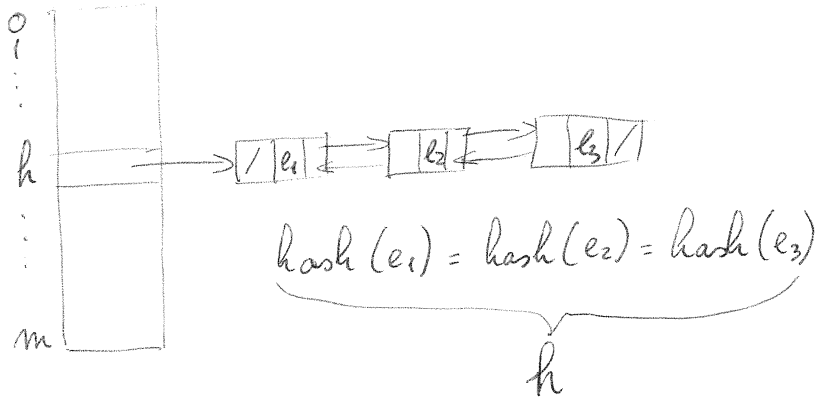
Tabelle hash: implementazione

Implementazione attraverso liste di tabocco (chaining)

La tabella è un array di m liste doppie.

Tabella[h] contiene le chiavi e dell'insieme S tali che $h = hash(e.chiave)$

Le chiavi che "collidono" vengono memorizzate nella medesima lista, etichettata con h.



Ricerca (k)

Scandisce la lista associata al valore hash della chiave

Inserimento (e)

Appurato che la chiave non è presente, inserisce un nuovo nodo con l'elemento in fondo alla lista

Cancellazione (k)

Verifica che la chiave sia presente e rimuove il corrispondente elemento dalla lista.

In Codice

```
Ricerca (K):
h = hash (K);
p = tabella[h]. Ricerca(K);          /* Invoca la funzione di ricerca sulla lista
IF (p != NULL) RETURN p.dato;
   ELSE RETURN NULL;
```

```
Inserisci (e):
IF (Ricerca (e.chiave) == NULL){
    h = hash (e.chiave);
    tabella[h]. InserisciFondo(e);          /* Invoca la funzione di inserimento in fondo alla lista
}
```

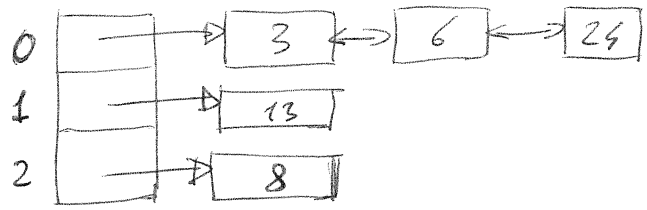
```
Cancella (K):
IF (Ricerca(K) != NULL){
    h = hash(K);
    tabella[h]. Cancella (K);          /* Invoca la funzione di cancellazione nella lista
}
```

Esempio. Sia la funzione hash definita come segue:

$$hash(K) = K \% m$$

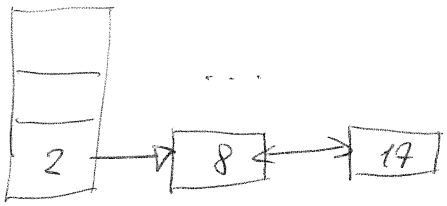
con $m = 3$, e sia $S = \{3, 6, 8, 13, 24\}$.

La tabella hash associata risulta:

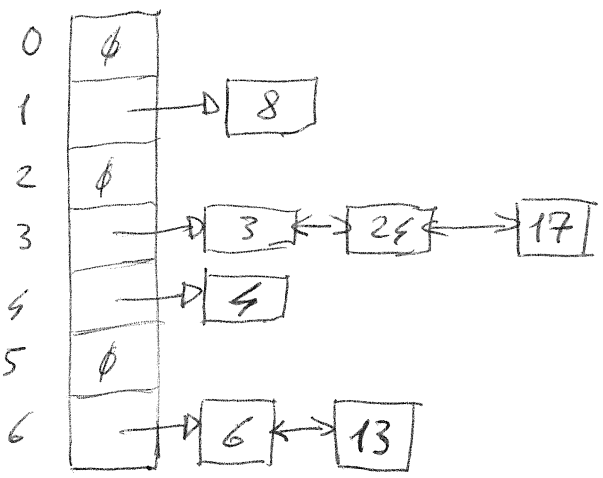


Inserendo un nuovo elemento con chiave 17 si ha:

$$hash(17) = 17 \% 3 = 2$$



A questo punto, essendo $m = 3$ e $n = 6$, un nuovo inserimento provoca un ridimensionamento della tabella. Poiché m deve essere primo, si sceglie $m = 7$.



