Estructuras de datos basadas en arreglos

Los arreglos son una representación directa, en un lenguaje de programación, de la memoria del computador

Vimos los arreglos simplemente como una forma conveniente de almacenar datos que queríamos ordenar

… y también como una forma de implementar diccionarios (complementados o no por listas ligadas)

Los arreglos también nos ayudan a modelar y resolver otros problemas

Colas de prioridades:

almacenar datos según cierta prioridad, consultar cuál es el más prioritario de los datos, y procesarlos en orden de prioridad

Los arreglos también nos ayudan a modelar y resolver otros problemas

Conjuntos disjuntos:

identificar a qué conjunto pertenece un elemento, y unir dos conjuntos disjuntos (dejando sólo la unión y eliminando los conjuntos originales)

Ya conocemos dos estructuras de datos simples basadas en el orden de llegada

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

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

La *prioridad* con la que procesamos un dato está dada por el orden de llegada del dato a 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

La *prioridad* con la que procesamos un dato está dada por el inverso del orden de llegada del dato al stack

1) Colas de prioridades (*highest priority first out*)

Estructura de datos con las siguientes operaciones:

- **insertar** un dato con una **prioridad dada**
- **extraer** el dato con **mayor prioridad**

(… e idealmente:

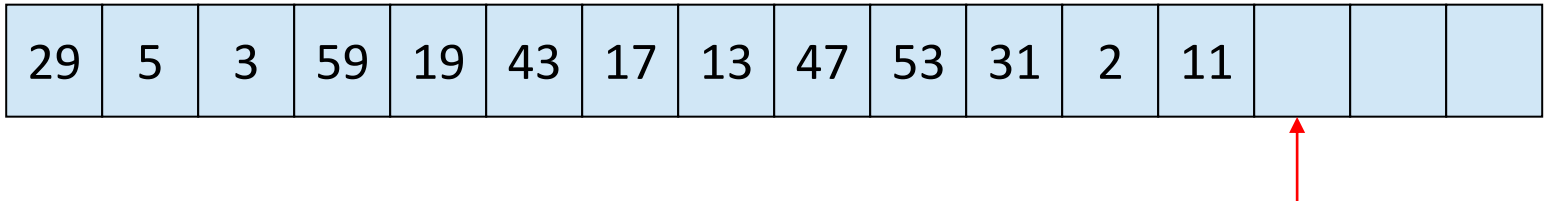
- cambiar la prioridad de un dato que ya está en la cola)

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

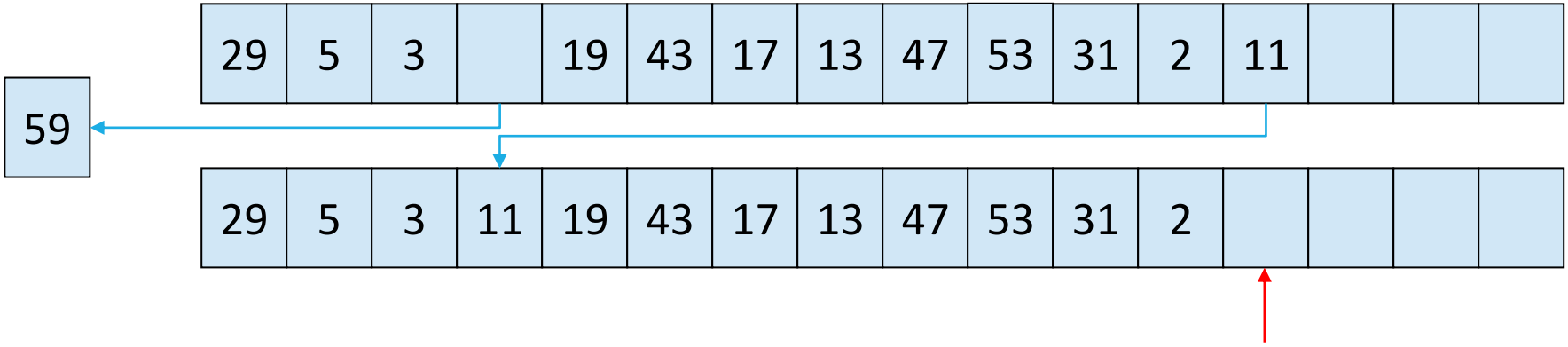
… sino de la prioridad que tiene (o se le asigna de alguna manera)

Implementación mediante un arreglo simple (sin "estructura"): *extracción*

los números son la prioridad

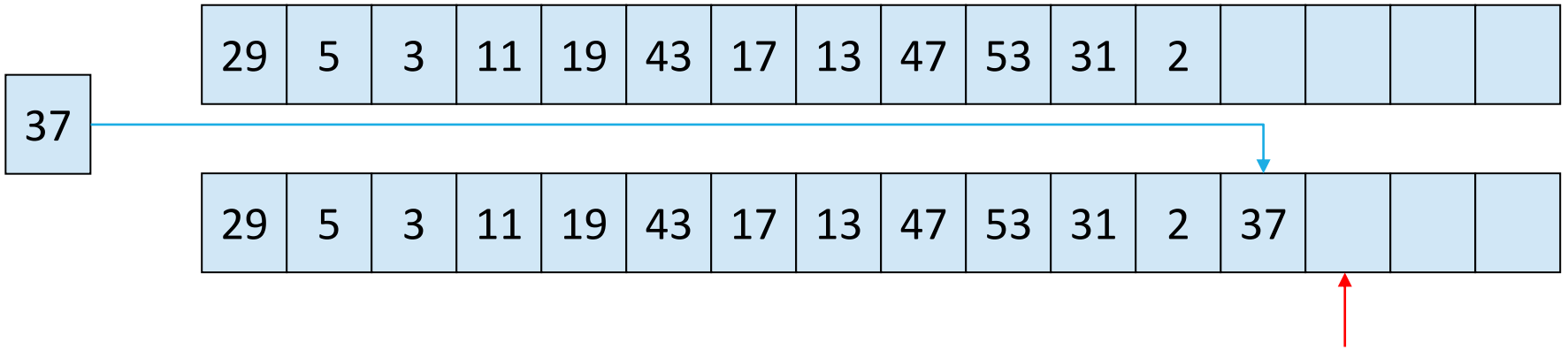


extraemos el dato más prioritario: hay que buscarlo secuencialmente → O(*n*)



Implementación mediante un arreglo simple (sin “estructura”): *inserción*

insertamos un dato con una cierta prioridad: lo ponemos al final → O(1)



Los dos “extremos” son O(*n*)

Arreglo simple (sin estructura):

- inserción: O(1)
- extracción: O(*n*)
- como en el ej. de la diap. anterior

Arreglo totalmente ordenado:

- inserción: O(*n*)
- extracción: O(1)
- como lo hemos “deducido” varias veces en clases

Propiedad de orden de la cola

Claramente, hay que mantener cierto orden de los datos

Pero … ¿es necesario un orden total?

- al hacer una extracción, 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?

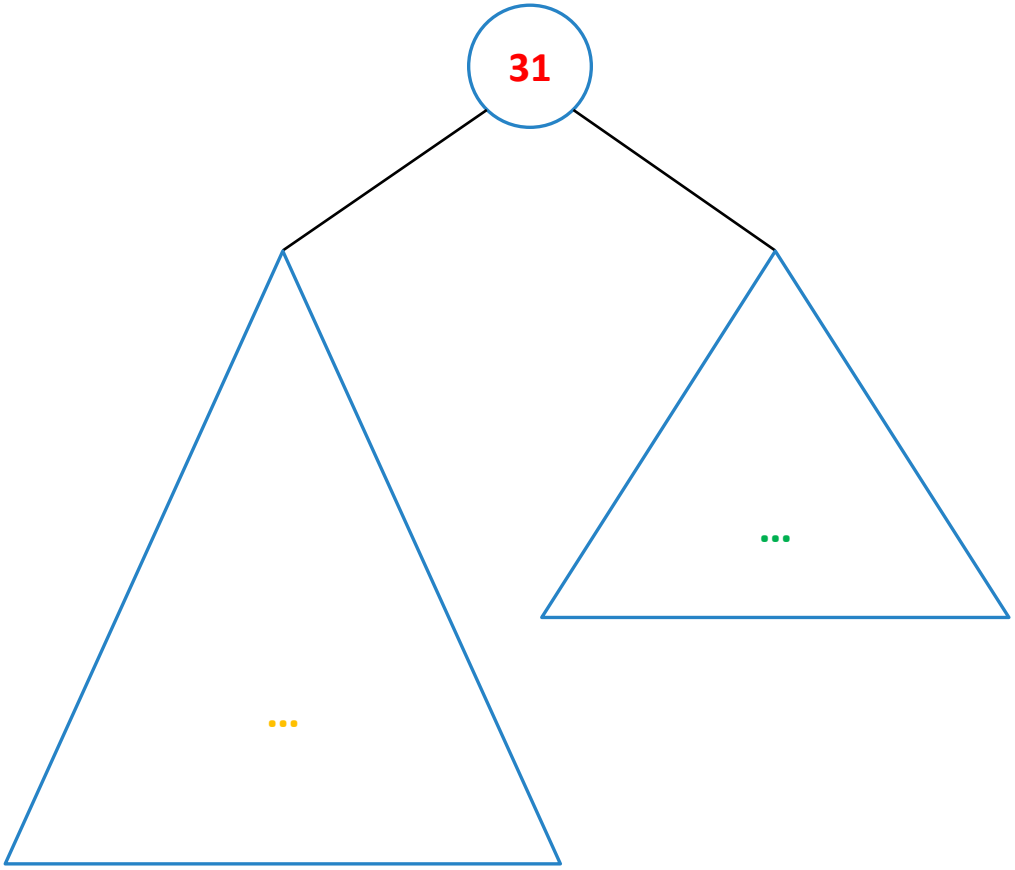
¿Qué información contenida en la estructura debe estar fácilmente disponible en todo momento?

¿Será posible hacer una estructura recursiva?

¿Por qué querríamos tener una estructura recursiva?

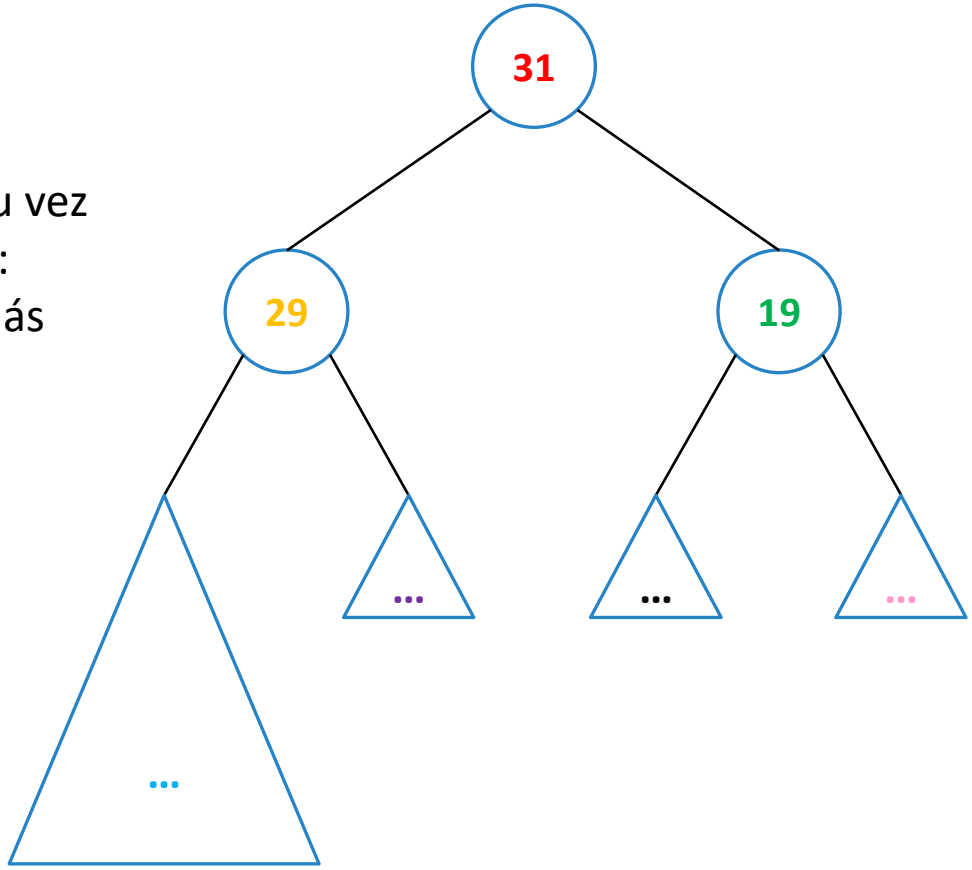
- los algoritmos para recorrerla o buscar información en ella son también recursivos y, por lo tanto, más simples
- la implementación de la estructura se simplifica

Un *heap* **binario** es un árbol binario, en que **el valor de la raíz** …



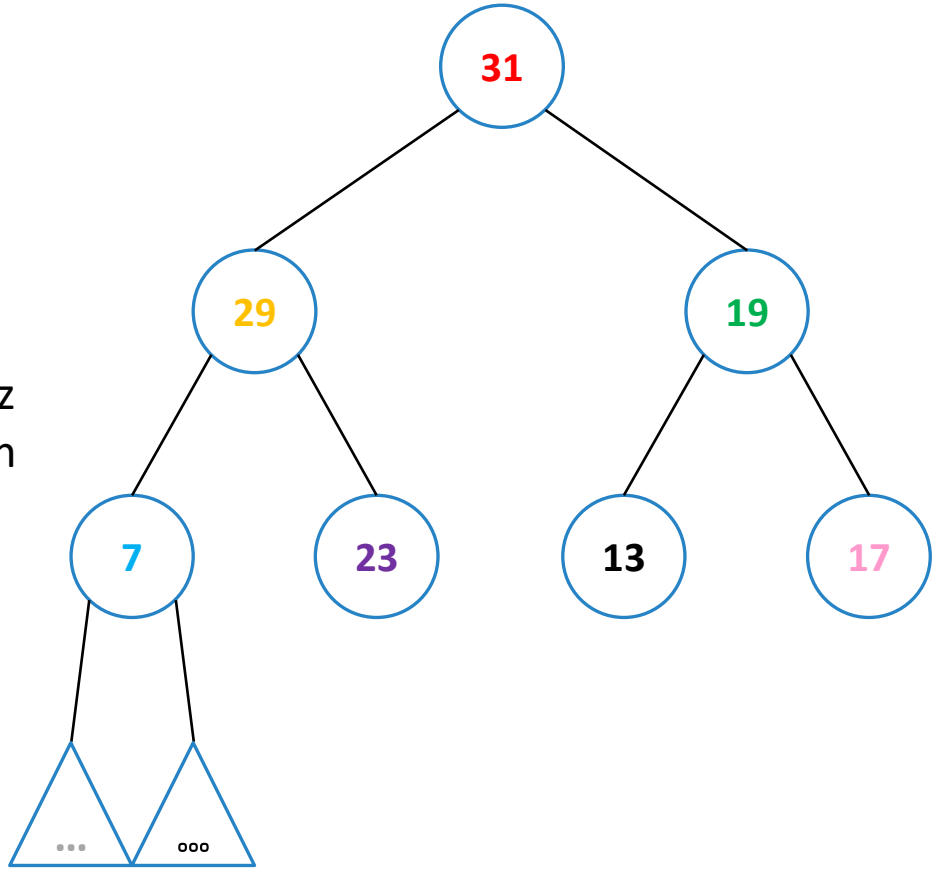
… es mayor (o igual) que el valor de cualquiera de sus hijos …

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



… y así recursivamente hasta las hojas

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



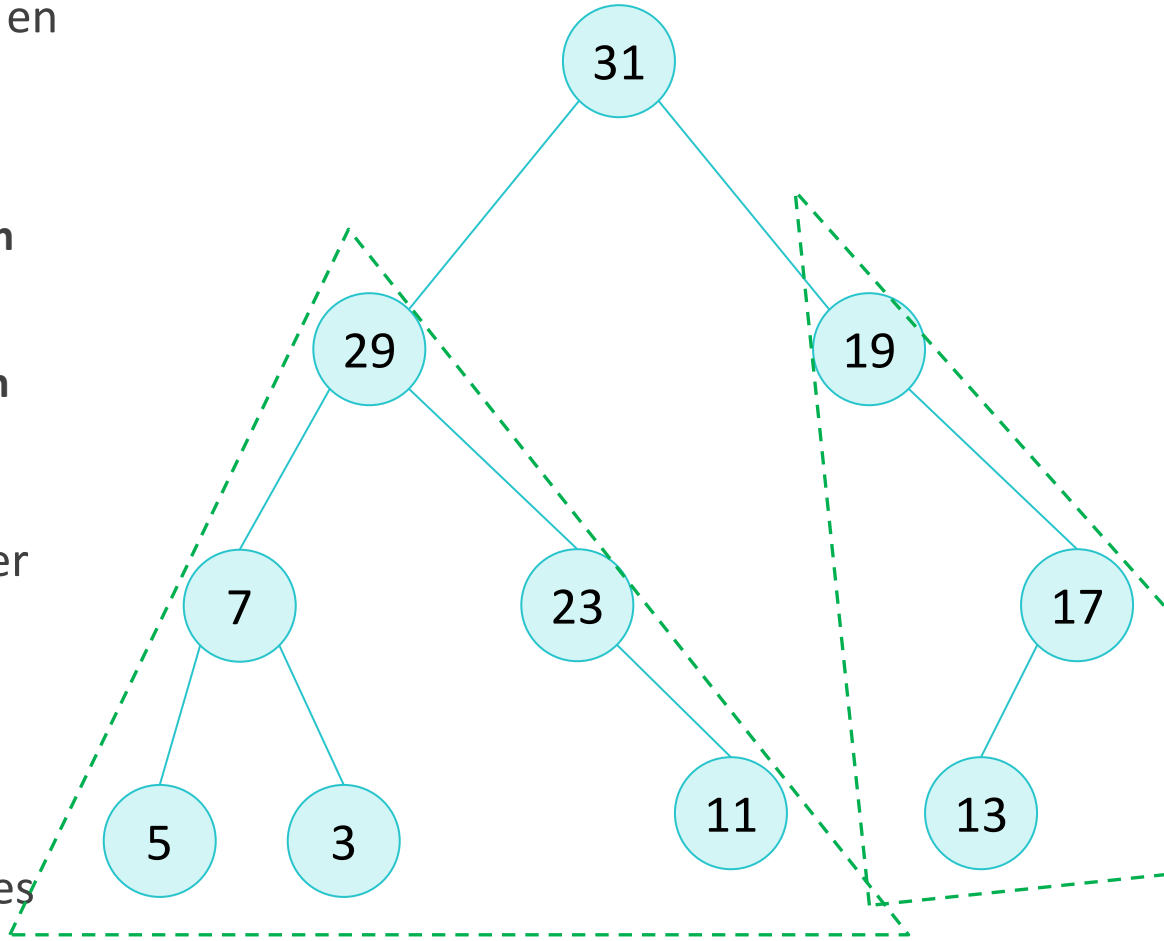
El heap binario: una estructura recursiva, con el elemento de mayor prioridad en la raíz

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**

Los valores a lo largo de cualquier ruta desde la raíz a una hoja aparecen en orden no creciente

… pero no hay ningún orden específico si uno revisa los valores que están en un mismo nivel



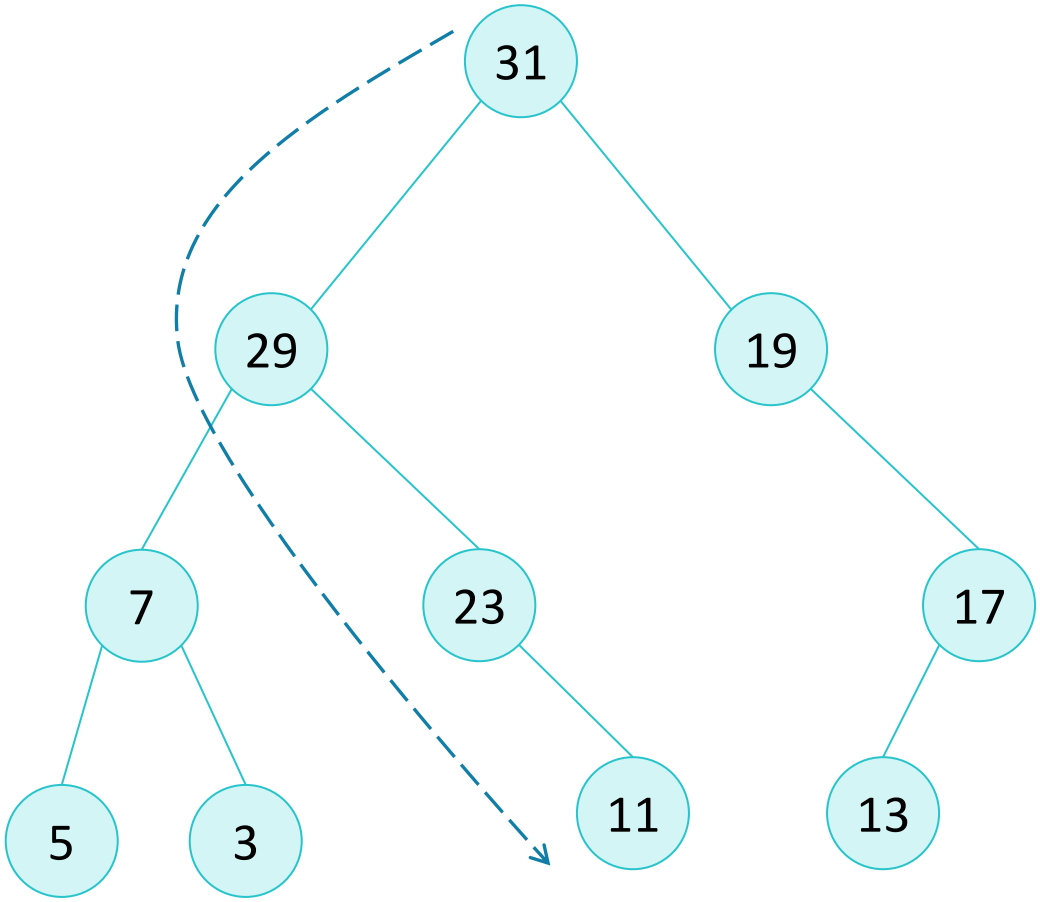
El heap binario: una estructura recursiva, con el elemento de mayor prioridad en la raíz

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

Los valores a lo largo de cualquier ruta desde la raíz a una hoja aparecen en orden no creciente

… pero no hay ningún orden específico si uno revisa los valores que están en un mismo nivel



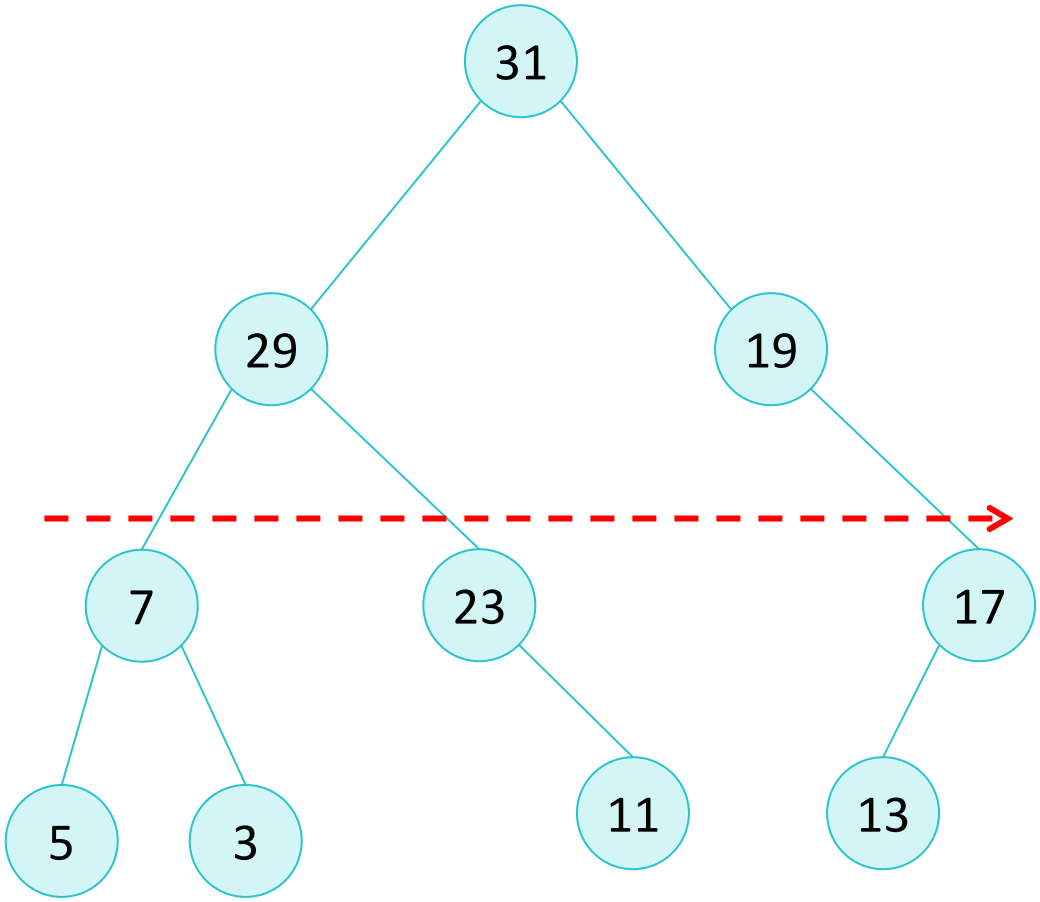
El heap binario: una estructura recursiva, con el elemento de mayor prioridad en la raíz

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

Los valores a lo largo de cualquier ruta desde la raíz a una hoja aparecen en orden no creciente

… pero no hay ningún orden específico si uno revisa los valores que están en un mismo nivel



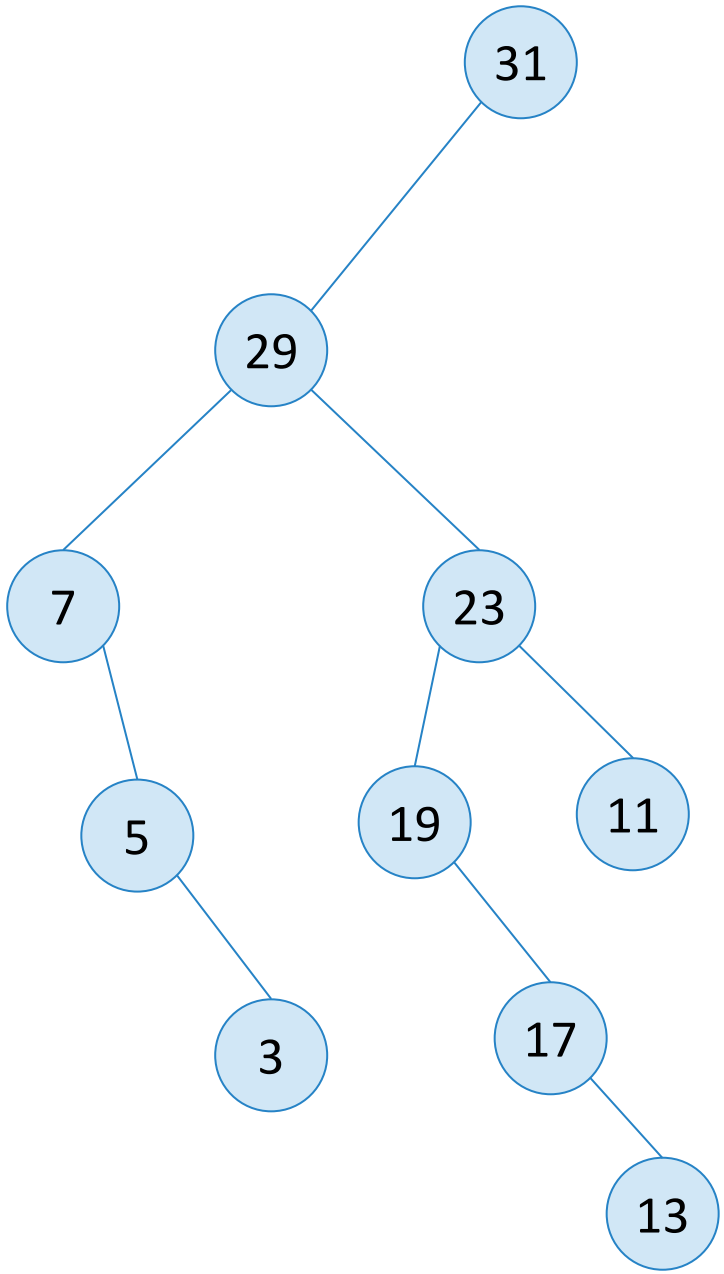
Altura de un heap binario

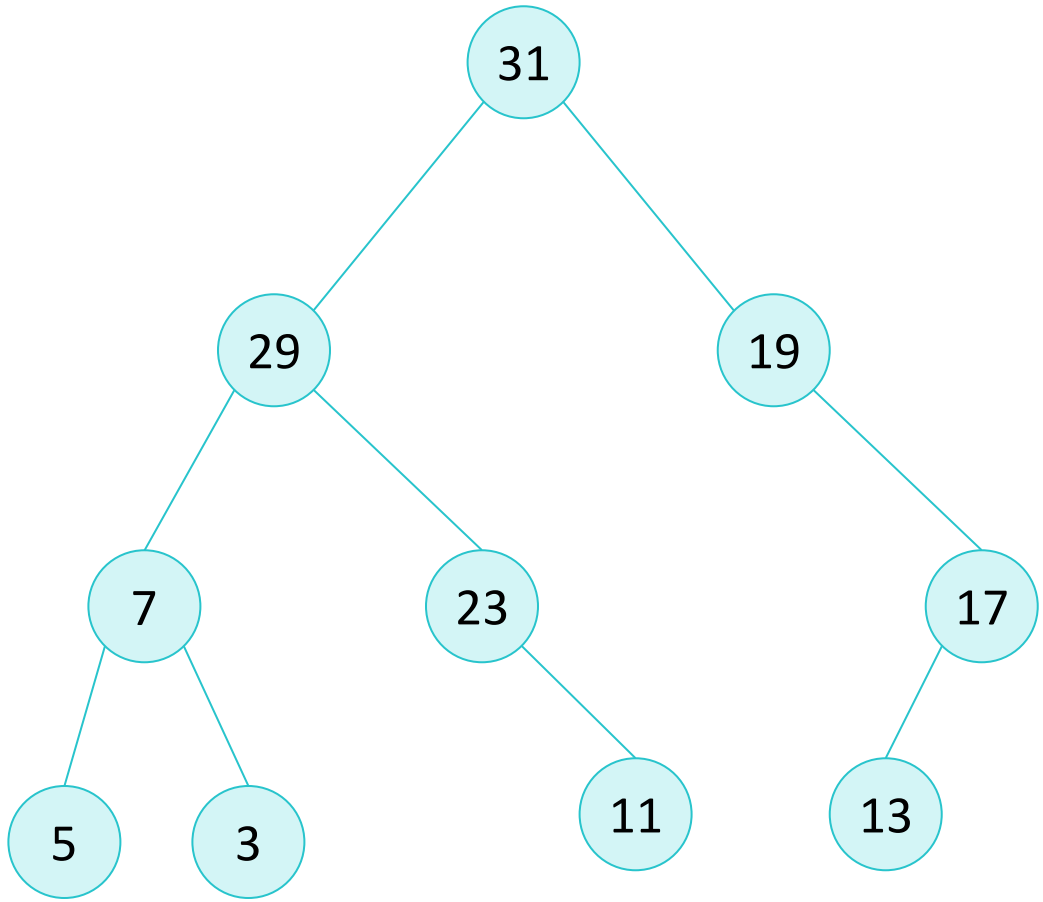
¿Cuál es la altura de un heap con n datos?

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

Heaps binarios como *árboles binarios llenos*

En principio, un heap binario podría tener cualquier forma que respete la estructura de árbol binario (p.ej., próx. diap.)


31
29
7
23
5
19
11
3
17
13


31
29
19
7
23
17
5
3
11
13

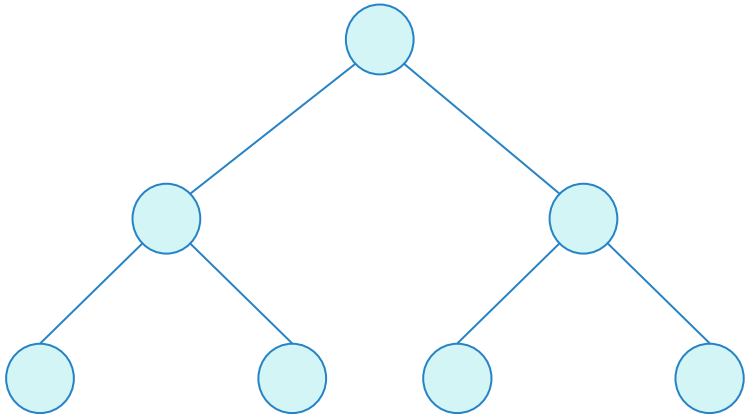
Heaps binarios como *árboles binarios llenos*

Sin embargo, como veremos, las dos operaciones fundamentales —*insertar* y *extraer*— requieren un número de pasos proporcional a la altura *h* del árbol

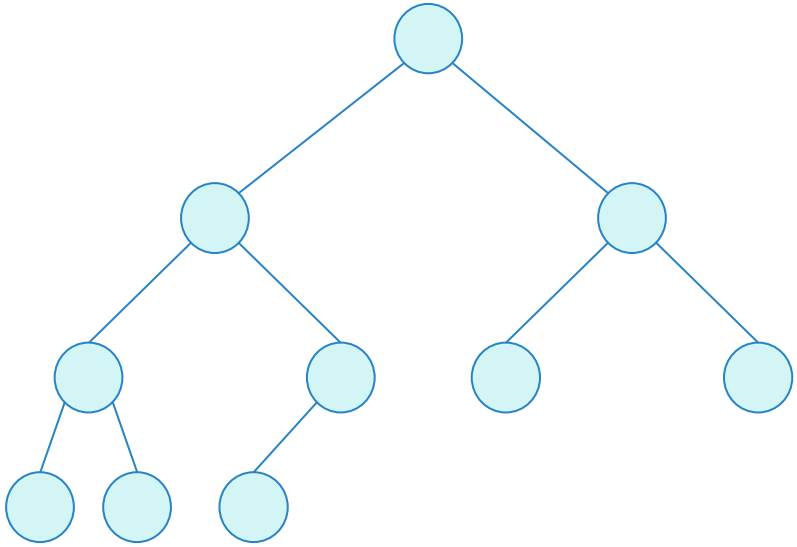
Heaps binarios como *árboles binarios llenos*

… por lo que, similarmente al caso de los ABBs, es preferible mantenerlos balanceados

… y, a diferencia de los ABBs, se los puede mantener balanceados eficientemente como *árboles binarios llenos* (próx. diap.)



árbol binario lleno, cuando
el número *n* de nodos cumple
$n = 2^d - 1$



árbol binario lleno, cuando
el número *n* de nodos cumple
$2^d \leq n < 2^{d+1}$

Una implementación simple de un heap binario

Al mantener los heaps balanceados, como árboles binarios llenos

… no sólo **minimizamos la altura del árbol**:

- de modo que insertar y extraer requieren un número de pasos O(log*n*)

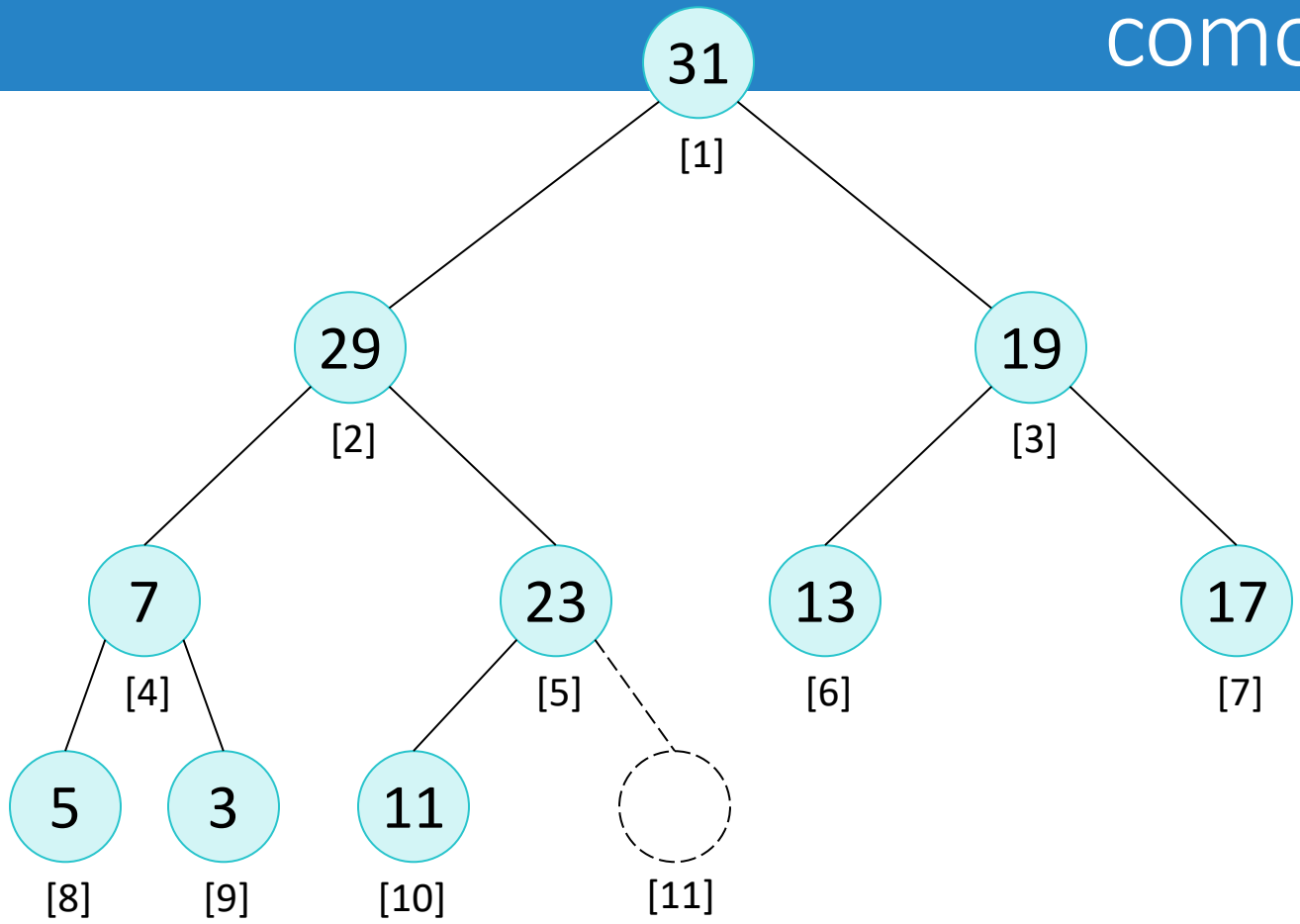
…

Una implementación simple de un heap binario … como **arreglo**

… sino que, además, **es posible implementar el heap de forma compacta en un arreglo**:

- nos ahorramos los punteros para recorrer el árbol —ir desde un nodo a uno de sus hijos o a su padre
- sólo tenemos que suponer una cantidad máxima de datos que pueden estar en el heap simultáneamente

Un heap binario (lleno) como un arreglo



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

Simplificación del recorrido del heap

La representación del heap como arreglo nos permite evitar los punteros para recorrer el heap

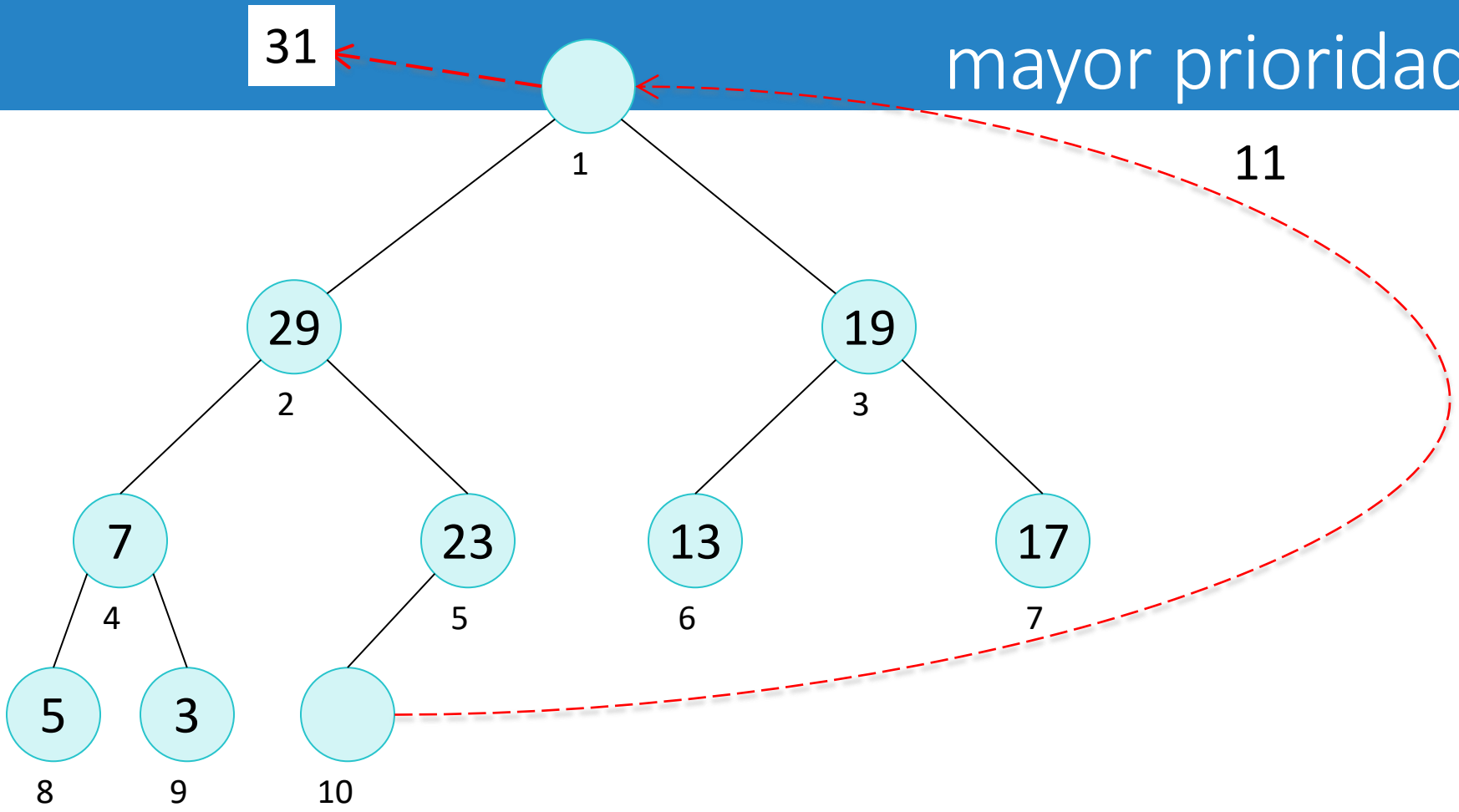
… en particular, si un elemento del heap ocupa la posición k del arreglo, entonces (fig. en diap. anterior):

- su hijo izquierdo está en la posición $2k$
- su hijo derecho está en la posición $2k+1$
- su padre está en la posición $k/2$ (división entera)

Operaciones sobre un heap

Al insertar y extraer elementos, el heap debe reestructurarse para que recupere sus propiedades

extraer el elemento de mayor prioridad



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

$\boldsymbol{extract}(\boldsymbol{H})$: —$\boldsymbol{H}$ *es el arreglo en que está almacenado el heap*

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

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

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

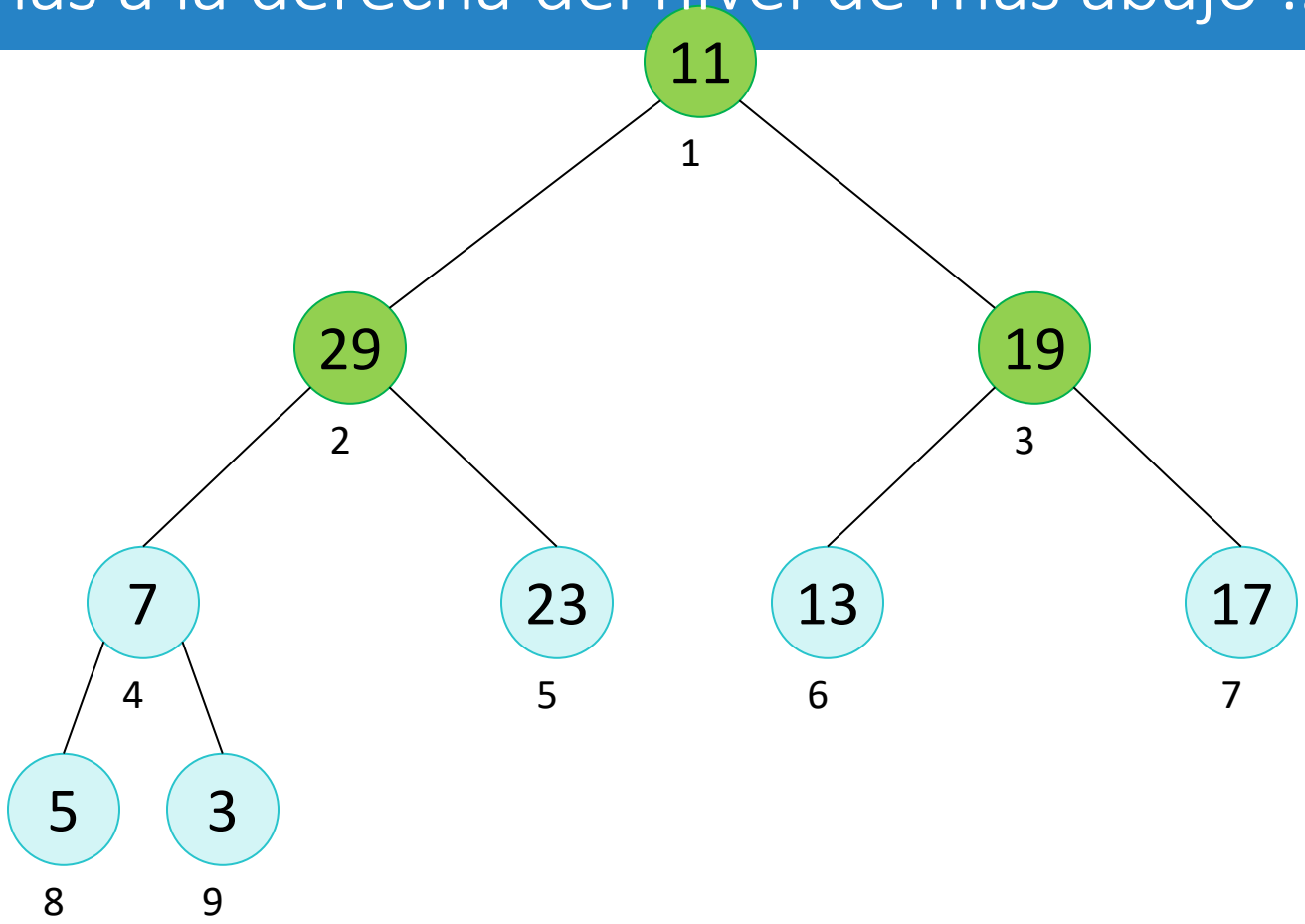
$\boldsymbol{H}[\boldsymbol{i}] \leftarrow \emptyset$

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

$\boldsymbol{return\ best}$

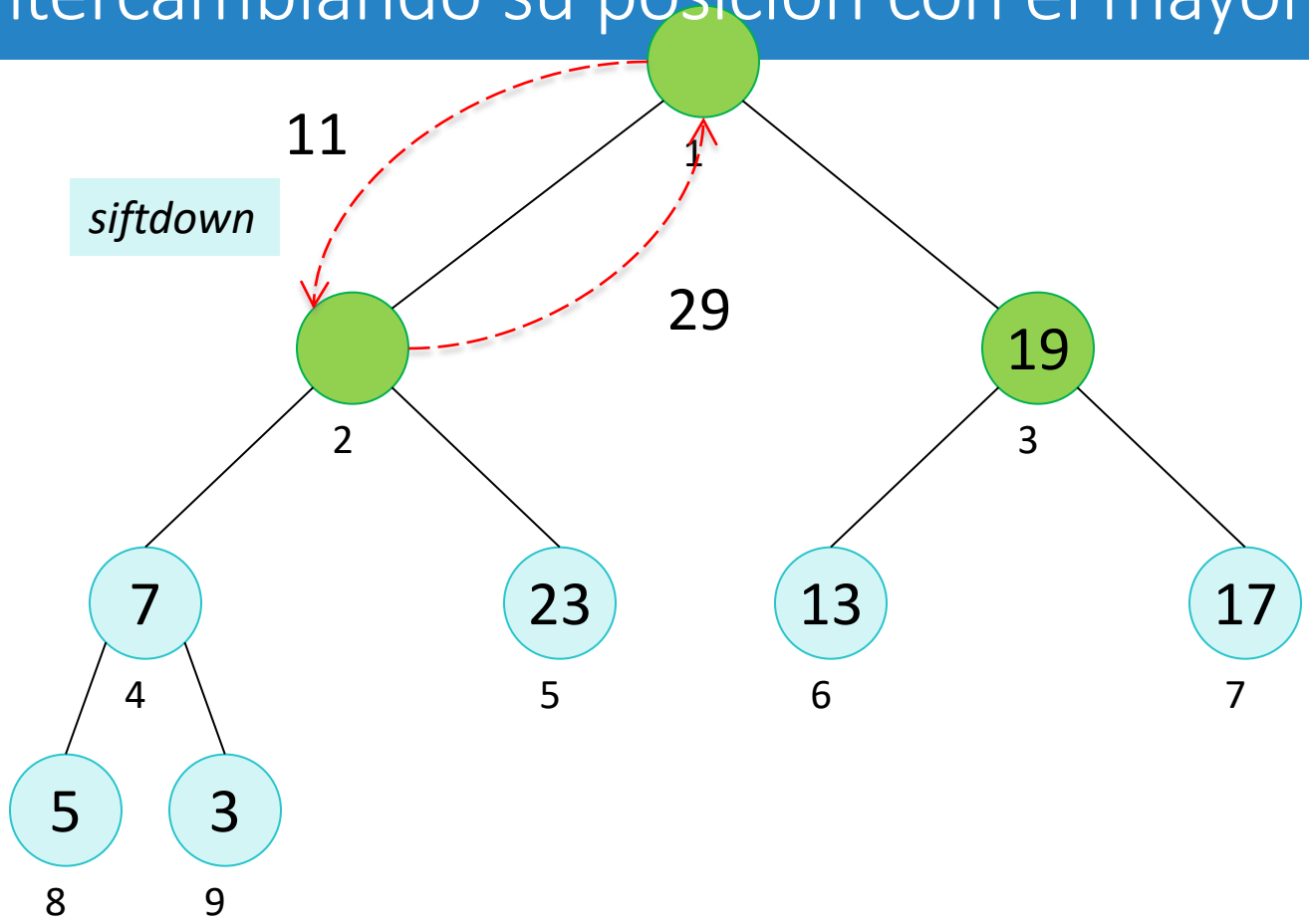
Reemplazamos, inicial y tentativamente, *H*[1] por *H*[*i*], porque de esta manera mantenemos el balance del heap; es decir, el heap sigue siendo un árbol binario lleno. Por supuesto, al poner *H*[*i*] en la posición de *H*[1] es posible que deje de cumplirse la propiedad de heap, lo cual hay que revisar y corregir en caso de ser necesario; para esto, llamamos a *siftdown*.

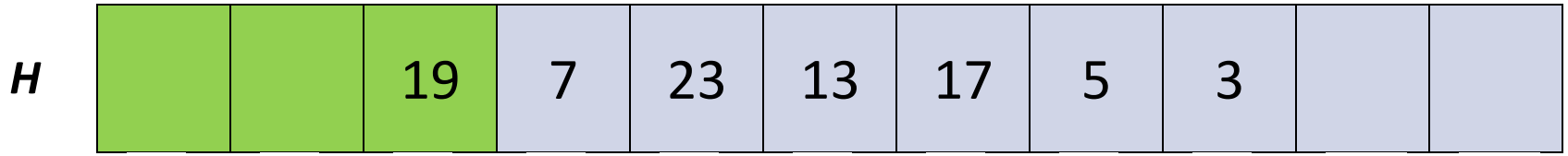
Ponemos (tentativamente) en la raíz el elemento de más a la derecha del nivel de más abajo …



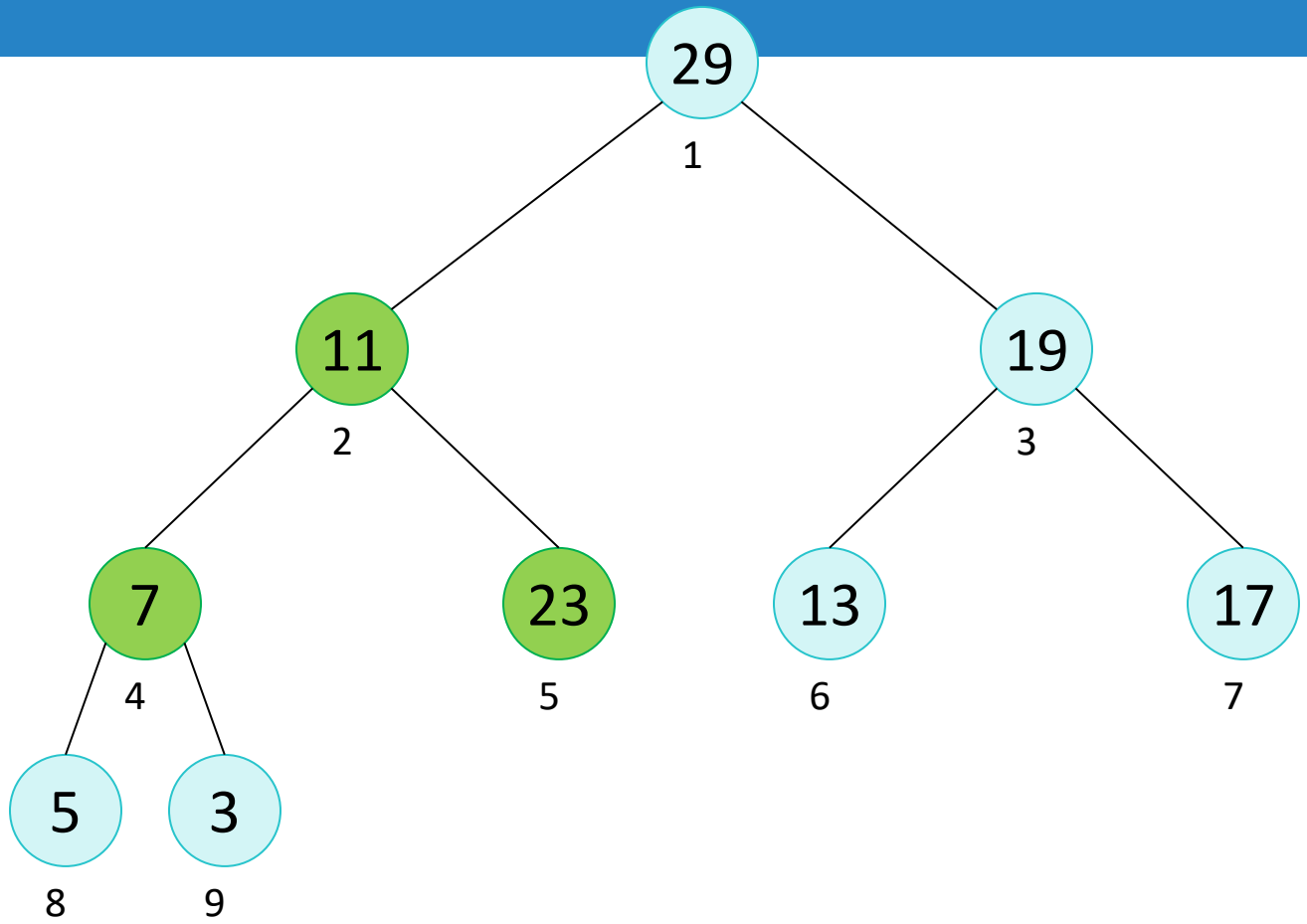
	1	2	3	4	5	6	7	8	9	10	11
H	11	29	19	7	23	13	17	5	3		

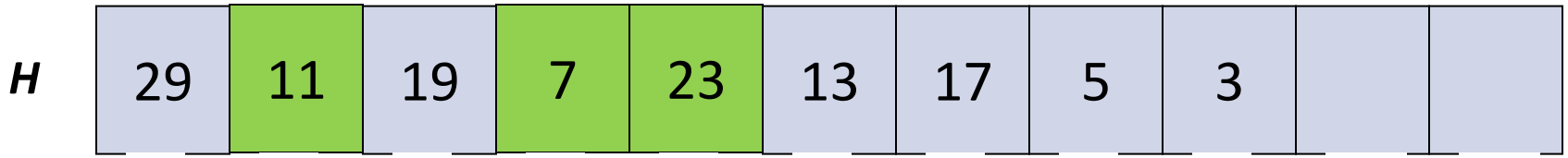
… y, si corresponde, hacemos ”bajar” este elemento intercambiando su posición con el mayor de sus hijos



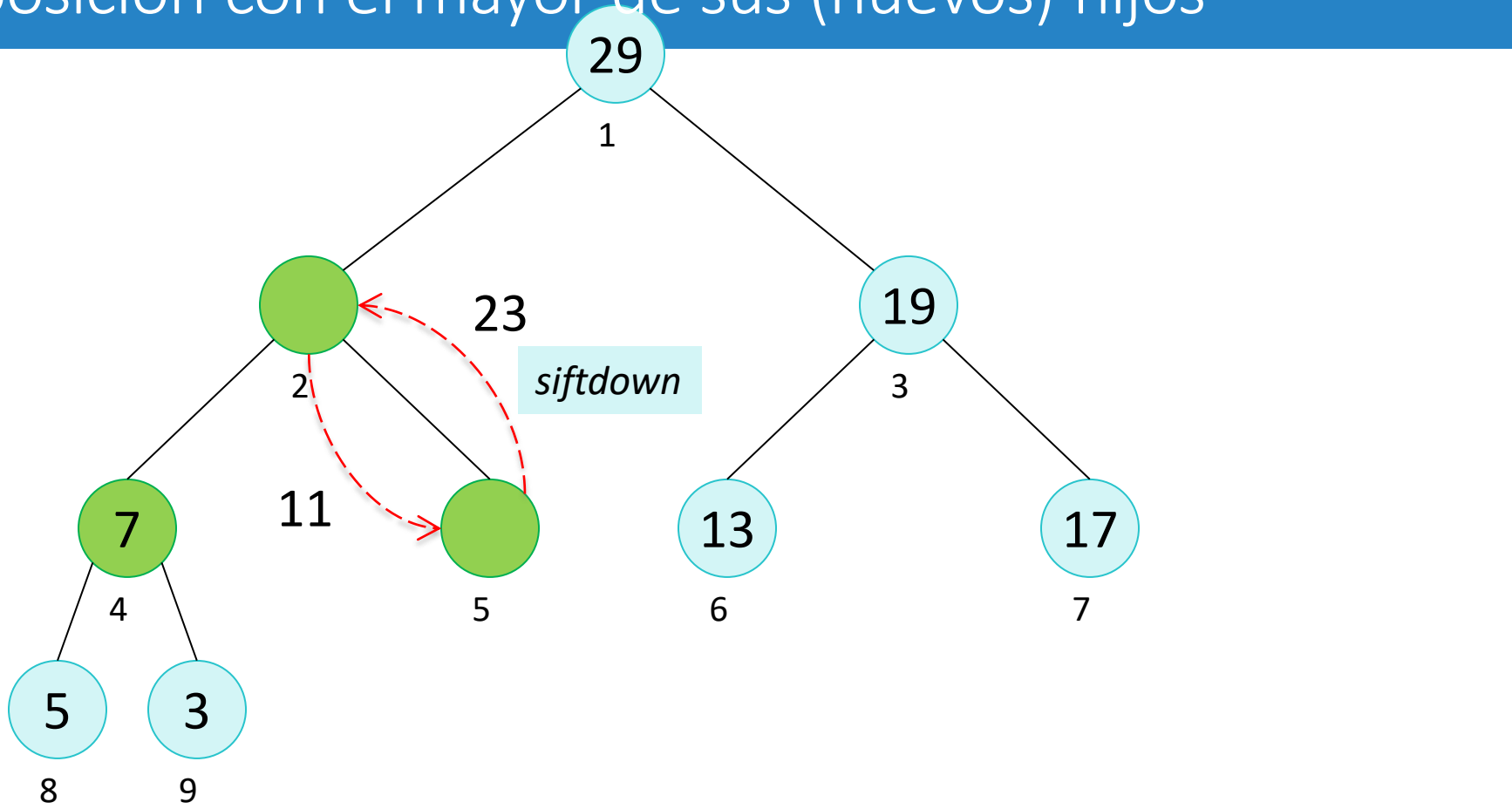


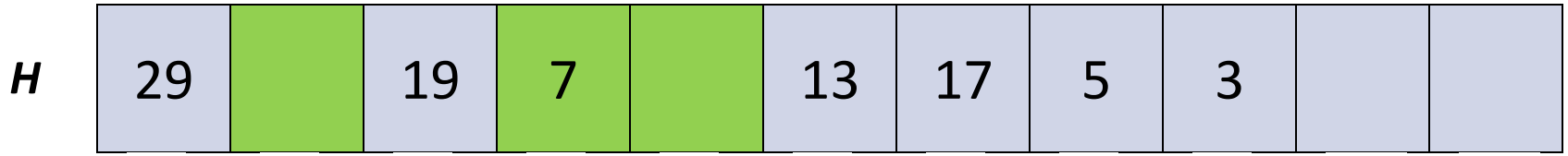
La nueva posición es nuevamente tentativa …

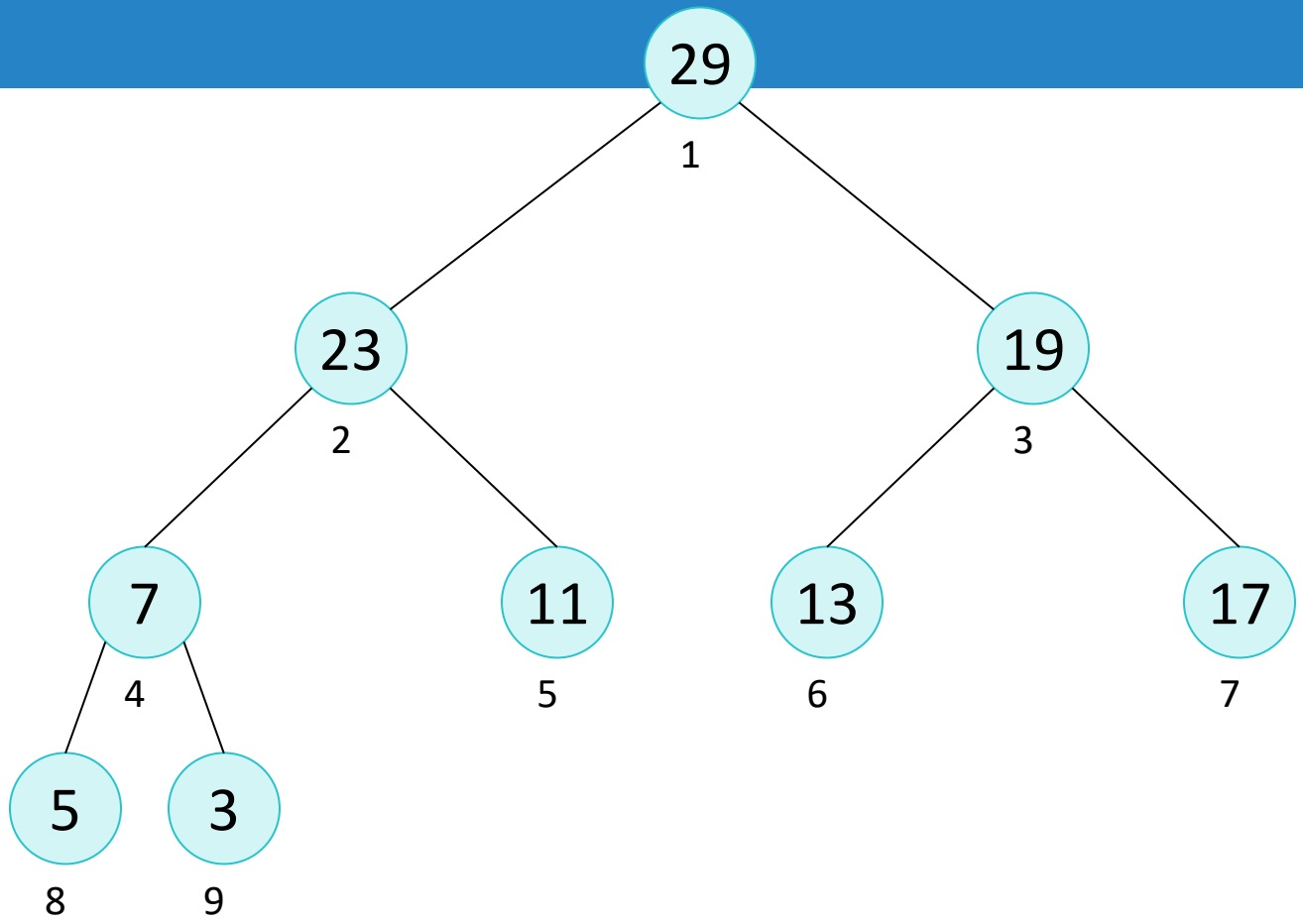




… y puede ser necesario volver a intercambiar esta posición con el mayor de sus (nuevos) hijos







