Coda con priorità: heap

Collezione di elementi in cui a ogni elemento è associato un valore, detto PRIORITÀ, appartenente ad un insieme totalmente ordinato

Es. Interi positivi, {BASSO, MEDIO, ALTO, ALTISSIMO} ...

Può essere vista come una "estensione della coda" con le stesse operazioni:

EMPTY, ENQUEUE, FIRST e DEQUEUE

L'unica sostanziale differenza è che le ultime operazioni devono restituire l'elemento di PRIORITÀ MASSIMA.

Ciascun elemento ha cioè un campo e.prio oltre al campo e.dato.

L'implementazione è più complessa ...

Ragioniamoci un po'!

Se usassimo una lista di elementi, l'inserimento "in testa" implicherebbe una lista NON ordinata $\Rightarrow$ tempo per FIRST e DEQUEUE $O(n)$. Se invece mantenessimo

la lista ordinata $\Rightarrow$ FIRST e DEQUEUE richiedono $O(1)$ ma ENQUEUE richiede tempo $O(n)$.

È possibile bilanciare il costo delle operazioni?

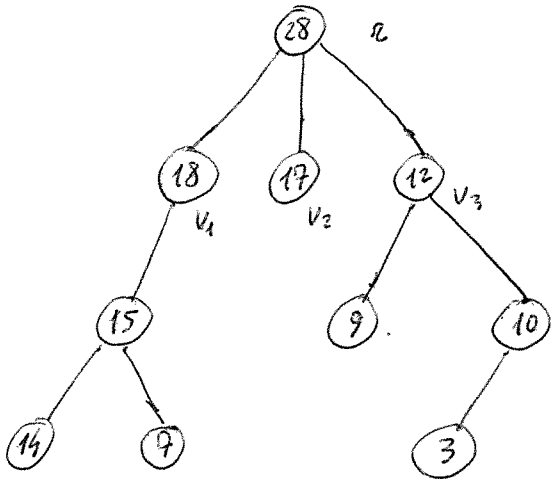
Sì può! Utilizzando una struttura di dati in cui gli elementi sono "parzialmente ordinati".

Osservazione: siamo di fronte al primo esempio in questo corso di come l'organizzazione dei dati - opportuna strutturazione - renda più efficienti nel complesso le operazioni su di essi.

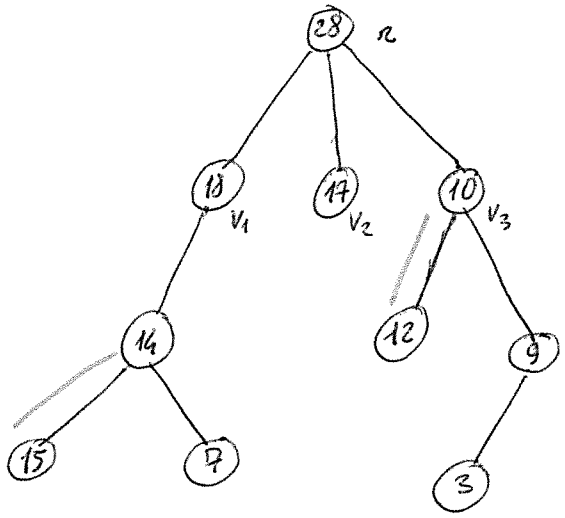
HEAP TREE

Uno heaptree è un albero vuoto oppure un albero H che soddisfa la seguente PROPRIETÀ DI HEAP, dove r indica la radice di H e $V_0, V_1, \dots V_{K-1}$ i suoi figli:

1. la priorità dell'elemento contenuto nella radice r di H è maggiore o uguale di quelle dei figli della radice, ovvero $r.prio \geq V_i.prio$, per $0 \leq i < K$.
2. l'albero radicato in V_i è uno heaptree, per $0 \leq i < K$.



SI !



NO !

I nodi possono essere classificati in base alla loro profondità.

La radice r ha profondità 0. I figli profondità 1.

I nipoti profondità 2...

Un nodo a profondità p ha figli a profondità p+1.

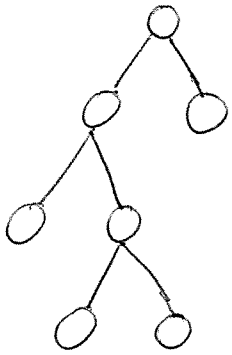
L'altezza h di uno heaptree è la massima profondità delle foglie.

