Algoritmi e Strutture Dati

Tabelle Hash

Maria Rita Di Berardini, Emanuela Merelli[1]

[1]Dipartimento di Matematica e Informatica
Università di Camerino

A.A. 2006/07

Tabelle

Una tabella è una sequenza di elementi E_i ciascuno dei quali è costituito da due parti:

- una chiave K_i, costituita da un gruppo di caratteri (o più generale di bit), che distingue un elemento dagli altri
- l'informazione I_i, associata alla chiave

Scopo della tabella è la memorizzazione delle informazioni I_i

Scopo della chiave è l'individuazione delle I_i dall'esterno

Esempi di tabelle: il tipo Dizionario

Dati: un insieme di coppie $(elem, chiave)$

Operazioni:

`insert`*(elem e, chiave k)*
aggiunge ad S una nuova coppia *(e, k)*

`delete`*(elem e, chiave k)*
cancella da S la coppia con chaive *k*

`search`*(chiave k)*
se la chiave *k* è presente in *s* restituisce l'elemento *e* ad esso associato altrimenti restituisce `null`

Operazioni su tabelle

1. **Ricerca** di un elemento E_i con chiave K_i
2. **Eliminazione** di un elemento presente in tabella
3. **Inserimento** di un elemento la cui chiave non risulti associata ad alcun elemento presente in tabella

Ricerca su tabelle

L'operazione fondamentale che si esegue su una tabella è la ricerca di un elemento nota la chiave

Per ogni elemento E_i della tabella si può definire la lunghezza di ricerca – denotata con S_i – come un numero di prove necessarie per raggiungere E_i (cioè numero di chiavi lette fino a trovare K_i)

La lunghezza media di ricerca S è la media delle S_i per tutti gli elementi della tabella

Come implementiamo una tabella

Esistono due possibili metodologie di implementazione

1. Tabelle ad indirizzamento diretto
2. Tabelle Hash

Parte I

Tabelle ad indirizzamento diretto

Tabelle ad indirizzamento diretto

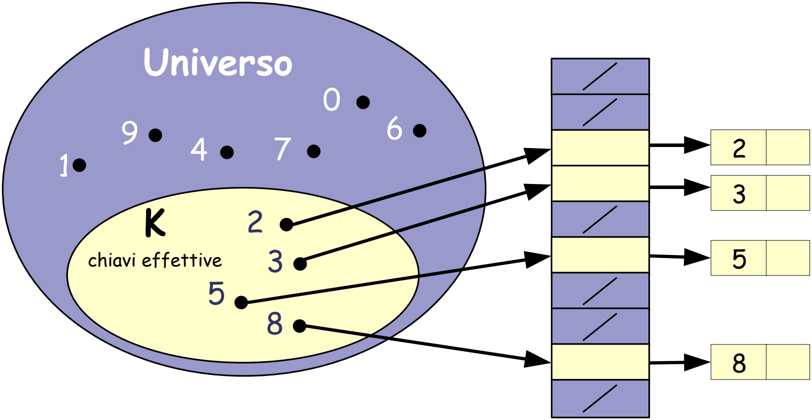
L'**indirizzamento diretto** è una tecnica semplice che funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo

Supponente di dover gestire un insieme dinamico i cui elementi abbiamo una chiave in $U = \{0, \ldots, m-1\}$; supponiamo inoltre che due elementi non possono avere la stessa chiave

Rappresentiamo l'insieme dinamico utilizzando un array, o tabella ad indirizzamento diretto, $T[0, \ldots, m-1]$, in cui ogni posizione (slot) corrisponde ad una chiave dell'universo U ($m = |U|$)

Abbiamo una corrispondenza biunivoca tra chiavi e posizioni della tabella: l'elemento con chiave k è memorizzato nella cella k

Tabelle ad indirizzamento diretto



Implementazione delle operazioni

Le operazioni di ricerca, inserimento e cancellazione sono facili da implementare ed hanno un costo computazionale costante $\Theta(1)$

