Fundamentos de análisis y diseño de algoritmos

Montículos y Heapsort

Comparación de algoritmos de ordenamiento

Propiedades de orden y forma de los montones Monticulo

Operaciones HEAPIFY, BUILD-HEAP Y HEAP-SORT ordenamiento

Análisis de complejidades

Colas de prioridad

Heapsort

Problema de ordenamiento

- Insertion

  $O(n^2)$ ← Peor, Promedio $O(n)$ = Mejor caso

  Ordena in-place

- Merge ← Divide y vencerás

  $\Theta(n\lg n)$ ← promedio

  No ordena in-place ← crea estructura nueva

- Heapsort

  $O(n\lg n)$

  Ordena in-place

- Quicksort

  Pivote

  Peor caso → $\Theta(n^2)$, 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

$\rightarrow O(n)$

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

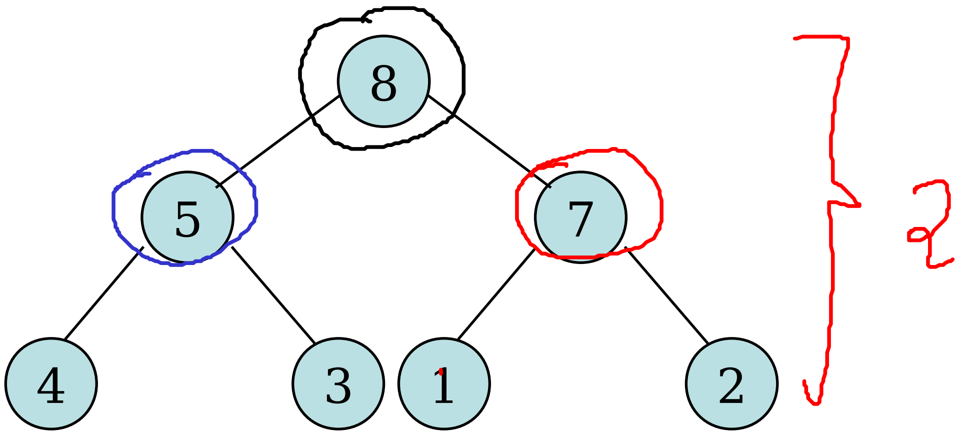
Montículos

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

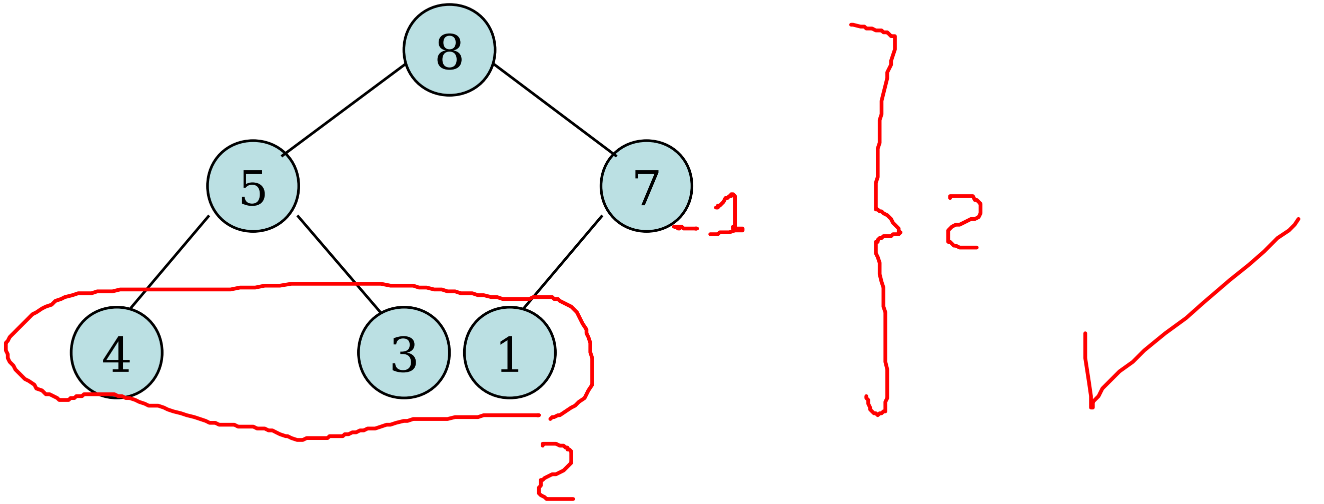
Heapsort

Analizar las propiedades de orden y de forma



Heapsort

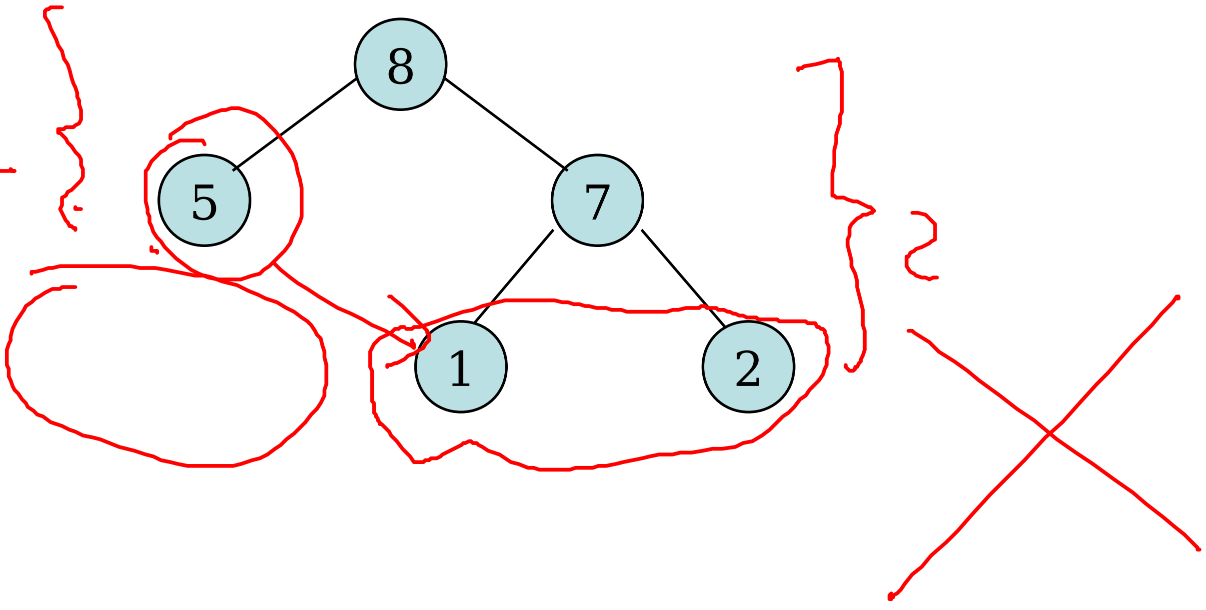
Analizar las propiedades de orden y de forma



Heapsort

Analizar las propiedades de orden y de forma

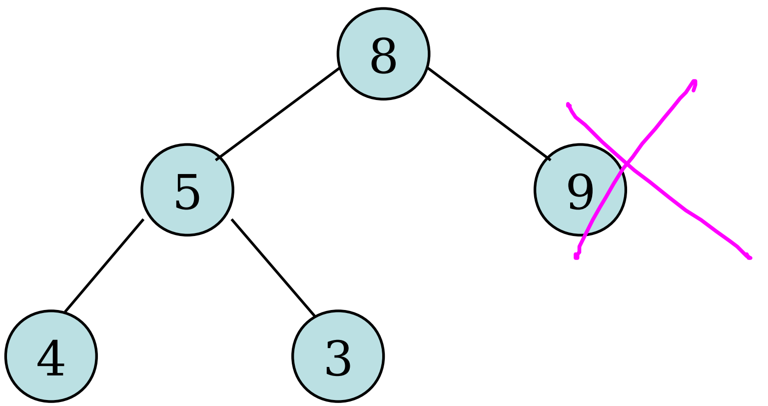
1



2

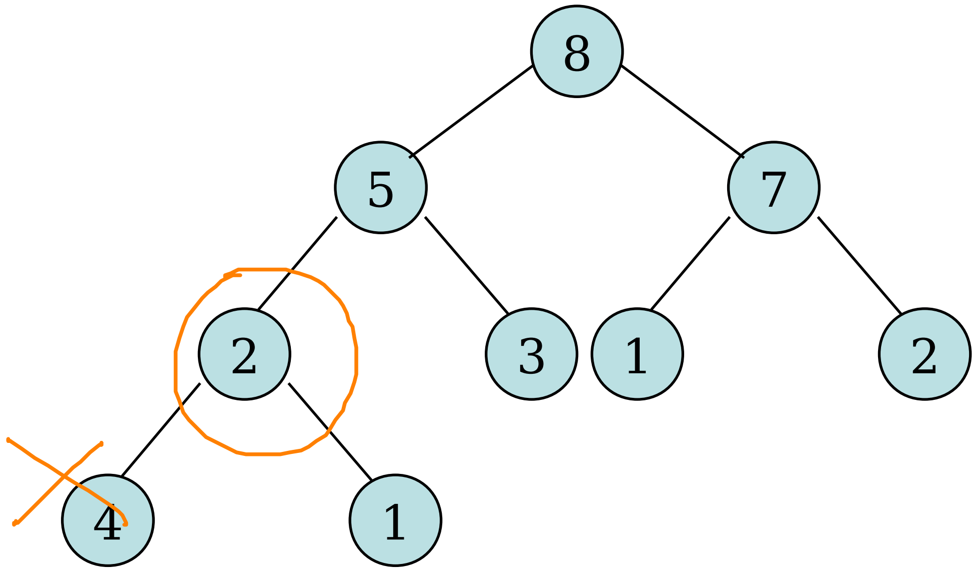
Heapsort

Analizar las propiedades de orden y de forma



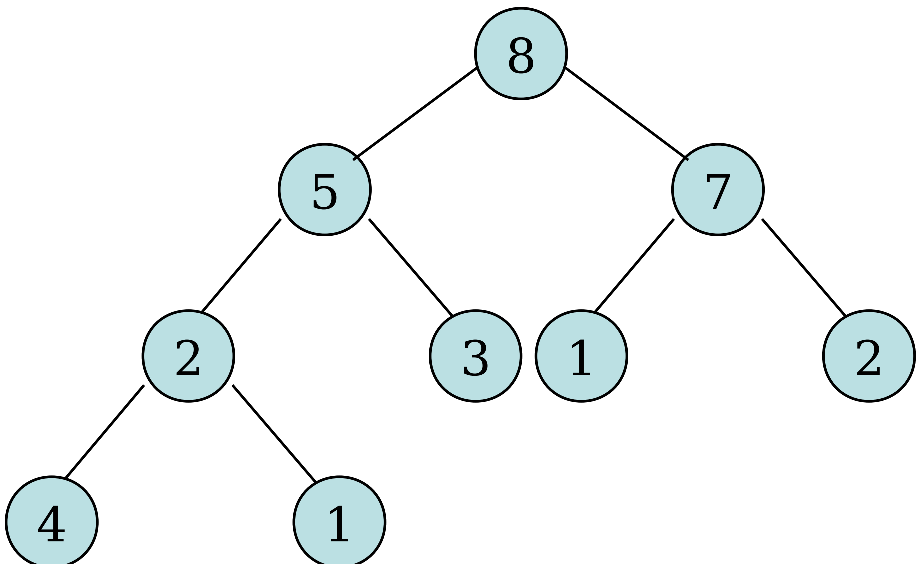
Heapsort

Analizar las propiedades de orden y de forma



Heapsort

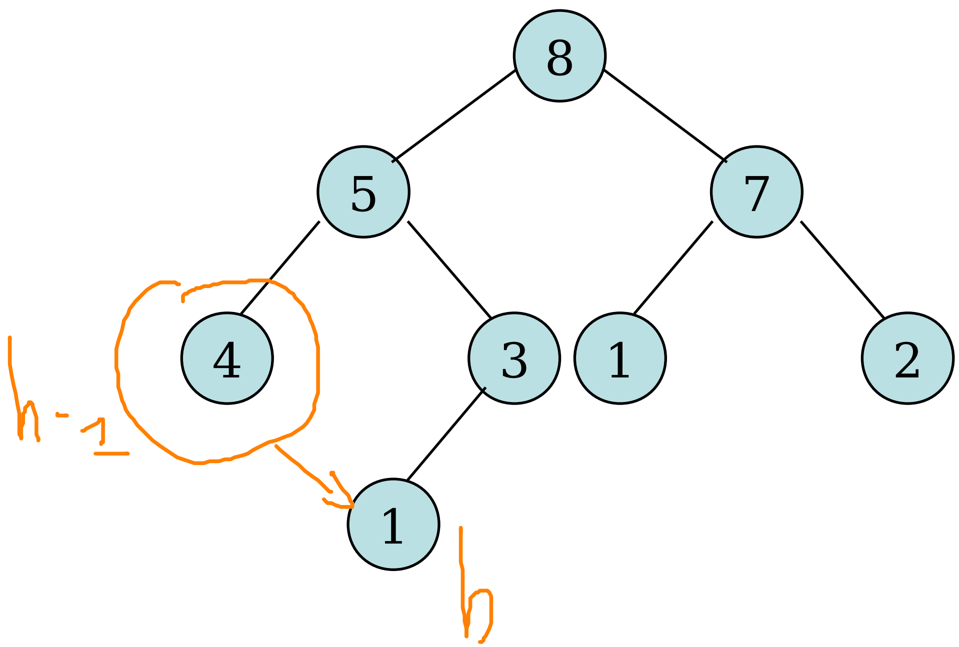
Analizar las propiedades de orden y de forma



