Tabelle

Salvatore Filippone
salvatore.filippone@uniroma2.it

Possiamo accedere e recuperare informazioni in tempo $O(1)$?

Possiamo accedere e recuperare informazioni in tempo $O(1)$?
In effetti, questo è quello che accade quando accediamo degli *array* mono-

$$rv = V[k]$$

o multi-dimensionali

$$ra = A[i][j]$$

Quindi, *se* le informazioni sono indirizzate da indici numerici, *e se* l'indice numerico è calcolabile in tempo $O(1)$, *allora* possiamo accedere in tempo $O(1)$. Ma cosa accade in effetti quando il compilatore traduce il codice che abbiamo appena visto?

Un esempio monodimensionale

Consideriamo il codice:

```
double v[10];
i=4;
x=v[i];
```

Per accedere `v[i]` il compilatore ha bisogno di conoscere:

- L'indirizzo di partenza del vettore `v`;
- Le dimensioni di ciascun elemento (per un `double`: 8 bytes);
- L'indice del primo elemento `xb` (in C e nei linguaggi derivati: 0);

Il compilatore può allora calcolare l'indirizzo in memoria dell'elemento da accedere:

```
addr = start(v) + size * ( i - xb)
     = start(v)+ 8*(4-0)
```

Array multidimensionali

Cosa è un array multidimensionale?

Una collezione di oggetti, tutti dello stesso tipo, identificati da un *insieme* di indici

Una prima versione:

Si può vedere un array 2D come una collezione di array 1D

ma questo va parzialmente contro la interpretazione normale del compilatore

Compiler view

Un array è una collezione di oggetti tale che sia possibile identificare la posizione in memoria di ciascuno di essi dati i suoi indici (ed un insieme di dati ausiliari di cardinalità limitata)

Un esempio 2-D

Consideriamo il codice:

```
double a[10][20];
i=4; j=3;
x=a[i][j];
```

Come viene memorizzato l'array 2D?

Un esempio 2-D

Consideriamo il codice:

```
double a[10][20];
i=4; j=3;
x=a[i][j];
```

Come viene memorizzato l'array 2D? Il problema è che si deve tradurre una struttura multidimensionale in una memoria che è monodimensionale (la memoria RAM è sostanzialmente un vettore, ogni locazione è identificata da uno ed un solo indirizzo).
In C, e nei linguaggi derivati, si usa la *memorizzazione per righe.*

Array multidimensionali

Per accedere `a[i][j]` il compilatore ha bisogno di conoscere:

- L'indirizzo di partenza in memoria dell'array `a`;
- La dimensione di ciascun elemento dell'array (per un `double`: 8 byte);
- L'indirizzo iniziale in ciascuna dimensione `xb` (in C: 0);
- Il numero di elementi `NC` in ciascuna riga (in questo caso: 20)

Allora il compilatore può calcolare

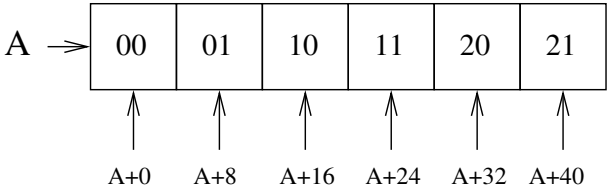
```
addr = start(a) + size * ( (i - xb)*NC + (j-xb) )
     = start(a) + 8*( (4-0)*20+(3-0))
```

La formula che abbiamo appena visto è un esempio di una *funzione di indice*

Array multidimensionali

A

00	01
10	11
20	21



Ci sono almeno due altri metodi per memorizzare un array multidimensionale.
Per cominciare, si possono memorizzare i dati: *per colonne*. E il primo indice potrebbe essere 1 invece che 0!

