Algoritmi e Strutture Dati

Tabelle di hash

Fabio Patrizi

Implementazioni Dizionario

Tempo richiesto dall'operazione più costosa:

- Liste	O(n)
- Alberi di ricerca non bilanciati	O(n)
- Alberi di ricerca bilanciati	O(log n)
- Tabelle hash	O(1)

Tabelle ad accesso diretto

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array

Idea:

- dizionario memorizzato in array **v** di **m** celle
- a ciascun elemento è associata una chiave intera nell'intervallo **[0,m-1]**
- elemento con chiave **k** contenuto in **v[k]**
- al più **n≤m** elementi nel dizionario

Implementazione

classe `TavolaAccessoDiretto` **implementa** `Dizionario`:

dati: $S(m) = \Theta(m)$

un array v di dimensione $m \geq n$ in cui $v[k] = elem$ se c'è un elemento $elem$ con chiave k nel dizionario, e $v[k] =$ `null` altrimenti. Le chiavi k devono essere interi nell'intervallo $[0,\ m-1]$.

operazioni:

`insert`$(elem\ e, chiave\ k)$ $T(n) = O(1)$

$v[k] \leftarrow e$

`delete`$(chiave\ k)$ $T(n) = O(1)$

$v[k] \leftarrow$ `null`

`search`$(chiave\ k) \rightarrow elem$ $T(n) = O(1)$

return $v[k]$

Fattore di carico

Misuriamo il grado di riempimento di una tabella usando il **fattore di carico**

$$\alpha = \frac{n}{m}$$

Dove **n** è il numero di elementi in essa memorizzati e **m** è la sua dimensione.

Esempio: tabella con nomi di studenti indicizzati da numeri di matricola a 6 cifre

n=100 e **m**=10^6 → $\alpha = 0{,}0001 = 0{,}01\%$

Grande spreco di memoria!

Pregi e difetti

Pregi:

- Tutte le operazioni richiedono tempo O(1)

Difetti:

- Le chiavi devono essere necessariamente interi in [0, m-1]
- Lo spazio utilizzato è proporzionale ad **m**, non al numero **n** di elementi: può esserci grande spreco di memoria!

Tabelle hash

Per ovviare agli inconvenienti delle tabelle ad accesso diretto ne consideriamo un'estensione: le **tabelle hash**

Idea:

- Chiavi prese da un universo totalmente ordinato **U** (possono non essere numeri)
- Funzione hash: $h: U \rightarrow [0, m-1]$ (**funzione che trasforma chiavi in indici**)
- Elemento con chiave **k** in posizione **v[h(k)]**

Collisioni

Le tabelle hash possono soffrire del fenomeno delle collisioni:

Si ha una collisione quando si deve inserire nella tabella hash un elemento con chiave **u**, e nella tabella esiste già un elemento con chiave **v** tale che **h(u)=h(v)** → il nuovo elemento andrebbe a sovrascrivere il vecchio!

Funzioni hash perfette

Un modo per evitare il fenomeno delle collisioni è usare **funzioni hash perfette**.

Una funzione hash si dice **perfetta** se è iniettiva, cioè per ogni $\mathbf{u,v} \in \mathbf{U}$:

$$u \neq v \Rightarrow h(u) \neq h(v)$$

Deve essere $|U| \leq m$

Implementazione

classe `TavolaHashPerfetta` **implementa** `Dizionario`:

dati: $S(m) = \Theta(m)$

un array v di dimensione $m \geq n$ in cui $v[h(k)] = e$ se c'è un elemento e con chiave $k \in U$ nel dizionario, e $v[h(k)] = \texttt{null}$ altrimenti. La funzione $h : U \to \{0, \ldots, m-1\}$ è una funzione hash perfetta calcolabile in tempo $O(1)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$ $\quad T(n) = O(1)$

