Dizionario

Insieme dinamico che offre solo le seguenti operazioni:

- inserimento di un elemento dato
- cancellazione di un elemento dato
- ricerca di un elemento dato (verifica dell'appartenenza di un elemento all'insieme)

Esempio di dizionario: tabella dei simboli mantenuta dal compilatore di un linguaggio di programmazione, in cui le chiavi degli elementi sono stringhe di caratteri corrispondenti agli identificatori del linguaggio

Tabella hash: conclusioni anticipate

- Struttura di dati efficace per realizzare dizionari
- Generalizzazione del concetto di array ordinario
- Alternativa efficace all'indirizzamento diretto di un array quando il n° n di chiavi effettivamente memorizzate è piccolo rispetto al n° totale di possibili chiavi (efficace perché usa un array di dimensione proporzionale a n)
- Invece di usare l'indice per indirizzare direttamente l'array, calcola l'indice usando la chiave

Tabella a indirizzamento diretto

Funziona bene se l'universo delle chiavi U = {0, 1, …, m – 1} è ragionevolmente piccolo

Assunzione: nessuna coppia di elementi ha la stessa chiave

Si realizza mediante un array T[0 .. m – 1] dove ogni posizione, o slot, corrisponde a una chiave appartenente a U ed è l'elemento che ha tale chiave o un puntatore allo stesso

T[k] = NIL $\leftrightarrow$ l'insieme non contiene elementi con chiave k

(Spesso) non è necessario memorizzare la chiave dell'oggetto perché coincide con l'indice del corrispondente elemento della tabella

Tabella a indirizzamento diretto (cont.)

