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 **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)

**la sequenza contiene tutti gli slot della tabella una sola volta*

Anche se ci sono collisioni, la sequenza di probing esplora potenzialmente tutta la tabella

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

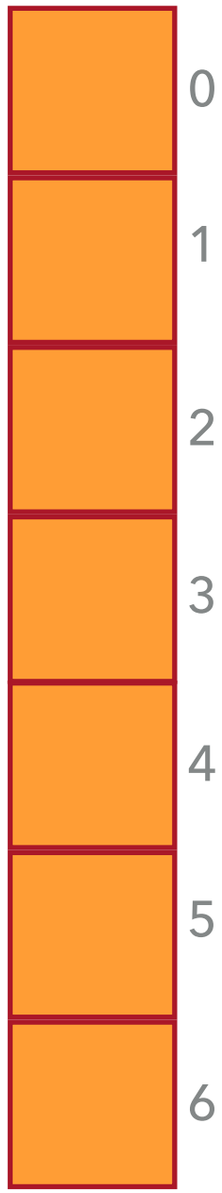
Inseriamo 16

Cancelliamo 23

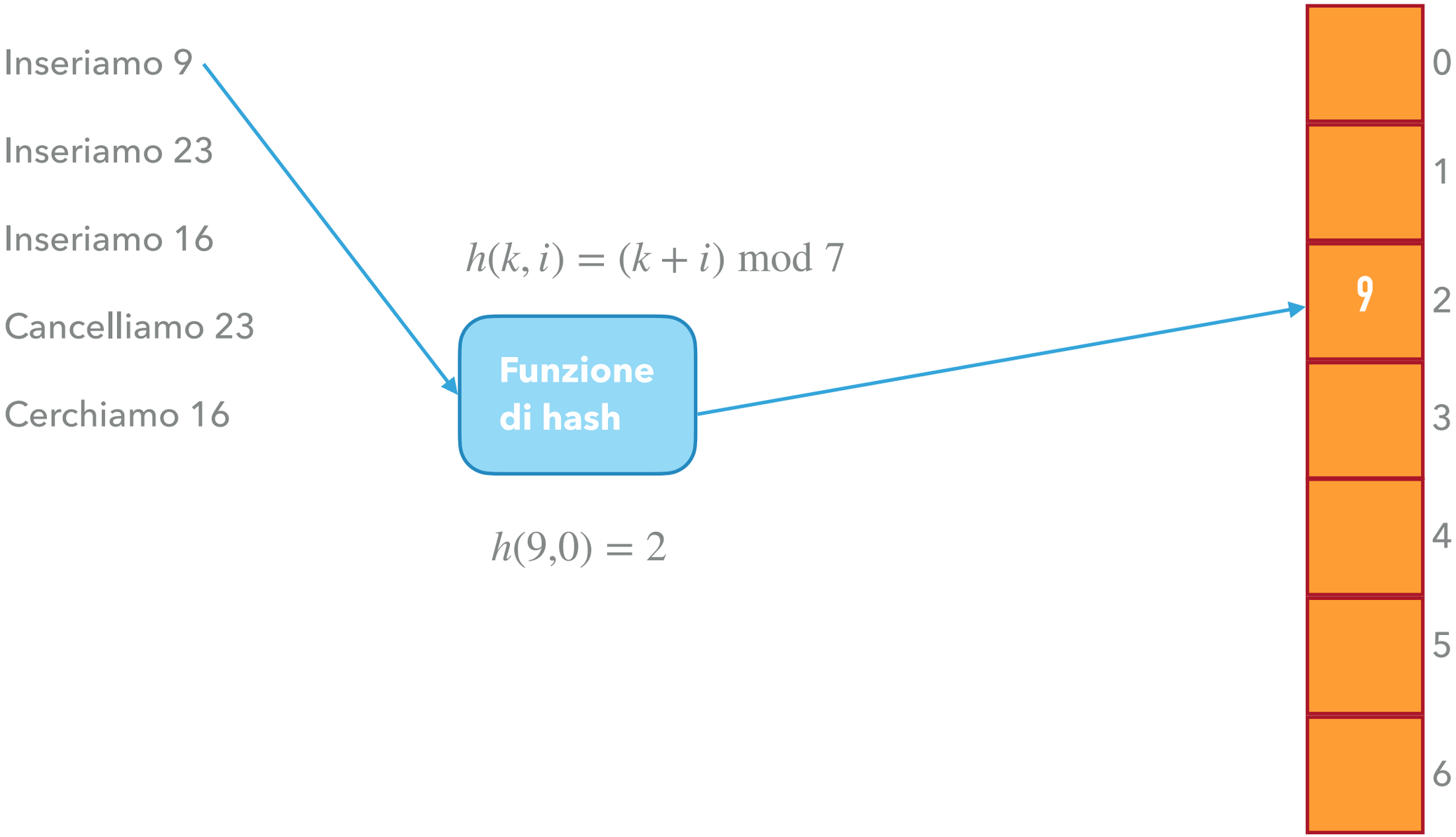
Cerchiamo 16

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

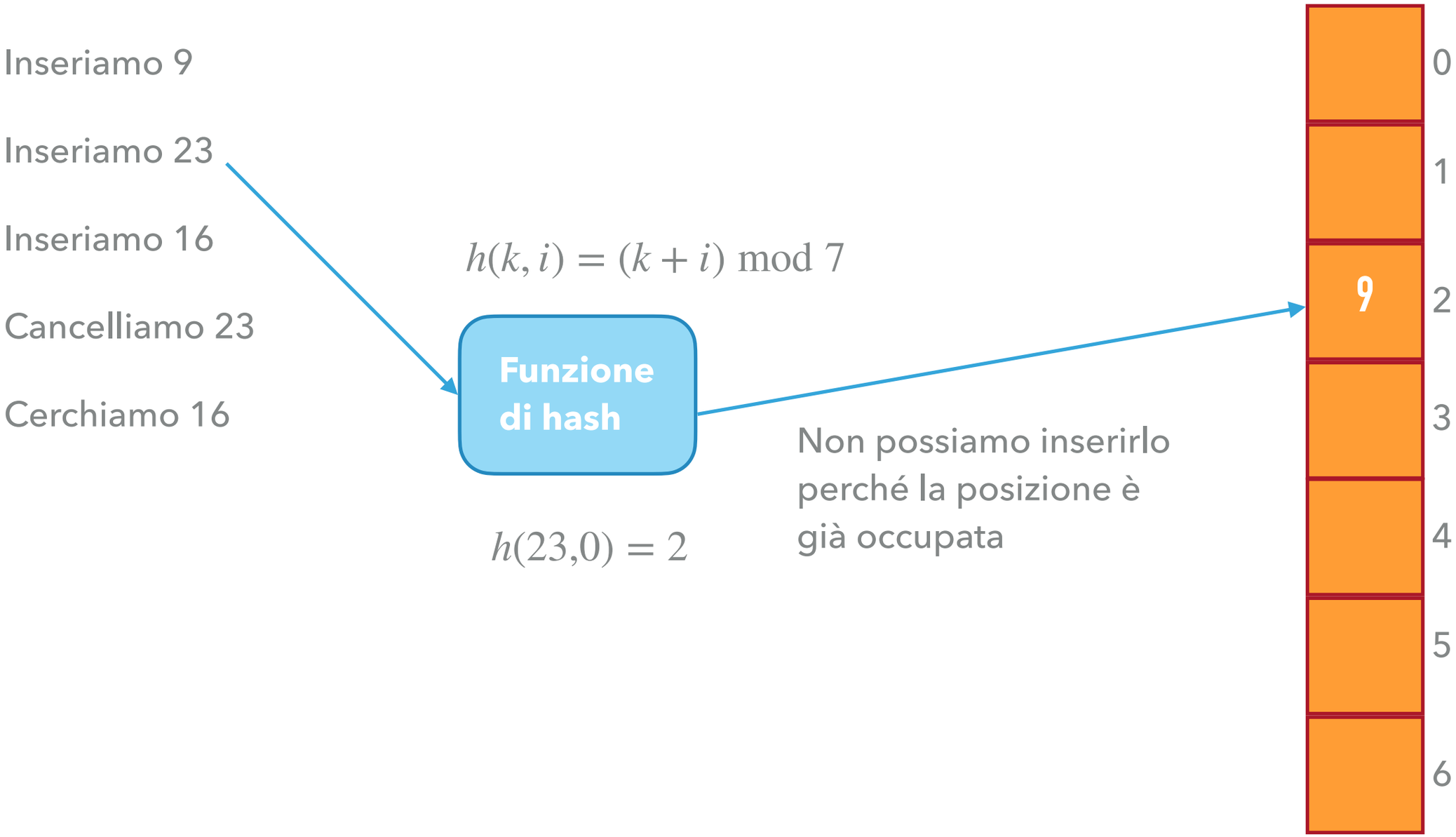
Funzione di hash



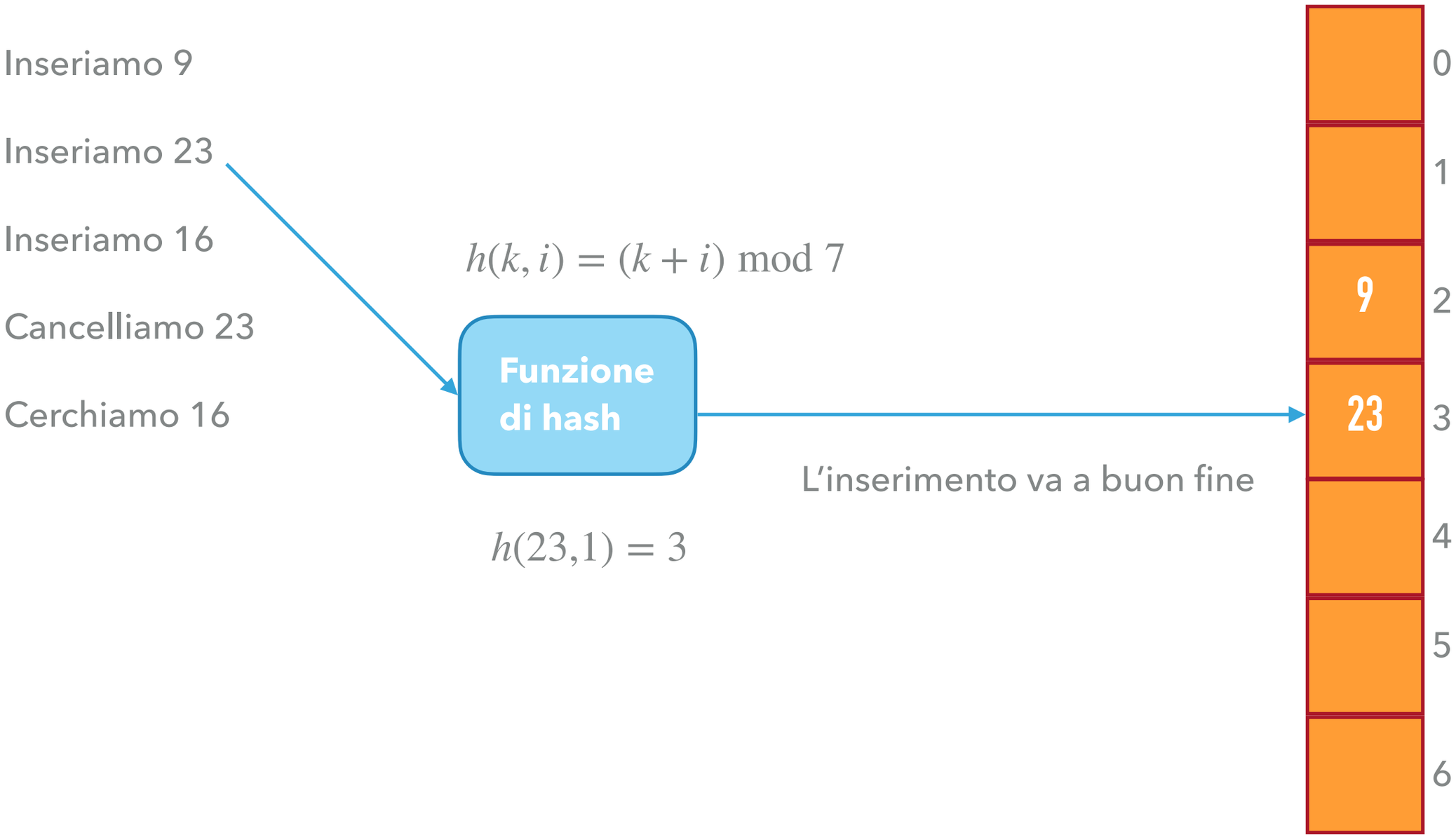
HASH: INSERIMENTO, RIMOZIONE E RICERCA



HASH: INSERIMENTO, RIMOZIONE E RICERCA



HASH: INSERIMENTO, RIMOZIONE E RICERCA



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

