

# Algoritmi e strutture dati

## Tabelle hash



# Tabelle hash

Una **tabella hash** è una struttura dati astratta. L'esigenza di questa struttura emerge in molti contesti molto diversi tra loro. Se immaginiamo di dover memorizzare oggetti, la scelta naturale potrebbe essere quella di utilizzare un array oppure utilizzare una lista. Nella sua versione più generale, il problema può essere descritto così: dato un numero di oggetti **relativamente piccolo**, ognuno dei quali è denotato da una chiave il cui universo è **relativamente grande**, trovare un modo efficiente di memorizzare in maniera dinamica questi oggetti e implementare le operazioni di inserimento, cancellazione, e ricerca. Nella nostra implementazione, le tabelle hash sono dinamiche, parzialmente compatte, e non basate sull'ordinamento.

# Tabelle hash

Le tabelle hash, e in generale la cosiddetta **indicizzazione hash**, che poi ha dato luogo a molti concetti su essa basati, sono state introdotte da Hans Peter Luhn, nel 1958.



# Tabelle hash

Tutte pos- avere un numero che potenzialmente molto grande

Le situazioni che richiedono una forma di indicizzazione possono emergere in molti contesti naturali. Tutte le volte che dobbiamo memorizzare oggetti complessi, dalle parole, ai testi, alle strutture multi-indice, solo per citarne alcuni, l'assegnamento di una chiave è complesso. Non è possibile costruire una tabella di assegnamento in maniera efficiente, ed i metodi impliciti generano chiavi molto grandi (vedremo degli esempi più avanti). La soluzione che prevede l'uso di un array (e si chiama **tabella hash ad accesso diretto**) ha ottime complessità:  $\Theta(1)$  per tutte le operazioni.

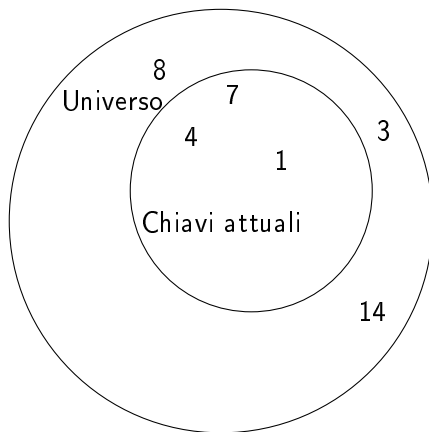
Purtoppo, la grandezza delle chiavi la rende inaccettabile nella maggioranza dei casi. La soluzione che prevede l'uso di una lista presenta  $\Theta(1)$  per l'inserimento, e  $\Theta(n)$  per ricerca e cancellazione, nei casi medio e pessimo. Ci proponiamo di trovare una soluzione più efficiente nel caso medio.

## Tabelle hash: notazione

Una tabella hash ad accesso diretto  $T$  è un semplice array di **puntatori** ad oggetti;  $T[key]$  punta all'oggetto la cui chiave assegnata è  $key$ . Diciamo che gli oggetti sono denotati con  $x, y, \dots$ , e che per ognuno di essi il campo  $key$  è la sua chiave. Per una tabella ad accesso diretto stiamo immaginando che:  $x.key$  è sempre **piccolo** (se chiamiamo  $m$  la dimensione della tabella, questo vuol dire  $x.key \leq m$  per ogni  $x$ ) e  $x.key \neq y.key$  per ogni coppia  $x \neq y$ . Purtroppo, nella realtà, per garantire queste proprietà bisogna usare valori di  $m$  troppo grandi. Per  $m$  molto grande, anche un'operazione elementare come **creare**  $T$  vuoto (che implica percorrere almeno una volta tutte le posizioni per portarle a nil) prende troppo tempo.

# Tabelle hash

Qual è una caratteristica comune a tutte le applicazioni tipiche in questo contesto? Che il numero di chiavi effettivamente utilizzate è molto inferiore alla cardinalità del dominio, che in questo contesto si chiama generalmente **universo**, e si denota con  $\mathcal{U}$ .



