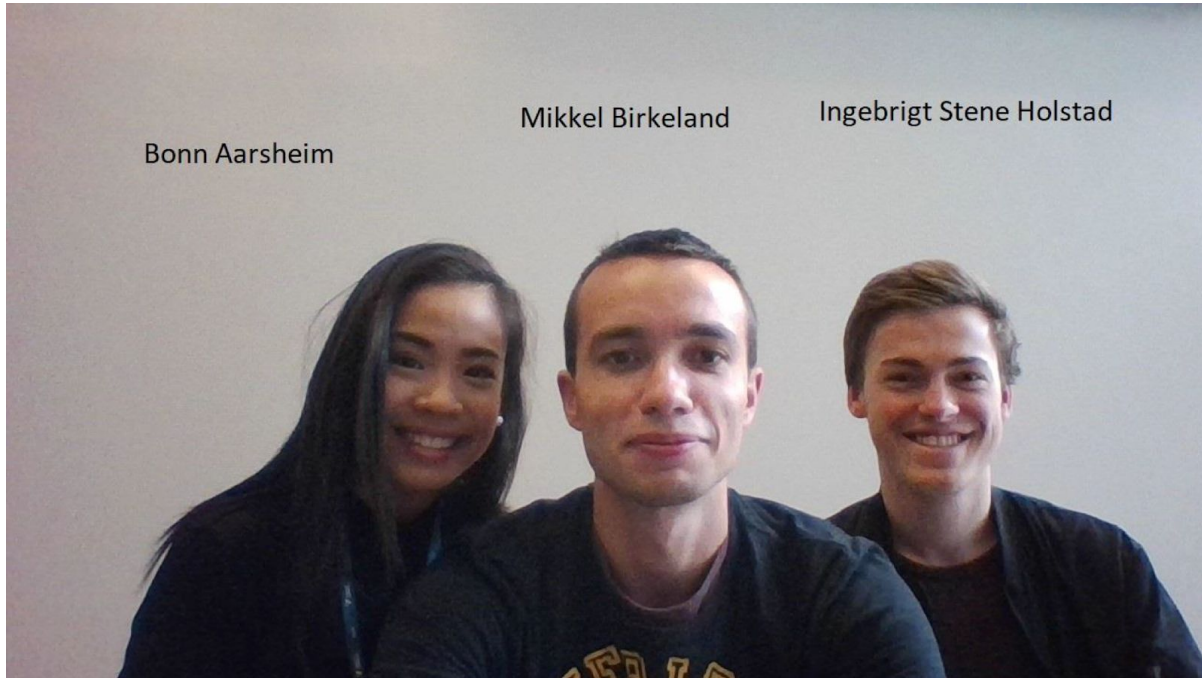


DAT102 ALGORITMER OG DATASTRUKTURER

Vår 2018 Oblig 5, øving 8



Oppgave 1)

a)

Et binærtre er en struktur hvor hver node har maksimalt to barn/undernoder. Disse er ofte definert som "høyre" og "venstre".

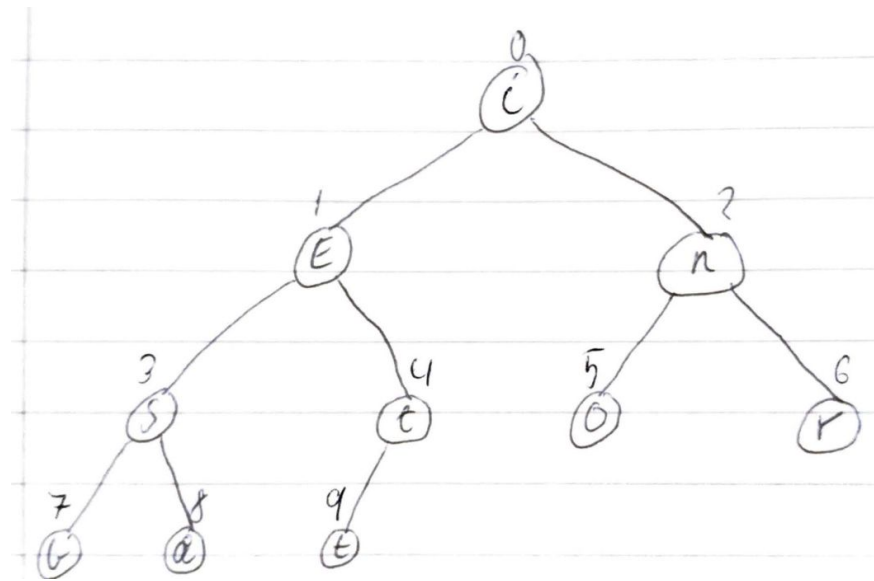
Høyden til et slikt binærtre defineres ved den lengste stien du kan ta fra roten av treet til et blad.

Et fullt binærtre er når alle noder har 2 eller 0 barn.

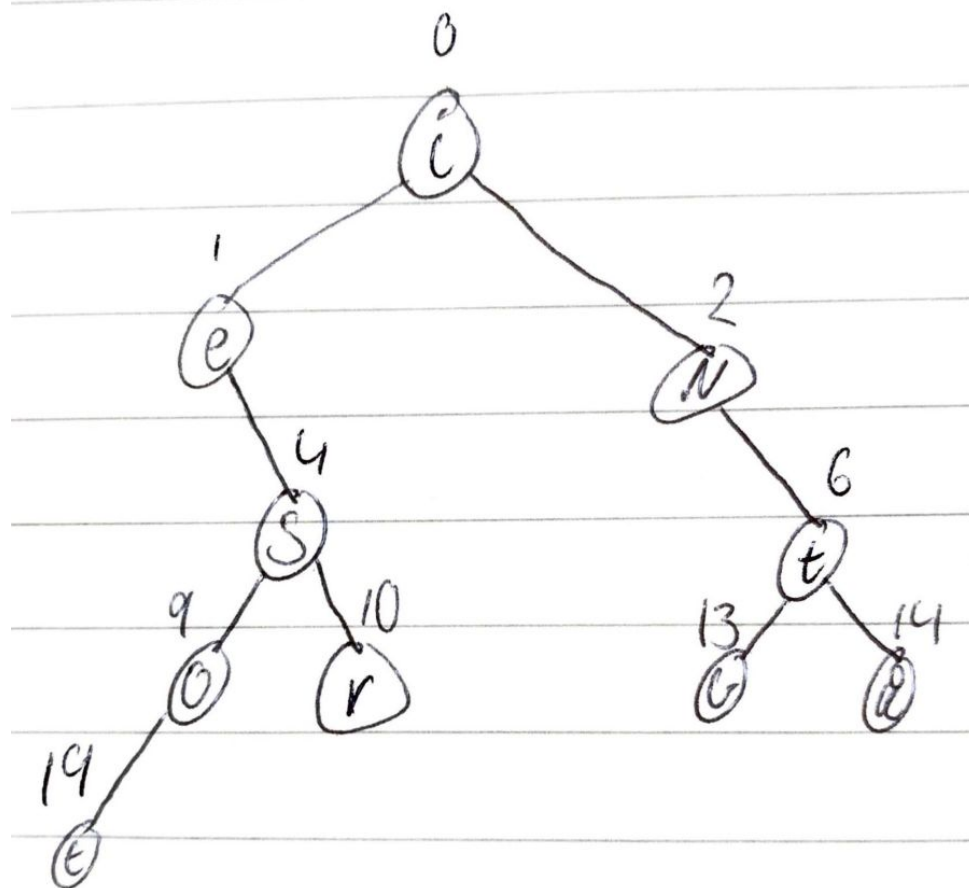
I et komplett tre vil alle noder på ethvert nivå utenom det siste være helt fulle. Alle noder på siste nivå må være plassert så langt til venstre som mulig.

b)

i)



ii)



c)

i) i-E-s-b-å-t-t-n-O-r

ii) i-e-S-o-t-r-N-t-b-å

ii) i) b-S-å-E-t-t-i-o-N-r

ii) t-o-S-r-e-i-b-t-å-n

iii) i) b-å-S-t-t-E-o-r-n-i

ii) t-o-r-S-e-b-å-t-N-i

iv) i) b-å-t-S-t-O-r-E-n-i

ii) t-O-r-b-å-S-t-e-N-i

Oppgave 2

a)

```
public int hentHoyde(BinaerTreeNode<T> node) {  
    if (node == null) {  
        return -1;  
    } else {  
        int vDybde = hentHoyde(node.getVenstre());  
        int hDybde = hentHoyde(node.getHoyre());  
  
        if (vDybde > hDybde)  
            return (vDybde + 1);  
        else  
            return (hDybde + 1);  
    }  
}
```

b)

```
Dette treet har 1024 noder
Maksimal teoretiske høyde blir 1023
Minimal teoretisk høyde blir 10.0
Dette treet har 1024 noder
Maksimal teoretiske høyde blir 1023
Minimal teoretisk høyde blir 10.0
Dette treet har 1024 noder
Maksimal teoretiske høyde blir 1023
Minimal teoretisk høyde blir 10.0
Dette treet har 1024 noder
Maksimal teoretiske høyde blir 1023
Minimal teoretisk høyde blir 10.0

Største høyde i løpet av kjøringen var 63
Minste høyde i løpet av kjøringen var 47
Gjennomsnittshøyde for hele kjøringen var 52
Treet inneholder verdier fra 0 til 30
```

c)

Snitt høyde for kjøringen med 1024 noder, hvor tallene kan være fra 0 – 30, ble 52

$$H = C * \log_2(1024)$$

$$52 = C * \log_2(1024)$$

$$C = 52 / \log_2(1024)$$

$$C = 5.2$$

Regner ut teoretisk høyde for $n = 4096$ med $C = 5.2$

$$H = 5.2 * \log_2(4096)$$

$$H = 62.4$$

Om vi lar programmet regne ut snitt høyde får vi 165

```

Dette treet har 4096 noder
Maksimal teoretiske høyde blir 4095
Minimal teoretisk høyde blir 12.0
Dette treet har 4096 noder
Maksimal teoretiske høyde blir 4095
Minimal teoretisk høyde blir 12.0

Største høyde i løpet av kjøringen var 181
Minste høyde i løpet av kjøringen var 154
Gjennomsnittshøyde for hele kjøringen var 165
Treet inneholder verdier fra 0 til 30
Test tre skal nå ha høyde -1, kalkulert høyde er -1
Test tre skal nå ha høyde 0, kalkulert høyde er 0

```

Dette stemmer ikke så godt overens med den teoretiske beregningen som fikk 62.4.

Om vi lar programmet velge verdier fra 0-5000 derimot, får vi en C på 2.1.

Da blir den teoretiske høydeutregningen for n=4096 lik 25.2.

Dette ser vi stemmer svært godt overens med det programmet regner ut som snitthøyde:

```

Maksimal teoretiske høyde blir 4095
Minimal teoretisk høyde blir 12.0

Største høyde i løpet av kjøringen var 33
Minste høyde i løpet av kjøringen var 23
Gjennomsnittshøyde for hele kjøringen var 26
Treet inneholder verdier fra 0 til 5000
Test tre skal nå ha høyde -1, kalkulert høyde er -1
Test tre skal nå ha høyde 0, kalkulert høyde er 0

```

d)

i) og ii)

```

/**
 * Tester finn
 *
 */
@Test
public final void erElementIBSTre() {
    /* Her legger du inn e0...e6 i treet i en vilkårlig rekkefølge.
     * Etterpå sjekker du om elementene fins og til slutt sjekker du
     * at e7 ikke fins
     */
    bs.leggTil(e0);
    bs.leggTil(e1);
    bs.leggTil(e2);
}

```

```

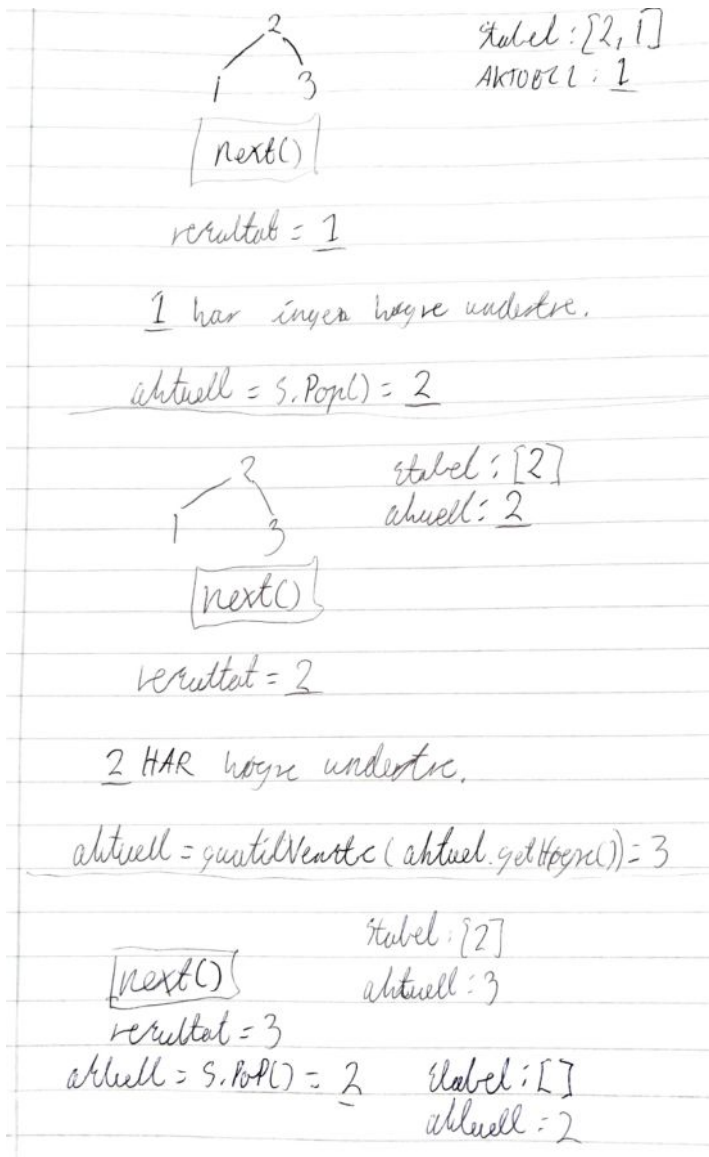
        bs.leggTil(e3);
        bs.leggTil(e4);
        bs.leggTil(e5);
        bs.leggTil(e6);
        assertEquals(e0, bs.finn(e0));
        assertEquals(e1, bs.finn(e1));
        assertEquals(e2, bs.finn(e2));
        assertEquals(e3, bs.finn(e3));
        assertEquals(e4, bs.finn(e4));
        assertEquals(e5, bs.finn(e5));
        assertEquals(e6, bs.finn(e6));
        assertFalse(e7.equals(bs.finn(e7)));
    }

    /**
     *1.  Tester ordning  ved å legge til elementer og fjerne minste
     *
     */
    @Test
    public final void erBSTreOrdnet() {
        /* Her legges du først inn e0...e6 i en vilkårlig rekkefølge
         * og så fjernes du minste hele tiden
         */
        bs.leggTil(e0);
        bs.leggTil(e1);
        bs.leggTil(e6);
        bs.leggTil(e4);
        bs.leggTil(e3);
        bs.leggTil(e5);
        bs.leggTil(e2);
        assertEquals(e0, bs.fjernMin());
        assertEquals(e1, bs.fjernMin());
        assertEquals(e2, bs.fjernMin());
        assertEquals(e3, bs.fjernMin());
        assertEquals(e4, bs.fjernMin());
        assertEquals(e5, bs.fjernMin());
        assertEquals(e6, bs.fjernMin());
    }

```



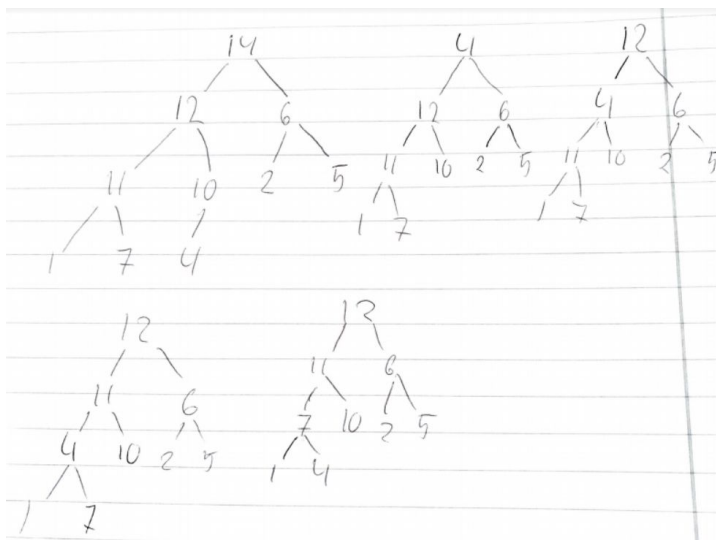
iii)



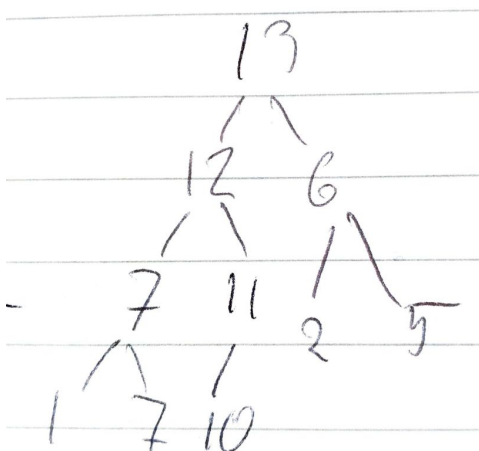
Oppgave 3.

- a) Heap/Haug er et komplett binært tre der hvert element er koblet til "barn". Elementet skal ha en verdi lik eller mindre en både barnet til venstre og høyre. Dette kalles en "minheap". Som sikrer at roten til treet vil være det minste i datamengden. Det finnes også "maxheap", som er det motsatte av "minheap". Dette sikrer at roten til treet vil være det største i datamengden.
- b) Både B og C danner makshauger, siden roten i ethvert tre/undertre som kan dannes, er det største elementet.

c) i)



ii)



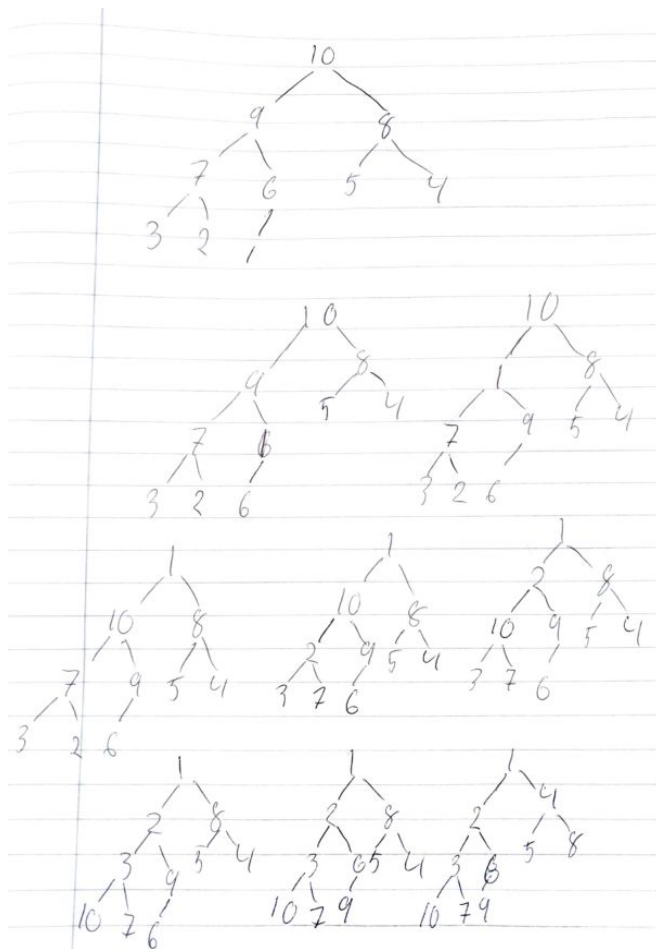
d)

Du kan bruke en makshaug til å sortere elementer ved å først lage en makshaug, og deretter fjerne elementene fra roten. Du vil ha alltid fjerne det største elementet i haugen, så du ender opp med en liste sortert fra høyeste til laveste.

e)

I en haug med n noder så er det i alt $(n/2)-1$ interne noder som skal sammenlignes. Høyden på haugen er $\log_2 n$ da blir antall sammenligninger $\log_2 n * n/2$ som blir $O(n \log_2 n)$.

f)



g)

```
private void reparerOpp() {
    int rettPlass = antall - 1;
    T tmp = data[rettPlass];
    int forelder = (rettPlass - 1) / 2;

    while (rettPlass > 0 && tmp.compareTo(data[forelder]) < 0) {
        data[rettPlass] = data[forelder];
        rettPlass = forelder;
        forelder = (rettPlass - 1) / 2;
    }
    data[rettPlass] = tmp;
}
```

```
terminated> Kjentnag [Java Application] C:\Program F...
Verdiene i tabellen er nå:
1 10 2 18 54 33 30 300 200 100

Haugen i sortert rekkefølge:
1 2 10 18 30 33 54 100 200 300
```

Oppgave 4

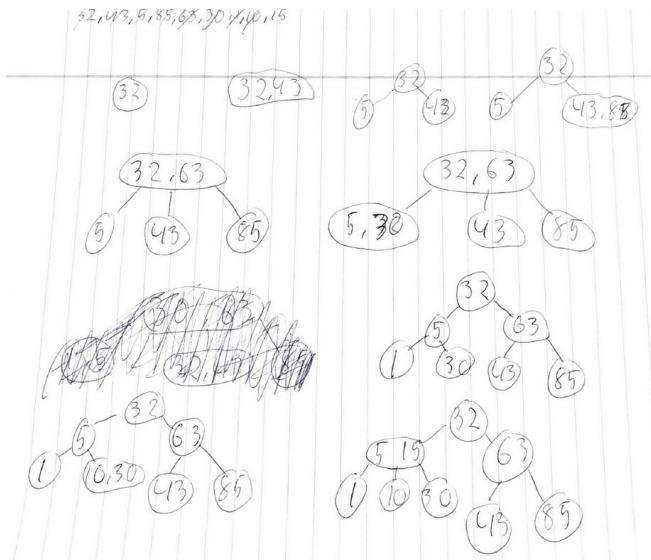
a) 2-3 tre er en “multi-way” søketre, der hver node har to eller tre “barn” og kan ha 2 eller tre elementer. Slike søketrær går raskere å søke i, siden høyden blir mindre. Så det brukes ofte til indeksering.

b) 2-node inneholder ett element, subtreeet til venstre inneholder element som er mindre eller lik dette elementet, mens subtreeet til høyre inneholder element som er større eller lik dette elementet.

3-node inneholder to element der det ene elementet er satt til å være det største og det andre til det minste. Den har enten ingen - eller tre “barn”. Hvis en 3-node har tre barn, vil subtreeet til venstre inneholde element som er mindre enn det minste elementet, og subtreeet til høyre vil inneholde element som er større eller lik elementet som er satt til å være størst. Det subtreeet som er i midten vil inneholde element som er større eller lik det minste elementet og element som er mindre enn elementet som er satt til å være størst.

For å søke etter elementet 42 i treet, går vi først til venstre der det står 30 fordi 42 er mindre enn 45. Her går vi videre til høyre fordi vi vet 42 er høyere enn 30 og må dermed ligge en plass til høyre. Her prøver vi videre å gå videre fra 40, men siden det ikke finnes en node fra 40, vet vi at 42 ikke finnes i treet.

- c) Å søke etter et element i en 2-3 tree er veldig likt søking i binære trær. Siden dataene i hver node er sortert, vil søket gå direkte til det riktige subtreeet og til slutt til den riktige noden som inneholder elementet man søker etter.



Oppgave 5

a)Bredde-først:

Kø	Besøkt
c	c
ef	ce
fbdf	cef
bdf	cefb
df	cefb
fa	cefb
a	cefbda
TOM	

b)

Dybde-først:

Stabel	Besøkt
c	
ef	c
bdf	ce
df	ceb
af	cebd
f	cebda
TOM	cebdaf

c)

$V = \{a, b, c, d, e, f\}$

$E = \{(a,d), (b,d), (b,e), (b,f), (c,e), (c,f), (d,e), (e,f)\}$

Nabolister:

(a, d), (b, d), (b, e), (b, f), (c, e), (c, f), (d, a), (d, b), (d, e), (e, b), (e, c), (e, d), (e, f), (f, b), (f, c), (f, e)

Nabomatrise:

	a	b	c	d	e	f
a	0	0	0	1	0	0
b	0	0	0	1	1	1
c	0	0	0	0	1	1
d	1	1	0	0	1	0
e	0	1	1	1	0	1
f	0	1	1	0	1	0

Oppgave 6

- a) I hashing er element lagret i en hashtabell, lokasjonen er bestemt av en hashfunksjon. Vi kan ta lagring av navn som et eksempel. Vi starter da med å lage en hashfunksjon *hash*, som bruker første bokstaven i navnet til å plassere navnet på riktig posisjon i tabellen. Dersom vi velger å bruke det engelske alfabetet, bør hash ha en verdimengde på 26.

Vi plasserer navnet i den posisjonen første bokstaven i navnet hører hjemme. Navnet "Anna" skal på posisjon 0, "Beate" skal på posisjon 1 og "Ida" skal på posisjon nr. 8. Når vi skal finne et navn, søker vi på den posisjonen bokstaven starter med. Dersom posisjonen er tom, vet vi at navnet ikke er i tabellen. Dersom posisjonen ikke er tom, men inneholder et navn som ikke er lik det vi ønsker å finne, går vi videre til neste posisjon. Er posisjonen tom, vet vi at navnet ikke eksisterer i tabellen.

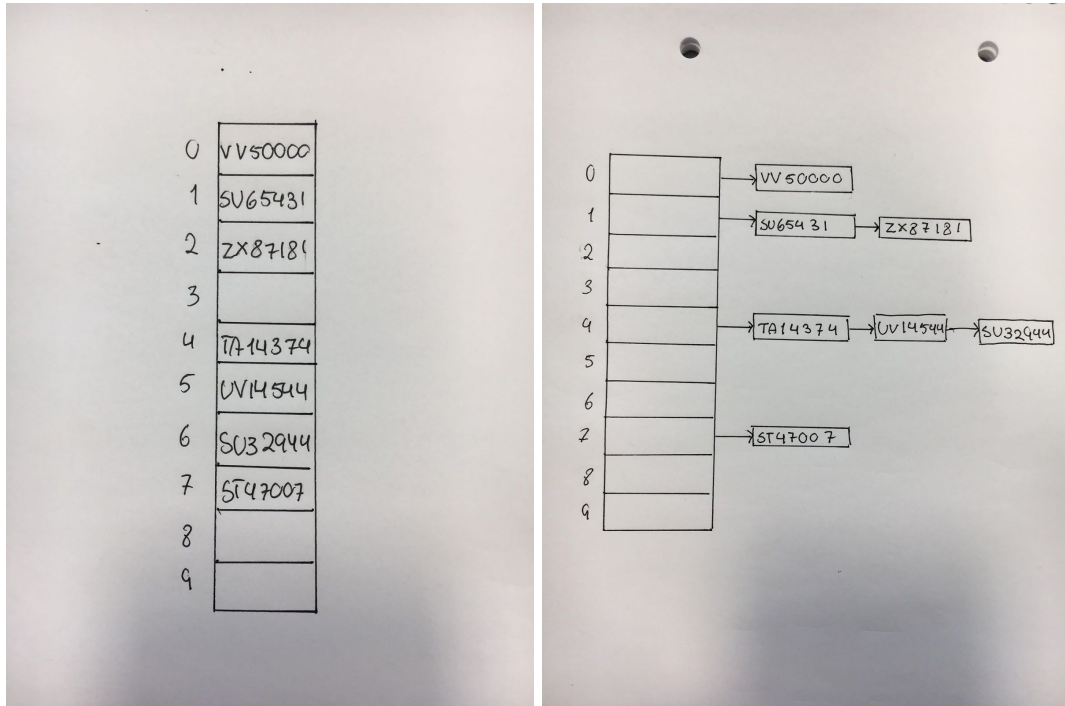
En hash-funksjon er en funksjon som plasserer elementer i hashtabellen. En god hash-funksjon vil fordele elementene uniformt over hele tabellen, og en perfekt hashfunksjon gir ingen kollisjoner.

En kollisjon oppstår når to nøkler kobles til samme posisjon. Når en kollisjon oppstår, vil vi prøve å finne en ledig plass til elementet, og vi leter med en fast "hoppelengde" på 1. Når vi finner en ledig plass, plasserer vi elementet der. Dersom det skjer mange kollisjoner i dette intervallet, vil det danne en opphoping av data. Dette fører til at vi må i gjennomsnitt gjøre mange "hopp" for å finne elementet vi leter etter senere.

Kriterier for en god hash-funksjon:

- Beregning av hashverdien må være rask, altså $O(1)$
- Spre verdiene jevnt i hashtabellen
- Minimerer antall kollisjoner

b)



c)

$$\begin{aligned}
 ab &= a=97 \quad b=98 \quad ; \quad 97 \cdot 31^{(2-1)} + 98 \cdot 31^{(2-2)} = \underline{3105} \\
 123 &= 1=49, 2=50, 3=51 \quad ; \quad 49 \cdot 31^{(3-1)} + 50 \cdot 31^{(3-2)} + 51 \cdot 31^{(3-3)} = \underline{\underline{618690}}
 \end{aligned}$$

```

public class ManuellHash {
    public static void main(String[] args) {
        // char a = 'a';
        // char b = 'b';
        // char one = '1';
        // char two = '2';
        // char three = '3';
        String sEn = "ab";
        String sTwo = "123";
        int fasitEn = 3105;
    }
}

```

```

        int fasitTwo = 48690;
        System.out.println("Streng " + sEn + " blir med hashCode " +
sEn.hashCode() + " og fasit er " + fasitEn);
        System.out.println("Streng " + sTwo + " blir med hashCode " +
sTwo.hashCode() + " og fasit er " + fasitTwo);

    }
}

```

```

Console
<terminated> ManuellHash [Java Application] C:\Program Files\Java\jre1.8.0_151\
Streng ab blir med hashCode 3105 og fasit er 3105
Streng 123 blir med hashCode 48690 og fasit er 48690

```

d)

Utskriften bruker hashCode() metoden som tilhører klassen Object. Denne bryr seg fint lite om hva objektvariablene er når den hasher objektet. Dermed vil objekt a alltid få samme hashkode uansett om navnet er Ole eller Tina osv.

For å få dem til å blir lik må vi lage en ny hashCode() metode i klassen Student.

```

public class Student {
    private int snr;
    private String navn;

    public Student(int snr, String navn) {
        this.snr = snr;
        this.navn = navn;
    }

    @Override
    public int hashCode() {
        return Integer.toString(snr).hashCode();
    }
}

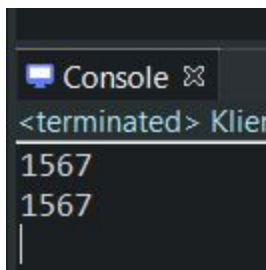
```

```

public class KlientStududent {
    public static void main(String[] args) {
        Student a = new Student(10, "Ole");
        Student b = new Student(10, "Ole");

        System.out.println(a.hashCode());
        System.out.println(b.hashCode());
    }
}

```



e)

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Random;

public class TestSokTid {
    public static void main(String[] args) {
        int antElement = 100000;
        int omraade = 999999;
        ArrayList<Integer> liste = new ArrayList<Integer>(antElement);
        HashSet<Integer> hSet = new HashSet<Integer>(antElement);
        Random random = new Random();

        int tall = 376; // Her kan vi bruke eit vilkårleg tal
        for (int i = 0; i < antElement; i++) {
            // legg tall til i HashSet og tabell
            tall = (tall + 45713) % 1000000; // Sjå nedanfor om 45713
            liste.add(tall);
            hSet.add(tall);
        }

        Collections.sort(liste);
    }
}

```



```

ArrayList<Integer> sokListe = new ArrayList<Integer>(10000);
for (int i = 0; i < 10000; i++) {
    int num = random.nextInt(omraade);
    sokListe.add(num);
}

System.out.println("Søker i hashSet..");
int funnetHash = 0;
long timeStart = System.nanoTime();

for (Integer i : sokListe) {
    if (hSet.contains(i)) {
        funnetHash++;
    }
}
long timeStop = System.nanoTime();

System.out.println(
    "Det tok " + (((double) timeStop - timeStart) / 1000000)
+ "ms, den fant " + funnetHash + " tall");

System.out.println("\nSøker med binærsøk..");
int funnetList = 0;

timeStart = System.nanoTime();


for (Integer i : sokListe) {
    int funnet = Collections.binarySearch(liste, i);
    if (funnet >= 0) {
        funnetList++;
    }
}
timeStop = System.nanoTime();

System.out.println(
    "Det tok " + (((double) timeStop - timeStart) / 1000000)
+ "ms, den fant " + funnetList + " tall");

}

}

```

```
Console 
<terminated> TestSokTid [Java Application] C:\Program File
Søker i hashSet..
Det tok 3.76234ms, den fant 1013 tall

Søker med binærsøk..|
Det tok 15.904399ms, den fant 1013 tall
```

Det er ikke spesielt overraskende at det gikk mye raskere å søke i hashSet. Dette har en tidskompleksitet på $O(1)$. Eneste grunnen til at de var såpass like er at binærsøk også er veldig raskt. Om vi hadde brukt lineaersøk ville det blitt store forskjeller.