Colas de prioridad

(continuación)

Agenda

- Aplicaciones
- Definición
- Distintas implementaciones
- Heap Binaria
  - Propiedad Estructural
  - Propiedad de Orden
  - Implementación
- Operaciones: Insert, DeleteMin, Operaciones adicionales
- Construcción de una Heap: operación BuildHeap
  - Eficiencia
- HeapSort

¿Cómo construir una heap a partir de una lista de elementos?

Para construir una heap a partir de una lista de *n* elementos:

✓ Se pueden insertar los elementos de a uno

⇒ se realizan (n log n) operaciones en total

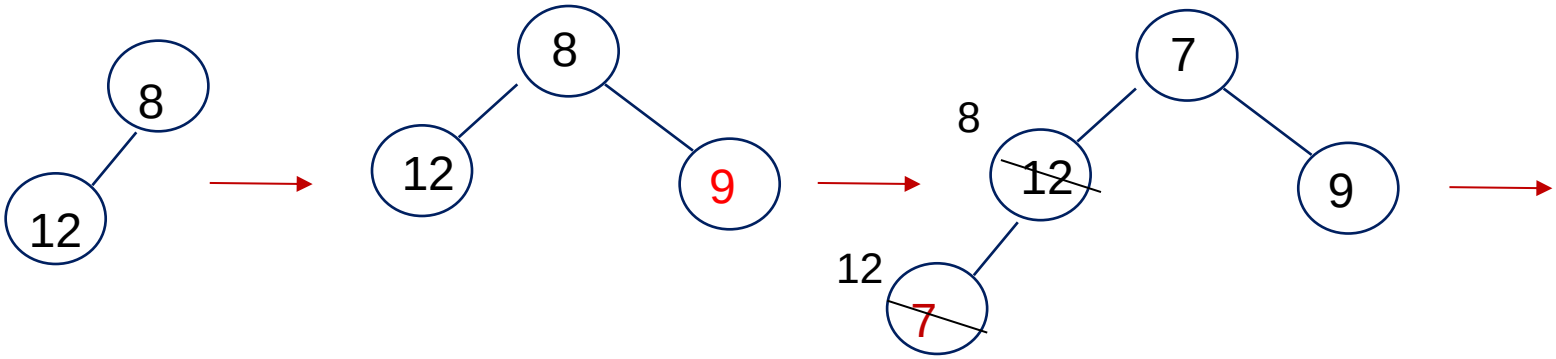
✓ Se puede usar un algoritmo de orden lineal, es decir, proporcional a los n elementos ⇒ **BuildHeap**

- Insertar los elementos desordenados en un árbol binario completo
- Filtrar hacia abajo cada uno de elementos

