Informationsteknologi
Tom Smedsaas 16 februari 2006

Hashmetoder

Detta papper innehåller en introduktion till hashtabeller och vi tar endast upp så kallad *länkad* kollisionshantering.

Inledning

Hashtabeller är en implementation av en abstrakt datatyp som lagrar poster med unika *nycklar* och som stöder operationer för *sökning*, *inläggning* och, ofta också, *uttag*.

Struktur	Tidskomplexitet för	
	sökning	*inlägg* (när platsen känd)
länkad lista	$\Theta(n)$	$O(1)$
sorterad array	$\Theta(\log n)$	$\Theta(n)$
binärt sökträd	$\Theta(\log n)$	$O(1)$
hashtabell	$O(1)$	$O(1)$

I hashtabeller kan man således söka, lägga in och ta bort poster med konstant tid *oavsett hur många poster som är lagrade i tabellen*!

Man kan bevisa att det, i genomsnitt, krävs $\Omega(\log n)$ operationer att lokalisera en given nyckel bland n om den enda tillåtna operationerna på nycklarna är storleksjämförelser. Med den begränsningen kan man säga att binärsökning i array och sökning i binärt (balanserat) sökträd är optimala – det går helt enkelt inte att göra bättre.

Knepet för att komma vidare är att använda nyckeln för att bättre förutsäga platsen. Om man t.ex. har en array med 500 sorterade heltal mellan 200 och 1000 och letar efter 220 så är det dumt att, som i binärsökning, börja att titta i mitten. Genom att i stället använda interpolationen

$$\frac{220 - 200}{1000 - 200} \cdot 500 = 12$$

som startpunkt kommer man förmodligen mycket närmare. Detta kan upprepas och leder fram till en metod som brukar kallas *interpolationssökning* och som tar $O(\log \log n)$ operationer. För alla praktiska ändamål kan detta betraktas som konstant. Tyvärr så kräver fortfarande inlägg $\Theta(n)$ operationer vilket gör att denna metod inte passar för mer dynamiska mängder.

Direktadressering

Vid interpolationssökning använder man ett sätt att lägga in nycklar (stoppas in i storleksordning) och ett annat att söka (beräkna plats med en interpolation). Man kan använda samma metod för att beräkna plats vid lagring som vid sökning. I ovanstående exempel med de 500 heltalen i intervallet $[200, 1000]$ skulle man kunna använda en array $A[801]$ och lagra nyckel k på plats $k - 200$. Varje nyckel får på detta sätt en unik plats. I själva verket behöver vi ej ens lagra nyckeln – bara information om att den finns med. Det räcker alltså med en array med logiska värden. Om värdet på plats k är *true* så finns nyckel $k + 200$ med, annars ej. I denna metod krävers således sökning, lagring och borttagning endast en subtraktion eller addition samt en kopiering av värdet. Tyvärr så förutsätter den att varje *möjlig* nyckel får en egen, unik adress vilket betyder att tabellen som krävs blir orimligt stor i de flesta tillämpningar. (Antag t.ex. att nycklarna är variabelnamn i ett programmeringsspråk. Hur många möjliga namn finns det?)

Hashtabeller

För att försöka hålla kvar effektiviteten i direktadressering men få en tabell som dimensioneras efter *förväntat* i stället för *möjligt* antal nycklar, släpper man kravet på att varje nyckel skall garanteras en egen plats.

Precis som vid direktadressering används en array T med m platser samt en funktion $h(k)$ som avbildar mängden nycklar på intervallet $[0, m - 1]$. Arrayen T kallas *hashtabell* och funktionen h *hashfunktion*. Tabellstorleken m väljes efter förväntat antal nycklar. Det värde (den adress) som en nyckel får av hashfunktionen h kallas dess *hashadress* eller *hashkod*.

Två väsentliga frågor finns: hur skall hashfunktionen se ut och hur skall man göra om två olika nycklar får samma plats?

Tills vidare antar vi att nycklarna är heltal. Som hashfunktion h kan vi då använda modulofunktionen

$$h(k) = k \bmod m$$

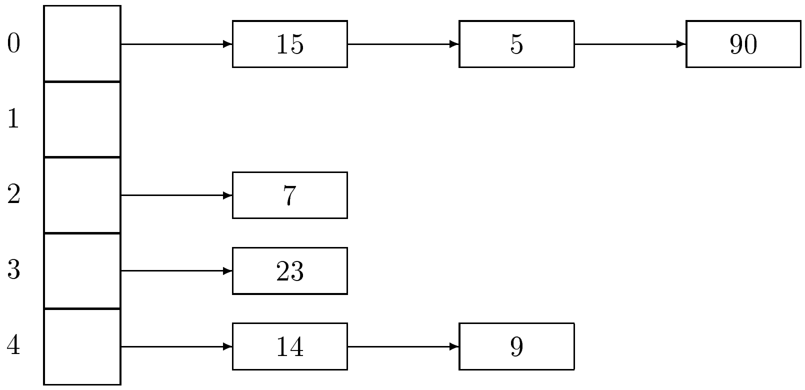
Modulo-operationen garanterar ju att funktionsvärdet kommer i intervallet $[0, m - 1]$.

Om två olika nycklar k_1 och k_2 får samma plats så kallas detta för en *kollision*. Ett sätt att hantera detta är att länka samman alla element som får samma i plats i en länkad lista. Vanligen låter man då hashtabellen T innehålla pekare till listelement. Detta kallas *länkad kollisionshantering*.

Exempel

Antag att nycklarna 14, 15, 5, 7, 90, 23 och 9 skall läggas in i en hashtabell med 5 platser ($m = 5$) och med länkad kollisionshantering.

k	14	15	5	7	90	23	9
$h(k)$	4	0	0	2	3	3	4



Om vi antar att varje inlagd nyckel är lika sannolik att bli eftersökt så kräver en lyckad sökning i denna tabell

$$s = \frac{4 \cdot 1 + 2 \cdot 2 + 1 \cdot 3}{7} = \frac{11}{7} = 1.57 \tag{1}$$

i genomsnitt.

Om vi inför en misslyckad sökning antar att varje lista är lika sannolik att bli genomsökt så kräver en misslyckad sökning

$$u = \frac{4 + 1 + 2 + 2 + 3}{5} = \frac{12}{5} = 2.4 \tag{2}$$

i genomsnitt (att leta i en tom lista kräver ett försök – för att upptäcka att listan är tom)

Analys av länkad kollisionshantering

För att analysera hashmetoder antar vi att en given nyckel har lika stor sannolikhet att hamna på var och en av de m platserna.

Om n nycklar lagrats så är de m listorna i genomsnitt n/m långa. För att upptäcka att en given nyckel *inte* finns (s.k. *misslyckad* sökning) så krävs då i genomsnitt

$$u(n, m) = 1 + n/m \tag{3}$$

försök (att leta i en tom lista kräver, som sagt, ett försök).

Sökning av nyckel som finns (*lyckad* sökning) är något svårare att analysera eftersom genomsnittet skall bildas över de n nycklarna - inte över antalet tabellplatser. Antagandet är att varje nyckel söks med lika stor sannolikhet. Vi kan här använda samma observation som vid analys av binära sökträd: *det krävs lika många försök att finna en existerande nyckel som det krävdes för att lägga in den nyckeln* vilket är samma som *antalet försök som krävdes för att söka den nyckeln innan den var inlagd.*

Sålunda är det genomsnittliga arbetet att söka en existerande nyckel lika stort som arbetet för att bygga upp tabellen

$$\begin{aligned}
s(n,m) &= \frac{1}{n}\left((1+\frac{0}{m})+(1+\frac{1}{m})+(1+\frac{2}{m})+\cdots+(1+\frac{n-1}{m})\right) \\
&= \frac{1}{n}\left(n+\frac{1}{m}\sum_{i=0}^{n-1} i\right) \\
&= 1+\frac{n(n-1)}{2mn} \\
&= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{1}{2m} \approx 1+\frac{\alpha}{2} \qquad (4)
\end{aligned}$$

där $\alpha = n/m$ är tabellens *fyllnadsgrad.*

Hashfunktioner

Funktionen

$$h(k) = k \bmod m$$

som vi har använt i exemplet är inte alls dålig under förutsättning att m är ett primtal. Om m inte är ett primtal blir funktionen mer känslig för systematik i nycklarna.

En mer generell funktion är

$$h(k) = ((c_1 k + c_2) \bmod p) \bmod m$$

där $p > m$ är ett stort primtal, $p > 2^{20}$ (t. ex. $p = 1048583$, c_1 ett positivt heltal mindre än p och c_2 ett icke-negativt heltal mindre än p. Konstanten c_2 väljs ofta till 0. Olika c_1 ger helt olika hashfunktioner!

Om nyckeln är en sträng $s_1 s_2 s_3 \cdots s_k$ så beräkna ett tal t.ex. genom

$$(((s_1 \cdot 37 + s_2) \cdot 37 + s_3) + \cdots) + s_k$$

och använder sedan någon av ovanstående hashfunktioner.