

Estructuras de Datos.

Grado en Ingeniería Informática, Ingeniería del Software e Ingeniería de Computadores

ETSI Informática

Universidad de Málaga

# Tema 4. Árboles

José E. Gallardo, Francisco Gutiérrez, Pablo López

Dpto. Lenguajes y Ciencias de la Computación

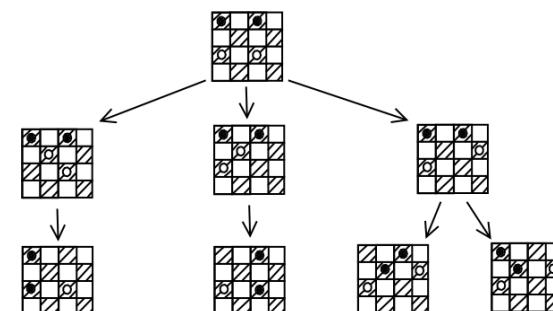
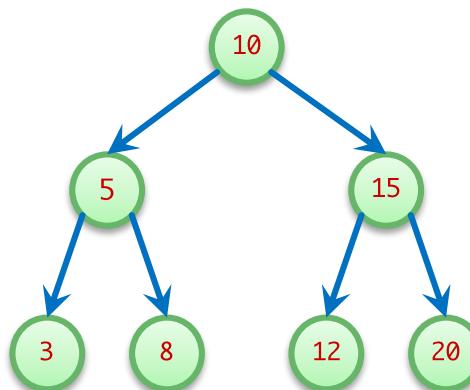
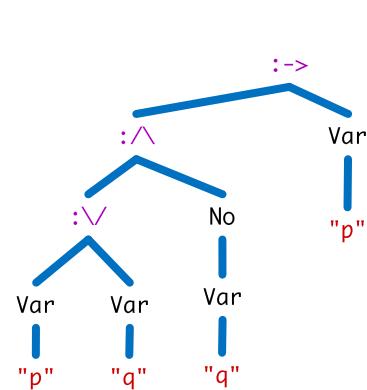
Universidad de Málaga

# Índice

- Árboles
  - Terminología, Definición formal
  - Árboles en Haskell
- Árboles binarios
  - Árboles binarios Auténticos / Perfectos / Completos
  - Relación entre el peso y la altura
  - Árboles binarios en Haskell; recorrido . Construcción de un árbol binario de altura mínima
- Colas con prioridad
  - Especificación. Implementación lineal (Haskell)
- Montículos. Propiedad del montículo (Heap-Order Prop.)
  - Montículo binario (Binary Heap) como AB completo
  - Inserción y eliminación.
  - Representación de un AB Completo con un Array
  - Montículos en Java; representación vía arrays
- Montículos zurdos (Weight Biased Leftist Heaps)
  - La espina derecha tiene longitud logarítmica
  - Mezcla de montículos zurdos (WBLH)
  - Representación de Colas de prioridad con montículos zurdos
  - Comportamiento de los WBLHs
  - Construcción de un montículo zurdo en tiempo lineal
  - Heap Sort
  - Montículos zurdos en Java
- Árboles binarios de búsqueda  
(Binary Search Tree,BST)
  - Inserción. Tree Sort
  - Búsqueda de un elemento, del máximo y del mínimo
  - Eliminación
  - Complejidad de las operaciones
  - BST en Java
- Iteradores para BST en Java
  - Iterador En-Orden para BST
  - Either: Tipo unión en Java
  - Implementación de iteradores
- Árboles AVL
  - Altura de un árbol AVL. El número de oro
  - Rotaciones en árboles binarios de búsqueda
  - Inserción, descompensación y rebalanceo por rotación
  - Búsqueda y eliminación en un árbol AVL
  - Complejidad de operaciones
  - Árboles AVL en Java
- Diccionarios
  - Signatura y axiomas
  - Implementación de diccionarios vía árboles AVL.

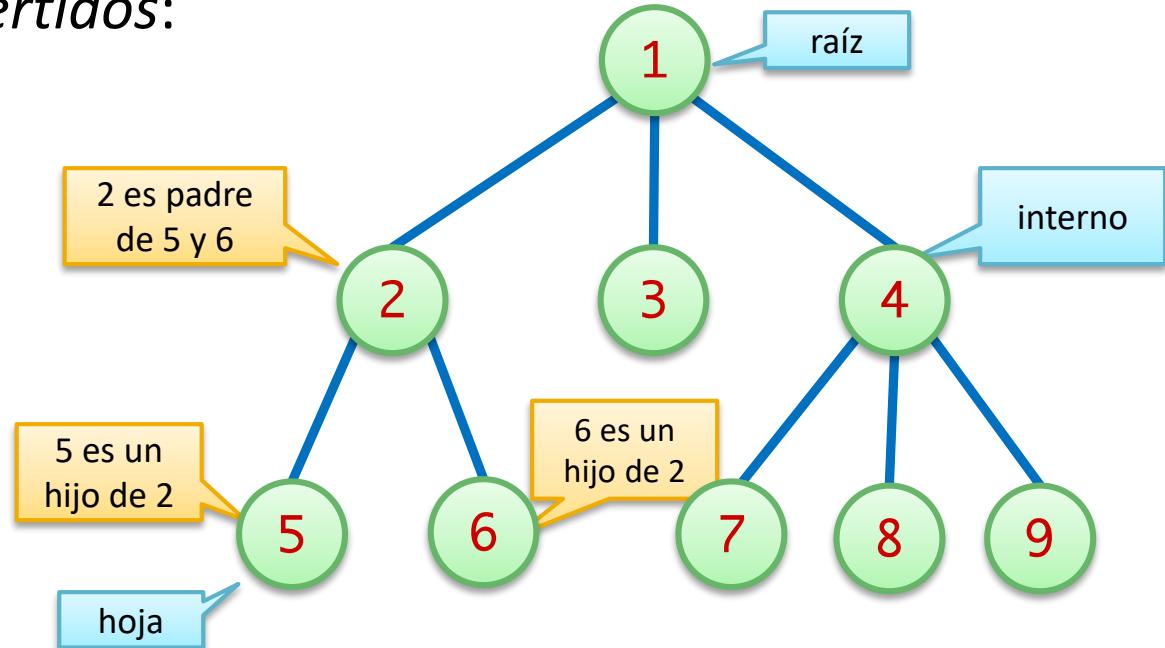
# Árboles

- Son las estructuras no lineales más importantes
- La relación entre los objetos es jerárquica
  - Más rica que la relación lineal (**antes – después**)
- Son **esenciales** en la organización de datos
  - Muchos algoritmos son más eficientes al usar árboles



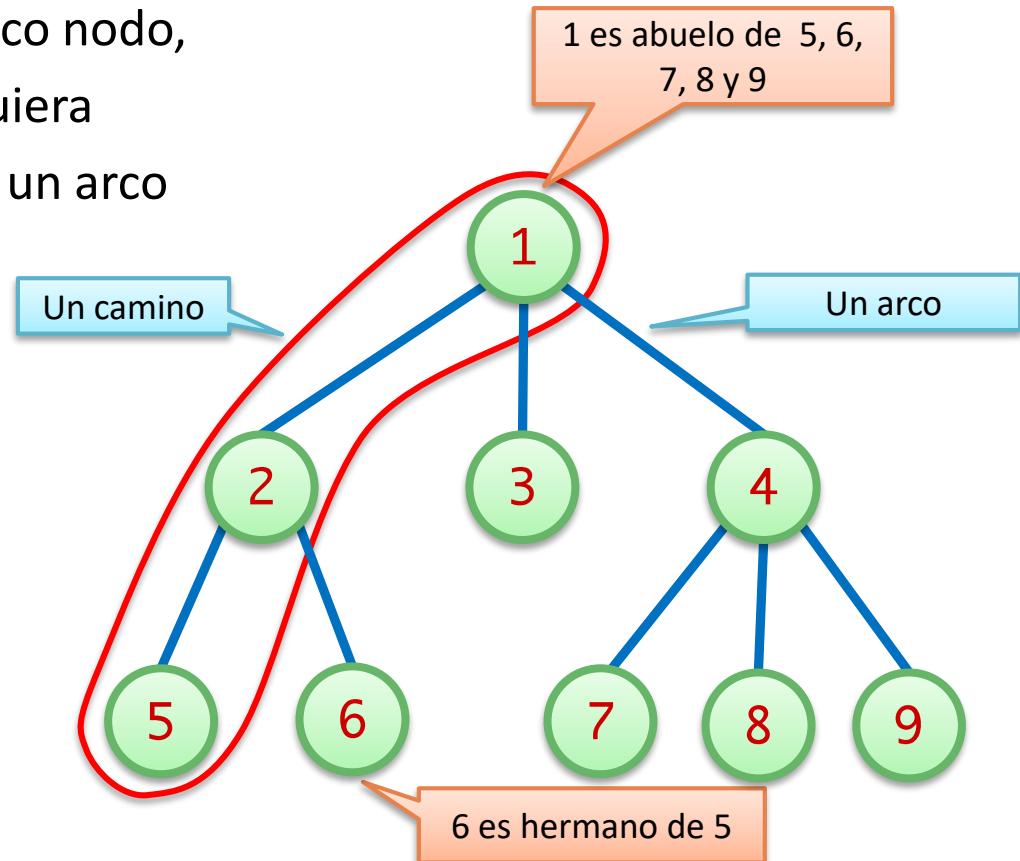
# Árboles. Terminología

- Los árboles almacenan datos jerárquicamente
  - Cada elemento (**nodo**) tiene un **padre** (salvo la **raíz**)
  - Cada elemento tiene cero o más **hijos**
  - Los nodos sin hijos se denominan **hojas**
  - Un nodo con algún hijo se llama **interno**
- Se suelen dibujar *invertidos*:



# Árboles. Terminología (II)

- El padre del padre es el **abuelo**
- Los hijos del mismo padre se llaman **hermanos**
- Un **arco** es un par de nodos  $(u,v)$  tal que  $u$  es el padre de  $v$
- Un **camino** es una secuencia de nodos tal que:
  - o es vacía, o tiene un único nodo,
  - o tiene varios, y cualesquiera dos consecutivos forman un arco



# Árboles. Definición Formal

- Un árbol es un conjunto de nodos junto a una relación *padre-hijo* tal que :
  - Si el conjunto es vacío, el árbol es el árbol vacío
  - En otro caso:
    - Es posible seleccionar un nodo **r**, llamado **raíz**
    - Los restantes nodos pueden agruparse en conjuntos disjuntos tales que cada uno de estos conjuntos es, a su vez, un árbol (llamados **subárboles**).

# Árboles en Haskell

```
data Tree a = Empty | Node a [Tree a] deriving Show
```

Constructor del árbol vacío

Valor raíz

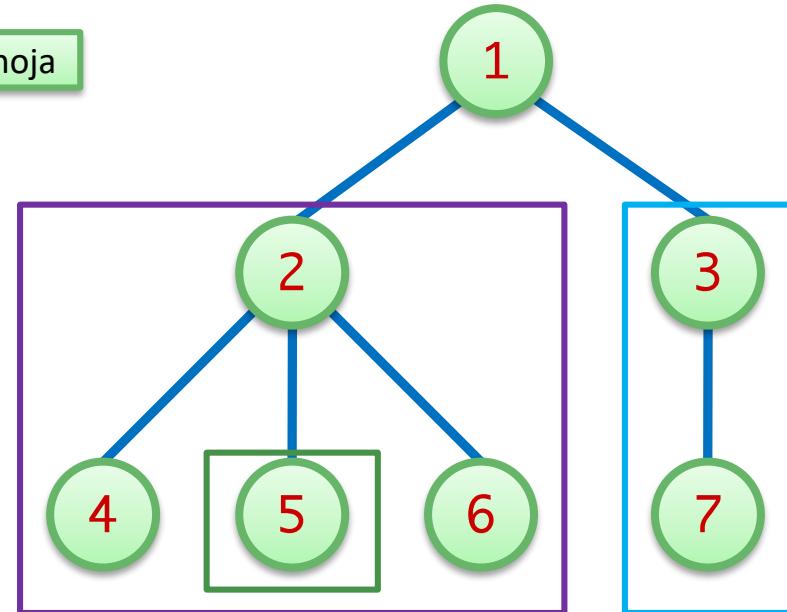
Lista de subárboles hijos

```
tree1 :: Tree Int
```

```
tree1 =
```

```
Node 1 [ Node 2 [ Node 4 []  
, Node 5 []  
, Node 6 []  
,  
, Node 3 [ Node 7 [] ] ] ]
```

Una hoja



Suma los valores de los nodos

```
sumT :: (Num a) => Tree a -> a
```

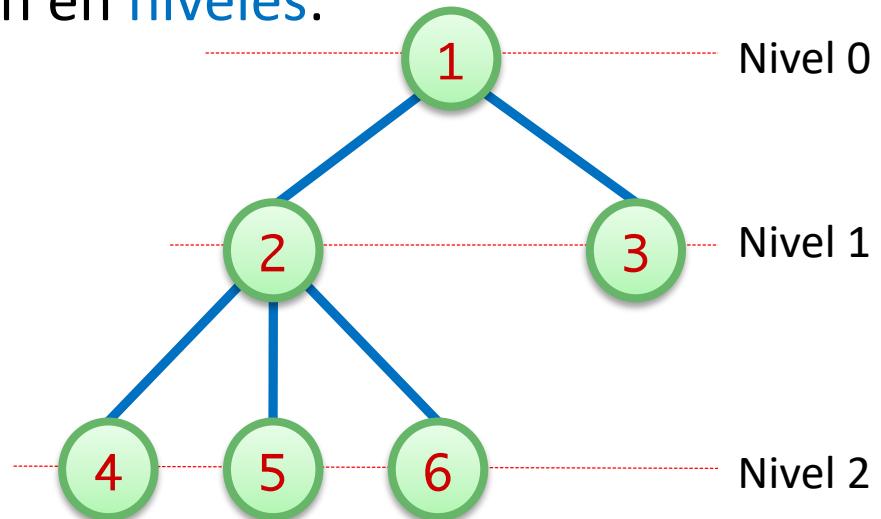
```
sumT Empty = 0
```

```
sumT (Node x ts) = x + sum [sumT t | t <- ts]
```

# Árboles en Haskell (II)

- Los datos en un árbol se organizan en **niveles**:

- La raíz está en el nivel 0
- Los hijos de la raíz en el nivel 1
- Los nietos en el nivel 2
- etc.

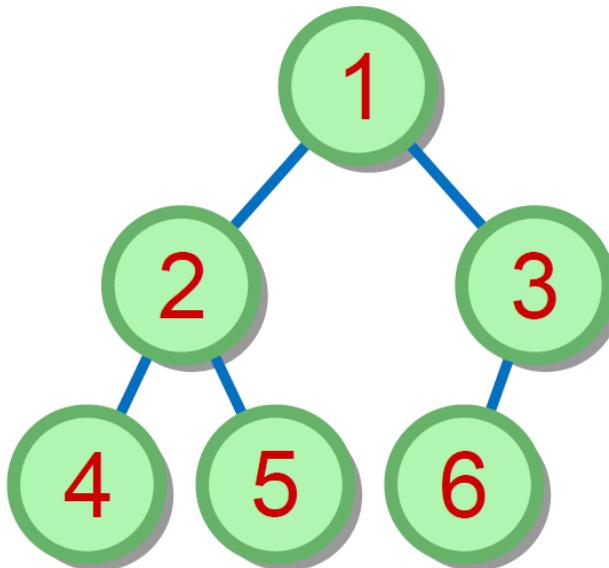


- La **altura** de un árbol se define como su nº de niveles:

```
heightT :: Tree a -> Int
heightT Empty      = 0
heightT (Node x []) = 1
heightT (Node x ts) = 1 + maximum [heightT t | t <- ts]
```

# Árboles Binarios

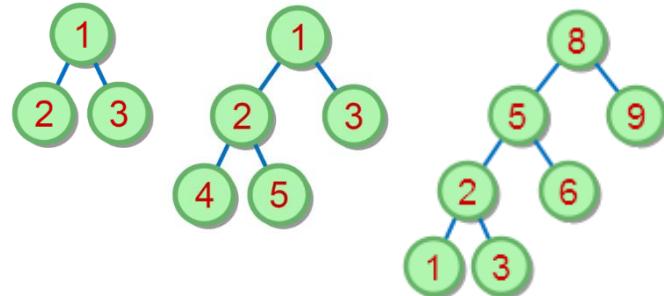
- Árbol binario:
  - Cada nodo tiene **a lo sumo** dos hijos
  - Los hijos se denominan hijo **izquierdo** e hijo **derecho**



# Árboles Binarios Auténticos, Perfectos y Completos

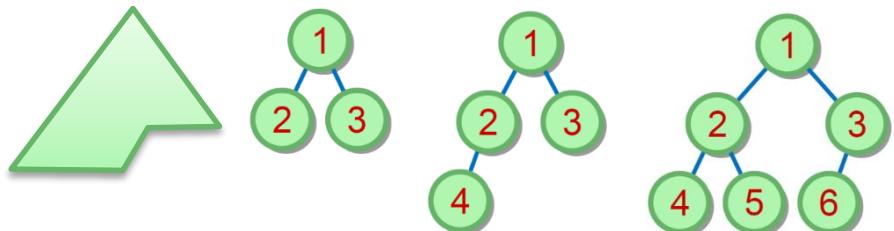
## ■ Árbol Binario Auténtico:

- Salvo las hojas, cada nodo tiene dos hijos



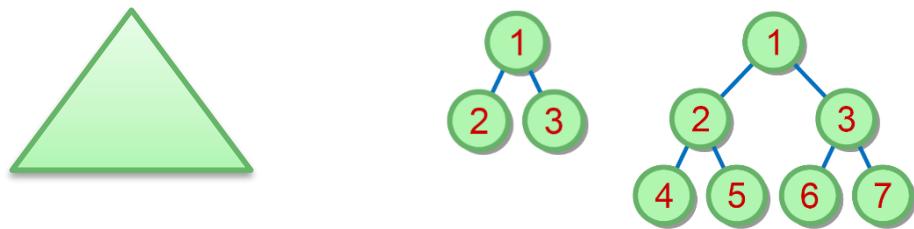
## ■ Árbol Binario Completo:

- Todos los niveles son completos salvo quizás el último nivel, en el que todos aparecen *pegados* a la izquierda



## ■ Árbol Binario Perfecto:

- Son auténticos con todas las hojas en el nivel máximo



# Relación entre el Número de Nodos y la Altura

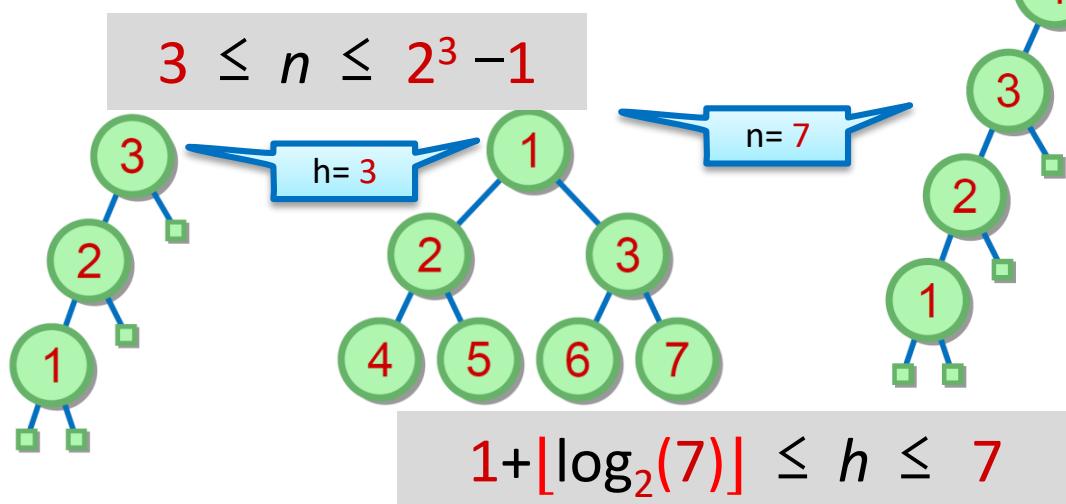
- $n$ : número de nodos
- $h$ : altura

Se da la igualdad para cada árbol perfecto

- Para todo árbol binario no vacío

- $h \leq n \leq 2^{h-1} = (1 + 2^1 + 2^2 + \dots + 2^{h-1})$

- $1 + \lfloor \log_2(n) \rfloor \leq h \leq n$



# Árboles Binarios en Haskell

```
data TreeB a = EmptyB | NodeB a (TreeB a) (TreeB a) deriving Show
```

Constructor del árbol vacío

Valor del nodo

Subárbol izquierdo

Subárbol derecho

```
tree2 :: TreeB Int
```

```
tree2 =
```

```
NodeB 1 (NodeB 2 (NodeB 4 EmptyB EmptyB)  
          (NodeB 5 EmptyB EmptyB))  
(NodeB 3 (NodeB 6 EmptyB EmptyB)  
          EmptyB)
```

-- suma de valores de los nodos

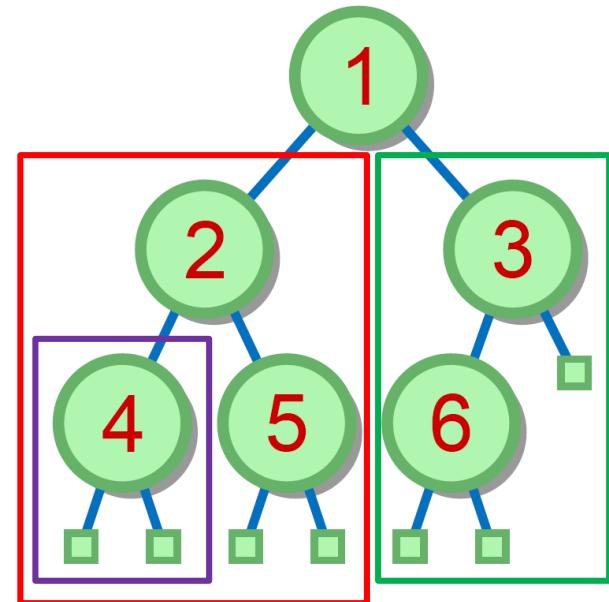
```
sumB :: (Num a) => TreeB a -> a
```

```
sumB EmptyB = 0
```

```
sumB (NodeB x lt rt) = x + sumB lt + sumB rt
```

Subárbol izquierdo  
(Left subtree)

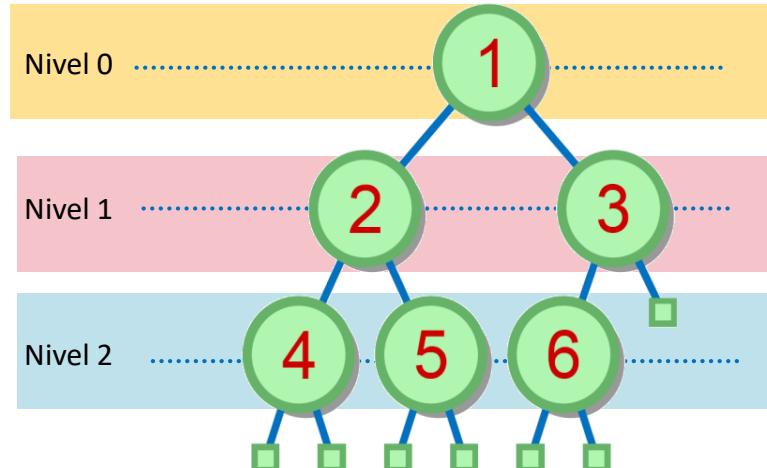
Right subtree



# Árboles Binarios en Haskell (II)

```
data TreeB a = EmptyB | NodeB a (TreeB a) (TreeB a) deriving Show

-- la lista de nodos en cierto nivel de un árbol
atLevelB :: Int -> TreeB a -> [a]
atLevelB _ EmptyB      = []
atLevelB 0 (NodeB x lt rt) = [x]
atLevelB n (NodeB x lt rt) = atLevelB (n-1) lt ++ atLevelB (n-1) rt
```

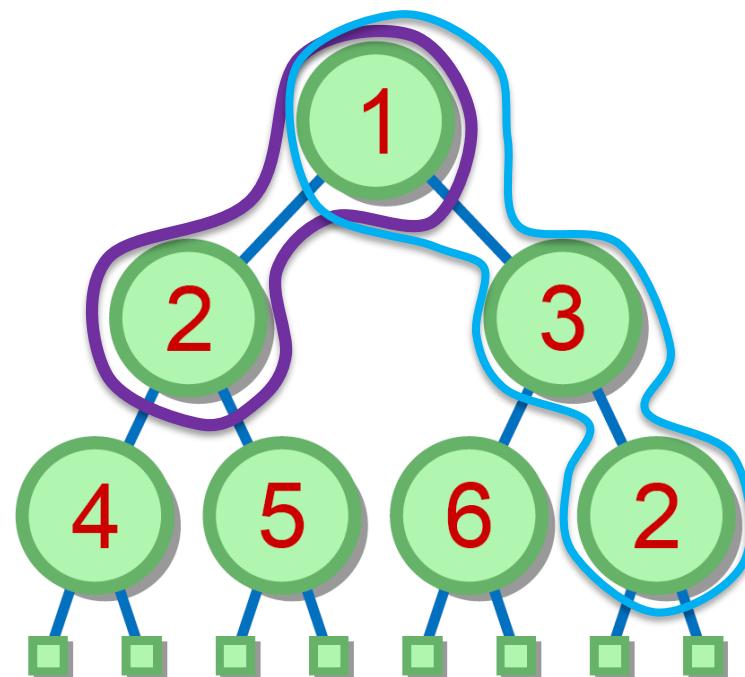


```
Main> atLevelB 0 tree2
[1]
Main> atLevelB 1 tree2
[2,3]
Main> atLevelB 2 tree2
[4,5,6]
```

# Árboles Binarios en Haskell (III)

```
-- caminos hasta cierto nodo (desde la raíz)
pathsToB :: (Eq a) => a -> TreeB a -> [[a]]
pathsToB x EmptyB = []
pathsToB x (NodeB y lt rt)
| x==y          = [y] : ps
| otherwise      = ps
where
ps = map (y:) (pathsToB x lt ++ pathsToB x rt)
```

```
Main> pathsToB 5 tree3
[[1,2,5]]
Main> pathsToB 2 tree3
[[[1,2], [1,3,2]]]
```



# Recorrido de Árboles Binarios

- Recorrido o exploración de un árbol: proceso que visita cada nodo exactamente una vez
- Los recorridos se clasifican según el orden de visita
- Los órdenes de visita más usuales para árboles binarios (que pueden generalizarse para cualquier árbol) son:
  - Pre-Orden
  - En-Orden (o en profundidad)
  - Post-Orden
  - En anchura (se verá más tarde)

# Recorrido de Árboles Binarios (II)

```
preOrderB :: TreeB a -> [a]
```

```
preOrderB EmptyB      = []
```

```
preOrderB (NodeB x lt rt) = [x] ++ preOrderB lt ++ preOrderB rt
```

```
inOrderB :: TreeB a -> [a] -- o recorrido en profundidad
```

```
inOrderB EmptyB      = []
```

```
inOrderB (NodeB x lt rt) = inOrderB lt ++ [x] ++ inOrderB rt
```

```
postOrderB :: TreeB a -> [a]
```

```
postOrderB EmptyB      = []
```

```
postOrderB (NodeB x lt rt) = postOrderB lt ++ postOrderB rt ++ [x]
```

```
Main> preOrderB tree4
```

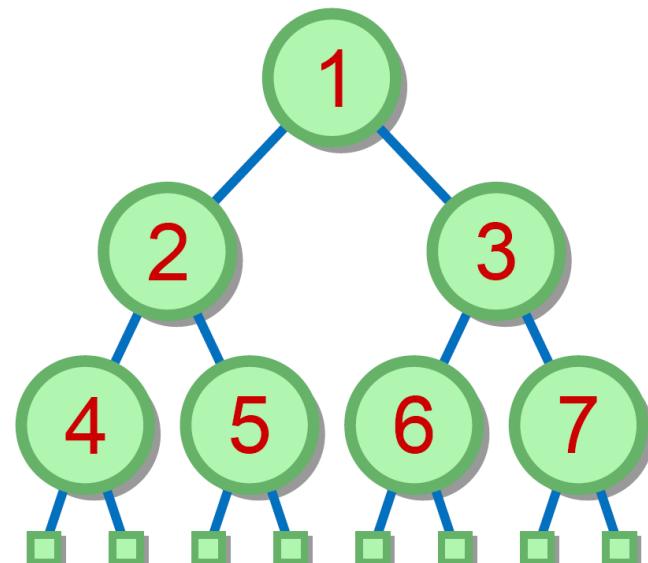
```
[1,2,4,5,3,6,7]
```

```
Main> inOrderB tree4
```

```
[4,2,5,1,6,3,7]
```

```
Main> postOrderB tree4
```

```
[4,5,2,6,7,3,1]
```



# Colas con Prioridad

- Consideraremos que existe una relación de orden entre los elementos a encolar (edad, sueldo,...), de forma que se atienda antes a los de mayor prioridad: los más jóvenes, los de menor sueldo, ... en definitiva, los menores para la relación de orden



- Los elementos pueden insertarse en cualquier orden pero son extraídos de la cola de acuerdo con su prioridad.
- En caso de tener la misma prioridad, se sigue política **FIFO**

# Colas con Prioridad: Signatura

```
-- devuelve una cola vacía
empty :: PQueue a

-- test para colas vacías
isEmpty :: PQueue a -> Bool

-- inserta un elemento en una cola de prioridad
enqueue :: (Ord a) => a -> PQueue a -> PQueue a

-- devuelve el elemento más prioritario (con menor valor)
first :: (Ord a) => PQueue a -> a

-- devuelve la cola obtenida al eliminar
-- el elemento más prioritario (con menor valor)
dequeue :: (Ord a) => PQueue a -> PQueue a
```

# Colas con Prioridad: Especificación

isEmpty empty -- ax1

not (isEmpty (enqueue x q)) -- ax2

first (enqueue x empty) == x -- ax3

dequeue (enqueue x empty) == empty -- ax4

Los anteriores axiomas coinciden con los de Queue. Dos adicionales capturan el comportamiento **SIFO** (smallest in, first out) :

first (enqueue y (enqueue x q)) ==  
first (enqueue (min x y) q) -- ax5

dequeue (enqueue y (enqueue x q)) ==  
enqueue (max x y) (dequeue (enqueue (min x y) q)) -- ax6

Los elementos con igual prioridad salen en orden FIFO

min x y = if x <= y then x else y  
max x y = if x <= y then y else x

# Colas con Prioridad: Implementación Lineal

```
module DataStructures.PriorityQueue.LinearPriorityQueue
( PQueue
empty
isEmpty
enqueue
first
dequeue
) where
```

```
data PQueue a = Empty | Node a (PQueue a)
```

```
empty :: PQueue a
empty = Empty
```

```
isEmpty :: PQueue a -> Bool
isEmpty Empty = True
isEmpty _ = False
```

Observa el uso de < (menor) en vez  
de <= (menor o igual) en enqueue



```
enqueue :: (Ord a) => a -> PQueue a -> PQueue a
enqueue x Empty = Node x Empty
enqueue x (Node y q)
| x < y = Node x (Node y q)
| otherwise = Node y (enqueue x q)
```

Los elementos son encolados en  
orden ascendente

Recursivamente, encola  
x detrás de y O(n) ⊕

# Colas con Prioridad: Implementación Lineal (II)

```
first :: PQueue a -> a
```

```
first Empty      = error "first on empty queue"
```

```
first (Node x _) = x
```

El elemento mínimo  
aparece al principio

```
dequeue :: PQueue a -> PQueue a
```

```
dequeue Empty      = error "dequeue on empty queue"
```

```
dequeue (Node _ q) = q
```

Ejercicio: implementar la  
Cola con Prioridad en Java  
usando una lista enlazada

# Colas con Prioridad: Implementación Lineal (y III)

- Coste para la implementación lineal:

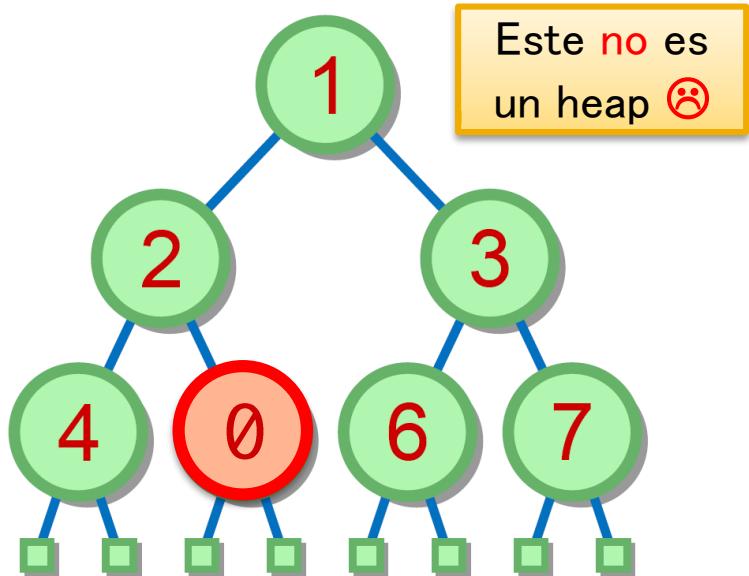
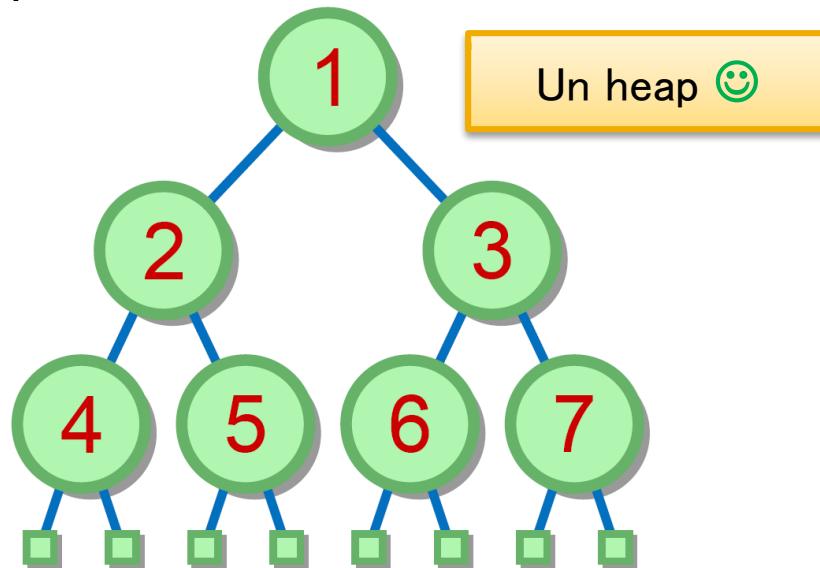
Operación	Coste
empty	$O(1)$
isEmpty	$O(1)$
enqueue	$O(n)$
first	$O(1)$
dequeue	$O(1)$

 podemos mejorarla

- Veremos implementaciones de colas de prioridad que mejoran enqueue, pero empeoran ligeramente dequeue.

