Esercitazione 5
Hashing

Corso di Fondamenti di Informatica II

BIAR2 (Ing. Informatica e Automatica) e BSIR2 (Ing. dei Sistemi)

A.A. 2013/2014

15 Novembre 2013

Sommario

Scopo della esercitazione è di implementare e paragonare differenti tecniche di gestione delle collisioni in una tabella hash.

1 Tabelle hash

Le chiavi di una tabella hash vengono memorizzate in un array A di lunghezza N (*bucket array*). Ogni cella dell'array è considerata come un bucket (secchio) e contiene una chiave. Il valore N definisce la capacità dell'array. Le celle occupate conterranno una chiave, mentre quelle libere conterranno `null`.

2 La funzione di hashing

La *funzione di hashing* è una funzione h che trasforma ogni chiave k in un numero intero compreso tra 0 e $N-1$, dove N è la capacità della tabella hash. L'idea di questo approccio è quella di memorizzare la chiave k nel bucket array in posizione $h(k)$. Naturalmente può capitare che due chiavi differenti possano avere lo stesso valore di hashing ed in tal caso si genera una *collisione* (vedi Sez. 3).

Il codice fornito a supporto (nel file `HashTable.java`) contiene già una semplice funzione di hashing, `sommaHash`, che si applica a chiavi di tipo `String`. Si aggiunga una ulteriore funzione di hashing `polyHash(String key)` che si basi sul concetto di codice hash polinomiale visto a lezione, usando la regola di Horner per il calcolo di polinomi.

3 Gestione delle collisioni

Quando per due chiavi distinte k_1, k_2 la funzione di hashing genera uno stesso valore numerico si ha una collisione. In questo caso, solo una delle due chiavi potrà essere memorizzata nella posizione dell'array identificata dal suo hashcode,

mentre l'altra dovrà essere memorizzata in una posizione differente, in modo però da poterla ritrovare. Per ottenere questo risultato vengono utilizzate varie tecniche come il *linear probing*, il *quadratic probing* (non presente nelle slides del corso, ma discussa nel libro di testo) e il *double hashing*.

Costruttore. Nella classe `HashTable`, si implementi innanzitutto un costruttore che dato un parametro `capacity` di tipo `int` allochi un bucket array vuoto di dimensione `capacity`.

Collisioni con linear probing. Nella classe `HashTable`, si implementi il metodo `putLP(String key)` che implementi un inserimento nella tabella hash basato sul *linear probing* per la gestione delle collisioni e sulla `polyHash` come funzione di hashing.

Collisioni con quadratic probing. Nella classe `HashTable`, si implementi il metodo `putQP(String key)` che implementi un inserimento nella tabella hash basato sul *quadratic probing* per la gestione delle collisioni e sulla `polyHash` come funzione di hashing. Il quadratic probing è molto simile al linear probing: la sequenza di probe è generata, anziché con un contatore $i \in [0, N-1]$, con un contatore $i^2, i \in [0, N-1]$.

Collisioni con double hashing. Nella classe `HashTable`, si implementi il metodo `putDH(String key)` che implementi un inserimento nella tabella hash basato sul *double hashing* per la gestione delle collisioni, sulla `polyHash` come funzione di hashing primaria e sulla `sommaHash` come funzione di hashing secondaria.

4 Confronto

Si scriva una classe `Test` contenente un metodo `main` in cui, per dei fissati valori di *carico*, C, e di *capacità*, N (ad esempio $C = 60$, $N = 127$):

- si creino C chiavi casuali di tipo `String`, utilizzando il metodo statico `randomKey()` della classe `HashUtil`;
- si inseriscano queste chiavi in tre diverse tabelle di hash di dimensione N, usando in ciascuna di esse un diverso metodo di gestione delle collisioni;
- si memorizzino, in tre array ausiliari, il numero cumulativo di probe effettuati dopo ciascun inserimento in ognuna delle tre tabelle;
- si vada a visualizzare, utilizzando il metodo statico `show` della classe `HashUtil`, l'efficienza dei vari metodi di gestione delle collisioni. Il metodo `show` deve prendere in ingresso una `ArrayList<int[]>` di tre elementi – i tre array costruiti al passo precedente.

Sperimentare l'efficienza dei vari metodi al variare di C ed N.

Riferimenti bibliografici

[1] M. T. Goodrich and R. Tamassia. *Strutture dati e algoritmi in Java.* Zanichelli, 2007.