Tabelle Hash

Per accedere agli elementi di una tabella è possibile utilizzare metodi di *accesso diretto* ai dati (detti anche metodi di *accesso calcolato*) che sono basati sulla possibilità di associare ciascuna elemento della tabella con una posizione di memoria mediante una funzione, detta *funzione di accesso*, che consente di determinare, sulla base della chiave dell'elemento (o di altre sue caratteristiche, come ad esempio le coordinate dell'elemento stesso) la corrispondente posizione. Un tipico esempio di funzione di accesso diretto, basato su un sistema di coordinate, è la matrice che consente di localizzare un elemento in base ai suoi indici.
In generale una funzione di accesso è una funzione f che ad ogni chiave associa univocamente una posizione della tabella in modo che date due chiavi K_i, e K_j con $K_i \neq K_j$ si abbia che $f(K_i) \neq f(K_j)$.
Nella realtà per chiavi alfanumeriche o per chiavi numeriche che possono assumere valori in un insieme molto più grande del numero effettivo di chiavi presenti, definire una funzione di accesso senza introdurre sprechi nel dimensionamento della tabella non è affatto semplice. Ad esempio, se dobbiamo memorizzare gli identificatori che si incontrano in un programma Pascal (stringhe alfanumeriche in genere di al più otto caratteri di cui il primo alfabetico) abbiamo che le possibili chiavi, cioè i nomi degli identificatori, sono almeno $26 \cdot 36^7$ e non è quindi pensabile pensare di utilizzare il valore numerico corrispondente alla stringa di caratteri come indice di tabella, soptrattutto se si considera che in realtà un programma Pascal avrà al più $200 \div 300$ identificatori.
Per un utilizzo pratico delle funzioni di accesso siamo costretti a perdere la caratteristica fondamentale di esse e cioè la biunivocità tra posizioni in tabella e possibili valori delle chiavi; quindi potrà avvenire che a chiavi diverse corrisponda la stessa posizione, che si verifichi, cioè, una *collisione*. In questo caso sarà necessario allocare una delle due chiavi collidenti (*sinonimi*) in un'altra posizione di memoria.
I problemi che sorgono sono dunque due:

— determinare una funzione di accesso non biunivoca ma che riduca al massimo la possibilità di collisioni (*funzione hash*),

— individuare dei criteri per allocare sinonimi (*gestione delle collisioni*).

Una *funzione hash* (di spezzettamento) è una funzione di accesso non biunivoca che, sfruttando opportunamente la struttura delle chiavi, distribuisce gli indirizzi calcolati nel modo più uniforme possibile su tutto lo spazio degli indirizzi disponibili all'interno della tabella in modo da ridurre il più possibile il numero di collisioni. Data una tabella con p posizioni disponibili e indicato con k la rappresentazione in binario di una chiave K (cioè, la stringa di bit della chiave è letta come se fosse il numero binario k), alcuni classici esempi di funzioni hash sono i seguenti:

— (*metodo del quadrato centrale*) viene determinato k^2 e vengono estratti m bit al centro del risultato, dove $m = \lfloor \log p \rfloor$; il numero risultante è ovviamente inferiore a p e viene utilizzato per indirizzare la tabella — per evitare di avere indirizzi non utilizzati conviene dimensionare p uguale esattamente a 2^m;

— (*metodo dell'avvolgimento*) la rappresentazione binaria k viene spezzata in segmenti di m bit (aggiungendo eventuali 0 all'ultimo segmento nel caso esso abbia meno di m bit), vengono sommati i vari segmenti e vengono prelevati gli m bit meno significativi della somma — m è definito come nel metodo del quadrato centrale;

— (*metodo del modulo*) come indirizzo viene utilizzato il resto della divisione di k per p; l'indirizzo assume valori tra 0 e p-1 — si noti che in questo caso è opportuno che p sia un numero dispari al fine di ridurre la probabilità di collisioni; ancora meglio se p è pari a un numero primo.

