Hemos visto **ordenación** como problema computacional

… y varios **algoritmos de ordenación**, como soluciones al problema

Todos ordenan correctamente

… pero difieren en cuanto a sus rendimientos

Cotas inferiores para algoritmos de ordenación:

- 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	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
?	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(1)$

? es un último algoritmo que veremos hoy

Pero primero…

Vamos a definir una **estructura de datos** —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

Ustedes ya conocen un par de estructuras de datos básicas

Colas (o colas FIFO—*first in first out*):

- inserción: insertamos al final, después del último dato que ya está en la cola
- extracción: sacamos el dato que está al principio, el que lleva más tiempo en la cola

Stacks (o colas LIFO—*last in first out*):

- inserción: insertamos al principio, antes (o arriba) del primer dato que ya está en el stack
- extracción: sacamos el dato que está al principio (o más arriba), el que lleva menos tiempo en el stack

Ambas se pueden implementar eficientemente tanto con arreglos, como con listas ligadas

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:

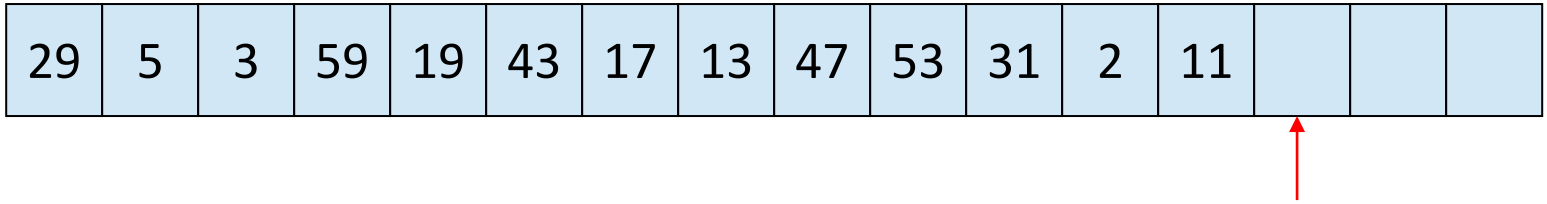
- cambiar la prioridad de un dato que ya está en la cola)

La idea es que la posición del dato en la cola no depende del orden de llegada (como en las colas FIFO y LIFO)

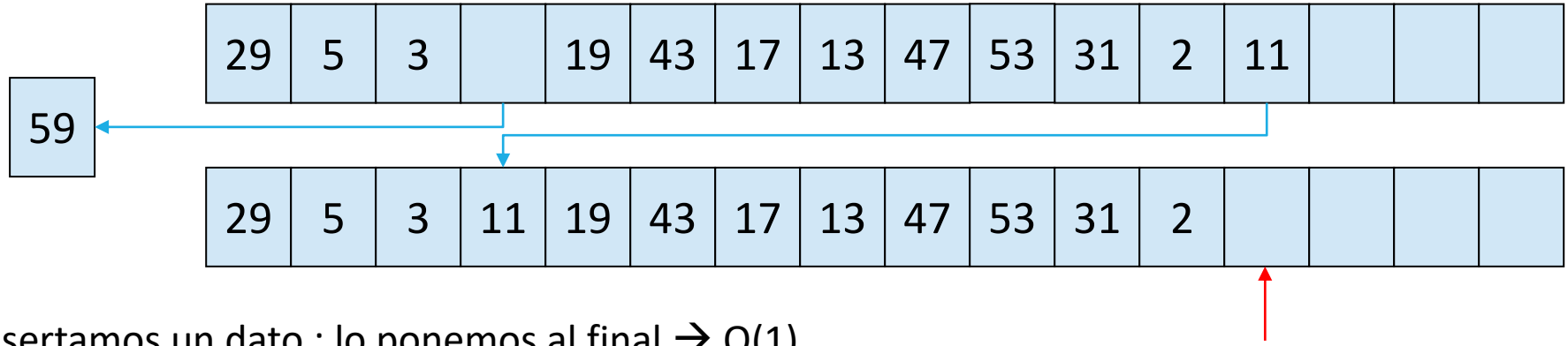
… sino de la prioridad que tiene (o se le asigna de alguna manera)

Implementación mediante un arreglo simple (sin “estructura”)

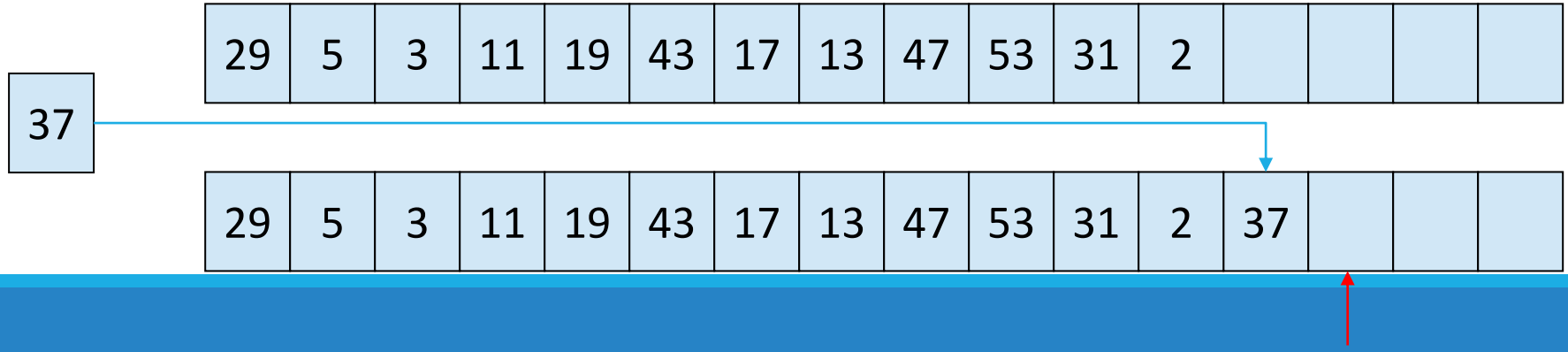
los números son la prioridad



sacamos el dato más prioritario: hay que buscarlo secuencialmente → O(*n*)



insertamos un dato : lo ponemos al final → O(1)



Propiedad de orden de la cola

Claramente, hay que (pensar en) 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?

