Esercitazione per il corso di Elementi di Algoritmi

Dott. Valerio Capozio

5 Giugno 2008

Esercizio 1 (Ripreso dall'appello del 6-06-07) A partire da un albero AVL vuoto, si mostrino le modifiche apportate all'albero in seguito agli inserimenti elencati in (a) e poi alle cancellazioni elencate in (b). L'albero risultante è un albero di Fibonacci? Si motivi la risposta.

(a) *Inserimenti*: 10, 16, 20, 13, 14, 12, 11;

(b) *Cancellazioni*: 20, 11, 16.

Soluzione Si vedano le soluzioni dello scritto.

Esercizio 2 (6.1 del libro) Definiamo una operazione concatenate il cui input è costituito da due insiemi S_1 ed S_2 tali che le chiavi in S_1 sono tutte minori o uguali delle chiavi in S_2 ed il cui output è la fusione dei due insiemi in uno. Progettare un algoritmo per concatenare due alberi binari di ricerca in un albero binario di ricerca. L'algoritmo deve avere tempo di esecuzione $O(h)$ nel caso peggiore, dove h è la massima altezza dei due alberi.

Soluzione Siano T e T' i due alberi binari di ricerca da concatenare. Se le chiavi di T sono tutte minori o uguali delle chiavi di T', appendiamo T' come sottoalbero destro del massimo di T, altrimenti appendiamo T come sottoalbero destro del massimo di T'. In ogni caso la proprietà di ricerca viene mantenuta. Per capire in quale caso ci si trova, è sufficiente confrontare le radici dei due alberi. Osserviamo che la ricerca del massimo, che richiede tempo non superiore all'altezza dell'albero, nel caso peggiore potrebbe essere effettuata sull'albero più alto. Poichè tutte le altre operazioni richiedono tempo $O(1)$, il tempo totale è $O(h)$, con $h = \max\{altezza(T), altezza(T')\}$.

Esercizio 3 (4.5 del libro) Implementare una nuova operazione su heap, $insert(H, k)$, che inserisce nell'heap H il nuovo valore k. Quanto costa, nel caso peggiore, costruire un heap inserendo ripetutamente elementi? Usare questa costruzione modifica il tempo di esecuzione dell'algoritmo heapSort nel caso peggiore?

Soluzione

Il codice sopra riportato inserisce il nuovo valore a partire dall'ultimo livello(quello delle foglie) iniziando a confrontare il valore inserito con quello del padre risalendo con i confronti, nel caso peggiore, fino alla radice. Il tempo richiesto da INSERT, nel caso peggiore, è quindi pari alla profondità dell'heap, ovvero $O(\log n)$. Ciò implica che inserire ripetutamente elementi richiederà, sempre nel caso peggiore, $O(n \log n)$ e quindi il tempo di esecuzione dell'heapsort non viene modificato.

```
Insert(heap H, chiave K)
n← n+1;
aggiungi nuova foglia H[n]=k;
i← n;
while i > 1 && H[i/2] < H[i] do
    scambia H[i/2] e H[i];
    i← i/2
end
```

Algorithm 1: Possibile pseudocodice

Esercizio 4 (4.6 del libro) Implementare le due nuove operazioni su heap descritte di seguito:

- $aumentaChiave(H, i, k)$, che imposta $H[i]$ al massimo tra il valore precedente e il nuovo valore k, aggiornando la struttura heap appropriatamente.
- $decrementaChiave(H, i, k)$, che imposta $H[i]$ al minimo tra il valore precedente e il nuovo valore k, aggiornando la struttura heap appropriatamente.

Soluzione Una possibile implementazione per heap basati sul massimo (come definiti nel testo):

- $aumentaChiave(H, i, k)$: l'heap può essere aggiornato spingendo verso l'alto l'elemento incrementato, come effettuato dall'operazione insert dell'esercizio precedente.
- $decrementaChiave(H, i, k)$: l'algoritmo $fixHeap(H, i)$ illustrato nella Figura 4.8 a pagina 92 del libro è sufficiente per eseguire l'operazione.

Esercizio 5 (4.9 del libro) Il problema della bandiera nazionale italiana è definito nel modo seguente. Sia A un array i cui elementi possono assumere solo uno di tre possibili valori: verde, bianco e rosso. Ordinare l'array in modo che tutti gli elementi verdi siano a sinistra, quelli bianchi al centro e quelli rossi a destra. L'algoritmo deve richiedere tempo lineare nel caso peggiore, può solo scambiare elementi, ed ha solo una variabile aggiuntiva per mantenere uno degli elementi. In particolare, non può usare contatori per mantenere il numero di elementi di un certo colore. Inoltre, una sola passata sull'array è sufficiente per completare l'ordinamento.

Soluzione Assumiamo che l'array A abbia indici nell'intervallo $[0, n-1]$. L'algoritmo effettuerà una scansione dell'array A, cercando di spostare gli elementi verdi all'inizio dell'array e gli elementi rossi alla fine dell'array. Per fare questo, l'algoritmo manterrà alcuni indici:

- Un indice **fineVerde**, tale che durante l'esecuzione dell'algoritmo $A[l] = verde$ per $0 = l = fineVerde$. All'inizio, l'algoritmo non avrà spostato nessun elemento verde all'inizio dell'array e quindi $fineVerde = -1$.
- Un indice inizioRosso, tale che durante l'esecuzione dell'algoritmo $A[l] = rosso$ per $inizioRosso = l = n - 1$. All'inizio, l'algoritmo non avrà spostato nessun elemento rosso alla fine dell'array e quindi $inizioRosso = n$.

- Un indice i, tale che durante l'esecuzione dell'algoritmo $A[l] = bianco$ per $fineVerde < l < i < inizioRosso$. All'inizio, $i = 0$.

```
algoritmo bandieraItaliana(array A);
fineVerde ← -1;
inizioRosso ← n;
i ← 0;
while i < inizioRosso do
    if A[i] = verde then
        fineVerde ← fineVerde + 1;
        scambia A[fineVerde] e A[i];
        i ← i + 1;
    else
        if A[i]=rosso then
            while A[inizioRosso - 1] = rosso do
                inizioRosso ← inizioRosso - 1;
            end
            scambia A[i] e A[inizioRosso - 1];
            inizioRosso ← inizioRosso - 1;
        else
            i ← i + 1
        end
    end
end
```

Algorithm 2: Possibile pseudocodice

Per analizzare il tempo di esecuzione dell'algoritmo, è sufficiente osservare che questo è dominato dai due cicli while innestati alle righe 4-16. Si può dimostrare che anche in questo caso i due cicli richiedono tempo $O(n)$ nel caso peggiore. Infatti è sufficiente osservare che:

- Nel caso in cui $A[i] = bianco$ oppure $A[i] = verde$, l'indice i viene incrementato di 1.
- Nel caso in cui $A[i] = rosso$, l'indice $inizioRosso$ viene decrementato di almeno 1.
- Si esce dal ciclo non appena l'indice i raggiunge l'indice inizioRosso.

Esercizio 6 (Ripreso dall'appello del 6-06-07 Dato un heap binario rappresentato tramite vettore posizionale, realizzare un algoritmo che ristrutturi l'heap in modo tale che ogni figlio sinistro sia sempre maggiore o uguale del corrispondente fratello, e che restituisca inoltre i tre elementi più grandi contenuti nell'heap. *Attenzione*: l'esercizio sarà valutato solo se corredato da adeguata descrizione del funzionamento dell'algoritmo, in base ai seguenti parametri: correttezza, efficienza e analisi di complessità.

Soluzione Si vedano le soluzioni dello scritto.