# Trær, Binære Søketrær og AVL-trær

## IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

# Oversikt uke 35

# Oversikt uke 35

- Vi skal lære om binære søketrær
  - En datastruktur for raskt oppslag
- Vi skal lære om AVL-trær, som er selvbalanserende varianter av binære søketrær
  - Vi skal nevne rød-svarte trær, som er et alternativ til AVL-trær

# Trær

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som
  - en tom liste — ofte representert med `null` — eller
  - en node som består av en peker til et element og en peker til en liste
- Alle lister *er trær*, men ikke alle trær er lister
- Vi kan se på trær som en enkel utvidelse av lister
  - der vi tillater at en node har flere neste-pekere

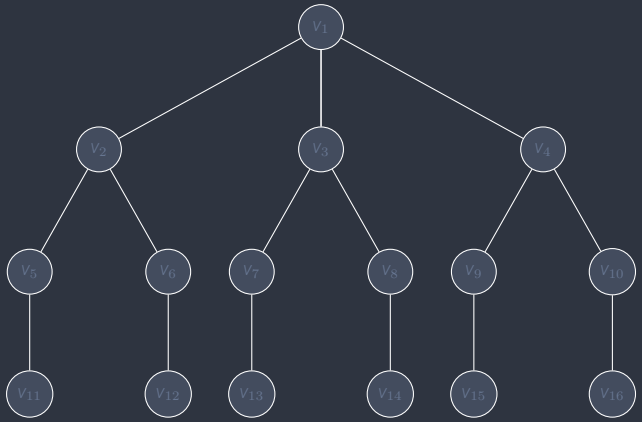
# Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*
- HTML er et filformat som lar deg uttrykke trær
  - Så en nettside er bare en bestemt måte å vise frem et tre
- Filsystemer er trær
- Alle mulige sjakkpartier kan representeres som et (enormt) tre

# Trær – definisjon

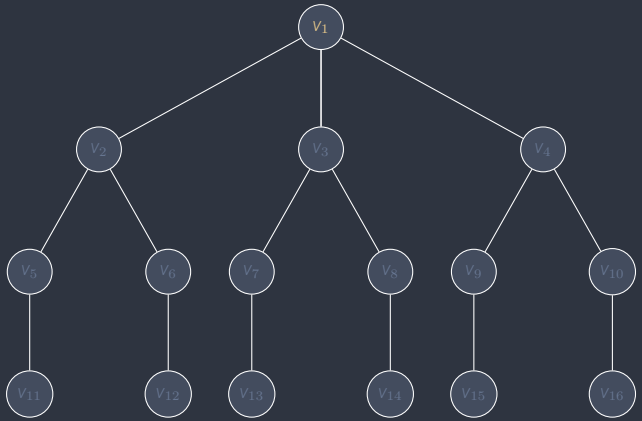
- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode
  - Et tre kan ikke inneholde sykler
    - Altså: fra en node *v* kan du ikke nå *v* ved å følge pekere fra *v*
- Merk: Boka tillater ikke tomme trær

