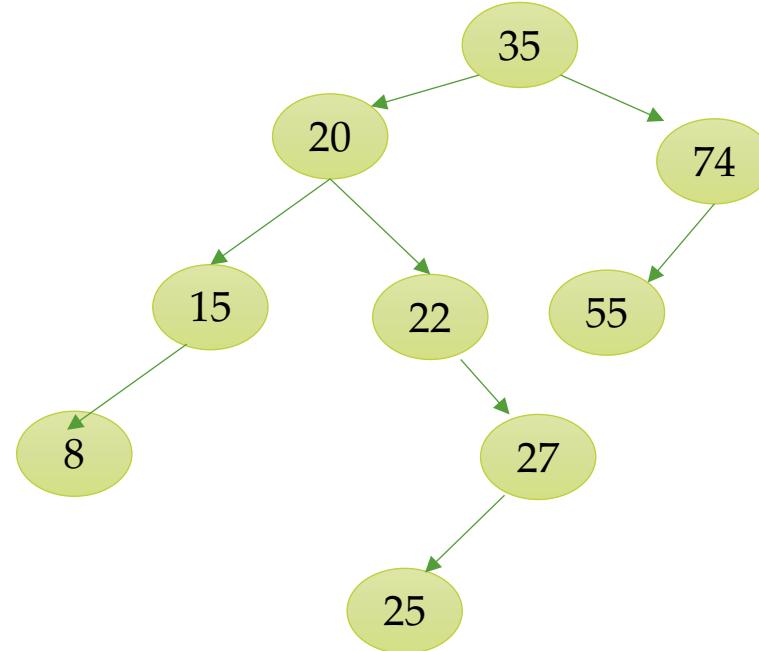


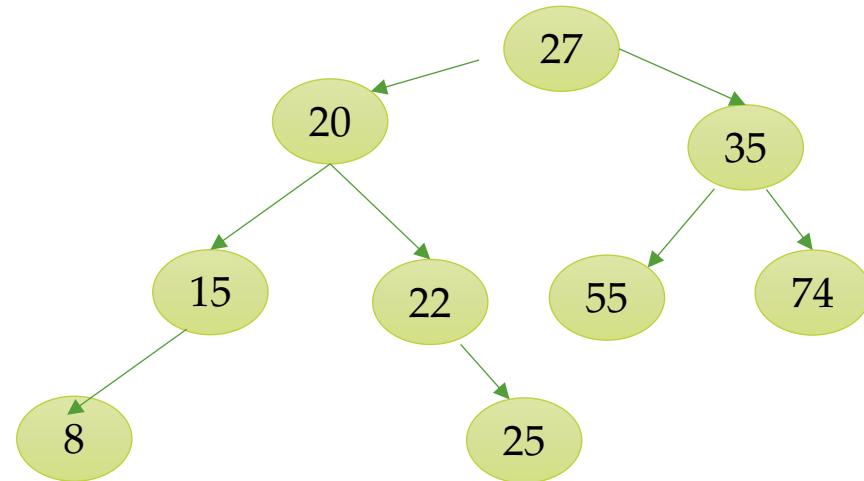
# BST y sus problemas

Este BST no es completo, no está lleno



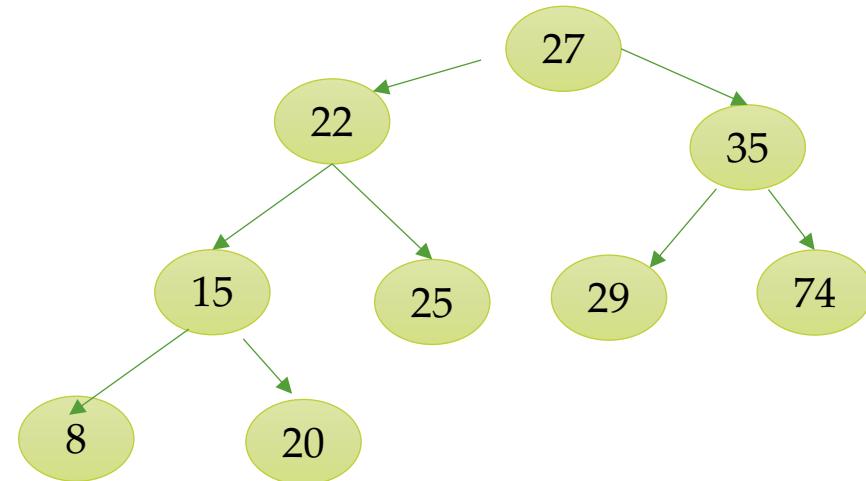
# BST y sus problemas

Este BST no es completo, no está lleno



# BST y sus problemas

Este BST es completo, no está lleno



Si en particular hablamos de los BST, que un árbol sea completo implica que la “forma del árbol es muy buena”.

En un árbol así: ¿Cuál es la complejidad temporal para búsqueda? ¿Para inserción? ¿Para borrado?

# BST y sus problemas

Recapitulando...

- **El peor caso es cuando está totalmente desbalanceado.**

En ese caso se pierde toda la ventaja de búsqueda binaria, necesaria inclusive para inserción/borrado/búsqueda en sí.

Si los elementos llegan en forma ordenada, **se obtiene un árbol degenerado**. Se obtiene algo que tiene la desventaja de la lista lineal en cuanto a cómo llegar a un elemento en particular, pero ocupando mucho más espacio!

- **En general, todas las operaciones son  $O(h)$  donde  $h$  es la altura del árbol. Por eso, cuando está completo es  $O(\log n)$  y sino puede llegar a ser  $O(n)$**

**Conclusión:** hay que tratar de que la altura del árbol sea la mínima posible. O, por lo menos, tenerla controlada.

# Algunas propiedades...

Pero podríamos no ser tan exigentes y tener una muy buena forma..

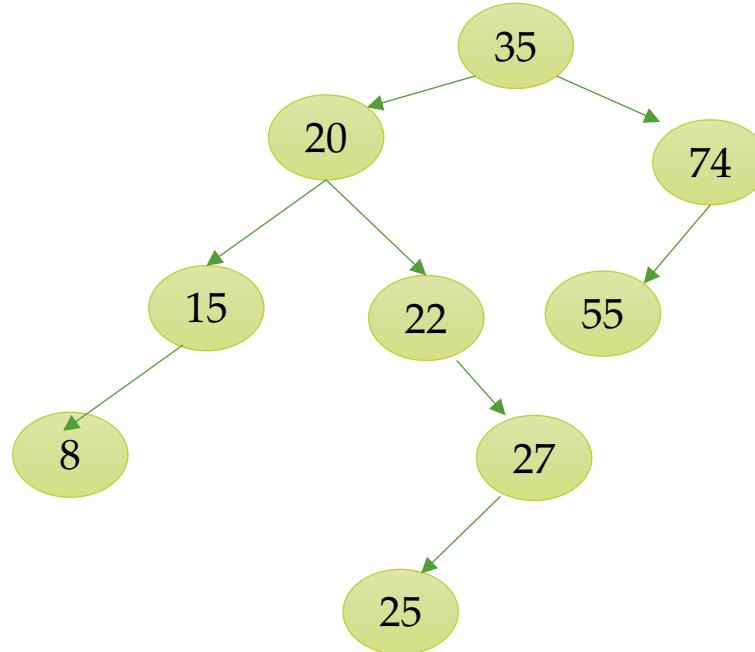
## **Definición AVL: “BST balanceado por altura”**

G.M. Adelson-Velskii y E.M. Landis propusieron en 1962 la primera versión de BST que se balancea dinámicamente.

Un AVL es un BST donde en cada nodo la diferencia de alturas entre sus 2 subárboles es a lo sumo 1.

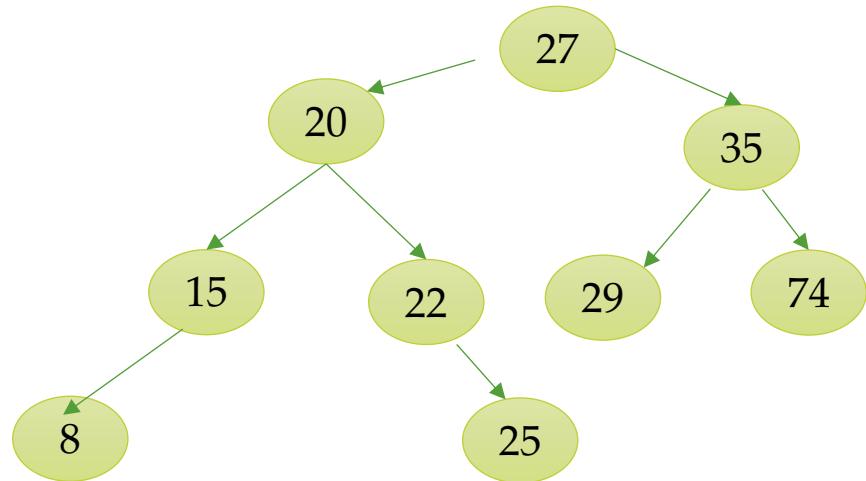
# AVL

Ej: no es AVL. ¿Cuáles son los nodos con problemas?



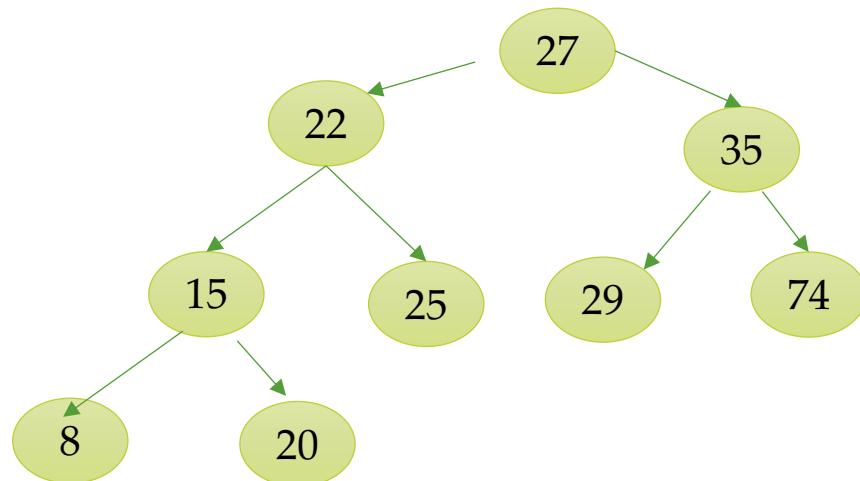
# AVL

Ej: es AVL (aunque no es completo)



# AVL

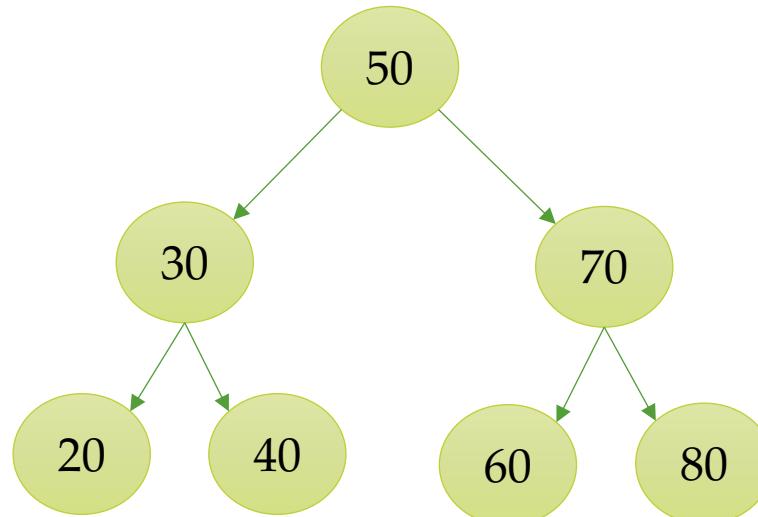
Ej: es AVL (y completo)



# AVL

**Ejercicio** ¿Cuál es AVL? Justificar calculando el factor de balance en cada nodo

Caso A:

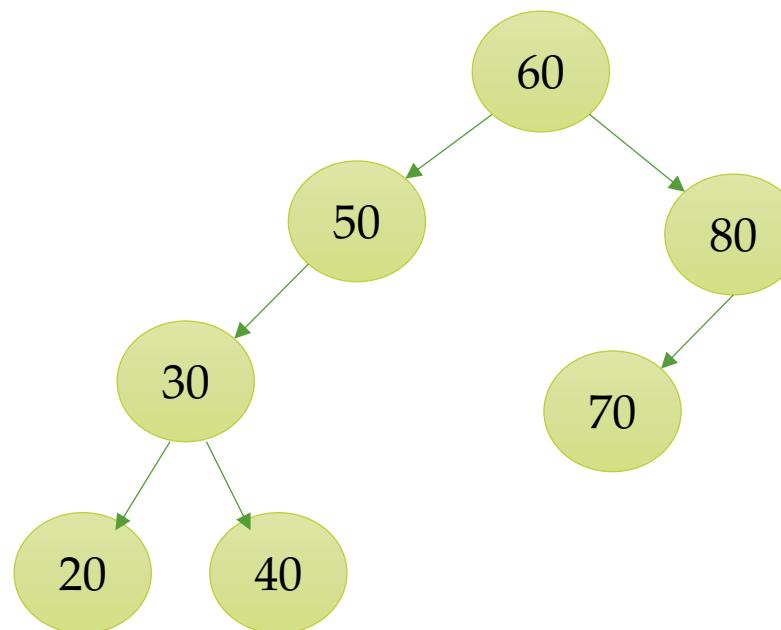


Rta: Sí. Todo nodo tiene  $fb = 0$

Un árbol lleno es AVL

# AVL

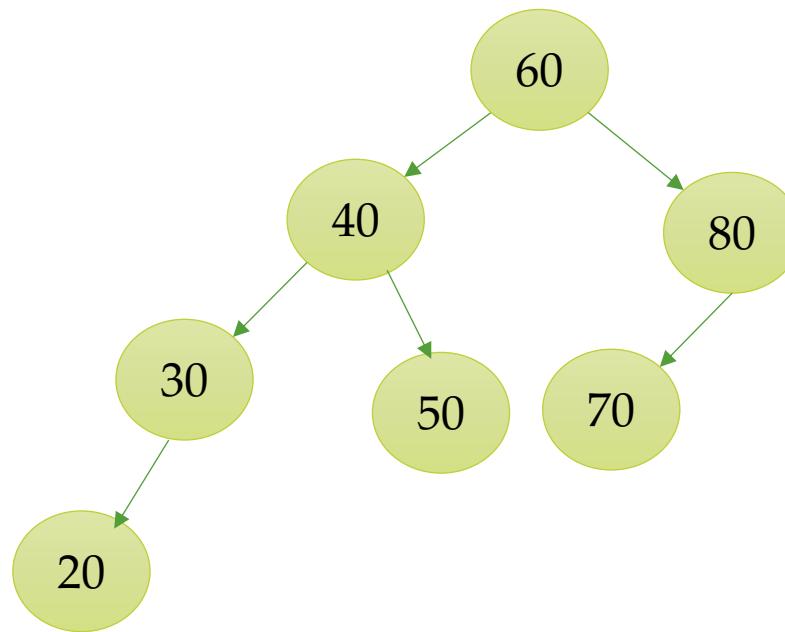
¿Cuál es AVL? Justificar calculando el factor de balance en cada nodo  
Caso B:



Rta: No. Del nodo 50 la h del subárbol izq es 1 y la del subárbol derecho es -1 o sea  $fb = 2$ . Es el único nodo que no satisface.

# AVL

¿Cuál es AVL? Justificar calculando el factor de balance en cada nodo  
Caso C:



Rta: Sí. Nodos 20, 50 y 70 tienen  $fb = -1 - (-1)$  o sea 0.

Los nodos 30 y 80 tienen  $fb = 0 - (-1)$  o sea 1. El nodo 40 tiene  $fb = 1 - 0$  o sea 1. El nodo 60 tiene  $fb = 2 - 1$  o sea 1

# AVL

Un AVL es un árbol con buena forma. Las inserciones y borrados en un AVL están definidas de tal manera que garantizan que se pueden realizar en  $O(\log n)$  y garantizan ser invariantes ante su propiedad:

**es un BST donde en cada nodo la diferencia de alturas entre sus 2 subárboles es a lo sumo 1.**

# AVL

## Operación Inserción

- Se inserta en el BST
- Si está desbalanceado se aplican rotaciones para que siga siendo AVL => se rota el árbol con pivote más joven desbalanceado (más cercano al nodo insertado). Más precisamente,
  - **Caso 1:** si se inserta a la izquierda de un nodo con factor de balance 1, ese nodo va a tener factor de balance 2 y deja de ser AVL. Hay que rotar.
  - **Caso 2:** Si se inserta a la derecha de un nodo con factor de balance -1, ese nodo va a tener factor de balance -2 y deja de ser AVL. Hay que rotar.

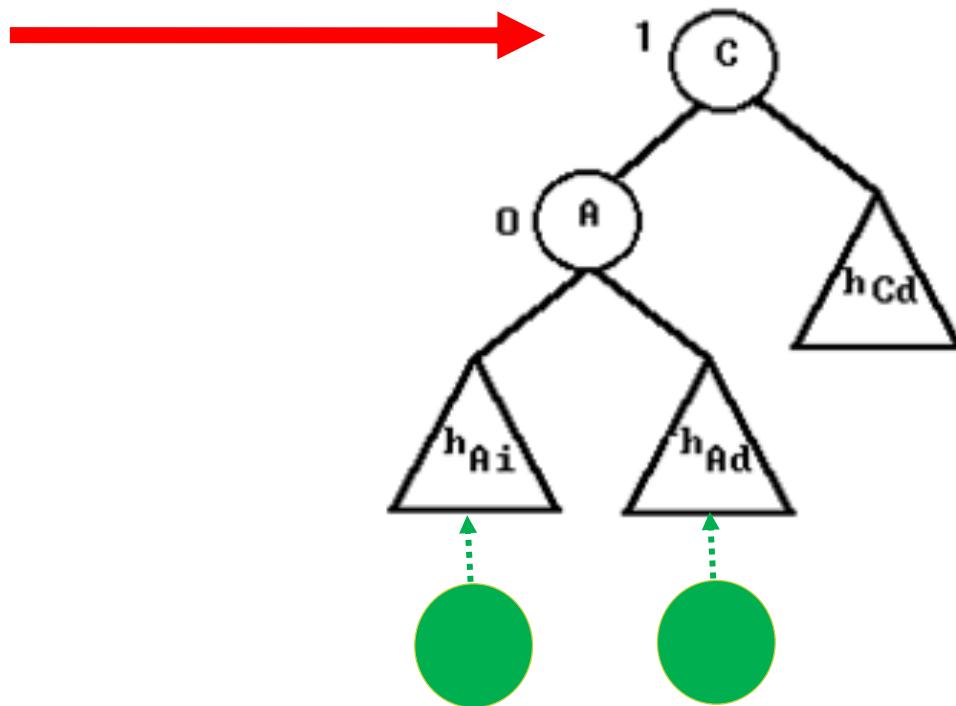
# AVL

## Operación : Rotación

Trasforma un árbol binario en otro, de tal manera que preserva el orden de sus elementos al navegarlo en inorder. Existen rotaciones **simples** y **dobles**.

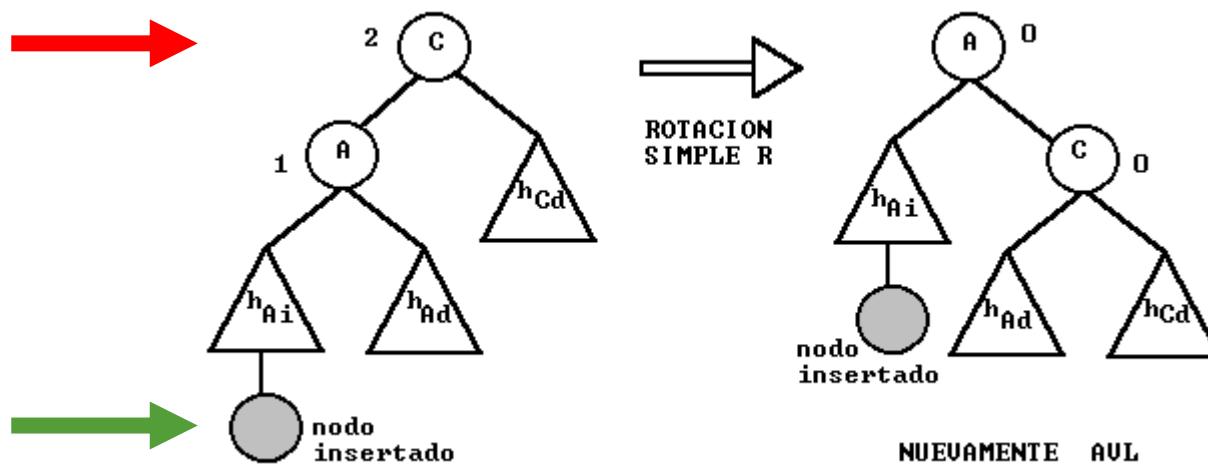
# AVL

- Caso A: Insertar a izquierda de un nodo con fb 1. Este nodo tendría fb 2 y dejaría de ser AVL. Tiene dos subcasos.



# AVL

- Caso A (insertar a izq de un nodo con fb 1) tiene dos subcasos.
  - Caso A1- Rotación simple a derecha (R): cuando se inserta respecto del hijo izquierdo del nodo con factor de balance 1, en el subárbol izquierdo



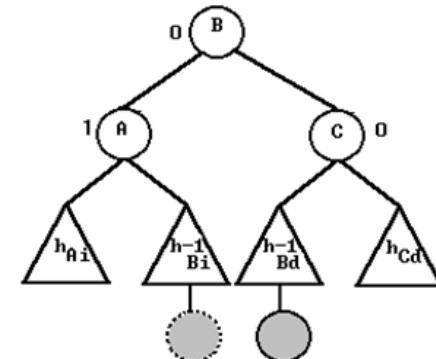
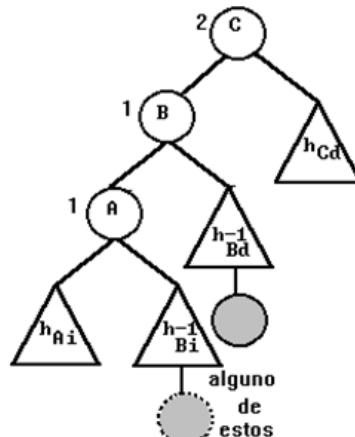
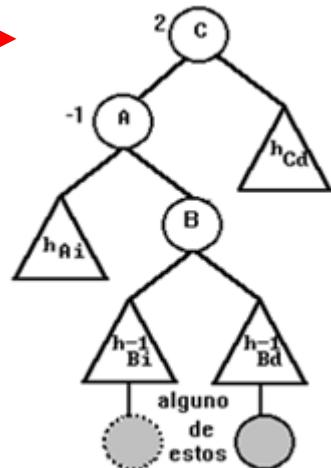
Rotación hacia la **derecha** con pivote en el nodo más joven inicialmente desbalanceado con 2, o sea "**C**"

Primera rotación hacia la **izquierda** con  
pivot en el hijo del nodo  
desbalanceado (del lado de la  
inserción), en nuestro caso “A”

Segunda rotación hacia la **derecha**  
con pivot en el nodo más joven  
initialmente desbalanceado con 2, o  
sea “C”

# AVL

- Caso A (insertar a izq de un nodo con fb 1) tiene dos subcasos.
  - Caso A2- Rotación doble izquierda a derecha (LR): cuando se inserta respecto del hijo izquierdo del nodo con factor de balance 1, en el subárbol derecho



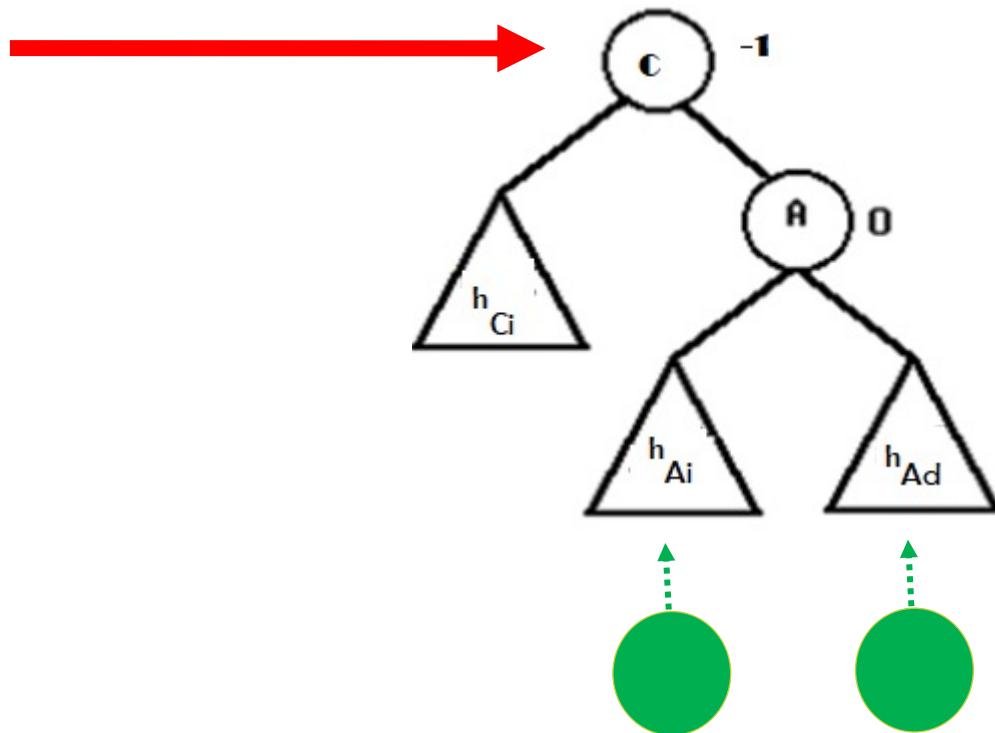
NUEVAMENTE UN AVL

Rotacion simple L con  
pivot en B

Rotacion simple R con  
pivot en C

# AVL

- Caso B: insertar a derecha de un nodo con  $fb = -1$ . Este nodo tendría  $fb = -2$  y dejaría de ser AVL. Tiene dos subcasos.

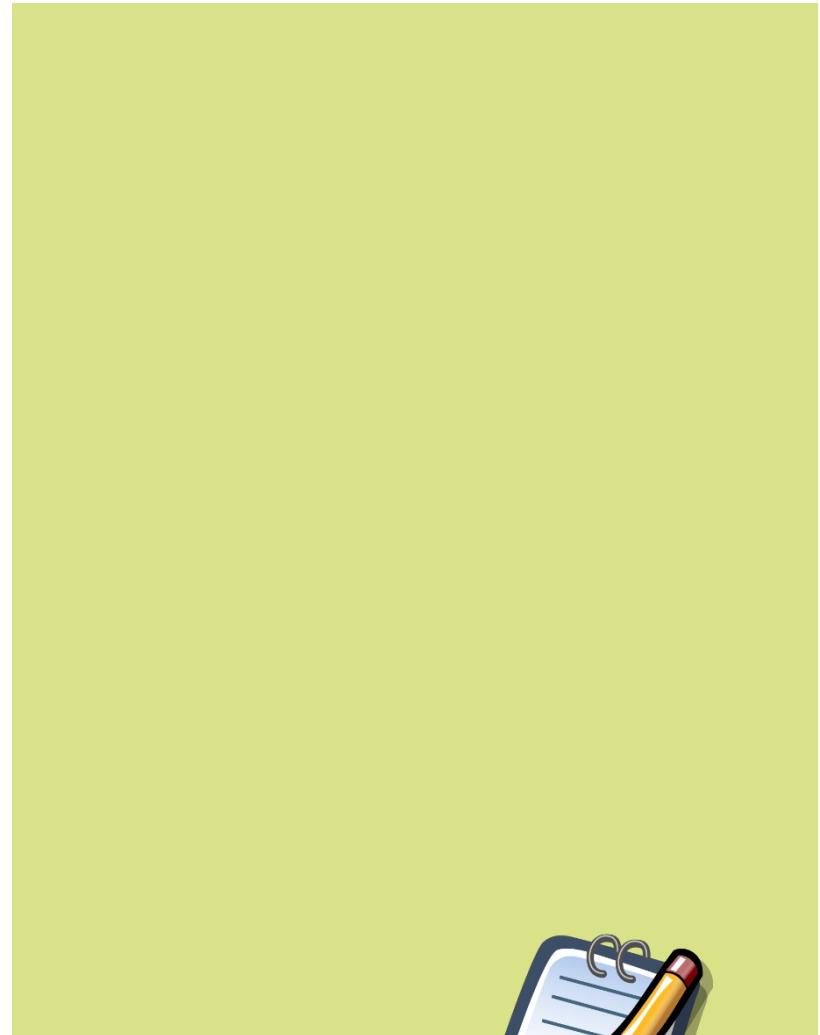


# AVL

- Caso B (insertar a derecha de un nodo con fb -1) tiene dos subcasos.
  - Caso B1- Rotación simple a izquierda (L): cuando se inserta respecto del hijo derecho del nodo con factor de balance -1, en el subárbol derecho
  - Caso B2- Rotación doble derecha a izquierda (RL): cuando se inserta respecto del hijo derecho del nodo con factor de balance -1, en el subárbol izquierdo

Análogas soluciones  
=> caso espejado...