# Tabelle hash con chaining

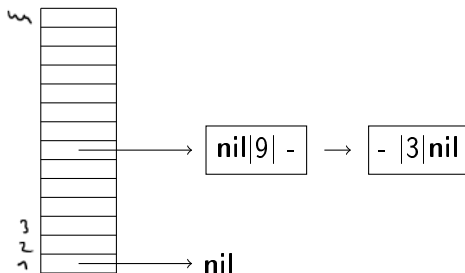
Se il numero effettivo  $n$  di elementi **effettivamente utilizzati** è molto più piccolo della cardinalità dell'universo, possiamo ancora implementare  $T$  con un array di posizioni  $1, \dots, m$ , ma nasce il problema di memorizzare una chiave  $k$  molto più grande di  $m$ , quindi senza accesso diretto. Per risolverlo creiamo una funzione

$$h : \mathcal{U} \rightarrow \{1, \dots, m\}$$

detta **funzione di hash**, per poter quindi indirizzare l'elemento  $k$  alla posizione  $h(k)$ . Ovviamente  $h$  non può essere iniettiva. Quando  $k_1 \neq k_2$  e si dà il caso che  $h(k_1) = h(k_2)$ , e chiameremo questa situazione di **conflitto**. Conflitti a parte, che adesso vedremo come risolvere, abbiamo già ottenuto un primo vantaggio:  $T$  è molto piccolo (ha esattamente  $m$  posizioni), e quindi diventa utilizzabile. Inoltre, il tempo di accesso è ancora costante **a meno dell'overload dovuto ad un conflitto**.

# Tabelle hash con chaining

Il nostro obiettivo è quello di progettare una funzione di hash che **minimizzi** i conflitti, ed una strategia per risolverli quando avvengono ugualmente. La tecnica chiamata **chaining** risolve i conflitti utilizzando una lista doppiamente collegata. La testa della lista è memorizzata in  $T[h(k)]$ , che quando è vuota contiene **nil**.





# Tabelle hash con chaining: operazioni

**proc HashInsert** ( $T, k$ )

$\left\{ \begin{array}{l} \text{let } x \text{ be a new node with key } k \\ i = h(k) \\ \text{ListInsert}(T[i], x) \end{array} \right.$

**proc HashSearch** ( $T, k$ )

$\left\{ \begin{array}{l} i = h(k) \\ \text{return ListSearch}(T[i], k) \end{array} \right.$

**proc HashDelete** ( $T, k$ )

$\left\{ \begin{array}{l} i = h(k) \\ x = \text{ListSearch}(T[i], k) \\ \text{ListDelete}(T[i], x) \end{array} \right.$

# Tabelle hash con chaining: complessità della ricerca

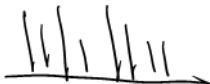
Queste operazioni sono chiaramente **corrette** e **terminanti**, e la loro **complessità** dipende dalla politica che utilizziamo sulle liste. Nel caso studiato da noi, l'inserimento è precisamente  $\Theta(1)$ . Anche nel caso dell'operazione di *HashDelete*, come nelle liste, la complessità è  $\Theta(1)$  più il costo di cercare il giusto nodo da cancellare. Ma quanto costa *HashSearch*? Fissiamo  $n$  come numero di elementi effettivamente inseriti in  $T$ , e  $m$  come dimensione di  $T$ . Se tutti gli  $m$  elementi finiscono nella stessa casella, la ricerca, e quindi anche la cancellazione, sono nel caso peggiore, e la loro complessità diventa  $\Theta(n)$ , che è una pessima prestazione. Non usiamo le tabelle hash per le loro prestazioni sul caso peggiore. Come trattiamo il caso medio?

# Tabelle hash con chaining: complessità della ricerca



Il caso medio dipende dalla bontà della funzione  $h$ , cioè da quanto bene essa distribuisce l'insieme delle chiavi sulle  $m$  posizioni. Sotto l'ipotesi di **hashing uniforme semplice**,  $h$  inserisce una chiave  $k$  in un determinato slot con la stessa probabilità con la quale la inserisce in qualsiasi altro slot, indipendentemente dalla presenza o meno di altre chiavi in  $T$ . Per ogni posizione  $j$  tra 1 e  $m$ , la lunghezza della lista  $T[j]$  ( $|T[j]|$ ) è tale che  $\sum_{j=1}^m |T[j]| = n$ , per cui il **valore atteso** di  $|T[j]|$  (in termini statistici:  $E[|T[j]|]$ ) è  $\frac{n}{m}$  (che viene detto **fattore di carico**). Quindi, sotto l'ipotesi di hashing uniforme, il caso medio di *HashSearch* su una tabella hash a collisioni risolte per chaining ed in caso di ricerca con esito **negativo** (cioè: la chiave non era mai stata inserita), ha complessità  $\Theta(1 + \frac{n}{m})$ , assumendo che il calcolo di  $h$  prenda tempo  $O(1)$ .