# Trær – terminologi



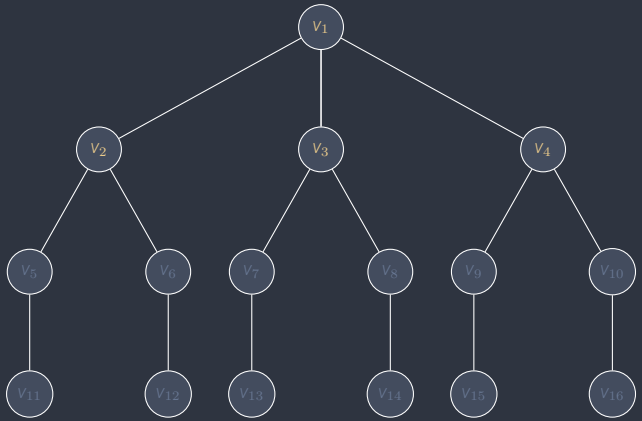

Dette er et tre, hver $v_i$ er en node

# Trær – terminologi



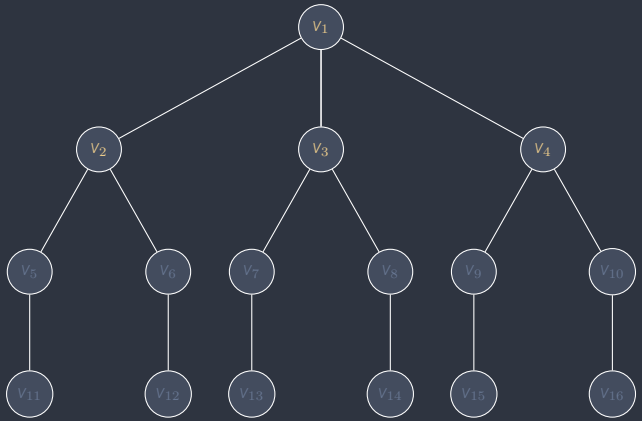

$v_1$ er *roten* av treet

# Trær – terminologi



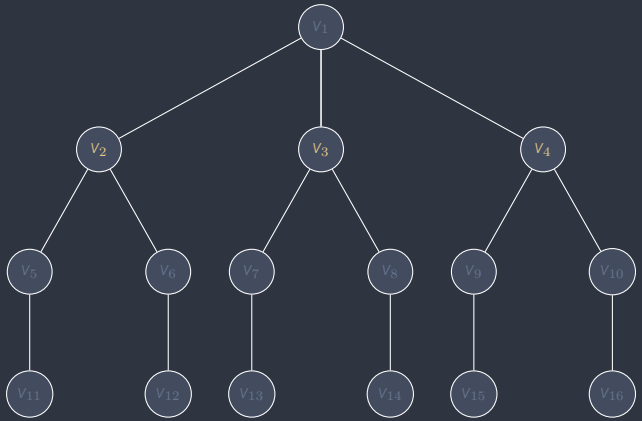

$v_2, v_3$ og $v_4$ er *barn* av $v_1$

# Trær – terminologi



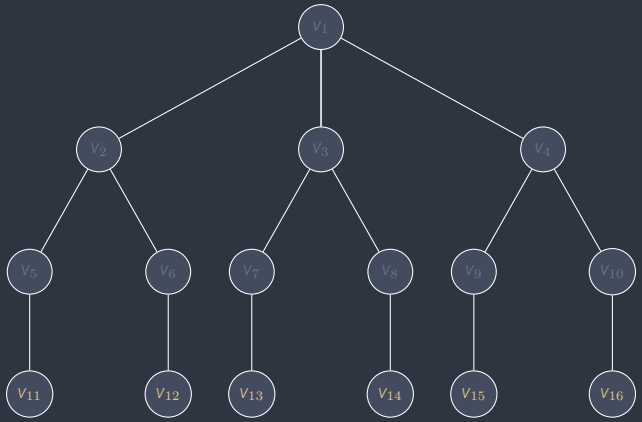

$v_1$ er *forelder* til $v_2, v_3$ og $v_4$

# Trær – terminologi




$v_2, v_3$ og $v_4$ er *søsken*

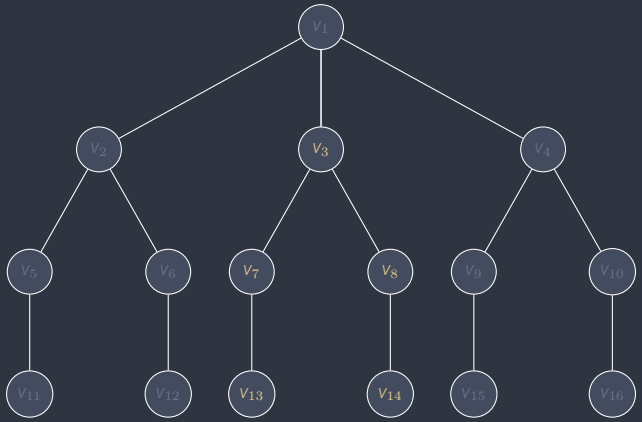
# Trær – terminologi




$v_{11}, \ldots, v_{16}$ er *løvnoder*, eller *eksterne* noder

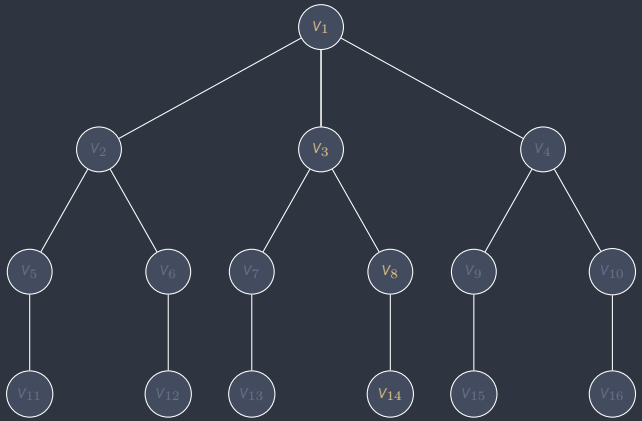
Nodene $v_1, \ldots, v_{10}$ er ikke løvnoder, eller *interne* noder

# Trær – terminologi



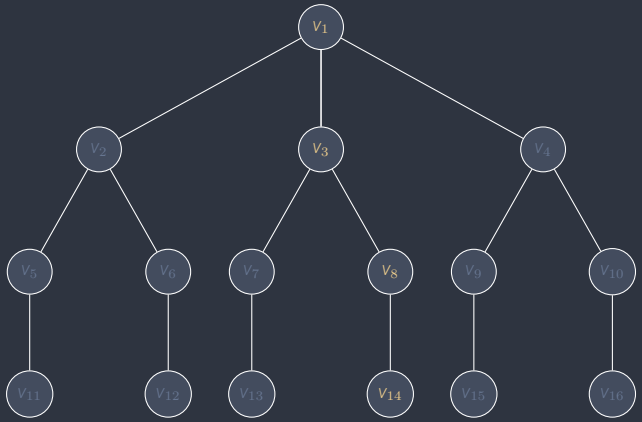

$v_3$, $v_7$, $v_8$, $v_{13}$ og $v_{14}$ utgjør et *subtre*, hvor $v_3$ er roten

# Trær – terminologi



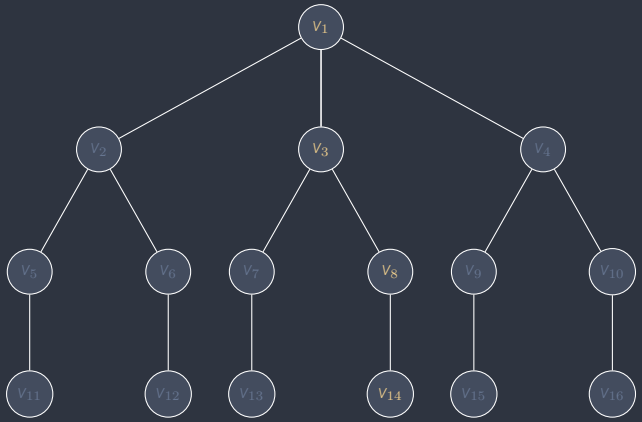

$v_1, v_3, v_8$ og $v_{14}$ er *forfedre* av $v_{14}$

# Trær – terminologi




$v_1, v_3, v_8$ og $v_{14}$ er *etterkommere* av $v_1$

# Trær – terminologi




Sekvensen $v_1, v_3, v_8, v_{14}$ kalles en *sti*

# Trær – Datastruktur

- `null` representerer et tomt tre
- Anta at *v* er en node, da gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`parent` foreldrenoden til *v*
  - *v*.`children` barnenodene til *v*

# Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde $0$
- Siden vi tillater et tomt tre gir vi det dybde $-1$

**Algorithm:** Finn dybden av en gitt node

```
Input: En node v
Output: Dybden av noden
Procedure Depth(v)
    if v = null then
        return −1
    return 1 + Depth(v.parent)
```

# Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

**Algorithm:** Finn høyden av en gitt node

```
Input: En node v
Output: Høyden av noden
1 Procedure height(v)
2 |   h ← −1
3 |   if v = null then
4 |   |   return h
5 |   for v′ ∈ v.children do
6 |   |   h ← Max(h, height(v′))
7 |   return 1 + h
```

# Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre
- For mange operasjoner har *rekkefølgen* vi utfører operasjonen i en betydning
  - *preorder* utfører operasjonen på seg selv først, og barna etterpå
  - *postorder* utfører operasjonen på barna først, og seg selv etterpå
- For å kopiere et tre kan man bruke *preorder*, men ikke *postorder*
- For å slette et tre kan man bruke *postorder*, men ikke *preorder*

# Trær – preorder og postorder

**Algorithm:** Preorder traversering

```
Input: En node v (som ikke er null)
Output: Utfør en operasjon på v først og barna til v etterpå
1 Procedure Preorder(v)
2 |   Operate on v
3 |   for c ∈ v.children do
4 |   |   Preorder(c)
```

**Algorithm:** Postorder traversering

```
Input: En node v (som ikke er null)
Output: Utfør en operasjon på barna til v først og v etterpå
1 Procedure Postorder(v)
2 |   for c ∈ v.children do
3 |   |   Postorder(c)
4 |   Operate on v
```

# Binære søketrær

# Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn
- Hvis *v* er en node i et binærtre, så gir
  - *v*.`element` dataen som er lagret i noden
  - *v*.`left` venstre barn av *v*
  - *v*.`right` høyre barn av *v*

# Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.`element`
    - større enn alle elementer i venstre subtre, og
    - mindre enn alle elementer i høyre subtre
- Merk at vi kan si større *eller lik* dersom vi ønsker å tillate duplikater
- For at vi skal kunne bruke binære søketrær må elementene være *sammenlignbare*
- Binære trær er spesielt gode når de er *balanserte*
  - Dette er tema for neste uke

# Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $\mathcal{O}(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*
  - En datastruktur hvor vi stadig legger til og fjerner elementer
- Et binært søketre er en datastruktur
  - som gjør binærsøk *enkelt*
  - støtter effektiv innsetting og sletting

# Innsetting

**Algorithm:** Innsetting i et binært søketre

**Input:** En node $v$ og et element $x$
**Output:** En oppdatert node $v$ der en node som inneholder $x$ er en etterkommer av $v$

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     return v
```

- Denne algoritmen har kompleksitet $\mathcal{O}(h)$
  - der $h$ er høyden på treet
- Dersom $n$ er antall noder i treet har vi $\mathcal{O}(n)$ i verste tilfelle
  - men hvis treet er balansert,
  - så er kompleksiteten $\mathcal{O}(log(n))$

# Oppslag

**Algorithm:** Oppslag i et binært søketre

**Input:** En node $v$ og et element $x$
**Output:** Dersom $x$ forekommer i en node $u$ som en etterkommer av $v$, returner $u$, ellers `null`.

```
1 Procedure Search(v, x)
2     if v = null then
3         return null
4     if v.element = x then
5         return v
6     if x < v.element then
7         return Search(v.left, x)
8     if x > v.element then
9         return Search(v.right, x)
```

- Oppslag i et binærtre har samme kompleksitet som innsetting

# Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle ”hull”
- Vi skiller mellom tre tilfeller
  - Noden vi vil slette har ingen barn
  - Noden vi vil slette har ett barn
  - Noden vi vil slette har to barn

# Finn minste

- For sletting trenger vi en prosedyre for å finne minste element

**Algorithm:** Finn minste node

**Input:** En node $v$
**Output:** Returner noden som inneholder den minste etterkommeren av $v$

```
1 Procedure FindMin(v)
2 |   Etterlatt som øvelse!
```

# Sletting

**Algorithm:** Slett en node i et binært søketre

