Esercizi Capitolo 5 - Alberi

Alberto Montresor

23 settembre 2010

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 Larghezza (Esercizio 5.4 del libro)

La larghezza di un albero ordinato è il numero massimo di nodi che stanno tutti al medesimo livello. Si fornisca una funzione che calcoli in tempo ottimo la larghezza di un albero ordinato T di n nodi.

Soluzione: Sezione 2.1

1.2 Visite (Esercizio 5.5 del libro)

Gli ordini di visita di un albero binario di 9 nodi sono i seguenti:

- A, E, B, F, G, C, D, I, H (anticipato)
- B, G, C, F, E, H, I, D, A (posticipato)
- B, E, G, F, C, A, D, H, I (simmetrico).

Si ricostruisca l'albero binario e si illustri *brevemente* il ragionamento.

Soluzione: Sezione 2.2

1.3 Albero inverso (Esercizio 5.9 del libro)

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).

Soluzione: Sezione 2.3

1.4 Aggiungi un figlio (Esercizio 5.10 del libro)

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.

Soluzione: Sezione 2.4

1.5 Albero binario completo (Esercizio 5.12 del libro)

Un albero binario completo di altezza k è un albero binario in cui tutti i nodi, tranne le foglie, hanno esattamente due figli, e tutte le foglie si trovano al livello k. Si dimostri per induzione che, in un albero binario completo di altezza k, il numero dei nodi è $2^{k+1} - 1$ ed il numero delle foglie è 2^k.

Soluzione: Sezione 2.5

1.6 Visite iterative (Esercizio 5.13 del libro)

Utilizzando le pile, si scrivano tre procedure iterative di complessità ottima per effettuare, rispettivamente, le visite anticipata, differita e simmetrica di un albero binario.

Soluzione: Sezione 2.6

1.7 Altezza minimale

Dato un albero binario T, definiamo *altezza minimale* di un nodo v la minima distanza di v da una delle foglie del suo sottoalbero.

- Descrivere un algoritmo che riceve in input un nodo v e restituisce la sua altezza minimale.
- Calcolare la complessità in tempo dell'algoritmo proposto.

Soluzione: Sezione 2.7

1.8 Alberi strutturalmente diversi

Due alberi si dicono "strutturalmente" diversi se disegnando correttamente i figli destri e sinistri, si ottengono figure diverse. Ad esempio, un nodo radice con un figlio destro è diverso da un nodo radice con un figlio sinistro.

1. Si dica quanti sono i possibili alberi binari strutturalmente diversi composti da 1, 2, 3 e 4 nodi.
2. Dare una formula di ricorrenza per il caso generale di n nodi.
3. Data la ricorrenza al punto 2) trovare il più stretto limite asintotico inferiore che riuscite a trovare. Suggerimento: la formula risultante non vi ricorda nulla? Guardate negli appunti.

Soluzione: Sezione 2.8

1.9 Alberi pieni

Un albero binario "pieno" è un albero binario in cui tutti i nodi hanno esattamente 0 o 2 figli, e nessun nodo ha 1 figlio. Scrivere un programma ricorsivo che valuti P_n, ovvero il numero di alberi binari strutturalmente diversi che si possono ottenere con n nodi. Si valuti la complessità dell'algoritmo risultante.

Soluzione: Sezione 2.9

1.10 Altezza specificata

Dato un albero binario con radice T e un intero k, scrivere un algoritmo in pseudocodice che restituisca il numero di nodi di T che hanno altezza k. Calcolare la complessità in tempo e in spazio dell' algoritmo proposto.

Bertossi, Montresor. Algoritmi e Strutture di Dati. ©2010 De Agostini Scuola

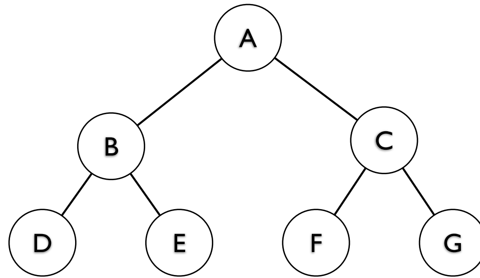

Figura 1: Un albero con lunghezza di cammino 10

Soluzione: Sezione 2.10

1.11 Lunghezza di cammino

In un albero binario, definiamo lunghezza di cammino come la somma delle distanze dei nodi dalla radice. Ad esempio, la lunghezza del cammino dell'albero in Figura 1 è pari a 10.
Scrivere un algoritmo in pseudo-codice per calcolare la lunghezza di cammino in un albero binario t. Discutere la complessità in tempo dell'algoritmo proposto in funzione del numero di nodi dell'albero.

