TABELLE HASH

Angelo Di Iorio
Università di Bologna

Esercizio 1

- Implementare in Java una classe `HashTable` per gestire una tabella Hash in cui sia le chiavi che i valori sono stringhe.
- Metodi (costruttore + metodi dizionario):
  - `HashTable(Integer capacity)`
  - `String lookup(String key)`
  - `void insert(String key, String value)`
  - `void remove(String key)`

- Aggiungere un metodo `print()` utility per stampare il contenuto della tabella

  - **Hashing: modular hashing (divisione, regola di Horner)**
  - **Collisioni: memorizzazione esterna (liste di trabocco)**

Una possible implementazione

- Definiamo una classe `HashTableEntry` per memorizzare le coppie chiave-valore contenute nella tabella
- Definiamo una classe `HashListaDiTrabocco` per memorizzare la lista di chiavi che collidono, che espone i metodi:
  - `public HashtableEntry find(String key)`
  - `public void add(String key, String value)`
  - `public void remove(String key)`
- `HashTableEntry` contiene un vettore di `HashListaDiTrabocco`
- La posizione nel vettore di una chiave *k* è calcolata applicando una *funzione hash*

Funzioni hash

- Definiamo una classe `HashCalculator` che implementa due metodi per calcolare funzioni hash
  - `badhash(String k, Integer m)` // la posizione nell'alfabeto del primo carattere in *k* (lettere minuscole)
  - `horner(String k, Integer m)` // divisione, Regola Horner

- Informazioni utili:
  - Il metodo `charAt(int i)` della classe `String` ritorna il carattere i-esimo della stringa
  - Le operazioni di casting tra `int` a `char` permettono di convertire in intero nel corrispondente carattere in ASCII (e viceversa)
  - 97 è il codice ASCII della lettera 'a'

- Diversi metodi per calcolare hash: *estrazione*, *XOR*, etc.

Hashing modulare – regola di Horner

- Il valore è calcolato come resto della divisione di `ord(bin(k))` per `m`
- Un polinomio di grado `q` può essere valutato con `q` prodotti e `q` addizioni
- Operazione di modulo ad ogni iterazione per ridurre 'overflow'

$$p(x) = a_q x^q + a_{q-1} x^{q-1} + \cdots + a_1 x + a_0 = x(\cdots(x((a_a x) + a_{a-1})\cdots) + a_1) + a_0$$

```
integer H(ITEM[] k, integer ℓ)
    integer b ← ord(k[1])
    for j ← 2 to ℓ do
        b ← ((b · 64) + ord(k[j])) mod m
    return b
```

lunghezza chiave

base (in pratica scelto anche per migliorare distribuzione chiavi e performance)

Esercizio 2

- Implementare una classe Java per gestire una tabella Hash in cui sia le chiavi che i valori sono stringhe.
- Metodi (dizionario):
  - `HashTable(Integer capacity)`
  - `String lookup(String key)`
  - `void insert(String key, String value)`
  - `void remove(String key)`

- Aggiungere un metodo `print()` utility per stampare il contenuto della tabella

  - **Hashing: modular hashing (divisione, regola di Horner)**
  - **Collisioni: memorizzazione interna, scansione lineare a passo unitario**

Una possible soluzione

- Usiamo le classi `HashTableEntry` e `HashCalculator` definite per l'esercizio precedente

- Implementiamo una classe `HashTableInterna` che memorizza un vettore di `HashTableEntry` ed implementa un metodo per eseguire la scansione:

```
private Integer scan(String key)
```

- Questo metodo è usato dalle operazioni di *lookup*, *insert* e *delete*

Scansione lineare: $H(k,i) = (H(k) + h \cdot i) \bmod m$

Esercizio 3

- Modificare la classe precedente per usare un metodo di scansione quadratica (con `h=1`), piuttosto che lineare
- (le altre specifiche restano invariate)

Scansione lineare: $H(k,i) = (H(k) + h \cdot i) \bmod m$

Scansione quadratica: $H(k,i) = (H(k) + h \cdot i^2) \bmod m$