Dizionari

- Collezione di elementi sui quali è possibile effettuare tre operazioni fondamentali: **ricerca**, **inserimento** e **cancellazione**.
- Ogni elemento e è identificato dalla sua **chiave di ricerca** $e.chiave$.
- Un elemento e può contenere altre informazioni che vengono dette *dati satellite* e indicate da $e.sat$

Dizionario è detto **statico** se consente solo oprazioni di `ricerca`, altrimenti è detto **dinamico**

Esempi di applicazioni:

- Dizionari linguistici: collezione di coppie (parola, definizione)
- Sistema DNS: collezione di coppie (nome dominio,indirizzo IP), Es.: (datastructures.net, 128.148.35.101)

Dizionari

Assumiamo che le chiavi degli elementi di un dizionario siano a due a due distinte.

- **Ricerca**(k): restituisce e tale che e.`chiave` $= k$; restituisce `null` se il dizionario non contiene elementi con chiave k
- **Inserisci**(e): inserisce l'elemento e nel dizionario se in esso non è gia presente un elemento con chiave uguale a quella di e
- **Cancella**(k): cancella l'elemento con chiave e; non fa niente se il dizionario non contiene alcun elemento con chiave uguale a k

Dizionario ordinato

- Per ogni coppia di chiavi $k, k' \in U$ vale $k \leq k'$ o $k' \leq k$
  - **Predecessore**$(k) =$ e se e.chiave $= \max_{f \in S}\{\text{f.chiave} \leq k\}$
  - **Successore**$(k) =$ e se e.chiave $= \min_{f \in S}\{\text{f.chiave} \geq k\}$
  - **Intervallo**$(k, k') = \{e \in S : k \leq \text{e.chiave} \leq k'\}$
  - **Rango**$(k) =$ cardinalità di $\{e \in S : \text{e.chiave} \leq k\}$

Liste doppie e dizionari

- Lista doppia `L` è un descrittore con tre campi
  - `L.inizio` (=`null` per la lista vuota)
  - `L.fine` (=`null` per la lista vuota)
  - `L.lunghezza` (=0 per la lista vuota)
- Ogni nodo `p` della lista ha tre campi (capitolo 3)
  - `p.pred`
  - `p.succ`
  - `p.dato`
    - `p.dato.chiave` è la chiave di ricerca dell'elemento nel nodo `p`
    - `p.dato.sat` sono gli eventuali dati satellite

Inserimento in cima e in fondo alla lista

```
1  InsCima(e):
2    p = NuovoNodo( );
3    p.dato = e;
4    lun = l.lunghezza;
5    IF (lun == 0) {
6      p.succ = p.pred = null;
7      l.inizio = p;
8      l.fine = p;
9    } ELSE {
10     p.succ = l.inizio;
11     p.pred = null;
12     l.inizio.pred = p;
13     l.inizio = p;
14   }
15   l.lunghezza = lun + 1;
16   RETURN l;
```

```
1  InsFondo(e):
2    p = NuovoNodo( );
3    p.dato = e;
4    lun = l.lunghezza;
5    IF (lun == 0) {
6      p.succ = p.pred = null;
7      l.inizio = p;
8      l.fine = p;
9    } ELSE {
10     p.succ = null;
11     p.pred = l.fine;
12     l.fine.succ = p;
13     l.fine = p;
14   }
15   l.lunghezza = lun + 1;
16   RETURN l;
```

Ricerca e cancellazione con complessità lineare

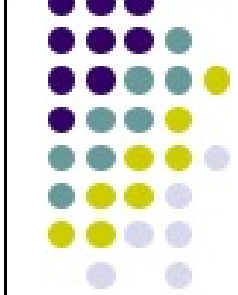
```
1  RicercaNodo( k ):
2    p = l.inizio;
3    WHILE ((p != null) && (p.dato.chiave != k))
4      p = p.succ;
5    RETURN p;
```

```
1  Ricerca k ):
2    p= RicercaNodo(k);
3    IF (p!= null) {
4      RETURN p.dato;
5    } ELSE {
6      RETURN null;
7    }
```