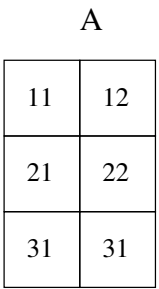
Array multidimensionali

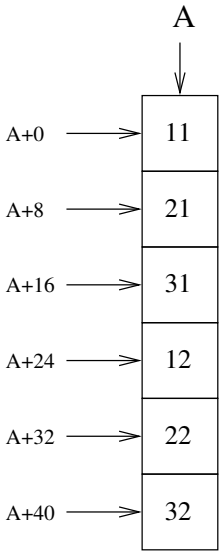
Ci sono almeno due altri metodi per memorizzare un array multidimensionale.
Per cominciare, si possono memorizzare i dati: *per colonne*. E il primo indice potrebbe essere 1 invece che 0!
Questo schema è utilizzato nei linguaggi Matlab, Fortran e Julia; la funzione di indice deve essere modificata usando il numero di *righe* invece che il numero di colonne:

```
addr = start(a) + size * ( (j - xb)*NR + (i -xb))
     = start(a) + 8*( (3-1)*10+(4-1))
```

- Matlab (e Julia) *richiede* che il primo indice sia 1;
- Fortran consente di scegliere l'indice iniziale `xb`, e 1 è il default.

Array multidimensionali





Cosa succede se il numero di righe/colonne non è noto a tempo di compilazione?

Per esempio, possiamo scrivere una funzione che accetta una variabile a: potremmo voler passare dinamicamente alla funzione il numero di righe e colonne, per interpretare correttamente la variabile. Se si può fare, allora il linguaggio che state usando supporta gli array bi- e multi-dimensionali, nel senso più generale.

Cosa succede se il numero di righe/colonne non è noto a tempo di compilazione?

Per esempio, possiamo scrivere una funzione che accetta una variabile `a`: potremmo voler passare dinamicamente alla funzione il numero di righe e colonne, per interpretare correttamente la variabile. Se si può fare, allora il linguaggio che state usando supporta gli array bi- e multi-dimensionali, nel senso più generale.

Questo non accade in C

Nel linguaggio c il compilatore interpreta l'espressione

```
x = mat[i][j];
```

in due modi completamente diversi, a seconda che le dimensioni della variabile `mat` siano note a tempo di compilazione oppure no.

In C quando le dimesioni sono note solo a runtime...

...il compilatore interpreta

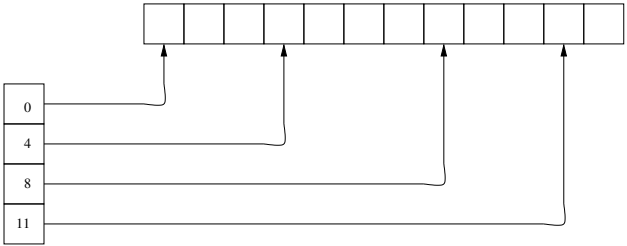
```
x = mat[i][j];
```

come:

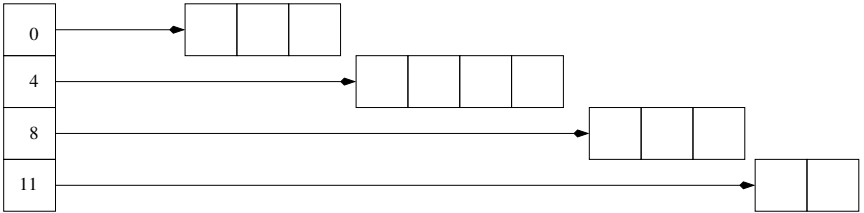
- `mat` è (un puntatore ad un) array di puntatori; con `mat[i]` si seleziona l'elemento (puntatore) all'indice `[i]`;
- Ciascun puntatore identifica un distinto array di riga; gli elementi di ciascuna riga sono indirizzati da `[j]`;
- Ciascuna riga è allocata indipendentemente, e non c'è alcuna ovvia relazione tra le posizioni in memoria degli elementi di due qualsiasi righe.

Array multidimensionali (*access vectors*, *dope vectors*

Versione 1



Versione 2



When the size is only known at runtime...

...In Fortran, Matlab e Julia, le dimensioni sono incluse nell'oggetto array

```
real :: A(:,:)
x = A(i,j)
```

ossia

- `A` contiene i dettagli sulle dimensioni dell'array;
- Il compilatore può quindi implementare la formula che abbiamo visto in precendenza;

Ma i vari linguaggi supportano gli array multidimensionali contigui?

Ma i vari linguaggi supportano gli array multidimensionali contigui?

