

Hashing

Dott. Ezio Bartocci, Dott. Francesco De Angelis

Laboratorio di Algoritmi e strutture Dati - AA 2006/2007

Dip. di Matematica e Informatica

Università di Camerino

ezio.bartocci@unicam.it, francesco.deangelis@unicam.it

28 dicembre 2007

- Gli algoritmi visti fino ad ora sono basati sull'operazione astratta di confronto
- Fa eccezione la ricerca indicizzata da chiavi che usa i valori delle chiavi con indici di array per ottenere accessi immediati
- Vediamo ora i metodi di **hashing** come estensione della ricerca indicizzata
- Il primo passo è la definizione di funzioni di hash che data una chiave generino l'indirizzo per una posizione nella tabella
- Il secondo passo è stabilire un metodo per la risoluzione delle *collisioni* (la situazione in cui la funzione hash applicata a chiavi diverse genera lo stesso indirizzo)

- L'hashing è un buon esempio di compromesso tra prestazioni in termini di spazio e tempo
 - senza limitazioni di memoria sarebbe possibile effettuare qualsiasi ricerca con un solo accesso impiegando la chiave come indirizzo (come nella ricerca indicizzata)
 - senza limitazioni temporali si potrebbe sfruttare una quantità limitata di memoria ed accederla sequenzialmente
- Con ipotesi favorevoli, l'hashing impiega tempo costante per inserimento e ricerca indipendentemente dalla dimensione della tabella
- L'hashing non è una panacea, il tempo di calcolo dipende comunque dalla lunghezza delle chiavi. Inoltre, non consente operazioni efficienti per la selezione e l'ordinamento

Funzioni di hash

- Il primo problema è la definizione di funzioni di hash con il compito di trasformare le chiavi di ricerca in indirizzi
- Data una tabella di M elementi è necessaria una funzione che trasformi la chiave in un valore nell'intervallo $[0, M - 1]$
- I possibili valori in output dovrebbero essere “equiprobabili” cioè bene distribuiti nell'intervallo
- Se le chiavi sono maggiori di s e minori di t , per s e t fissati, per ottenere valori da usare come indirizzi possiamo sottrarre s e dividere per $t - s$ moltiplicando poi per M . Otteniamo così indirizzi per la tabella

Hashing modulare

- Un altro metodo semplice ed efficiente è quello di scegliere come dimensione M un numero primo e, per ogni chiave intera k , calcolare il resto della divisione di k per M

$$h(k) = k \bmod M$$

- L'hashing modulare si può applicare quando abbiamo accesso ai bit delle chiavi
- Scegliamo un numero primo perchè si hanno strani effetti quando il numero scelto è una potenza di 2. Si rischia di trascurare bit della chiave durante la generazione.
- L'hashing modulare è semplice da implementare ma occorre rendere la tabella delle dimensioni di un numero primo.
 - Può bastare un numero piccolo, oppure si cerca in una lista il numero che più si avvicina alla dimensione voluta. Ad esempio un numero nella forma $2^t - 1$ per t primo è un *numero primo di Mersenne*

- In alcune situazioni si hanno chiavi lunghe che sono difficili da rappresentare, in questo caso è possibile applicare l'hashing modulare trasformando le chiavi una porzione alla volta
- La possibilità di far in modo che chiavi reali assomiglino a chiavi casuali ci porta a considerare algoritmi di hashing randomizzati cioè a funzioni che producano indici casuali della tabella, indipendentemente dalle chiavi.
- La randomizzazione non è difficile da ottenere: ciò che vogliamo è un metodo che tenga conto di tutti i bit delle chiavi per calcolare un intero minore di M (la dimensione della tabella)

hash per chiavi stringa

```
static int hash(String s, int M)
{
    int h = 0, a = 127;
    for (int i = 0; i < s.length(); i++)
        h = (a*h + s.charAt(i)) % M;
    return h;
}
```

- Il metodo `hash` usa come base un numero primo invece della potenza di una potenza di due che viene usualmente richiesta quando si cerca l'intero corrispondente alla rappresentazione ASCII della stringa

info

Questa rappresentazione è posizionale:

$$\text{now} = 110 * 128^2 + 111 * 128^1 + 119 * 128^0$$

dove 110, 111, e 119 rappresentano le lettere `n`, `o`, `w` secondo la codifica ASCII

Hashing Universale

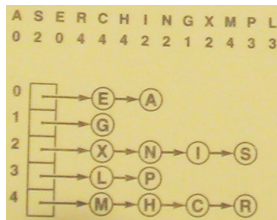
- Possiamo scegliere il valore del moltiplicatore a caso in modo da ottenere un algoritmo randomizzato
- Possiamo usare valori casuali per i coefficienti e per ogni cifra della chiave, ottenendo l'*hashing universale*
- Una funzione di hashing universale fa sì che la probabilità di collisioni tra chiavi distinte sia precisamente pari a $1/M$
- Può però rivelarsi più lento poiché esegue due operazioni aritmetiche per ogni carattere della chiave

metodo di hashing universale

```
static int hashU(String s, int M)
{
    int h = 0, a = 31415, b = 27183;
    for (int i = 0; i < s.length(); i++)
    {
        h = (a*h + s.charAt(i)) % M;
        a = a*b % (M-1);
    }
    return h;
}
```

