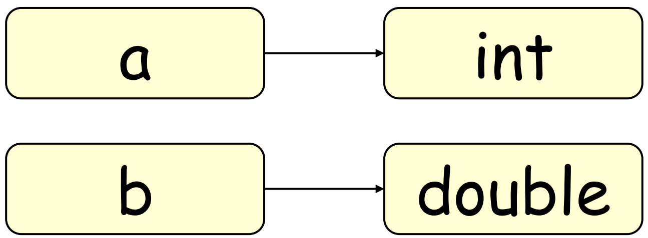
Algoritmi e Strutture Dati

Capitolo 7 - Tabelle hash

Alberto Montresor
Università di Trento

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



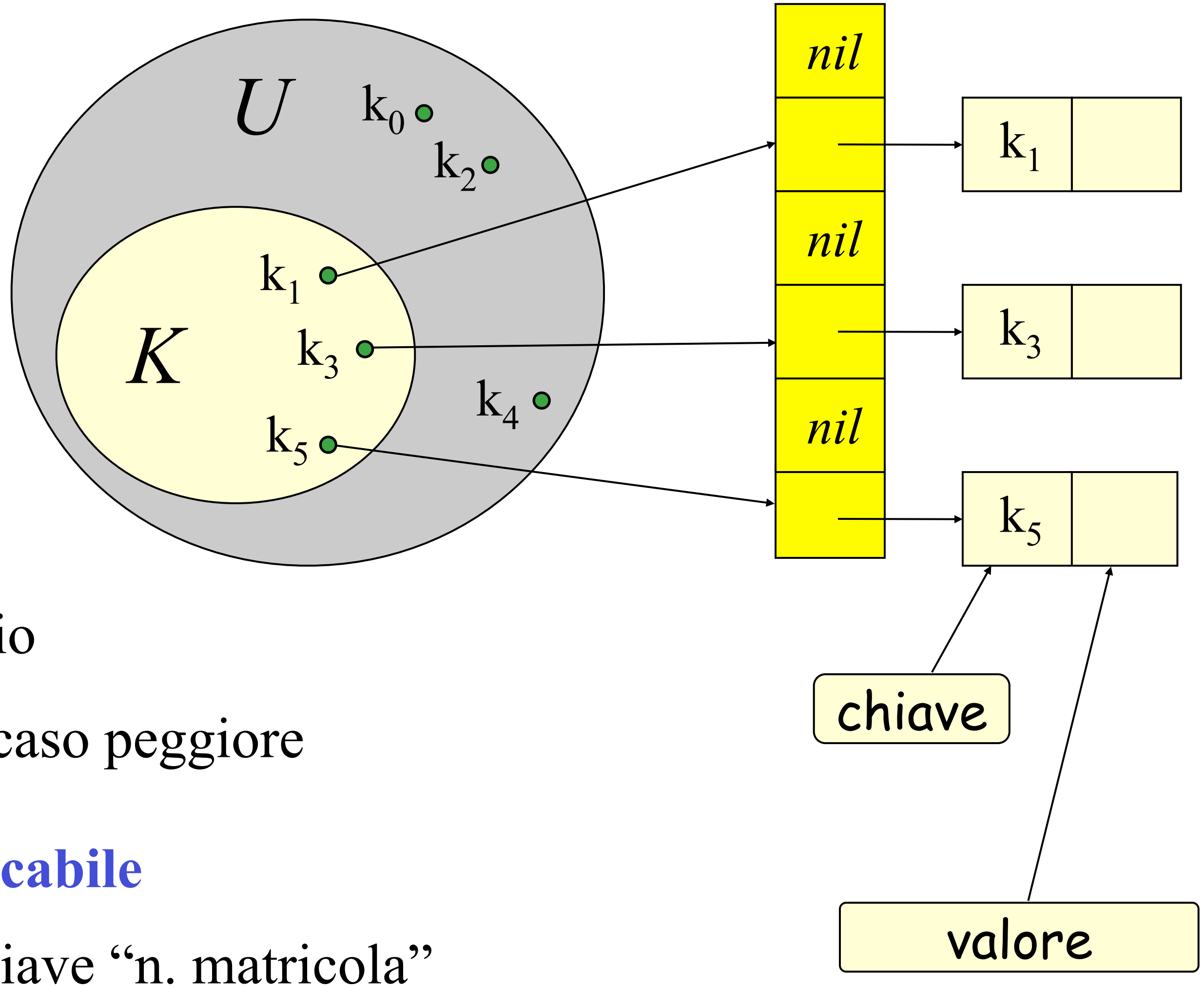
- **Possibili implementazioni e relativi costi**

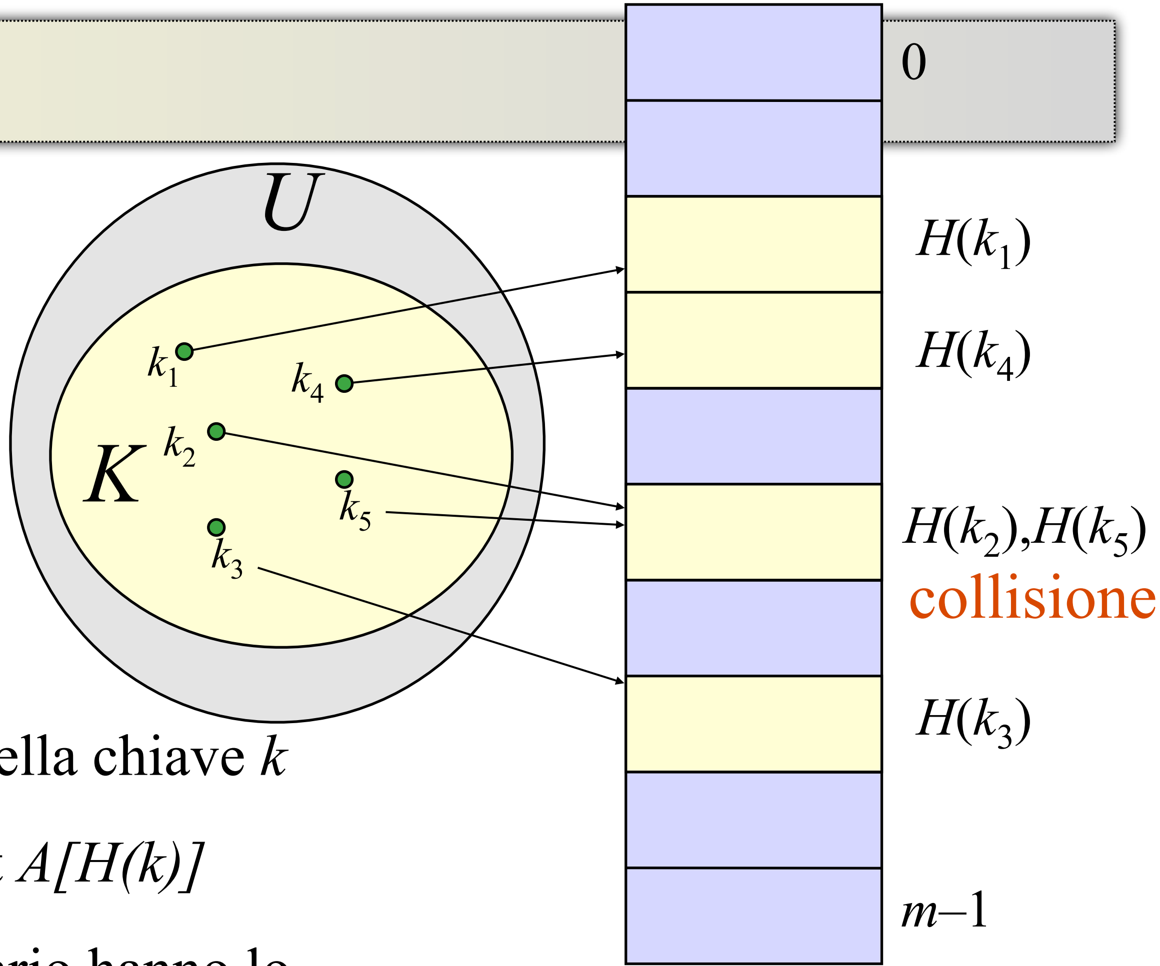
	Array non ordinato	Array ordinato	Lista	Alberi (abr, rb, ...)	Performance ideale
insert()	`O(1)`	`O(n)`	`O(1)`	`O(log n)`	`O(1)`
looup()	`O(n)`	`O(log n)`	`O(n)`	`O(log n)`	`O(1)`
remove()	`O(n)`	`O(n)`	`O(n)`	`O(log n)`	`O(1)`