# Montículo. Propiedad de Orden (Heap-Order Property)

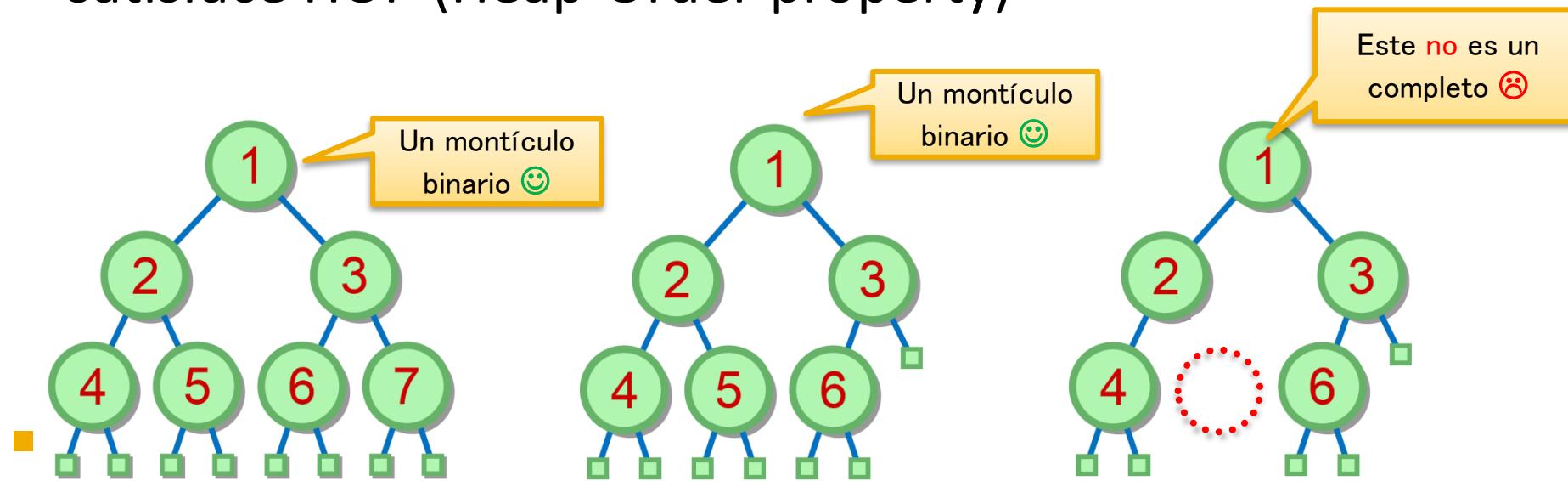
- Un **montículo (heap)** es un árbol que verifica la siguiente propiedad de orden (**Heap-Order Property** o HOP):
  - El valor de cada nodo distinto de la raíz es mayor o igual al valor de su padre



- Equivalentemente: la secuencia de nodos de **cualquier** camino desde la raíz a una hoja es ascendente
- El valor mínimo está en la raíz 😊

# Montículo Binario (Binary Heap)

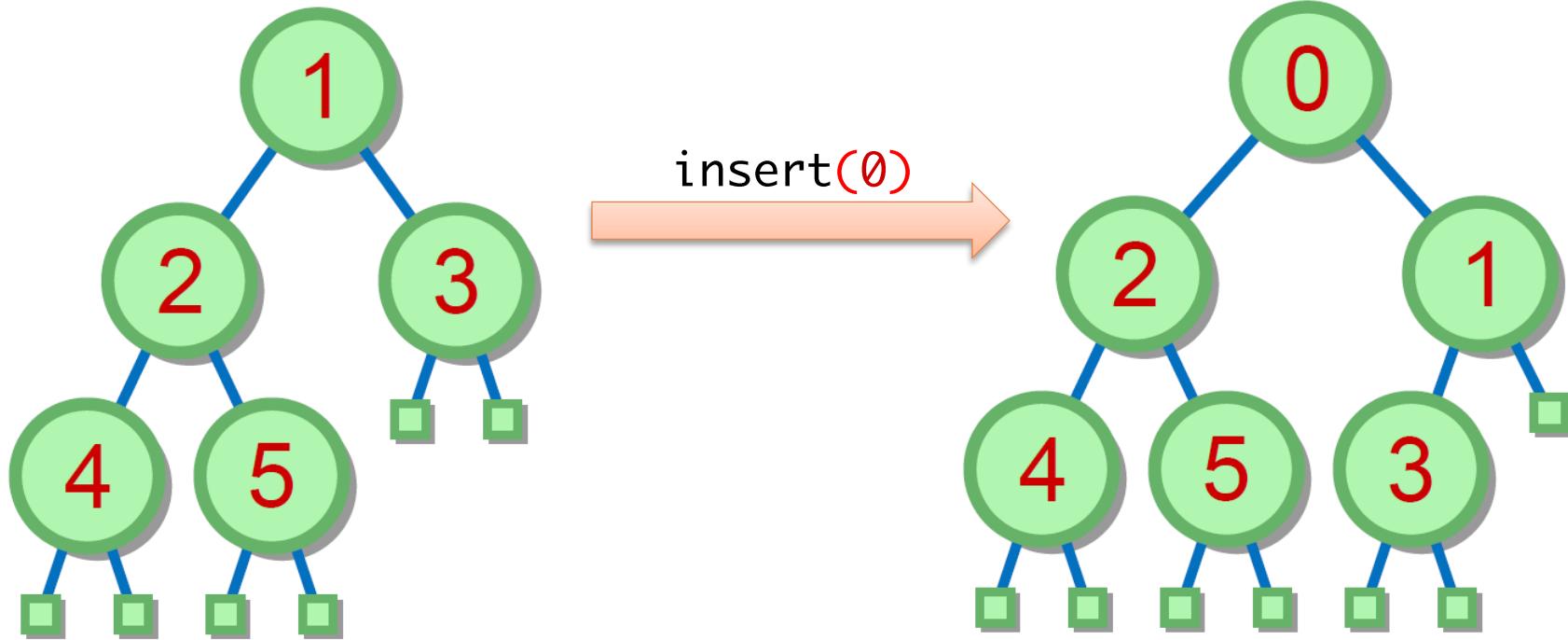
- Un montículo binario es un árbol binario **completo** que satisface HOP (Heap-Order property)



- Recordemos: la altura de un árbol binario completo con  $n$  elementos es  $1 + \lfloor \log_2(n) \rfloor$ . Luego la altura de un montículo binario será *mínima*.

# Inserción en un Montículo Binario

- Queremos añadir un elemento a un montículo binario de forma que el resultado siga siendo un montículo binario

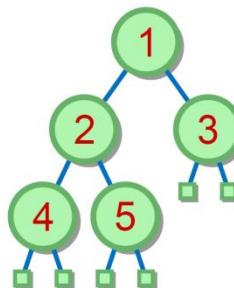


# Inserción en un Montículo Binario (II)

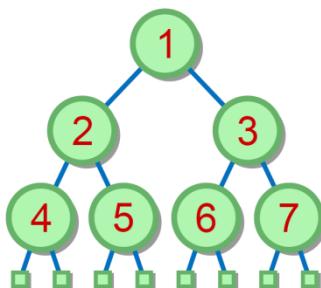
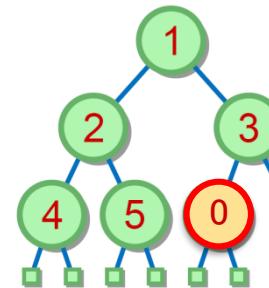
- Inserción: se realiza en dos fases (colocar al fondo, y reflotar)

- 1) Colocar el nuevo elemento en la primera posición libre:
  - A la derecha del último nodo del último nivel, o bien
  - A la izquierda del siguiente nivel si el último está completo
- 2) Mientras que el nuevo elemento sea menor que el padre:
  - Intercambiar el nuevo con el padre

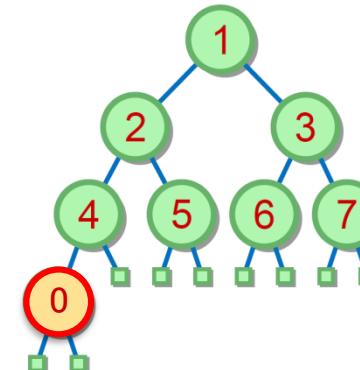
El resultado es un  
árbol completo 😊



insert(0)



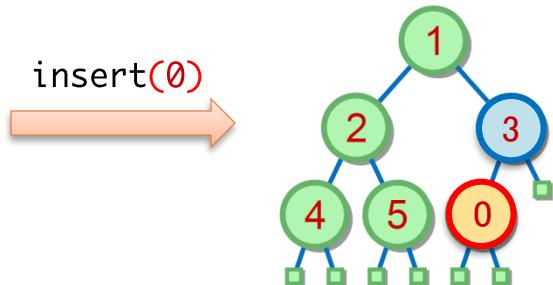
insert(0)



# Inserción en un Montículo Binario (III)

- Inserción: se realiza en dos fases
    - 1) Colocar el nuevo elemento en la primera posición libre:
      - A la derecha del último nodo del último nivel, o bien
      - A la izquierda del siguiente nivel si el último está completo
    - 2) Mientras que el nuevo elemento sea menor que el padre
      - Intercambiar el nuevo con el padre

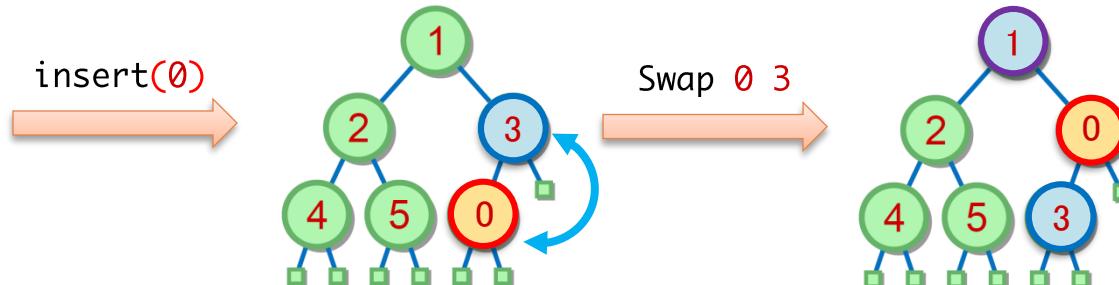
Para reestablecer la  
propiedad HOP ☺



# Inserción en un Montículo Binario (IV)

- Inserción: se realiza en dos fases
  - 1) Colocar el nuevo elemento en la primera posición libre:
    - A la derecha del último nodo del último nivel, o bien
    - A la izquierda del siguiente nivel si el último está completo
  - 2) Mientras que el nuevo elemento sea menor que el padre:
    - Intercambiar el nuevo con el padre

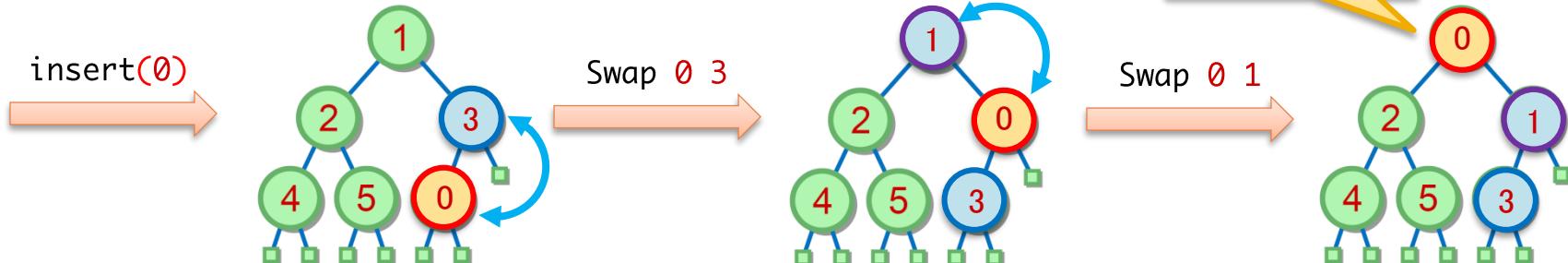
Para reestablecer la  
propiedad HOP 😊



# Inserción en un Montículo Binario (V)

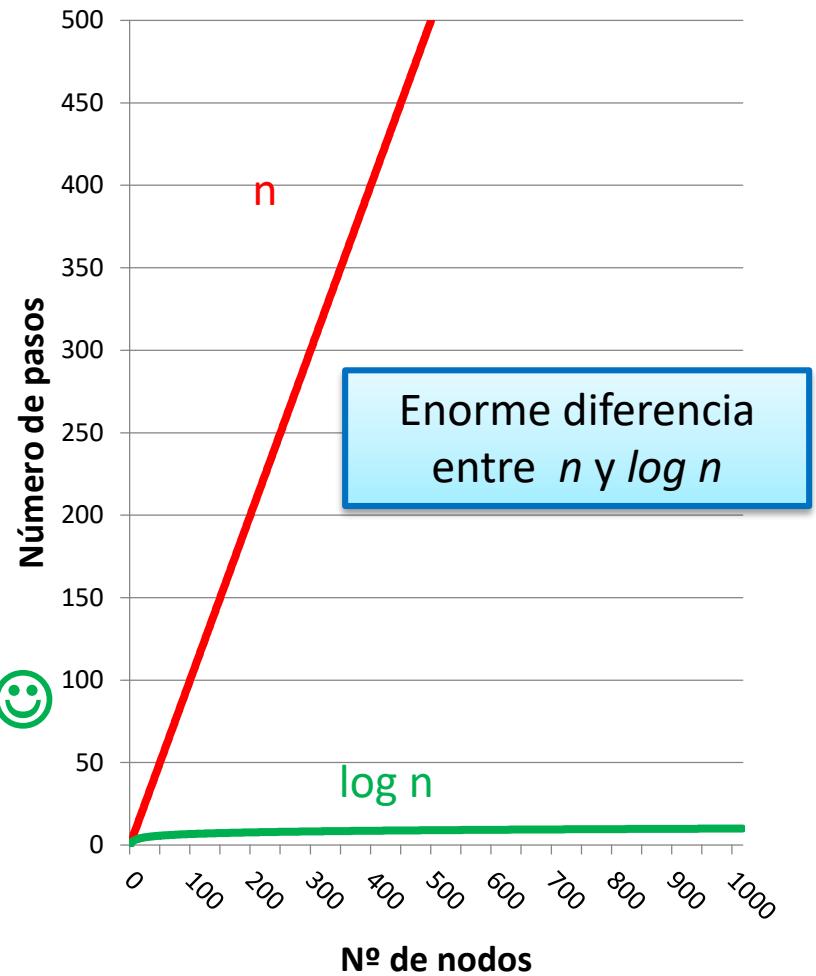
- Inserción: se realiza en dos fases
  - 1) Colocar el nuevo elemento en la primera posición libre:
    - A la derecha del último nodo del último nivel, o bien
    - A la izquierda del siguiente nivel si el último está completo
  - 2) Mientras que el nuevo elemento sea menor que el padre:
    - Intercambiar el nuevo con el padre

Para re establecer la  
propiedad HOP 😊



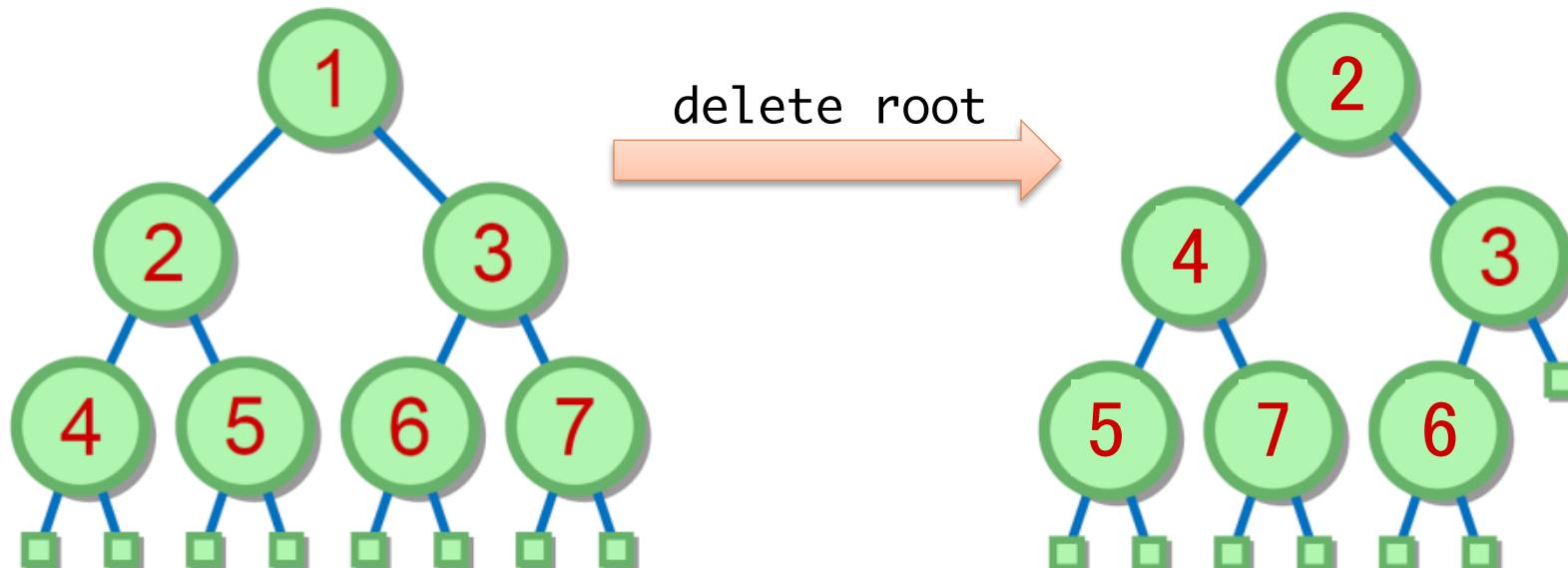
# Inserción en un Montículo Binario (y VI)

- Recordemos que un árbol completo con  $n$  elementos tiene altura  $1 + \lfloor \log_2(n) \rfloor$
- En el peor caso, la inserción necesita un número de pasos proporcional a  $\lfloor \log_2(n) \rfloor$
- Luego, insert está en  $O(\log n)$  😊



# Eliminación de la Raíz en un Montículo Binario

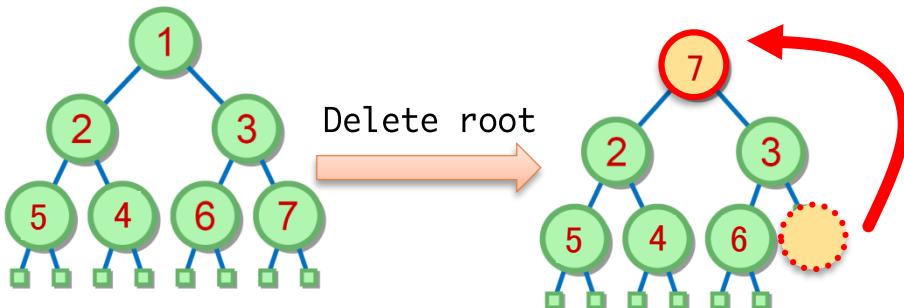
- Queremos eliminar la raíz de un montículo binario (el menor elemento) de forma que el resultado siga siendo un montículo binario



# Eliminación de la Raíz en un Montículo Binario (II)

- Sea  $\text{minChild}(u)$  el valor mínimo de los dos hijos del nodo  $u$ 
  - Este puede ser el izquierdo o el derecho
- La eliminación de la raíz se realiza en dos fases (subir el último, y hundirlo)
  - 1) Pasar el elemento más a la derecha del el último nivel ( $u$ ) a la raíz
  - 2) Mientras  $u > \text{minChild}(u)$  :
    - Intercambiar  $\text{minChild}(u)$  con  $u$

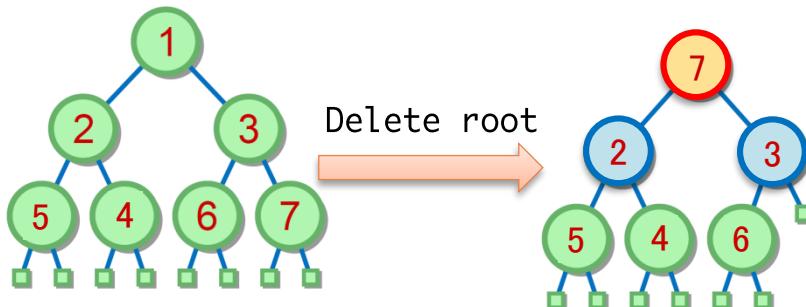
El árbol resultante sigue siendo completo 😊



# Eliminación de la Raíz en un Montículo Binario (III)

- Sea  $\text{minChild}(n)$  el valor mínimo de los dos hijos del nodo  $n$ 
  - Este puede ser el izquierdo o el derecho
- La eliminación de la raíz se realiza en dos fases
  - 1) Pasar el elemento más a la derecha del el último nivel ( $u$ ) a la raíz
  - 2) Mientras  $u > \text{minChild}(u)$  :
    - Intercambiar  $\text{minChild}(u)$  con  $u$

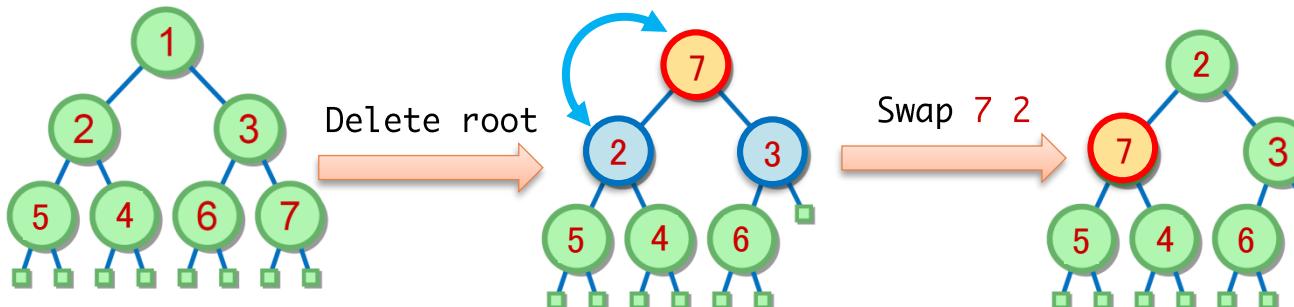
Reestablecer la  
propiedad HOP 😊c



# Eliminación de la Raíz en un Montículo Binario (IV)

- Sea  $\text{minChild}(n)$  el valor mínimo de los dos hijos del nodo  $n$ 
  - Este puede ser el izquierdo o el derecho
- La eliminación de la raíz se realiza en dos fases
  - 1) Pasar el elemento más a la derecha del el último nivel ( $u$ ) a la raíz
  - 2) Mientras  $u > \text{minChild}(u)$  :
    - Intercambiar  $\text{minChild}(u)$  con  $u$

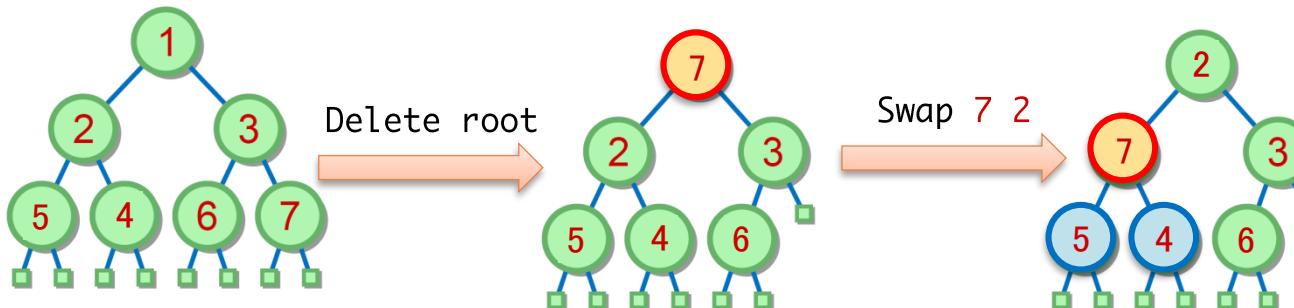
Reestablecer la  
propiedad HOP 😊c



# Eliminación de la Raíz en un Montículo Binario (V)

- Sea  $\text{minChild}(n)$  el valor mínimo de los dos hijos del nodo  $n$ 
  - Este puede ser el izquierdo o el derecho
- La eliminación de la raíz se realiza en dos fases
  - 1) Pasar el elemento más a la derecha del el último nivel ( $u$ ) a la raíz
  - 2) Mientras  $u > \text{minChild}(u)$  :
    - Intercambiar  $\text{minChild}(u)$  con  $u$

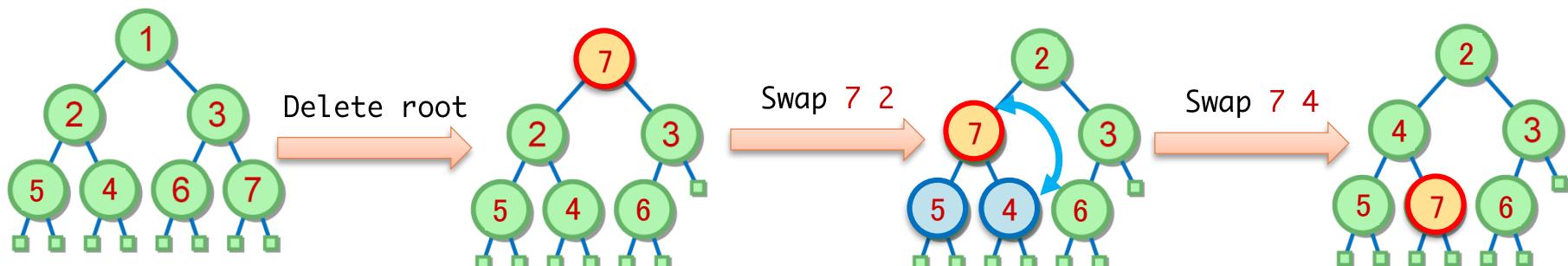
Reestablecer la  
propiedad HOP 😊c



# Eliminación de la Raíz en un Montículo Binario (y VI)

- Sea  $\text{minChild}(n)$  el valor mínimo de los dos hijos del nodo  $n$ 
  - Este puede ser el izquierdo o el derecho
- Eliminación de la raíz: se realiza en dos fases
  - 1) Pasar el elemento más a la derecha del el último nivel ( $u$ ) a la raíz
  - 2) Mientras  $u > \text{minChild}(u)$  :
    - Intercambiar  $\text{minChild}(u)$  con  $u$

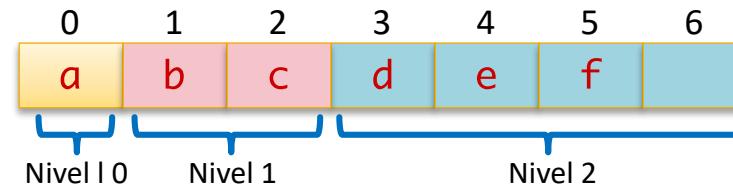
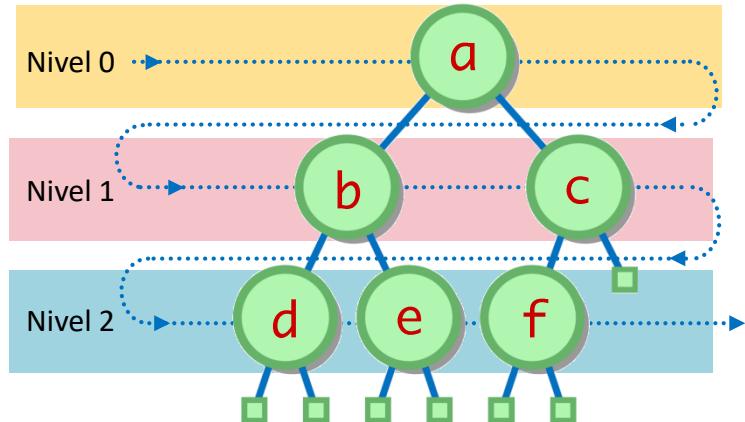
Reestablecer la  
propiedad HOP 😊c



El montículo  
final 😊

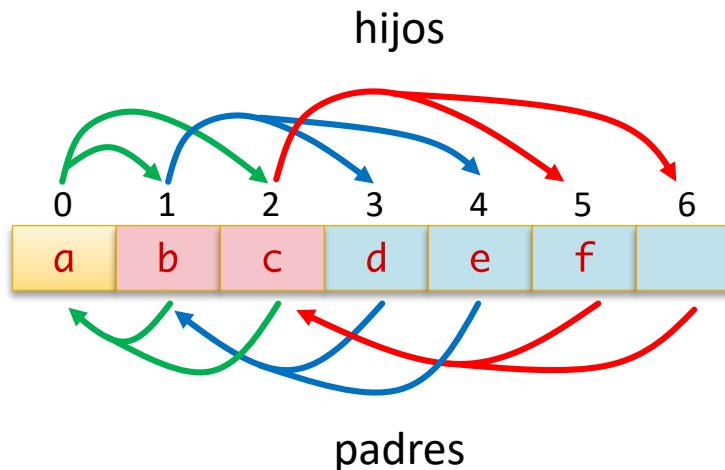
# Representación de un Árbol Binario Completo con un Array

- Los nodos de un árbol binario completo pueden colocarse fácilmente en forma consecutiva dentro de un array por niveles:



Padres e hijos pueden localizarse fácilmente vía **aritmética elemental**:

- La raíz tiene índice **0**
- Para el nodo de índice  $i$ :
  - El hijo izquierdo ocupa la pos.  $2*i+1$
  - El hijo derecho ocupa la pos.  $2*i+2$
  - El parente ocupa la pos.  $\lfloor (i-1)/2 \rfloor$



# Montículos en Java

```
package dataStructures.heap;

public interface Heap<T extends Comparable<? super T>> {

    boolean isEmpty();
    int size();
    void insert(T x);
    T minElem();
    void delMin();
}
```

El tipo T de los elementos del montículo debe implementar el método (relación de orden) compareTo

# Montículos Binarios en Java vía Arrays

```
package dataStructures.heap;

public class BinaryHeap<T extends Comparable<? super T>>
    implements Heap<T> {

    private T elems[]; // array to store heap elements
    private int size; // number of elements in heap

    private static int INITIAL_CAPACITY = 128;

    public BinaryHeap() {
        elems = (T[]) new Comparable[INITIAL_CAPACITY];
        size = 0;
    }

    private void ensureCapacity() {
        if(size == elems.length)
            elems = Arrays.copyOf(elems, 2*elems.length);
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

El tipo T de los elementos del montículo debe implementar el método (relación de orden) compareTo

# Montículos Binarios en Java vía Arrays (II)

```
// compares two elements: true if elems[idx1] < elems[idx2]
private boolean lessThan(int idx1, int idx2) {
    return elems[idx1].compareTo(elems[idx2]) < 0;
}

// swaps elements in array elems at positions idx1 and idx2
private void swap(int idx1, int idx2) {
    T aux = elems[idx1];
    elems[idx1] = elems[idx2];
    elems[idx2] = aux;
}
```

# Montículos Binarios en Java vía Arrays (III)

```
private static int ROOT_INDEX = 0; // root of heap is at index 0
private static boolean isRoot(int idx) { // true if idx is index at root of tree
    return idx==ROOT_INDEX;
}
private static int parent(int idx) { // index for parent of node with index idx
    return (idx-1) / 2; // integer division
}
private static int leftChild(int idx) { // index for left child of node with index idx
    return 2*idx+1;
}
private static int rightChild(int idx) { // index for right child of node with index idx
    return 1+leftChild(idx);
}
private boolean isNode(int idx) { // true if idx corresponds to index of a node in tree
    return idx<size;
}
private boolean hasLeftChild(int idx) { // true if node with index idx has a left child
    return leftChild(idx)<size;
}
private boolean isLeaf(int idx) { //returns true if node with index idx is a leaf node
    return !hasLeftChild(idx); // !hasLeftChild(idx) => !hasRightChild(idx)
}
```

# Montículos Binarios en Java vía Arrays (IV)

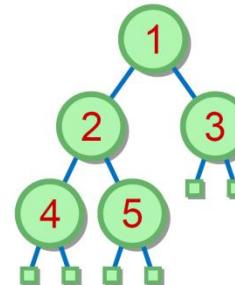
```
private void heapifyUp(int idx) {  
    while(!isRoot(idx)) {  
        int idxParent = parent(idx);  
  
        if(lessThan(idx, idxParent)) {  
            swap(idx, idxParent);  
            idx = idxParent; // move to parent node for next iteration  
        } else  
            break;  
    }  
  
    public void insert(T x) {  
        ensureCapacity();  
        elems[size] = x; // put new element at right in last level of tree  
        heapifyUp(size); // move element up to enforce heap-order property  
        size++;  
    }
```

heapifyUp es  $O(\log n)$  😊

# Montículos Binarios en Java vía Arrays (V)

```
public T minElem() {  
    if(isEmpty())  
        throw new EmptyHeapException("minElem on empty heap");  
    else  
        return elems[ROOT_INDEX];  
}
```

El mínimo siempre  
se encuentra en la  
raíz 😊



# Montículos Binarios en Java vía Arrays (VI)

```
private void heapifyDown() {  
    int idx = ROOT_INDEX; // start at root of tree  
  
    while(!isLeaf(idx)) {  
        int idxChild = leftChild(idx);  
        int idxRightChild = rightChild(idx);  
        if(isNode(idxRightChild) && lessThan(idxRightChild, idxChild))  
            idxChild = idxRightChild;  
        // idxChild es el índice del hijo con el menor de los valores  
        if(lessThan(idxChild, idx)) {  
            swap(idxChild, idx);  
            idx = idxChild; // move to child node for next iteration  
        } else  
            break;  
    }  
  
    public void delMin() {  
        ifisEmpty())  
            throw new EmptyHeapException("delMin on empty heap");  
        else {  
            elems[ROOT_INDEX] = elems[size-1]; // move last child to root of tree  
            size--;  
            heapifyDown(); // move element down to enforce heap-order property  
        }  
    }
```

heapifyDown es O(log n) 😊

# Montículos Binarios en Java vía Arrays (y VII)

## ■ Coste de las operaciones:

Operación	Coste
BinaryHeap	$O(1)$
isEmpty	$O(1)$
minElem	$O(1)$
insert	$O(\log n)$
delMin	$O(\log n)$

$O(n)$  si redimensionamos el array

# Colas de Prioridad en Java

```
package dataStructures.priorityQueue;  
  
public interface PriorityQueue  
    <T extends Comparable<? super T>> {  
  
    boolean isEmpty();  
  
    void enqueue(T x);  
  
    T first();  
  
    void dequeue();  
}
```

# Cola de Prioridad implementada con un Montículo Binario

```
package dataStructures.priorityQueue;  
import dataStructures.heap.BinaryHeap;  
public class BinaryHeapPriorityQueue<T extends Comparable<? super T>>  
    implements PriorityQueue<T> {  
    private BinaryHeap<T> heap;  
    public BinaryHeapPriorityQueue() {  
        heap = new BinaryHeap<T>();  
    }  
    public boolean isEmpty() {  
        return heap.isEmpty();  
    }  
    public void enqueue(T x) {  
        heap.insert(x);  
    }  
    public T first() {  
        if (isEmpty())  
            throw new EmptyPriorityQueueException("first on empty priority queue");  
        else  
            return heap.minElem();  
    }  
    public void dequeue() {  
        if (isEmpty())  
            throw new EmptyPriorityQueueException("first on empty priority queue");  
        else  
            heap.delMin();  
    }  
}
```

## Implementation trivial:

Todos los métodos delegan en los métodos correspondientes del montículo

¿Qué pasa cuando se insertan dos o más valores con la misma prioridad?

# Cola de Prioridad Lineal vs Cola de Prioridad con Montículo Binario

- Test experimental
- Medimos el tiempo de ejecución para realizar 100000 operaciones aleatorias (enqueue or dequeue) en una cola de prioridad inicialmente vacía.
- Usando un procesador Intel i7 860 CPU:
  - El Montículo Binario es aproximadamente **150 más rápido** que la implementación lineal 😊

# Montículos Zurdos (Weight Biased Leftist Heaps)

- Llamamos **peso** de un nodo el número de elementos que *cuelgan* desde el nodo
- Representaremos un montículo vía un **árbol binario aumentado**: en cada nodo aparece (además de su valor y los hijos) su peso:

```
data Heap a = Empty | Node a Int (Heap a) (Heap a)
```

```
weight :: Heap a -> Int
```

```
weight Empty = 0
```

```
weight (Node _ w _) = w
```

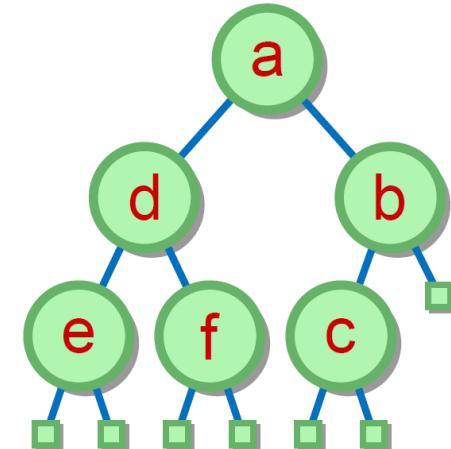
Peso del nodo

```
h1 :: Heap Char  
h1 = Node 'a' 6 (Node 'd' 3 (Node 'e' 1 Empty Empty))
```

Árbol con raíz 'a'  
con 6 elementos

