MODULO 2: Tabelle hash

Prof.ssa Giulia Cisotto

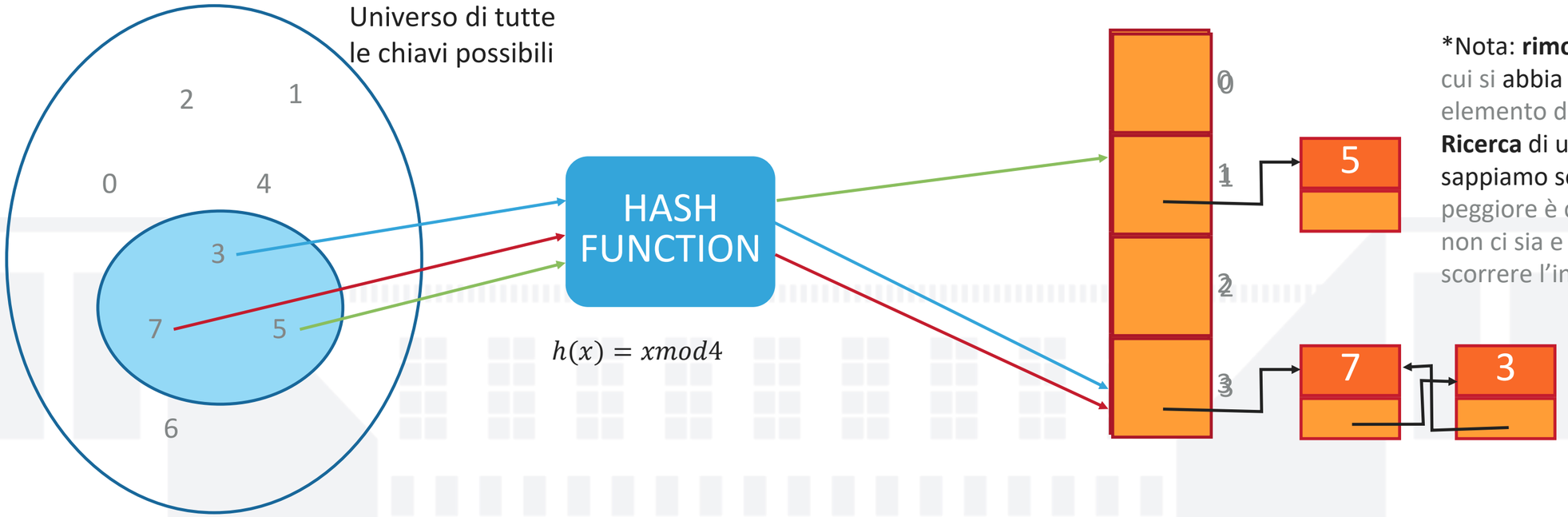
giulia.cisotto@units.it

Trieste, 8 maggio 2025

RECAP

n il numero di elementi contenuti nella tabella
m la dimensione della tabella (numero di *bucket*)

Dato un array, una **chiave k** ed una **funzione di hash h**, **la posizione in cui inserire/trovare k nell'array è $h(k)$.**



*Nota: **rimozione** nel caso in cui si abbia già puntatore ad elemento da rimuovere.
Ricerca di un elemento: non sappiamo se c'è, quindi il caso peggiore è che l'elemento non ci sia e dobbiamo scorrere l'intera lista.

Quando due chiavi distinte hanno lo stesso hash (ovvero $k_1 \neq k_2$ ma $h(k_1) = h(k_2)$), si genera una **collisione.**
Gestione delle collisioni:

1. **Chaining** o "**hash con concatenazione**"
2. *Indirizzamento aperto*

Inserimento in testa alla lista: O(1)
Rimozione* (con lista concatenata doppia): O(1)
Ricerca*: O(n)

$\alpha = n/m$ **load factor** o **fattore di carico** della tabella.

Stimiamo il **tempo medio** di ricerca.

UNIVERSITÀ DEGLI STUDI DI TRIESTE

TABELLE HASH: ALCUNE DEFINIZIONI E NOTE AGGIUNTIVE

- 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

La scelta della funzione di hash va fatta in modo opportuno per far sì che gli elementi vengano mappati in modo bilanciato le chiavi nei bucket. In questo caso, possiamo dimostrare che abbiamo prestazioni ottimali. Per mostrare le prestazioni che si potrebbero raggiungere nel caso in cui la mappatura fosse perfettamente uniforme, dobbiamo considerare una funzione di hash che associ casualmente ogni chiave ad uno slot.

