

## Øving 5

### Deloppgave 1:

#### Kjøring

Programmet kjører automatisk og leser filen som er oppgitt i oppgaven fra nettet. Om man heller vil kjøre en egen fil kan man kompilere programmet og sette filen som args[0].

#### Løsning

Kollisjoner blir håndtert ved bruk av en lenket liste gjennom bruk av klassen «Node». Når en Node hasher til samme verdi som en allerede eksisterende Node, blir den eksisterende noden satt som «next» for den nye noden, mens den nye noden blir plassert i listen.

#### Utskrift oppgave 1:

```
Collision between Node:Frederik Bache Ruud and Node:Ari Hanan at index 48
Collision between Node:Håkon Sprli and Node:Martin Clementz at index 127
Collision between Node:Aurora Schmidt-Ersklen Vollvik and Node:Jeffrey Yaw Annor Tabiri at index 51
Collision between Node:Henrik Werner Lervåg and Node:Joachim Grisen Westgaard at index 113
Collision between Node:Gia Hy Nguyen and Node:Adele Iran Westrum Kjølstad at index 81
Collision between Node:Håkon Fredrik Fjellanger and Node:Emil Skogheim at index 26
Collision between Node:Oscar Stentun Stadskleiv and Node:Sondre Fjellving Andersen at index 160
Collision between Node:Elias Tvenning Trana and Node:Harry Linrui Xu at index 25
Collision between Node:Birtha Emilie Christiansen and Node:Kristians Janis Matrevics at index 83
Collision between Node:Jostein Johansen Aune and Node:Ramin Forouzandehjoo Samavat at index 138
Collision between Node:Erik Sandvik and Node:Erik Turmo Nordsether at index 72
Collision between Node:Sebastian Wessel and Node:Martin Sannes Hviistendahl at index 80
Collision between Node:Ina Martini and Node:Helle Vosnik Rosellind at index 105
Collision between Node:Andrea Amundsen and Node:Elias Tvenning Trana at index 25
Collision between Node:Jens Christian Aarnestad and Node:Jostein Johansen Aune at index 138
Collision between Node:Jens Martin Jahle and Node:Hallvard Torsvik Bamrud at index 153
Collision between Node:Emil Johnsen and Node:Håkon Rene Billingsstad at index 44
Collision between Node:Edvin Amot Stava and Node:Oda Libak at index 165
Collision between Node:Knut Skoe and Node:Vemund Ellingsson Ape at index 38
Collision between Node:Vilde Min Vikan and Node:Tor Håkon Reyes Aasebø at index 143
Collision between Node:Magnus Gjerstad and Node:Sondre Adrian Øksvik at index 134
Collision between Node:Steinar Nilsskog and Node:Eline Evje at index 1
Collision between Node:Torbjørn Antonsen and Node:Valdemar Åstorp Beere at index 23
Collision between Node:Maria Elizabeth Pauna Lane and Node:Oscar Stentun Stadskleiv at index 160
Collision between Node:Bragge Tiller Naustan and Node:Henrik Werner Lervåg at index 113
Collision between Node:Ortve Christine Wella Aune and Node:Jens Christian Aarnestad at index 138
Collision between Node:Henrik Tekle Sandok and Node:Eric Blaszcak-Stie at index 88
Collision between Node:Magnus Grini and Node:Anders Emil Bergan at index 102
Collision between Node:Frisk Balder Ormestad Larsen and Node:Snorre Stenshaug Roe at index 87
Collision between Node:Henrik Dybdahl Berg and Node:Madeleine Stenberg Jonassen at index 7
Collision between Node:Agnethe Kval-Engstad and Node:Sigrid Rønnestad at index 149
Collision between Node:Julia Vik Remy and Node:Maria Elizabeth Pauna Lane at index 160
Collision between Node:Henrik Tefre and Node:Miroja Sivachandran at index 136
Collision between Node:Olav Skjolen Asprem and Node:Bragge Tiller Naustan at index 113
Collision between Node:Kenny Bach Phong Tran and Node:Vegard Johnsen at index 56
Collision between Node:Kristina Ødegård and Node:Svein Kåre Særestad at index 130
Collision between Node:Nicolai Forsberg Sommerfelt and Node:Erik Støwer at index 82
Collision between Node:Jacob Forsdahl Iqbal and Node:Martin Nordli Almenningen at index 12
Collision between Node:Henrik Møen Vadet and Node:Markus Hysing Jessund at index 107
Collision between Node:Christian Stensøe and Node:Henrik Tekle Sandok at index 88
Collision between Node:Sandor Kvenild and Node:Knut Skoe at index 38
Collision between Node:Yahideh Rezaei and Node:Henrik Tobias Fredrikssen at index 150
```

Figur 1: Kollisjoner som oppstår ved kjøring av vedlagt fil.

```
PART 1  
Finds Tobias Skipevåg Oftedal: true  
Finds Ola Nordmann: false  
TableSize: 167  
Total collisions: 42  
Collisions per person: 0.3111111111111111  
Load: 135  
Load-factor: 0.8083832335329342
```

*Figur 2: Oversikt over kollisjoner og load for kjøringen*

## Deloppgave 2:

### Oppgave 1

Valg av tabellstørrelse «m» er påvirket av hashe-funksjonen som brukes i denne oppgaven.

Oppgaven er løst ved bruk av hashfunksjon basert på restdivisjon, noe som gjør at «m» blir ideelt et primtall. Som beskrevet i oppgaven må «m» også være over 10-millioner. Koden under lar m være 1008676.

### Oppgave 2

```
public static Integer[] createRandomIntegerArray(int length) {
    Integer[] randomArray = new Integer[length];

    int a = 1;
    Random random = new Random();
    for (int i = 0; i < length; i++) {
        a += random.nextInt(200);
        randomArray[i] = a;
    }
    Collections.shuffle(Arrays.asList(randomArray));
    return randomArray;
}
```

Figur 3: Oversikt over metode som lager en tabell med m unike tilfeldige tall.

### Oppgave 3

Underliggende bilder beskriver en implementasjon av linærprobing og en dobbelhashing.

Variablen «collision» beskriver hvor mange kollisjoner som kan oppstå når man hopper. Load

```
@Override
public void put(Integer integer) {
    if (tableSize <= load){
        throw new RuntimeException("Full");
    }

    //hash
    int pos = hash(integer);

    while (table[pos] != null) {
        pos = (pos + 1) % tableSize;
        collisions++;
    }
    table[pos] = integer;
}
```

variablen beskriver hvor mange tall som har blitt lagt til i hashtabellen.

Figur 4: Oversikt av hopp ved bruk av linær probing.

```
public void put(Integer integer) {
    if (load >= tableSize){
        throw new RuntimeException("Full");
    }
    for (int i = 0; i < tableSize; ++i) {
        int j = probe(hash(integer), hash2(integer), i, tableSize);
        if (table[j] == null) {
            table[j] = integer;
            load++;
            return;
        }
        collisions++;
    }
}
```

Figur 5: Oversikt av hopp ved bruk av dobbelhashing.

## Oppgave 4

PART 2

Size of table: 10000019 (Prime 10000019 ~ 10 million)

Amount of collisions:

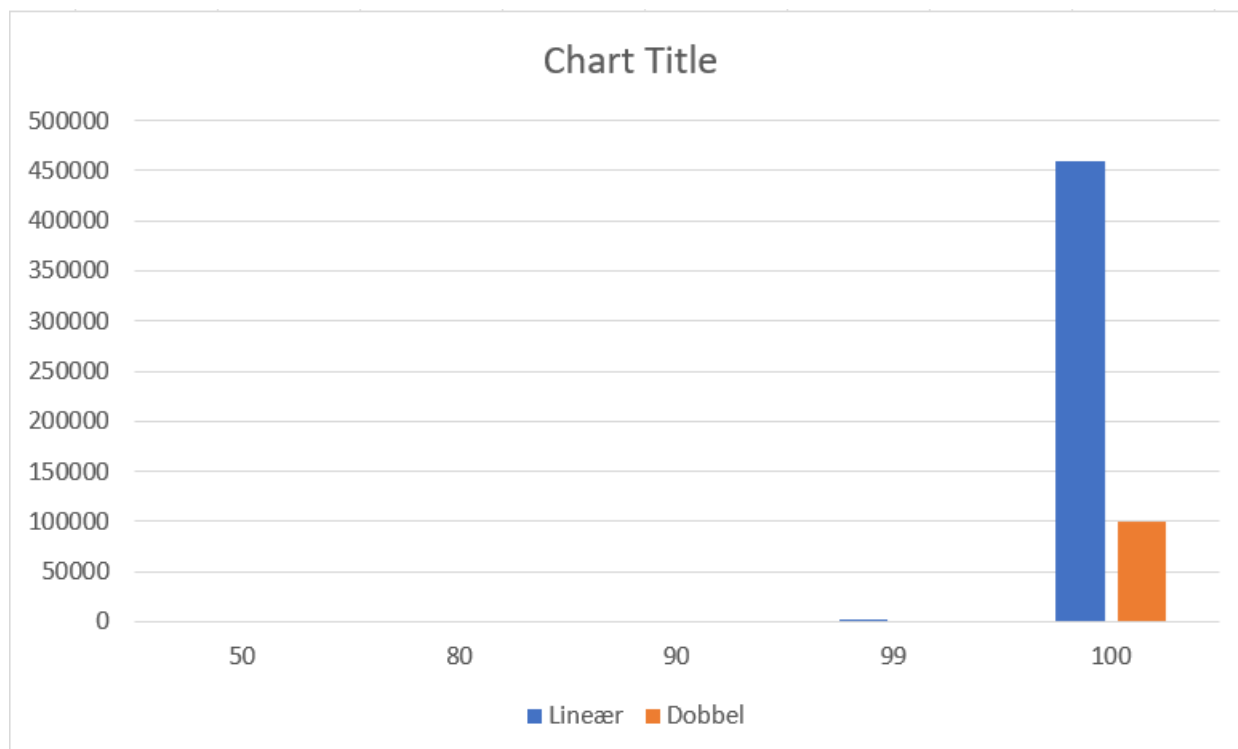
Percentage	Linear Probing	Double Hashing
50.0%	2393610	1965666
80.0%	13700022	8869258
90.0%	28850364	16417696
99.0%	223242727	87463049
100.0%	200622451122	74068074713

Size of table: 10000019 (Prime 10000019 ~ 10 million)

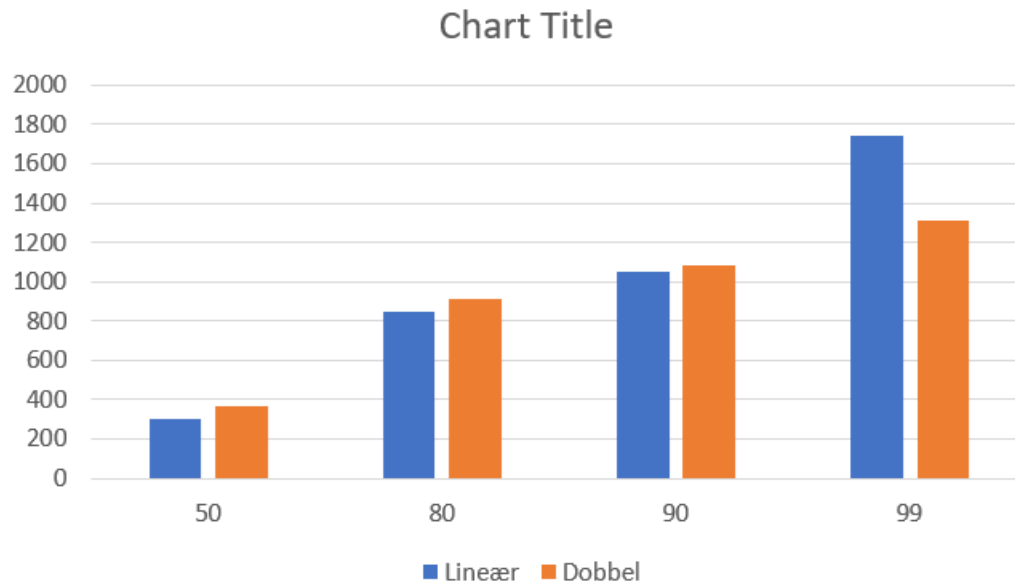
Amount of time in ms:

Percentage	Linear Probing	Double Hashing
50.0%	304	370
80.0%	849	908
90.0%	1053	1080
99.0%	1740	1309
100.0%	459526	99463

Vi ser at bruk av lineære prober generelt sett gir et høyere forbruk av tid, samt et større antall kollisjoner. I eksempelet har til og med den lineære proben gitt et resultat som overstiger Integer.Max\_VALUE for 100% fyllingsgrad, noe som førte til at denne verdien måtte byttes til en long. Det interessante her er hvor mye stigningen er fra bare 99% til 100%. Skulle man prøve å lage en graf av dette kunne det se slik ut:



Grafen viser hvor mye tidsforbruket øker for den lineære proben sammenlignet med dobbel hashing. For lavere fyllingsgrad ser man at økningen i tidsbruk ikke er like stor:



Generelt har dobbel probing lavere tidsbruk når man går mot 100% fyllingsgrad, men forskjellen er ikke i nærheten så stor som for 100%. Dette baseres på at når man f.eks skal sette inn det siste elementet vha lineær probing, vil man i gjennomsnitt bruke  $(\text{tabellstørrelse} / 2)$  prober før man finner rett posisjon, noe som er veldig mye.

## Oppgave 6

- a) Sammenligner man kollisjoner og tid brukt ser man at det er en viss korrelasjon mellom tidsbruk og kollisjoner. Noe som gir mening, ettersom enhver kollisjon vil føre til en ekstra probe, som tar tid. Sammenhengen er ikke lineær, noe man lett ser om man ser på de første verdiene, som har veldig stor forskjell i antall kollisjoner selv om tidtakingen er ganske lik, men dette kan i en viss grad skyldes andre faktorer.
- b) Det er grenser for hvor full en hashtabell bør være for å sikre optimal ytelse og raske operasjoner. Denne grensen avhenger av tabellens lastfaktor, som er forholdet mellom antall elementer i tabellen og tilgjengelige plasser i den. Ideelt sett ønsker vi å holde lastfaktoren lav nok til å unngå mange kollisjoner, samtidig som man utnytter tabellens

plass. Hvis lastfaktoren blir for høy, fører det til flere kollisjoner, noe som negativt påvirker kjøretiden. Generelt er en lastfaktor på rundt 0,7 eller lavere det mest optimale.

- c) Ytelsen til ulike typer hashing, som lineær og dobbel hashing, kan variere avhengig av faktorer som størrelsen på hashtabellen og lastfaktoren. Lineær probing gir best ytelse ved lav fyllingsgrad, men ved høy fyllingsgrad kan det oppstå en betydelig økning i antall kollisjoner, noe som fører til redusert ytelse. Dobbelt hashing anvender en annen hashfunksjon enn lineær probing, og får dermed en mer jevnere fordeling av elementer. Dette fører til bedre ytelse enn lineær probing ved høyere fyllingsgrad.

Gjennomsnittelig tidsbruk for innsetning av et nytt tall i en hash-tabell er begrenset av  $\frac{1}{1-a}$  der  $a$  betegner lastfaktoren. I og med at hashfunksjonene er  $O(1)$ , avhenger de fremdeles på mengden kollisjoner som oppstår. Forskjell på tid ved kjøring er grunnet mindre probesekvenser i dobbel hashfunksjonen.