

Algoritmi e Strutture Dati

Tabelle Hash

Alberto Montresor and Davide Rossi

Università di Bologna

19 settembre 2024

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



Sommario

1 Introduzione

- 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

Introduzione

Ripasso

Un dizionario è una struttura dati utilizzata per memorizzare insieme 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
- ...

Operazioni di dizionario

- **INSERT**
- **SEARCH**
- **DELETE**

Possibili implementazioni

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

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**

- Hash: From French **hacher** (“to chop”), from Old French **hache** (“axe”)
- L’insieme delle possibili chiavi è rappresentato dall’insieme universo \mathcal{U} di dimensione u
- Si memorizzano le chiavi in un vettore $T[0 \dots m - 1]$ di dimensione m , detto **tabella hash**

Tabelle hash – Definizioni

- Una **funzione hash** è definita come $h : \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\}$
- La coppia chiave–valore $\langle k, v \rangle$ viene memorizzata in un vettore nella posizione $h(k)$

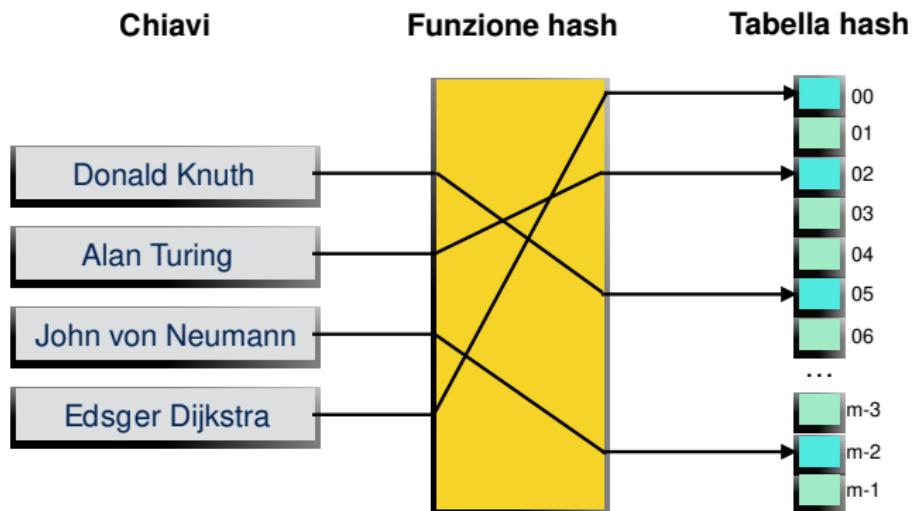


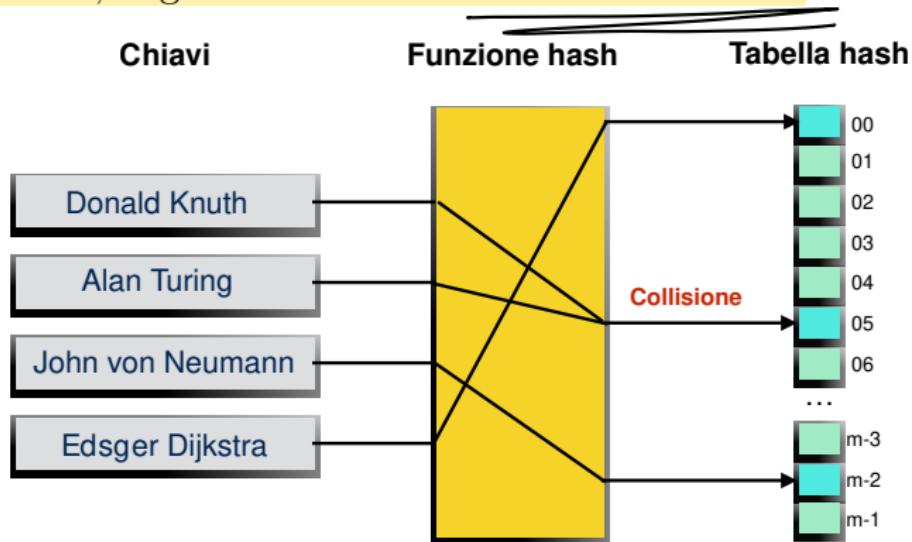
Tabella Hash

Una tabella Hash è una struttura dati efficace per implementare dizionari. Sebbene la ricerca di un elemento in una tabella hash possa richiedere fino allo stesso tempo richiesto per ricercare un elemento in una lista concatenata, cioè $O(n)$ nel caso peggiore, l'hashing si comporta molto bene nella pratica. Il tempo medio per cercare un elemento in una tabella hash è $O(1)$.

Quando il numero di chiavi effettivamente memorizzate è piccolo rispetto al numero totale di chiavi possibili, le tabelle hash diventano una valida alternativa all'indirizzamento diretto di un array, in quanto una tabella hash, tipicamente, usa un array di dimensione proporzionale al numero di chiavi effettivamente memorizzate.

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



Possibili implementazioni

	Array non ordinato	Array ordinato	Lista	Alberi RB	Hash table
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	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(m)$

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

Tabella a indirizzamento diretto

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo delle chiavi U è ragionevolmente piccolo.

Supponiamo che abbiano bisogno di un insieme dinamico in cui ogni elemento ha una chiave diversa da quella degli altri, e che questa chiave E $U = \{0, 1, \dots, m-1\}$, dove m non è troppo grande.

Per rappresentare l'insieme dinamico, possiamo usare un array oppure una tabella a indirizzamento diretto, che indichiamo con $T[0:m-1]$, dove ogni posizione o cella di una tabella a indirizzamento diretto corrisponde a una chiave nell'universo U .