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

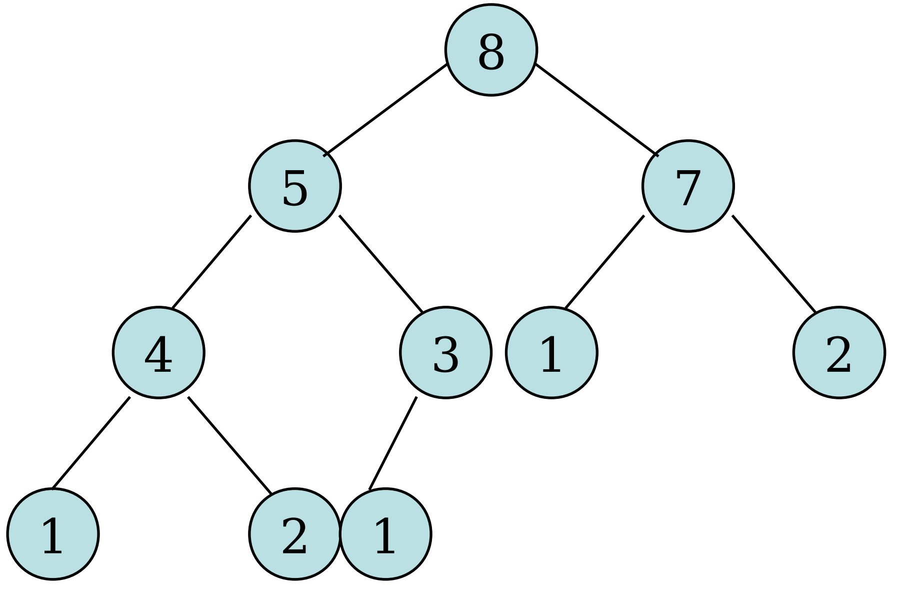
Heapsort

Indique si se cumplen las propiedades de orden y de forma



Heapsort

Indique si se cumplen las propiedades de orden y de forma

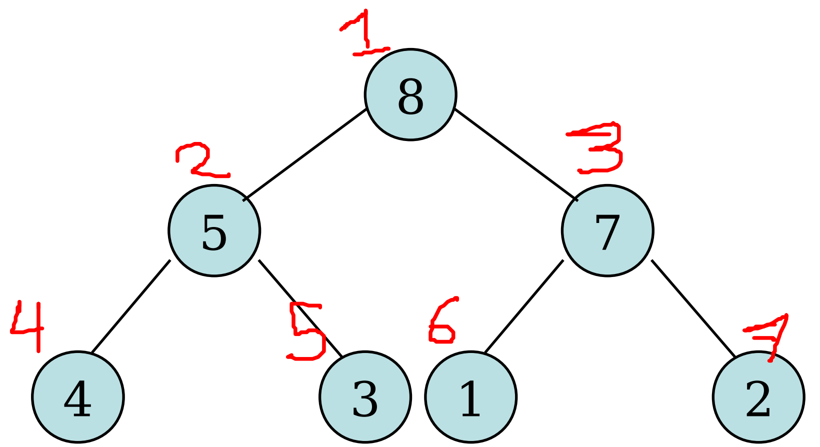


Heapsort

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

Izq

Der

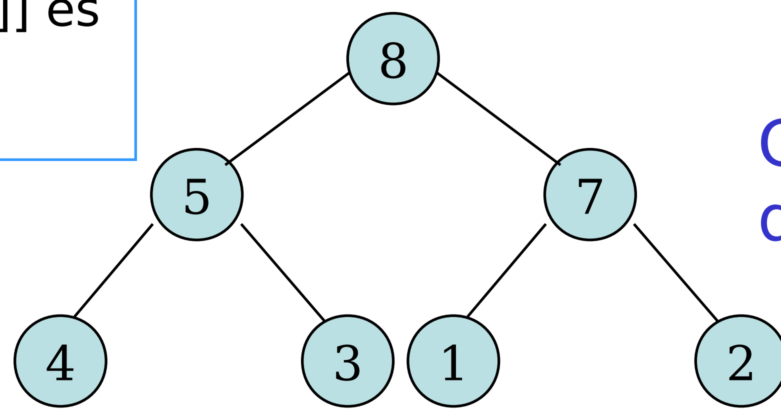


1	2	3	4	5	6	7			
8	5	7	4	3	1	2			

Heapsort

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

A[1, …, heap-size[A]] es el montículo representado



Capacidad máx del heap

Tamaño del heap

8	5	7	4	3	1	2			

heap-size[A]=7

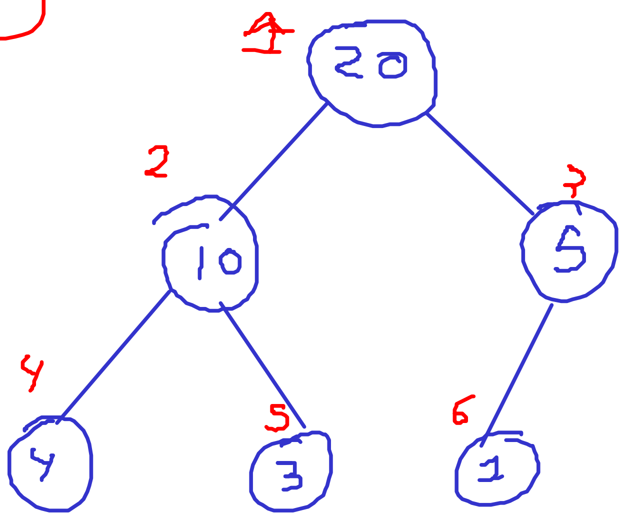
length[A]=10

Heapsort

Indique si se cumplen las propiedades de orden y de forma

1 2 3 4 5 6

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

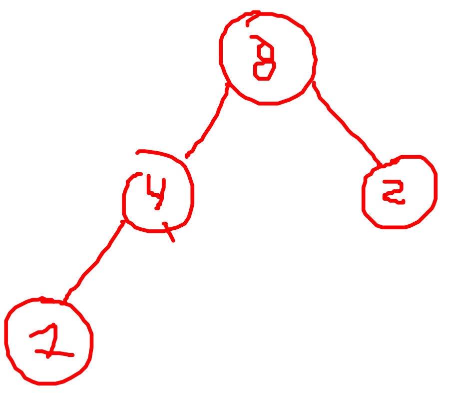


Heapsort

Indique si se cumplen las propiedades de orden y de forma

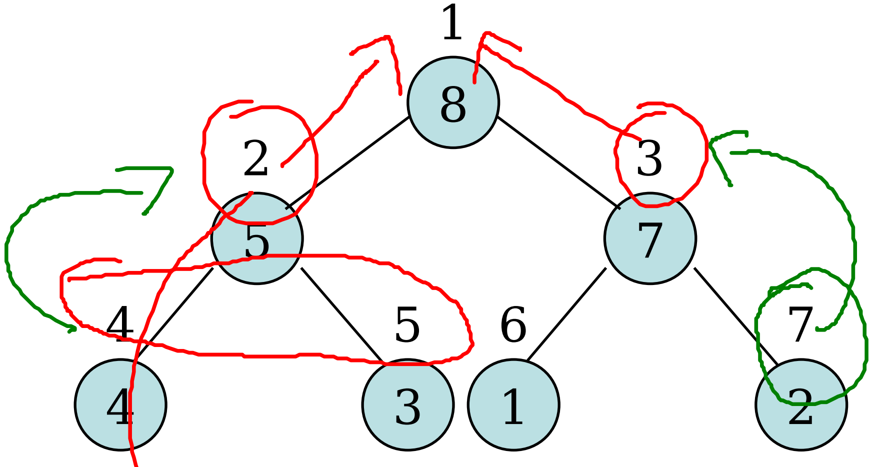
A={20, 10, 5, 4, 3, 1, } donde heap-size[A]=6 y length[A]=10

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



Heapsort

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



Pos

1	2	3	4	5	6	7	8	9	10
8	5	7	4	3	1	2			

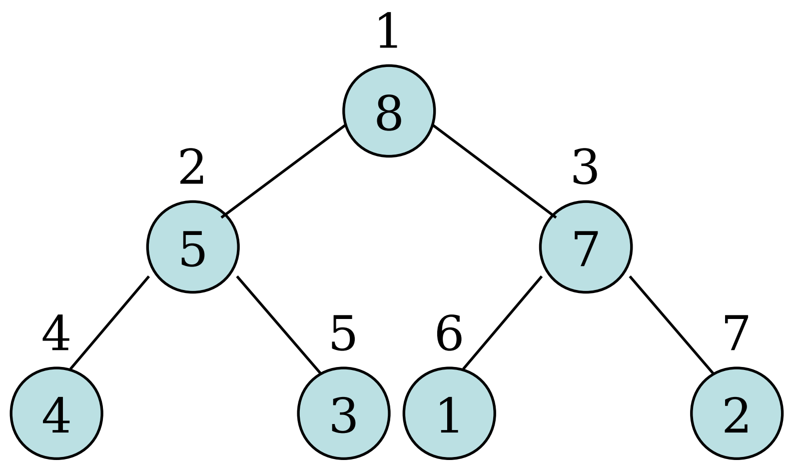
Evalue ⌊i/2⌋ para i=2 y 3

Evalue ⌊i/2⌋ para i=4 y 5

Evalue ⌊i/2⌋ para i=6 y 7

Heapsort

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



1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
8	5	7	4	3	1	2			

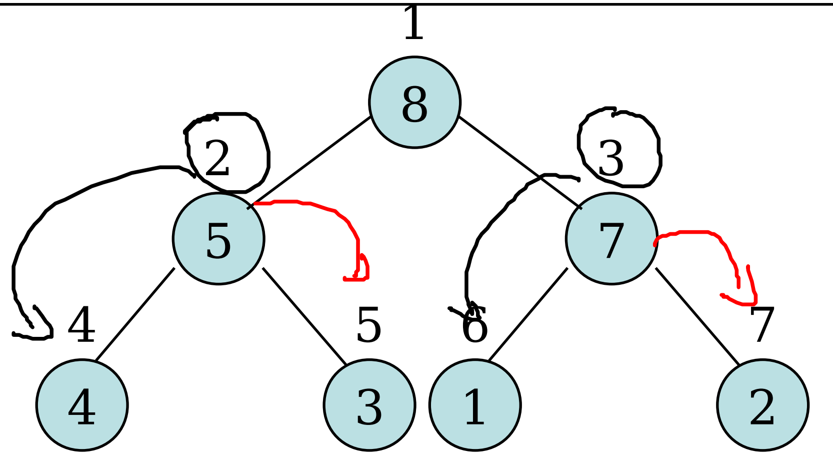
Evalue ⌊i/2⌋ para i=2 y 3

Evalue ⌊i/2⌋ para i=4 y 5

Evalue ⌊i/2⌋ para i=6 y 7

Padre(i): ⌊i/2⌋

Heapsort



1	2	3	4	5	6	7	8	9	10
8	5	7	4	3	1	2			

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) → O(h)
- Build-Heap: O(n)
- HeapSort: O(nlgn)
- 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 $\Theta(\lg n)$

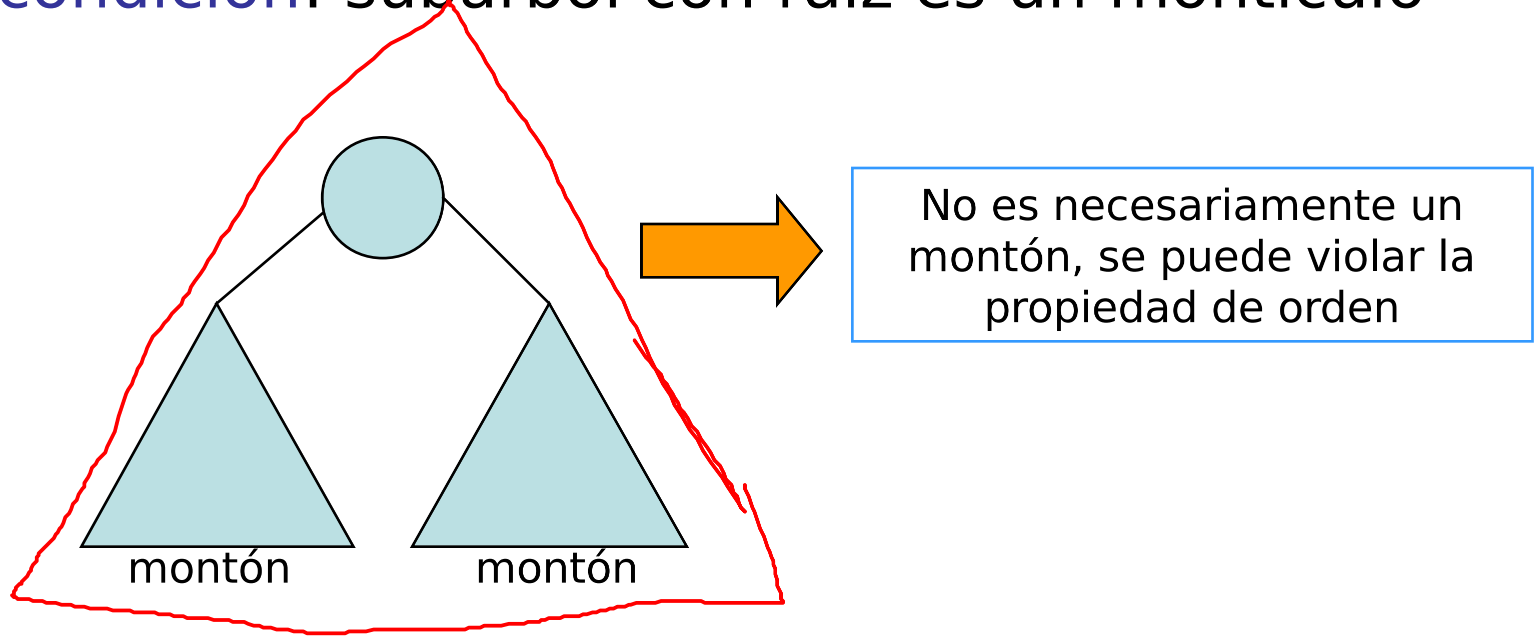
binario balanceado

Heapsort

Heapify

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

Poscondición: subárbol con raíz es un montículo

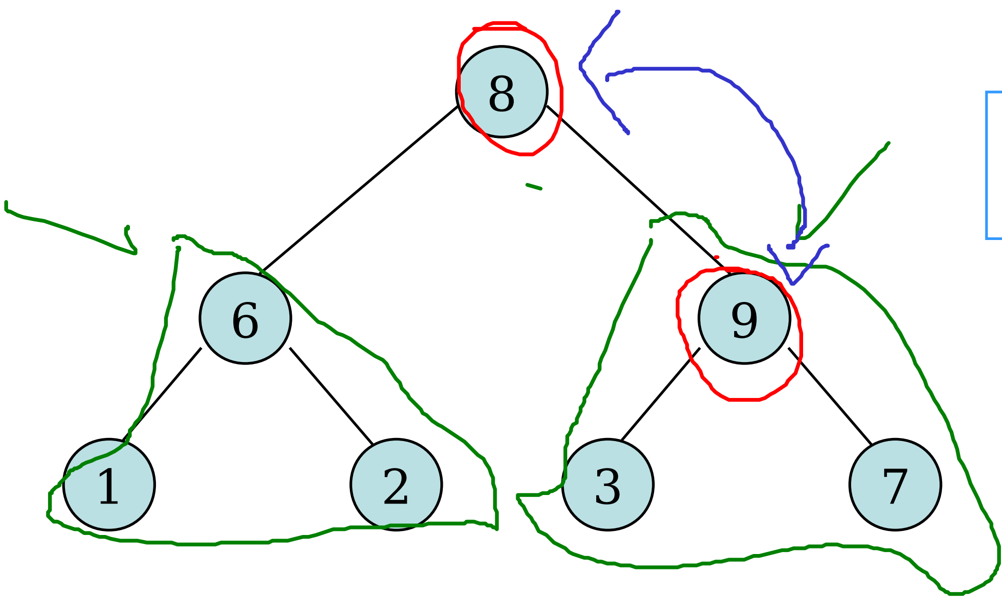


Heapsort

Heapify

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

Poscondición: subárbol con raíz es un montículo



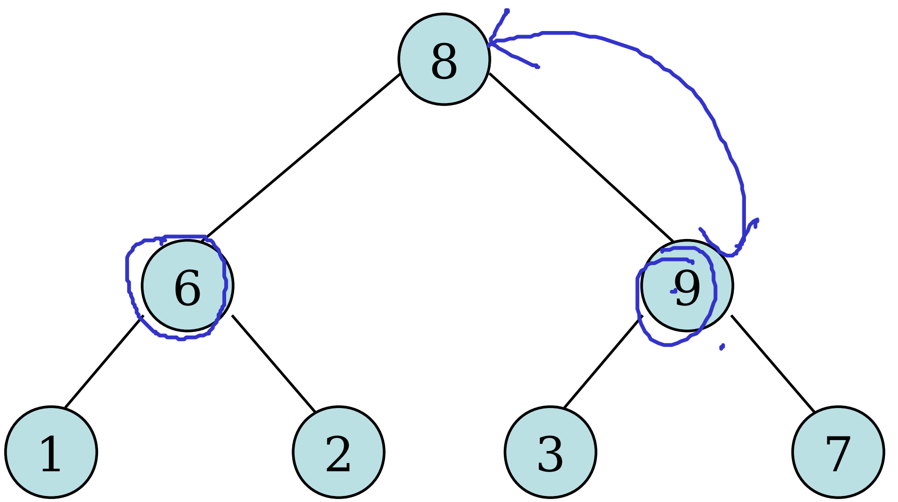
¿Cómo sería el montón resultante?

