Algoritmi e Strutture Dati

Code di priorità

Basato su materiale di C. Demetrescu, I. Finocchi, G.F. Italiano

Tipo di dato CodaPriorità (1/2)

tipo `CodaPriorita`:
dati:
un insieme S di n elementi di tipo $elem$ a cui sono associate chiavi di tipo $chiave$ prese da un universo totalmente ordinato.

operazioni:

`findMin`$() \rightarrow elem$
restituisce l'elemento in S con la chiave minima.

`insert`$(elem\ e, chiave\ k)$
aggiunge a S un nuovo elemento e con chiave k.

`delete`$(elem\ e)$
cancella da S l'elemento e.

`deleteMin`$()$
cancella da S l'elemento con chiave minima.

Tipo di dato CodaPriorità (2/2)

`increaseKey`$(elem\ e, chiave\ d)$
incrementa della quantità d la chiave dell'elemento e in S.

`decreaseKey`$(elem\ e, chiave\ d)$
decrementa della quantità d la chiave dell'elemento e in S.

`merge`$(\texttt{CodaPriorita}\ c_1, \texttt{CodaPriorita}\ c_2) \rightarrow \texttt{CodaPriorita}$
restituisce una nuova coda con priorità $c_3 = c_1 \cup c_2$.

Tre implementazioni

- d-heap: generalizzazione degli heap binari visti per l'ordinamento
- heap binomiali
- heap di Fibonacci

d-heap

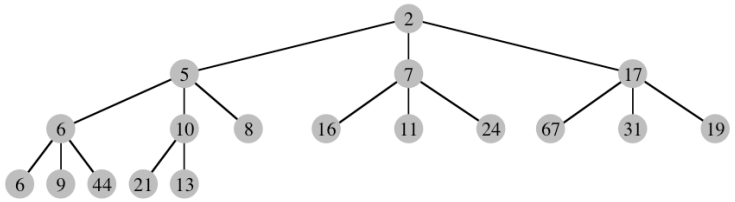
Definizione

Un d-heap è un albero radicato d-ario con le seguenti proprietà:

1. Struttura: è completo almeno fino al penultimo livello
2. Contenuto informativo: ogni nodo v contiene un elemento elem(v) ed una chiave chiave(v) presa da un dominio totalmente ordinato
3. Ordinamento a heap: chiave(v) ≥ chiave(parent(v)) per ogni nodo v diverso dalla radice

Esempio

Heap d-ario con 18 nodi e d=3



Proprietà

1. Un d-heap con n nodi ha altezza $O(\log_d n)$
2. La radice contiene l'elemento con chiave minima (per via della proprietà di ordinamento a heap)
3. Può essere rappresentato implicitamente tramite vettore posizionale grazie alla proprietà di struttura

Procedure ausiliarie

Utili per ripristinare la proprietà di ordinamento a heap su un nodo v che non la soddisfi

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

procedura `muoviAlto`(v)
  while ($v \neq radice(T)$ **and** $chiave(v) < chiave(padre(v))$) **do**
    scambia di posto v e $padre(v)$ in T

T(n)=O(d $\log_d$ n)

procedura `muoviBasso`(v)
  repeat
    sia u il figlio di v con la minima $chiave(u)$, se esiste
    if (v non ha figli o $chiave(v) \leq chiave(u)$) **break**
    scambia di posto v e u in T

findMin

`findMin`() $\rightarrow elem$
restituisce l'elemento nella radice di T.

T(n)=O(1)

insert(elem e, chiave k)

crea un nuovo nodo v con elemento e e chiave k, in modo che diventi una foglia sull'ultimo livello di T. La proprietà dell'ordinamento a heap viene poi ripristinata spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

T(n)=O($\log_d$ n) per l'esecuzione di *muoviAlto*

delete(elem e) e deleteMin

scambia il nodo v contenente l'elemento e con una qualunque foglia u sull'ultimo livello di T, e poi elimina v. Ripristina infine la proprietà dell'ordinamento a heap spingendo il nodo u verso la sua posizione corretta scambiandolo ripetutamente con il proprio padre o con il proprio figlio contenente la chiave più piccola

T(n)=O(d $\log_d$ n) per l'esecuzione di *muoviBasso*

Può essere usata anche per implementare la cancellazione del minimo

decreaseKey(elem e, chiave q)

decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta q. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

$T(n)=O(\log_d n)$ per l'esecuzione di muoviAlto

increaseKey(elem e, chiave q)

aumenta il valore della chiave nel nodo contenente l'elemento e della quantità richiesta q. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso il basso tramite ripetuti scambi di nodi.