# TP 5D – Ejer 1



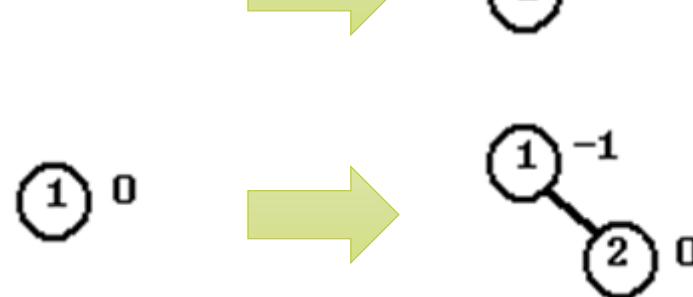
# AVL

Mostrar paso a paso, gráficamente, cómo queda el árbol AVL luego de realizar cada una de las siguientes operaciones de inserción: 1, 2, 4, 7, 15, 3, 10, 17, 19 y 16

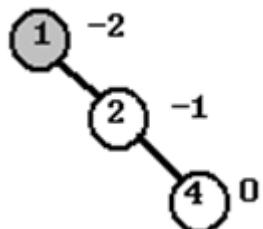
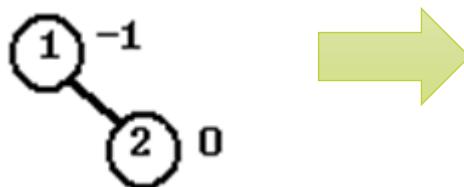
- Insertar 1



- Insertar 2



- Insertar 4



simple  
rotacion  
L

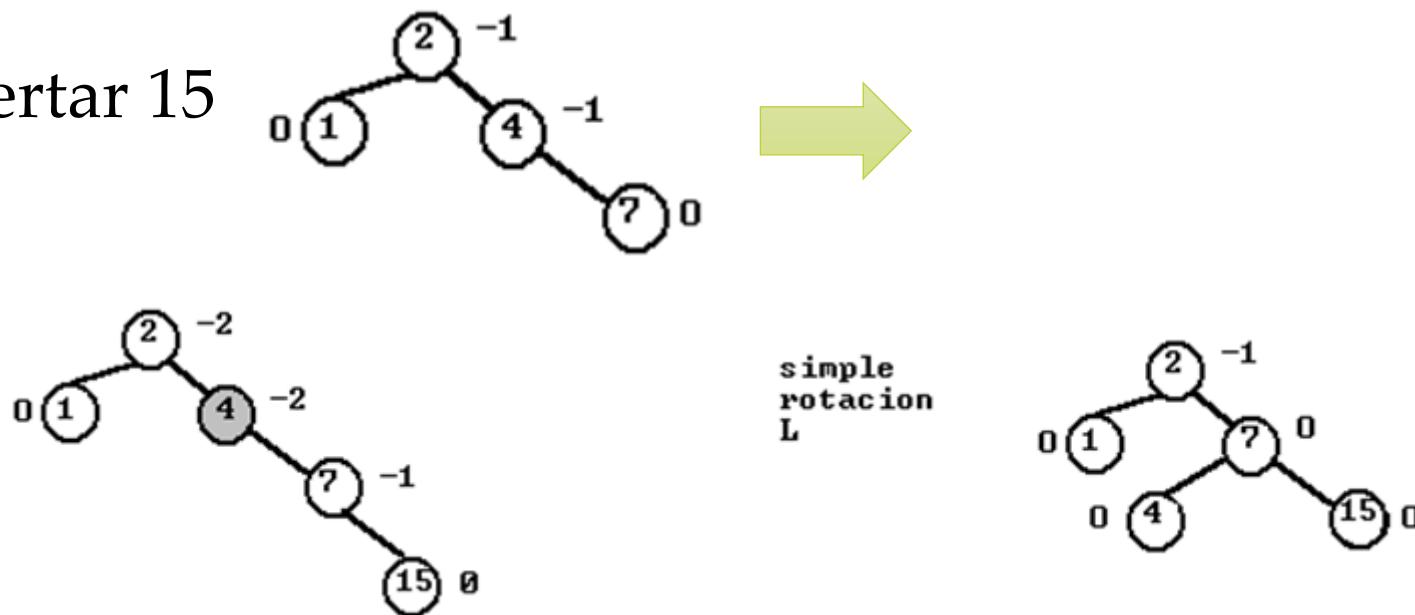


perfectamente  
balanceado, completo

- Insertar 7

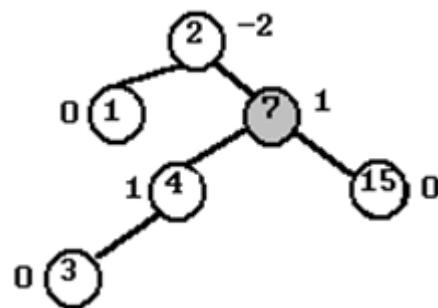
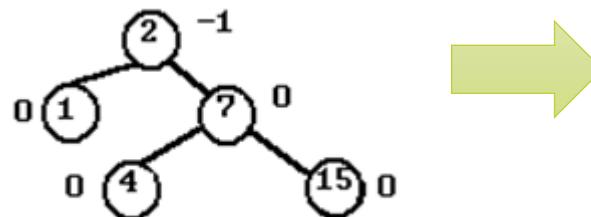


- Insertar 15

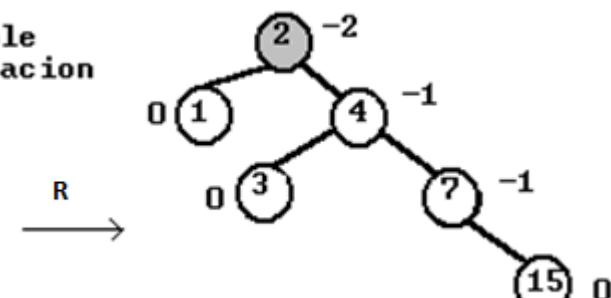


simple  
rotacion  
L

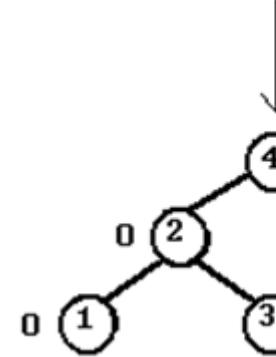
- Insertar 3



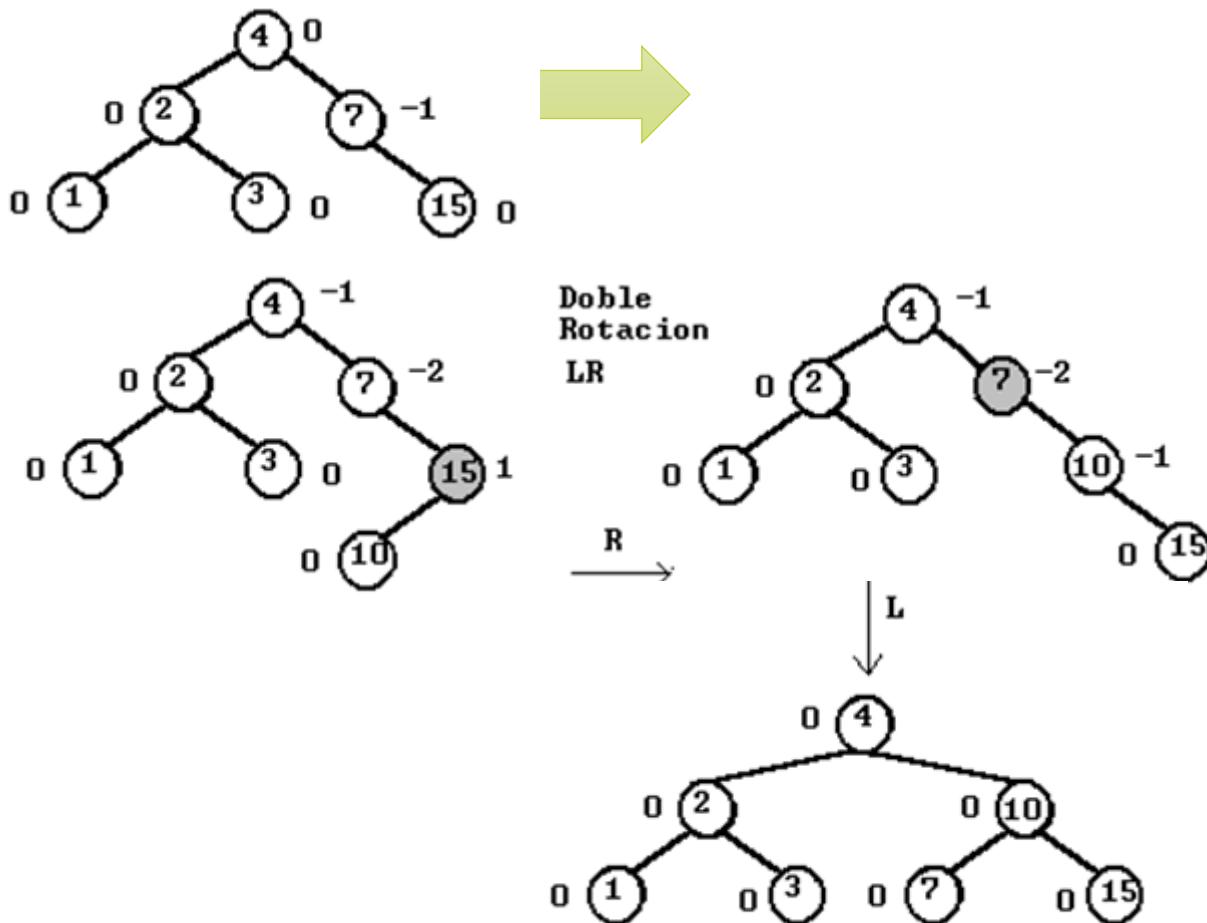
Doble Rotacion



R

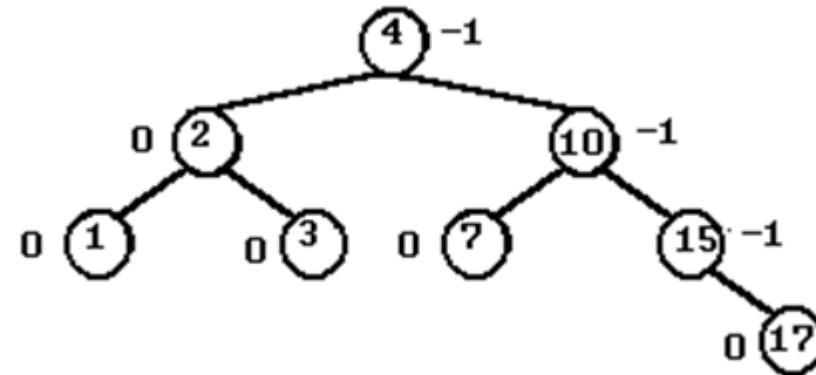
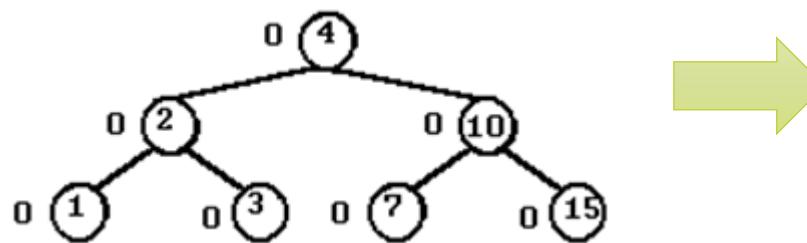


- Insertar 10

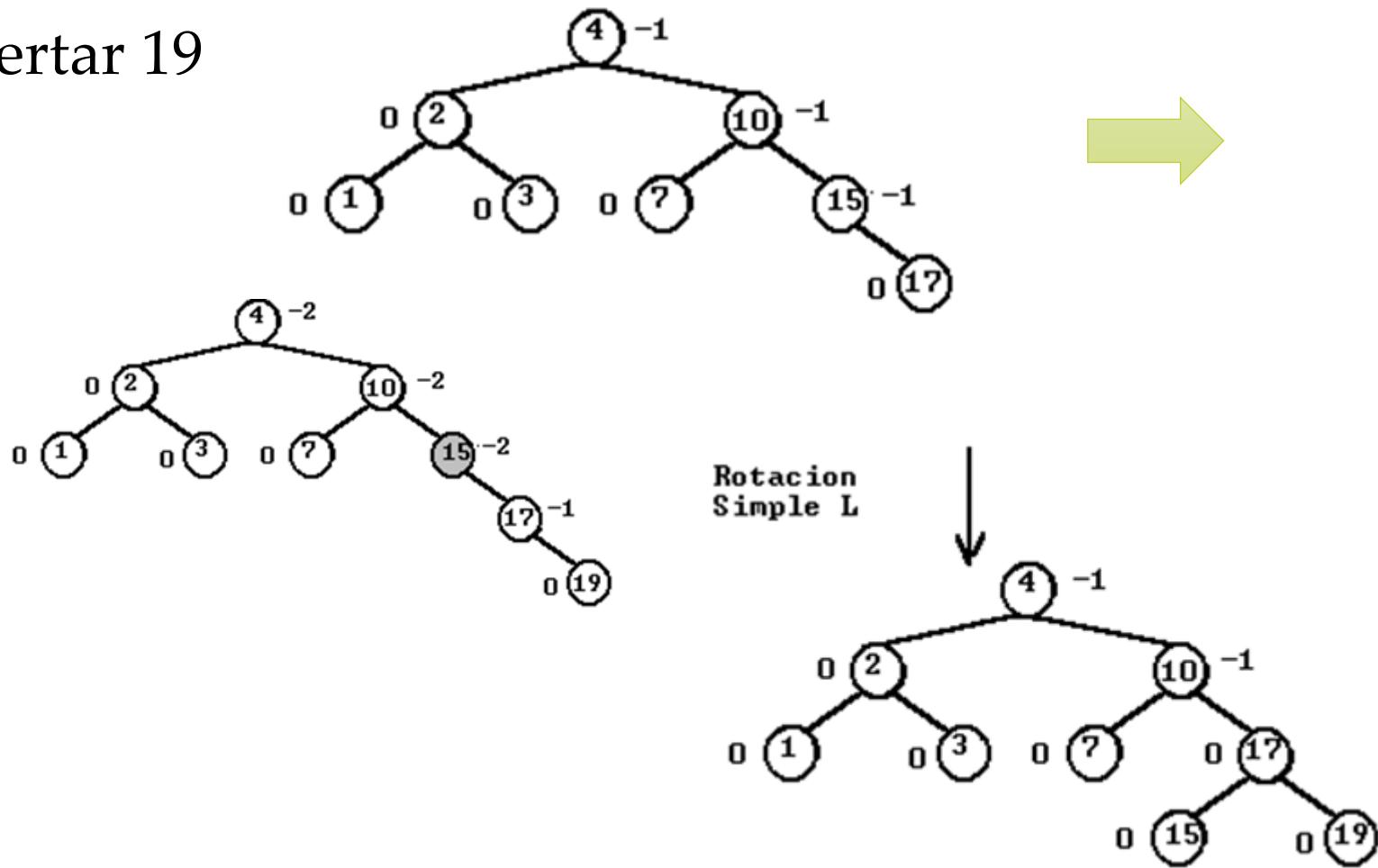


Perfectamente balanceado,  
completo.

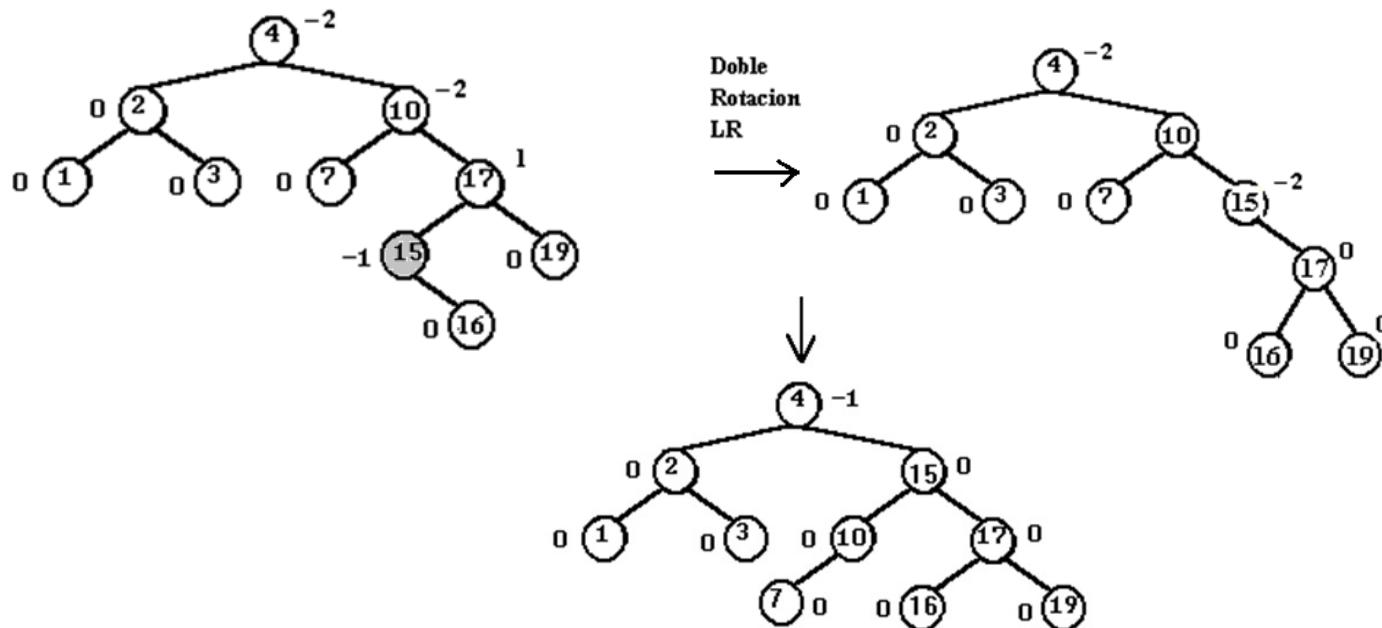
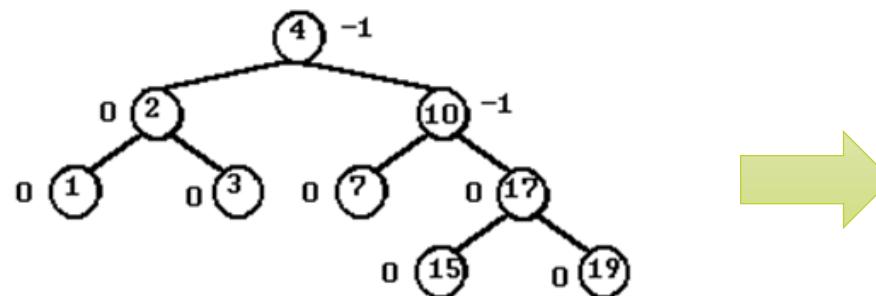
- Insertar 17



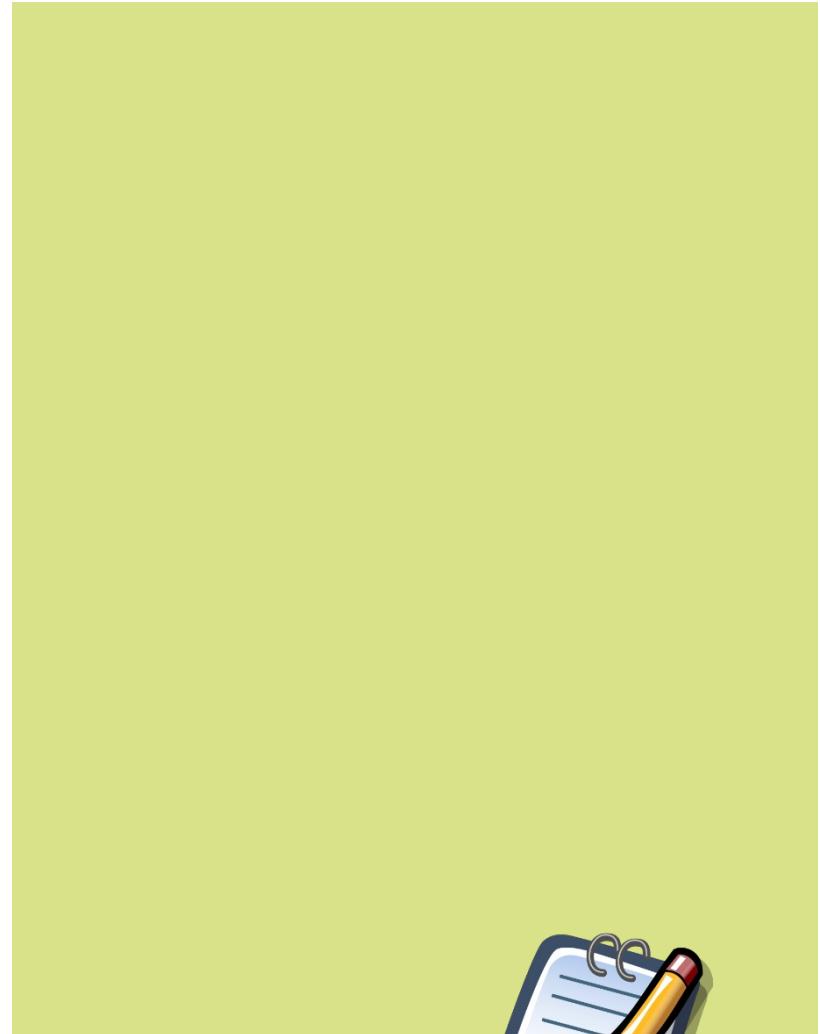
- Insertar 19



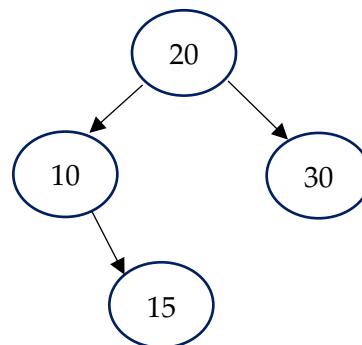
- Insertar 16



# TP 5D – Ejercicio 2



Tomando como comienzo el siguiente árbol AVL

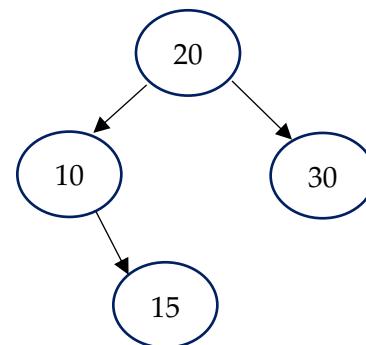


Mostrar **gráficamente** cómo va quedando si se le aplican las operaciones solicitadas en secuencia.

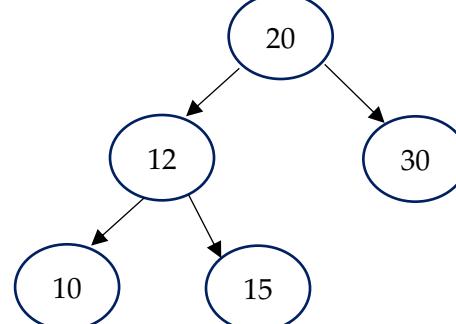
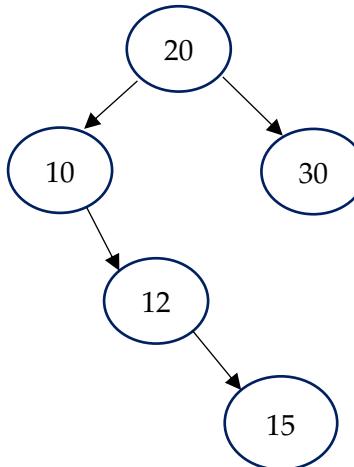
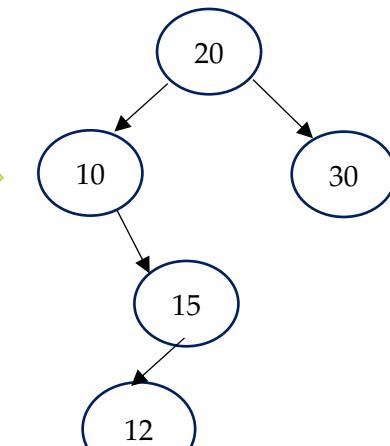
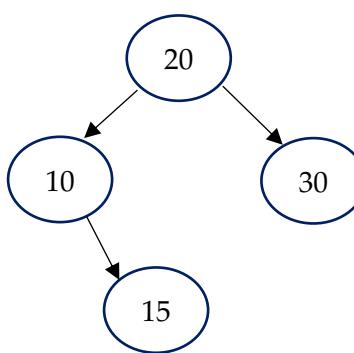
Para cada operación se pide:

- **Mostrar primero gráficamente** dónde se inserta el valor.
- **Analizar si genera o no un desbalance.** Si genera un desbalanceo, indicar **cuál es el tipo de rotación** que lo soluciona. Además **mostrar gráficamente** cómo queda el **árbol** luego de aplicar cada rotación correspondiente. Si es doble, hacerlo en 2 pasos. No mostrar solo el resultado final, sino cada paso intermedio.

- Insertar valor 12
- Insertar valor 14
- Insertar valor 25
- Insertar valor 70
- Insertar valor 90
- Insertar valor 45
- Insertar valor 23
- Insertar valor 27

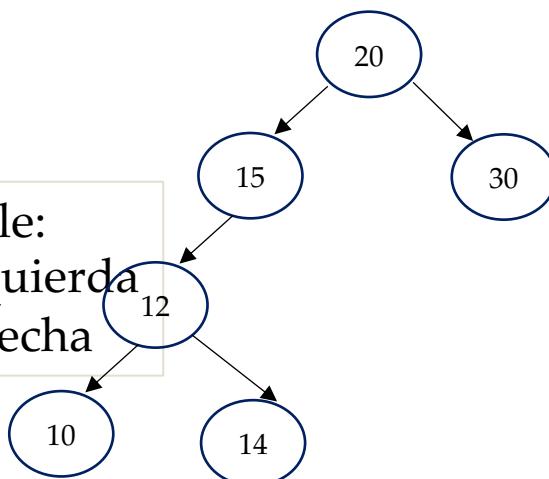
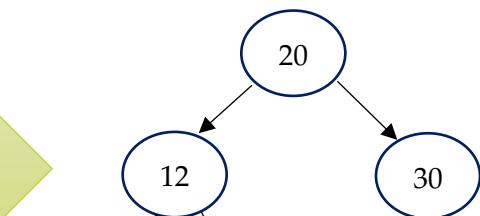
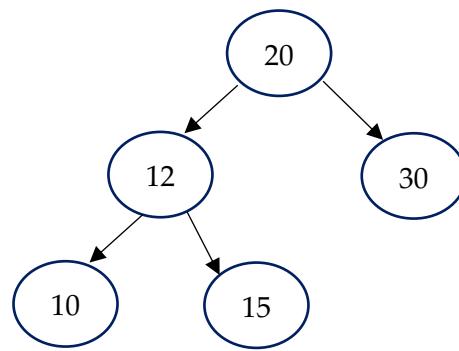


## • Insertar 12

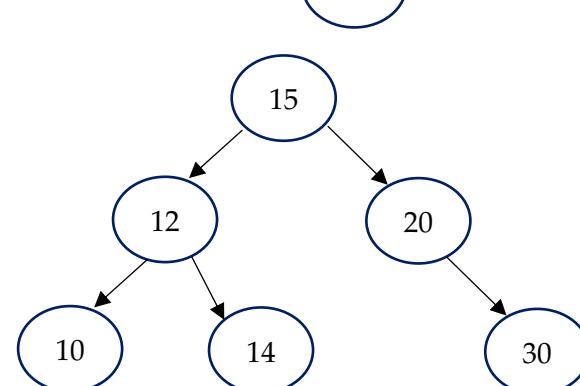


Rotacion doble:  
Primero a derecha y  
Luego a izquierda

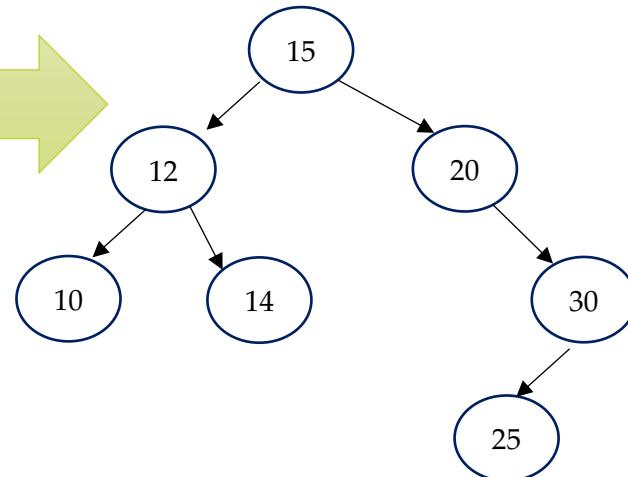
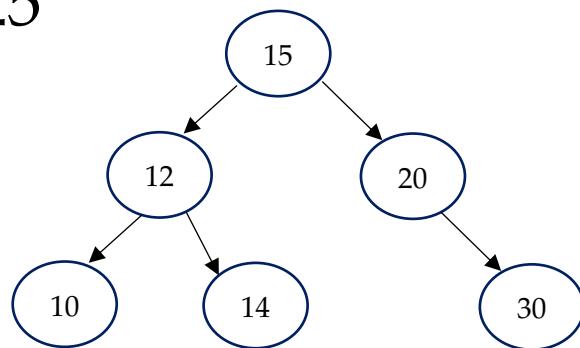
## • Insertar 14



