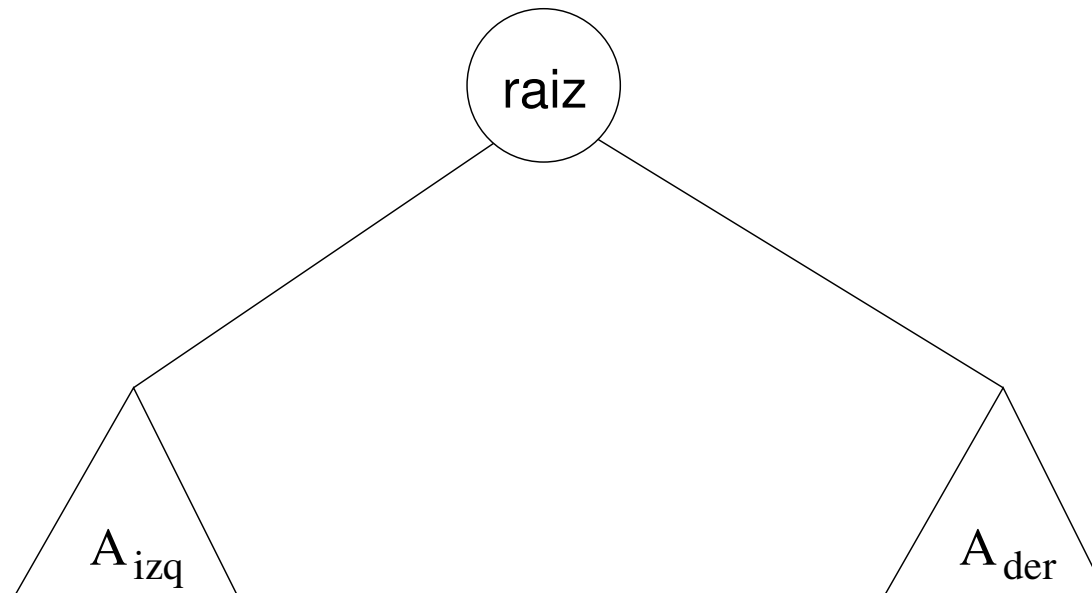


# ***Arboles Binarios de Búsqueda***

# Arboles Binarios



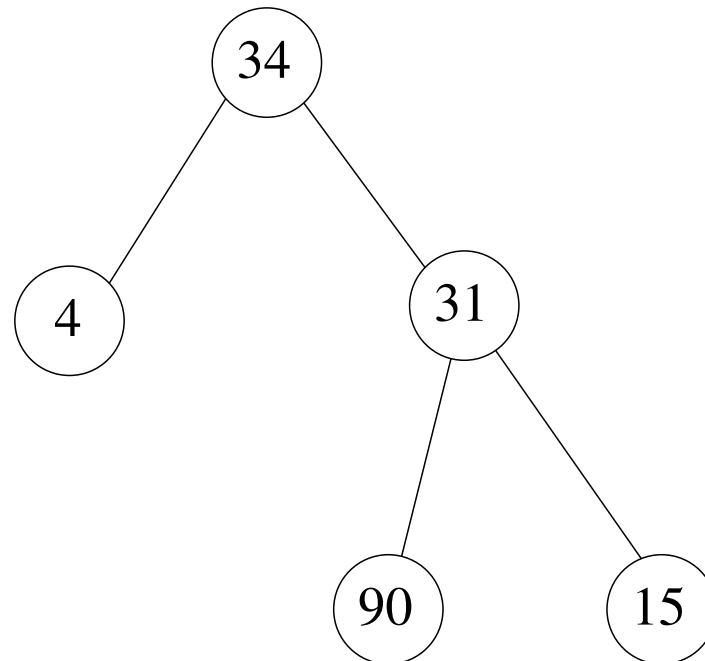
- ▷ **Arbol binario:** árbol ordenado de grado 2, que puede estar vacío o puede estar formado por un **nodo raíz** del que cuelgan dos subárboles binarios disjuntos, denominados **subárbol izquierdo** y **subárbol derecho**.



# Arboles Binarios

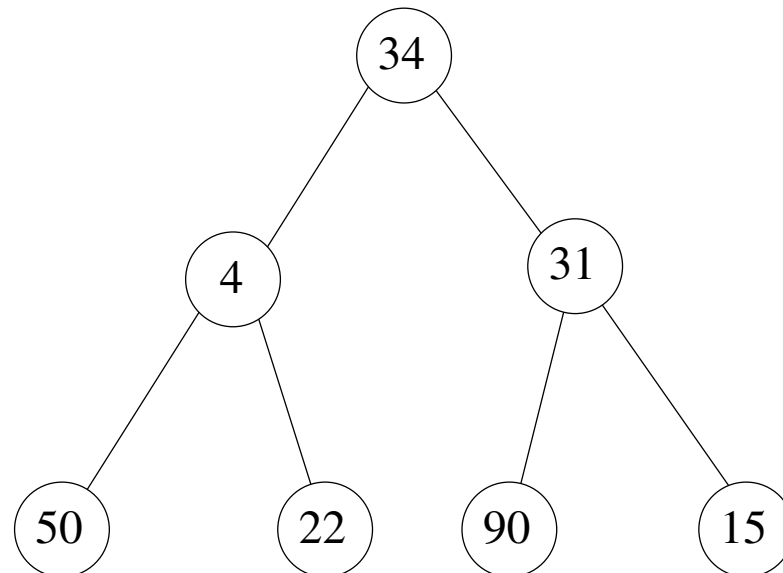


- ▶ Un árbol binario se dice **relleno** si todos sus nodos o bien tienen dos hijos o bien son hojas, es decir, si no contiene nodos con un solo hijo.
- ▶ El número de hojas en un árbol binario relleno es siempre igual al número de nodos internos más uno.



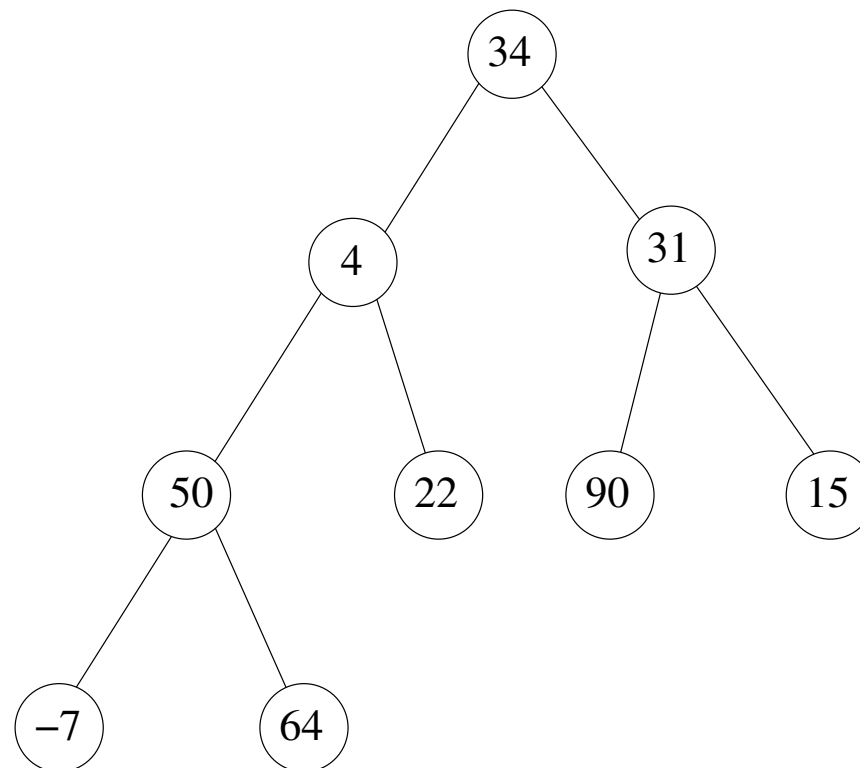
# Arboles Binarios

- ▷ Un árbol binario de altura  $h$  se dice **completo** si todos sus nodos interiores tienen dos hijos no vacíos y todas sus hojas están en el nivel  $h$ .
- ▷ El número de nodos de un árbol binario completo de altura  $h$  es igual a  $2^{h+1} - 1$ . Como todo árbol binario completo es también relleno,  $2^h$  de esos nodos son hojas y  $2^h - 1$  son nodos internos.



# Arboles Binarios

- Un árbol binario de altura  $h$  se dice **semicompleto** si los nodos de los niveles  $h$  y  $h-1$  son los únicos de grado inferior a 2 y las hojas del último nivel ocupan las posiciones más a la izquierda del mismo.



# Arboles de Búsqueda



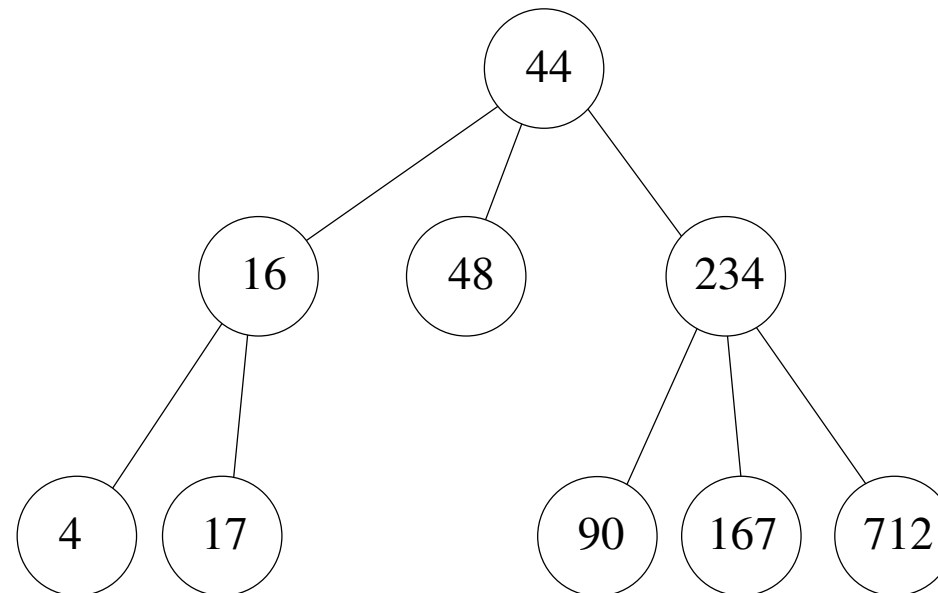
- ▷ La aplicación más importante de los árboles es organizar la información de manera jerárquica para acelerar los procesos de búsqueda, inserción y borrado.
- ▷ Normalmente, la clave de búsqueda se extrae de la propia información, directamente o mediante transformaciones adecuadas.
- ▷ Para clasificar la información sin ambigüedad, es necesario establecer entre las claves de búsqueda un conjunto de condiciones mutuamente excluyentes, tal que una y sólo una de ellas sea cierta.