Construcción de una Heap

insertando los elementos uno a uno

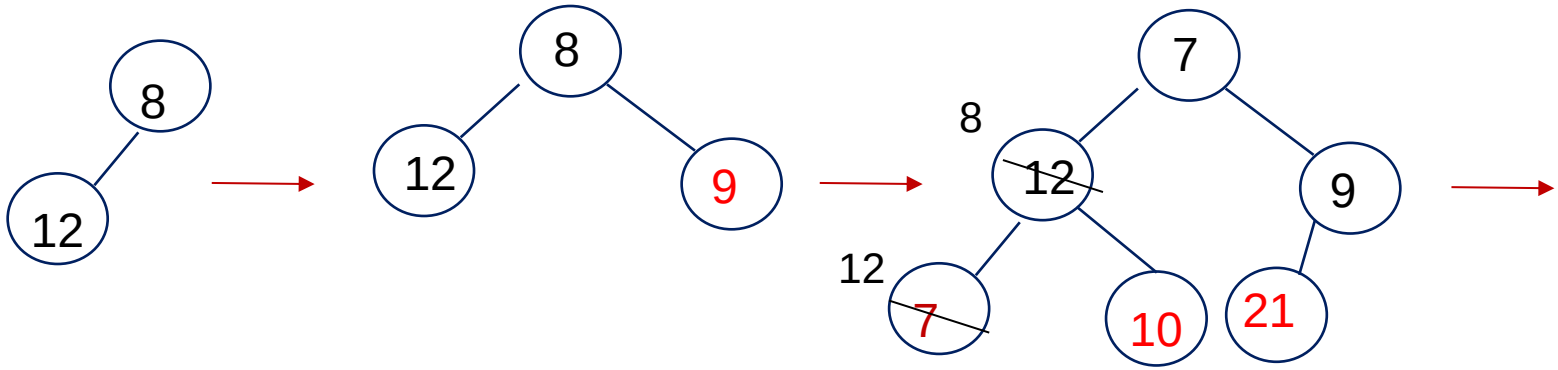
Minheap 8 12 9 7 10 21 6 4



Construcción de una Heap

insertando los elementos uno a uno

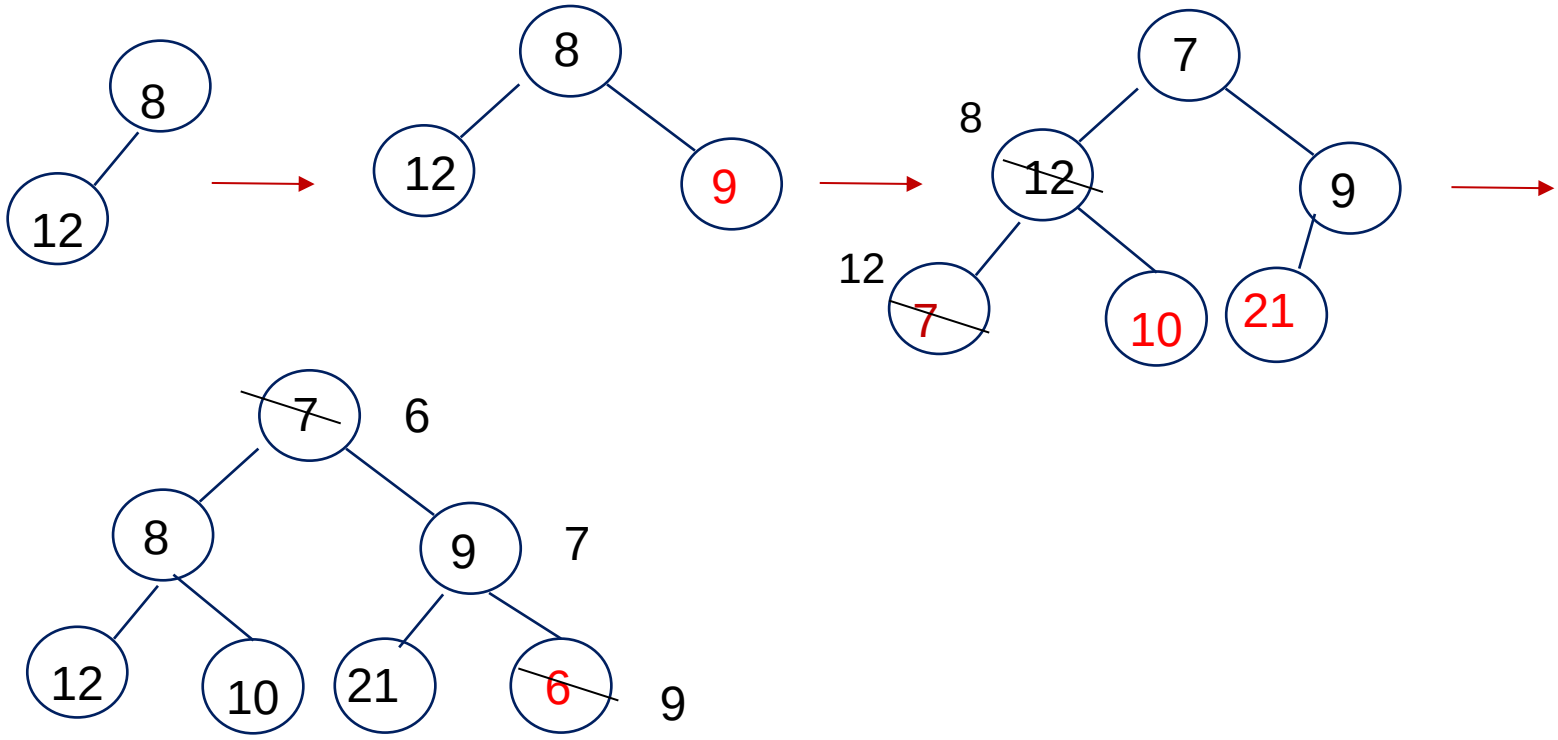
Minheap 8 12 9 7 10 21 6 4



Construcción de una Heap

insertando los elementos uno a uno

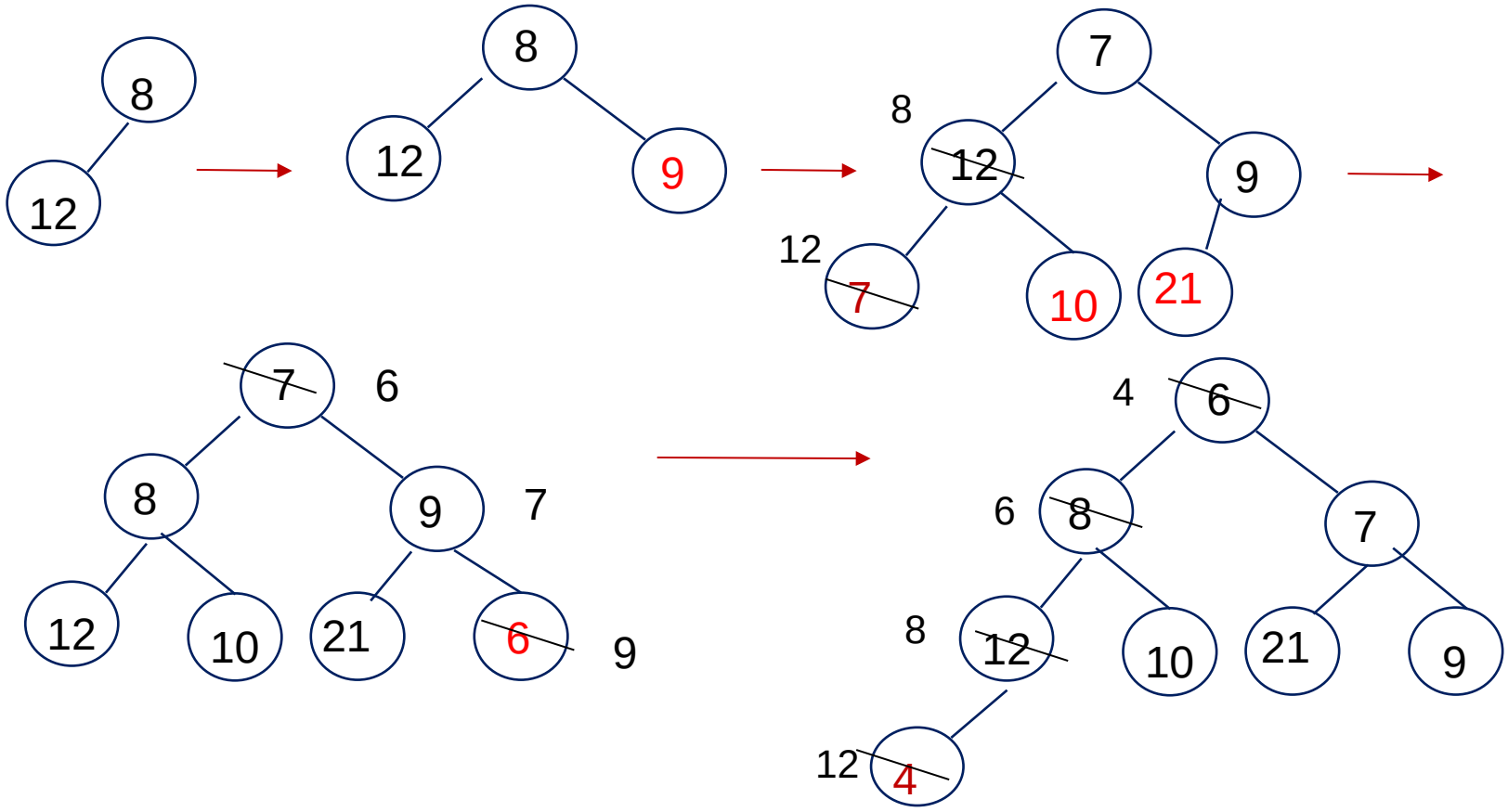
Minheap 8 12 9 7 10 21 6 4



Construcción de una Heap

insertando los elementos uno a uno

Minheap 8 12 9 7 10 21 6 4



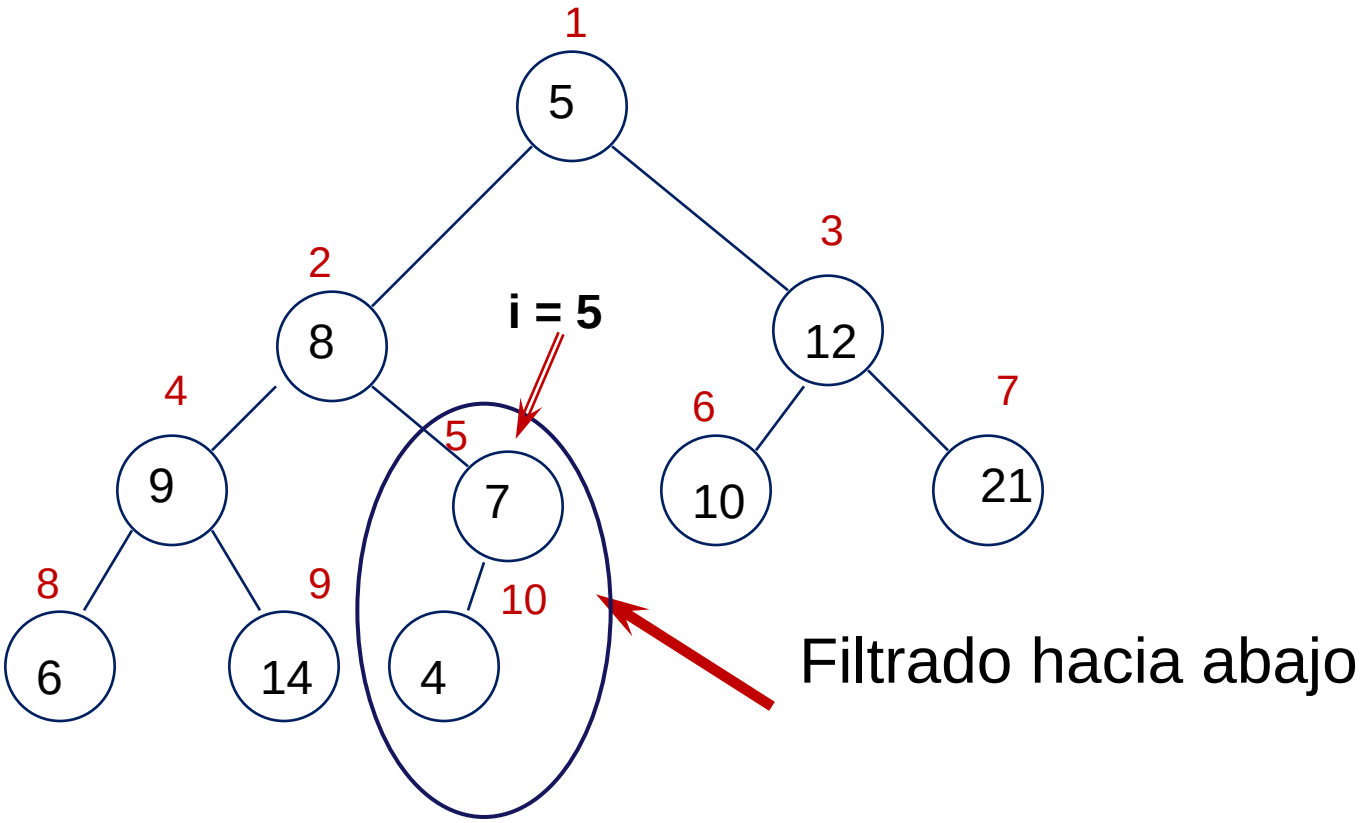
Algoritmo BuildHeap

- Para filtrar:
  - se elige el menor de los hijos
  - se compara el menor de los hijos con el padre

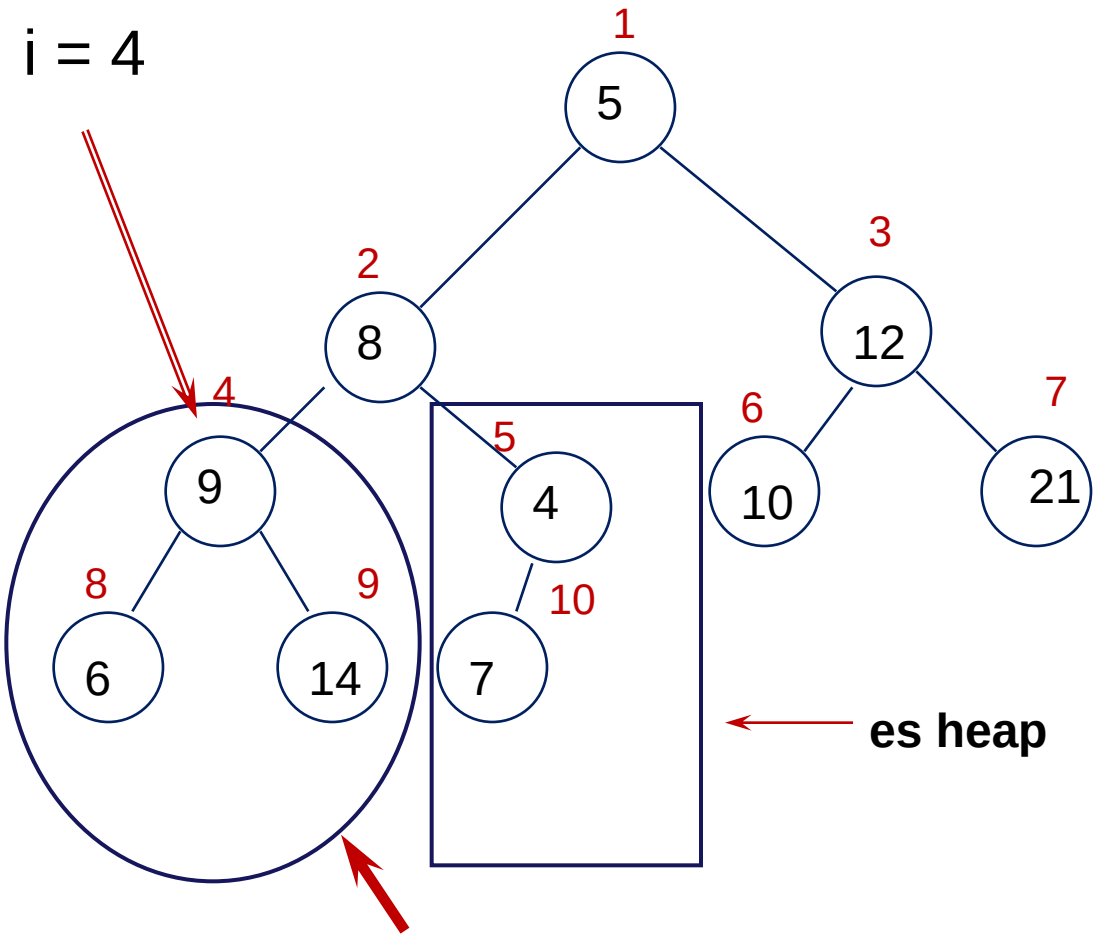
- Se empieza filtrando desde el elemento que está en la posición (tamaño/2):
  - se filtran los nodos que tienen hijos
  - el resto de los nodos son hojas