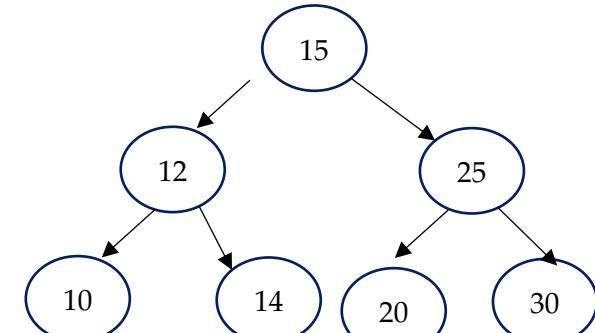
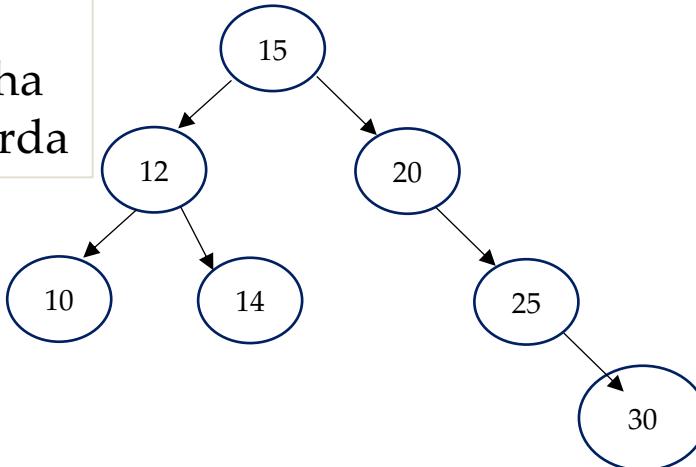
Rotación doble:  
Primero a izquierda  
Y luego a derecha



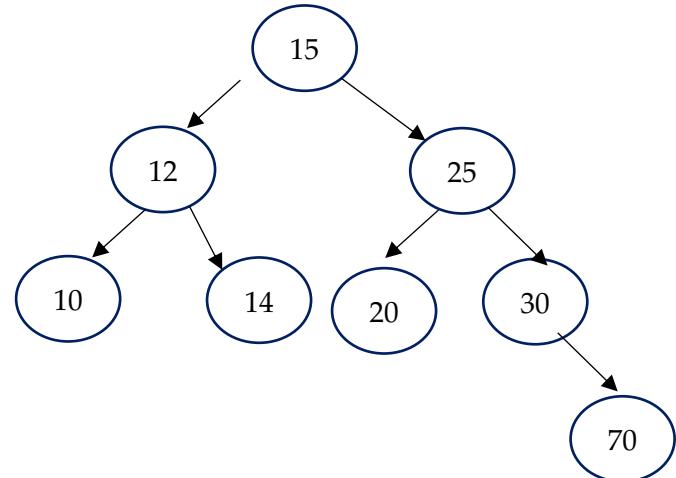
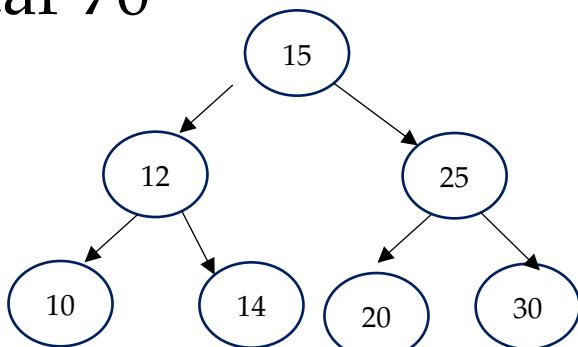
- Insertar 25



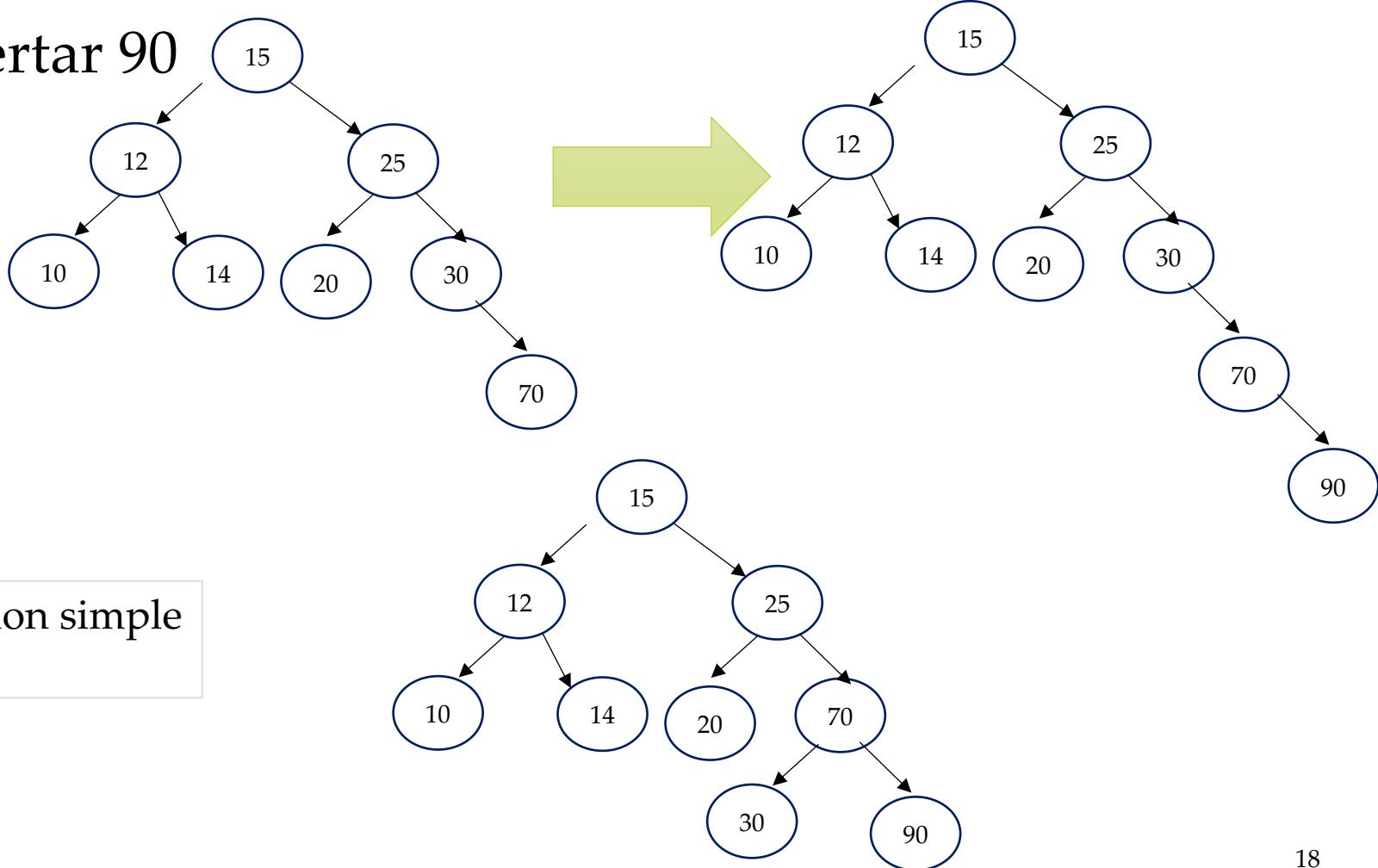
Rotación doble  
Primero a derecha  
Y luego a izquierda

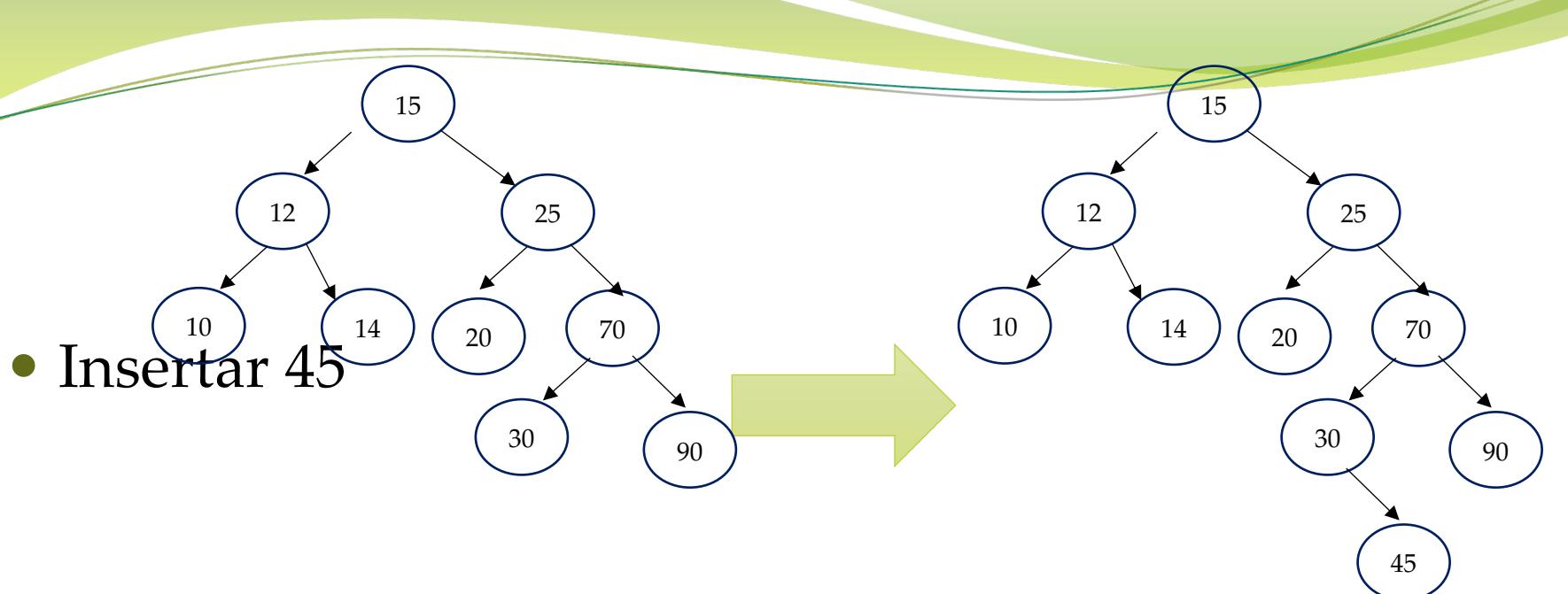


- Insertar 70

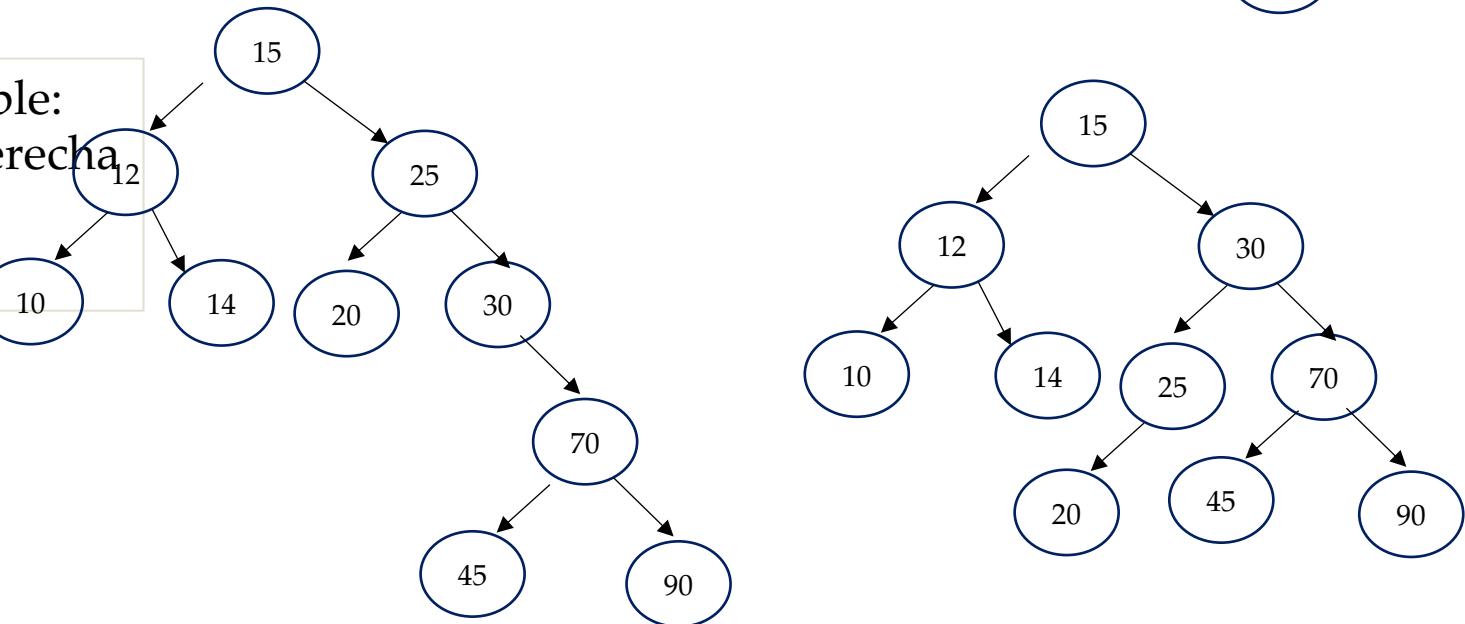


- Insertar 90

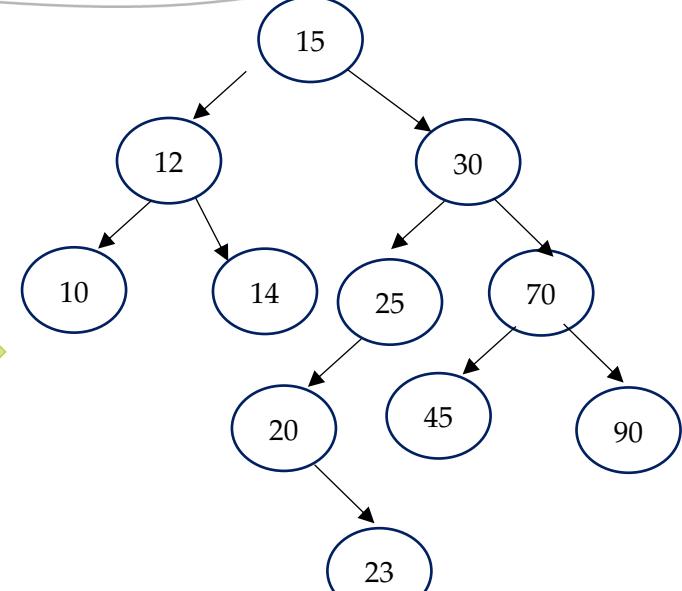
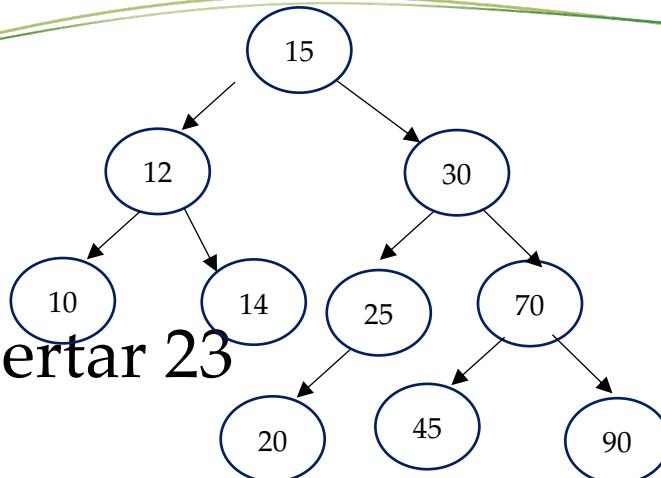




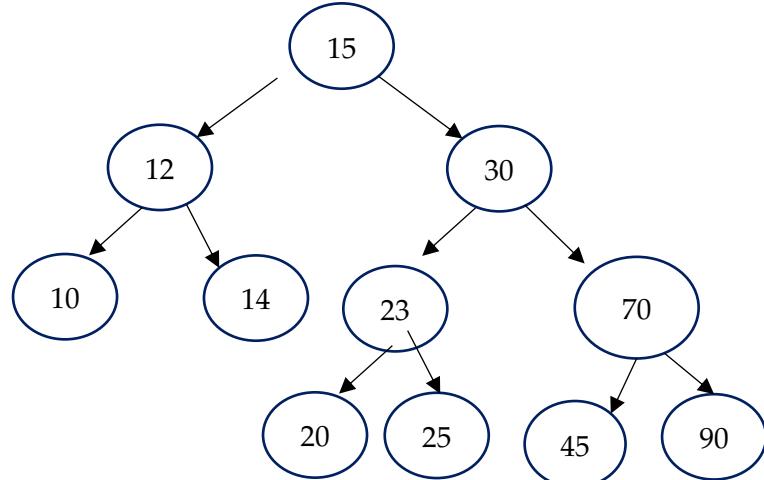
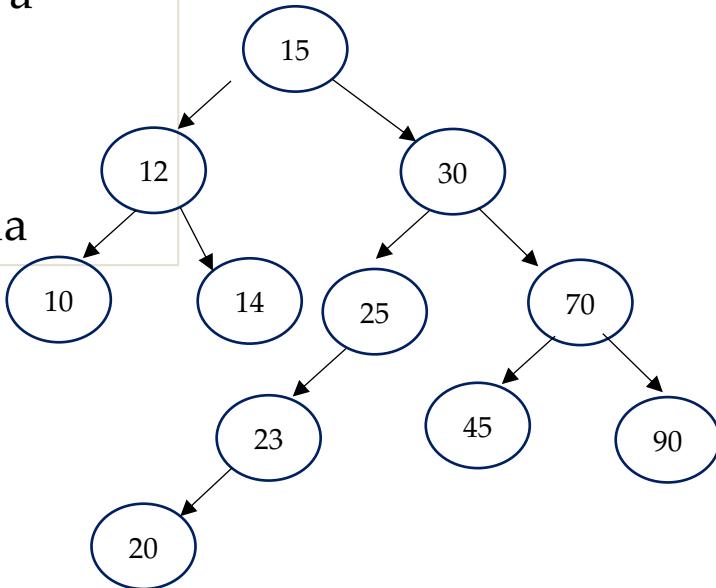
Rotacion doble:  
Primero a derecha  
Y luego  
A izquierda



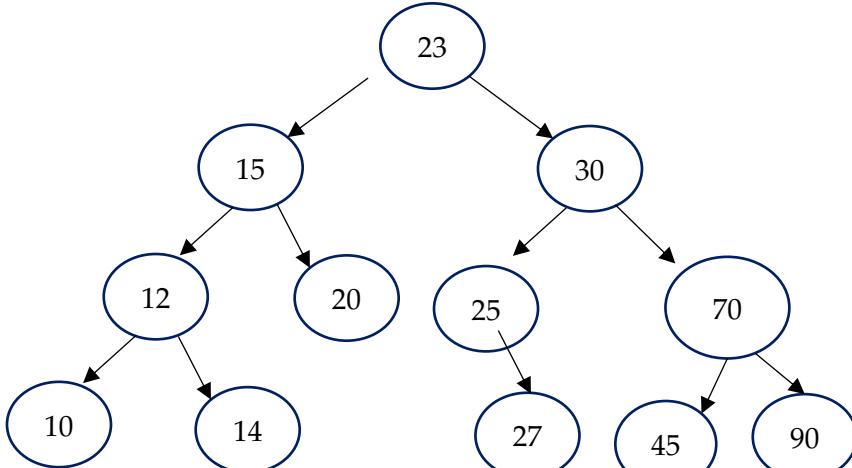
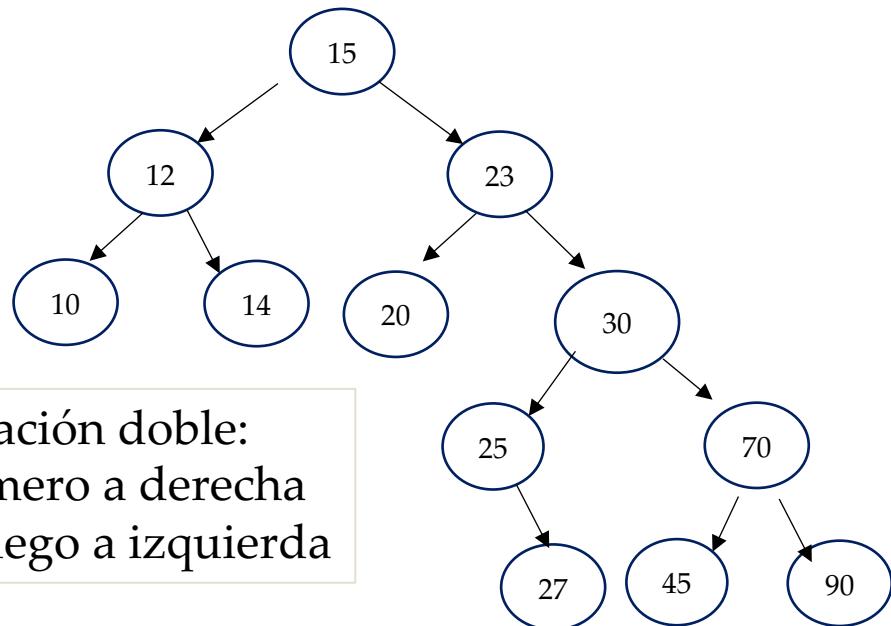
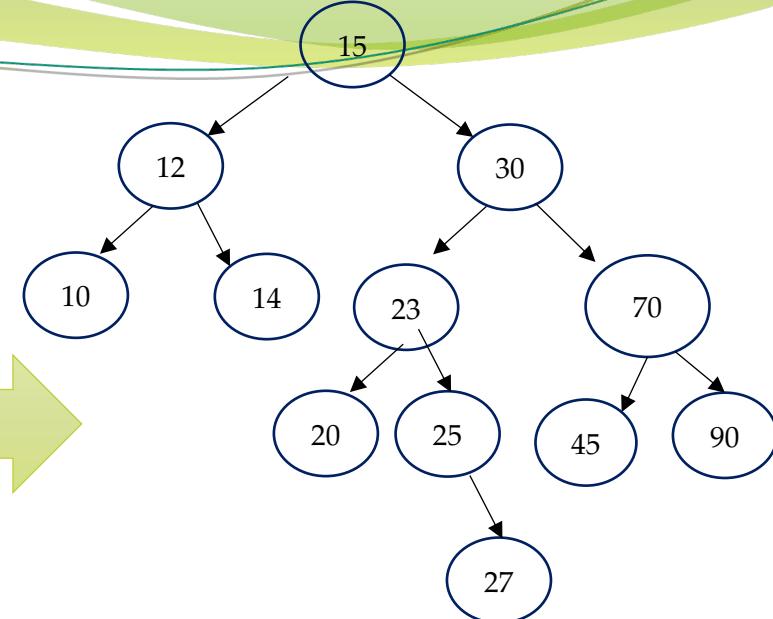
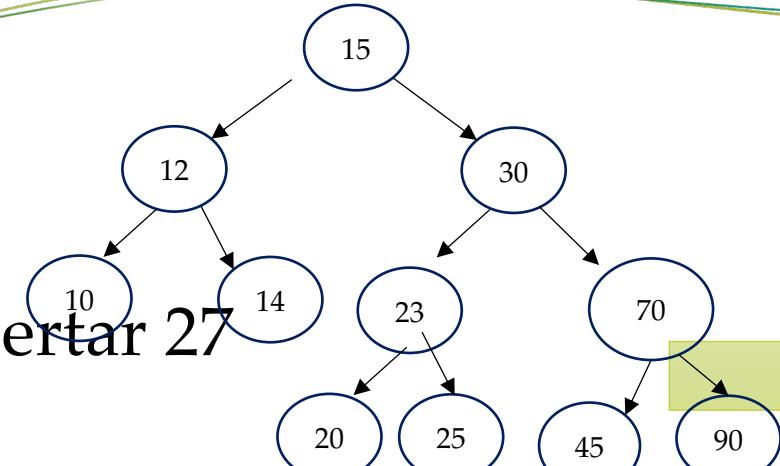
- Insertar 23



Rotación doble:  
Primero a  
Derecha  
Y luego  
A  
izquierda



- Insertar 27



Rotación doble:  
Primero a derecha  
Y luego a izquierda

## Consideraciones sobre la implementación

- Como en inserciones/borrados tenemos que calcular la diferencia entre la altura del subárbol izq y la altura del subárbol derecho => esta operación es MUY FRECUENTE.

Puede convenir almacenar en Node dicha información.

```
public class AVL<T extends Comparable<? super T>> implements  
BSTreeInterface<T> {  
...  
  
class Node implements NodeTreeInterface<T> {  
  
    private T data;  
    private Node left;  
    private Node right;  
  
    // para AVL  
    private int height;  
    ...  
}  
}
```

# Idea de cómo implementar la inserción (idea a completar...)

```
@Override  
public void insert(T myData) {  
    if (myData == null)  
        throw new RuntimeException("element cannot be null");  
  
    root= insert(root, myData);  
}
```

```

private Node insert(Node currentNode, T myData) {
    if (currentNode == null)
        return new Node(myData);

    if (myData.compareTo(currentNode.data) <= 0)
        currentNode.left= insert(currentNode.left, myData);
    else
        currentNode.right= insert(currentNode.right, myData);

    // agregado para AVL
    int i = currentNode.left==null?-1:currentNode.left.height;
    int d = currentNode.right==null?-1:currentNode.right.height;
    currentNode.height = 1 + Math.max(i, d);

    int balance = getBalance(currentNode);

    // Op: Left left
    if (balance > 1 && myData.compareTo(currentNode.left.data) <= 0)
        return rightRotate(currentNode);

    // Op: Right Right
    if (balance < -1 && myData.compareTo(currentNode.right.data) > 0)
        return leftRotate(currentNode);

    // Op: Left Right
    if (balance > 1 && myData.compareTo(currentNode.left.data) > 0) {
        currentNode.left = leftRotate(currentNode.left);
        return rightRotate(currentNode);
    }

    // Op: Right Left
    if (balance < -1 && myData.compareTo(currentNode.right.data) <= 0) {
        currentNode.right = rightRotate(currentNode.right);
        return leftRotate(currentNode);
    }

    return currentNode;
}

```

esta es la típica inserción en BST

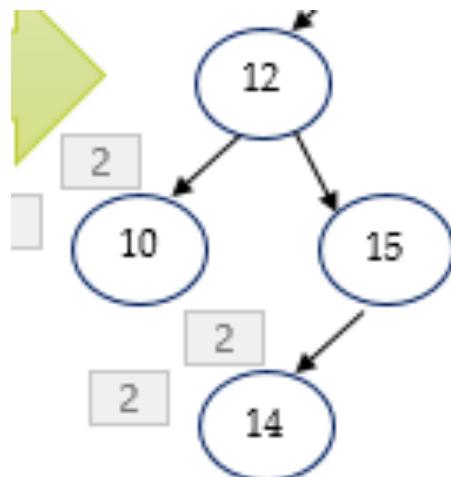
para AVL: actualizo altura

para AVL: calculo balance

esta es la típica inserción en BST

# Idea de cómo implementar la rotación.

Ej a izquierda con  
Pivote en 12



```
private Node leftRotate(Node pivot) {  
  
    Node newRoot = pivot.right;  
    pivot.right= newRoot.left;  
    newRoot.left = pivot;  
  
    // Update heights  
    pivot.height = Math.max(pivot.left.height, pivot.right.height) + 1;  
    newRoot.height = Math.max(newRoot.left.height, newRoot.right.height) + 1;  
  
    return newRoot;  
}
```

# AVL

Toda la eficiencia de un árbol BST está en su altura...  
¿Cuál es la altura de un AVL? ¿Cuál es su peor caso?

Rta: el árbol de Fibonacci. Es el AVL que presenta el peor desbalanceo posible.

# Árbol de Fibonacci

El Árbol de Fibonacci se define así:

1. Fibonacci de orden 0 es el **ÁRBOL NULO**
2. Fibonacci de orden 1 es un **NODO**
3. Fibonacci de orden  $h \geq 2$  es un árbol que tiene:
  - a) como hijo izquierdo un Fibonacci de orden  $h - 1$
  - b) como hijo derecho un Fibonacci de orden  $h - 2$ .

# TP 5D – Ejer 5.1

Dibujar el Arbol de Fibonacci  
de orden 6

Cuántos nodos tiene?  
Cuál es su altura?



Rta: Fibo(0) null

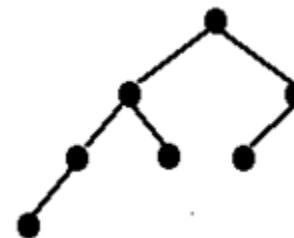
Fibo(1) •

Fibo(2) 

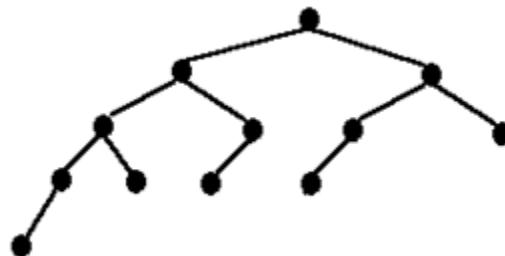
Fibo(3)



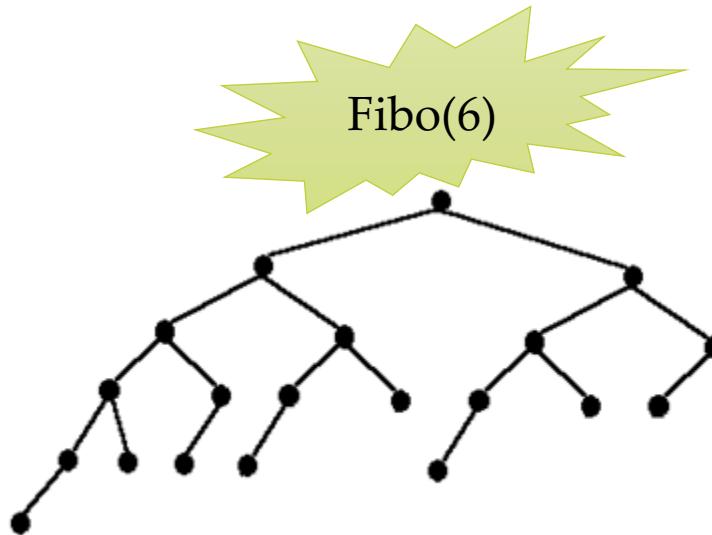
Fibo(4)



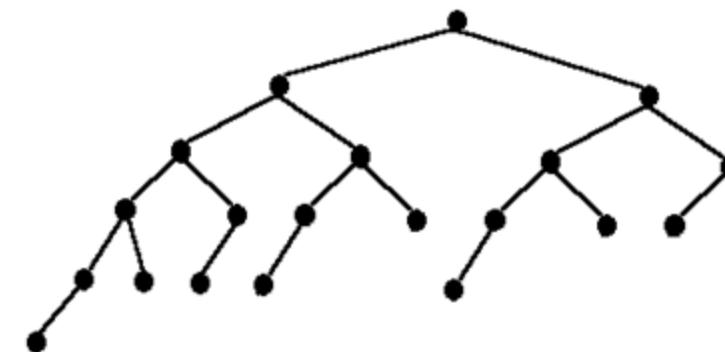
Fibo(5)



Fibo(6)



¿Cuántos nodos tiene el Fibonacci de orden 6? ¿Qué altura tiene?



