Estructuras de Datos.
Grado en Ingeniería Informática, Ingeniería del Software e Ingeniería de Computadores
ETSI Informática
Universidad de Málaga

Tema 4. Árboles
Móntículos Maxifóbicos

José E. Gallardo, Francisco Gutiérrez, Pablo López, Laura Panizo
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga

Montículos Maxifóbicos (I)

- Llamamos peso de un nodo al número de elementos que *cuelgan* desde el nodo

- Representaremos un montículo vía un árbol binario aumentado: en cada nodo se almacena (además de su valor y los hijos) su peso:

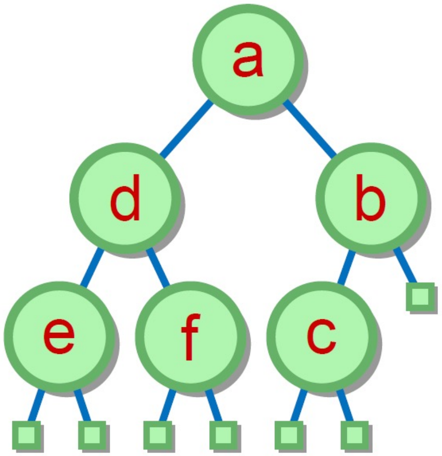
```haskell
data Heap a  = Empty | Node a Int (Heap a) (Heap a)

weight :: Heap a -> Int
weight Empty           = 0
weight (Node _ w _ _)  = w
```

Peso del nodo

Árbol con raíz 'a' con 6 elementos

```haskell
h1 :: Heap Char
h1 = Node 'a' 6 (Node 'd' 3 (Node 'e' 1 Empty Empty)
                            (Node 'f' 1 Empty Empty))
                (Node 'b' 2 (Node 'c' 1 Empty Empty)
                             Empty)
```

Árbol con raíz 'b' con 2 elementos



Montículos Maxifóbicos (II)

- Un árbol es Montículo Maxifóbico que se representa con un árbol binario aumentado y que mantiene la propiedad HOP:
  - En cada nodo la clave es menor que la clave de sus hijos
  - El `minElem` se encuentra en la raíz

- Los montículos maxifóbicos no están balanceados

- Las operaciones de inserción y borrado tienen complejidad logarítmica gracias a la operación de mezcla que utilizan
  - `delMin` descarta la raíz y mezcla sus dos hijos
  - `insert` mezcla el árbol existente con un nuevo árbol que contiene únicamente el elemento a insertar (`singleton`)

Montículos Maxifóbicos (II)

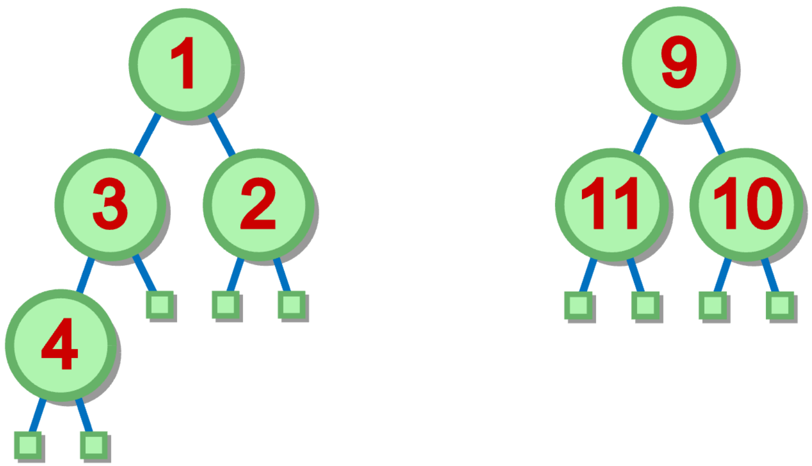
- Operación de mezcla de dos montículo maxifóbicos:
  - Se compara el valor de las raíces de los dos árboles
    - El valor menor se convierte en la raíz del montículo mezcla (`hm`)
    - El resto de información se distribuye en 3 árboles: ganador (`winner`) (contiene la clave mayor) e hijo izquierdo (`lhl`) y derecho del perdedor (`rhl`)
  - De los 3 árboles `winner`, `lhl`, `rhl`:
    - El que tenga más nodos (mayor peso) se convierte en uno de los hijos del montículo mezcla `hm`, por ejemplo el izquierdo
    - Los otros 2 árboles se mezclan y se convierten en el otro hijo de el montículo mezcla `hm`, por ejemplo el derecho

