

TABELLE HASH

---

**INFORMATICA**

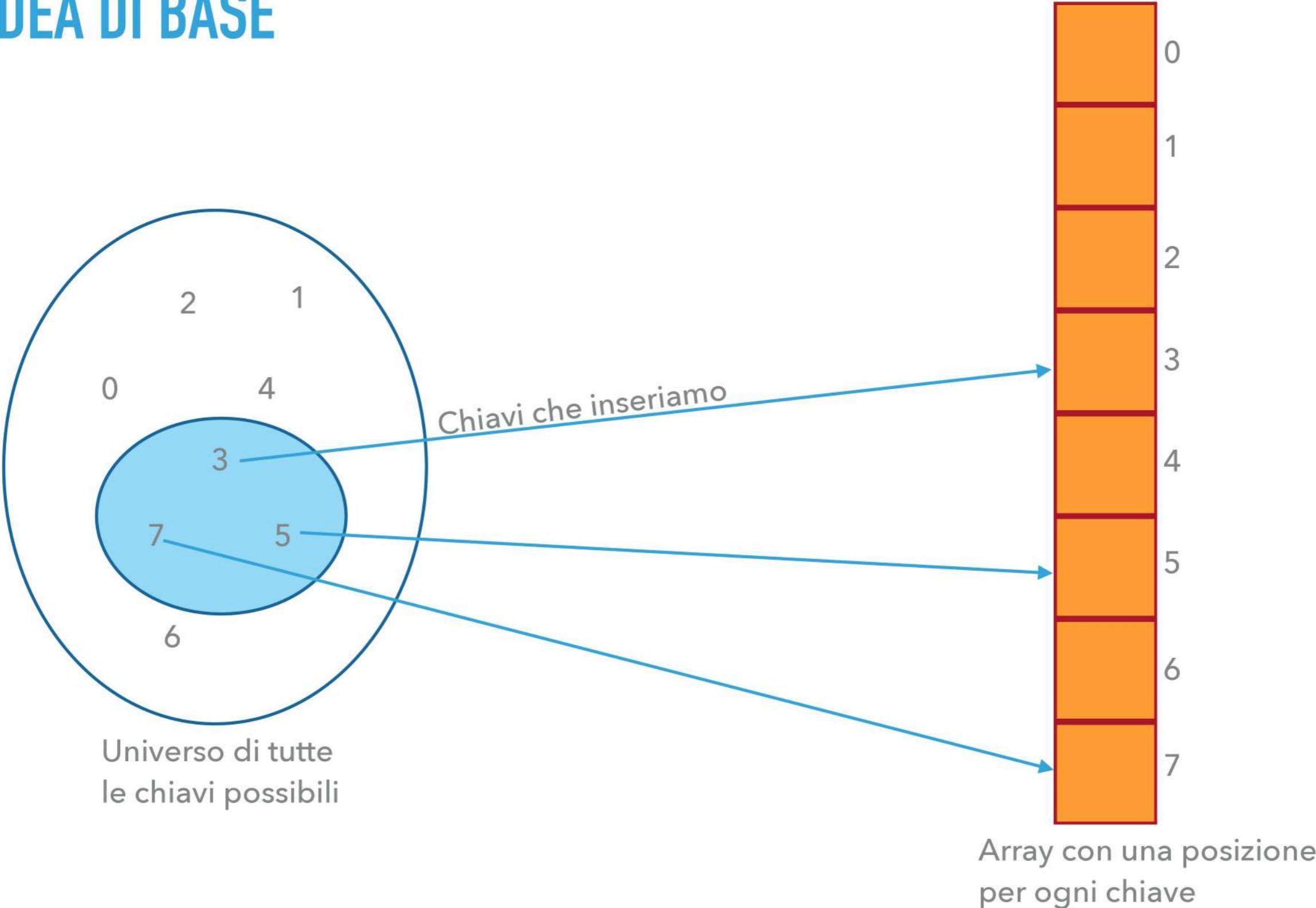
## DIZIONARI: OLTRE GLI ALBERI BINARI DI RICERCA

- ▶ Abbiamo visto come possiamo implementare un dizionario usando alberi binari di ricerca
- ▶ Con gli alberi bilanciati (AVL, rosso-neri) possiamo ottenere tempi di ricerca, inserimento e rimozione  $O(\log n)$
- ▶ Possiamo fare di meglio se, per esempio, assumiamo di non dover implementare minimo, massimo, successore, predecessore?

