

Algoritmi e Strutture Dati

Capitolo 7 Tabelle hash

Camil Demetrescu, Irene Finocchi,
Giuseppe F. Italiano



Implementazioni Dizionario

Tempo richiesto dall'operazione più costosa:

- Liste $O(n)$
- Alberi di ricerca non bilanciati $O(n)$
- Alberi di ricerca bilanciati $O(\log n)$
- **Tabelle hash** $O(1)$



Tabelle ad accesso diretto

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array

Idea:

- dizionario memorizzato in array v di m celle
- a ciascun elemento è associata una chiave intera nell'intervallo $[0, m-1]$
- elemento con chiave k contenuto in $v[k]$
- al più $n \leq m$ elementi nel dizionario



Implementazione

classe TavolaAccessoDiretto **implementa** Dizionario:

dati: $S(m) = \Theta(m)$
 un array v di dimensione $m \geq n$ in cui $v[k] = elem$ se c 'è un elemento $elem$ con chiave k nel dizionario, e $v[k] = \text{null}$ altrimenti. Le chiavi k devono essere interi nell'intervallo $[0, m - 1]$.

operazioni:

$\text{insert}(elem\ e, chiave\ k)$ $T(n) = O(1)$
 $v[k] \leftarrow e$

$\text{delete}(chiave\ k)$ $T(n) = O(1)$
 $v[k] \leftarrow \text{null}$

$\text{search}(chiave\ k) \rightarrow elem$ $T(n) = O(1)$
return $v[k]$



Fattore di carico

Misuriamo il grado di riempimento di una tabella usando il fattore di carico

$$\alpha = \frac{n}{m}$$

Esempio: tabella con nomi di studenti indicizzati da numeri di matricola a 6 cifre

$$n=100 \quad m=10^6 \quad \alpha = 0,0001 = 0,01\%$$

Grande spreco di memoria!



Pregi e difetti

Pregi:

- Tutte le operazioni richiedono tempo $O(1)$

Difetti:

- Le chiavi devono essere necessariamente interi in $[0, m-1]$
- Lo spazio utilizzato è proporzionale ad m , non al numero n di elementi: può esserci grande spreco di memoria!



Tabelle hash

Per ovviare agli inconvenienti delle tabelle ad accesso diretto ne consideriamo un'estensione: le **tabelle hash**

Idea:

- Chiavi prese da un universo totalmente ordinato U (possono non essere numeri)
- Funzione hash: $h: U \rightarrow [0, m-1]$
(funzione che trasforma chiavi in indici)
- Elemento con chiave k in posizione $v[h(k)]$



Collisioni

Le tabelle hash possono soffrire del fenomeno delle **collisioni**.

Si ha una collisione quando si deve inserire nella tabella hash un elemento con chiave u , e nella tabella esiste già un elemento con chiave v tale che **$h(u)=h(v)$** : il nuovo elemento andrebbe a sovrascrivere il vecchio!



Funzioni hash perfette

Un modo per evitare il fenomeno delle collisioni è usare **funzioni hash perfette**.

Una funzione hash si dice **perfetta** se è iniettiva, cioè per ogni $u, v \in U$:

$$u \neq v \Rightarrow h(u) \neq h(v)$$

Deve essere $|U| \leq m$



Implementazione

classe `TavolaHashPerfetta` **implementa** Dizionario:

dati:

$$S(m) = \Theta(m)$$

un array v di dimensione $m \geq n$ in cui $v[h(k)] = e$ se c'è un elemento e con chiave $k \in U$ nel dizionario, e $v[h(k)] = \text{null}$ altrimenti. La funzione $h : U \rightarrow \{0, \dots, m - 1\}$ è una funzione hash perfetta calcolabile in tempo $O(1)$.

operazioni:

`insert(elem e , chiave k)`
 $v[h(k)] \leftarrow e$

$$T(n) = O(1)$$

`delete(chiave k)`
 $v[h(k)] \leftarrow \text{null}$

$$T(n) = O(1)$$

`search(chiave k)` \rightarrow *elem*
return $v[h(k)]$

$$T(n) = O(1)$$



Esempio

Tabella hash con nomi di studenti aventi
come chiavi numeri di matricola
nell'insieme $U=[234717, 235717]$

Funzione hash perfetta: $h(k) = k - 234717$

$n=100$ $m=1000$ $\alpha = 0,1 = 10\%$

L'assunzione $|U| \leq m$ necessaria per avere
una funzione hash perfetta è raramente
conveniente (o possibile)...