BuildHeap

5	8	12	9	7	10	21	6	14	4
1	2	3	4	5	6	7	8	9	10

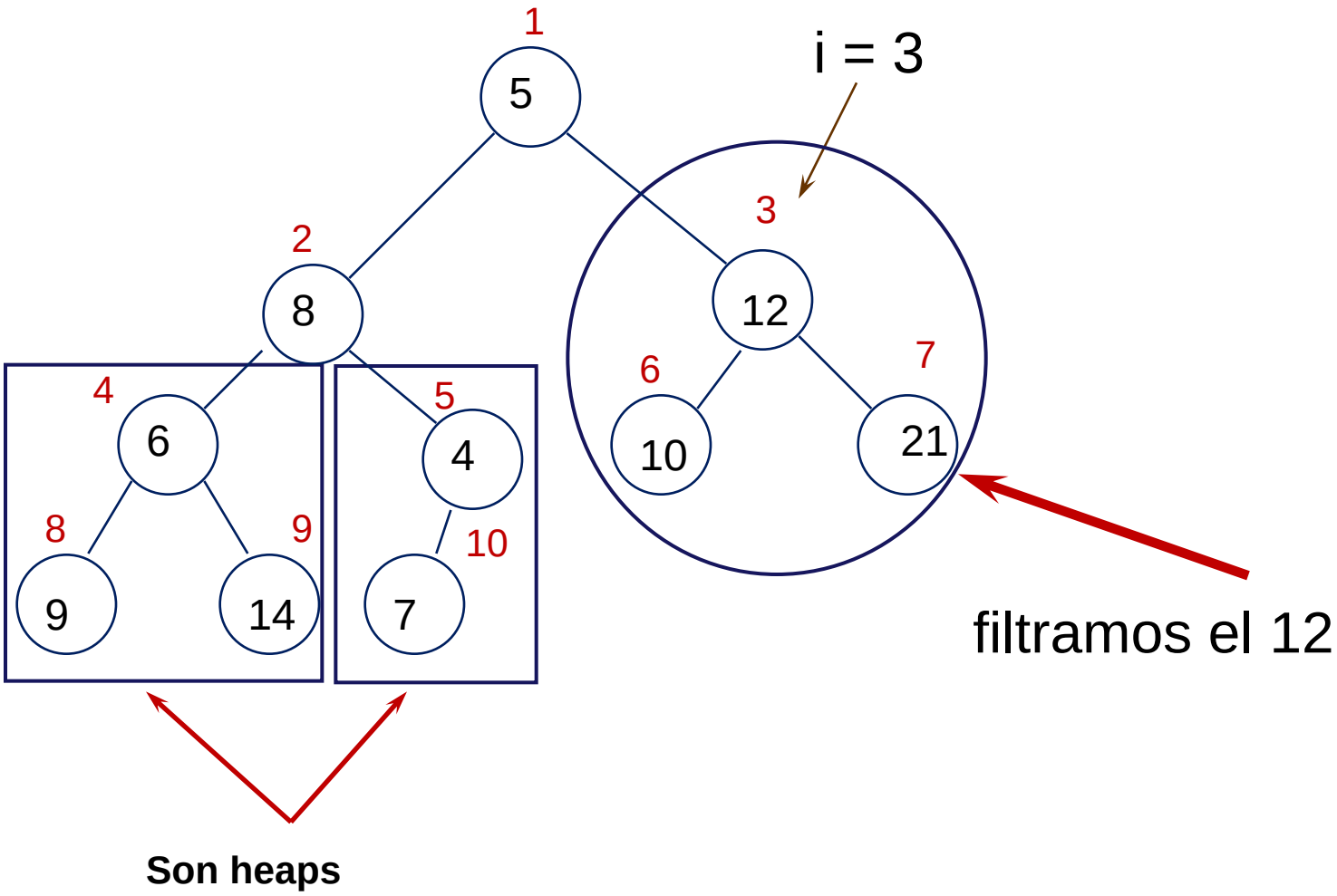


BuildHeap

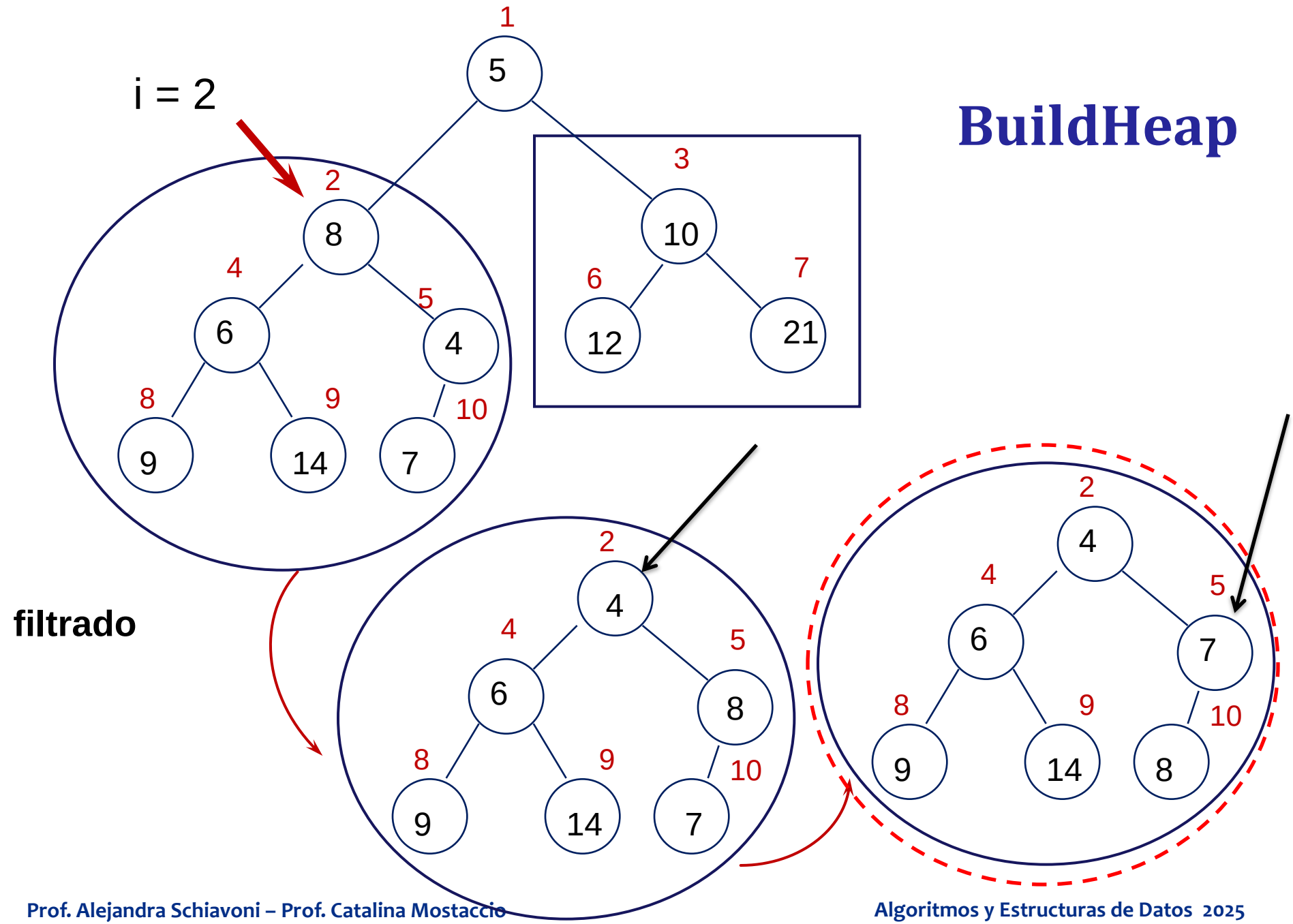


filtramos el 9

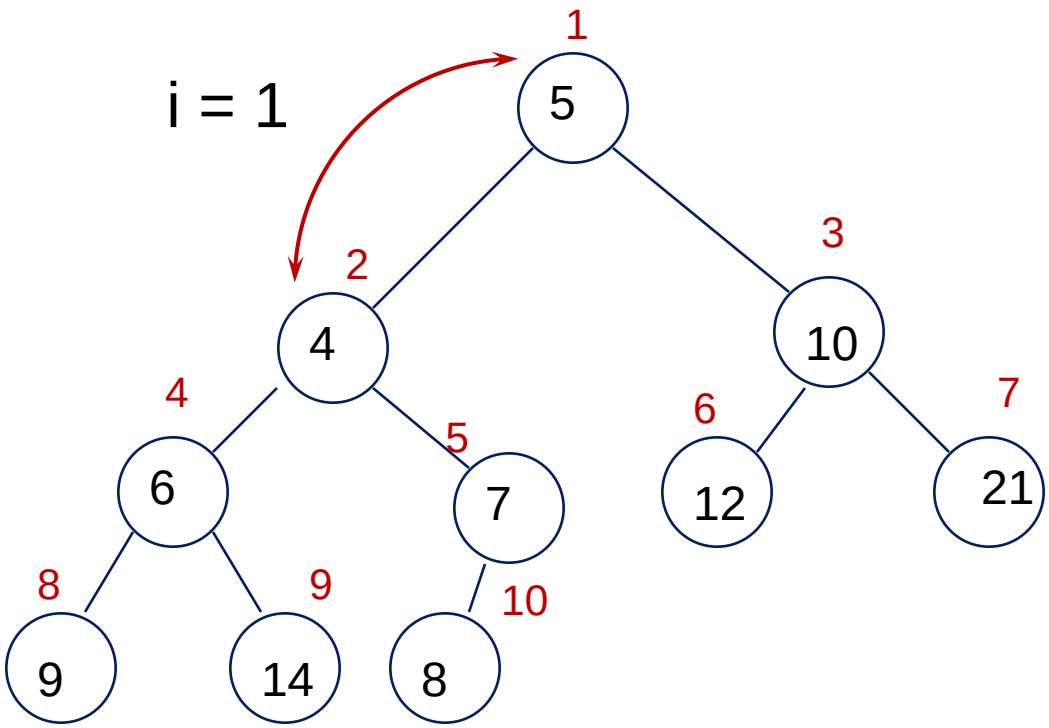
BuildHeap

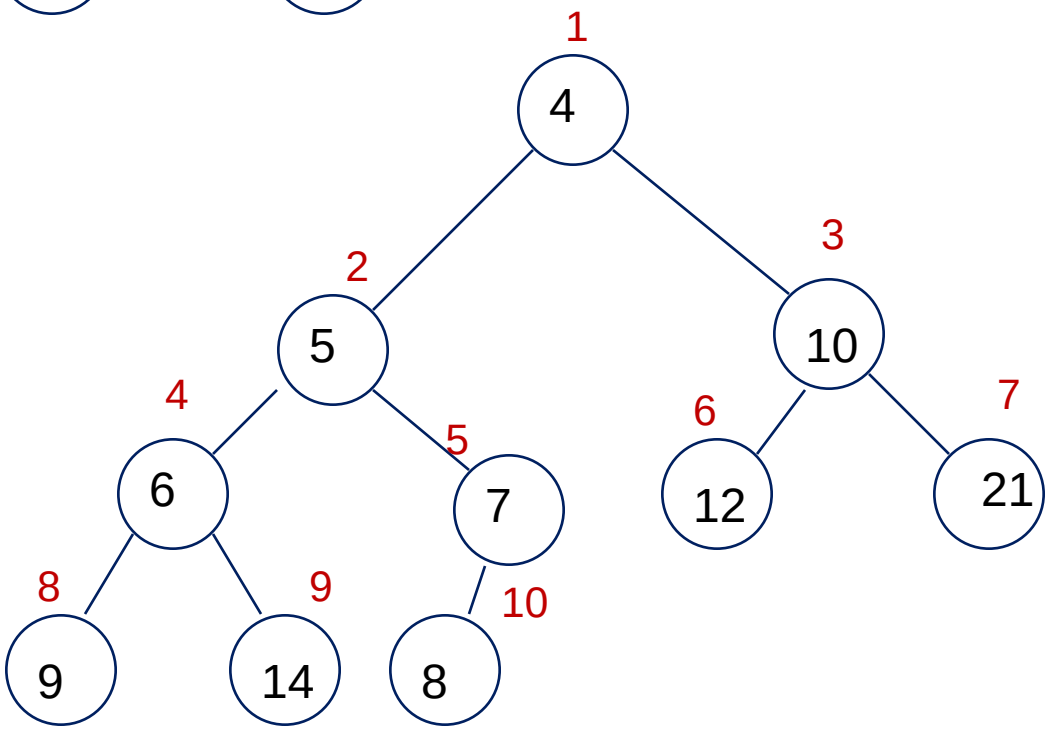


BuildHeap


i = 2
1
5
2
8
3
10
4
6
5
4
6
7
12
21
8
9
9
14
7
10
filtrado
2
4
4
6
5
8
8
9
9
14
7
10
2
4
4
6
5
7
8
9
9
14
8
10