Heapsort

Heapify

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

Poscondición: subárbol con raíz es un montículo



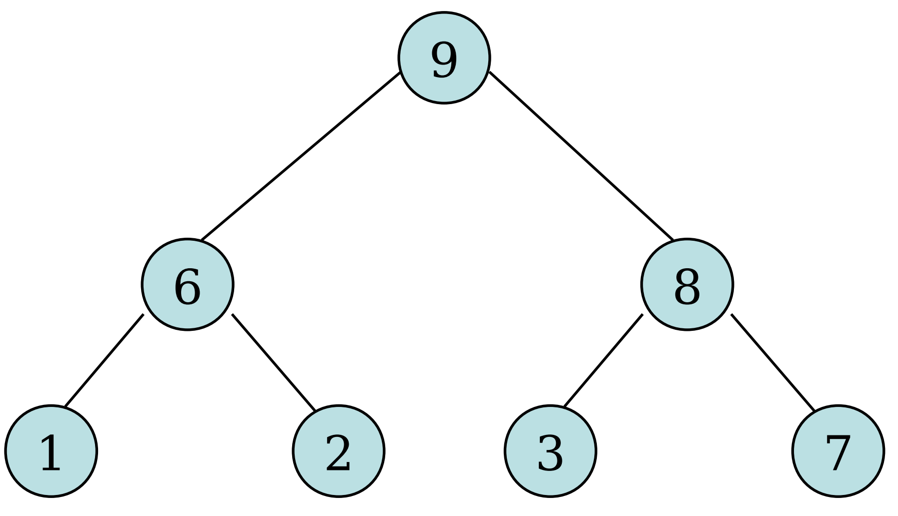
Se debe conocer cuál es el mayor entre la raíz Izq(i), la raiz Der(i) e A[i]

Heapsort

Heapify

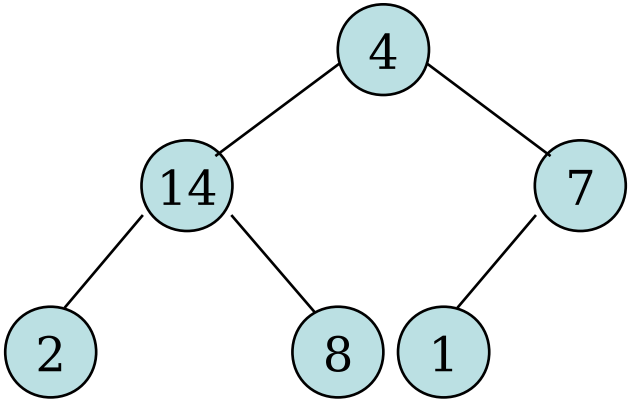
Precondición: subarbol con raíz Izq(i) y subarbol con raíz Der(i) son montículos

Poscondición: subárbol con raíz es un montículo

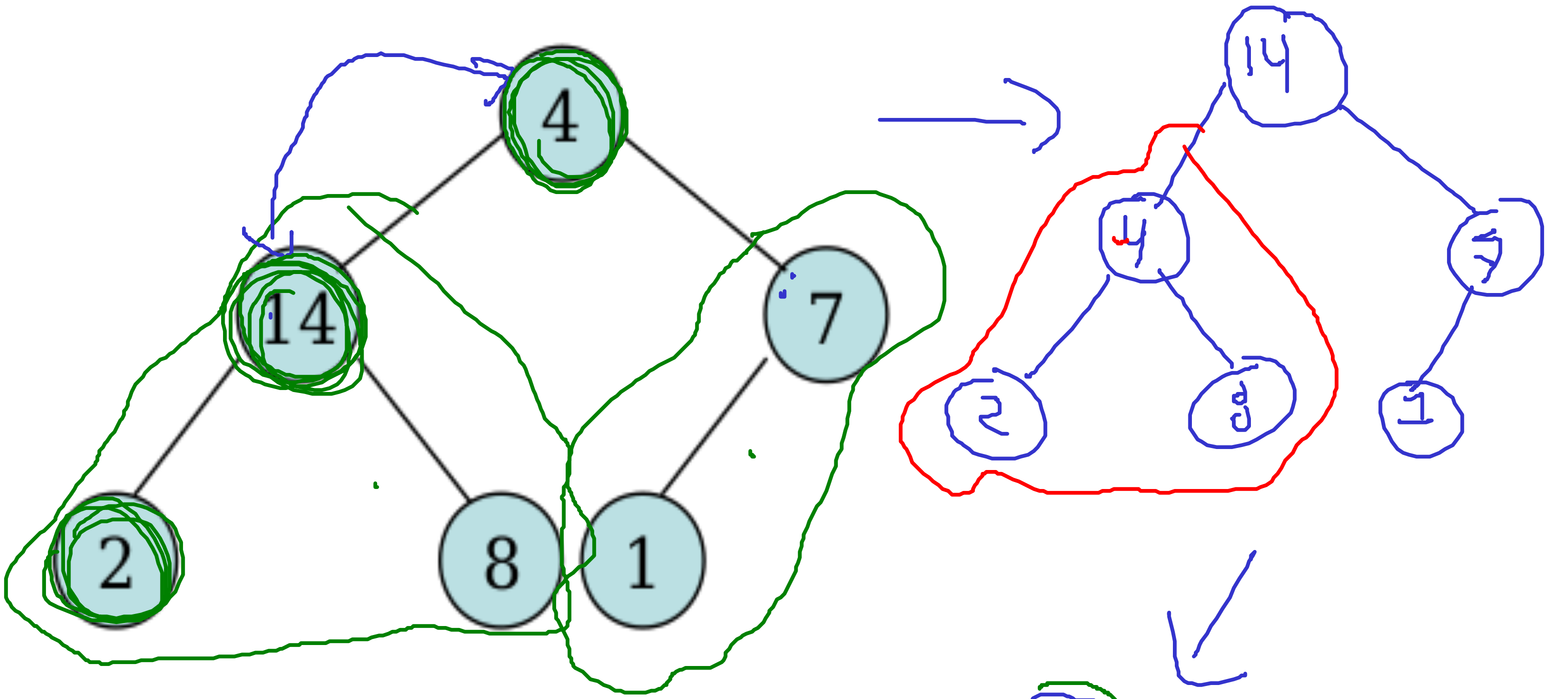


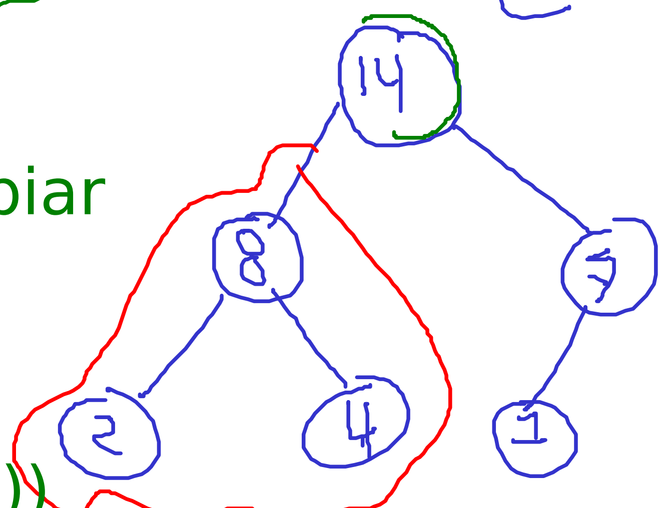
Al hacer el cambio de valores se debe verificar que el montón 3-8-7 cumpla la propiedad de orden

```
HEAPIFY(A, i)
 l←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)
```



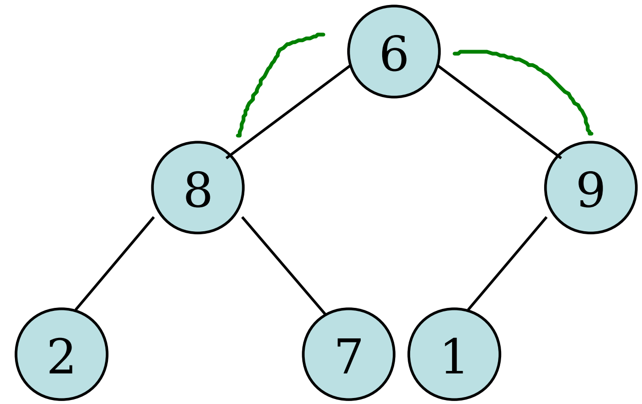
Aplique el algoritmo HEAPIFY(A, 1)



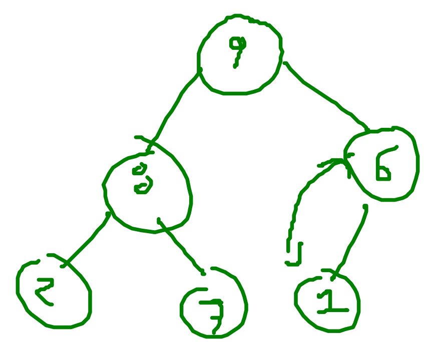


Comparar e intercambiar
O(1)
Esto lo repetimos
la altura del arbol
h = log(n) --> O(log(n))

```
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)
```

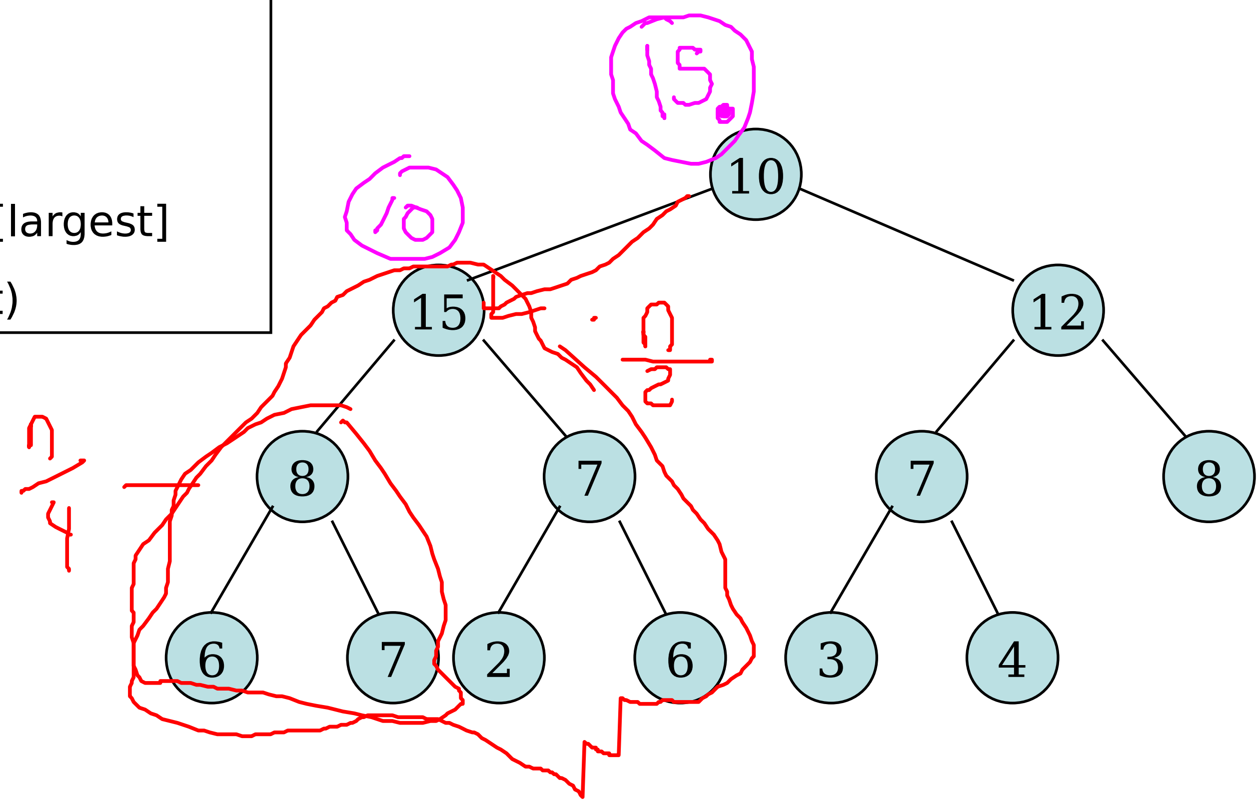


Aplique el algoritmo HEAPIFY(A, 1)



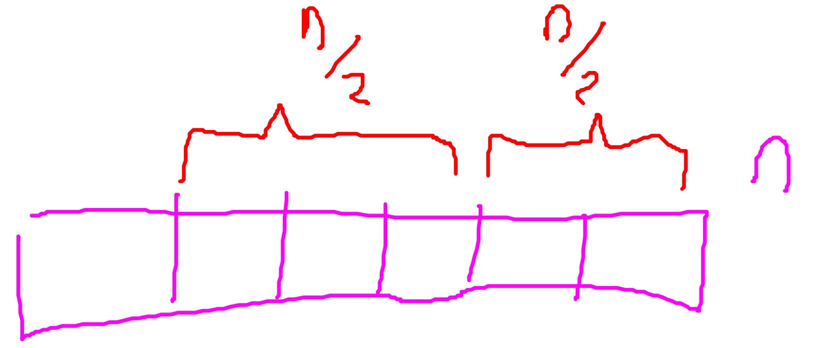
```
HEAPIFY(A, i)
 l←LEFT(i)
 r←RIGHT(i)
 if l≤heap-size[A] and A[l]>A[i]
     then largest←l
     else largest←i
 if r≤heap-size[A] and A[r]>A[largest]
    then largest←r
 if largest≠i
   then exchange A[i]↔A[largest]
         HEAPIFY(A,largest)
```

Aplique el algoritmo HEAPIFY(A, 1)



```
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)
```