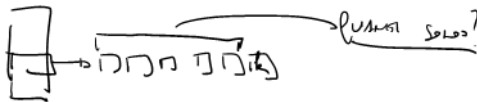
valore atteso

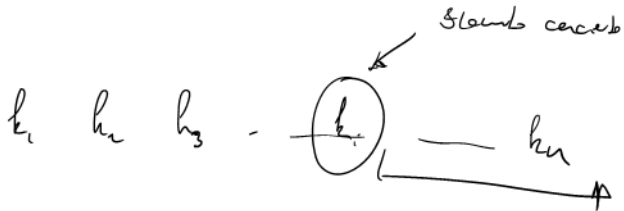
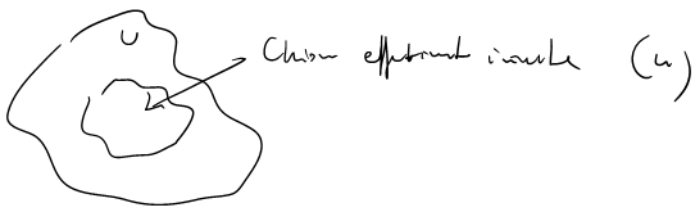


# Tabelle hash con chaining: complessità della ricerca

Ma cosa succede quando la ricerca ha esito positivo, cioè la chiave è **presente** in  $T$ ? La ragione principale per cui il calcolo del tempo medio di esecuzione di *HashSearch* in questo caso è più complesso rispetto al caso di esito negativo è che il tempo dipende dal **numero di elementi della lista per la posizione  $h(k)$**  che bisogna guardare prima di arrivare a  $k$  stesso.

Osserviamo che questo numero è esattamente il numero di elementi tali che, primo, hanno chiave  $k'$  dove  $h(k) = h(k')$ , e, secondo, sono stati inseriti **dopo** il momento in cui è stato inserito  $k$ . Infatti, le liste sono strutturate in modo tale che gli elementi nuovi appaiono prima (ricordiamo la procedura di inserimento in una lista), e quindi se cerco un elemento a chiave  $k$  nella lista puntata da  $h(k)$  tale che  $l$  elementi hanno chiave  $k'$  con  $h(k) = h(k')$  sono stati inseriti dopo  $k$ , dovrò guardare  $l$  elementi prima di avere successo.






(Shun smazzet Puzchisti)

# Tabelle hash con chaining: complessità della ricerca

L'elemento cercato con chiave  $k$  ha la stessa probabilità di essere uno degli  $n$  elementi correntemente in  $T$ . La probabilità che due chiavi siano indirizzate alla stessa posizione è  $Pr(h(k) = h(k'))$ , che, sotto l'ipotesi di hashing uniforme semplice, è precisamente  $\frac{1}{m}$ . Quindi, il valore atteso per la variabile probabilistica  $V$  **numero di elementi da esaminare per giungere a  $k$**  è:


$$E[V] = \frac{1}{n} \cdot (\sum_{i=1}^n (1 + \sum_{j=i+1}^n \frac{1}{m})).$$

Leggendo: fissato l' $i$ -esimo elemento, il numero di elementi da cercare per arrivarci è 1 più il numero di elementi prima di  $i$  sulla stessa lista, dato dalla somma delle probabilità che il  $j$ -esimo elemento (diverso da  $i$ ) sia presente, anch'esso, sulla lista. La media tra tutti i possibili  $i$ -esimi elementi, si ottiene l'espressione.

$$\bar{C}(V) = \frac{1}{n} \left( \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n \frac{1}{m} \right) \right)$$

$$= \frac{1}{n} \left( \sum_{i=1}^n \left( 1 + \frac{1}{m} \sum_{j=i+1}^n 1 \right) \right)$$

$$= \frac{1}{n} \left( \sum_{i=1}^n \left( 1 + \frac{1}{m} (n-i) \right) \right)$$

