Tabelle hash

Luca Becchetti

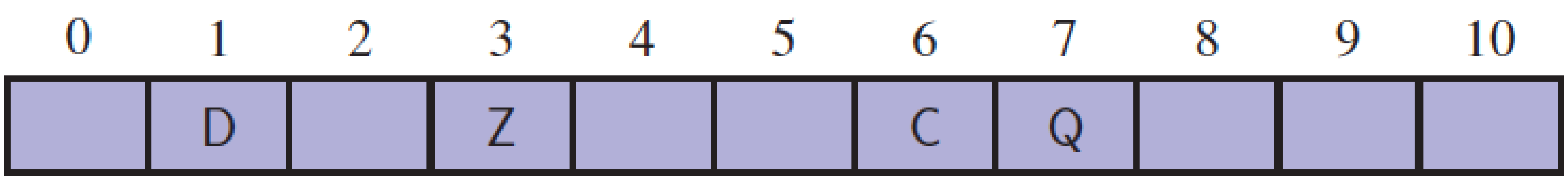
Presentazione tratta dalle slide che accompagnano il testo Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Il tipo di dato Mappa - recap

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Desiderata per una mappa

- Accesso efficiente agli elementi in base alla loro chiave
- Estensione del concetto di array
- Caso estremo: le chiavi sono interi ≤ N - 1

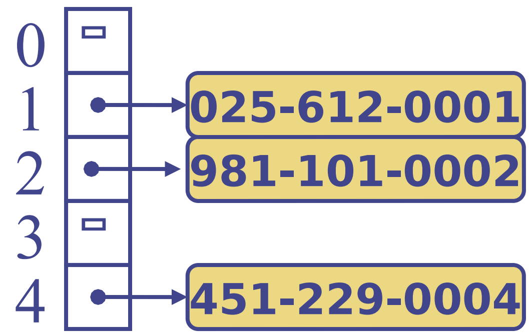


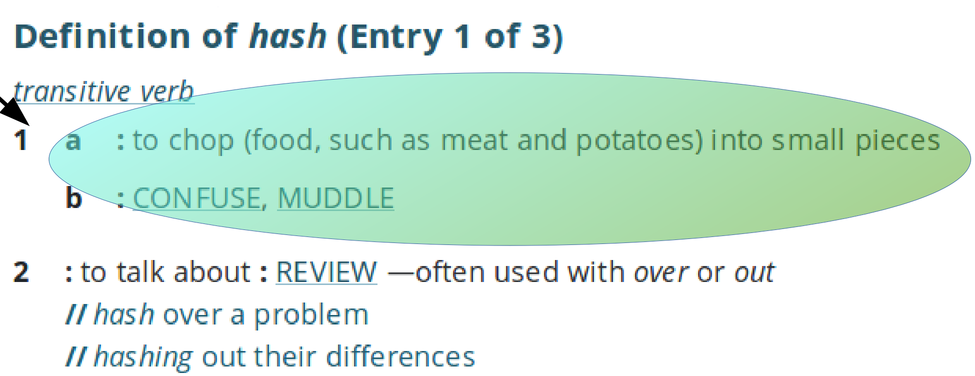
Funzioni hash

Gestire tipi generali di chiave

- Usiamo un array di N elementi
- Usiamo una **funzione hash** per associare a ciascuna chiave un intero nell'intervallo [0 … N-1]

Esempio: usiamo le ultime 4 cifre di un codice identificativo





Definition of *hash* (Entry 1 of 3)

transitive verb

1 **a** **:** to chop (food, such as meat and potatoes) into small pieces

b **:** CONFUSE, MUDDLE

2 **:** to talk about **:** REVIEW —often used with *over* or *out*

// *hash* over a problem

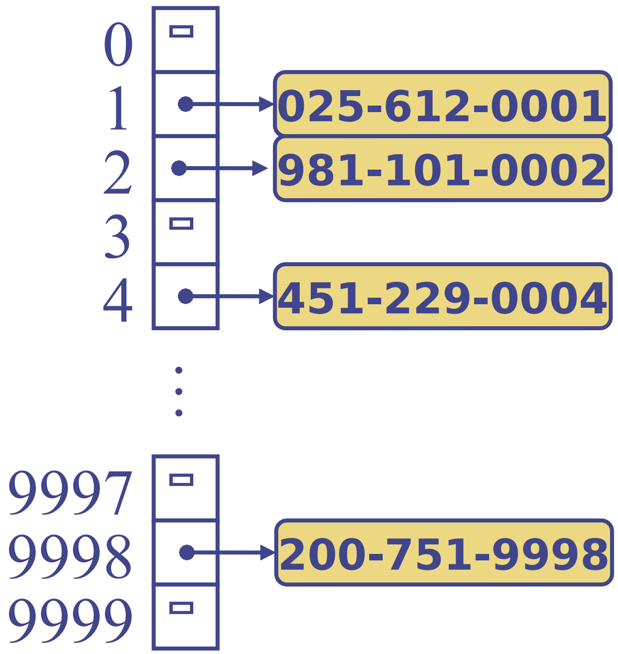
// *hashing* out their differences

Funzioni hash e tabelle hash

- **Funzione hash:** mappa chiavi in un intervallo [0 … N-1] di interi
  - Esempio $h(x) = x \bmod N$
  - $h(x)$ è detto *valore hash* della chiave x
- **Tabella hash:**
  - Funzione hash h
  - Array (detto tabella) di dimensione N
  - Implementazione di una mappa con tabella hash → obiettivo è memorizzare la coppia (k, v) in posizione h(k) nell'array
- Problemi
  - Chiavi non intere
  - Collisioni

Esempio (US)

- SSN (social security number): intero positivo a 9 cifre
- Usiamo un array di N = 10000 posizioni
- Ultime 4 cifre dell'SSN usate per indicizzare array
- Quindi: h(x) = <ultime 4 cifre di x>



Gestire chiavi qualsiasi → hash code

Chiavi non intere

- Esempio (IT): le chiavi sono codici fiscali
  - Dato alfanumerico → stringa
- Procediamo in due passi
  - Trasformiamo le chiavi in interi → **Hash code**
    - h_1: <chiavi> → <interi>
  - **Funzione di compressione**
    - h_2: <interi> → [0 … N-1]
- Funzione complessiva
  - $h(x) = h_2(h_1(x))$, dove x è una chiave

Requisiti per gli hash code

- Comportarsi come funzioni
  - Due oggetti con la stessa chiave devono avere lo stesso hash code
- Iniettività (o quasi)
  - Chiavi distinte sono mappate su interi distinti …
  - … almeno nella stragrande maggioranza dei casi

Esempio (chiavi di 4 lettere, N = 10000)

- Chiave = “ABCD”
- Hash code → intero a 32 bit ottenuto considerando codifica ASCII di ciascun carattere
  - A → 01000001, B → 01000010, C → 01000011, D → 01000100
  - Hash code = 01000001010000100100001101000100 → x = 1094861636
- Funzione di compressione: x mod 10000 = 1636
- *Nota:* questo è soltanto un esempio

Metodi per ottenere hash code

- Indirizzo di memoria
  - l’indirizzo di memoria di un oggetto è interpretato come una chiave (es. a 32 bit) associata all’oggetto
  - È la scelta standard per in Java
  - Non va bene per chiavi numeriche o stringhe → **Perche?**
- Conversione in intero
  - Va bene quando le chiavi appartengono a un tipo la cui rappresentazione ha lunghezza inferiore alla rappresentazione binaria dell’intero
    - Come nell’esempio precedente
- Somma delle componenti
  - Si partizionano I bit della chiave in sottosequenze di lunghezza non superiore a quella della rappresentazione binaria dell’intero
  - es.: “ABCDEFGH”
  - Molte collisioni potenziali → perché?

Uso di polinomi

- Adatto a chiavi di lunghezza variabile (numero di bit)
- Si partiziona la chiave in componenti di lunghezza fissa b (8, 16, 32 bit …) trattate come interi
  - Esempio, k = 1001000000001000, b = 8, hash code a 8 bit
  - $a_0 = 10010000 \rightarrow 144$, $a_1 = 00001000 \rightarrow 8$
- Siano $a_0 a_1 \ldots a_{n-1}$ le componenti
- Si calcola $p(z) = a_0 + a_1 z + \ldots + a_{n-1} z^{n-1}$ per un valore fisso z
  - Si trascurano eventuali overflow
  - Esempio: se z = 10 nel nostro caso avremmo: p(10) = 80 + 144 = 224 (no overflow)

Calcolo di p(z)

- Quanto costa calcolare $p(z) = a_0 + a_1 z + \dots + a_{n-1} z^{n-1}$?
- Regola di Horn (O(n)):
  - $p_0(z) = a_{n-1}$
  - $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$, $i = 1, \dots, n-1$
  - $p(z) = p_{n-1}(z)$
- Ad esempio, $z = 33$ → al più 6 collisioni su un insieme di 50000 termini della lingua inglese
- Varianti: shift ciclici → v. libro di testo

Shift ciclici

- 00111101100101101010100010101000

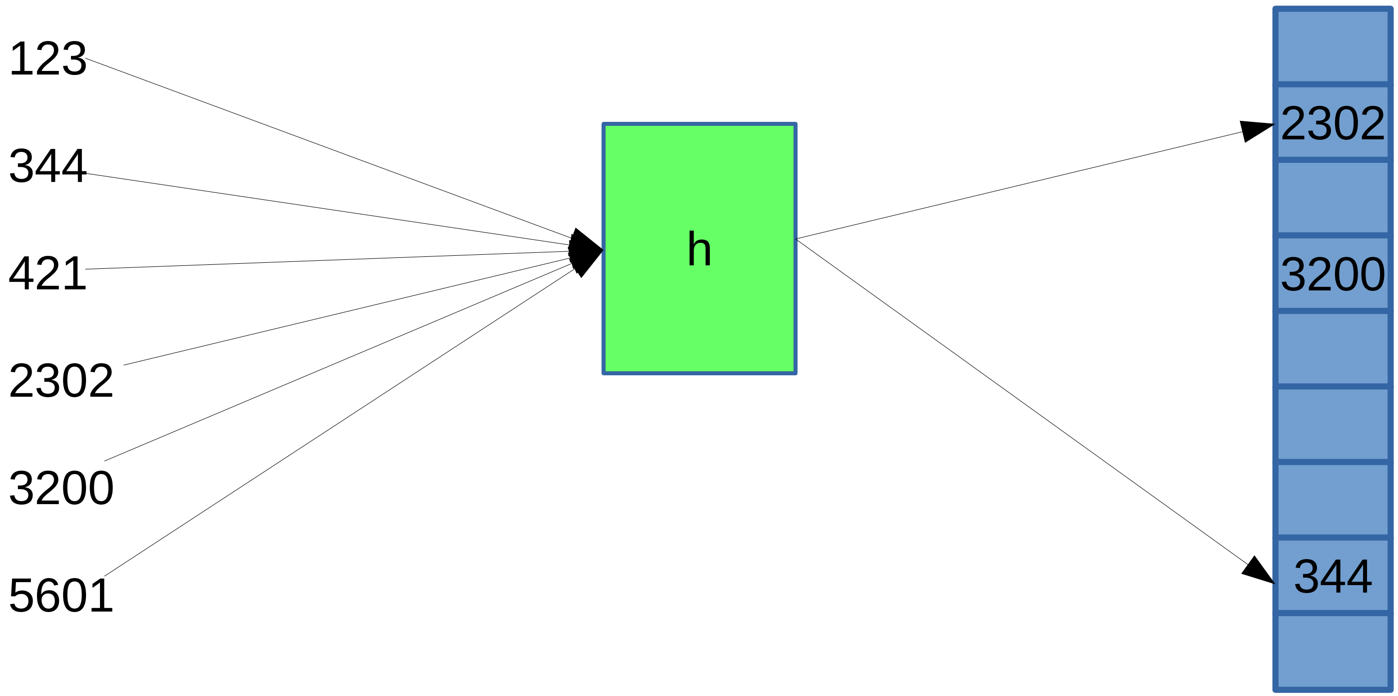
- 10110010110101010001010100000111

```
static int hashCode(String s) {
  int h=0;
  for (int i=0; i<s.length(); i++) {
    h = (h << 5) | (h >>> 27);          // 5-bit cyclic shift of the running sum
    h += (int) s.charAt(i);             // add in next character
  }
  return h;
}
```

Compressione

Lo scenario a questo punto ...

- Nota bene: sono mostrate soltanto le chiavi (i loro hash code)



Obiettivi:
1. L’associazione tra valori delle chiavi e posizione nell’array “sembra” uniforme
2. Le collisioni sono rare

Funzioni di compressione

- Divisione
  - $h_2(x) = x \bmod N$
  - La dimensione N della tabella di solito un numero primo
    - Motivo → teoria dei numeri
- MAD (Multiply Add and Divide)
  - Molto spesso: $h_2(x) = ((ax + b) \bmod p) \bmod N$
  - p è un primo, $p > N$
  - $a \in [1 \ldots p-1]$, $b \in [0 \ldots p-1]$
  - Di solito, a e b sono scelti *indipendentemente e uniformemente a caso*
  - Questo garantisce che $P(h_2(x) = h_2(y))$ con $y \neq x$ sia molto piccola
  - Anche $h_2(x) = (ax + b) \bmod N$ dà risultati accettabili in molti casi

Esercizio

- Si supponga di usare una funzione di compressione $h_2(x)$ tale che:
  - $P(h_2(x) = i) = 1/N$
  - Si supponga che vengano inserite n chiavi (coppie) nella tabella hash
  - Considerata la generica posizione i, calcolare il valore atteso del numero di chiavi che vengono assegnate a i

Gestione delle collisioni

Collisioni

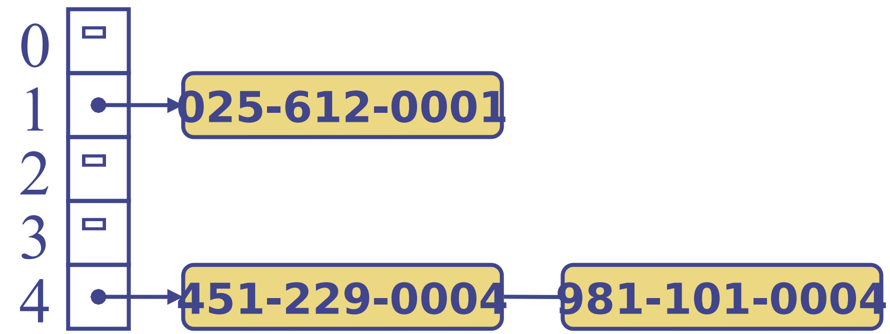
- L’universo delle chiavi è tipicamente >> dimensione della tabella hash
  - Es.: N = 10000, chiavi → insieme delle stringhe di 30 caratteri (26^{30})
- Si ha una collisione perché
  - Due chiavi hanno lo stesso hash code
    - Non molto frequente ma può capitare (ad esempio con le stringhe)
  - La funzione di compressione non è biettiva
    - Due interi possono essere mappati sullo stesso indice

Gestione delle collisioni

- Liste di trabocco (chaining)
  - Si associa una lista ad ogni elemento dell'array
  - Memoria addizionale esterna alla tabella
- Indirizzamento aperto
  - Si cerca un'altra posizione libera nella tabella
  - Non usa memoria addizionale
  - Casi particolari interessanti
    - Scansione lineare (linear probing)
    - Scansione quadratica
    - Hashing doppio (double hashing)

Gestione con liste di trabocco

- Ogni cella della tabella contiene il riferimento a una lista di tutte le coppie le cui chiavi sono state mappate nella cella
- Costo della ricerca nel caso peggiore?



Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = **null then** {k è una nuova chiave}
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ **null then** {k trovata}
    n = n - 1

Scansione lineare

- Si inserisce la coppia che genera collisione nella prossima posizione disponibile (mod N)
- Genera agglomerazione (clustering) primaria
  - Sequenze di celle occupate

- Esempio:
  - $h(x) = x \bmod 13$
  - Chiavi 18, 41, 22, 44, 59, 32, 31, 73, in questo ordine

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	

Scansione lineare: get(k)

- Si inizia alla posizione h(k)
- Si esaminano le posizioni a partire dall h(k)-esima in sequenza finché
  - Si trova una coppia con chiave k oppure
  - Si trova una cella vuota oppure
  - Si sono esaminate N posizioni senza successo

```
Algorithm get(k)
    i = h(k)
    p = 0
    repeat
        c = A[i]
        if c = null
            return null
        else if c.getKey () = k
            return c.getValue()
        else
            i = (i + 1) mod N
            p = p + 1
    until p = N
    return null
```

Scansione lineare: remove(k) e put(k, o)

- Usiamo uno speciale oggetto DEFUNCT che sostituisce elementi rimossi
- remove(k)
  - Cerchiamo k
  - Se troviamo (k, o) la sostituiamo con DEFUNCT e restituiamo o
  - Altrimenti restituiamo *null*
- put(k, o)
  - Se la tabella è piena → eccezione
    - *Costoso raddoppiare la dimensione*
  - Esaminiamo posizioni consecutive iniziando a h(k) finché
    - Troviamo una cella i vuota o contenente DEFUNCT
    - Abbiamo esaminato N celle
  - Memorizziamo (k, o) nella cella i

Scansione quadratica

- In caso di collisione
  - Si cerca la prima cella libera nelle posizioni (h(k) + f(i)) mod N, per i = 1, … , N-1
  - *Scansione quadratica:* $f(i) = i^2$
- Può dar luogo ad agglomerazione in porzioni della tabella lontane dalla posizione h(k)
  - Agglomerazione secondaria
- Se N è primo
  - Si ha garanzia di trovare una cella vuota se fattore di carico < 0.5

Hashing doppio

- Usa una seconda funzione hash d(k)
- Inserisce la coppia nella prima cella disponibile nelle posizioni (h(k) + jd(k))mod N, j = 0 … N – 1
- N-1 primo garantisce che tutte le celle siano esaminate
- Scelta tipica:
  - d(k) = q – k mod q
    - q < N
    - q primo
    - Possibili valori di d(k): 1 … q

q = 7

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Rehashing

- Fattore di carico (load factor)
  - Frazione di riempimento della tabella = (numero chiavi presenti)/N
  - E’ buona norma tenerla sotto 1 per evitare degrado nelle prestazioni
  - Meglio se non oltre 0.5 nel caso di scansione lineare
    - A causa dell’agglomerazione
  - Non oltre 0.75 – 0.9 se si usano liste di trabocco
  - Non oltre 0.5 se si usa hashing doppio

Rehashing/cont.

- Liste di trabocco
  - Non strettamente necessario
  - Tuttavia: degrado considerevole delle prestazioni per fattori di carico superiori a 1
- Scansione lineare
  - Necessario quando la tabella è piena
- Due aspetti
  - Hash code associati alle chiavi –> non vanno ricalcolati
  - Funzione di compressione → va riapplicata a tutti gli elementi presenti nella tabella per tener conto del nuovo valore di N
  - Effettuare il rehashing sparpaglia nuovamente gli elementi all’interno della tabella hash

Esercizio

- Si supponga che una tabella hash sia inizializzata con un array di dimensione costante
- Si supponga di effettuare rehashing ogni volta che il fattore di carico supera 0.5
- Rehashing → Raddoppio la dimensione dell'array
- Si consideri una successione di n inserimenti
  - Qual è il costo complessivo delle operazioni di rehashing dovute agli n inserimenti?
  - Qual è il costo medio (ammortizzato) di rehashing per elemento?

Prestazioni tabelle di hashing

Operazione	Lista	Tabella hash Atteso	Caso peggiore
get	O(n)	O(1)	O(n)
put	O(n)	O(1)	O(n)
remove	O(n)	O(1)	O(n)