Matlab	Si (per colonne)
Fortran	Si (per colonne)
Julia	Si (per colonne)

Ma i vari linguaggi supportano gli array multidimensionali contigui?

Matlab	Si (per colonne)
Fortran	Si (per colonne)
Julia	Si (per colonne)
Java	No (per righe)

Array multidimensionali

Ma i vari linguaggi supportano gli array multidimensionali contigui?

Matlab	Si (per colonne)
Fortran	Si (per colonne)
Julia	Si (per colonne)
Java	No (per righe)
C	È complicato (ed è per righe)
C++	È complicato (ed è per righe)

In C/C++/Java come abbiamo visto il codice

```
void compute(double a[][])
```

comporta che a sia interpretato come

Un array 1-D di puntatori (ad array 1-D)

e naturalmente questi puntatori (anche nel caso in cui le righe abbiano tutte la stessa lunghezza) possono puntare ad una qualunque posizione di memoria, per cui per sapere dove trovare `mat[2][3]` bisogna esaminare il *puntatore* contenuto in `mat[2]`, e poi accedere gli elementi puntati, all'indice 3.

Soluzioni per array 2-D contigui in memoria

- Con Fortran, Matlab, Julia, siamo già a posto;

Soluzioni per array 2-D contigui in memoria

- Con Fortran, Matlab, Julia, siamo già a posto;
- In Java, non c'è modo (ma vedi anche sotto);

Soluzioni per array 2-D contigui in memoria

- Con Fortran, Matlab, Julia, siamo già a posto;
- In Java, non c'è modo (ma vedi anche sotto);
- In C o C++, si possono dichiarare gli array *come se* fossero 1-D, per poi realizzare la funzione di indirizzamento "a mano"

```
void compute(int nr, int nc, double mat[])
x = mat[my2Dindex(i,j,nr,nc)]

int my2Dindex(i,j,nr,nc) { return(i*nc+j);}
```

In questo caso è *opportuno* che il compilatore faccia *inlining*.

Ma c'è ancora una variante possibile

Soluzioni per array 2-D contigui in memoria

Se stiamo usando il C, *e se* si può forzare il compilatore ad usare la semantica dello standard *C99*, *allora* si può scrivere il codice

```
void compute(int nr, int nc, double mat[nr][nc])
```

ed il compilatore interpreterà l'array 2D come memorizzato in modo contiguo.
Notare che si *devono* mettere `nr,nc` *prima di* `mat` nella lista degli argomenti function argument list.

Se volete divertirvi

provate a cercare

How do I declare a 2d array in C++ using new?

Ancora sulle funzioni di indice

Supponiamo di avere una certa quantità di dati, con chiavi numeriche, e che possano avere delle ripetizioni (ossia diversi record possono contenere la stessa chiave)

Ancora sulle funzioni di indice

Supponiamo di avere una certa quantità di dati, con chiavi numeriche, e che possano avere delle ripetizioni (ossia diversi record possono contenere la stessa chiave) Si potrebbero memorizzare i puntatori ai record in una *tabella bidimensionale*

R1	R2	nil	nil
R3	nil	nil	nil
R4	R5	nil	nil

Quali possono essere vantaggi/svantaggi di questa soluzione?

Ancora sulle funzioni di indice

Supponiamo ora di avere una certa quantità di dati, con chiavi memorizzate su 8 bytes, quindi con valori nell'insieme $[1 : 2^{64}]$.

- Usare le chiavi come indici in una tabella consente di fare ricerche in tempo $O(1)$;
- L'idea potrebbe essere generalizzata al caso di chiavi non numeriche (p.es. stringhe di caratteri), semplicemente usando la loro rappresentazione binaria.

Qual è il problema fondamentale di questo schema?

Ancora sulle funzioni di indice

Supponiamo ora di avere una certa quantità di dati, con chiavi memorizzate su 8 bytes, quindi con valori nell'insieme $[1 : 2^{64}]$.

- Usare le chiavi come indici in una tabella consente di fare ricerche in tempo $O(1)$;
- L'idea potrebbe essere generalizzata al caso di chiavi non numeriche (p.es. stringhe di caratteri), semplicemente usando la loro rappresentazione binaria.

Qual è il problema fondamentale di questo schema?
Predisporre una tabella accessibile per tutti i valori possibili della chiave comporta l'uso di una grande quantità di memoria.
Ad esempio, con chiavi da 8 caratteri ci sono $26^8 \approx 2 \times 10^{11}$ valori possibili, e quindi righe della tabella!

Ancora sulle funzioni di indice

Supponiamo ora di avere una certa quantità di dati, con chiavi memorizzate su 8 bytes, quindi con valori nell'insieme $[1 : 2^{64}]$.

- Usare le chiavi come indici in una tabella consente di fare ricerche in tempo $O(1)$;
- L'idea potrebbe essere generalizzata al caso di chiavi non numeriche (p.es. stringhe di caratteri), semplicemente usando la loro rappresentazione binaria.

Qual è il problema fondamentale di questo schema?
Predisporre una tabella accessibile per tutti i valori possibili della chiave comporta l'uso di una grande quantità di memoria.
Ad esempio, con chiavi da 8 caratteri ci sono $26^8 \approx 2 \times 10^{11}$ valori possibili, e quindi righe della tabella!
Se l'insieme dei valori delle chiavi che si presentano *in pratica* è più piccolo dell'insieme dei valori *teoricamente possibili*, si ha uno spreco di memoria.

Consideriamo quindi di nuovo il nostro problema:

Funzioni indice

Data una chiave $K \in U$ appartenente ad un universo U, cerchiamo una funzione $h(K) : U \to \{1, \ldots, m\}$ iniettiva, ossia tale che

$$K_1 \neq K_2 \Rightarrow h(K_1) \neq h(K_2)$$

L'iniettività richiede che i valori di $h(K)$ appartengano ad un insieme che abbia la stessa cardinalità di U, ossia $m = |U|$.

Un esempio

Supponiamo di dover memorizzare in una tabella delle informazioni su un insieme di musicisti:

- Albinoni
- Offenbach
- Bach
- Prokofiev
- Puccini
- Rossini

È evidente che una funzione di indirizzamento che consenta di gestire qualsiasi cognome immaginabile (p.es. Palestrina, Verdi, Gershwin, Mozart, Monteverdi) e che sia iniettiva dovrebbe avere un codominio di dimensione enorme.
Si noti che stiamo assumendo che ci sia un limite superiore alla lunghezza di un cognome, perché l'insieme di tutte le stringhe di lunghezza arbitraria ha una cardinalità infinita.

Cosa succede se rilassiamo la richiesta di iniettività?

Cosa succede se rilassiamo la richiesta di iniettività?

Funzioni di *hashing*

La dimensione della tabella può essere $m \ll |U|$; occasionalmente si può avere una *collisione*

$$K_1 \neq K_2,\ h(K_1) = h(K_2).$$

Sarà necessario avere un meccanismo per la risoluzione delle collisioni; se queste sono sufficientemente rare, la loro gestione potrebbe non costare troppo.
La funzione $h(K)$ viene normalmente detta *hash*, o funzione di hashing.

Requisiti per una tabella *hash*

- Che sia disponibile una funzione $h()$ calcolabile velocemente ($O(1)$);
- Che la funzione $h()$ sia tale che i valori calcolati sull'insieme delle chiavi (attese) siano ben distribuiti;
- Che sia disponibile un metodo per la risoluzione delle collisioni;
- (Che la dimensione m della tabella sia una sovrastima della cardinalità dell'insieme delle chiavi da gestire, in modo da evitare un riempimento completo).

L'ultimo requisito è *necessario* per almeno una tipologia di tabelle *hash*.

La funzione $h(K)$

La funzione h dovrebbe essere *uniforme*, ossia la probabilità che una chiave qualsiasi K venga inserita alla posizione i nella tabella (di dimensione m) dovrebbe corrispondere ad una distribuzione uniforme:

$$Q(i) = \sum_{K:h(K)=i} P(K,i) \approx \frac{1}{m}.$$

La funzione $h(K)$

La funzione h dovrebbe essere *uniforme*, ossia la probabilità che una chiave qualsiasi K venga inserita alla posizione i nella tabella (di dimensione m) dovrebbe corrispondere ad una distribuzione uniforme:

$$Q(i) = \sum_{K:h(K)=i} P(K,i) \approx \frac{1}{m}.$$

Come si costruisce una funzione di *hashing*?

- Troncamento;
- *Folding*.

Per gestire le chiavi si considera sempre la loro rappresentazione numerica per bytes (p.es. nella tabella ASCII base, al carattere 'A' corrisponde il byte di valore 65)

Troncamento

Spesso si usa solo un sottoinsieme della chiave.

In molte applicazioni si usano degli identificatori con parti comuni e parti varianti, come `Type1`, `Type2`, `Type3` etc., si cerca di eliminare la parte comune, p.es. estraendo da una stringa solo la sua porzione centrale.

Folding

Si spezza una chiave in più parti, e le parti vengono combinate attraverso una funzione ausiliaria.

La funzione $h(K)$

Andiamo ora a vedere qualche esempio di funzione di hash.

1. Estrazione: $h(K) = int(b)$ dove b è un sottoinsieme dei bit della rappresentazione binaria di K, di solito la parte centrale;
2. XOR: $h(K) = int(b)$, $b = p_1(K) \oplus p_2(K) \oplus \cdots \oplus p_n(K)$ dove $p(K)$ è una partizione della chiave K e $\oplus$ è l'operatore di OR-esclusivo bit a bit;
3. Moltiplicazione: $h(K) = \lfloor m(iC - \lfloor iC \rfloor) \rfloor$, dove m è un numero intero qualsiasi, i è $int(bin(K))$, C è un numero reale $0 < C < 1$;
4. Divisione: si usa il resto della divisione $h(K) = int(bin(K)) \mod m$

- Nel metodo di moltiplicazione si può scegliere un valore m arbitrario, ma serve una buona scelta di C; D. Knuth propone $C = (\sqrt{5} - 1)/2$;
- Nel metodo della divisione serve un valore di m dispari; infatti, assumendo la rappresentazione binaria dei dati, con $m = 2^p$ tutti i valori che abbiano gli stessi p bit finali andranno a collidere.

In particolare, nel metodo di divisione si usano spesso dei valori primi distanti dalle potenze di due; ad esempio, 13, 23, 47, 97, 193, 383, 769, 1531, 6143, 12289

La funzione $h(K)$

Algorithm 1: Integer h($Item[], int\ l$)

int $b \leftarrow ord(k(1))$;
for $j = 2$ *to* l **do**
  $b \leftarrow (b \cdot 256 + ord(k([j]))) \mod m$;
return b

cfr. algoritmo di Horner

Un esempio

Un esempio di funzione h (non molto efficiente per la verità): dato un cognome, la funzione h prende il valore 0 se la prima lettera è A, 1 se è B e così via.

$$h(\text{Albinoni}) = 0$$
$$h(\text{Offenbach}) = 14$$
$$h(\text{Bach}) = 1$$
$$h(\text{Prokofiev}) = 15$$
$$h(\text{Puccini}) = 15$$
$$h(\text{Rossini}) = 17$$

La risoluzione delle collisioni

Ci sono due classi di metodi principali per la risoluzione delle collisioni:

- Metodi interni: La scansione interna (*open addressing*).
- Metodi esterni: Le liste di trabocco (*chaining*);

Scansione interna (lineare)

Si estende la funzione h con un secondo argomento

$$h(K, i),\ i = 0, \ldots m-1;$$

se l'elemento della tabella indirizzato da $h(K, i)$ è occupato, si passa al valore successivo di i, fin quando si sia scandita tutta la tabella; se la ricerca di una posizione libera è infruttuosa, la tabella è piena.

Una versione molto comune è la scansione *lineare* in cui

$$h(K, i) = (h(k) + s \cdot i) \mod m;$$

se $s = 1$ si sta semplicemente incrementando il risultato del precedente calcolo di h.
Attenzione: se si cancella un elemento, non si può marcare una posizione come "libera".

Scansione lineare

Si noti la formazione di un *cluster*, ossia di un *assembramento*. Gli assembramenti tendono ad espandersi.

