Algoritmi e strutture dati

Roberto Cordone

A. A. 2015-16

Capitolo 3

Implementazioni dei dizionari ordinati

Nota: queste dispense sono un rapido riassunto delle lezioni svolte nel dicembre 2015 e gennaio 2016. In buona parte ripetono gli argomenti delle dispense del prof. Goldwurm, con alcune aggiunte, qualche piccola modifica nella presentazione, qualche esempio. Chi notasse errori, incoerenze, oscurità o avesse dubbi è invitato a contattarmi per segnalarli.

3.4 B-alberi

- I B-alberi generalizzano sia gli alberi binari sia gli alberi 2-3:
  - gli alberi binari hanno una chiave e due sottoalberi, uno con elementi che precedono la chiave e uno con elementi che la seguono;
  - gli alberi 2-3 hanno due soglie (non chiavi) e tre sottoalberi, uno con elementi che precedono la prima soglia, uno con elementi compresi fra le due soglie e uno con elementi che seguono la seconda soglia;
  - i B-alberi hanno k chiavi ordinate e $k+1$ sottoalberi ordinati, e ogni chiave segue gli elementi di un sottoalbero e precede quelli del sottoalbero successivo;
- Topologicamente, un B-albero rispetta le seguenti proprietà, dove m è un parametro detto *indice di ramificazione*:
  1. tutte le foglie sono allo stesso livello;
  2. ogni nodo interno ha un numero di figli compreso fra m e $2m$;
  3. la radice ha un numero di figli compreso fra 2 e $2m$ (altrimenti, si complicherebbe la cancellazione);
  4. gli elementi del dizionario e le loro chiavi stanno nei nodi interni
  5. ogni nodo interno v contiene k_v elementi del dizionario ($A_v = \{a_{\pi_1}, \ldots, a_{\pi_{k_v}}\}$) e $k+1$ sottoalberi (T_0, T_1, …, T_{k_v});
  6. il numero k_v è diverso da nodo a nodo, ma sempre $\leq 2m-1$, perché i $k+1$ sottoalberi sono al massimo $2m$; vedremo che m definisce lo spazio disponibile per le chiavi e i sottoalberi, e questo ne determina il valore;
  7. gli elementi di A_v sono ordinati per chiavi crescenti e ogni elemento segue tutti quelli del sottoalbero di indice precedente (se non è vuoto) e precede tutti quelli del sottoalbero di indice uguale (se non è vuoto):

     $a_{\pi_i} \succ a$ per ogni $a \in A_u, u \in T_{i-1}(v)$ $a_{\pi_i} \prec a$ per ogni $a \in A_u, u \in T_i(v)$

In altre parole, gli elementi di T_0 precedono a_{π_1}, che precede gli elementi di T_1, e così via, terminando con gli elementi di T_{k_v}.

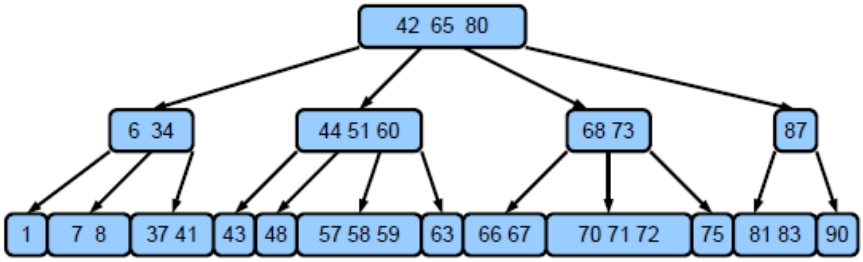

Figura 3.1: Esempio di B-albero: struttura topologica e proprietà di ordinamento

- Si può pensare un B-albero come un albero binario, nel quale alcuni livelli siano "collassati", cioè ogni sottoalbero costituito da nodi fra loro connessi in questi livelli viene compattato in un unico nodo: questo nodo risultante contiene più elementi e più sottoalberi, il numero dei sottoalberi è pari a quello degli elementi più uno e vale la proprietà di ordinamento sopra descritta.