Un albero binario è completo se ogni nodo interno ha esattamente due figli

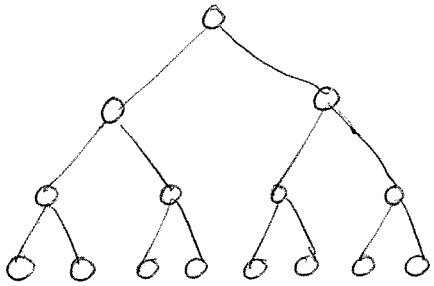
L'albero è completamente bilanciato se, oltre ad essere completo, tutte le foglie hanno la stessa profondità.

Un albero binario di altezza h è completo a sinistra se i nodi di altezza minore di h formano un albero completamente bilanciato, e se i nodi di profondità h sono tutti accumulati a sinistra.

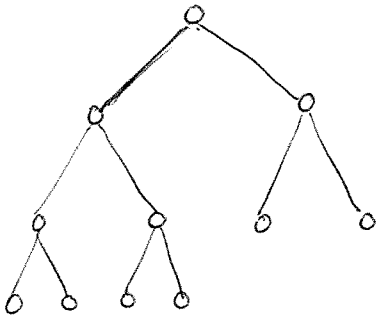
Esempi



Albero binario completo



Albero binario completamente bilanciato



Albero binario completo a sinistra

Lemma 2.1 L'altezza h di un albero completamente bilanciato con n nodi è uguale a $\log(n+1) - 1$.

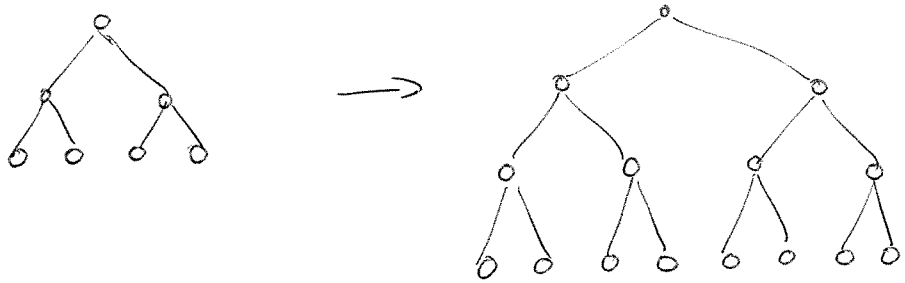
Dimostrazione. Cominciamo con far vedere che un albero completamente bilanciato di altezza h ha $2^h - 1$ nodi interni e 2^h foglie. Ragioniamo per induzione.

Base dell'induzione: $h = 0$. L'albero di altezza 0 è costituito solo dalla radice.

Pertanto, il numero di nodi interni è $0 = 2^0 - 1$, e il numero di foglie è $1 = 2^0$, in accordo all'enunciato.

Ipotesi induttiva: $h > 0$. Un albero completamente bilanciato di altezza h ha $2^h - 1$ nodi interni e 2^h foglie.

Passo induttivo: $h + 1$. Consideriamo un albero completamente bilanciato di altezza $h+1$. Tale albero si può ottenere da un albero completamente bilanciato di altezza h, facendo diventare ogni foglia un nodo interno a cui sono aggiunti due figli (foglie).



Pertanto, risultano:

$$\#\text{ nodi interni} = 2^h - 1 \ (\text{Ipotesi induttiva}) + 2^h \ (\text{foglie precedenti})$$

$$= 2^h + 2^h - 1 = 2 \cdot 2^h - 1 = 2^{h+1} - 1$$

$$\#\text{ foglie} = 2 \cdot 2^h = 2^{h+1}.$$

L'affermazione è dunque vera in generale.

Relativamente all'altezza h, possiamo allora dire che:

$$n = 2^h - 1 + 2^h = 2^{h+1} - 1 \iff$$

$$2^{h+1} = n + 1 \iff h + 1 = \log(n+1) \iff$$

$$h = \log(n+1) - 1.$$

Il lemma è, pertanto, dimostrato. ∎

Corollario. Se h è l'altezza di un albero completo e sinistra con n nodi, allora $h = O(\log n)$.