Los dos “extremos” son O(*n*)

Arreglo simple (sin estructura):

- inserción: O(1)
- extracción: O(*n*)

Arreglo totalmente ordenado:

- inserción: O(*n*)
- extracción: O(1)

Pero … ¿es necesario un orden total?

Sólo necesitamos saber cuál es el dato más prioritario

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

¡ Necesitamos algún tipo de estructura interna en el arreglo !

Cómo aprovechar la propiedad de orden parcial

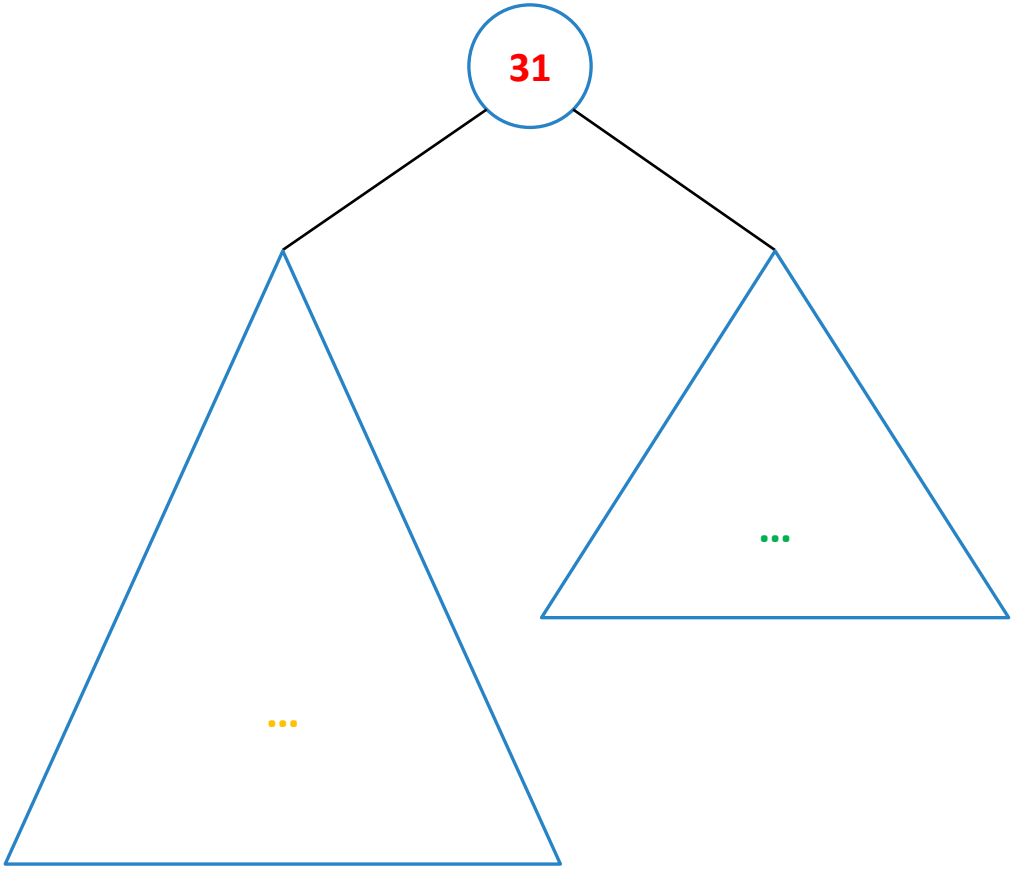
¿Qué información contenida en la estructura debe estar fácilmente disponible en todo momento?

¿Será posible hacer una estructura recursiva?

¿Por qué querríamos tener estructuras recursivas?

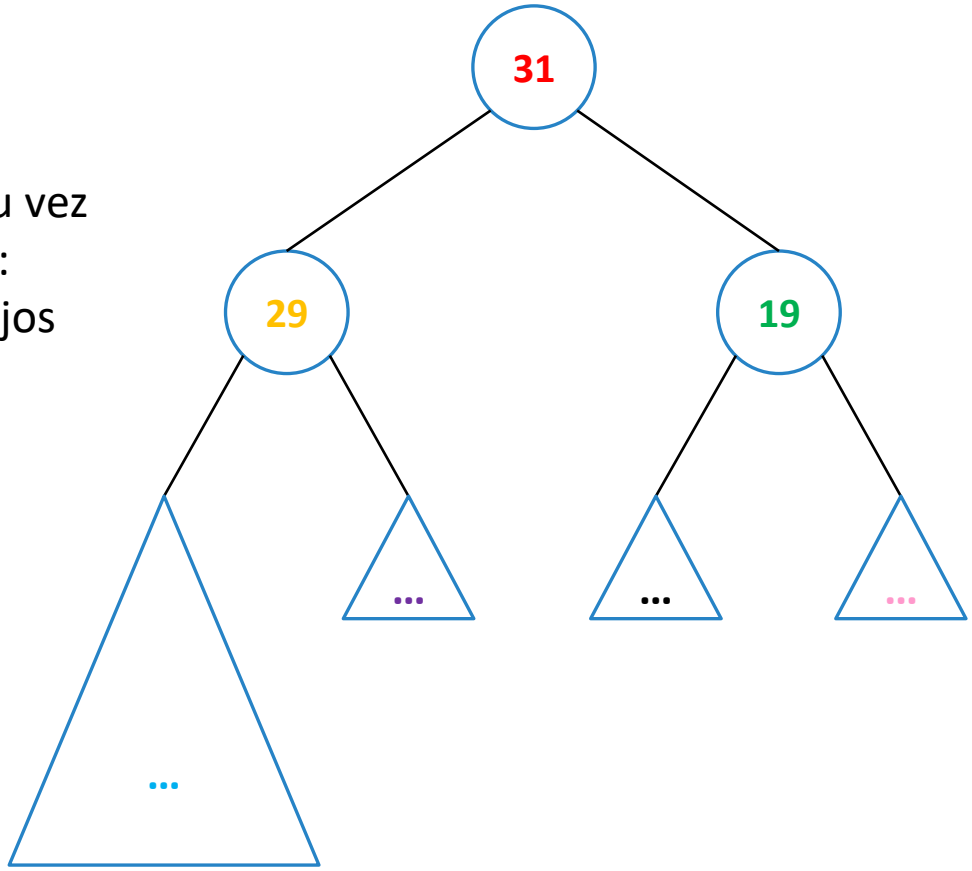
- los algoritmos para recorrerlas o buscar información en ellas son también recursivos y, por lo tanto, más simples
- la implementación de la estructura se simplifica