$v[h(k)] \leftarrow e$

`delete`$(chiave\ k)$ $\quad T(n) = O(1)$

$v[h(k)] \leftarrow \texttt{null}$

`search`$(chiave\ k) \to elem$ $\quad T(n) = O(1)$

return $v[h(k)]$

Esempio

Tabella hash con nomi di studenti aventi come chiavi numeri di matricola nell'insieme U=[234717, 235717]

Funzione hash perfetta: h(k) = k - 234717

n=100 m=1000 $\alpha = 0,1 = 10\%$

L'assunzione $|U| \leq m$ necessaria per avere una funzione hash perfetta è raramente conveniente (o possibile)…

Esempio

Tabella hash con elementi aventi come chiavi lettere dell'alfabeto U={A,B,C,…}

Funzione hash non perfetta (ma buona in pratica con **m** numero primo):

h(k) = ascii(k) mod m

Ad esempio, per m=11: h('C') = h('N')

⇒ se volessimo inserire sia 'C' che 'N' nel dizionario avremmo una collisione!

Uniformità delle funzioni hash

Per ridurre la probabilità di collisioni, una buona funzione hash dovrebbe essere in grado di distribuire in modo uniforme le chiavi nello spazio degli indici della tabella

Questo si ha ad esempio se la funzione hash gode della proprietà di uniformità semplice

Uniformità semplice

Sia *P(k)* la probabilità che la chiave k sia presente nel dizionario e sia:

$$Q(i) = \sum_{k:h(k)=i} \mathcal{P}(k)$$

la probabilità che la cella i sia occupata.

Una funzione hash **h** gode **dell'uniformità semplice** se per ogni **i** si ha:

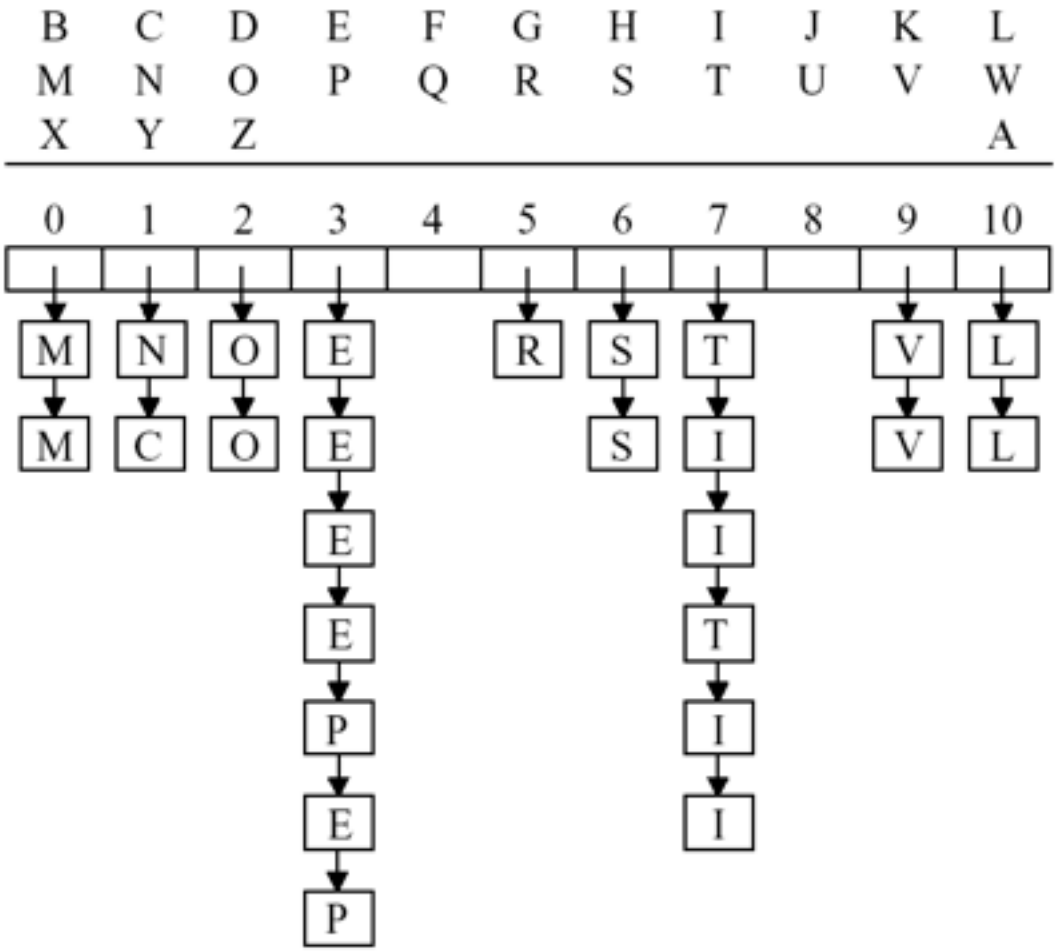
$$Q(i) = \frac{1}{m}$$

Risoluzione delle collisioni

Nel caso in cui non si possano evitare le collisioni, dobbiamo trovare un modo per risolverle. Due metodi classici sono i seguenti:

1. **Liste di collisione**. Gli elementi sono contenuti in liste esterne alla tabella: v[i] punta alla lista degli elementi tali che h(k)=i
2. **Indirizzamento aperto**. Tutti gli elementi sono contenuti nella tabella: se una cella è occupata, se ne cerca un'altra libera

Liste di collisione



Esempio di tabella hash basata su liste di collisione contenente le lettere della parola:

PRECIPITEVOLISS IMEVOLMENTE

Liste di collisione - costo

- La lunghezza media di una lista di collisione sarà pari al fattore di carico α = **n/m**. Quindi, assumento di usare una funzione di hash che gode della uniformità semplice, il tempo medio per la ricerca di un elemento sarà: $T_{avg}(n, m) = O(1 + n/m)$

- A differenza delle tabelle ad accesso diretto e delle tabelle di hash con funzione hash perfetta, usando liste di collisione possiamo avere **fattori di carico** $\alpha > 1$

Implementazione

classe `TavolaHashListeColl` **implementa** `Dizionario`:

dati: $S(m,n) = \Theta(m+n)$

un array v di dimensione m in cui ogni cella contiene un puntatore a una lista di coppie $(elem, chiave)$. Un elemento e con chiave $k \in U$ è nel dizionario se e solo se (e,k) è nella lista puntata da $v[h(k)]$, con $h : U \to \{0,\ldots,m-1\}$ funzione hash con uniformità semplice calcolabile in tempo $O(1)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$ $T(n) = O(1)$

aggiungi la coppia (e,k) alla lista puntata da $v[h(k)]$.

`delete`$(chiave\ k)$ $T_{avg}(n) = O(1+n/m)$

rimuovi la coppia (e,k) nella lista puntata da $v[h(k)]$.

`search`$(chiave\ k) \to elem$ $T_{avg}(n) = O(1+n/m)$

se (e,k) è nella lista puntata da $v[h(k)]$, allora restituisci e, altrimenti restituisci `null`.

trade-off spazio/tempo

Le tabelle di hash con liste di collisione forniscono un ottimo esempio di *bilanciamento spazio-tempo*:

•Per **m = 1** (minimo spazio) tutte le **n** chiavi sono in una sola lista e la tabella diventa una struttura a ricerca sequenziale (**T(n,m) = O(1 + n/m) = O(n)**) con spazio **S(n) = O(n)**

•Se invece siamo disposti ad utilizzare molto spazio, allora possiamo usare una funzione di hash perfetta ottenendo tempo per la ricerca **T(n) = O(1)** con spazio **O(|U|)** dove **U** è l'universo delle chiavi associabili agli elementi del dizionario.

Esercizio: liste di collisione

Sia dato l'insieme di chiavi **K** = { 35, 83, 57, 26, 15, 63, 97, 46 } e sia **m** = 11.

1. Calcolare per ogni chiave *k* di **K** la funzione di hash

$$h(\mathbf{k}) = \mathbf{k} \bmod \mathbf{m}$$

2. Inserire le chiavi dell'insieme *K* in una tabella hash (inizialmente vuota) di dimensione **m** usando le **liste di collisione.**

Soluzione: liste di collisione / 1

K = { 35, 83, 57, 26, 15, 63, 97, 46 }; **m** = 11.

1. Calcolare per ogni chiave *k* di **K** la funzione di hash

$$h(\mathbf{k}) = \mathbf{k} \bmod \mathbf{m}$$

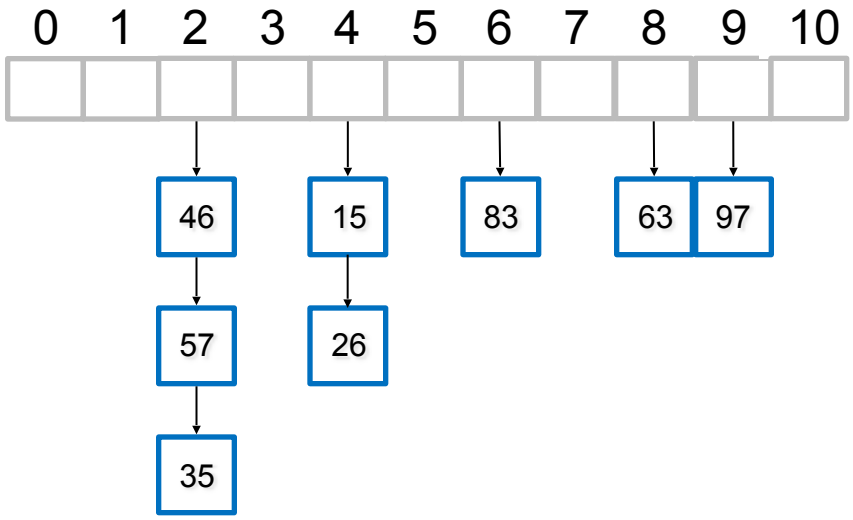
- h(35) = 2
- h(83) = 6
- h(57) = 2
- h(26) = 4
- h(15) = 4
- h(63) = 8
- h(97) = 9
- h(46) = 2

Soluzione: liste di collisione / 2

K = { 35, 83, 57, 26, 15, 63, 97, 46 }; **m** = 11.

2. Inserire le chiavi dell'insieme *K* in una tabella hash (inizialmente vuota) di dimensione **m** usando le **liste di collisione.**

- h(35) = 2
- h(83) = 6
- h(57) = 2
- h(26) = 4
- h(15) = 4
- h(63) = 8
- h(97) = 9
- h(46) = 2



Indirizzamento aperto

Supponiamo di voler inserire un elemento con chiave k e che la sua posizione "naturale" **h(k)** sia già occupata

L'indirizzamento aperto consiste nell'occupare un'altra cella, anche se naturalmente associata ad un'altra chiave

Cerchiamo la cella vuota (se c'è) scandendo le celle secondo una sequenza di indici:

$$c(k,0), c(k,1), c(k,2),\ldots c(k,m-1)$$

Implementazione

classe `TavolaHashAperta` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$

```
1.  for i = 0 to m − 1 do
2.      if (v[ c(k, i) ].elem = null) then
3.          v[ c(k, i) ] ← (e, k)
4.          return
5.  errore tavola piena
```

`delete`$(chiave\ k)$

errore operazione non supportata

`search`$(chiave\ k) \rightarrow elem$

```
1.  for i = 0 to m − 1 do
2.      if (v[ c(k, i) ].elem = null) then
3.          return null
4.      if (v[ c(k, i) ].chiave = k) then
5.          return v[ c(k, i) ].elem
6.  return null
```