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, $|K| << |U|$ $\rightarrow$
    - tabelle hash

- **Implementazione:**
  - Basata su array ordinari
  - L'elemento con chiave *k* è memorizzato nel *k*-esimo "slot"del vettore
- **Se |*K*| ~ |*U*|:**
  - Non sprechiamo (troppo) spazio
  - Operazioni in tempo *O(1)* nel caso peggiore
- **Se |*K*| << |*U*|: soluzione non praticabile**
  - Esempio: studenti ASD con chiave "n. matricola"





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

$$\forall i \in \{0, \ldots, m-1\} : Q(i) = 1/m.$$

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"

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
    - 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*("DOG") → *ord*('D') *ord*("O") *ord*("G")
→ 01000100 01001111 01000111
      - *int*("DOG") → $68 \cdot 256^2 + 79 \cdot 256 + 71$
→ 4.476.743

Come realizzare una funzione hash

Nei prossimi esempi, utilizziamo codice ASCII a 8 bit

$$
\begin{aligned}
bin(\text{“}\mathtt{DOG}\text{”}) &= ord(\text{“}\mathtt{D}\text{”}) \quad ord(\text{“}\mathtt{O}\text{”}) \quad ord(\text{“}\mathtt{G}\text{”}) \\
&= \mathtt{01000100} \quad \mathtt{01001111} \quad \mathtt{01000111} \\
int(\text{“}\mathtt{DOG}\text{”}) &= 68 \cdot 256^2 + \quad 79 \cdot 256 + \quad 71 \\
&= 4,476,743
\end{aligned}
$$

Domanda: come fate a trasformare questa sequenza di bit o questo numero in un valore compreso in $[0, m-1]$?

Funzione hash - Estrazione

Estrazione

- $m = 2^p$
- $H(k) = int(b)$, dove b è un sottoinsieme di p bit presi da $bin(k)$

Problemi

- Selezionare bit presi dal suffisso della chiave può generare collisioni con alta probabilità
- Tuttavia, anche prendere parti diverse dal suffisso o dal prefisso può generare collisioni.

Funzioni hash

- **Nei prossimi esempi**
  - *ord*(`a`) = 1, *ord*(`b`)=2, ..., *ord*(`z`)=26, *ord*(`b̲`)=32
    - b̲ rappresenta lo spazio
  - Sono sufficienti 6 bit per rappresentare questi caratteri
- **Si considerino le seguente due stringhe:** "`weberb̲`" **e** "`webern`"
- **Rappresentazione binaria**
  - *bin*("`weberb̲`") = 010111 000101 000010 000101 010010 100000
  - *bin*("`webern`") = 010111 000101 000010 000101 010010 001110
- **Rappresentazione intera**
  - *int*("`weberb̲`") = $23 \cdot 64^5 + 5 \cdot 64^4 + 2 \cdot 64^3 + 5 \cdot 64^2 + 18 \cdot 64^1 + 32 \cdot 64^0$ = 24.780.493.966
  - *int*("`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

Funzioni hash - Estrazione

- **Assunzioni**
  - $m=2^p$
- **Come calcolare H(k)**
  - $H(k) = int(b)$, dove b è un sottoinsieme di p bit presi da $bin(k)$
- **Esempio:**
  - $m = 2^8 = 256$, bit presi dalla posizione 15 alla posizione 22
    - *bin*("`weberb`") = 010111 000101 00<u>0010 0001</u>01 010010 100000
    - *bin*("`webern`") = 010111 000101 00<u>0010 0001</u>01 010010 001110
  - da cui si ottiene:
    - *H*("`weberb`") = *bin*(00100001) = 33
    - *H*("`webern`") = *bin*(00100001) = 33

Funzioni hash: XOR

- **Assunzioni**
  - $m=2^p$
- **Come calcolare H(k)**
  - $H(k) = int(b)$, dove b è dato dalla somma modulo 2, effettuata bit a bit, di diversi sottoinsiemi di p bit di $bin(k)$
- **Esempio:**
  - $m = 2^8 = 256$, 5 gruppi di 8 bit, 40 bit ottenuti con 4 zeri di "padding"
    - $H($"`weberb`"$)$ $= int(01011100 \oplus 01010000 \oplus 10000101 \oplus 01001010 \oplus 0000\underline{0000})$
      $= int(11000011) = 195$
    - $H($"`webern`"$)$ $= int(01011100 \oplus 01010000 \oplus 10000101 \oplus 01001000 \oplus 1110\underline{0000})$
      $= int(00100001) = 33$