Ricerca e cancellazione con complessità lineare

```
1  Cancella( k ):
2    p = RicercaNodo( k );
3    IF (p != null) {
4      IF (l.lunghezza == 1) {
5        l.inizio = l.fine = null;
6      } ELSE IF (p.pred == null) {
7        p.succ.pred = null; l.inizio = p.succ;
8      } ELSE IF (p.succ == null) {
9        p.pred.succ = null; l.fine = p.pred;
10     } ELSE {
11       p.succ.pred = p.pred; p.pred.succ = p.succ;
12     }
13     l.lunghezza = l.lunghezza - 1;
14   }
15   RETURN l;
```

Tabella Hash



- Modo di implementare il dizionario mediante un array

- Problemi dell'implementazione semplice con un array A:
  - Un'entrata con chiave k viene memorizzata nella cella A[k]
    - **Problema: L'array A deve avere dimensione pari alla chiave più grande del dizionario**

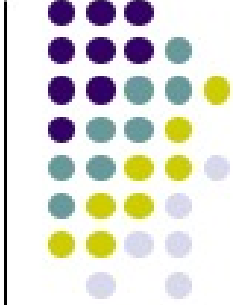
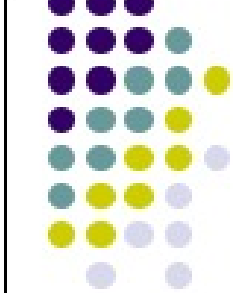


Tabella Hash

- Una tabella hash per un dato tipo di chiavi consiste di
  - Una funzione hash h
  - Un array (tabella) chiamato bucket array



Funzioni hash

- Una funzione hash h mappa un insieme chiavi in un intervallo prefissato di interi [0, m–1]
- h(x) è chiamato valore hash di x
- Lo scopo di una funzione hash è di distribuire le chiavi uniformemente nell'intervallo [0, m-1]

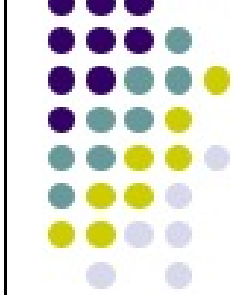
- Si verifica una collisione quando due chiavi del dizionario hanno lo stesso valore hash

Funzione Hash

- m = capacità bucket array

- Una buona funzione hash dovrebbe garantire che la probabilità che due chiavi vengano "mappate" nello stesso bucket è 1/m.
  - Questo e` lo stesso comportamento che si avrebbe nel caso in cui la funzione distribuisse con probabilita` uniforme gli elementi nelle celle dell'array.
  - NB: la funzione hash e` deterministica (associa ad una stessa chiave sempre lo stesso valore) ma si comporta come se avesse un comportamento casuale.

Funzione hash



- Per semplicita` assumiamo che le chiavi siano numeri naturali (interi non negativi)
- Nel caso in cui le chiavi non fossero numeri neturali, occorrerebbe applicare un metodo per trasformarli in numeri naturali
- Esempio: se la chiave e` una stringa, si possono sommare i valori ASCII dei caratteri della stringa

Funzione Hash

- Esempi di funzioni hash
  - Funzione modulo
    - hash(k)= k % m , le collisioni sono meno probabili se m è un primo
      - Genera molte collisioni se molte chiavi hanno un hash code della forma

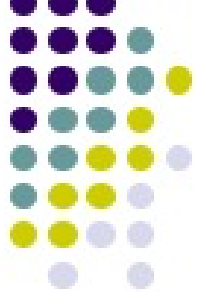
        pm+q per diversi valori di p

        Es. Chiavi {200, 205, 210, 215, …600} e bucket size 100
  - Funzione iterativa
    - hash(k)= $k_0 \oplus k_1 \oplus \ldots \oplus k_{s-1}, 0 \le k_i \le m-1$, dove $k_0, k_1, \ldots k_{s-1}$ sono ottenuti dividendo la rappresentazione binaria di k in s segmenti di uguale lunghezza.
      - Genera molte collisioni quando molte chiavi sono partizionate in sequenze di segmenti uguali a meno di una permutazione

