

Algoritmi e Strutture Dati

Capitolo 4 Ordinamento: Heapsort

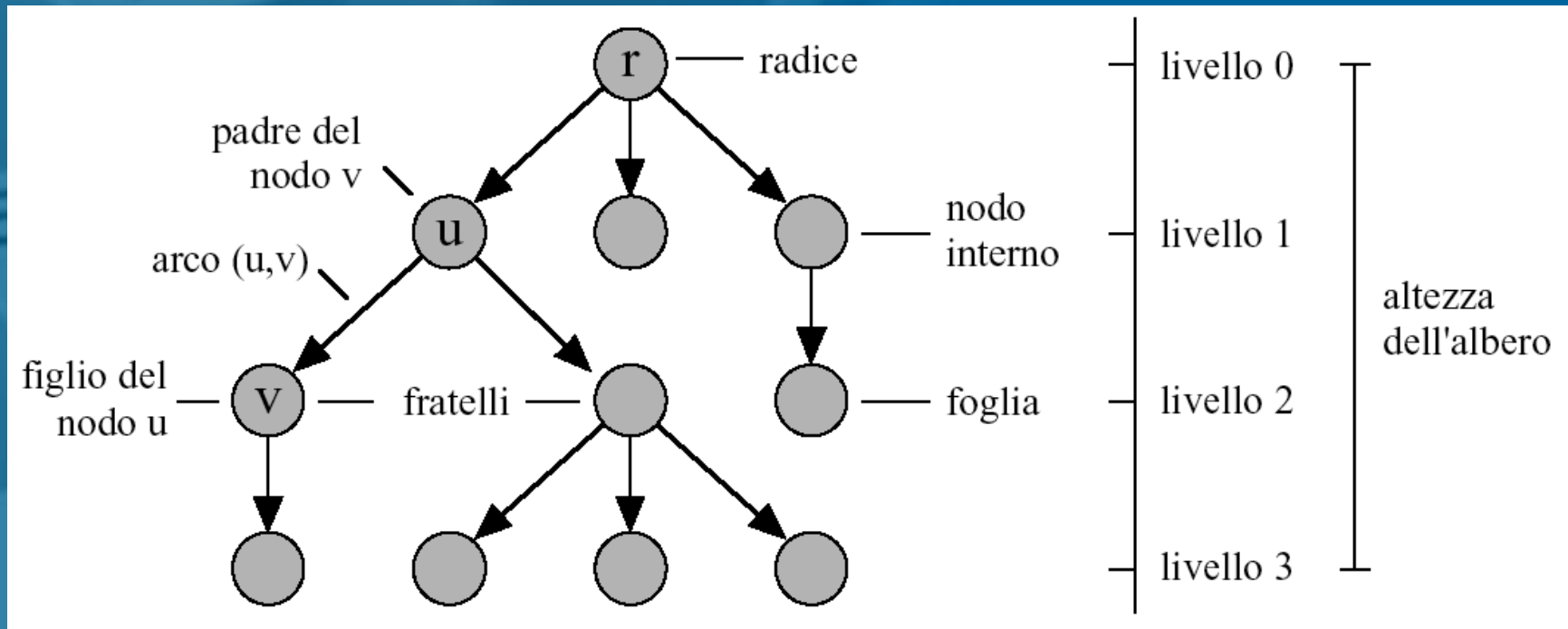
Punto della situazione

- Problema dell'ordinamento:
 - Lower bound – $\Omega(n \log n)$ Albero di decisione
 - Upper bound – $O(n \log n)$ Mergesort (**non in loco e complessità $\Theta(n \log n)$**)
 - Algoritmi quadratici: Insertion, Selection (**in loco**)
- Proviamo a costruire un nuovo algoritmo ottimo, che ordini **in loco e che costi $O(n \log n)$**)

HeapSort

- Stesso approccio incrementale (invertito) del **SelectionSort**
 - seleziona gli elementi dal più grande al più piccolo...
 - ... ma usa una **struttura dati efficiente (heap binario)**, per cui l'estrazione del prossimo elemento massimo avviene in tempo $O(\log n)$, invece che $O(n)$
- **Struttura dati (efficiente)**
 - Organizzazione specifica (e memorizzazione) di una collezione di dati che consente di supportare le **operazioni previste** su di essi usando meno risorse di calcolo possibile
- **Obiettivo**: progettare una struttura dati **H** su cui eseguire efficientemente le seguenti operazioni:
 - dato un array **A**, **genera H**
 - **estrai** il più grande elemento da **H**
 - **ripristina** l'organizzazione specifica dei dati in **H** (ovvero mantieni **invariate** le proprietà strutturali di **H**)