```
(Node 'f' 1 Empty Empty)  
(Node 'b' 2 (Node 'c' 1 Empty Empty))  
Empty)
```

Árbol con raíz 'b'  
con 2 elementos

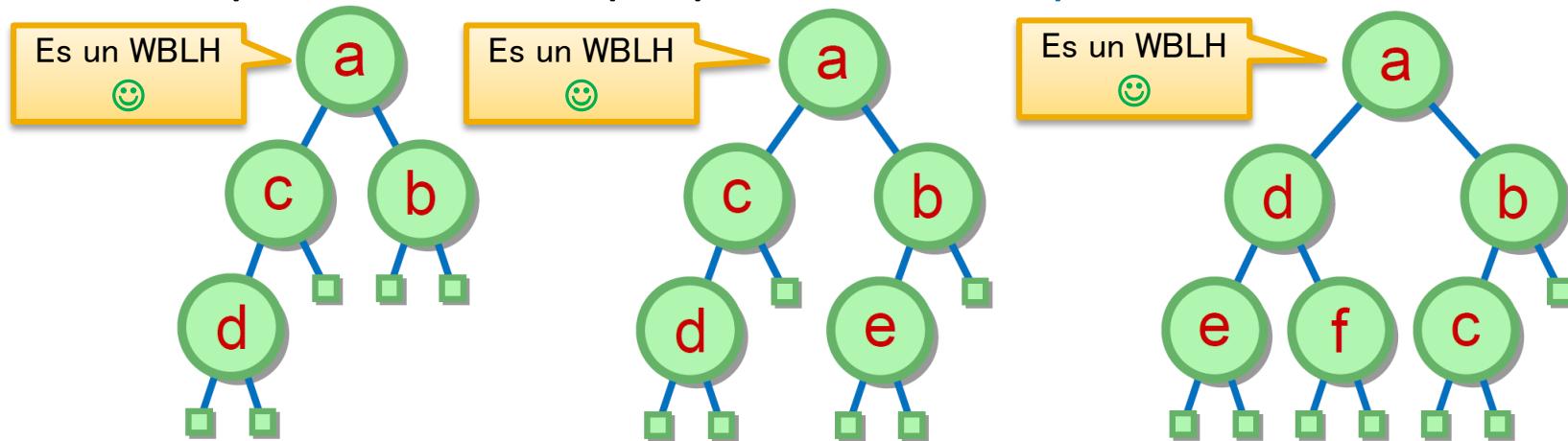


# Montículos Zurdos (WBLH) (II)

- Un árbol es Weight Biased Leftist si, para cualquier nodo en el árbol, el peso de su hijo izquierdo es mayor o igual que el peso de su hijo derecho

```
isWeightedLeftist :: Heap a -> Bool  
isWeightedLeftist Empty          = True  
isWeightedLeftist (Node _ _ lh rh) = weight lh >= weight rh  
                                    && isWeightedLeftist lh  
                                    && isWeightedLeftist rh
```

- Un heap Weight Biased Leftist (WBLH) es un árbol Weight Biased Leftist que satisface la propiedad de Heap-order



# Montículos Zurdos (WBLH) (III b)

```
module DataStructures.Heap.WBLeftistHeap
( Heap
, empty
, isEmpty
, minElem
, delMin
, insert
) where

data Heap a = Empty | Node a Int (Heap a) (Heap a) deriving Show

empty :: Heap a
empty = Empty

isEmpty :: Heap a -> Bool
isEmpty Empty = True
isEmpty _ = False

minElem :: Heap a -> a
minElem Empty = error "minElem on empty heap"
minElem (Node x _ _ _) = x
```

El mínimo del montículo  
es la raíz del arbol

# Montículos Zurdos (WBLH) (III)

```
delMin :: (Ord a) => Heap a -> Heap a  
delMin Empty          = error "delMin on empty heap"  
delMin (Node _ _ lh rh) = merge lh rh
```

elimina la raíz (el menor) y mezcla los hijos

```
singleton :: a -> Heap a  
singleton x = Node x 1 Empty Empty
```

```
insert :: (Ord a) => a -> Heap a -> Heap a  
insert x h = merge (singleton x) h
```

Crea un montículo de un elemento y lo mezcla con el original

```
merge :: (Ord a) => Heap a -> Heap a -> Heap a
```

...

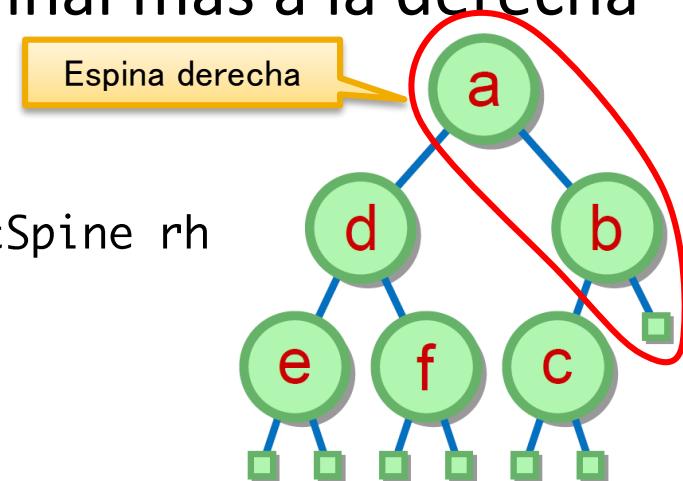
En definitiva: las operaciones delicadas (`insert` y `delMin`) y su complejidad dependen de la definición de `merge`.

Definiremos `merge` para que su complejidad sea logarítmica 😊

# Montículos Zurdos WBLH (III c)

- Sea llamada **espina derecha** de un árbol binario al camino desde la raíz hasta el nodo terminal más a la derecha

```
rightSpine :: Heap a -> [a]  
rightSpine Empty      = []  
rightSpine (Node x _ _ rh) = x : rightSpine rh
```



- Para cualquier WBLT con  $n$  elementos,  
la longitud de su espina derecha es  $\leq \lfloor \log_2(n+1) \rfloor$

esto implicará que la complejidad de merge es logarítmica

# La Espina Derecha tiene Longitud Logarítmica

```
-- longitud de la espina derecha
lrs :: Heap a -> Int
lrs Empty          = 0
lrs (Node x _ _ rh) = 1 + lrs rh
```

- Teorema.- Para cualquier montículo zurdo (WBLT)  $t$  con  $n$  elementos, la longitud de la espina derecha satisface

$$\text{lrs } t \leq \lfloor \log_2(n+1) \rfloor$$

- Por inducción sobre  $t$ ; hay que probar:
- Caso base:  
$$\text{lrs Empty} \leq \lfloor \log_2(0+1) \rfloor$$
- Paso inductivo: siendo  $p = \text{weight lh}$ ,  $q = \text{weight rh}$

si  $\text{lrs lh} \leq \lfloor \log_2(p+1) \rfloor$  &&  $\text{lrs rh} \leq \lfloor \log_2(q+1) \rfloor$   
entonces  $\text{lrs (Node x lh rh)} \leq \lfloor \log_2(p+q+2) \rfloor$

# La Espina Derecha tiene Longitud Logarítmica (II)

## ■ Caso base:

```
-- length of right spine
lrs :: Heap a -> Int
lrs Empty          = 0
lrs (Node x _ _ rh) = 1 + lrs rh
```

$$lrs \text{ Empty} \leq \lfloor \log_2(0+1) \rfloor$$

↔ {- 1<sup>a</sup>) ecuación de lrs (parte izquierda) -}

$$0 \leq \lfloor \log_2(0+1) \rfloor$$

↔ {- aritmética de log (parte derecha) -}

$$0 \leq 0$$

↔

True

# La Espina Derecha tiene Longitud Logarítmica (III)

## Paso inductivo:

Sean  $p = \text{weight lh}$  y  $q = \text{weight rh}$

si  $\text{lrs lh} \leq \lfloor \log_2(p+1) \rfloor$  &  $\text{lrs rh} \leq \lfloor \log_2(q+1) \rfloor$   
entonces  $\text{lrs } (\text{Node } x \ _ \ _ \text{ lh } \text{ rh}) \leq \lfloor \log_2(p+q+2) \rfloor$

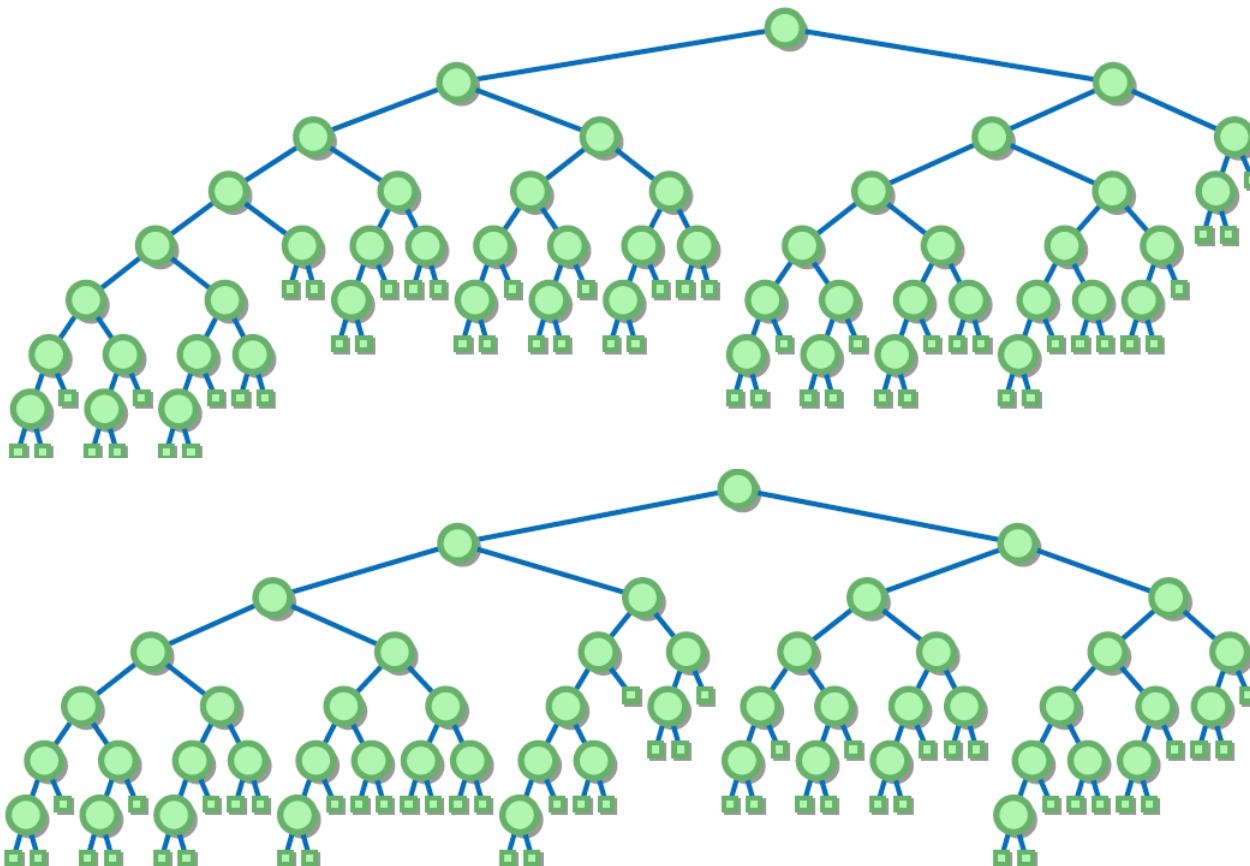
```
-- longitud de la espina derecha
lrs :: Heap a -> Int
lrs Empty          = 0
lrs (Node x _ _ rh) = 1 + lrs rh
```

Hipótesis de inducción

$\text{lrs } (\text{Node } x \ _ \ _ \text{ lh } \text{ rh}) \leq \lfloor \log_2(p+q+2) \rfloor$   
 $\Leftrightarrow \{- 2^a\} \text{ lrs al miembro izdo. de la desigualdad } -\}$   
 $1 + \text{lrs rh} \leq \lfloor \log_2(p+q+2) \rfloor$   
 $\Leftarrow \{- \text{Hipótesis de inducción, transitividad de } \leq -\}$   
 $1 + \lfloor \log_2(q+1) \rfloor \leq \lfloor \log_2(p+q+2) \rfloor$   
 $\Leftarrow \{- 1 = \log_2 2, \quad \log x + \log y = \log(x \cdot y) -\}$   
 $\lfloor \log_2(2q+2) \rfloor \leq \lfloor \log_2(p+q+2) \rfloor$   
 $\Leftarrow \{- \lfloor \rfloor \text{ y } \log_2 \text{ son monótonas}\}$   
 $2q+2 \leq p+q+2$   
 $\Leftrightarrow \{- p \geq q \text{ porque el montículo es zurdo } -\}$   
True

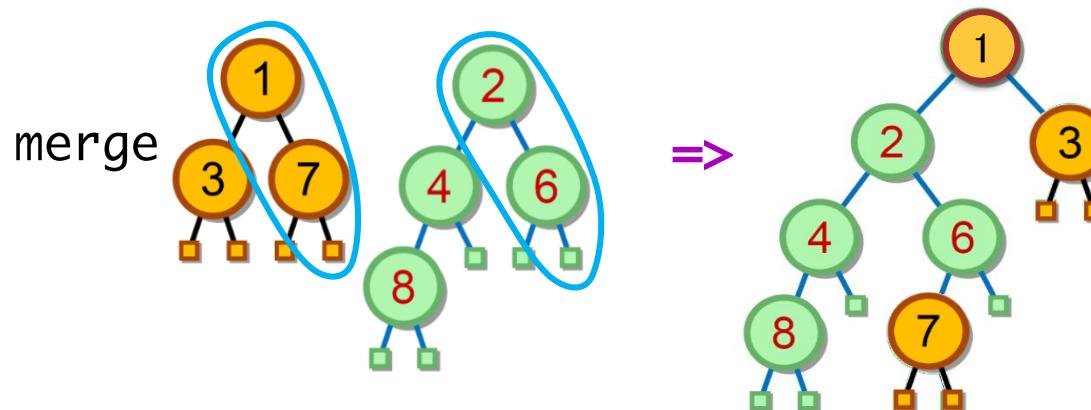
# La Espina Derecha tiene Longitud Logarítmica (y IV)

- Dos montículos zurdos construidos con 50 elementos aleatorios:  
¡la espina derecha es muy corta!



# Mezcla de Montículos Zurdos (WBLH)

- Pretendemos obtener un nuevo WBLH mezclando dos WBLHs



- Y pretendemos obtenerlo eficientemente:
  - Mezclándolos a través de sus espinas derechas ( $O(\log n)$ )

# Intermezzo: Mezcla de dos Listas Ordenadas

- Queremos mezclar dos listas ordenadas para obtener una lista ordenada:

merge [1,7,9] [0,2,4]

→ -- sacamos la menor de las cabezas y seguimos mezclando

0 : merge [1,7,9] [2,4]

→ 0 : 1 : merge [7,9] [2,4]

→ 0 : 1 : 2 : merge [7,9] [4]

→ 0 : 1 : 2 : 4 : merge [7,9] []

→ 0 : 1 : 2 : 4 : [7,9]

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge []      ys    = ys
merge xs     []    = xs
merge ls@(x:xs) rs@(y:ys)
| x <= y        = x : merge xs rs
| otherwise      = y : merge ls ys
```

El algoritmo anterior puede generalizarse para montículos

# Mezcla de Montículos Zurdos (WBLH) (II)

- Necesitamos una función auxiliar para construir un árbol zurdo a partir de otros dos y de un valor:

```
node :: a -> Heap a -> Heap a -> Heap a
```

```
node x h h'
```

```
| w >= w'      = Node x s h h'  
| otherwise     = Node x s h' h
```

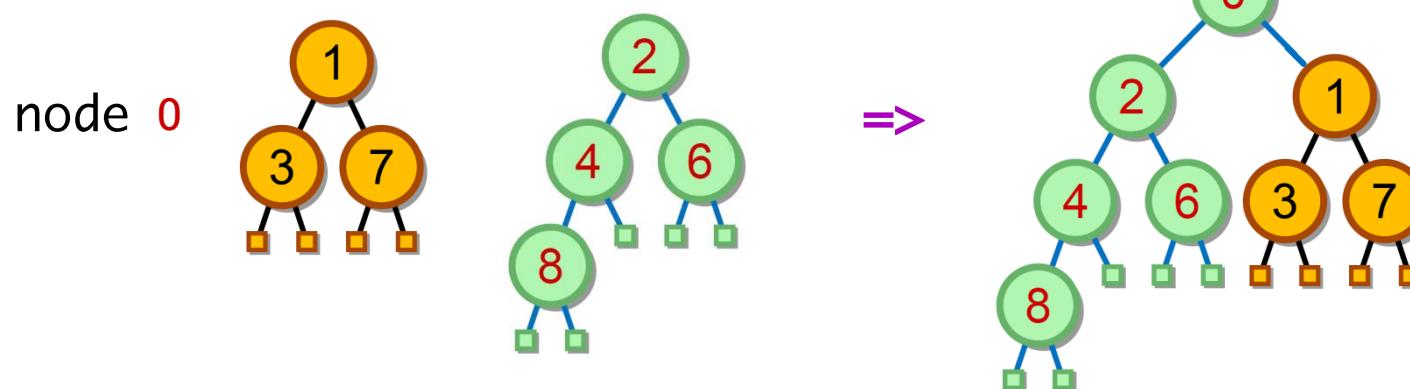
where

```
w   = weight h  
w'  = weight h'  
s   = w + w' + 1
```

Colocamos el de más peso a la izquierda

Manteniendo el invariante: si los argumentos son zurdos, el resultado también

El resultado es zurdo, pero **no** necesariamente ordenado, salvo que 0 no supere a las raíces



# Mezcla de Montículos Zurdos (WBLH) (III)

```
merge :: (Ord a) => Heap a -> Heap a -> Heap a  
merge Empty h'      = h'  
merge h    Empty     = h  
merge h@(Node x w lh rh) h'@(Node x' w' lh' rh')  
| x <= x'           = node x lh (merge rh h')  
| otherwise          = node x' lh' (merge h rh')
```

Conservando el invariante: el resultado es un montículo zurdo

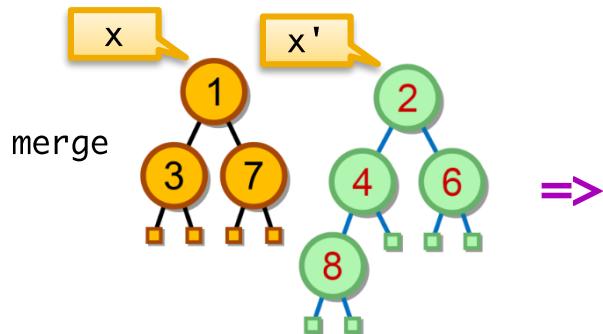
*extraemos* el menor nodo junto a su hijo izquierdo,  
y mezclamos el resto

- Si los argumentos son montículos zurdos, es posible demostrar por inducción:
  1. la mezcla es un montículo zurdo 😊
  2. el número de llamadas a merge es menor que la suma de las longitudes de las espinas derechas de los argumentos 😊

# Mezcla de Montículos Zurdos (WBLH) (IV)

```
merge :: (Ord a) => Heap a -> Heap a -> Heap a
merge Empty h'      = h'
merge h    Empty   = h
merge h@(Node x w lh rh) h'@(Node x' w' lh' rh')
| x <= x'          = node x lh (merge rh h')
| otherwise         = node x' lh' (merge h rh')
```

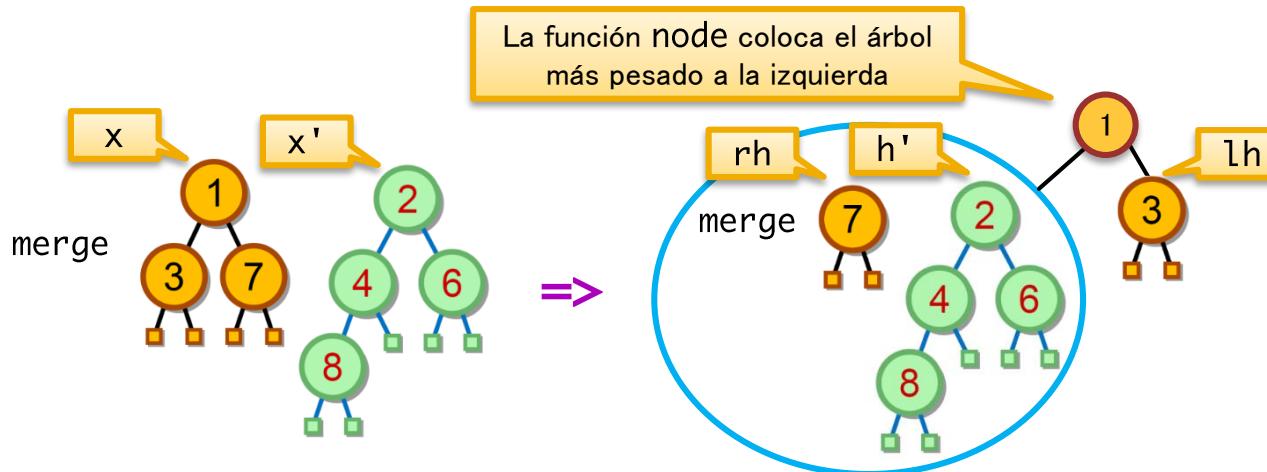
Conservando el  
invariante: el  
resultado es un  
montículo zurdo



# Mezcla de Montículos Zurdos (WBLH) (V)

```
merge :: (Ord a) => Heap a -> Heap a -> Heap a  
merge Empty h'      = h'  
merge h    Empty     = h  
merge h@(Node x w lh rh) h'@(Node x' w' lh' rh')  
| x <= x'           = node x lh (merge rh h')  
| otherwise          = node x' lh' (merge h rh')
```

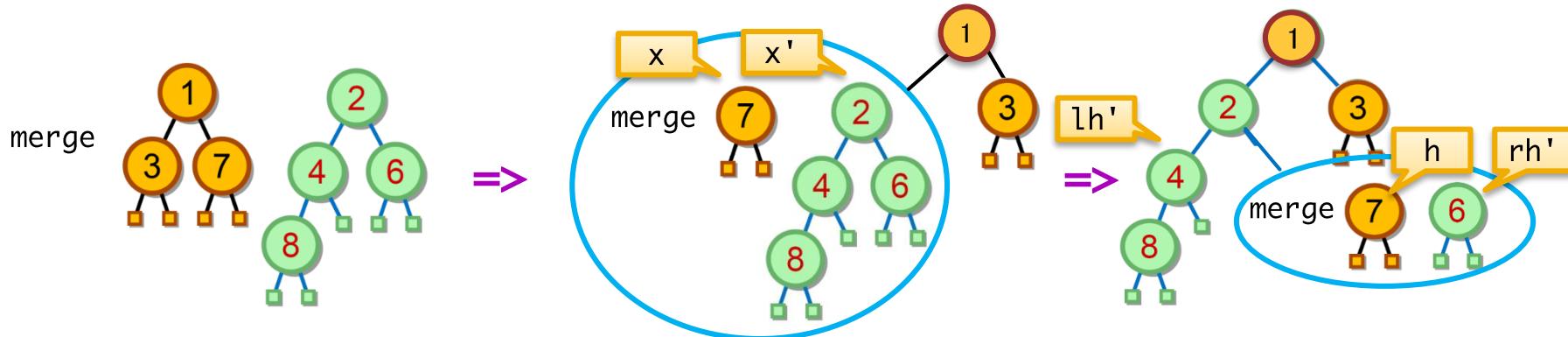
Conservando el invariante: el resultado es un montículo zurdo



# Mezcla de Montículos Zurdos (WBLH) (VI)

```
merge :: (Ord a) => Heap a -> Heap a -> Heap a
merge Empty h'      = h'
merge h    Empty     = h
merge h@(Node x w lh rh) h'@(Node x' w' lh' rh')
| x <= x'           = node x lh (merge rh h')
| otherwise          = node x' lh' (merge h rh')
```

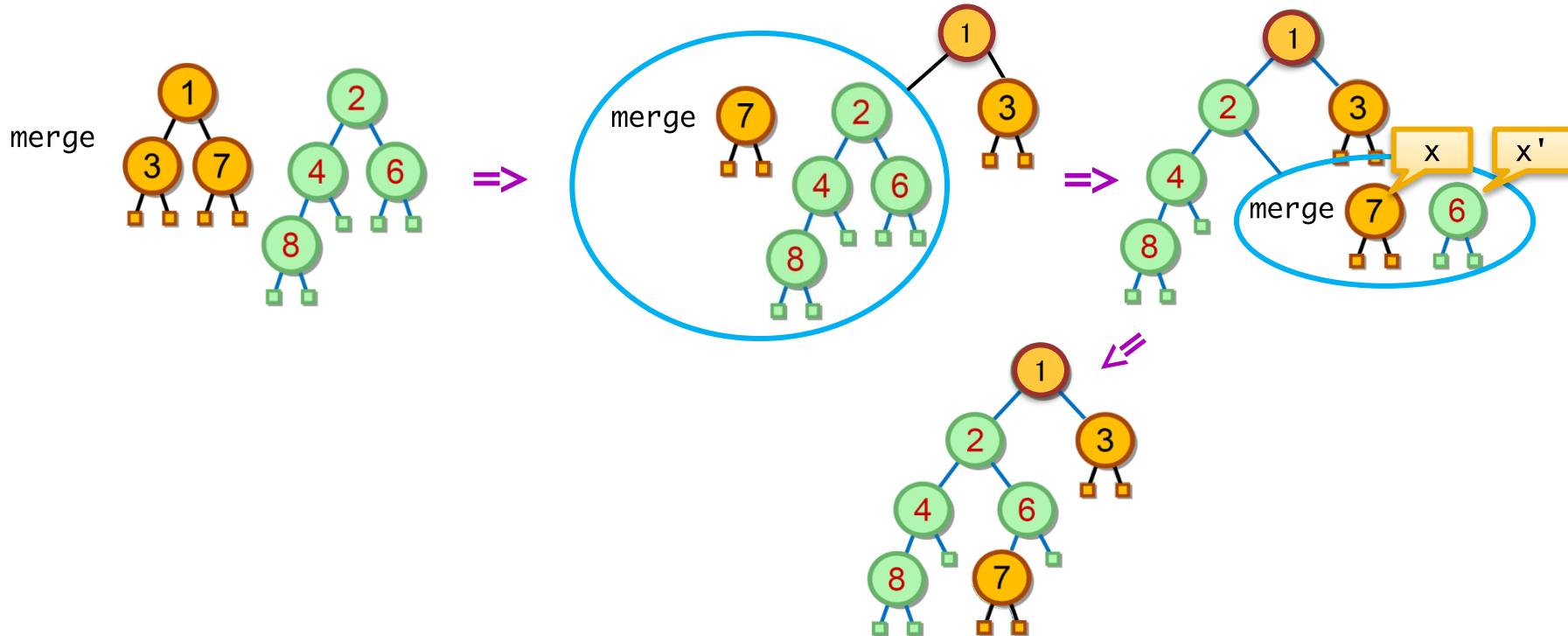
Conservando el  
invariante: el  
resultado es un  
montículo zurdo



# Mezcla de Montículos Zurdos (WBLH) (y VII)

```
merge :: (Ord a) => Heap a -> Heap a -> Heap a  
merge Empty h'      = h'  
merge h    Empty     = h  
merge h@(Node x w lh rh) h'@(Node x' w' lh' rh')  
| x <= x'           = node x lh (merge rh h')  
| otherwise          = node x' lh' (merge h rh')
```

Conservando el invariante: el resultado es un montículo zurdo



# Comportamiento de WBLeftistHeap

- Eficiencia de las operaciones:

Operación	Coste
empty	$O(1)$
isEmpty	$O(1)$
minElem	$O(1)$
merge	$O(\log n)$
insert	$O(\log n)$
delMin	$O(\log n)$

# Representación de Colas de Prioridad con Montículos Zurdos WBLHs

```
module DataStructures.PriorityQueue.WBLeftistPriorityQueue
  (PQueue
  ,empty
  ,isEmpty
  ,first
  ,dequeue
  ,enqueue
  ) where

import qualified DataStructures.Heap.WBLeftistHeap as H

data PQueue a = PQ (H.Heap a)

empty :: PQueue a
empty = PQ H.empty

isEmpty :: PQueue a -> Bool
isEmpty (PQ h) = H.isEmpty h

enqueue :: (Ord a) => a -> PQueue a -> PQueue a
enqueue x (PQ h) = PQ (H.insert x h)

first :: PQueue a -> a
first (PQ h) = H.minElem h

dequeue :: (Ord a) => PQueue a -> PQueue a
dequeue (PQ h) = PQ (H.delMin h)
```

# Construcción de un Montículo Zurdo a partir de una Lista en Tiempo Lineal

-- construcción en forma ascendente (bottom-up):  $O(n)$

`mkHeap :: (Ord a) => [a] -> Heap a`

`mkHeap []` = empty

`mkHeap xs` = mergeLoop (map singleton xs)

where

`mergeLoop [h]` = h

`mergeLoop hs` = mergeLoop (mergePairs hs)

Construimos una lista de montículos simples

Mezclamos dos a dos hasta obtener un solo elemento

`mergePairs []`

= []

mezcla dos a dos

`mergePairs [h]`

= [h]

`mergePairs (h:h':hs)` = merge h h' : mergePairs hs

- Sea  $T(n)$  el número de pasos de `mkHeap` para una lista con  $n$  elementos:

- $T(n) = n/2 \cdot O(\log_2 1) + n/4 \cdot O(\log_2 2) + n/8 \cdot O(\log_2 4) + \dots + 1 \cdot O(\log_2 (n/2))$

$n/2$  mezclas de montículos de 1 elemento

$n/4$  mezclas de montículos de 2 elementos

$n/8$  mezclas de montículos de 4 elementos

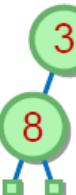
1 mezcla de montículos de  $n/2$  elementos

La solución  $T(n)$  es  $O(n)$  😊

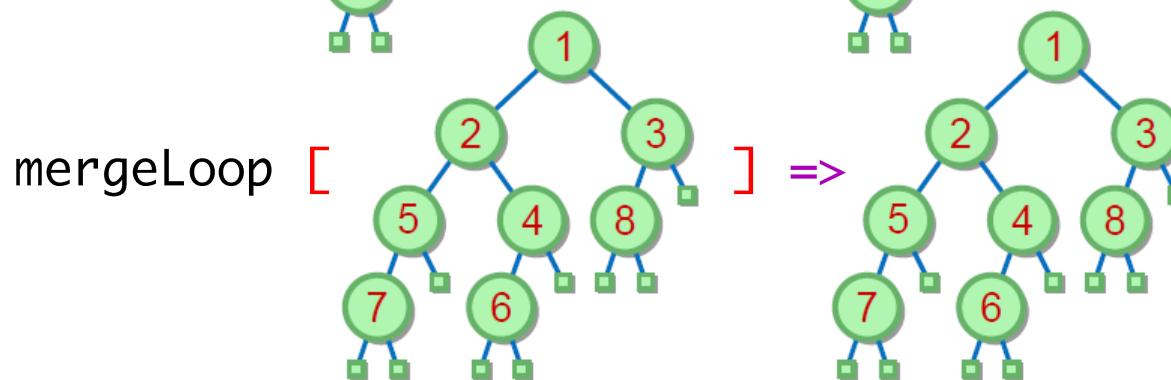
# Construcción de un Montículo Zurdo a partir de una Lista en Tiempo Lineal (y II)

mergeLoop (map singleton [8, 3, 1, 6, 2, 4, 5, 7]) =>

mergeLoop (mergePairs[, , , , , , , ]) =>

mergeLoop (mergePairs [, , , , , , , ]) =>

mergeLoop (mergePairs [, , , , , , ]) =>



# Heap Sort (Ordenación usando un Montículo)

```
heapSort :: (Ord a) => [a] -> [a]  
heapSort = heapToList . mkHeap
```

heapSort está en  
 $O(n \cdot \log n)$

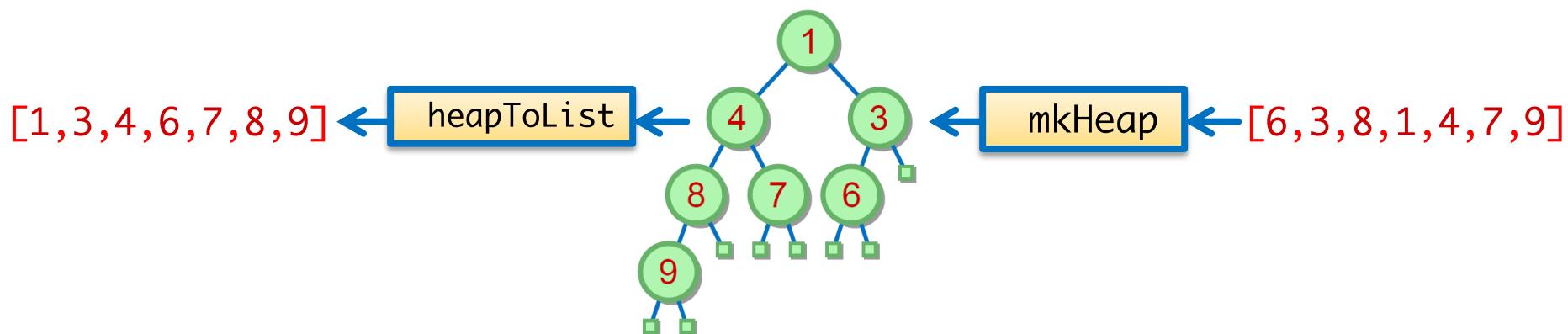
```
heapToList :: (Ord a) => Heap a -> [a]
```

```
heapToList h
```

```
| isEmpty h = []
```

```
| otherwise = minElem h : heapToList (delMin h)
```

$n$  eliminaciones donde cada una  
realiza  $O(\log n)$  pasos +  $O(n)$  para  
la operación mkHeap



# Montículos Zurdos en Java

```
package dataStructures.heap;

public class WBLeftistHeap <T extends Comparable<? super T>>
    implements Heap<T>{

    protected static class Tree<E> {
        E elem;           // elemento raíz del MZ
        int weight;       // peso o número de elementos
        Tree<E> left;   // hijo izdo
        Tree<E> right;  // hijo dcho
    }

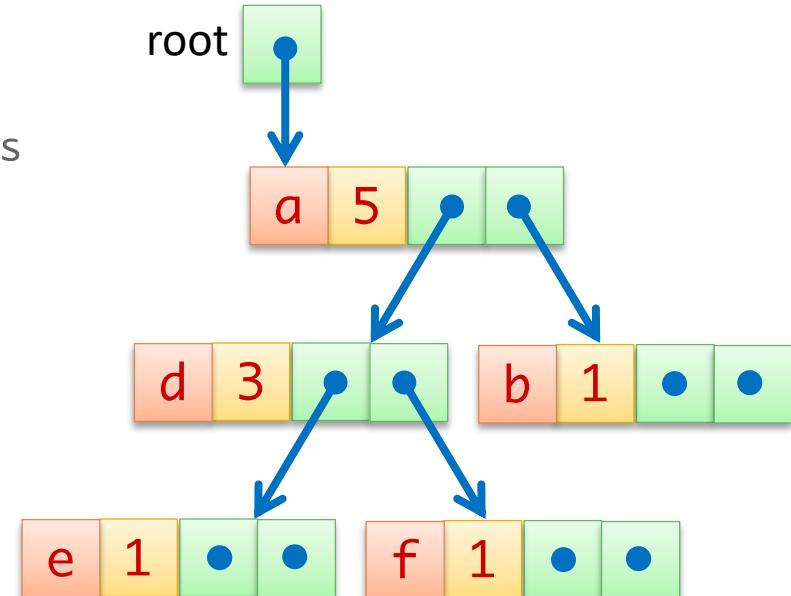
    // referencia a la raíz del montículo
    protected Tree<T> root;

    public WBLeftistHeap() {
        root = null;
    }

    public boolean isEmpty() {
        return root == null;
    }

    private static<T> int weight(Tree<T> t) {
        return t==null ? 0 : t.weight;
    }

    public int size() {
        return isEmpty() ? 0 : root.weight;
    }
}
```



● significa null

# Montículos Zurdos en Java (II)

```
private static <T> Tree<T> node(T x, Tree<T> h1, Tree<T> h2) {  
    int w1 = weight(h1);  
    int w2 = weight(h2);  
  
    Tree<T> tree = new Tree<T>();  
    tree.elem = x;  
    tree.weight = w1 + w2 + 1;  
    if (w1 >= w2) {  
        tree.left = h1;  
        tree.right = h2;  
    } else {  
        tree.left = h2;  
        tree.right = h1;  
    }  
    return tree;  
}
```

node :: a  $\rightarrow$  Heap a  $\rightarrow$  Heap a  $\rightarrow$  Heap a  
node x h h'  
| w  $\geq$  w' = Node x s h h'  
| otherwise = Node x s h' h  
where  
w = weight h  
w' = weight h'  
s = w + w' + 1

