Sesta Esercitazione di Algoritmi 1

Beniamino Accattoli

6 dicembre 2007

Nei seguenti esercizi i vettori iniziano dalla locazione 1 e finiscono alla locazione n

1 Alberi, visite e rappresentazioni

Una visita di un albero è un algoritmo che passa per ogni nodo dell'albero. Può essere fatto in diversi modi. La visita simmetrica è il seguente algoritmo ricorsivo, dove $visita(nodo\ n)$ è una qualche operazione che bisogna fare su ogni nodo:

Algoritmo visitaSimmetrica(albero T)
1. **if** (T.figlioSin non è vuoto) **then** visitaSimmetrica(T.figlioSin)
2. visita(T)
3. **if** (T.figlioDes non è vuoto) **then** visitaSimmetrica(T.figlioDes)

La visita in pre-ordine consiste nel visitare prima il nodo corrente e poi i figli, ovvero nello scambiare le prime due righe dell'algoritmo dato. La visita in post-ordine invece consiste nello scambiare le ultime due righe.

1. Sia GDHBAECJIKF la sequenza prodotta dalla visita simmetrica (in-order) di un albero e sia ABDGHCEFIJK la sequenza prodotta da una visita in pre-ordine dello stesso albero T, che ha 11 nodi etichettati con A, B, C, D, E, F, G, H, I, J, K. Ricostruire T.

   Risposta: in rappresentazione vettoriale (vedere l'ultimo esercizio di questa sezione per la definizione di questa rappresentazione)

   $$[A, B, C, D, -, E, F, G, H, -, -, -, -, I, -, -, -, -, -, -, -, -, -, -, -, -, -, J, K]$$

2. Scrivere un algoritmo ricorsivo $buildTree(vettori\ inOrd,\ preOrd) \rightarrow albero$ che prende due vettori di uguale dimensione rappresentanti la visita in-ordine e in pre-ordine di un albero e che ricostruisce l'albero in questione. Si può assumere di avere una funzione $trovaIndice(chiave\ c,\ vettore\ vett) \rightarrow intero$ che prende una chiave, un vettore e restituisce l'indice della chiave nel vettore. Si assume anche che se $vett$ è un vettore allora per $i \leq j$ la notazione $vett[i; j]$ indica il sottovettore di $vett$ composto dalle locazioni comprese tra i e j (compresi), mentre se $i > j$ indica il vettore vuoto.

   Risposta:

Algoritmo buildTree(vettori inOrd, preOrd) → albero
1. **if** (dim(inOrd)=0) **then return null**
2. **else**
3. **new** nodo
4. nodo.val ← preOrd[1]
5. i ← trovaIndice(preOrd[1], inOrd)
6. nodo.figlioSin ← buildTree(inOrd[1;i-1], preOrd[2;i])
7. nodo.figlioDes ← buildTree(inOrd[i+1;dim(inOrd)], preOrd[i+1;dim(preOrd)])
8. **return** nodo

Se i due vettori sono vuoti (hanno uguale dimensione per ipotesi, quindi basta controllarne uno) si restituisce null, altrimenti si prende la prima chiave nella visita in pre-ordine, ovvero la radice dell'albero, e la si mette in un nuovo nodo. Dopodicché si ricerca l'indice i di tale chiave nella visita in-ordine. Quest'indice divide la visita in-ordine in due sottovettori, quello con le chiavi del sotto albero sinistro (a sinistra dell'indice) e quello con le chiavi del sotto albero destro (a destra dell'indice). Questo permette di identificare tali sotto alberi anche nella visita in pre-ordine. Le chiamate ricorsive sfruttano quest'idea.

3. Scrivere una procedura *visitaPerLivelli*(*albero* T) che prende un albero T rappresentato come una struttura linkata e lo visita per livelli senza passare alla rappresentazione posizionale (o vettoriale, vedi l'esercizio successivo per la definizione). Supporre che la visita di un nodo v consista nel chiamare una procedura $visita(v)$. Suggerimento: usare una struttura dati d'appoggio.

Algoritmo visitaPerLivelli(albero T)
1. Coda Q
2. Q.accoda(T)
3. **while** (Q non è vuota) **do**
4. nodo ← Q.estraiTesta
5. visita(nodo)
6. **if** (nodo.figlioSin non è vuoto) **then** Q.accoda(nodo.figlioSin)
7. **if** (nodo.figlioDes non è vuoto) **then** Q.accoda(nodo.figlioDes)

L'algoritmo visita un nodo e mette i figli nella coda. Dopodicché come prossimo nodo da visitare prende la testa della coda. Per capire come funziona si consiglia di eseguirlo su un qualsiasi albero binario completo di 3 livelli.

