

Algoritmi di ordinamento ottimali

L'algoritmo **merge sort** ha complessita' $O(n \log(n))$. → algoritmo di ordinamento ottimale.

A differenza degli algoritmi IS,BS,SS, merge sort non e' pero' un algoritmo di **ordinamento in loco**. Infatti:

- la procedura merge richiede un vettore di appoggio in cui effettuare la fusione

Cio' comporta (insieme alla gestione della struttura ricorsiva) una spesa in termini di:

- ✓ **Tempo** (notare → non modifica l'andamento asintotico, ma s riflette sulle costanti moltiplicative);

- ✓ **Spazio**.

Algoritmo di ordinamento heapsort

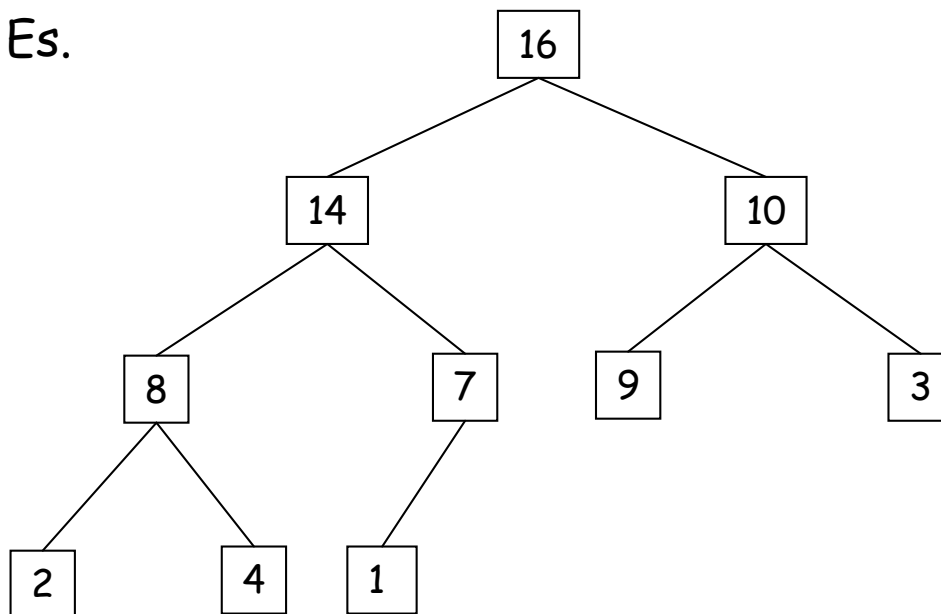
Introduciamo un algoritmo di ordinamento detto **heap sort** che ha le seguenti caratteristiche:

- $T(n) = O(n \log(n)) \rightarrow$ Alg. Ordinamento ottimale
- Ordina in loco.

Per fare cio' dobbiamo introdurre una struttura di dati detta heap.

Heap binario = albero binario in cui ogni nodo figlio ha valore minore o uguale del proprio nodo parente.

Es.