Un **heap binario**: la raíz y sus dos hijos



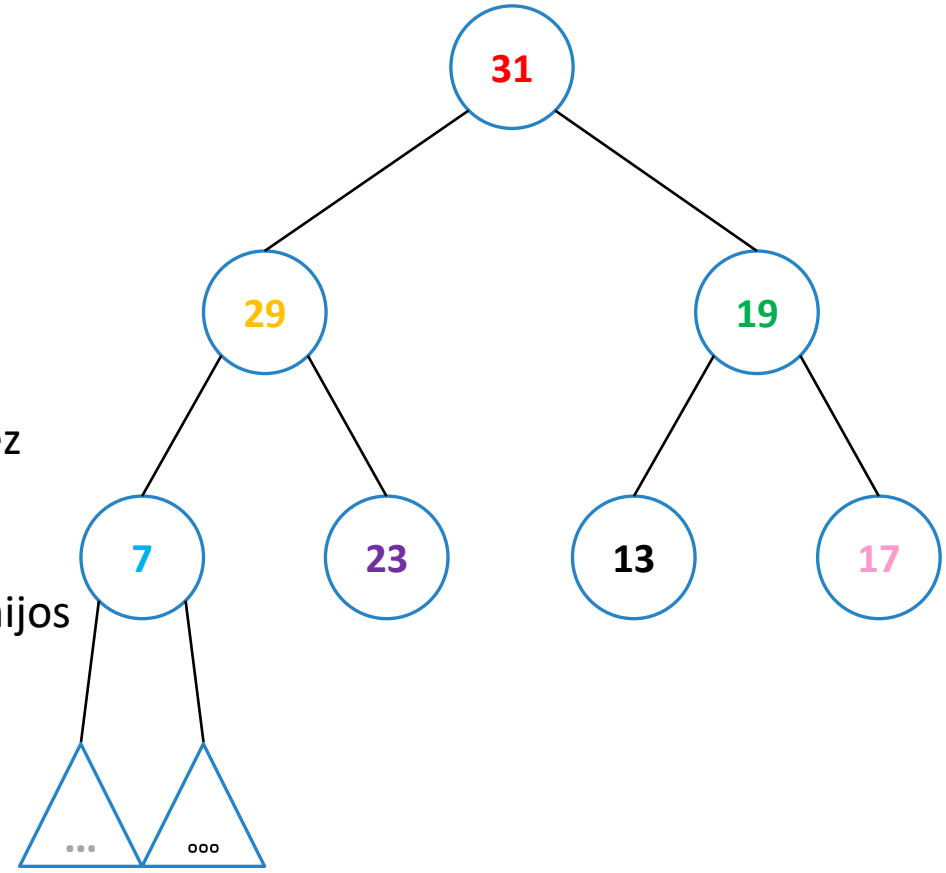
Un heap binario: recursivamente

Cada hijo es a su vez
un heap binario:
con raíz y dos hijos

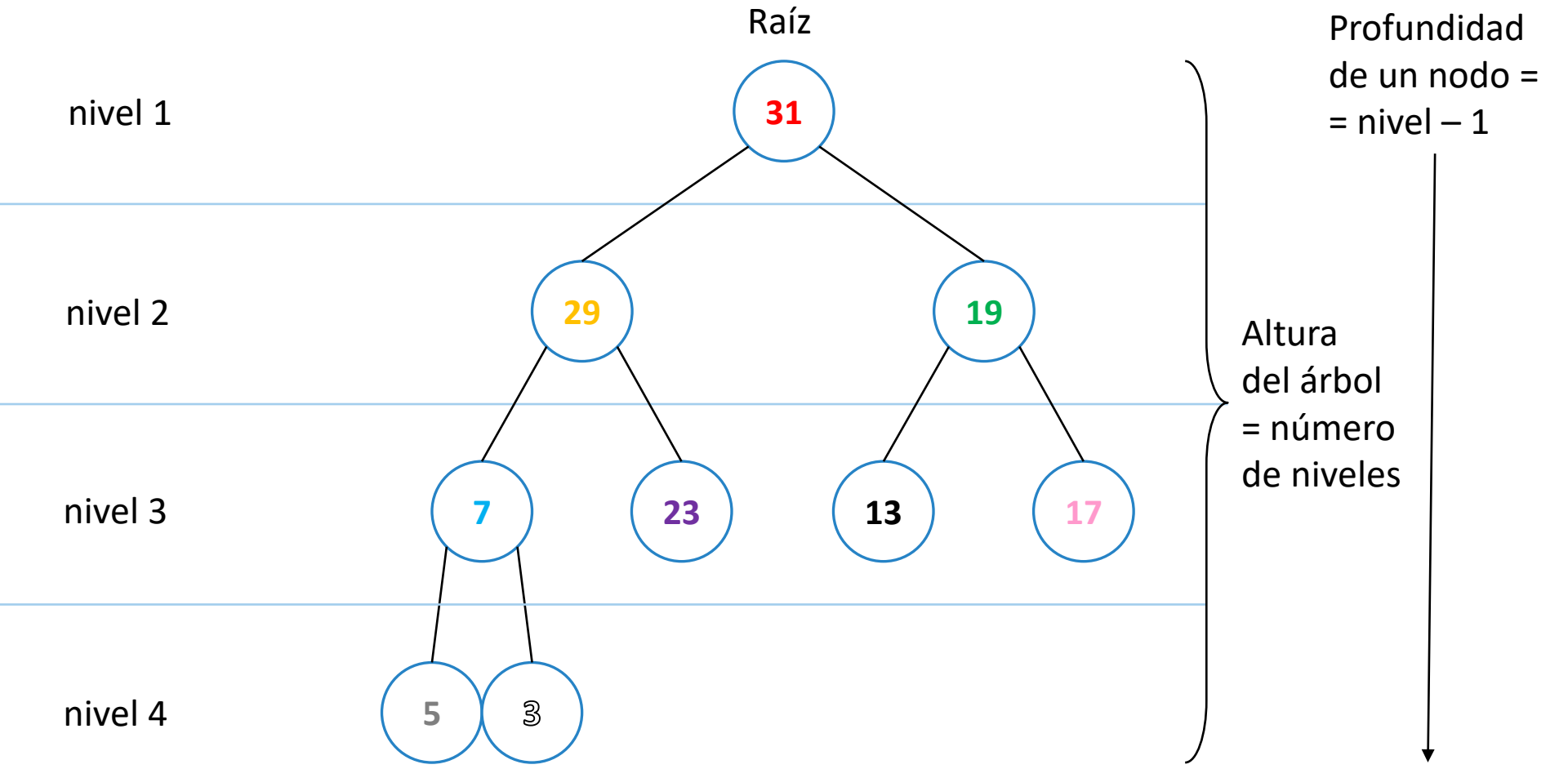


Un heap binario: recursivo a todo nivel

Cada hijo es a su vez un heap binario: con raíz y dos hijos (cualquiera de los hijos puede ser nulo)



