

Algoritmi e Strutture Dati

Capitolo 7 - Tabelle hash

Alberto Montresor
Università di Trento

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.

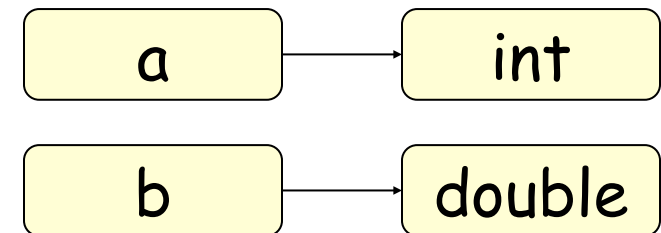
Introduzione

♦ Dizionario (reloaded):

- ♦ Struttura dati per memorizzare insiemi dinamici di coppie (chiave, valore)
- ♦ Il valore è un “dato satellite”
- ♦ Dati indicizzati in base alla chiave
- ♦ Operazioni: insert(), remove() e lookup()

♦ Applicazioni:

- ♦ Le tabelle dei simboli di un compilatore
- ♦ La gestione della memoria nei sistemi operativi



◆ Possibili implementazioni e relativi costi

	Array non ordinato	Array ordinato	Lista	Alberi (abr, rb, ...)	Performance ideale
insert()	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
looup()	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
remove()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$

Notazione

- ♦ U – Universo di tutte le possibili chiavi
- ♦ K – Insieme delle chiavi effettivamente memorizzate
- ♦ Possibili implementazioni
 - ♦ $|U|$ corrisponde al range $[0..m-1]$, $|K| \sim |U| \rightarrow$
 - ♦ tabelle ad indirizzamento diretto
 - ♦ U è un insieme generico, $|K| \ll |U| \rightarrow$
 - ♦ tabelle hash

Tabelle a indirizzamento diretto

♦ Implementazione:

- ♦ Basata su array ordinari
- ♦ L'elemento con chiave k è memorizzato nel k -esimo “slot” del vettore

♦ 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 “n. matricola”

