

# Tabelle hash

Damiano Macedonio  
Università Ca' Foscari di Venezia

[mace@unive.it](mailto:mace@unive.it)

Original work Copyright © Alberto Montresor, University of Trento  
(<http://www.dit.unitn.it/~montreso/asd/index.shtml>)

Modifications Copyright © 2009, 2010, 2011, Moreno Marzolla, Università di Bologna

Modifications Copyright © 2013, Damiano Macedonio, Università Ca' Foscari di Venezia

*This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit*

*<http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*

# Introduzione

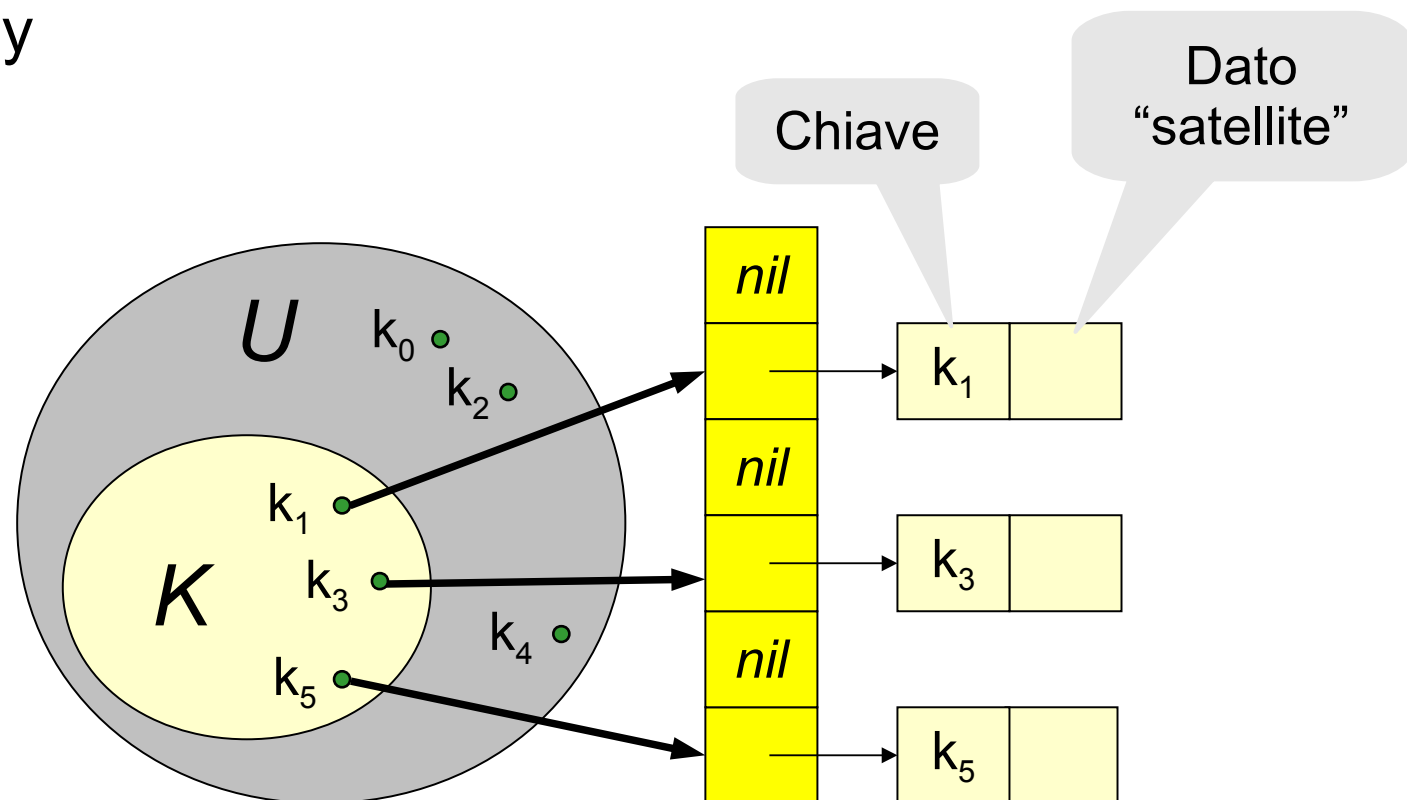
- Dizionario:
  - Struttura dati per memorizzare insiemi **dinamici** di coppie **(chiave, valore)**
  - Il valore è un “dato satellite”
  - Dati indicizzati in base alla chiave
  - Operazioni: **insert(key, item)**, **search(key)** e **delete(key)**
- Costo delle operazioni:
  - $O(\log n)$  nel caso pessimo, usando alberi bilanciati
  - Sarebbe bello poter scendere a  $O(1)$

# Notazione

- **U**—Universo di tutte le possibili chiavi
- **K**—Insieme delle chiavi effettivamente memorizzate
- Possibili implementazioni
  - U corrisponde all'intervallo  $[0..m-1]$ ,  $|K| \sim |U|$ 
    - **tabelle ad indirizzamento diretto**
  - U è un insieme generico,  $|K| \ll |U|$ 
    - **tabelle hash**

# Tabelle a indirizzamento diretto

- Implementazione:
  - Basata su array
  - L'elemento con chiave  $k$  è memorizzato nel  $k$ -esimo "slot" dell'array



# Esempio

- Azienda con un certo numero di dipendenti, ciascun dipendente è identificato da un numero univoco di 2 cifre (00..99)
- Chiave: numero di matricola
- Dati satellite: nome e cognome, numero di telefono, indirizzo...