BuildHeap


i = 1
1
5
2
4
3
10
4
6
5
7
6
12
7
21
8
9
9
14
10
8

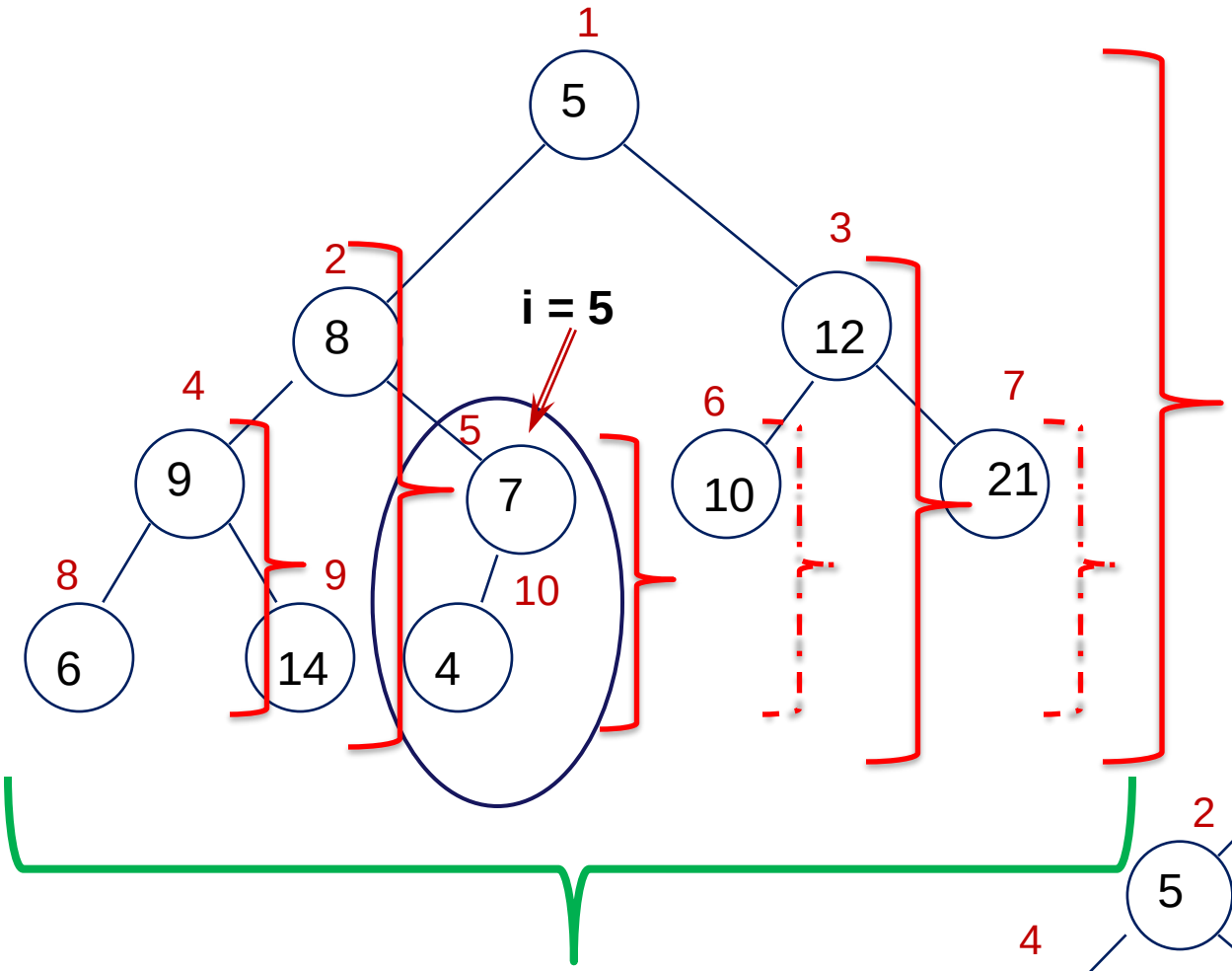

1
4
2
5
3
10
4
6
5
7
6
12
7
21
8
9
9
14
10
8

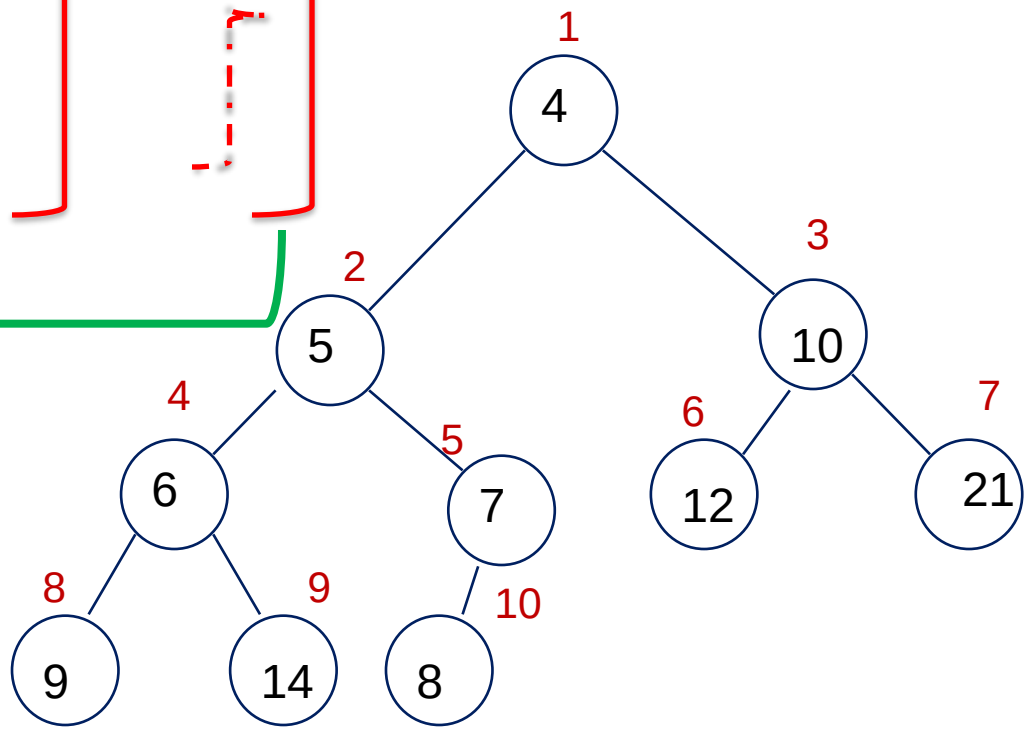
Cantidad de operaciones requeridas

- En el filtrado de cada nodo recorremos su altura

- Para acotar la cantidad de operaciones (*tiempo de ejecución*) del algoritmo BuildHeap, debemos calcular la suma de las alturas de todos los nodos

BuildHeap





Σ alturas de cada nodo

Cantidad de operaciones requeridas (cont.)

Teorema:

En un árbol binario lleno de altura h que contiene $2^{h+1} – 1$ nodos, la suma de las alturas de los nodos es: $2^{h+1} – 1 – (h + 1)$

Demostración:

Un árbol tiene 2^i nodos de altura h – i

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$S = h + 2 (h-1) + 4 (h-2) + 8 (h-3) + \ldots\ldots 2^{h-1} (1)$$

Cantidad de operaciones requeridas (cont.)

$S = h + 2\,(h-1) + 4\,(h-2) + 8\,(h-3) + \ldots\ldots\ldots\ldots + 2^{h-1}\,(1)$ (A)

$2S = 2h + 4\,(h-1) + 8\,(h-2) + 16\,(h-3) + \ldots\ldots\ldots + 2^{h}\,(1)$ (B)

Restando las dos igualdades (B) – (A)

$S = -h + 2\,(h-(h-1)) + 4((h-1)-(h-2)) + 8\,((h-2)-(h-3)) + \ldots + 2^{h-1}(2-1) + 2^{h}$

$S = -h + 2 + 4 + 8 + 16 + \ldots\ldots\ldots + 2^{h-1} + 2^{h}$

$S + 1 = -h + 1 + 2 + 4 + 8 + 16 + \ldots\ldots\ldots + 2^{h-1} + 2^{h}$

$S + 1 = -h + (2^{h+1} - 1)$

$S = (2^{h+1} - 1) - (h + 1)$

Cantidad de operaciones requeridas (cont.)

- Un árbol binario completo no es un árbol binario lleno, pero el resultado obtenido es una cota superior de la suma de las alturas de los nodos en un árbol binario completo
- Un árbol binario completo tiene entre 2^h y $2^{h+1} - 1$ nodos, el teorema implica que esta suma es de O(n) donde n es el número de nodos.
- Este resultado muestra que la operación BuildHeap es lineal

Ordenación de vectores usando Heap

Dado un conjunto de *n* elementos y se los quiere ordenar en forma creciente, existen dos alternativas:

a) *Algoritmo que usa una heap y requiere una cantidad aproximada de (n log n) operaciones.*

- Construir una MinHeap, realizar **n** DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.
  - Desventaja: **requiere el doble de espacio**

Ordenación de vectores usando Heap

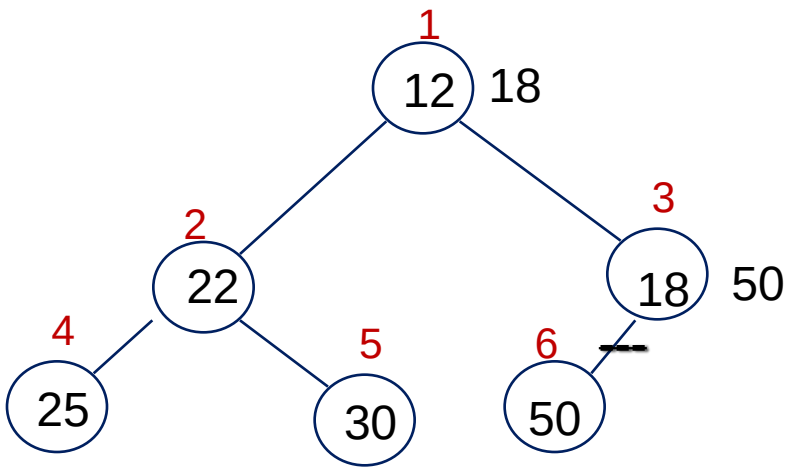
*Ejemplo: Construir una MinHeap, realizar **6** DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.*

