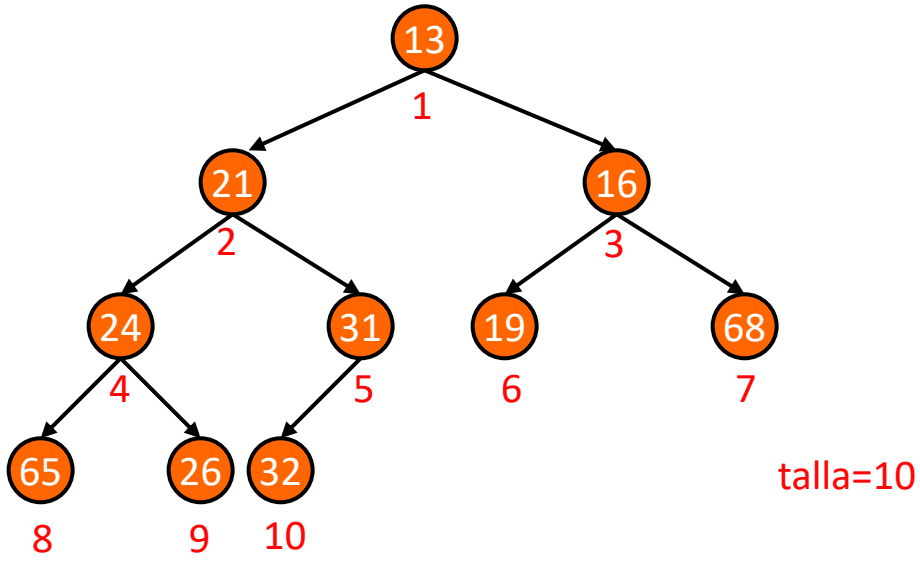
2. Montículo (*Heap*) Binario-*Representación minHeap (cont.)*

Representación

- elArray[1] representa a su Nodo Raíz
- si elArray[i] representa a su i-ésimo Nodo (Por Niveles)
  - Su Hijo Izquierdo es elArray[2i], si $2i \leq$ talla
  - Su Hijo Derecho es elArray[2i+1], si $2i + 1 \leq$ talla
  - Su Padre es elArray[i/2], excepto para i = 1