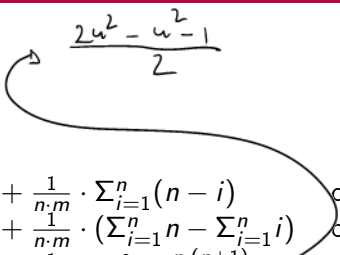
$$= \frac{1}{n} \sum_{i=1}^n 1 + \frac{1}{n} \sum_{i=1}^n \frac{1}{m} (n-i)$$

$$= \underbrace{\frac{1}{n} \sum_{i=1}^n 1}_{=1} + \frac{1}{n} \cdot \frac{1}{m} \sum_{i=1}^n (n-i)$$

$$= 1 + \frac{1}{nm} \sum_{i=1}^n (n-i)$$

# Tabelle hash con chaining: complessità della ricerca

Quindi:



A handwritten formula  $\frac{2u^2 - u^2 - 1}{2}$  is written at the top. A curved arrow originates from the third term of the summation in the equation below,  $(n^2 - \frac{n \cdot (n+1)}{2})$ , and points to the handwritten formula.

$$\begin{aligned} E[V] &= 1 + \frac{1}{n \cdot m} \cdot \sum_{i=1}^n (n - i) && \text{calcolo algebrico} \\ &= 1 + \frac{1}{n \cdot m} \cdot (\sum_{i=1}^n n - \sum_{i=1}^n i) && \text{calcolo algebrico} \\ &= 1 + \frac{1}{n \cdot m} \cdot (n^2 - \frac{n \cdot (n+1)}{2}) && \text{sommatoria} \\ &= 1 + \frac{\frac{n \cdot m}{n \cdot m}}{2 \cdot m} && \text{calcolo algebrico} \\ &= \Theta(1 + \frac{n}{m}) \end{aligned}$$

E quindi il tempo totale atteso per una ricerca che ha successo è  $\Theta(1 + \frac{n}{m})$ , come nel caso di esito negativo. Questo ci permette di concludere che il tempo medio per una ricerca è proprio  $\Theta(1 + \frac{n}{m})$ .



# Funzioni di hash per il chaining

Da una funzione di hash vorremmo che fosse uniforme semplice e che fosse computazionalmente semplice da calcolare. La letteratura ci offre varie possibilità che tengono conto delle diverse esigenze anche sul valore di  $m$ . Ne vediamo alcune: se abbiamo, ad esempio, chiavi naturali, con  $m$  numero primo, lontano da una potenza di 2, allora usiamo il metodo della **divisione** (modulo  $m$ ); se, invece, abbiamo chiavi naturali, con  $m = 2^p$  per qualche  $p$ , allora usiamo il metodo della **moltiplicazione**; infine, se abbiamo chiavi che sono stringhe, oppure insiemi di stringhe, o oggetti complessi, con  $m$  numero primo, lontano da una potenza di 2, possiamo usare il metodo dell'**addizione** (modulo  $m$ ) per poi tornare al primo caso.

# Funzioni di hash per il chaining: divisione

Nel caso di chiavi come numeri naturali, una soluzione molto intuitiva è usare il resto della divisione per  $m$ :

$$h(k) = k \bmod m + 1$$

Quando questa funzione è una (buona approssimazione di una) funzione di hash uniforme semplice? Quando si verifica che  $m$  è un numero primo ed lontano da una potenza di 2. Infatti, quando  $m$  non è un numero primo, allora i suoi divisori danno luogo a liste di chaining particolarmente lunghe. Inoltre, se  $m = 2^p$  per qualche  $p$ , allora  $k \bmod m$  dipende unicamente dagli ultimi  $p$  bits della rappresentazione binaria di  $k$ . Se le chiavi  $k$  sono uniformemente distribuite, nessuno di questi due problemi è realmente importante. Ma nella realtà le chiavi raramente sono uniformemente distribuite, e quindi usiamo questi accorgimenti per rimediare.

Given:  $f(k)$  is the number of nodes at level  $k$  in a binary tree of height  $n$ .  
 To prove:  $f(k) = 2^k \quad \forall k \in \{0, 1, \dots, n\}$

