Effektiv implementation av hashtabeller

TOMMY PETTERSSON
och HELENA SJÖBERG

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2012

Effektiv implementation av hashtabeller

T O M M Y P E T T E R S S O N
o c h H E L E N A S J Ö B E R G

DD143X, Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik 300 högskolepoäng
Kungliga Tekniska Högskolan år 2012
Handledare på CSC var Mads Dam
Examinator var Mårten Björkman

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/
pettersson_tommy_OCH_sjoberg_helena_K12056.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Effektiv implementation av hashtabeller

Sammanfattning

Denna rapport syftar till att undersöka effektiviten hos de två hashningsmetoderna Cuckoohashning och den relativt nya Hopscotchhashning jämfört med Javas inbyggda java.util.hashmap. För att utföra detta implementerar vi två hashtabeller som använder sig av de respektive metoderna i Java och mäter deras prestanda med hänsyn till snabbhet och minnesåtgång. HashMap visar sig vara snabbare på enskilda operationer överlag, men våra implementationer är snabbare i värsta fallet. De resultaten analyseras och diskuteras. Vi diskuterar också hur de två hashtabellerna skulle kunna parallelliseras. I slutet av rapporten bifogar vi våra implementationer i helhet.

Effective implementation of hashtables

Abstract

This report is written in order to evaluate how efficient two different hashing schemes, Cuckoo Hashing and the fairly new Hopscotch Hashing, are in comparison to the native HashMap provided in java.util.hashmap. In order to investigate this, we implement the two hashing schemes in Java and measure the performance in respect to speed and memory usage. HashMap is found to be quicker for individual operations, but our implementations are quicker in the worst-case. These results are analyzed and discussed. We also discuss how the two schemes could be parallelised. Finally, we provide our implementation of CuckooHashMap and HopscotchHashMap.

Innehållsförteckning

1. Introduktion

En hashtabell är en grundläggande datastruktur vars prestanda många program idag vilar på. I datastrukturen är elementens nycklar är kopplade till ett visst index i tabellen som man får fram via att låta nyckeln köras genom en hashfunktion.När man vill hämta ett objekt hashar man objektets nyckel och får fram det index i tabellen där elementet finns att hämta, om det existerar. När man implementerar en hashtabell är en viktig fråga hur man hanterar kollisioner, det vill säga när flera olika nycklar hashas till samma position. Vi har valt att titta närmare på två algoritmer för kollisionshantering och se dels hur de kan optimeras var för sig och dels hur de beter sig i jämförelse med den hashtabell som Javas standardbibliotek erbjuder, java.util.hashmap.

Cuckoohashning är en i teorin enkel algoritm där man använder sig av flera hashfunktioner istället för en som i konventionella hashtabeller. Varje element kan placeras i lika många antal hinkar som det finns hashfunktioner, men bara i en hink åt gången. Vi implementerar en version med två hashfunktioner. Vid kollisioner knuffas det existerande elementet ut och placeras i sin alternativa hinkar. Är den hinken full så knuffas det elementet ut i sin tur och proceduren upprepas tills en ledig plats hittats eller ett förutbestämt antal iterationer uppnås, varvid hashtabellen byggs om med nya hashfunktioner.

Hopscotchhashning är en annan algoritm som löser kollisioner genom att koppla varje hink i tabellen till ett grannskap av närliggande hinkar. Istället för att lägga det nya elementet på första bästa plats oavsett om det är upptaget eller inte försöker man här hitta en ledig plats. Finns det ingen ledig plats i grannskapet flyttar man ett element till en annan plats inom dennes grannskap för att sedan sätta in det nya elementet på den lediga platsen som uppstått.

2. Problemformulering

Vår problemformulering är som följer:

- *Hur implementerar man hashning effektivt?*

För att hashningen ska vara effektiv bör insättning, borttagning och uppslagning ske så pass snabbt som möjligt. Hur pass snabbt dessa går att göra avgör effektiviteten. Även mängden utrymme man väljer för lagringen påverkar hastigheten men också utrymmet som krävs vid exekveringen, vilket gör att vi även behöver ha med och testa den faktorn. Vilken prestanda ger de olika algoritmerna vi tittar på? Vilken/vilka är mest effektiv och vilken bör man välja i vilken situation?

Vår målsättning är att utreda vilka fördelar och nackdelar våra två huvudalgoritmer har och i vilka situationer ett val av respektive kan vara att föredra i jämförelse med andra.

3. Redogörelse av samarbete

Helena Sjöberg skrev implementationen av HopscotchHashMap, all testkod som användes vid testerna samt utförde alla tester. Tommy Pettersson skrev implementationen av CuckooHashMap samt den testkod för ConcurrentHashMap som Helenas slutliga parallelliserade testkod byggde på. Båda har deltagit i analysen av resultaten.

4. Teori

Hashning används för att kunna hantera stora mängder data på ett sådant sätt att det ändå går snabbt att nå just det man letar efter. En hashfunktion gör om varje nyckel till ett värde som fungerar som ett index, en position, som leder till den plats elementet ska lagras. Detta ger själva grundidén till hashning och skapar en metod att lagra data med en i bästa fall konstant tid för insättning och snabb uppslagning och borttagning. Datastrukturen som erhålles kallas för en hashtabell. En effektiv hashfunktion ska sprida innehållet så jämnt som möjligt över de index som finns i hashtabellen, där varje index leder till en hink där elementen ifråga lagras. En viktig fråga är hur dessa hinkar ska hanteras, framför allt när kollisioner uppstår, det vill säga när flera olika nycklar genom hashfunktionen får samma index. Hur man väljer att behandla det påverkar självklart prestandan för tabellen.

4.1.1. Dynamisk storlek

En hashtabells storlek bör kunna varieras dynamiskt. I allmänhet är det en fördel för prestandan om tabellen hålls gles, vilket innebär att storleken kan behöva utökas innan tabellen är helt full. Detta tar mer plats i minnet, men gör datastrukturen snabbare. När en tabell utökas utförs också en så kallad omhashning. Det innebär att alla elements positioner i strukturen räknas om för att matcha den nya storleken. På samma sätt kan man förstås också minska storleken på tabellen vid borttagning för att på det sättet inte få en alltför gles struktur som tar onödigt mycket minne.

4.1.2. Perfekt hashning

Optimalt vill man att en hashfunktion tilldelar en unik nyckel för varje element man inför i tabellen, det vill säga att den är injektiv. Lyckas man med detta har man garanterad konstant värstafallstid för uppsökning i tabellen eftersom man bara behöver titta på den platsen i tabellen som hashfunktionen ger, istället för att utföra någon form av sökning. En sådan funktion är dock opraktisk för stor indata och kräver generellt att man vet något om indata, hur många nycklar det finns totalt i datamängden. Om hashtabellens storlek dessutom blir lika med antalet nycklar kallas det för en *minimal perfekt hashning*. Med andra ord, om vi har n nycklar så kopplas de till index 0, 1, ..., n-1 i tabellen.

4.1.3. Om parallellisering av algoritmerna

De algoritmer vi tänker implementera kommer vara sekventiella, det vill säga de utförs i en sekvens och endast en kodrad exekveras i taget. Det går dock att parallellisera dessa algoritmer. Det innebär att delar av koden körs samtidigt, parallellt med varandra, men att de ändå slås samman i slutet så att de ger ett korrekt resultat. Det kan ske på olika nivåer. Dels kan olika metoder genomföras parallellt, men det går att parallellisera även inom enskilda metoder också.

I och med att flerkärniga processorer blir allt vanligare finns det större möjligheter att kunna utnyttja den potential dessa har genom just parallellisering. När flera anrop går att utföra parallellt minskar den genomsnittliga väntetiden om parallelliseringen har utförts på ett bra sätt. Detta i och med att man utnyttjar den fulla kapaciteten i datorn istället för att stora delar får vila medan de väntar på att det första anropet ska bli färdigt.

Det är dock viktigt att notera att för att den högre effektiviteten ska uppkomma krävs det att man har parallelliserat på ett effektivt sätt - ett sätt att hantera att flera trådar vill nå samma data

samtidigt är att använda sig av lås, så att datan är låst för alla utom de som har tillgång till en. Risken med det är att deadlock uppstår, det vill säga att alla trådar väntar på varandra och det hela står still.

4.1.4. HashMap

Javas implementation av en hashtabell ingår i dess standardbibliotek under java.util.HashMap. Kollisionshantering sköts genom separat länkning, vilket innebär att varje hink består av en länkad lista. Om flera nycklar hashas till samma hink läggs de till i den länkade listan. Värsta fallet blir när alla element hashas till samma index - då urartar hashtabellen till en länkad lista och uppslagning får tidskomplexitet O(n) istället för O(1).

4.1.5. Concurrent HashMap

Detta är en trådsäker implementation av HashMap och återfinns i Javas standardbibliotek i java.util.concurrent.ConcurrentHashMap. Den tillåter att flera trådar kan läsa och skriva till den utan att den returnerar felaktig eller utdaterad data. Detta uppnår man genom att man, istället för att använda metoden put, använder sig av metoden putIfAbsent, som atomärt lägger till elementet. Detta innebär att operationen utförs utan att andra trådar kan komma in och störa utförandet.

4.2. Cuckoohashning

Cuckoohashning är en metod där man har två eller fler hashfunktioner (h1,h2, …, hn) som ger olika möjliga index placerade i en hashtabell där varje hink bara får innehålla ett element. När data ska sättas in i hashtabellen sätts den på det index den får av en av hashfunktionerna. Om platsen var upptagen omplaceras den data som fanns där till en ny plats. Om denna nya plats också var upptagen omplaceras datan som fanns där. Så fortsätter det tills man hamnar på en ledig plats eller man nått ett förutbestämt antal iterationer. I det senare fallet görs tabellen om. För att hitta elementet behöver man då även i värsta fallet enbart leta på samma antal ställen som antalet hashfunktioner. Då antalet hashfunktioner är konstant blir också tiden för att hitta elementet konstant även i värsta fallet.

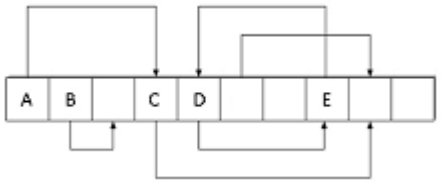
4.2.1. Insättning

Låt oss titta på ett exempel av insättning i en datastruktur med två hashfunktioner. När vi vill att elementet x ska sättas in i datastrukturen försöker vi först placera den på position h1(x) i motsvarade hashtabell. Om denna plats är upptagen så placerar vi den istället i position h2(x) i den andra hashtabellen. Utifall det är så att båda positionerna är upptagna så omplacerar vi detta element y till dess alternativa plats. Härifrån har metoden fått sitt namn, då den efterliknar hur gökungen knuffar ut sina syskon ur deras bo.

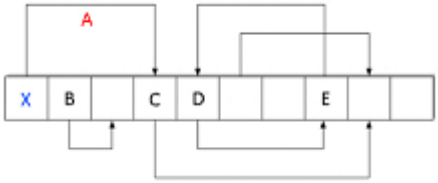
Detta upprepas enligt en girig algoritm tills dess att vi lyckas hitta en ledig position eller att vi når ett förutbestämt antal iterationer, för att undvika att fastna i en oändliga loop där vi skyfflar om värdena för evigt. Skulle detta inträffa så utförs en omhashning - hela datastrukturen beräknas om med nya hashfunktioner. *Se pseudokod för insättning nedan.*

```
insert(X){
      if T[ h1(X) ] == x or T[ h2(X) ] then do return;
      pos :=  h1(X)
      for 0 to maxLoops do {
            if T[ pos ] == null then do { T[ pos ] := X; return}
            swap(X,T[ pos ])
            if pos == h1(X) then do pos := h2(X)
            else do pos := h1(X)
      }
      rehash(X)
      insert(X)
}
```

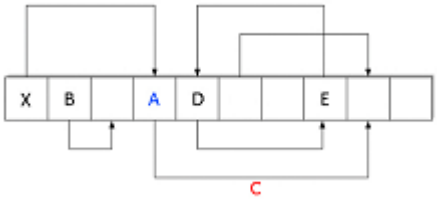
Bild 1 *Illustrerad insättning för Cuckoohashning. En tom ruta representerar en tom position i tabellen. En pil pekar på ett elements alternativa position i tabellen.*



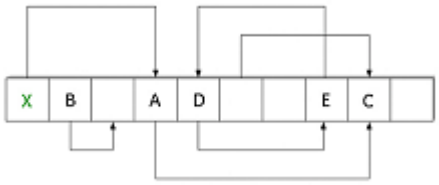
En representation av en tabell med elementen A, B, C, D och E insatta.



Element X sätts in, men hashas till samma position som A låg i. Därför sparkas A ut och ska sättas in i sin alternativa position.



Element A har nu satts in i sin alternativa position, men där låg redan element C. C sparkas ut på samma sätt som A nyss gjordes, och placeras i sin egen alternativa position.



Element C's alternativa position var ledig, varvid insättningsskedjan tar slut och insättningen lyckades. Notera att om X hade hashats till element D eller E hade algoritmen fastnat i en oändlig loop där de knuffar ut varandra fram och tillbaka, vilket visar på behovet av variabeln maxLoops.

4.2.2. Hitta och ta bort

För att hitta ett element behöver man i värsta fallet enbart leta på samma antal ställen som antalet hashfunktioner. Då antalet hashfunktioner är konstant blir tidskomplexiteten för att hitta elementet också konstant även i värsta fallet. Vid borttagning sätts elementet till null innan return. *Se pseudokod för uppsökning nedan.*

```
contains(X){
      if T[ h1(X) ] == X then do return true;
      if T[ h2(X) ] == X then do return true;
      return false;
}
```

4.2.3. Nackdelar

Cuckoohashning är beroende av en bra hashfunktion.[1] Med bra menas i det här avseendet att den är snabb och att den sprider nycklarna jämt över adressrymden. Man behöver dessutom två eller fler unika hashfunktioner och möjlighet att skapa nya när det behövs. Att skriva en sådan hashfunktion ligger inte inom avgränsningen för den här rapporten, därför tänker vi använda oss av en redan befintlig hashfunktion som uppfyller dessa krav. Att hitta en sådan kommer vara kritiskt för vår implementation av Cuckoohashning.

4.2.4. Prestanda

Cuckoohashning påstås av författarna prestera dåligt när tabellen är mer än 50% full.[2] Detta eftersom kedjorna när ett element skall sättas in blir för långa, vilket också ger fler omhashningar som tar tid att utföra. Vår implementation kommer därför kräva någon form av mekanism som håller reda på när tabellen blir för full och vid behov öka storleken.

4.2.5. Parallellisering av Cuckoohashning

Cuckoohashning kan parallelliseras för bättre prestanda[3], speciellt vid hög belastning. I Real-time Parallel Hashing on the GPU[4] parallelliserar författarna Cuckoohashning på en grafikkortsprocessor med hjälp av NVIDIA's CUDA, som är en arkitektur för att möjliggöra körning av vanlig kod på grafikkortet.

De använde sig utav tre hashfunktioner och subtabeller T1, T2 och T3. Först försöker de simultant placera alla element i T1. Deras metod förlitar sig på att endast en skrivning lyckas vid kollisioner och att trådarna kan identifiera vilken som faktiskt lyckades. De element som inte lyckades skrivs då in i T2, och de som inte heller lyckas där skrivs in i T3. De som inte lyckas nu försöker sparka ut element i T1 till T2. Sedan fortsätter processen så tills alla element är inlagda i datastrukturen eller för många iterationer har nåtts och omhashning vidtas.

4.3. Hopscotchhashning

Hopscotchhashning är en algoritm som bara använder en hashfunktion för att få en hink för elementet men där elementet ändå kan placeras på en av ett flertal olika platser. Hinkarna är nämligen sammankopplade så att varje hink ingår i ett grannskap, vars storlek bestäms vid implementeringen. Idén är att man vid insättning ska kunna garantera att elementet om det inte hamnar just i sin hink ändå hamnar i sitt grannskap. Det gör att man när man ska hämta ett

[1] Herlihy mfl, 2008, s. 2

[2] Herlihy mfl, 2008, s. 2

[3] Mitzenmacher, M., s. 8

[4] Acantara mfl

element eller undersöka om ett element finns i hashtabellen enbart behöver titta i det grannskap hashfunktionen ger en.

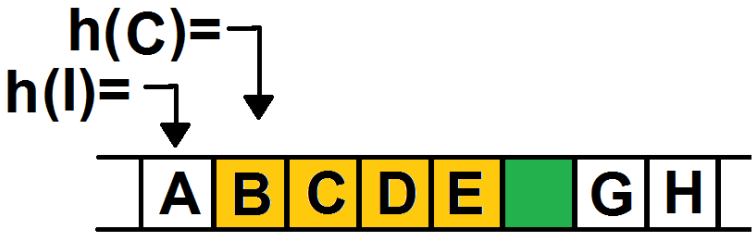
4.3.1. Insättning

Metoden för insättning kan liknas vid att hoppa hage, vilket på engelska blir hopscotch, och är anledningen till metodens namn hopscotchhashning. Det finns två huvudfall av insättning. I båda fallen börjar man med en linjärsökning genom tabellen med start i den hink hashfunktionen leder en till. Denna linjärsökning fortgår tills man antingen hittat en ledig plats eller tills man har gått ett varv och tabellen behöver omhashas. Om den lediga plats man hittar ingår i grannskapet för hinken hashfunktionen först gav placeras elementet där. Annars blir det mer komplicerat.

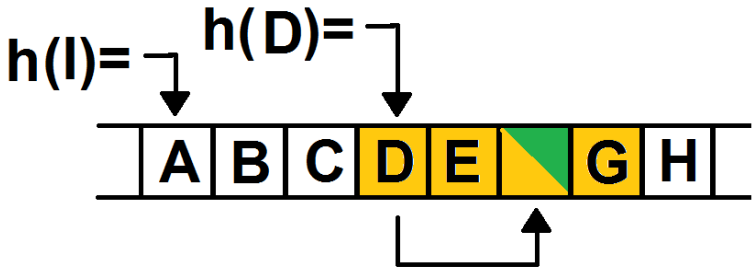
När den första lediga platsen ligger utanför grannskapet försöker man hitta ett element som ligger tidigare i tabellen för att flytta till den lediga platsen. Då uppstår en ny ledig plats. Om den ligger inom grannskapet lägger man det nya elementet där, annars får man leta efter ett annat element att lägga där genom att upprepa proceduren. I värsta fall går det inte och då får man istället göra en omhashning.

