Heap, Heapsort e code con priorità

Matteo Ferrara

Dipartimento di Informatica - Scienza e Ingegneria

matteo.ferrara@unibo.it

Strutture dati

Definizione informale

Una struttura dati è costituita da un insieme di procedure atte ad organizzare i dati sulla memoria dei calcolatori.

Le strutture dati sono di fondamentale importanza in ogni ambito dell'informatica.

Spesso, sono strettamente legate agli algoritmi.
Infatti, organizzare opportunamente i dati che devono essere elaborati da un algoritmo **può incidere in modo significativo sulla sua efficienza**.

Heap (1)

Definizione

Uno **heap** è una struttura dati che memorizza gli elementi di un array in un *albero binario* quasi completo.

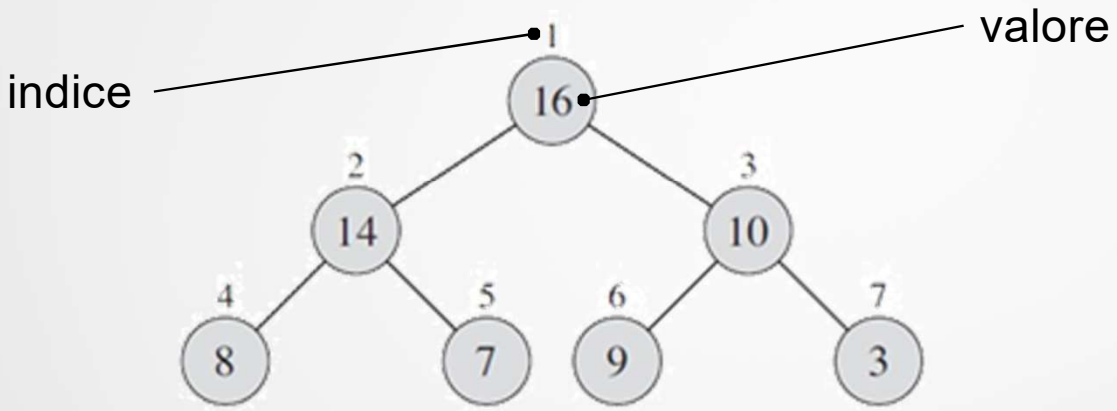
Questa struttura dati è di uso comune e risulta versatile.
Una volta costruita su un array permette di risolvere facilmente due diversi problemi:

- ordinamento dell'array
- gestione di una coda a priorità

Proprietà fondamentale dello Heap

In uno *heap*, l'elemento contenuto in ogni nodo ha valore **maggiore (minore) o uguale** a quello degli elementi contenuti nei nodi figli.

Heap (2)



Proprietà degli indici:

- La *radice* dello heap ha indice 1.
- Il *figlio sinistro* del nodo *i*-esimo ha indice 2*i*.
- Il *figlio destro* del nodo *i*-esimo ha indice 2*i* + 1.
- Il *padre* del nodo *i*-esimo ha indice $\lfloor i/2 \rfloor$.

Heap (3)

Questo significa che, a livello fisico, uno *heap* può consistere semplicemente in un *array* sul quale si opera seguendo la gestione degli indici formalizzata in precedenza.

Nell'immagine sottostante, ad esempio, la terza cella dell'array (i=3) è direttamente connessa alle celle 6 ($2i$) e 7 ($2i$ + 1).

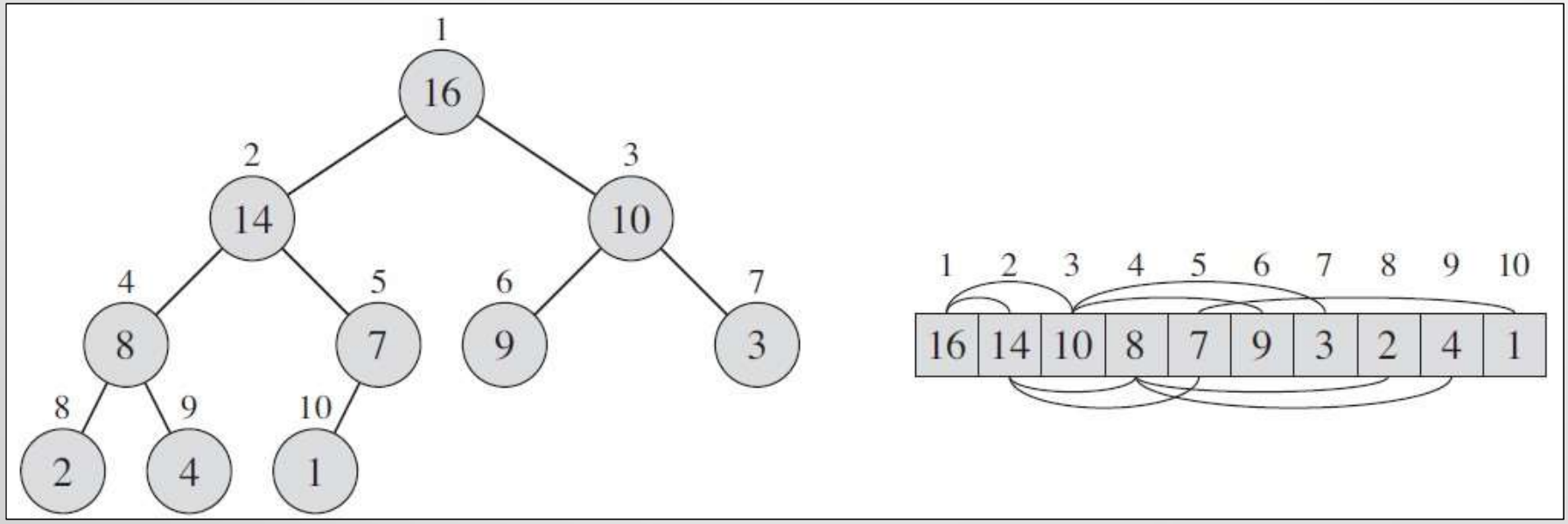


Immagine tratta da: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms, 3rd Edition.

Heap (4)

MaxHeapify $O(\log n)$

INPUT: $A = [a_1,\ldots,a_n], i \in \{1,\ldots,n\}$ OUTPUT: $A = [a_1,\ldots,a_n]$

```
1. l ← 2i, r ← 2i+1
2. if l≤n and A[l]>A[i] then largest ←l
3. else largest ←i
4. if r≤n and A[r]>A[largest] then largest ← r
5. if largest≠i then                                   | scambia
6.     t ← A[i]                                        |
7.     A[i] ← A[largest]                               |
8.     A[largest] ← t                                  |
9.     MaxHeapify(A, largest)
```

La procedura **MaxHeapify** consente di riorganizzare la struttura dati in modo che il valore passato in input (*A*[i]) rispetti la proprietà fondamentale dello heap. Si assume che gli alberi destro e sinistro che hanno origine nel nodo *i*-esimo siano già correttamente organizzati.

Heap (5)

BuildMaxHeap $O(n)$

INPUT: $A = [a_1,\ldots,a_n]$ OUTPUT: $A = [a_1,\ldots,a_n]$

1. i$\leftarrow \lfloor n/2 \rfloor$
2. **while** i≥1 **do**
3. **MaxHeapify**(*A*, i)
4. i ← i −1

La procedura **BuildMaxHeap** esegue *n*/2 chiamate a *MaxHeapify* su un array dato in input. Questo perché la seconda metà dell'array rappresenta le foglie dello heap. Poiché non ha senso valutare la proprietà fondamentale per nodi privi di figli, *MaxHeapify* viene eseguita solo per gli elementi che si trovano nella prima metà dell'array.

Heap (6)

Esempio

Organizzazione dell'array A=[1,9,3,7,10] mediante *BuildMaxHeap*:

BuildMaxHeap(A)	[1,9,3,7,10]
MaxHeapify(*A*,2) (*scambia*)	[1,**10**,3,7,**9**]
MaxHeapify(*A*,5)	
MaxHeapify(*A*,1) (*scambia*)	[**10**,**1**,3,7,9]
MaxHeapify(*A*,2) (*scambia*)	[10,**9**,3,7,**1**]
MaxHeapify(*A*,5)	

Output A=[10,9,3,7,1]

Obiettivo

1. Implementare la procedura **MaxHeapify** seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato sui seguenti array forniti nel materiale dell'esercitazione: *data.txt*, *data2.txt*.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

Visualizzazione di uno heap

```
Lettura del file data2.txt
1->100
2->114
3->116
4->800
5->200
6->443
7->221
8->310
9->109
10->705
11->191
12->457
13->387
14->238
15->396

Costruzione dello heap...
Struttura a heap:

                                        800

                    705                                         457

        310                         200                 443                 396

114             109             100             191             116             387             238             221
```

Heapsort (1)

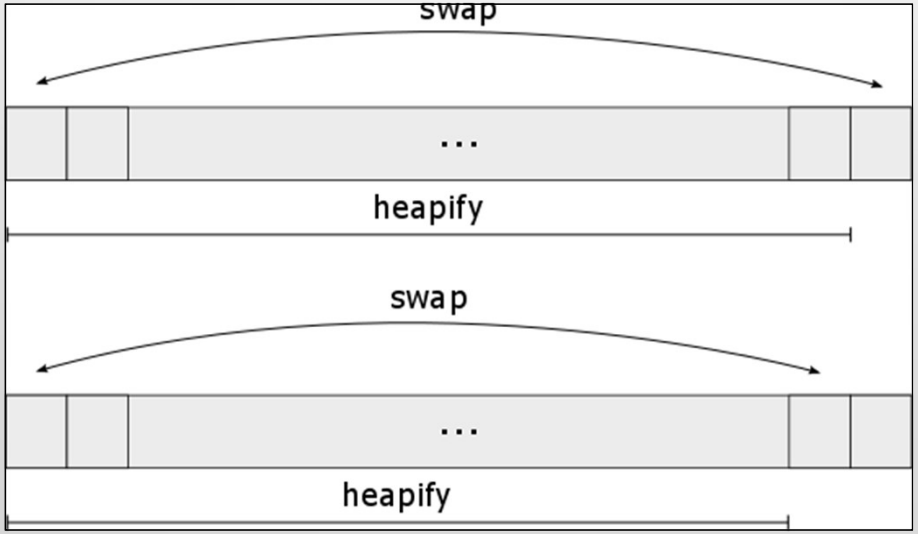
Heapsort è un buon algoritmo di ordinamento che viene eseguito in tempo *quasi lineare* ($O(n \log n)$) e non utilizza memoria aggiuntiva (cioè, è un metodo *in place*).

L'algoritmo sfrutta l'ordinamento parziale che caratterizza la struttura dati heap per ottimizzare il tempo calcolo.

Sostanzialmente, la **radice** (elemento di maggior valore) viene **scambiata con l'ultima foglia** e la dimensione dello heap viene ridotta di un'unità.

La proprietà fondamentale dello heap viene preservata con una opportuna chiamata a *MaxHeapify*.

Iterando la procedura per ogni elemento dello heap, si ottiene l'array ordinato.



Heapsort (2)

Heapsort $O(n \log n)$

INPUT: $A = [a_1, \ldots, a_n]$

OUTPUT: $A = [a_1, \ldots, a_n]$

1. **BuildMaxHeap**(A)
2. **while** $n>1$ **do**
3. $t \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. $A[n] \leftarrow t$
6. $n \leftarrow n - 1$
7. **MaxHeapify**($A, 1$)

Obiettivo

1. Implementare l'algoritmo *Heapsort* seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato sugli array non ordinati forniti nel materiale dell'esercitazione misurando il tempo richiesto.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

Code a priorità (1)

Definizione

Una **coda con priorità** è una struttura dati che permette di rappresentare un insieme di elementi su cui è definita una relazione d'ordine.

Tipicamente le code con priorità memorizzano ogni elemento associandolo a un valore di **chiave** (numerico) sul quale è definita una relazione d'ordine (*maggiore di* o *minore di*).

Sono utilizzate in molteplici domini applicativi:

- routing di pacchetti in reti di comunicazione;
- politiche di accesso a risorse condivise;
- lavorazione dei task nei sistemi di cloud computing;
- problemi di ottimizzazione.

Code a priorità (2)

ExtractMax $O(\log n)$

INPUT: $H = [h_1,\ldots,h_n]$ OUTPUT: $H = [h_1,\ldots,h_{n-1}]$, max

1. $max \leftarrow H[1]$
3. $H[1] \leftarrow H[n]$
4. $n \leftarrow n - 1$
5. **MaxHeapify**$(H, 1)$
6. **return** max

La funzione **ExtractMax** restituisce il massimo elemento contenuto nello heap *H*, prelevandolo dalla radice, che è sempre l'elemento di massimo (o minimo) valore. Il costo computazionale è dovuto alla conseguente riorganizzazione della struttura a heap, che viene fatta sostituendo la radice con l'ultima foglia e richiamando *MaxHeapify* su di essa ($O(\log n)$).

Obiettivo

1. Implementare la funzione **ExtractMax** seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato sui seguenti array forniti nel materiale dell'esercitazione: *data.txt*, *data2.txt*.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.