Balanserte trær:

Ofte er det ønskelig (for å få slikt som insert, remove/delete og search mest mulig effektive) at trær er noe/mye/fullstendig ***balansert***. Dvs. at noden(e) på det laveste nivået (nærmest rota) ikke har for stor forskjell ift. de(n) på det høyeste nivået (lengst unna rota). Hvor stor relativ forskjellen vi tillater, avgjør hvor *totalt balansert* treet er.

Noen algoritmer/metoder for å håndtere/sikre balanserte trær: **2-3-4 trær, Red-Black trær**, AVL trær, Splay trær, AA trær, Scapegoat trær og Treap.

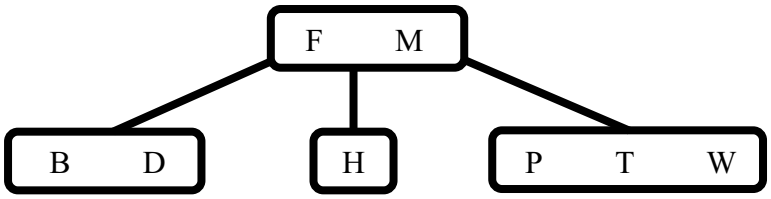
Vi skal se på de to førstnevnte algoritmene (i fet). Disse sikrer at nodene på *det høyeste nivået er max. dobbelt så langt unna rota som de på de laveste nivået*. I praksis er det ofte ikke *så* stor forskjell en gang, og tegner vi opp trærne ser de rimelig greie og balanserte ut.

Men *selve koden* for begge disse algoritmene er både litt lang og småtuklete. Derfor skal vi faktisk *ikke* se på konkret kode, men *kun* gjennomgå, studere og bruke *prinsippene for hvordan de virker*. (Noen Google-søk gir raskt ulike kodeforslag for begge algoritmene.)

2-3-4 trær:

Hittil har vi sett mye på *binære trær*. Disse kan også kalles *2-nodes trær*. Har noden derimot to keyer, og da tre barn/subtrær, kalles den 3-nodes. En 4-node har da tre keyer og fire barn/subtrær. Har vi et tre som altså kan bestå av *både* 2-noder, 3-noder og 4-noder, kaller vi dette et **2-3-4 tre** (eller sagt på en annen og mer generell måte: et *B-tre av orden 4*).

Eksempel på et slikt tre:

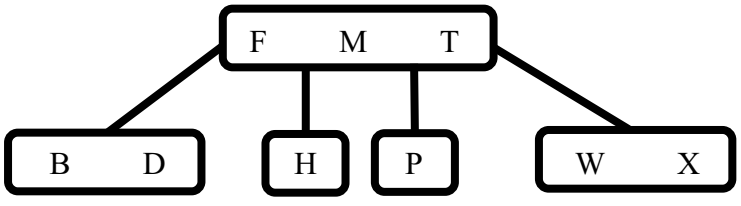


3-noden til venstre for 'F' har verdier som er *mindre eller lik* 'F'. 2-noden mellom 'F' og 'M' har verdier som er *større eller lik* 'F', og *mindre eller lik* 'M'. 4-noden til høyre for 'M' har verdier som er *større eller lik* 'M'. (**NB:** «insert» legger *alltid* inn *til høyre om verdien er lik*, men pga. splitting (se rett nedenfor) *kan* det bli like verdier på *begge sider* av en lik verdi.)

Skal vi nå legge inn en bokstav som er mindre enn 'F', er det plass til å legge den inn i noden med 'B' og 'D'. Dermed blir denne gjort om fra en 3-node til en 4-node. Det er plass til to bokstaver mellom 'F' og 'M' inn i noden med 'H', før denne er full og gjort om til en 4-node. Men, i 4-noden med 'P', 'T' og 'W' er det fullt! Der er det ikke plass til flere verdier, for vi skal jo ikke ha en 5-node! Hva gjør vi da? **Jo, vi må splitte!**

Skal vi f.eks. legge inn 'X', vil treet etterpå bli seende slik ut:

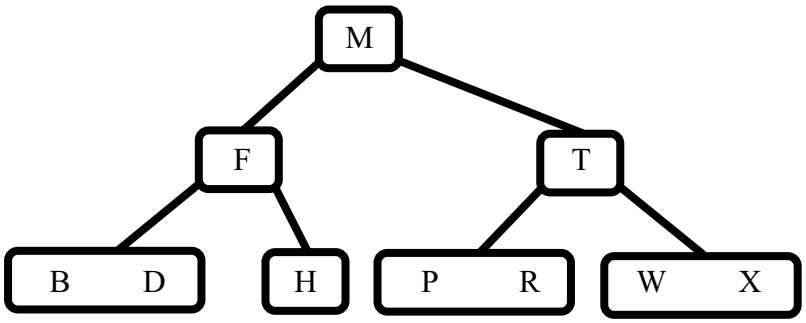
Eksempel på et slikt tre:



Dvs. *4-noden med 'P', 'T' og 'W' har* ***blitt splittet****, slik at 'P' og 'W' har* ***blitt to 2-noder****, og den* ***midterste 'T' er sendt om til mora!*** Dermed er det greit plass hos 2-noden 'W' til å få plass til 'X', slik at denne blir en 3-node.

<u>1.prinsipp:</u> Enhver 4-node man kommer til når man går nedover i treet for å sette inn en verdi, *må* splittes (i to 2-noder, og den midterste sendes opp til mora).

Skulle vi nå videre ha satt inn f.eks. 'R' i treet ovenfor, så ville det ha blitt seende slik ut:



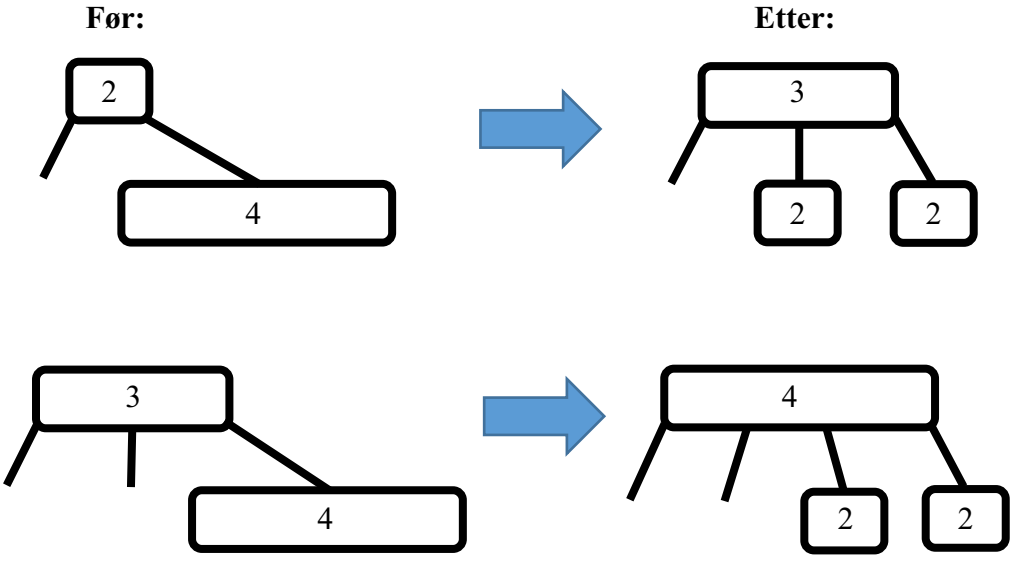
Dermed har rota (som var en 4-node, og skal splittes(!) blitt tre 2-noder. Den har ingen steder å sende den midterste bokstaven, derfor ***øker treets høyde med 1-en.***

<u>2.prinsipp:</u> Treets høyde øker *kun* (og *bare* da) med 1-en når rota splittes!

<u>3.prinsipp:</u> Pga. splitting når vi går nedover i treet, vil det *alltid* være plass i en ny verdi på det *aller nederste nivået* i treet (i en 2- eller 3-node). Det skal *aldri* lages en ny singel 2-node som alene legges på et nytt og høyere nivå enn alle de andre!

Vi ser at et slikt tre ser meget pent, jevnt og balansert ut!

Generelt vil splitting (av 4-noder) foregå slik (evt. det speilvendte av figurene):



Nodene (2-node eller 3-node) øverst i de venstre figurene *før* splitting av 4-noden nederst kan *aldri* være en 4-node, for ellers hadde den jo vært splittet allerede (på vei nedover i søket)!

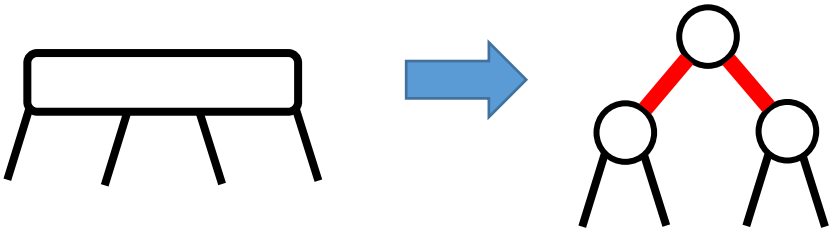
<u>Orden for et 2-3-4 tre med N noder:</u>

- ***Søk* trenger aldri å besøke mer enn lg N + 1 noder.**
- ***Innsetting* krever i *verste fall* færre enn lg N + 1 nodesplittinger, og later *gjennomsnittlig* ut til å kreve mindre enn *en* splitting.**

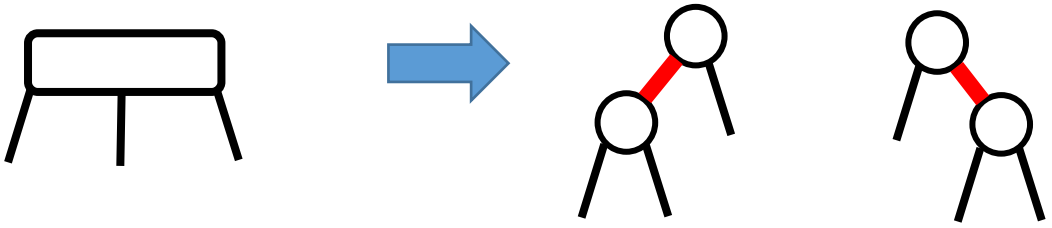
Men slik 2-3-4 trær kan håndteres/implementeres som binære trær!
Til dette har vi:

Red-Black trær:

En 4-node kan oppfattes/tegnes slik:

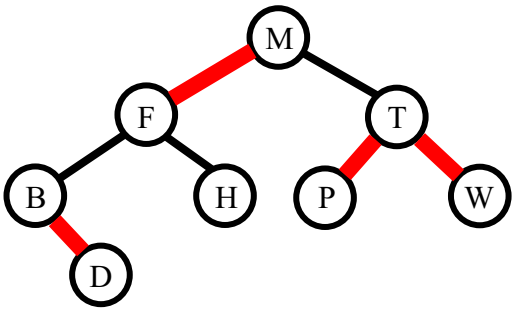


Mens en 3-node kan oppfattes/tegnes slik: eller slik:



Dvs. vi oppfatter/håndterer at en binær node *egentlig har en nær tilknytning til foreldre/mor (red link) eller ei (black link).* Dette *kan* kodingsmessig gjøres ved at hver node utvides med et «bit» (`int`, `bool`, `enum`) som angir om den er en del av mor («Red») eller ei («Black»).

Det første 2-3-4 treet med bokstaver ovenfor, *kunne* derfor ha vært tegnet Red-Black som:

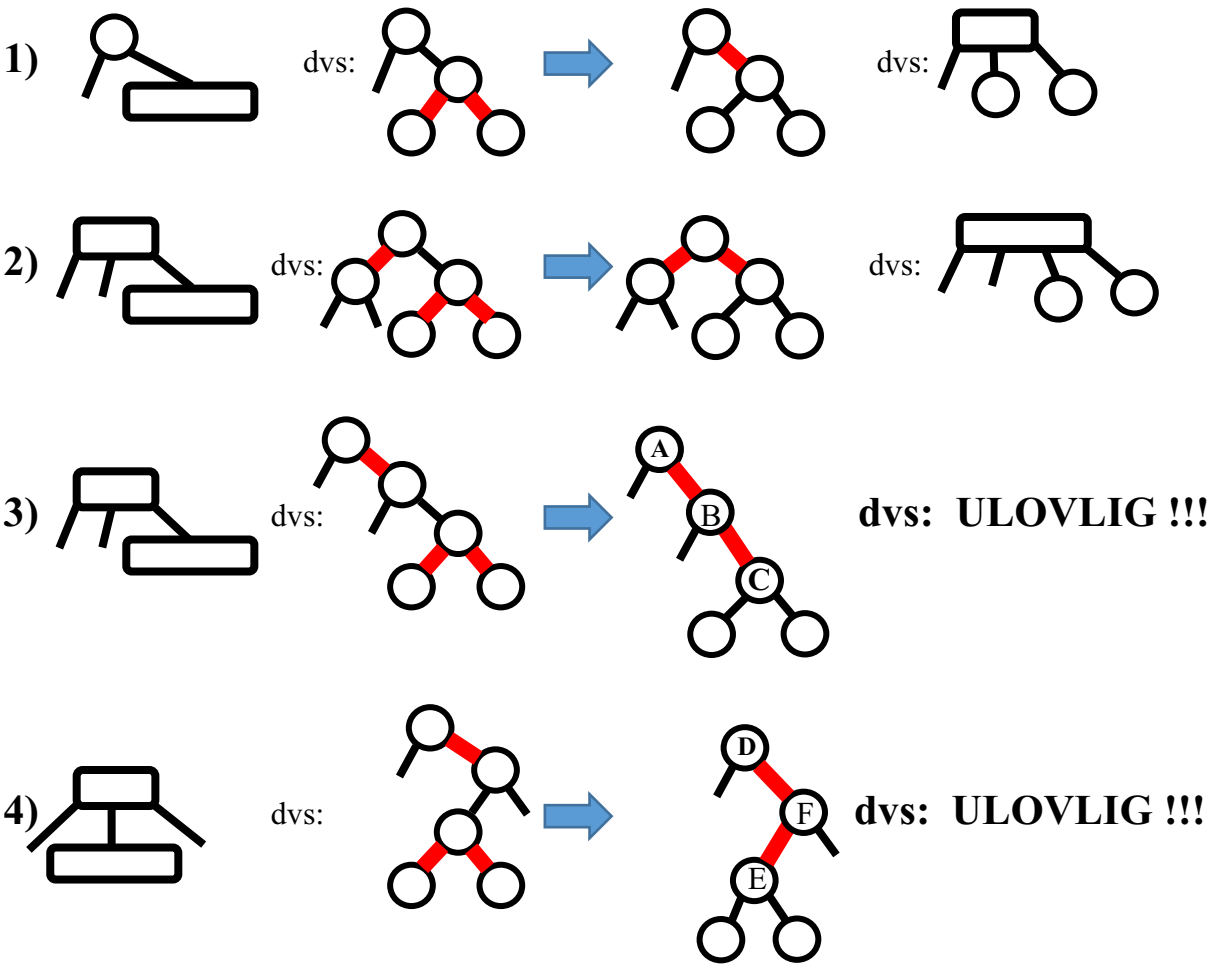


Her er 'FM' tegnet som venstre-rotert, men kunne like gjerne ha vært høyre-rotert (dvs. 'F' er foreldre/mor til 'M'). Likeledes med 'BD' (som er tegnet høyre-rotert). 'PTW' er en 4-node, og *må* tegnes slik.

<u>Noen egenskaper for Red-Black trær:</u>

1) **Det er *aldri to Red-linker rett etter hverandre på enhver sti* fra rota og ned til bladnodene.** (Ingen node kan være del av mor, og mor er del av bestemor også.)
2) **Alle stier har det samme antall Black-linker.** (Det ser vi enklest av det tilsvarende 2-3-4 treet, der alle bladnoder slutter likt, dvs. samme antall Black ned til dem.)
3) **Sti med annenhver gang Red- og Black link kan max. være dobbelt så lang som en sti med *kun* Black.** (Derfor er høyeste nivå max. det dobbelte av laveste.)
4) **Verdier/keyer kan, pga. splitting, havne på begge sider av en lik key!** (Derfor må en evt. 'search'-funksjon håndtere dette også.)

Splitting i et Red-Black tre (2-3-4 tre ytterst til venstre og høyre):

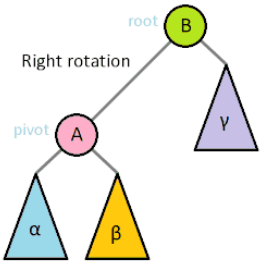


(Alle figurene ovenfor kunne også ha vært tegnet/forekommet speilvendt.)

I case 1 og 2 er 4-noden «rettvendt» ift. mor, og kan uten videre trygt bare splittes.

I case 3 og 4 får vi begge steder to Red-link etter hverandre, dette er ulovlig, og må rettes opp i før man leter videre nedover i treet. Dette gjør vi ved å gjøre om casene:

Case 3 → case 2: Dette får vi ved å venstre-rotere nodene omkring 'B'. Dvs. 'B' blir mor til 'A' og 'C', og begge har Red-linker opp til 'B'. Og dermed har vi nettopp case 2!



(se: http://upload.wikimedia.org/wikipedia/commons/3/31/Tree_rotation_animation_250x250.gif)

Case 4 → case 3 (som deretter gjøres om til case 2). Dette får vi ved å høyre-rotere omkring 'E', slik at 'D' blir mor til 'E', og 'E' blir mor til 'F' – altså case 3. Som igjen venstre-roteres til at 'E' blir mor til 'D' og 'F' (dvs. case 2).

Orden for et Red-Black tre med N noder (bygd av helt tilfeldige verdier):

- ***Søk* later *gjennomsnittlig* ut til å kreve omtrent lg N sammenligninger.**
- ***Innsetting* later *gjennomsnittlig* ut til å kreve mindre enn en rotasjon.**