FIGURA DA FARE

Figura 3.2: Un B-albero visto come albero binario collassato

- Lo scopo dei B-alberi è minimizzare il numero di operazioni di discesa e risalita lungo l'albero. Di conseguenza, il valore di m è tipicamente molto alto. Il motivo è che queste operazioni potrebbero essere molto più costose delle operazioni svolte sul singolo nodo.
- L'applicazione principale dei B-alberi è la gestione dei *file system*, nei quali un nodo corrisponde a un blocco di dati caricato dal disco fisso nella memoria RAM: operazioni sul blocco sono molto veloci, mentre scendere o salire lungo l'albero comporta di cambiare blocco, e quindi di accedere al disco, un'operazione molto più lenta.
- La scelta di m si basa sul fatto che un blocco deve contenere un nodo di un B-albero: la quantità di memoria disponibile per un blocco e lo spazio occupato da ogni singola chiave e puntatore a sottoalbero determinano il numero massimo di chiavi e sottoalberi contenuto in un nodo.
- Valutiamo l'altezza nel caso pessimo (dunque massima) di un B-albero che contiene n elementi.
  - l'altezza è massima quando la radice ha solo 2 figli e gli altri nodi interni solo m;
  - quindi, la radice contiene 1 chiave e gli altri nodi interni $m-1$;
  - il numero totale di chiavi contenute nell'albero è pari a 1 (chiave contenuta nella radice) più il numero di nodi in ciascun livello k moltiplicato per il numero di chiavi per nodo $(m-1)$; al livello k ci sono $2m^{k-1}$ nodi:

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

– la sommatoria geometrica si risolve facilmente

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

– essendo questo il numero di elementi minimo possibile, in generale

$$2m^h - 1 \geq n \Rightarrow h \geq \log_m \frac{n+1}{2}$$

Nota: sulle dispense compare un valore incrementato di 1 perché si suppone che le foglie non contengano chiavi, e quindi ci sia un livello in più.

- Una possibile implementazione di un B-albero con strutture a puntatori:

```
struct _elemento {
  int m;                        /* ordine */
  boolean leaf;                 /* indica se il nodo e' foglia */
  int kv;                       /* numero effettivo di chiavi */
  int Key[kv];                  /* vettore delle chiavi */
  A  *Av[kv];                   /* vettore degli elementi */
  struct elemento *Tv[kv+1];    /* vettore dei sottoalberi */
  struct elemento *father;      /* nodo padre */
};

typedef struct elemento *node;
typedef struct elemento *Btree;

Btree T;
```

dove un albero `T` è visto come un puntatore al nodo radice, e i nodi contengono vettori di puntatori agli elementi del dizionario `A`, ai sottoalberi `Tv` e vettori di chiavi `Key`, oltre al solito puntatore `father` verso il nodo padre. Il campo `leaf` serve solo per comodità, dato che basterebbe testare se il sottoalbero `Tv[0]` è vuoto.

- La funzione `MEMBER` o `SEARCH` è del tutto analoga a quelle degli altri alberi di ricerca, ma complicata dal fatto che nella radice vi sono più chiavi e che i sottoalberi sono molti.

```
A *SEARCH (A *a, Btree T)
{
  if (T == NULL) return NULL;
  k = KEY(a);
  r = ROOT(T);
  if (r->leaf == TRUE) return NULL;
  i = LOCATE(k,r->Key,kv);
  if (k == r->Key[i])
    return r->Av[i];
  else if (k <= r->M)
    return SEARCH(a,r->Tv[i]);
}
```

- Occorre una funzione ausiliaria `LOCATE(A *a, int Key[], int kv)`, che restituisce l'indice dell'elemento uguale a quello cercato oppure il sottoalbero in cui cercarlo. Siccome le chiavi sono ordinate, la funzione `LOCATE` può essere realizzata con una ricerca binaria sulle $k \leq 2m - 1$ chiavi, in tempo $O(\log_2 k) = O(\log_2 m)$. Se la chiave fa parte del vettore Key, la funzione ne restituisce l'indice `i`; se non esiste, restituisce l'indice `i` della massima chiave inferiore a quella cercata, che è anche l'indice del sottoalbero in cui l'elemento potrebbe trovarsi.

- Ovviamente, si può costruire facilmente una versione iterativa della funzione `SEARCH`, che sarà quella effettivamente usata.

- Esercizio: nel B-albero di Figura 3.1 si cerchino gli elementi 34, 58, 74.