È opportuno che la funzione di *scansione* tocchi *tutti* i valori tra 0 e $m - 1$, in modo che se esiste una posizione vuota nella tabella essa venga opportunamente utilizzata.

Altre strategie:

- Scansione quadratica

$$h(K, i) = (h(k) + s_1 \cdot i + s_2 \cdot i^2) \mod m;$$

- Scansione pseudocasuale

$$h(K, i) = (h(k) + r_i) \mod m;$$

- Doppio hashing:

$$h(K, i) = (h(k) + ih'(K);$$

Vantaggi della scansione (lineare)

- Semplicità dell'algoritmo;
- Memorizzazione delle sole chiavi.

Svantaggi:

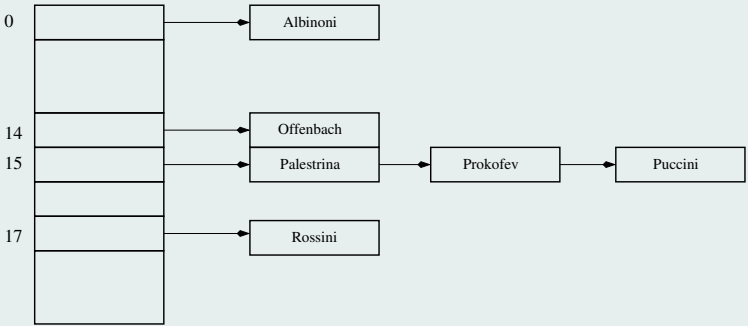
- Creazione di *cluster* che tendono ad espandersi;
- Necessità di sovradimensionamento della tabella

L'algoritmo va in crisi quando la tabella è vicina ad essere totalmente riempita, quindi si deve sovradimensionarla opportunamente. Una regola empirica è che le tabelle con scansione interna tendono a funzionare bene fino ad una occupazione di circa il 60 % dello spazio disponibile.

Liste di trabocco

Per ogni indice della tabella, si memorizza una lista, a cui vengono via via aggiunte le chiavi reinviate in quella posizione dalla funzione $h(K)$;

Liste di trabocco



Vantaggi delle liste di trabocco:

- La tabella richiede solo una lista di puntatori,
- La tabella principale può essere allocata alla dimensione minima;
- Non ci sono limiti al numero di entità memorizzate.

Tuttavia, se le chiavi sono di piccole dimensioni, si ha un aggravio di memoria dovuto ai puntatori;

Nel caso peggiore la ricerca di un elemento in una tabella *hash* ha un costo *lineare* nella dimensione della tabella stessa:

- Nella scansione interna, la ricerca può avere costo $O(m)$ dove m è la dimensione della tabella;
- Nella implementazione con liste di trabocco, la ricerca può avere costo $O(n)$ dove n è il numero di chiavi memorizzate.

Definizione

Fattore di carico

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

Definizione

Tempo medio per una ricerca infruttuosa

$$I(\alpha)$$

Definizione

Tempo medio per una ricerca fruttuosa

$$S(\alpha)$$

I tempi di inserzione e cancellazione sono determinati da $I(\alpha)$ e $S(\alpha)$.

- Per inserire un nuovo elemento in una tabella, bisogna prima cercare la sua chiave:
  - Se la ricerca è infruttuosa, si otterrà la posizione in cui inserire il nuovo elemento;
  - Se la ricerca è fruttuosa, ossia si trova la chiave, bisogna gestire la collisione.
- Per cancellare un elemento bisogna prima cercare la posizione della sua chiave.

Complessità media

Scansione	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \leq \alpha \leq 1$	$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$	$\frac{1-\alpha/2}{1-\alpha}$
Hashing doppio	$0 \leq \alpha \leq 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$
Liste di trabocco	$\alpha \geq 0$	$1+\alpha$	$1+\alpha/2$

Complessità media

	Lookup	insert	remove	min	Scansione	Ordine
Vettore booleano	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$	✓
Lista non ordinata	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	✗
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	✓
Hash (mem interna)	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	✗
Hash (mem esterna)	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$	$O(m+n)$	✗
Alberi bilanciati	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	✓