Esempio

Tabella hash con elementi aventi come chiavi lettere dell'alfabeto $U = \{A, B, C, \dots\}$

Funzione hash non perfetta (ma buona in pratica per m primo): $h(k) = \text{ascii}(k) \bmod m$

Ad esempio, per $m=11$: $h('C') = h('N')$
 \Rightarrow se volessimo inserire sia 'C' and 'N' nel dizionario avremmo una collisione!



Uniformità delle funzioni hash

Per ridurre la probabilità di collisioni, una buona funzione hash dovrebbe essere in grado di distribuire in modo uniforme le chiavi nello spazio degli indici della tabella

Questo si ha ad esempio se la funzione hash gode della proprietà di **uniformità semplice**



Uniformità semplice

Sia $P(k)$ la probabilità che la chiave k sia presente nel dizionario e sia:

$$Q(i) = \sum_{k:h(k)=i} P(k)$$

la probabilità che la cella i sia occupata.

Una funzione hash h gode **dell'uniformità semplice** se:

$$Q(i) = \frac{1}{m}$$



Esempio

Se U è l'insieme dei numeri reali in $[0,1]$ e ogni chiave ha la stessa probabilità di essere scelta, allora si può dimostrare che la funzione hash:

$$h(k) = \lfloor km \rfloor$$

soddisfa la proprietà di uniformità semplice



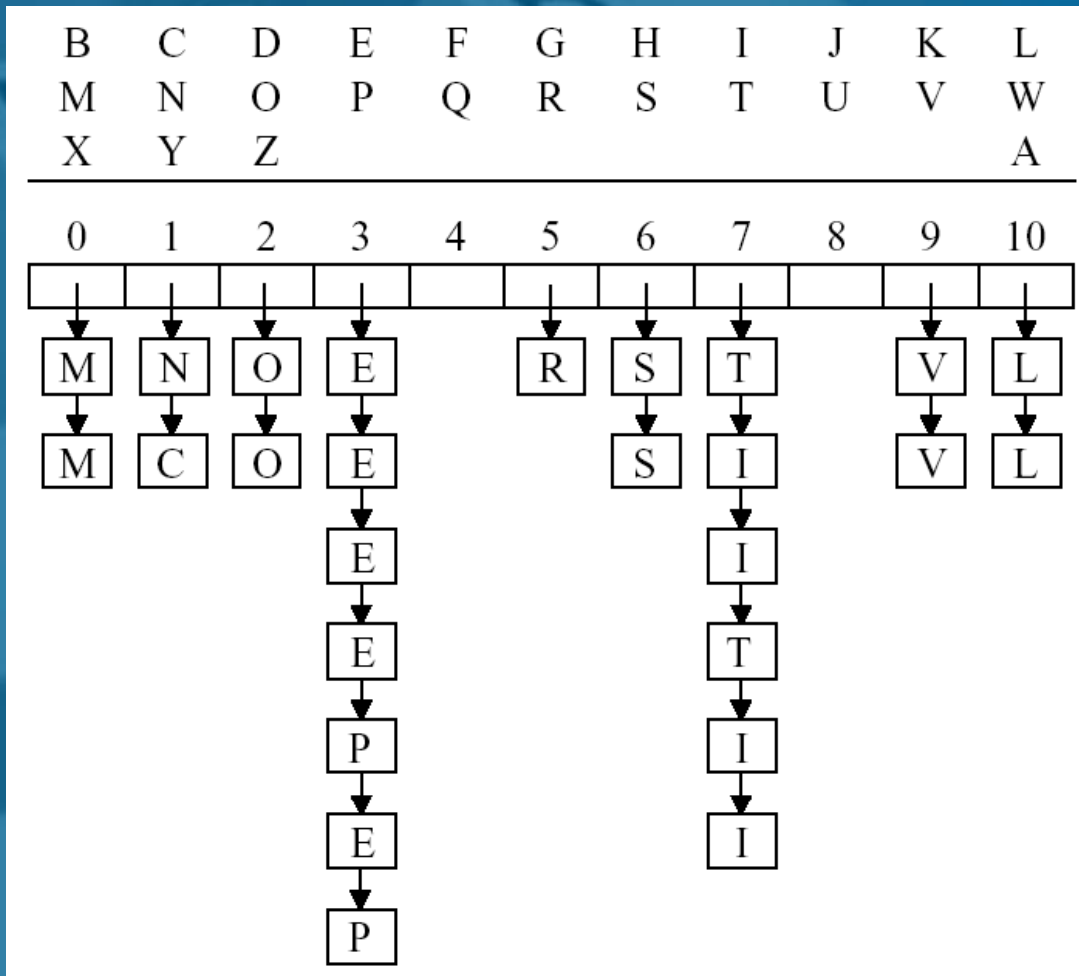
Risoluzione delle collisioni

Nel caso in cui non si possano evitare le collisioni, dobbiamo trovare un modo per risolverle. Due metodi classici sono i seguenti:

1. **Liste di collisione.** Gli elementi sono contenuti in liste esterne alla tabella: $v[i]$ punta alla lista degli elementi tali che $h(k)=i$
2. **Indirizzamento aperto.** Tutti gli elementi sono contenuti nella tabella: se una cella è occupata, se ne cerca un'altra libera



Liste di collisione



Esempio di tabella hash basata su liste di collisione contenente le lettere della parola:

**PRECIPITEVOLIS
SIMEVOLMENTE**



Implementazione

classe `TavolaHashListeColl` **implementa** `Dizionario`:

dati:

$$S(m, n) = \Theta(m + n)$$

