

HEAPSORT

Método de ordenación basado en un montículo.



UNIVERSIDAD
COMPLUTENSE
MADRID

ALBERTO VERDEJO

Heapsort abstracto

- ▶ Utiliza una cola de prioridad en vez de un montículo directamente.

```
template <typename T>
void heapsort_abstracto(std::vector<T> & v) {
    PriorityQueue<T> colap;
    for (auto const& e : v)
        colap.push(e);
    for (int i = 0; i < v.size(); ++i) {
        v[i] = colap.top(); O(1)
        colap.pop(); log(N)
    }
}
```

Para ordenar un vector usando una cola de prioridad, recorremos todos los elementos insertándolos en una cola de prioridad y después extraerlos uno a uno guardándolos en las posiciones del vector.

$O(N \log N)$.

El coste en tiempo está en $\Theta(N \log N)$, y en espacio adicional en $\Theta(N)$.

Heapsort

- Podemos ahorrarnos ese espacio adicional si utilizamos el mismo vector para representar el montículo auxiliar.

FASE 1

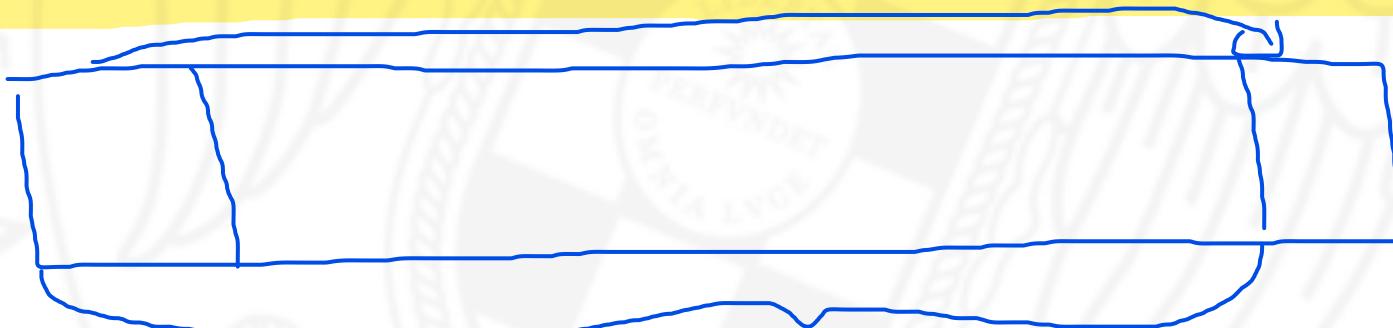
- Primero el vector se convierte en un montículo.

En el vector los elementos estarían ordenados de cualquier manera y hay que reordenarlos para que el vector represente un montículo.

FASE 2

- Después se va extrayendo sucesivamente el elemento más prioritario para colocarlo al final del vector, en la parte ya no necesaria para almacenar el montículo, cada vez más pequeño.

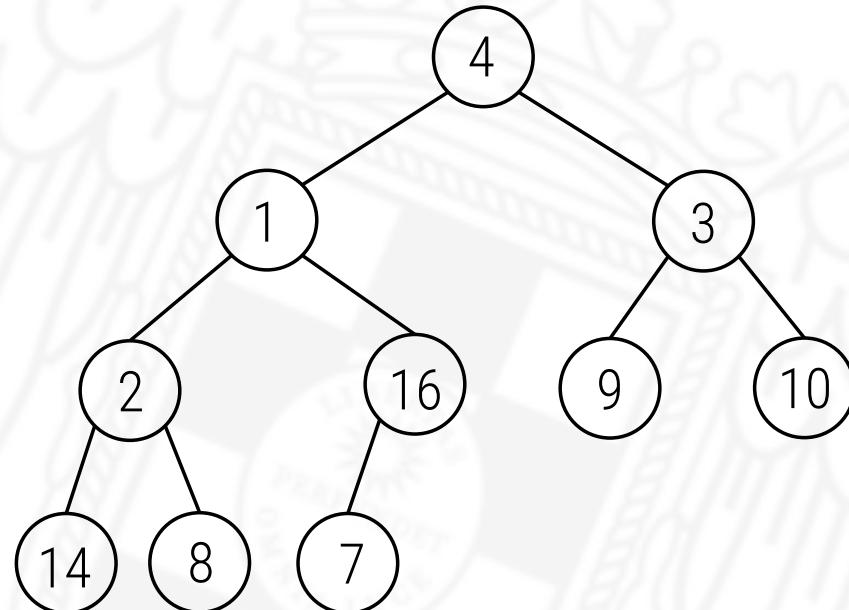
En la segunda fase los elementos van saliendo del montículo y los vamos guardando al final del vector. Al principio, el montículo ocupa todo el vector pero si sacamos el elemento más prioritario se deja de usar la última posición, para guardar ahí el elemento que hemos sacado,



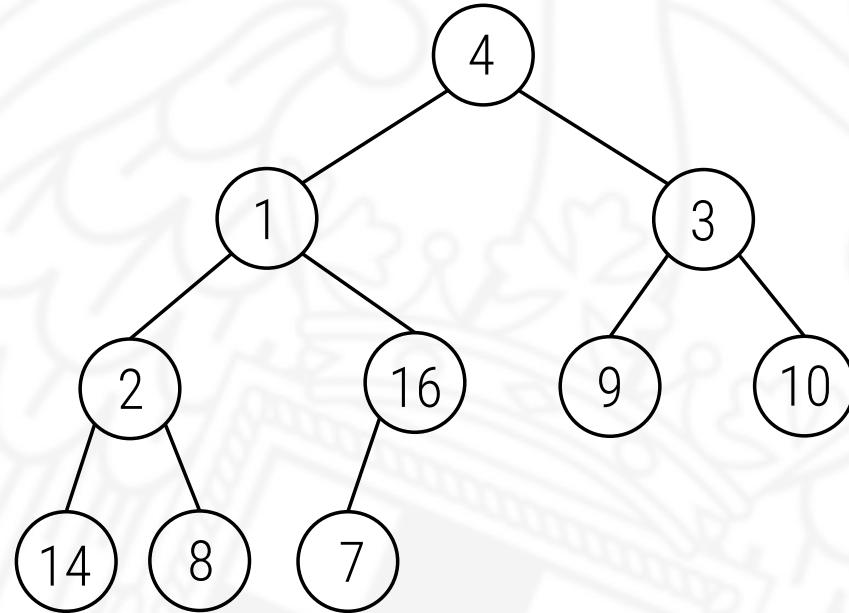
Convertir el vector en un montículo

Vector que no es un montículo.

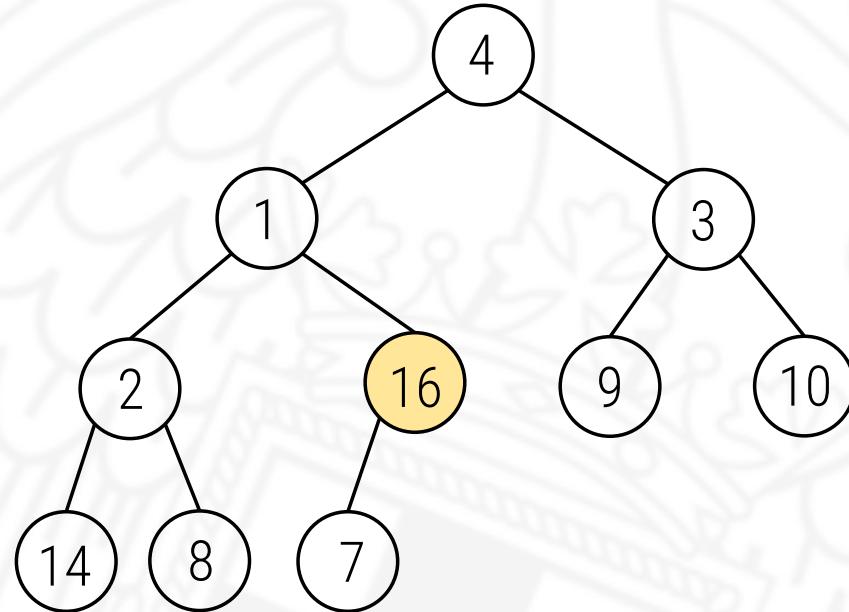
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



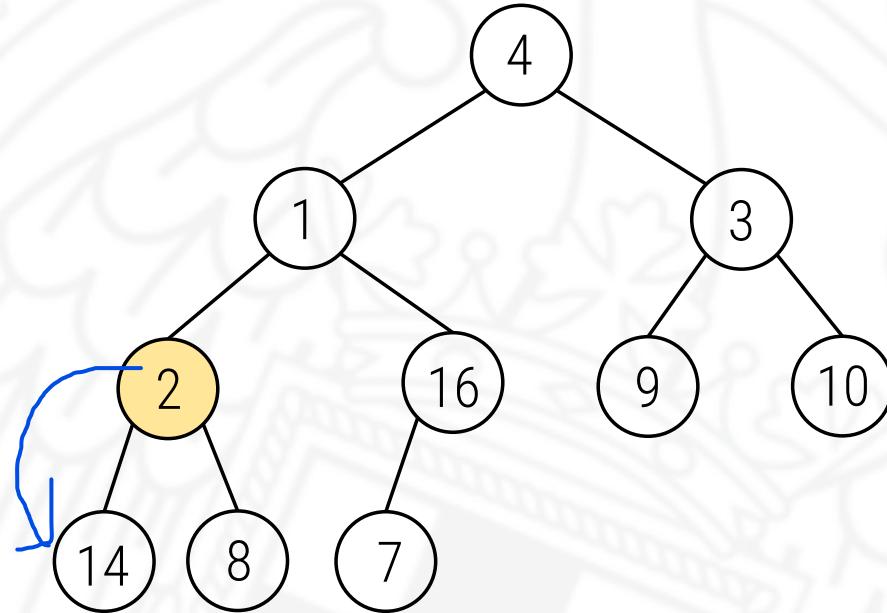
Convertir el vector en un montículo



Convertir el vector en un montículo

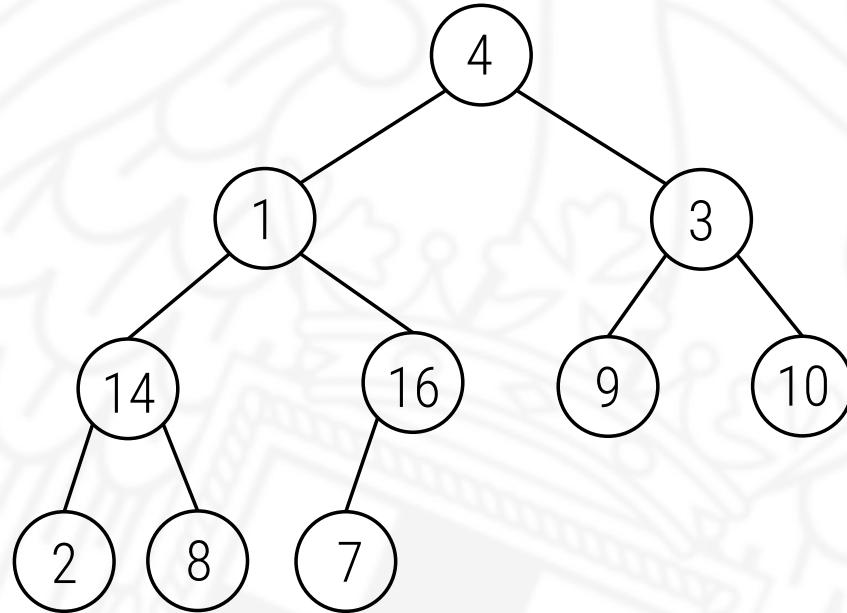


Convertir el vector en un montículo

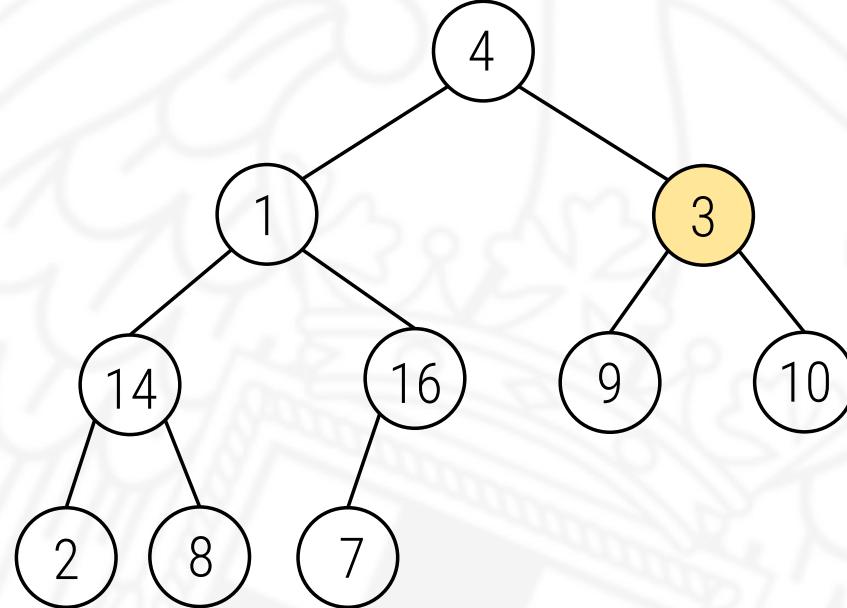


Es un montículo de máximos.

