

Algoritmi e strutture dati

Alberi B (BT)



Menú di questa lezione

In questa lezione introduciamo un'altra specializzazione della struttura dati ad albero, chiamata alberi B, e studiamo le complessità delle operazioni ad essi associate.

Alberi B: introduzione

Un albero B (BT) generalizza un albero RB con fini differenti, ma mantenendo la sua proprietà fondamentale di bilanciamento. Anzi, un albero B è sempre completo. Nel modello più semplice e più diffuso di computazione, dobbiamo distinguere tra memoria **principale** e **secondaria** - come un disco. La memoria secondaria è in generale vari ordini di grandezza più capace, ma anche più lenta, della memoria primaria. I dischi a stato solido risolvono in parte questo problema, ad un alto costo, ma non possono essere considerare una soluzione definitiva per la maggioranza dei casi aziendali, con tecnologie più datate e con necessità di memorizzazione altissime. Gli alberi B sono una struttura dati ottimizzata per minimizzare gli accessi a disco; la complessità delle operazioni si dá lungo due direttive - tempo di CPU, come sempre, e numero di accessi a disco. La loro applicazione principale è nelle basi di dati: gli indici normalmente sono alberi B. Gli alberi B sono una struttura dinamica, basata sull'ordinamento, e memorizzata in maniera sparsa.

Un **albero B (BT)** si caratterizza per: possedere una **arietà** (che in questo contesto si conosce come **branching factor**) superiore a 2, spesso dell'ordine delle migliaia, un'altezza proporzionale a un logaritmo a base molto alta di n , dove n è il numero di chiavi, per avere nodi che contengono molte chiavi, tra loro ordinate, e per crescere verso l'alto, non verso il basso: un nodo comincia con essere la radice, e poi si converte in nodo interno generando una nuova radice. La complessità delle operazioni è proporzionale all'altezza: quindi i BT possono essere usati come gli RBT. Nella pratica questo non succede, per differenti ragioni: sono complessi da implementare, e i nodi hanno molte chiavi - trovare quella giusta è una operazione che si fa nella memoria principale, e non è distinto da navigare tra i nodi di un RBT.

Un nodo x in un BT si caratterizza per avere comunque il puntatore al nodo padre ($x.p$), ma gli altri dati sono diversi dai nodi degli alberi binari visti fino ad adesso. Infatti, abbiamo: il numero delle chiavi memorizzate nel nodo ($x.n$), l'informazione sull'essere, o meno, una foglia ($x.leaf$, che è a uno se e solo se x è foglia), i puntatori agli $n + 1$ figli di x ($x.c_1, \dots, x.c_{x.n+1}$), che sono indefiniti se x è foglia, e n chiavi ($x.key_1, \dots, x.key_{x.n}$), invece di una. Il sotto-albero puntato da $x.c_i$ è legato alle chiavi $x.key_{i-1}$ (se c'è) e $x.key_i$ (se c'è); il legame viene formalizzato nelle proprietà degli alberi B.

Alberi B: introduzione


In un nodo x il numero di chiavi, e quindi il branching factor, è vincolato da un parametro che si chiama **grado minimo**, si denota con t , ed è sempre maggiore o uguale a 2. Le proprietà di un albero B, che sostituiscono quelle di un albero red-black e generalizzano quelle di un albero binario di ricerca sono:

- 1 Ogni nodo, tranne la radice, ha almeno $t - 1$ chiavi;
- 2 Ogni nodo può contenere al massimo $2 \cdot t - 1$ chiavi;
- 3 Per ogni nodo x , $x.key_1 \leq x.key_2 \leq \dots \leq x.key_{x.n}$ *↔ Ordering Children*
- 4 Per ogni nodo x , se un nodo y è contenuto nel sotto-albero radicato in $x.c_i$, allora tutte le sue chiavi sono minori o uguali a $x.key_i$ (se c'è);
- 5 Per ogni nodo x , se un nodo y è contenuto nel sotto-albero radicato in $x.c_i$, allora tutte le sue chiavi sono maggiori di $x.key_{i-1}$ (se c'è).

Generalizzazione delle regole B+.