Alberi: qualche richiamo



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

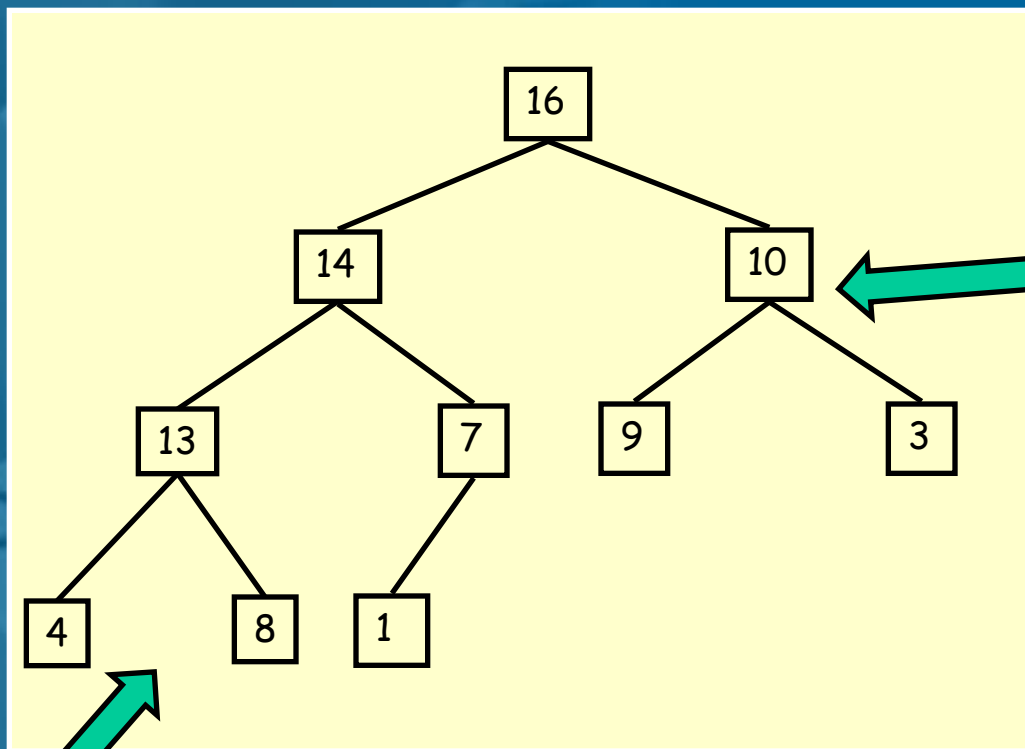
$d=2 \rightarrow$ **albero binario**

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

Heap Binario

- Struttura dati **heap (catasta) binario** associata ad un insieme totalmente ordinato **S**: albero binario radicato con le seguenti proprietà:
 - 1) **Quasi completo**, ovvero completo fino al penultimo livello, con tutte le foglie sull'ultimo livello 'compattate' a sinistra
 - 2) gli elementi di **S** sono memorizzati nei nodi dell'albero (ogni nodo **v** memorizza uno e un solo elemento di **S**, denotato con **chiave(v) ∈ S**)
 - 3) per ogni nodo **v** dell'albero, **chiave(padre(v)) ≥ chiave(v)** (proprietà di **ordinamento parziale** dell'heap)

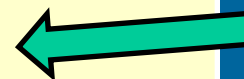
...un esempio



In questa direzione **è** presente un ordinamento



Tutti i livelli tranne al più l'ultimo sono completi



Le foglie dell'ultimo livello sono tutte compattate a sinistra dell'albero



In questa direzione **non è** presente un ordinamento



Proprietà salienti degli heap

- 1) Ogni nodo interno contiene un valore maggiore o uguale del valore contenuto in **tutti i suoi discendenti** (deriva banalmente dalla proprietà di ordinamento parziale)
⇒ L'elemento **massimo** è contenuto **nella radice**
- 2) L'albero binario associato ad un heap di **n** elementi ha **altezza $\Theta(\log n)$**

Osservazione

- La struttura dati presentata è più propriamente denominata **max-heap**, per via del fatto che il **massimo** è contenuto nella radice
- In alcuni contesti che vedremo più avanti (ad esempio, algoritmi su grafi), avrà più senso definire la struttura duale **min-heap**, in cui la relazione di ordine parziale diventa:

chiave(padre(v)) ≤ chiave(v) per ogni nodo v

e conseguentemente la radice conterrà il **minimo**.

Altezza logaritmica di un heap binario

- Abbiamo già dimostrato (vedi LB per il problema dell'ordinamento) che un albero binario con **k foglie** in cui ogni nodo interno ha (al più) due figli, ha **altezza** $h(k) \geq \log k$.
- Adesso vogliamo dimostrare che un albero binario **quasi completo** di **n** nodi, ha altezza **$h := h(n) = \Theta(\log n)$**
- Ma se l'albero fosse completo di altezza **h**:

$$n = 1 + 2 + 2^2 + \dots + 2^{h-1} + 2^h =$$

(somma parziale **h**-esima della **serie geometrica** di ragione **2**)