¿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

$T(n) \le \underbrace{T(2n/3)}_{} + \Theta(1)$ ← Intercambio

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

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

$f(n)$

$T(n) = \Theta(\underline{n^{\log_b a}} \log(n))$

$T(n) = \Theta(\log(n))$

$T(n) = T(n/2) + \Theta(1)$ $\quad T(1) = \Theta(1)$

$a = 1 \quad b = 2 \quad \log_2 1 = 0$

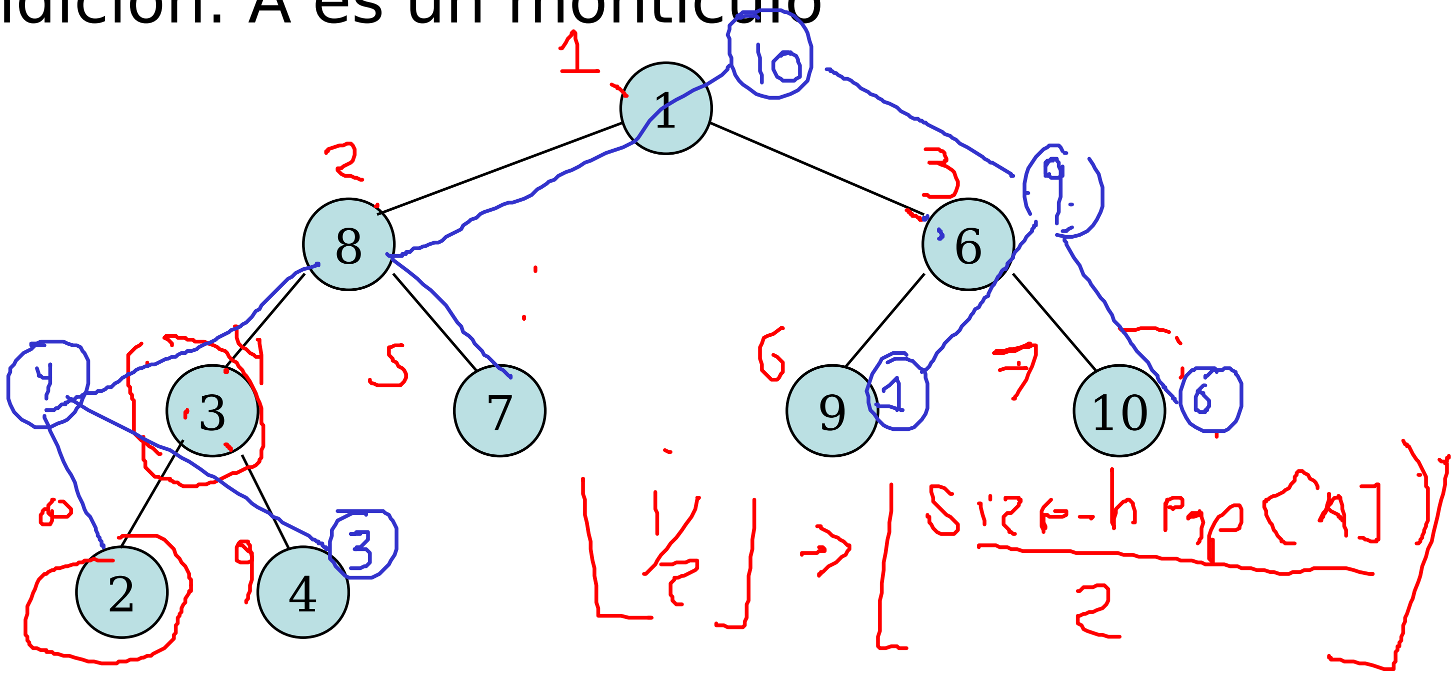
caso 2) $\Theta(n^0)$ es $\Theta(1)$

Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo



1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	8	6	3	7	9	10	2	4	

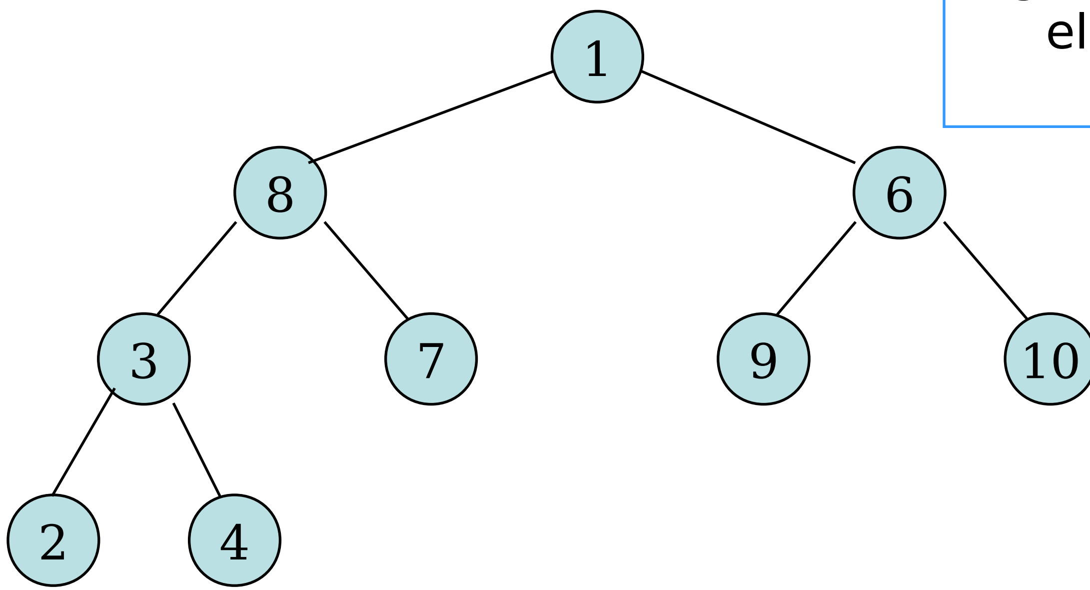
Heapsort

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



1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	8	6	3	7	9	10	2	4	

Heapsort

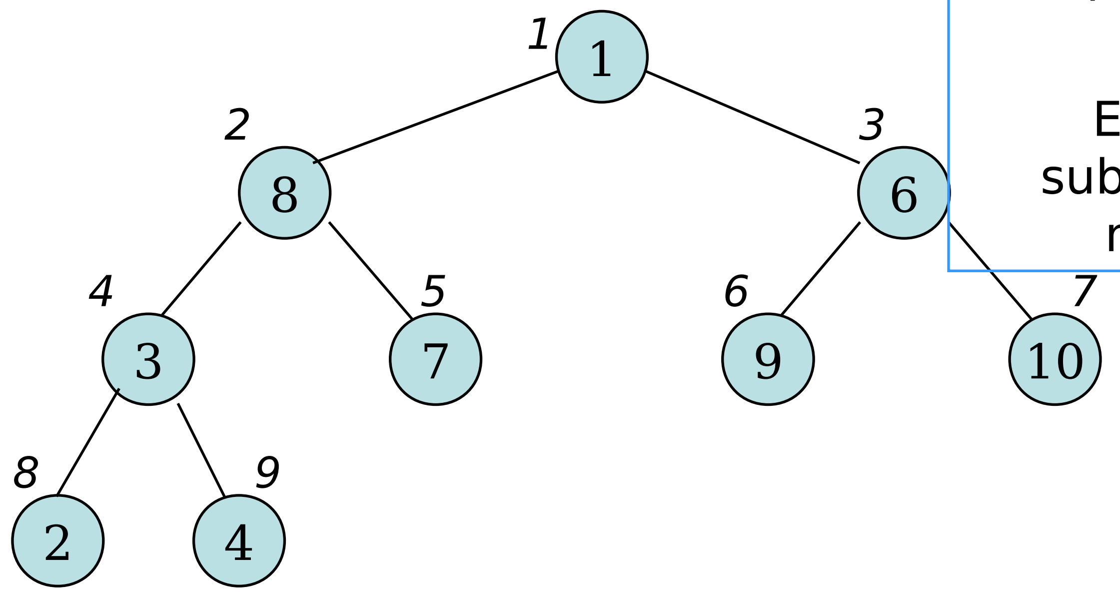
BUILD-HEAP(A)

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



1	2	3	4	5	6	7	8	9	10
1	8	6	3	7	9	10	2	4	

Heapsort

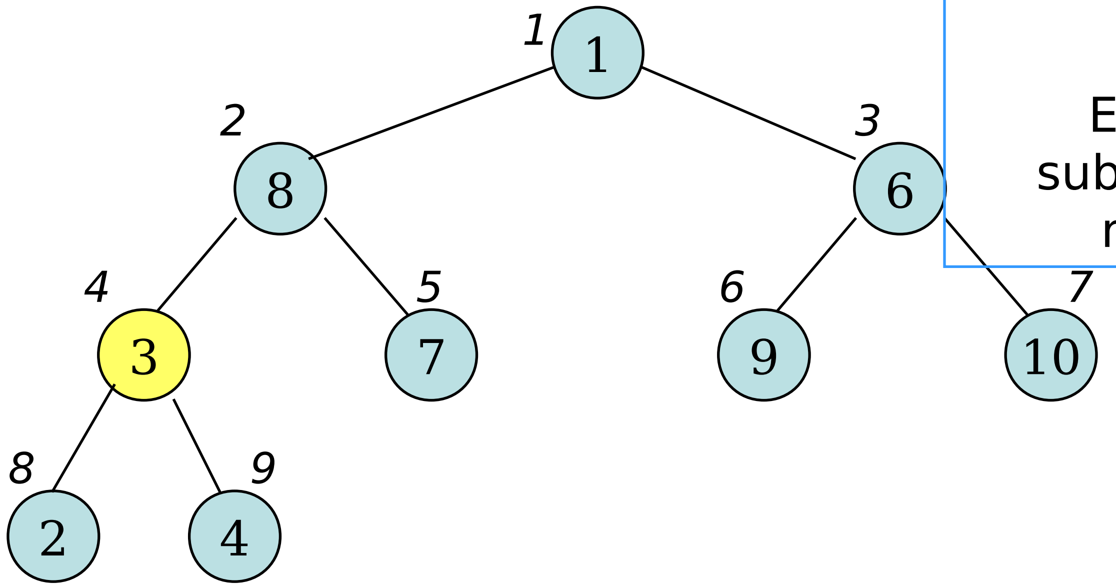
BUILD-HEAP(A)

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



1	2	3	4	5	6	7	8	9	10
1	8	6	3	7	9	10	2	4	

Heapsort

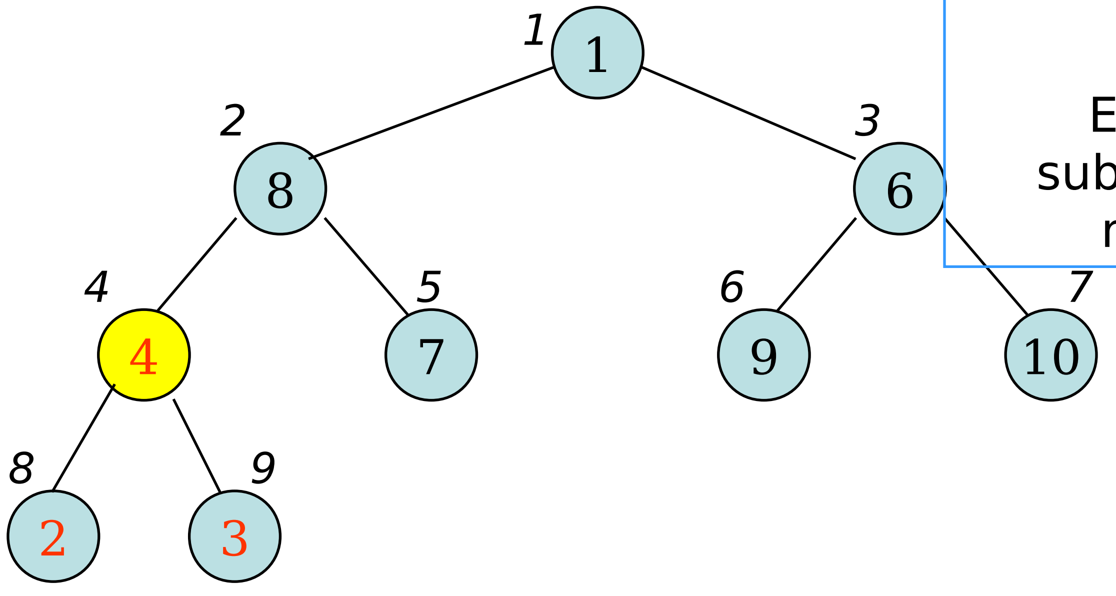
BUILD-HEAP(A)

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



1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	8	6	4	7	9	10	2	3	

Heapsort

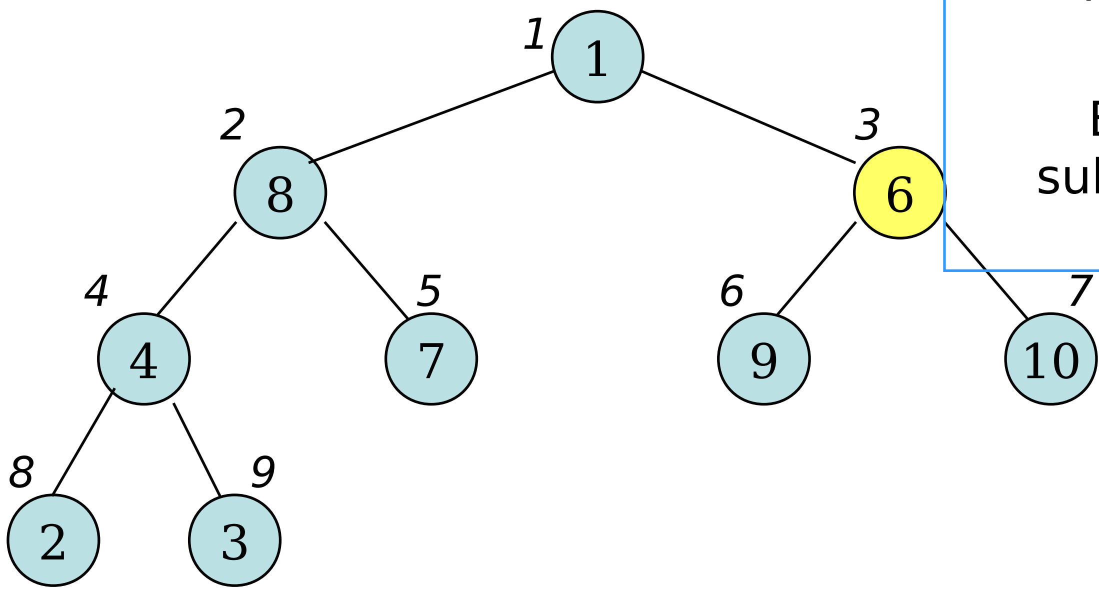
BUILD-HEAP(A)

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



1	2	3	4	5	6	7	8	9	10
1	8	6	4	7	9	10	2	3	

Heapsort

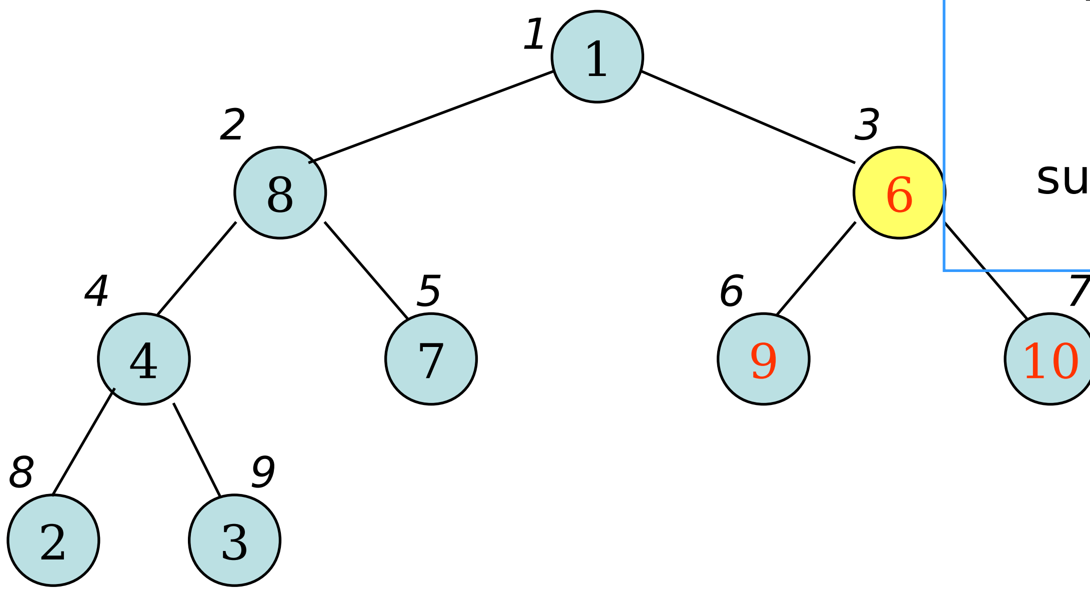
BUILD-HEAP(A)

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



1	2	3	4	5	6	7	8	9	10
1	8	6	4	7	9	10	2	3	

Heapsort

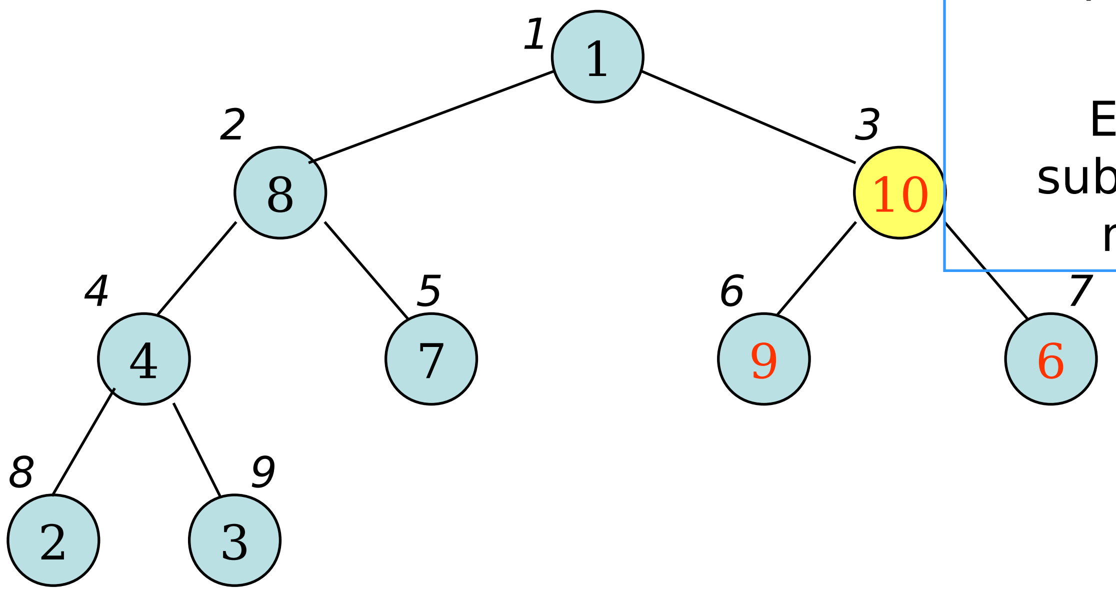
BUILD-HEAP(A)

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



1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	6	2	3	

Heapsort

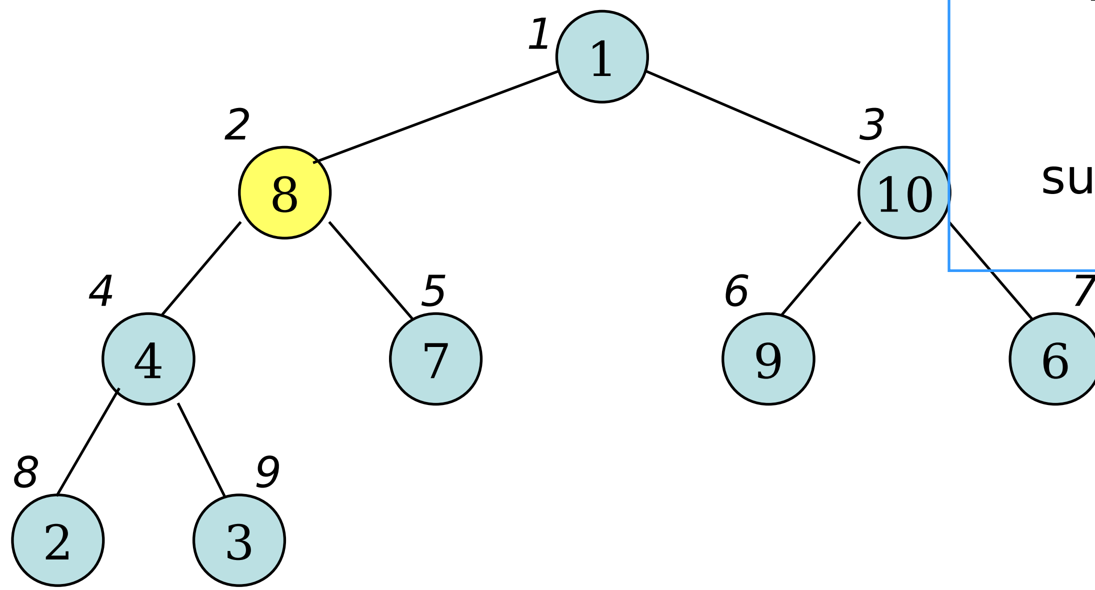
BUILD-HEAP(A)