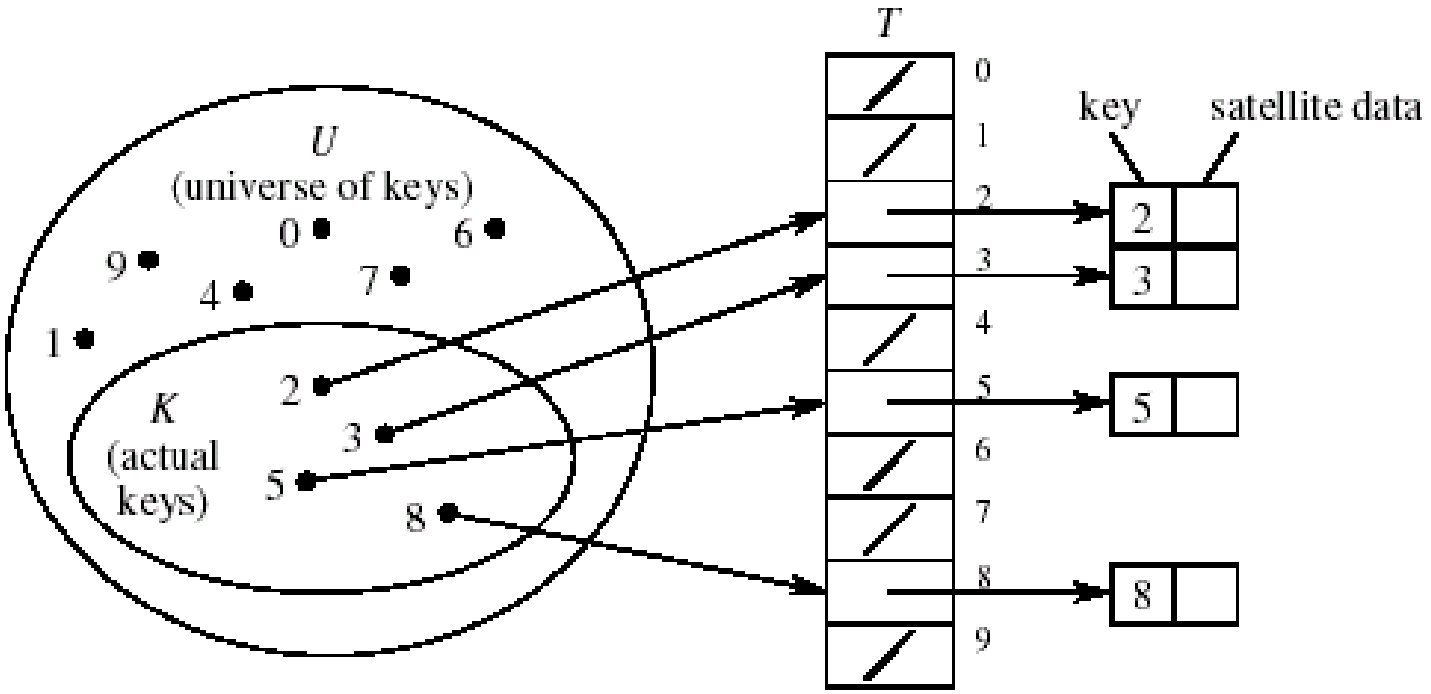


Tabella a indirizzamento diretto: operazioni

DIRECT-ADDRESS-SEARCH(T, k)
  return T[k]

O(1)

DIRECT-ADDRESS-INSERT(T, x)
  T[key[x]] ← x

O(1)

DIRECT-ADDRESS-DELETE(T, x)
  T[key[x]] ← NIL

O(1)

Tabelle a indirizzamento diretto vs. tabelle hash

Se |U| è grande, memorizzare T può essere impraticabile a causa della limitatezza della memoria disponibile; inoltre l'insieme $K \subseteq U$ delle chiavi effettivamente utilizzate può essere così piccolo rispetto a U che la maggior parte dello spazio occupato da T sarebbe di fatto inutilizzato

↓

per ovviare a queste difficoltà si usano le tabelle hash

che richiedono una memoria $\Theta(|K|)$ e un tempo di ricerca di un oggetto O(1) (limite valido per il tempo medio mentre per l'indirizzamento diretto vale nel caso peggiore)

Tabella hash

Funzione hash: definisce una corrispondenza deterministica fra U e le posizioni della tabella hash T[0 .. m – 1] che contiene gli oggetti, dove m < |U|:

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

h(k) = valore hash di k = indice dell'oggetto con chiave k in T

L'obiettivo di h è ridurre l'intervallo degli indici dell'array, cioè ridurre m rispetto a |U|

Problema: due chiavi possono avere lo stesso valore di hash (fenomeno di collisione)

Metodi per la risoluzione delle collisioni

- Concatenazione
- Indirizzamento aperto

Risoluzione delle collisioni per concatenazione

Tutti gli elementi che collidono nella stessa posizione j di T vengono posti in una lista concatenata e nella posizione j si memorizza il puntatore alla testa della lista