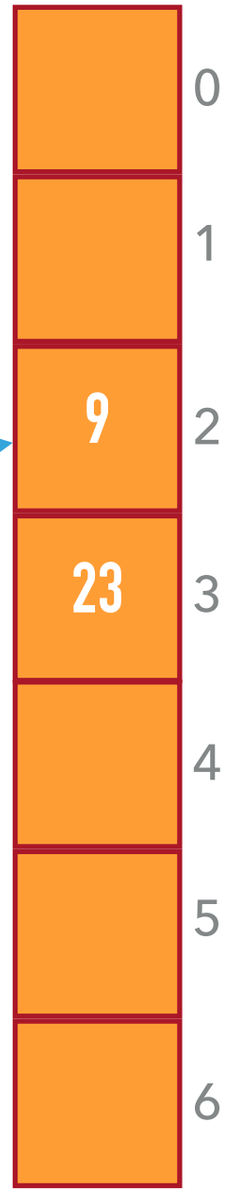
Cerchiamo 16

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

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

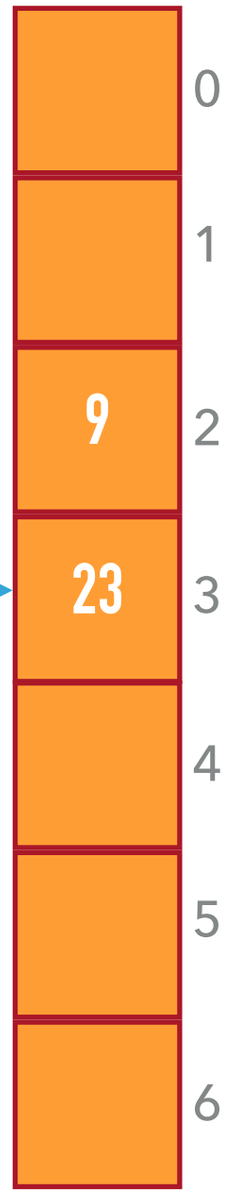
Cerchiamo 16

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

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

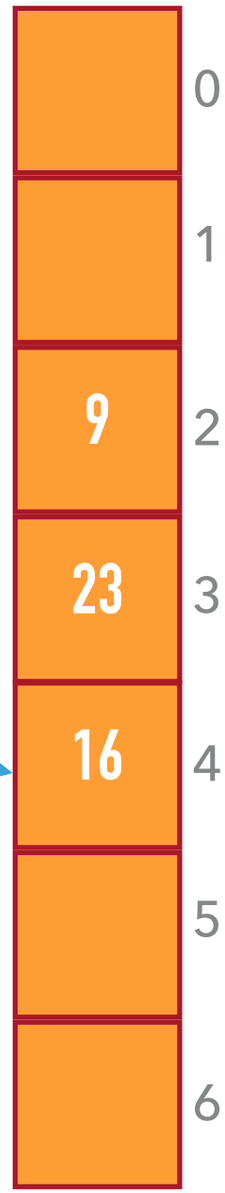
