INF2220 - Algoritmer og datastrukturer

HØSTEN 2007

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 4:

B-trær, Rød-svarte trær, collection, sets og maps

Dagens plan:

- B-trær (kap. 4.7)
- Rød-svarte trær (kap. 12.2)
- Interface `Collection` og `Iterator` (kap. 3.3)
- Set og maps (kap. 4.8)

Repetisjon: Binære søketrær

For enhver node i et binært søketre gjelder:

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

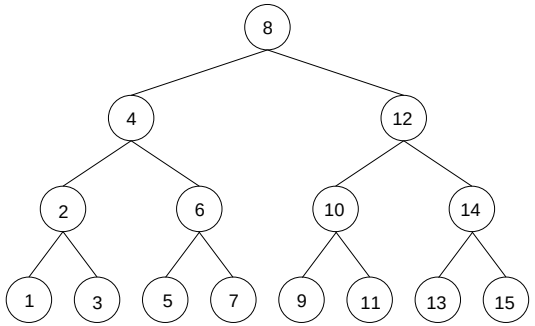
Kode for å sjekke at et binærtre er et binært søketre

```
boolean binSokeTre( BinNode t ){
    return bst(t, -UENDELIG, UENDELIG )
        }

boolean bst( BinNode t, int min, int max ){
    if (t==null) return true;
    int d=t.element;
    if (d<min || d>max) return false;
    return (bst(venstre, min, d) && bst(hoyre, d+1, max));
}
```


Repetisjon: Binære søketrær

Eksempel på perfekt balansert tre



Høyde: $\lfloor \log_2(N) \rfloor$

Antall noder: $2^{h+1} - 1$.

Lengden på alle grener er logaritmisk!

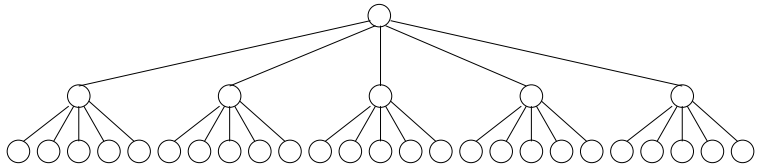
B-trær

- **Forrige uke**: løsninger for utvidbar hashing
- Problem: Ikke plass i internminnet til hele datastrukturen
- Må lagre på disk
- Disk-operasjoner er veldig kostnadskrevende

- **I dag**: Vi kombinerer disse teknikkene med binære søketrær
- En annen type søketrær.
- Brukes først og fremst når ikke hele treet får plass i internminnet.
- Har stor bredde (hver node har mange barn) og er balansert.
- De øverste nivåene (iallfall rotnoden) lagres i internminnet, resten på disk.
- Brukes særlig i *databasesystemer*

Hovedidé

Hvis treet ligger på disk lar vi treet være *lavere* enn et vanlig binærtre. Da må treet også være bredere.



Ønsker samtidig å bevare sorteringen vi kjenner fra binære søketrær.

Begrenser antall disk-aksesser for å komme til en løvnode

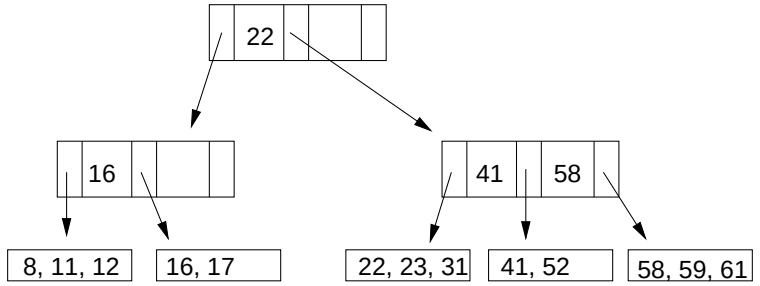
Her ser vi et B-tre av **orden 5**.

Definisjon: B-trær av orden M

1. Alle data (eller pekere til data) er lagret i bladnodene.
2. Interne noder lagrer inntil $M-1$ nøkler for bruk i søking; nøkkel i angir den minste verdien i subtre $i+1$.
3. Roten er enten en bladnode, eller har mellom 2 og M barn.
4. Alle andre interne noder har mellom $\lceil M/2 \rceil$ og M barn.
5. Alle bladnoder har samme dybde og har mellom $\lceil L/2 \rceil$ og L dataelementer (eller datapekere), der L er en konstant felles for alle bladnoder.

Merk: Lærebokens (og våre) B-trær er ellers i litteraturen kjent som $\mathbf{B}^+$-trær. Tradisjonelle B-trær har data(pekere) i alle noder.

Eksempel: B-tre av orden 3

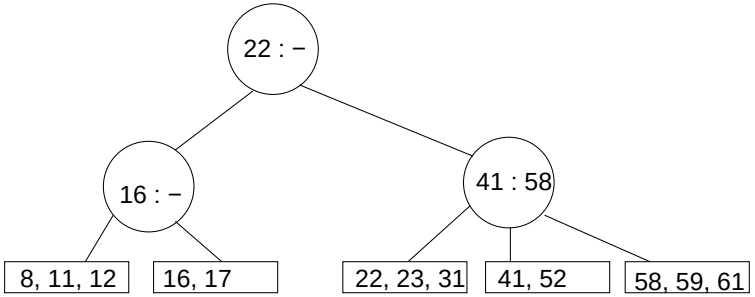


Her er $M = 3$ og $L = 3$.

Merk: Interne noder minner om kataloger i utvidbar hashing

- **M** angir katalogstørrelsen (antall pekere fra hver interne node) og
- **L** angir kapasiteten på hver diskblokk

Vi forenkler tegningene litt



Representerer blokker (løvnoder) som firkanter, og interne noder som sirkler

Husker på hvordan vi går videre fra interne noder...

Søking etter element x

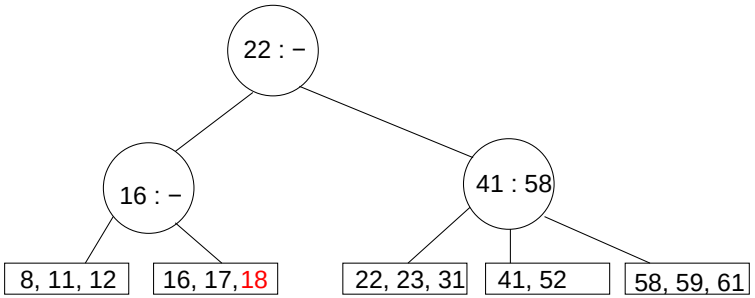
1. Start i roten.
2. Så lenge vi ikke er i en bladnode:
   La nøkkel-verdiene bestemme hvilket barn vi skal gå til.
3. Let etter x i bladnoden.

Oppnår kontroll på antall disk-operasjoner

Innsetting av element x

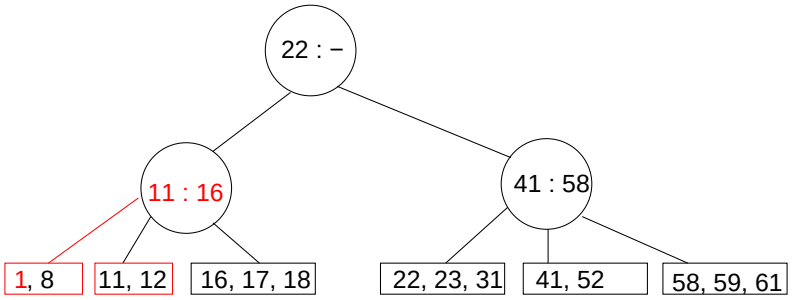
1. Let etter riktig bladnode for x (som for søking).
2. Dersom det er plass, setter vi inn x og oppdaterer eventuelt nøkkel-verdiene langs veien vi gikk.

   Eksempel: Sett inn 18 i treet i eksempelet under.



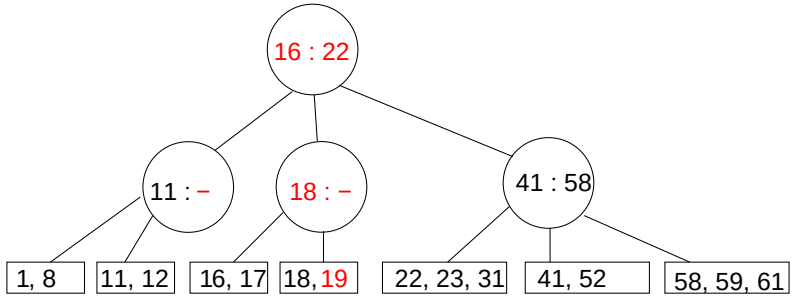
3. Dersom bladnoden er full, deler vi den i to og fordeler de $L + 1$ nøklene jevnt på de to nye bladnodene.

   Eksempel: Sett inn 1 i treet på forrige foil.



4. Dersom splittingen medfører at foreldernoden får for mange barn, splitter vi den også, gir den nye noden til besteforelderen, osv.

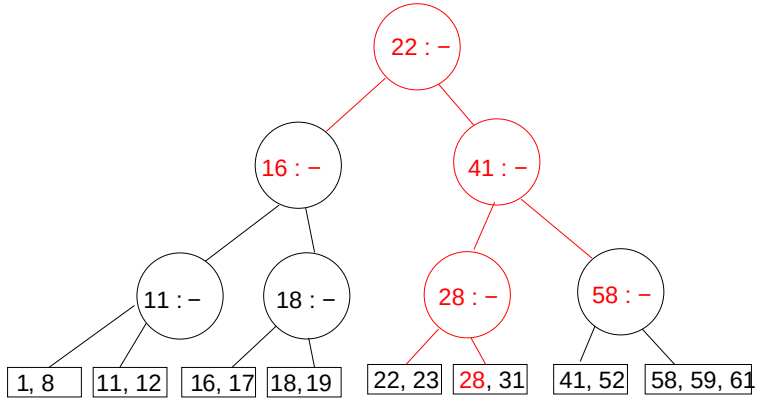
   Eksempel: Sett inn 19 i treet på forrige foil.



5. Dette kan medføre at vi til slutt må splitte roten i to. (Hvis roten får $M+1$ barn.) Lager da ny rot med gammel rot og ny node som barn.

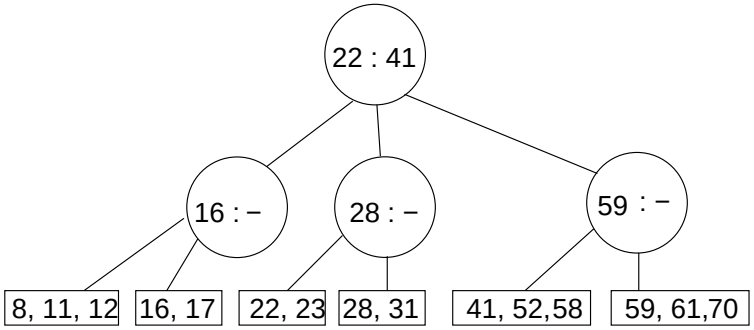
   Merk: Dette er den eneste måten et B-tre kan vokse i høyden på!

   Eksempel: Sett inn 28 i treet på forrige foil.



Sletting

Eksempel: Fjern først 17, deretter 23 fra følgende tre



Problem: Når vi fjerner 17, blir det for få elementer i bladnoden!
Problem: Når vi fjerner 23 er det få elementer i nabo-noden til å benytte samme løsning!

Sletting av element x

1. Finn riktig bladnode B for x ved søking.
2. Dersom B har minst $\lceil L/2+1 \rceil$ elementer, kan vi enkelt slette x.
3. Hvis ikke, må vi *kombinere B med en av nabosøsknene*:
   - 3.1 Dersom venstre (høyre) søsken har $\lceil L/2+1 \rceil$ elementer eller mer, flytter vi det største (minste) elementet til B.
   - 3.2 Dersom nabosøskenen har akkurat $\lceil L/2 \rceil$ elementer, slår vi de to nodene sammen til én node (med L eller $L-1$ elementer)
4. Hvis foreldrenoden nå har et barn for lite, må vi gjøre det samme med denne. Osv...
5. Hvis roten til slutt bare har ett barn: Slett roten, og la barnet bli ny rot. (Treet krymper nå ett nivå.)
6. Husk å oppdatere nøkkelverdiene underveis!

Tidsforbruk

- Vi antar at M og L er omtrent like ($M \geq 2$)
- Siden hver interne node unntatt roten har minst $\lceil M/2 \rceil$ barn, er dybden til et B-tre maksimalt $\lceil \log_{\lceil M/2 \rceil} N \rceil$.
- På hver node må vi utføre $O(\log M)$ arbeid (ved søk i binært søketre) for å avgjøre hvilken gren vi skal gå.
- Dermed tar søking $O(\log M \cdot \log_{M/2} N) = O(\log N)$ tid.
- Ved innsetting og sletting kan det hende at vi må utføre $O(M)$ arbeid på hver node for å rydde opp (f.eks. flytte alle nøkkelverdiene i tabellen en plass til venstre).
- Så innsetting og sletting kan ta $O(M \log_{M/2} N) = O((\frac{M}{\log M}) \log N)$ tid.

Hvor stor skal M være?

Hvor mange barn skal en node få lov å ha?

- Hvis hele B-treet får plass i internminnet, har empiriske målinger vist at $M = 3$ og $M = 4$ er de beste valgene (innsetting og sletting tar for lang tid hvis M blir for stor).
- Men B-trær har sin store styrke når ikke hele treet får plass i internminnet.
- Siden en diskoperasjon tar nesten 1 000 000 ganger mer tid enn en operasjon i internminnet, gjelder det å minimalisere antall diskaksesser.
- Hvis det er plass, er det en god idé å lagre alle internnoder i internminnet og alle bladnoder på harddisk.
- Da kan man velge $M = 4$ og L så stor at hver bladnode fyller en diskblokk (eventuelt et disk cluster).

Hvor stor skal M være når hele treet ligger på disk?

- Treet blir bredere og får mindre dybde desto større M er.
- Mindre dybde betyr færre diskaksesser, mens vi kan se bort fra det ekstra oppryddingsarbeidet i nodene som en stor M medfører fordi det foregår i internminnet.
- I praksis velger man M så stor at en internnode fortsatt får plass på én diskblokk (eller cluster), typisk i området $32 \leq M \leq 256$.
- Man velger L slik at det samme gjelder for bladnodene.
- Analyser viser at B-trær blir $\ln \mathbf{2} = \mathbf{69}\%$ fulle (samme fyllingsgrad som utvidbar hashing).

Rød-svarte trær

Problem: Hvordan få et binært søketre balansert?

Et **rød-svart tre** er et binært søketre der hver node er farget enten rød eller svart slik at:

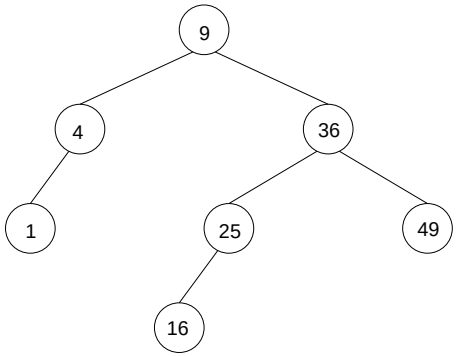
- Roten er svart.
- Hvis en node er rød, må barna være svarte.
- Enhver vei fra en node til en `null`-peker må inneholde samme antall svarte noder.

Disse fargeleggingsreglene sikrer at

høyden på et rød-svart tre er maksimalt $2 \log_2(N + 1)$!

Oppgave

- Farg nodene i følgende tre slik at det blir et rød-svart tre:



- Sett inn tallet 64 på riktig plass og med riktig farge.

- Sett inn tallet 64 på riktig plass og med riktig farge.

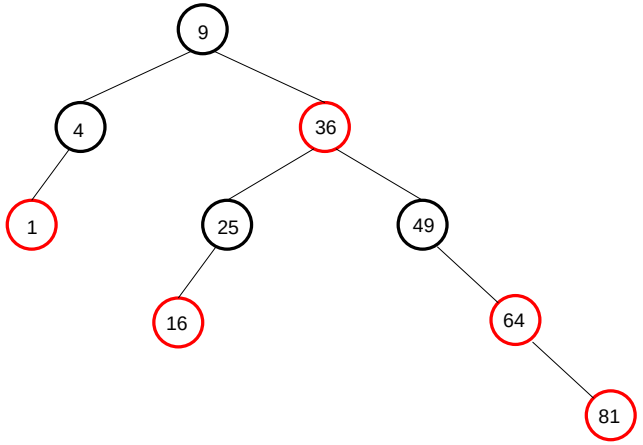
Rotasjoner i treet

- Innsetting går bra hvis foreldernoden er **svart**.
- Da blir den nye noden **rød**, og
  antall svarte noder langs veien er uforandret.
- Hvis forelder er **rød**, kan ikke den nye noden være **rød**.
  - Nødvendig å endre på treet
  - Dette gjøres vha *rotasjoner*
  - Flere ulike tilfeller som må dekkes
  - To ulike rotasjoner *zig* og *zig-zag*
  - Hver av disse rotasjonene har et *symmetrisk* tilfelle!

Hovedidé: Setter inn den nye noden og lar den være **rød**. Den nye treet er ikke lenger korrekt, men ved å rotere det får vi et tre som

- fremdeles er et binært søketre
- igjen er rød-svart

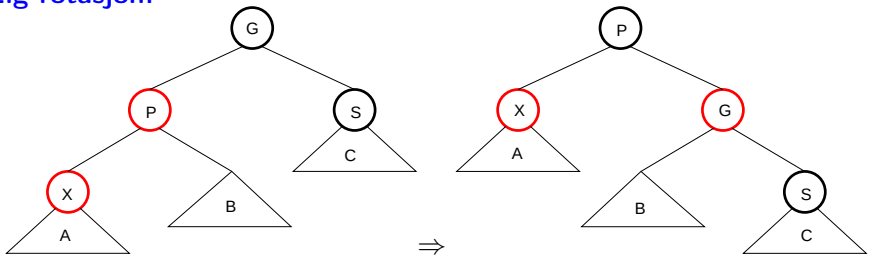
Tilbake til eksempelet



Rotasjoner på binære søketrær

Vi tenker oss nå at hver av nodene i eksempelet er rot i et subtre...
Rotnoden G er **svart**, antar først at node S er **svart**

Zig rotasjon:



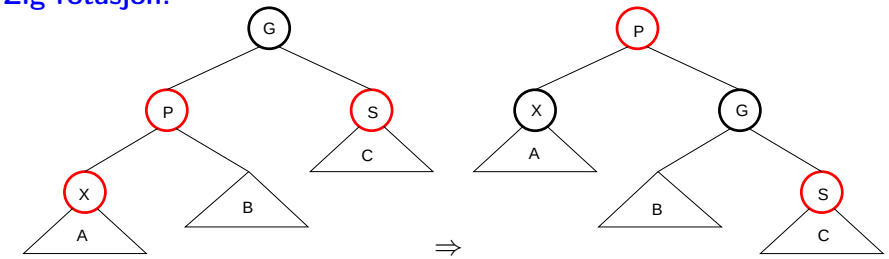
+ symmetrisk tilfelle...

Antall svarte noder langs veiene i treet er uforandret av rotasjonen, men de to røde nodene er ikke lenger etter hverandre

Rotasjoner på binære søketrær

Rotnoden G er **svart**, antar nå at noden S er **rød**

Zig rotasjon:



+ symmetrisk tilfelle. . . (som i eksempelet)

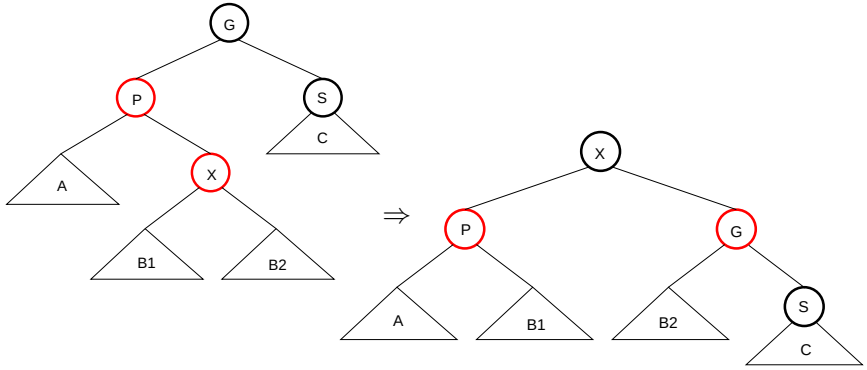
Samme rotasjon, men ulik farging av nodene. . .
Antall svarte noder langs veiene er uforandret

Problem: Hva hvis forelder til subtreet er **rød**?

Merk: Rotasjonen gjelder generelt, ikke bare for rød-svarte trær!

Zig-zag rotasjon

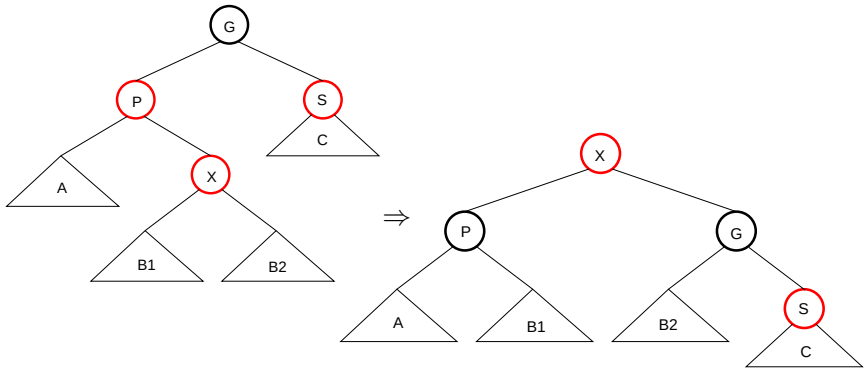
Rotnoden G er **svart**, antar først at node S er **svart**



+ symmetrisk tilfelle...

Zig-zag rotasjon

Rotnoden G er **svart**, antar først at node S er **rød**



+ symmetrisk tilfelle...

Innsetting i rød-svart tre

1. Gjør innsetting som i vanlig binært søketre, der den nye noden **X** farges rød.
2. La **P** og **G** være forelder og besteforelder til **X**.
3. Hvis **P** er svart: Alt ok, innsetting ferdig.
4. Hvis **P** er rød:
   - 4.1 Hvis **X** og **P** er begge venstre eller begge høyre barn: Gjør zig rotasjon med nødvendige fargeendringer.
   - 4.2 Hvis **X** er venstre og **P** er høyre barn eller motsatt: Gjør zig-zag rotasjon med nødvendige fargeendringer.
   - 4.3 Sett **X** til å være den nye roten i det roterte subtreet.
   - 4.4 Hvis **X** er roten i selve treet: Farg denne svart.
     Ellers: Gjenta fra steg 2.

Implementasjon

Bottom-up

- Først søke nedover treet for å finne riktig sted å sette inn element
- Så permutere treet hvis nødvendig for å gjenopprette rød-svart kravet
- Hvis S-noden er **rød**, blir det nødvendig å rekursere oppover i treet.

Top-down

- Gjøre endringer på vei ned
- Tricks for omfarging av noder
- Kan garantere at S-noder ikke blir røde
- Dermed blir det aldri nødvendig med mer enn én rotasjon
- Mer detaljer i boka

Anvendelser av rød-svarte trær

- Java: brukes bl.a. til effektive implementasjoner av interfacene `Set` og `Map`
- Disse er eksempler på *kolleksjonsklasser*: Objekter som tilbyr effektiv implementering av hent/finn/fjern-operasjoner
- Angitt ved interface `Collection`

Java Collection

```
public interface Collection<AnyType>
    extends Iterable<AnyType> {
    int size();
    boolean isEmpty();
    void clear();
    boolean contains( AnyType x );
    boolean add( AnyType x );
    boolean remove( AnyType x );
    java.util.Iterator<AnyType> iterator();
}
```

- Definisjonen bruker parametriserte typer — ikke så viktig for kurset sin del (men kan jo være greit å se litt på likevel!)
- `Set` og `Map` er interfacer som utvider `Collection`
- `TreeSet` og `TreeMap` implementerer disse grensesnittene og lagrer objekter sortert (etter en ordning gitt ved `Comparator`-objekt).

Java Iterator

```
public interface Iterator<AnyType> {
    boolean hasNext();
    AnyType next();
    void remove();
}
```

Iterator tilbyr `next`-peker inn i datastrukturen
⇒ muliggjør **for**-løkker...

Merk: remove krever forsiktighet!

La oss se på hvordan vi kan implementere interface `Set`...

```
class BinærTreSet implements Set {
  BinNode rot;
  ...
  Iterator iterator(){
    returns new InfiksTreTravers(rot);
  }
}
```

Problem: Iteratoren er ikke en rekursiv struktur

I stedet skal `next`-metoden flytte seg til nytt element hver gang den kalles

Mulige løsninger:

- Vi kan lagre noder på en stakk som vi bruker under traverseringen. Dette er en vanlig måte å eliminere rekursjon
- Vi kan utvide treet med foreldrepekere (se ukeoppgave)
- Legge på lenker som gir en listestruktur gjennom treet (forrige- og neste-pekere). dette øker minnebruken...
- Liste-pekere kan begrenses til å erstatte null-pekere i treet: ofte en god løsning!

La oss prøve oss på første løsning
(vi ser ikke på remove-metoden — prøv gjerne å gjøre denne!)


```
public class InfiksTreTravers implements Iterator {

    /* StakkImplementasjon er en hvilken som helst
       klasse som implementerer Stakk interface */

    private Stakk stk = new StakkImplementasjon();

    public InfiksTreTravers( BinNode rot ) {
        skliVenstre(rot);
    }

    public boolean hasNext() {
        return ! stk.isEmpty();
    }

    public Object next () {
        BinNode denne = (BinNode) stk.pop();
        skliVenstre(denne.hoyre);
        return denne.element;
    }

    private void skliVenstre( BinNode p ) {
        while (p != null) {
            stk.push(p);
            p=p.venstre;
        }
    }
}
```



```
public class PrefiksTreTravers implements Iterator {

    public prefiksTreTravers( BinNode rot ){
        stk.push(rot);
    }

    private Stakk stk = new StakkImplementasjon();

    private void skliVenstre( BinNode p ){
        while (p != null) {
            stk.push(p);  p=p.venstre;
        }
    }

    public boolean hasNext() {
        return ! stk.isEmpty();
    }

    public Object next() {
        BinNode denne = (BinNode) stk.top();
        if (denne.venstre != null)
            stk.push(denne.venstre);
        else {
            BinNOde foreldrenode = (BinNode) stk.pop();
            while (foreldrenode.hoyre == null) {
                if (stk.isEmpty()) return denne.element;
                foreldrenode = (BinNode) stk.pop();
            }
            stk.push(foreldrenode.hoyre);
        }
        return denne.element;
    }
}
```


Oppsummering

- **Binære søketrær**: problem med balansering
- **B-trær**: løsning for *store* trær
  - Minimere antall disk-operasjoner
  - Begrense dybden på datastrukturen
- **Rød-svarte trær**: løsning for balansering av binære trær
  - Fargingsreglene tvinger frem balanserte trær
  - *zig* og *zig-zag* rotasjoner
    $\Rightarrow$ Kan brukes på alle binære søketrær
- Brukes i Java til implementasjon av **kolleksjonsklasser**
  - Disse tilbyr iterator-støtte
  - Spesielt: hvordan implementere `next`