

ANÁLISIS Y DISEÑO DE ALGORITMOS I Periodo I – 2023

Jesús Aranda

Universidad del Valle
Escuela de Ingeniería de Sistemas y Computación

Este documento es una adaptación del material original del profesor Oscar Bedoya



Comparación de algoritmos de ordenamiento

Propiedades de orden y forma de los montones

Operaciones HEAPIFY, BUILD-HEAP Y HEAP-SORT

Análisis de complejidades

Colas de prioridad

Heapsort

Problema de ordenamiento

- Insertion

 - $O(n^2)$

 - Ordena in-place

- Merge

 - $\Theta(n \lg n)$

 - No ordena in-place

- Heapsort

 - $O(n \lg n)$

 - Ordena in-place

- Quicksort

 - $\Theta(n^2)$, Caso promedio: $\Theta(n \lg n)$

 - Ordena in-place

Heapsort

Insertion, Merge, Heapsort y Quicksort son algoritmos de ordenamiento basados en comparación. Estos tienen la característica de que son del orden de $\Omega(n \lg n)$

¿Es posible bajar esta cota?

Heapsort

Insertion, Merge, Heapsort y Quicksort son algoritmos de ordenamiento basados en comparación. Estos tienen la característica de que son del orden de $\Omega(n \lg n)$

¿Es posible bajar esta cota?

- Counting sort
- Radix sort
- Bucket sort

Heapsort

Heapsort

Idea: utilizar las fortalezas de MergeSort y de InsertionSort

Utiliza un representación lógica, conocida como montículo (*heap*), de un arreglo que permite ordenar los datos del arreglo in-place

Heapsort

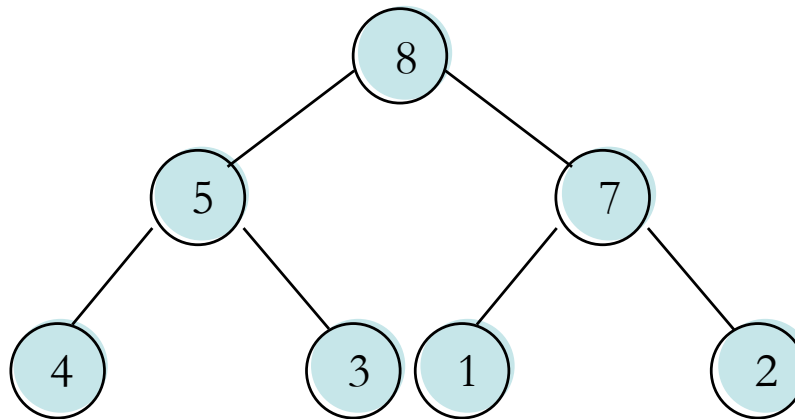
Montículos

Es un arreglo que puede ser visto como un árbol binario que cumple dos propiedades:

- *Propiedad del orden*: La raíz de cada subarbol es mayor o igual que cualquier de sus nodos restantes
- *Propiedad de forma*: La longitud de toda rama es h o $h-1$, donde h es la altura del árbol. Además, no puede existir un rama de longitud h a la derecha de una rama de longitud $h-1$

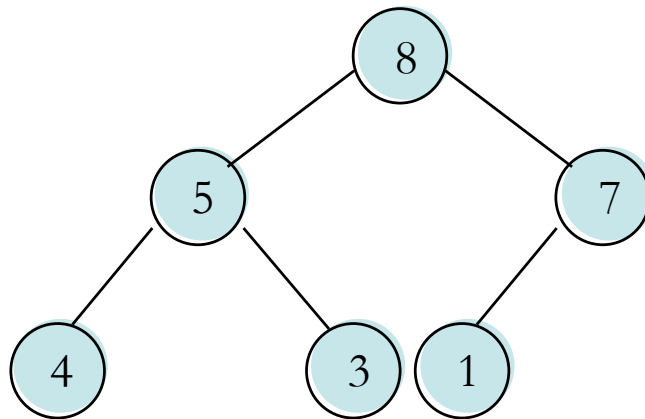
Heapsort

Analizar las propiedades de orden y de forma



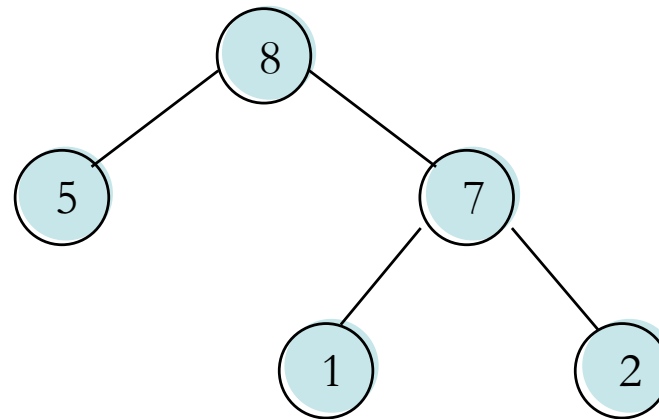
Heapsort

Analizar las propiedades de orden y de forma



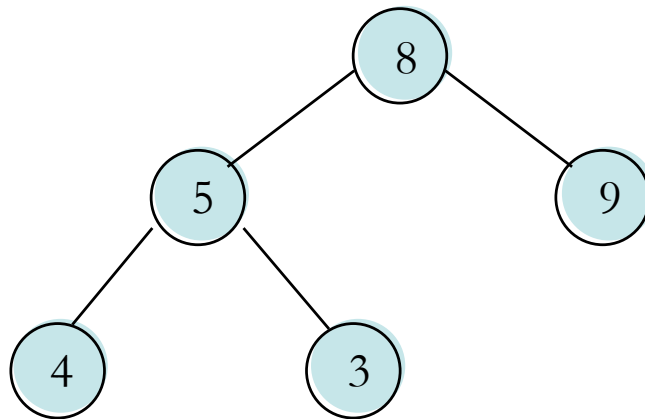
Heapsort

Analizar las propiedades de orden y de forma



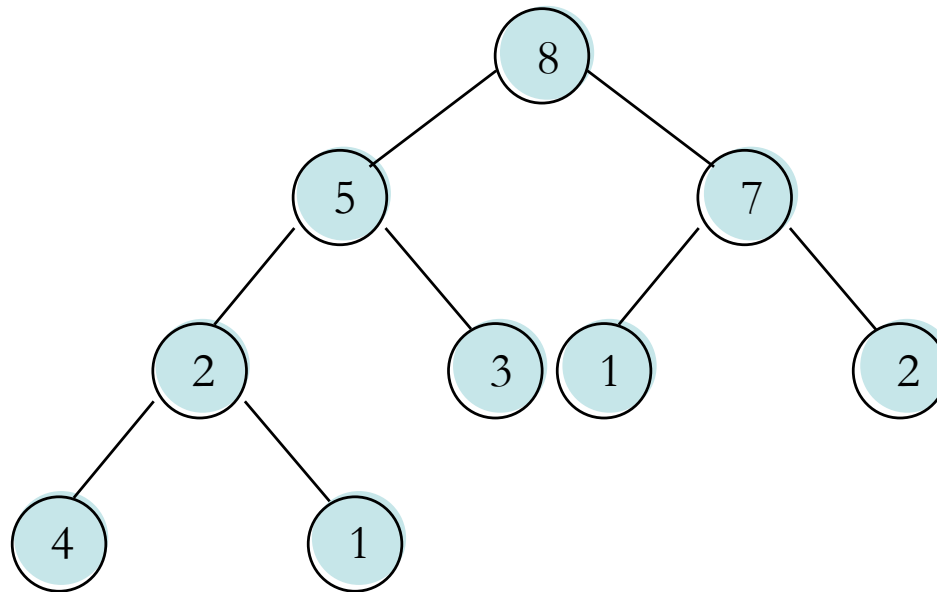
Heapsort

Analizar las propiedades de orden y de forma



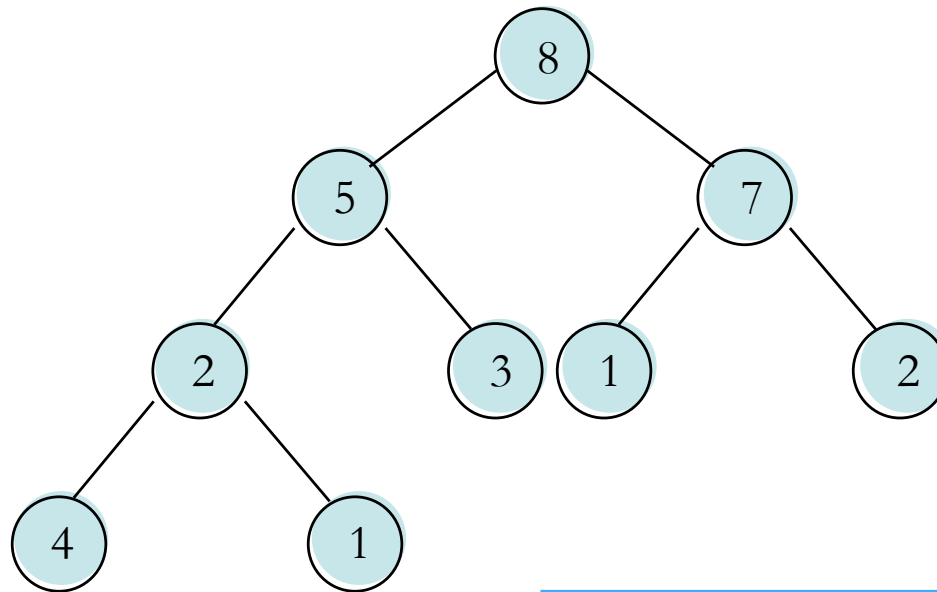
Heapsort

Analizar las propiedades de orden y de forma



Heapsort

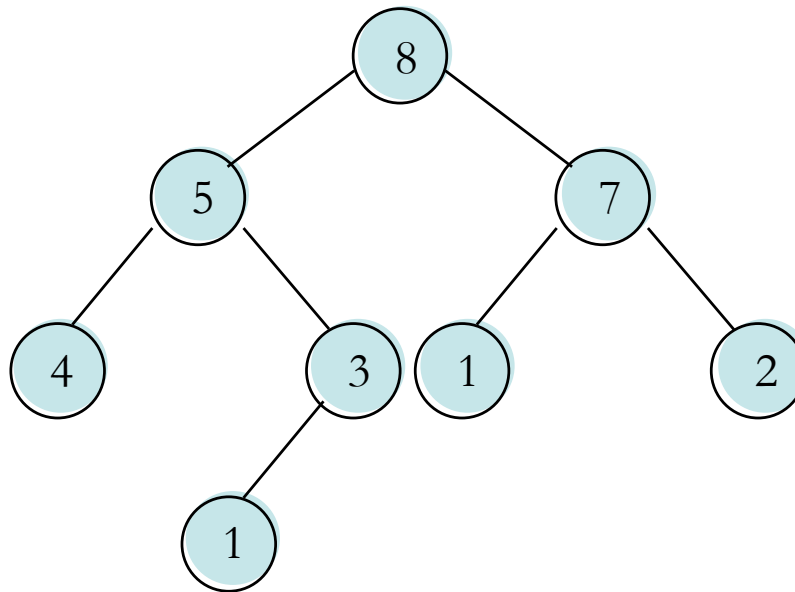
Analizar las propiedades de orden y de forma



Falla la propiedad de orden, en el subarbol 4-2-1, la raíz no cumple con ser mayor o igual los demás elementos

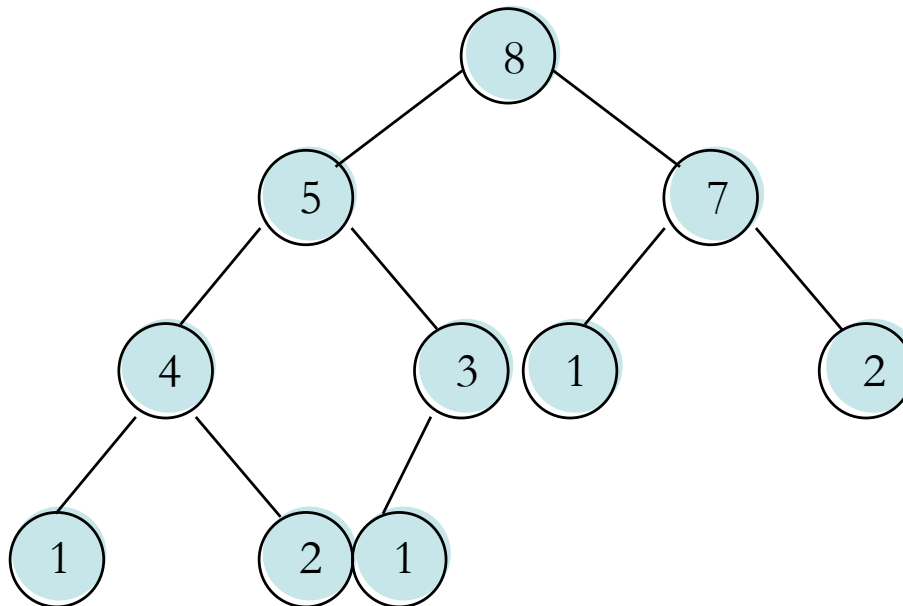
Heapsort

Indique si se cumplen las propiedades de orden y de forma



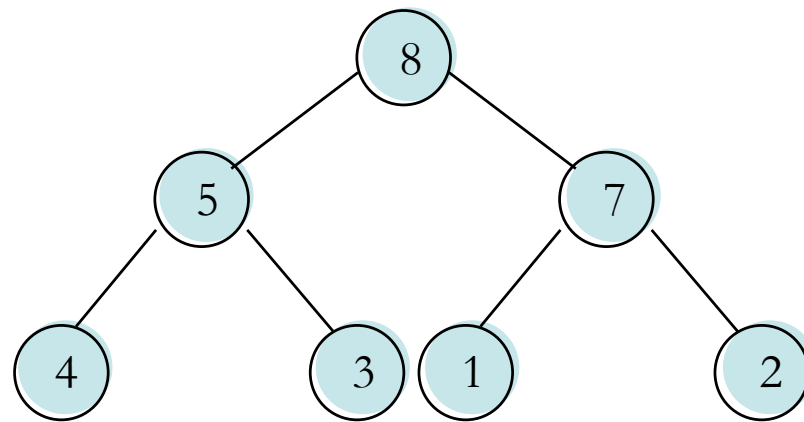
Heapsort

Indique si se cumplen las propiedades de orden y de forma



Heapsort

Los datos se almacenan en el arreglo recorriendo, por niveles, de izquierda a derecha