Nella pratica non è possibile garantire la perfetta casualità del mapping. Inoltre, **noi abbiamo bisogno di una funzione deterministica** per poter fare anche le cancellazioni e la ricerca (se la funzione di hash fosse davvero casuale, non potremmo più rimuovere un elemento inserito in precedenza, perchè non lo troveremmo più!)

CHAINING / CONCATENAZIONE

Assumiamo che la funzione di hash distribuisca uniformemente le chiavi negli m slot.

DISCLAIMER: In questo caso, ipotizziamo che la funzione di hash NON sia più deterministica, ma agisca come se fosse una persona che, ad ogni chiave, lanciasse un dado ad *m* facce e decidesse così in quale bucket mappare la chiave. Se il dado non è truccato, la probabilità che una chiave finisca in un certo slot, tra gli m possibili, è 1/m (uguale per tutti gli slot e per ogni chiave).

Sotto queste assunzioni, mostriamo che **il <u>tempo medio</u> per cercare un elemento in una tabella hash è $\boldsymbol{\Theta(1+\alpha)}$**

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

CHAINING / CONCATENAZIONE

Se l'elemento cercato non è presente:

1. Calcolare l'hash della chiave → si ottiene l'indice di un bucket → **tempo O(1)**
2. Andare nel bucket corrispondente → contiene una **lista concatenata**
3. Scorrere tutta la lista per cercare la chiave → alla fine ci accorgiamo che **non c'è**

Ipotesi: la chiave k con cui lo cerchiamo ha uguale probabilità di finire in uno qualsiasi degli m slot.

Il tempo medio per **scoprire che la chiave non è presente** è dato dalla lunghezza media della lista di indice $h(k)$, ovvero il fattore di carico $\alpha = n/m$

Quindi il tempo medio richiesto per una ricerca senza successo è: $\Theta(1 + \alpha)$

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 $\boldsymbol{x_1, \ldots, x_n}$ **gli elementi inseriti** di chiavi $k_1, \ldots, k_n$

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

CHAINING / CONCATENAZIONE

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

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

ovvero uno più tutti gli elementi inseriti dopo di lui

Dato che $\boldsymbol{x}$ **potrebbe essere uno qualsiasi degli** $\boldsymbol{x_i}$,facciamo la media su tutte le possibili posizioni in cui potrebbe essere x, ottenendo il **seguente tempo medio**:

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

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

Nota: la probabilità di avere una collisione è 1/m*1/m. Ma la collisione può avvenire in una qualsiasi delle m celle della tabella. Quindi la probabilità di collisione nell'intera tabella è 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 $\boldsymbol{\Theta(1 + \alpha)}$**.**

CONCLUSIONE: Finché $\boldsymbol{n = O(m)}$, abbiamo $\alpha = \frac{O(m)}{m} = O(1)$ e **quindi il tempo medio per la ricerca è** $\boldsymbol{O(1)}$.

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 $\boldsymbol{n = O(m)}$, quindi se la tabella si riempie troppo servirà sostituirla con una più grande (e.g., raddoppiando il numero di slot) e reinserire tutti i valori contenuti nella tabella vecchia.

Materiale per la lezione

- Cormen et al. **CAP. 11**

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

Ci sono diverse strategie per effettuare il probing (lineare, quadratico, double hashing).

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 prove di **probe**, ovvero $h : U \times \{0,\ldots,m-1\} \rightarrow \{0,\ldots,m-1\}$ 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 $h(k,0), h(k,1), \ldots, h(k,m-1)$ è una permutazione di $0,1,\ldots,m-1$ 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