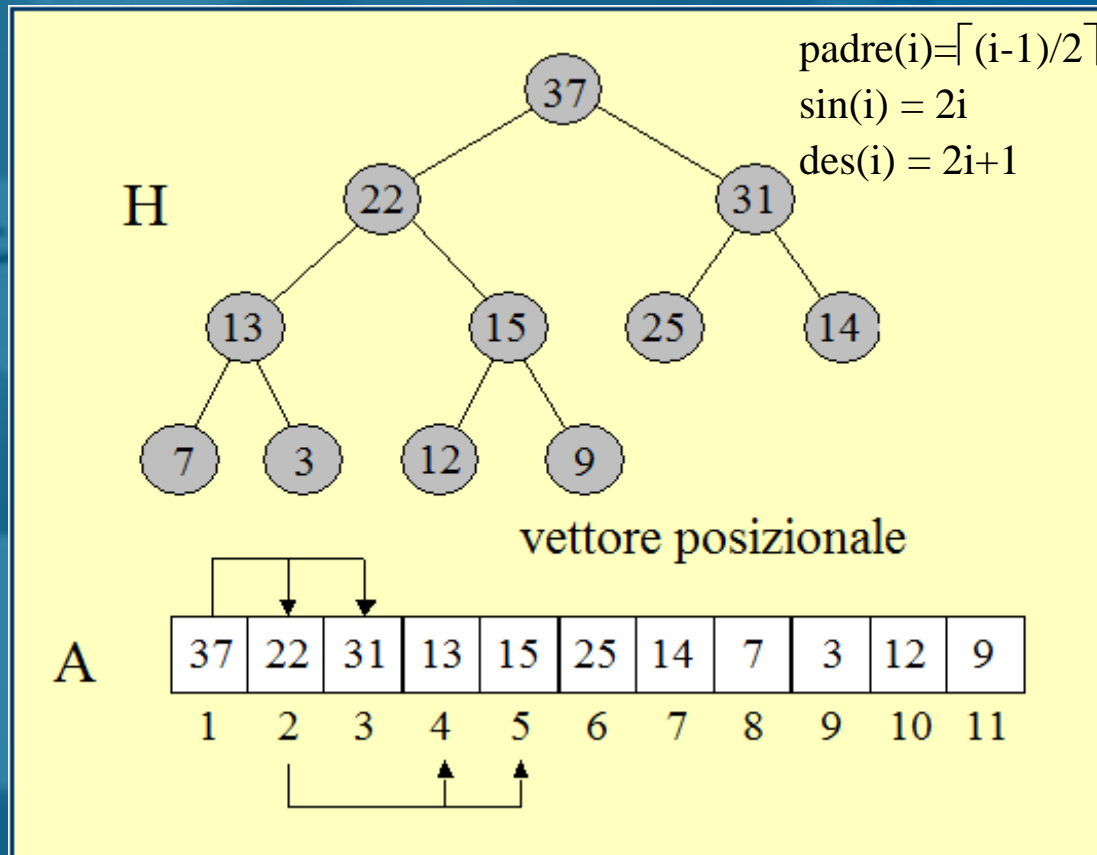
$$= (2^{h+1} - 1) / (2 - 1) = 2^{h+1} - 1$$

e quindi se fosse completo di altezza **h-1** avremmo $n = 2^h - 1$

⇒ Quindi, se l'albero è **quasi completo**:

$$2^h - 1 < n \leq 2^{h+1} - 1 \quad \Rightarrow \quad h = \lfloor \log n \rfloor \quad \Rightarrow \quad h = \Theta(\log n)$$

Rappresentazione con array posizionale



Se l'heap inizialmente contiene **n** elementi, è sufficiente allocare un vettore di dimensione **n**. In seguito, alcune celle dell'array potrebbero rimanere vuote, perché potremmo cancellare elementi dall'heap attraverso l'operazione di **estrazione del massimo**

⇒ nello pseudocodice il numero di elementi correntemente nell'heap **H** rappresentato mediante l'array posizionale **A** sarà indicato con **heapsize[A]** (può quindi essere minore della dimensione dell'array), la cosiddetta **dimensione logica** dell'array

La procedura **fixHeap**

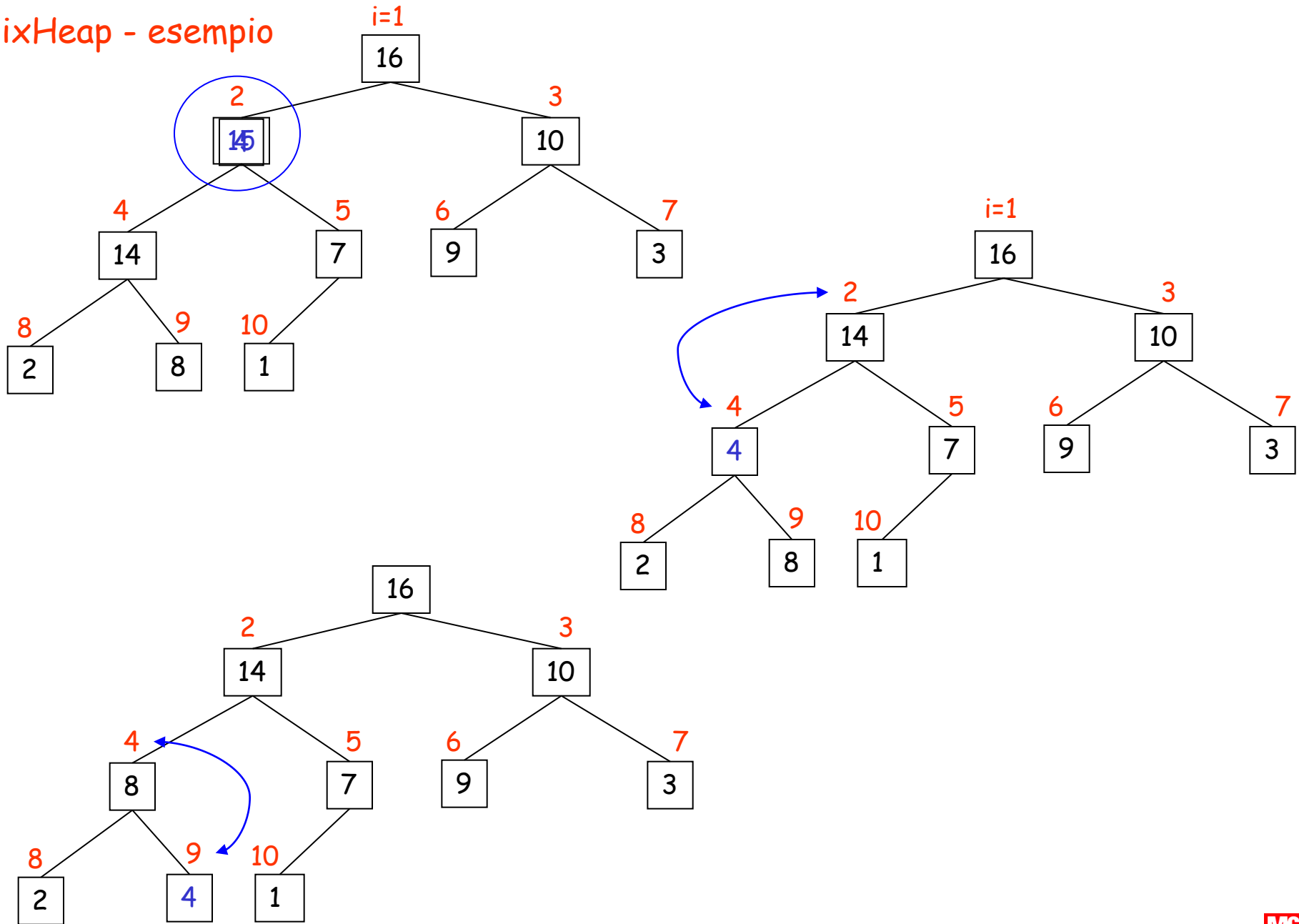
Sia **H** un max-heap dato in forma di albero binario. Supponiamo che la chiave di un certo nodo **v** di **H** venga **decrementata**. In tal caso, potrebbe essere violata la proprietà di ordinamento parziale di **H** (ovvero, la chiave di **v** potrebbe essere diventata **MINORE** di quella di almeno uno dei figli). Allora, possiamo riaggiustare l'heap come segue:

fixHeap(nodo **v**, heap binario **H**)

1. **if** (**v** è una foglia) **then** return
2. **else**
3. sia **u** il figlio di **v** con chiave massima
4. **if** (**chiave**(**v**) < **chiave**(**u**)) **then**
5. scambia **chiave**(**v**) e **chiave**(**u**)
6. **fixHeap**(**u**, **H**)

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

fixHeap - esempio



Pseudocodice di fixHeap per l'array posizionale

fixHeap-posizionale (i, array 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** $\text{massimo} = s$
5. **else** $\text{massimo} = i$
6. **if** ($d \leq \text{heapsize}[A]$ e $A[d] > A[\text{massimo}]$)
7. **then** $\text{massimo} = d$
8. **if** ($\text{massimo} \neq i$)
9. **then** scambia $A[i]$ e $A[\text{massimo}]$
10. **fixHeap-posizionale**($\text{massimo}, A$)

Costruzione dell'heap

Osservazione: **fixHeap** opera solo sul sottoalbero radicato nel nodo su cui viene chiamata, ed assume ovviamente che i due sottoalberi radicati in tale nodo soddisfino invece la proprietà di ordinamento parziale

⇒ posso pensare di costruire un heap applicando ricorsivamente in modo **bottom-up** la procedura di **fixHeap**!

Heapify: Algoritmo ricorsivo basato sul *divide et impera* per la costruzione dell'heap **H** in forma di albero binario.

heapify(albero binario quasi completo H)

1. **if** (H è vuoto) **then** return
2. **else**
3. **heapify**(sottoalbero sinistro di H)
4. **heapify**(sottoalbero destro di H)
5. **fixHeap**(radice di H, H)

heapify-posizionale(array A)

1. $n \leftarrow$ lunghezza di A
2. $i = \lceil n/2 \rceil$
3. **if** ($n=0$) **then** return **else**
4. **heapify-posizionale**(A[1,i-1])
5. **heapify-posizionale**(A[i,n])
6. **fixHeap-posizionale**(i,A)

Versione iterativa di **heapify** con array posizionale

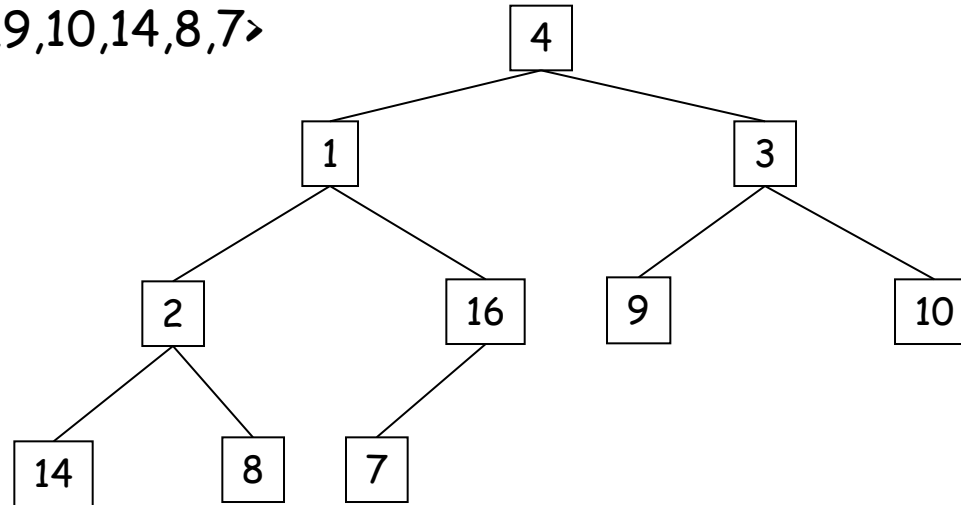
heapify-posizionale-iterativo(array A)

