ESTRUCTURAS DE DATOS

ÁBOLES AVL Y MONTÍCULOS

Profesora: Mª José Domínguez Alda

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.

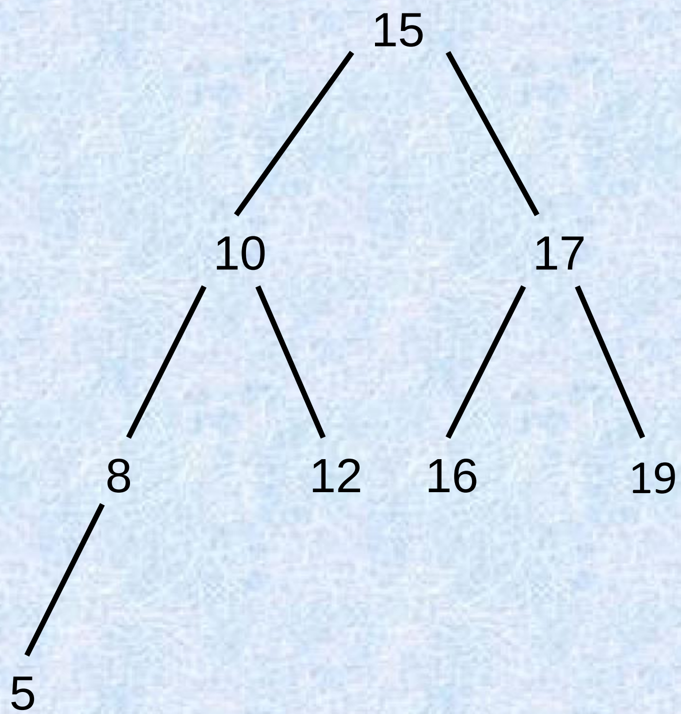
- Se caracterizan porque están balanceados: las alturas de los hijos izquierdo y derecho de un nodo se diferencian en 1 como máximo.
- Permiten las operaciones típicas de los árboles de búsqueda binarios: ***`está?`***, ***`insertar`*** y ***`borrar`***.
- Las operaciones se realizan de igual manera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

ÁRBOLES AVL: INSERCIÓN (1)

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

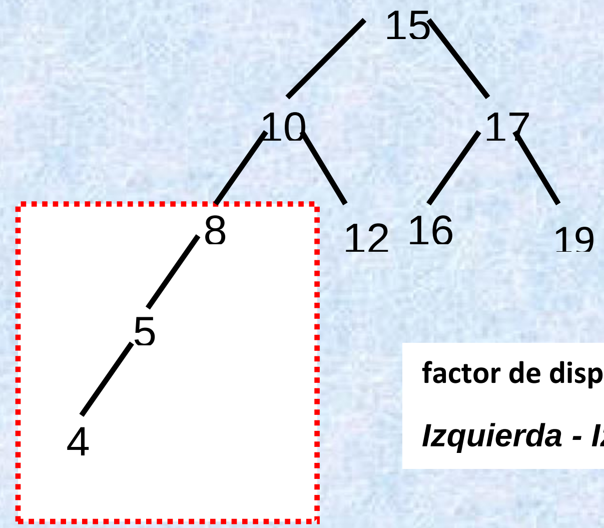
- Se tiene que comprobar cuál es el factor de balanceo de los antecesores del último nodo insertado.
- Si el factor de desequilibrio (diferencia entre el factor de balanceo de las ramas derecha e izquierda) es 0, 1 o –1 no es necesario balancear el árbol.
- Si el factor de desequilibrio es –2 se distinguen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

