

STRUCTURAS DE DATOS TEORÍA 2015/2016

ÁBOLES AVL Y MONTÍCULOS

J. Lázaró

M. J. Domínguez

ÁRBOLES AVL

Los **árboles AVL** (denominados así por las iniciales de sus creadores, G.M. Adelson-Velskii y E.M. Landis) son un tipo especial de árboles binarios de búsqueda.

- Se caracterizan porque están balanceados¹: las alturas de los hijos izquierdo y derecho de un nodo se diferencian en 1 como máximo.
- Permiten las operaciones típicas de los árboles de búsqueda binarios: *está?*, *insertar* y *borrar*.
- Las operaciones se realizan de igual manera que en los árboles de búsqueda binarios.

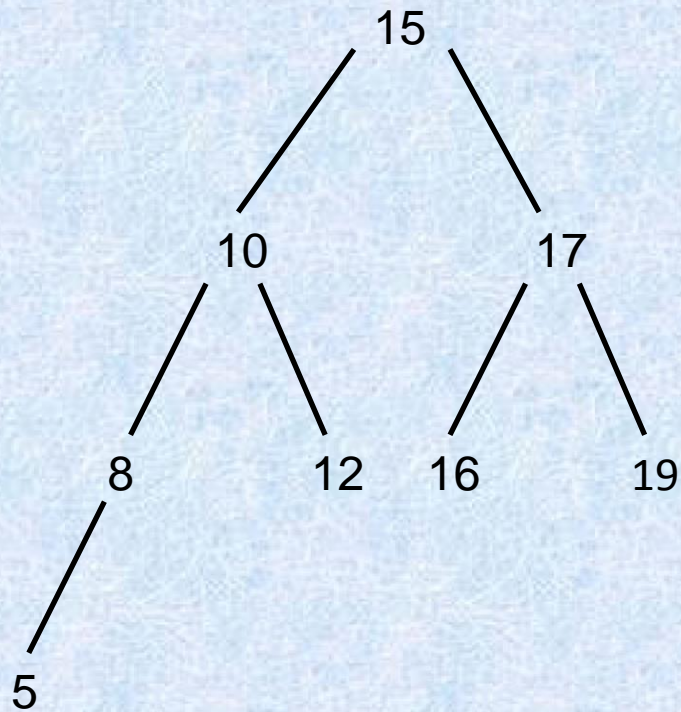
¹ Se denomina **factor de balanceo** a la diferencia de altura entre el hijo derecho y el izquierdo.

ÁRBOLES AVL: INSERCIÓN (1)

Cuando se inserta un nodo se puede desequilibrar el árbol, y hay que rebalancearlo.

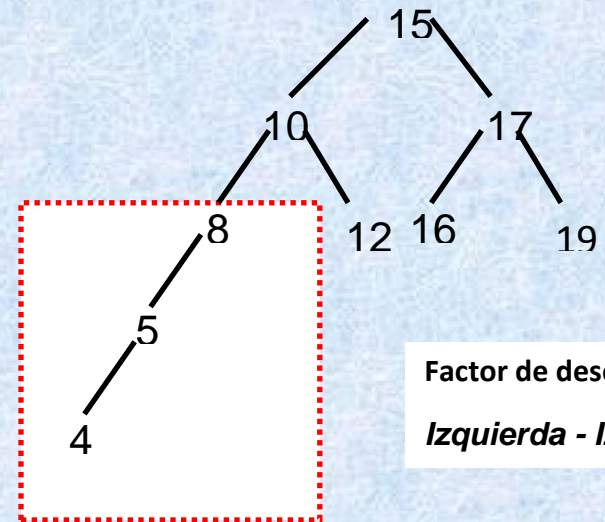
- Se tiene que comprobar cuál es el factor de balanceo de los antecesores del último nodo insertado.
- Si el factor de desequilibrio (diferencia entre el factor de balanceo de las ramas derecha e izquierda) es 0, 1 o -1 no es necesario balancear el árbol.
- Si el factor de desequilibrio es -2 se distinguen dos casos:
 - Izquierda – Izquierda
 - Izquierda – Derecha
- Si el factor de desequilibrio es $+2$ se distinguen dos casos:
 - Derecha – Derecha
 - Derecha – Izquierda

ÁRBOLES AVL: INSERCIÓN (2)



