

Algoritmi e Strutture Dati

Capitolo 8 Code di priorità


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 di base:

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

Si può analogamente definire la variante di coda di priorità con `findMax` e `deleteMax`

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

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

Suppongo che mi venga dato un riferimento diretto all'elemento da cancellare

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

Tipo di dato CodaPriorità (2/2)

Operazioni aggiuntive

Suppongo che mi venga dato un riferimento diretto all'elemento da modificare

$\text{increaseKey}(\text{elem } e, \text{chiave } \Delta)$

incrementa della quantità Δ la chiave dell'elemento e in S .

$\text{decreaseKey}(\text{elem } e, \text{chiave } \Delta)$

decrementa della quantità Δ la chiave dell'elemento e in S .

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

Applicazioni: gestione code in risorse condivise, gestione priorità in processi concorrenti, etc. Si noti che la coda di priorità **non supporta** l'operazione di **ricerca** di un elemento

Obiettivo

Fornire un'implementazione di una coda di priorità di n elementi che consenta di fare **tutte** le operazioni descritte in tempo $O(\log n)$.

Quattro implementazioni elementari

1. Array non ordinato
2. Array ordinato
3. Lista non ordinata
4. Lista ordinata

Ci focalizzeremo soltanto sulle
operazioni di base

NOTA BENE: Si noti che la coda di priorità è un tipo di dato **dinamico** (cioè di dimensione variabile), in quanto soggetto ad **inserimenti** e **cancellazioni**. L'uso degli array va quindi inteso pensando alla loro versione **dinamica**, che implica **riallocazioni/deallocazioni** di memoria che **raddoppiano/dimezzano** lo spazio utilizzato. Con tale accorgimento, i costi di riallocazione/deallocazione sono assorbiti (asintoticamente) dai costi per le **insert** e le **delete**

Array non ordinato

Tengo traccia del numero **n** di elementi effettivamente presenti nella coda (**dimensione logica** dell'array) in una variabile di appoggio, e gestisco la **dimensione fisica** dell'array mediante allocazione dinamica

- **FindMin**: $\Theta(n)$ (devo guardare tutti gli elementi)
- **Insert**: $O(1)$ (inserisco in fondo all'array)
- **Delete**: $O(1)$ (poiché mi viene fornito il riferimento diretto all'elemento da cancellare, lo posso cancellare in $O(1)$ sovracopiando l'ultimo elemento)
- **DeleteMin**: $\Theta(n)$ (devo prima cercare il minimo in $\Theta(n)$, poi lo posso cancellare in $O(1)$)

Array ordinato

Gestione dinamica come sopra; l'array viene inoltre tenuto **ordinato** in ordine **decrescente**

- **FindMin**: $O(1)$ (l'elemento minimo è in fondo all'array)
- **Insert**: $O(n)$ (trovo in $\Theta(\log n)$ la giusta posizione (**ricerca binaria**), ma poi devo fare $O(n)$ spostamenti verso destra)
- **Delete**: $O(n)$ (devo fare $O(n)$ spostamenti verso sinistra)
- **DeleteMin**: $O(1)$ (l'elemento minimo è in fondo all'array, non devo fare spostamenti)

Lista non ordinata

La considero **bidirezionale**, e mantengo un puntatore alla **testa** ed uno alla **coda**



- **FindMin**: $\Theta(n)$ (devo guardare tutti gli elementi)
- **Insert**: $O(1)$ (inserisco in coda o in testa)
- **Delete**: $O(1)$ (poiché mi viene fornito il riferimento diretto all'elemento da cancellare, lo posso cancellare in $O(1)$ agendo sui puntatori)
- **DeleteMin**: $\Theta(n)$ (devo prima cercare il minimo in $\Theta(n)$, poi lo posso cancellare in $O(1)$)

Lista ordinata

Tengo la lista bidirezionale **ordinata** in ordine **crescente**

- **FindMin**: $O(1)$ (il minimo è in testa alla lista)
- **Insert**: $O(n)$ (trovo in $O(n)$ la giusta posizione, e poi faccio in $O(1)$ l'inserimento)
- **Delete**: $O(1)$ (agisco sui puntatori)
- **DeleteMin**: $O(1)$ (basta far puntare la testa della lista al secondo elemento della lista stessa, e modificare il puntatore al predecessore di quest'ultimo a *nil*)

Riepilogo implementazioni elementari

	FindMin	Insert	Delete	DeleteMin
Array non ord.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$

⇒ Ogni implementazione elementare ha almeno un'operazione che comporta un **costo lineare**! Voglio fare meglio...

