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

```
def rotateLeft(t, node):
    # Caso especial donde debe realizarse una rotación previa
    if node.rightnode.bf > 0:
        rotateRight(t, node.rightnode)

    a = node # raíz actual
    b = node.rightnode # nueva raíz

    # Convertir a 'b' en 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' para a ser hijo izquierdo de 'b'
    b.leftnode = a
    a.parent = b

    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

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

    # Calcula el balance factor de cada nodo, calculando
    # previamente la altura de cada sub árbol. Orden O(n)
    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 # Retornamos el árbol
```

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
# Balancea un nodo y sus hijos
# Retorna True si ocurrió un balanceo de forma exitosa
# Si no ocurrieron, o fallaron, retorna False
def reBalanceNode(t, node):
    if not node:
        return False

    # Balanceamos los subárboles del nodo
    # Si un nodo inferior fue balanceado, por propiedades
    # de los AVL ya no es necesario balancear este nodo
    wasReBalanced = reBalanceNode(t, node.leftnode) or reBalanceNode(t, node.rightnode)

    if wasReBalanced:
        return True

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

    if node.bf == 2:
        return rotateRight(t, node) # Balanceamos
    elif node.bf == -2:
        return rotateLeft(t, node) # Balanceamos
    elif abs(node.bf) > 2:
        print(f'Node con key = {node.key} tiene un balance factor = {node.bg} incorregible!')
        return False # Algo salió muy mal

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

    calculateBalance(t) # Calculamos los balances por si no existen o están desactualizados
    reBalanceNode(t, t.root) # Balanceamos
    calculateBalance(t) # Recalculamos los balances
    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: # Insertamos a la izquierda
        if not currentNode.leftnode: # Si no hay hijo, pasa a ser el nuevo hijo
            currentNode.leftnode = newNode
            newNode.parent = currentNode
            return newNode.key
        else: # Si lo hay, llamamos a la recursión
            return _insertNode(currentNode.leftnode, newNode)
    elif newNode.key > currentNode.key: # Insertamos a la derecha
        if not currentNode.rightnode: # Si no hay hijo, pasa a ser el nuevo hijo
            currentNode.rightnode = newNode
            newNode.parent = currentNode
            return newNode.key
        else: # Si lo hay, llamamos a la recursión
            return _insertNode(currentNode.rightnode, newNode)
    else:
        print("Error! Ya existe un elemento para la key indicada!")
        return None # Si ya existe la key en el árbol, devolvemos None

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

    # Creamos el nuevo nodo
    newNode = AVLNode()
    newNode.key = key
    newNode.value = element
    newNode.bf = 0

    # Si no hay raíz, pasa a ser la raíz
    if not t.root:
        t.root = newNode
        return key

    key = _insertNode(t.root, newNode) # Si no, insertamos
    reBalance(t) # Rebalanceamos
    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

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

    # El nodo a eliminar es reemplazado por el más pequeño de su subárbol derecho
    # o el más grande de su subárbol izquierdo para mantener el orden
    # _delteNode lo elimina de su posición y nos lo devuelve
    newNode = _deleteNode(B, _findSmallest(node.rightnode) or
    ↪  _findLargest(node.leftnode))

    # Si existe un reemplazo, reemplazamos
    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

    # Apuntamos el padre al nuevo nodo
    if not node.parent:
        B.root = newNode
    elif node.parent.leftnode == node:
        node.parent.leftnode = newNode
    else:
        node.parent.rightnode = newNode

    return node # Retornamos el nodo eliminado

def delete(B, element):
    node = _deleteNode(B, _findNodeByValue(B.root, element)) # Buscamos el nodo y lo
    ↪  eliminamos
    reBalance(B) # Rebalanceamos el árbol
    return node.key if node else None # Retornamos la key
```

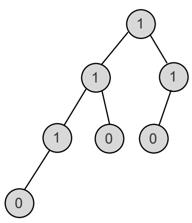
Parte 2

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 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

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 si 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

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

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

        # Utilizando el balance factor, elegimos el subárbol
        # de mayor altura y seguimos la recursión por ahí.
        longest = node.rightnode if node.bf < 0 else node.leftnode

        # Retornamos la altura del subárbol más alto + 1
        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.