Factor de desequilibrio: -1

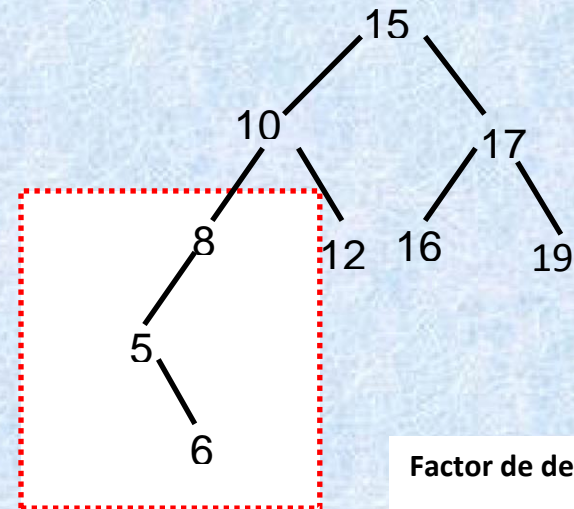
Insertar 4



Factor de desequilibrio: -2

Izquierda - Izquierda

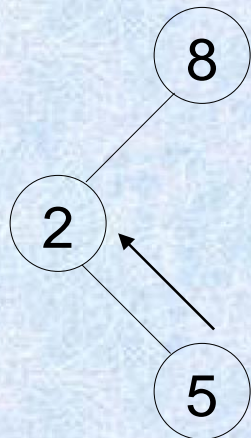
Insertar 6



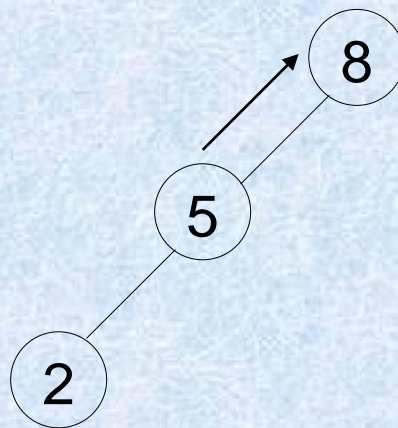
Factor de desequilibrio: -2

Izquierda - derecha

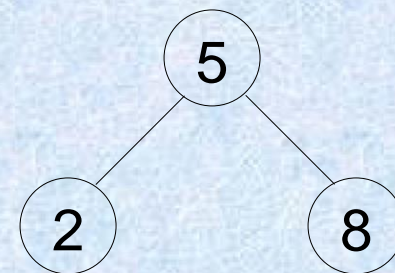
ÁRBOLES AVL: INSERCIÓN (3)



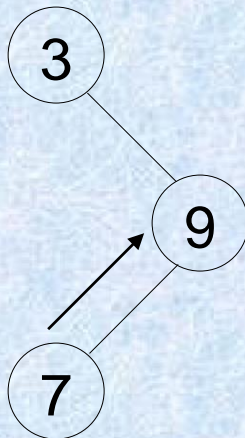
Izquierda - derecha



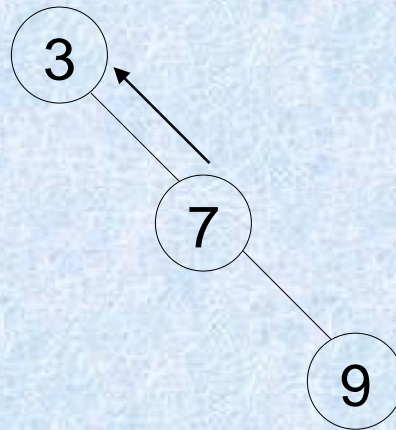
Izquierda - izquierda



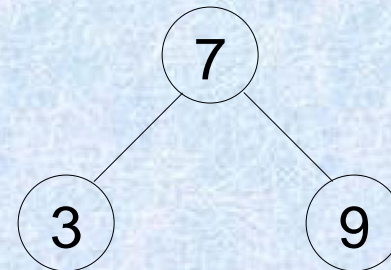
Balanceado



Derecha - izquierda



Derecha - derecha



Balanceado

ÁRBOLES AVL: BORRADO

A la hora de borrar un nodo hay distintos casos. Si el nodo es una hoja o tiene un único hijo se elimina directamente, pero si tiene dos hijos el nodo puede reemplazarse por uno de estos elementos

- El mayor de la rama izquierda (predecesor en inorden)
- El menor de la rama derecha (sucesor en inorden)

Para mantener el árbol como árbol de búsqueda según lo vemos en la asignatura (donde los elementos iguales se encuentran a la izquierda de la raíz), el nodo borrado se sustituirá por el **elemento mayor del hijo izquierdo**.

