Heap, heapsort, coda di priorità con heap

May 8, 2019

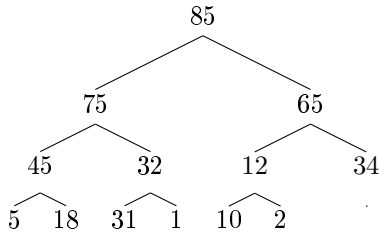
Obiettivi: sviluppo di operazioni sulla struttura heap, applicazioni dello heap.
Argomenti: definizione dello heap, inserimento, estrazione, heapsort, code di priorità.

1 Heap

Definizione di heap massimo: un albero binario etichettato con numeri interi è un **heap massimo** se:

- l'albero è completo salvo al più l'ultimo livello e l'ultimo livello è riempito da sinistra;
- per ogni nodo (tranne la radice), l'etichetta del genitore del nodo è maggiore dell'etichetta del nodo.

Per esempio, il seguente albero binario è un heap massimo:



In un **heap minimo**, per ogni nodo, l'etichetta del genitore del nodo è minore dell'etichetta del nodo.

Data la struttura rigida dell'albero, è conveniente rappresentare lo heap con un array in cui vengono elencate l'etichette livello per livello. Per lo heap precedente l'array è

$$(85, 75, 65, 45, 32, 12, 34, 5, 18, 31, 1, 10, 2)$$

Sia H un vettore che rappresenta uno heap e l'indice della radice sia 1. Quindi $H[1]$ è l'etichetta della radice. Inoltre, i figli di $H[i]$ sono $H[2i]$ (figlio sinistro) e $H[2i+1]$ (figlio destro). Il genitore di $H[i]$ è $H[\lfloor i/2 \rfloor]$.

Introduciamo le seguenti algoritmi per calcolare la posizione del genitore e dei figli dell'i-esimo elemento in H (se esistono). ($H.N$ denota il numero di elementi nello heap H.)

```
PARENT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del genitore se esiste, 0 altrimenti
return ⌊i/2⌋
```

```
LEFT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del figlio sinistro se esiste, i altrimenti
if 2i ≤ H.N then
    return 2i
else
    return i
end if
```

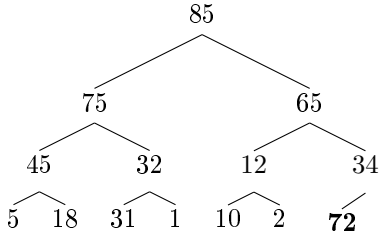
RIGHT(H, i)
▷ Pre: $1 \leq i \leq H.N$
▷ Post: restituisce la posizione del figlio destro se esiste, i altrimenti
if $2i + 1 \leq H.N$ **then**
return $2i + 1$
else
return i
end if

1.1 Operazioni su un heap

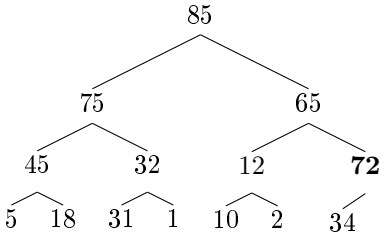
Inserimento. L'etichetta da inserire viene inserito inizialmente in fondo dell'array e poi viene fatta risalire tramite scambi verso la radice fino a trovare una posizione corretta per quanto riguarda le caratteristiche dello heap.

Inserimento dell'etichetta 72 nello heap precedente richiede due scambi per ripristinare le caratteristiche dello heap.

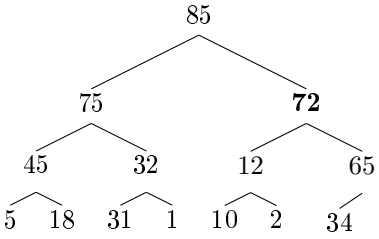
Inserimento in fondo, lo heap è scorretto:



Primo scambio, lo heap è sempre scorretto:



Secondo scambio, lo heap è corretto:



Algoritmo dell'inserimento:

HEAPINSERT(H, x)
▷ Pre: H è un heap
▷ Post: H è un heap con x inserito
$H.N \leftarrow H.N + 1$
$p \leftarrow H.N$
$H[p] \leftarrow x$
while $p > 1 \wedge H[p] > H[\text{PARENT}(H, p)]$ **do**
scambia $H[p]$ e $H[\text{PARENT}(H, p)]$
$p \leftarrow \text{PARENT}(H, p)$
end while

L'algoritmo effettua al massimo $l - 1$ scambi dove l è il numero di livelli dell'albero. La complessità quindi è $O(\log n)$ dove n è il numero di nodi dell'albero.

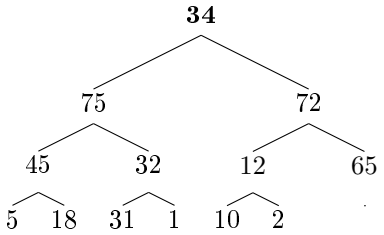
La correttezza si può dimostrare considerando il seguente invariante: il sottoalbero che ha come radice il nodo con l'etichetta nuova è uno heap.

