Hashstrategier

H E N R I K F L I N C K
o c h O L I V E R K R Ü G E R

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Hashstrategier

H E N R I K F L I N C K
o c h O L I V E R K R Ü G E R

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Mikael Goldmann
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
flinck_henrik_OCH_kruger_oliver_K10004.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Referat

Denna rapport innehåller en presentation av flertalet hashstrategier, dessa är Knuth-hashning, universell hashning, statisk perfekt hashning, linjär hashning, dynamisk perfekt hashning och gök-hashning. De tre första strategierna är statiska hashstrategier och de tre senare är dynamiska.

Ett experiment på de olika hashstrategierna presenteras i senare delen av rapporten. Det jämför hashstrategierna med avseende på antalet läsningar och skrivningar på hashtabellen.

Slutsatsen av experimentet är att statisk perfekt hashning är den bästa statiska strategin medan linjär hashning är den bästa dynamsiska.

Abstract

Hashing schemes

This paper contains a presentation of several hashing schemes. These are: Knuth-hashing, universal hashing, static perfect hashing, linear hashing, dynamic perfect hashing and cuckoo-hashing, where the first three are static hashing schemes and the latter three dynamic hashing schemes.

An experiment is presented in the later parts of this paper. It compares the hashing schemes with repect to file accesses on the hashtable.

The conclusion of the experiment is that static perfect hashing is the best static scheme and linear hashing the best dynamic scheme.

Innehåll

Kapitel 1

Introduktion

Denna rapport innehåller en presentation och praktisk jämförelse av sex olika hashstrategier. Av dessa är tre statiska: Knuth-hashning, universell hashning och statisk perfekt hashning. De tre övriga är dynamiska: linjär hashning, dynamisk perfekt hashning och gök-hashning. Strategierna presenteras grundligt där det beskrivs hur de fungerar. Egenskaperna för strategierna redovisas, såsom vilken förväntad eller garanterad tidskomplexitet som operationerna på tabellen har.

I rapportens senare del presenteras utförande och resultat av ett experiment på de sex hashstrategierna som behandlas i rapporten. I experimentet jämförs strategiernas effektivitet vad det gäller läsningar och skrivningar på datastrukturen. Mätning av läsningar och skrivningar är relevant för tabeller som ligger på långsamma minnen vilket stora tabeller ofta gör.

Slutsatserna som dras från experimentets resultat är att statisk perfekt hashning är den bästa statiska hashstrategin medan linjär hashning är den bästa dynamiska.

1.1 Bakgrund

Arbetet görs i ett praktiskt datalogiskt sammanhang, det vill säga; vi studerar datastrukturer som redan visats ha goda teoretiska egenskaper för att se hur väl de presterar i praktiken.

Det har på senaste tiden presenterats ett antal hashstrategier vilka har mycket goda teoretiska egenskaper, däribland gök-hashning och dynamisk perfekt hashning. Området är välstuderat inom teoretisk datalogi och har även praktiska användningsområden.

1.2 Litteratursammanfattning

Introduction to Algorithms av *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.*

Generellt om hashning. Beskriver universella hashfunktioner och därutöver delar om statisk perfekt hashning och kringstrategier. Endast kapitlet om

hashning har använts.

Dynamic Perfect Hashing: Upper and Lower Bounds av *Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, Robert E. Tarjan.*

Originalartikel där dynamisk perfekt hashning presenteras. Det är en av de hashstrategier som vi ska testa. Innehåller mest teori och bevis för att den presenterade strategin är perfekt.

Storing a Sparse Table with 0(1) Worst Case Access Time av *Michael L. Fredman, János Komlós, Endre Szemerédi.*

Artikel om statisk perfekt hashning som innehåller en grundligare beskrivning och bevisföring än motsvarande sektion i Introduction to Algorithms.

The Art of Computer Programming av *Donald E. Knuth.*

Allmänt om hashning och specifikt om Knuth-hashning. Innehåller i princip grunden för hashning i allmänhet och används ofta som standardreferens för klassiska hashstrategier. Bara kapitlet om hashning har använts.

Linear Hashing: A New Tool For File And Table Addressing av *Witold Litwin.*

Presentation av den linjära hashning som uppfanns av Litwin. Medtagen i testet eftersom den används i databassystem som Postgres.

Cuckoo Hashing av *Rasmus Pagh, Flemming Friche Rodler.*

Originalartikel av Pagh et al. som presenterar gök-hashning. Presenterar hashstrategin som uppfunnits och jämför den med en uppsättning andra hashfunktioner i tester.

Kapitel 2

Hashning

Hashning är en mycket vanlig metod som används för flera olika saker inom datalogi. Denna rapport innehåller en studie i hashning som används för datastrukturen hashtabell. Hashtabell är en datastruktur som har stöd för insättning, borttagning och uppslagning av element. De används eftersom hashtabeller har egenskapen att det tar väldigt lite tid(få operationer) att göra uppslagning; under många förutsättningar har uppslagningsoperationen förväntad konstant tidskomplexitet[4].

Användningsområdena för hashtabeller är otaliga eftersom det på den mest grundläggande nivån handlar om att hämta sparad data från någonstans i minnet så snabbt som möjligt. Detta kan givetvis vara användbart för många olika ändamål som till exempel databaser [10]. Det finns även många programmeringsspråk som har stöd för en typ av hashning i sina standardbibliotek, såsom Java[8] och Python[11].

Grundidén bakom hashtabeller är att man har någon hashfunktion $h(x)$ som ger värden i en mängd $\{0, 1, ..., m-1\}$. Värdet x är en nyckel och hör ihop med det element man vill sätta in i tabellen, det kan vara elementet själv, en del av hela elementet eller en tolkning av till exempel en sträng som ett heltal. Man använder sedan funktionen $h(x)$ för att finna på vilket index i en tabell T, vilken har plats för m element (vi indexerar dem $T[0], T[1], ..., T[m-1]$), som vi ska sätta in på. De olika index som man kan sätta in på kallas ibland hinkar[1]. Sökning och borttagning sker på samma sätt, man använder $h(x)$ för att finna vilken hink elementet ligger i och sedan utför man operationen på just den hinken.

2.1 Knuth-hashning

Vanligtvis vill vi att hashfunktionen ska vara komprimerande, det vill säga att rummet av alla nycklar U är en större mängd än $\{0, 1, ..., m-1\}$ eftersom vi annars kan finna en bijektion som hashfunktion vilket skulle resultera i teoretiskt konstant värstafallstid för uppslagning. Det är dock för det mesta så att nyckelrummet U är

[1] Eng. bucket.

en alldeles för stor mängd för att ha så många hinkar i minnet. U kan till exempel vara alla 32-bitars heltal.

En hashfunktion som Knuth föreslår[4], ofta är det första exempel på hashfunktion man ser, är att räkna

$$h(k) = k \pmod{m}$$

där nyckeln k är ett heltal. Funktionen uppfyller ovanstående krav för hashfunktioner. Om man så skulle lägga in ett element x som vi associerar med nyckeln k i tabellen T av storlek m skulle vi då göra det på följande vis:

1. Beräkna $h(k)$ som $k \pmod{m}$, vilket ger ett värde j i mängden $\{0, 1, ..., m-1\}$.

2. Sätt in x i hink $T[j]$ (hink med index j i tabell T).

Hur bra hashfunktionen ovan fungerar i praktiken visar sig bero mycket på valet av m. Om man till exempel väljer m som ett jämt tal får man egenskapen att alla jämna nycklar ger ett jämnt hashvärde $h(k)$ vilket är en negativ egenskap. Knuth anser att ett bra val av värde på m är att välja ett primtal[4], helst ska det inte vara alltför nära en hel tvåpotens 2^d [1].

