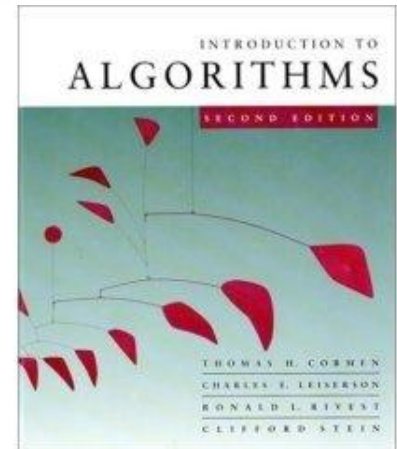




Hashing

Algoritmi, Strutture Dati e Calcolo Parallelo

- ❑ Questo materiale è tratto dalle trasparenze del corso Introduction to Algorithms (2005-fall-6046) tenuto dal Prof. Leiserson all'MIT (<http://people.csail.mit.edu/cel/>)
- ❑ E dalle trasparenze del corso "Algoritmi e Strutture Dati" del prof. Alberto Montresor dell'Università di Trento
- ❑ T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein Introduction to Algorithms, Second Edition, The MIT Press, Cambridge, Massachusetts London, England McGraw-Hill Book Company
- ❑ Queste trasparenze sono disponibili sui siti <http://webpace.elet.polimi.it/lanzi>
<http://www.slideshare.net/pierluca.lanzi>



Dizionari in cui memorizzare insiemi
dinamici di copie (chiave, valore)

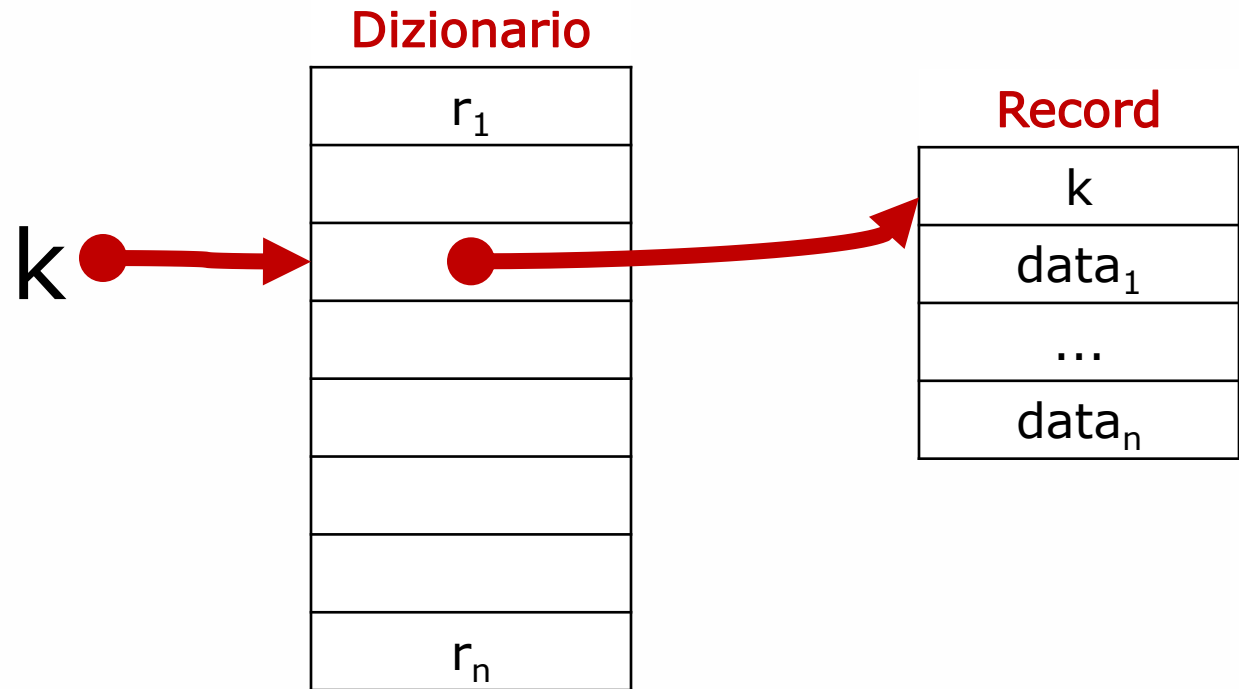
Tipiche operazioni: search, insert, delete

Tabella Hash = struttura dati efficiente
per implementare dizionari

Nel caso peggiore la ricerca di un elemento potrebbe avere
complessità $\Theta(n)$

Facendo assunzioni ragionevoli la ricerca
nel caso medio diventa $\Theta(1)$

Compilatori, Cache dei browser, Tabu-search, ...



Operazioni

- INSERT(S, x)
- DELETE(S, x)
- SEARCH(S, k)

Quale struttura dati per il dizionario?