Rta: 20 nodos y tiene altura 5

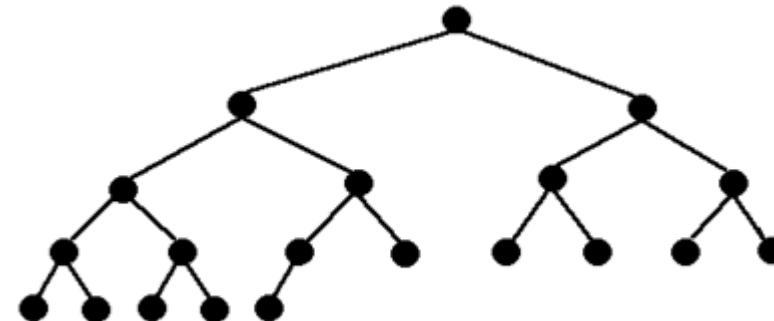
# TP 5D – Ejer 5.2

Cual sería la altura de uno completo (perfectamente balanceado) con esa misma cantidad de nodos?

Comparar con fibo



Rta: con 20 nodos este es el árbol perfectamente balanceado. Tiene altura 4.  
Solo un valor menos de altura!



Formalizando...

Sea un AVL de altura  $h$  **con la menor cantidad de nodos posibles en esa altura**

Avl de altura 0 tiene 1 nodo



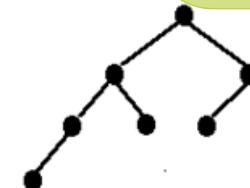
Avl de altura 1 tiene 2 nodos



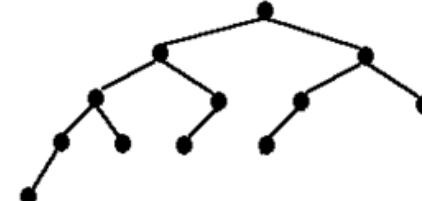
Avl de altura 2 tiene 4 nodos



AVL de altura 3 tiene 7 nodos



AVL de altura 4 tiene 12 nodos



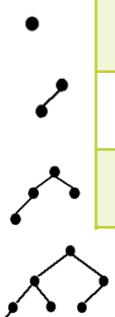
AVL de altura  $H$  tendrá  
1 nodo + cantnodos avl altura  $h-1$  +  
cantnodos avl altura  $h-2$

AVL de altura h tendrá

1 nodo + cantnodos avl altura h-1 + cantnodos avl  
altura h-2

¿Qué relación tiene con los números de Fibonacci?

$nrofibo(0)=0$ ,  $nfibo(1)=1$ ,  $nrofibo(2)=1$ ,  $nrofibo(3)=2$ ,  
 $nrofibo(4)=3$ ,  $nrofibo(5)=5$ ,  $nrofibo(6)=8$ ,  $nrofibo(7)=13$ ,  
etc



AVL altura h con menor cant de nodos	Cant nodos en relación a los números de fibo?
Altura 0 CantNodos=1	$nrofibo(altura+3)-1 = 1$
Altura 1 CantNodos=2	$nrofibo(altura+3)-1 = 2$
Altura 2 CantNodos=1 + 1 + 2 = 4	$nrofibo(altura+3)-1 = 4$
Altura 3 CantNodos=1 + 4 + 2 = 7	$nrofibo(altura+3)-1 = 7$
Altura 4 CantNodos=1 + 7 + 4 = 12	$nrofibo(altura+3)-1 = 12$
Altura 5 CantNodos=1 + 12 + 7 = 20	
...	
Altura h CantNodos=?	$nrofibo(h+3)-1$

Números de fibonacci :

$nrofibo(0)=0$ ,  $nrofibo(1)=1$ ,  $nrofibo(2)=1$ ,  $nrofibo(3)=2$ ,  $nrofibo(4)=3$ ,  $nrofibo(5)=5$ ,  
 $nrofibo(6)=8$ ,  $nrofibo(7)=13$ ,  $nrofibo(8)=21$ , etc

El árbol AVL que mínima cantidad de nodos tiene para cierta altura es el Árbol de Fibonacci (es más esparcido posible por construcción).

Si ese árbol con mínima cantidad nodos y altura  $h$  sabemos que tiene  $\text{CantNodos} = \text{nrosfibo}(h+3)-1$  donde  $h$  es su altura, entonces, otros árboles AVL con misma altura tendrán posiblemente más nodos

$$\text{CantNodos} \geq \text{nrosfibo}(h+3)-1$$

Pero los numeros de Fibonacci tienen una propiedad.  
Ver [https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)

Se sabe que  $nrofib(w) \geq \frac{a^w}{\sqrt{5}}$

Donde  $a$  es el “golden number”

$$a = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

Sea  $n$  la cantNodos de un avl de altura  $h$ ,  
donde sabemos que  $n \geq nrofib(h+3) - 1$

$$n \geq \frac{a^{h+3}}{\sqrt{5}} - 1$$

Despejar  $h$

Rta:  $h$  es  $O(\log n)$  O sea, para un AVL con  $n$  nodos, la altura está acotada por  $O(\log n)$

## Recapitulando

Peor caso de AVL con  $N=20$

$\Rightarrow$  Altura = 5

Perfectamente Balanceado con  $N=20$

$\Rightarrow$  Altura = 4



**Diferencia poco significativa en altura**

Mantener **AVL** es **menos costoso** que un completo (perfectamente balanceado)



**Preferible un árbol AVL**

- Hay diferentes algoritmos para garantizar que un árbol sea balanceado y esta propiedad sea invariante antes inserciones/borrados. Algunos más baratos que otros...

Ej: AVL trees, Red-Black Tree

Las transformaciones que hay que hacer para mantenerlo apropiado llegan en general  $O(\log n)$ , o sea, vale la pena...

Estudiar cómo son las inserciones (algorítmicamente, gráficamente) en Red Black Tree que es otro tipo de algoritmo que garantiza operaciones en  $O(\log N)$ .

Este árbol es el que tiene implementado Java.

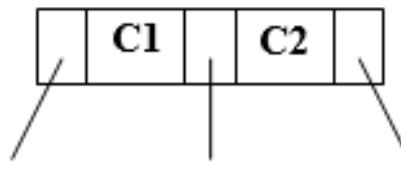
# Otra familia de árboles

## *Árbol Multicamino M-ario (orden M)*

Los nodos guardan hasta  $M-1$  claves de información, con un máximo de  $M$  hijos. Cada clave  $C_i$  de un cierto nodo será tal que las claves almacenadas en su subárbol izquierdo serán menores y las almacenadas en su subárbol derecho serán mayores que él.

*Ejemplo:*

Un árbol multicaminos con  $M=3$  podría ser:



Este nodo tiene dos claves  $C1 < C2$  y además todas las claves del subárbol izquierdo de  $C1$  serán menores que él y las de su subárbol derecho (que es el mismo que el izquierdo de  $C2$ ) serán mayores que él. Análogamente ocurre para  $C2$ .

## *Árboles Multicaminos Balanceados*

El equilibrio perfecto resulta muy costoso de mantener y es poco práctico.

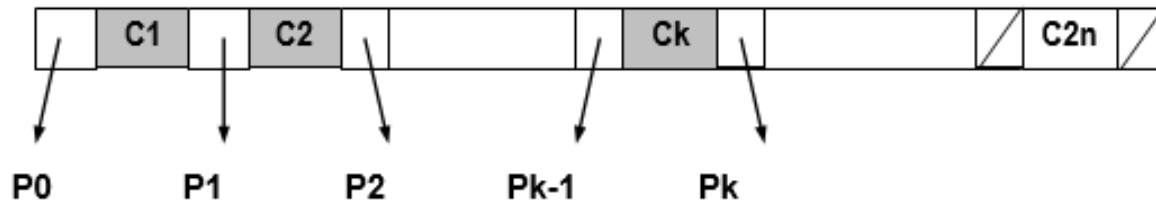
En 1970, R. Bayer y E. M. Mc Creight postularon un **criterio razonable** que permite implementar algoritmos relativamente sencillos para búsquedas, inserciones y eliminaciones. Para mantener éstos árboles multicaminos balanceados se utiliza una estructura subyacente de la **familia de los árboles B**, que veremos a continuación

# Árbol B de Orden N

Un árbol B de orden N es un árbol de búsqueda (ordenado) que cumple con los siguientes axiomas:

- Cada nodo contiene a lo sumo  $2 * N$  claves.
- Cada nodo, excepto la raíz, contiene por lo menos N claves.
- Cada nodo o es hoja o tiene  $M+1$  descendientes donde M es el número de claves que posee realmente ese nodo
- Todas las hojas están al mismo nivel
- En cuanto al orden: si un nodo tiene  $c_1 \ c_2 \ ... \ c_m$  elementos  $c_1 < c_2 < \dots < c_m$ , pero además para cada  $c_i$  ( $1 \leq i \leq m$ ) los elementos del subárbol izquierdo de  $c_i$  son menores que  $c_i$  y los elementos del subárbol derecho de  $c_i$  son mayores que  $c_i$ .

El nodo genérico de un árbol B de orden N, será de la forma:



En este caso el nodo posee realmente k claves (presentes con información) y por lo tanto k+1 punteros (los demás son nulos).

## Algoritmo de Búsqueda

- Buscamos la clave X en un nodo. Para ello lo **recorremos secuencialmente** desde C<sub>1</sub> hasta C<sub>k</sub>, siendo k el número de claves que realmente posee dicho nodo, hasta que se den alguno de estos casos:
  - Si X < C<sub>1</sub>, como en el nodo las claves están ordenadas no tiene sentido seguir buscando en ese nodo, luego sigo buscando en el subárbol apuntado por P<sub>0</sub>
  - Si X=C<sub>i</sub> para algún i<=k entonces lo encontré
  - Si C<sub>i</sub><X<C<sub>i+1</sub> para algún i prosigo en la búsqueda en el subárbol apuntado por P<sub>i</sub>
  - Si C<sub>k</sub> < X siendo k la cantidad de claves que posee, entonces sigo la búsqueda en el subárbol apuntado por C<sub>k</sub>

Si en algún caso el puntero por donde hay que seguir la búsqueda fuera null, entonces el elemento buscado no está.

## Algoritmo de Inserción

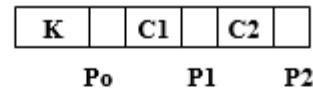
Si se quiere insertar una clave X en un árbol B de orden N, se procede de la siguiente manera:

- La inserción siempre se hace **en las hojas** (para poder detectar si el nodo a insertar ya está presente o no)
- Para insertar se coloca el elemento X en la hoja que corresponda (el nodo debe estar ordenado)
- Si el elemento nuevo hace que la cantidad nueva k sea mayor que el  $2^*n$  permitido, el nodo se abre en dos, subiendo la clave del medio al nodo antecesor de dicho nodo. Este algoritmo es recursivo hasta la raíz, o sea si al ubicar la clave del medio en el nodo antecesor ocasiona que el nodo viole la condición de árbol B de orden n ( $k>2^*n$ ) el procedimiento de repite.

# TP 5D – Ejercicio 9



Insertemos claves en un árbol B de orden 1. Sus nodos son de la forma:



- El mínimo k será 1, excepto para la raíz
- El máximo k será 2
- Si se tienen k elementos se tendrán k+1 punteros.

Insertar las claves 100, 80, 40, 20 y 60

- Insertar 100

- Insertar 100

100	
-----	--

- Insertar 100

100	
-----	--

- Insertar 80

- Insertar 100

100	
-----	--

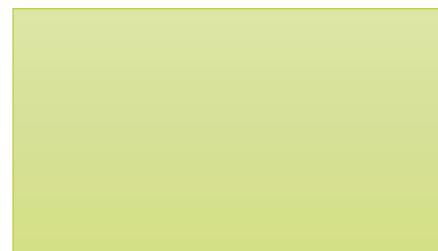
- Insertar 80

80	100
----	-----

80	100
----	-----

- Insertar 40

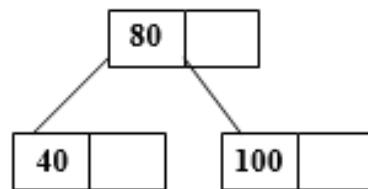
Quedaria ( 40 , 80 , 100 ) pero es imposible porque no podía tener más de 2 elementos, luego el nodo se parte en dos y sube el del medio como padre.

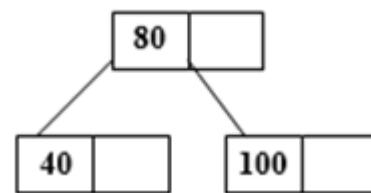


80	100
----	-----

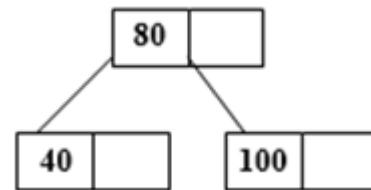
- Insertar 40

Quedaria ( 40 , 80 , 100 ) pero es imposible porque no podía tener más de 2 elementos, luego el nodo se parte en dos y sube el del medio como padre.

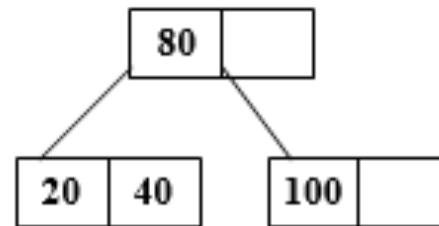


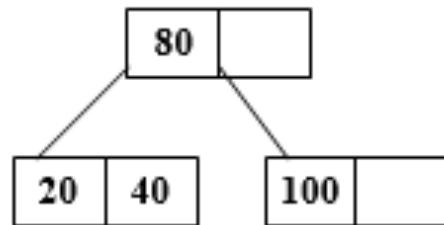


- Insertar 20

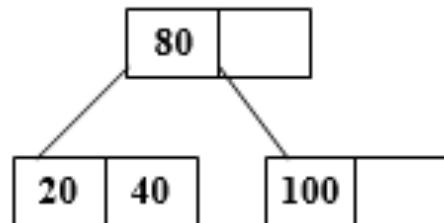


- Insertar 20



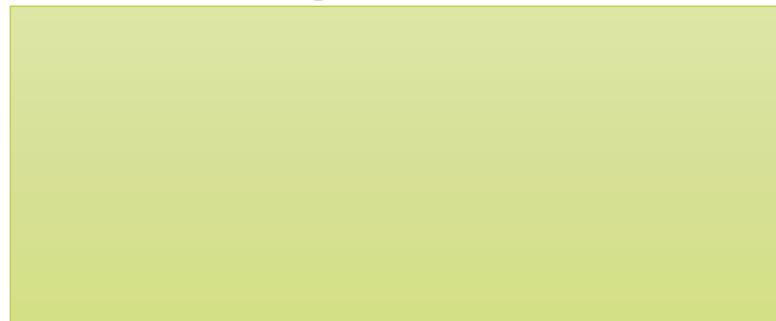


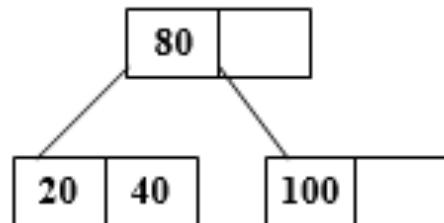
- Insertar 60 ¿Donde va?



- Insertar 60

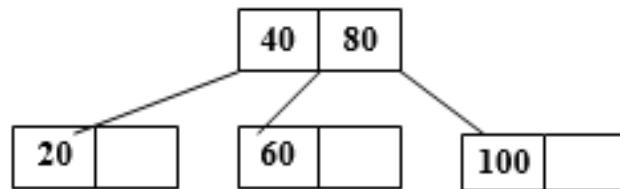
Como 20 , 40 , 60 no es nodo posible en un árbol B de orden 1, se partitiona y la clave 40 forma parte del nodo de arriba junto con el 80.



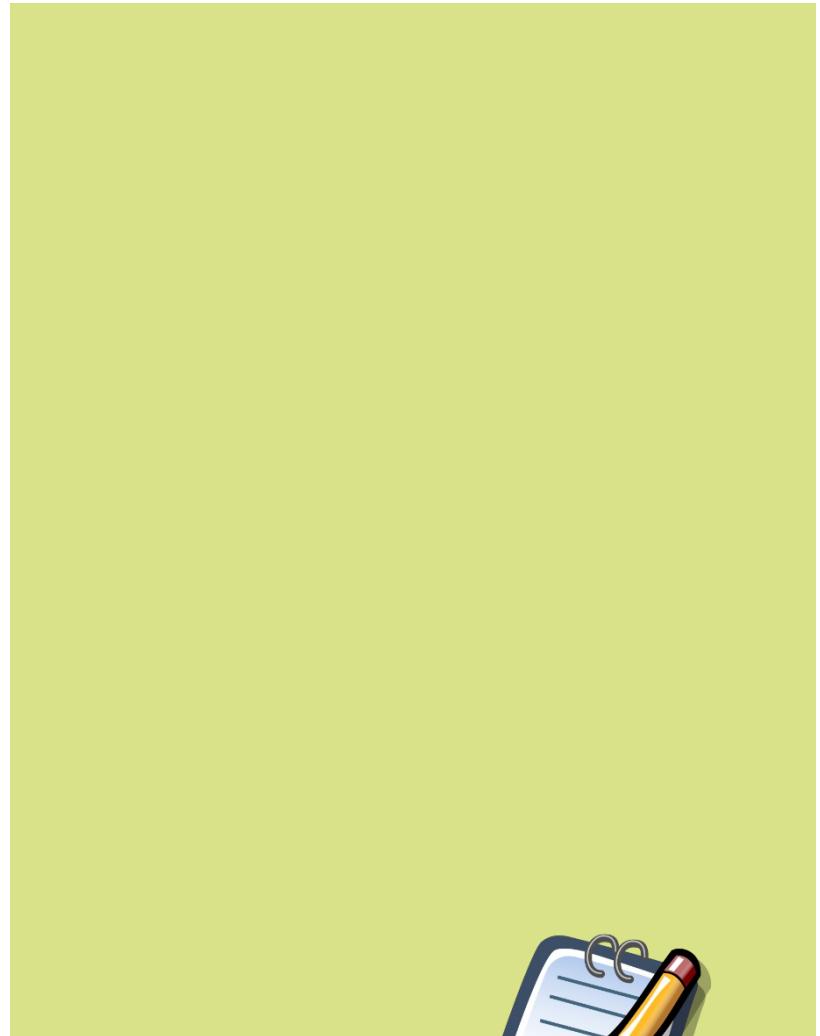


- Insertar 60

Como ( 20 , 40 , 60 ) no es nodo posible en un árbol B de orden 1, se partitiona y la clave 40 forma parte del nodo de arriba junto con el 80.



# TP 5D – Ejercicio 10



Insertaremos las siguientes claves en un árbol B de orden 1: 10, 20, 30, 50, 70, 100, 150, 130, 120, 220, 180, 200, 240, 140, 160

Insertaremos las siguientes claves en un árbol B de orden 1: 10, 20, 30, 50, 70, 100, 150, 130, 120, 220, 180, 200, 240, 140, 160