Dizionari implementati con tabelle hash

- Schemi di risoluzione di una collisione:
  - Liste di trabocco (chaining): le entrate che hanno generato la collisione vengono memorizzate in una stessa lista
  - Indirizzamento aperto (open addressing): se un elemento genera una collisione con un elemento gia` presente nella tabella allora viene sistemato in un'altra cella della tabella

Liste di trabocco (chaining)

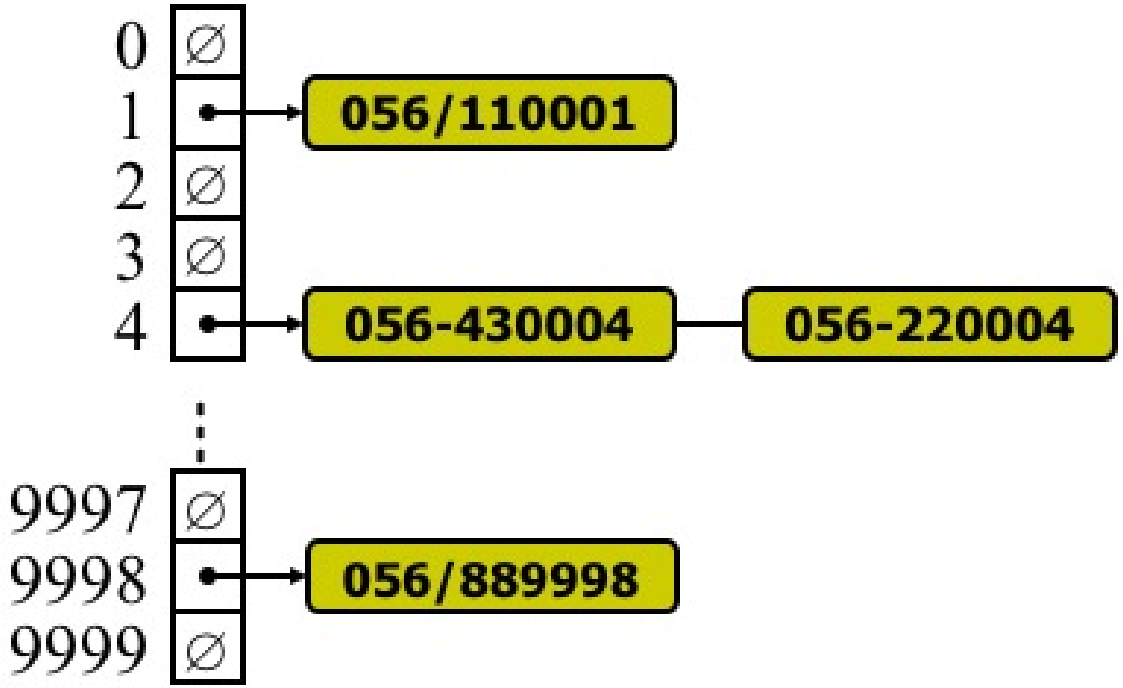


- Ciascuna entrata dell' array è l'indirizzo di una lista
- tabella[**h**] → S_h
  - S_h memorizza tutte le chiavi che hanno valore hash uguale ad **h**
  - S_h può essere visto come un piccolo dizionario implementato con una lista

Esempio

- tabella hash per un dizionario che contiene elementi della forma (matricola, nome studente), dove la matricola e` la chiave e il nome dello studente e` il dato satellite (per semplicita` nel disegno compaiono solo le chiavi)
  - Array di dimensione m=10000
  - Valore hash di h(x)= ultime 4 cifre di x
  - Risoluzione delle collisioni con chaining