# Arboles de Búsqueda



▷ Por ejemplo, dada una clave entera  $k$ , cualquier otra clave  $m$  podría clasificarse de acuerdo a las siguientes condiciones:

1.  $m \leq k/2$
2.  $k/2 < m < 2k$
3.  $m \geq 2k$



# Arboles de Búsqueda



- ▷ Un **árbol de búsqueda** se define como un árbol en el que, para cada nodo, las claves de los subárboles hijos satisfacen una y sólo una condición de un conjunto de  $n$  condiciones mutuamente excluyentes.
- ▷ Si  $n = 2$ , se tendrá un árbol de búsqueda **binario**; si  $n = 3$ , se tendrá un árbol de búsqueda **ternario**; etc.
- ▷ Así pues, un **árbol binario de búsqueda** (ABB) es un árbol binario en el que para cada nodo se definen dos condiciones mutuamente excluyentes, de forma que las claves de los nodos del subárbol izquierdo cumplen una de ellas, y las del subárbol derecho la otra.
- ▷ Habitualmente estas condiciones determinan una relación de orden, de manera que el recorrido en orden simétrico del árbol produce una secuencia ordenada de nodos.



- ▷ Un cierto conjunto de datos puede representarse mediante distintos ABB.
- ▷ El recorrido simétrico de estos ABB producirá la misma secuencia de nodos.
- ▷ Sin embargo, tendrán diferentes alturas y por tanto el coste promedio de búsqueda será distinto.
- ▷ Suponiendo datos equiprobables, serán óptimos aquellos ABB cuya altura sea mínima. De ahí el interés de los ABB equilibrados.
- ▷ Véanse los ABB de la página siguiente: ambos representan el mismo conjunto de datos; en el primer caso se trata de un ABB semicompleto de altura 2 (mínima), y en el segundo de un ABB completamente degenerado de altura 4 (máxima), equivalente a una lista ligada.

# Arboles de Búsqueda



ABB óptimo

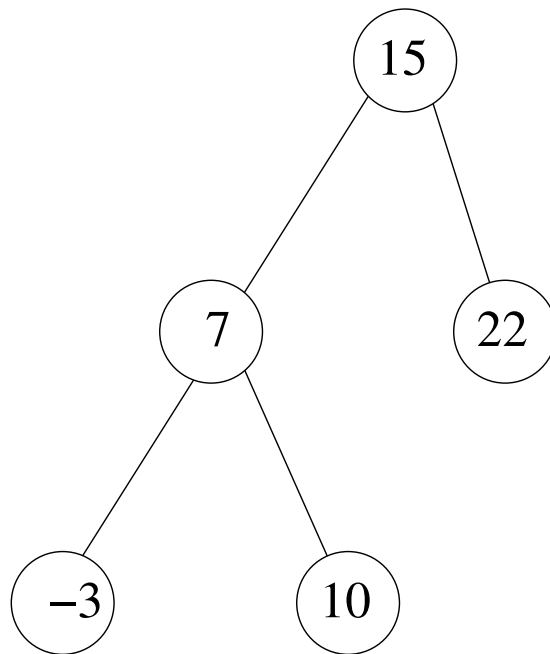
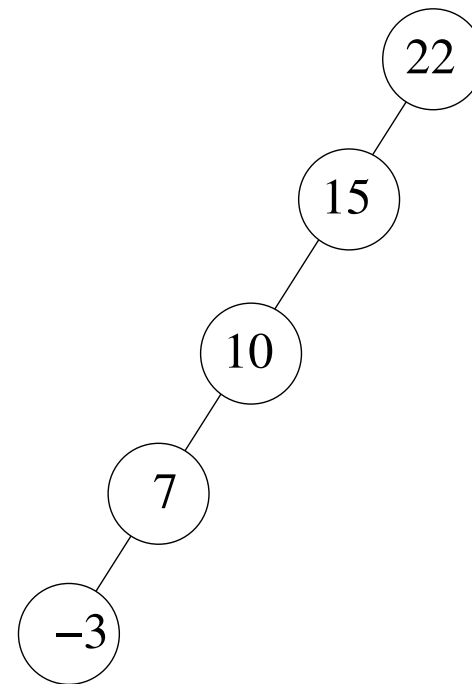


ABB completamente degenerado



# Arboles Binarios de Búsqueda

- ▷ **Operaciones básicas:** *Búsqueda, Inserción y Borrado.*
- ▷ La **complejidad** de estas operaciones está en  $O(h)$ , siendo  $h$  la altura del árbol; en ABB equilibrados,  $h \approx \log_2(n)$ , siendo  $n$  el número de nodos.
- ▷ Las tres operaciones se basan en un sencillo **esquema de búsqueda**:
  - ▷ Si el árbol  $t$  está vacío, el elemento buscado  $x$  no se encuentra en el mismo
  - ▷ En caso contrario, se pueden dar tres casos:
    - $clave(x) = clave(raíz(t)) \Rightarrow$  el elemento ha sido encontrado
    - $clave(x) < clave(raíz(t)) \Rightarrow$  se repite la búsqueda en el subárbol izquierdo
    - $clave(x) > clave(raíz(t)) \Rightarrow$  se repite la búsqueda en el subárbol derecho

# ***TAD ABB: definición***

## **Tipos**

- ▷ **ELEMENTO**
- ▷ **NODO**
- ▷ **ABB**

## **Constantes**

- ▷ **NODO\_NULO**

## **Funciones**

- ▷ **función** *clave*(x: ELEMENTO): entero

# ***TAD ABB: definición***

## **procedimiento** *inicializar*(ref t: ABB)

- ▷ Inicializa la variable t de tipo ABB
- ▷ Realiza las reservas dinámicas de memoria necesarias
- ▷ Como resultado, se tendrá un ABB t vacío
- ▷ *inicializar*(t): primera acción a realizar sobre t

# ***TAD ABB: definición***

**función** *buscar*(t: ABB, k: entero): NODO

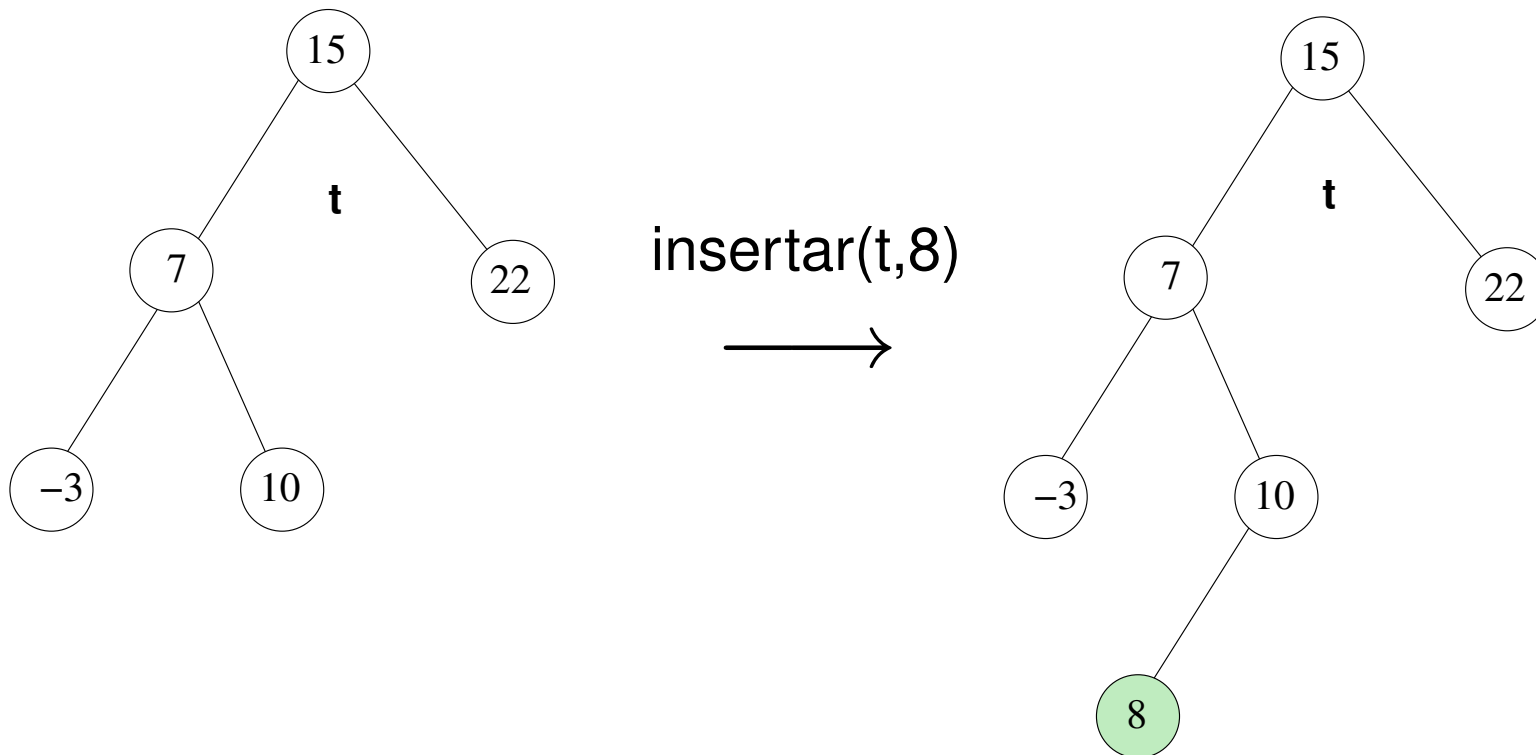
- ▷ Busca la clave k en el ABB t
- ▷ Si no la encuentra, retorna NODO\_NULO
- ▷ Si la encuentra, retorna el nodo correspondiente

# ***TAD ABB: definición***

**función** *insertar*(**ref** t: ABB, x: ELEMENTO): booleano

- ▷ Busca en el ABB t la posición de inserción para la información x
- ▷ Si existe ya un nodo con la información x, devuelve FALSO
- ▷ En caso contrario, crea un nuevo nodo con la información x, lo añade en la posición donde acabó la búsqueda y devuelve VERDADERO
- ▷ Nótese que los nuevos nodos se insertan siempre en las hojas, ya que la búsqueda termina cuando se accede a un subárbol vacío