Concatenazioni separate

- La seconda componente di un algoritmo di hash è come gestire il caso di mappatura di due chiavi su uno stesso indirizzo
- Il metodo immediato è quello di costruire una lista concatenata per ciascun indirizzo della tabella contenente le chiavi assegnate a quell'indirizzo
- Gestiamo M liste separate e chiamiamo il metodo *concatenazioni separate*
- Possiamo scegliere di mantenere le liste ordinate o no



- Per una search miss possiamo assumere che la funzione di hash sparpagli i valori delle chiavi in modo abbastanza uniforme in modo da uniformare la quantità di accessi per lista

Concatenazioni separate

Proprietà

Il metodo delle concatenazioni separate riduce il numero di confronti di una ricerca sequenziale mediamente di un fattore M , utilizzando uno spazio aggiuntivo necessario alla memorizzazione di M puntatori

- Una buona approssimazione della lunghezza *media* di una lista è data da N/M con N numero delle chiavi e considerando ciascuno degli M valori di hash “equiprobabile”
- Usare liste non ordinate per implementare concatenazioni esterne è agevole ed efficiente perchè l’inserimento impiega tempo costante mentre la ricerca ha tempo proporzionale a N/M

Proprietà

In una tabella hash con M liste concatenate separate ed N chiavi. la probabilità che il numero di chiavi in ciascuna lista sia prossimo a N/M (a meno di una piccola costante moltiplicativa) è molto vicino a 1

Concatenazioni separate

- Possiamo usare le concatenazioni separate con fiducia purchè la funzione hash generi valori approssimativamente casuali
- Nelle implementazioni conviene scegliere un valore di M piccolo in modo da non sprecare array di memoria con link nulli. D'altra parte abbiamo anche l'esigenza di rendere le liste corte in modo da rendere la ricerca sequenziale efficiente
- Una buona regola empirica è M pari a $1/5$ o $1/10$ del numero di chiavi

Concatenazioni separate

- Nei casi in cui la memoria non sia una risorsa critica scegliendo M grande si ottiene un tempo di ricerca costante
- Viceversa, quando la memoria è critica possiamo ottenere un fattore M di miglioramento prestazioni scegliendo per M il massimo valore consentito
- Nella pratica usiamo liste non ordinate
 - inserimento estremamente rapido
 - funzionamento da stack (quindi possiamo rimuovere gli elementi usati più di recente in maniera efficiente)
 - Quest'ultima caratteristica è particolarmente interessante per applicazioni particolari come la costruzione di compilatori
- L'hashing non è adatto in applicazioni con operazioni ulteriori come la selezione e l'ordinamento
- Sono usati invece in applicazioni con un gran numero di operazioni di ricerca, inserimento, e cancellazione

- Quando è possibile stimare il numero di elementi nella tabella e si ha una quantità di memoria contigua sufficiente a contenere tutte le chiavi più un po' di spazio aggiuntivo può non essere conveniente usare puntatori a liste concatenate
- Si usa lo spazio libero per gestire collisioni ottenendo l'hashing *ad indirizzamento aperto*
- Il più semplice metodo prende il nome di *scansione lineare*

Tabella di simboli con scansione lineare

```
private ITEM[] st;
private int N, M;
ST(int maxN)
{ N = 0; M = 2*maxN; st = new ITEM[M]; }
void insert(ITEM x)
{ int i = hash(x.key(), M);
  while (st[i] != null) i = (i+1) % M;
  st[i] = x; N++;
}
ITEM search(KEY key)
{ int i = hash(key, M);
  while (st[i] != null)
    if (equals(key, st[i].key()))
      return st[i];
  else i = (i+1) % M;
  return null;
}
```

Scansione lineare

- Quando si verifica una collisione è sufficiente sondare la posizione successiva della tabella. Tre possibilità:
 - le chiavi coincidono e si ha una search hit
 - se la posizione non è occupata da nessun elemento si ha una search miss
 - altrimenti si sonda la posizione successiva proseguendo fino alla fine (e continuando dall'inizio della tabella)
- se dopo una search miss si vuole inserire il record in question basta mettere l'elemento nella posizione libera che mette fine alla ricerca