4. Un albero binario può essere rappresentato con un vettore. L'idea è che la prima posizione contenga la chiave della radice, e che se la posizione di un nodo è i allora le posizioni dei suoi due figli sono $2i$ e $2i+1$. Inoltre si usa un carattere speciale '-' per indicare l'assenza di un nodo. Scrivere una procedura *cambiaRappresentazione*(*vettore* v) → *albero* che prende un albero rappresentato come vettore e passa alla rappresentazione linkata in tempo $O(dim(v))$.

Risposta: Si crea un vettore di nodi in tutto uguale al vettore originale,

e poi lo si linka secondo la regola nota. Si vede facilmente che la complessità dell'algoritmo è $O(dim(v))$ nel caso peggiore, perché il numero di operazioni per ogni locazione del vettore è maggiorato da una costante. L'algoritmo seguente procede dalla fine del vettore verso l'inizio cosicché possa linkare i padri ai figli mentre crea i nodi, altrimenti bisognerebbe passare due volte per il vettore. Tale ottimizzazione è un buon esempio di ottimizzazione poco utile dal nostro punto di vista, perché non cambia la complessità asintotica dell'algoritmo, ma solo la costante moltiplicativa nascosta nella notazione asintotica. Si veda il prossimo esercizio per un'ottimizzazione sostanziale:

Algoritmo cambiaRappresentazione(vettore v) → albero
1. n ← dim(v)
2. **vettore** vettNodi di n nodi
3. inizioFoglie ← $\lfloor \frac{n}{2} \rfloor + 1$
4. **for** i ← n **downto** inizioFoglie **do**
5. **if** (v[i] ≠ -) **then**
6. vettNodi[i].chiave ← v[i]
7. **for** i ← inizioFoglie -1 **downto** 1 **do**
8. **if** (v[i] ≠ -) **then**
9. vettNodi[i].chiave ← v[i]
10. vettNodi[i].figlioSin ← vettNodi[2i]
11. vettNodi[i].figlioSin ← vettNodi[2i+1]

5. La rappresentazione vettoriale ha un problema: la dimensione del vettore può essere esponenzialmente più grande del numero dei nodi non vuoti nel vettore. Ripetere l'esercizio precedente impiegando tempo $O(n)$ dove n è il numero dei nodi non vuoti nell'albero. E' consentito aggiungere dei parametri alla procedura.

Risposta: la soluzione data in aula è la seguente. E' la più semplice, ma anche poco elegante (dove l'istruzione 'new nodo(v[i])' crea un nodo con il valore v[i] come chiave):

Algoritmo cambiaRapp(vettore v, intero i) → albero
1. if (v[1] ≠ -) then 2. nodoCorrente ← **new** nodo(v[i])
3. **if** ($i < \frac{dim(v)}{2}$) **then**
4. **if** (v[2i]≠ -) **then** nodo.figlioSin ← cambiaRapp(v, 2i)
5. **if** (v[2i+1]≠ -) **then** nodo.figlioSin ← cambiaRapp(v, 2i+1)
6. **return** nodo
7. **else return null**

L'algoritmo viene chiamato la prima volta con l'indice i pari ad 1 (il primo elemento del vettore). Si smette di fare chiamate ricorsive quando si è superata la metà del vettore, perché da lì in poi vi sono solo foglie.

Tale soluzione non è la migliore possibile perché il vettore v diventa una variabile globale alla quale ogni chiamata ricorsiva accede, e questo stile di programmazione aumenta le probabilità di commettere un errore, e ne rende più difficile l'individuazione. Inoltre l'indice i non è propriamente

un paramentro del problema, né ha un significato al di là dell'implementazione. Per risolvere questi difetti bisogna inevitabilmente complicare l'algoritmo. L'idea è quella di usare una coda, come nella visita per livelli. La relativa difficoltà è che bisogna portarsi dietro allo stesso tempo l'indice per il quale creare il nodo, il padre del nodo, e l'informazione su quale figlio bisogna inserire, se destro o sinistro (in realtà tale informazione è deducibile analizzando l'indice, ma per semplicità si evita questa ottimizzazione di spazio). Dunque si usa un tipo record Terna, che ha la forma (indice, padre, tipoFiglio). Il campo tipoFiglio è un booleano, e si assume che vi siano due costanti booleane *sinistro* e *destro*, con valori opposti. Si noti che nella rappresentazione posizionale scorrendo il vettore si effettua già una visita per livelli. Quello che si vuole è una visita per livelli che eviti le locazioni vuote, per questo serve la coda. Come prima si smette di inserire elementi nella coda quando si è superata la metà del vettore. L'algoritmo:

Algoritmo cambiaRappresentazione(vettore v) → albero
1. coda Q di Terne
3. **if** (v[1]≠ -) **then**
4. radice ← **new** nodo(v[1])
5. **if** (v[2]≠ -) **then**
6. Q.accoda(**new** terna(2, radice, sinistro))
7. **if** (v[3]≠ -) **then**
8. Q.accoda(**new** terna(3, radice, destro))
9. **while** (Q non è vuota) **do**
10. ternaCorrente ← Q.estraiTesta
11. i ← ternaCorrente.indice
12. nuovoNodo ← **new** nodo(v[i])
13. **if** (ternaCorrente.figlio=sinistro) **then**
14. ternaCorrente.nodo.figlioSin ← nuovoNodo
15. **else** ternaCorrente.nodo.figlioDes ← nuovoNodo
16. **if** ($i < \frac{dim(v)}{2}$) **then**
17. **if** (v[2i]≠ -) **then**
18. Q.accoda(**new** terna(2i,nuovoNodo, sinistro))
19. **if** (v[2i+1]≠ -) **then**
20. Q.accoda(**new** terna(2i+1,nuovoNodo, destro))
21. **return** radice
22. **else return null**

2 Tavole Hash

1. Sia t un qualunque numero intero, m la dimensione di una tavola hash e h la funzione di hashing, di dominio U e codominio $\{1, .., m\}$, ovvero $h : U \to \{1, .., m\}$. Dimostrare che se la cardinalità dell'universo U delle chiavi soddisfa $|U| \geq t \cdot m$, allora esistono almeno t chiavi distinte $k_1, .., k_t$ tali che $h(k_1) = h(k_2) = .. = h(k_t)$. Per capire meglio: se avete m cassetti, e almeno $t \cdot m$ magliette da disporre in questi cassetti, allora dovete dimostrare che esiste un cassetto con almeno t magliette.

Risposta: si supponga, per assurdo, che la tesi non sia vera. Dunque disposte tutte le magliette, ogni cassetto ha meno di t magliette. Quindi sommando il contenuto di tutti i cassetti si ottiene un numero di magliette strettamente minore di $t \cdot m$. Ma per ipotesi le magliette sono *almeno* $t \cdot m$, quindi si ha un assurdo. Q.e.d.

2. *Quest'esercizio non è stato svolto in aula per mancanza di tempo.* Una funzione hash h è perfetta se è iniettiva, ovvero manda elementi diversi su elementi diversi. Se si utilizza una funzione hash perfetta allora la dimensione della tavola dev'essere almeno quanto quella dell'universo delle chiavi. Trovare una funzione hash perfetta per l'insieme S delle stringhe di lunghezza 3 sull'alfabeto $\{A, B, C\}$. Suggerimento: qual'è la cardinalità di S? Ispirarsi alla numerazione posizionale (ovvero la nostra, non quella romana).

   Risposta: La cardinalità di S è 27. quindi una funzione hash perfetta, e minima rispetto a questo requisito, avrà la forma $h : S \to \{1, .., 27\}$. Un modo è quello di vedere le lettere A, B e C come le cifre 0, 1 e 2 e di interpretare una stringa in S come un numero in ternario. Difatto in ternario, con tre cifre, si possono rappresentare i numeri da 0 a $3^3 - 1 = 26$, ovvero esattamente 27 numeri. Quindi basta prendere per h la funzione che ne dà la lettura decimale. Volendo essere formali si deve usare una funzione (iniettiva) di traduzione dell'alfabeto $t : \{A, B, C\} \to \{0, 1, 2\}$ tale che, ad esempio, $t(A) = 0$, $t(B) = 1$ e $t(C) = 2$ e poi definire $h(xyz) = 3^2 \cdot t(x) + 3 \cdot t(y) + t(z)$, dove xyz indica la stringa ottenuta concatenando le cifre ternarie x, y e z. La funzione è iniettiva per costruzione.