50	30	18	25	22	12

entrada

12					

salida 1 2 3 4 5 6



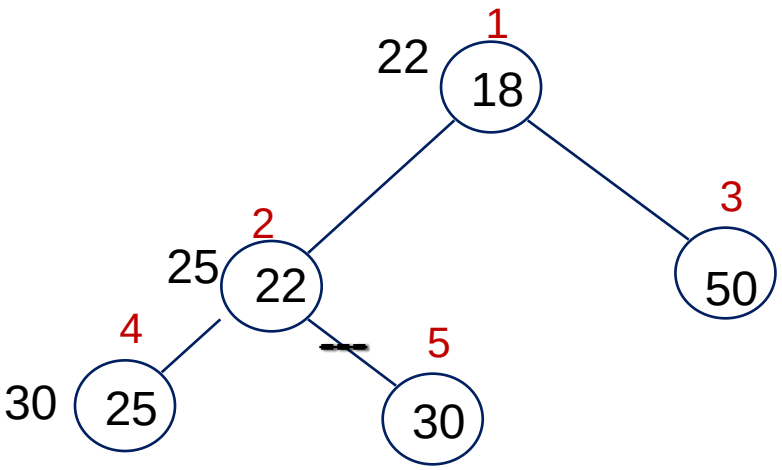
12	22	18	25	30	50

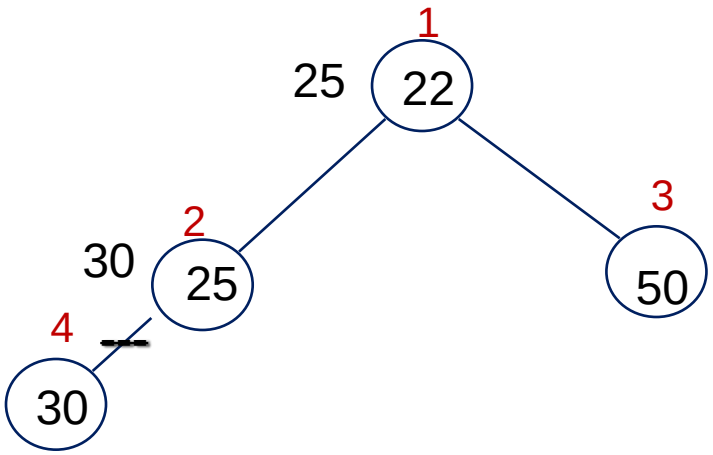
1 2 3 4 5 6

18	22	50	25	30	~~50~~

1 2 3 4 5 6

Ordenación de vectores usando Heap





salida

12	18	22			
1	2	3	4	5	6

18	22	50	25	30	~~30~~
1	2	3	4	5	6

22	25	50	30	~~30~~	~~50~~
1	2	3	4	5	6

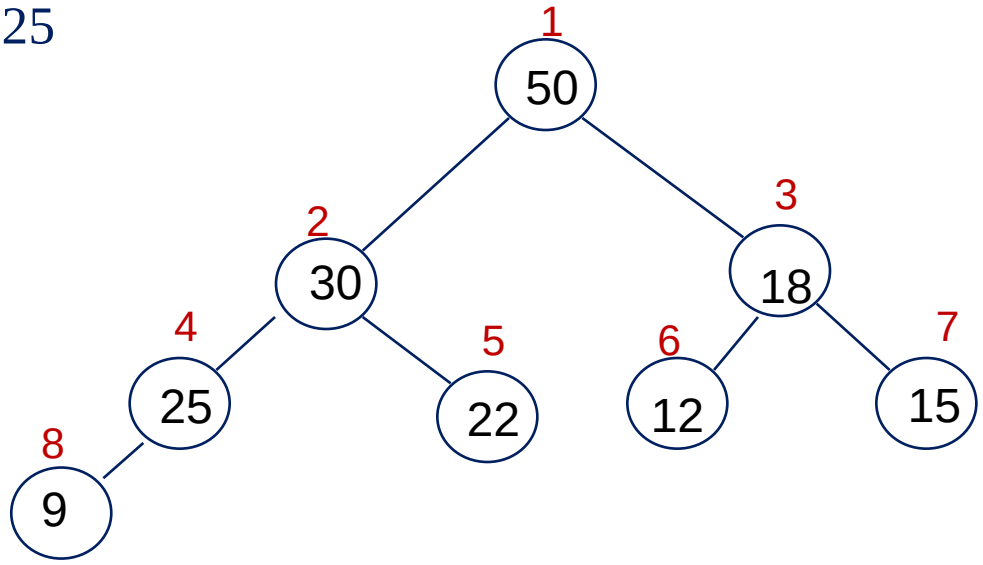
25	30	50	~~30~~	~~30~~	~~50~~
1	2	3	4	5	6

Ordenación de vectores usando Heap: Algoritmo HeapSort

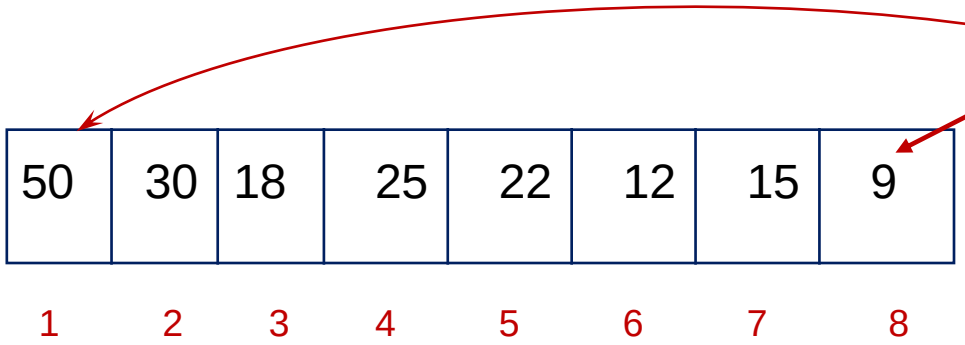
b) *Algoritmo* ***HeapSort*** *que requiere una cantidad aproximada de (n log n) operaciones, pero* ***menos espacio****.*

➢ Construir una MaxHeap con los elementos que se desean ordenar, intercambiar el último elemento con el primero, decrementar el tamaño de la heap y filtrar hacia abajo. Usa sólo el espacio de almacenamiento de la heap.

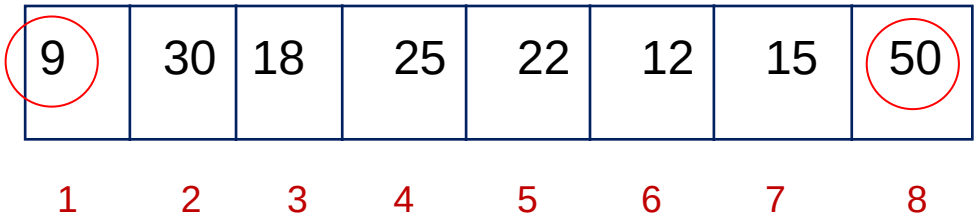
Ejemplo: 9, 50, 18, 30, 22, 12, 15, 25

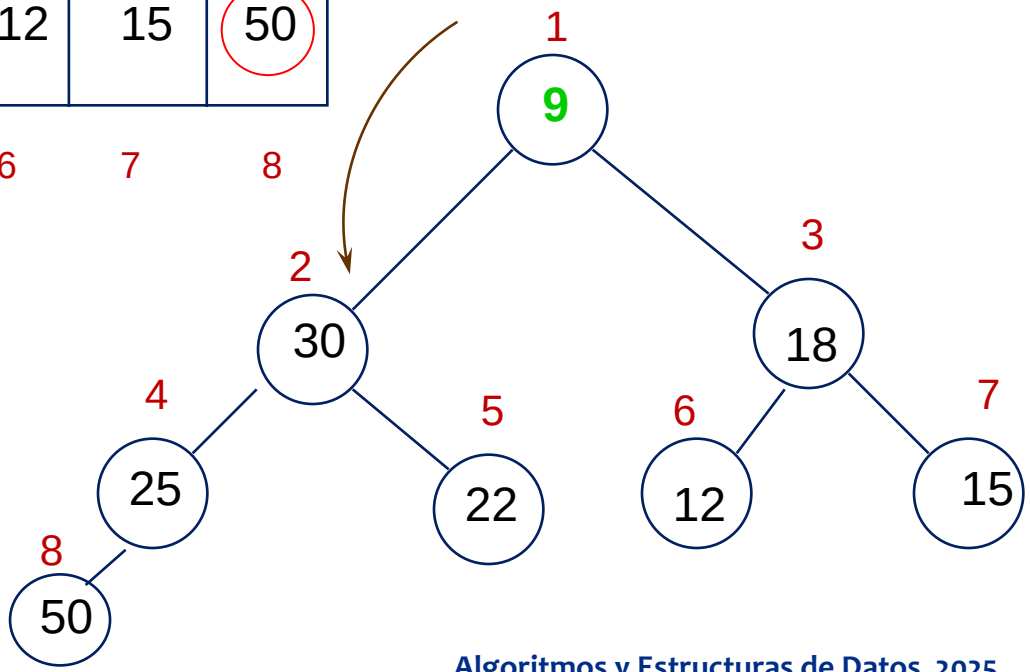


HeapSort (cont.)