Propiedad de orden en un minHeap: elArray[padre(i)] <= elArray[i], 2<=i<=talla

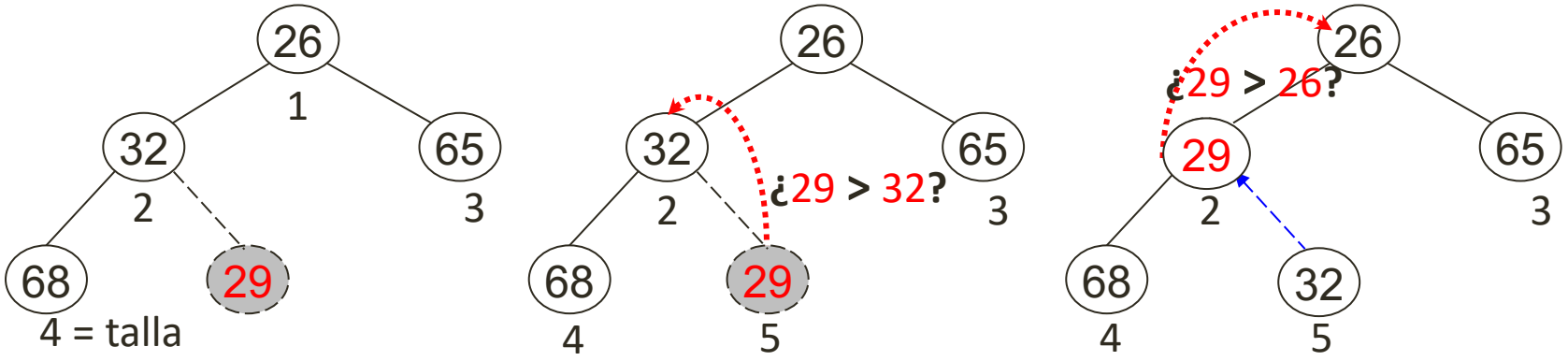


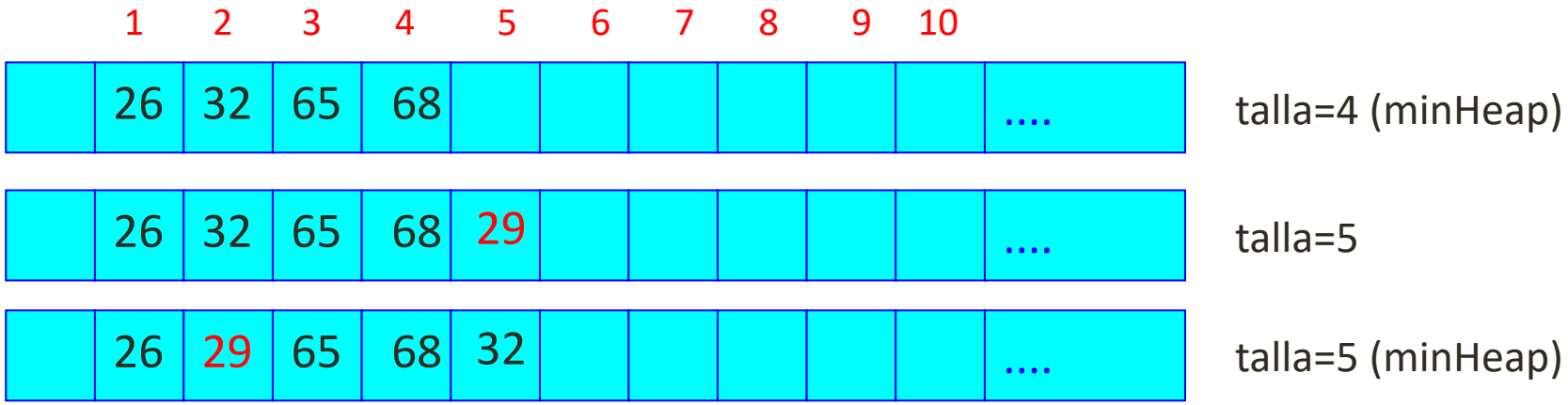
1	2	3	4	5	6	7	8	9	10	
	13	21	16	24	31	19	68	65	26	32	....

* E[] elArray
* int talla

3. La clase Java `MonticuloBinario` –

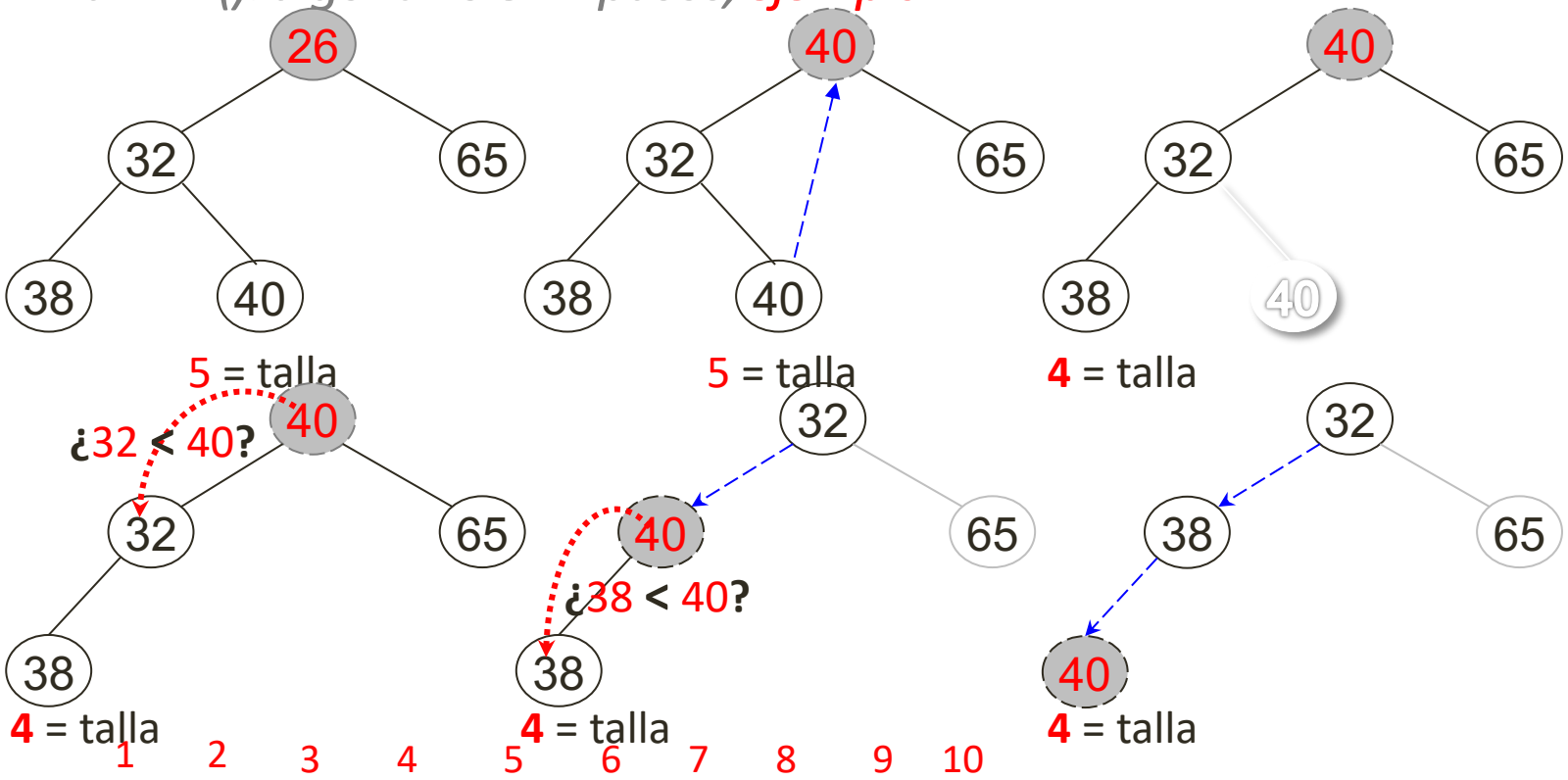
Método insertar(e): algoritmo en 2 pasos, ejemplo con e = 29





