Esercizi Capitolo 10 - Code con priorità e insiemi disgiunti

Alberto Montresor

19 Agosto, 2014

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.
Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

1 Problemi

1.1 Esecuzione heapsort() (Esercizio 10.2 del libro)

Si specifichi, passo dopo passo, il contenuto dello *heap* eseguendo la procedura heapsort() per ordinare alfabeticamente gli 11 elementi: O, R, D, I, N, A, M, E, N, T, O. Sorgono difficoltà dalla presenza di elementi con lo stesso valore?

Soluzione: Sezione 2.1

1.2 Dimensione dei livelli (Esercizio 10.4 del libro)

Si dimostri che in un albero *heap* di n elementi ci sono al più $\lceil n/2^{m+1} \rceil$ nodi di altezza m.

Soluzione: Sezione 2.2

1.3 Versione iterativa (Esercizio 10.5 del libro)

Per chiarezza di esposizione, maxHeapRestore() è una funzione ricorsiva. Nel caso pessimo, l'altezza dell'albero di ricorsione è potenzialmente pari a quello dell'albero *heap*; lo spazio aggiuntivo richiesto durante l'esecuzione dell'algoritmo Heapsort (oltre cioè a quello richiesto per memorizzare gli n elementi da ordinare) è quindi $O(\log n)$. Scrivere una versione iterativa di maxHeapRestore() che richieda uno spazio aggiuntivo costante.

Soluzione: Sezione 2.3

1.4 Indice iniziale dei vettori (Esercizio 10.6 del libro)

Per chiarezza di esposizione, le implementazioni della coda con priorità e di Heapsort viste in questo capitolo sono basate su vettori in cui l'indice del primo elemento è pari ad 1. Ma molti linguaggi di programmazione (Java e C/C++, per esempio) sono basati su vettori il cui primo elemento è 0. Scrivere una coda con priorità e un algoritmo Heapsort per un linguaggio con vettori di questo tipo.

Soluzione: Sezione 2.4

1.5 Altezza Heap

Dimostrare che uno *heap* con n nodi ha altezza $\Theta(\log n)$.

Soluzione: Sezione 2.5

1.6 heapBuild() basato su inserimenti

(a) Scrivere lo pseudocodice di una versione alterntiva di heapBuild() basata sulla insert(). (b) Le due procedure creano lo stesso *heap*? Dimostrare che lo fanno o produrre un esempio contrario. (c) Qual è la complessità della versione alternativa di heapBuild()?

Soluzione: Sezione 2.6

1.7 k-merging

Supponete di avere k vettori, ognuno di m elementi. Si vuole ottenere un vettore ordinato di $n = km$. Per ordinarlo, un possibile algoritmo è il seguente: si fa il merge del primo vettore con il secondo, ottenendo un nuovo vettore di $2m$ elementi; poi si fa il merge del vettore così ottenuto con il terzo, ottenendo $3m$ elementi; e così via.

1. Calcolare la complessità di questo algoritmo
2. Mostrare un algoritmo più efficiente.

Soluzione: Sezione 2.8

1.8 *Heap* ternario

Uno *heap* ternario è simile ad uno *heap* binario, tranne che per il fatto che i nodi interni possono avere fino a tre figli.

1. È possibile rappresentare uno *heap* ternario in un vettore? Spiegare
2. Qual è l'altezza di uno *heap* ternario di n elementi (in funzione di n)?
3. Si scriva la funzione deleteMax() per gli *heap* ternari. Cosa cambia rispetto agli *heap* binari? Calcolare la complessità computazionale di tale funzione.
4. Si scriva la funzione insert() per gli *heap* ternari. Cosa cambia rispetto agli *heap* binari? Calcolare la complessità computazionale di tale funzione.

Soluzione: Sezione 2.9

1.9 Max in *min-heap*

Una semplice domanda. Supponete di avere un *min-heap* di n elementi, e di cercare il valore **massimo** (non minimo). In quali posizioni del vettore cercate? Giustificare la risposta.

Soluzione: Sezione 2.10

1.10 Verifica *heap*

Sia dato un vettore A contenente n valori numerici. Scrivere un algoritmo che verifica se tale vettore è un *max-heap* di n elementi. L'algoritmo effettua semplicemente la verifica, senza modificare lo *heap*.

Soluzione: Sezione 2.11

2 Soluzioni

2.1 Esecuzione heapsort() (Esercizio 10.2 del libro)

Nella tabella successiva sono illustrati i vari passi dell'algoritmo. Nella prima colonna, lo stato del vettore ad ogni passo. Nella seconda colonna, l'operazione che ha portato il vettore in quello stato.
La tabella è ulteriormente divisa in due righe; la prima rappresenta l'esecuzione di heapBuild(), mentre la seconda rappresenta lo spostamento dell'elemento massimo dello *heap* nella posizione corretta.
Si noti che gli elementi che fanno effettivamente parte dello *heap* sono sottolineati; gli elementi non sottolineati sono elementi ordinati. Come è possibile vedere, elementi dello stesso valore non comportano difficoltà.