Estrazione del massimo. La cancellazione del massimo (che si trova per forza nella radice) si effettua in due fasi:

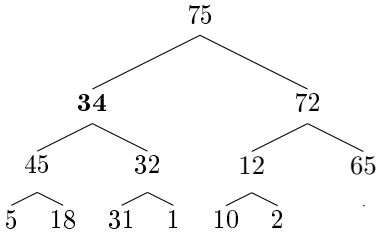
1. l'elemento più a destra dell'ultimo livello rimpiazza la radice;
2. l'elemento ora in radice viene fatto discendere lungo l'albero finché non sia maggiore di entrambi i figli; nel discendere si sceglie sempre il figlio con l'etichetta maggiore.

Cancellazione della radice nell'ultimo heap disegnato:

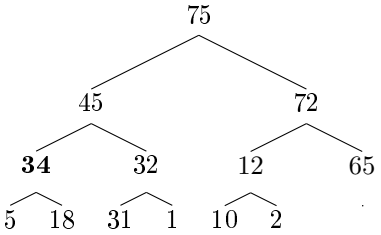
L'ultimo elemento rimpiazza la radice, lo heap è scorretto:



Primo scambio, lo heap rimane scorretto:



Secondo scambio, lo heap è corretto:



Algoritmi per l'estrazione:

HeapExtract(H)
▷ Pre: H è un heap
▷ Post: H è un heap con etichetta massimo eliminata
$H[1] \leftarrow H[H.N]$
$H.N \leftarrow H.N - 1$
Heapify($H, 1$)

Heapify(H, i)
▷ Pre: $1 \leq i \leq H.N$, i sottoalberi con radice in Left(H, i) e Right(H, i) sono heap
▷ Post: l'albero con radice in i è heap
$m \leftarrow$ index of Max$\{H[i], H[\text{Left}(H, i)], H[\text{Right}(H, i)]\}$
if $m \neq i$ **then**
scambia $H[m]$ e $H[i]$
Heapify(H, m)
end if

L'algoritmo effettua al massimo $l - 1$ scambi dove l è il numero di livelli dell'albero. La complessità quindi è $O(\log n)$ dove n è il numero di nodi dell'albero.

La correttezza si può dimostrare considerando la seguente caratteristica: i due sottoalberi del nodo che contiene l'etichetta che era nell'ultimo nodo prima dell'estrazione (nodo con etichetta 34 nella figura sopra) sono heap.

2 Heapsort

L'idea del **heapsort**: Consideriamo il seguente vettore

$$V = (\underbrace{V[1], V[2], ..., V[N-1], V[N]}_{\text{minoranti di } \{V[N+1],...,V[M]\}}, \underbrace{V[N+1], V[N+2], ..., V[M-1], V[M]}_{\substack{\text{maggioranti di } \{V[1],...,V[N]\} \\ \text{è ordinati}}})$$

Se $(V[1], ...V[N])$ rapresentasse uno heap allora sfruttando l'estrazione del massimo si può allargare la parte ordinata del vettore.

Dato un vettore V qualsiasi di M elementi, si può riorganizzarlo in modo che sia uno heap. Gli elementi che sono in posizione foglia sono già heap (un singolo nodo è heap). Una posizione i corrisponde ad una foglia se $2i > M$ (il nodo non deve avere un figlio sinistro per essere una foglia). Di conseguenza, le foglie sono nelle posizioni $\lfloor M/2 \rfloor + 1, ..., M$. Bisogna iterare dunque Heapify a partire dalla posizione $\lfloor M/2 \rfloor$ fino alla posizione 1. Denoteremo con $V.M$ il numero di elementi in V e con $V.N$ il numero di elementi su cui devono operare gli algoritmi introdotti nella sezione precedente.

BuildHeap(V)
▷ Pre: V è un vettore di M elementi

```
    ▷ Post: V rappresenta una heap
V.N ← V.M
for i ← ⌊V.M/2⌋ down to 1 do
    HEAPIFY(V, i)
end for
```

A partire da un vettore V che rappresenta uno heap, si può ottenere un vettore ordinato tramite $V.N$ estrazione del massimo:

```
HEAPSORT(V)
    ▷ Pre: V è un vettore di M elementi
    ▷ Post: V è ordinato
BUILDHEAP(V)
for i ← V.N down to 2 do
    scambia V[i] e V[1]
    V.N ← V.N − 1
    HEAPIFY(V, 1)
end for
```

Complessità di ordinare un vettore di n elementi:

- HEAPBUILD effettua $\lceil n/2 \rceil$ volte HEAPIFY e quindi ha complessità $O(n \log n)$;
- poi ci sono $n - 1$ estrazioni del massimo che ha in totale complessità $O(n \log n)$;
- in totale HEAPSORT ha complessità $O(n \log n)$ e quindi un algoritmo di ordinamento ottimo.

3 Coda di priorità con heap

In una coda di priorità ogni elemento è associato con un valore che indica la sua priorità. L'estrazione toglie dalla coda l'elemento che ha priorità più elevata.

Una coda di priorità si può realizzare con uno heap massimo. Le operazioni ENQUEUEPR e DEQUEUEPR si possono implementare utilizzando direttamente i meccanismi dell'inserimento e dell'estrazione dello heap massimo. L'operazione FRONTPR richiede soltanto restituire il primo elemento del vettore dello heap che rappresenta la coda di priorità.