La matricola 056-430004
e la matricola 056-220004
collidono



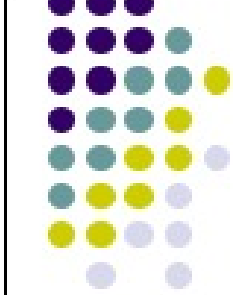
Codice per tabelle hash con liste di trabocco

```
Ricerca( k ):
  h = Hash(k);
  RETURN tabella[h].Ricerca( k );

Inserisci( e ):
  IF (Ricerca( e.chiave ) == null) {
    h = Hash( e.chiave );
    tabella[h].Insfondo( e );
  }

Cancella( k ):
  IF (Ricerca( k ) != null) {
    h = Hash(k);
    tabella[h].Cancella( k );
  }
```

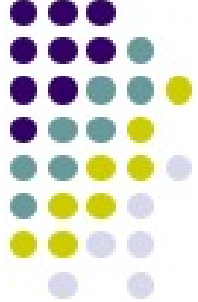
Fattore di caricamento



- m = dimensione bucket array
- n = numero entrate nella tabella
- Load factor: $\alpha = n/m$
- Se la funzione hash distribuisce con probabilita` uniforme gli elementi nelle m liste di trabocco, le liste di trabocco contengono un numero **medio** di elementi pari a $n/m = \alpha$
  - In questo caso il costo **medio** delle operazioni sulla tabella hash con liste di trabocco e` $O(1+\alpha)$
  - se $\alpha = O(1)$ allora il costo **medio** delle operazioni sulla tabella hash con liste di trabocco e` $O(1)$

- NB: Nel caso pessimo il costo delle operazioni e` $O(n)$

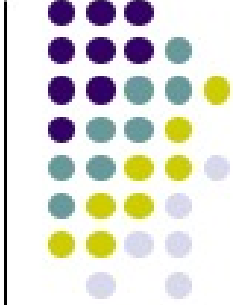
Indirizzamento aperto (open addressing)



- Il metodo dell'*open addressing* risolve le collisioni sistemando l'elemento che provoca una collisione in un'altra locazione dell'array
  - Di conseguenza, il fattore di caricamento α e` ≤ 1

- Questo metodo è utile quando non si ha molto spazio a disposizione
  - Non si utilizzano strutture dati ausiliare a differenza di quanto avviene nel chaining

Indirizzamento aperto (open addressing)

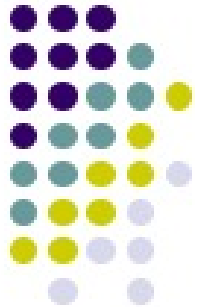


- Si usano m funzioni hash:
  - Hash[0], Hash[1],…,Hash[m-1]
  - Queste funzioni producono la cosiddetta sequenza di scansione (probing)
  - Ogni volta che si vuole inserire un elemento con una nuova chiave k nella tabella, vengono scandite le celle della tabella con indice Hash[0](k), Hash[1](k),…,Hash[m-1](k), in quest'ordine, fino a che non si arriva ad una cella di indice Hash[i](k) libera
  - Gli indici Hash[0](k), Hash[1](k),…,Hash[m-1](k) formano una permutazione degli indici dell'array 0,1,…,m-1

Indirizzamento aperto (open addressing)

- Ogni volta che si cerca una chiave k nella tabella, vengono scandite le celle della tabella con indice Hash[0](k), Hash[1](k),…,Hash[m-1](k), in quest'ordine, fino a che non si arriva ad una cella di indice Hash[i](k) che
  - contiene k oppure
  - e` libera (in questo caso l'algoritmo conclude che la chiave k non e` nella tabella)

Indirizzamento aperto (open addressing)



- Per cancellare una chiave k dalla tabella:
  - si effettua la ricerca della chiave k come descritto nella slide precedente
  - si sostituisce l'elemento con chiave k con un elemento marcatore che indica
    - all'algoritmo di inserimento che la cella puo` essere occupata dal nuovo elemento
    - all'algoritmo di ricerca che la cella in precedenza era occupata e che quindi non deve arrestare la ricerca.

