Algoritmi e Strutture di Dati I

Massimo Franceschet

http://www.sci.unich.it/∼francesc

Tabelle hash

Una **tabella hash** (in inglese **hash table**) è una struttura di dati efficiente per implementare **dizionari**, cioè insiemi dinamici che supportano le operazioni di inserimento, cancellazione e ricerca.

Sotto ragionevoli assunzioni, le tabelle hash implementano queste tre operazioni con **complessità media costante** $\Theta(1)$.

Dunque, nel caso medio, la ricerca su tabelle hash è più efficiente della ricerca su liste ($\Theta(n)$) e di quella su alberi binari di ricerca ($\Theta(\log n)$).

Tabelle ad indirizzamento diretto

Supponiamo di voler rappresentare un insieme dinamico di oggetti dotati di un **campo chiave univoco**.

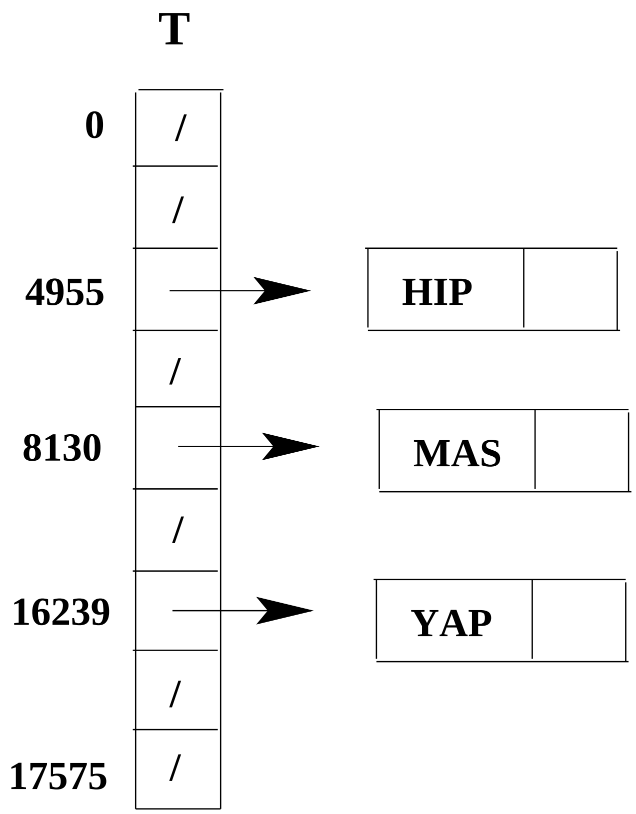
Sia $U = \{0, 1, \ldots, u-1\}$ l'**universo di tutte le possibili chiavi**. Se U non è molto grande, possiamo usare una **tabella ad indirizzamento diretto** (direct address table), cioè un vettore $T[0 \ldots u-1]$, per rappresentare un insieme dinamico di oggetto con chiavi in U.

Ogni posizione i del vettore T contiene un puntatore all'oggetto con chiave i, se tale oggetto è presente in tabella, oppure NIL, se tale oggetto non è presente in tabella.

Tabelle ad indirizzamento diretto

Supponiano che la chiave sia una parola di tre lettere dell'alfabeto inglese. Possiamo ordinare le chiavi in ordine lessicografico e convertire le chiavi alfabetiche in **chiavi numeriche** corrispondenti al numero d'ordine lessicografico.

Ad esempio $MAS = 12 \cdot 26^2 + 0 \cdot 26 + 18 = 8130$. Abbiamo $26^3 = 17576$ possibili chiavi, e quindi ci serve un vettore $T[0, \ldots, 17575]$ di circa 17 KB di memoria.


T
0
/
/
4955
HIP
/
8130
MAS
/
16239
YAP
/
17575
/

Operazioni

DirectAddressInsert(T,x)

1: $T[key[x]] \leftarrow x$

DirectAddressDelete(T,x)

1: $T[key[x]] \leftarrow$ NIL

DirectAddressSearch(T,k)

1: **return** $T[k]$

La complessità delle tre procedure è **costante**.

Inconvenienti

1. La dimensione della tabella è pari alla cardinalità u dello spazio U delle chiavi. Se u è molto grande, è costoso e talvolta impossibile creare un vettore di dimensione u.