00			
01	Carlo Rossi	123-4567	Via Tizio
02	Giuseppe Verdi	234-1432	Via Caio
03			
⋮			
97			
98	Antonio Bianchi	012-9876	Via Alta
99			

*Record vuoto: non c'è alcun dipendente con n. di matricola 99*

# Tabelle a indirizzamento diretto

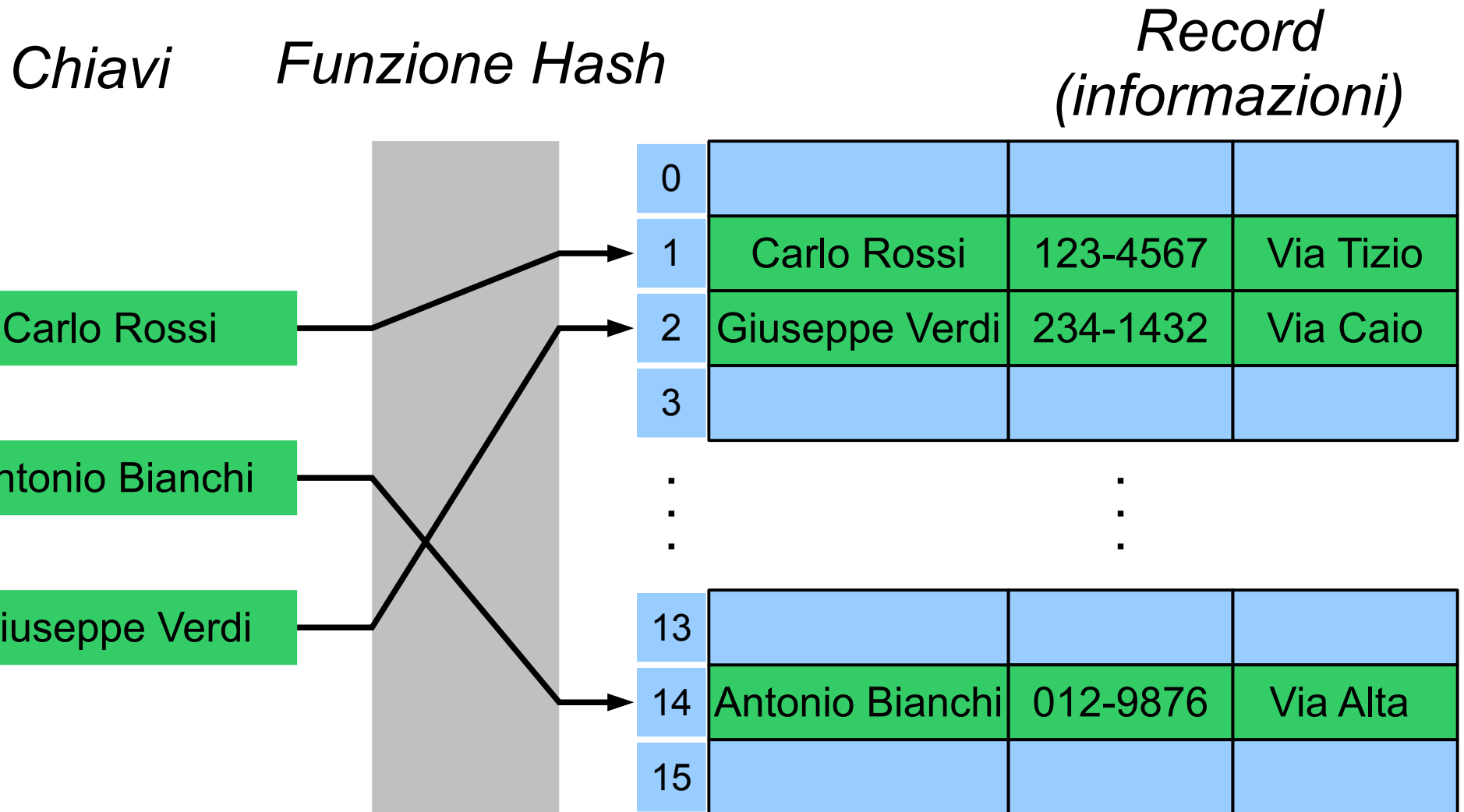
- Se  $|K| \sim |U|$ :
  - Non sprechiamo (troppo) spazio
  - Operazioni in tempo  $O(1)$  nel caso peggiore
- Se  $|K| \ll |U|$ : soluzione non praticabile
  - Esempio: studenti ASD con chiave “numero di matricola”
  - Se il numero di matricola ha 6 cifre, l'array deve avere spazio per contenere  $10^6$  elementi
  - Se gli studenti del corso sono ad esempio 30, lo spazio realmente occupato dalle chiavi memorizzate è  $30/10^6 = 0.00003 = 0.003\%$  !!!