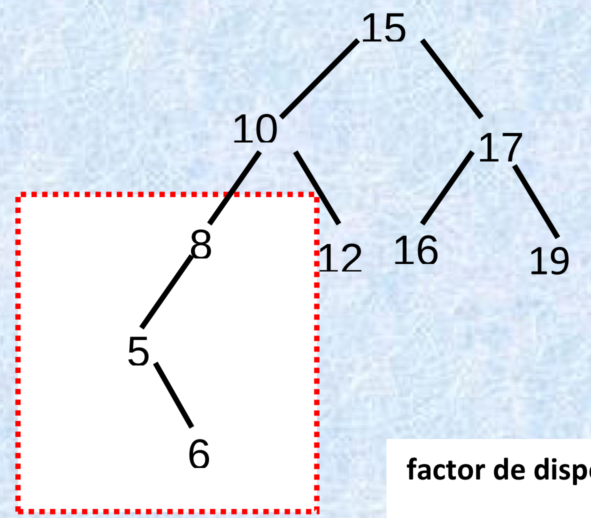
Insertar 4



factor de dispersión:- 2

Izquierda - Izquierda

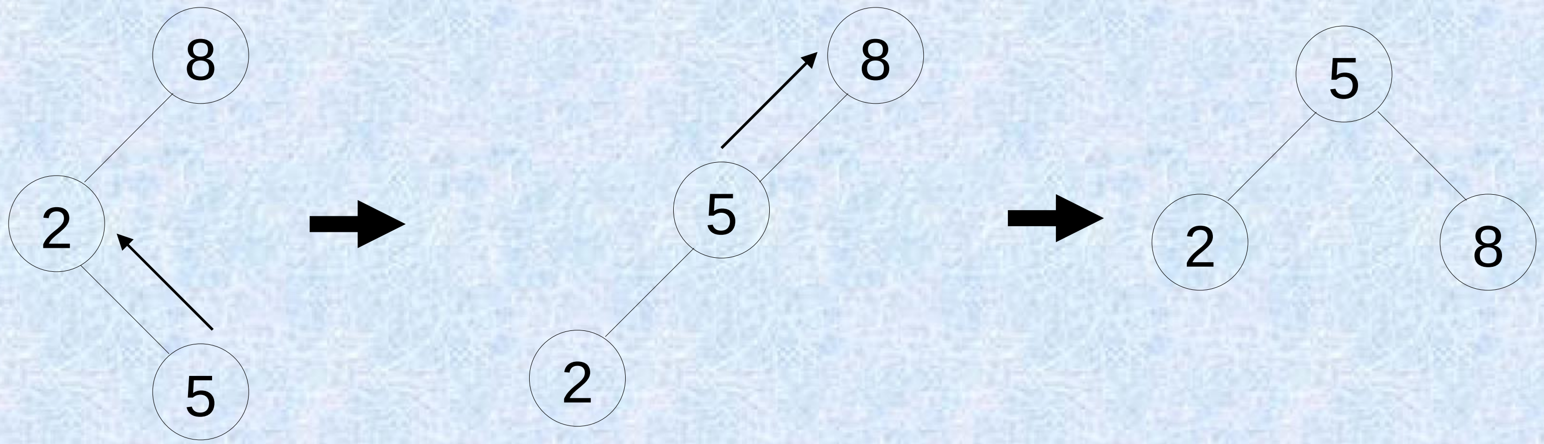
Insertar 6



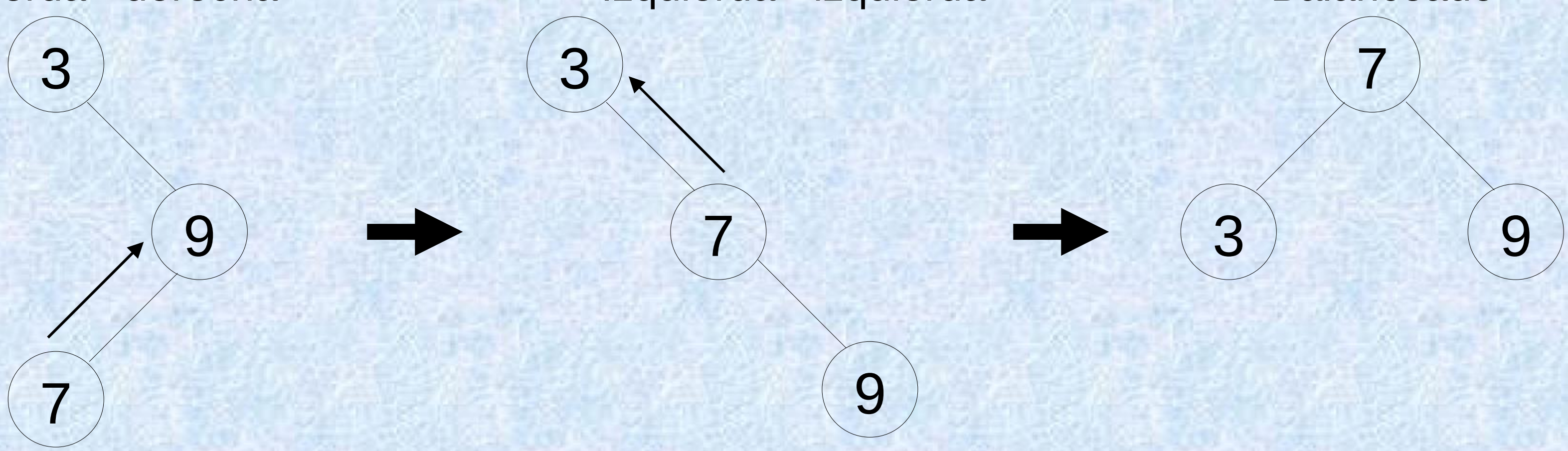
factor de dispersión: -2

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 AVL[1]. 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
```

[1] Las operaciones crear árbol vacio y las observadoras raíz, izquierdo, derecho y vacio? 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 vacío?(a^.izq) entonces
            si vacío?(a^.der) entonces a^.altura←0
            si no a^.altura ← 1 + altura(a^.der)
            finsi
        si no
            si vacío?(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
```

```
             giroizq = T
        X
       / \              Y
      Y   d            /   \
     / \      ==>     Z     X
    Z   c            / \   / \
   / \              a   b c   d
  a   b

             giroizq = F
    X
   / \                  Y
  a   Y                /   \
     / \      ==>     X     Z
    b   Z            / \   / \
       / \          a   b c   d
      c   d
```

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
```

```
             giroizq = T
         X
        / \                 Z
       Y   d              /   \
      / \      ==>       Y     X
     a   Z              / \   / \
        / \            a   b c   d
       b   c

             giroizq = F
     X
    / \                     Z
   a   Y                  /   \
      / \      ==>       X     Y
     Z   d              / \   / \
    / \                a   b c   d
   b   c
```

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))
      entonces {desequilibrio simple izq_izq}
        rotación_simple(a, true)
      si no {desequilibrio doble izq_der}
        rotación_doble(a, true)
      finsi
    (…)
```

ÁRBOLES AVL. MODIFICADORAS (5)

```
    (…)
    si no
      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)
        finsi
      finsi
    finsi
