Franco Yudica 13922

Trabajo práctico árboles AVL

Franco Yudica 13922

Parte 1, Implementación de AVL

Hay ciertas diferencias respecto a las especificaciones del trabajo práctico. Implementé el código dentro de una clase de python llamada AVL Tree, espero que se entienda el motivo, pues me resulta más práctico para cuando requiera usar este tipo de árbol.
Es por eso que en el código se va a observar la variable self, que es equivalente a tree

Empezando por rotateLeft y rotateRight

Implementación rotateLeft

```
newRoot = rotNode.rightnode

rotNode.rightnode = newRoot.leftnode

# Le copia el leftnode y actualiza su parent
if newRoot.leftnode is not None:
    newRoot.leftnode.parent = rotNode

newRoot.parent = rotNode.parent

if rotNode.parent is None:
    # En este caso es la raíz del árbol
    self.root = newRoot
else:
    # En este caso, rotNode puede ser el hijo derecho o izquierdo de su parent
    # en cualquier caso, actualizamos su referencia a newRoot
    if rotNode.parent.rightnode == rotNode:
        rotNode.parent.rightnode = newRoot
    else:
        rotNode.parent.leftnode = newRoot

newRoot.leftnode = rotNode
rotNode.parent = newRoot

# El orden en que se actualizan los nodos importa, pues rotnNode es el hijo
# de newRoot. Y para que newRoot tenga un bf correcto, la altura de rotNode
# debe estar actualizada
self._update_node(rotNode)
self._update_node(newRoot)
```

Franco Yudica 13922

Implementación rotate Right

```
# 1) Nueva raíz
newRoot = rotNode.leftnode

# 2) Cambia a newRoot por su hijo derecho
rotNode.leftnode = newRoot.rightnode

# Y actualiza el parent del hijo copiado
if newRoot.rightnode is not None:
    newRoot.rightnode.parent = rotNode

# Le copia el parent
newRoot.parent = rotNode.parent

# Ahora hago checkeos sobre el parent
# 1. Si rotNode es la raíz del árbol
if rotNode.parent is None:
    self.root = newRoot

# En este caso no es la raíz, por lo tanto hay que actualizar
# la referencia del parent a newRoot
else:
    if rotNode.parent.rightnode == rotNode:
        rotNode.parent.rightnode = newRoot
    else:
        rotNode.parent.leftnode = newRoot

newRoot.rightnode = rotNode
rotNode.parent = newRoot

# El orden en que se actualizan los nodos importa, pues rotnNode es el hijo
# de newRoot. Y para que newRoot tenga un bf correcto, la altura de rotNode
# debe estar actualizada
self._update_node(rotNode)
self._update_node(newRoot)
```

Franco Yudica 13922

Implementación calculate Balance

Para el siguiente método, asumo que no se ha calculado el balance factor ni las alturas de ningún nodo. Por eso es que necesito calcular esos valores desde las hojas hasta las raíces, razón por la cuál la llamada recursiva se encuentra antes de los cálculos de las alturas

```
def calculate_balance(self):

    def _update_node_upwards(node):
        if node is None:
            return

        # Si el nodo es hoja
        if node.leftnode is None and node.rightnode is None:
            node.height = 0
            node.bf = 0
        else:
            # Cuando el nodo no es hoja, tenemos que calcular
            # las alturas de los nodos hijos para luego poder
            # determinar la altura del nodo y su balance factor
            _update_node_upwards(node.leftnode)
            _update_node_upwards(node.rightnode)
            self._update_node(node)

    _update_node_upwards(self.root)
```

Franco Yudica 13922

Implementación reBalance

La estrategia utilizada en mi implementación es la siguiente:

1. Acceder mediante recursividad a las hojas del árbol, de tal forma que realizamos el proceso desde las hojas hasta la raíz
2. Para cada nodo llamo _update_node, calculando el balance factor y altura de cada nodo.
3. Evalúo si es necesario realizar una rotación para balancear el nodo, nótese que por haber partido desde las hojas siempre basta con una rotación simple o doble del nodo. De esta forma puedo garantizar que los subárboles izquierdo y derecho de cada nodo siempre van a estar balanceados.
4. Una vez balanceado el nodo, repito el proceso con el padre, y así hasta llegar a la raíz