# Tabelle hash

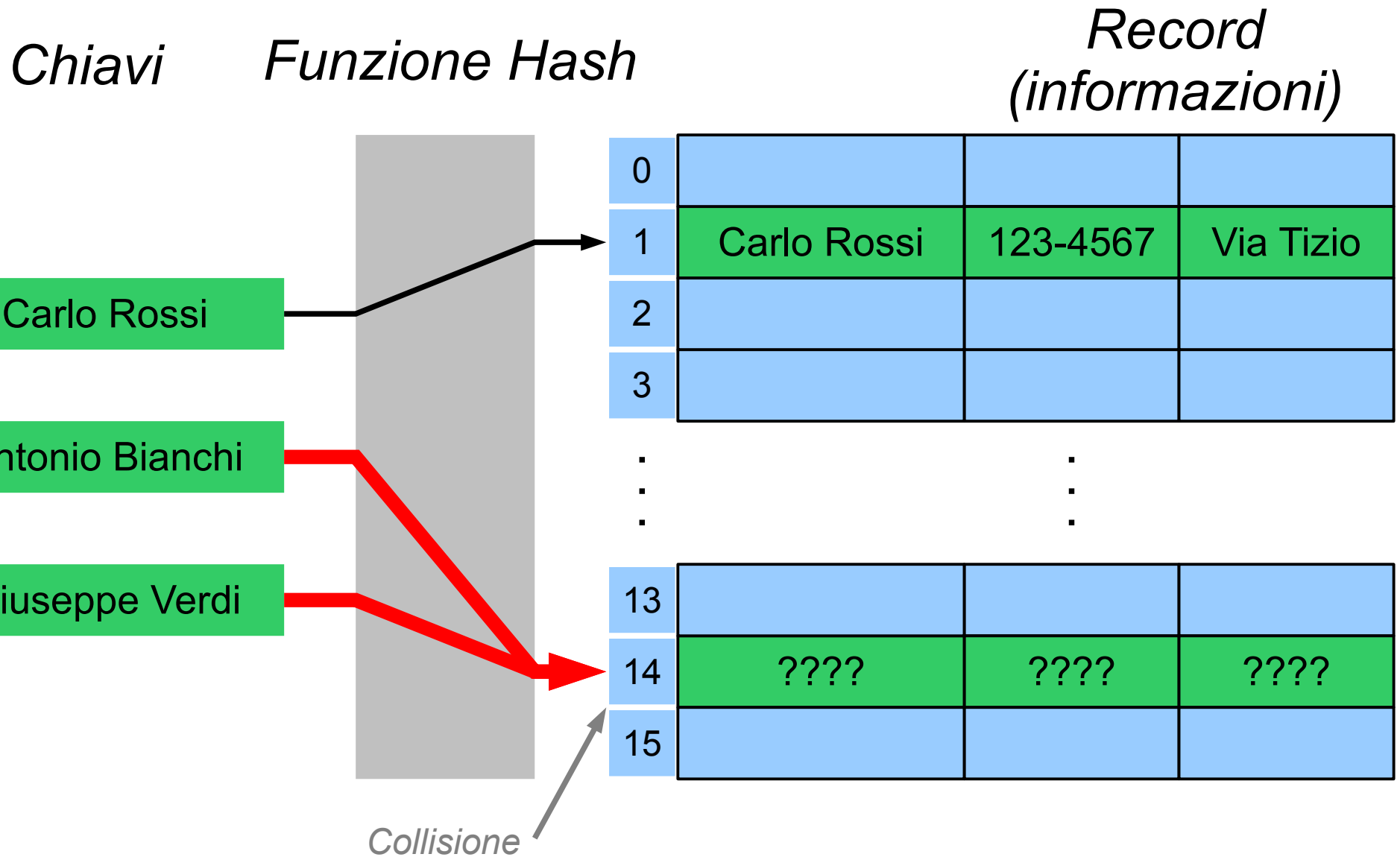
- Tabelle hash:
  - ◊ Un array  $A[0..m-1]$
  - ◊ Una funzione hash  $h: U \rightarrow \{0, \dots, m-1\}$
- Indirizzamento hash:
  - ◊ Diciamo che  $h(k)$  è il **valore hash** della chiave  $k$
  - ◊ La funzione  $h$  trasforma una chiave nell'indice della tabella
  - ◊ La chiave  $k$  viene inserita (se possibile) nello slot  $A[h(k)]$
  - ◊ Quando due o più chiavi hanno lo stesso valore hash, diciamo che è avvenuta una **collisione**
- Idealmente: vogliamo funzioni hash senza collisioni