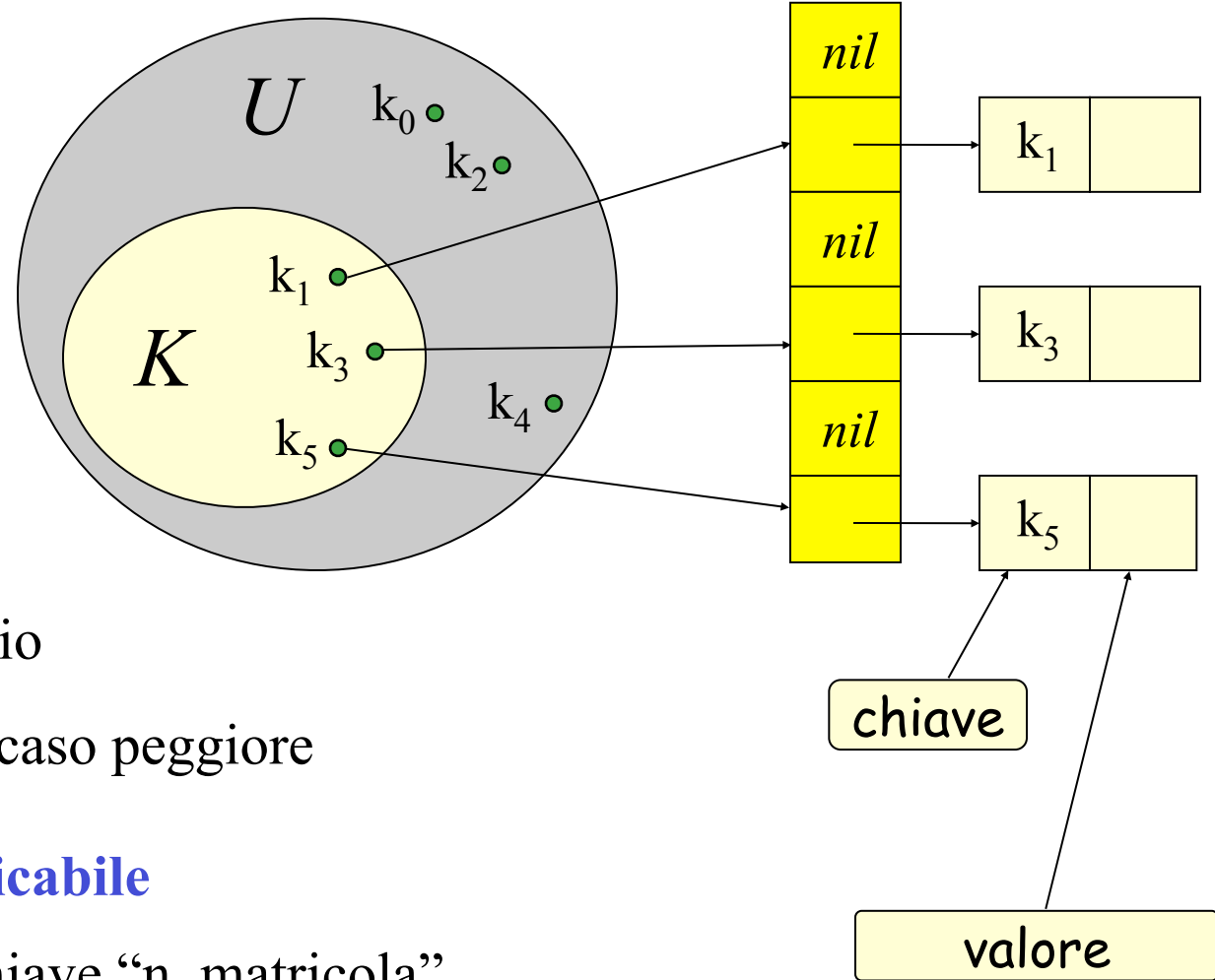


Tabelle hash		0

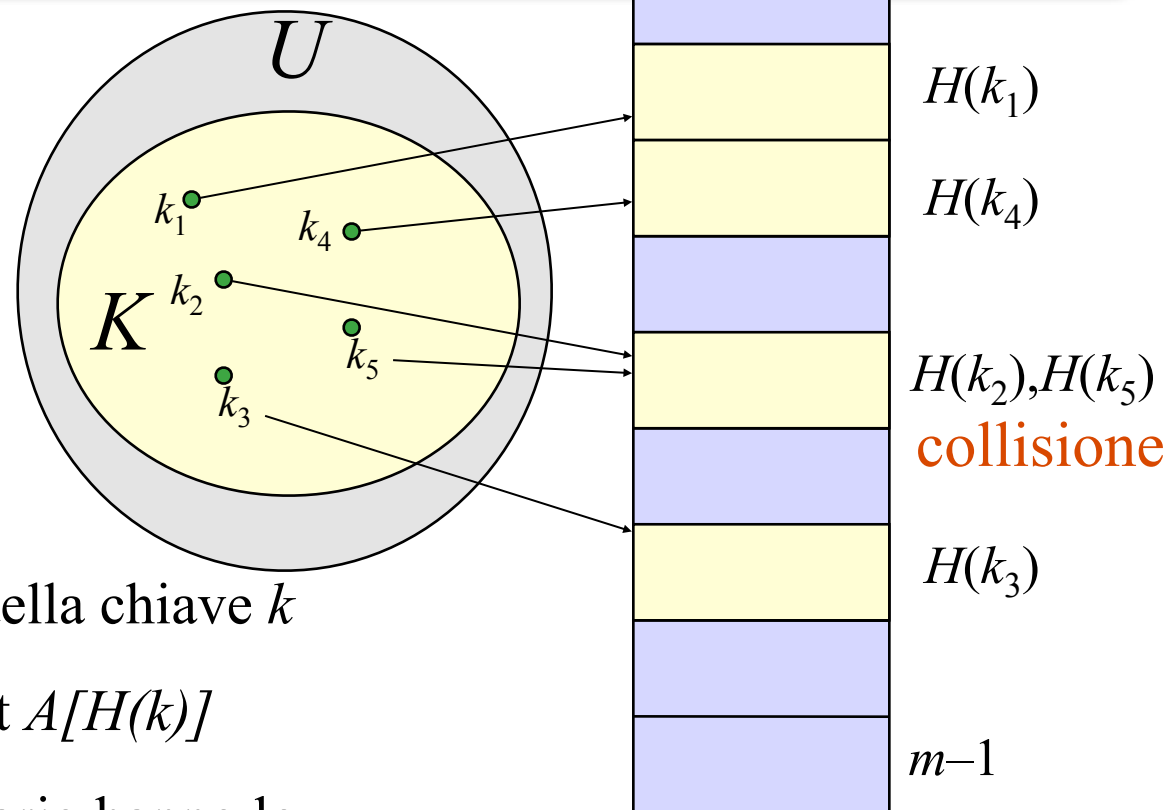
✦ **Tabelle hash:**

- ✦ Un vettore $A[0..m-1]$
- ✦ Una *funzione hash*
 $H: U \rightarrow \{0, \dots, m-1\}$

✦ Indirizzamento hash:

- ✦ Diciamo che $H(k)$ è il valore hash della chiave k
- ✦ Chiave k viene “mappata” nello slot $A[H(k)]$
- ✦ Quando due o più chiavi nel dizionario hanno lo stesso valore hash, diciamo che è avvenuta una *collisione*

- ✦ **Idealmente: vogliamo funzioni hash senza collisioni**



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 $m \geq |U|$

- ♦ **Esempio:**

- ♦ 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, non conosciuto**

- ♦ E' spesso 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 inserita nella tabella
 - ♦ sia $Q(i) = \sum_{k: H(k)=i} P(k)$ la probabilità che una chiave qualsiasi, 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 numeri reali in $[0,1]$ e ogni chiave ha la stessa probabilità di essere scelta, allora

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

soddisfa la proprietà di uniformità semplice

- ♦ **Nella realtà**

- ♦ La distribuzione esatta può non essere (completamente) nota
 - ♦ Si utilizzano allora tecniche “euristiche”

♦ Assunzioni:

- ♦ Tutte le chiavi sono equiprobabili: $P(k) = 1 / |U|$
 - ♦ Semplificazione necessaria per proporre un meccanismo generale
- ♦ Le chiavi sono valori numerici non negativi
 - ♦ E' possibile trasformare una chiave complessa in un numero
 - ♦ $ord(c)$: valore ordinale del carattere c
 - ♦ $bin(k)$: rappresentazione binaria della chiave k , concatenando i valori ordinali dei caratteri che lo compongono
 - ♦ $int(k)$: valore numerico associato ad una chiave k
 - ♦ Esempio:
 - ♦ $bin(\text{"DOG"}) \rightarrow ord('D') \text{ } ord(\text{"O"}) \text{ } ord(\text{"G"})$
 $\rightarrow 01000100 \text{ } 01001111 \text{ } 01000111$
 - ♦ $int(\text{"DOG"}) \rightarrow 68 \cdot 256^2 + 79 \cdot 256 + 71$
 $\rightarrow 4.476.743$

✦ Nei prossimi esempi

- ✦ $ord('a') = 1, ord('b')=2, \dots, ord('z')=26, ord(' \underline{b} ') = 32$

- ✦ b rappresenta lo spazio

- ✦ Sono sufficienti 6 bit per rappresentare questi caratteri

✦ Si considerino le seguenti due stringhe: “weberb” e “webern”

✦ Rappresentazione binaria

- ✦ $bin(\text{“weber}\underline{b}\text{”}) = 010111\ 000101\ 000010\ 000101\ 010010\ 100000$

- ✦ $bin(\text{“webern”}) = 010111\ 000101\ 000010\ 000101\ 010010\ 001110$

✦ Rappresentazione intera

- ✦ $int(\text{“weber}\underline{b}\text{”}) = 23 \cdot 64^5 + 5 \cdot 64^4 + 2 \cdot 64^3 + 5 \cdot 64^2 + 18 \cdot 64^1 + 32 \cdot 64^0 = 24.780.493.966$

- ✦ $int(\text{“webern”}) = 23 \cdot 64^5 + 5 \cdot 64^4 + 2 \cdot 64^3 + 5 \cdot 64^2 + 18 \cdot 64^1 + 14 \cdot 64^0 = 24.780.493.984$

- ♦ **Assunzioni**

- ♦ $m=2^p$

- ♦ **Come calcolare $H(k)$**

- ♦ $H(k) = \text{int}(b)$, dove b è un sottoinsieme di p bit presi da $\text{bin}(k)$

- ♦ **Esempio:**

- ♦ $m=2^8=256$, bit presi dalla posizione 15 alla posizione 22

- ♦ $\text{bin}(\text{"weber\underline{b}}") = 010111\ 000101\ 00\underline{0010}\ \underline{0001}01\ 010010\ 100000$

- ♦ $\text{bin}(\text{"webern"}) = 010111\ 000101\ 00\underline{0010}\ \underline{0001}01\ 010010\ 001110$

- ♦ da cui si ottiene:

- ♦ $H(\text{"weber\underline{b}}") = \text{bin}(00100001) = 33$

- ♦ $H(\text{"webern"}) = \text{bin}(00100001) = 33$

Funzioni hash: XOR

- ♦ **Assunzioni**

- ♦ $m=2^p$

- ♦ **Come calcolare $H(k)$**

- ♦ $H(k) = \text{int}(b)$, dove b è dato dalla somma modulo 2, effettuata bit a bit, di diversi sottoinsiemi di p bit di $\text{bin}(k)$

- ♦ **Esempio:**

- ♦ $m=2^8=256$, 5 gruppi di 8 bit, 40 bit ottenuti con 4 zeri di “padding”

- ♦ $H(\text{“weberb”}) = \text{int}(01011100 \oplus 01010000 \oplus 10000101 \oplus 01001010 \oplus 00000000)$
 $= \text{int}(11000011) = 195$

- ♦ $H(\text{“webern”}) = \text{int}(01011100 \oplus 01010000 \oplus 10000101 \oplus 01001000 \oplus 11100000)$
 $= \text{int}(00100001) = 33$

Funzioni hash: metodo della divisione

- ♦ **Assunzioni:**

- ♦ m dispari, meglio se primo

- ♦ **Procedimento di calcolo**

- ♦ $H(k) = k \bmod m$

- ♦ **Esempio:**

- ♦ $m = 383$
 - ♦ $H(\text{"weber\underline{b}}") = 24.780.493.966 \bmod 383 = ?$
 - ♦ $H(\text{"webern"}) = 24.780.493.984 \bmod 383 = 242$

- ♦ **Nota: il valore m deve essere scelto opportunamente**

Funzioni hash

- ♦ **Non vanno bene:**

- ♦ $m=2^p$: solo i p bit più significativi vengono considerati
- ♦ $m=2^p-1$: permutazione di stringhe in base 2^p hanno lo stesso valore hash
 - ♦ Domanda: Dimostrazione

- ♦ **Vanno bene:**

- ♦ Numeri primi, distanti da potenze di 2 (e di 10)

Funzioni hash: Moltiplicazione

♦ Assunzioni

- ♦ m numero qualsiasi (potenze 2 consigliate)
- ♦ C una costante reale, $0 < C < 1$

♦ Procedimento di calcolo

- ♦ $i = \text{int}(\text{bin}(k))$
- ♦ $H(k) = \lfloor m(iC - \lfloor iC \rfloor) \rfloor$

