Esercizi Capitolo 7 - Hash

Alberto Montresor

19 Agosto, 2014

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.
Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

1 Problemi

1.1 Inserimento in tabella hash (Esercizio 7.2 del libro)

Si consideri una tabella hash di dimensione $m = 11$ inizialmente vuota. Si mostri il contenuto della tabella dopo aver inserito nell'ordine, i valori 33, 10, 24, 14, 16, 13, 23, 31, 18, 11, 7. Si assuma che le collisioni siano gestite mediante indirizzamento aperto utilizzando come funzione di hash $h(k)$, definita nel modo seguente:

$$h(k) = (h'(k) + 3i + i^2) \bmod m$$
$$h'(k) = k \bmod m$$

Mostrare il contenuto della tabella al termine degli inserimenti e calcolare il numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella.

Soluzione: Sezione 2.1

1.2 Scansione lineare e quadratica (Esercizio 7.4 del libro)

Si forniscano procedure per realizzare i metodi di scansione lineare e quadratico.

Soluzione: Sezione 2.2

1.3 Confronto file (Esercizi 7.6, 7.9 del libro)

Supponete di avere un albero di directory, contenenti un numero n di file, alcuni dei quali potrebbero essere replicati una o più volte, sotto nomi e percorsi diversi. Il vostro compito è elencare tutti i file replicati (ovvero, con lo stesso contenuto e la stessa lunghezza). Proponente un metodo efficiente ($O(n)$) per risolvere questo problema.

Soluzione: Sezione 2.3

1.4 Liste di trabocco ordinate (Esercizio 7.10 del libro)

Supponiamo di mantenere ordinate le liste di trabocco. Quali sono i vantaggi e/o svantaggi di tale scelta in termini di efficienza?

Soluzione: Sezione 2.4

1.5 Hashing perfetto

Problema: dato un insieme di chiavi statiche (che non cambiano nel tempo), realizzare una tabella hash (o una struttura simile) il cui costo di ricerca sia $O(1)$ nel caso pessimo.
Possibili utilizzazioni: per esempio, la tabella dei file di un cd-rom.

Soluzione: Sezione 2.5

2 Soluzioni

2.1 Inserimento in tabella hash (Esercizio 7.2 del libro)

La tabella hash è composta nel modo seguente:

0	1	2	3	4	5	6	7	8	9	10
33	23	24	14	11	16	13	18		31	10

Il valore 18 non può essere inserito, perché la funzione di scan scorre 11 posizioni (molte ripetute) e non trova l'unica cella libera rimasta (la 8). Il numero di accessi medi è 1.2 per i valori che hanno trovato collocazione.

2.2 Scansione lineare e quadratica (Esercizio 7.4 del libro)

È sufficiente modificare la funzione scan(). La versione per la scansione lineare è la seguente:

```
integer scan(ITEM k, boolean insert)
integer c ← m                                   % Prima posizione deleted
integer i ← 0                                   % Numero di ispezione
integer j ← H(k)                                % Posizione attuale
while A[j] ≠ k and A[j] ≠ nil and i < m do
    if A[j] = deleted and c = m then c ← j
    i ← i + 1
    j ← (H(k) + h · i) mod m
if insert and A[j] ≠ k and c < m then j ← c
return j
```

Quella per la scansione quadratica è la seguente:

```
integer scan(ITEM k, boolean insert)
integer c ← m                                   % Prima posizione deleted
integer i ← 0                                   % Numero di ispezione
integer j ← H(k)                                % Posizione attuale
while A[j] ≠ k and A[j] ≠ nil and i < m do
    if A[j] = deleted and c = m then c ← j
    i ← i + 1
    j ← (H(k) + h · i^2) mod m
if insert and A[j] ≠ k and c < m then j ← c
return j
```

2.3 Confronto file (Esercizi 7.6, 7.9 del libro)

È possibile calcolare una funzione hash sul contenuto di tutti i file. Una possibile funzione per questo scopo potrebbe essere SHA-1, una funzione hash crittografica che produce chiavi di 160 bit.
Questo richiede la lettura di n file. A questo punto, si confrontano le chiavi SHA-1(in tempo $O(n \log n)$, oppure in tempo $O(n)$ inserendole in una tabella hash), e si verificano solo file che hanno lo stesso valore di chiave SHA-1. La possibilità di collisione con 160 bit è molto bassa.

