Algoritmi e Strutture Dati (12/10/2021)

* Ordinamento
  - → altri algoritmi
  - → limite inferiore $\Omega(n \log n)$
  - → ordinamento in tempo lineare

* definizione
  INPUT: $a_1, \dots, a_n$ sequenza di interi
  OUTPUT: $a_1', \dots, a_n'$ permutazione t.c. $i \leq j \Rightarrow a_i' \leq a_j'$

* Insertion Sort
  - ⇝ incrementale
  - ⇝ complessità $\Theta(n^2)$

* Merge Sort
  - → divide et impera
  - → complessità $\Theta(n \log n)$

* Complessità in spazio?

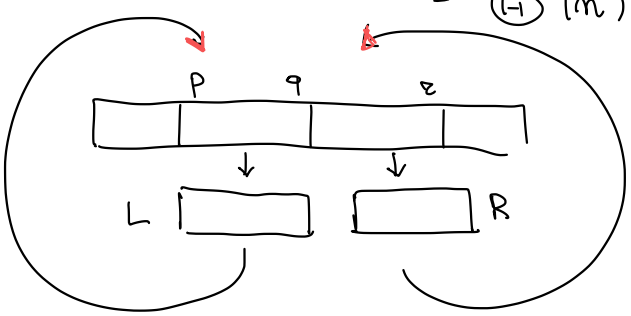
  * Insertion sort

    spazio $O(1)$ "in loco"

  * MergeSort: indichiamo con n = # elem. array

    * merge: $\Theta(n)$
    * mergesort $M^{MS}(n) = \max\left\{M^{MS}\left(\frac{n}{\lfloor 2 \rfloor}\right), M^{MS}\left(\frac{\lceil n \rceil}{2}\right), \Theta(n)\right\}$

      $= \Theta(n)$



ESERCIZIO 1 : MergeSort con allocazione statica dell'array di supporto

```
MergeSort ( A , n)
    alloca  B[1..n]
    MergeSortRec (A, B, 1, n)
```

ESERCIZIO 2 : Evitare la copia delle due parti ordinate delle quali fare il merge

ESERCIZIO 3 : MergeSort in loco

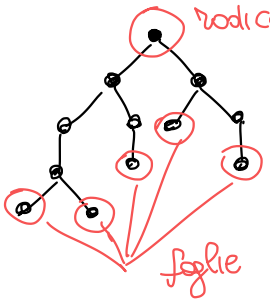
* HeapSort

↝ complessità $O(n \lg n)$

→ in loco

↳ struttura dati heap (code con priorità)

* Alberi



h altezza : lunghezza del cammino più lungo radice → foglie

(4)

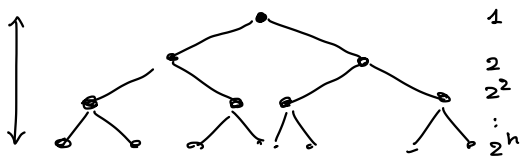
binario : ogni nodo ha max 2 figli

ordinati : se binari figlio sx e figlio dx

albero binario completo :

→ ogni nodo non foglia ha due figli

→ ogni cammino radice → foglia ha la stessa lunghezza



$$\#\text{nodi} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$\sum_{i=0}^{h} 2^i = x$$

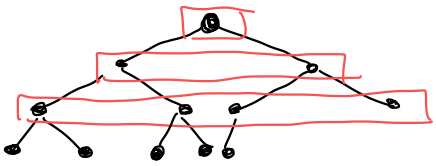
$$2x = 2\sum_{i=0}^{h} 2^i = \sum_{i=0}^{h} 2^{i+1}$$

$$= \sum_{j=1}^{h+1} 2^j = x - 2^0 + 2^{h+1}$$

$$x = -2^0 + 2^{h+1}$$
$$= 2^{h+1} - 1$$

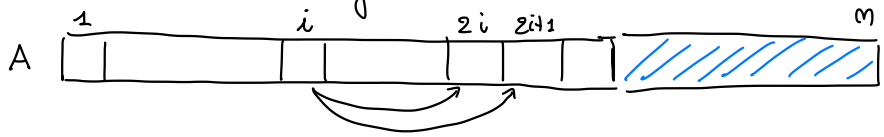
alberi binari quasi completi

ogni livello completo, tranne eventualm. l'ultimo con foglie tutte a sx



* **Heap**: albero binario ordinato quasi completo

implementato come array



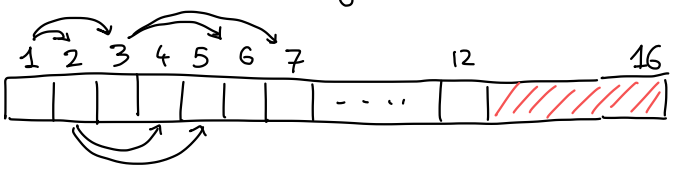
→ A[1] è radice

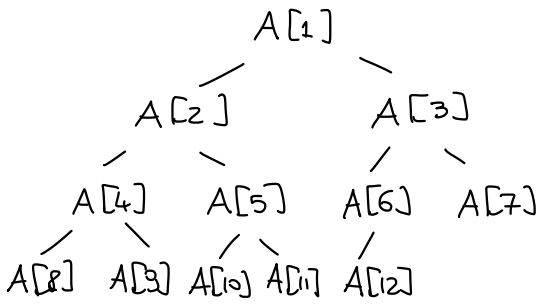
→ nodo A[i] figlio sinistro A[2i]
figlio destro A[2i+1]

parent A[⌊i/2⌋]