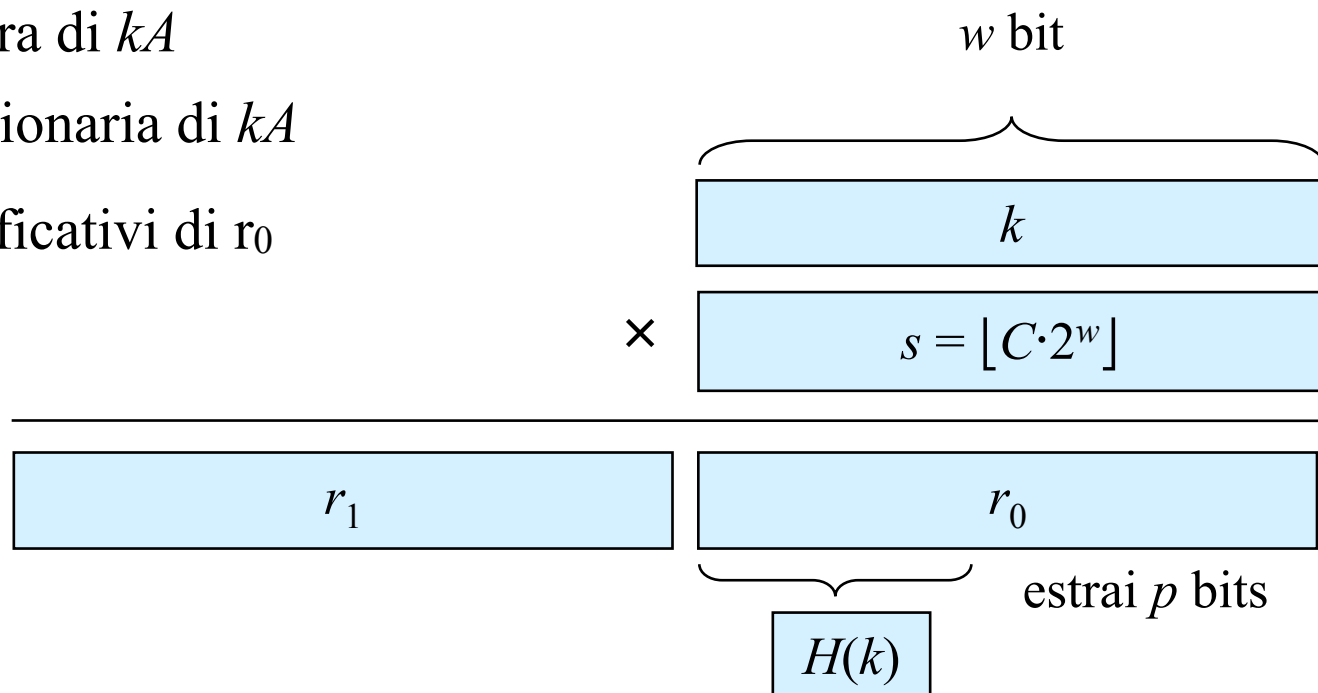
♦ Esempio

- ♦ $C = (\sqrt{5} - 1)/2$ e $m = 256$.
- ♦ $H(\text{Webern}) = \lfloor m(iC - \lfloor iC \rfloor) \rfloor = \lfloor 256 \cdot 0.9996833801 \dots \rfloor = 255$.

Funzioni hash

♦ Come implementare il metodo della moltiplicazione:

- ♦ Si scelga un valore $m=2^p$
- ♦ Sia w la dimensione in bit della parola di memoria: $k, m \leq 2^w$
- ♦ Sia $s = \lfloor C \cdot 2^w \rfloor$
- ♦ $k \cdot s$ può essere scritto come $r_1 \cdot 2^w + r_0$
 - ♦ r_1 contiene la parte intera di kA
 - ♦ r_0 contiene la parte frazionaria di kA
- ♦ Ritorniamo i p bit più significativi di r_0



Funzioni hash - continua

- ♦ **Non è poi così semplice...**
 - ♦ Il metodo della moltiplicazione suggerito da Knuth non è poi così buono....
- ♦ **Test moderni per valutare**
 - ♦ *Avalanche effect*:
 - ♦ Se si cambia un bit nella chiave, deve cambiare almeno la metà dei bit del valore hash
 - ♦ Test statistici (Chi-square)
 - ♦ Funzioni crittografiche (SHA-1)

Problema delle collisioni

- ♦ **Abbiamo ridotto, ma non eliminato, il numero di collisioni**
- ♦ **Come gestire le collisioni residue?**
 - ♦ Dobbiamo trovare collocazioni alternative per le chiavi
 - ♦ Se una chiave non si trova nella posizione attesa, bisogna andare 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:**
 - ♦ *Liste di trabocco* o memorizzazione esterna
 - ♦ *Indirizzamento aperto* o memorizzazione interna

