

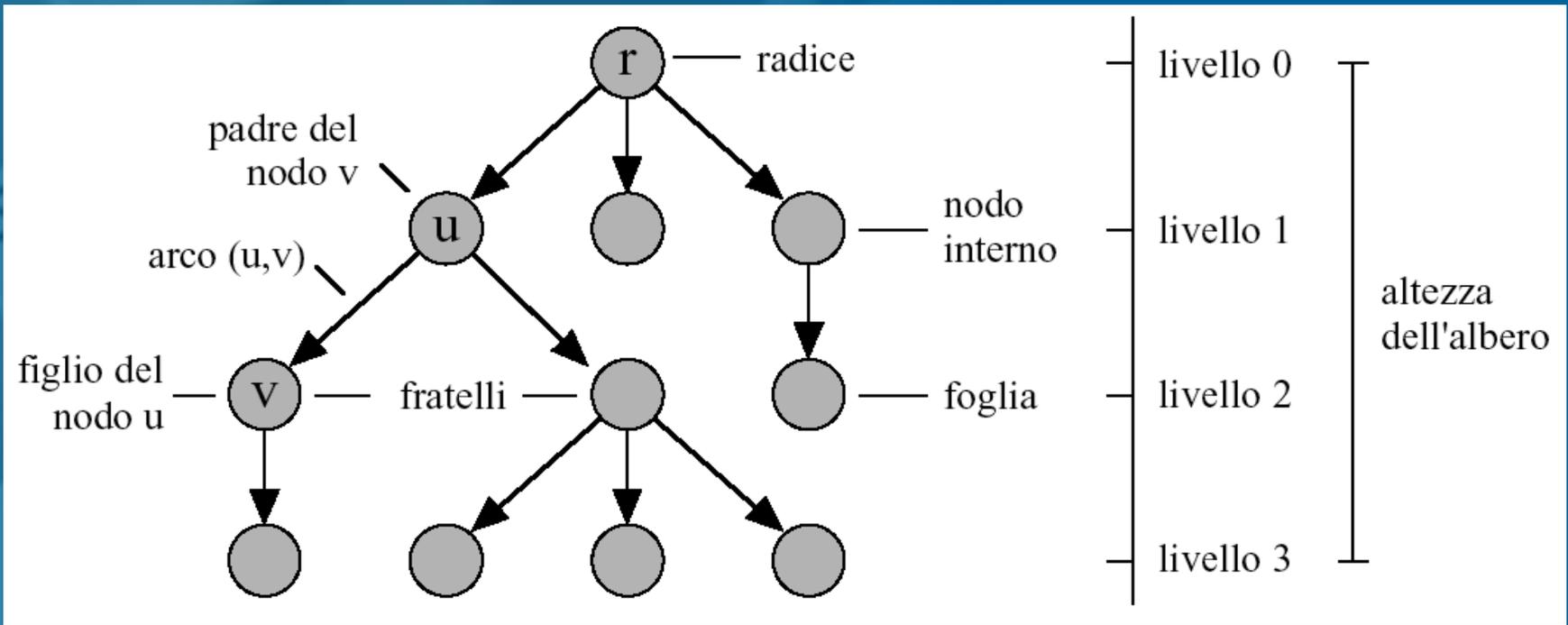
Progettare algoritmi veloci usando strutture dati efficienti

Un esempio: HeapSort

HeapSort

- Stesso approccio incrementale del selectionSort
 - seleziona gli elementi dal più grande al più piccolo
 - usa una **struttura dati efficiente**
 - estrazione in tempo $O(\log n)$ del massimo
- **Tipo di dato**
 - Specifica una collezione di oggetti e delle operazioni di interesse su tale collezione (es. Dizionario: mantiene un insieme di elementi con chiavi soggetto a operazioni di inserimento, cancellazione, ricerca)
- **Struttura dati**
 - Organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile
- **Cruciale: progettare una struttura dati H su cui eseguire efficientemente le operazioni:**
 - dato un array A, generare velocemente H
 - trovare il più grande oggetto in H
 - cancellare il più grande oggetto da H

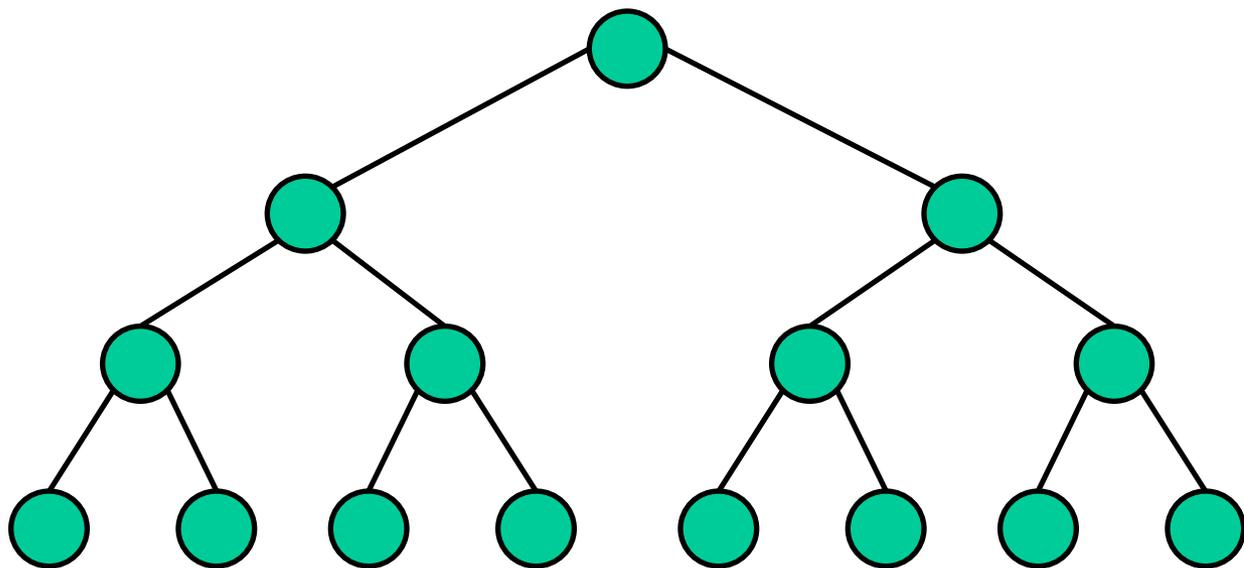
Alberi: qualche altra definizione



albero d-ario: albero in cui tutti i nodi interni hanno (al più) d figli

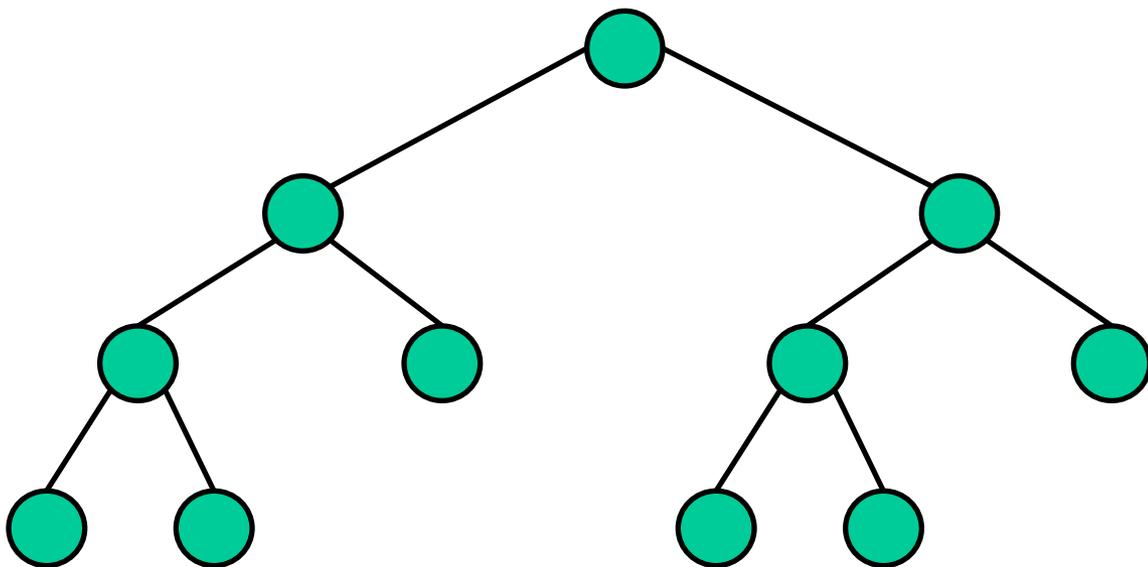
$d=2 \rightarrow$ **albero binario**

un albero d -ario è **completo**: se tutti nodi interni hanno esattamente d figli e le foglie sono tutte allo stesso livello



albero binario
completo

albero binario
(non completo)