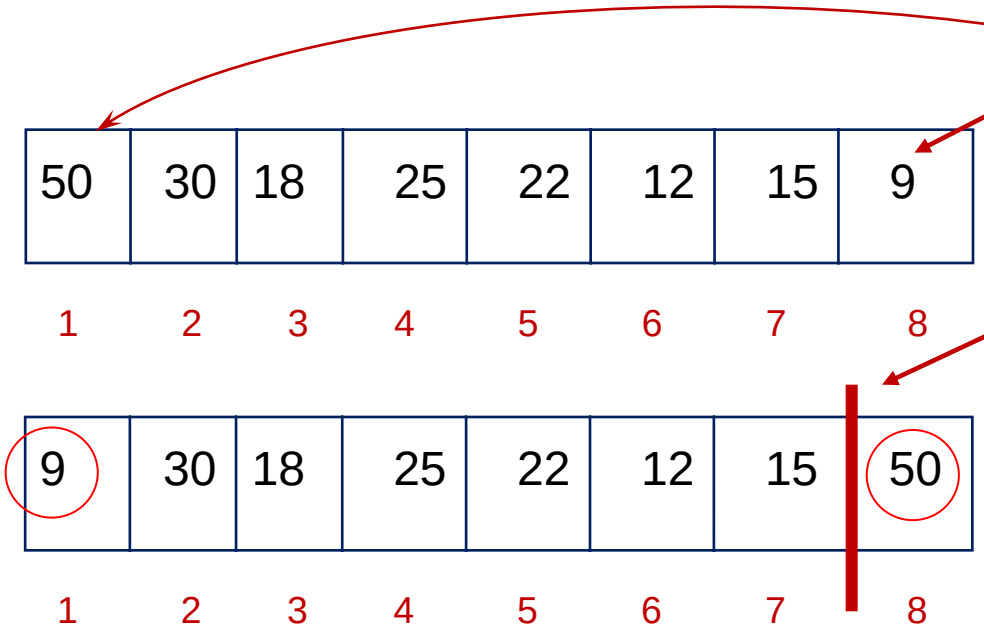


- Intercambio el primero con el último y

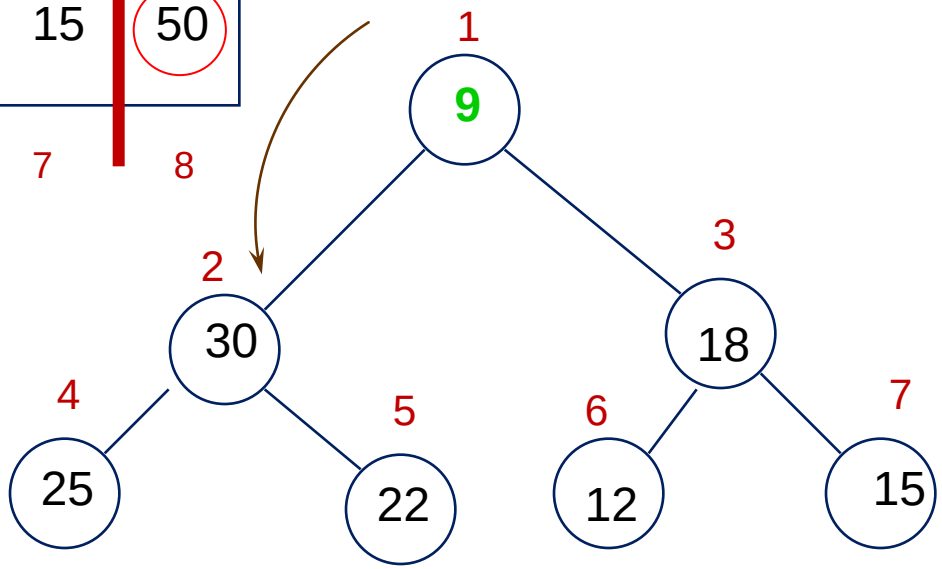




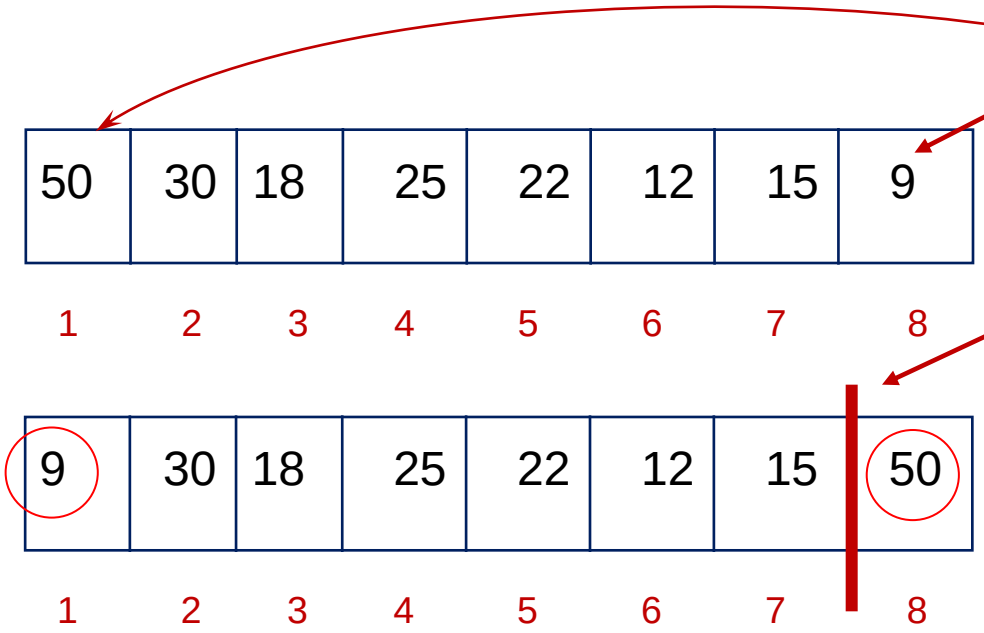
HeapSort (cont.)



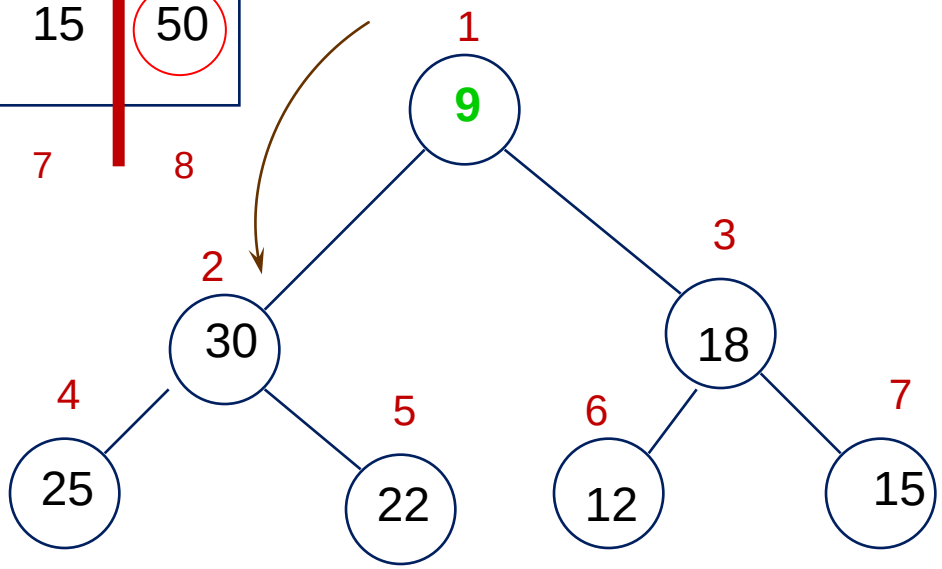
- Intercambio el primero con el último y
- decremento el tamaño



HeapSort (cont.)

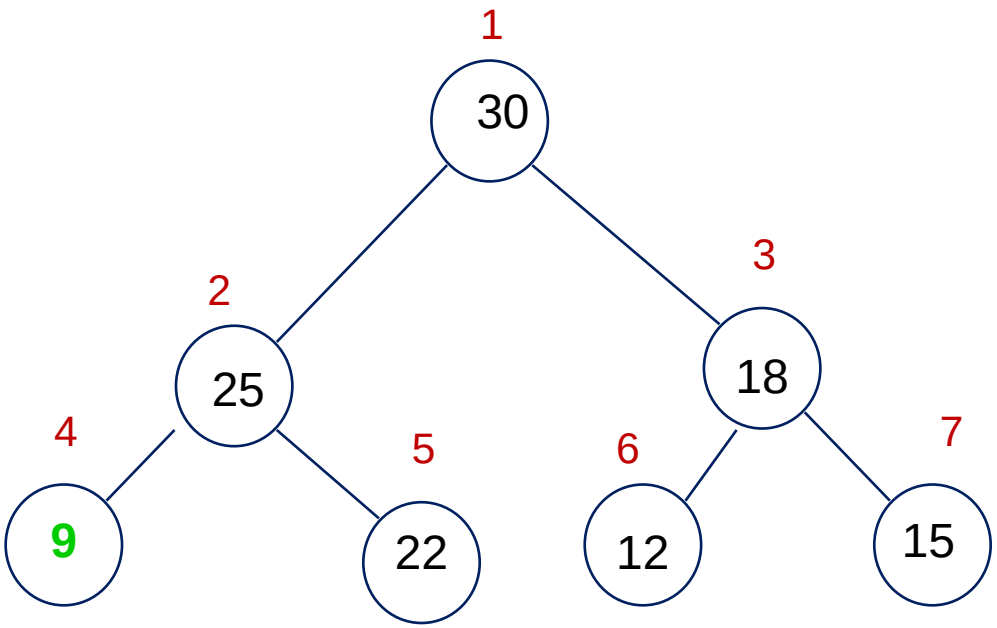


- Intercambio el primero con el último y
- decremento el tamaño

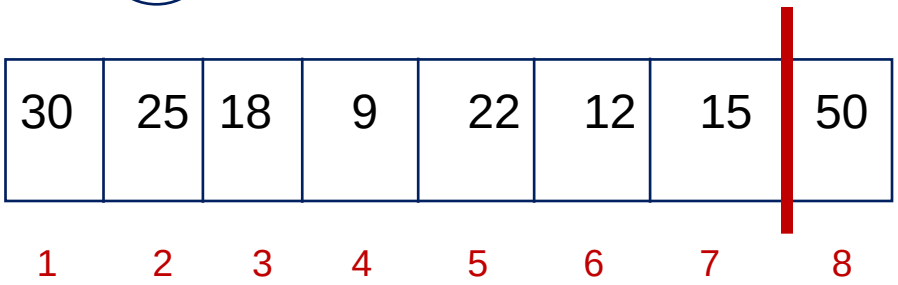


Filtrar el **9**

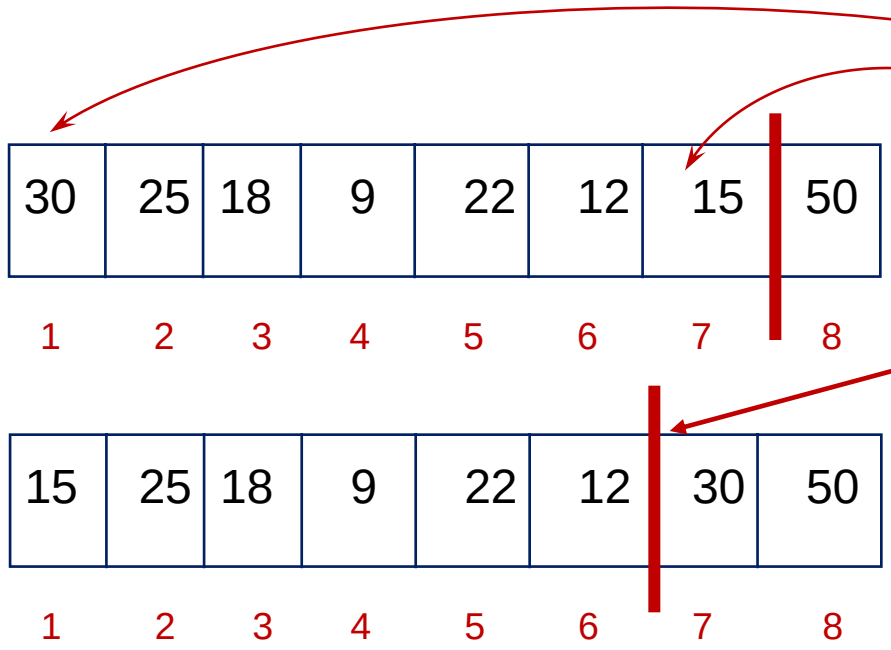
HeapSort (cont.)



