Una mappa modella una collezione di elementi a cui vi si accede usando una CHIAVE

- Gli elementi della mappa sono considerabili CHIAVE - VALORE, <K, V>

alcuni metodi:

- get (K) —> value derivato da chiave
- put (K,V) —> aggiunge K e la associa al value
- remove (K) —> rimuove key e value associato
- keySet () restituisce insieme iterabile di tutte le key
- values () restituisce insieme iterabile di tutti i valori
- size () restituisce num elementi coppie <K, V>

è possibile implementare la mappa come SCL, con possibilità di scorrere ambo le parti, o come lista.

Si può usare una MULTIMAPPA che consiste nella stessa coppia <K, V>, in cui il valore è UN INSIEME DI ELEMENTI

TABELLE HASH

Struttura dati che implementa una mappa, che rende più efficiente le operazioni set(), put() e remove().

Se si usa un array l'accesso, data la posizione, è COSTANTE, poiché l'indice dell'array è la posizione diretta della locazione in memoria.

Le chiavi possono essere numeri interi, è possibile quindi implementare una tabella HASH, in cui vi è un'associazione CHIAVE - INTERO

> la funzione hash è una specie di SCATOLA NERA, che prende in ingresso una chiave e dà in uscita un intero.
> la funzione ha costo COSTANTE, e sarà SOLO in un VERSO!!!
> ci sono molte chiavi relativi ad uno stesso indice, MA UN SOLO INDICE per ogni chiave.

Nello scenario che si considera l'array è mai pieno.

La distribuzione dell'Hash dovrebbe essere

- DISTRIBUITA
- implementata tale che le collisioni (più chiavi un indice) siano MINIME
- NON DIPENDERE, a colpo d'occhio, dal valore input!

FUNZIONE HASH

- Mappa chiavi in un intervallo [0...N-1] di interi
  - h(x) = x mod N
  - h(x) è valore hash di chiave x

- è importante saper gestire collisioni! (es hashing come modulo, -5 e 5 stesso valore hash)

- come trasformare chiavi NON intere ad indici?
  oggetto → valore intero, anche 64 bit → indice hash

  ↳ hashCode - già "visto" in Java -

ES. SSN. (us.)
Si può indicizzare l'SSN con l'ultime 4 cifre usuali.
ALTA collisione.

Italia → codice fiscale (STRINGA)
S. fa passi:

- dato alfanumerico → intero (hashCode non limitato da array)
- funzione di COMPRESSIONE → <interi> : [0...N-1]

- h(x) = compress(hashCode(x))

Usata per fare si che ad ogni hashCode (sequenze anche di 32 bit, che non potrebbero quindi essere inserite 1:1 nella tabella) corrisponde 1 indice per la tabella

- oggetti con STESSA CHIAVE hanno STESSO HASHCODE
- la funzione sia QUASI-INIETTIVA, cercando di ridurre al minimo le collisioni

Es. Banale

- chiave ABCD
- hashCode → intero a 32 bit ottenuto mediante codifica ASCII con ciascun carattere.

ABCD → 1094861636 (codifica intero del binario)

funzione di compressione → last 4 char 1636

POSSIBILE HASHCODE

- standard è l'indirizzo di memoria di un oggetto che è interpretato come chiave. In generale NON è una buona idea oggetti uguali in memoria DIVERSE.
- conversione in intero
- somma di componenti: alto numero collisioni

hashCode ≠ hash function

hashCode key → interi

hash interi → $x \in [0, N-1]$

HASH CODE CON POLINOMI

- adatto a chiavi di lunghezza variabile

Si PARTIZIONA la chiave in componenti di lunghezza fissa b (8, 16, 32 bit...) come interi.

ES

10010000,00001000, $b = 8$

$a_0 = 10010000 \rightarrow 144$
$a_1 = 00001000 \rightarrow 8$

Siano $a_0, a_1, \dots, a_{n-1}$ le componenti:

- Si calcola $p(z) = a_0 + a_1 z + \dots + a_{n-1} z^{n-1}$

Si trascurano eventuali overflow

per l'esercizio precedente, ad esempio

$p(10) = 144 + 8 \cdot 10 = \boxed{224}$

Uno dei migliori hashCode

Ci sono valori migliori di b e z

QUANTO COSTA CALCOLO hashCode?

n, in questo caso, è num bit/scala
il costo della somma è lineare, anche se sono considerabili costi lineari

REGOLA DI HORNER

il calcolo $p(z)$ è lineare

$$p(z) = \sum_{i=0}^{n-1} a_i z^i$$

per l'iesimo termine $a_i z^i$ saranno necessarie i operazioni, se facessimo così sarebbe