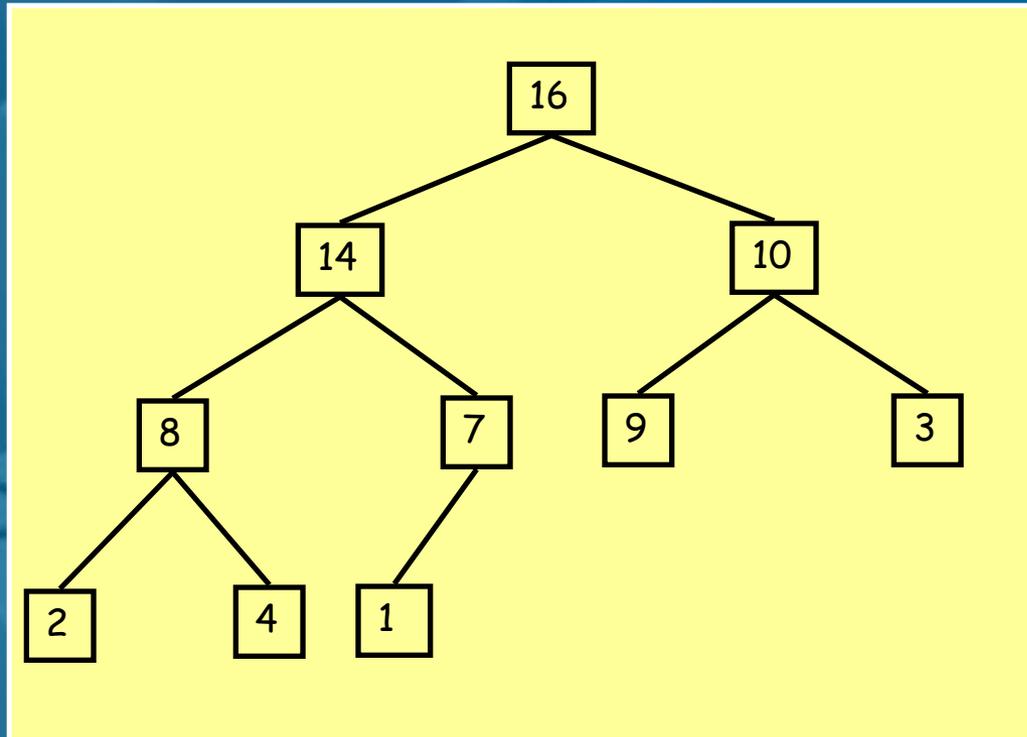


HeapSort

- **Struttura dati heap** associata ad un insieme $S =$ albero binario radicato con le seguenti proprietà:
 - 1) completo fino al penultimo livello (struttura rafforzata: foglie sull'ultimo livello tutte compattate a sinistra)
 - 2) gli elementi di S sono memorizzati nei nodi dell'albero (ogni nodo v memorizza uno e un solo elemento, denotato con $\text{chiave}(v)$)
 - 3) **$\text{chiave}(\text{padre}(v)) \geq \text{chiave}(v)$** per ogni nodo v diverso dalla radice

...un esempio

In questa
direzione è
presente un
ordinamento



il massimo è
contenuto
nella radice!



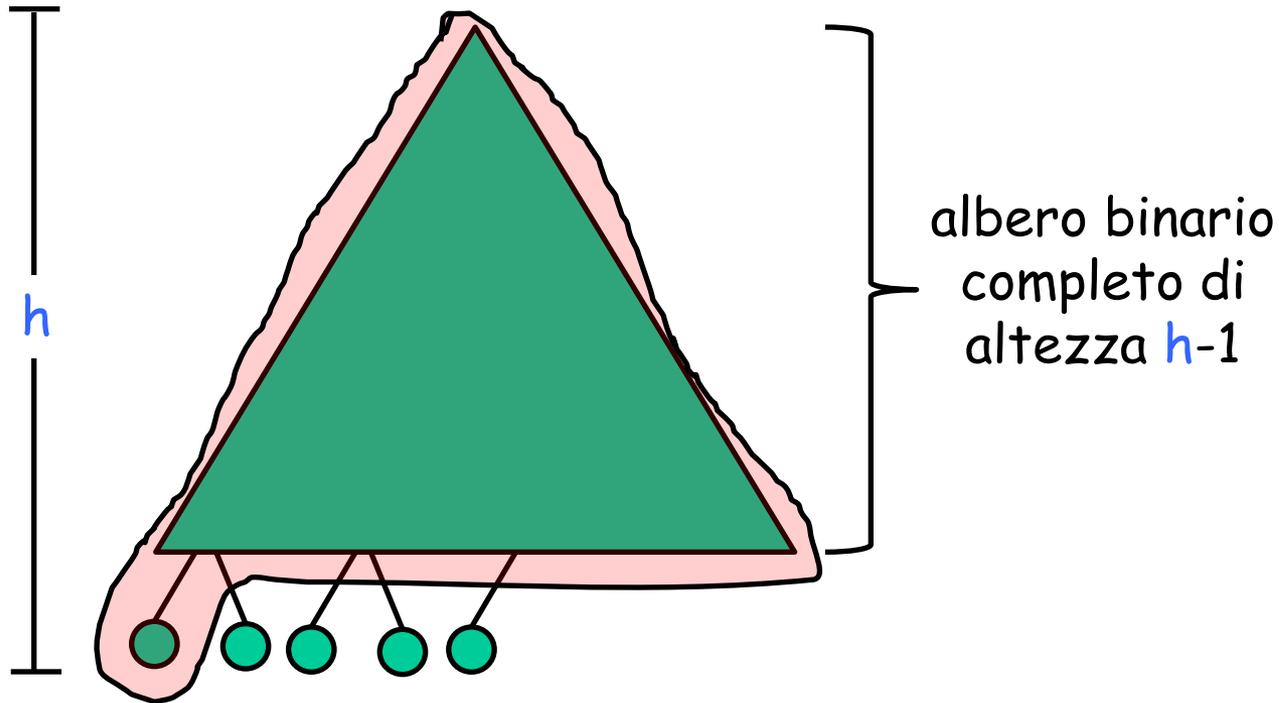
In questa direzione non è
presente un ordinamento

Proprietà salienti degli heap

- 1) Il **massimo** è contenuto **nella radice**
- 2) L'albero ha **altezza $O(\log n)$**
- 3) Gli heap con struttura rafforzata possono essere rappresentati in un **array di dimensione pari a n**

Altezza di un heap

Sia H un heap di n nodi e altezza h .