1. Heapsize[A]=n
2. **for** $i=\lfloor n/2 \rfloor$ **down to** 1 **do**
3. **fixHeap-posizionale**(i, A)

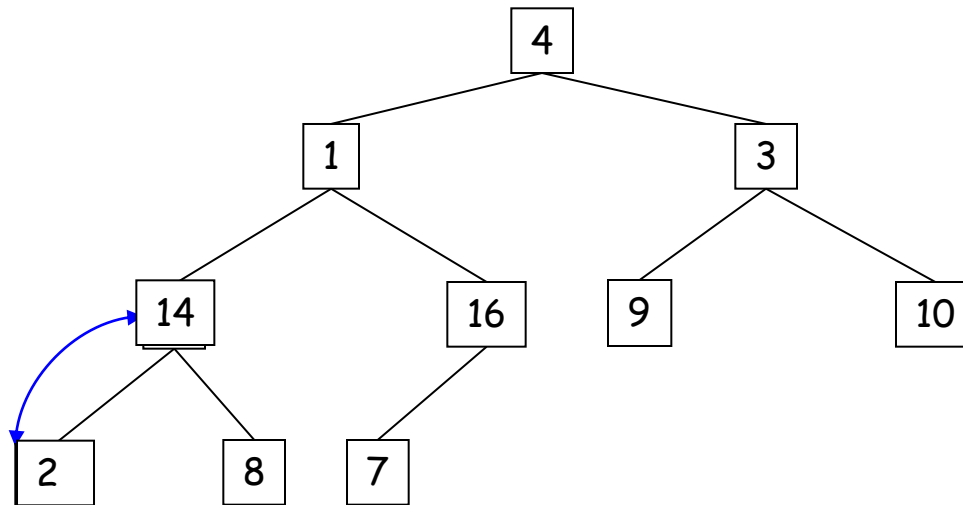
Nota: gli elementi $A[\lfloor n/2 \rfloor + 1], \dots, A[n]$ sono foglie dell'albero

Esempio

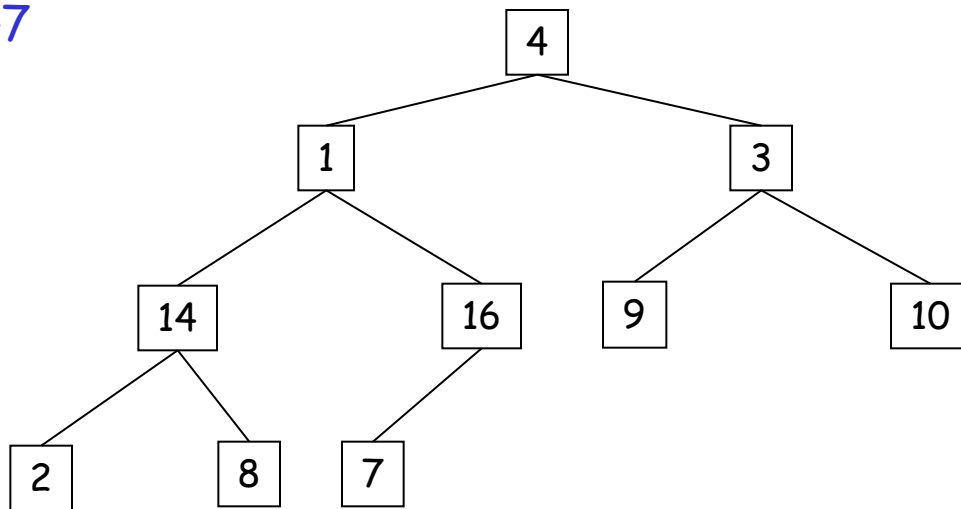
Input: $H = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$



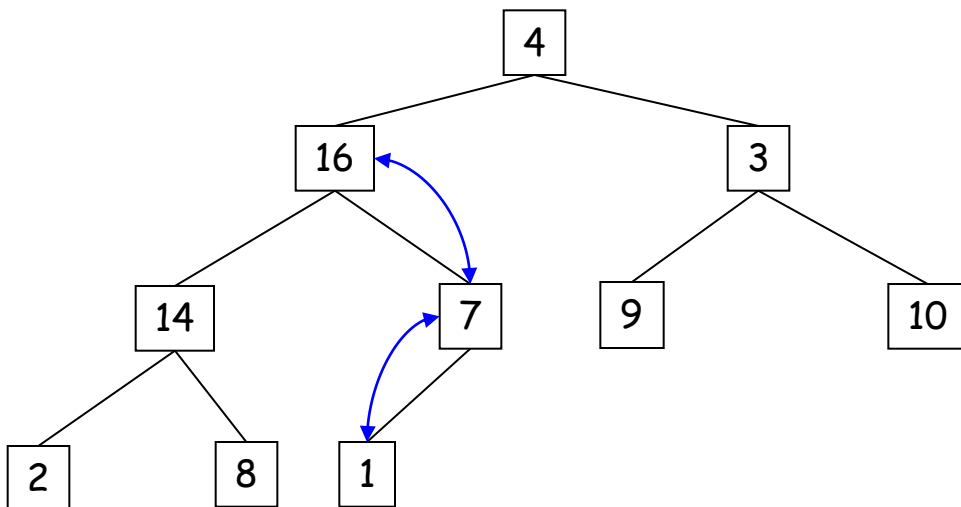
Procedendo bottom-up arrivo fino alle foglie 14 e 8, sulle quali non faccio nulla, e poi applico la fixHeap al nodo 2, che viene scambiato col 14, perché $2 < 14$



