# Trær

## IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

# Trær

# Trær

- Det er to store kategorier med anvendelser av trær:

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger (eksempelvis mengder og ordbøker)

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger (eksempelvis mengder og ordbøker)
- Dere kjenner allerede til lister, som er definert som

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger (eksempelvis mengder og ordbøker)
- Dere kjenner allerede til lister, som er definert som
  - en tom liste — ofte representert med `null` — eller

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger (eksempelvis mengder og ordbøker)
- Dere kjenner allerede til lister, som er definert 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

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger (eksempelvis mengder og ordbøker)
- Dere kjenner allerede til lister, som er definert 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

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger (eksempelvis mengder og ordbøker)
- Dere kjenner allerede til lister, som er definert 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

# Trær

- Det er to store kategorier med anvendelser av trær:
  - Det du jobber med har en iboende hierarkisk struktur
  - Effektiv implementasjon for datasamlinger (eksempelvis mengder og ordbøker)
- Dere kjenner allerede til lister, som er definert 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

# Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
  - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*

# 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

# 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 kan ses på som en bestemt måte å vise frem et tre

# 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 kan ses på som en bestemt måte å vise frem et tre
  - (Dette treet kalles DOM-treet, der DOM står for Document Object Model)

# 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 kan ses på som en bestemt måte å vise frem et tre
  - (Dette treet kalles DOM-treet, der DOM står for Document Object Model)
- En stamtavle (eller slektstre eller familietre) viser etterfølgerne til en person

# 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 kan ses på som en bestemt måte å vise frem et tre
  - (Dette treet kalles DOM-treet, der DOM står for Document Object Model)
- En stamtavle (eller slektstre eller familietre) viser etterfølgerne til en person
- Filsystemer er trær

# 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 kan ses på som en bestemt måte å vise frem et tre
  - (Dette treet kalles DOM-treet, der DOM står for Document Object Model)
- En stamtavle (eller slektstre eller familietre) viser etterfølgerne til en person
- Filsystemer er trær
- Alle mulige sjakkpartier kan representeres som et (enormt) tre

# Trær – definisjon

- Et tre er definert som

# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller

# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til

# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element

# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og

# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode (med mindre $v$ er roten)

# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode (med mindre $v$ er roten)
  - Et tre kan ikke inneholde sykler

# Trær – definisjon

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

# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode (med mindre $v$ er roten)
  - 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 – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode (med mindre $v$ er roten)
  - 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
- Vi skal definere trær mer presist som en spesiell type grafer

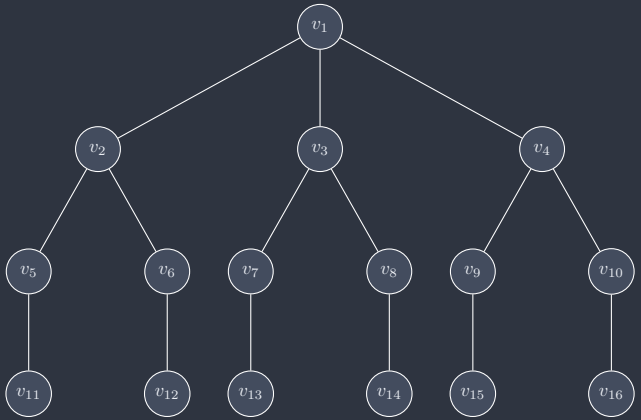
# Trær – definisjon

- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode (med mindre $v$ er roten)
  - 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
- Vi skal definere trær mer presist som en spesiell type grafer
  - Da vil disse trærne kalles trær med *rot*

# Trær – definisjon

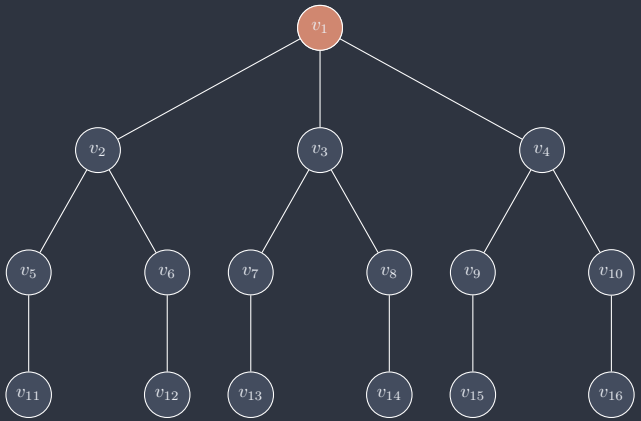
- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node $v$ med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én foreldernode (med mindre $v$ er roten)
  - Et tre kan ikke inneholde sykler
    - Altså: fra en node $v$ kan du ikke nå $v$ ved å følge pekere fra $v$
- Merk: Boka tillater ikke tomme trær
- Vi skal definere trær mer presist som en spesiell type grafer
  - Da vil disse trærne kalles trær med *rot*
- En samling trær kalles en skog

# Trær – terminologi



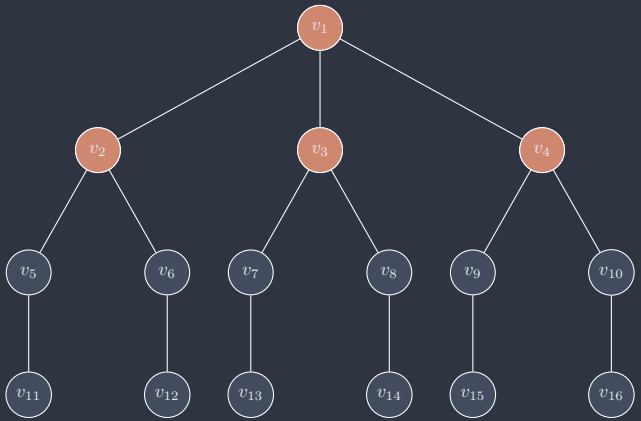

Dette er et tre, hver $v_i$ er en node

# Trær – terminologi



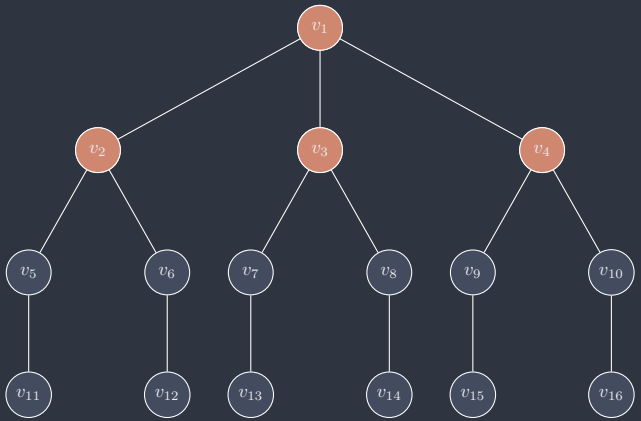

$v_1$ er *roten* av treet

# Trær – terminologi



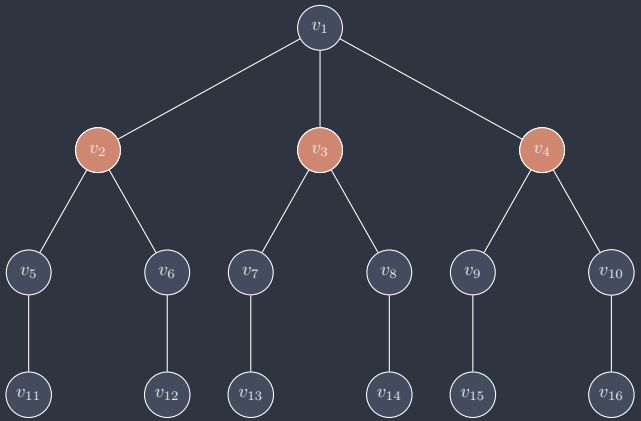

$v_2, v_3$ og $v_4$ er *barn* av $v_1$

# Trær – terminologi



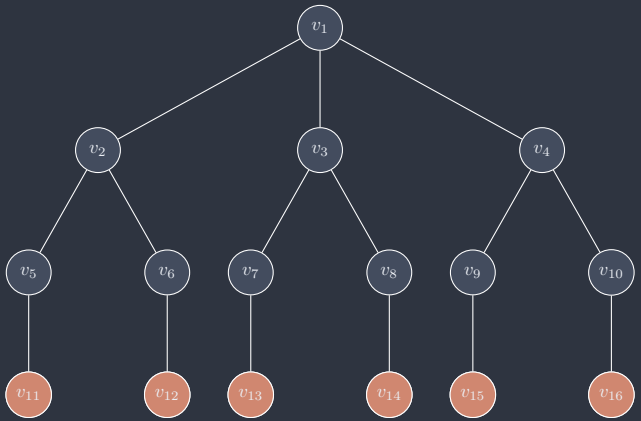

$v_1$ er *forelder* til $v_2, v_3$ og $v_4$

# Trær – terminologi




$v_2, v_3$ og $v_4$ er *søsken*

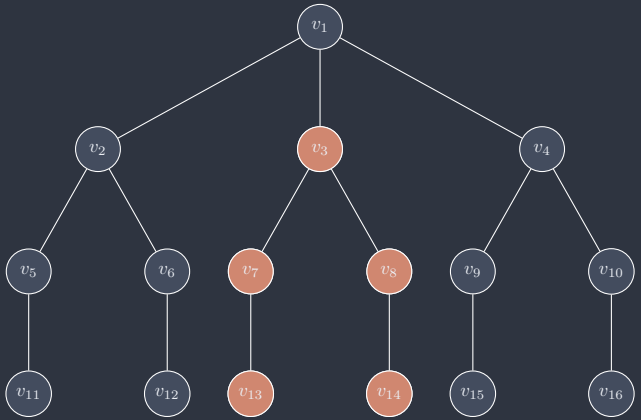
# Trær – terminologi




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

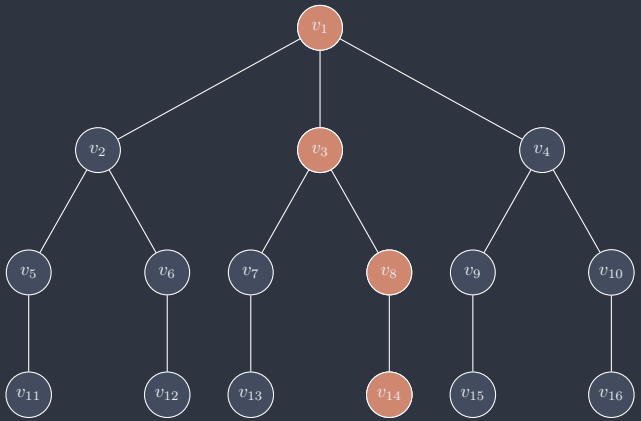
Nodene $v_1, \ldots, v_{10}$ er ikke løvnoder, eller *interne* noder