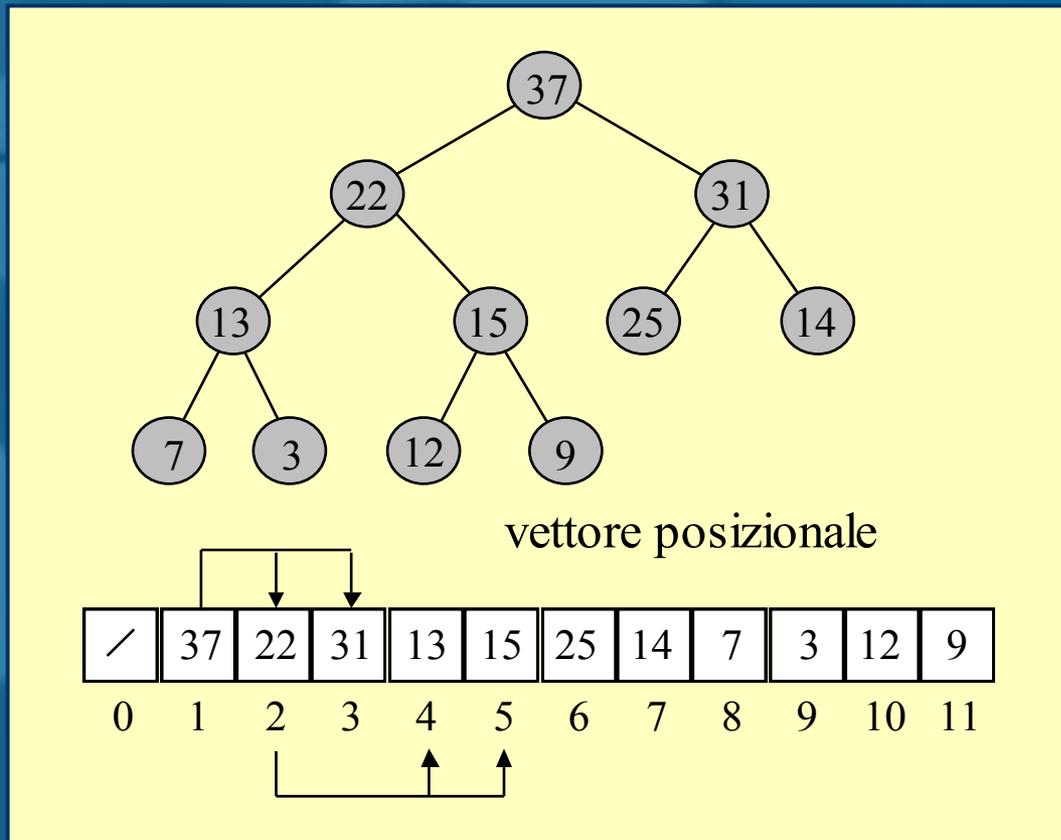


$$n \geq 1 + \sum_{i=0}^{h-1} 2^i = 1 + 2^h - 1 = 2^h$$

➔ $h \leq \log_2 n$

Struttura dati heap

Rappresentazione con vettore posizionale



$$\text{sin}(i) = 2i$$

$$\text{des}(i) = 2i+1$$

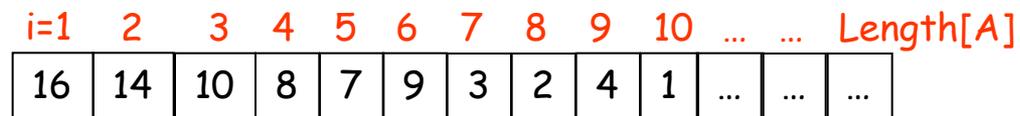
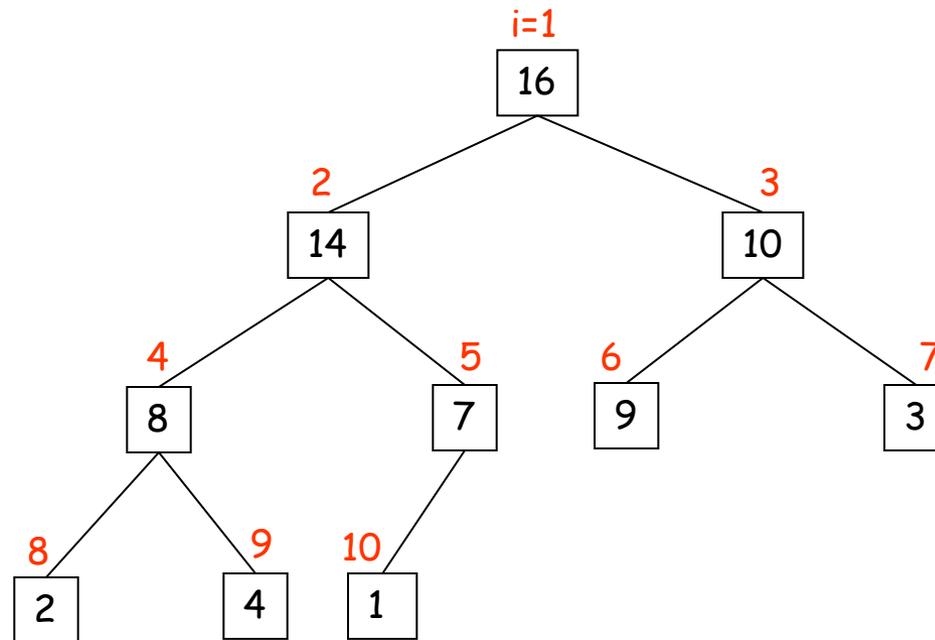
$$\text{padre}(i) = \lfloor i/2 \rfloor$$

è sufficiente un vettore di
dimensione n

in generale
dimensione vettore
diverso da numero
elementi

nello pseudocodice numero oggetti
indicato con $\text{heapsize}[A]$

...ancora un esempio



Heap-size[A]

Heap-size[A] ≠ Length[A]

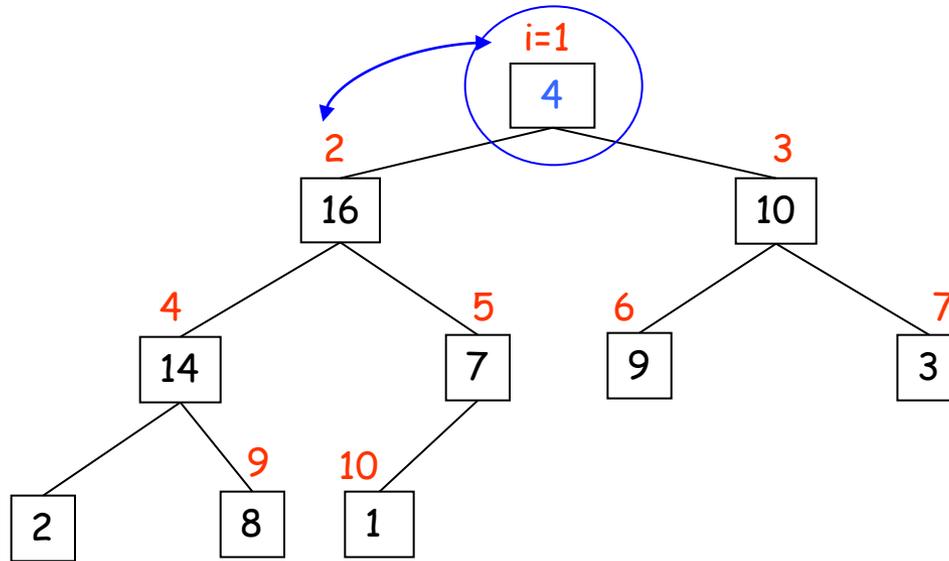
La procedura `fixHeap`

Sia v la radice di H . Assume che i sottoalberi radicati nel figlio sinistro e destro di v sono heap, ma la proprietà di ordinamento delle chiavi non vale per v . Posso ripristinarla così:

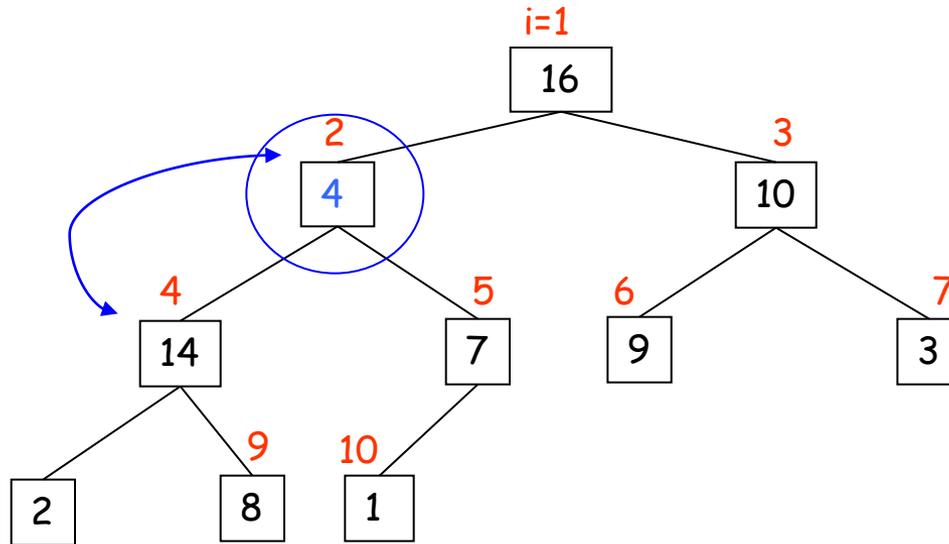
```
fixHeap(nodo v, heap H)
  if (v non è una foglia) then
    sia u il figlio di v con chiave massima
    if (chiave(v) < chiave(u)) then
      scambia chiave(v) e chiave(u)
      fixHeap(u, H)
```