Cerchiamo 16

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

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

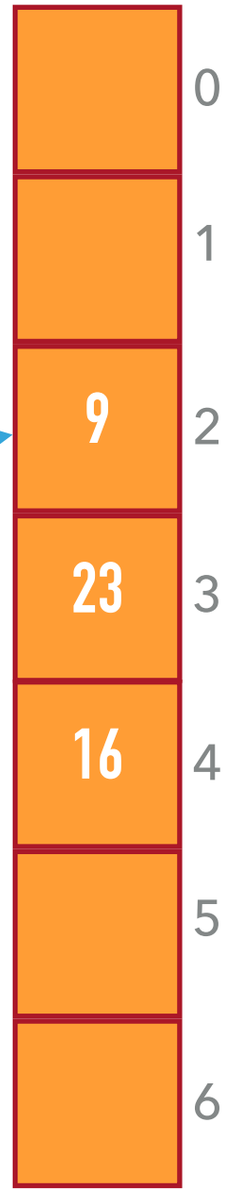
Cerchiamo 16

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

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

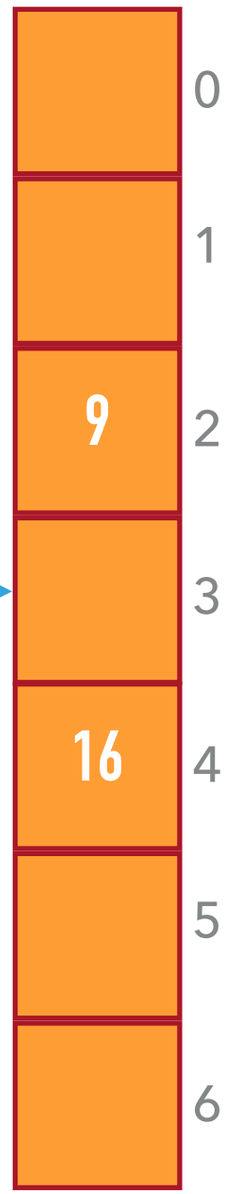
Cancelliamo 23

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