Puede ser necesario reajustar el balanceo, empezando por el lugar original del nodo sustituto y subiendo hacia la raíz.

ÁRBOLES AVL: TIPOS

tipos

```
nodo_avl = reg  
    altura:natural  
    valor: elemento  
    izq: a_avl  
    der: a_avl
```

freg

```
a_avl = puntero a nodo_avl
```

ftipos

ÁRBOLES AVL¹. CONSTRUCTORAS

{Crear un árbol binario a partir de un elemento y dos árboles binarios _._._}

```
func crea_árbol(e:elemento,hi,hd:a_avl) dev a:a_avl
var aux: nodo_avl
    reservar (aux)
    aux^.valor←e
    aux^.izq←hi
    aux^.der←hd
    a ← aux
    actualizar_altura(a)
finfunc
```

¹Las operaciones crear árbol vacío y las observadoras raíz, izquierdo, derecho y vacío? no cambian respecto a las definidas para árboles binarios.

ÁRBOLES AVL. OBSERVADORAS

{Calcular la altura del árbol binario}

```
func altura (a:a_avl) dev natural
  si vacio?(a) entonces error(Árbol vacío)
  si no
    devolver a^.altura
  finsi
finfunc
```


ÁRBOLES AVL. MODIFICADORAS

```
proc actualizar_altura (a:avl)
  si (!es_vacio(a)) entonces
    si vacio?(a^.izq) entonces
      si vacio?(a^.der) entonces a^.altura ← 0
      si no a^.altura ← 1 + altura(a^.der)
    finsi
  si no
    si vacio?(a^.der) entonces
      a^.altura ← 1 + altura(a^.izq)
    si no
      a^.altura ← 1 + max(altura(a^.izq), altura(a^.der))
    finsi
  finsi
finsi
finproc
```


ÁRBOLES AVL. MODIFICADORAS (2)

{giro simple:izq_izq o der_der, según giroizq sea T o F}

proc rotación_simple (a:avl, giroizq:bool)