# TAD ABB: definición





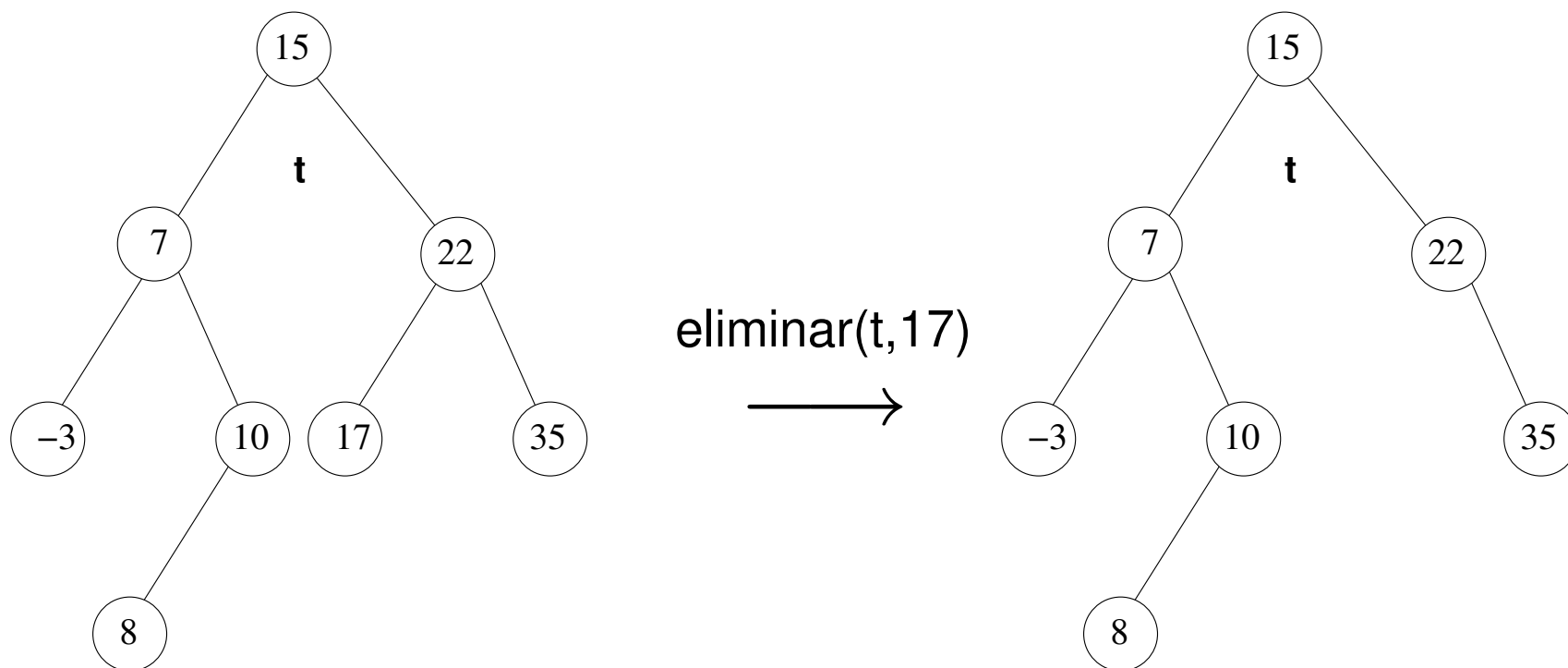
# ***TAD ABB: definición***

**función** *eliminar*(**ref** t: ABB, k: entero): booleano

- ▷ Busca en el ABB t un nodo de clave k
- ▷ Si no lo encuentra, devuelve FALSO
- ▷ Si lo encuentra, elimina el nodo de clave k y devuelve VERDADERO
- ▷ En la operación de borrado pueden darse tres casos:
  1. Eliminar una hoja
  2. Eliminar un nodo con un solo hijo
  3. Eliminar un nodo con dos hijos

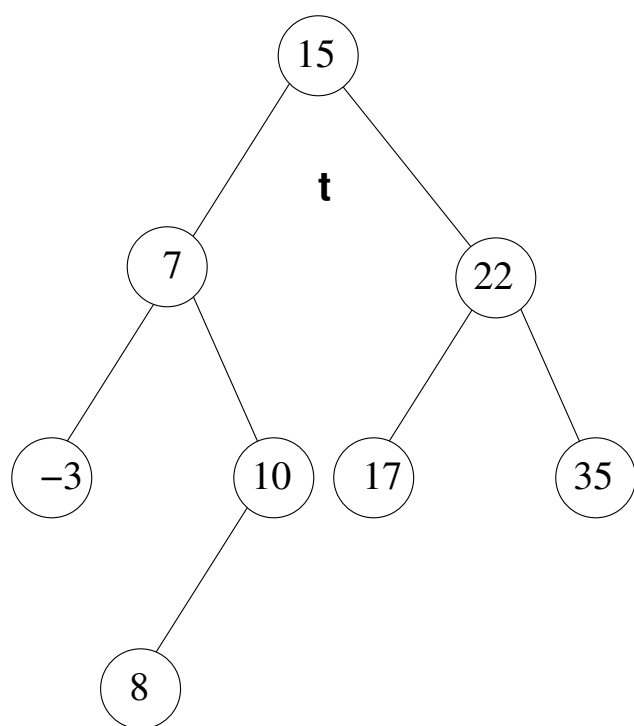
# TAD ABB: definición

**Eliminar una hoja:** se elimina el nodo en cuestión, y se actualiza a NODO\_NULO el campo correspondiente del nodo padre

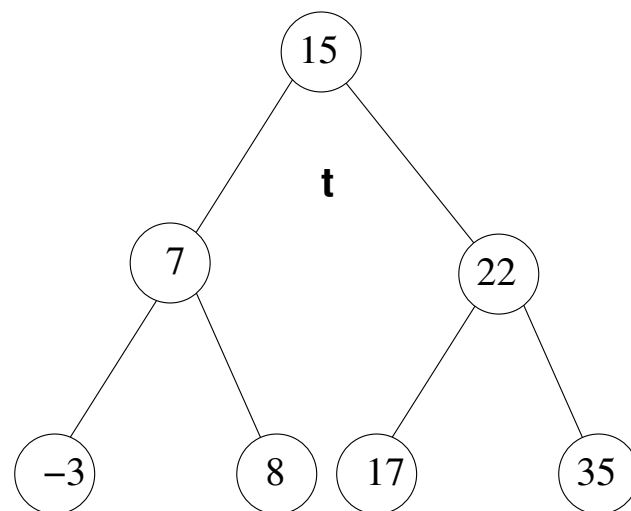
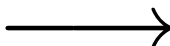


# TAD ABB: definición

**Eliminar un nodo con un solo hijo:** se elimina el nodo en cuestión, y se actualiza el campo correspondiente del nodo padre para que apunte al hijo del nodo eliminado

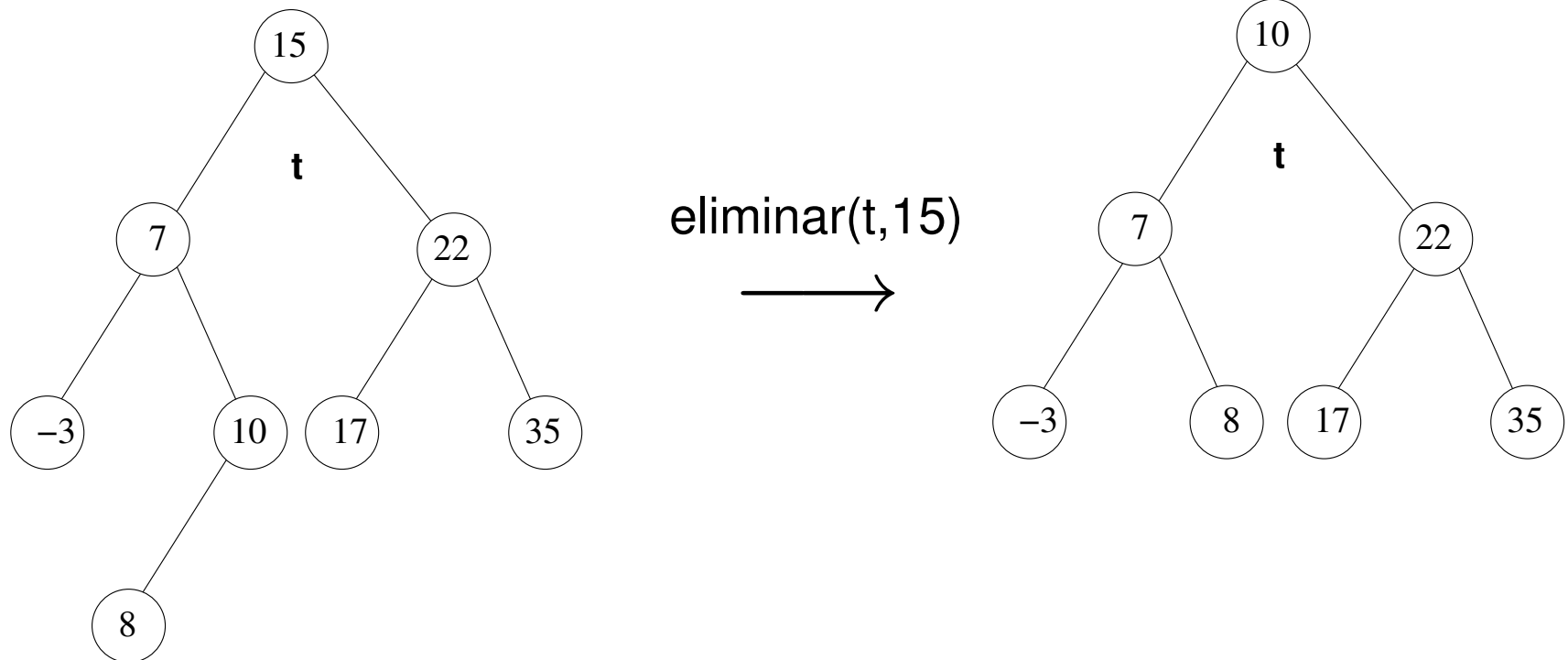


eliminar(t,10)



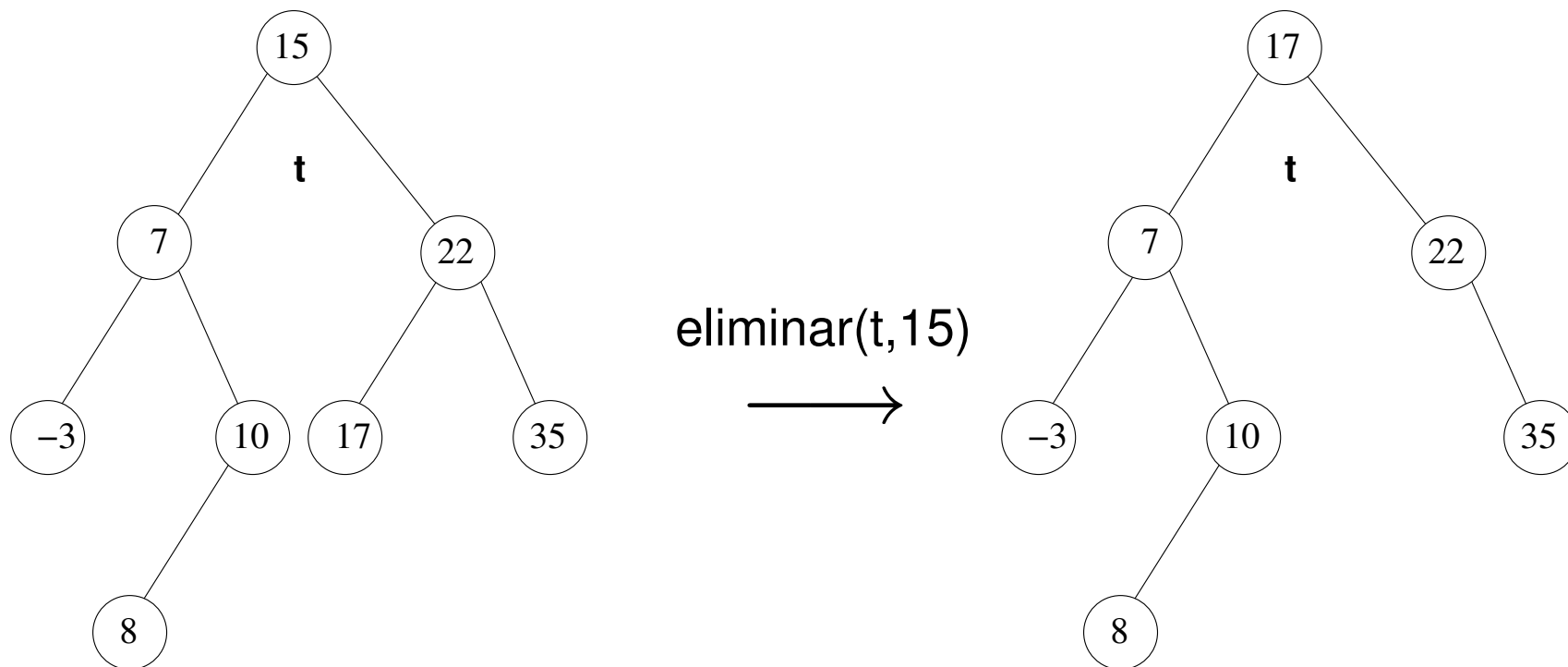
# TAD ABB: definición

**Eliminar un nodo con dos hijos:** se sustituye la información del nodo en cuestión por la de su predecesor/sucesor en un recorrido simétrico. A continuación se elimina el nodo predecesor/sucesor, que por fuerza ha de ser una hoja o un nodo con un solo hijo (casos anteriores).