De este modo el montículo mezcla contiene todas las claves de los originales y además verifica la propiedad HOP

Montículos Maxifóbicos (II)

- Vamos a generar un nuevo montículo maxifóbico mezclando dos montículos maxifóbicos

```
merge h1 h2            ➔          hm
```



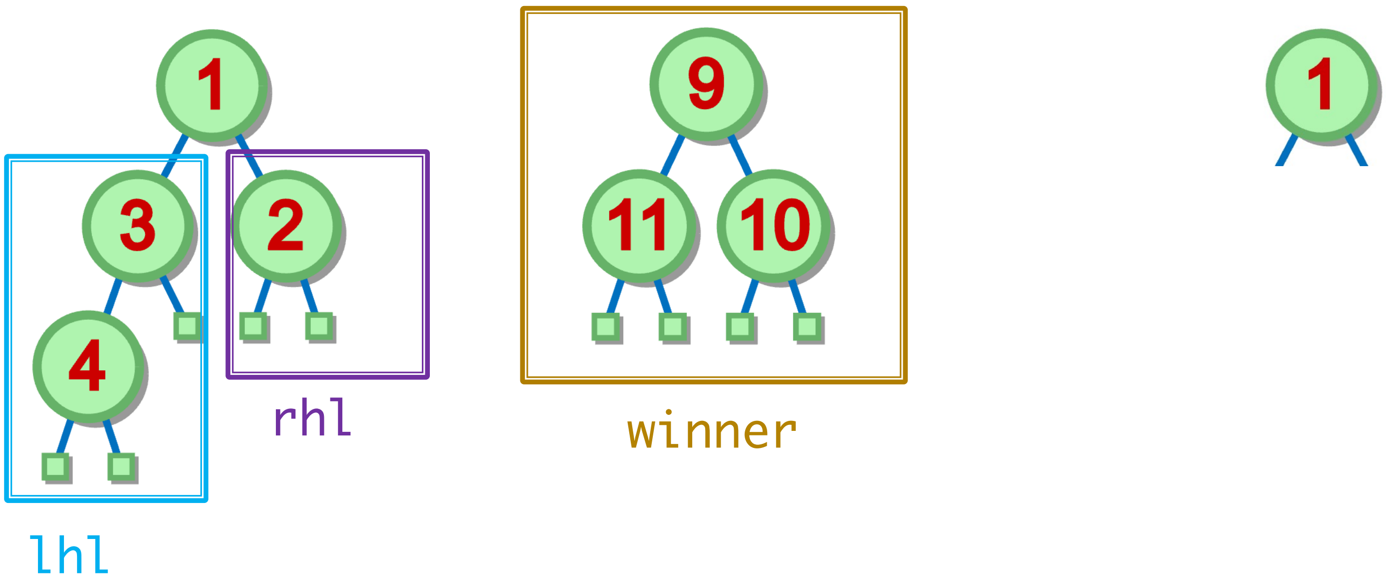
Montículos Maxifóbicos (II)