Tempo di esecuzione: $O(\log n)$

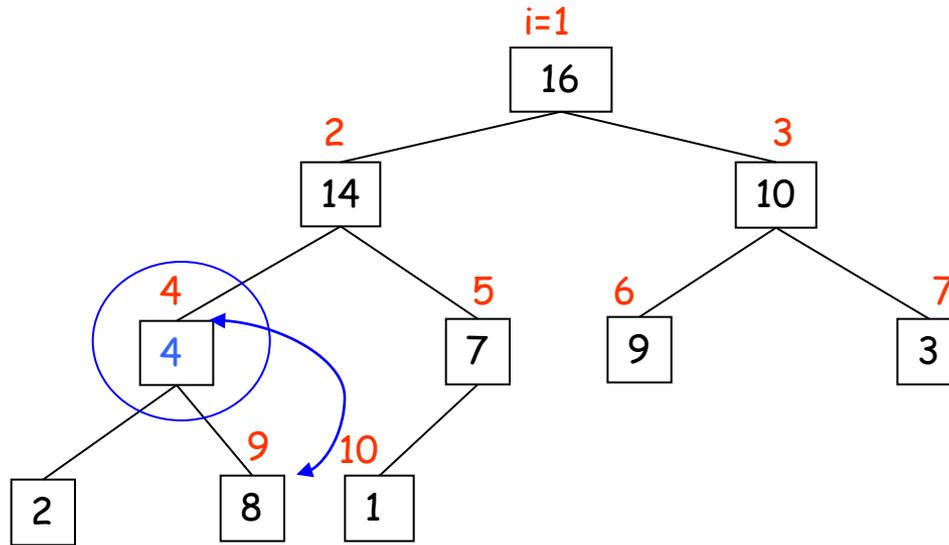
FixHeap - esempio



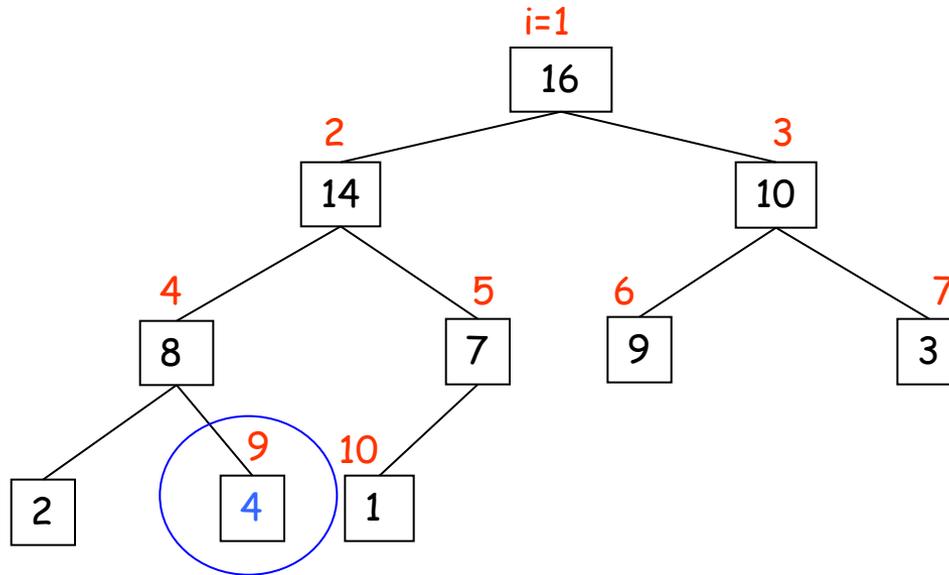
FixHeap - esempio



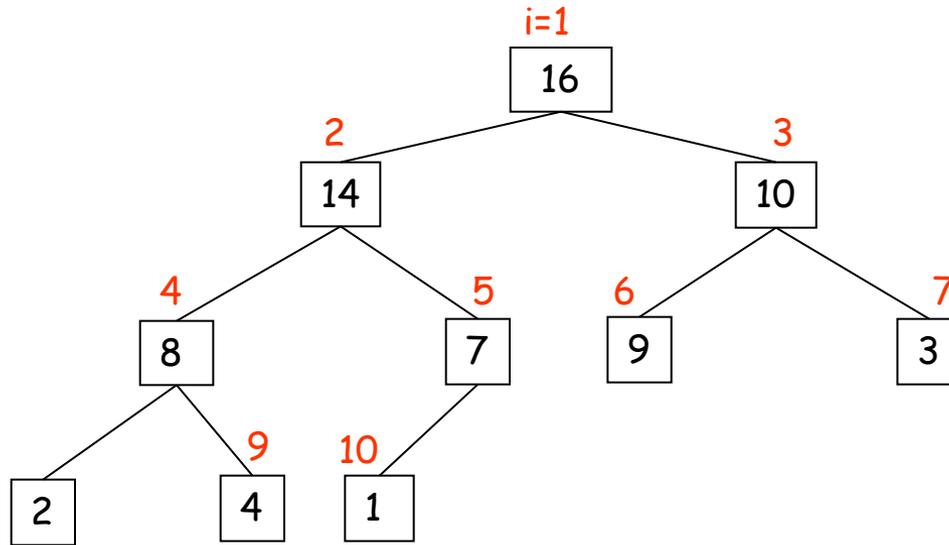
FixHeap - esempio



FixHeap - esempio



FixHeap - esempio



uno pseudocodice di **fixHeap** più dettagliato
(l'heap è mantenuto attraverso un vettore posizionale)

fixHeap (i,A)

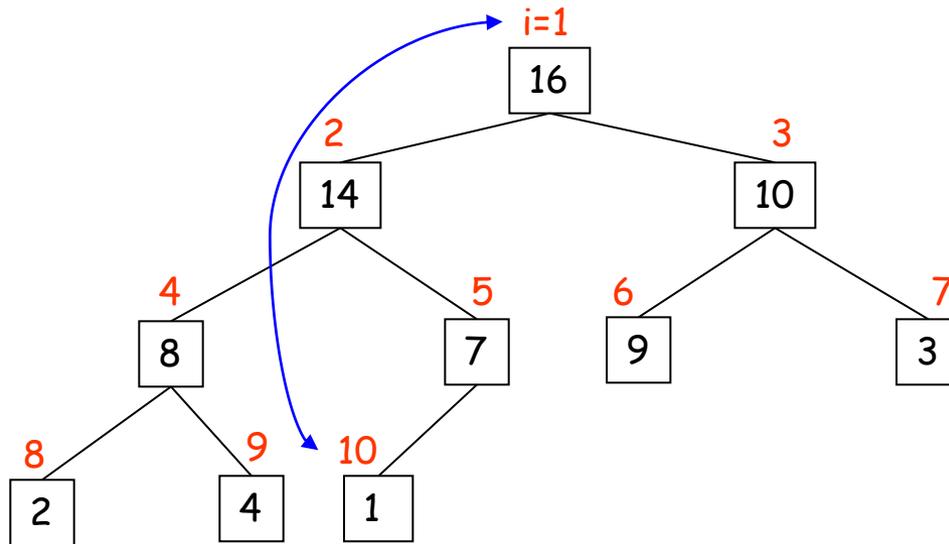
1. $s = \text{sin}(i)$
2. $d = \text{des}(i)$
3. **if** ($s \leq \text{heapsize}[A]$ e $A[s] > A[i]$)
4. **then** massimo=s
5. **else** massimo=i
6. **if** ($d \leq \text{heapsize}[A]$ e $A[d] > A[\text{massimo}]$)
7. **then** massimo=d
8. **if** (massimo≠i)
9. **then** scambia $A[i]$ e $A[\text{massimo}]$
10. fixHeap(massimo,A)

Estrazione del massimo

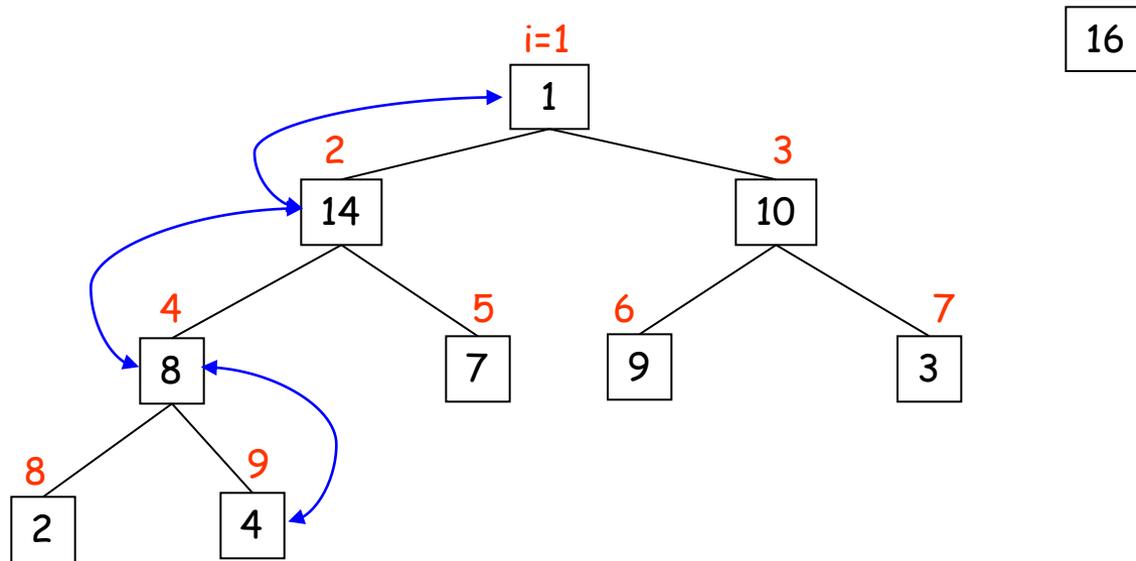
- Copia nella radice la chiave contenuta nella la foglia più a destra dell'ultimo livello
 - **nota:** è l'elemento in posizione n (n : dimensione heap)
- Rimuovi la foglia
- Ripristina la proprietà di ordinamento a heap richiamando `fixHeap` sulla radice

Tempo di esecuzione: $O(\log n)$

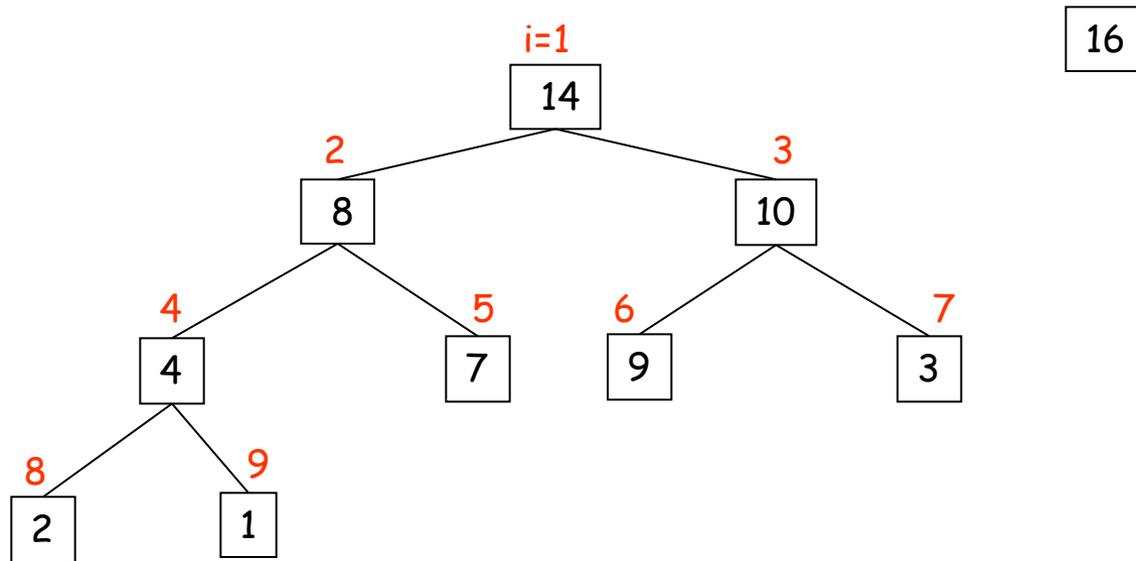
Estrazione del massimo



Estrazione del massimo



Estrazione del massimo



Costruzione dell'heap

Algoritmo ricorsivo basato sulla tecnica del divide et impera

```
heapify(heap H)
```

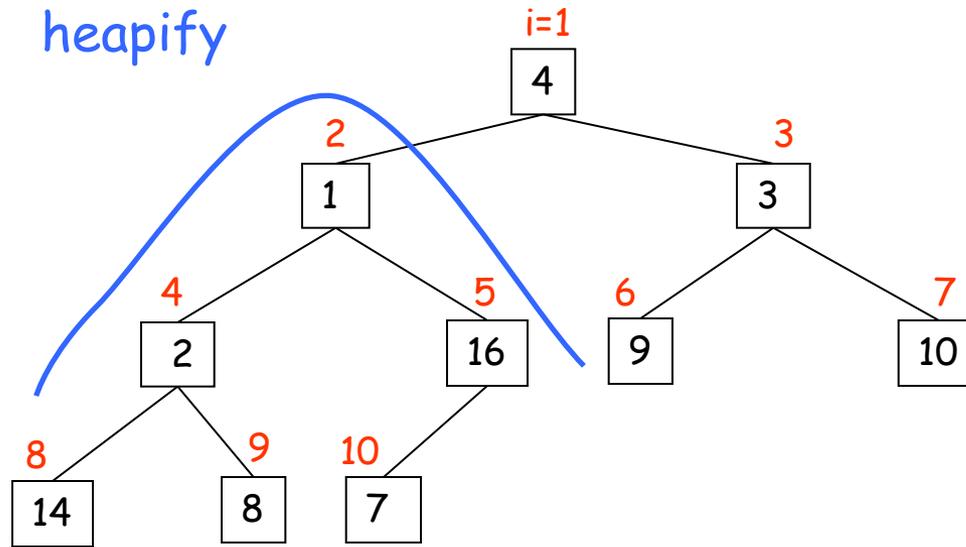
```
  if (H non è vuoto) then
```

```
    heapify(sottoalbero sinistro di H)
```

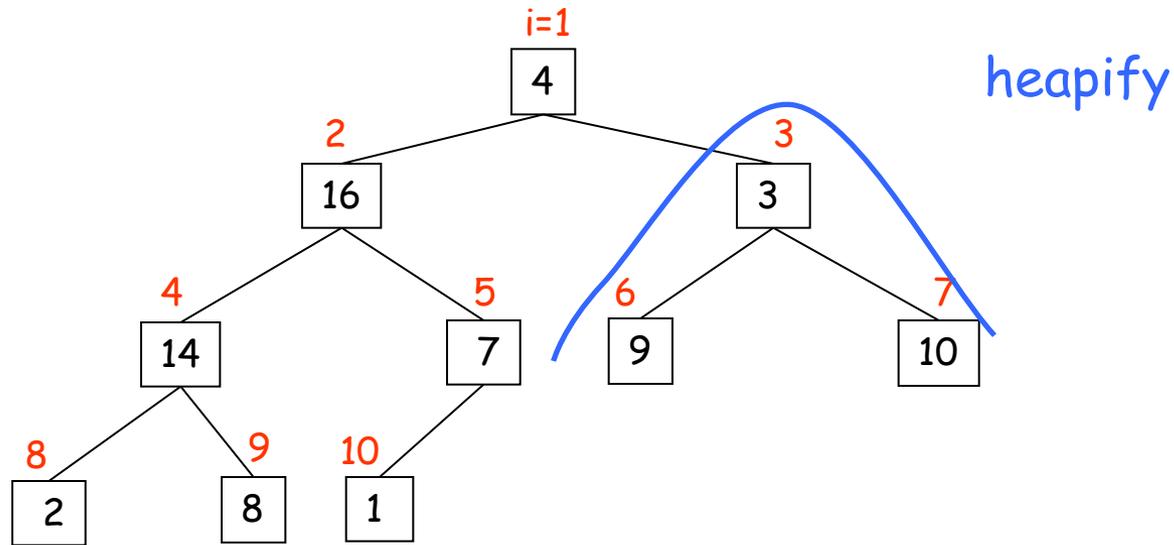
```
    heapify(sottoalbero destro di H)
```

```
    fixHeap(radice di H,H)
```

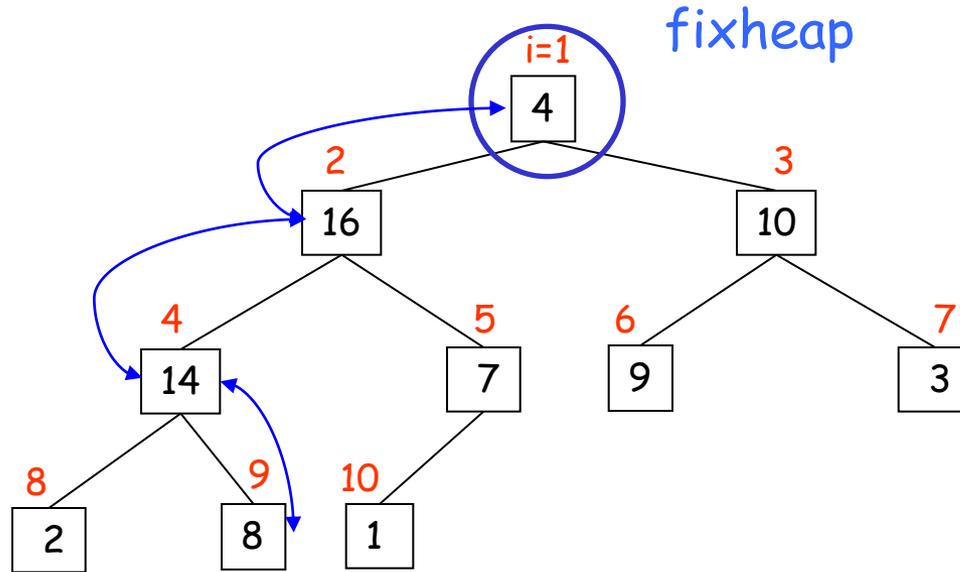
Heapify - Un esempio



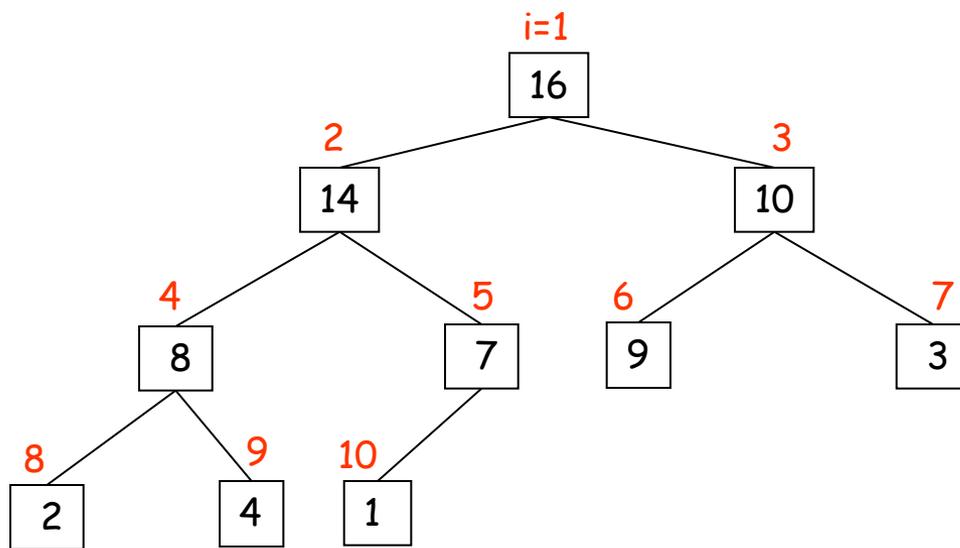
Heapify - Un esempio



Heapify - Un esempio



Heapify - Un esempio



E' un **heap**!