Inserendo un nuovo elemento con chiave 4 la tabella cresce e gli elementi riposizionati in base a $hash(K) = K \% 7$.

Osservazioni: nel caso peggiore (tutte le chiavi danno luogo allo stesso valore hash) la complessità delle operazioni è $O(n)$, dove n è il numero di chiavi in tabella.

In pratica, se la funzione hash è buona, cioè distribuisce uniformemente gli n elementi di S nelle m liste, le operazioni sono efficienti.

La lunghezza media delle liste è $O(n/m)$.

Il parametro $\alpha = n/m$ è chiamato fattore di carico. Supponendo che il costo del calcolo di $h = \text{hash}(\text{e.chiave})$ sia costante e che $n = O(m)$ (i.e., il numero di elementi nella tabella è proporzionale alle celle m della tabella), allora il costo delle operazioni è $O(1+\alpha)$ risulta costante, essendo $\alpha = O(1)$. Precisamente:

Teorema 4.1 Il costo medio delle operazioni sulle tabelle hash a liste di trabocco è costante se $\alpha = O(1)$.

Nell'implementazione manterremo $n \leq 2 \cdot m$ (con raddoppio per $n = 2m$)

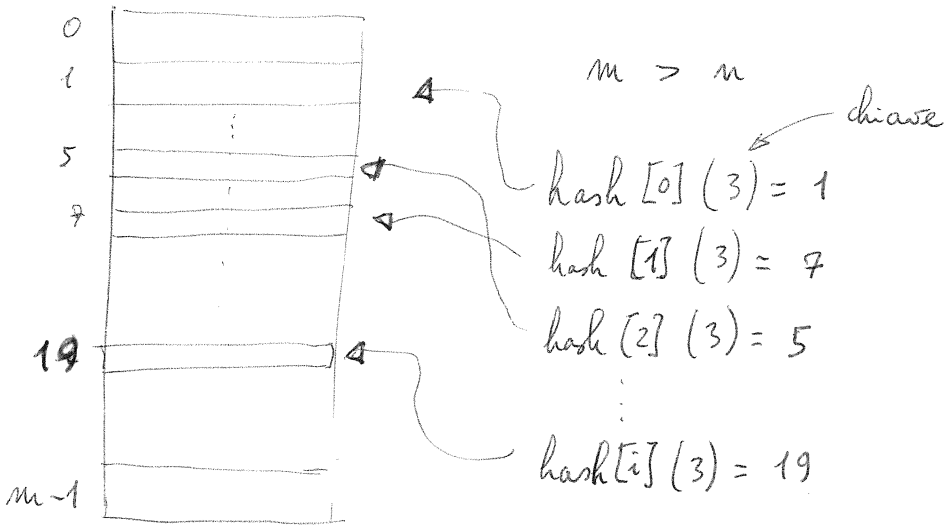
Implementazione con indirizzamento aperto (open addressing)

Tabella è un array di m celle in cui porre gli n elementi, con $m > n$. Pertanto $\alpha = n/m < 1$.

NULL viene usato per indicare una cella vuota.

Per ogni chiave $K \in \mathcal{U}$, useremo una sequenza di funzioni $hash[i](K)$, con $0 \le i \le m-1$, detta sequenza di scansione (probing). che determinano una permutazione delle posizioni $0, 1, \ldots, m-1$.

La permutazione rappresenta l'ordine con cui esaminiamo le posizioni della tabella durante le operazioni sul dizionario.



Inserimento: iniziamo ad esaminare le posizioni hash[0](K), hash[1](K), .. fino a trovare la prima posizione libera (essendo $n < m$ esiste certamente). A quel punto, inseriamo l'elemento.

