¿Qué hemos visto?

Ordenación como **problema computacional**

Cotas inferiores para algoritmos que lo resuelven:

- Comparando e intercambiando elementos adyacentes: $\Omega(n^2)$
- Comparando e intercambiando elementos: $\Omega(n \cdot \log n)$

Resumen de algoritmos de ordenación

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
?	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(1)$

MergeSort, nuestro viejo amigo de matemáticas discretas

Y un último algoritmo que veremos esta semana

Pero primero…

Vamos a definir una estructura —un poco más "estructurada" que simplemente un arreglo— que nos permita almacenar datos según cierta prioridad

La idea es poder consultar a la estructura cuál es el más prioritario de los datos para poder procesarlos en orden de prioridad

La cola de prioridades

Una estructura de datos con las siguientes operaciones:

- Insertar un dato con una prioridad dada
- Extraer el dato con mayor prioridad

(E idealmente:

- Cambiar la prioridad de un dato ya insertado)

Propiedad de orden de la cola

Claramente, hay que mantener cierto orden de los datos

¿Cuál es el costo —en términos del número de operaciones o pasos básicos— de mantener los datos **ordenados**?

¿Y al llegar n datos nuevos?

¿Es necesario un orden total?

Solo necesitamos saber cuál es el dato más prioritario

Quizás podamos darnos el lujo de no tener un orden total —solo un orden parcial— de los datos

¡ Necesitamos algún tipo de estructura interna !

Cómo aprovechar la propiedad de orden parcial

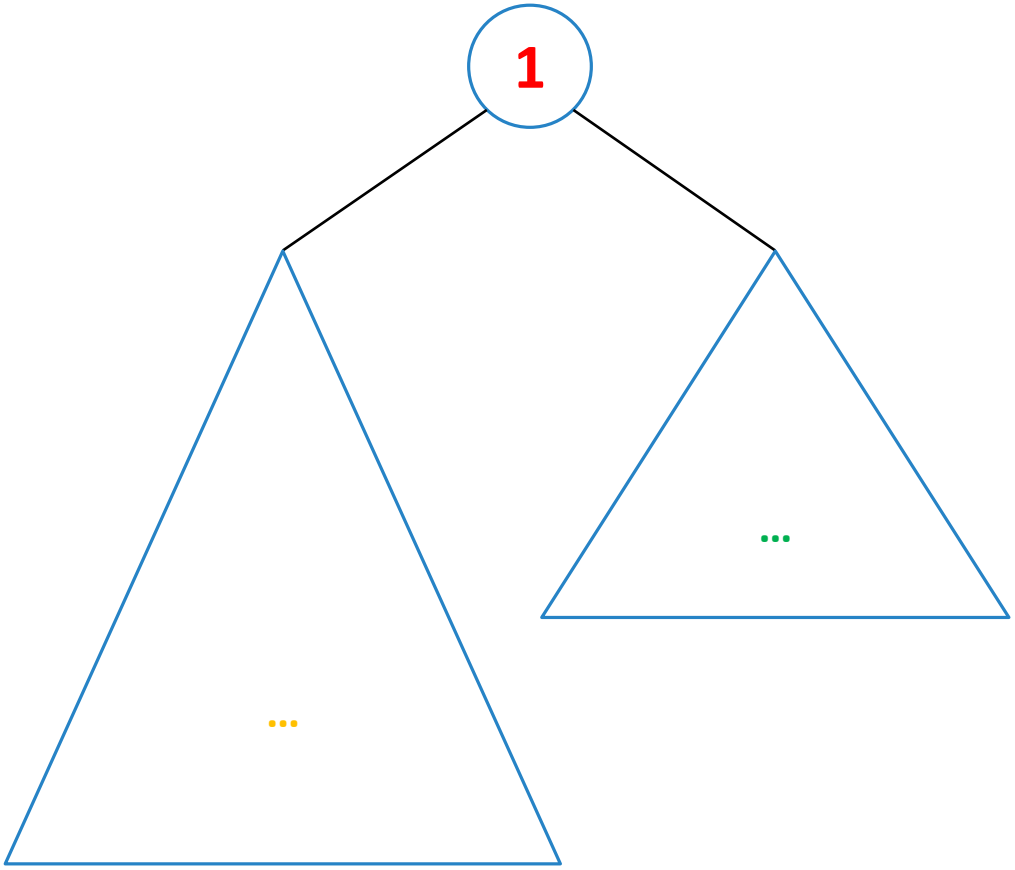
¿Qué información contenida en la estructura debe estar fácilmente disponible en todo momento?

¿Será posible hacer una estructura recursiva?

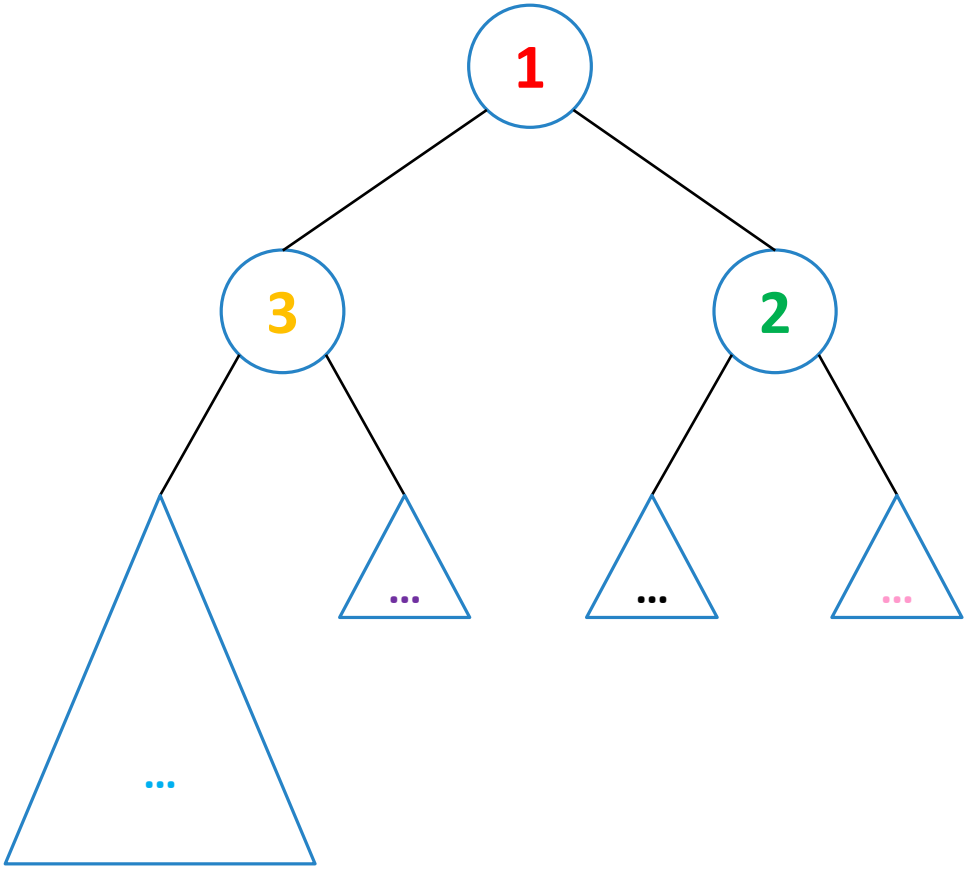
¿Por qué querríamos tener estructuras recursivas?

- los algoritmos para recorrerlas o buscar información en ellas son también recursivos y, por lo tanto, más simples
- la implementación de la estructura se simplifica

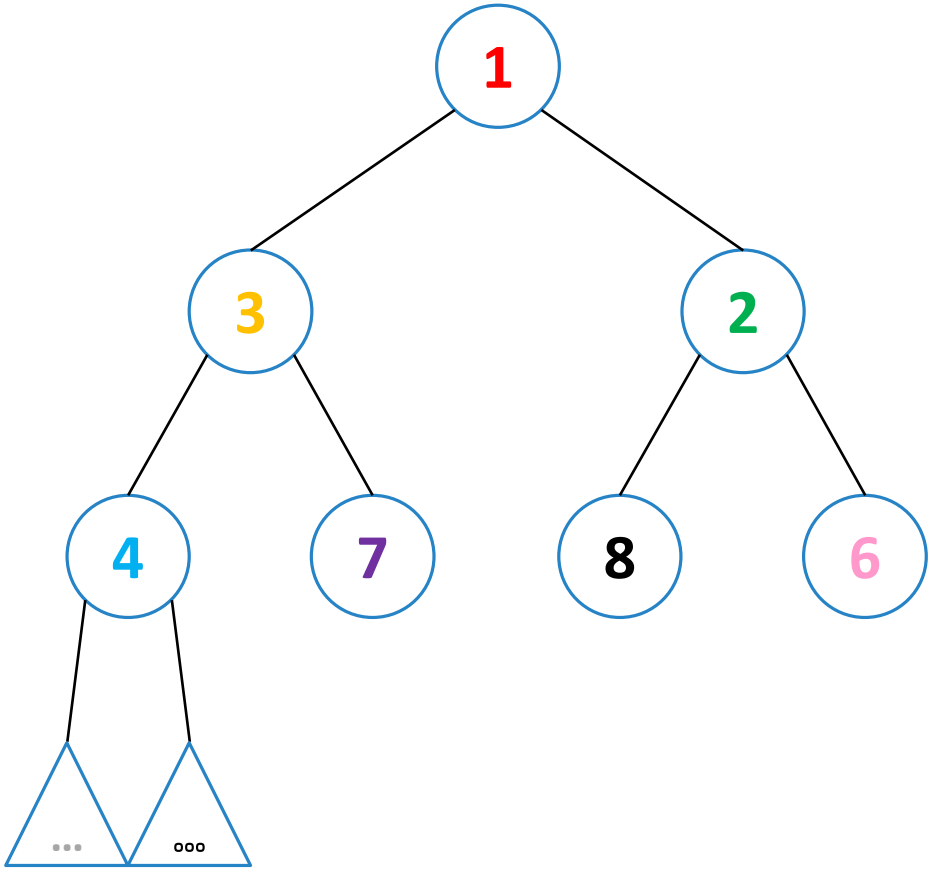
Un heap binario: La raíz y sus dos hijos