(Nótese que en este caso no fue necesario llamar a la función calculate_balance, pues el balance factor de cada nodo se calcula a medida que se escala en el árbol y cada vez que se rota el nodo)

```
def balance(self):

    """
    Si el AVL tiene desbalances, por haberse copiado de un bst,
    se utiliza balance.
    Luego de llamar al método balance, el árbol resulta ser un AVL balanceado
    """

    def _balance_node_upwards(node):
        """
        Parte desde las hojas hasta llegar hasta la raíz, por eso es upwards
        El objetivo es balancear los sub-árboles mas chicos hasta llegar a la raíz,
        de esta manera el problema se simplifica y siempre sabemos que los sub-árboles
        izquierdos y derechos se encuentran balanceados
        """
        if node is None:
            return

        # Primero bajamos hasta llegar a las hojas
        _balance_node_upwards(node.rightnode)
        _balance_node_upwards(node.leftnode)

        # Si este fuera el primer nodo, la altura sería 0 ya que la altura de los
        # nodos está inicializada en 0. Es por eso que no hace falta hacer un caso
        # especial para los nodos hoja
        self._update_node(node)

        # Si se encuentra desbalanceado
        if abs(node.bf) > 1:

            # Balanceamos el nodo
            self._balance_node(node)

            # Continúo con el proceso hasta llegar hasta la raíz
            _balance_node_upwards(node.parent)

    _balance_node_upwards(self.root)
```

Franco Yudica 13922

Implementación insert

Lo primero que hago es insertar al nodo de igual manera que en un árbol binario de búsqueda

```
def insert(self, value, key):

    # Creo el nodo a insertar
    node = AVLNode()
    node.key = key
    node.value = value
    node.bf = 0
    node.height = 0

    # Inserto el nodo
    if self.root is None:
        self.root = node

    else:
        self._insert_bst(self.root, node)

    self._update_upwards(node)
```

```
def _insert_bst(self, currentNode, node):
    """
    Inserción en arbol binario de búsqueda
    """
    if currentNode.key < node.key:
        if currentNode.rightnode is None:
            currentNode.rightnode = node
            node.parent = currentNode
            return

        self._insert_bst(currentNode.rightnode, node)

    else:
        if currentNode.leftnode is None:
            currentNode.leftnode = node
            node.parent = currentNode
            return

        self._insert_bst(currentNode.leftnode, node)
```

Luego, voy a encontrar (en caso de que exista) el nodo desbalanceado, el cuál se encuentra en el camino desde el padre del nodo insertado a la raíz del árbol.
Si se encuentra entonces balanceo el nodo y termina el proceso. De esta forma puedo garantizar que el árbol es AVL luego de la inserción

Franco Yudica 13922

```python
def _update_upwards(self, current):
    """
    Update upwards es un método que dado un nodo, escala hasta
    la raíz del árbol en caso de ser necesario. Y para cada uno
    de los nodos en ese camino, verifica que se encuentren balanceados.
    En caso de que se encuentre un nodo balanceado, se lo balancea y termina
    de escalar.
    Este método se utiliza en insert y en delete, para balancear el árbol
    """
    if current is None:
        return

    self._update_node(current)

    # Found the unbalanced node, re balance the node and quits recursion
    if abs(current.bf) > 1:
        self._balance_node(current)
        return

    self._update_upwards(current.parent)
```

Franco Yudica 13922

Implementación delete

De manera muy semejante a insert, lo que hago es eliminar el nodo de la misma forma que en un árbol binario de búsqueda.

```
def delete(self, key):

    node = self._access_bst(self.root, key)
    if node is None:
        return False

    # Keeps track of the parent
    parent = node.parent

    # Removes the node fron the tree
    sucessor = self._delete_recursive_bst(node)

    # Actualiza balance factors y balancea
    # en caso de ser necesario desde el sucesor
    if sucessor is not None:
        self._update_upwards(sucessor)

    # En caso de no tener sucesor, parte desde
    # el parent
    elif parent is not None:
        self._update_upwards(parent)
```

Nótese que _delete_recursive_bst es un método que retorna el sucesor del nodo eliminado. El cuál se utiliza para poder balancear hacia arriba de la misma manera que con el insert, y en caso de que no tenga sucesor (si el nodo eliminado es hoja) entonces puedo partir desde el parent