Soluzione: Sezione 2.11

2 Soluzioni

2.1 Larghezza (Esercizio 5.4 del libro)

È possibile modificare semplicemente un algoritmo di visita in ampiezza per assegnare un livello ad ognuno dei nodi, e contare quanti nodi appartengono allo stesso livello. Il costo dell'algoritmo risultante è ovviamente $O(n)$.

```
visitaAmpiezza(TREE t)
    integer larghezza ← 1
    integer level ← 1
    integer count ← 1
    QUEUE Q ← Queue()
    Q.enqueue(t)
    t.level ← 0
    while not Q.isEmpty() do
        TREE u ← Q.dequeue()
        if u.level ≠ level then
            level ← t.level
            count ← 0
        count ← count + 1
        if count > larghezza then larghezza ← count
        u ← u.leftmostChild()
        while u ≠ nil do
            u.level ← t.level + 1
            Q.enqueue(u)
            u ← u.rightSibling()
    return larghezza
```

2.2 Visite (Esercizio 5.5 del libro)

L'albero risultante è mostrato in Figura 2. È possibile ragionare nel modo seguente:

- A è la radice, perchè visitata per prima nell'ordine anticipato;
- A ha figlio sinistro e destro, perchè B,E,G,F,C appartengono al suo sottoalbero sinistro e D,H,I al suo sottoalbero destro (per la visita in ordine simmetrico);
- E è il figlio sinistro di A, perchè il primo ad essere visitato dopo A nella visita anticipata;
- B ha figlio sinistro e destro, perchè B appartiene al sottoalbero sinistro di E e G,F,C al suo sottoalbero destro (per la visita in ordine simmetrico);
- B è figlio sinistro di E (per la visita in ordine simmetrico);
- F è figlio destro di E (per la visita in ordine anticipato);
- G e C sono figli sinistri e destro di F (per la visita in ordine simmetrico);
- D è figlio destro di A (in quanto primo ad essere esaminato dopo il sottoalbero sinistro di A nell'ordine anticipato);
- Restano H e I; poichè I viene visitato prima di H nell'ordine anticipato, I deve essere padre di H e figlio di D;
- poichè I non viene visitato prima di D nell'ordine simmetrico, non può essere suo figlio sinistro; è quindi figlio destro di D;

- poichè H è visitato prima di I nell'ordine simmetrico, H deve essere figlio sinistro di I;

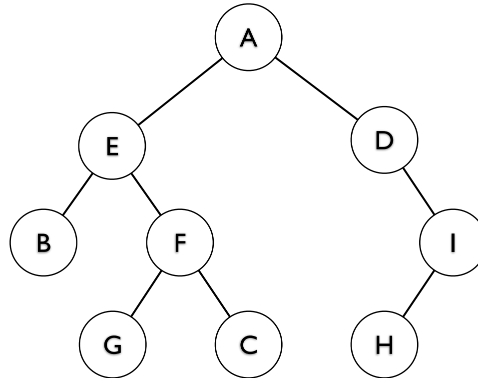

Figura 2: Albero risultante

2.3 Albero inverso (Esercizio 5.9 del libro)

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

```
inverti(TREE t)
if t = nil then
  return
t.left ↔ t.right
inverti(t.left)
inverti(t.right)
```

2.4 Aggiungi un figlio (Esercizio 5.10 del libro)

La procedura seguente risolve il problema in tempo $O(n)$. La chiamata iniziale è addChild$(t, 0)$.

```
addChild(TREE t, integer v)
if t = nil then
  return
v ← v + t.value
if t.left = t.right = nil then
  t.insertLeft(v)
else
  addChild(t.left, v)
  addChild(t.right, v)
```

2.5 Albero binario completo (Esercizio 5.12 del libro)

Si procede per induzione su k. Sia T_k un albero completo di altezza k; sia n_k il numero di nodi contenuti in T_k. Per $k = 0$, T_0 è formato da un solo nodo, quindi $n_0 = 2^{k+1} - 1 = 1$ è banalmente vero; inoltre, questo nodo è foglia, quindi il numero di foglie corrisponde a $2^0 = 1$.

È possibile notare che ogni foglia in T_{k-1} ha due figli foglia in T_k. Ragionando per induzione, sappiamo che il numero di nodi in T_k è pari a $2^k - 1$ nodi, di cui 2^{k-1} sono al livello $k-1$. Il numero di foglie in T_k è quindi pari a $2^{k-1} \cdot 2 = 2^k$; sommato ai nodi di T_{k-1} abbiamo $2^k - 1 + 2^k = 2^{k+1} - 1$, come volevasi dimostrare.

2.6 Visite iterative (Esercizio 5.13 del libro)

Questo è un esempio di pre-visita iterativa basata su stack.

```
previsita-iterativa(TREE t)
  precondition: t ≠ nil
  STACK S ← Stack()
  S.push(t)
  while not S.isEmpty() do
    TREE u ← S.pop()
    esame "per livelli" del nodo u
    if u.right ≠ nil then S.push(u.right)
    if u.left ≠ nil then S.push(u.left)
```

Nel caso della postvisita e della invisita, è necessario aggiungere un flag ai nodi; quando vengono inseriti per la prima volta, il flag viene settato a **false**; quando vengono estratti con flag a **false**, vengono re-inseriti con flag a **true**, avendo premura di intervallare i nodi figli e il nodo padre nel modo corretto. Quando un nodo viene estratto con flag a **true**, la visita viene eseguita effettivamente.

```
postvisita-iterativa(TREE t)
  precondition: t ≠ nil
  STACK S ← Stack()
  S.push(⟨t, false⟩)
  while not S.isEmpty() do
    ⟨u, f⟩ ← S.pop()
    if f then
      esame "per livelli" del nodo u
    else
      S.push(⟨u, true⟩)
      if u.right ≠ nil then S.push(⟨u.right, false⟩)
      if u.left ≠ nil then S.push(⟨u.left, false⟩)
```

```
invisita-iterativa(TREE t)
  precondition: t ≠ nil
  STACK S ← Stack()
  S.push(⟨t, false⟩)
  while not S.isEmpty() do
    ⟨u, f⟩ ← S.pop()
    if f then
      esame "per livelli" del nodo u
    else
      if u.right ≠ nil then S.push(⟨u.right, false⟩)
      S.push(⟨u, true⟩)
      if u.left ≠ nil then S.push(⟨u.left, false⟩)
```

2.7 Altezza minimale

L'altezza minimale del nodo radice di un albero è pari al minimo delle altezze minimali dei due sottoalberi, aumentata di 1. Quindi un semplice algoritmo ricorsivo per calcolare l'altezza minimale è il seguente:

```
integer altezza-minimale(TREE T)
  if T = nil then
    └ return +∞
  if T.left() = nil and T.right() = nil then
    └ return 0
  h_l ← altezza-minimale(T.left())
  h_r ← altezza-minimale(T.right())
  return min(h_l, h_r) + 1
```

L'algoritmo risultante viene eseguito in tempo $O(n)$, in quanto deve visitare tutti i nodi.

2.8 Alberi strutturalmente diversi

Ottieniamo una risposta per la domanda (1) risolvendo la domanda (2) e applicando poi la formula.
Per definire una ricorrenza, ci basiamo sulla seguente osservazione. Un albero di n nodi ha sicuramente una radice; i restanti $n-1$ nodi possono essere distribuiti nei sottoalberi destro e sinistro della radice. Ad esempio, un albero di 4 nodi può avere 3 nodi nel sottoalbero destro e 0 nel sinistro; oppure 2 nodi nel destro e 1 nel sinistro; oppure 1 nel destro e 2 nel sinistro; oppure 0 nel destro e 3 nel sinistro.
Detto quindi k il numero di nodi nell'albero sinistro, una formula ricorsiva per calcolare il numero di nodi $P(n)$ è la seguente:

$$P(n) = \sum_{k=0}^{n-1} P(k)P(n-1-k)$$

I casi base sono rappresentati da $P(0) = 1$ e $P(1) = 1$.
I valori per $n = 1, 2, 3, 4$ corrispondono a 1, 2, 5, 14.
Questa sequenza corrisponde all'n-esimo numero catalano (vedi programmazione dinamica), per il quale limiti asintotici inferiori sono:

$$\Omega(\frac{4^n}{n^{3/2}}) \text{ oppure } \Omega(2^n)$$

2.9 Alberi pieni

Il numero di alberi pieni strutturalmente diversi è calcolato nel modo seguente. Innanzitutto, è impossibile costruire un albero pieno se n è pari. Questo perché gli alberi pieni si ottengono partendo dalla radice e aggiungendo via via coppie di figli. Per i valori dispari, il numero di nodi pieni è ottenuto ricorsivamente partendo dal caso base $n = 1$ (che corrisponde ad un albero solo). Per $n > 1$, si crea una radice e poi si dividono i rimanenti $n-1$ figli fra il sottoalbero destro e il sottoalbero sinistro. Se i nodi vanno a destra, $1 \leq i \leq n-2$, $(n-1)-i$ nodi vanno a sinistra.

$$T[n] = \begin{cases} 0 & \text{se } n \text{ è pari} \\ 1 & \text{se } n = 1 \\ \sum_{i=1}^{n-2} T(i) \cdot T((n-1)-i) & \end{cases}$$

Bertossi, Montresor. Algoritmi e Strutture di Dati. ©2010 De Agostini Scuola

È possibile calcolare il valore con un semplice programma ricorsivo, ma questo avrebbe complessità superpolinomiale. Per velocizzare l'algoritmo, è opportuno utilizzare la programmazione dinamica.

```
integer alberipieni(integer n)
  if n è pari then
    return 0
  else
    integer[] D ← new integer[1...n]
    D[1] ← 1
    for integer i = 3 to n step 2 do
      D[i] ← 0
      for integer j = 1 to i − 2 step 2 do
        D[i] ← D[i] + D[j] · D[(n − 1) − j]
  return D[n]
```

La complessità di questo algoritmo è $O(n^2)$.

2.10 Altezza specificata

Ricordiamo che in un albero T , l'altezza di un nodo v è la massima distanza di v da una sua foglia. Vogliamo descrivere un algoritmo che calcoli l'altezza di v, per ogni nodo v dell'albero T. È chiaro che l'altezza di una foglia è 0; invece, l'altezza di un nodo interno v si calcola facilmente una volta note le altezze dei suoi figli: è sufficiente considerare la massima tra queste e incrementarla di 1.
Definiamo una procedura ricorsiva visita(v) che, su input v, restituisce l'altezza di v. Tale procedura restituisce il valore 0 se v è una foglia; altrimenti richiama se stessa sui figli di v determinando il valore massimo ottenuto, incrementa di 1 tale valore e lo assegna alla variabile h. Se inoltre h coincide con k si incrementa un opportuno contatore rappresentato da una variabile globale s (inizialmente posta a 0). Il valore finale di s sarà l'output dell'algoritmo.
Il costo della visita postordine $O(n)$.

```
integer countK(TREE v)
  s ← 0
  visita(v)
  return s
```

```
integer visita(TREE v)
  integer L ← iif(v.left = nil, −1, visita(v.left))
  integer R ← iif(v.right = nil, −1, visita(v.right))
  integer h ← max(L, R) + 1
  if h = k then
    s ← s + 1
  return h
```

Bertossi, Montresor. Algoritmi e Strutture di Dati. ©2010 De Agostini Scuola

2.11 Lunghezza di cammino

La soluzione è una semplice procedura ricorsiva di costo $O(n)$. La chiamata iniziale è lunghezzaCammino$(t, 0)$.

```
integer lunghezzaCammino(TREE t, integer v)
    if t = nil then
        return 0
    return v + lunghezzaCammino(t.left, v + 1) + lunghezzaCammino(t.right, v + 1)
```

Bertossi, Montresor. Algoritmi e Strutture di Dati. ©2010 De Agostini Scuola

3 Problemi aperti

3.1 Raggruppa le foglie con 0 e 1 (Esercizio 5.8 del libro)

Dato un albero binario le cui foglie contengono 0 od 1 e i cui nodi interni contengono solo 0, si vuole cambiare il contenuto delle foglie in modo che, visitandole da sinistra verso destra, si incontrino prima tutti gli 0 e poi tutti gli 1. Si scriva una procedura ricorsiva di complessità ottima.

3.2 Discendenti specificati

Dato un albero con radice t e un intero k, scrivere un algoritmo che restituisca il numero di nodi in t il cui numero di *discendenti* è pari a k.

3.3 Potatura

Dato un albero binario completo, con radice T, rappresentato con puntatori primo figlio/fratello, scrivere un algoritmo che "poti" l'albero, ovvero elimini tutti i nodi foglia.