Ricerca: esaminiamo le posizioni hash[0](K), hash[1](K), .. fino a trovare:

- l'elemento cercato
- posizione libera (elemento non c'è)

Cancellazione: realizzata in modo virtuale

l'elemento da cancellare viene sostituito con UNA MARCA speciale che indica che:

- la posizione è libera se si sta cercando di inserire un nuovo elemento
- la posizione è occupata, se si sta effettuando una ricerca, con un elemento diverso da quello che si sta cercando.

Osservazione: se la cancellazione non marcasse le posizioni che si liberano, la ricerca di un elemento inserito in tabella quando la posizione "liberata" era occupata, non andrebbe a buon fine!

Restituirebbe ASSENTE (avendo trovato una posizione libera applicando la sequenza di scansione) mentre invece l'elemento c'è!

Se prevediamo di effettuare molte cancellazioni, conviene usare una implementazione attraverso liste di trabocco.

Codice per ricerca e inserimento. (Assumiamo zero cancellazioni)

```
Ricerca (K);                                   n < m
for (i=0; i<m; i=i+1){
  h = hash[i](K);                              /* sequenza di
                                                  scansione
  IF (tabella[h] == NULL) RETURN NULL;
  IF (tabella[h].chiave == K) RETURN tabella[h];
}
```

Inserisci (e):

```
IF (Ricerca(e.chiave) == NULL) {
    i = -1;
    DO {
        i = i + 1;
        h = hash[i](e.chiave);
        IF (tabella[h] == NULL) tabella[h] = e;
    } WHILE (tabella[h] != e)
```

/* sandisce e quando trova una posizione vuota inserisce e termina

Complessità: caso pessimo $O(n)$, in pratica se le funzioni hash sono buone, molto veloci.

Teorema 4.2. Se per ogni chiave K, le posizioni hash[0](K), hash[1](K), ... hash[m-1](K) formano una delle $m!$ permutazioni possibili in modo uniformemente casuale, il costo medio delle operazioni su tabelle hash a indirizzamento aperto è costante se $(1-\alpha)^{-1} = O(1)$.

Dimostrazione: Il costo è direttamente proporzionale all'indice i tale che $hash[i](k)$ è la posizione individuata per la chiave.

Sia $T(n, m)$ il valore medio di i.

↓

Indica il # medio di accessi effettuati quando inseriamo un elemento in una tabella di taglia m con n posizioni già occupate, dove $n < m$.

Utilizziamo una ricorrenza per definire $T(n, m)$.

Poiché in una tabella vuota ogni posizione è disponibile, poniamo

$$T(0, m) = 1.$$

Per $n > 0$, poiché sono occupate n posizioni su m, la posizione esaminata risulta occupata n volte su m. Cioè:

$$\text{Prob}(\text{posizione esaminata occupata}) = \frac{n}{m}.$$

$$\text{Prob}(\text{posizione esaminata libera}) = 1 - \frac{n}{m} = \frac{m-n}{m}.$$

Pertanto:

- effettuiamo un solo accesso e ci fermiamo se la posizione è libera. Questo evento accade con probabilità $\frac{m-n}{m}$

- effettuiamo uno più $T(n-1, m-1)$ accessi alle rimanenti posizioni se la posizione è occupata. Questo evento accade con probabilità $\frac{n}{m}$.

Facendo la media pesata dei due costi risulta:

$$1 \cdot \left(\frac{m-n}{m}\right) + \left(1 + T(n-1, m-1)\right) \cdot \frac{n}{m} = 1 - \frac{n}{m} + \frac{n}{m} + \frac{n}{m} T(n-1, m-1)$$

($n-1$ elementi presenti; tabella di $m-1$ posizioni)

$$= 1 + \frac{n}{m} T(n-1, m-1)$$

e, quindi,

$$T(n, m) \leq \begin{cases} 1 & \text{se } n = 0 \\ 1 + T(n-1, m-1) \cdot \frac{n}{m} & \text{altrimenti.} \end{cases}$$

Proviamo per induzione che $T(n, m) \leq \frac{m}{m-n}$ per $m > n \geq 0$

Caso base: $n = 0$ $\quad T(0, m) \leq \frac{m}{m-0} = \frac{m}{m} = 1$ $\quad$ OK.

Ipotesi induttiva. $T(n-1, m-1) \leq \frac{m-1}{m-1-(n-1)} = \frac{m-1}{m-n}$.