# Montículos Zurdos en Java (III)

```
private static <T extends Comparable<? super T>>
    Tree<T> merge(Tree<T> h1, Tree<T> h2) {
    if (h1 == null)
        return h2;
    if (h2 == null)
        return h1;

    T x1 = h1.elem;
    T x2 = h2.elem;
    if (x1.compareTo(x2) <= 0) { // x1 <= x2
        return node(x1, h1.left, merge(h1.right, h2));
    } else {
        return node(x2, h2.left, merge(h1, h2.right));
    }
}
```

merge :: (0rd a) => Heap a	->	Heap a	->	Heap a
merge Empty h'	=	h'		
merge h	Empty	=	h	
merge h@(Node x w lh rh)	h'@(Node x' w' lh' rh')			
x <= x'	=	node x lh (merge rh h')		
otherwise	=	node x' lh' (merge h rh')		

# Montículos Zurdos en Java (IV)

```
// mezcla dos MZs a lo largo de sus espinas derechas, devolviendo un MZ
private static <T extends Comparable<? super T>> Tree<T> merge(Tree<T> h1, Tree<T> h2) {
    if (h1 == null)
        return h2;
    if (h2 == null)
        return h1;

    if (h2.elem.compareTo(h1.elem) < 0) {// forzamos que h1 sea el de clave menor
        // intercambiamos h1 and h2
        Tree<T> aux = h1;
        h1 = h2;
        h2 = aux;
    } // la mezcla quedará referenciada por h1

    h1.right = merge(h1.right, h2); // mezcla sobre la espina derecha
    int wL = weight(h1.left);
    int wR = weight(h1.right);
    h1.weight = wL + wR + 1; // reajustamos el nuevo peso

    // intercambiamos los hijos de h1 si fuera necesario para que h1 quede zurdo
    if (wL < wR) {
        Tree<T> aux = h1.left;
        h1.left = h1.right;
        h1.right = aux;
    }

    return h1;
}
```

Versión eficiente en uso de memoria: reusa nodos de los árboles mezclados

# Montículos Zurdos en Java (y V)

```
public T minElem() {  
    if(isEmpty())  
        throw new EmptyHeapException("minElem on empty heap");  
    else  
        return root.elem;  
}
```

El mínimo en la raíz

```
public void delMin() {  
    if(isEmpty())  
        throw new EmptyHeapException("delMin on empty heap");  
    else  
        root = merge(root.left,root.right);  
}
```

Mezclamos los hijos  
sin la raíz

```
public void insert(T x) {  
    Tree<T> tree = new Tree<>();  
    tree.elem = x;  
    tree.weight = 1;  
    tree.left = null;  
    tree.right = null;  
    root = merge(root, tree);  
}
```

Creamos un nuevo MZ  
con un elemento

La inserción se produce  
al mezclar la raíz con el  
nuevo MZ

# Colas de Prioridad Enlazadas vs Zurdos vs Montículos Binarios

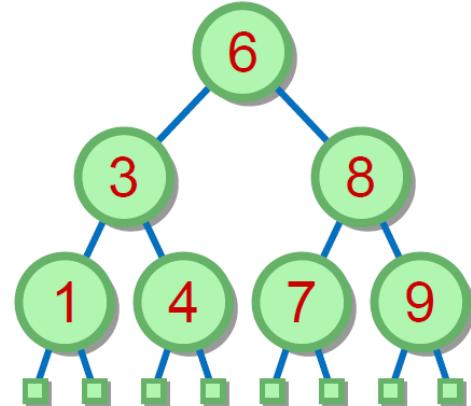
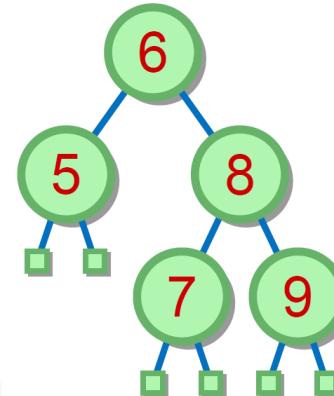
## ■ Test experimental

- Medimos el tiempo de ejecución para realizar 100000 operaciones aleatorias (enqueue or dequeue) en una cola de prioridad inicialmente vacía.
- Usando un procesador Intel i7 860 CPU:
  - Montículo Binario es aproximadamente 150 veces más rápido que la implementación enlazada
  - Zurdo es aproximadamente 290 veces más rápido que la implementación enlazada 😊
  - Zurdo es aproximadamente 2 veces más rápido que Binario 😊

# Árboles Binarios de Búsqueda. Binary Search Trees (BST)

- Un BST (Binary Search Tree) es un árbol binario tal que para **cada** nodo  $v$ ,
  - todos los elementos del subárbol izquierdo son menores que  $v$ , y
  - todos los elementos del subárbol derecho son mayores que  $v$

Invariante de un BST



```
data BST a = Empty | Node a (BST a) (BST a) deriving Show
```

```
isBST :: (Ord a) => BST a -> Bool
```

```
isBST Empty = True
```

```
isBST (Node x lt rt) = forAll (<x) lt && forAll (>x) rt  
                      && isBST lt && isBST rt
```

where

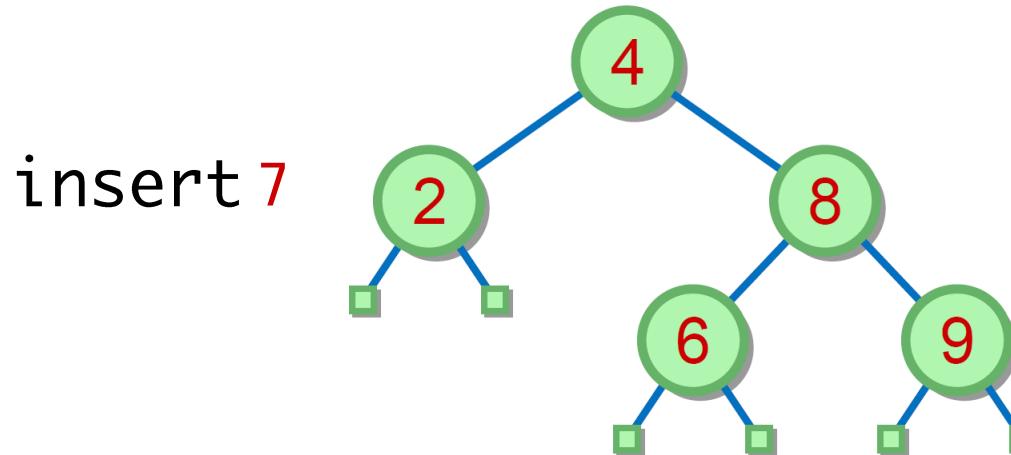
```
forAll :: (a -> Bool) -> BST a -> Bool
```

```
forAll p Empty = True
```

```
forAll p (Node x lt rt) = p x && forAll p lt && forAll p rt
```

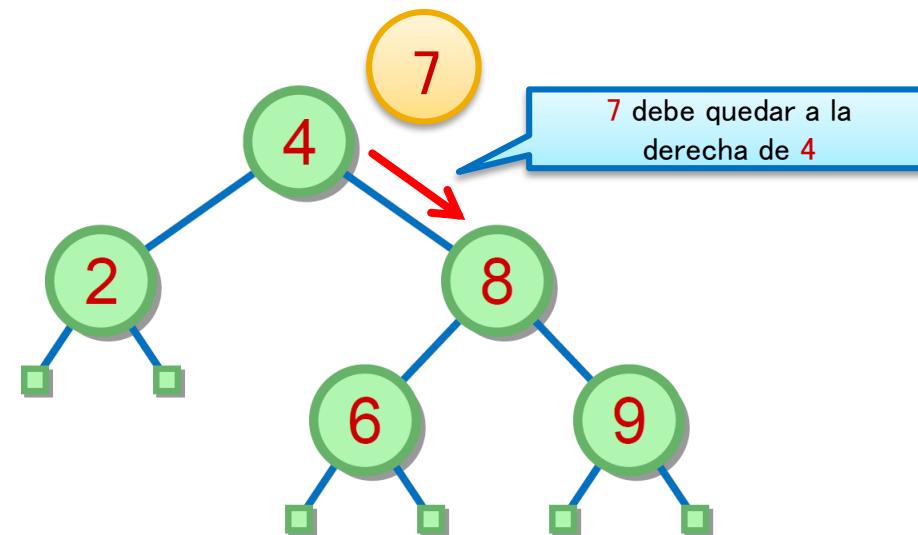
# Inserción en un BST

- Problema: insertar un **nuevo** elemento en un BST de forma que el resultado sea otro BST (se conserve el invariante)



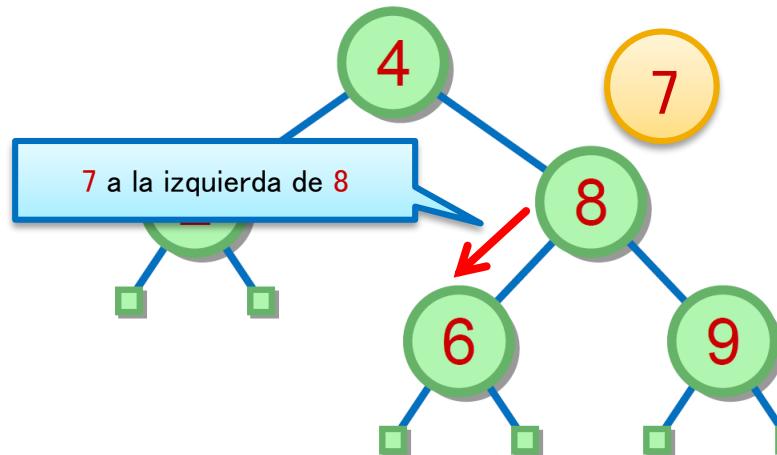
# Inserción en un BST (II)

- Problema: insertar un **nuevo** elemento en un BST de forma que el resultado sea otro BST (se conserve el invariante)



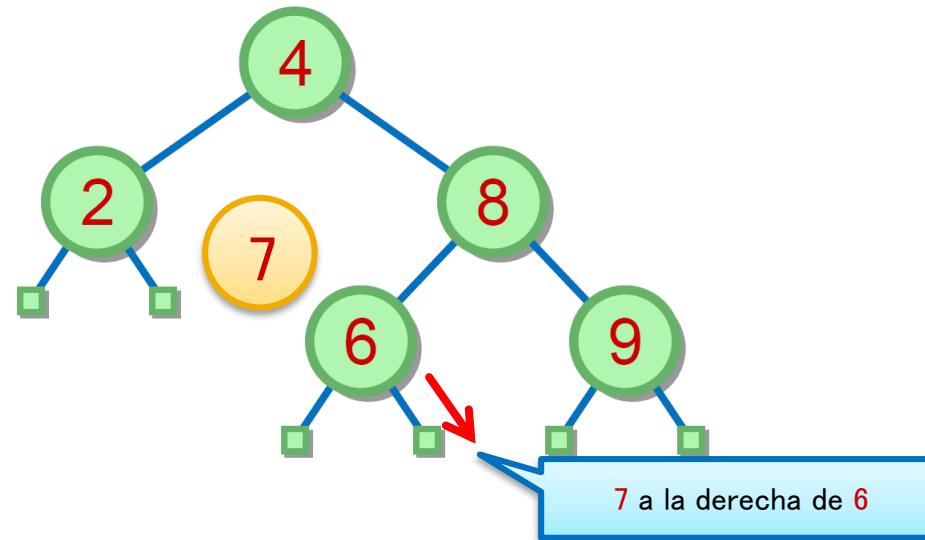
# Inserción en un BST (III)

- Problema: insertar un **nuevo** elemento en un BST de forma que el resultado sea otro BST (se conserve el invariante)



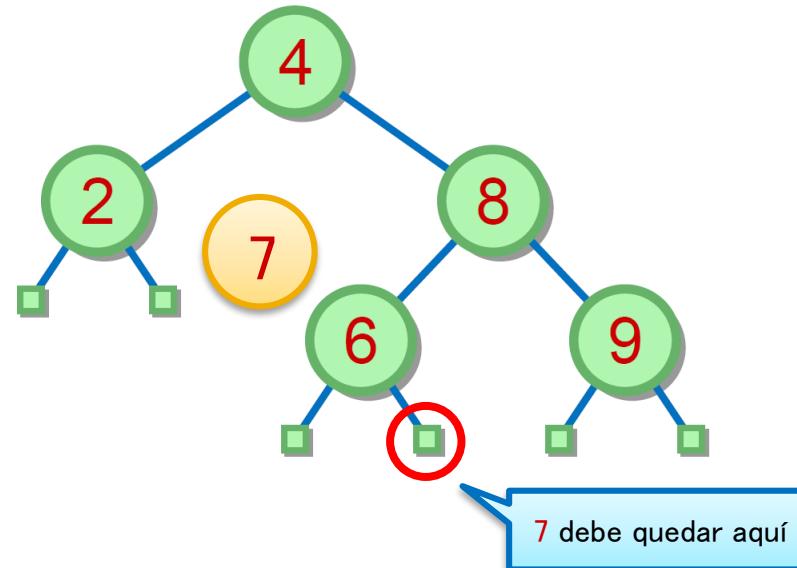
# Inserción en un BST (IV)

- Problema: insertar un **nuevo** elemento en un BST de forma que el resultado sea otro BST (se conserve el invariante)



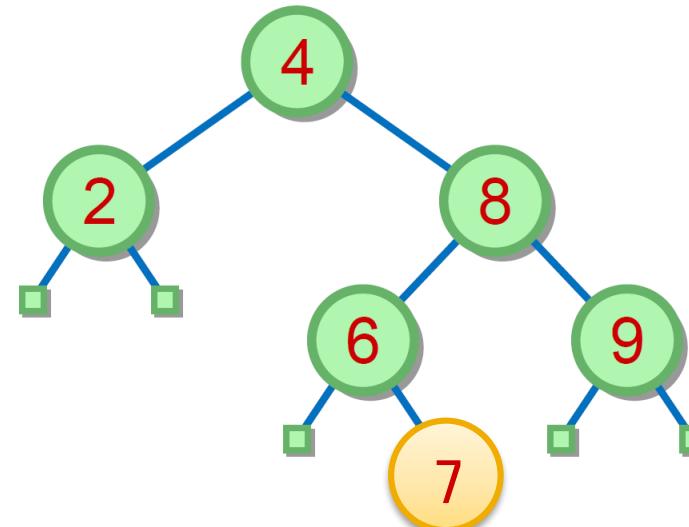
# Inserción en un BST (V)

- Problema: insertar un **nuevo** elemento en un BST de forma que el resultado sea otro BST (se conserve el invariante)



# Inserción en un BST (VI)

- Problema: insertar un **nuevo** elemento en un BST de forma que el resultado sea otro BST (se conserve el invariante)

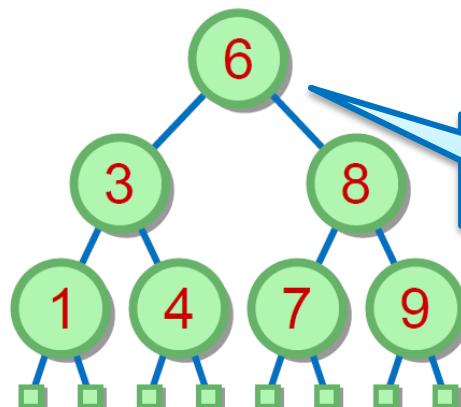


# Inserción en un BST (VII)

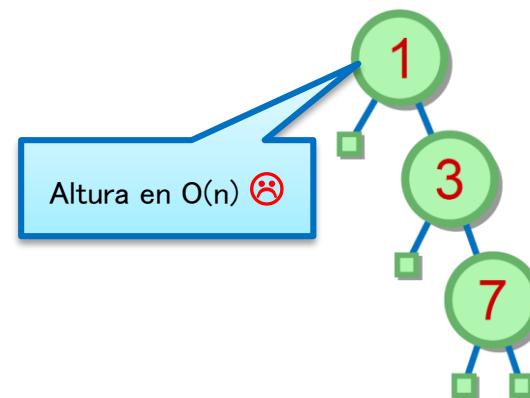
```
insert :: (Ord a) => a -> BST a -> BST a
insert x' Empty = Node x' Empty Empty
insert x' (Node x lt rt)
| x' == x    = Node x' lt rt
| x' < x     = Node x (insert x' lt) rt
| otherwise   = Node x lt (insert x' rt)
```

Si  $x' == x$ , entonces  $x$  es sustituido por  $x'$

- El número de pasos es proporcional a la altura del árbol:
- Para un árbol con  $n$  claves,  $\text{insert} \in O(n)$ , pero podría conseguirse, para cierto tipo de árboles,  $\text{insert} \in O(\log n)$ ; si, p.e., la altura está en  $O(\log n)$ .



Altura en  $O(\log n)$  😊

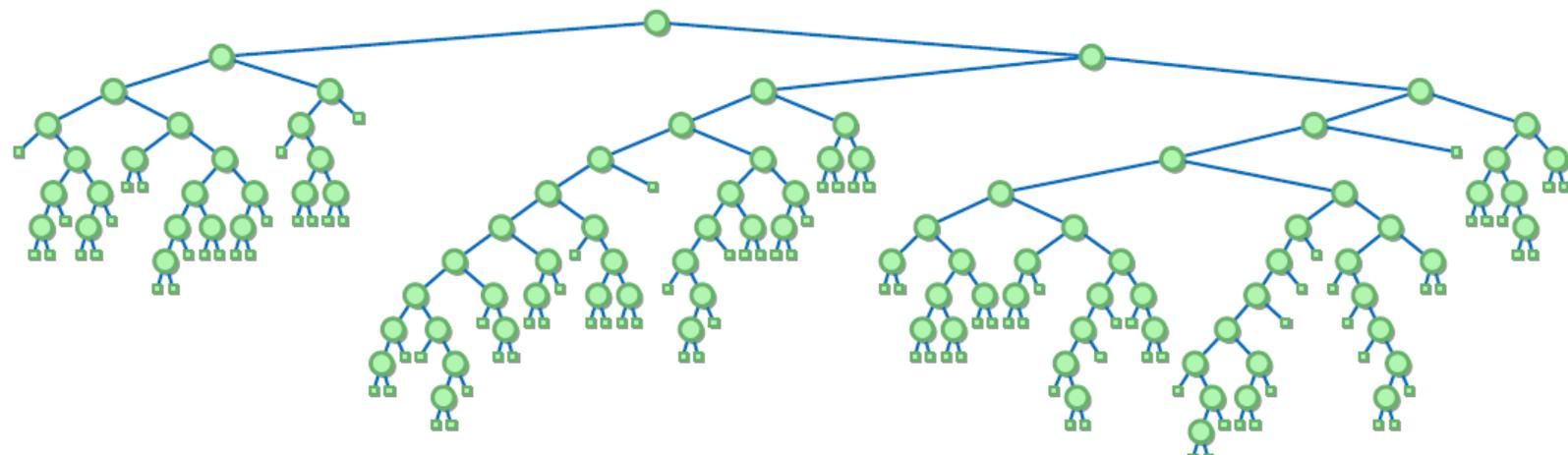
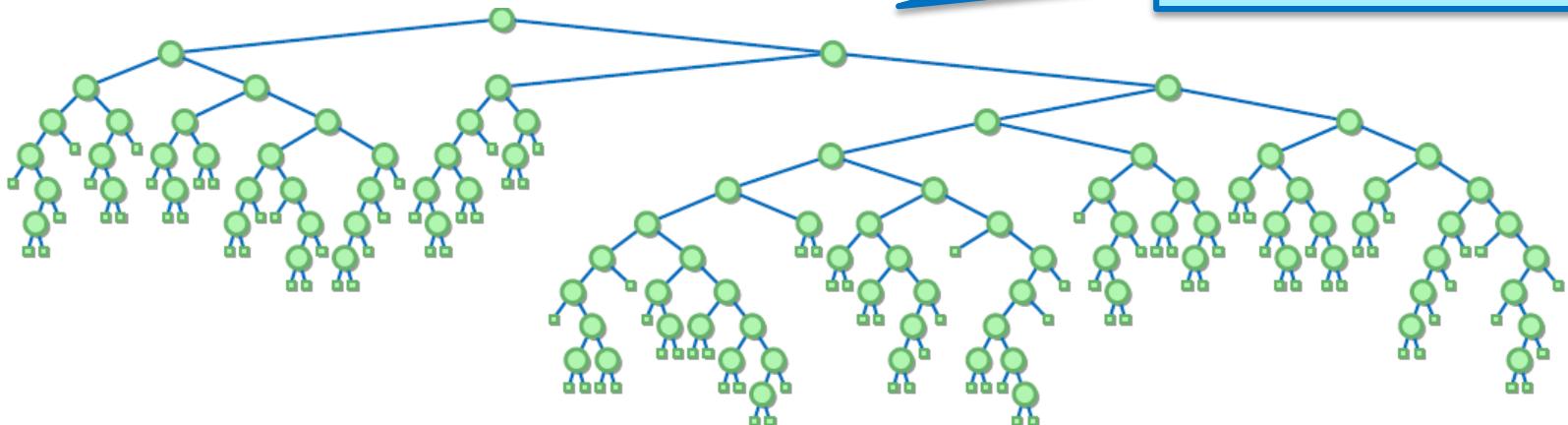


Altura en  $O(n)$  😞

# Inserción en un BST (VIII)

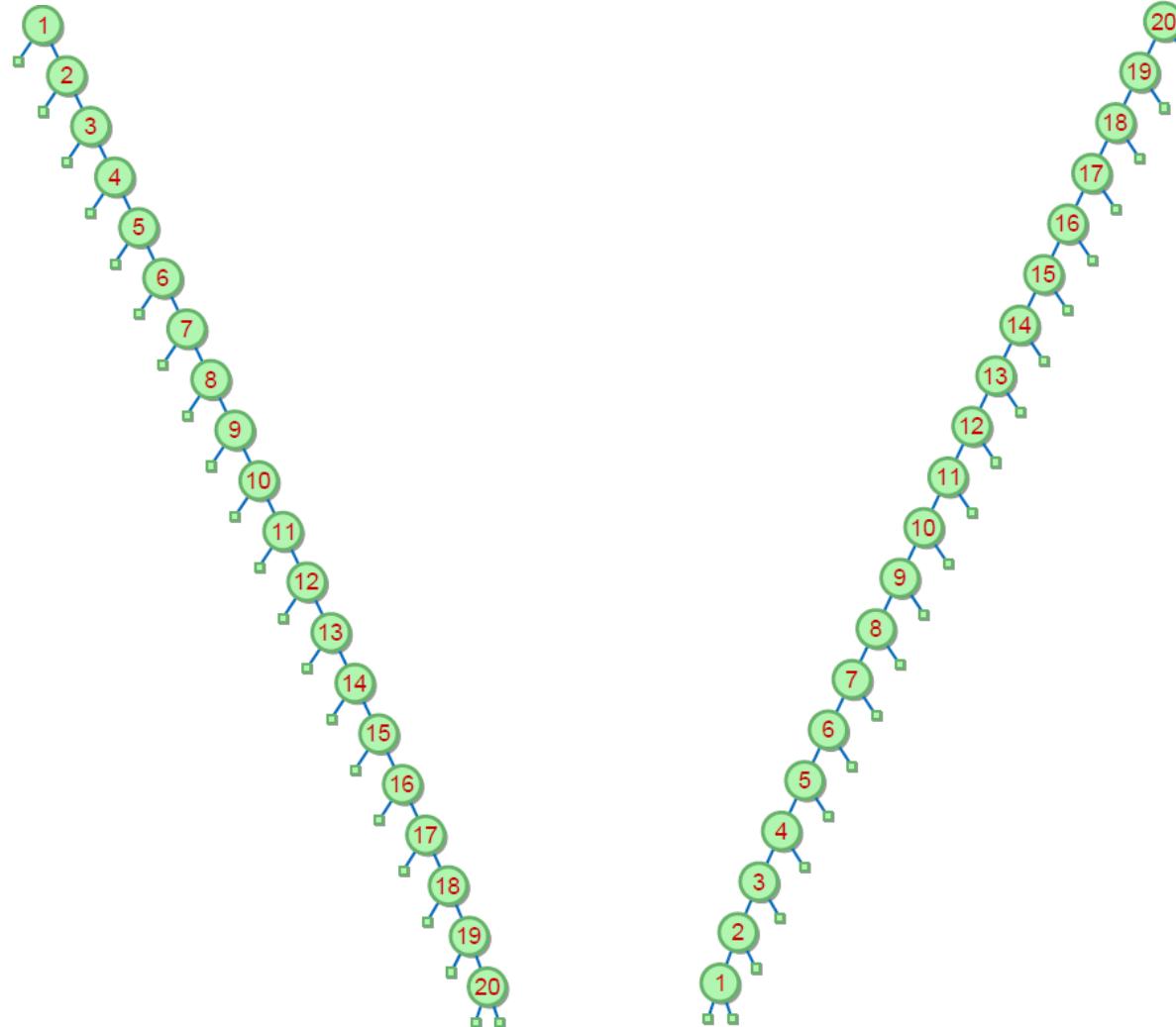
- Dos BSTs generados con elementos aleatorios:

Los árboles no son excesivamente desequilibrados 😊



# Inserción en un BST (y IX)

- La inserción de elementos ordenados produce BSTs degenerados

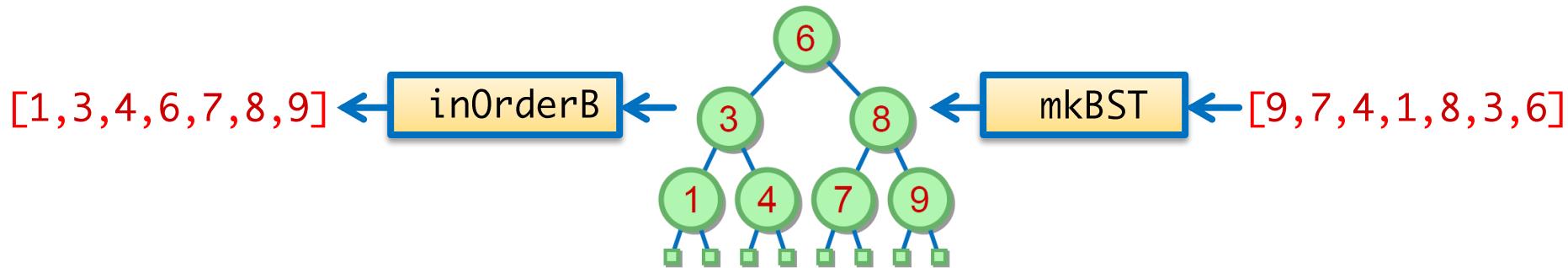


# Tree Sort (Ordenación usando un BST)

- Al recorrer *en orden* un BST obtenemos la lista ordenada de sus elementos (algoritmo Tree Sort)

```
mkBST :: (Ord a) => [a] -> BST a  
mkBST xs = foldr insert empty xs
```

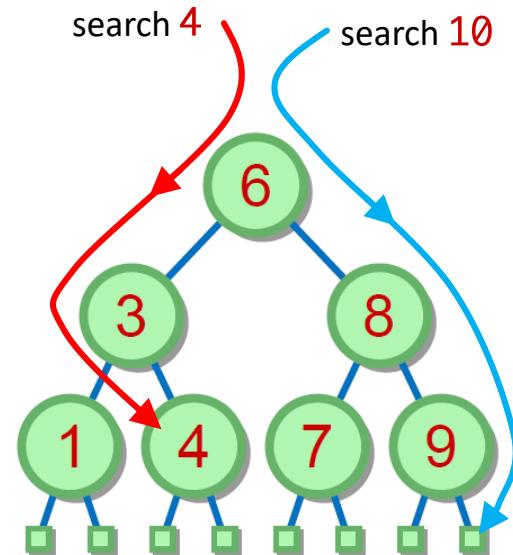
```
treeSort :: (Ord a) => [a] -> [a]  
treeSort = inOrderB . mkBST
```



# Búsqueda de un Elemento en un BST

```
search :: (Ord a) => a -> BST a -> Maybe a
search x' Empty = Nothing
search x' (Node x lt rt)
| x' == x      = Just x
| x' < x       = search x' lt
| otherwise     = search x' rt
```

```
isElem :: (Ord a) => a -> BST a -> Bool
isElem x t = isJust (search x t)
```



```
data Maybe a = Nothing | Just a
```

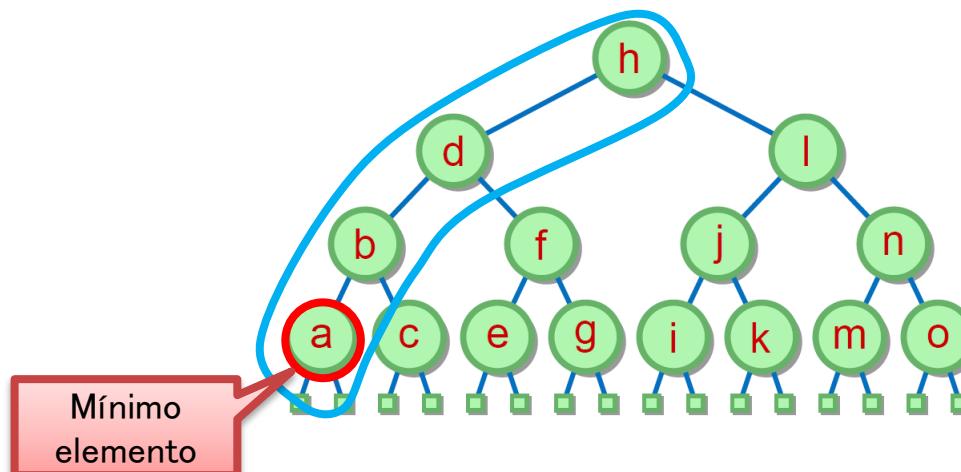
Maybe

```
isJust :: Maybe a -> Bool
isJust (Just _) = True
isJust Nothing = False
```

# Localización del Elemento Mínimo en un BST

- El mínimo elemento se encuentra en la posición más a la izquierda (al final de la espina izquierda):

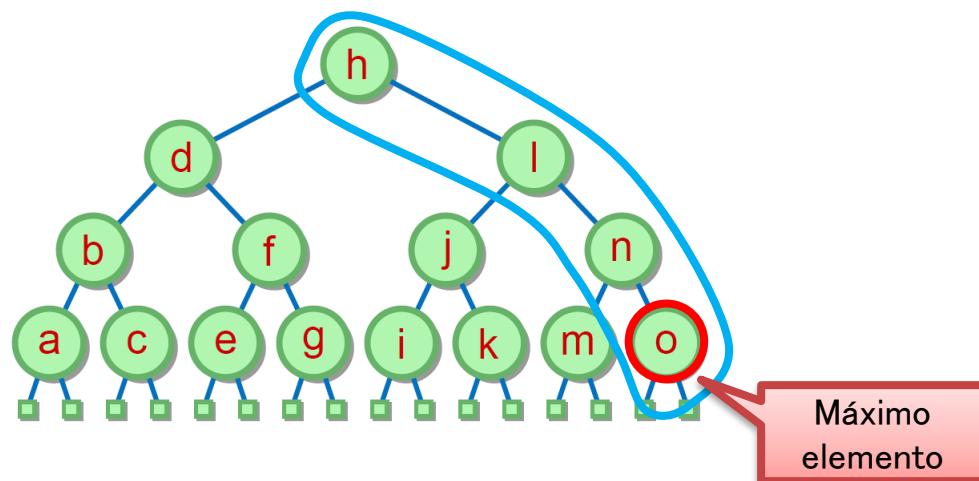
```
minim :: BST a -> a
minim Empty          = error "minim on empty tree"
minim (Node x Empty rt) = x
minim (Node x lt rt)   = minim lt
```



# Localización del Elemento Máximo en un BST

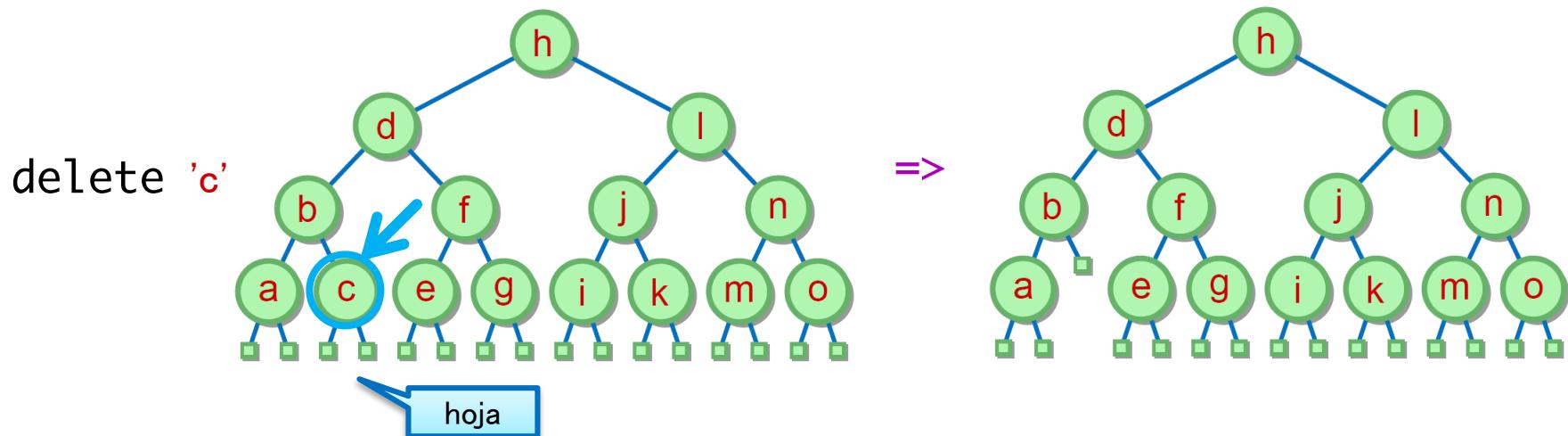
- El máximo elemento se encuentra al final de la espina derecha:

```
maxim :: BST a -> a
maxim Empty          = error "maxim on empty tree"
maxim (Node x lt Empty) = x
maxim (Node x lt rt)   = maxim rt
```



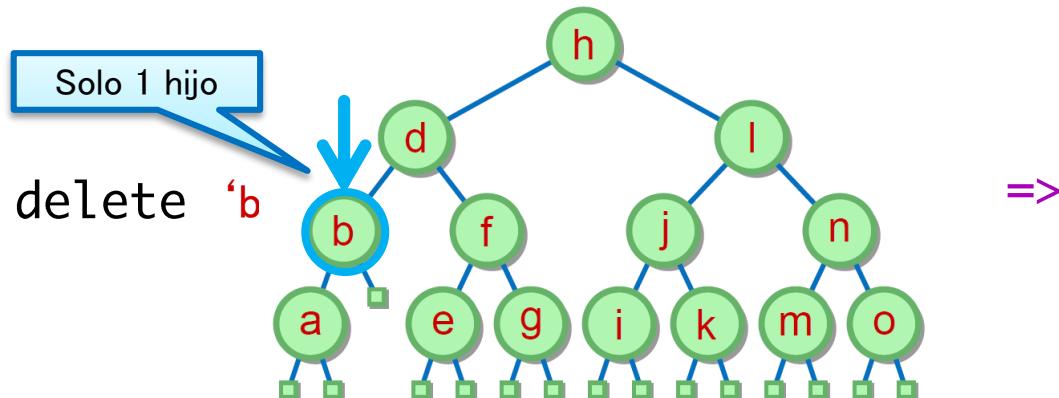
# Eliminación de un Elemento de un BST

- En la primera fase localizamos el elemento siguiendo un algoritmo parecido al de inserción.
- (a) Si el nodo a eliminar es una **hoja**, se elimina sin más



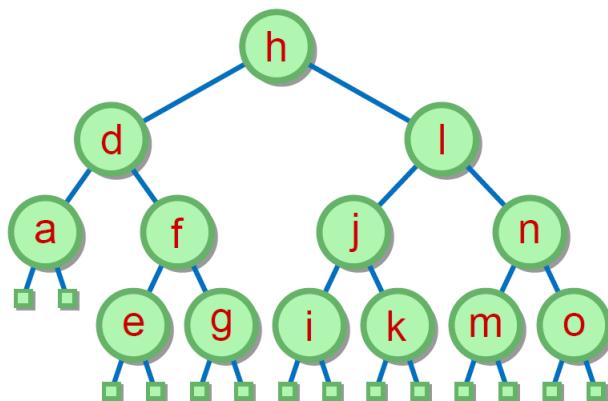
# Eliminación de un Elemento de un BST (II)

- En la primera fase localizamos el elemento siguiendo un algoritmo parecido al de inserción.
- (b) Si el nodo a eliminar tiene **un solo hijo**, el nodo padre puede conectarse con el nodo hijo.



