Heap, heap indiretti e code di priorità

Paolo Boldi

15 marzo 2002

1 Introduzione

Uno *heap* (letteralmente: mucchio) è (almeno idealmente) un albero binario i cui nodi contengono dei dati, ciascuno caratterizzato da un campo *chiave*, che soddisfa le seguenti proprietà (vedi Fig. 1):

1. tutte le foglie dello heap hanno altezza (cioè distanza dalla radice) h o $h-1$ per un valore opportuno di h;
2. le foglie di altezza $h-1$ (se ce ne sono) stanno "a destra" delle foglie di altezza h;
3. per ogni nodo, la chiave del nodo è minore o uguale della chiave dei figli.

Conseguenza della definizione, naturalmente, è che la radice ha chiave minima.

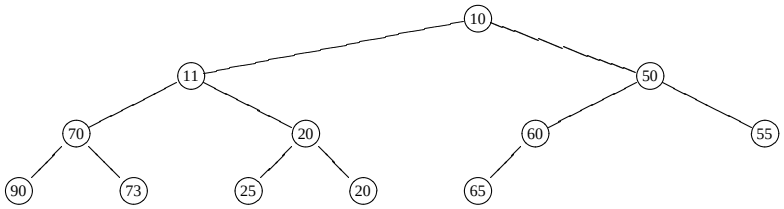

Figura 1: Esempio di heap

2 Rappresentazione mediante vettore

Di solito, per memorizzare uno heap, non si utilizza una struttura dinamica ma un semplice vettore. Più precisamente, uno heap con N nodi si memorizza in un vettore[1] $a[1], \ldots, a[N]$ in cui il nodo $a[i]$ ha come figlio sinistro il nodo $a[2i]$ e come figlio destro il nodo $a[2i+1]$ (vedi Fig. 2, dove si mostra la rappresentazione mediante vettore dello heap in Fig. 1).

Notate che:

- un nodo di indice i tale che $2i + 1 \leq N$ ha entrambi i figli;
- l'unico (eventuale) nodo che ha un solo figlio (il figlio sinistro) ha indice i tale che $2i + 1 > N$ e $2i \leq N$;
- gli altri nodi sono foglie.

In altri termini: i nodi con indici[2] $1, \ldots, \lfloor (N-1)/2 \rfloor$ hanno due figli; i nodi con indice maggiore di $N/2$ sono foglie; se N è pari, c'è un nodo (quello di indice $N/2$) con un solo figlio.

Indice	1	2	3	4	5	6	7	8	9	10	11	12
Contenuto	10	11	50	70	20	60	55	90	73	25	20	65

Figura 2: Rappresentazione dello heap di Fig. 1 mediante un vettore

3 Come si cambia il contenuto di un nodo

Supponiamo di avere uno heap (rappresentato mediante un vettore) e di aver modificato il contenuto (e la chiave) di un certo nodo $a[i]$. In generale, questo dà luogo a una violazione delle proprietà dello heap. In primo luogo, può accadere che il padre di $a[i]$ si trovi ora ad avere una chiave *maggiore* della chiave di $a[i]$. Bisogna quindi far "risalire" il nodo modificato sull'albero fino a quando siamo sicuri che il padre di $a[i]$ abbia una chiave minore o uguale a quella di $a[i]$.

[1] In questa nota si assume che il vettore abbia indici che partono da 1.

[2] La notazione $\lfloor x \rfloor$ significa "x arrotondato per difetto".

Può viceversa succedere che $a[i]$ non abbia chiave minore o uguale di entrambi i figli. Per sistemare le cose, bisognerà scambiare il suo contenuto con quello del figlio avente chiave minore, e far "scendere" il nodo lungo l'albero.