- Insertar 10

10	
----	--

- Insertar 20

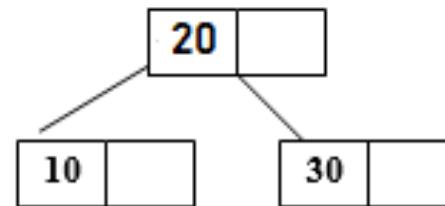
10	20
----	----

10	20
----	----

- Insertar 30

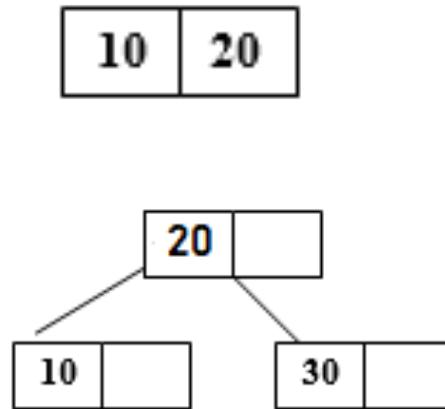
10	20
----	----

- Insertar 30

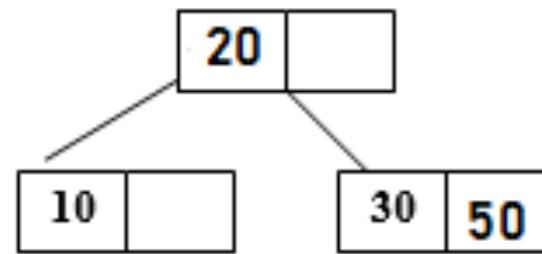


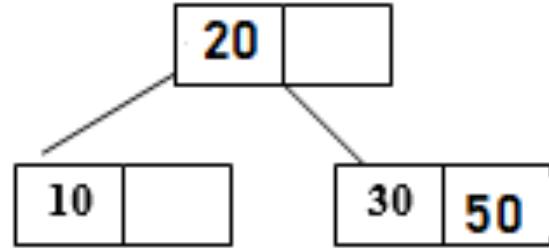
- Insertar 50

- Insertar 30

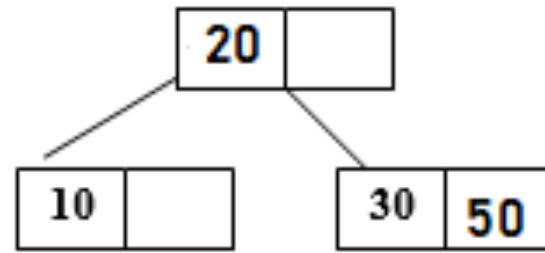


- Insertar 50

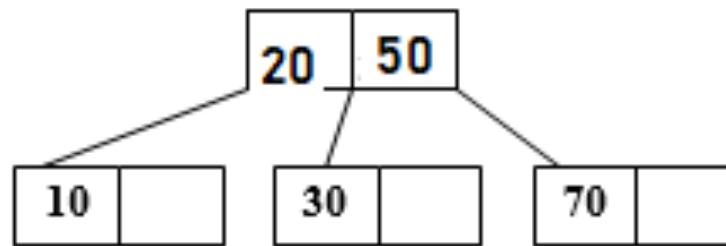




- Insertar 70

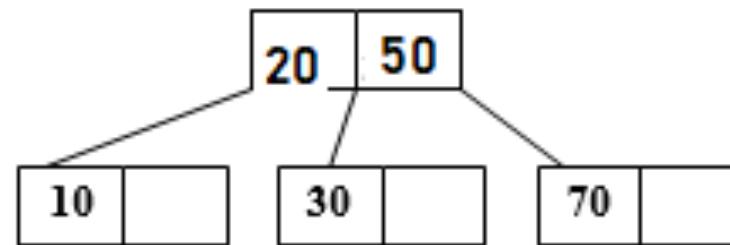


- Insertar 70

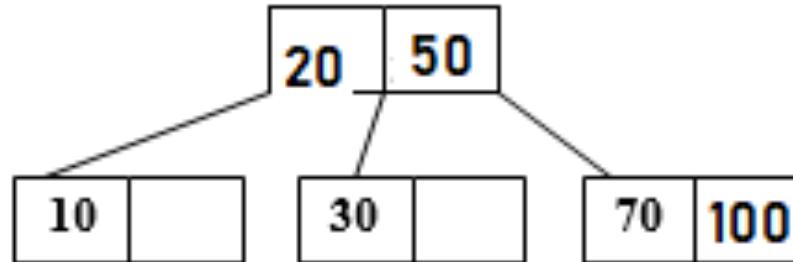


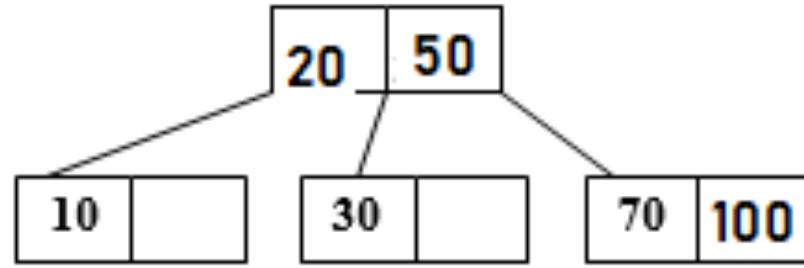
- Insertar 100

- Insertar 70

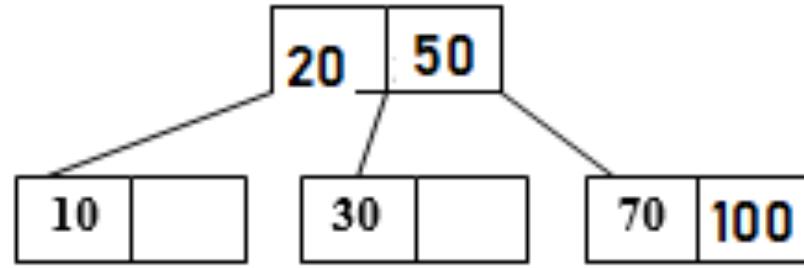


- Insertar 100

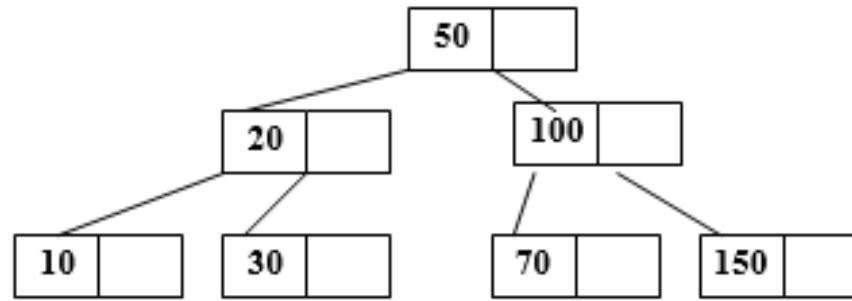




- Insertar 150

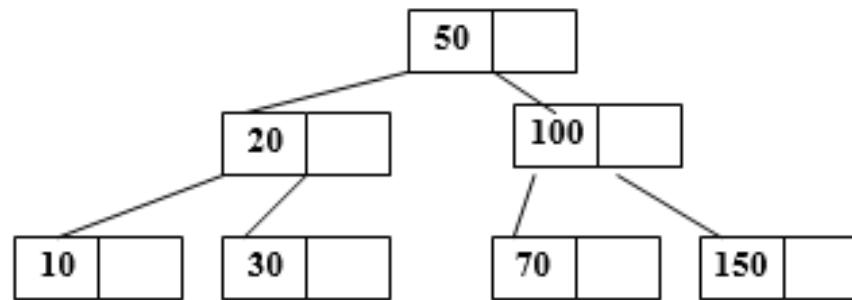


- Insertar 150

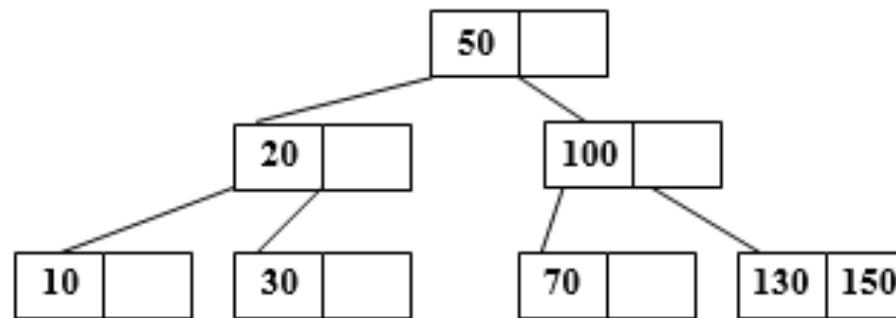


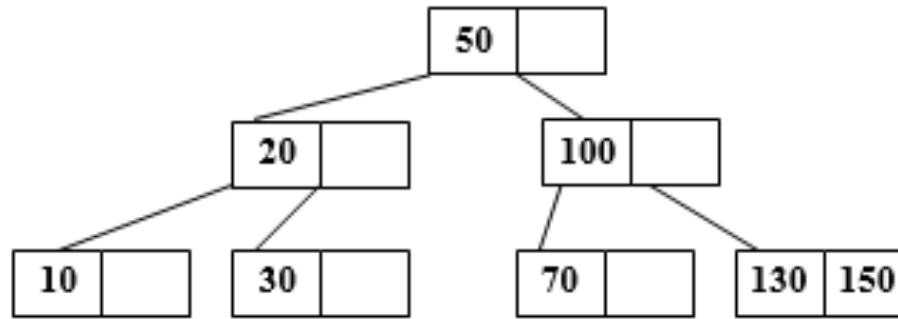
- Insertar 130

- Insertar 150

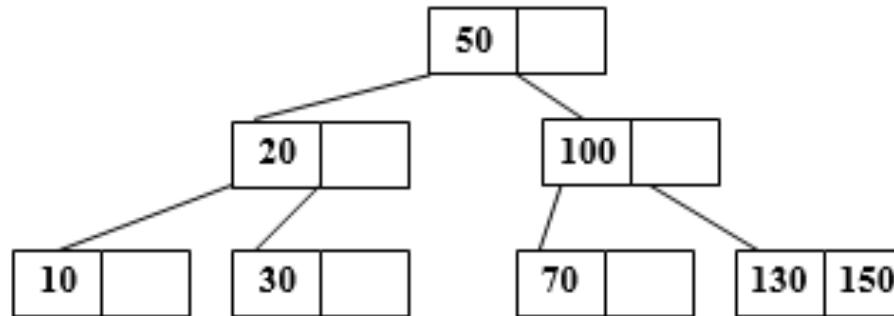


- Insertar 130

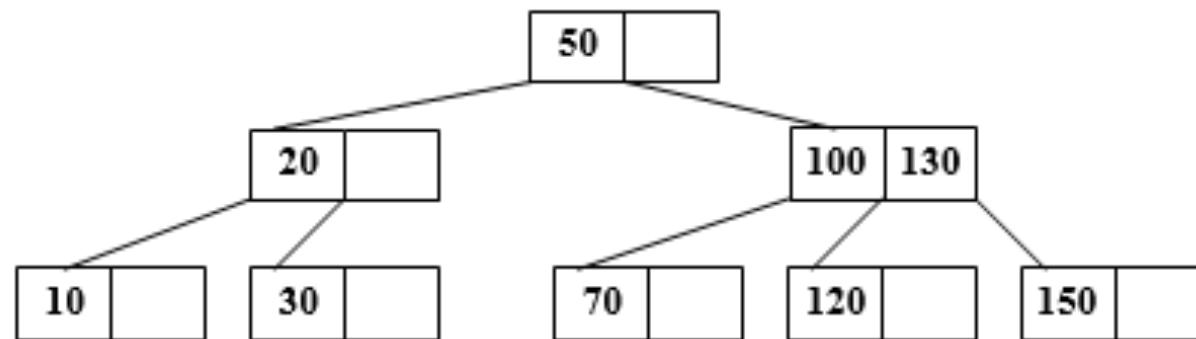




- Insertar 120

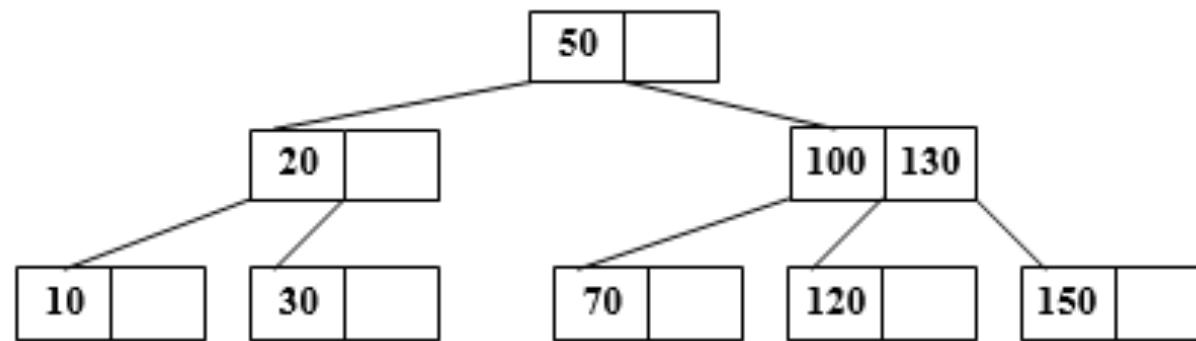


- Insertar 120

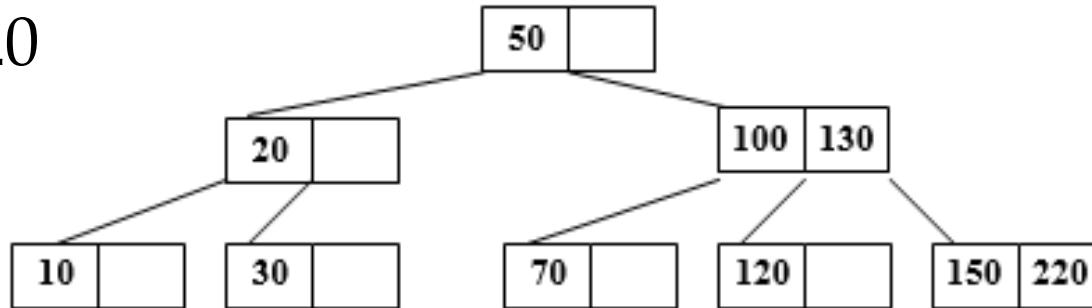


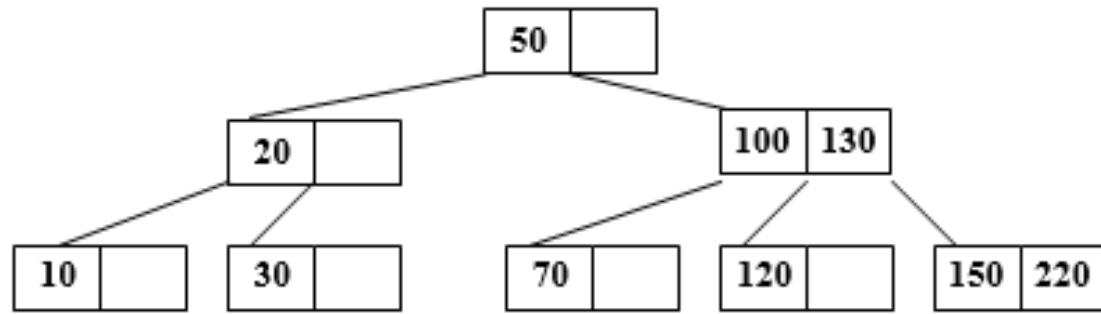
- Insertar 220

- Insertar 120

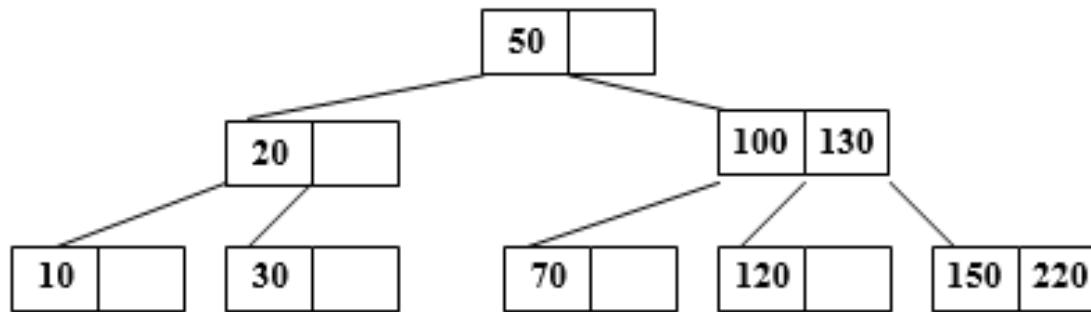


- Insertar 220

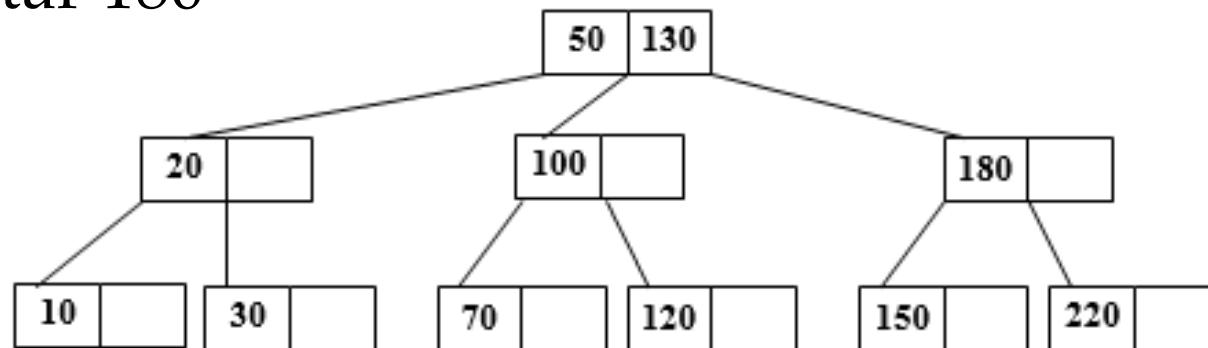




- Insertar 180

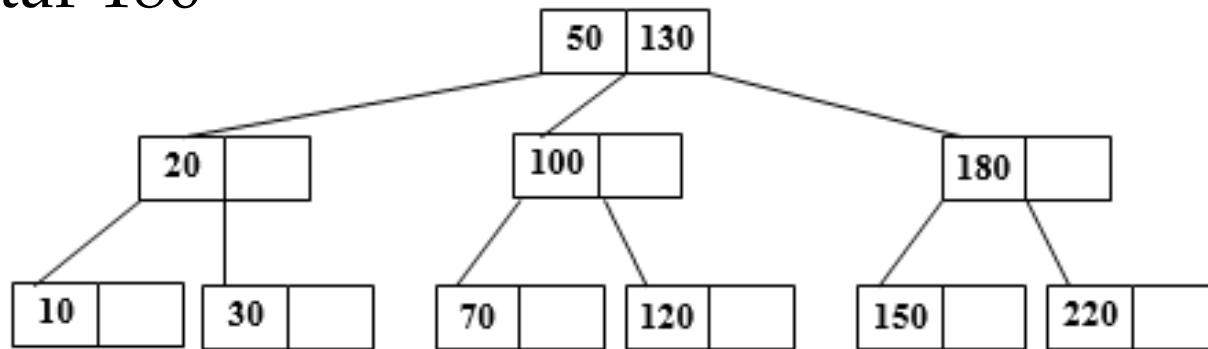


- Insertar 180

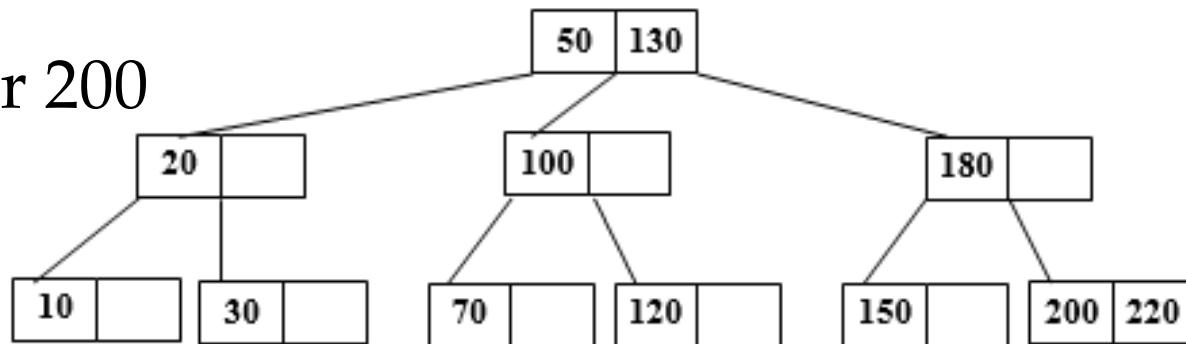


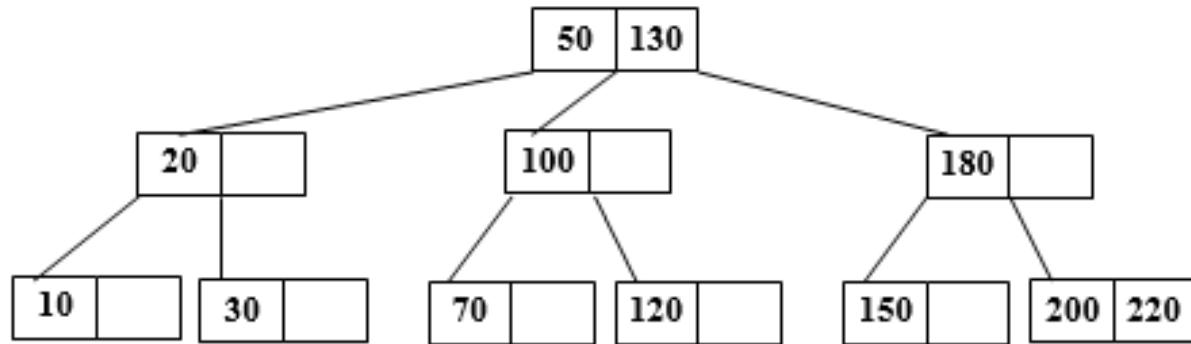
- Insertar 200

- Insertar 180

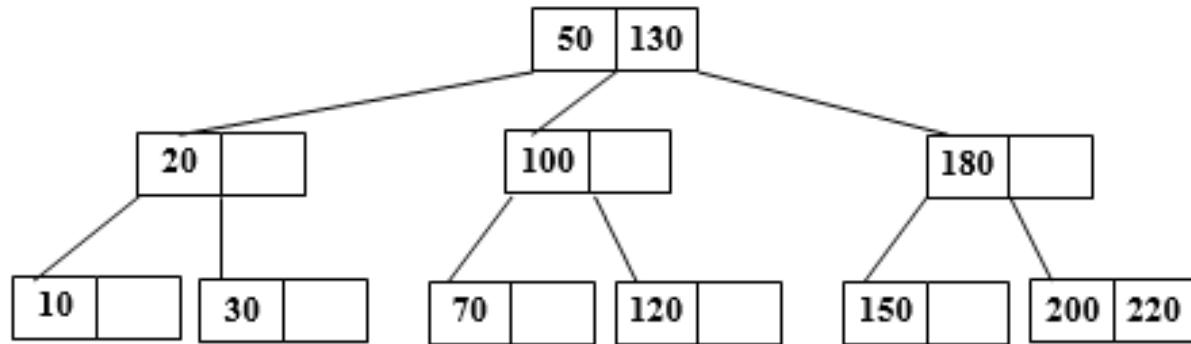


- Insertar 200

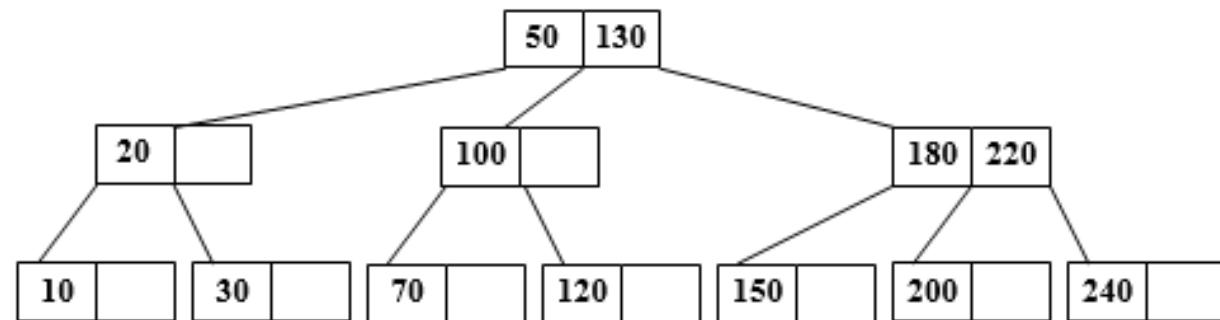




- Insertar 240

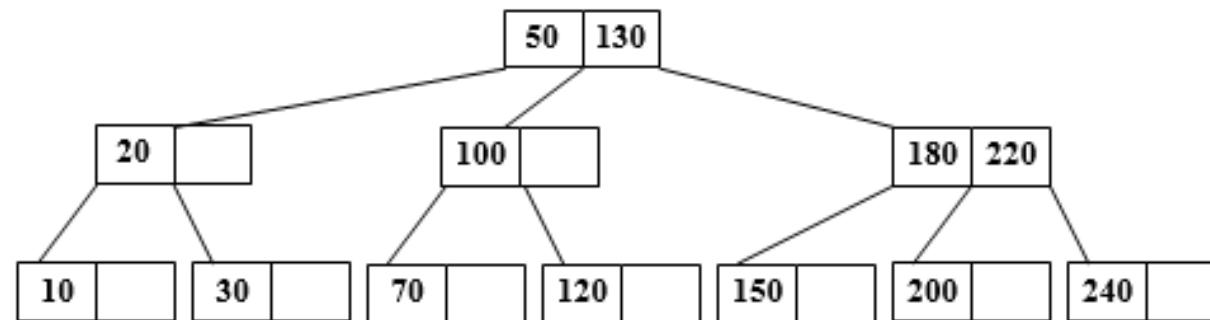


- Insertar 240

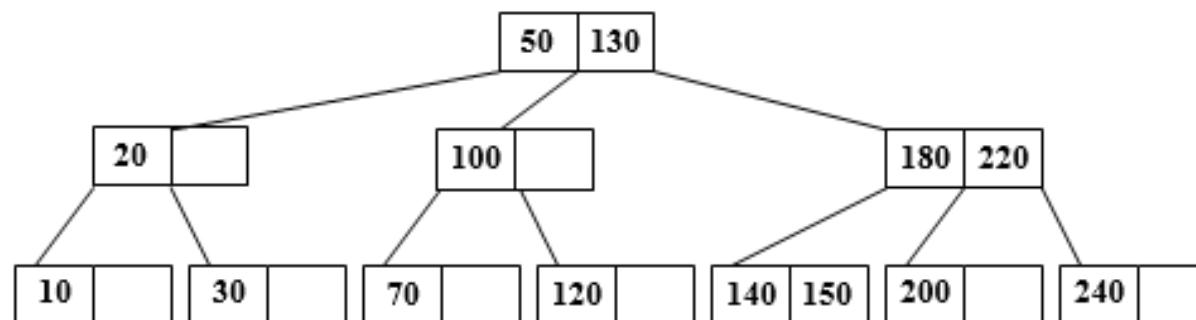


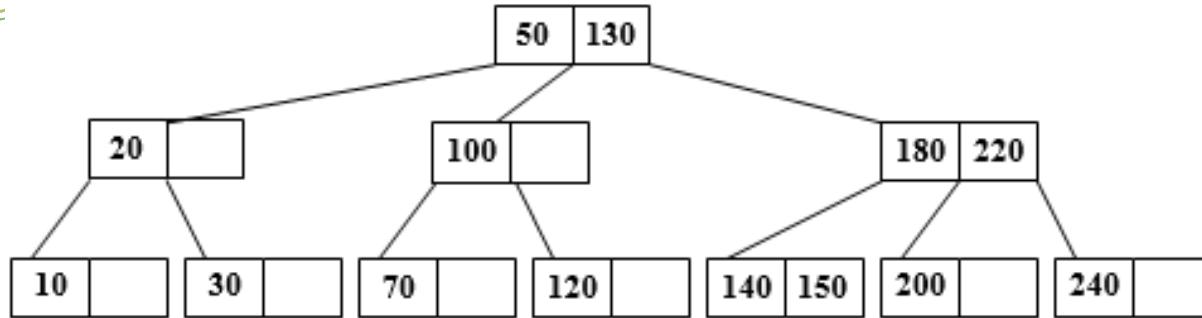
- Insertar 140

- Insertar 240

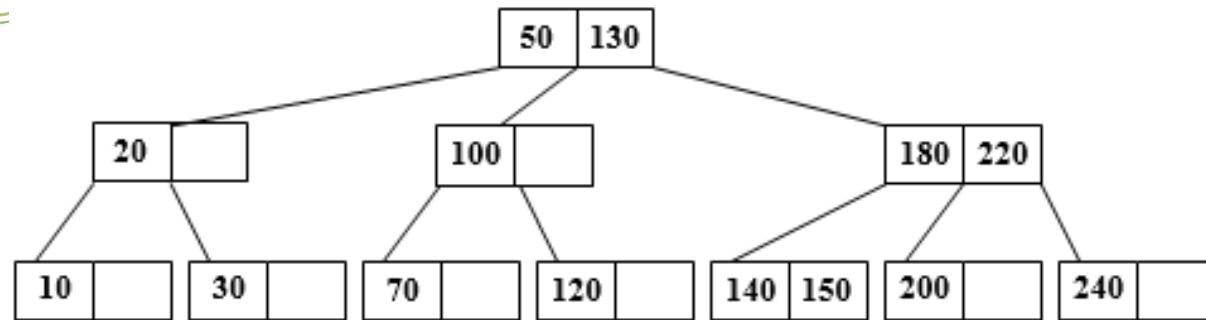


- Insertar 140

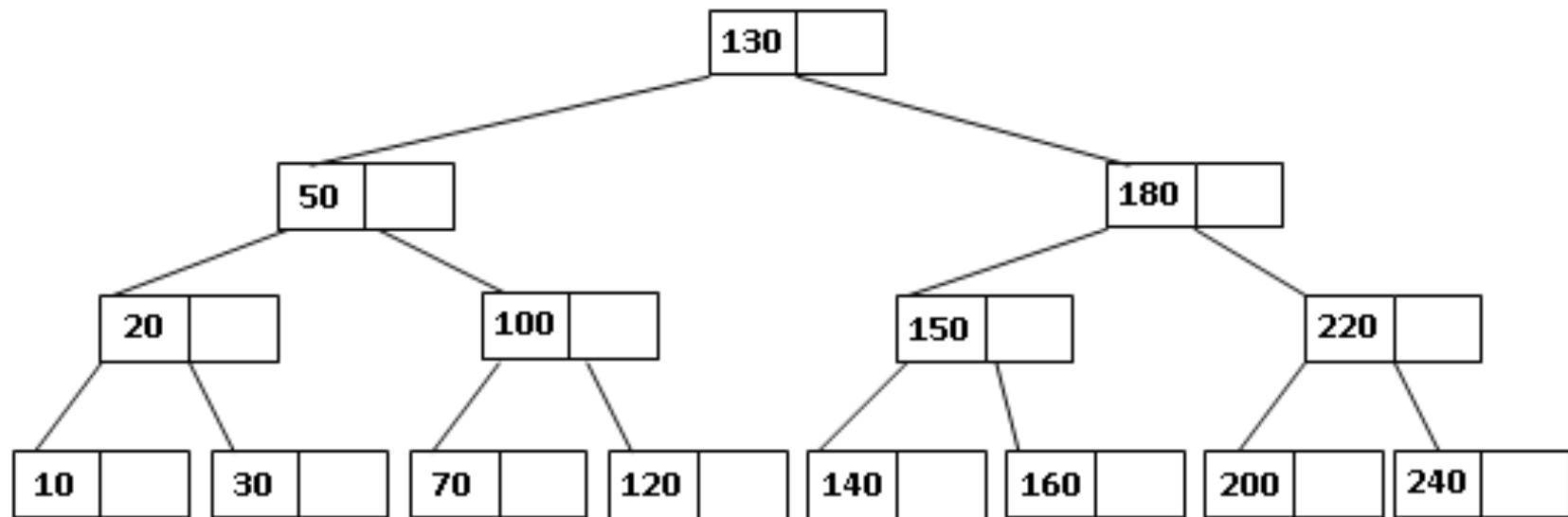




- Insertar 160



- Insertar 160



# TP 5D – Ejerc 11



# B Tree

Insertar en un árbol **B de orden 2**, las claves ordenadas del 0 al 19 inclusive

# TP 5D – Ejercicio 12



## **Crear un Proyecto Java y usar la implementación publicada en:**

[https://github.com/phishman3579/java-algorithms-implementation/blob/master/src/com/jwetherell/algorithms/data\\_structures/BTree.java](https://github.com/phishman3579/java-algorithms-implementation/blob/master/src/com/jwetherell/algorithms/data_structures/BTree.java)

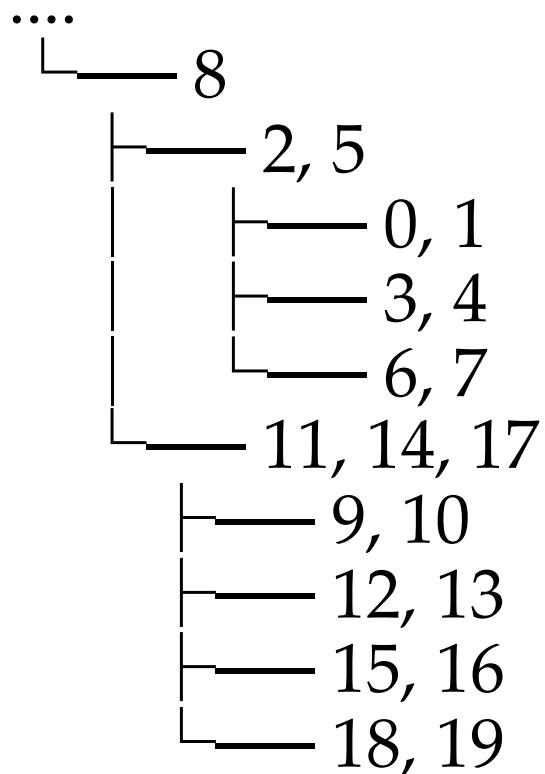
[https://github.com/phishman3579/java-algorithms-implementation/blob/master/src/com/jwetherell/algorithms/data\\_structures/interfaces/ITree.java](https://github.com/phishman3579/java-algorithms-implementation/blob/master/src/com/jwetherell/algorithms/data_structures/interfaces/ITree.java)

# B tree

Chequear el resultado anterior, es decir:

```
public class Test {  
  
    public static void main(String[] args) {  
        BTree<Integer> st = new BTree<>(2);  
  
        for(int rec= 0; rec < 20; rec++)  
        {  
            st.add(rec);  
            System.out.println( st.toString() );  
  
            System.out.println("....");  
  
        }  
    }  
}
```

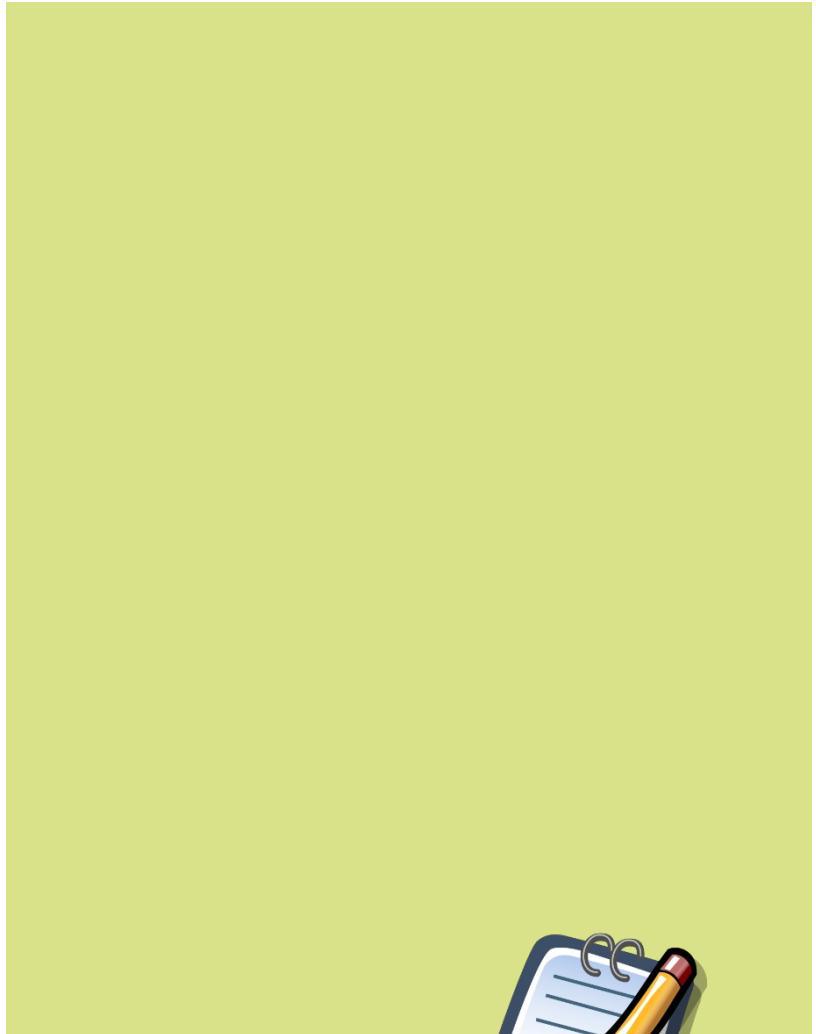
# Resultado



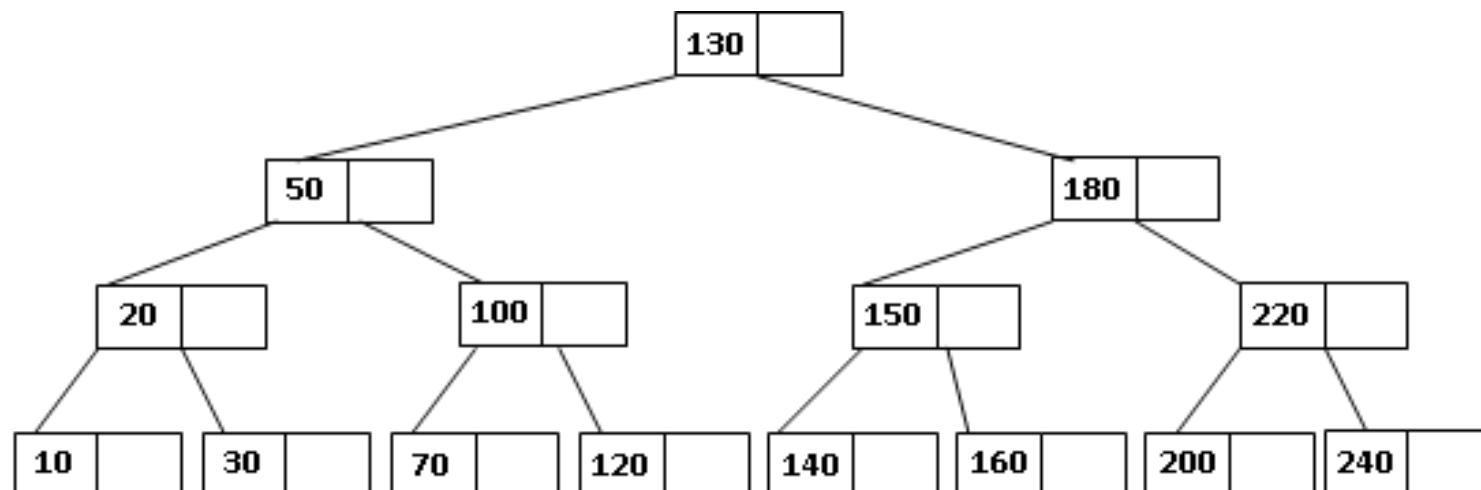
## Operación de Borrado

- Si no encuentra en un nodo hoja, se lo reemplaza por una clave lexicográficamente adyacente, por ejemplo el sucesor in order y se lo elimina de dicha hoja. Si fuera hoja se lo elimina directamente.
- Luego, para la hoja que colaboró el borrado se analiza si cumple las condiciones de árbol B de orden N. Si ha quedado en rojo (tiene menos elementos que los permitidos), se une dicho nodo con su hermano y medio antecesor (el cual es eliminado del nodo al cual pertenece, porque acude en ayuda de su hijo) armando un sólo nodo. Se verifica si cumple las condiciones de árbol B de orden N, y sino se lo partitiona subiendo el elemento del medio. Después se analiza qué sucede con el nodo donde estaba su medio antecesor, y se sigue el proceso recurrentemente hasta llegar a la raíz.