# TAD ABB: definición

**Eliminar un nodo con dos hijos:** se sustituye la información del nodo en cuestión por la de su predecesor/sucesor en un recorrido simétrico. A continuación se elimina el nodo predecesor/sucesor, que por fuerza ha de ser una hoja o un nodo con un solo hijo (casos anteriores).



## ***TAD ABB: definición***

**función** *máximo*(t: ABB): NODO

- ▷ Devuelve el nodo de clave máxima en el ABB t
- ▷ Si t está vacío, devuelve NODO\_NULO

# ***TAD ABB: definición***

**función** *mínimo*(t: ABB): NODO

- ▷ Devuelve el nodo de clave mínima en el ABB t
- ▷ Si t está vacío, devuelve NODO\_NULO

# ***TAD ABB: definición***

**función** *predecesor*(t: ABB, k: entero): NODO

- ▷ Devuelve el nodo que precede al nodo de clave k en un recorrido simétrico del ABB t
- ▷ Si la clave k no se encuentra en t, o el nodo correspondiente no tiene predecesor (por ser el primero en el recorrido simétrico de t), la función devuelve NODO\_NULO



# ***TAD ABB: definición***

**función** *sucesor*(t: ABB, k: entero): NODO

- ▷ Devuelve el nodo que sucede al nodo de clave k en un recorrido simétrico del ABB t
- ▷ Si la clave k no se encuentra en t, o el nodo correspondiente no tiene sucesor (por ser el último en el recorrido simétrico de t), la función devuelve NODO\_NULO

# ***TAD ABB: definición***

**función** *raíz*(t: ABB): NODO

- ▷ Devuelve el nodo raíz del ABB t
- ▷ Si t está vacío, devuelve NODO\_NULO

# ***TAD ABB: definición***

**función** *sub\_izq*(t: ABB): ABB

- ▷ Devuelve el subárbol izquierdo del ABB t
- ▷ **Importante:** no devuelve un nuevo árbol sino una referencia a la parte izquierda del ABB t
- ▷ Precondición: t no vacío

# ***TAD ABB: definición***

**función** *sub\_der*(t: ABB): ABB

- ▷ Devuelve el subárbol derecho del ABB t
- ▷ **Importante:** no devuelve un nuevo árbol sino una referencia a la parte derecha del ABB t
- ▷ Precondición: t no vacío

# ***TAD ABB: definición***

**función** *valor*(t: ABB, n: NODO): ELEMENTO

- ▷ Devuelve el valor almacenado en el nodo n del ABB t
- ▷ Precondición:  $n \neq \text{NODO\_NULO}$

# ***TAD ABB: definición***

**función** *vacío?*(t: ABB): booleano

▷ Devuelve:

VERDADERO

si t está vacío

FALSO

en caso contrario

# ***TAD ABB: definición***

**función** *altura*(t: ABB): entero

- ▷ Devuelve la altura del ABB t

# ***TAD ABB: implementación***

**constantes** NODO\_NULO = NULO **fin\_constantes**

**tipos**

ITEM = **registro**

valor: ELEMENTO

padre: apuntador a ITEM

hijo\_izq: apuntador a ITEM

hijo\_der: apuntador a ITEM

**fin\_registro**

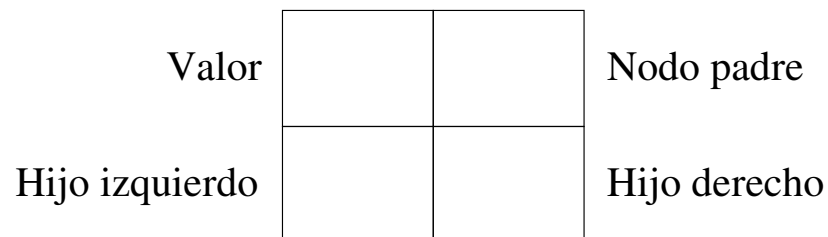
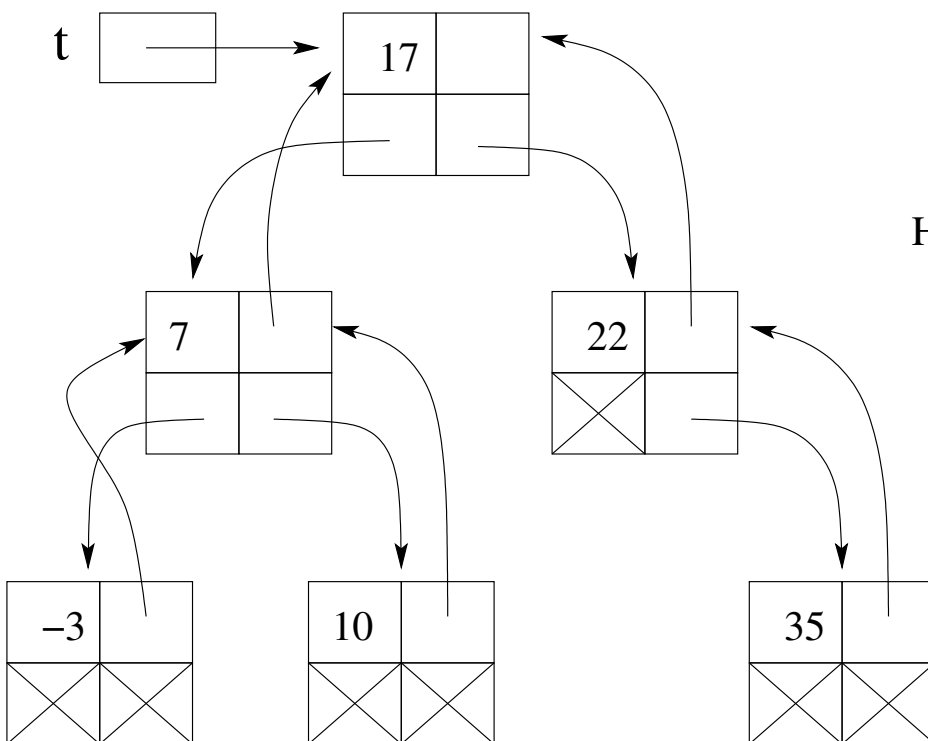
NODO = apuntador a ITEM

ABB = NODO

**fin\_tipos**



# TAD ABB: implementación



# ***TAD ABB: implementación***

**procedimiento** *inicializar*(**ref** t: ABB)

**principio**

t  $\leftarrow$  NODO\_NULO

**fin**

# ***TAD ABB: implementación***

```

función buscar(t: ABB, k: entero): NODO
var   p: NODO   fin_var
usa   clave()  fin_usa
principio
    p ← t
    si p = NODO_NULO OR clave(apuntado(p).valor) = k entonces
        devolver p
    fin_si
    si clave(apuntado(p).valor) > k entonces
        devolver buscar(apuntado(p).hijo_izq, k)
    si_no
        devolver buscar(apuntado(p).hijo_der, k)
    fin_si
fin

```

# TAD ABB: implementación

**función** *insertar*(**ref** t: ABB, x: ELEMENTO): booleano

**var** p, q, r: NODO **fin\_var**

**usa** *clave()* **fin\_usa**

**principio**

**si** t = NODO\_NULO **entonces**

t ← *reservar*(1,ITEM)

apuntado(t).valor ← x

apuntado(t).padre ← NODO\_NULO

apuntado(t).hijo\_izq ← NODO\_NULO

apuntado(t).hijo\_der ← NODO\_NULO

**si\_no**

q ← t

**mientras** q ≠ NODO\_NULO **hacer**

**si** *clave*(apuntado(q).valor) = *clave*(x) **entonces**

**devolver** FALSO

**fin\_si**

r ← q

[ continua en la página siguiente ... ]

# TAD ABB: implementación

[ ... viene de la página anterior ]

**si** clave(apuntado(q).valor) > clave(x) **entonces**

q ← apuntado(q).hijo\_izq

**si\_no**

q ← apuntado(q).hijo\_der

**fin\_si**

**fin\_mientras**

p ← *reservar*(1,ITEM)

apuntado(p).valor ← x

apuntado(p).padre ← r

apuntado(p).hijo\_izq ← NODO\_NULO

apuntado(p).hijo\_der ← NODO\_NULO

**si** clave(apuntado(r).valor) > clave(x) **entonces**

apuntado(r).hijo\_izq ← p

**si\_no**

apuntado(r).hijo\_der ← p

**fin\_si**

**fin\_si**

**devolver** VERDADERO

**fin** /\* insertar \*/

# TAD ABB: implementación

```

función eliminar(ref t: ABB, k: entero): booleano
var  p, q, r: NODO  fin_var
usa  clave()  fin_usa
principio
    p ← buscar(t,k)
    si p = NODO_NULO entonces
        devolver FALSO
    fin_si
    si apuntado(p).hijo_izq = NODO_NULO OR
        apuntado(p).hijo_der = NODO_NULO entonces
        q ← apuntado(p).padre
        si q ≠ NODO_NULO entonces /* casos 1 y 2 con p ≠ raíz */
            si apuntado(p).hijo_izq ≠ NODO_NULO entonces /* caso 2:izquierda */
                si p = apuntado(q).hijo_izq entonces
                    apuntado(q).hijo_izq ← apuntado(p).hijo_izq
                si_no
                    apuntado(q).hijo_der ← apuntado(p).hijo_izq
            fin_si
        apuntado(apuntado(p).hijo_izq).padre ← q

```

[ continua en la página siguiente ... ]

# TAD ABB: implementación

[ ... viene de la página anterior ]

```

si_no /* casos 2:derecha y 1*/
    si apuntado(p).hijo_der ≠ NODO_NULO entonces
        si p = apuntado(q).hijo_izq entonces
            apuntado(q).hijo_izq ← apuntado(p).hijo_der
        si_no
            apuntado(q).hijo_der ← apuntado(p).hijo_der
        fin_si
        apuntado(apuntado(p).hijo_der).padre ← q
    si_no /* caso 1 */
        si p = apuntado(q).hijo_izq entonces
            apuntado(q).hijo_izq ← NODO_NULO
        si_no
            apuntado(q).hijo_der ← NODO_NULO
        fin_si
    fin_si
fin_si

```

[ continua en la página siguiente ... ]

# TAD ABB: implementación

[ ... viene de la página anterior ]

**si\_no** /\* casos 1 y 2 con p = raíz \*/

**si** apuntado(p).hijo\_izq  $\neq$  NODO\_NULO **entonces**

t  $\leftarrow$  apuntado(p).hijo\_izq

apuntado(t).padre  $\leftarrow$  NODO\_NULO

**si\_no**

t  $\leftarrow$  apuntado(p).hijo\_der

**si** t  $\neq$  NODO\_NULO **entonces**

apuntado(t).padre  $\leftarrow$  NODO\_NULO

**fin\_si**

**fin\_si**

**fin\_si**

liberar(p)