1) seleccionamos el montículo con la clave menor en la raíz

h1 es el montículo perdedor → su raíz será la raíz del montículo mezcla

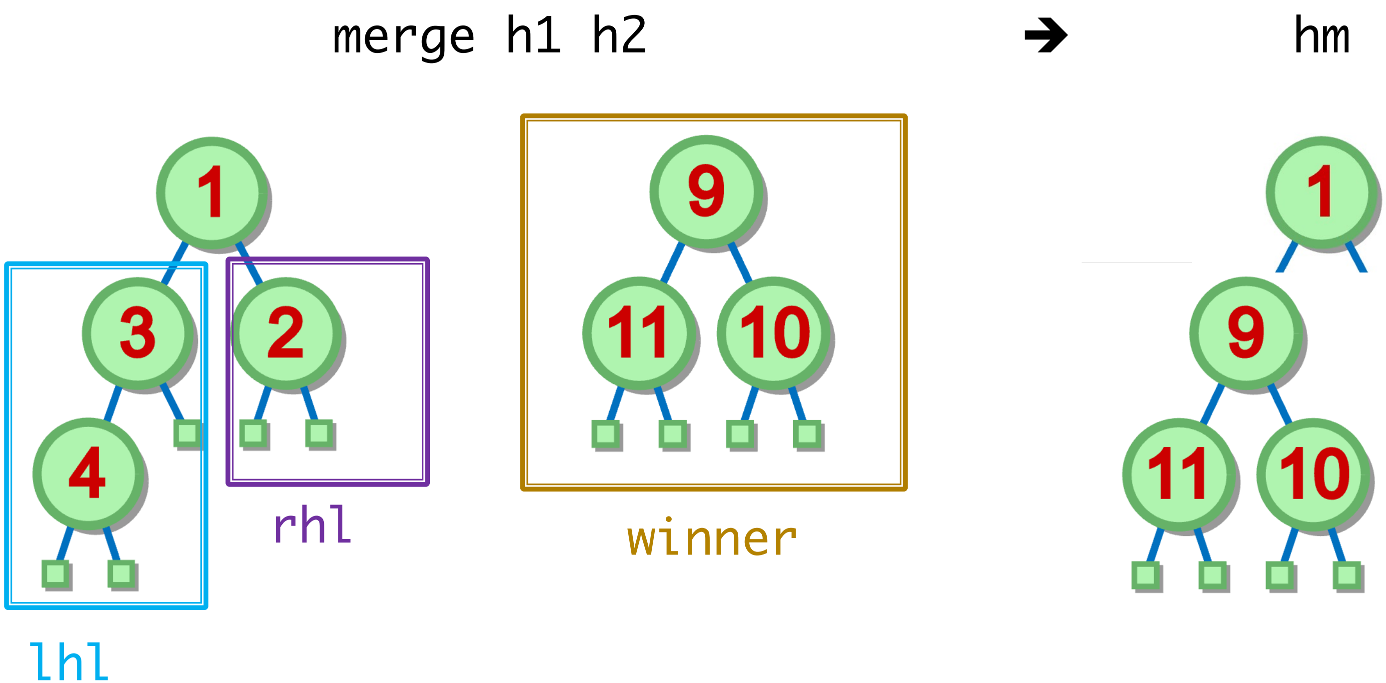
h2 es el montículo ganador

`merge h1 h2` → `hm`



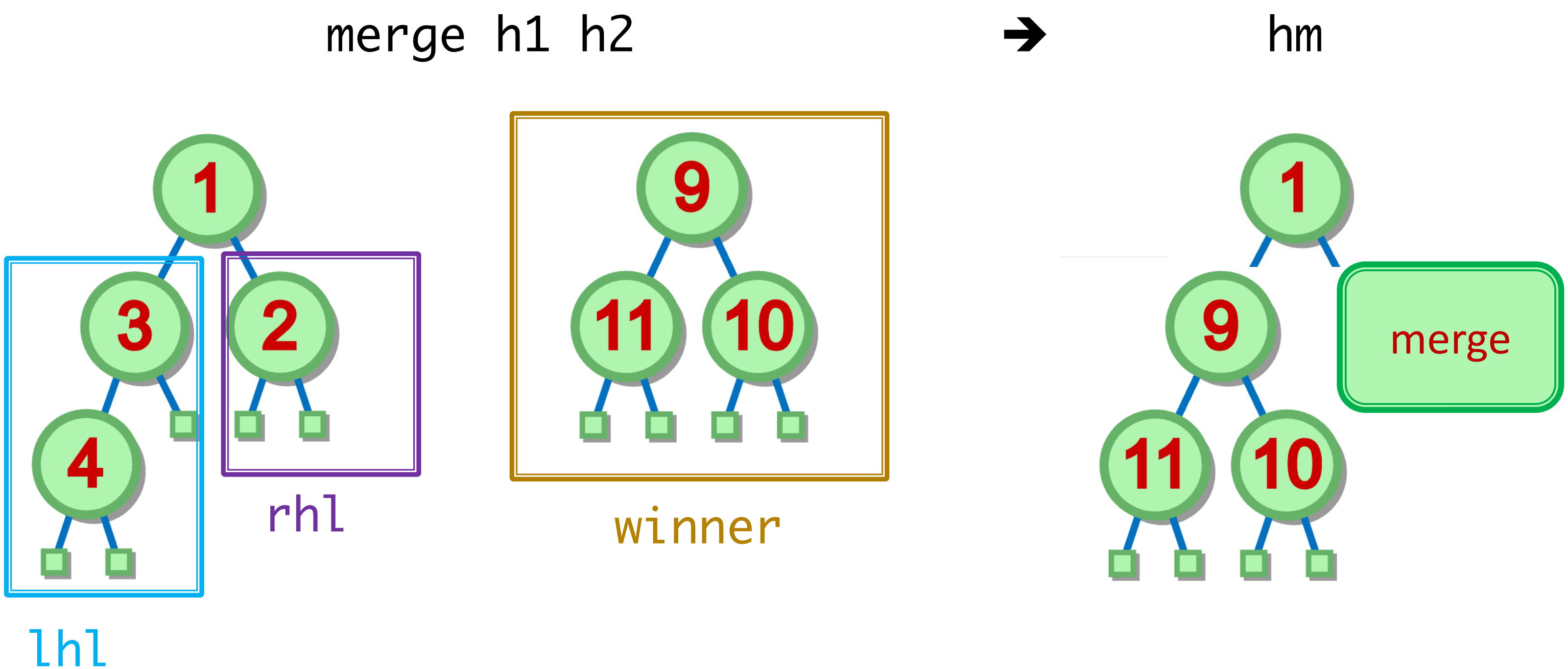
Montículos Maxifóbicos (II)

2) Dados `lhl`, `rhl` y `winner`, el montículo con mayor número de nodos se convierte en uno de los hijos de hm (en el ejemplo el izquierdo)



Montículos Maxifóbicos (II)

3) Dados `lhl`, `rhl` y `winner`, los dos árboles con menor número de nodos se mezclan y el resultado se convierte en el otro hijo de hm

