Capitolo 6

Heap e code di priorità

6.1 Heap

Definizione 6.1. Uno *heap* è un albero binario quasi completo che rispetta la *proprietà heap* definita come segue:

- nel caso di *min-heap*, il padre ha associata una chiave minore o uguale alle chiavi di tutti i suoi figli;
- nel caso di *max-heap*, il padre ha associata una chiave maggiore o uguale alle chiavi associati ai suoi figli.

Solitamente uno heap viene rappresentato utilizzando un vettore (a partire dalla posizione di indice 1) dove, per un generico nodo in posizione i, si ha:

- il padre in posizione $\lfloor i/2 \rfloor$;
- il figlio sinistro in posizione $2 \cdot i$;
- il figlio destro in posizione $2 \cdot i + 1$.

Lemma 6.1. *Uno heap con n nodi ha altezza $O(\log n)$.*

Dimostrazione. Sia h l'altezza dello heap con n nodi; poiché lo heap è completo almeno fino a profondità $h-1$, e un albero binario completo di profondità $h-1$ ha $\frac{2^h-1}{2-1} = 2^h - 1$ nodi, segue che $2^h - 1 < n$; con semplici passaggi algebrici, si giunge alla relazione $h < \log_2(n+1) = O(\log n)$. □

Nel seguito faremo riferimento solo a max-heap, ma quanto detto (con opportune modifiche dovute alla condizione heap diversa) può essere applicato anche a min-heap.

6.1.1 Costruzione

Per costruire un max-heap, si utilizza una funzione ausiliaria `heapifyDown`.

funzione ausiliaria `heapifyDown` (*vettore A, indice i*) → *void*

```
1 k = indiceArgomentoMassimo(i, figlioSx(i), figlioDx(i))
2 if(k != i)
3   swap(A[k], A[i])
4   heapifyDown(A, k)
```

Tale funzione verifica che il valore di $A[i]$ sia maggiore di quello dei figli, se presenti, e, in caso contrario, $A[i]$ migra verso il basso fino a che non vale la proprietà heap; la complessità è $O(h)$, dove h è l'altezza dell'albero. La funzione per la costruzione dello heap è la seguente:

funzione `buildHeap` (*vettore A*) → *void*

```
1 for i = length(A)/2} downto 1 do
2   heapifyDown(A, i)
```

Per dimostrare la correttezza della funzione, usiamo il seguente invariante di ciclo:

$$\forall k : i+1 \leq k \leq length(A), A[k] \text{ è radice di uno heap.}$$

Inizializzazione: si ha $i = length(A)/2$; $\forall k : i+1 \leq k \leq length(A), A[k]$ è una foglia e dunque, banalmente, radice di uno heap.

Mantenimento: assumiamo che la proprietà valga all'inizio di una generica iterazione e dimostriamo che vale anche alla fine; si ricorda che, ad ogni passo, i viene decrementata, nonostante non sia esplicitamente indicato. La chiamata `heapifyDown(A, i)` fa diventare $A[i]$ radice di uno heap, senza togliere questa proprietà agli altri: al termine dell'iterazione si ha che $A[(i-1)+1], A[(i-1)+2], ..., A[length(A)]$ sono radici di heap, ossia quanto volevamo dimostrare.

Terminazione: si ha $i = 0$ e che $A[1], A[2], ..., A[length(A)]$ sono tutti radici di heap, dunque A è uno heap.

Complessità

Per il calcolo della complessità, useremo la seguente formula:

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2} \tag{6.1}$$

Dimostrazione. Ricordiamo intanto il limite della serie geometrica:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Derivando ambo i membri della precedente uguaglianza otteniamo:

$$\begin{aligned}
\frac{d}{dx}\sum_{k=0}^{\infty} x^k &= \frac{d}{dx}\left(\frac{1}{1-x}\right) \\
\sum_{k=0}^{\infty} k \cdot x^{k-1} &= \frac{1}{(1-x)^2} \\
x \cdot \sum_{k=0}^{\infty} k \cdot x^{k-1} &= \frac{x}{(1-x)^2} \\
\sum_{k=0}^{\infty} k \cdot x^{k} &= \frac{x}{(1-x)^2}
\end{aligned}$$

Per procedere, dobbiamo calcolare quante volte la funzione `buildHeap` richiama `heapify`; dividiamo i vertici in base all'altezza: per ogni altezza h ce ne sono al più $\frac{n}{2^{h+1}}$. Per nodi alla stessa altezza h, la complessità di `heapify` è $O(h)$; la complessità totale risulta essere:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O\left(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil \right)$$

Sostituendo $x = 1/2$ nell'equazione (6.1) otteniamo:

$$\sum_{h=0}^{\infty} \left\lceil \frac{h}{2^{h+1}} \right\rceil = \frac{1/2}{(1-1/2)^2} = 2$$

Segue che:

$$O\left(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil \right) = O\left(n \cdot \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^{h+1}} \right\rceil \right) = O(n)$$

6.1.2 Realizzazione

Per realizzare le operazioni di inserimento e rimozione, risulta utile un'ulteriore funzione d'appoggio, simmetrica a `heapifyDown`.

funzione ausiliaria `heapifyUp` (*vettore A, indice i*) → *void*

```
1 if(i != 1 && A[i] < A[padre(i)]) then
2   swap(A[i], A[padre(i)])
3   heapifyUp(A, padre(i))
```

Inserimento: viene creato un nuovo nodo e inserito come foglia nella prima posizione libera del vettore usato per realizzare lo heap; a questo punto, per ripristinare la proprietà heap, viene richiamata la funzione `heapifyUp` sul nodo appena inserito. La complessità dell'operazione è $O(\log n)$.

Cancellazione: il nodo da rimuovere viene scambiato con l'ultimo elemento occupato del vettore; a questo punto, non sapendo se il nodo appena scambiato debba scendere o salire nell'albero, vengono richiamate su di esso entrambe le funzioni `heapifyUp` e `heapifyDown` per ripristinare la proprietà heap. La complessità dell'operazione, anche in questo caso, è $O(\log n)$.

6.2 Code di priorità

Le code di priorità possono essere realizzate usando degli heap: a seconda del fatto che sia usato un min-heap o un max-heap, si possono avere *code a min-priorità* e *code a max-priorità*. Una coda a min-priorità è descritta dal seguente schema generale:

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()` → *elem*
Restituisce l'elemento di S con la chiave minima.

`insert(`*elem e, chiave k*`)` → *void*
Aggiunge a S un nuovo elemento e con chiave K.

`delete(`*elem e*`)` → *void*
Cancella da S l'elemento e.

`deleteMin()` → *void*
Cancella da S l'elemento con chiave minima.

`increaseKey(`*elem e, chiave d*`)` → *void*
Incrementa della quantità d la chiave dell'elemento e in S.

`decreaseKey(`*elem e, chiave d*`)` → *void*
Decrementa della quantità d la chiave dell'elemento e in S.

Una coda a max-priorità, invece, al posto di funzioni per ricercare ed estrarre l'elemento con chiave minima, fornirà funzioni per ricercare ed estrarre quello con chiave massima.