Complessità heapify

Sia h l'altezza di un heap con n elementi

Sia $n' \geq n$ l'intero tale che un heap con n' elementi ha

1. altezza h
2. è completo fino all'ultimo livello

Vale: $T(n) \leq T(n')$ e $n' \leq 2n$

Tempo di esecuzione: $T(n') = 2 T((n'-1)/2) + O(\log n')$
 $\leq 2 T(n'/2) + O(\log n')$

➔ $T(n') = O(n')$ dal Teorema Master

Quindi: $T(n) \leq T(n') = O(n') = O(2n) = O(n)$

Max-Heap e Min-Heap

e se volessi una struttura dati che mi permette di estrarre il **minimo** velocemente invece del **massimo**?

Semplice: costruisco un **min-heap** invertendo la proprietà di ordinamento delle chiavi. Cioè richiedo che

$$\text{chiave}(\text{padre}(v)) \leq \text{chiave}(v)$$

per ogni v (diverso dalla radice)

e come mai noi abbiamo progettato un max-heap e non un min-heap?

...fra un po' lo capiremo 😊

L'algoritmo HeapSort

- Costruisce un heap tramite heapify
- Estrae ripetutamente il massimo per $n-1$ volte
 - ad ogni estrazione memorizza il massimo nella posizione dell'array che si è appena liberata

heapSort (A)

1. Heapify(A)
2. Heapsize[A]=n
3. **for** i=n **down to** 2 **do**
4. scambia A[1] e A[i]
5. Heapsize[A] = Heapsize[A] - 1
6. fixHeap(1,A)

} $O(n)$
n-1
estrazioni
di costo
} $O(\log n)$

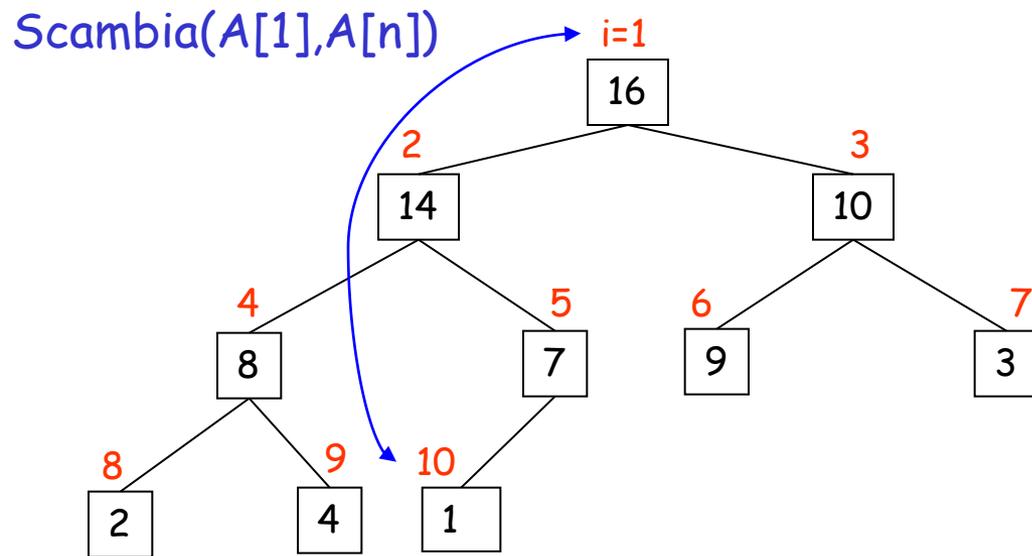


ordina in loco in tempo $O(n \log n)$

Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

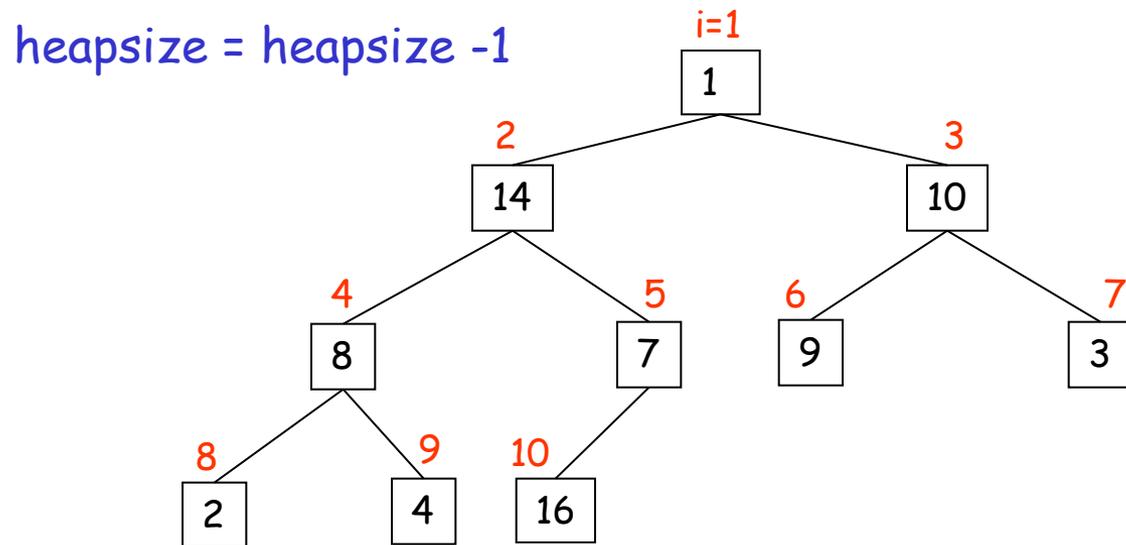
Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

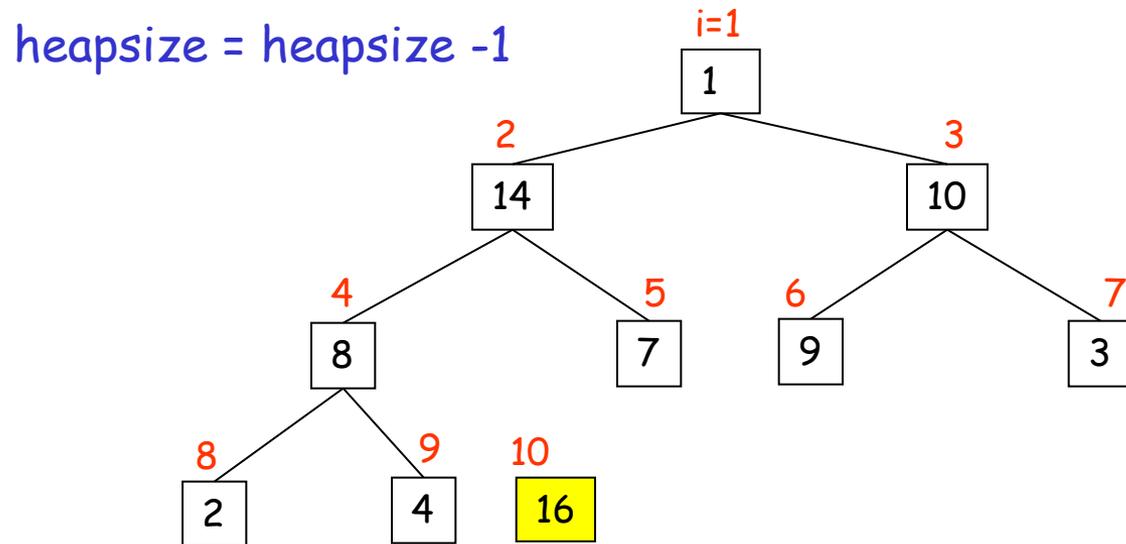
Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