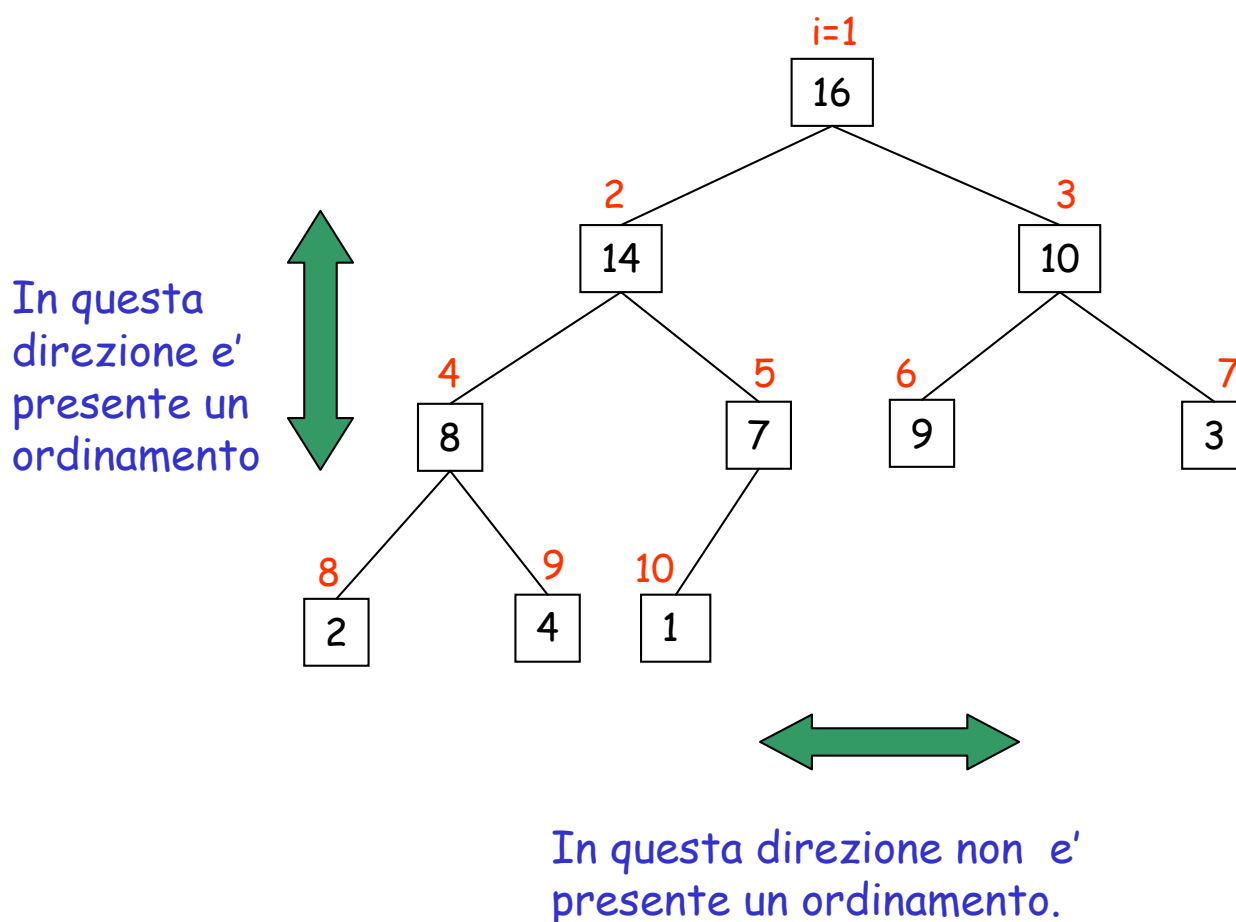


L'albero e' **quasi completo** \rightarrow Completo su tutti i livelli tranne eventualmente sul livello piu' basso che e' riempito da sinistra.

Altezza di un nodo \rightarrow lunghezza del piu' lungo cammino discendente dal nodo ad una foglia.

Proprieta' di uno heap

Notiamo che:



Proprieta' di ordinamento parziale dello heap:

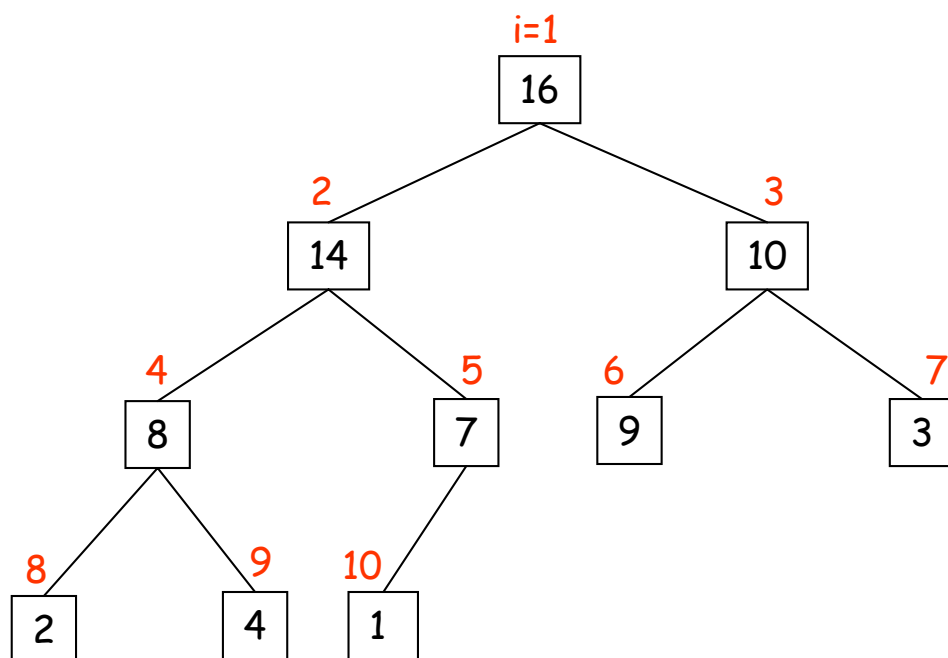
Per ogni nodo i diverso dalla radice:

$$A(\text{parent}[i]) \geq A[i]$$

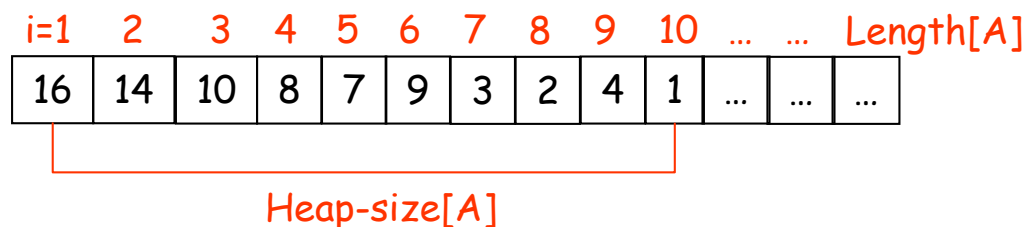
Da cio' segue che l'elemento piu' grande dello heap e' memorizzato nella radice.

Come rappresentare uno heap

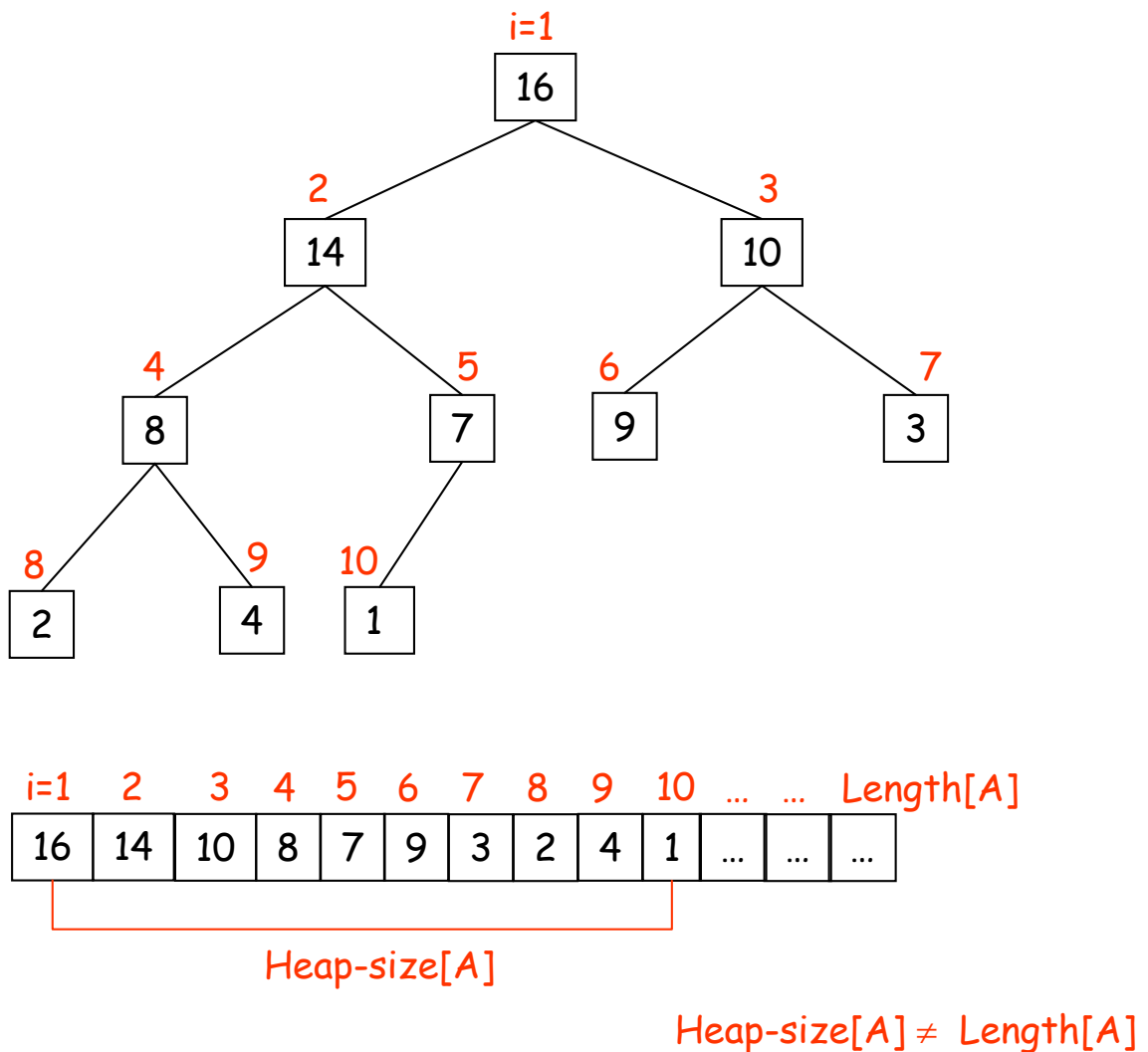
Albero binario - Insieme di elementi su cui e' definita una relazione di "parentela".



Un albero binario quasi completo puo' essere descritto da un **array** in cui il figlio sinistro ed il figlio destro di un nodo i si trovano nelle posizioni $2i$ e $2i+1$ rispettivamente.



N.B. Stiamo assumendo che l'indice i parta da 1.



Per muoverci all'interno dell'albero definiamo le seguenti procedure:

Parent(i)
return $\lfloor i/2 \rfloor$

Left(i)
Return $2i$

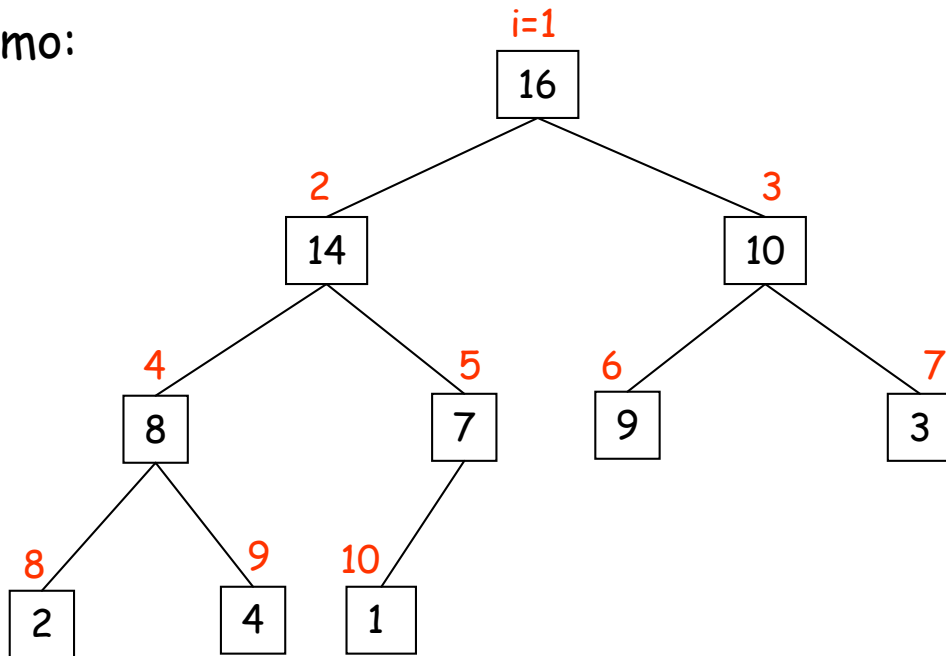
Right(i)
Return $2i+1$

Affinche' un array A che rappresenta un albero binario sia uno heap deve valere la relazione:

$$A(\text{parent}[i]) \geq A[i] \quad \text{per ogni } i \neq 1$$

Mostreremo che un array generico puo' essere ordinato trasformandolo in un heap ed eseguendo operazioni che hanno tempo di esecuzione al piu' proporzionale alla altezza h dello heap e quindi impiegano un tempo $O(\log(n))$.

Notiamo:



h = altezza albero binario

Se l'albero e' completo:

$$N_{\text{nodi}} = 1 + 2 + 2^2 + \dots + 2^h = 2^h * (1 + (1/2) + \dots + (1/2)^h) = 2^h * (2 - (1/2)^h) = 2^{h+1} - 1$$

Se l'albero non e' completo:

$$2^h - 1 < N_{\text{nodi}} < 2^{h+1} - 1 \rightarrow h = \Theta(\log(n))$$

Algoritmo heapsort - strategia

L'idea di base e' la seguente:

- ✓ Trasformiamo un array A di dimensione n in un heap di dimensione n (creazione dello heap);
 - ✓ Per le proprieta' dello heap, sappiamo che l'elemento massimo della sequenza iniziale e' memorizzato in $A[1]$. Scambiamo $A[1]$ con l'elemento $A[n]$;
 - ✓ La sequenza che $A[1].....A[n-1]$ che otteniamo non e' piu' uno heap ($A[1]$ e' fuori posto). Permutiamo i suoi elementi in moto da trasformarla in un heap di dimensione $n-1$ (mantenimento dello heap);
 - ✓ Per le proprieta' dello heap, sappiamo che l'elemento massimo della sequenza $A[1].....A[n-1]$ e' memorizzato in $A[1]$. Scambiamo $A[1]$ con l'elemento $A[n-1]$;
 - ✓
 - ✓
- Ripetendo $n-1$ volte otteniamo una sequenza ordinata.

Le operazioni che effettuiamo sono:

Al **primo** step:

- o Creazione di un heap $O(????)$
- o Scambio di due elementi $O(1)$ (temporale e spaziale)

Agli step **$k=1,2,...n-1$** :

- o Mantenimento dello heap $O(????)$
- o Scambio di due elementi $O(1)$ (temporale e spaziale)

Poiche' abbiamo $n-1$ iterazioni se vogliamo

$T(n)=O(n\log(n))$:

- Creazione dello heap $O(n \log(n))$
- Mantenimento dello heap $O(\log(n))$

Procedura di mantenimento dello heap

Supponiamo che A sia uno heap. Alteriamo il valore di $A[1]$. L'array che otteniamo non e' piu' un heap. I sottoalberi con radice in $A[\text{right}(1)]$ ed $A[\text{left}(1)]$ sono ancora heap. Dobbiamo scrivere una procedura che permuti gli elementi $A[1], A[2], \dots, A[\text{heap-size}[A]]$ in modo da ricostruire uno heap.

```
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 scambia(A[i] , A[largest])
        Heapify(A, largest)
```

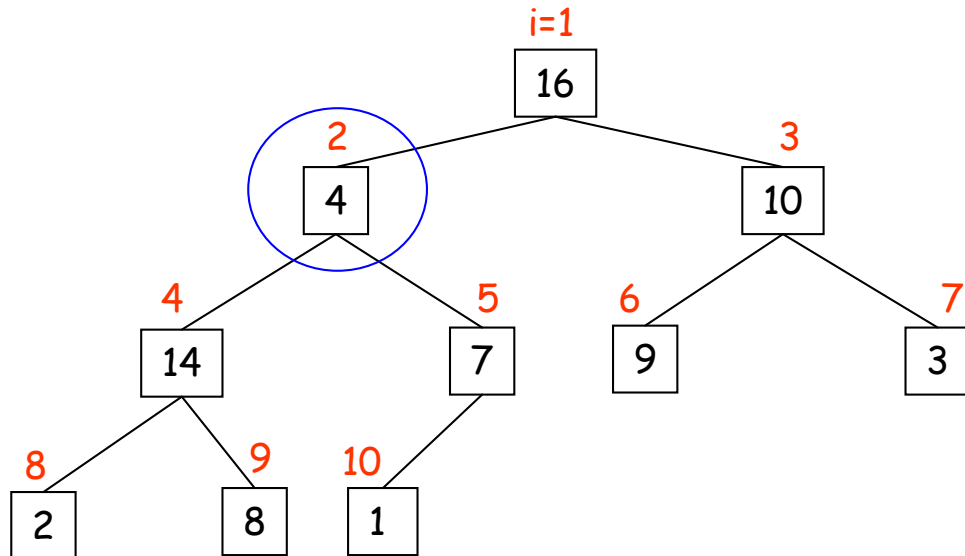
La procedura Heapify fa "scendere" $A[i]$ nella prima posizione utile.

Ipotesi - I sottoalberi con radice in $A[\text{right}(i)]$ ed $A[\text{left}(i)]$ sono heap.