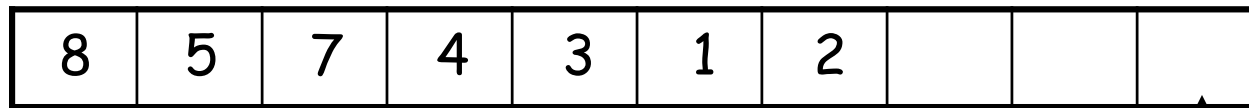
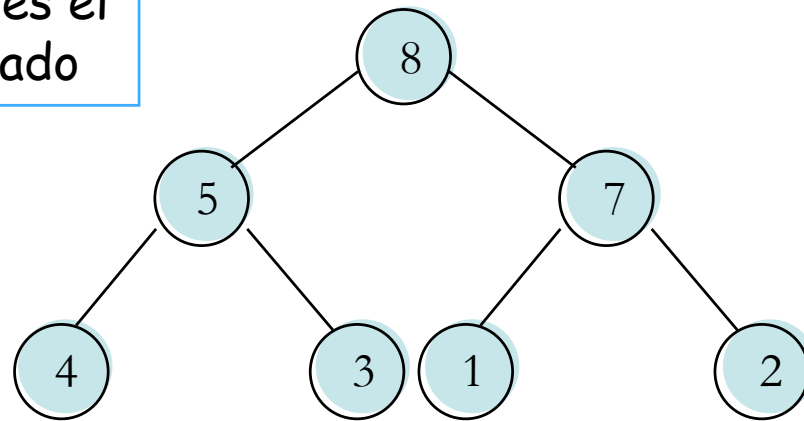


8	5	7	4	3	1	2			
---	---	---	---	---	---	---	--	--	--

Heapsort

Los datos se almacenan en el arreglo recorriendo, por niveles, de izquierda a derecha

$A[1, \dots, \text{heap-size}[A]]$ es el montículo representado



↑
 $\text{heap-size}[A]=7$ $\text{length}[A]=10$

Heapsort

Indique si se cumplen las propiedades de orden y de forma

$A = \{20, 10, 5, 4, 3, 1\}$ donde $\text{heap-size}[A] = 6$ y $\text{length}[A] = 10$

Heapsort

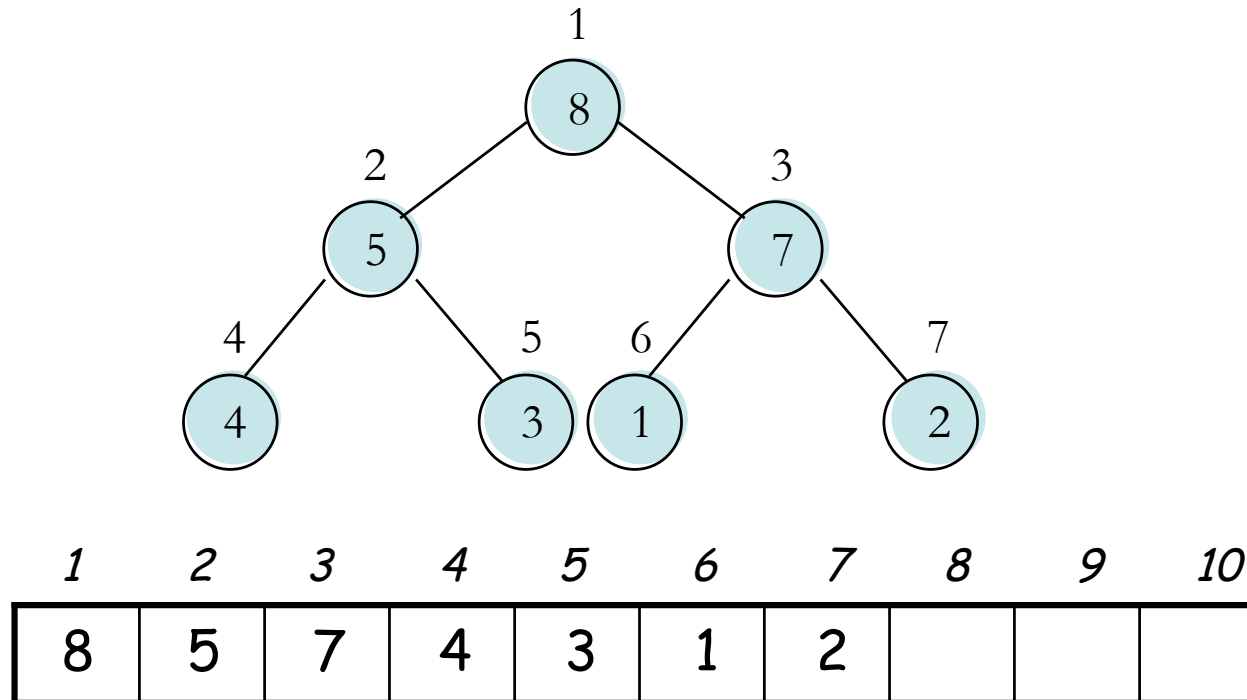
Indique si se cumplen las propiedades de orden y de forma

$A = \{20, 10, 5, 4, 3, 1, \}$ donde $\text{heap-size}[A] = 6$ y $\text{length}[A] = 10$

$A = \{8, 4, 2, 1, 7, 9 \}$ donde $\text{heap-size}[A] = 4$ y $\text{length}[A] = 10$

Heapsort

Los datos se almacenan en el arreglo recorriendo, por niveles, de izquierda a derecha



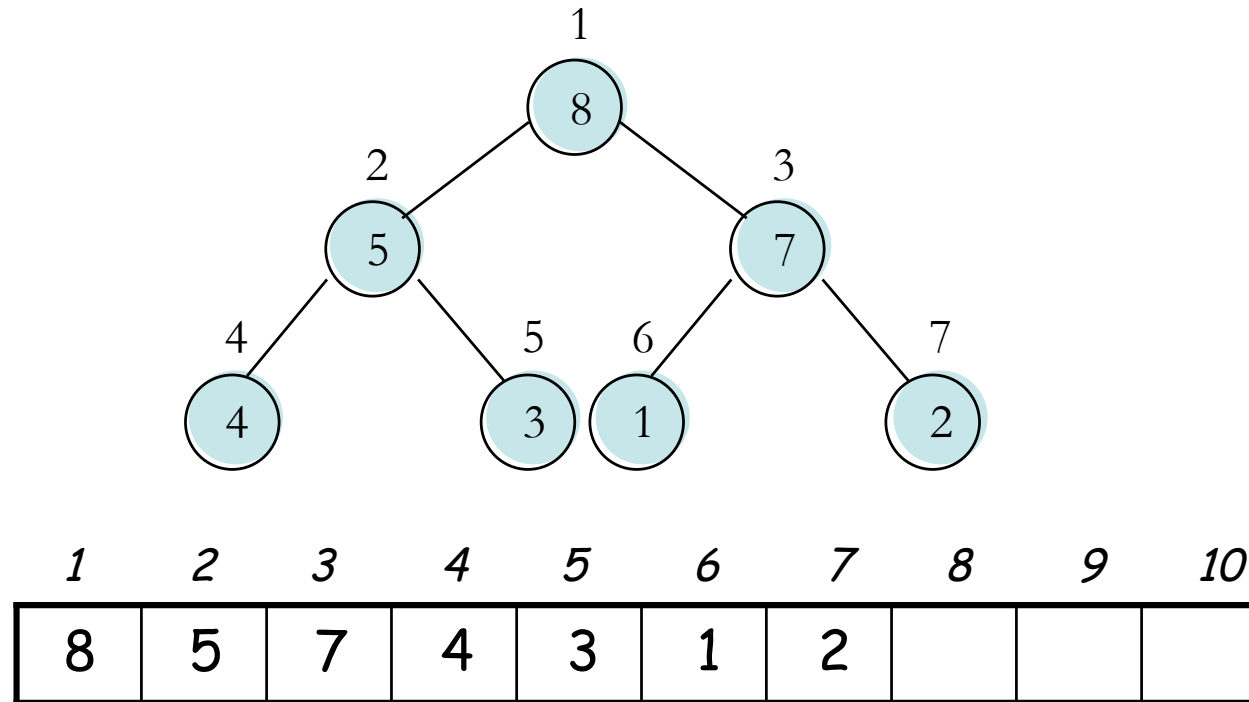
Evalue $\lfloor i/2 \rfloor$ para $i=2$ y 3

Evalue $\lfloor i/2 \rfloor$ para $i=4$ y 5

Evalue $\lfloor i/2 \rfloor$ para $i=6$ y 7

Heapsort

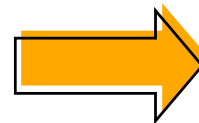
Los datos se almacenan en el arreglo recorriendo, por niveles, de izquierda a derecha



Evalue $\lfloor i/2 \rfloor$ para $i=2$ y 3

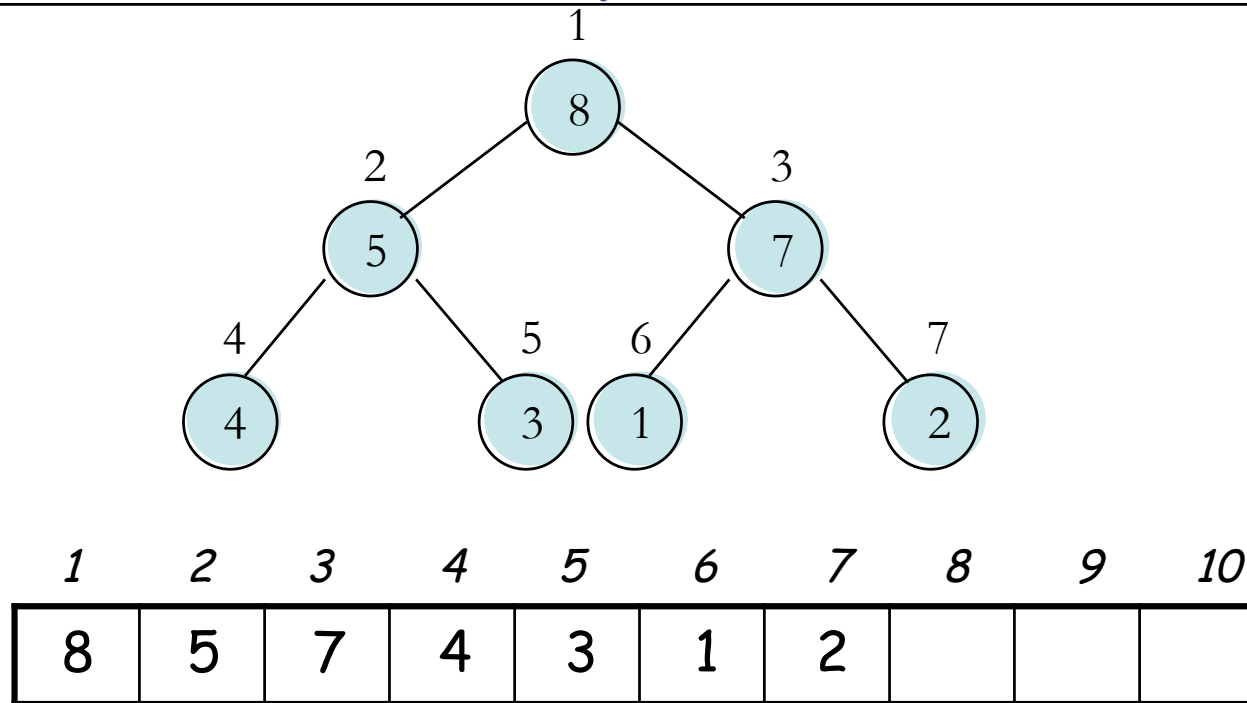
Evalue $\lfloor i/2 \rfloor$ para $i=4$ y 5

Evalue $\lfloor i/2 \rfloor$ para $i=6$ y 7



Padre(i): $\lfloor i/2 \rfloor$

Heapsort



Raíz del árbol: $A[1]$

Padre(i): $\lfloor i/2 \rfloor$

Izq(i): $A[2*i]$

Der(i): $A[2*i + 1]$

Heapsort

Operaciones con montículos:

- **Heapify**: $O(\lg n)$
- **Build-Heap**: $O(n)$
- **HeapSort**: $O(n \lg n)$
- Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key, Heap-Maximum: $O(\lg n)$ Colas de prioridad

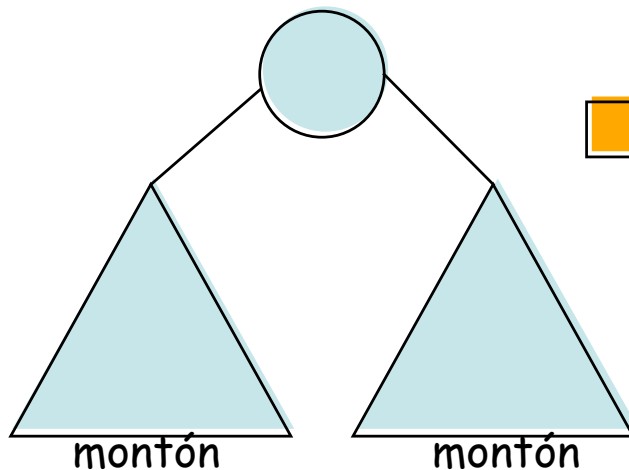
La altura de un montículo de n elementos es $\Theta(\lg n)$

Heapsort

Heapify

Precondición: subarbol con raíz Izq(i) y subarbol con raíz Der(i) son montículos

Poscondición: subárbol con raíz es un montículo



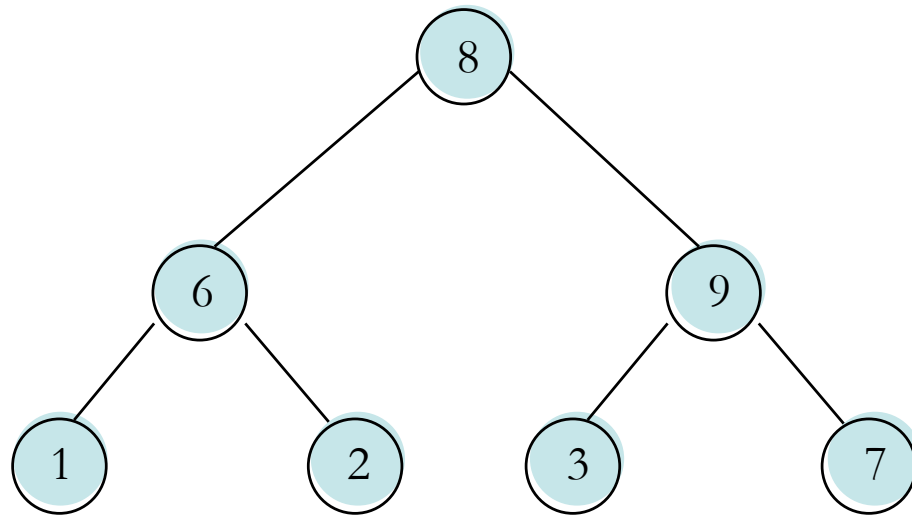
No es necesariamente un montón, se puede violar la propiedad de orden

Heapsort

Heapify

Precondición: subarbol con raíz Izq(i) y subarbol con raíz Der(i) son montículos

Poscondición: subárbol con raíz es un montículo



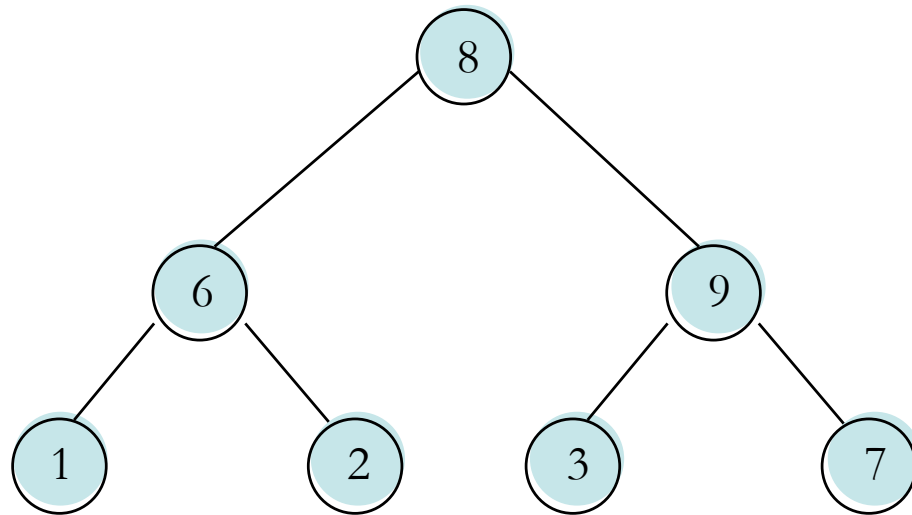
¿Cómo sería el montón resultante?

