Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2023/24

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

Esercizio svolto- 1

Data una matrice $m \times n$, si vogliono rimescolare i suoi elementi in modo che tutte le righe e tutte le colonne siano ordinate in modo non decrescente.

Soluzione 1. Consideriamo la matrice come un unico lungo array e ordiniamolo con un algoritmo efficiente:

21	13	2	8
12	7	5	17
3	9	19	14

21	13	2	8	12	7	5	17	3	9	19	14

2	3	5	7	8	9	12	13	14	17	19	21

2	3	5	7
8	9	12	13
14	17	19	21

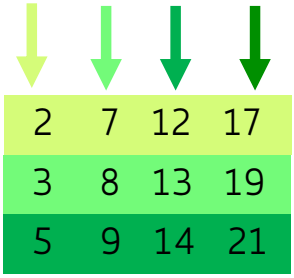
$T(n)=\Theta(nm \log nm)=$
$=\Theta(nm \log n+nm \log m)$

Esercizio svolto- 2

Soluzione 2. Ordino ogni riga ed ogni colonna separatamente con un algoritmo efficiente:

21	13	2	8
12	7	5	17
3	9	19	14

2	8	13	21
5	7	12	17
3	9	14	19



$T(n)=\Theta(m\ n \log n + n\ m \log m)=\Theta(nm \log n+nm \log m)$

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

La struttura dati Heap (1)

La struttura dati *heap* è stata introdotta per eseguire in modo efficiente una sequenza di operazioni di:

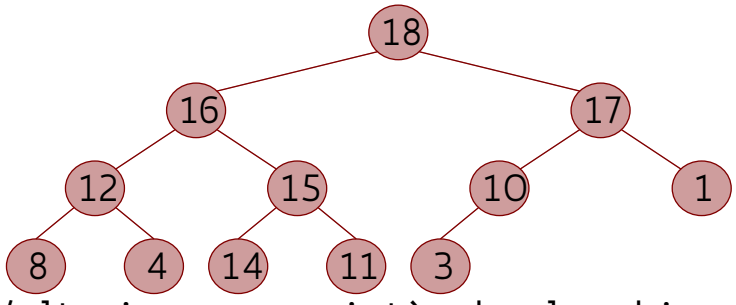
- estrazione (=ricerca+rimozione) del massimo
- inserimento e cancellazione.

Infatti, per compiere queste operazioni, ripristinando la struttura perché sia pronta ad un'altra operazione, il tempo richiesto è:

	Estraz. max	Inserim. di una chiave	Cancellaz. di una chiave
Array qualunque	$\Theta(n)$	O(1)	O(1)
Array ordinato	O(1)	O(n)	O(n)
heap	O(log n)	O(log n)	O(log n)

La struttura dati Heap (2)

Uno *heap* è un albero binario *completo o quasi completo*, ossia un albero binario in cui tutti i livelli sono pieni, tranne l'ultimo, i cui nodi sono addensati a sinistra…



… con l'ulteriore proprietà che la chiave di ogni nodo è *maggiore o uguale* alla chiave dei suoi figli (proprietà di ordinamento verticale).

La struttura dati Heap (3)

Il modo più naturale per memorizzare uno heap è utilizzare un array *A*, con indici che vanno da *0* fino al numero di nodi dell'heap, ***heap_size***, diminuito di 1; gli elementi possono infatti essere messi in corrispondenza con i nodi dell'heap:

La struttura dati Heap (4)

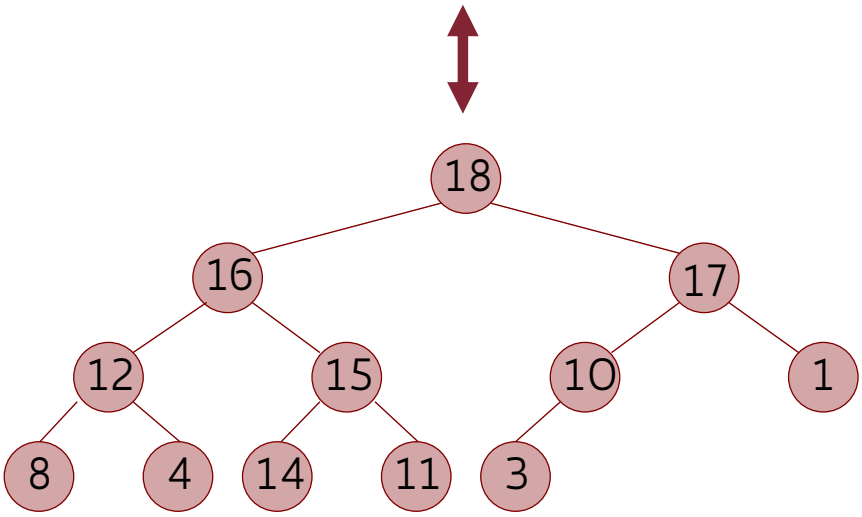
- l'array è riempito a partire da sinistra; se contiene più elementi del numero *heap_size* di nodi dell'albero, allora i suoi elementi di indice *> heap_size* non fanno parte dell'heap;
- ogni nodo dell'albero binario corrisponde a uno e un solo elemento dell'array *A*;
- la radice dell'albero corrisponde ad *A[0]*;
- …

