## L10. TABELLE HASH

Una **tabella hash** è una struttura dati usata per mettere in corrispondenza una data chiave con un dato valore, sia la chiave che il valore possono appartenere a diversi tipi primitivi o strutturati. Ad esempio, associare l'età, che è un intero, ad una persona utilizzando il suo nome, che è una stringa. Il linguaggio C non fornisce un supporto diretto per le tabelle hash, per surrogarle possiamo utilizzare un array, a patto però che si associno prima gli indici interi alle persone e poi l'età agli indici.

Per creare tabelle hash dobbiamo memorizzare delle coppie chiave-valore, chiamate **entry**, implementeremo tre operatori di base:

- INSERT(key, value): Inserisce un elemento nuovo, con un certo valore (unico) di un campo chiave, cioè una nuova coppia chiave-valore;
- SEARCH(key): Determina se un elemento con un certo valore della chiave esiste, se esiste, lo restituisce;
- DELETE(key): Elimina l'elemento identificato dal campo chiave, se esiste, cioè forniamo la chiave associata a quella entry.

Oltre a questa funzione di base è possibile supportare una serie di altre funzioni come l'ordinamento dell'insieme dei dati oppure la ricerca del massimo o del successore e così via. Alcune definizioni:

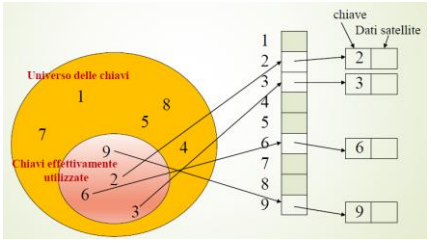
- **U** - indichiamo l'**universo di tutte le possibili chiavi**;
- **K** - l'insieme delle **chiavi effettivamente memorizzate**.

**CHIAVI INTERE – INDIRIZZAMENTO DIRETTO:**

Se l'universo delle chiavi è piccolo e le chiavi sono intere, allora possiamo utilizzare una **tabella ad indirizzamento diretto**, che è un array.

Nell'array avremo quindi che ad una data chiave intera corrisponde una posizione, detta anche **slot**, all'interno della tabella. Come sappiamo possiamo accedere in tempo costante a ciascuna posizione all'interno della tabella data la chiave, perché il meccanismo di **indirizzamento** è esattamente quello dell'array.

Quando l'universo delle chiavi è piccolo, come gli interi che vanno da 1 a 9, soltanto una parte di queste chiavi sono effettivamente utilizzate e quello che possiamo fare è usare la chiave come indice.
Per esempio, preso l'elemento con chiave 3 lo andiamo a memorizzare all'interno dell'elemento 3 dell'array e questo elemento 3 avrà un puntatore al dato che è che una coppia chiave-valore, cioè una entry all'interno della tabella.
Conviene utilizzare questo schema quando l'universo delle chiavi è piccolo e solo una parte significativa di questo universo delle chiavi viene effettivamente utilizzato.




Ma se le chiavi non sono intere oppure se l'universo delle possibili chiavi è molto grande, allora diventa non più conveniente o impossibile utilizzare questo metodo delle tabelle ad indirizzamento diretto. Se l'universo delle chiavi è troppo grande, può non essere possibile avere una tabella con tutte queste righe oppure se comunque il numero di chiavi utilizzate è molto minore rispetto alla taglia dell'universo delle chiavi, avremo delle tabelle con pochi dati, quasi vuote, ed avremo quindi uno spreco di spazio inaccettabile.

Ricapitolando abbiamo due casi:

- Se **|K|~|U|**, avremo che la taglia delle chiavi effettivamente utilizzate è simile alla taglia dell'universo delle chiavi:
  In questo caso possiamo utilizzare una tabella ad **indirizzamento diretto** perché non sprechiamo molto spazio e abbiamo il vantaggio che le operazioni possono essere fatte in *tempo costante*.
- Se **|K|<<|U|**, avremo che la taglia delle chiavi effettivamente utilizzate è molto minore rispetto alla taglia dell'universo delle chiavi:
  La soluzione della tabella indirizzamento diretto non è più praticabile, ad esempio, se volessimo memorizzare in una tabella ad indirizzamento diretto il record degli studenti utilizzando la matricola come chiave, quello che succede è che se la matricola a 6 cifre l'array deve avere spazio per contenere $10^6$ elementi. Supponiamo che gli studenti siano 30, lo spazio realmente occupato dalle chiavi memorizzate è $30/10^6$ = 0,003% dello spazio effettivamente allocato.