Franco Yudica 13922

Implementación _update node

Este método lo usé para todas las implementaciones de las funciones pedidas, simplemente calcula el balance factor y la altura del nodo en base a las alturas de los nodos hijos

```python
def _update_node(self, node):
    """
    Calcula la altura del nodo basándose en las alturas de los nodos hijos
    Y luego calcula balance factor
    Nótese que se inicializan las alturas de los hijos con -1, de tal forna
    que si no tiene hijos la diferencia da 0 (bf) y luego, la altura del
    nodo también sera 0
    """
    left_height = -1
    right_height = -1

    if node.leftnode is not None:
        left_height = node.leftnode.height
    if node.rightnode is not None:
        right_height = node.rightnode.height

    node.height = 1 + max(left_height, right_height)
    node.bf = left_height - right_height
```

Franco Yudica 13922

Parte 2

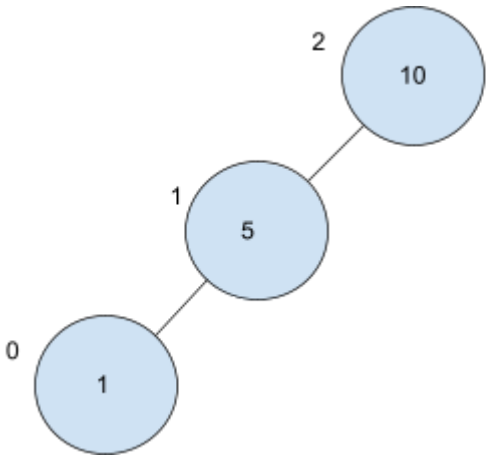
Ejercicio 6

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

AVL => penúltimo nivel tiene que estar completo

Supongo que es falso, entonces: En un AVL, no es necesario que el penúltimo nivel sea completo. (1)

Busco un contraejemplo a esta implicación

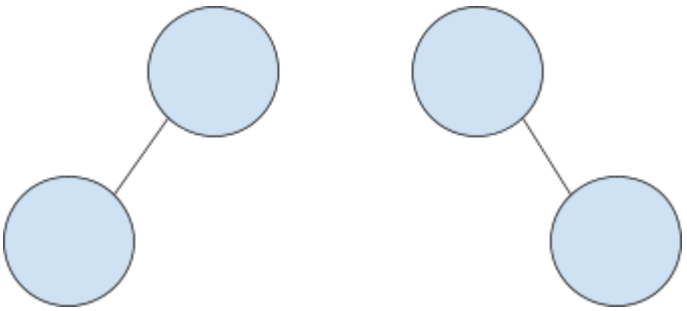


En este caso nos encontramos con un árbol binario de búsqueda, pero nótese que está desbalanceado, pues el penúltimo nivel no está completo.
Es decir que la implicación (1) es falsa. Y por ser (1) la implicación contraria a la del enunciado, sabemos que tiene distinto valor de verdad y consecuentemente el enunciado a) es **verdadero**

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

AVL donde bf = 0 para todo nodo => arbol completo

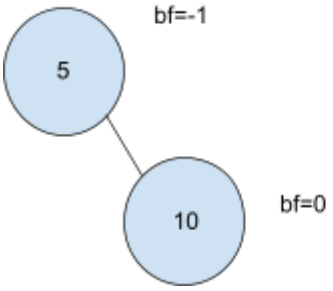
Supongo que la implicación es falsa, es decir que existe un AVL donde el balance factor de todos los nodos es 0 y no es completo.
Sabemos que si el árbol no es completo, va a ser de la forma



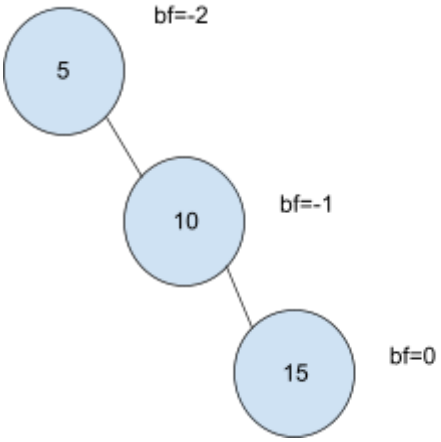
Pero claramente, en estos casos va a existir un nodo (en este caso las raíces) con un balance factor distinto de 0. Lo cuál demuestra que la negación del enunciado es Falsa, por lo tanto el enunciado es **verdadero.**

c) En la inserción en un AVL, si al actualizar 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.