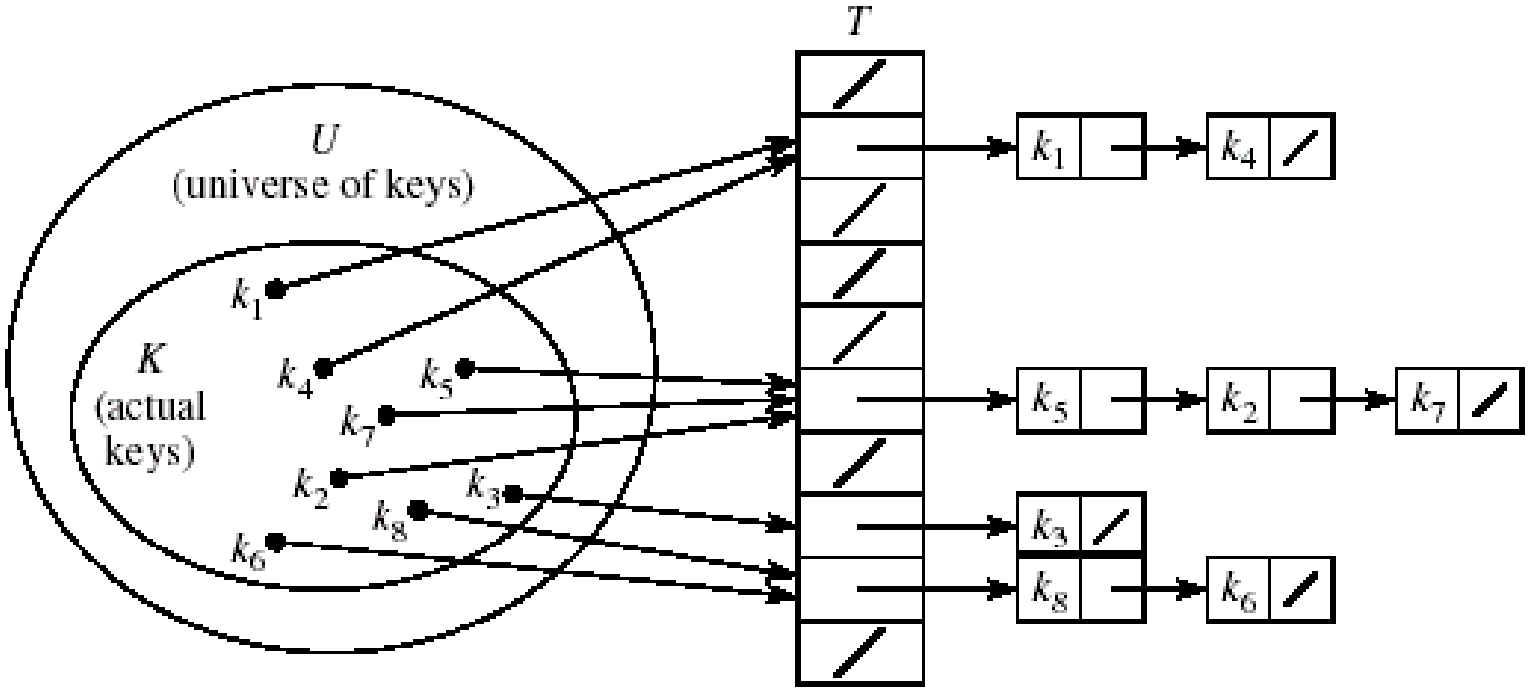


Tabella hash con concatenazione: operazioni

CHAINED-HASH-INSERT(T, x)
 inserisci x in testa alla lista T[h(key[x])]

O(1) nel caso pessimo

CHAINED-HASH-SEARCH(T, k)
 ricerca un elemento con chiave k
 nella lista T[h(k)]

O(n) nel caso pessimo
n = lunghezza della lista T[h(k)]

CHAINED-HASH -DELETE(T, x)
 cancella x dalla lista T[h(key[x])]

O(n) nel caso pessimo perché richiede innanzi tutto la ricerca dell'elemento da cancellare

Analisi della ricerca entro una tabella hash con concatenazione

Fattore di carico α di una tabella hash con m posizioni che memorizza n elementi (le cui chiavi appartengono a un universo U) ≡ numero medio di elementi memorizzati in ogni lista concatenata = n/m

α può essere <, >, = 1

Ricerca entro una tabella hash con concatenazione: caso pessimo

Tutte le n chiavi corrispondono alla stessa posizione → lista di lunghezza n → tempo di ricerca $\Theta(n)$ + tempo per calcolare h(k) → tempo non migliore di quello necessario quando, per rappresentare il dizionario, si usa una lista concatenata

Ricerca entro una tabella hash con concatenazione: caso medio

Ipotesi:

1) $\forall$ valore hash h(k) è calcolato in tempo O(1)
2) Uniformità semplice della funzione hash: qualunque chiave corrisponde in modo equamente probabile a una delle m posizioni

Due teoremi affermano che, date entrambe le precedenti ipotesi, in una tabella hash in cui le collisioni sono risolte per concatenazione, una ricerca, sia essa con o senza successo, richiede in media un tempo $\Theta(1 + \alpha)$

$\downarrow$

Se m è proporzionale a n $\rightarrow$ n = O(m) $\rightarrow \alpha$ = n/m = O(m)/m = O(1), cioè la ricerca richiede un tempo medio costante

$\downarrow$

tutte le operazioni del dizionario possono essere eseguite in media con tempo O(1)