Hashfunktioner som ger krockar

Det första problemet man ser är att om man har en komprimerande hashfunktion som ovan kommer det att finnas åtminstonde två tal $i, j \in U$ sådana att $h(i) = h(j)$. När det händer vid insättning kallas det för en "krock" eller "kollision". Om vi tar Knuth-hashning som exempel så krockar $i = 2$ och $j = m + 2$ eftersom $h(i) = 2 = h(j)$. Det finns en uppsjö av metoder för att hantera krockar varav ett par presenteras kort senare i sektion 2.2. Generellt kan man säga att krockar är någonting dåligt som försämrar effektiviteten hos uppslagning avsevärt, mer om det i sektion 2.3.

Som vi kommer att se har dock Knuth-hashning ett par icke så gynnsamma egenskaper, vilket är anledningen till att det finns andra hashstrategier. Det är därför sällan så att man använder Knuth-hashning i implementationer av hashtabeller där det verkligen är viktigt att det blir få krockar. Den finns ändå fortfarande i vissa bibliotek till exempel GNU Libc[12].

Andra Knuth-hashningar

Det vi benämner för Knuth-hashning kallas även för hashning genom division eftersom Knuth föreslår ett par andra hashfunktioner[4] som till exempel hashning genom multiplikation som går ut på att man för nyckeln k låter $A = xk$ för något flyttal $x \in (0, 1)$ och sedan tar man ut decimaldelen av A och baserar resultatet av $h(k)$ på den (multiplicerar med 2^m). Den algoritmen lider dock av många av de problem som hashning genom division lider av.

2.2 Kringstrategier

Det finns många aspekter som man kan studera och förbättra vad det gäller hashtabeller. Vi ska inte gå djupt in på dessa områden men det är användbart att ha dessa idéer klara för sig för att förstå vad som egentligen sker i implementationer.

Tolkning av olika datatyper

Eftersom de flesta hashfunktioner tar nycklar ur en mängd av heltal behöver man något sätt på vilket man kan tolka andra datatyper. Det är alltid möjligt att direkt tolka binära representationen av datan som ett heltal, men det ger ett väldigt stort tal som kan vara svårt att räkna med när man ska beräkna hashvärdet.

En vanlig datatyp att hasha på är strängar som mer effektivt kan tolkas genom att ta heltalsvärdet som representerar varje tecken $c_1, c_2, ..., c_n$ och beräkna heltalet $\sum_{i=1}^{n} c_i a^{i-1}$ där a är alfabetets längd i det avseende att alfabetet är mängden av möjliga tecken-värden. Det fungerar även väl att välja ett mindre tal än så, i Java är a konstant 31[9] vilket antagligen är valt så att man ska undvika jämna tvåpotenser.

Krockhantering

Som är nämnt kan flera nycklar hashas till samma hink vilket resulterar i en krock. För att ändå lagra värdet behöver man hantera krocken på något sätt. Den triviala lösningen är att man lagrar en lista i varje hink. När man sätter in i en hink lägger man till det som ska lagras i listan. När man sedan gör en sökning eller en borttagning av ett element måste man gå igenom hela listan för den hink med index som motsvarar hashvärdet på den nyckel man letar efter. Detta leder till att man i Knuth-hashning får en värstafallstidskomplexitet på $\Theta(n)$ när man söker, där n är antalet insatta nycklar i strukturen. Detta inträffar när många av nycklarna ger samma hashvärde ($\Theta(n)$ stycken av n). Denna metod kallas krocklösning med kedjning[2].

En annan lösning är öppen adressering där vi istället för att använda listor utforskar tabellen genom att successivt går igenom positioner enligt ett mönster tills vi finner en hink där ingen nyckel ligger. För att kunna behandla mönstret måste vi utöka mängden över vilken hashfunktionen är definierad till $U \times \{0, 1, ..., m-1\}$ där U är nyckelrummet och m är antalet hinkar i tabellen T. Alltså tar hashfunktionen ett par som indata. När man då gör en sökning efter nyckeln k räknar man ut hashvärdet $h(k, 0)$ och om det står något på den plats man är på i tabellen som inte är det man söker efter fortsätter man med $h(k, h(k, 0))$, och så vidare på samma sätt tills man finner den nyckel man söker eller en tom plats. Hur man beräknar $h(k, i)$ kan variera, till exempel används linjär och kvadratisk utforskning[3].

Fördelen denna krocklösning har jäntemot kedjning är att man slipper använda pekare och att man inte lagrar någon data utanför tabellen; all data skrivs

[2]Eng. collision resolution by chaining.
[3]Eng. probing.

på minne som man kan allokera i förväg. Nackdelen är däremot att det försvårar delete-operationen[1] eftersom om man bara skriver tillbaka en tom plats så kan det förstöra sökoperationens rekursion för andra element.

2.3 Bra strategier

Allmänt är en bra hashningsstrategi sådan att det tar lite tid att utföra operationerna på hashtabellen. Det som tar mycket tid är vanligtvis de läsningar respektive skrivningar som man behöver göra på datastrukturen. Om det är många krockar (många element i samma hink) försämrar det tidskomplexiteten avsevärt. Det är detta som gör att Knuth-hashning inte är så effektivt eftersom givet vissa mönster i indata kan man få att många (till och med alla) element hamnar i samma hink, sannolikheten att detta inträffar är inte försumbar.

Det man helst vill för hashfunktionen är således att $h(k)$ ger ett värde som utfaller med likformig fördelning över mängden $\{0, 1, ..., m-1\}$. Det är då en bra egenskap om $h(k)$ antar hashvärden med fördelning oberoende av fördelningen hos indata. Det är dock teoretiskt omöjligt att finna en funktion som uppfyller detta till fullo för icke slumpmässig indata [4]. Istället vill man finna en hashfunktion som uppfyller dessa egenskaper approximativt. Om vi tittar på Knuth-hashning som ett exempel ser vi att den inte uppfyller detta ens approximativt eftersom $h(k)$ ger värden som beror väldigt mycket på värdena på k.

Vad som är kritiskt för prestanda

Hur effektivt man kan beräkna själva hashvärdet $h(k)$ är för det mesta inte tidskritiskt eftersom för det första så behöver man varken läsa eller skriva i datastrukturen och för det andra är det ofta så att just den beräkningen är förhållandevis lätt. I Knuth-hashning så består till exempel bara hashningen av en modulo-operation; de flesta andra hashfunktioner är mer komplicerade än så men ändå inte kritiskt. Inte heller lagringsutrymmet är särskilt kritiskt givet att det håller sig inom rimliga gränser.

Den operation på strukturen som man vill ska vara snabbast (om man måste välja) är sökning i de flesta användningsområden. Det är oftast så att den operationen är den vanligaste, särskilt eftersom de flesta implementationer använder sig av en sökning som inleder både borttagning och insättning för att hitta elementet som ska tas bort respektive hitta platsen där elementet ska sättas in. Oavsett vilken strategi man använder för krocklösning tar sökningen i värsta fall $\Theta(n)$ tid om man använder sig av Knuth-hashning.

I det experiment som presenteras i sektion 5 jämförs strategierna med avseende på antalet läsningar/skrivningar i datastrukturen där vad som anses som läsning respektive skrivning också förtydligas i samma sektion. Anledningen till valet är att det är ofta avgörande för tidskomplexiteten och att om det inte är kritiskt kan man ofta ersätta hashtabellen med ett träd utan förlorad prestanda, fördelen hos hashtabeller är ju att man slipper göra många läsningar då man följer pekare. Mätning

av hur mycket minne som används görs också för att se att minnesutvecklingen inte är orimlig när datamängden skalas upp.

Kapitel 3

Statisk hashning

Statisk hashning innebär att man aldrig ändrar nyckelrummet U, det vill säga att man vet mängden U innan man skapar datastrukturen och att alla nycklar som man därefter använder under strukturens levnadstid är element i U. Till exempel så kan man begränsa U till att vara alla 32-bitars heltal om man vet innan man skapar tabellen att alla nycklar kommer att vara sådana tal. Motsatsen till denna typ av hashning kallas dynamisk hashning och presenteras i sektion 4 där det även står mer om vad som skiljer de två typerna åt.