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:

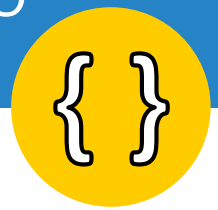
- cada grupo está organizado a su vez —recursivamente— como un heap binario
- estos dos heaps binarios cuelgan de la raíz como sus hijos

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?

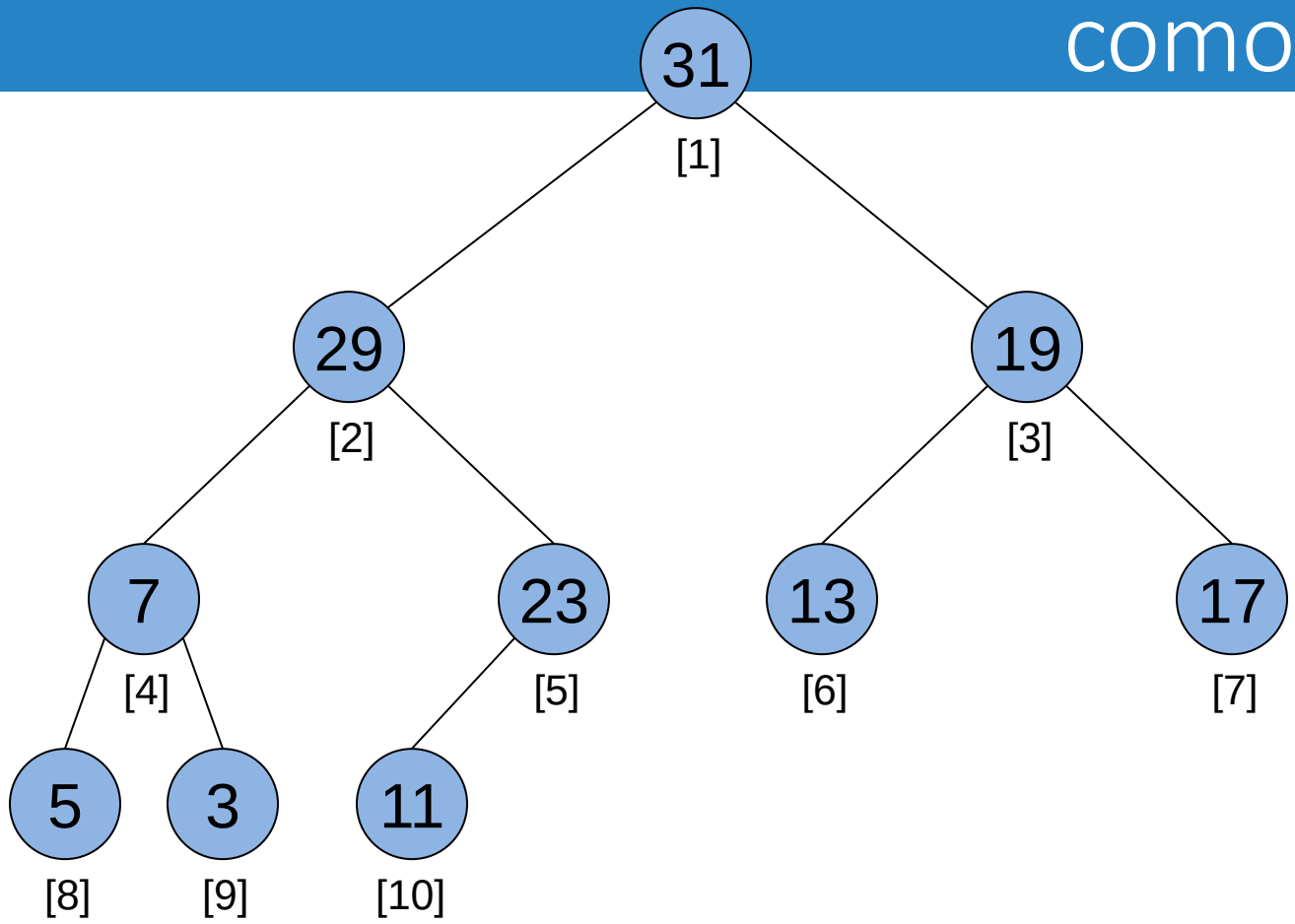
Una implementación simple de un heap binario