Después de filtrar el **9** hacia abajo

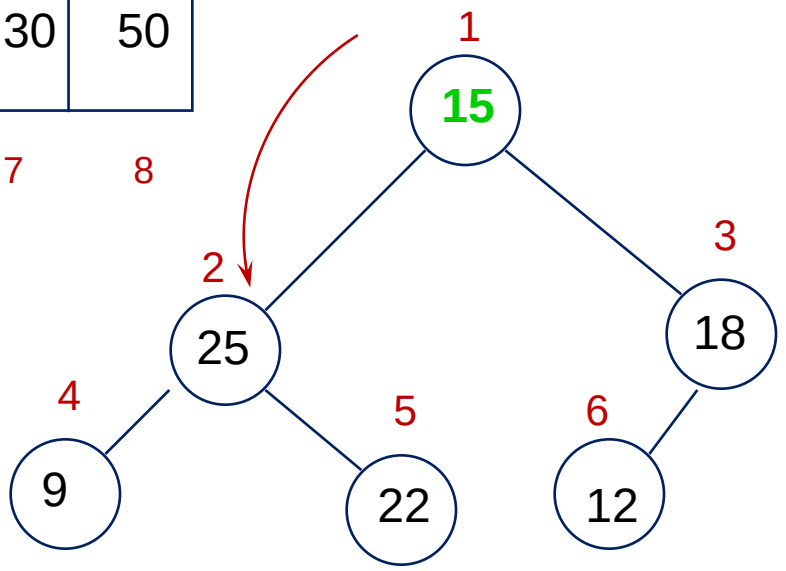


HeapSort (cont.)

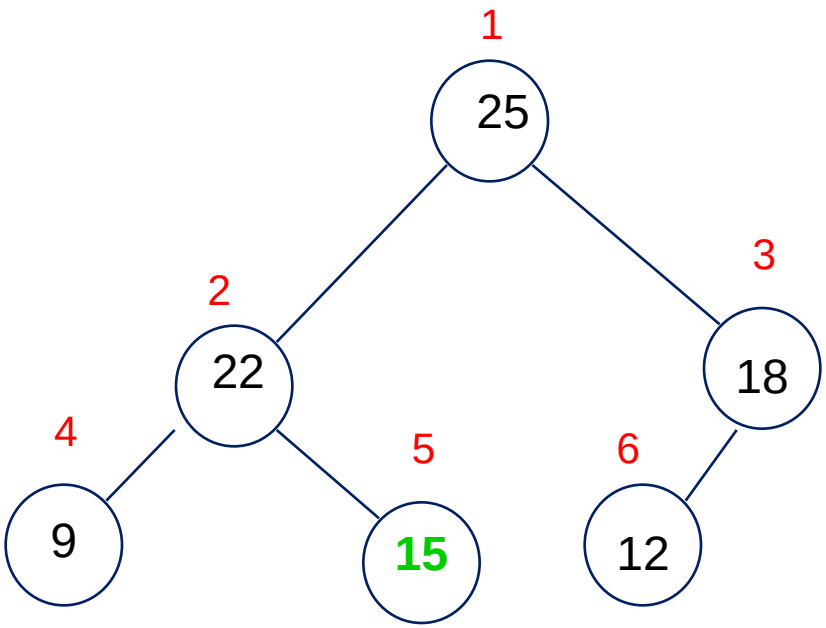


- Intercambio el primero con el último y
- decremento el tamaño

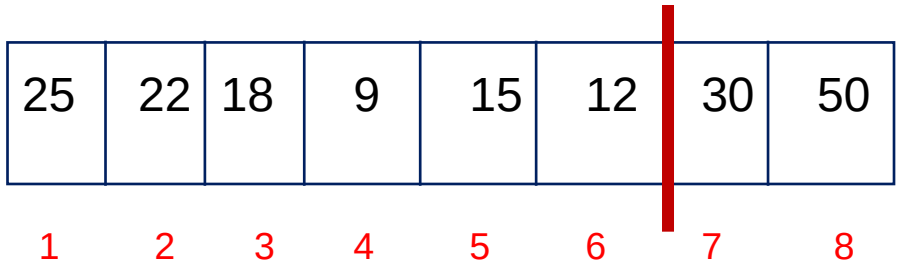
Filtrar el **15**



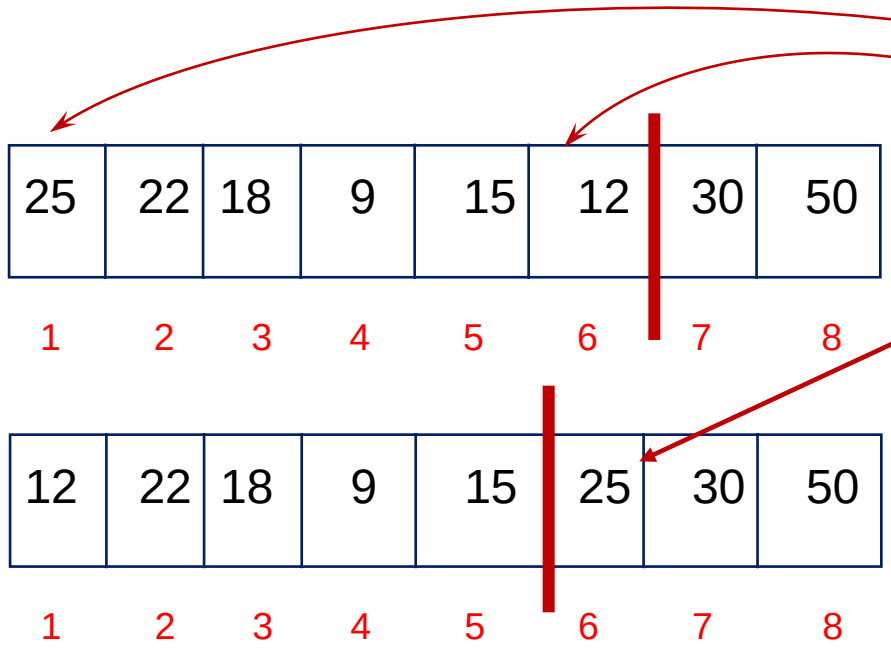
HeapSort (cont.)



Después de filtrar el **15** hacia abajo

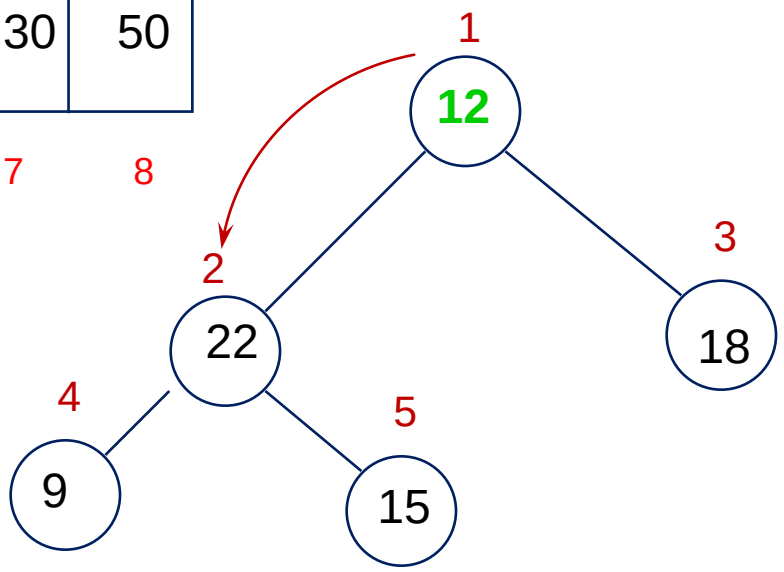


HeapSort (cont.)



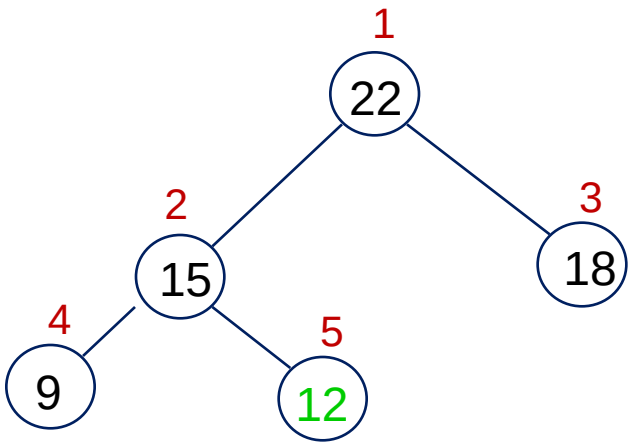
- Intercambio el primero con el último y
- decremento el tamaño

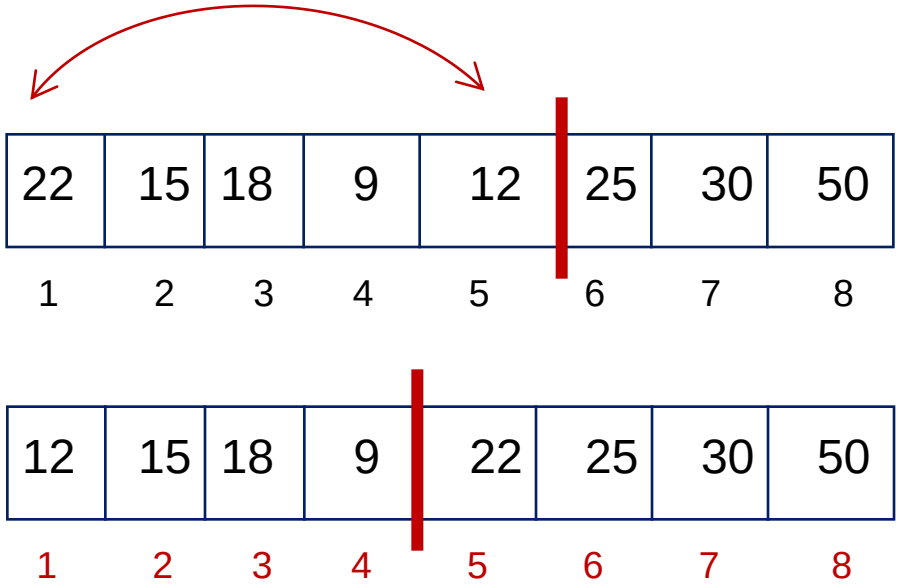
Filtrar el **12**

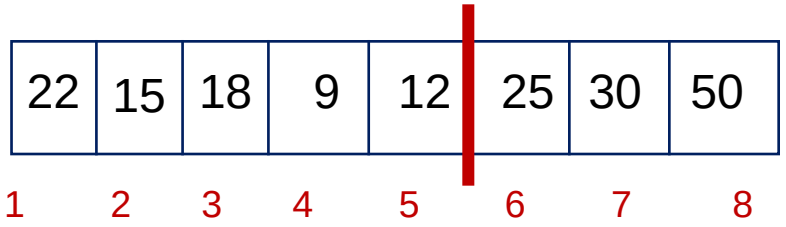


HeapSort (cont.)