var aaux:avl

si giroizq **entonces**

 aaux ← a^.izq

 a^.izq ← aaux^.der

 aaux^.der ← a

si no

 aaux ← a^.der

 a^.der ← aaux^.izq

 aaux^.izq ← a

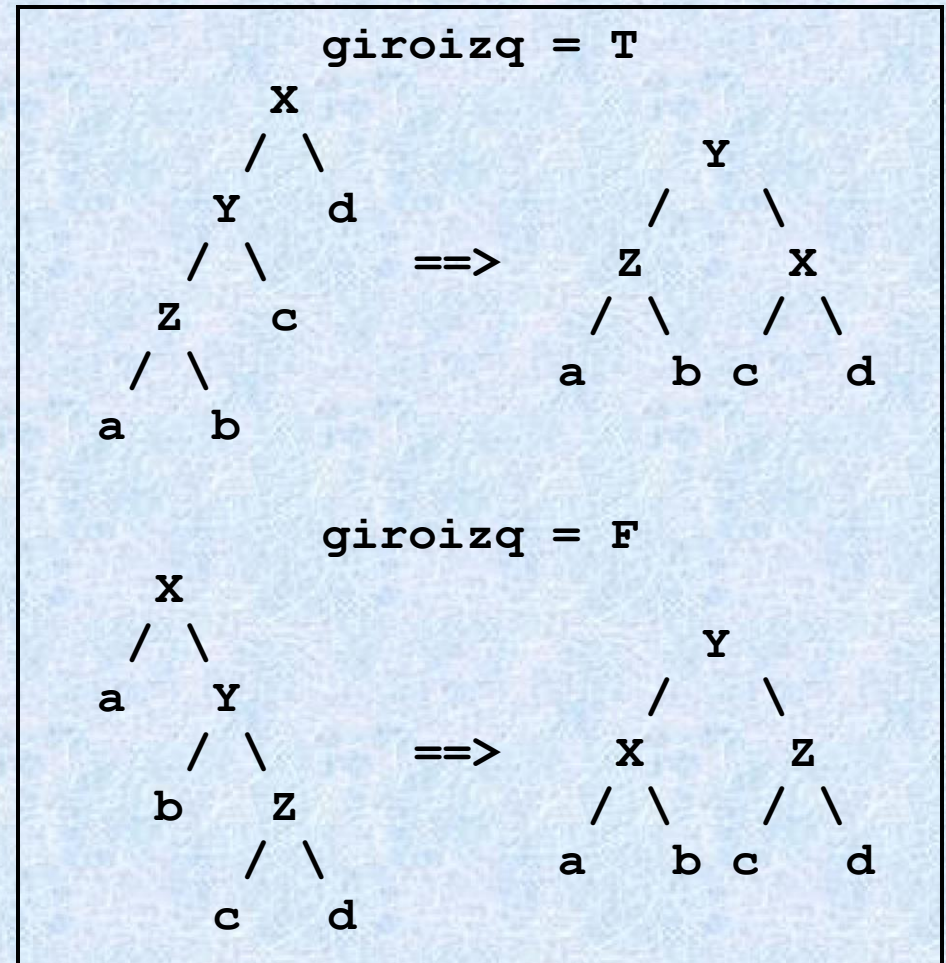
finsi

 actualizar_altura(a)

 actualizar_altura(aaux)

 a ← aaux

finproc



ÁRBOLES AVL. MODIFICADORAS (3)

```
proc rotación_doble(a:avl, giroizq:bool)
```

```
  {la actualización de las  
   alturas se realiza en las  
   rotaciones simples}
```

```
si giroizq entonces
```

```
  {rotación izq-der}
```

```
  rotación_simple(a^.izq, F)
```

```
  rotación_simple(a, T)
```

```
si no
```

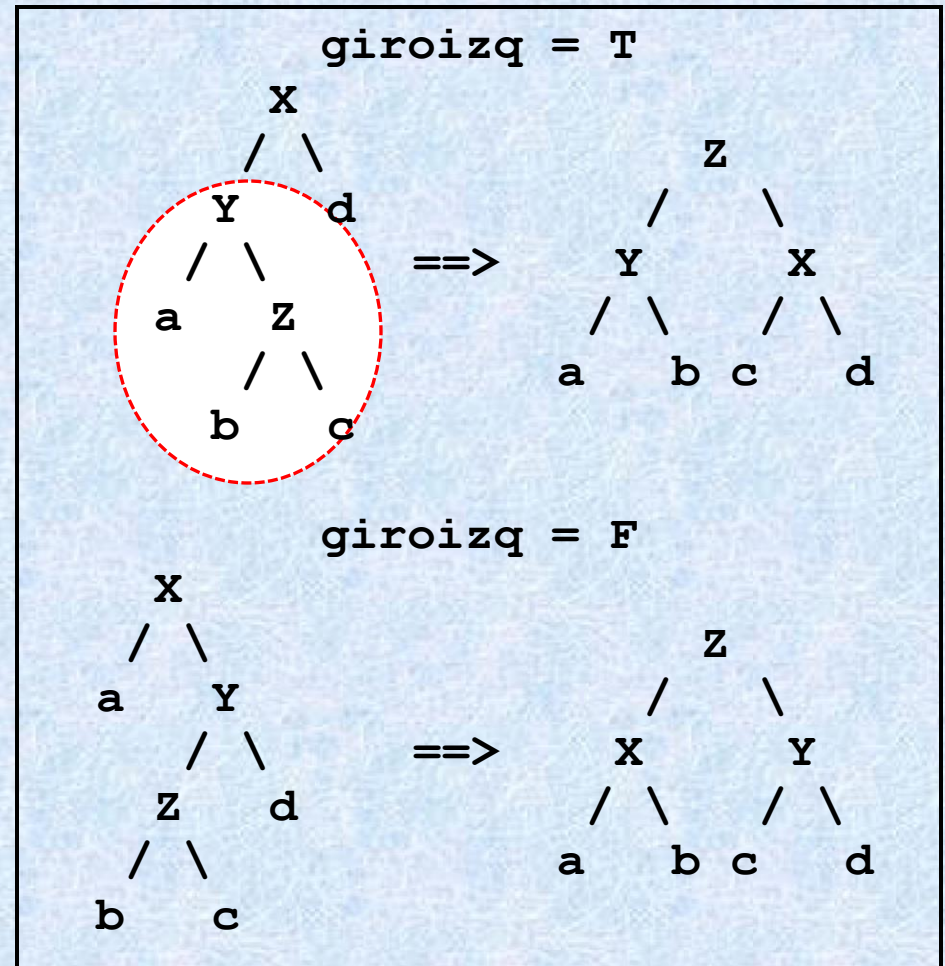
```
  {rotación der-izq}
```

```
  rotación_simple(a^.der, T)
```

```
  rotación_simple(a, F)
```

```
finsi
```

```
finproc
```