Funzioni hash: metodo della divisione

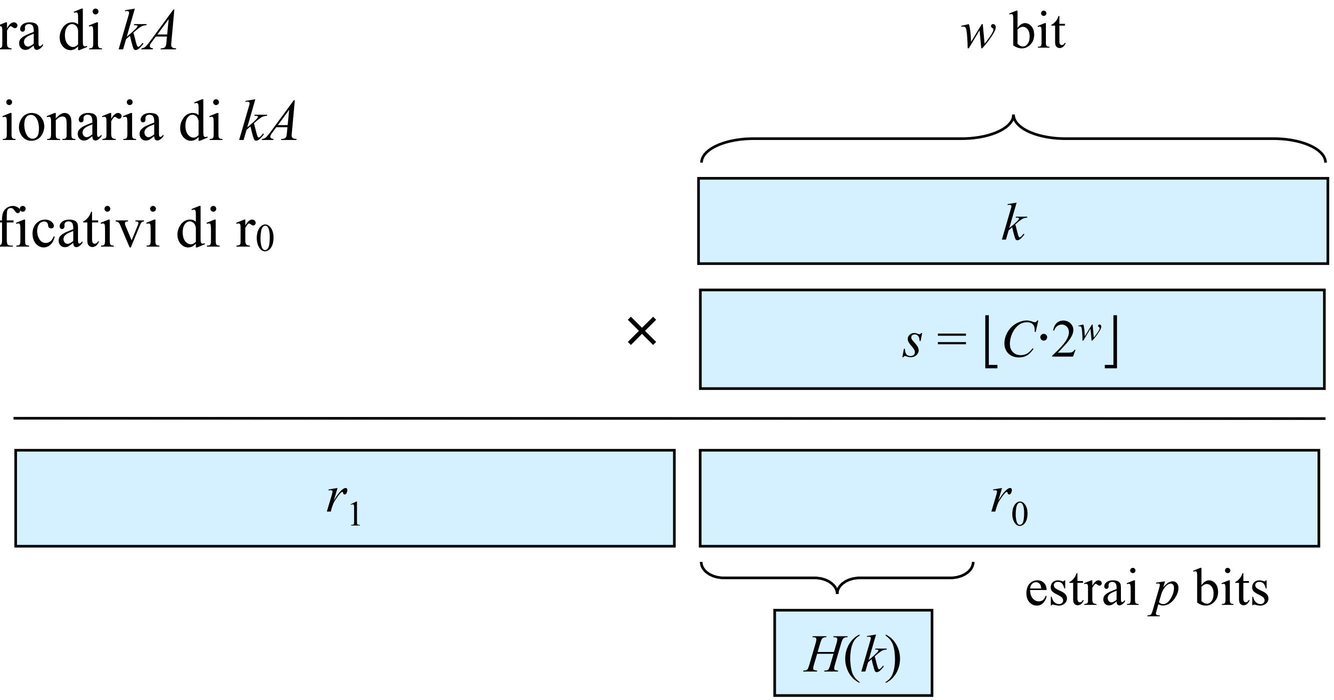
- **Assunzioni:**
  - m dispari, meglio se primo
- **Procedimento di calcolo**
  - $H(k) = k \bmod m$
- **Esempio:**
  - m =383
    - H(“`weberb`”) = 24.780.493.966 mod 383 = ?
    - H(“`webern`”) = 24.780.493.984 mod 383 = 242
- **Nota: il valore *m* deve essere scelto opportunamente**

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)

- **Assunzioni**
  - m numero qualsiasi (potenze 2 consigliate)
  - C una costante reale, $0 < C < 1$
- **Procedimento di calcolo**
  - $i = int(bin(k))$
  - $H(k) = \lfloor m(iC - \lfloor iC \rfloor) \rfloor$
- **Esempio**
  - $C = (\sqrt{5} - 1)/2$ e $m = 256$.
  - $H(\texttt{Webern}) = \lfloor m(iC - \lfloor iC \rfloor) \rfloor = \lfloor 256 \cdot 0.9996833801 \ldots \rfloor = 255$.

- **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)

Funzioni hash moderne

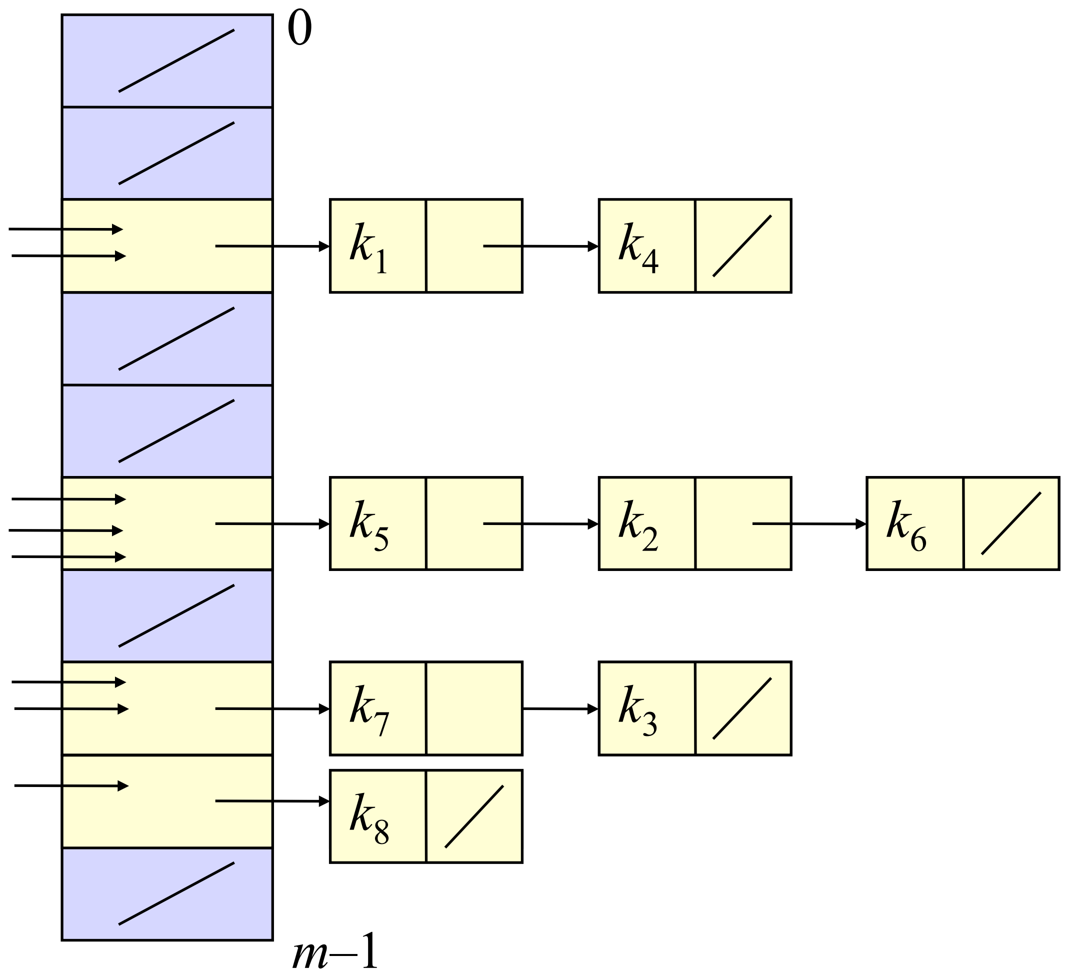
Nome	**Note**	**Link**
FNV Hash	Funzione hash non crittografica, creata nel 1991.	[Wikipedia] [Codice]
Murmur Hash	Funzione hash non crittografica, creata nel 2008, il cui uso è ormai sconsigliato perchè debole.	[Wikipedia] [Codice]
City Hash	Una famiglia di funzioni hash non-crittografiche, progettate da Google per essere molto veloce. Ha varianti a 32, 64, 128, 256 bit.	[Wikipedia] [Codice]
Farm Hash	Il successore di City Hash, sempre sviluppato da Google.	[Codice]