A.length , A.size
↑ ↑ spazio occupato dallo heap

Esempio A[1..16] A.length = 16 A.size = 12





Parent(i)
return ⌊i/2⌋

Left(i)
return i*2

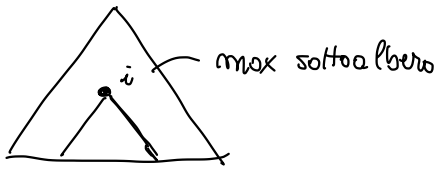
Right(i)
return i*2+1

* Max Heap

è uno heap

→ ogni elemento è ≥ discendenti

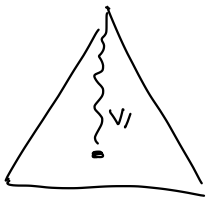
$(\forall i \quad A[i] \geq A[Left(i)], A[Right(i)])$



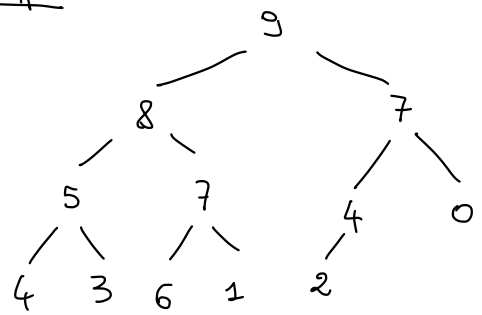
equivalentemente

→ ogni elemento è ≤ degli antenati

$(\forall i \quad A[i] \leq A[Parent(i)])$



Esempio:



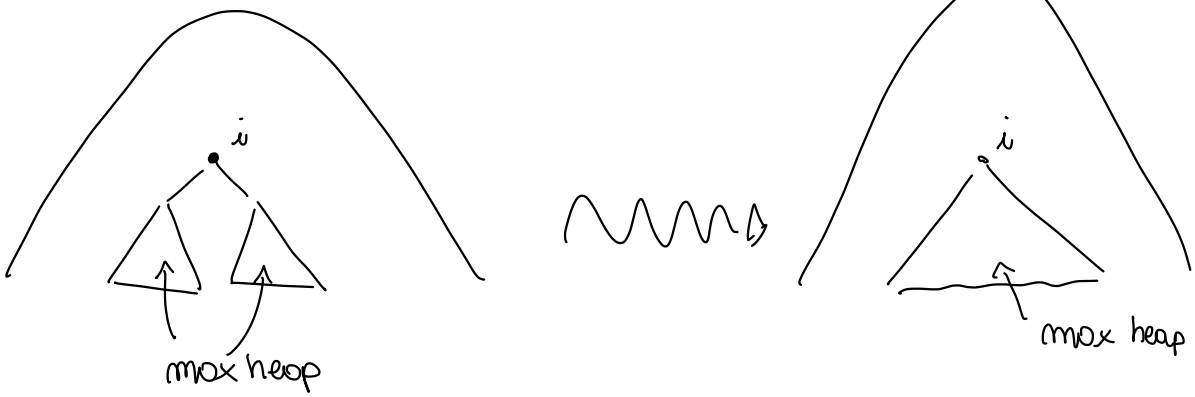
9	8	7	5	7	4	0	4	3	6	1	2

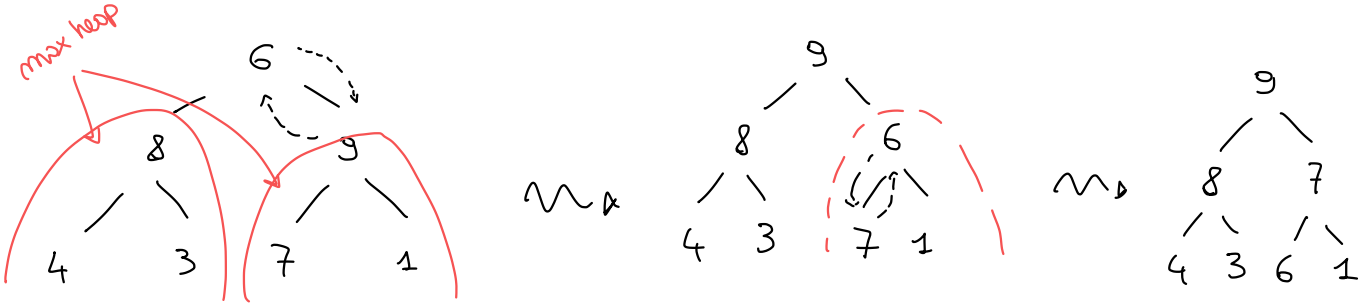
* Come si ottiene un max-heap ?

