Uke 6,
Forelesning 1

- Essensen i faget, matematiske forutsetninger
- Metodekall og rekursjon
- Kombinasjoner og permutasjoner
- Analyse av algoritmer

Uke 1 og Uke 2

- Abstrakte datatyper (ADT'er)
- Lister, stabler og køer

Uke 3 og Uke 4.1

- Generelle trær
- Binære trær
- Binære søketrær

Uke 4.2 Uke 5.1

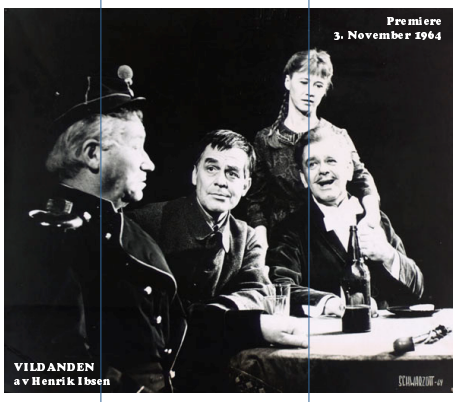
- Oblig. 1 og Dronning konkurransen

Uke 5.2

OVERSIKT – *Uke 6, Forelesning 1 (W6.L1)*

- BINÆRE SØKETRÆR (MAW kapittel 4.3): Vi avslutter Ibsen-eksemplet og binære søketrær.
- Vi skall snakke litt om INVARIANTER
- Vi skal begynne å se på HASHING (MAW kapittel 5)
- Neste gang (W6.L2) avslutter vi hashing ved å se på kollisjonshåndtering, åpen og lukket hashing, rehashing og til slutt utvidbar hashing (extendible hashing).
- Uken deretter (W7.L1) er vi tilbake i skogen og tar opp B-trær
- OBS! Den andre obligatoriske oppgaven er lagt ut!

TEMA: BINÆRE SØKETRÆR



Einar Sissener som Gamle Ekdal
Per Sunderland som Grøgers Werle
Liv Thorsen som Hedvig
Per Aabel som Hjalmar Ekdal

Binære søketrær avsluttes med Vildanden...

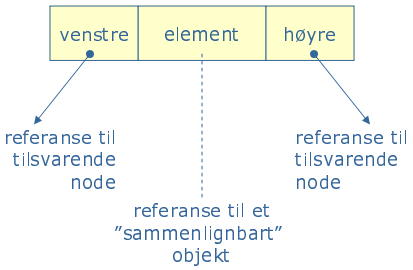
BINÆRE SØKETRÆR – *Et større eksempel #2*

```
import inf110.*;                                  // Vi henter inn IbsenTre og litt til her.
public class Ibsen
{   public static void main(String[] args)        // HOVEDPROGRAM
    {   String ord;
        int antallOrd = 0;

        String sep = ". :,;-?!\"";                // Ordskillere
        Inn innfil = new Inn(args[0]);            // Navnet til innfil er i kommandolinjen
        IbsenTre tre = new IbsenTre();            // IbsenTre er et "søketre"

        innfil.skipSep(sep);                      // Finn ordbegynnelsen etter ordskiller
        while (!innfil.endOfFile())               // Så lenge det ikke er "end-of-file"...
        {   ord = innfil.inString(sep).toLowerCase();  // Les ord og konverter til småbokstaver
            tre.settInn(new IbsenElem(ord));      // Sett ordet i treet
            antallOrd++;                          // Tell opp ("inkrementer") antallet ord
            innfil.skipSep(sep);                  // Hopp over ordskiller og finn begynnelsen
        }                                         //    av neste ord.

        System.out.println("Totalt antall ord: " + antallOrd);
        System.out.println("Antall ulike ord: " + tre.size());

        IbsenFrekTre frekTre = new IbsenFrekTre();
        frekTre.innsetting(tre.getRot());

        frekTre.skrivInnfiks();
    }
}
```


