Corso di laurea in Informatica
Algoritmi 1
A.A. 2025/26

Il problema dell'ordinamento: L'Heapsort

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Sommario

Un algoritmo di ordinamento efficiente: Heapsort

Uso della struttura dati heap per ordinare

Pseudocodice e costo computazionale

Heapsort (1)

L'algoritmo ***heapsort*** è piuttosto complesso ma esibisce ottime caratteristiche:

- come Merge sort ha un costo computazionale di *O(n log n)* anche nel caso peggiore
- come Selection sort ordina in loco.

Sfrutta una opportuna organizzazione dei dati, ossia una ***struttura dati***, che garantisce una o più specifiche proprietà, il cui mantenimento è *essenziale* per il corretto funzionamento dell'algoritmo.

Struttura dati Heap

Heapsort (2)

Fase 1:

- Trasforma un array A di dimensione n in un heap di n nodi, mediante la funzione `Build_heap`.
- Ora il max dell'array è in A[0] e, per metterlo nella corretta posizione dell'ordinamento, basta scambiarlo con A[n-1].

Heapsort (3)

Fase 2:

- La dimensione dell'heap viene ridotta ad (n – 1), e:
  - i due sottoalberi della radice sono ancora degli heap
  - solo la nuova radice (ex foglia più a destra) può violare la proprietà del nuovo heap
- Ripristina la proprietà di heap sui residui (n – 1) elementi usando la funzione `Heapify`;

Riapplica il procedimento riducendo via via la dimensione dell'heap a (n – 2), (n –3), ecc., fino ad arrivare a 2.

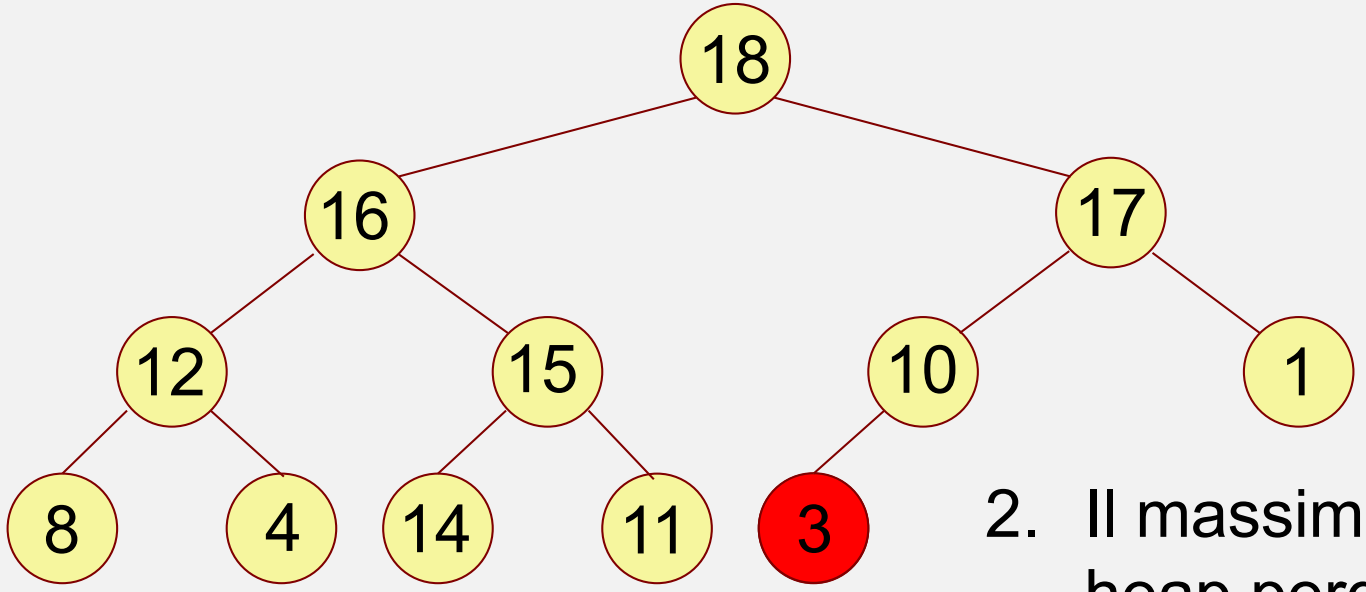
Heapsort (4)