**si\_no** /\* caso 3 \*/

q  $\leftarrow$  máximo(apuntado(p).hijo\_izq)

apuntado(p).valor  $\leftarrow$  apuntado(q).valor

eliminar(apuntado(p).hijo\_izq, clave(apuntado(q).valor))

**fin\_si**

**devolver** VERDADERO

**fin** /\* eliminar \*/



# ***TAD ABB: implementación***

```

función máximo(t: ABB): NODO
var
    p: NODO
fin_var
principio
    si t = NODO_NULO entonces
        devolver NODO_NULO
    fin_si
    p  $\leftarrow$  t
    mientras apuntado(p).hijo_der  $\neq$  NODO_NULO hacer
        p  $\leftarrow$  apuntado(p).hijo_der
    fin_mientras
    devolver p
fin

```

# ***TAD ABB: implementación***

**función** *mínimo*(t: ABB): NODO

**var**

p: NODO

**fin\_var**

**principio**

**si** t = NODO\_NULO **entonces**

**devolver** NODO\_NULO

**fin\_si**

p  $\leftarrow$  t

**mientras** apuntado(p).hijo\_izq  $\neq$  NODO\_NULO **hacer**

p  $\leftarrow$  apuntado(p).hijo\_izq

**fin\_mientras**

**devolver** p

**fin**

# TAD ABB: implementación

```

función predecesor(t: ABB, k: entero): NODO
var
    p, q: NODO
fin_var
principio
    p  $\leftarrow$  buscar(t,k)
    si p = NODO_NULO entonces
        devolver NODO_NULO
    fin_si
    si apuntado(p).hijo_izq  $\neq$  NODO_NULO entonces
        devolver máximo(apuntado(p).hijo_izq)
    fin_si
    q  $\leftarrow$  apuntado(p).padre
    mientras q  $\neq$  NODO_NULO AND apuntado(q).hijo_izq = p hacer
        p  $\leftarrow$  q
        q  $\leftarrow$  apuntado(q).padre
    fin_mientras
    devolver q
fin

```

# ***TAD ABB: implementación***

```

función sucesor(t: ABB, k: entero): NODO
var
    p, q: NODO
fin_var
principio
    p  $\leftarrow$  buscar(t,k)
    si p = NODO_NULO entonces
        devolver NODO_NULO
    fin_si
    si apuntado(p).hijo_der  $\neq$  NODO_NULO entonces
        devolver mínimo(apuntado(p).hijo_der)
    fin_si
    q  $\leftarrow$  apuntado(p).padre
    mientras q  $\neq$  NODO_NULO AND apuntado(q).hijo_der = p hacer
        p  $\leftarrow$  q
        q  $\leftarrow$  apuntado(q).padre
    fin_mientras
    devolver q
fin

```

# ***TAD ABB: implementación***

**función** *raíz*(t: ABB): NODO

**principio**

**devolver** t

**fin**

# ***TAD ABB: implementación***

$P \equiv \{ t \text{ no vacío} \}$

**función** *sub\_izq*(t: ABB): ABB

**principio**

**devolver** apuntado(t).hijo\_izq

**fin**

# ***TAD ABB: implementación***

$P \equiv \{ t \text{ no vacío} \}$

**función** *sub\_der*(t: ABB): ABB

**principio**

**devolver** apuntado(t).hijo\_der

**fin**

# ***TAD ABB: implementación***

$P \equiv \{ n \neq \text{NODO\_NULO} \}$

**función** *valor*(t: ABB, n: NODO): ELEMENTO

**principio**

**devolver** apuntado(n).valor

**fin**



# ***TAD ABB: implementación***

**función** *vacío?*(t: ABB): booleano

**principio**

**si** t = NODO\_NULO **entonces**

**devolver** VERDADERO

**si\_no**

**devolver** FALSO

**fin\_si**

**fin**

# ***TAD ABB: implementación***

```

función altura(t: ABB): entero
var   h, h_aux: entero   fin_var
principio
    si t = NODO_NULO entonces
        devolver 0
    fin_si
    h ← 0
    si apuntado(t).hijo_izq ≠ NODO_NULO entonces
        h ← 1 + altura(apuntado(t).hijo_izq)
    fin_si
    si apuntado(t).hijo_der ≠ NODO_NULO entonces
        h_aux ← 1 + altura(apuntado(t).hijo_der)
        si h_aux > h entonces
            h ← h_aux
        fin_si
    fin_si
    devolver h
fin

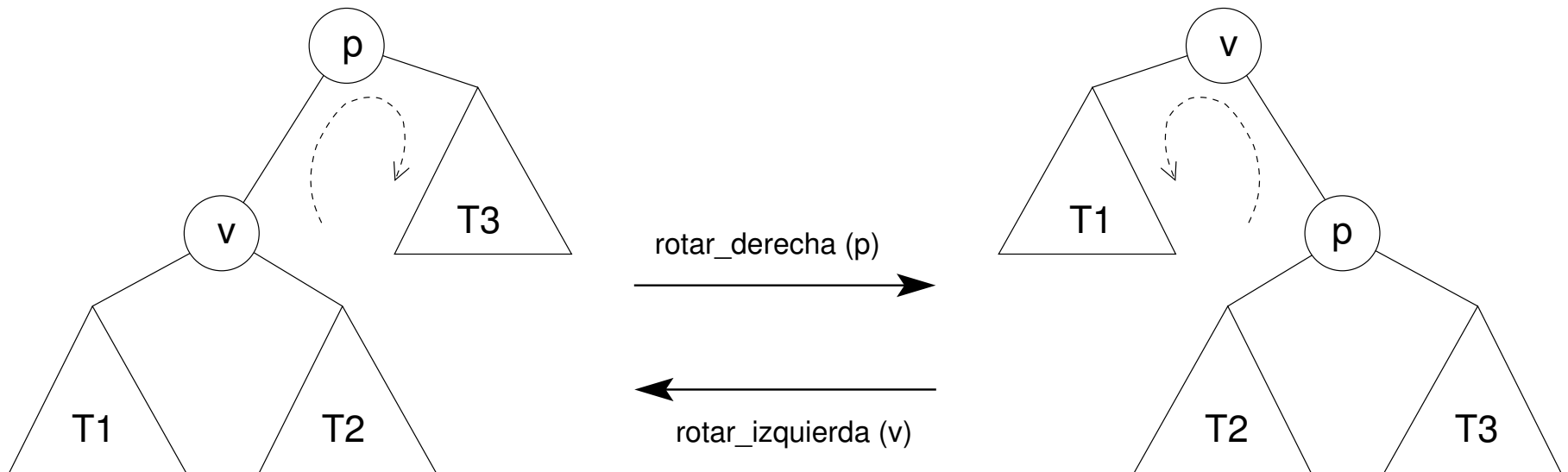
```

- ▷ Se dice que un ABB con  $n$  nodos es *equilibrado* si su altura es próxima (es decir, igual o ligeramente superior) a  $\log(n)$ .
- ▷ Los ABB equilibrados proporcionan tiempos de búsqueda en  $\Theta(\log n)$ .
- ▷ Las técnicas para conseguir ABB equilibrados se basan en reasignar nodos o subárboles dentro del ABB tras una operación de inserción o borrado que ha producido un desequilibrio en la estructura.
- ▷ Estas reasignaciones (conocidas como *rotaciones*) preservan la propiedad fundamental de los ABB (ver página siguiente).
- ▷ Atendiendo al *criterio de equilibrio*, se distinguen varios tipos de ABB equilibrados: **árboles 2-3**, **árboles rojo-negro**, **árboles AVL**, etc.

# ABB Equilibrados: Rotaciones



$\text{clave}(T1) < \text{clave}(v) < \text{clave}(T2) < \text{clave}(p) < \text{clave}(T3)$

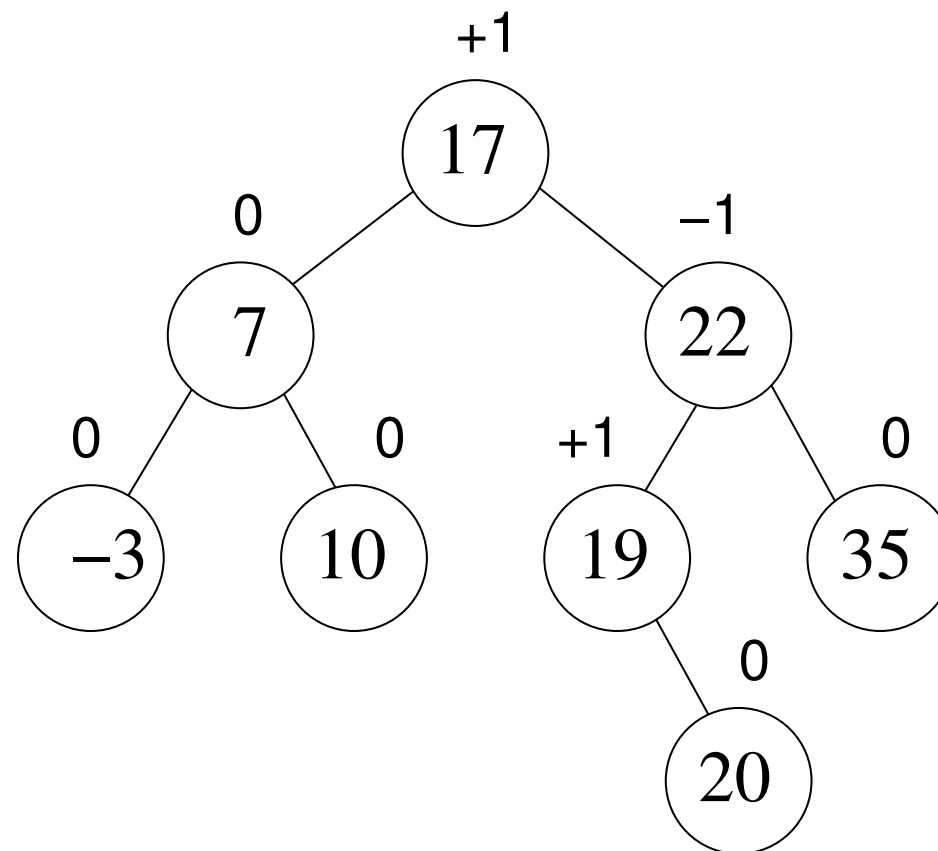


- ▷ Un **Arbol AVL** (G.M. Adelson-Velskii y E.M. Landis, 1962) es *un ABB en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren a lo sumo en 1*.
- ▷ Esta restricción se conoce como **propiedad de los árboles AVL**.
- ▷ La altura de un árbol AVL con  $n$  nodos está en  $\Theta(\log n)$  (véase Heileman, pp. 194-195).
- ▷ Los árboles AVL se representan igual que un ABB, sólo que añadiendo a cada nodo un campo adicional que indica su grado de equilibrio.
- ▷ Este valor se calcula como la diferencia entre las alturas de los subárboles derecho e izquierdo, que puede ser +1, 0 o -1 (véase el ejemplo siguiente).

