INF1020 - HØSTEN 2005

Kursansvarlige

- Ragnhild Kobro Runde
  E-post: ragnhilk@ifi.uio.no
  Kontor: 4306
- Almira Karabeg
  E-post: almira@ifi.uio.no
  Kontor: 3306

Dagens tema

- Praktiske opplysninger
- Analyse av algoritmer (kapittel 2)
- Introduksjon til trær (kapittel 4)

INF1020

- Et av de mest sentrale grunnkursene i informatikkutdanningen — og et av de vanskeligste!
- Kurset hever programmering fra et håndverk til et universitetsfag.
- Eksamen krever både teoretiske og praktiske ferdigheter.
- Et arbeidskrevende modningsfag. Løs oppgaver gjennom hele semesteret!
- Lærebok "Data Structures & Algorithm Analysis in Java" av Mark Allen Weiss.
- Sjekk fagets hjemmeside regelmessig. Forelesningsplanen kan bli endret underveis.
- Kurset forutsetter INF1010. Spesielt: Rekursjon.

Gruppeundervisningen

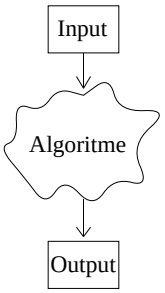
Tre typer grupper:

- Regneøvelser (orakeltimer):
  Gruppe 1, 2 og 4.
- Undervisning for studenter som ønsker å forberede seg på ukeoppgavene:
  Gruppe 5, 6 og 7.
- Undervisning for studenter som ikke har forberedt seg:
  Gruppe 3, 8 og 119.

Påmelding skjer ved oppmøte første gruppetime (neste uke) på den gruppen du ønsker å gå på. NB: Obligatorisk oppmøte første gruppetime!

Hva er en algoritme?

Vanlig sammenligning: Oppskrift.



Knuth†: I tillegg til å være et endelig sett med regler som gir en sekvens av operasjoner for å løse en bestemt type problem, har en algoritme fem viktige kjennetegn:

1. **Endelighet.**
2. **Definerthet.**
3. **Input.**
4. **Output.**
5. **Effektivitet.**

† Donald E. Knuth: The Art of Computer Programming. Volume 1/Fundamental Algorithms.

Analyse av tidsforbruk

Hvor mye øker kjøretiden når vi øker størrelsen på input?

To typer analyse:

- Gjennomsnittlig tidsforbruk (average-case)
- "Verste tilfelle" (worst-case)

Alternative metoder:

- Kode algoritmen og ta tiden for ulike størrelser på input.
- Finne en enkel funksjon som vokser "på samme måte" som eksekveringstiden til programmet.

Logaritmer

Logaritmer har et grunntall X, for eksempel X = 2 eller X = 10. Vi bruker stort sett X = 2.

Logaritmen til et tall **B** er det tallet **A** vi må opphøye grunntallet **X** i for å få **B**, det vil si: $X^A = B \Leftrightarrow A = \log_X B$

Eksempler:

$2^1 = 2 \Leftrightarrow 1 = log_2\ 2$
$2^2 = 4 \Leftrightarrow 2 = log_2\ 4$
$2^3 = 8 \Leftrightarrow 3 = log_2\ 8$
$2^4 = 16 \Leftrightarrow 4 = log_2\ 16$
$2^{10} = 1\ 024 \Leftrightarrow 10 = log_2\ 1\ 024$
$2^{20} = 1\ 048\ 576 \Leftrightarrow 20 = log_2\ 1\ 048\ 576$

O-notasjon

Generelt er vi ikke interessert i nøyaktig hvor mye tid et program bruker, men heller prøve å angi i hvilken **størrelsesorden** løsningen ligger.

Definisjon
La $T(n)$ være kjøretiden til programmet.

- $T(n) = O(f(n))$ hvis det finnes positive konstanter c og n_0 slik at $T(n) \leq c * f(n)$ når $n > n_0$.

$O(f(n))$ er da en øvre grense for kjøretiden. Problemet er å finne en $f(n)$ som er minst mulig.

Vanlige funksjoner for O( )

Funksjon	Navn
1	Konstant
log n	Logaritmisk
n	Lineær
n log n	
n^2	Kvadratisk
n^3	Kubisk
2^n, n!	Eksponensiell

O-notasjon er en veldig grov måte å angi tidsforbruk på, og vi forkorter så mye som mulig. Merk at konstanter allerede ligger i definisjonen.

Eksempler:

- $n/2$, n, $2n$ er alle $O(n)$.
- $n^2 + n + 1$ regnes som $O(n^2)$.
- $log_2\ n$, og $log_{10}\ n$ skilles bare av en konstant, begge angis med $O(log\ n)$.

Enkel beregning av tid

Enkel setning:

```
x = y + z;
```

Enkel for-løkke:

```
for (int i = 0; i < n; i++) {
   brukt[i] = false;
}
```

Nestede for-løkker:

```
for (int i = 0; i < n; i++) {
   for (int k = 0; k < n; k++) {
      avstand[i][k] = 0;
   }
}
```


Sekvens av setninger:

```
x = y + z;
for (int i = 0; i < n; i++) {
   brukt[i] = false;
}
```

Betinget setning:

```
if (n < 0) {
   System.out.println("Ingen elementer");
} else {
   for (int i = 0; i < n; i++) {
      sum += i;
   }
}
```


Trær

Ulike typer trær:

- Generelle trær (kap. 4.1)
- Binærtrær (kap. 4.2)
- Binære søketrær (kap. 4.3)
- B-trær (kap. 4.7)

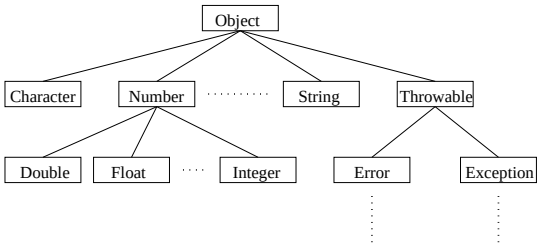
Aktuelle temaer:

- Bruksområder
- Implementasjon
- Innsetting/fjerning av elementer
- Søking etter elementer
- Traversering

Trær brukes gjerne til å representere en rekke typer av data som er organisert hierarkisk.

Typiske eksempler er:

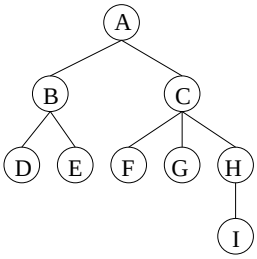
- Filorganisering
- Slektstrær
- Organisasjonskart
- Klassehierarki (i for eksempel Java)



Definisjon

Et tre er en samling noder. Et ikke-tomt tre består av en spesiell node r, kalt roten, og null eller flere ikke-tomme subtrær. Fra r går det en rettet kant til roten i hvert subtre.

Terminologi



Eksempler:

- A er roten (rot-noden)
- B er forelder til D og E
- D og E er barna til B
- C er søsken til B
- D, E, F, G og I er bladnoder (løvnoder)
- A, B, C og H er interne noder

Implementasjoner

Forslag 1

I tillegg til data kan hver node inneholde en peker til hvert av barna.

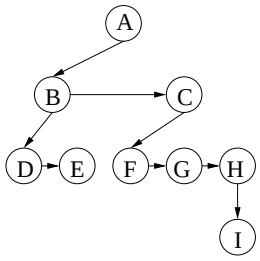
Problemer:

- Antall barn kan variere veldig fra node til node.
- Vet ikke nødvendigvis antall barn på forhånd.

Forslag 2

Hver node inneholder en **liste** med pekere til barna.

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



Traversering

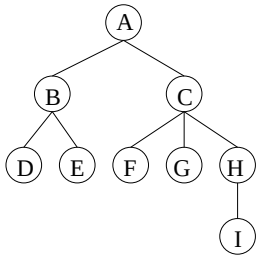
Å traversere et tre vil si å besøke alle (eventuelt bare noen av) nodene i treet, for eksempel fordi vi:

- Leter etter en bestemt node (element)
- Vil sette inne en ny node (med et nytt element)
- Vil fjerne en node/element
- Vil gjøre en eller annen beregning (eller utskrift) på alle nodene i treet.

Hvilken rekkefølge vi besøker nodene i bestemmes av det problemet vi skal løse.

To populære traverseringsmåter:

- Prefiks (preorder): Gjør noe med en node **før** vi går videre til barna.
- Postfiks (postorder): Gjør noe **etter** at vi har besøkt **alle** barna til noden.



Prefiks:

Postfiks:

Mer terminologi

En vei (sti) fra en node n_1 til en node n_k er definert som en sekvens av noder $n_1, n_2, \ldots, n_k$ slik at n_i er forelder til n_{i+1} for $1 \leq i \leq k$.

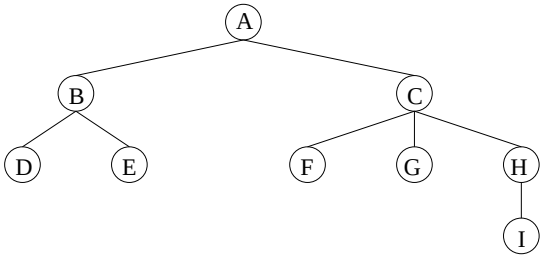
Lengden av denne veien er antall kanter på veien, det vil si $k - 1$.

For enhver node er dybden definert som lengden av (den unike) veien fra roten til noden. Roten har altså dybde 0.

Rekursiv metode for å beregne dybden til alle nodene i et tre:

```
// Kall: rot.beregnDybde(0)

void beregnDybde(int d) {
   this.dybde = d;
   for ( < hvert barn b > ) {
      b.beregnDybde(d + 1);
   }
}
```



Høyden til en node er definert som lengden av den lengste veien fra noden til en bladnode. Alle bladnoder har dermed høyde 0.

Høyden til et tre er lik høyden til roten.

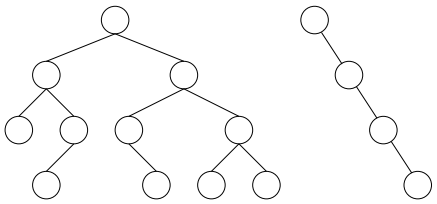
Rekursiv metode for å beregne høyden til alle nodene i et tre:

```
// Kall: rot.beregnHoyde()

int beregnHoyde() {
   int tmp;
   this.hoyde = 0;
   for ( < hvert barn b > ) {
      tmp = b.beregnHoyde() + 1;
      if (tmp > this.hoyde) {
         this.hoyde = tmp;
      }
   }
   return this.hoyde;
}
```


Binærtrær

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



I verste fall blir dybden N-1!

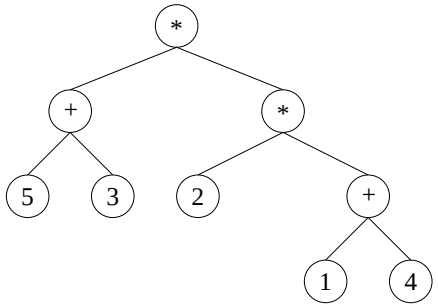
Implementasjon

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.



Mulige skrivemåter for dette uttrykket:

- Prefiks:
- Infiks:
- Postfiks:

Traversering av binærtrær — oppsummering

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