Bild 2 *Illustrerad insättning för hopscotchhashning. Orange symboliserar grannskap och grönt symboliserar ledig position i tabellen.*

Elementet I ska sättas in och hashvärdet av I beräknas. Den först lediga positionen ligger dock inte inom grannskapet.



C är det första elementet som potentiellt kan ha den lediga platsen i sitt grannskap. Cs hashvärde ger dock en position där den lediga platsen inte ingår i grannskapet.



Nästa potentiella element är D. Ds hashvärde ger ett grannskap där den lediga positionen ingår. D flyttas därför till den lediga platsen.

När D har flyttats upstår en ny ledig plats, som nu ligger inom I:s grannskap. Då kan I sättas in på den lediga platsen.

Så här blir då slutresultatet.

4.3.2. Borttagning, innehåller och hämtning

För borttagning, för hämtning och för att se om ett element finns i tabellen gör man helt enkelt en linjärsökning i det grannskap som tillhör den hink hashfunktionen ger en.

```
delete/containsKey/get( key){
      pos := hash ( key );
      while pos < hash( key ) + neighbourhoodSize{
            if t[pos]. key = key{
                  do action
            }
      }
}
```

4.3.3. Fördelar och nackdelar

Jämfört med `HashMap` bör insättningar gå långsammare eftersom förflyttningar inom grannskapet kan behöva utföras. Om hashfunktionen ger en sned fördelning kan tabellen tvingas bli gles och ta stor plats. Den är dock inte lika beroende av sin hashfunktion som Cuckoohashning, och denna kan därför vara väldigt enkel.[5]

4.3.4. Prestanda

Hopscotchhashnings insättningskedjor kan inte bli cykliska som i cuckoohashning. Det medför att Hopscotch har en mycket bättre prestanda även när tabellen är tätare än 50%. Vidare är det också så att om storleken på grannskapen är väl vald går det att lagra hela grannskapet i cachen. Det ger bra rumslokalitet och linjärsökning efter rätt element ska därför ske snabbt.[6]

4.3.5. Parallellisering av Hopscotch

Hopscotchhashning lämpar sig väl för parallellisering, och en sådan version är implementerad i rapporten Hopscotch Hashing.[7]

[5] Herlihy mfl, s. 2
[6] Herlihy mfl, s. 2
[7] Herlihy mfl, s. 2

5. Genomförande

Vi har skrivit kod för de två hashningalgoritmer vi har studerat, hopscotchhashning och cuckoohashning, i Java. Då dessa två ska jämföras med Javas egna `HashMap` så har vi valt att använda samma metodnamn för de grundläggande funktionerna och också låta dessa ha samma effekt. Det innebär bland annat att vår "insert" kallas put och om man försöker sätta in ett nytt element med en nyckel som tidigare redan finns i tabellen byts bara datan för denna nyckel ut till den nya datan, precis som i den put som finns i `HashMap`.[8] Vi har också skrivit en egen klass för att generera hashfunktioner, även om detta inte var den ursprungliga planen. Tester har genomförts, både för att bestämma värden på de variabler som används i algoritmerna men också för att bedöma hur pass bra slutresultatet var i jämförelse med just HashMap och den trådade varianten av den.

5.1. Krav

I de implementationer som vi gör måste tre funktioner finnas, insättning, borttagning, hämtning och uppslagning. Tabellen ifråga får inte ta för stor storlek i förhållande till antal element, minst 1-2% täthet är definitivt ett minimumkrav. Elementen ska kunna lagras med valfritt objekt som nyckel och valfritt objekt som data, vilket innebär att användaren måste få möjlighet att definiera detta. Det bör också finnas en gräns för den maximala tätheten, kallad `loadFactor`, för att undvika långa försök med insättning när risken är stor att en omhashning ändå kommer att behövas.

5.2. Element

`Element` är en enkel klass som används av både `CuckooHashMap` och `HopscotchHashMap`, som båda har en enkel lista av `Element`. Den har en enkel konstruktor tillsammans med tre fält. `key` och `data` lagrar just nyckeln respektive datan, medan `hashinfo` lagrar den information hashfunktionen som använts för att placera in elementet ger. Detta för att inte behöva beräkna om den hashade positionen vid omflyttning i tabellen, vilket gör att insättningar kan gå betydligt snabbare, även om det samtidigt gör att tabellen tar större utrymme att lagra.

5.3. Hashfunktioner

Vi valde initialt att avgränsa vårt arbete på ett sådant sätt att vi inte skulle skapa helt egna hashfunktioner för att kunna fokusera på algoritmerna för hashningen i sig och för att det redan idag finns väldigt mycket kring just hashfunktioner. Den stora problematiken rörde dock att ha tillgång till en generator som enkelt kunde ge två hashfunktioner med väsentligt skilda resultat då cuckoohashning kräver minst två olika hashfunktioner. Det faktum att det behövdes flera hashfunktioner gjorde också att hastigheten på hashfunktionen i sig blev en prioritet, förutom de klassiska frågorna kring att få en så jämn fördelning som möjligt.

5.3.1. hashCode()

Det första alternativet var att använda sig av den `hashcode`-metod som alla Java-objekt har.[9] Detta innebär att den kan beräknas för alla potentiella nycklar, vilket dels gör den enkel att använda, men ger också användaren en möjlighet att skapa egna klasser för sina nycklar med egen-definierade hashfunktioner som genererar det returvärde som `hashcode` då ger, något som

[8] Oracle

[9] Goetz, B.

kan vara speciellt användbart vid nycklar med en sned fördelning eller med speciella egenskaper. `Hashcode` ger dock en int, det vill säga ett heltal mellan -2^31 och 2^31-1, dessa värden inkluderat.[10] Vår tabell kan ha en storlek som är betydligt mindre än så, vilket gör att vi behöver begränsa värdena från 0 till tabellens storlek subtraherat med 1. Det naturliga valet var då att använda oss av modulusoperatorn %. Problemet med denna är att den bara ger ett värde för varje nyckel. Det gör att den går utmärkt att använda in hopscotch, men tyvärr inte i cuckoohashning förutsatt att de använda tabellerna ska ha samma storlek. Det gör också att det inte går att hasha om en tabell utan att ändra storlek, enda skillnaden man kan få är att elementen byter plats, men en tidigare krock kommer fortfarande att kvarstå. Med olika tabellstorlekar i cuckoohashning och därmed också olika moduli går det visserligen att komma runt problemet med att skapa fler hashvärden för samma nycklar men då krockproblemed fortfarande kvarstår och det kräver en annan implementation av tabellerna anser vi att den här hashfunktionen ändå är olämplig att använda för implemenationen av cuckoohashning. Samtidigt finns fördelen att det här i allmänhet går relativt snabbt då de operationer som görs i normalfallet inte tar någon längre tid att utföra.

5.3.2. MAD-hashning

Nästa alternativ var att använda oss av en så kallad MAD-hashning, Multiply-Add-Divide, där vi hittade ett program skrivet av Michelangelo Grigni.[11] Upphovsmannen till den här hashfunktionen skriver själv att hans lösning är en komprimerad hashcode som kan ge problem om flera element har samma hashcode. Vad den bygger på är att den har två olika heltalsvariabler, som den utför en addition och multiplikation på elementets hashcode med, för att sedan ta modulus tabellens längd. Variablerna genererades slumpmässigt mellan 0 och tabellens storlek. Vissa ytterligare beräkningar gjordes på multiplikationsvariabeln. Pseudokod, där mul och add är våra två heltalsvariabler:

```
hash = (element.hashCode() * mul + add) % array.length
```

Genom att låta faktorn som multipliceras med och termen som adderas ha olika värden i respektive hashfunktion kan två olika hashfunktioner fås, vilket är precis vad som behövs för CuckooHashMap. Testning utav hashfunktionen visade dock på problem med kollisioner, där samma kollisioner kvarstod även vid ombyggnad av hashfunktionen. Varför det blir så härleds i sektion *6.1 Hashfunktioner*.

5.3.3. Pseudoslumpmässig hashfunktion

Detta var den hashfunktion som fungerade bäst i testningen. Den använder sig utav javas inbyggda pseudoslumptalsgenerator. När ett element ska hashas används elementets `hashcode` som frö vilket garanterar att samma hash kan fås igen när man vill undersöka om ett element redan är inlagt i tabellen. Om tabellen ännu inte har hashats om och nya hashfunktioner byggts kommer generatorn direkt generera ett heltal mellan 0 och tabellens storlek och använda det som index. Hashfunktion nummer två låter generatorn göra precis samma arbete, bara att den istället returnerar nästa pseudoslumpmässiga heltal med samma frö som tidigare, det vill säga heltal nummer två i den följd av heltal generatorn skapar.

[10] Java2s. Integer: MAX, MIN VALUE

[11] Michelangelo Grigni

När hashfunktionerna byggs om låter vi bara generatorn generera fler tal och väljer dessa som index. Hashfunktion *i* kommer alltså generera *i* slumptal och returnera det *i*:e talet som hashkod. Detta gör att anrop till funktionen tar längre tid ju fler omhashningar som gjorts. Hashfunktionen hämtades från en fritt tillgänglig implementation av en `CuckooHashMap`.[12]

5.4. CuckooHashMap

Här följer en överblick av de intressanta metoderna i `CuckooHashMap`. Framförallt tittar vi närmare på de mer komplicerade mekanismerna som utgör insättning, där även omhashning av tabellen sker.

5.4.1. Put

Den publika metoden `put` utför själva insättningen, tar två parametrar, `key` och `data`, och returnerar en boolean som talar om huruvida insättningen lyckats. Den kallar i sin tur på en privat hjälpmetod som tar en extra parameter som håller reda på hur många gånger elementet försökts lägga in.

Först av allt kollar `put` att tätheten i tabellen inte är större än den maximala som loadfactor specificerar. Om detta inträffat (och storleken på tabellen inte är alltför stor redan) så tvingas en omhashning fram. Sedan anropas hjälpmetoden `insertElement`, som returnerar null om insättningen av elementet är lyckad, vilket gör att även `put` returnerar null och insättningen kan avslutas. I annat fall returneras det element som misslyckades att sättas in. Mer om `insertElement` följer senare i texten.

Efter det genomförs ytterligare en kontroll - om det är så att listans storlek överskrider maxstorleken och omhashning har genomförts ett flertal gånger utan att alla element har kunnat bli inlagda har insättningen misslyckats och `put` returnerar false. Annars körs en omhashning och ett nytt försök med insättning av elementet genomförs.

5.4.2. insertElement

`insertElement` är den metod som utför själva insättningen av ett element. Genom att utnyttja variabeln `hashinfo` i den privata `Element`-klassen på så sätt att vi alltid låter den innehålla hashkoden för elementets alternativa plats, behövs endast två hashfunktionsberäkningar per insättning. Om insättningen lyckas returneras null, i annat fall returneras det element vi misslyckat att sätta in. Om en nyckel redan existerar i tabellen så uppdaterar istället dess datafält med den nya datan. Detta räknas som en lyckad insättning.

Själva implementationen av att sätta in elementet, sparka ut det existerande, låta det sparka ut nästa och så vidare görs i for-loopen. Inga hashberäkningar utförs, utan om nästa plats är tom sätts helt enkelt elementet in och insättningen är klar. I annat fall byts elementet ut och `hashinfo`-fältet uppdateras till det korrekta värdet. Om antal omflyttningar av element uppgår till det antal som specifierats som max antal försök tidigare returneras det element som vid just det tillfället saknar en position i tabellen, så att den metod som kallat på `insertElement` kan hantera den.

[12] Alvergren, J.

5.4.3. Rehash

Omhashningsfunktionen har två olika lägen. Om insättningen av ett element misslyckas är avsikten att genomföra försöket med insättningen igen, fast med nya hashfunktioner. Därför skapas då ett par nya hashfunktioner och alla element läggs in igen. Detta utförs i else-delen av if-satsen.
I det andra fallet, då försök till insättning har genomförts `MAX_BOUNCE` gånger utan att det lyckats (eller `loadFactor` är nådd), görs samma sak som i det första läget, bara att storleken på tabellen dubblas först. `MAX_BOUNCE` är i vår implementation hårdkodad till 5.

5.4.4. containsKey, delete och get

`containsKey` är mycket simpel - eftersom vi kan garantera att ett element ligger i någon av de två positioner hashfunktionen ger behöver man enbart titta i dessa två positioner och undersöka om den nyckel som efterfrågas finns där. Både `get` och `delete` fungerar på samma sätt som `containsKey`, även om olika operationer självklart genomförs när själva nyckeln hittats. Värt att notera för `delete`-funktionen är att någon justering av tabellens storlek inte genomförst när element tas bort.

5.4.5. Parametrar som påverkar

Vid skapandet av en ny `CuckooHashMap` så kan följande tre parametrar ställas in.

`initialSize` - Storleken som listan har vid skapandet när den är tom. Sätts denna stor kommer tabellen uppenbarligen i början vara mycket gles.

`maxLoops` - Denna parameter begränsar hur många gånger element flyttas runt vid insättning. Man vill undvika att hamna i en oändlig loop, där samma element bara flyttas till nya positioner utan att det kan sättas in. Om denna är för liten kommer det resultera i onödiga omhashningar.

`loadFactor` - Hur full tabellen kan bli innan en omhashning tvingas utföras. Om `loadFactor` sätts till 0.5 kommer tabellen aldrig bli tätare än 50%.

5.5. HopscotchHashMap

Här tittar vi närmare på `HopscotchHashMap`. Precis som för `CuckooHashMap` så är det insättningen som är mest intressant.

5.5.1. Put

`Put` är metoden för insättning och är den som är mest komplicerad. Vi börjar med att undersöka om antalet element överstiger det maximala antalet element som bör vara i tabellen med avseende på den inställda `loadFactor`, som avgör den maximala tätheten. Om den är för stor genomförs en omhashning med dubblering av storleken genast. Sedan görs en linjärsökning genom grannskapet som elementet bör ligga i om nyckeln tidigare lagts in i tabellen och fortfarande finns kvar. Hittas den ersätts den gamla datan med den nya och metoden returnerar.

Efter det påbörjas en loop. Från den plats i tabellen som hashfunktionen ger med nyckeln som argument gås plats för plats genom i tabellen tills ett varv har uppnåtts för att leta efter en ledig plats. Om den lediga platsen ligger inom grannskapet läggs det nya elementet in i tabellen där. Om den ligger utanför anropas `moveTo` med indexet för den lediga platsen som argument som då försöker flytta ett element med ett lägre index till den lediga platsen, på ett sådant sätt att

elementet som flyttar fortfarande ligger inom sitt grannskap. Elementet som flyttas är det som potentialt ligger så långt bort ifrån den lediga platsen som möjligt för att den nya lediga platsen ska ligga så nära den plats där vårt ursprungliga element ska sättas in. Genom att använda sig av `hashinfo` kan detta göras snabbare då inga nya beräkningar med hashfunktionen behöver genomföras.

`moveTo` anropas upprepade gånger ända tills den lediga platsen ligger inom det ursprungliga elementets grannskap eller tills `moveTo` inte längre lyckas flytta ett element, i så fall går det inte att göra en insättning med det nya elementet, vilket gör att en omhashning tvingas fram. Efter omhashningen upprepas försöket att sätta in det nya elementet.

5.5.2. containsKey, delete och get

`containsKey, delete` och `get` är alla väldigt lika. De tar en en nyckel som argument, som i sin tur används som argument i hashfunktionen för att få ut ett startindex. Med utgångspunkt i detta startindex genomförs en linjärsökning inom grannskapet med kontroll av alla icke-tomma platser för att se om nyckeln ifråga finns lagrad där. Om den gör det utförs den faktiska operationen. `containsKey` returnerar true om nyckeln funnits, annars false, `delete` raderar elementet ifråga och `get` returnerar datan nyckeln leder till.

5.5.3. Rehash

Omhashningsmetoden är i sig relativt simpel även den. Den räknar om de fält vi har så att de stämmer överens med den nya storleken på tabellen som är dubbla den som tidigare var. Den skapar sedan en ny tabell av den nya storleken och sätter helt enkelt in alla element i den gamla tabellen i den nya. Om detta misslyckas görs ytterligare ett omhashningsförsök.

5.5.4. Parametrar som påverkar

Vid skapandet av en ny HopscotchHashMap kan följande parametrar ställas in:

`initialSize` - Storleken som listan har vid skapandet när den är tom. Sätts denna stor kommer vi uppenbarligen börja med en väldigt gles tabell.

`neighbourhoodSize` - Storleken på grannskapet. Antal element som vi i värsta fall behöver gå igenom för att hitta ett element.

`loadFactor` - Hur full tabellen kan bli innan en omhashning tvingas utföras. Om loadFactor sätts till 0.5 kommer tabellen aldrig bli tätare än 50%.

5.6. Test

Det finns i huvudsak två viktiga aspekter som behöver testas, minnes- och tidsåtgång. Då båda är kritiska faktorer krävs att båda undersöks noggrant. Med för hög minnesåtgång riskerar programmet att krascha och med för stor tidsåtgång finns en risk att det inte determinerar inom rimlig tid. Med hashtabeller blir det också oftast en avvägning mellan de två, en liten tidsåtgång kräver i allmänhet mer minnesutrymme då tabellen måste vara gles medan mindre minnesutrymme innebär att det tar längre tid då en tät tabell automatiskt blir resultatet och insättning då tar längre tid. Minnesutrymme har mätts i antal lagrade element i förhållande till antal platser i tabellen då detta förenklar för jämförelse och inte är beroende av vilken typ av data som hashtabellen lagrar.

Olika typer av operationer på tabellen behöver också testas. De olika operationer som bygger på uppslagning, det vill säga borttagning, returnering av element och undersökande om elementet finns lagrade är väldigt lika i sin struktur. Insättning å andra sidan är avsevärt annorlunda. Balansen mellan dessa två avgör typen av användning, och vad som bör prioriteras, vilket i sin tur påverkar värden på olika parametrar. Optimering av det ena kan ske på kostnad av det andra vilket innebär att testfall med grund i olika typer av användningen behöver skapas.