ÁRBOLES AVL. MODIFICADORAS (4)

proc balancear (a:avl)

{Detecta y corrige por medio de un número finito de rotaciones un desequilibrio en el árbol "a". Dicho desequilibrio no debe tener una diferencia de alturas de más de 2}

si !vacio?(a) **entonces**

si (altura(a^.der) - altura(a^.izq) = -2)

entonces {desequilibrio hacia la izquierda!}

si (altura(a^.izq^.izq) >= altura(a^.izq^.der))¹

entonces {desequilibrio simple izq_izq}

rotación_simple(a, true)

si no {desequilibrio doble izq_der}

rotación_doble(a, true)

finsi

¹ Comprobamos si el hijo izquierdo está desequilibrado hacia la izquierda (**desequilibrio simple izq-izq**) o hacia la derecha (**desequilibrio doble izq-der**).

ÁRBOLES AVL. MODIFICADORAS (5)

```
si no {desequilibrio hacia la derecha!}
  si (altura(a^.der) - altura(a^.izq) = 2)
    entonces {desequilibrio hacia la derecha!}
      si (altura(a^.der^.der) >= altura(a^.der^.izq))
        entonces {desequilibrio simple der_der}
          rotación_simple(a, false)
      si no {desequilibrio doble der_izq}
        rotación_doble(a, false)
    fin si
  fin si
fin si
fin si
finproc
```


ÁRBOLES AVL. MODIFICADORAS (6)

{insertar un dato en un árbol AVL}

```
proc insertar1 (e:elemento, a:avl)
  si vacio?(a) entonces a ← crea_árbol(e, nil, nil)
  si no
    si e ≤ (a^.valor) entonces
      insertar (e, a^.izq)
    si no
      insertar (e, a^.der)
    finsi
    balancear(a)
    actualizar_altura(a)
  finsi
finproc
```

¹ El procedimiento es el mismo que el de insertar en un árbol binario de búsqueda, excepto la llamada a balancear y actualizar_altura una vez insertado el nodo.

ÁRBOLES AVL. MODIFICADORAS (7)

{borrar un dato en un árbol AVL}

```
proc borrar1 (e:elemento, a:avl)
  si !vacio?(a) entonces
    si (a^.valor=e) entonces {e encontrado}
      si (a^.izq=nil) ^ (a^.der=nil) entonces {es hoja}
        a←nil
      si no
        si (a^.izq=nil) entonces a←a^.der
        si no
          si (a^.der=nil) entonces a←a^.izq
          si no max ← máximo(a^.izq)
            borrar(max, a^.izq) {repite la búsqueda}
            a^.valor←max
          finsi
        finsi
      finsi
    (...)
  finsi
```

¹ El procedimiento es el mismo que el de borrar en un árbol binario de búsqueda, excepto la llamada a balancear y actualizar_altura una vez eliminado el nodo

ÁRBOLES AVL. MODIFICADORAS (8)

{borrar un dato en un árbol AVL}

(...)

si no *{se sigue buscando "e" en algún hijo.}*

si $(e < a^{\text{valor}})$ **entonces**

borrar(e, a^{izq})

si no *{en este caso, se sabe que $(e > a^{\text{valor}})$ }*

borrar(e, a^{der})

finsi

finsi

balancear(a) {tras el borrado, se balancea el árbol...}

actualizar_altura(a) {... y se actualiza la altura}

finsi *{del caso !vacío?(a)}*

finproc

MONTÍCULOS

Un **montículo** (de mínimos) es un árbol binario semicompleto¹ de elementos ordenables que verifica

- El árbol está vacío, o
- El elemento de la raíz es **menor o igual** que el resto de elementos en el árbol, y los hijos son también montículos.

También puede definirse un montículo de máximos si la relación entre los elementos es de *mayor o igual*.

Los montículos permiten las operaciones de *insertar_mínimo* y *eliminar_mínimo*.

¹ Un árbol binario es **completo** si todas sus hojas están en el mismo nivel, y es **semicompleto** si es completo o si las hojas que le faltan están consecutivas (empezando por la derecha).