3.1 Universell hashning

Ett problem med till exempel Knuth-hashning är att man alltid kan välja indata så att alla nycklar hashar till samma värde och på det sättet orsakar dålig prestanda ($\Theta(n)$ förväntad tidskomplexitet vid uppslagning). För att övervinna detta kan man utnyttja universell hashning. Det som gör Knuth-hashning och alla liknande hashningsmetoder dåliga i avseendet är att man fixerar en speciell funktion som man beräknar hashvärden med. I universell hashning väljer man istället ut, slumpmässigt, en hashfunktion ur ett funktionsrum som man sedan använder för att hasha. Om funktionsrummet är ett universellt funktionsrum, vilket vi definierar nedan, får vi vad som kallas för en universell hashfunktion.

En följd av att man slumpar fram en hashfunktion är att hashningen kan bete sig olika trots att man använder samma indata. Det vill säga om man skapar två tabeller och sätter in samma nycklar i vardera tabell kan tabellerna se olika ut.

Målet med den universella hashningen är att sannolikheten för att man väljer en hashfunktion som beter sig på oönskat sätt för någon specifik mängd av nycklar är liten. Om det till exempel som ovan beskrivet finns en hashfunktion som för viss indata hashar samtliga värden till samma hink så skulle den funktionen bara ha en liten sannolikhet att väljas ur funktionsrummet.

Definition 1. (Universellt funktionsrum) Låt $\mathcal{H}$ vara ett ändligt funktionsrum av

funktioner som avbildar nyckelrummet U på $\{0, 1, ..., m-1\}$.

$$\mathcal{H} = \{h | h : U \to \{0, 1, ..., m-1\}\}$$

$\mathcal{H}$ sägs vara universellt om

$$\forall k, l \in U : \mathcal{J} = \{h | h \in \mathcal{H} \wedge h(k) = h(l)\} \Rightarrow |\mathcal{J}| \leq \frac{|\mathcal{H}|}{m}$$

Med andra ord är för alla nycklar k, l alltså mängden av funktioner som gör att de hashas till samma hink liten, kravet på liten är i det här fallet att den är mindre än antalet hashfunktioner i funktionsrummet $|\mathcal{H}|$ delat på antalet hinkar i tabellen (eller värdemängden för samtliga funktioner i funktionsrummet).

Vi har ännu inte förklarat hur man finner ett universellt funktionsrum i praktiken eller att det ens finns ett funktionsrum som är universellt. Vi väntar med det till senare och betraktar vilka egenskaper som hashtabeller får om man använder en universell hashfunktion.

Det visar sig att om man väljer en universell hashfunktion (en slumpmässigt vald funktion ur ett universellt funktionsrum) att det tar $\Theta(n)$ förväntad tid att utföra en sekvens av n operationer, varav $O(m)$ är insättningsoperationer, på hashtabellen T av storlek m [1]. Övriga operationer är de som vi tidigare definierade; borttagning och sökning.

Hur man hittar ett universellt funktionsrum

En metod för att hitta funktioner till ett universellt funktionsrum $\mathcal{H}$ är:

1. Välj ett primtal p sådant att $\forall k \in U : 0 \leq k \leq p-1$, det vill säga ett primtal som är större än alla nycklarna i nyckelrummet.

2. Vi konstaterar att $|U| > m$ eftersom vi vill ha en komprimerande funktion och detta leder till att $p > m$, där m är storleken på värdemängden för hashfunktionerna $\{0, 1, ..., m-1\}$.

3. Vi definierar funktionen $h_{ab} : U \to \{0, 1, ..., m-1\}$ där $a \in \mathbb{Z}_p^*$ och $b \in \mathbb{Z}_p$ som

$$h_{ab}(k) = (ak + b \pmod p) \pmod m$$

Vi låter nu funktionsrummet $\mathcal{H}$ vara mängden av alla sådana funktioner, det vill säga

$$\mathcal{H}_{pm} = \{h_{ab} | a \in \mathbb{Z}_p^* \wedge b \in \mathbb{Z}_p\}$$

Det går att bevisa att ovanstående funktionsrum uppfyller alla krav för att vara ett universellt funktionsrum [1]. När man använder funktionsrummet slumpar man bland $|\mathcal{H}| = p(p-1)$ funktioner: $|\mathbb{Z}_p| = p$ och $|\mathbb{Z}_p^*| = p - 1$.

Vi har nu funnit en hashstrategi som ger bra amorterad förväntad tidskomplexitet, givet att man vet statiska nyckelrummet U i förväg. Man behöver veta U för att kunna välja ett primtal p i metoden för att finna hashfunktioner.

3.2 Statisk perfekt hashning

Statisk perfekt hashning baserar sig mycket på universell hashning och använder sig av universella hashfunktioner. Istället för att bara använda en slumpad hashfunktion har vi hashfunktioner i två nivåer. Det som gör att hashfunktionen kan kallas perfekt är att man kan garantera att inga krockar kommer att uppträda vilket i sin tur gör att man får en värstafallstidskomplexitet på $O(1)$ för uppslagning i hashtabellen. Denna metod förutsätter dock att nyckelrummet U är statiskt, vi ska senare titta på strategier som är perfekta för dynamiska nyckelrum.

Första nivån

Den första nivån består av en vanlig hashtabell T med m hinkar. Vi använder samma metod här som för universell hashning. Vi väljer en hashfunktion som ger värden i mängden $\{0, 1, ..., m-1\}$ precis som för universell hashning, det vill säga slumpmässigt ur ett universellt funktionsrum. Därefter testar vi om antalet krockar för den hashfunktionen för alla nycklar är färre än $m+1$, om så godkänns den. Om den inte godkänns så slumpas en ny och testet upprepas. Det tar förväntad $O(1)$ tid tills vi hittar en godkänd hashfunktion eftersom förväntat antal krockar för en slumpmässigt vald universell hashfunktion är $\frac{m-1}{2}$ [1] vilket är mindre än m.

Vi måste välja m så att den är i storleksordningen av $|U|$, speciellt kan vi välja att $m = |U|$. Detta leder till att vi kommer att behöva en liten nyckelmängd för att inte få en för stor tabell.

Istället för att lagra nycklarna i en länkad lista för varje hink, som när vi använder kedjning för krocklösning, använder vi en egen hashtabell i varje hink.

Andra nivån

I hink j har vi en hashtabell T_j till vilken vi associerar hashfunktionen h_j. Vi måste välja hashfunktionen på ett sådant sätt att den garanterar att inga krockar förekommer i tabellen T_j.

Låt m_j vara storleken på hashtabellen T_j och n_j vara antalet nycklar som hashar till värdet j:

$$n_j = |\{k | k \in U \wedge h(k) = j\}|$$

Vi kommer behöva bestämma storleken för varje hashtabell T_j till att vara kvadratiskt stort i förhållande till antalet element som hashas till hink j, det vill säga $m_j = n_j^2$. Andranivåtabellernas storlek är avgörande för att man ska kunna hitta en bra hashfunktion snabbt.

Man kan göra detta utan att få värre värstafallslagringsutrymme för hela hashtabellen än $O(m)$ [1] vilket gör att metoden är praktisk även i minneskritiska sammanhang, i alla fall i teorin. Eftersom $m = |U|$ ser vi att storleken blir proportionell mot antalet element i nyckelrummet.

Val av hashfunktioner

Med samma notation som i föregående sektion väljer vi hashfunktion i förstanivån slumpmässigt ur funktionsmängden $\mathcal{H}_{pm}$.

På den andra nivån väljer vi istället en hashfunktion ur funktionsrummet $\mathcal{H}_{pm_j}$ för hashfunktionen för tabellen T_j. Vi väljer alltså funktioner som ser ut på formen

$$h_{ab} = (ak + b \pmod p) \pmod m$$

Det visar sig emellertid att om vi skulle välja en slumpmässig hashfunktion ur $\mathcal{H}_{pm_j}$ så är sannolikheten att den ger en eller flera krockar i tabellen T_j, då man sätter in alla nycklar som hashas till hink j, är mindre än $1/2$ [1]. Vi kan därför välja h_j slumpmässigt och testa hurvida den ger några krockar för de nycklar $k_1, k_2, ..., k_{n_j}$ som hashas till j i förstanivån: $h(k_1) = h(k_2) = ... = h(k_{n_j}) = j$. Inom ett litet antal försök finner vi en hashfunktion helt utan krockar, vi kan alltså alltid bara välja en hashfunktion slumpmässigt tills vi finner en krocklös h_j [3].

Praktiska detaljer

Man kan tycka att det då inte verkar finnas någon användning för andra hashfunktioner då denna ger ett asymptotiskt optimalt resultat tidskomplexitetsmässigt. Det är dock ett problem att man ändå behöver utföra mycket arbete för att väl skapa tabellen, eftersom man måste finna en hashfunktion för varje hink $j \in \{0, 1, ..., m-1\}$.

Det andra praktiska problemet är att det inte alltid är möjligt, eller i alla fall sällan lätt, att ställa upp en komplett modell över vilka nycklar som man använder vilket kan leda till att man använder mycket mer utrymme än vad man egentligen behöver och att man kanske inte alls kan bestämma statiska nyckelrummet U i förväg. Om man inte kan bestämma nyckelrummet håller inte egenskapen som garanterar att inga krockar sker. Om man väl lyckas hitta en modell kan $|U|$ vara orimligt stort för att man ska kunna spara en statisk tabell av den storleksordningen.

Med anledning av dessa problem ska vi studera dynamisk hashning i sektion 4.

Kapitel 4

Dynamisk hashning

Dynamisk hashning syftar på att nyckelrummet U är en dynamisk mängd. Det vill säga man kan uppdatera mängden allteftersom man lägger till nycklar. Vi kan ofta välja att tolka mängden som en oändlig mängd istället för en dynamisk mängd, det vill säga att det vi gör är att skapa en dynamisk bijektion från nyckelmängden till $\mathbb{N}$ genom att bara associera nästa nyckel med heltalet associerat med senaste nyckeln plus 1.

Strategierna är också dynamiska i det avseende att man utökar utrymmet man har allokerat för att lagra datastrukturen dynamiskt allteftersom man lägger till nya nycklar i strukturen.

4.1 Linjär hashning (Litwin)

Vid linjär hashning så vill vi utöka storleken på den tabell som innehåller de hinkar till vilka vi hashar på med en hink varje gång vi ser att det blir en krock. Därför är en dynamisk array en lämplig datastruktur att lagra hinkarna i för denna hashningsstrategi.

När linjär hashning uppfanns av Witold Litwin 1980 så var det den bästa kända hashningsmetoden [5]. Det är en strategi som används i praktiken; den används till exempel i databassystemet Postgres [10].

Uppdatering av hashfunktionen

Om man ändrar tabellens storlek måste man också ändra hashfunktionen för att den ska kunna ge hashvärden som motsvarar den nya hinken. Vi vill ha den egenskapen att den nya hashfunktionen inte är sådan att vi behöver flytta om många element. I linjär hashning använder hashtabellen alltid två hashfunktioner samtidigt, vi uppdaterar successivt vilken hashfunktion som används för vilken hink.

Om vi börjar med en hashfunktion

$$h_0 : U \to \{0, 1, ..., m-1\}$$

så kan vi definiera en *delningsfunktion*[1].

Definition 2. (Delningsfunktion) Funktionen h_i sägs vara en delningsfunktion om den uppfyller

1. $h_i : U \to \{0, 1, ..., 2^i m - 1\}$
2. $\forall k$ antingen
$$h_i(k) = h_{i-1}(k)$$
eller
$$h_i(k) = h_{i-1}(k) + 2^{i-1}$$

Vi kan som ett exempel säga att om vi tar h_0 till att vara funktionen från Knuth-hashning, $h_0(k) = k \pmod m$ så är $h_i(k)$ en delningsfunktion om

$$h_i(k) = k \pmod{2^i m}$$

eftersom punkt 1 i definitionen uppfylls på grund av modulo-operationens värdemängd och punkt 2 uppfylls eftersom för heltal är

$$k \pmod{2m} = \begin{cases} k \pmod m & \text{om } 2 \not| m \\ k \pmod m + k & \text{för övrigt} \end{cases}$$

Delning

Delningen sker linjärt. Vi inleder med att välja ett index $l = 0$ som ska fungera som en pekare på hinkar i hashtabellen T av storleken m. l inleds att peka på den första hinken i T. Vi definierar en hashfunktion h_0 som vi ska använda sådan att det är lätt att finna delningsfunktioner. Vi kan till exempel använda Knuth:s hashningsfunktion.

Vi inleder hashningen med att bara använda h_0 på vanligt vis genom att sätta in nyckel k i hink $h_0(k)$. Detta gör vi tills första krocken inträffar. När det sker en krock använder vi först vanlig krocklösning genom kedjning. Därefter lägger vi till en ny hink i slutet på T och kör delningsfunktionen h_1 på alla element som ligger i hink l (vid första krocken är $l = 0$). Om det inte ligger några element i hink l gör vi inget annat än att skapa den nya hinken. Om det däremot ligger element i hinken kommer de fördelas på hink l och hink $m + l$.

Eftersom vi gör delningar och utökar tabellen varje gång vi sätter in ett element som krockar med något annat, får vi att det görs fler delningar av hinkar ju fler krockar vi finner, vilket är en bra egenskap för effektiviteten hos strategin.

Efter att vi har delat en hink inkrementerar vi l och vid nästa gång det sker en krock kör vi liksom tidigare hashfunktionen på de element som ligger i den hink som l anger. Vi utför alltså delning av hinkar linjärt då den första hink som delas är hink 1, den andra hink 2 och så vidare.

[1] Eng. split function.

När vi når $l = m$ vet vi att alla element i samtliga hinkar är i den hink de hashas till med funktionen h_1 eftersom vi har använt delningsfunktionen på alla hinkar $0, 1, ..., m-1$ och allt vi lagt i hinkarna $m, m+1, ..., 2m-1$ har kommit dit för att de hashats med h_1. Detta innebär att vi är i exakt samma situation som vi började i, vi kan ersätta h_0 med h_1, h_1 med h_2 och m med $2m$ och återupprepa proceduren ovan genom att bara sätta $l = 0$. Så nästa gång vi finner en krock vid insättning delar vi den första hinken igen.

Uppslagning

Eftersom vi nästan hela tiden har några nycklar som är hashade med olika hashfunktioner i tabellen måste vi ändra lite i hur vi utför uppslagning. När vi gör en uppslagning efter nyckeln k beräknar vi först $h_0(k)$ och eftersom l anger nästa hink att göra delning på vet vi att om $h_0(k) < l$ har vi utfört delning med h_1 på hink $h_0(k)$. Därför väljer vi den hink s som vi letar i som

$$s = \begin{cases} h_0(k) & \text{om } h_0(k) < l \\ h_1(k) & \text{för övrigt} \end{cases}$$

4.2 Dynamisk perfekt hashning

Dynamisk perfekt hashning grundas i statisk perfekt hashning som presenteras i sektion 3.2. Vi väljer därför att använda notationen från den sektionen utan ytterligare förklaring. Vi övervinner problemet med ett dynamiskt nyckelrum med två idéer:

1. Välj hashfunktion i andranivån dynamiskt.
2. Dubblera/halvera tabellstorleken vid behov.

Uppdatering av andranivåtabell vid krock

Antag att vi vid en viss tidpunkt t har n nycklar som vi vill sätta in i en hashtabell. Vi skapar då en statisk perfekt hashtabell för $2n$ nycklar och sätter in nycklarna i den tabellen. Eftersom vi inte vet vilka nycklar som kommer att sättas in efter tidpunkt t , då nyckelrummet U är dynamiskt, kan det bli krock vid insättning. Om en nyckel k hashas så att det blir en krock så bygger vi om hela tabellen $T_{h(k)}$ så att det inte blir några krockar. Vi måste alltså hitta en ny hashfunktion ur det lämpliga funktionsrummet som inte ger någon krock. Som tur är inträffar en krock sällan.