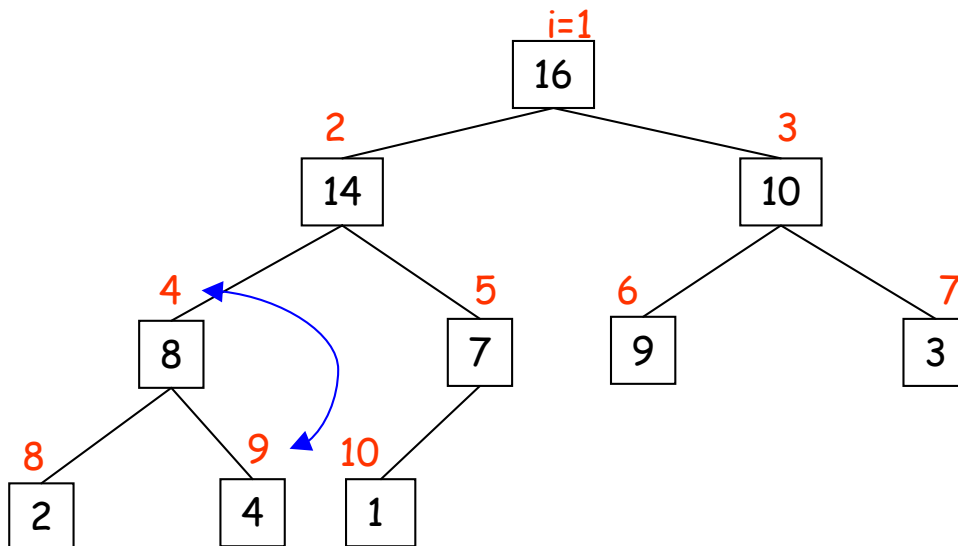
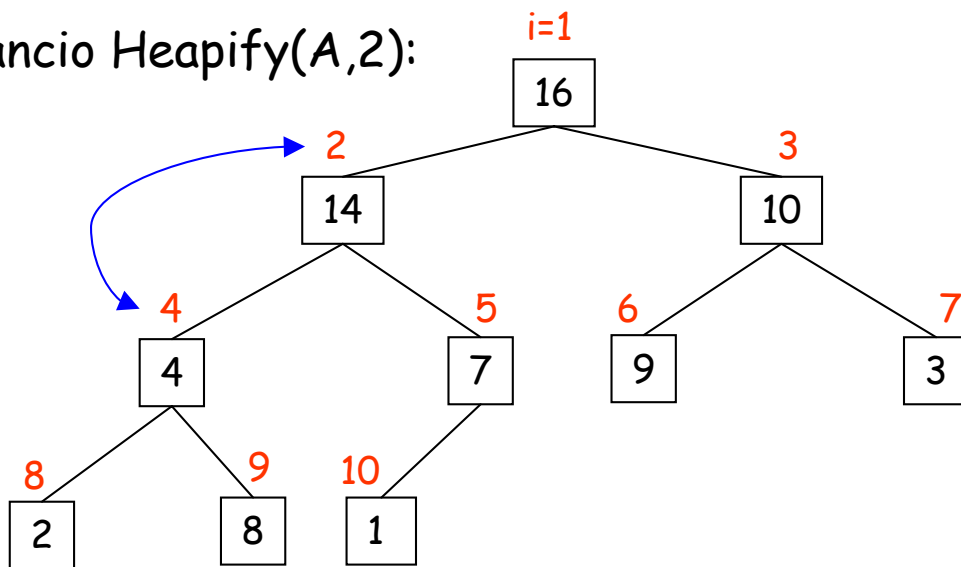
Tesi - Qualunque sia il valore $A[i]$, per effetto della procedura $\text{Heapify}(A,i)$, l'albero con radice in $A[i]$ diviene uno heap.

Tempo di esecuzione di $\text{Heapify}(A,i) \rightarrow O(h_i)$
 h_i = altezza nodo i

Heapify - esempio



Lancio Heapify(A,2):



Procedura di costruzione di uno heap

La procedura Heapify puo' essere usata in modo bottom-up per convertire un array $A[1...n]$, in uno heap di lunghezza n .

```
Build-heap(A)
heap-size[A] ← lenght[A]
For i ← ⌊ lenght[A]/2 ⌋ down to 1
    do Heapify(A,i)
```

Build-heap "attraversa" i nodi non foglia dell' heap, dal basso verso l'alto ed esegue heapify su ciascuno di essi.
→ Ad ogni step le ipotesi necessarie per applicare heapify sono automaticamente verificate.

Tempo di esecuzione di build-heap?

$$T(n) \approx \sum_{\text{NODI}} T_{\text{HEAPIFY}} = \sum_{h=0}^H N_{\text{nodi}}(h) \cdot T_{\text{HEAPIFY}}(h) = \sum_{h=0}^H N_{\text{nodi}}(h) \cdot O(h)$$