- La cella K punta a un elemento dell'insieme con chiave K . Se l'insieme non contiene nessun elemento con chiave K , allora $T[K] = \text{NIL}$

Svantaggi dell'indirizzamento diretto

- Se l'universo delle chiavi U è molto grande, memorizzare una tabella T di dimensione $|U|$ può essere irrealizzabile. (spazio insufficiente) (U universo delle chiavi)
- Se l'insieme delle chiavi effettive K è piccolo rispetto a U , la maggior parte di T rimane vuota (spreco di memoria)

*** Perché esiste un "universo delle chiavi" U e non basta K?**

L'universo delle chiavi **U** rappresenta l'insieme di **tutte le chiavi possibili** che potrebbero essere inserite nella tabella hash, mentre **K** è l'insieme delle chiavi **effettivamente memorizzate** in un dato momento.

**** 1. Perché U è necessario?**

- **Progettazione della funzione di hash**:
 - La funzione 'h(k)' deve essere definita su **tutti i possibili valori di 'k'**, non solo su quelli attualmente presenti.
 - Ad esempio, se le chiavi sono numeri interi, 'U' potrebbe essere ' \mathbb{Z} ' (tutti gli interi), mentre 'K' è solo un piccolo sottinsieme.
 - **Garantire la copertura di tutti i casi**;
- Se una nuova chiave ' $k \notin K$ ' viene inserita, ' $h(k)$ ' deve comunque produrre un valore valido nella tabella.
- Se definissimo 'h' solo su 'K', ogni inserimento di una nuova chiave richiederebbe una ri-definizione di 'h'.
 - **Analisi teorica delle prestazioni**;
- L'hashing uniforme assume che ' $h(k)$ ' distribuisce le chiavi in modo casuale in ' $T[0..m-1]$ ', **indipendentemente da quali chiavi sono in 'K'**.
- Se 'h' fosse definita solo su 'K', non avrebbe senso parlare di distribuzione uniforme.

**** 2. Perché non possiamo usare solo K?**

- **dinamicità delle tabella hash**:
 - 'K' cambia nel tempo (inserimenti/cancellazioni), mentre 'U' è fisso.
- Se 'h' dipendesse da 'K', dovremmo ri-calcolarla ogni volta che 'K' cambia, il che sarebbe inefficiente.
 - **Collisioni inevitabili**;
- Poiché ' $|U| > m$ ' (la tabella è più piccola dell'universo), **esistono necessariamente chiavi diverse ' $k_1, k_2 \in U$ ' tali che ' $h(k_1) = h(k_2)$ '**, anche se nessuna delle due è ancora in 'K'.
- Se 'h' fosse definita solo su 'K', potremmo illuderci di evitare collisioni, ma appena inseriamo una nuova chiave potremmo scoprire che collide con una già presente.

**** 3. Esempio Pratico**

- **Scenario**:
 - 'U' = tutti i possibili nomi di lunghezza ≤ 20 caratteri (un insieme enorme).
 - 'K' = { "Alice", "Bob", "Charlie" } (solo 3 chiavi memorizzate).
 - 'm' = 5 (tabella hash piccola).
 - **Problema se 'h' fosse definita solo su 'K'**:
- Potremmo definire ' $h("Alice") = 0$ ', ' $h("Bob") = 1$ ', ' $h("Charlie") = 2$ '.
- Ma se inseriamo "David", dovremmo ridefinire ' $h("David")$ ', rischiando collisioni o spreco di spazio.
 - **Soluzione con 'U'**:
- Definiamo ' $h(k) = (\text{somma dei codici ASCII}) \bmod m$ ' per **tutti i nomi possibili**.
- Anche se "David" non è ancora in 'K', ' $h("David")$ ' è già ben definito.

**** 4. Conclusione**

L'universo 'U' è necessario perché:

1. **La funzione di hash deve essere universale**, non dipendente dall'insieme corrente 'K'.
2. **I' analisi delle prestazioni** si basa su proprietà statistiche di 'h' su 'U', non solo su 'K'.
3. **Collisioni sono inevitabili** a causa di ' $|U| > m$ ', anche se ' $|K|$ ' è piccolo.
 - Se usassimo solo 'K', perderemmo l'efficienza e la generalità delle tabelle hash.

Sommario

1 Introduzione

- 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

Con l'indirizzamento diretto, un elemento con chiave k è memorizzato nella cella K , ma con l'hashing, questo elemento è memorizzato nella cella $h(k)$; cioè utilizziamo una funzione di hash h per calcolare il numero della cella a partire dalla chiave k .

La funzione di hash h associa gli elementi dell'universo U delle chiavi alle celle di una tabella hash $T[0, 1, \dots, m-1]$

$$h: U \rightarrow \{0, 1, \dots, m-1\}$$

La dimensione m della tabella hash è generalmente molto più piccola di $|U|$.

Diciamo che un elemento con chiave k viene mappato nella cella $h(k)$ e che $h(k)$ è il valore di hash della chiave k

Funzioni hash perfette

Definizione

Una funzione hash h si dice **perfetta** se è **iniettiva**, ovvero se non dà origine a collisioni:

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

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:

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

Uniformità Scopulice nelle Funzioni Hash

L'uniformità scopulice è un criterio teorico che definisce quanto "bené" una funzione di hash distribuisce le chiavi tra le celle di una tabella hash. L'obiettivo è minimizzare le collisioni garantendo che ogni cella abbia la stessa prob. di essere utilizzata.