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



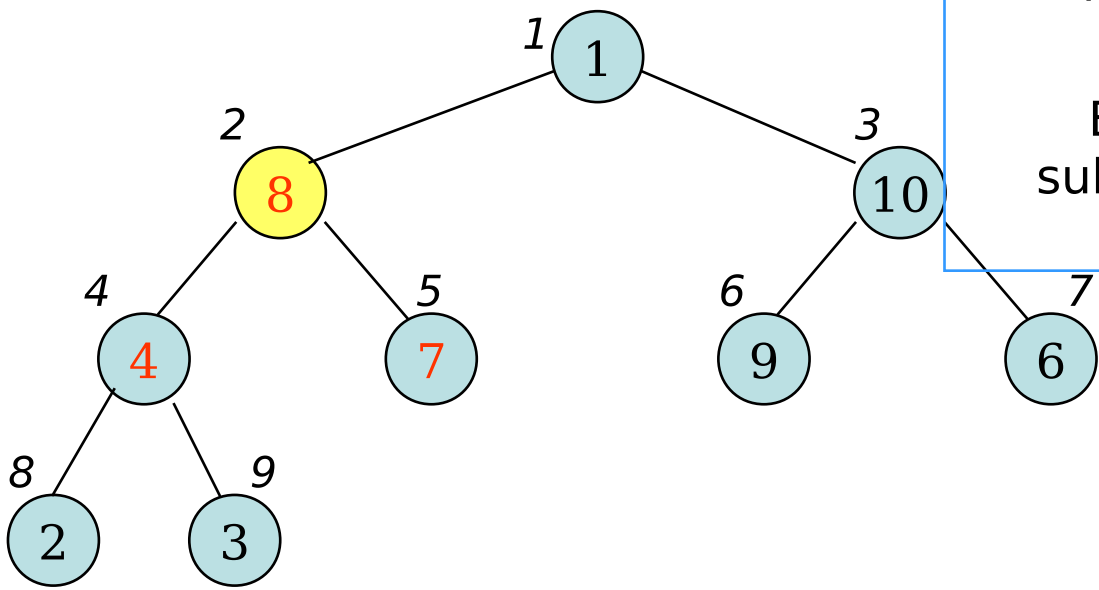
1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	6	2	3	

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

1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	6	2	3	

Heapsort

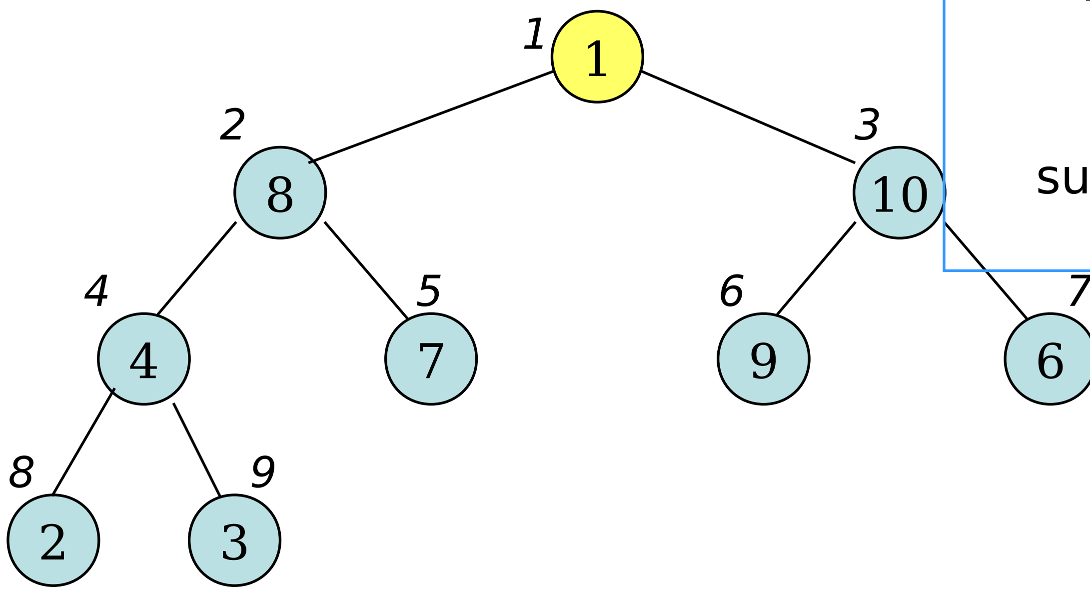
BUILD-HEAP(A)

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



1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	6	2	3	

Heapsort

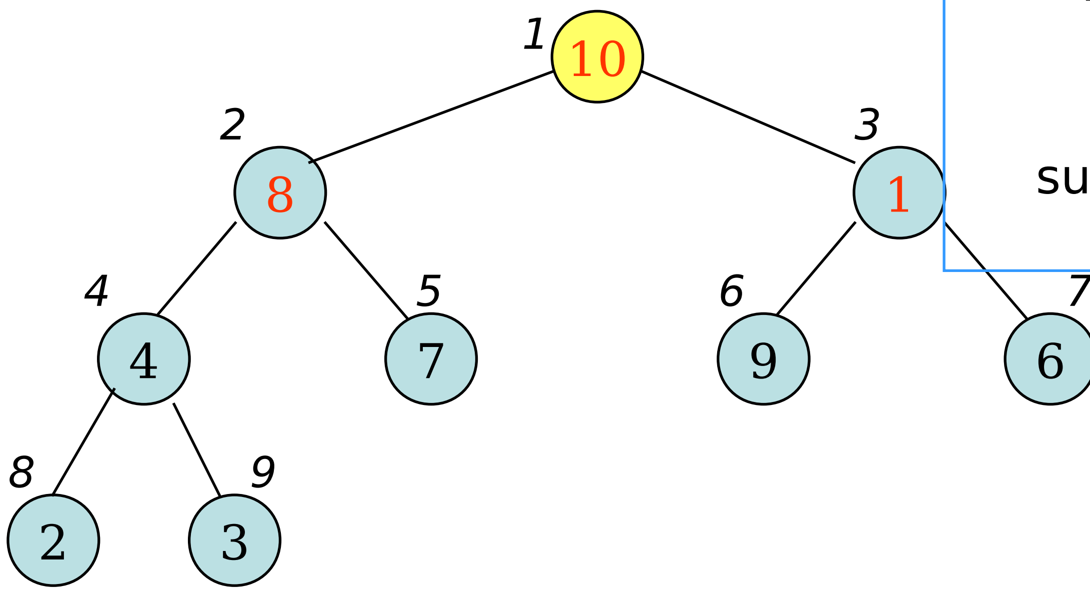
BUILD-HEAP(A)

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



1	2	3	4	5	6	7	8	9	10
10	8	1	4	7	9	6	2	3	

Heapsort

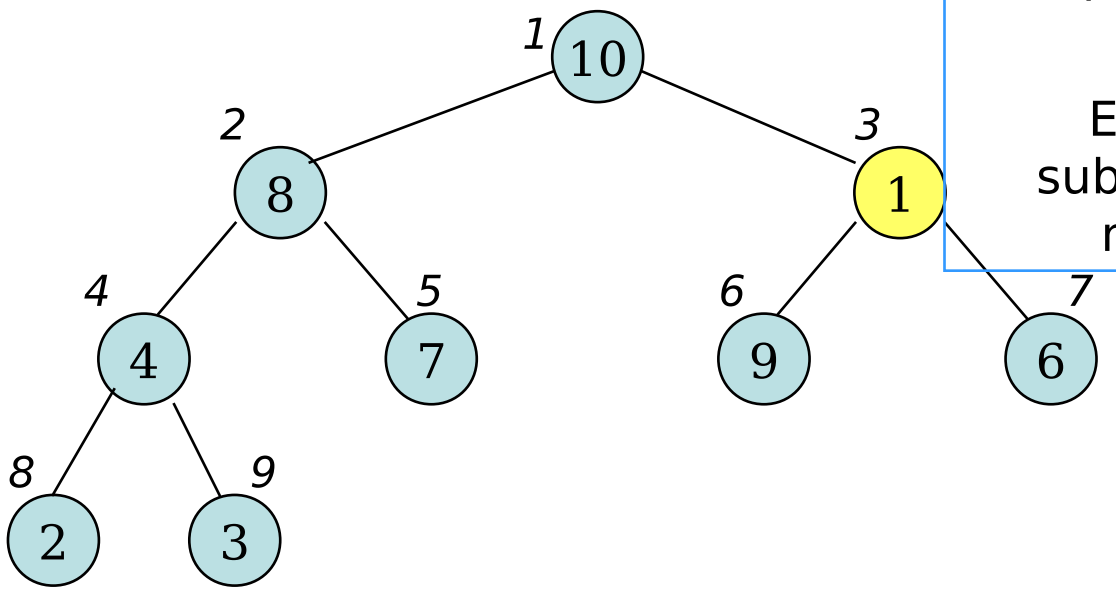
BUILD-HEAP(A)

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



1 10	2	3	4	5	6	7	8	9	
10	8	1	4	7	9	6	2	3	

Heapsort

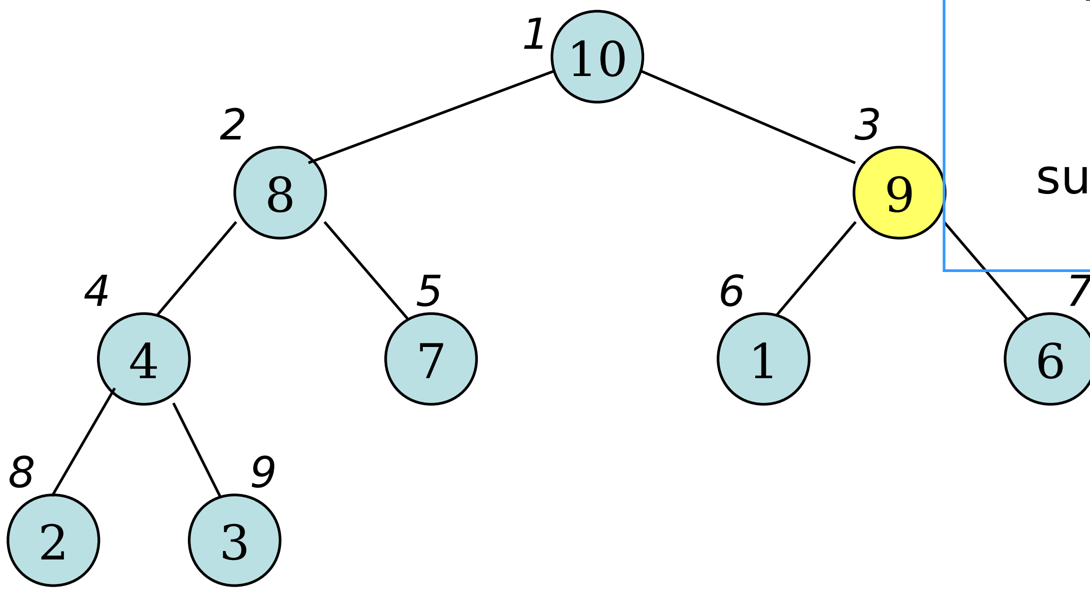
BUILD-HEAP(A)

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



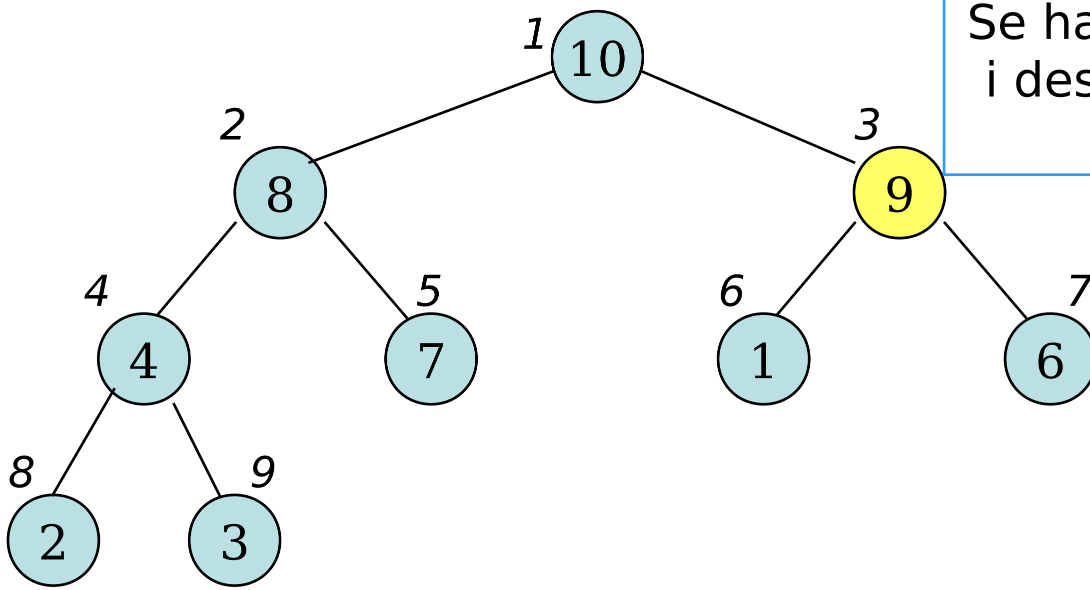
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	6	2	3	

Heapsort

BUILD-HEAP(A)

Precondición: A es un arreglo de elementos

Poscondición: A es un montículo



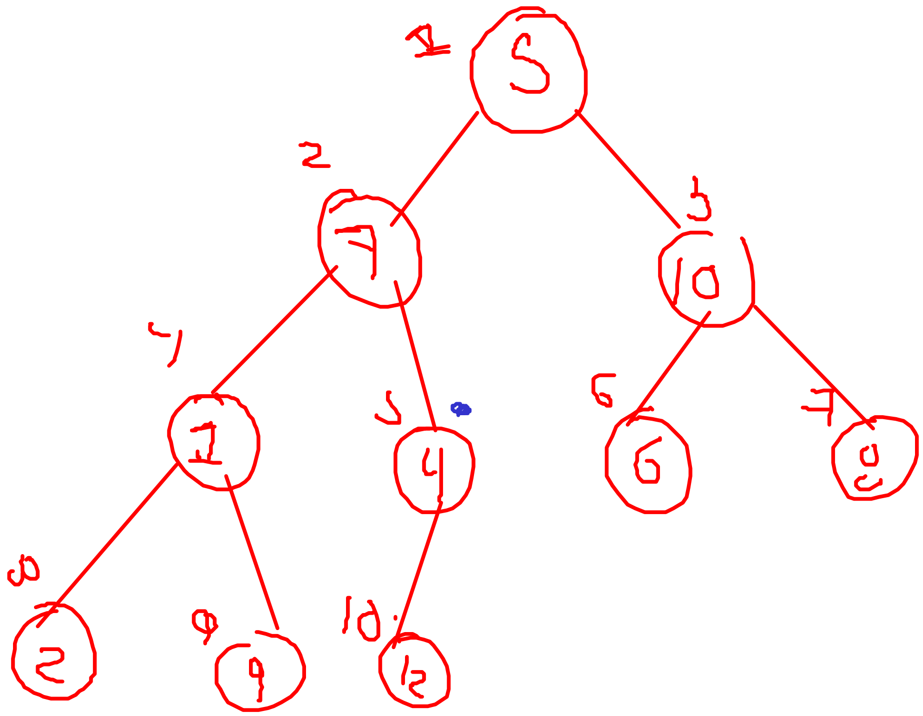
Se hace HEAPIFY para i desde $\lfloor length[A]/2 \rfloor$ hasta 1