# Arboles AVL



Ejemplo de árbol AVL con el grado de equilibrio de cada nodo



# Arboles AVL

- ▷ En un árbol AVL se emplean los mismos procedimientos de consulta definidos genéricamente para un ABB.
- ▷ Además, como la altura de un árbol AVL de  $n$  nodos está en  $\Theta(\log n)$ , se tiene la seguridad de que esas operaciones tendrán en el peor caso un coste logarítmico.
- ▷ Sin embargo, las operaciones *insertar()* y *eliminar()* podrían modificar el equilibrio de los nodos más allá del rango  $[-1,1]$  permitido.
- ▷ Así sucede en el ejemplo anterior si se añade el valor 21 o si se elimina el valor 35.
- ▷ Será necesario, por tanto, añadir el código necesario para que estas operaciones mantengan la propiedad de los árboles AVL.

# Arboles AVL: Inserción

- ▷ Para insertar un nuevo nodo en un árbol AVL, en primer lugar se aplica el algoritmo genérico de inserción definido para ABB. El coste de esta operación está en  $O(\log n)$ .
- ▷ A continuación se recorre el camino de regreso desde la hoja insertada hacia la raíz y se van actualizando los equilibrios de los nodos.
- ▷ El camino de vuelta podría llegar hasta la raíz sin que se produjeran disequilibrios, por lo que no sería necesario modificar la estructura del árbol. Nótese que el coste de vuelta estaría también en  $O(\log n)$ .
- ▷ Sin embargo, si al actualizar el equilibrio de un nodo, éste pasa de +1 a +2 o de -1 a -2, entonces es necesario ajustar el subárbol de este nodo para recuperar la propiedad de los árboles AVL.



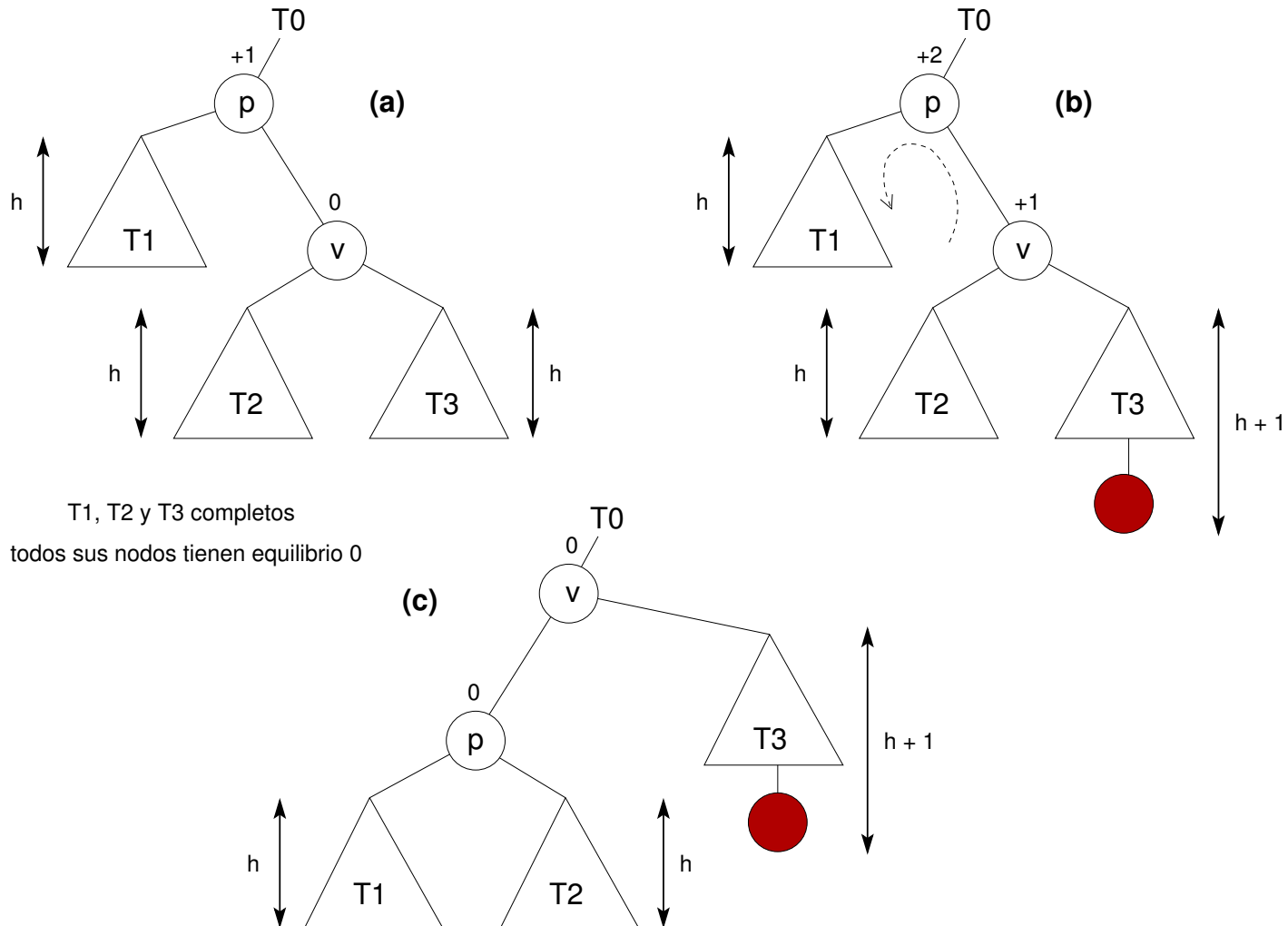
# Arboles AVL: Inserción

- ▷ El nodo en cuestión se denomina **pivote**, y sobre él será necesario aplicar una de las siguientes operaciones:
  - ▷ una **rotación simple** a izquierda o a derecha
  - ▷ una **rotación doble** izquierda-derecha o derecha-izquierda
- ▷ Estas rotaciones mantienen la altura que tenía el pivote antes de la inserción. Así, una vez corregida la estructura por debajo del pivote, la propiedad de los árboles AVL se mantiene en el resto de la estructura, y no es necesario seguir explorándola.
- ▷ De hecho, el único **pivote potencial** a considerar es el primer nodo en el camino de vuelta con equilibrio igual a +1 o -1. Si este nodo no cambia a +2 o -2, la estructura del árbol no tendrá que ser modificada.
- ▷ Finalmente, las rotaciones simples o dobles tienen un coste  $\Theta(1)$ , por lo que el coste de inserción en un árbol AVL estará en  $O(\log n)$ .