  finsi
finproc
```

ÁRBOLES AVL. MODIFICADORAS (6)

```
                         {insertar un dato en un árbol AVL}
proc insertar[1] (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 borrar[1] (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
        (…)
```

[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^.valor) entonces
            borrar(e, a^.izq)
        si no {en este caso, se sabe que (e>a^.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 semicompleto[1] 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≤]*

usa *ÁRBOLES_BINARIOS[ELEMENTO]*

parametro formal

generos *elemento*

operaciones

$\_ \leq \_ : elemento\ elemento \rightarrow bool$

fparametro

ESPECIFICACIÓN: ÁRBOLES COMPLETOS (2)

operaciones

es_completo? : a_bin → bool

es_semicompleto? : a_bin →bool

es_montículo? : a_bin → bool

operaciones auxiliares

niveles : a_bin → natural {profundidad del árbol}

menor_igual? : elemento a_bin → bool

var

x, y: elemento

i, d: a_bin

ESPECIFICACIÓN: ÁRBOLES COMPLETOS (3)

ecuaciones

```
niveles(Δ) = 0

niveles(i•x•d) = suc(altura(i•x•d))

es_completo?(Δ) = T

es_completo?(i•x•d) =

    es_completo?(i) ∧ es_completo?(d) ∧

                (niveles(i) == niveles(d))
```

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, Δ) = T

menor_igual?(y, i•x•d) = (y ≤ x)

∧ menor_igual?(y, i) ∧ menor_igual?(y, d)

es_montículo?(Δ) = T

es_montículo?(i•x•d) = es_semicompleto?(i•x•d)

∧ menor_igual?(x, i) ∧ menor_igual?(x, d)

∧ es_montículo?(i) ∧ es_montículo?(d)

fespec

IMPLEMENTACIÓN: MONTÍCULOS

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

- $\mathbf{2^i}$ nodos de nivel i, para $0 \leq i \leq h$,
- $\mathbf{2^h}$ hojas (que son los nodos de nivel h),
- $\mathbf{2^{h+1} - 1}$ 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 $\mathbf{2^{h+1} - 1}$ 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[2k+1], T[2k+2],…T[2(k+1)-1].
- El padre del nodo almacenado en T[i] está en la posición igual a “i div 2”.