$P(k) \rightarrow$ Probabilità che una chiave k (scelta dall'universo $|U|$) venga inserita nella tabella

Ex: Se U contiene novi di persone alcuni novi sono più comuni. $P(\text{"Alice"}) > P(\text{"Zorro"})$

$Q(i) \rightarrow$ Probabilità che una chiave finisca nella cella i della tabella hash

\Rightarrow Si calcola come la somma delle probabilità $P(k)$ di tutte le chiavi k t.c.
 $h(k)=i$

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

: \rightarrow t.c.

Uniformità Scopulice \rightarrow Una funzione h soddisfa l'uniformità scopulice se, per cogni cella i , la probabilità $Q(i)$ è uguale a $\frac{1}{m}$ ($m \rightarrow$ numero di celle)

$$\forall i \in [0, 1, \dots, m-1] : Q(i) = \frac{1}{m}$$

Perché è importante?

Se $Q(i) = \frac{1}{m}$ per cogni i , le chiavi si distribuiscono equamente, riducendo la probabilità che due chiavi finiscano nella stessa cella

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**” \rightarrow queste sono un modello idealizzato

Come realizzare una funzione hash

Assunzione

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

Esempio: trasformazione stringhe

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



Come realizzare una funzione hash

Nei prossimi esempi, utilizziamo codice ASCII a 8 bit

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

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

Per trasformare un numero grande (come 4'476'743) in un indice compreso tra 0 e $m-1$, si fa la divisione modulo del numero \rightarrow Formula $h(k) = k \bmod m$

$$\text{es. se } m=100 \quad | \quad \begin{array}{l} m=100 \\ 4'476'743 \bmod 100 = 3 \\ 4'476'743 \bmod 100 = 43 \end{array}$$

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

Domanda: Quali possono essere i problemi di questo approccio?

Collisioni Prevedibili per Chiavi Simili \rightarrow Se due chiavi K_1 e K_2 differiscono solo nei bit non estratti, avranno lo stesso hash

$$\text{Eg. } K_1 = 1010\,\underline{1101} \text{ (decimale 173)}$$

$$K_2 = 0010\,\underline{1101} \text{ (decimale 45)}$$

$p=4$

$$\Rightarrow H(K_1) = H(K_2) = \{1101\} = \text{cella 13}$$

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 $bin(k)$

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

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

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

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

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

Domanda: Quali possono essere i problemi di questo approccio?

Collisioni con chiavi simmetriche o Permutate \rightarrow l'XOR è commutativo e associativo, quindi chiavi con bit permutati o invertiti in modo simmetrico possono produrre lo stesso hash.

$$\text{es. } k_1 = 1100$$

$$k_2 = 0011$$

Calcolando lo xor dei primi 2 bit con gli ultimi 2

$$H(k_1) = 11 \oplus 00 = 11 \quad H(k_2) = 00 \oplus 11 = 11$$

Entrambe le chiavi collidono nonostante siano diverse

\hookrightarrow Aumento delle collisioni per chiavi con pattern complementari o riordinati

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
- Mitigazione: eseguire una rotazione dei bit dipendente dalla posizione.

Ottimizzare Problema \rightarrow Sensibilità alla Ridondanza dei Bit

Se un Bit è presente in numero pari di volte nei sottoinsiemi estratti, il suo contributo all'XOR è annullato ($\text{pk } x \oplus x = 0$)

Eg. Se $K=101101 \rightarrow$ applico XOR $\rightarrow 101 \oplus 101 = 000$

Funzione hash - XOR

Esempio

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

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

01101101 01101111 \oplus

01101110 01110100 \oplus

01110010 01100101 \oplus

01110011 01101111 \oplus

01110010 00000000

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

01110011 01101111 \oplus

01101110 01110100 \oplus

01110010 01100101 \oplus

01101101 01101111 \oplus

01110010 00000000

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

$int(01110000\ 00010001) =$

28.689

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

$int(01110000\ 00010001) =$

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 (caveat: se la funzione hash perturba uniformemente tutti i bit questo non è un problema)
- $m = 2^p - 1$: permutazione di stringhe con set di caratteri di dimensione 2^p hanno lo stesso valore hash

Vanno bene:

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

Reality check

- Ottenere una “buona” funzione hash non è poi così semplice...
- 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 veloci. Ha varianti a 32, 64, 128, 256 bit.	Wikipedia Codice
Farm Hash	Il successore di City Hash, sempre sviluppato da Google.	Codice

Sommario

1 Introduzione

- 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

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

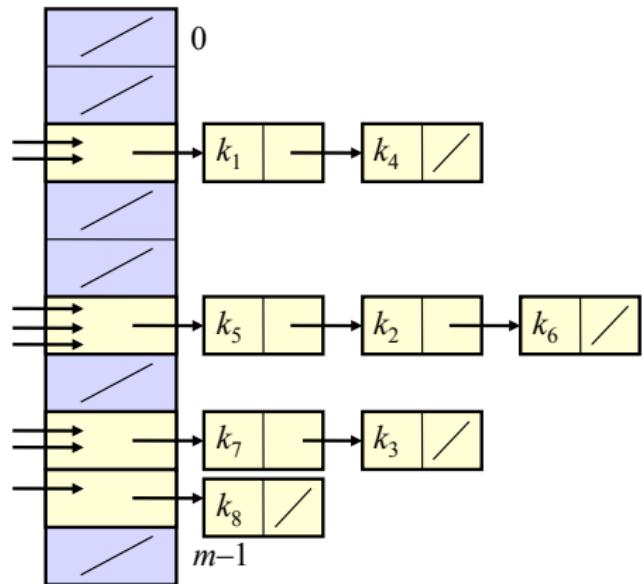
Due possibili tecniche

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

Liste/vettori di trabocco (Concatenamento o Chaining)

