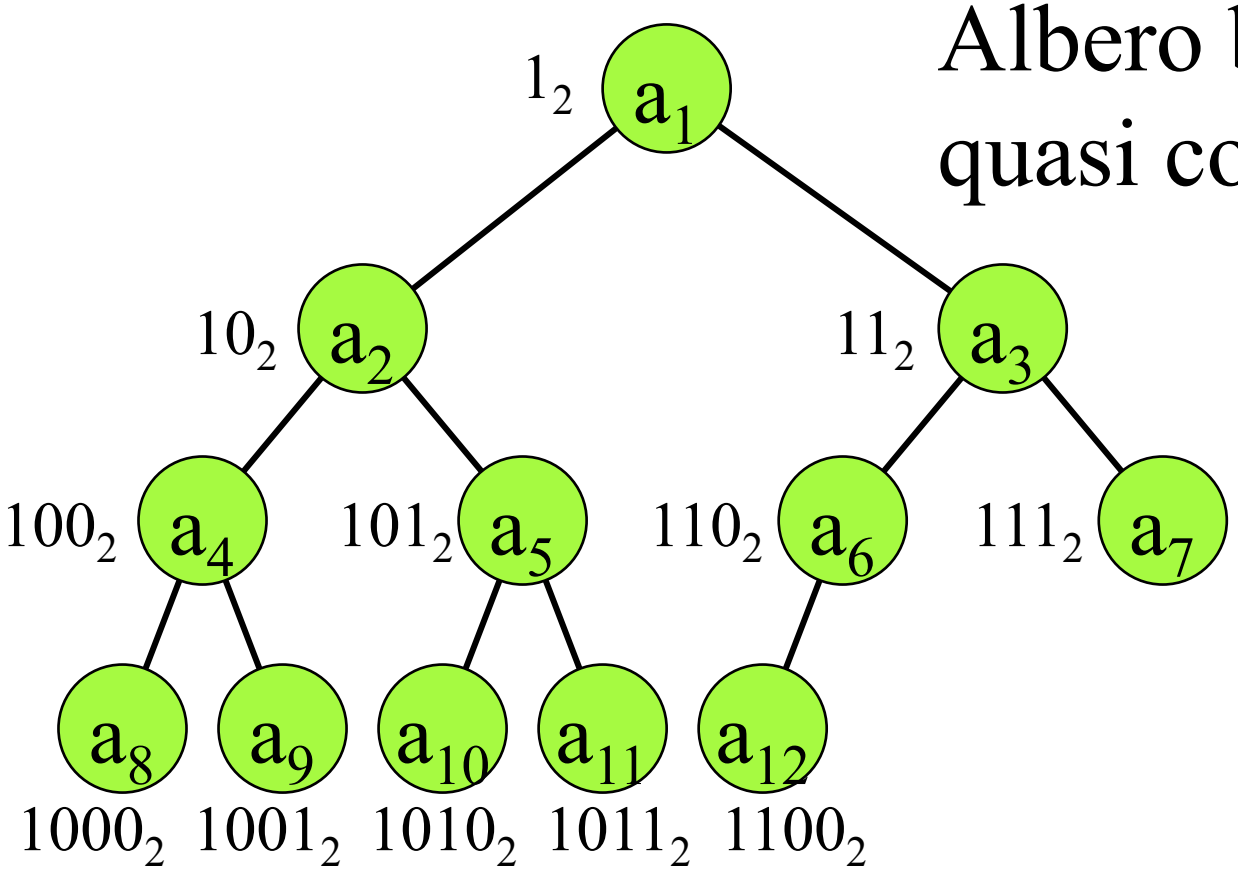
Soluzione: Algoritmo Heap-Sort

Un array $A[1..n]$ può essere interpretato come un albero binario:

- $A[1]$ è la radice,
- $A[2i]$ e $A[2i+1]$ sono i figli di $A[i]$
- $A[\lfloor i / 2 \rfloor]$ è il padre di $A[i]$

1	2	3	4	5	6	7	8	9	10	11	12
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}

Albero binario quasi completo



Proprietà di un heap (mucchio)