# Arboles AVL: Inserción



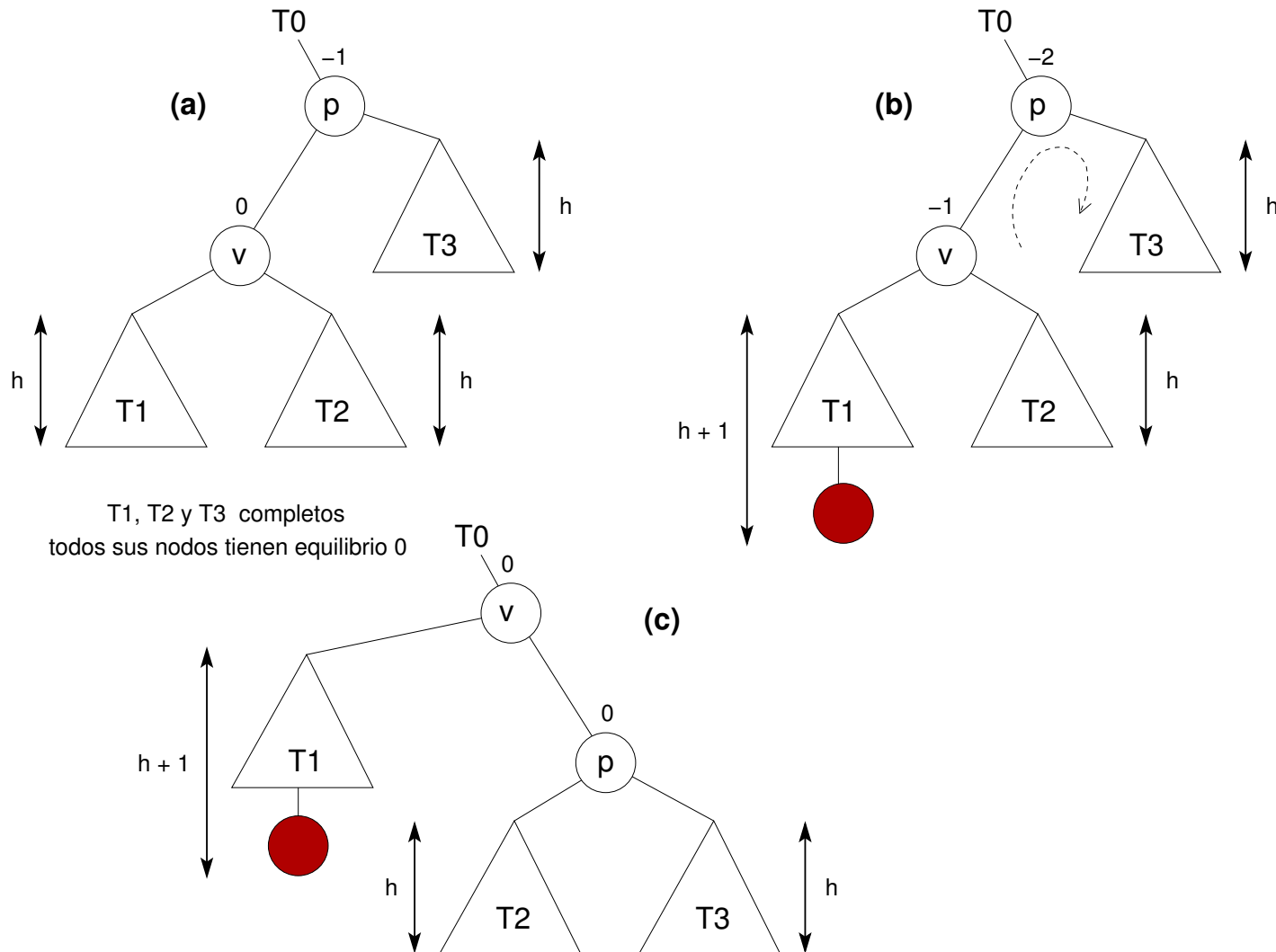
## Rotación simple a izquierda



# Arboles AVL: Inserción



## Rotación simple a derecha

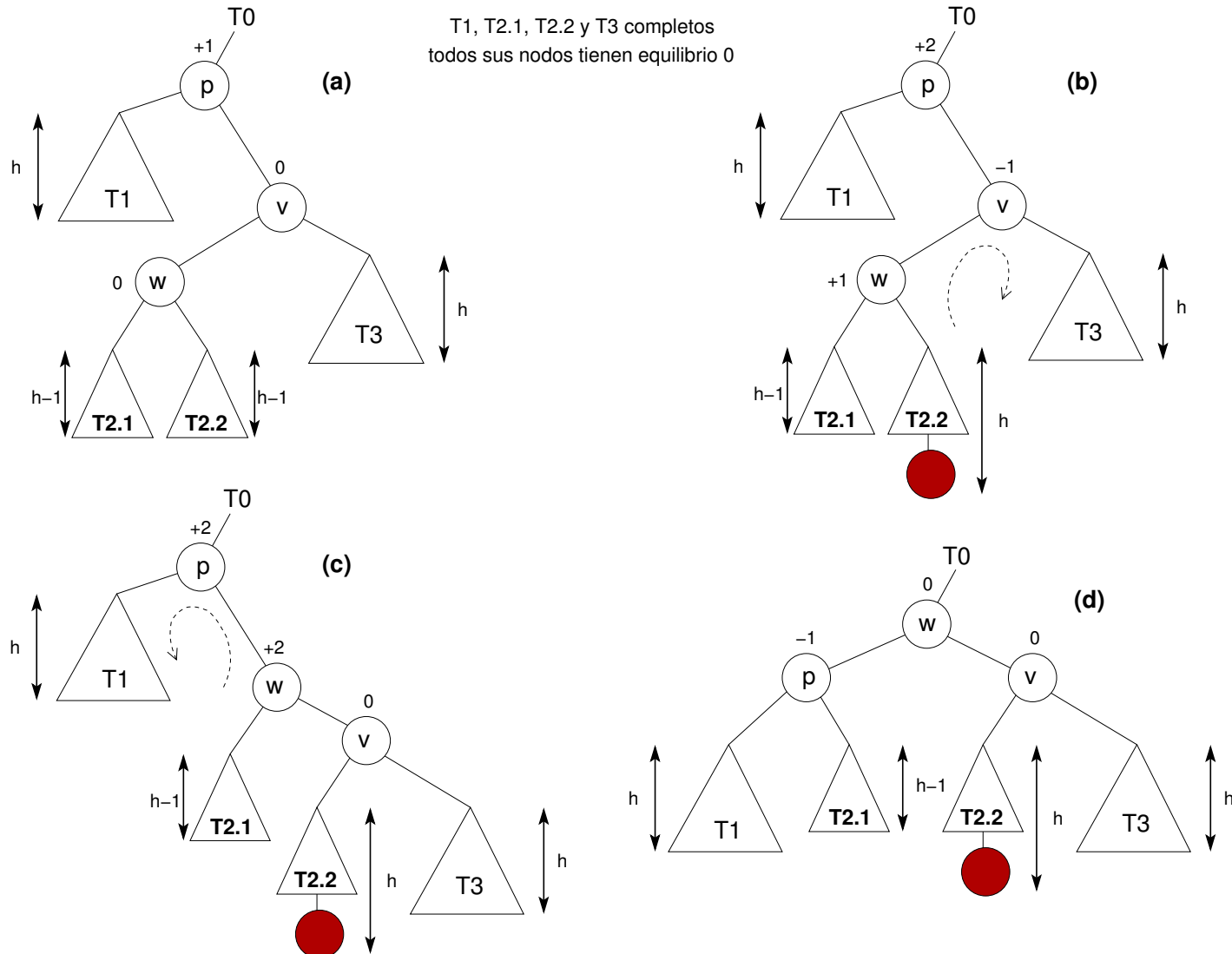


# Arboles AVL: Inserción



## Rotación doble derecha-izquierda

T1, T2.1, T2.2 y T3 completos  
todos sus nodos tienen equilibrio 0

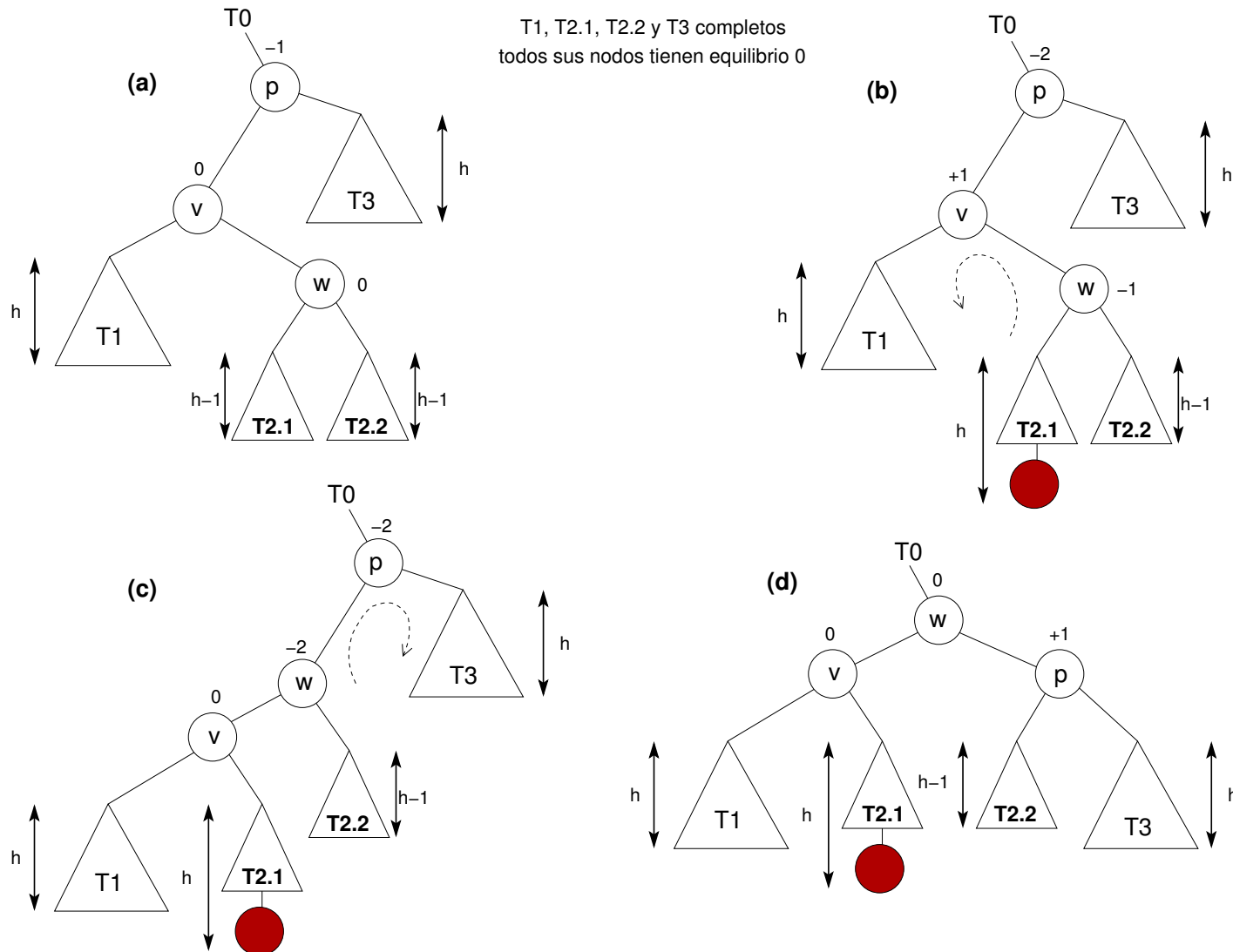


# Arboles AVL: Inserción



## Rotación doble izquierda-derecha

T1, T2.1, T2.2 y T3 completos  
todos sus nodos tienen equilibrio 0



# Arboles AVL: Inserción

## Cómo elegir el tipo de rotación tras una inserción en un árbol AVL

- ▷ Si el pivote potencial (con equilibrio  $+1$  o  $-1$ ) no cambia a  $+2$  o  $-2$ : no es necesario modificar la estructura del árbol
- ▷ Si el pivote cambia de  $+1$  a  $+2$  y su hijo derecho cambia de  $0$  a  $+1$ : rotación simple a izquierda
- ▷ Si el pivote cambia de  $+1$  a  $+2$  y su hijo derecho cambia de  $0$  a  $-1$ : rotación doble derecha-izquierda
- ▷ Si el pivote cambia de  $-1$  a  $-2$  y su hijo izquierdo cambia de  $0$  a  $-1$ : rotación simple a derecha
- ▷ Si el pivote cambia de  $-1$  a  $-2$  y su hijo izquierdo cambia de  $0$  a  $+1$ : rotación doble izquierda-derecha

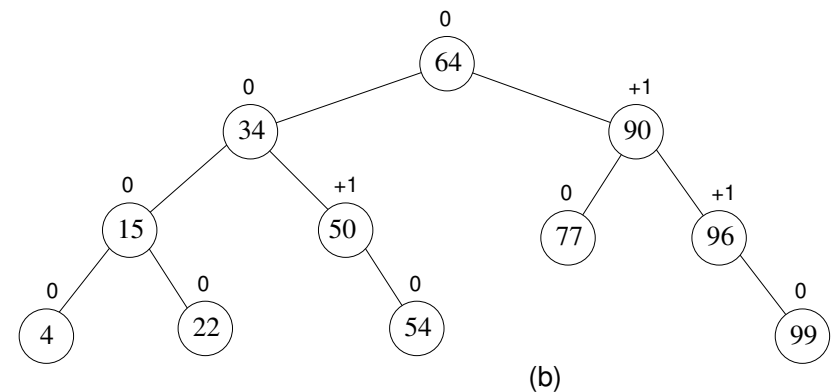
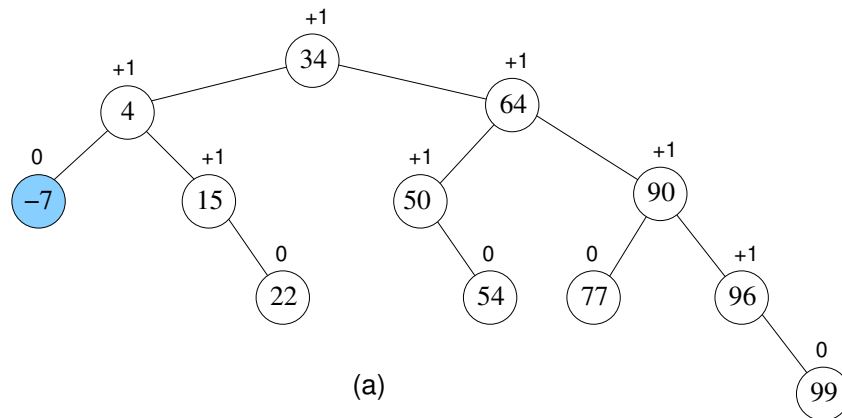
# Arboles AVL: Borrado

- ▶ Para eliminar un nodo en un árbol AVL, se aplica el algoritmo genérico de borrado definido para ABB. El coste de esta operación está en  $O(\log n)$ .
- ▶ A continuación se recorre el camino de vuelta desde el padre del nodo eliminado hacia la raíz y se actualizan los equilibrios.
- ▶ El camino de vuelta podría llegar hasta la raíz sin que se produjeran desequilibrios, por lo que no sería necesario modificar la estructura del árbol. Nótese que el coste de vuelta estaría en  $O(\log n)$ .
- ▶ Si al actualizar el equilibrio de un nodo, éste pasa de +1 a +2 o de -1 a -2, deberá aplicarse al menos una rotación simple o doble para recuperar la propiedad de los árboles AVL.
- ▶ El reequilibrado de un subárbol AVL tras una operación de borrado no conserva la altura, por lo que podría ser necesario aplicar varias rotaciones en el camino de vuelta hacia la raíz.