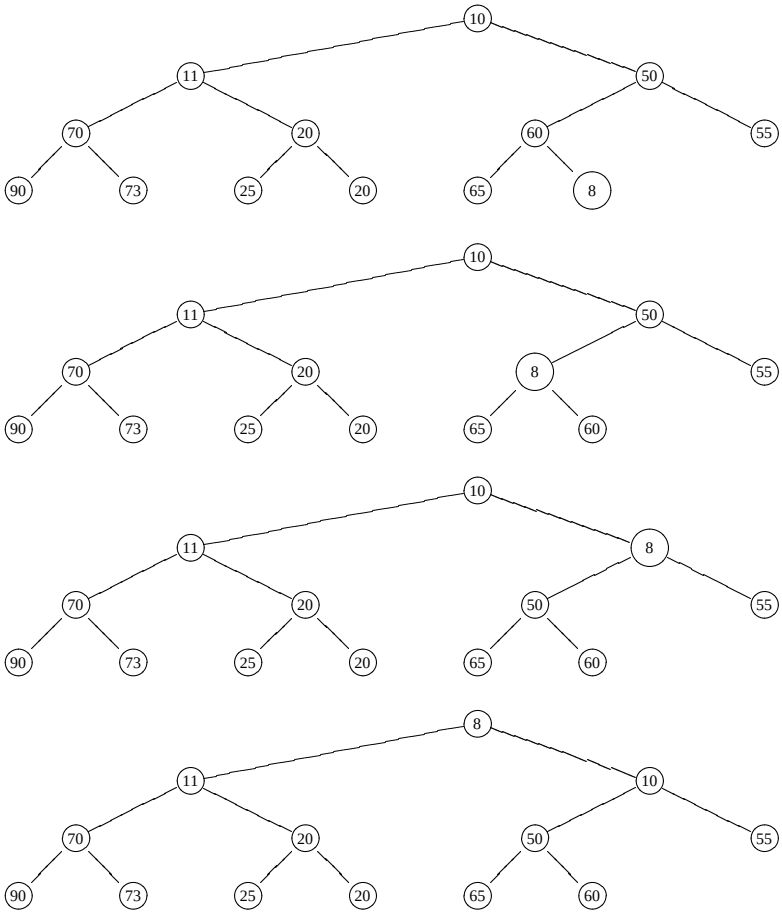

Figura 3: Ricostruzione dello heap dopo un inserimento

La seguente procedura "risistema le cose" come indicato. Per generalità, la procedura assume che lo heap si trovi nel frammento di vettore $a[l], \ldots, a[r]$, e che l'elemento modificato sia $a[i]$. Alla procedura viene passato l'indice i del nodo modificato, e due indici l e r che corrispondono al primo e all'ultimo indice dello heap considerato.

```
void heapify (int i, int l, int r) {
```

```
  DATA temp=a[i];
  while (i>=l && temp.key<a[i/2].key) {
    a[i]=a[i/2];
    i/=2;
  }
  a[i]=temp;
  while (i<=r/2) {
    child=2*i;
    if (child+1<=r && a[child+1].key<a[child].key)
      child++;
    if (temp.key<=a[child].key) break;
    a[i]=a[child];
    i=child;
  }
  a[i]=temp;
}
```

Naturalmente, si può utilizzare la procedura `heapify` anche per ripristinare le proprietà di uno heap dopo che si è inserito un nuovo elemento: è sufficiente inserire l'elemento nuovo in ultima posizione del vettore (cioè come una nuova foglia dello heap) e poi invocare la procedura perché il nuovo elemento "trovi" la sua posizione. In Fig. 3 mostriamo come si verifica l'inserimento di un nuovo elemento di chiave 8 nello heap di Fig. 1: il nuovo nodo risale l'albero (primo ciclo while) o ridiscende lungo l'albero (secondo ciclo while) fino a trovare la sua posizione naturale.

4 Aggiungere/togliere elementi da uno heap; costruzione di uno heap da zero

Disponendo della procedura `heapify` possiamo facilmente disporre di uno strumento per aggiungere/togliere elementi da uno heap:

1. per aggiungere un elemento allo heap $a[1], \ldots, a[N]$ possiamo procedere come segue:

```
a[++N]=nuovo elemento;
heapify(N,1,N);
```

2. per eliminare l'elemento i-esimo dallo heap $a[1], \ldots, a[N]$ possiamo procedere come segue:

```
a[i]=a[N--];
heapify(i,1,N);
```

Se si dispone già di un vettore $a[1], \ldots, a[N]$ e lo si vuole rendere uno heap si può o utilizzare ripetutamente la procedura di inserimento di un elemento indicata sopra, oppure alternativamente costruire lo heap "dal basso" come segue:

```
for (i=N/2;i>0;i--)
  heapify(i,i,N);
```

Il vantaggio di quest'ultima soluzione è che richiede, anche nel caso peggiore, un numero di confronti proporzionale a N (mentre l'idea di inserire un elemento dopo un altro dà luogo a un numero di confronti proporzionale a $N \log N$).

5 Heap indiretti

In qualche caso si dispone di un vettore $b[1], \ldots, b[N]$ che si vuole gestire come uno heap ma *senza modificare l'ordine degli elementi* nel vettore. In tal caso, conviene usare due vettori ausiliari p e q, con il seguente significato:

- $q[i]$ dice in quale posizione dello heap si trova l'elemento $b[i]$;
- $p[j]$ dà l'indice (nel vettore b) dell'elemento che si trova nella posizione j dello heap.