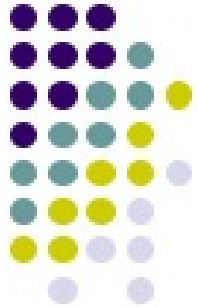
Codice per tabelle hash con indirizzamento aperto

```
1  Ricerca( k ):
2    FOR (i = 0; i < m; i = i+1) {
3      h = Hash[i](k);
4      IF (tabella[h] == null) RETURN null;
5      IF (tabella[h].chiave == k) RETURN tabella[h];
6    }
```

```
1  Inserisci( e ):
2    IF (Ricerca( e.chiave ) == null) {
3      i = -1;
4      DO {
5        i = i+1;
6        h = Hash[i]( e.chiave );
7        IF (tabella[h] == null || tabella[h]==avail) tabella[h] = e;
8      } WHILE (tabella[h] != e);
9    }
```

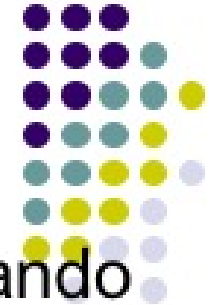
```
1  Cancella( k ):
2    FOR (i = 0; i < m; i = i+1) {
3      h = Hash[i](k);
4      IF (tabella[h] == null) RETURN;
5      IF (tabella[h].chiave == k) {
         tabella[h]=avail; RETURN;
6      }
7    }
```

Indirizzamento aperto (open addressing)



- Se per ogni chiave k, la sequenza di scansione Hash[0](k), Hash[1](k),…,Hash[m-1](k) forma una delle m! permutazioni degli indici dell'array con probabilita` uniforme allora il costo medio delle operazioni sulla tabella e` O(1/(1-α))=O(1)

Open Addressing con scansione lineare (linear probing)

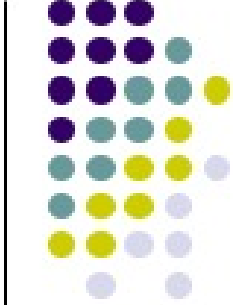


- Il metodo del *linear probing* risolve le collisioni sistemando l'elemento che provoca una collisione nella prossima cella disponibile della tabella (vista come struttura circolare)

$$Hash[i](k)=(Hash(k)+i)\%\ m,\ i=0,\ldots,m-1$$

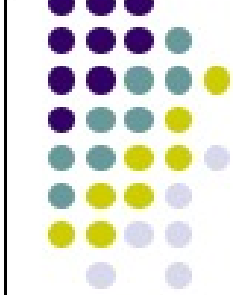
- Problema:
  - Primary clustering: le entrate tendono ad essere disposte in lunghi blocchi di celle consecutive aumentando così il numero di probe necessari per cercare un' entrata o inserire una nuova entrata

Metodi alternativi al linear probing



- *Scansione quadratica (quadratic probing):* $Hash[i](k)=(Hash(k)+ai^2+bi+c) \% m$, a, b e c costanti con a ≠0
  - m, a, b e c devono essere scelti in modo appropriato altrimenti potrebbero esserci locazioni inutilizzate
  - Problema: come nel linear probing se $Hash(k_1) = Hash(k_2)$ allora $Hash[i](k_1) = Hash[i](k_2)$ per ogni i ➔ le sequenze dei probe effettuati per k_1 e per k_2 sono le stesse ➔ secondary clustering

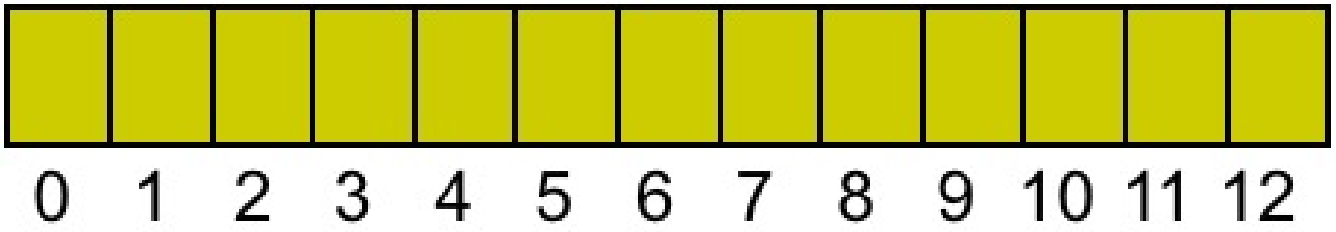
Metodi alternativi al linear probing



- *double hashing (hash doppio):*
  Hash[i](k)=(Hash(k)+i(1+Hash'(k)))% m,
  dove Hash e Hash' devono essere scelte in modo che Hash ≠Hash' e che per ogni k vengano ottenuti tutti gli indici 0,1,…,m-1.