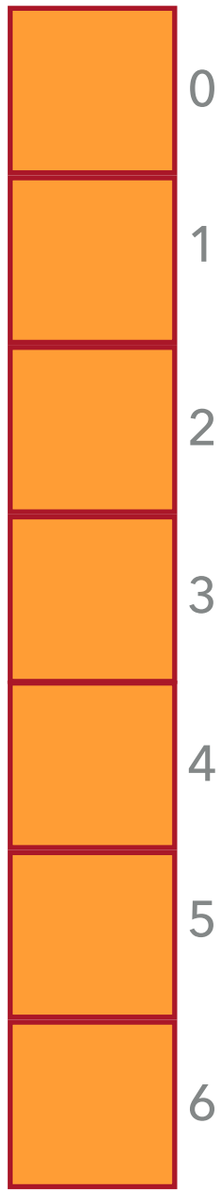
Inseriamo 16

Cancelliamo 23

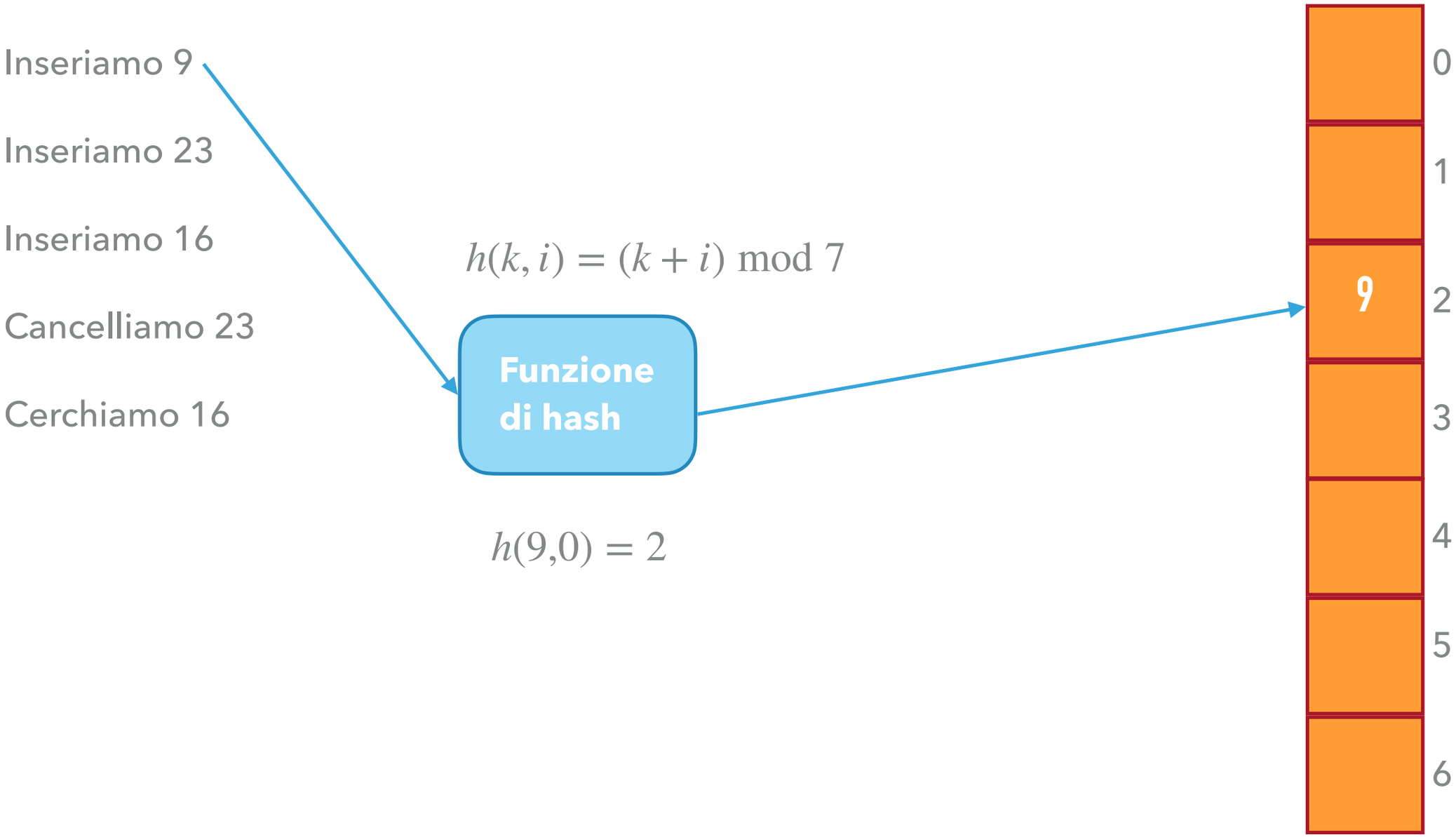
Cerchiamo 16

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

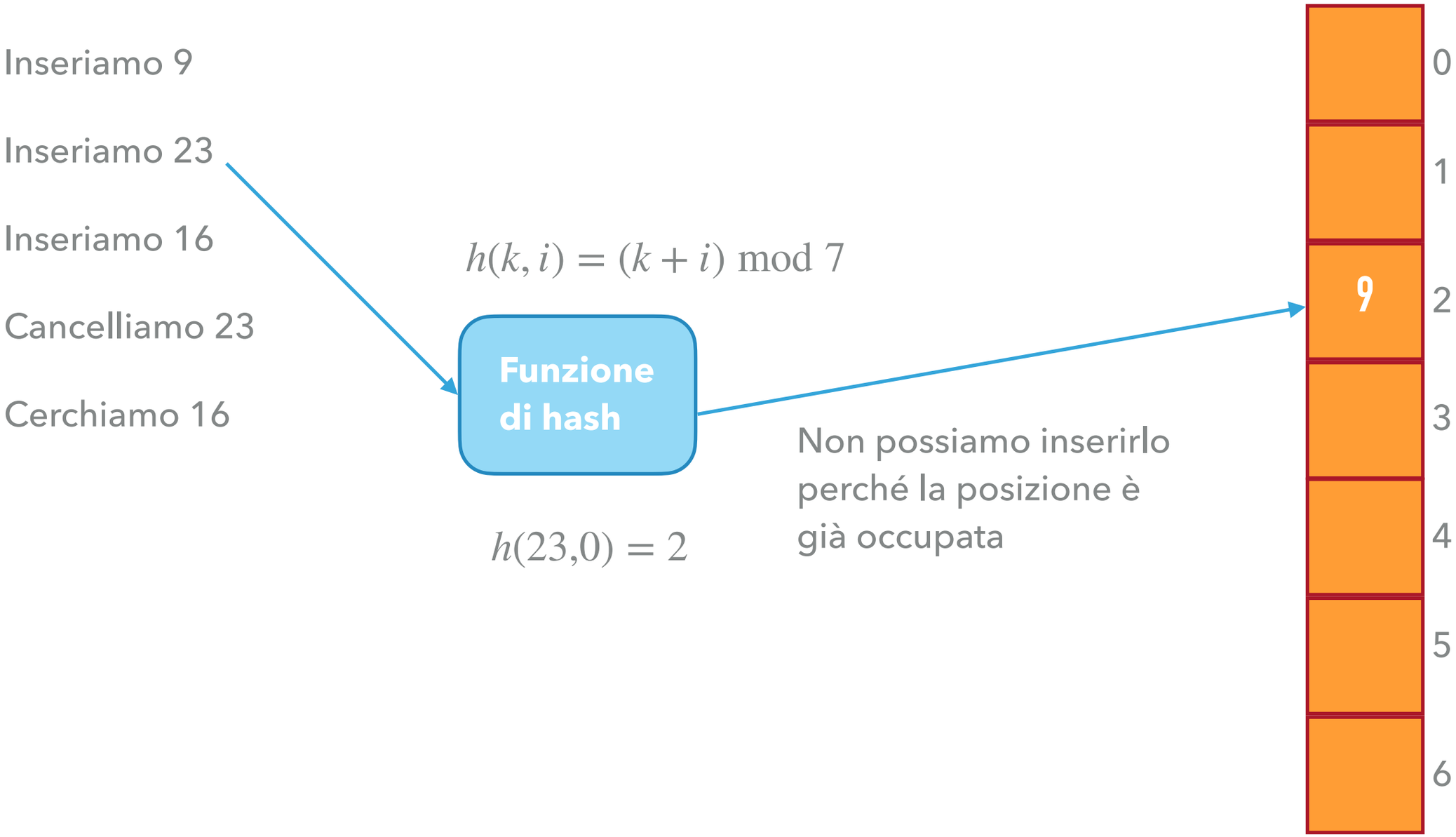
Funzione di hash



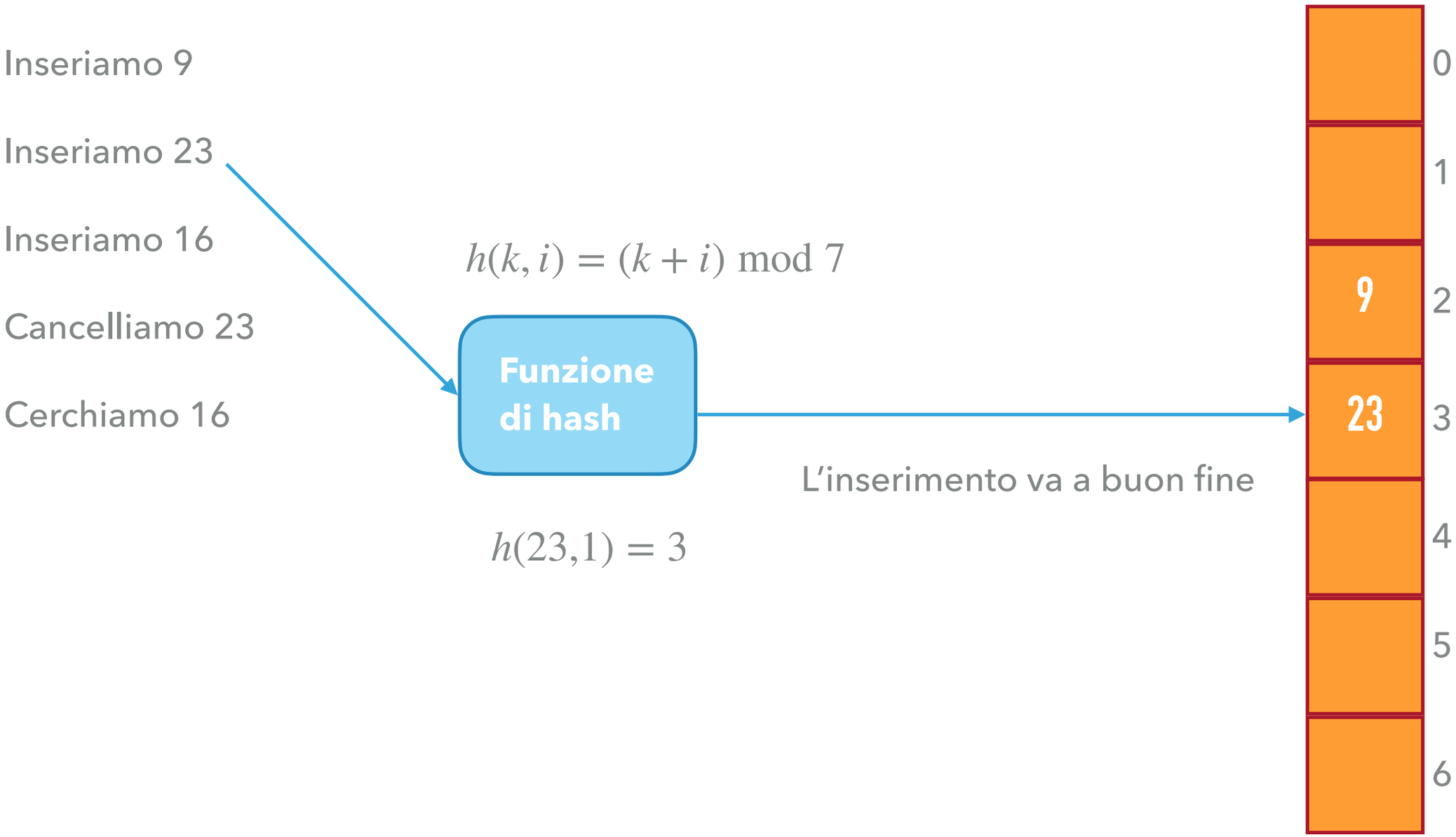
HASH: INSERIMENTO, RIMOZIONE E RICERCA



HASH: INSERIMENTO, RIMOZIONE E RICERCA



HASH: INSERIMENTO, RIMOZIONE E RICERCA



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

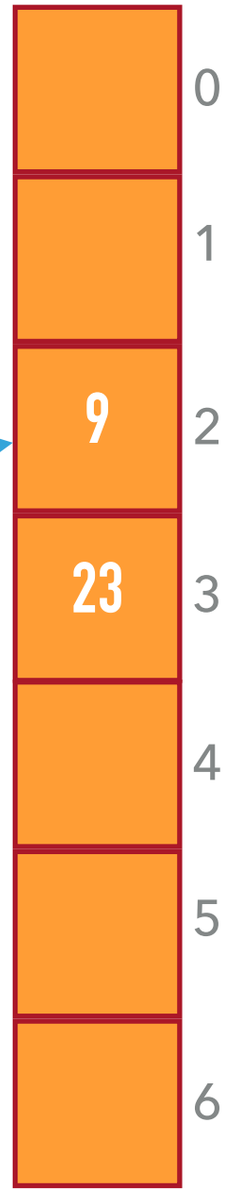
Cerchiamo 16

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

Funzione di hash

$h(16,0) = 2$

Non possiamo inserirlo perché la posizione è già occupata



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

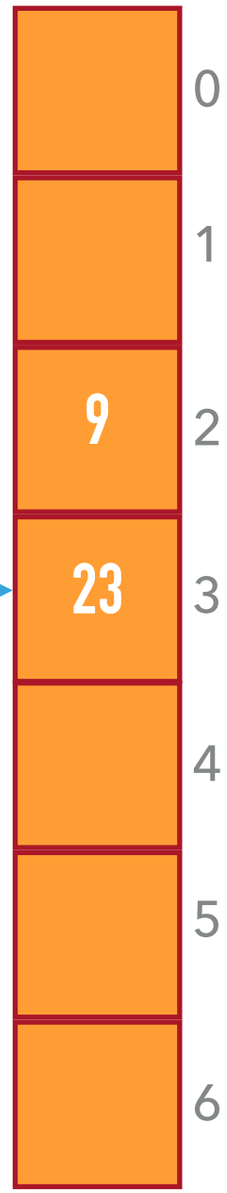
Cerchiamo 16

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

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

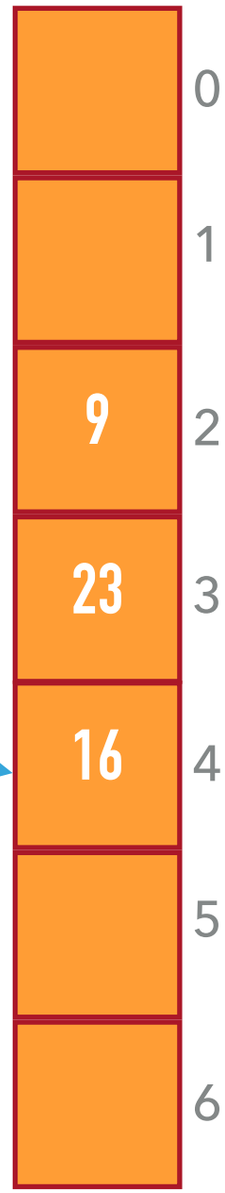
Cerchiamo 16

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

Funzione di hash

$h(16,2) = 4$

L'inserimento va a buon fine



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

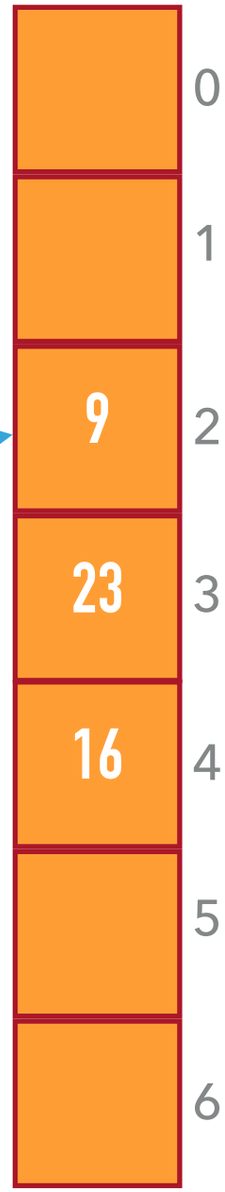
Cerchiamo 16

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

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

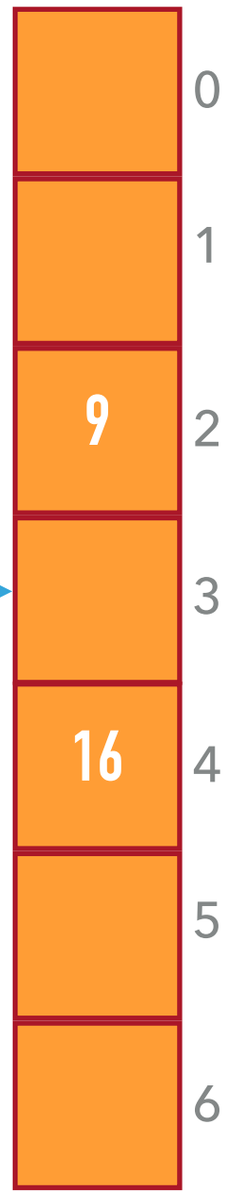
Cancelliamo 23

Cerchiamo 16

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

$h(23,1) = 3$

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

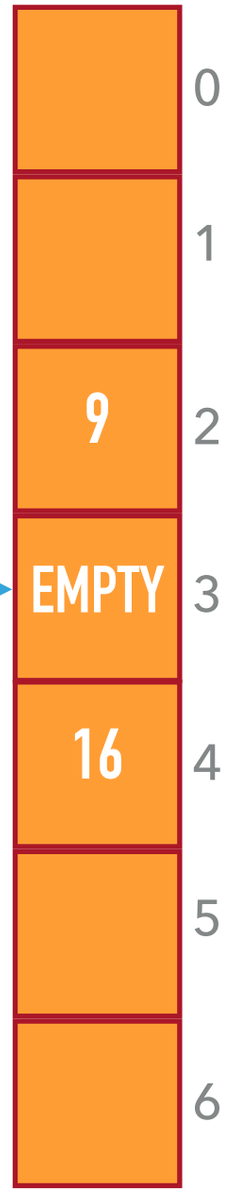
Cerchiamo 16

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

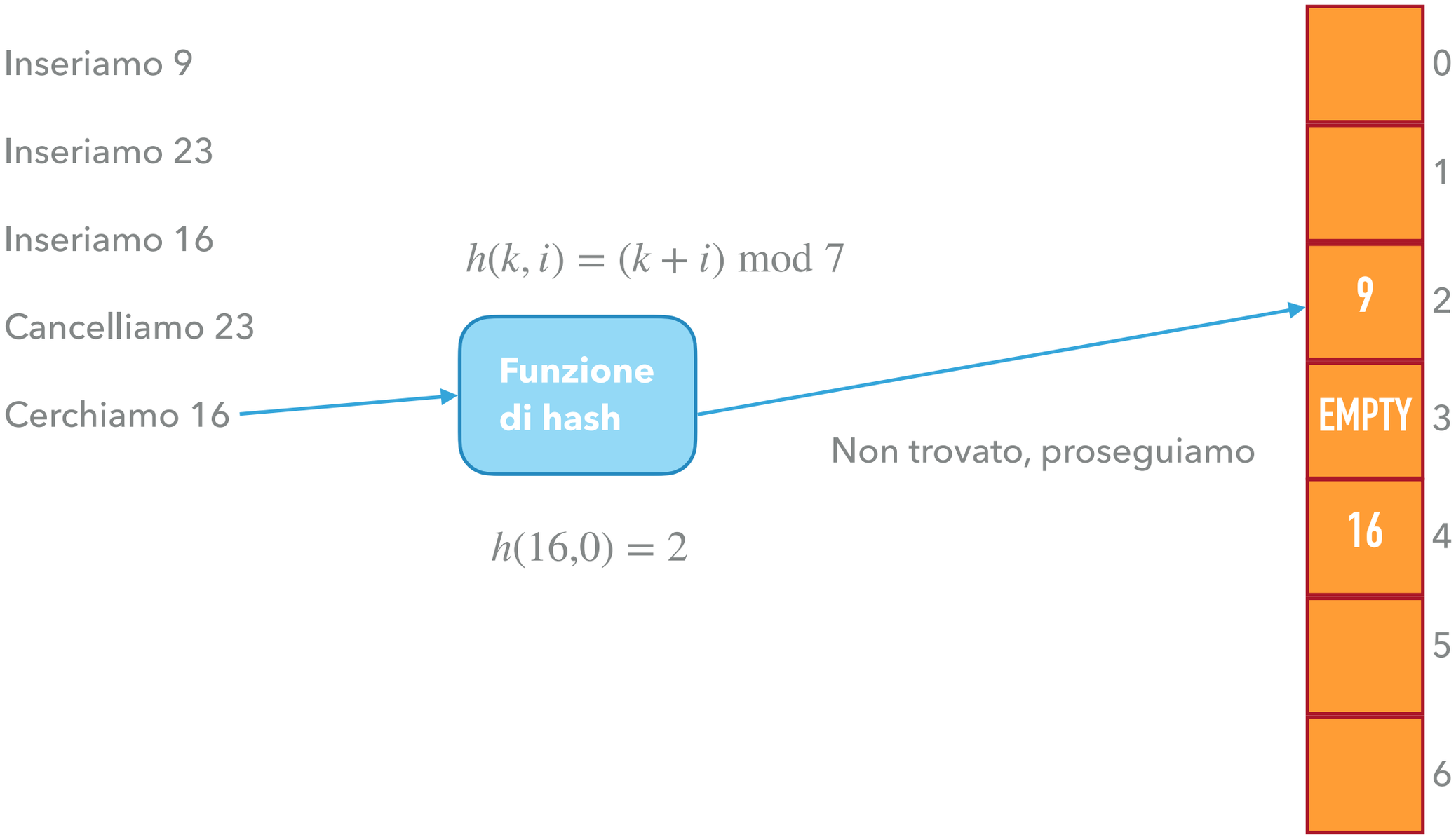
Funzione di hash

$h(23,1) = 3$