- La complessità delle funzioni `MEMBER` e `SEARCH` è data dalla ripetizione nodo per nodo, al limite fino alle foglie, della funzione `LOCATE` e di un numero costante di operazioni elementari:

$$h \cdot \log_2 m = \log_m n \log_2 m = \log_2 n$$

- Le funzioni `INSERT` e `DELETE` si basano su tre principi di fondo, che è importante aver ben chiari per seguire il gioco dei casi che si possono presentare:

  1. inserimenti e cancellazioni si fanno sempre partendo dalle foglie, perché i nodi interni hanno una struttura "rigida", costituita da sottoalberi e chiavi rigorosamente ordinati e alternati fra loro, difficile da aggiornare;
     - cercheremo quindi sempre di aggiungere nuovi elementi nelle foglie, eventualmente facendo poi degli spostamenti a posteriori, solo se necessario;
     - dovendo cancellare una chiave interna, lo faremo cercando in una foglia una chiave che possa sostituirla, copieremo tale chiave nel nodo interno, sovrascrivendolo, e infine cancelleremo la foglia;
  2. quando un nodo ha più chiavi del dovuto ($> 2m - 1$) o meno del dovuto ($< m - 1$), si cerca di risolvere la questione "in famiglia", scambiando una chiave con un nodo fratello;
  3. se questo non è possibile, si procede a scindere il nodo sovraccarico o a fondere il nodo sottocarico con un nodo fratello, coinvolgendo il nodo padre; eventualmente, questo può rendere inammissibile il nodo padre, che va quindi trattato allo stesso modo.

- La funzione `INSERT` ha bisogno di conoscere la posizione di inserimento per il nuovo elemento. Ci pensa una funzione `SEARCH_NODE` che deve tener conto delle seguenti circostanze:

  - se l'elemento è già nell'albero, la funzione lo trova e restituisce `NULL`, a indicare che non occorre inserire il nuovo elemento;
  - la funzione si ferma nelle foglie, dato che gli inserimenti partono sempre dalle foglie;
  - invece di avere due o tre sottoalberi, ogni nodo v ne ha k_v e occorre la funzione `LOCATE` per capire con quale proseguire.

- Trovato il nodo `v`, il nuovo elemento andrebbe semplicemente aggiunto ad esso nella posizione giusta in base alla chiave:

  - se il nuovo elemento è uguale a uno dei vecchi, non occorre fare nulla;

- altrimenti, la funzione `ADD` inserisce il nuovo elemento e la nuova chiave nel nodo, scalando le successive in avanti di un passo;
- si noti che questo non crea problemi all'alternanza fra chiavi e sottoalberi perché il nodo `v` ha sottoalberi tutti vuoti;
- se le chiavi diventano $2m$, però, si supera il limite, e occorre una procedura `REDUCE` che rimetta a posto le cose.

```
Btree INSERT (A *a, Btree T)
{
  if (T == NULL)
    T = CREATE_BTREE(a);
  else
  {
    v = SEARCH_NODE(a,T);
    k = KEY(a);
    i = LOCATE(a,v->Key,v->kv);
    if (k == v->Key[i])
      return T;
    else
    {
      ADD(a,v->Key,v->kv);
      if (v->kv == 2 * v->m) REDUCE(p,T);
    }
  }
  return T;
}
```

- La funzione `REDUCE(node v, Btree T)` ripara l'albero `T` nel caso in cui il nodo `v` abbia $2m$ chiavi, sforando quindi il limite superiore:
  - se possibile, lo fa scaricando una chiave a un nodo fratello che abbia rigorosamente meno di $2m-1$ chiavi (se sono possibili entrambi, la scelta è convenzionale):
    - o scarica l'ultima chiave al fratello successivo;
    - o scarica la prima chiave al fratello precedente;
  - scaricare l'ultima chiave $a_{p_{2m-1}}$ al fratello successivo `u` richiede
    - di decrementare k_v;
    - di scalare in avanti le chiavi e i sottoalberi di `u` per lasciare spazio ai nuovi arrivati;
    - di cedere a `u` anche il sottoalbero T_{2m-1} di `v`; questo potrebbe trovare posto come T_0 di `u`, ma così facendo non si può mettere la chiave $a_{p_{2m-1}}$ espulsa da `v` come prima chiave a_{p_0} di `u`, dato che si invertirebbe l'ordine fra sottoalbero e chiave;
    - per risolvere il problema e mantenere corretto l'ordinamento di tutte le chiavi, basta mettere la chiave $a_{p_{2m-1}}$ espulsa da `v` nel nodo padre e spostare la chiave del padre come prima chiave a_{p_0} di `u`.
  - scaricare la prima chiave a_{p_0} di `v` al fratello precedente `u` richiede passaggi del tutto complementari:
    - aumentare k_u per lasciare spazio ai nuovi arrivati (qui non occorre scalare);