3. La clase Java `MonticuloBinario` – *Método*

eliminarMin(): algoritmo en 2 pasos, ejemplo 1



	1	2	3	4	5	6	7	8	9	10	
	26	32	65	38	40						....
	40	32	65	38							....
	32	40	65	38							....
	32	38	65	40							....

talla=5 (minHeap)

talla=4

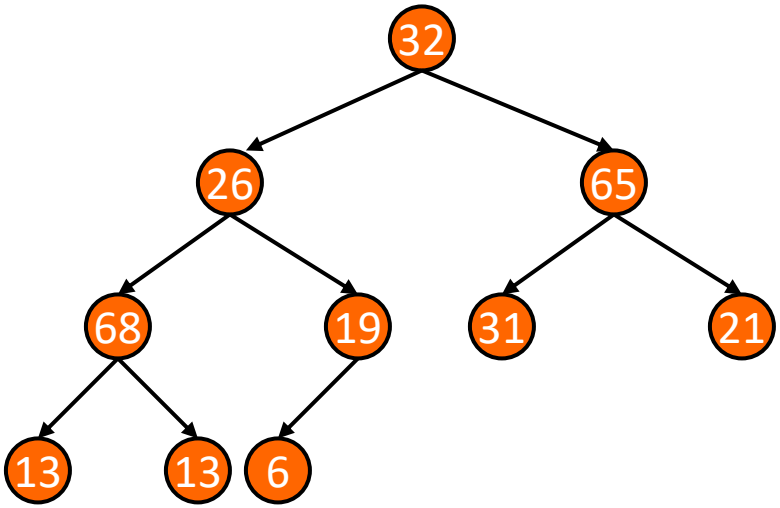
talla=4

talla=4 (minHeap)

3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)

Construir maxHeap a partir del vector: 32, 26, 65, 68, 19, 31, 21, 13, 13, 6



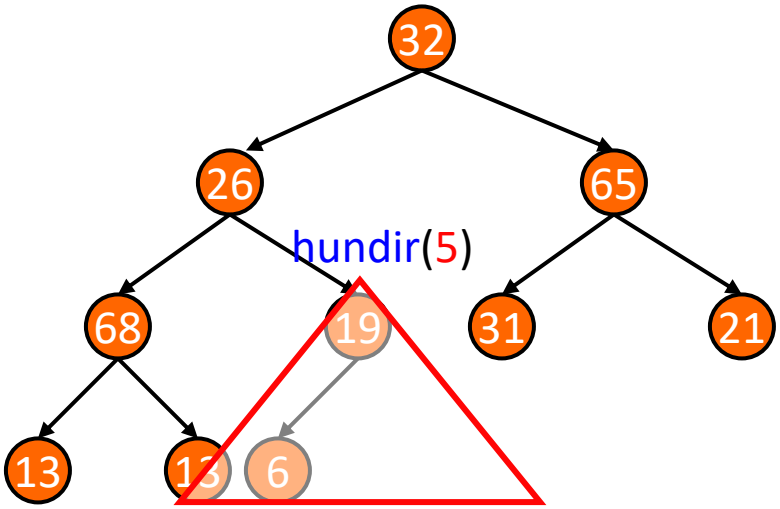
1	2	3	4	5	6	7	8	9	10	
	32	26	65	68	19	31	21	13	13	6	....