Requisiti di una buona funzione hash per una tabella hash con concatenazione

Soddisfare (approssimativamente) l'ipotesi di uniformità semplice, come sotto formulata:

probabilità che k sia estratta (in modo indipendente) da U

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

Sfortunatamente la distribuzione di probabilità P di solito non è conosciuta

Esempio in cui P è nota:
le chiavi sono numeri reali casuali distribuiti in modo indipendente e uniforme nell'intervallo $0 < k < 1$
La funzione hash $h(k) = \lfloor km \rfloor$ soddisfa l'ipotesi di uniformità semplice

Creazione di funzioni hash per una tabella hash con concatenazione

Si possono adottare tecniche euristiche, che tengano conto di info qualitative su P

Ad es. una buona funzione hash per la tabella dei simboli di un compilatore deve minimizzare la possibilità che identificatori simili (quali `pt` e `pts`), che sono frequenti in un programma, corrispondano alla stessa posizione

Creazione di funzioni hash per una tabella hash con concatenazione (cont.)

Molte funzioni hash assumono U = insieme numeri naturali → se k non è un naturale, bisogna interpretarlo come tale

Esempio
Tabella dei simboli di un compilatore → le chiavi sono stringhe
k = `pt`
Codifica ASCII: `p` = 112, `t` = 116

k = (112, 116) dove 112 e 116 sono trattati come due cifre in base 128 = $(112 \cdot 128 + 116)_{10} = 14452_{10}$

Schemi per la creazione di funzioni hash per una tabella hash con concatenazione

Assunzione: le chiavi sono numeri naturali

- Per divisione
- Per moltiplicazione
- Universale

Metodo di divisione

$h(k) = k \bmod m$

Sono valori di m da evitare i seguenti:

- $m = 2^p$ perché altrimenti $h(k) = k \bmod 2^p$ = p bit meno significativi di k → il valore di hash non dipende da tutti i bit della chiave
- in generale, $m = b^p$ se le chiavi sono numeri in base b perché altrimenti $h(k) = k \bmod b^p$ = valore che non dipende da tutte le cifre in base b
- $m = 2^p - 1$ se k è una stringa di caratteri interpretata in base 2^p perché altrimenti a due stringhe identiche eccetto che per uno scambio di due caratteri adiacenti corrisponde lo stesso valore di hash

Sono buoni valori di m valori primi non troppo vicini a potenze esatte di 2

Metodo di divisione (cont.)

Esempio
Si desidera realizzare una tabella hash con concatenazione. Scegliere un valore di m appropriato per una funzione hash che adotta il metodo di divisione.

Dati:
- n° chiavi = 2000
- lunghezza media accettabile di una lista concatenata = 3

$$\downarrow$$

m = n° primo vicino a 2000/3 = 701

Effettuata questa scelta, si dovrebbe controllare se effettivamente h distribuisce regolarmente le chiavi fra le 701 posizioni, provando con chiavi vere

Metodo di moltiplicazione

$h(k) = \lfloor m(kA \bmod 1) \rfloor$

dove

- A è una costante, $0 < A < 1$
- con "kA mod 1" si è rappresentata la parte frazionaria di kA (cioè $kA - \lfloor kA \rfloor$)

Vantaggio: il valore di m non è critico, tipicamente si sceglie $m = 2^p$ perché h si può calcolare facilmente come segue

$h(k) = \lfloor m(kA \bmod 1) \rfloor = \lfloor 2^p ((k \lfloor A \cdot 2^w \rfloor \bmod 2^w) / 2^w) \rfloor$

dove

- w = lunghezza di parola
- k occupa una sola parola

Metodo di moltiplicazione (cont.)