A	S	E	R	C	H	I	N	G	X	M	P	
7	3	9	9	8	4	11	7	10	12	0	8	
							(A)					
	(S)						A					
	S						A	(E)				
	S						A	E	(R)			
	S						A	(C)	E	R		
	S	(H)					A	C	E	R		
	S	H					A	C	E	R	(I)	
	S	H					A	C	E	R	(N)	
(G)	S	H					A	C	E	R	I	N
G	(X)	S	H				A	C	E	R	I	N
G	X	(M)	S	H			A	C	E	R	I	N
G	X	M	S	H	(P)		A	C	E	R	I	N
0	1	2	3	4	5	6	7	8	9	10	11	12

Scansione lineare

- Come per le concatenazioni separate, le prestazioni dipendono dal rapporto $\alpha = N/M$.
- Per le concatenazioni separate indicava il numero medio di elementi per lista
- Per l'ind. aperto è il *fattore di carico* cioè la frazione delle posizioni occupate nella tabella (vale al più 1)
 - Per α piccolo bastano pochi sondaggi per le ricerche
 - Per α prossimo a 1 potrebbero essere necessari molti sondaggio addirittura un ciclo infinito se la tabella è piena
- Usiamo spazi liberi nella tabella per ridurre la lunghezza delle scansioni
- Il costo medio della scansione dipende dai raggruppamenti (cluster) di elementi all'interno della tabella

Consideriamo due casi limite per una tabella piena a metà ($M = 2N$):

- le posizioni occupate e libere sono alternate (ad esempio sono liberi gli indici pari, abbiamo $1 + 1/2$ tentativi per una search miss)
- la metà degli elementi occupati è contigua (ad esempio partendo dalla metà della tabella), abbiamo $1 + N/4$ tentativi per una search miss
- In media il numero di sondaggi è proporzionale al quadrato delle lunghezze dei cluster
- Nonostante la forma relativamente semplice dei risultati l'analisi di questo algoritmo è un problema complesso. La soluzione di **Knuth** nel 1962 è una pietra miliare nei metodi di analisi algoritmica

Proprietà

Quando le collisioni sono risolte tramite scansione lineare, il numero medio di sondaggi richiesti da una ricerca all'interno di una tabella di dimensione M contenente $N = \alpha M$ chiavi è pari circa a:

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

per una search hit, e a

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

per una search miss.

Una search miss è sempre più costosa di una search hit. Entrambe richiedono però in media solo alcuni sondaggi su tabelle piene a metà.

Scansione lineare

- Lasciamo al client la gestione dei duplicati
- Il processo di costruzione della tabella inserisce le chiavi in ordine casuale. Ordinamento e selezione richiedono uno degli algoritmi già visti.
- La scansione lineare non è adeguata per applicazioni che fanno intenso uso di queste operazioni
- Come rimuoviamo un elemento? Non è possibile semplicemente cancellarlo poichè le seguenti ricerche potrebbero arrestarsi prematuramente
- Una soluzione è di ricalcolare l'hashing per gli elementi che si trovano tra l'elemento cancellato e la prima posizione libera
- Calcolo di pochi valori se la tabella è sparsa

- La scansione lineare funziona in considerazione del fatto che si controllano tutte le chiavi che la funziona hash trasforma nello stesso indirizzo nel quale è stata trasformata la chiave di ricerca
- Chiaramente vengono confrontate anche alle chiavi e la cosa peggiore è che l'inserimento di una chiave con un dato valore hash puà far incrementare il tempo di ricerca a causa della creazione dei cluster
- L'**hashing doppio** elimina virtualmente il problema.
- Invece di esaminare gli elementi che seguono la posizione in cui si ha la collisione si impiega una seconda funzione di hash per ottenere un incremento fisso da usare nella sequenza dei sondaggi

Tale funziona va scelta con cura:

- deve essere evitata la situazione in cui la seconda funzione restituisca 0 (creerebbe cicli infiniti!)
- occorre che il valore generato e la dimensione della tabella siano numeri primi tra loro per evitare di generare sequenze troppo corte
- ciò è facilmente ottenibile scegliendo M primo e scegliendo una funzione che restituisca sempre valori minori di M
- Ogni perdita di efficienza dovuta a questa semplificazione è irrilevante. Se abbiamo tabelle molto grandi e sparse la loro dimensione può non essere un numero primo poichè saranno sufficienti solo alcuni sondaggi in ogni ricerca

Hashing Doppio

- L'hashing doppio è equivalente ad un algoritmo di *hashing randomizzato* in cui si usano sequenze di sondaggi dipendenti dalle chiavi e dove ciascun sondaggio ha pari probabilità
- L'hashing doppio è facile da implementare, mentre l'hashing randomizzato è facile da analizzare.
- Le prestazioni dei due metodi sono simili per tabelle sparse anche se per l'hashing doppio possiamo gestire convenientemente anche tabelle con fattori di carico elevati.
- L'hashing doppio consente di, a parità di tempi medi di ricerca, usare tabelle più piccole di quelle della scansione lineare