Heapsort

Heapify

Precondición: subarbol con raíz Izq(i) y subarbol con raíz Der(i) son montículos

Poscondición: subárbol con raíz es un montículo



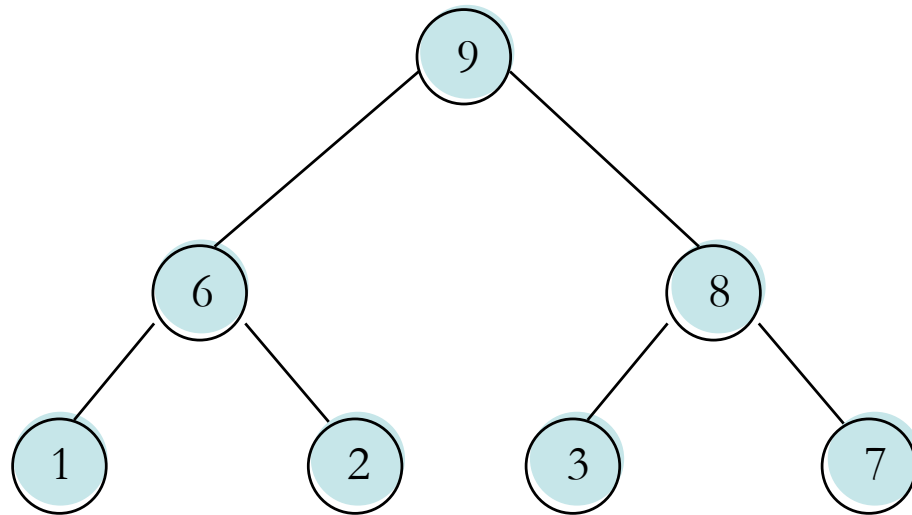
Se debe conocer cuál es el mayor entre la raíz Izq(i), la raíz Der(i) e $A[i]$

Heapsort

Heapify

Precondición: subarbol con raíz Izq(i) y subarbol con raíz Der(i) son montículos

Poscondición: subárbol con raíz es un montículo



Al hacer el cambio de valores se debe verificar que el montón 3-8-7 cumpla la propiedad de orden

HEAPIFY(A, i)

$l \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$

then $\text{largest} \leftarrow l$

else $\text{largest} \leftarrow i$

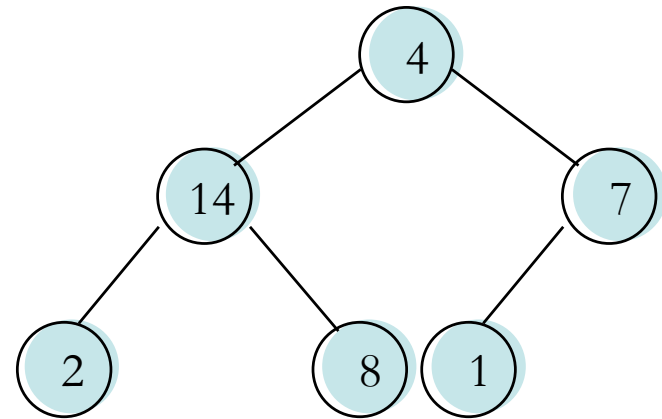
if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$

then $\text{largest} \leftarrow r$

if $\text{largest} \neq i$

then exchange $A[i] \leftrightarrow A[\text{largest}]$

HEAPIFY(A, largest)



Aplique el algoritmo
HEAPIFY(A, 1)

HEAPIFY(A, i)

$l \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$

then $\text{largest} \leftarrow l$

else $\text{largest} \leftarrow i$

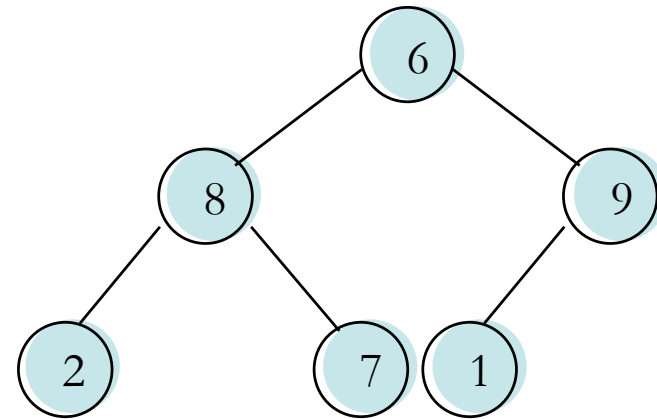
if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$

then $\text{largest} \leftarrow r$

if $\text{largest} \neq i$

then exchange $A[i] \leftrightarrow A[\text{largest}]$

HEAPIFY(A, largest)



Aplique el algoritmo
HEAPIFY(A, 1)

HEAPIFY(A, i)

$l \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$

then $\text{largest} \leftarrow l$

else $\text{largest} \leftarrow i$

if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$

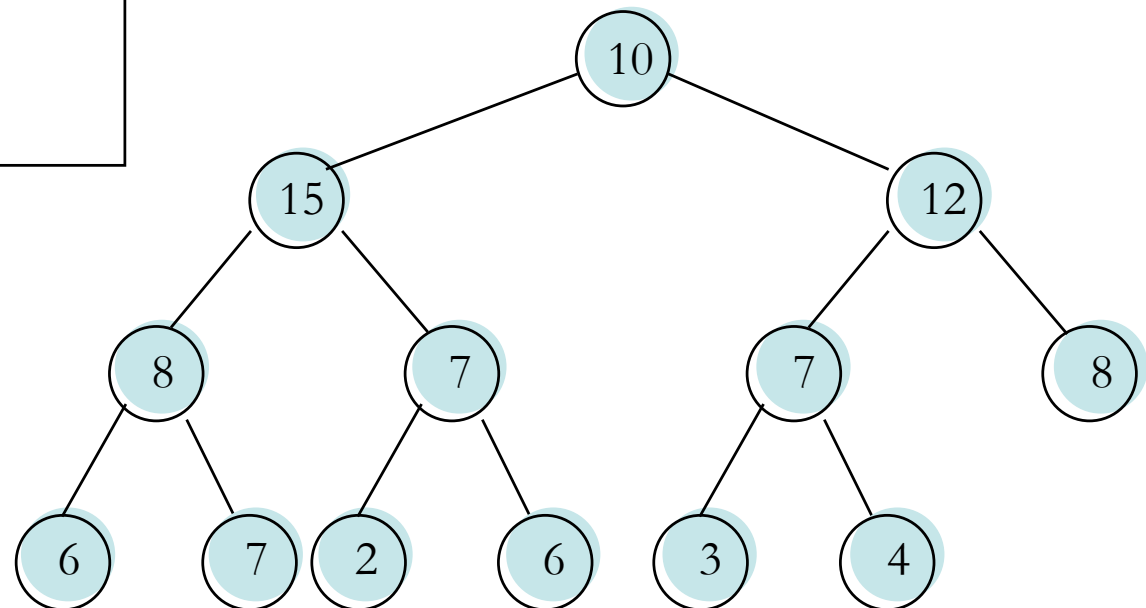
then $\text{largest} \leftarrow r$

if $\text{largest} \neq i$

then exchange $A[i] \leftrightarrow A[\text{largest}]$

HEAPIFY(A, largest)

Aplique el algoritmo
HEAPIFY(A, 1)



HEAPIFY(A, i)

$l \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$

then $\text{largest} \leftarrow l$

else $\text{largest} \leftarrow i$

if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$

then $\text{largest} \leftarrow r$

if $\text{largest} \neq i$

then exchange $A[i] \leftrightarrow A[\text{largest}]$

HEAPIFY(A, largest)

¿Cuál es la complejidad del algoritmo?

HEAPIFY(A, i)

l ← LEFT(i)

r ← RIGHT(i)

if l ≤ heap-size[A] and A[l] > A[i]

then largest ← l

else largest ← i

if r ≤ heap-size[A] and A[r] > A[largest]

then largest ← r

if largest ≠ i

then exchange A[i] ↔ A[largest]

HEAPIFY(A, largest)

Complejidad

$$T(n) \leq T(2n/3) + \Theta(1)$$

*$\Theta(1)$ para calcular el mayor +
Heapify con 2/3 de los
elementos en el peor de los
casos*

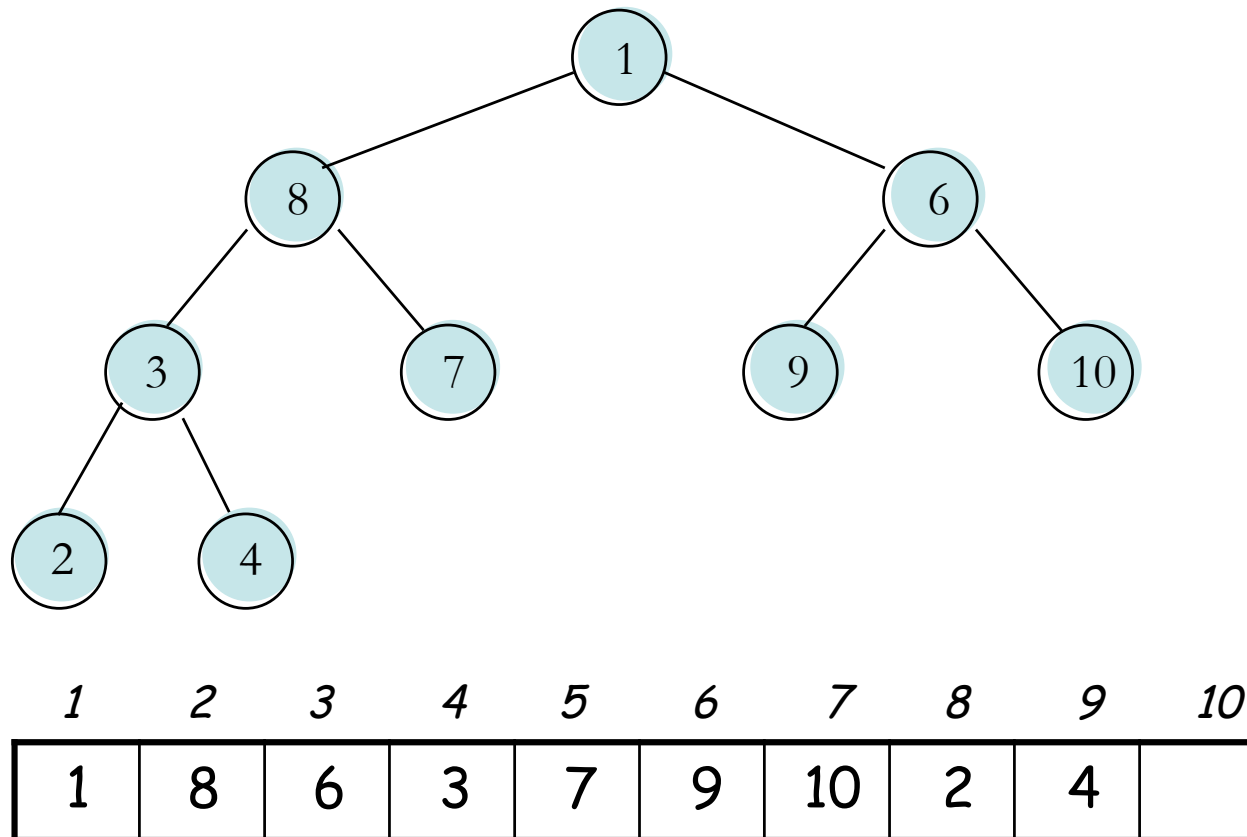
*Por teorema maestra, caso 2,
 $T(n) = O(\lg n)$*

Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo



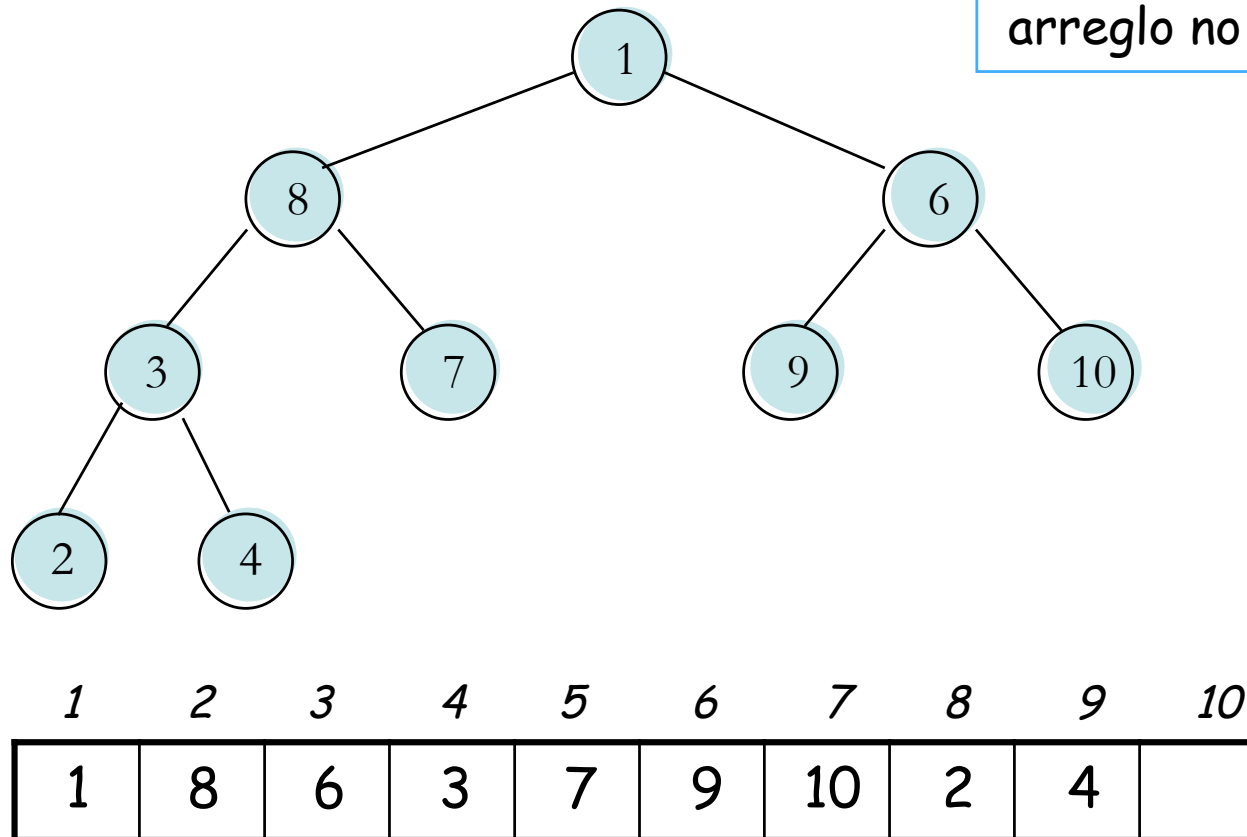
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