talla=10

- Restablece la propiedad de orden a partir de un Árbol Binario Completo para obtener un Montículo Binario
- Se basa en hundir los nodos en orden inverso al recorrido por niveles

3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)

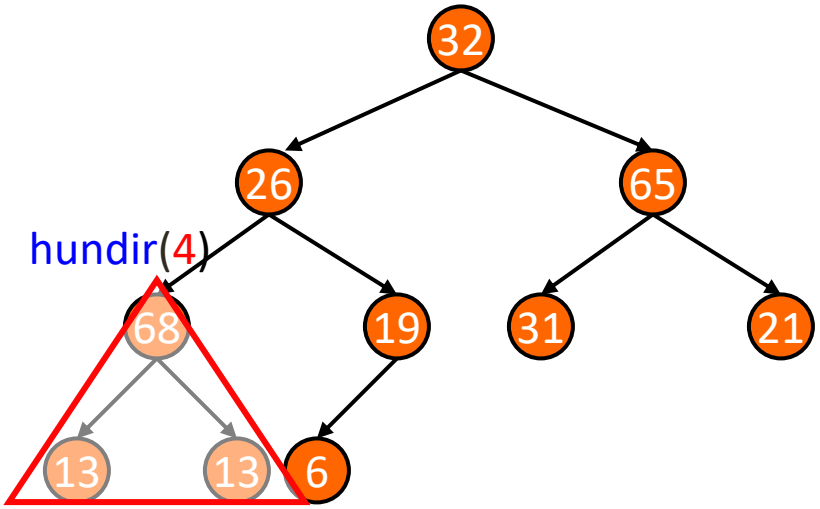


1	2	3	4	5	6	7	8	9	10	
	32	26	65	68	**19**	31	21	13	13	6	....

hundir(5)

3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)

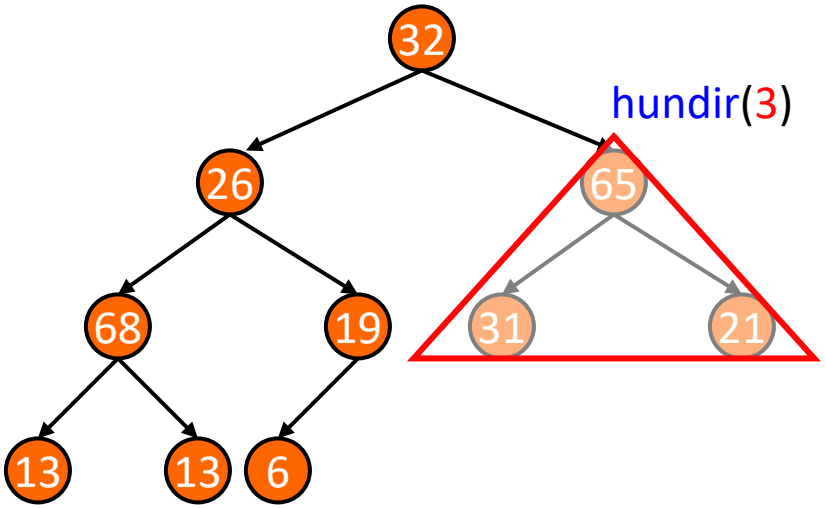


	1	2	3	4	5	6	7	8	9	10	
	32	26	65	**68**	19	31	21	13	13	6	....

hundir(4)

3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)