Creo que el enunciado es falso, pero para demostrarlo debo encontrar un contraejemplo.
Supongamos que tenemos el siguiente AVL, el cuál se encuentra balanceado



Ahora, lo que voy a hacer es insertar un nodo con un key = 15, mayor que 10



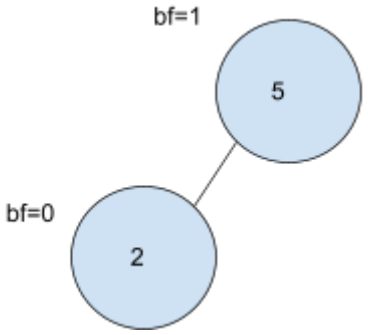
Se observa que luego de haber insertado, el desbalance se produce en el nodo raíz. Según el supuesto de c) solo hay que verificar si se desbalancea el padre del nuevo nodo. Pero mediante un ejemplo podemos observar que es necesario balancear otro nodo ancestro del padre del insertado. Luego por contraejemplo sabemos que el enunciado es **Falso**

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

Es **verdadero**, pues las hojas siempre tienen factor de balance 0.

e) En todo *AVL* existe al menos un nodo que no sea hoja con factor de balance 0.

Vamos al caso más simple



Se observa que es un AVL, pero no hay nodos, que no sean las hojas que tengan balance factor igual a 0. Por contra ejemplo he demostrado que el enunciado d) es **falso**

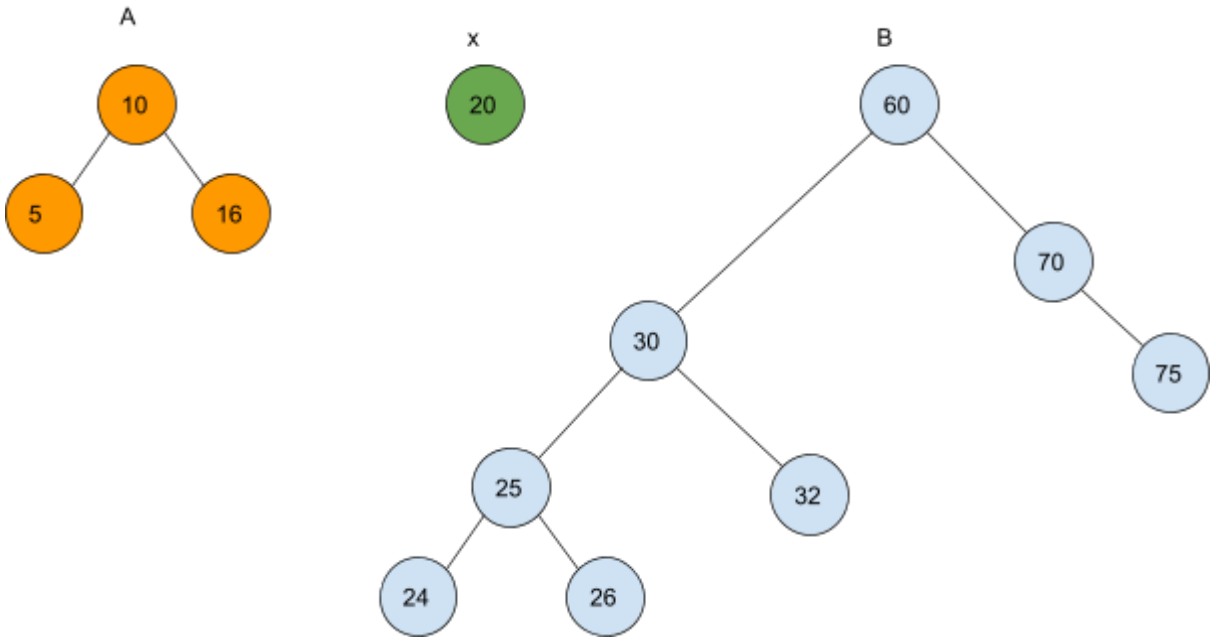
Ejercicio 7

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*.