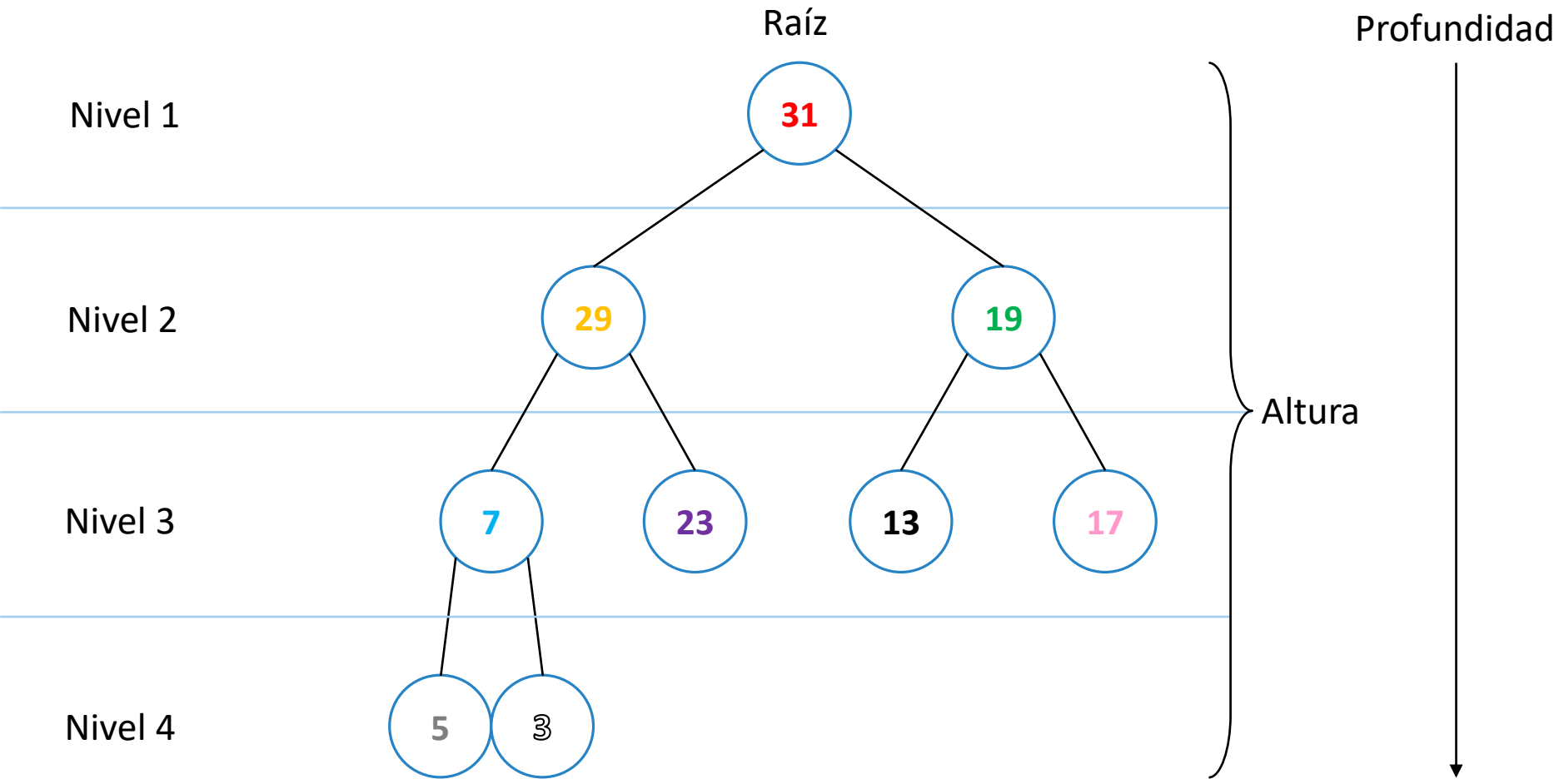
Un heap binario: Recursivamente



Un heap binario: Recursivo a todo nivel



Anatomía de un heap binario



El *heap* binario: Una estructura recursiva

Es un **árbol binario**, con el elemento más prioritario como raíz

Los demás datos están divididos en dos grupos:

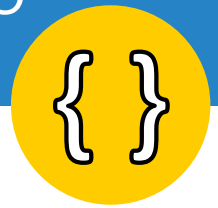
- cada grupo está organizado a su vez —recursivamente— como un heap binario
- estos dos heaps binarios cuelgan de la raíz como sus hijos

Altura de un heap binario

¿Cuál es la altura de un heap con n datos?

¿Cómo podemos garantizar que se mantenga lo más baja posible?

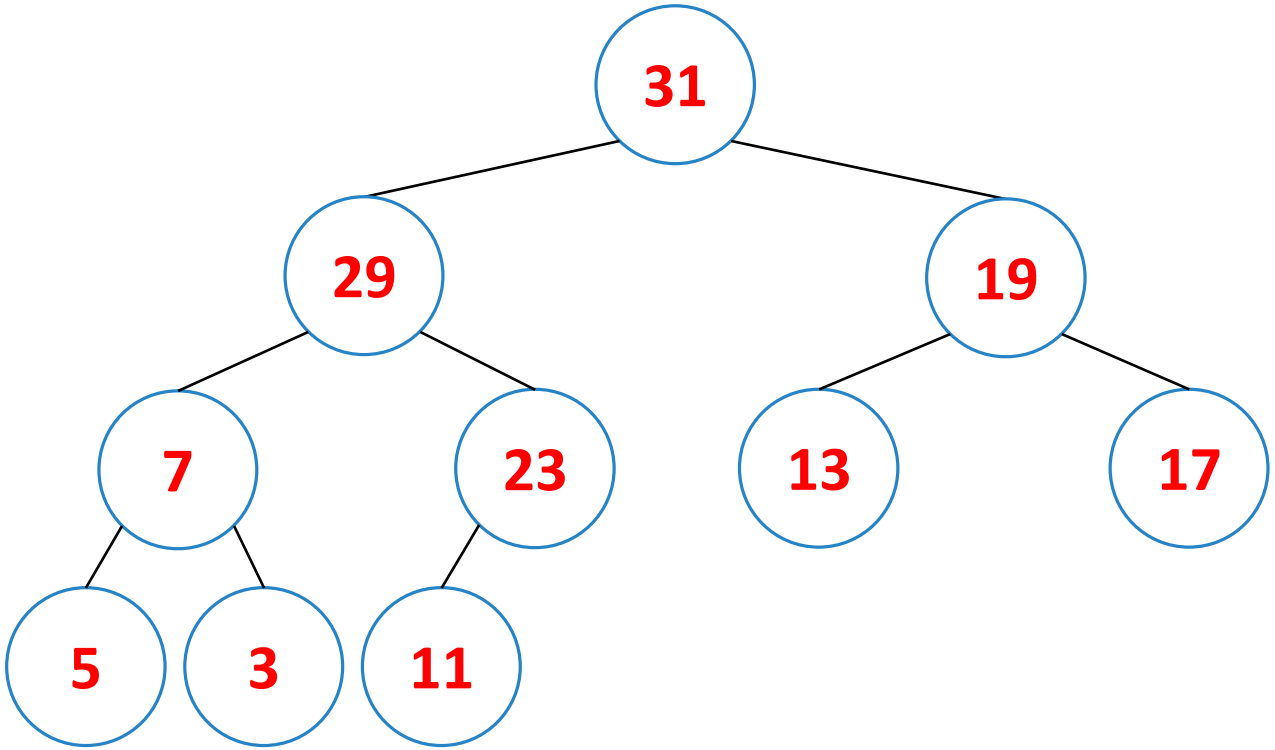
Una implementación simple de un heap binario



Si podemos suponer una cantidad máxima de datos que pueden estar en el heap simultáneamente