ESPECIFICACIÓN: ÁRBOLES COMPLETOS

espec *ÁRBOLES_BINARIOS_SEMICOMPLETOS[ELEMENTO \leq]*

usa *ÁRBOLES_BINARIOS[ELEMENTO]*

parametro formal

generos *elemento*

operaciones

$_ \leq _ : \textit{elemento elemento} \rightarrow \textit{bool}$

fparametro

ESPECIFICACIÓN: ÁRBOLES COMPLETOS (2)

operaciones

es_completo? : a_bin \rightarrow bool

es_semicompleto? : a_bin \rightarrow bool

es_montículo? : a_bin \rightarrow bool

operaciones auxiliares

niveles : a_bin \rightarrow natural {cantidad de niveles del árbol}

menor_igual? : elemento a_bin \rightarrow bool

var

x, y: elemento

i, d: a_bin

ESPECIFICACIÓN: ÁRBOLES COMPLETOS (3)

ecuaciones

$$\text{niveles}(\Delta) = 0$$

$$\text{niveles}(i \bullet x \bullet d) = \text{suc}(\text{altura}(i \bullet x \bullet d))$$

$$\text{es_completo?}(\Delta) = T$$

$$\text{es_completo?}(i \bullet x \bullet d) =$$

$$\begin{aligned} &\text{es_completo?}(i) \wedge \text{es_completo?}(d) \wedge \\ &\quad (\text{niveles}(i) == \text{niveles}(d)) \end{aligned}$$

ESPECIFICACIÓN: ÁRBOLES COMPLETOS (4)

$es_semicompleto?(\Delta) = T$

$es_semicompleto?(i \bullet x \bullet d) =$

$(es_completo?(i \bullet x \bullet d))$

\vee

$(es_completo?(i) \wedge es_semicompleto?(d)$

$\wedge (niveles(i) == niveles(d)))$

\vee

$(es_semicompleto?(i) \wedge es_completo?(d)$

$\wedge (niveles(i) == niveles(d)+1))$

ESPECIFICACIÓN: ÁRBOLES COMPLETOS (y 5)

$menor_igual?(y, \Delta) = T$

$menor_igual?(y, i \bullet x \bullet d) = (y \leq x)$

$\wedge menor_igual?(y, i) \wedge menor_igual?(y, d)$

$es_montículo?(\Delta) = T$

$es_montículo?(i \bullet x \bullet d) = es_semicompleto?(i \bullet x \bullet d)$

$\wedge menor_igual?(x, i) \wedge menor_igual?(x, d)$

$\wedge es_montículo?(i) \wedge es_montículo?(d)$

fespec

IMPLEMENTACIÓN: MONTÍCULOS

Un árbol binario completo de altura $h \geq 0$ tiene

- 2^i nodos de nivel i , para $0 \leq i \leq h$,
- 2^h hojas (que son los nodos de nivel h),
- $2^{h+1} - 1$ nodos en total.

Un árbol binario semicompleto de n nodos tiene una altura igual a $\log_2(n)$, redondeando hacia abajo.

IMPLEMENTACIÓN: MONTÍCULOS (2)

Un árbol binario semicompleto de altura $h \geq 0$ puede almacenarse en un vector de tamaño $2^{h+1} - 1$ como sigue:

- La raíz del árbol ocupa la primera posición del vector.
- Un nodo almacenado en la posición i -ésima tiene a su hijo izquierdo en la posición $2 \cdot i$, y a su hijo derecho en $2 \cdot i + 1$.
- Los nodos de profundidad k del árbol se almacenan leídos de izquierda a derecha en: $T[2^k], T[2^k+1], \dots, T[2^{k+1}-1]$.
- El padre del nodo almacenado en $T[i]$ está en la posición igual a " $i \text{ div } 2$ ".

IMPLEMENTACIÓN: MONTÍCULOS (3)