2.4 Liste di trabocco ordinate (Esercizio 7.10 del libro)

Il vantaggio è che quando si cerca un elemento, è possibile fermarsi non appena si trova un elemento maggiore o uguale dell'elemento cercato, invece di arrivare in fondo alla lista in caso di assenza.

L'operazione di inserimento, che richiede comunque di cercare un elemento nella lista, può essere implementata allo stesso costo.

2.5 Hashing perfetto

Alcune definizioni:

- sia p un numero primo;
- sia $Z_p = [0 \dots p-1]$;
- sia $Z_p^* = [1 \dots p-1]$;
- m una costante che rappresenta la dimensione della tabella hash (di primo livello).

p deve essere sufficientemente grande da fare sì che Z_p possa contenere l'universo delle possibili chiavi: $U \subseteq Z_p$.
Definiamo una *classe* di funzioni hashing in questo modo:

$$\mathcal{H}_{p,m} = \{h_{a,b}^{p,m} | a, b \in Z_p^*\}$$

dove

$$h_{a,b}^{p,m}(k) = ((ak + b) \bmod p) \bmod m$$

Questo insieme è un classe *universale* di funzioni hash: questo significa che per ogni coppia di chiavi distinte k, l, il numero di funzioni hash $h \in \mathcal{H}_{p,m}$ tali per cui $h(k) = h(l)$ è al massimo $\mathcal{H}_{p,m}/m$. In altre parole, non esistono coppie di chiavi che generano più collisioni di altre.

Teorema 1. Se memorizziamo n chiavi in una tabella hash con $m = n^2$ slot utilizzando una funzione hash h scelta a caso da una classe universale di funzioni hash, la probabilità che si verifichi una collisione è minore di 1/2.

Dimostrazione. Ci sono $\binom{n}{2}$ coppie di chiavi che possono collidere, ognuna con probabilità $1/m$, se h è scelta a caso da una famiglia universale di funzioni hash.
Quindi, detta X una variabile casuale che conta il numero di collisioni, abbiamo:

$$\begin{aligned} E[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\ &< 1/2 \end{aligned}$$

Per la disequezione di Markov, $Pr\{X \geq t\} \leq E[X]/t$, con $t = 1$, si conclude la dimostrazione.
Notate che n può essere troppo grande per essere portato al quadrato; per questo motivo, si utilizza uno schema a due livelli.
Si utilizza una prima funzione tabella h grande quanto il numero di chiavi ($n = m$); questo genererà un certo numero di collisioni; sia n_j il numero di chiavi in collisione per l'indice j (con $1 \leq j \leq m$).
Invece di puntatore ad una lista di concatenamento, i puntatori della tavola hash di primo livello puntano a n tavole hash di secondo livello; la tavola dell'indice j è grande n_j^2.
Se la funzione hash di primo livello è stata ben scelta, i valori n_j sono relativamente bassi. A questo punto, si procede in questo modo: per ogni indice $1 \leq j \leq m$, si sceglie a caso una funzione h_{p,n_j^2} da $\mathcal{H}_{p,n_j^2}$ e la utilizziamo per gestire una tabella hash $A_j[1 \dots n_j^2]$. Se nella tabella secondaria non ci sono collisioni, memorizziamo le caratteristiche di h_{p,n_j^2} assieme alla tabella hash A_j. Altrimenti, proviamo una nuova funzione hash casuale finché non ne troviamo una senza collisioni.
È possibile dimostrare, ma non lo faremo qui, che scegliendo a caso una funzione da una classe universale per l'hashing di primo livello, la dimensione totale attesa delle sottotabelle è

$$E\left[\sum_{j=0}^{n-1} n_j^2\right] < 2n$$