[LABORATION 2]

7 december 2015

Victor Persson, Viktor Fällman, Adam Lowert
Luleå Tekniska Universitet
971 87 Luleå
Algoritmer och Datastrukturer, D0012E

Innehåll

Inledning

Syftet med denna laboration är att implementera dels ett Hashtable med linear probing och även 2 varianter på detta. De olika versionerna kommer vara dessa:

Version 1, en variation på vanlig linear probing som tar ett keyvärde x, detta värde körs i en hashfunktion för att bilda $h(x)$. Från $h(x)$ fås en position i hashtable genom modulo tableLängd. Om positionen är tom så placeras x där, annars beror nästa steg på två variabler, l_{Up} och l_{Down}. Om $l_{Down} \leq l_{Up}$ så kommer nästa position som kollas vara $h(x)$+1 modulo tableLängd och när en tom position hittas så sätts l_{Down}+=1, annars så blir nästa position som kollas $h(x)$-1 modulo tableLängd och därefter blir l_{Up} += 1.

Version 2, en annan version av linear probing, först kollar vi om positionen är tom, om inte så återfinns den första tomma positionen enligt $h(x)$+A modulo tableLängd där A är positiva konstanter $(1, 2, 3, \ldots, tableLängd - 1)$. Om denna position J är $\leq C$, (C*är en konstant som är bestämd*), steg från $h(x)$ så placeras x där. Om inte så kollar vi varje element som är sparat från $h(x)$ till $h(x)$+C efter ett y på position J'. Om avståndet från $h(y)$ till $J \leq C$ så placeras y på plats J och x sparas på J'. Om inget sådant element y återfinns, så kommer rehashar man hashtablet. Detta genomförs genom att ett nytt hashtable skapas som är R gånger så stor som tidigare, och sedan sätts varje element i det gamla hashtablet in där.

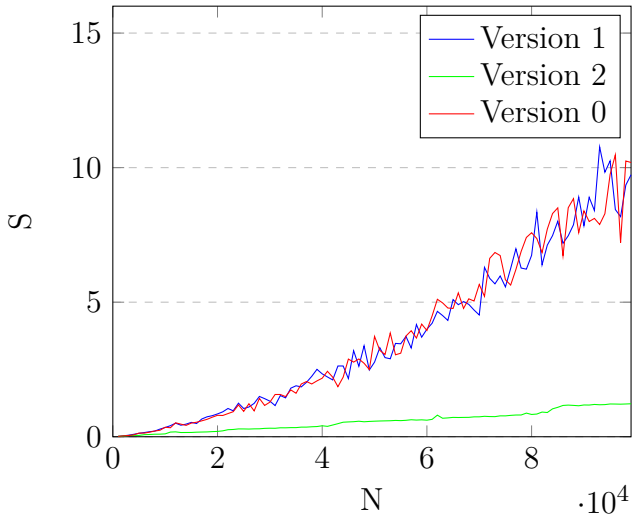
Version 0, vanlig linear probing, $h(x)$ och positionen i hashtablet tas fram på samma sätt som tidigare, om platsen är tom så placeras x där med sin tillhörande data, om x är full så kollas $h(x)$+A modulo tableLängd och så fort som en tom plats hittas så placeras x där.

Detta för att testa skillnaderna mellan dem i olika aspekter. Det som kommer tas upp är bland annat tiden för att sätta in N mängd element i ett hashtable med storlek L, tiden för att sätta in N element i ett hashtable med konstant storlek L. För version 2 kommer olika värden för C testas samt olika värden för R för att se dess påverkan på storleken på hashtablet samt den totala tiden för insättningarna. Sedan kommer vi kolla på kollisioner och hur olika C värden påverkar antalet samt kollisionkedjor. Hur loadfactorn samt antal probes för Version 2 tillhandahåller sig jämfört med de två andra versionerna beroende på antal element.