… entonces, es posible implementar el heap de forma compacta en un **arreglo**

Un heap binario como un arreglo



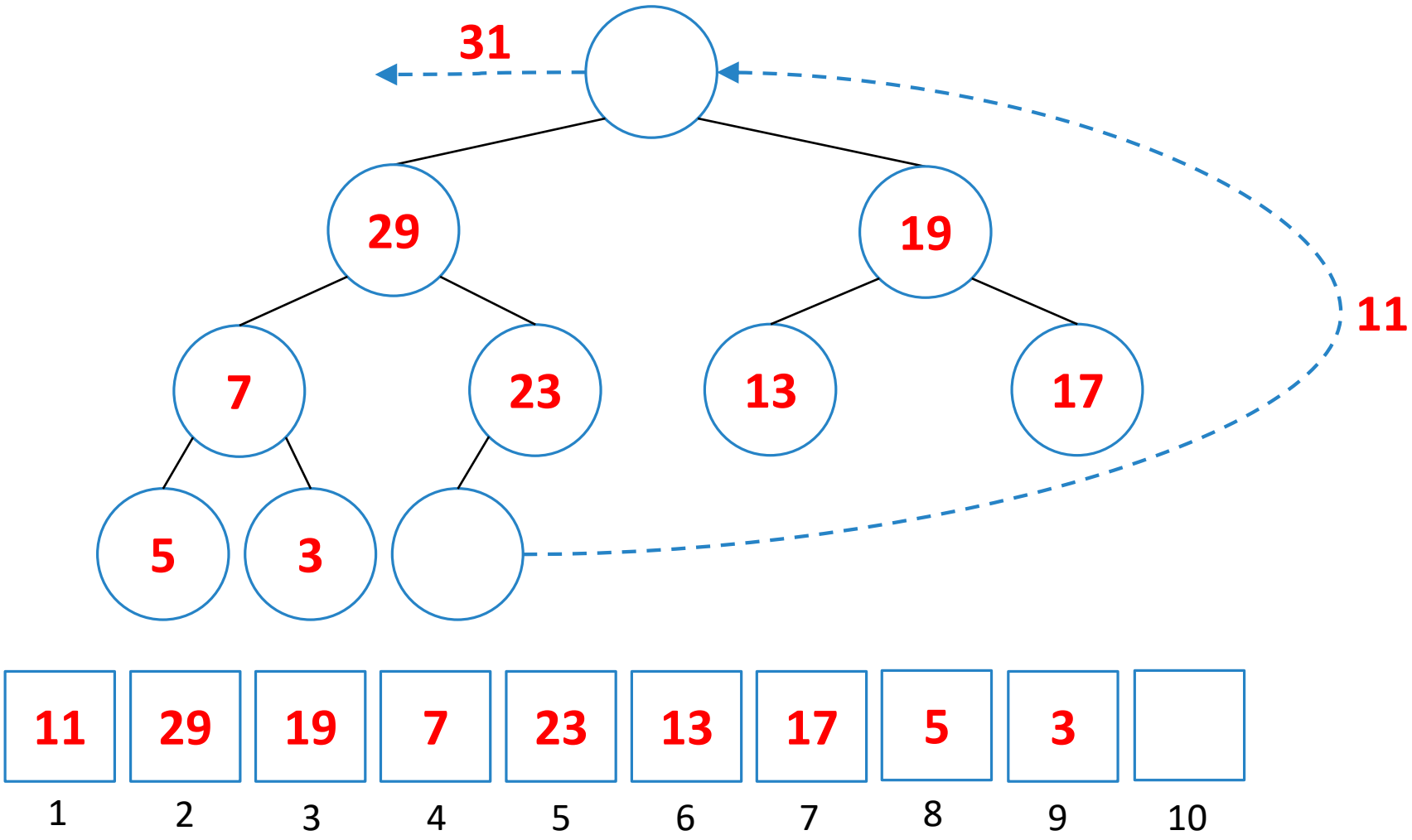
31	29	19	7	23	13	17	5	3	11
1	2	3	4	5	6	7	8	9	10

Operaciones de un heap

Queremos definir las operaciones para insertar y extraer

Al insertar y extraer elementos, el heap debe reestruc-

turarse para que recupere sus propiedades


31
29
19
11
7
23
13
17
5
3
11
29
19
7
23
13
17
5
3
1
2
3
4
5
6
7
8
9
10

$\boldsymbol{extract}(\boldsymbol{H})$:

$\boldsymbol{i} \leftarrow$ la última celda no vacía de $\boldsymbol{H}$

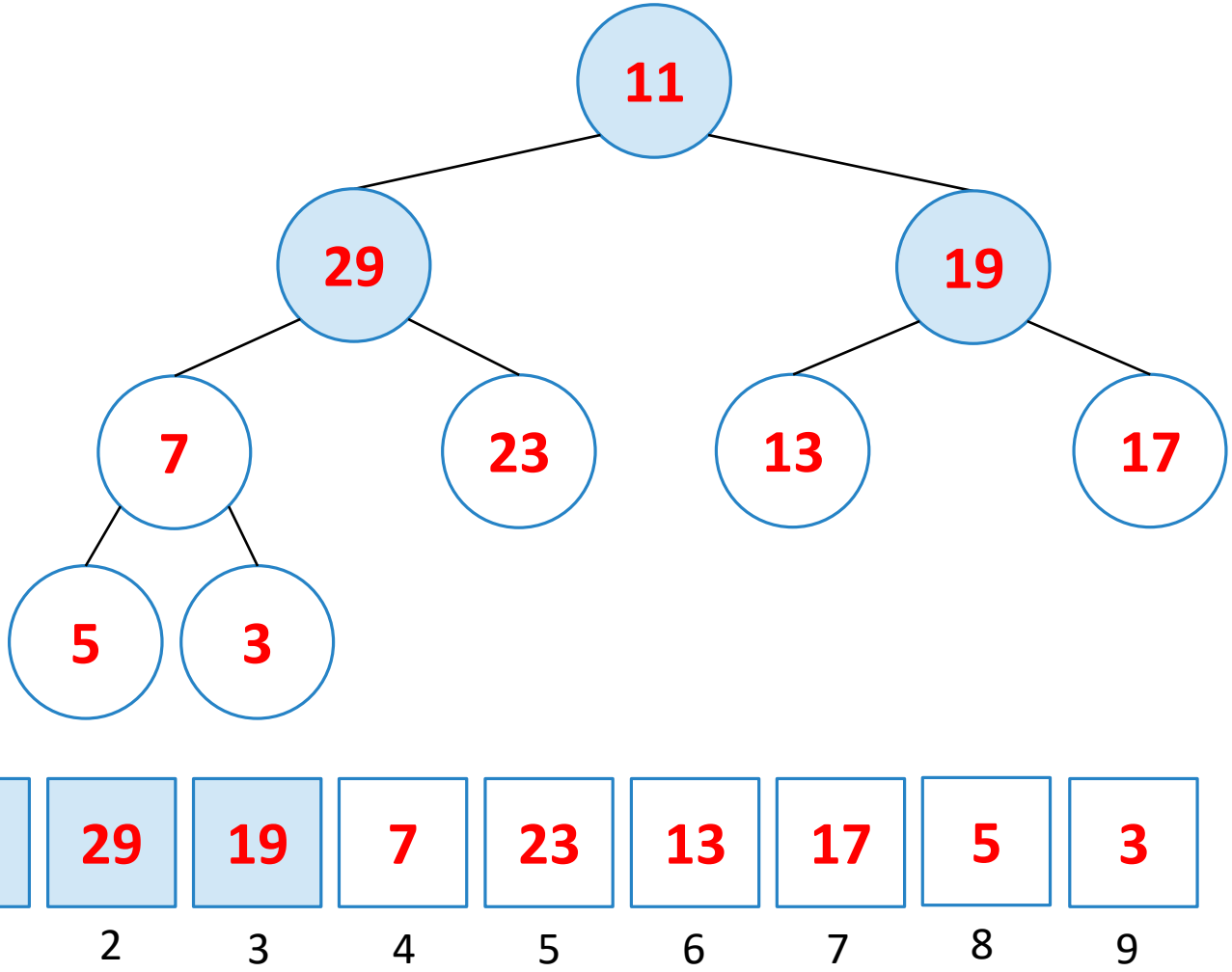
$\boldsymbol{best} \leftarrow \boldsymbol{H}[\mathbf{1}]$

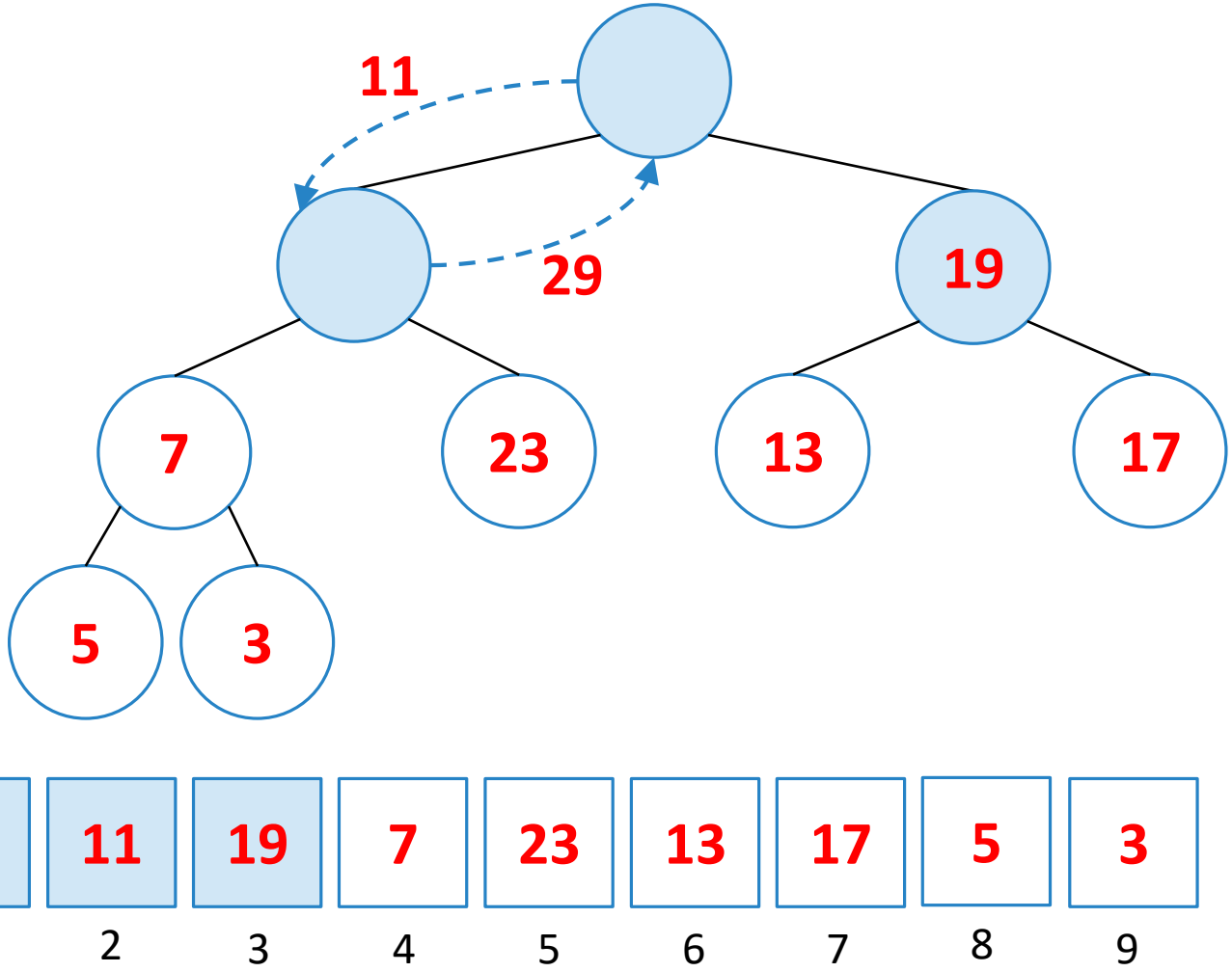
$\boldsymbol{H}[\mathbf{1}] \leftarrow \boldsymbol{H}[\boldsymbol{i}]$

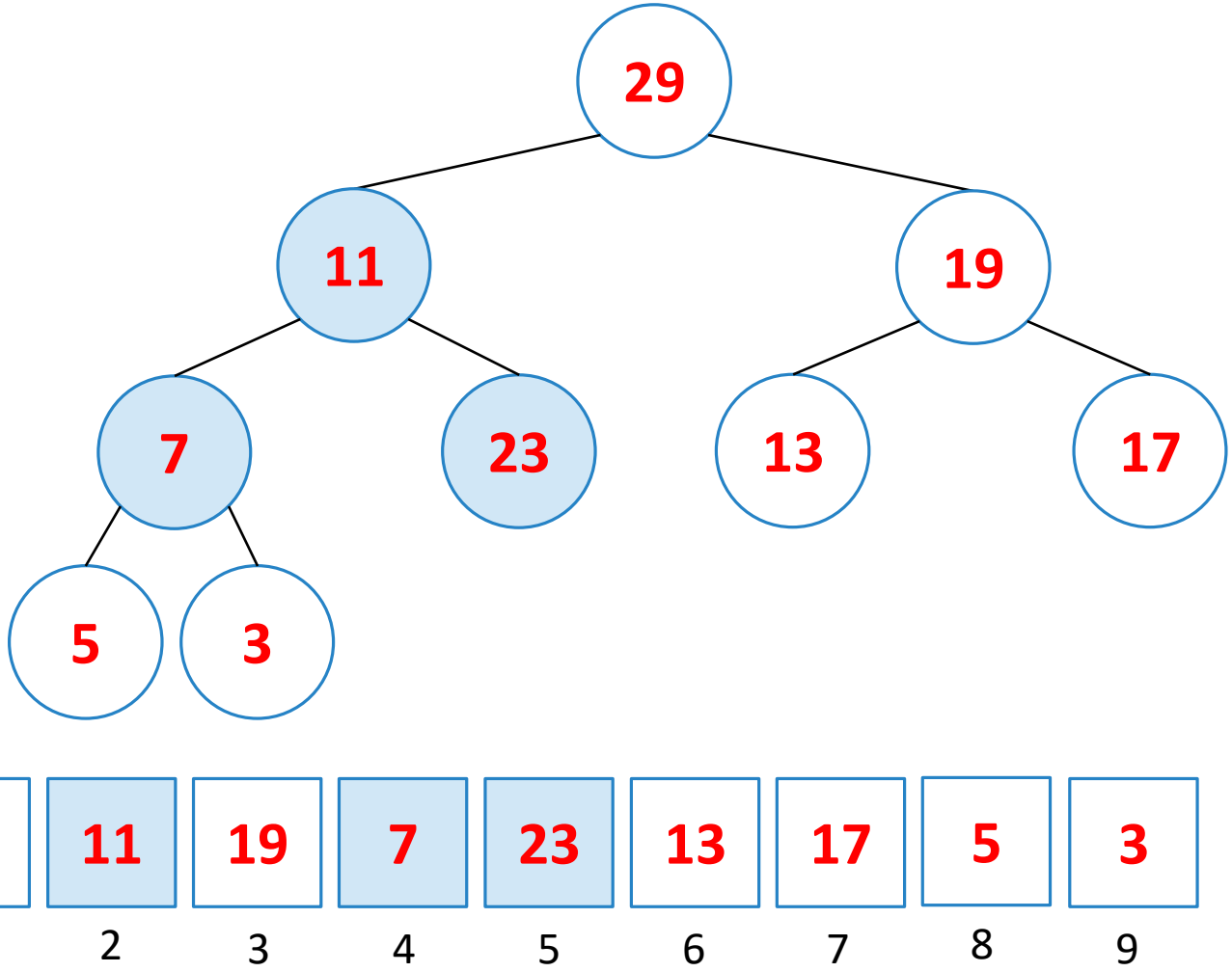
$\boldsymbol{H}[\boldsymbol{i}] \leftarrow \emptyset$

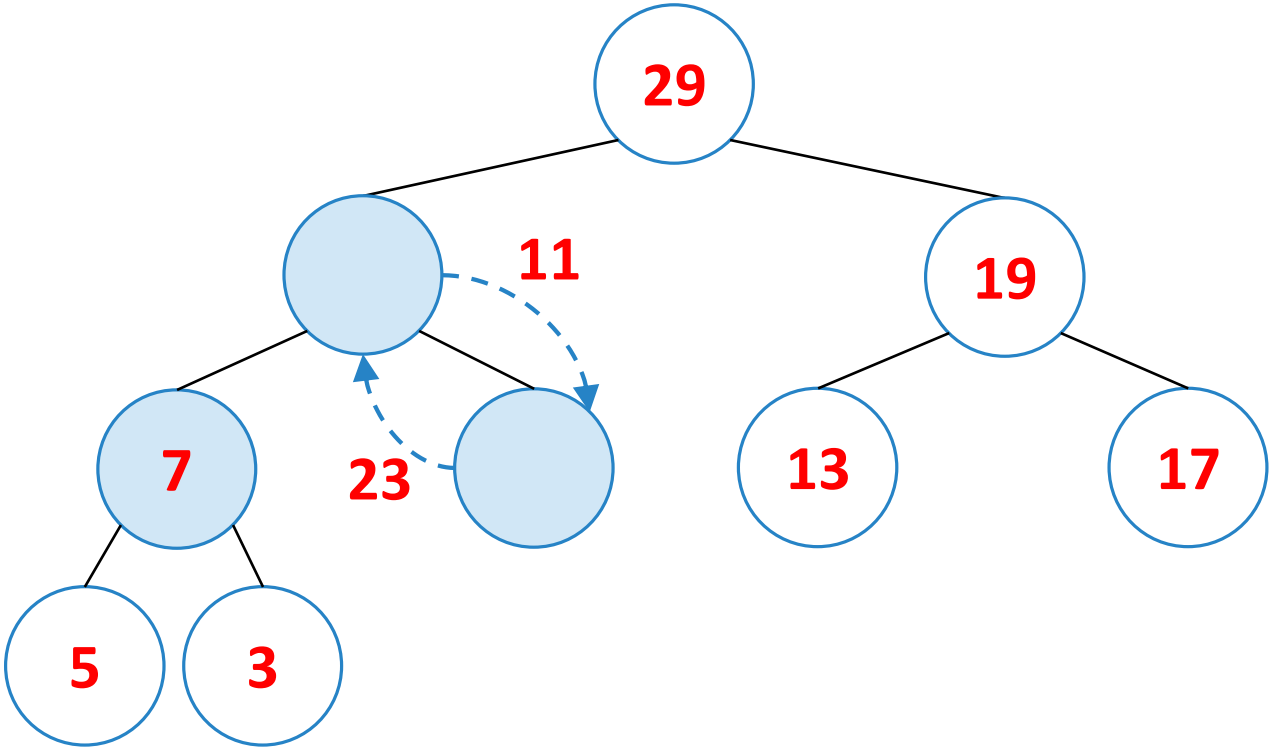
$\boldsymbol{sift\ down}(\boldsymbol{H}, \mathbf{1})$

$\boldsymbol{return\ best}$

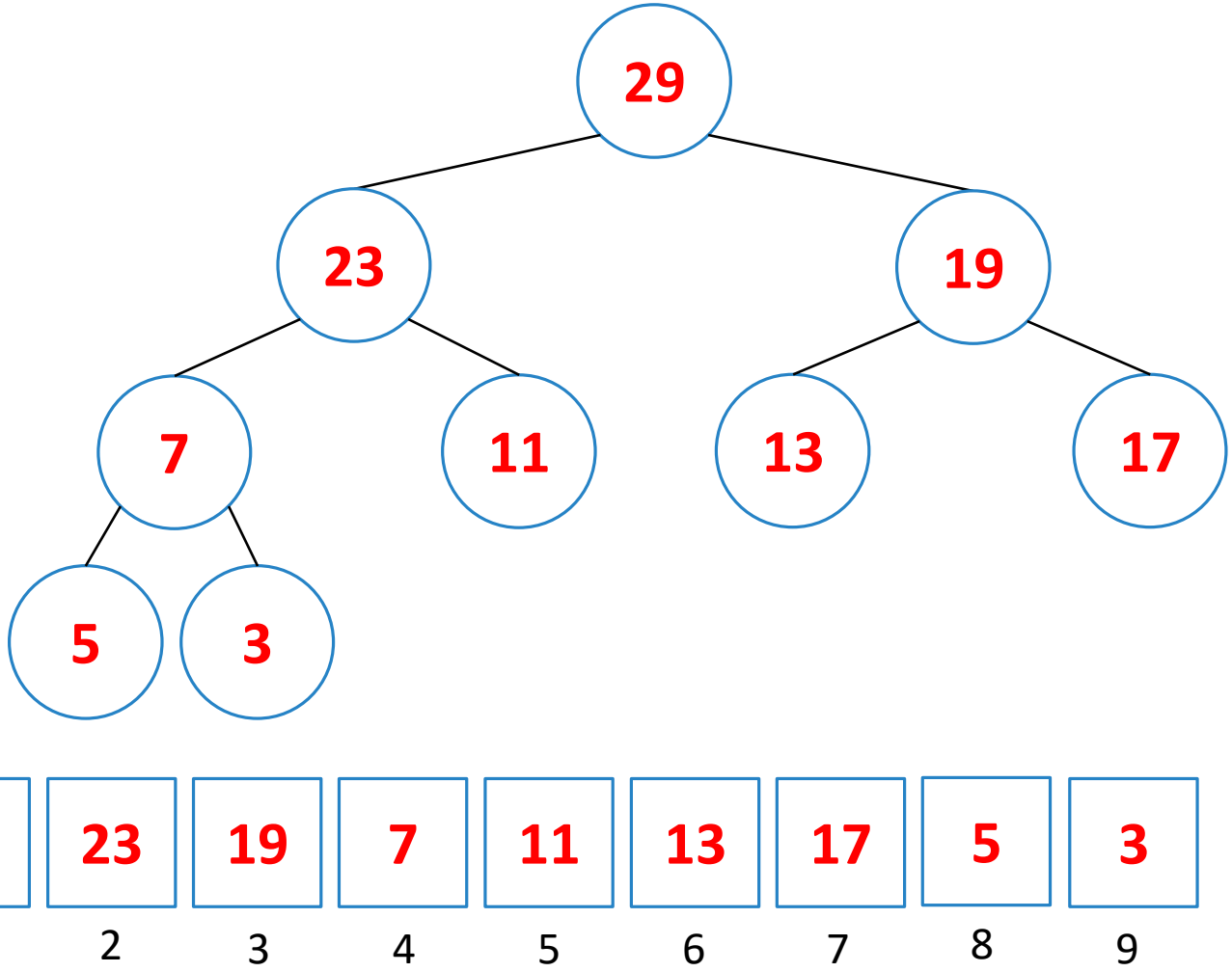

11
29
19
7
23
13
17
5
3
11
29
19
7
23
13
17
5
3
1
2
3
4
5
6
7
8
9


11
29
19
7
23
13
17
5
3
29
11
19
7
23
13
17
5
3
1
2
3
4
5
6
7
8
9


29
11
19
7
23
13
17
5
3
29
11
19
7
23
13
17
5
3
1
2
3
4
5
6
7
8
9



29	23	19	7	11	13	17	5	3
1	2	3	4	5	6	7	8	9


29
23
19
7
11
13
17
5
3
29
23
19
7
11
13
17
5
3
1
2
3
4
5
6
7
8
9

$\mathit{sift\ down}(H, i)$:

    if i tiene hijos:

        $i' \leftarrow$ el hijo de i de mayor prioridad

        if $H[i'] > H[i]$*:*

            $H[i'] \rightleftarrows H[i]$

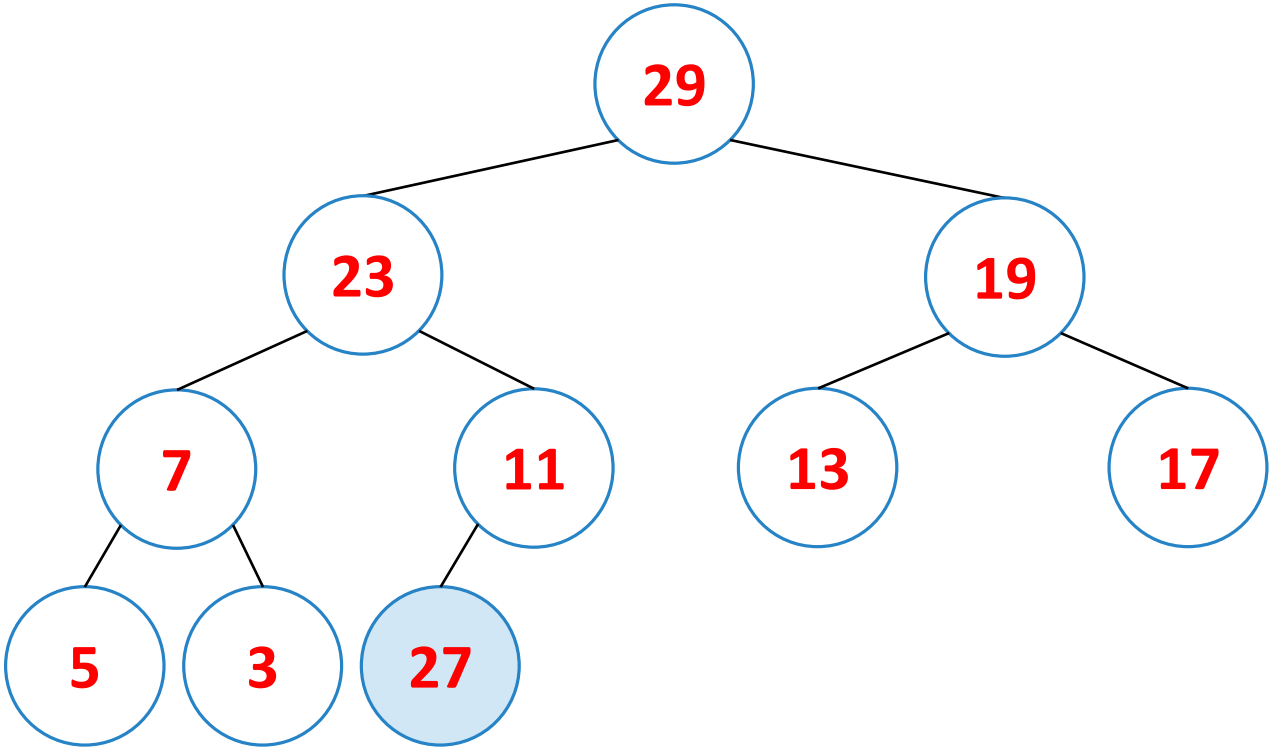
            $\mathit{sift\ down}(H, i')$

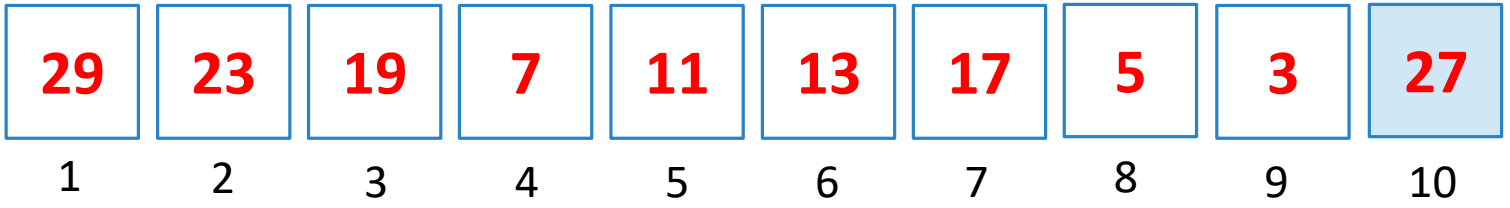
$\boldsymbol{insert}(\boldsymbol{H}, \boldsymbol{e})$:

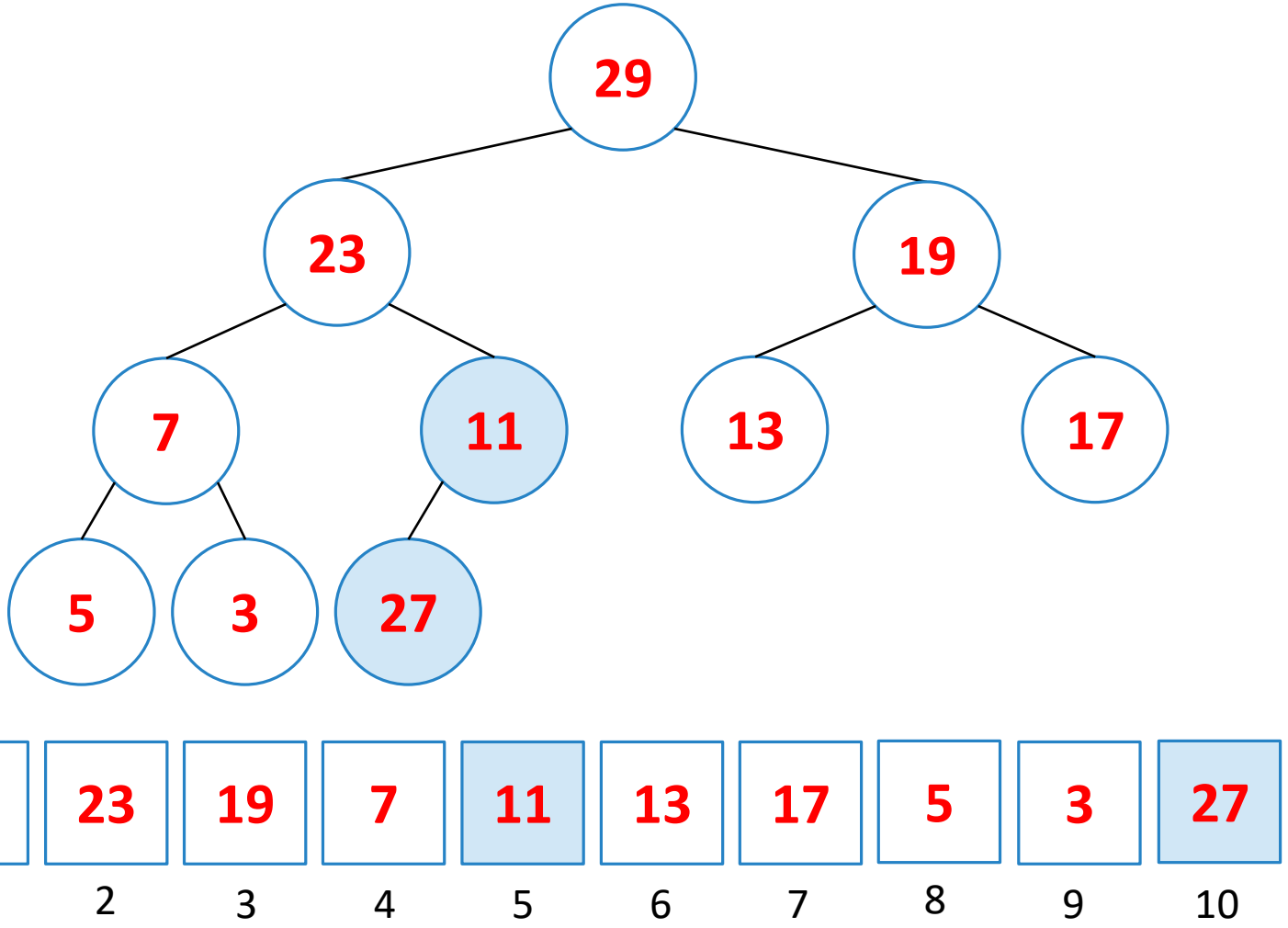
$\boldsymbol{i} \leftarrow$ la primera celda en blanco de $\boldsymbol{H}$

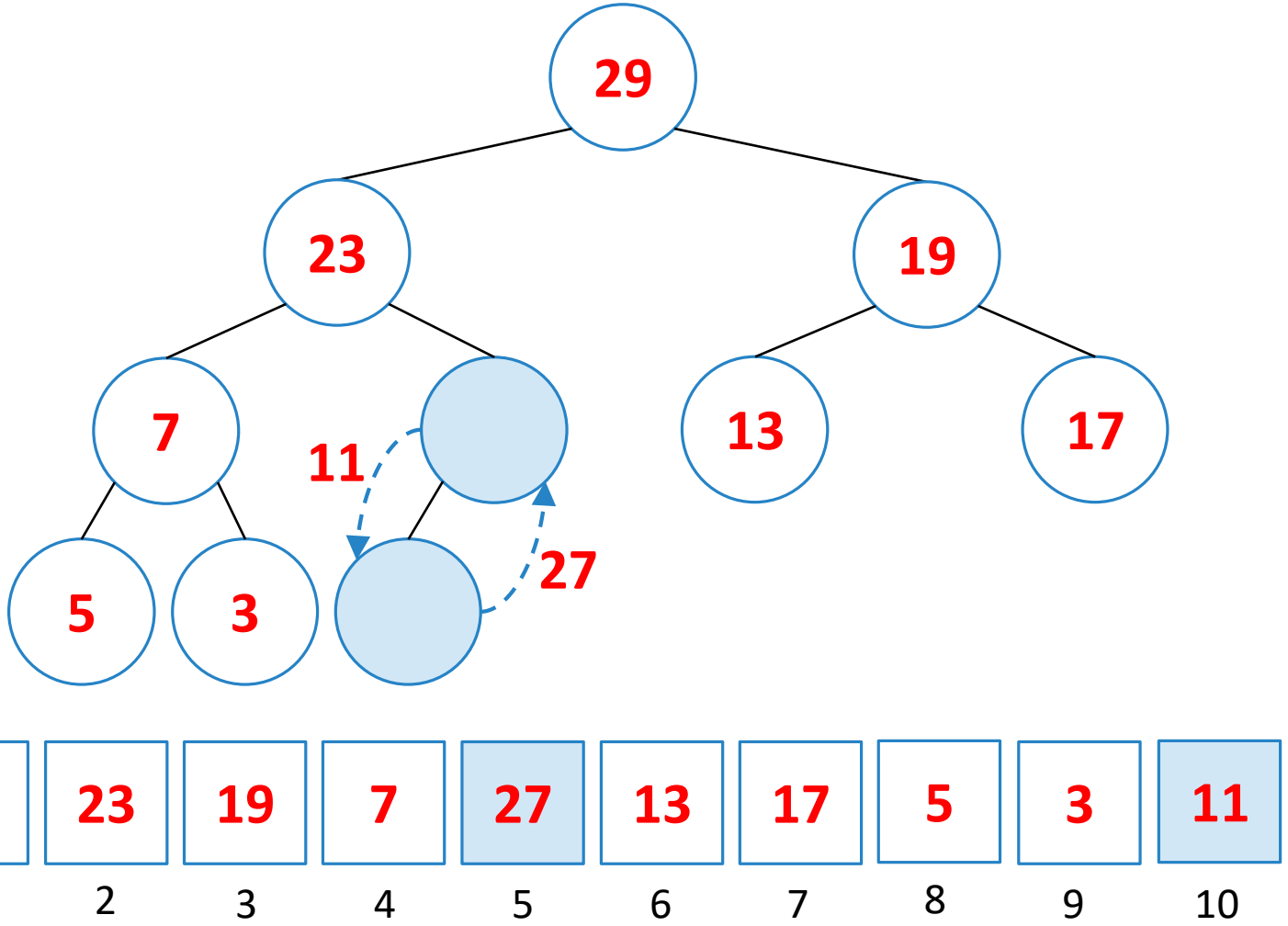
$\boldsymbol{H}[\boldsymbol{i}] \leftarrow \boldsymbol{e}$

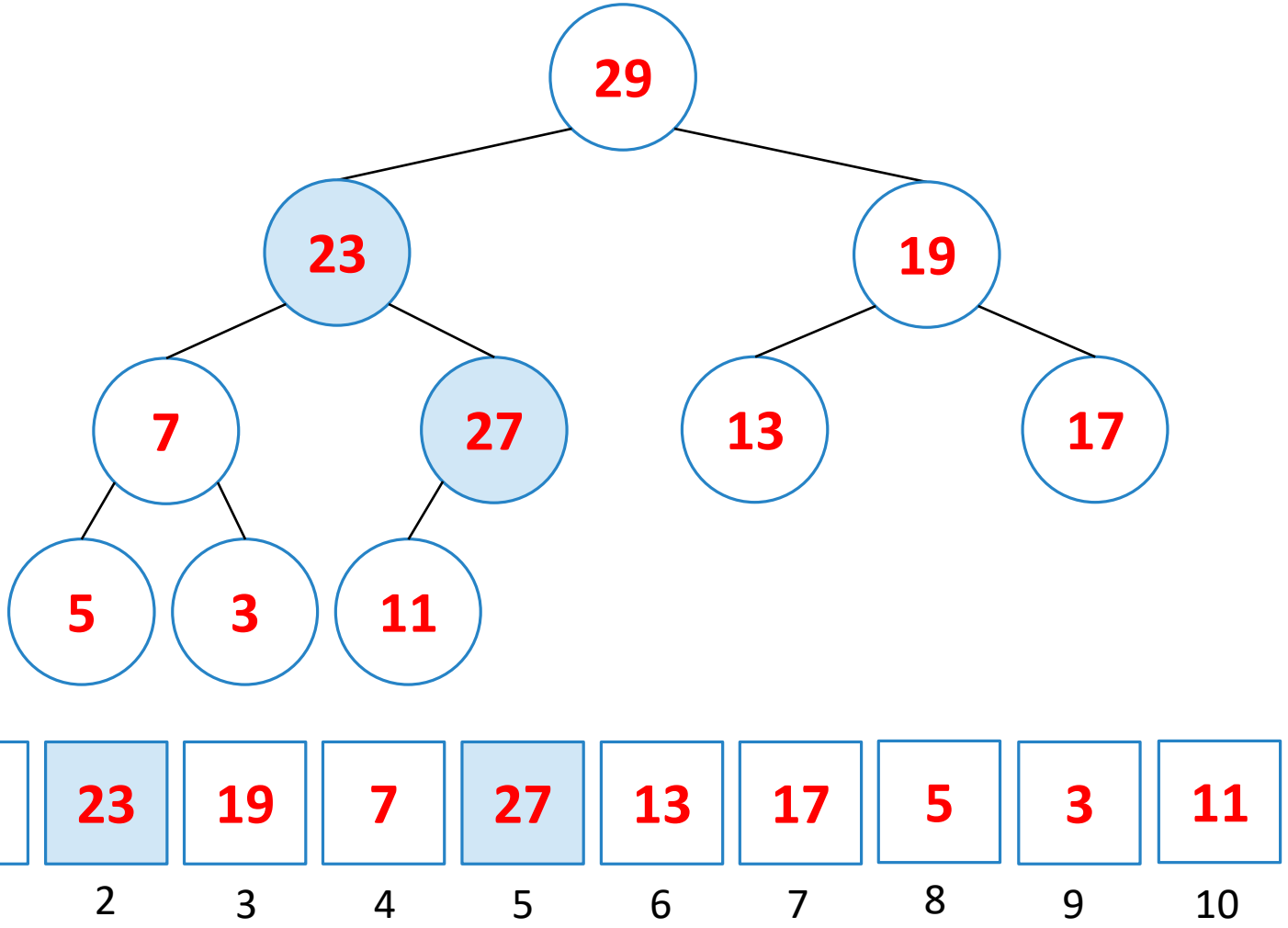
$\boldsymbol{sift\ up}(\boldsymbol{H}, \boldsymbol{i})$