Dimostrazione. Dal lemma precedente, se m è il numero di nodi a profondità minore di h (che formano un albero completamente bilanciato di altezza $h-1$), risulta:

$$h - 1 = \log(m+1) - 1.$$

Pertanto,

$$h = \log(m+1) \leq \log(n+1) = O(\log n)$$

∎

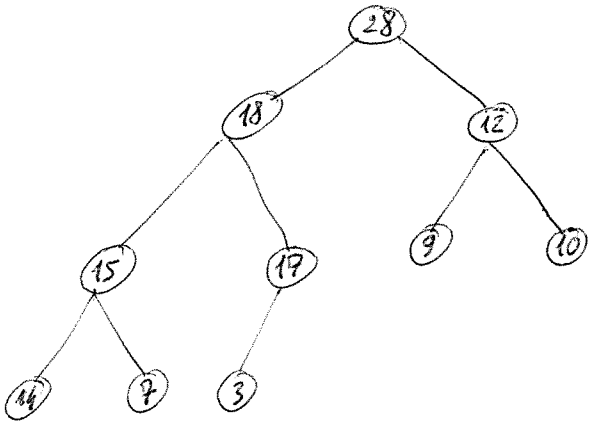
Ricordo che, per provare che $\log(n+1) = O(\lg n)$, occorre far vedere che esistono costanti $c, n_0 > 0$ tali che $\log(n+1) \leq c \cdot \lg n$, $\forall n \geq n_0$.

Risulta: $\log(n+1) \leq c \cdot \log(n) \iff \log(n+1) \leq \log n^c \iff$ $(n+1) \leq n^c$. Scegliendo $c = 2$ ed $n_0 = 2$, $n + 1 \leq n^2$ $\forall n \geq 2$.

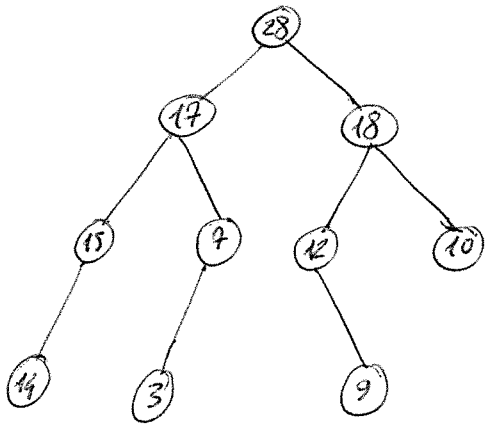
HEAP

Uno heap è uno heaptree con vincoli aggiuntivi di essere binario e completo a sinistra. Pertanto uno heap con n nodi ha altezza $h = O(\log n)$ (corollario precedente)

Esempi



SI !



NO !

L'altezza logaritmica di uno heap consente di effettuare in tempo $O(\log n)$ le operazioni Enqueue e Dequeue..

Prima di guardare il codice, soffermiamoci su uno schema generale per le due operazioni.

Indichiamo l'heap con H.

ENQUEUE: di e in H

Inseriamo un nuovo nodo v contenente e come foglia di H, in modo da mantenere H completo a sinistra.

Iterativamente, confrontiamo la priorità di e con quella dell'elemento f contenuta nel padre di v e, se e.prio > f.prio, scambiamo i due nodi.
Il processo termina o quando v diventa la radice di H, oppure quando e.prio ≤ f.prio. In entrambi i casi la proprietà di heap è stata ristabilita.

DEQUEUE:

Estraiamo la radice r di H, in modo da restituire l'elemento in essa contenuto alla fine dell'operazione.

