

TABELLE HASH

---

# ALGORITMI E STRUTTURE DATI

# INDIRIZZAMENTO APERTO

- ▶ Utilizzando il chaining andavamo ad utilizzare spazio al di fuori di quello dell'array
- ▶ Con l'indirizzamento aperto vogliamo invece tenere tutto all'interno dell'array
- ▶ Al contrario del chaining è quindi richiesto che  $m \geq n$ , dato che abbiamo solo  $m$  posti in cui inserire i valori

## INDIRIZZAMENTO APERTO

- ▶ Chiaramente, dobbiamo applicare una nuova strategia per risolvere le collisioni
- ▶ La strategia di base è quella di avere una sequenza di posizioni da provare ed inserire nella prima che si trova libera
- ▶ Vogliamo inoltre che se esiste un posto libero questo sia nella sequenza di posizioni da trovare

## INDIRIZZAMENTO APERTO

- ▶ Estendiamo la funzione di hashing con il concetto di **probe function**, ovvero  $h : U \times \{0, \dots, m - 1\} \rightarrow \{0, \dots, m - 1\}$  dove  $U$  è l'insieme delle possibili chiavi
- ▶  $h(k, 0)$  indicherà la prima posizione in cui provare a inserire  $k$ ,  $h(k, 1)$  la seconda posizione, etc.
- ▶ Se  $h(k, 0), h(k, 1), \dots, h(k, m - 1)$  è una permutazione di  $0, 1, \dots, m - 1$  allora potenzialmente se esiste un posto libero lo troveremo (visitiamo tutte le posizioni dell'array)

# INDIRIZZAMENTO APERTO

- ▶ Mentre l'inserimento è relativamente facile da definire dobbiamo stare attenti a definire la ricerca e, soprattutto, la cancellazione
- ▶ Per la ricerca, data la chiave  $k$ , non ci dobbiamo fermare a  $h(k,0)$ , ma continuare finché non troviamo  $k$  o una posizione vuota

# INSERIMENTO (OPEN ADDRESSING)

### Inserimento

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

```
 $i = 0$ 
```

```
pos =  $h(x.key, i)$ 
```

```
while  $T[pos]$  is not None and  $i < m$ :
```

```
    # iteriamo fino a quando non troviamo un posto libero
```

```
     $i = i + 1$ 
```

```
    pos =  $h(x.key, i)$ 
```

```
if  $i == m$ : # se non c'è un posto dopo  $m$  iterazioni la tabella è piena
```

```
    Errore: Tabella piena
```

```
else:
```

```
     $T[pos] = x$ 
```

# RICERCA (OPEN ADDRESSING)

### Ricerca

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

```
i = 0
pos = h(k, i)
while T[pos] is not None and i < m and T[pos].key != k:
    # iteriamo fino a quando non troviamo un posto libero o non troviamo k
    i = i + 1
    pos = h(x.key, i)
if i == m or T[pos] is none: # non abbiamo trovato l'elemento
    return None
else:
    return T[pos]
```

# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

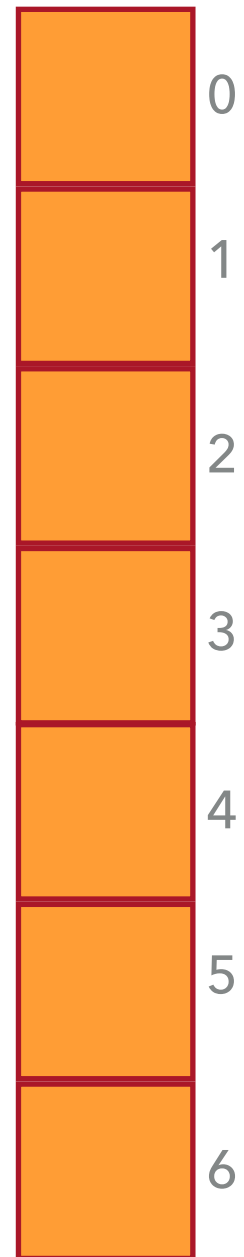
Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

**Funzione  
di hash**





# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

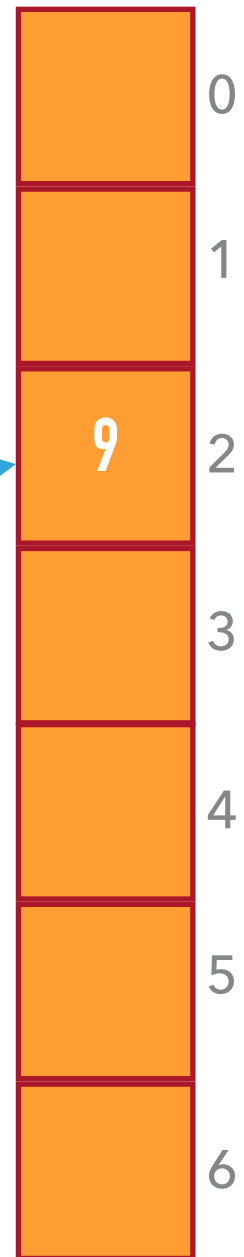
Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

**Funzione  
di hash**

$$h(9, 0) = 2$$



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

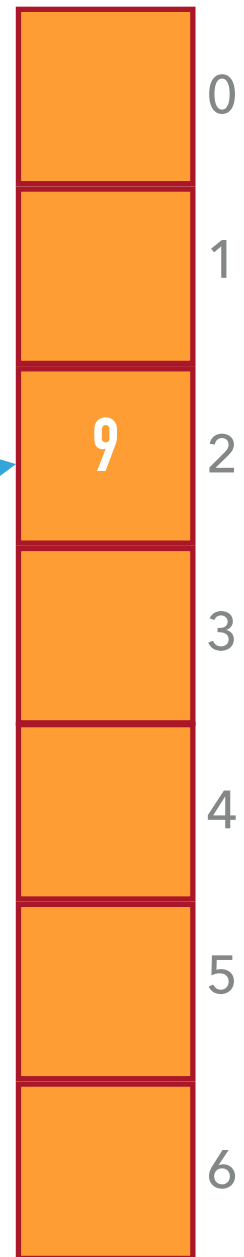
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione di hash**

