Idén med hashning

Binärsökning i en ordnad vektor går visserligen snabbt, men sökning i en hashtabell är oöverträffat snabbt. Och ändå är tabellen helt oordnad (hash betyder ju hackmat, röra). Låt oss säga att vi söker efter Kalas i en hashtabell av längd 10000. Då räknar vi först fram *hashfunktionen* för *söknyckeln* Kalas och det ger detta resultat.

```
"Kalas".hashCode() -> 72260712
```

Hashvärdets rest vid division med 10000 beräknas nu

```
72260712 % 10000 -> 712
```

och när vi kollar hashtabellens index 712 hittar vi Kalas just där!

Hur kan detta vara möjligt? Ja, det är inte så konstigt egentligen. När Kalas skulle läggas in i hashtabellen gjordes samma beräkning och det är därför ordet lagts in just på 712. Hur hashfunktionen räknar fram sitt stora tal spelar just ingen roll. Huvudsaken är att det går fort, så att inte den tid man vinner på inbesparade jämförelser äts upp av beräkningstiden för hashfunktionen.

Komplexiteten för sökning

Linjär sökning i en oordnad vektor av längd N tar i genomsnitt N/2 jämförelser, binär sökning i en ordnad vektor log N men hashning går direkt på målet och kräver bara drygt en jämförelse. Varför drygt? Det beror på att man aldrig helt kan undvika *krockar*, där två olika namn hamnar på samma index.

Dimensionering av hashtabellen

Ju större hashtabell man har, desto mindre blir risken för krockar. En tumregel är att man bör ha femtio procents luft i vektorn. Då kommer krockarna att bli få. En annan regel är att tabellstorleken bör vara ett *primtal*. Då minskar också krockrisken, som vi ska se nedan.

Hashfunktionen

Egentligen skulle man vilja ha en *perfekt* hashfunktion, dvs en funktion som ger olika värden för olika söknycklar. I regel är dock detta inte praktiskt möjligt, eftersom det kräver

1. en hashtabell som har minst lika många platser som det finns söknycklar,
2. en perfekt hashfunktion som inte är för tidskrävande att beräkna,
3. att man lyckas hitta en perfekt hashfunktion (dom är sällsynta).

I vissa specialfall (t ex när man vill skapa en tabell över reserverade ord) kan det finnas anledning att försöka hitta en perfekt hashfunktion och det finns algoritmer för detta, t.ex. Cichelli's metod och FHCD-algoritmen i Drozdeks bok.

Men vi begränsar oss här till enklare metoder. Ofta gäller det först att räkna om en `String` till ett stort tal. I programmeringspråk gör man ingen skillnad på en bokstav och dess nummer i teckenuppsättningen (A=65, B=66, a=92), därför kan `ABC` uppfattas som `656667`. Det man då gör är att multiplicera den första bokstaven med 10000, den andra med 100, den tredje med 1 och slutligen addera talen. På liknande sätt gör metoden `hashCode()` i java men den använder 31 i stället för 100. Ur javadokumentaitonen;

…

```
    Returns a hashcode for this string. The hashcode for a String
object is computed as

              s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

using int arithmetic, where s[i] is the ith character of the string, n
is the length of the string, and ^ indicates exponentiation (…)
```

Om man vill söka på datum eller personnummer kan man använda det som stort heltal utan särskild hashfunktion. Exempel: sexsiffriga datum kan hashas in i hashvektorn med `990323 % size`. En olämplig storlek är 10000, ty
`990323 % 10000 --> 323`
och vi ser att endast 366 av de 10 000 platserna kommer att utnyttjas. Det säkraste sättet att undvika sådan snedfördelning är att byta 10000 mot ett närliggande *primtal*, till exempel 10007. Det visar sej nämligen att primtalsstorlek ger bäst spridning.

Krockhantering

Det naturliga är att lägga alla namn som hashar till ett visst index som en länkad *krocklista*. Om man har femtio procents luft i sin vektor blir krocklistorna i regel mycket korta. Krocklistorna bör behandlas som stackarna, och hashtabellen innehåller då bara toppekarna `top` till stackarna.
Den andra idén är att vid krock lägga posten på första lediga plats (*linear probing*). En fördel är att man slipper alla pekare. En nackdel blir att man sedan inte enkelt kan ta bort poster utan att förstöra hela systemet, vilket man kan lösa genom att markera poster som borttagna istället för att ta bort dom. Ett annat problem man brukar råka ut för här är *klustring*, man får stora klumpar med poster, vilket gör att det kan ta lång tid att hitta nästa lediga plats. Om man vid en krock på plats n istället väljer att titta på plats $n+1^2$, $n+2^2$, $n+3^2$ osv (*quadratic probing*) får man bättre fördelning av posterna.