Proprietà

Possiamo garantire che il costo medio delle ricerche sia minore di t sondaggi se manteniamo il fattore di carico minore di $1 - 1/\sqrt{t}$ per la scansione lineare, e minore di $1 - 1/t$ per l'hashing doppio

- Il metodo di cancellazione visto per la scansione lineare non funziona con l'hashing doppio.
- Dobbiamo in questo caso usare una sentinella che segna la posizione come occupata ma senza corrispondere ad alcuna chiave
- Analogamente alla scansione lineare l'hashing doppio non rappresenta una base adeguata per le operazioni di selezione e ordinamento

Ricerca e inserimento per l'hashing doppio

```
void insert(ITEM x) {
    KEY key = x.key();
    int i = hash(key, M);
    int k = hashtwo(key);
    while (st[i] != null) i = (i+k) % M;
    st[i] = x; N++;
}

ITEM search(KEY key) {
    int i = hash(key, M);
    int k = hashtwo(key);
    while (st[i] != null)
        if (equals(key, st[i].key()))
            return st[i];
    else
        i = (i+k) % M;
    return null;
}
```

Tabelle hash dinamiche

- Quando il numero di chiavi della tabella cresce le prestazioni tendono a peggiorare
- Nelle concatenazioni separate il tempo di ricerca cresce gradualmente, così come per l'indirizzamento aperto, la scansione lineare, e l'hashing doppio su tabelle sparse (anche se qui il costo cresce più rapidamente)
- Tale situazione è in netto contrasto con l'uso di alberi che gestiscono in maniera naturale la crescita della tabella
- Un modo di far crescere la tabella hash è quello di raddoppiarne la dimensione quando la tabella inizia a riempirsi sopra una certa soglia
- L'operazione di raddoppio è costosa poiché tutti gli elementi devono essere reinseriti, ma si tratta di un'operazione poco frequente

Tabelle hash dinamiche

- Ogni volta che la tabella si riempie almeno per metàne raddoppiamo la dimensione
- Dopo la prima espansione la tabella avrà un fatto redi carico compreso tra $1/4$ e $1/2$ quindi il costo medio della ricerca è inferiore a 3 sondaggi
- L'andamento delle prestazioni di inserimento è piuttosto irregolare: molte operazioni sono veloci, altre (poche) richiedono la ricostruzione della tabella
- Comportamento accettabile in alcune applicazioni ma non in quelle che richiedono *garanzie sulle prestazioni assolute*
- Se supportiamo la cancellazione potrebbe essere vantaggioso supportare il ridimensionamento della tabella
 - dobbiamo utilizzare una soglia diversa per evitare sequenze di raddoppi e dimezzamenti causate da un modesto numero di inserimenti e cancellazioni

Proprietà

Una sequenza di t operazioni fra ricerche, inserimenti e cancellazioni, può essere eseguita in tempo proporzionale a t e usando semplicemente una quantità di memoria al più pari a una costante moltiplicativa per il numero di chiavi della tabella

- Il metodo visto è adatto per una libreria di *uso generale*
- Gestisce tabelle di ogni dimensione
- Il principale inconveniente è dato dal costo della redistribuzione degli elementi nella nuova tabella...
- ...e dall'allocazione di memoria quando la tabella varia di dimensione

- La scelta del metodo di hashing dipende da diversi fattori
- Tutti i metodi sono in grado di ridurre la ricerca e l'inserimento a operazioni in tempo costante
- la *scansione lineare* è il più rapido dei tre
 - se abbiamo abbastanza memoria per assicurare tabella sparsa
- l'*hashing doppio* fa un uso più efficiente della memoria
 - ma richiede tempo per il calcolo del secondo hash
- le *concatenazioni separate* sono il metodo di più facile implementazione
 - necessario un buon allocatore di memoria

- La scelta fra scansione lineare e hashing doppio dipende dal costo della funzione di hash e dal fattore di carico
 - per α piccolo (tabella sparsa) entrambi usano poche scansioni a l'hashing deve calcolare due funzioni
 - per α prossimo a 1 l'hashing doppio supera la scansione lineare
- Il confronto tra scansione lineare (e hashing doppio) con il metodo delle concatenazioni separate è più complesso
 - Le concatenazioni usano memoria extra per i link, mentre i metodi di indirizzamento aperto usano memoria extra implicitamente per arrestare le sequenze di sondaggi

- Di solito non è giustificato scegliere le concatenazioni rispetto agli altri metodi basandoci su considerazioni di prestazioni
- Il metodo viene scelto per altre ragioni:
 - facile da implementare
 - poca memoria extra
 - il peggioramento prestazionale (inevitabile) difficilmente potrà nuocere all'applicazione che è sempre più veloce della ricerca sequenziale di un fattore M



[Sed03] §14 Hashing