Idea

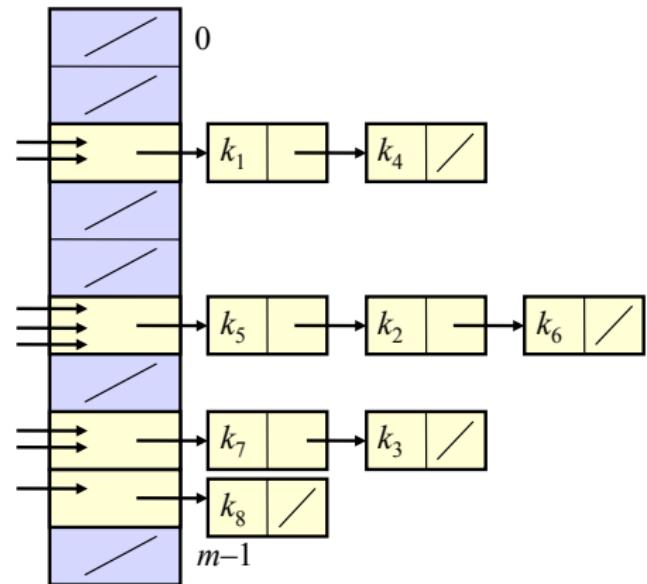
- Le chiavi con lo stesso valore hash h vengono memorizzate in una lista monodirezionale / vettore dinamico
- Si memorizza un puntatore alla testa della lista / al vettore nello slot $H(k)$ -esimo della tabella hash



Liste/vettori di trabocco (Concatenamento o Chaining)

Operazioni

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



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?

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 $\rightarrow \alpha = n/m$
- **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?

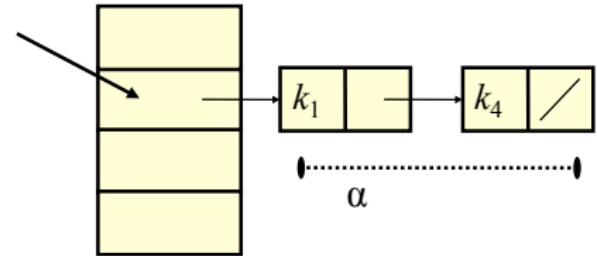
Liste/vettori di trabocco: analisi complessità

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

Ricerca con successo

- Una ricerca con **successo** tocca in media metà delle chiavi nella lista corrispondente

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**

Ricerca di un elemento

Per cercare un elemento si esaminano sistematicamente le celle della tabella relative alle possibili scelte per quell'elemento, in ordine di scelta decrescente, finché non si trova quello desiderato oppure si trova una cella vuota, e si verifica quindi che l'elemento non è presente nella tabella.

Indirizzamento aperto

Ispezione

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

Funzione hash

Estesa nel modo
seguente:

$$H : \mathcal{U} \times \underbrace{[0 \dots m-1]}_{\substack{\text{Universo delle} \\ \text{Chiavi}}} \rightarrow \underbrace{[0 \dots m-1]}_{\text{Indice vettore}}$$

Funzione Hash Estesa

La funzione hash viene modificata per includere un numero di ispezioni.

Si calcola $H(k, 0)$. Se la cella contiene k , la ricerca termina.

→ Altrimenti, si continua con $H(k, 1)$, ecc... finché:

- Si trova k (successo)
- Si incontra una cella vuota (fallimento) (k non è presente)