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



HASH: INSERIMENTO, RIMOZIONE E RICERCA



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

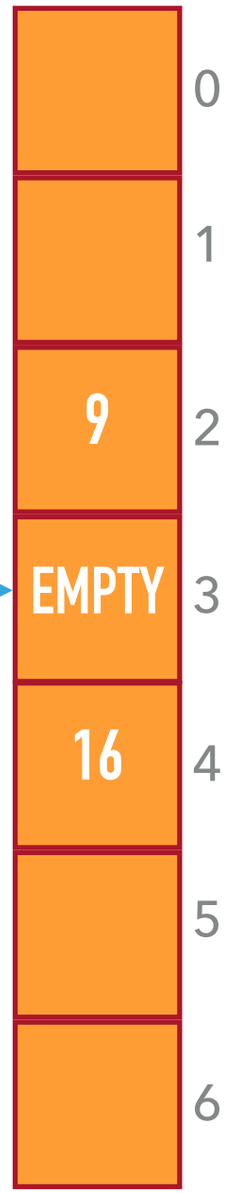
Cerchiamo 16

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

Funzione di hash

$h(16,1) = 3$

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

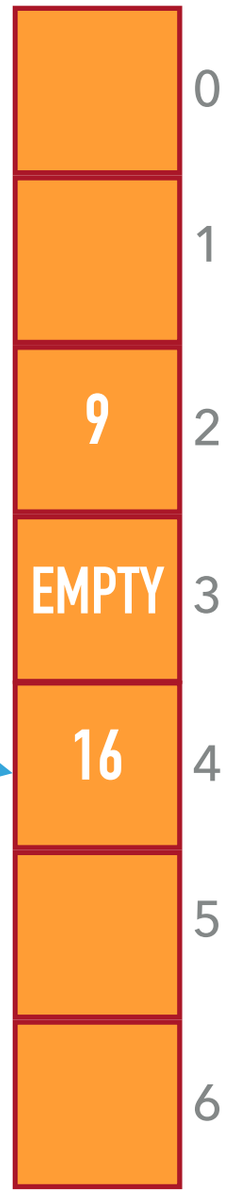
Cerchiamo 16

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

Funzione di hash

$h(16,2) = 4$

Trovato!



CANCELLAZIONE (OPEN ADDRESSING)

Cancellazione

```
Parametri: x (oggetto da cancellare) e la tabella T
i = 0
pos = h(k, i)
while T[pos] != x:
    # iteriamo fino a quando non troviamo x (assumiamo x contenuto in T)
    i = i + 1
    pos = h(x.key, i)
T[pos] = 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

α è definito ancora come n/m

Con indirizzamento aperto: ogni slot contiene al massimo un elemento → $\alpha \leq 1$

- Se non cancelliamo otteniamo dei buoni bound sul tempo necessario alla ricerca che dipenderanno dal fattore di carico α
- 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

Le cancellazioni con "segnaposto" distorcono il fattore di carico effettivo e fanno degradare le prestazioni anche di 2–3× o più, soprattutto per ricerche senza successo.

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.

1. Rehashing periodico: quando ci sono troppi segnaposto, si crea una nuova tabella e si reinseriscono solo gli elementi validi.

2. Contatori separati: si tiene traccia del numero reale di segnaposto e si imposta una soglia massima

3. Lazy deletion + reinserimento attivo in fase di ricerca/inserimento