La organización es lógica, aun cuando en el arreglo no se especifica



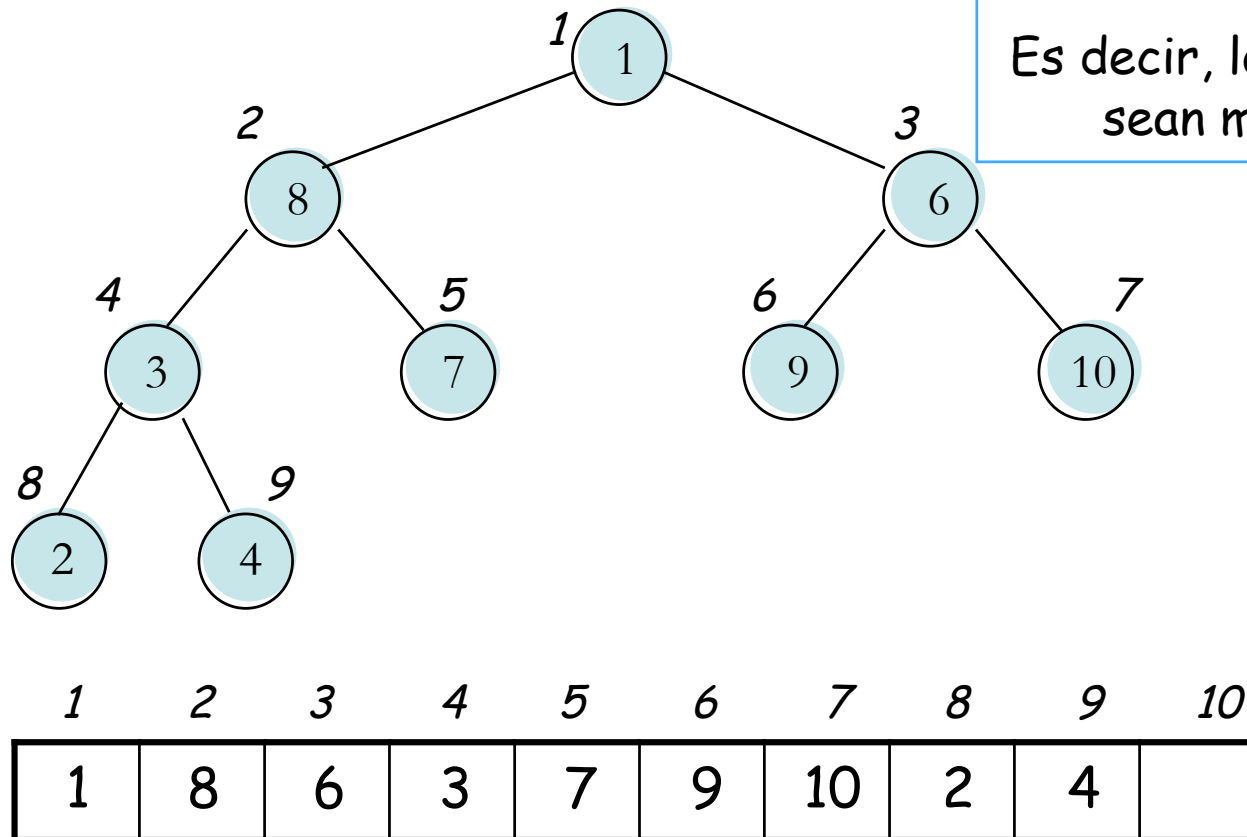
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



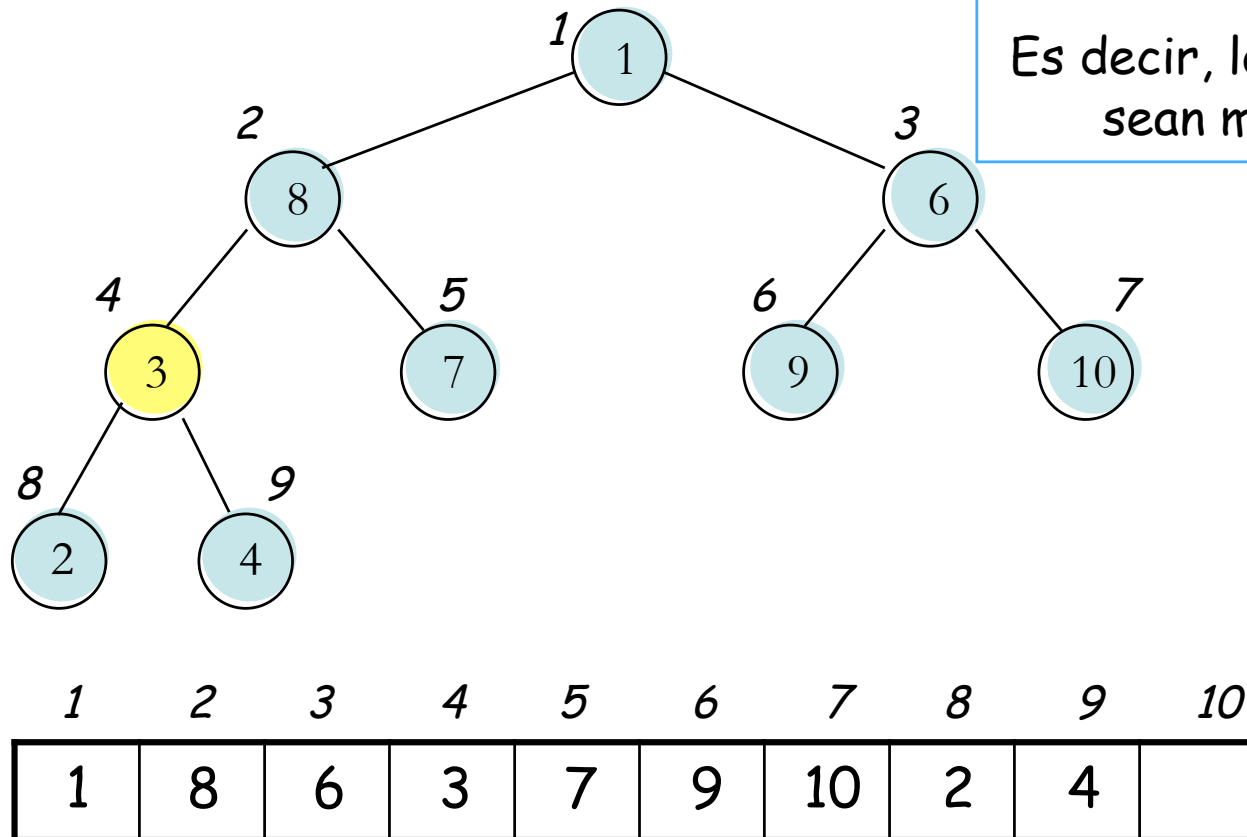
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subárboles
sean montículos



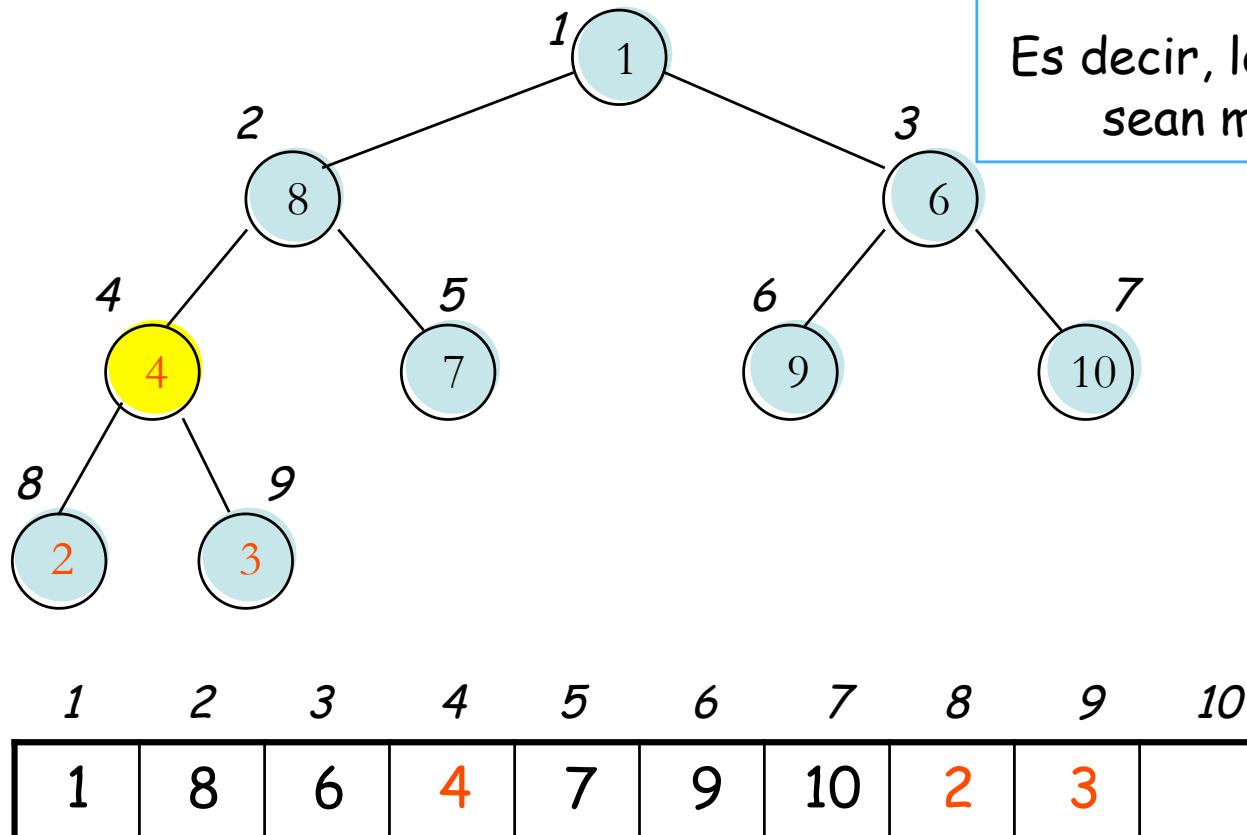
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subárboles
sean montículos



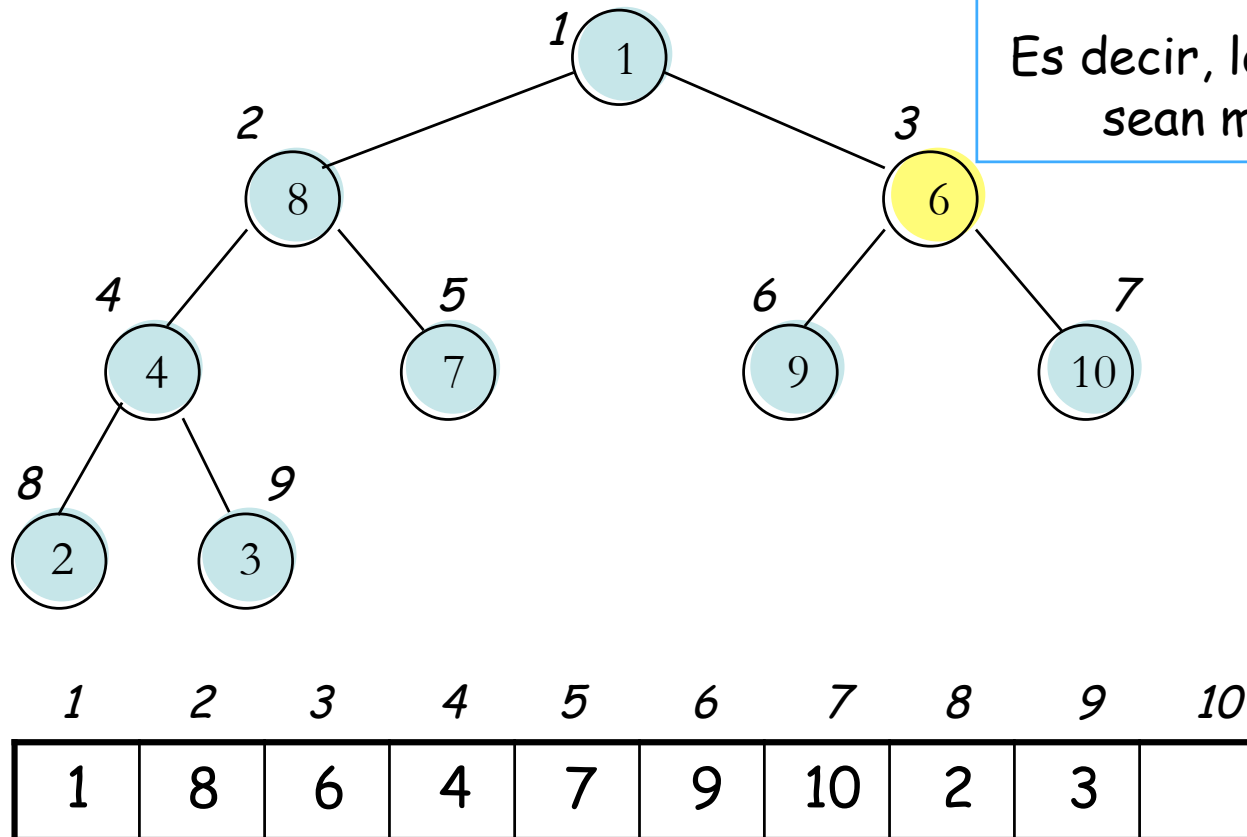
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