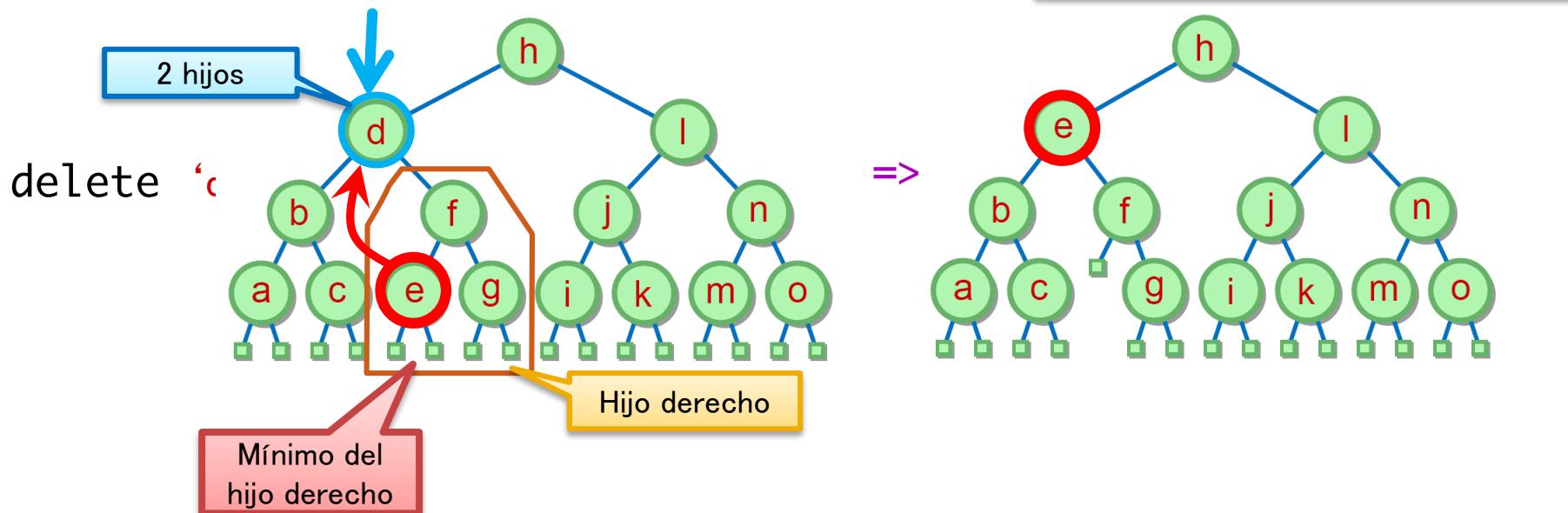
=>

Mantiene invariante el orden



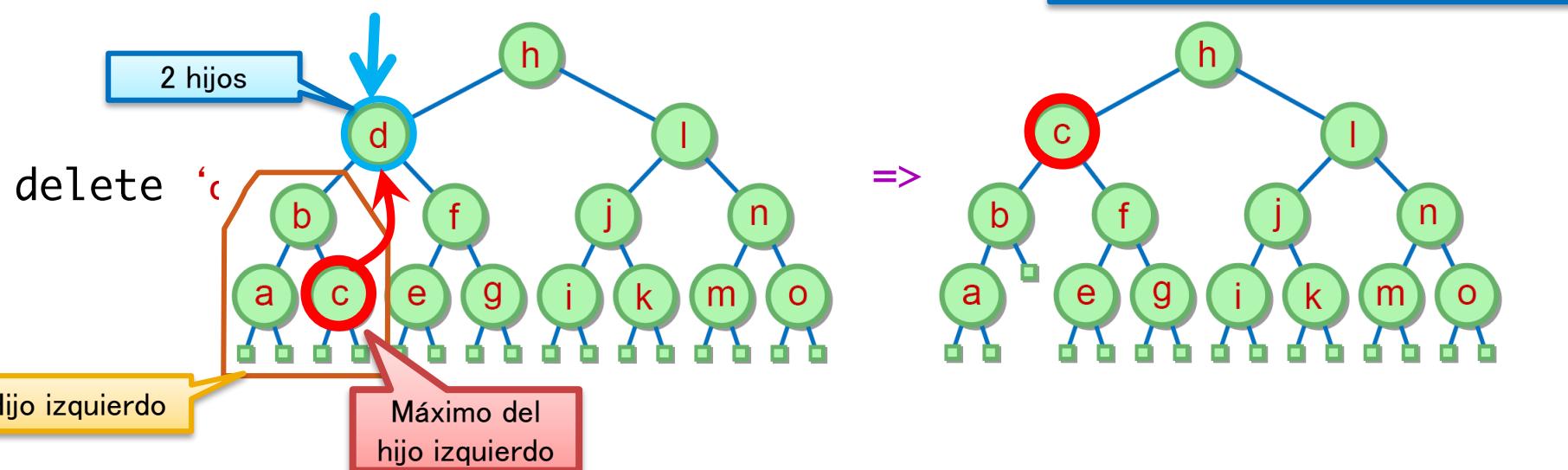
# Eliminación de un Elemento de un BST (III)

- En la primera fase localizamos el elemento siguiendo un algoritmo parecido al de inserción.
- (c) Si el nodo a borrar tiene **dos hijos**:
  - El mínimo elemento del hijo derecho sustituirá al elemento a borrar. El árbol resultante sigue siendo un BST



# Eliminación de un elemento de un BST (IV)

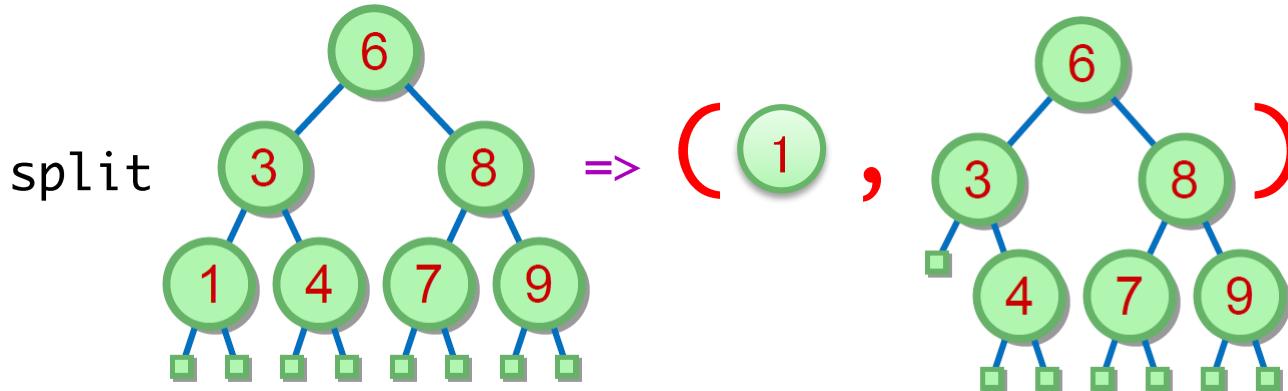
- En la primer fase localizamos el elemento siguiendo un algoritmo parecido al de inserción.
- (c) Si el nodo a borrar tiene **dos hijos**:
  - Alternativamente: El máximo elemento del hijo izquierdo sustituirá al elemento a borrar. El árbol resultante sigue siendo un BST



# Eliminación de un Elemento de un BST (V)

- Quitar y devolver el mínimo elemento de un árbol:

```
split :: BST a -> (a, BST a)
split (Node x Empty rt) = (x, rt)
split (Node x lt   rt) = (m, Node x lt' rt)
  where (m,lt') = split lt
```



# Eliminación de un Elemento de un BST (VI)

```
delete :: (Ord a) => a -> BST a -> BST a
delete x' Empty = Empty
delete x' (Node x lt rt)
| x' == x      = combine lt rt
| x' < x       = Node x (delete x' lt) rt
| otherwise     = Node x lt (delete x' rt)
```

```
combine :: BST a -> BST a -> BST a
combine Empty rt      = rt
combine lt Empty      = lt
combine lt rt          = Node x' lt rt'
where (x',rt') = split rt
```

Mínimo elemento del árbol derecho

Árbol derecho sin el elemento mínimo

Todas las claves de lt son menores que las de rt, luego x' es mayor que las claves de lt

# Plegados para BST

- `foldInOrder :: (a -> b -> b) -> b -> BST a -> b`
- `foldPreOrder :: (a -> b -> b) -> b -> BST a -> b`
- `foldPostOrder :: (a -> b -> b) -> b -> BST a -> b`

```
Main> let t = foldr insert empty [1,7,2,9,3,4]
Main> foldInOrder (+) 0 t
26
Main> foldInOrder (*) 1 t
1512
Main> foldInOrder (:) [] t
[1,2,3,4,7,9]
Main> foldPreOrder (:) [] t
[4,3,2,1,9,7]
```

# BST: Complejidad

Operación	Coste
empty	$O(1)$
isEmpty	$O(1)$
insert	$O(n)$
isElem	$O(n)$
delete	$O(n)$
minim	$O(n)$
maxim	$O(n)$

- (\*) Podemos obtener complejidades en  $O(\log n)$  si imponemos a los BST condiciones de balanceo; por ejemplo, AVL

# BST en Java. Interfaz

Será interesante disponer de una interfaz para distintas versiones de los BST (los puros y los equilibrados AVL)

```
package dataStructures.searchTree;  
  
import dataStructures.tuple.Tuple2;  
  
public interface SearchTree<K extends Comparable<? super K>, V> {  
  
    public boolean isEmpty();  
    public int size();  
    public int height();  
    public void insert(K k, V v);  
    public V search(K k);  
    public boolean isElem(K k);  
    public void delete(K k);  
    public Iterable<K> inOrder();  
    public Iterable<K> postOrder();  
    public Iterable<K> preOrder();  
    public Iterable<V> values();  
    public Iterable<Tuple2<K,V>> keysValues();  
}
```

Implementaremos una variante en la que cada nodo del árbol guarda, además de las referencias a los hijos, otros dos datos: una **clave** y un **valor**.

Los nodos en el árbol estarán **ordenados** según sus **claves**.

# BST en Java (II)

```
package dataStructures.searchTree;

public class BST<K extends Comparable<? super K>, V> implements SearchTree<K,V> {

    private static class Tree<C,D> {
        private C key;
        private D value;
        private Tree<C,D> left, right;

        public Tree(C k, D v) {
            key = k;
            value = v;
            left = null;
            right = null;
        }
    }

    private Tree<K,V> root;
    private int size;

    public BST() {
        root = null;
        size = 0;
    }

    public boolean isEmpty() {
        return root == null;
    }

    public int size () {
        return size;
    }
}
```

# BST en Java (III). Búsqueda

```
public V search(K key) {  
    return BST.searchRec(root, key);  
}
```

Devuelve el valor del árbol asociado a la clave **key**, o **null** si la clave no está en el árbol

```
private static <C extends Comparable<? Super C>, D>  
D searchRec(Tree<C,D> tree, C key) {  
    if (tree == null)  
        return null;  
    else if (key.compareTo(tree.key) == 0)  
        return tree.value;  
    else if (key.compareTo(tree.key) < 0)  
        return searchRec(tree.left, key);  
    else  
        return searchRec(tree.right, key);  
}
```

Recordemos que los árboles están ordenados según clave

```
public boolean isElem(K key) {  
    return search(key) != null;  
}
```

Devuelve **true** si el árbol contiene la clave **key**

# BST en Java (IV). Inserción

```
public void insert(K key, V value) {  
    root = insertRec(root, key, value);  
}  
  
// returns modified tree  
private Tree<K,V> insertRec(Tree<K,V> node, K key, V value) {  
    if (node == null) {  
        node = new Tree<>(key,value); // new node  
        size++;  
    }  
    else if (key.compareTo(node.key) == 0)  
        node.value = value; // key was already present  
    else if (key.compareTo(node.key) < 0)  
        node.left = insertRec(node.left, key, value);  
    else  
        node.right = insertRec(node.right, key, value);  
    return node;  
}
```

Inserta un nodo con clave **key** y valor **value** en el árbol. Si el nodo con key ya existía se reemplaza el valor con el nuevo

insert  $x'$  Empty = Node  $x'$  Empty Empty  
insert  $x'$  (Node  $x' \text{ lt rt}$ )  
|  $x'==x$  = Node  $x' \text{ lt rt}$   
|  $x'<x$  = Node  $x$  (insert  $x' \text{ lt}$ ) rt  
| otherwise = Node  $x \text{ lt }$  (insert  $x' \text{ rt}$ )

# BST en Java (IV). Eliminación(i)

```
// precondición: node no es un árbol vacío
// Elimina la clave mínima y su valor asociado del árbol primer
// argumento, dejando la clave mínima y su valor en el segundo
// argumento

private static <C, D> Tree<C,D>
    split(Tree<C,D> node, Tree<C,D> temp) {
    if (node.left == null) {
        // min encontrado: copia clave y valor al 2º arg. temp
        temp.key = node.key;
        temp.value = node.value;
        return node.right; // devuelve el hijo derecho
    } else {
        // elimina el mínimo de la subrama izquierda
        node.left = split(node.left, temp);
        return node;
    }
}
```

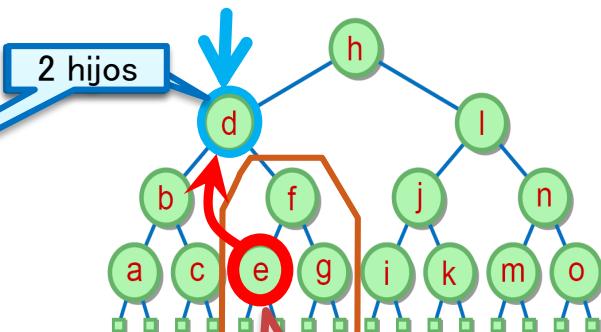
```
split (Node x Empty rt)  = (x,rt)
split (Node x lt   rt)  = (x',Node x lt' rt)
where (x',lt') = split lt
```

# BSTs en Java (V). Eliminación(ii)

```
public void delete(K key) {  
    root = deleteRec(root, key);  
}
```

Borra el nodo con clave  
key

```
// Devuelve el árbol modificado  
private Tree<K,V> deleteRec(Tree<K,V> node, K key) {  
    if (node == null)  
        ; // key no encontrada; no hacemos nada  
    else if (key.compareTo(node.key) == 0) {  
        if (node.left == null)  
            node = node.right;  
        else if (node.right == null)  
            node = node.left;  
        else // tiene dos hijos  
            node.right = split(node.right, node);  
            size--;  
    } else if (key.compareTo(node.key) < 0)  
        node.left = deleteRec(node.left, key);  
    else  
        node.right = deleteRec(node.right, key);  
    return node;  
}
```



Quitamos el mínimo de  
node.right, pero  
copiando antes su  
contenido en node

# Iteradores para BST

Recordemos las interfaces:

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
    void remove();  
}
```

```
public interface SearchTree<K extends Comparable<? super K>, V> {  
    public boolean isEmpty();  
    public int size();  
    public int height();  
    public void insert(K k, V v);  
    ...  
    public Iterable<K> preOrder();  
    public Iterable<K> postOrder();  
    public Iterable<K> inOrder();  
    public Iterable<V> values(); //itera valores inOrder  
    public Iterable<Tuple2<K,V>> keysValues(); // //itera pares inOr.  
}
```

Consideraremos cinco métodos para generar un objeto del tipo Iterable<\*> para recorrer (iterar) un BST, manipulando claves, valores, o ambos, utilizando los recorridos estándar:

- Por claves

- Preorden: Iterable<K> preOrder();
- PostOrden: Iterable<K> postOrder();
- InOrden: Iterable<K> inOrder();

- Por valores:

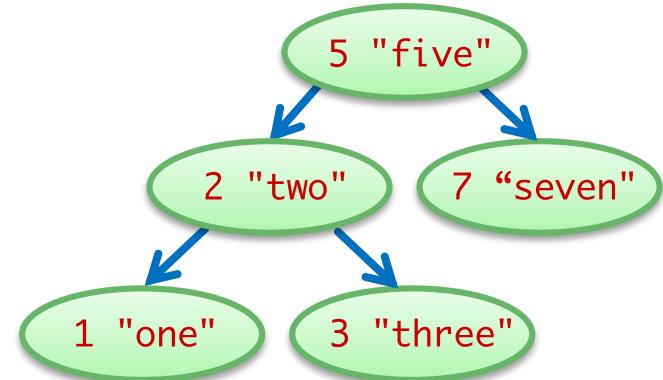
Iterable<V> values();

- Por pares (clave, valor): Iterable<Tuple2<K,V>> keysValues();

# Iteradores para BST. Ejemplo de Uso (I)

```
BST<Integer, String> bst = new BST<>();
```

```
bst.insert(5, "five");
bst.insert(2, "two");
bst.insert(7, "seven");
bst.insert(1, "one");
bst.insert(3, "three");
```



```
System.out.println("InOrder traversal:");
for(int i : bst.inOrder()) {
    System.out.printf("%d ",i);
}
System.out.println();
```

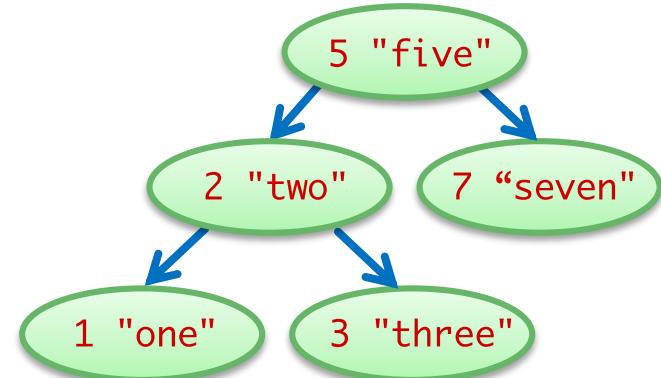
de tipo  
Iterable<Integer>

Imprime:  
1  
2  
3  
5  
7

# Iteradores para BST. Ejemplo de Uso (II)

```
BST<Integer, String> bst = new BST<>();
```

```
bst.insert(5, "five");
bst.insert(2, "two");
bst.insert(7, "seven");
bst.insert(1, "one");
bst.insert(3, "three");
```



```
System.out.println("InOrder traversal:");
Iterator<Integer> it = bst.inOrder().iterator();
while(it.hasNext())
    System.out.printf("%d ", it.next());
System.out.println();
```

```
System.out.println("KeysValues inOrder traversal:");
for(Tuple2<Integer, String> par : bst.keysValues()) {
    System.out.printf("%d %s", par._1(), par._2());
```

```
}
```

```
System.out.println();
```

Devuelve un iterador que recorre las claves del árbol en orden

Imprime:  
1  
2  
3  
5  
7

Imprime:  
1 one  
2 two  
3 three  
5 five  
7 seven

# Linked Sorted Sets vs Sets con BST

- Test experimental
- Medimos el tiempo de ejecución para realizar 50000 operaciones aleatorias (insert, delete, o isElem) sobre un conjunto inicialmente vacío.
- Usando un procesador Intel i7 860 CPU:
  - BST son 256 veces más rápidos que las Linked Sorted Set 😊

# Árboles Balanceados en Altura

- Si las operaciones de inserción y eliminación en un BST mantienen la altura pequeña, el árbol se dice **balanceado en altura**; tales operaciones pueden llegar a tener una complejidad óptima  $O(\log n)$ ; sin balanceo pueden elevarse hasta  $O(n)$

[http://en.wikipedia.org/wiki/Self-balancing\\_binary\\_search\\_tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)

- Existen varias aproximaciones a los árboles balanceados:
  - Árboles AVL
  - Árboles 2-3 y 2-3-4
  - Árboles Rojo-negros (Red-Black)

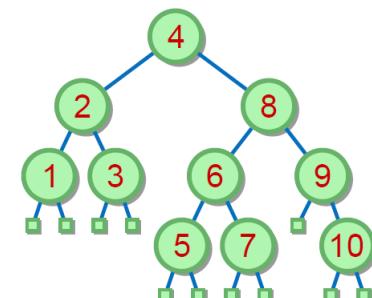
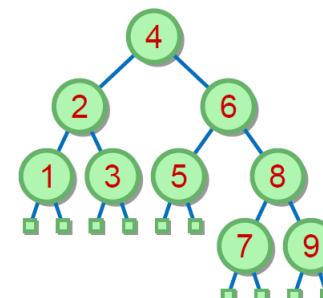
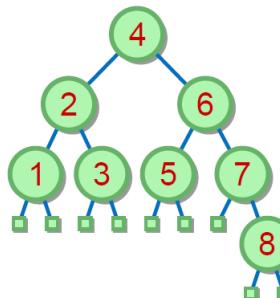
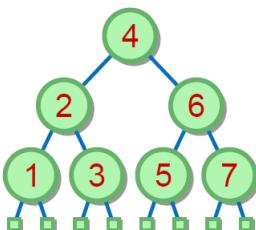
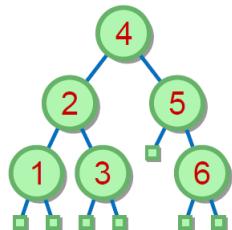
# Árboles AVL

[http://en.wikipedia.org/wiki/AVL\\_tree](http://en.wikipedia.org/wiki/AVL_tree)

- Llamados así por el nombre de sus creadores, **Adelson-Velskii** y **Landis**:



- "An algorithm for the organization of information" (1962)
- Los árboles AVL son árboles binarios de búsqueda que satisfacen la propiedad de **balanceado en altura (height-balance)**:  
**En cada nodo**, las alturas de sus hijos difieren a lo sumo en 1



Todos estos son árboles AVL

# Árboles AVL (II)

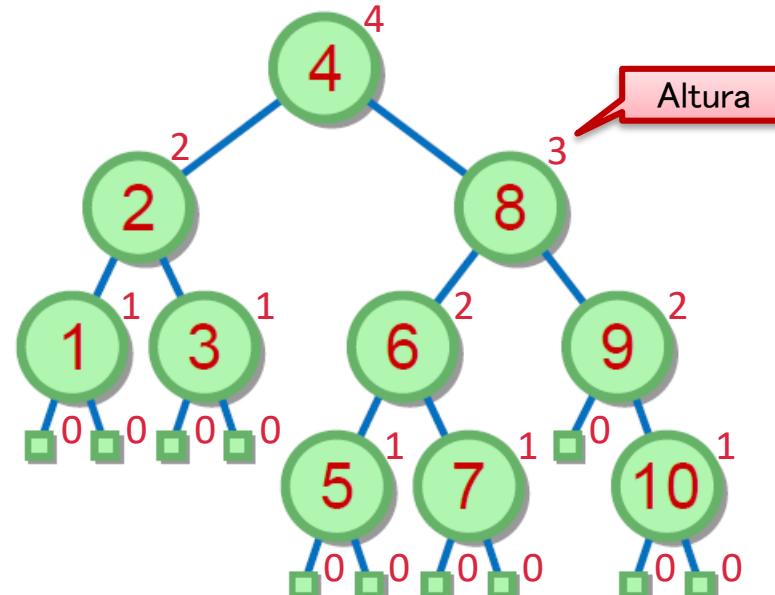
```
data AVL a = Empty | Node a Int (AVL a) (AVL a)
```

En cada nodo mantenemos su altura

```
height :: AVL a -> Int
```

```
height Empty = 0
```

```
height (Node k h lt rt) = h
```



```
isAVL :: (Ord a) => AVL a -> Bool
```

```
isAVL Empty = True
```

```
isAVL (Node k h lt rt) = forAll (<k>) lt && forAll (>k) rt  
&& abs (height lt - height rt) <= 1  
&& isAVL lt && isAVL rt
```

where

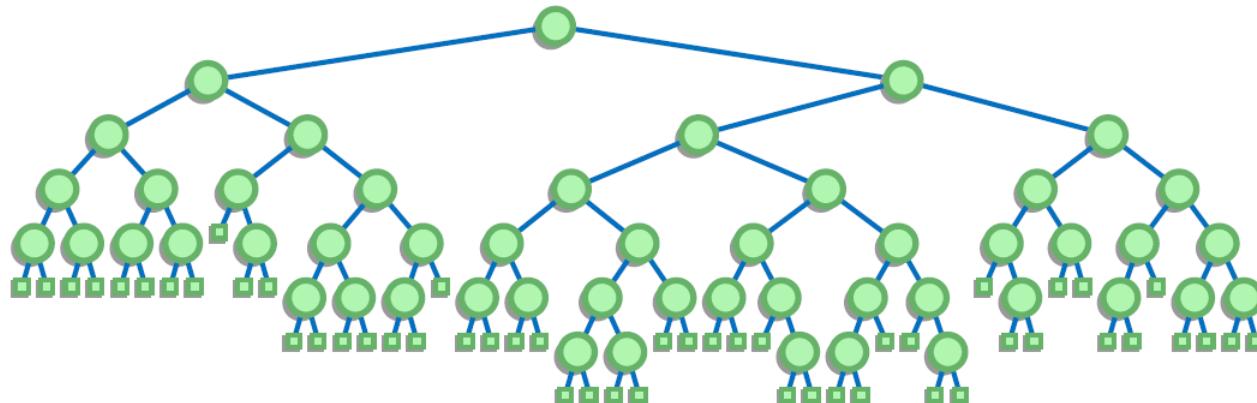
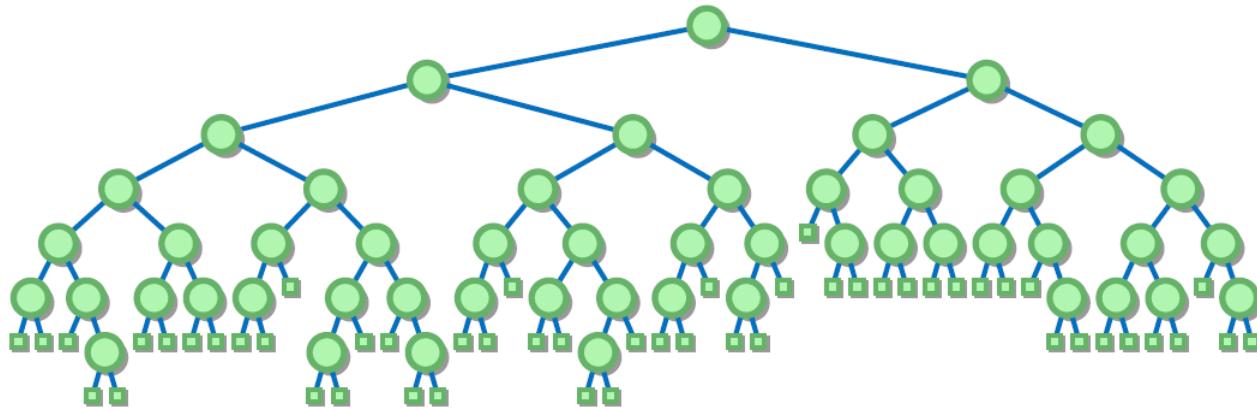
```
forAll :: (a -> Bool) -> AVL a -> Bool
```

```
forAll p Empty = True
```

```
forAll p (Node x h lt rt) = p x && forAll p lt && forAll p rt
```

# Árboles AVL (III)

- Dos árboles AVL con 50 elementos aleatorios:



# Altura de árboles AVL

- La altura  $h$  de un árbol AVL con  $n$  elementos está en  $O(\log_{\varphi} n)$
- Sea  $N(h)$  el mínimo número de nodos de un AVL de altura  $h$ :

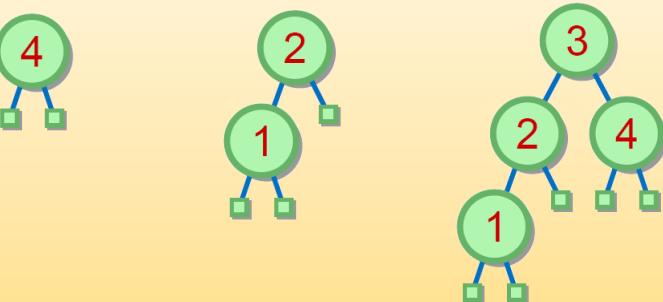
$$N(1) = 1$$

$$N(2) = 2$$

$$N(h) = 1 + N(h - 1) + N(h - 2), h > 2$$

Nodo de la raíz de un árbol de altura  $h$

La altura de un hijo debe ser  $h - 1$



Árboles AVL trees con alturas 1,2 y 3

La diferencia de las alturas de los hijos debe ser  $\leq 1$ , y el mínimo de nodos corresponde a  $h-2$

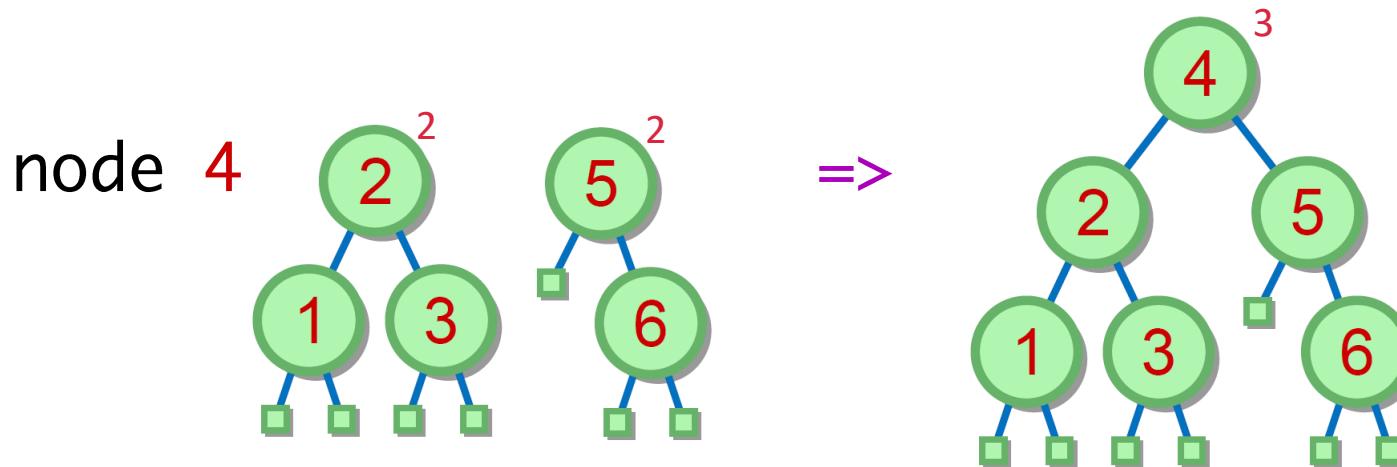
$$N(h) = O(\varphi^h) \text{ por lo que } h = O(\log_{\varphi} n)$$

$$\varphi = \frac{1+\sqrt{5}}{2}$$

# Constructor Auxiliar

- Construye un nuevo nodo a partir de dos árboles AVL y un valor, memorizando la altura del nuevo nodo