2. Lo spazio allocato è indipendente dal numero di elementi effettivamente inseriti nel vettore. Questo porta ad uno spreco di memoria qualora il numero di elementi inseriti nella tabella sia di molto inferiore alla dimensione della tabella.

3. Infine, l'informazione sulla chiave degli oggetti è ridondante, in quanto corrisponde all'indice del vettore.

Esempio: Dizionario

Supponiamo di voler rappresenare un dizionario della lingua inglese mediante una tabella ad indirizzamento diretto.

Assumendo che una parola sia lunga al massimo 10 caratteri, abbiamo

$$26^{10} > (2^4)^{10} = 2^{40} = 1 \text{ TB}$$

possibili chiavi, quando un dizionario medio contiene $100000 = 100$ KB possibili voci.

Quindi il **fattore di carico** della tabella sarebbe $1/1000$.

Tabelle hash

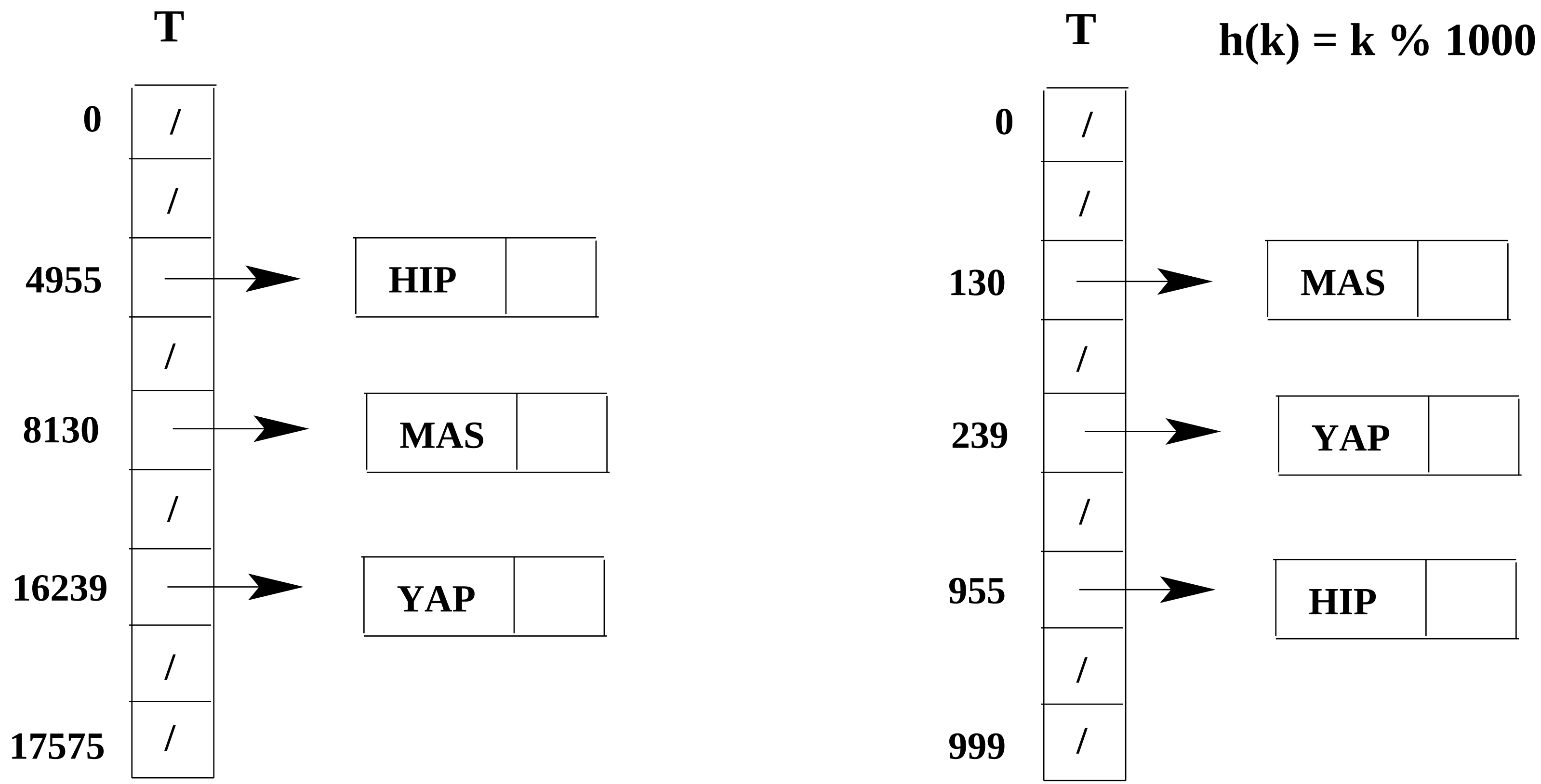
Sia $K \subseteq U$ l'insieme delle chiavi effettivamente contenute nel dizionario e siano $n = |K|$ e $u = |U|$.

