Hashmetoder

Detta papper beskriver så kallade hashtabeller. Det innehåller också en beskrivning av *ordnad hashning* som är en metod att förbättra prestanda vid misslyckade sökningar.

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*. Det finns två huvudprinciper för att hantera kollisioner:

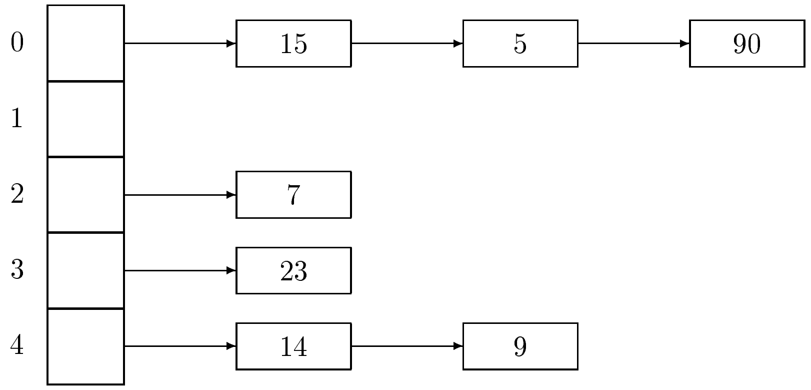
1. Element med samma hashadress länkas samman i en linjär lista. Vanligen låter man då hashtabellen T innehålla pekare till listelement. Detta kallas *länkad kollisionshantering*.

2. Välj en annan plats i samma tabell enligt någon algoritm. Olika algoritmer ger olika bra resultat. Detta kallas (ofta) för *öppen adressering*.

Länkad kollisionshantering

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.*

Öppen kollisionshantering

Alla nycklar (poster) i tabellen. Om en kollision uppstår provar man en annan plats. En enkel men ganska dålig metod är följande som brukar kallas *linjär* kollisionshantering:

Algoritm: Linjär kollisionshantering

```
Hash-insert( k, T )
    i ← h(k)
    while T[i]≠nil do
        i ← (i+1) mod m
    T[i] ← k
```

(En verklig implementation måste också hantera fallet att tabellen redan är full)

Exempel

Antag att vi har en tabell med $m = 11$ platser och använder hashfunktionen $h(k) = k \bmod m$ samt att vi har följande nycklar:

Nycklar k	14	7	96	6	412	3	16	25	22
$h(k)$	3	7	8	6	5	3	5	3	0

Om dessa läggas in i tabellen med algoritmen ovan erhålles följande tabell

Index	0	1	2	3	4	5	6	7	8	9	10
Innehåll	22			14	3	412	6	7	96	16	25
Antal försök	1			1	2	1	1	1	1	5	8

Totala antalet försök vid uppbyggnaden är 21 vilket innebär att sökning lyckad sökning i tabellen kräver $21/9 \approx 2.2$ försök.

Problemet med den linjära metoden är att de nycklar som en gång kolliderat kommer att använda samma strategi för att hitta en ledig plats dvs de kommer att fortsätta att kollidera med varandra. Detta gör att de bildas klumpar (s.k. kluster) med kolliderande nycklar. För att motverka detta kan man använda olika sätt att välja nästa plats för olika nycklar. Boken tar upp kvadratisk kollisionshantering men det finns egentligen ingenting som talar för denna. En mycket bra metod är däremot så kallad *dubbel hashning*. Idén är att i stället för att man vid kollision provar nästa, tar ett steg framåt där stegets längd beror på nyckeln via en ny funktion $g(k)$.

Algoritm: Dubbelhashning

Hash-insert(k, T)
  i ← h(k)
  while T[i]≠**nil do**
    i ← (i+g(k)) **mod** m
  T[i] ← k

(En verklig implementation måste också hantera fallet att tabellen redan är full)

Samma metod används naturligtvis för sökning.

Exempel

k	14	7	96	6	412	3	16	25	22
$h(k) = k \bmod 11$	3	7	8	6	5	3	5	3	0
$g(k) = (k \bmod 9) + 1$	6	8	7	7	8	4	8	8	5

Om dessa läggas in i tabellen med algoritmen ovan erhålles följande tabell

Index	0	1	2	3	4	5	6	7	8	9	10
Innehåll	3		16	14	22	412	6	7	96		25
Antal försök	3		2	1	4	1	1	1	1		6

Totala antalet försök vid uppbyggnaden är 21 vilket innebär att sökning lyckad sökning i tabellen kräver $20/9 \approx 2.0$ försök. (Det här exemplet gör inte metoden riktig rättvisa)

Är man säker på att alla platser besöks? Ja, om steget $g(k)$ och tabellstorleken m inte har några gemensamma primfaktorer vilket enklast garanteras genom att välja m till ett *primtal*.

Hur skall man välja $g(k)$? Vettiga steg är tal i intervallet $[1, m-1]$. En vanlig rekommendation är att använda $g(k) = (k \bmod m-2)+1$ som skall vara speciellt bra om förutom m, även $m-2$, är ett primtal (s.k. *primtalstvillingar*).

Likformig hashning - en idealiserad modell

Likformig hashning (eng. *uniform hashing*) är en modell för en idealiserad öppen hashmetod som vi använder för att analysera metoden.

När vi söker eller lagrar en nyckel provar vi en följd av olika adresser. Vi bygger analysen på följande antagande:

Vid varje försök har var och en av de oprövade platserna lika stor chans att bli vald.