```
node :: a -> AVL a -> AVL a -> AVL a  
node k lt rt = Node k h lt rt  
where h = 1 + max (height lt) (height rt)
```



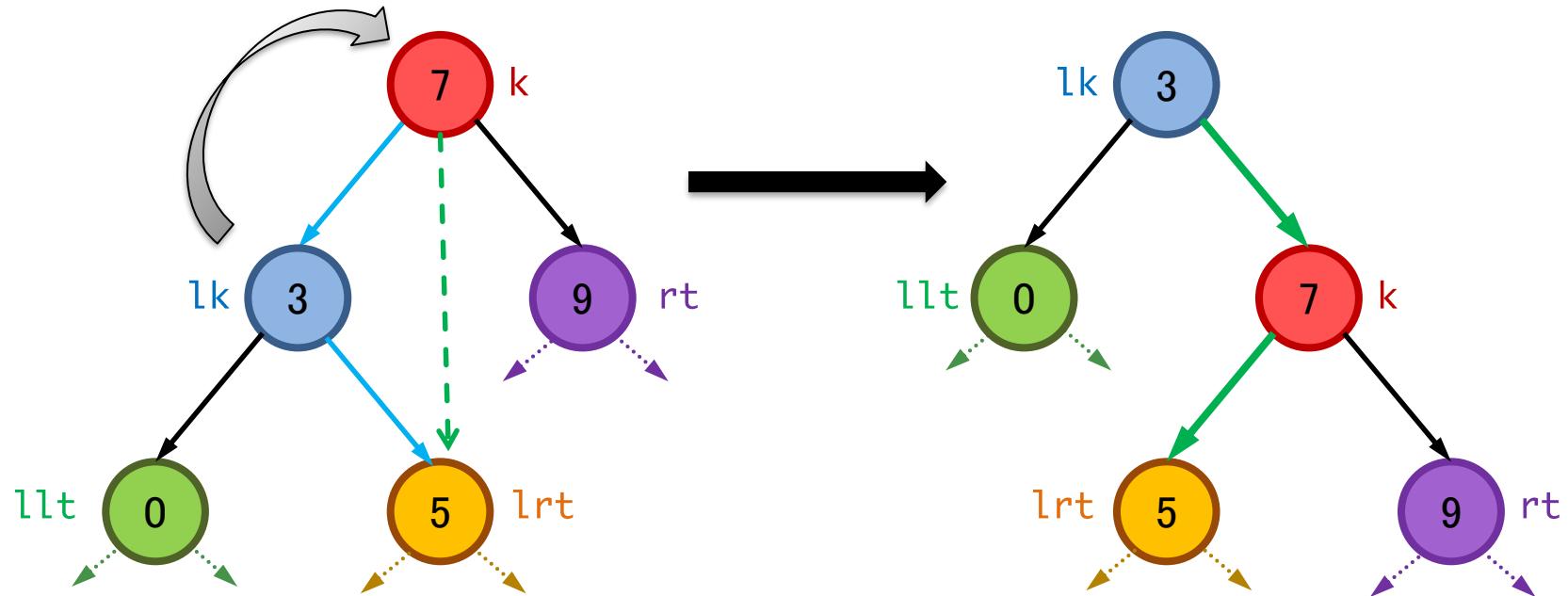
# Rotaciones en AVLs

- Una **rotación** es una operación que modifica la forma de un AVL sin violar la **propiedad de orden**
- Una rotación mueve un nodo arriba y otro abajo. Un subárbol se desconecta del nodo movido arriba y se conecta al nodo movido abajo
- Las rotaciones se usan para rebalancear un árbol binario, mejorando el rendimiento de las operaciones en el peor caso

# Rotación a la Derecha

rotR :: AVL a -> AVL a

rotR (Node k h (Node lk lh llt lrt) rt) = node lk llt (node k lrt rt)

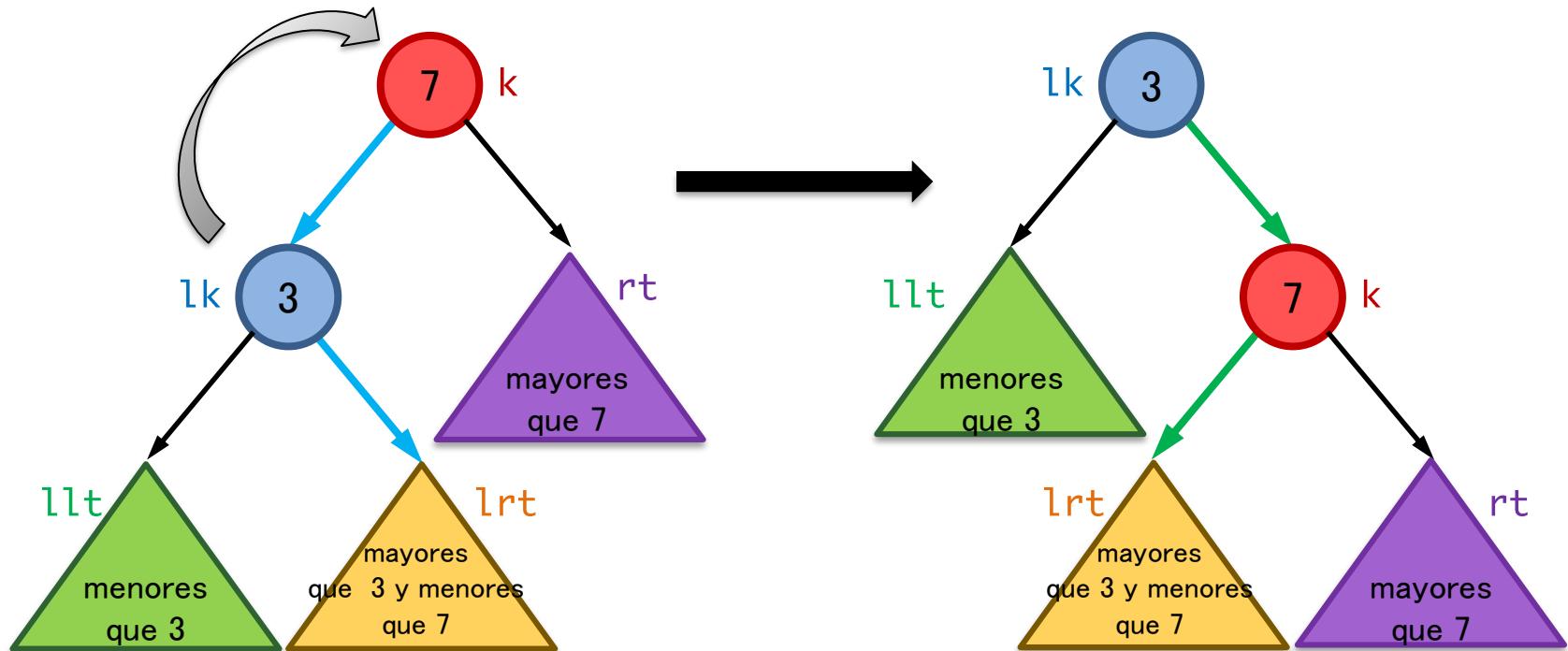


# Rotación a la Derecha (II)

rotR :: AVL a  $\rightarrow$  AVL a

rotR (Node k h (Node lk lh llt lrt) rt) = node lk llt (node k lrt rt)

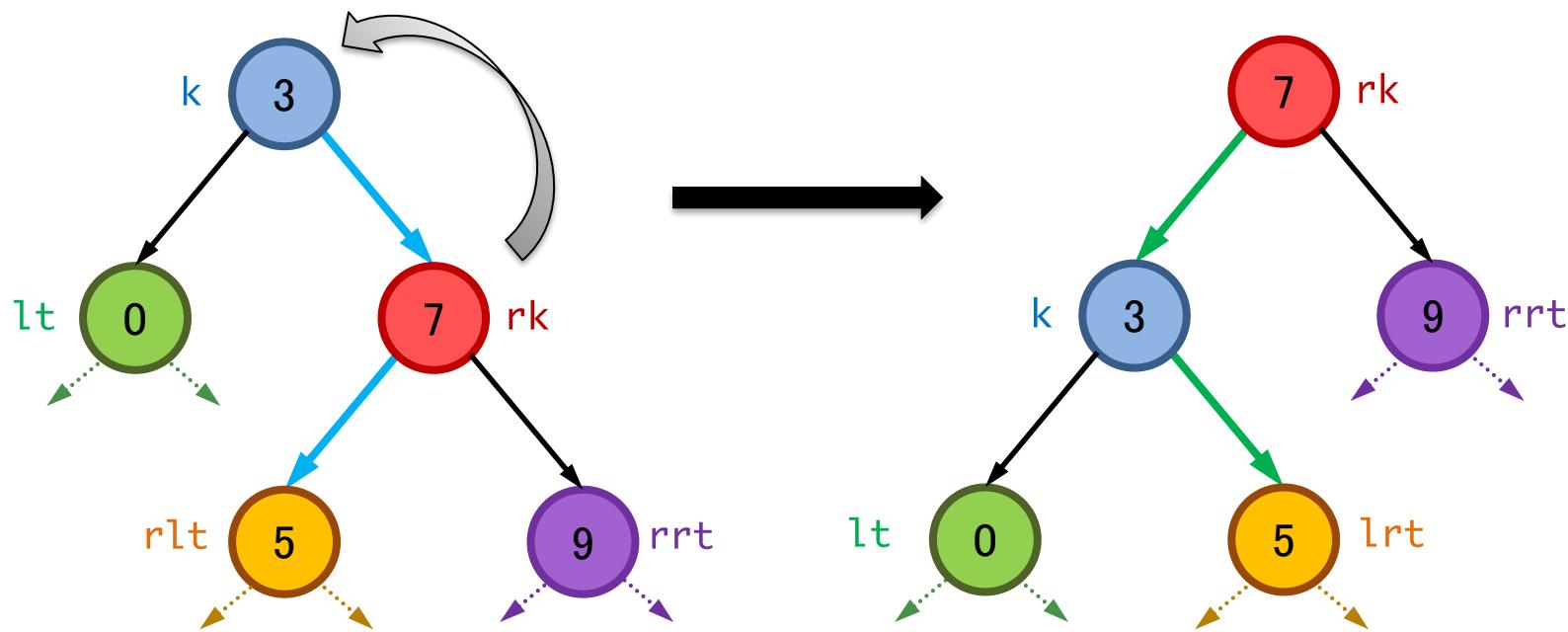
Mantiene invariante el orden



# Rotación a la izquierda

rotL :: AVL a -> AVL a

rotL (Node k h lt (Node rk rh rlt rrt)) = node rk (node k lt rlt) rrt

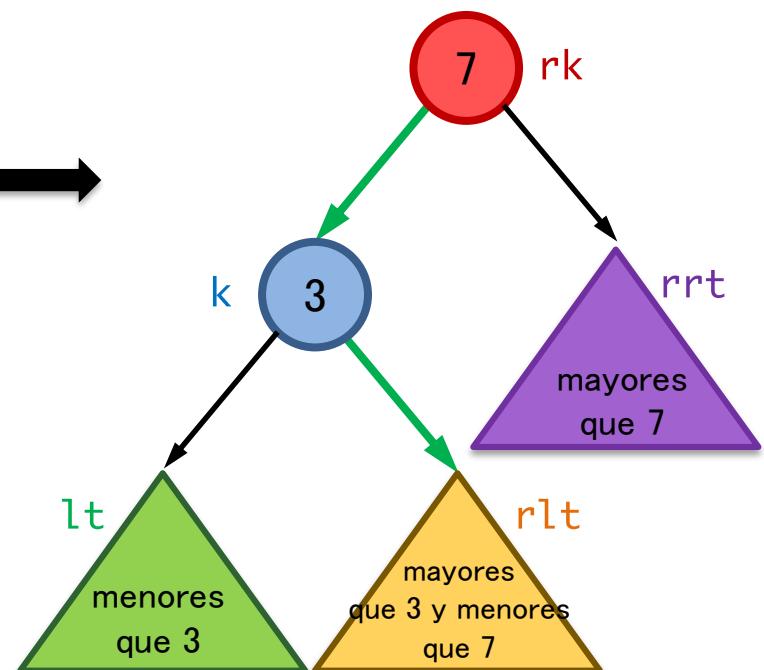
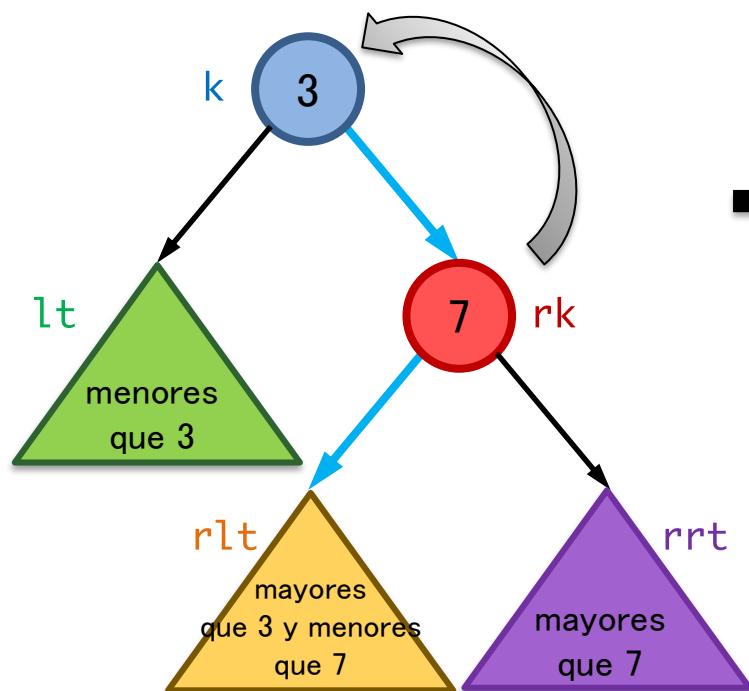


# Rotación a la izquierda (II)

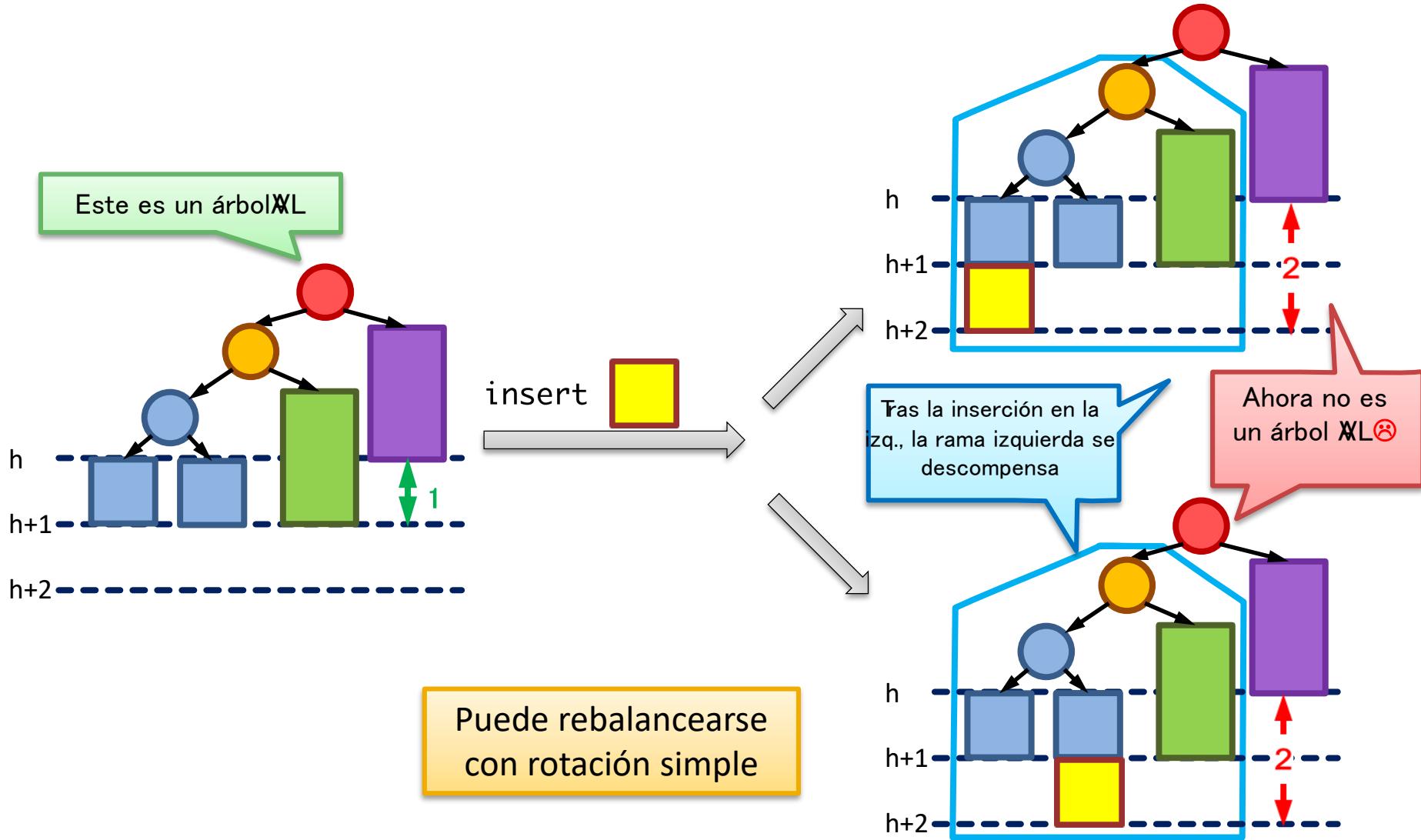
rotL :: AVL a  $\rightarrow$  AVL a

rotL (Node k h lt (Node rk rh rlt rrt)) = node rk (node k lt rlt) rrt

Mantiene invariante el orden

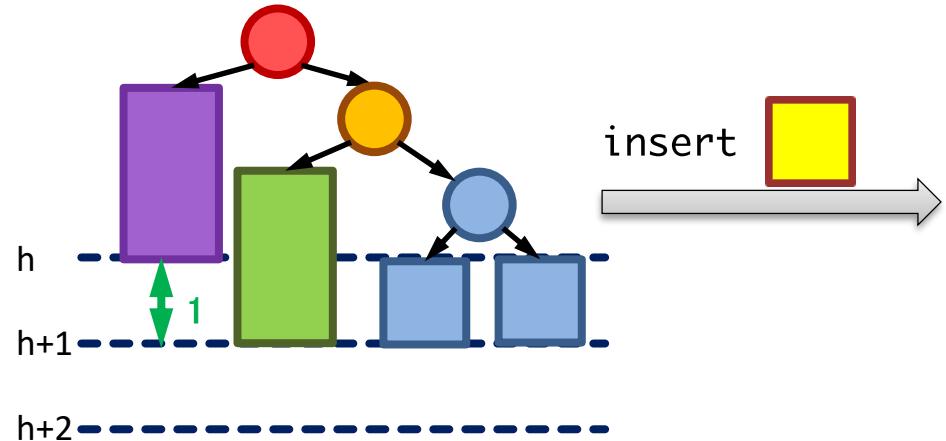


# Inserción y Descompensación

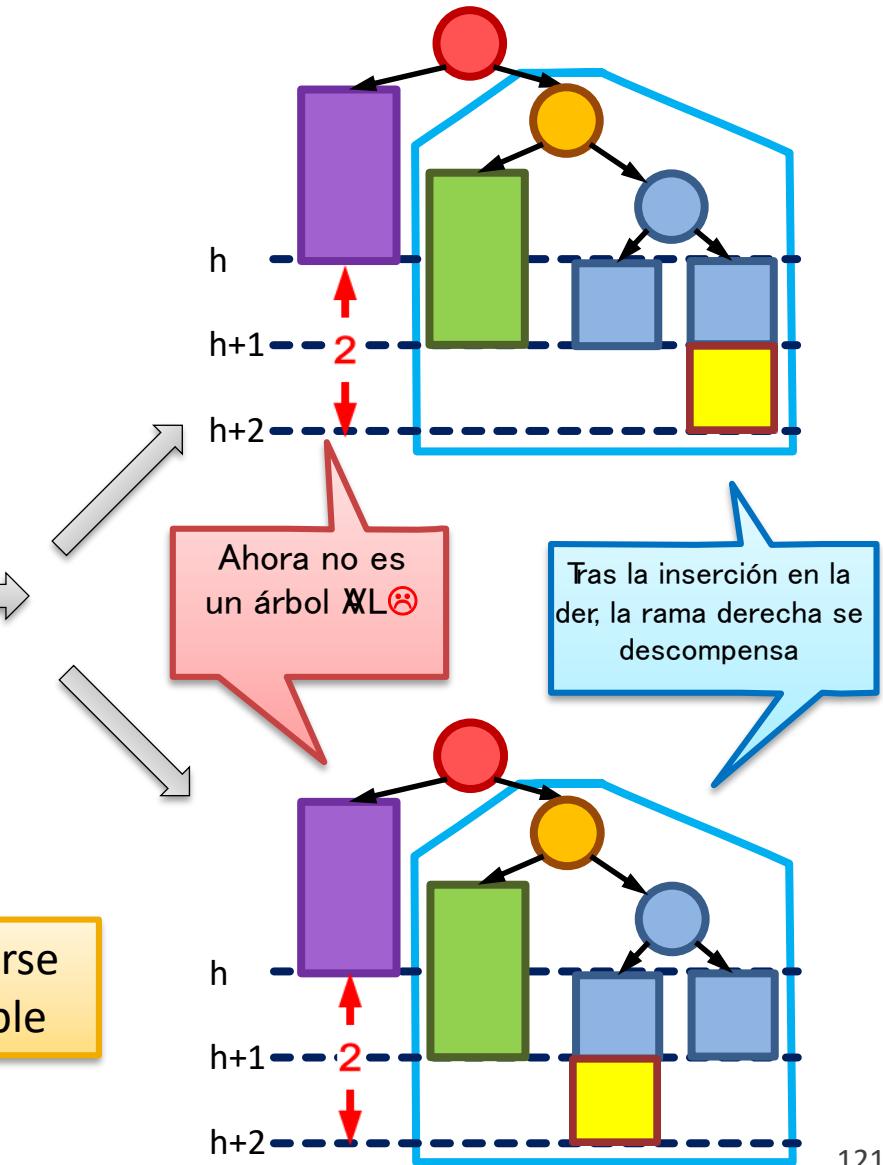


# Inserción y descompensación (II)

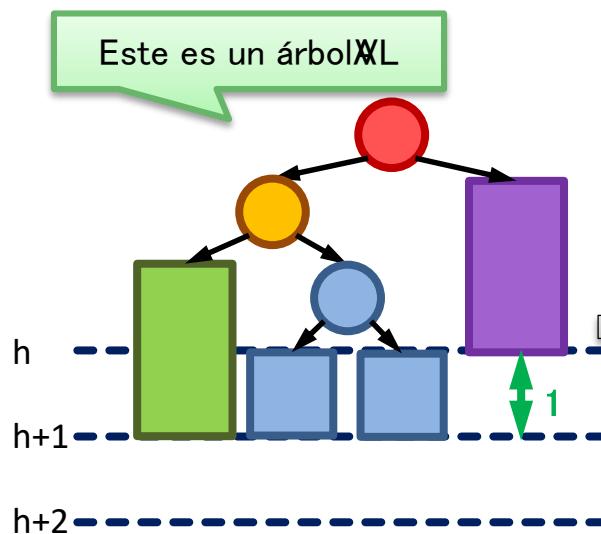
Este es un árbol AVL



Puede rebalancearse con rotación simple

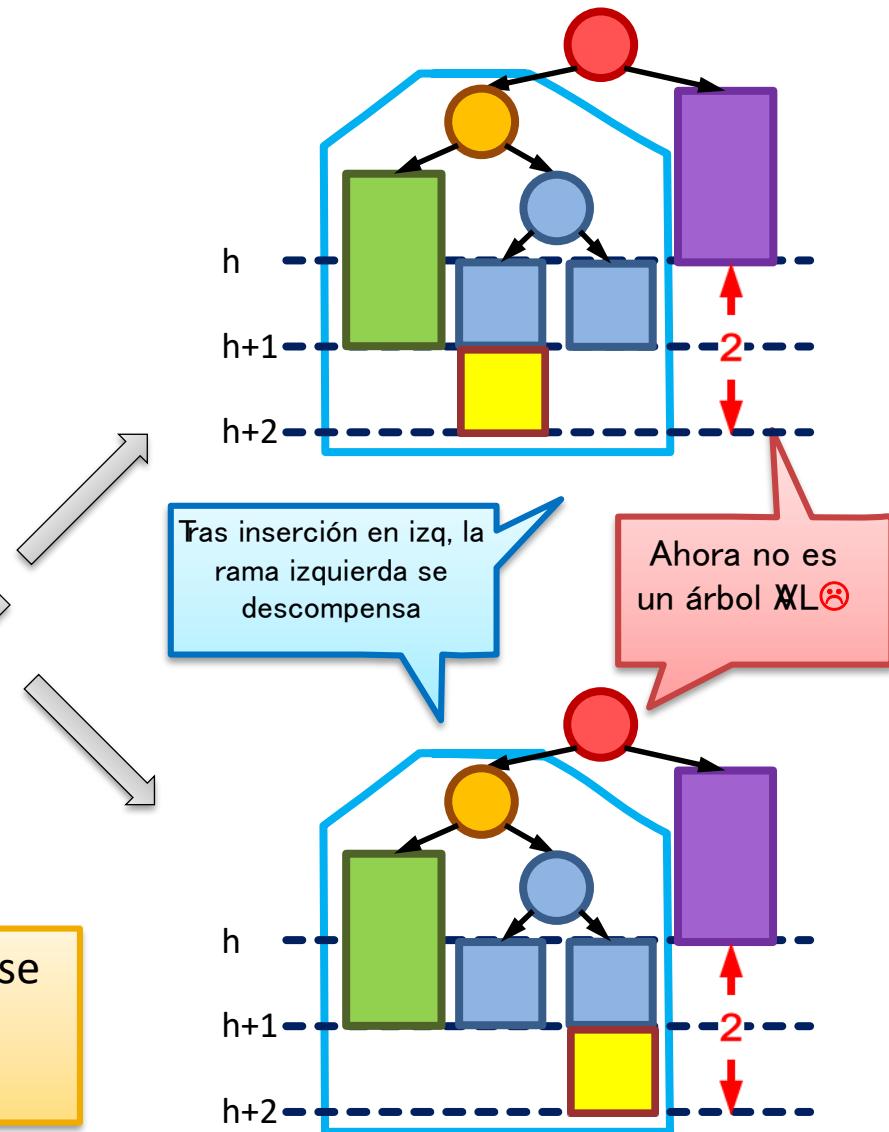


# Inserción y descompensación (III)



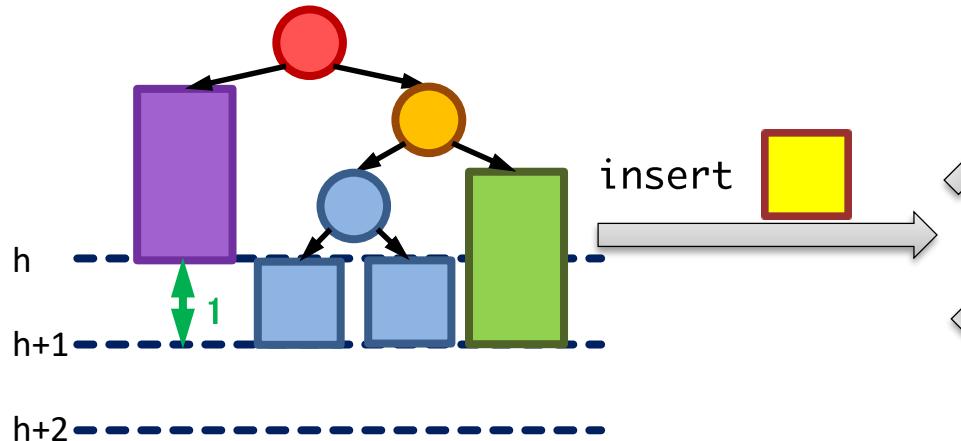
insert

Puede rebalancearse con una doble rotación

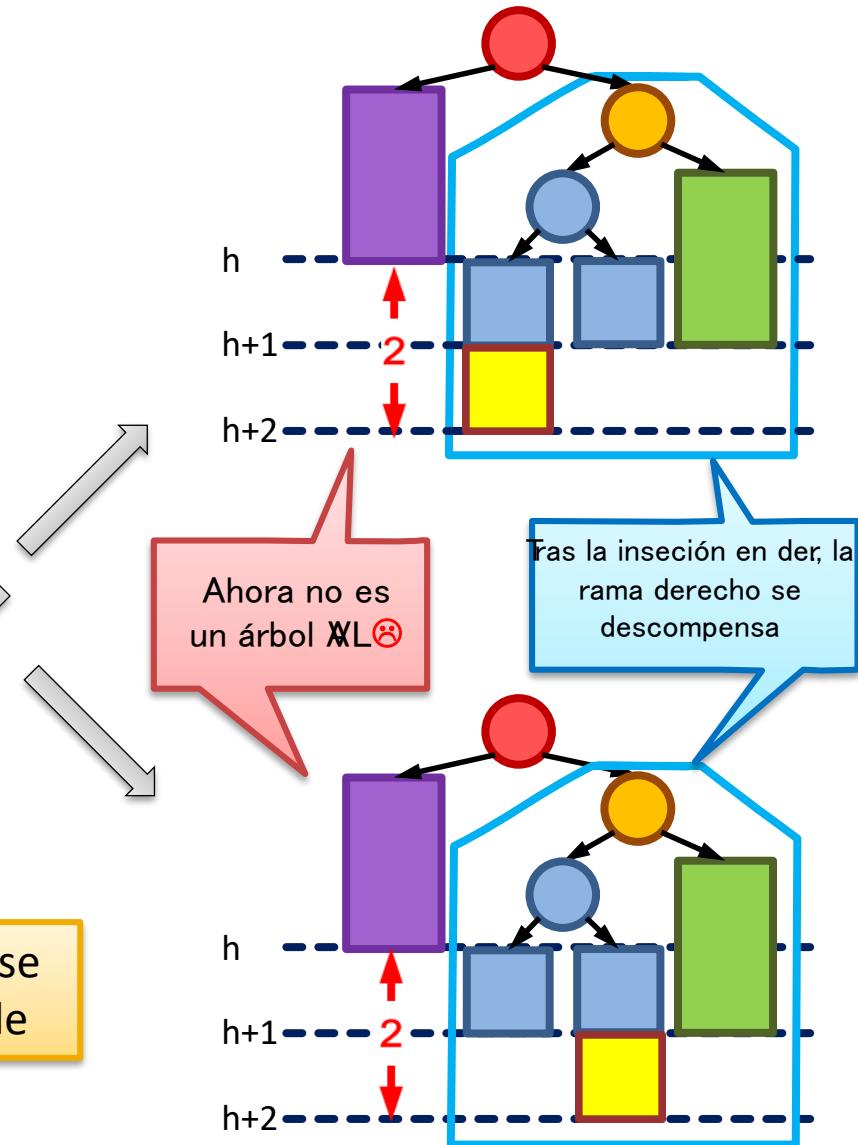


# Inserción y Descompensación (y IV)

Este es un árbol AVL



Puede rebalancearse con rotación simple



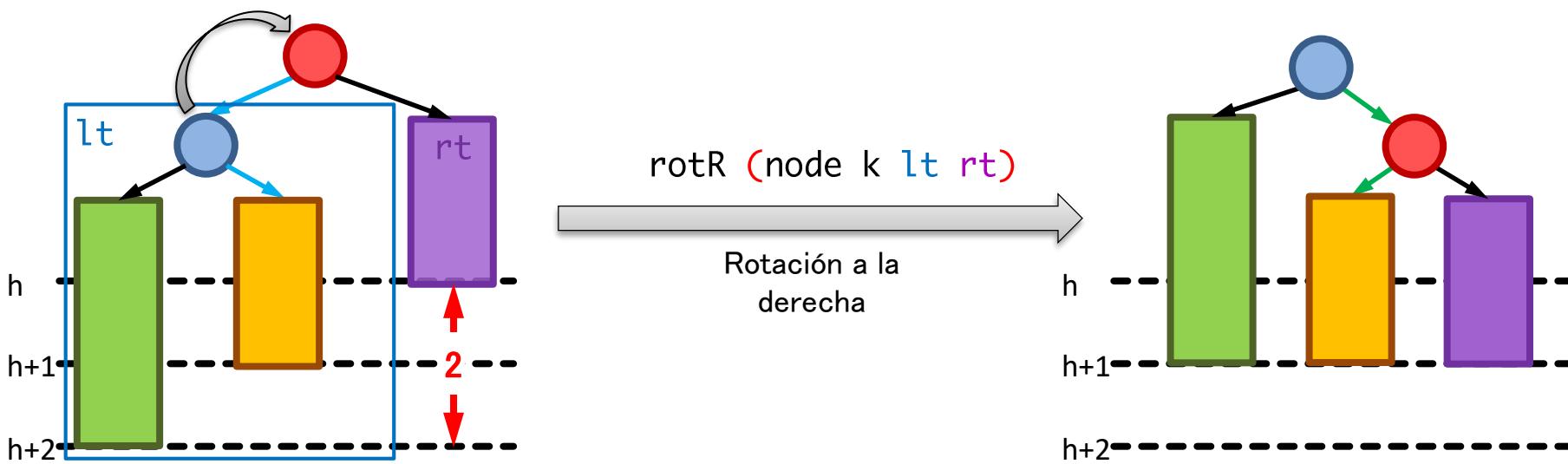
# Inclinación a Izquierda y Derecha

```
rightLeaning :: AVL a -> Bool  
rightLeaning (Node x h lt rt) = height lt < height rt
```

```
leftLeaning :: AVL a -> Bool  
leftLeaning (Node x h lt rt) = height lt > height rt
```

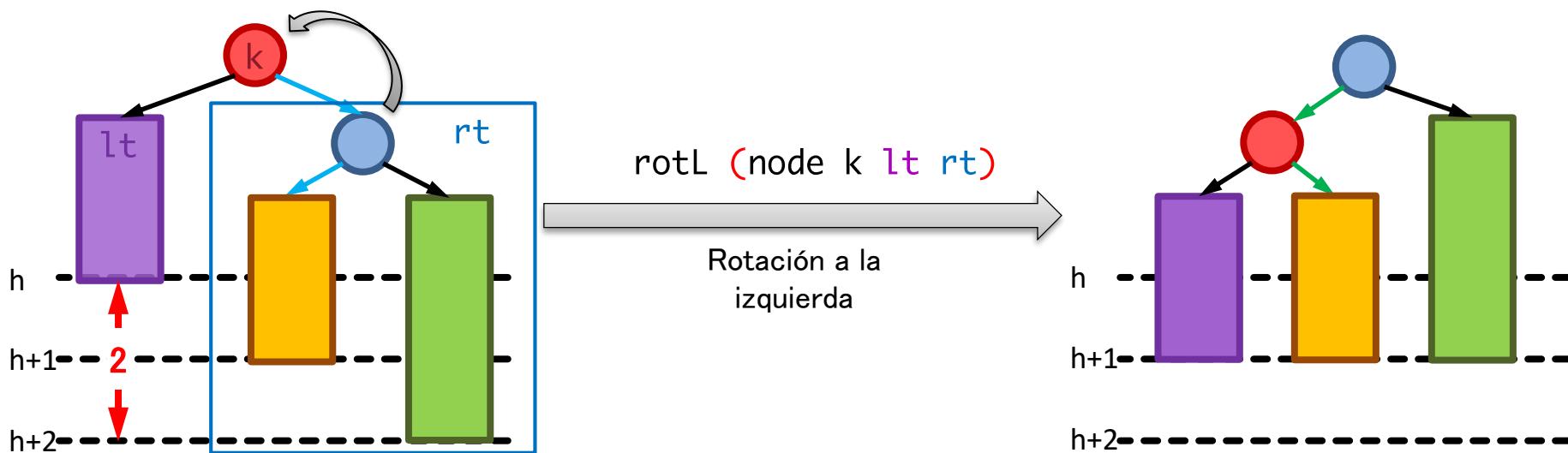
# Restaurando el balanceo (I)

```
balance :: a -> AVL a -> AVL a -> AVL a
balance k lt rt
| (lh-rh > 1) && leftLeaning lt = rotR (node k lt rt)
| (lh-rh > 1) = rotR (node k (rotL lt) rt)
| (rh-lh > 1) && rightLeaning rt = rotL (node k lt rt)
| (rh-lh > 1) = rotL (node k lt (rotR rt))
| otherwise
where lh = height lt
      rh = height rt
```



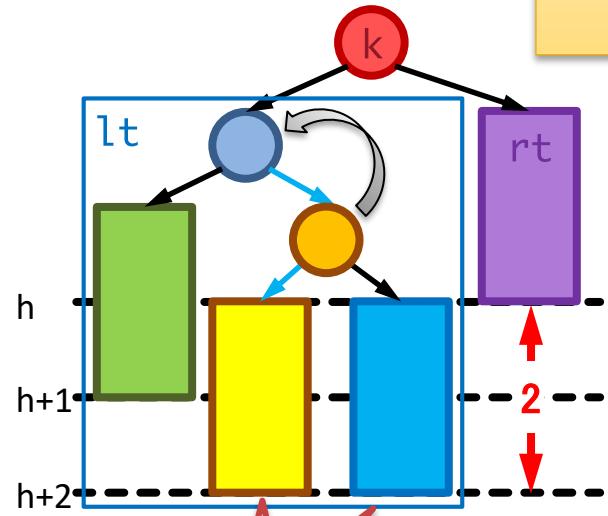
# Restaurando el Balanceo (II)

```
balance :: a -> AVL a -> AVL a -> AVL a
balance k lt rt
| (lh-rh > 1) && leftLeaning lt = rotR (node k lt rt)
| (lh-rh > 1)
| (rh-lh > 1) && rightLeaning rt = rotL (node k lt rt)
| (rh-lh > 1)
| otherwise
where lh = height lt
      rh = height rt
```

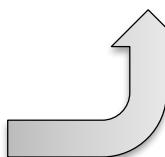
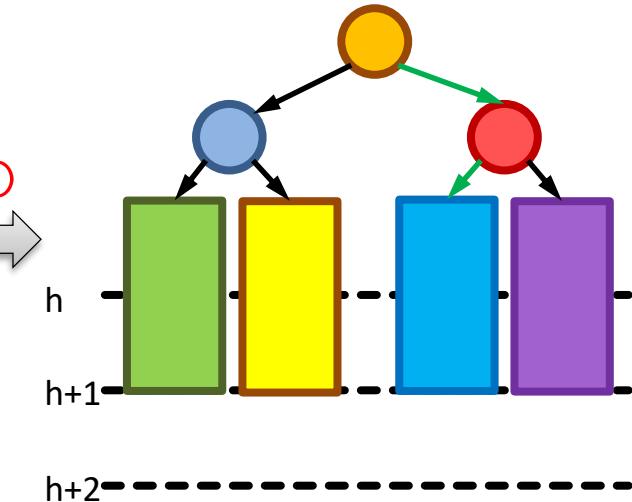
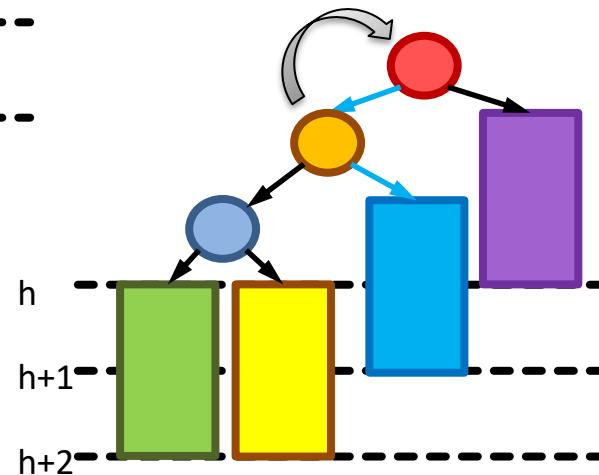


# Restaurando el Balanceo (III)