# Trær – terminologi



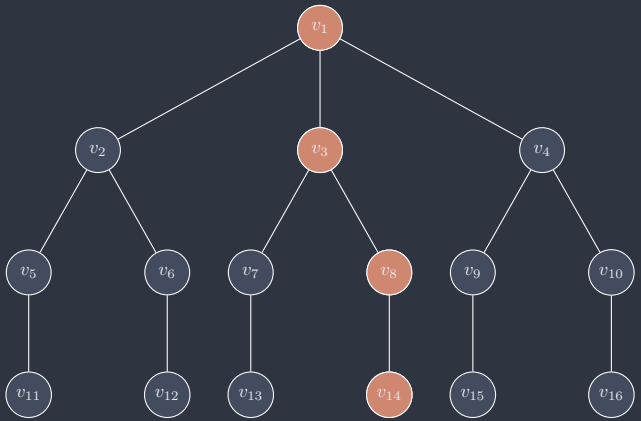

$v_3$, $v_7$, $v_8$, $v_{13}$ og $v_{14}$ utgjør et *subtre*, hvor $v_3$ er roten

# Trær – terminologi



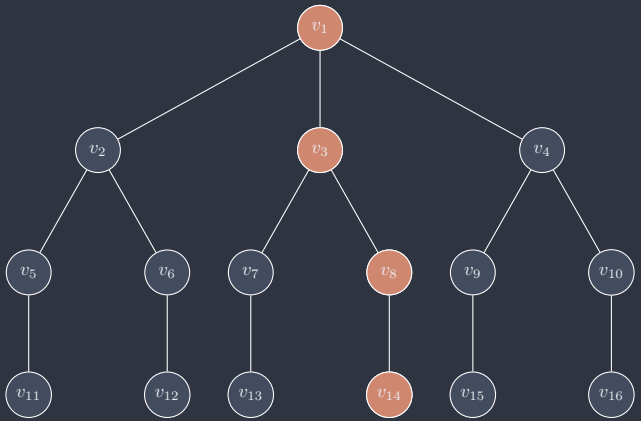

$v_1, v_3, v_8$ og $v_{14}$ er *forfedre* av $v_{14}$

# 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

# Trær – Datastruktur

- `null` representerer et tomt tre
- Anta at $v$ er en node, da gir

# Trær – Datastruktur

- `null` representerer et tomt tre
- Anta at $v$ er en node, da gir
  - $v$.`element` dataen som er lagret i noden

# 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 $v$

# 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

# Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde $0$

# 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 $-1$

# 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 $-1$

# 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

# 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

```
1 Procedure Depth(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

```
1 Procedure Depth(v)
2     if v = null then
3         return −1
```

# 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 $-1$

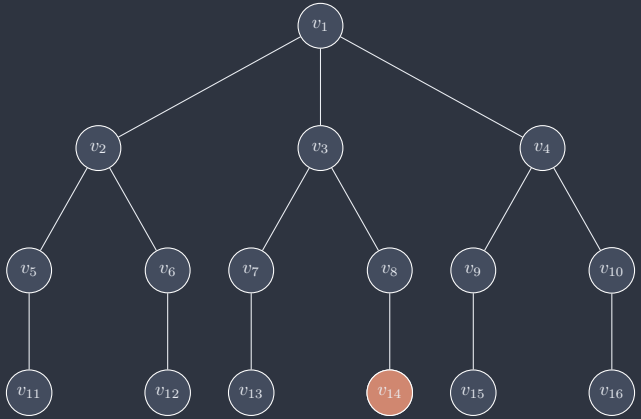
**Algorithm: Finn dybden av en gitt node**

**Input:** En node $v$
**Output:** Dybden av noden

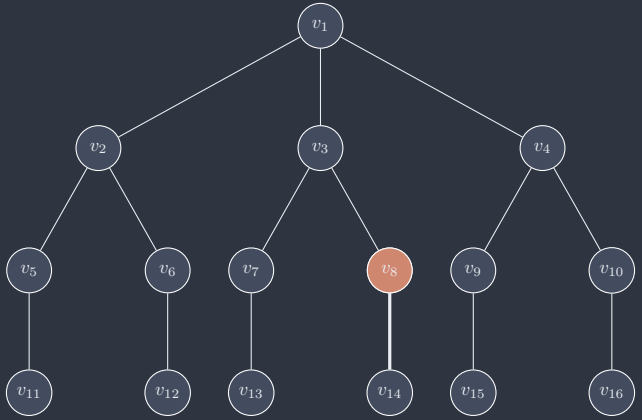
```
1 Procedure Depth(v)
2     if v = null then
3         return −1
4     return 1 + Depth(v.parent)
```

# Trær – dybde (eksempel)

`Depth`$(v_{14})$

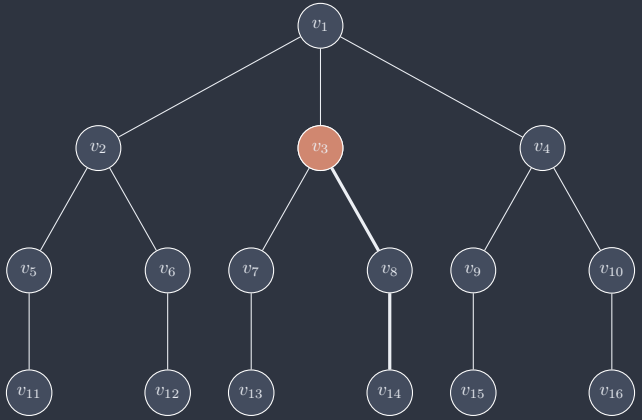


# Trær – dybde (eksempel)



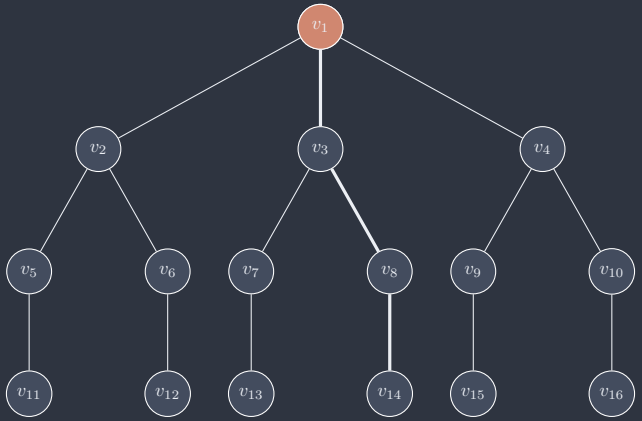

$\texttt{Depth}(v_{14})$
$= (1 + \texttt{Depth}(v_8))$

# Trær – dybde (eksempel)



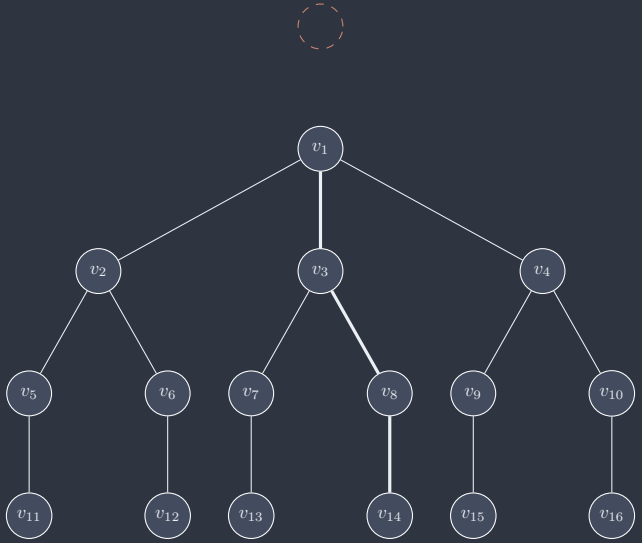

$$\begin{aligned}&\texttt{Depth}(v_{14})\\&= (1 + \texttt{Depth}(v_8))\\&= (1 + (1 + \texttt{Depth}(v_3)))\end{aligned}$$

# Trær – dybde (eksempel)



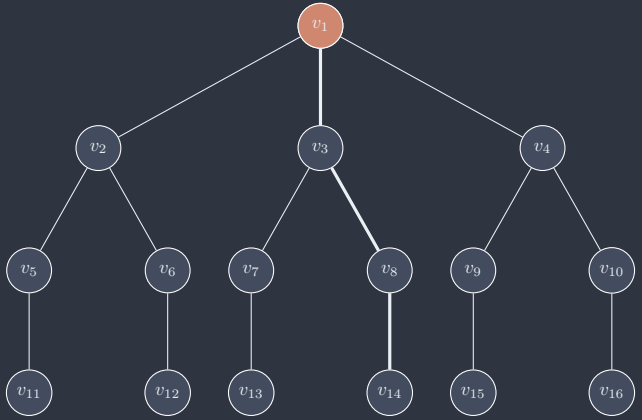

$$\begin{aligned}&\texttt{Depth}(v_{14})\\&= (1 + \texttt{Depth}(v_8))\\&= (1 + (1 + \texttt{Depth}(v_3)))\\&= (1 + (1 + (1 + \texttt{Depth}(v_1))))\end{aligned}$$

# Trær – dybde (eksempel)



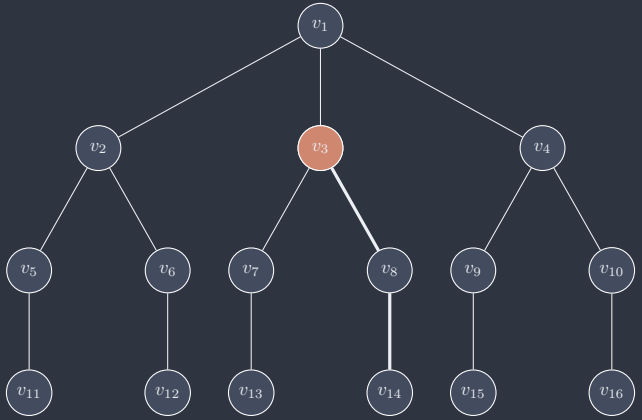

$$
\begin{aligned}
&\texttt{Depth}(v_{14}) \\
&= (1 + \texttt{Depth}(v_8)) \\
&= (1 + (1 + \texttt{Depth}(v_3))) \\
&= (1 + (1 + (1 + \texttt{Depth}(v_1)))) \\
&= (1 + (1 + (1 + (1 + \texttt{Depth}(\textbf{null})))))
\end{aligned}
$$

# Trær – dybde (eksempel)



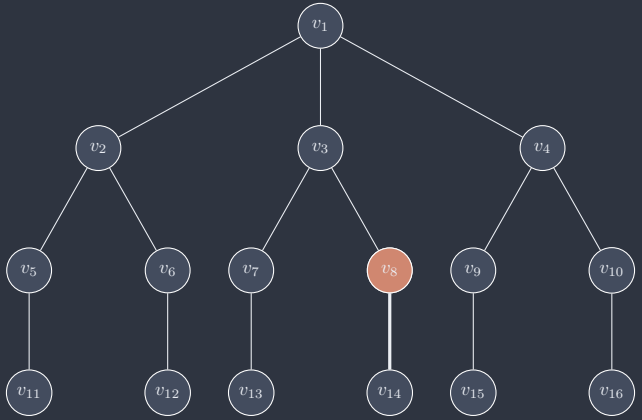

$$
\begin{aligned}
&\texttt{Depth}(v_{14}) \\
&= (1 + \texttt{Depth}(v_8)) \\
&= (1 + (1 + \texttt{Depth}(v_3))) \\
&= (1 + (1 + (1 + \texttt{Depth}(v_1)))) \\
&= (1 + (1 + (1 + (1 + \texttt{Depth}(\textbf{null}))))) \\
&= (1 + (1 + (1 + (1 + -1))))
\end{aligned}
$$

# Trær – dybde (eksempel)



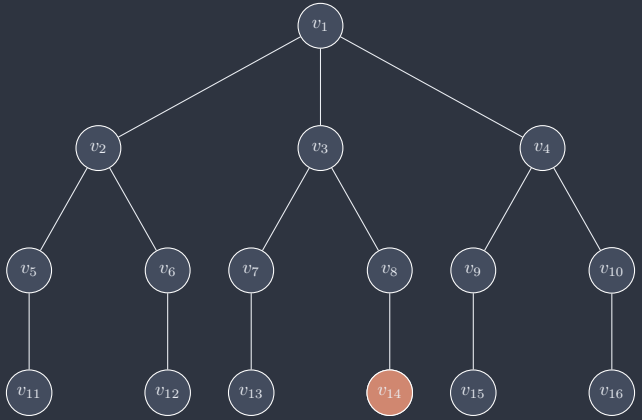

$$
\begin{aligned}
&\texttt{Depth}(v_{14}) \\
&= (1 + \texttt{Depth}(v_8)) \\
&= (1 + (1 + \texttt{Depth}(v_3))) \\
&= (1 + (1 + (1 + \texttt{Depth}(v_1)))) \\
&= (1 + (1 + (1 + (1 + \texttt{Depth}(\textbf{null}))))) \\
&= (1 + (1 + (1 + (1 + -1)))) \\
&= (1 + (1 + (1 + 0)))
\end{aligned}
$$

# Trær – dybde (eksempel)



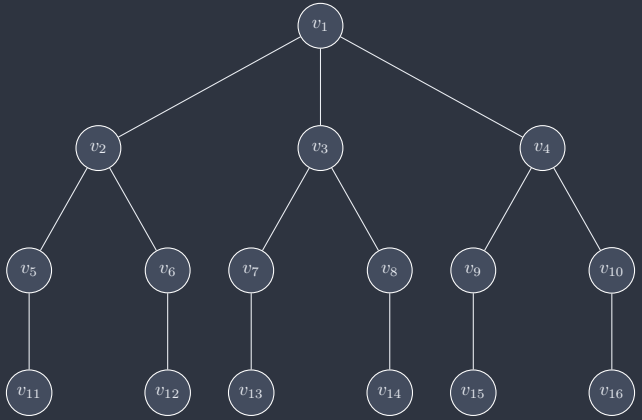

$$
\begin{aligned}
&\texttt{Depth}(v_{14}) \\
&= (1 + \texttt{Depth}(v_8)) \\
&= (1 + (1 + \texttt{Depth}(v_3))) \\
&= (1 + (1 + (1 + \texttt{Depth}(v_1)))) \\
&= (1 + (1 + (1 + (1 + \texttt{Depth}(\textbf{null}))))) \\
&= (1 + (1 + (1 + (1 + -1)))) \\
&= (1 + (1 + (1 + 0))) \\
&= (1 + (1 + 1))
\end{aligned}
$$

# Trær – dybde (eksempel)




$$
\begin{aligned}
&\texttt{Depth}(v_{14}) \\
&= (1 + \texttt{Depth}(v_8)) \\
&= (1 + (1 + \texttt{Depth}(v_3))) \\
&= (1 + (1 + (1 + \texttt{Depth}(v_1)))) \\
&= (1 + (1 + (1 + (1 + \texttt{Depth}(\textbf{null}))))) \\
&= (1 + (1 + (1 + (1 + -1)))) \\
&= (1 + (1 + (1 + 0))) \\
&= (1 + (1 + 1)) \\
&= (1 + 2)
\end{aligned}
$$

# Trær – dybde (eksempel)




$$
\begin{aligned}
&\texttt{Depth}(v_{14}) \\
&= (1 + \texttt{Depth}(v_8)) \\
&= (1 + (1 + \texttt{Depth}(v_3))) \\
&= (1 + (1 + (1 + \texttt{Depth}(v_1)))) \\
&= (1 + (1 + (1 + (1 + \texttt{Depth}(\textbf{null}))))) \\
&= (1 + (1 + (1 + (1 + -1)))) \\
&= (1 + (1 + (1 + 0))) \\
&= (1 + (1 + 1)) \\
&= (1 + 2) \\
&= 3
\end{aligned}
$$

# Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer

# Trær – høyde

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

# Trær – høyde

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

# 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)
```