$\{52, 68, 94\}$

$$n = 4$$

$$f(k) = (k \text{ mod } n) + 1$$



$$f(52) = 1$$

$$f(68) = 1$$

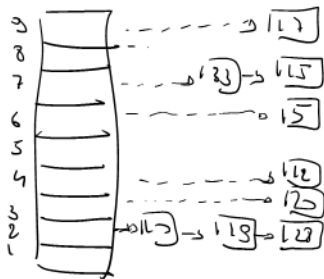
$$f(94) = 1$$

110100

1000100

1010100

Σ 55 ncrw 170



$$h(k) = (k \text{ mol } g)_{+1}$$

$$u = 3$$

$$h(5) = 6$$

$$h(28) = 2$$

$$h(18) = 2$$

$$h(15) = 7$$

$$h(25) = 3$$

$$h(33) = 7$$

$$h(12) = 4$$

$$h(17) = 9$$

$$h(10) = 2$$

10

# Funzioni di hash per il chaining: moltiplicazione

Per esempio quando si vuole  
e scegliere

Se non vogliamo che la scelta di  $m$  influenzi le prestazioni, quindi per esempio con  $m = 2^p$  per qualche  $p$ , possiamo usare il metodo della moltiplicazione. Scegliamo una costante  $A$  tra 0 e 1 reale, e costruiamo:

$$h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor + 1.$$

Questo metodo funziona bene per ogni costante di  $A$ , esistendo però certe costanti 'note' per cui la letteratura riporta ottimi comportamenti, come ad esempio  $A = \frac{\sqrt{5}-1}{2}$ .

## Funzioni di hash per il chaining: stringhe (addizione)



Come ci comportiamo invece quando le chiavi sono stringhe di un alfabeto  $\Sigma$ ? Una soluzione naturale consiste nell'interpretare ogni stringa come un numero naturale. Questo può essere ottenuto utilizzando la codifica ASCII, che assegna ad ogni carattere un numero univoco tra 0 e 127 (255 nel caso esteso); quindi una stringa diventa un numero in base 127 che viene trasformato in a base 10 (potenzialmente molto grande):

(97, 98, 55)

$$ab7 = 97 \cdot (127)^2 + 98 \cdot (127) + 55 = 1564513 + 12446 + 55 = 1577014$$

Effettuata la conversione, ci si può poi ridurre al caso dei numeri naturali. Ma è davvero efficiente questo calcolo?

unhp

097091055

# Funzioni di hash per il chaining: stringhe (addizione)

Trasformare una stringa in un numero intero è un'operazione che costa, utilizzando la regola di Horner,  $\Theta(d)$  dove  $d$  è il numero di caratteri della stringa. Infatti, data una stringa:

$$a_1 a_2 \dots a_d$$

e la cardinalità  $B$  di  $\Sigma$ , la trasformazione corrisponde a calcolare:

$$a_1 \cdot B^{d-1} + a_2 \cdot B^{d-2} + \dots + a_d \cdot B^0,$$

cioè:

$$((a_1 \cdot B + a_2) \cdot B + a_3) \cdot B + \dots a_d.$$

usare  
polinomio  
nell'  
sistema  
2b2

## Funzioni di hash per il chaining: stringhe (addizione)

Il problema che emerge nelle applicazioni reali è quello della dimensione dei numeri ottenuti. Infatti, questi devono essere sottoposti poi ad operazioni aritmetiche, quali appunto il modulo oppure il confronto. Queste ultime, però, non possono (semplicemente) essere effettuate quando i numeri hanno dimensione troppo grande, nè prendono, come abbiamo sempre assunto, costo  $\Theta(1)$  in questo caso. Nuovamente, l'aritmetica modulare ci viene incontro. Ricordiamo che l'operazione modulo è invariante rispetto all'addizione e, se  $m$  è primo, anche rispetto alla moltiplicazione. Allora, abbiamo che:

$$\begin{array}{lll} (a_i \cdot B + a_{i+1}) & \equiv_m & (z \cdot B + a_{i+1}) \quad \text{se e solo se} \\ (a_i \cdot B) & \equiv_m & (z \cdot B) \quad \text{se e solo se} \\ a_i & \equiv_m & z \end{array}$$

Poichè questa può essere generalizzata, ci dà un metodo semplice per calcolare  $h(h) = k \bmod m + 1$ , con  $k$  **molto grande**, senza, in effetti, dover mai memorizzare  $k$ .



# Funzione modulo, correttezza, e complessità

```
proc HashComputeModulo ( $w, B, m$ )  
  { let  $d = |w|$   
     $z_0 = 0$   
    for ( $i = 1$  to  $d$ )  $z_{i+1} = ((z_i \cdot B) + a_i) \bmod m$   
    return  $z_d + 1$ 
```

*HashComputeModulo* prende i parametri  $w = a_1 a_2 \dots a_d$  (parola qualsiasi di un alfabeto),  $B$  (base, cioè dimensione dell'alfabeto) e  $m$  (numero primo tale che  $(m + 1) \cdot B$  possa essere memorizzato in una parola di memoria). Evidentemente questa funzione **termina** sempre, ha **complessità**  $\Theta(d)$ , ed è **corretta** perchè utilizza le proprietà dell'aritmetica modulare per  $m$  numero primo. Quindi abbiamo un'altra ottima ragione per scegliere  $m$  primo!

# Tabelle hash: open hashing

Concludiamo adesso studiando una tecnica chiamata **open hashing** che ha le seguenti caratteristiche importanti: si eliminano le liste e quindi il chaining: una tabella hash di  $m$  elementi potrà ospitare al massimo  $m$  elementi, e, per ottenere questo risultato, si rinuncia (in pratica) ad implementare la funzione di cancellazione. L'indirizzamento aperto si basa sull'idea di provare più di una posizione sulla tabella, finché se ne trova una libera oppure si ha la certezza che la tabella è piena.

# Open hashing: operazioni

Data una chiave  $k$  da inserire in una tabella ad indirizzamento aperto, possiamo usare una funzione di hash qualsiasi tra quelle viste precedentemente, per ottenere una posizione  $h(k)$ . Se questa è libera, la chiave può essere inserita. Se invece la posizione è già occupata, proviamo **un'altra posizione**: questa sequenza di tentativi si chiama **sequenza di probing**. Non tutte le sequenze di probing funzionano bene: la condizione è che **tutte le posizioni della tabella devono essere provate** prima o poi. Questa condizione è chiamata **hashing uniforme** e generalizza la **condizione di hashing uniforme semplice**. Quindi, data una funzione di hashing qualsiasi  $h$ , definiamo una sequenza di probing come:

$$h(k, i)$$

che dipende dalla chiave ma anche dalla posizione **precedentemente tentata**.

# Open hashing: operazioni

**proc *OaHashInsert* ( $T, k$ )**

```
{  $i = 0$   
  repeat  
    {  $j = h(k, i)$   
      if  $T[j] = \text{nil}$   
        then  
          {  $T[j] = k$   
            return  $j$   
          }  
        else  $i = i + 1$   
      }  
    until  $(i = m)$   
  return "overflow"  
}
```

**proc *OaHashSearch* ( $T, k$ )**

```
{  $i = 0$   
  repeat  
    {  $j = h(k, i)$   
      if  $T[j] = k$   
        then return  $j$   
       $i = i + 1$   
    }  
  until  $((T[j] = \text{nil}) \text{ or } (i = m))$   
  return  $\text{nil}$   
}
```

# Open hashing: operazioni

Come si ottiene una sequenza di probing uniforme, partendo da una funzione di hashing uniforme semplice qualsiasi? Ci sono vari schemi, noi ne vediamo due molto semplici: **probing lineare** ed il **probing quadratico**. Nel primo caso la sequenza viene stabilita così:

$$h(k, i) = ((h'(k) + i) \bmod m) + 1$$

dove  $h'$  è qualunque funzione di hashing uniforme semplice. Si vede chiaramente che la proprietà di uniformità è rispettata per ogni chiave. Nel secondo caso, si scelgono due costanti  $c_1$  e  $c_2$ , e si impone:

$$h(k, i) = ((h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m) + 1,$$

ed anche in questo caso la sequenza è uniforme.

Non studiamo in maniera esplicita e formale la **correttezza**, **complessità** e **terminazione** delle due operazioni in questo caso. La letteratura ci dice che tendenzialmente l'indirizzamento aperto si comporta meglio del chaining, e che il probing quadratico si comporta meglio di quello lineare. In quanto alla funzione di cancellazione, succede che eliminare un elemento sostituendolo con **nil** può rendere scorretta l'operazione di ricerca, dovuto al probing. Pertanto, quando si utilizza indirizzamento aperto, l'eliminazione di un elemento tipicamente avviene in forma virtuale (cioè utilizzando un flag).

Sebbene ad una prima esposizione non sembri, le tabelle hash sono tra le strutture dati piú usate nella pratica. Ci sono moltissime varianti e miglioramenti delle versioni che noi abbiamo visto; a seconda dell'uso che uno vuole farne, è conveniente studiare queste varianti e vederne vantaggi e svantaggi.