# TP 5C – Ejercicio 12



Dado el ultimo árbol de Orden 1 obtenido, eliminar  
200, 220, 50

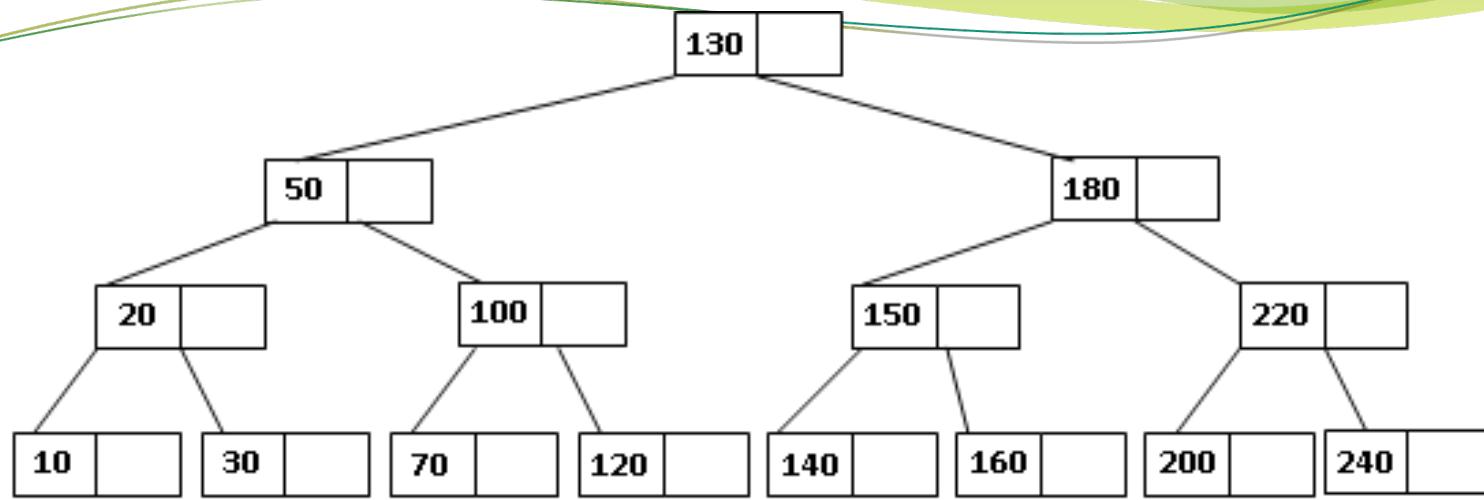


- Eliminar 200

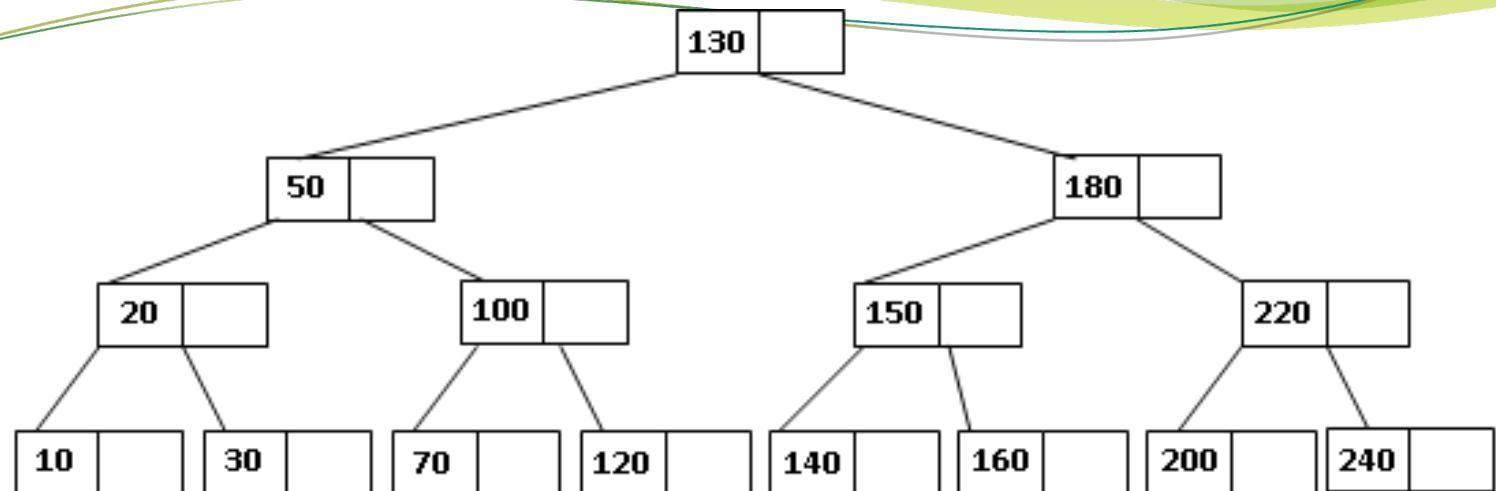
Como el nodo que contenía al 200 **queda en rojo**, se une con su hermano y baja su medio antecesor formando un solo nodo (220, 240), que se transforma en un nodo de árbol B de orden 1.

Pero el nodo donde estaba el medio antecesor 220 **queda en rojo**, entonces se une con su hermano y baja el medio antecesor formando un solo nodo (150, 180). ¿Es éste un nodo de un árbol B de orden 1? Sí, pero ¿qué pasa con el nodo donde estaba el medio antecesor? **Queda en rojo !!!** Entonces se une con su hermano y baja el medio antecesor formando un solo nodo (50, 130). Es éste un nodo de árbol B de orden 1 ? Sí.

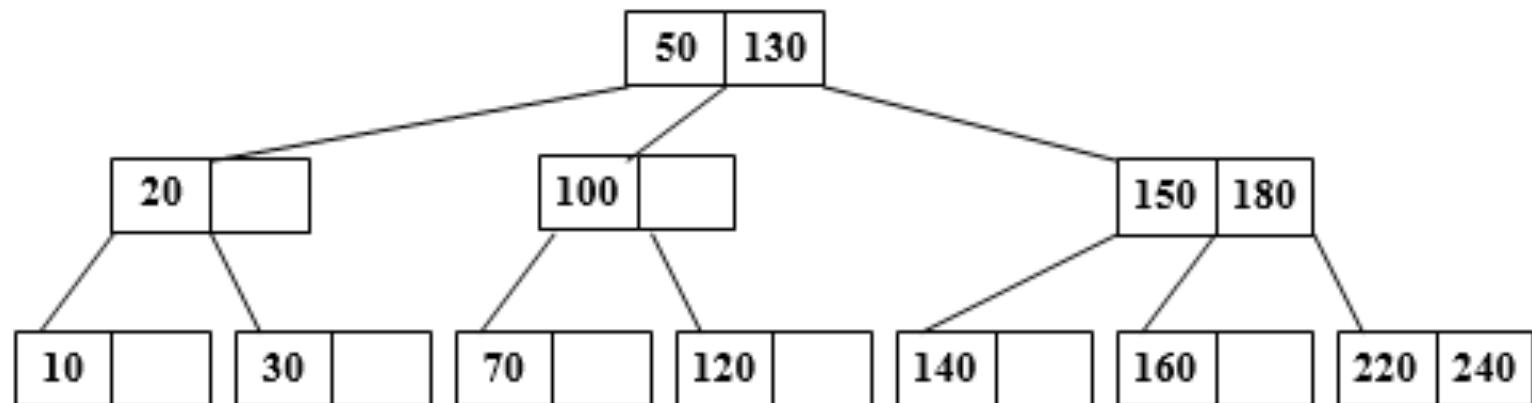
El árbol B, después de la eliminación del 200, queda así:

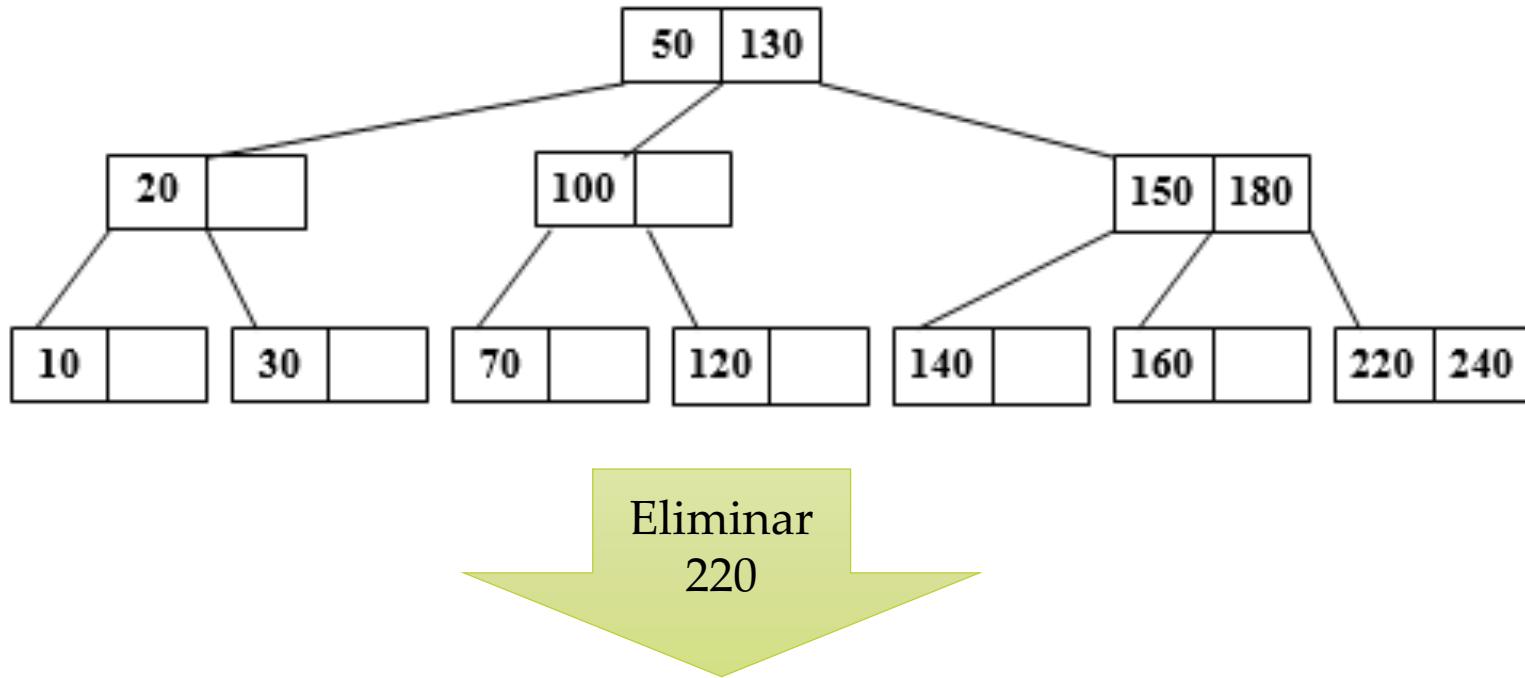


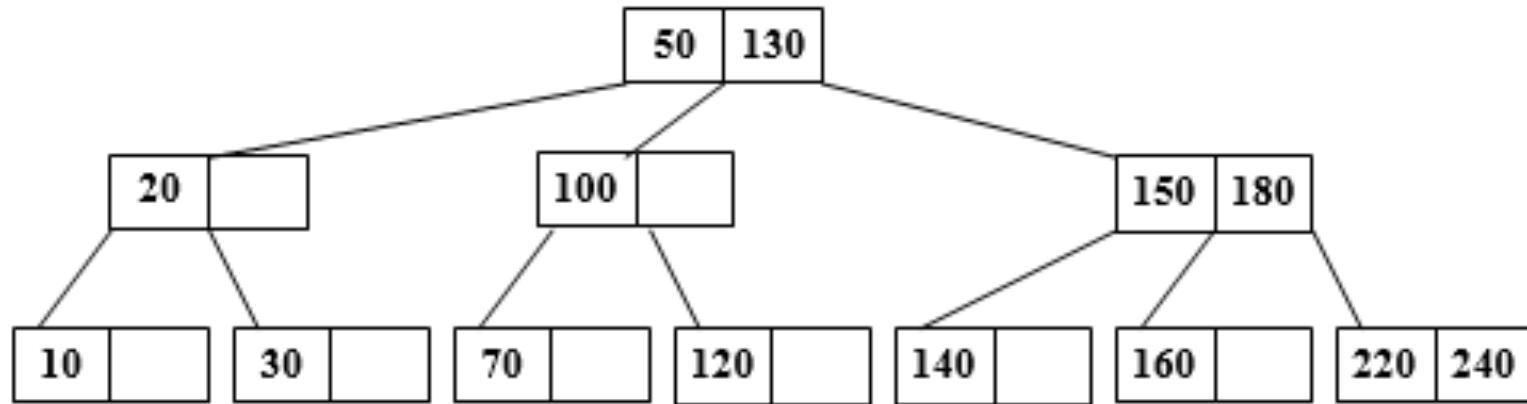
Eliminar  
200



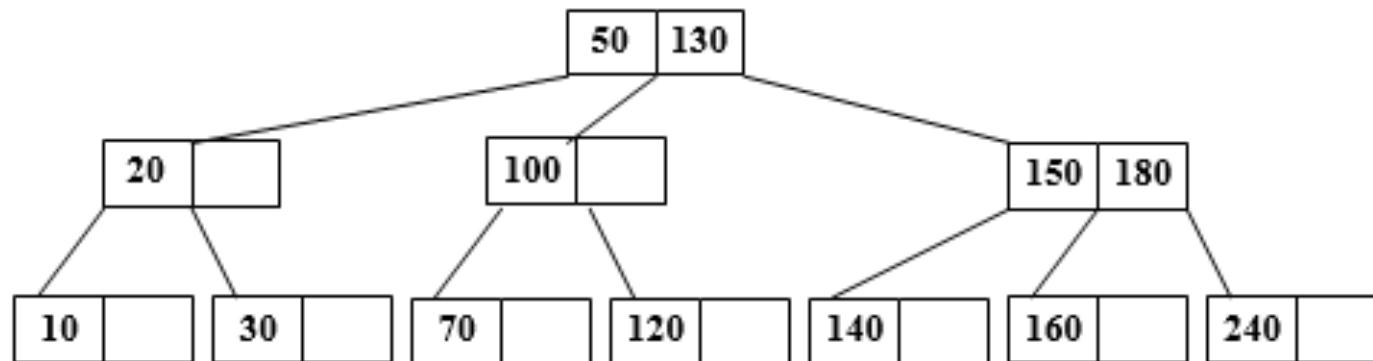
Eliminar  
200







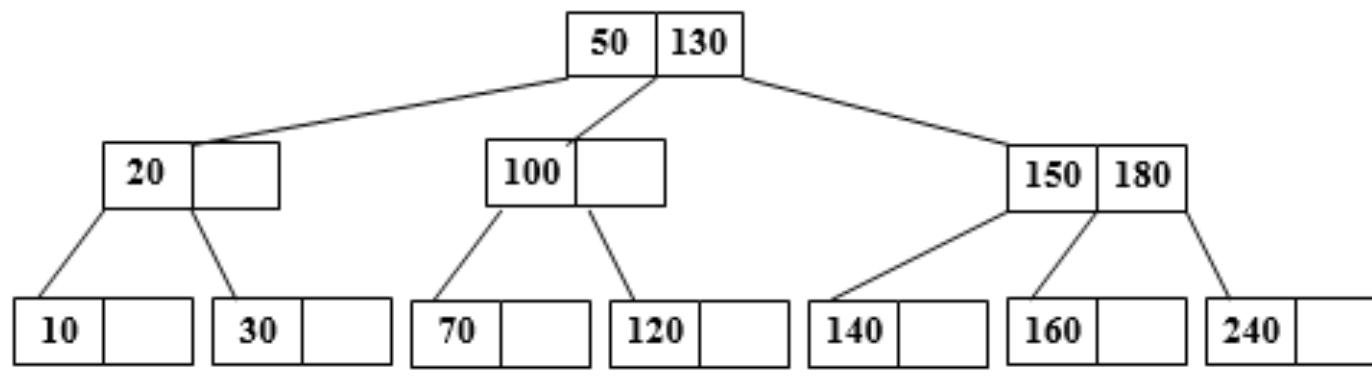
Eliminar  
220



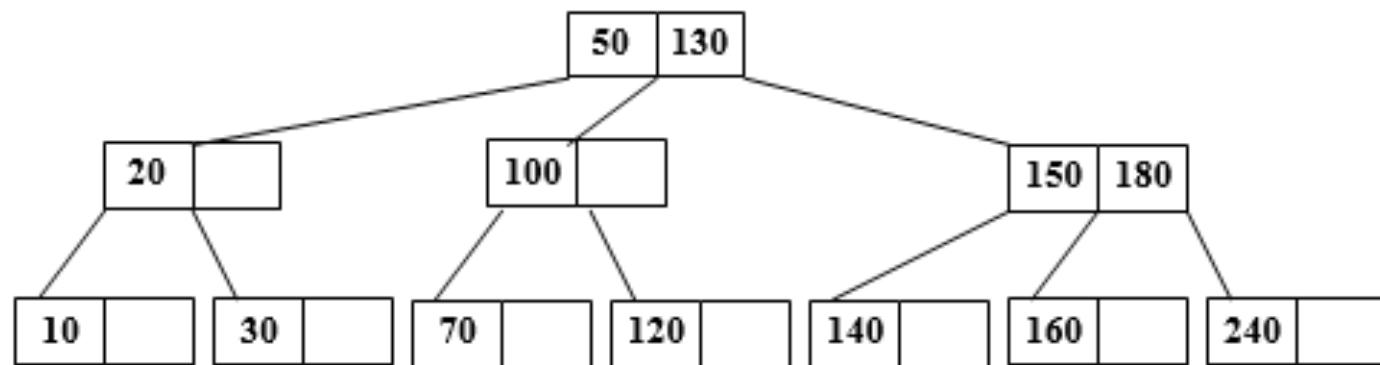
- Eliminamos 50

Como **no es un nodo hoja** se lo reemplaza por una clave lexicográficamente adyacente, **por ejemplo su sucesor in order: 70**. Pero el nodo donde se encontraba la clave 70 es una hoja que **queda en rojo**, luego se une con su hermano y baja el medio antecesor formando un nodo (100, 120). Si bien ese nodo es correcto, el nodo donde estaba el 100 **queda en rojo**, luego, se une con su hermano y su medio antecesor.

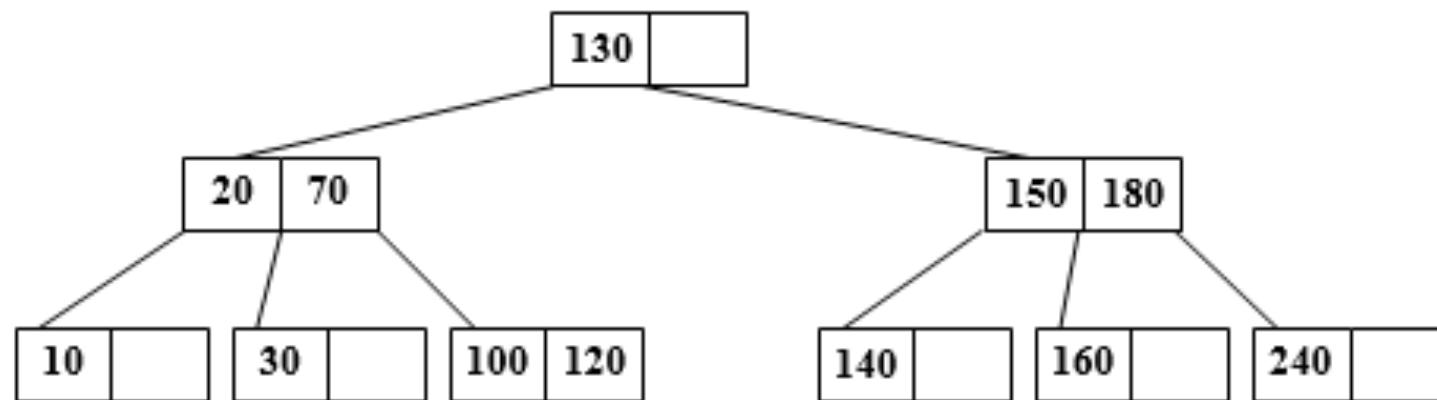
Como tiene dos medio hermanos (20 y 150) hay que elegir una convención, por ejemplo, si tiene medio hermano izquierdo se toma ése, sino el de la derecha. No importa la convención utilizada, siempre que se la respete durante todo el algoritmo. Tomaremos esta convención, por lo cual queda el nodo (20, 70). ¿Es éste un nodo de un árbol B de orden 1? Sí. ¿Qué pasa con el nodo donde estaba el medio antecesor? Sigue siendo un nodo de un árbol B de orden.



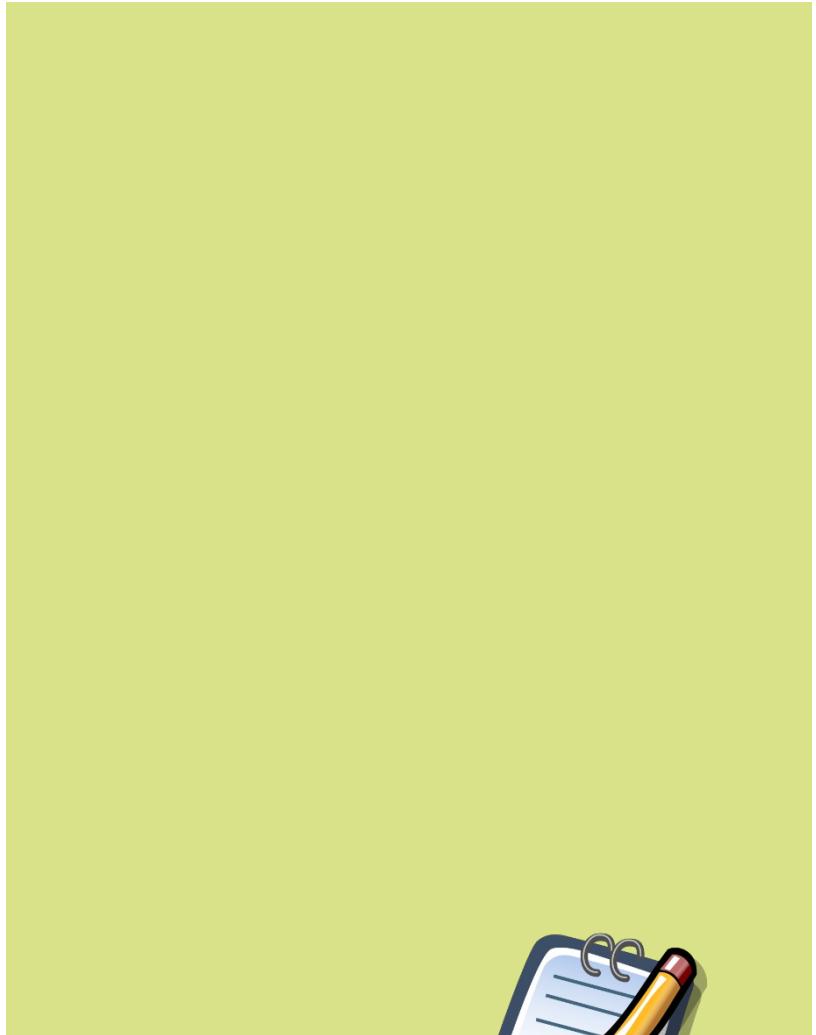
Eliminar  
50



Eliminar  
50

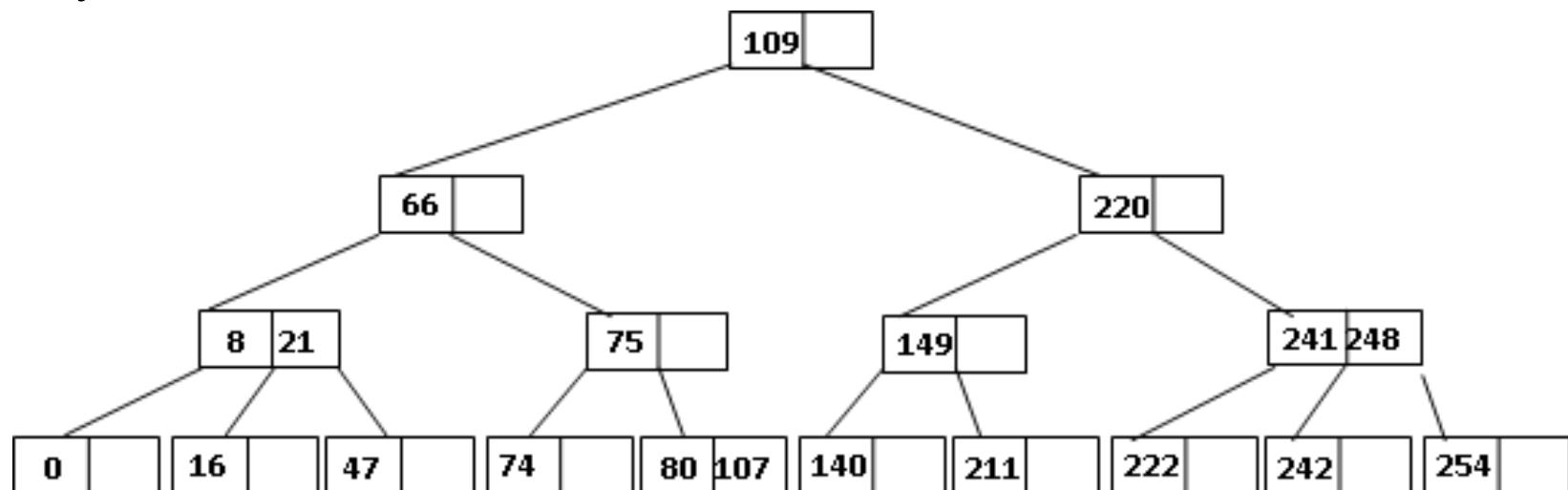


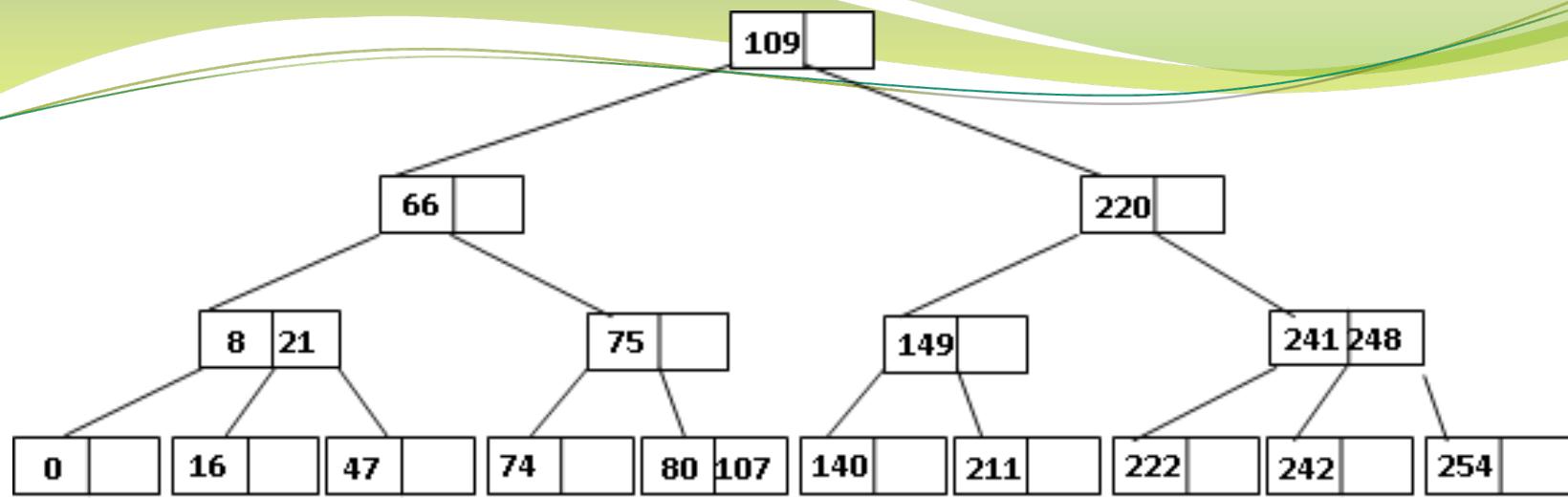
# TP 5C – Ejercicio 13



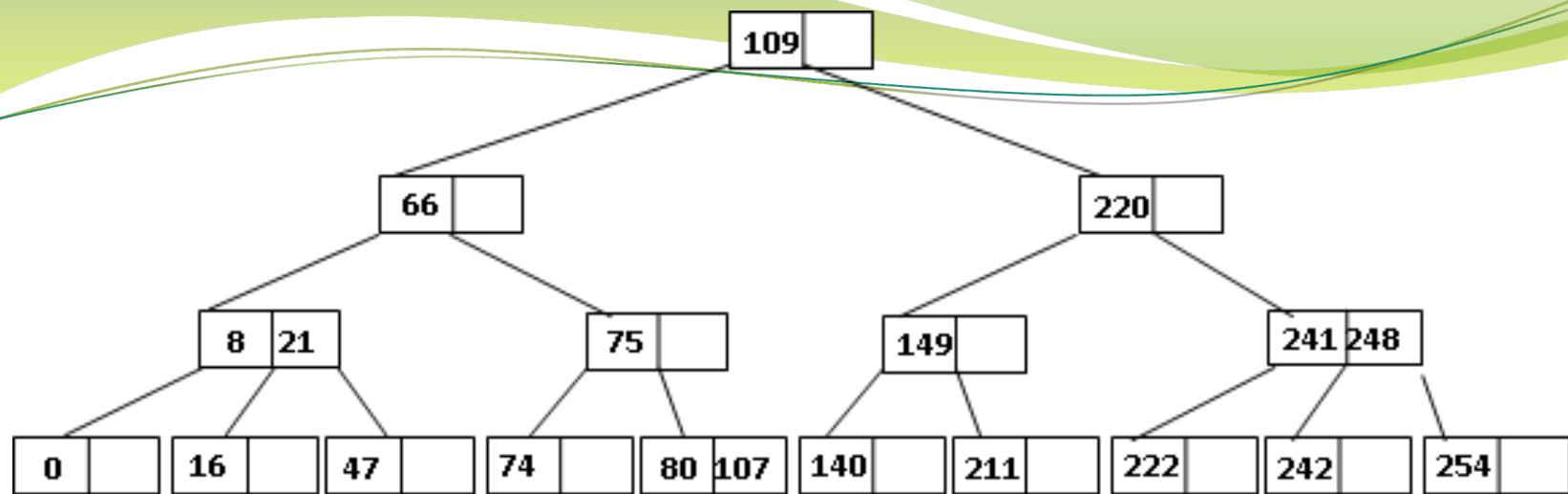
## (ej: lexi mayor, pref. hno izq)

El siguiente árbol surgió de inserción en un árbol B de orden 1 las claves 0, 8, 109, 220, 222, 241, 149, 107, 75, 248, 254, 140, 16, 66, 74, 21, 211, 47, 80 y 242. Eliminar las claves 66, 21, 109, 241, 149, 140, 211, 220 y 242.