# 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
```

# 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 c ∈ v.children do
6         h ← Max(h, height(c))
```

# 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 c ∈ v.children do
6         h ← Max(h, height(c))
7     return 1 + h
```

# Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på

# Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre

# 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

# 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

# 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

# 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 egner *preorder* seg bedre enn *postorder*

# 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 egner *preorder* seg bedre enn *postorder*
- For å slette et tre kan egner *postorder* seg bedre enn *preorder*

# Trær – preorder og postorder

**Algorithm:** Preorder traversering

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på $v$, deretter barna til $v$

```
1 Procedure Preorder(v)
2     if v = null then
3         return
```

# Trær – preorder og postorder

**Algorithm:** Preorder traversering

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på $v$, deretter barna til $v$

```
1 Procedure Preorder(v)
2     if v = null then
3         return
4     Operate on v
```

# Trær – preorder og postorder

**Algorithm:** Preorder traversering

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på $v$, deretter barna til $v$

```
1 Procedure Preorder(v)
2     if v = null then
3         return
4     Operate on v
5     for c ∈ v.children do
6         Preorder(c)
```

# Trær – preorder og postorder

**Algorithm:** Preorder traversering

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på $v$, deretter barna til $v$

```
1 Procedure Preorder(v)
2     if v = null then
3         return
4     Operate on v
5     for c ∈ v.children do
6         Preorder(c)
```

**Algorithm:** Postorder traversering

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på barna til $v$, deretter $v$

```
1 Procedure Postorder(v)
2     if v = null then
3         return
```

# Trær – preorder og postorder

**Algorithm:** Preorder traversering

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på $v$, deretter barna til $v$

```
1 Procedure Preorder(v)
2     if v = null then
3         return
4     Operate on v
5     for c ∈ v.children do
6         Preorder(c)
```

**Algorithm:** Postorder traversering

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på barna til $v$, deretter $v$