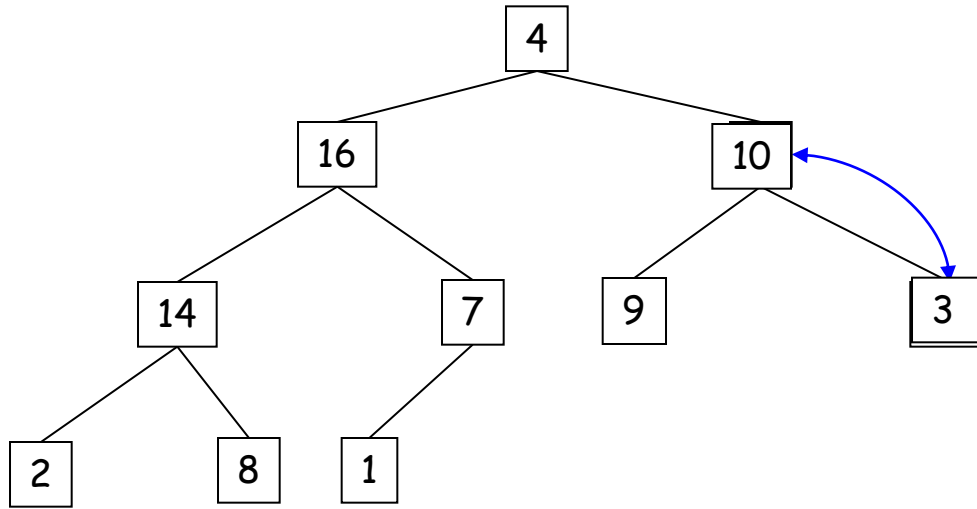
Quindi passo alla foglia 7 e poi al nodo 16, sul quale la fixHeap non fa nulla perché $16 > 7$



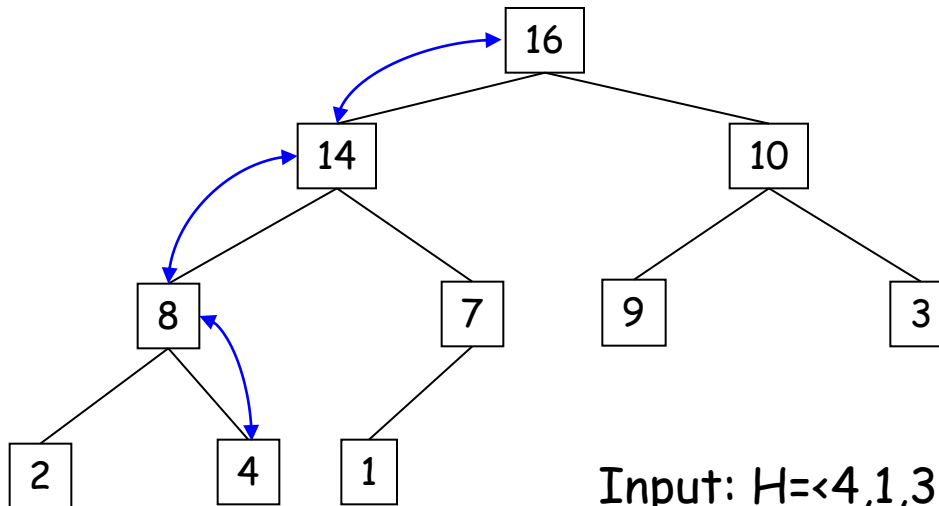
Passo quindi al nodo 1, e lo faccio ridiscendere come in figura



Passo quindi alle foglie 9 e 10, e poi al nodo 3 che viene scambiato col 10



Infine passo al nodo 4, e lo faccio ridiscendere come in figura



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

Complessità heapify

Ci concentriamo sulla versione ricorsiva non posizionale (le altre versioni sono analoghe); per semplificare l'analisi, completiamo l'albero binario aggiungendo le foglie mancanti. In questo modo avremo un totale di n' nodi, con ovviamente $n' = \Theta(n)$

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

➔ $T(n') = \Theta(n') = \Theta(n)$ (caso 1 del Teorema Master: $a=b=2$,
e $f(n) = O(\log n) = O(n^{\log_2 2 - \varepsilon})$ per $\varepsilon > 0$, quindi $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$)

Domanda: Analizzando la procedura **heapify** senza l'ipotesi di cui sopra, quale problema sorge?

Risposta: sorge il problema che una delle due sottosequenze potrebbe in realtà contenere **più di $n/2$** elementi, e quindi non posso impostare la relazione di ricorrenza nella forma del teorema master

Estrazione del massimo

- Leggi e memorizza la chiave contenuta nella radice
 - Copia nella radice la chiave contenuta nella foglia più a destra dell'ultimo livello
 - **nota:** nella rappresentazione posizionale, è l'elemento in posizione $\text{heapsize}[A]$
 - Rimuovi la foglia e **diminuisci di 1** la dimensione dell'heap
 - Ripristina la proprietà di ordinamento a heap richiamando **fixHeap** sulla radice
- Tempo di esecuzione: $O(\log n)$ (n è la dimensione corrente dell'heap)

L'algoritmo HeapSort

- Costruisce un heap (in forma di array posizionale) 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 (array A)

1. **heapify-posizionale(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-posizionale**(`1`, A)

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

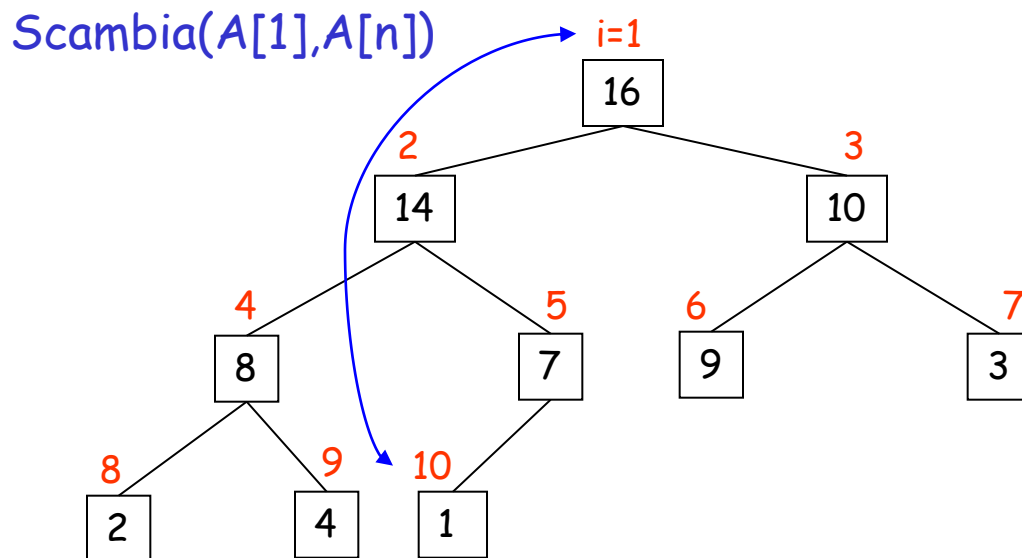


ordina in loco in tempo $O(n \log n)$

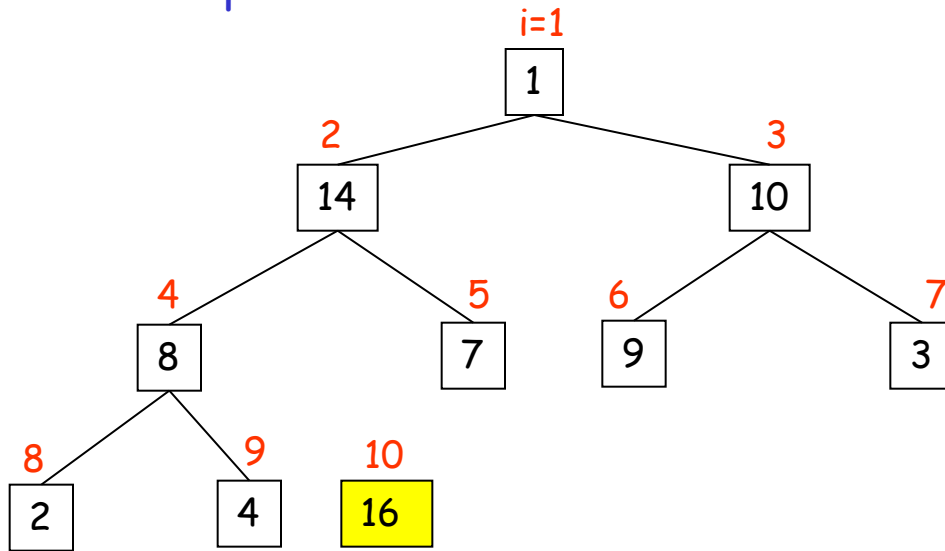
Esempio

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

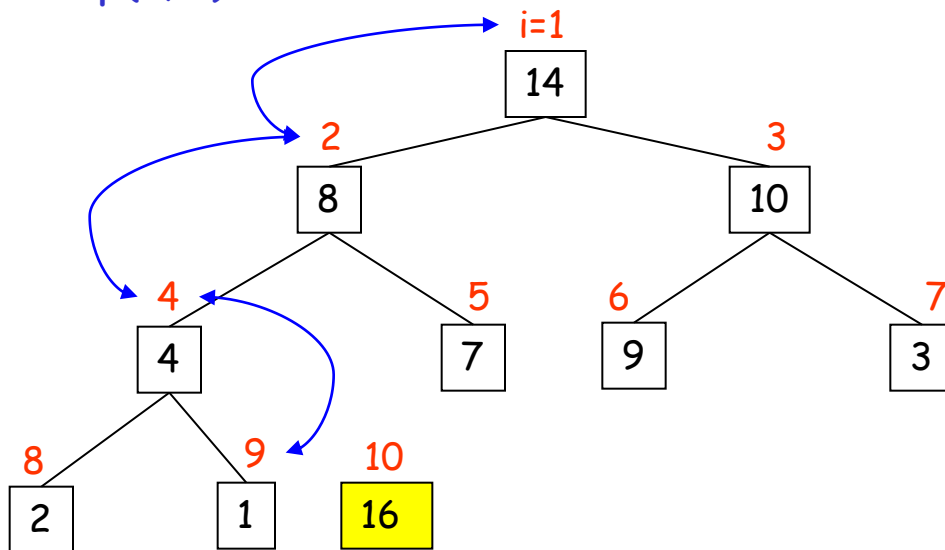
Heapify(A) $\rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