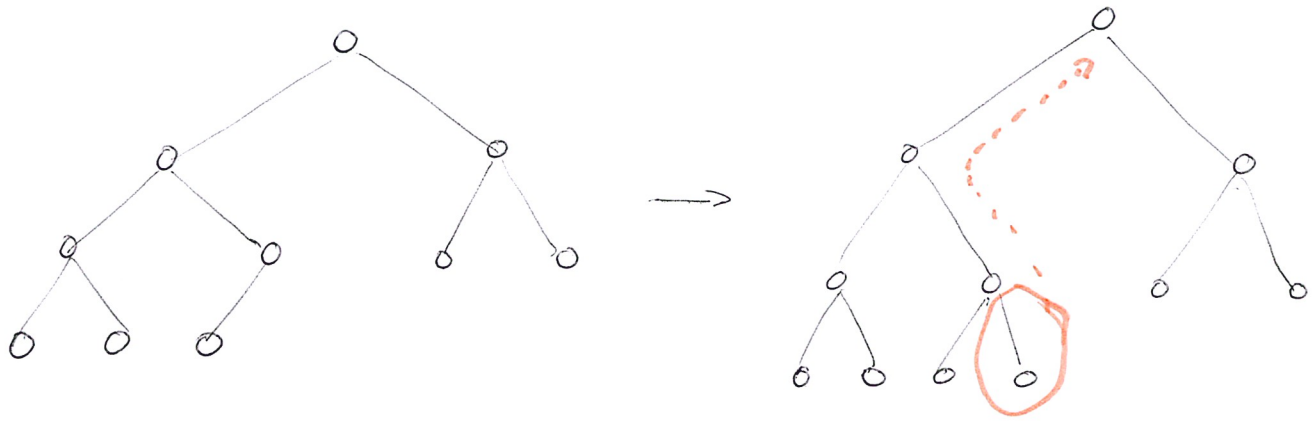
Rimuoviamo l'ultima foglia di H (quella più a destra nell'ultimo livello) per inserirla come radice r al posto di quella estratta. Manteniamo così H completo a sinistra.

Iterativamente, confrontiamo la priorità dell'elemento in r con quelle dei figli e, se il massimo fra le priorità non è quella di r, il nodo viene scambiato con il figlio contenente la priorità massima.

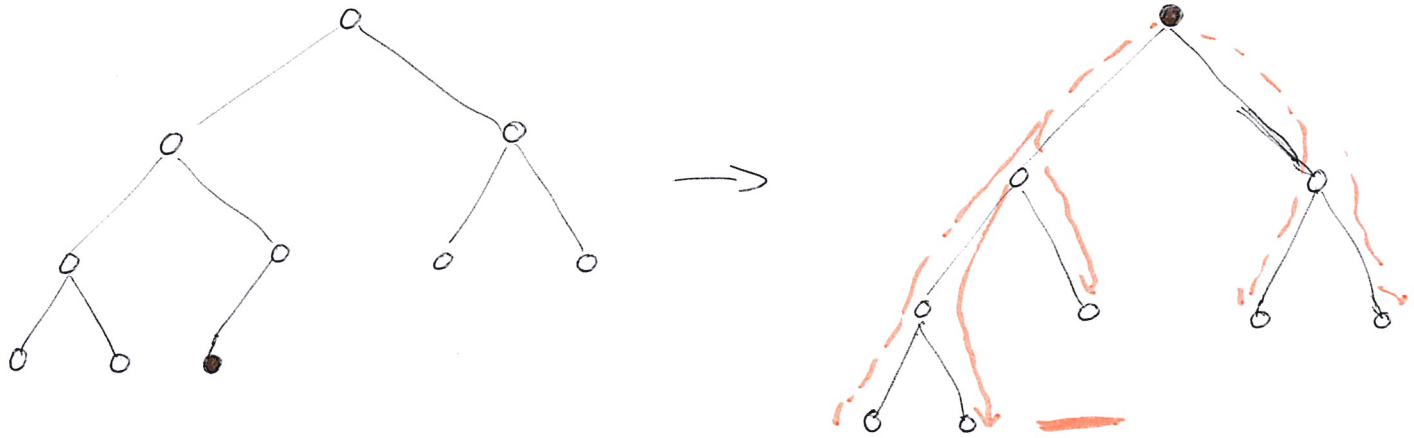
Il processo termina o quando si raggiunge una foglia oppure quando il nodo padre ha la priorità maggiore o uguale a quella dei suoi figli.

Es.

ENQUEUE



DEQUEUE



Nel caso dell'ENQUEUE, l'operazione di "risalita" nell'albero costa naturalmente $O(\log n)$.

Parimenti, nel caso della DEQUEUE, l'operazione di "discesa" lungo l'albero costa $O(\log(n))$.