Tabelle hash

Moreno Marzolla
Dip. di Scienze dell'Informazione
Università di Bologna

marzolla@cs.unibo.it
http://www.moreno.marzolla.name/

Introduzione

- Dizionario:
  - Struttura dati per memorizzare insiemi dinamici di coppie (chiave, valore)
  - Il valore è un "dato satellite"
  - Dati indicizzati in base alla chiave
  - Operazioni: `insert()`, `search()` e `delete()`
- Costo delle operazioni:
  - O(log n) nel caso pessimo, usando alberi bilanciati
  - Idealmente, sarebbe bello poter scendere a O(1)

Notazione

- U—Universo di tutte le possibili chiavi
- K—Insieme delle chiavi effettivamente memorizzate
- Possibili implementazioni
  - U corrisponde all'intervallo [0..m-1], |K| ~ |U|
    - tabelle ad indirizzamento diretto
  - U è un insieme generico, |K| << |U|
    - tabelle hash

Tabelle a indirizzamento diretto

- Implementazione:
  - Basata su array
  - L'elemento con chiave *k* è memorizzato nel *k*-esimo "slot" dell'array

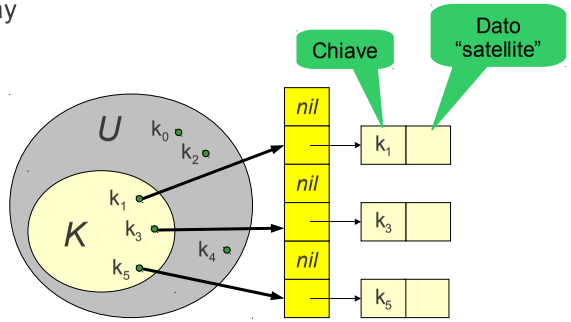


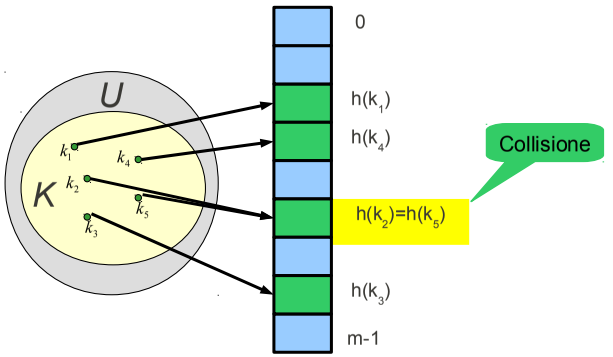
Tabelle a indirizzamento diretto

- Se |K| ~ |U|:
  - Non sprechiamo (troppo) spazio
  - Operazioni in tempo O(1) nel caso peggiore
- Se |K| << |U|: soluzione non praticabile
  - Esempio: studenti ASD con chiave "n. matricola"
  - Se il numero di matricola ha 6 cifre, l'array deve contenere 10^6 elementi
  - Se gli studenti del corso sono ad esempio 70, lo spazio realmente occupato dalle chiavi memorizzate è $70/10^6$ = 0.00007 = 0.0007% !!!

Tabelle hash

- Tabelle hash:
  - Un array A[0..m-1]
  - Una funzione hash h: U → {0,..,m-1}
- Indirizzamento hash:
  - Diciamo che h(k) è il valore hash della chiave k
  - La funzione h trasforma una chiave nell'indice della tabella
  - La chiave k viene "mappata" nello slot A[h(k)]
  - Quando due o più chiavi nel dizionario hanno lo stesso valore hash, diciamo che è avvenuta una collisione
- Idealmente: vogliamo funzioni hash senza collisioni

Tabelle hash



Problema delle collisioni

- Utilizzo di funzioni hash perfette
  - Una funzione hash h si dice perfetta se è iniettiva, ovvero:

$$\forall u, v \in U : u \neq v \rightarrow h(u) \neq h(v)$$

  - Si noti che questo richiede che |U| ≤ m
- Esempio:
  - Numero di matricola degli studenti ASD solo negli ultimi tre anni
  - Distribuiti fra 234.717 e 235.716
  - h(k) = k- 234.717, m = 1000
- Problema: spazio delle chiavi spesso grande, sparso
  - È impraticabile ottenere una funzione hash perfetta

Funzioni hash

- Se le collisioni sono inevitabili
  - almeno cerchiamo di minimizzare il loro numero
  - vogliamo funzioni che distribuiscano uniformemente le chiavi negli indici [0..m-1] della tabella hash
- Uniformità semplice:
  - sia P(k) la probabilità che una chiave k sia presente nella tabella
  - La quantità $Q(i) = \sum_{k:h(k)=i} P(k)$