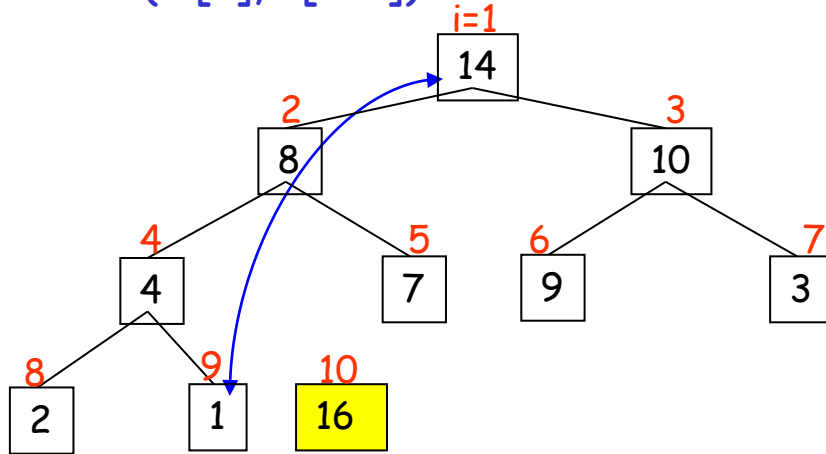
heapsize = heapsize - 1



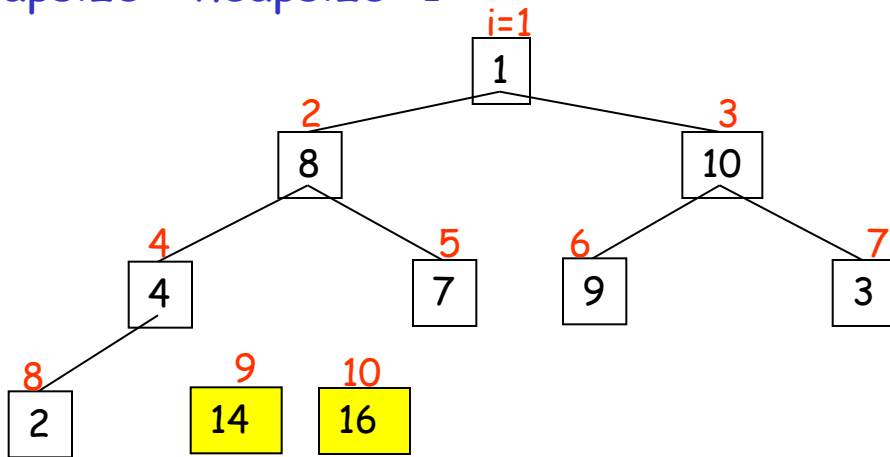
fixHeap(1,A)



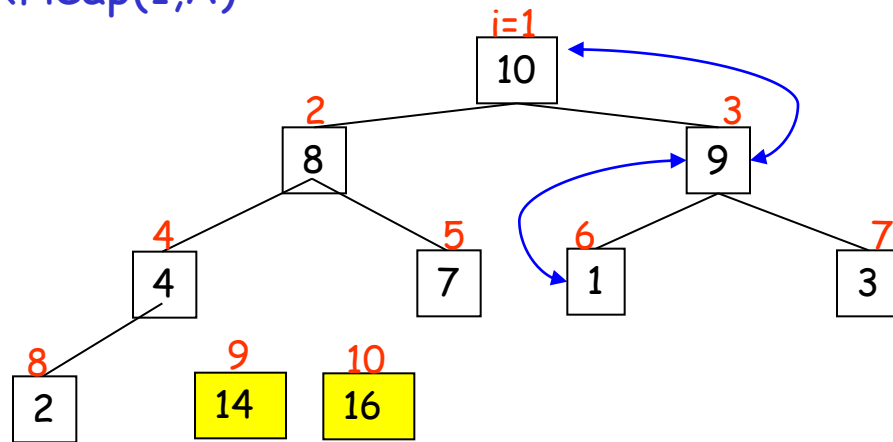
Scambia(A[1],A[n-1])



heapsize = heapsize - 1

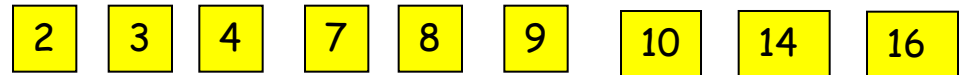


fixHeap(1,A)



E così via, sino ad arrivare a

1



Esercizio: È possibile definire un'istanza di input su cui l'HeapSort costa $o(n \log n)$?

Risposta: NO (se gli elementi sono distinti): Si può dimostrare che nel caso migliore l'HeapSort richiede circa $\frac{1}{2} n \log n$ operazioni ☹️ \Rightarrow l'HeapSort costa $\Theta(n \log n)$