```
1 Procedure Postorder(v)
2     if v = null then
3         return
4     for c ∈ v.children do
5         Postorder(c)
```

# Trær – preorder og postorder

**Algorithm: Preorder traversering**

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på $v$, deretter barna til $v$

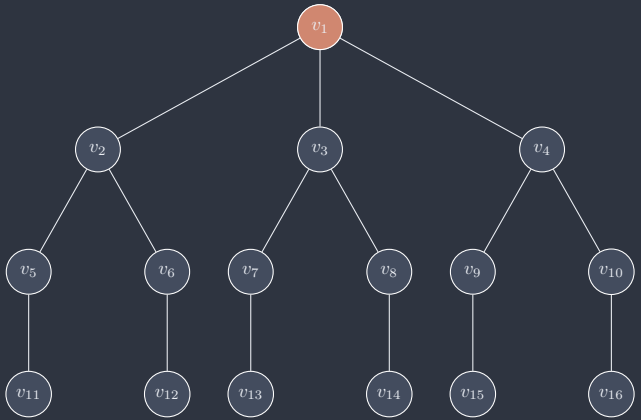
```
1 Procedure Preorder(v)
2     if v = null then
3         return
4     Operate on v
5     for c ∈ v.children do
6         Preorder(c)
```

**Algorithm: Postorder traversering**

**Input:** En node $v$ (som ikke er `null`)
**Output:** Utfør en operasjon på barna til $v$, deretter $v$

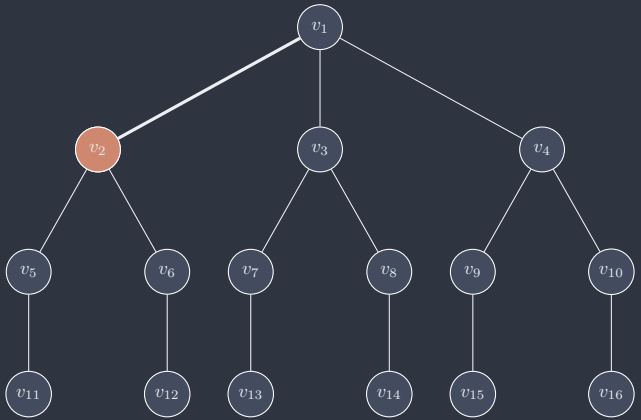
```
1 Procedure Postorder(v)
2     if v = null then
3         return
4     for c ∈ v.children do
5         Postorder(c)
6     Operate on v
```

# Trær – preorder (eksempel)



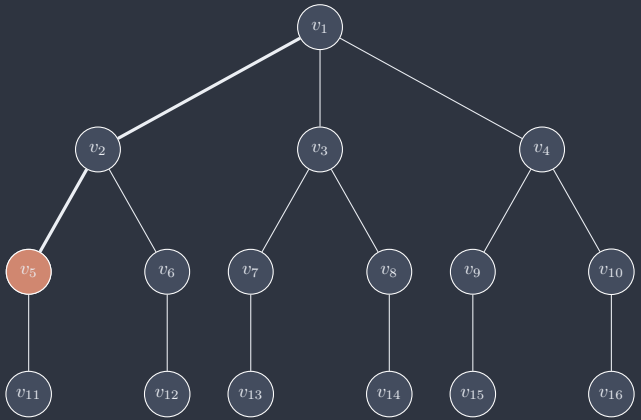

$v_1,$

# Trær – preorder (eksempel)



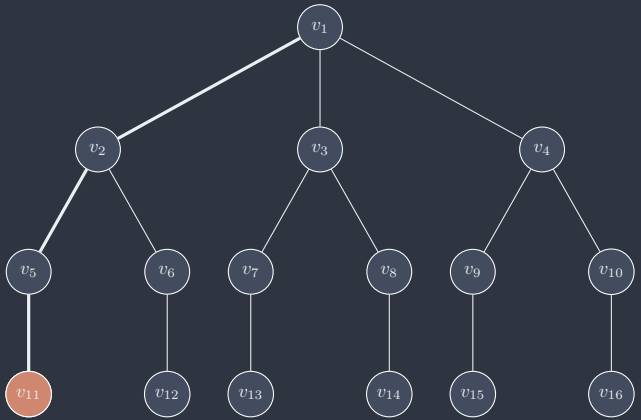

$v_1, v_2,$

# Trær – preorder (eksempel)



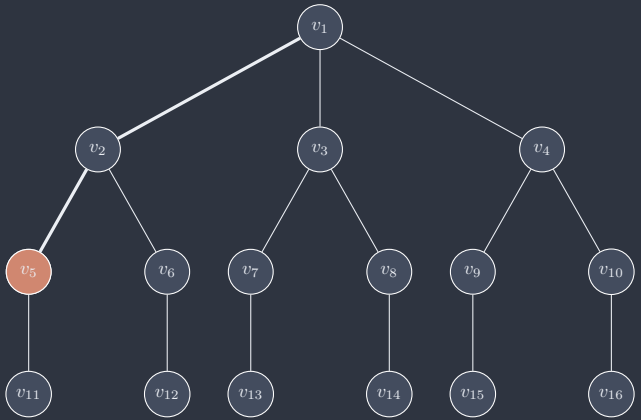

$v_1, v_2, v_5,$

# Trær – preorder (eksempel)



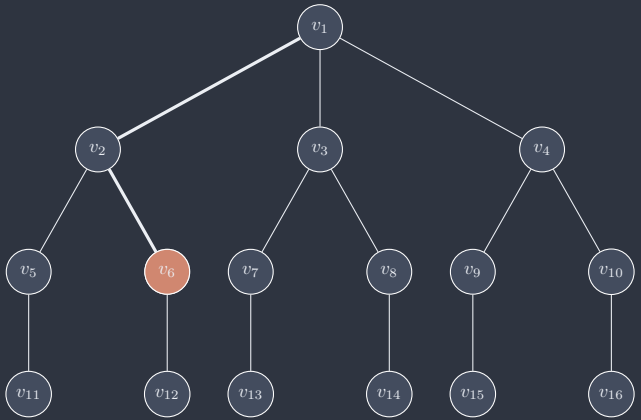

$v_1, v_2, v_5, v_{11},$

# Trær – preorder (eksempel)



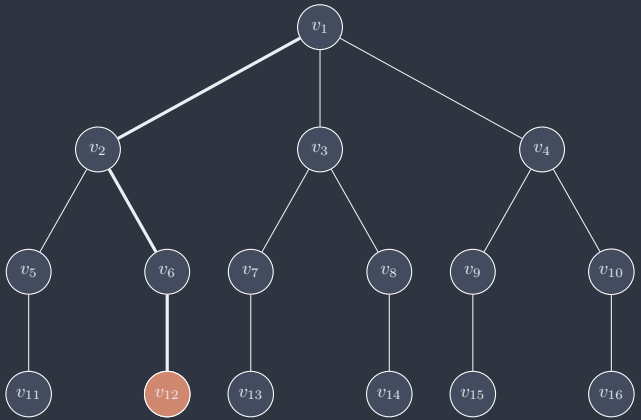

$v_1, v_2, v_5, v_{11},$

# Trær – preorder (eksempel)



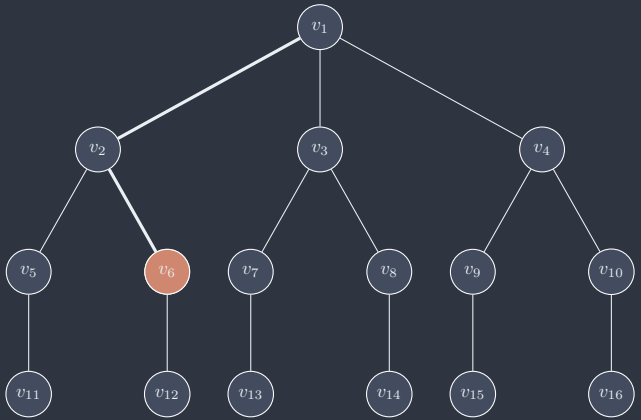

$v_1, v_2, v_5, v_{11},$
$v_6,$

# Trær – preorder (eksempel)



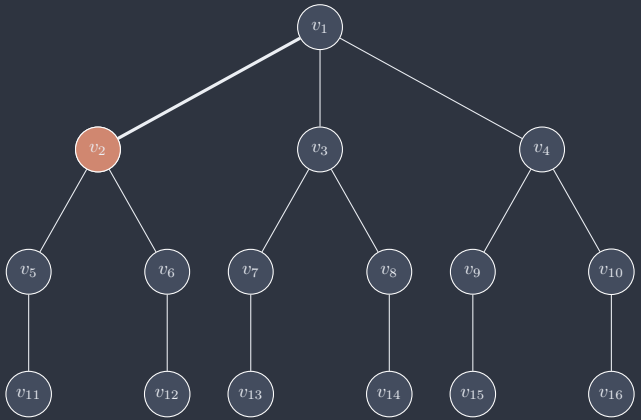

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,

# Trær – preorder (eksempel)



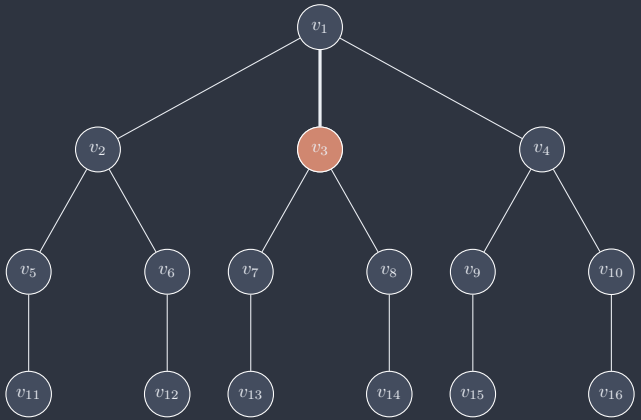

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$

# Trær – preorder (eksempel)



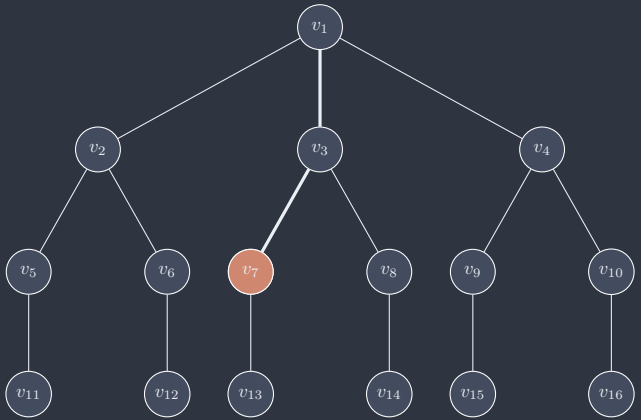

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$

# Trær – preorder (eksempel)



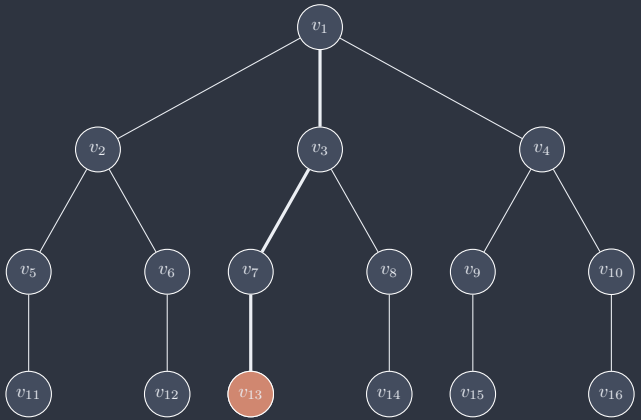

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$,

# Trær – preorder (eksempel)



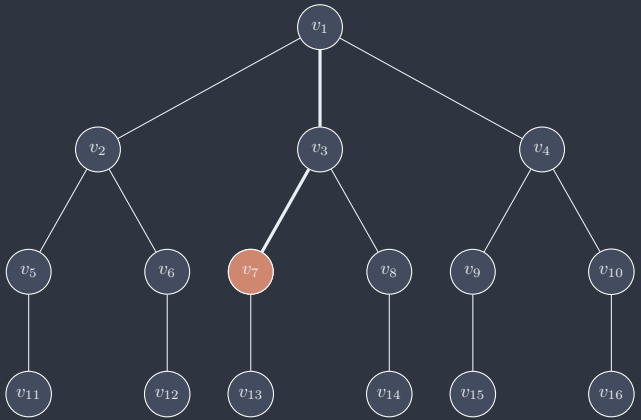

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$,

# Trær – preorder (eksempel)



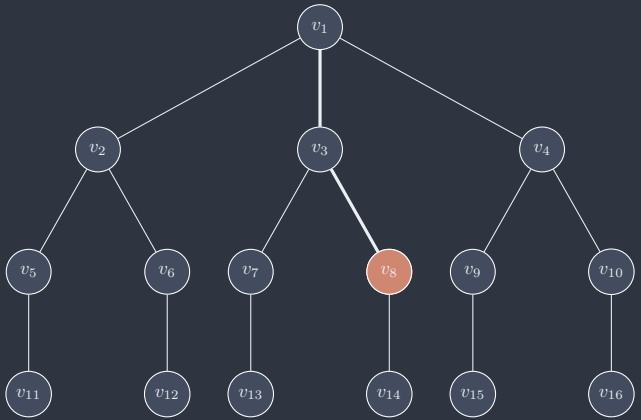

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,

# Trær – preorder (eksempel)



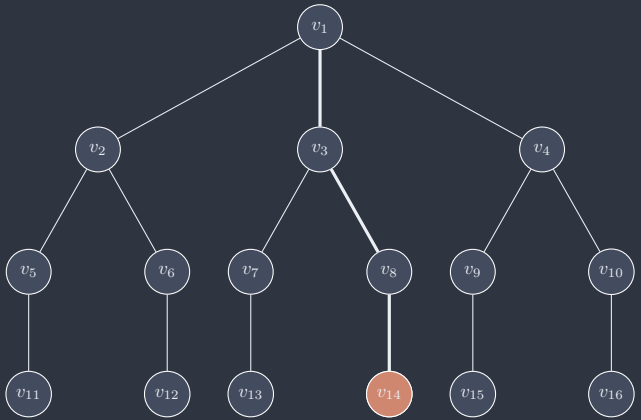

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$

# Trær – preorder (eksempel)



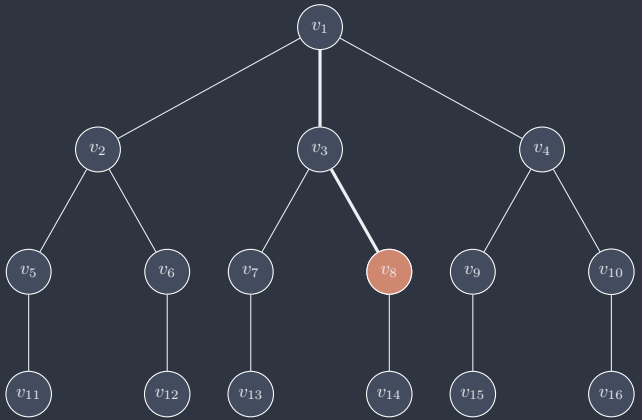

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8,$