Procedimento di calcolo di $h(k) = \lfloor 2^p ((k \lfloor A \cdot 2^w \rfloor \bmod 2^w) / 2^w) \rfloor$

1) Calcola $\lfloor A \cdot 2^w \rfloor$ = intero su w bit
2) Calcola $k \lfloor A \cdot 2^w \rfloor$ = intero su 2w bit = $r_1 \cdot 2^w + r_0$ (w bit per r_1 e w per r_0)
3) Isola h(k) = p bit più significativi di r_0

Qualche valore di A funziona meglio di altri; Knuth suggerisce

$A \approx (\sqrt{5} - 1) /2 = 0.6180339887....$

(cioè l'opposto del coniugato del rapporto aureo dei numeri di Fibonacci)

Hashing universale

Caso pessimo quando la funzione hash è nota a priori: le chiavi vengono scelte dispettosamente in modo dipendente da h → a n chiavi può corrispondere la stessa posizione → tempo medio di ricerca $\Theta(n)$

Obiettivo:
migliorare il tempo medio di ricerca nel caso peggiore

Metodo:
hashing universale = scegliere la funzione hash da usare per la tabella hash corrente in modo casuale da una classe di funzioni attentamente definite cosicché sia indipendente dalle chiavi scelte dall'avversario

Hashing universale (cont.)

Sia

$H = \{h \mid h: U \rightarrow \{0, 1, \ldots, m - 1\}\}$ dove $|H| =$ n° finito

Sia

$H' \subseteq H$, $H' = \{h \mid h \in H, h(x) = h(y), x \in U, y \in U, x \neq y\}$

H è detto insieme universale di funzioni hash per l'universo U di chiavi considerato e per il valore di m fissato se, $\forall x, y \in U$, $|H'| = |H| / m$

In altre parole, scelta h in modo casuale entro H, la probabilità di collisione fra due chiavi x e y, $x \neq y$, è 1/m, valore che coincide con la probabilità di collisione corrispondente a una scelta casuale dei due valori di hash entro $\{0, 1, \ldots, m - 1\}$

Hashing universale: comportamento nel caso medio

Teorema

Se h è scelta da un insieme universale di funzioni hash ed è usata per n chiavi su una tabella di dimensione m, $n \leq m$, il n° medio di collisioni che coinvolge ciascuna chiave è < 1

Hashing universale: costruzione di una classe universale

Problema: come si costruisce una classe universale di funzioni hash?

Soluzione: $H = \bigcup_{a} \{h_a\}$ dove $h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$

m = n° primo

$x = \langle x_0, x_1, \ldots, x_r \rangle$: scomposizione di x in r + 1 sottostringhe binarie di ampiezza fissa, $x_i < m$, $\forall i \in [0, .., r]$

$a = \langle a_0, a_1, \ldots, a_r \rangle$: sequenza di r + 1 elementi scelti a caso da $\{0, 1, \ldots, m - 1\}$ $\rightarrow m^{r+1}$ sequenze distinte $\rightarrow |H| = m^{r+1}$

Teorema
La classe H sopra definita è una classe universale di funzioni hash

Indirizzamento aperto

Gli elementi sono memorizzati nella tabella hash stessa → ogni elemento della tabella contiene o un elemento o NIL (valore convenzionale per identificare una posizione vuota)

$\nexists$ puntatori (vantaggio dell'indirizzamento aperto perché risparmia memoria)
$\nexists$ elementi memorizzati fuori dalla tabella

$\alpha \leq 1$

Indirizzamento aperto: inserimento di un elemento

Si scandiscono le posizioni della tabella finché non se ne trova una vuota; se la ricerca fosse sequenziale richiederebbe un tempo lineare → meglio effettuare una scansione che dipende dalla chiave → obiettivo: associare a ciascuna chiave una sua sequenza di scansione (usata sia per un inserimento, sia per una ricerca)

A tal fine si estende la funzione hash perché includa il n° di posizioni già esaminate (che parte da 0) come secondo input

$h(k, i) : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\}$

Vincolo: per ogni chiave k, la sequenza di scansione

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

è una permutazione di $\{0, 1, \ldots, m - 1\}$

Indirizzamento aperto: inserimento di un elemento (cont.)

```
HASH-INSERT(T, k)
1   i ← 0
2   repeat j ← h(k, i)
3       if T[j] = NIL
4           then    T[j] ← k
5           ▷ si assume, per semplicità, che gli
6           elementi non contengano info satellite
7                   return j
8           else    i ← i + 1
9   until i = m
10  error "overflow sulla tabella hash"
```