Det förväntade antalet försök vid en misslyckad sökning är

$$\begin{aligned} u(m,n) &= \sum_{c>0} cPr(c \text{ försök}) \\ &= \sum_{c\geq 1} Pr(c \text{ försök}) + \sum_{c\geq 2} Pr(c \text{ försök}) + \sum_{c\geq 3} Pr(c \text{ försök}) + \cdots \\ &= Pr(\geq 1 \text{ försök}) + Pr(\geq 2 \text{ försök}) + Pr(\geq 3 \text{ försök}) + \cdots \\ &= \sum_{c\geq 1} Pr(\geq c \text{ försök}) \end{aligned}$$

För att det skall bli minst c försök måste det bli kollision i de $c-1$ första försöken

$$Pr(\geq c \text{ försök}) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \cdots \cdot \frac{n-(c-2)}{m-(c-2)} \approx \left(\frac{n}{m}\right)^{c-1}$$

Detta ger

$$u(n,m) = \sum_{c\geq 1} \left(\frac{n}{m}\right)^{c-1} = \sum_{c\geq 0} \left(\frac{n}{m}\right)^{c} = \frac{1}{1-\frac{n}{m}}$$

Eftersom en sökning efter ett existerande element kräver lika många försök som det tog att lägga in det elementet vilket är tar lika lång tid som att söka elementet innan det är inlagt så blir antalet försök vid lyckas sökning

$$s(n,m) = \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1-\frac{i}{m}} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i}$$

Summan, som ju är besläktad med den harmoniska summan, kan approximeras med en integral:

$$\sum_{i=0}^{n-1} \frac{1}{m-i} \approx \int_0^n \frac{dx}{m-x} = \ln \frac{m}{m-n}$$

vilket ger

$$s(\alpha) = s(m,n) = \frac{m}{n} \ln\left(\frac{m}{m-n}\right) = \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right) \tag{5}$$

där $\alpha = n/m$ är tabellens fyllnadsgrad. Dessa funktioners värden är tabulerade nedan. Sista raden i tabellen innehåller de exakta värdena för $n = 100$. Denna visar att approximationer ger en liten överskattning av antalet erforderliga försök.

Metod	Lyckad sökning					Misslyckad sökning				
Fyllnadsgrad	50	75	85	90	95	50	75	85	90	95
Linjär	1.5	2.5	3.8	5.5	10.5	2.5	8.5	22.7	50.5	200.5
Dubbelhashning	1.4	1.9	2.2	2.6	3.2	2.0	4.0	6.7	10.0	20.0
Exakt för $m = 100$	1.4	1.9	2.1	2.4	2.9	2.0	3.9	6.3	9.2	16.8

Ordnad hashning

Som framgår av tabellen ovan så kräver misslyckad sökning betydligt fler försök än lyckad när fyllnadsgraden börjar bli stor. Den finns en enkel metod att råda bot på detta som brukar kallas *ordnad hashning*.

Vi utgår från dubbelhashning (även om man det går lika bra att använda vilken öppen metod som helst) och modifierar sökalgoritmen något:

Algoritm: Sökning i ordnad tabell

Hash-search(k, T)
    i ← h(k)
    while k < T[i] **do**
        i ← (i+g(k)) **mod** m
    if $T(i) = k$ **then** *lyckad*
            else *misslyckad*

Vi har alltså ändrat iterationsvillkoret; iterationen avbryts när det lagrade är mindre än eller lika med den sökta nycken. För att detta skall fungera måste dock tabellen byggas upp på ett särskilt sätt.

Antag att vi använder den vanliga inläggningsalgoritmen men lägger in nycklarna i storleksordning med den största först. Då kommer sökalgoritmen ovan att fungera eftersom vi provar exakt samma platser vid sökning som vid inlägg och allt som lades in före k var större än k.

Detta är naturligtvis inget realistiskt sätt att fylla tabellen i de flesta tillämpningar men man kan få samma effekt med följande algoritm:

Algoritm: Inlägg i ordnad tabell

Hash-insert(k, T)
    $i \leftarrow h(k)$
    while $T[i] \neq$ **nil do**
        if $T[i] < k$ **then** swap(k, $T[i]$)
        $i \leftarrow (i + g(k))$ **mod** m
    $T[i] \leftarrow k$

(En implementation måste också hantera fallen att nyckeln redan finns och att tabellen är full)

Om tabellen är ordnad så gäller att när vi söker efter en nyckel k så passerar vi bara nycklar som är större än k. Ovanstående inläggningsalgoritm minskar aldrig en nyckel på en plats.

Den slutliga ordningen är oberoede av inläggningsordning!

Förväntat antal försök vid inlägg och lyckad sökning är oförändrat men för misslyckad sökning väsentligt förbättrad:

$$u(m,n) = s(m,n+1)$$

Metod	Lyckad sökning					Misslyckad sökning				
Fyllnadsgrad	50	75	85	90	95	50	75	85	90	95
Linjär	1.5	2.5	3.8	5.5	10.5	2.5	8.5	22.7	50.5	200.5
Dubbelhashning	1.4	1.9	2.2	2.6	3.2	2.0	4.0	6.7	10.0	20.0
Ordnad hashning	1.4	1.9	2.2	2.6	3.2	1.4	1.9	2.2	2.6	3.2

Hashfunktioner

Funktionen

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

som vi genomgående har använt i exemplen 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.

Det finns mer avancerade hashfunktioner som kan sprida även mycket ojämnt fördelade nycklar jämnt. Ofta bygger dessa på någon form av bitvis xor-operation med slumptal.