ORDINAMENTO	Iniziale
ORDITAMENNO	maxHeapRestore(5)
ORDNTAMEINO	maxHeapRestore(4)
ORMNTADEINO	maxHeapRestore(3)
OTMNRADEINO	maxHeapRestore(2)
TRMNOADEINO	maxHeapRestore(1)
ORMNOADEINT	$A[1] \leftrightarrow A[11]$
ROMNOADEINT	maxHeapRestore(1)
NOMNOADEIRT	$A[1] \leftrightarrow A[10]$
OOMNNADEIRT	maxHeapRestore(1)
IOMNNADEORT	$A[1] \leftrightarrow A[9]$
ONMINADEORT	maxHeapRestore(1)
ENMINADOORT	$A[1] \leftrightarrow A[8]$
NNMIEADOORT	maxHeapRestore(1)
DNMIEANOORT	$A[1] \leftrightarrow A[7]$
NIMDEANOORT	maxHeapRestore(1)
AIMDENNOORT	$A[1] \leftrightarrow A[6]$
MIADENNOORT	maxHeapRestore(1)
EIADMNNOORT	$A[1] \leftrightarrow A[5]$
IEADMNNOORT	maxHeapRestore(1)
DEAIMNNOORT	$A[1] \leftrightarrow A[4]$
EDAIMNNOORT	maxHeapRestore(1)
ADEIMNNOORT	$A[1] \leftrightarrow A[3]$
DAEIMNNOORT	maxHeapRestore(1)
ADEIMNNOORT	$A[1] \leftrightarrow A[2]$

2.2 Dimensione dei livelli (Esercizio 10.4 del libro)

Si può dimostrare la proprietà per induzione su m. Sia n_m il numero di nodi ad altezza m nell'albero heap T.
Nel caso $m = 0$, vogliamo contare le foglie di T. Abbiamo visto che tutti i nodi compresi fra 1 e $\lfloor n/2 \rfloor$ sono nodi interni, quindi il valore di n_0 è pari a $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$. Questo prova il caso base.
Nel passo induttivo, supponiamo provata la proprietà per $m-1$ e cerchiamo di dimostrarla per m. Sia T' l'albero ottenuto da T rimuovendo tutte le foglie. Per quanto detto sopra, T' ha $n' = \lfloor n/2 \rfloor$ nodi (gli ex nodi interni); inoltre, i nodi ad altezza m in T hanno altezza $m-1$ in T'; in altre parole, il numero n'_{m-1} di nodi ad altezza $m-1$ in T' è pari a n_m. Per la proprietà induttiva, si ha:

$$n_m = n'_{m-1} = \lceil \frac{n'}{2^m} \rceil = \lceil \frac{\lfloor n/2 \rfloor}{2^m} \rceil \leq \frac{n}{2^{m+1}}$$

Questo conclude la dimostrazione.

2.3 Versione iterativa (Esercizio 10.5 del libro)

Il codice della versione iterativa è molto semplice: fa uso di una variabile *stop* per decidere quando sospendere l'esecuzione e la variabile i assume, di volta in volta, l'indice dell'elemento del vettore di cui verificare la proprietà *heap*.

```
maxHeapRestore(ITEM[] A, int i, int dim)
boolean stop = false
while not stop do
    int max = i
    if l(i) ≤ dim and A[l(i)] > A[max] then
        max = l(i)
    if r(i) ≤ dim and A[r(i)] > A[max] then
        max = r(i)
    if i ≠ max then
        A[i] ↔ A[max]
        i = max
    else
        stop = true
```

2.4 Indice iniziale dei vettori (Esercizio 10.6 del libro)

Una versione in Java può essere trovata al seguente indirizzo:
http://www.disi.unitn.it/~montreso/asd/appunti/codice/Heap.java

2.5 Altezza *heap*

Il numero di nodi in uno *heap* di altezza h è compreso fra 2^h (tutti i primi $h-1$ livelli sono completi e c'è un nodo a livello h) e $2^{h+1}-1$ (tutti i livelli sono completi):

$$2^h \leq n \leq 2^{h+1} - 1 \leq 2^{h+1}$$

Applicando il logaritmo ad entrambi i lati della disequazione, si ottiene:

$$h \leq \log n \leq h + 1$$

Notando che $h + 1 \leq 2h$ per $h \geq 1$, si dimostra che $\log n = \Theta(h)$ (infatti, $c_1 h \leq \log n \leq c_2 h, \forall h \geq m$, con $c_1 = 1$, $c_2 = 2$ e $m = 1$. Per la proprietà di simmetria di Θ, questo dimostra che $h = \Theta(\log n)$.