Indirizzamento aperto: ricerca di un elemento

Scandisce la stessa sequenza di posizioni percorsa dall'algoritmo di inserimento quando l'elemento è stato inserito; ritorna l'indice della posizione che contiene l'elemento o NIL, se l'elemento non è presente

```
HASH-SEARCH(T, k)
1   i ← 0
2   repeat j ← h(k, i)
3       if T[j] = k
4           then    return j
5       i ← i + 1
6   until T[j] = NIL o i = m
7 ▷ la ricerca termina quando la scansione trova
  una posizione vuota poiché l'elemento di chiave k
  sarebbe stato inserito proprio lì (l'assunzione implicita
  è che gli elementi non vengano mai cancellati)
8 return NIL
```

Indirizzamento aperto: sequenza di scansione

Ipotesi di uniformità della funzione hash = generalizzazione dell'ipotesi di uniformità semplice della funzione hash al caso in cui la funzione produca non un singolo valore ma una sequenza di scansione (cioè al caso in cui h sia funzione di due variabili di ingresso): ogni chiave ha, in modo equamente probabile, una qualunque delle m! permutazioni di {0, 1, …, m – 1} come sequenza di scansione, cioè ogni possibile sequenza di scansione è equamente probabile → vincolo: una h uniforme deve generare m! sequenze di scansione distinte

Problema: difficoltà di realizzare una h uniforme

Indirizzamento aperto: tecniche per calcolare la sequenza di scansione

- Scansione lineare
- Scansione quadratica
- Hashing doppio

Nessuna di esse può generare più di m^2 sequenze di scansione diverse → nessuna di esse rispetta l'ipotesi di uniformità

L'hashing doppio genera il maggior n° di sequenze di scansione → fra le tre è la tecnica che dà i risultati migliori

Indirizzamento aperto: scansione lineare

$h(k, i) = (h'(k) + i) \bmod m$ dove h' è una funzione hash

Il valore h'(k) determina l'intera sequenza di scansione → sono usate solo m sequenze di scansione distinte

Esempio.
m = 8, h'(k) = 3 → sequenza di scansione

$h(k, 0) = h'(k) \bmod 8 = 3 \bmod 8 = 3$
$h(k, 1) = 4 \bmod 8 = 4$
$h(k, 2) = 5 \bmod 8 = 5$
$h(k, 3) = 6 \bmod 8 = 6$
$h(k, 4) = 7 \bmod 8 = 7$
$h(k, 5) = 8 \bmod 8 = 0$
$h(k, 6) = 9 \bmod 8 = 1$
$h(k, 7) = 10 \bmod 8 = 2$

Fra una posizione e la successiva esaminata la distanza è unitaria

Indirizzamento aperto: scansione lineare (cont.)

Problema: agglomerazione primaria = le posizione occupate tendono ad accumularsi in lunghi tratti, fatto che aumenta il tempo medio di ricerca

Esempio
Si assuma che il n° di elementi presenti in tabella sia m/2

CASO 1. Le posizioni con indice pari e dispari sono rispettivamente libere e occupate: la ricerca senza successo richiede in media 1,5 accessi

CASO 2. Sono occupate le prime m/2 posizioni: la ricerca senza successo richiede in media m/8 accessi

Indirizzamento aperto: scansione lineare (cont.)

Perché le posizione occupate tendono ad accumularsi, ovvero perché si verificano gli agglomerati?