Indirizzamento aperto

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

	k_1		k_2	k_3	k_4		k_5		
--	-------	--	-------	-------	-------	--	-------	--	--

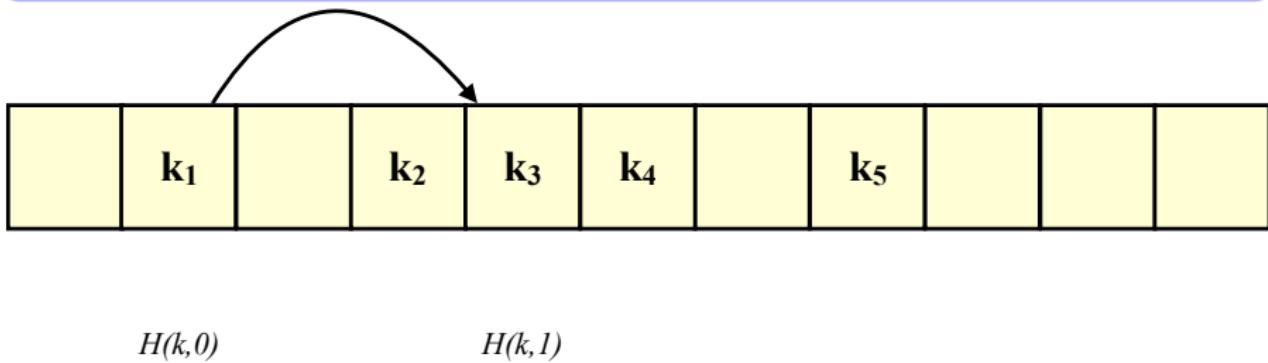
$$H(k, 0)$$

Indirizzamento aperto

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

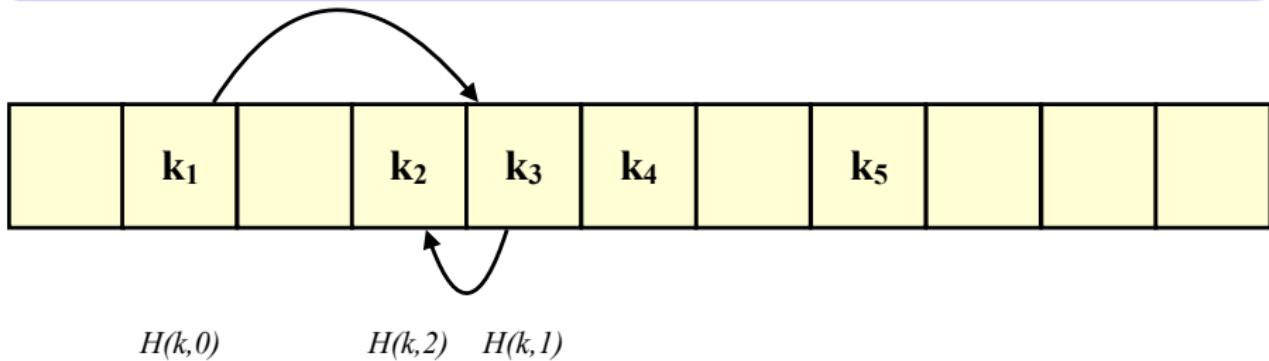


Indirizzamento aperto

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

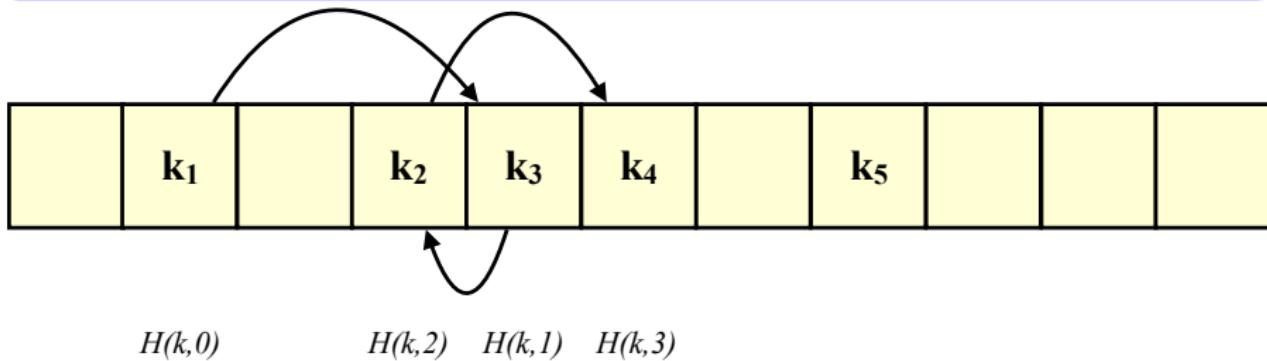


Indirizzamento aperto

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

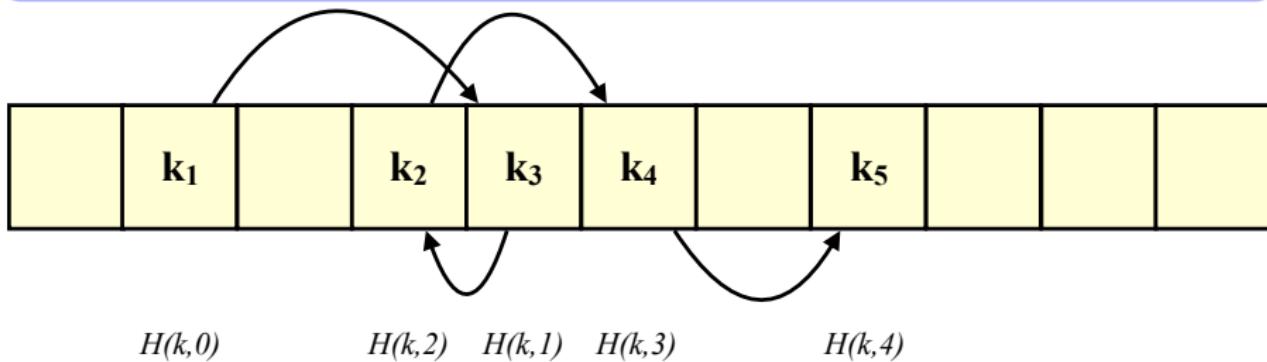


Indirizzamento aperto

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

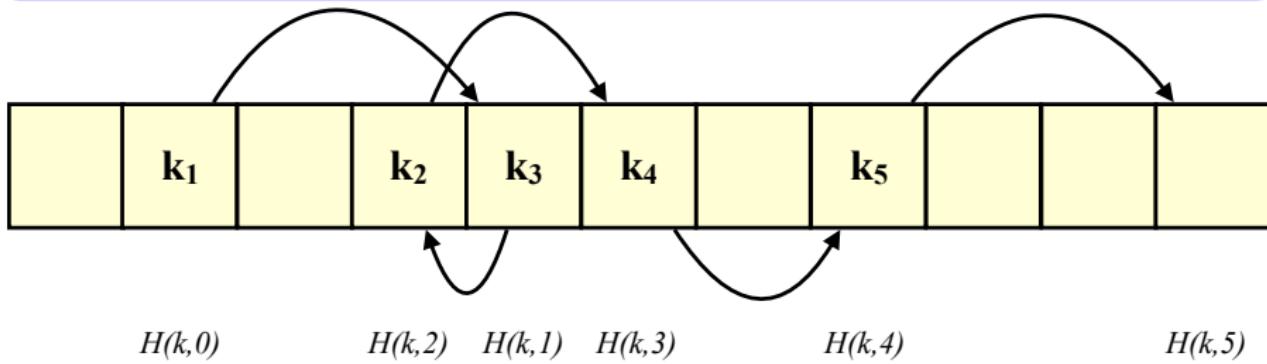


Indirizzamento aperto

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



Fattore di carico

Cosa succede al fattore di carico α ?

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

Nell'indirizzamento aperto, la tabella hash puo' "riempirsi" al punto tale che non e' più possibile effettuare ulteriori inserimenti. Una conseguenza di ciò e' che il fattore di carico α non puo' mai superare 1.

$$\alpha = \frac{n}{m} = \frac{\text{numero di chiavi memorizzate in tabella hash}}{\text{capacita' della tabella hash}}$$

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:
 - È difficile implementare il vero hashing uniforme
 - Ci si accontenta di ottenere almeno una permutazione
- Tecniche diffuse:
 - Ispezione lineare
 - Ispezione quadratica
 - Doppio hashing

Per ogni chiave, ognuna delle $m!$ permutazioni di $(0, 1, \dots, m-1)$ ha la stessa probabilità di essere la sua sequenza di ispezione.

Ispezione lineare

→ Caso particolare di hashing doppio

$h = \text{passo costante}$ ($\frac{\text{spessore}}{h-1}$)

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

- 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

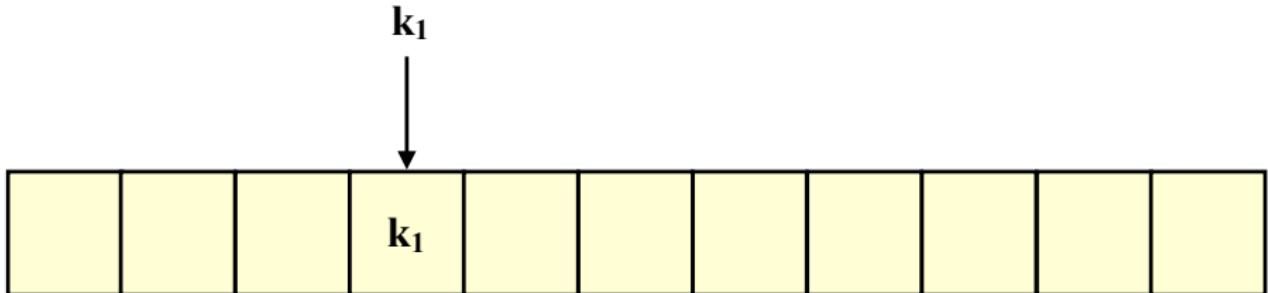
oss

Determinismo → La sequenza dipende solo da $H_1(k)$ e h . Se due chiavi hanno lo stesso $H_1(k)$, seguono la stessa sequenza → rischio di collisioni

Agglomerazione primaria

Agglomerazione primaria (primary clustering)

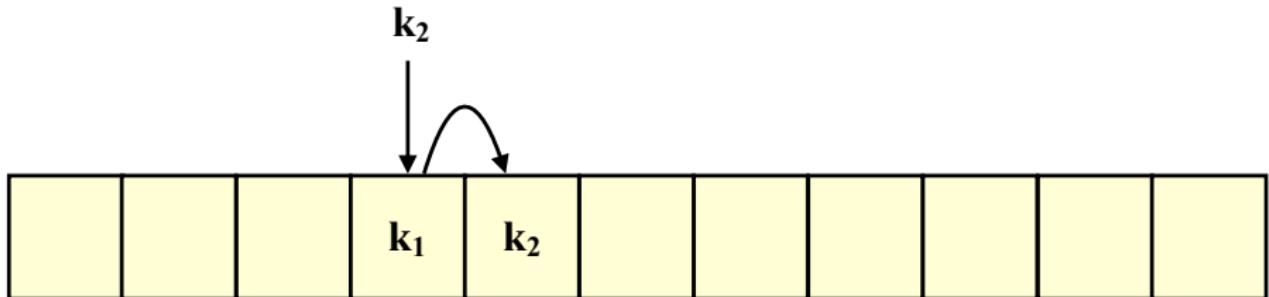
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

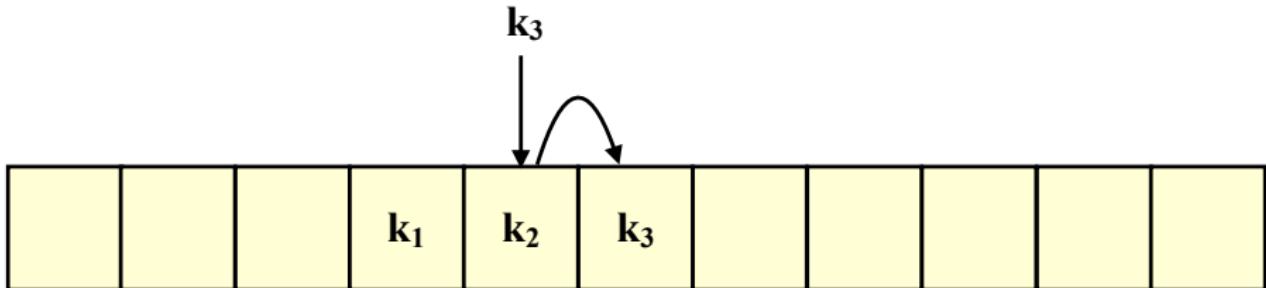
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

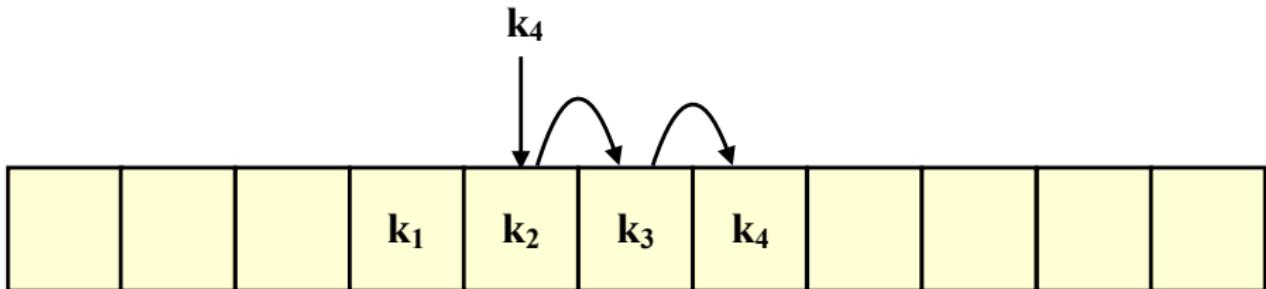
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

Agglomerazione primaria (primary clustering)

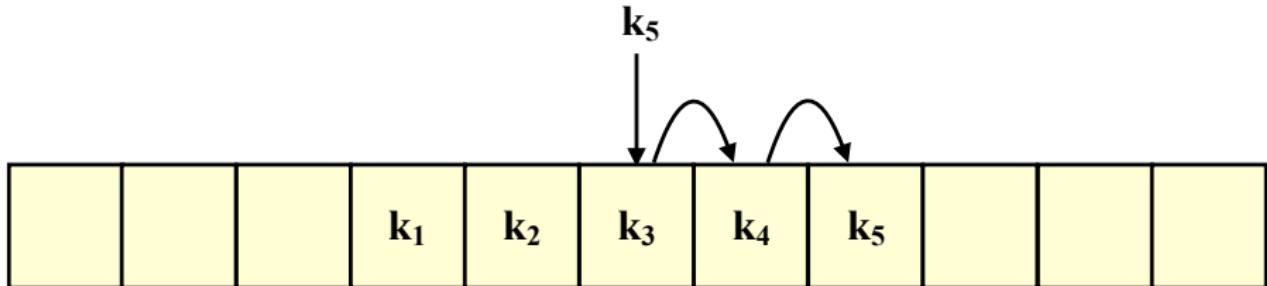
- Lunghe sotto-seguenze occupate...
- ... che tendono a diventare più lunghe: uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
- I tempi medi di inserimento e cancellazione crescono



Agglomerazione primaria

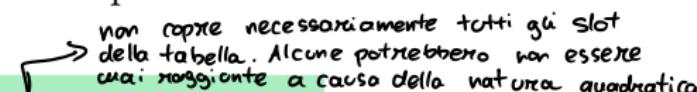
Agglomerazione primaria (primary clustering)

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

- 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


non copre necessariamente tutti gli slot della tabella. Alcune potrebbero non essere mai raggiunte a causa della natura quadratica
- 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: $H(k, i) = (H_1(k) + i \cdot H_2(k)) \text{ mod } m$

- 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
↳ reduce la probabilità di collisioni ripetute
- Per garantire una permutazione completa, $H_2(k)$ deve essere relativamente primo con m ($\text{MCD} = 1$)
 - 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

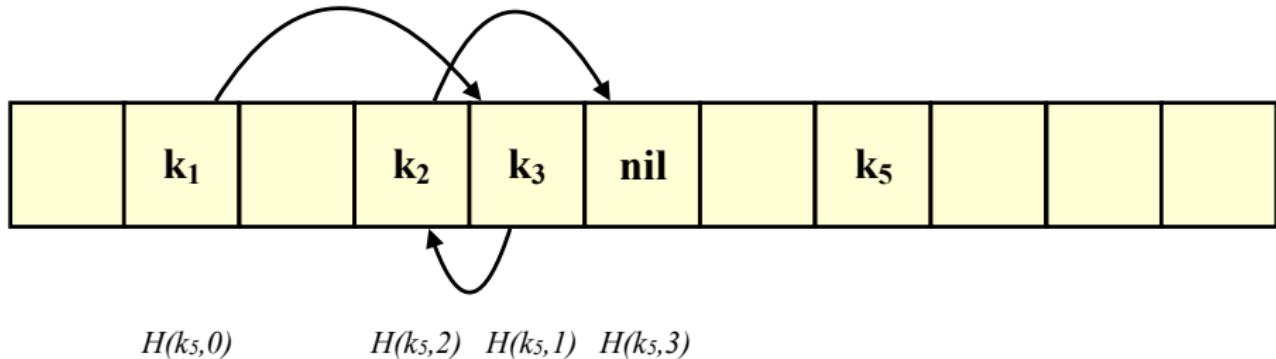
Cancellazione

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

Cancellazione

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

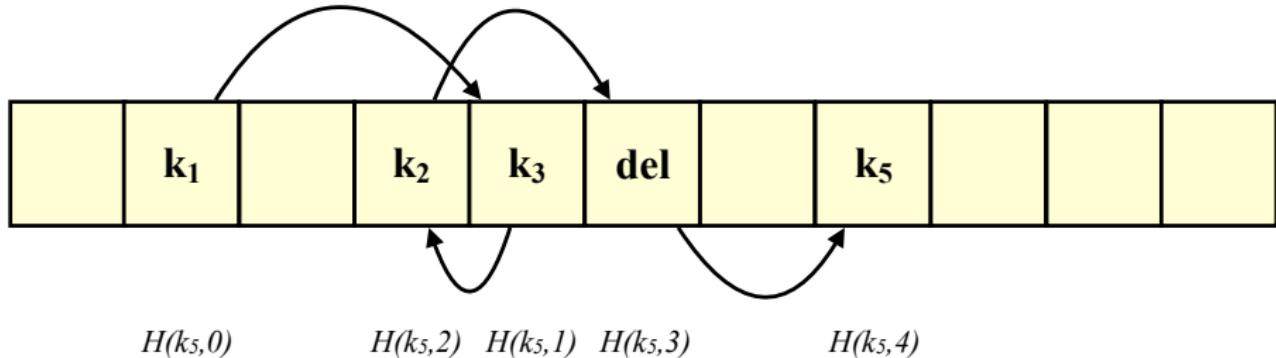
Annullerebbe l'ispezione



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 $t = \text{new HASH}$

$t.m = dim$

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

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

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

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

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 keys[j] ≠ k and keys[j] ≠ nil and i < m do
```

```
        if keys[j] == deleted and delpos == m then
```

```
            delpos = j
```

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

```
            i = i + 1
```

```
    if insert and keys[j] ≠ k and delpos < m then
```

```
        return delpos
```

```
    else
```

```
        return j
```

Implementazione - Hashing doppio

```
ITEM lookup(ITEM k)
```

```
    int i = scan(k, false)
    if keys[i] == k then
        return values[i]
    else
        return nil
```

```
insert(ITEM k, ITEM v)
```

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

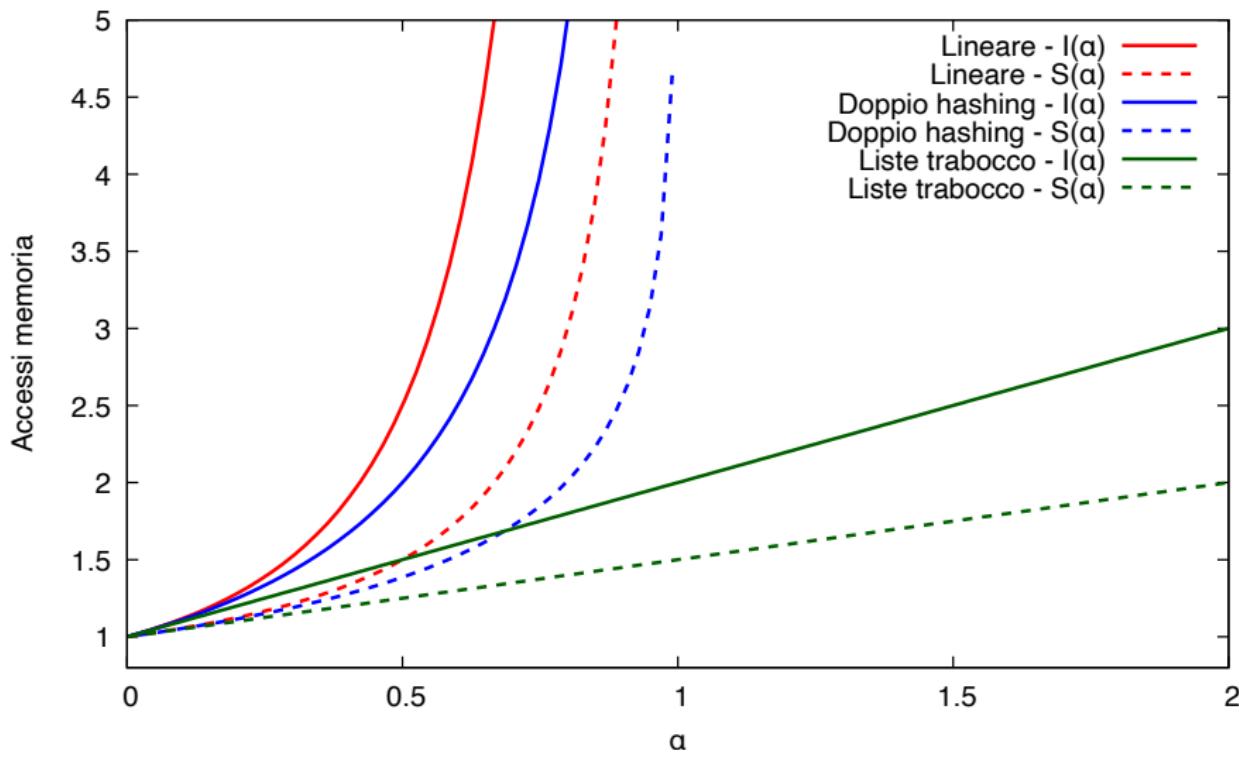
Implementazione - Hashing doppio

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

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



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`)

Il numero *magico*

Perché 31???

- 31 è un numero primo di Mersenne ($M_p = 2^p - 1$)
- Esprimiamo il prodotto generico come una somma di prodotti per potenze di 2, quale influenza ha la rappresentazione binaria del moltiplicatore?

Java hashCode()

Cosa non fare!

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

Come si fa

```
public int hashCode()
{
    return Objects.hash(this.foo, this.bar, ...);
}
```

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	

Sommario

1 Introduzione

- 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

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)