1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	6	2	3	

```
BUILD-HEAP(A)

 heap-size[A]←length[A]

 for i←⌊length[A]/2⌋ downto
1

    do HEAPIFY(A,i)
```

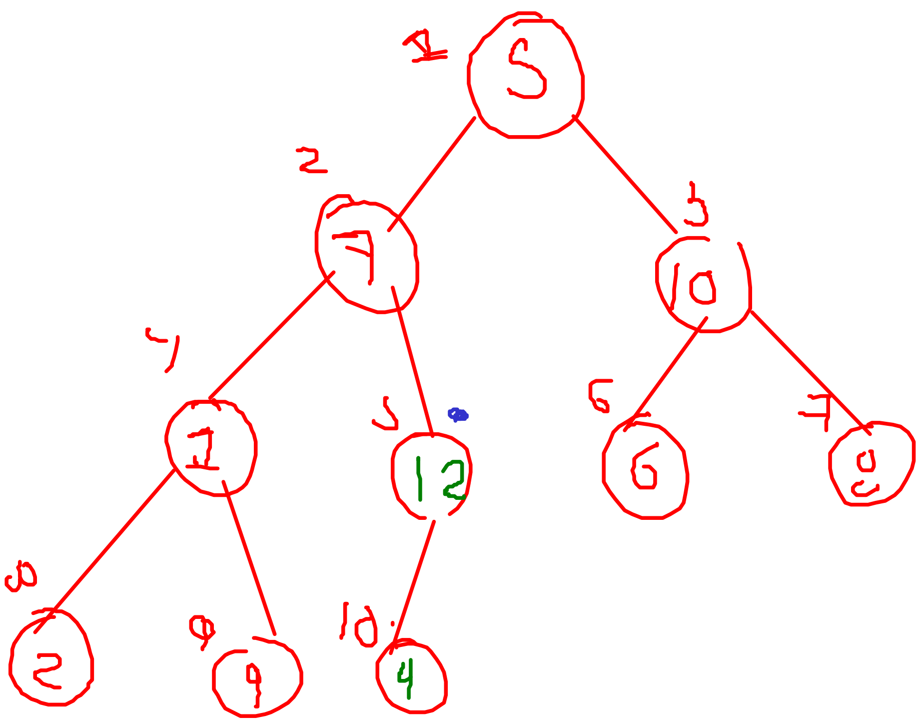
Aplique el algoritmo BUILD-HEAP(A), para

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

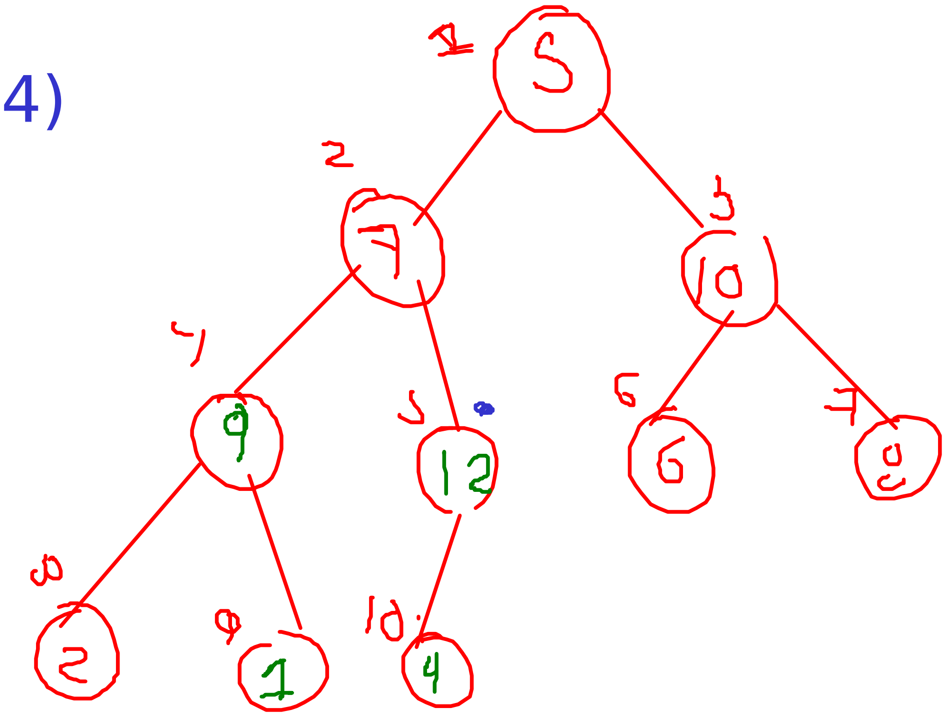


$$\left\lfloor \frac{10}{2} \right\rfloor = 5$$

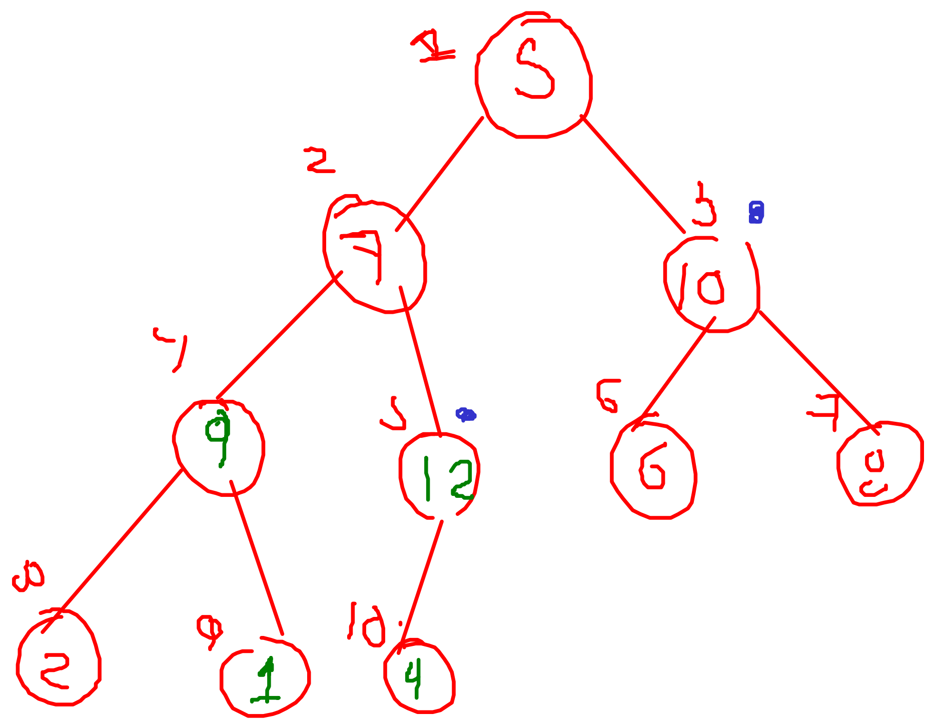
heapify(A,5)



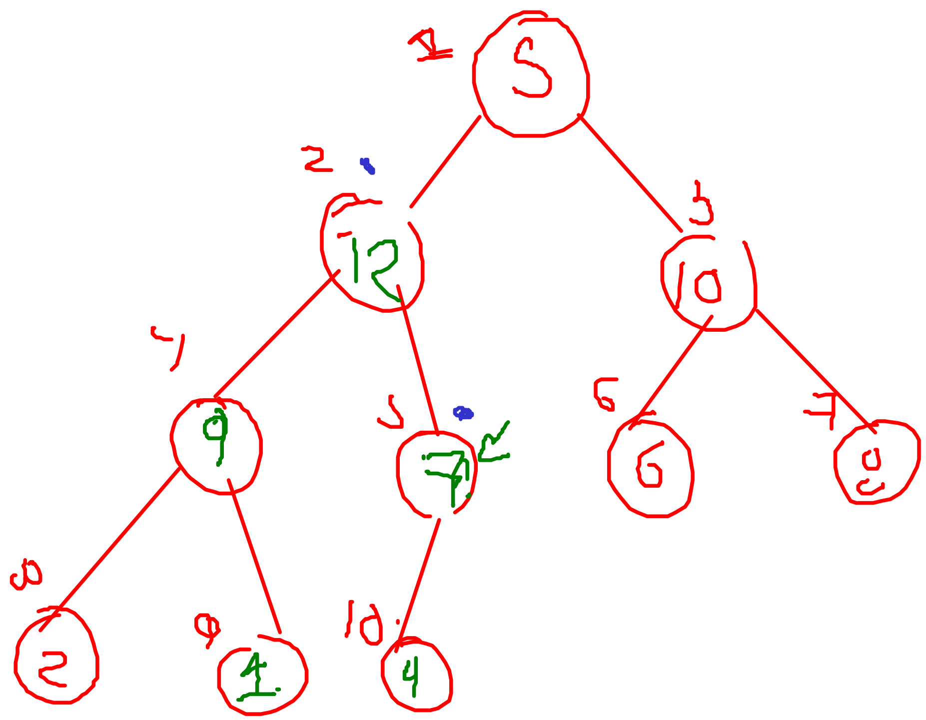
heapify(A,4)



heapify(A,3)

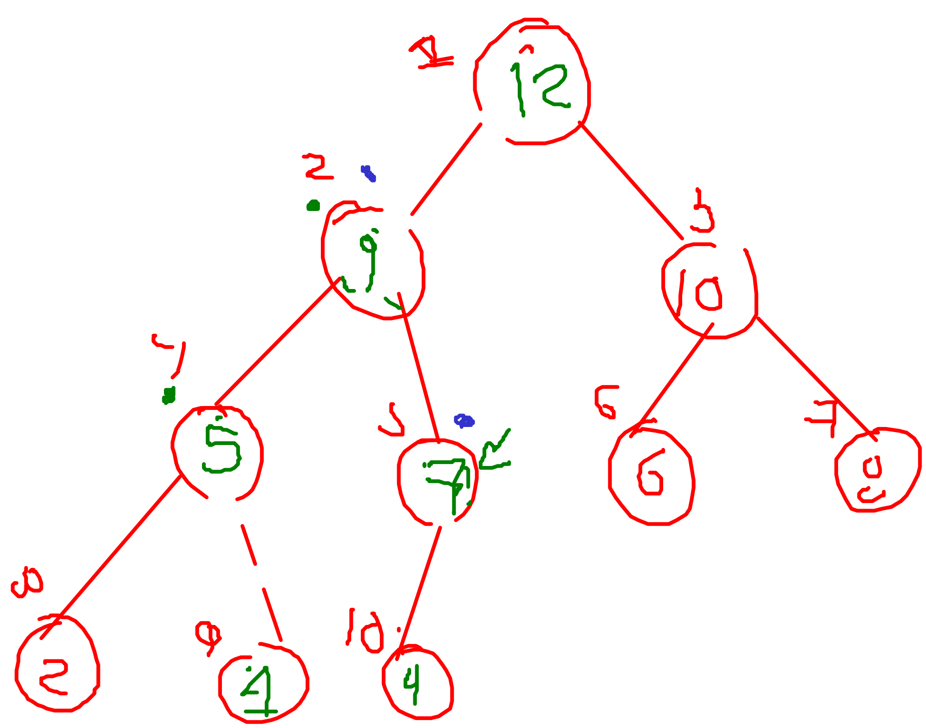


heapfy(A,2)



heapify(A,1)

$\lfloor n/2 \rfloor$



BUILD-HEAP(A)

```
heap-size[A]←length[A]
for i←⌊length[A]/2⌋ downto 1
    do HEAPIFY(A,i)
```

$n/2$

$O(\log(n))$

$\frac{n}{2}\log(n)$

$O(n\log(n))$

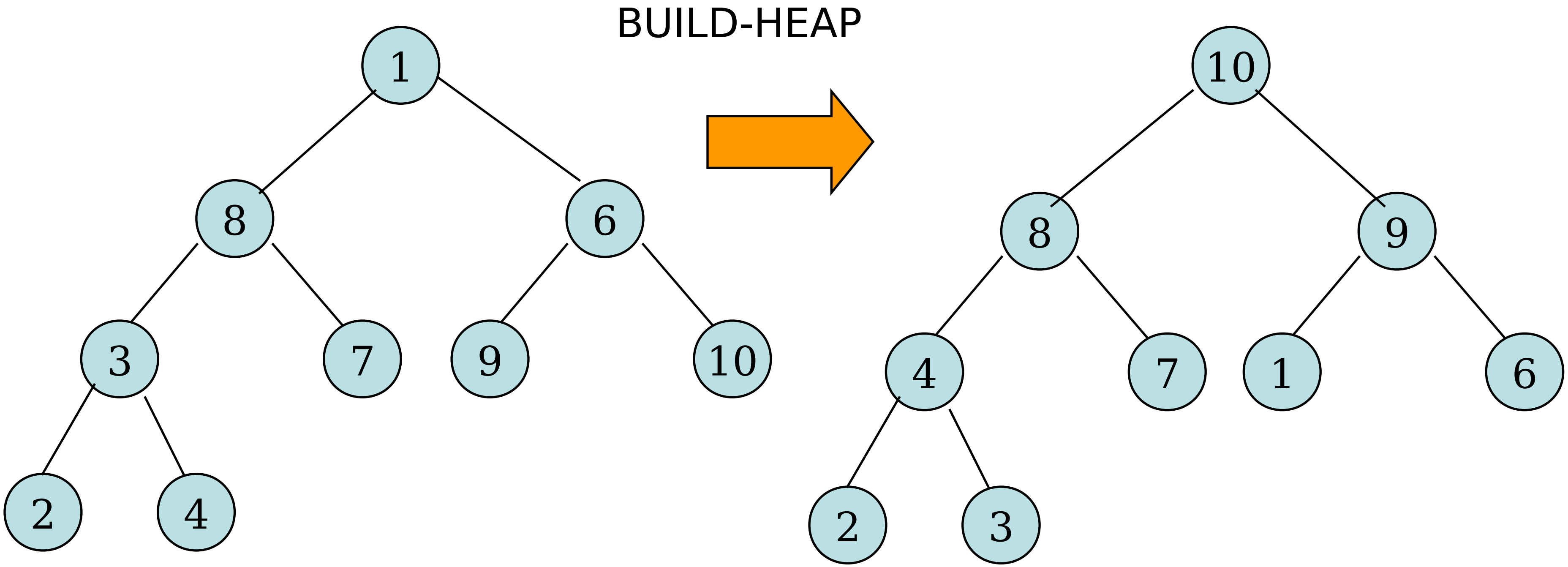
Complejidad

- Cada llamado a HEAPIFY cuesta O(lgn)
- Se hacen O(n) llamados
- Estimacion: O(nlgn)

  -O(n) es una estimación más precisa

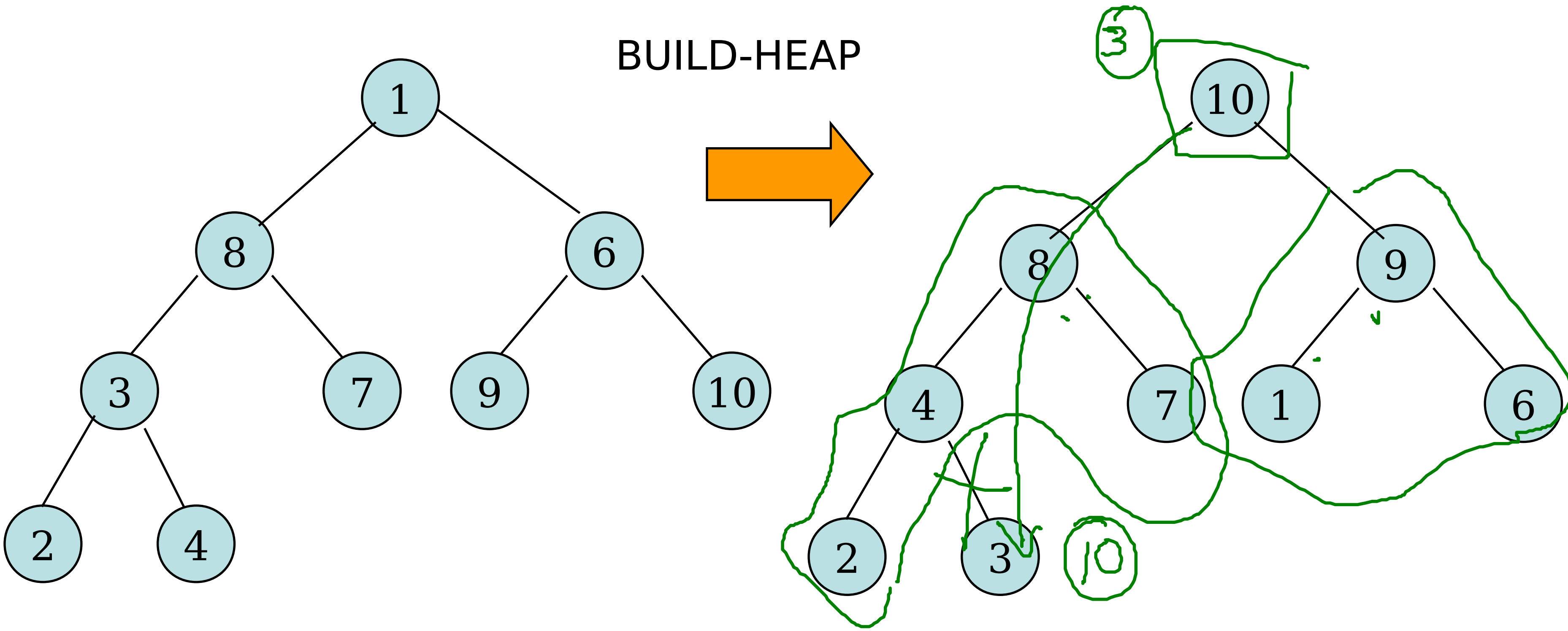
Heapsort

HEAP-SORT(A)