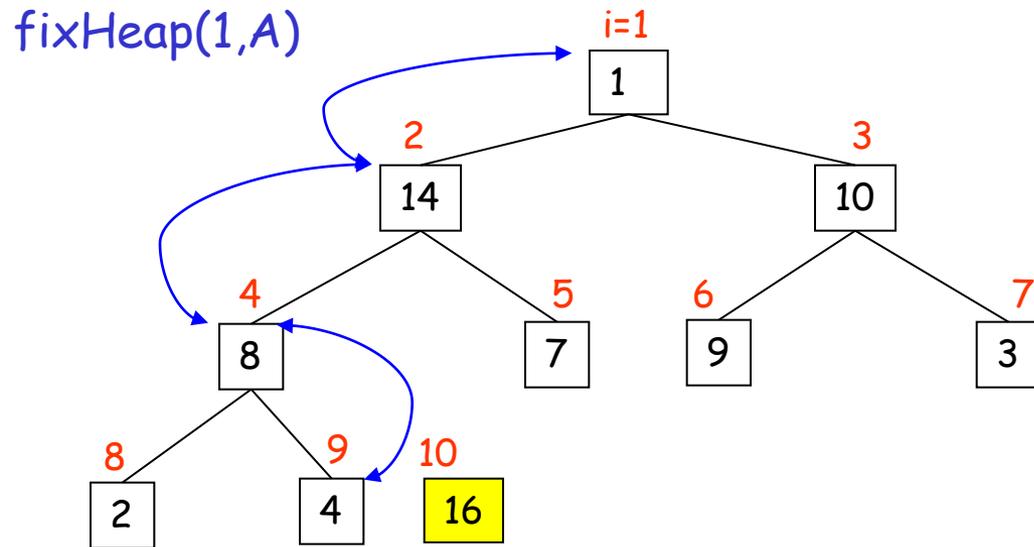
Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

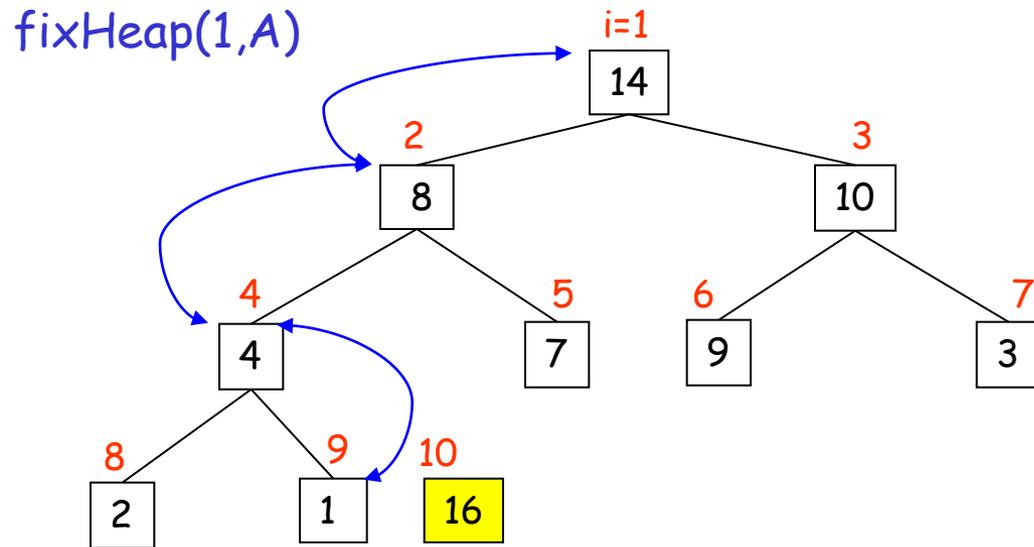
Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

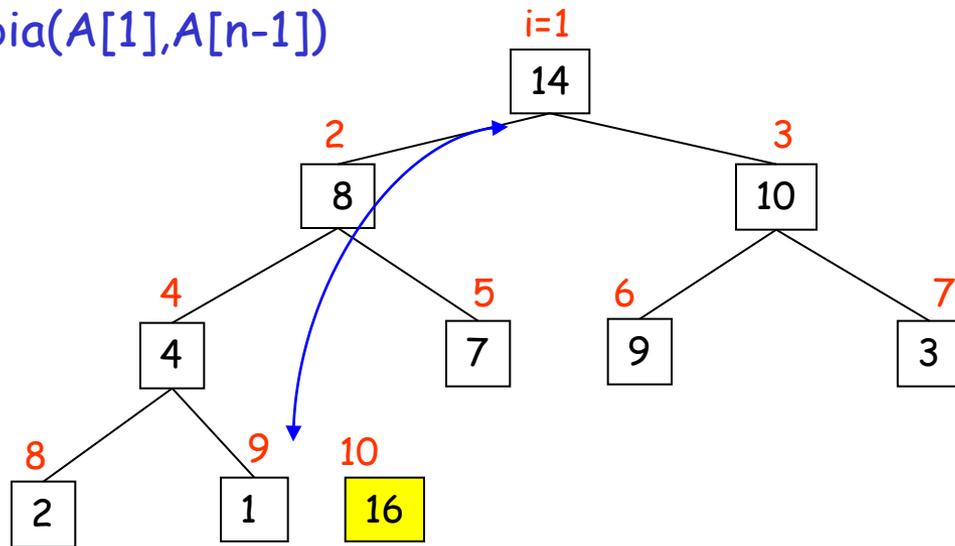


Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

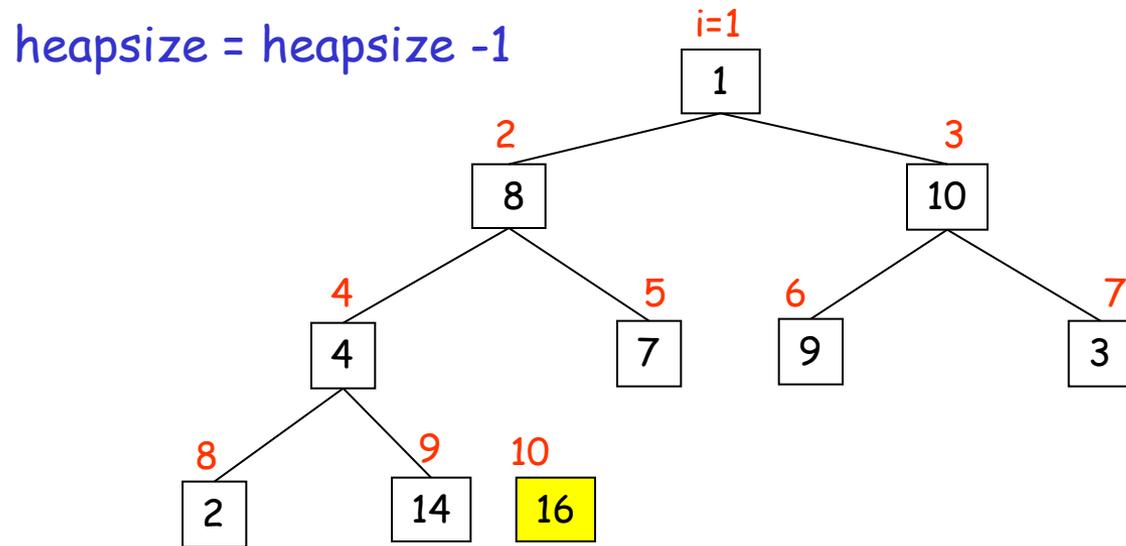
Scambia($A[1], A[n-1]$)



Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

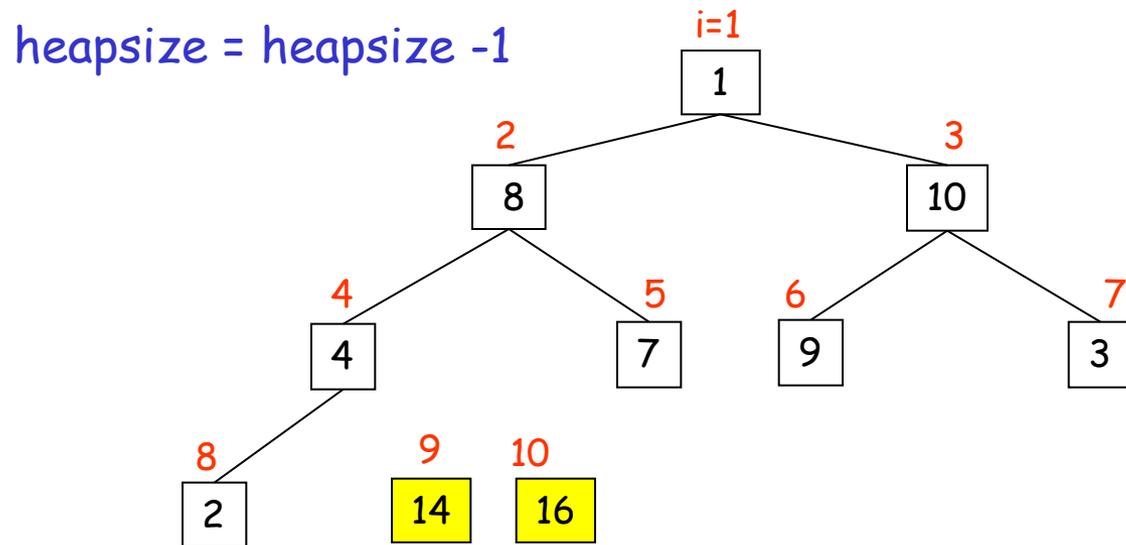
Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

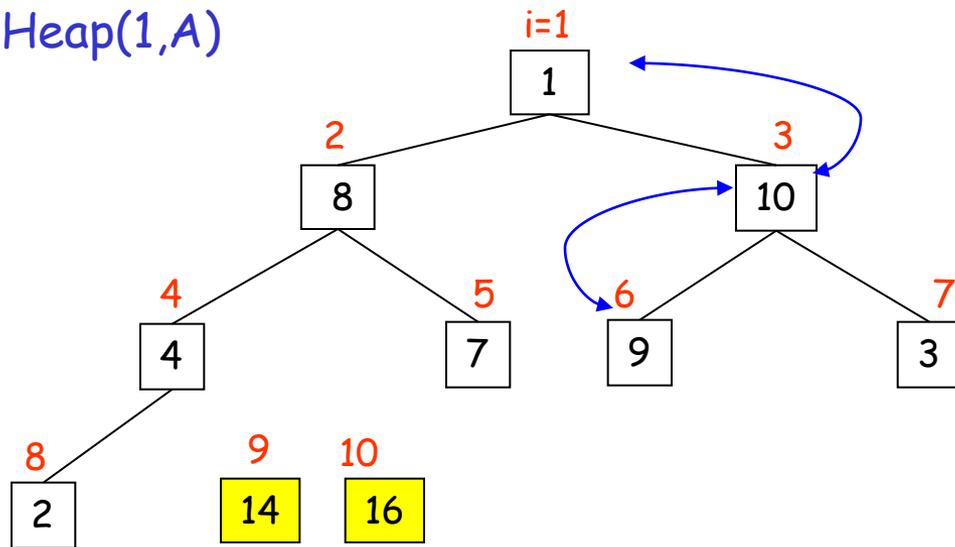


Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

fixHeap(1, A)

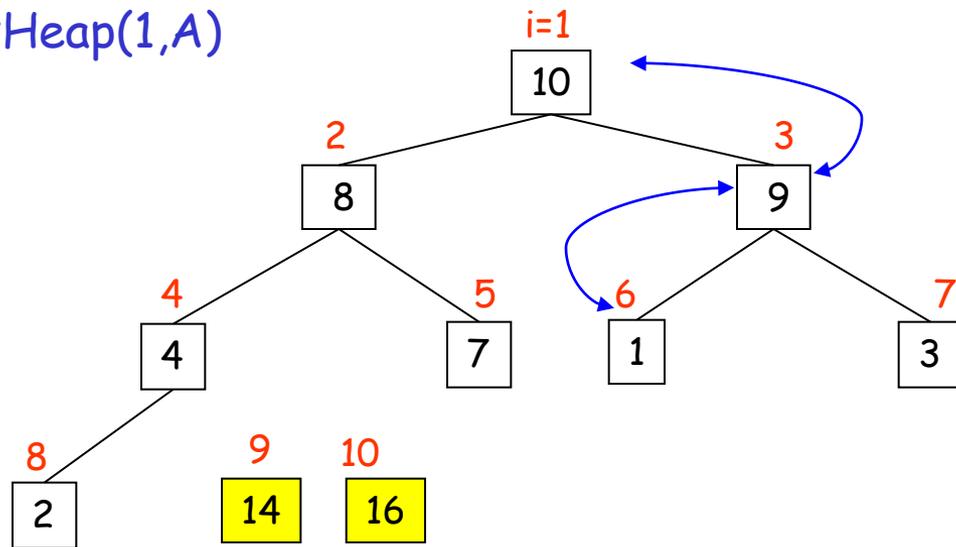


Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

fixHeap(1, A)

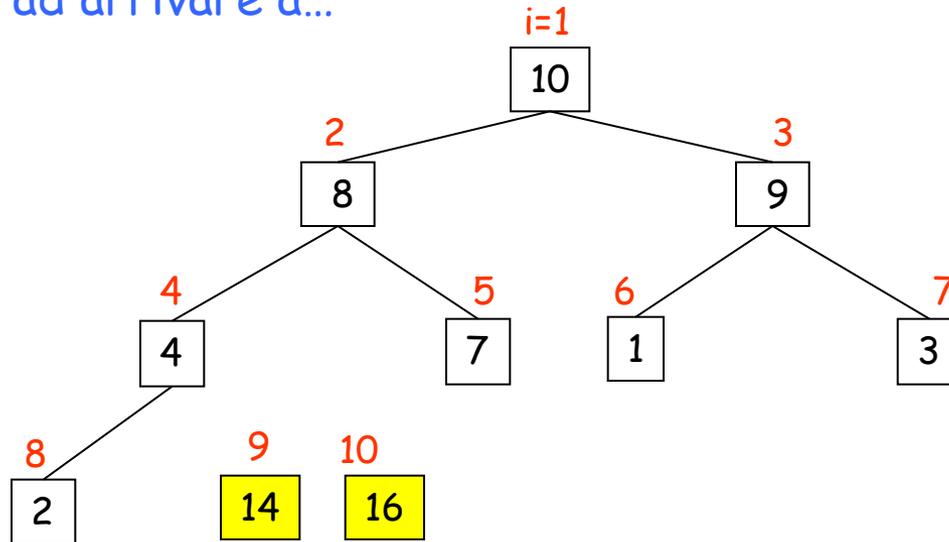


Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

E così via, sino ad arrivare a...

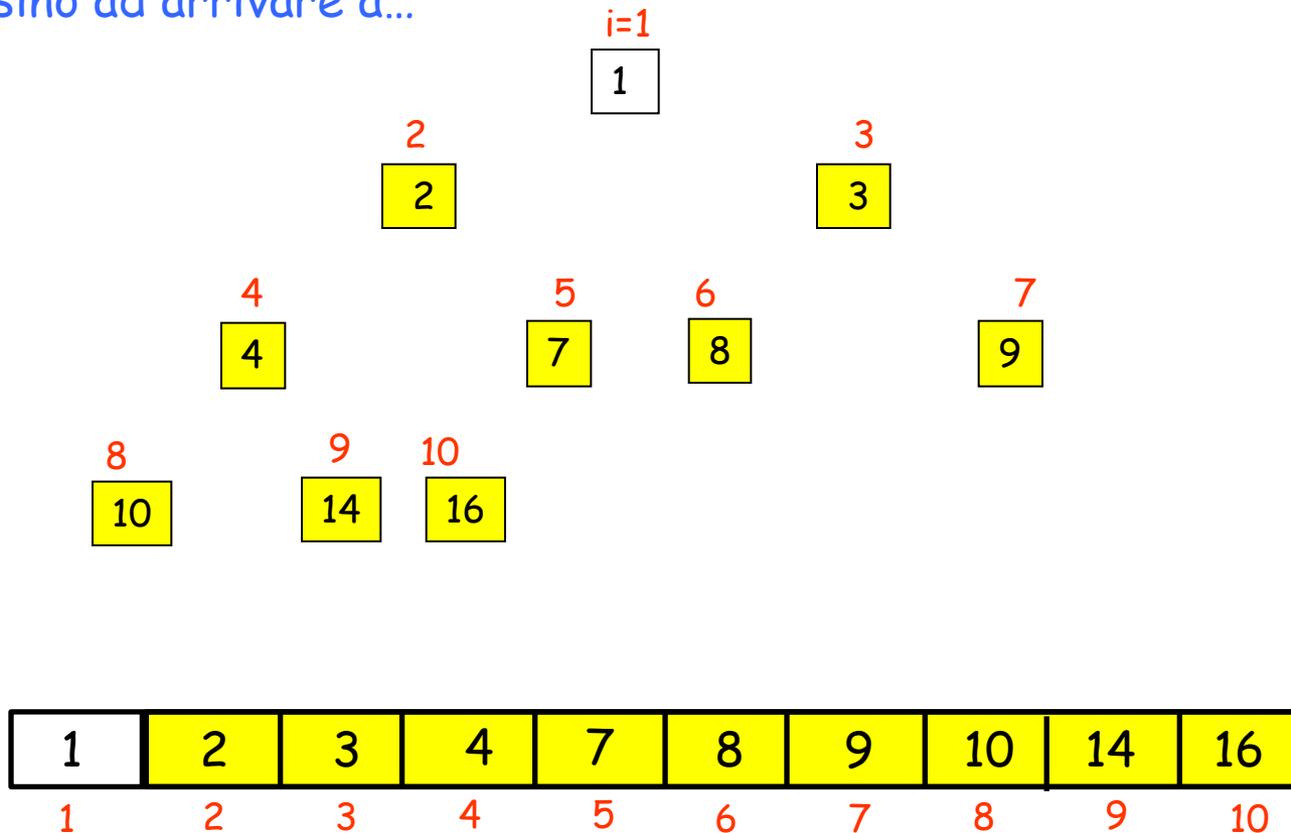


Esempio

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

E così via, sino ad arrivare a...



Max-Heap e Min-Heap

Quindi: come mai abbiamo usato un max-heap e non un min-heap? Potevamo usare anche un min-heap?

...l'uso del max-heap (implementato con un vettore posizionale) ci permette di usare solo memoria ausiliare costante! 😊

Teorema

L'algoritmo **HeapSort** ordina *in loco* un array di lunghezza n in tempo $O(n \log n)$ nel caso peggiore.