La struttura dati Heap (5)

- …
- il figlio sinistro del nodo che corrisponde ad *A[i]*, se esiste, corrisponde all'elemento *A[2i+1]*: *left(i) = 2i+1*;
- il figlio destro del nodo che corrisponde ad *A[i]*, se esiste, corrisponde all'elemento *A[2i + 2]*: *right(i) = 2i+2*;
- il padre del nodo che corrisponde ad *A[i]* corrisponde all'elemento A[⌊(i-1)/2⌋]: *parent(i) =* ⌊(i-1)/2⌋.

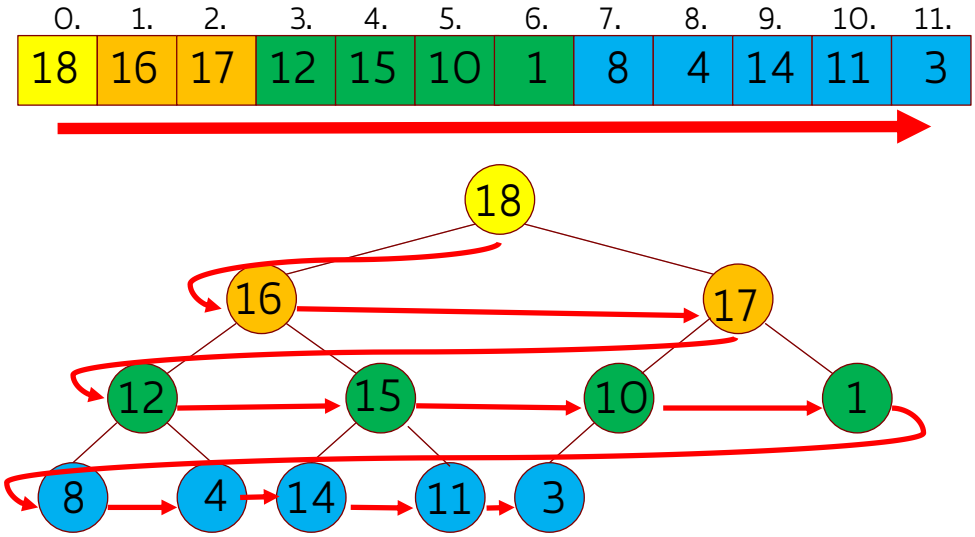
La struttura dati Heap (6)

0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
18	16	17	12	15	10	1	8	4	14	11	3



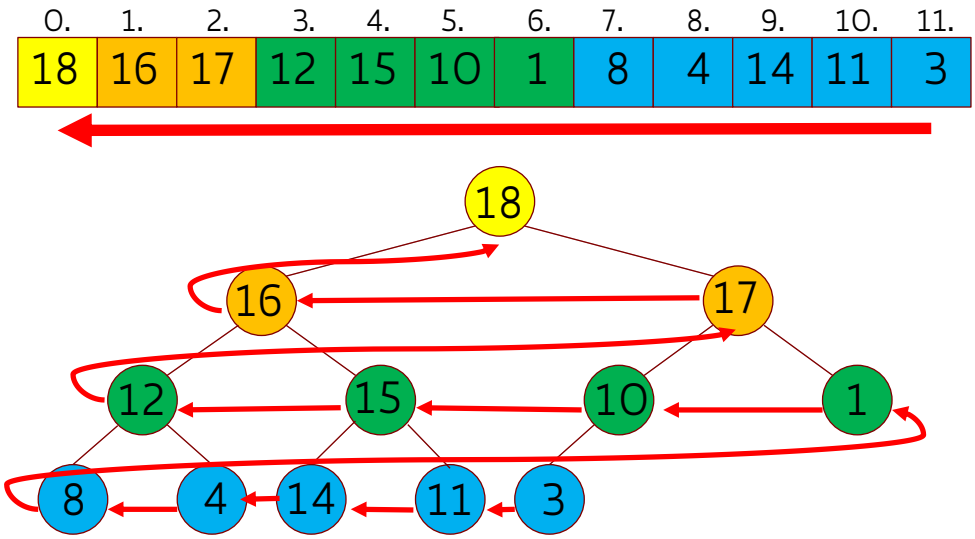
La struttura dati Heap (7)