Heapsort

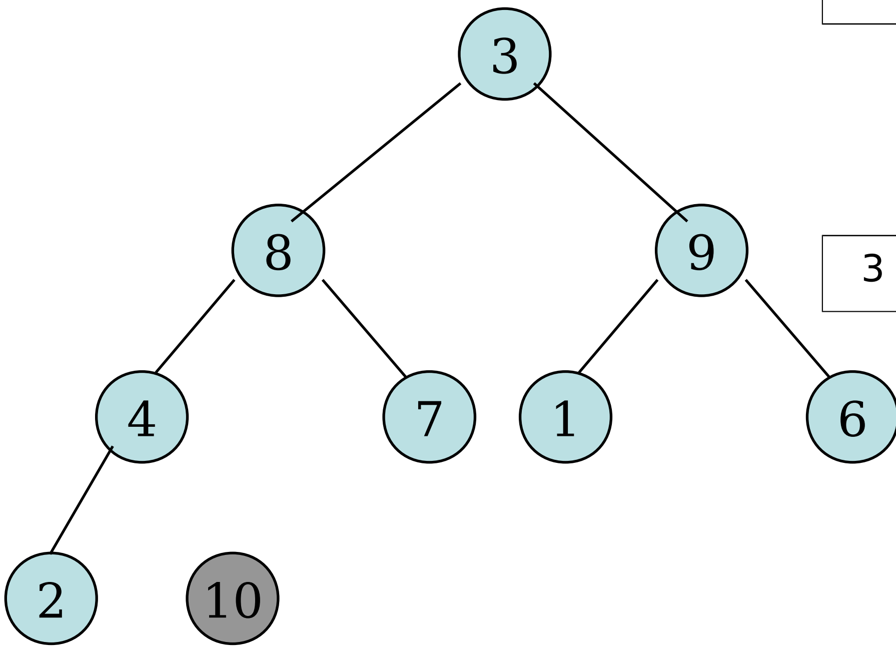
HEAP-SORT(A)



El valor más grande quedará en la raíz del árbol

Heapsort

HEAP-SORT(A)



10	8	9	4	7	1	6	2	3	

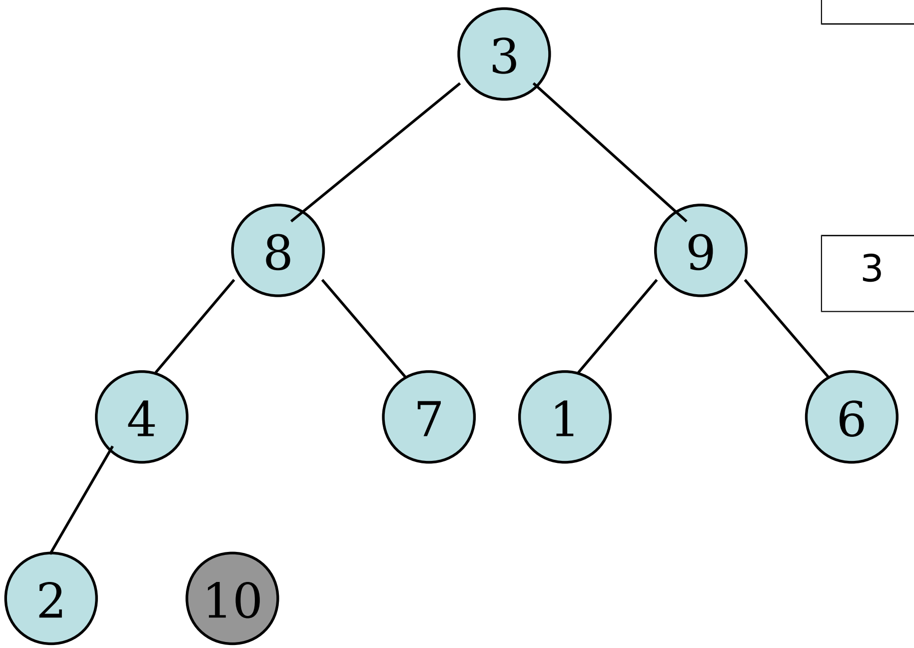
3	8	9	4	7	1	6	2	10	

heap-size[A]=8

Se intercambio el valor A[1], el mayor, con el valor A[heap-size[A]] y se disminuye en 1 valor heap-size[A]

Heapsort

HEAP-SORT(A)



10	8	9	4	7	1	6	2	3	

3	8	9	4	7	1	6	2	10	

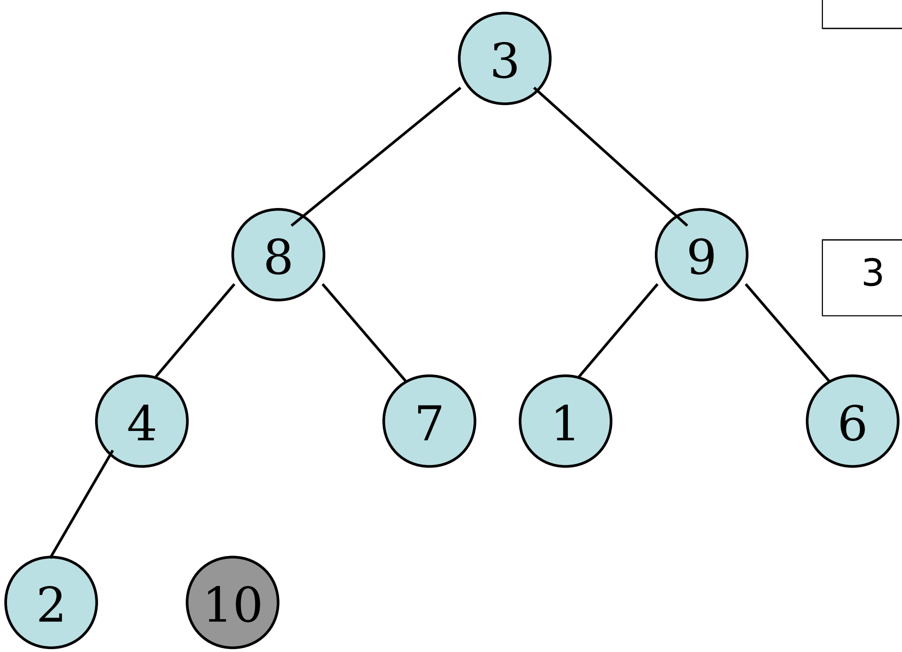
heap-size[A]=8

Se intercambio el valor A[1], el mayor, con el valor A[heap-size[A]] y se disminuye en 1 valor heap-size[A]

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	

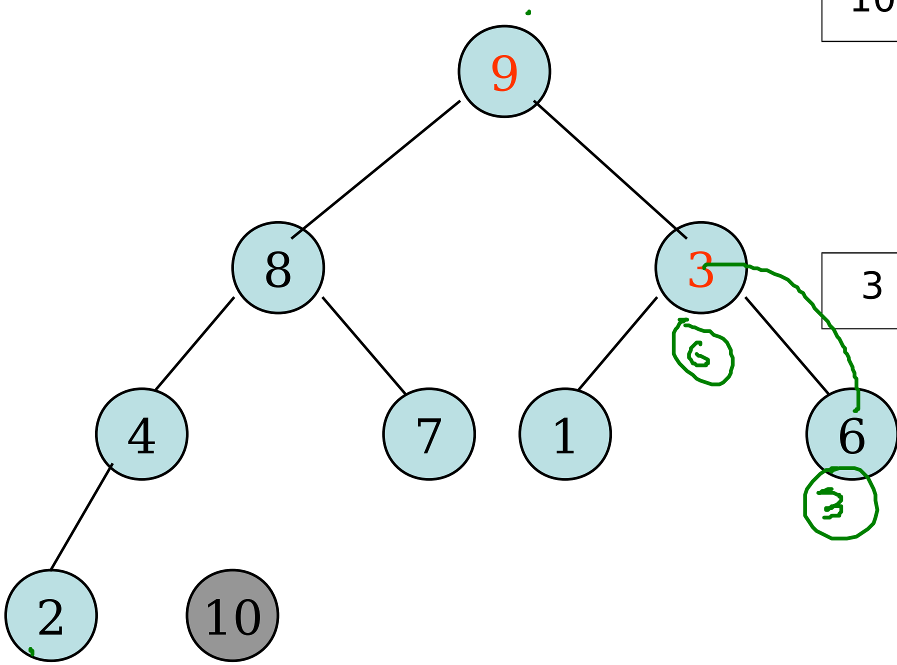
3	8	9	4	7	1	6	2	10	

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	

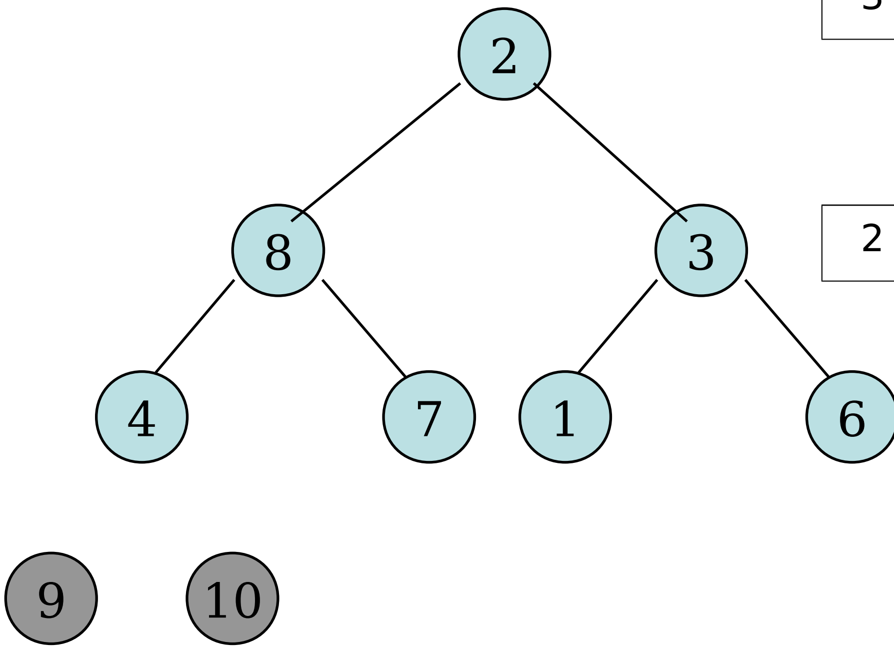
3	8	9	4	7	1	6	2	10	

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	

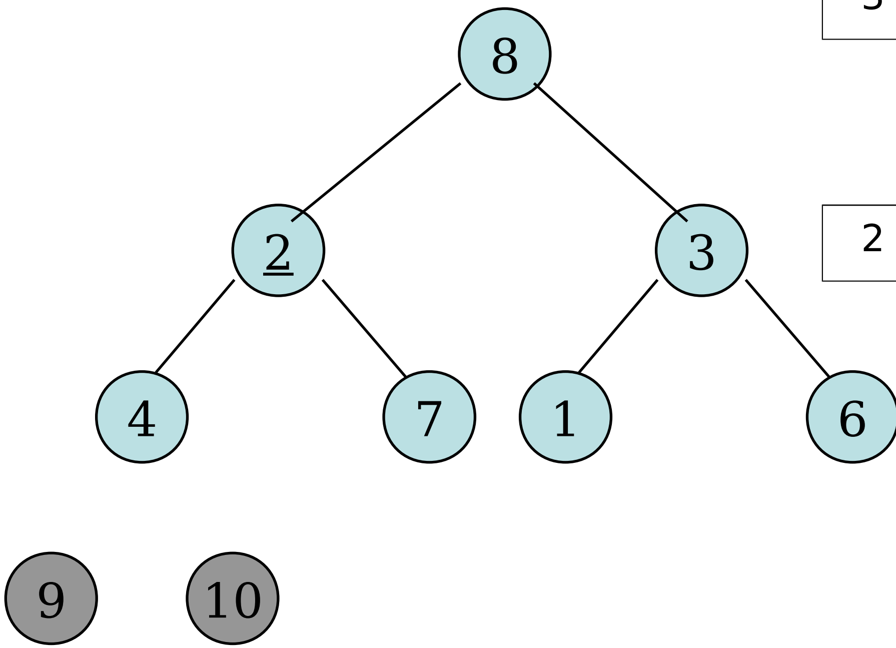
2	8	3	4	7	1	6	9	10	

heap-size[A]=7

Se repite el proceso de hacer 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	

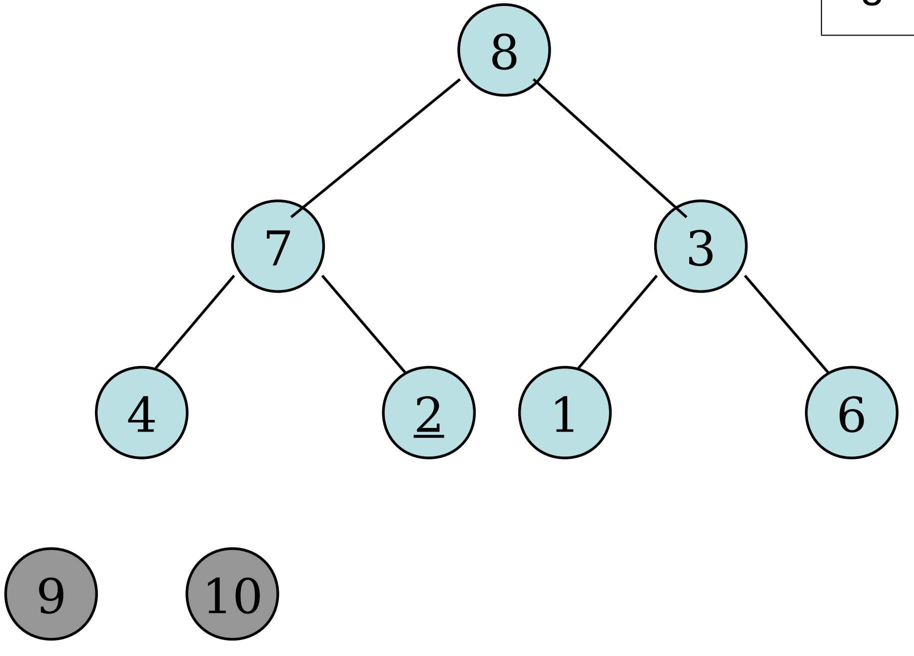
heap-size[A]=7

Se repite el proceso de hacer 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	



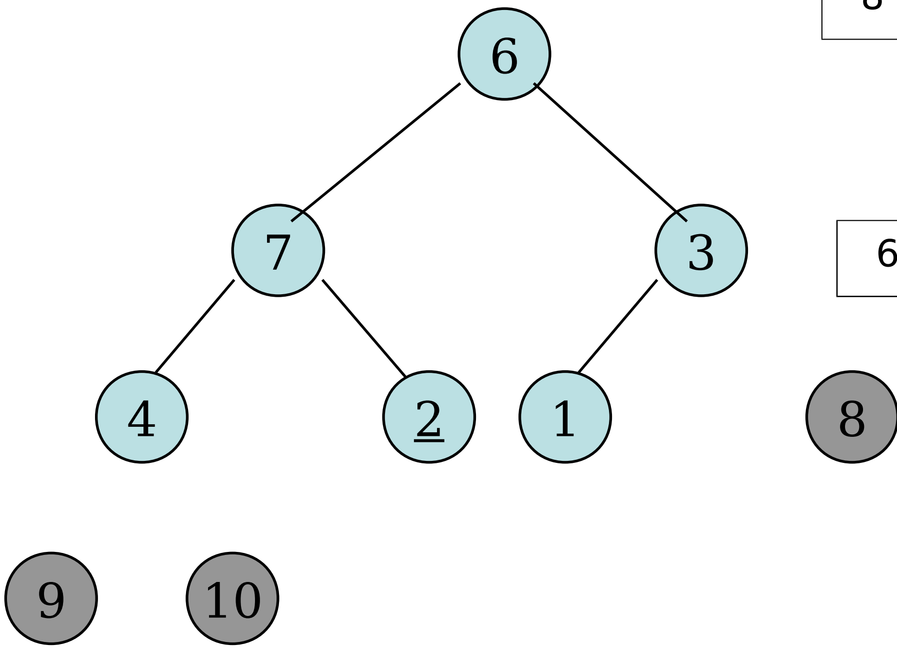
Se repite el proceso de hacer heapify(A,1) para que el mayor valor quede en la raíz, intercambiar y disminuir el tamaño del montón

Heapsort

HEAP-SORT(A)

8	7	3	4	2	1	6	9	10	

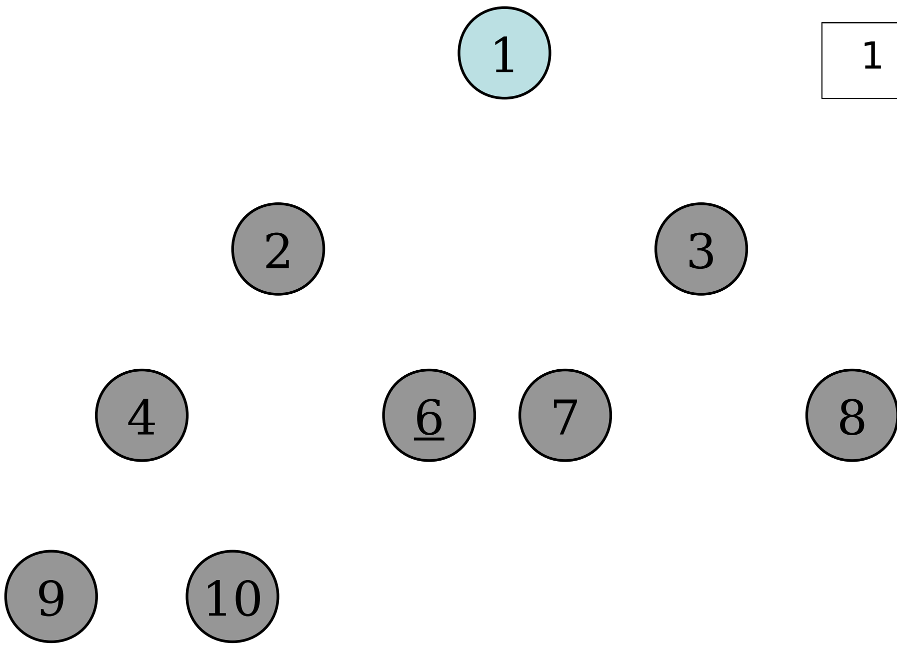
6	7	3	4	2	1	8	9	10	



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

Heapsort

HEAP-SORT(A)



1	2	3	4	6	7	8	9	10	

```
HEAP-SORT(A)
 BUILD-HEAP(A)
 for i←length[A] downto 2
    do exchange A[1]↔A[i]
       heap-size[A]← heap-size[A] -1
       HEAPIFY(A,1)
```

n log(n)

n log(n)

log(n)

O(n log(n))

Aplique el algoritmo HEAP-SORT(A), para

- A={12, 9, 10, 7, 8, 1} y heap-size(A)=6

BUILD-HEAP(A)

1 2 3 4 5 6

12, 9, 10, 7, 8, 1

Heapify(A,3) $p(i) = \lfloor i/2 \rfloor$

1 2 3 4 5 6

12, 9, 10, 7, 8, 1

Heapify(A,2) $H_{izq} = 2i$

$H_{der} = 2i+1$

1 2 3 4 5 6

12, 9, 10, 7, 8, 1

Heapify(A,1)

for i = Heapsize(A) downto 1
    A[i] <-> A[1]
    Heapsize[A]-- //Reducimos en 1
    Heapify(A,1)

1 2 3 4 5 6

1, 9, 10, 7, 8 | 12

Heapify(A,1)

10, 9, 1, 7, 8 | 12

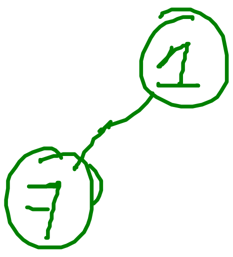
8, 9, 1, 7 | 10, 12

9, 8, 1, 7 | 10, 12

7, 8, 1 | 9, 10, 12

8, 7, 1 | 9, 10, 12

1, 7 | 8, 9, 10, 12



7, 1 | 8, 9, 10, 12

1 | 7, 8, 9, 10, 12

1, 7, 8, 9, 10, 12

```
HEAP-SORT(A)
 BUILD-HEAP(A)
 for i←length[A] downto 2
    do exchange A[1]↔A[i]
        heap-size[A]← 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

1 2 3 4 5 10

{5, 7, 10, 7, 4, 6, 8, 2, 9, 12}

S = 10

$\lfloor \frac{S}{2} \rfloor = 5$

BUILD-HEAP

heapify(A,5)

4 8 9

{5, 7, 10, 7, 12, 6, 8, 2, 9, 4}

heapify(A,4)

{5, 7, 10, 9, 12, 6, 8, 2, 7, 4}

heapify(A,3)

3 6 7

{5, 7, 10, 9, 12 6 8, 2, 7 4}

heapify(A,2)

{5, 7, 10, 9, 12, 6, 8, 2, 1, 4}

{5, 12, 10, 9, 7, 6, 8, 2, 1, 4}

heapify(A,1)

{5, 12, 10, 9, 7, 6, 8, 2, 1, 4}

{12, 5, 10, 9, 7, 6, 8, 2, 1, 4}

{12, 9, 10, 5, 7, 6, 8, 2, 1, 4}

Ciclo

{4, 9, 10, 5, 7, 6, 8, 2, 1 | 12} S = 9

1

{10, 9, 4, 5, 7, 6, 8, 2, 1 | 12}

{10, 9, 8, 5, 7, 6, 4, 2, 1 | 12}

2

{1, 9, 8, 5, 7, 6, 4, 2 | 10, 12}

{9, 1, 8, 5, 7, 6, 4, 2 | 10, 12}

{9, 7, 8, 5, 1, 6, 4, 2 | 10, 12}

(3)

$\{2, 7, 8, 5, 1, 6, 4 \mid 9, 10, 12\}$

$\{8, 7, 2, 5, 1, 6, 4 \mid 9, 10, 12\}$

$\{8, 7, 6, 5, 1, 2, 4 \mid 9, 10, 12\}$

(4)

$\{4, 7, 6, 5, 1, 2 \mid 8, 9, 10, 12\}$

$\{7, 4, 6, 5, 1, 2 \mid 8, 9, 10, 12\}$

$\{7, 5, 6, 4, 1, 2 \mid 8, 9, 10, 12\}$

$2i$

$2i+1$ > 3, 2

No ~~e~~ extra, stm

⑤ {2, 5, 6, 4, 1 | 7, 8, 9, 10, 12}

{6, 5, 2, 4, 1 | 7, 8, 9, 10, 12}

⑥ {1, 5, 2, 4 | 6, 7, 8, 9, 10, 12}

{5, 1, 2, 4 | 6, 7, 8, 9, 10, 12}

{5, 4, 2, 1 | 6, 7, 8, 9, 10, 12}

⑦ $\{1, 4, 2 \mid 5, 6, 7, 8, 9, 10, 12\}$

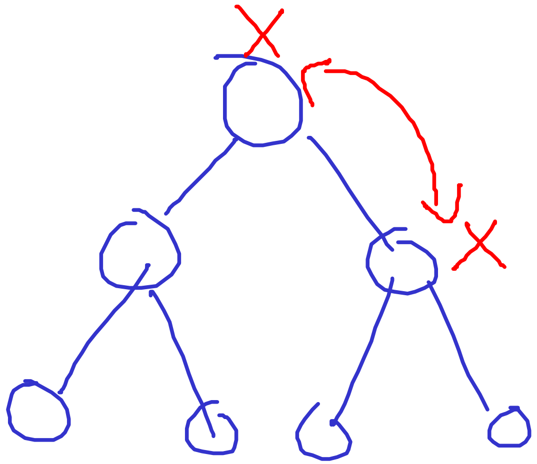
$\{4, 1, 2 \mid 5, 6, 7, 8, 9, 10, 12\}$

⑧ $\{2, 1 \mid 4, 5, 6, 7, 8, 9, 10, 12\}$

⑨ $\{1 \mid 2, 4, 5, 6, 7, 8, 9, 10, 12\}$

⑩ $\{1, 2, 4, 5, 6, 7, 8, 9, 10, 12\}$ ✓

```
HEAP-SORT(A)
 BUILD-HEAP(A)
 for i←length[A] downto 2
    do exchange A[1]↔A[i]
        heap-size[A]← heap-size[A] -1
        HEAPIFY(A,1)
```



¿Cuál es la complejidad?

```
HEAP-SORT(A)
 BUILD-HEAP(A)
 for i←length[A] downto 2
    do exchange A[1]↔A[i]
         heap-size[A]← 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(lgn)
- La complejidad es de O(nlgn)

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 → $\Theta(1)$

  Primero

  -EXTRACT-MAX(C): elimina y devuelve el elemento de máxima prioridad → $\Theta(1)$

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]]

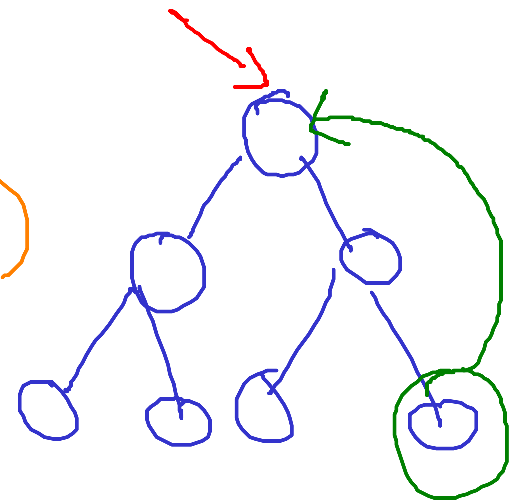
  heap-size[A]←heap-size[A]-1

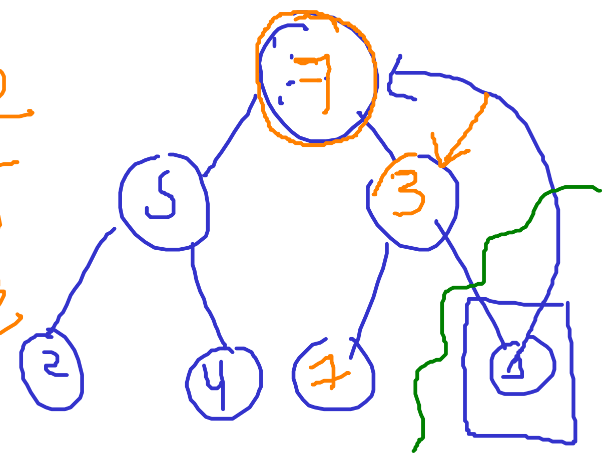
  HEAPIFY(A,1)

  return max

$\Theta(1)$

$O(\lg n)$





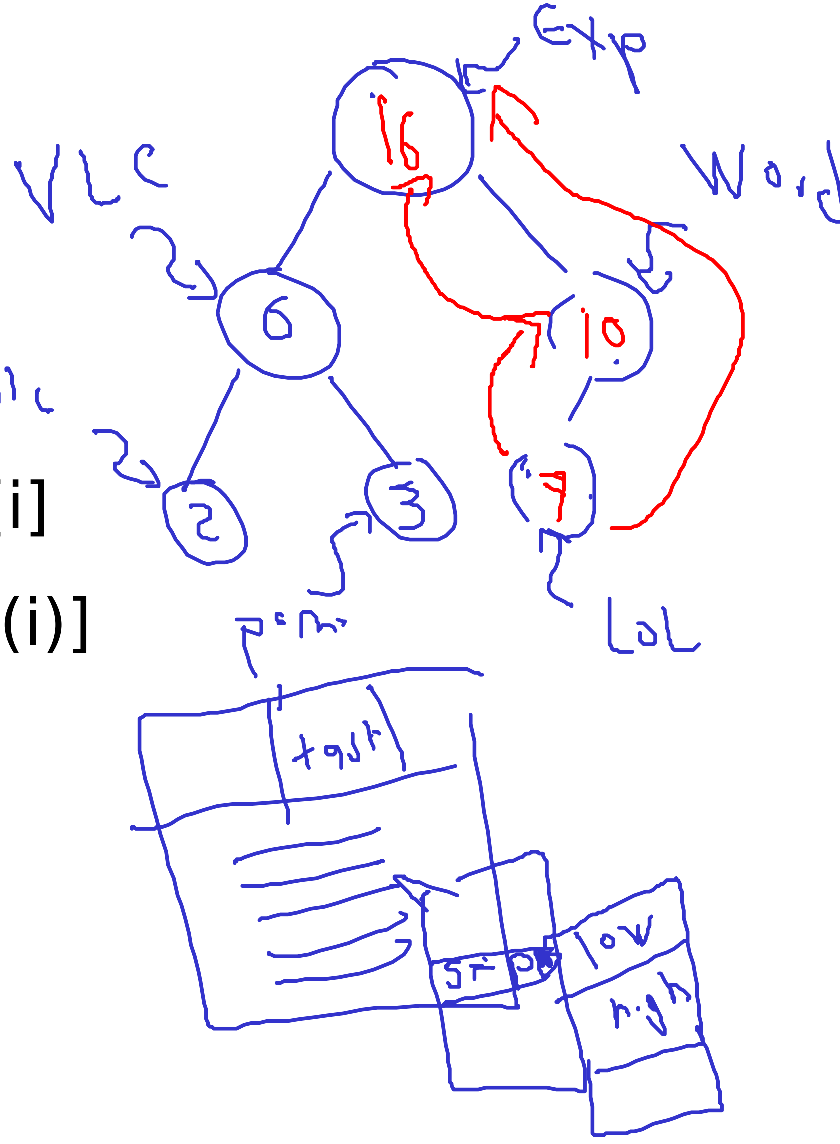
Tiempo de ejecución: O(lgn)

Heapsort

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

```
if key<A[i]
 then error "key error "
A[i]←key        Θ(1)
while i>1 and A[PARENT(i)]<A[i]
    do exchange A[i]↔A[PARENT(i)]
        i←PARENT(i)
```

O(log(n))



Tiempo de ejecución: O(lgn)

Heapsort

MAX-HEAP-INSERT(A, key)

heap-size[A]← heap-size[A]+1

i←heap-size[A]

$O(1)$

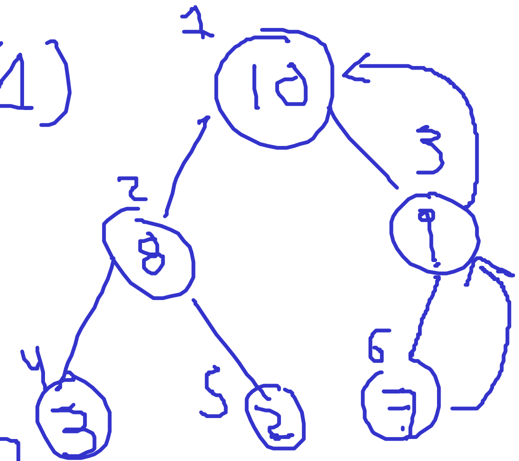
while i>1 and A[PARENT(i)]<key

do exchange A[i]↔A[PARENT(i)]

i←PARENT(i)

A[i]←key

$O(\log n)$



Tiempo de ejecución: O(lgn)

Referencias

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press. Chapter 6

Gracias

Próximo tema:

Ordenamiento: Quicksort