- ❑ Supponiamo che l'universo delle chiavi possibili $U \in \{0, 1, \dots, m-1\}$, contenga m chiavi distinte
- ❑ Utilizziamo un array $T[0 \dots m-1]$ così definito
 - ▶ $T[k] = x$, se $k \in U$ e $\text{key}[x] = k$
 - ▶ $T[k] = \text{NIL}$ altrimenti
- ❑ L'operazione di accesso è $\Theta(1)$

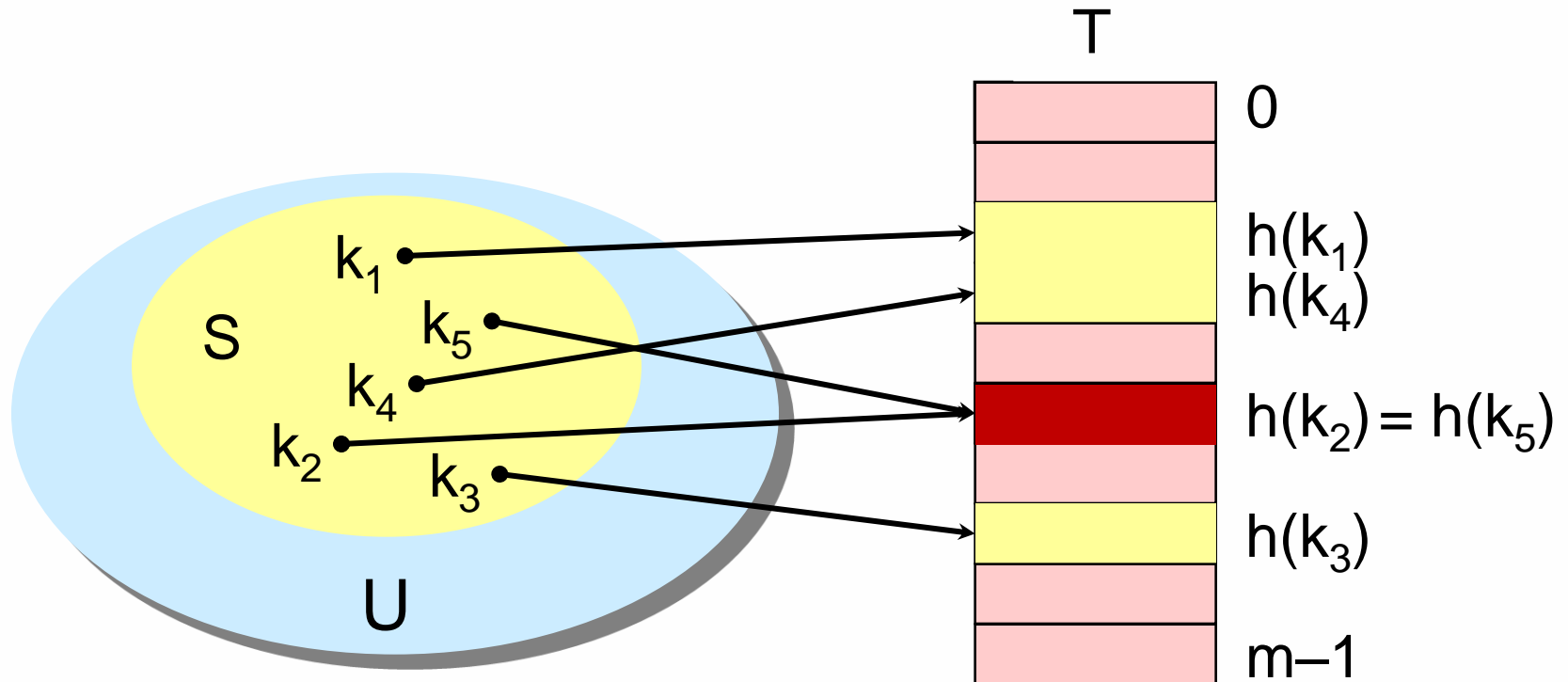
Problema: l'universo U può essere molto grande e l'array T così definito non può essere utilizzato

Esempio: una chiave a 64 bit rappresenta
18,446,744,073,709,551,616 valori

Funzioni di hash

Funzioni di Hash

Soluzione: definire una funzione di hash h che mappa l'universo U di tutte le chiavi in $\{0, 1, \dots, m-1\}$



Quando una chiave k_i mappata in uno slot già occupato si genera una **collisione**

□ Tabelle hash

- ▶ Un array $T[0..m-1]$
- ▶ Una funzione hash $h: U \rightarrow \{0, \dots, m-1\}$