Come esempio mostriamo come derivare k a partire da una chiave di tipo *Stringa*. Supponiamo per semplicità che il valore k restituito sia un intero. Sommiamo i codici ASCII dei vari caratteri

componenti la stringa e applichiamo la funzione modulo per evitare di avere valori superiori a 4.294.967.295.

```
// file funzhash.h
#include <string.h>
#include <limits>

int stringaHash( char * s )
{     long int nMax = numeric_limits<long int>::max(); //4294967295 di solito
      int C = 0;
      for ( int i = 1; i <= strlen(s); i++ )
            C = (C + s[i]) % nMax;
      return C;
}
```

Assumendo che la lunghezza della stringa sia una costante, la complessità della funzione è anch'essa costante.
Una volta ottenuto il valore *k*, utilizziamo il metodo del modulo per calcolare l'indirizzo nella tabella con *p* entrate con la formula:

```
(k % p) + 1;
```

Ovviamente supponiamo che *p* sia minore di 4.294.967.296.
Per quanto riguarda la gestione delle collisioni una tecnica semplice e abbastanza utilizzata consiste nel collegare ad ogni entrata della tabella una lista che collega gli eventuali sinonimi (*catena di overflow*)
Possiamo ora definire una *tabella hash* organizzata come vettore di liste di elementi e con funzione hash *moduloHash*. Per l'effettivo utilizzo della tabella per un tipo *T*, è necessario definire un operatore di uguaglianza e una classe astratta *ChiaveHash* contenente la funzione di eguaglianza di chiave e una funzione virtuale *hash* che restituisca *k* a partire dalla chiave di *T* — se la chiave è una stringa tale funzione può fare uso a sua volta della funzione *stringaHash*. La funzione "toInt" di conversione a intero deve essere sempre definita.

```
template <class T>
class ChiaveHash
{
      public:
            ChiaveHash ( ) {}
            virtual int  hash ( const T& a )=0;
            virtual bool equivalenti ( const T& a1, const T& a2 )=0;
};

//NB è richiesto che sia definito correttamente l'operatore == per T
template <class T>
class TabellaHash
{
      protected:
            typedef Lista<T> lisEl;
            Vettore<lisEl> tab;
            int nEl;
            ChiaveHash<T>& ch;
      public:
            TabellaHash( int p, ChiaveHash<T>& vch);
            ~TabellaHash();
            int nCelle() const;
            bool ins( T& a );
            bool canc( const T& a );
            void svuota();
            int n() const;
            bool cerca ( T& a );
};
```

```
// TabellaHash.cpp: implementation of the TabellaHash<T> class.
//
//////////////////////////////////////////////////////////////////////
#include "TabellaHash.h"
template<class T>
TabellaHash<T>::TabellaHash( int p, ChiaveHash<T>& vch):  tab(1,p), ch(vch)
{ nEl = 0;}

template<class T>
TabellaHash<T>::~TabellaHash()
{     for ( int i = 1; i <= nCelle(); i++ )
      tab[i].svuota();
}

template<class T>
int TabellaHash<T>::nCelle() const
{     return tab.n();}

template<class T>
bool TabellaHash<T>::ins( T& a )
{
      int i = (ch.hash(a) % nCelle())+1;

      if ( cerca(a)==false )
            {
            tab[i].addInCoda(a); nEl++;
            return true;
            }
      else
            return false;
}

template<class T>
bool TabellaHash<T>::canc( const T& a )
{
      int i = (ch.hash(a) % nCelle())+1;
      Iteratore<T> it(tab[i]);
      bool trovato=false;
      it.vaiInTesta();
      while (!it.isNull() && !trovato)
            if (ch.equivalenti(it.getCurrentValue(),a))
                  trovato=true;
            else
                  it.moveNext();
      if ( trovato )
      {
            it.cancellaCorrente(); nEl--;
            return true;
      }
      else
            return false;
}

template<class T>
void TabellaHash<T>::svuota()
{     for ( int i = 1; i <= nCelle(); i++ )
      tab[i].svuota();  //funzione che cancella tutti i nodi nella lista
      nEl = 0;
}



template<class T>
int TabellaHash<T>::n() const
{     return nEl; }

template<class T>
bool TabellaHash<T>::cerca ( T& a )
{
      int i = (ch.hash(a) % nCelle())+1;
      Iteratore<T> it(tab[i]);
      bool trovato=false;
      it.VaiInTesta();
      while (!it.isNull() && !trovato)
            if (ch.equivalenti(it.getCurrentValue(),a))
```

```
            {
                  a=it.getCurrentValue();
                  trovato=true;
            }
            else
                  it.MoveNext();
      return trovato;
}
```

Esempio 1. Consideriamo la solita classe *Dipendente* in cui la chiave è codice. Vogliamo costruire una tabella hash con 100 entrate avendo come chiave il *codice* che è un intero.

```
#include "dipendente.h"
#include  "tabhash.h"
int toInt (const Dipendente& d ) {return d.codice;}
class HashDipC : public ChiaveHash<Dipendente>
{     public:
      int hash( const Dipendente& d ) {   return d.codice;  }
      bool equivalenti(const Dipendente& d1, const Dipendente& d2)
      { return (d1.codice==d2.codice);}
};

HashDipC chd;
TabellaHash<Dipendente> tabDip(100, chd);
```

Costruiamo ora una tabella hash per *Dipendente* con chiave *nome*, nell'ipotesi che non esistono due dipendenti con lo stesso nome:

```
#include "dipendente.h"
#include  "tabhash.h"
class HashDipN : public ChiaveHash<Dipendente>
{     public:
      int hash( const Dipendente& d ) {   return stringaHash(d.nome);   }
      bool equivalenti(const Dipendente& d1, const Dipendente& d2)
      { return (strcmp(d1.nome,d2.nome)==0);}
};

void main(){
      HashDipN hdNome;
      TabellaHash<Dipendente> tabDip1(100,hdNome);
      Dipendente d1;
      cin>>d1;
      tabDip1.ins(d1);
      Dipendente ricercato;
      cin>>ricercato.nome;
      if tabDip1.cerca(ricercato) cout<<ricercato;

}
```

Analizziamo ora la complessità delle funzioni della tabella hash. Assumiamo con ragionevolezza che la funzione *chiaveHash* abbia complessità costante in quanto la dimensione di *T* è in generale trascurabile rispetto alla dimensione della tabella. La funzione *svuota* ha complessità $\Theta(n)$; inoltre le funzioni *cerca*, *ins*, e *canc* hanno complessità nel caso migliore $\Theta(1)$ e nel caso peggiore $\Theta(n)$. E' possibile effettuare un'analisi di caso medio nell'ipotesi che la funzione hash restituisca un qualsiasi indice con la stessa probabilità $1/n$. Di seguito riportiamo i numeri medi di accessi (*na*) in una lista necessari per determinare la chiave desiderata, uno per ogni possibile *fattore di caricamento* indicato come $fc = n/nCelle$:

fc	0,2	0,5	0,7	0,9	1,0	1,1	1,5	2,0	3,0
na	1,10	1,25	1,35	1,45	1,5	1,55	1,75	2,00	2,5

Tali numeri valgono nel caso di ricerca con successo; in apparenza stranamente, per il caso di ricerca con insuccesso e $fc \leq 1$, i numeri medi di accessi sono leggermente inferiori. Ciò dipende dal fatto che in questo caso sono valutati anche i casi in cui la lista dei collidenti sia vuota; ovviamente, per *fc* più elevati la probabilità di trovare una lista vuota si riduce praticamente a zero e, quindi, i numeri medi di accessi diventano maggiori che nel caso di ricerca con successo.

Dalle tabelle si può rilevare che per fattori di caricamento contenuti (inferiore a 0,9) il numero di accessi è inferiore a 1,5, cioè possiamo affermare che le funzioni *cerca*, *ins* e *canc* hanno complessità media costante. Quando la tabella degrada, cioè *fc* diventa elevato conviene allargare la tabella per mantenere il numero di accessi praticamente uguale a 1.