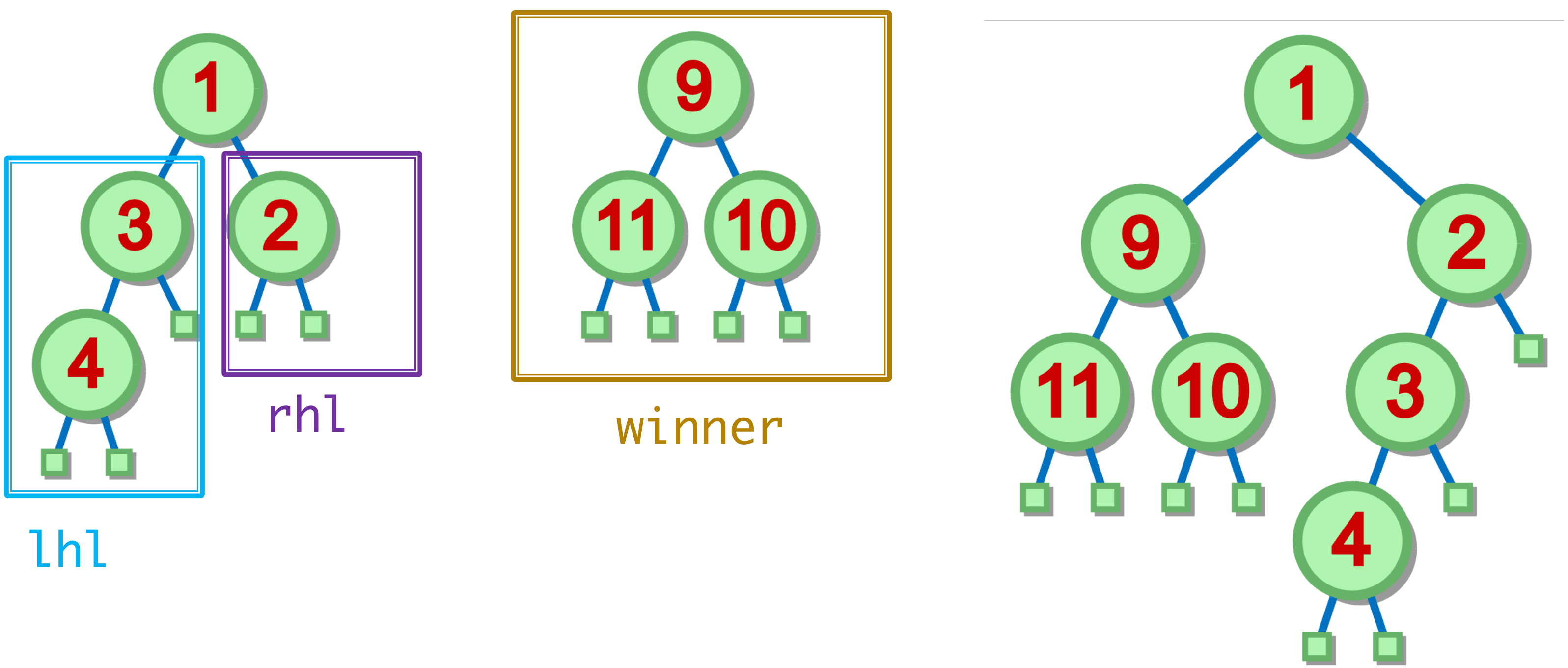


En el ejemplo peso (`winner`) > peso (`lhl`) > peso (`rhl`) → el más pesado se convierte en hijo izquierdo de hm y la mezcla de los otros dos en el hijo derecho

Montículos Maxifóbicos (II)

3) Dados `lhl`, `rhl` y `winner`, los dos árboles con menor número de nodos se mezclan y el resultado se convierte en el otro hijo de hm

`merge h1 h2` ➔ `hm`



Montículos Maxifóbicos (III)

```haskell
module DataStructures.Heap.MaxiphobicHeap
  ( 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 árbol

Montículos Maxifóbicos(IV)

```haskell
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

```haskell
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

```haskell
merge :: (Ord a) => Heap a -> Heap a -> Heap a
merge h1 h2 = undefined
```

Las operaciones delicadas (`insert` y `delMin`) y su complejidad dependen de la definición de `merge`.

La definición de `merge` tiene complejidad logarítmica ☺

Construcción de un Montículo Maxifóbico a partir de una Lista

```haskell
-- 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)

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

Construimos una lista de montículos unitarios

Mezclamos dos a dos hasta obtener un solo elemento

Mezcla dos a dos

- Sea T(n) el número de pasos de mkHeap para una lista con n elementos:
  - $T(n) = n/2 \cdot O(\log_2 1) + n/4 \cdot O(\log_2 2) + n/8 \cdot O(\log_2 4) + \ldots + 1 \cdot O(\log_2 (n/2))$