Direct-Address-Search(T, k)
  return $T[k]$

Direct-Address-Insert(T, x)
  $T[key[x]] \leftarrow x$

Direct-Address-Delete(T, x)
  $T[key[x]] \leftarrow \text{NIL}$

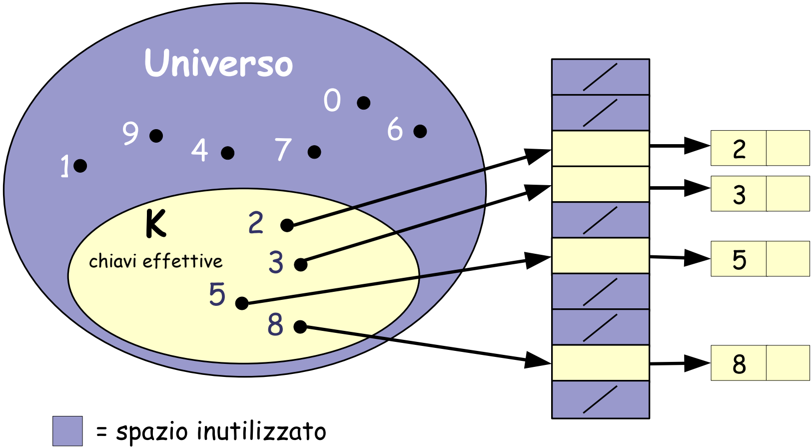
Inconvenienti

- La dimensione della tabella è data dalla cardinalità m dell'universo U delle chiavi; se U (e quindi m) è molto grande, implementare questa tabella può essere impraticabile, se non addirittura impossibile

- Lo spazio allocato è indipendente dal numero di elementi effettivamente memorizzati (le chiavi effettive)

- Se l'insieme K delle chiavi effettivamente utilizzate (dette anche effettive) è piccolo rispetto ad U la maggior parte dello spazio allocato per la tabella T sarebbe inutilizzato

Inconvenienti



Fattore di carico

Misuriamo il grado di riempimento di una tabella introducendo il fattore di carico:

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

dove $m = |U|$ (dimensione della tabella) e $n = |K|$ (numero di chiavi effettivamente utilizzate)

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

$$n = 100, m = 10^6, \alpha = 0,0001 = 0,01\%$$

Grande spreco di memoria!

Una possibile soluzione ...

Possiamo ridurre l'occupazione di spazio da $\Theta(|U|)$ a $\Theta(|K|)$ usando le liste collegate

Problema (non finiscono mai): Inserimento, Cancellazione e Ricerca costano $\Theta(|K|)$ invece di $\Theta(1)$

Un compromesso tra tempo e spazio

L'uso di tabelle (o tavole) hash consente di ottenere un buon compromesso tra tempo e memoria

Memoria richiesta: $\Theta(|K|)$

Tempo di ricerca: $\Theta(1)$, ma nel caso **medio** e non in quello pessimo

Parte II

Tabelle Hash

Tabelle Hash

Dimensionare la tabella in base al numero di elementi attesi ed utilizzare una speciale funzione (funzione hash) per indicizzare la tabella

Una funzione hash è una funzione che data una chiave $k \in U$ restituisce la posizione della tabella in cui l'elemento con chiave k viene memorizzato

$$h : U \mapsto [0, 1, \ldots m - 1]$$

N.B: la dimensione m della tabella può non coincidere con la $|U|$, anzi in generale $m < |U|$

Tabelle Hash

L'idea è quella di definire una funzione d'accesso che permetta di ottenere la posizione di un elemento in data la sua chiave

Con l'hashing, un elemento con chiave k viene memorizzato nella cella $h(k)$

Pro: riduciamo lo spazio necessario per memorizzare la tabella

Contro:

- perdiamo la corrispondenza tra chiavi e posizioni in tabella
- le tabelle hash possono soffrire del fenomeno delle **collisioni**

Collisioni

Due chiavi k_1 e k_2 **collidono** quando corrispondono alla stessa posizione della tabella, ossia quando $h(k_1) = h(k_2)$

Soluzione ideale: eliminare del tutto le collisioni scegliendo un'opportuna (= perfetta) funzione hash. Una funzione hash si dice **perfetta** se è iniettiva, cioè se per ogni $k_1, k_2 \in U$

$$k_1 \neq k_2 \Longrightarrow h(k_1) \neq h(k_2)$$

Deve essere $|U| \leq m$

Se $|U| > m$, evitare del tutto le collisioni è impossibile (Ad es. supponente di dover inserire un nuovo elemento in una tabella piena)

Risoluzione delle collisioni

Una possibile alternativa: utilizzare una buona funzione hash (per minimizzare le collisioni) e prevedere nel contempo dei metodi di risoluzione delle collisioni