## DIZIONARI: OLTRE GLI ALBERI BINARI DI RICERCA

- ▶ Supponiamo di conoscere tutti i possibili valori che una chiave può assumere:  $K = \{k_1, \dots, k_m\}$ . Assumiamo  $K = \{0, \dots, m - 1\}$
- ▶ Se costruiamo un array di  $m$  elementi possiamo assegnare ad ogni chiave uno slot.
  - ▶ Inserimento: mettere il valore nello slot  $k_i$
  - ▶ Ricerca: ritornare il contenuto dello slot  $k_i$
  - ▶ Rimozione: cancellare il valore contenuto nello slot  $k_i$

# IDEA DI BASE



# OPERAZIONI DEI DIZIONARI

### **Inserimento**

Parametri:  $x$  (l'oggetto da inserire) e la tabella  $T$

$T[x.key] = v$

### **Ricerca**

Parametri:  $k$  (la chiave) e la tabella  $T$

return  $T[k]$

### **Rimozione**

Parametri:  $x$  (l'oggetto da rimuovere) e la tabella  $T$

$T[x.key] = \text{None}$

### CI SONO DEI PROBLEMI?

- ▶ Questo approccio ci permette di fare ricerca, inserimento e rimozione in tempo costante!
- ▶ Problema: l'insieme di tutte le chiavi potrebbe non essere abbastanza piccolo da permettere questo approccio.
  - ▶ Esempio: con numeri di 32 bit si avrebbero circa quattro miliardi di slot, anche salvando solo un byte per ogni slot avremmo 4GB di memoria occupati!
- ▶ Possiamo riformulare l'idea in modo che possa funzionare?

## TABELLE HASH

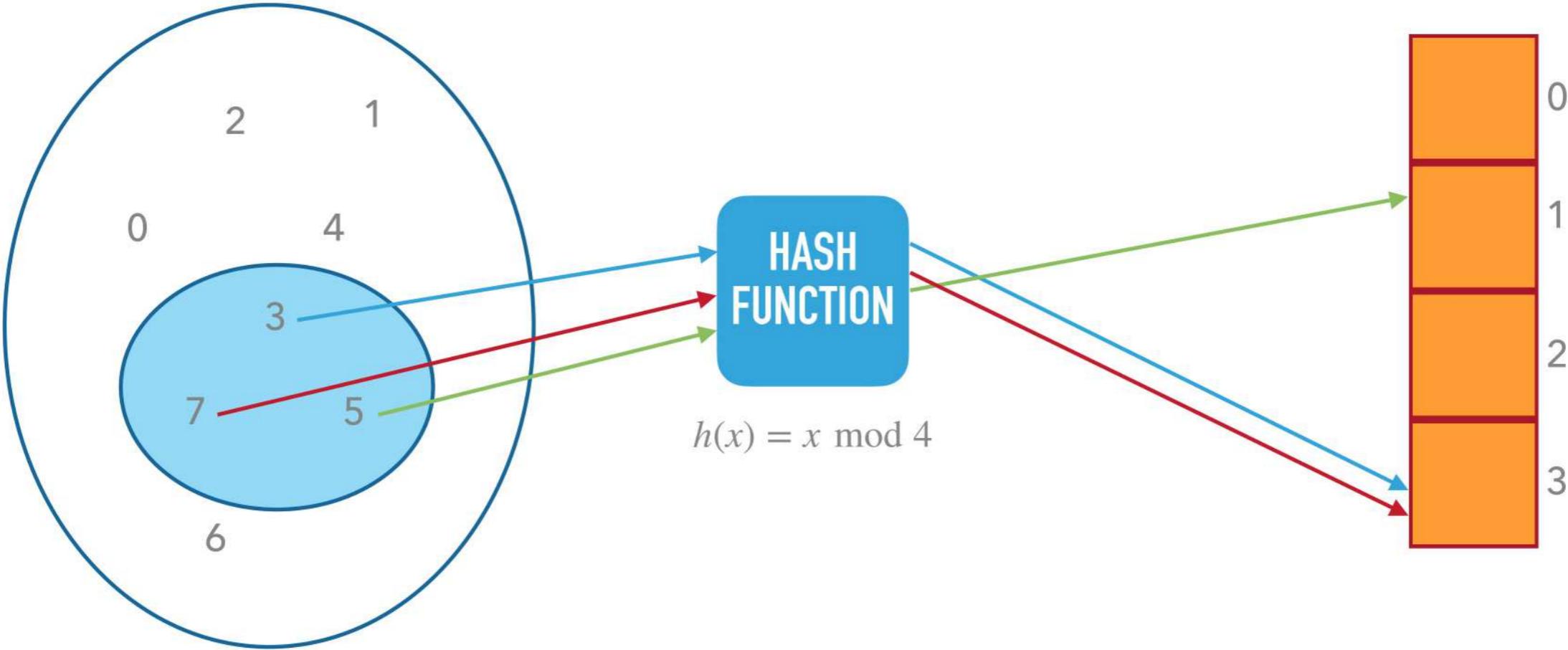
$U \equiv$  UNIVERSO delle chiavi

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

$m = |T|$

- ▶ Invece di utilizzare direttamente le chiavi come indici, usiamo una funzione (detta *funzione di hash*) che, data una chiave, ci dice dove trovarla all'interno di una tabella
- ▶ Questo ci permette di definire quelle che sono chiamate le *tabelle hash*:
  - ▶ Dato un array, una chiave  $k$ , ed una funzione di hash  $h$ , la posizione in cui inserire/trovare  $k$  nell'array è  $h(k)$ .
  - ▶ Il condominio di  $h$  può essere molto ridotto!

# UNA PRIMA TABELLA HASH



Universo di tutte le chiavi possibili

# TABELLE HASH

- ▶ Finché non abbiamo due chiavi distinte con lo stesso hash (i.e.,  $k_1 \neq k_2$  ma  $h(k_1) = h(k_2)$ ) tutte le operazioni continuano ad essere effettuabili in tempo costante (assumendo che  $h$  richieda tempo costante)
- ▶ Però dobbiamo gestire questo caso (le **collisioni**).
- ▶ A seconda di come decidiamo di gestirlo abbiamo diverse varianti di tabelle hash.