Scorrere l'array da sinistra a destra corrisponde a muoversi sull'albero per livelli, dall'alto verso il basso e da sinistra a destra



La struttura dati Heap (8)

Simmetricamente, scorrere l'array da destra a sinistra corrisponde a muoversi sull'albero per livelli, dal basso verso l'alto e da destra a sinistra in ciascun livello



La struttura dati Heap (9)

Proprietà:

- Poiché lo heap ha tutti i livelli completamente pieni tranne al più l'ultimo, la sua altezza è $\Theta(\log n)$.
- Con l'implementazione tramite array, la proprietà di ordinamento verticale implica che per tutti gli elementi tranne $A[0]$ (che non ha genitore) vale:

$$A[i] \leq A[parent(i)].$$

- L'elemento massimo risiede nella radice, quindi può essere trovato in tempo $O(1)$.

Funzione ausiliarie

L'algoritmo Heapsort si avvale di due funzioni ausiliarie, necessarie per il suo corretto funzionamento:

- Funzione Heapify
- Funzione Buildheap

Illustreremo tali funzioni prima di descrivere l'algoritmo Heapsort.

Funzione Heapify (1)

La funzione Heapify mantiene la proprietà di ordinamento verticale, sotto l'ipotesi che nell'albero su cui viene fatta lavorare sia garantita la proprietà di heap per entrambi i sottoalberi (sinistro e destro) della radice. Di conseguenza, l'unico nodo che può violare la proprietà di heap è la radice dell'albero, che può essere minore di uno o di entrambi i figli.

...

Funzione Heapify (2)

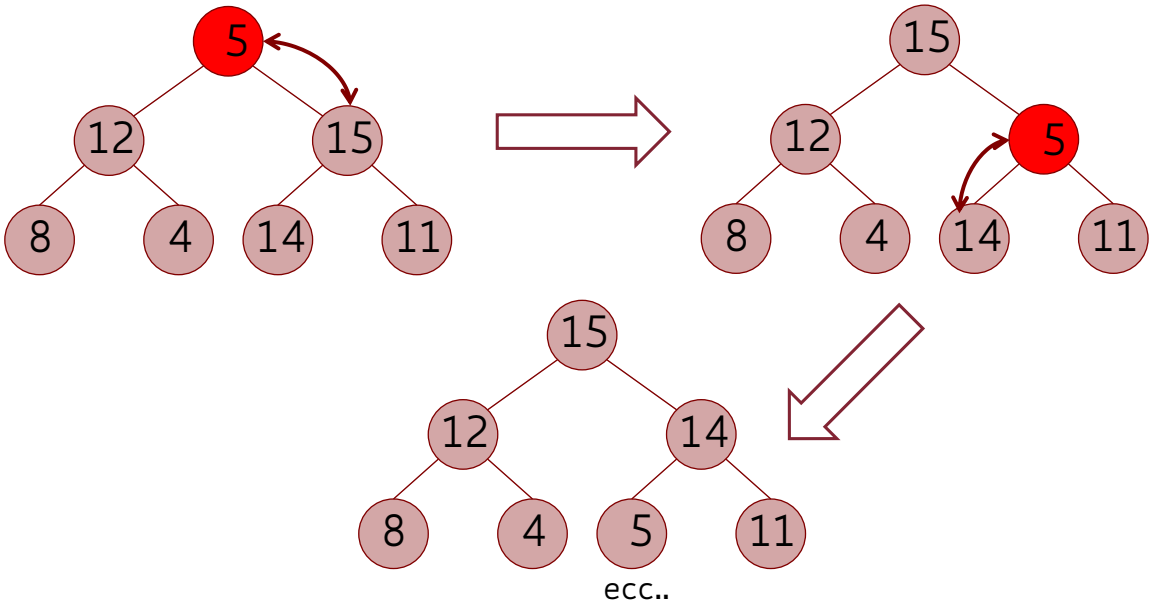
...