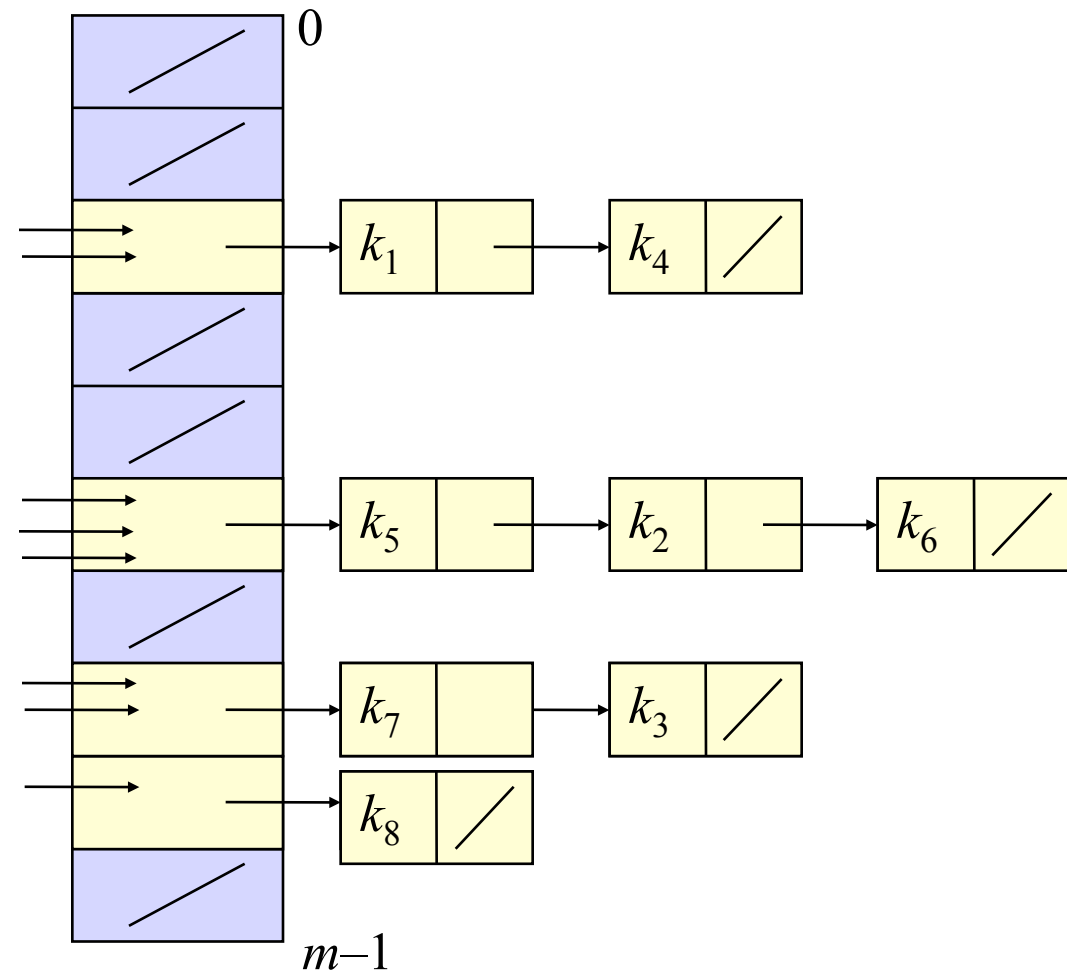
Tecniche di risoluzione delle collisioni

♦ Liste di trabocco (chaining)

- ♦ Gli elementi con lo stesso valore hash h vengono memorizzati in una lista
- ♦ Si memorizza un puntatore alla testa della lista nello slot $A[h]$ della tabella hash

♦ Operazioni:

- ♦ Insert:
inserimento in testa
- ♦ Lookup, Delete:
richiedono di scandire la lista alla ricerca della chiave



Liste di trabocco: complessità

n	=	numero di chiavi memorizzate nella tabella hash
m	=	dimensione della tabella hash
α	=	n/m (fattore di carico)
$I(\alpha)$	=	numero medio di accessi alla tabella per la ricerca di una chiave non presente nella tabella (<i>ricerca con insuccesso</i>)
$S(\alpha)$	=	numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella (<i>ricerca con successo</i>)

♦ Analisi del caso pessimo:

- ♦ Tutte le chiavi sono collocate in unica lista
 - ♦ Insert: $\Theta(1)$
 - ♦ Search, Delete: $\Theta(n)$

♦ Analisi del caso medio:

- ♦ Dipende da come le chiavi vengono distribuite
- ♦ Assumiamo hashing uniforme semplice
- ♦ Costo funzione di hashing f : $\theta(1)$

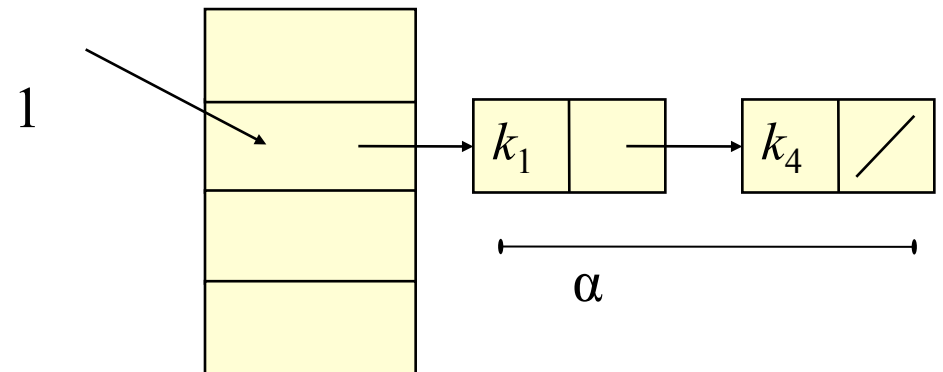
Liste di trabocco: complessità

♦ Teorema:

- ♦ In tavola hash con concatenamento, una ricerca senza successo richiede un tempo atteso $\Theta(1 + \alpha)$

♦ 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: $\alpha \rightarrow \Theta(1 + \alpha)$



Liste di trabocco: complessità

♦ Teorema:

- ♦ In tavola hash con concatenamento, una ricerca con successo richiede un tempo atteso di $\Theta(1 + \alpha)$
- ♦ Più precisamente: $\Theta(1 + \alpha/2)$

♦ Dimostrazione: idee chiave

- ♦ Si assuma che l'elemento cercato k sia uno qualsiasi degli n elementi presenti nella tabella
- ♦ Il numero di elementi esaminati durante una ricerca con successo:
 - ♦ 1 (l'elemento cercato) +
 - ♦ in media, dovrò scandire metà della lista (di lunghezza attesa α)

Liste di trabocco: complessità

- ♦ **Qual è il significato del fattore di carico:**
 - ♦ Influenza il costo computazionale delle operazioni sulle tabelle hash
 - ♦ 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 **nil**
 - ♦ 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

- ♦ **Ispezione:** Uno slot esaminato durante una ricerca di chiave
- ♦ **Sequenza di ispezione:** La lista ordinata degli slot esaminati
- ♦ **Funzione hash:** estesa come
 - ♦ $H : U \times [0 \dots m-1] \rightarrow [0 \dots m-1]$
 - ♦

<i>n. sequenza</i>	<i>indice array</i>
--------------------	---------------------
- ♦ **La sequenza di ispezione $\{ H(k, 0), H(k, 1), \dots, H(k, m-1) \}$ è una permutazione degli indici $[0 \dots m-1]$**
 - ♦ Può essere necessario esaminare ogni slot nella tabella
 - ♦ Non vogliamo esaminare ogni slot più di una volta

Indirizzamento aperto

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

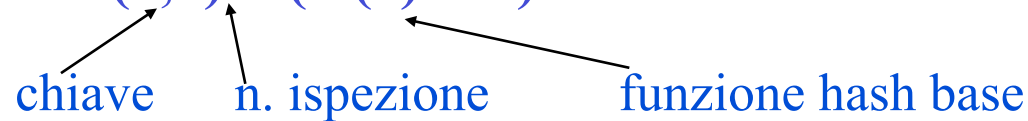
Tecniche di ispezione

- ♦ **La situazione ideale prende il nome di *hashing uniforme***
 - ♦ Ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una qualsiasi delle $m!$ permutazioni di $[0...m-1]$
 - ♦ Generalizzazione dell'hashing uniforme semplice
- ♦ **Nella realtà:**
 - ♦ E' difficile implementare il vero uniform hashing
 - ♦ Ci si accontenta di ottenere semplici permutazioni
- ♦ **Tecniche diffuse:**
 - ♦ Ispezione lineare
 - ♦ Ispezione quadratica
 - ♦ Doppio hashing

Ispezione lineare

- ♦ **Funzione:** $H(k, i) = (H(k) + h \cdot i) \bmod m$

chiave n. ispezione funzione hash base



- ♦ **Il primo elemento determina l'intera sequenza**

- ♦ $H(k), H(k)+h, H(k)+2 \cdot h \dots, H(k)+(m-1) \cdot h$ (tutti modulo m)

- ♦ Solo m sequenze di ispezione distinte sono possibili

- ♦ **Problema: agglomerazione primaria (primary clustering)**

- ♦ Lunghe sotto-sequenze occupate...

- ♦ ... che tendono a diventare più lunghe:

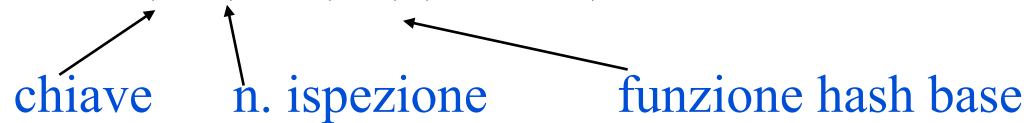
- ♦ uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i+1)/m$

- ♦ I tempi medi di inserimento e cancellazione crescono

Ispezione quadratica

- ♦ **Funzione:** $H(k, i) = (H(k) + h \cdot i^2) \bmod m$

chiave n. ispezione funzione hash base



- ♦ **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
- ♦ Solo m sequenze di ispezione distinte sono possibili

- ♦ **Problema: la sequenza così risultante non è una permutazione**

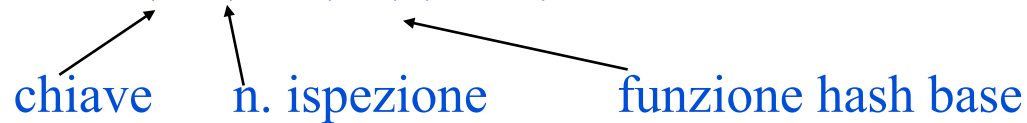
- ♦ **Problema: agglomerazione secondaria (secondary clustering)**

- ♦ Se due chiavi hanno la stessa ispezione iniziale, le loro sequenze sono identiche

Ispezione pseudo-casuale

- ♦ **Funzione:** $H(k, i) = (H(k) + r_i) \bmod m$

chiave n. ispezione funzione hash base



- ♦ **Sequenza di ispezioni:**

- ♦ L'ispezione iniziale è in $H(k)$
- ♦ r_i è l' i -esimo elemento restituito da un generatore di numeri casuali fra $[0 \dots m-1]$
- ♦ Solo m sequenze di ispezione distinte sono possibili

- ♦ **La sequenza così risultante è una permutazione**

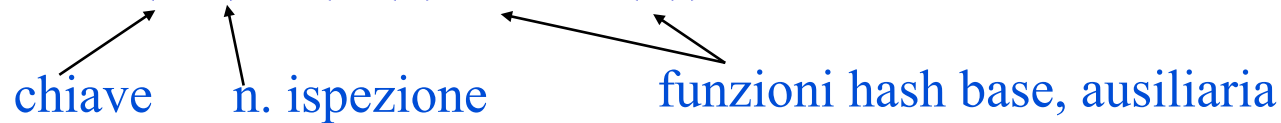
- ♦ **Problema: agglomerazione secondaria (secondary clustering)**

- ♦ Questo problema rimane

Doppio hashing

- ♦ **Funzione:** $H(k, i) = (H(k) + i \cdot H'(k)) \bmod m$

chiave n. ispezione funzioni hash base, ausiliaria



- ♦ **Due funzioni ausiliarie:**

- ♦ H fornisce la prima ispezione
- ♦ H' fornisce l'offset delle successive ispezioni
- ♦ m^2 sequenze di ispezione distinte sono possibili

- ♦ **Nota:** Per garantire una permutazione completa, $H'(k)$ deve essere relativamente primo con m

- ♦ Scegliere $m = 2^p$ e $H'(k)$ deve restituire numeri dispari
- ♦ Scegliere m primo, e $H'(k)$ deve restituire numeri minori di m

Cancellazione

- ✦ **Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un nil. Perché?**
- ✦ **Approccio**
 - ✦ Utilizziamo un speciale valore **deleted** al posto di **nil** per marcare uno slot come vuoto dopo la cancellazione
 - ✦ Ricerca: **deleted** trattati come slot pieni
 - ✦ Inserimento: **deleted** trattati come slot vuoti
- ✦ **Svantaggio: il tempo di ricerca non dipende più da α .**
- ✦ **Concatenamento più comune se si ammettono cancellazioni**

Codice - Hashing doppio

HASH

ITEM[] A

% Tabella delle chiavi

ITEM[] V

% Tabella dei valori

integer m

% Dimensione della tabella

HASH Hash(**integer** $capacità$)

 HASH $t = \text{new HASH}$

$t.m \leftarrow capacità$

$t.A \leftarrow \text{new Item}[0 \dots t.m - 1]$

$t.V \leftarrow \text{new Item}[0 \dots t.m - 1]$

for $i \leftarrow 0$ **to** $t.m - 1$ **do** $t.A[i] \leftarrow \text{nil}$

return t

Codice - Hashing doppio

integer scan(ITEM k , boolean $insert$)

integer $c \leftarrow m$

integer $i \leftarrow 0$

integer $j \leftarrow H(k)$

while $A[j] \neq k$ **and** $A[j] \neq \text{nil}$ **and** $i < m$ **do**

if $A[j] = \text{deleted}$ **and** $c = m$ **then** $c \leftarrow j$

$j \leftarrow (j + H'(k)) \bmod m$

$i \leftarrow i + 1$

if $insert$ **and** $A[j] \neq k$ **and** $c < m$ **then** $j = c$

return j

% Prima posizione *deleted*

% Numero di ispezione

% Posizione attuale

Codice - Hashing doppio

ITEM lookup(ITEM k)

integer $i \leftarrow \text{scan}(k, \text{false})$

if $A[i] = k$ **then**

 | **return** $V[i]$

else

 | **return** nil

insert(ITEM k , ITEM v)

integer $i \leftarrow \text{scan}(k, \text{true})$

if $A[i] = \text{nil}$ **or** $A[i] = \text{deleted}$ **or** $A[i] = k$ **then**

 | $A[i] \leftarrow k$

 | $V[i] \leftarrow v$

else

 | % Errore: tabella hash piena

remove(ITEM k)

integer $i \leftarrow \text{scan}(k, \text{false})$

if $A[i] = k$ **then**

 | $A[i] \leftarrow \text{deleted}$

Indirizzamento aperto: Complessità hashing doppio

♦ Assunzioni

- ♦ Hashing uniforme
- ♦ Nessuna cancellazione
- ♦ Nella ricerca con successo, tutte le chiavi hanno la stessa probabilità di essere cercate

♦ Analisi

- ♦ n chiavi inserite in una tabella di m slot
- ♦ $n < m$, ovvero il fattore di carico $\alpha < 1$
- ♦ Analisi basata sul valore di α

Indirizzamento aperto: Complessità hashing doppio

- ♦ **Il numero atteso di ispezioni $I(\alpha)$ per una ricerca senza successo è $O(1/(1-\alpha))$**
 - ♦ Esempio: $\alpha=0.5$, $I(\alpha)=2$; $\alpha=0.9$, $I(\alpha)=10$
 - ♦ A cosa tende questo valore per $\alpha \rightarrow 1$?
- ♦ **Il numero atteso di ispezioni $S(\alpha)$ per una ricerca con successo è $O(-1/\alpha \ln(1-\alpha))$**
 - ♦ Esempio: $\alpha=0.5$, $S(\alpha)=1.387$; $\alpha=0.9$, $S(\alpha)=2.559$