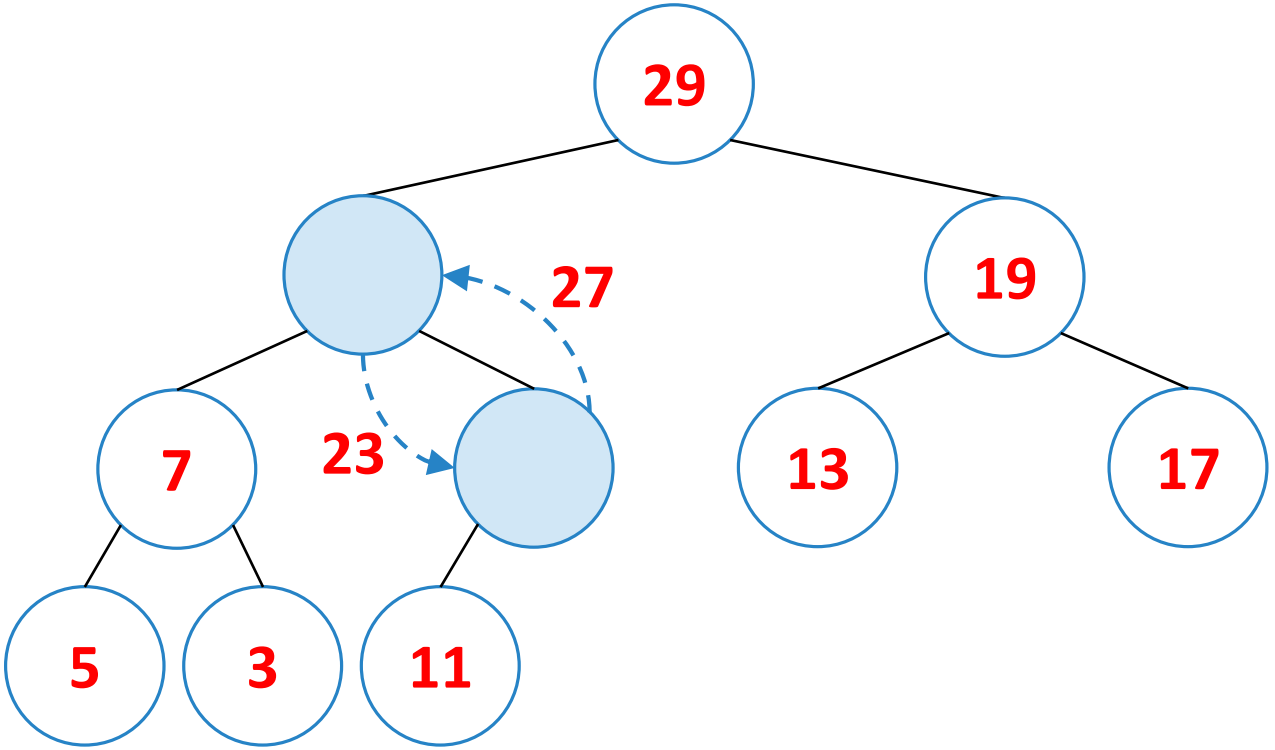

29
23
19
7
11
13
17
5
3
27


29
23
19
7
11
13
17
5
3
27
1
2
3
4
5
6
7
8
9
10

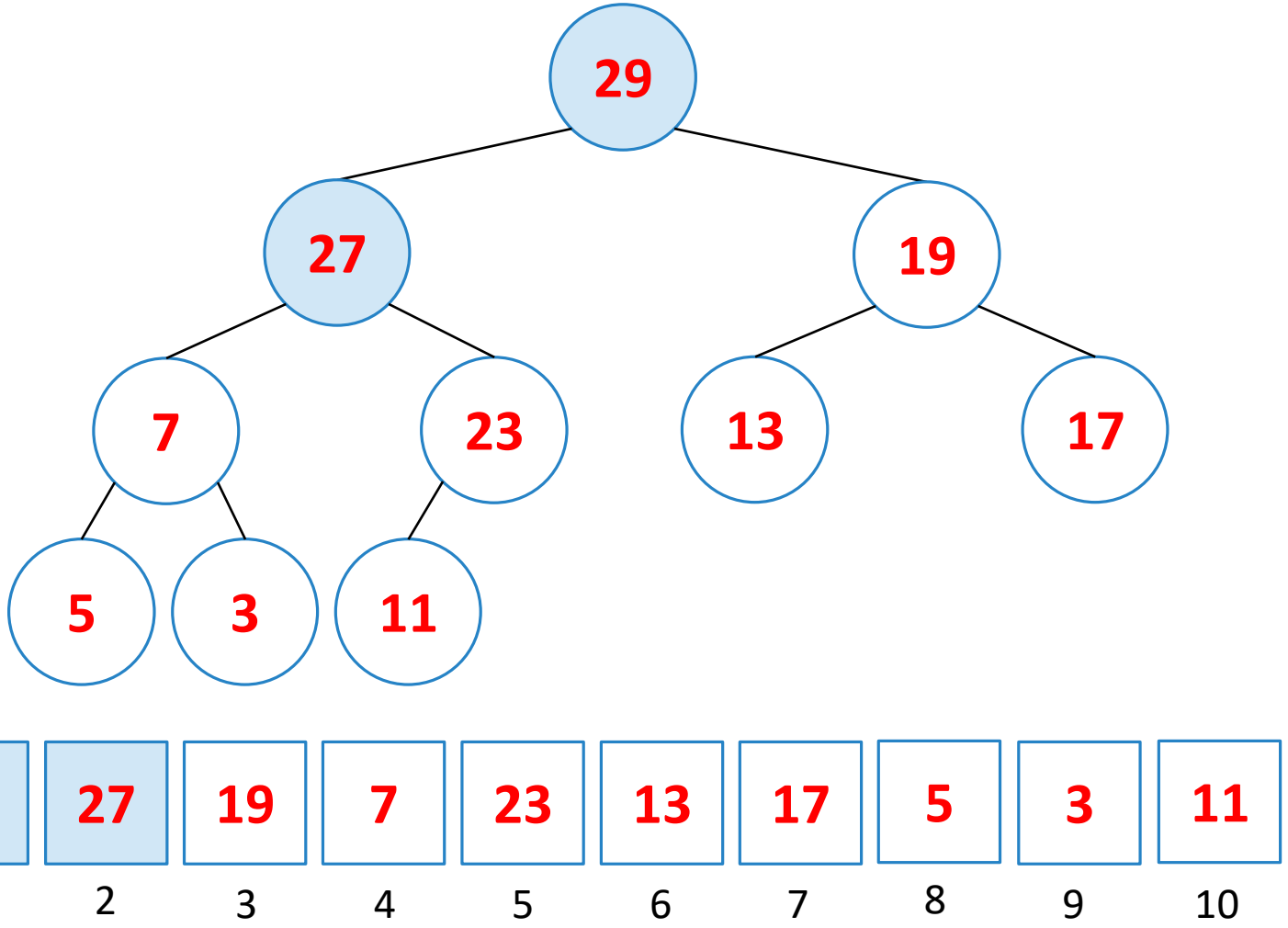

29
23
19
7
11
13
17
5
3
27
29
23
19
7
11
13
17
5
3
27
1
2
3
4
5
6
7
8
9
10

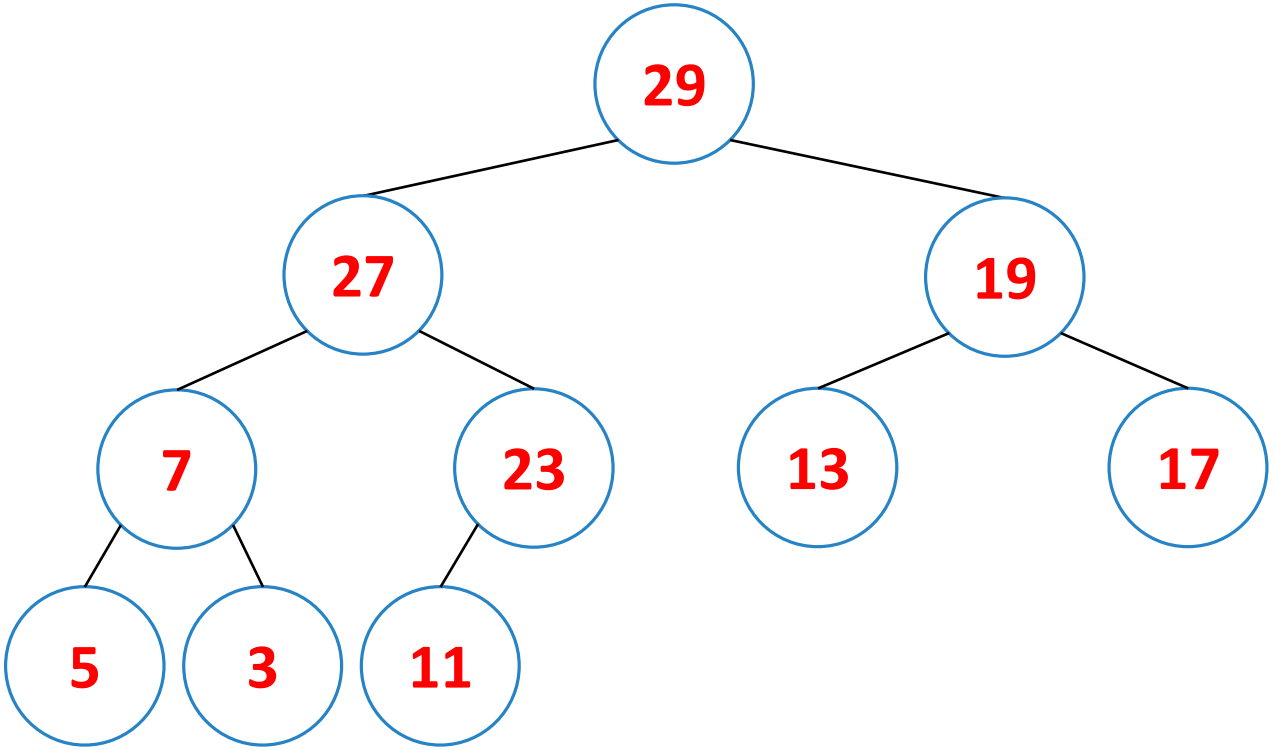

29
23
19
7
11
13
17
5
3
27
29
23
19
7
27
13
17
5
3
11
1
2
3
4
5
6
7
8
9
10


29
23
19
7
27
13
17
5
3
11
29
23
19
7
27
13
17
5
3
11
1
2
3
4
5
6
7
8
9
10



29	27	19	7	23	13	17	5	3	11
1	2	3	4	5	6	7	8	9	10


29
27
19
7
23
13
17
5
3
11
29
27
19
7
23
13
17
5
3
11
1
2
3
4
5
6
7
8
9
10



29	27	19	7	23	13	17	5	3	11
1	2	3	4	5	6	7	8	9	10

$\mathit{sift\ up}(H, i)$:

    if i tiene padre:

        $i' \leftarrow$ el padre de i

        if $H[i'] < H[i]$:

            $H[i'] \rightleftarrows H[i]$

            $\mathit{sift\ up}(H, i')$

Complejidad de las operaciones

¿Cómo se relaciona el número de datos con la altura del heap?