Esempio: Prima iterazione, `Heap_size`= 12

1. Scambio del massimo:

18	16	17	12	15	10	1	8	4	14	11	3

3. Lavoro di `Heapify` (su 11 elementi):



2. Il massimo è a posto, lo heap perde un elemento

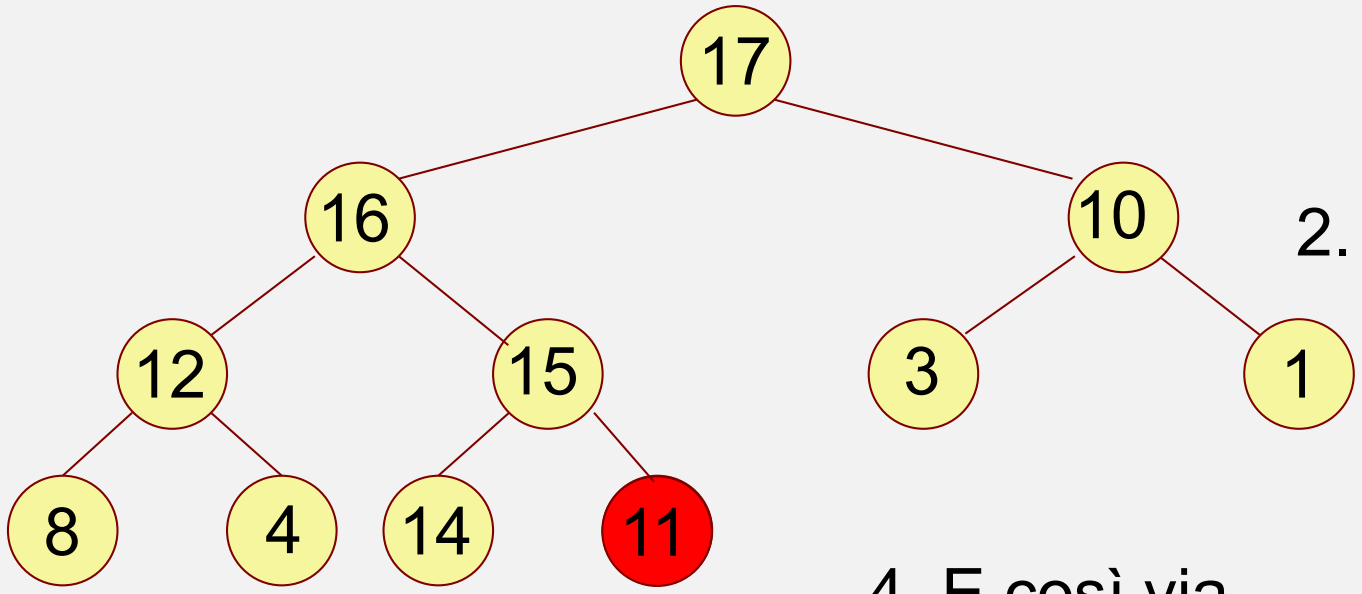
Heapsort (5)

Segue esempio: Seconda iterazione, `Heap_size`= 11

1. Scambio del massimo:

17	16	10	12	15	3	1	8	4	14	11	18

3. Lavoro di `Heapify` (su 10 elementi):



2. Il massimo è a posto, lo heap perde un elemento

4. E così via…

Heapsort (6)

Vediamo ora lo pseudocodice di `Heapsort`:

```
def Heapsort (A):
  Build_heap(A)                                  O(n)
  for x in reversed(range(1,len(A))):            (n-1) iteraz.
    A[0],A[x]=A[x],A[0]                            Θ(1)
    Heapify(A, 0, x)                               O(log n)
```

$$T(n) = O(n) + (n-1)(\Theta(1) + O(\log n)) = O(n \log n)$$

Heapsort (7)

L'algoritmo Heapsort chiude tutto il percorso fatto partendo dagli algoritmi naif, passando per il teorema sulla limitazione inferiore al costo degli algoritmi di ordinamento per confronti, arrivando a questo algoritmo che mantiene le proprietà di:

- essere efficiente (costo O(n log n) nel caso peggiore
- ordinare in loco