Quando u è **molto grande** e n è modesto rispetto a u, è preferibile usare una tabella hash $T[0 \ldots m-1]$ di dimensione $m << u$ piuttosto di una tabella ad indirizzamento diretto di dimensione u.

Useremo una **funzione hash**

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

per mappare ogni chiave k nell'universo U in una posizione o **slot** $h(k)$ di T. Diremo che $h(k)$ è il **valore di hash** di k.


T
0
4955
8130
16239
17575
HIP
MAS
YAP
T
h(k) = k % 1000
0
130
239
955
999
MAS
YAP
HIP

Il problema delle collisioni

Di solito $m << u$. Dunque sono possibili **collisioni**, cioè chiavi diverse k_1 e k_2 tali che $h(k_1) = h(k_2)$.

Ci aspettiamo che le collisioni siano rare quando n (il numero di elementi inseriti in tabella) è piccolo rispetto a m (la dimensione della tabella), e diventino più frequenti quando n si avvicina a m. Quando $n \geq m$ le collisioni sono inevitabili.

Funzioni hash

Una **buona funzione hash** è una funzione che distribuisce le chiavi in modo uniforme sulle posizioni della tabella e quindi minimizza le collisioni quando possibile.

Il termine inglese **hash** significa *miscuglio, accozzaglia* e denota anche il nome che viene dato al *polpettone*, cioè al piatto di carne avanzata dal giorno prima e tritata assieme alle verdure.

Funzioni hash

Una buona funzione hash deve:

1. essere facile da calcolare (**costo costante**);
2. soddisfare il requisito di **uniformità semplice**: ogni chiave ha la medesima probabilità di vedersi assegnato un qualsiasi valore di hash ammissibile, indipendentemente dagli altri valori di hash già assegnati.

Il requisito di uniformità semplice è difficile da verificare perché in generale le chiavi non vengono estratte in modo casuale.
Solitamente ci si accontenta di funzioni che si avvicinano all'ipotesi di uniformità semplice.

Metodi per generare funzioni hash

Il **metodo della divisione** consiste nell'associare alla chiave k il valore di hash

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

Occorre evitare che m sia una potenza di 2. Se $m = 2^p$, allora $k \bmod m$ corrisponde ai p bit meno significativi di k. Questo limita la casualità della funzione hash, in quanto essa è funzione di una porzione (di dimensione logaritmica) della chiave. Una buona scelta è un numero primo non troppo vicino ad una potenza di due.

Metodi per generare funzioni hash

Il **metodo della moltiplicazione** consiste nell'associare alla chiave k il valore di hash

$$h(k) = \lfloor m \cdot (kA - \lfloor kA \rfloor) \rfloor,$$

ove $0 < A < 1$ è una costante.

Questo metodo ha il vantaggio che il valore di m non è critico. Possiamo scegliere m come una potenza di 2 per facilitare il calcolo della funzione hash. Per quanto riguarda il valore di A, viene suggerito in letteratura un valore prossimo a $(\sqrt{5} - 1)/2$.