```
Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6          return v
7      if x > v.element then
8          v.right ← Remove(v.right, x)
9          return v
10     if v.left = null then
11         return v.right
12     if v.right = null then
13         return v.left
14     u ← FindMin(v.right)
15     v.element ← u.element
16     v.right ← Remove(v.right, u.element)
17     return v
```

# AVL-TRÆR

# AVL-TRÆR

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær
- I *tillegg* må de oppfyle følgende egenskap:
  - for hver node i et AVL-tre, så må høydeforskjellen på venstre og høyre subtre være mindre eller lik $1$
- Denne invarianten må opprettholdes ved innsetting og sletting
  - (oppslag er helt uforandret)

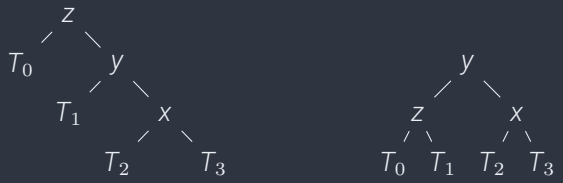
# Høyde

- Fordi vi ofte vil referere til høyden i treet, utvider vi nodene i treet
- Hvis $v$ er en node i et AVL-tre, så gir
  - $v$.`element` dataen som er lagret i noden
  - $v$.`left` venstre barn av $v$
  - $v$.`right` høyre barn av $v$
  - $v$.`height` høyden til $v$
- Husk at høyden til et tomt tre er definert som $-1$
  - og at høyden til en node er én mer enn sitt høyeste subtre
- Ved innsetting og sletting må vi vedlikeholde høydene
- Vi antar at vi har en prosedyre `Height` som:
  - returnerer $-1$ dersom den får `null` som input
  - returnerer $v$.`height` for alle noder $v$

# Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær
- Etter operasjonen er utført, balanserer vi hver node lokalt fra der operasjonen ble utført og opp til roten (hvis det er nødvendig)
  - Vi balanserer når høydeforskjellen mellom venstre og høyre subtre er mer enn $1$
- For å balansere en node, vil vi anvende *rotasjoner*
- Underveis må vi passe på å oppdatere høydene
- Husk at AVL innsetting og sletting bare fungerer på AVL-trær!
  - Altså kan vi anta at treet ikke har høydeforskjeller større enn $1$
  - Ved én innsetting eller sletting i et AVL-tre vil vi bare forårsake en midlertidig høydeforskjell på $2$

# Rotasjoner – venstrerotasjon

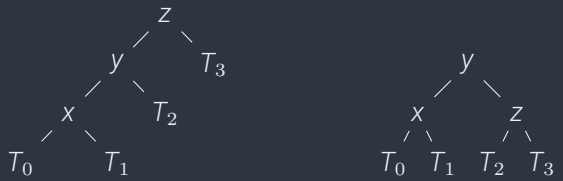



**Algorithm:** Venstrerotasjon av et Binærtre

**Input:** En node $z$
**Output:** Roter treet til venstre, slik at $y$ blir den nye roten

```
1 Procedure LeftRotate(z)
2     y ← z.right
3     T1 ← y.left
4     y.left ← z
5     z.right ← T1
6     z.height ← 1 + Max(Height(z.left), Height(z.right))
7     y.height ← 1 + Max(Height(y.left), Height(y.right))
8     return y
```

# Rotasjoner – høyrerotasjon



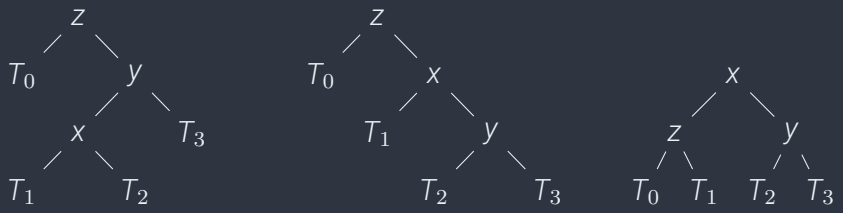

**Algorithm:** Høyrerotasjon av et Binærtre

**Input:** En node $z$
**Output:** Roter treet til høyre, slik at $y$ blir den nye roten

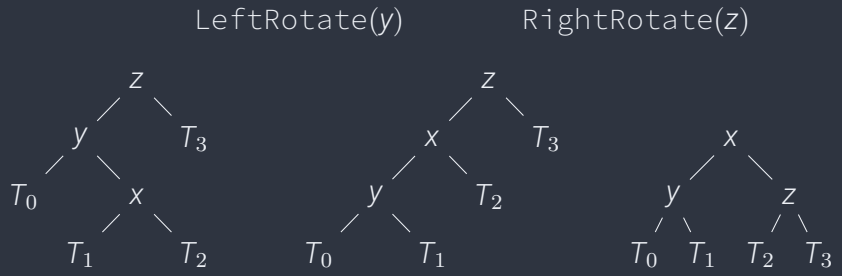
```
1 Procedure RightRotate(z)
2     y ← z.left
3     T2 ← y.right
4     y.right ← z
5     z.left ← T2
6     z.height ← 1 + Max(Height(z.left), Height(z.right))
7     y.height ← 1 + Max(Height(y.left), Height(y.right))
8     return y
```

# Rotasjoner – doble rotasjoner

`RightRotate(`$y$`)` `LeftRotate(`$z$`)`



# Rotasjoner – doble rotasjoner



# Balansefaktor

**Algorithm:** Balansefaktoren av en node

**Input:** En node $v$
**Output:** Returner høydeforskjellen på $v$ sitt venstre- og høyrebarn

```
1 Procedure BalanceFactor(v)
2     if v = null then
3         return 0
4     return Height(v.left) − Height(v.right)
```

- En hjelpeprosedyre som sier hvor vestre- eller høyretungt $v$ er
- $0$ betyr at $v$ er balansert
- Et positivt tall betyr at treet er venstretungt
- Et negativt tall betyr at treet er høyretungt

# Balansering

**Algorithm:** Balansering av et AVL-tre

```
Input: En node v
Output: En balansert node
Procedure Balance(v)
    if BalanceFactor(v) < −1 then
        if BalanceFactor(v.right) > 0 then
            v.right = RightRotate(v.right)
        return LeftRotate(v)
    if BalanceFactor(v) > 1 then
        if BalanceFactor(v.left) < 0 then
            v.left = LeftRotate(v.left)
        return RightRotate(v)
    return v
```

# Innsetting

**Algorithm:** Innsetting i et AVL-tre

**Input:** En node $v$ og et element $x$
**Output:** En oppdatert node $v$ der en node som inneholder $x$ er en etterkommer av $v$

```
1 Procedure Insert(v, x)
2     if v = null then
3         v ← new Node(x)
4     else if x < v.element then
5         v.left ← Insert(v.left, x)
6     else if x > v.element then
7         v.right ← Insert(v.right, x)
8     v.height ← 1 + Max(Height(v.left), Height(v.right))
9     return Balance(v)
```

# Sletting

**Algorithm:** Sletting i et AVL-tre

**Input:** En node $v$ og et element $x$
**Output:** Dersom $x$ forekommer i en node $u$ som en etterkommer av $v$, fjern $u$.

```
1  Procedure Remove(v, x)
2      if v = null then
3          return null
4      if x < v.element then
5          v.left ← Remove(v.left, x)
6      else if x > v.element then
7          v.right ← Remove(v.right, x)
8      else if v.left = null then
9          v ← v.right
10     else if v.right = null then
11         v ← v.left
12     else
13         u ← FindMin(v.right)
14         v.element ← u.element
15         v.right ← Remove(v.right, u.element)
16     v.height ← 1 + Max(Height(v.left), Height(v.right))
17     return Balance(v)
```

# Rød-svarte trær

# Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag
  - De anvender begge rotasjoner for å bevare et krav om balanse
- De store forskjellene mellom rød-svarte- og AVL-trær er:
  - Rød-svarte trær har *svakere* krav om balanse enn AVL-trær
  - Rød-svarte trær bruker mindre minne, siden vi ikke trenger å lagre høydene
  - Rød-svarte trær bruker færre rotasjoner

# HVEM ER BEST!?

- Rød-svarte trær er raskere enn AVL-trær når innsetting og sletting forekommer ofte, til sammenligning med oppslag
  - Dette er fordi rød-svarte trær trenger færre rotasjoner
- AVL-trær er raskere enn rød-svarte trær når oppslag forekommer ofte, til sammenligning med innsetting og sletting
  - Dette er fordi AVL-trær er mer balanserte

# Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart
3. En rød node kan ikke ha et rødt barn
4. Hver gren fra roten til et tomt tre (eller `null`) inneholder *like mange svarte noder*

# Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare svarte noder
  - en annen gren med annenhver svarte og røde noder
- Da har vi en gren som er dobbelt så lang som en annen!
  - Men dobbelt så langt er ikke så mye lenger
  - Så vi bevarer $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag