Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

# Dizionari: Intoduzione e tabelle hash

**Giancarlo Bongiovanni**
**Ivano Salvo**


SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide
preparate da T. Calamoneri e G. Bongiovanni
per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

# Dizionari (1)

Un ***dizionario*** è una struttura dati che permette di gestire un insieme dinamico di dati, che di norma è un *insieme totalmente ordinato*, tramite queste tre sole operazioni:

- ***insert***: si inserisce un elemento;
- ***search***: si ricerca un elemento;
- ***delete***: si elimina un elemento.

## Dizionari (2)

Fra le strutture dati che abbiamo descritto, le uniche che supportano in modo semplice (anche se non efficiente) tutte queste tre operazioni sono i vettori, le liste e le liste doppie.

Infatti:

- le code (con o senza priorità, inclusi gli heap) e le pile non consentono né la ricerca né l'eliminazione di un elemento arbitrario,
- negli **alberi**, l'**eliminazione** di un elemento comporta la disconnessione di una parte dei nodi dall'altra (cosa che può accadere anche nei grafi) e quindi è un'operazione che in genere richiede delle **successive azioni correttive**.

# Dizionari (3)

Quando l'esigenza è quella di realizzare un dizionario, ossia una struttura dati che rispetti la definizione data sopra, si ricorre quindi a soluzioni specifiche.

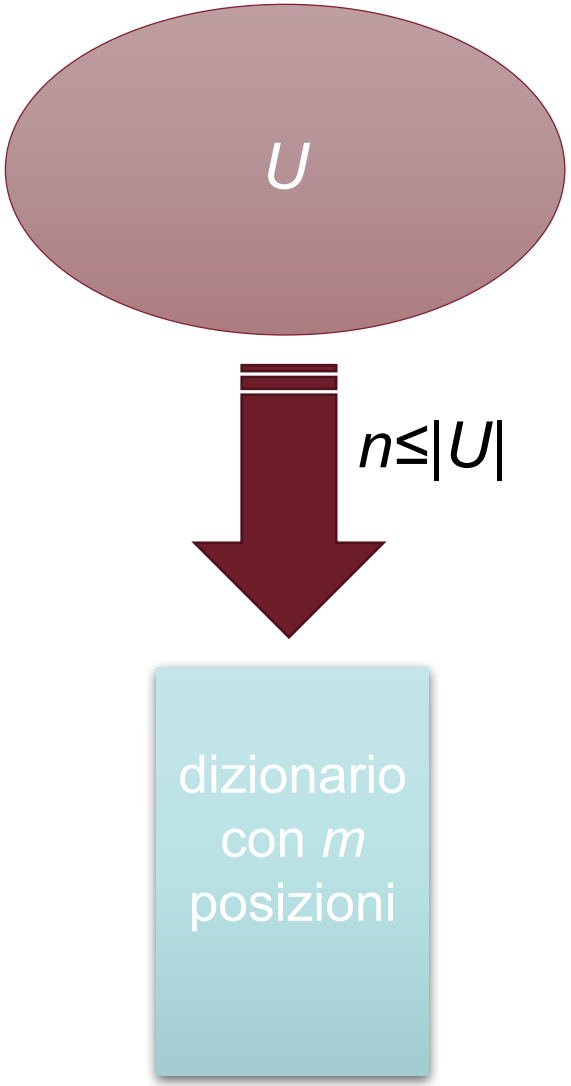
Qui ne illustreremo tre:

- tabelle ad indirizzamento diretto;
- tabelle hash;
- alberi binari di ricerca (**red-black**, nella prossima lezione).

# Dizionari (4)

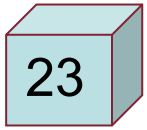
Assunzioni e nomenclatura:

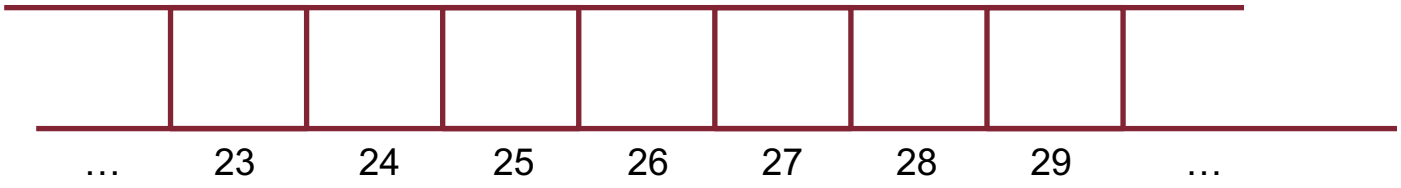
- *U*: insieme universo dei valori che le chiavi possono assumere; *U* è costituito da valori interi;
- *m*: numero delle posizioni a disposizione nella struttura dati;
- *n*: numero degli elementi da memorizzare nel dizionario; i valori delle chiavi degli elementi da memorizzare sono tutti diversi fra loro.