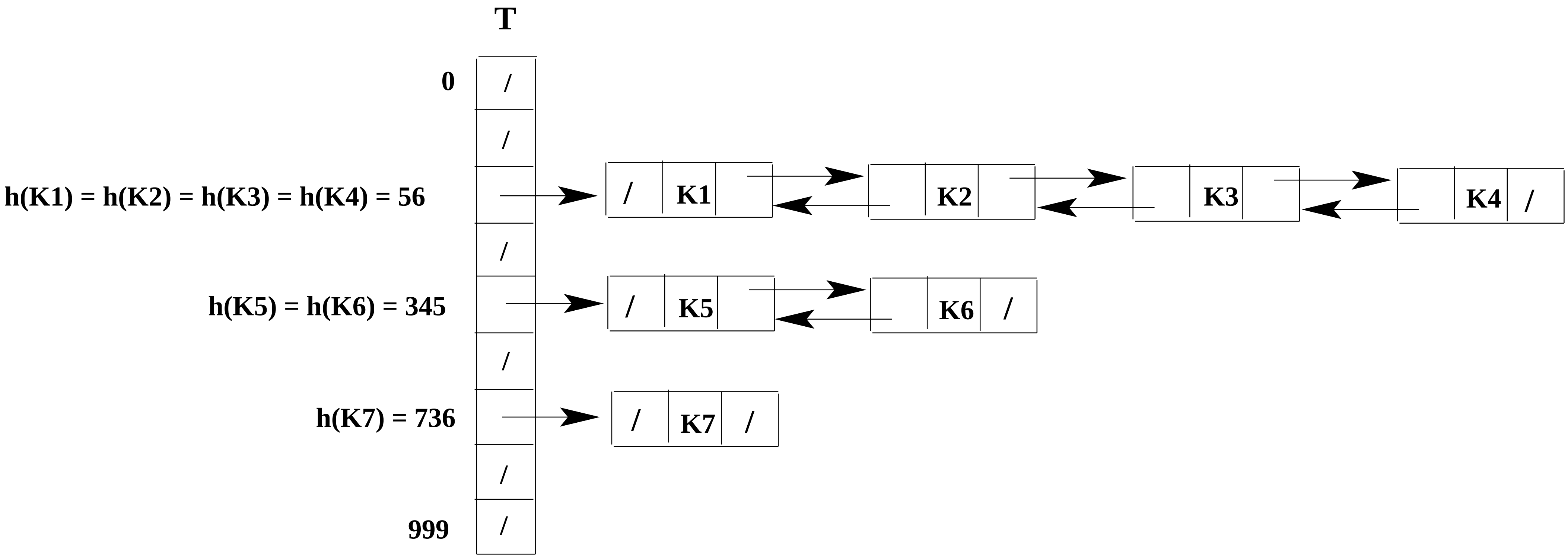
Metodi di risoluzione delle collisioni

Una buona funzione hash non elimina il **problema delle collisioni**. In particolare, quando $n \geq m$ (la tabella è piena), ogni nuovo inserimento genera certamente una collisione, qualsiasi sia la funzione hash usata.

Dove metto gli elementi che collidono? Vedremo due metodi per gestire le collisioni: **concatenazione** (*chaining*) e **indirizzamento aperto** (*open addressing*).

Risoluzione per concatenazione

Con questo metodo, una tabella hash $T[0 \ldots m-1]$ contiene in posizione i un puntatore alla testa di una **lista** di oggetti con valore di hash i.


T
0
/
/
h(K1) = h(K2) = h(K3) = h(K4) = 56
/
K1
K2
K3
K4
/
/
h(K5) = h(K6) = 345
/
K5
K6
/
/
h(K7) = 736
/
K7
/
/
999
/

ChainedHashInsert(T,x)

```
head ← T[h(key[x])]
next[x] ← head
prev[x] ← NIL
if head ≠ NIL then
    prev[head] ← x
end if
head ← x
```

ChainedHashDelete(T,x)

```
head ← T[h(key[x])]
if prev[x] ≠ NIL then
    next[prev[x]] ← next[x]
else
    head ← next[x]
end if
if next[x] ≠ NIL then
    prev[next[x]] ← prev[x]
end if
```

ChainedHashSearch(T,k)

$x \leftarrow T[h(k)]$
while $x \neq$ NIL **and** $key[x] \neq k$ **do**
    $x \leftarrow next[x]$
end while
return x

Complessità pessima

Supponiamo di usare una **buona funzione di hash**.

- **Operazioni di modifica**: hanno costo costante $\Theta(1)$.
- **Operazioni di ricerca**: nel caso pessimo, tutti gli inserimenti hanno generato una collisione e quindi sono stati concatenati nella stessa lista. La complessità pessima della ricerca è dunque $\Theta(n)$, ove n è il numero di elementi nella tabella.

Complessità media della ricerca

Definiamo il **fattore di carico** della tabella di hash come

$$\alpha = n/m$$

Si noti che $\alpha \geq 0$ è un numero *razionale*. Se la funzione hash h soddisfa l'ipotesi di uniformità semplice, α corrisponde alla **lunghezza media** di una qualsiasi lista di concatenazione.

Distinguiamo i casi di **ricerca con insuccesso** (cerchiamo una chiave non presente in tabella) e di **ricerca con successo** (cerchiamo una chiave presente in tabella).

In generale, ci aspettiamo che il costo medio di una ricerca con successo sia minore del costo di una ricerca con insuccesso.

Complessità media della ricerca con insuccesso

Teorema *In una tabella hash con risoluzione delle collisioni mediante concatenazione, adottando una buona funzione hash il costo medio della ricerca con insuccesso è* $\Theta(1+\alpha)$.

Dimostrazione

La ricerca con insuccesso consiste nel calcolo di un valore della funzione hash per individuare la lista di concatenazione in cui cercare e nella scansione esaustiva di tale lista.

Il costo del calcolo della funzione hash è $\Theta(1)$. La lunghezza media di una lista di concatenazione è α, e dunque il costo medio di una ricerca con insuccesso è pari a $\Theta(1+\alpha)$.

Complessità media della ricerca con successo

Teorema *In una tabella hash con risoluzione delle collisioni mediante concatenazione, adottando una buona funzione hash il costo medio della ricerca con successo è* $\Theta(1 + \alpha)$.

Dimostrazione

La ricerca con successo consiste nel calcolo di un valore della funzione hash per individuare la lista di concatenazione in cui cercare e nella scansione di tale lista fino a trovare l'elemento cercato. Mediamente dovrò scandire metà della lista prima di trovare l'elemento cercato.

Il costo del calcolo della funzione hash è $\Theta(1)$. La lunghezza media di una lista di concatenazione è α. Dunque il costo medio di una ricerca con successo è pari a $\Theta(1 + \alpha/2) = \Theta(1 + \alpha)$.

Complessità media della ricerca

Se

$$n = O(m)$$

allora

$$\alpha = n/m = O(m)/m = O(1)$$

e il costo medio della ricerca risulta

$$\Theta(1 + \alpha) = \Theta(1 + 1) = \Theta(1)$$

Risoluzione per indirizzamento aperto

Con questa tecnica, la collisione viene risolta utilizzando le posizioni libere della tabella hash, se ne esistono.

Se non ci sono posizioni libere, l'inserimento non è possibile.

Quindi tutti gli oggetti sono archiviati all'interno della tabella e il fattore di carico $\alpha \leq 1$.

Una funzione hash ora è una **funzione binaria**

$$h : U \times \{0, 1, \ldots m-1\} \rightarrow \{0, 1, \ldots m-1\}$$

tale che, per ogni chiave $k \in U$, la **sequenza di scansione**

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

è una **permutazione** di $\langle 0, 1, \ldots, m-1 \rangle$.

A esempio, se $m = 11$, una sequenza di scansione potrebbe essere:

$$\langle 0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 \rangle$$

Inserimento

L'inserimento di un nuovo elemento consiste nel visitare la sequenza di scansione

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

a partire dall'inizio fino a che una posizione libera viene trovata.

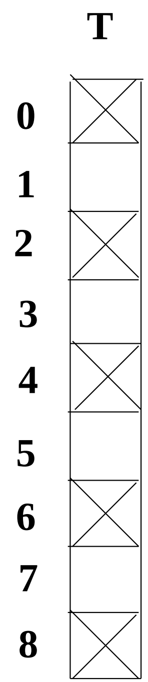
Se una posizione libera è stata trovata, il nuovo elemento verrà inserito in quella posizione. Altrimenti, l'inserimento non è possibile.

OpenAdressingInsert(T,x)

```
i ← 0
k ← key[x]
repeat
    j ← h(k, i)
    if T[j] = NIL then
        T[j] ← x
    else
        i ← i + 1
    end if
until i = m
error OVERFLOW
```

Supponiamo di voler inserire una chiave k con sequenza di scansione

$$\langle 0, 2, 4, 6, 8, 1, 3, 5, 7 \rangle$$



Ricerca

La ricerca di un oggetto consiste nel visitare la sequenza di scansione

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$$

a partire dall'inizio fino a che l'elemento cercato è stato trovato oppure una posizione libera è stata trovata.

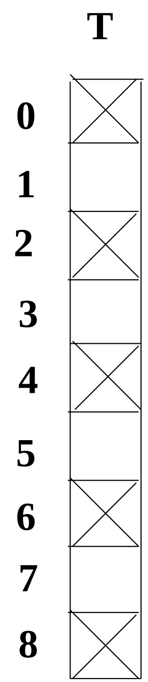
Infatti, per effetto della procedura di inserimento, l'elemento cercato non potrebbe essere stato inserito dopo la prima posizione libera.

OpenAdressingSearch(T,k)

$i \leftarrow 0$
repeat
  $j \leftarrow h(k, i)$
  if $key[T[j]] = k$ **then**
    return $T[j]$
  else
    $i \leftarrow i + 1$
  end if
until $T[j] =$ NIL **or** $i = m$
return NIL

Supponiamo di voler cercare una chiave k con sequenza di scansione

$$\langle 0, 2, 4, 6, 8, 1, 3, 5, 7 \rangle$$



Cancellazione

Per cancellare un elemento, non posso semplicemente mettere a NIL la corrispondente posizione nella tabella.

Se facessi in questo modo, la procedura di ricerca **non sarebbe corretta**, perché potrebbe terminare con insuccesso anche in presenza dell'elemento cercato.

Una soluzione è marcare un elemento cancellato con la costante DELETED diversa da NIL e modificare la procedura di inserimento in modo che tratti gli elementi cancellati (marchiati DELETED) come se fossero liberi sovrascrivendoli.

La procedura di ricerca rimane invariata, perché non distingue un elemento pieno da uno cancellato.

Esercizio *Si implementi l'operazione di cancellazione usando la costante* DELETED *per indicare che un elemento è stato cancellato. Si individui un caso in cui il fattore di carico è nullo ma la ricerca impiega tempo* $\Theta(m)$ *per cercare una qualsiasi chiave.*

Questa implementazione ha lo svantaggio che il tempo speso dalla ricerca non è più proporzionale al fattore di carico.

Buona funzione hash

Una buona funzione hash deve:

1. essere facile da calcolare (**costo costante**),
2. soddisfare il requisito di **uniformità**: ogni chiave ha la medesima probabilità di vedersi assegnata una qualsiasi delle $m!$ sequenze di scansione.

Complessità media della ricerca con insuccesso

Teorema *In una tabella hash con risoluzione delle collisioni mediante indirizzamento aperto e con fattore di carico* $\alpha < 1$*, adottando una buona funzione hash binaria il costo medio della ricerca con insuccesso è al più* $1/(1-\alpha)$.

Dimostrazione

Dato che il costo di calcolo della funzione hash è costante, è sufficiente contare il numero medio di ispezioni di posizioni della tabella hash in caso di ricerca con insuccesso.

Nell'ipotesi di uniformità, la probabilità di trovare una posizione occupata nella tabella hash è pari al fattore di carico $\alpha = n/m$.

Nel caso di ricerca con insuccesso, una ispezione viene sempre fatta. Ne faccio una seconda qualora la prima posizione ispezionata sia occupata, cioè con probabilità α. Ne faccio una terza qualora le prime due posizioni ispezionate siano occupate. Assumendo l'ipotesi di uniformità, questo avviene con probabilità $\alpha \cdot \alpha = \alpha^2$. E così via fino a quando tutta la tabella è stata ispezionata.

Dunque il numero medio di posizioni ispezionate sarà

$$1 + \alpha + \alpha^2 + \ldots \alpha^{m-1} \leq \sum_{i=0}^{\infty} \alpha^i = 1/(1 - \alpha).$$

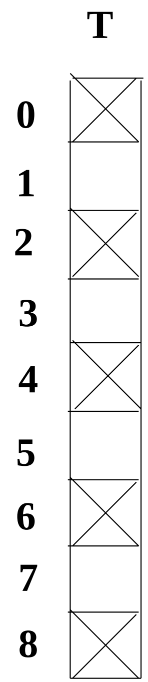
Complessità media della ricerca con successo

Teorema *In una tabella hash con risoluzione delle collisioni mediante indirizzamento aperto e con fattore di carico* $\alpha < 1$, *adottando una buona funzione hash binaria il costo medio della ricerca con successo è al più* $(1/\alpha)\ln(1/(1-\alpha))$.

Dimostrazione

Dato che il costo di calcolo della funzione hash è costante, è sufficiente contare il numero medio di ispezioni di posizioni della tabella hash in caso di ricerca con successo.

Il numero di ispezioni di una ricerca con successo della chiave k corrisponde al numero di ispezioni di una ricerca con insuccesso dopo che la chiave k è stata cancellata.



Sia β il fattore di carico della tabella priva della chiave k e sia α il fattore di carico della tabella con chiave k. Notate che β varia tra 0 e α.

Fissato β, il costo della ricerca con insuccesso è $1/(1-\beta)$. Il costo della ricerca con successo è la **media** del costo della ricerca con insuccesso al variare di β tra 0 e α.

Dato che β è un razionale e la funzione $1/(1-\beta)$ è crescente, tale media risulta essere minore o uguale a:

$$(1/\alpha)\int_0^\alpha 1/(1-\beta)d\beta = (1/\alpha)[\ln(1/(1-\beta))]_0^\alpha = (1/\alpha)\ln(1/(1-\alpha)).$$

Complessità media dell'inserimento

Infine, dato che un inserimento corrisponde ad una ricerca con insuccesso più una operazione di assegnamento, abbiamo il seguente corollario.

Corollario *In una tabella hash con risoluzione delle collisioni mediante indirizzamento aperto e con fattore di carico* $\alpha < 1$, *adottando una buona funzione hash binaria il costo medio dell'inserimento è al più* $1/(1-\alpha)$.

Si dimostra analiticamente che per ogni $0 < \alpha < 1$,

(I) $\frac{1}{1-\alpha} > 1 + \alpha$

(S) $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} > 1 + \frac{\alpha}{2}$

Dunque, in ogni caso, la complessità del metodo con indirizzamento aperto è maggiore della complessità del metodo con concatenazione. Le due complessità sono vicine se il fattore di carico è basso.

Inoltre, abbiamo una conferma del fatto che cercare con insuccesso costa mediamente più che cercare con successo. Infatti, per ogni $0 < \alpha < 1$:

$\frac{1}{1-\alpha} > \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

Generazione di funzioni hash

E' difficile ottenere buone funzioni hash, e quindi ci accontentiamo di approssimazione. Tre euristiche per generare funzioni hash sono:

1. **Scansione lineare**: data una funzione hash unaria $h' : U \rightarrow \{0, 1, \ldots m-1\}$, questo metodo definisce la funzione hash binaria come

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

2. **Scansione quadratica**: data una funzione hash unaria $h' : U \rightarrow \{0, 1, \ldots m-1\}$, e due costanti c_1 e $c_2 \neq 0$, questo metodo definisce la funzione hash binaria come

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

3. **Hashing doppio**: date due funzioni di hash unarie h_1 e h_2, questo metodo definisce la funzione hash binaria come

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m.$$

Un metodo per generare sequenze che sono permutazioni è scegliere m primo,

$$\begin{aligned} h_1(k) &= k \bmod m \\ h_2(k) &= 1 + (k \bmod (m-1)) \end{aligned}$$

Ad esempio, se $m = 11$ e $k = 1$, allora $h(k, i) = 1 + 2i$ e la sequenza di scansione associata a k è:

$$1, 3, 5, 7, 9, 11, 2, 4, 6, 8, 10$$

Tutti e tre i metodi fanno partire la sequenza di scansione di una chiave k da un valore determinato da una funzione hash ausiliaria.

La differenza consiste nel **passo** che determina la distanza tra due consecutive posizioni nella sequenza di scansione.

Nel primo caso il passo è lineare, nel secondo è quadratico, nel terzo è casuale.

Il metodo di scansione lineare soffre del **problema della accumulazione**: le posizioni occupate si accumulano in lunghi tratti contigui, aumentando il tempo medio di ricerca e di inserimento.

Il metodo di scansione quadratica soffre dello stesso problema ma in forma minore, in quanto lo spiazzamento è quadratico.

Il metodo di hashing doppio non ha questo problema perché il passo è casuale.

Queste considerazioni fanno preferire il metodo di hashing doppio ai primi due.