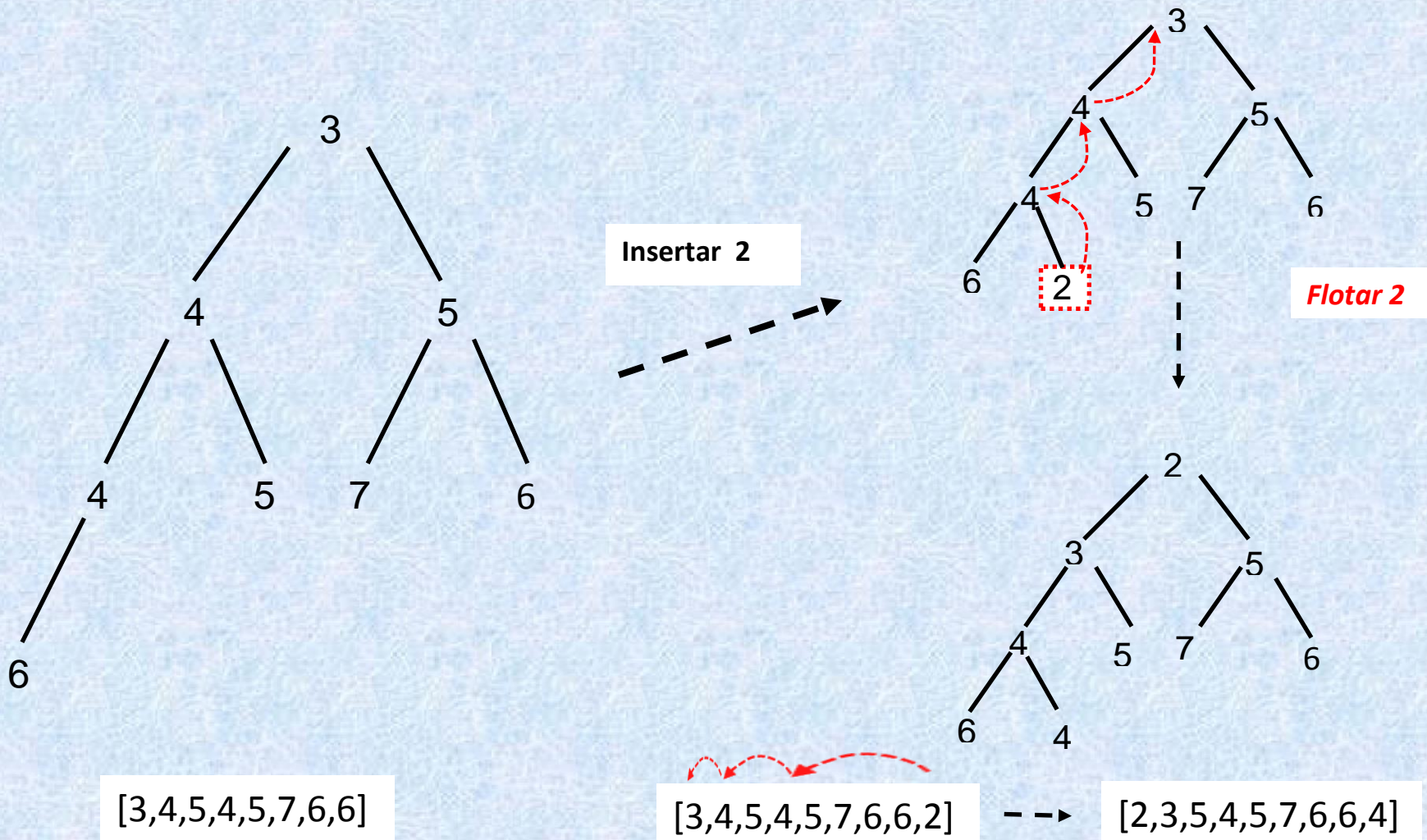
Cuando se inserta un elemento, hay que **flotar** dicho elemento si no se encuentra en su posición correcta:

- el elemento se añade en la última posición libre, y
- se intercambia con su padre hasta que llegue a su posición correcta.

Cuando se elimina un elemento (que siempre es el menor por construcción del tipo), hay que **hundir** el último elemento para colocarlo en su posición correcta:

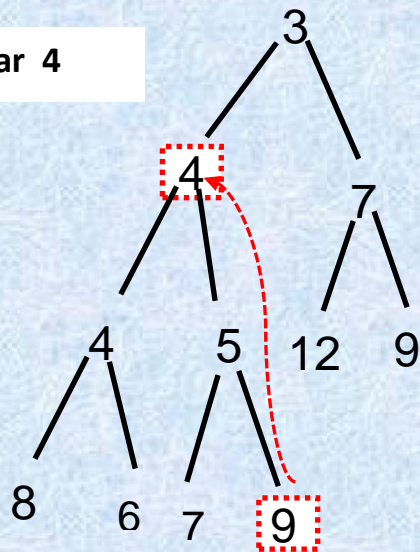
- se coloca el último elemento en la primera posición, y
- si es necesario se intercambia con el menor de los hijos mientras sea necesario.