H	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' ← el hijo de i de mayor prioridad
        if H[i'] > H[i]:
            H[i'] ⇄ H[i]
            sift down(H, i')
```

siftdown elige el elemento de mayor prioridad entre (a lo más) tres elementos, y, si es necesario, intercambia las posiciones de dos de ellos. En este caso, baja un nivel y hace una llamada recursiva (a *siftdown*). Por lo tanto, en cada nivel, *siftdown* toma tiempo O(1) … y, tal como lo anticipamos en la diap. 19, ***extract* toma tiempo O(log*n*)** para un heap con *n* elementos.

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

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

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

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

Al insertar un elemento en un heap, lo hacemos de modo de mantener el balance. Por supuesto, esto puede hacer que deje de cuplirse la propiedad de heap, lo cual hay que revisar y, en caso de ser necesario, corregir; para esto, llamamos a *siftup*.

$sift\ up(H, i)$:

    if i tiene padre:

        $i' \leftarrow$ el padre de i

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

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

            $sift\ up(H, i')$

siftup elige el elemento de mayor prioridad entre (a lo más) tres elementos, y, si es necesario, intercambia las posiciones de dos de ellos. En este caso, sube un nivel y hace una llamada recursiva (a *siftup*). Por lo tanto, en cada nivel, *siftup* toma tiempo O(1) … y, tal como lo anticipamos en la diap. 19, ***insert* toma tiempo O(log*n*)** para un heap con *n* elementos.

2) Conjuntos disjuntos: representación y operaciones

Nos interesan sólo dos operaciones:

Identificar en qué conjunto está un elemento (*find*)

Unir dos conjuntos (*union* … y que sólo quede esta unión y no los conjuntos originales)

¿Cómo podemos hacer esto de manera eficiente?

Representación, conceptualmente

Para cada conjunto, escogemos un **representante** : uno de sus elementos:

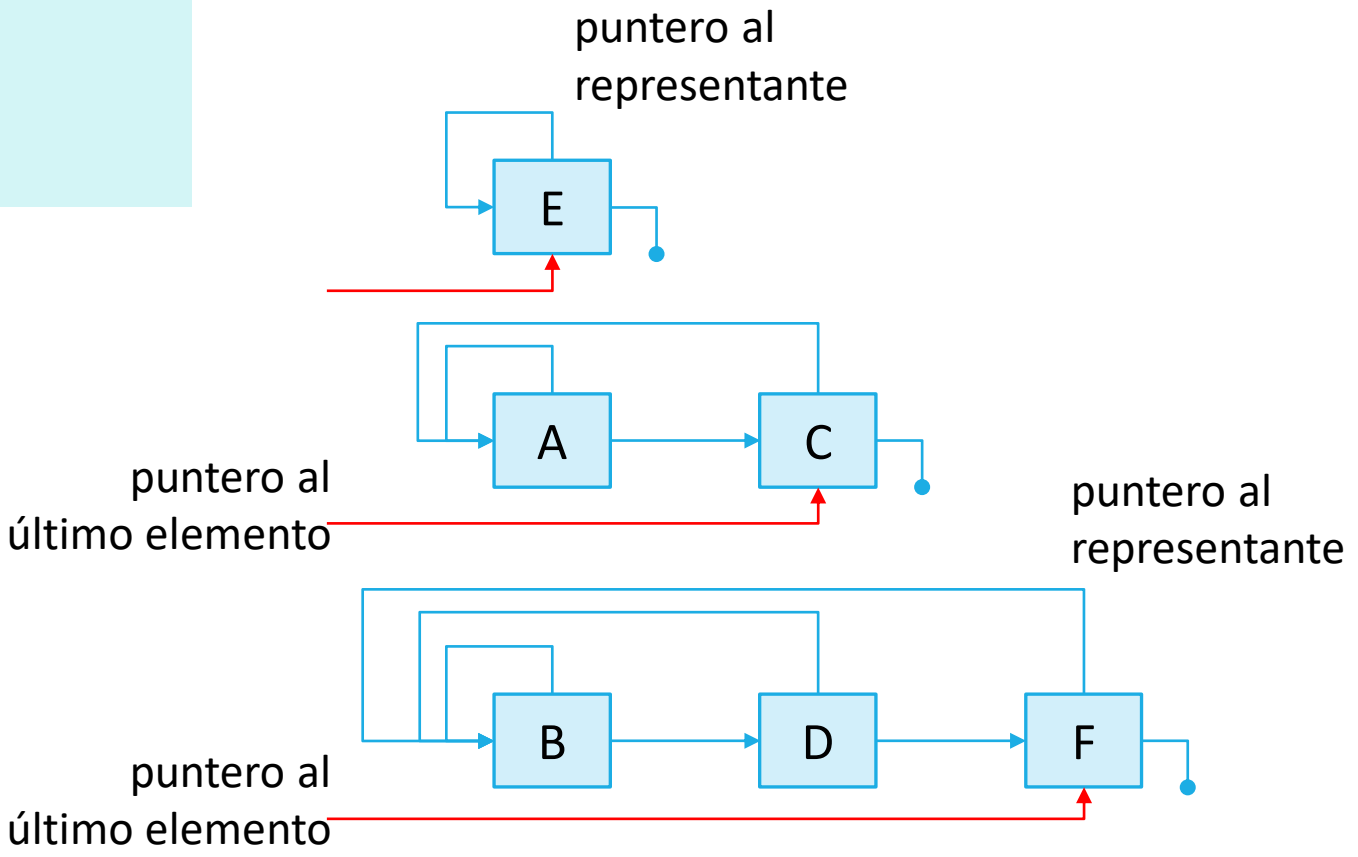
- si preguntamos por el representante de un conjunto dos veces, sin modificar el conjunto entre las consultas, debemos obtener la misma respuesta ambas veces

Cada elemento tiene una **referencia** a su representante, incluyendo el propio representante

Dos elementos están en el **mismo** conjunto si y sólo si tienen el mismo representante

Representación usando listas ligadas

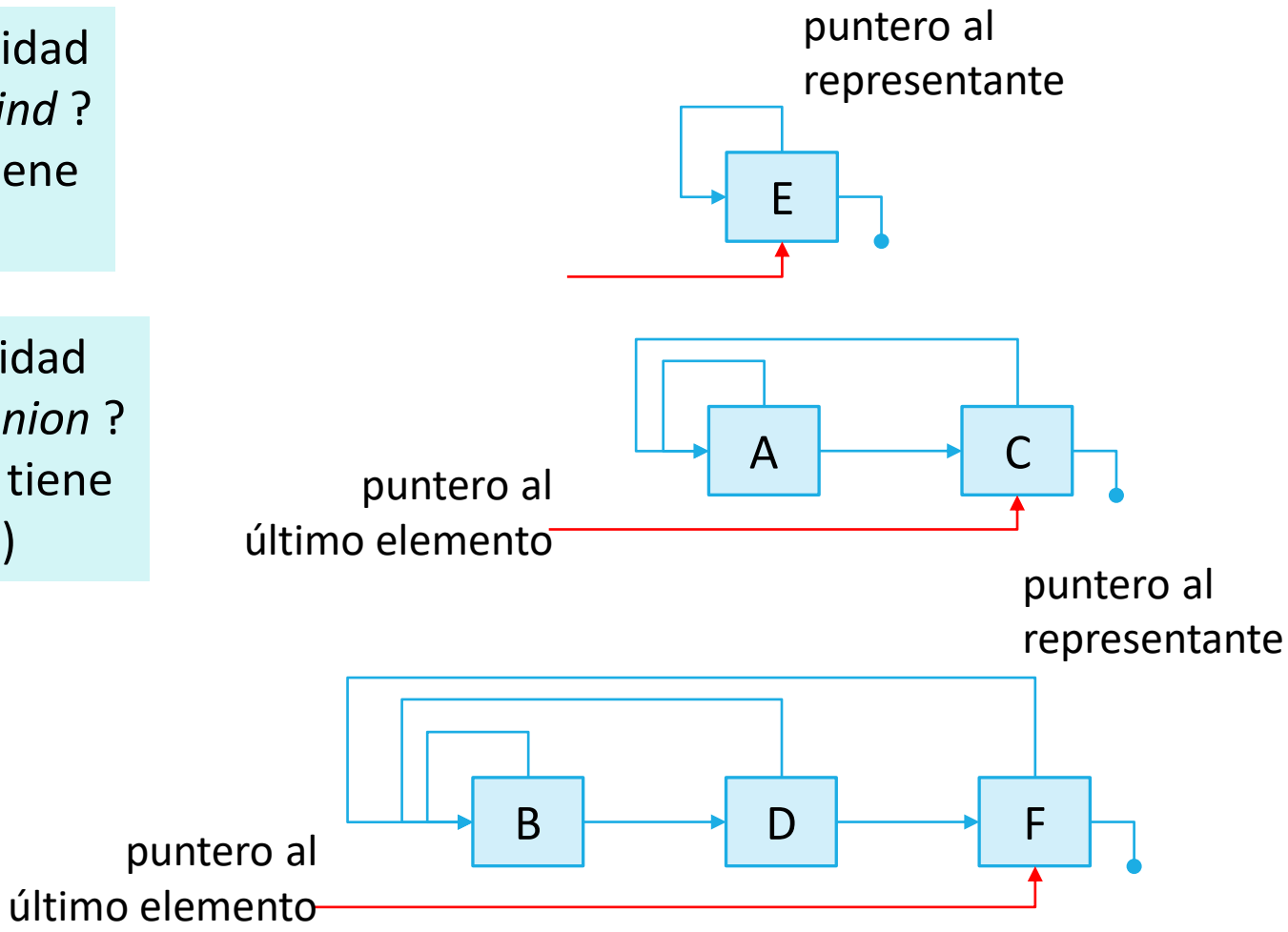
Tres conjuntos disjuntos:
{E}
{A, C}
{B, D, F}



… y su complejidad

¿Cuál es la complejidad de una operación *find* ?
La operación *find* tiene la forma *find*(D)

¿Cuál es la complejidad de una operación *union* ?
La operación *union* tiene la forma *union*(C, D)



Medimos la complejidad de la estructura en función de dos parámetros

n, el número de elementos individuales involucrados, que normalmente están **cada uno en un conjunto por sí mismo al inicio**

m, el número total de operaciones *union* y *find* ejecutadas, p.ej., hasta que todos los elementos pertenezcan a un mismo único conjunto final:

- cada *union* reduce el número de conjuntos en uno (los conjuntos son disjuntos)
- después de *n*–1 operaciones *union* sólo queda un conjunto (a lo más podemos hacer *n*–1 operaciones *union*)

Caso de uso: Construir un laberinto en un tablero de $c \times c$ casillas.

Inicialmente, cada casilla —c^2 elementos— forma un conjunto *singleton* por sí sola: ninguna está conectada a ninguna otra; la entrada está en la casilla 0 y la salida en la 24.

A lo largo del algoritmo, dos casillas están *conectadas* —se puede ir de una a otra— si y sólo si pertenecen al mismo conjunto.

Entrada

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Salida

La idea es botar muros (líneas verticales u horizontales) que conecten casillas desconectadas

Conectar casillas desconectadas (y permitir que se pueda ir de una a otra) significa unir los conjuntos en que está cada una.

Revisamos los muros aleatoriamente: si un muro separa casillas aún desconectadas, p.ej., el muro 13 / 18, entonces lo botamos; por el contrario, dejamos en pie muros que separan casillas que ya están conectadas, p.ej., el muro 8 / 13.

Estado del laberinto (en construcción) después de botar los muros 0 / 1, 4 / 9, 6 / 7, 7 / 8, 8 / 9, 9 / 14, 13 / 14, 10 / 11, 10 / 15, 16 / 17, 17 / 18 y 17 / 22.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Botar un muro que separa casillas desconectadas equivale a unir los conjuntos de cada casilla

P.ej., si en el estado anterior revisamos el muro 8 / 13, lo dejamos en pie, porque las casillas 8 y 13 ya están conectadas entre ellas (pertenecen al mismo conjunto); en cambio, si revisamos el muro 13 / 18, lo botamos porque las casillas 13 y 18 no están conectadas (pertenecen a conjuntos distintos) y así las conectamos.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Repetimos hasta que todas las casillas queden conectadas y haya un camino de la 0 a la 24

En este caso hicimos $n-1$ *unions* en total ($n = c^2$) y O(n) *finds* (el máximo número de *finds* es $2c^2 + 2c$).

Cada *find* toma tiempo O(1), pero ¿cuánto tiempo toman las $n-1$ *unions*?

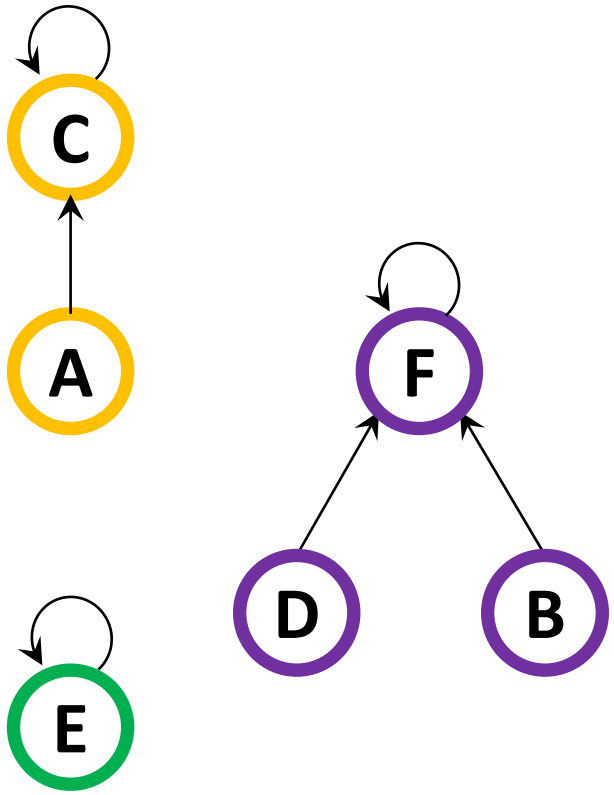
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Representación usando árboles

Un elemento cualquiera del conjunto ocupa la posición de la raíz del árbol y es el representante del conjunto; los otros elementos del conjunto son nodos intermedios u hojas del árbol. Cada nodo sólo triene un puntero a su padre en el árbol

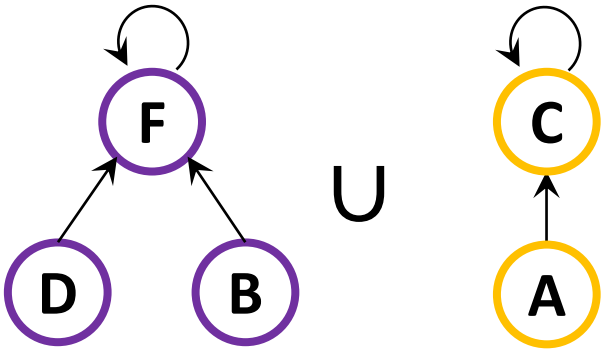
¿Cuál es la complejidad de una operación *find* ?

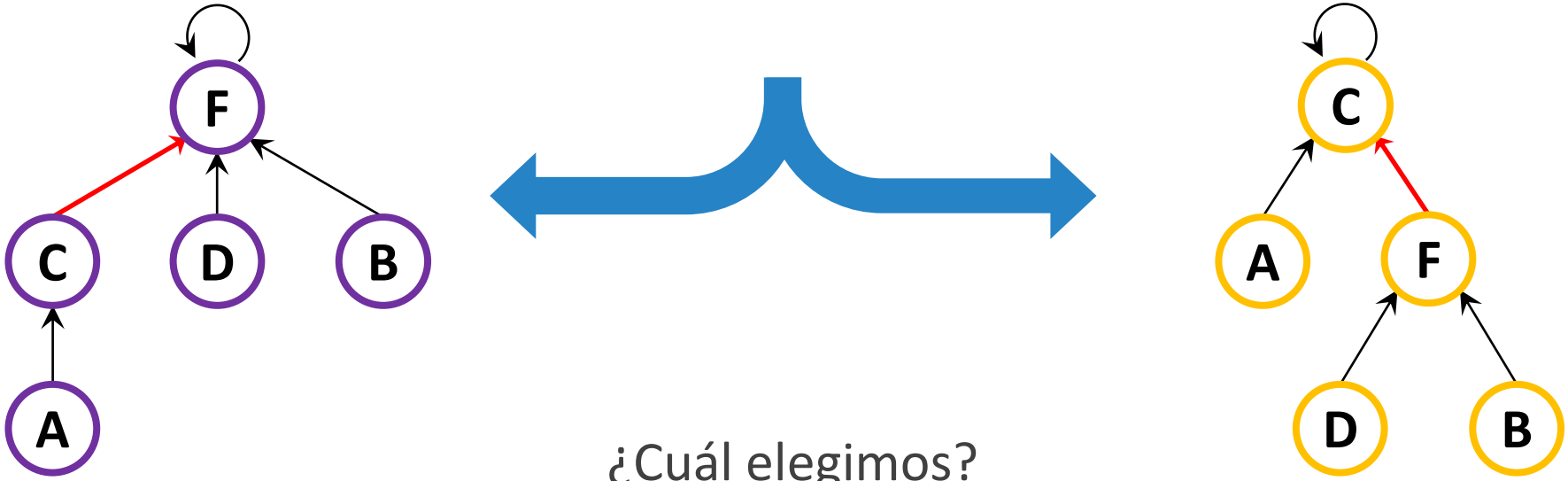
¿Cuál es la complejidad de una operación *union* ?



union: hacemos que el representante de un conjunto apunte al representante del otro

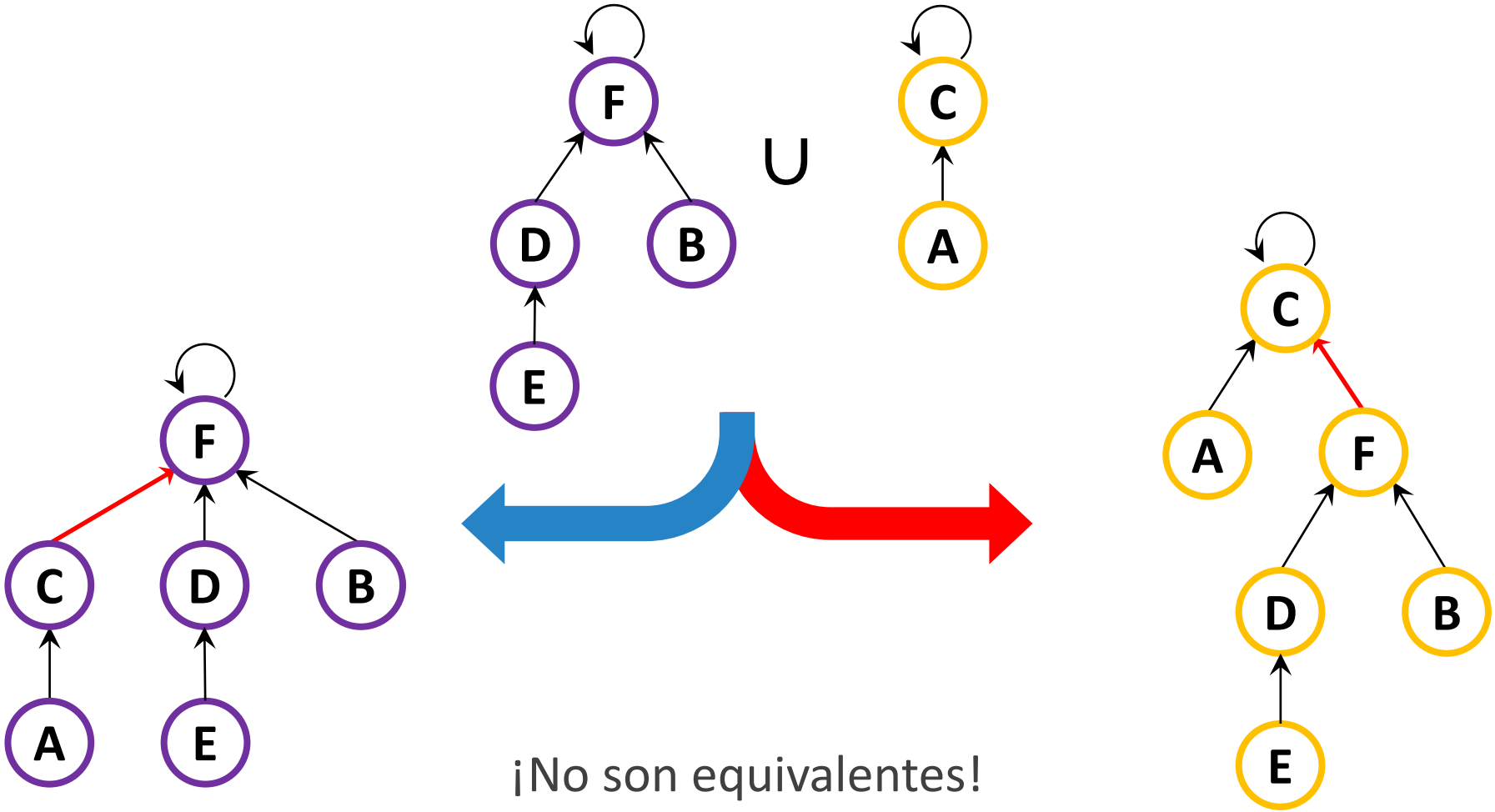
union es O(1)



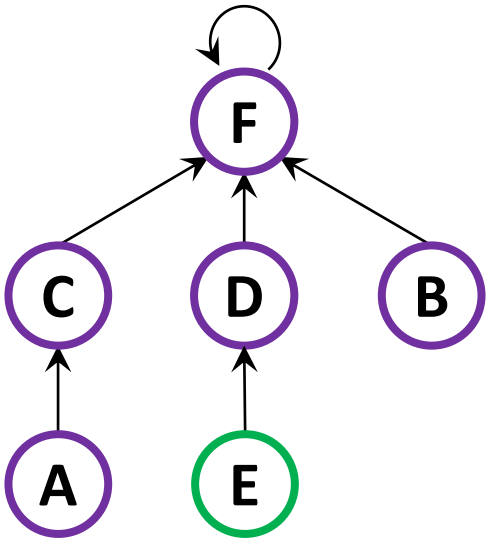


¿Cuál elegimos?

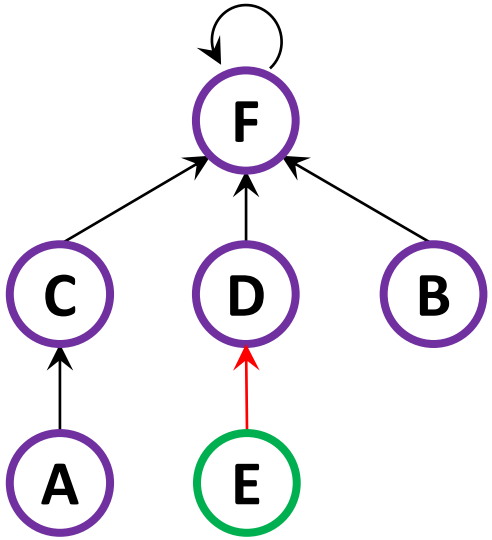
find ahora debe recorrer punteros; por lo tanto, una de las dos *union* es preferible a la otra



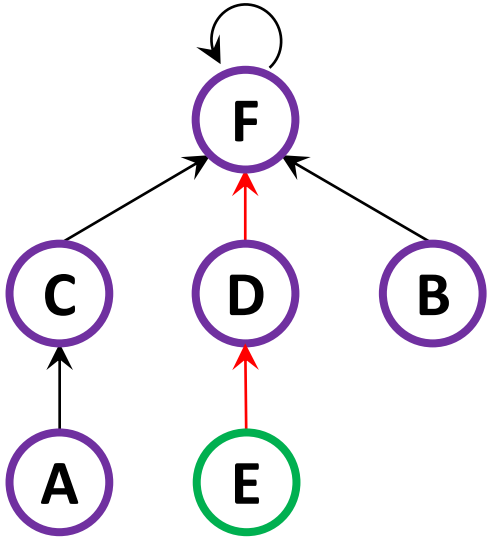
find(*E*) = …



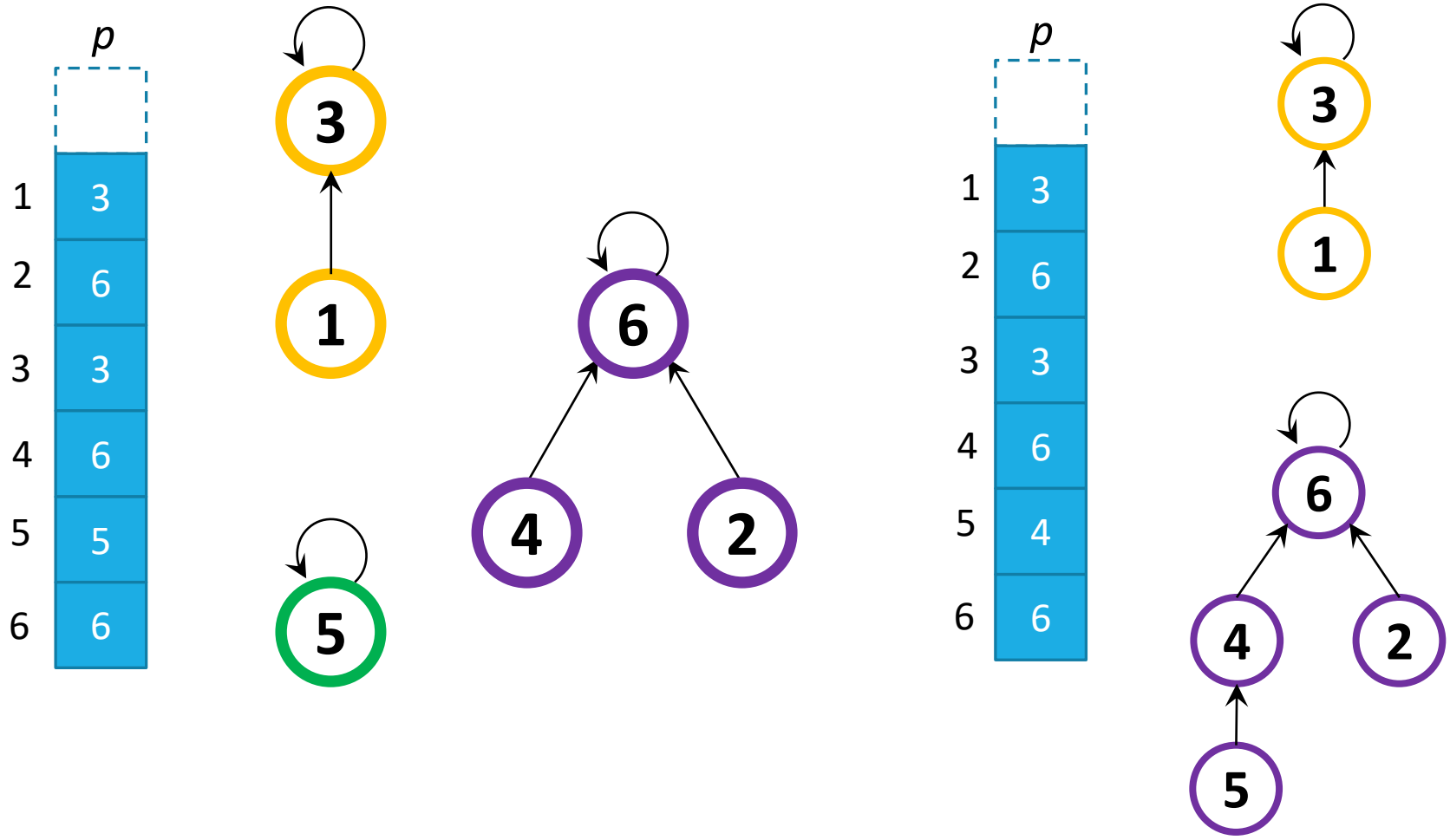
find(E) = *find*(D)



find(E) = *find*(F)



Representación de los árboles en un único **arreglo** *p*: *p*[*k*] es el padre de *k*



Con estas ideas, podemos idear una solución sencilla (pero subóptima).

Tenemos un arreglo auxiliar *h* donde *h*[*x*] guarda la altura de *x*.

```
find(x):
    if p[x] = x:
        return x
    else:
        return find(p[x])
```

```
union(x, y):
    x* ← find(x)
    y* ← find(y)
    if h[x*] ≤ h[y*]
        p[x*] ← y*
        h[y*] ← max(h[y*], h[x*] + 1)
    else:
        p[y*] ← x*
        h[x*] ← max(h[x*], h[y*] + 1)
```

¿Y cuánto tarda esto?

Union–Find logarítmico (subóptimo)

Una llamada a $find(x)$ toma tiempo $O(h[x^*])$, y $union$ toma lo mismo que $find$.

Vamos a demostrar que si x es la raíz de un arbol de tamaño k, entonces $h[x] \leq [\log_2(k)] + 1$.

Union–Find logarítmico (subóptimo)

Por dem: Si x es la raíz de un arbol de tamaño k, entonces $h[x] \leq \log_2 k + 1$.

Caso Base: Si $k = 1$, entonces $h[x] = 1 = 0 + 1 = \log_2(1) + 1$.

Paso Inductivo: Sean y y z raíces de dos árboles de tamaños a y b, donde $a + b = k$, que al hacer $union$ resulta el árbol de raíz x.

Supongamos s.p.g. que $h[y] \geq h[z]$. Entonces, tenemos $h[x] \leq h[y] + 1$.

Caso 1: $a \geq b$. Vemos que $h[x] \leq \log_2 a + 2$

$= \log_2 a + \log_2 2 + 1 = \log_2 2a + 1 \leq \log_2(a + b) + 1 = \log_2 k + 1$.

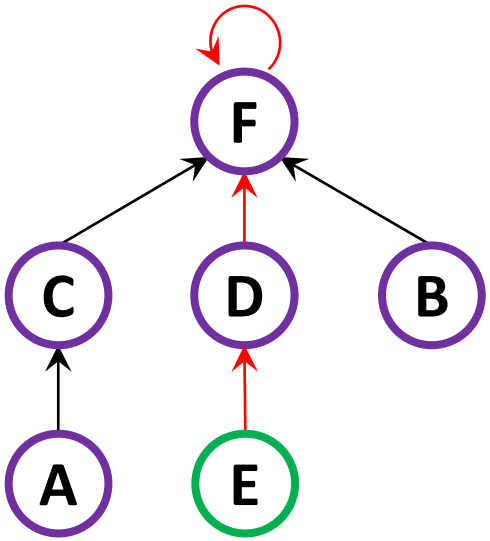
Caso 2: $a < b$. Usamos el hecho que $h[y] \geq h[z] \geq \log_2 b + 1$.

Luego, $h[x] \leq \log_2 b + 2 \leq \cdots \leq \log_2 k + 1$ usando la misma lógica. □

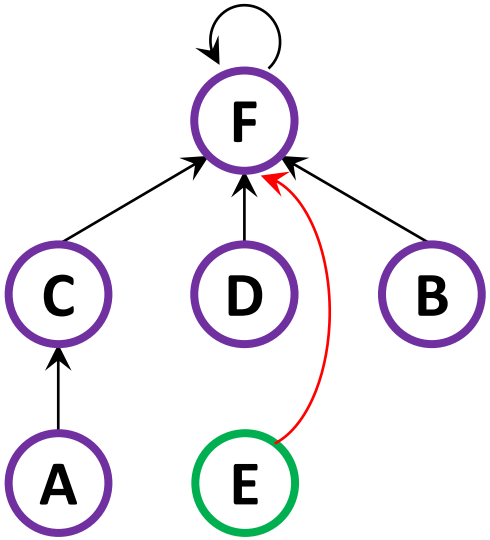
En consecuencia, cada operación toma $O(\log n)$.

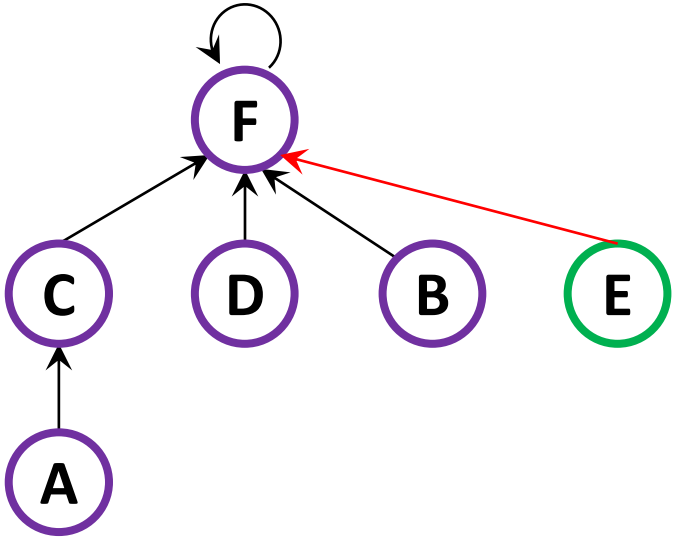
Ahora veremos como mejoramos esto.

Compresión o acortamiento de rutas



find(E) = F





Esto nos da el Union–Find real:

Tenemos un arreglo auxiliar ***r*** que guarda una estadística muuy similar a la altura.

$$\textbf{find}(x):$$
$$\quad \textbf{if } p[x] = x:$$
$$\quad\quad \textbf{return } x$$
$$\quad \textbf{else}:$$
$$\quad\quad x^* \leftarrow \textbf{find}(p[x])$$
$$\quad\quad p[x] \leftarrow x^*$$
$$\quad\quad \textbf{return } x^*$$

$$\textbf{union}(x, y):$$
$$\quad x^* \leftarrow \textbf{find}(x)$$
$$\quad y^* \leftarrow \textbf{find}(y)$$
$$\quad \textbf{if } r[x^*] \leq r[y^*]$$
$$\quad\quad p[x^*] \leftarrow y^*$$
$$\quad\quad r[y^*] \leftarrow \textbf{max}(r[y^*], r[x^*] + 1)$$
$$\quad \textbf{else}:$$
$$\quad\quad p[y^*] \leftarrow x^*$$
$$\quad\quad r[x^*] \leftarrow \textbf{max}(r[x^*], r[y^*] + 1)$$

Complejidad de las operaciones

La complejidad de una operación *find* depende de a cuál elemento se aplica

… aunque en el largo plazo todos los árboles podrían terminar teniendo profundidad 1, si hay suficientes operaciones *find*

Se puede demostrar que el costo promedio de una operación *find* en un conjunto de n elementos es $O(\log^* n)$

… en que log* es el número de veces que $\log_2$ tiene que ser aplicado iterativamente hasta que el resultado sea ≤ 1

P.ej., leyendo de derecha a izquierda

$$0.54 = \log_2(\ 1.45 = \log_2(\ 2.73 = \log_2(6.64 = \log_2(\ 100\ ))))$$

… de modo que $\log^*(100) = 4$

La función log* crece muy lentamente

El n más pequeño para el cual $\log^* n$ es 5 es $n = 2^{16} = 65536$

… y va a quedarse en 5 para todos los números razonables (hasta 2^{65536})

→ para cualquier uso práctico, consideramos que $\log^* n$ es casi constante (aunque teóricamente tiende a ∞)

Así, las $O(n)$ operaciones *find* toman $O(n \log^* n)$

… y la complejidad de las $O(n)$ *finds* más las n–1 *unions* es

… $O(n \log^* n + n)$