```
balance :: a -> AVL a -> AVL a -> AVL a
balance k lt rt
| (lh-rh > 1) && leftLeaning lt = rotR (node k lt rt)
| (lh-rh > 1) = rotR (node k (rotL lt) rt)
| (rh-lh > 1) && rightLeaning rt = rotL (node k lt rt)
| (rh-lh > 1) = rotL (node k lt (rotR rt))
| otherwise
where lh = height lt
      rh = height rt
```



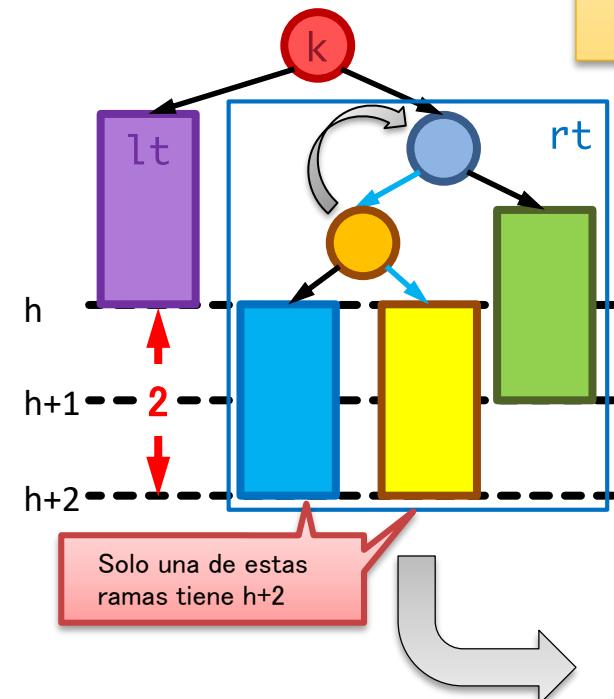
Rotación a la izquierda en el subárbol izquierdo (**lt**)



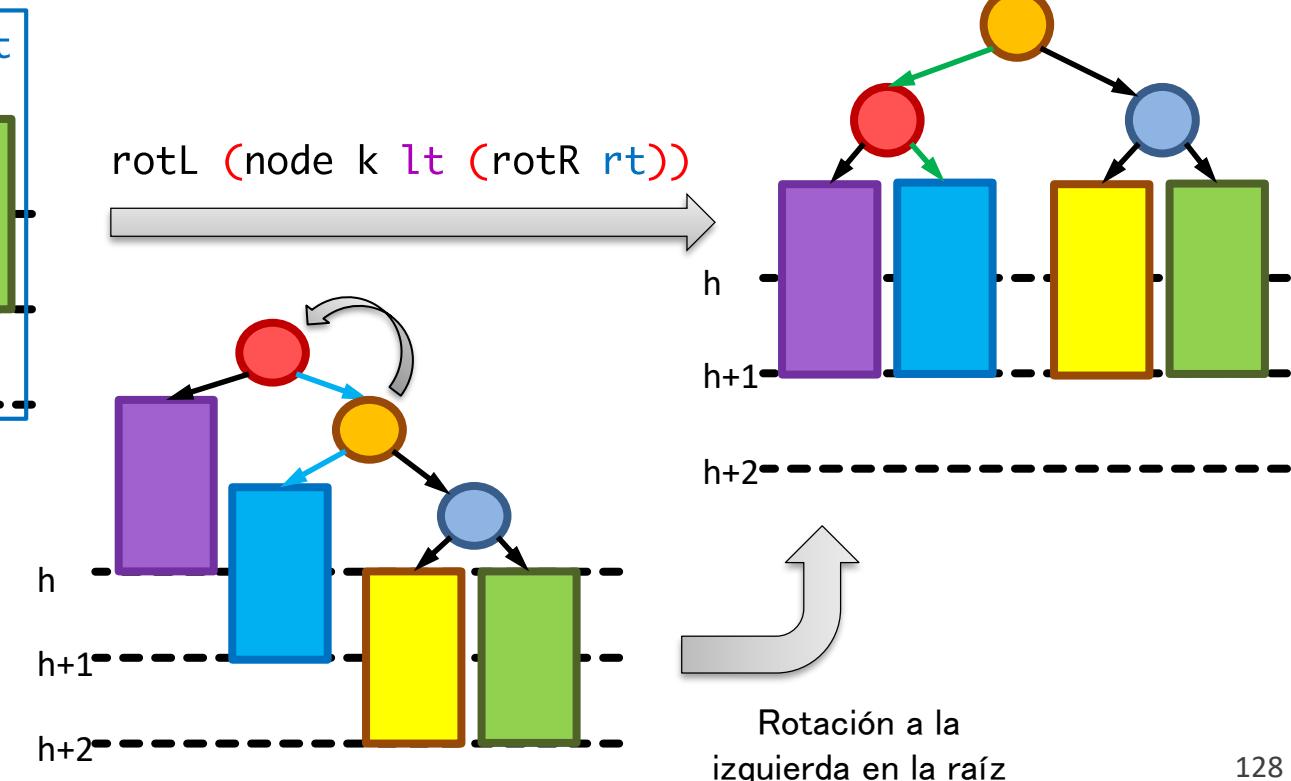
Rotación a la derecha en la raíz

# Restaurando el Balanceo (y IV)

```
balance :: a -> AVL a -> AVL a -> AVL a
balance k lt rt
| (lh-rh > 1) && leftLeaning lt = rotR (node k lt rt)
| (lh-rh > 1)
| (rh>lh > 1) && rightLeaning rt = rotL (node k lt rt)
| (rh-lh > 1)
| otherwise
where lh = height lt
      rh = height rt
```



Rotación a la derecha en el subárbol derecho (*rt*)



# Inserción en Árboles AVL. Código

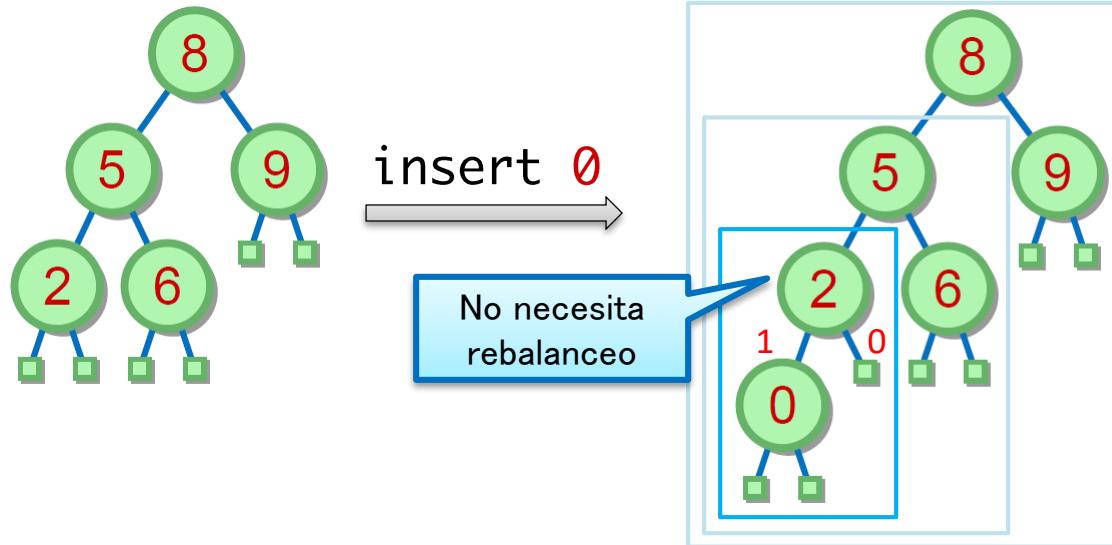
- Igual que la inserción en un Árbol Binario de Búsqueda pero restaurando el balanceo en todos los nodos modificados:

```
insert :: (Ord a) => a -> AVL a -> AVL a
insert k' Empty = node k' Empty Empty
insert k' (Node k h lt rt)
| k' == k      = Node k' h lt rt
| k' < k       = balance k (insert k' lt) rt
| otherwise     = balance k lt (insert k' rt)
```

Solo deben ser rebalanceados los nodos modificados en el camino desde la raíz hasta el punto de inserción

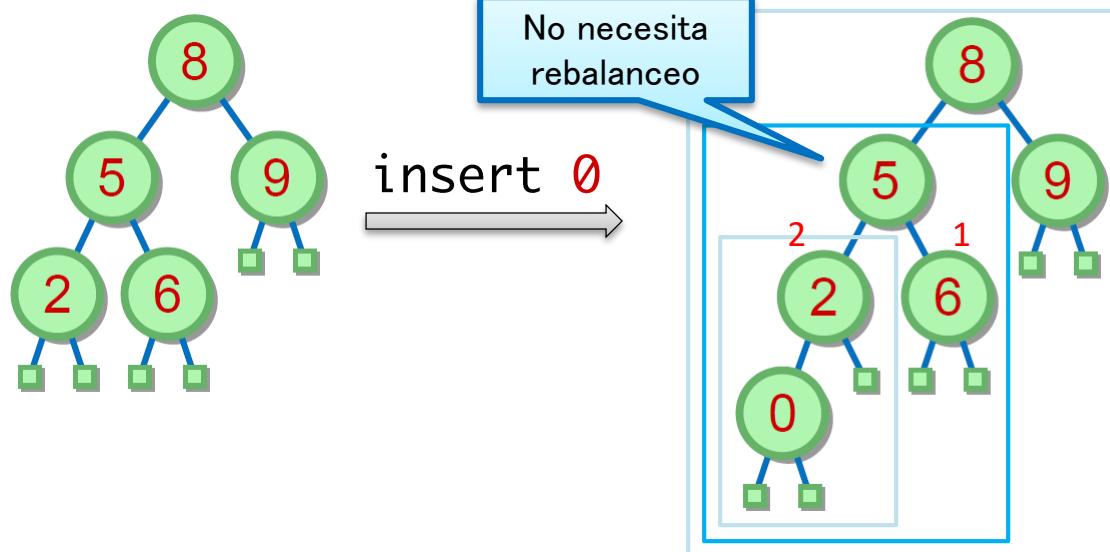
# Inserción en AVL. Ejemplo 1

- La inserción en un AVL tiene dos fases:
  - Inserción como en un BST
  - Rebalanceo de los nodos modificados:



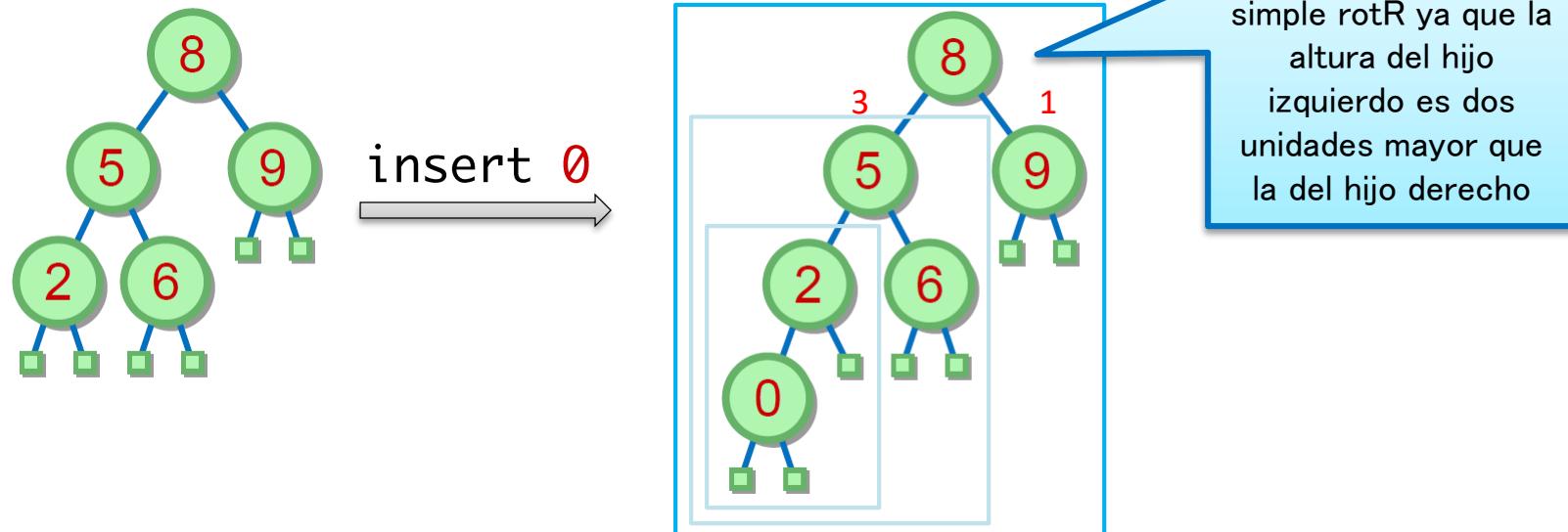
# Inserción en AVL. Ejemplo 1(II)

- La inserción en un AVL tiene dos fases:
  - Inserción como en un BST
  - Rebalanceo de los nodos modificados:



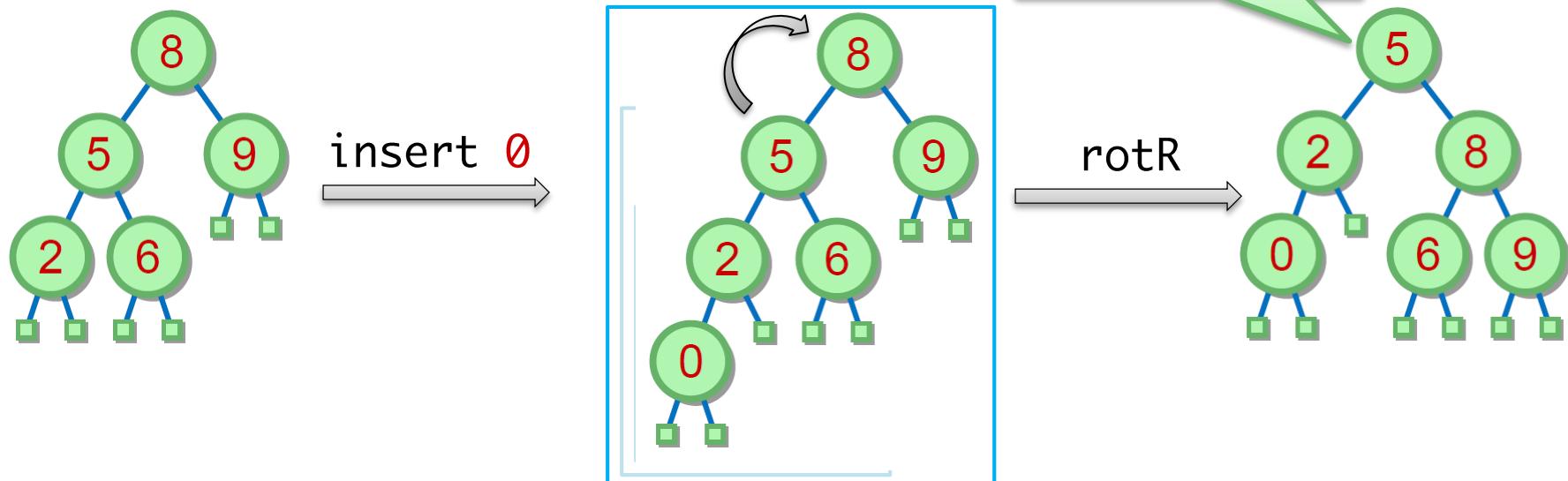
# Inserción en AVL. Ejemplo 1 (II)

- La inserción en un AVL tiene dos fases:
  - Inserción como en un BST
  - Rebalanceo de los nodos modificados:

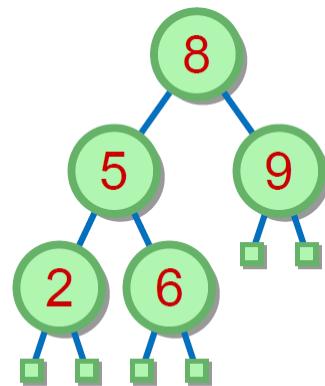


# Inserción en AVL. Ejemplo 1(III)

- La inserción en un AVL tiene dos fases:
  - Inserción como en un BST
  - Rebalanceo de los nodos modificados:

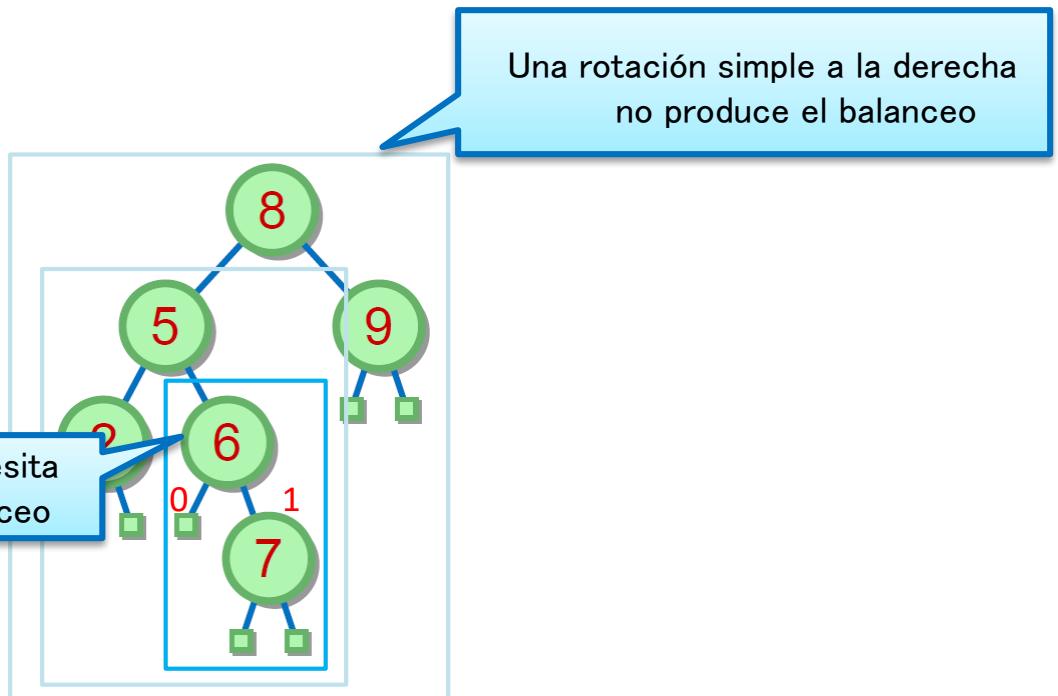


# Inserción en AVL. Ejemplo 2



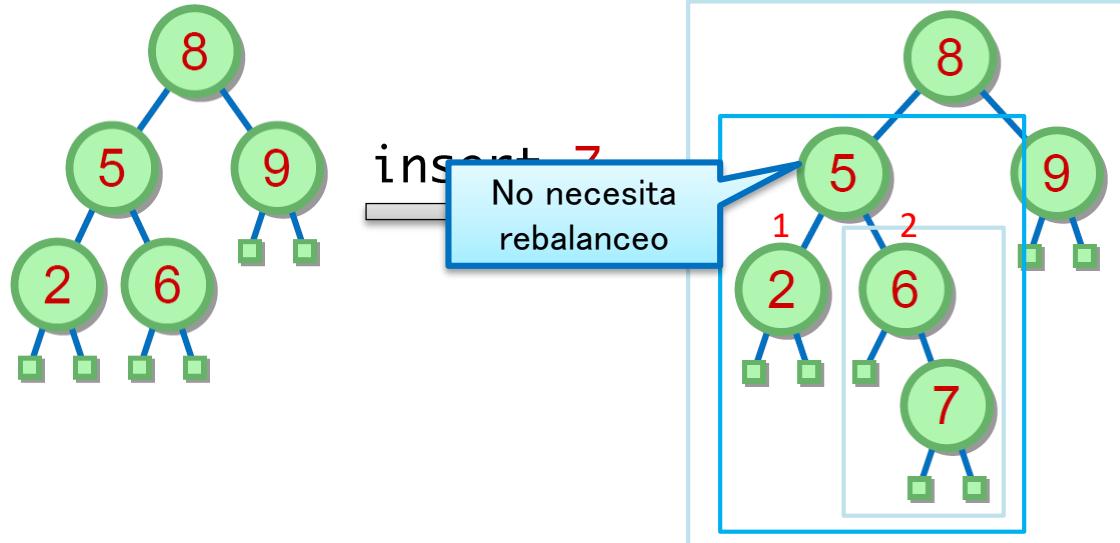
insert 7

No necesita rebalanceo

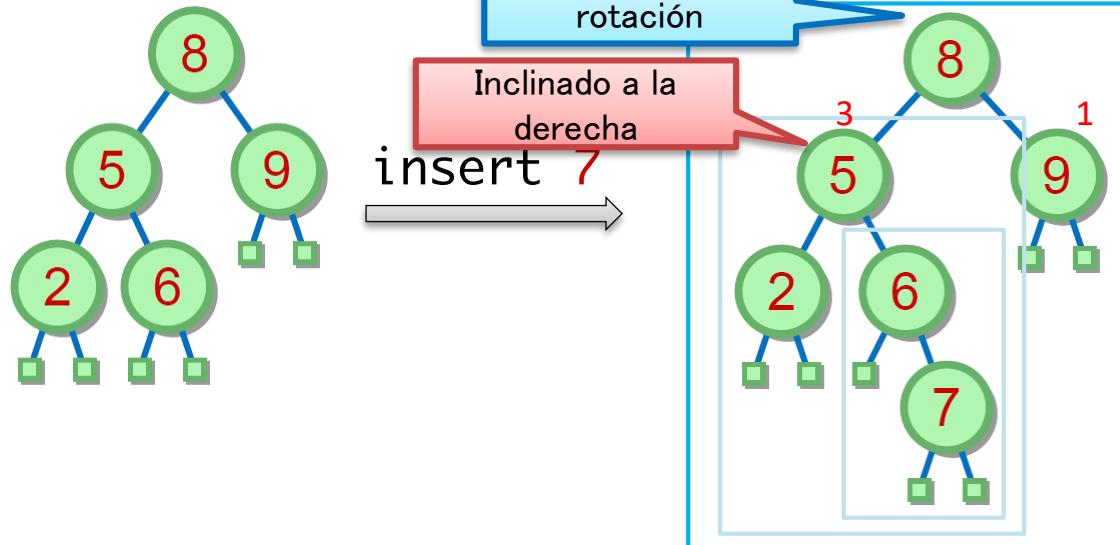


Una rotación simple a la derecha  
no produce el balanceo

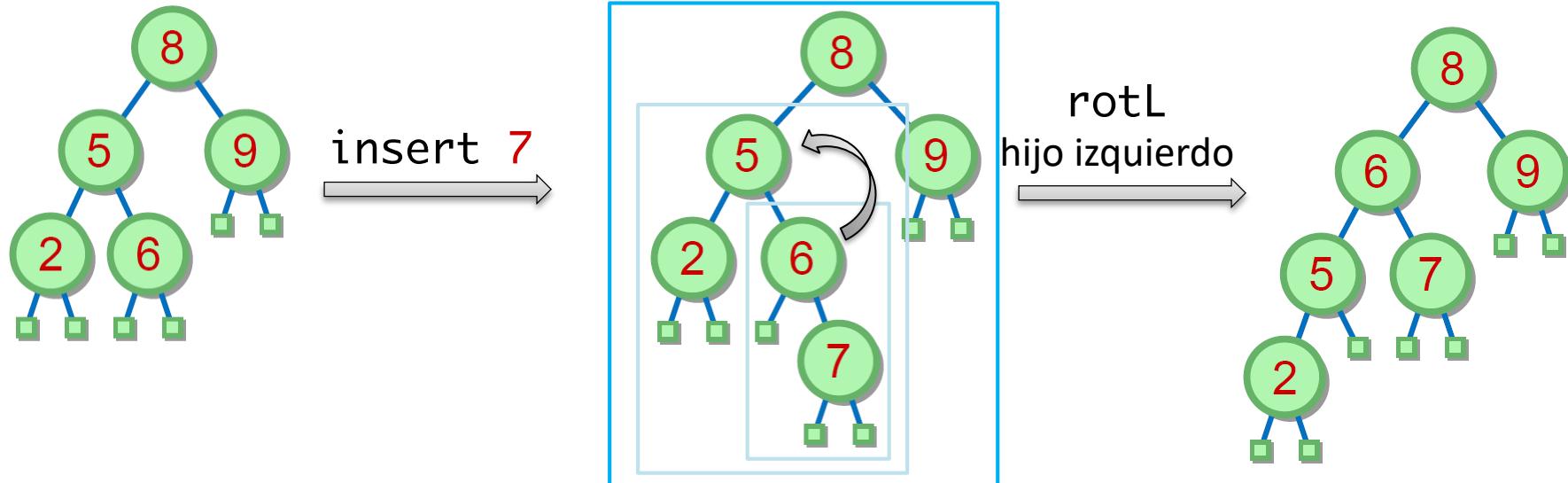
# Inserción en AVL. Ejemplo 2 (III)



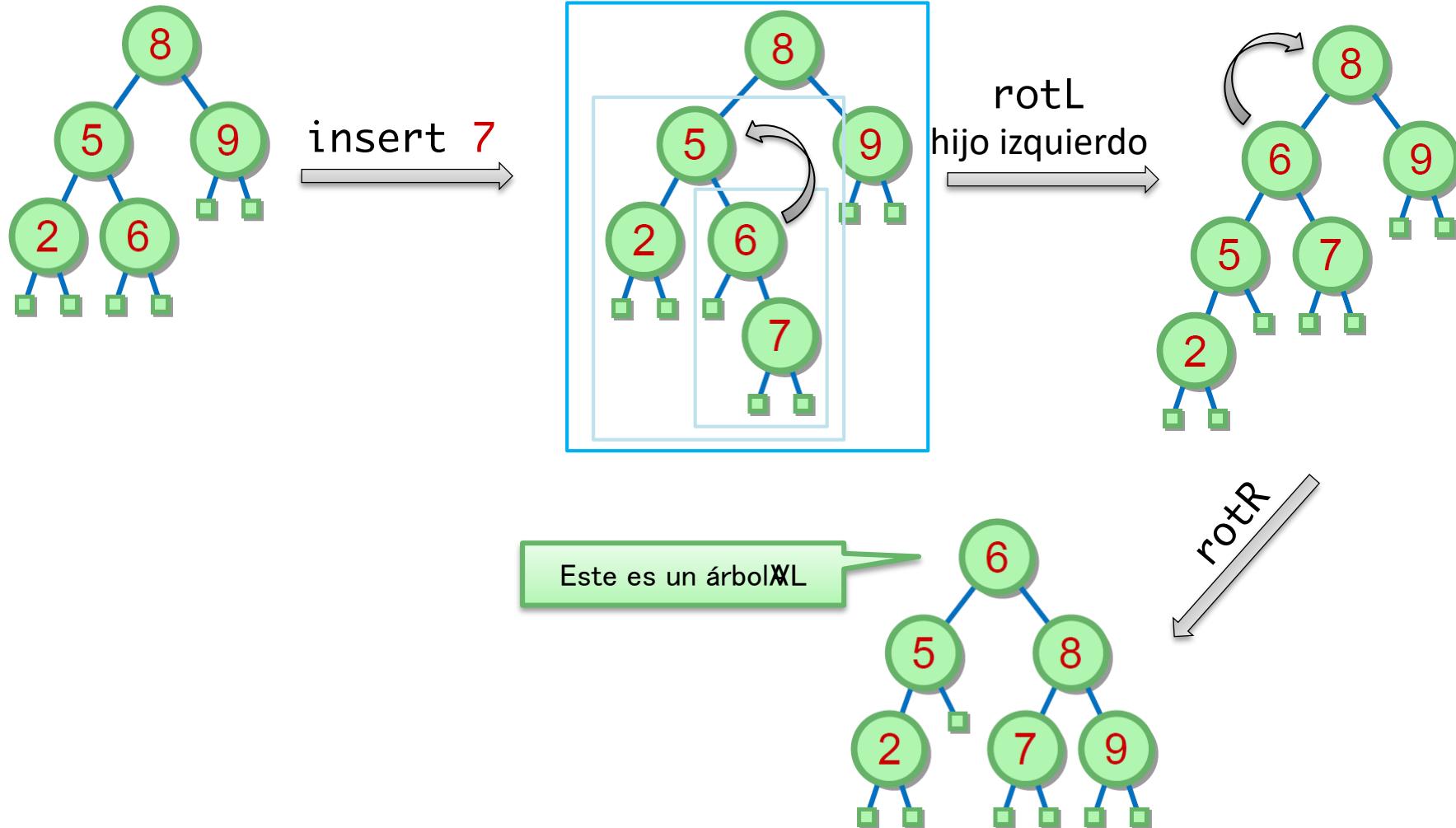
# Inserción en AVL. Ejemplo 2 (III)



# Inserción en AVL. Ejemplo 2 (IV)



# Inserción en AVL. Ejemplo 2 (V)



# Búsqueda en un Árbol AVL

- Igual que en Árboles Binarios de Búsqueda:

```
search :: (Ord a) => a -> AVL a -> Maybe a
```

```
search k' Empty = Nothing
```

```
search k' (Node k h lt rt)
```

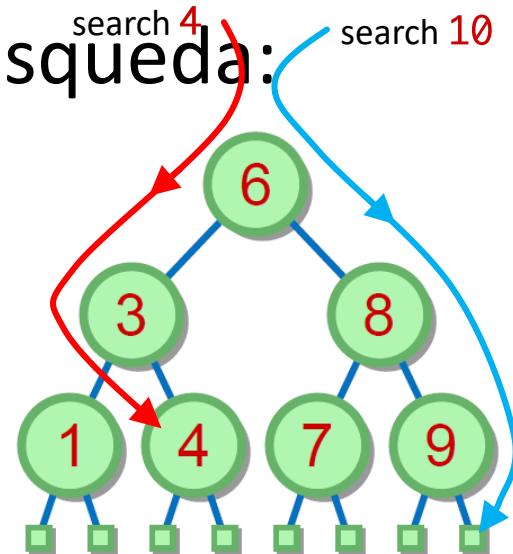
- | k' == k = Just k

- | k' < k = search k' lt

- | otherwise = search k' rt

```
isElem :: (Ord a) => a -> AVL a -> Bool
```

```
isElem k t = isJust (search k t)
```



```
data Maybe a = Nothing | Just a
```

Maybe isJust :: Maybe a -> Bool

isJust (Just \_) = True

isJust Nothing = False

# Eliminación en un árbol AVL

- Igual que en un Árbol Binario de Búsqueda, pero restaurando el balanceo en los nodos modificados:

```
delete :: (Ord a) => a -> AVL a -> AVL a
delete k' Empty = Empty
delete k' (Node k h lt rt)
| k' == k      = combine lt rt
| k' < k       = balance k (delete k' lt) rt
| otherwise     = balance k lt (delete k' rt)
```

```
combine :: AVL a -> AVL a -> AVL a
combine Empty rt      = rt
combine lt  Empty     = lt
combine lt  rt        = balance k' lt rt'
where (k',rt') = split rt
```

```
-- Elimina y devuelve el mínimo elemento de un árbol
split :: AVL a -> (a,AVL a)
split (Node k h Empty rt) = (k,rt)
split (Node k h lt  rt)  = (k',balance k lt' rt)
where (k',lt') = split lt
```

# Árboles AVL: Complejidad

Operación	Coste
empty	$O(1)$
isEmpty	$O(1)$
insert	$O(\log n)$
isElem	$O(\log n)$
delete	$O(\log n)$
minim	$O(\log n)$
maxim	$O(\log n)$

(\*) Compárense con los costes para un BST estándar

# Linked Sorted Sets vs Sets con BST vs Sets con AVL

- Test experimental: elementos insertados en orden aleatorio.
- Medimos el tiempo de ejecución para realizar 50000 operaciones aleatorias (insert, delete, o isElem) sobre un conjunto inicialmente vacío.
- Usando un procesador Intel i7 860 CPU:
  - BST son 256 veces más rápidos que las Linked Sorted Set 😊
  - AVL son 199 veces más rápidos que las Linked Sorted Set 😊
  - BST son 1.33 veces más rápidos que AVL con datos aleatorios

# Linked Sorted Sets vs Sets con BST vs Sets con AVL

- Test experimental: elementos insertados en orden ascendente (**árboles BST degenerados**).
- Medimos el tiempo de ejecución para realizar 50000 operaciones aleatorias (insert, delete, o isElem) sobre un conjunto inicialmente vacío.
- Usando un procesador Intel i7 860 CPU:
  - BST son 2.5 veces *más lentos* que las Linked Sorted Set 😞
  - AVL son 172 veces más rápidos que las Linked Sorted Set 😊
  - AVL son 444 veces más rápidos que BST con datos en orden

# Diccionarios: Signatura

Un diccionario permite manejar asociaciones entre un conjunto de claves  $a$  y un conjunto de valores  $b$ :

```
data Dict a b
```

-- un diccionario vacío

```
empty :: Dict a b
```

```
isEmpty :: Dict a b -> Bool
```

-- inserta/añade una asociación en un diccionario

```
insert :: (Eq a) => a -> b -> Dict a b -> Dict a b
```

-- recupera el valor en el diccionario

```
valueOf :: (Eq a) => a -> Dict a b -> Maybe b
```

El contexto para `insert` y `valueOf` puede ser `(Ord a) =>` si se usa el orden de las claves

# Diccionarios: Especificación

True  $\implies$  isEmpty empty

True  $\implies$  not (isEmpty (insert k v d))

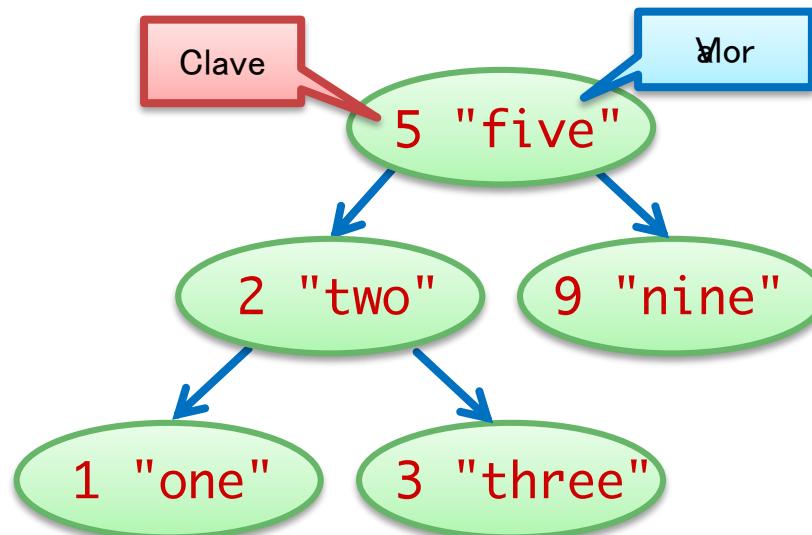
True  $\implies$  valueOf k empty == Nothing

k==k'  $\implies$  valueOf k (insert k' v' d) == Just v'

k $\neq$ k'  $\implies$  valueOf k (insert k' v' d) == valueOf k d

# Diccionarios: Implementación

- Un diccionario puede ser implementado eficientemente usando un árbol AVL con claves y valores en los nodos
- Los nodos deberían ordenarse según la clave



# Diccionarios: Implementación (II)

```
module DataStructures.Dictionary.AVLDictionary
( Dict
, empty
, isEmpty
, insert
, valueOf
) where
```

```
import qualified DataStructures.BinarySearchTree.AVL as T
```

```
data Rel a b = a :-> b
```

```
instance (Eq a) => Eq (Rel a b) where
  (k :-> _) == (k' :-> _) = (k == k')
```

```
instance (Ord a) => Ord (Rel a b) where
  (k :-> _) <= (k' :-> _) = (k <= k')
```

Dos asociaciones son iguales si tienen la misma clave

# Diccionarios: Implementación (III)

```
data Dict a b = D (T.AVL (Rel a b))
```

```
empty :: Dict a b  
empty = D T.empty
```

```
isEmpty :: Dict a b -> Bool  
isEmpty (D avl) = T.isEmpty avl
```

Un diccionario es vacío si lo es como árbol AVL

```
insert :: (Ord a) => a -> b -> Dict a b -> Dict a b  
insert k v (D avl) = D (T.insert (k :> v) avl)
```

```
valueOf :: (Ord a) => a -> Dict a b -> Maybe b  
valueOf k (D avl) =  
  case T.search (k :> undefined) avl of  
    Nothing          -> Nothing  
    Just (_ :> v)   -> Just v
```

# Diccionarios: Complejidad

- La complejidad de las operaciones en esta implementación son las correspondientes a un árbol AVL:

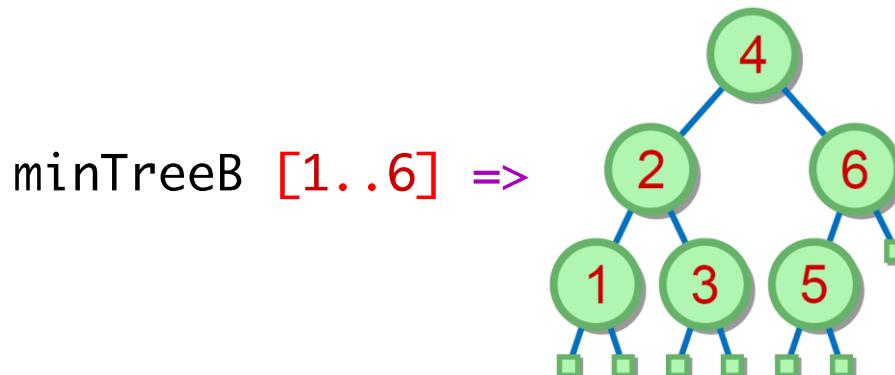
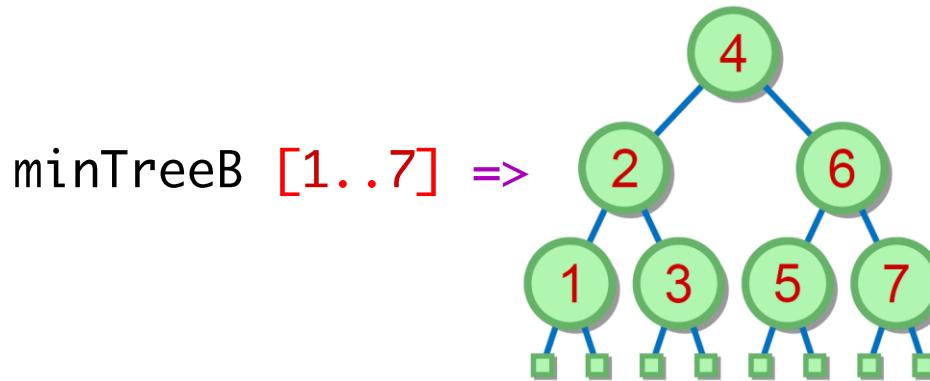
Operación	Coste
empty	$O(1)$
isEmpty	$O(1)$
insert	$O(\log n)$
valueOf	$O(\log n)$

# Material Complementario

Para profundizar

# Construcción de un árbol binario de altura mínima

- Sea  $xs$  una lista
- Pretendemos construir un árbol binario  $t$  de altura mínima, verificando además  $\text{inOrderB } t = xs$



# Construcción de un árbol binario de altura mínima (II)