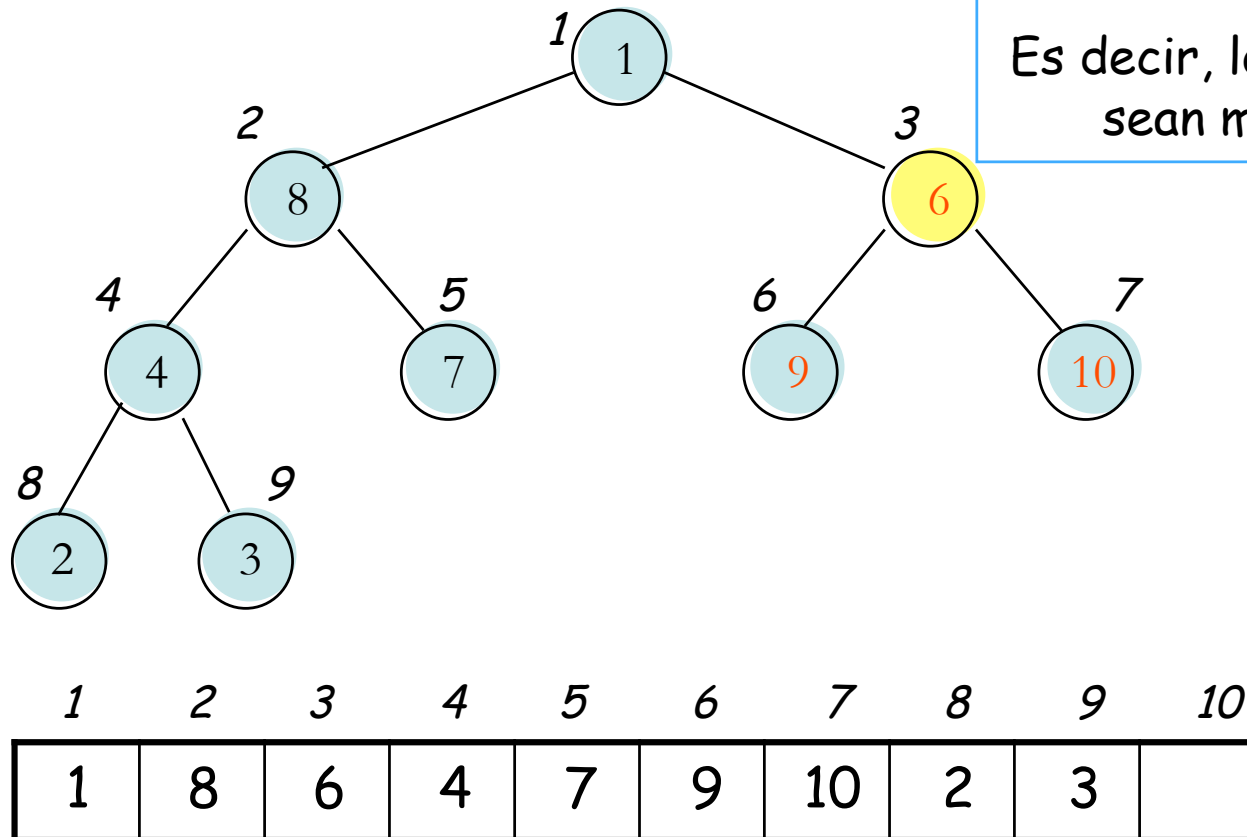
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



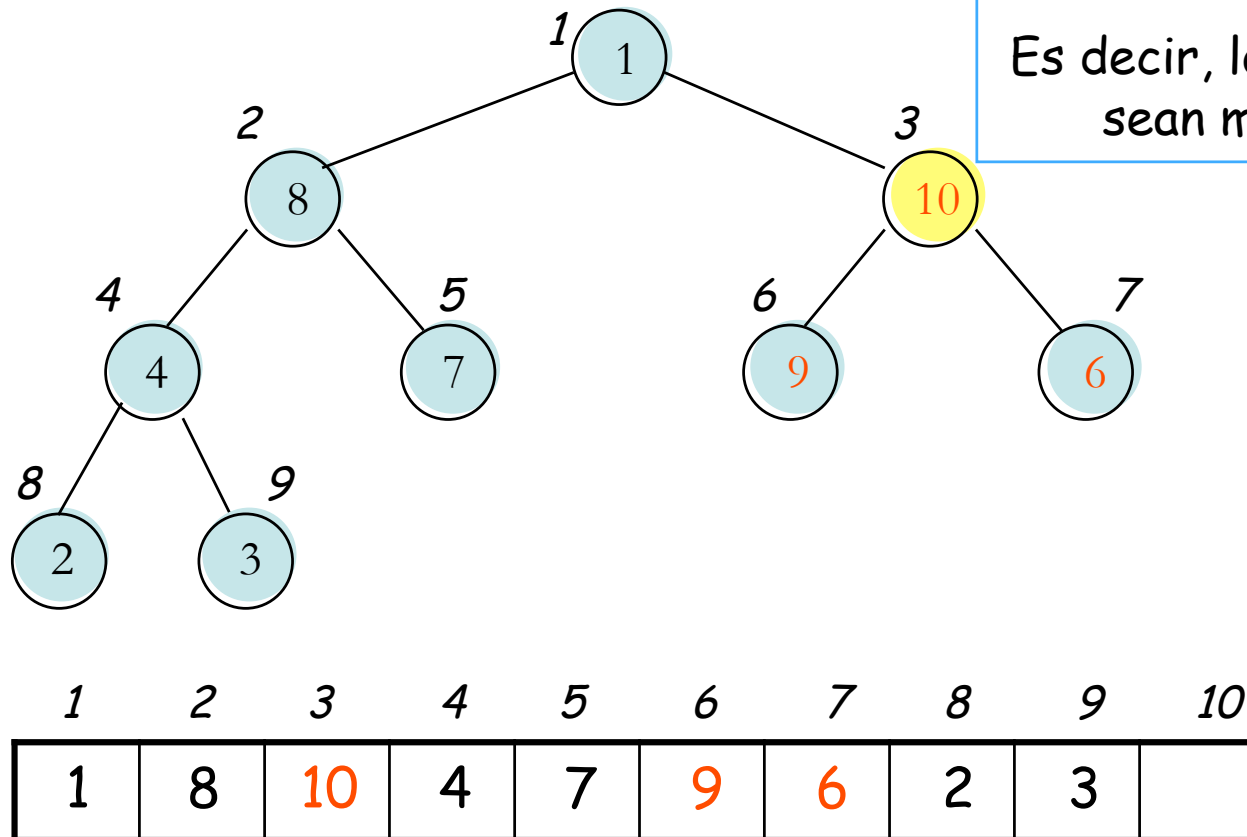
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



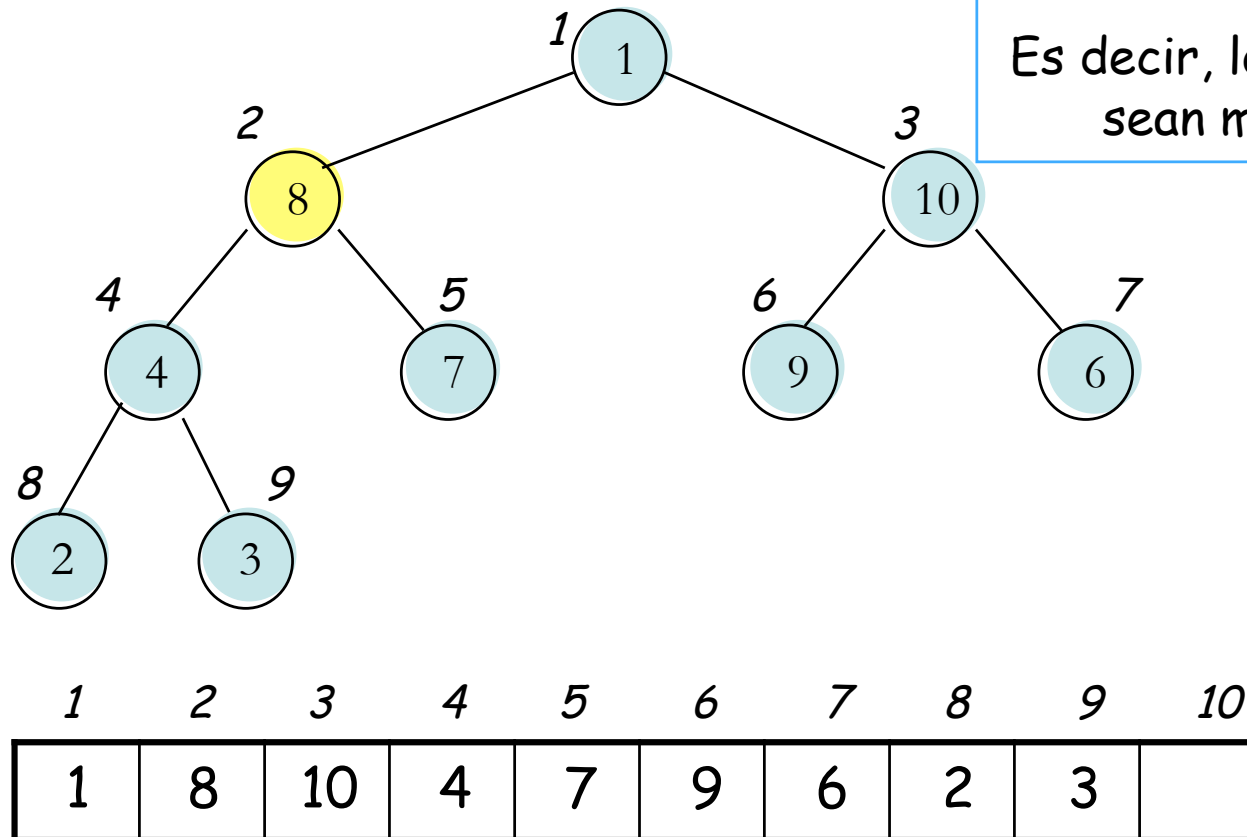
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



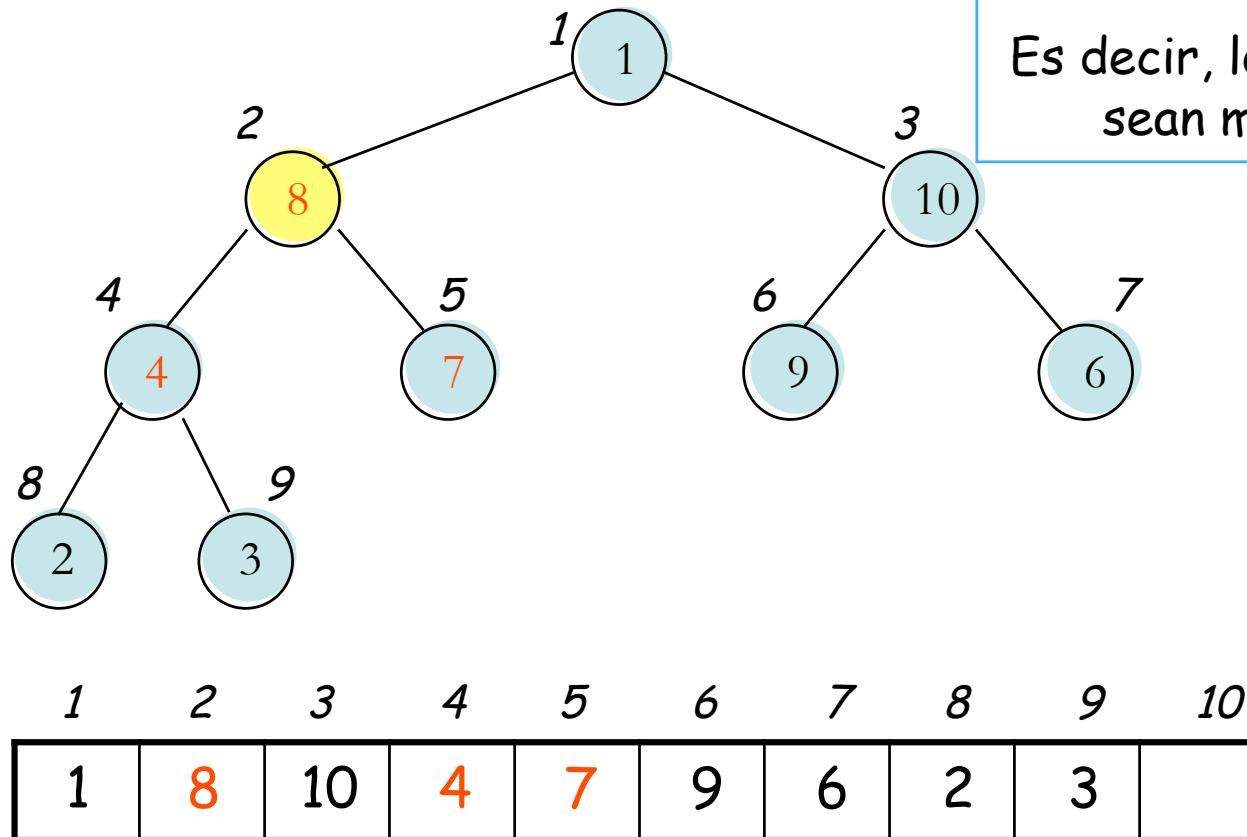
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



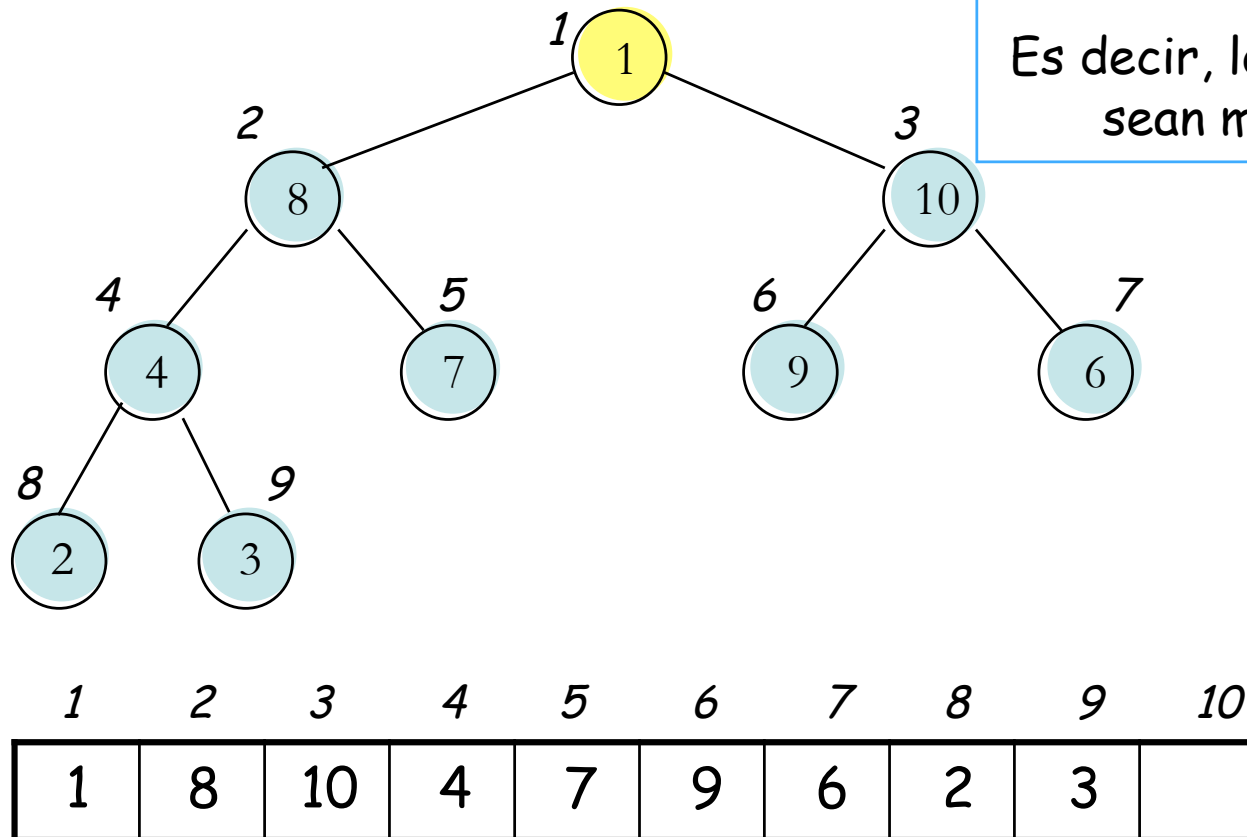
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