Convertir el vector en un montículo

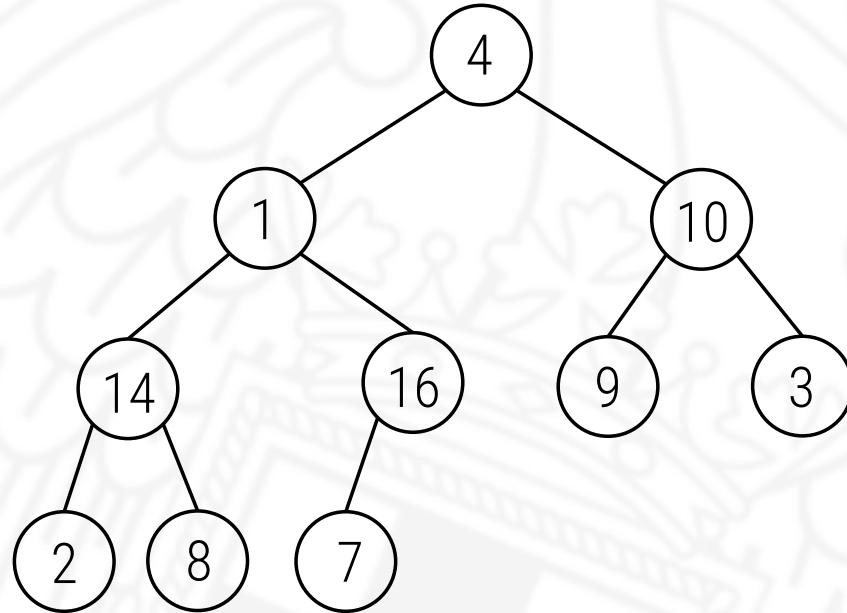


Convertir el vector en un montículo



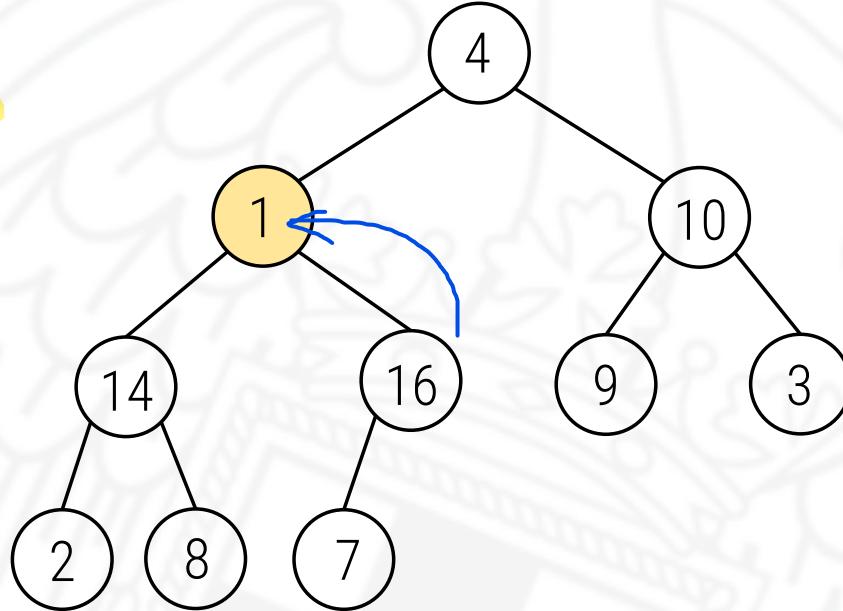
Se intercambia con el mayor de sus hijos.

Convertir el vector en un montículo

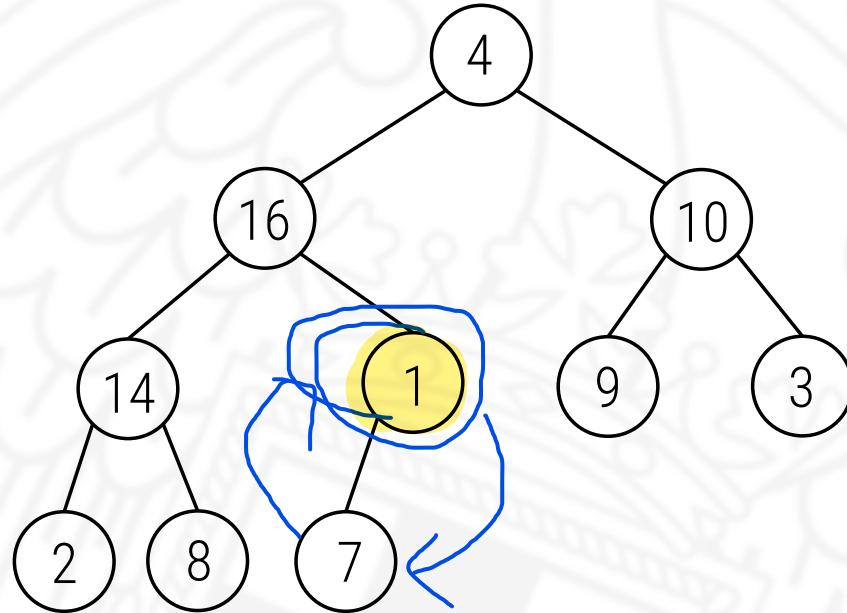


Convertir el vector en un montículo

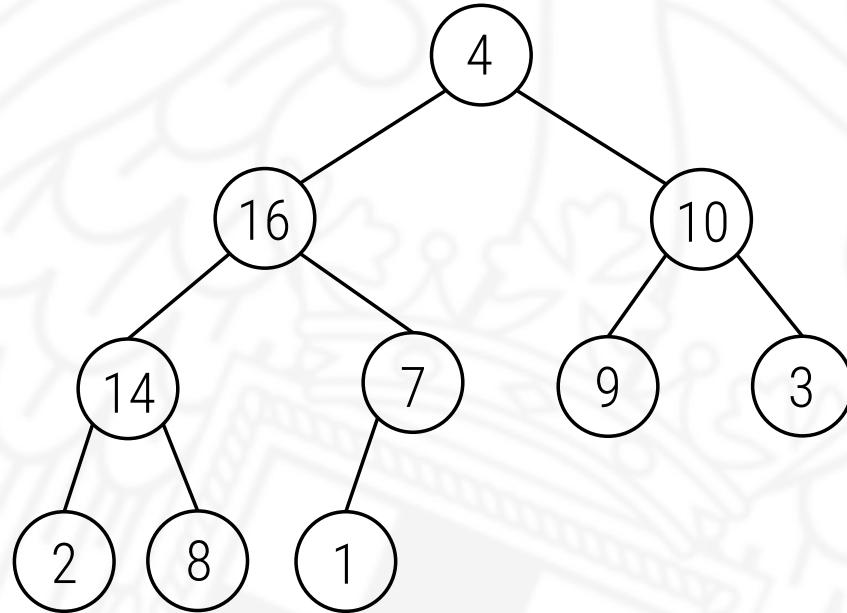
Se intercambia con el mayor de sus hijos.