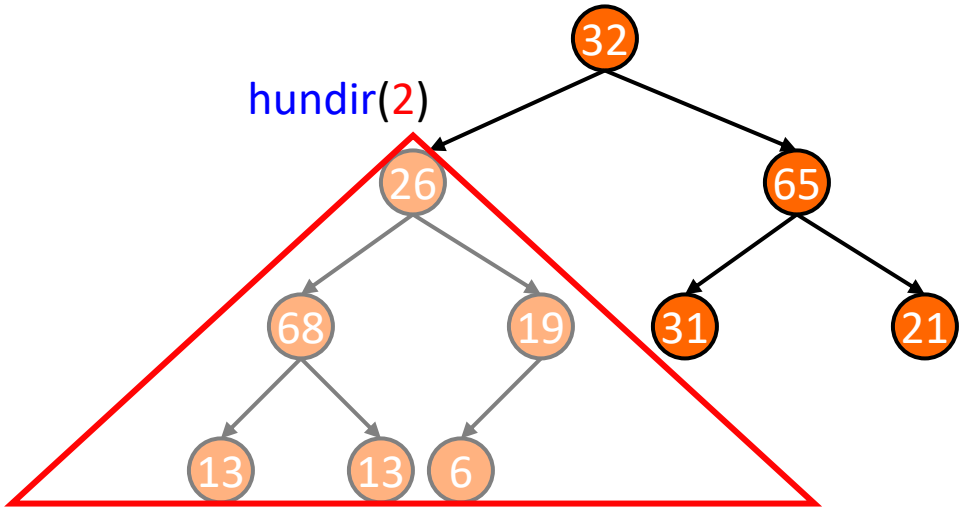


1	2	3	4	5	6	7	8	9	10	
	32	26	**65**	68	19	31	21	13	13	6	....

hundir(3)

3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



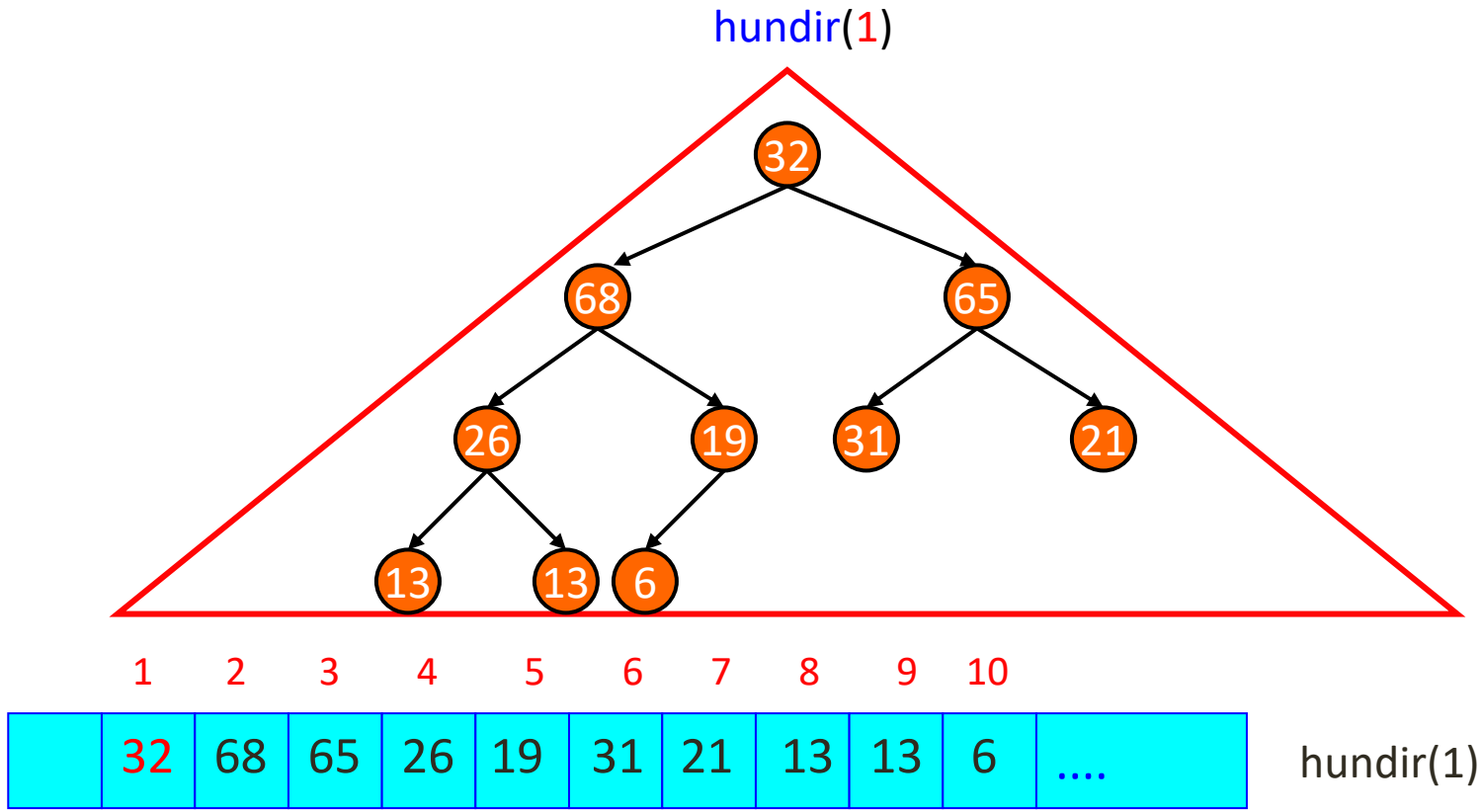
	1	2	3	4	5	6	7	8	9	10	
	32	**26**	65	68	19	31	21	13	13	6	....
	32	68	65	**26**	19	31	21	13	13	6	....

