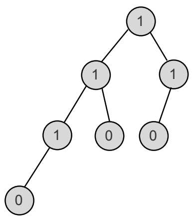
Parte 2

Ejercicio 6

1. Responder V o F y justificar su respuesta:

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

   El siguiente contra ejemplo 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) 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.

Si el balance factor del padre del nodo insertado no se desbalanceó, esto quiere decir que el padre tenía previamente un solo hijo y, por lo tanto, un balance factor de -1 o 1, que pasó luego a 0. Si ya tenía 1 hijo, entonces al agregarle otro a su lado, no cambia la altura del subárbol formado a partir del nodo padre. Por lo tanto, no cambia la altura de ningún subárbol formado a partir de sus nodos superiores, y no es necesario actualizar sus balance factors.

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

Todos los nodos terminales tienen balance factor 0, por lo que es verdadero. Si no se toman en cuenta los mismos, 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 terminal con balance factor 0, y la raíz, que es el único nodo interior, tiene balance factor 1, entonces el árbol no tiene nodos 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

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

```
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