För vårt val av data att sätta in i hashtabellerna finns det flera olika alternativ. En redan färdig genererad och sparad data gör att exakt samma test kan köras flera gånger, vilket gör att vi lättare kan justera för avvikelser som uppstår av de program som körs i bakgrunden på datorn och andra saker vi inte kan påverka för varje enskild körning och få ett statistiskt säkrare medelvärde för just den körningen. Nackdelen är att bara det enskilda fallet testas, vilket gör att vi i så fall skulle behöva rendera många sådana fall, där också datamängden i varje fall måste vara mycket stor, över en miljon element, för att kunna testa beteende även för stora datamängder och för att få en bättre helhetsbild av prestandan med många olika sorters data. Denna renderade data skulle behöva sparas på fil för att vara tillgänglig vid varje körning och helst läsas in i förväg för att inte själva inläsningstiden ska påverka resultatet för själva körningen. När vi försökte med detta var påverkan stor på hur pass stora tabellerna vi kunde genomföra test på, nivån som krävdes för att en krasch skulle uppstå på grund av minnesbrist var avsevärt mycket lägre än med data som skapades vid behov. Vi valde att prioritera att kunna testa även med stora datamängder, vilket gjorde att denna metod inte kunde användas.

Helt slumpmässiga tal ger ett testfall med ny indata varje gång, vilket visserligen gör att man tappar kontrollen över vad som mer exakt testas, men man får ändå en vidd på testningen, speciellt då vi har valt att göra många körningar (100) i så många fall som möjligt. Vidden på testningen innebär rent konkret att både icke-optimal data och optimal data kommer att testas, vilket självklart ger en stor spridning både i krävt lagringsutrymme och tidsåtgång. Detta innebär att man vid enbart några körningar kan få vitt skilda resultat som därmed inte blir jämförbara mellan de olika algoritmerna. Att genomföra ett stort antal körningar, precis som gjordes, blir därmed ett krav.

5.6.1. Testa för att optimera

I det första teststadiet var fokus på att optimera de enskilda algoritmerna. Målet var att hitta en grundinställning på de parametrar som används för att algoritmen ska fungera så bra som möjligt i så många fall som möjligt utan att användaren själv ska behöva göra några specifika val. Finns det parametrar som den enskilda användaren ska ha möjlighet att påverka? Finns det gränser för parametrarna som de aldrig får hamna utanför? Finns det parametrar som inte påverkar alls? För hopscotch ska inte hashfunktionen spela någon större roll, men kan det fortfarande vara idé? För cuckoo ska det spela en större roll. Vilken av de vi hittat är bäst?

5.6.2. Test för att jämföra

När testen för att optimera de enskilda klasserna var utförda och värden på parametrar valts genomfördes också test för att jämföra de två klasser vi hade skapat med de två klasserna ur javas standardbibliotek, HashMap och ConcurrentHashMap. Detta för att få en referenspunkt för vilken nivå våra egna implementationer höll. Då det tyvärr inte var möjligt att jämföra den faktiska tabellstorleken har det enbart varit tid som studerats här, men då både för insättningar och för uppslagningar i tabellerna.

5.6.3. Miljö

Det fanns två potentiella alternativ inför valet av testmiljö. Det ena var de datorer som finns tillgängliga på KTH i de datasalar som vi hade tillgång till, medan det andra var egen bärbar dator. Nackdelen med det första är att testerna skulle köras centralt och möjlighet skulle saknas att påverka vad som exekverades parallellt, vilket skulle riskera att ge avsevärt mycket större störningar på nätvärdena. Valet föll därför på den bärbara datorn, med en 1.6 GHz E-350 AMD Dual-Core processor och ett 4 GB Sodimm DDR3 1066 MHz-minne. Datorn har Windows 7 som operativsystem.

6. Resultat och analys

Resultaten i denna del redovisas med hjälp av diagram, men återfinns även som tabeller i bilaga 5 sist i rapporten.

I de flesta fall har genomsnitt av ett antal körningar valt att användas för att på så sätt få värden som är lättare att jämföra och kunna bortse från enskilda avsevärt lägre och högre värden som självklart har uppstått. Det har också skett ett val att inte räkna på standardavvikelser eller något annat mått för spridningen. Att spridningen skulle vara stor var förväntat, både baserat på att varje version av varje testfall görs på en mer eller mindre unik uppsättning data men också beroende på att vi inte helt kan styra vad datorn testerna kördes på exekverar samtidigt. Då vi inte kan skilja det ena från det andra och inte heller har en bra bild av vad som kan anses vara normen valde vi därför att fokusera på genomsnitten.

6.1. Hashfunktioner

Det uppstod tidigt frågor kring hur de hashfunktioner som funnits fungerade i jämförelse med varandra. Tester av en mindre omfattning utfördes därför på de olika hashfunktionerna.

För `CuckooHashMap` märktes det snabbt att när MAD användes som hashfunktion växte storleken på tabellen ibland katastrofalt snabbt, på den nivån att programmet krashade innan tillfredsställande antal element var inlagd och tätheter så låga som 1 % var inte ovanliga. Det beslutades därför att detta skulle undersökas närmare.

Vid tester av MAD upptäcktes det att två nycklar där hashfunktionen resulterat i samma index fortsätter att få samma index även om hashfunktionen byggs om så länge storleken på tabellen förblir konstant. Att bygga om hashfunktionen hade därmed alltså ingen effekt överhuvudtaget. Det innebar också att de två olika värden som kunde genereras för de olika uppsättningarna värden för `add` och `mul` ändå resulterade i en fortsatt krock. Det vill säga, om två element uppvisade krock i den ena hashfunktionen skulle samma resultat även ges för den andra. De element som kunde genereras som krockade visade sig ha något annat gemensamt. Differensen mellan deras `hashCode` var en multipel av storleken på tabellen. Studie av själva koden gav dock förklaringen.

Förutsatt att dessa tre rader gäller:

$$k_1.\text{hashcode}() = t_1$$

$$k_2.\text{hashcode}() = t_2 = t_1 + X \cdot \text{tabellstorlek}$$

$$H(k_1) = (t_1 \cdot \text{MUL} + \text{ADD}) \ \% \ \text{tabellstorlek}$$

gäller också detta:

$$H(k_2) = (t_2 \cdot \text{MUL} + \text{ADD}) \ \% \ \text{tabellstorlek} =$$

$$= ((t_1 + X \cdot \text{tabellstorlek}) \cdot \text{MUL} + \text{ADD}) \ \% \ \text{tabellstorlek} =$$

$$= (t_1 \cdot \text{MUL} + X \cdot \text{tabellstorlek} \cdot \text{MUL} + \text{ADD}) \ \% \ \text{tabellstorlek} =$$

$$= (t_1 \cdot \text{MUL} + \text{ADD}) \ \% \ \text{tabellstorlek} = H(k_1)$$

vilket innebär att om differensen mellan två nycklars hashcode är en multipel av tabellstorleken så kommer dessa två alltid få identiska värden som den andra oavsett vilka värden MUL och ADD har, vilket gör att en ombyggnad av hashfunktionen inte ger någon som helst effekt.

Detta gör att MAD är olämplig att använda för att generera hashfunktioner för `CuckooHashMap`. Enda alternativet som kvarstod var då för att kunna generera ett flertal hashfunktioner var den pseudoslumpmässiga. För `HopscotchHashMap` var dock vinsten att kunna generera nya hashfunktioner så pass liten att den inte kunde uppväga den extra tid beräkningarna tog jämfört med att bara använda sig av `hashCode`.

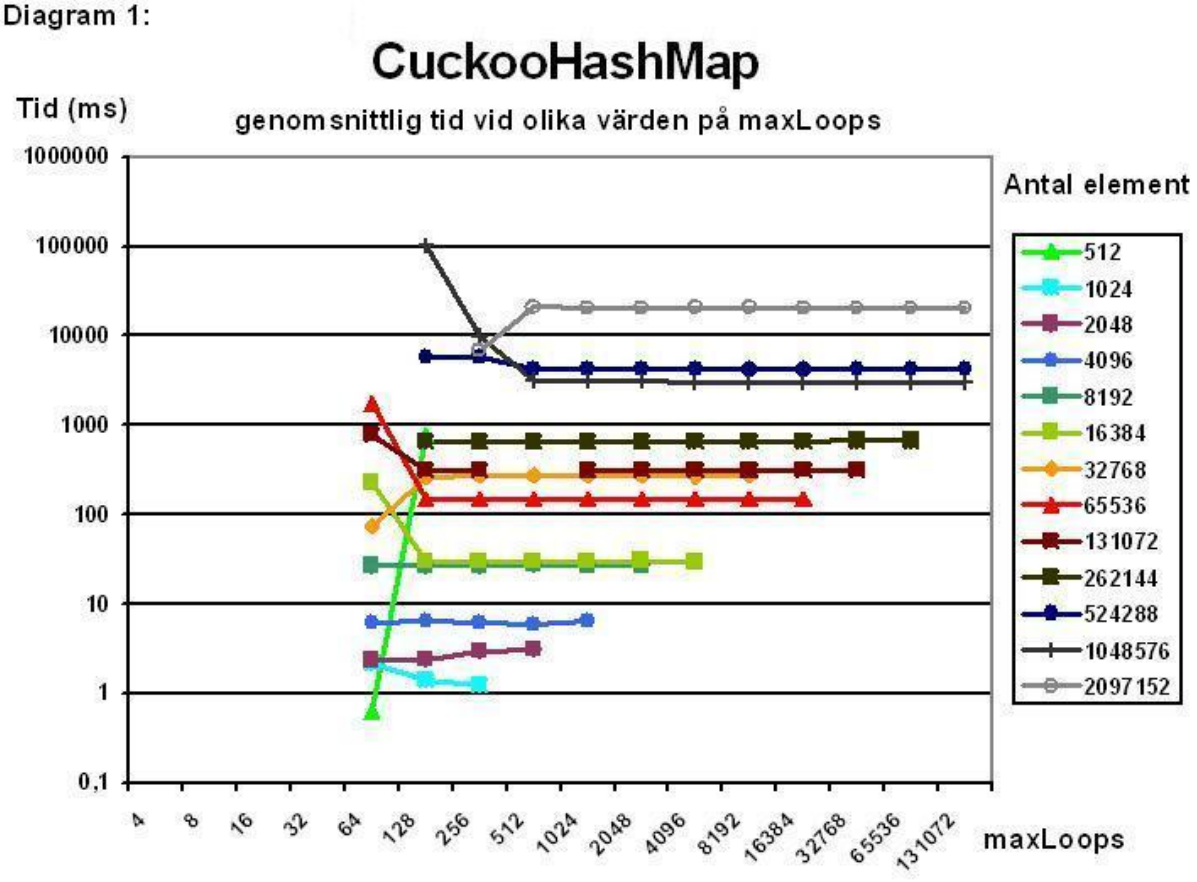
6.2. CuckooHashMap

Elementen som satts in har renderats genom javas `Math.random`()-funktion, vilket gör att det i de fall det funnits dubletter så har datafältet för den befintliga posten uppdaterats. Detta innebär också att den faktiska antalet element i tabellen kan ha varit något lägre än antalet insättningar som genomförts.

6.2.1. maxLoops

I diagram 1 och 2 har vi satt in ett antal element i vår `CuckooHashMap`. Vilka antal det rör sig om kan avläsas i listan till höger i diagrammet. Vi har gjort detta med `maxLoops` satt till respektive värde på x-axeln. Varje insortering har skett 100 gånger, där sedan ett genomsnitt beräknats på tiden som förflutit respektive den slutgiltiga storlek som tabellen har fått. I diagram 1 har vi tiden uttryckt i millisekunder på y-axeln och i diagram 2 har vi storleken på tabellen uttryckt i antal element på y-axeln. loadFactor har här varit 0.5, det vill säga den maximalt möjliga tätheten i tabellen är 50%. maxBounce har varit 5, vilket innebär att om omhashning behövs så kommer det första att ske en ombyggnad av hashfunktionen upp till fem gånger innan storleken dubblas. Observera att tiden som redovisas här är tiden för att lägga in andra halvan av elementenmängden, det vill säga det optimala just för steget från antalet element i mängden innan till den nuvarande.

Diagram 1:

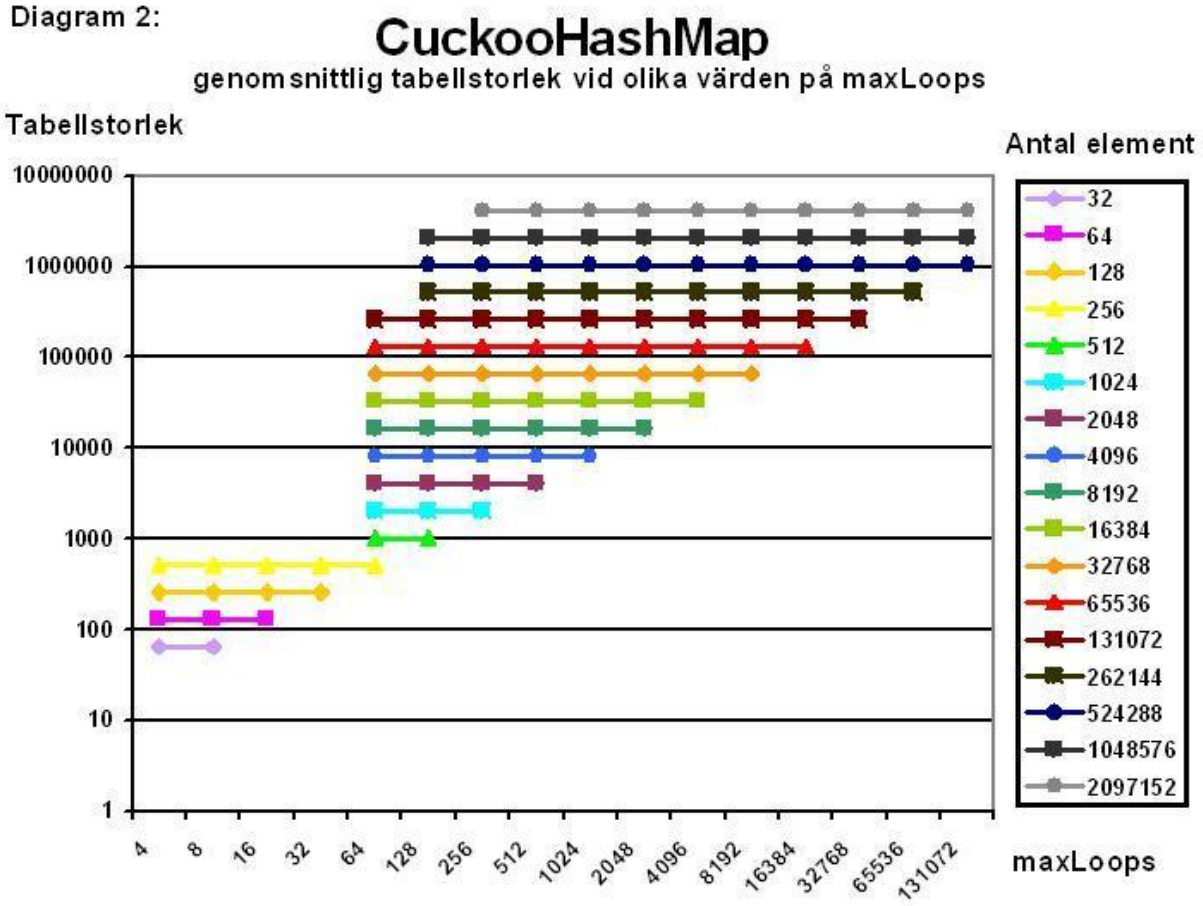


Av diagram 1, där vi testar att lägga in ett fast antal element men varierar `maxLoops`, kan vi se att variabeln inte påverkar hastigheten speciellt mycket. Det viktigaste tycks vara att att den inte är för liten eller för stor, eftersom tabellen då kraschar. Ett rimligt värde för `maxLoops` som fungerade på de flesta storlekar indata kan utläsas vara någonstans mellan 256 och 512. Vid jämförelse med andras implementationer av `CuckooHashMap` tillgängliga på internet, kan man se att det finns två olika dominerande tillvägagångssätt att bestämma `maxLoops`. Dels den statiska som vi använder här[13][14], dels att låta `maxLoops` motsvara tabellens storlek.[15][16] Det stämmer överens med våra resultat och en indikation på att vårt val med statisk `maxLoops` är beprövad.

I en rapport[17] vi läst argumenterar de för att sätta `maxLoop` dynamiskt, de sätter den till $3*log_2 r$, där r är lika med tabellstorleken. Vi utförde själva försök med `maxLoops` satt till $3*log_2 r$, $12*log_2 r$, $15*log_2 r$ och $18*log_2 r$. För vår implementation gav inte detta någon förbättring. Detta eftersom vi sällan kommer upp i max antal iterationer, och ett för lågt antal maximala iterationer tvingar fram väldigt många omhashningar. Dessa tar mycket längre tid än att snurra runt i vår loop och kasta om elementen, då vår insättning är väloptimerad och endast utför två hashningar per insättning tack vare att vi sparat `hashinfo` för varje element. För vår implementation är det alltså viktigast att antalet iterationer är tillräckligt högt för att inte tvinga fram onödiga rehashningar. Resultaten för dessa försök återfinns i tabellerna sist i rapporten.

[13] Monson, L.
[14] TenHarmsel, C
[15] Schwarz, K.
[16] Alvergren, J.
[17] Korwar, A.

Diagram 2:



I diagram 2 är det tydligt att tabellstorleken är oberoende av `maxLoops`, förutsatt att värdet på `maxLoops` är satt inom de värden vi har testat. Detta beror med största sannolikhet på att vi har använt oss av ett relativt högt värde på `maxBounce`, vilket gör att risken är mindre att tabellens storlek ökas när ett element inte går att sättas in. Ett mindre värde på `maxLoops` borde visserligen göra att färre insättningar lyckas, men då omhashning utan storleksökning sker istället för att öka storleken direkt gör detta att den slutgiltiga storleken ändå blir densamma. Hade denna distinktion inte gjorts mellan omhashning på grund av tabellens täthet och omhashning på grund av misslyckade insättning hade resultatet antagligen varit att storleken på tabellen hade minskat när `maxLoops` ökat.