    è la probabilità che una chiave finisca nella cella i.
  - Una funzione h() gode della proprietà di uniformità semplice se

$$\forall i \in [0...m-1] : Q(i) = 1/m$$

Funzioni hash

- Per poter ottenere una funzione hash con uniformità semplice, la distribuzione delle probabilità *P* deve essere nota
- Esempio:
  - U è composto da numeri reali in [0,1) e ogni chiave k ha la stessa probabilità di essere scelta. Allora

$$h(k) = \lfloor km \rfloor$$

    soddisfa la proprietà di uniformità semplice
- Nella realtà la distribuzione esatta può non essere (completamente) nota

Estremo superiore escluso

Funzioni hash: assunzioni

- Tutte le chiavi sono equiprobabili: P(k) = 1 / |U|
  - Semplificazione necessaria per proporre un meccanismo generale
- Le chiavi sono valori numerici non negativi
  - È possibile trasformare una chiave complessa in un numero, ad esempio considerando il valore decimale della sua rappresentazione binaria

Funzioni hash

- Metodo della divisione: $h(k)=k \bmod m$
  - Basata sul resto della divisione per *m:*
  - Esempio: *m=12, k=100* → *h(k) = 4*
- Vantaggio
  - Molto veloce (richiede una divisione intera)
- Svantaggio
  - Il valore *m* deve essere scelto opportunamente: ad esempio vanno bene numeri primi, distanti da potenze di 2 (e di 10)

Funzioni hash

- Metodo della moltiplicazione: $h(k)=\lfloor m(kA-\lfloor kA \rfloor)\rfloor$
  - Sia *A* una costante, *0 < A < 1*
  - Moltiplichiamo *k* per *A* e prendiamo la parte frazionaria
  - Moltiplichiamo quest'ultima per *m* e prendiamo la parte intera
- Esempio: *m = 1000, k = 123, A ≈ 0.6180339887...* → *h(k)=18*
- Svantaggi: più lento del metodo di divisione
- Vantaggi: il valore di m non è critico
- Come scegliere A? Knuth suggerisce $A \approx (\sqrt{5} - 1)/2$.

java.lang.String.hashCode()

http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html#hashCode()

hashCode

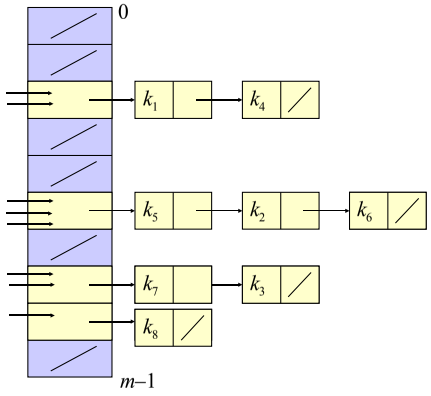
```
public int hashCode()

    Returns a hash code for this string. The hash code for a
String object is computed as

         s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

    using int arithmetic, where s[i] is the ith character of
the string, n is the length of the string, and ^ indicates
exponentiation. (The hash value of the empty string is zero.)
```


Problema delle collisioni

- Abbiamo ridotto, ma non eliminato, il numero di collisioni
- Come gestire le collisioni residue?
  - Dobbiamo trovare collocazioni alternative per le chiavi
  - Se una chiave non si trova nella posizione attesa, bisogna andare a cercare nelle posizioni alternative
  - Le operazioni possono costare Θ(n) nel caso peggiore...
  - ...ma hanno costo Θ(1) nel caso medio
- Due delle possibili tecniche:
  - Concatenamento
  - Indirizzamento aperto

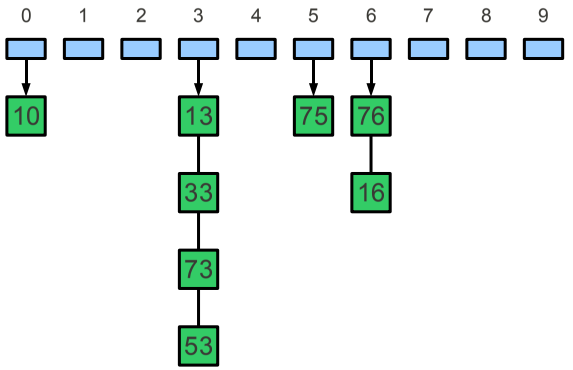
Risoluzione delle collisioni

- Concatenamento (chaining)
  - Gli elementi con lo stesso valore hash *h* vengono memorizzati in una lista concatenata
  - Si memorizza un puntatore alla testa della lista nello slot *A[h]* della tabella hash
- Operazioni:
  - Insert: inserimento in testa
  - Search, Delete: richiedono di scandire la lista alla ricerca della chiave