BINÆRE SØKETRÆR – *Et større eksempel #3*

// Vanlig node i binært søketre: Basis node-element

```
class BinNode
{ Comparable element;

  BinNode venstre;
  BinNode hoyre;

  BinNode(Comparable x)
  { element = x;
  }
}
```



BINÆRE SØKETRÆR – *Et større eksempel #4*

```
// Vanlig binært søketre der "noe gjøres" ved like elementer ved oppdater(n, x) nederst:
public class BinSokeTre
{   protected BinNode rot;
    protected int antallNoder = 0;

    // Legg inn en ny node med sammenlignbare elementet x i den ved node n
    private void settInn(Comparable x, BinNode n)
    {   int i = sammenlign(x, n.element);      // Sammenlign elementet x med nodens element
        if (i < 0)                             // Hvis det nye elementet er "mindre"...
        {   if (n.venstre == null)             // Sjekk om venstre subtre er tom...
            {   n.venstre = new BinNode(x);    // Hvis tom, legg in nye elementet der, og
                antallNoder++;                 //    oppdater antallet noder.
            }
            else settInn(x, n.venstre);        // Hvis IKKE tom, legg inn under dens venstre
        }
        else if (i > 0)                        // Hvis det nye elementet er "større"...
        {   if (n.hoyre == null)               // Sjekk om høyre subtre er tom...
            {   n.hoyre = new BinNode(x);      // Hvis tom, legg in nye elementet der, og
                antallNoder++;                 //    oppdater antallet noder.
            }
            else settInn(x, n.hoyre);          // Hvis IKKE tom, legg inn under dens høyre
        }
        else oppdater(n, x);                   // Hvis det nye elementet ikke er hverken
    }                                          //    "mindre" eller "større", gjør noe (oppdater).
    // Flere metoder her...
}
```

Oppgave: Lag en ikke-rekursiv settInn!

BINÆRE SØKETRÆR – *Et større eksempel #5*

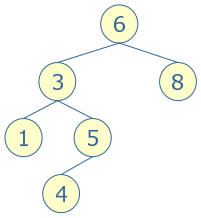
```
public void settInn(Comparable x)    // NB! ≠ settInn(Comparable, BinNode)
{  if (rot == null)                  // Hvis tomt tre...
   {  rot = new BinNode(x);          // Legg new node som rot-noden,
      antallNoder++;                 //    og tell opp antallet noder.
   }
   else settInn(x, rot);             // Hvis IKKE-tomt tre, bruk den andre
}                                    //    settInn(Comparable, BinNode).

protected void oppdater(BinNode n, Comparable x)
{                                    // Default: Ingenting gjøres for like
}                                    //    elementer.

protected int sammenlign(Comparable n1, Comparable n2)
{  return n1.compareTo(n2);          // Bruker metoden "compareTo()" som
}                                    //    er en instans-metode i Comparable
                                     //    klassen (dvs. en metode for
                                     //    klassens objekter (instanser)...
                                     // Returnerer int < 0 for n1 < n2,
                                     //    int > 0 for n1 > n2,
                                     //    og 0 for n1 = n2.
```

BINÆRE SØKETRÆR – *Et større eksempel #6*

```
public void skrivInnfiks()          // Skriver i innfiks rekkefølge.
{  innfiks(rot);                    // Se neste foil!
}

private void innfiks(BinNode n)     // Se neste foil!
{  if (n != null)
   {  innfiks(n.venstre);
      System.out.println(n.element.toString());
      innfiks(n.hoyre);
   }
}

public int size()
{  return antallNoder;
}

public BinNode getRot()
{  return rot;
}
```


BINÆRE SØKETRÆR – *Et større eksempel #7*

```
public void skrivInnfiks()
{  innfiks(rot);
}

private void innfiks(BinNode n)
{  if (n != null)
   {  innfiks(n.venstre);
      System.out.println(n.element.toString());
      innfiks(n.hoyre);
   }
}
```



HÅNDKJØRING (les tallene som noder):

01) innfiks(rot: node-6)
02) node-6 != null; innfiks(6.venstre: 3)
03) 3 != null; innfiks(3.venstre: 1)
04) 1 != null; innfiks(1.venstre: null)
05) null == null; … returnerer til node 1:
06) …println(1.element.toString())
07) innfiks(1.hoyre: null)
08) null == null; … returnerer til node 1:
09) returnerer... til 3
10) …println(3.element.toString())
11) innfiks(3.hoyre: 5)
12) 5 != null; innfiks(5.venstre: 4)
13) 4 != null; innfiks(4.venstre: null)
14) null == null; … returnerer til node 4:
15) …println(4.element.toString())
16) innfiks(4.hoyre: null)
17) null == null; … returnerer… til 4:
18) returnerer... til 5
19) …println(5.element.toString())
20) innfiks(5.hoyre: null)
21) Evt. til 6, skriver den, og så til hoyre
22) Evt. til 8, skriver den og avslutter.

BINÆRE SØKETRÆR – *Et større eksempel #8*

```
// Klasse for å ta vare på ordene og telle opp antall forekomster:
class IbsenElem implements Comparable
{  String ord;
   int antall;

   IbsenElem (String s)
   {  ord = s;
      antall = 1;
   }

   public int compareTo(Object x)
   {  IbsenElem e = (IbsenElem) x;
      return ord.compareTo(e.ord);
   }

   public String toString()
   {  return (ord + " " + antall);
   }
}
```

IbsenElem
ord: String antall: int
compareTo(Object): int toString(): String

BINÆRE SØKETRÆR – *Et større eksempel #9*

```
// Tre sortert på ord:
class IbsenTre extends BinSokeTre
{ protected void oppdater(BinNode n, Comparable e)
   { IbsenElem ie = (IbsenElem) n.element;
     ie.antall++;
   }
}
```

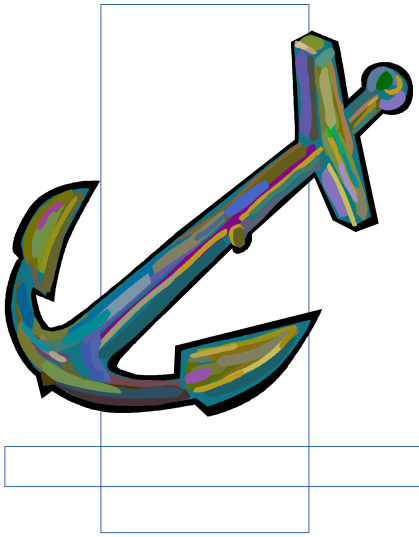
BINÆRE SØKETRÆR – *Et større eksempel #10*

```
// Tre sortert på frekvens:
class IbsenFrekTre extends BinSokeTre
{
    protected int sammenlign(Comparable c1, Comparable c2)
    {   IbsenElem e1 = (IbsenElem) c1;
        IbsenElem e2 = (IbsenElem) c2;
        return e1.antall - e2.antall;
    }

    protected void oppdater(BinNode n, Comparable e)
    {   if (n.venstre == null) n.venstre = new BinNode(e);
        else
        {   BinNode b = new BinNode(e);
            b.venstre = n.venstre;
            n.venstre = b;
        }
        antallNoder++;
    }

    public void innsetting(BinNode n)
    {   if (n != null)
        {   settInn(n.element);
            innsetting(n.venstre);
            innsetting(n.hoyre);
        }
    }
}
```


BINÆRE SØKETRÆR – *Et større eksempel #11, RESULTATER*

De mest brukte ordene (fra UiB):

	Antall	%	Kum.	ORD
1	19506	3.09	3.09	det
2	18241	2.89	5.98	jeg
3	17746	2.81	8.80	og
4	13232	2.10	10.89	i
5	13165	2.09	12.98	er
6	9639	1.53	14.51	at
7	9312	1.48	15.98	du
8	8927	1.42	17.40	ikke
9	8311	1.32	18.72	de
10	7711	1.22	19.94	en
11	7299	1.16	21.10	har
12	7152	1.13	22.23	som
13	6971	1.11	23.34	mig
14	6901	1.09	24.43	for
15	6795	1.08	25.51	til
16	6671	1.06	26.56	så
17	6314	1.00	27.56	med
18	5543	0.88	28.44	han
19	5524	0.88	29.32	den
20	5311	0.84	30.16	på



Litt om invarianter...

To hovedtyper:

- Representasjonsinvarianter:
  Betingelser vi vil skal gjelde for en instans av en ADT.
  Gjelder "hele tiden" etter initialiseringen, bortsett fra når en ADT-operasjon utføres.
- Løkkeinvarianter:
  Utsagn om variabelverdiene som skal gjelde hver gang utførelsen passerer et punkt i løkken (typisk: start/slutt).

To bruksmåter:

- Teknikk for å kunne bevise at et program har visse egenskaper.
- Måte å holde fast på hovedidéen når man skal lage/sette seg inn i et program.

INVARIANTER – *Løkkeinvarianter*

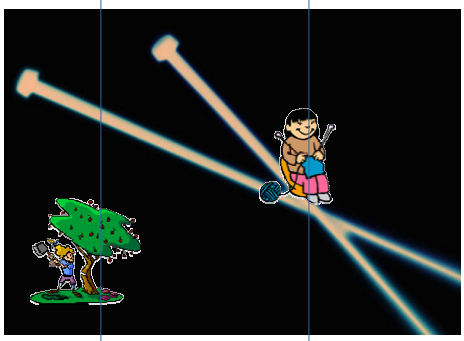
Eksempel: Summering av verdiene i array'en A.

```
int i = -1;
int sum= 0;
// Invarianten skal gjelde her
while (i != n - 1)
{  i++;
   sum += A[i];
   // Invarianten skal gjelde her
}
```

- Hva er en passende invariant: ???

- Bevis for at løkken gjør det den skal:
  - Invarianten gjelder umiddelbart før løkken.
  - Dersom løkken ikke terminerer vil innmaten av løkken "bevare invarianten".
  - Invarianten sammen med betingelsen som gjorde at løkken stoppet gir det ønskede resultatet.

TEMA: HASHING



Hashing :
Chopping with an "hachette"

Hashing :
One type of embroidering

Introduksjon til hashing...

HASHING – *Innhold*

- Hashtabeller
- Hash-funksjoner
- Kollisjonshåndtering
  - Åpen hashing (kap. 5.3)
  - Lukket hashing (kap. 5.4)
- Rehashing (kap. 5.5)

HASHING – *Motivasjon*

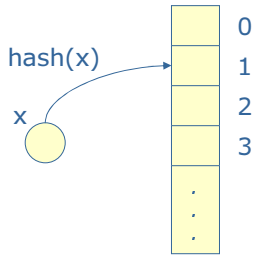
- Anta at en bilforhandler har 50 ulike modeller han ønsker å lagre data om.
- Hvis hver modell har et entydig nummer mellom 0 og 49 kan vi enkelt lagre dataene i en array som er 50 lang.

- Hva hvis numrene ligger mellom 0 og 49 999?
  - Array som er 50 000 lang: sløsing med plass!
  - Array som er 50 lang: søking tar lineær tid. . .

- ADT'en **hashtabell** tilbyr innsetting, sletting og søking med **konstant** gjennomsnittstid.
- Men: operasjoner som *finnMinste()* og *skrivSortert()* har **ingen** garantier.
- Brukes gjerne når vi først og fremst ønsker et raskt svar på om et gitt element finnes i datastrukturen eller ikke.

- Eksempler:
  - kompilatorer
    (Er variabel y deklarert?)
  - stavekontroller
    (Finnes ord x i ordlista?)
  - spill
    (Har jeg allerede vurdert denne stillingen via en annen trekkrekkefølge?)

- Ideen i hashing er å lagre elementene i en array (hashtabell), og la verdien til elementet x (eller en del av x, kalt nøkkelen til x) bestemme plasseringen (indeksen) til x i hashtabellen.



- Egenskaper til en god hash-funksjon:
  - Rask å beregne
  - Kan gi alle mulige verdier fra 0 til *tableSize* - 1.
  - Gir en god fordeling utover tabellindeksene.

HASHING – *Den teoretiske idealen*

Vi kan tenke oss følgende perfekte situasjon:

- n elementer
- tabell med n plasser
- hash-funksjon slik at
  - den er lett (rask) å beregne
  - forskjellige nøkkelverdier gir forskjellige indekser

Eksempel

- Hvis modellene er nummerert
  0, 1 000, 2 000, . . . , 48 000, 49 000
- kan data om modell *i* lagres på indeks *i*=1 000 i en tabell som er 50 stor.

Problem:

- Hva hvis modell 4 000 får nytt nummer 3 999?

HASHING – *Den praktiske realiteten*

- Vanligvis vil vi ikke finne noen perfekt hash-funksjon.
- Men vi ønsker at hash-funksjonen skal gi en så jevn fordeling som mulig.

Temaer

- Hvordan velge hash-funksjon?
  - Ofte er nøklene tekster (string'er)
- Hvordan håndtere kollisjoner?
- Hvor stor bør hashtabellen være?
  - Størrelsen er en del av ADT'en!

- For heltall: *key % tableSize*
- Gir jevn fordeling for tilfeldige tall.

- Pass på at ikke nøklene har spesielle egenskaper:
  Hvis *tableSize* = 10, og alle nøklene slutter på 0, vil alle elementene havne på samme indeks!

- Huskeregel:
  La alltid tabellstørrelsen være et primtall.

Tekst (eller "String") som nøkkel:

- Vanlig strategi: ta utgangspunkt i ascii/unicode-verdiene til hver bokstav og "gjør noe lurt".
- Funksjon 1: Summer verdiene til hver bokstav.

```
public static int hash1(String key, int tableSize)
{ int hashVal = 0;
  for (int i = 0; i < key.length(); i++)
  {   hashVal += key.charAt(i);
  }
  return (hashVal % tableSize);
}
```

Fordel:

- Enkel å implementere og beregne.

Ulempe:

- Dårlig fordeling hvis tabellstørrelsen er stor.

Funksjon 2

Bruk bare de tre første bokstavene og vekt disse.

```
public static int hash2(String key, int tableSize)
{ return (key.charAt(0) + 27 * key.charAt(1) + 729 * key.charAt(2))
        % tableSize;
}
```

- **Fordel**:
  Grei fordeling for tilfeldige strenger.
- **Ulempe**:
  Vanlig språk er ikke tilfeldig!

Funksjon 3 $\sum_{i=0}^{keySize-1} key[keySize - i - 1] \cdot 37^i$

```
public static int hash3(String key, int tableSize)
{  int hashVal = 0;
   for (int i = 0; i < key.length(); i++)
      hashVal = 37 * hashVal + key.charAt(i);
   hashVal = hashVal % tableSize;
   if (hashVal < 0)
      hashVal += tableSize;
   return hashVal;
}
```

Fordel:
- Enkel og relativt rask å beregne. Stort sett bra nok fordeling.

Ulempe:
- Beregningen tar lang tid for lange nøkler.

HASHING – *Et instruktivt eksempel*

Input: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Hash-funksjon:

$$hash(x, tableSize) = \sqrt{x}$$

0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81

Hva hvis hash-funksjonen hadde vært

$$hash(x, tableSize) = x \bmod tableSize$$

?

0	?
1	?
2	?
3	?
4	?
5	?
6	?
7	?
8	?
9	?

istedenfor?

NESTE GANG – *Oversikt*

Vi skal avslutte hashing:

- Kollisjoner og kollisjonshåndtering
- Åpen hashing
- Lukket hashing
  - Lineær prøving
  - Kvadratisk prøving
  - Dobbel hashing
- Rehashing
- Utvidbar hashing