Perché la probabilità che una posizione vuota sia la prossima a essere riempita vale

- $(i + 1)/m$ se tale posizione è preceduta da i piene
- $1/m$ se tale posizione è preceduta da una vuota

$(i + 1)/m > 1/m \rightarrow$ i tratti di posizioni occupate tendono a diventare ancora più lunghi

Scansione quadratica

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

dove

- h' è una funzione hash
- c_1 e c_2 sono costanti, $c_2 \neq 0$

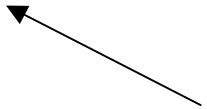
Il valore $h'(k)$ determina l'intera sequenza di scansione $\rightarrow$ sono usate solo m sequenze di scansione distinte

Fra una posizione e la successiva esaminata la distanza dipende in modo quadratico da i

Problema: agglomerazione secondaria = le posizione occupate tendono ad accumularsi quando a due chiavi k_1 e k_2 corrisponde la stessa posizione iniziale (ciò avviene se $h'(k_1) = h'(k_2)$)

Hashing doppio

$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$ dove h_1 e h_2 sono funzioni hash



<u>Esempio</u>

$h_1(k) = k \bmod 13$
$h_2(k) = 1 + k \bmod 11$

Si vuole inserire la chiave k = 14 in una tabella con dimensione m = 13

$h(14, 0) = (14 \bmod 13) \bmod 13 = 1 \bmod 13 = 1$ (posizione occupata)

$h(14, 1) = (14 \bmod 13 + 1 + 14 \bmod 11) \bmod 13 = (1 + 1 + 3) \bmod 13 = 5$ (posizione occupata)

$h(14, 2) = [14 \bmod 13 + 2(1 + 14 \bmod 11)] \bmod 13 = [1 + 2(1 + 3)] \bmod 13 = 9 \bmod 13 = 9$ (posizione libera in cui viene inserita la chiave)

Hashing doppio (cont.)

Prima dell'inserimento

0	
1	65
2	
3	
4	77
5	105
6	
7	43
8	
9	
10	
11	48
12	

Dopo

0	
1	65
2	
3	
4	77
5	105
6	
7	43
8	
9	14
10	
11	48
12	

Hashing doppio (cont.)

Ogni possibile coppia $(h_1(k), h_2(k))$ implica una diversa sequenza di scansione, dove $h_1(k)$ e $h_2(k)$ variano in modo indipendente $\rightarrow \Theta(m^2)$ sequenze di scansione distinte

Hashing doppio: scelta della funzione hash

Vincolo

$h_2(k)$ deve essere primo rispetto a m per $\forall k \in U$, altrimenti, se $\exists k \mid m$ e $h_2(k)$ hanno un MCD d >1, la ricerca corrispondente a k andrebbe a esaminare solo una frazione (1/d) della tabella

↓

Metodi alternativi per soddisfare il vincolo

1) Scegliere $m = 2^p$ e h_2 che produce sempre un n° dispari
2) Scegliere m = n° primo e h_2 che produce sempre un n° intero positivo < m

Secondo il metodo 2) in generale si può scegliere

$h_1(k) = k \bmod m$
$h_2(k) = 1 + k \bmod m'$ dove m' è di poco minore di m (ad es. m – 1 o m – 2)

Hashing doppio (cont.)

Esempio

Determinare una funzione hash che realizzi un hashing doppio secondo il metodo 2) per una tabella di hash a indirizzamento aperto contenente approssimativamente 700 posizioni

m= 701 (n° primo)

$h_1(k) = k \bmod 701$
$h_2(k) = 1 + k \bmod 700$

Operazione di inserimento in una tabella hash con indirizzamento aperto: caso medio

Teorema

Data una tabella hash a indirizzamento aperto con $\alpha < 1$, il n° medio di accessi di una ricerca senza successo è al più $1/(1 - \alpha)$, assumendo l'uniformità della funzione hash

Corallario

L'inserzione in una tabella hash a indirizzamento aperto con $\alpha < 1$ richiede al più $1/(1 - \alpha)$ accessi in media, assumendo l'uniformità della funzione hash

Se α = costante

n° medio accessi per una ricerca senza successo = n° medio accessi per una inserzione = O(1)

Operazione di ricerca in una tabella hash con indirizzamento aperto: caso medio

Teorema

Data una tabella hash a indirizzamento aperto con $\alpha < 1$, il n° medio di accessi di una ricerca con successo è al più

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

assumendo l'uniformità della funzione hash e nell'ipotesi che ogni chiave sia ricercata nella tabella in modo equiprobabile