Esercitazione 4
Algorithmi e Strutture Dati (Informatica)
A.A 2015/2016

Tong Liu

April 5, 2016

Alberi

Esercizio 1 *

[Libro 5.2] Dato un albero ordinato i cui nodi contengono valori interi, se ne vogliono cancellare tutte le foglie per le quali il percorso radice-foglia ha somma complessiva dei valori uguale a k. Fornire una procedura di complessità ottima.

Suggerimento: utilizzare l'algoritmo pre-visita(DFS), modificare tale funzione in modo che la funzione ritorna un valore bool per avvisare al suo parente della cancellazione.

Miglioramento: riguardo la soluzione del libro, un vostro collega ha proposto due modifiche per rendere la soluzione più efficiente. 1) non usare più la variabile somma e usare solo k, decrementa il valore k con valore letto ad ogni node. 2) mettere una condizione, se $k \leq 0$, ferma la visita di rami successivi. Questa procedura ha una condizione, è che tutti i valori contenenti sono valori positivi.

Esercizio 2.1

[Libro 5.3] La larghezza di un albero ordinato é il numero massimo di nodi che stanno tutti al medesiimo livello. Si fornisca una funzione che calcoli in tempo ottimo la larghezza di un albero T di n nodi.

Suggerimento: Usare un'array globale A che ha indice il livello e contenuto il numero di nodi in livello dell'indice. Eseguire la vista in profondità DFS e aggiornare A nel processo di visita. Finita la visita, scorre A per tornare un valore massimo che sarà la largezza dell'albero.

Svolgimento

L'algoritmo è descritto in Pseudocodice 1. Si basa su DFS ,che si appoggia su un vettore dinamico, indicizzato sui livelli, per contare il numero di nodi per livello, e poi selezionare il valore massimo. La complessità in tempo è $O(n)$, la complessità in spazio è $O(n)$

Algorithm 1 Larghezza-alberti-binari

```
1: procedure LARGHEZZA(Tree t, VECTOR A, Integer level)
2:     Tree u ← t.leftmostChild()
3:     level ← level + 1
4:     while u ≠ nil do
5:         A[level] ← A[level] + 1
6:         larghezza(u,A,level)
7:         u ← u.rightSibling()
8:     end while
9: end procedure
```

Esercizio 2.2

Nell'esercizio 2.1, in caso di un albero binario, come sarebbe la procedura?

Suggerimento:
Invece di usare v.leftmostchild() e v.rightsibling(), avremmo v.left e v.right, provate a capire la loro differenza e cercate di usarlo in modo opportuno.

Svolgimento

Sempre basato su DFS e un vettore dinamico e alla fine calcolare il valore massimo in vettore A. La complessità in tempo è sempre $O(n)$, la complessità in spazio è sempre $O(n)$. Dettagli in Pseudocodice 2

Algorithm 2 Larghezza-alberti-binari

```
1: procedure LARGHEZZA(Tree t, VECTOR A, Integer level)
2:     if T ≠ nil then
3:         A[level] ← A[level] + 1
4:         larghezza(t.left, A, level + 1)
5:         larghezza(t.right, A, level + 1)
6:     end if
7: end procedure
```

Esercizio 3

[Libro 5.9] Dato un albero binario, i cui nodi contengono elementi interi, si scriva una procedura di complessità ottima per ottenere l'albero inverso, ovvero un albero in cui il figlio destro (con relativo sottoalbero) è scambiato con il figlio sinistro (con relativo sottoalbero).

Suggerimento:
Per l'invertire i due figli di un nodo v, basta fare v.left $\leftrightarrow$ v.right, modificare DFS e usare bene v.left e v.right.

Svolgimento

L'algoritmo 3, molto semplicemente, scambia il sottoalbero destro e sinistro, e lavora ricorsivamente su di essi allo stesso modo. Il tempo di calcolo è $O(n)$.