Dubblering och halvering

När antalet element når den gräns som vi skapade den statiska hashtabellen för ($2n$ ovan) bygger vi om hela hashtabellen så att den har plats för dubbelt så många

element som antalet element i tabellen för tillfället. För att fortsätta på exemplet ovan gör vi en helt ny statisk perfekt hashtabell för $4n$ nycklar när antalet element i hashtabellen når $2n$.

Vi växer och krymper de inre tabellerna också. Vi vill garantera att vi alltid storleken på dessa är minst $4n^2$ där n är antalet element i den inre tabellen. Det vill säga kvadratiskt stor för $2n$ element. Vi kan använda samma amorterade teknik som vanligt men bara öka tabellens storlek med en faktor 4 varje gång det behövs.

Vi kan också halvera genom att bygga om hela tabellen om antalet insatta nycklar halveras från det att tabellen återskapades senast.

En följd av att vi alltid dubblerar och halverar antalet nycklar som vi bygger tabellen för är att vi får en amorterad tidskomplexitet för insättningsoperationen. Insättning har $O(1)$ amorterad tidskomplexitet[2].

Uppslagning och borttagning

Uppslagning är helt analogt med uppslagning i en vanlig statisk perfekt hashtabell och vi gör inga ändringar när vi söker i tabellen så värstafallstidskomplexiteten blir konstant [2].

Borttagning sker genom att man markerar de element som man vill ta bort för att sedan lämna själva borttagningen till när man omstrukturerar antingen hela tabellen eller andranivåtabellen, då man ändå måste flytta elementet kan man ta bort de element som är markerade för borttagning. Därför har borttagningsoperationen $O(1)$ amorterad tidskomplexitet [2].

Minnesåtgång

Eftersom dynamisk perfekt hashning bara använder sig av vanliga statiska perfekta hashtabeller som har $O(n)$ minneskomplexitet, har även dynamisk perfekt hashning det. Den enda skillnaden är att dynamisk perfekt hashning lagrar statiska tabeller med plats för dubbelt så många nycklar som antalet insatta nycklar, i värsta fall, vilket gör att storleken blir $O(2n) = O(n)$.

4.3 Gök-hashning

En annan perfekt dynamisk hashstrategi är gök-hashning[2]. Den använder ett helt annat tillvägagångssätt än dynamisk perfekt hashning. Den har ändå också konstant värstafallstid för uppslagning.

Tabellen består egentligen av två stycken universella hashtabeller med olika hashfunktioner h_1 och h_2, som har värdemängden $0, 1, ..., m-1$ där m är tabellernas storlek: $|T_1| = |T_2| = m$.

[2]Eng. Cuckoo-hashning.

Val av hashfunktion

Vi kan inte välja vilken universell hashfunktion som helst, den hashfunktion vi väljer måste ha universella egenskaper för vilken mängd, av storlek m^2, med nycklar som helst. En familj som ger funktioner med sådana egenskaper, med stor sannolikhet, finns. Den tar konstant tid att beräkna och inte tar mer än linjärt minne att lagra[7].

Insättning

För att sätta in ett nyckel k i tabellen beräknar vi $h_1(k)$ och sätter in k i den hinken i tabell T_1. Om det redan fanns en nyckel k' i den hinken då plockar vi ut den och sätter in den i tabell T_2 i hink $h_2(k')$. Vi fortsätter på samma sätt tills vi inte ersätter ett element i någon tabell. Om det blir en krock i T_i tar vi den nyckel som stod i tabellen och sätter in den i tabellen T_j, där $i \neq j$.

Om det inte går att sätta in nyckeln för att rekursionen bildar en slinga som fortsätter oändligt måste man avsluta insättningen och bygga om hela tabellen. Vi väljer då nya hashfunktioner till tabellerna och försöker sätta in alla värden igen. Det händer som tur är inte så ofta, eftersom vi använder oss av universella hashfunktioner. Om byte av hashfunktioner inte sker på m^2 insättningar tvingar vi ett byte trots att ingen slinga är funnen [6]. Anledningen till detta är att funktioner ur det universella funktionsrum vi använder bara har de egenskaper som krävs för mängder av storlekar upp till m^2.

Vi får att den förväntade tidskomplexiteten för operationen insättning är $O(1)$ [6], detta eftersom det kostsamma bytet av hashfunktioner sker så sällan.

Uppslagning och borttagning

Ovanstående definition av insättningsoperationen garanterar att en nyckel k hashas antingen till hink $h_1(k)$ i tabell T_1 eller hink $h_2(k)$ i tabell T_2. Man behöver därför bara kolla i dessa två hinkar när man söker. Således har uppslagning konstant värstafallstidskomplexitet [6].

Vid borttagning behöver man bara ta bort elementet från där det står i tabellen eftersom inga kedjor används. Det tar alltså också $O(1)$ tid i värsta fall [6].

Dubblering och halvering

Precis som för dynamisk perfekt hashning förstorar man tabellerna så att m är minst $2n$ där n är antalet insatta element i tabellen genom att dubblera eller halvera tabellen när m blir $2n$ eller $n/2$ respektive (genom att man tar ut och sätter in element i tabellen). Detta kräver en hel ombyggnad av tabellerna, men eftersom det sker sällan kan man visa att den amorterade tidskomplexiteten för insättning fortfarande konstant [6].

Minnesåtgång

Allt som behövs är två tabeller i vilka man lagrar alla element. Därför kan man lagra dessa på $O(n)$ minnesutrymme, vilket motsvarar minnesutrymmet asymptotiskt för dynamisk perfekt hashning. Det visar sig dock att den konstanta termen för gök-hashning är mycket mindre än för dynamisk perfekt hashning och således i praktiken är mer minneseffektivt [6]. I dynamisk perfekt hashning har man en tabell för varje hink till skillnad från gök-hashning där vi har två, T_1 och T_2. Tanken är att de två tabellerna ska ta mindre lagringsutrymme än summan av alla de små tabellerna i den dynamiska perfekta hashtabellen. Approximativt är storleken $2n$ på tabellen i gök-hashning mot cirka $35n$ i dynamisk perfekt hashing [6].

Kapitel 5

Experiment

Följande experimentella resultat avser att ge en grund för jämförelse av de presenterade hashstrategierna. Det ger information om hur bra strategierna presterar i praktiken.

5.1 Upplägg

För experimentet har nya implementationer av strategierna skrivits. De är tillgängliga för nedladdning på Internet [13]. De må inte vara optimala, men då vi inte mäter tidseffektiviteten är det irrelevant.

Mätningar

Det som mäts i experimentet är antalet läsningar och skrivningar. Anledningen till att vi mäter dessa är att om man har mycket stora tabeller så är dessa operationer flaskhals då tabellen antagligen lagras på ett långsamt minne. Resultaten tar alltså inte hänsyn till hur cache-effektiva de olika strategierna är. Trots att storleken på experimentdata inte motsvarar den storlek på tabeller som man skulle behöva lagra på sekundärminnet bör den ge information om hur antalet läsningar respektive skrivningar förändras jämtmed att tabellen växer.

Vad som räknas är antalet läsningar och skrivningar till datastrukturen och interna datastrukturer som eventuellt används. Vi räknar inte varje temporär variabel eftersom vi antar att dessa får plats i något primärminne. Vi antar även att man kan läsa hela datastrukturer som innehåller ett litet antal bytes med en läsning, men att det givetvis tar ytterligare läsningar att leta igenom tabeller och följa pekare. Även kostnaden för att allokera minne försummas, inte heller att nollställa nyallokerat minne, räknas.

Kostnaden för att initialt bygga tabeller försummas. För statisk hashning leder detta till att ingen tabellkonstruktion räknas medan för dynamisk hashning så ökar respektive minskar tabellstorleken och då den byggs om adderas läsnings- och skrivnings-kostnaden för ombyggnationen till antalet läsningar och skrivningar som

redan gjorts på strukturen. Modellen för hur de statiska hashtabellerna används är således att antalet operationer efter skapandet är såpass många att de överväger kostnaden för initieringen.

Eftersom implementationerna är nya och specifika för detta experiment är det svårt att göra dessa optimala vad det gäller tid och minnesåtgång. Om man då mäter det är det väldigt svårt att ge en objektiv bild av hur väl dessa fungerar i praktiken. Det blir värre om man använder befintliga implementationer eftersom det är svårt att kontrollera att dessa är optimala eller ens jämförbara. Detta är ytterligare en anledning till att antalet läsningar och skrivningar är det som mäts eftersom det är mycket lättare att kontrollera att alla implementationer räknar sådana under samma förutsättningar.

Implementationer

För att hantera krockar i de tabeller som behöver göra det, det vill säga Knuthhashning, universell hashning och linjär hashning, så använder vi kedjning som beskrivet i sektion 2.2. Vi använder en enkellänkad lista i varje hink i vilken vi sätter in alla element som hashas till hinken. Eftersom statisk perfekt hashning, dynamisk perfekt hashning samt gök-hashning inte får några krockar så behöver vi i dessa implementationer inte använda någon krockhantering. De perfekta hashstrategierna kan således bara få krockar om insättning utförs på en nyckel som redan är insatt i tabellen, det är en misslyckad insättning. En misslyckad borttagning är ett försök till borttagning av en nyckel som inte finns i tabellen. Begreppen lyckad och misslyckad operation är demsamma för icke perfekta hashstrategier.

Alla tabeller lagrar enbart nycklar som tillhör mängden av 32-bitars heltal. Eftersom vi utför våra tester på strängar använder vi följande summation av ASCII-värden $x_1, x_2, ..., x_n$ där x_i motsvarar tecken i i strängen:

$$\sum_{i=0}^{n} 31^i x_i$$

Ovanstående formel är precis samma som används i Java för att hasha strängar[9].

Implementationerna beskrivs nedan framförallt med avseende på hur vi väljer att räkna läsningar och skrivningar

Länkad lista

Eftersom vi använder en länkad lista i många av implementationerna så presenteras här hur den implementationen räknar läsningar och skrivningar.

- Vid sökning görs en läsning för varje pekare man följer, alltså är antalet läsningar mellan 1 och k, där k är antalet element i listan.
- Vid insättning kan vi antingen kontrollera så att vi inte sätter in dubbletter med en sökning eller så kan vi bara sätta in elementet först i listan. Utöver den eventuella sökningen utförs, förutsatt att elementet inte hittades med sökningen, alltid en läsning och en skrivning.

- Vid borttagning räknar vi precis som för sökning när vi letar efter elementet att ta bort. Borttagningen tar sedan en skrivning om vi hittar elementet som ska tas bort.

Knuthhashning

Implementationen är rättfram då tabellen bara består av en tabell som har fyllnadsfaktor[1] $\frac{1}{2}$. Detta innebär att antalet hinkar är som minst två gånger antalet insatta element. Eftersom tabellen är statisk tar vi reda på en lämplig storlek i förväg, när tabellen skapas.

Att läsa pekaren till den länkade listan i varje hink från tabellen räknar vi som en läsning. Detta görs en gång vid alla tre operationer, insättning, borttagning och sökning. Utöver detta räknas givetvis de läsningar och skrivningar som görs i de länkade listorna på motsvarande operationer enligt ovan.

Universell hashning

Universell hashning fungerar precis på samma sätt som Knuthhashning, enda skillnaden är hur man väljer hashfunktionen när tabellen skapas.

Statisk perfekt hashning

Att bygga hashtabellen är kostsamt vad det gäller läsningar och skrivningar men vi har valt att inte räkna det för någon av strategierna vilket gynnar denna tabell om man inte gör många uttagningar och insättningar.

- För alla operationer läser vi pekaren till de inre tabellerna i den yttre tabellen med en läsning på strukturen.
- För sökning så gör vi bara en läsning i den inre tabellen.
- Insättning och borttagning gör först en sökning och därefter en skrivning om sökningen misslyckades respektive lyckades.

Linjär hashning

Den linjära hashningen fungerar likadant som Knuthhashning och universell hashning förutom att det används en delningsfunktion som delar upp elementen i en hink på två hinkar vid krock. Delningsfunktionen behöver alltså också göra läsningar och skrivningar.

Funktionen som delar hinken läser alla element i den länkade listan med en läsning för varje följd pekare och utför sedan en insättning, utan dubblettkontroll, på någon av de två länkade listorna, vilket räknas som beskrivet ovan.

Implementationen använder en dynamisk array som strukturen för tabellen, vilket innebär att i verklgheten dubbleras eller halveras tabellens storlek vid behov men vi kan lika gärna se det som att den utökas med en hink i taget, den amorterade tidskomplexiteten blir konstant.

[1] Eng. load factor.

Dynamisk perfekt hashning

För dynamisk perfekt hashning skapar vi först en liten tabell vars byggnation inte kostar några läsningar eller skrivningar. För lyckade operationer, som inte orsakar ombyggnation av inre eller yttre tabeller, fungerar dynamisk perfekt hashning precis som statisk perfekt hashning.

När det blir en krock byggs den inre tabellen om. Detta görs genom att vi söker igenom alla positioner i tabellen och då gör en läsning för varje position och placerar alla element i en temporär länkad lista utan dubblettkontroll. Sedan går vi igenom hela listan och sätter in alla element i en ny tabell, då gör vi en läsning i länkade listan och sedan en insättningsoperation som vanligt. Vi summerar alltså antalet läsningar och skrivningar på temporära listan till tabellens totala antal.

Om antalet element når kapaciteten för tabellen behöver vi göra den yttre tabellen större. Vi gör då på samma sätt som när vi bygger om den inre listan, vi lägger alla element i en lista och gör upprepade insättningar av alla elementen tills vi inte får några krockar. Under denna ombyggnad kan således många inre ombyggnationer äga rum. Exakt samma sak gäller då vi minskar tabellens storlek.

Gök-hashning

För gök-hashning räknar vi under samma förutsättningar som de andra, men eftersom den inte är lik någon annan ser det lite annorlunda ut. Vi gör läsningar och alternativt skrivning vid alla operationer men vi behöver även bygga om hela tabellen emellanåt.

Eftersom Siegels hashfunktion[7] är komplicerad att implementera används i [13] en hashfunktion h som har ekvivalenta universella egenskaper men för generella nycklar tidskomplexiteten $O(n)$ där n är antalet bitar i nyckeln:

$$h(k) = \left(\sum_{l=0}^{n-1} a_i k^l \right) \pmod{p} \pmod{m}$$

där $a_0, a_1, ..., a_{n-1} \in \mathbb{Z}_p$ och m, p precis som i konstruktionen av en universellt funktionsrum i sektion 3.1. Eftersom den dock bara tar 32-bitars heltal som nycklar har den ändå kostant tidskomplexitet. Det görs varken skrivningar eller läsningar när man beräknar hashfunktionen oavsett om man använder Siegels hashfunktion eller h alltså påverkas ändå inte resultaten.

För gök-hashning behöver man ett speciellt element som representerar att inget står i en viss hink. Låt oss benämna det för `NIL`, i implementationen [13] så definierar vi `NIL` som `0xffffffff`.

- Vid sökning gör man först en läsning i den första tabellen. Om man inte finner den nyckel man söker efter gör man ytterligare en läsning i den andra tabellen.

- Vid borttagning gör man precis som vid sökning, men om man finner elementet så skriver man `NIL` över det. Om tabellens storlek ska minskas tillkommer kostnaden för att bygga om tabellen.
- Insättning gör först en sökning efter det man vill sätta in för att se om den kan sätta in. För varje gång man ersätter ett element som redan står i en tabell så gör man en läsning och sedan en skrivning, detsamma gäller även om man läser `NIL` och kan avsluta slingan. Dessutom tillkommer kostnaden för att bygga om tabellen om man når gränsen för antal varv i slingan.
- När vi bygger om tabellen så gör vi liksom för dynamisk perfekt hashning så att vi tar ut alla element ur tabellen och lägger det på en lista inklusive det element som ska sättas in i tabellen, vilket ger en läsning för varje hink. Vi behöver inte använda dubblettkontroll i listan eftersom vi redan vet att inga dubbletter kan finnas.

  Vi går igenom listan av alla element med en läsning för varje som vi kollar på och utför en insättningsoperation på alla dessa in i en förstorad eller förminskad tabell.

Källkoden [13] var för testerna kompilerad med GNU C Compiler version 4.4.3 på Arch Linux. Kärnan är av version 2.6.33 och följande parametrar användes vid kompilering: `-Wall -O2 -std=gnu99 -lgmp`. Koden länkades alltså med GMP. Versionen av GMP som användes var 5.0.1. Testerna utfördes på en Intel T7200-processor. Dessa detaljer borde inte spela någon roll för mätningarna av antalet läsningar och skrivningar men kan ha inverkan på resultatet vad det gäller minnesåtgång.

Mätning av minnesåtgången gjordes med GNU time version 1.7 genom att använda `gnutime -f ''%M'' <testprogram>`.

Testfall

Testfall genererades genom att välja ord från en text rensad från specialtecken och skriven i gemener. Varje ord som valdes ur texten gjordes om till ett 32-bitars heltal enligt metoden som är beskriven tidigare i rapporten. Varje sådant tal fick en operation som valdes med sannolikheterna

$$p(X) = \begin{cases} 1/5 & \text{Om } X \text{ är insättning eller borttagning} \\ 3/5 & \text{Om } X \text{ är sökning} \end{cases}$$

Genom att gå igenom texten ord för ord och slumpa operationer skapades testfall av olika många operationer. Det skapades fem testfall för varje storlek i mängden $\{10^4 + 4k * 10^4 | k \in [0, 12]\}$. Det vill säga fem testfall med 10 000 operationer, fem testfall med 50 000 operationer och så vidare ända upp till fem testfall med 490 000 operationer, hela tiden med en ökning av 40 000 operationer.

Det valdes att inte slumpa fram testfall helt utan grunda de i en text eftersom helt slumpmässig data inte liknar verkligheten och testningen saknar poäng då man kan uppnå perfekt uniform fördelning med en enkel hashfunktion.

För alla testfall användes samma text och de n första orden valdes för ett testfall av storlek n. Det vill säga att det är precis samma nycklar som används i testfall av samma storlek. Eftersom operationerna väljes slumpmässigt skiljer sig testfallen åt och den verkliga nyckelmängden (lyckade insättningar) är inte samma i alla testfall.

Antal op.	Insättningar		Borttagningar		Sökningar	
	Misslyckade	Lyckade	Misslyckade	Lyckade	Misslyckade	Lyckade
10 000	2.8%	17.0%	16.9%	3.0%	51.2%	9.0%
50 000	3.7%	16.2%	16.3%	3.7%	49.0%	11.0%
90 000	4.0%	16.1%	16.1%	4.0%	48.2%	11.7%
130 000	4.1%	16.0%	15.9%	4.0%	47.8%	12.2%
170 000	4.1%	15.8%	15.8%	4.1%	47.7%	12.4%
210 000	4.2%	15.8%	15.7%	4.2%	47.3%	12.7%
250 000	4.3%	15.7%	15.8%	4.3%	47.2%	12.8%
290 000	4.4%	15.7%	15.6%	4.3%	47.0%	13.0%
330 000	4.3%	15.7%	15.7%	4.3%	47.0%	13.0%
370 000	4.3%	15.7%	15.7%	4.3%	47.2%	12.9%
410 000	4.3%	15.8%	15.7%	4.3%	47.0%	13.0%
450 000	4.3%	15.7%	15.7%	4.3%	47.1%	12.9%
490 000	4.3%	15.7%	15.7%	4.3%	47.0%	12.9%

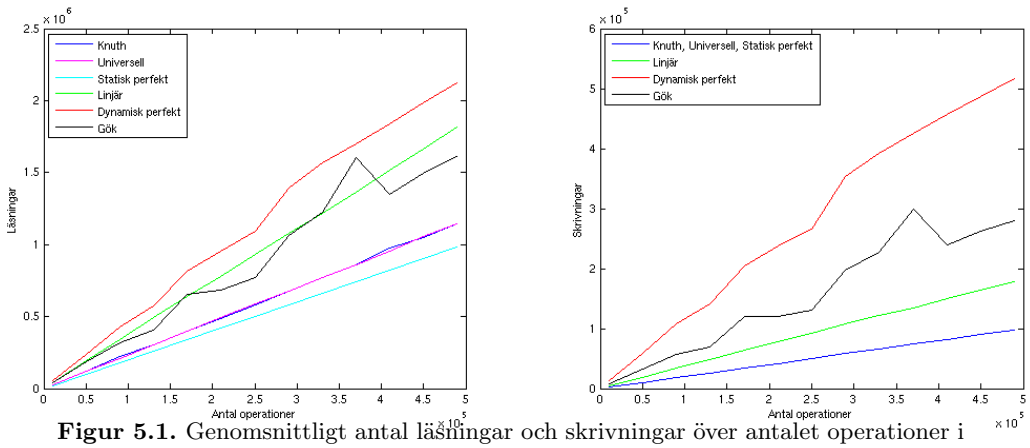
Tabell 5.1. Ungefärlig fördelning av operationerna i de olika testfallen. Andelen ges för medelvärden av flera testfall av samma storlek (i antal operationer).

I tabell 5.1 presenteras genomsnittliga siffror på hur testfallen ser ut. Hur stor andel av operationerna är av en viss typ där typ kan vara insättning, borttagning eller sökning varav varje antingen är misslyckad eller lyckad. Vi ser fördelningen av operationer är någorlunda konsekvent över de olika testfallsstorlekarna. Det betyder dock inte att alla testfall är lika eftersom ordningen och data varierar för operationerna.

5.2 Presentation av resultat

Man bör se på statisk hashning och dynamisk hashning som två olika tester, eftersom de inte har samma användningsområde. För de statiska tabellerna måste vi ha en fixerad nyckelmängd när vi skapar tabellen vilket gör att vi aldrig behöver bygga om tabellen eller ändra dess storlek. Det är dock vanligt att det inte är praktiskt möjligt att sätta upp en modell för data som ger en tillräckligt liten nyckelmängd, det är i sådana fall man behöver dynamiska hashtabeller.

I figur 5.1 ser vi hur för en statisk nyckelmängd de dynamiska hashtabellerna presterar sämre än alla andra både vad det gäller läsningar och skrivningar. Det är inte oväntat då det är kostsamt att behöva bygga om tabellen och således svårt för dessa att konkurrera med statiska tabeller.

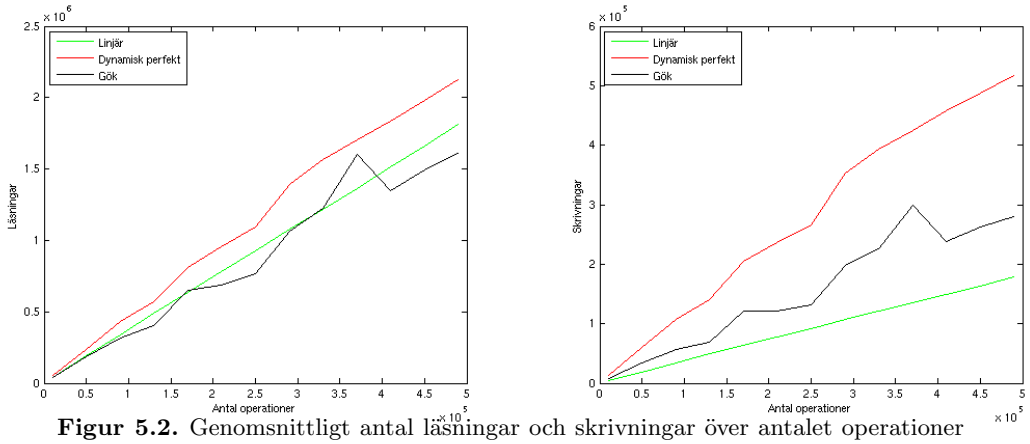


Figur 5.1. Genomsnittligt antal läsningar och skrivningar över antalet operationer i testfallen för alla strategier. Observera skillnaden på axeldimensioner i graferna ovan.

Anledningen till att alla statiska hashstrategier ger samma antal skrivningar är att det bara skrivs vid insättning, att skriva till en länkad lista för universell hashning och Knuth-hashning tar precis en skrivning. Det tar givetvis också bara en skrivning för statisk perfekt hashning, eftersom den bara behöver skriva elementet på den plats där man hashat den till.

Antalet läsningar är ungefär lika för universell hashning och Knuth-hashning på grund av att de i praktiken är väldigt lika. Skillnaden är att man får teoretiskt bra egenskaper om man slumpar fram hashfunktionen. Som vi ser är det nästan ingen skillnad mellan dem i praktiken. Om man väljer nycklar speciellt för att orsaka krockar kan man möjligtvis se större skillnader.

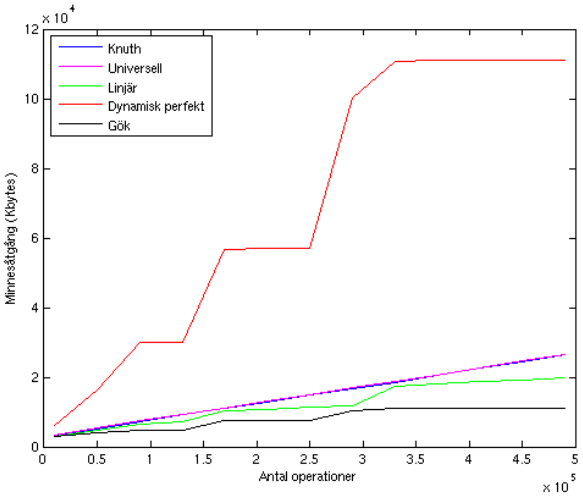
Eftersom man inte bör jämföra statiska respektive dynamiska strategier då de inte har samma användningsområde presenteras i figur 5.2 resultaten från enbart de dynamiska strategierna.



Figur 5.2. Genomsnittligt antal läsningar och skrivningar över antalet operationer i testfallen för dynamiska strategier. Observera skillnaden på axeldimensioner i graferna ovan.

Vi ser att dynamisk perfekt hashning har både fler läsningar och skrivningar än de andra dynamiska strategierna enligt figur 5.2. Anledningen till detta är antagligen att kostnaden för att bygga om tabellen när man minskar eller ökar storleken på den är stor eftersom, alla element då tas ur tabellen och sätts in igen. Man bygger både om inre och yttre tabeller i dynamisk perfekt hashning, medan man bara bygger om yttre tabeller i till exempel gök-hashning (det finns inga inre tabeller att bygga om). I linjär hashning bygger man aldrig om hela tabeller utan man använder bara delningsfunktionen på en hink i taget.

Vi ser att vad det gäller läsningar presterar gök-hashning och linjär hashning ungefär lika bra, däremot är linjär hashning bättre vad det gäller skrivningar.



Figur 5.3. Genomsnittligt minnesåtgång över antalet operationer i testfallen för alla strategier utom statisk perfekt hashning.

Figur 5.3 innehåller en graf över uppmätt maximal minnesåtgång vid körning av testfallen. Vi lägger inte så stort fokus på detta. Man kan dock notera att dynamisk perfekt hashning tar mer minne än övriga strategier. Minnesåtgången för statisk perfekt hashning är inte med i grafen, men vi kan konstatera att det är möjligt att implementera den utan prestandaförlust så att den tar lika mycket minne som dynamisk perfekt hashning[2].

Minnesåtgången är ganska jämn för de andra strategierna. Det är möjligt att implementationerna inte är minnesoptimala, resultatet ska således bara tas som ett exempel på hur minnesåtgången kan se ut.

[2]Man kan sätta in alla element i nyckelmängden och sedan tömma den dynamiska tabellen utan att ändra storleken.

5.3 Slutsatser

Enligt resultaten som presenteras i sektion 5.2 är det bästa valet för statiska nyckelmängder statisk perfekt hashning. Strategin presterar bäst med avseende på läsningar och skrivningar. Problemet är att den tar mycket minne och för stora nyckelmängder blir det ett möjligt problem. Att skapa tabellen är kostsamt och man ska inte använda statisk perfekt hashning om man inte utför många operationer efter att man har skapat tabellen. Det motsvarar vad teorin säger eftersom det givetvis är kostsamt för antalet läsningar att hantera krockar i tabellen.

Om man vill välja en enkel hashtabell med kedjning så ger det inte någon fördel att välja universell hashning före Knuth-hashning eftersom de ger nästintill identiska resultat för rimliga data. Det är då lämpligt att välja Knuth-hashning då den har en aningen enklare hashfunktion än universell hashning.

Vad det gäller dynamiska hashtabeller är det inte lönsamt att använda dynamisk perfekt hashning. Gök-hashning och linjär hashning visar sig vara ungefär lika bra, men linjär hashning är lite bättre vad det gäller skrivningar. Även fast linjär hashning inte är perfekt utan bara en heuristik så presterar den alltså bäst av de dynamiska strategierna vad det gäller läsningar och skrivningar. Både linjär hashning och gök-hashning har dessutom mycket mindre minnesåtgång än vad dynamisk perfekt hashning har.

En tänkbar anledning till att att linjär hashning är bättre än perfekta hashstrategier är att, man till skillnad från de perfekta strategierna, aldrig behöver bygga om hela tabellen. Strategin tjänar på att vara enkel. Som med många andra heuristiska lösningar fungerar den bra oftast, men det finns undantagsfall, vilket gör att den är användbar i praktiken.

Litteraturförteckning

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms, Third Edition* (2009) MIT Press, sid 262-282.

[2] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, Robert E. Tarjan *Dynamic Perfect Hashing: Upper and Lower Bounds* (1994) SIAM Journal on Computing.

[3] Michael L. Fredman, János Komlós, Endre Szemerédi *Storing a Sparse Table with 0(1) Worst Case Access Time* (1984) Journal of the ACM, Volume 31, Issue 3.

[4] Donald E. Knuth *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*, (1998) Massachusetts: Addison-Wesley, sid 513-549.

[5] Witold Litwin *Linear Hashing: A New Tool For File And Table Addressing* (1980) Proc. 6th Conference on Very Large Databases.

[6] Rasmus Pagh, Flemming Friche Rodler *Cuckoo Hashing* (2003).

[7] Alan Siegel *On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications* (1989) Annual IEEE Symposium on Foundations of Computer Science, sid 20-25.

[8] Java 1.4.2 library documentation: Hashtable, `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html`, hämtad 2010-03-02.

[9] Java 1.4.2 library documentation: String, `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html`, hämtad 2010-03-02.

[10] PostgreSQL 8.1 documentation, `http://www.postgresql.org/docs/8.1/static/sql-createindex.html`, hämtad 2010-03-02.

[11] Python standard library documentation, `http://docs.python.org/library/stdtypes.html#mapping-types-dict`, hämtad 2010-03-02.

[12] GNU Libc, version 2.9, källkodsfil `misc/hsearch_r.c`. Tillgänglig `http://ftp.gnu.org/gnu/glibc/`.

[13] Implementationer av hashstrategier, av H. Flinck, O. Krüger. Tillgänglig `http://www.nada.kth.se/~hflinck/hashstrategier/`.