ESERCIZI DI
ALGORITMI E STRUTTURE DATI 2
Parte 2

Livio Colussi
Dipartimento di Matematica Pura ed Applicata
Università di Padova

19 maggio 2006

Esercizio 18 Perché in un B-albero non possiamo avere grado minimo $t = 1$?

Soluzione. I nodi di un B-albero con grado minimo $t = 1$ contengono 0 o 1 chiavi ed hanno 1 o 2 figli. Un nodo contenente 1 chiave è pieno. Siccome, se l'albero non è vuoto, la radice contiene almeno una chiave essa è sempre piena. Di conseguenza ogni INSERT(T, x) spezza la radice in due nodi vuoti ed aggiunge una nuova radice. Dopo n inserimenti l'albero ha altezza $h = n$.

Esercizio 19 Calcolare il numero massimo di chiavi che può contenere un B-albero in funzione del suo grado minimo t e della sua altezza h.

Soluzione. Il numero massimo M di chiavi si ha quando tutti i nodi sono pieni:

$$M = (2t-1)\sum_{i=0}^{h}(2t)^i = (2t-1)\frac{(2t)^{h+1}-1}{2t-1} = (2t)^{h+1}-1$$

Esercizio 20 Dire quale struttura dati si ottiene se in ogni nodo nero di un albero rosso-nero conglobiamo i suoi eventuali figli rossi.

Soluzione. Siccome ogni nodo rosso ha padre nero tutti i nodi rossi vengono assorbiti dai rispettivi padri.

Ogni nodo nero assorbe 0, 1 o 2 nodi rossi e quindi alla fine avrà 1, 2 o 3 chiavi e, se non è foglia avrà 2, 3 o 4 figli.

Le foglie dell'albero ottenuto saranno tutte alla stessa altezza: l'altezza nera che avevano nell'albero rosso nero.

Otteniamo quindi un B-albero con grado minimo $t = 2$, ossia un 2-3-4-albero.

Esercizio 21 Scrivere una funzione che cerca la chiave minima contenuta in un B-albero ed una che data una chiave k cerca la chiave successiva, ossia la minima chiave $k' > k$ presente nel B-albero.

Soluzione. La chiave minima è la prima chiave della prima foglia del B-albero:

```
MINIMO(T)
    x ← root[T]
    if x = nil then return nil
    while not leaf[x] do
        x ← c1[x] ,DISKREAD(x)
    return key1[x]
```

La chiave successiva alla chiave k si trova con una versione modificata della funzione $Search$ che usa un diverso invariante per la ricerca binaria.

SEGUENTE(T, k)
  if $root[T] = nil$ **then return** nil
  return SEGUENTERIC$(root[T], k)$

SEGUENTERIC(x, k)
  $i \leftarrow 1\ ,\ j \leftarrow n[x] + 1$
  /* INVARIANTE: $key_{1..i-1}[x] \leq k < key_{j..n[x]}[x]$ */
  while $i < j$ **do**
    $m \leftarrow \lfloor (i + j)/2 \rfloor$
    if $key_m[x] \leq k$ **then** $i \leftarrow m + 1$ **else** $j \leftarrow m$
  /* $key_{1..i-1}[x] \leq k < key_{i..n[x]}[x]$ */
  if $i \leq n[x]$ **then** $ks \leftarrow key_i[x]$ **else** $ks \leftarrow nil$
  /* ks è la minima chiave maggiore di k nel nodo x */
  $ks' \leftarrow nil$
  if not $leaf[x]$ **then**
    DISKREAD$(c_i[x])$, $ks' \leftarrow$ SEGUENTERIC$(c_i[x], k)$
  /* ks' è la minima chiave maggiore di k nel sottoalbero di radice $c_i[x]$ */
  if $ks' \neq nil$ **then** $ks \leftarrow ks'$
  /* ks è la minima chiave maggiore di k nel sottoalbero di radice x */
  return ks

Esercizio 22 In un B-albero con grado minimo $t = 2$ vengono inserite le chiavi $1, 2, 3, \ldots, n$ nell'ordine. Valutare il numero m di nodi del B-albero risultante in funzione di n.

Soluzione. L'inserzione avviene sempre nell'ultima foglia a destra e quindi si percorre sempre l'ultimo ramo a destra del B-albero. Le eventuali operazioni SPLITCHILD() vengono eseguite quando un nodo contiene 3 chiavi. In tal caso una chiave viene spostata nel padre (che si trova sempre sul ramo di destra) e vengono creati du nodi figli con una sola chiave. A quello dei due figli che non si trova sul ramo di destra non verranno mai aggiunte altre chiavi. Pertanto i nodi che non

stanno sull'ultimo ramo hanno sempre una sola chiave mentre i nodi sull'ultimo ramo possono avere 1, 2 o 3 chiavi.

Indichiamo con c_i il numero di chiavi contenute nel nodo di altezza i sull'ultimo ramo.

Ad ognuna delle c_i chiavi ad altezza i contenute nel nodo di altezza i dell'ultimo ramo rimane associato un sottoalbero di altezza $i-1$ costituito da nodi che non appartengono all'ultimo ramo.

Tali sottoalberi hanno $2^i - 1$ nodi e altrettante chiavi. Il numero totale di chiavi è quindi:

$$n = \sum_{i=0}^{h}(c_i + 2^i - 1) = 2^{h+1} - 1 + \sum_{i=0}^{h}(c_i - 1)$$

mentre il numero totale di nodi è:

$$m = \sum_{i=0}^{h}(1 + 2^i - 1) = 2^{h+1} - 1$$

Questo fornisce i limiti:

$$n - 2h \leq m \leq n$$

Esercizio 23 Supponiamo che la dimensione di una pagina di disco si possa scegliere arbitrariamente e che il tempo di accesso sia $a + bt$ dove t è il grado minimo di un B-albero i cui nodi occupano una pagina della dimensione scelta ed a e b sono due costanti. Suggerire un valore ottimo di t nel caso $a = 30$ms e $b = 40\mu$s.

Soluzione. Basta calcolare il minimo della funzione:

$$f(t) = (a + bt)\log_t n = \ln n \frac{a + bt}{\ln t}$$

la cui derivata è:

$$f'(t) = \ln n \frac{bt(\ln t - 1) - a}{t \ln^2 t}$$

che si annulla per $t(\ln t - 1) = a/b$. Nel nostro caso $a/b = 30000/40 = 750$. Poiché

$$\begin{aligned} 179(\ln 179 - 1) &= 749.54 \\ 180(\ln 180 - 1) &= 754.73 \end{aligned}$$

la derivata si annulla in un punto t_0 compreso tra 179 e 180. Siccome la derivata è negativa per $t < t_0$ e positiva per $t > t_0$ il punto t_0 è un punto di minimo. Per t intero il punto di minimo è o 179 o 180. Siccome

$$\begin{aligned} f(179) &= 7163.53 \ln n \\ f(180) &= 7163.55 \ln n \end{aligned}$$

il punto di minimo è 179.

Esercizio 24 Mostrare come sia possibile aggiungere ad ogni nodo interno x di un B-albero i campi $size_i[x]$ che contengono il numero di chiavi presenti nei sottoalbero di radici $c_i[x]$. Dire quali sono le modifiche da apportare a INSERT(T, k) e DELETE(T, k). Assicurarsi che la complessità asintotica non aumenti.

Soluzione. Sia x un nodo interno e sia $y = c_i[x]$ figlio di x. Vale la seguente formula:

$$size_i[x] = \begin{cases} n[y] & \text{se } y \text{ è foglia} \\ n[y] + \sum_{j=1}^{n[y]+1} size_j[y] & \text{altrimenti} \end{cases}$$

INSERT(T, k) percorre un cammino dalla radice fino alla foglia in cui si deve inserire la nuova chiave. Prima di scendere da un nodo x al figlio $c_i[x]$ aumentiamo $size_i[x]$ di 1.

Quando eseguiamo una SPLITCHILD$(x, i, c_i[x])$ dobbiamo ricalcolare $size_i[x]$ e $size_{i+1}[x]$ usando la formula precedente.

DELETE(T, k) percorre un cammino dalla radice fino alla foglia da cui viene rimossa o la chiave data o la successiva. Prima di scendere da un nodo x al figlio $c_i[x]$ diminuiamo $size_i[x]$ di 1.

Quando eseguiamo una AUGMENTCHILD$(x, i, c_i[x])$ dobbiamo ricalcolare i campi $size_i[x]$ relativi ai figli che vengono modificati usando la formula precedente.

Esercizio 25 Mostrare come sia possibile aggiungere ad ogni nodo x di un B-albero un campo $height$ che contiene l'altezza del sottoalbero di radice x. Dire quali sono le modifiche da apportare a INSERT(T, x) e DELETE(T, x). Assicurarsi che la complessità asintotica non aumenti.

Soluzione. L'altezza di un nodo di un B-albero non può cambiare. È sufficiente quindi assegnare l'altezza corretta ai nodi quando essi vengono creati da una INSERT(T, x). In particolare:

1. quando si inserisce il primo nodo in un albero vuoto gli viene assegnata altezza 0;

2. quando si divide un nodo in due parti si assegna al nuovo nodo l'altezza del nodo iniziale;

3. quando si crea una nuova radice si assegna ad essa l'altezza della vecchia radice aumentata di 1.

Esercizio 26 Siano dati due B-alberi T' e T'' ed una chiave k tale che ogni chiave in T' sia minore di k ed ogni chiave in T'' sia maggiore di k. Scrivere un algoritmo che riunisce T', T'' e k in un'unico B-albero T (operazione JOIN(T', k, T'')). Se h' ed h'' sono le altezze rispettive di T' e T'' l'algoritmo deve avere complessità $O(1 + |h' - h''|)$.

Soluzione. Assumiamo che i B-alberi siano aumentati con il campo $height$ come visto nel precedente esercizio.

JOIN(r', k, r'')
/* r', r'' radici di T' e T'', alla fine r è la radice di T */
if $height[r'] = height[r'']$ **then**
if $n[r'] \geq t - 1$ **and** $n[r''] \geq t - 1$ **then**
"Crea un nuovo nodo r con la sola chiave k e figli r' ed r''."
DISKWRITE(r)
else
"Aggiungi alla fine di r' la chiave k, le chiavi di r'' e i puntatori ai discendenti di r''."
DISKWRITE(r')
$r \leftarrow r'$
else
/* $height[r'] \neq height[r'']$ */
if $height[r'] > height[r'']$ **then**
if $n[r'] < 2t - 1$ **then**
$r \leftarrow r'$
else
"Crea un nodo r vuoto ($n[r] = 0$) e con unico figlio r'."
SPLITCHILD$(r, 1, r')$
$x \leftarrow r\ ,\ y \leftarrow c_{n[x]+1}[x]$
DISKREAD(y)
while $height[y] > height[r'']$ **do**
if $n[y] = 2t - 1$ **then**
SPLITCHILD$(x, n[x] + 1, y),\ y \leftarrow c_{n[x]+1}[x]$
$x \leftarrow y\ ,\ y \leftarrow c_{n[x]+1}[x]$
DISKREAD(y)
if $n[y] + n[r''] + 1 < 2t$ **then**
"Aggiungi alla fine di y la chiave k, le chiavi di r'' e i puntatori ai discendenti di r''."
DISKWRITE(y)

else
if $n[r''] \geq t - 1$ **then**
"Aggiungi alla fine del nodo x la chiave k ed il figlio r''."
DISKWRITE(x)
else
"Aggiungi, all'inizio di r'' la chiave k, le ultime $t-2-n[r'']$ chiavi di y e gli ultimi $t-2-n[r'']$ puntatori ai discendenti di y."
"Aggiungi alla fine di x l'ultima chiave di y non spostata in r'' ed il puntatore ad r''."
DISKWRITE(r''), DISKWRITE(x), DISKWRITE(y)

else
/* $height[r'] < height[r'']$ */
if $n[r''] < 2t - 1$ **then**
$r \leftarrow r''$
else
"Crea un nodo r senza chiavi ($n[r] = 0$) e con unico figlio r''."
SPLITCHILD($r, 1, r''$)
$x \leftarrow r\ ,\ y \leftarrow c_1[x]$
DISKREAD(y)
while $height[y] > height[r']$ **do**
if $n[y] = 2t - 1$ **then**
SPLITCHILD($x, 1, y$), $y \leftarrow c_1[x]$
$x \leftarrow y\ ,\ y \leftarrow c_1[x]$
DISKREAD(y)
if $n[y] + n[r'] + 1 < 2t$ **then**
"Aggiungi all'inizio di y la chiave k, le chiavi di r' e i puntatori ai discendenti di r'."
DISKWRITE(y)
else
if $n[r'] \geq t - 1$ **then**
"Aggiungi all'inizio di x la chiave k ed il figlio r'."
DISKWRITE(x)
else
"Aggiungi, alla fine di r' la chiave k, le prime $t - 2 - n[r']$ chiavi di y e i primi $t - 2 - n[r']$ puntatori ai discendenti di y."
"Aggiungi all'inizio di x la prima chiave di y non spostata in r' ed il puntatore ad r'."
DISKWRITE(r'), DISKWRITE(x), DISKWRITE(y)

return r

Complessità. Se $h' = h''$ l'algoritmo richiede un tempo proporzionale a t. Siccome t è una costante l'algoritmo ha complessità $O(1) = O(1 + |h' - h''|)$.

Se $h' > h''$ l'algoritmo richiede un tempo proporzionale a t più un tempo proporzionale ad $(h' - h'')t$ per il ciclo while. Siccome t è una costante l'algoritmo ha complessità $O(1 + |h' - h''|)$. Il caso $h' < h''$ è simmetrico.

Esercizio 27 Siano dati un B-albero T ed una chiave k di T. Trovare un algoritmo che divide T in un B-albero T' contenente tutte le chiavi di T minori di k, un B-albero T'' contenente tutte le chiavi di T maggiori di k, e la chiave k (operazione SPLIT(T, k)). L'algoritmo deve avere complessità $O(h)$ in cui h è l'altezza di T.

Soluzione.

SPLIT(r, k)
/* r radice di T, alla fine ritorne le radici di T' e T'' */
"Con una ricerca una binaria trova l'indice i tale che $key_{1..i}[r] \leq k < key_{i+1}[r]$."
if $i > 0$ **and** $key_i[r] = k$ **then**
/* La chiave k si trove nella radice. */
if $i = 1$ **then**
/* La chiave k è la prima. */
$r' \leftarrow c_1[r]$
else
"Crea un nuovo nodo r'."
"Copia da r in r' le chiavi $key_1[r], \ldots, key_{i-1}[r]$ ed i puntatori $c_1[r], \ldots, c_i[r]$."
DISKWRITE(r')
if $i = n[r]$ **then**
/* La chiave k è l'ultima. */
$r'' \leftarrow c_{n[r]+1}[r]$
else
"Crea un nuovo nodo r''."
"Copia da r in r'' le chiavi $key_{i+1}[r], \ldots, key_{n[r]}[r]$ ed i puntatori $c_{i+1}[r], \ldots, c_{n[r]+1}[r]$."
DISKWRITE(r'')
else
/* La chiave k si trove nel sottoalbero $c_{i+1}[r]$. */
DISKREAD$(c_{i+1}[r])$

$x', x'' \leftarrow$ SPLIT$(c_{i+1}[r], k)$
if $i = 0$ **then**
/* $c_{i+1}[r]$ è il primo figlio. */
$r' \leftarrow x'$
else
"Crea un nuovo nodo y'."
"Copia da r in y' le chiavi $key_1[r], \ldots, key_{i-1}[r]$ ed i puntatori $c_1[r], \ldots, c_i[r]$."
DISKWRITE(y')
$r' \leftarrow$ JOIN$(y', key_i[r], x')$
if $i = n[r]$ **then**
/* $c_{i+1}[r]$ è l'ultimo figlio. */
$r'' \leftarrow x''$
else
"Crea un nuovo nodo y''."
"Copia da r in y'' le chiavi $key_{i+2}[r], \ldots, key_{n[r]}[r]$ ed i puntatori $c_{i+2}[r], \ldots, c_{n[r]+1}[r]$."
DISKWRITE(y'')
$r'' \leftarrow$ JOIN$(x'', key_{i+1}[r], y'')$
return r', r''

Complessità. Calcoliamo la complessità in funzione di $h = height[r]$. Per dimostrare che l'algoritmo richiede tempo $O(h) = O(\log n)$ calcoliamo separatamente il tempo $T_1(h)$ richiesto per eseguire le chiamate alla funzione $Join$ necessarie a costruire r', il tempo $T_2(h)$ richiesto per eseguire le chiamate alla funzione $Join$ necessarie a costruire r'' ed il tempo $T_3(h)$ richiesto per eseguire tutte le altre operazioni.

Il tempo richiesto nel caso pessimo per eseguire tutte le altre operazioni è dato dalla relazione di ricorrenza $T_3(h) = t + T_3(h-1)$ e quindi $T_3(h) = O(ht)$ che è $O(h)$ dato che t è una costante.

Il tempo richiesto nel caso pessimo per eseguire le $Join$ necessarie a costruire r' è il tempo necessario ad eseguire la JOIN$(y', key_i[r], x')$ più il tempo per eseguire le $Join$ necessarie a costruire x'.

Siccome x' è la radice del primo sottoalbero ritornato da SPLIT$(c_{i+1}[r], k)$ la sua altezza $h' = height[x']$ è sicuramente minore di h.

Il tempo necessario ad eseguire JOIN$(y', key_i[r], x')$ è $O(1 + |height[y'] - height[x']|) = O(h - h')$ (dato che $height[y'] = h - 1$). Dunque il tempo totale $T_1(h)$ per le $Join$ necessarie a costruire r' è dato dalla relazione di ricorrenza

$T_1(h) = O(h - h') + T_1(h')$ ed è quindi $O(h)$. Simmetricamente si dimostra che il tempo totale $T_2(h)$ per le $Join$ necessarie a costruire r'' è $O(h)$.

Quindi l'intero algoritmo richiede tempo $O(h)$.

Esercizio 28 Supponiamo che non esista una rappresentazione della chiave $-\infty$. Riscrivere DELETE(H, x) per un mucchio binomiale H in modo che essa non usi la chiave $-\infty$. Assicurarsi che la complessità rimanga $O(\log n)$.

Soluzione. La funzione DELETE(H, x) usa la funzione DECREASEKEY$(H, -\infty)$ per far diventare il nodo x una radice e quindi la funzione EXTRACTMIN(H) per rimuovere la radice ninima che a questo punto è proprio il nodo x. Occorre effettuare le stesse operazioni senza cambiare chiave ad x.

DELETE(H, x)
  $y \leftarrow parent[x]$
  while $y \neq nil$ **do**
    $k \leftarrow key[x]$, $key[y] \leftarrow key[x]$, $key[x] \leftarrow k$
    /* Scambia anche eventuali informazioni ausiliarie */
    $x \leftarrow y$, $y \leftarrow parent[x]$
  /* Ora x è una radice. La tolgo dalla lista delle radici. */
  if $x = cima[H]$ **then**
    $cima[H] \leftarrow sibling[x]$
  else
    $z \leftarrow cima[H]$
    while $sibling[z] \neq x$ **do** $z \leftarrow sibling[z]$
    $sibling[z] \leftarrow sibling[x]$
  /* Costruisco un heap $H1$ con i figli di x. */
  $cima[H1] \leftarrow nil$
  while $child[x] \neq nil$ **do**
    $y \leftarrow child[x]$, $child[x] \leftarrow sibling[y]$
    $parent[y] \leftarrow nil$, $sibling[y] \leftarrow cima[H1]$, $cima[H1] \leftarrow y$
  UNION$(H, H1)$

Esercizio 29 Nella funzione EXTRACTMIN(T) abbiamo dovuto percorrere tutta la lista dei figli del nodo estratto per invertirne l'ordine. Questo perché la lista delle radici è ordinata per grado crescente mentre le liste dei figli sono ordinate per grado decrescente. Cosa succede se ordiniamo le due liste in modo concorde?

Soluzione. Dobbiamo comunque percorrere la lista dei figli per porre a nil tutti i puntatori $parent$.