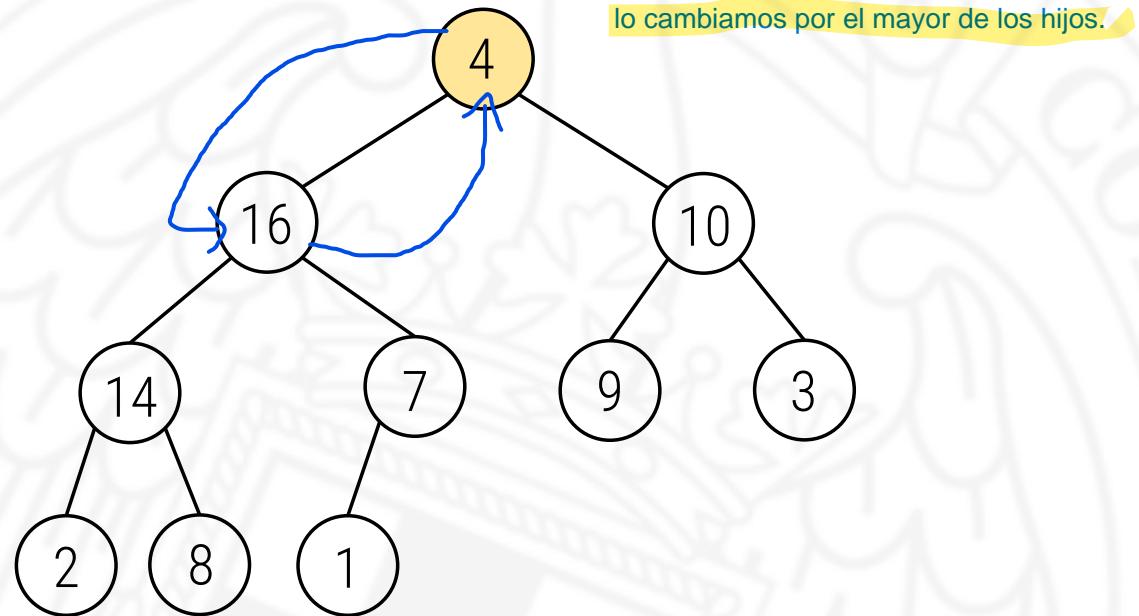
Convertir el vector en un montículo



Convertir el vector en un montículo

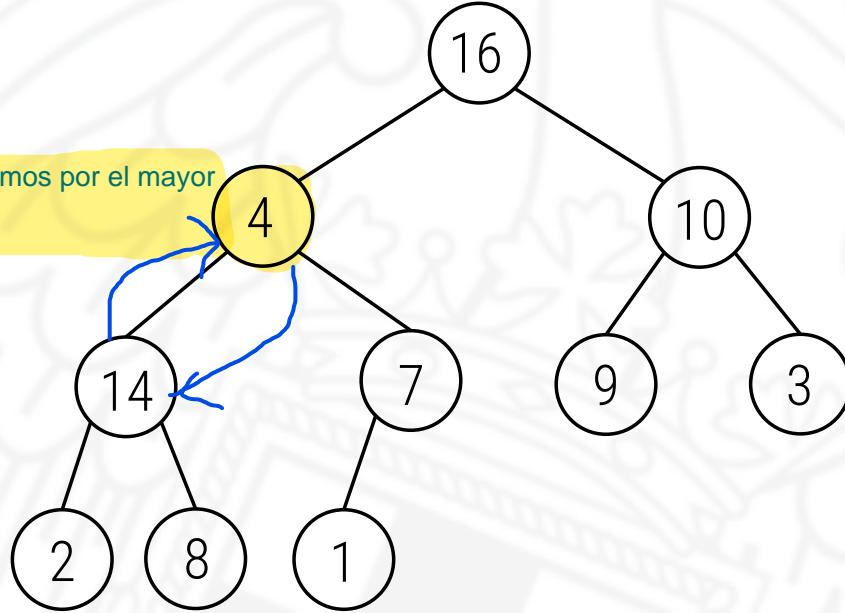


Convertir el vector en un montículo

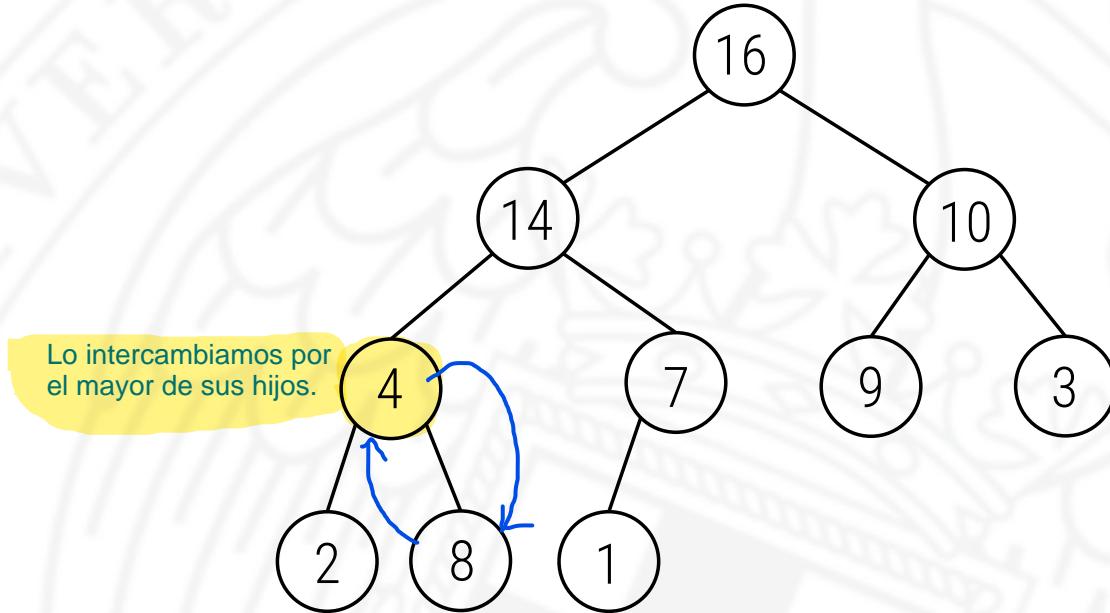


Convertir el vector en un montículo

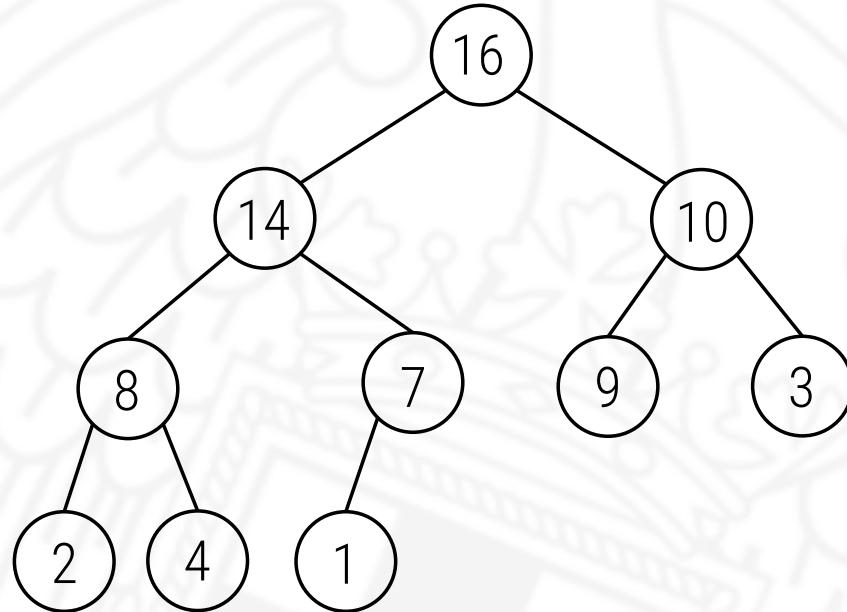
Lo intercambiamos por el mayor de sus hijos.



Convertir el vector en un montículo



Convertir el vector en un montículo



16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---

Coste (amortizado)

ultimo nivel si árbol está completo

nivel	nodos	hunden
h	2^{h-1}	nada
$h - 1$	2^{h-2}	cada uno 1
$h - 2$	2^{h-3}	cada uno 2
:	:	
i	2^{i-1}	cada uno $h - i$
:	:	
1	1	$h - 1$

No puedes hundir a un nodo que se encuentra en el último nivel porque NO HAY MÁS NIVELES.

Es como si cada hundimiento fuera de coste constante.

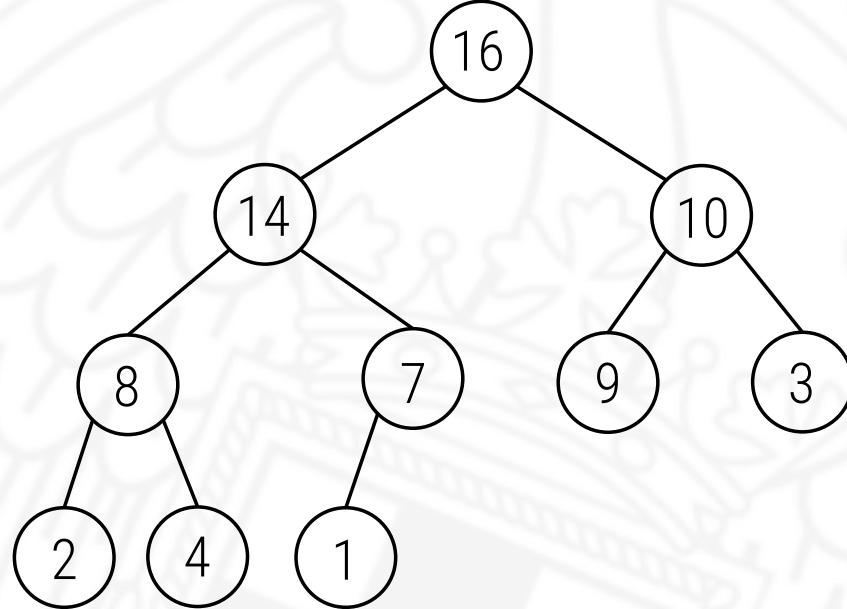
$$\sum_{i=1}^{h-1} (h-i)2^{i-1} = \sum_{j=2}^h (j-1)2^{h-j} < \sum_{j=1}^h j2^{h-j} = 2^h \sum_{j=1}^h \frac{j}{2^j}$$

$$= 2^h \left(2 - \frac{h+2}{2^h} \right) \leq 2^{h+1} = 2^{\lfloor \log N \rfloor + 2}$$

$\in O(N)$

Cota mejor que
 $O(N \log(N))$

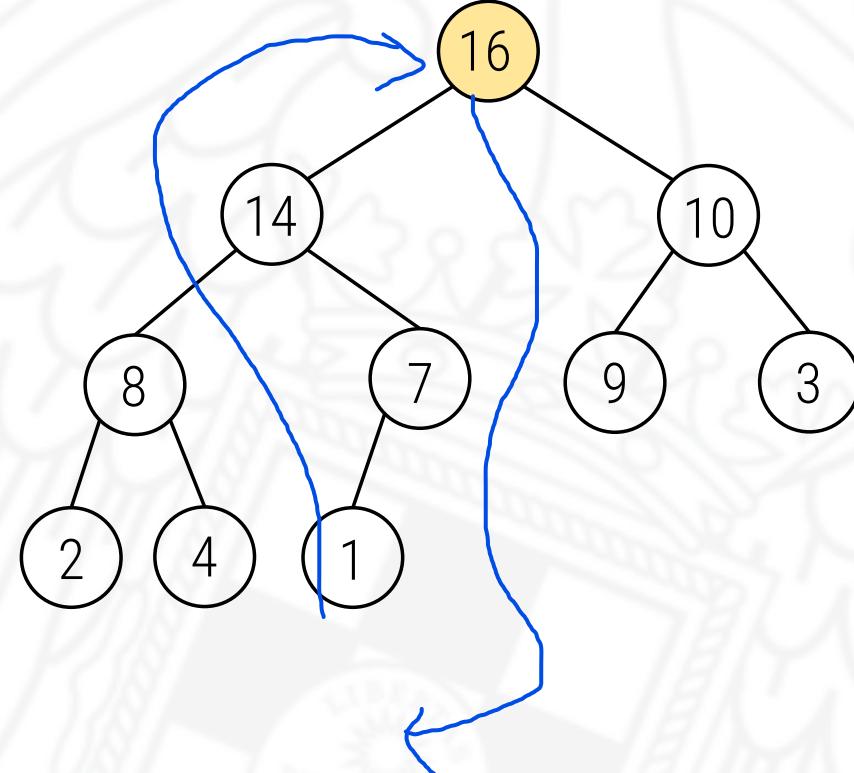
Ordenar



16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---

Vector que representa un montículo del tema anterior.

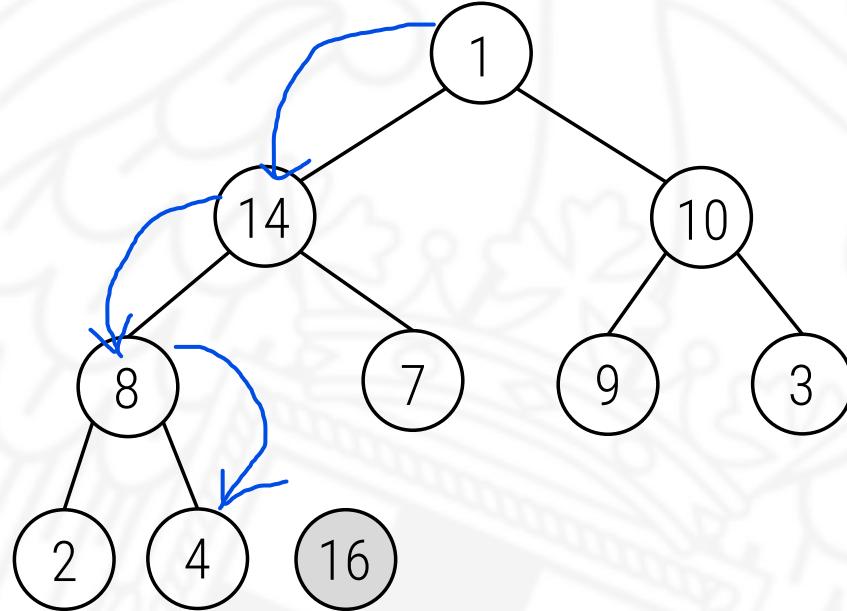
Ordenar



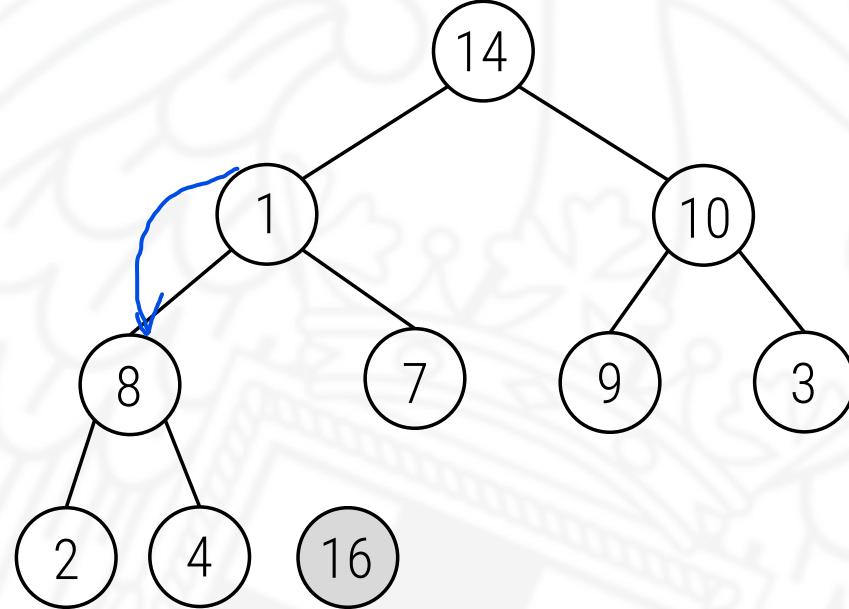
El de la raíz se intercambia con el último. Y el que estaba en la raíz ahora deja de estar en el árbol.