Per conservare l'efficienza nelle operazioni ma avere anche un compromesso riguardante i requisiti di memoria, quindi utilizzare una memoria poco più grande rispetto a quella effettivamente utilizzata, ricorriamo alle tabelle hash.

La differenza sostanziale tra il **metodo di indirizzamento diretto** e le **tabelle hash**, consiste nel fatto che mentre il primo memorizza nella tabella in posizione k un elemento con chiave k, nel metodo hash invece un elemento con chiave k viene memorizzato nella tabella in posizione h(k) dove h è una funzione detta **funzione hash**, il cui scopo è di definire una corrispondenza tra l'universo U delle chiavi e le posizioni di una tabella hash T[0…m-1], con m minore rispetto alla taglia dell'universo delle chiavi. Questa funzione h è definita come h:U→{0, 1, …m-1}.

**GESTIONE DELLE COLLISIONI:**

La **funzione hash** ci consente di ricavare l'indice in cui andare a mettere il valore corrispondente ad una data chiave. L'inconveniente è che la funzione hash non può essere ***iniettiva***, cioè due chiavi distinte possono produrre lo stesso valore hash, quando questo avviene abbiamo due chiavi $k_i \neq k_j$ e avviene che $h(k_i)=h(k_j)$, diremo che si è verificata una **collisione**.

Per avere una struttura dati efficiente occorre minimizzare il numero di collisioni e quindi ottimizzando la **funzione hash**, però comunque gestire le collisioni che comunque possono avvenire, cioè quando avviene una collisione deve essere possibile inserire più elementi all'interno della stessa locazione oppure trovare un modo per inserire nella tabella entry che hanno una stessa chiave.

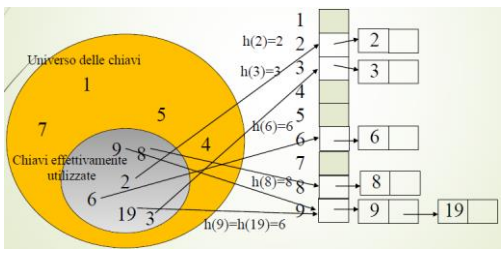
Esistono due metodi principali per **risolvere le collisioni**:

- **Metodo di concatenazione**;
- **Metodo di indirizzamento aperto**.

**METODO DI CONCATENAZIONE:**

Sfrutta l'idea di mettere tutti gli elementi che collidono in una lista concatenata, quindi la tabella avrà nella posizione j-esima una lista che contiene tutti gli elementi associati a quel dato valore hash e un puntatore nullo se non ci sono elementi. L'implementazione di questo metodo realizzerà la tabella hash come un vettore di liste.

Supponiamo di avere un universo delle chiavi che corrisponde ai numeri interi e poche chiavi effettivamente utilizzate, abbiamo una tabella che è un array indicizzato da 1 a 9, a ciascuna locazione è associata una lista, in particolare nella locazione 9 abbiamo due elementi con chiavi diverse, 9 e 19, però la cui funzione hash associa lo stesso valore, ovvero 6, quindi due elementi sono inseriti nella stessa locazione utilizzando una lista.



**FUNZIONI HASH**:

Come facciamo, dato un qualunque valore, ad ottenere l'indice con cui andare ad indicizzare la entry nella tabella. Una caratteristica importante è che deve avere una funzione hash è il cosiddetto **criterio di uniformità semplice** che dice che il valore hash di una chiave k è uno dei valori 0…m-1 in modo equiprobabile, questo ci consente di minimizzare le collisioni.

Un altro requisito è che una buona funzione hash dovrebbe usare tutte le cifre della chiave per produrre un valore hash.