Algorithm 3 inverti

```
1: procedure INVERTI(Tree T)
2:     if T = nil then
3:         return
4:     end if
5:     v.left ↔ t.value
6:     inverti(t.left)
7:     inverti(t.right)
8: end procedure
```

Esercizio 4

[Libro 5.10] Dato un albero binario i cui nodi contengono interi, si vuole aggiungere ad ogni foglia un figlio contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia. Si scriva una procedura ricorsiva di complessità ottima.

Suggerimento:
Portare un valore v nella funzione ricorsiva di visita e lo aggiorna ad ogni lettura del nodo. Mettere una condizione if nella funzione per vedere se un nodo è foglia(2 figli sono nil), se lo è, si può quindi fare t.insertLeft(v) che associate un nodo sinistro che ha valore v ad albero t. Infatti, ci sono pochissime righe di differenza tra questo esercizio e esercizio 3.

Svolgimento

La procedura seguente (Algoritmo 4) risolve il problema in tempo $O(n)$ e viene iniziato da addChild(t,0).

Algorithm 4 addChild

```
1: procedure ADDCHILD(Tree T,Integer v)
2:     if T = nil then
3:         return
4:     end if
5:     v ← v + t.value
6:     if t.left = t.right = nil then
7:         t.insertLeft(v)
8:     else
9:         addChild(t.left,v)
10:        addChild(t.right,v)
11:    end if
12: end procedure
```

Alberi binari di ricerca

Esercizio 5

[Libro 6.6] Dati due alberi binari di ricerca T_1 e T_2 tali che le chiavi in T_1 sono tutte minori delle chiavi in T_2, si scrive una procedura che restituisce un albero di ricerca contenente tutte le chiavi in tempo $O(h)$, dove h è l'altezza massia dei due alberi.

Svolgimento

Si cerca il massimo valore v contenuto in T_1, e si attacca la radice di T_2 come figlio destro di v, attraverso l'operazione link() definita nel libro. Si noti che v non può avere un figlio destro, altrimenti questo sarebbe il massimo. Il costo dell'operazione è dato dalla ricerca del minimo, che è $O(h)$ dove h è l'altezza massima fra i due alberi.

Algorithm 5 concatenate

```
1: procedure CONCATENATE(Tree T1,Tree T2)
2:     TREE v ← max(T1)
3:     link (v, T2, T2.value)
4: end procedure
```

Algorithm 6 link

```
1: procedure LINK(Tree p,Tree u, ITEM x)
2:     if u ≠nil then
3:         u.parent ← p
4:     end if
5:     if p ≠nil then
6:         if x < p.key then
7:             p.left ← u
8:         else
9:             p.right ← y
10:        end if
11:    end if
12: end procedure
```

Algorithm 7 max

```
1: procedure MAX(Tree T_1)
2:     while T.right ≠ nil do
3:         T ← T.right
4:     end while
5:     Return T
6: end procedure
```

Nota

Fino a qui, abbiamo visto quanto importante la funzione DFS(visita in profondità) per gli aberi, la logica essenziale e la PARTENZA di tutte le soluzioni. Da notare anche, per alberi generali, il formato della procedura è descritto in Algoritmo 8, per alberi binari invece si usa l'Algoritmo 9. Si consiglia vivamente di memorizzare bene entrambi, confrontarli bene e non fare confusione!

Algorithm 8 DFS per Alberi

```
1: procedure DFS(Tree t)
2:     % operazioni varie su t
3:     Tree u ← t.leftmostChild()
4:     while u ≠ nil do
5:         DFS(t)
6:         u ← u.rightSibling()
7:     end while
8: end procedure
```

Algorithm 9 DFS per Alberi Binari

```
1: procedure DFS(Tree t)
2:     if t = nil then
3:         return
4:     end if
5:     % operazioni varie su t
6:     DFS(t.left)
7:     DFS(t.right)
8: end procedure
```