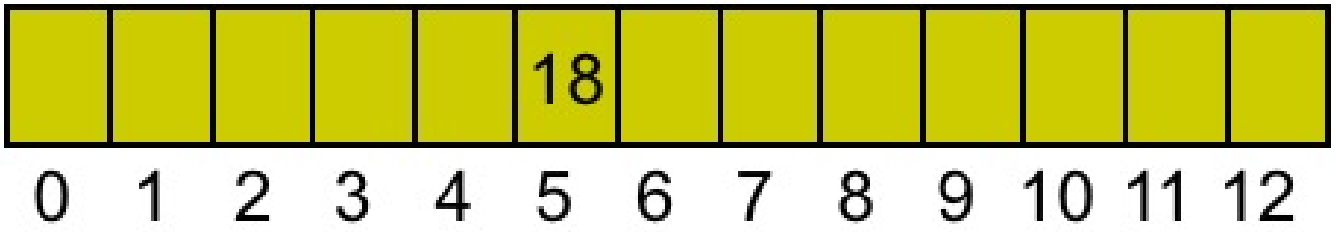
Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



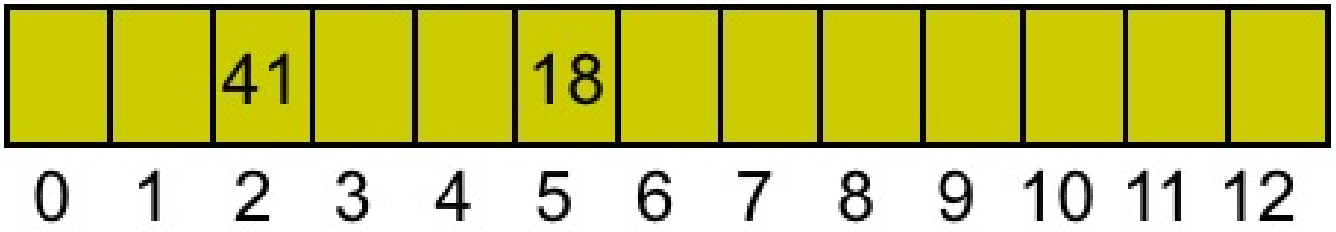
Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



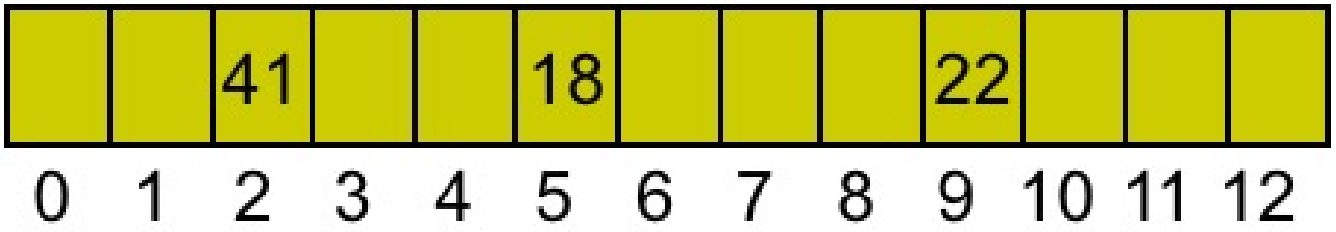
Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