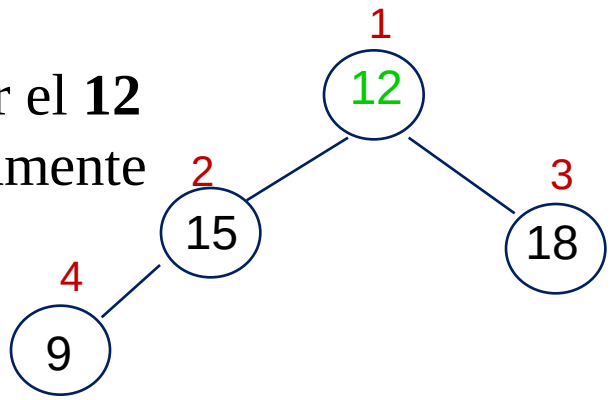
Después de filtrar el **12** hacia abajo





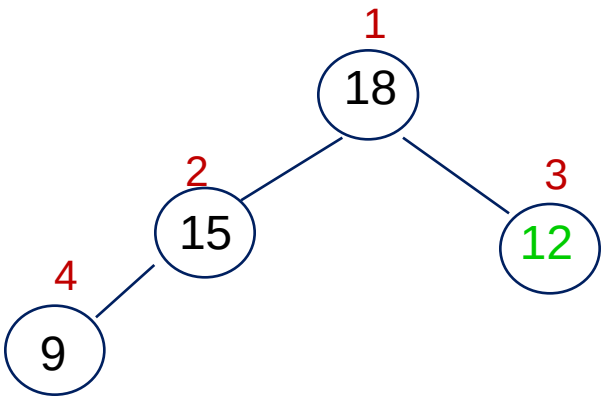


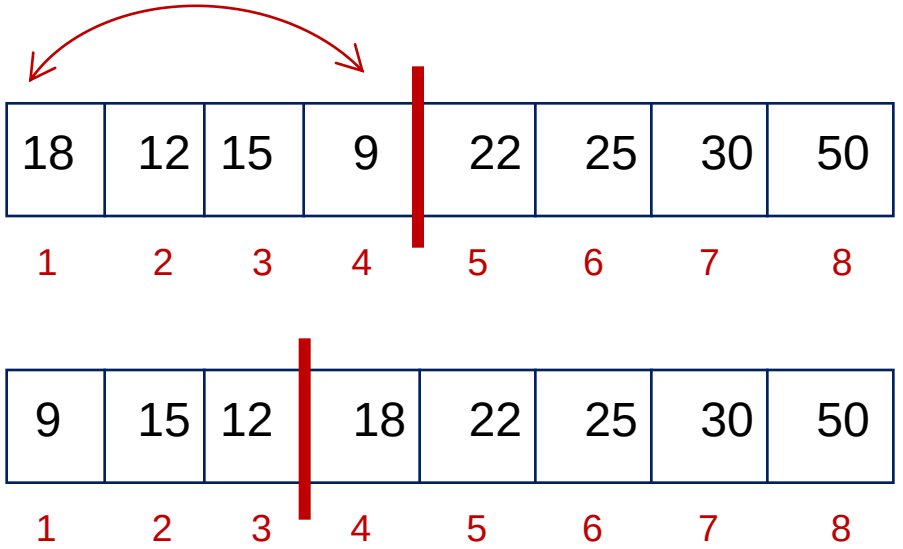
Filtrar el **12** nuevamente

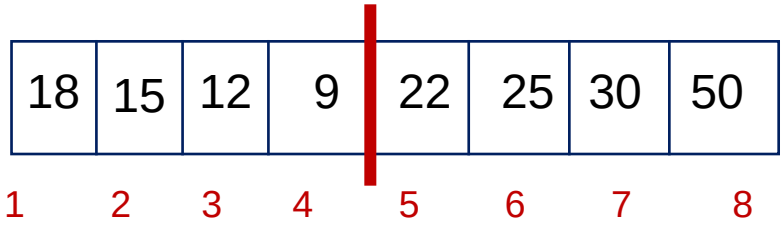


HeapSort (cont.)

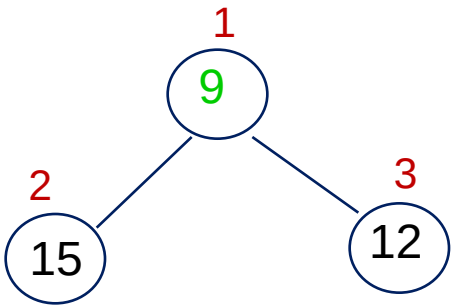
Después de filtrar el **12** hacia abajo





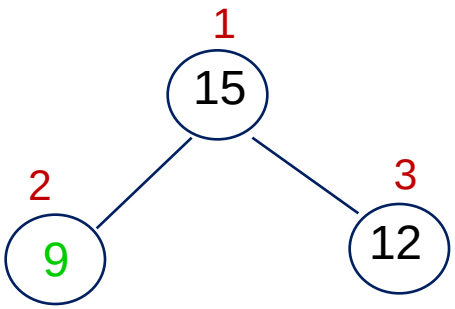


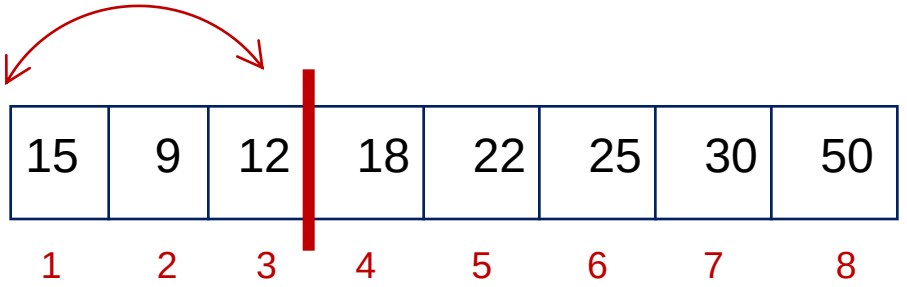
Filtrar el **9**

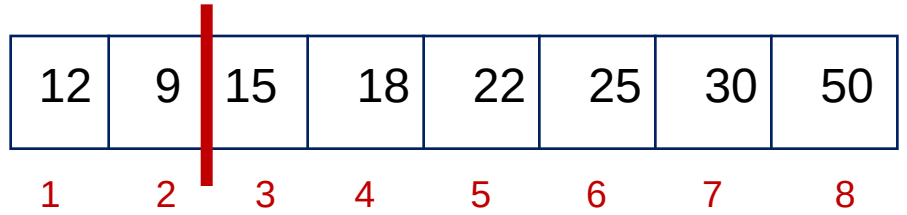


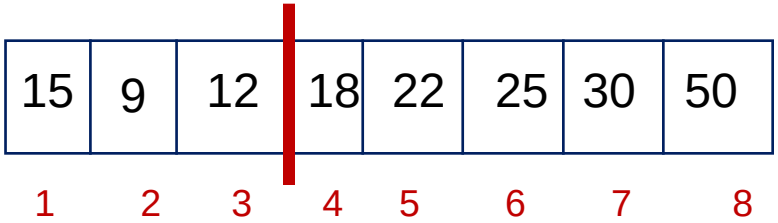
HeapSort (cont.)

Después de filtrar el **9** hacia abajo

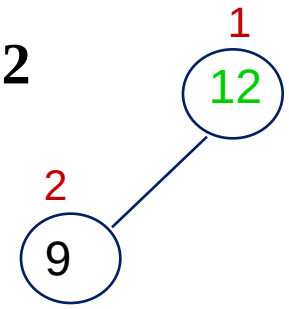






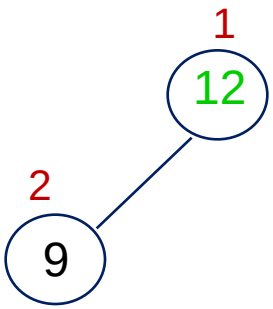


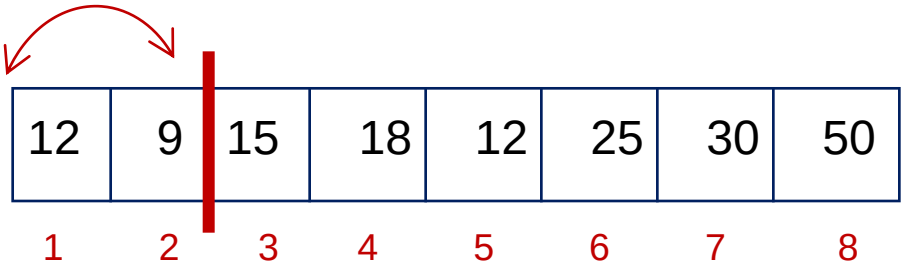
Filtrar el **12**

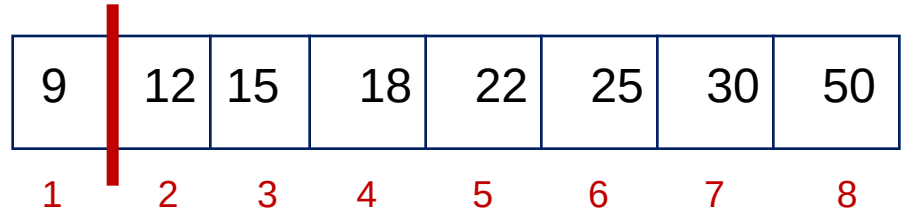


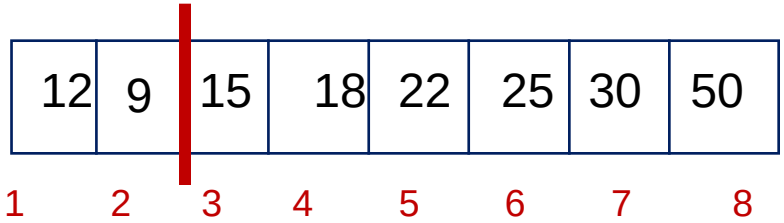
HeapSort (cont.)

Después de filtrar el **12** hacia abajo

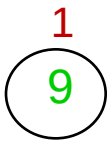








Filtrar el **9**



HeapSort (cont.)

Datos almacenados internamente

Heap conceptual