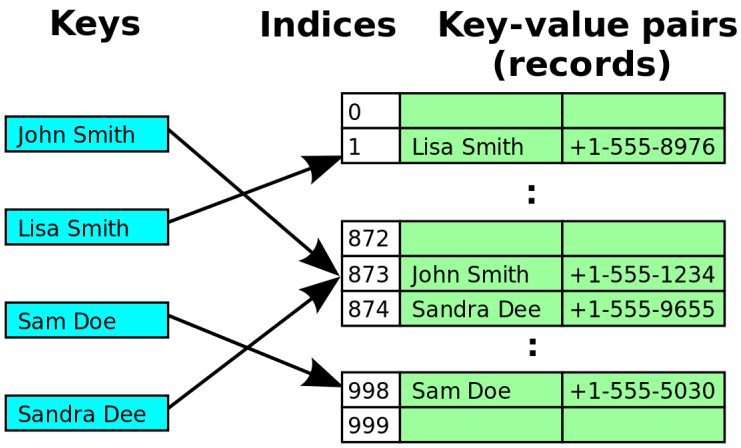
Algoritmer och datastrukturer, laboration 2



Axel Kärnebo, axekrn-7@student.ltu.se
Rickard Bemm, ricbem-8@student.ltu.se
William Gradin, wilgra-8@student.ltu.se

LULEÅ TEKNISKA UNIVERSITET

5 januari 2020

Del I - Implementation och Test

Teori

I första delen av denna laboration har en hashtabell implementerats med två olika probing metoder. Den första var en standard linear probing metod där en hemadress $h(x)$ hittas för varje nyckel x genom att ta $x \mod maxSize$. Om denna hemadress i hashtabellen redan är upptagen så söks närmaste tomma adress fram via linear probing som görs genom att ta $(h(x)+i) \mod maxSize$ där $i = (0, 1, 2 \ldots, maxSize-1)$. När den tomma platsen som är närmast $h(x)$ är funnen så placeras nyckeln x där.

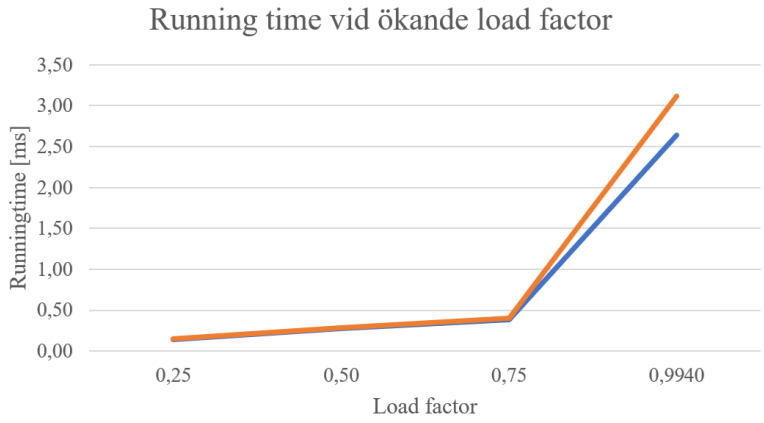
I variationen av linear probing så testas inte bara platserna under hemadressen utan även de som är över. Dvs. denna modifierade version tittar först åt vilket håll som färre nycklar har stoppats in, sedan börjar den leta efter en tom plats mot hållet där färre nycklar har placerats. Den håller reda på två siffror, l_{down} som är antalet nycklar x som stoppats in i hashtabellen med funktionen $(h(x)+i) \mod maxSize, i = (0, 1, 2 \ldots, maxSize-1)$, och l_{up} som är antalet nycklar x som stoppats in i hashtabellen med funktionen $(h(x)-i) \mod maxSize, i = (0, 1, 2 \ldots, maxSize-1)$. På så sätt borde primary clustering inte vara en lika stor faktor, det borde bli färre kollisioner och totala runtimen borde bli lägre.

Resultat

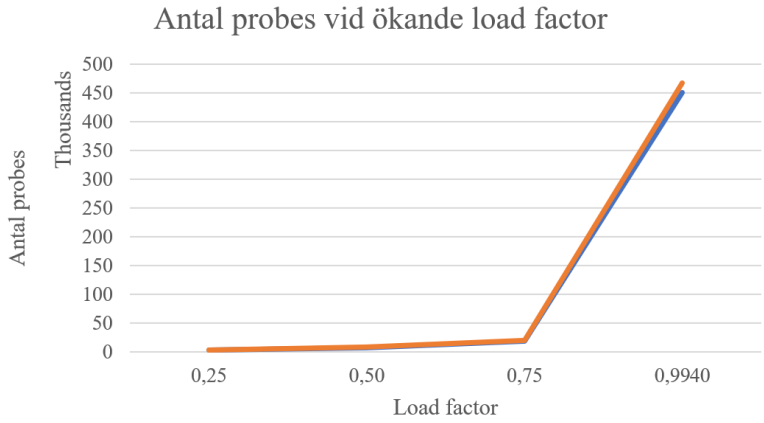
Resultat från tester på implementationen av dessa funktioner står i tabell 1. Under testerna så skapas arrayer av varierande storlek som innehåller ett antal unika värden mellan 1 och 100000. Sedan försöks det att inserta värdena i en hashtable med kapaciteten 10000. Tabell 1 visar snittet av resultaten för 1000 tester avrundat neråt.

Tabell 1: Resultat från utförda mätningar

	Insertions	**2 500**	**5 000**	**7 500**	**10 000**
	Capacity	**10 000**	**10 000**	**10 000**	**10 000**
Regular	**Running time[ms]**	0.1524	0.2839	0.4055	3.1115
	Longest Collision-chain	5	16	88	5 948
	No, Collisions	276	1 996	10 454	440 483
	No, Probes	2 776	6 995	17 953	450 423
	No, Hash calls	5 276	11 995	25 453	460 423
	No, Insertions	2 500	5 000	7 500	9 940
	No, Overflows	0	0	0	59
	Load factor	0.25	0.5	0.75	0.994
Variant	**Running time[ms]**	0.1447	0.2744	0.3862	2.6427
	Longest Collision-chain	5	15	87	6 277
	No, Collisions	270	1 867	9 723	452 273
	No, Probes	2 984	7 937	19 848	467 051
	No, Hash calls	2 984	7 937	19 848	467 051
	No, Insertions	2 500	5 000	7 499	9 956
	No, Overflows	0	0	1	43
	Load factor	0.25	0.5	0.75	0.9956



Figur 1: Graf för ändring av running time vid en load-factor i ett hashTable med konstant antal *slots*, 10 000. Den orange och blåa linjen visar running time för *regular* respektive *variant*.



Figur 2: Graf för ändring i antal probes vid förändring av load factor i en hashTable med konstant antal *slots*, 10 000. Den orange och blåa linjen visar antal probes för *variant* respektive *regular*.

Diskussion

Som vi kan se i tabell 1 så är varianten oftast snabbare än standard versionen. Vi kan klart se hur load factorn påverkar exekveringstiden för funktionerna då tiden för båda ökar väldigt kraftigt när load factorn närmar sig 1. Vi ser som förväntat att varianten oftast hanterar färre kollisioner än standard versionen.

Overflow i tabell 1 innebär att det inte hittades en tom plats för nyckeln i hashtablen. Detta eftersom implementationen gjordes på ett sådant sätt att insättningen stoppas när proben har nått slutet

av hashtablen. Därtill förbättra algoritmerna så att de alltid stoppar in nyckeln om loadfactor är mindre än noll och på så sätt undvika overflow när det egentligen finns plats. Detta då det är en stor nackdel att inte veta om nyckeln stoppas in även fast det finns plats.

Det syns även att vår implementation av standard versionen använder sig av fler hashcalls än variationen, vilket antagligen går att minska.

Del II - Design av algoritm

Den input som algoritmen har att arbeta med är en osorterad lista med n distinkta element. Listan delas upp tills det blir n antal sublistor med 1 element per sublista, sedan mergas dem ihop parvis, tills man når 4 element per sublista, då den från och med den punkten bara tar hänsyn till de 4 minsta elementen i varje merge. När detta är klart tar man ut det högsta av de sorterade talen i listan som skapats, vilket då är det fjärde minsta talet i original-listan.

Algoritmen är baserad på en vanlig mergesort algoritm där inputlistan rekursivt delas upp i halvor och sedan mergeas med de andra uppdelade sublistorna. Detta kan representeras av ett träd som delas upp och sedan sammanfogas igen. Där denna algoritm skiljer sig från en standard mergesort är att den enbart kollar på de fyra minsta elementen när den mergear. Detta innebär att den har lägre kostnad och tidskomplexitet.

Då uppdelningen av listan är konstant så är den även försumbar när tidskomplexiteten söks. Det som kan påverka algoritmens exekveringstid är rekursionen och mergening. Vid en vanlig mergesort så är det relativt enkelt att räkna ut dess tidskomplexitet, då man hittar denna genom att multiplicera antalet merge-nivåer, $\log n$, med antalet element som mergeas varje nivå. Detta antal element är för konstant för mergesort, då alla element sammanfogas med andra element i någon lista. I denna algoritm så förändras denna mängd element. Då basfallet nås för denna algoritm så kommer alla element mergeas två och två till $\frac{n}{2}$ olika listor. Antalet element som mergas vid basfallet är då n.

För nästa nivå så mergas dessa listor med två element ihop till listor med fyra element och antalet element som mergeas är även då n. Dock för nästa nivå kommer antalet element som mergeas att förändras. När elementen är uppdelade i listor med storlek fyra så kommer enbart de fyra minsta elementen inom de två listor som jämförs att sorteras och föras vidare. Detta innebär att mängden element som mergeas på den nivån är $\frac{n}{2}$ och därefter minskas med hälften för varje ny nivå. Detta kan representeras som summan i ekvation 1.

$$2n + \sum_{i=1}^{\log n} \frac{n}{2^i} = 2n + n - 1 = 3n - 1 \implies O(n) \tag{1}$$

Denna summa kan då skrivas om till en enklare ekvation som också syns i ekvation 1, och där går det att hitta tidskomplexiteten för algoritmen som blir $O(n)$.

Det går även att kalkylera den totala mängd jämförelser som algoritmen utför. För att göra detta så kan vi tänka oss att algoritmen gör lika många jämförelser för varje merge, alltså fyra per merge. Totala antalet merges är $n - 1$ och då fås antalet jämförelser till $4n - 4$. Problemet med detta är att denna algoritm inte utför lika många jämförelser vid varje merge. I första och andra merge nivån utförs en respektive tre jämförelser så då måste de överflödiga jämförelsena dras bort. Under

första nivån så utförs $\frac{n}{2}$ merges och vid varje merge sker en jämförelse, under andra nivån så utförs $\frac{n}{4}$ merges och vid varje merge sker tre jämförelser. Därav måste det dras bort $3\frac{n}{2}$ och $\frac{n}{4}$ från det tidigare totala mängden jämförelser. Det ger oss svaret på totala antalet jämförelser, som syns i ekvation 2.

$$4n - 4 - 3 \cdot \frac{n}{2} - \frac{n}{4} = 9 \cdot \frac{n}{4} - 4 \tag{2}$$

Korrekthet

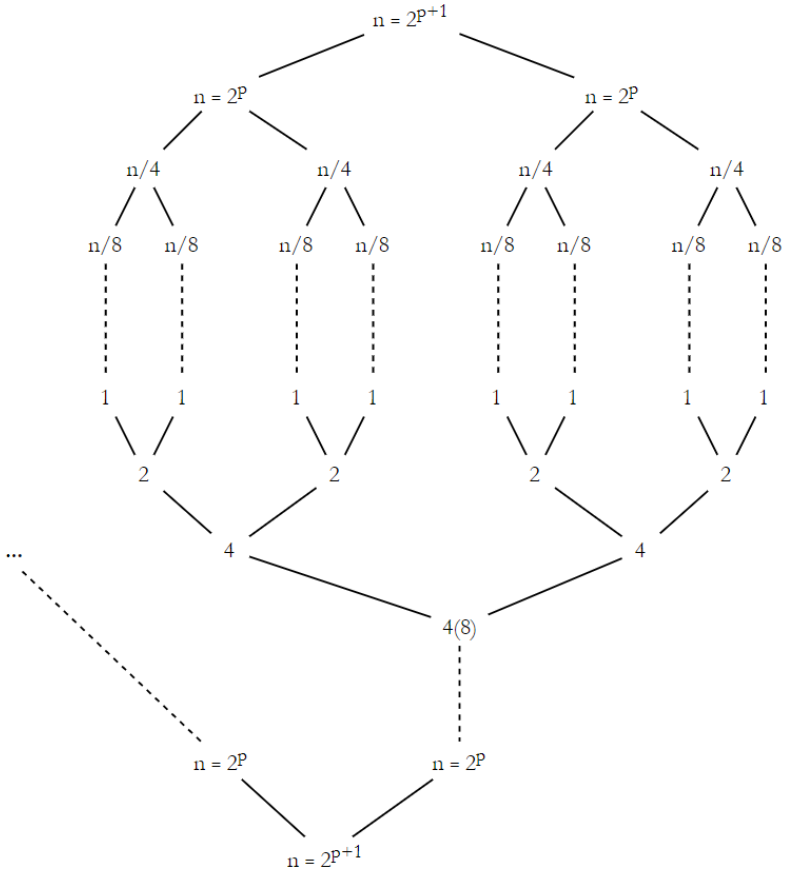
Basfall Då antalet element i input listan är $n = 2^k, k = 2$ så sorteras listan enligt vanlig mergesort och mergesort algoritmen antas alltid ge tillbaka rätt sortering av inputlistan. Alltså blir outputen då $n = 2^k, k = 2$, det sista elementet i den sorterad listan vilket är det fjärde minsta elementet.

Antagande Anta att algoritmen ger tillbaka rätt output då antalet element är $n = 2^k, k = p$.

k=p+1 Då antalet element är $n = 2^{p+1}$ så delas listan upp i två sublistor, det vill säga att antalet element i dessa två sublistor är

$$\frac{n}{2} = \frac{2^{p+1}}{2} = 2^p.$$

Dessa sublistor slås sedan ihop för att bilda svaret då $n = 2^{p+1}$. Eftersom antagandet att $n = 2^p$ ger korrekt output har gjorts så måste även algoritmen ge rätt output då $n = 2^{p+1}$. Figur visar hur detta fungerar.



Figur 3: En graf som visar hur listor delas upp och mergas i algoritmen.