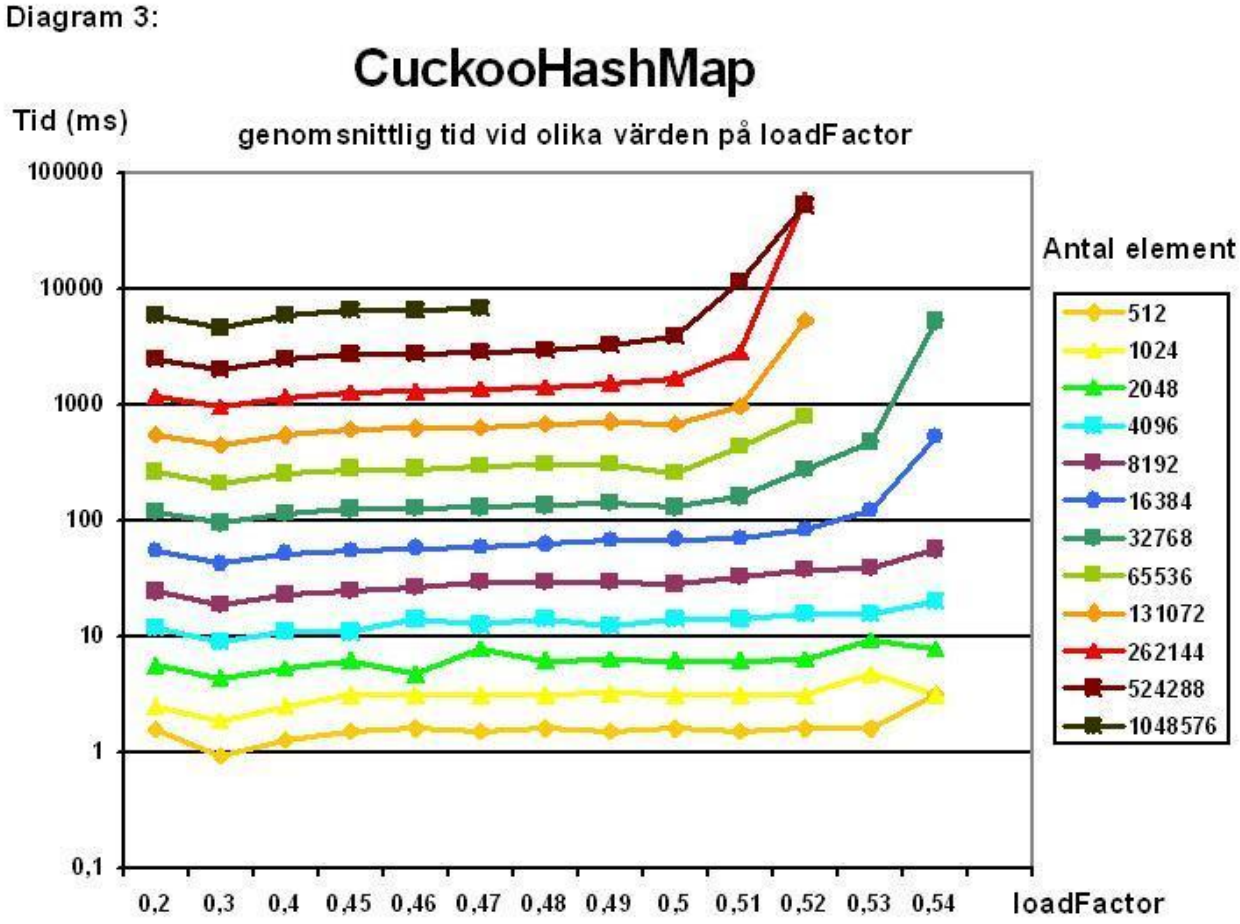
6.2.2. loadFactor

Vidare gjordes också försök på att bestämma vilket värde på `loadFactor` som var optimalt. Även här genomfördes testerna på olika mängder av antal element som sattes in i en `CuckoHashMap`. 100 körningar av varje enskilt testfall gjordes även här, precis som att genomsnitt av storlek och tid beräknades. Här var dock istället `maxLoops` konstant och satt till 300, medan `loadFactor` läts variera. Enligt den litteratur vi tidigare läst låg det optimala värdet för `loadFactor` kring 0.5 vilket gjorde att vi valde att koncentrera oss kring det värdet.[18] Den övre begränsning för `loadFactor` var inte ett medvetet val så pass mycket som att den tid som förflöt för insättning ökade så dramatiskt att det dels var tydligt att det optimala värdet inte skulle gå att finna i den riktningen men också för att den tid som skulle krävas för att genomföra körningen så pass många gånger att vi skulle få ett användbart och relativt säkert resultat. På grund av detta minskar också den största `loadFactorn` vi har testat med för varje mängd element minskat när antalet element har ökat.

[18] Herlihy mfl

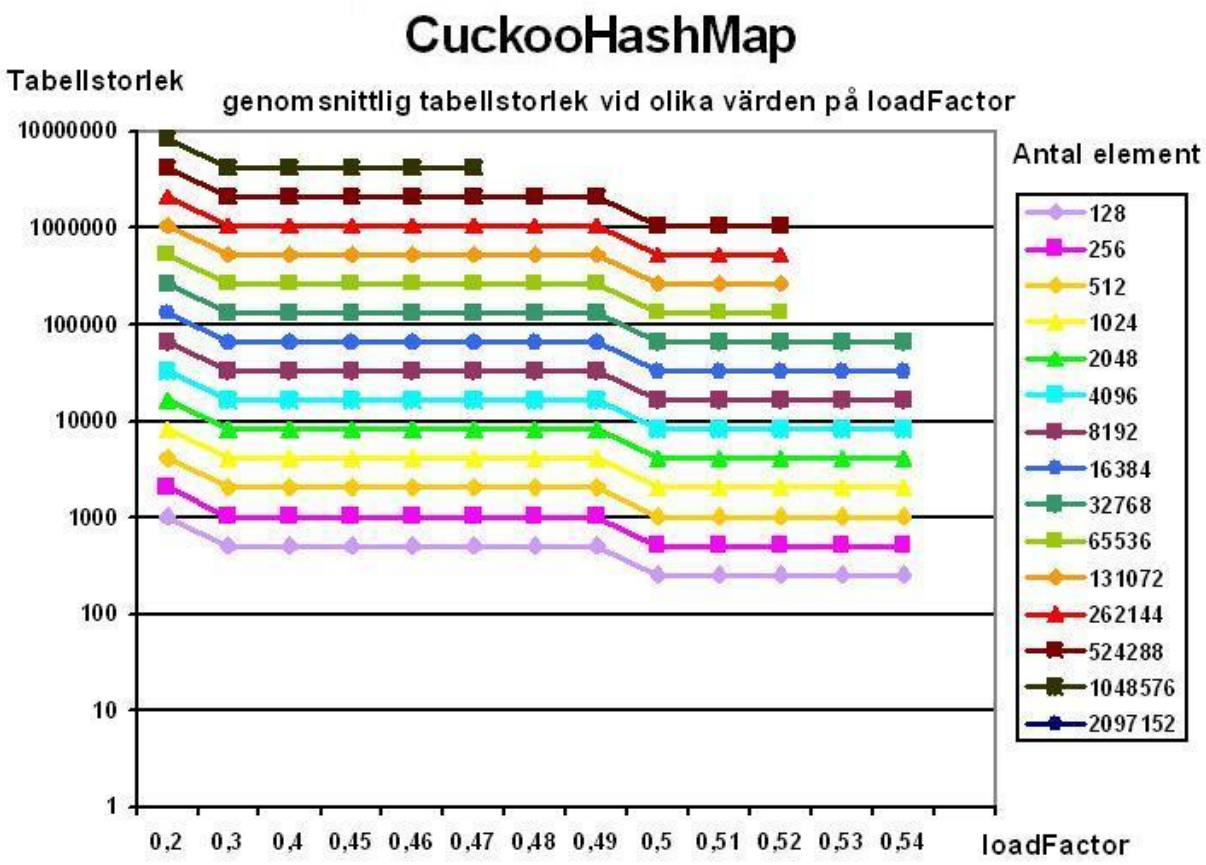
Diagram 3 visar hur tiden som insättningen tar varierar med `loadFactor`, medan diagram 4 visar hur storleken på tabellen varierar med `loadFactor`.

Diagram 3:



Försöken med varierande `loadFactor` som vi ser i diagram 3 visar att nånstans under 50% täthet ger bra prestanda. Fyller vi tabellen mer än så blir insättningar långsamt då tabellen blir för tät och sannolikheten för att fler omhashningar måste göras skjuter i höjden. En glesare tabell än 50% ger en förbättring av tiden för insättningarna, eftersom både färre omhashningar och omflyttningar behöver utföras, då vart element med större sannolikhet kan placeras i endera av sina två positioner direkt. Nackdelen är att tabellen då också tar mer minne, vilket vi ser i diagram 4.

Diagram 4:



Här ser vi att det finns två trösklar där minnesåtgången ökar, dels mellan en täthet av 20% och 30%, och dels en halvering av storleken vid ökning av tätheten från 49% till 50%. Den senare tröskeln är speciellt intressant, då tidsvinsten mellan den är liten, men vinsten i minnesåtgång är stor.

Vi anser därför att en täthet på 50% är det bästa valet och en bra kompromiss mellan de två egenskaperna snabbhet och minnesåtgång. Dessa resultat stämmer även väl överens med det som författarna till artikeln Hopscotch Hashing säger: *"A bigger disadvantage is that Cuckoo hashing tends to perform poorly when the table is more than 50% full because displacement sequences become too long, and the table needs to be resized."*[19]

6.3. HopscotchHashMap

För `HopscotchHashMap` var det två parametrar som ansågs intressanta att undersöka. Dels storleken på grannskapet som är grundläggande för algoritmen, men det uppstod också ett intresse av huruvida en övre gräns på täthet kunde ha positiv effekt på prestandan. Nedan redogörs för resutlaten av test på dessa två parametrar.

Elementen som satts in har även här renderats genom javas `Math.random`()-funktion, vilket har gett samma effekt som tidigare nämnts rörande den faktiska mängden element lagrade i tabellen och uppdatering av befintliga värden.

[19] Herlihy mfl, s. 2

6.3.1. Grannskapsstorlek

En av de viktigaste parametern att bestämma ett värde på för hopscotchhashning är grannskapsstorleken. För att göra detta sattes till en början olika mängder element, 128 till 1 048 576 element, in i `HopscotchHashMap`, med olika storlek på grannskap, från 16 upp till hälften v det antal element som sattes in eller tills den tid som krävdes blev så pass stor att det inte ansågs nödvändigt, med några få undantag för de lägsta antalet element. Värt att minnas är dock att grannskapet som maximalt kan vara halva tabellstorleken, vilket innebär att den vid mindre mängder element ändå är dynamisk. Varje uppsättning av grannskapstorlek och mängd element testades 100 gånger och genomsnittet beräknades för tidsåtgång och slutgiltig tabellstorlek. För att få med den effekt omhashning har så valdes den initiala storleken på varje hashtabell till 2.

Diagram 5:

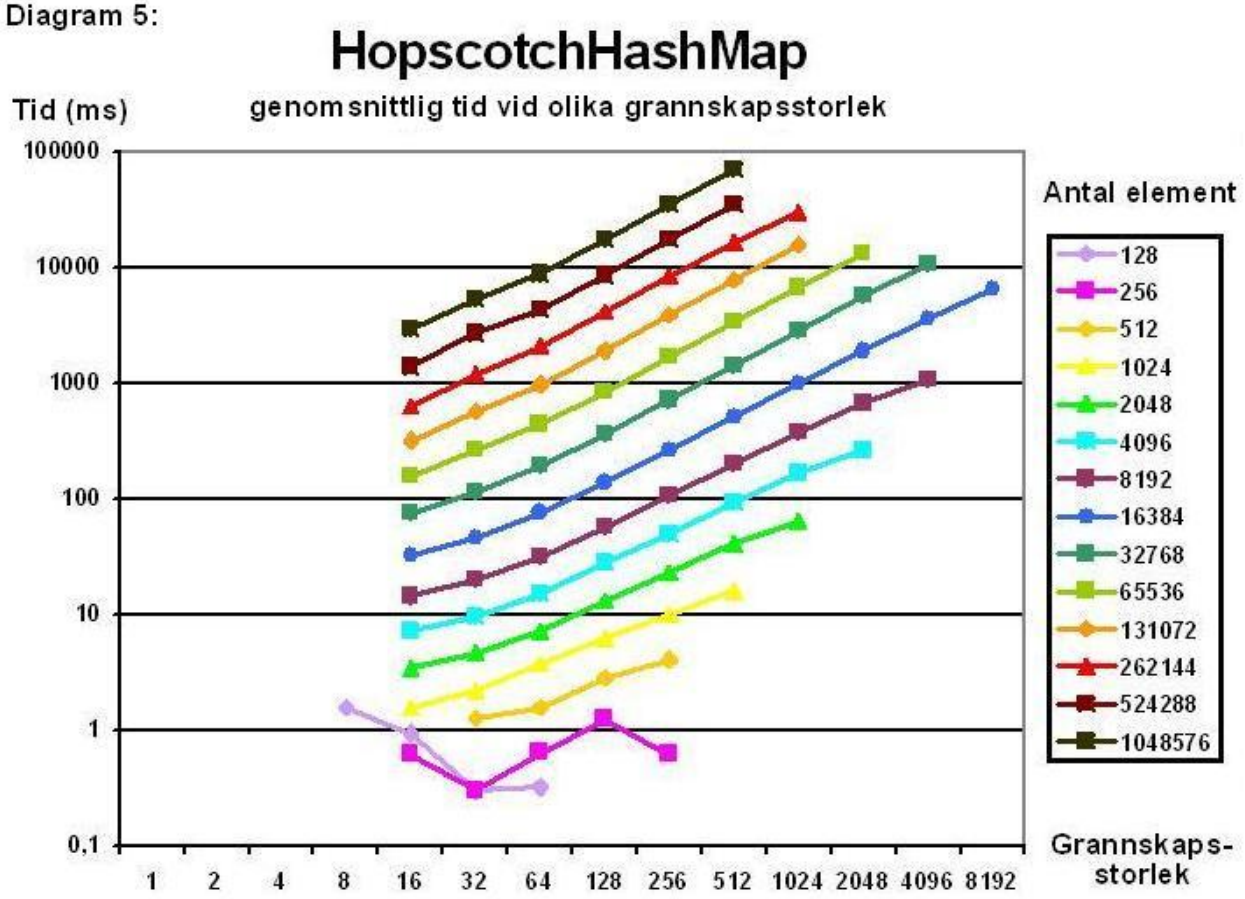
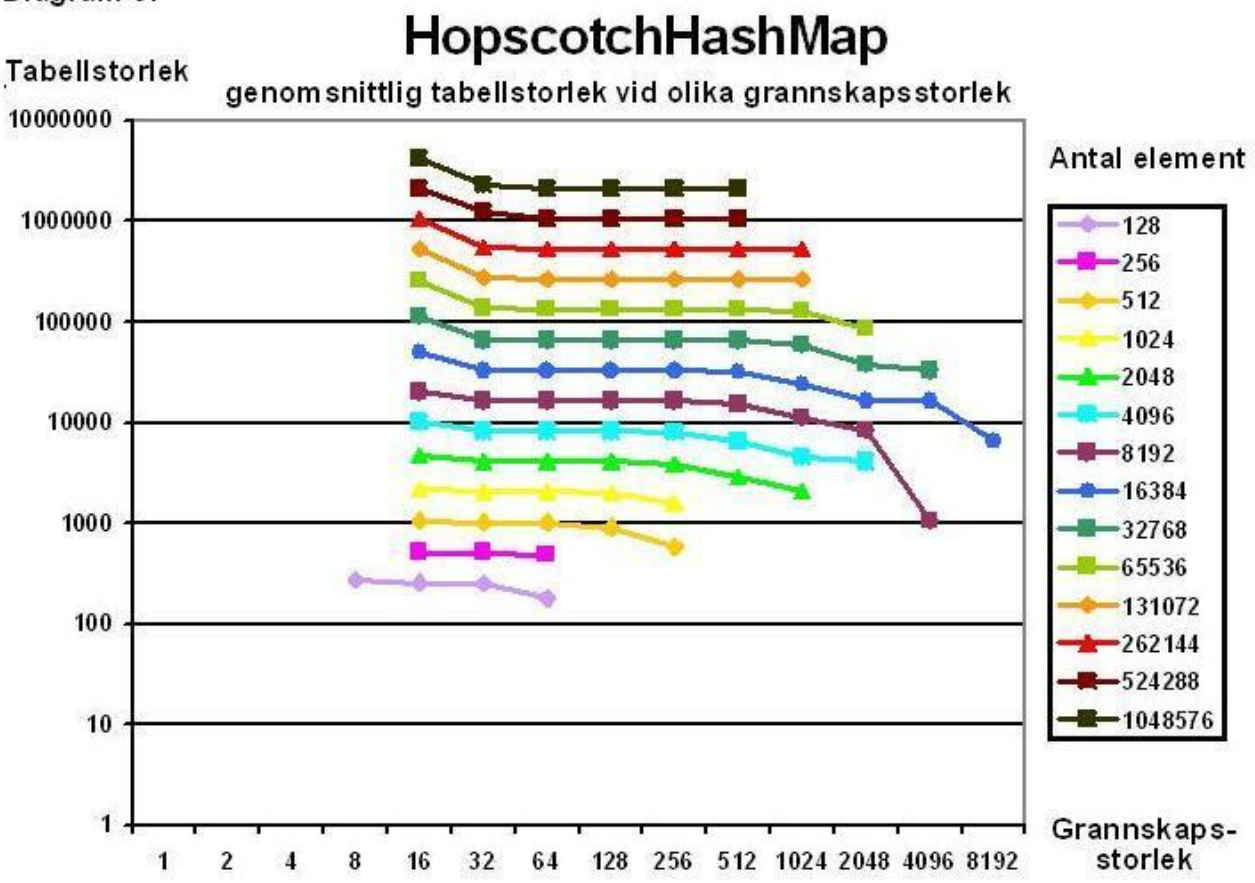


Diagram 5 visar klart och tydligt att ju mindre grannskapsstorlek, desto snabbare går insättningen. Då samma sak även gäller operationer för att hitta och radera element är det klart och tydligt att ur ett tidsperspektiv bör fokus vara på att hålla grannskapsstorlek så lågt som möjligt, oavsett tabellstorlek eller antalet element. Det faktum att resultatet redan här är så pass entydigt gör att test med kombinationer av insättningar och andra operationer inte behövs.

