Tema NR 4: Hashtabeller

Annika Svedin `ansv9785`
Felix Törnqvist `fetr0498`

12 februari 2017

Hashfunktion

En hashtabell är den mest effektiva datastrukturen när det gäller sökning. I en hashtabell finns en *nyckel* och ett *värde*. Med rätt nyckel kommer man åt rätt värde. Som ett exempel kan vi ta en samling objekt av personer från en klass Person, där varje person har ett unikt personnummer, telefonnummer och adress. Man kan med fördel lägga in personerna i en hashtabell och använda de unika personnumrerna som nycklar och objekten som värden. Genom att söka på ett visst personnummer i hashtabellen returneras då personen med just det personnumret. Det ger en komplexitet på O(1) för insättning och uttag.

Hashmapen är egentligen bara en vanlig array, som personobjekten ligger i. Personnummer blir väldigt stora tal, så att lägga in personen Annika på array[8701267869] är en väldigt dålig idé. Arrayen blir för stor och går ändå inte att skapa.

Det vi istället behöver göra är att ha en smart metod som kan transformera personnumret till ett realistiskt index i arrayen. Ett annat ord för det är hashning, och det är vad en hashfunktion gör. Nu kan arrayen få en storlek som är anpassad till antalet uppskattade objekt som den ska hålla.

Ett enkelt sätt att implementera en hashfunktion är att göra en restdivision på personnumret och arrayens storlek. Den rest vi får fram kan då bli det index objektet ska läggas in på. Det vanliga är att man implementerar hashfunktionen i klassen för objekten och inte i hashmapen. Annat vore dumt, en hashmap ska vara generisk.

Men det finns problem med att hashfunktionen i det här fallet är så simpel att `index = personnr % arrayLength` . Många av personnumrerna kommer tilldelas samma index och på så sätt kollidera med varandra. Det går att få en bättre spridning genom att göra uträkningen mer komplicerad. Ett vanligt sätt är att multiplicera in primtal i nyckeln(personnumret i vårt fall). Man kan göra betydligt mer komplexa varianter för att minimera antalet kollisioner, men det man vinner i säkerhet förlorar man oftast i hastighet. Så komplexiteten har ett pris och man får fråga sig om det är värt det eller inte.

Kollisionshantering

Med en mer komplex hashfunktion har spridningen blivit betydligt bättre, men det verkar vara svårt att undvika kollisioner helt. Men det finns ett antal sätt att hantera dem.

Linear probing

Ett sätt är att inkrementera index tills en ledig plats hittats. Om fyra nycklar mappar över samma index x blir placeringen av dessa $x, x+1, x+2$ och $x+4$. Det första av de fyra som sätts in sätts in på x, nr två sätts in på indexet efter x o.s.v. Söker man nyckeln som ligger på $x+4$ så börjar man kolla på x, följt av $x+1, x+2$ och $x+3$. Nyckel fyra finns på $x+3$ och kan returneras. Skulle nyckeln vi söker inte finnas på $x0 - x4$, avbryts sökningen då $x+5$ visar sig vara tom. Sökningen blir linjär och därav benåmningen linear probing.

Börjar hashtabellen/listan bli fylld innebär det långa kluster som behöver genomsökas, vilket tar tid. Av den anledningen skapar man dubbelt så många platser jämfört med antal nycklar. Då får man en större spridning av elementen och kan spendera kortare tid på sökning.

Den här metoden fungerar bra till en viss gräns. I värsta fall behöver man söka sig igenom alla nycklar i klustret innan man får hittar det man vill. Samtidigt är det många index i arrayen som gapar tomma.

Quadratic probing

För att komma undan problemet med långa kluster och öka spridningen kan man undvika ansamlingarna genom att ta längre skutt för varje ny nyckel på samma index. Istället för $x+0$, $x+1$, $x+2$, $x+3$ lägger man till tvåpotenserna av ökningen av x; $x+0^2$, $x+1^2$, $x+2^2$, $x+3^2$ osv.

Avståndet mellan elementen ökar (mer än)kvadratiskt för

varje insättning, därav benämningen quadratic probing. Alla nycklar som är bundna till ett visst index x kommer finnas inom samma intervall i förhållande till x.

Double hashing

Ett tredje sätt att angripa problemet med kollisioner är att, som i föregående exempel låta index x öka med ett visst antal steg, men istället för att det ska bli samma antal steg mellan nycklarna i för alla index låter man istället antalet steg bestämmas av värdet på x. Man låter x genomgå en ny hashning, och låter resultatet bli storleken på intervallen mellan nycklarna under x. Metoden kallas double hashing.

Länkad lista

Men man kanske vill komma bort från den dubbla hashningen eller den allt mer komplexa strukturen i en växande samling. Då är en idé att låta varje index i arrayen vara en länkad lista. Genomsökningen kommer i värsta fall ta lika lång tid som i det första exemplet, med linear probing, men man slipper göra en dubbelt så stor lista som antalet nycklar. Att låta nycklarna bestå av länkade listor kallas separate chaining. En stor nackdel med double hashing, quadratic- och linear probing är att de index som drabbats av kollisioner faktiskt snor åt sig platserna för andra öskyldiga"index. En genomsökning krävs för att hitta rätt nyckel från indexet med flera nycklar, men ska en ny nyckel sättas in på ett index som ockuperats av ett kluster drabbas ju även den nyckeln av klustrets försinkning. Det kommer man

ifrån om man använder en länkad lista, ansamlingarna under ett index drabbar inga andra index.

Källor

`https://en.wikipedia.org/wiki/Hash_table`

Data Structures and Algorithm Analysis in Java av Mark Allen Weiss