Metodi classici di risoluzione delle collisioni:

1. Liste di collisione: gli elementi collidenti sono contenuti in liste esterne alla tabella; $T[i]$ punta alla lista di 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

Una buona funzione hash

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

hash[1] pl -es n

1. rifrittura, carne rifritta con cipolla, patate o altri vegetali
   hash browns (spec US) pasticcio fritto di patate lesse passate con cipolla
2. fiasco, pasticcio, guazzabuglio
   to make a hash of st (coll) pasticciare qs, far male qs, fare fiasco in qs
3. (fig) rifrittume
4. (spec radio) segnali parassiti
5. nella loc slang to settle sbs hash mettere in riga qn, zittire o sottomettere qn, sistemare o mettere a posto qn una volta per tutte
6. anche hash sign (tipog) il simbolo tipografico.

Una buona funzione hash

Una buona funzione hash deve:

1. essere facile da calcolare (costo costante)
2. soddisfare il requisito di uniformità semplice: ogni chiave deve avere la stessa probabilità di vedersi assegnata una qualsiasi posizione ammissibile, indipendentemente da altri valori hash già assegnati

   Sia $P(k)$ la probabilità che sia estratta una chiave k tale che $h(k) = j$, allora

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ per } j = 0, \ldots, m-1$$

Una buona funzione hash

Il requisito di uniformità semplice è difficile da verificare perchè raramente è nota la funzione di distribuzione di probabilità con cui vengono estratte le chiave (la funzione Pr)

Nella pratica però è possibile usare delle euristiche (metodo di approssimazione) per realizzare delle funzioni hash con buone prestazioni:

1. Metodo della divisione
2. Metodo della moltiplicazione

Metodo della divisione

Consiste nell'associare alla chiave k il valore hash

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

Semplice e veloce, ma occorre evitare certi valori di m; m non dovrebbe essere una potenza di 2

Se $m = 2^p$, $h(k)$ rappresenta solo i p bit meno significativi di k. Questo limita la casualità di h, in quanto è funzione di una porzione (di dimensione logaritmica) della chiave

Bisogna rendere la funzione h dipendente da tutti i bit della chiave; una buona scelta per m è un numero primo non troppo vicino ad una potenza di due

Metodo della moltiplicazione

Consiste nell'associare alla chiave k il valore hash

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

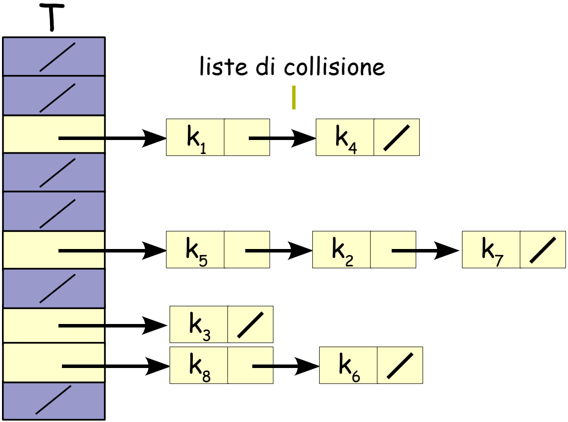
$kA - \lfloor kA \rfloor$ è la parte frazionaria di kA

Ha il vantaggio che il valore di m non è critico; di solito si sceglie $m = 2^p$

Per quanto riguarda il valore di A, in letteratura viene suggerito un valore prossimo a $(\sqrt{5} - 1)/2$

Risoluzione delle collisioni per concatenazione (chaining)

Gli elementi collidenti vengono inseriti nella stessa posizione della tabella in una lista concatenata



Implementazione delle operazioni

Chained-Hash-Search(T, k)
ricerca un elemento con chiave k nella lista $T[h(k)]$

Chained-Hash-Insert(T, x)
inserisci x in testa alla lista $T[h(key[x])]$

Chained-Hash-Delete(T, x)
cancella x dalla lista $T[h(key[x])]$

Analisi del costo delle operazioni

Chained-Hash-Search(T, k)

il tempo di esecuzione è, nel caso peggiore, proporzionale alla lunghezza della lista

Chained-Hash-Insert(T, x)

il tempo di esecuzione nel caso peggiore è $O(1)$

Chained-Hash-Delete(T, x)

se si usano liste bidirezionali, può richiedere un tempo $O(1)$
con liste semplici, richiede lo stesso tempo della ricerca

Costo della ricerca: analisi nel caso peggiore

Data una tabella T con m posizioni ed n elementi, quanto tempo richiede la ricerca di un elemento data la sua chiave?