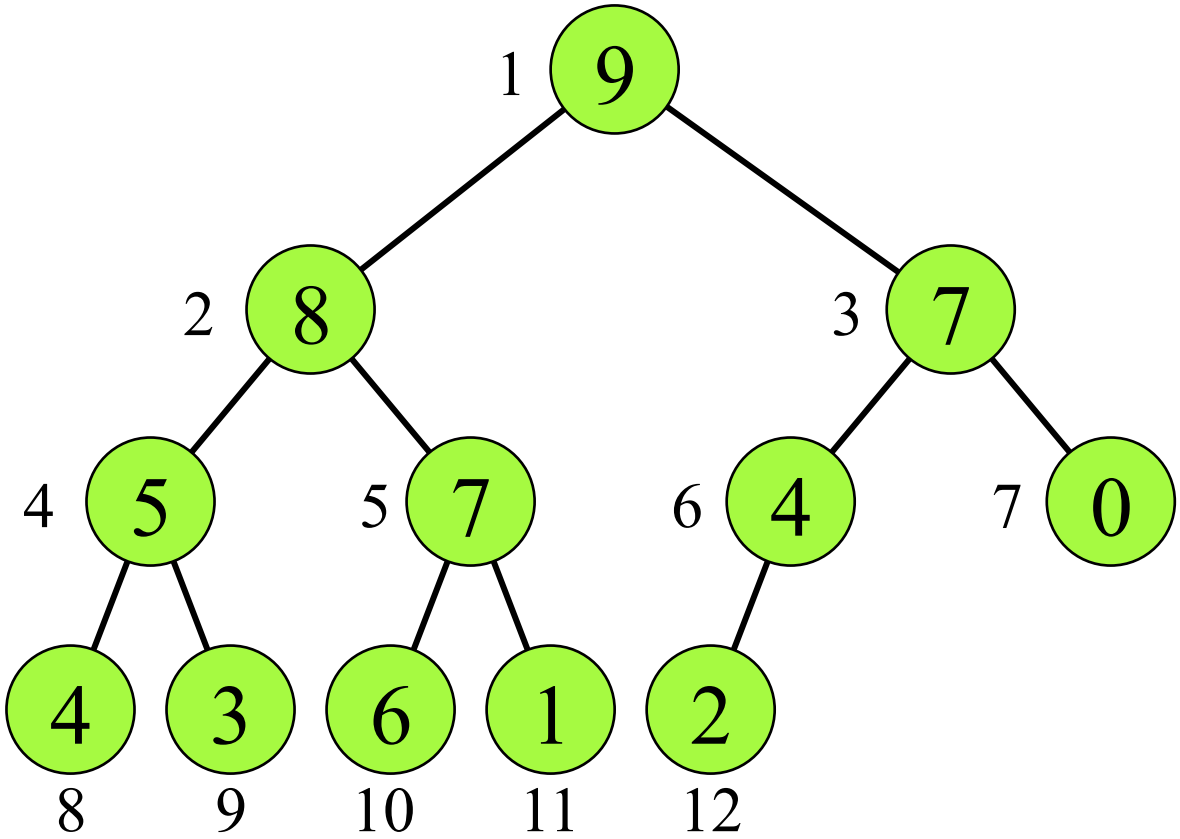
Diciamo che $A[1..n]$ è un (è ordinato a) max-heap se ogni elemento $A[i]$ soddisfa la seguente proprietà:

"$A[i]$ è maggiore o uguale di ogni suo discendente in $A[1..n]$"

Per brevità indicheremo questa proprietà con $H(i)$

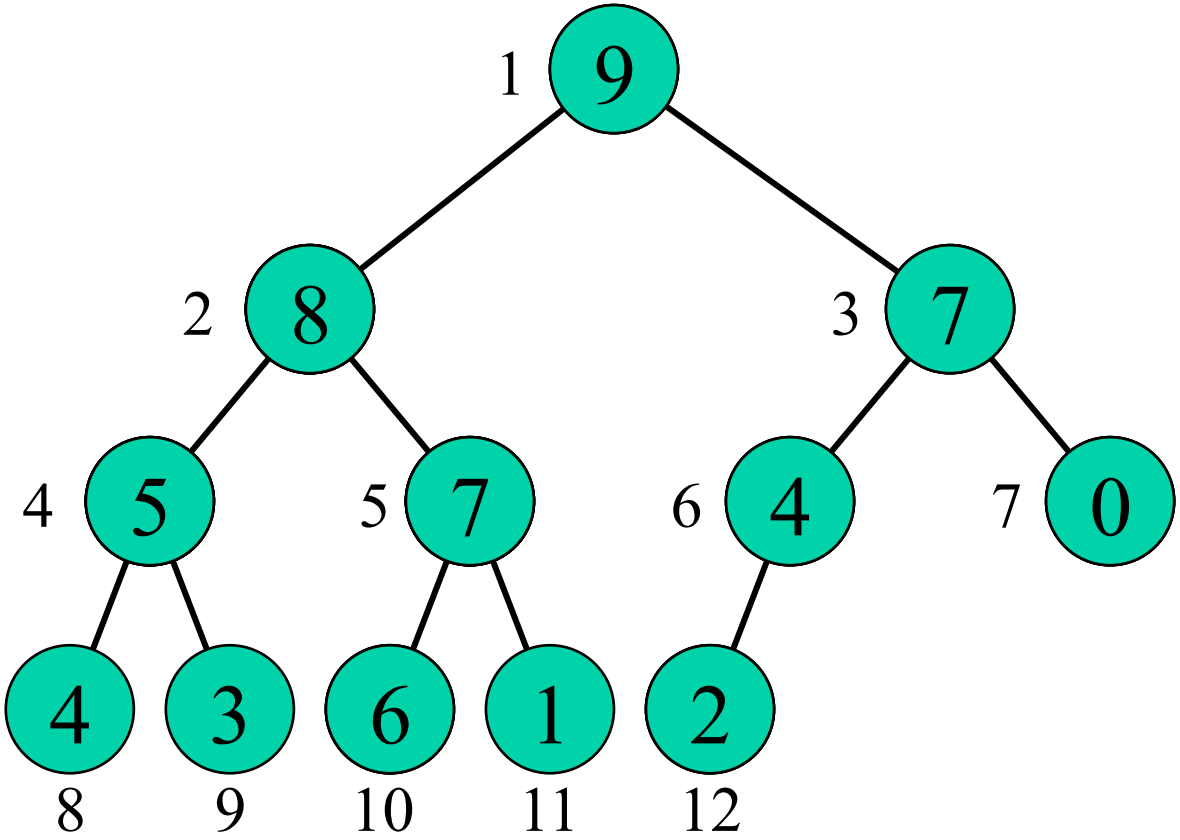
Un max-heap

1	2	3	4	5	6	7	8	9	10	11	12
9	8	7	5	7	4	0	4	3	6	1	2



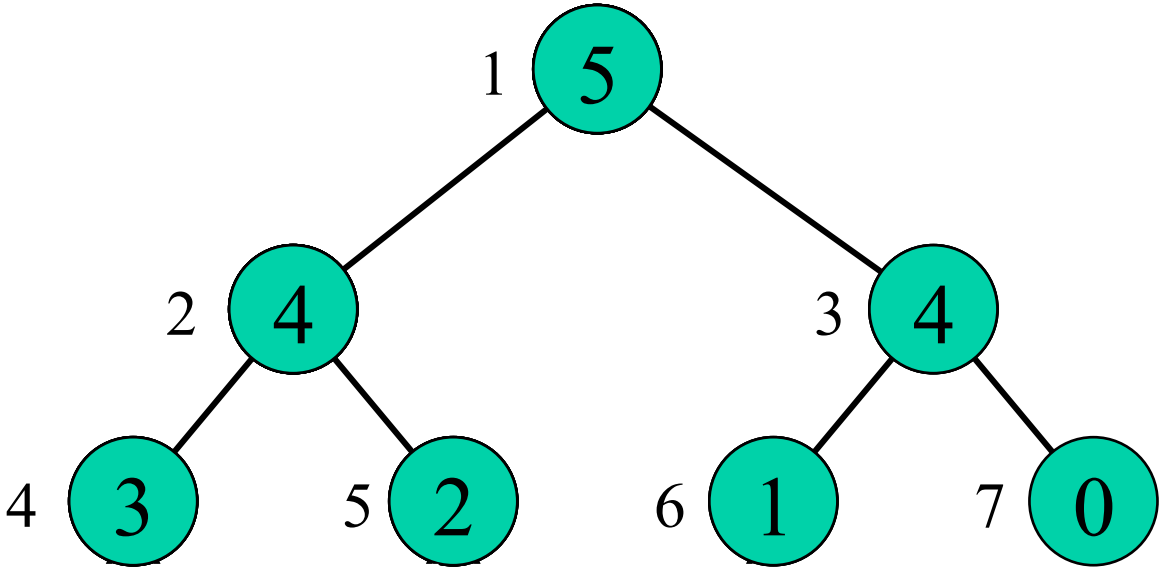
Costruzione di un max-heap

1	2	3	4	5	6	7	8	9	10	11	12
9	8	7	5	7	4	0	4	3	6	1	2



Ordinamento dell'array

1	2	3	4	5	6	7	8	9	10	11	12
5	4	4	3	2	1	0	6	7	7	8	9



```
Max-Heapfy(A,i)
  l = 2i, r =2i+1
  m = i
  if l ≤ A.heapsize and A[l] > A[m]
     m = l
  if r ≤ A.heapsize and A[r] > A[m]
     m = r
  if m ≠ i
     t = A[i], A[i] = A[m], A[m] = t
     Max-Heapfy(A,m)
```

```
Build-Max-Heap (A)
   A.heapsize = A.length
   for i = ⌊A.lenght/2⌋ downto 1
      Max-Heapfy(A,i)
```

```
Heap-Sort (A)
  Build-Max-Heap(A)
  for i = A.length downto 2
    t = A[i], A[i] = A[1], A[1] = t
    A.heapsize = A.heapsize - 1
    Max-Heapfy(A,1)
```

Abbiamo visto che la complessità nel caso pessimo di ogni algoritmo di ordinamento sul posto che confronta e scambia tra loro elementi consecutivi dell'array è $\Omega(n^2)$.

Per ottenere algoritmi più efficienti dobbiamo quindi operare confronti e scambi tra elementi "***distanti***" dell'array.

L'algoritmo Heap-Sort confronta elementi non consecutivi e possiamo quindi sperare che la sua complessità sia minore.

In effetti Heap-Sort richiede tempo O(n log n) per ordinare un array di n elementi (vedi Libro 6.2, 6.3, 6.4)

Implementazione di code con priorità

Gli heap binari si possono usare, oltre che per ordinare un array, anche per implementare delle *code con priorità*.

Le code con priorità sono delle strutture dati in cui è possibile immagazzinare degli oggetti x a cui è attribuita una priorità $x.key$ ed estrarli uno alla volta in ordine di priorità.

Le operazioni fondamentali sulle code con priorità sono:

Insert(S, x): aggiunge x alla coda S

Maximum(S): ritorna $x \in S$ con $x.key$ massima

Extract-Max(S): toglie e ritorna $x \in S$ con $x.key$ massima.

Possono inoltre essere definite anche:

Increase-Key(S,x,p): aumenta la priorità di x

Change-Key(S,x,p): cambia la priorità di x

```
Heap-Maximum(A)          // A è un max-heap
    if A.heapsize < 1
      error "underflow"
    else
      return A[1]
```