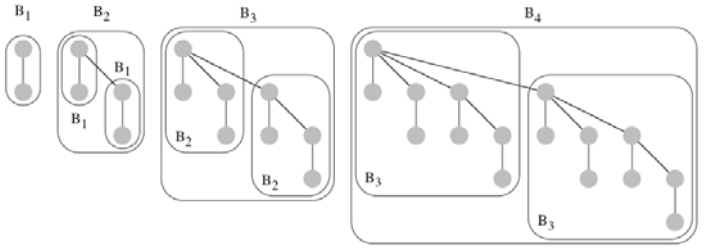
$T(n)=O(d \log_d n)$ per l'esecuzione di muoviBasso

Heap binomiali

Alberi binomiali

Un albero binomiale B_h è definito ricorsivamente come segue:

1. B_0 consiste di un unico nodo
2. Per $i \geq 0$, B_{i+1} è ottenuto fondendo due alberi binomiali B_i con la radice dell'uno come figlia della radice dell'altro



Proprietà strutturali

Un albero binomiale B_h gode delle seguenti proprietà:

1. *Numero di nodi ($|B_h|$):* $n = 2^h$.
2. *Grado della radice:* $D(n) = \log_2 n$
3. *Altezza:* $H(n) = h = \log_2 n$.
4. *Figli della radice: i sottoalberi radicati nei figli della radice di B_h sono $B_0, B_1, \ldots, B_{h-1}$.*

Definizione di heap binomiale

Un heap binomiale è una foresta di alberi binomiali con le seguenti proprietà:

1. Struttura: ogni albero B_i nella foresta è un albero binomiale
2. Unicità: per ogni i, esiste al più un B_i nella foresta
3. Contenuto informativo: ogni nodo v contiene un elemento elem(v) ed una chiave chiave(v) presa da un dominio totalmente ordinato
4. Ordinamento a heap: chiave(v) ≥ chiave(parent(v)) per ogni nodo v diverso da una delle radici

Proprietà

In un heap binomiale con *n* nodi, vi sono al più $log_2\ n$ alberi binomiali,
ciascuno con grado ed altezza O(log n)

Procedura ausiliaria

Utile per ripristinare la proprietà di unicità in un heap binomiale

procedura `ristruttura`()
  $i = 0$
  while (esistono ancora due B_i) **do**
    si fondono i due B_i per formare un albero B_{i+1}, ponendo la radice con chiave più piccola come genitore della radice con chiave più grande
    $i = i + 1$

T(n) è proporzionale al numero di alberi binomiali in input

Realizzazione (1/3)

classe `HeapBinomiale` **implementa** `CodaPriorita`:
dati:
una foresta H con n nodi, ciascuno contenente un elemento di tipo $elem$ e una chiave di tipo $chiave$ presa da un universo totalmente ordinato.

operazioni:
`findMin`$() \rightarrow elem$
scorre le radici in H e restituisce l'elemento a chiave minima.

`insert`$(elem\ e, chiave\ k)$
aggiunge ad H un nuovo B_0 con dati e e k. Ripristina poi la proprietà di unicità in H mediante fusioni successive dei doppioni B_i.

Realizzazione (2/3)

`deleteMin`$()$
trova l'albero T_h con radice a chiave minima. Togliendo la radice a T_h, esso si spezza in h alberi binomiali $T_0, \ldots, T_{h-1}$, che vengono aggiunti ad H. Ripristina poi la proprietà di unicità in H mediante fusioni successive dei doppioni B_i.

`decreaseKey`$(elem\ e, chiave\ d)$
decrementa di d la chiave nel nodo v contenente l'elemento e. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

`delete`$(elem\ e)$
richiama `decreaseKey`$(e, -\infty)$ e poi `deleteMin`$()$.

Realizzazione (3/3)

`increaseKey`$(elem\ e, chiave\ d)$
richiama `delete`(e) e poi `insert`$(elem, k + d)$, dove k è la chiave associata all'elemento e.

`merge`$(\texttt{CodaPri}.\ c_1, \texttt{CodaPri}.\ c_2) \rightarrow \texttt{CodaPri}.$
unisce gli alberi in c_1 e c_2 in un nuovo heap binomiale c_3. Ripristina poi la proprietà di unicità nell'heap binomiale c_3 mediante fusioni successive dei doppioni B_i.

Tutte le operazioni richiedono tempo T(n) = O(log n)
Durante l'esecuzione della procedura ristruttura esistono infatti al più tre B_i, per ogni $i \geq 0$

Heap di Fibonacci

Heap di Fibonacci

Heap binomiale rilassato: si ottiene da un heap binomiale rilassando la proprietà di unicità dei B_i ed utilizzando un atteggimento più "pigro" durante l'operazione insert (perché ristrutturare subito la foresta quando potremmo farlo dopo?)

Heap di Fibonacci: si ottiene da un heap binomiale rilassato rilassando la proprietà di struttura dei B_i che non sono più necessariamente alberi binomiali

Analisi sofisticata: i tempi di esecuzione sono ammortizzati su sequenze di operazioni

Conclusioni: tabella riassuntiva

	DHeap	HeapBin.	HeapBin.Ril.	HeapFib.
findMin	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
delete	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
increaseKey	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
merge	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$

L'analisi di DHeap e HeapBinomiale è nel caso peggiore, mentre quella per HeapBinomialeRilassato e HeapFibonacci è ammortizzata