# Trær – preorder (eksempel)



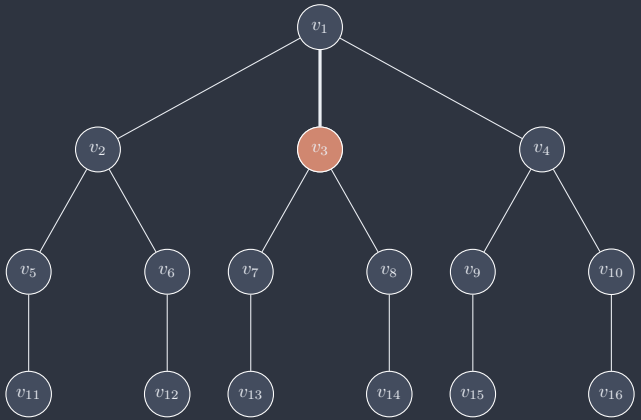

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,
$v_8$, $v_{14}$,

# Trær – preorder (eksempel)



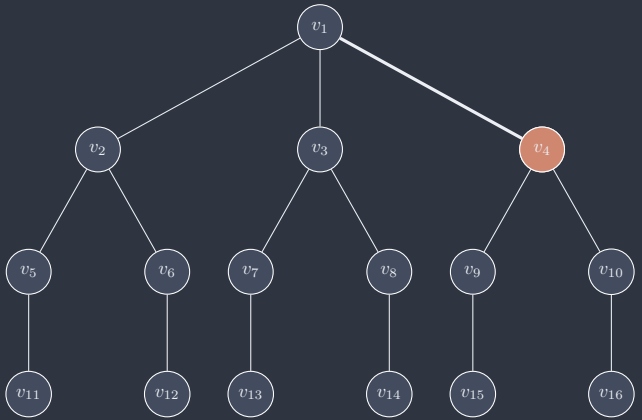

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8, v_{14},$

# Trær – preorder (eksempel)



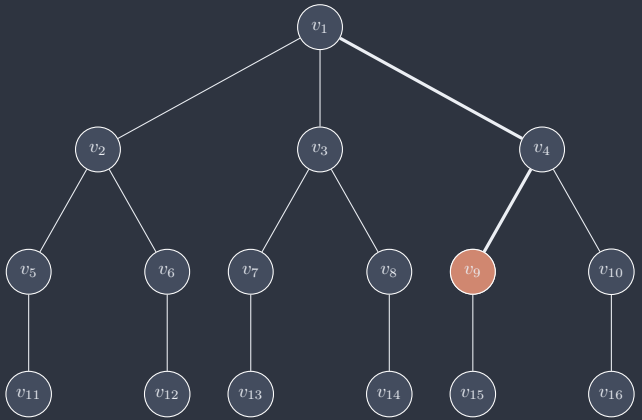

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8, v_{14},$

# Trær – preorder (eksempel)



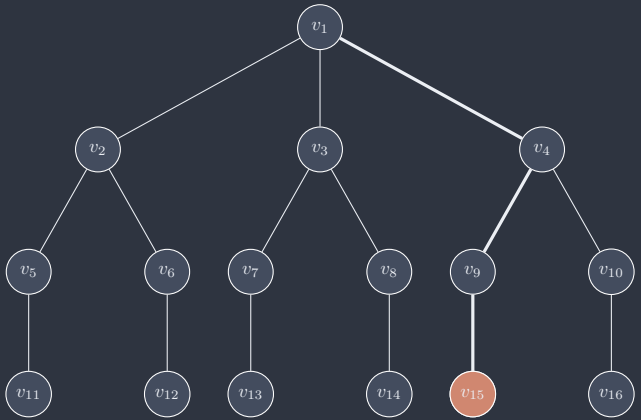

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,
$v_8$, $v_{14}$,
$v_4$,

# Trær – preorder (eksempel)



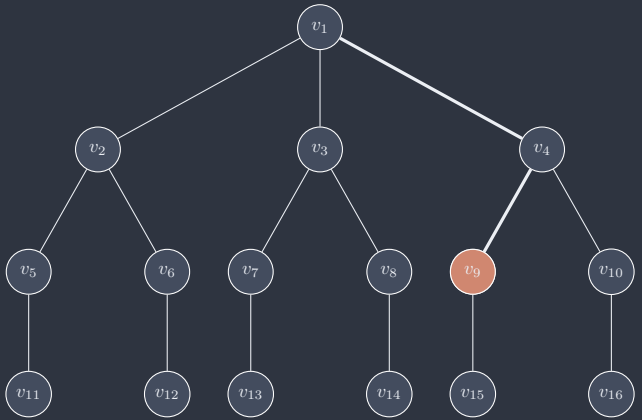

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,
$v_8$, $v_{14}$,
$v_4$, $v_9$,

# Trær – preorder (eksempel)



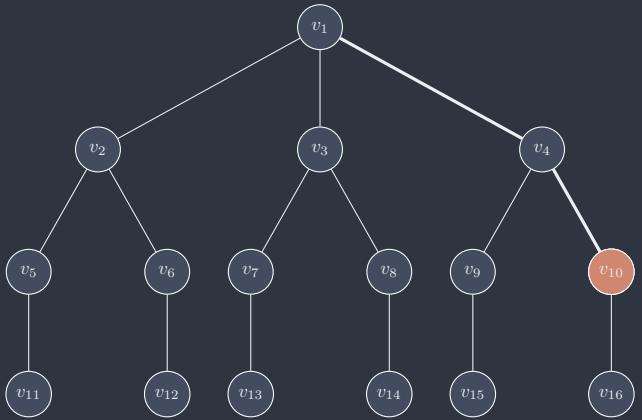

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8, v_{14},$
$v_4, v_9, v_{15},$

# Trær – preorder (eksempel)



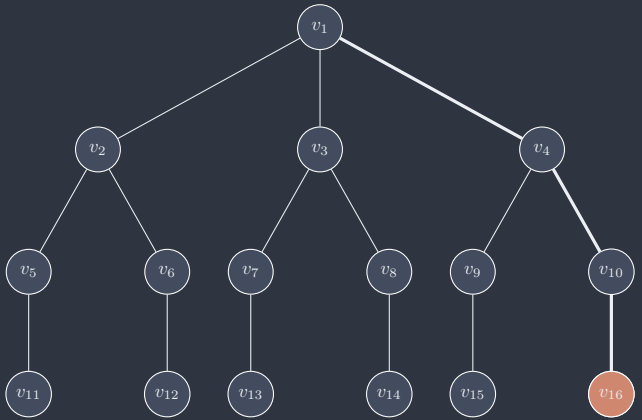

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8, v_{14},$
$v_4, v_9, v_{15},$

# Trær – preorder (eksempel)



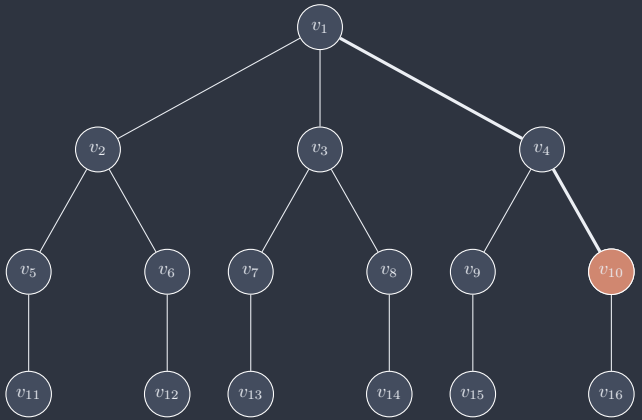

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8, v_{14},$
$v_4, v_9, v_{15},$
$v_{10},$

# Trær – preorder (eksempel)



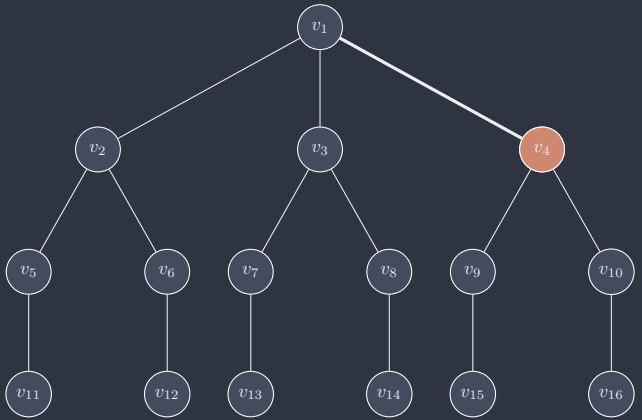

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,
$v_8$, $v_{14}$,
$v_4$, $v_9$, $v_{15}$,
$v_{10}$, $v_{16}$

# Trær – preorder (eksempel)



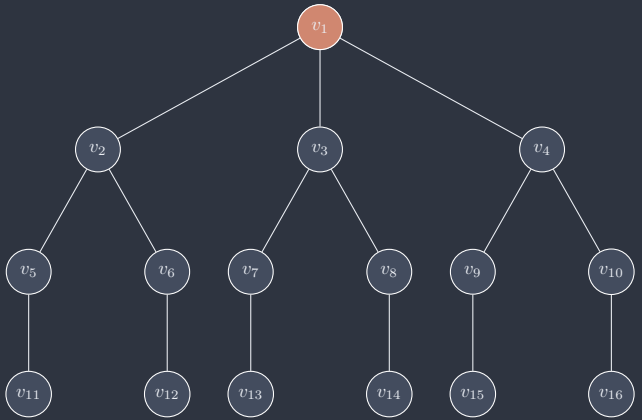

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,
$v_8$, $v_{14}$,
$v_4$, $v_9$, $v_{15}$,
$v_{10}$, $v_{16}$

# Trær – preorder (eksempel)



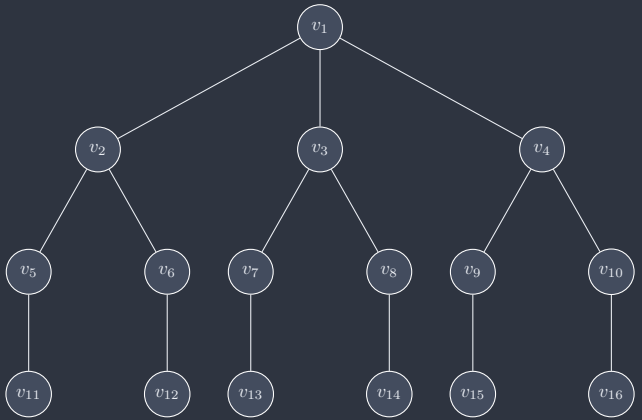

$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,
$v_8$, $v_{14}$,
$v_4$, $v_9$, $v_{15}$,
$v_{10}$, $v_{16}$

# Trær – preorder (eksempel)



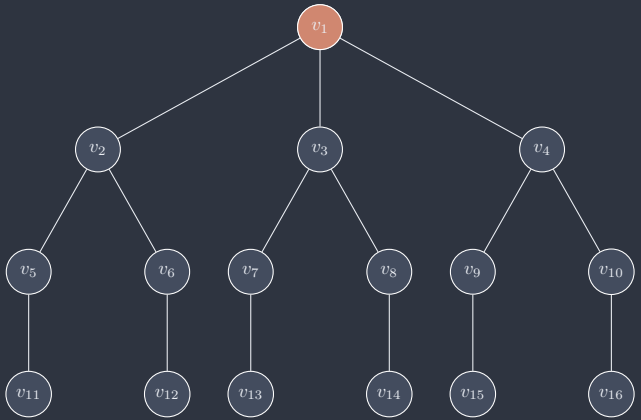

$v_1, v_2, v_5, v_{11},$
$v_6, v_{12},$
$v_3, v_7, v_{13},$
$v_8, v_{14},$
$v_4, v_9, v_{15},$
$v_{10}, v_{16}$

# Trær – preorder (eksempel)

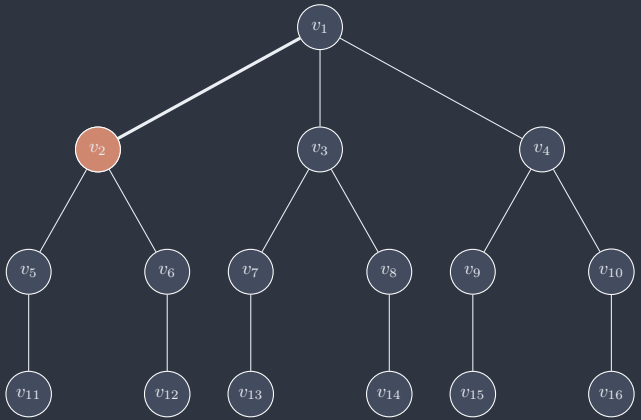



$v_1$, $v_2$, $v_5$, $v_{11}$,
$v_6$, $v_{12}$,
$v_3$, $v_7$, $v_{13}$,
$v_8$, $v_{14}$,
$v_4$, $v_9$, $v_{15}$,
$v_{10}$, $v_{16}$

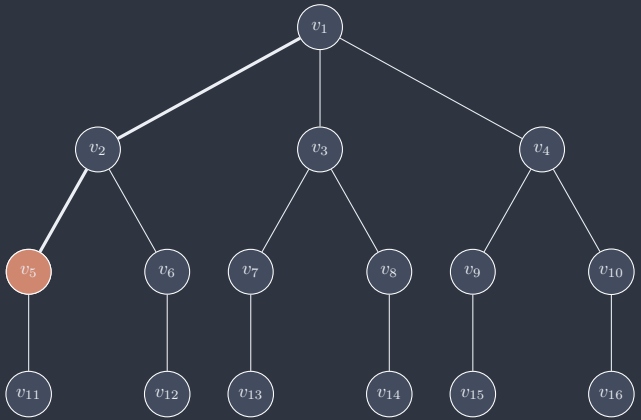
# Trær – postorder (eksempel)



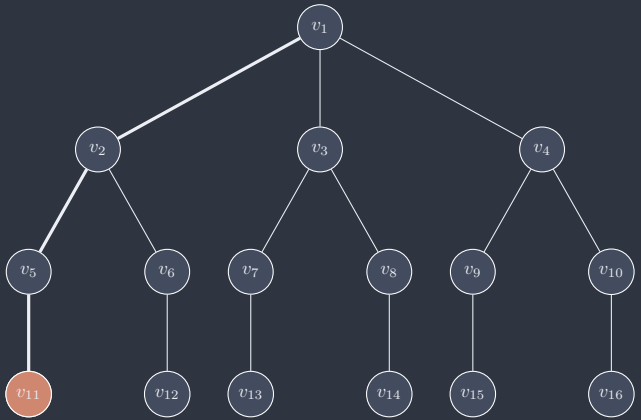
# Trær – postorder (eksempel)



# Trær – postorder (eksempel)

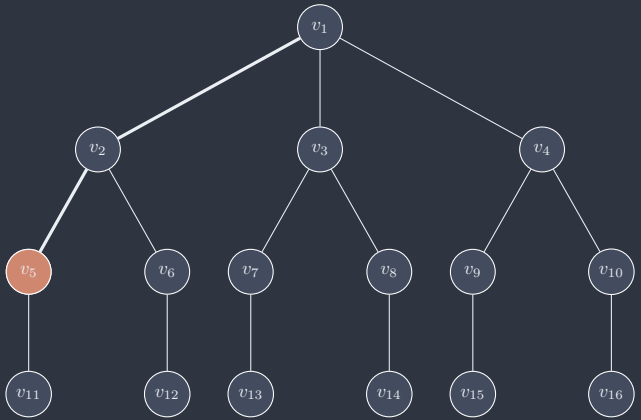


# Trær – postorder (eksempel)



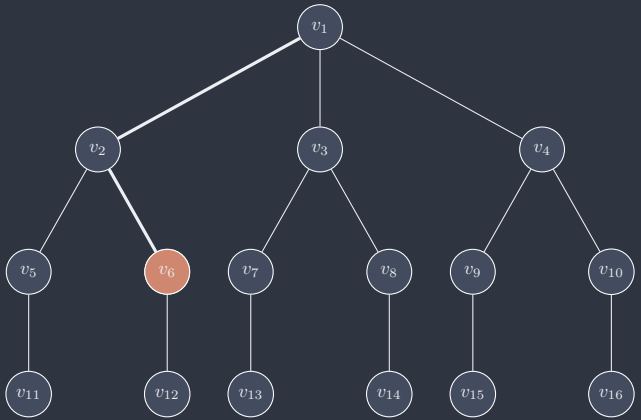

$v_{11}$,

# Trær – postorder (eksempel)



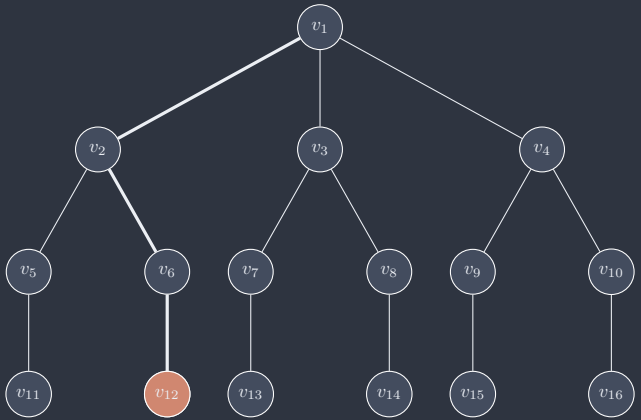

$v_{11}$, $v_5$,

# Trær – postorder (eksempel)



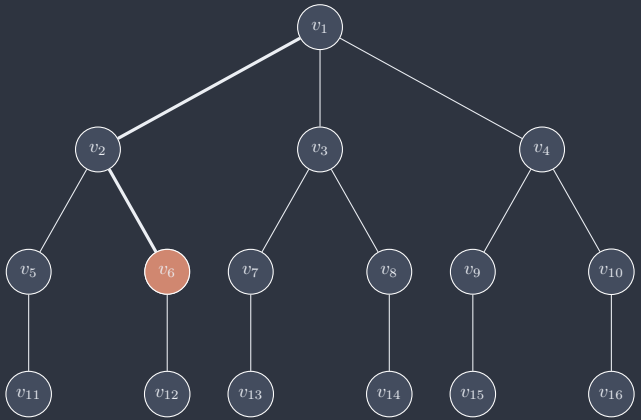

$v_{11}, v_5,$

# Trær – postorder (eksempel)



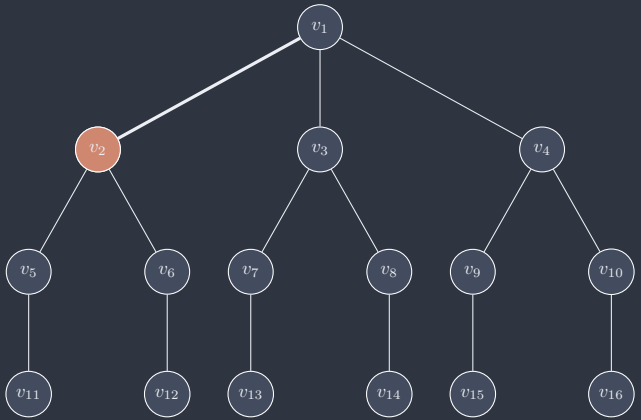

$v_{11}, v_5, v_{12},$

# Trær – postorder (eksempel)



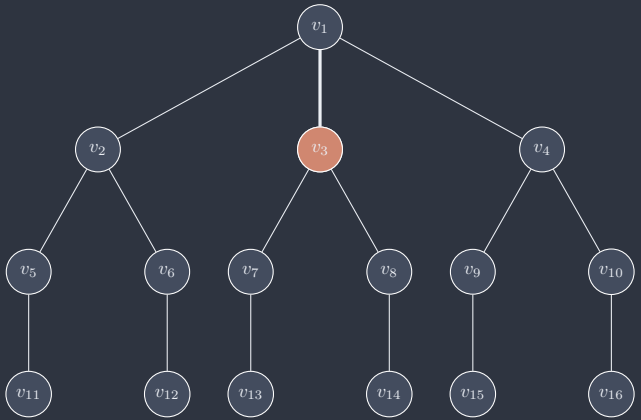

$v_{11}, v_5, v_{12}, v_6,$

# Trær – postorder (eksempel)



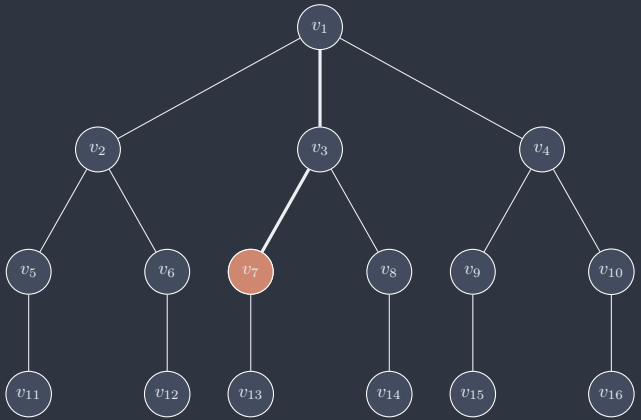

$v_{11}, v_5, v_{12}, v_6, v_2,$

# Trær – postorder (eksempel)



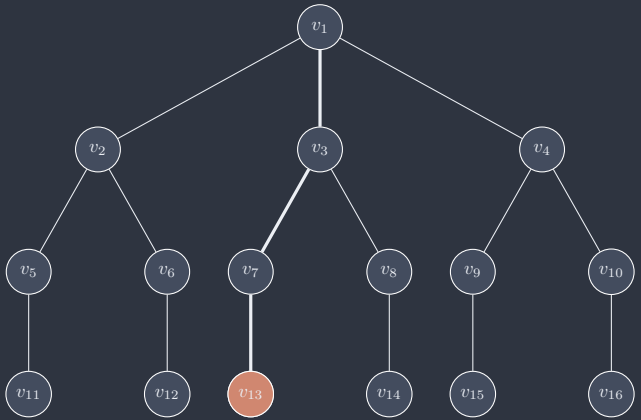

$v_{11}, v_5, v_{12}, v_6, v_2,$

# Trær – postorder (eksempel)



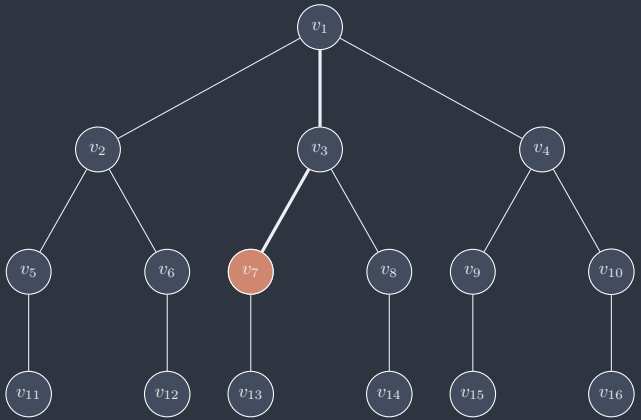

$v_{11}, v_5, v_{12}, v_6, v_2,$

# Trær – postorder (eksempel)



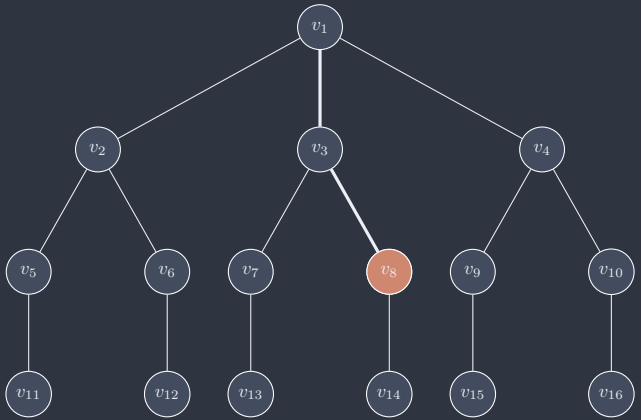

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13},$

# Trær – postorder (eksempel)



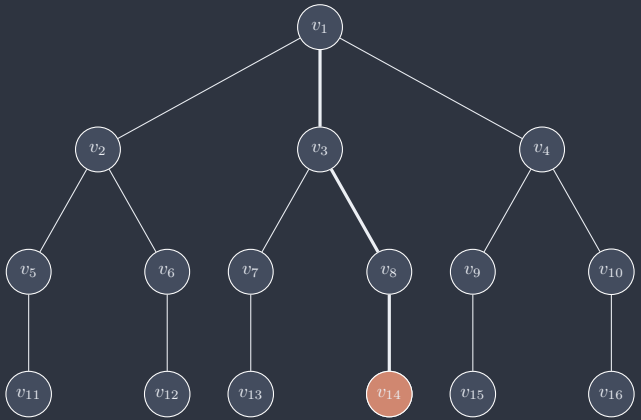

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7,$

# Trær – postorder (eksempel)



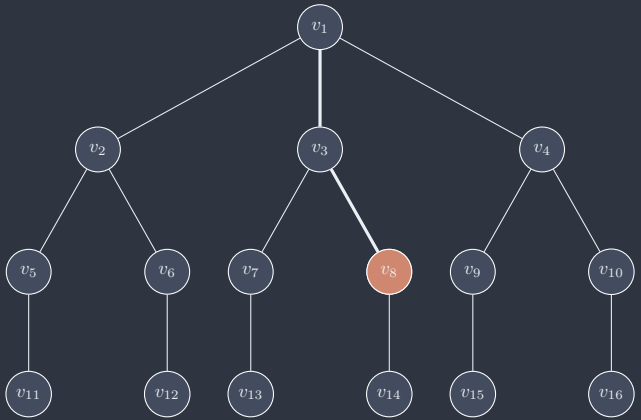

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7,$

# Trær – postorder (eksempel)



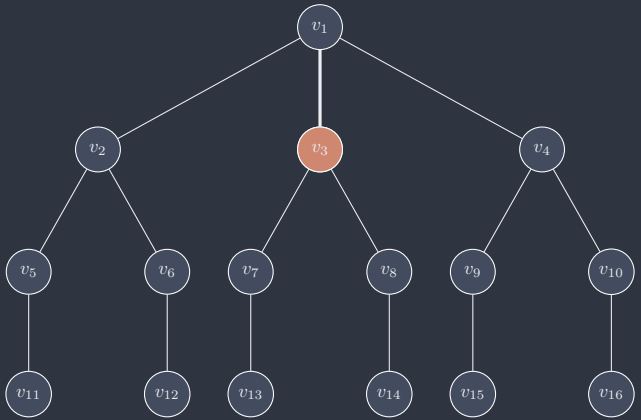

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14},$

# Trær – postorder (eksempel)



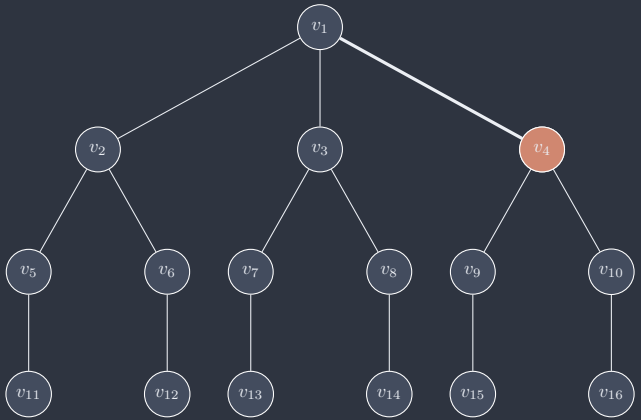

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8,$

# Trær – postorder (eksempel)



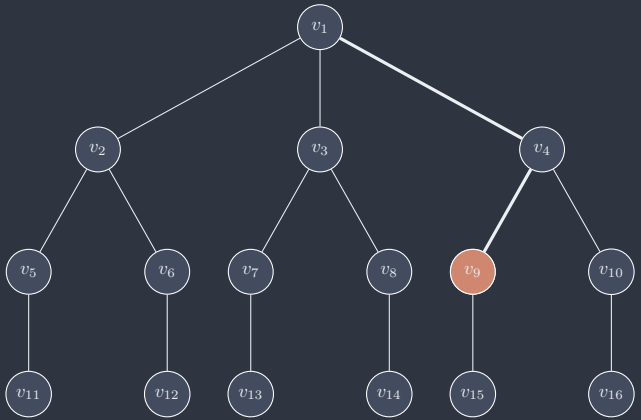

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$

# Trær – postorder (eksempel)



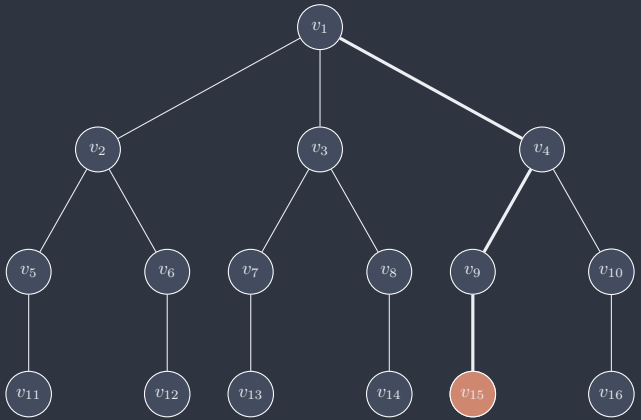

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$

# Trær – postorder (eksempel)



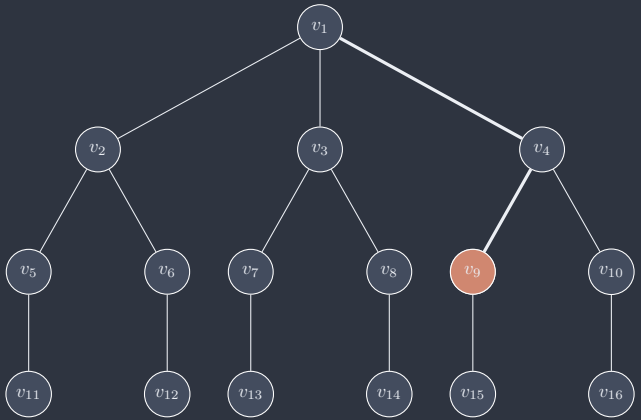

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$

# Trær – postorder (eksempel)



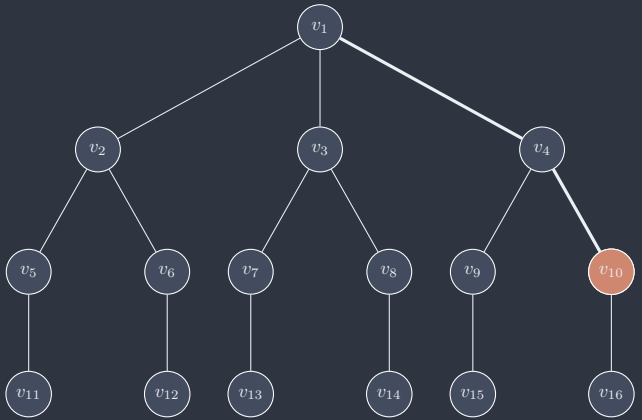

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15},$

# Trær – postorder (eksempel)



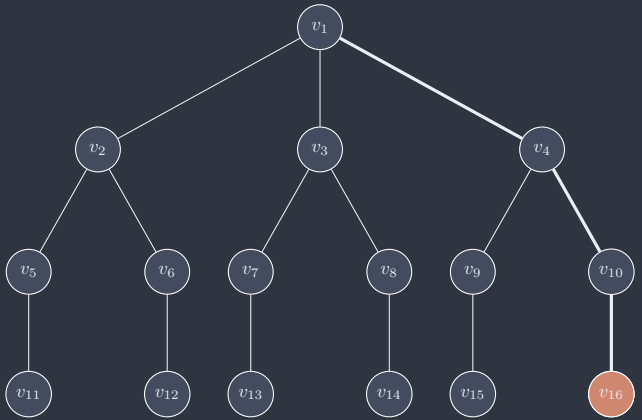

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9,$

# Trær – postorder (eksempel)



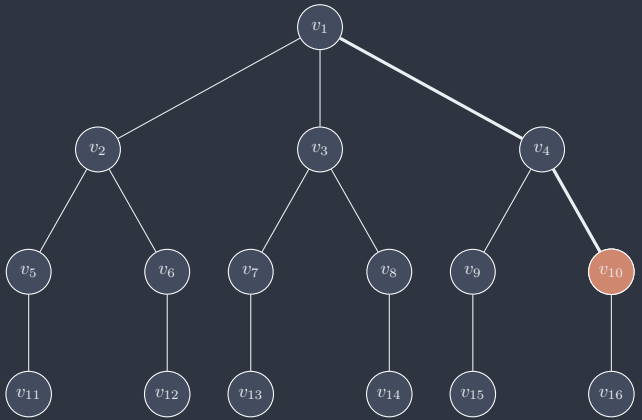

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9,$

# Trær – postorder (eksempel)



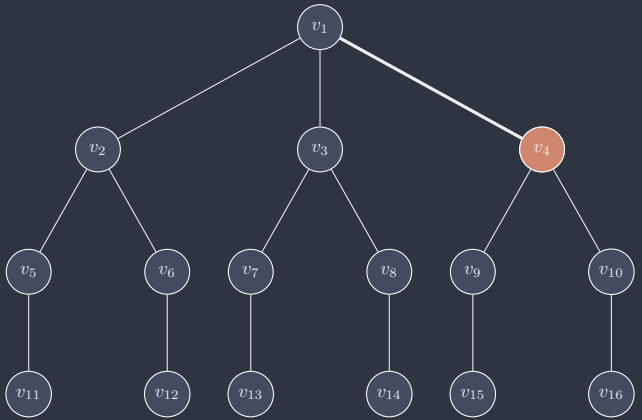

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9, v_{16},$

# Trær – postorder (eksempel)



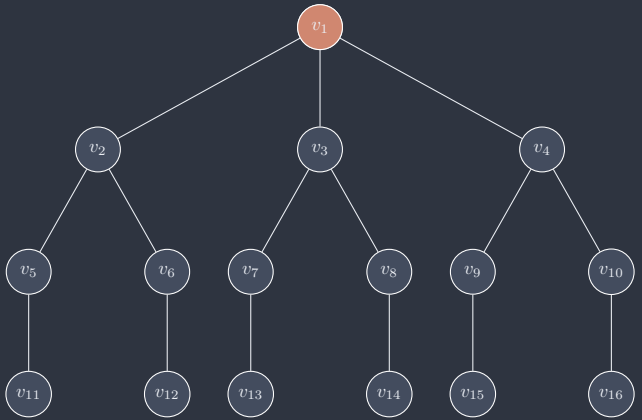

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9, v_{16}, v_{10},$

# Trær – postorder (eksempel)



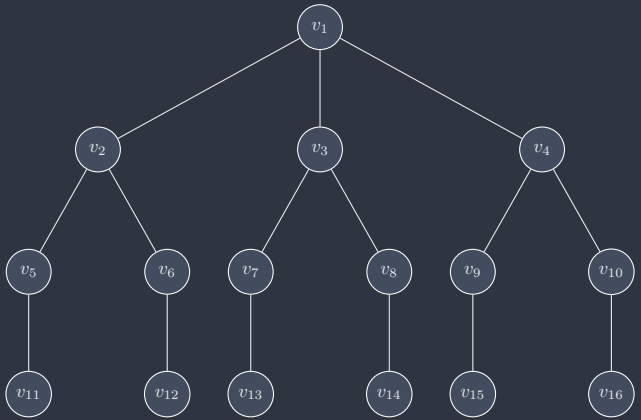

$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9, v_{16}, v_{10}, v_4,$

# Trær – postorder (eksempel)




$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9, v_{16}, v_{10}, v_4,$
$v_1$

# Trær – postorder (eksempel)




$v_{11}, v_5, v_{12}, v_6, v_2,$
$v_{13}, v_7, v_{14}, v_8, v_3,$
$v_{15}, v_9, v_{16}, v_{10}, v_4,$
$v_1$

# Binære søketrær

# Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn

# Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn

# 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

# 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

# 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 $v$

# 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

# Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node $v$ så er $v$.`element`

# 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

# 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

# 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

# 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 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*

# 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

# 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 det forutsetter at vi jobbber på et *sortert* array

# 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 det forutsetter at vi jobbber på et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*

# 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 det forutsetter at vi jobbber på et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*
  - En datastruktur hvor vi stadig legger til og fjerner elementer

# 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 det forutsetter at vi jobbber på 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

# 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 det forutsetter at vi jobbber på 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*

# 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 det forutsetter at vi jobbber på 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 $v$

# 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`$(v, x)$

# 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)
```