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \text{ operazioni.}$$

Regola di Horner

$$p_0(z) = a_{n-1}$$
$$p_1(z) = z p_0 + a_{n-2} = a_{n-2} + a_{n-1} z$$
$$p_2(z) = z(p_1) + a_{n-3} = a_{n-3} + a_{n-2} z + a_{n-1} z^2$$
$$\vdots \qquad \vdots$$
$$p_{n-1}(z) = z p_{n-2}(z) = a_0 + a_1 z + \dots + a_{n-1} z^{n-1}$$

il costo dell'algoritmo è $O(n)$

- $P(h_2(x) = i) = 1/N$
- Si assumono n chiavi nella tabella hash
- Calcolare valore atteso del numero di chiavi che vengono assegnate a i

$$\frac{m}{N}$$

L'universo di tutte le possibili chiavi è GENERALMENTE MOLTO MAGGIORE delle possibili hash

ES $Key \in \mathbb{R}^n \rightarrow index \in [0, 1000]$

Si hanno COLLISIONI, poiché la funzione di compressione non è BIIETTIVA.

GESTIONE COLLISIONI

2 possibilità:

- **Liste di trabocco**: Ogni cella dell'array è una lista, alla collisione si aggiunge in coda.
  CONTRO:
  costo memoria aggiuntivo
  serve ricerca di chiave → O(n)

- **Indirizzamento aperto**: Si cerca una nuova posizione libera nella tabella, occupando altre posizioni di indici
  - non si usa altro spazio
  - serve tipo di dato aggiuntivo.

  → 2 casi possibili:
  - SCANSIONE LINEARE.
  - SANSIONE QUADRATICA

SCANSIONE LINEARE

idea di base che si inserisce la coppia che genera collisione nella prossima posizione $(i = (i+1) \bmod N)$

Ciò porta inevitabilmente a CLUSTERING, e se si ha la tabella piena (o quasi-piena, vedi fattore di carico) vi è il costo del raddoppio della tabella (REHASHING)

remove(key k):

per gestire rimozioni: si può usare l'oggetto DEFUNCT, che rende la gestione di eventuali ricerche più semplice.

put(k, o):

- se tabella piena: raddoppio
- else scorri da $h(k)$ finché non trovo spazio vuoto o DEFUNCT

SCANSIONE QUADRATICA

MOLTO SIMILE alla scansione lineare, ma genera meno clustering in quanto gli oggetti si "disperdono" meglio

Si cerca la prima cella libera nella posizione $(h(k) + i^2) \bmod N$ per $i = 1, \dots, N-1$

Se N è primo è garantita la possibilità di trovare una cella libera con un fattore di carico < 0.5

HASHING DOPPIO

Si utilizza una seconda funzione hash $d(k)$, di regole

- $d(k) = q - k \bmod q$
  - q numero primo
  - $q < N$
  - $d(k) \in [1, \dots q]$

e si inserisce la coppia nella prima posizione disponibile a

$$\big(h(k) + j\,d(k)\big) \bmod N \qquad j = 0, 1, \dots N-1$$

REHASHING e PROBLEMA del RADDOPPIO

Fattore di carico λ (load factor)

è la frazione di riempimento della tabella

= numkeys / N

ed è un ottimo strumento per valutare le prestazioni della tabella hash, e decidere per un RADDOPPIO

$\lambda < 1$ (buona norma

$\lambda < 0.9$ {per liste di trabocco

$\lambda < 0.5$ { per scansione lineare e quadratica, a causa dell'agglomerazione che porta a rallentamenti

per il raddoppio va ricalcolata solo la funzione di compressione per l'hash.
Il raddoppio "sparpaglia" di nuovo le chiavi nella tabella, secondo quello che si è detto sulla distribuzione hashCode e hash Function.

RADDOPPIO

Si suppone di partire da un array di dimensione costante. Si vogliono aggiungere n elementi, e al riempimento di essa si raddoppia la lista, e si copiano i valori del vecchio array.

QUANTO COSTA L'OPERAZIONE?

- dato un rehashing, numero di chiavi aggiunte $= c \cdot 2^i$ — i = i-esimo rehashing
- Numero minimo di rehashing per far spazio a n valori $c \cdot 2^i \geq n$

$$i = \log_2\left(\frac{n}{c}\right)$$

$$c \cdot \sum_{i=1}^{\log_2(\frac{n}{c})} 2^{i-1} = c \sum_{i=0}^{\log_2(\frac{n}{c})-1} 2^i = \frac{2^{\log(\frac{n}{c})} - 1}{2^0 - 1} \therefore n - c$$

costo rehashing $O(n)$