Una **funzione hash** molto semplice, ma anche molto utilizzata, quando abbiamo a che fare con chiavi a **valori interi** è il **metodo di divisione**, la funzione hash è del tipo: h(k)=k mod m, cioè il valore hash è il resto della divisione di k per m. Questo metodo è molto veloce e può essere calcolata con una semplice operazione.

Invece una **funzione hash** per il **tipo stringa**, per convertire la stringa in un numero naturale che poi è l'indice della tabella, consideriamo la stringa come un numero in base 128, infatti questo numero è il numero di simboli diversi per ogni cifra di una stringa.

Esiste una codifica che associa un simbolo ad un numero naturale ad esempio la codifica ASCII, la conversione viene fatta nel modo tradizionale cioè prendendo ogni carattere che compone la stringa e moltiplicando per 128 elevato alla posizione, per esempio se dobbiamo convertire in numero la stringa "pt" faremo "p"*$128^1$ + "t"*$128^0$ che verrà 14452.

Questo tipo di conversione può funzionare bene per stringhe molto corte, ma già se abbiamo una stringa un po' più lunga la cosa diventa non fattibile perché il numero che otteniamo comincia a diventare troppo grande per poter essere rappresentato. Conviene avere una funzione hash modulare che trasforma un pezzo di chiave alla volta, sfruttiamo il fatto che l'equazione, con i vari prodotti per 128 elevato alla posizione, può essere riscritta in modo più conveniente. Questo modo è dato dalla **regola di Horner** che ci consente di ridurre notevolmente il numero di moltiplicazioni che dobbiamo fare per valutare la nostra espressione.

((((((((((97*128 + 118)*128 + 101)*128 + 114)*128+121)*128 + 108)*128 + 111)*128 + 110)*128 + 103)*128 + 107)*128 + 101)*128 + 12).

Come possiamo vedere nella nuova espressione non abbiamo più l'elevamento a potenza ma abbiamo delle semplici moltiplicazioni alternate a delle addizioni.

```
int hash(char*v, int m){
  int h = 0, a = 128;
  for (; *v != '\0'; v++)
    h = (h*a + *v) % m;
  return h;
}
```

La funzione restituisce un valore intero che è l'indice, prende come parametro una stringa v e un intero m che è la taglia della tabella. Inizializza un valore h=0 e a=128, il valore h sarà il valore che ritorneremo come indice. Scorriamo l'intera stringa e andiamo ad inserire in h il valore di questa espressione, h*a + il valore ASCII della stringa, il tutto modulo m.

In ogni iterata facciamo un'operazione di modulo che ci consente di mantenere basso il numero che dobbiamo restituire. Questo calcolo ci consente di evitare l'overflow degli interi.

**METODO DI INDIRIZZAMENTO APERTO:**

In questo metodo non utilizziamo più una lista per ogni cella ma utilizziamo tutta la tabella, quindi memorizziamo gli elementi sempre all'interno della tabella ma in una locazione differente se quella iniziale risulta essere occupata.

Il metodo con cui troviamo una nuova locazione in cui andare a salvare un elemento è quello di generare un nuovo valore hash fino a trovare una nuova posizione vuota in cui inserire l'elemento, più nel dettaglio estendiamo la nostra funzione hash perché generi non solo un valore hash ma una **sequenza di scansione**.

Quindi la nostra funzione hash diventa una funzione di due variabili, una prima variabile presa dall'universo delle chiavi, una seconda variabile intera che vada 0 a m-1 dove m è la dimensione della tabella e il risultato è un valore compresi tra 0 ed m-1, h : U x {0,1,…,m-1}→{0,1,…,m-1}, cioè prenda in ingresso una chiave e un indice di posizione e generi una nuova posizione.

La nostra funzione hash non genera più un singolo indice ma genera una **sequenza di scansione**, data una chiave k.

Per prima cosa si genera un indice con h(k, 0), se questo indice generato corrisponde ad una posizione già occupata, avremmo avuto quindi una collisione, genereremo un secondo indice utilizzando h(k, 1), in caso di ulteriori collisioni genereremo h(k, 2) e così via con h(k, i).

I valori generati dalla funzione hash sono una sequenza <h(k, 0), …h(k, m-1)>.

INSERIMENTO

```
Hash-Insert(T,k)
1 i ← 0
2 repeat j ← h(k,i)
3  if T[j]=NIL
4  then  T[j] ← k
5     return j
6  else i ← i+1
7 until i=m
8 error "overflow"
```

RICERCA

```
Hash-Search(T,k)
1 i ← 0
2 repeat j ← h(k,i)
3  if T[j]=k
4  then return j
6  i ← i+1
7 until i=m o T[j]=NIL
8 return NIL
```

ELIMINAZIONE

Questa operazione risulta essere abbastanza difficile. Il motivo è che non possiamo marcare una posizione vuota con un valore Nullo, perché comporterebbe il fallimento del codice della funzione di ricerca, questa viene interrotta quando si trova un valore Nullo perché vuol dire che la chiave non è stata trovata, ma è possibile che sia stata inserita in una posizione di scansione successiva, quindi il valore Nullo non può essere utilizzato come marcatore per indicare un elemento cancellato. Se invece si vuole supportare la cancellazione bisogna utilizzare uno speciale marcatore. Una posizione cancellata si può poi sovrascrivere quando viene effettuato un inserimento.

**CARATTERISTICHE DI h**:

Una proprietà desiderabile delle funzioni hash è quella di **uniformità semplice** riferita in particolare alla proprietà di una buona funzione hash di poter generare con la stessa probabilità tutti i valori interi dell'intervallo compreso nel range della tabella. Estendiamo tale concetto anche per le funzioni hash utilizzate nello schema ad **indirizzamento aperto**, in particolare la proprietà di uniformità della funzione hash prevede che per ogni chiave k, la sequenza di scansione generata da h, deve essere una qualunque delle m fattoriale permutazioni degli interi compresi tra 0 ed m-1, quindi per essere ideale la nostra funzione h deve poter generare tutti i valori del range della tabella una volta sola.

Purtroppo, rispettare questo criterio di uniformità per una funzione hash è molto difficile, quindi si ricorre a delle approssimazioni. Ci sono diversi schemi per poter realizzare funzioni che hanno una buona approssimazione di questa proprietà di uniformità. In particolare, abbiamo:

- **Scansione lineare;**
- **Scansione quadratica;**
- **Hashing doppio**.

Queste funzioni riescono a generare una permutazione ma non riescono a generare tutte le m fattoriale permutazioni.

Data una funzione h': U →{0, 1, …m-1}, il metodo di scansione lineare costruisce una h(k,i)=(h'(k)+i) mod m, questo equivale a generare tutti gli indici di posizione consecutivamente, per prima cosa si genera la posizione h'(k), quindi la posizione h'(k)+1, e così via fino alla posizione m-1, poi si scandisce in modo circolare la posizione 0,1,2 fino a tornare a h'(k)-1. Abbiamo quindi generato tutte le posizioni a partire da una data posizione e ciclando circolarmente su tutta la tabella.

## ADT Key

| Sintattica | Semantica |
|---|---|
| Nome del tipo: Key<br>Tipi usati: Int, boolean | Dominio: $k \in U$ |
| equals(Key, Key) → boolean | equals(k1, k2) → b<br>• Post: b = true se k1=k2; b = false altrimenti |
| hashValue(Key, int) → int | hashValue(k, size) → index<br>• Pre: k != nil, size > 0<br>• Post: 0 <= index < size |
| inputKey() → Key<br>outputKey(Key) | |

## ADT Entry

| Sintattica | Semantica |
|---|---|
| Nome del tipo: Entry<br>Tipi usati: Item, Key | Dominio: Coppia (chiave, valore)<br>chiave è di tipo Key, valore è di tipo Item |
| newEntry(Key, Item) → Entry | newEntry(key, value) → e<br>• Post: e = (key, value) |
| getKey(Entry) → Key | getKey(e) → key<br>• Post: e = (key, value) |
| getValue(Entry) → Item | getValue(e) → value<br>• Post: e = (key, value) |

## ADT HashTable

| Sintattica | Semantica |
|---|---|
| Nome del tipo: Hashtable<br>Tipi usati: Entry, boolean, Key | Dominio: insieme di elem. T={a1,...,an} di tipo Entry |
| newHashtable() → Hashtable | newHashtable() → t<br>• Post: t = { } |
| insertHash(Hashtable, Entry) → Hashtable | insertHash(t, e) → t'<br>• Post: t = {a1, a2, ... an}, t' = {a1, a2, ..., e,..., an} |
| searchHash(Hashtable, Key) → Entry | searchHash(t, k) → e<br>• Pre: t = <a1, a2, ..., an> n>0<br>• Post: e = ai con $1 \le i \le n$ se ai(key) = k |
| deleteHash(Hashtable, Key) → Hashtable | deleteHash(t, k) → t'<br>• Pre: t = {a1, a2, ..., an} n>0, $a_i$(key) = k, $1 \le i \le n$<br>• Post: t' = <$a_1$, $a_2$, ..., $a_{i-1}$, $a_{i+1}$, ..., $a_n$> |

*key.h*

```
typedef void *Key;

int equals(Key, Key);
int hashValue(Key, int);
Key inputKey();
void outputKey(Key);
```

*key-string.c*

```
#include "key.h"
#define SIZE 30

int equals(Key k1, Key k2){
  char *chiave1, *chiave2;
  chiave1=k1;
  chiave2=k2;
  return (strcmp(chiave1, chiave2)==0);
}

int hashValue(Key k, int size){
  int h=0, a=128;
  char *v=k;
  for(;*v!='\0'; v++)
    h=(h*a + *v)%size;
  return h;
}

Key inputKey(){
  char *k=malloc(SIZE*sizeof(char));
  scanf("%s", k);
  return k;
}

void outputKey(Key k){
  char *v=k;
  printf("%s ", v);
}
```

*entry.h*

```
#include "key.h"
#include "item.h"
typedef struct entry *Entry;

Entry newEntry(Key, Item);
Key getKey(Entry);
Item getValue(Entry);
```

*entry.c*

```
#include "entry.h"
#include "key.h"
#include "item.h"

struct entry{
  Key key;
  Item value;
};

Entry newEntry(Key k, Item value){
  Entry e = malloc(sizeof(struct entry));
  e->key=k;
  e->value=value;
  return e;
}

Key getKey(Entry e){
  if(e==NULL)     return NULL;
  return (e->key);
}

Item getValue(Entry e){
  if(e==NULL)     return NULL;
  return (e->value);
}
```

*hashtable.h*

```
#include "entry.h"
#include "key.h"
typedef struct hashtable *HashTable;

HashTable newHashtable(int);
int insertHash(HashTable, Entry);
Entry searchHash(HashTable, Key);
Entry deleteHash(HashTable, Key);
```

*hashtable.c*

```
#include "hashtable.h"
#include "list.h"
#include "entry.h"
#include "key.h"
#define N 20
struct hashtable{
  int size;
  List *entries;
};

HashTable newHashtable(int size){
  HashTable h = malloc(sizeof(struct hashtable));
  h->size=size;
  h->entries= malloc(size*sizeof(List));
  for(int i=0; i<size; i++)
    h->entries[i]=newList();
  return h;
}

int insertHash(HashTable h, Entry e){
  if(h==NULL)     return 0;
  Key k = getKey(e);
  int index= hashValue (k, h->size);
  addHead(h->entries[index], e);
  return 1;
}

Entry searchHash(HashTable h, Key k){
  if(h==NULL)     return NULL;
  int index= hashValue (k, h->size);
  Entry e = newEntry(k, NULL);
  int pos;
  return searchList(h->entries[index], e, &pos);
}

Entry deleteHash(HashTable h, Key k){
  if(h==NULL)     return NULL;
  int index= hashValue (k, h->size);
  Entry e = newEntry(k, NULL);
  return removeListItem (h->entries[index], e);
}
```

*item-entry.c*

```
#include "item.h"
#include "key.h"
#include "entry.h"

int cmpItem(Item item1, Item item2){
   Entry val1, val2;
   val1 = item1;
   val2 = item2;
   return !equals(getKey(val1), getKey(val2));
}
```