IMPLEMENTACIÓN: MONTÍCULOS (4)

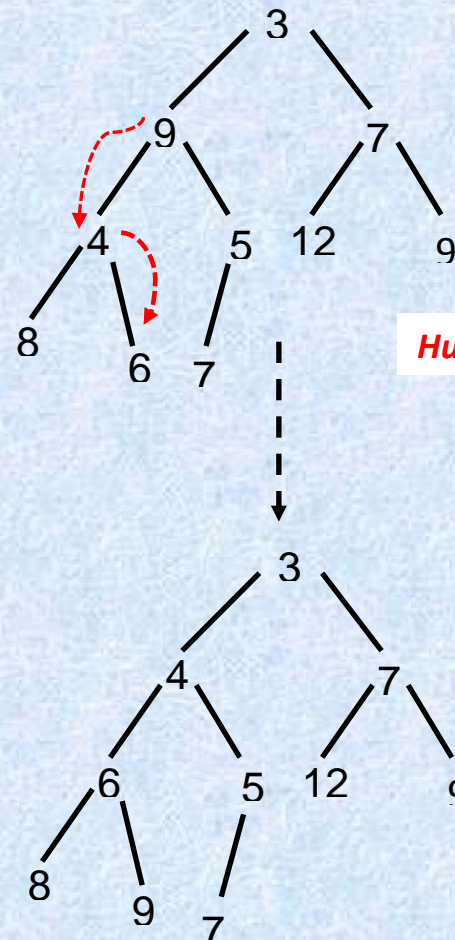


IMPLEMENTACIÓN: MONTÍCULOS (5)

Borrar 4



Subir 9



Hundir 9

[3,4,7,4,5,12,9,8,6,7,9]

→

[3,9,7,4,5,12,9,8,6,7]

→

[3,4,7,6,5,12,9,8,9,7]

IMPLEMENTACIÓN: MONTÍCULOS (6)

constante

máximo = ...

fconstante

tipos

montículo = **reg**

tamaño : 0..máximo

datos : vector[1..máximo] de tipo_elemento

freg

ftipos

MONTÍCULOS: OPERACIONES

```
proc inicializar_montículo(m: montículo)
```

```
    m.tamaño ← 0
```

```
finproc
```

```
func vacio?(m: montículo) dev b:bool
```

```
    b ← (m.tamaño = 0)
```

```
finfunc
```


MONTÍCULOS: OPERACIONES (2)

```
proc flotar (m:montículo, i:1..tamaño)
  mientras (i > 1)  $\wedge$  (m.datos[i div 2] > m.datos[i])
    {mientras el padre es mayor que el hijo}
  hacer
    intercambiar(m.datos[i div 2], m.datos[i])
                                {lo intercambiamos...}
    i  $\leftarrow$  i div 2           {... y continuamos con el padre}
  finmientras
finproc
```


MONTÍCULOS: OPERACIONES (3)

```
proc insertar(x: tipo_elemento, m: montículo)
  si (m.tamaño = máximo) entonces
    error('montículo lleno')
  si no
    m.tamaño  $\leftarrow$  m.tamaño + 1
    m.datos[m.tamaño]  $\leftarrow$  x
    {insertamos en la primera posición libre del montículo}
    flotar(m, m.tamaño)
    {flota hacia arriba hasta su posición en el montículo}
  finsi
finproc
```


MONTÍCULOS: OPERACIONES (4)

proc hundir (m:montículo, i:1..tamaño)

var hijoIzq, hijoDer: 1..máximo

repetir

hijoIzq \leftarrow 2*i

hijoDer \leftarrow 2*i+1

j \leftarrow i

{buscamos el dato a intercambiar con el i-ésimo, el menor entre sus hijos si ambos son menores que él}

si (hijoDer \leq m.tamaño) \wedge

(m.datos[hijoDer] < m.datos[i]) **entonces**

i \leftarrow hijoDer

finsi

(...)

MONTÍCULOS: OPERACIONES (4)

(...) *{hundir}*

```
si (hijoIzq <= m.tamaño ) ^  
    (m.datos[hijoIzq] < m.datos[i]) entonces  
    i ← hijoIzq
```

finsi

```
si (i≠j) entonces
```

```
    intercambiar (m.datos[j], m.datos[i])
```

finsi

```
hasta j=i      {si j=i el nodo alcanzó su posición final,  
                ninguno de sus hijos es menor}
```

finproc

MONTÍCULOS: OPERACIONES (5)

```
func eliminarmin(m: montículo) dev x: tipo_elemento
    si vacío?(m) entonces error('montículo vacío)
    si no
        x ← m.datos[1]
        m.datos [1] ← m.datos [m.tamaño]
        {ponemos el último dato en lugar del mínimo}
        m.tamaño ← m.tamaño - 1;
        si (m.tamaño > 0) entonces
            hundir(m, 1) {lo hundimos hasta su posición}
        finsi
    finsi
finfunc
```