# Tabelle ad indirizzamento diretto (1)

E’ semplicemente un vettore nel quale ogni indice corrisponde al valore della chiave dell’elemento da memorizzare in tale posizione.



# Tabelle ad indirizzamento diretto (2)

Nell'**ipotesi $n \leq |U| = m$** un vettore $V$ di $m$ posizioni assolve perfettamente il compito di dizionario, e per giunta con grande efficienza. Infatti, tutte e tre le operazioni hanno costo computazionale $\Theta(1)$:

```
Funzione Insert_Indirizz_Diretto (V: vettore; k: chiave)
   V[k] ← dati dell'elemento di chiave k
   return

Funzione Search_Indirizz_Diretto (V: vettore; k: chiave)
   return(dati dell'elemento di V con indice k)

Funzione Delete_Indirizz_Diretto (V: vettore; k: chiave)
   V[k] ← null
   return
```

# Tabelle ad indirizzamento diretto (2)

Purtroppo le cose non sono così semplici nel caso dei problemi reali, poiché:

- l'insieme *U* può essere enorme, tanto grande da rendere impraticabile l'allocazione in memoria di un vettore V di sufficiente capienza;
- il numero delle chiavi effettivamente utilizzate può essere molto più piccolo di $|U|$: in tal caso vi è un rilevante spreco di memoria, in quanto la maggioranza delle posizioni del vettore *V* resta inutilizzata.

Perciò, si ricorre spesso a differenti implementazioni dei dizionari, a meno che non ci si trovi nelle condizioni che permettono l'uso dell'indirizzamento diretto.

# Tabelle ad indirizzamento diretto (3)

Ad esempio, si pensi al caso dei Codici Fiscali.

Un CF è costituito da 8 lettere (più una lettera di controllo) e 7 cifre. Considerando un alfabeto di 26 lettere si hanno:

$$26^8 * 10^7 \approx 2*10^{18}$$

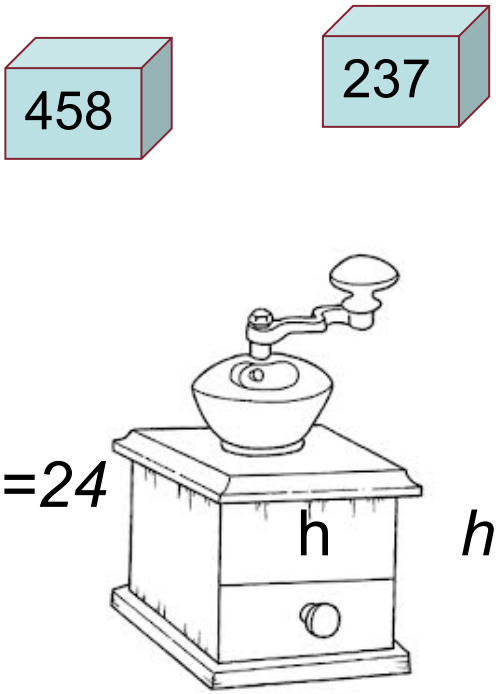
Mentre ci sono circa $6*10^7$ cittadini. Anche considerando che non tutte le lettere e le cifre vengono usate in ogni posizione, la sproporzione è enorme.

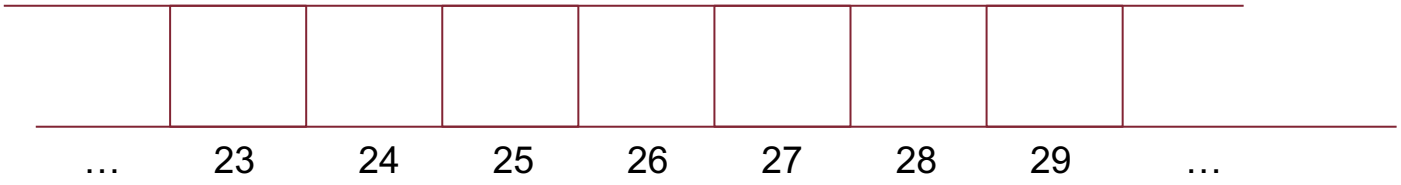
# Tabelle hash (1)

Vi si ricorre quando l'insieme *U* dei valori che le chiavi possono assumere è molto grande e l'insieme *K* delle chiavi da memorizzare effettivamente è invece molto più piccolo di *U*.

Idea: utilizzare di nuovo un vettore di *m* posizioni, ma questa volta non è possibile mettere in relazione direttamente la chiave con l'indice corrispondente, poiché le possibili chiavi sono molte di più rispetto agli indici. Allora si definisce una opportuna funzione *h*, detta ***funzione hash***, che viene utilizzata per calcolare la posizione di un elemento sulla base del valore della sua chiave.

# Tabelle hash (2)



# Tabelle hash (3)

Problema: anche se le chiavi da memorizzare sono meno di *m,* non si può escludere che due chiavi $k_1 \neq k_2$ siano tali per cui $h(k_1) = h(k_2)$, ossia che la funzione hash restituisca lo stesso valore per entrambe le chiavi, che quindi andrebbero memorizzate nella stessa posizione della tabella.

Tale situazione viene chiamata ***collisione***, ed è un fenomeno che va evitato il più possibile e, altrimenti, risolto.

# Tabelle hash (4)

Una buona funzione hash deve essere tale da rendere il più possibile *equiprobabile* il valore risultante dall'applicazione della funzione, ossia tutti i valori fra *0* ed *(m – 1)* dovrebbero essere prodotti con uguale probabilità.

In altre parole, la funzione dovrebbe far apparire come "casuale" il valore risultante, disgregando qualunque regolarità della chiave.

Inoltre, la funzione deve essere **deterministica**, ossia se applicata più volte alla **stessa chiave** deve fornire sempre lo **stesso risultato**.

La situazione ideale è quella in cui ciascuna delle *m* posizioni della tabella è scelta con la stessa probabilità, ipotesi che viene detta ***uniformità semplice della funzione hash***.

# Tabelle hash (5)

Non discuteremo qui a fondo come ottenere una funzione hash con l'ipotesi di uniformità semplice.

Partiamo, invece, dalla constatazione che le collisioni possono avvenire (anche se cerchiamo di renderle il più improbabili possibile!).

Infatti, per quanto bene sia progettata la funzione hash, è impossibile evitare del tutto le collisioni perché se $|U| >> m$ è inevitabile che esistano chiavi diverse che producono una collisione.

Dobbiamo, quindi, risolverle.

# Euristiche per avere buone funzioni Hash (1)

**Metodo della divisione**: viene trasformata la chiave *k* in un valore numerico e si prende il resto della divisione per *m* (dimensione della tabella)

$$h(k) = \#k \bmod m$$

Occorre scegliere *m* con cura: ad esempio se devo memorizzare identificatori di un programma, è bene **evitare potenze di due per *m***: in tal caso infatti h(k) dipende solo dai bit meno significativi, cioè gli ultimi caratteri! Una buona scelta è un numero primo "abbastanza lontano" dalle potenze di 2.

**Esempio**: suppongo di dover memorizzare circa $n$=2000 stringhe. Accettando di esaminare in media 3 elementi, scegliamo m=701, numero primo tale che il fattore di carico $\alpha = n/m \sim 3$.

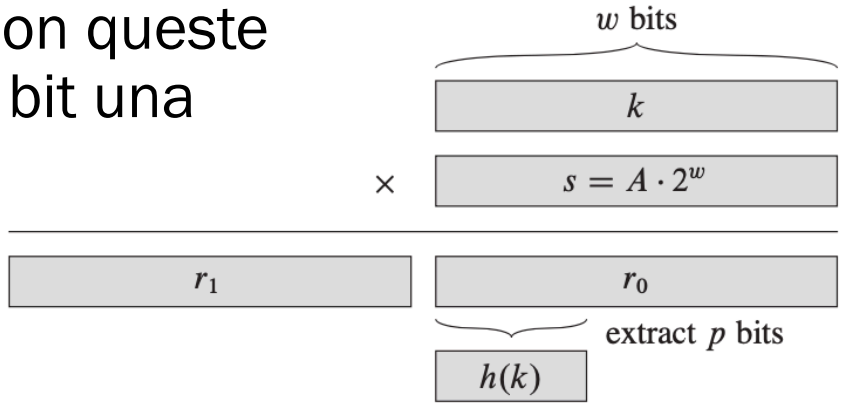
# Euristiche per avere buone funzioni Hash (2)

**Metodo della moltiplicazione**: viene trasformata la chiave *k* in un valore numerico poi la si moltiplica per un valore 0<A<1, si estrae la parte frazionaria di *kA, la si moltiplica per m e si prende la parte intera:*

$$h(k) = \lfloor m(\#kA - \lfloor \#kA \rfloor) \rfloor$$

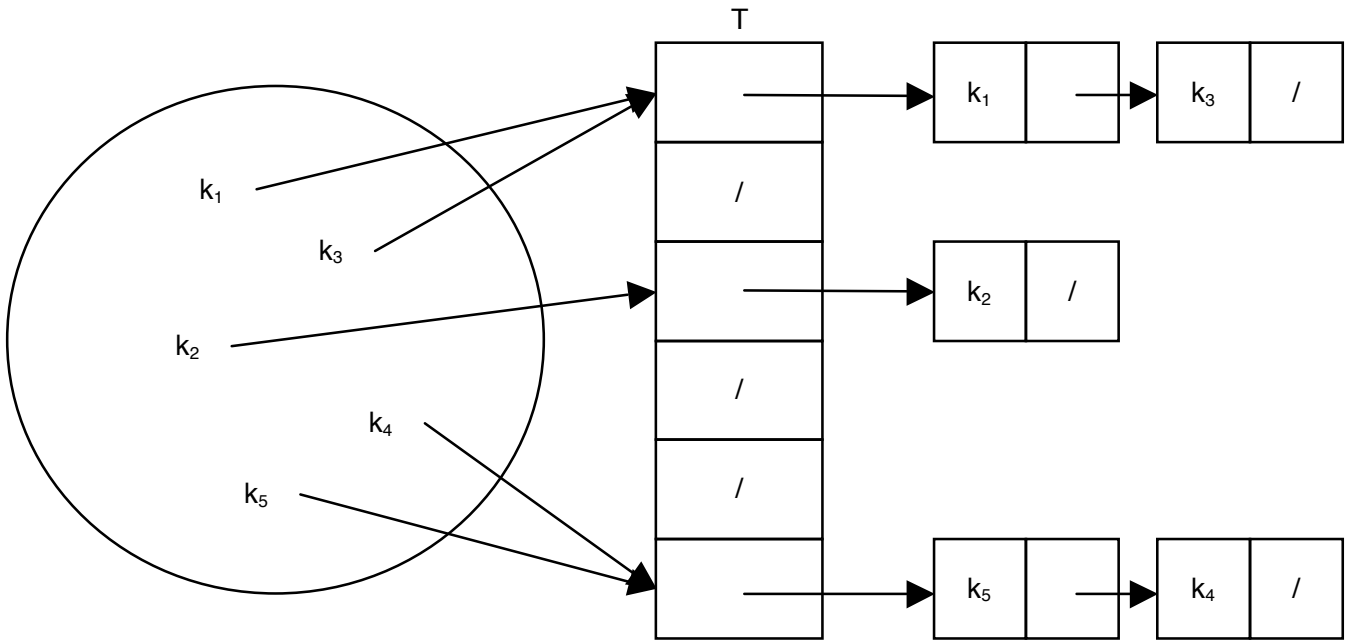
In questo caso, ***m* non è critico** (e può essere scelto come una potenza $2^p$ per rendere la moltiplicazione efficiente nelle usuali architetture). Sempre per motivi di efficienza risulta conveniente scegliere A una frazione nella forma $s/2^w$.

Nella figura si fa vedere che con queste scelte è sufficiente estrarre p bit una volta fatta la moltiplicazione.



# Liste di trabocco (1)

Questa tecnica prevede di inserire tutti gli elementi le cui chiavi mappano nella stessa posizione in una lista concatenata, detta *lista di trabocco*.



# Liste di trabocco (2)

Operazioni elementari:

```
Funz. Insert_Liste_Trabocco (T: tabella; x: elemento)
  inserisci x nella lista puntata da T[h(chiave(x))]
  return
```

Costo computazionale: $\Theta(1)$, anche nel caso peggiore, con l'inserezione in testa alla lista.

# Liste di trabocco (3)

Operazioni elementari (segue):

```
Funz. Search_ListeTrabocco (T: tabella; k: chiave)
   ricerca k nella lista puntata da T[h(k)]
   if k è presente
      then return puntatore all'elemento contenente k
      else return null
```

Costo computazionale: *O(lunghezza della lista puntata da T[h(k)])* il che, nel **caso peggiore**, diviene *O(n)* quando tutti gli *n* elementi memorizzati nella tabella hash mappano nella medesima posizione, ma nel **caso medio** (quando la funzione hash gode di uniformità semplice) è *O(1+ α)*

*dove* ***α = n/m*** *viene detto* ***fattore di carico*** della tabella.

# Osservazioni sulla complessità

Siccome la lunghezza totale delle liste è $n$ (=elementi memorizzati nella tabella hash) ed ho $m$ liste (=dimensione della tabella) la lunghezza media delle liste è $\boldsymbol{\alpha = n/m}$ sotto **le ipotesi di hashing uniforme**.

Le ricerche senza successo hanno chiaramente come valore atteso di complessità $\theta(1+\alpha)$ (ricordiamo che $\alpha$ in generale può dipendere da $n$, quindi $\theta(1+\alpha)$ significa $\theta(1)$ se $\alpha$ è una costante e $\theta(\alpha)$ altrimenti).

Per le ricerche con successo la questione è più complicata, perché le liste più lunghe hanno maggiore probabilità di essere oggetto di ricerche (contengono più elementi).

Tuttavia, calcolando il valor medio, si ottiene $1+ \alpha/2 + \alpha/2n$ che ugualmente $\theta(1+\alpha)$.

# Liste di trabocco (4)

Operazioni elementari (segue):

```
Funz. Delete_ListeTrabocco (T: tabella; x: elemento)
   cancella x dalla lista puntata da T[h(chiave(x))]
   return
```

Costo computazionale: dipende dall'implementazione delle liste di trabocco e valgono, pertanto, tutte le osservazioni fatte per il costo dell'operazione di cancellazione nelle liste.

# Indirizzamento aperto (1)

Questa tecnica prevede di inserire tutti gli elementi direttamente nella tabella, senza far uso di strutture dati aggiuntive.

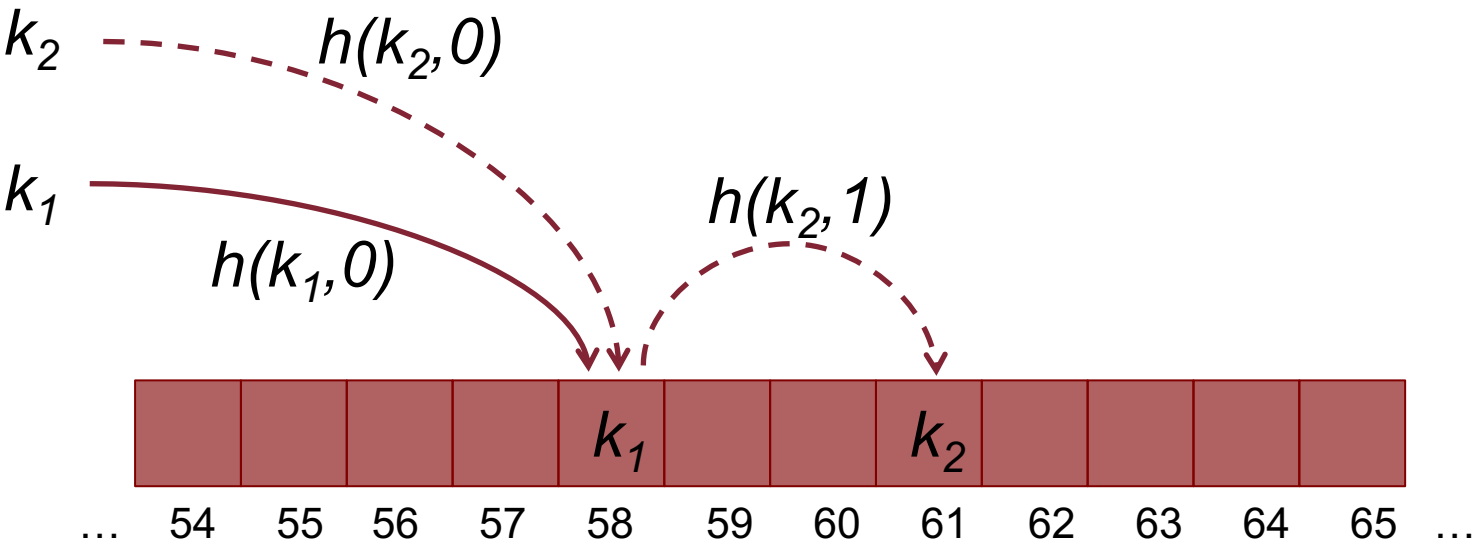
Essa è applicabile quando:

- $m$ è maggiore o uguale al numero $n$ di elementi da memorizzare (quindi il fattore di carico non è mai maggiore di 1);
- $|U| >> m$.

# Indirizzamento aperto (2)

Idea: invece di seguire dei puntatori, calcoliamo (nei modi che vedremo fra breve) la sequenza delle posizioni da esaminare.

La funzione hash dipende ora da 2 parametri: la chiave *k* e il numero di collisioni già trovate.



# Indirizzamento aperto (3)

## Inserimento

Se la posizione iniziale relativa alla chiave *k* è occupata, si scandisce la tabella fino a trovare una posizione libera nella quale l'elemento con chiave *k* può essere memorizzato.

La scansione è guidata da una sequenza di funzioni hash ben determinata: h(k, 0), h(k, 1), …, h(k, m – 1).

Costo computazionale: *O(lunghezza della sequenza che è necessario scandire)* il che, nel **caso peggiore**, diviene *O(n)* quando tutti gli *n* elementi memorizzati nella tabella hash mappano nella medesima posizione, ma nel **caso medio** (quando la funzione hash gode di uniformità semplice) è $1/(1-\alpha)$

dove $\alpha = n/m$ è il fattore di carico della tabella.

# Indirizzamento aperto (4)

## Ricerca senza successo

Si scandisce la tabella mediante la stessa sequenza di funzioni hash utilizzata per l'inserimento fino a quando si incontra una casella vuota, deducendo che l'elemento non è presente.

Costo computazionale: Come per l'inserimento: nel **caso peggiore** $O(n)$, ma nel **caso medio** (quando la funzione hash gode di uniformità semplice) $1/(1-\alpha)$.

dove $\alpha = n/m$ è il fattore di carico della tabella.

# Indirizzamento aperto (5)

## Cancellazione

Questa operazione è piuttosto critica. Infatti:

- se si lascia la casella vuota, ciò impedisce di recuperare qualunque elemento sia stato memorizzato in caselle successive nella sequenza di funzioni hash (N.B. una ricerca viene definita senza successo quando si incontra una casella vuota).
- se si marca con un apposito valore *deleted* la casella da cui è stato cancellato l'elemento, il costo computazionale della ricerca non dipende più esclusivamente dal fattore di carico poiché è influenzata anche dal numero delle posizioni precedentemente occupate da elementi e successivamente marcate.

Per queste ragioni di solito la cancellazione non è supportata con l'indirizzamento aperto.

# Indirizzamento aperto (6)

## Ricerca con successo

Costo computazionale: nell'ipotesi di uniformità semplice, il numero di accessi atteso per una ricerca con successo (ossia la ricerca di un elemento presente nella tabella) è:

$$\frac{1}{\alpha}\log_e \frac{1}{1-\alpha} + \frac{1}{\alpha}$$

dove $\alpha = n/m$ è il fattore di carico

Esempio: con una tabella piena al 50% il numero di accessi atteso è meno di 3,387; con una tabella piena al 90% il numero di accessi atteso è meno di 3,67; tutto questo indipendentemente dalle dimensioni della tabella!

# Indirizzamento aperto (7)

## Hashing doppio

Una tecnica che, pur non realizzando l'uniformità semplice, nella pratica ci si avvicina molto è l'***hashing doppio***.

L'idea è di usare due diverse funzioni hash, una per determinare l'accesso iniziale alla tabella e l'altra per determinare il passo di scansione:

$$h(k, i) = [h_1(k) + i\, h_2(k)] \bmod m,\ i = 0, 1, \ldots, m - 1$$

La bontà di questo metodo risiede nel fatto che, se le due funzioni sono ben progettate, è estremamente improbabile che due chiavi $k_1 \neq k_2$ producano una collisione su *entrambe* le funzioni hash (nel qual caso le due chiavi scandirebbero l'intera tabella nello stesso modo, situazione indesiderabile).