Alberi B: introduzione

Le prime due regole implicano che ogni nodo interno, tranne la radice, deve avere almeno t figli. Se l'albero non è vuoto, allora la radice ha almeno una chiave; inoltre, un nodo interno può avere fino a $2 \cdot t$ figli: in questo caso lo chiamiamo **nodo pieno**. Come conseguenza delle regole, l'altezza massima di un BT T con n chiavi e grado minimo $t \geq 2$ è:

$$h \leq \log_t \left(\frac{n+1}{2} \right)$$


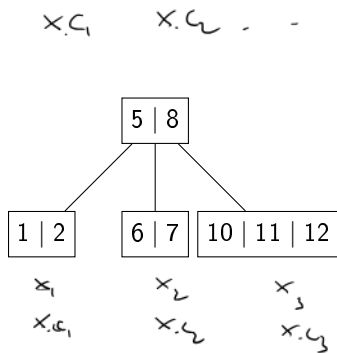
e tutte le foglie sono alla stessa altezza. Dunque un albero B è sempre un albero completo, sebbene bisogna generalizzare questa definizione rispetto a quella che abbiamo dato per gli alberi binari.

Alberi B: introduzione

Ecco un esempio di albero B:

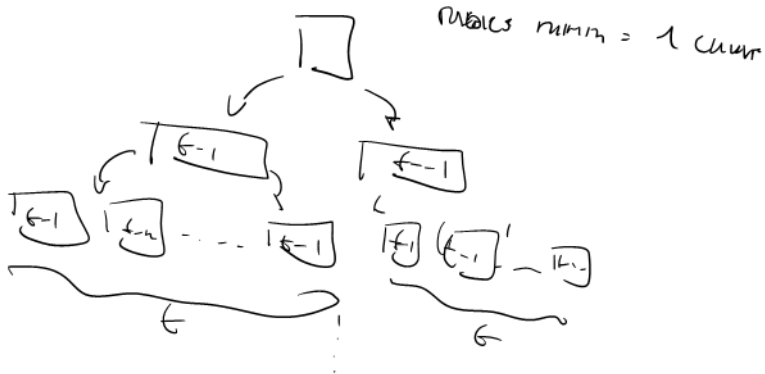
$radice(x) : n = 2, .c_1, .c_2, .c_3$

$x.c_1 : n = 2, x.leaf = True$
 $x.c_2 : n = 2, x.leaf = True$
 $x.c_3 : n = 3, x.leaf = True$



E' immediato osservare che per $t = 2$, si ottiene un albero che è sempre **isomorfo** a un albero red-black. Questi particolari alberi B sono noti come **alberi 2-3-4**.

CAD PSGLINE 125N CUCULINE WITHIN MIDDLE



Alberi B: caratteristiche e altezza

Per dimostrare la proprietà sull'altezza logaritmica, osserviamo che nel caso peggiore la radice di T contiene una sola chiave e tutti gli altri almeno $t - 1$. Perciò, T di altezza h ha almeno due nodi ad altezza 1, almeno $2 \cdot t$ ad altezza 2, almeno $2 \cdot t^2$ ad altezza 3, e così via, fino a $2 \cdot t^{h-1}$ ad altezza h . Perciò il numero di chiavi é:

$$1 + 2 \cdot (t - 1) + 2 \cdot t \cdot (t - 1) + 2 \cdot t^2 \cdot (t - 1) + \dots + 2 \cdot t^{h-1} (t - 1).$$

Abbiamo quindi la seguente disuguaglianza:

$n \geq 1 + (t - 1) \cdot \sum_{i=1}^h 2 \cdot t^{i-1}$	risultato precedente
$n = 1 + 2 \cdot (t - 1) \cdot \frac{t^h - 1}{t - 1}$	calcolo algebrico
$n = 2 \cdot t^h - 1$	calcolo algebrico, che implica
$t^h \leq \frac{n+1}{2}$	calcolo algebrico
$h \leq \log_t \left(\frac{n+1}{2} \right)$	proprietá dei logaritmi.

Alberi B: caratteristiche e altezza

Facciamo delle osservazioni sul modo che abbiamo usato per calcolare l'altezza massima di un albero in 4 casi diversi: nel caso delle **heap**, abbiamo usato la proprietà di essere alberi binari quasi completi, osservando che un certo sotto-albero deve contenere almeno tanti nodi interni; nel caso dei **BST**, abbiamo osservato che non ci sono limiti minimi al numero di figli di un nodo, potendo quindi l'albero degenerare in una lista; nel caso dei **RBT** abbiamo utilizzato l'altezza nera ed il fatto che questa è la stessa per tutti i rami: quindi, se guardassimo solo i nodi neri, l'albero sarebbe completo; infine, negli **alberi B**, abbiamo usato proprio il fatto che sono completi.

Descriviamo adesso le operazioni di ricerca, creazione, e inserimento di una chiave. Facciamo le seguenti convenzioni: usiamo esplicitamente le operazioni di *DiskRead* e *DiskWrite* per tenere conto degli accessi al disco; diciamo che la radice *T.root* è sempre in memoria principale; e rispetto a un nodo passato come parametro, assumiamo che sia già stata eseguita *DiskRead* su di esso. Nel caso degli alberi B, a differenza della altre strutture dati, calcoliamo le complessità **sia in termini di uso della CPU** (come sempre) **sia in termini di numero di accessi al disco**. In fatti questi due aspetti sono separati, e possiamo avere procedure che migliorano solo uno di essi, o entrambi.

L'operazione di *BTreeSearch* è la generalizzazione della ricerca su BST. Dobbiamo prendere una decisione tra molte possibilità per ogni nodo esplorato: invece di scegliere tra 2 figli, scegliamo tra $x.n + 1$ possibili figli. *BTreeSearch* prende in input un puntatore x ad un nodo di T , ed una chiave da cercare, e ritorna un puntatore ad un nodo y più un indice i (nel nodo) nei casi positivi, o **nil** se la chiave cercata non esiste nel sotto-albero radicato in x . La chiamata principale è *BTreeSearch*($T.root, k$).



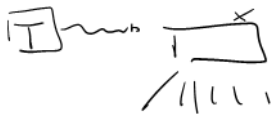
```
proc BTreeSearch( $x, k$ )  
{  
   $i = 1$   
  while (( $i \leq x.n$ ) and ( $k > x.key_i$ ))  $i = i + 1$   
  if (( $i \leq x.n$ ) and ( $k = x.key_i$ ))  
  then return ( $x, i$ )  
  if ( $x.leaf = True$ )  
  then return nil  
  DiskRead( $x.c_i$ )  
  return BTreeSearch( $x.c_i, k$ )  
}
```

Complessità di *BTreeSearch*

La **correttezza** di questa procedura è immediata. Per quanto riguarda la sua **complessità**, è facile vedere che si operano, al massimo, $O(h) = O(\log_t(n))$ accessi al disco. Inoltre, se utilizziamo la ricerca lineare (come nel codice) sul nodo, otteniamo che il tempo totale di CPU nel caso peggiore è $\Theta(t \cdot h) = \Theta(t \cdot \log_t(n))$. La forza dei BT non si apprezza nella notazione $O()$. Per vederla, bisogna tenere conto che $\log_t()$ può crescere **molto** più lentamente di $\log()$, portando, nei casi pratici, il numero di accessi al disco a non più di qualche unità.

Alberi B: inserimento

Prima di poter fare inserimento di una chiave, dobbiamo, diversamente dagli altri alberi che abbiamo visto, dobbiamo assicurare che l'albero esista (vuoto). A questo scopo, utilizziamo *BTreeCreate*, dove assumiamo di poter chiamare una procedura *Allocate()* che si occupa di creare ed occupare uno spazio sufficiente nella memoria secondaria. Assumiamo che questo prenda tempo costante. Questa fase di creazione (che prende tempo costante così come un numero di accessi a disco costante) mette in evidenza il fatto che un BT cresce a partire dalle foglie e non dalla radice: il primo nodo creato è infatti una foglia.



```
proc BTreeCreate (T)
```

```
{  
  x = Allocate()  
  x.leaf = True  
  x.n = 0  
  DiskWrite(x)  
  T.root = x  
}
```

Alberi B: inserimento



Adesso possiamo vedere l'inserimento, che é abbastanza complesso. In sintesi, cerchiamo di riempire un nodo fino a quando diventa pieno; quando il nodo è pieno, **dividiamo** il nodo in questione (che ha $2 \cdot t - 1$ chiavi) in due nodi di $t - 1$ chiavi ciascuno, ed inseriamo la nuova chiave nel nodo padre: se il padre diventa pieno in seguito a tale inserimento, ripetiamo l'operazione un livello più in alto. Quindi T cresce solamente quando la divisione ha luogo sulla radice: in questo caso si crea un nuovo nodo radice e si opera la divisione. Descriviamo quindi in primo luogo **BTreeSplitChild**.

Alberi B: inserimento

PIÙ DI UN
FIGLIO PIÙ

Atte
su x

Prima su
Memoria
Noi 8

proc BTreeSplitChild (x, i)

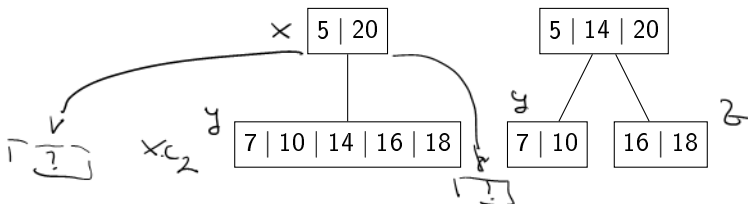
```
(  
  z = Allocate()  
  y = x.ci  
  z.leaf = y.leaf  
  for j = 1 to t - 1 z.keyj = y.keyj+t  
  if (y.leaf = False)  
    then for j = 1 to t z.cj = y.cj+t  
  y.n = t - 1  
  for j = x.n + 1 downto i + 1 x.cj+1 = x.cj  
  x.ci+1 = z  
  for j = x.n downto i x.keyj+1 = x.keyj  
  x.keyi = y.keyt  
  x.n = x.n + 1  
  DiskWrite(y)  
  DiskWrite(z)  
  DiskWrite(x)  
)
```



La procedura assume che x sia un nodo interno non pieno già nella memoria principale, e che il nodo figlio $x.c_i$, anche lui già nella memoria principale, sia pieno. Il nodo $x.c_i$ è diviso in due nodi, ognuno con la metà delle chiavi ($2 \cdot t - 1$ è sempre dispari!).

Alberi B: inserimento

5 | 10 (situazione
risorse non x)



Così, ad esempio, nell'albero in figura a sinistra, l'esecuzione di *BTreeSplitChild*($x, 2$) dove x è il nodo più in alto (per esempio, come passo necessario all'inserimento della chiave 6) provoca che il risultato sia quello a destra. Si osservi che nella figura a sinistra x ha in realtà tre figli (solo uno è visualizzato), e in quella a destra ne ha 4 (solo due sono visualizzati).

333nawo $\boxed{111}$:

$\boxed{k=2}$

Split + Ghad (~~8~~, 2)

1 2
x $\boxed{B|L}$

$\boxed{}$
 t_1

$\boxed{D|S|F|G|H}$
 \Downarrow


$\boxed{}$
 t_2

\boxed{B}

x $\boxed{B|F|L}$
 \swarrow \downarrow \searrow
 t_1 $\boxed{D|G}$ $\boxed{G|H}$ t_2

Il numero di operazioni CPU di *BTreeSplitChild* è $\Theta(t)$, mentre il costo in termini di operazioni su disco è $\Theta(1)$. Adesso possiamo vedere l'inserimento vero e proprio. La procedura *BTreeInsert* utilizza una seconda procedura (*BTreeInsertNonFull*) che si occupa, ricorsivamente, di inserire una chiave assumendo che il nodo considerato (ma non ancora scelto per l'inserimento) non sia pieno. Poiché la chiave nuova va sempre inserita in una foglia, se quello considerato è già una foglia, allora la chiave si inserisce semplicemente. Se invece non è una foglia, allora scendendo ricorsivamente da un nodo non pieno, potremmo trovarci su un nodo pieno; in questo caso, eseguiamo la divisione e procediamo ricorsivamente.

preinsert plus



Quindi la coppia *BTreeInsert* più *BTreeInsertNonFull* inserisce una nuova chiave, e si occupa che tutti i nodi pieni che si trovano sul percorso che dalla radice arriva alla foglia corretta non siano pieni. Questo corrisponde al 'fixup' che abbiamo visto nei RBT. Invece di inserire prima, e aggiustare dopo, sistemiamo la situazione di tutti i nodi coinvolti mentre troviamo la posizione corretta per l'inserimento. In questo modo, otteniamo una minimizzazione sia del numero di operazioni su disco sia del tempo di CPU.

Alberi B: inserimento

8.5.1.14
CMB
di nuovo
P.1.1.1.1

proc BTreeInsert (T, k)

```
{  
   $r = T.root$   
  if ( $r.n = 2 \cdot t - 1$ )  
  then  
  {  
     $s = Allocate()$   
     $T.root = s$   
     $s.leaf = False$   
     $s.n = 0$   
     $s.c_1 = r$   
    BTreeSplitChild( $s, 1$ )  
    BTreeInsertNonFull( $s, k$ )  
  }  
  else BTreeInsertNonFull( $r, k$ )  
}
```



Analizzando *BTreeInsert*, ci accorgiamo che tutto il codice meno una riga è relativo al caso in cui la radice sia piena. Questo caso è gestito allocando un nuovo nodo (la nuova radice), e poi usando *BTreeSplitChild* con indice 1 per assicurare che la radice non rimanga vuota e che si crei il secondo figlio. Il primo figlio è la radice originale. Questo è il momento in cui un albero B cresce di altezza. Quindi al chiamare *BTreeInsertNonFull*, lo si fa o sulla radice originale che non era piena, o su quella nuova, che per costruzione non è piena (anzi, ha una sola chiave).



```
proc BTreeInsertNonFull (x, k)
{
  i = x.n
  if (x.leaf = True)
  then
    while ((i ≥ 1) and (k < x.keyi))
    {
      x.keyi+1 = x.keyi
      i = i - 1
      x.keyi+1 = k
      x.n = x.n + 1
      DiskWrite(x)
    }
  else
    while ((i ≥ 1) and (k < x.keyi)) i = i - 1
    i = i + 1
    DiskRead(x.ci)
    if (x.ci.n = 2 · t - 1)
    then
      BTreeSplitChild(x, i)
      if (k > x.keyi)
      then i = i + 1
      BTreeInsertNonFull(x.ci, k)
    }
}
```


Quindi la condizione di **non essere nodo pieno** è certamente vera all'entrata di *BTreeInsertNonFull*. Se x è nodo foglia, e non è pieno, ci limitiamo a scegliere la posizione della nuova chiave e a muovere le altre chiavi per mantenere l'ordine (inserimento in una lista ordinata). I puntatori ai figli sono tutti **nil**, e così rimangono (anzi, non sono definiti: non si accede ad essi perché abbiamo la sentinella $x.leaf$). Altrimenti, si va a cercare la posizione corretta per la scelta del figlio di x dove proseguire la ricerca della foglia corretta. Il figlio corretto viene caricato in memoria principale, e due cose possono succedere: o non è pieno, ed in questo caso semplicemente si fa una chiamata ricorsiva, giacché sono rispettate le condizioni, oppure lo è: in questo caso, operiamo la divisione (possibile, perché x non è pieno), scegliamo tra i due nodi nuovi qual'è quello giusto per proseguire (entrambi non sono pieni), e facciamo la chiamata ricorsiva.

Correttezza e complessità di *BTreeInsert*

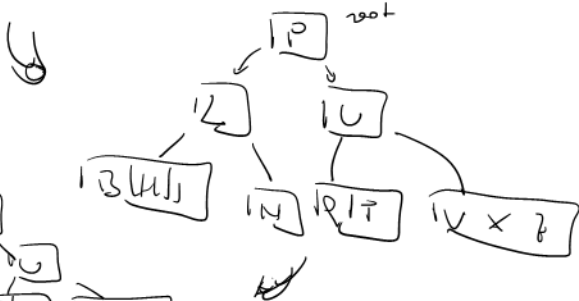
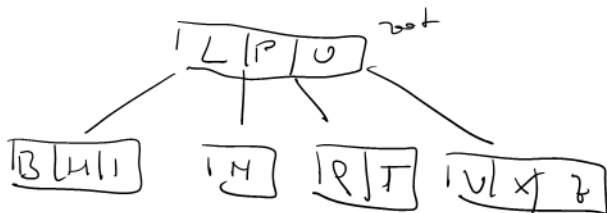
L'inserimento in BT è una procedura tail-ricorsiva (*BTreeInsertNonFull*). Quindi possiamo mostrare la **correttezza** usando una invariante. Scegliamo: **all'inizio di ogni esecuzione di *BTreeInsertNonFull* x è non pieno, e k va inserito nel sottoalbero radicato in x** . L'invariante è vera all'inizio (**caso base**), perchè la procedura viene chiamata sulla radice dell'albero. Supponiamo adesso che sia vera all'inizio di una certa esecuzione di *BTreeInsertNonFull* (**caso induttivo**). Poichè stiamo assumendo che ci sarà una prossima esecuzione, non siamo nel caso base della ricorsione, quindi entriamo nel ciclo **while**. Si trova il posto corretto per k , e poichè x non è una foglia, carica il giusto figlio. Poichè x non è pieno per ipotesi, se il nodo caricato fosse pieno potrebbe essere eseguito lo split, rendendolo non pieno. Questo è il nodo su cui poi verrà richiamato *BTreeInsertNonFull*, e quindi l'invariante è ancora vera. Nel caso in cui x fosse una foglia, l'invariante dice che non è piena e che k va inserito esattamente in x . La **complessità** in numero di operazioni su disco è chiaramente $\Theta(h)$; invece, quella in termini di operazioni CPU è $\Theta(h \cdot t) = \Theta(t \cdot \log_t(n))$.

Gli alberi sono una struttura fondamentale in innumerevoli applicazioni. Noi abbiamo visto quattro varianti: gli alberi (binari) non ordinati, che sono una struttura **grezza**, senza operazioni predeterminate, che si adatta caso per caso, e tre varianti di alberi semi-ordinati: BST, RBT, e BT. Ogni variante ha vantaggi e svantaggi, e deve essere scelta in base all'applicazione in questione. Ce ne sono molte altre, ma tutte sono guidate dagli stessi principi che governano queste tre che abbiamo visto.

Задано LD

$t = 2$

$Print(T, v)$



VA

