

Algoritmi e Strutture Dati

Hashing

Alberto Montresor

Università di Trento

2019/01/13

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

1 Introduzione

- Motivazioni
- Definizioni base
- Tabelle ad accesso diretto

2 Funzioni hash

- Introduzione
- Funzioni hash semplici
- Reality check

3 Gestione collisioni

- Liste/vettori di trabocco
- Indirizzamento aperto
- Reality check

4 Conclusioni

Array associativi, mappe e dizionari

Python

```
>>> v = {}  
>>> v[10] = 5  
>>> v["10"] = 42  
>>> print(v[10]+v["10"])  
47
```

Go

```
ages := make(map[string]int)  
ages["alice"]=45  
ages["alberto"]=45  
ages["alberto"]++  
delete(ages, "alice")
```

Java

```
Map<String, String> capoluoghi = new HashMap<>();  
capoluoghi.put("Toscana", "Firenze");  
capoluoghi.put("Lombardia", "Milano");  
capoluoghi.put("Sardegna", "Cagliari");
```

Introduzione

Ripasso

Un dizionario è una struttura dati utilizzata per memorizzare insiemi dinamici di **coppie (chiave, valore)**

- Le coppie sono indicizzate in base alla chiave
- Il valore è un "dato satellite"

Operazioni:

- $\text{lookup}(key) \rightarrow value$
- $\text{insert}(key, value)$
- $\text{remove}(key)$

Applicazioni:

- Le tabelle dei simboli di un compilatore
- I dizionari di Python
- ...

Possibili implementazioni

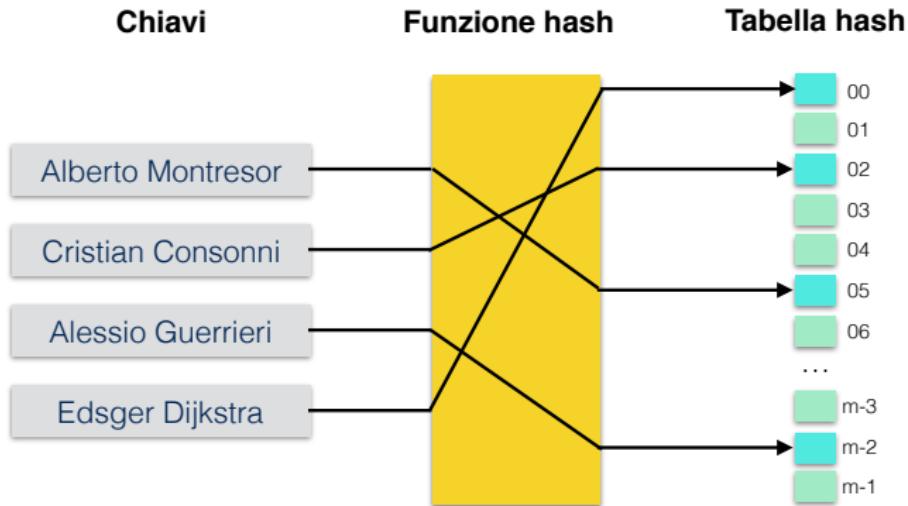
	Array non ordinato	Array ordinato	Lista	Alberi RB	Implemen. ideale
insert()	$O(1), O(n)$	$O(n)$	$O(1), O(n)$	$O(\log n)$	$O(1)$
lookup()	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
remove()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
foreach	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Implementazione ideale: **tabelle hash**

- Si sceglie una **funzione hash** h che mappa ogni chiave $k \in \mathcal{U}$ in un intero $h(k)$
- La coppia chiave–valore $\langle k, v \rangle$ viene memorizzata in un vettore nella posizione $h(k)$
- Questo vettore viene detto **tabella hash**
- Hash: From French **hacher** (“to chop”), from Old French **hache** (“axe”)

Tabelle hash – Definizioni

- L'insieme delle possibili chiavi è rappresentato dall'**insieme universo \mathcal{U}** di dimensione u
- Il vettore $T[0 \dots m - 1]$ ha dimensione m
- Una funzione hash è definita come $h : \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\}$



Collisioni

- Quando due o più chiavi nel dizionario hanno lo stesso valore hash, diciamo che è avvenuta una **collisione**
- Idealmente, vogliamo funzioni hash senza collisioni

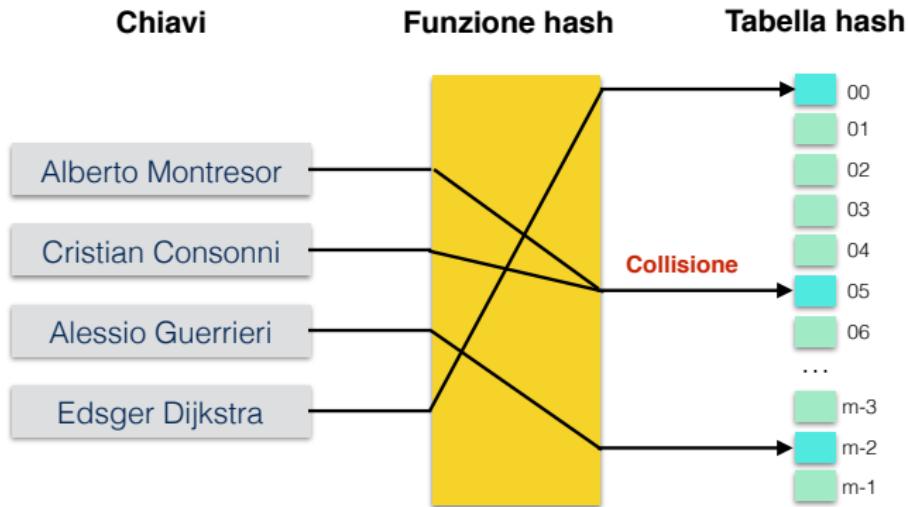


Tabelle ad accesso diretto

Caso particolare: l'insieme \mathcal{U} è già un sottoinsieme (piccolo) di \mathbb{Z}^+

- L'insieme dei giorni dell'anno, numerati da 1 a 366
- L'insieme dei Pokemon di Kanto, numerati da 1 a 151



Tabella a accesso diretto

- Si utilizza la funzione hash identità $h(k) = k$
- Si sceglie un valore m pari a u

Problemi

- Se u è molto grande, l'approccio non è praticabile
- Se u non è grande ma il numero di chiavi effettivamente registrate è molto minore di $u = m$, si spreca memoria

Funzioni hash perfette

Definizione

Una funzione hash h si dice **perfetta** se è **iniettiva**, ovvero

$$\forall k_1, k_2 \in \mathcal{U} : k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$$

Esempi

- Studenti ASD 2005-2016
N. matricola in $[100.090, 183.864]$
$$h(k) = k - 100.090, m = 83.774$$
- Studenti immatricolati 2014
N. matricola in $[173.185, 183.864]$
$$h(k) = k - 173.185, m = 10.679$$

Problemi

- Spazio delle chiavi
spesso grande, sparso,
non conosciuto
- È spesso impraticabile
ottenere una funzione
hash perfetta

Funzioni hash

Se non possiamo evitare le collisioni

- almeno cerchiamo di minimizzare il loro numero
- vogliamo funzioni che distribuiscano **uniformemente** le chiavi negli indici $[0 \dots m - 1]$ della tabella hash

Uniformità semplice

- Sia $P(k)$ la probabilità che una chiave k sia inserita in tabella
- Sia $Q(i)$ la probabilità che una chiave finisca nella cella i

$$Q(i) = \sum_{k \in \mathcal{U}: h(k)=i} P(k)$$

- Una funzione hash h gode 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

Se \mathcal{U} è dato dai 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**"

Come realizzare una funzione hash

Assunzione

Le chiavi possono essere tradotte in valori numerici non negativi, anche interpretando la loro rappresentazione in memoria come un numero.

Esempio: trasformazione stringhe

- $ord(c)$: valore ordinale binario del carattere c in qualche codifica
- $bin(k)$: rappresentazione binaria della chiave k , concatenando i valori binari dei caratteri che lo compongono
- $int(b)$: valore numerico associato al numero binario b
- $int(k) = int(bin(k))$

Come realizzare una funzione hash

Nei prossimi esempi

- Utilizziamo codice ASCII a 8 bit

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

Funzione hash - Estrazione

Estrazione

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

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.

Funzione hash - Estrazione

Esempio 1

$m = 2^p = 2^{16} = 65536$; 16 bit meno significativi di $\text{bin}(k)$

$\text{bin}(\text{"Alberto"}) = 01000001 01101100 01100010 01100101$
 01110010 **01110100 01101111**

$\text{bin}(\text{"Roberto"}) = 01010010 01101111 01100010 01100101$
 01110010 **01110100 01101111**

$H(\text{"Alberto"}) = \text{int}(\textbf{0111010001101111}) = 29.807$

$H(\text{"Roberto"}) = \text{int}(\textbf{0111010001101111}) = 29.807$

Funzione hash - Estrazione

Esempio 2

$m = 2^p = 2^{16} = 65536$; 16 bit presi all'interno di $\text{bin}(k)$

$\text{bin}(\text{"Alberto"}) = 0100\textcolor{red}{0001} \ 01101100 \ 0110\textcolor{red}{0010} \ 01100101$
 $01110010 \ 01110100 \ 01101111$

$\text{bin}(\text{"Alessio"}) = 0100\textcolor{red}{0001} \ 01101100 \ 0110\textcolor{red}{0101} \ 01110011$
 $01110011 \ 01101001 \ 01101111$

$H(\text{"Alberto"}) = \text{int}(\textcolor{red}{0001011011000110}) = 5.830$

$H(\text{"Alessio"}) = \text{int}(\textcolor{red}{0001011011000110}) = 5.830$

Funzione hash - XOR

XOR

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

Problemi

- Permutazioni (anagrammi) della stessa stringa possono generare lo stesso valore hash

Funzione hash - XOR

Esempio

$m = 2^{16} = 65536$; 5 gruppi di 16 bit ottenuti con 8 zeri di "padding"

$\text{bin}(\text{"montresor"}) =$

01101101 01101111 \oplus

01101110 01110100 \oplus

01110010 01100101 \oplus

01110011 01101111 \oplus

01110010 00000000

$\text{bin}(\text{"sontremor"}) =$

01110011 01101111 \oplus

01101110 01110100 \oplus

01110010 01100101 \oplus

01101101 01101111 \oplus

01110010 00000000

$H(\text{"montresor"}) =$

$\text{int}(01110000 00010001) =$

28.689

$H(\text{"sontremor"}) =$

$\text{int}(01110000 00010001) =$

28.689

Funzione hash - Metodo della divisione

Metodo della divisione

- m dispari, meglio se numero primo
- $H(k) = \text{int}(k) \bmod m$

Esempio

$m = 383$

$$H(\text{"Alberto"}) = 18.415.043.350.787.183 \bmod 383 = 221$$

$$H(\text{"Alessio"}) = 18.415.056.470.632.815 \bmod 383 = 77$$

$$H(\text{"Cristian"}) = 4.860.062.892.481.405.294 \bmod 383 = 130$$

Funzione hash - Metodo della divisione

Non vanno bene:

- $m = 2^p$: solo i p bit meno significativi vengono considerati
- $m = 2^p - 1$: permutazione di stringhe con set di caratteri di dimensione 2^p hanno lo stesso valore hash (Esercizio: dimostrare)

Vanno bene:

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

Funzione hash - Metodo della moltiplicazione (Knuth)

Metodo della moltiplicazione

- m qualsiasi, meglio se potenza di 2
- C costante reale, $0 < C < 1$
- Sia $i = \text{int}(k)$
- $H(k) = \lfloor m(C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$

Esempio

$$m = 2^{16}$$

$$C = \frac{\sqrt{5}-1}{2}$$

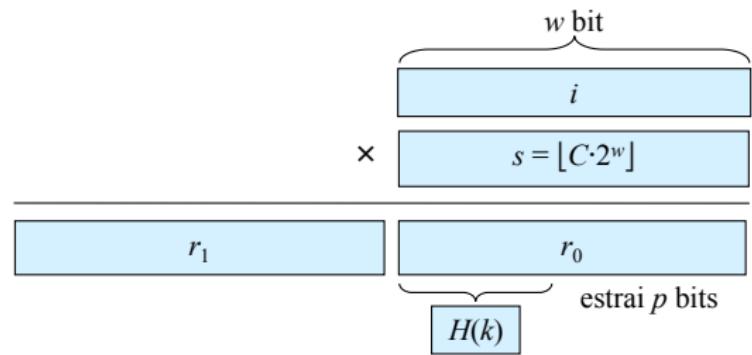
$$H(\text{"Alberto"}) = 65.536 \cdot 0.78732161432 = 51.598$$

$$H(\text{"Alessio"}) = 65.536 \cdot 0.51516739168 = 33.762$$

$$H(\text{"Cristian"}) = 65.536 \cdot 0.72143641000 = 47.280$$

Metodo della moltiplicazione - Implementazione

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



Reality check

- Non è poi così semplice...
 - Il metodo della moltiplicazione suggerito da Knuth non fornisce hashing uniforme
- Test moderni per valutare la bontà delle funzioni hash
 - **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)
 - Deve essere molto difficile o quasi impossibile risalire al testo che ha portato ad un dato hash;

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 perché 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

Come gestire le collisioni?

- Dobbiamo trovare posizioni alternative per le chiavi
- Se una chiave non si trova nella posizione attesa, bisogna cercarla nelle posizioni alternative
- Questa ricerca:
 - dovrebbe costare $O(1)$ nel caso medio
 - può costare $O(n)$ nel caso pessimo

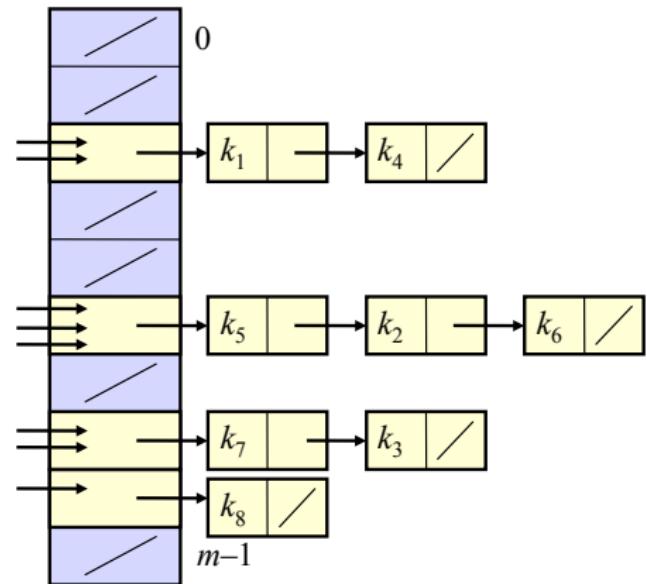
Due possibili tecniche

- **Liste di trabocco** o memorizzazione esterna
- **Indirizzamento aperto** o memorizzazione interna

Liste/vettori di trabocco (Concatenamento o Chaining)

Idea

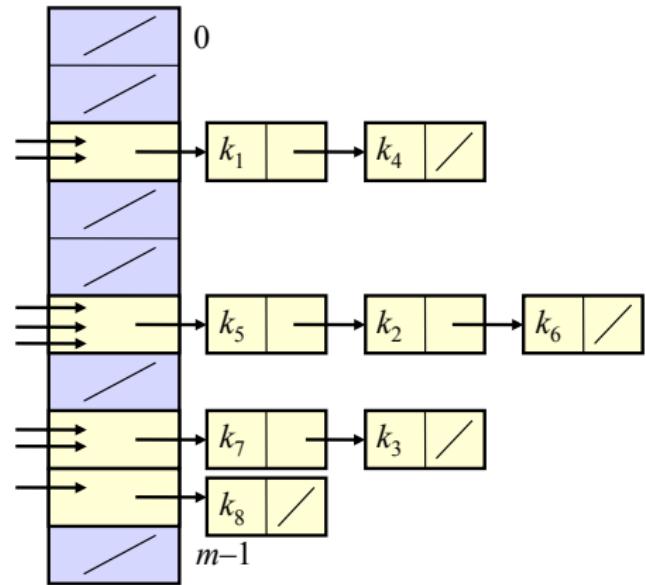
- Le chiavi con lo stesso valore hash h vengono memorizzate in una **lista monodirezionale** / vettore dinamico
- Si memorizza un puntatore alla testa della lista / al vettore nello slot $H(k)$ -esimo della tabella hash



Liste/vettori di trabocco (Concatenamento o Chaining)

Operazioni

- **insert()**: inserimento in testa
- **lookup(), remove()**: scansione della lista per cercare la chiave



Liste/vettori di trabocco: analisi complessità

n	Numero di chiavi memorizzati in tabella hash
m	Capacità della tabella hash
$\alpha = n/m$	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()**: $\Theta(1)$
- **lookup()**, **remove()**: $\Theta(n)$

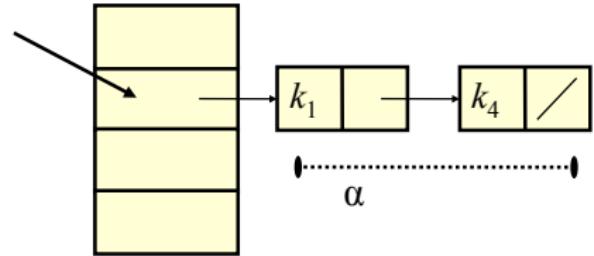
Liste/vettori di trabocco: analisi complessità

Analisi del caso medio: assunzioni

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

Quanto sono lunghe le liste / i vettori?

- Il valore **atteso** della lunghezza di una lista è pari a $\alpha = n/m$



Liste/vettori di trabocco: analisi complessità

Costo hashing

- Una chiave **presente** o **non presente** in tabella può essere collocata in uno qualsiasi degli m slot
- Costo di hashing: $\Theta(1)$

Ricerca senza successo

- Una ricerca **senza successo** tocca tutte le chiavi nella lista corrispondente
- Costo atteso: $\Theta(1) + \alpha$

Ricerca con successo

- Una ricerca con **successo** tocca in media metà delle chiavi nella lista corrispondente
- Costo atteso: $\Theta(1) + \alpha/2$

Liste/vettori di trabocco: analisi 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 $O(1)$

Indirizzamento aperto

Problemi delle liste/vettori di trabocco

- 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**

Definizioni

Ispezione

Un'**ispezione** è l'esame di uno slot durante la ricerca.

Funzione hash

Estesa nel modo seguente:

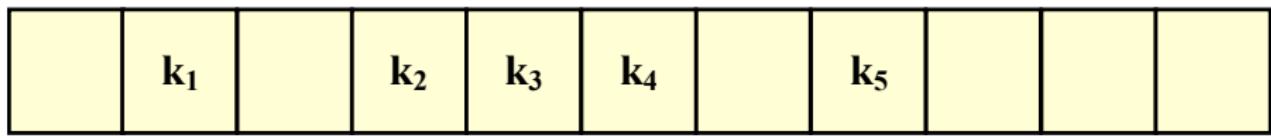
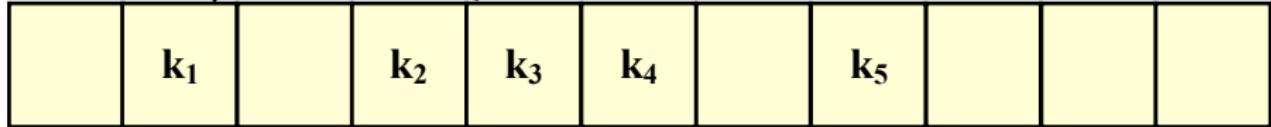
$$H : \mathcal{U} \times \overbrace{[0 \dots m - 1]}^{\text{Numero ispezione}} \rightarrow \overbrace{[0 \dots m - 1]}^{\text{Indice vettore}}$$

Sequenza di ispezione

Una **sequenza di ispezione** $[H(k, 0), H(k, 1), \dots, H(k, m - 1)]$ è una permutazione degli indici $[0, \dots, m - 1]$ corrispondente all'ordine in cui vengono esaminati gli slot.

- Non vogliamo esaminare ogni slot più di una volta
- Potrebbe essere necessario esaminare tutti gli slot nella tabella

Esempio

 $H(k, 0)$  $H(k, 0)$ $H(k, 1)$ 

Fattore di carico

Cosa succede al fattore di carico α ?

- Compreso fra 0 e 1
- La tabella può andare in overflow

Tecniche di ispezione

Hashing uniforme

La situazione ideale prende il nome di **hashing uniforme**, in cui ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una qualsiasi delle $m!$ permutazioni di $[0, \dots, m - 1]$.

- Generalizzazione dell'hashing uniforme semplice
- Nella realtà:
 - E' difficile implementare il vero hashing uniforme
 - Ci si accontenta di ottenere almeno una permutazione
- Tecniche diffuse:
 - Ispezione lineare
 - Ispezione quadratica
 - Doppio hashing

Ispezione lineare

Funzione: $H(k, i) = (H_1(k) + h \cdot i) \text{ mod } m$

- La sequenza $H_1(k), H_1(k) + h, H_1(k) + 2 \cdot h, \dots, H_1(k) + (m - 1) \cdot h$ (modulo m) è determinata dal primo elemento
- Al massimo m sequenze di ispezione distinte sono possibili

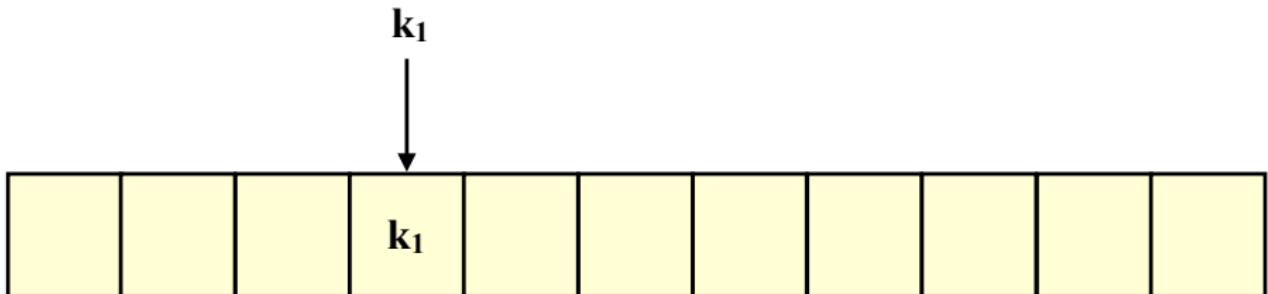
Agglomerazione primaria (primary clustering)

- 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)

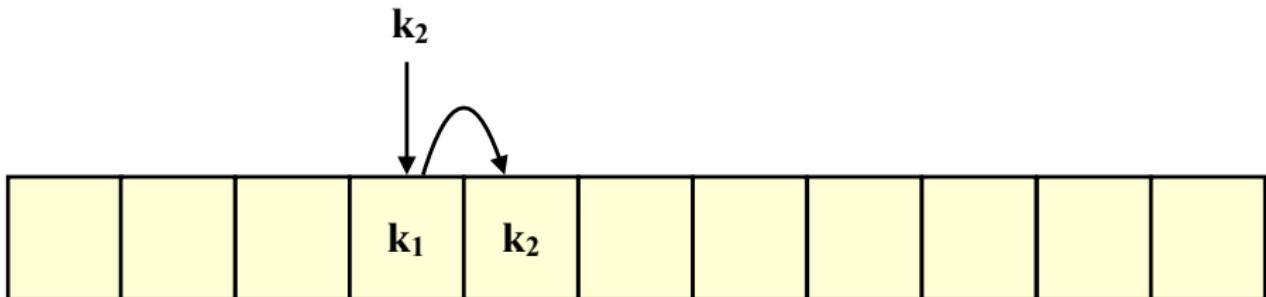
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

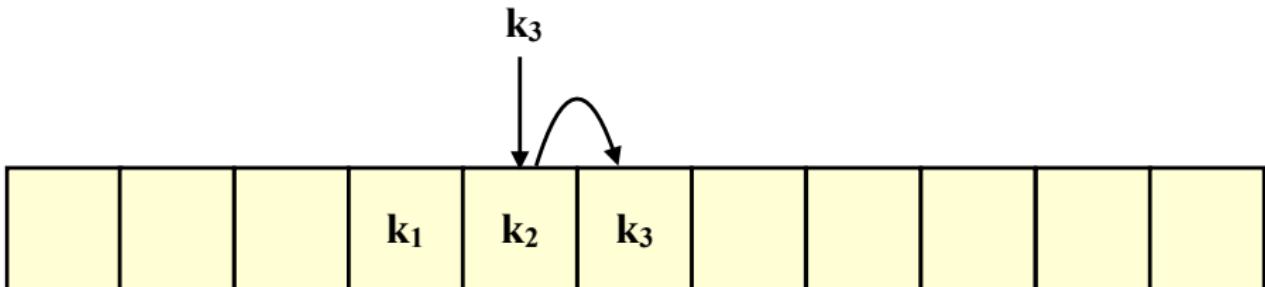
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

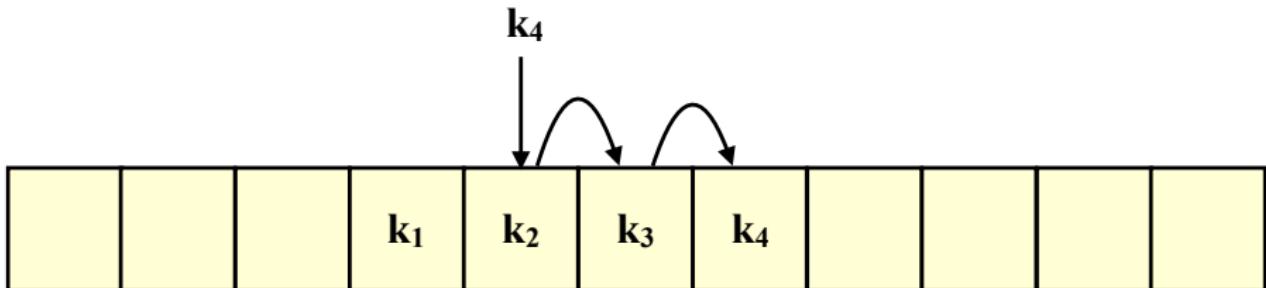
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

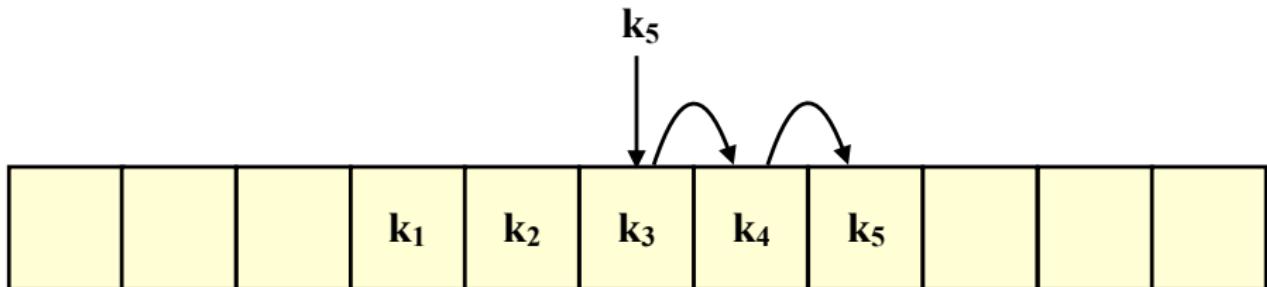
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

- Lunghe sotto-seguenze 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: $H(k, i) = (H_1(k) + h \cdot i^2) \bmod m$

- Dopo il primo elemento $H_1(k, 0)$, le ispezioni successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione i
- La sequenza risultante **non è una permutazione!**
- Al massimo m sequenze di ispezione distinte sono possibili

Agglomerazione secondaria (secondary clustering)

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

Doppio hashing

Funzione: $H(k, i) = (H_1(k) + i \cdot H_2(k)) \bmod m$

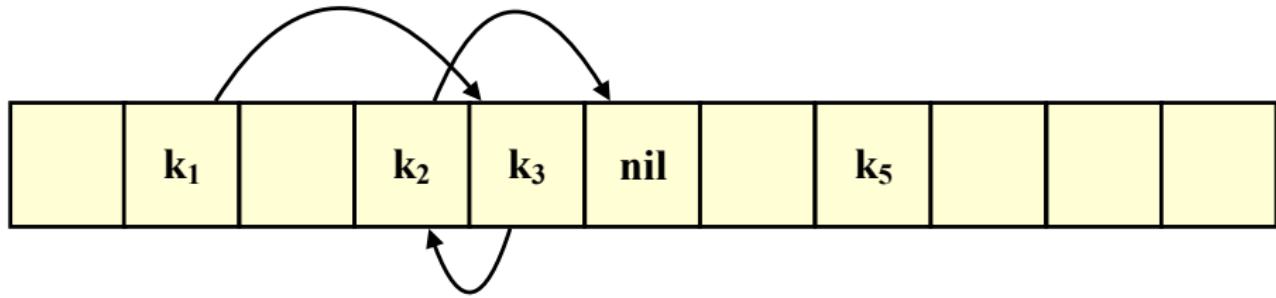
- Due funzioni ausiliarie:
 - H_1 fornisce la prima ispezione
 - H_2 fornisce l'offset delle successive ispezioni
- Al massimo m^2 sequenze di ispezione distinte sono possibili
- Per garantire una permutazione completa, $H_2(k)$ deve essere relativamente primo con m
 - Scegliere $m = 2^p$ e $H_2(k)$ deve restituire numeri dispari
 - Scegliere m primo, e $H_2(k)$ deve restituire numeri minori di m

Cancellazione

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

Cancellazione

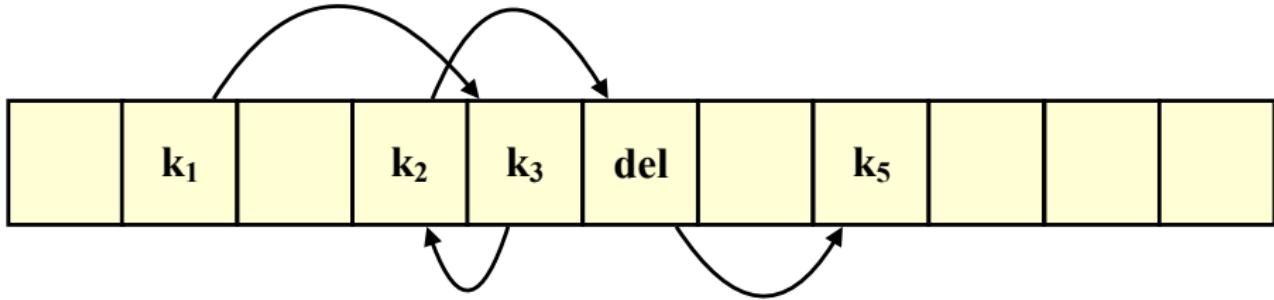
Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un **nil**. Perché?



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 $t = \text{new HASH}$

$t.m = dim$

$t.K = \text{new Item}[0 \dots dim - 1]$

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

for $i = 0$ **to** $dim - 1$ **do**

$t.K[i] = \text{nil}$

return t

Implementazione - Hashing doppio

```
int scan(ITEM k, boolean insert)
```

```
    int c = 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 c == m then
```

```
            c = j
```

```
            j = (j + H'(k)) mod m
```

```
            i = i + 1
```

```
        if insert and K[j] ≠ k and c < m then
```

```
            j = c
```

```
    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
```

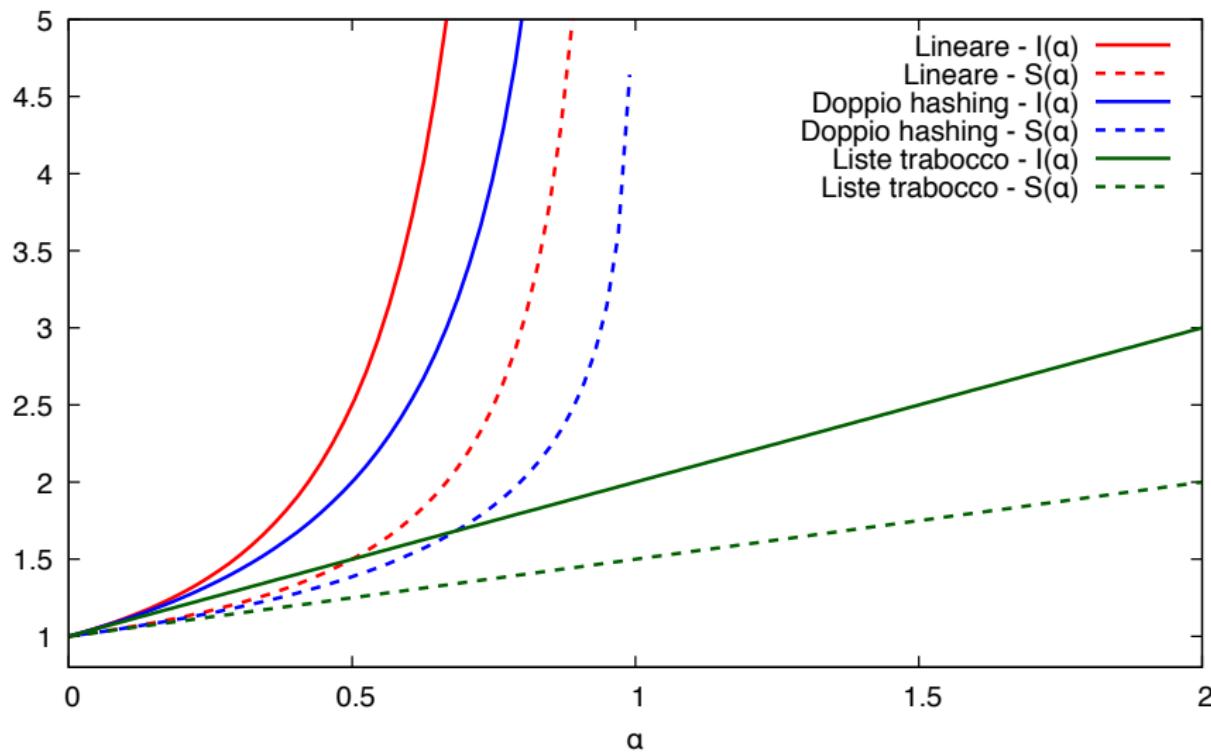
Implementazione - Hashing doppio

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

Complessità

Metodo	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	$\alpha \geq 0$	$1 + \alpha$	$1 + \alpha/2$

Complessità



Ristrutturazione

- Non è conveniente che α cresca troppo
 - In particolare con la scansione interna
 - Ma vale anche per le liste di trabocco
- Sopra un soglia t_α prefissata (tipicamente 0.5-0.75)
 - 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)

Reality check

Linguaggio	Tecnica	t_α	Note
Java 7 HashMap	Liste di trabocco basate su <code>LinkedList</code>	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++ <code>sparse_hash</code>	Ind. aperto, scansione quadratica	?	Overhead: $2n$ bit
C++ <code>dense_hash</code>	Ind. aperto, scansione quadratica	0.5	X byte per chiave-valore $\Rightarrow 2-3X$ overhead
C++ STL <code>unordered_map</code>	Liste di trabocco basate su liste	1.00	MurmurHash
Python	Indirizzam. aperto, scansione quadratica	0.66	

Java hashCode()

Dalla documentazione di java.lang.Object

The general contract of `hashCode()` is:

- ➊ Whenever it is invoked on the same object more than once during an execution of a Java application, *the `hashCode()` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified.* This integer need not remain consistent from one execution of an application to another execution of the same application.
- ➋ *If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.*
- ➌ It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, *the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.*

Java hashCode()

Se una classe non fa override di `equals()`:

- Eredita i metodi `equals()` e `hashCode()` così come definiti da `java.lang.Object`:
 - `x.equals(y)` ritorna **true** se e solo se `x == y`
 - `x.hashCode()` converte l'indirizzo di memoria di `x` in un intero

Se una classe fa ovveride di `equals()`:

- "Always override hashCode when you override equals", in Bloch, Joshua (2008), Effective Java (2nd ed.)
- Se non fate override, oggetti uguali finiscono in posizioni diverse nella tabella hash

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;
}
```

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)