Diagram 6:



Tar man även diagram 6 i beaktning, som visar tabellens storlek vid olika grannskapsstorlekar, tycks storlek 32 på grannskapet vara en bra utgångspunkt då det ger en lagom kompromiss mellan snabbhet och minnesåtgång. Detta resultat stämmer överens med det som författarna till rapporten Hopscotch Hashing valt som grannskapsstorlek i sin implementation.[20]

6.3.2. loadFactor

Då en övre gräns för tätheten gav klar effekt för `CuckooHashMap` ansågs det vara värt att undersöka om motsvarande även gällde för `HopscotchHashMap`. Det undersöktes på liknande sätt där vi gjort 100 körningar där vi satt in olika stora mängder av element, från 512 element till 1 048 576 element, i tabellen och beräknat genomsnittet på tabellens storlek och tiden insättningarna tagit för att få ett värde att jämföra med. `loadFactor` anges precis som för `CuckooHashMap` som ett decimaltal mellan 0 och 1, inklusive 1, där detta utgör den maximala tätheten i tabellen. När mängden element i tabellen uppnår den nivån sker en automatisk omhashning, som innebär att tabellstorleken dubblas och alla element sätts in på nytt. Då ingen referens påträffats för vilket värde som skulle vara optimalt på denna `loadFactor` valdes en lägre gräns på 0.30 strikt ur perspektivet att lägre tal skulle riskera att öka minnesåtgången drastiskt och en högre gräns på 1, då detta är det största möjliga värde och detta då skulle motsvara att göra insättningen utan att egentligen använda sig av `loadFactor`, mer än att beräkningarna och uppdateringarna som görs i rehash ändå skulle genomföras. Samma sak gäller också jämförelsen i `put` som då också ändå genomförs. Test gjordes med steget 0.05 för att få klara beräkningspunkter. Hade några värden på `loadFactor` visat på speciellt intressanta resultat

[20] Herlihy mfl, s. 2

skulle vi då kunna fokusera extra på dessa för att få en tydligare bild just där.

Enligt resultaten kring grannskapsstorlek tydde på att 32 var optimal storlek var det också det värde på den parameter som användes här. Även i det här fallt användes en initial tabellstorlek på 2 av samma skäl som tidigare.

Diagram 7:

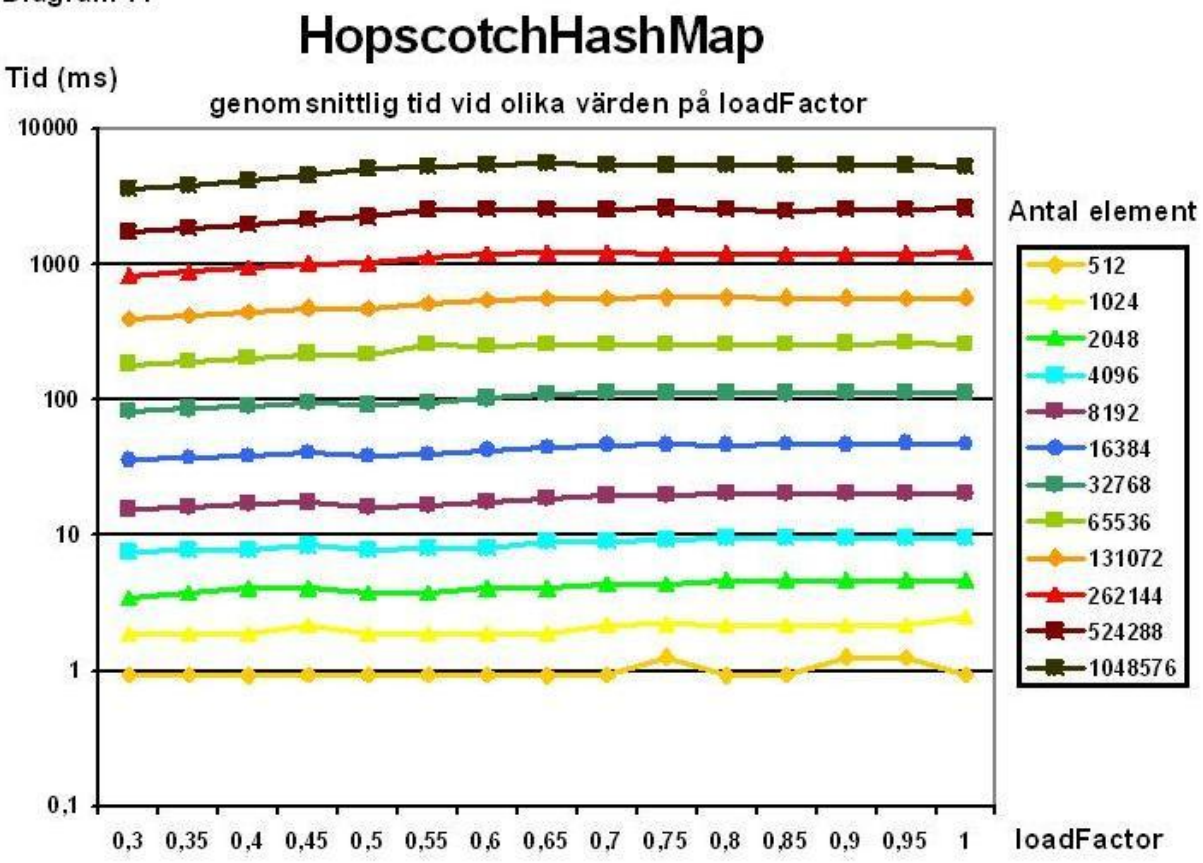


Diagram 7 visar att `HopscotchHashMap` presterar väl under alla former av maximal täthet i tabellen, till och med vid så hög täthet som över 90%. Detta är överrensstämmande med vad annan litteratur kommit fram till.[21][22] Detta även om det finns en klar tendens att tiden för insättning i hashtabellen tar mindre tid vid en låg maximal täthet såsom 30-35%, så är skillnaden inte stor. Från 65% till 100%, där 100% innebär att omhashning med fördubblande av storlek aldrig sker på grund av hur många element som finns lagrade i förhållande till storleken på tabellen, är det knappast någon utläsbar skillnad alls. Jämför man värdena för 50% och 100% är skillnaden knappt synbar, även om det kan vara värt att påpeka att den logaritmiska skalan gör att det samtidigt är lätt att underskatta skillnaderna.

[21] Herlihy mfl, s.6

[22] Hariesh mfl, s.6

Diagram 8:

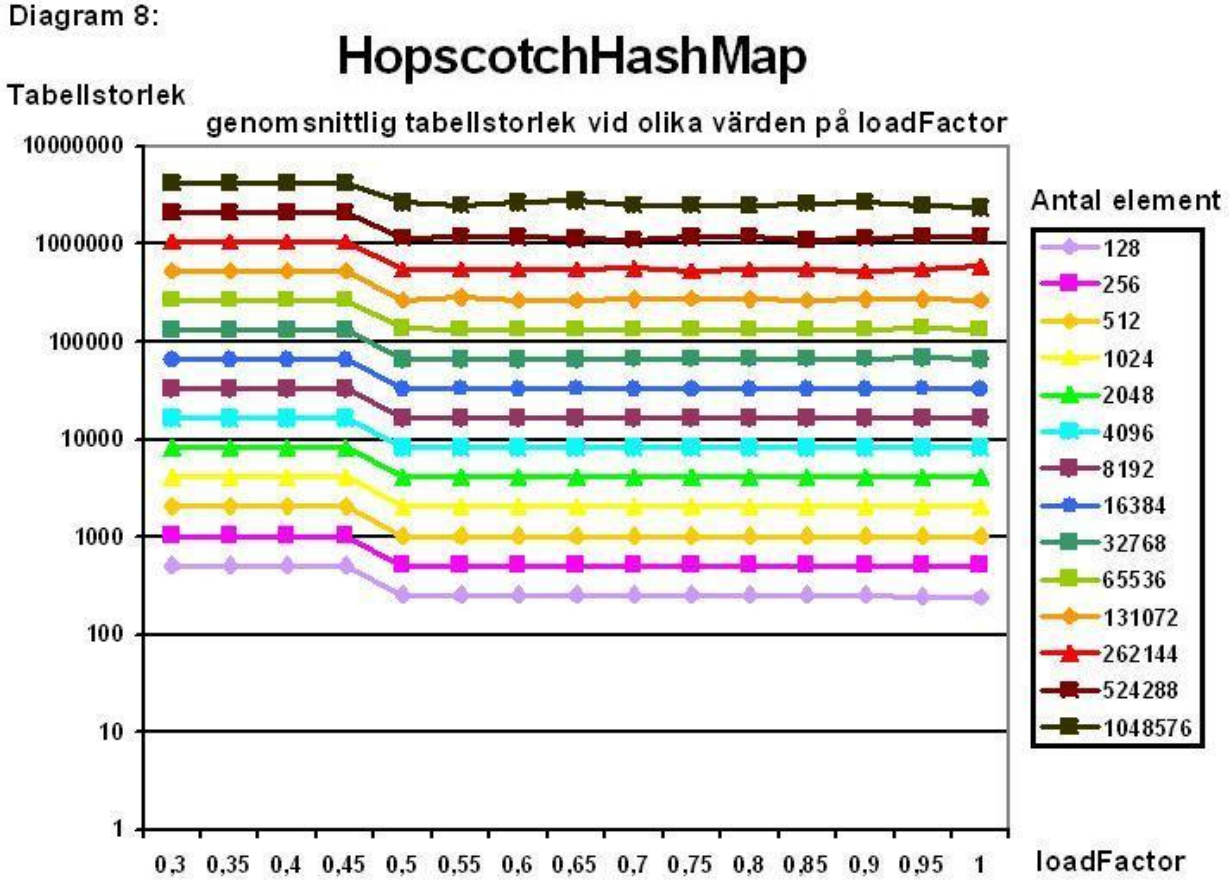


Diagram 8 visar att storleken på tabellen klart och tydligt påverkas av `loadFactor`. På steget från 45% till 50% halveras den genomsnittliga storleken, vilket definitivt är värt att notera. Det tyder på att det ur minnesutrymmesperspektiv är värt att överväga att låta `loadFactor` vara över 50%, för att det inte ska påverka för pass mycket. Problemet är att det där inte ger särskilt stor positiv påverkan alls, vilket gör att det kan vara idé att ifrågasätta behovet av att använda `loadFactor` över huvud taget. Om effekten är så pass liten finns risken att jämförelser vid varje insättning och uppdateringar och beräkningar av parametrar i rehash tar ut den lilla effekt som eventuellt uppstår. I de rapporter vi läst om hopscotchhashning tycks de inte ha implementerat någon `loadFactor` heller.[23][24] Då vi inte kunnat tagit del av deras källkod kan vi inte säga helt säkert att de inte implementerat det, men de nämner inte det i texterna.

Baserat på resultaten redovisade ovan och kring den data vi funnit har vi valt att låta grannskapsstorleken i vår slutgiltiga `HopscotchHashMap` vara 32 som grundläggande inställning, med möjlighet för användaren att ställa in den även på andra värden om det skulle vara aktuellt. Detta för att underlätta användning vid tillfällen där mindre eller mer tyngd läggs på minnes- och tidsåtgång än vad vi har valt att göra. Resultatet kring att använda sig av en `loadFactor` för att ha en övre gräns på tätheten för att potentiellt underlätta insättning har gjort att vi inte ser någon nytta tillräckligt stor för att väga upp det beräkningsarbete en sådan parameter skulle innebära. Vi har därför valt att inte använda oss av en sådan i vår slutgiltiga version.

6.4. Jämförelse

För att få en referensram för `CuckooHashMap`s och `HopscotchHashMaps` prestationsförmåga har jämförelser gjorts med andra hashtabeller som mappar nycklar och data. Som tidigare nämnts

[23] Herlihy mfl

[24] Askitis, N.

valdes javas egna `HashMap` och javas egna `ConcurrentHashMap`, där den senare valdes för att också få med potentialen som parallellisering ger. Redogörelser för dessa finns i det tidigare avsnittet “Om parallellisering av algoritmer”. Då dessa är standardmotsvarigheterna i java borde de vara ordentligt optimerade och fungera väl. De bygger dessutom på en annan hashningsalgoritm än de som har skrivits här vilket gör att det också blir en jämförelse mellan algoritmer i sig.

Alla hashtabeller har haft en initial storlek på 16 element, vilket är det som sker i javas versioner om man inte specificerar något annat. `HashMap` och `ConcurrentHashMap` har båda haft 75% inställt som maximal täthet, vilket är vad den tillhörande dokumentationen rekommenderar. `CuckooHashMap` har haft en maximal täthet på 50%, `maxLoops` satt till 300 och en `maxBounce` på 5, vilket är de inställningar baserat på tidigare resultat vi skulle rekommendera. `HopscotchHashMap` har haft en grannskapsstorlek på 32, men saknat en gräns för maximal täthet då denna valts bort.

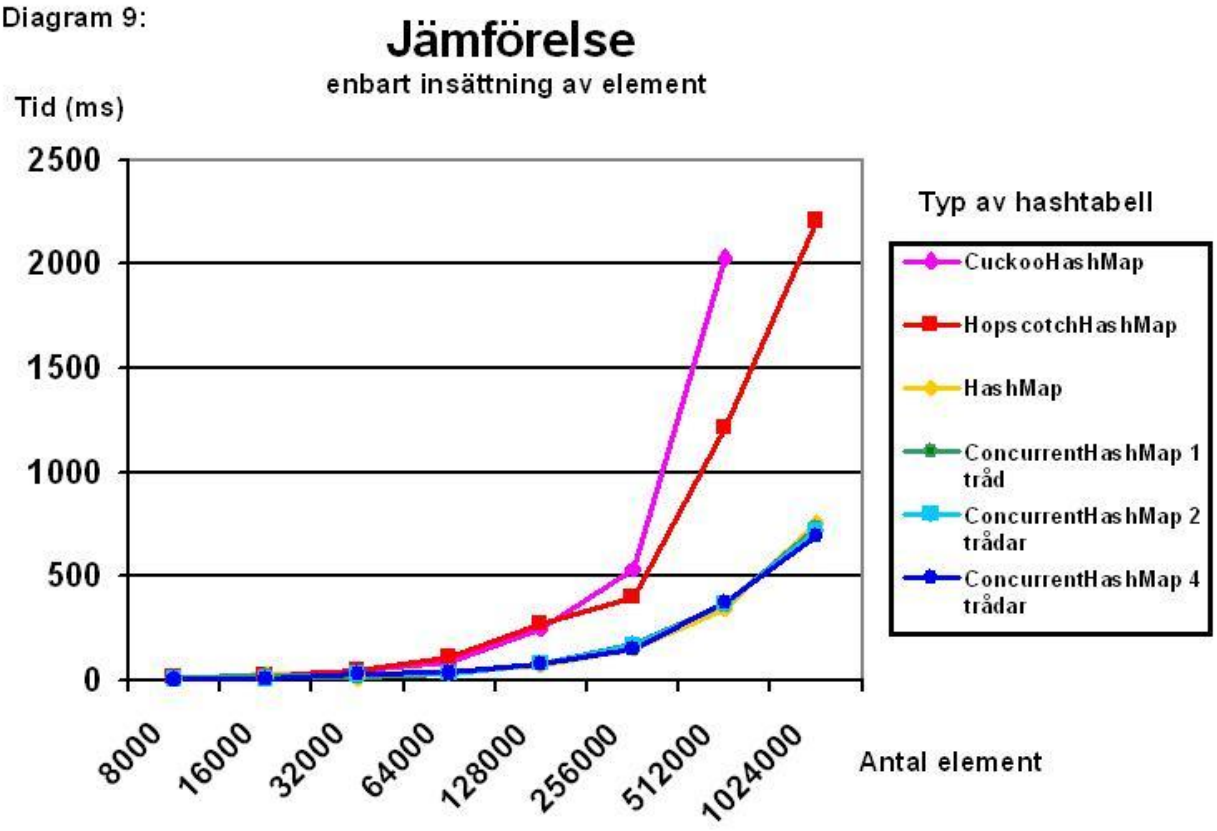
Test har både gjorts för insättningar, som är de mer tidskrävande operationerna, och på `containsKey`, som anses representativ för övriga operationer såsom `delete` och `get` och som tillhör den snabbare kategorin. Tyvärr saknas det möjlighet att få tillgång till tabellstorleken för hashtabellerna skapade genom javas klasser vilket gör att en storleksjämförelse inte kan genomföras, något som annars hade önskats.

Självklart finns det en risk för att inställningarna för hashtabellerna inte är de optimala för de testfall som valts ut, och någon med ett tydligt fokus på prestation skulle självklart mycket väl kunnat valt andra värden. Då vi inte har kunnat jämföra storlekarna på tabellerna, bara mängden element insatta, vilket har gjort att fokus ligger på tid hade vi kunnat optimera för tidsaspekten, men lagringsutrymme är fortfarande viktigt och högst hastighet är irrelevant om tabellen blir så pass stor att den inte kan lagras.

6.4.1. Insättning

I det första testet genomfördes enbart insättning. För varje algoritm sattes 8000, 16000, 32 000, 64 000, 128 000, 256 000 och 512 00 element, där varje insättning skedde 50 gånger. 512 000 element testades dock inte för `CuckooHashMap` då detta tog avsevärt mycket längre tid än övriga och värdet för 256 000 element ansågs tillräckligt för att kunna analysera datan tillfredsställande. Ett genomsnitt på den tid som tagit beräknades och det är dessa värden som syns i diagram 9 och diagram 10 nedan. `ConcurrentHashMap` testades med en, två respektive fyra trådar för att se hur pass stor påverkan detta skulle ge på tidsåtgången.

Diagram 9:



I diagram 9 visar en jämförelse mellan den genomsnittliga tid insättningarna tagit med en linjär skala på y-axeln för tiden. Det är rätt tydligt att javas båda versioner av `HashMap`, både den parallelliserade och icke-parallelliserade är avsevärt snabbare än de två som skapats i det här arbetet, och att skillnaden i tidsåtgång mellan parallelliserad och icke-parallelliserad, oavsett mängden trådar är relativt liten. Det innebär att man förlorar väldigt lite på att använda sig av en parallelliserad version av `HashMap`, samtidigt som man kan tjäna mycket på möjligheten att flera trådar kan använda sig av hashtabellen samtidigt. På större mängder element tar `CuckooHashMap` väldigt mycket mer tid på sig, och även om `HopscotchHashMap` också ligger långt ifrån `HashMaps` tider presterar `HopscotchHashMap` ändå klart bättre än `CuckooHashMap` på större mängder element.