IMPLEMENTACIÓN: MONTÍCULOS (3)

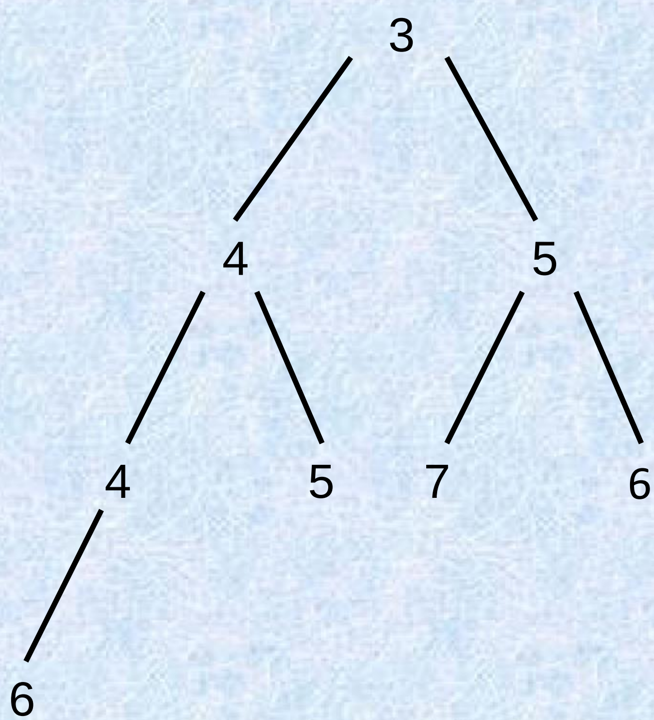
Cuando se inserta un elemento, hay que **flotar** dicho elemento si no se encuentra en su posición correcta:

- 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:

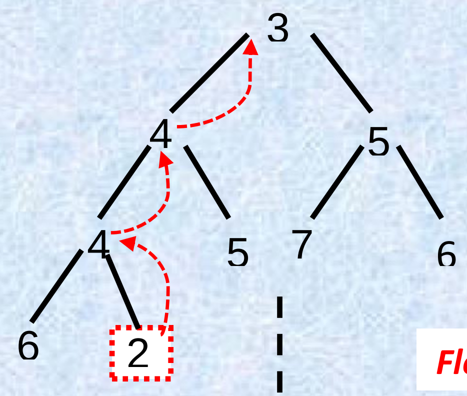
- se coloca el último elemento en la primera posición, y
- si es necesario se intercambia con el menor de los hijos mientras sea necesario.

IMPLEMENTACIÓN: MONTÍCULOS (4)

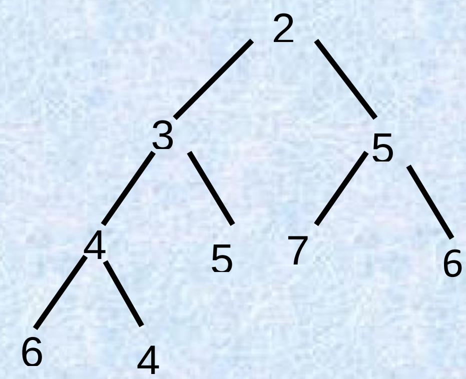


[3,4,5,4,5,7,6,6]

Insertar 2



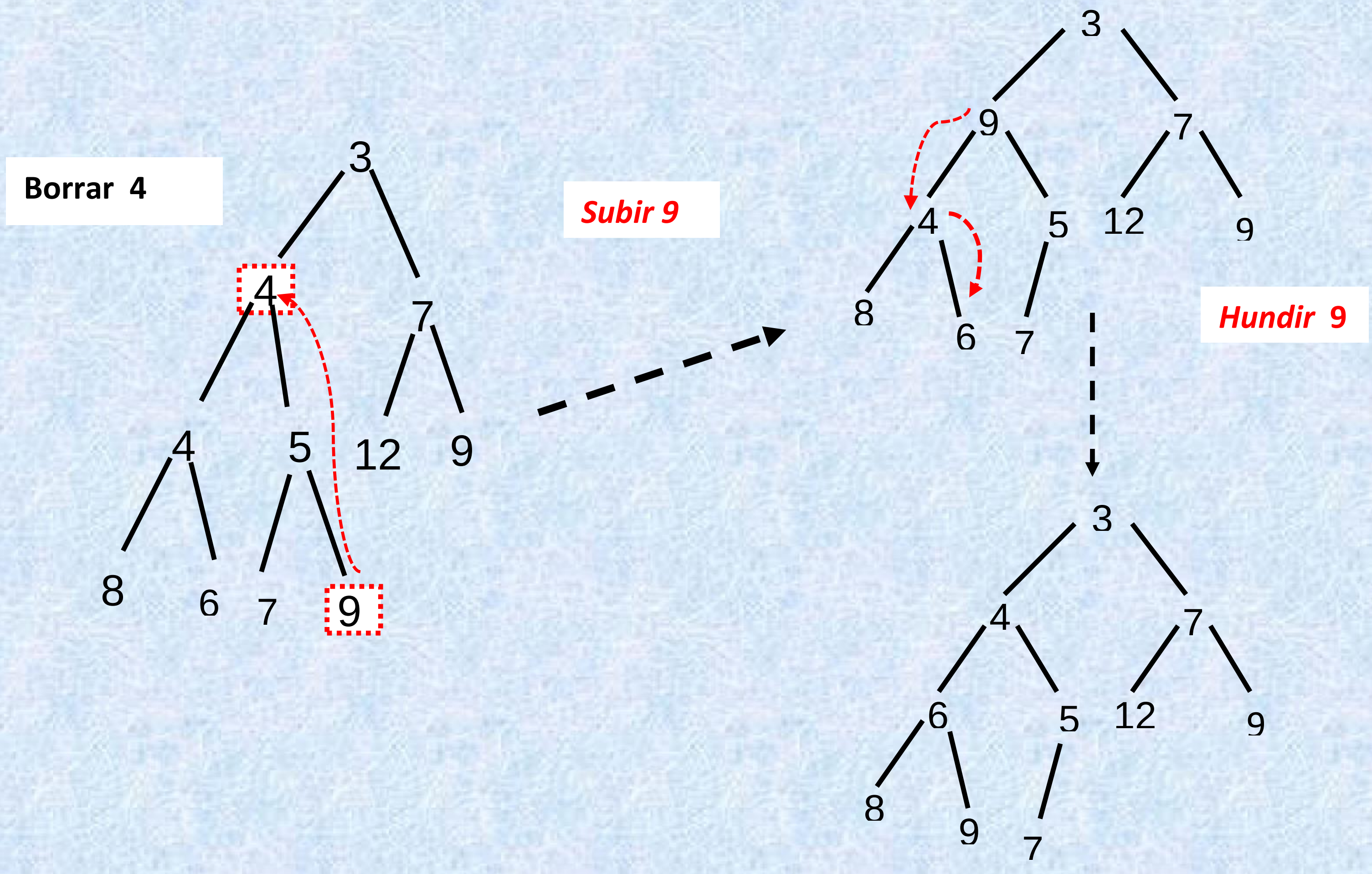
Flotar 2



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

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

IMPLEMENTACIÓN: MONTÍCULOS (5)



[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) ∧ (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 ← 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 ← m.tamaño + 1
      m.datos[m.tamaño] ← 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 ← 2*i
      hijoDer ← 2*i+1
      j ← i
         {buscamos el dato a intercambiar con el i-ésimo, el
         menor entre sus hijos si ambos son menores que él}
      si (hijoDer <= m.tamaño) ∧
            (m.datos[hijoDer] < m.datos[i]) entonces
         i ← 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
```