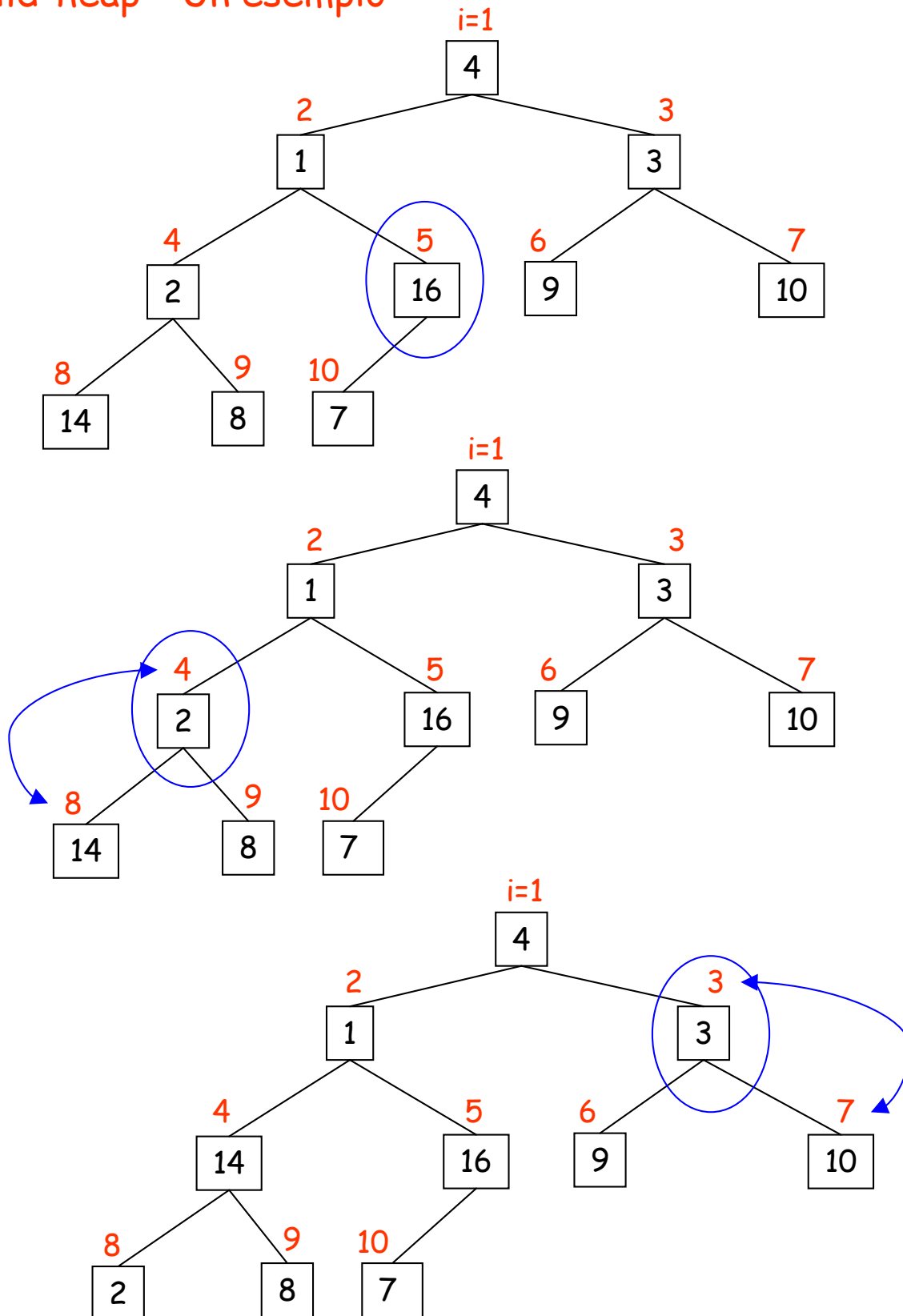
H = altezza albero $\approx \log_2(n)$

$N_{\text{nodi}}(h)$ = numero di nodi con altezza $h \approx n/2^{h+1}$

$$T(n) \leq C \cdot \sum_{h=0}^H \frac{n}{2^{h+1}} \cdot h \leq C \cdot n \cdot \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} = C \cdot n = O(n)$$

Notare - Si puo' costruire un heap da un array non ordinato in tempo lineare

Build-heap - Un esempio



E così via ...

Algoritmo Heapsort

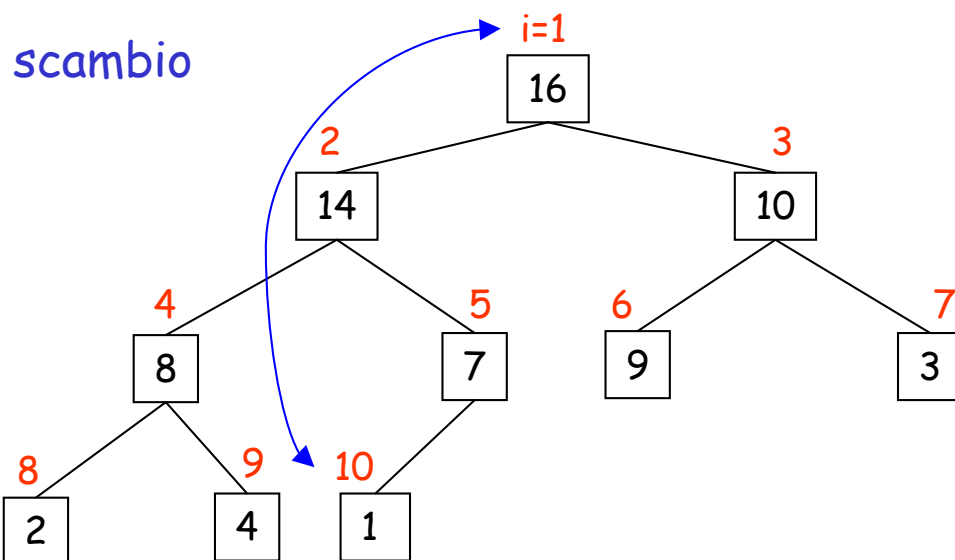
```
Heapsort(A)
Build-heap(A)                                O(n)
For i ← length(A) down to 2                  O(n)
do scambia(A[1], A[i])                       O(n)
    Heapsize[A] ← Heapsize[A] - 1            O(n)
    Heapify(A, 1)                             O(n log(n))
```

Tempo di esecuzione → $O(n \log(n))$
Ordinamento in loco

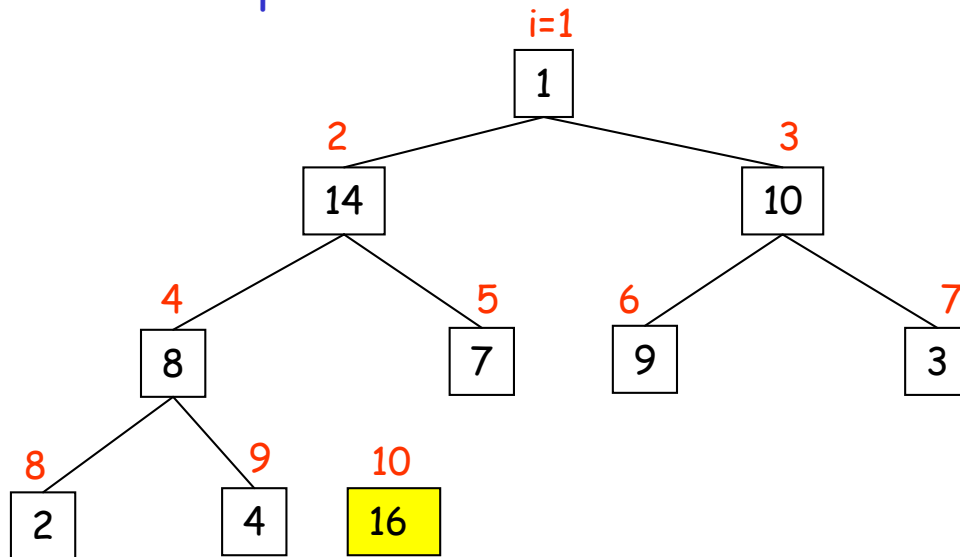
Esempio

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

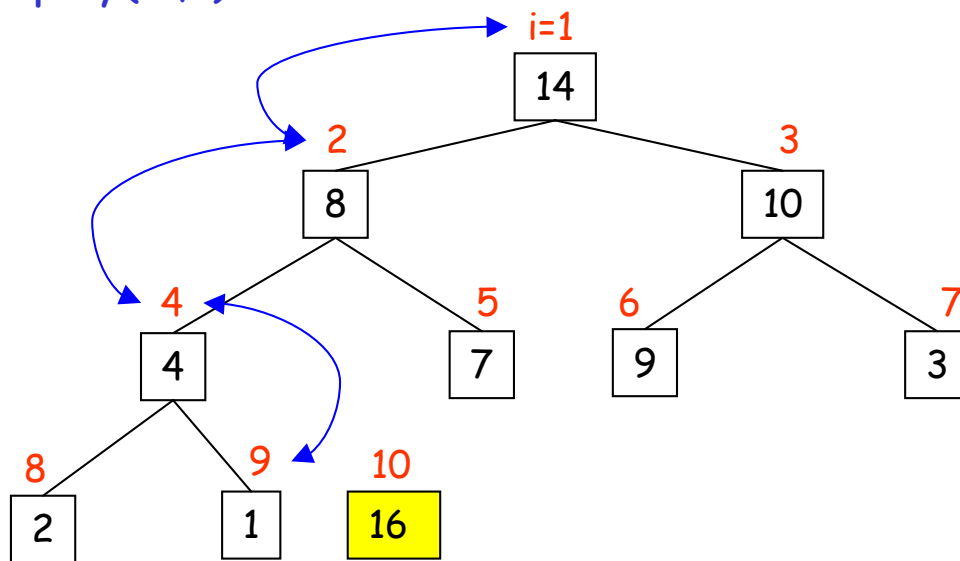
Build-heap(A) → $A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Heap-size = heap-size - 1



Heapify(A,1)



E così via