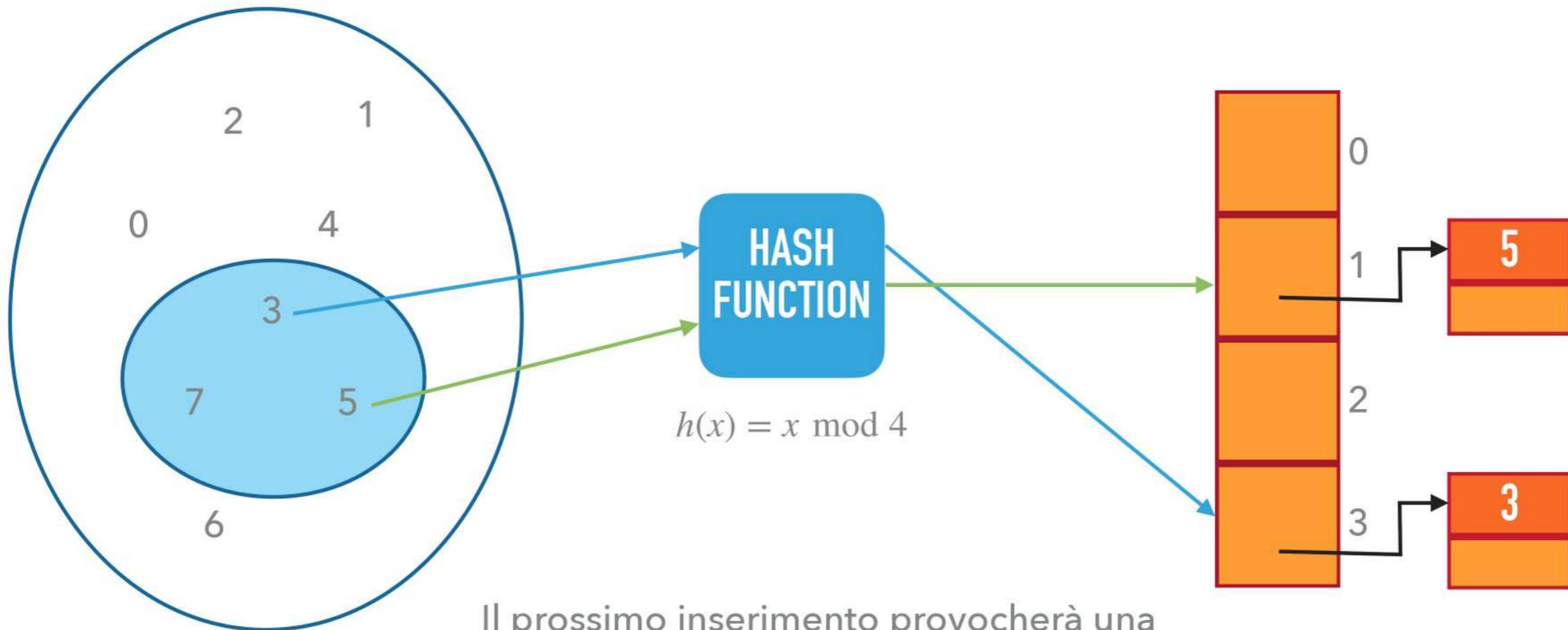
# GESTIRE LE COLLISIONI

- ▶ Possiamo tenere per ogni slot una lista concatenata di valori che hanno lo stesso hash: **chaining** o **"hash con concatenazione"**
- ▶ Possiamo invece cercare un altro posto libero nella tabella: **open addressing** o **indirizzamento aperto**. Ne esistono diverse varianti, tra cui:
  - ▶ Ispezione lineare o quadratica
  - ▶ Doppio hashing

## CHAINING / CONCATENAZIONE

- ▶ Abbiamo un array di  $m$  elementi, ognuno una lista concatenata (di solito doppia)
- ▶ L'inserimento di un elemento  $x$  di chiave  $k$  si riconduce a un inserimento in testa alla lista di indice  $h(k)$
- ▶ La rimozione di un elemento  $x$  di chiave  $k$  si riconduce a una rimozione dalla lista di indice  $h(k)$
- ▶ La ricerca data una chiave  $k$  si riconduce alla ricerca di  $k$  nella lista di indice  $h(k)$

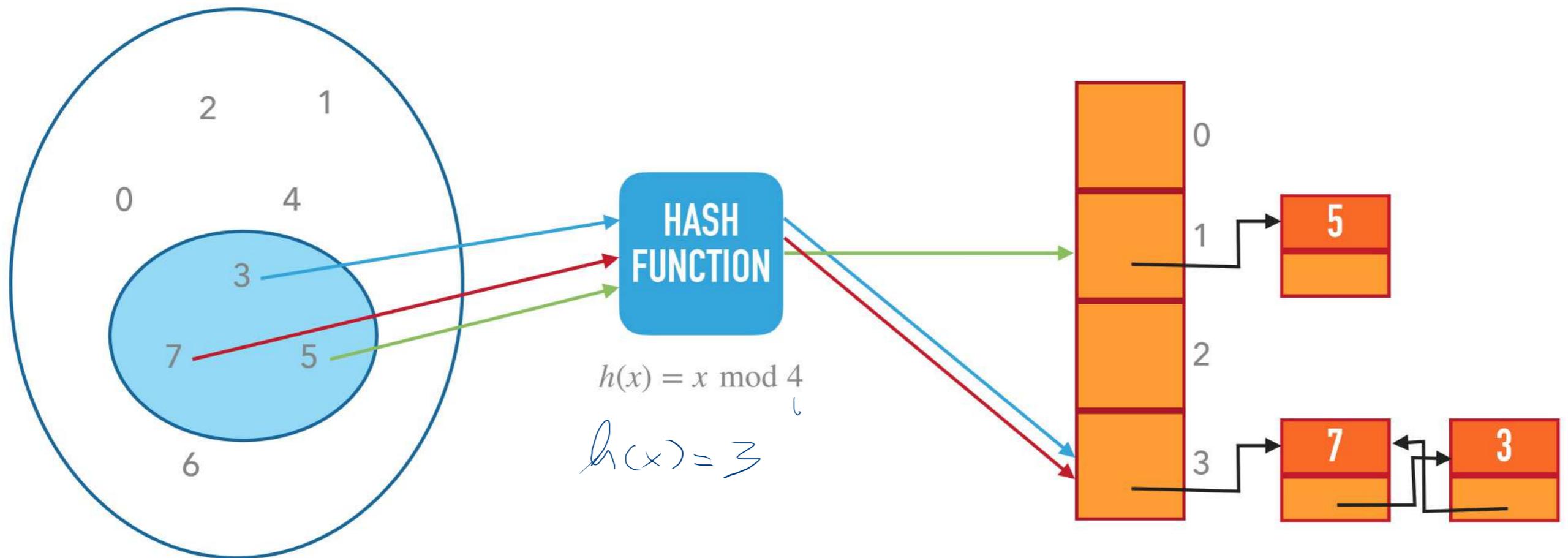
# HASH CON CHAINING



Il prossimo inserimento provocherà una collisione. Come lo gestiamo?

# HASH CON CHAINING

Ricerca  $O(n)$  caso peggiore



Effettuiamo l'inserimento in testa per avere sempre inserimenti in  $O(1)$ .  
 Usiamo una lista concatenata doppia per avere rimozioni in  $O(1)$ .

# CHAINING / CONCATENAZIONE

### **Inserimento**

Parametri:  $x$  (l'oggetto da inserire) e la tabella  $T$   
inserisci  $x$  in testa a  $T[h(x.key)]$

### **Ricerca**

Parametri:  $k$  (la chiave) e la tabella  $T$   
ricerca lineare nella lista  $T[h(k)]$

### **Rimozione**

Parametri:  $x$  (l'oggetto da rimuovere) e la tabella  $T$   
rimuovi  $x$  da  $T[h(x.key)]$

## CHAINING / CONCATENAZIONE

- ▶ Perché questo ci dovrebbe aiutare?
- ▶ Nel caso peggiore abbiamo tutti gli  $n$  elementi nella stessa lista concatenata
- ▶ Quindi, anche se inserimento e rimozione richiedono tempo  $O(1)$ , la ricerca richiede invece tempo  $O(n)$
- ▶ L'analisi è corretta per il caso peggiore, ma possiamo ottenere qualcosa di meglio guardando il tempo medio?

# TABELLE HASH: ALCUNE DEFINIZIONI

- ▶ Come al solito, indichiamo con  $n$  il numero di elementi contenuti nella tabella
- ▶ Indichiamo con  $m$  la dimensione della tabella
- ▶  $\alpha = n/m$  è il **load factor** o **fattore di carico** della tabella.
- ▶ Il fattore di carico indica quanto "piena" è la tabella:
  - ▶  $\alpha < 1$  abbiamo più posti nella tabella che elementi inseriti
  - ▶  $\alpha > 1$  abbiamo più elementi inseriti che posti nella tabella

## CHAINING / CONCATENAZIONE

- ▶ Assumiamo che la funzione di hash distribuisca uniformemente le chiavi negli  $m$  slot.
- ▶ Sotto queste assunzioni, mostriamo che il tempo medio per cercare un elemento in una tabella hash è  $\Theta(1 + \alpha)$
- ▶ Dividiamo la dimostrazione in due parti:
  - ▶ Il caso in cui l'elemento cercato non sia nella tabella
  - ▶ Il caso in cui l'elemento cercato sia nella tabella

# ANALISI PROBABILISTICA DI COMPLESSITÀ

$$\mathcal{X} = \mathcal{X}_1 \cup \dots \cup \mathcal{X}_n \cup \dots \quad \mathcal{Y} \text{ è output}$$

$\uparrow$  INPUT                       $\uparrow$  INPUT di dimensione  $n$

$$g: \mathcal{X} \rightarrow \mathcal{Y} \quad \text{costo di } g(x) \quad |g(x)| = n$$

$\text{cost}(g, x)$

$$T(n) = \max_{x \in \mathcal{X}_n} \text{cost}(g, x) \quad \text{in ordine di crescita con } n$$

ASSOCIO A  $\mathcal{X}_n$  una probabilità, per ogni  $n$ .       $|\mathcal{X}_n| < \infty$

$$p: \mathcal{X}_n \rightarrow [0, 1] \quad \sum_{x \in \mathcal{X}_n} p(x) = 1 \quad p(A) = \sum_{x \in A} p(x)$$

$A \in \mathcal{X}_n$

$$E[\text{cost}(g, x)] = \sum_{x \in \mathcal{X}_n} \text{cost}(g, x) p(x)$$

# ANALISI MEDIA TABELLE di HASH

$U$  univ. delle chiavi,  $|U| < \infty$

$p: U \rightarrow [0, 1]$  probabilità  $p(k)$

$k \rightarrow h(k) \in \{0, \dots, m-1\}$

$$P(j) = \sum_{k: h(k)=j} p(k)$$

$j \in \{0, \dots, m-1\}$

$p: \{0, \dots, m-1\} \rightarrow [0, 1]$  sia UNIFORME

$$P(j) = \frac{1}{m} \quad \forall j \in \{0, \dots, m-1\}$$

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



SCELGO  $j \in \{0, \dots, m-1\}$

$$\{x \in U \mid h(x) = j\} = U_j$$

$$n_j = |U_j|$$

$$P(x) = \begin{cases} 1/n_j & \text{se } x \in U_j \\ 0 & \text{otherwise} \end{cases}$$

$$P(i) = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{otherwise} \end{cases}$$

## CHAINING / CONCATENAZIONE

Se l'elemento cercato non è presente, la chiave  $k$  con cui lo cerchiamo ha uguale probabilità di finire in uno qualsiasi degli  $m$  slot.

Quindi il tempo atteso per scoprire che la chiave non è presente è dato dalla lunghezza attesa della lista di indice  $h(k)$

Ma la lunghezza attesa della lista è esattamente il fattore di carico

$$\alpha = n/m.$$

Quindi il tempo atteso richiesto è un numero costante di passi più il tempo di cercare una lista di lunghezza  $\alpha$ :  $\Theta(1 + \alpha)$

(VARIABILE ALEATORIA)

$$X: \underbrace{(\Omega)} \rightarrow \mathbb{R} \quad \Leftrightarrow \quad P_{\mathbb{R}}(z) = P_{\Omega}(X^{-1}(z))$$

$$X_{ij} = \begin{cases} 1, & \text{se la } i\text{-esima chiave inserita \bar{e} inserita in posizione } j \\ 0, & \text{otherwise} \end{cases}$$

$K_1 \dots K_n$   $n$  chiavi inserite

$$h(K_i) = j \Leftrightarrow X_{ij} = 1 \quad \sum_{j=0}^{m-1} X_{ij} = 1$$

$$\underline{N_{\bar{j}}} = \sum_{i=1}^n X_{ij}$$

$$\underline{E[N_{\bar{j}}]} = E\left[\sum_{i=1}^n X_{ij}\right] = \sum_{i=1}^n \underline{E[X_{ij}]} = \sum_{i=1}^n P(X_{ij}=1) = \frac{n}{m} = \alpha$$

$$E[X] = 0 \cdot P(X=0) + 1 \cdot P(X=1) = P(X=1)$$

$$P(X_{ij}=1) = \frac{1}{m}$$

per uniformit\`a del

$N$  ce valori in  $\{0, \dots, n\}$ ,  $\underbrace{P(N=j)}_{P(j)} = P_j$ ,  $\sum_{j=0}^n P_j = 1$

$$E[N] = \sum_{j=0}^n j \cdot P_j$$

$N_1, N_2 \in \{0, \dots, n\}$   $P(N_1=j)$   $P(N_2=i)$

$$E[N_1 + N_2] = \sum (j+i) P_j P_i = \sum j P_j + \sum i P_i$$

$$\begin{aligned} & \left[ \sum P_i = 1 \right] \\ & = \underbrace{\sum_{j,i} j P_j P_i}_{\sum_j j P_j \left( \sum_i P_i \right)} + \sum_{j,i} i P_j P_i \end{aligned}$$

## CHAINING / CONCATENAZIONE

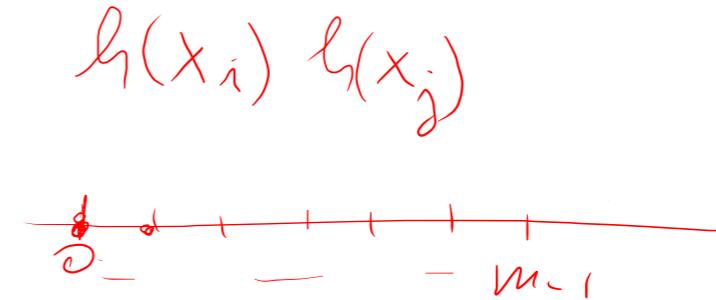
Se assumiamo che l'elemento  $x$  di chiave  $k$  sia presente nella lista, allora è egualmente probabile che sia uno qualsiasi degli  $n$  elementi presenti nella lista.

Gli elementi sono inseriti in testa alla lista, quindi il tempo richiesto per trovare  $x$  dipende da quanti elementi con lo stesso hash sono stati inseriti dopo di lui

Siano  $x_1, \dots, x_n$  gli elementi inseriti di chiavi  $k_1, \dots, k_n$

Indichiamo con  $X_{i,j} = 1$  il caso  $h(k_i) = h(k_j)$  e  $X_{i,j} = 0$  altrimenti  
 $i, j \in \{1, \dots, n\}$

## CHAINING / CONCATENAZIONE



Il numero di elementi da visitare prima di trovare  $x_i$  sarà quindi:

$$1 + \sum_{j=i+1}^n X_{i,j}$$

$$P(X_{i,j} = 1) = \frac{1}{m}$$

ovvero uno più tutti gli elementi inseriti dopo di lui

Dato che  $x$  potrebbe essere uno qualsiasi degli  $x_i$ , facciamo la media su tutte le possibili posizioni in cui potrebbe essere  $x$ , ottenendo il seguente tempo atteso:

$$E \left[ \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n X_{i,j} \right) \right]$$

## CHAINING / CONCATENAZIONE

Possiamo portare dentro il valore atteso per linearità:

$$\frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n E[X_{i,j}] \right)$$

Dato che assumiamo che la funzione di hash distribuisca in modo uniforme le chiavi, abbiamo che la probabilità che  $X_{i,j}$

sia 1 è  $\frac{1}{m}$ , quindi  $E[X_{i,j}] = \frac{1}{m}$

## CHAINING / CONCATENAZIONE

Otteniamo quindi

$$\frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n \frac{1}{m} \right) = \frac{1}{n} \sum_{i=1}^n 1 + \frac{1}{n} \sum_{i=1}^n \left( \frac{1}{m} \sum_{j=i+1}^n 1 \right)$$

$$= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) = 1 + \frac{1}{nm} \left( \sum_{i=1}^n n - \sum_{i=1}^n i \right)$$

$$= 1 + \frac{1}{nm} \left( n^2 - \frac{n(n+1)}{2} \right) = 1 + \frac{n}{2m} - \frac{1}{2m}$$

## CHAINING / CONCATENAZIONE

Sostituiamo quindi  $\alpha = n/m$  ovunque ottenendo

$$1 + \frac{n}{2m} - \frac{1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1 + \alpha)$$

Sommato al tempo di calcolo per la funzione di hash (che è costante), otteniamo  $\Theta(1 + \alpha)$ .

Questo mostra che finché  $n = O(m)$ , abbiamo

$\alpha = \frac{O(m)}{m} = O(1)$  e quindi il tempo medio per la ricerca è  $O(1)$ .

## CHAINING / CONCATENAZIONE: DISCUSSIONE

- ▶ Questi risultati valgono sotto le assunzioni di avere un "buona" funzione di hash che distribuisce in modo uniforme le chiavi
- ▶ Serve inoltre che  $n = O(m)$ , quindi se la tabella su riempie troppo servirà sostituirla con una più grande (e.g., raddoppiando il numero di slot) e reinserire tutti i valori contenuti nella tabella vecchia.

# FUNZIONI DI HASH

- ▶ Fino ad ora non abbiamo detto quali funzioni di hash sono considerate buone.
- ▶ Una buona funzione di hash dovrebbe rispettare la proprietà di distribuire le chiavi uniformemente negli  $m$  slot a disposizione...
- ▶ ... ma solitamente non sappiamo con che distribuzione di probabilità sono prese le chiavi che inseriamo

# FUNZIONI DI HASH

- ▶ Se abbiamo buona conoscenza della distribuzione con cui sono ottenute le chiavi possiamo costruire una funzione di hash ad-hoc
- ▶ In generale utilizziamo alcune euristiche che funzionano bene in pratica e in cui vediamo le chiavi come numeri naturali:
  - ▶ Il metodo della divisione
  - ▶ Il metodo della moltiplicazione

## METODO DELLA DIVISIONE

- ▶ Data una tabella di dimensione  $m$ , per ogni chiave definiamo  $h(k)$  come  $h(k) = k \bmod m$
- ▶ Generalmente un metodo di hashing rapido
- ▶ Vogliamo però evitare alcuni valori di  $m$  che possono essere problematici.
- ▶ Vediamo alcuni di questi casi

## METODO DELLA DIVISIONE

- ▶ Se  $m = 2^p$  per qualche  $p \in \mathbb{N}$ , il valore di  $h(x)$  dipende solo dai  $p$  bit meno significativi di  $x$
- ▶ Si vede bene in base 10: se  $m = 100$  abbiamo che  $h(x)$  dipende solo dalle ultime due cifre di  $x$ , quindi 8298, 43298, 198 hanno tutti lo stesso hash
- ▶ Generalmente buone scelte per  $m$  sono primi vicini a potenze di 2. Questo permette di avere un valore di hash che dipende da tutti i bit della chiave

## METODO DELLA MOLTIPLICAZIONE

- ▶ Creazione di una funzione di hash in due passi
- ▶ Si sceglie una costante  $A$  con  $0 < A < 1$
- ▶ Si moltiplica la chiave  $x$  per  $A$  e si prende la parte frazionaria:  $Ax - \lfloor Ax \rfloor$
- ▶ La parte frazionaria si moltiplica per  $m$  e del risultato si prende la parte intera:  
$$h(x) = \lfloor m(Ax - \lfloor Ax \rfloor) \rfloor$$

## METODO DELLA MOLTIPLICAZIONE

- ▶ Il vantaggio di questo metodo è che il valore di  $m$  non è critico, quindi possiamo scegliere una potenza di 2.
- ▶ La parte critica è invece il valore di  $A$ , ma questo non deve cambiare se dobbiamo ingrandire la tabella.
- ▶ Knuth suggerisce un valore  $A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887$