INF2220 - Algoritmer og datastrukturer

HØSTEN 2007

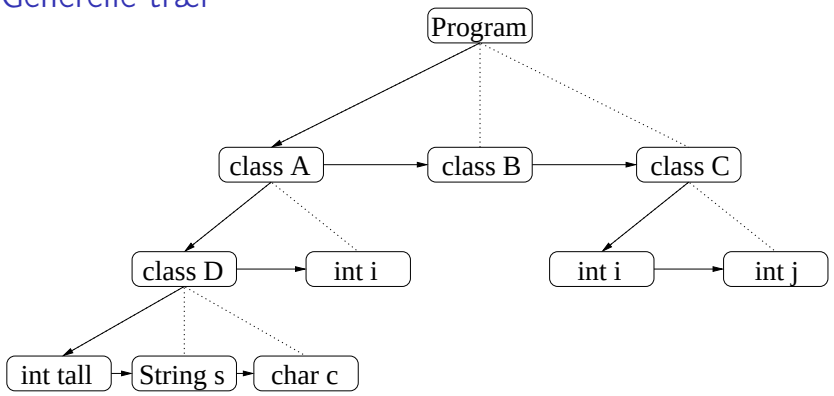
Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 2:
Binærtrær og abstrakte datatyper (ADT)

Dagens tema

- Kort repetisjon
  - Generelle trær (kap. 4.1)
- Binærtrær (kap. 4.2)
  - Implementasjon
  - Traversering
- Binære søketrær (kap. 4.3)
  - Definisjon
  - Søking, innsetting og sletting
  - Gjennomsnitts-analyse (!)
  - Eksempel: Ibsens skuespill
- Abstrakte datatyper (kap. 3.1)
  - Lister (kap. 3.2), stakker (kap. 3.3), køer (kap. 3.4)

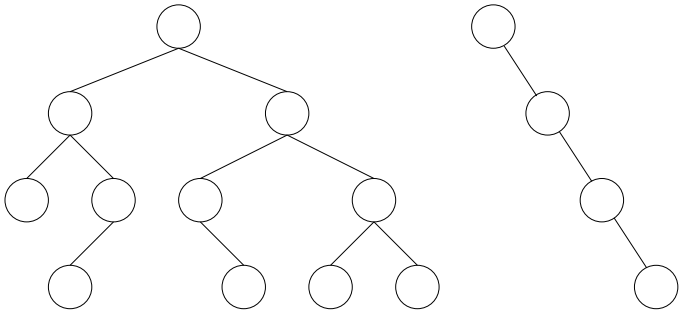
Generelle trær




```
class TreNode {
    Object element;
    TreNode forsteBarn;
    TreNode nesteSosken;
}
```


Binærtrær

Et binærtre er et tre der hver node aldri har mer enn to barn. Hvis det bare er ett subtre, må det være angitt om dette er venstre eller høyre subtre.



I verste fall blir dybden N-1!

Implementasjon av binærtrær

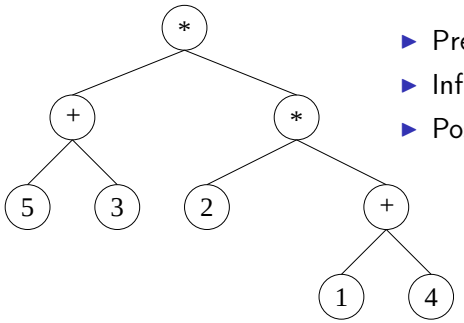
Siden hver node har maks to barn, kan vi ha pekere direkte til dem:

```
class BinNode {
    Object element;
    BinNode venstre;
    BinNode hoyre;
}
```


Eksempel: Uttrykkstrær

I uttrykkstrær inneholder bladnodene operander (konstanter, variable, ...), mens de interne nodene inneholder operatorer.

Eksempel



Mulige skrivemåter for dette uttrykket:

- Prefiks: * + 5 3 * 2 + 1 4
- Infiks: (5 + 3) * (2 * (1 + 4))
- Postfiks: 5 3 + 2 1 4 + * *

Alle disse eksemplene bruker dybde-først traversering. Dette er det vanligste for trær. (Bredde-først er også mulig, da brukes gjerne en kø.)

Traversering av binærtrær — oversikt

```
void traverser(BinNode n) {
    if (n != null) {
        < Gjør PREFIKS-operasjonene >
        traverser(n.venstre);
        < Gjør INFIKS-operasjonene >
        traverser(n.hoyre);
        < Gjør POSTFIKS-operasjonene >
    }
}
```

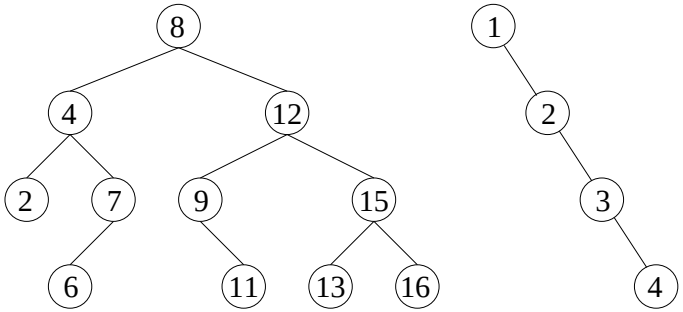

Oppgave 4.5

Vis at et binærtre med høyde h har maksimalt $2^{h+1} - 1$ noder.

Binære søketrær

Variant av binærtrær hvor følgende gjelder for hver node i treet:

- Alle verdiene i nodens **venstre** subtre er **mindre** enn verdien i noden.
- Alle verdiene i nodens **høyre** subtre er **større** enn verdien i noden.



Hva gjør vi med like verdier? (ukeoppgave)

Søking: Rekursiv metode

Antar klasse `BinNode` med attributter `tall`, `venstre`, `hoyre`.

```
public BinNode finn(int x, BinNode t) {
    if (t == null) {
        return null;
    } else if (x < t.tall) {
        return finn(x, t.venstre);
    } else if (x > t.tall) {
        return finn(x, t.hoyre);
    } else {
        return t;
    }
}
```

Antagelse: Alle verdiene i treet er forskjellige.

NB! Det må være mulig å sammenligne verdiene i treet — i Java, interface `Comparable`.

Søking: Ikke-rekursiv metode

```
public BinNode finn(int x, BinNode t) {
    BinNode n = t;
    while (n != null && n.tall != x) {
        if (x < n.tall) {
            n = n.venstre;
        } else {
            n = n.hoyre;
        }
    }
    return n;
}
```

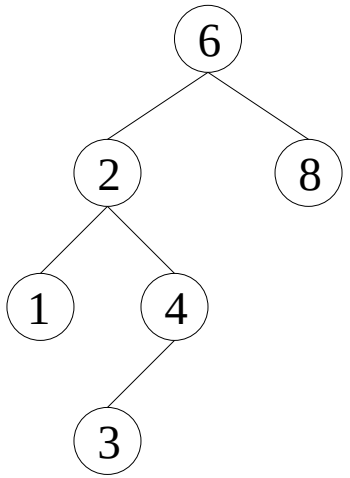
Her ser vi på det minst sannsynlige tilfellet *sist*.

Innsetting

Ideen er enkel:

- Gå nedover i treet på samme måte som ved søking.
- Hvis tallet finnes i treet allerede gjøres ingenting.
- Hvis du kommer til en null-peker uten å ha funnet tallet: sett inn en ny node (med tallet) på dette stedet.

Eksempel: Hvor plasseres tallet 5?



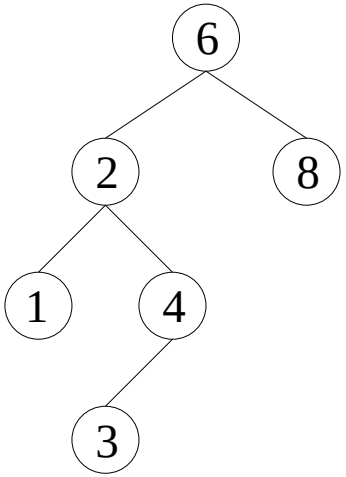
Sletting

Sletting er vanskeligere. Etter å ha funnet noden som skal fjernes, har vi flere mulige situasjoner.

- Noden er en bladnode:
  - Kan fjernes direkte.
- Noden har bare ett barn:
  - Foreldrenoden kan enkelt hoppe over den som skal fjernes.

(Hvis elementet ikke finnes i treet trenger vi opplagt ikke å gjøre noe.)

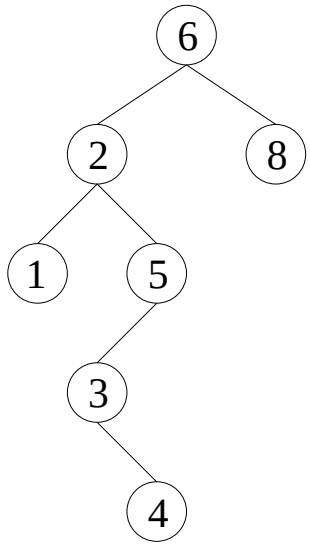
Eksempel: Fjerning av tallet 4.



Sletting av node med 2 barn

- Noden har to barn:
  - Erstatt verdien i noden med den minste verdien i **høyre** subtre.
  - Slett noden som inneholder denne minste verdien i subtreet.

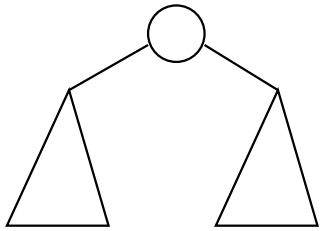
Eksempel: Fjerning av tallet 2.



Rekursiv metode for å fjerne et tall

```
public BinNode fjern(int x, BinNode t) {
    if ( t == null) { return null; }

    if (x < t.tall) {
        t.venstre = fjern(x, t.venstre);

    } else if (x > t.tall) {
        t.hoyre = fjern(x, t.hoyre);

    } else if (t.venstre != null && t.hoyre != null) {
        t.tall = finnMinste(t.hoyre);
        t.hoyre = fjern(t.tall, t.hoyre);

    } else if (t.venstre != null) { t = t.venstre;

    } else { t = t.hoyre; }

    return t;
}
```


Gjennomsnitts-analyse



Intuitivt vil vi forvente at alle operasjonene vi utfører på et binært søketre vil ta $O(\log n)$ tid siden vi hele tiden grovt sett halverer størrelsen på treet vi jobber med.

Det kan bevises at den **gjennomsnittlige dybden** til nodene i treet er $O(\log n)$ når *alle innsettingsrekkefølger* er like sannsynlige (se MAW kapittel 4.3.5).

Eksempel: Ibsen

Problem: Vi ønsker å analysere Henrik Ibsens skuespill.

- **Hvilke** ord har Ibsen brukt?
- Hvor mange **forskjellige** ord?
- Hvilke ord er **mest** brukt?

De mest brukte ordene (fra UiB):

	Antall	%	Kum.	
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

Spesielt interessert i: hvilke ord er mest brukt?

Dette brukes til å finne ut hvem som har skrevet verk med ukjent forfatter.

Vildanden

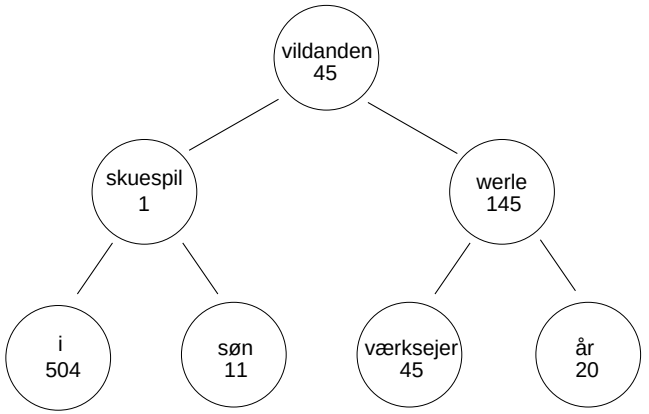
Vi skal se på ett av Ibsens skuespill:

```
VILDANDEN. SKUESPIL I FEM AKTER
AF HENRIK IBSEN. København. 1884

PERSONERNE:
Grosserer Werle, værksejer o+s+v+. Gregers Werle, hans søn.
Gamle Ekdal. Hjalmar Ekdal, den gamles søn, fotograf.
Gina Ekdal, Hjalmars hustru. Hedvig, deres datter, 14 år.
Fru Sørby, grossererens husbestyrerinde. Relling, læge.
Molvik, forhenværende teolog. Bogholder Gråberg. Pettersen,
grossererens tjener. Lejetjener Jensen. En blegfed herre.
En tyndhåret herre. En nærsynt herre. Sex andre herrer,
middagsgæster hos grossereren. Flere lejetjenere.

Første akt foregår hos grosserer Werle, de fire
følgende akter hos fotograf Ekdal..

FØRSTE AKT.
_____
I grosserer Werles hus. Kostbart og bekvemt indrettet
arbejdsværelse; bogskabe og stoppede møbler; skrivebord
med papirer og protokoller midt på gulvet; tændte lamper
med grønne skærme, således at værelset er dæmpet belyst.
Åben fløjdør med fratrukne forhæng på bagvæggen.
```


Implementasjon

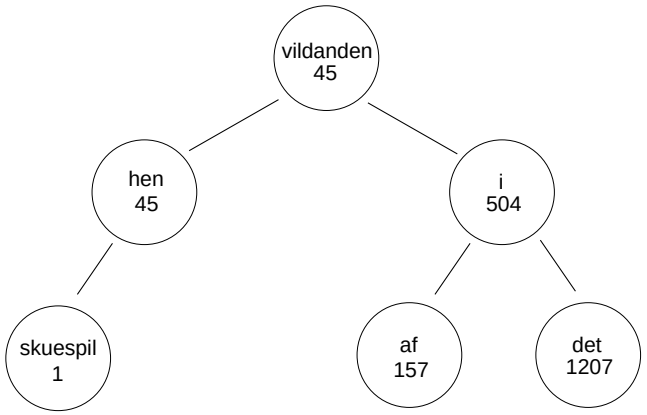
Alternativ 1: Hvilke ord har Ibsen brukt?

Ide: Leser inn ordene fra fil og lagrer dem i et binært søketre.
Like ord telles opp.



Alternativ 2: Hvilke ord er mest brukt?

Ide: Lager et nytt søketre sortert på antall ganger hvert ord er brukt!




```
import easyIO.*;

public class Ibsen {
    public static void main(String[] args) {
        String ord;
        int antallOrd = 0;
        IbsenTre tre = new IbsenTre();
        FrekvensTre frekTre = new FrekvensTre();

        String sep = ".␣:,;-?!\"\n";
        In innfil = new In(args[0]);
        innfil.skipSep(sep);

        while (!innfil.endOfFile()) {
            ord = innfil.inWord(sep).toLowerCase();
            tre.settInn(new IbsenElem(ord));
            antallOrd++;
            innfil.skipSep(sep);
        }

        System.out.println("Antall␣ord:␣" + antallOrd);
        System.out.println("Ulike␣ord:␣" + tre.size());

        frekTre.innsetting(tre.getRot());
        frekTre.skrivInnfiks();
    }
}
```

Antall ord: 31268
Antall forskjellige: 3547

Klassen BinNode

Vanlig node i binært søketre:

```
class BinNode {
    Comparable element;
    BinNode venstre;
    BinNode hoyre;

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


Klassen IbsenElem

Tar vare på ordene og teller 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);
    }
}
```

Klassen BinSokeTre

Generell klasse for binære søketrær der "noe gjøres" ved like elementer:

- void settInn(Comparable x)
  - Innsetting av element x i treet.
- void oppdater(BinNode n, Comparable x)
  - Kalles av settInn hvis x allerede finnes i treet (i node n).
    Default: Ingenting gjøres.
- int sammenlign(Comparable n1, Comparable n2)
  - Kalles av settInn for å sammenligne elementet som skal settes inn med de som allerede finnes i treet.
    Default: n1.compareTo(n2).
- int size()
  - Antall noder i treet.
- BinNode getRot()
  - Peker til roten i treet.

Tre sortert på ord

```
class IbsenTre extends BinSokeTre {

    void oppdater(BinNode n, Comparable e) {
        IbsenElem ie = (IbsenElem) n.element;
        ie.antall++;
    }

}
```

NB! Treet blir veldig skjevt, med 3388 noder i venstre subtre og bare 158 noder i høyre subtre.

Hva er årsaken til dette?

Tre sortert på frekvens

```
class FrekvensTre extends BinSokeTre  {

    int sammenlign(Comparable c1, Comparable c2) {

        IbsenElem e1 = (IbsenElem) c1;
        IbsenElem e2 = (IbsenElem) c2;

        return e1.antall - e2.antall;
    }

    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++;
    }

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


Binære søketrær: oppsummering

Hvordan blir balansen i forrige eksempel?

Venstre: 3444 Høyre: 102

Hvis infiks/postfiks-innsetting fra det andre treet: 2051/1496!

Treet blir veldig høyt! (2053) - skyldes alle like antall.

Viktig å passe for skjeve trær...

Abstrakte datatyper

En **ADT** består av:

- et sett med objekter
- spesifikasjon av operasjoner på disse

Eksempler:

```
ADT: binært søketre
Operasjoner: Innsetting, søking, fjerning, ...
```

```
ADT: mengde
Operasjoner: union, snitt, element, ...
```

```
ADT: stakk
Operasjoner: push, pop, top, ...
```


Hvorfor bruke ADTer?

ADTer skiller det som er viktig (funksjonaliteten) fra detaljene (den konkrete implementasjonen). Dermed kan vi:

- Gjenbruke ADTen i andre programmer.
- Enklere overbevise oss om at programmet er riktig.
- Forandre innmaten (kodingen) av ADTen uten å forandre resten av programmet fordi grensesnittet er det samme.
- Lage modulære programmer.

I Java er det naturlig å spesifisere en ADT som et `interface`.

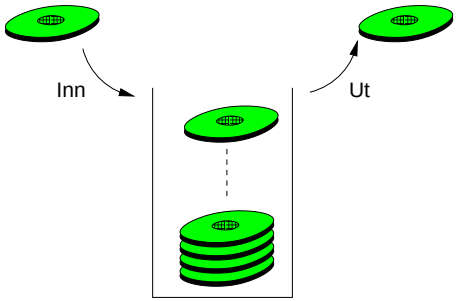
Jfr. interface `Comparable`.

Lister, stakker og køer

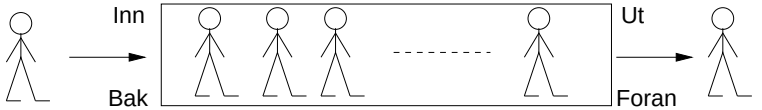
- Lister (kap. 3.2)

$$A_1,\ A_2,\ A_3,\ \ldots,\ A_n$$

- Stakker (kap. 3.3)



- Køer (kap. 3.4)



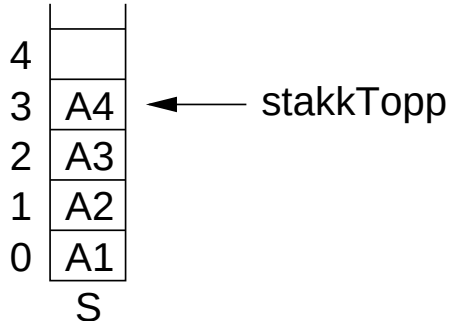
Stakker

Vi bare kan sette inn og slette elementer fra en bestemt ende av "listen".

```
public interface StakkInterface {
    /* Legge et element på toppen av stakken */
    void push(Object x);

    /* Fjerne et element fra toppen av stakken */
    void pop();

    /* Returnere elementet på toppen av stakken */
    Object top();

    /* Lege en ny stakk/tømme stakken */
    void create();

    /* Sjekke om stakken er tom */
    boolean isEmpty();
}
```

Ofte vil pop også returnere elementet som fjernes.
En stakk er det samme som en LIFO-kø ("Last In First Out").

Array-implementasjon av stakk



Brukes ofte hvis antall elementer på stakken alltid er begrenset.

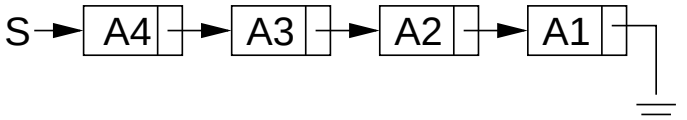

```
public void push(Object x) {
    stakkTopp++;
    S[stakkTopp] = x;
}
```

```
public void pop() {
    stakkTopp--;
}
```

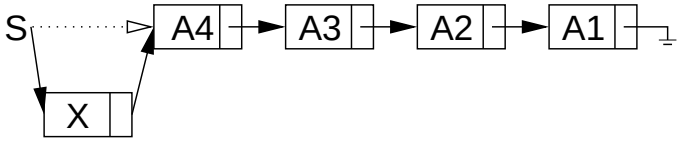
```
public Object top() {
    return S[stakktopp];
}
```

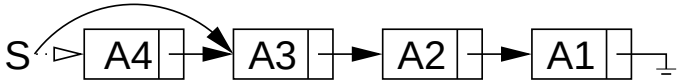
I tillegg: eventuell feilhåndtering.

Pekerkjede-implementasjon av stakk



Innsetting og **sletting** vil da normalt skje på begynnelsen av listen:





Stakkoperasjonene - tidsforbruk

Uansett hvor mange elementer vi har på stakken, er vi garantert konstant tidsforbruk, det vil si $O(1)$ for alle operasjonene. Dette gjelder uansett om implementasjonen bruker en array eller en pekerkjede.

Bedre er det ikke mulig å få det, derfor er stakken en veldig populær ADT. Samtidig kan mange "naturlige" problemer løses ved hjelp av en stakk.

Typisk bruksmønster er mange stakk-operasjoner, men få elementer på stakken om gangen.

På mange maskiner kan disse oversettes til kun to-tre instruksjoner i maskinkode.

Eksempel: Beregning av postfiks uttrykk

Ved hjelp av en stakk er det lett å beregne et postfiks uttrykk:

- For hvert symbol i input:
  - Hvis symbolet er et tall, legges det på stakken.
  - Hvis symbolet er en operator, "popper" vi to tall fra stakken, anvender operatoren på disse to tallene og dytter svaret tilbake på stakken.
- Hvis input var et ekte postfiks uttrykk, vil nå svaret ligge som det eneste elementet på stakken.

Eksempel: 6 5 2 3 + 8 * + 3 + *