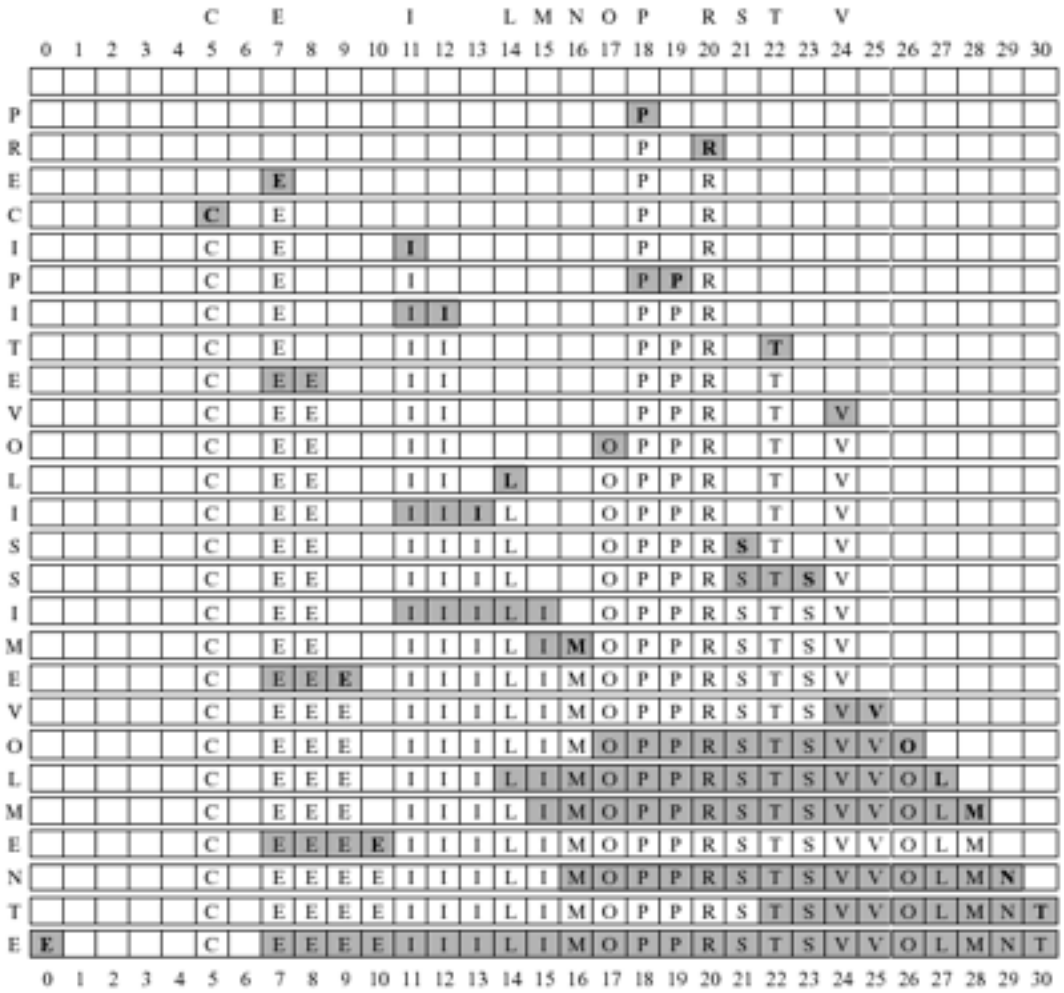
Metodi di scansione: scansione lineare

Scansione lineare:

$$c(k,i) = (h(k) + i) \bmod m$$

per $0 \leq i < m$

Esempio



Inserimenti in tabella hash basata su indirizzamento aperto con scansione lineare delle lettere della parola:

PRECIPITEVOLISS IMEVOLMENTE

4,8 celle scandite in media per inserimento

Metodi di scansione: hashing doppio

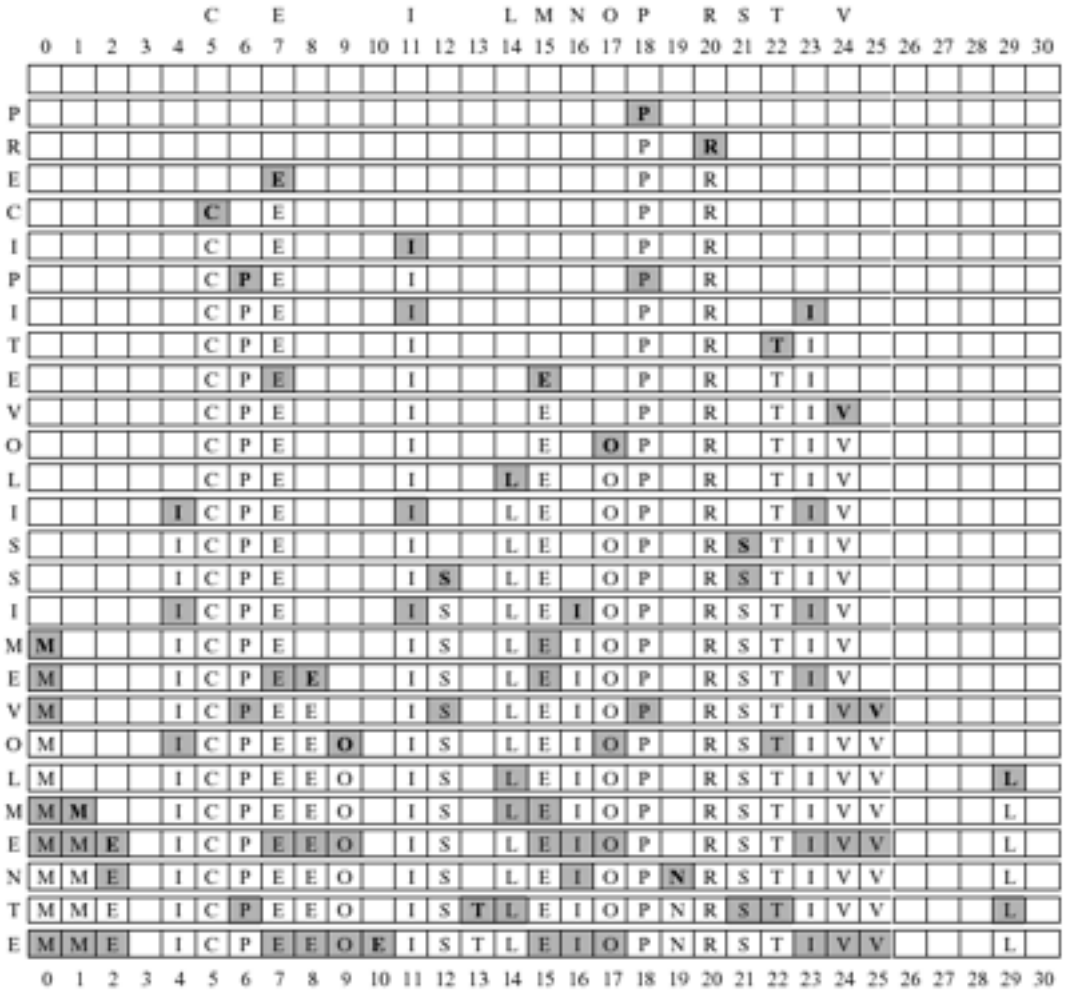
La scansione lineare provoca effetti di agglomerazione, cioè lunghi gruppi di celle consecutive occupate che rallentano la scansione

L'hashing doppio riduce il problema:

$$c(k,i) = \lfloor h_1(k) + i \cdot h_2(k) \rfloor \bmod m$$

per $0 \leq i < m$, h_1 e h_2 funzioni hash

Esempio



Inserimenti in tabella hash basata su indirizzamento aperto con hashing doppio delle lettere della parola:

PRECIPITEVOLISS IMEVOLMENTE

3,1 celle scandite in media per inserimento

Analisi del costo di scansione

Usando l'indirizzamento aperto la ricerca di un elmento o di una cella vuota può richiedere tempo O(n).

Tempo richiesto in media da un'operazione di ricerca di una chiave, assumendo che le chiavi siano prese con probabilità uniforme da U dipenderà dal fattore di carico e dalla particolare funzione c(k, i) utilizzata.