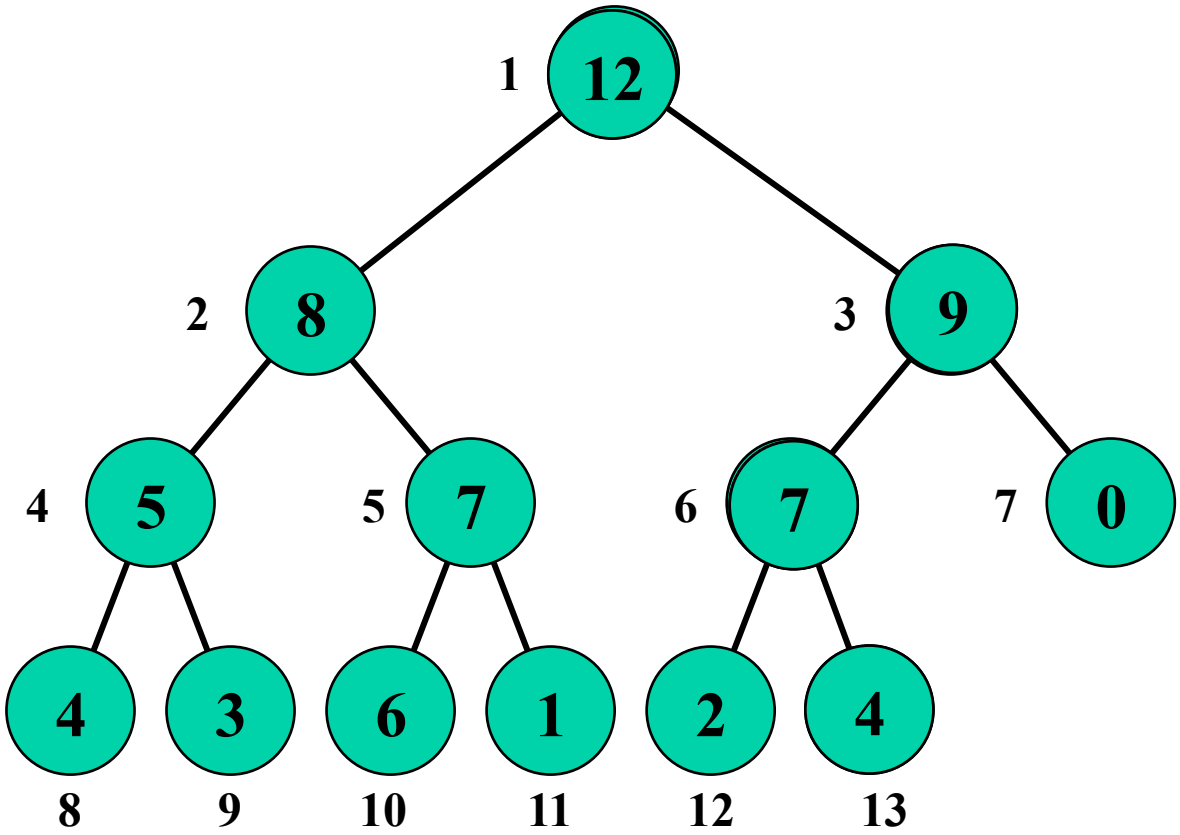
$$T^{Maximum}(n) = \Theta(1)$$

```
Heap-Extract-Max(A)      // A è un max-heap
    if A.heapsize < 1
      error "underflow"
    else
      max = A[1]
      A[1] = A[A.heapsize]
      A.heapsize = A.heapsize - 1
      Max-Heapfy(A,1)
      return max
```

$$T_{\max}^{ExtractMax}(n) = O(\log n)$$

Heap-Insert

1	2	3	4	5	6	7	8	9	10	11	12	13
12	8	9	5	7	7	0	4	3	6	1	2	4



Per realizzare
Heap-Insert e *Heap-Increase-Key*
ci serve una *Max-Heapfy* diversa che invece della proprietà:
"$A[i]$ è maggiore o uguale di ogni suo discendente"
usa la proprietà simmetrica:
"$A[i]$ è minore o uguale di ogni suo ascendente"

entrambe, se vere per ogni elemento dell'array, ci assicurano l'ordinamento a max-heap di A.

La nuova versione *Max-HeapfyR* ricostruisce lo heap quando tutti gli elementi dell'array sono minori o uguali dei loro ascendenti tranne al più quello in posizione *i*.

Max-HeapfyR(*A*,*i*)

// solo *A*[*i*] può non soddisfare la proprietà

while *i* >1 and *A*[$\lfloor i/2 \rfloor$].*key* < *A*[*i*].*key*

scambia *A*[$\lfloor i/2 \rfloor$] con *A*[*i*]

// solo *A*[$\lfloor i/2 \rfloor$] può non soddisfarla

***i* = $\lfloor i/2 \rfloor$**

$$\boxed{T_{\max}^{HeapfyR}(i) = O(\log i)}$$

Heap-Increase-Key(*A*,*i*,*p*) // *A* **max-heap**

```
if p < A[i].key
    error "la nuova priorità è minore"
else
    A[i].key = p
    Max-HeapfyR(A,i)
```

$$T_{\max}^{IncreaseKey}(i) = O(\log i)$$

Max-Heap-Insert(*A*,*x*) // *A* **max-heap**

```
A.heapsize = A.heapsize+1
A[A.heapsize] = x
Max-HeapfyR(A,A.heapsize)
```

$$T_{\max}^{Insert}(n) = O(\log n)$$

Possiamo facilmente realizzare anche una Heap-Change-Key nel modo seguente:

```
Heap-Change-Key(A,i,p)  // A max-heap
    if p < A[i].key
        A[i].key = p
        Max-Heapfy(A,i)
    else
        A[i].key = p
        Max-HeapfyR(A,i)
```

$$\boxed{T_{\max}^{ChangeKey}(n) = O(\log n)}$$