    - cedere a `u` anche il sottoalbero T_0 di `v`, che potrebbe trovare posto come ultimo sottoalbero T_{2k_u-1} di `u`, ma così facendo non si può mettere la chiave a_{p_o} espulsa da `v` come ultima chiave $a_{p_{k_u}}$ di `u`, dato che si invertirebbe l'ordine fra sottoalbero e chiave;
    - per risolvere il problema e mantenere corretto l'ordinamento di tutte le chiavi, basta mettere la chiave a_{p_0} espulsa da `v` nel nodo padre e spostare la chiave del padre come ultima chiave $a_{p_{k_u}}$ di `u`.
    - infine, occorre scalare all'indietro tutte le chiavi e i sottoalberi di `v`.
  - se non ci sono fratelli disponibili ad accettare la chiave e il sottoalbero in eccesso, occorre chiamare in causa il nodo padre:
    - spezzare il nodo sovraccarico in due, dividendo a metà chiavi e figli;
    - dal punto di vista del padre, questo crea due sottoalberi al posto di uno, e quindi nel padre bisogna fare posto per il nuovo sottoalbero scalando in avanti chiavi e sottoalberi;
    - inoltre, una volta che il padre ha due sottoalberi al posto di uno, occorre una nuova chiave intermedia fra loro: quindi, l'ultimo elemento del primo nodo viene estratto e usato come chiave nel nodo padre;
    - a questo punto, l'ordinamento è corretto, ma il padre ha una chiave e un sottoalbero in più; se le chiavi sono diventate $2m$, occorre riapplicare la funzione `REDUCE`
  - può capitare che non esista un padre, perché il nodo sovraccarico è la radice; in tal caso:
    - si alloca una nuova radice;
    - i due nodi risultanti dalla scissione diventano sottoalberi della nuova radice;
    - la chiave finale del primo dei due nodi diventa la prima e unica chiave della nuova radice.

- Esercizio: si consideri il B-albero della Figura 3.3, dove $m = 2$

FIGURA DA FARE, PARTENDO DALL'ALBERO DI FIGURA 3.1 E AGGIUNGENDO: alla quinta foglia (48) le chiavi 49 e 50, e alla settima (63) le chiavi 61 e 62

Figura 3.3: Inserimento di nuovi elementi in un B-albero

Si inserisca 82:

1. la posizione corretta per 56 è la penultima foglia;
2. si fa scalare in avanti la chiave 83 per far posto alla nuova (i sottoalberi sono tutti vuoti);
3. si aggiunge 82.

- Esercizio: sul B-albero risultante, si inserisca 56:

1. la posizione corretta per 56 è la sesta foglia
2. si fanno scalare in avanti tutte le chiavi (57, 58 e 59) per far posto alla nuova (i sottoalberi sono tutti vuoti);
3. si aggiunge 56
4. ora il nodo è sovraccarico;

5. il fratello precedente e quello successivo sono al limite massimo: non possono accettare nuove chiavi;
6. si spezza il nodo $((56, 57, 58, 59))$ in due: (56) e $(58, 59)$, che diventeranno sottoalberi del nodo padre;
7. la chiave intermedia 57 va aggiunta al nodo padre, racchiusa fra i due sottoalberi;
8. bisogna scalare la chiave 60 e il sottoalbero finale avanti di un passo per far posto alla chiave 57 e ai due nuovi sottoalberi;
9. ora il nodo padre è sovraccarico: $(44, 51, 57, 60)$;
10. il fratello successivo ha posto libero;
11. si fa posto scalando in avanti 68, 73 e tutti e tre i sottoalberi;
12. si attacca il sottoalbero finale $(61, 62, 63)$ come albero iniziale del fratello;
13. si mette la chiave 60 nel padre al posto di 65;
14. si mette la chiave 65 nel fratello come chiave iniziale, fra i sottoalberi $(61, 62, 63)$ e $(66, 67)$