Caso peggiore: tutte le chiavi vengono inserite nella stessa posizione della tabella creando un'unica lista di collisione di lunghezza n

In questo caso il tempo di ricerca è $\Theta(n)$ (ossia il costo della ricerca nella lista di collisione) $+$ il tempo di calcolo di h

Costo della ricerca: analisi del caso medio

Si definisce **fattore di carico** il rapporto tra il numero n degli elementi memorizzati e la dimensione m della tabella

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

Nell'ipotesi di uniformità semplice della funzione hash α è il numero medio di elementi memorizzati in ogni lista concatenata

$\alpha < 1$ molte posizioni disponibili rispetto agli elementi memorizzati
$\alpha = 1$ numero di elementi memorizzati è uguale alla dimensione della tabella
$\alpha > 1$ situazione attesa: molti elementi memorizzati rispetto alla dimensione della tabella

Analisi nel caso medio

Il comportamento nel caso medio dipende da come la funzione hash distribuisce le chiavi sulle m posizioni della tabella

Ipotesi:

- **uniformità semplice della funzione di hash**
- $h(k)$ è calcolata in $O(1)$
  così che il costo della ricerca di un elemento con chiave k dipenda esclusivamente dalla lunghezza della lista $T[h(k)]$

Costo della ricerca: analisi del caso medio

Teorema (ricerca senza successo): in una tabella hash in cui le collisioni sono risolte mediante liste di collisione, nell'ipotesi di uniformità semplice della funzione hash, una ricerca senza successo richiede in media un tempo $\Theta(1 + \alpha)$

Proof:
Occorre O(1) tempo per il calcolo della funzione $h(k)$; inoltre si deve scorrere completamente la lista $T[h(k)]$.

Poichè la lunghezza media di tale lista è α, abbiamo un costo medio totale di $\Theta(1 + \alpha)$ tempo

Analisi nel caso medio

Teorema (ricerca con successo): in una tabella hash in cui le collisioni sono risolte mediante liste di collisione, nell'ipotesi di uniformità semplice della funzione hash, una ricerca con successo richiede in media un tempo $\Theta(1 + \alpha)$

Proof: Dato $j = 1, \ldots, n$, sia T_j la tabella T contenente solo le prime j chiavi $k_1, \ldots, k_j$. Sia, inoltre, $\alpha_j = j/m$ il fattore di carico di T_j

La ricerca con successo della chiave $k = k_i$ in T ha lo stesso costo della ricerca senza successo di k in T_{i-1} più uno

Analisi nel caso medio

La ricerca con successo della chiave $k = k_i$ in T ha lo stesso costo della ricerca senza successo di k in T_{i-1} più uno: infatti, le chiavi esaminate nella lista $T[h(k)]$ sono esattamente le chiavi esaminate nella lista $T_{i-1}[h(k)]$ più uno (cioè la chiave k)

Quindi, il costo della ricerca con successo di $k = k_i$ è dato da

$$C_i = 1 + \alpha_{i-1} = 1 + \frac{i-1}{m}$$

ed il costo medio totale è

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

Analisi nel caso medio

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

$$\frac{1}{n}\left(n+\frac{1}{m}\sum_{i=1}^{n}(i-1)\right) = \frac{1}{n}\left(n+\frac{1}{m}\sum_{i=0}^{n-1}i\right) =$$

$$\frac{1}{n}\left(n+\frac{1}{m}\frac{n(n-1)}{2}\right) = 1+\frac{n-1}{2m} =$$

$$1+\frac{\alpha}{2}-\frac{1}{2m} = \Theta(1+\alpha)$$

Costo della ricerca: analisi del caso medio

Teorema (ricerca senza successo): in una tabella hash in cui le collisioni sono risolte mediante liste di collisione, nell'ipotesi di uniformità semplice della funzione hash, una ricerca senza successo richiede in media un tempo $\Theta(1+\alpha)$

Teorema (ricerca con successo): in una tabella hash in cui le collisioni sono risolte mediante liste di collisione, nell'ipotesi di uniformità semplice della funzione hash, una ricerca con successo richiede in media un tempo $\Theta(1+\alpha)$

Se n è proporzionale ad m, cioè se $n = O(m)$, allora

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

e il costo medio della ricerca risulta

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

Indirizzamento Aperto

La rappresentazione non fa uso di puntatori

Le collisioni vengono gestite memorizzando elementi collidenti in altre posizione della tabella

Invece di seguire le liste di collisione, calcoliamo la sequenza di posizioni da esaminare

Il fattore di carico non può mai superare 1

Si usa meno memoria rispetto alla rappresentazione con liste di collisione perchè non ci sono puntatori

Indirizzamento Aperto

Prevede che si usi solo lo spazio della tabella, senza uso di zone di trabocco, allocando gli elementi che determinano collisioni in posizioni diverse da quella che loro competerebbe

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

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

$$\begin{array}{llll} c(k,0) & c(k,1) & \ldots & c(k,m) \\ c(k,0) = h(k) & c(k,1) = h(k)+1 & \ldots & c(k,m) = h(k)+m \end{array}$$

Indirizzamento Aperto

Per inserire una nuova chiave si esamina una successione di posizioni della tabella, si esegue una **scansione**, finchè non si trova una posizione vuota in cui inserire la chiave

La sequenza di posizioni esaminate dipende dalla chiave che deve essere inserita

Estendiamo la funzione hash in modo che possa tener conto anche del numero di posizioni già esaminate

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

Operazione di Inserimento

Hash-Insert(T, k)
$i \leftarrow 0$
repeat
$j \leftarrow h(k, i)$
if $T[j] = \text{NIL}$ or $T[j] = \text{DELETED}$
then $T[j] = k$ **return** j
else $i \leftarrow i + 1$
until $i = m$
error "overflow sulla tabella hash"

Operazione di Ricerca

Hash-Search(T, k)
  $i \leftarrow 0$
  repeat
    $j \leftarrow h(k, i)$
    if $T[j] = k$ **return** j
    $i \leftarrow i + 1$
  until $T[j] = \text{NIL}$ or $i = m$
  return NIL

Operazione di Cancellazione

Il problema della cancellazione di un elemento in una tabella hash ad indirizzamento aperto è appena un pò più complesso

Se eliminiamo un elemento da una posizione *i* della tabella non si può semplicemente marcare questa posizione con NIL: spezzeremo la sequenza di scansione delle eventuali chiavi collidenti con la chiave da cancellare

Possiamo marcare la posizione con un valore speciale, DELETED, e modificare consistentemente la procedura **Hash-Insert**

Così facendo i tempi di ricerca non dipendo più solo dal fattore di carico α; l'uso di liste di collisione è più comune se si ammettono cancellazioni frequenti

Analisi del costo di scansione

Il costo viene espresso in termini del fattore di carico $\alpha = n/m$ dove n è il numero di elementi presenti nella tabella ed m è la dimensione della tabella

Nel caso dell'indirizzamento aperto, $n \leq m$ e quindi $\alpha \leq 1$

Ipotesi: viene usata una funzione hash uniforme; data una chiave k, la sequenza di scansione

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

è in modo equamente probabile, una qualunque permutazione di $\langle 0, 1 \ldots, m-1 \rangle$

Ricerca senza successo

Teorema (ricerca senza successo): in una tabella hash ad indirizzamento aperto con fattore di carico $\alpha = n/m < 1$, il numero medio di accessi per una ricerca senza successo è al più

$$\frac{1}{(1-\alpha)}$$

assumendo l'uniformità della funzione hash

Ricerca senza successo – dimostrazione

In una ricerca senza successo ogni accesso, tranne l'ultimo, è un accesso a una posizione occupata

Sia $i = 0, 1, 2, \ldots$ denotiamo con

$$p_i = Pr\{\text{esattamente } i \text{ accessi a posizioni occupate}\}$$

Abbiamo al più n posizioni occupate; quindi, se $i > n$, $p_i = 0$

Il numero medio di accessi è

$$1 + \sum_{i=0}^{\infty} i \cdot p_i$$

Ricerca senza successo – dimostrazione

Fact

$$\sum_{i=0}^{\infty} i \cdot Pr\{X = i\} = \sum_{i=1}^{\infty} Pr\{X \geq i\}$$

Se denotiamo con

$$q_i = Pr\{\text{almeno } i \text{ accessi a posizioni occupate}\}$$

allora

$$\sum_{i=0}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i \cdot q_i$$

Quanto vale q_i per $i \geq 1$?

Ricerca senza successo – dimostrazione

q_1 è la probabilità che la prima posizione che esaminiamo sia occupata, quindi

$$q_1 = \frac{n}{m} \leq \alpha$$

q_2 è la probabilità che le prime due posizioni che esaminiamo siano occupate, quindi

$$q_2 = \Big(\frac{n}{m}\Big)\Big(\frac{n-1}{m-1}\Big) \leq \alpha^2$$

Per $i = 1, \ldots, n$, q_i è la probabilità che le prime i posizioni che esaminiamo siano occupate, quindi

$$q_i = \Big(\frac{n}{m}\Big)\Big(\frac{n-1}{m-1}\Big) \ldots \Big(\frac{n-(i-1)}{m-(i-1)}\Big) \leq \alpha^i$$

Ricerca senza successo – dimostrazione

Infine, per ogni $i > n$ abbiamo che $q_i = 0 \leq \alpha^i$

Ricapitolando, Il numero medio di accessi è

$$1 + \sum_{i=0}^{\infty} i \cdot p_i = 1 + \sum_{i=1}^{\infty} q_i \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

Inserimento

Teorema (inserimento): l'inserimento in una tabella hash ad indirizzamento aperto con fattore di carico α, richiede un numero medio di accessi pari al più a

$$\frac{1}{(1-\alpha)}$$

assumendo l'uniformità della funzione hash

Inserimento – dimostrazione

Un elemento viene inserito solo se vi è spazio nella tabella e quindi se $\alpha < 1$

L'inserimento richiede una ricerca senza successo seguita dalla memorizzazione della chiave nell'ultima posizione vuota

Il numero medio di accessi è

$$\frac{1}{(1-\alpha)}$$

Ricerca con successo

Teorema (ricerca con successo): in una tabella hash ad indirizzamento aperto con fattore di carico $\alpha = n/m < 1$, il numero medio di accessi per una ricerca con successo è al più

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

assumendo l'uniformità della funzione hash

Nota: $\ln = \log_e$ è il logaritmo naturale

Ricerca con successo – dimostrazione

La ricerca con successo di una chiave k segue la stessa sequenza di scansione seguita per l'inserimento della stessa chiave k

Se k è la $(i+1)$ chiave inserita nella tabella hash, il numero medio di accessi per l'inserimento/ricerca con successo è

$$\frac{1}{(1-\alpha_i)} = \frac{1}{(1-i/m)} = \frac{1}{(m-i/m)} = \frac{m}{(m-i)}$$

Eseguendo la media su tutte le n chiavi nella tabella hash si ottiene un numero medio di accessi

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{(m-i)} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{(m-i)} = \frac{1}{\alpha}\sum_{i=0}^{n-1}\frac{1}{(m-i)}$$

Ricerca con successo – dimostrazione

$$\sum_{i=0}^{n-1} \frac{1}{(m-i)} = \sum_{k=m-n+1}^{m} \frac{1}{k} \qquad \text{(ponendo } k = m - i\text{)}$$

$$\sum_{k=m-n+1}^{m} \frac{1}{k} \leq \int_{n}^{m-n} \frac{1}{x} dx = \ln m - \ln(m-n) = \ln(\frac{m}{m-n}) = \ln(\frac{1}{1-\alpha})$$

Ricapitolando, il numero medio di accessi

$$\frac{1}{\alpha} \sum_{i=0}^{n-1} \frac{1}{(m-i)} \leq \frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$$

Parte III

Indirizzamento aperto: tecniche di scansione

Scansione Lineare

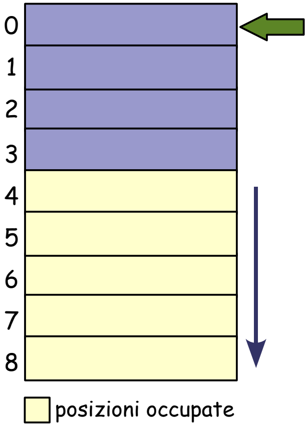
Sia $h' : U \longrightarrow \{0, 1, \ldots, m-1\}$ una funzione hash "ordinaria"

Il metodo di scansione lineare usa la funzione hash (estesa) definita come

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

$h(k,0) = h'(k) \bmod m$, $h(k,1) = (h'(k)+1) \bmod m$,
$h(k,2) = (h'(k)+2) \bmod m$, ...

Scansione Lineare



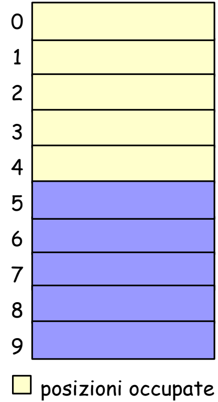
Hash-Insert(T,k) con h′(k) = 4

$h(k,0) = (h'(k) + 0) \bmod 9 = 4 \bmod 9 = 4$
$h(k,1) = (h'(k) + 1) \bmod 9 = 5 \bmod 9 = 5$
$h(k,2) = (h'(k) + 2) \bmod 9 = 6 \bmod 9 = 6$
$h(k,3) = (h'(k) + 3) \bmod 9 = 7 \bmod 9 = 7$
$h(k,4) = (h'(k) + 2) \bmod 9 = 8 \bmod 9 = 8$
$h(k,5) = (h'(k) + 3) \bmod 9 = 9 \bmod 9 = 0$

Scansione Lineare

La scansione lineare presenta un fenomeno conosciuto come **agglomerazione primaria**

Le posizioni occupate della tabella si accumulano per lunghi tratti, aumentando così il tempo medio di ricerca

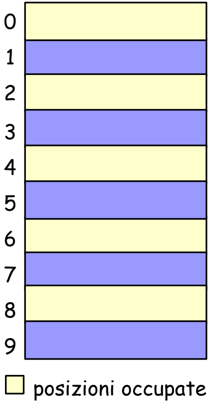


Calcoliamo il numero medio di accessi per una ricerca senza successo

$$\sum_{i=0}^{9} \#\text{acc}[h'(k) = i] \cdot \overbrace{Pr\{h'(k) = i\}}^{=\frac{1}{10}}$$

$h'(k)$	#acc
0	6
1	5
2	4
3	3
4	2
5–9	1

$$\frac{1}{10}\sum_{i=0}^{9} \#\text{acc}[h'(k) = i] = \frac{1}{10} \cdot 25 = 2.5$$



Di nuovo, il numero medio di accessi è

$$\sum_{i=0}^{9} \#\text{acc}[h'(k) = i] \cdot \overbrace{Pr\{h'(k) = i\}}^{=\frac{1}{10}}$$

$h'(k)$	#acc
pari	2
dispari	1

$$\frac{1}{10}((2 \cdot 5) + (1 \cdot 5)) = 1.5$$

Inoltre ...

La prima posizione esaminata determina l'intera sequenza di scansione; quindi abbiamo solo m sequenze di scansione distinte

Il numero ottimo è $m!$ ed è dato dall'ipotesi di unformità della funzione hash: ognuna delle $m!$ permutazioni di $\langle 0, \ldots, m-1 \rangle$ è equiprobabile

Siamo molto lontani dal numero ottimo

Scansione Quadratica

Sia $h' : U \rightarrow \{0, 1, \ldots, m-1\}$ una funzione hash "ordinaria"

Il metodo di scansione quadratica usa la funzione hash (estesa) definita come

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

dove, c_1 e c_2 sono delle costanti ausiliarie (con $c_2 \neq 0$)

Scansione Quadratica

Un esempio:

$$h(k,i) = \left(h'(k) + i + i^2\right) \bmod m$$

dove, $c_1 = c_2 = 1$

$h(k,0) = h'(k),$
$h(k,1) = h'(k) + 1 + 1 = h'(k) + 2,$
$h(k,2) = h'(k) + 2 + 4 = h'(k) + 6,$
$h(k,3) = h'(k) + 3 + 9 = h'(k) + 12,$
$h(k,4) = h'(k) + 4 + 16 = h'(k) + 20$

Cosa succede se $m = 20$? Viene scandita solo una porzione (in realtà $1/4$) della tabella

Scansione Quadratica

Elimina il problema dell'agglomerazione primaria, ma ...

(1) viene usata l'intera tabella; solo per alcune combinazioni di c_1, c_2 ed m; se $m = 2^p$ una buona scelta è $c_1 = c_2 = 1/2$, perchè i valori $h(k, i)$ per $i \in [0, m-1]$ sono tutti distinti

(2) $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$
questo porta ad una forma di addensamento (più lieve rispetto a quella primaria) detta **agglomerazione secondaria**

(3) di nuovo, la prima posizione determina l'intera sequenza di scansione ed abbiamo solo m sequenze di scansione distinte

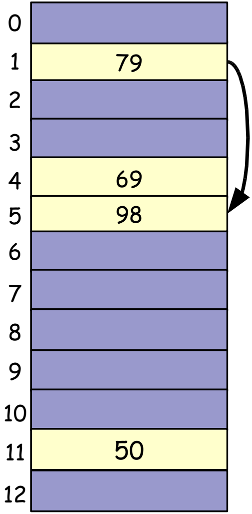
Hashing Doppio

L'**hashing doppio** usa una funzione hash (estesa) della forma

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

dove h_1, h_2 sono delle funzioni hash (ordinarie) ausiliarie

Hashing Doppio



$k = 14$

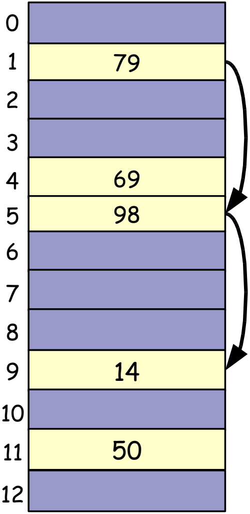
$h_1(k) = k \bmod 13 = 1$

$h_2(k) = 1 + (k \bmod 11) = 4$

$h(k,0) = 1 \bmod 13 = 1$

$h(k,1) = (1 + 1 \cdot 4) \bmod 13 = 5$

Hashing Doppio



$k = 14$

$h_1(k) = k \bmod 13 = 1$

$h_2(k) = 1 + (k \bmod 11) = 4$

$h(k,0) = 1 \bmod 13 = 1$

$h(k,1) = (1 + 1 \cdot 4) \bmod 13 = 5$

$h(k,1) = (1 + 2 \cdot 4) \bmod 13 = 9$

Hashing Doppio

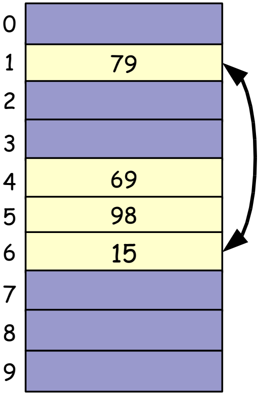
La prima posizione esaminata è $T[h_1(k)] \bmod m$; ogni posizione esaminata successivamente è distanziata dalla precedente di una quantità $h_2(k) \bmod m$

La sequenza di scansione dipende da k in due modi: a seconda della chiave, possono variare sia la posizione iniziale che il passo

L'hashing doppio non soffre di fenomeni di agglomerazione perchè il passo è casuale inoltre ...

Ogni possibile coppia $(h_1(k), h_2(k))$ produce una sequenza di scansione distinta: abbiamo $O(m^2)$ sequenze di scansione distinte ed in questo senso è migliore sia della scansione lineare che quadratica

Hashing Doppio



$K = 14$

$h_1(k) = k \bmod 13 = 1$

$h_2(k) = 1 + (k \bmod 10) = 5$

$h_2(k)$ non è primo con m

$h(k,0) = (1 + 0 \cdot 5) \bmod 10 = 1$

$h(k,1) = (1 + 1 \cdot 5) \bmod 10 = 6$

$h(k,2) = (1 + 2 \cdot 5) \bmod 10 = 1$

$h(k,3) = (1 + 3 \cdot 5) \bmod 10 = 6$

$h(k,4) = (1 + 4 \cdot 5) \bmod 10 = 1$

....

Hashing Doppio

Il valore di $h_2(k)$ deve essere primo con la dimensione m della tabella per ogni chiave da cercare

Infatti se $\mathrm{MCD}(h_2(k), m) = d > 1$ per qualche k, allora la ricerca di tale chiave andrebbe ad esaminare solo una porzione $(1/d)$ della tabella

Se m è una potenza di 2, basta definire h_2 in maniera tale che restituisca sempre un numero dispari

Altro modo è scegliere m primo e porre $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$ dove m' è appena più piccolo di m (ad esempio $m' = m - 1$ oppure $m' = m - 2$)