- Se observa que los árboles no tienen necesariamente la misma altura
- El valor X es mayor que todas las keys del árbol A, pero menor que las del B

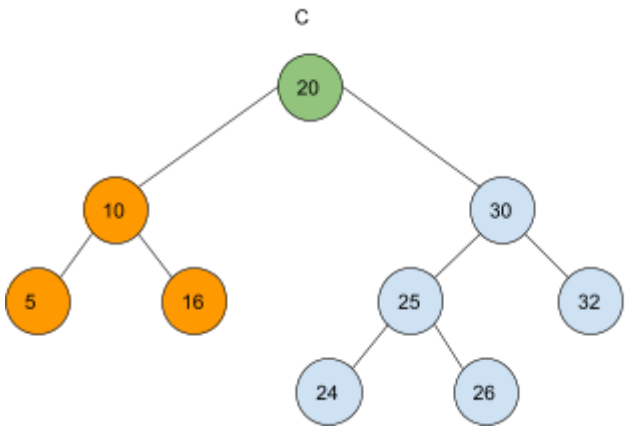
Para realizar la interpretación del problema voy a trabajar con los siguientes árboles

Franco Yudica 13922

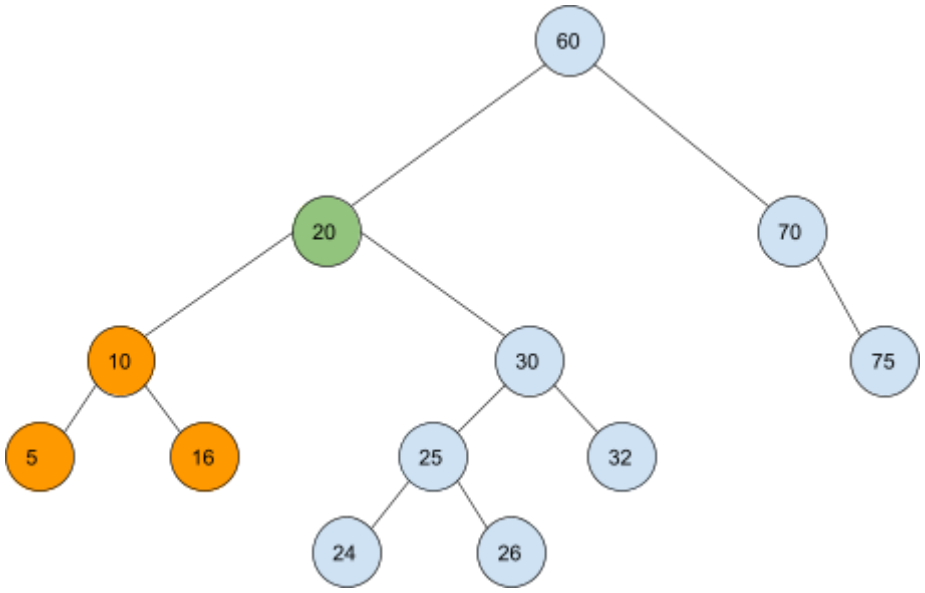


1. Lo primero que voy a hacer es tomar el árbol de menor altura, que este es el árbol A, la altura del árbol A es 2.
2. Voy a localizar al nodo del árbol B, cuya altura (la propiedad altura del nodo) sea igual a la del árbol A. En este caso es el nodo con el valor 30.
3. Ahora, voy a armar un nuevo árbol, llamemoslo C. El cuál va a tener como subárbol izquierdo al árbol A y como subárbol derecho al árbol con raíz del nodo 30. Y claramente C, tiene como raíz al nodo con el valor x=20.