Ordenar

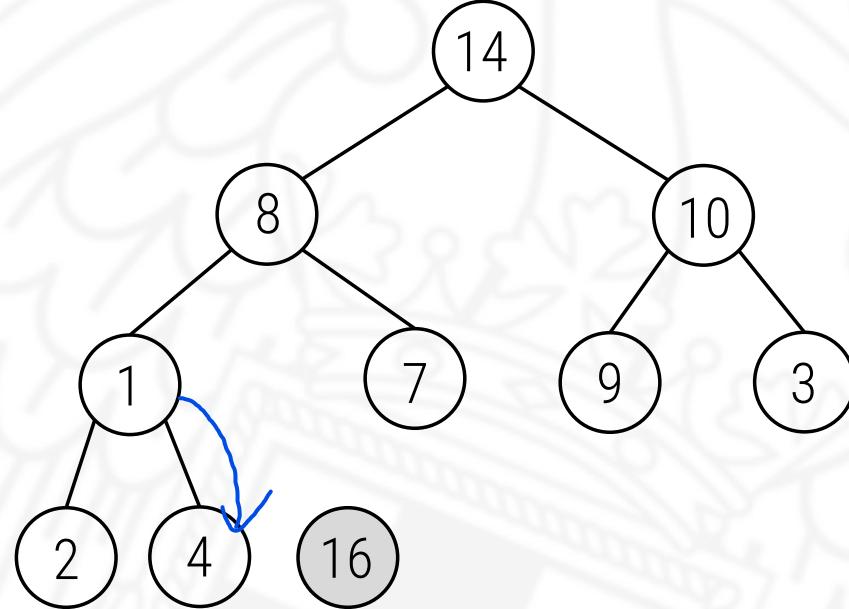
El uno ahora está en la raíz, pero el árbol NO está bien ordenado. Tenemos que hundir el 1. Para ello asciende el de mayor valor (14) y después volvemos a comparar. Volvería a ascender el de mayor valor (8) y después igual con el 4.



Ordenar



Ordenar

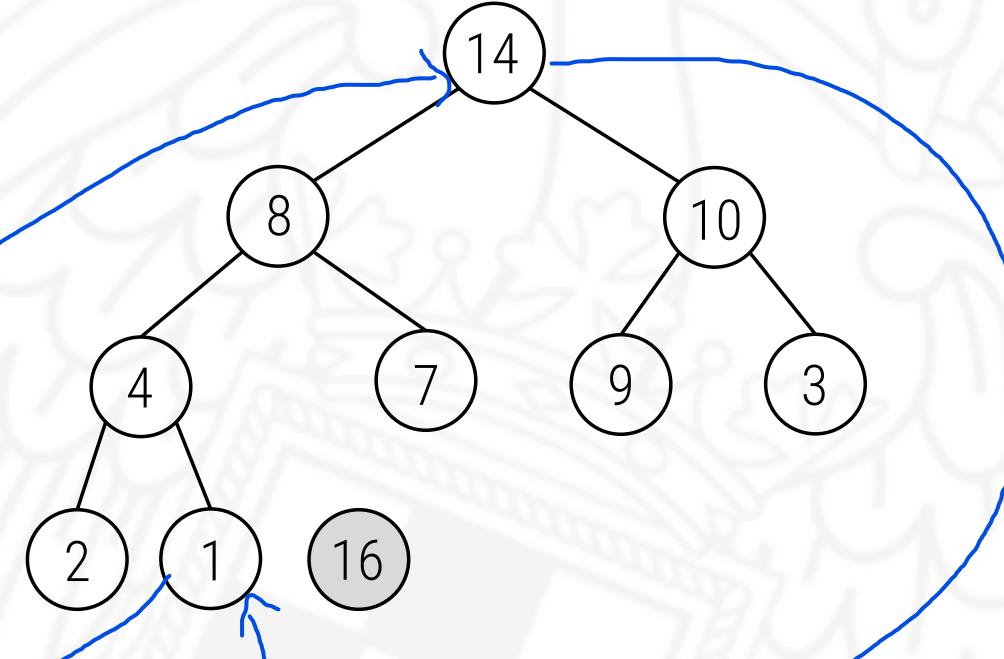


Ordenar

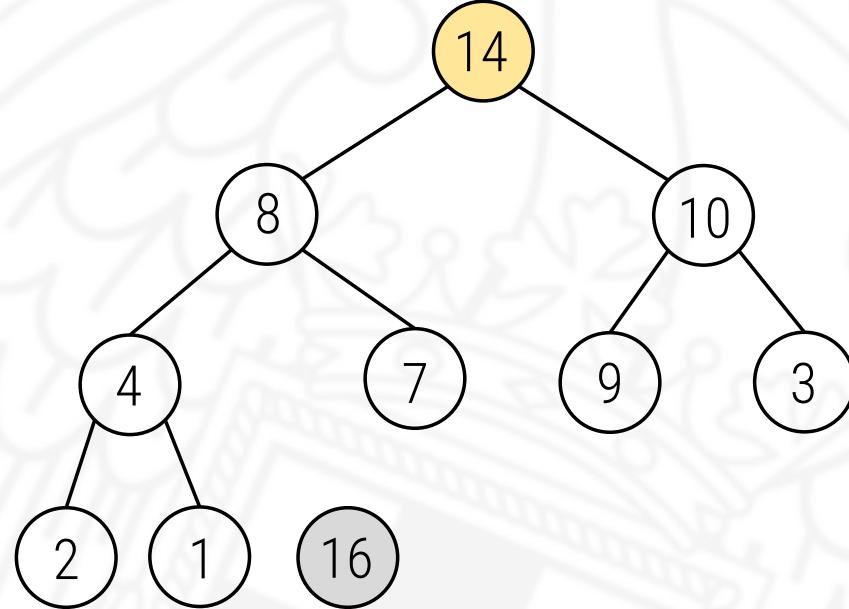
Ahora es el turno de quitar el 14 por el último.

Se pone en la raíz del árbol.

Sale del árbol para colocarse justo antes del 16.

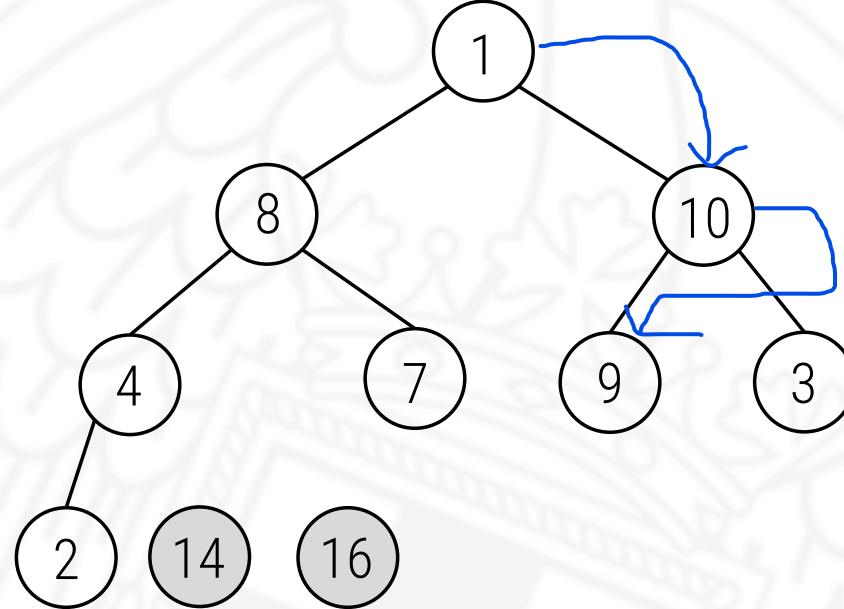


Ordenar

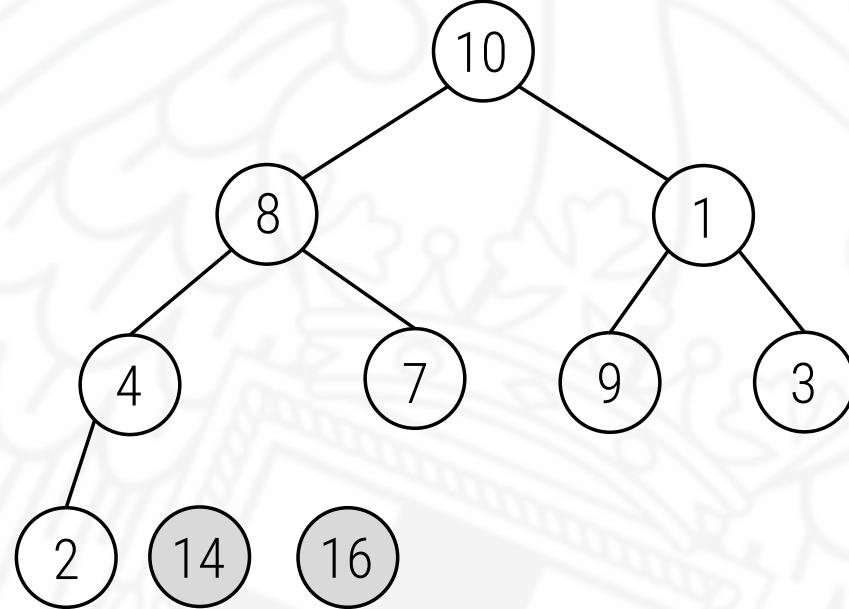


Ordenar

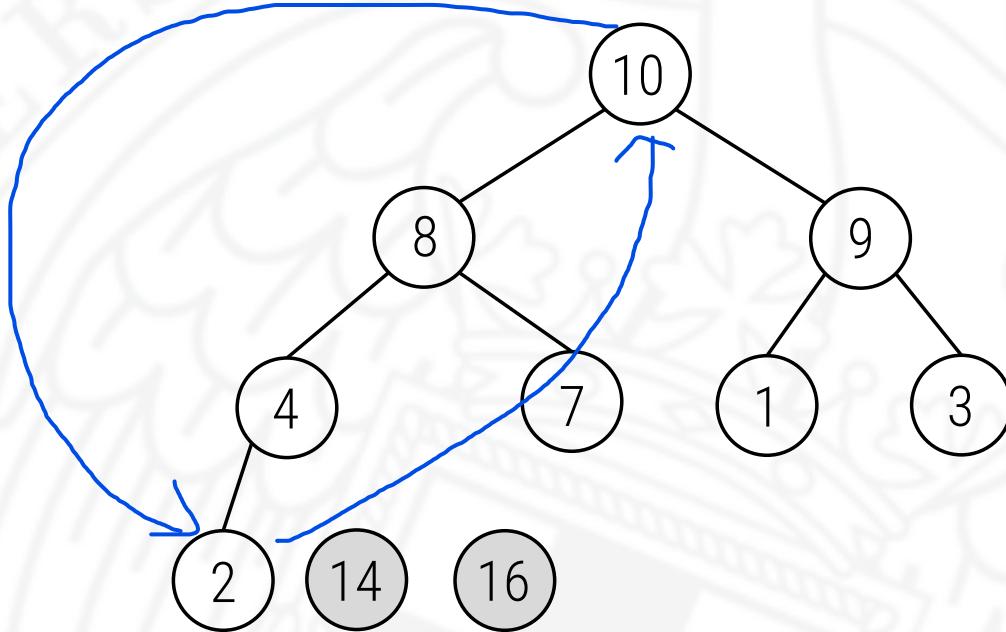
El 1 debe seguir el mismo proceso de hundirse anterior.



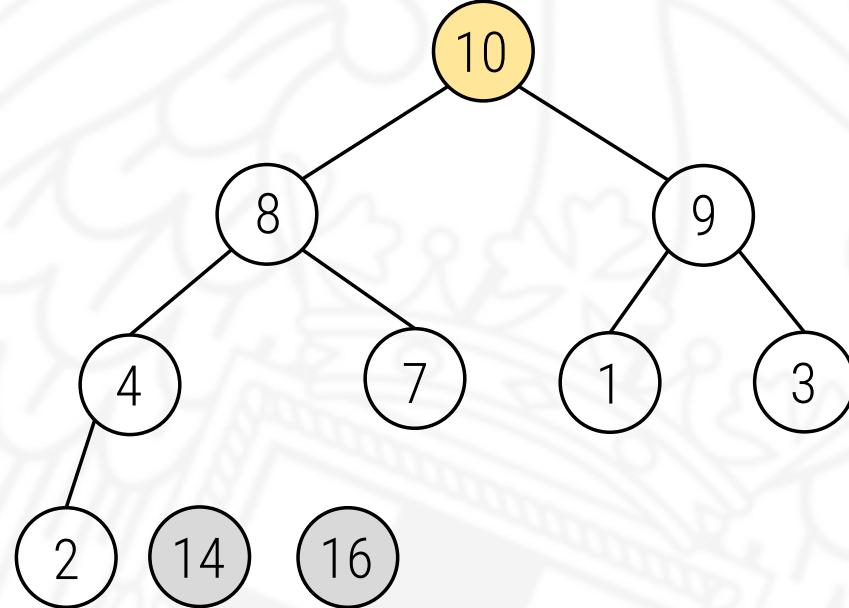
Ordenar



Ordenar

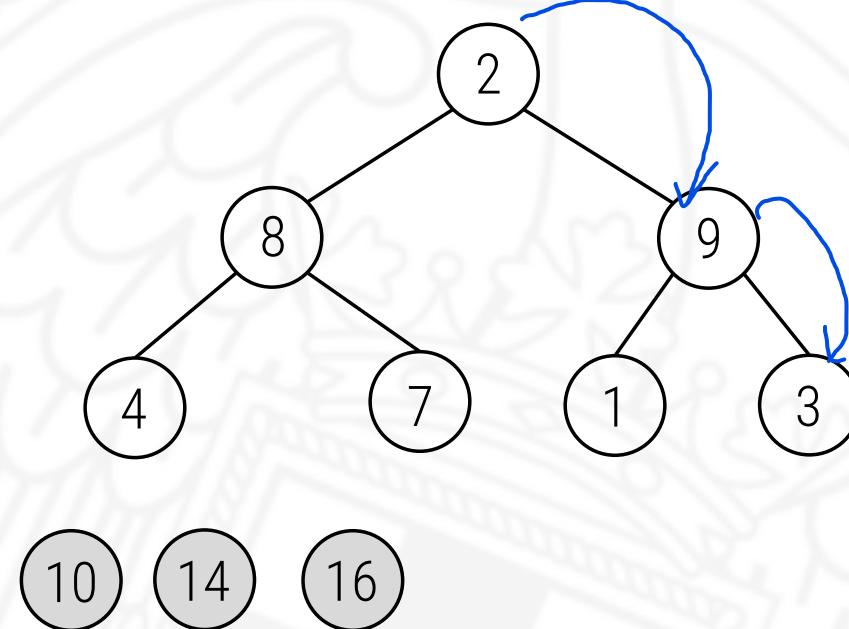


Ordenar

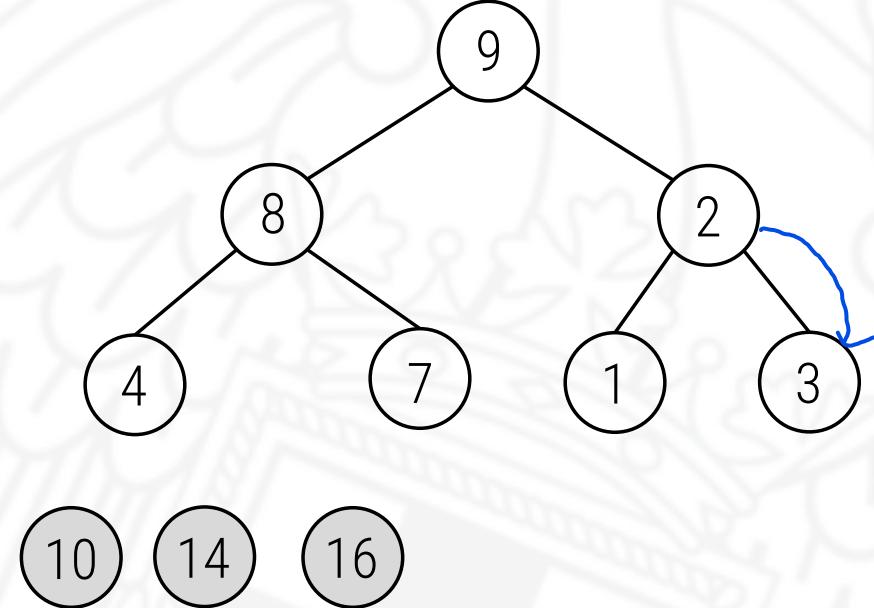


Ordenar

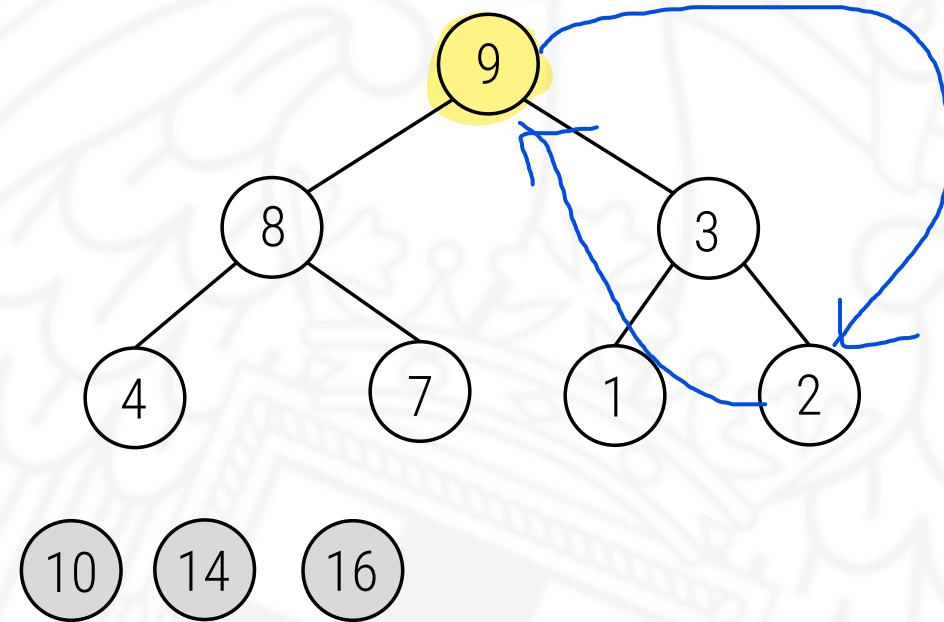
Mismo proceso de reordenamiento.



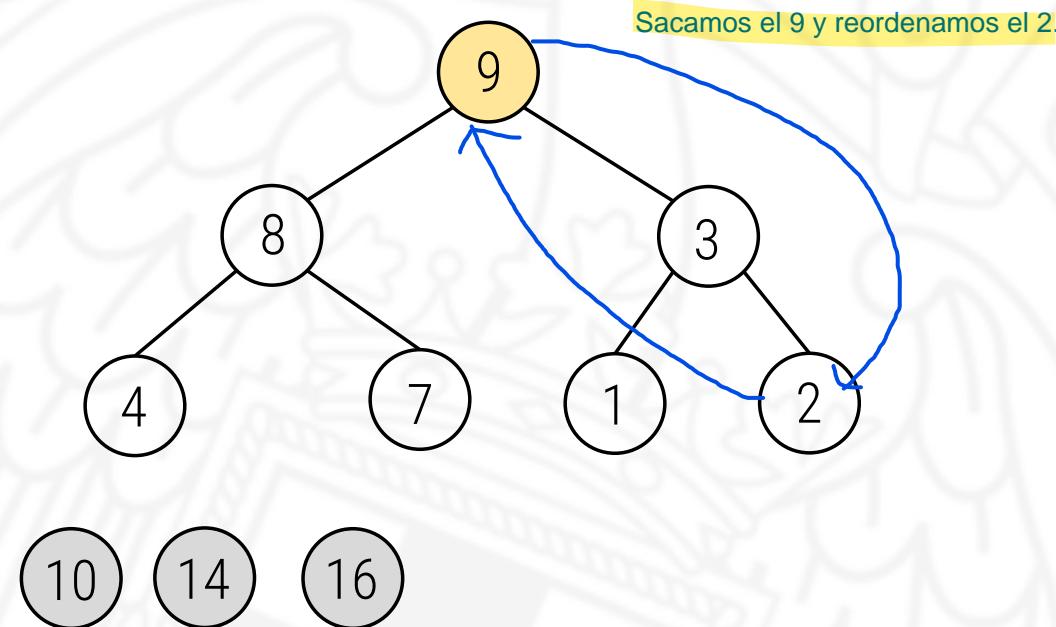
Ordenar



Ordenar

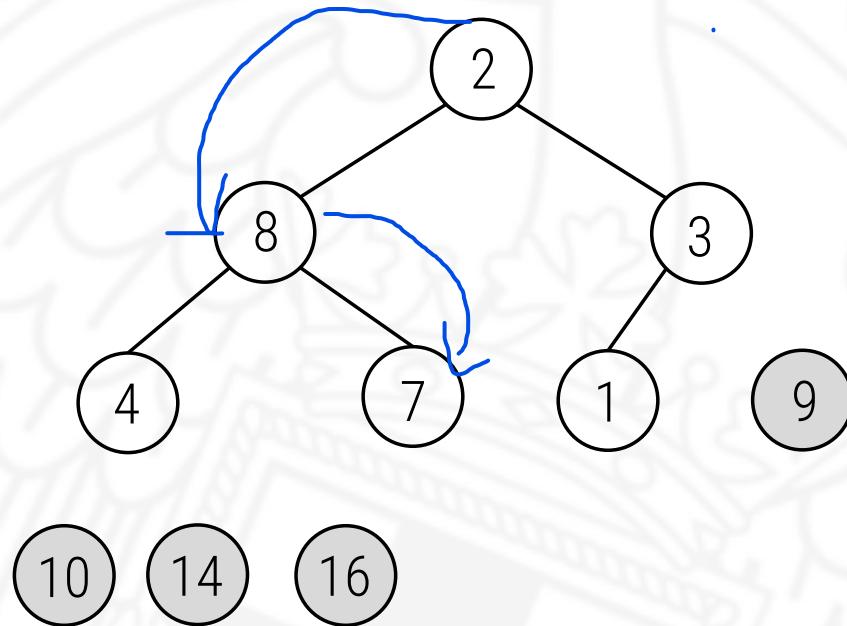


Ordenar

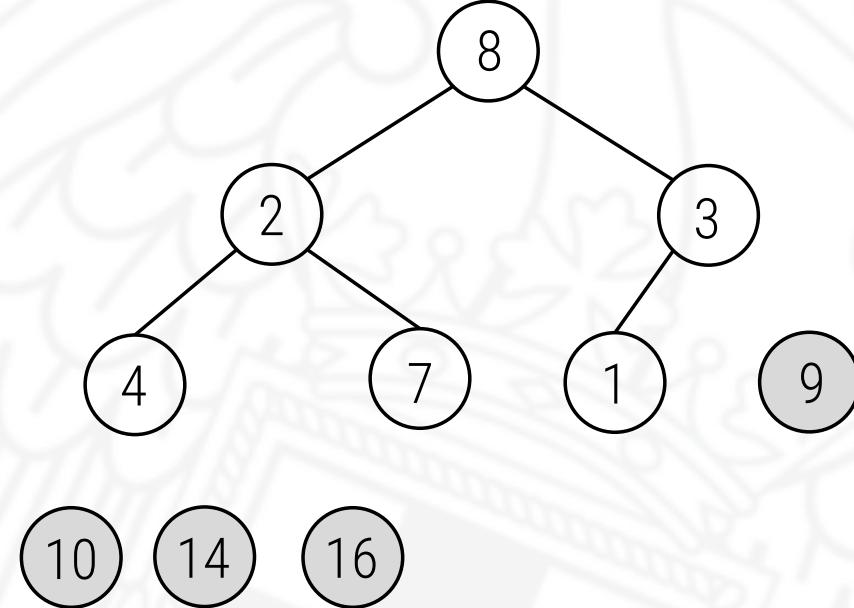


Ordenar

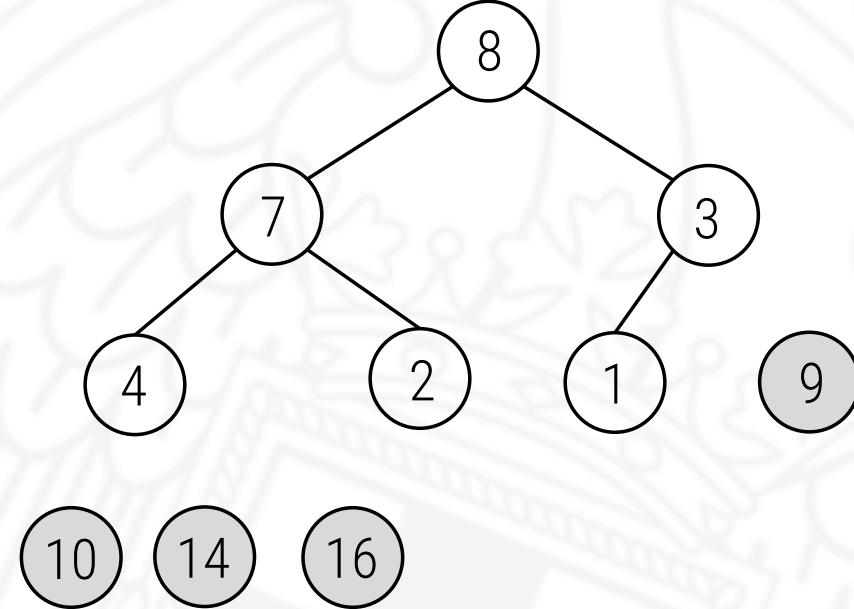
Como resultado debe quedar el montículo de mínimos si sacamos primero los máximos y viceversa.



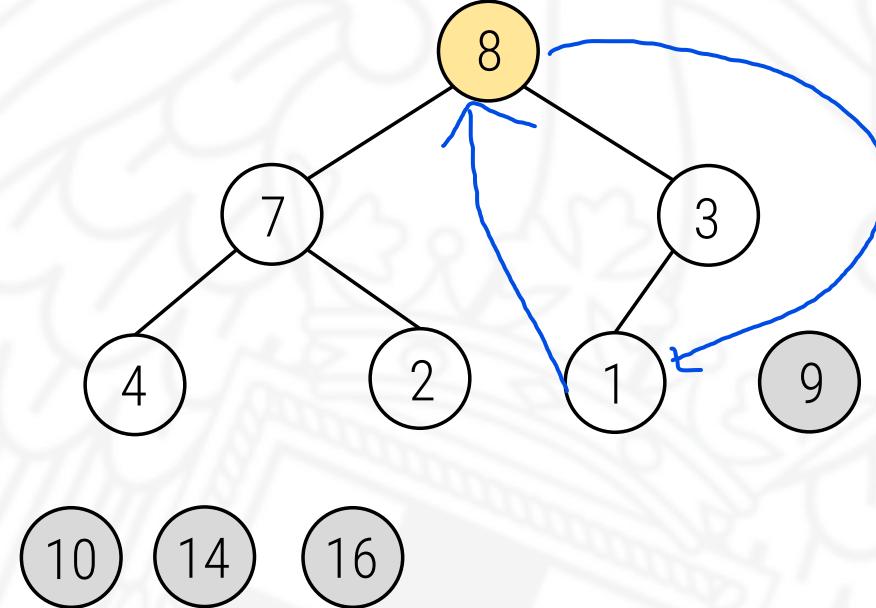
Ordenar



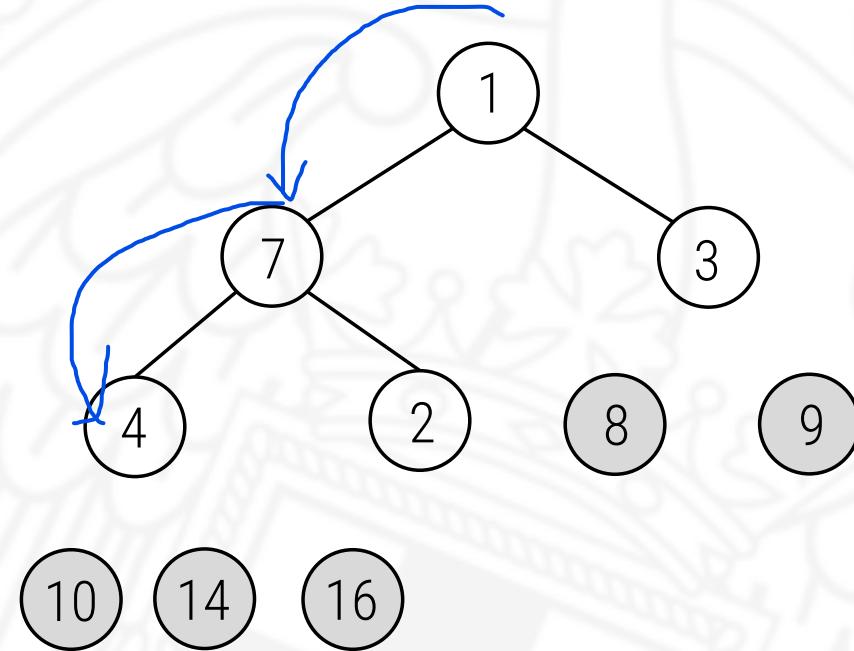
Ordenar



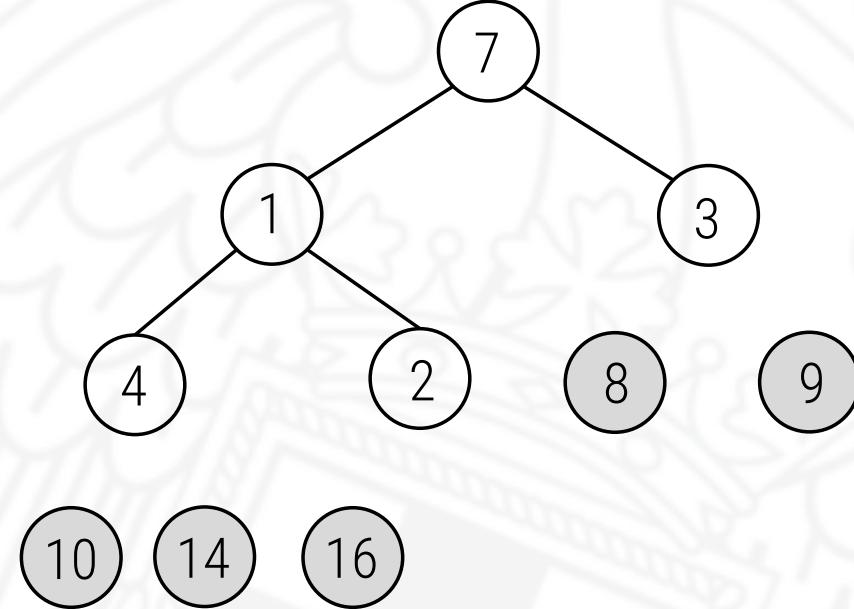
Ordenar



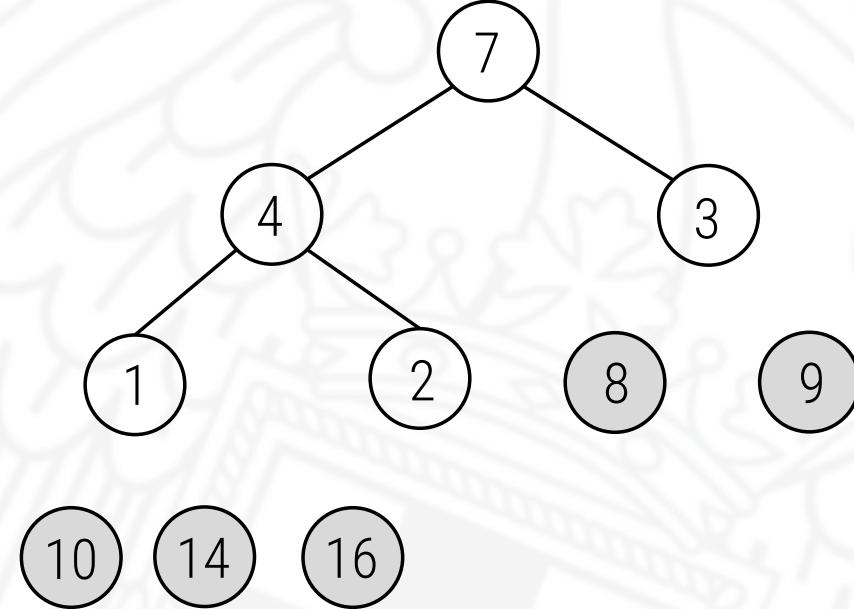
Ordenar



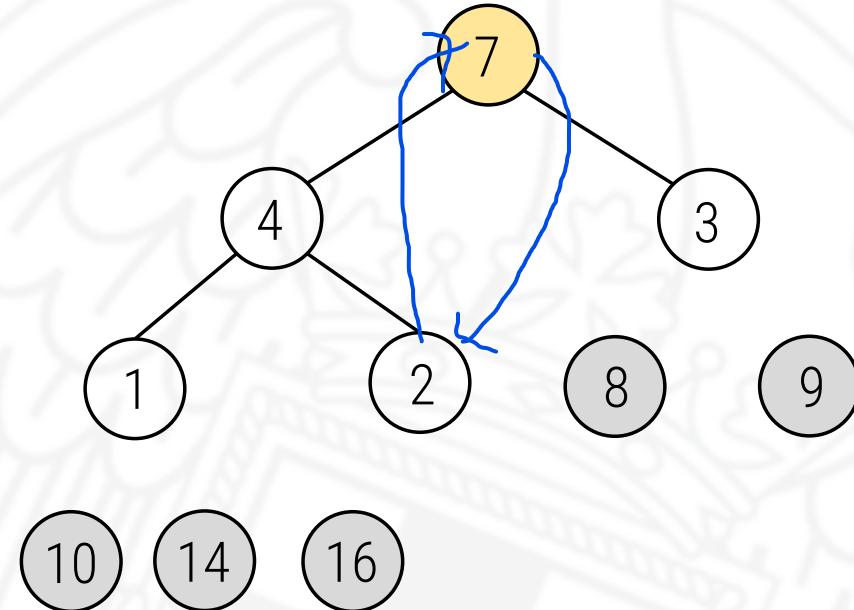
Ordenar



Ordenar

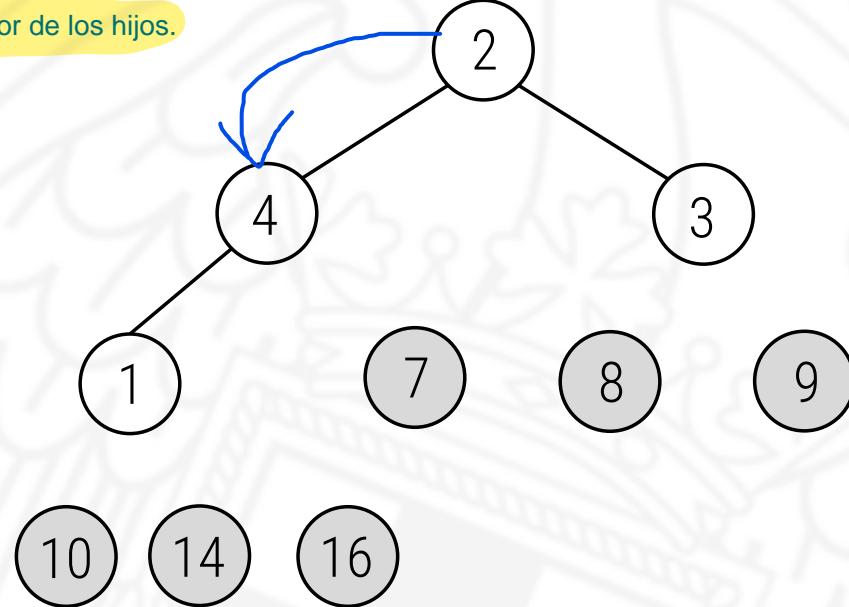


Ordenar

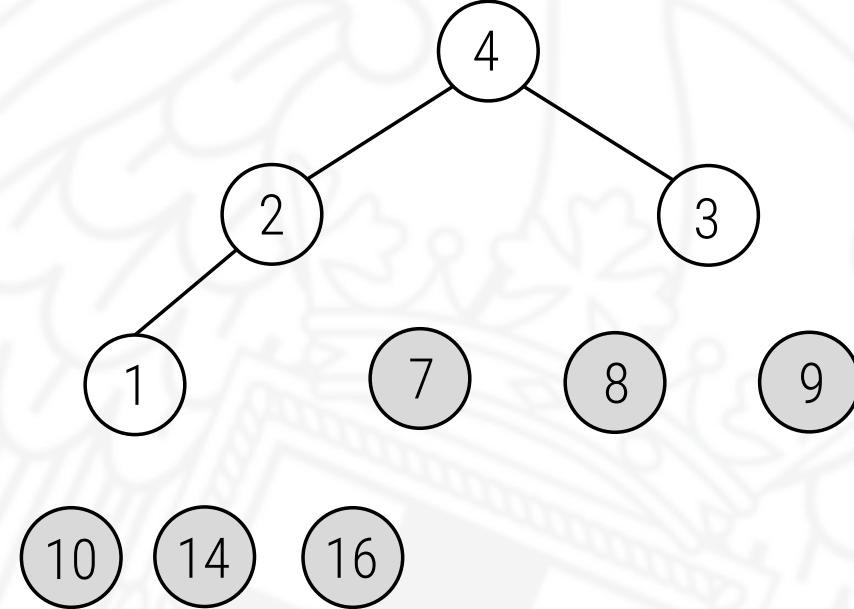


Ordenar

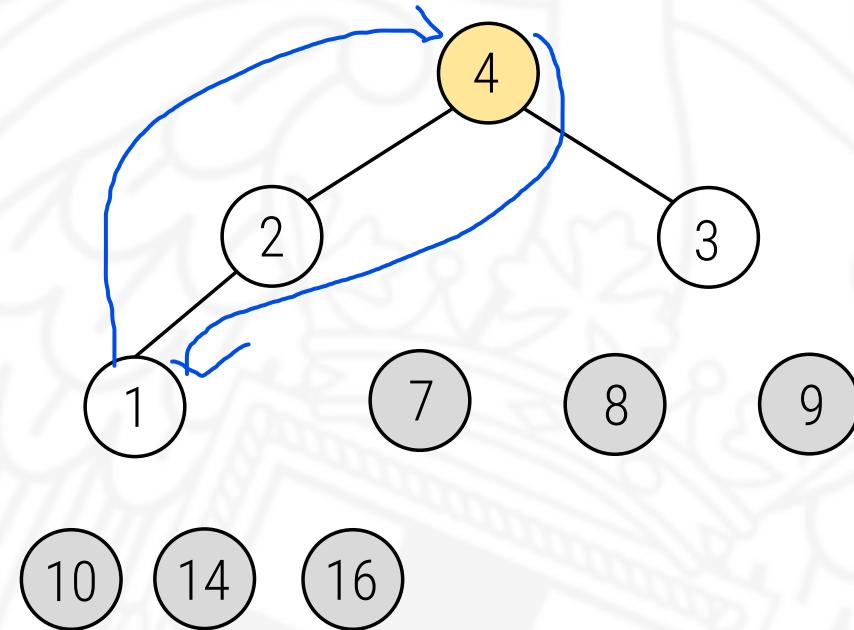
Sube el mayor de los hijos.



Ordenar

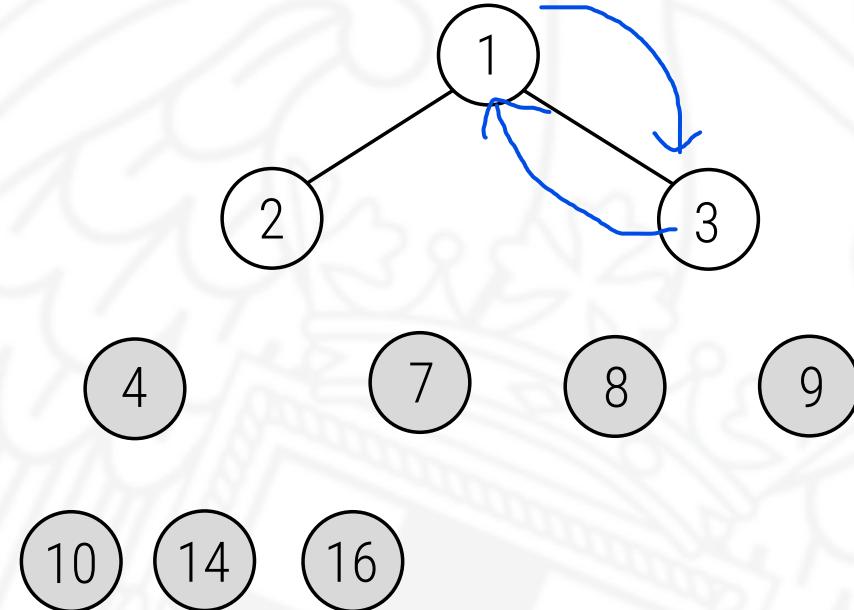


Ordenar

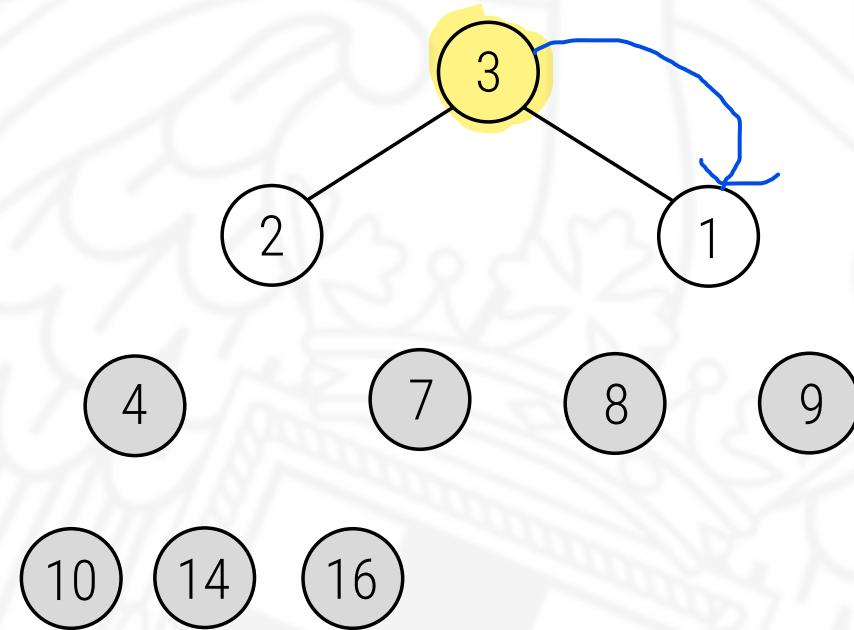


Ordenar

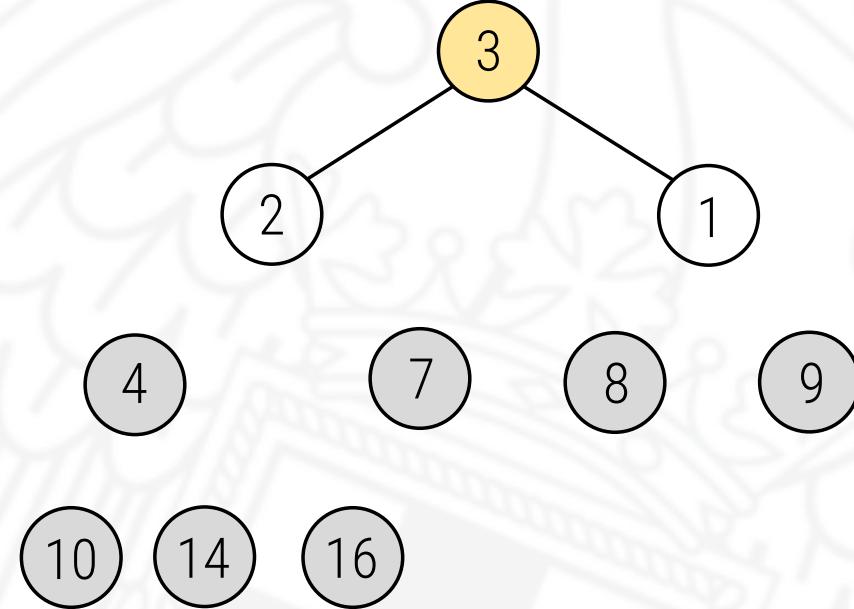
Comparamos para saber cual es el mayor de sus hijos.



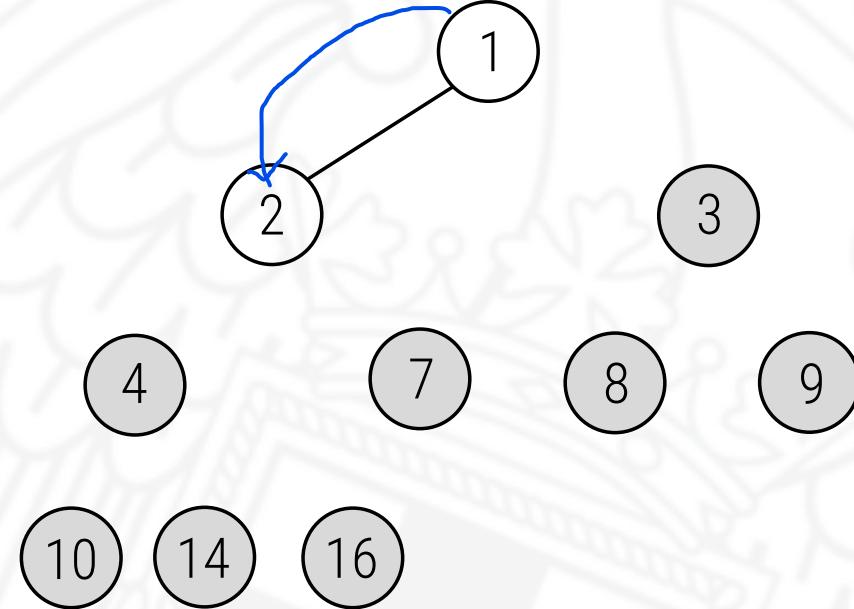
Ordenar



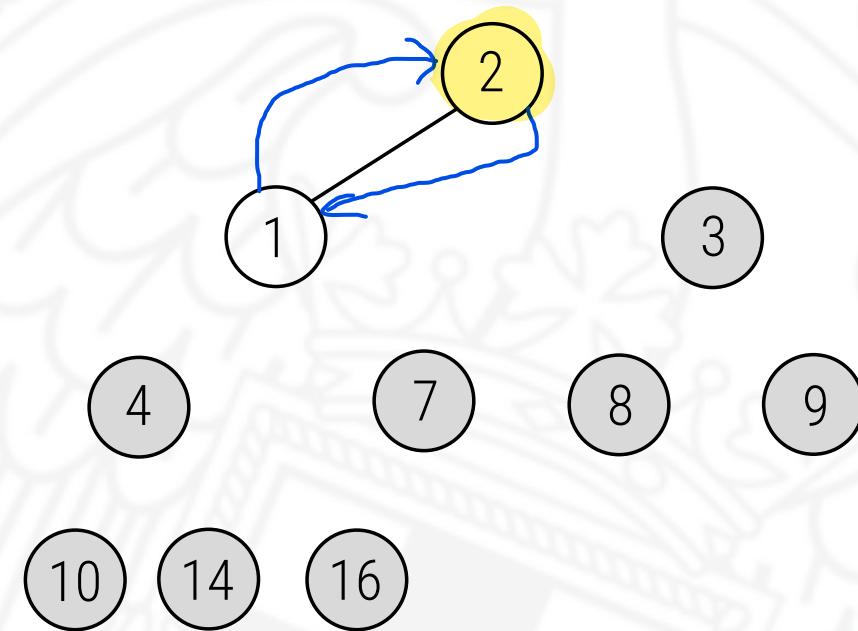
Ordenar



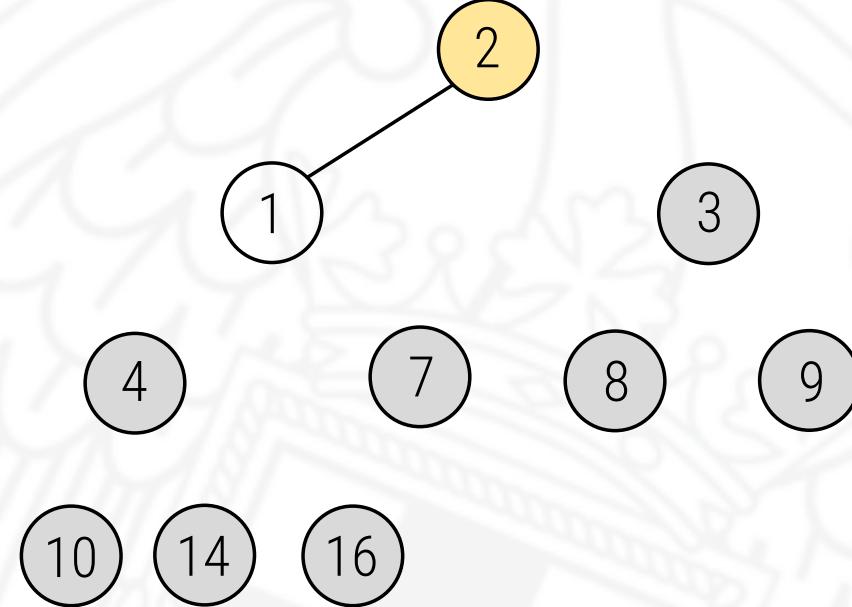
Ordenar



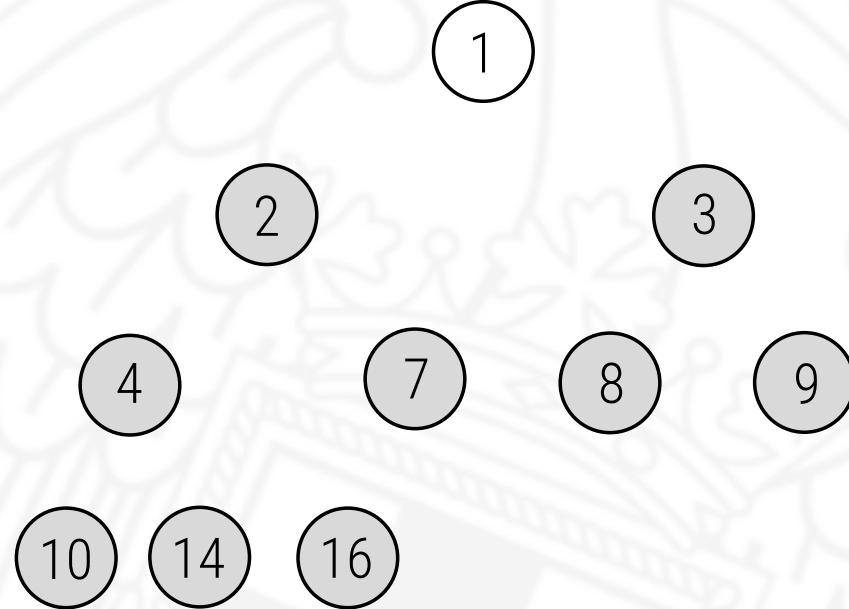
Ordenar



Ordenar

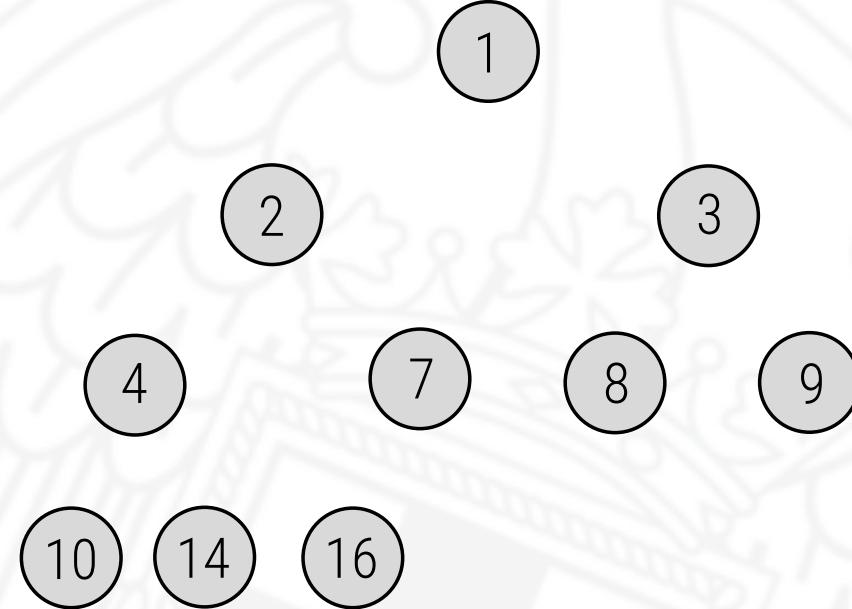


Ordenar



Montículo donde el top es el menor.

Ordenar



Ordenar



1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

Esta fase tiene un coste $O(N \log(N))$

Esto es porque hemos ido extrayendo el elemento más prioritario del montículo N veces. Por eso el coste es ese.

Implementación

```
template <typename T, typename Comparador = std::less<T>>
void heapsort(std::vector<T> & v, Comparador cmp = Comparador()) {
    // monticulizar
    for (int i = (v.size() - 1) / 2; i >= 0; --i)
        hundir_max(v, v.size(), i, cmp);
    // ordenar
    for (int i = v.size() - 1; i > 0; --i) {
        std::swap(v[i], v[0]);
        hundir_max(v, i, 0, cmp);
    }
}
```

Fase en la que el vector se convierte en un montículo. Recorriendo la primera mitad del vector de derecha a izquierda hundiendo cada uno de los elementos si hace falta.

Ahora llega hasta la posición i.

El que hundimos.

El que hundimos

Por si queremos ordenar de una manera distinta a de menor a mayor.

Cambiamos porque siempre tiene el máximo de los elementos del montículo con la posición i que se va decrementando.

Implementación

```
template <typename T, typename Comparador>
void hundir_max(std::vector<T> & v, int N, int j, Comparador cmp) {
    // montículo en v en posiciones de 0 a N-1
```

T elem = v[j];

Parecida a la función hundir que vimos en la implementación de las colas de prioridad.

int hueco = j;

int hijo = 2*hueco + 1; // hijo izquierdo, si existe

while (hijo < N) {

// cambiar al hijo derecho si existe y va antes que el izquierdo

if (hijo + 1 < N && cmp(v[hijo], v[hijo + 1]))
 ++hijo;

// flotar el hijo mayor si va antes que el elemento hundiéndose

if (cmp(elem, v[hijo])) {
 v[hueco] = v[hijo];
 hueco = hijo; hijo = 2*hueco + 1;

} else break;

}

v[hueco] = elem;

Recibe el vector y el tamaño del montículo.

Hundimos el elemento de la posición j.

Para calcular el hijo izquierdo necesitamos que sea $2 \cdot \text{hueco} + 1$ porque los elementos están colocados en el vector desde la posición 0. Por ejemplo: hijo izquierdo de la raíz está en la posición 1 (ver dibujo)

Estamos construyendo montículo de máximos es más prioritario el elemento mayor.



Coste de hundir es $\log(N)$ siendo N el número de nodos del montículo.

Ejemplo

```
vector<string> datos;
```

Vector desordenado.

datos	Zorro	leon	abeja	Lobo	perro	gato
-------	-------	------	-------	------	-------	------

```
heapsort(datos);
```

si a ese vector le aplicamos heapsort tenemos esto.

datos	Lobo	Zorro	abeja	gato	leon	perro
-------	------	-------	-------	------	------	-------

Puede parecer que NO está ordenado ya que zorro está delante de abeja. Pero esto es porque estamos aplicando el operador MENOR en vez de mayor. Hace comparación caracter y caracter teniendo en cuenta el código ASCII de los caracteres. Y la Z es mayúscula y la L también lo es.

Ejemplo

```
string a_minusculas(string s) {
    for (char & c : s) c = tolower(c);
    return s;
}

class ComparaString {
public:
    bool operator()(string a, string b) {
        return a_minusculas(a) < a_minusculas(b);
    }
};

heapsort(datos, ComparaString());
```

Operador de cadenas de caracteres y pasamos las palabras a minúsculas antes de compararlas.

A la función tenemos que pasarle un objeto de la clase ComparaString que construimos usando la constructora por defecto.

datos

abeja	gato	leon	Lobo	perro	Zorro
-------	------	------	------	-------	-------

Ejemplo mejorado

```
class ComparaString { public:
    bool operator()(std::string const& a, std::string const& b) {
        int i = 0;
        while (i != a.length()) {
            if (i == b.length() || tolower(b[i]) < tolower(a[i])) return false;
            else if (tolower(a[i]) < tolower(b[i])) return true;
            ++i;
        }
        return i != b.length();
    }
};

heapsort(datos, ComparaString());
```

Comparación de los caracteres. Antes de ordenarlos los pasamos a minúsculas.

datos	abeja	gato	leon	Lobo	perro	Zorro
-------	-------	------	------	------	-------	-------