2 osservazione

→ un albero con un solo nodo è un maxheap

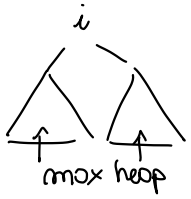
→ dato uno heap





MaxHeapify (A, i)

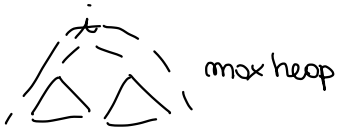
IN: A heap
i nodo di riferimento



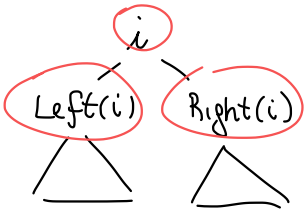
l = Left(i)
r = Right(i)

if ($l \leq$ A.size) and
(A[l] > A[i])
max = l
else
max = i

OUT:



if ($r \leq$ A.size) and
(A[r] > A[max])
max = r

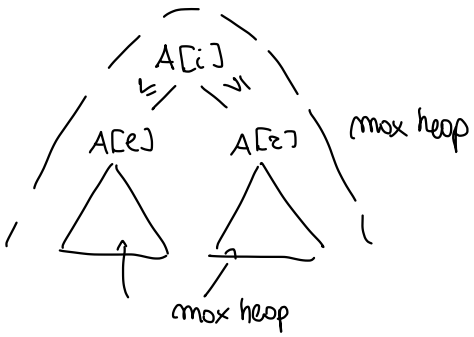


if max ≠ i
A[i] ↔ A[max]
MaxHeapify (A, max)

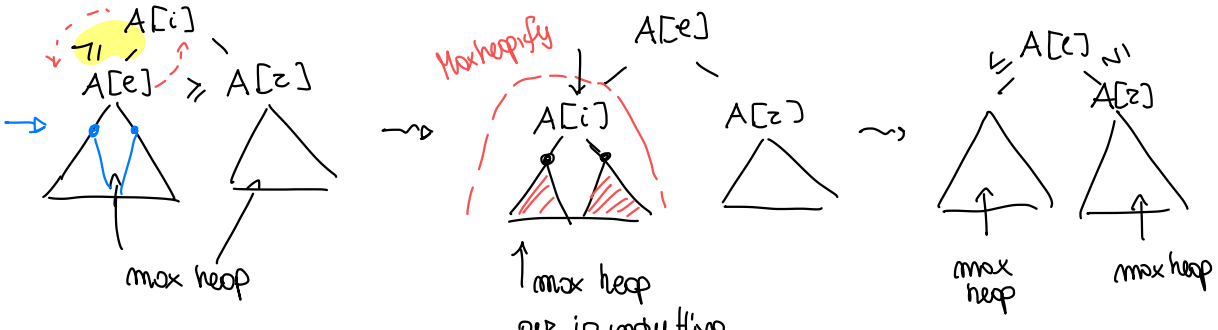
* Correttezza ?

base: max = i

- i foglia
- i non foglia
  A[i] ≥ A[l], A[r]

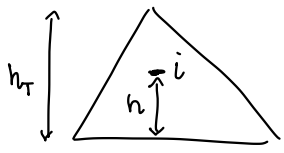


caso induttivo:
supponiamo che max = l



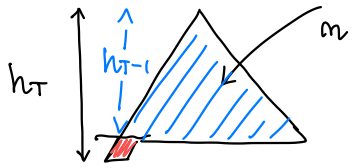
* Complessità ?

(1)



$h \leq h_T$ $\quad O(h_T)$

se il heap contiene n elementi ?



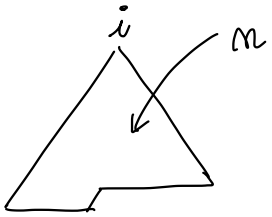
$$n = 2^{(h_T - 1) + 1} - 1 + 1$$

$$= 2^{h_T}$$

$$h_T \leq \log_2 n$$

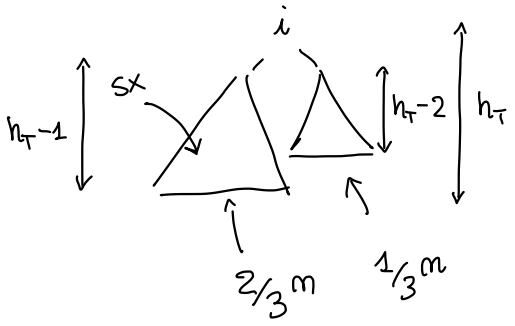
⟶ complessità $O(\log n)$

② ricorrenza



$$T(n) = T\left(\tfrac{2}{3}n\right) + c$$

$a = 1$ $b = \frac{3}{2}$ $f(n) = c$



master theorem

$f(n) = c$ $\quad n^{\log_b a} = n^{\log_{\frac{3}{2}} 1}$

$= n^0 = 1$

$f(n) = c = (1)\ (n^{\log_b a}) = (1)(n^0)$

CASO 2

$T(n) = (1)\ (n^{\log_b a} \cdot \log n)$

n^0

$= (1)\ (\log n)$