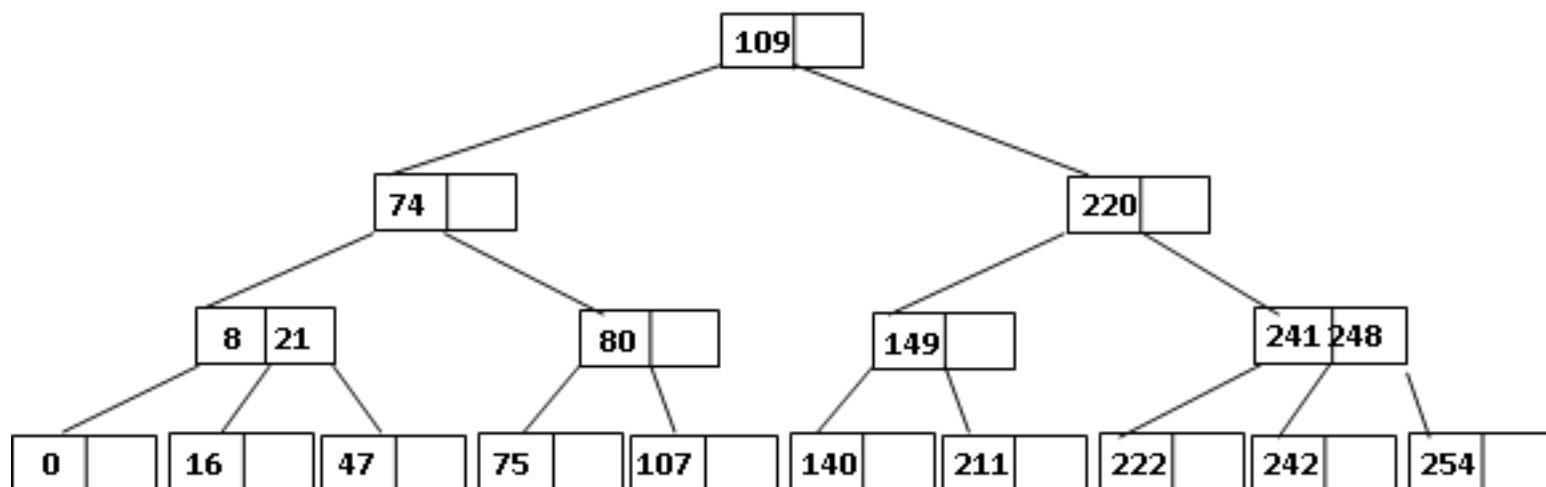


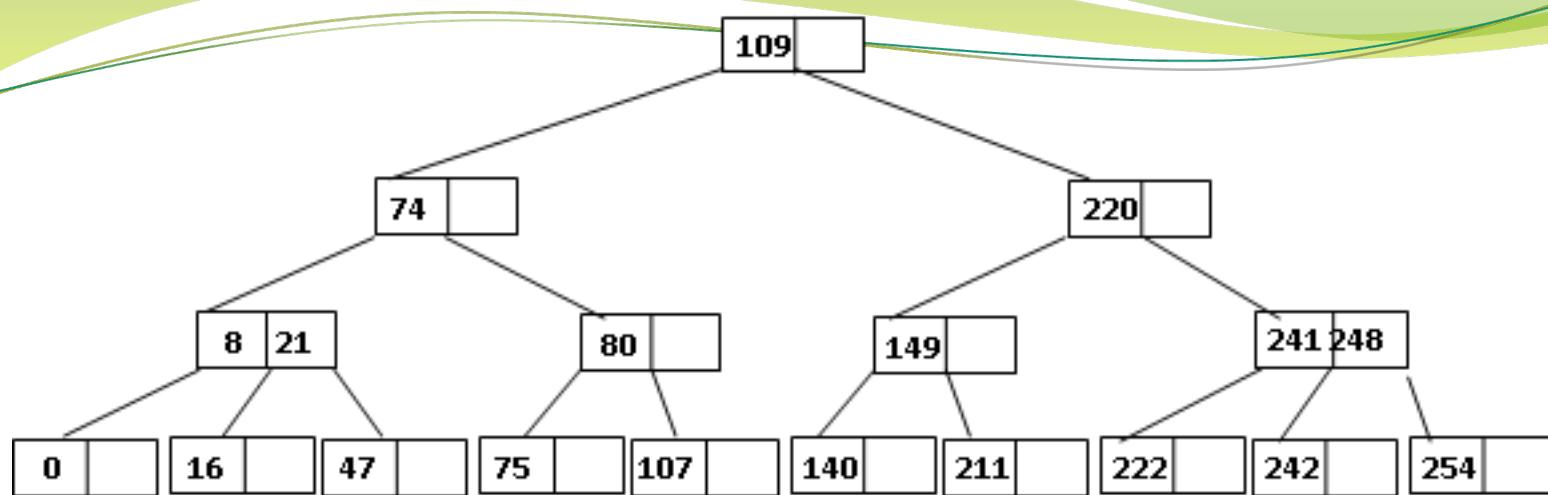


- Eliminar 66

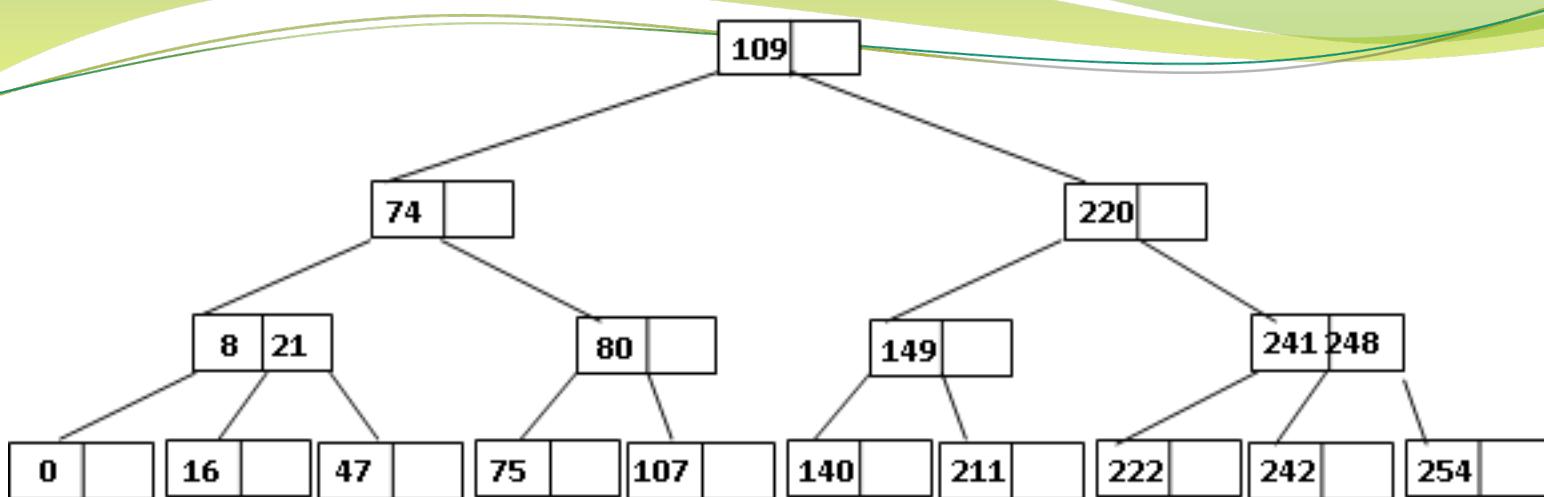


- Eliminar 66

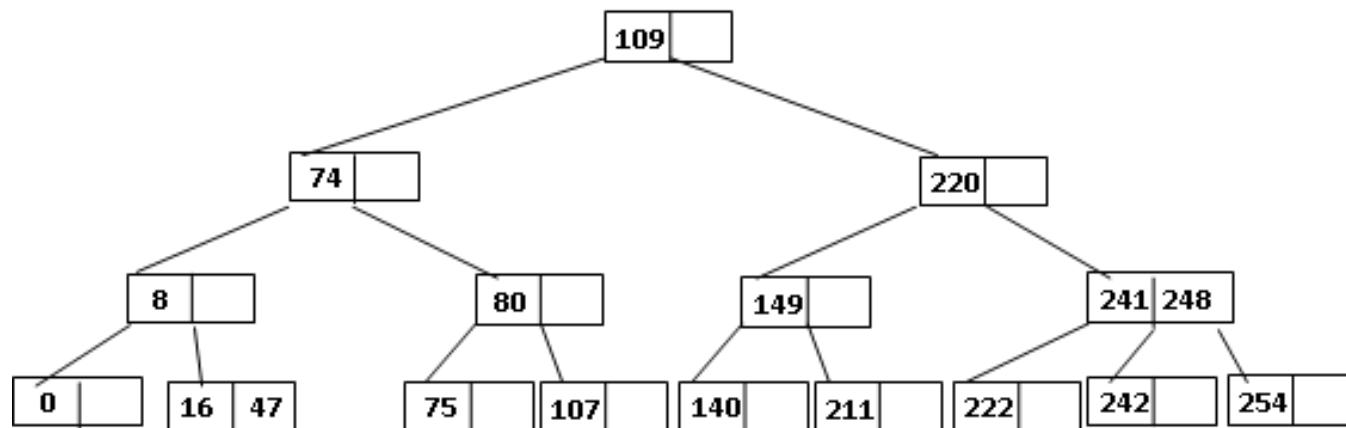


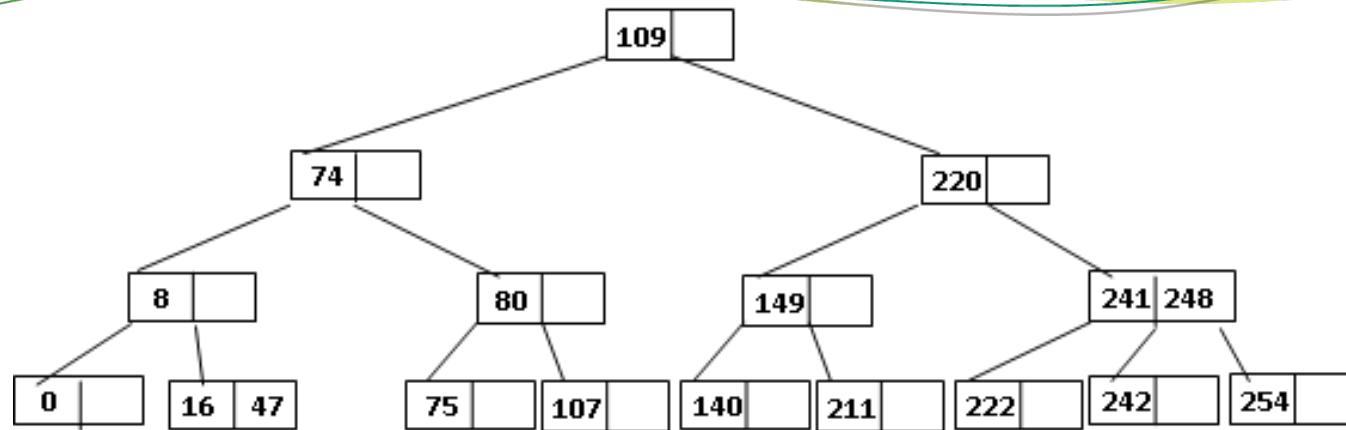


- Eliminar 21

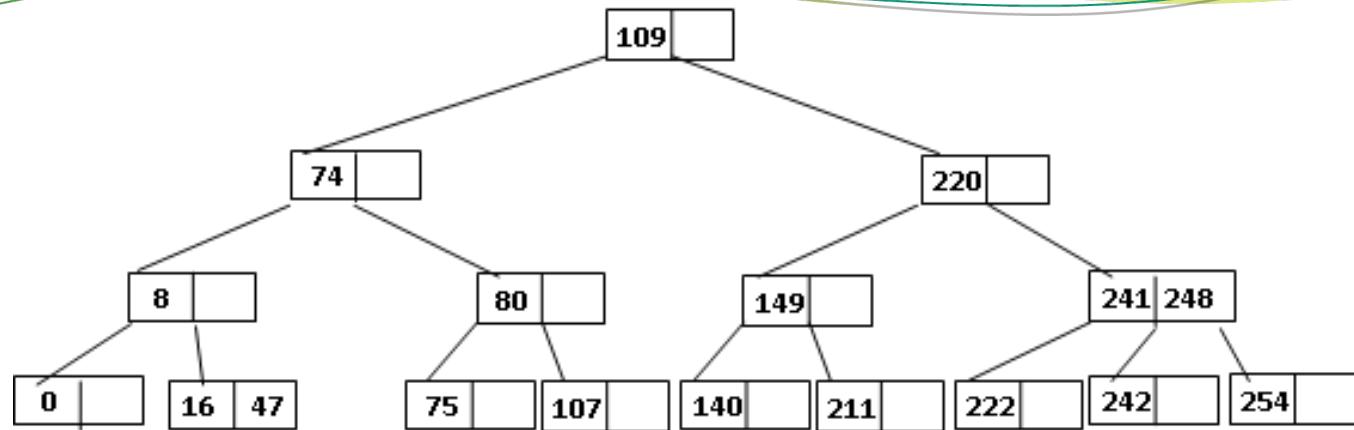


- Eliminar 21

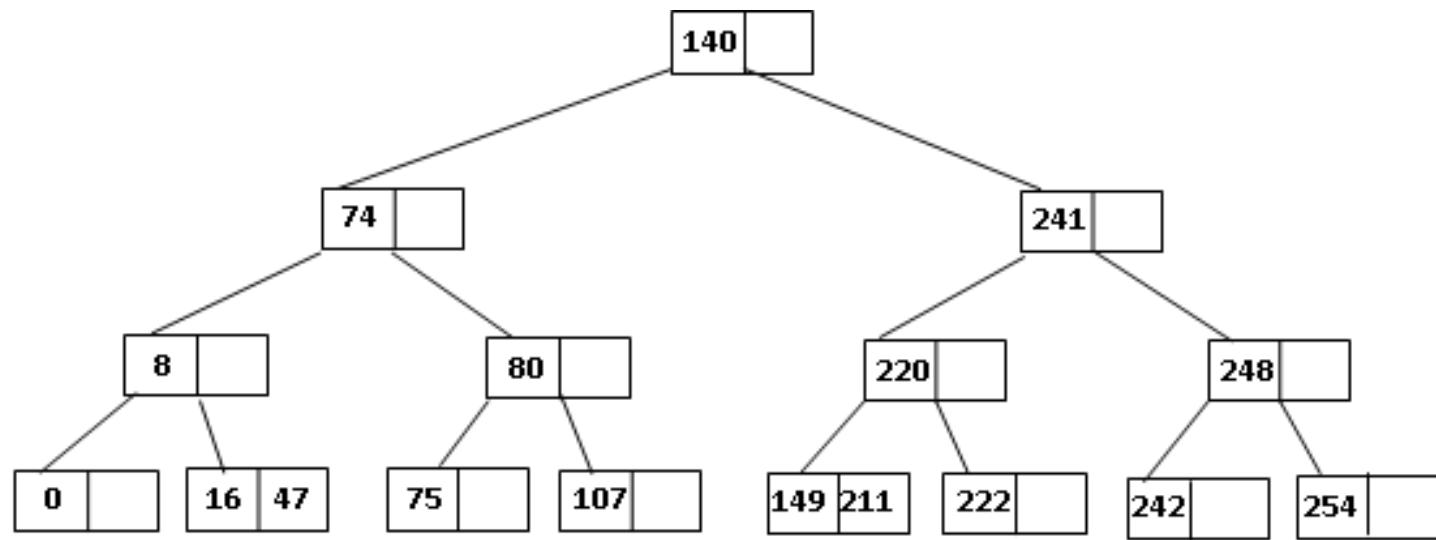


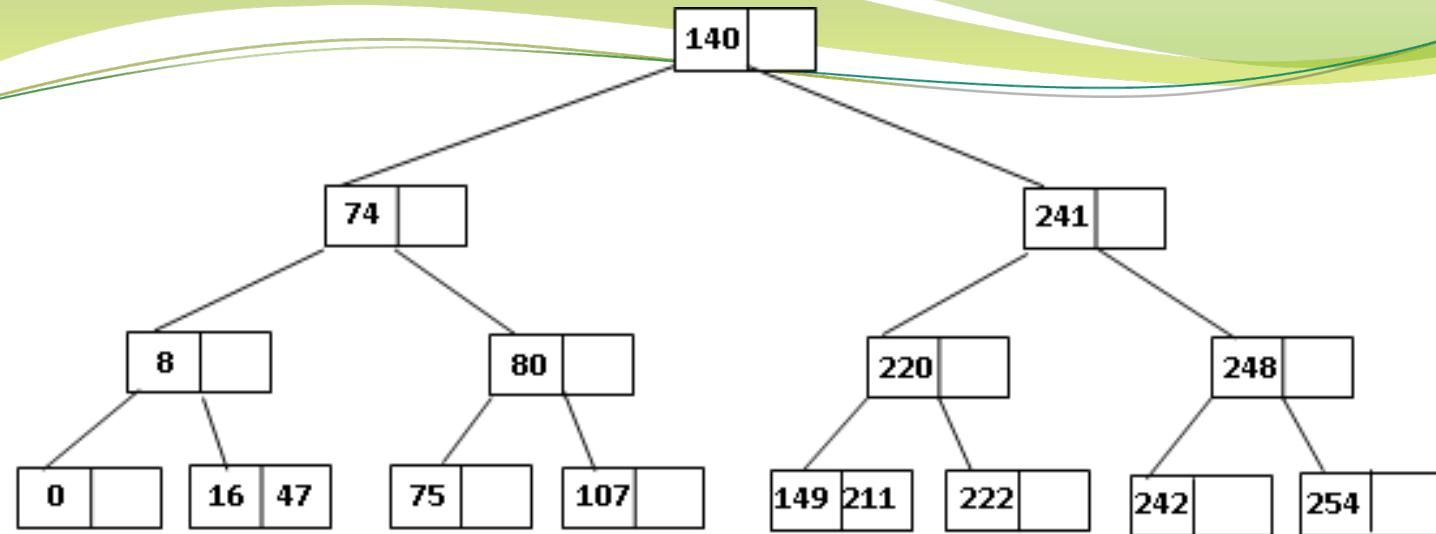


- Eliminar 109

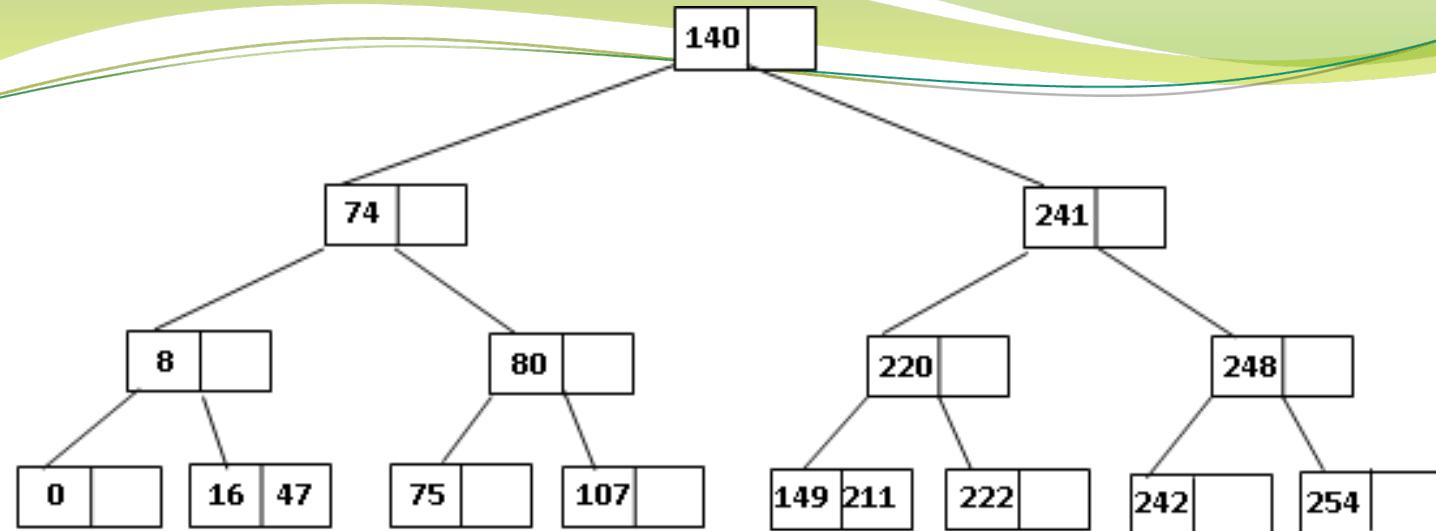


- Eliminar 109

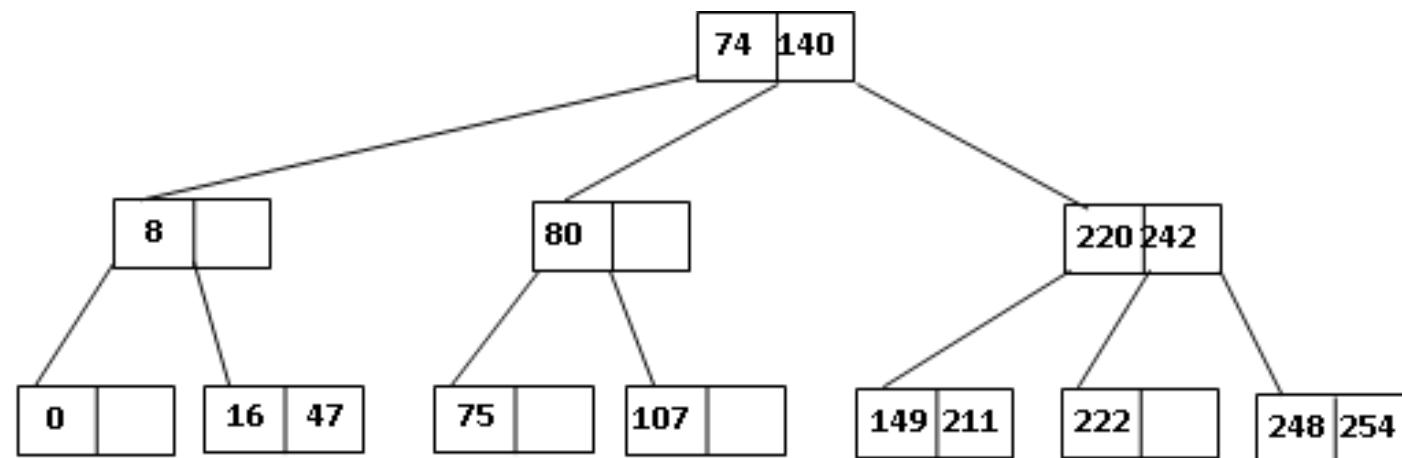


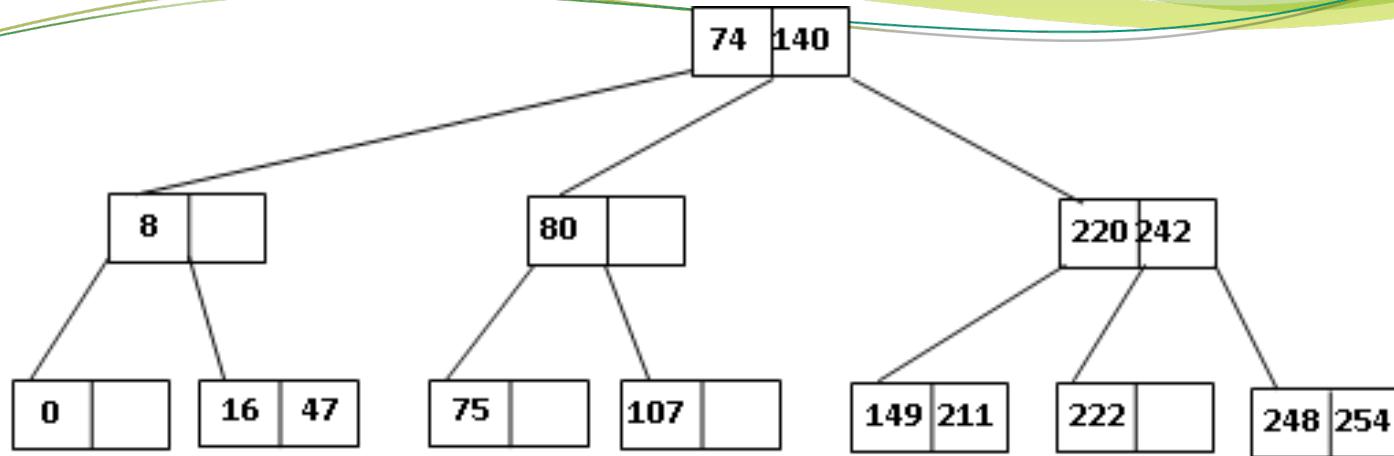


- Eliminar 241

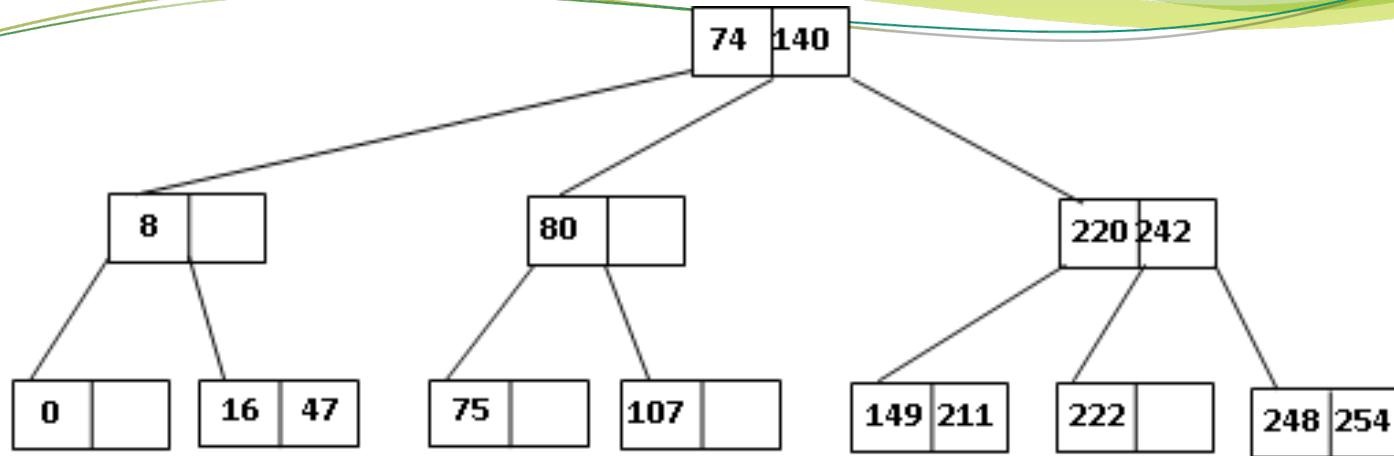


- Eliminar 241

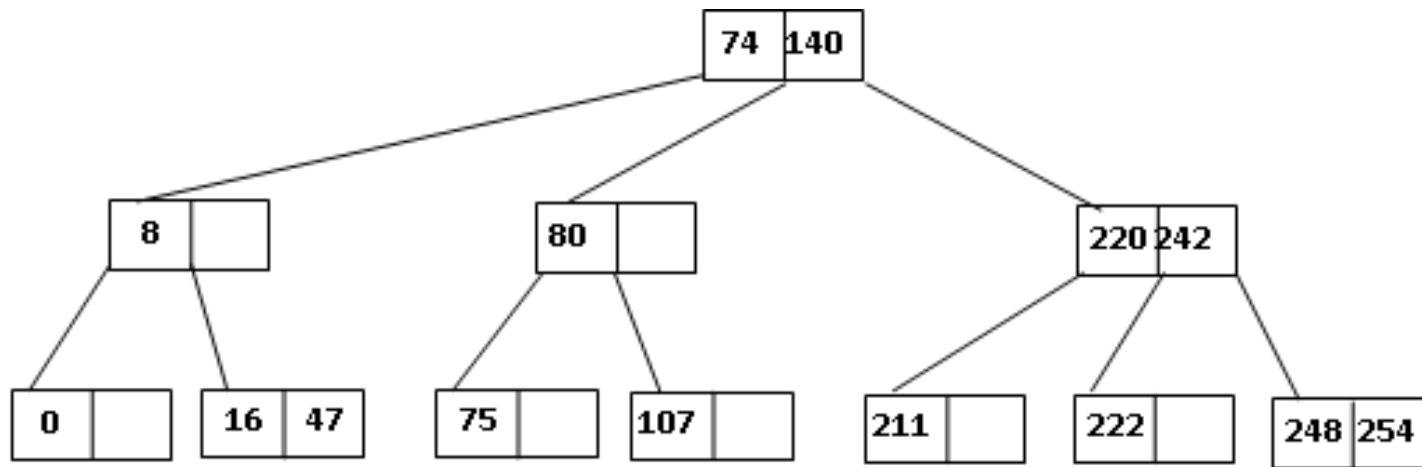


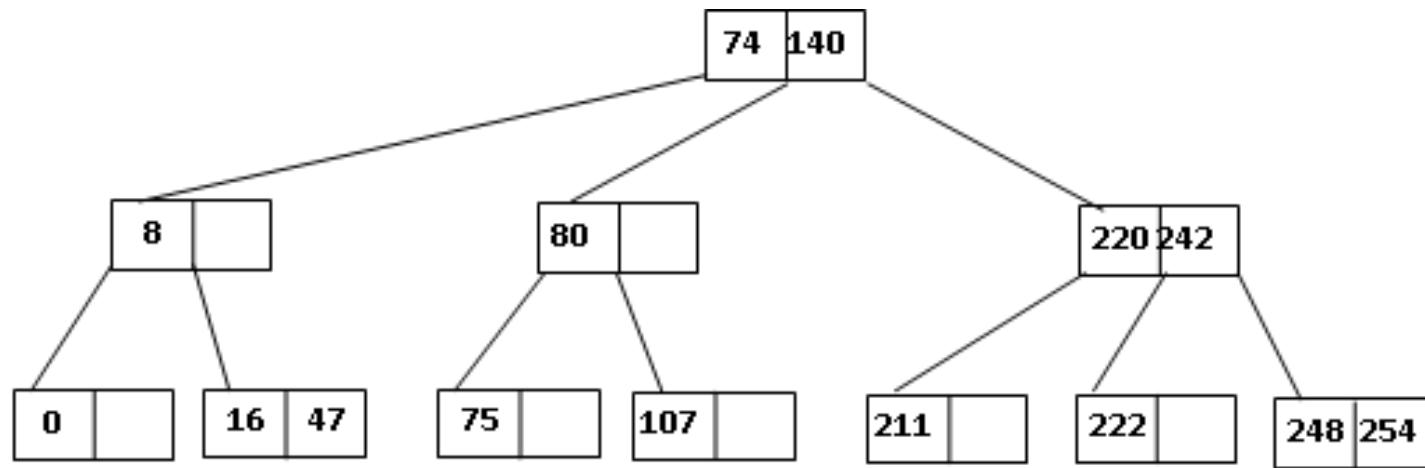


- Eliminar 149

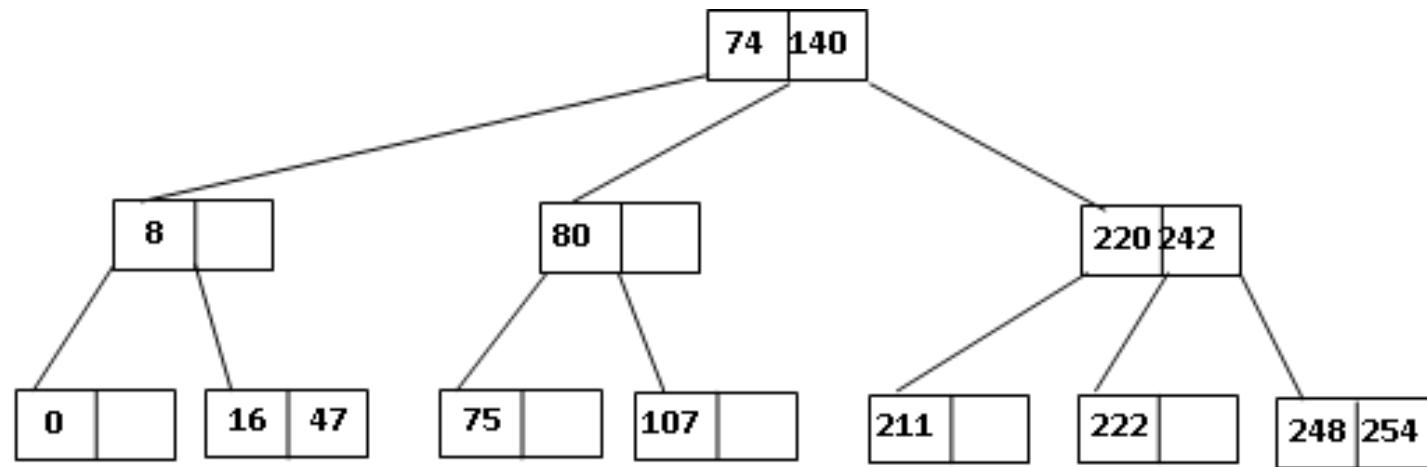


- Eliminar 149

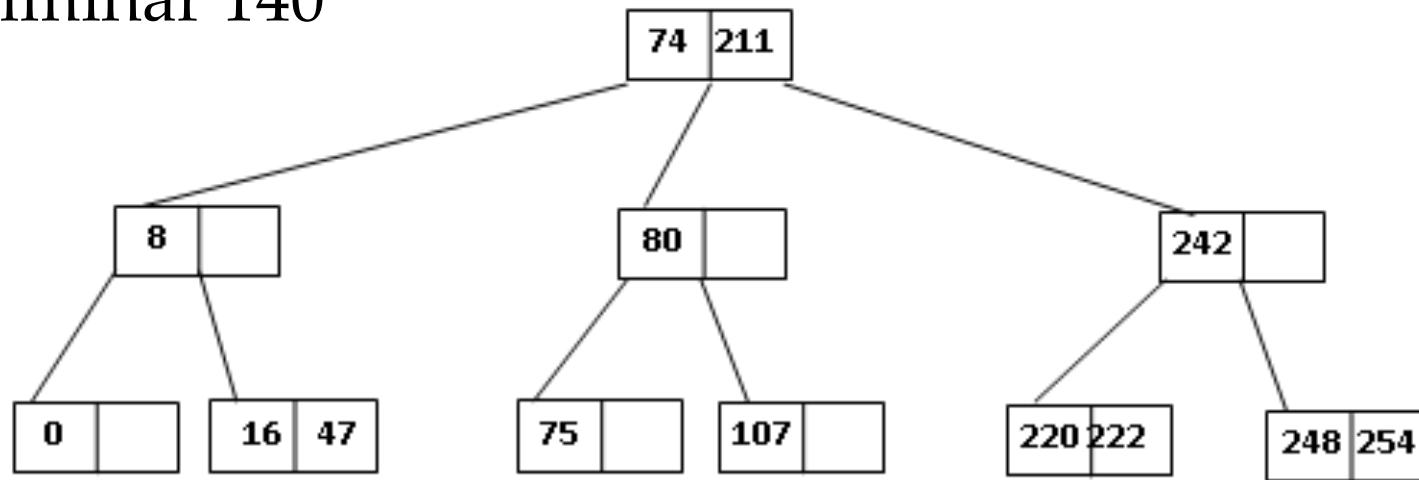


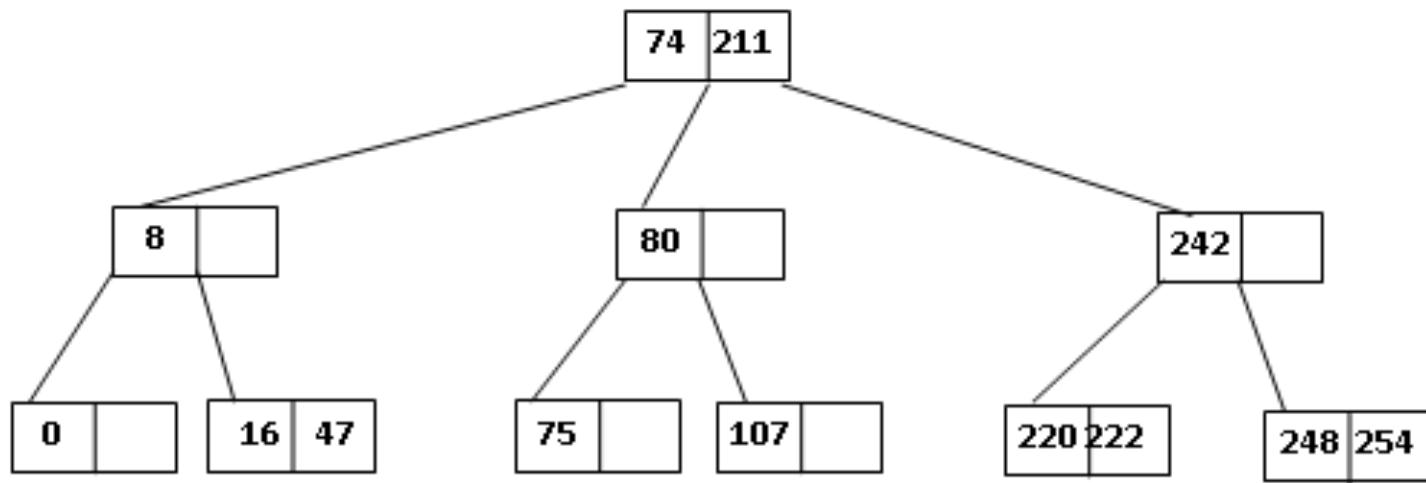


- Eliminar 140

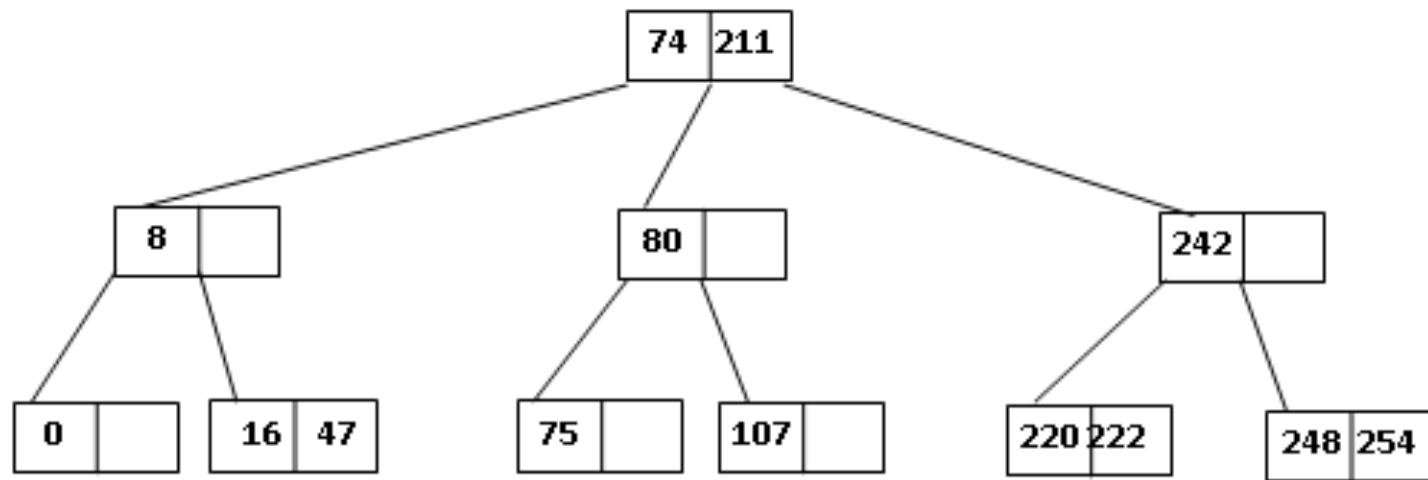


- Eliminar 140

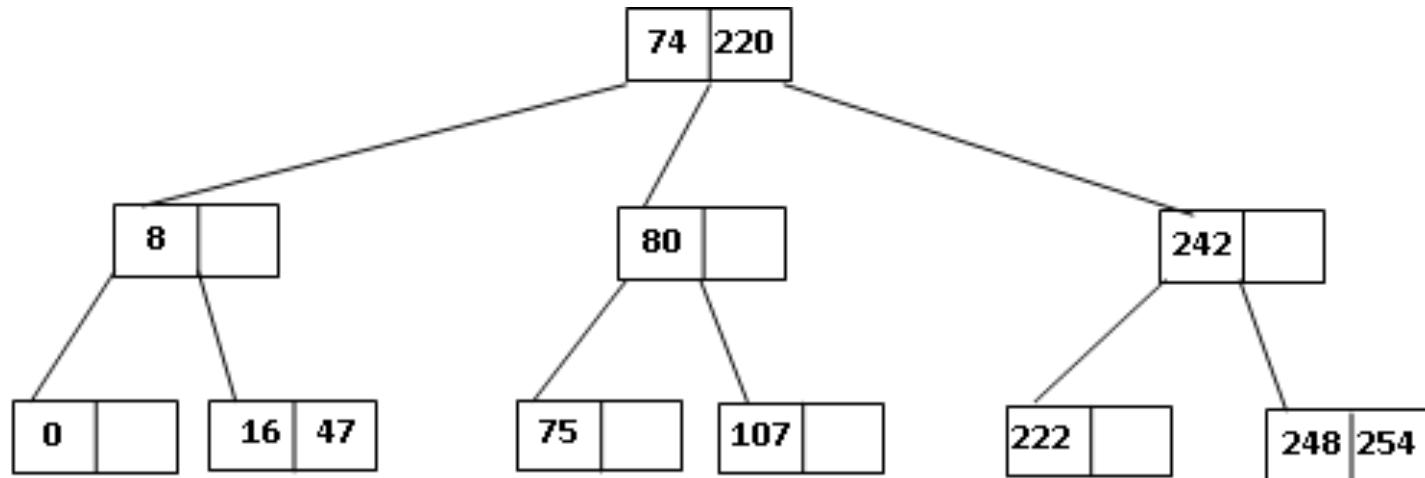


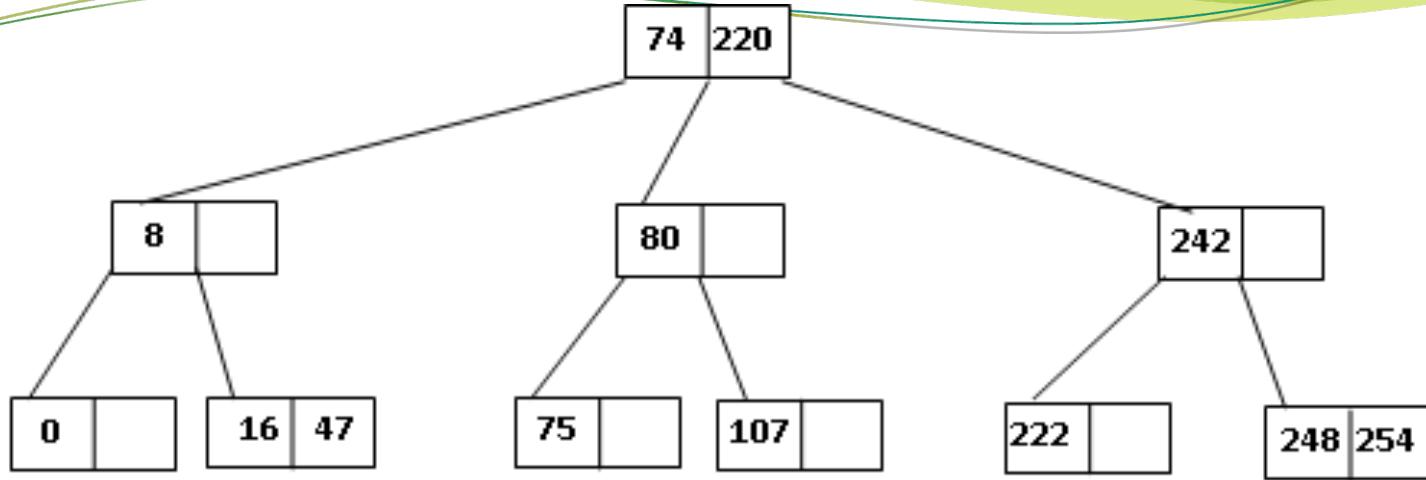


- Eliminar 211

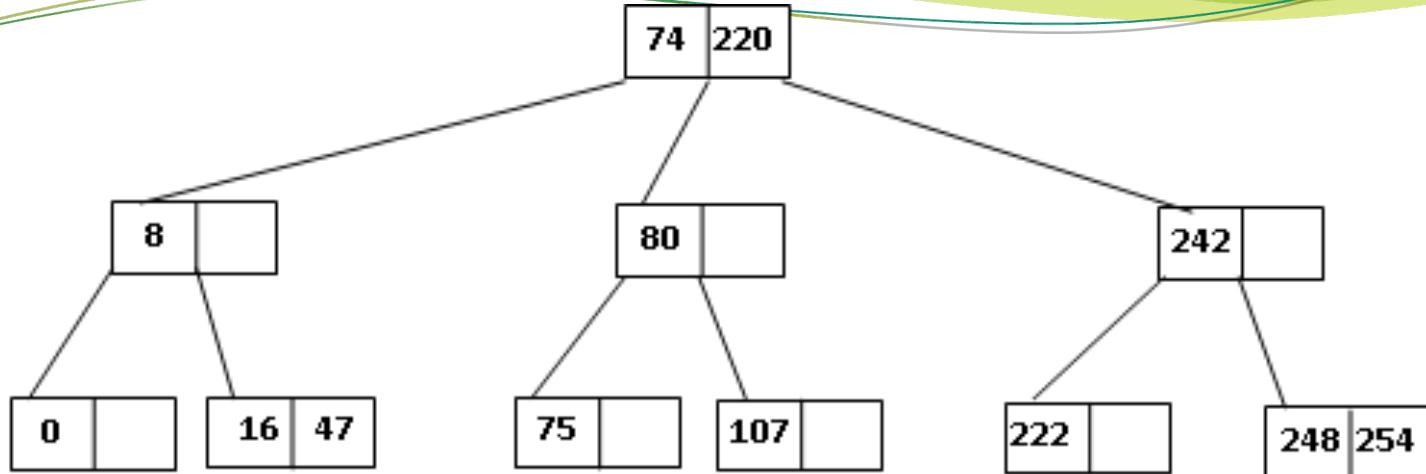


- Eliminar 211

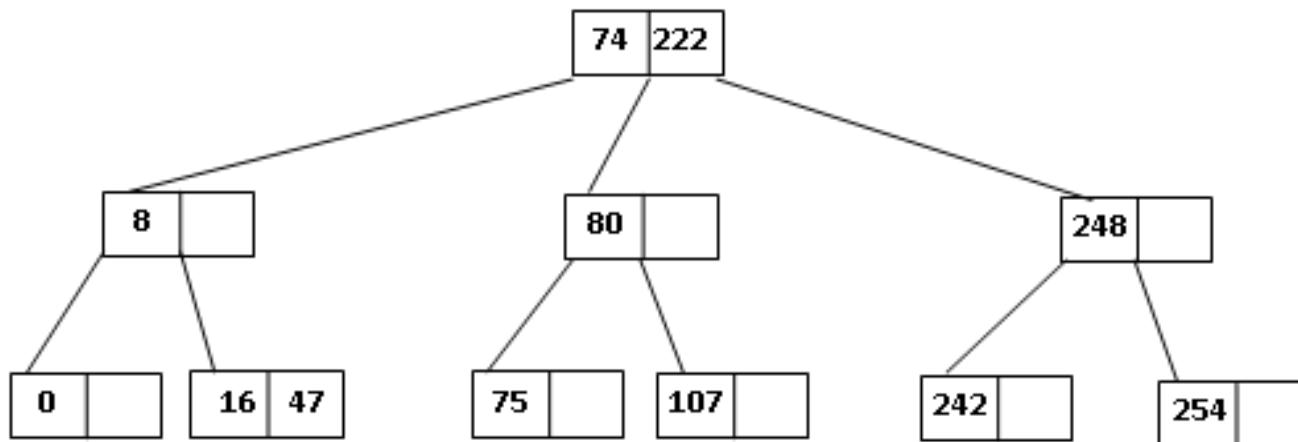


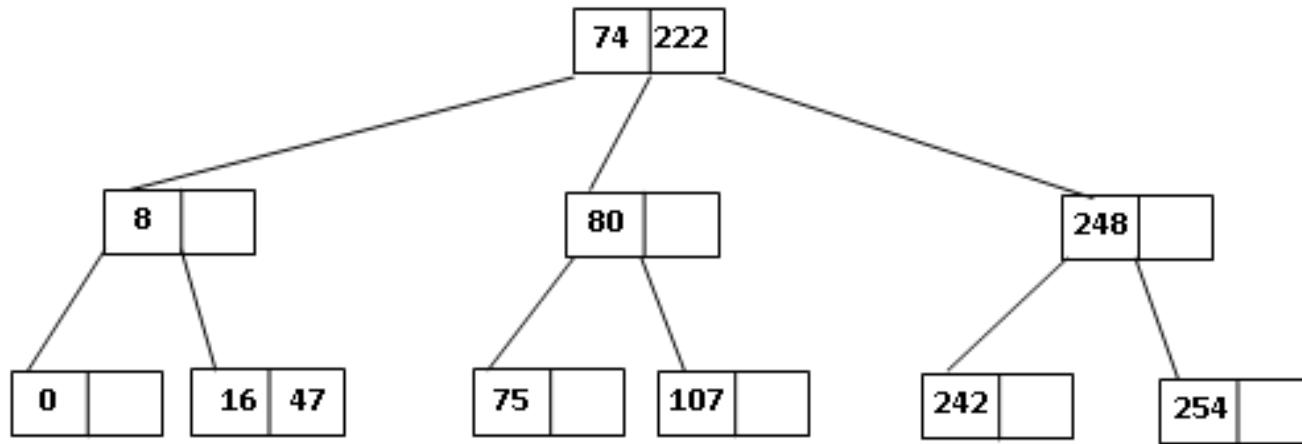


- Eliminar 220

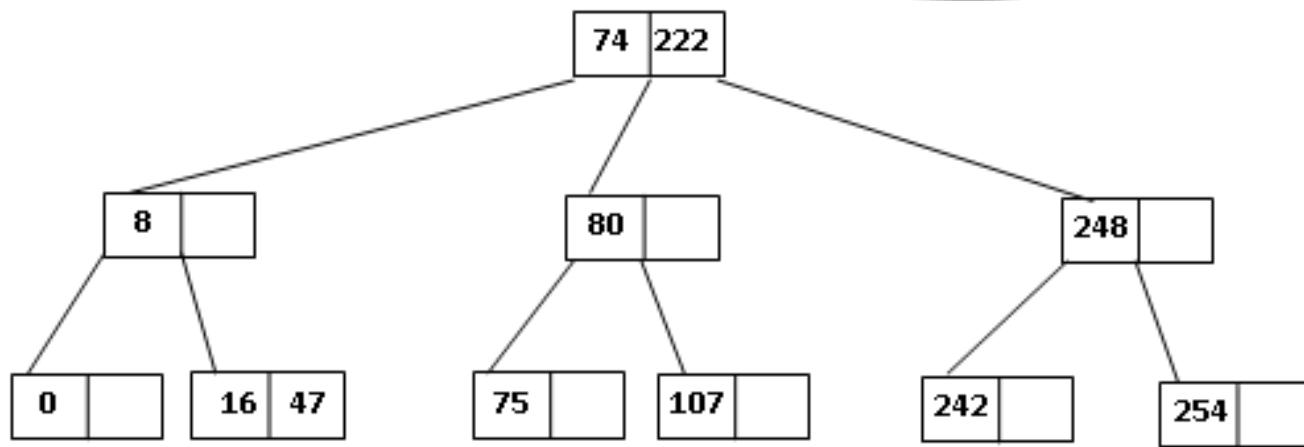


- Eliminar 220

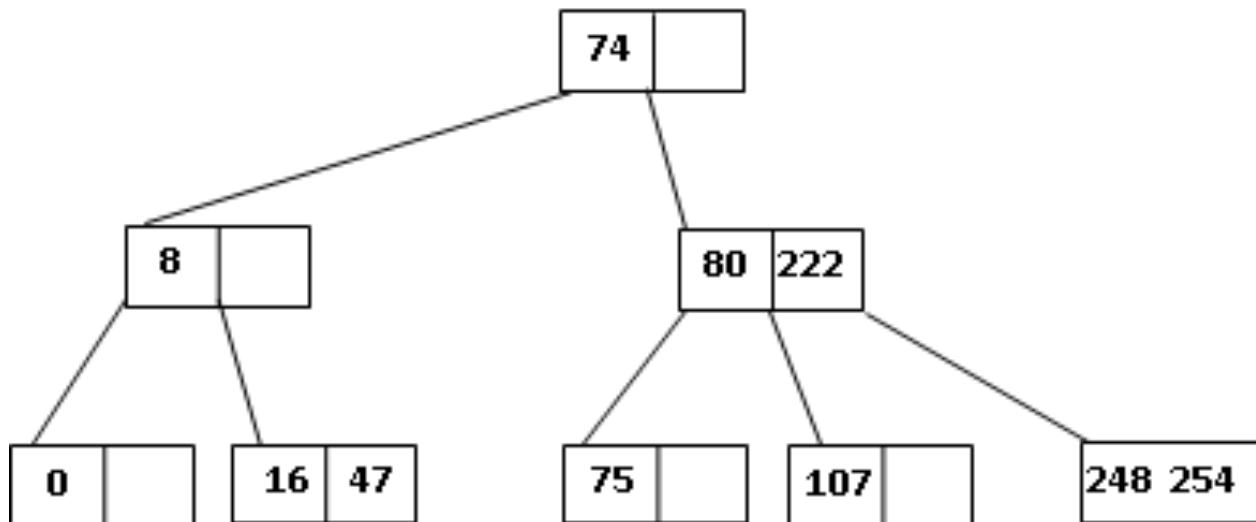




- Eliminar 242



- Eliminar 242