```
minTreeB :: [a] -> TreeB a  
minTreeB [] = EmptyB  
minTreeB xs = NodeB z (minTreeB yz) (minTreeB zs)  
where
```

```
m = length xs `div` 2  
(yz,z:zs) = splitAt m xs
```

```
p_minTreeB xs = n>0 ==> inOrderB t == xs  
                           && heightB t == 1 + log2 n
```

where

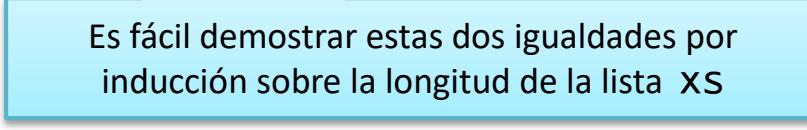
```
t = minTreeB xs  
n = length xs
```

```
log2 :: Int -> Int  
log2 x = truncate (logBase 2 (fromIntegral x))
```

```
Prelude> splitAt 2 [10..15]  
([10,11],[12,13,14,15])
```

```
Prelude> splitAt 3 [10..15]  
([10,11,12],[13,14,15])
```

```
Prelude> quickCheck p_minTreeB  
+++ OK, passed 100 tests.
```



Es fácil demostrar estas dos igualdades por inducción sobre la longitud de la lista `xs`

# Construcción de un árbol binario de altura mínima (III)

```
minTreeB :: [a] -> TreeB a
minTreeB [] = EmptyB
minTreeB xs = NodeB z (minTreeB ys) (minTreeB zs)
where
  m = length xs `div` 2
  (ys,z:zs) = splitAt m xs
```

- Sea  $T(n)$  el número de pasos para evaluar `minTreeB` sobre una lista con  $n$  elementos:
  - $T(0) = 1$
  - $T(n) = O(n) + 2 T(n/2)$ , si  $n > 0$
- La solución  $T(n)$  está en  $O(n \cdot \log n)$  ☹

splitAt es  $O(n)$

# Construcción de un árbol binario de altura mínima (y IV)

```
minTreeB' :: [a] -> TreeB a  
minTreeB' xs = fst (aux (length xs) xs)
```

```
aux :: Int -> [a] -> (TreeB a, [a])
```

```
aux 0 xs = (EmptyB, xs)
```

```
aux 1 xs = (NodeB (head xs) EmptyB EmptyB, tail xs)
```

```
aux n xs = (NodeB y t1 t2, zs)
```

where

```
m = div n 2
```

```
(t1, y:ys) = aux m xs
```

```
(t2, zs) = aux (n-m-1) ys
```

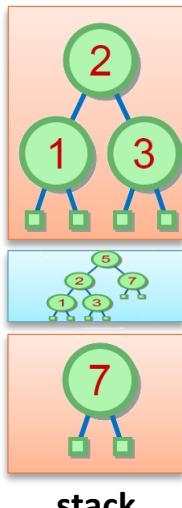
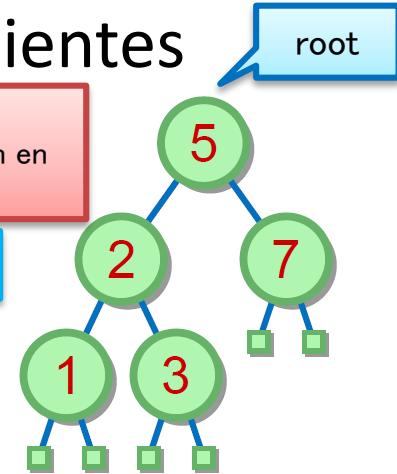
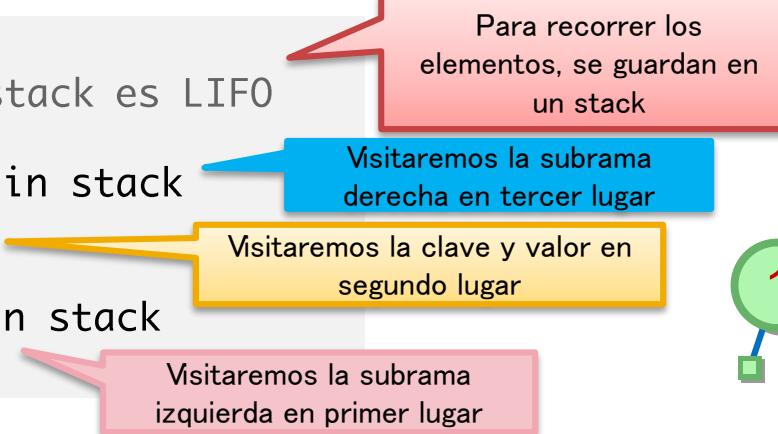
aux n xs devuelve un árbol con los n primeros elementos de xs y el resto de la lista (drop n xs). Estamos resolviendo un problema más general

- Sea  $T(n)$  el número de pasos para evaluar  $\text{minTreeB}'$  sobre una lista con  $n$  elementos:
  - $T(0) = 1$
  - $T(n) = O(1) + 2 T(n/2)$ , si  $n > 1$
  - La solución  $T(n)$  es de orden  $O(n)$  😊

# Iterador En-Orden para BST

- Usa un stack para organizar los nodos pendientes

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



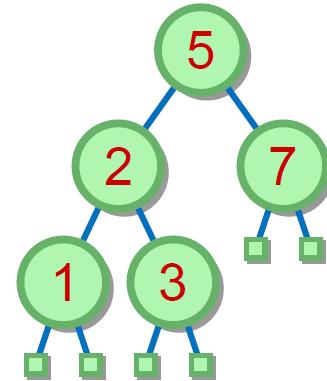
El stack se usará organizar el orden de los elementos a visitar.

El stack debe guardar dos tipos de árboles

- árboles para visitar (red tree)
- árboles de los que solo interesa la clave y el valor del nodo raíz (blue tree)

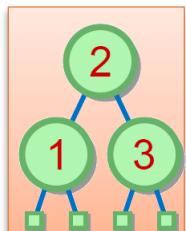
# Iterador En-Orden para BST (II)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree() es un método de la clase Traversal que encapsula el stack; las sucesivas llamadas cambian el stack y generan una secuencia de referencias a nodos

nextTree =>

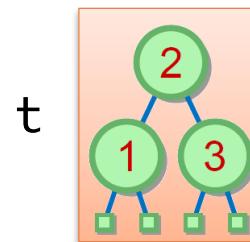
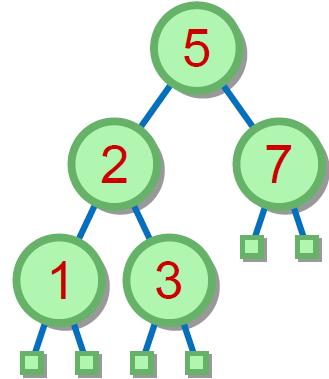


stack

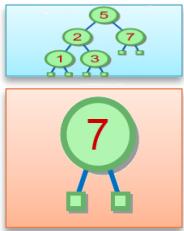
```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (III)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push right subtree in stack  
    push key in stack  
    if (node.left != null)  
        push left subtree in stack  
}
```



t  
nextTree =>

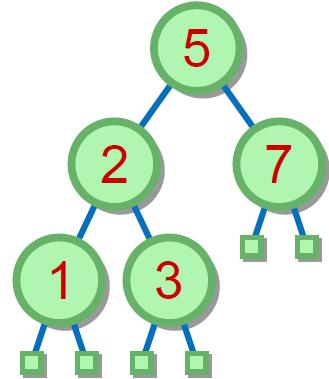
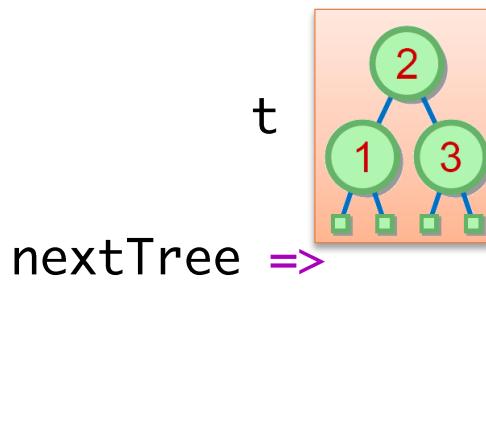


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (IV)

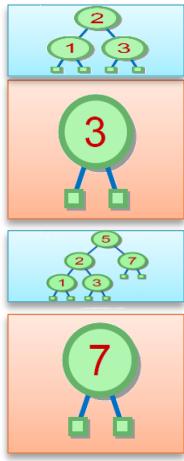
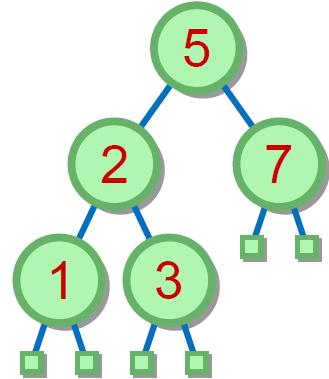
```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (VII)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



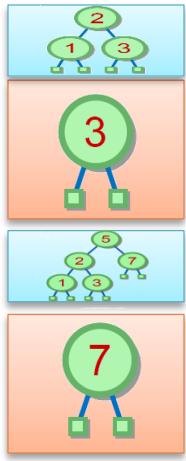
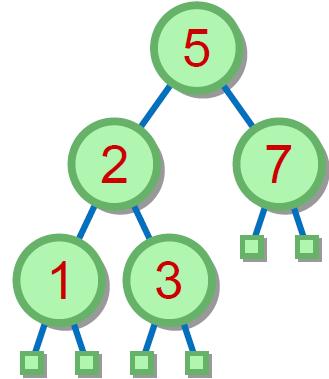
t

nextTree =>

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (VIII)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



t

nextTree =>

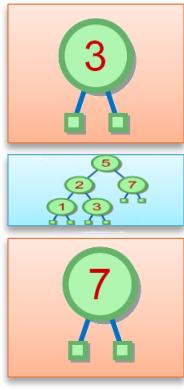
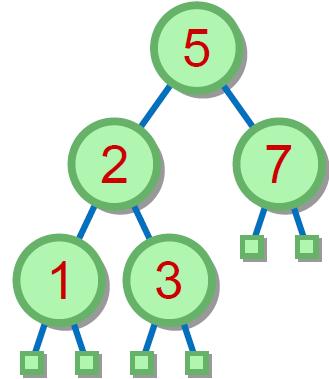


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (IX)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

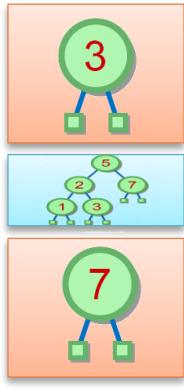
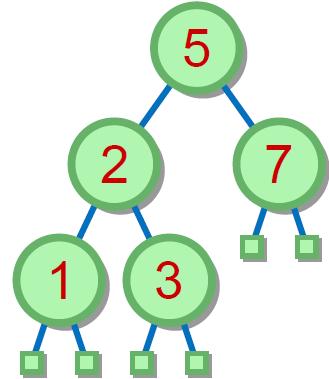


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (X)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

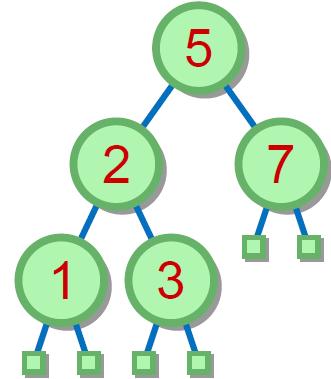


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XI)

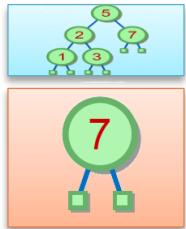
```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>



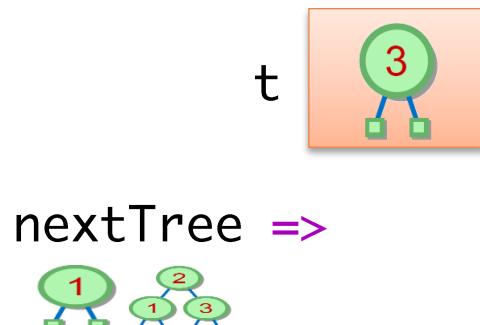
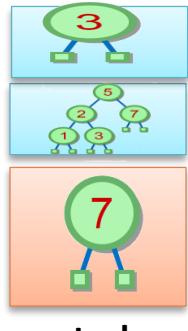
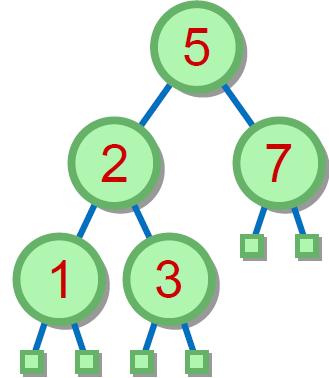
stack



```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XII)

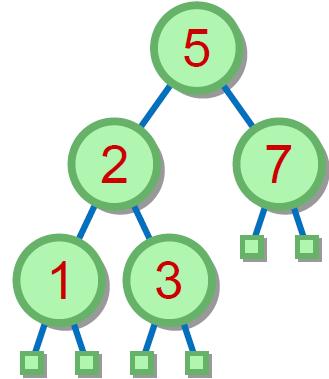
```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XIII)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

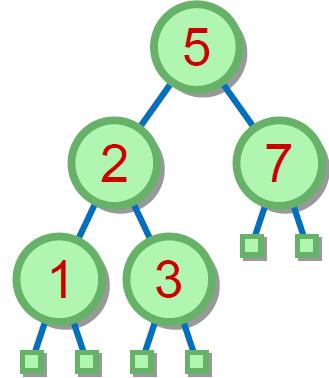


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XIV)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

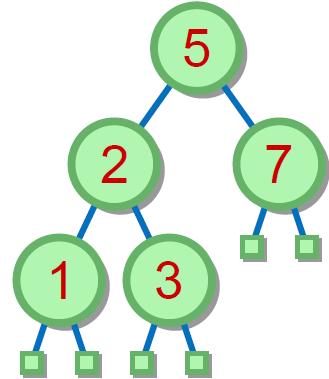


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XV)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

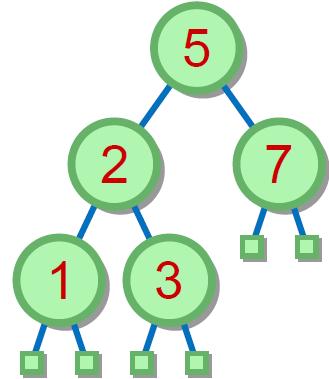


stack

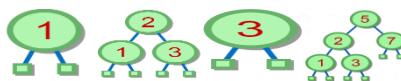
```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XVI)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

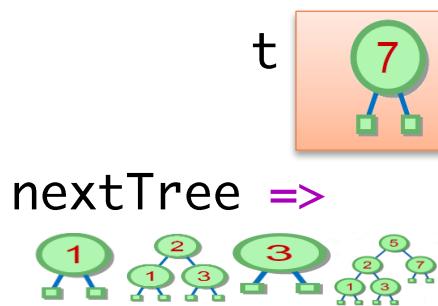
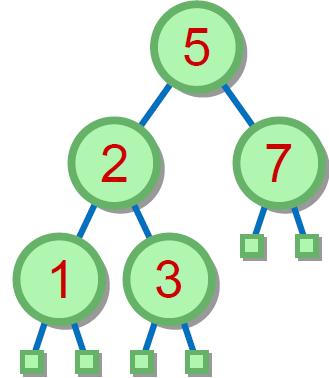


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XVII)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```

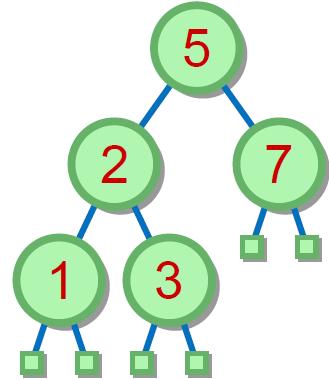


stack

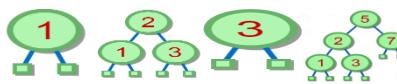
```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XVIII)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

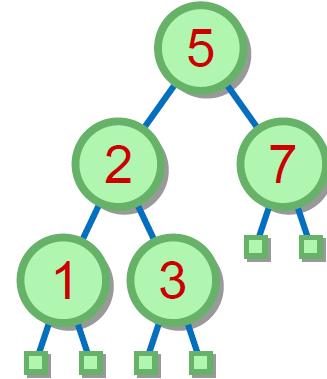


stack

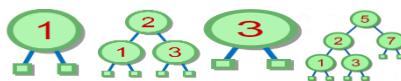
```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XIX)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



nextTree =>

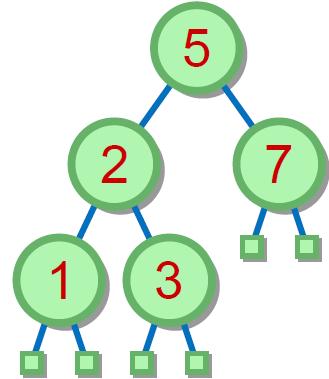


stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Iterador En-Orden para BST (XX)

```
void save(Tree<K,V> node) {  
    // en orden inverso, el stack es LIFO  
    if (node.right != null)  
        push red right subtree in stack  
    push blue node in stack  
    if (node.left != null)  
        push red left subtree in stack  
}
```



t

nextTree =>



stack

```
public Tree<K,V> nextTree() {  
    t = stack.top();  
    stack.pop();  
  
    while (t is a red tree) {  
        save(t);  
        t = stack.top();  
        stack.pop();  
    }  
    return t; //blue tree  
}
```

# Either: Tipo unión en Java

- El stack debe ser capaz de diferenciar los árboles almacenados (red o blue)
- El tipo base del stack será Either< Tree<K,V>, Tree<K,V> >
- Un objeto del tipo Either<A,B> pueden guardar:
  - o un valor de tipo A (izquierda), o un valor de tipo B (derecha)
  - Los árboles blue son los de la izquierda y red los de la derecha

## EJEMPLO

```
Either<Integer, String> either = new Left(5);
```

```
if(either.isLeft())  
    Integer n = either.left();
```

true si either guarda un Integer

Either guarda el Integer 5

Devuelve el Integer de either

```
either = new Right("five");
```

Either guarda el String "five"

```
if(either.isRight())  
    String s = either.right();
```

true si either guarda un String

Devuelve el String de either

# Either: Tipo unión en Java (II)

```
package dataStructures.either;
public interface Either<A,B> {
    boolean isLeft();
    boolean isRight();
    A left();
    B right();
}

public class Left<A,B> implements Either<A,B> {
    private A left;

    public Left(A x) { left = x; }

    public boolean isLeft() { return true; }
    public boolean isRight() { return false; }
    public A left() { return left; }
    public B right() { throw new NoSuchElementException("right on Left object"); }

    public String toString() { return "Left("+left+");" }
}

public class Right<A,B> implements Either<A,B> {
    private B right;

    public Right(B x) { right = x; } ...
}
```

# Iteradores para BST

```
private abstract class Traversal {  
    Stack<Either<Tree<K,V>, Tree<K,V>>> stack = new LinkedStack<>();  
  
    abstract void save(Tree<K,V> node);  
  
    public Traversal() {  
        if (root != null)  
            save(root);  
    }  
  
    public boolean hasNext() {  
        return !stack.isEmpty();  
    }  
  
    public Tree<K,V> nextTree() {  
        if (!hasNext())  
            throw new NoSuchElementException();  
  
        Either<Tree<K,V>, Tree<K,V>> either = stack.top();  
        stack.pop();  
  
        while (either.isRight()) {  
            Tree<K,V> node = either.right();  
            save(node);  
            either = stack.top();  
            stack.pop();  
        }  
        return either.left();  
    }  
}
```

Será definido para cada recorrido: inOrder preOrder y postOrder

Stack puede guardar o bien un árbol del que nos interesan valor y raíz (valor a la izquierda) o bien un árbol para visitar (valor a la derecha)

No hay más elementos ya que el stack está vacío

Mientras sea un árbol de la derecha hay que visitarlo

Es un árbol de la izquierda. Nos interesa solo la clave y el valor

# Iteradores para BST (II)

```
public class InOrderTravesal extends Travesal {  
    void save(Tree<K,V> node) {  
        // in reverse order, cause stack is LIFO  
        if (node.right != null)  
            stack.push(new Right<>(node.right));  
        stack.push(new Left<>(node));  
        if (node.left != null)  
            stack.push(new Right<>(node.left));  
    }  
}
```

Visitamos el subárbol derecho en tercer lugar (right)

Visitamos árbol con clave y valor en segundo lugar (left)

Visitamos el subárbol izquierdo en primer lugar (right)

```
private class InOrderIt extends InOrderTraversal implements Iterator<K> {  
    public K next() {  
        return super.nextTree().key;  
    }  
}
```

Solo se necesita la clave del nodo raíz

```
public class BST<K extends Comparable<? super K>, V>  
    implements SearchTree<K,V> {  
    public Iterable<K> inOrder() {  
        return new Iterable<K> {  
            public Iterator<K> iterator() {  
                return new InOrderIt();  
            }  
        };  
    }...  
}
```

# Iteradores para BST (III)

```
public class PreOrderTraversal extends Traversal {  
    void save(Tree<K,V> node) {  
        // in reverse order, cause stack is LIFO  
        if (node.right != null)  
            stack.push(new Right<>(node.right));  
        if (node.left != null)  
            stack.push(new Right<>(node.left));  
        stack.push(new Left<>(node));  
    }  
  
private class PreOrderIt extends PreOrderTraversal implements Iterator<K> {  
    public K next() {  
        return super.nextTree().key;  
    }  
  
public class BST<K extends Comparable<? super K>, V>  
    implements SearchTree<K,V> {  
    public Iterable<K> PreOrder() {  
        return new Iterable<K> {  
            public Iterator<K> iterator() {  
                return new PreOrderIt();  
            }  
        };  
    }...  
}
```

Visitamos el subárbol derecho en tercer lugar (right)

Visitamos el subárbol izquierdo en segundo lugar (right)

Visitamos árbol con clave y valor en primer lugar (left)

Solo se necesita la clave del nodo raíz

# Iteradores para BST (IV)

```
public class PostOrderTraversal extends Traversal {  
    void save(Tree<K,V> node) {  
        // in reverse order, cause stack is LIFO  
        stack.push(new Left<>(node));  
        if (node.right != null)  
            stack.push(new Right<>(node.right));  
        if (node.left != null)  
            stack.push(new Right<>(node.left));  
    }  
}
```

Visitamos árbol con clave y valor en tercer lugar (left)

Visitamos el subárbol derecho en segundo lugar (right)

Visitamos el subárbol izquierdo en primer lugar (right)

```
private class PostOrderIt extends PostOrderTraversal implements Iterator<K> {  
    public K next() {  
        return super.nextTree().key;  
    }  
}
```

Solo se necesita la clave del nodo raíz

```
public class BST<K extends Comparable<? super K>, V>  
    implements SearchTree<K,V> {  
    public Iterable<K> postOrder() {  
        return new Iterable<K> {  
            public Iterator<K> iterator() {  
                return new PostOrderIt();  
            }  
        };  
    }...  
}
```

# Iteradores para BST (V)

Solo definimos la iteración por valores para el recorrido InOrder

```
private class ValuesIt extends InOrderTraversal implements Iterator<V> {  
    public V next() {  
        return super.nextTree().value;  
    }  
}
```

Solo se necesita el valor  
del nodo raíz

```
public class BST<K extends Comparable<? super K>, V>  
    implements SearchTree<K, V> {  
    public Iterable<V> values() {  
        return new Iterable<V> {  
            public Iterator<V> iterator() {  
                return new ValuesIt();  
            }  
        };  
    }...  
}
```

# Iteradores para BST (VI)

Solo definimos la iteración por pares para el recorrido InOrder

```
private class KeyValuesIt extends InOrderTraversal
    implements Iterator<Tuple2<K, V>> {
    public Tuple2<K, V> next() {
        Tree<K, V> node = super.nextTree();
        return new Tuple2<>(node.key, node.value);
    }
}
```

Se necesita clave y valor del nodo raíz.  
Se devuelve una Tuple2 con los dos datos

```
public class BST<K extends Comparable<? super K>, V>
    implements SearchTree<K, V> {
    public Iterable<Tuple2<K, V>> keysValues() {
        return new Iterable<Tuple2<K, V>>() {
            public Iterator<Tuple2<K, V>> iterator() {
                return new KeyValuesIt();
            }
        };
    }
}
```

# El número de oro

- El número de oro  $\varphi$  es la raíz positiva de la ecuación

- $x^2 = 1 + x$ , es decir

$$\Phi = \frac{1 + \sqrt{5}}{2} = 1.618033988749895\dots$$

- Por tanto,  $\Phi^2 = 1 + \Phi$ , y de aquí:

$$\Phi^{n+2} = \Phi^n + \Phi^{n+1}$$

# El número de oro (II)

- Usando  $\Phi^h = \Phi^{h-1} + \Phi^{h-2}$ , es fácil demostrar (por inducción sobre  $h$ ) que la función  $N(h)$  definida por la recurrencia:

$$N(0) = 0,$$

$$N(1) = 1,$$

$$N(h) = 1 + N(h - 1) + N(h - 2), h > 1$$

- satisface  $\Phi^h - 1 \leq N(h) \leq \Phi^{h+1} - 1$
- Pero si  $n$  es número de nodos de un árbol AVL de altura  $h$ , entonces  $N(h) \leq n$ , de donde
  - $\Phi^h - 1 \leq n$
  - y de aquí:  $h \leq \lfloor \log_{\varphi} (n+1) \rfloor$
  - Recordemos que para todo árbol:  $\lceil \log_2 (n+1) \rceil \leq h$

# El número de oro (III)

- El número de oro  $\Phi$  (de Fidias?) y los números de Fibonacci aparecen de múltiples formas
  - – en la arquitectura y el arte
    - el Partenón de Fidias, la Gioconda de Leonardo da Vinci, ...
  - – en el reino vegetal, animal, astronomía
    - girasol, concha del Nautilus,
    - la forma de algunas galaxias ...
  - – en todas las ciencias, y en la computación !
    - geometría, grafos, recurrencias,
- Relación con los números de Fibonacci:

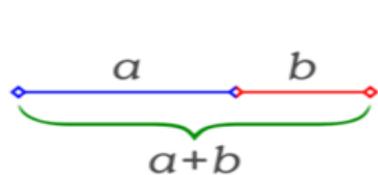
$$f_n = \frac{1}{\sqrt{5}} (\Phi^n - (-\Phi)^{-n}) \quad \Phi = \lim \frac{f_{n+1}}{f_n} \quad \Phi^n = \Phi f_n + f_{n-1}$$



# El número de oro (IV)

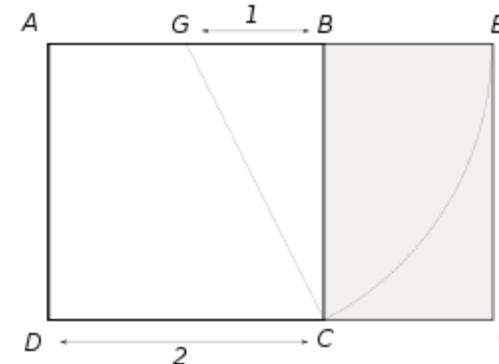
Origen del número áureo o divina proporción

Según Euclides en sus *Elementos* (Libro VI), es la proporción entre media y extrema razón: *el todo es a la parte como la parte al resto*. Es decir,



$$\frac{a+b}{a} = \frac{a}{b} \quad \text{o también} \quad \frac{x}{1} = \frac{1}{x-1}$$

Construcción de un Rectángulo áureo



- Fernando Corbalán, *La proporción áurea*, RBA (2010)
- Entrevista a Fernando Corbalán (You Tube)

# Árboles AVL en Java

```
package dataStructures.searchTree;

public class AVL<K extends Comparable<? super K>, V> implements SearchTree<K,V>{

    private static class Tree<C,D> {
        private C key;
        private D value;
        private int height;
        private Tree<C,D> left, right;

        public Tree(C k, D v) {
            key = k;
            value = v;
            height = 1;
            left = null;
            right = null;
        }
        // Aquí vendrán más métodos
    }

    private Tree<K,V> root;

    public AVL() {
        root = null;
    }

    public boolean isEmpty() {
        return root == null;
    }
}
```

Implementaremos una variante en la que en cada nodo almacenamos: una **clave** y un **valor**.

Los nodos en el árbol estarán **ordenados** según las **claves**.

# Árboles AVL en Java (II). Métodos de Tree

```
public static int height(Tree<?,?> tree) {  
    return tree == null ? 0 : tree.height;  
}  
  
public boolean rightLeaning() {  
    return height(left) < height(right);  
}  
  
public boolean leftLeaning() {  
    return height(left) > height(right);  
}  
  
void setHeight() {  
    height = 1 + Math.max(height(left), height(right));  
}
```

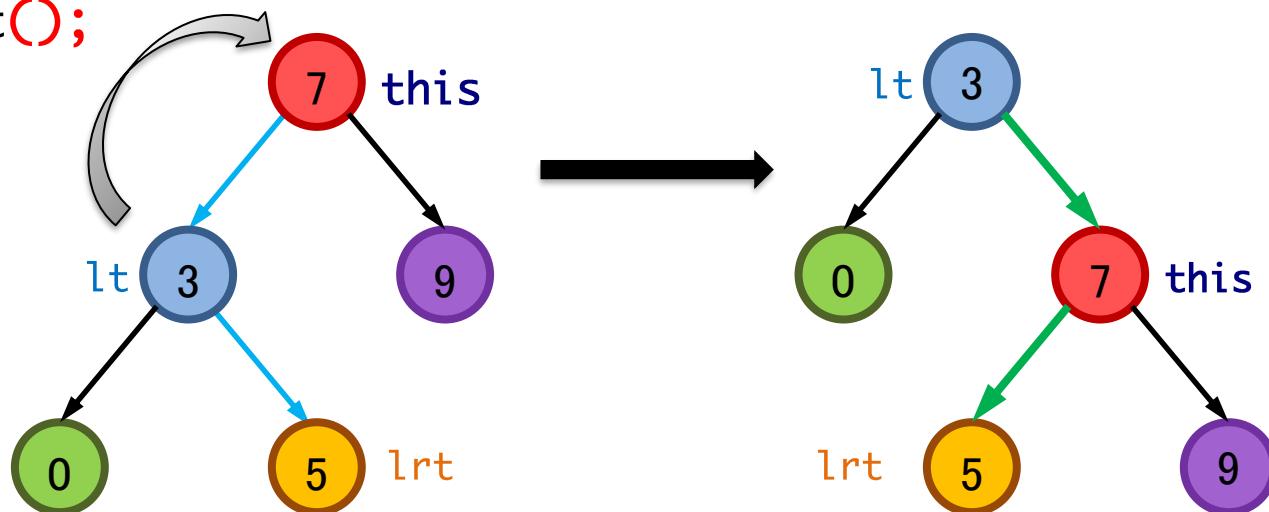
# Árboles AVL en Java (III). Métodos de Tree

```
// rota a la derecha el receptor. Devuelve la nueva raíz
public Tree<K,V> rotR() {
    Tree<K,V> lt = this.left;
    Tree<K,V> lrt = lt.right;

    this.left = lrt;
    this.setHeight();

    lt.right = this;
    lt.setHeight();

    return lt;
}
```



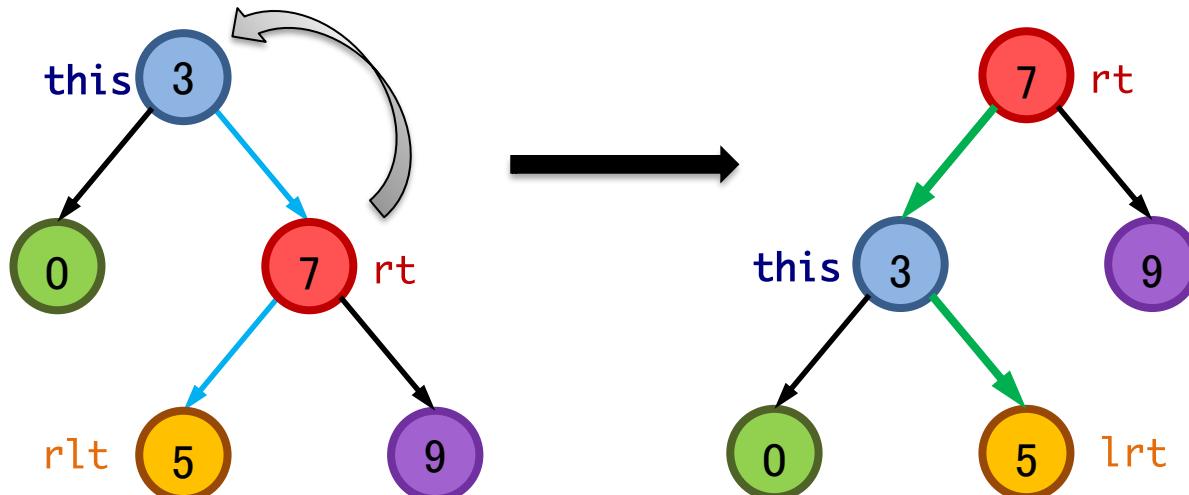
# Árboles AVL en Java (IV). Métodos de Tree

```
// Rota a la izquierda el receptor. Devuelve la nueva raíz
public Tree<K,V> rotL() {
    Tree<K,V> rt = this.right;
    Tree<K,V> rlt = rt.left;

    this.right = rlt;
    this.setHeight();

    rt.left = this;
    rt.setHeight();

    return rt;
}
```



# Árboles AVL en Java (V). Métodos de Tree

```
// Balancea el receptor. Devuelve el nodo despues de balancearlo
public Tree<K,V> balance() {
    int lh = height(left);
    int rh = height(right);

    Tree<K,V> balanced;

    if (lh - rh > 1 && left.leftLeaning()) {
        balanced = this.rotR(); //Se necesita simple rotación
    } else if (lh - rh > 1) {
        left = left.rotL(); //Se necesita doble rotación
        balanced = this.rotR();
    } else if (rh - lh > 1 && right.rightLeaning()) {
        balanced = this.rotL(); //Se necesita simple rotación
    } else if (rh - lh > 1) {
        right = right.rotR(); //Se necesita doble rotación
        balanced = this.rotL();
    } else {
        balanced = this; //No nec. rotacion
        balanced.setHeight();
    }
    return balanced;
}
```

```
balance :: a -> AVL a -> AVL a -> AVL a
balance k lt rt
| (lh-rh > 1) && leftLeaning lt = rotR (node k lt rt)
| (lh-rh > 1) = rotR (node k (rotL lt) rt)
| (rh>lh > 1) && rightLeaning rt = rotL (node k lt rt)
| (rh-lh > 1) = rotL (node k lt (rotR rt))
| otherwise = node k lt rt
where lh = height lt
      rh = height rt
```

# Árboles AVL en Java (VI)

- Igual que en un Árbol Binario de Búsqueda:

```
public V search(K key) {  
    return AVL.searchRec(root, key);  
}  
  
private static <C extends Comparable<? super C>, D>  
D searchRec(Tree<C,D> tree, C key) {  
    if (tree == null)  
        return null;  
    else if (key.compareTo(tree.key) == 0)  
        return tree.value;  
    else if (key.compareTo(tree.key) < 0)  
        return searchRec(tree.left, key);  
    else  
        return searchRec(tree.right, key);  
}  
  
public boolean isElem(K key) {  
    return search(key) != null;  
}
```

# Árboles AVL en Java (y VII)

- Igual que la inserción en un Árbol Binario de Búsqueda pero restaurando el balanceo en todos los nodos modificados:

```
public void insert(K k, V v) {  
    root = AVL.insertRec(root, k, v);  
}  
  
// returns modified tree  
private static <C extends Comparable<? super C>, D>  
    Tree<C,D> insertRec(Tree<C,D> node, C key, D value) {  
    if (node == null)  
        node = new Tree<>(key, value);  
    else if (key.compareTo(node.key) == 0)  
        node.value = value;  
    else if (key.compareTo(node.key) < 0) {  
        node.left = insertRec(node.left, key, value);  
        node = node.balance();  
    } else {  
        node.right = insertRec(node.right, key, value);  
        node = node.balance();  
    }  
    return node;  
}
```