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



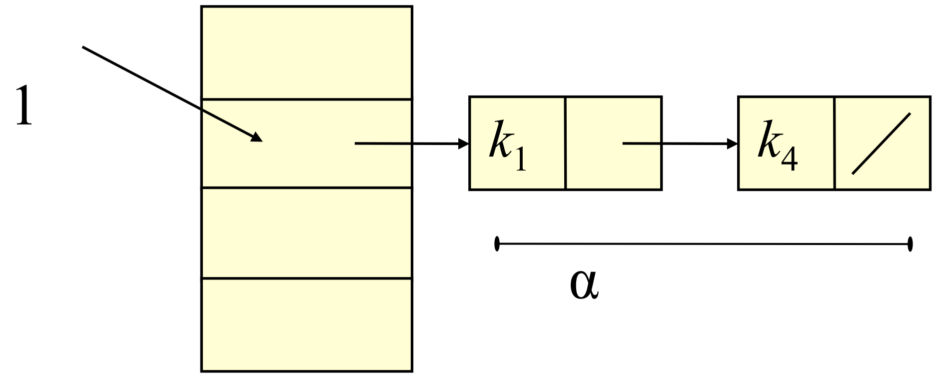
Liste di trabocco: complessità

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 (*ricerca con insuccesso*)
$S(\alpha)$	$=$	numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella (*ricerca con successo*)

- **Analisi del caso pessimo:**
  - Tutte le chiavi sono collocate in unica lista
    - Insert: *Θ(1)*
    - Search, Delete: *Θ(n)*
- **Analisi del caso medio:**
  - Dipende da come le chiavi vengono distribuite
  - Assumiamo hashing uniforme semplice
  - Costo funzione di hashing f: *θ(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**
  - $H : U \times [0 \ldots m\text{-}1] \rightarrow [0 \ldots m\text{-}1]$
- *n. sequenza* *indice array*
- **La sequenza di ispezione { $H(k, 0)$, $H(k, 1)$, …, $H(k, m{-}1)$ } 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

	$\mathbf{k_1}$		$\mathbf{k_2}$	$\mathbf{k_3}$	$\mathbf{k_4}$		$\mathbf{k_5}$			

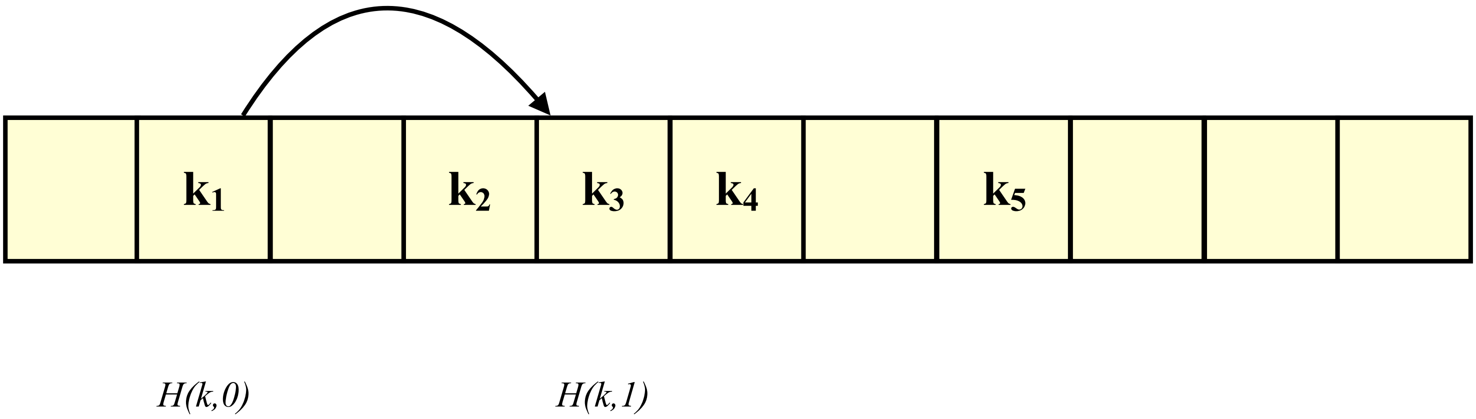
$H(k,0)$

Esempio

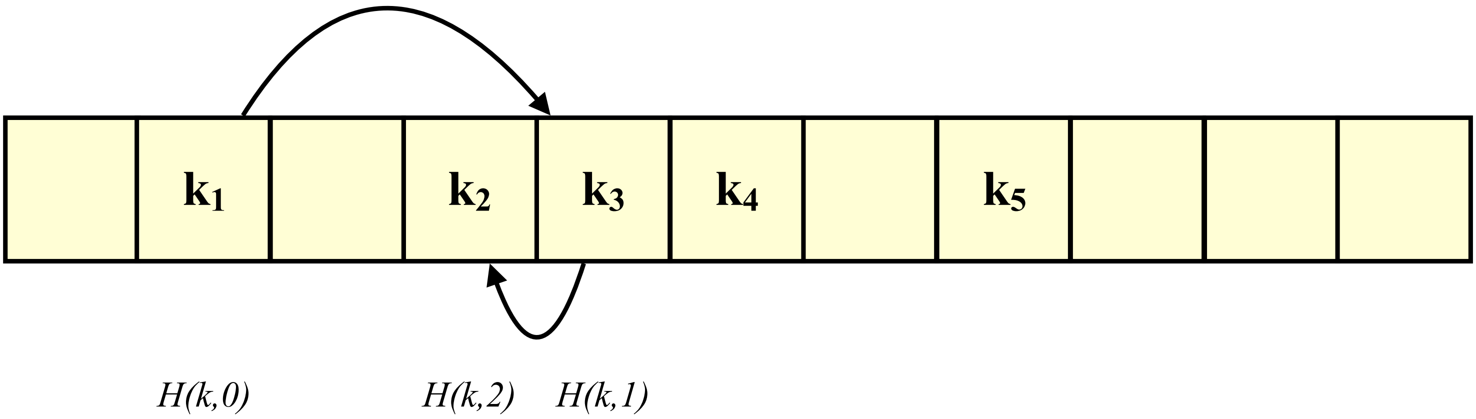
	k_1		k_2	k_3	k_4		k_5			

H(k,0)

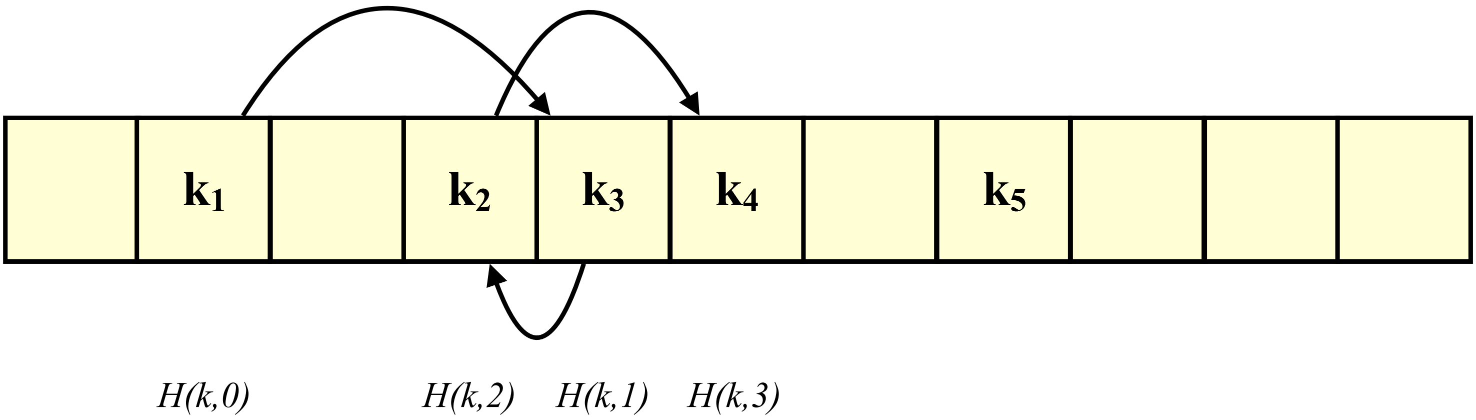
Esempio



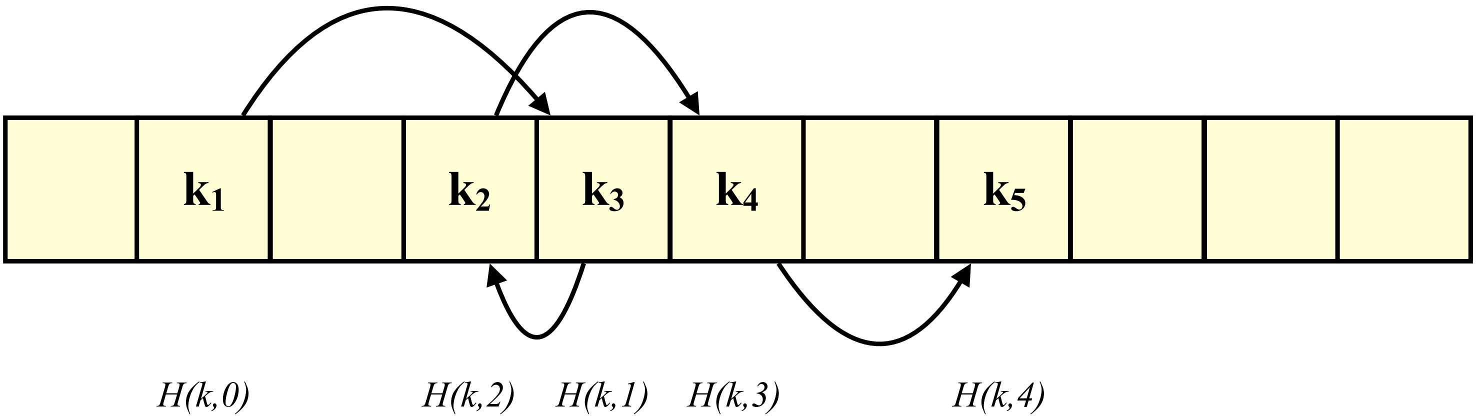
Esempio



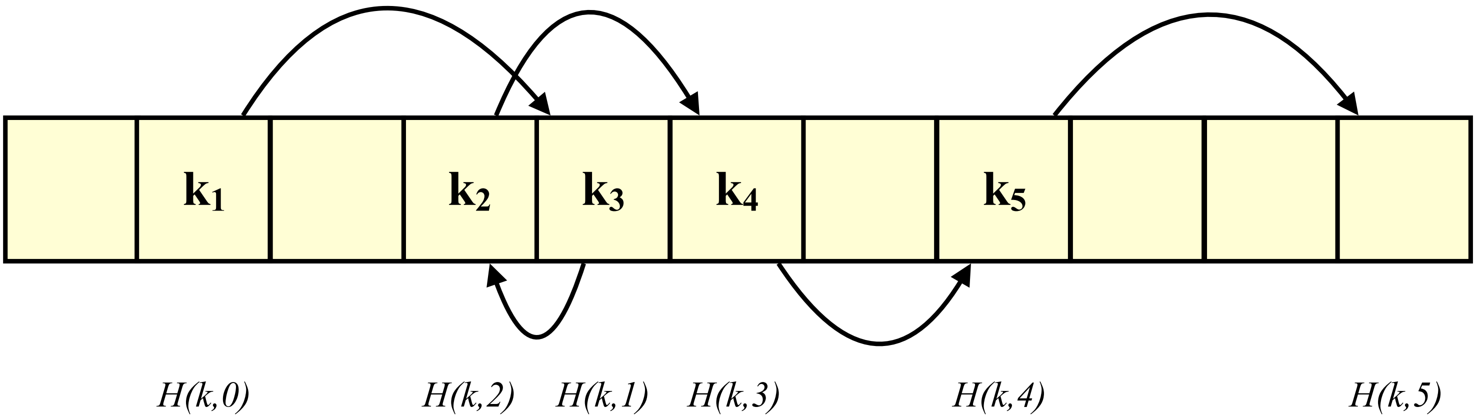
Esempio



Esempio



Esempio



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

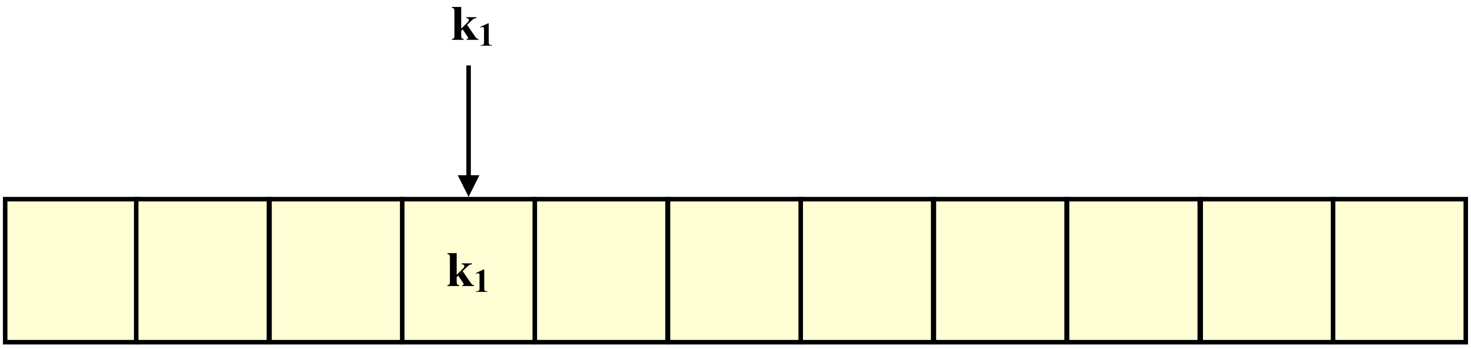
- **Il primo elemento determina l'intera sequenza**
  - $H(k), H(k)+h, H(k)+2 \cdot h \ldots, H(k)+(m-1) \cdot h$ (tutti modulo m)
  - Solo m sequenze di ispezione distinte sono possibili

- **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