hundir(2)

hundir(4)

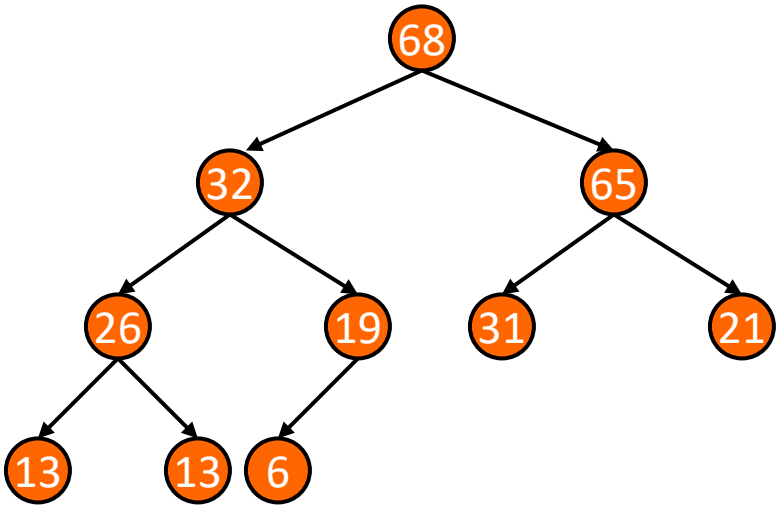
3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)



3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)

