Polpette

Le tabelle hash *(in inglese* hash = *polpetta) costituiscono uno dei più convenienti metodi di memorizzazione e ricerca dell'informazione; usiamo* Mathematica *per esplorarne "sul campo" le caratteristiche*

di Francesco Romano

Introduzione

Un tipo di operazione che ricorre frequentemente è la ricerca di uno specifico elemento (detto **chiave)** in un insieme di dati dello stesso tipo (detto **tabella)**. Talvolta interessa solo sapere se la chiave è presente in tabella ma più spesso alla chiave è associata una informazione e lo scopo della ricerca è di accedere alle informazioni attraverso la chiave. L'implementazione più immediata di una tabella consiste nel memorizzarne gli elementi in un vettore e di scandire lo stesso alla ricerca del "tesoro". Con *Mathematica* si può usare **AppendTo** per inserire e **Position** per ricercare.

In[1]:=
```
lista={pippo,pluto,minnie};
AppendTo[lista,gastone];
lista
```

Out[3]=
```
{pippo, pluto, minnie, gastone}
```

In[4]:=
```
Position[lista,minnie]
```

Out[4]=
```
{{3}}
```

In[5]:=
```
Position[lista,gambadilegno]
```

Out[5]=
```
{}
```

L'inconveniente letale di questo metodo è il tempo lineare necessario per la ricerca. In altre parole se cerchiamo un dato in una lista di n elementi "disordinati" sono necessari, in media, n/2 accessi per trovare l'oggetto e, nel caso peggiore, n accessi per concludere che l'oggetto è assente. Un'idea della scomodità di questo metodo la si può avere cercando nell'elenco telefonico un abbonato che abiti in via "Paolo Rossi", partendo da "Abate Abia" e arrivando eventualmente fino a "Zurlì Zobeide"

Se la tabella è organizzata in modo ordinato si può andare più spediti dividendola a metà e decidendo con un solo confronto in quale metà guardare; un facile (!) calcolo mostra che la complessità della ricerca è circa il logaritmo in base 2 del numero degli elementi della tabella. La ricerca umana in un elenco telefonico segue un algoritmo simile a questo e per 3 milioni di abbonati bastano meno di 20 passi. Questo spiega ampiamente perché gli elenchi telefonici si stampino in ordine alfabetico.

Se le chiavi appartengono ad un insieme di cardinalità piccola (numeri tra 1 e 1000, stringhe di 2 *byte*, etc.) esiste un modo molto efficiente di implementare gli algoritmi di inserzione, ricerca e cancellazione, la cui complessità è sempre uguale ad 1 in ogni caso: la **tabella ad accesso diretto**.

Si costruisce un vettore che contiene spazio per le informazioni associate a tutte le possibili chiavi. Se l'informazione è assente il vettore è costituito da valori di verità che dicono solamente se la chiave è presente.

Il punto debole di questo metodo è la quantità di memoria richiesta: in genere, quando le chiavi sono molto lunghe solo una parte trascurabile delle chiavi possibili viene in pratica utilizzata. Per esempio: volendo memorizzare un elenco telefonico di una grande città con 3 milioni di utenti con una tabella ad accesso diretto che riservi 20 *byte* per la chiave servirebbe uno spazio di memoria di $26^{20} \approx 1.99\ 10^{28}$ *byte* (!).

Tabelle hash

Anche se l'eccessiva quantità di memoria richiesta rende quasi sempre inapplicabile l'algoritmo dell'accesso diretto, questo ci offre uno spunto per una realizzazione più sofisticata e compatta e quasi altrettanto efficiente. Nel caso visto precedentemente la funzione che associava alla chiave una posizione nella tabella era biunivoca, ciò risolveva ogni problema di accesso ma richiedeva una tabella di dimensioni pari alla cardinalità dell'intero spazio delle chiavi. Si consideri ora una funzione che trasforma una chiave in un numero intero che assume valori in un intervallo molto più ristretto (per esempio tra 1 e 1000, qualunque sia la lunghezza della chiave). Ogni chiave ha ancora un unico posto nella tabella ma purtroppo a più chiavi è riservato lo stesso posto.

La prima chiave da inserire trova certamente il suo posto libero e viene collocata senza problemi ed è molto probabile che anche la seconda chiave vada in un posto libero, ma quasi certamente si dovrà, prima o poi, inserire una chiave in un posto già occupato, provocando una **collisione**. L'idea che sta alla base del metodo è quella di risolvere le collisioni assegnando opportunamente i posti liberi. Questa tecnica è detta delle **tabelle *hash***. I punti caratteristici di algoritmi di inserzione e ricerca di questo tipo consistono nei diversi metodi di calcolo della funzione che assegna l'indirizzo alla chiave, detta **funzione *hash*** e nella strategia di gestione delle collisioni. Esaminiamo separatamente i due aspetti.

Le caratteristiche principali di una buona funzione *hash* sono quelle di garantire una uniforme distribuzione delle chiavi

all'interno della tabella. Per fare ciò la funzione deve dipendere da tutti i *bit* della chiave in maniera sufficientemente casuale. Da evitare per esempio nel caso di chiavi alfanumeriche una dipendenza dai primi *byte* (potrebbero esserci molte chiavi che cominciano con le stesse lettere) o dagli ultimi (potrebbero essere quasi tutti spazi). Vediamo ora alcuni esempi di buone funzioni *hash*.

1) Se la chiave è numerica, una buona distribuzione nell'intervallo **0, LEN-1** è data da:

In[1]:=
```
hash[x_Integer]:=Mod[x,LEN])
```

Se **LEN** è potenza di due l'operazione di modulo si riduce a considerare i *bit* meno significativi di **c** e, per quanto detto sopra, ciò è sconsigliabile. I migliori risultati si ottengono scegliendo per **LEN** un numero primo.

2) Se la chiave non è numerica ci si può ridurre al caso precedente spezzando la chiave in tanti segmenti di lunghezza pari alla parola di macchina usata per rappresentare i numeri interi (per esempio 16 o 32 *bit*) e poi combinando tra loro tali segmenti con opportune operazioni (ad esempio lo "or esclusivo").

Mathematica mette a disposizione una funzione *hash* capace di "triturare" qualsiasi cosa:

In[2]:=
```
?Hash
Hash[expr] gives an integer hash code for any
expression expr.
```

In[3]:=
```
Hash[{Pi/2, "MCMicrocomputer"}]
```

Out[3]=
```
1945769929
```

Una volta calcolata la funzione *hash* della chiave da trattare, questa ci dà un indirizzo in tabella; se la posizione è libera la scansione è terminata. Se la cella in questione è occupata da un'altra chiave, si deve generare un indirizzo alternativo (interno alla tabella) e il procedimento continua finché non si trova la chiave cercata oppure una cella libera.

La strategia più semplice di scelta del nuovo indirizzo è quella della **scansione lineare**: se **LEN** è la dimensione della tabella, allora la sequenza di indirizzi in cui si va a cercare la chiave in tabella è data da:

In[4]:=
```
NuovoIndirizzo[h0_,i_]:=
      Mod[h0+i,LEN];
```

In questo modo sono probabili i cosiddetti **agglomerati primari di chiavi**: in caso di collisioni si formano, cioè, delle zone piene che favoriscono ulteriori collisioni.

Vediamo un esempio disegnato con *Mathematica*. Si considera una tabella di 11 posizioni, l'indirizzo *hash* è ottenuto a partire dalla prima lettera (scelta terribile in una tabella vera, ma consigliabile a scopo didattico per poter seguire a mano il procedimento)

In[4]:=
```
hash[x_String]:=Mod[
   First[ToCharacterCode[
      First[Characters[x]]]],LEN]
```

Entrano successivamente vari membri della famiglia Disney. Alcuni di essi trovano il posto occupato e devono peregrinare un po'. Si vede come mano a mano che il tasso di occupazione cresce si formano larghe zone di collisione. **(Figura 1)**

Un'alternativa più efficiente consiste nel far variare il passo di scansione in modo quadratico (**scansione quadratica**):

In[5]:=
```
NuovoIndirizzo[h0_,i_]:=
   Mod[h0+i^2,LEN];
```

Con questo metodo è necessario che **LEN** sia un numero primo.

Se gli stessi elementi entrano nello stesso ordine vi sono meno collisioni.

(Figura 2)

Anche in questo caso si possono avere agglomerati (detti **secondari)**: tutte le chiavi con lo stesso indirizzo seguono ovviamente lo stesso percorso di scansione e se per due indi-

Figura 1

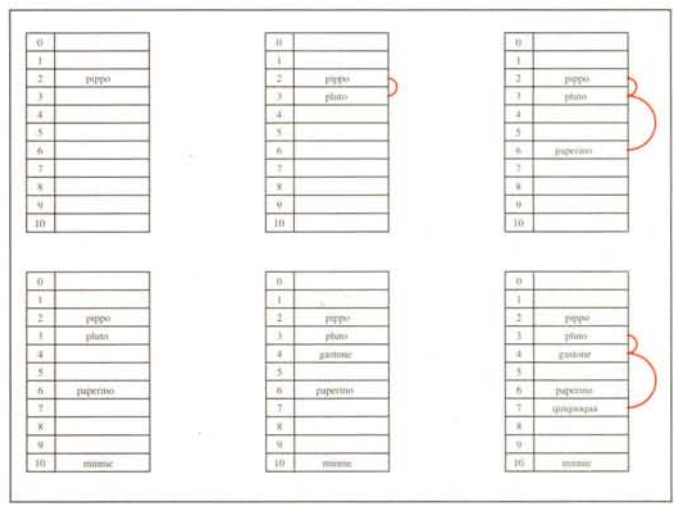

Figura 2

rizzi *hash* diversi **h1** e **h2** ed un certo **i** si ha:

```
Mod[h1+i^2,LEN] = = Mod[h2+i^2,LEN]
```

da quel punto in poi le due successioni collideranno.

Esperimenti "in grande"

Un rimedio agli agglomerati secondari consiste nel "pesare" la scansione con un coefficiente **CP** che dipende anch'esso dalla chiave (scansione quadratica a coefficienti pesati).

Per esempio se definiamo la funzione *hash* come

In[1]:=
```
hash[x_Integer]:=
  (CP=1+Floor[x/LEN];
   Mod[x,LEN]);
```

Il nuovo indirizzo è dato da

In[2]:=
```
NuovoIndirizzo[h_,i_]:=
   Mod[h0+CP i^2,LEN];
```

In generale si può dimostrare che se la tabella contiene **m** chiavi il tempo medio di ricerca dipende solo dal rapporto

$$\alpha = \mathbf{m}/(\mathbf{LEN}+1)$$

Tipicamente si misurano due grandezze,

S(α): il numero medio di accessi necessario ricerca di una chiave presente in tabella e

I(α): il numero di accessi richiesto per l'inserzione.

Vediamo adesso i valori di S(α) e I(α) calcolati sperimentalmente per una tabella di 7919 posizioni (7919 è il millesimo numero primo) con i tre tipi di scansione. **colore** è una variabile che stabilisce di che colore deve essere il grafico e vengono stampati i valori

{**m** , α, I(α) e S(α)}.

Scansione Lineare

In[3]:=
```
colore=Red;
slin=prova[1000];
 LEN = 7919
{ 395, 0.05,  1.013, 1.009}
{ 790, 0.1,   1.068, 1.032}
{1185, 0.15,  1.104, 1.07}
{1580, 0.2,   1.215, 1.091}
{1975, 0.25,  1.337, 1.152}
{2370, 0.3,   1.438, 1.214}
{2765, 0.35,  1.514, 1.214}
{3160, 0.4,   1.77,  1.321}
{3555, 0.45,  1.972, 1.366}
{3950, 0.5,   2.23,  1.538}
{4345, 0.55,  2.732, 1.561}
{4740, 0.6,   3.172, 1.69}
{5135, 0.65,  3.61,  1.792}
{5530, 0.7,   5.62,  1.857}
{5925, 0.75,  6.904, 2.5}
{6320, 0.8,   9.651, 2.891}
{6715, 0.85, 16.45,  3.317}
{7110, 0.9,  29.21,  5.381}
{7505, 0.95, 69.77, 9.422}
```

Scansione Quadratica

In[4]:=
```
colore=Blue;
squad=prova[1000];
 LEN = 7919
{ 395, 0.05,  1.013, 1.013}
{ 790, 0.1,   1.071, 1.047}
{1185, 0.15,  1.106, 1.074}
{1580, 0.2,   1.195, 1.09}
{1975, 0.25,  1.314, 1.149}
{2370, 0.3,   1.385, 1.188}
{2765, 0.35,  1.476, 1.233}
{3160, 0.4,   1.678, 1.291}
{3555, 0.45,  1.856, 1.311}
{3950, 0.5,   2.096, 1.432}
{4345, 0.55,  2.22,  1.469}
{4740, 0.6,   2.648, 1.559}
{5135, 0.65,  2.81,  1.744}
{5530, 0.7,   3.448, 1.852}
{5925, 0.75,  4.089, 1.911}
{6320, 0.8,   4.549, 2.081}
{6715, 0.85,  6.504, 2.397}
{7110, 0.9,   8.192, 2.765}
{7505, 0.95, 14.44,  3.462}
```

Scansione Quadratica a coefficienti pesati

In[5]:=
```
colore=Green;
scp=prova[1000];
 LEN = 7919
{ 395, 0.05,  1.015, 1.013}
{ 790, 0.1,   1.066, 1.055}
{1185, 0.15,  1.104, 1.052}
{1580, 0.2,   1.195, 1.078}
{1975, 0.25,  1.352, 1.163}
{2370, 0.3,   1.372, 1.192}
{2765, 0.35,  1.527, 1.202}
{3160, 0.4,   1.585, 1.291}
{3555, 0.45,  1.706, 1.288}
{3950, 0.5,   1.944, 1.398}
{4345, 0.55,  1.987, 1.461}
{4740, 0.6,   2.37,  1.532}
{5135, 0.65,  2.572, 1.551}
{5530, 0.7,   2.818, 1.768}
{5925, 0.75,  3.744, 1.872}
{6320, 0.8,   4.392, 1.996}
{6715, 0.85,  5.813, 2.214}
{7110, 0.9,   7.035, 2.438}
{7505, 0.95, 13.53,  3.143}
```

Facendo un grafico di S(α) si nota il notevole miglioramento ottenuto per alte percentuali di occupazione della tabella dalle scansioni quadratiche rispetto a quella lineare (in rosso).

In[6]:=
```
Show[slin,squad,scp,
     PlotRange->{{0,1},{1,4}}];
```

(Figura 3)

Per stimare la variabilità di S(α) in funzione della lunghezza della tabella la scansione quadratica è stata provata altre due volte con **LEN = 1223** e **LEN = 17389**. Si nota una buona stabilità del numero di accessi specie per i valori di **LEN** più grandi.

In[7]:=
```
colore=Magenta;
scp200=prova[200];
 LEN = 1223
...
colore=Cyan;
scp2000=prova[2000];
 LEN = 17389
...
Show[scp,scp200,scp2000,
     PlotRange->{{0,1},{0,4}}];
```

(Figura 4)

I programmi per disegnare tabelle

Vediamo ora brevemente i programmi usati per generare le figure 1 e 2.

ins inserisce una scritta usando il font Monaco 9 in una cella della tabella:

In[1]:=
```
ins[e_,i_]:=
   Text[FontForm[e,
    {"Monaco", 9}],{1.5,LEN-i-0.5}];
```

arco disegna un arco tra due celle toccate a seguito di una collisione:

In[2]:=
```
arco[i_,j_]:=arco[j,i]     /; j<i;
arco[i_,j_]:=
 {Red,
  Circle[{2, LEN-(i+j)/2-0.5},
   {(j-i)/10,(j-i)/2},
   {-Pi/2,Pi/2}]}          /; i<j;
```

Init inizializza la tabella e le liste **testo**, **connessioni** e **figura** che conterranno rispettivamente le scritte in formato grafico, gli archi rossi e i vari fotogrammi della tabella:

In[4]:=
```
Init[n_]:=(
  Clear[tabella];
  LEN=n;
  tabella[_]:=VUOTA;
  testo={};
  connessioni={};
  figura={};);
```

Disegna crea un fotogramma di una tabella in formato grafico:

In[5]:=
```
Disegna:=Show[
 Graphics[{
  Line[{{0.7,0},{0.7,LEN}}],
  Line[{{1,0},{1,LEN}}],
  Line[{{2,0},{2,LEN}}],
  Line/@
   Table[{{0.7,i},{2,i}},{i,0,LEN}],
  Table[
   Text[FontForm[ToString[i],
     "Monaco",9}],{0.85,LEN-i-0.5}],
   {i,0,LEN-1}],
  testo,
  connessioni}],
 AspectRatio->1,
 PlotRange->{{0.7,3},{0,12}}];
```

Inserisci realizza l'algoritmo di inserzione:

In[6]:=
```
Inserisci[x_]:=Module[{h,h0,h1,i},
 h=h0=hash[x];
 i:=0;
 While[(tabella[h]=!=VUOTA)&&
       (tabella[h]=!=x),
   i++;
   h1=NuovoIndirizzo[h0,i];
   AppendTo[connessioni,
            {arco[h,h1]}];
   h=h1];
 If[tabella[h]= =VUOTA,
```

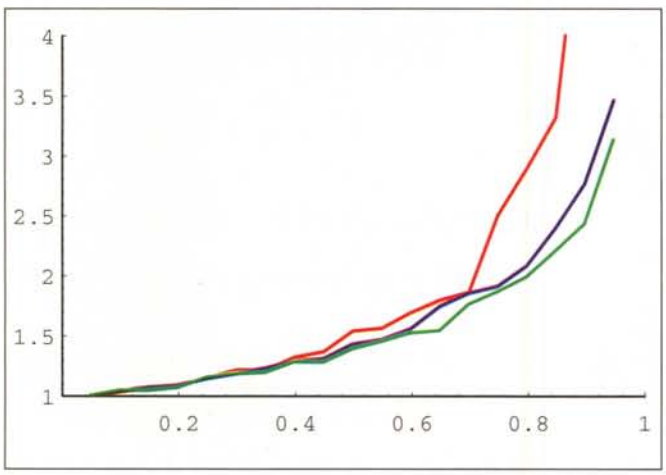

Figura 3

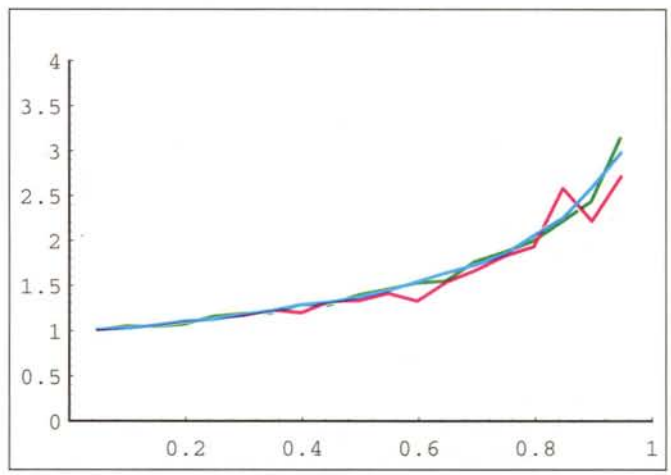

Figura 4

```
    tabella[h]=x;
   AppendTo[testo, ins[x,h]]];
 AppendTo[figura,Disegna];
 connessioni={};];
```

prova effettua una serie di inserzioni e infine mostra le varie immagini in **figura** in un **GraphicsArray**. Si noti l'uso del costrutto **Block** per assegnare temporaneamente il valore **Identity** alla variabile di sistema **$DisplayFunction** ed inibire così le stampe parziali.

In[7]:=

```
prova:=(
 Block[{$DisplayFunction=Identity},
  Init[11];
  Inserisci["pippo"];
  Inserisci["pluto"];
  Inserisci["paperino"];
  Inserisci["minnie"];
  Inserisci["gastone"];
  Inserisci["quiquoqua"];
  Inserisci["ziopaperone"];
  Inserisci["brigitta"];];
 Show[GraphicsArray[Partition
    [figura,3]]];)
```

I programmi per le prove estensive

Generiamo un campione *random* di 20.000 elementi tra 1 e 1.000.000 ed eliminiamo le ripetizioni:

In[1]:=

```
Campione=Union[Table[Random[
  Integer,{0,1000000}],{20000}]];
```

Init inizializza la tabella e alcune variabili globali:

In[2]:=

```
Init[n_]:=(
  Clear[tabella];
  LEN=n;
  KKK=Floor[LEN/20];
  m=0;
  r=0;
  tabella[_Integer]:=VUOTA);
```

Inserisci è simile a quella già vista prima ed anche **Ricerca** è analoga:

In[3]:=

```
Inserisci[x_]:=(
 h=h0=hash[x];
 i:=0;
 s++;
 While[(tabella[h]=!=VUOTA)&&
       (tabella[h]=!=x),
   i++;
   h=NuovoIndirizzo[x,h0,i];
   s++];
   If[tabella[h]= =VUOTA,
      m++;
      tabella[h]=x];)/;m<LEN;
SetAttributes[Inserisci,Listable]
```

In[4]:=

```
Ricerca[x_]:=(
   h=h0=hash[x];
   i:=0;
   r++;
   While[(tabella[h]=!=VUOTA)&&
         (tabella[h]=!=x),
      i++;
      h=NuovoIndirizzo[x,h0,i];
      r++];
   TrueQ[tabella[h]= =x]);
SetAttributes[Ricerca,Listable]
```

prova1 inserisce in tabella **KKK** elementi distinti e ne ricerca **3 KKK** scelti a caso tra quelli presenti:

In[5]:=

```
prova1[ii_]:=(
 s=0;
 Inserisci[Table[
  Campione[[(ii-1) KKK +j]],
    {j,KKK}]];
 r=0;
 Ricerca[Table[
   Campione[[Random[Integer,
       {1,ii KKK}]]], {3 KKK}]];
 Print[{m,N[m/LEN,2],
        N[s/KKK,4],
        N[r/3/KKK,4]}];
 {N[m/LEN],N[r/3/KKK]})
```

prova[k] inizializza una tabella con **LEN=Prime[k]** e chiama 10 volte **prova1** con la percentuale di occupazione che va da **0.05** a **0.95** a passi di **0.05**, viene così prodotta sia la lista dei valori che il grafico di S(α).

In[6]:=

```
prova[k_]:=(Init[Prime[k]];
  Print[" LEN = ",LEN,"\t"];
 ListPlot[Table[
      prova1[ii],{ii,19}],
  PlotJoined->True,
  PlotRange->{{0,1},Al,
  PlotStyle->colore])
```

Bibliografia

Per chi fosse interessato a maggiori dettagli sulle tabelle *hash* citiamo alcuni testi in italiano (ovviamente esiste una smisurata bibliografia in inglese).

F. Luccio, **Strutture Linguaggi, Sintassi**, Boringhieri, 1972.

F. Romani, **Introduzione alla teoria degli algoritmi**, ECIG, Genova, 1989.

G. Fiorentino, M.R.Laganà, F. Romani, F. Turini, **Pascal, Laboratorio di Programmazione**, McGraw-Hill, 1996.

MC

Francesco Romani è raggiungibile tramite Internet all'indirizzo romani@di.unipi.it

COLORADO

MAIN-BOARDS

LE ALTRE SCHEDE MADRI PENTIUM INTEL TRITON HANNO...

LA POSSIBILITA' DI MONTARE IL P6 OVER DRIVE ?

GLI SLOT PER LE DIMM A 168 PIN ?

LA POSSIBILITA' DI SUPPORTARE FINO A 200MHZ ?

512K DI CACHE SINCRONA DI SERIE ?

UNA GARANZIA DI 5 ANNI ?

UNA HOT LINE PER L'ASSISTENZA TECNICA ?

LA FORMULA "SOSTITUZIONE IN 36 ORE IN CASO DI GUASTO" ?

IL TRASPORTO GRATUITO IN CASO SIA NECESSARIA LA SOSTITUZIONE ?

IL MANUALE IN ITALIANO ?

A CORREDO TUTTO IL NECESSARIO PER IL MONTAGGIO ?

UNA CONFEZIONE SIGILLATA CHE NE GARANTISCE LA QUALITA' ?

NO? E ALLORA PERCHE' ACQUISTARE LE ALTRE SCHEDE MADRI ?

ANNUNCIO RISERVATO AI SIGG. RIVENDITORI
PER INFORMAZIONI TEL. 06/51.333.51 (6 LINEE R.A.)

Distribuito da QUADRA S.R.L. - Via C. Colombo, 193 a/b - 00147 ROMA - Tel. 51.333.51 R.A. - Fax 51.333.54 R.A.