# Eficiencia de los árboles

## Aclaraciones

- Aunque el código que analizamos para B orden N solo guarda claves, típicamente se define:

```
public class BTree<T extends Comparable<T>, V>  
implements ITree<T, V> {...}
```

Donde V es para asociar al Key un Record.

Ej: para record student: legajo, nombre, email.

<10, Ana, ana@Hotmail.com>

<2, Juan, juan@Hotmail.com>

<7, Fer, fer@Hotmail.com>

Si B ordena por legajo

7, <7, Fer, fer@Hotmail.com>

2, <2, Juan, juan@Hotmail.com>

10, <10, Ana, ana@Hotmail.com>



Ej: para record student: legajo, nombre, email.

<10, Ana, ana@Hotmail.com>

<2, Juan, juan@Hotmail.com>

<7, Fer, fer@Hotmail.com>

Si B ordena por **nombre**

Fer, <7, Fer, fer@Hotmail.com>



Juan, <2, Juan, juan@Hotmail.com>

# Eficiencia de los árboles

Si la colección tiene mucha información el Record se deja en disco. Se accede al archivo con RandomAccessFile (fixed length records) por su offset.

Así, el árbol quedaría

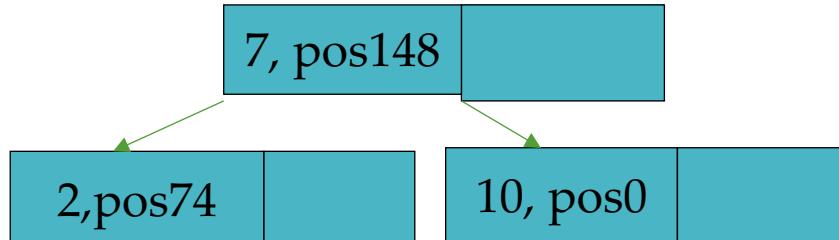
```
public class BTree<T extends Comparable<T>, V>  
implements ITree<T, V> {...
```

Donde V es el “index u offset” del archivo en disco.

Ej: para record student: legajo, nombre, email. Todos los registros ocupan, por ejemplo, 4+50+20 bytes.

<10, Ana, ana@Hotmail.com>  
<2, Juan, juan@Hotmail.com>  
<7, Fer, fer@Hotmail.com>

Si B ordena por legajo



Si B ordena por nombre