Complessità - Riassunto

SCANSIONE	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \leq \alpha < 1$	$\frac{(1 - \alpha)^2 + 1}{2(1 - \alpha)^2}$	$\frac{1 - \alpha/2}{1 - \alpha}$
Hashing doppio	$0 \leq \alpha < 1$	$\frac{1}{1 - \alpha}$	$-\frac{1}{\alpha} \ln(1 - \alpha)$
Liste di trabocco	$\alpha \geq 0$	$1 + \alpha$	$1 + \alpha/2$

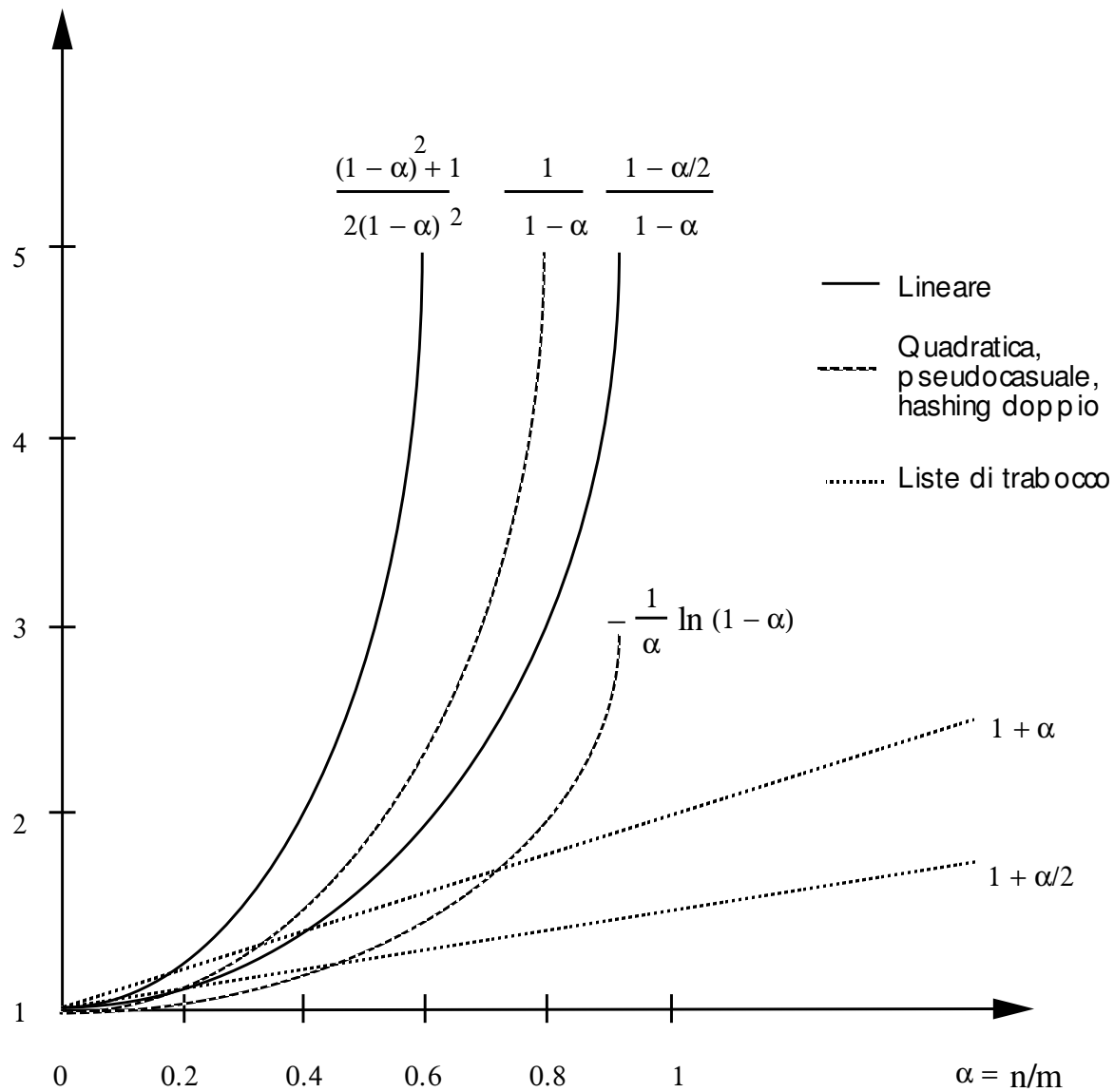


Figura 7.3: Numero medio di accessi per ricerche in tabelle hash.

Ristrutturazione

- ♦ Usando la scansione interna, non è conveniente che α cresca troppo
- ♦ Sopra un soglia prefissata (tipicamente 0.5)
 - ♦ Si alloca una nuova tabella di dimensione $2m$
 - ♦ Si reinseriscono tutte le chiavi presenti nella nuova tabella
- ♦ Risultato
 - ♦ Fattore di carico dimezzato (tipicamente 0.25)
 - ♦ Nessun elemento **deleted**
- ♦ Costi
 - ♦ Costo $O(m)$ per la ristrutturazione nel caso pessimo
 - ♦ Costo ammortizzato costante (vedi dimostrazione per vettori)

- ♦ **Problemi con hashing**

- ♦ Scarsa “locality of reference” (cache miss)
- ♦ Non è possibile ottenere le chiavi in ordine

- ♦ **Implementazione**

- ♦ HashMap in Java
- ♦ `hash_map` della Standard Template Library del C++

- ♦ **Un'interessante estensione**

- ♦ Distributed Hash Table (DHT)