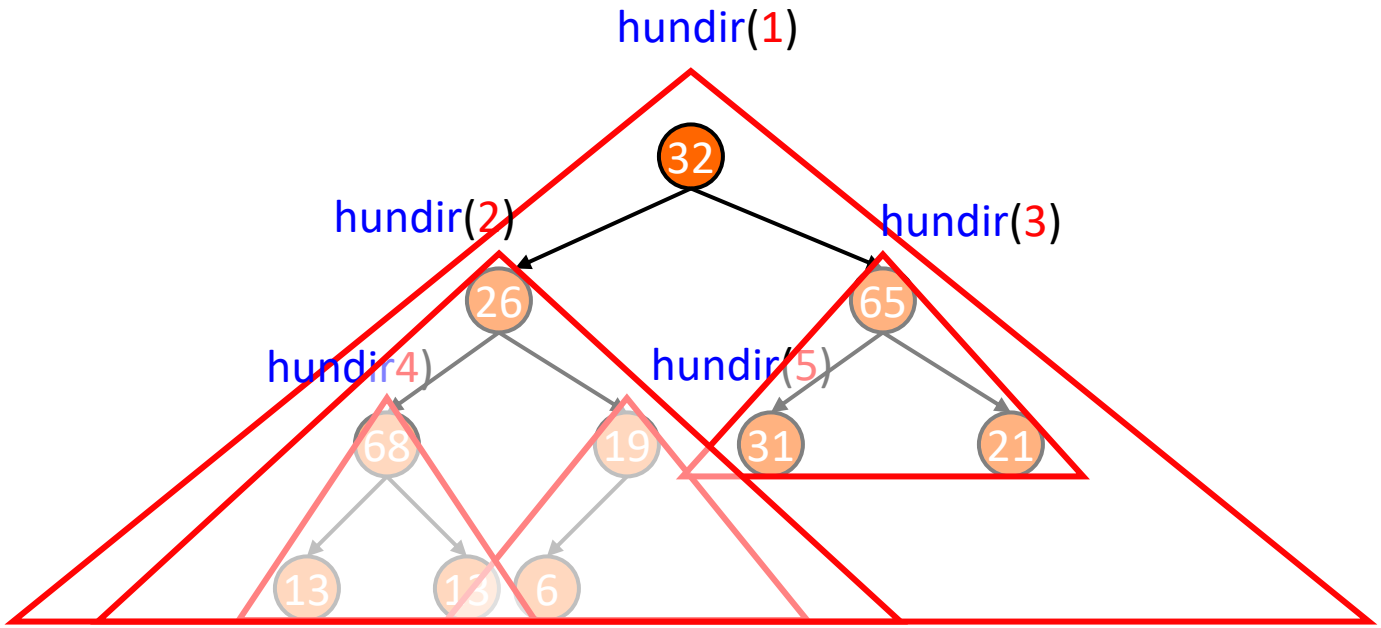


1	2	3	4	5	6	7	8	9	10	
	68	32	65	26	19	31	21	13	13	6	....

maxHeap

3. La clase Java `MonticuloBinario` -

Método arreglarMonticulo() iterativo (buildHeap)

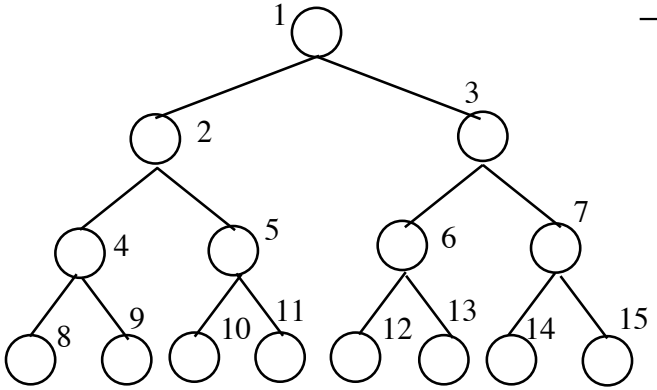



```
/*  Restablece la propiedad de orden de un Heap */
//  "hunde" Por-Niveles  y Descendente los nodos Internos
//  de elArray, pues las Hojas ya son Heaps
public void arreglar() {
     for (int i = talla / 2; i > 0; i --) {
          hundir(i);
     }
}
```

Método *arreglarMonticulo() iterativo (buildHeap) cont.*

Complejidad temporal:

- Cota NO ajustada: $O(n)$ llamadas a Heapify (hundir): $O(n \log n)$
- Cota ajustada: $O(n)$
  - Coste Heapify de un nodo es proporcional a su altura $O(h)$
  - Propiedad: En un Heap de n elementos hay, como mucho, $2^{\lfloor \log n \rfloor}/2^h$ nodos de altura h.



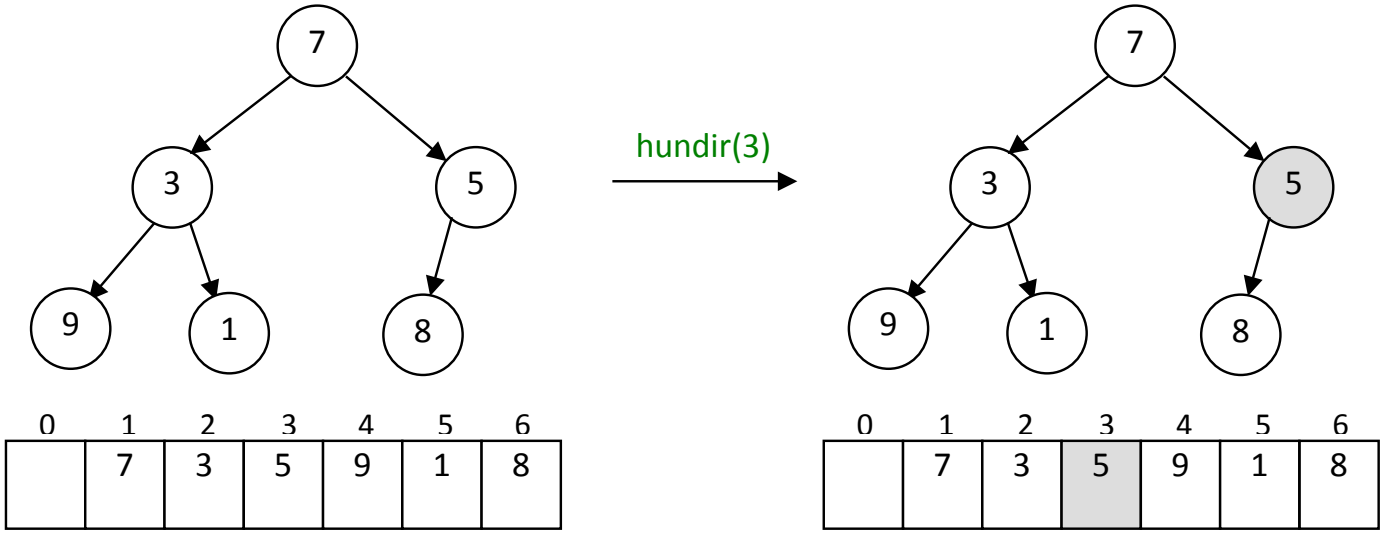
Altura	Número nodos	Coste cada llamada
h	1	h
$h-1$	2	$h-1$
$h-2$	4	$h-2$
$h-i$	2^i	$h-i$
1	2^{h-1}	1