Passo induttivo. $T(n, m) \leq 1 + \frac{n}{m} T(n-1, m-1)$

$\leq 1 + \frac{n}{m} \cdot \left(\frac{m-1}{m-n} \right)$ (Applicando l'ipotesi induttiva)

$\leq 1 + \frac{n}{m} \cdot \frac{m}{m-n}$ ($m > m-1$)

$\leq 1 + \frac{n}{m-n} = \frac{m-n+n}{m-n}$

$= \frac{m}{m-n}$ ∎

Pertanto, poiché $T(n, m) \leq \frac{m}{m-n}$ e $\frac{m}{m-n} = \frac{1}{1-\frac{n}{m}} = \frac{1}{1-\alpha} = (1-\alpha)^{-1}$,

nell'ipotesi che $(1-\alpha)^{-1} = O(1)$, allora

$$T(n, m) = O(1)$$

ovvero le operazioni hanno costo costante.

Nota che mantenendo l'invariante che m è circa il doppio di n, ovvero $m \geq 2n$ risulta $\frac{1}{1-\alpha} \leq \frac{1}{1-\frac{1}{2}} = 2$, cioè $(1-\alpha)^{-1} = O(1)$.

Nella pratica non possiamo generare una permutazione veramente casuale delle posizioni sondate con

$$hash[0](K),\ hash[1](K),\ \ldots$$

ed usiamo delle semplificazioni / approssimazioni.

- Scansione lineare : tecnica più facile da realizzare

$$hash[i](K) = (hash(K) + i) \,\%\, m$$

- Scansione quadratica : scelti a, b e c in modo opportuno

$$hash[i](K) = (hashK(K) + a \cdot i^2 + b \cdot i + c) \,\%\, m$$

Nota : a, b e c devono essere tali da far sì che $hash[i](K)$ generi tutte le m posizioni. Per esempio, scegliendo

a	b	c
↓	↓	↓
1	0	1

$i^2 + 1$

va bene !

a	b	c
↓	↓	↓
2	0	0

i^2

no ! genera solo posizioni pari.

- Scansione con hash doppio: usa due funzioni hash differenti

$$hash[i](K) = \left(hash(K) + i \times (1 + hash'(K))\right) \% m$$

Occorre essere certi che vengano scandite tutte le posizioni.
Ciò accade se $(1 + hash'(K))$ è primo rispetto a m,
ovvero l'unico divisore comune che hanno è 1.
Esistono diversi modi per farlo.

La scansione lineare e quella quadratica danno luogo al fenomeno del clustering. Per esempio, la scansione lineare genera le posizioni

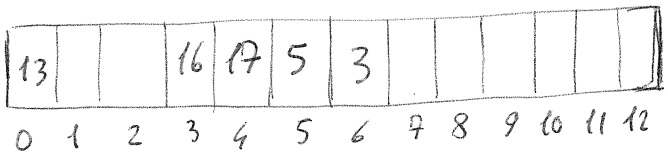
$$h = hash(K),\quad h+1,\quad h+2,\ \ldots$$

che producono aggregazioni (cluster) di elementi,
che occupano un segmento di posizioni contigue in tabella.

Una ricerca di un elemento che "ricade nel cluster" lo deve percorrere tutto. Per esempio:

Supponiamo di usare $hash(K) = K \% m$ dove $m = 13$

Inseriamo nella tabella vuota le chiavi 5, 13, 16 e 17



$(hash(K) + i) \% m$

$i = 0, 1, \ldots$

$hash(5) = 5 \% 13 = 5$

$hash(13) = 13 \% 13 = 0$

$hash(16) = 16 \% 13 = 3$

$hash(17) = 17 \% 13 = 4$

nessun problema.

Provando ad inserire $K = 3$

$hash(3) = 3 \% 13 = 3$

è risulta occupata

$(hash(3) + 1) \% m = (3+1) \% 13 = 4$ NO

$(hash(3) + 2) \% m = (3+2) \% 13 = 5$ NO

$(hash(3) + 3) \% m = (3+3) \% 13 = 6$ SI

l'elemento $K = 3$ viene allora inserito alla "fine del cluster", in posizione 6.

La scansione quadratica non migliora la situazione: il cluster si ripresenta soltanto in forma diversa seguendo la funzione quadratica.

Con l'hash doppio la situazione migliora.