Diagram 10:

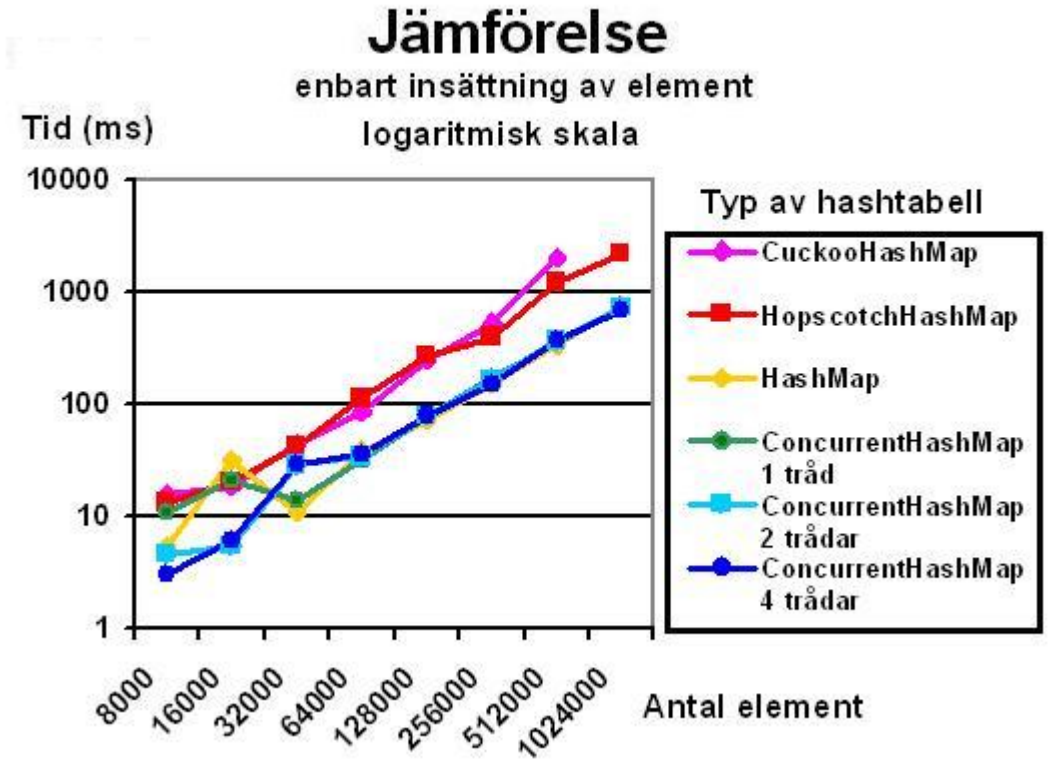
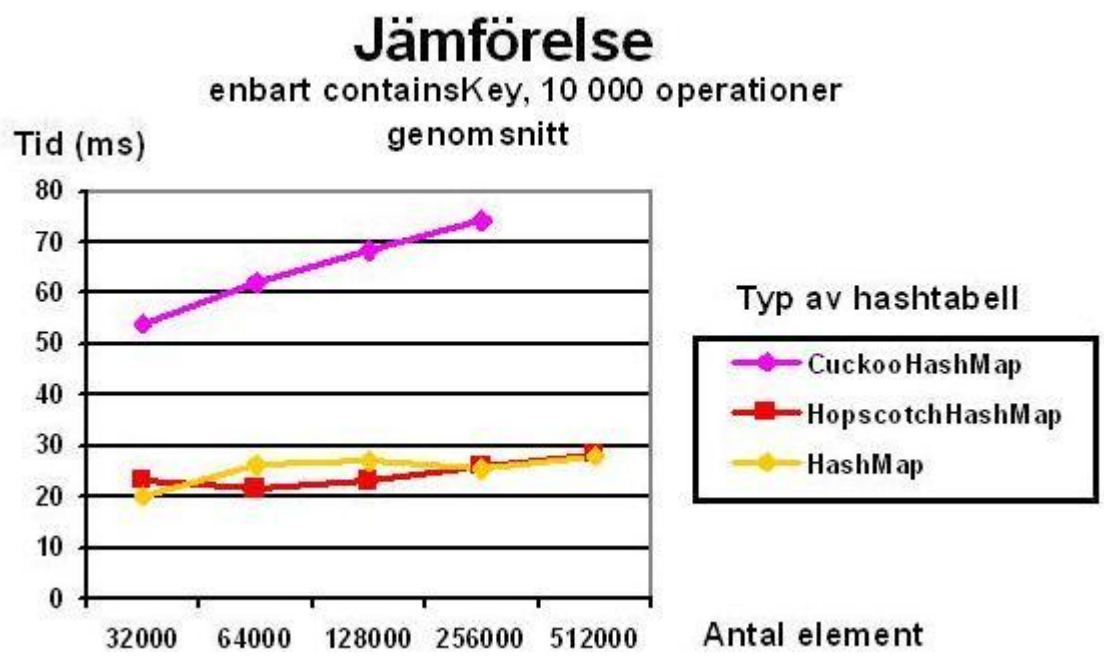


Diagram 10 visar samma resultat som diagram 9, men med en logaritimisk skala för tiden. Då även x-axeln är logaritmisk innebär det att det nästintill linjära beteende som hashtabellerna här med denna skala uppvisar för insättning av de olika mängderna element borde kunna översättas till att insättningen har en tidskomplexitet på O(n).

6.4.2. containsKey

Nästa jämförelse fas var anrop på `containsKey`. Här lades 32 000, 64 000, 128 000, 256 000 och 512 000 element in i `CuckooHashMap` (dock ej högsta värdet här av samma anledning som tidigare redogjordes för), `HopscotchHashMap` och `HashMap` med samma inställningar som tidigare. När insättningen var klar och tabellen faktiskt innehöll ovan nämnda antal element gjordes 10 000 `containsKey`-anrop med slumpade heltal som nycklar, där tiden för alla alla dessa anrop att exekveras mättes. Detta genomfördes 100 gånger för varje hashtabell och varje antal element för diagram 11 där genomsnittet för tiden beräknades.

Diagram 11:

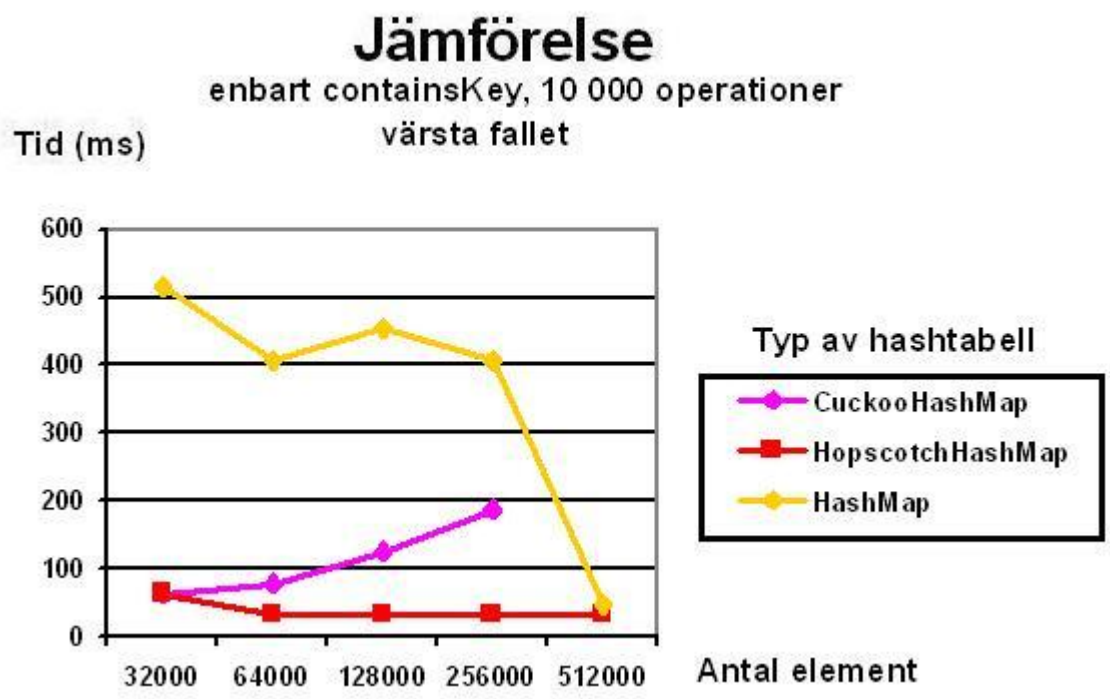


I diagram 11 visar på att `HopscotchHashMap` och `HashMap` är klart jämförbara när det kommer till den tid `containsKey` och motsvarande operationer tar. Det visar också att denna tid, när man tittar på genomsnittet, är i stort sett konstant, och verkar inte variera tydligt med mängden element. De skillnader som finns är relativt små och går åt båda håll. `CuckooHashMap` tar avsevärt mycket längre tid än de båda andra vilket är delvis oväntat då det bara är maximalt två index i tabellen som behöver kontrolleras, medan antalet platser som maximalt kan behöva undersökas i de två andra är avsevärt fler. En möjlig förklaring till detta är den hashfunktion som `CuckooHashMap` använder sig av. De två övriga använder sig av `hashCode`, vilket kan förväntas ta betydligt mindre tid beräkningsmässigt än `CuckooHashMaps` variant som använder sig av en slumpgenerator.

Hashfunktionen kan också förklara varför tiden ökar med mängden element för `CuckooHashMap`. För varje omhashning som sker utan att storleken ökar den tid beräkningen hashfunktionen genomför då ytterligare ett `nextRandom`-anrop måste genomföras. Just större storlek tabellen har desto fler element kan sättas in innan en storleksändring kommer att ske, vilket gör att tiden för exekvering av anrop till hashfunktionen också kommer att öka.

Med en bättre hashfunktion hade antagligen värdena för `CuckooHashMap` varit bättre.

Diagram 12:



En påstådd fördel med både hopscotchhashning och cuckoohashning rör dock inte den genomsnittliga tiden för `containsKey`-operationer utan att värstafallet aldrig ska kunna bli dramatiskt mycket större än genomsnittet. Problemet är att det är svårt att testa just värstafallet. Ett anrop till `containsKey` går för snabbt att exekvera för alla hashtabeller för att det ska gå att mäta tiden.

Vad som istället gjorts är liknande som för diagram 11, men istället för att bara genomföra det hela 100 gånger och ta genomsnittet har insättningen körts 500 gånger, varje gång med en ny tabellen och den längsta tiden för de 10 000 operationerna har sparats. Då detta ändå handlar om en kombination av 10 000 så visar det inte på det värsta fallet i sig nödvändigtvis, men ger ändå ett perspektiv på variationen i hur pass mycket den tid operationerna tar baserat på tabellen.

I diagram 12 är det tydligt att `HashMap` har avsevärt högre värden än `CuckooHashMap` eller `HopscotchHashMap`, bortsett från för 512 000 element. Den stora skillnaden skulle visserligen teoretiskt kunna bero på andra faktorer kopplade till datorn mer än koden, men i så fall borde rimligtvis även `CuckooHashMap` och `HopscotchHashMap` ha påverkats. De två senare har visserligen några värden som är en bit högre men fortfarande inte i närheten av det beteende som `HashMap` uppvisar. Det är också mycket möjligt att `HashMap` hade gett ett högre värstavärde även för 512 000 element ifall fler körningar genomförts.

Det senaste tyder dock på att dessa situationer inte uppträder ofta, åtminstone inte med slumpmässig data. Problemet är att i verkligheten så är datan sällan slumpmässig och sneda fördelningar kan uppstå. I sådana fall visar `HopscotchHashMap` och `CuckooHashMap` på klara fördelar jämfört med `HashMap` om det främst är `containsKey`, `delete` och liknande som kommer att utföras. Även om insättningen kommer ta längre tid kan det fortfarande i ett sådant läge vara värt att överväga att använda en annan datastruktur än `HashMap`.

7. Slutsats och diskussion

7.1. CuckooHashMap och HopscotchHashMap

Sammanfattningsvis kan man konstatera att för vår implementation av cuckoohashning visade det sig vara bäst att låta `maxLoops` vara statisk och ligga någonstans mellan 256 och 512 - i vår slutgiltiga version av `CuckooHashMap` låter vi därför standardvärdet för maxLoops vara 300. Det finns ingen större vinst med att finjustera denna parameter så vi låter den därför vara hårdkodad. För `loadFactor` är 0.5 en rimlig avvägning, även om användaren bör ha möjlighet att ställa in den beroende på vad denne prioriterar. För implementationen av hopscotchhashing så har vi kommit fram till att låta standardvärdet för grannskapsstorleken vara hårdkodat till 32. `loadFactor` kommer inte att användas i den slutgiltiga versionen då den ej ger tillräckligt stark effekt för att motivera extra beräkningar vid de värden som är rimliga att låta parametern anta. Parametern för den initiala storleken används av både `HopscotchHashMap` och `CuckooHashMap` och kommer ha ett värde på 16, samma som hashtabellerna vi undersökt i javas standardbibliotek, såvida inte användaren specificerar något annat. Detta då utifall vetskap finns om den slutgiltiga mängden element så kan man genom att specificera en större initial storlek undvika ett antal tidskrävande omhashningar.

Det kan också konstateras att alla de slutsatser som dragits här angående parametrar också styrks av litteratur vi funnit.

Det finns dock alltid ett dilemma i att prioritera mellan utrymme och tid i sådana här frågor och den balans som här valts delas inte nödvändigtvis av alla. De faktum att vissa delar av vår implementation, såsom omflyttningar av element i `CuckooHashMap` och `HopscotchHashMap`, är väldigt effektiva ur beräkningssynpunkt och på så sätt oerhört optimerade samtidigt som andra, såsom omhashningsmetoderna, är betydligt mer tidskrävande gör att de optimala värdena på de olika parametrarna kan förskjutas i jämförelse med en annan implementation av samma algoritm. Skulle omhashningen kunna effektiviseras ytterligare skulle dessa parametrar i vissa fall mycket väl kunna anta andra värden. Vi misstänker att `maxLoops` är en sådan parameter.

Detta visar också vitsen på att optimera koden i sig, och inte bara parametrarna. Det finns mycket små saker som kan göras och som i vissa fall är kopplade till programmeringsspråket som används. Detta är säkerligen minst lika viktigt som parameterval och bör definitivt prioriteras. Tyvärr är kodoptimering betydligt svårare att visa och kontrollera och kräver stora kunskaper i programmeringsspråket ifråga. Även om vi som skrivit klasserna anser oss ha klart dugliga kunskaper i java är vi tyvärr inte experter och det kan mycket väl finnas saker som vi ytterligare kunnat göra.

7.2. Val av hashtabell

Våra jämförelser med Javas standardbibliotek visar på att våra implementationer tar avsevärt mycket längre tid på sig för insättningar än de från javas standardbibliotek som jämförelserna har gjorts med. Samma sak gäller också genomsnittlig tid för att slå upp element. Det kan i sig tyckas att varken `Cuckoo-` eller `HopscotchHashMap` egentligen har något faktiskt användningsområde men försöket till studie av värstafallet visar att det finns en klar fördel med dessa just när det kommer till värstafallet, vilket också stödjs av den teori som finns bakom de två algoritmerna.

Valet av hashtabell bör därmed dels bero på vilken typ av data man förväntar sig, om man alls vet något om den, och dels vilken typ av operationer som främst kommer utföras. Om det finns kunskap om att de nycklar som kommer användas kommer ge en jämn fördelning i hashtabellen finns det ingen anledning att undvika `HashMap` eller dess parallelliserade motsvarighet. Finns det däremot risk, vilket det enligt vår erfarenhet finns, att nycklarna inte kommer vara jämt fördelade är det värt att överväga att använda sig av hopscotchhashning eller cuckoohashning. Samma sak gäller om mängden uppslagningar i tabellen dominerar över mängden insättningar samtidigt som det är viktigt att tiden för uppslagningar inte riskerar att bli för lång.

I det här fallet vinner `HopscotchHashMap` över `CuckooHashMap` på varje område, men det är med största sannolikhet kopplat till hashfunktionen som `CuckooHashMap` använder sig av. Med en snabbare hashfunktion bör skillnaden bli betydligt mindre och `CuckooHashMap` har till och med potential att kunna genomföra uppslagningar snabbare än `HopscotchHashMap`. Det gör att vi även om vi i nuläget rekommenderar användning av `HopscotchHashMap` före `CuckooHashMap` inte helt anser behovet av cuckoohashning icke-existerande. Vid avsevärt mycket större fokus på uppslagning än insättning bör en effektivare variant av `CuckooHashMap` vara det mest optimala. Detta trots att `HopscotchHashMap` hävdar sig ha "*tagit det bästa från cuckoohashning".*

Då en parallelliserad version av hopscotchhashning finns tillgänglig innebär det att det även kan vara värt att överväga att använda sig av denna istället för `ConcurrentHashMap` om det är viktigt att begränsa den tid det tar för en uppslagning.

7.3. Tillförlitlighet och generaliserbarhet

Självklart finns det mängder av potentiella felkällor i det här arbetet. Förutom de mer uppenbara kring körningarna och det faktum att vi utgått från genomsnitt utan vidare analys av spridningen så finns det också en rad andra mindre uppenbara. Det är relativt stor sannolikhet att de personer som har skrivit hashtabellerna från javas standardbibliotek som vi jämfört med har större erfarenhet och kunskaper än vad vi har haft vilket å andra sidan snarast tyder på att det finns en potential för förbättring för värdena för de klasser vi har skrivit. Att inte fler olika hashfunktioner har studerats är en tydlig brist, vilket klart kan ha påverkat skillnaden mellan prestandan hos `CuckooHashMap` i jämförelse med de övriga. Vid möjlighet till fördjuping skulle det senaste vara klart intressant att studera. Det finns också studier som tyder på att cuckoohashning kan prestera betydligt bättre med fler än bara två hashfunktioner,[25] något som också skulle vara intressant att implementera och jämföra.

Vi anser ändå att det går att dra en mer generell slutsats. Hashtabeller är i sig en bra datastruktur på många sätt men den datan som kommer lagras och de operationer som kommer behöva exekveras kan göra att prestandan man slutligen får kan variera oerhört. Man bör därför göra ett medvetet val om vilken typ av hashtabell man använder.

[25] Askitis, N., s.1

8. Referenser

Acantara m.fl. Real-Time Parallell Hashing on the GPU. http://www.idav.ucdavis.edu/func/return_pdf?pub_id=973 (Hämtad 2012-04-12)

Alvergren, J. *CuckooHashMap.java*. (2010) https://github.com/joacima/Cuckoo-hash-map/blob/master/CuckooHashMap.java (Hämtad 2012-04-12)

Askitis, N. *Fast and Compact Hash Tables for Integer Keys.* RMIT Uniersity, Melbourne.http://crpit.com/confpapers/CRPITV91Askitis.pdf (Hämtad 2012-04-12)

Goetz, B. *Java theory and practice: Hashing it out,* http://www.ibm.com/developerworks/java/library/j-jtp05273/index.html (Hämtad 2012-04-12)

Hariesh, Monteiro, och Pandiyan. *Hopscotch Hashing.* New York University, New York. http://cs.nyu.edu/~lerner/spring11/proj_hopscotch.pdf (Hämtad 2012-03-25)

Herlihy, Shavit, och Tzarfir. Hopscotch Hashing. Tel Aviv University och Brown University. http://people.csail.mit.edu/shanir/publications/disc2008_submission_98.pdf (Hämtad 2012-03-20)

Java2s. Integer: MAX, MIN VALUE http://www.java2s.com/Tutorial/Java/0040__Data-Type/IntegerMAXMINVALUE.htm (Hämtad 2012-04-12)

Korwar, A. *Cuckoo Hashing.* (2010) http://www.cse.iitk.ac.in/users/arpk/CuckooHashing.pdf (Hämtad 2012-04-11)

Michelangelo Grigni. *CuckooHash* http://www.mathcs.emory.edu/~cs323000/share/0217/CuckooHash.java (Hämtad 2012-04-12)

Mitzenmacher, M. Some Open Questions Related to Cuckoo Hashing. Harvard University, Cambridge. http://www.eecs.harvard.edu/~michaelm/postscripts/esa2009.pdf (Hämtad 2012-04-12)

Monson, L. *Cuckoo*. http://lmonson.com/downloads/javahash/src.zip (Hämtad 2012-04-12)

Oracle. *HashMap.* http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html#put(K, V) (Hämtad 2012-04-10)

Pagh, R. *Cuckoo Hashing for Undergraduates.* (2006) IT University of Copenhagen, Köpenhamn. http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf (Hämtad 2012-04-12)

Pagh och Rodler F. *Cuckoo Hashing*. AArhus University, Århus. http://www.it-c.dk/people/pagh/papers/cuckoo-jour.pdf (Hämtad 2012-04-12)

Schwarz, K. *CuckooHashMap.* http://www.keithschwarz.com/interesting/code/cuckoo-hashmap/CuckooHashMap.java.html (Hämtad 2012-04-12)

TenHarmsel, C. *CuckooHash.* http://www.staticmethod.net/wp-content/uploads/2009/05/cuckoo_hash.rb (Hämtad 2012-04-12)

9. Bilagor

Bilaga 1: CuckooHashMap

```
/**
 * This is a class for a hashmap built on the cuckoo hashing algorithm. It maps
 * a K key to some Data D. It can take any initial size value and is not
 * limitied to a size which is the power of two. It uses the class Element<K, D>
 * to store the data for in an array.
 *
 * It has a maxLoops for insertion at 300. Unless specified the maximum load of
 * the table will be set to 50%, if the load gets above that a resize and rehash
 * will be automaticly preformed. If nothing else specified it starts off with
 * an initial size of 16 elements.
 *
 * This version of cuckoo hashing has two hash functions which means that for
 * look ups only two places in the table are need to examine.
 *
 * Null keys are not allowed. Neither are doublets of keys.
 *
 * @author Tommy Petterson
 * @version 2012-04-12
 */
public class CuckooHashMap<K, D> {
    public static final int MAX_SIZE = 10000000, DEFAULT_MAX_LOOPS = 300,
            DEFAULT_INITIAL_SIZE = 16;
    public static final double DEFAULT_LOADFACTOR = 0.5;
    private final int MAX_BOUNCE = 5;
    protected Element<K, D>[] t;
    protected double loadFactor;
    protected int size, load, loadMax, maxLoops;
    private HashFunction hf;

    /**
     * Constructor for CuckooHashMap using the default values for initial size
     * and load factor.
     */
    public CuckooHashMap() {
        this(DEFAULT_INITIAL_SIZE, DEFAULT_LOADFACTOR);
    }

    /**
     * Constructor for CuckooHashMap that sets initial size and load factor to
     * the values specified as parameters if these comply with the limits set.
     * Initial size has to be above 0 and load factor above 0, but not above 1.
     *
     * @param initialSize The initial size for the table used
     * @param loadFactor The highest load factor allowed before the table gets
     * resized
     */
    public CuckooHashMap(int initialSize, double loadFactor) {
        if (initialSize > 0) {
            size = initialSize;
        } else {
            size = DEFAULT_INITIAL_SIZE;
        }
        t = new Element[size];
        load = 0;
        if (loadFactor > 0 && loadFactor <= 1) {
            this.loadFactor = loadFactor;
        } else {
            this.loadFactor = DEFAULT_LOADFACTOR;
        }
        loadMax = (int) (size * this.loadFactor + 0.5);
        hf = new HashFunction();
    }

    /**
     * Puts the key and data specified in the arguments into the table. If the
     * key is null no insert will be preformed. If the table already contains
```

```
     * the key the data mapped to the key will be changed to the data in the
     * arguments. If the table size exceeds the maximum table size the key and
     * data may not be inserted.
     *
     * Returns true if the insert was successful, and false if it was not or if
     * the key was null
     *
     * @param key The key to be inserted
     * @param data The data to be inserted
     * @return True if the the key and data was instered, otherwise false
     */
    public boolean put(K key, D data) {
        if (key == null) {
            return false;
        }
        return put(key, data, 0);
    }

    /**
     * Puts the key and data specified in the arguments into the table. If the
     * table already contains the key the data mapped to the key will be changed
     * to the data in the arguments. If the table size exceeds the maximum table
     * size the key and data may not be inserted.
     *
     * If the load exceeds the maximum load allowed according to load factor a
     * resize of the table will be preformed.
     *
     * If the element cannot be inserted five attempts to rehash the table will
     * be preformed before resizing the table.
     *
     * Returns true if the insert was successful, and false if it was not or if
     * the key was null
     *
     * @param key The key to be inserted
     * @param data The data to be inserted
     * @param bounce Value used to check the number of tries
     * @return True if the the key and data was instered, otherwise false
     */
    private boolean put(K key, D data, int bounce) {
        if ((load > loadMax) && (size < MAX_SIZE)) {
            rehash(Integer.MAX_VALUE);
        }

        Element<K, D> e = insertElement(key, data);
        if (e == null) {
            return true;
        }

        bounce++;
        if (bounce == MAX_BOUNCE && size > MAX_SIZE) {
            return false;
        }

        rehash(bounce);
        return put(e.key, e.getData(), bounce);
    }

    /**
     * Insert the actual key and data as an element in the table. Here it is
     * assumed that the key is not null. If the key already exists in the table
     * the data previous mapped to the key will be replaced by the new data
     * specified as an argument.
     *
     * Returns null if the element was successfully inserted and no other
     * element stands without a place. However, if when maxLoops have been
     * reached an element still is outside the table that element will be
     * returned
     *
     * @param key The key to be inserted
```

```
     * @param data The data to be inserted
     * @return null if the insertion was successfull, the Element not in the
     * table if unsuccessful.
     */
    private Element insertElement(K key, D data) {
        int newPos = hf.hash(key, t.length, 1);
        if (t[newPos] != null && t[newPos].key.equals(key)) {
            t[newPos].setData(data);
            return null;
        }
        int currentPos = hf.hash(key, t.length, 2);
        if (t[currentPos] != null && t[currentPos].key.equals(key)) {
            t[currentPos].setData(data);
            return null;
        }

        Element<K, D> e = new Element(key, data, newPos);

        for (int i = 0; i < maxLoops; i++) {
            newPos = e.hashinfo;

            if (t[newPos] == null) {
                e.hashinfo = currentPos;
                t[newPos] = e;
                load++;
                return null;
            }

            Element<K, D> e2 = t[newPos];
            e.hashinfo = currentPos;
            t[newPos] = e;
            currentPos = newPos;
            e = e2;
        }
        return e;
    }

    /**
     * Rehashes the hash table. If called with a bounce higher than the
     * MAX_BOUNCE it will resize the table to a size twice the size it had
     * before. Otherwise only the hash function will be rebuilded and the
     * elements inserted again.
     *
     * @param bounce the number specifying what should be done
     */
    private void rehash(int bounce) {
        if (bounce > MAX_BOUNCE) {
            Element<K, D>[] old_t = t;
            int oldSize = size;
            size = oldSize * 2;
            loadMax = (int) (size * loadFactor + 1);
            load = 0;
            hf = new HashFunction();

            t = new Element[size];

            for (int i = 0; i < oldSize; i++) {
                if (old_t[i] != null) {
                    Element<K, D> e = insertElement(old_t[i].key, old_t[i].getData());
                    if (e != null) {
                        hf.rebuildHash();
                        i = -1;
                        load = 0;
                        t = new Element[size];
                    }
                }
            }
        } else {
            Element<K, D>[] old_t = t;
```

```
        int oldSize = size;
        size = oldSize;
        loadMax = (int) (size * loadFactor + 1);
        load = 0;
        hf.rebuildHash();

        t = new Element[size];

        for (int i = 0; i < oldSize; i++) {
            if (old_t[i] != null) {
                Element<K, D> e = insertElement(old_t[i].key, old_t[i].getData());
                if (e != null) {
                    hf.rebuildHash();
                    i = -1;
                    load = 0;
                    t = new Element[size];
                }
            }
        }
    }
}

/**
 * Builds and returns a string representation of the table.
 *
 * @return A string representation of the table.
 */
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("[ ");
    for (int i = 0; i < size - 1; i++) {
        if (t[i] != null) {
            sb.append((t[i].key + ":" + t[i].getData() + ", "));
        } else {
            sb.append(" null, ");
        }
    }

    if (t[size - 1] != null) {
        sb.append((t[size - 1].key + ":" + t[size - 1].getData() + " ]"));
    } else {
        sb.append("null ]");
    }
    return sb.toString();
}

/**
 * Checks if a K key recieved as an argument in the method is stored in the
 * hash table. Returns false if the key is null or the key was not stored in
 * the hash table. Returns false if the key was stored in the hash table.
 *
 * @param key The key to check if the table contains
 * @return True if the table contains the key, otherwise or if the key is
 * null false is returned
 */
public boolean containsKey(K key) {
    if (key == null) {
        return false;
    }
    int hash = hf.hash(key, t.length, 1);
    Element<K, D> e = t[hash];
    if (e != null && e.key.equals(key)) {
        return true;
    }


    hash = hf.hash(key, t.length, 2);
    if (t[hash] != null && t[hash].key.equals(key)) {
```

```
            return true;
        }

        return false;
    }

    /**
     * Removes the key specified as an arguement from the table if the key is
     * stored there and returns the data associated with it. If the key is null
     * or if the table does not contain the key null is returned
     *
     * @param key The key to remove from the hash table
     * @return The data associated with the key, or null if the key is null or
     * the table does not contain the key.
     */
    public boolean removeKey(K key) {
        if (key == null) {
            return false;
        }
        int hash = hf.hash(key, t.length, 1);
        if (t[hash] != null && t[hash].key.equals(key)) {
            t[hash] = null;
            load--;
            return true;
        }

        hash = hf.hash(key, t.length, 2);
        if (t[hash] != null && t[hash].key.equals(key)) {
            t[hash] = null;
            load--;
            return true;
        }
        return false;
    }

    /**
     * Returns the number of elements stored in the table.
     *
     * @return The number of elements stored
     */
    public int getSize() {
        return load;
    }

    /**
     * Returns the length of the table.
     *
     * @return the table length
     */
    public int getLength() {
        return size;
    }
}
```

Bilaga 2: Element

```
/**
 * This simple element class have been created to work for storing both data and
 * keys in hashmaps. It only has three fields, K key, D data and hashinfo, an
 * int. The first one has to be specified during the constructor and is final
 * to avoid problems with a key suddenly being out of place in the hashtable.
 *
 * Hashinfo has a purpose in storing hashinginfo to minimize the number of
 * evaluations of hashfunctions.
 *
 * @author Helena Sjöberg
 * @version 2012-03-23
 */
public class Element<K, D> {
```

```
    public final K key;
    private D data;
    public int hashinfo;

    /**
     * Constructor for the class Element. Key have to be specified here
     * and is final, so no changes are allowed. Hashinfo is a field
     * enabeling storage of hashing calculations.
     *
     * @param key The key for the element to be stored
     * @param data The data to store
     * @param hashinfo Hashing information needing to be stored
     */
    public Element(K key, D data, int hashinfo) {
        this.key = key;
        this.data = data;
        this.hashinfo = hashinfo;
    }

    /**
     * Returns the stored data.
     * @return the data stored
     */
    public D getData(){
        return data;
    }

    /**
     * Sets the data field to the value given as an argument.
     * @param data The new data that will be stored
     */
    public void setData(D data){
        this.data=data;
    }
}
```

Bilaga 3: HashFunction

```
import java.util.Random;

/**
 * Originally created by
 * @author Joacim Alvergren
 * at 2010-06-06
 *
 * Edited by
 * @author Tommy Petterson
 * @version 2012-03-28
 */
public class HashFunction {

    private final Random ENGINE = new Random();
    private int rounds1, rounds2;

    public HashFunction() {
        this.rounds1 = 1;
        this.rounds2 = 2;
    }

    public int hash(Object key, int limit, int select) {
        int rounds = (select == 1) ? rounds1 : rounds2;
        ENGINE.setSeed(key.hashCode());
        int h = ENGINE.nextInt(limit);
        for (int i = 1; i < rounds; i++) {
            h = ENGINE.nextInt(limit);
        }

        return h;
    }

    public int hash(Object key, int limit){
        return hash(key, limit, 1);
    }

    public void rebuildHash() {
        rounds1++;
        rounds2++;
    }
}
```

Bilaga 4: HopscotchHashMap

```
/**
 * This is a class for a hashmap built on the hopscotch hashing algorithm. It
 * maps a K key to some Data D. It can take any initial size value and is not
 * limitied to a size which is the power of two. It uses the class Element<K, D>
 * to store the data for in an array.
 *
 * It has a neigbourhood size of 32, unless that is more than half the table
 * size. If that is the case, half the table is the neighbourhood size. This
 * means that the methods conatainsKey(K key), delete(K key) and get(K key)
 * never will need to look through more than 32 elements.
 *
 * If nothing else specified it starts off with an initial size of 16 elements.
 *
 * Null keys are not allowed. Neither are doublets of keys.
 *
 * @author Helena Sjöberg
 * @version 2012-04-09
 */
public class HopscotchHashMap<K, D> {
    protected Element<K, D>[] t;
    protected double hFactor;
    protected int length, load, neighbourhood;
    public static final int DEFAULT_SIZE = 16, DEFAULT_H_MAX = 32;
    public static final double DEAFAULT_H_FACTOR = 0.5;

    /**
     * Constructor for HopscotchHash using the default value for initial size of
     * 32.
     */
    public HopscotchHashMap() {
        this(DEFAULT_SIZE);
    }

    /**
     * Constructor for Hopscotch with the specified value given as input as the
     * initial size of the table.
     *
     * @param initialSize The initial size of the table
     */
    public HopscotchHashMap(int initialSize) {
        if (length > 0) {
            this.length = initialSize;
        } else {
            length = DEFAULT_SIZE;
        }
        t = new Element[length];
        neighbourhood = Math.min((int) (length * this.hFactor + .5), DEFAULT_H_MAX);
        load = 0;
    }

    /**
     * Searches the hashtable for the key given as input and if found it returns
     * the index of that key. If the key was not found it returns -1. Does not
     * take null as input.
     *
     * @param key The key to be searched for
     * @return The index of the place the key was found in, or -1 if not found.
     */
    protected int getIndex(K key) {
        int pos = hashCode(key);
        for (int i = 0; i <= neighbourhood; i++) {
            if (t[pos] != null) {
                if (t[pos].key.equals(key)) {
                    return pos;
                }
            }
```

```
            pos = (pos + 1) % length;
        }
        return -1;
    }

    /**
     * Takes a K key as input and calculates the hash value for that key using
     * the hashCode() method modulus the length of the hash table.
     *
     * @param key The key to be calculated a hash value for
     * @return The hash value for the key specified
     */
    protected int hashCode(K key) {
        return key.hashCode() % length;
    }

    /**
     * Inserts a Element consisting of the K key and the D data specified as the
     * arguments of this method. This method assumes that the key is not null
     * and that the key haven't been inserted before.
     *
     * Returns true if the element was successfully inserted, otherwise false.
     *
     * @param key The key to be inserted
     * @param data The data to be inserted
     * @return true if the key and data was inserted, otherwise false.
     */
    protected boolean insert(K key, D data) {
        int hashPos = hashCode(key);
        int pos = hashPos;
        for (int i = 0; i < length; i++) {
            if (t[pos] == null) {
                if (((pos - hashPos + length) % length) <= neighbourhood) {
                    t[pos] = new Element(key, data, hashPos);
                    load++;
                    return true;
                } else {
                    int p = moveTo((pos + length) % length);
                    if (p == -1) {
                        return false;
                    } else {
                        pos = p;
                        i = 0;
                    }

                }
            } else {
                pos = (pos + 1) % length;
            }
        }
        return false;
    }

    /**
     * Takes an index in the table assumed to be free and tries to move another
     * element there. The element being moved will be positioned at an index as
     * far away as possible but not more than a neighbourhood away.
     *
     * Returns the index of the new free place where the element was moved from
     * is successful, otherwise -1 is returned.
     *
     * @param indexFree the free index where another element will be moved
     * @return the index of the new free space is successful, otherwise -1
     */
    protected int moveTo(int indexFree) {
        int free = indexFree;
        int pos = (free - neighbourhood + length) % length;
        while (((free - pos + length) % length) > 0) {
            if (t[pos] != null) {
```

```
            if (((t[pos].hashinfo + neighbourhood) % length) >= free) {
                t[free] = t[pos];
                t[pos] = null;
                return pos;
            }
        }
        pos = (pos + 1) % length;
    }
    return -1;
}

/**
 * The rehash method for HopscotchHashMap. Doubles the size of the table,
 * recalculates neighboorhood and inserts all the element that were in the
 * old table.
 */
protected void rehash() {
    length = length * 2;
    if (neighbourhood != DEFAULT_H_MAX) {
        neighbourhood = Math.min(DEFAULT_H_MAX, (int) (length * hFactor + .5));
    }
    load = 0;
    Element<K, D>[] t2 = t;
    t = new Element[length];
    for (int i = 0; i < t2.length; i++) {
        if (t2[i] != null) {
            if (!insert(t2[i].key, t2[i].getData())) {
                t = t2;
                rehash();
                break;
            }
        }
    }
}

/**
 * Replaces the data stored with the K key specified as an argument with
 * the new D data, also specified as an argument. Assumes the key is not
 * null.
 *
 * Returns true if the key was stored in the table and the replace was
 * successful, otherwise false.
 *
 * @param key The key that should lead to the new data
 * @param data The data that should be stored
 * @return true if the key was stored in the table and the replacement was
 * successful, otherwise false
 */
protected boolean replace(K key, D data) {
    int pos = getIndex(key);
    if (pos == -1) {
        return false;
    } else {
        t[pos].setData(data);
        return true;
    }
}

/**
 * Checks if a K key recieved as an argument in the method is stored in the
 * hash table. Returns false if the key is null or the key was not stored
 * in the hash table. Returns false if the key was stored in the hash
 * table.
 *
 * @param key The key to check if the table contains
 * @return True if the table contains the key, otherwise or if the key is
 * null false is returned
 */
public boolean containsKey(K key) {
```

```
    if (key == null) {
        return false;
    }
    int i = getIndex(key);
    if (i == -1) {
        return false;
    } else {
        return true;
    }
}

/**
 * Returns the data associated with the key provided as an argument. If the
 * key is null or if the key is not stored in the table null will be
 * returned.
 *
 * @param key The key to get the data associated with
 * @return The D data the key is linked too, or null if the key is null or
 * the table does not contain the key
 */
public D getData(K key) {
    if (key == null) {
        return null;
    }
    int i = getIndex(key);
    if (i == -1) {
        return null;
    } else {
        return t[i].getData();
    }
}

/**
 * Returns the length of the table.
 *
 * @return the table length
 */
public int getLength() {
    return length;
}

/**
 * Returns the number of elements stored in the table.
 *
 * @return The number of elements stored
 */
public int getSize() {
     return load;
}

/**
 * Removes the key specified as an arguement from the table if the key is
 * stored there and returns the data associated with it. If the key is
 * null or if the table does not contain the key null is returned
 *
 * @param key The key to remove from the hash table
 * @return The data associated with the key, or null if the key is null or
 * the table does not contain the key.
 */
public D remove(K key) {
    if (key == null) {
        return null;
    }
    int pos = getIndex(key);
    if (pos == -1) {
        return null;
    }
    D d = t[pos].getData();
    t[pos] = null;
```

```
        return d;
    }

    /**
     * Puts the K key and the D data given as arguments into the table. If the
     * key is null nothing will happen. If the table already contains the key
     * the data associated with the key will be overwritten with the new data.
     *
     * If the key cannot be inserted in the current table a rehash will be
     * preformed and the size of the table will be doubled
     *
     * @param key The key to be inserted
     * @param data The data to be inserted
     */
    public void put(K key, D data) {
        if (key == null) {
            return;
        }
        if (load == length) {
            rehash();
        }
        if (replace(key, data)) {
            return;
        }
        if (insert(key, data)) {
            return;
        }
        rehash();
        put(key, data);
    }

    /**
     * Builds and returns a string representation of the table.
     *
     * @return A string representation of the table.
     */
  @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[ ");
        for (int i = 0; i < length - 1; i++) {
            if (t[i] != null) {
                sb.append((t[i].key + ":" + t[i].getData() + ", "));
            } else {
                sb.append(" null, ");
            }
        }

        if (t[length - 1] != null) {
            sb.append((t[length - 1].key + ":" + t[length - 1].getData() + " ]"));
        } else {
            sb.append("null ]");
        }
        return sb.toString();
    }
}
```

Bilaga 5: Tabeller

Tabell 1:

CuckooHashMap

genomsnittlig tid vid olika värden på maxLoops

Antal element \ maxLoops	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
32	0,62	0,00														
64	0,47	0,16	0,00													
128	29,96	0,00	0,16	0,16												
256	2044,03	0,15	0,62	0,00	0,32											
512					0,63	777,00										
1024					2,20	1,42	1,24									
2048					2,34	2,36	2,98	3,14								
4096					6,23	6,39	6,23	5,90	6,42							
8192					26,80	26,66	26,78	27,14	26,87	26,84						
16384					226,66	29,35	29,66	29,63	29,33	30,09	29,02					
32768					74,12	262,84	270,49	270,97	270,41	271,33	269,85	270,86				
65536					1746,87	147,59	149,48	150,69	148,55	148,95	148,66	150,07	147,41			
131072					779,98	311,02	313,54	311,07	310,59	311,23	313,15	310,29	312,47	311,02		
262144						657,42	649,58	648,32	649,86	651,46	657,42	653,32	654,84	670,44	671,70	
524288						5864,66	5719,10	4270,94	4212,56	4290,98	4278,06	4211,04	4206,68	4218,64	4220,80	4222,26
1048576						102553,10	9979,30	3196,70	3179,30	3120,20	2987,60	3001,40	3015,50	2982,70	3009,40	3034,10
2097152							6807,70	21063,10	20733,70	20554,40	20830,70	20768,10	20521,80	20726,30	20511,00	20693,60

Tabell 2:

CuckooHashMap

genomsnittlig tabellstorlek vid olika värden på maxLoops

Tabellstorlek \ maxLoops	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
32	64	64														
64	128	128	128													
128	256	256	256	256												
256	512	512	512	512	512											
512					1024	1024										
1024					2048	2048	2048									
2048					4096	4096	4096	4096								
4096					8192	8192	8192	8192	8192							
8192					16384	16384	16384	16384	16384	16384						
16384					32768	32768	32768	32768	32768	32768	32768					
32768					65536	65536	65536	65536	65536	65536	65536	65536				
65536					131072	131072	131072	131072	131072	131072	131072	131072	131072			
131072					262144	262144	262144	262144	262144	262144	262144	262144	262144	262144		
262144						524288	524288	524288	524288	524288	524288	524288	524288	524288	524288	
524288						1048576	1048576	1048576	1048576	1048576	1048576	1048576	1048576	1048576	1048576	1048576
1048576						2097152	2097152	2097152	2097152	2097152	2097152	2097152	2097152	2097152	2097152	2097152
2097152							4194304	4194304	4194304	4194304	4194304	4194304	4194304	4194304	4194304	4194304

Tabell 3:

CuckooHashMap

genomsnittlig tid vid olika värden på loadFactor

Antal element \ loadFactor	0,2	0,3	0,4	0,45	0,46	0,47	0,48	0,49	0,5	0,51	0,52	0,53	0,54
128	1,88	0,64	0,32	0,00	0,00	1,50	0,00	0,00	1,60	0,00	4,70	1,50	0,00
256	1,88	0,32	0,94	0,00	0,00	0,00	0,00	1,60	0,00	1,50	0,00	0,00	1,60
512	1,56	0,92	1,26	1,50	1,60	0,00	1,60	1,50	1,60	1,50	1,60	1,60	3,10
1024	2,50	1,88	2,50	3,10	3,10	3,10	3,10	3,20	3,10	3,10	3,10	4,70	3,10
2048	5,62	4,36	5,32	6,20	4,70	7,80	6,20	6,30	6,20	6,20	6,30	9,30	7,80
4096	11,86	9,06	10,92	10,90	14,00	12,50	14,10	12,40	14,10	14,00	15,60	15,60	20,30
8192	24,32	18,72	22,78	25,00	26,50	29,60	29,60	29,60	28,10	32,70	37,50	39,00	56,10
16384	54,62	42,42	51,8	54,60	57,70	59,30	62,40	67,10	68,60	70,20	84,20	123,20	525,80
32768	120,12	94,54	114,2	124,80	127,90	129,50	134,10	140,40	129,50	159,50	273,00	472,70	5213,50
65536	260,52	206,86	250,86	277,60	277,70	290,20	301,10	302,60	252,70	429,00	775,30		
131072	552,88	446,16	540,38	608,40	622,40	631,80	6740,00	709,80	680,10	954,70	5290,00		
262144	1177,80	959,72	1161,26	1266,70	1296,40	1346,30	1400,90	1528,80	1692,60	2811,10	57970,00		
524288	2496,00	2026,44	2472,30	2719,10	2754,90	2853,30	2923,40	3254,20	3896,90	11473,80	52322,50		
1048576	5968,24	4592,02	5913,04	6534,80	6499,00	6809,40							

Tabell 4:

CuckooHashMap

genomsnittlig tabellstorlek vid olika värden på loadFactor

Antal element \ loadFactor	0,2	0,3	0,4	0,45	0,46	0,47	0,48	0,49	0,5	0,51	0,52	0,53	0,54
128	1024	512	512	512	512	512	512	512	256	256	256	256	256
256	2048	1024	1024	1024	1024	1024	1024	1024	512	512	512	512	512
512	4096	2048	2048	2048	2048	2048	2048	2048	1024	1024	1024	1024	1024
1024	8192	4096	4096	4096	4096	4096	4096	4096	2048	2048	2048	2048	2048
2048	16384	8192	8192	8192	8192	8192	8192	8192	4096	4096	4096	4096	4096
4096	32768	16384	16384	16384	16384	16384	16384	16384	8192	8192	8192	8192	8192
8192	65536	32768	32768	32768	32768	32768	32768	32768	16384	16384	16384	16384	16384
16384	131072	65536	65536	65536	65536	65536	65536	65536	32768	32768	32768	32768	32768
32768	262144	131072	131072	131072	131072	131072	131072	131072	65536	65536	65536	65536	65536
65536	524288	262144	262144	262144	262144	262144	262144	262144	131072	131072	131072		
131072	1048576	524288	524288	524288	524288	524288	524288	524288	262144	262144	262144		
262144	2097152	1048576	1048576	1048576	1048576	1048576	1048576	1048576	524288	524288	524288		
524288	4194304	2097152	2097152	2097152	2097152	2097152	2097152	2097152	1048576	1048576	1048576		
1048576	8388608	4194304	4194304	4194304	4194304	4194304							

Tabell 5:

HopscotchHashMap

genomsnittlig tid vid olika grannskapsstorlek

Tid (ms)

Antal element \ Grannskapsstorlek	8	16	32	64	128	256	512	1024	2048	4096	8192
128	1,56	0,94	0,3	0,32							
256		0,62	0,30	0,64	1,24	0,62					
512			1,26	1,56	2,80	4,06					
1024		1,56	2,18	3,74	6,24	9,98	15,92				
2048		3,44	4,68	7,16	13,12	23,08	40,88	63,96			
4096		7,18	9,68	15,16	28,08	50,22	92,36	166,60	261,16		
8192		14,34	19,98	31,82	57,40	107,02	201,24	374,08	671,74	1063,62	
16384		32,44	46,5	76,12	139,16	262,70	509,18	989,66	1902,9	3593,62	6524,24
32768		75,70	113,88	194,70	366,60	713,86	1427,08	2847,32	5694,94	10642,34	
65536		154,74	261,46	439,92	842,10	1682,00	3355,56	6673,38	13255,96		
131072		317,62	570,64	972,52	1898,82	3836,36	7760,08	15779,74			
262144		626,80	1198,70	2088,22	4126,84	8254,60	16412,16	29899,94			
524288		1390,00	2684,80	4308,70	8520,70	17596,90	34758,30				
1048576		2936,00	5299,30	8817,10	17536,00	34983,10	70782,00				

Tabell 6:

HopscotchHashMap

genomsnittlig tabellstorlek vid olika grannskapsstorlek

Tabellstorlek

Antal element \ Grannskapsstorlek	8	16	32	64	128	256	512	1024	2048	4096	8192
128	276	256	250	179							
256		512	512	481	332,8						
512		1065	1024	1024	901	584					
1024		2171	2048	2048	2028	1556	1126,4				
2048		4669	4096	4096	4096	3850	2867	2088			
4096		10158	8192	8192	8192	8028	6471	4506	4096		
8192		19988	16384	16384	16384	16384	15237	11141	8356	1064	
16384		50463	32768	32768	32768	32768	32112	24248	16384	16384	16384
32768		114033	65536	65536	65536	65536	65536	59637	38010	32768	
65536		254280	136315	131072	131072	131072	131072	128450	83886		
131072		524288	277872	262144	262144	262144	262144	262144			
262144		1069548	555745	524288	524288	524288	524288	524288			
524288		2097152	1258291	1048576	1048576	1048576	1048576				
1048576		4194304	2306867	2097152	2097152	2097152	2097152				

Tabell 7:

HopscotchHashMap

genomsnittlig tid vid olika värden på loadFactor

Tid (ms) / Antal element	loadFactor 0,3	0,35	0,4	0,45	0,5	0,55	0,6	0,65	0,7	0,75	0,8	0,85	0,9	0,95	1
128	1,24	0,94	0,30	0,32	0,32	0,30	0,32	0,00	0,30	0,32	0,32	0,30	0,32	0,30	0,32
256	0,30	0,62	0,32	0,62	0,62	0,32	0,62	0,32	0,62	0,32	0,62	0,62	0,32	0,62	0,62
512	0,94	0,94	0,92	0,94	0,94	0,94	0,94	0,92	0,94	1,26	0,92	0,94	1,26	1,24	0,94
1024	1,86	1,88	1,88	2,18	1,86	1,88	1,88	1,86	2,18	2,20	2,18	2,18	2,18	2,18	2,50
2048	3,44	3,74	4,06	4,06	3,74	3,74	4,06	4,06	4,36	4,36	4,68	4,68	4,68	4,68	4,68
4096	7,50	7,80	7,80	8,42	7,80	8,12	8,10	9,06	9,04	9,36	9,68	9,66	9,68	9,66	9,68
8192	15,6	16,22	17,16	17,48	16,22	16,54	17,78	18,72	19,66	19,96	20,28	20,28	20,28	20,28	20,28
16384	35,88	37,14	38,7	40,56	38,36	39,94	42,44	44,92	46,50	46,80	46,48	47,42	46,80	47,74	47,42
32768	82,38	86,1	90,48	96,10	91,74	95,78	102,64	109,82	113,26	112,32	112,64	112,00	113,88	114,5	112,64
65536	182,52	191,88	203,42	216,22	214,98	255,56	247,42	255,22	255,22	253,66	255,84	255,22	257,72	260,82	256,78
131072	392,18	415,28	441,80	473,92	468,00	514,80	541,32	563,78	562,54	570,04	567,84	564,08	565,36	563,8	565,02
262144	826,56	879,74	942,56	1009,44	1019,30	1132,88	1191,22	1214,00	1213,36	1197,14	1205,88	1188,72	1180,6	1205,88	1221,18
524288	1729,52	1841,40	1976,84	2150,66	2235,16	2513,88	2549,04	2548,42	2516,6	2573,08	2567,32	2491,64	2554,66	2557,16	2583,04
1048576	3640,00	3832,60	4124,32	4501,86	5060,96	5318,06	5450,04	5549,86	5390,44	5371,08	5387,32	5420,38	5436,00	5372,60	5288,10

Tabell 8:

HopscotchHashMap

genomsnittlig tabellstorlek vid olika värden på loadFactor

Tabellstorlek / Antal element	loadFactor 0,3	0,35	0,4	0,45	0,5	0,55	0,6	0,65	0,7	0,75	0,8	0,85	0,9	0,95	1
128	512	512	512	512	256	256	256	256	256	256	256	256	256	246	241
256	1024	1024	1024	1024	512	512	512	512	512	512	512	512	512	512	512
512	2048	2048	2048	2048	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
1024	4096	4096	4096	4096	2048	2048	2048	2048	2048	2048	2048	2048	2048	2048	2048
2048	8192	8192	8192	8192	4096	4096	4096	4096	4096	4096	4096	4096	4096	4096	4096
4096	16384	16384	16384	16384	8192	8192	8192	8192	8192	8192	8192	8192	8192	8192	8192
8192	32768	32768	32768	32768	16384	16384	16384	16384	16384	16384	16384	16384	16384	16384	16384
16384	65536	65536	65536	65536	32768	33423	32768	33423	32768	32768	32768	32768	32768	33423	32768
32768	131072	131072	131072	131072	65536	65536	65536	65536	66847	66847	66847	66847	66847	68157	65536
65536	262144	262144	262144	262144	136315	133693	131072	133693	131072	131072	131072	133693	133693	138936	131072
131072	524288	524288	524288	524288	262144	283116	262144	262144	272630	277873	272630	262144	272630	272630	267387
262144	1048576	1048576	1048576	1048576	545260	545260	545260	555745	566231	534774	555745	545260	534774	555745	587203
524288	2097152	2097152	2097152	2097152	1153434	1195377	1174405	1132462	1111491	1174405	1195377	1090519	1153434	1216348	1216348
1048576	4194304	4194304	4194304	4194304	2642412	2516582	2642412	2768241	2516582	2516582	2474639	2600468	2684355	2474639	2348810

Tabell 9 & 10:

Jämförelse

enbart insättning av element

Typ av hashtabell / Tid (ms)	Antal element 8000	16000	32000	64000	128000	256000	512000	1024000
CuckooHashMap	15,60	18,70	43,70	86,60	246,45	531,20	2029,55	
HopscotchHashMap	13,25	20,25	42,90	112,35	270,7	396,25	1210,55	2205,10
HashMap	5,45	32,00	10,90	38,20	74,10	151,35	340,85	754,25
Concurrent 1 tråd	10,90	21,05	14,05	32,00	78,75	171,60	362,75	732,45
Concurrent 2 trådar	4,65	5,50	28,85	34,30	79,60	162,20	368,95	720,30
Concurrent 4 trådar	3,10	6,25	29,65	35,85	80,35	152,10	371,30	692,65

Tabell 11:

Jämförelse

enbart containsKey, 10 000 operationer

genomsnitt

Typ av hashtabell / Tid (ms)	Antal element 32000	64000	128000	256000	512000
CuckooHashMap	53,93	61,95	68,31	74,15	
HopscotchHashMap	23,09	21,51	23,08	25,95	28,35
HashMap	20,15	26,16	26,98	25,37	27,91

Tabell 12:

Jämförelse

enbart containsKey, 10 000 operationer

värsta fallet

Typ av hashtabell / Tid (ms)	Antal element 32000	64000	128000	256000	512000
CuckooHashMap	63	78	125	187	
HopscotchHashMap	63	32	32	32	32
HashMap	515	406	453	406	46