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

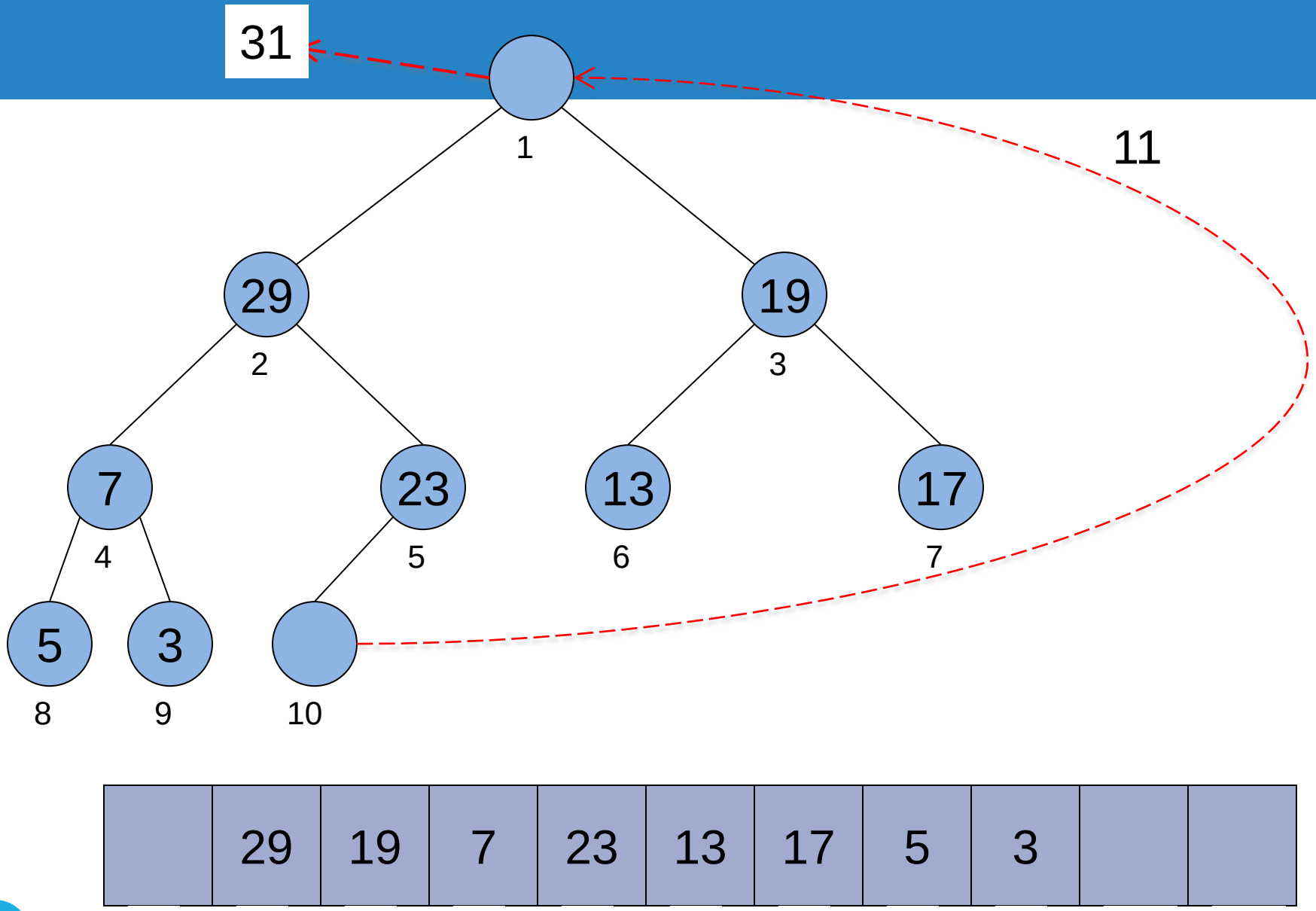


31	29	19	7	23	13	17	5	3	11	
1	2	3	4	5	6	7	8	9	10	11

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

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

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

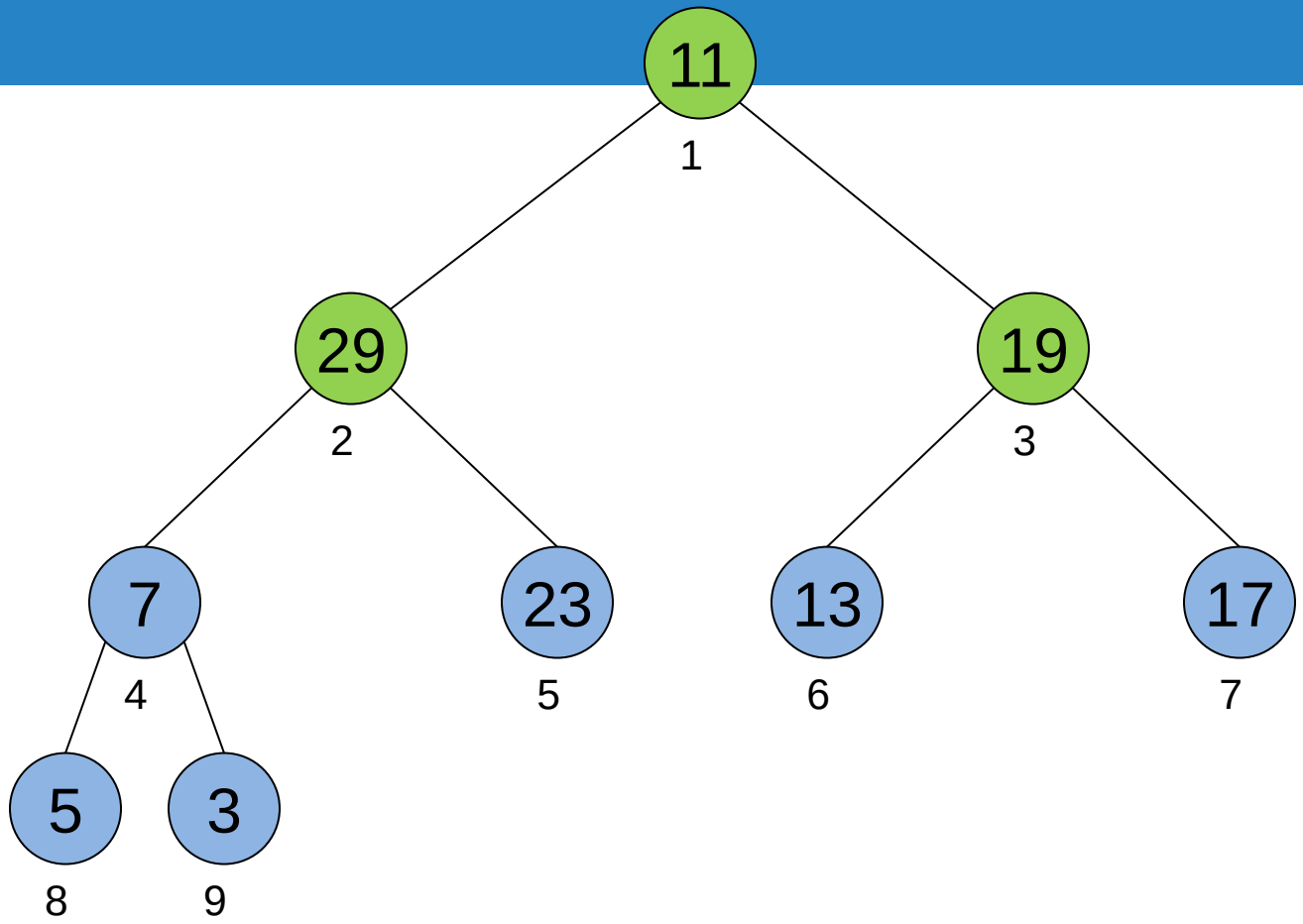
$\boldsymbol{best} \leftarrow \boldsymbol{H}[\mathbf{0}]$