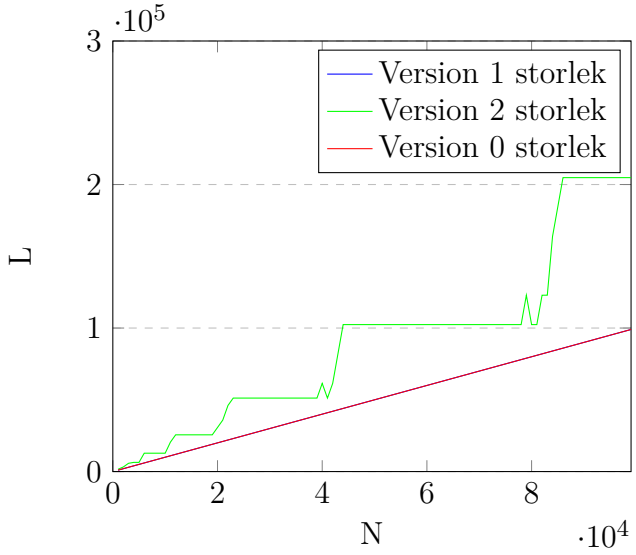
Mätdata

R: är variabeln som användes för förstoring av hashtablet vid rehash.
C: är variabeln C som specifierades i labinstruktionen.
L: är längden på hashtablet.
N: är antalet element som insätts.
S: är tiden det tar för insättning av N antal element.
$frac$: är andelen positioner i hashtablet som används.
probes: är antalet positioner som kollas fram tills man hittar en tom position.
collisions: är antalet försök att sätta in ett element x på sin orginalplats $h(x)$ som inte går.
chainlength: är antalet element i rad som inte kan placeras på sin orginalposition.

En mängd mätningar genomfördes och mätdatan visas på grafer här under. Dessa mätningar genomfördes genom att ta medelvärdet på 5 olika försök. Först ut så testades en insättning av mellan $1,000$ och $99,000$ element och resultaten finns i Figur. 1 och Figur. 2. För version 1 samt version 0 så användes en storlek på hashtable som motsvarade antalet element, $1,000$ element sattes t.ex. in i ett hashtable med storlek 1,000. För version 2 så användes värdena $R = 2$, $C = 100$ om inget annat anges vid graferna, och startstorleken på hashtablet var 100.

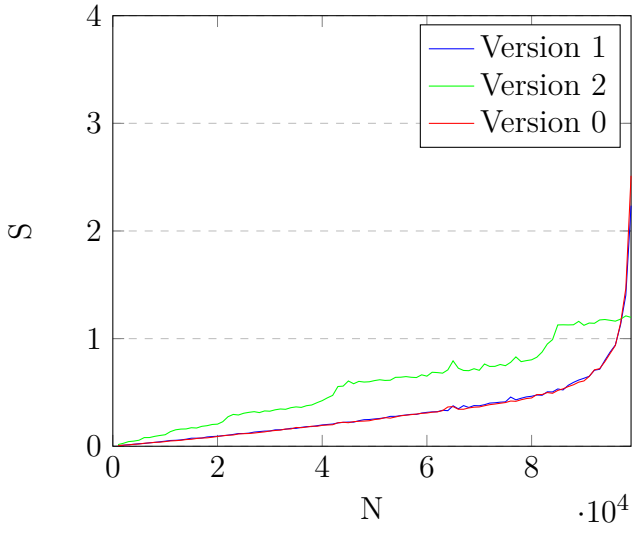


Figur 1: Insättning av 1000-99000 element, tid för insättning.

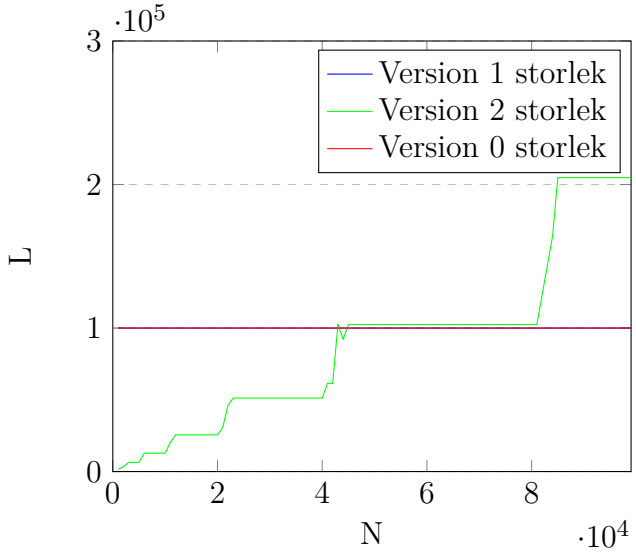


Figur 2: Insättning av 1000-99000 element, storlek på hashtable.

Figur. 3 och Figur. 4 visar mätdatan från tester med en fast storlek på hashtablet. Även här insattes $1,000$ till $99,000$ element, men med ett hashtable med storlek $100,000$. För Version 2 användes värdena $R = 2$ samt $C = 100$ som tidigare.

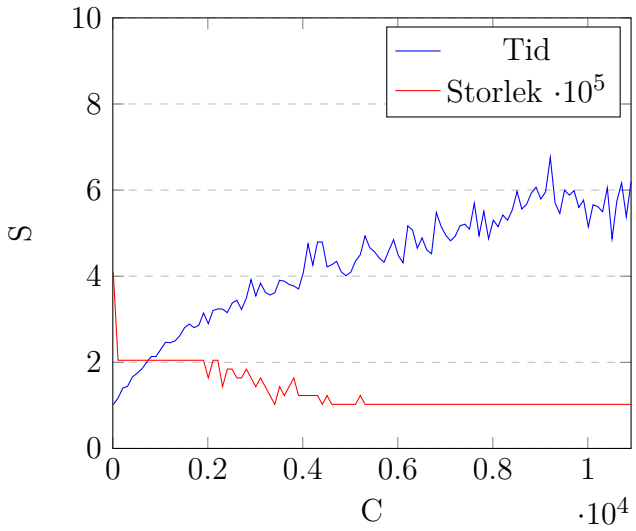


Figur 3: Insättning av 1000-99000 keys med hashtablestorlek 100.000



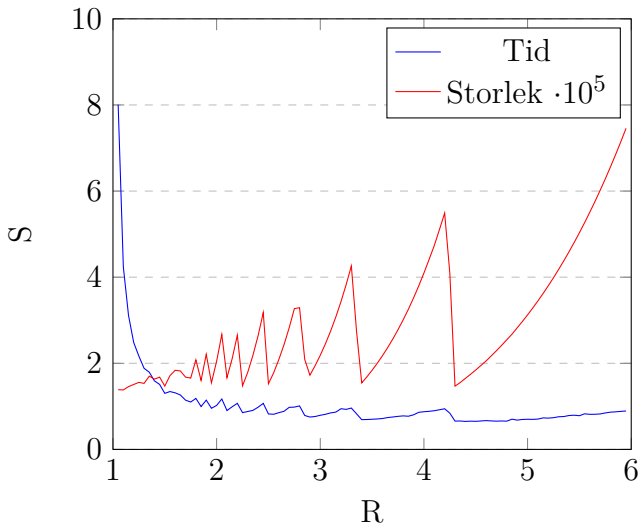
Figur 4: Insättning av 1000-99000 keys med hashtablestorlek 100.000

Figur. 5 visar mätvärden från tester på version 2. Här ändrades C, (distansen som probas), från 10 - 10910. Antalet element som insätts är $100,000$ och $R = 2$. I grafen visas tiden som insättningen tar samt storleken på hashtablet efter insättningen. Storleken ska utläsas som $S \cdot 10^5$.



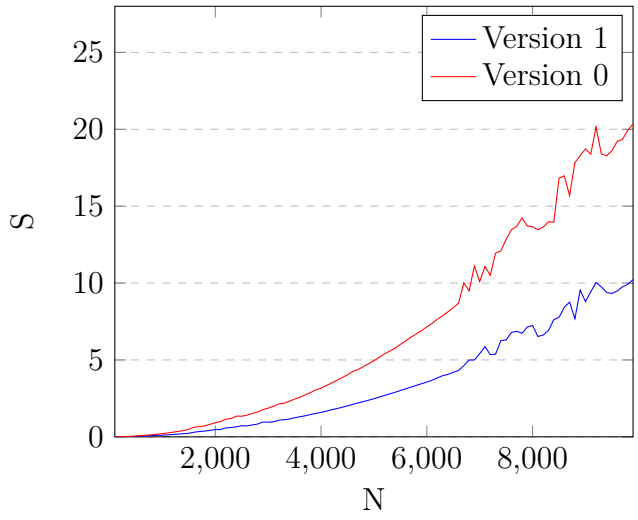
Figur 5: Insättning 100000 keys, rehash faktor 2, med varierade C värdens tider och storlek

Figur. 6 visar mätvärden från tester på version 2. Här ändrades R, (förstoringen för rehashfunktionen), från 1.05 - 5.95. Antalet element som insätts är $100,000$ och $C = 50$. I grafen visas tiden som insättningen tar samt storleken på hashtablet efter insättningen. Storleken ska utläsas som $S \cdot 10^5$.



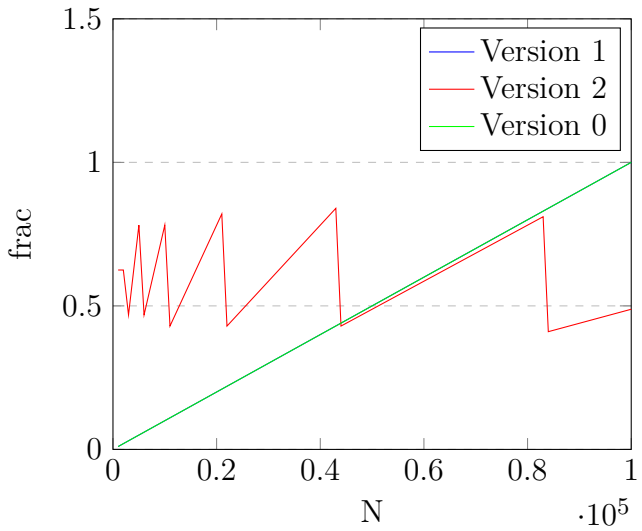
Figur 6: Version 2 med olika multiplikationsfaktorer för rehashförstorings tid och storlek

Figur. 7 visar mätvärden från tester på version 1 och 0. Här sätts mellan 100 - 9900 element in i ett hashtable som är lika stort som antal element. Skillnaden mot tidigare är att alla element kommer hashas till samma position $h(x)$. Skillnaderna i hur de två funktionerna letar upp en tom position i förhållande till $h(x)$ visas i grafen i form av insättningstid.



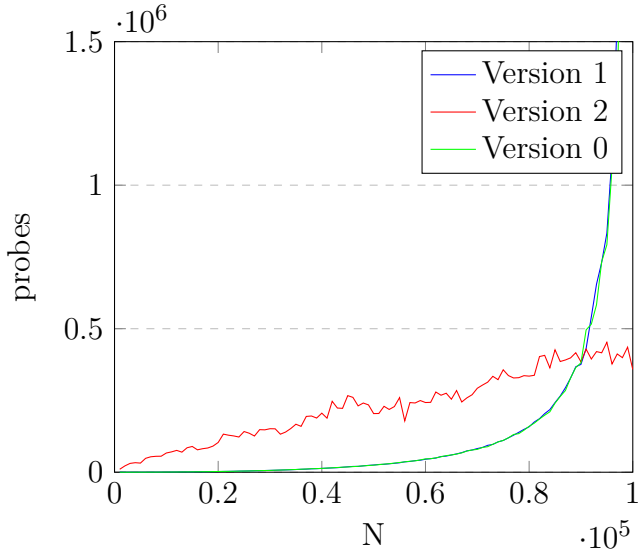
Figur 7: Version 1 och 0, samma position, 100 - 9900 element.

Figur. 8 Load factor är hur stor del av hashtabelet som är fyllt, 1000 - 100000 element i 100000 lista. C = 100



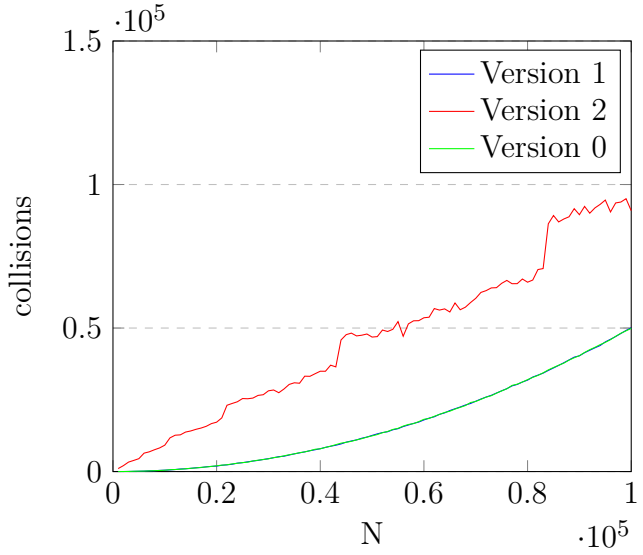
Figur 8: load factor

Figur. 9 antal försök att sätta in 1000 - 100000 element i ett hashtabele av längd 100000.



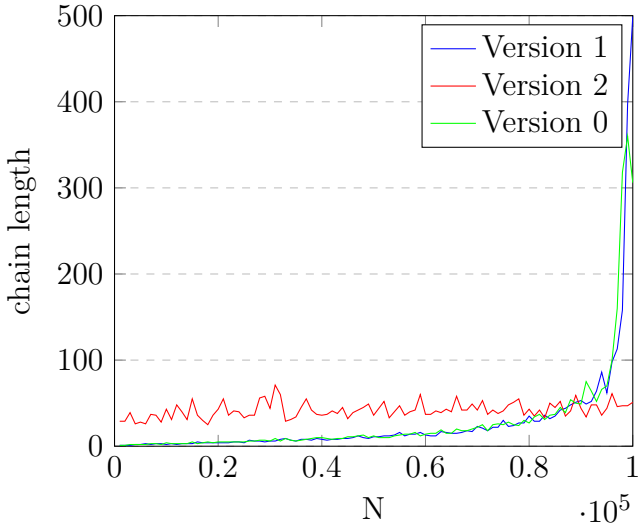
Figur 9: probes

Figur. 10 antal element som inte kan sättas in på den plats som ges av $h(x)$. 1000 - 100000 element i 100000 lista. C = 100



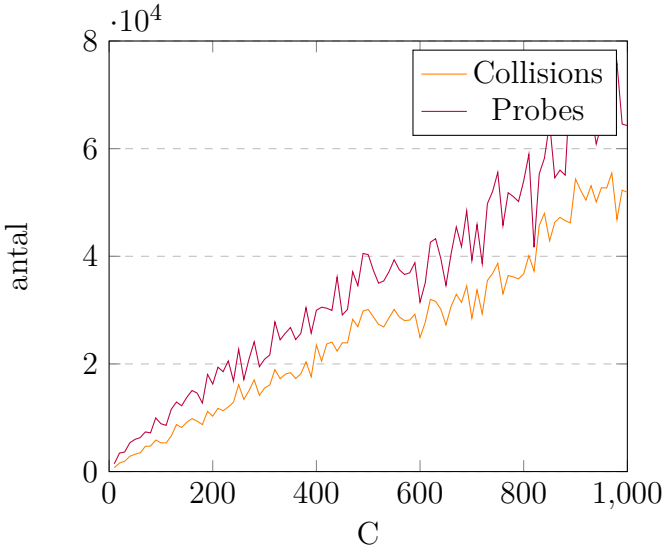
Figur 10: collisions

Figur. 11 maximalt antal element i rad som inte kan sättas in där de där de bör vara. 1000 - 100000 element i 100000 lista. C = 100



Figur 11: collision chain

Figur. 12 antal collisions och probes för varierande C och 100000 element i verson 2.



Figur 12: collisions och probes för olika C

Diskussion

Från figur. 1 kan vi se tydligt hur skillnaderna i insättningstid mellan version 2 och de andra är. Från figur. 3 så kan vi se att skillnaden mellan version 2 och de andra endast blir tydlig när man närmar sig ett fullt hashtable. Faktumet är att fram tills runt 90 procent av elementen blivit insatta så är Version 2 inte lika snabb som de andra funktionerna. Men de sista 10 procenten är det som leder till tidsskillnaderna i figur. 1.
Vi kan se på figur. 2 att minnesanvändningen för version 2 alltid är större än den för de andra två versionerna. Dock så ser vi att storleksskillnaderna blir minimala för vissa antal element. Dessa platser är beroende på startstorleken på hashtablet och multiplikationsvärdet på rehashfunktionen. Så ett mindre värde på R och ett större C hade kunnat gett en funktion som närmar sig minnesanvändningen för de andra versionerna medan den ännu tillhandahåller en snabbare insättningstid. Här får man värdera hur viktigt tiden kontra minnesanvädningen är, samtidigt som ett litet C värde ger en snabb tid för att finna och radera element. Detta då varje värde x, kommer vara placerad som max C positioner från h(x). Intressant i figur. 1 är att skillnaden mellan version 1 och version 0 är obefintlig, men nu i efterhand så är det resultatet självklart. De två variablerna lUp och lDown har endast positiv påverkan då samma element ska sättas på samma position h(x), vilket kan ses i figur. 7. Antalet probes kommer här vara N/2 för version 0 då hälften av alla element som sattits in är före h(x) och efter under h(x), medans den för version 1 är N, då alla element sattits in efter h(x).
En intressant detalj är hur vi i version 2 bestämt att rehasha tablet när vi inte kan hitta ett element att placera på tomma platsen, här hade man kunnat implementera att hashtablet rehashas när det är fullt till hälften istället t.ex. Då skulle antalet rehashar kunna minskas när man har otur med vilka element som sätts in, men då skulle man inte kunna lova att element x är inom C positioner från h(x). Från figur. 4 så kan vi se en intressant detalj om minnesanvändningen när det gäller version 2 mot de två andra versionerna. Faktumet är att version 1 och version 0 kommer ha en fast storlek på hashtablet, och med denna implementering så måste storleken vara större än antalet element som ska sättas in annars kommer det resultera i att data kommer förloras. Och om man inte vet exakt hur många element som kommer finnas i hashtablet alltid så kommer version 2 resultera i att storleken mer matchar antalet element. Något som skulle kunnat implementeras är en förminsking av hashtablet när element tas bort, så att om hashtablet bara är till hälften fullt så testar man att rehasha tablet som en storlek mindre. Antalet element skulle kunna vara någon variabel som ändras för varje insättning och varje radering.

Anledningen till att vi på figur.10 ser att antalet collisions, alltså antalet försök att sätta in ett element x på sin position h(x) misslyckas, för version 2 är större än de andra två versionerna är rehashfunktionen. För de två andra funktionerna kommer varje element endast räknas med en gång, medans vid version 2 så kommer de insatta elementen vid en rehash att möjligtvis räknas flera gånger då det blir nya kollisioner vid rehashandet. Trots denna stora mängd collisions så ser vi på figur.11 att när maximala kollisionskedjan är relativt konstant på version 1, medans när vi närmar oss ett fullt hashtable så stiger den dramatiskt för version 1 och 0, och de följer varandra. Detta då storleken på kollisionskedjan är beroende på storleken på hashtablet och antalet lediga platser, och där kan vi se från figur.8 att version 2 har en loadfactor på ungefär 0,5 medans de andra två kommer ha en loadfactor på närmare 1.
Från figur.12 ser vi att för varierande C värden så följer antalet kollisioner och probes varandra i antalet. Detta då antalet rehashes som genomförs sjunker med stigande värden från C vilket kan ses på figur.5 där storleken på hashtablet sjunker för stigande C. Desto färre rehashes som genomförs, desto fler probes som kommer behöva genomföras när man går igen de upptagna positionerna fram till C. Även för kollisionerna så är det på grund av att storleken på hashtablet inte ökar lika snabbt då elementen ofta kan placeras någonstans.
Vanligtvis när man raderar ett entry ifrån hashTables så kan man inte bara ta bort det, utan istället ”flaggar” man indexet, och/eller fyller detta med ett värde som aldrig kan uppkomma igenom sin hashing metod, detta värde ligger kvar tills man nästa gång rehashar sitt table, då detta utelämnas, här kan man se en anledning till att genomföra rehashar lite oftare om man har mycket temporär data.
Anledningen till varför man inte bara kan ta bort ett element direkt är för att det inte ska bli fel i placering utav objekt som blivigt placerade längre ned i listan som kanske fått sin placering på grund av just det objektet.
Det är upp till programmeraren att bestämma när och hur rehashing sker, det vanligast förekommande är att gå på loadFraction, det vill säga: $\frac{numElements}{listLen}$, och vanligtvis försöker man hålla listan dubbelt så stor som antalet element i den. En annan vanlig standard är att rehasha beroende på antal collisions, eller bara en collision i vissa fall, men mest troligt beroende på en collisionChain. Man vill begränsa hur ofta man rehashar, då detta tar mycket lång tid.