□ Indirizzamento hash

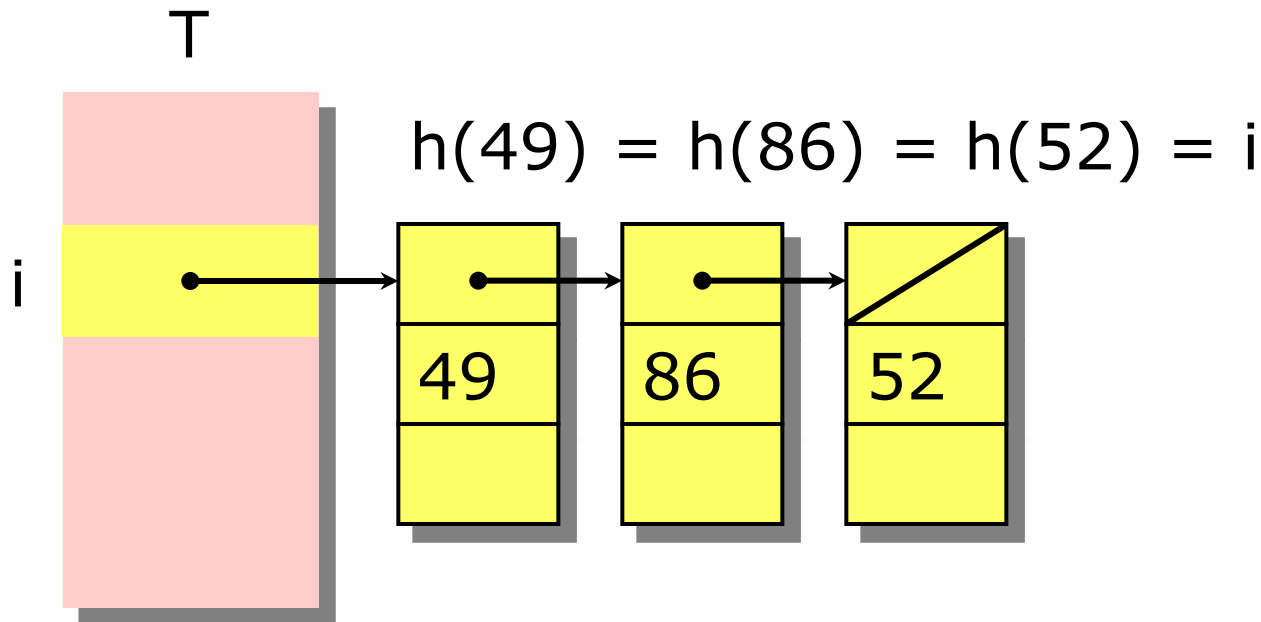
- ▶ Diciamo che $h(k)$ è il valore hash della chiave k
- ▶ Chiave k viene “mappata” nello slot $T[h(k)]$

□ Quando due o più chiavi nel dizionario hanno lo stesso valore hash, diciamo che è avvenuta una collisione

□ Idealmente: vogliamo funzioni hash senza collisioni

Come Risolviamo le Collisioni?

I record mappati nello stesso slot
sono inseriti in una lista



Caso peggiore: tutte le chiavi sono mappate nello
stesso slot, il tempo di accesso è $\Theta(n)$

- ❑ Vorremmo funzioni hash perfette
- ❑ Una funzione hash h si dice perfetta se è iniettiva, ovvero:
 - $$\forall u, z \in U \ u \neq v \rightarrow h(u) \neq h(v)$$
- ❑ Questo richiede che $m \geq |U|$
- ❑ Esempio:
 - ▶ Le matricole degli studenti di ASD degli ultimi tre anni
 - ▶ Distribuiti fra 234.717 e 235.716
 - ▶ $h(k) = k - 234.717, m = 1000$
- ❑ Problema: spazio delle chiavi spesso grande, sparso
- ❑ È spesso impraticabile ottenere una funzione hash perfetta

- ❑ **Le collisioni sono inevitabili**
 - ▶ Si cerca quindi di minimizzare il loro numero
 - ▶ Funzioni di hash che distribuiscano **uniformemente** le chiavi negli **indici [0..m-1]** della tabella hash

- ❑ **Uniformità semplice**
 - ▶ sia $P(k)$ la probabilità che una chiave k sia inserita nella tabella
 - ▶ sia $Q(i) = \sum_{k:h(k)=i} P(k)$ la probabilità che una chiave qualsiasi, finisca nella cella i .

- ❑ Una funzione h gode della proprietà di **uniformità semplice** se

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

- ❑ 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 e quindi si utilizzano euristiche

□ Assunzioni

- ▶ Tutte le chiavi sono equiprobabili: $P(k) = 1 / |U|$
- ▶ Le chiavi sono valori numerici non negativi
- ▶ È possibile trasformare una chiave complessa in un numero, "DOG" -> 'D'*256*256+'O'*256+'G'

□ Nota

- ▶ Esistono proprietà più vincolanti dell'uniformità semplice
- ▶ Ad esempio, "chiavi vicine devono essere collocate in posizioni distanti"

Metodo della divisione

❑ Metodo della Divisione

- ▶ Basata sul resto della divisione per m : $h(k) = k \bmod m$
- ▶ Esempio: $m=12, k=100 \rightarrow h(k) = 4$

❑ **Vantaggio:** molto veloce (richiede solo una divisione)

❑ **Svantaggio:** il valore m deve essere scelto opportunamente

❑ 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

❑ **Vanno bene:** Numeri primi, distanti da potenze di 2 (e di 10)

Funzioni Hash: Metodo della Moltiplicazione

- ❑ **Metodo della moltiplicazione**
 - ▶ Una costante A , $0 < A < 1$, $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
 - ▶ Moltiplichiamo k per A e prendiamo la parte frazionaria che moltiplichiamo per m e prendiamo la parte intera
 - ▶ Esempio: $m = 1000$, $k = 123$, $A \approx 0.6180339... \rightarrow h(k) = 18$

- ❑ **Svantaggi:** più lento del metodo di divisione
- ❑ **Vantaggi:** il valore di m non è critico

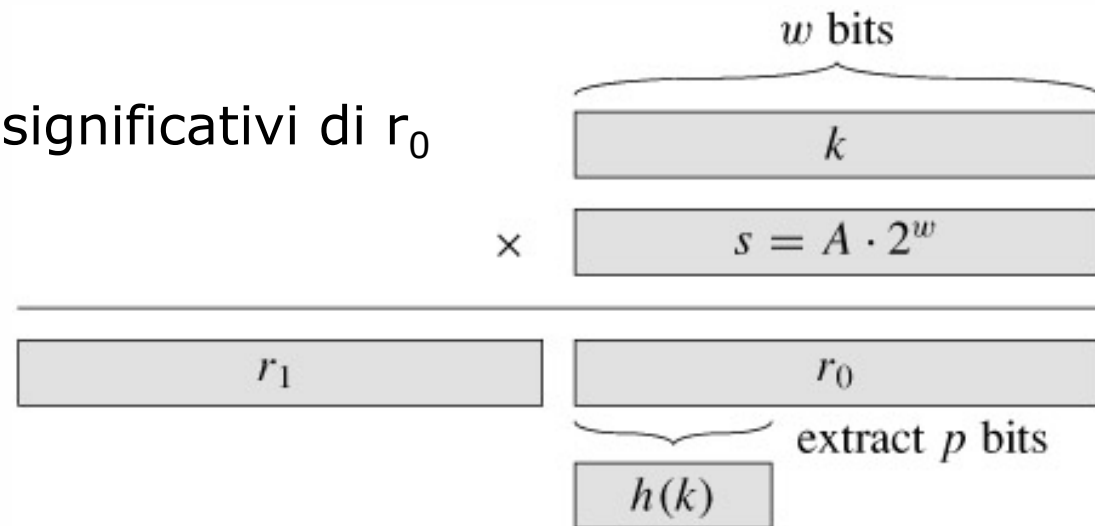
- ❑ Si può scegliere una potenza di 2 ($m = 2^p$), che semplifica l'implementazione

- ❑ Come scegliere A ? Knuth suggerisce $A \approx (\sqrt{5} - 1)/2$

Implementazione del Metodo della Moltiplicazione

- Si sceglie un valore $m=2^p$
- Sia w la dimensione in bit della parola di memoria:
 $k, m \leq 2^w$
- Sia $s = \lfloor A2^w \rfloor$
 - ▶ ks può essere scritto come $r_12^w + r_0$
 - ▶ 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



- ❑ Una buona funzione di hash può ridurre, ma non elimina le numero di collisioni

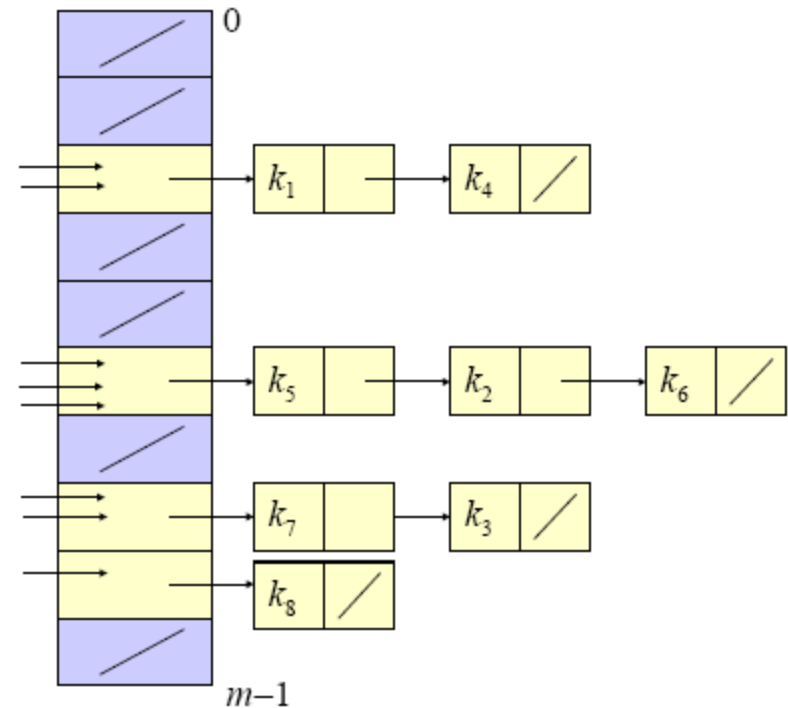
- ❑ Come gestire le collisioni residue?
 - ▶ Dobbiamo trovare collocazioni alternative per le chiavi
 - ▶ Se una chiave non si trova nella posizione attesa, bisogna andare a cercare nelle posizioni alternative
 - ▶ Le operazioni possono costare $\Theta(n)$ nel caso peggiore...
 - ▶ ...ma hanno costo $\Theta(1)$ nel caso medio

- ❑ Due tecniche:
 - ▶ **Concatenamento**
 - ▶ **Indirizzamento aperto**

Concatenamento

Risoluzione delle collisioni: Concatenamento (chaining)

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



□ Notazione

- ▶ n : # elementi nella tabella
- ▶ m : # slot nella tabella

□ Fattore di carico

- ▶ α : # medio di elementi nelle liste ($\alpha = n/m$)

□ Caso pessimo

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

□ Caso medio

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

- ❑ **Teorema:** In una tabella hash con concatenamento, una ricerca senza successo richiede un tempo atteso di $\Theta(1 + \alpha)$

- ❑ **Teorema:** In una tabella hash con concatenamento, una ricerca con successo richiede un tempo atteso di $\Theta(1 + \alpha)$. Più precisamente $\Theta(2 + \alpha/2 + \alpha/2n)$, dove n è il numero di elementi

- ❑ **Qual è il significato?**
 - ▶ costo della funzione di hashing
 - ▶ se $n = O(m)$, $\alpha = O(1)$
 - ▶ quindi tutte le operazioni sono $\Theta(1)$

Indirizzamento aperto

La gestione delle collisioni tramite concatenamento richiede una struttura dati complessa, con liste, puntatori, ecc.

- ❑ Idea: memorizzare tutte le chiavi nella tabella stessa, non necessita di memoria oltre la tabella stessa
- ❑ Ogni slot contiene una chiave oppure nil
- ❑ Inserimento: Se lo slot prescelto è utilizzato, si cerca uno slot "alternativo" generando una **sequenza di probing**
- ❑ Ricerca: Si cerca nello slot prescelto, e poi negli slot "alternativi" fino a quando non si trova la chiave oppure nil

Indirizzamento Aperto: Soluzione delle Collisioni

- ❑ L'inserimento aperto sistematicamente esamina la tabella fino a quando uno slot vuoto viene trovato
- ❑ La funzione di hash dipende sia dalla chiave $h(k,0)$ che dall'indice di probing $h(k,i)$:

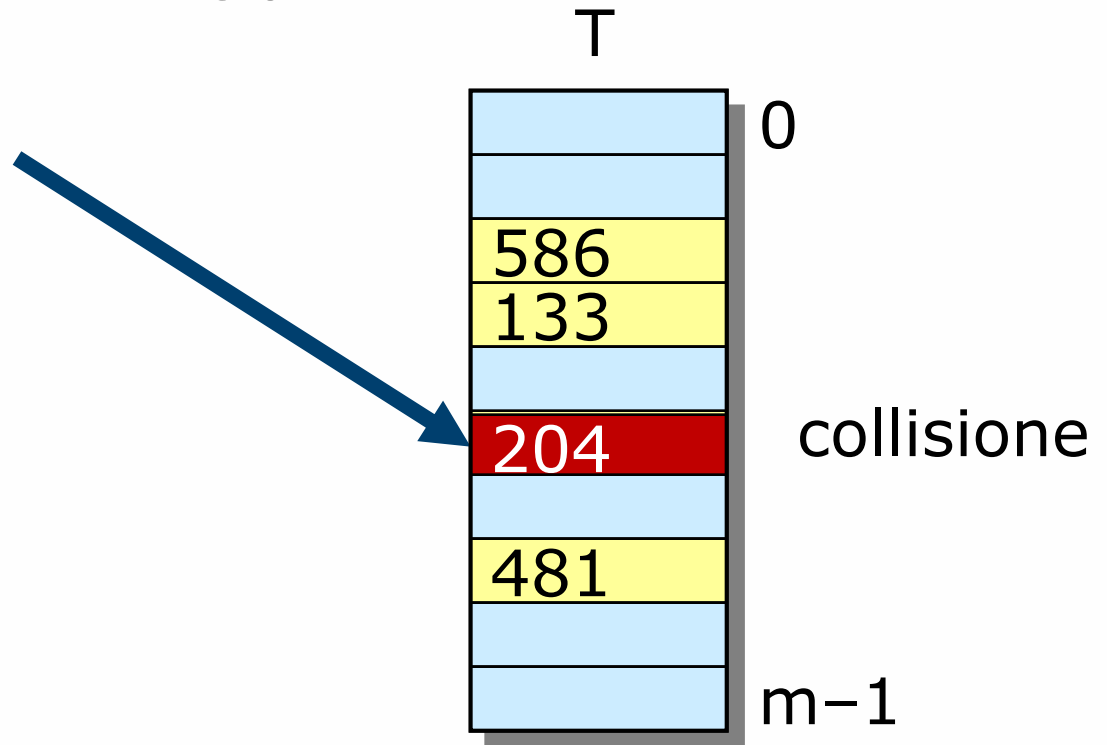
$$h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

- ❑ La sequenza di probing $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$ deve essere una permutazione di $\{0, 1, \dots, m-1\}$
- ❑ **Nota**
 - ▶ Può essere necessario esaminare ogni slot nella tabella
 - ▶ Non vogliamo esaminare ogni slot più di una volta
 - ▶ La tabella tende a riempirsi e la cancellazione può essere complicata

Esempio

Inserisce la chiave $k = 496$:

0. Probe $h(496, 0)$

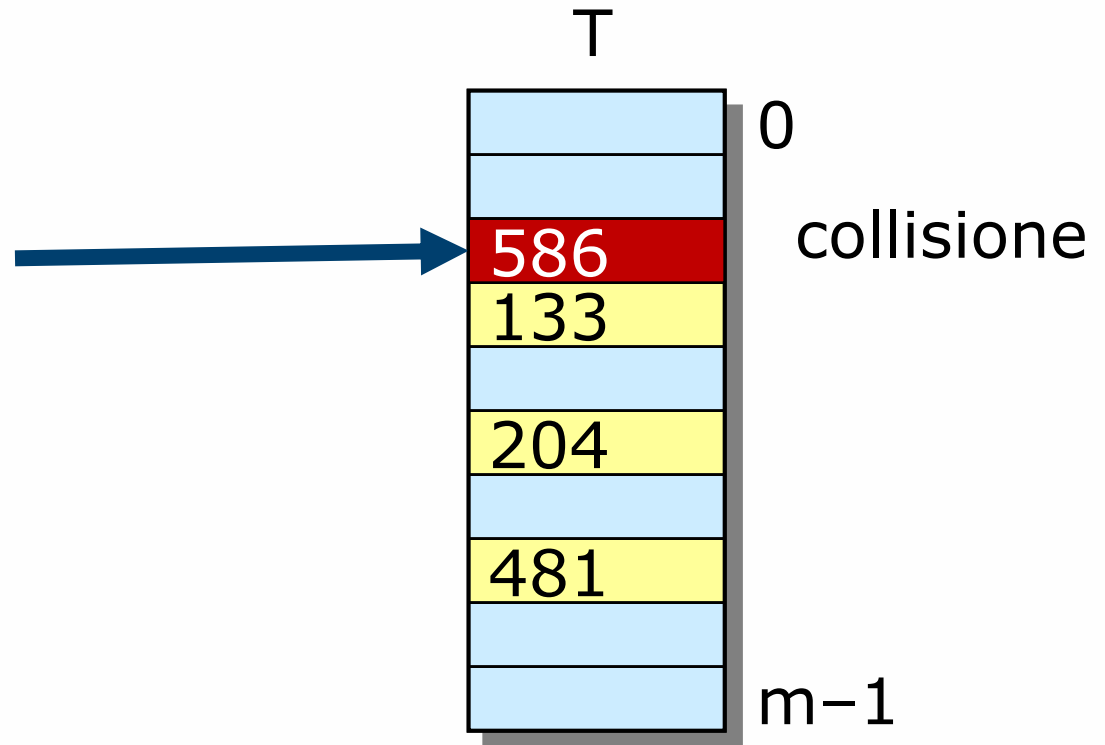


Esempio

Inserisce $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$



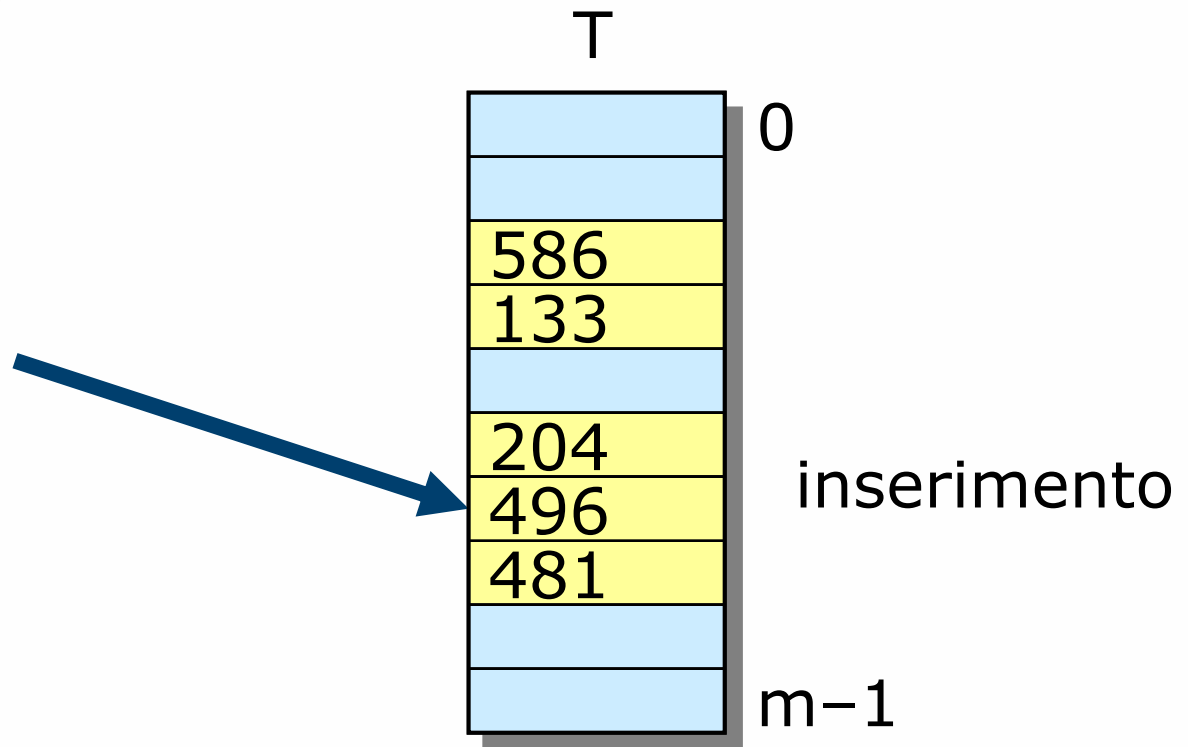
Example of open addressing

Inserisce $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$

2. Probe $h(496,2)$



Esempio

Sequenza di probing per $k = 496$:

0. Probe $h(496,0)$

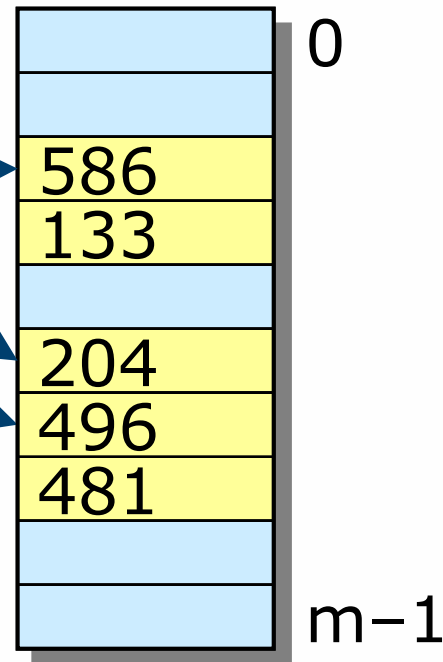
1. Probe $h(496,1)$

2. Probe $h(496,2)$

La ricerca utilizza la stessa sequenza di probing

La ricerca ha successo se trova la chiave e fallisce se incontra uno slot vuoto

T



Hash-Insert(A, k)

1. $i := 0$
2. repeat $j := h(k, i)$
3. if $A[j] = \text{nil}$ then
4. $A[j] := k$
5. return j
6. else
7. $i := i + 1$
8. until $i = m$
9. error "hash table overflow"

Hash-Search (A, k)

1. $i := 0$
2. repeat $j := h(k, i)$
3. if $A[j] = k$ then
4. return j
5. $i := i + 1$
6. until $A[j] = \text{nil}$ or $i = m$
7. return nil

Non è possibile sostituire la chiave che vogliamo cancellare con un nil (perché?)

- ❑ **Soluzione**
 - ▶ Utilizziamo un speciale valore DELETED (una tombstone) 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

Probing

- ❑ Hashing Uniforme (ideale)
 - ▶ Generalizzazione dell'hashing uniforme semplice
 - ▶ Ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una qualsiasi delle $m!$ permutazioni di $[0..m-1]$
 - ▶ Difficile da implementare
 - ▶ Ci si accontenta di ottenere semplici permutazioni

- ❑ Tecniche diffuse:
 - ▶ Ispezione lineare
 - ▶ Ispezione quadratica
 - ▶ Doppio hashing

❑ **Funzione di hash:** $h(k,i) = (h'(k)+i) \bmod m$

Funzione hash
ausiliaria

❑ Il primo elemento determina l'intera sequenza

- ▶ $h'(k), h'(k)+1, \dots, m-1, 0, 1, \dots, h'(k)-1$
- ▶ Solo m sequenze di ispezione distinte

❑ **Clustering Primario**

- ▶ Lunghe sotto-sequenze occupate che tendono a diventare via via sempre 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

- ❑ **Funzione:** $h(k,i) = (h'(k) + c_1i + c_2i^2) \bmod m$ con $c_1 \neq c_2$

- ❑ Sequenza di ispezioni
 - ▶ L'ispezione iniziale è in $h'(k)$
 - ▶ Le ispezioni successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione i
 - ▶ Solo m sequenze di ispezione distinte sono possibili

- ❑ c_1, c_2, m devono essere scelti in modo da garantire la permutazione di $[0..m-1]$

- ❑ **Clustering secondario**
 - ▶ Se due chiavi hanno la stessa ispezione iniziale, poi le loro sequenze sono identiche

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

- ❑ Due funzioni ausiliarie:
 - ▶ h_1 fornisce la prima ispezione
 - ▶ h_2 fornisce l'offset delle successive ispezioni
 - ▶ m^2 sequenze di ispezione distinte sono possibili

- ❑ Nota: 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

□ Assunzioni

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

□ Analisi

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

- ❑ **Teorema 1:** Il numero atteso di ispezioni per una ricerca senza successo è al massimo $1/(1-\alpha)$
- ❑ **Teorema 2:** Il numero atteso di ispezioni per una ricerca con successo è al massimo $1/\alpha \ln 1/(1-\alpha)$
- ❑ **Teorema 3:** Il numero atteso di ispezioni per un inserimento è al massimo $1/(1-\alpha)$

- ❑ Implicazioni
 - ▶ Se α è costante, allora l'accesso a una tabella hash con indirizzamento aperto richiede un tempo costante
 - ▶ Se la tabella è mezza piena, allora il numero atteso di ispezioni è $1/(1-0.5) = 2$.
 - ▶ Se la tabella è piena al 90%, allora il numero atteso di ispezioni è $1/(1-0.9) = 10$.

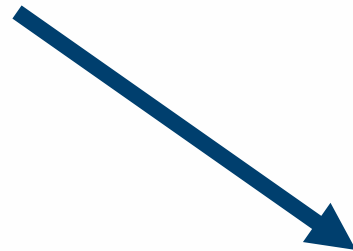
- ❑ **Problema:** Per una qualsiasi funzione di hash h , può essere generato un insieme di chiavi che causa un alto tempo medio di accesso.
- ❑ Ad esempio, un utente potrebbe scegliere un insieme di n chiavi che si mappano tutte sullo stesso slot portando il tempo di accesso a $\Theta(n)$
- ❑ **Soluzione:** la funzione di hash viene scelta in maniera casuale in maniera indipendente dalle chiavi che poi verranno memorizzate

Universal Hashing

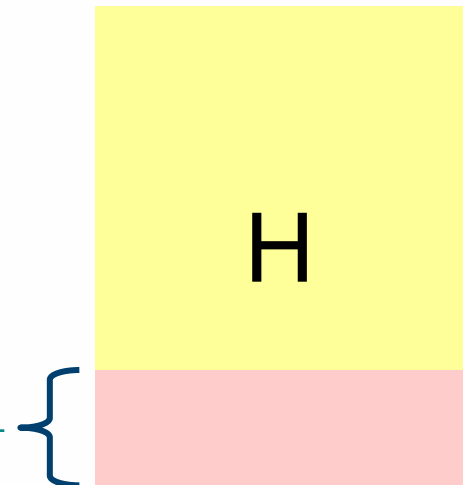
Universal Hashing

- **Definizione.** Dato l'universo di chiavi U e una collezione finita H di funzioni di hash, ciascuna delle quali mappa U in $\{0, 1, \dots, m-1\}$. Diciamo che H è universale se per tutti gli $x, y \in U$, $x \neq y$, abbiamo $|\{h \in H : h(x) = h(y)\}| \leq |H| / m$
- Ovvero, H è universale se la probabilità di collisione fra x e y è $\leq 1/m$ se scegliamo la funzione di hash fra quelle di H

$$\{h : h(x) = h(y)\}$$



$$\frac{|H|}{m}$$



- Teorema. Data una funzione di hash h scelta in maniera casuale (con probabilità uniforme) da un insieme universale di funzioni di hash H . Se h è utilizzata per mappare n chiavi negli m slot (con $n \leq m$) della tabella T , allora, per una chiave x abbiamo,

$$E[\# \text{ collisioni per } x] < n/m \text{ (ovvero } < 1)$$

Come fare il design di H ?

- ❑ Scegliamo m dall'insieme dei numeri primi e scomponiamo la chiave k in $r + 1$ digit (o byte o sottosequenze binarie) con valore in $\{0, 1, \dots, m-1\}$.
- ❑ La chiave k può essere rappresentata come $\langle k_0, k_1, \dots, k_r \rangle$, dove $0 \leq k_i < m$
- ❑ **Randomized strategy**
 - ▶ Scegliamo $a = \langle a_0, a_1, \dots, a_r \rangle$ dove a_i è scelto in maniera casuale da $\{0, 1, \dots, m-1\}$
 - ▶ Definiamo $h_a(k) = \sum_{i \in 0..r} a_i k_i \text{ mod } m$
 - ▶ Definiamo $H = \{h_a(k)\}$ e quindi $|H| = m^{r+1}$
- ❑ **Teorema:** $H = \{h_a(k)\}$ è universale

Sommario

□ Dizionari

- ▶ Strutture dati in cui memorizzare coppie (chiave, valore).
- ▶ Tipiche operazioni: search, insert, delete

□ Tabella Hash

- ▶ Struttura dati efficiente per implementare dizionari
- ▶ Funzione hash mappa l'universo U di tutte le chiavi nelle posizioni $\{0, 1, \dots, m-1\}$ della tabella di hash T

□ Collisioni

- ▶ La funzione di hash mappa due chiavi nella stessa posizione
- ▶ Concatenamento (lista puntata delle collisioni)
- ▶ Indirizzamento aperto (probing lineare, quadratico, o doppio hashing)