2.6 heapBuild() basato su inserimenti

- Versione alternativa:

```
heapBuild(ITEM[] A, int n)
PRIORITYQUEUE H = new PriorityQueue(n)
for i = 1 to n do
    H.insert(A[i])
```

- Controesempio per l'uguaglianza degli *heap*: Costruire un *max-heap* a partire dai valori $4, 7, 9, 10$.

- Complessità di heapBuild(): Consideriamo $n = 2^{h+1} - 1$, quindi un albero perfetto di altezza h. Al livello più basso, ci sono $\lceil n/2 \rceil$ nodi. Se i valori sono già ordinati, ogni nuovo inserimento deve risalire fino alla radice. Quindi $T(n) \geq n/2 \log n = \Omega(n \log n)$.

  Inoltre, il costo totale può essere limitato superiormente in questo modo:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lfloor \log n \rfloor} 2^i i \\ &\leq \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \log n \\ &= \log n \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \\ &= \log n (2^{\lfloor \log n \rfloor + 1} - 1) = O(n \log n) \end{aligned}$$

  Quindi $T(n) = \Theta(n \log n)$.

2.7 Foglie in heapBuild()

Supponiamo che per assurdo che $\lfloor n/2 \rfloor + 1$ non sia una foglia. Allora $2\lfloor n/2 \rfloor + 2$ fa parte dello *heap*. Ma $2\lfloor n/2 \rfloor + 2$ è uguale a $n + 2$ o $n + 1$, che sono entrambi fuori dallo *heap*. Assurdo.

2.8 k-merging

Parte (1)

$$T(n) = 2m + 3m + \ldots + km = \sum_{i=2}^{k} im = m \sum_{i=2}^{k} i = O(mk^2) = O(nk)$$

Parte (2) Utilizziamo un *min-heap* binario di dimensione k, inizializzato con i primi valori di tutte le liste. Viene estratto il valore minimo dallo *heap*, che viene collocato nella prima posizione libera del vettore finale. Si inserisce quindi nello heap il prossimo valore preso dalla lista cui appartaneva il valore minimo appena rimosso (se ne esistono ancora). Il costo totale è quindi $n \log k$. Si è utilizzata la notazione $\langle i, v \rangle$ per operare su una struttura contenente una coppia di elementi.

```
int [ ][ ] merge(int[ ][ ] A, int k, int m)
    PRIORITYQUEUE Q = new PriorityQueue(k)      % Coda con priorità per decidere il minimo
    int[ ] V = new int[km]                      % Vettore unito
    int[ ] p = new int[k]                       % Posizione attuale di ognuno dei vettori da unire
    for i = 1 to k do
        p[i] = 2
        Q.insert(⟨i, A[i][1]⟩, A[i][1])
    int c = 1                                   % Posizione nel vettore unito
    while not Q.isEmpty() do
        ⟨i, v⟩ = Q.deleteMin()
        V[c] = v
        c = c + 1
        if p[i] ≤ m then
            Q.insert(⟨i, A[i][p[i]]⟩, A[i][p[i]])
            p[i] = p[i] + 1
    return V
```

2.9 *Heap* ternario

1. La radice è memorizzata nella casella $A[1]$; i figli della posizione i sono memorizzati nelle posizioni $3i-1$, $3i$, $3i+1$.

2. Dato che ogni nodo ha 3 figli, l'altezza di uno *heap* ternario con n nodi è $\Theta(\log_3 n)$.

3. La procedura non viene modificata, l'unica modifica deve essere apportata alla maxHeapRestore(), che deve confrontare il nodo passato con tre figli invece che con due. La complessità della deleteMax() è quindi ancora quella della maxHeapRestore(), che essendo proporzionale all'altezza dell'albero è $\Theta(\log_3 n)$.

4. La procedura non viene modificata, rimane proporzionale all'altezza dell'albero: $\Theta(\log_3 n)$

2.10 Max in *min-heap*

Ogni posizione interna (non-foglia) i ha almeno un figlio $2i$ che ha valore maggiore di quello in i. Quindi non può essere massimo. Si può quindi cercare nelle sole posizioni $[\lfloor n/2 \rfloor + 1 \dots n]$, il cui numero è comunque $O(n)$.

2.11 Verifica *heap*

```
CHECKMAXHEAP(int[] A, int n)
for i = 2 to n do
    if A[i/2] < A[i] then
        return false;
return true
```

3 Problemi aperti

3.1 Complessità (Esercizio 10.3 del libro)

Si dimostri che la procedura minHeapRestore() ha complessità $O(1 + \log(dim/i))$.