$$h(23,0) = 2$$

Non possiamo inserirlo perché la posizione è già occupata



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

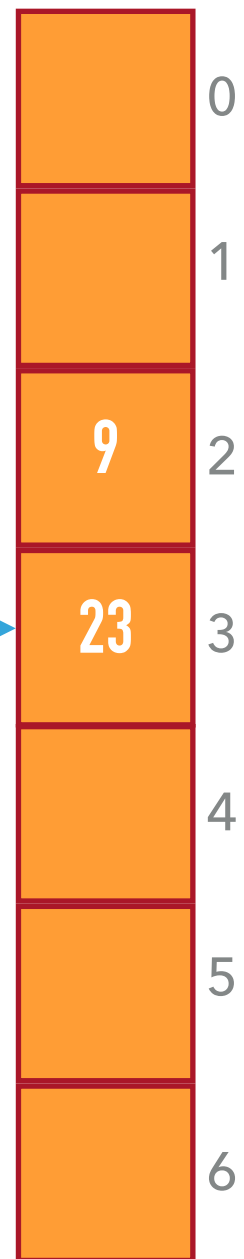
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione  
di hash**

$$h(23, 1) = 3$$

L'inserimento va a buon fine



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

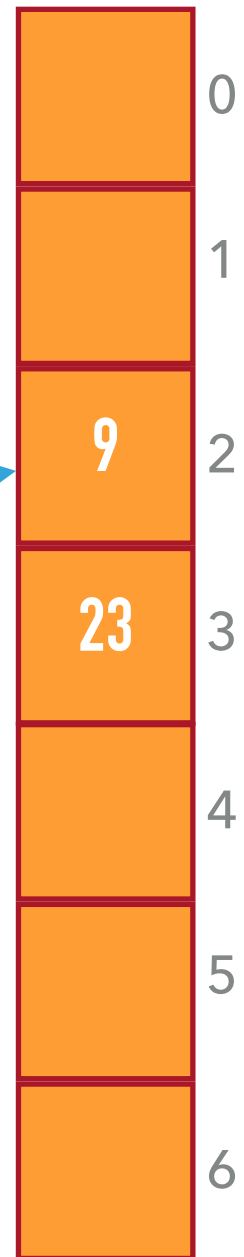
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione di hash**

$$h(16,0) = 2$$

Non possiamo inserirlo perché la posizione è già occupata



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

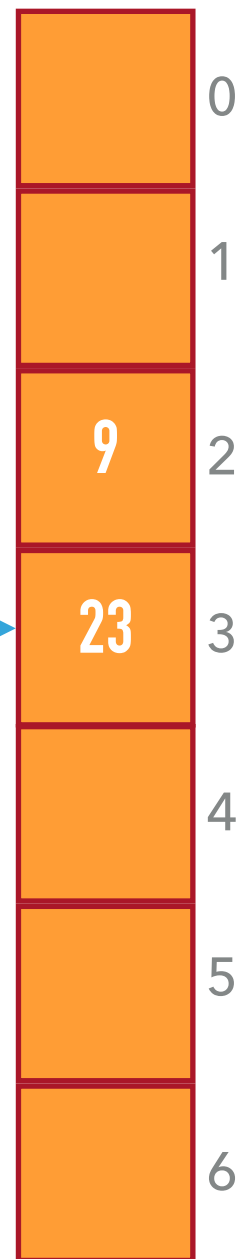
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione  
di hash**

$$h(16,1) = 3$$

Non possiamo inserirlo  
perché la posizione è  
già occupata



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione di hash**

$$h(16, 2) = 4$$

L'inserimento va a buon fine

	0
	1
9	2
23	3
16	4
	5
	6

# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

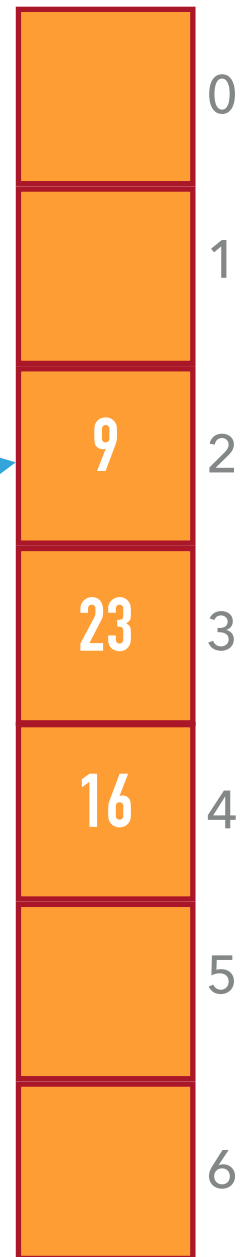
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione di hash**

$$h(23,0) = 2$$

Non cancelliamo perché la chiave salvata in posizione 2 non è 23



## HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

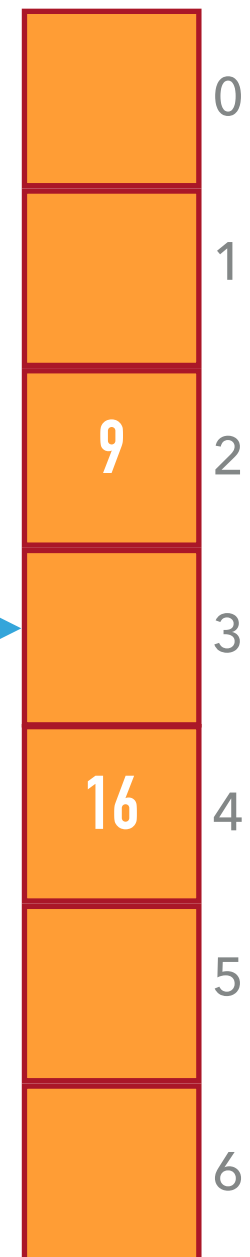
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$



$$h(23, 1) = 3$$

Questa volta cancelliamo perché la chiave in posizione 3 è 23.



Attenzione, è corretto cancellare semplicemente il contenuto in posizione 3?

NO! Se cercassimo "16" incontreremmo un posto vuoto e ci fermeremmo erroneamente prima di aver completato la ricerca



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

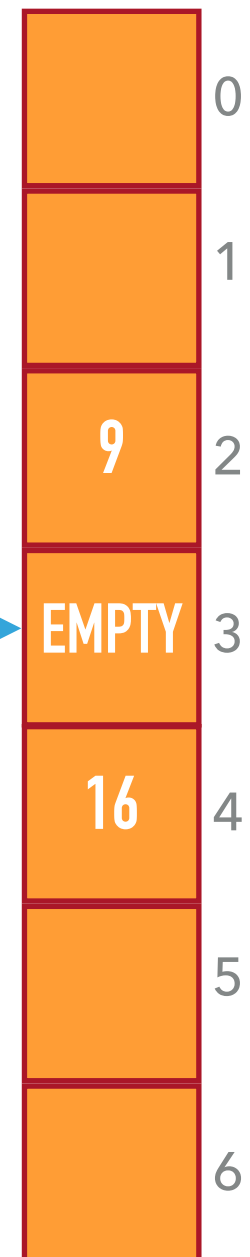
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione di hash**

$$h(23, 1) = 3$$

Inseriamo un valore "segnaposto"  
Indica che la casella è libera ma  
conteneva un elemento cancellato



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione  
di hash**

$$h(16,0) = 2$$

Non trovato, proseguiamo

	0
	1
9	2
EMPTY	3
16	4
	5
	6

# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

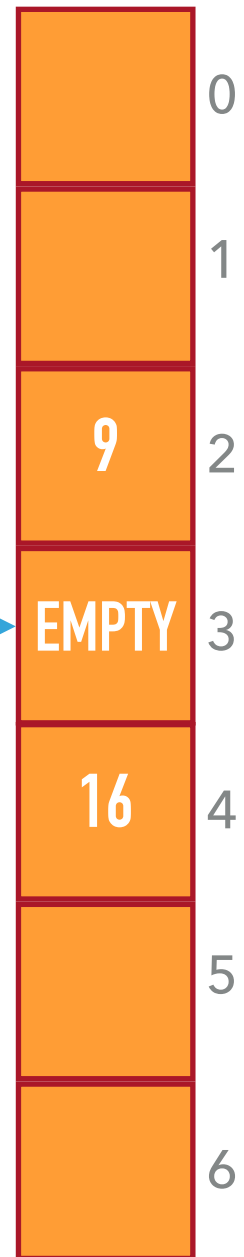
Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

**Funzione  
di hash**

$$h(16,1) = 3$$

Non trovato, proseguiamo  
anche se la casella è vuota  
perché è lo speciale valore  
segnaposto



# HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \text{ mod } 7$$

Funzione di hash

$$h(16, 2) = 4$$

Trovato!

	0
	1
9	2
EMPTY	3
16	4
	5
	6

# CANCELLAZIONE (OPEN ADDRESSING)

### Cancellazione

Parametri:  $x$  (oggetto da cancellare) e la tabella  $T$

```
 $i = 0$ 
```

```
pos =  $h(k, i)$ 
```

```
while  $T[pos] \neq x$ :
```

```
    # iteriamo fino a quando non troviamo  $x$  (assumiamo  $x$  contenuto in  $T$ )
```

```
     $i = i + 1$ 
```

```
    pos =  $h(x.key, i)$ 
```

```
 $T[pos] = \text{EMPTY}$  # valore segnaposto
```

Attenzione, dobbiamo modificare la procedura di inserimento per inserire nelle caselle "EMPTY"

# INSERIMENTO (OPEN ADDRESSING) – CON CANCELLAZIONE

### Inserimento

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

```
 $i = 0$ 
```

```
pos = h(x.key, i)
```

```
while T[pos] is not None and  $i < m$  and T[pos] is not EMPTY:
```

```
    # iteriamo fino a quando non troviamo un posto libero
```

```
     $i = i + 1$ 
```

```
    pos = h(x.key, i)
```

```
if  $i == m$ : # se non c'è un posto dopo  $m$  iterazioni la tabella è piena
```

```
    Errore: Tabella piena
```

```
else:
```

```
    T[pos] = x
```

# ALCUNE NOTE SULLA CANCELLAZIONE

- ▶ Se non cancelliamo otteniamo dei buoni bound sul tempo necessario alla ricerca che dipenderanno dal fattore di carico  $\alpha$
- ▶ Se consentiamo la cancellazione, invece la questione è molto più complessa:
  - ▶ Immaginate di riempire la tabella e poi svuotarla completamente
  - ▶ Ora ogni ricerca deve comunque passare per tutte le posizioni (che sono EMPTY e non None) prima di fallire

# ALCUNE NOTE SULLA CANCELLAZIONE

- ▶ A causa di questi problemi il chaining viene di solito scelto se è necessario cancellare.
- ▶ Rimangono comunque alcune alternative: delle operazioni di "pulizia" tramite re-inserimento in una tabella nuova, "spostare" gli elementi invece marcare come EMPTY, etc.
- ▶ Noi ora proseguiremo assumendo di non avere la cancellazione ma solo inserimenti e ricerche (quindi niente valori segnaposto EMPTY)



# TIPOLOGIE DI PROBING

- ▶ Ci sono diverse strategie per effettuare il probing, noi ne vediamo tre diversi tipi:
  - ▶ Linear probing / Probing lineare
  - ▶ Quadratic probing / Probing quadratico
  - ▶ Double hashing / Doppio hashing
- ▶ Vediamo per ognuna vantaggi e svantaggi

# LINEAR PROBING

- ▶ Data una funzione di hash  $h' : U \rightarrow \{0, \dots, m - 1\}$  la modifichiamo definendo una funzione  $h$  come:  
$$h(x, i) = (h'(x) + i) \bmod m$$
- ▶ In pratica significa che iniziamo a testare da  $h'(x)$ , poi  $h'(x) + 1$ , etc. incrementando di uno alla volta ed eventualmente ricominciando da zero quando raggiungiamo il valore  $m$

# HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash  $h'(k)$

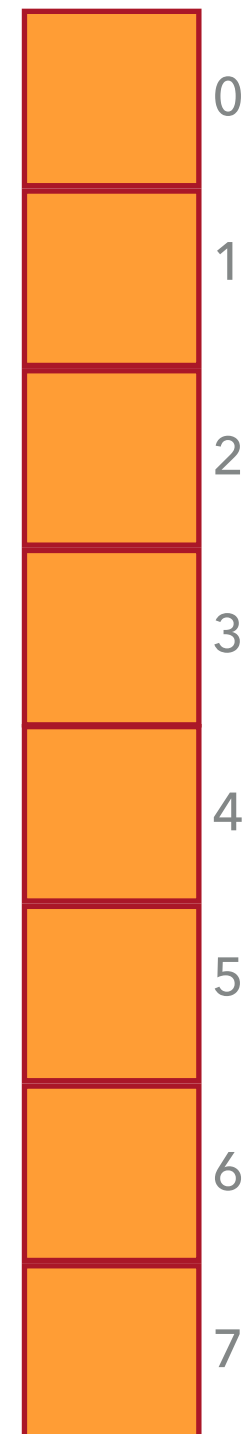
Inseriamo 33

Inseriamo 47

Inseriamo 12



$$h(k) = (h'(k) + i) \bmod m$$



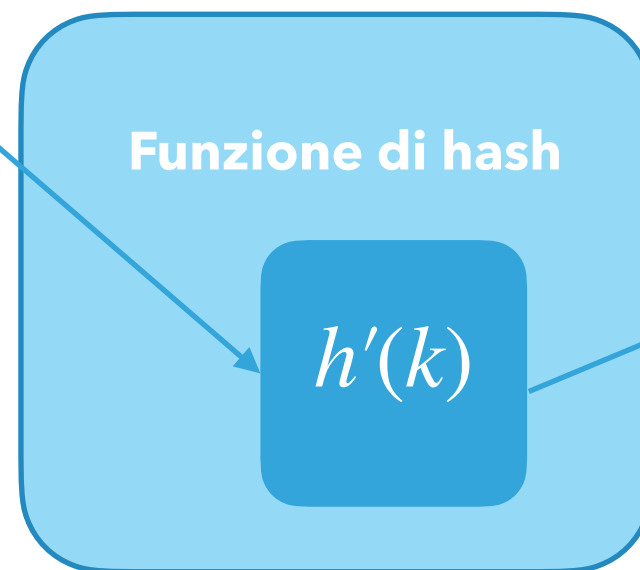
# HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash  $h'(k)$

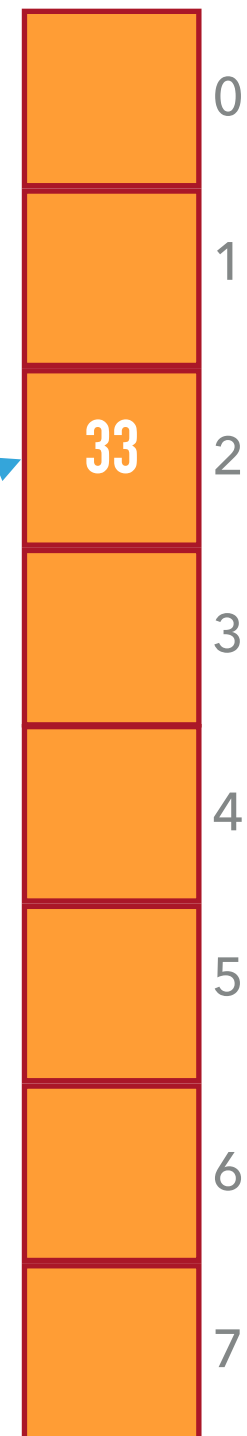
Inseriamo 33

Inseriamo 47

Inseriamo 12



$$h(k) = (h'(k) + i) \bmod m$$



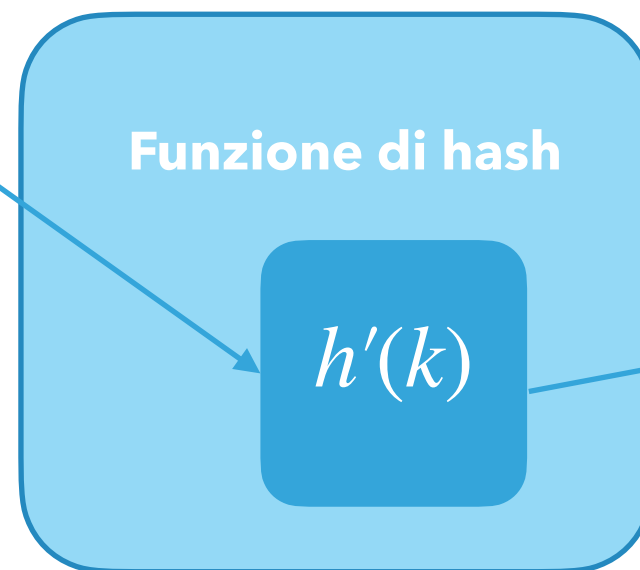
# HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash  $h'(k)$

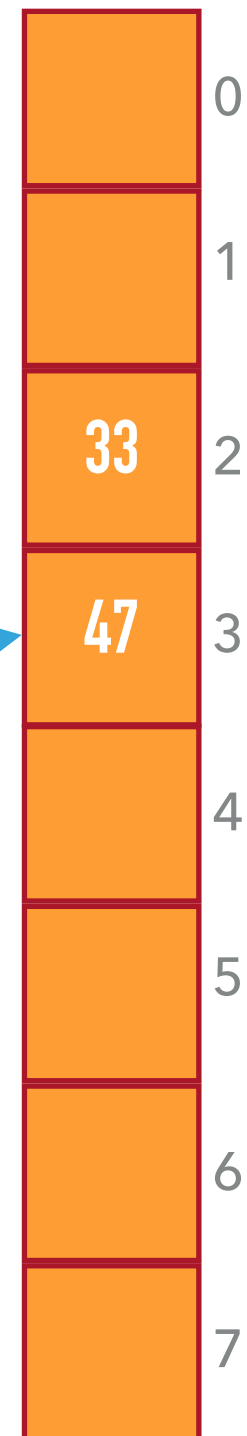
Inseriamo 33

Inseriamo 47

Inseriamo 12



$$h(k) = (h'(k) + i) \bmod m$$



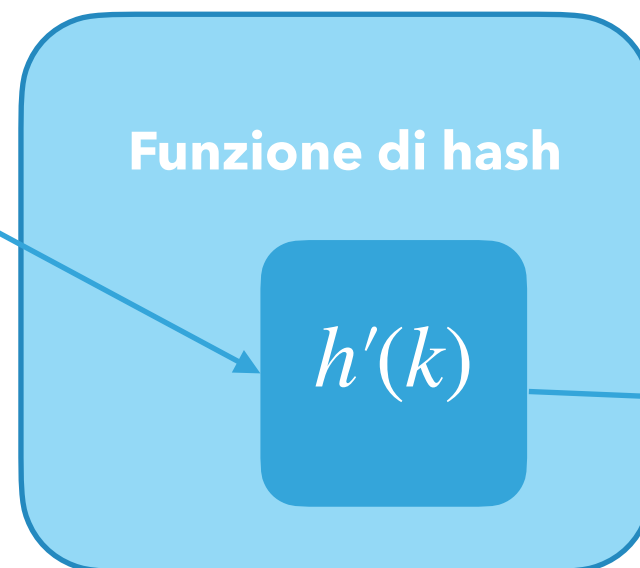
# HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash  $h'(k)$

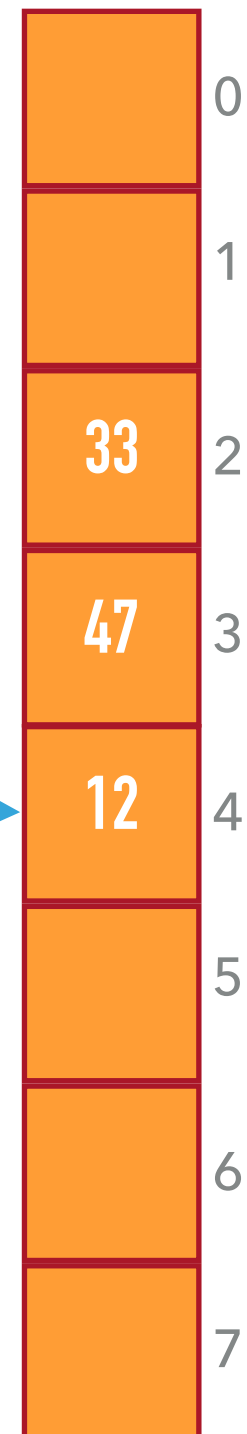
Inseriamo 33

Inseriamo 47

Inseriamo 12



$$h(k) = (h'(k) + i) \bmod m$$



## HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash  $h'(k)$

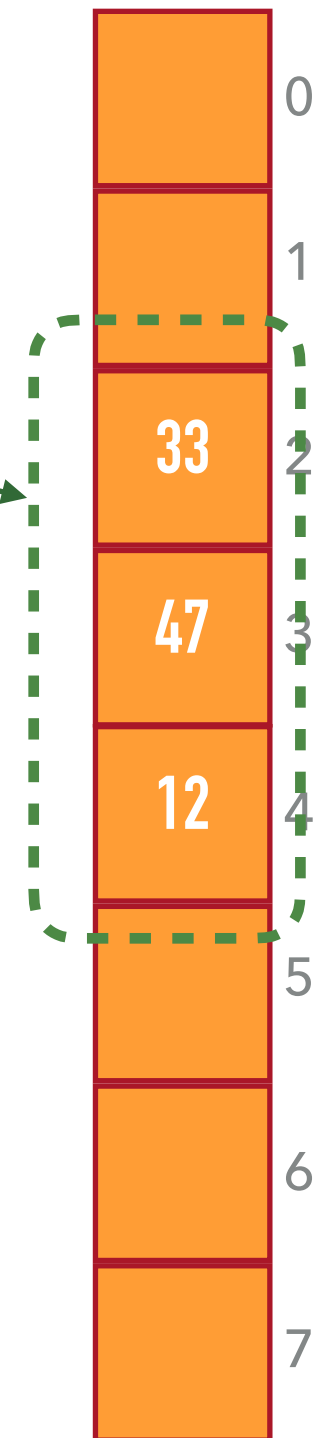
Otteniamo un raggruppamento di valori consecutivi

Questo si chiama "primary clustering" o "clustering primario"

Perché è un problema?

La posizione 5 ora raccoglie tutti i valori con hash 2, 3, 4 e 5

In generale una posizione preceduta da  $i$  posizioni occupate sarà scelta per inserire il prossimo valore con probabilità  $\frac{i+1}{m}$



## QUADRATIC PROBING

- ▶ Data una funzione di hash  $h' : U \rightarrow \{0, \dots, m - 1\}$  la modifichiamo definendo una funzione  $h$  come:  
$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$
- ▶ Dobbiamo scegliere attentamente le costanti  $c_1$  e  $c_2$  in modo di visitare tutte le posizioni
- ▶ Alcuni esempi: se  $m$  è una potenza di 2 possiamo scegliere  $c_1 = c_2 = 1/2$ , questo genererà la sequenza  
 $h'(k), h'(k) + 1, h'(k) + 3, h'(k) + 6, \dots$



## QUADRATIC PROBING

- ▶ Non otteniamo lo stesso problema del clustering primario come con il probing lineare
- ▶ Otteniamo comunque una forma meno forte di clustering chiamato secondary clustering o clustering secondario
- ▶ Perché entrambe le strategie di probing provocano problemi (sebbene diversi) col clustering?

# I PROBLEMI DI QUADRATIC E LINEAR PROBING

- ▶ Sia quadratic e linear probing hanno un problema in comune dovuto alla loro costruzione
- ▶ Dati  $k_1$  e  $k_2$  con  $h'(k_1) = h'(k_2)$  abbiamo che per ogni  $i$  vale  $h(k_1, i) = h(k_2, i)$ , ovvero hanno uguale sequenza di probing
- ▶ In pratica due chiavi con uguale hash  $h'(k)$  testano le stesse posizioni: abbiamo solo  $m$  sequenze di probing distinte

## DOUBLE HASHING

- ▶ Date **due** funzioni di hash  $h_1 : U \rightarrow \{0, \dots, m - 1\}$  e  $h_2 : U \rightarrow \{0, \dots, m - 1\}$  definiamo 
$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$
- ▶ Per poter visitare tutte le posizioni  $h_2(k)$  deve essere coprimo rispetto a  $m$ . Possiamo assicurarci in più modi:
  - ▶  $h_2(k)$  è sempre dispari e  $m$  una potenza di 2
  - ▶  $m$  primo e  $h_2(k)$  ritorna solo valori in positivi minori di  $m$

# DOUBLE HASHING

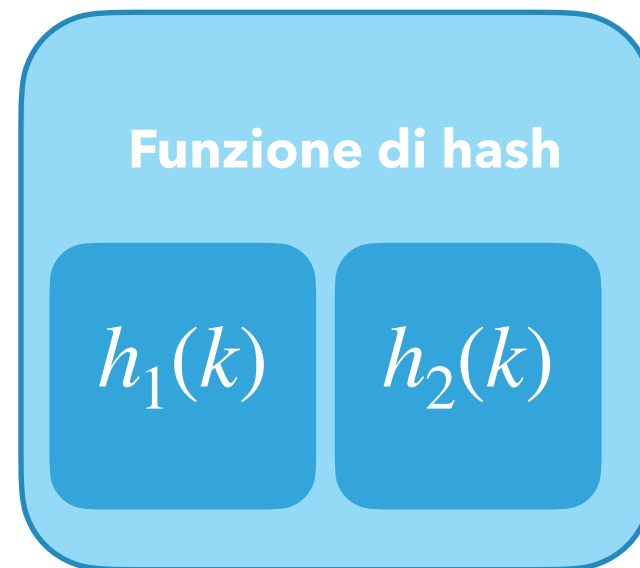
- ▶ Il vantaggio di utilizzare due funzioni di hash è che se anche abbiamo  $k_1 \neq k_2$  con  $h_1(k_1) = h_1(k_2)$  non è detto che  $k_1$  e  $k_2$  abbiano la stessa sequenza di probing
- ▶ Due chiavi  $k_1$  e  $k_2$  hanno la stessa sequenza di probing solo quando hanno lo stesso hash sia con  $h_1$  che con  $h_2$
- ▶ Abbiamo quindi  $m^2$  diverse sequenze di probing (una per ogni coppia di valori  $(h_1(k), h_2(k))$  possibile)

# HASH CON DOUBLE HASHING

Inseriamo 9

Inseriamo 23

Inseriamo 16



$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$

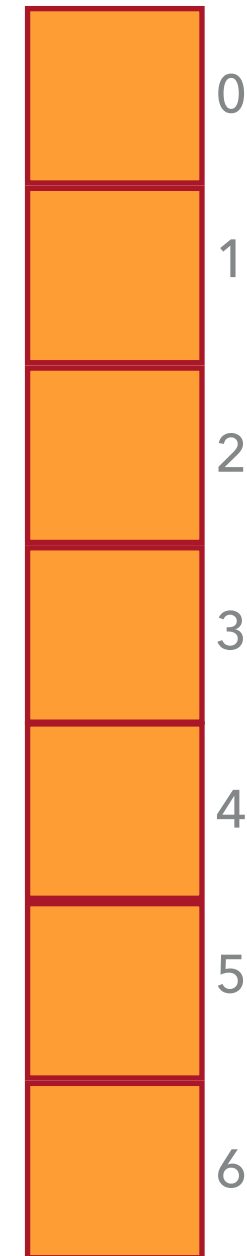
$$h_1(k) = k \bmod 7$$

$$h_2(k) = 1 + (k \bmod 6)$$

$h_2$  ritorna solo valori positivi minori di  $m$ .

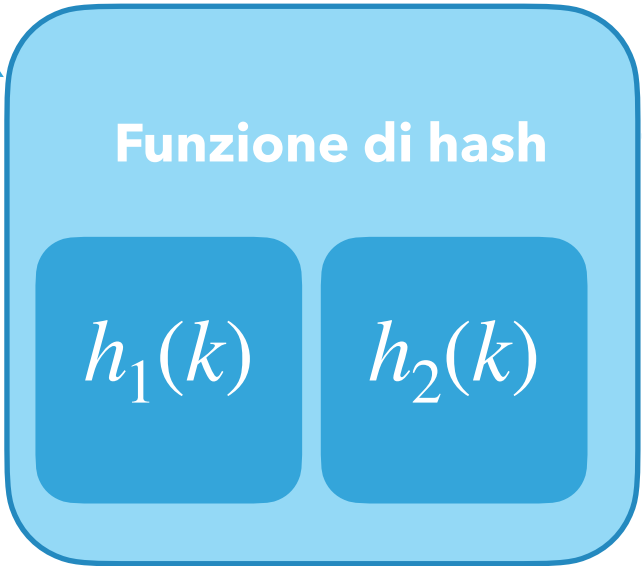
In generale funzione prendendo

$$h_2(k) = 1 + k \bmod (m - 1)$$



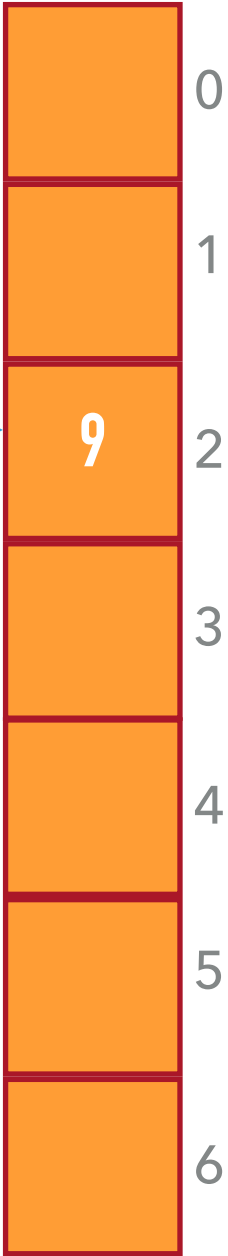
# HASH CON DOUBLE HASHING

Inseriamo 9  
Inseriamo 23  
Inseriamo 16



$$h(k) = (2 + 0 \times 4) \bmod 7 = 2$$

$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$
$$h_1(k) = k \bmod 7$$
$$h_2(k) = 1 + (k \bmod 6)$$



# HASH CON DOUBLE HASHING

Inseriamo 9  
Inseriamo 23  
Inseriamo 16



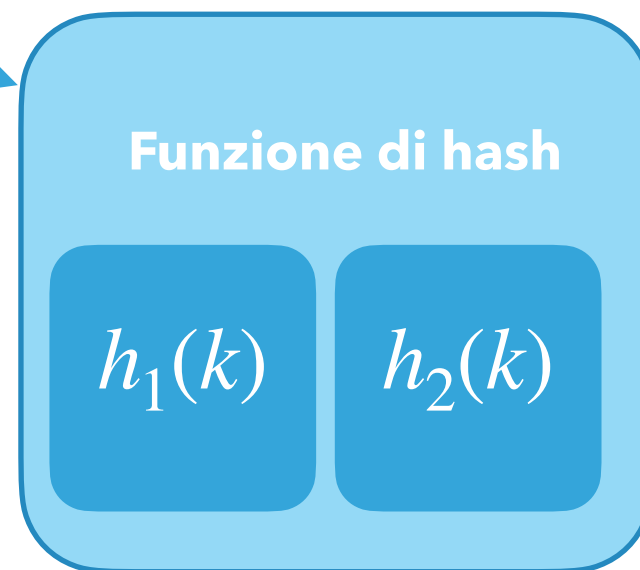
$$h(k) = (2 + 0 \times 6) \bmod 7 = 2$$

$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$
$$h_1(k) = k \bmod 7$$
$$h_2(k) = 1 + (k \bmod 6)$$

	0
	1
9	2
	3
	4
	5
	6

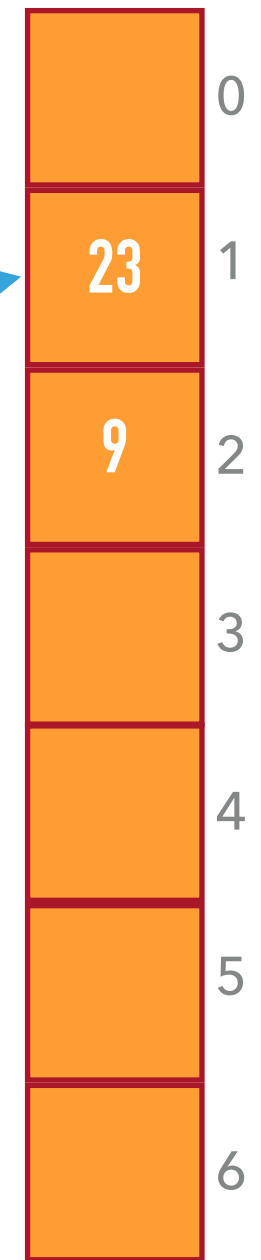
# HASH CON DOUBLE HASHING

Inseriamo 9  
Inseriamo 23  
Inseriamo 16



$$h(k) = (2 + 1 \times 6) \bmod 7 = 1$$

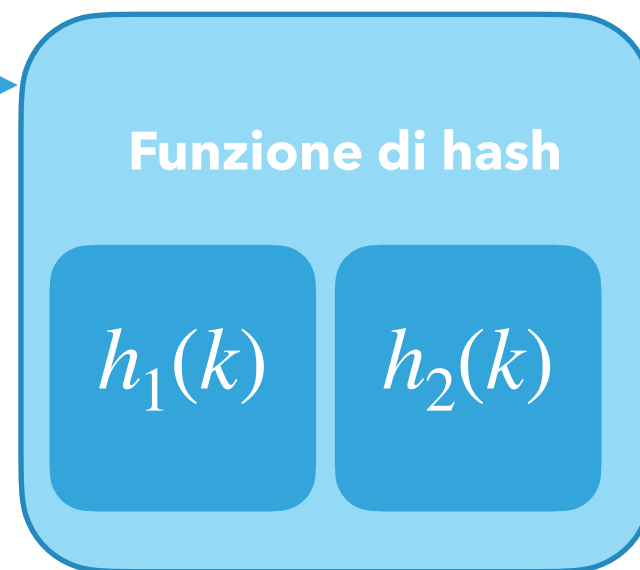
$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$
$$h_1(k) = k \bmod 7$$
$$h_2(k) = 1 + (k \bmod 6)$$





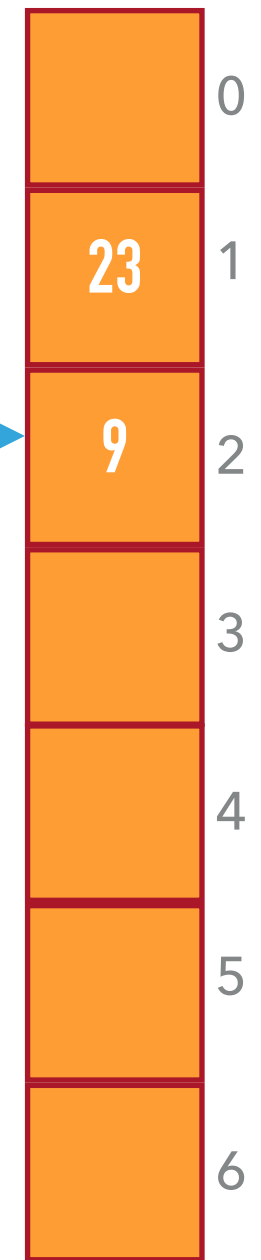
# HASH CON DOUBLE HASHING

Inseriamo 9  
Inseriamo 23  
Inseriamo 16



$$h(k) = (2 + 0 \times 5) \bmod 7 = 2$$

$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$
$$h_1(k) = k \bmod 7$$
$$h_2(k) = 1 + (k \bmod 6)$$



# HASH CON DOUBLE HASHING

Inseriamo 9

Inseriamo 23

Inseriamo 16



$$h(k) = (2 + 1 \times 5) \bmod 7 = 0$$

$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$
$$h_1(k) = k \bmod 7$$
$$h_2(k) = 1 + (k \bmod 6)$$

16	0
23	1
9	2
	3
	4
	5
	6

# ANALISI DEL TEMPO

- ▶ Dobbiamo ora studiare il tempo che ci impieghiamo a inserire e cercare un elemento all'interno di una tabella hash
- ▶ Assumiamo uniform hashing (o hashing uniforme):
  - ▶ La sequenza di probing di ogni chiave è con uguale probabilità una qualsiasi delle  $m!$  permutazioni di  $\{0, 1, \dots, m - 1\}$
- ▶ Il vero hashing uniforme è difficile da implementare, di solito ci accontentiamo di una sua approssimazione

# ANALISI DEL TEMPO

- ▶ Sotto l'assunzione di hashing uniforme
- ▶ Se assumiamo di mantenere il fattore di carico  $\alpha = n/m$  limitato da una costante minore di 1 (e.g.,  $\alpha \leq 0.5$ , tabella al più piena per metà)
- ▶ Allora sia inserimento che ricerca richiedono, in media, tempo  $O(1)$

## RICERCA CHE NON HA SUCCESSO

- ▶ Assumendo uniform hashing, fattore di carico  $\alpha < 1$ , la ricerca di una chiave  $k$  non presente richiede di esplorare in media al più  $\frac{1}{1 - \alpha}$  posizioni

Poiché la chiave  $k$  non è presente, tutte le posizioni visitate tranne l'ultima saranno occupate.

Indichiamo con  $A_1, A_2, \dots, A_m$  gli eventi "l' $i$ -esima posizione nella sequenza di probing era occupata"

## RICERCA CHE NON HA SUCCESSO

La probabilità di dover visitare *almeno*  $i$  posizioni è data da  $P(A_1, A_2, \dots, A_{i-1})$  ovvero la probabilità **congiunta** che le prime  $i - 1$  posizioni siano tutte occupate

Usando il fatto che  $P(A, B) = P(A | B)P(B)$  possiamo riscrivere come:

$$P(A_1, A_2, \dots, A_{i-1}) = P(A_1)P(A_2 | A_1) \dots P(A_{i-1} | A_1, \dots, A_{i-2})$$

Queste sono probabilità che possiamo stimare

# RICERCA CHE NON HA SUCCESSO

$P(A_1) = n/m$  ovvero il fattore di carico. Ovvero la probabilità di trovare uno slot libero su  $m$  dato che  $n$  sono occupati

$$P(A_j | A_1, \dots, A_{j-1}) = \frac{n - (j - 1)}{m - (j - 1)}$$

perché se abbiamo trovato  $j - 1$  posizioni occupate, abbiamo la probabilità di trovarne una occupata scegliendo tra le  $m - (j - 1)$  rimanenti di cui  $n - (j - 1)$  sono occupate.

In questo punto stiamo sfruttando l'assunzione di hashing uniforme per dire che scegliamo in modo uniforme tra tutte le posizioni rimanenti

## RICERCA CHE NON HA SUCCESSO

Osserviamo che  $\frac{n - (j - 1)}{m - (j - 1)} \leq \frac{n}{m}$  per poter mostrare che

$$\begin{aligned} P(A_1, \dots, A_{i-1}) &= \frac{n}{m} \frac{n-1}{m-1} \dots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} \end{aligned}$$

Quindi la probabilità di dover guardare almeno  $i$  posizioni non è più di  $\alpha^{i-1}$



## RICERCA CHE NON HA SUCCESSO

Possiamo esprimere il numero atteso di posizioni da visitare  $E[X]$  come la probabilità di dover visitare una, due, etc. posizioni:

$$E[X] \leq \sum_{i=1}^{+\infty} \alpha^{i-1} = \sum_{i=0}^{+\infty} \alpha^i = \frac{1}{1-\alpha}$$

Questo ci mostra che una ricerca senza successo e anche un inserimento (che richiede di trovare la prima posizione libera) richiedono in media di visitare al più  $\frac{1}{1-\alpha}$  posizioni

### RICERCA CHE HA SUCCESSO

- ▶ Assumendo uniform hashing, fattore di carico  $\alpha < 1$  e che tutte le chiavi presenti abbiano uguale probabilità di essere cercate, la ricerca di una chiave  $k$  presente richiede di esplorare in media  $\frac{1}{\alpha} \log_e \frac{1}{1-\alpha}$  posizioni

La ricerca di una chiave  $k$  richiede di esplorare un numero di posizioni che dipende da quante chiavi sono state inserite in precedenza.

## RICERCA CHE HA SUCCESSO

Se  $k$  è stata la  $(i + 1)$ -esima chiave inserita, sfruttiamo il risultato precedente per dire che in media abbiamo visitato al più  $\frac{1}{1 - (i/m)} = \frac{1}{(m - i)/m} = \frac{m}{m - i}$  posizioni per inserirla

Facciamo la media su tutti i possibili valori di  $i$ , ovvero  $k$  potrebbe essere stata con uguale probabilità la prima, seconda, terza, etc. chiave inserita.

Questo ci fornirà un bound sul numero atteso di posizioni da cercare

## RICERCA CHE HA SUCCESSO

$$\frac{1}{n} \sum_{i=1}^n \frac{m}{m-i} = \frac{m}{n} \sum_{i=1}^n \frac{1}{m-i} = \frac{1}{\alpha} \sum_{i=1}^n \frac{1}{m-i}$$

Ora cambiamo l'indice della somma

$$\frac{1}{\alpha} \sum_{p=m-n+1}^m \frac{1}{p} \leq \frac{1}{\alpha} \int_{m-n}^m \frac{1}{p} dp$$

Calcoliamo il valore dell'integrale

$$\frac{1}{\alpha} \log_e \frac{m}{m-n} = \frac{1}{\alpha} \log_e \frac{(1/m)m}{(1/m)(m-n)} = \frac{1}{\alpha} \log_e \frac{1}{1-\alpha}$$

Mostrando che in media dobbiamo visitare  $\frac{1}{\alpha} \log_e \frac{1}{1-\alpha}$  posizioni