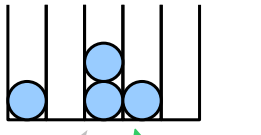
Esempio
h(k) = k mod 10

Attenzione, m=10 andrebbe evitato nei casi reali



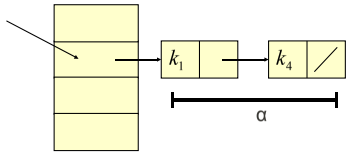
Concatenamento: complessità

- Notazione
  - n: numero di elementi nella tabella
  - m: numero di slot nella tabella
- Fattore di carico
  - α: numero medio di elementi nelle liste (α = n/m)
- Caso pessimo: tutte le chiavi sono in una unica lista
  - Insert: Θ(1)
  - Search, Delete: Θ(n)
- Caso medio: dipende da come le chiavi vengono distribuite
  - Assumiamo hashing uniforme, da cui ogni slot della tabella avrà mediamente α chiavi



Concatenamento: complessità

- Teorema:
  - In tabella hash con concatenamento, una ricerca senza successo richiede un tempo atteso di Θ(1 + α)
- Dimostrazione:
  - Una chiave non presente nella tabella può essere collocata in uno qualsiasi degli *m* slot
  - Una ricerca senza successo tocca tutte le chiavi nella lista corrispondente
  - Tempo di hashing: 1 + lunghezza attesa lista: $\alpha \rightarrow \Theta(1+\alpha)$



Concatenamento: complessità

- Teorema:
  - In una tabella hash con concatenamento, una ricerca con successo richiede un tempo atteso di $\Theta(1 + \alpha)$
  - Più precisamente: $\Theta(2 + \alpha/2 + \alpha/2n)$
    - (dove n è il numero di elementi)
- Qual è il significato:
  - se $n = O(m)$, $\alpha = O(1)$
  - quindi tutte le operazioni sono $\Theta(1)$

Indirizzamento aperto

- Problema della gestione di collisioni tramite concatenamento
  - Struttura dati complessa, con liste, puntatori, etc.
- Gestione alternativa: indirizzamento aperto
  - Idea: memorizzare tutte le chiavi nella tabella stessa
  - Ogni slot contiene una chiave oppure `null`
  - **Inserimento**:
    - Se lo slot prescelto è utilizzato, si cerca uno slot "alternativo"
  - **Ricerca**:
    - Si cerca nello slot prescelto, e poi negli slot "alternativi" fino a quando non si trova la chiave oppure `nil`

Indirizzamento aperto

- Cosa succede al fattore di carico α?
  - Compreso fra 0 e 1
  - La tabella può andare in overflow
    - Inserimento in tabella piena
  - Esistono tecniche di crescita/contrazione della tabella
    - linear hashing

Indirizzamento aperto

- Ispezione: Uno slot esaminato durante una ricerca di chiave
- Sequenza di ispezione: La lista ordinata degli slot esaminati
- Funzione hash: estesa come
  - $h : U \times [0..m-1] \rightarrow [0..m-1]$

  chiave "tentativo"
- La sequenza di ispezione $\{ h(k,0), h(k,1),\ldots,h(k,m-1) \}$ è una permutazione degli indici $[0..m-1]$
  - Può essere necessario esaminare ogni slot nella tabella
  - Non vogliamo esaminare ogni slot più di una volta

Inserimento e Ricerca (attenzione...)

```
Hash-Insert(A, k)
  i := 0
  repeat
      j := h(k, i)
      if A[j] == null then
         A[j] := k
         return j
      else
         i := i + 1
      endif
  until i == m
  error "hash table overflow"
```

```
Hash-Search (A, k)
  i := 0
  repeat
      j := h(k, i)
      if A[j] == k then
         return j
      endif
      i := i + 1
  until A[j] == null or i == m
  return nil
```


Cancellazione

- Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un `null`. Perché?
- Approccio
  - Utilizziamo un speciale valore **`DELETED`** al posto di `null` per marcare uno slot come vuoto dopo la cancellazione
  - *Ricerca*: **`DELETED`** trattati come slot pieni
  - *Inserimento*: **`DELETED`** trattati come slot vuoti

Inserimento e Ricerca

```
Hash-Insert(A, k)
  i := 0
  repeat
      j := h(k, i)
      if A[j] == null ||
         A[j] == DELETED then
         A[j] := k
         return j
      else
         i := i + 1
      endif
  until i == m
  error "hash table overflow"
```

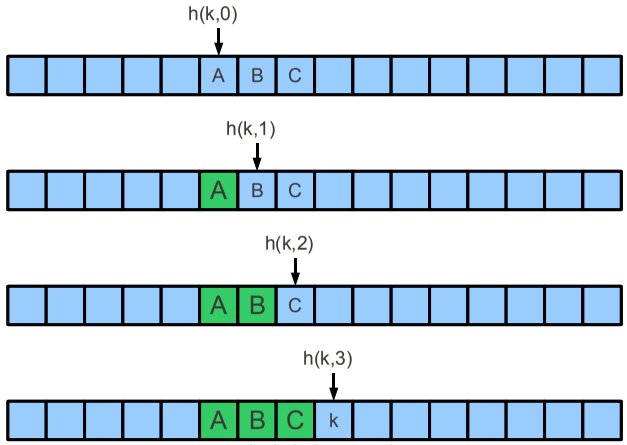
```
Hash-Search (A, k)
  i := 0
  repeat
      j := h(k, i)
      if A[j] == k then
         return j
      endif
      i := i + 1
  until A[j] == null or i == m
  return nil
```


Ispezione lineare

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

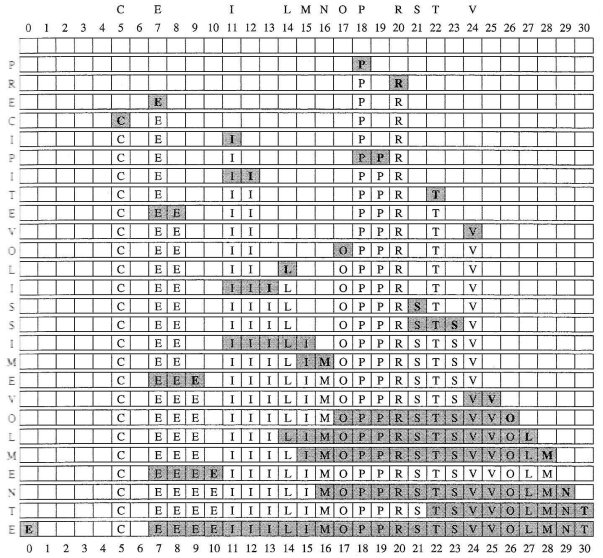
  chiave, n. ispezione, funzione hash ausiliaria
- Il primo elemento determina l'intera sequenza
  - $h'(k), h'(k)+1, \ldots, m-1, 0, 1, \ldots, h'(k)-1$
  - Solo m sequenze di ispezione distinte sono possibili
- Problema: primary clustering
  - Lunghe sotto-sequenze occupate...
  - ... che tendono a diventare più lunghe:
    - uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i+1)/m$
  - I tempi medi di inserimento e cancellazione crescono

Esempio



Esempio

(C. Demetrescu, I. Finocchi, G. F. Italiano, "Algoritmi e strutture dati", seconda edizione, McGraw-Hill, fig. 7.7 p. 189)



Ispezione quadratica

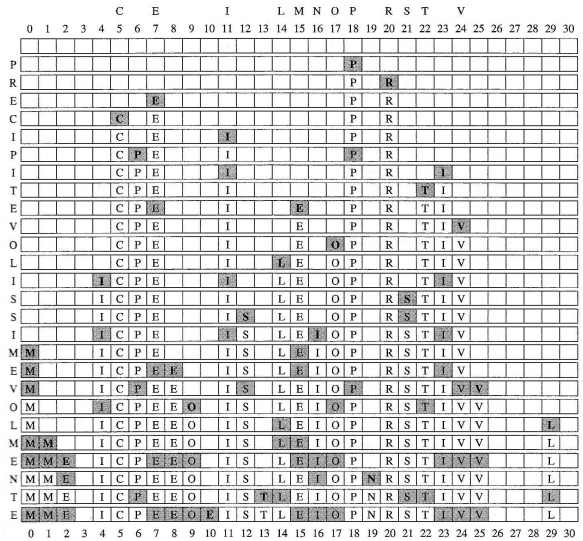
- Funzione: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ $\quad c_1 \neq c_2$

  chiave — n. ispezione — funzione hash ausiliaria

- Sequenza di ispezioni:
  - L'ispezione iniziale è in h'(k)
  - Le ispezione successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione i
  - Solo m sequenze di ispezione distinte sono possibili
- Nota: c_1, c_2, m devono essere tali da garantire la permutazione di [0..m-1].
- Problema: clustering secondario
  - Se due chiavi hanno la stessa ispezione iniziale, poi le loro sequenze sono identiche