n/2 mezclas de montículos de 1 elemento

n/4 mezclas de montículos de 2 elementos

n/8 mezclas de montículos de 4 elementos

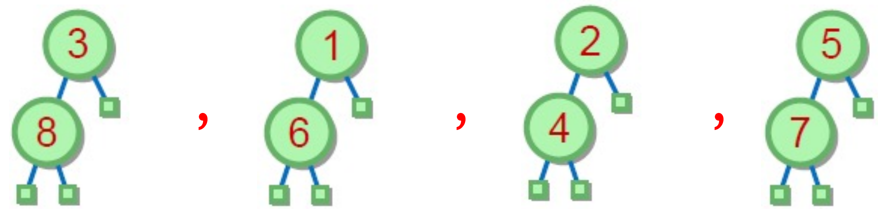
1 mezcla de montículos de n/2 elementos

La solución a T(n) es O(n) ☺

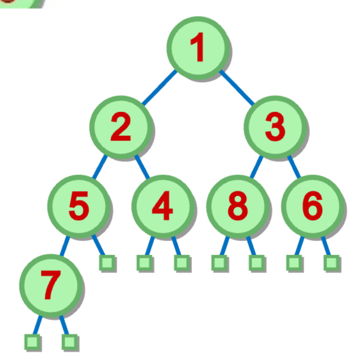
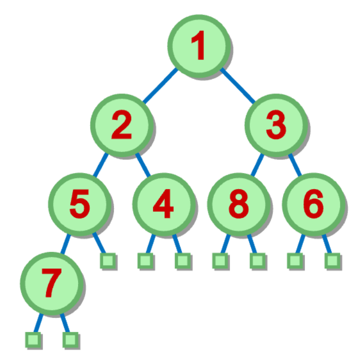
Construcción de un Montículo Zurdo a partir de una Lista en Tiempo Lineal (y II)

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

mergeLoop (mergePairs[]) =>

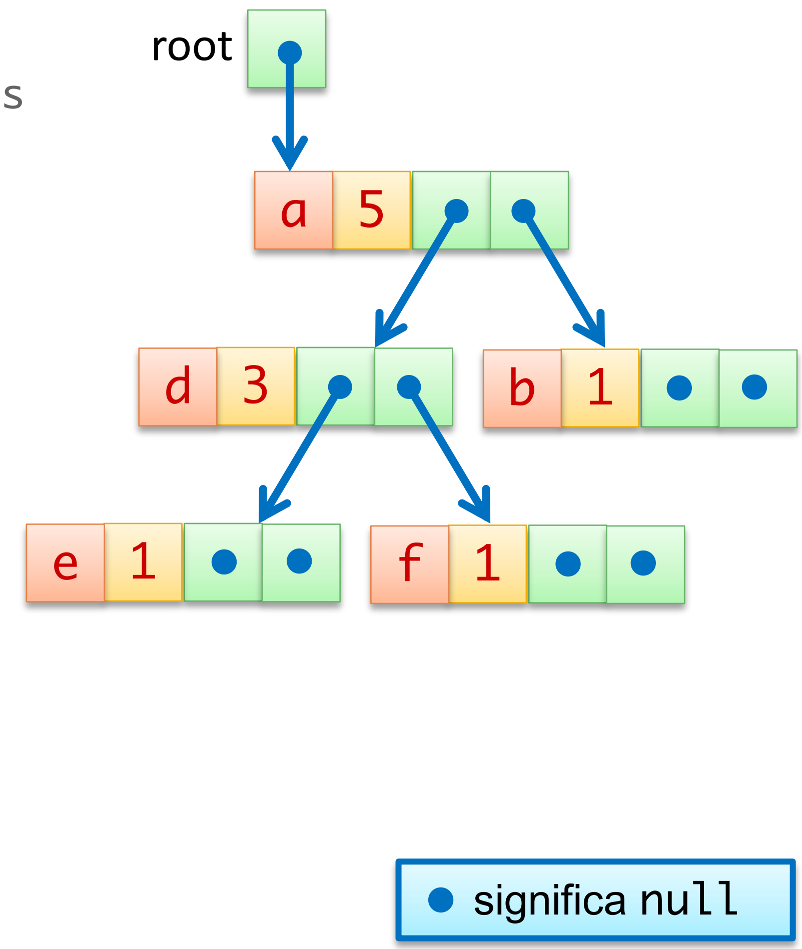
mergeLoop (mergePairs []) =>

mergeLoop (mergePairs []) =>

mergeLoop [] => 

Montículos Maxifóbicos en Java (I)

```java
package dataStructures.heap;

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

 protected static class Tree<E> {
   E elem;         // elemento raíz del MZ
   int size;     // 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 MaxiphobicHeap () {
   root = null;
 }

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

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

 public int size() {
   return root == null ? 0 : root.size;
 }
```



Montículos Maxifóbicos en Java (II)

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

El mínimo en la raíz

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

Mezclamos los hijos sin la raíz

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

Creamos un nuevo montículo con un elemento

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

Montículos Maxifóbicos en Java (III)

```java
private static <T extends Comparable<? super T>>
        Tree<T> merge(Tree<T> h1, Tree<T> h2) {

    Tree<T> hm;

    /* TODO : Implementación eficiente que reusa los nodos de los
              árboles mezclados */

}
```