Ovviamente dovrà valere sempre che $q[p[i]] = i$ e $p[q[j]] = j$. Inizialmente $p[i] = q[i] = i$ per ogni indice i. Tutte le operazioni sullo heap viste continuano a funzionare, con le seguenti modifiche:

- le occorrenze di `a[x]` vanno sostituite con `a[p[x]]`;
- ogni assegnamento del tipo `a[i]=a[j]` va sostituito con `q[i]=q[j]; p[q[i]]=j;`

6 Un'applicazione: Code con priorità

Mostreremo come utilizzare la struttura di heap indiretto per realizzare una struttura dati molto comune, detta "coda con priorità". Supponiamo di avere un vettore $a[1], \ldots, a[n]$ contenente dei dati (generici), dichiarato come segue:

```
typedef .... DATA;
DATA a[MAXN];
```

Vogliamo costruire una struttura (che chiameremo *coda con priorità*) che consenta di memorizzare elementi presi dal vettore, associando a ciascuno un valore di *priorità* (un numero intero o reale), e che renda disponibili le seguenti operazioni:

- `enqueue(i,a)` (inserisce nella coda l'elemento $a[i]$ con priorità a);
- `isInQueue(i)` (dice se l'elemento $a[i]$ è presente nella coda);
- `dequeue()` (restituisce l'indice dell'elemento con priorità minima fra quelli presenti nella coda, e lo elimina dalla coda);
- `changePriority(i,b)` (cambia la priorità dell'elemento $a[i]$, che deve essere presente nella coda, e la setta a b).

Stiamo assumendo, per motivi di generalità, che la priorità degli elementi sia un dato non contenuto nel vettore, e che fa parte della struttura stessa della coda. Spesso, in realtà, la priorità è parte del vettore di partenza a di partenza: in tal caso, il vettore di priorità non serve, e tutte le procedure non hanno veramente bisogno del parametro di priorità perché possono leggerlo dal vettore.

Procediamo utilizzando uno heap indiretto. Dichiariamo perciò i vettori p e q e la costante N per memorizzare il numero di elementi presenti in coda:

```
int p[MAXN],q[MAXN],priority[MAXN],N;

void init() {
        int i;
        N=0;
        for (i=1;i<=n;i++)
                p[i]=q[i]=-1;
}
```

Realizziamo ora le procedure:

```
/* Ricostruisce lo heap dopo che l'elemento che nello heap si
   trova in pos. i ha cambiato priorita'. Considera solo lo
   heap dall'indice l all'indice r.
*/
void heapify (int j, int l, int r) {
        int tempq,tempp,child;
        tempq=q[j]; /* Salvo la posizione di j nello heap... */
        tempp=j;  /*...e salvo j */
        /* Faccio risalire l'elemento */
        while (j>l && priority[q[j]]<priority[q[j/2]]) {
                q[j]=q[j/2];
                p[q[j]]=j/2;
                j/=2;
        }
        /* Lo faccio scendere */
        while (j<=r/2) {
                child=2*j;
                if (child+1<=r &&
                    priority[q[child+1]]<priority[q[child]])
                        child++;
                if (priority[q[j]]<=priority[q[child]]) break;
                q[j]=q[child];
                p[q[j]]=child;
                j=child;
        }
        /* Ora rimetto i valori salvati nella posizione finale */
        q[j]=tempq;
        p[q[j]]=tempp;
}

/* Guarda se l'elemento i e' nella coda. */
int isInQueue(int i) {
        return (q[i]!=-1);
}

/* Accoda l'elemento i con priorita' a. */
```

```
void enqueue(int i, int a) {
        if (isInQueue(i)) return; /* Elemento gia‘ in coda */
        /* Metto l’elemento in fondo alla coda */
        q[i]=++N;
        p[N]=i;
        priority[i]=a;
        /* Heapifico in modo che l’elemento inserito trovi
           la sua posizione corretta */
        heapify(N,1,N);
}

/* Restituisce e toglie dalla coda l’elemento di priorita‘ minima. */
int dequeue() {
        int res;
        if (N==0) return -1; /* Coda vuota */
        /* Tolgo l’elemento in testa */
        res=q[1];
        /* Al suo posto metto l’ultima foglia */
        q[1]=q[N];
        p[q[1]]=N--;
        /* Heapifico in modo che l’elemento messo in posizione
           1 ritrovi la sua posizione corretta */
        heapify(1,1,N);
        return res;
}

/* Cambia la priorita‘ dell’elemento i a b. */
void changePriority(int i, int b) {
        if (!isInQueue(i)) return; /* Elemento non in coda */
        /* Modifico la priorita‘ e heapifico */
        priority[q[i]]=b;
        heapify(i,1,N);
}
```