Esempio

(C. Demetrescu, I. Finocchi, G. F. Italiano, "Algoritmi e strutture dati", seconda edizione, McGraw-Hill, Fig 7.8 p. 191)



Doppio hashing

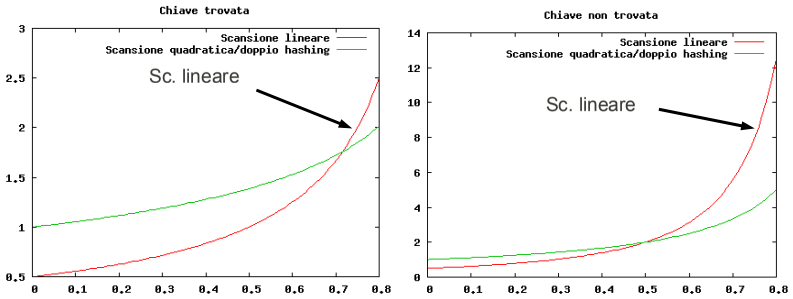
- Funzione: $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

  chiave, n. ispezione, funzioni hash ausiliarie
- Due funzioni ausiliarie:
  - h_1 fornisce la prima ispezione
  - h_2 fornisce l'offset delle successive ispezioni

Analisi dei costi di scansione nel caso medio

Esito ricerca	Concatenamento	Scansione lineare	Scansione quadratica / hashing doppio
Chiave trovata	$\Theta(1+\alpha)$	$\frac{1}{2}+\frac{1}{2(1-\alpha)}$	$-\frac{1}{\alpha}\ln(1-\alpha)$
Chiave non trovata	$\Theta(1+\alpha)$	$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$

Scansione lineare vs scansione quadratica



Osservazione

- Nel caso di gestione degli overflow mediante liste concatenate, è possibile avere $\alpha>1$
- Nel caso di gestione degli overflow mediante indirizzamento aperto, è possibile solo avere $\alpha\leq 1$
  - Una volta che l'array è pieno, non è più possibile aggiungere altri elementi
  - Come fare in questo caso?

Usi pratici delle Hash Tables

- Array associativi in PHP, Perl, AWK...
  - a[“pippo”] = 3
- Cache
  - La forma di cache piú semplice è una tabella hash
- Strutture dati per rappresentare insiemi
  - Cioè collezioni dinamiche di elementi in cui è importante testare la presenza o meno di un elemento in un insieme
  - java.util.HashSet<E>
- Rappresentazione degli oggetti nei linguaggi OO
  - Python, Ruby, Javascript usano tabelle hash per associare i nomi dei metodi e attributi degli oggetti alla loro rappresentazione

Algorithmic Complexity Attack

- **Domanda**: ma in pratica, quanto è importante l'assunzione che la funzione hash h(k) produca valori il più possibile uniformemente distribuiti in [0..m-1] ?
- **Risposta**: MOLTO importante
  - denial-of-service attack
  - Un avversario malevolo studia la vostra implementazione della funzione hash, producendo una sequenza di input che causano una sequenza di collisioni
  - Manipolare la tabella hash “degenere” costa O(n)

Esempi

- Bro server (Intrusion Detection System)
  - Send carefully chosen packets to DOS the server
- Perl 5.8.0
  - Insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel
  - Save files with carefully chosen names
  - Routing cache DOS
- Per maggiori dettagli
  - http://www.cs.rice.edu/~scrosby/hash/
  - http://www.enyo.de/fw/security/notes/linux-dst-cache-dos.html

Commenti finali

- Problemi con hashing
  - Scarsa “locality of reference” (cache miss)
  - In base all'implementazione, è in genere difficile (se non impossibile) ottenere le chiavi in ordine
  - Sebbene il costo *medio* per operazione sia basso, la singola operazione può risultare molto costosa, ad esempio se occorre ridimensionare la tabella e redistribuire le chiavi