Agglomerazione primaria

Agglomerazione primaria (primary clustering)

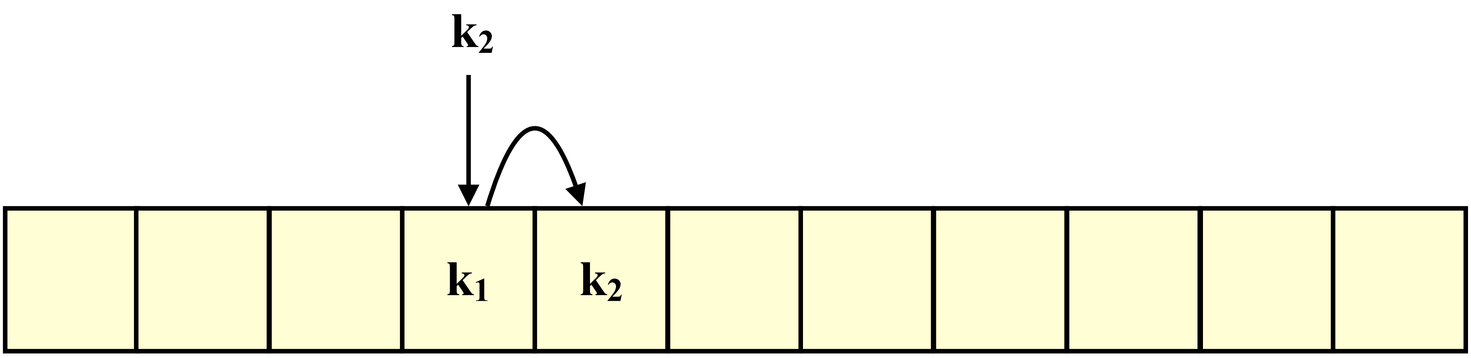
- Lunghe sotto-sequenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

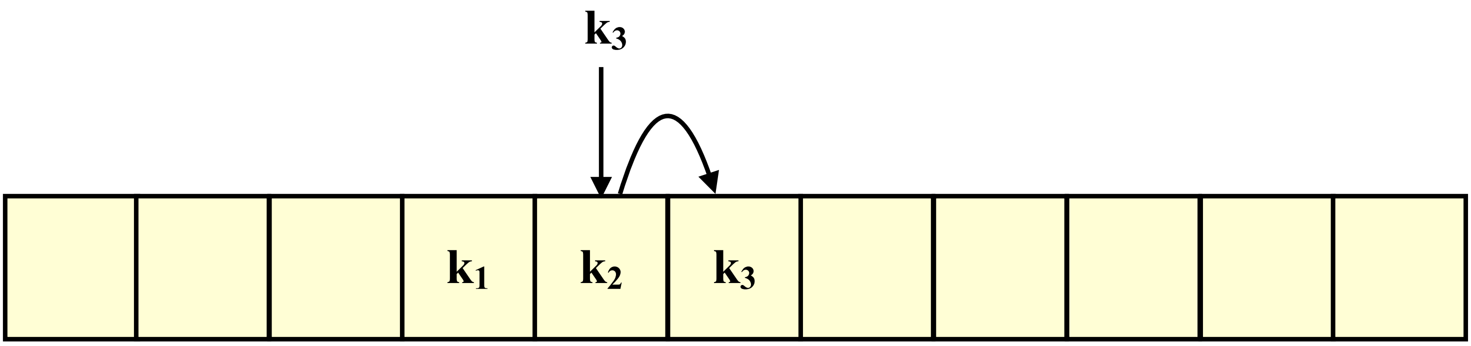
- Lunghe sotto-sequenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

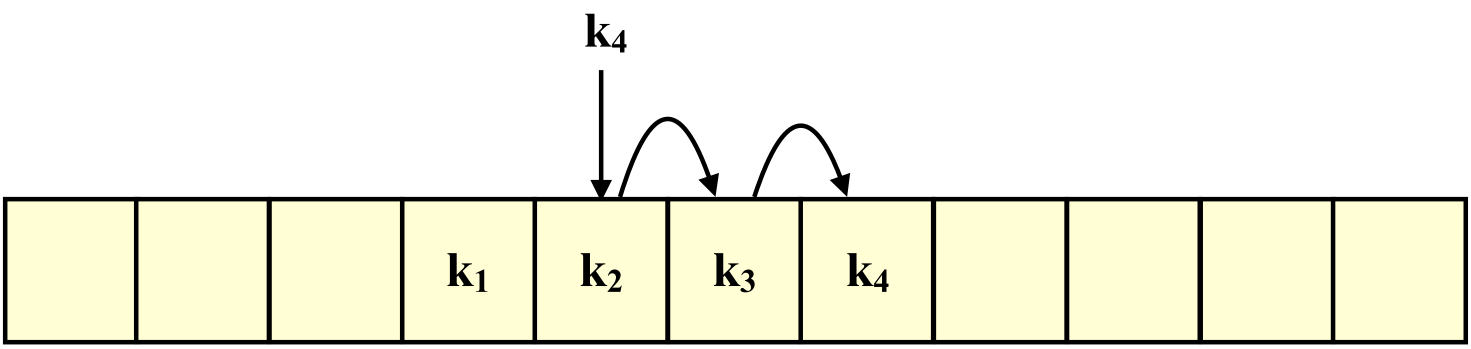
- Lunghe sotto-sequenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i+1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

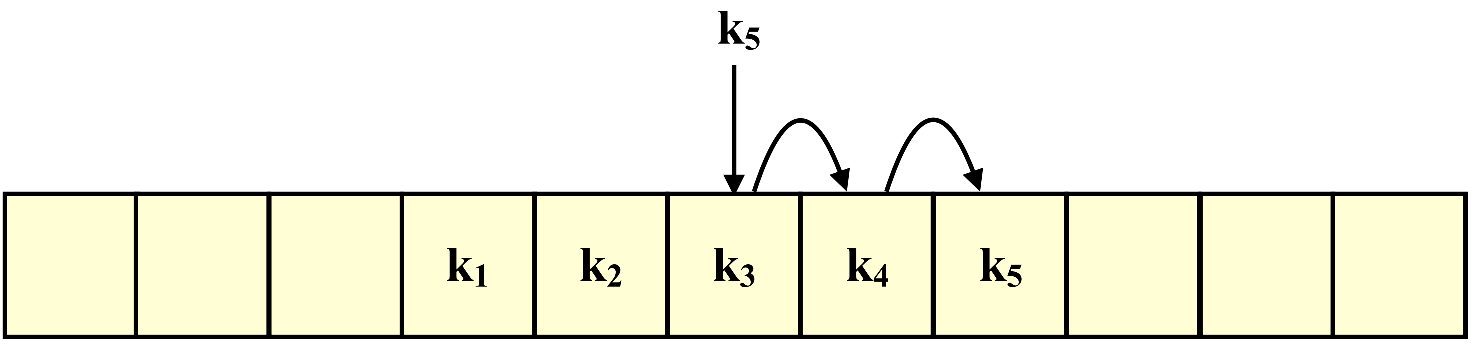
- Lunghe sotto-sequenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i+1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

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 $[0 \ldots m-1]$
  - 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?**

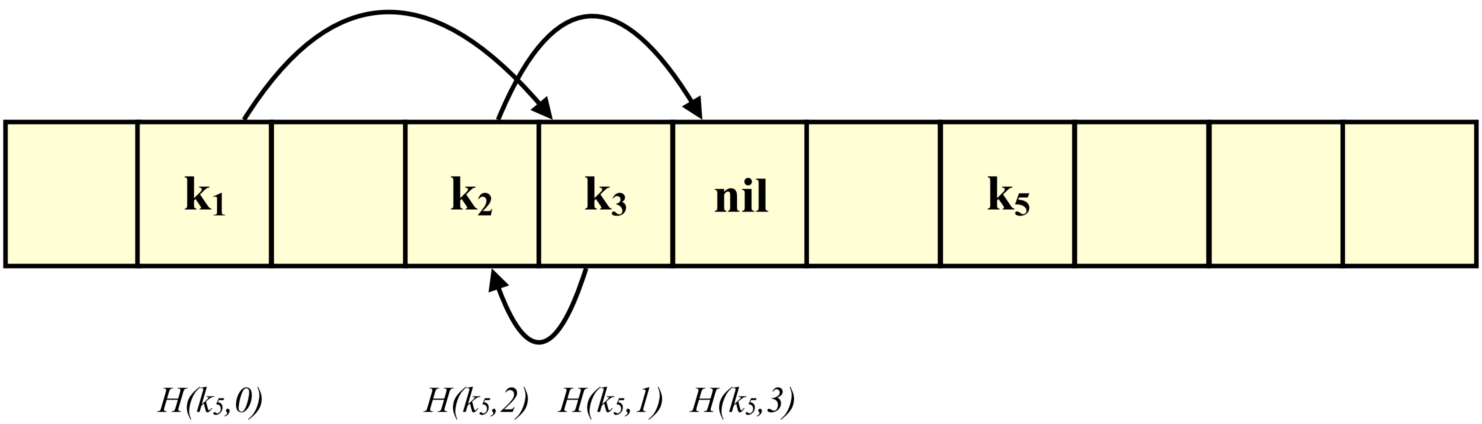
- **Approccio**
  - Utilizziamo un speciale valore **deleted** al posto di **nil** per marcare uno slot come vuoto dopo la cancellazione
  - Ricerca: **deleted** trattati come slot pieni
  - Inserimento: **deleted** trattati come slot vuoti

- **Svantaggio: il tempo di ricerca non dipende più da α.**

- **Concatenamento più comune se si ammettono cancellazioni**

Cancellazione

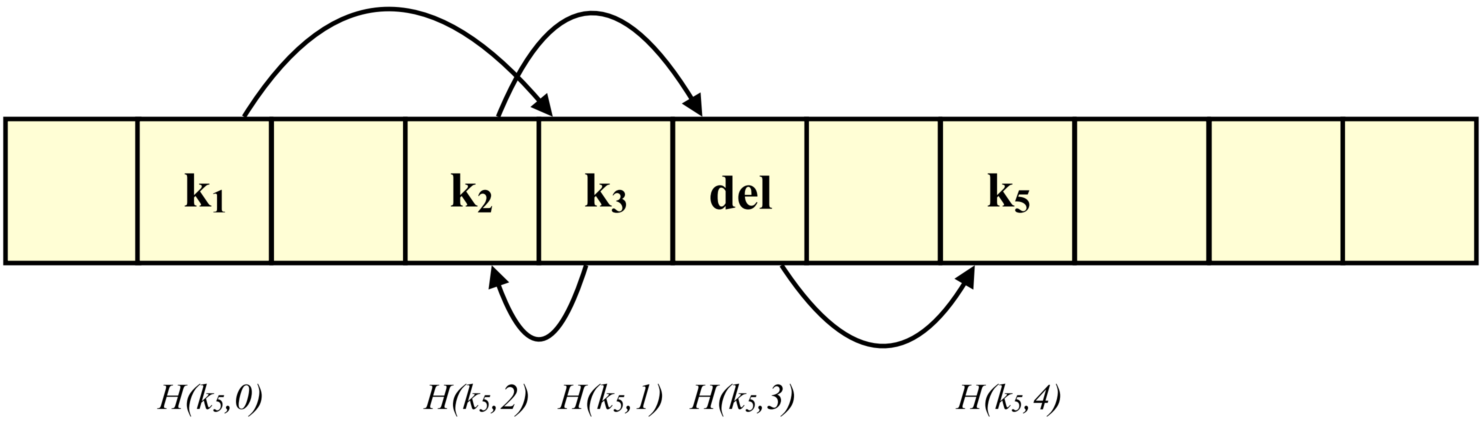
Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un **nil**. Perché?



Cancellazione

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



Implementazione - Hashing doppio

```
HASH

ITEM[] K                                   % Tabella delle chiavi
ITEM[] V                                   % Tabella dei valori
int m                                      % Dimensione della tabella

HASH Hash(int dim)
    HASH t = new HASH
    t.m = dim
    t.K = new ITEM[0 ... dim − 1]
    t.V = new ITEM[0 ... dim − 1]
    for i = 0 to dim − 1 do
        t.K[i] = nil
    return t
```

Implementazione - Hashing doppio

```
int scan(ITEM k, boolean insert)
    int delpos = m                                    % Prima posizione deleted
    int i = 0                                         % Numero di ispezione
    int j = H(k)                                      % Posizione attuale
    while K[j] ≠ k and K[j] ≠ nil and i < m do
        if K[j] == deleted and delpos == m then
            delpos = j
        j = (j + H'(k)) mod m
        i = i + 1
    if insert and K[j] ≠ k and delpos < m then
        return delpos
    else
        return j
```

Implementazione - Hashing doppio

```
ITEM lookup(ITEM k)
    int i = scan(k, false)
    if K[i] == k then
        return V[i]
    else
        return nil

insert(ITEM k, ITEM v)
    int i = scan(k, true)
    if K[i] == nil or K[i] == deleted or K[i] == k then
        K[i] = k
        V[i] = v
    else
        % Errore: tabella hash piena
```

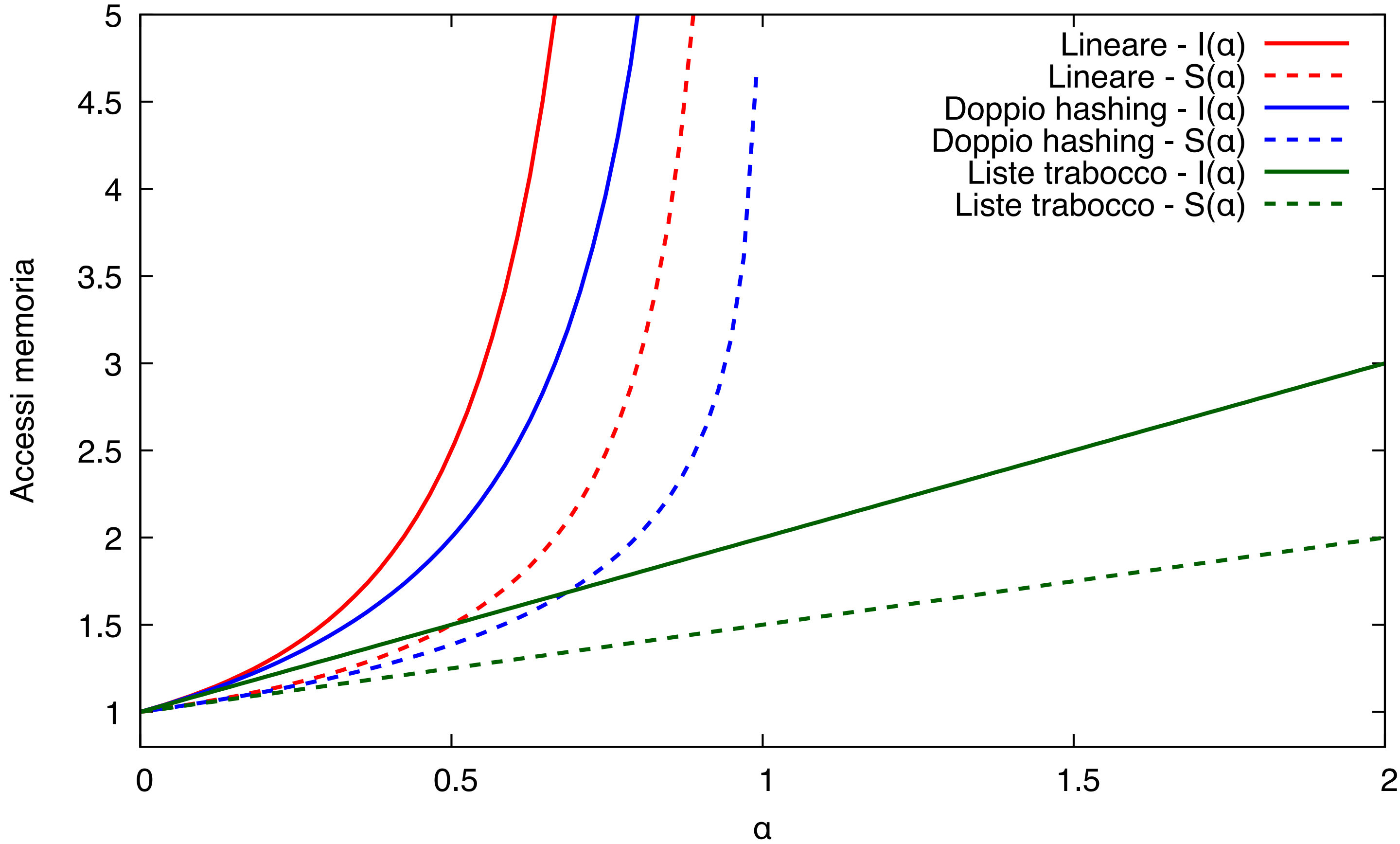
Implementazione - Hashing doppio

```
remove(Item k)
    int i = scan(k, false)
    if K[i] == k then
        K[i] = deleted
        V[i] = nil
```

Complessità

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

Complessità



Java `hashcode()`

Esempio: `java.lang.String`

- Override di `equals()` per controllare l'uguaglianza di stringhe
- `hashCode()` in Java 1.0, Java 1.1
  - Utilizzati 16 caratteri della stringa per calcolare l'`hashCode()`
  - Problemi con la regola (3) - cattiva performance nelle tabelle
- `hashCode()` in Java 1.2 e seguenti:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

(utilizzando aritmetica `int`)

Java hashcode()

Cosa non fare

```
public int hashCode()
{
  return 0;
}
```

Reality check

Linguaggio	**Tecnica**	t_α	**Note**
Java 7 `HashMap`	Liste di trabocco basate su `LinkedList`	0.75	$O(n)$ nel caso pessimo Overhead: $16n + 4m$ byte
Java 8 `HashMap`	Liste di trabocco basate su RB Tree	0.75	$O(\log n)$ nel caso pessimo Overhead: $48n + 4m$ byte
C++ `sparse_hash`	Ind. aperto, scansione quadratica	?	Overhead: $2n$ bit
C++ `dense_hash`	Ind. aperto, scansione quadratica	0.5	X byte per chiave-valore $\Rightarrow$ 2-3X overhead
C++ STL `unordered_map`	Liste di trabocco basate su liste	1.00	MurmurHash
Python	Indirizzam. aperto, scansione quadratica	0.66	

Considerazioni finali

Problemi con hashing

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

Hashing utilizzato in altre strutture dati

- Distributed Hash Table (DHT)
- Bloom filters

Oltre le tabelle hash

- Data deduplication
- Protezioni dati con hash crittografici (MD5)