Soluzione della quinta esercitazione di Algoritmi 1

Beniamino Accattoli

29 novembre 2007

1 AVL

Gli AVL sono degli alberi binari di ricerca bilanciati, nel senso che per ogni nodo l'altezza del sottoalbero destro e l'altezza del sottoalbero sinistro differiscono al più di uno. Su ogni albero binario di ricerca T si possono effettuare la ricerca, l'inserimento e la cancellazione di una chiave in tempo $O(h)$, dove h è l'altezza di T. Si dimostra che un albero AVL con n nodi ha altezza $O(\log n)$, e quindi le tre operazioni hanno complessità $O(\log n)$. Dopo l'inserimento e la cancellazione di un nodo si potrebbe esser perso il bilanciamento dell'albero AVL, quindi bisogna applicare delle rotazioni.

1. Siano T_1 e T_2 due alberi binari di ricerca. Si supponga che: o tutte le chiavi di T_1 sono minori di tutte le chiavi di T_2 o sono maggiori di tutte le chiavi di T_2. Definendo $h = max\{altezza(T_1), altezza(T_2)\}$, scrivere una procedura in pseudo codice che concatena i due alberi binari di ricerca in un unico albero binario di ricerca in tempo $O(h)$, senza utilizzare funzioni già viste.

   risposta:
   Algoritmo concatena(alberi T_1 e T_2)
   1. **if**($T_1.val > T_2.val$) **then**
   2. esplora $\leftarrow T_1$
   3. minT $\leftarrow T_2$
   4. **else**
   5. esplora $\leftarrow T_2$
   6. minT $\leftarrow T_1$
   7. **while** (esplora ha figli sinistri) **do**
   8. esplora $\leftarrow$ esplora.figlioSinistro
   9. esplora $\leftarrow minT$

   L'algoritmo confronta le radici dei due alberi per capire qual'è l'albero maggiore (il cui puntatore è messo in *esplora*). Dopodicché si trova il minimo di tale albero e l'altro albero ne diventa il figlio sinistro. La complessità è data dalla ricerca del minimo, che in un albero binario di ricerca si fa in tempo proporzionale all'altezza dell'albero. Poiché non si sa quale dei due alberi sia il maggiore la complessità nel caso peggiore è il massimo tra le due altezze.

2. Costruire un AVL inserendo successivamente le chiavi: 16, 18, 8, 5, 4, 28, 25, 7, 13, 2, 10

Risposta: devono essere effettuate 3 rotazioni: una singola SS coinvolgente i nodi 8,5 e 4, una doppia DS coinvolgente i nodi 18, 28, e 25, e una doppia SD coinvolgente i nodi 16, 5 e 8. Nella rappresentazione posizionale la soluzione è

$$[8, 5, 16, 4, 7, 13, 25, 2, -, -, -, 10, -, 18, 28]$$

3. Si scriva una procedura $aggiungi(chiave, incremento)$ che aggiunge un valore intero *incremento* alla chiave *chiave*, se esiste. Si può assumere che non vi sono due nodi con la stessa chiave. L'operazione deve avere complessità $O(\log n)$, dove n è il numero di nodi dell'albero.

   risposta: Non c'è bisogno di implementare le rotazioni, basta notare che la complessità richiesta è pari a quella delle operazioni fornite da un AVL e che quindi si possono usare queste per implementare *aggiungi*:

   Algoritmo aggiungi(intero chiave, intero incremento)
   1. nodo ← trova(chiave)
   2. **if** (nodo non è vuoto) **then**
   3. cancella(nodo)
   4. inserisci(chiave+incremento)

4. Supporre di avere a disposizione una struttura dati AVL+ che estende gli AVL in modo che ogni nodo u dell'albero abbia un campo *somma* contenente la somma di tutte le chiavi del sottoalbero radicato in u. Scrivere una procedura $SommaChiaviMinoriDi(intero\ k, Albero\ T)$ in pseudo codice che prende un intero k e restituisce la somma di tutte le chiavi minori di k nell'albero AVL+ T. Si può supporre che k sia effettivamente una chiave in T

   risposta: Non basta trovare la chiave k e restituire il campo somma del suo sotto albero sinistro, perché k potrebbe essere figlio destro di un qualche nodo x. Tale nodo per definizione di albero binario di ricerca ha chiave minore di k, ma non è nel suo sottoalbero sinistro. L'algoritmo corretto:

   Algoritmo SommaChiaviMinoriDi(intero k, Albero T) → intero
   1. esplora ← T
   2. sommaTot ← 0
   3. **while** (esplora.chiave ≠ k)
   4. **if** (esplora.chiave > k) **then**
   5. esplora ← esplora.figlioSinistro
   6. **else**
   7. sommaTot ← sommaTot + esplora.chiave + esplora.figlioSinistro.somma
   8. esplora ← esplora.figlioDestro
   9. **if** (esplora.figlioSinistro è vuoto) **return** sommaTot
   10. else **return** (sommaTot + esplora.figlioSinistro.somma)

5. Definiamo induttivamente l'altezza $alt(v)$ di un nodo v in un albero binario T. Se v è una foglia allora $alt(v) = 1$. Se v non è una foglia allora

$alt(v) = 1 + max\{alt(v_1), alt(v_2)\}$ dove v_1 e v_2 sono i figli di v. Per ogni nodo x definiamo due quantità:

- l_x='lunghezza del cammino più lungo da x ad una foglia sotto x'
- c_x='lunghezza del cammino più corto da x ad una foglia sotto x'

dimostrare per induzione sull'altezza di un nodo che ogni nodo x di un albero AVL verifica $l_x \leq 2 \cdot c_x$, seguendo i seguenti tre passi:

- dimostrare la base induttiva, con $alt(x) = 1$
- per un generico nodo x con due figli esprimere l_x e c_x in funzione dei valori l_y, l_z, c_y, c_z dei figli y e z. Considerare anche il caso in cui x ha un solo figlio.
- dimostrare il passo induttivo, considerando prima il caso in cui x ha un solo figlio e poi quello in cui ne ha due.

risposta: *base induttiva*) se $alt(x) = 1$ allora x è una foglia e si ha $l_x = c_x = 0$ e la disuguaglianza è verificata. Se un generico nodo x ha due figli allora si ha che $l_x = 1 + max\{l_y, l_z\}$ e $c_x = 1 + min\{c_y, c_z\}$. Se invece ha un solo figlio y si ha $l_x = l_y + 1$ e $c_x = c_y + 1$. *Passo induttivo*) se $alt(x) > 1$ i figli di x hanno altezza minore di x quindi possiamo utilizzare l'ipotesi induttiva. Se x ha un solo figlio y l'ipotesi induttiva ci dice che $l_y \leq 2 \cdot c_y$, e quindi $l_y + 1 = l_x \leq 2 \cdot c_y + 1 < 2 \cdot (c_y + 1) = 2 \cdot c_x$. Se x ha due figli y e z l'ipotesi induttiva consiste in $l_y \leq 2 \cdot c_y$ e $l_z \leq 2 \cdot c_z$. Si deve ora mostrare che

$$l_x = 1 + max\{l_y, l_z\} \leq 2 \cdot (1 + min\{c_y, c_z\}) = 2 \cdot c_x$$

ovvero

$$1 + max\{l_y, l_z\} \leq 2 + min\{2 \cdot c_y, 2 \cdot c_z\}$$

in un AVL l_y e l_z possono differire al più di uno. Se sono uguali allora la disequazione è vera per le ipotesi induttive. Se sono diversi supponiamo che sia $l_y < l_z = l_y + 1$. Si deve quindi mostrare che

$$2 + l_y = 2 + min\{2 \cdot c_y, 2 \cdot c_z\}$$

ovvero che $l_y \leq min\{2 \cdot c_y, 2 \cdot c_z\}$. Poiché per ipotesi induttiva vale $l_y \leq 2 \cdot c_y$ l'unica cosa da provare è che se $c_z \leq c_y$ vale $l_y \leq 2 \cdot c_z$. Basta ricordarsi che per l'ipotesi induttiva e per l'assunzione che l_z è maggiore di 1 di l_y vale $l_y < l_z \leq 2 \cdot c_z$. Q.e.d.

2 Ordinamento

[Questa parte degli esercizi non è stata svolta a lezione per mancanza di tempo]

1. Scrivere un algoritmo $Preprocess(array\ A, intero\ k)$ che dato un vettore A di n interi nell'intervallo $[1, k]$ lo preprocessa in modo da poter poi rispondere a interrogazioni del tipo: quanti interi cadono nell'intervallo $[a, b]$?, per ogni a e b in tempo $O(1)$. L'algoritmo deve richiedere tempo di

preprocessamento $O(n+k)$. Scrivere anche la procedura di interrogazione $QuantiInteri(intero\ a, intero\ b) \rightarrow intero$. Per tutta l'esercitazione i vettori iniziano dalla locazione 1 e finiscono alla locazione n

risposta:
Algoritmo Preprocess(*array* A, *intero* k)
1. **for** i ← 1 **to** n **do**
2. occorrenze[A[i]]++
3. **for** j ← 2 **to** k **do**
4. occorrenze[j] ← occorrenze[j]+occorrenze[j-1]

Algoritmo QuantiInteri(a,b) → intero
1. **return** occorrenze[b]-occorrenze[a]

2. Scrivere un algoritmo $ordina(array\ A)$ che ordina un vettore di valori booleani. Tale algoritmo deve impiegare tempo lineare nella dimensione dell'array, deve scorrere l'array senza mai tornare indietro e deve scorrerlo non più di una volta, può solo scambiare elementi e non può usare altri vettori.

risposta:
Algoritmo ordina(*array* A)
1. inizioUni ← n+1
2. i←1
3. **while** $i < inizioUni$ **do**
4. **if** $(A[i] = 1)$ **then**
5. inizioUni ← inizioUni-1
6. Scambia A[i] e A[inizioUni]
7. **else** i ← i+1

3. Generalizzare l'algoritmo precedente: supporre che i valori siano ora ternari (cioè 0, 1 o 2), e scrivere un algoritmo analogo (sotto gli stessi vincoli, eccetto che è ora possibile scambiare l'elemento corrente con uno precedente)

risposta:
Algoritmo ordina(*array* A)
1. inizioDue ← n+1
2. fineZeri ← 0
3. i←1
4. **while** $i < inizioDue$ **do**
5. **if** $(A[i] = 0)$ **then**
6. fineZeri ← fineZeri+1
7. Scambia A[i] e A[fineZeri]
8. i ← i+1
9. **else**
10. textbfif $(A[i] = 2)$ **then**
11. inizioDue ← inizioDue-1
12. Scambia A[i] e A[inizioDue]