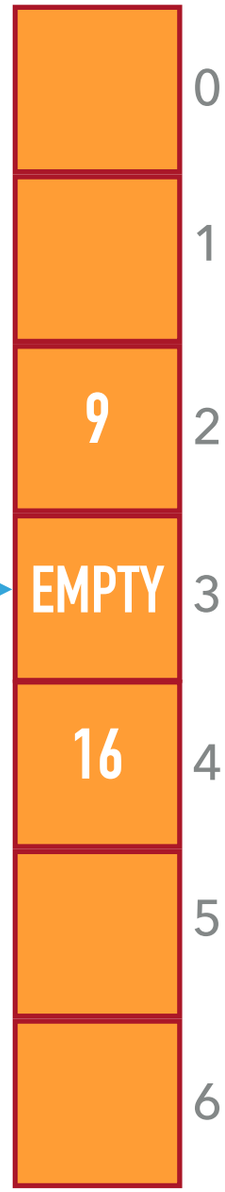
Cerchiamo 16

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

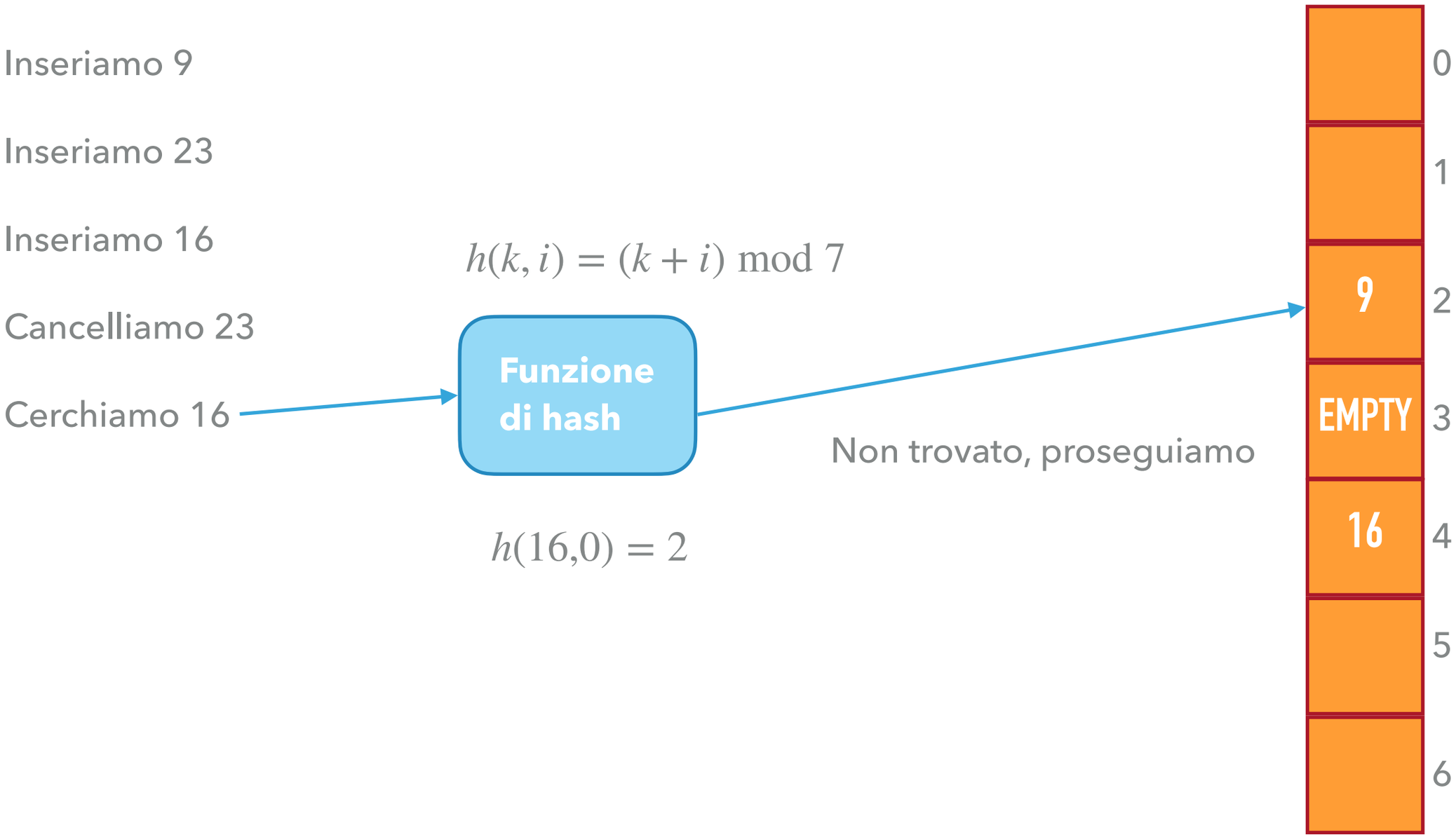
Funzione di hash

$h(23,1) = 3$

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

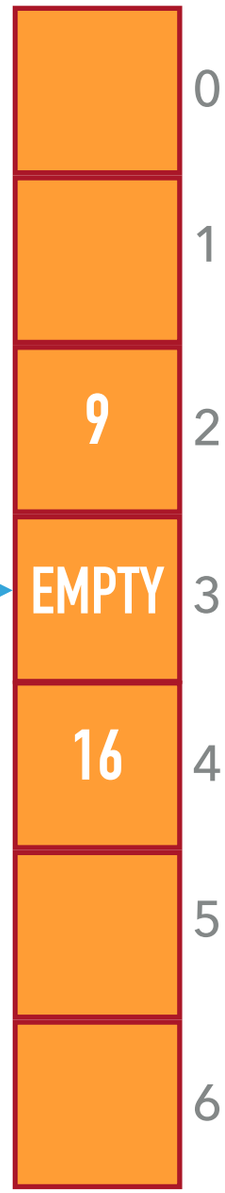
Cerchiamo 16

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

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

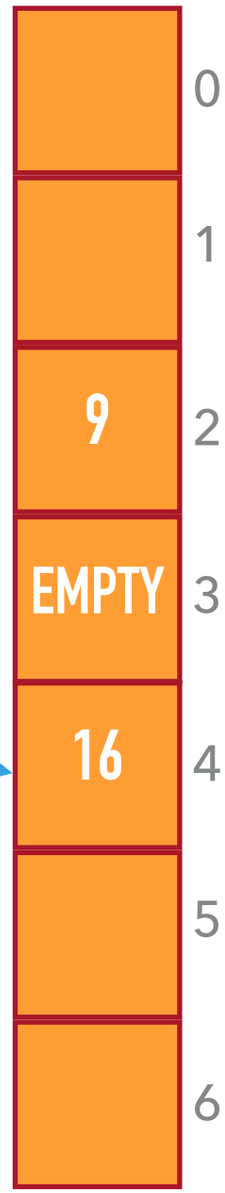
Cerchiamo 16

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

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 (ovvero ci saranno segnaposto EMPTY in tutti gli slot)
  - 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 (sposto gli elementi dopo il segnaposto, uno alla volta man mano che viene chiesto di cercarli)