un array v di dimensione m in cui ogni cella contiene un puntatore a una lista di coppie $(elem, chiave)$. Un elemento e con chiave $k \in U$ è nel dizionario se e solo se (e, k) è nella lista puntata da $v[h(k)]$, con $h : U \rightarrow \{0, \dots, m-1\}$ funzione hash con uniformità semplice calcolabile in tempo $O(1)$.

operazioni:

`insert(elem e, chiave k)` $T(n) = O(1)$
 aggiungi la coppia (e, k) alla lista puntata da $v[h(k)]$.

`delete(chiave k)` $T_{avg}(n) = O(1 + n/m)$
 rimuovi la coppia (e, k) nella lista puntata da $v[h(k)]$.

`search(chiave k) → elem` $T_{avg}(n) = O(1 + n/m)$
 se (e, k) è nella lista puntata da $v[h(k)]$, allora restituisci e , altrimenti restituisci `null`.



Indirizzamento aperto

Supponiamo di voler inserire un elemento con chiave k e la sua posizione “naturale” $h(k)$ sia già occupata.

L’indirizzamento aperto consiste nell’occupare un’altra cella, anche se potrebbe spettare di diritto a un’altra chiave.

Cerchiamo la cella vuota (se c’è) scandendo le celle secondo una sequenza di indici:

$$c(k,0), c(k,1), c(k,2), \dots, c(k,m-1)$$



Implementazione

classe TavolaHashAperta **implementa** Dizionario:

dati: $S(m) = \Theta(m)$
 un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

insert($elem\ e, chiave\ k$)

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** $(v[c(k, i)].elem = \text{null})$ **then**
3. $v[c(k, i)] \leftarrow (e, k)$
4. **return**
5. **errore** tavola piena

delete($chiave\ k$)

errore operazione non supportata

search($chiave\ k$) $\rightarrow elem$

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** $(v[c(k, i)].elem = \text{null})$ **then**
3. **return** null
4. **if** $(v[c(k, i)].chiave = k)$ **then**
5. **return** $v[c(k, i)].elem$
6. **return** null



Metodi di scansione: scansione lineare

Scansione lineare:

$$c(k,i) = (h(k) + i) \bmod m$$

per $0 \leq i < m$



Esempio

Inserimenti in
tabella hash basata
su indirizzamento
aperto con
scansione lineare
delle lettere della
parola:

**PRECIPITEVOLIS
SIMEVOLMENTE**

	C	E	I	L	M	N	O	P	R	S	T	V									
0																					
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					
9																					
10																					
11																					
12																					
13																					
14																					
15																					
16																					
17																					
18																					
19																					
20																					
21																					
22																					
23																					
24																					
25																					
26																					
27																					
28																					
29																					
30																					
P									P												
R								P	R												
E								P	R												
C								P	R												
I								P	R												
P								P	P	R											
I								P	P	R											
T								P	P	R	T										
E								P	P	R	T										
V								P	P	R	T	V									
O								O	P	P	R	T	V								
L								O	P	P	R	T	V								
I								O	P	P	R	T	V								
S								O	P	P	R	S	T	V							
S								O	P	P	R	S	T	S	V						
I								O	P	P	R	S	T	S	V						
M								O	P	P	R	S	T	S	V						
E								O	P	P	R	S	T	S	V						
V								O	P	P	R	S	T	S	V	V					
O								O	P	P	R	S	T	S	V	V	O				
L								O	P	P	R	S	T	S	V	V	O	L			
M								O	P	P	R	S	T	S	V	V	O	L	M		
E								O	P	P	R	S	T	S	V	V	O	L	M		
N								O	P	P	R	S	T	S	V	V	O	L	M	N	
T								O	P	P	R	S	T	S	V	V	O	L	M	N	T
E	E							O	P	P	R	S	T	S	V	V	O	L	M	N	T

4,8 celle scandite in media per inserimento



Metodi di scansione: hashing doppio

La scansione lineare provoca effetti di **agglomerazione**, cioè lunghi gruppi di celle consecutive occupate che rallentano la scansione

L'**hashing doppio** riduce il problema:

$$c(k,i) = \lfloor h_1(k) + i \cdot h_2(k) \rfloor \bmod m$$

per $0 \leq i < m$, h_1 e h_2 funzioni hash



Analisi del costo di scansione

Tempo richiesto in media da un'operazione di ricerca di una chiave, assumendo che le chiavi siano prese con probabilità uniforme da U :

<i>esito ricerca</i>	<i>sc. lineare</i>	<i>hashing doppio</i>
chiave trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$-\frac{1}{\alpha} \log_e(1 - \alpha)$
chiave non trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$

dove $\alpha = n/m$ (fattore di carico)



Cancellazione elementi con indir. aperto

classe TavolaHashApertaBis **implementa** Dizionario:

dati: $S(m) = \Theta(m)$

un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

insert($elem\ e, chiave\ k$)

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** $(v[c(k, i)].elem = \text{null} \text{ or } v[c(k, i)].elem = \text{canc})$ **then**
3. $v[c(k, i)] \leftarrow (e, k)$
4. **return**
5. **errore** tavola piena

delete($chiave\ k$)

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** $(v[c(k, i)].elem = \text{null})$ **then**
3. **errore** chiave non in dizionario
4. **if** $(v[c(k, i)].chiave = k \text{ and } v[c(k, i)].elem \neq \text{canc})$ **then**
5. $v[c(k, i)].elem \leftarrow \text{canc}$
6. **errore** chiave non in dizionario

search($chiave\ k$) $\rightarrow elem$

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** $(v[c(k, i)].elem = \text{null})$ **then**
3. **return** null
4. **if** $(v[c(k, i)].chiave = k \text{ and } v[c(k, i)].elem \neq \text{canc})$ **then**
5. **return** $v[c(k, i)].elem$
6. **return** null



Riepilogo

- La proprietà di accesso diretto alle celle di un array consente di realizzare **dizionari con operazioni in tempo $O(1)$** indicizzando gli elementi usando le loro stesse chiavi (purché siano intere)
- L'array può essere molto grande se lo spazio delle chiavi è grande
- Per ridurre questo problema si possono usare **funzioni hash** che trasformano chiavi (anche non numeriche) in indici
- Usando funzioni hash possono aversi **collisioni**
- Tecniche classiche per risolvere le collisioni sono **liste di collisione** e **indirizzamento aperto**