Tre implementazioni evolute

- ➔ d-heap: generalizzazione degli **heap binari** visti per l'ordinamento
- ➔ Heap binomiali
- ➔ Heap di Fibonacci (cenni)

d-heap

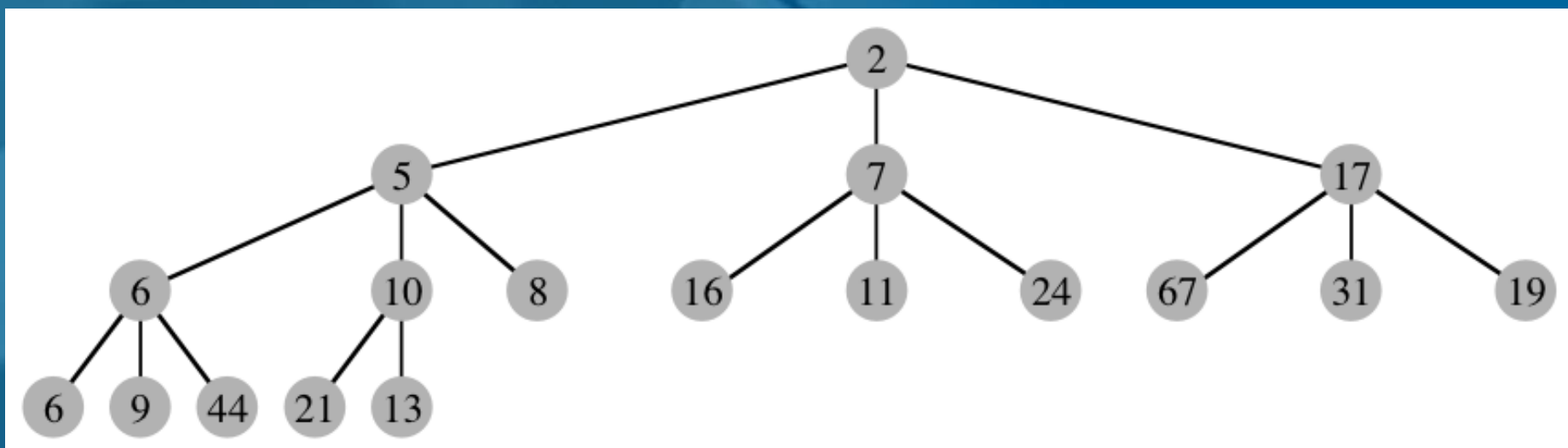
Definizione

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

1. **Struttura**: è **quasi completo**, ovvero è completo fino al penultimo livello, e tutte le foglie sull'ultimo livello sono compattate verso sinistra
2. **Contenuto informativo**: ogni nodo v contiene un elemento $\text{elem}(v)$ ed una chiave $\text{chiave}(v)$ presa da un dominio totalmente ordinato
3. **Ordinamento parziale** (inverso rispetto a quello usato per l'heapsort non decrescente) **dell'heap (min-heap)**: $\text{chiave}(v) \geq \text{chiave}(\text{parent}(v))$ 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** $\Theta(\log_d n)$
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.

Approfondimento: Dimostrare che, supponendo che il primo elemento dell'array sia in **posizione 1**, valgono le seguenti relazioni posizionali per l'elemento in **posizione $i \geq 1$** :

$$\text{padre}(i) = \lceil (i-1)/d \rceil \quad \text{figlio}_j(i) = (i-1) \cdot d + j + 1, \text{ per } 1 \leq j \leq d$$

Procedure ausiliarie

Nel prosieguo supporremo che l'heap d-ario venga mantenuto mediante un albero d-ario di nome T . Le seguenti procedure sono utili per ripristinare la proprietà di ordinamento parziale dell'heap allorché la **chiave** di un nodo v si trovi a non soddisfarla

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

procedura *muoviAlto*(v)

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

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

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

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)

scambia il nodo v contenente l'elemento e con la foglia u più a destra 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

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

deleteMin()

scambia la radice v contenente la chiave minima con la foglia u più a destra 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 figlio contenente la chiave più piccola

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

decreaseKey(elem e , chiave Δ)

decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta Δ . 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 Δ)

aumenta il valore della chiave nel nodo contenente l'elemento e della quantità richiesta Δ . 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**

merge(heap d-ario c_1 , heap d-ario c_2)

Analogamente a quanto mostrato per l'heap binario, la creazione di un heap d-ario (con d **costante**) di n elementi può essere eseguita in $\Theta(n)$. Infatti, il tempo di esecuzione di **heapify** diventa ora:

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

ove il fattore $O(d \log_d n)$ è relativo all'esecuzione della procedura **muoviBasso** (**fixheap** nell'heap binario).

Siamo quindi di nuovo nel Caso 1 del Teorema Master:

$$d \log_d n = f(n) = O(n^{\log_d d - \varepsilon}) \text{ per } \varepsilon > 0, \text{ e quindi } T(n) = \Theta(n^{\log_d d}) = \Theta(n)$$

\Rightarrow Il merge può quindi essere eseguito in $\Theta(n)$, ove $n = |c_1| + |c_2|$, generando un nuovo heap d-ario che contiene tutti gli elementi in c_1 e c_2

Riepilogo

	Find Min	Insert	Delete	DelMin	Incr. Key	Decr. Key	merge
Array non ord.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$???	???	???
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$???	???	???
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$???	???	???
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$???	???	???
d-Heap	$O(1)$	$O(\log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(n)$

⇒ Il d-heap purtroppo ancora non soddisfa il nostro obiettivo di implementare una coda di priorità con **costi logaritmici!**

Esercizi di approfondimento

1. Fornire un'implementazione dell'operazione di **merge**, in cui gli elementi di uno dei due heap vengono aggiunti sequenzialmente all'altro heap. Analizzarne quindi la convenienza asintotica rispetto all'implementazione appena fornita.
2. Valutare i costi delle operazioni aggiuntive (**IncreaseKey**, **DecreaseKey** e **Merge**) sulle implementazioni elementari (vettori e liste).