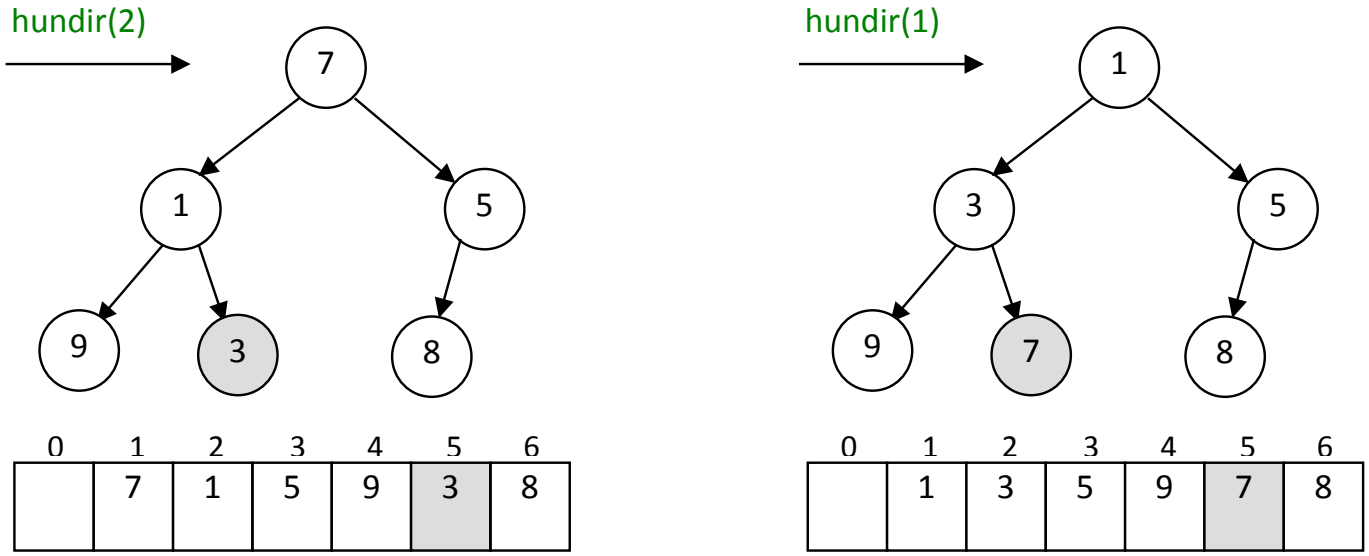
$$\text{Coste total} = \sum_{j=0}^{h-1} 2^j (h-j) \in O(2^h) = O(n)$$

3. La clase Java `MonticuloBinario` –

buildHeap - Ejemplo

Ejercicio: Hacer una traza del método arreglarMonticulo sobre el árbol binario completo [7, 3, 5, 9, 1, 8] para obtener un minHeap.


7
3
5
9
1
8
0 1 2 3 4 5 6
7 3 5 9 1 8
hundir(3)
7
3
5
9
1
8
0 1 2 3 4 5 6
7 3 5 9 1 8


hundir(2)
7
1
5
9
3
8
0 1 2 3 4 5 6
7 1 5 9 3 8
hundir(1)
1
3
5
9
7
8
0 1 2 3 4 5 6
1 3 5 9 7 8