# 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)
```

# 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)
```

# 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
```

# 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 $\mathcal{O}(h)$

# 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

# 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

# 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`**.

# 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)
```

# 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
```

# 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
```

# 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)
```

# 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

**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

# Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!

# Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle «hull»

# 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

# 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

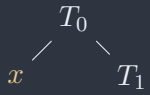
# Sletting

- 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

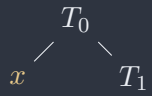
# Sletting

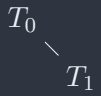
- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle «hull»
- Vi skiller mellom tre tilfeller
  - Noden vi vil slette har ingen barn
  - Noden vi vil slette har ett barn
  - Noden vi vil slette har to barn

# Sletting – ingen barn

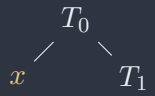


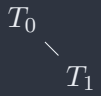
# Sletting – ingen barn



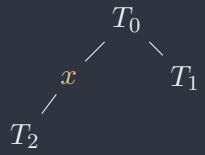
# Sletting – ingen barn



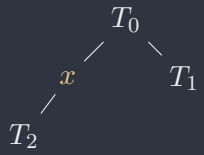



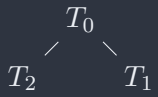

- Når det ikke er noen barn er det tilstrekkelig å fjerne pekeren til $x$

# Sletting – ett barn (venstre)



# Sletting – ett barn (venstre)



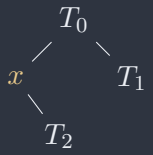



# Sletting – ett barn (venstre)

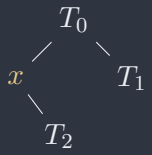


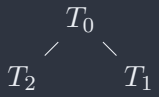

- Når noden har ett barn på venstre side, erstatt $x$ med $T_2$

# Sletting – ett barn (høyre)



# Sletting – ett barn (høyre)



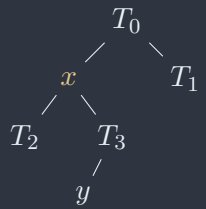



# Sletting – ett barn (høyre)

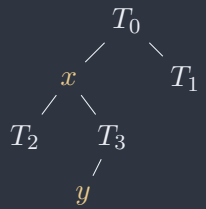



- Helt tilsvarende, når noden har ett barn på høyre side, erstatt $x$ med $T_2$

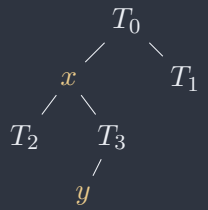
# Sletting – to barn

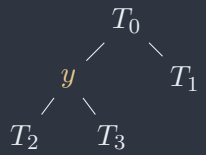


# Sletting – to barn

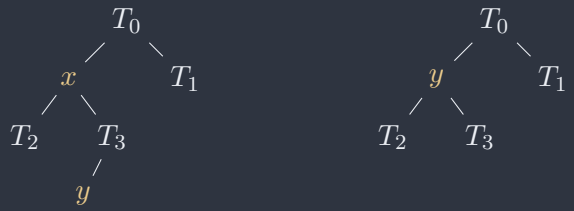


# Sletting – to barn



# Sletting – to barn




- Når noden har to barn, erstatt $x$ med det minste elementet $y$ i høyre subtre

# 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 $v$

# 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 $v$

# 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 $u$.

# 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
```

# 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
```

# 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
```

# 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
```

# 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
```

# 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
```

# 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)
```

# 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
```