$\boldsymbol{H}[\mathbf{0}] \leftarrow \boldsymbol{H}[\boldsymbol{i}]$

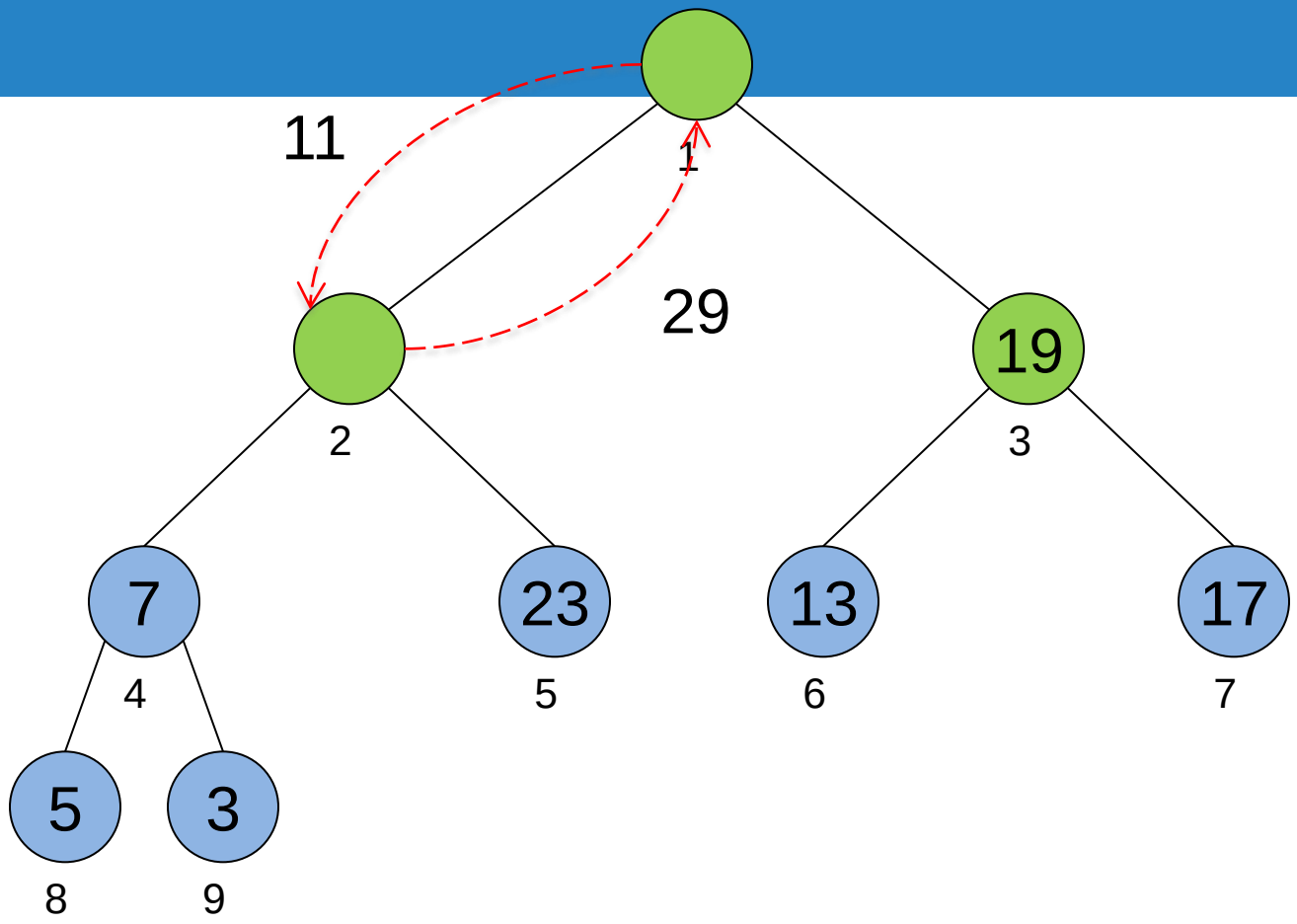
$\boldsymbol{H}[\boldsymbol{i}] \leftarrow \emptyset$

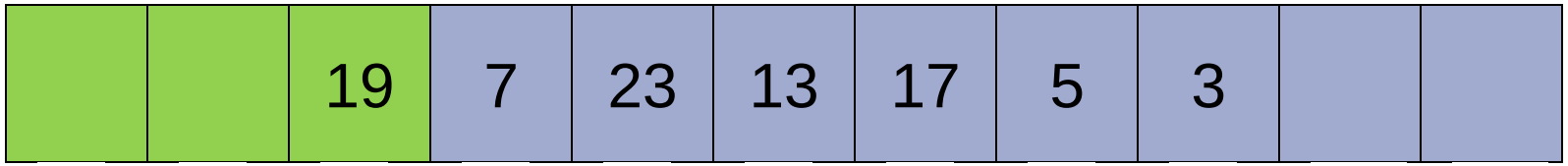
$\boldsymbol{sift\ down}(\boldsymbol{H}, \mathbf{0})$