- Esercizio: sul B-albero risultante, si inserisca 64:
  1. la posizione corretta è la settima foglia: $(61, 62, 63, 64)$;
  2. il nodo è sovraccarico e non ci sono fratelli disponibili;
  3. si spezza il nodo in (61) e $(63, 64)$ accodando la chiave intermedia 62 a quelle del padre: $(44, 51, 57, 62)$;
  4. ora il padre è sovraccarico e il fratello successivo è al limite;
  5. ...
- Esercizio: si inserisca 52
  1. propagazione fino alla radice
- La funzione `DELETE` applica la funzione `SEARCH` per trovare l'elemento.
  - Se non lo trova, non fa nulla.
  - Se lo trova ed è l'unico nodo, l'albero diventa vuoto.
  - Se lo trova, ed è in una foglia lo cancella, scalando all'indietro le chiavi seguenti.
  - Se è in un nodo interno, cerca l'elemento massimo precedente, applicando la funzione `MAX` (non presentata, ma banale da realizzare: basta scendere sempre lungo il sottoalbero di indice massimo T_{k_v} finché non si arriva a una foglia, e allora prendere la chiave massima. Trovato l'elemento massimo, si copia al posto dell'elemento da cancellare e si cancella la foglia, riducendo k_v.
  - Ora potrebbe esserci un nodo con $k_v < m - 1$. La funzione `EXPAND` deve ristabilire la struttura corretta.
- La funzione `EXPAND`
- Esercizio: si consideri il B-albero della Figura 3.3, dove $m = 2$. Si elimini 20:
  1. si cerca 20 e non lo si trova: niente da fare
- Esercizio: dal B-albero risultante si elimini 61:

1. si cerca 61 e lo si trova nella settima foglia;
2. si riscalano le chiavi successive (non ci sono sottoalberi, trattandosi di una foglia) sovrascrivenbdo 61 in modo da cancellarlo.

- Esercizio: dal B-albero risultante si elimini 87:

  1. si cerca 87 e lo si trova nell'ultimo nodo del livello $k = 1$;
  2. non essendo una foglia, occorre cercare il massimo elemento precedente, applicando la funzione `MAX` al sottoalbero immediatamente precedente (T_0): si trova 83;
  3. si sovrascrive la chiave 87 e l'elemento con 83 e l'elemento corrispondente;
  4. si cancella 83 nella foglia (senza scalare nulla, dato che è l'ultima chiave: basta ridurre k_v).

- Esercizio: dal B-albero risultante si elimini 1:

  1. si cerca 1 e lo si trova nella prima foglia;
  2. si cancella 1 (non ci sono altre chiavi da scalare);
  3. la foglia è troppo scarica (vuota: $0 < m - 1$); cerca fratelli che possano cedere chiavi;
  4. il fratello successivo può cedere la prima: 7 sale nel padre a sostituire 6, che scende nella prima foglia al posto della chiave cancellata;
  5. nella seconda foglia, le chiavi (8) scalano a coprire il buco rimasto.

- Esercizio: dal B-albero risultante si elimini 81:

  1. si cerca 81 e lo si trova nell'undicesima foglia;
  2. si cancella 81 (non ci sono altre chiavi da scalare);
  3. la foglia è troppo scarica (vuota: $0 < m - 1$); cerca fratelli che possano cedere chiavi;
  4. non ci sono fratelli adatti; si passa al padre
  5. il nodo scarico e il fratello successivo (o il precedente, a scelta: qui c'è solo quello successivo) si fondono, unendo i due sottoalberi in uno solo; la chiave intermedia scende nel nuovo nodo complessivo: si ottiene una foglia $(83, 90)$ e un padre vuoto;
  6. siccome il padre è scarico, si ripete la funzione `EXPAND`: si può prendere un nodo dal fratello precedente;
  7. si fa scendere la chiave 80 dal padre al nodo scarico, salire l'ultima chiave 73 dal fratello al padre e si sposta il sottoalbero finale del fratello (75) come primo.

- La complessità delle funzioni `INSERT` e `DELETE` è determinata da

  – operazioni compiute su ciascun nodo: si tratta al massimo di scalare tutte le chiavi e i sottoalberi del nodo, dunque $O(m)$ operazioni;

  – numero di nodi elaborati: si tratta al massimo di risalire l'intero albero, elaborando $O(h)$ nodi.

  Di conseguenza, le operazioni complessivamente sono:

$$O(m \cdot h) = O(m \log_m n) = O\left(\frac{m}{\log_2 m} \log_2 n\right)$$

quindi ancora logaritmiche nella dimensione del dizionario. Rispetto alla funzione `SEARCH`, compare il termine $m/\log_2 m$, che è costante, dato che m è una costante. Però, dovendo fissare il valore di m, questo termine suggerisce che sia meglio non esagerare, dato che può diventare controproducente: l'albero diventa più piatto, ma si rischia di perdere molto tempo nell'aggiornare i singoli nodi, scalando chiavi e sottoalberi avanti e indietro.