# Arboles AVL: Borrado



- ▶ En el caso peor, al eliminar una de las hojas menos profundas del árbol, podría tener que realizarse una rotación en cada nodo del camino hacia la raíz.
- ▶ Aún así, como el coste de una rotación está en  $\Theta(1)$  y la longitud del camino está en  $\Theta(\log n)$ , el coste de borrado en el peor caso estará también en  $\Theta(\log n)$ .
- ▶ Ejemplo (2 rotaciones a izquierda sucesivas, sobre los nodos 4 y 34):

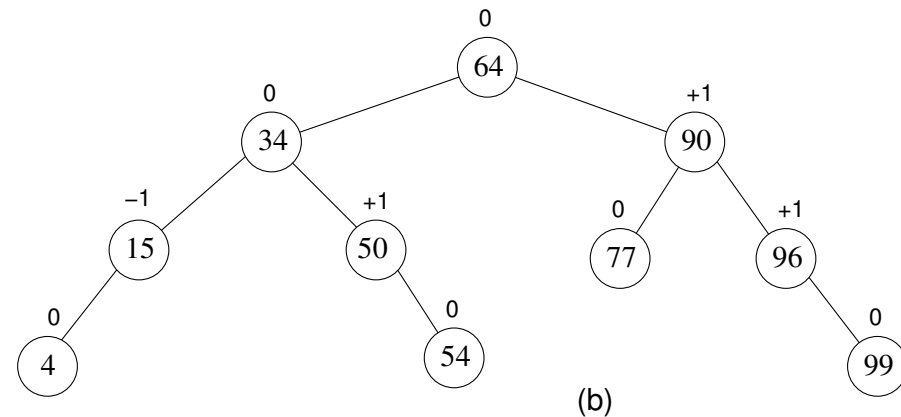
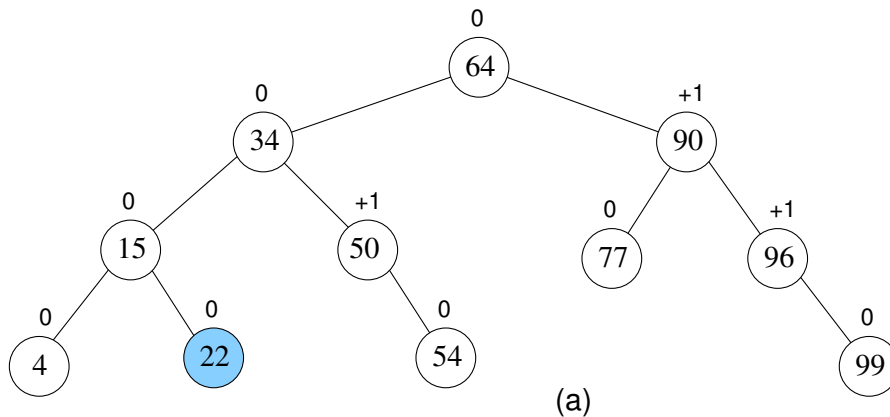




# Arboles AVL: Borrado



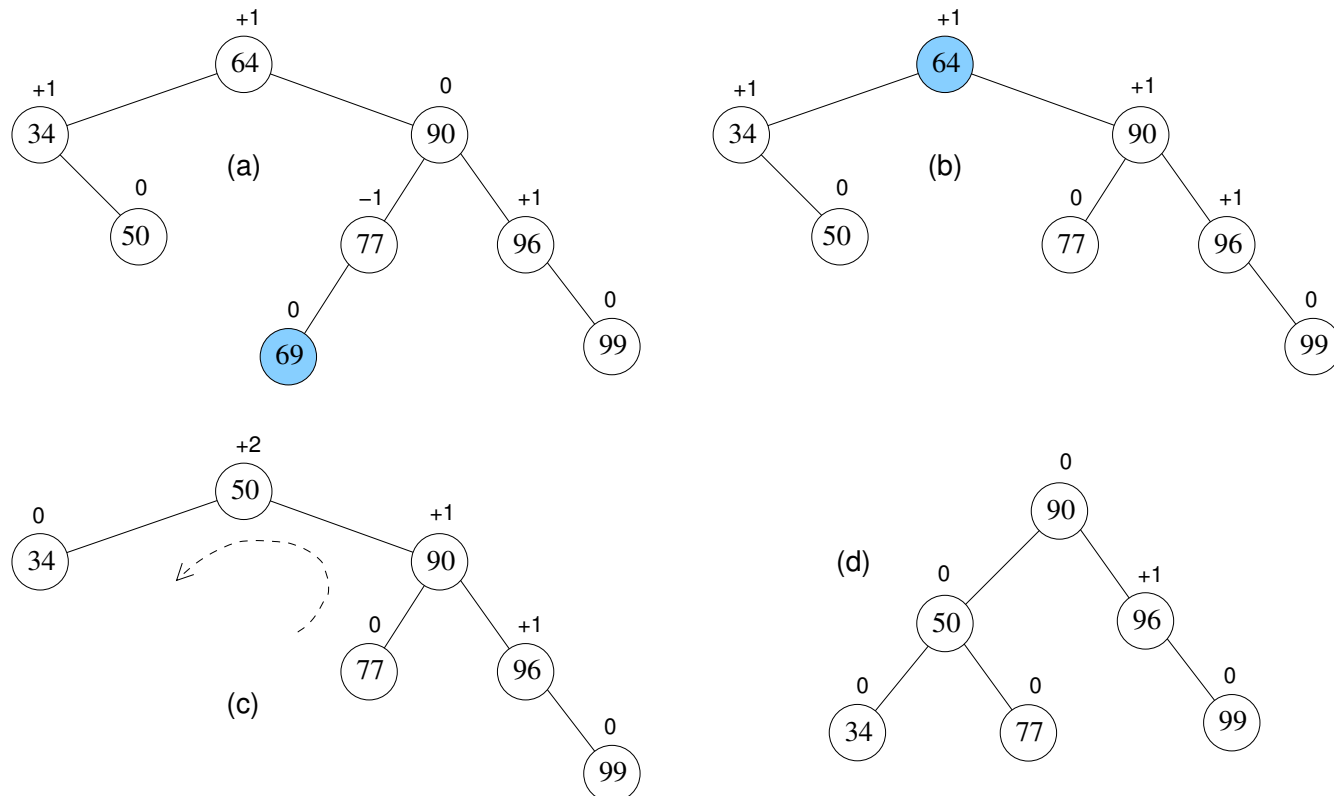
- ▷ Si el padre del nodo eliminado (o cualquier otro en la vuelta hacia el nodo raíz) pasa de equilibrio 0 a +1 o -1, no es necesario seguir explorando el árbol, porque la altura del subárbol correspondiente no ha cambiado, lo que significa que la propiedad de los árboles AVL se sigue cumpliendo en el resto de la estructura.
- ▷ Ejemplo:



# Arboles AVL: Borrado



- ▷ Si el padre del nodo eliminado pasa de +1 o -1 a 0, la altura del subárbol correspondiente se ve modificada, por lo que cambiará el equilibrio del nodo abuelo, y quizá también el de nodos superiores.
- ▷ Esto podría implicar ninguna, una o varias rotaciones, dependiendo del estado del árbol. Por ejemplo:



# Arboles AVL: Borrado

*En resumen: ¿cómo se gestiona el borrado de un elemento en un árbol AVL?*

- ▷ Si el equilibrio del padre pasa de 0 a +1 o -1: el algoritmo termina
- ▷ Si el equilibrio del padre pasa de +1 o -1 a 0: continuar reequilibrando nodos hacia el nodo raíz
- ▷ Si el equilibrio del padre pasa de +1 a +2:
  - ▷ Si el equilibrio del hijo derecho es -1: rotación doble derecha-izquierda y continuar reequilibrando nodos hacia el nodo raíz
  - ▷ Si el equilibrio del hijo derecho es 0: rotación simple a izquierda y el algoritmo termina
  - ▷ Si el equilibrio del hijo derecho es +1: rotación simple a izquierda y continuar reequilibrando nodos hacia el nodo raíz
- ▷ Si el equilibrio del padre pasa de -1 a -2:
  - ▷ Si el equilibrio del hijo izquierdo es +1: rotación doble izquierda-derecha y continuar reequilibrando nodos hacia el nodo raíz
  - ▷ Si el equilibrio del hijo izquierdo es 0: rotación simple a derecha y el algoritmo termina
  - ▷ Si el equilibrio del hijo izquierdo es -1: rotación simple a derecha y continuar reequilibrando nodos hacia el nodo raíz

# TAD ABB: Ejercicios (1)

- ▷ Demostrar que en un árbol binario de  $n$  nodos hay  $n + 1$  hijos nulos.
- ▷ Demostrar que el número máximo de nodos en un árbol binario de altura  $h$  es  $2^{h+1} - 1$ .
- ▷ Demostrar que el número de hojas de un árbol binario relleno es igual al número de nodos internos más 1.
- ▷ Mostrar el resultado de insertar los valores 3, 1, 4, 6, 9, 2, 5 y 7 en un ABB inicialmente vacío. Repetir la operación insertando los valores en orden creciente. Repetir de nuevo la operación insertándolos en el orden siguiente: 5, 2, 4, 3, 7, 9, 6, 1. ¿Cuál es la probabilidad de obtener este último ABB a partir de una permutación aleatoria de los valores enumerados? ¿Cuál es la altura promedio del árbol binario obtenido a partir de una permutación aleatoria de dichos valores? Escribir en lenguaje C una función que calcule la altura promedio del ABB obtenido a partir de  $n$  valores distintos, considerando para ello todas las permutaciones posibles.

## TAD ABB: Ejercicios (2)

- ▷ En un ABB con  $n$  nodos se tienen  $n + 1$  referencias nulas a hijos. Para aprovechar ese espacio no utilizado, convenimos en que si el hijo izquierdo de un nodo  $v$  es nulo, entonces hacemos que dicho hijo apunte al predecesor de  $v$  en un recorrido simétrico del ABB. Análogamente, si el hijo derecho de  $v$  es nulo, hacemos que apunte al sucesor de  $v$  en un recorrido simétrico. Los ABB que resultan reciben el nombre de **árboles enhebrados** y las referencias adicionales se llaman **hebras**.
  - ▷ ¿Cómo pueden distinguirse las hebras de las referencias reales a hijos?
  - ▷ Modifíquese la implementación del TAD ABB para que represente ABB enhebrados.
  - ▷ ¿Cuál es la ventaja de los ABB enhebrados?

# TAD ABB: Ejercicios (3)

- ▷ Escribir en lenguaje algorítmico la operación *buscar\_k\_ésimo()*, que se añadirá al repertorio de operaciones del TAD ABB. La operación *buscar\_k\_ésimo(t,i)* devuelve el nodo con la *i*-ésima clave más pequeña. Suponiendo que todos los elementos del ABB tienen claves distintas, modifíquese la implementación del TAD ABB para que esta operación pueda realizarse en tiempo  $O(\log n)$ , sin que el coste de las demás operaciones se vea afectado.
- ▷ Escribir una función que tome como entrada un ABB  $t$  y dos claves  $k_1$  y  $k_2$  ( $k_1 \leq k_2$ ), e imprima la secuencia de elementos  $x$  tales que  $k_1 \leq \text{clave}(x) \leq k_2$ . La función ha de tener complejidad  $O(K + \log n)$ , donde  $K$  es el número de elementos que verifican la condición y  $n$  el número de nodos del ABB.

## TAD ABB: Ejercicios (4)

- ▷ Considerando la implementación básica del TAD ABB, escríbanse sendos procedimientos para efectuar bien una rotación a izquierda, bien una rotación a derecha sobre un nodo  $v$  de un ABB  $t$ .
- ▷ Partiendo de la implementación básica del TAD ABB, escríbase una implementación del TAD AVL. Nótese que tan sólo será necesario modificar la definición de los tipos y añadir dos nuevas operaciones, *añadir\_AVL()* y *eliminar\_AVL()*, que harán uso de las operaciones básicas del TAD ABB, así como de los procedimientos de rotación escritos previamente (que también formarán parte del TAD, como operaciones privadas).