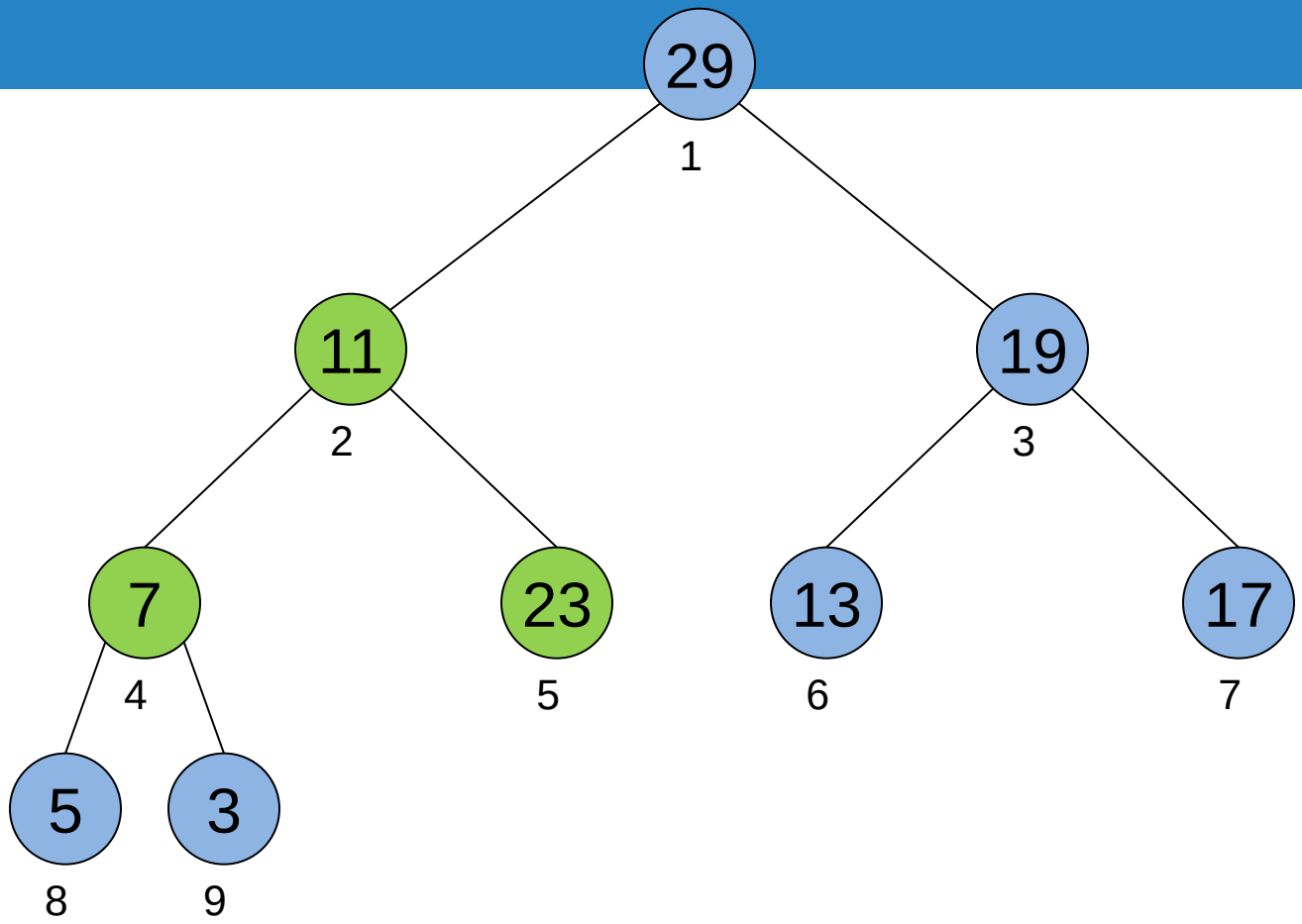
$\boldsymbol{return\ best}$



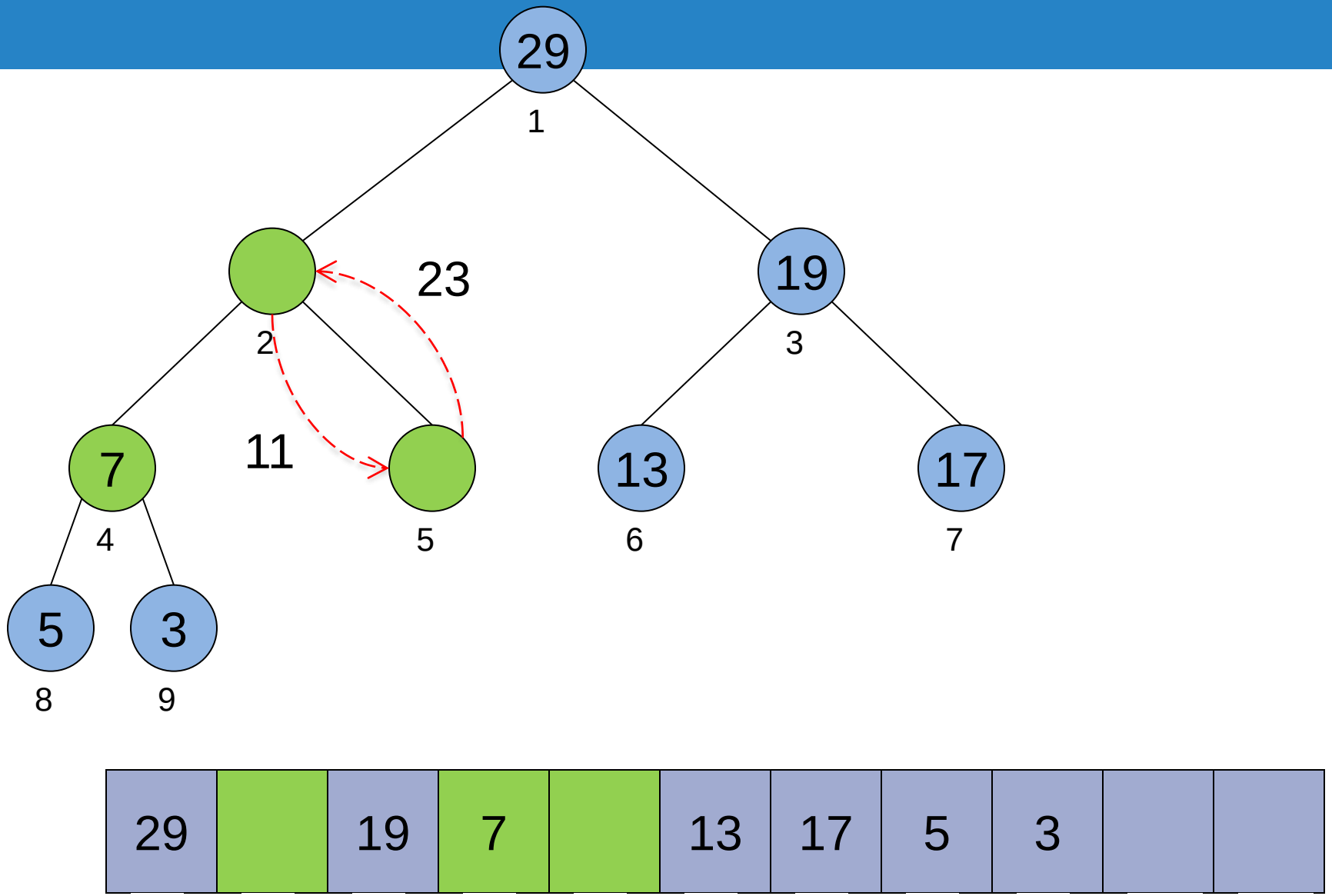
11	29	19	7	23	13	17	5	3		
1	2	3	4	5	6	7	8	9	10	11