   Nótese que al hacer esto, el árbol C siempre está balanceado

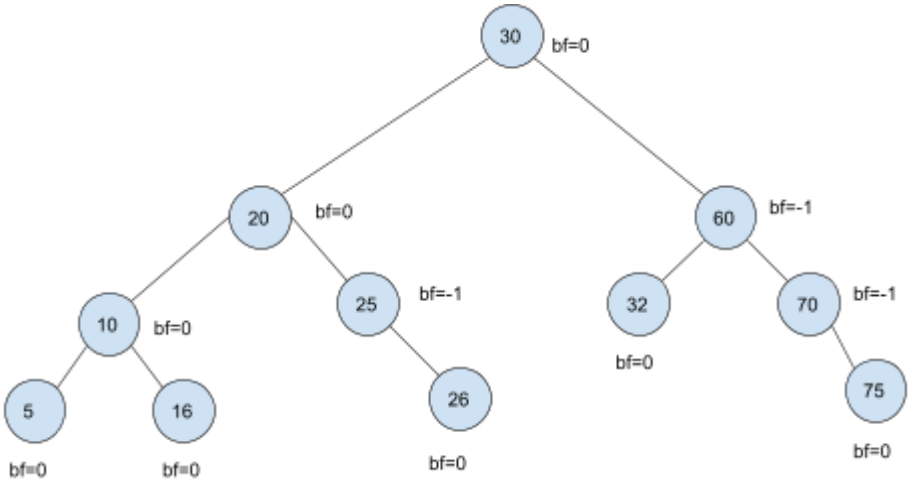


4. Luego, se reemplaza todo el subárbol con la raíz 30 por el nuevo árbol C



5. Se observa que el árbol se encuentra desbalanceado, vamos a tener que comprobar cuales de los nodos que conectan el nodo 20 y el 60 están desbalanceados y balancearlos.

   En este caso, el único nodo desbalanceado es el 60, por lo tanto se balancea y el árbol AVL final resulta ser el siguiente



El algoritmo es de orden O(log(n) + log(m)) pues primero hay que calcular las alturas de los árboles y luego hay que insertar el árbol C y realizar las rotaciones

Ejercicio 8

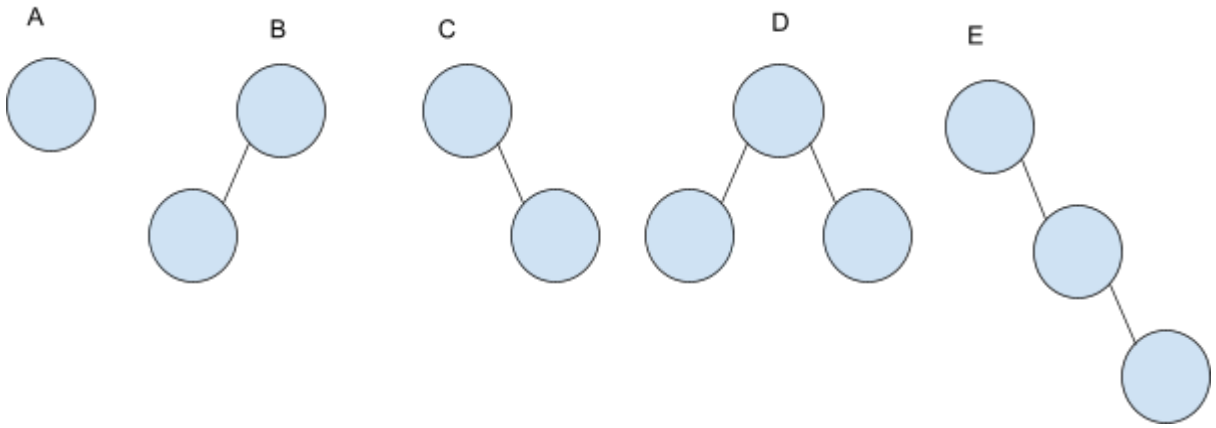
Considere una rama truncada en un AVL como un camino simple desde la raíz hacia un nodo que tenga una referencia None (que le falte algún hijo). Demuestre que la mínima longitud (cantidad de aristas) que puede tener una rama truncada en un AVL de altura h es h/2 (tomando la parte entera por abajo).

Franco Yudica 13922

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

Supongamos que el enunciado es falso, es decir que se puede encontrar una rama truncada cuya longitud sea menor que h/2.

Para la demostración vamos a trabajar con los casos más simples posibles



- En el caso A se observa que la altura es h = 0, el nodo raíz se puede considerar como una rama truncada pues tiene hijos nulos, según la regla, la longitud de la rama truncada es como mínimo h/2, 0/2, 0, lo cuál se cumple en este caso
- Caso B: h = 1, podemos encontrarnos una rama truncada, con altura h=½=0, que es el nodo raíz y se cumple el enunciado
- Caso C: Misma situación que el caso B
- Caso D: h=1, nos encontramos con dos ramas truncadas, cuya longitud es 1. Esto no rompe con el supuesto, dado a que este supone que tiene una longitud como mínimo de 0, y en este caso es 1
- Caso E: h=2, observamos que el nodo raíz no tiene hijo izquierdo y por lo tanto nos encontramos con una rama truncada con una longitud menor que h=2/2=1, pero se observa que el árbol no es AVL, por lo tanto, el supuesto no aplica en este caso

Se observa que para encontrar una rama truncada con longitud menor que h/2 es necesario que el árbol no sea AVL, por lo tanto podemos decir que el supuesto es **verdadero**