Il tempo medio richiesto per le operazioni di search, insert e delete è:

- $O\left(\frac{m^2}{(m-n)^2}\right)$ usando la *scansione lineare*
- $O\left(\frac{m}{m-n}\right)$ usando la *scansione quadratica* o l'*hashing doppio*

Esercizio: indirizzamento aperto

Sia dato l'insieme di chiavi **K** = { 35, 83, 57, 26, 15, 63, 97, 46 } e sia **m** = 11.

1. Calcolare per ogni chiave *k* di **K** la funzione di hash

$$h(\mathbf{k}) = \mathbf{k} \bmod \mathbf{m}$$

2. Inserire le chiavi dell'insieme *K* in una tabella hash (inizialmente vuota) di dimensione **m** usando l'indirizzamento aperto e con scansione lineare data da:

$$\mathbf{c(k, i) = (h(k) + i) \bmod 11}$$

Soluzione: indirizzamento aperto

$h(35) = 2$	$c(35,0) = (2 + 0) \bmod 11 = 2$	$c(97,0) = (9 + 0) \bmod 11 = 9$
$h(83) = 6$	$c(83,0) = (6 + 0) \bmod 11 = 6$	$c(46,0) = (2 + 0) \bmod 11 = 2$
$h(57) = 2$	$c(57,0) = (2 + 0) \bmod 11 = 2$	$c(46,1) = (2 + 1) \bmod 11 = 3$
$h(26) = 4$	$c(57,1) = (2 + 1) \bmod 11 = 3$	$c(46,2) = (2 + 2) \bmod 11 = 4$
$h(15) = 4$	$c(26,0) = (4 + 0) \bmod 11 = 4$	$c(46,3) = (2 + 3) \bmod 11 = 5$
$h(63) = 8$	$c(15,0) = (4 + 0) \bmod 11 = 4$	$c(46,4) = (2 + 4) \bmod 11 = 6$
$h(97) = 9$	$c(15,1) = (4 + 1) \bmod 11 = 5$	$c(46,5) = (2 + 5) \bmod 11 = 7$
$h(46) = 2$	$c(63,0) = (8 + 0) \bmod 11 = 8$	

0	1	2	3	4	5	6	7	8	9	10
		35	57	26	15	83	46	63	97	

Cancellazione elementi con indir. aperto

- Se per cancellare un elemento sostituisco il valore della cella che lo contiene con *null* l'implementazione del metodo di ricerca non funzionerebbe più!

- **Idea:** utilizzo un valore speciale *canc* invece di *null*.

Cancellazione elementi con indir. aperto

classe `TavolaHashApertaBis` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$

```
1.  for i = 0 to m − 1 do
2.      if (v[c(k,i)].elem = null or v[c(k,i)].elem = canc) then
3.          v[c(k,i)] ← (e,k)
4.          return
5.  errore tavola piena
```

`delete`$(chiave\ k)$

```
1.  for i = 0 to m − 1 do
2.      if (v[c(k,i)].elem = null) then
3.          errore chiave non in dizionario
4.      if (v[c(k,i)].chiave = k and v[c(k,i)].elem ≠ canc) then
5.          v[c(k,i)].elem ← canc
6.  errore chiave non in dizionario
```

`search`$(chiave\ k) \rightarrow elem$

```
1.  for i = 0 to m − 1 do
2.      if (v[c(k,i)].elem = null) then
3.          return null
4.      if (v[c(k,i)].chiave = k and v[c(k,i)].elem ≠ canc) then
5.          return v[c(k,i)].elem
6.  return null
```

Riepilogo

- La proprietà di accesso diretto alle celle di un array consente di realizzare dizionari con operazioni in tempo O(1) indicizzando gli elementi usando le loro stesse chiavi (purché siano intere)
- L'array può essere molto grande se lo spazio delle chiavi è grande
- Per ridurre questo problema si possono usare funzioni hash che trasformano chiavi (anche non numeriche) in indici
- Usando funzioni hash possono aversi collisioni
- Tecniche classiche per risolvere le collisioni sono liste di collisione e indirizzamento aperto