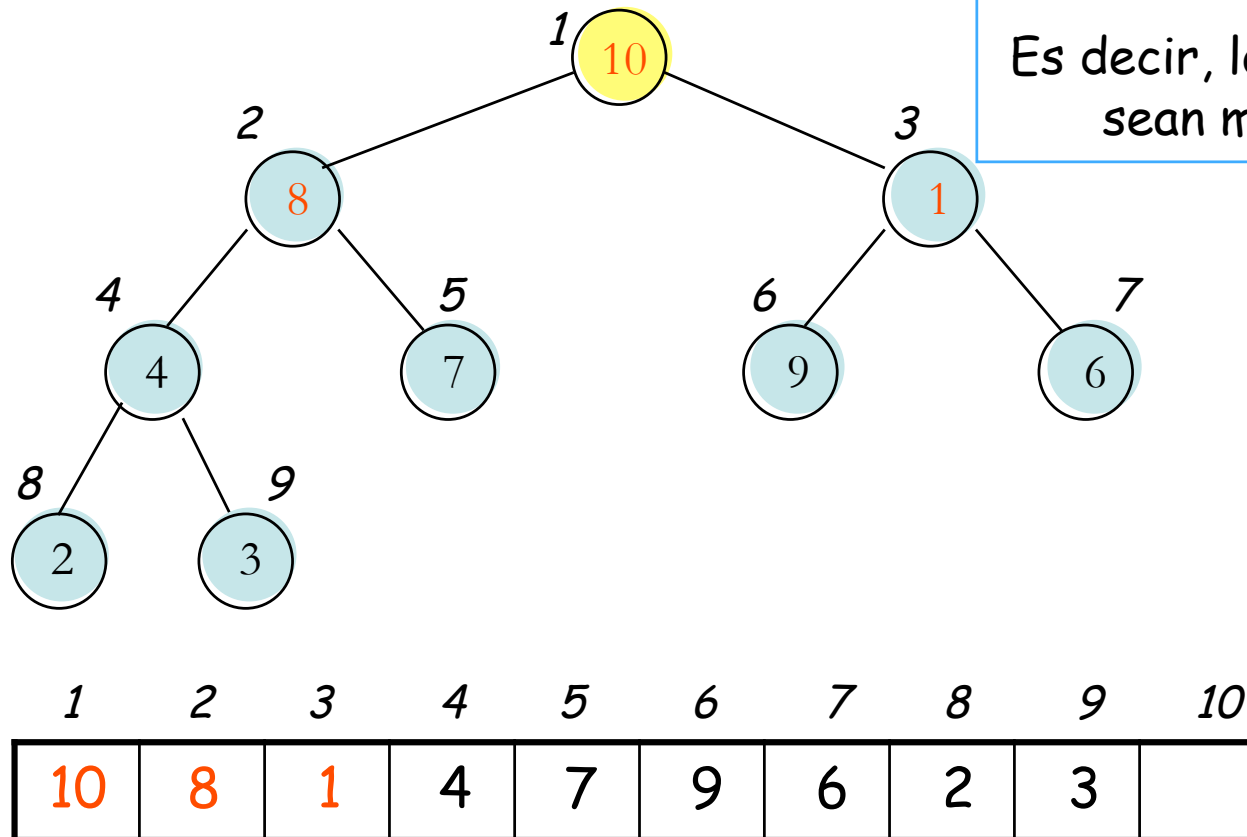
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subárboles
sean montículos



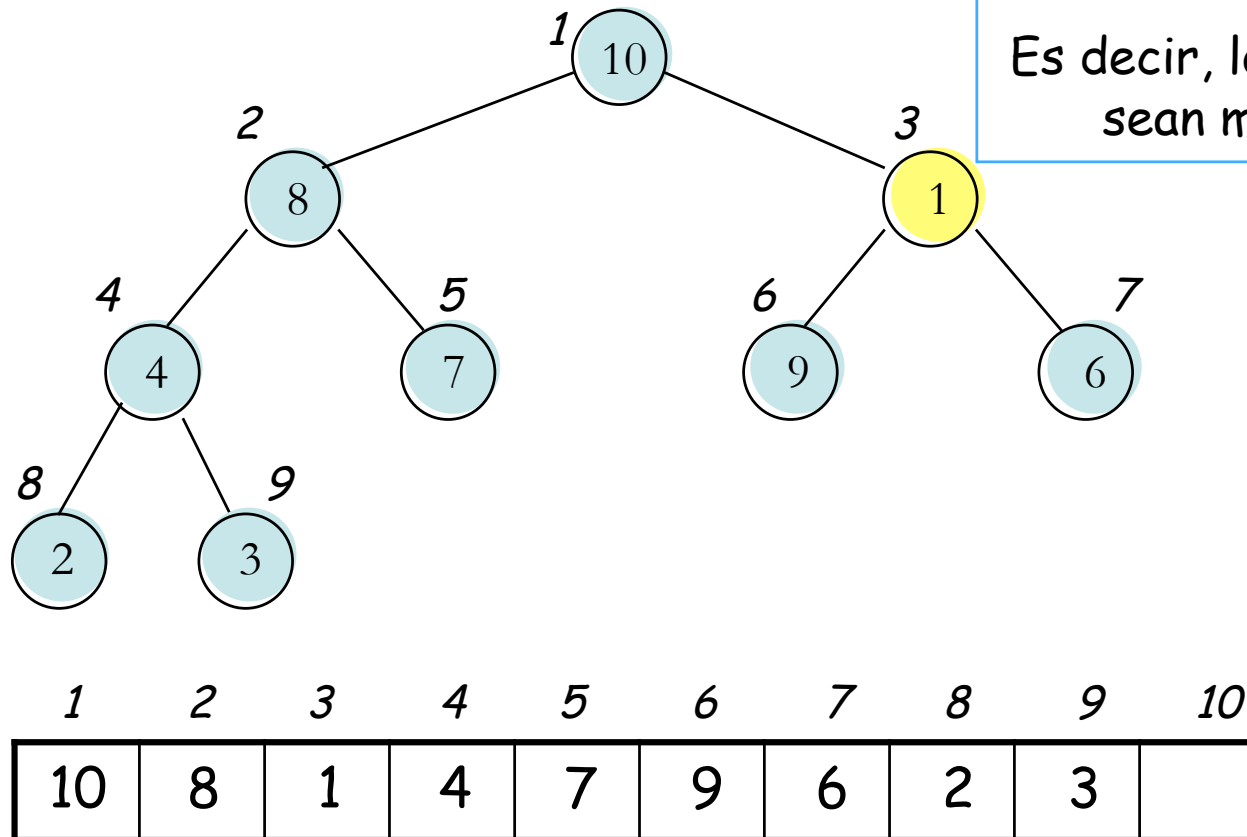
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos



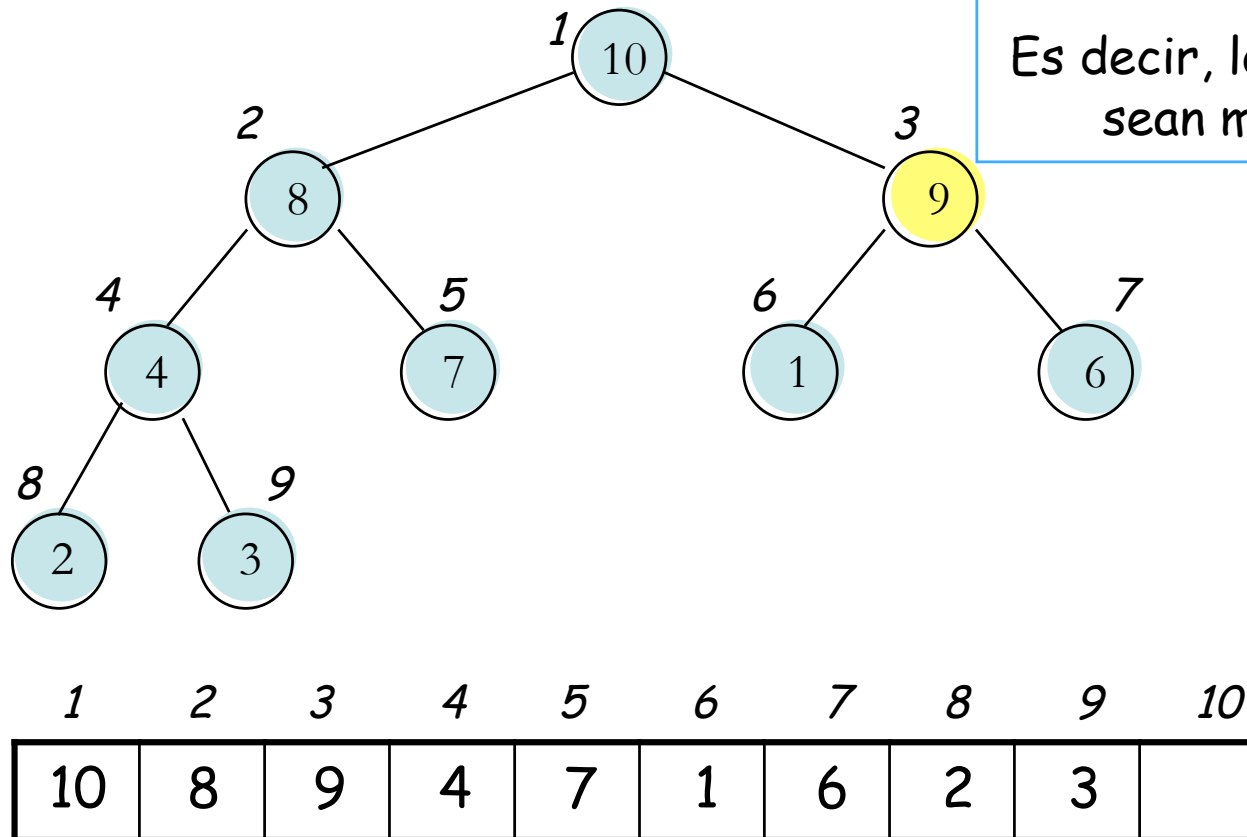
Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo

¿Sobre qué nodo se
puede hacer HEAPIFY?
Es decir, los subarboles
sean montículos

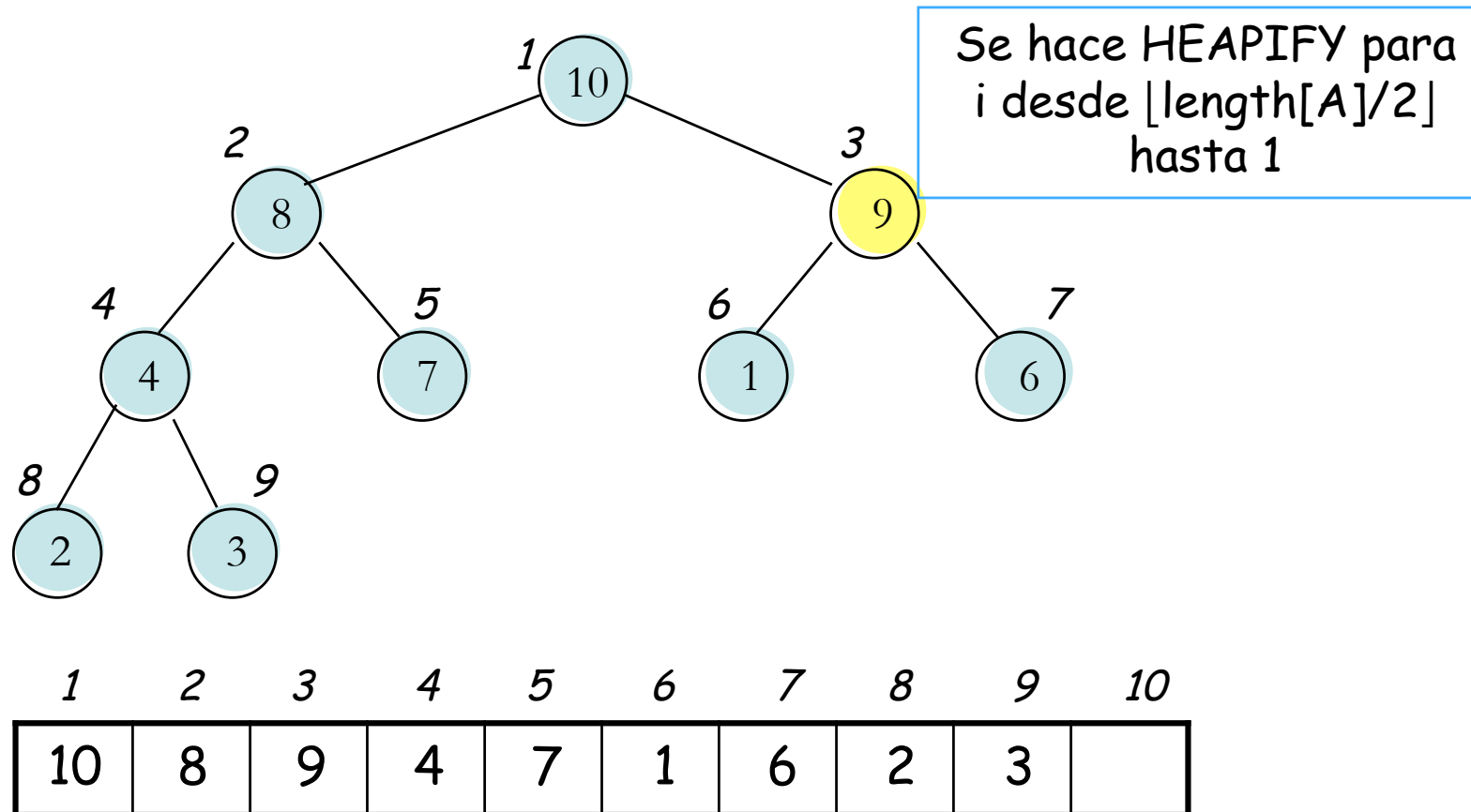


Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo



BUILD-HEAP(A)

heap-size[A] \leftarrow length[A]

for $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ downto 1

do HEAPIFY(A, i)

Aplique el algoritmo BUILD-HEAP(A), para

$A = \{5, 7, 10, 1, 4, 6, 8, 2, 9, 12\}$ y heap-size(A)=10

BUILD-HEAP(A)

heap-size[A] \leftarrow length[A]

for $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ downto 1

do HEAPIFY(A, i)

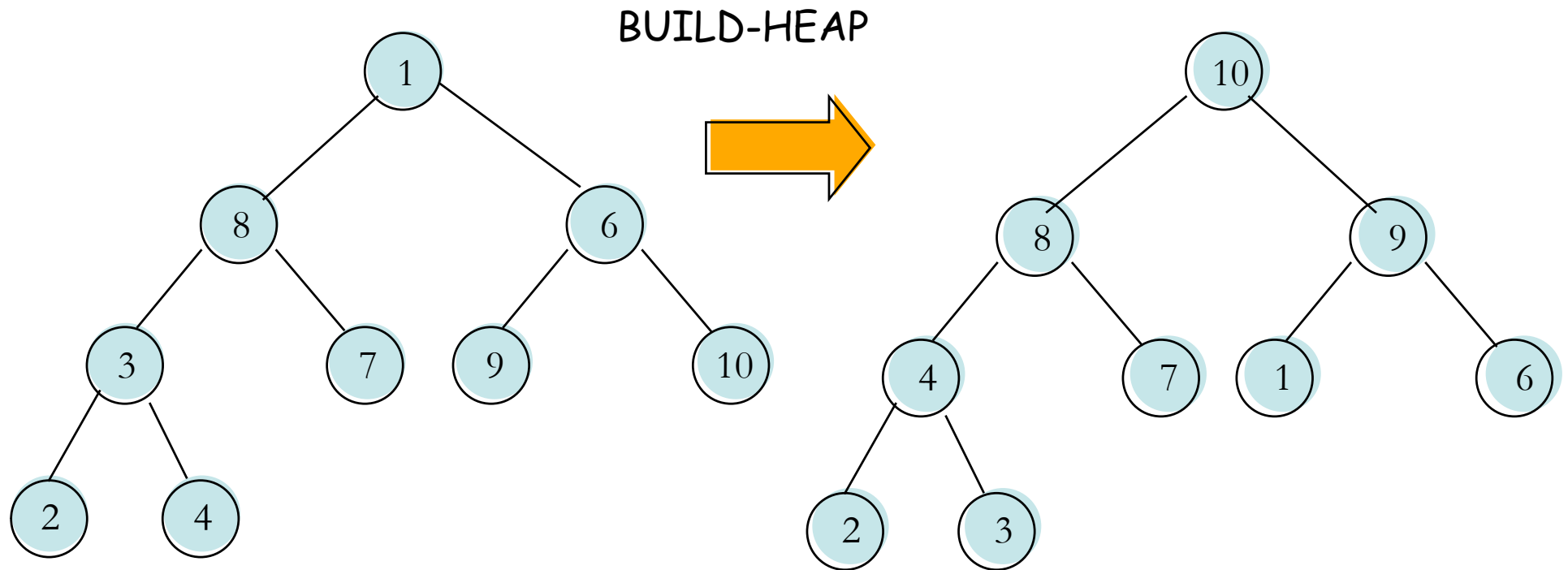
Complejidad

- Cada llamado a HEAPIFY cuesta $O(\lg n)$
- Se hacen $O(n)$ llamados
- Estimación: $O(n \lg n)$

- $O(n)$ es una estimación más precisa

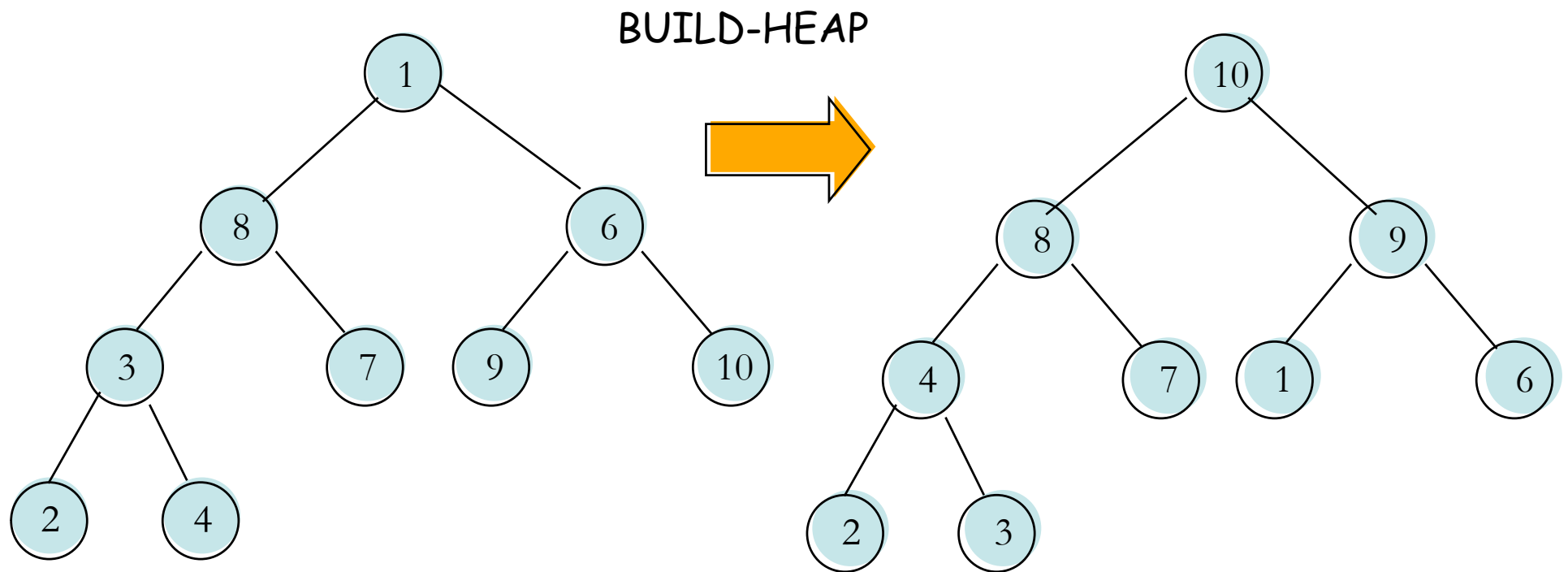
Heapsort

HEAP-SORT(A)



Heapsort

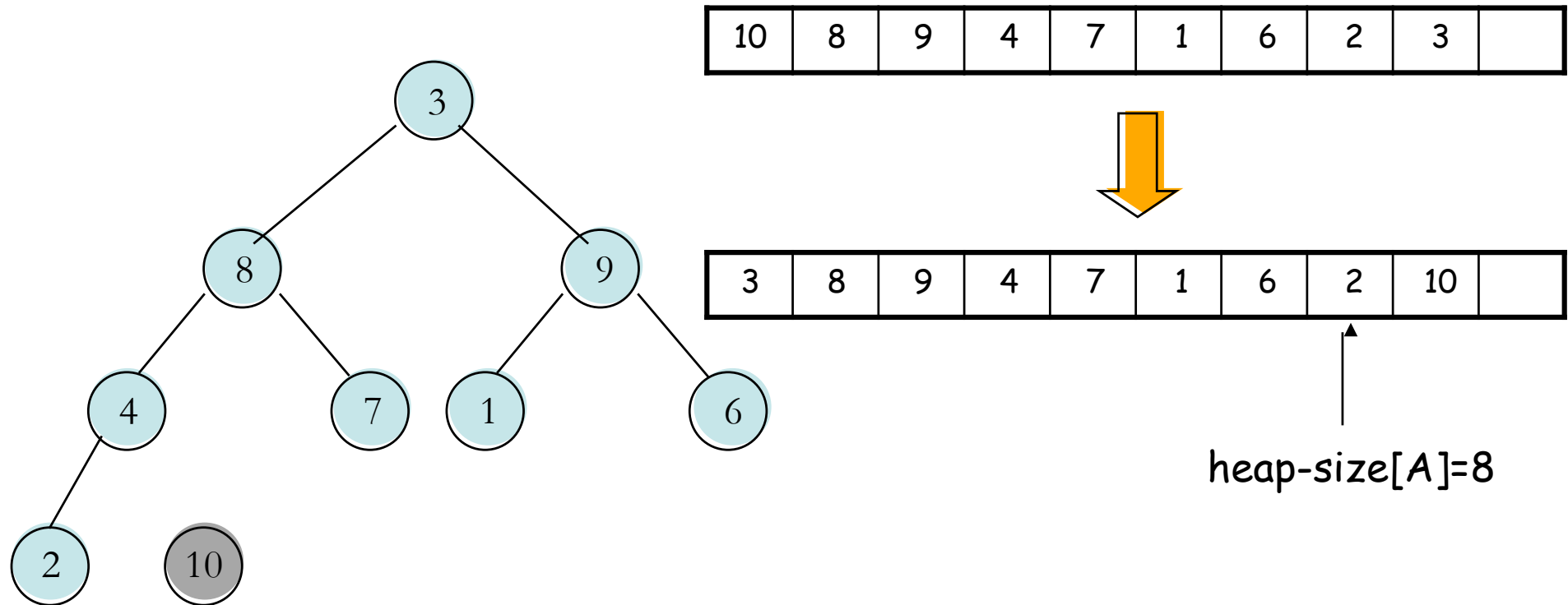
HEAP-SORT(A)



El valor más grande quedará en la raíz del árbol

Heapsort

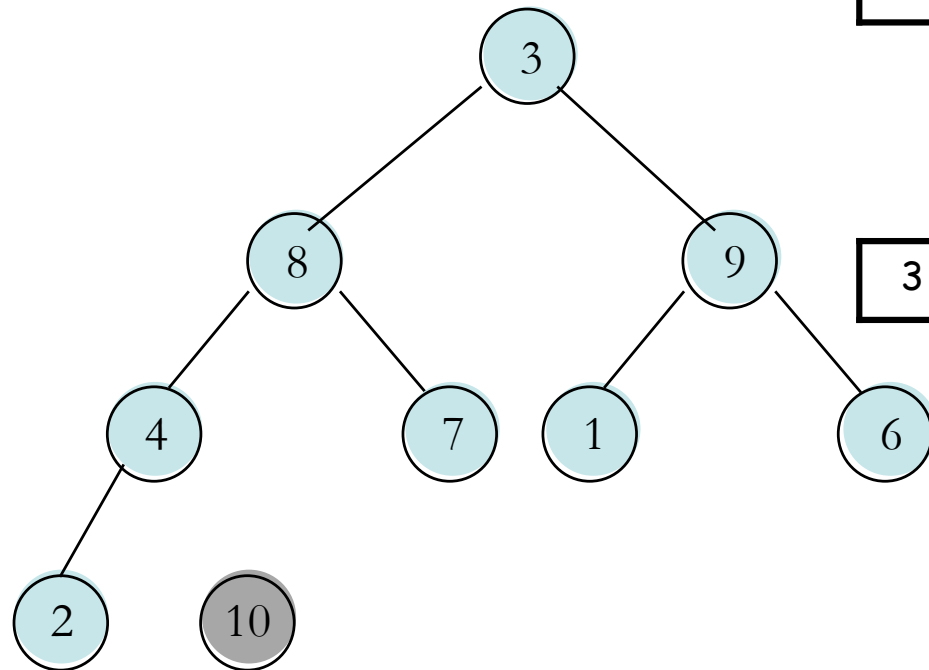
HEAP-SORT(A)



Se intercambia el valor $A[1]$, el mayor, con el valor $A[\text{heap-size}[A]]$ y se disminuye en 1 valor $\text{heap-size}[A]$

Heapsort

HEAP-SORT(A)



Se intercambia el valor $A[1]$, el mayor, con el valor $A[\text{heap-size}[A]]$ y se disminuye en 1 valor $\text{heap-size}[A]$

10	8	9	4	7	1	6	2	3	
----	---	---	---	---	---	---	---	---	--



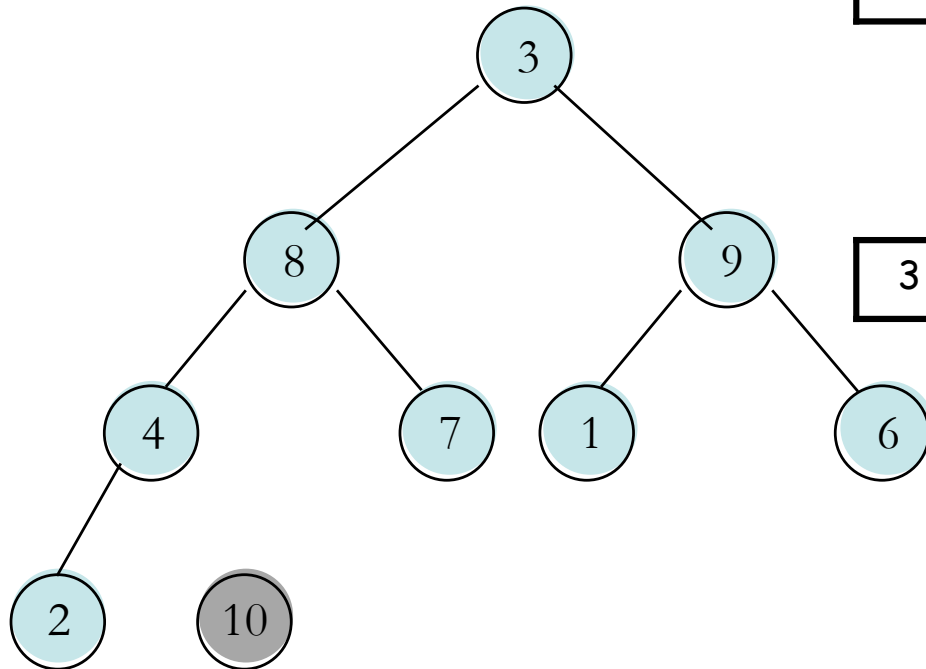
3	8	9	4	7	1	6	2	10	
---	---	---	---	---	---	---	---	----	--

$\text{heap-size}[A]=8$

Se repite el proceso de hacer heapify para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)



10	8	9	4	7	1	6	2	3	
----	---	---	---	---	---	---	---	---	--



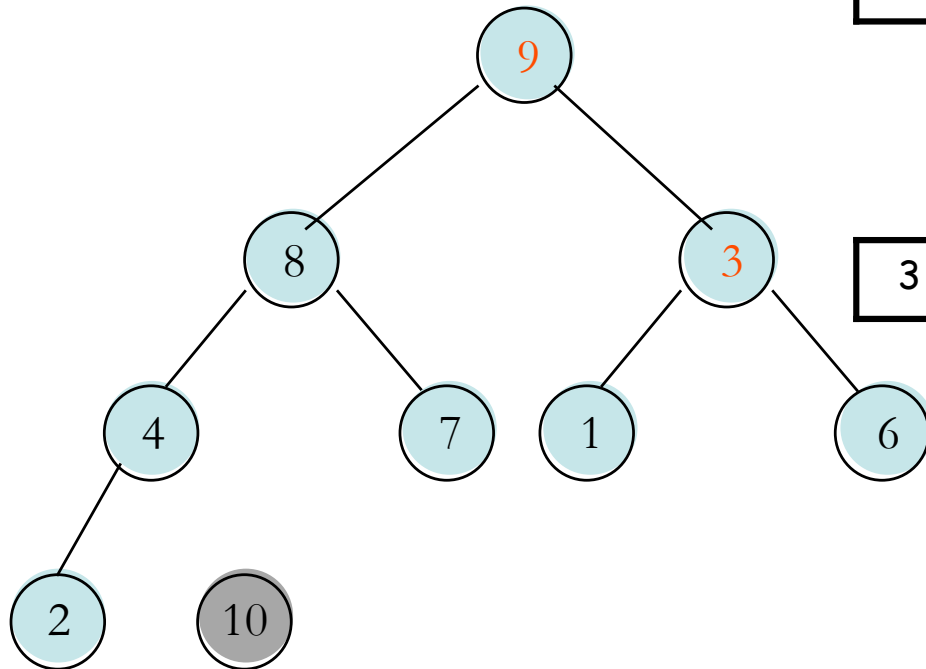
3	8	9	4	7	1	6	2	10	
---	---	---	---	---	---	---	---	----	--

heap-size[A]=8

Se repite el proceso de hacer heapify(A,1) para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)



10	8	9	4	7	1	6	2	3	
----	---	---	---	---	---	---	---	---	--



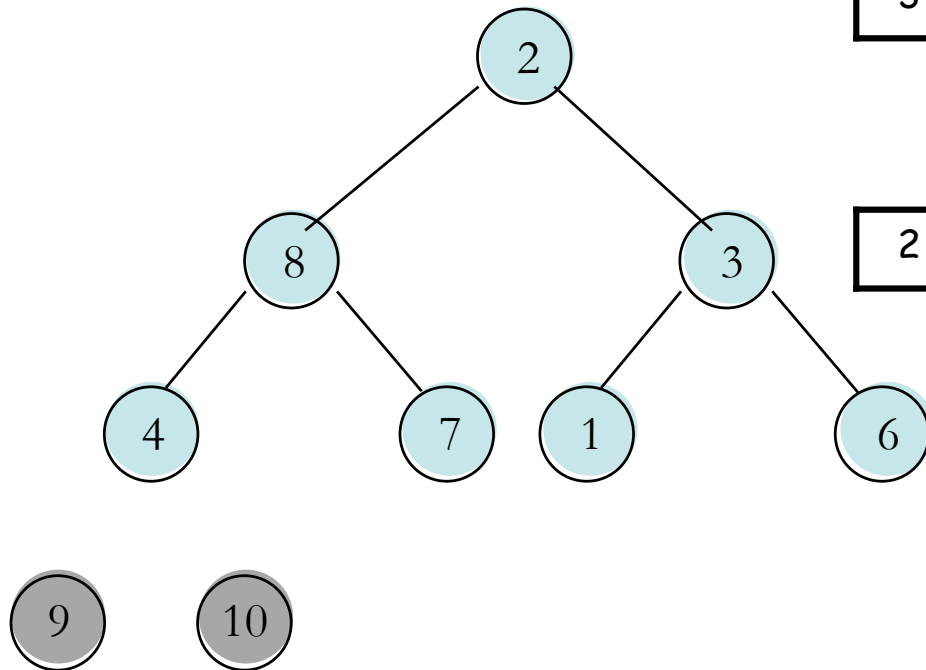
3	8	9	4	7	1	6	2	10	
---	---	---	---	---	---	---	---	----	--

heap-size[A]=8

Se repite el proceso de hacer heapify(A,1) para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)



3	8	9	4	7	1	6	2	10	
---	---	---	---	---	---	---	---	----	--



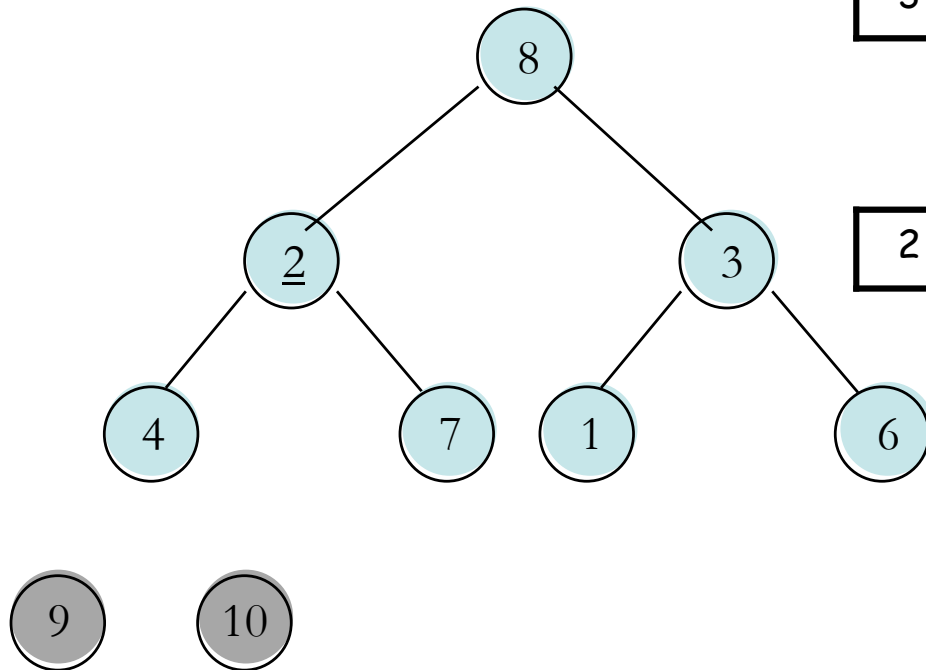
2	8	3	4	7	1	6	9	10	
---	---	---	---	---	---	---	---	----	--

↑
heap-size[A]=7

Se repite el proceso de hacer $\text{heapify}(A,1)$ para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)



3	8	9	4	7	1	6	2	10	
---	---	---	---	---	---	---	---	----	--



2	8	3	4	7	1	6	9	10	
---	---	---	---	---	---	---	---	----	--

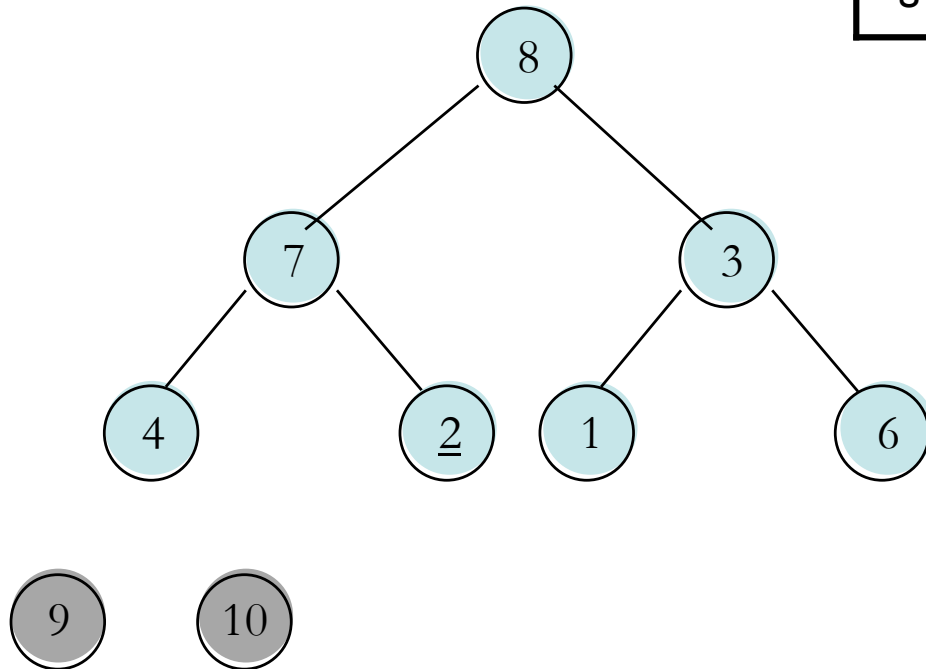
↑
heap-size[A]=7

Se repite el proceso de hacer $\text{heapify}(A,1)$ para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)

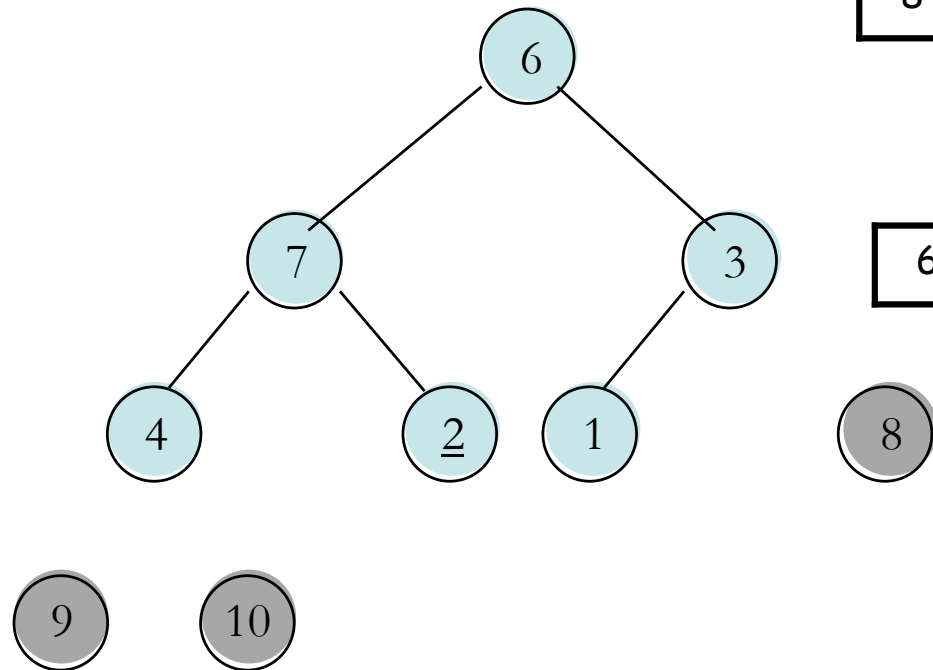
8	7	3	4	2	1	6	9	10	
---	---	---	---	---	---	---	---	----	--



Se repite el proceso de hacer $\text{heapify}(A,1)$ para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)



8	7	3	4	2	1	6	9	10	
---	---	---	---	---	---	---	---	----	--

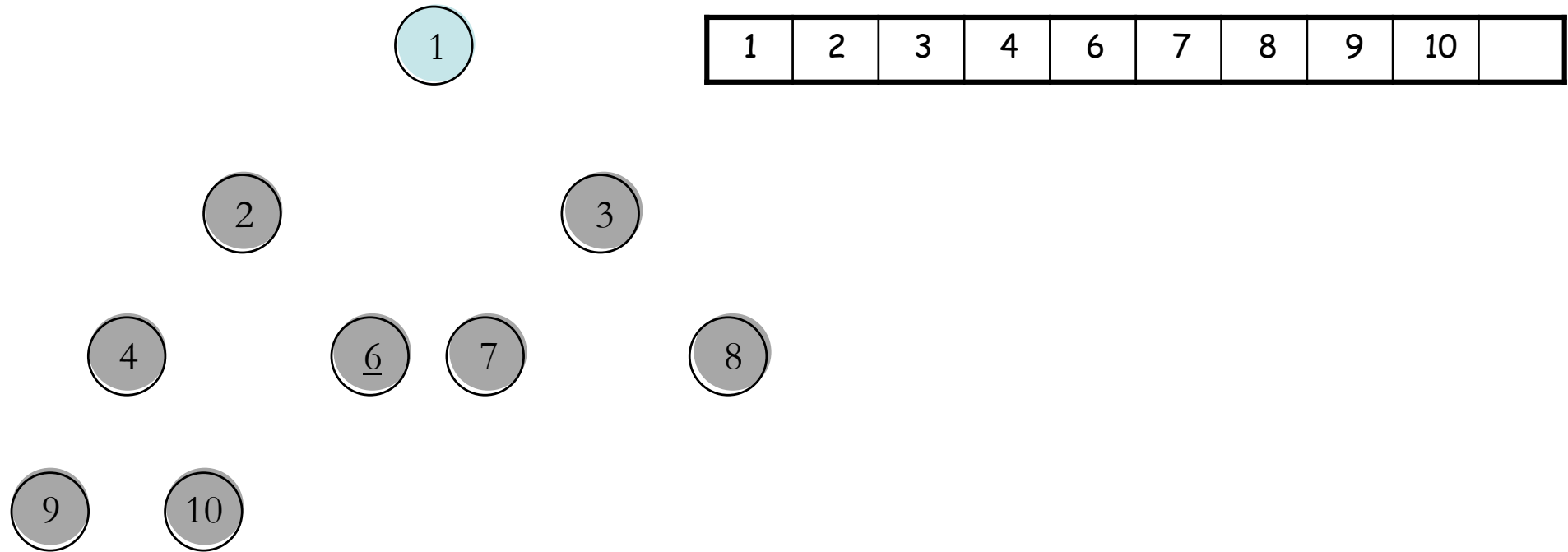


6	7	3	4	2	1	8	9	10	
---	---	---	---	---	---	---	---	----	--

Se repite el proceso de hacer $\text{heapify}(A,1)$ para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)



HEAP-SORT(A)

BUILD-HEAP(A)

for $i \leftarrow \text{length}[A]$ downto 2

do exchange $A[1] \leftrightarrow A[i]$

heap-size[A] \leftarrow heap-size[A] - 1

HEAPIFY(A,1)

Aplique el algoritmo HEAP-SORT(A), para
 $A = \{12, 9, 10, 7, 8, 1\}$ y heap-size(A)=6

HEAP-SORT(A)

BUILD-HEAP(A)

for $i \leftarrow \text{length}[A]$ downto 2

do exchange $A[1] \leftrightarrow A[i]$

heap-size[A] \leftarrow heap-size[A] - 1

HEAPIFY(A,1)

Aplique el algoritmo HEAP-SORT(A), para

$A = \{5, 7, 10, 1, 4, 6, 8, 2, 9, 12\}$ y heap-size(A)=10

HEAP-SORT(A)

BUILD-HEAP(A)

for $i \leftarrow \text{length}[A]$ downto 2

do exchange $A[1] \leftrightarrow A[i]$

heap-size[A] \leftarrow heap-size[A] - 1

HEAPIFY(A,1)

¿Cuál es la complejidad?

HEAP-SORT(A)

BUILD-HEAP(A)

for $i \leftarrow \text{length}[A]$ downto 2

do exchange $A[1] \leftrightarrow A[i]$

heap-size[A] \leftarrow heap-size[A] - 1

HEAPIFY(A,1)

¿Cuál es la complejidad?

- BUILD-HEAP toma $O(n)$
- Se llama $(n-1)$ veces a HEAPIFY que toma $O(\lg n)$
- La complejidad es de $O(n \lg n)$

Heapsort

Colas de prioridad

- Es una estructura de datos con servicios de inserción y retiro de elementos con base en una prioridad (valor numérico almacenado en el árbol)
- Se retira (atiende) al elemento con mayor prioridad
- Las operaciones básicas son:
 - **INSERT**(C,x): insertar el elemento con clave x
 - **MAX**(C): devuelve el elemento de máxima prioridad
 - **EXTRACT-MAX**(C): elimina y devuelve el elemento de máxima prioridad

Heapsort

HEAP-MAXIMUM(C)

return $A[1]$

Tiempo de ejecución: $\Theta(1)$

Heapsort

HEAP-EXTRACT-MAX(C)

```
if heap-size[A] < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
HEAPIFY(A, 1)
return max
```

Tiempo de ejecución: $O(\lg n)$

Heapsort

HEAP-INCREASE-KEY(A, i, key)

if $\text{key} < A[i]$

then error "key error "

$A[i] \leftarrow \text{key}$

while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$

do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$

$i \leftarrow \text{PARENT}(i)$

Tiempo de ejecución: $O(\lg n)$

Heapsort

MAX-HEAP-INSERT(A , key)

heap-size[A] \leftarrow heap-size[A]+1

$i \leftarrow$ heap-size[A]

while $i > 1$ and $A[\text{PARENT}(i)] < key$

do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$

$i \leftarrow \text{PARENT}(i)$

$A[i] \leftarrow key$

Tiempo de ejecución: $O(\lg n)$