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

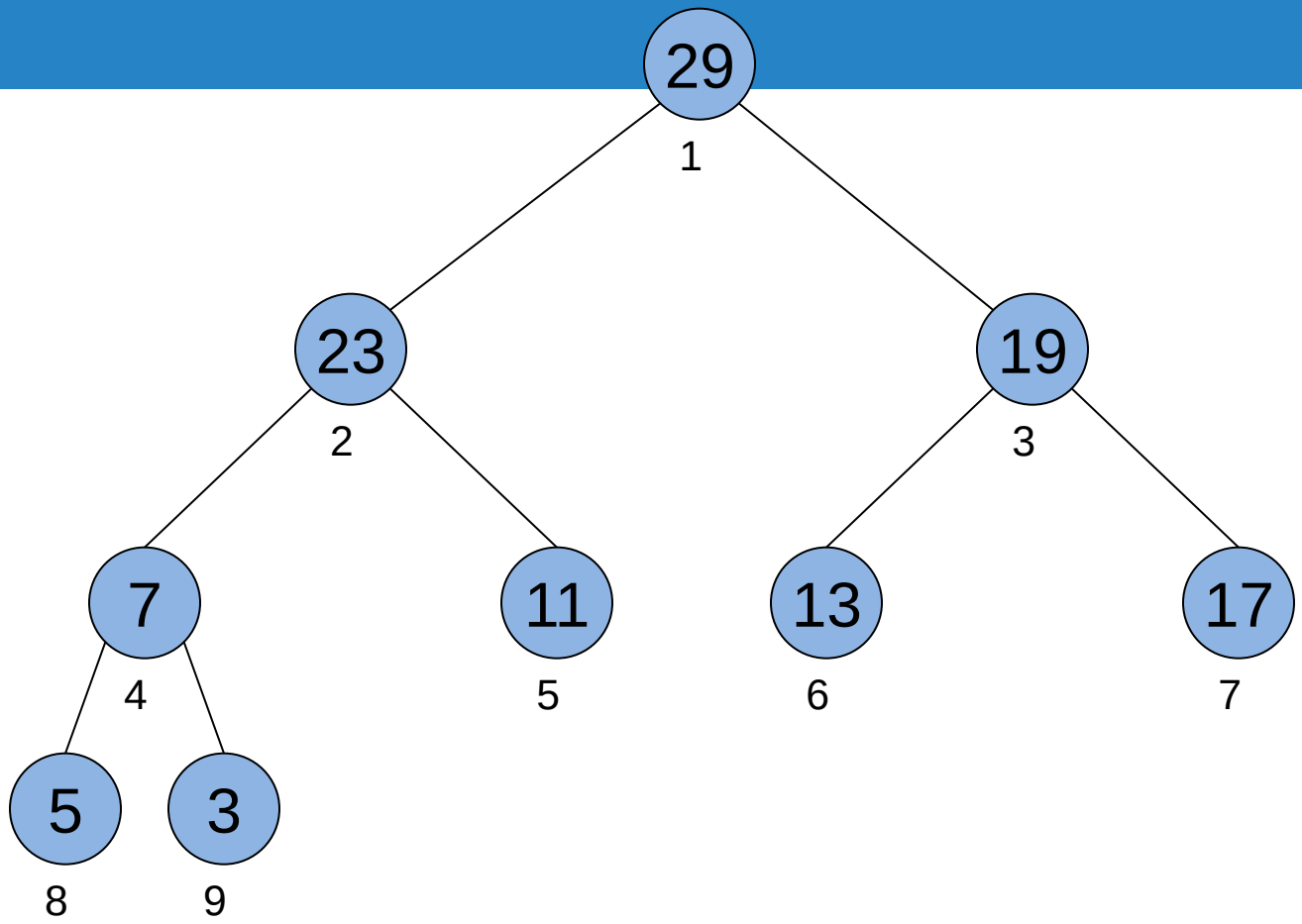

19
7
23
13
17
5
3
1
2
3
4
5
6
7
8
9
10
11



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


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

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



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

$sift\ down(H, i)$:

    if i tiene hijos:

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

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

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

            $sift\ down(H, i')$

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

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

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

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

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

    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?

(¿Qué complejidad tiene el cambio de prioridad?)

¿De qué nos sirve todo esto?

Anteriormente vimos selection sort

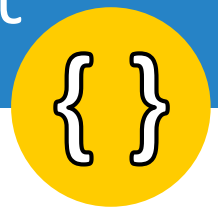
¿Será posible usar un heap para mejorar su rendimiento?

HeapSort

Para la secuencia inicial de datos, A.

1. Convertir A en un heap con los datos
2. Definir una secuencia ordenada, B, inicialmente vacía
3. Extraer el menor dato x de A e insertarlo al final 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

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	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(1)$