Dagens plan:

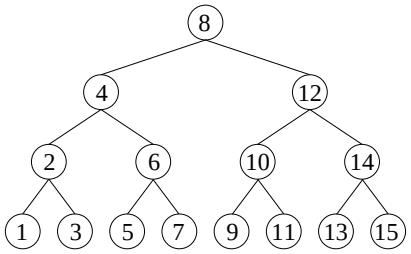
- Rød-svarte trær (kap. 12.2)
- B-trær (kap. 4.7)
- Abstrakte datatyper (kap. 3.1)
- Stakker (kap. 3.3)

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.

Eksempel på perfekt balansert tre:



Høyde: $\lfloor \log_2(N) \rfloor$

Rød-svarte trær

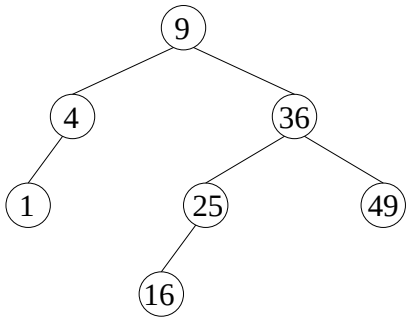
Et rød-svart tre er et binært søketre der hver node er farget enten rød eller svart slik at:

- 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:

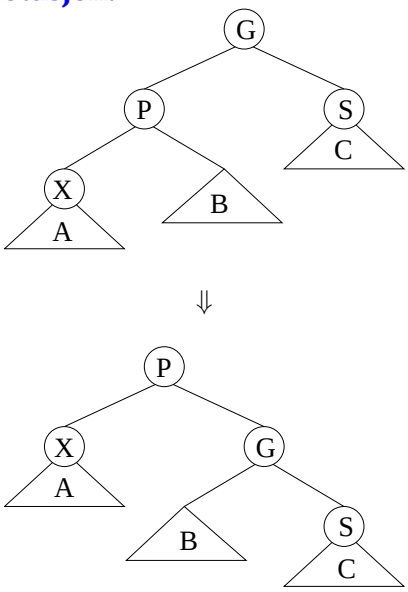


- Sett inn tallet 64 på riktig plass og med riktig farge.
- Forsøk så å sette inn tallet 81.

Rotasjoner på binære søketrær

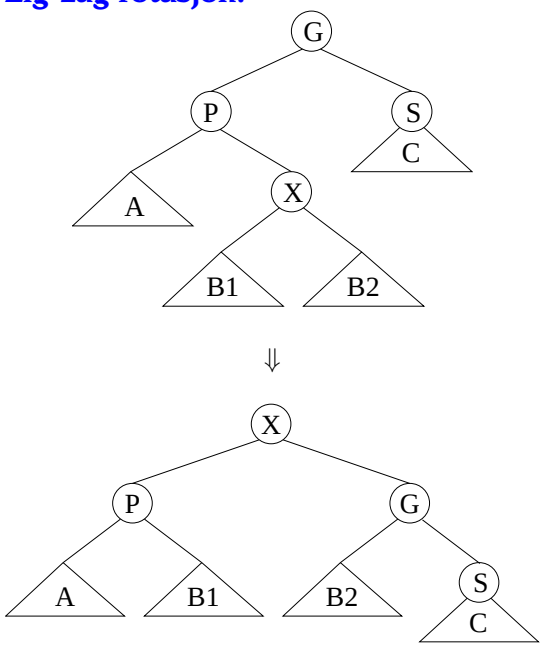
Gjelder generelt, ikke bare for rød-svarte trær!

Zig rotasjon:



+ symmetrisk tilfelle...

Zig-zag rotasjon:



+ 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:
   (a) Hvis **X** og **P** er begge venstre eller begge høyre barn: Gjør zig rotasjon med nødvendige fargeendringer.
   (b) Hvis **X** er venstre og **P** er høyre barn eller motsatt: Gjør zig-zag rotasjon med nødvendige fargeendringer.
   (c) Sett **X** til å være den nye roten i det roterte subtreet.
   (d) Hvis **X** er roten i selve treet: Farg denne svart.
   Ellers: Gjenta fra steg 2.

B-trær

- 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.

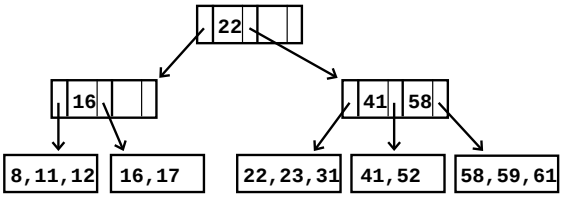
Definisjon: B-trær av orden M

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 B$^+$-trær. Tradisjonelle B-trær har data(pekere) i alle noder.

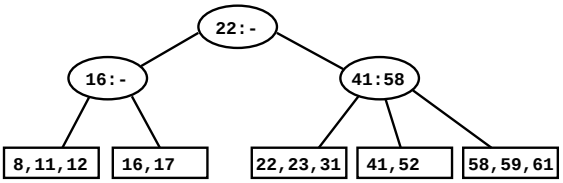
Eksempel

B-tre av orden 3 (også kalt et 2-3 tre):



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

Vi forenkler tegningene litt:



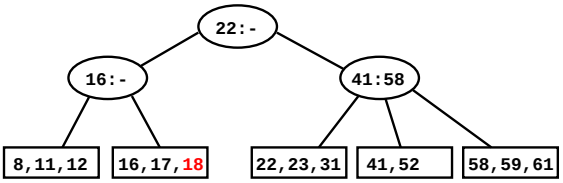
Søking etter element x

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.

Innsetting av element x

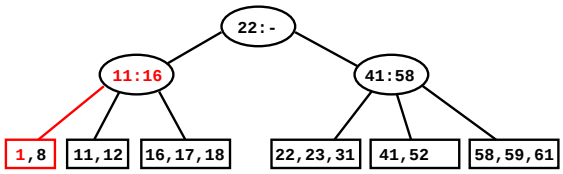
1. Let etter riktig bladnode for x (som for søking).
2. Dersom det er plass, setter vi inn x og oppdaterer eventuelt nøkkel-verdiene langs veien vi gikk.

   Eksempel: Sett inn 18 i treet i eksempelet over.



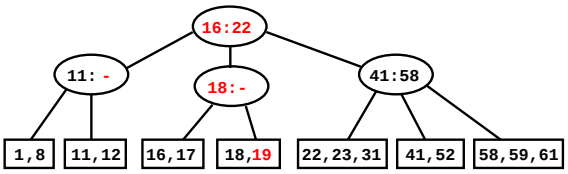
3. Dersom bladnoden er full, deler vi den i to og fordeler de $L+1$ nøklene jevnt på de to nye bladnodene.

   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.

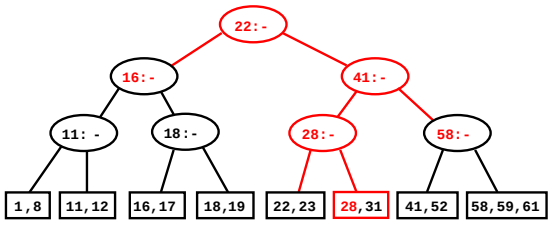
   Eksempel: Sett inn 19 i treet på forrige foil.



5. Dette kan medføre at vi til slutt må splitte roten i to. (Hvis roten får $M+1$ barn.) Da lager vi en ny rot med den gamle roten og den nye noden som barn.

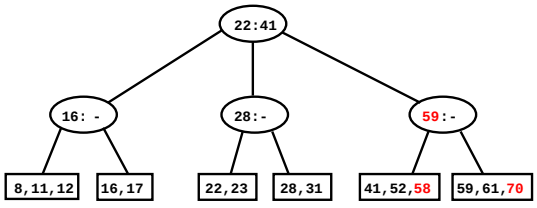
   Merk: Dette er den eneste måten et B-tre kan vokse i høyden på!

   Eksempel: Sett inn 28 i treet på forrige foil.



Sletting: eksempel

Fjern først 17, deretter 23 fra følgende tre:



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:
   (a) Dersom venstre (høyre) søsken har $\lceil L/2+1 \rceil$ elementer eller mer, flytter vi det største (minste) elementet til B.
   (b) 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.
- 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 binærsøk i sortert array) 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 $\ln 2 = 69\%$ fulle (samme fyllingsgrad som utvidbar hashing).

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, finn, ...

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.

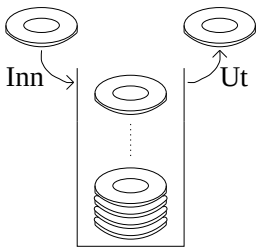
I Java er det naturlig å spesifisere en ADT som et `interface`.

Lister, stakker og køer

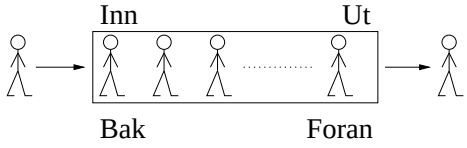
- Lister (kap. 3.2)

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

- Stakker (kap. 3.3)



- Køer (kap. 3.4)



Stakker

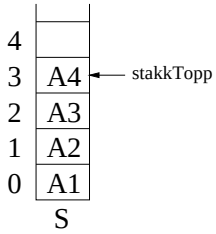
En variant av lister, der vi bare har lov til å sette inn og slette elementer fra en bestemt ende av listen.

```
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



Brukes ofte hvis antall elementer på stakken alltid er begrenset.

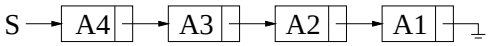
```
public void push(Object x) {
    stakkTopp++;
    S[stakkTopp] = x;
}
```

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

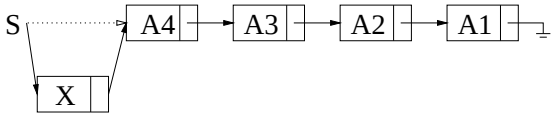
```
public Object top() {
    return S[stakktopp];
}
```

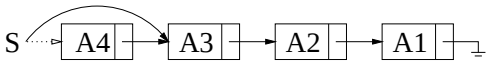
I tillegg: eventuell feilhåndtering.

Pekerkjede-implementasjon



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 på følgende måte:

- 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 + *