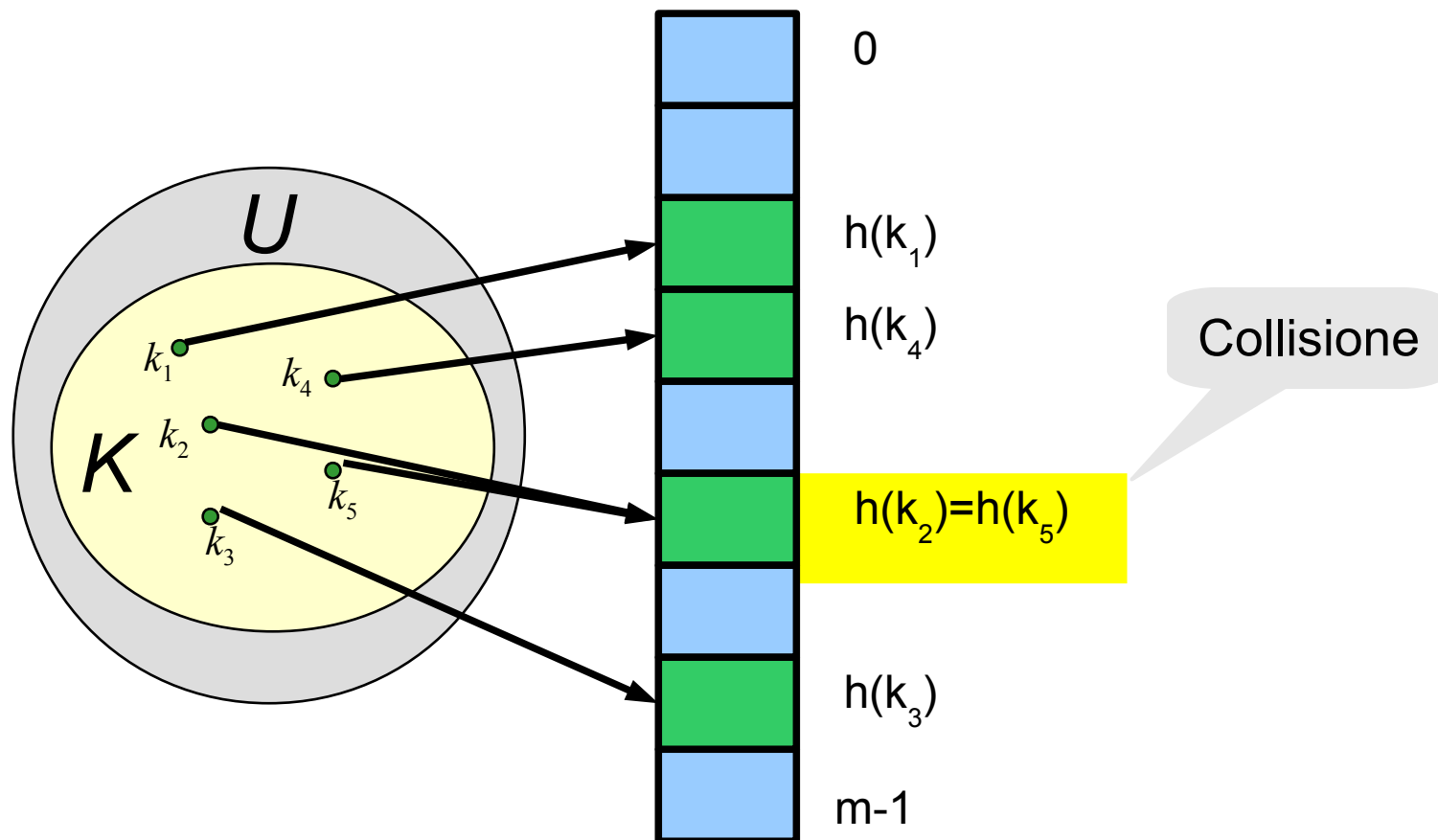
# Tabelle hash



# Tabelle hash



# Tabelle hash



# Problema delle collisioni

- Utilizzo di funzioni hash perfette
  - Una funzione hash  $h$  si dice **perfetta** se è iniettiva, ovvero:
$$\forall u, v \in U : u \neq v \rightarrow h(u) \neq h(v)$$
  - Si noti che questo richiede che  $|U| \leq m$
- Esempio:
  - Numero di matricola degli studenti ASD solo negli ultimi tre anni
  - Distribuiti fra 234.717 e 235.716
  - $h(k) = k - 234.717$ ,  $m = 1000$
- Problema: spazio delle chiavi spesso grande, sparso
  - È impraticabile ottenere una funzione hash perfetta

# Funzioni hash

- Se le collisioni sono inevitabili
  - almeno cerchiamo di minimizzare il loro numero
  - vogliamo funzioni che distribuiscano **uniformemente** le chiavi negli indici **[0..m-1]** della tabella hash
- Uniformità semplice:
  - sia **P(k)** la probabilità che una chiave **k** sia presente nella tabella
  - La quantità  $Q(i) = \sum_{k: h(k)=i} P(k)$   
è la probabilità che una chiave finisca nella cella **i**.
  - Una funzione **h()** gode della proprietà di **uniformità semplice** se

$$\forall i \in [0 \dots m - 1]: Q(i) = 1/m$$

# Funzioni hash

- Per poter ottenere una funzione hash con uniformità semplice, la distribuzione delle probabilità  $P$  deve essere nota
- Esempio:
  - $U$  è composto da numeri reali in  $[0, 1)$  e ogni chiave  $k$  ha la stessa probabilità di essere scelta. Allora

Estremo superiore  
escluso

$$h(k) = \lfloor km \rfloor$$

soddisfa la proprietà di uniformità semplice

- Nella realtà la distribuzione esatta può non essere (completamente) nota

# Funzioni hash: assunzioni

- Tutte le chiavi sono equiprobabili:  $P(k) = 1 / |U|$ 
  - Semplificazione necessaria per proporre un meccanismo generale
- Le chiavi sono valori numerici non negativi
  - È possibile trasformare una chiave complessa in un numero, ad esempio considerando il valore decimale della sua rappresentazione binaria

# Funzioni hash

- Metodo della divisione:  $h(k) = k \bmod m$ 
  - Resto della divisione intera di  $k$  per  $m$ :
  - Esempio:  $m=12, k=100 \rightarrow h(k) = 4$
- Vantaggio
  - Molto veloce (richiede una divisione intera)
- Svantaggio
  - Il valore  $m$  deve essere scelto opportunamente: ad esempio vanno bene numeri primi, distanti da potenze di 2 (e di 10)



# Funzioni hash

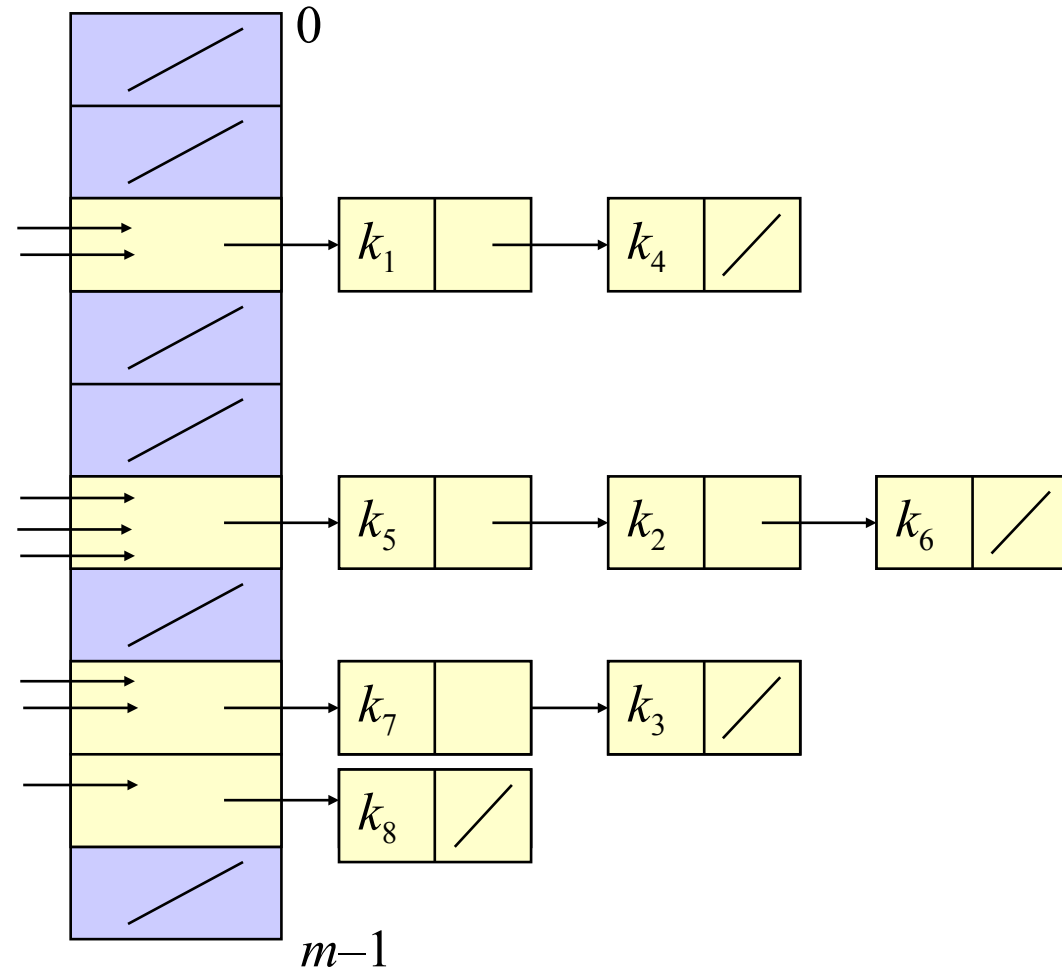
- Metodo della moltiplicazione:  $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ 
  - $A$  è una costante compresa tra 0 e 1:  $0 < A < 1$
  - Moltiplichiamo  $k$  per  $A$  e prendiamo la parte frazionaria
  - Moltiplichiamo quest'ultima per  $m$  e prendiamo la parte intera
- Esempio:  $m = 1000$ ,  $k = 123$ ,  $A \approx 0.6180339887... \rightarrow h(k) = 18$
- Svantaggi: più lento del metodo di divisione
- Vantaggi: il valore di  $m$  non è critico
- Come scegliere  $A$ ? Knuth suggerisce  $A \approx (\sqrt{5} - 1)/2$ .

# Problema delle collisioni

- Abbiamo ridotto, ma non eliminato, il numero di collisioni
- Come gestire le collisioni residue?
  - Dobbiamo trovare collocazioni alternative per le chiavi che collidono
  - Se una chiave non si trova nella posizione attesa, bisogna andarla a cercare nelle posizioni alternative
  - Le operazioni possono costare  $\Theta(n)$  nel caso peggiore...
  - ...ma hanno costo  $\Theta(1)$  nel caso medio
- Due delle possibili tecniche:
  - Concatenamento
  - Indirizzamento aperto

# Risoluzione delle collisioni

- Concatenamento (chaining)
  - Gli elementi con lo stesso valore hash  $h$  vengono memorizzati in una lista concatenata
  - Si memorizza un puntatore alla testa della lista nello slot  $A[h]$  della tabella hash
- Operazioni:
  - **Insert:** inserimento in testa
  - **Search, Delete:** richiedono di scandire la lista alla ricerca della chiave

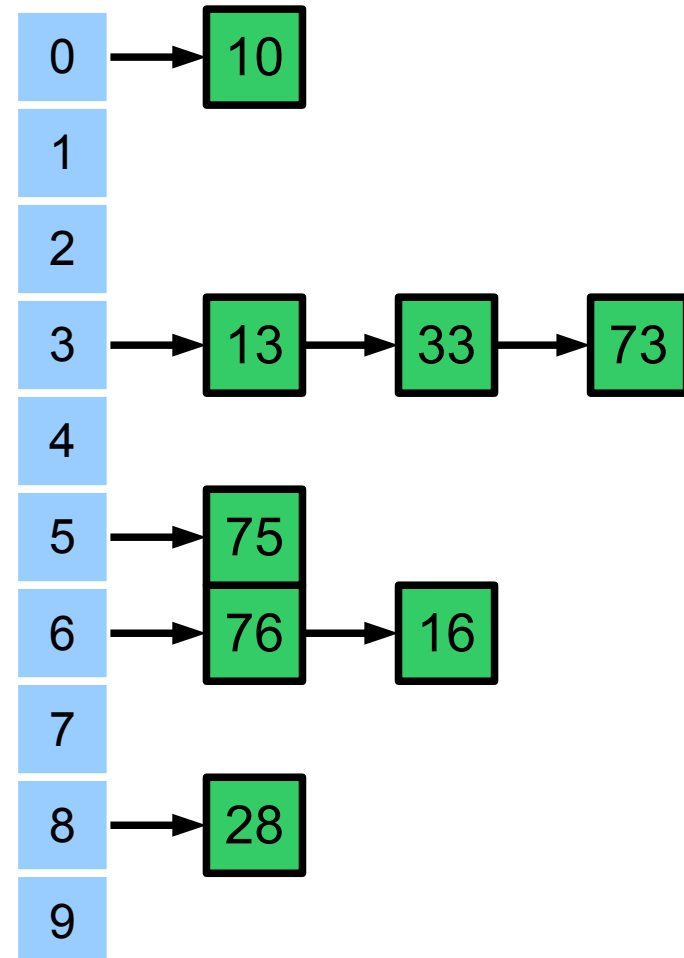


# Esempio

$$h(k) = k \bmod 10$$

Attenzione  
 $m=10$  non è una  
buona scelta

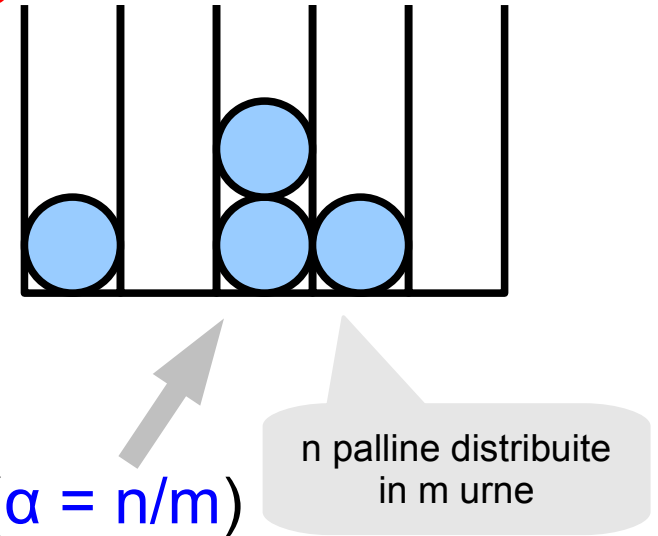
- Inserire le chiavi seguenti, nell'ordine indicato:
  - 10, 73, 16, 33, 76, 13, 75, 28
- Cosa devo fare per trovare la chiave 73?
- Cosa devo fare per cancellare la chiave 33?
- Cosa devo fare per trovare la chiave 24?



# Concatenamento

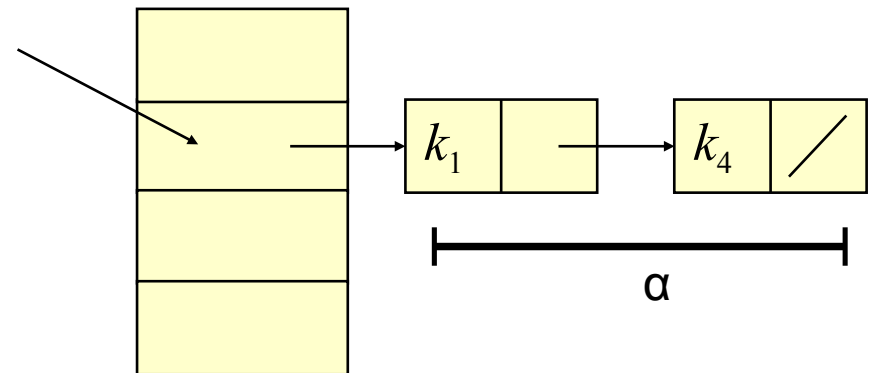
## Costo computazionale

- Notazione
  - $n$ : numero di elementi nella tabella
  - $m$ : numero di slot nella tabella
- **Fattore di carico**
  - $\alpha$ : numero medio di elementi nelle liste ( $\alpha = n/m$ )
- **Caso pessimo: tutte le chiavi sono in una unica lista**
  - Insert:  $\Theta(1)$
  - Search, Delete:  $\Theta(n)$
- **Caso medio: dipende da come le chiavi vengono distribuite**
  - Assumiamo hashing uniforme, da cui ogni slot della tabella avrà mediamente  $\alpha$  chiavi



# Concatenamento: complessità

- Teorema:
  - In tabella hash con concatenamento, una ricerca senza successo richiede un tempo atteso di  $\Theta(1 + \alpha)$
- Dimostrazione:
  - Una chiave non presente nella tabella può essere collocata in uno qualsiasi degli  $m$  slot
  - Una ricerca senza successo tocca tutte le chiavi nella lista corrispondente
  - Tempo di hashing:  $1 +$   
lunghezza attesa lista:  $\alpha \rightarrow$   
 $\Theta(1+\alpha)$



# Concatenamento: complessità

- Teorema:
  - In una tabella hash con concatenamento, una ricerca con successo richiede un tempo atteso di  $\Theta(1 + \alpha)$
  - Più precisamente:  $\Theta(2 + \alpha/2 + \alpha/2n)$ 
    - (dove  $n$  è il numero di elementi)
- Qual è il significato:
  - se  $n = O(m)$ ,  $\alpha = O(1)$
  - quindi tutte le operazioni sono  $\Theta(1)$

# Indirizzamento aperto

- Problema della gestione di collisioni tramite concatenamento
  - Struttura dati complessa, con liste, puntatori, etc.
- Gestione alternativa: **indirizzamento aperto**
  - Idea: memorizzare tutte le chiavi nella tabella stessa
  - Ogni slot contiene una chiave oppure **null**
  - **Inserimento**:
    - Se lo slot prescelto è utilizzato, si cerca uno slot “alternativo”
  - **Ricerca**:
    - Si cerca nello slot prescelto, e poi negli slot “alternativi” fino a quando non si trova la chiave oppure **nil**



# Indirizzamento aperto

- Cosa succede al fattore di carico  $\alpha$ ?
  - Compreso fra 0 e 1
  - La tabella può andare in overflow
    - Inserimento in tabella piena
  - Esistono tecniche di crescita/contrazione della tabella
    - linear hashing

# Indirizzamento aperto

- **Ispezione**: Uno slot esaminato durante una ricerca di chiave
- **Sequenza di ispezione**: La lista ordinata degli slot esaminati
- Funzione hash: estesa come
  - $h : U \times [0..m-1] \rightarrow [0..m-1]$ 
    - chiave
    - “tentativo”
- La sequenza di ispezione  $\{ h(k,0), h(k,1), \dots, h(k,m-1) \}$  è una permutazione degli indici  $[0..m-1]$ 
  - Può essere necessario esaminare ogni slot nella tabella
  - Non vogliamo esaminare ogni slot più di una volta

# Esempio

- Vettore con  $m=5$  elementi  
 $h(k,i) = ((k \bmod 5) + i) \bmod 5$
- Insert 14
- Insert 19
- Insert 34
- Lookup 34
- Lookup 11
- Delete 19
- Lookup 34

# Inserimento e Ricerca (attenzione...)

Hash-Insert ( $A, k$ )

```
 $i := 0$ 
repeat
     $j := h(k, i)$ 
    if  $A[j] == \text{null}$  then
         $A[j] := k$ 
        return  $j$ 
    else
         $i := i + 1$ 
    endif
until  $i == m$ 
error "hash table overflow"
```

Hash-Search ( $A, k$ )

```
 $i := 0$ 
repeat
     $j := h(k, i)$ 
    if  $A[j] == k$  then
        return  $j$ 
    endif
     $i := i + 1$ 
until  $A[j] == \text{null}$  or  $i == m$ 
return nil
```

# Cancellazione

- Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un `null`. **Perché?**
- Approccio
  - Utilizziamo un speciale valore **DELETED** al posto di `null` per marcare uno slot come vuoto dopo la cancellazione
  - *Ricerca*: **DELETED** trattati come slot pieni
  - *Inserimento*: **DELETED** trattati come slot vuoti

# Inserimento e Ricerca

Hash-Insert ( $A, k$ )

```
 $i := 0$ 
repeat
   $j := h(k, i)$ 
  if  $A[j] == \text{null} \ ||$ 
     $A[j] == \text{DELETED}$  then
     $A[j] := k$ 
    return  $j$ 
  else
     $i := i + 1$ 
  endif
until  $i == m$ 
error "hash table overflow"
```

Hash-Search ( $A, k$ )

```
 $i := 0$ 
repeat
   $j := h(k, i)$ 
  if  $A[j] == k$  then
    return  $j$ 
  endif
   $i := i + 1$ 
until  $A[j] == \text{null}$  or  $i == m$ 
return nil
```

# Ispezione lineare

- Funzione:  $h(k, i) = (h'(k) + i) \bmod m$ 
  - chiave
  - n. ispezione
  - funzione hash ausiliaria
- Il primo elemento determina l'intera sequenza
  - $h'(k), h'(k)+1, \dots, m-1, 0, 1, \dots, h'(k)-1$
  - Solo  $m$  sequenze di ispezione distinte sono possibili
- Problema: **primary clustering**
  - Lunghe sotto-sequenze occupate...
  - ... che tendono a diventare più lunghe:
    - uno slot vuoto preceduto da  $i$  slot pieni viene riempito con probabilità  $(i+1)/m$
  - I tempi medi di inserimento e cancellazione crescono

# Esempio

(C. Demetrescu, I. Finocchi, G. F. Italiano, "Algoritmi e strutture dati", seconda edizione, McGraw-Hill, fig. 7.7 p. 189)

		C	E		I	L	M	N	O	P	R	S	T	V																			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30		
P																			P														
R																			P	R													
E							E												P	R													
C						C	E												P	R													
I						C	E					I							P	R													
P						C	E					I							P	P	R												
I						C	E					I	I						P	P	R												
T						C	E					I	I						P	P	R												
E						C	E	E				I	I						P	P	R												
V						C	E	E				I	I						P	P	R												
O						C	E	E				I	I						O	P	P	R											
L						C	E	E				I	I	L					O	P	P	R											
I						C	E	E				I	I	I	L				O	P	P	R											
S						C	E	E				I	I	I	L				O	P	P	R	S										
S						C	E	E				I	I	I	L				O	P	P	R	S	T	S								
I						C	E	E				I	I	I	L	I			O	P	P	R	S	T	S	V							
M						C	E	E				I	I	I	L	I	M		O	P	P	R	S	T	S	V							
E						C	E	E	E			I	I	I	L	I	M	O	P	P	R	S	T	S	V								
V						C	E	E	E			I	I	I	L	I	M	O	P	P	R	S	T	S	V	V							
O						C	E	E	E			I	I	I	L	I	M	O	P	P	R	S	T	S	V	V	O						
L						C	E	E	E			I	I	I	L	I	M	O	P	P	R	S	T	S	V	V	O	L					
M						C	E	E	E			I	I	I	L	I	M	O	P	P	R	S	T	S	V	V	O	L	M				
E						C	E	E	E	E		I	I	I	L	I	M	O	P	P	R	S	T	S	V	V	O	L	M				
N						C	E	E	E	E		I	I	I	L	I	M	O	P	P	R	S	T	S	V	V	O	L	M	N			
T						C	E	E	E	E		I	I	I	L	I	M	O	P	P	R	S	T	S	V	V	O	L	M	N	T		
E	E					C	E	E	E	E		I	I	I	L	I	M	O	P	P	R	S	T	S	V	V	O	L	M	N	T		



# Ispezione quadratica

- Funzione:  $h(k,i) = (h'(k) + c_1i + c_2i^2) \bmod m$   $c_1 \neq c_2$ 
  - chiave
  - n. ispezione
  - funzione hash ausiliaria
- Sequenza di ispezioni:
  - L'ispezione iniziale è in  $h'(k)$
  - Le ispezione successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione  $i$
- Nota:  $c_1, c_2, m$  devono essere tali da garantire la permutazione di  $[0..m-1]$ .
- Problema: **clustering secondario**
  - Se due chiavi hanno la stessa ispezione iniziale, poi le loro sequenze sono identiche

# Esempio

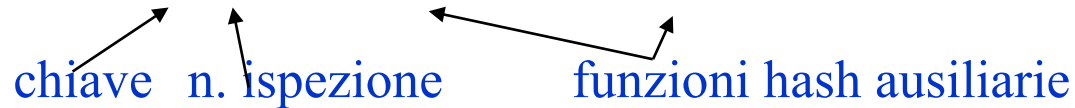
(C. Demetrescu, I. Finocchi, G. F. Italiano, "Algoritmi e strutture dati", seconda edizione, McGraw-Hill, Fig 7.8 p. 191)

					C	E					I		L	M	N	O	P		R	S	T		V														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30						
P																		P																			
R																		P	R																		
E							E											P	R																		
C						C	E											P	R																		
I						C	E				I							P	R																		
P						C	P	E			I							P	R																		
I						C	P	E			I							P	R					I													
T						C	P	E			I							P	R					T	I												
E						C	P	E			I					E		P	R					T	I												
V						C	P	E			I				E		P	R					T	I	V												
O						C	P	E			I				E	O	P	R					T	I	V												
L						C	P	E			I				L	E	O	P	R				T	I	V												
I						I	C	P	E			I			L	E	O	P	R				T	I	V												
S						I	C	P	E			I			L	E	O	P	R				S	T	I	V											
S						I	C	P	E			I	S		L	E	O	P	R				S	T	I	V											
I						I	C	P	E			I	S		L	E	I	O	P	R			S	T	I	V											
M	M					I	C	P	E			I	S		L	E	I	O	P	R			S	T	I	V											
E	M					I	C	P	E	E		I	S		L	E	I	O	P	R			S	T	I	V											
V	M					I	C	P	E	E		I	S		L	E	I	O	P	R			S	T	I	V	V										
O	M					I	C	P	E	E	O		I	S		L	E	I	O	P	R		S	T	I	V	V										
L	M					I	C	P	E	E	O		I	S		L	E	I	O	P	R		S	T	I	V	V									L	
M	M	M				I	C	P	E	E	O		I	S		L	E	I	O	P	R		S	T	I	V	V										
E	M	M	E			I	C	P	E	E	O		I	S		L	E	I	O	P	R		S	T	I	V	V									L	
N	M	M	E			I	C	P	E	E	O		I	S		L	E	I	O	P	N	R	S	T	I	V	V									L	
T	M	M	E			I	C	P	E	E	O		I	S	T	L	E	I	O	P	N	R	S	T	I	V	V									L	
E	M	M	E			I	C	P	E	E	O	E		I	S	T	L	E	I	O	P	N	R	S	T	I	V	V									L

# Doppio hashing

- Funzione:  $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

chiave    n. ispezione    funzioni hash ausiliarie

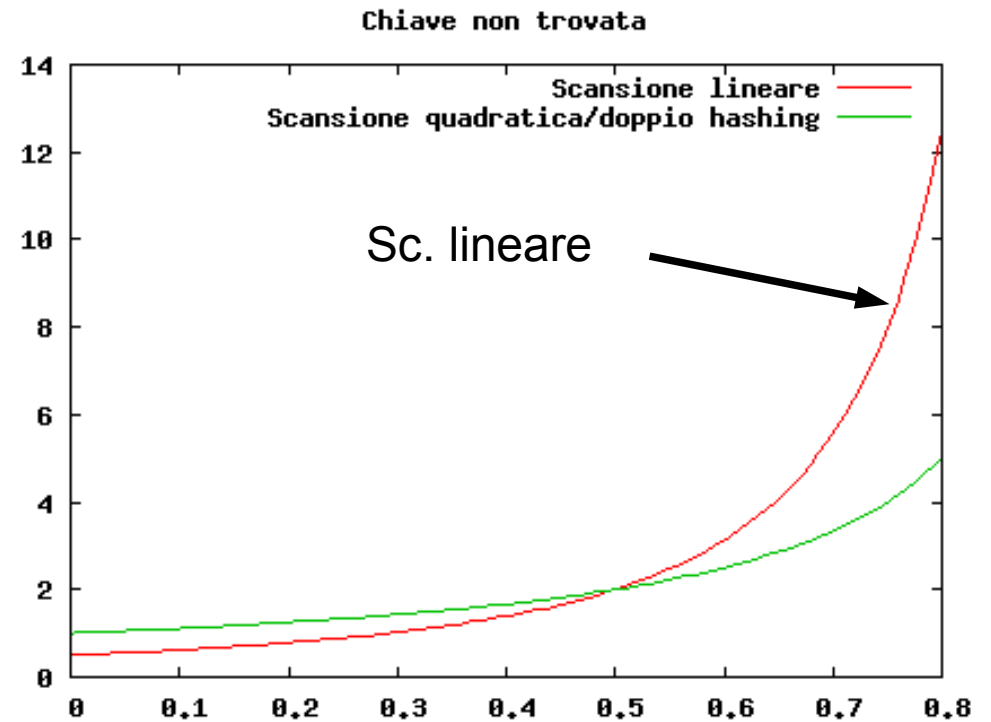
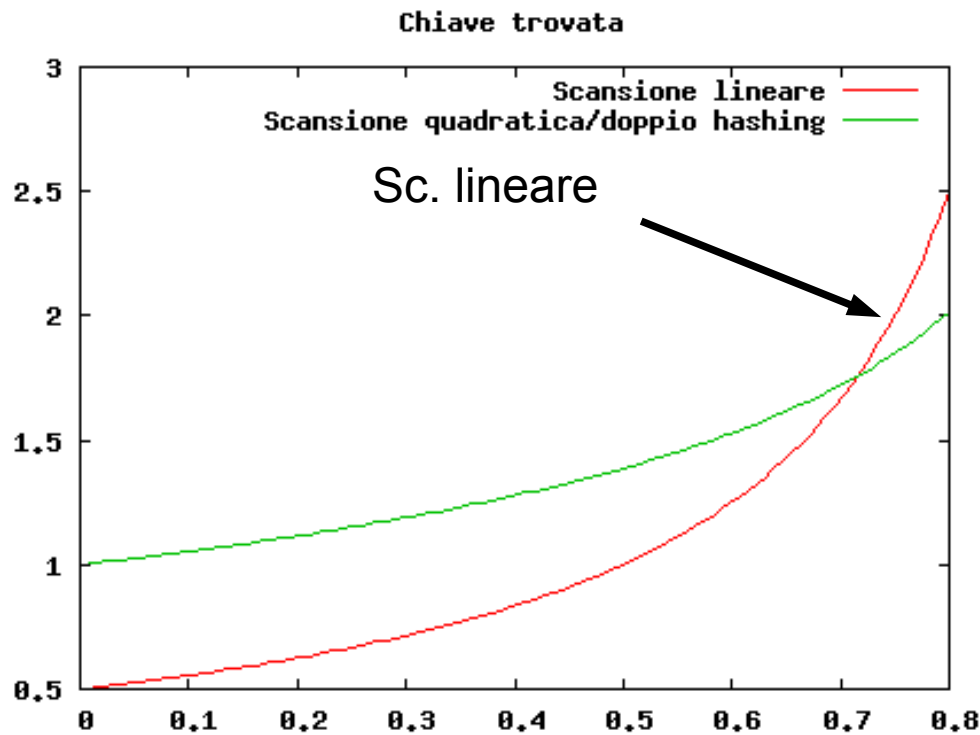
The diagram consists of three blue labels with arrows pointing to the corresponding parts of the formula  $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$ . The label 'chiave' has an arrow pointing to  $k$ . The label 'n. ispezione' has an arrow pointing to  $i$ . The label 'funzioni hash ausiliarie' has an arrow pointing to  $h_1(k)$  and  $h_2(k)$ .

- Due funzioni ausiliarie:
  - $h_1$  fornisce la prima ispezione
  - $h_2$  fornisce l'offset delle successive ispezioni

# Analisi dei costi di scansione nel caso medio

Esito ricerca	Concatenamento	Scansione lineare	Scansione quadratica / hashing doppio
Chiave trovata	$\Theta(1 + \alpha)$	$\frac{1}{2} + \frac{1}{2(1 - \alpha)}$	$-\frac{1}{\alpha} \ln(1 - \alpha)$
Chiave non trovata	$\Theta(1 + \alpha)$	$\frac{1}{2} + \frac{1}{2(1 - \alpha)^2}$	$\frac{1}{1 - \alpha}$

# Scansione lineare vs scansione quadratica



# Osservazione

- Nel caso di gestione degli overflow mediante liste concatenate, è possibile avere  $\alpha > 1$
- Nel caso di gestione degli overflow mediante indirizzamento aperto, è possibile solo avere  $\alpha \leq 1$ 
  - Una volta che l'array è pieno, non è più possibile aggiungere altri elementi
  - Come fare in questo caso? (con array dinamici, ma cambiando la funzione hash!)