La funzione opera sulla radice confrontandola coi suoi figli e, se necessario, la scambia col maggiore di suoi figli.

Dopo lo scambio si verifica se la violazione si sia trasferita sul figlio scambiato e, se necessario, si ripete ricorsivamente l'operazione su tale nodo.

Funzione Heapify (3)

Esempio:



Funzione Heapify (4)

```
def Heapify (A, i, heap_size)
   L=left(i); R=right(i); indice_max=i
   if ((L < heap_size) and (A[L] > A[i])):
      indice_max=L
   if ((R ≤ heap_size) and (A[R] >
                              A[indice_max]))
      indice_max=R
   if (indice_max ≠ i)
      A[i], A[indice_max]=A[indice_max], A[i]
      Heapify (A, indice_max, heap_size)
```

T(h)

Θ(1)

T(h-1)

$T(h) = \Theta(1) + T(h-1)$ e $T(1) = \Theta(1)$.

Sappiamo che $h = \Theta(\log n)$ per cui la soluzione nel caso peggiore è: $T(n) = \Theta(\log n)$. In generale $T(n) = O(\log n)$.

Funzione Buildheap (1)

La funzione Buildheap trasforma un array qualunque di n elementi in uno heap, chiamando ripetutamente Heapify.

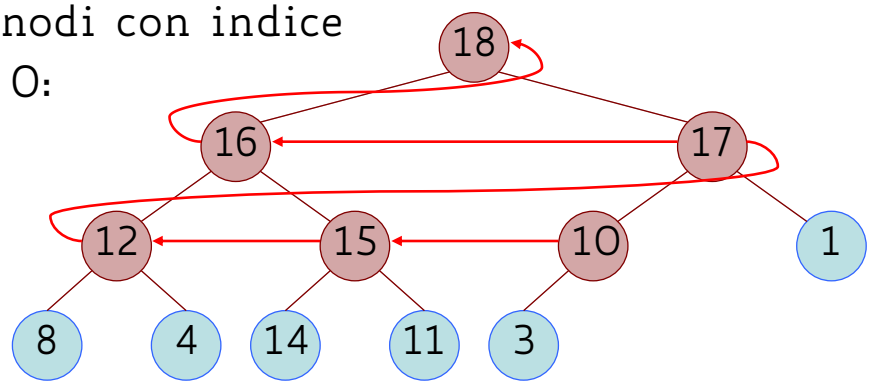
Osservazioni:

- poiché Heapify assume che entrambi i sottoalberi della radice siano heap, va chiamata scorrendo l'albero per livelli dal basso verso l'alto;
- ogni foglia è già uno heap, quindi basta chiamare Heapify a partire dal nodo interno più a destra, che ha indice $\lfloor n/2 \rfloor$

Funzione Buildheap (2)

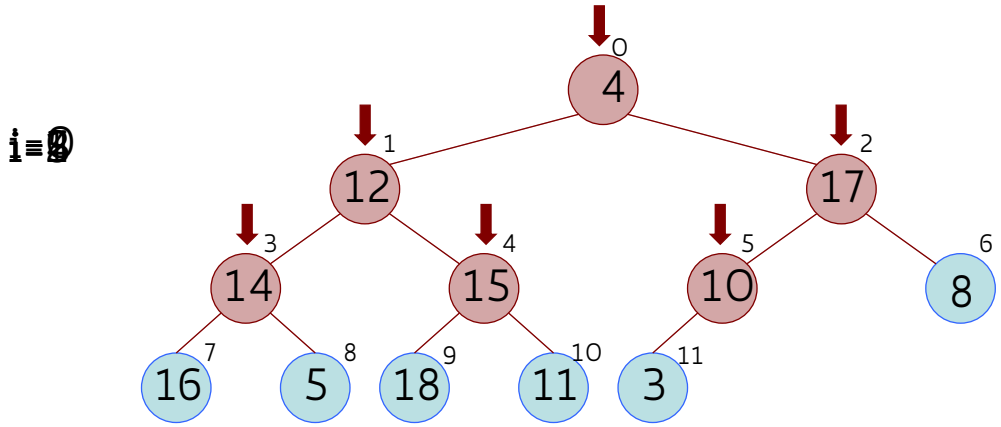
```
def Build_heap (A):
  for i in revered(range(len(A)//2)):
    Heapify (A, i,heap_size)
```