¿Qué complejidad tiene insertar?

¿Qué complejidad tiene extraer?

¿De qué nos sirve todo esto?

Anteriormente vimos selection sort

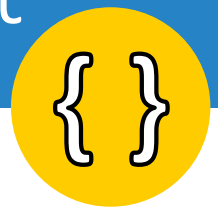
¿Será posible usar un heap para mejorar su rendimiento?

HeapSort

Para la secuencia inicial de datos, A.

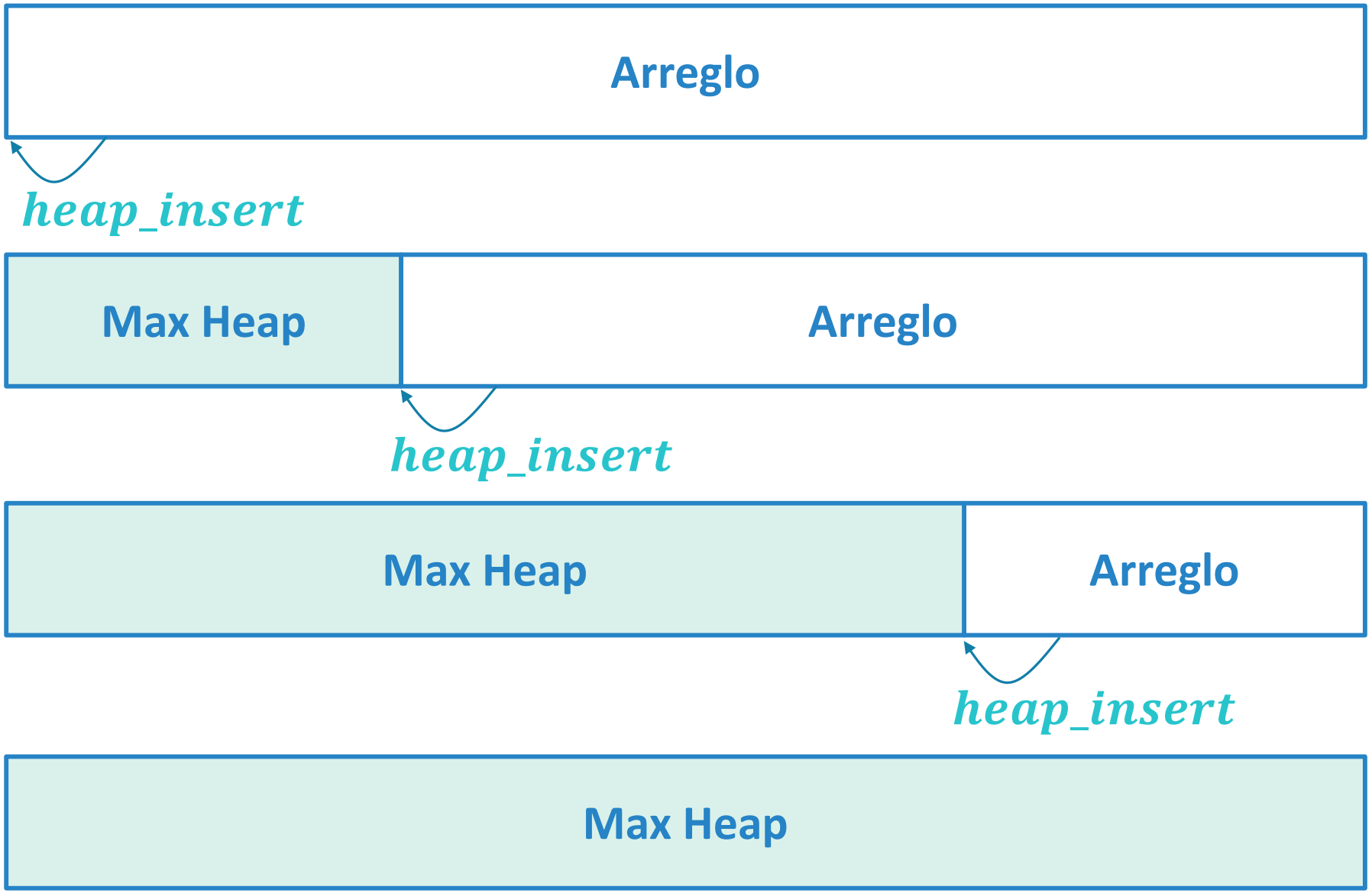
1. Convertir A en un **max heap** con los datos
2. Definir una secuencia ordenada, B, inicialmente vacía
3. Extraer el mayor dato x de A e insertarlo en la última posición vacía de B
4. Si quedan elementos en A, volver a 2

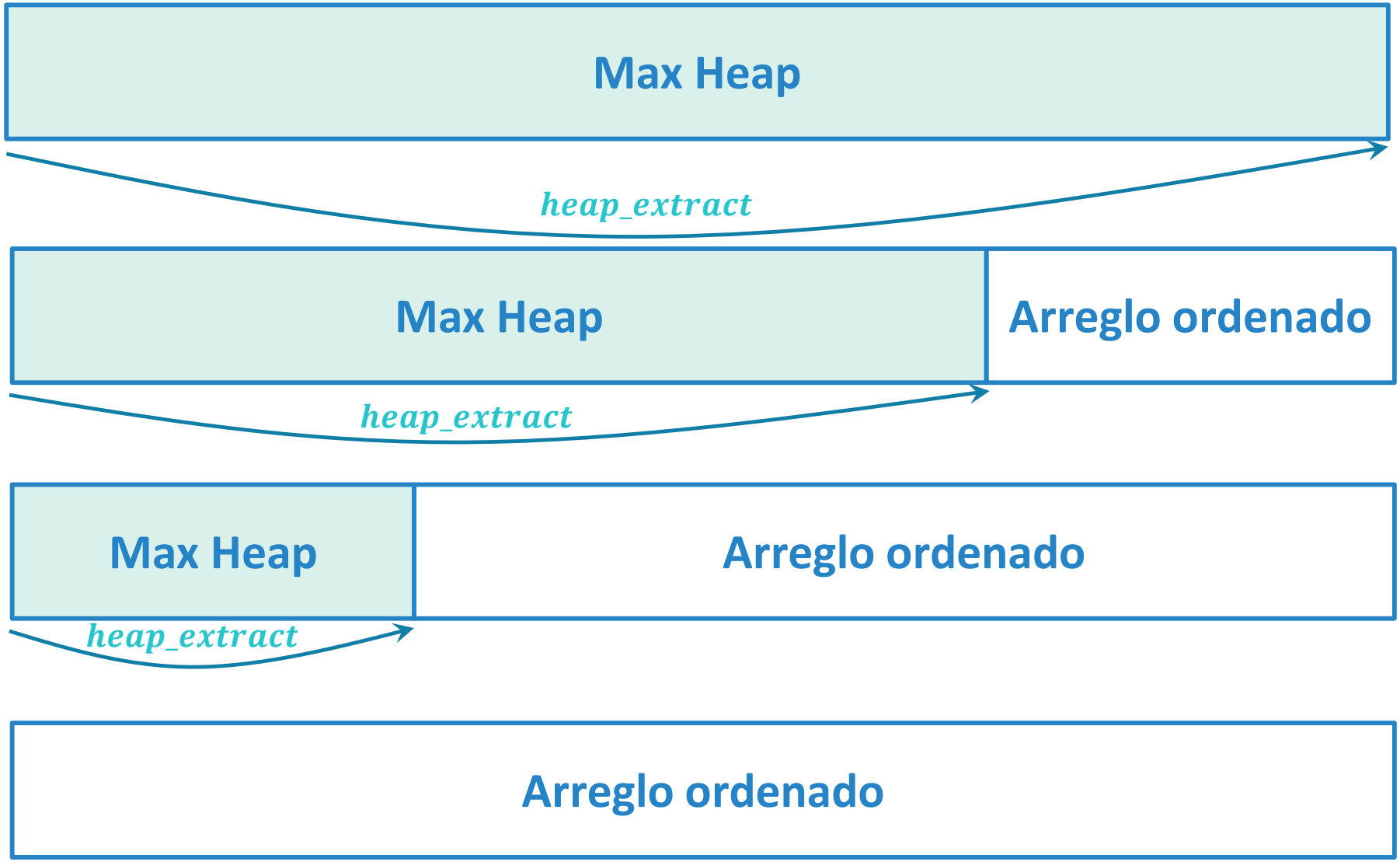
Necesidad de memoria para heapsort



En la práctica, se usa un mismo arreglo para el arreglo inicial A y el arreglo resultado B

Eso significa que **heapsort** no requiere memoria adicional


Arreglo
heap_insert
Max Heap
Arreglo
heap_insert
Max Heap
Arreglo
heap_insert
Max Heap


Max Heap
heap_extract
Max Heap
Arreglo ordenado
heap_extract
Max Heap
Arreglo ordenado
heap_extract
Arreglo ordenado

Algoritmos de Ordenación

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
HeapSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(1)$