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 cardinale, cioè un intero non negativo minore di 4.294.967.296. 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>
#include <limits>

typedef unsigned long int card; // tipo cardinale

card stringaHash( char * s )
{     long int nMax = numeric_limits<long int>::max(); //4294967295 di solito
      card C = 0;
      for ( card 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. Ovviamente supponiamo che *p* sia minore di 4.294.967.296.

```
inline card moduloHash( card k, card p )
{     return (k % p) + 1;
}
```

Ovviamente anche questa funzione ha complessità costante.

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*. L'operatore "card" di conversione a cardinale deve essere sempre definito.

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.

Si riporta di seguito un'implementazione proprio della classe TabellaHash.

```
#include <algorithm>
#include <list>
#include <vector>
using namespace std;

// ChiaveHash si occupa dell'hash degli oggetti che conterrà la HashTable
template <typename T>
class ChiaveHash {
public:
      ChiaveHash () {}
            // calcola l'hash relativo all'istanza di tipo T
      virtual unsigned long hash (const T& a) = 0;
            // ritorna true se le istanze a1 e a2 hanno lo stesso hash
      virtual bool equivalenti (const T& a1, const T& a2) = 0;
};


// questa è la classe che implementa la tabella hash
// il tipo T indica il tipo di oggetti che conterrà l'hashtable
template <typename T>
class HashTable {
protected:
       // istanza di ChiaveHash
       // deve essere definita ed inizializzata all'esterno
    ChiaveHash<T>& key;
       // elementi totali inseriti nella HashTable
    unsigned long nElementi;

        // tabella: ogni cella è una lista di T
       // ogni cella del vector è un bucket e ogni cella è indicizzata da un indice ricavato dall'hash
       // ogni lista permette di salvare tutti gli elementi e di gestire le collisioni
       // esempio: se in una lista ho due oggetti, allora questi sono in collisione (=> hanno lo stesso hash)
    vector<list<T> > table;

public:
       // costruttore: size_it => long int
       // tableSize => il numero di bucket (=> la dimensione del vector table)
       // k => istanza della ChiaveHash<T>
    HashTable(const size_t& tableSize, ChiaveHash<T>& k):
                              table(tableSize), key(k), nElementi(0) {};
```

```
   // indica qual è la dimensione di table => quanti bucket ho dentro la hashtable
size_t nCelle() const { return table.size(); };
unsigned n() const { return nElementi; };

   // Inserisce l'elemento a nella hashtable se non già presente
   // Step per inserire l'elemento a dentro alla hashtable:
   // 1. identificare il bucket in cui inserire a => fare l'hash di a e poi prendere l'indice relativo all'hash
   // 2. inserire a nel bucket corrispondente gestendo le collisioni
bool ins(T& a) {
    // step 1
    unsigned long i = key.hash(a) % this->nCelle();
                                      // i in [0, table.size()-1]

    bool found = this->cerca(a);
    if (!found) {
        // step 2
        table.at(i).push_back(a);
        this->nElementi++;
        return true;
    }

    return false;
};

   // Rimuove l'elemento a dalla hashtable se presente
bool canc(T& a) {
   // in quale bucket sta a dentro la tabella hash?
    unsigned long i = key.hash(a) % this->nCelle();

   // return true se a è equivalente con una istanza element
    auto compare = [a, this](T& element)
              { return this->key.equivalenti(element, a); };
   // itero su table.at(i) e rimuovo le istanze che sono equivalenti ad a
    auto removed = std::remove_if(table.at(i).begin(), table.at(i).end(), compare);

    if (removed) {
        this->nElementi--; // aggiorno il contatore di tutti gli elementi
        return true;
    }

   // a non sta dentro la hashtable => non cancello alcunché
    return false;
};

   // Cerca l'elemento a (sfruttando il valore indicizzato) nella hashtable.
   // Se presente, l'istanza a (esterna) conterrà l'istanza trovata (interna).
bool cerca(T& a) {
    unsigned long i = key.hash(a) % this->nCelle();

   // lambda function <-- una funzione anonima
 auto compare = [a, this](T& element) { return this->key.equivalenti(element, a); };

   // found è un puntatore ad un elemento dentro table.at(i)
   // se esiste una istanza equivalente ad a dentro alla hashtable
```

```
     auto found = std::find_if(table.at(i).begin(), table.at(i).end(), compare);

        if (found != table.at(i).end()) {
            // dentro a metto il valore di found
            a = *found;
            return true;
        }

        return false;
    };

      // Rimuove tutti gli elementi presenti nella hashtable
    void svuota() {
      // itero su ogni vettore, e rimuovo tutti gli elementi all'interno del vettore.
        for (auto& x: table)
            x.clear();
        this->nElementi = 0;
    };
};
```

Definiamo ora la classe VoceRubrica, su cui costruremo un'Hash Table:

```
#include <iostream>
#include <string>
using namespace std;

class VoceRubrica {
private:
    string cognome;
    string nome;
    string numeroTelefonico;

public:
    VoceRubrica() {};
    VoceRubrica(const string& c, const string& n, const string& nt) :
        cognome(c), nome(n), numeroTelefonico(nt) {};

    const string& getCognome() const { return this->cognome; };
    const string& getNome() const { return this->nome; };
    const string& getNumeroTelefonico()const { return this->numeroTelefonico; };

    void setCognome(const string& cognome) { this->cognome = cognome; };
    void setNome(const string& nome) { this->nome = nome; };
    void setNumeroTelefonico(const string& numeroTelefonico)
                { this->numeroTelefonico = numeroTelefonico; };

    // cout << voceRubrica
    friend ostream& operator<<(ostream& out, const VoceRubrica& vr) {
        out << vr.cognome << ", " << vr.nome << ", " << vr.numeroTelefonico;
        return out;
    }

    // cin >> voceRubrica
    friend istream& operator>>(istream& in, VoceRubrica& vr) {
        VoceRubrica vrInterna;
        string cognome, nome, numeroTelefonico;
```

```
        in >> cognome;
        in >> nome;
        in >> numeroTelefonico;

        vrInterna.setCognome(cognome);
        vrInterna.setNome(nome);
        vrInterna.setNumeroTelefonico(numeroTelefonico);

        vr = vrInterna;

        return in;
    }
};
```

Definiamo infine, due main di prova:

// (1) Creare un main che definisca una tabella hash di voci di rubrica,
// (2) legga da standard input 4 voci ed infine,
// (3) dato un cognome stampi in output la voce di rubrica relativa a quel cognome (si assuma che il cognome sia un campo chiave, quindi non potranno essere inserite due voci aventi lo stesso cognome)

```
#include <iostream>
#include <limits>
#include <vector>
#include "HashTable.hpp"
#include "VoceRubrica.h"
using namespace std;

// funzione che calcola un numero che identifica univocamente la stringa
// somma il valore numerico di ogni carattere nella stringa s (modulo il limite superiore per i long int)
unsigned long string2hash( const string s )
{
      long int nMax = numeric_limits<long int>::max();
      unsigned C = 0;
      for (unsigned i = 1; i <= s.size(); i++ )
            C = (C + s[i]) % nMax;
      return C;
}

// Chiave Hash di VoceRubrica da calcolare sul cognome
class HashVoceRubricaCognome : public ChiaveHash<VoceRubrica> {
public:
  // l'hash è da calcolare sul cognome
    unsigned long hash(const VoceRubrica& vr) {
    // l'hash per una istanza di vocerubrica è rappresentato dall'hash della stringa contenente il cognome
        return string2hash(vr.getCognome());
    }
```

```
  // due voci di rubrica sono equivalenti quando il loro cognome è uguale
  // quindi due istanze sono equivalenti quando il campo su cui si calcola l'hash è lo stesso
    bool equivalenti(const VoceRubrica& vr1, const VoceRubrica& vr2) {
        return vr1.getCognome() == vr2.getCognome();
    }
};


// c++11
using CoppiaSI = pair<string, unsigned>; // .first => il primo elemento della coppia;
                                         // .second => il secondo elemento della coppia
//typedef pair<string, unsigned> CoppiaSI;

// Chiave Hash di CoppiaSI da calcolare sul primo valore della coppia
class HashCoppiaSI : public ChiaveHash<CoppiaSI> {
    unsigned long hash(const CoppiaSI& c) { return string2hash(c.first); }
    bool equivalenti(const CoppiaSI& c1, const CoppiaSI& c2)
                                         { return c1.first == c2.first; }
};



int main() {

  // (1)
    HashVoceRubricaCognome key;
    HashTable<VoceRubrica> ht(10, key);

  // (2)
    for (unsigned i = 0; i < 4; ++i) {
        cout << "Inserisci una VoceRubrica:" << '\n';
        VoceRubrica vr;
        cin >> vr;
        ht.ins(vr);
    }

    cout << "Numero di elementi nella hashtable: " << ht.n() << '\n';

  // (3)
    string cognome;
    cout << "Cognome da cercare: ";
    cin >> cognome;
    VoceRubrica daCercare;
    daCercare.setCognome(cognome);
    if (ht.cerca(daCercare)) {
                  // esiste una istanza di VoceRubrica dentro ht che ha il cognome pari a cognome
        cout << "Trovato!" << '\n' << daCercare << '\n';
    } else {
        cout << "Non trovato :(" << '\n';
    }

    return 0;
}
```

// (1) Creare un main che legga da standard input 4 voci di rubrica e le memorizzi in un vettore.
// (2) Definire due tabelle hash (di conseguenza le opportune classi derivate da ChiaveHash):
// –la prima che memorizza coppie costituite da: cognome, indice del vettore contenente la voce di rubrica associata a quel cognome
// –la seconda che memorizza coppie costituite da: numeroTelefonico, indice del vettore contenente la voce di rubrica associata a quel numero telefonico.
// (3) esempio di ricerca dentro le tabelle hash

```
int main() {
  // (1)
    vector<VoceRubrica> v(4);
    for (unsigned i = 0; i < 4; ++i) {
        cout << "Inserisci una VoceRubrica:" << '\n';
        VoceRubrica vr;
        cin >> vr;
        v[i] = vr;
    }

  // (2) poiché le tabelle hash richieste devono memorizzare delle coppie, abbiamo bisogno di rappresentare queste coppie
  // usiamo un pair<string, unsigned>
    HashCoppiaSI key;
    HashTable<CoppiaSI> ht1(10, key);
    HashTable<CoppiaSI> ht2(10, key);
  // itero su tutte le vocirubrica, creo ed inserisco le coppie nelle rispettive hashtable
    for (unsigned i = 0; i < v.size(); ++i) {
        CoppiaSI c1(v[i].getCognome(), i);          // la coppia che inserirò in ht1
        CoppiaSI c2(v[i].getNumeroTelefonico(), i); // la coppia che inserirò in ht2

        ht1.ins(c1);
        ht2.ins(c2);
    }

  // (3)
    string cognome, numeroTelefonico;
    cout << "Cognome da cercare: "; cin >> cognome;
    cout << "Numero telefonico da cercare: "; cin >> numeroTelefonico;

    CoppiaSI temp;

    temp.first = cognome;
    cout << "Ricerca per cognome: " << cognome << '\n';
    if (ht1.cerca(temp)) {
        cout << "Trovato in ht1!" << '\n' << v[temp.second] << '\n';
                    // temp.second è l'indice della voce rubrica in v
    } else {
        cout << "Non trovato in ht1! :(" << '\n';
    }
    if (ht2.cerca(temp)) {
        cout << "Trovato in ht2!" << '\n' << v[temp.second] << '\n';
```

```
    } else {
        cout << "Non trovato in ht2! :(" << '\n';
    }

    temp.first = numeroTelefonico;
    cout << "Ricerca per numeroTelefonico: " << numeroTelefonico << '\n';
    if (ht1.cerca(temp)) {
        cout << "Trovato in ht1!" << '\n' << v[temp.second] << '\n';
    } else {
        cout << "Non trovato in ht1! :(" << '\n';
    }
    if (ht2.cerca(temp)) {
        cout << "Trovato in ht2!" << '\n' << v[temp.second] << '\n';
    } else {
        cout << "Non trovato in ht2! :(" << '\n';
    }

    return 0;
}
```

LE TABELLE HASH IN STL SI CHIAMANO unordered_map

Le unordered_map sono **contenitori associativi** che immagazzinano elementi formati dalla combinazione di **una coppia <chiave, valore>,** e che permettono il recupero veloce di elementi individuali basati sulle loro chiavi.
In una unordered_map, il valore chiave è generalmente usato per identificare in modo univoco l'elemento, mentre il valore mappato è un oggetto con il contenuto associato a questa chiave. I tipi di chiave e di valore mappato possono essere diversi.
Internamente, gli elementi in unordered_map non sono ordinati in alcun ordine particolare rispetto alla loro chiave o ai loro valori mappati, ma organizzati in *buckets* in base ai loro valori di hash per consentire un accesso veloce ai singoli elementi direttamente dai loro valori chiave (**con una complessità temporale media costante**).
I contenitori unordered_map **sono più veloci dei contenitori map** per accedere ai singoli elementi in base alla loro chiave, anche se sono generalmente **meno efficienti per l'iterazione all'interno di un range** che definisce un sottoinsieme dei loro elementi.
Le mappe non ordinate **implementano l'operatore di accesso diretto (operator[])** che permette l'accesso diretto al valore mappato usando il suo valore chiave come argomento.

Di seguito un semplice esempio d'uso delle unordered_map:

```
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
  std::unordered_map<std::string,std::string> mymap;

  mymap["Bakery"]="Barbara";  // new element inserted
  mymap["Seafood"]="Lisa";    // new element inserted
  mymap["Produce"]="John";    // new element inserted

  std::string name = mymap["Bakery"];   // existing element accessed (read)
  mymap["Seafood"] = name;              // existing element accessed (written)

  mymap["Bakery"] = mymap["Produce"];   // existing elements accessed
                                       //(read/written)

  name = mymap["Deli"];    // non-existing element: new element "Deli" inserted!

  mymap["Produce"] = mymap["Gifts"];    // new element "Gifts" inserted,
                                       //"Produce" written

  for (auto& x: mymap) {
    std::cout << x.first << ": " << x.second << std::endl;
  }

  return 0;
}

OUTPUT:
Gifts:
Deli:
Produce:
Seafood: Barbara
Bakery: John
```

N.B. In questo esempio il “valore” è una stringa (Barbara, Lisa, John…) ma potrebbe tranquillamente essere una VoceRubrica, mentre la chiave potrebbe essere il cognome, o il numero di telefono.
Al suo interno, la classe unordered_map conserva la coppia <chiave,valore>.
Nella costruzione della mappa, è possibile in linea di principio specificare una funzione hash personalizzata. Tuttavia, nella maggior parte dei casi è conveniente lasciare quella di default.