In questo esempio, Buildheap chiama Heapify sui nodi con indice 5, 4, 3, 2, 1, 0:



Funzione Buildheap (3)

```
def Build_heap (A):
   for i in reversed(range(len(A)//2)):
      Heapify (A, i, heap_size)
```



Funzione Buildheap (4)

```
def Build_heap (A):
   for i in reversed(range(len(A)//2)):
      Heapify (A, i, heap_size)
```

La funzione Build_heap effettua $\Theta(n)$ chiamate di Heapify, che sappiamo avere ciascuna costo $O(\log n)$, quindi:

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

Con un calcolo più accurato si può mostrare che $T(n)=\Theta(n)$:

Funzione Buildheap (5)

Se |x| < 1:

$\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$

Mostriamo che *T(n)=O(n)*.

- Il tempo richiesto da Heapify applicata ad un nodo che è radice di un albero di altezza *h* è *O(h)* per quanto già detto;
- il numero di nodi che sono radice di un sottoalbero di altezza h è al massimo $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Quindi: T(n) = $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$

Ricordando che $\sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$, si ha:

$$T(n) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O\left(\frac{n}{2} \sum_{h=0}^{\infty} \left(\frac{1}{2}\right)^h\right) = O\left(\frac{n}{2} 2\right) = O(n)$$

La struttura dati Heap (10)

Abbiamo introdotto la struttura dati HEAP per parlare dell'Heap Sort, ma essa viene utilizzata indipendentemente dagli algoritmi di ordinamento e rientra, come vedremo in seguito, tra le CODE CON PRIORITA'.

E' utile ogni qual volta sia necessario estrarre da un insieme di dati il valore massimo (oppure minimo) ripetutamente.

Per questo:

Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2023/24

Esercizi per casa sugli heap

Esercizi (1)

- Progettare un algoritmo che, dato in input un array che rappresenta uno heap, restituisca il valore minimo. Fare le opportune considerazioni sul costo computazionale.
- Uno *heap minimo* è un albero binario completo o quasi completo con la proprietà che la chiave su ogni nodo è minore o uguale alla chiave dei suoi figli. Si modifichi l'algoritmo di Heap sort in modo che la struttura dati di riferimento sia uno heap minimo e non un heap.

Esercizi (2)

- Progettare un algoritmo che, dato in input un array che rappresenta uno heap ed un elemento x, inserisca x nello heap, collocandolo nella giusta posizione.
  Calcolare il costo computazionale.

Heapsort (1)

E finalmente, vediamo il funzionamento di Heapsort.

Fase 1:

- Trasforma un array A di dimensione *n* in un heap (di *n* nodi), mediante 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 (2)

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 con Heapify;

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

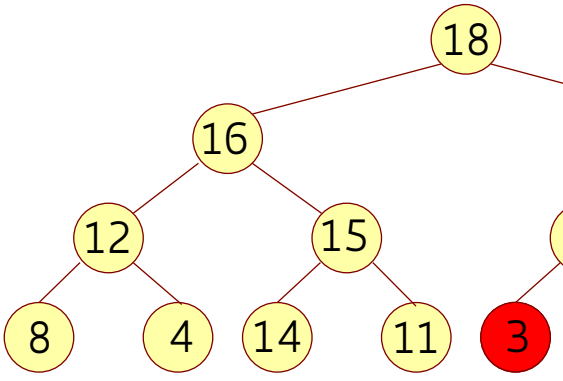
Esempio di Heapsort (1)

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

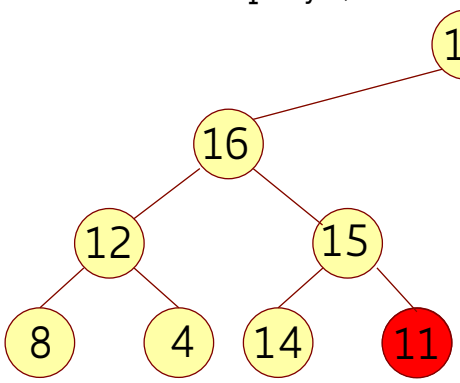
Esempio di Heapsort (2)

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 (9)

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)
```

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