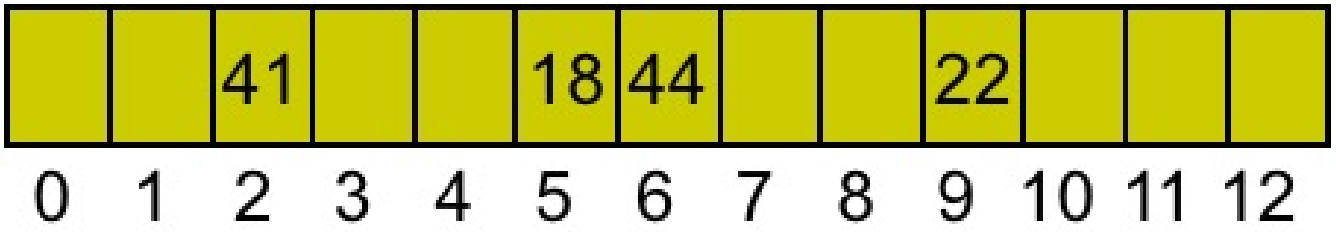
Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



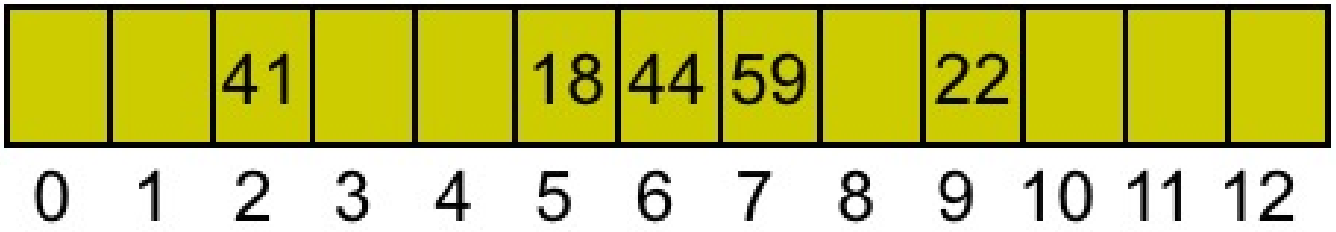
Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



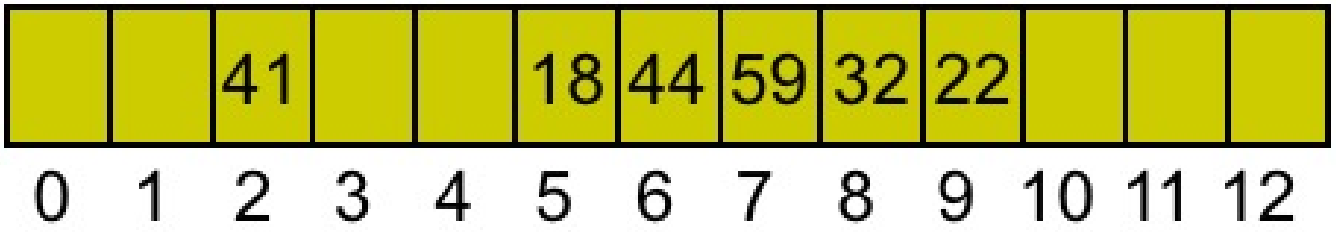
Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



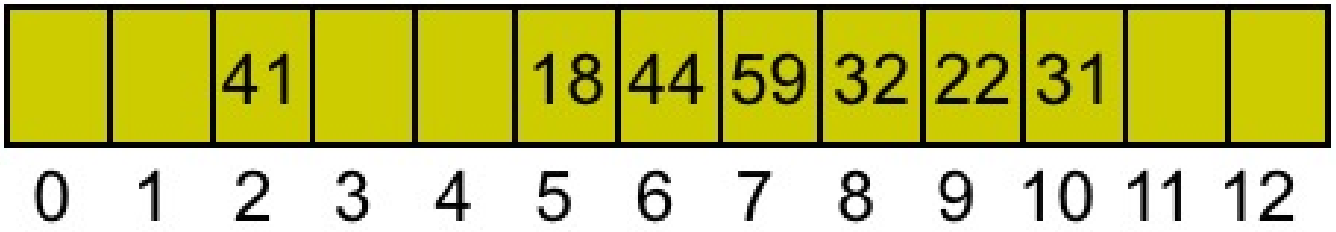
Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine