Balanserte søketrær

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

AVL-træer

AVL-trær

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær

AVL-trær

- AVL-trær oppfyller de samme egenskapene som ordinære binære søketrær
- I *tillegg* må de oppfyle følgende egenskap:

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

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

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

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

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

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

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

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

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

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

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

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 for en en gitt node v:

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 for en en gitt node v:
  - returnerer -1 dersom $v =$ **`null`**

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 for en en gitt node v:
  - returnerer -1 dersom $v =$ **`null`**
  - returnerer v.`height` ellers

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 for en en gitt node v:
  - returnerer -1 dersom $v =$ **`null`**
  - returnerer v.`height` ellers
- Vi antar at vi har en prosedyre `SetHeight` som for en gitt node v:

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 for en en gitt node v:
  - returnerer -1 dersom $v =$ **`null`**
  - returnerer v.`height` ellers
- Vi antar at vi har en prosedyre `SetHeight` som for en gitt node v:
  - ikke gjør noe hvis $v =$ **`null`**

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
  - $v.\texttt{element}$ dataen som er lagret i noden
  - $v.\texttt{left}$ venstre barn av v
  - $v.\texttt{right}$ høyre barn av v
  - $v.\texttt{height}$ 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 for en en gitt node v:
  - returnerer -1 dersom $v = \textbf{null}$
  - returnerer $v.\texttt{height}$ ellers
- Vi antar at vi har en prosedyre `SetHeight` som for en gitt node v:
  - ikke gjør noe hvis $v = \textbf{null}$
  - setter $v.\texttt{height}$ til $1 + \texttt{Max}(\texttt{Height}(v.\texttt{left}), \texttt{Height}(v.\texttt{right}))$ ellers

Overordnet idé

- Vi bruker metodene for sletting og innsetting fra ordinære binære søketrær

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)

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

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*

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

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 når AVL innsetting og sletting gjøres på på AVL-trær så:

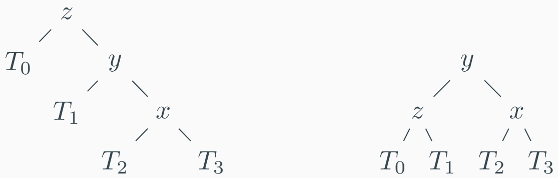
Overordnet idé

- 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 når AVL innsetting og sletting gjøres på på AVL-trær så:
    - kan vi anta at treet ikke har høydeforskjeller større enn 1

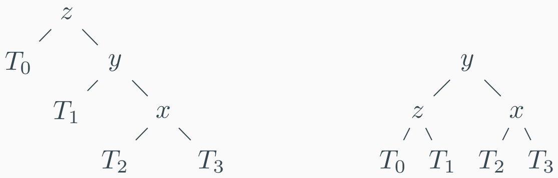
Overordnet idé

- 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 når AVL innsetting og sletting gjøres på på AVL-trær så:
    - 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



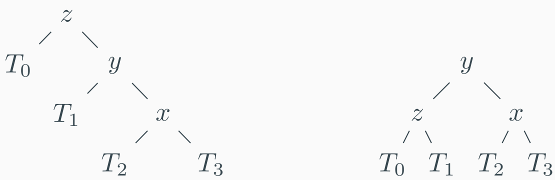
Rotasjoner – venstrerotasjon



ALGORITHM: VENSTREROTASJON AV ET BINÆRTRE

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

Rotasjoner – venstrerotasjon

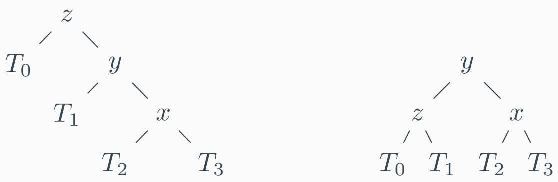


Algorithm: Venstrerotasjon av et Binærtre

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

```
Procedure LeftRotate(z)
```

Rotasjoner – venstrerotasjon



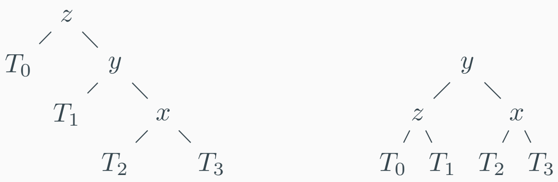
Algorithm: Venstrerotasjon av et Binærtre

Input: En node z

Output: Roter treet til venstre, slik at y blir den nye roten

```
Procedure LeftRotate(z)
    y ← z.right
    T1 ← y.left
```

Rotasjoner – venstrerotasjon



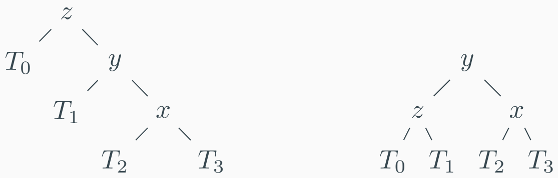
ALGORITHM: VENSTREROTASJON AV ET BINÆRTRE

Input: En node z

Output: Roter treet til venstre, slik at y blir den nye roten

```
Procedure LeftRotate(z)
    y ← z.right
    T1 ← y.left
    y.left ← z
    z.right ← T1
```

Rotasjoner – venstrerotasjon

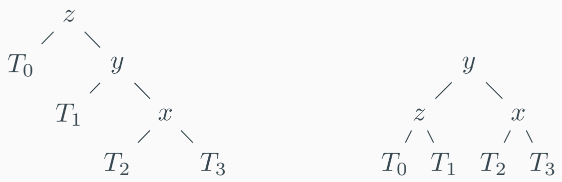


ALGORITHM: VENSTREROTASJON AV ET BINÆRTRE

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

```
Procedure LeftRotate(z)
    y ← z.right
    T1 ← y.left

    y.left ← z
    z.right ← T1

    SetHeight(z)
    SetHeight(y)
```

Rotasjoner – venstrerotasjon



ALGORITHM: VENSTREROTASJON AV ET BINÆRTRE

Input: En node z

Output: Roter treet til venstre, slik at y blir den nye roten

Procedure LeftRotate(z)

  $y \leftarrow z$.right

  $T_1 \leftarrow y$.left

  y.left $\leftarrow z$

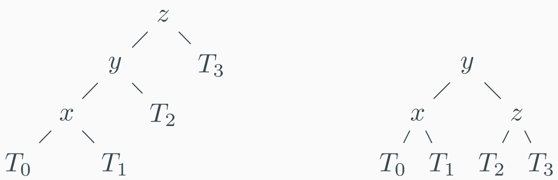
  z.right $\leftarrow T_1$

  SetHeight(z)

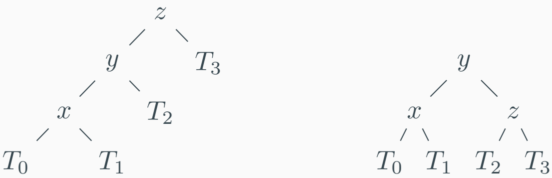
  SetHeight(y)

  return y

Rotasjoner – høyrerotasjon



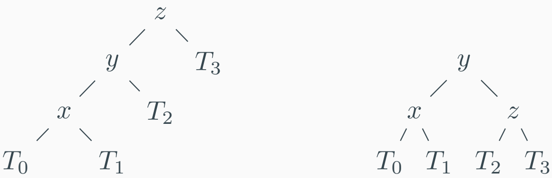
Rotasjoner – høyrerotasjon



ALGORITHM: HØYREROTASJON AV ET BINÆRTRE

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

Rotasjoner – høyrerotasjon



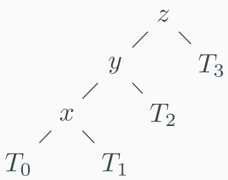
Algorithm: Høyrerotasjon av et Binærtre

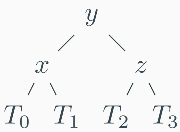
Input: En node z

Output: Roter treet til høyre, slik at y blir den nye roten

Procedure RightRotate(z)

Rotasjoner – høyrerotasjon





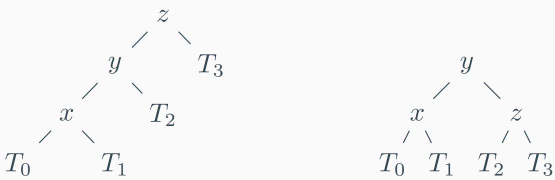
ALGORITHM: HØYREROTASJON AV ET BINÆRTRE

Input: En node z

Output: Roter treet til høyre, slik at y blir den nye roten

```
Procedure RightRotate(z)
    y ← z.left
    T2 ← y.right
```

Rotasjoner – høyrerotasjon

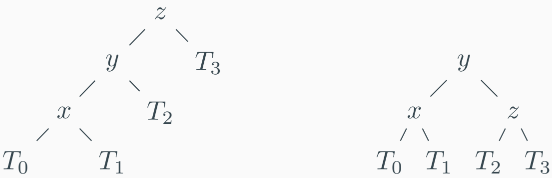


Algorithm: Høyrerotasjon av et Binærtre

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

```
Procedure RightRotate(z)
    y ← z.left
    T2 ← y.right
    y.right ← z
    z.left ← T2
```

Rotasjoner – høyrerotasjon



ALGORITHM: HØYREROTASJON AV ET BINÆRTRE

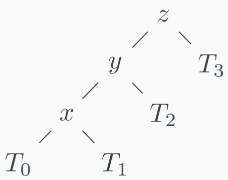
Input: En node z
Output: Roter treet til høyre, slik at y blir den nye roten

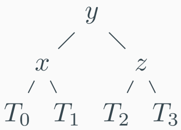
Procedure RightRotate(z)
  $y \leftarrow z$.left
  $T_2 \leftarrow y$.right

  y.right $\leftarrow z$
  z.left $\leftarrow T_2$

  SetHeight(z)
  SetHeight(y)

Rotasjoner – høyrerotasjon



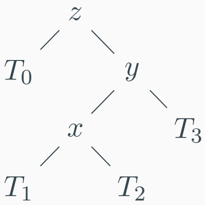


ALGORITHM: HØYREROTASJON AV ET BINÆRTRE

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

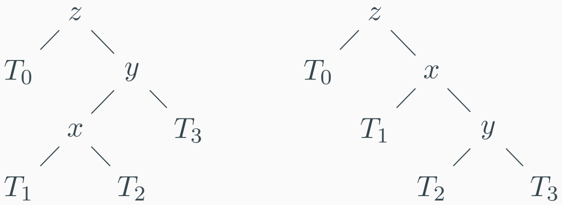
```
Procedure RightRotate(z)
    y ← z.left
    T2 ← y.right

    y.right ← z
    z.left ← T2

    SetHeight(z)
    SetHeight(y)

    return y
```

Rotasjoner – doble rotasjoner



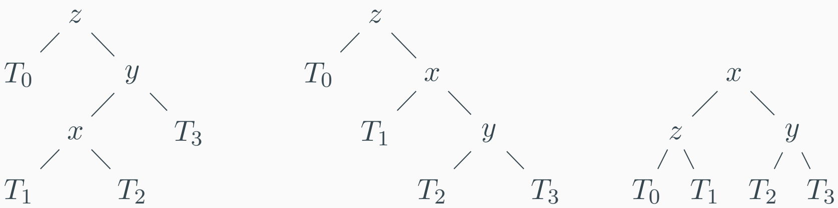
Rotasjoner – doble rotasjoner

$\texttt{RightRotate}(y)$

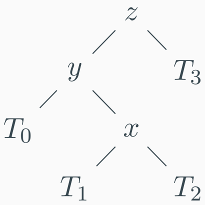


Rotasjoner – doble rotasjoner

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

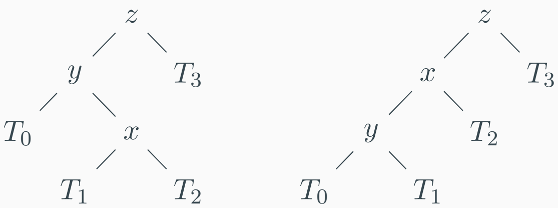


Rotasjoner – doble rotasjoner



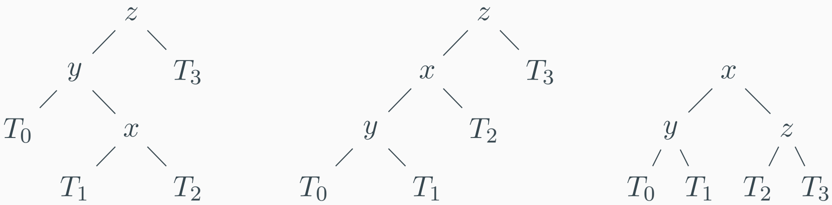
Rotasjoner – doble rotasjoner

$\texttt{LeftRotate}(y)$



Rotasjoner – doble rotasjoner

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



Balansefaktor

ALGORITHM: BALANSEFAKTOREN AV EN NODE

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

Balansefaktor

ALGORITHM: BALANSEFAKTOREN AV EN NODE

```
Input: En node v
Output: Returner høydeforskjellen på v sitt venstre- og høyrebarn
Procedure BalanceFactor(v)
    if v = null then
        return 0
    return Height(v.left) − Height(v.right)
```

- En hjelpeprosedyre som sier hvor venstre- eller høyretungt v er

Balansefaktor

Algorithm: Balansefaktoren av en node

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

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

- En hjelpeprosedyre som sier hvor venstre- eller høyretungt v er
- 0 betyr at v er balansert

Balansefaktor

Algorithm: Balansefaktoren av en node

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

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

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

Balansefaktor

Algorithm: Balansefaktoren av en node

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

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

- En hjelpeprosedyre som sier hvor venstre- 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

Balansering

Algorithm: Balansering av et AVL-tre

Input: En node v
Output: En balansert node

```
Procedure Balance(v)
```

Balansering

ALGORITHM: BALANSERING AV ET AVL-TRE

Input: En node v
Output: En balansert node

```
Procedure Balance(v)
    if BalanceFactor(v) < −1 then
```

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 $\leftarrow$ RightRotate(v.right)
        return LeftRotate(v)

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
```

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)
```

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)
```

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

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

```
Procedure Insert(v, x)
```

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

```
Procedure Insert(v, x)
    if v = null then
        v ← new Node(x)
```

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

```
Procedure Insert(v, x)
    if v = null then
        v ← new Node(x)
    else if x < v.element then
        v.left ← Insert(v.left, x)
```

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

```
Procedure Insert(v, x)
    if v = null then
        v ← new Node(x)
    else if x < v.element then
        v.left ← Insert(v.left, x)
    else if x > v.element then
        v.right ← Insert(v.right, x)
```

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

```
Procedure Insert(v, x)
    if v = null then
        v ← new Node(x)
    else if x < v.element then
        v.left ← Insert(v.left, x)
    else if x > v.element then
        v.right ← Insert(v.right, x)
    SetHeight(v)
    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.

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.

```
Procedure Remove(v, x)
    if v = null then
        return null
```

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.

```
Procedure Remove(v, x)
    if v = null then
        return null
    if x < v.element then
        v.left ← Remove(v.left, x)
```

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.

```
Procedure Remove(v, x)
    if v = null then
        return null
    if x < v.element then
        v.left ← Remove(v.left, x)
    else if x > v.element then
        v.right ← Remove(v.right, x)
```

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.

```
Procedure Remove(v, x)
    if v = null then
        return null
    if x < v.element then
        v.left ← Remove(v.left, x)
    else if x > v.element then
        v.right ← Remove(v.right, x)
    else if v.left = null then
        v ← v.right
```

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.

```
Procedure Remove(v, x)
    if v = null then
        return null
    if x < v.element then
        v.left ← Remove(v.left, x)
    else if x > v.element then
        v.right ← Remove(v.right, x)
    else if v.left = null then
        v ← v.right
    else if v.right = null then
        v ← v.left
```

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.

```
Procedure Remove(v, x)
    if v = null then
        return null
    if x < v.element then
        v.left ← Remove(v.left, x)
    else if x > v.element then
        v.right ← Remove(v.right, x)
    else if v.left = null then
        v ← v.right
    else if v.right = null then
        v ← v.left
    else
        u ← FindMin(v.right)
        v.element ← u.element
        v.right ← Remove(v.right, u.element)
```

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.

```
Procedure Remove(v, x)
    if v = null then
        return null
    if x < v.element then
        v.left ← Remove(v.left, x)
    else if x > v.element then
        v.right ← Remove(v.right, x)
    else if v.left = null then
        v ← v.right
    else if v.right = null then
        v ← v.left
    else
        u ← FindMin(v.right)
        v.element ← u.element
        v.right ← Remove(v.right, u.element)
    SetHeight(v)
    return Balance(v)
```

Rød-svarte trær

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:

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

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

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

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
- Forskjellene mellom rød-svarte- og AVL-trær er:

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
- Forskjellene mellom rød-svarte- og AVL-trær er:
  - Rød-svarte trær har *svakere* krav om balanse enn AVL-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
- Forskjellene mellom rød-svarte- og AVL-trær er:
    - Rød-svarte trær har *svakere* krav om balanse enn AVL-trær
    - Rød-svarte trær bruker *litt* mindre minne, siden vi ikke trenger å lagre høydene

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
- Forskjellene mellom rød-svarte- og AVL-trær er:
  - Rød-svarte trær har *svakere* krav om balanse enn AVL-trær
  - Rød-svarte trær bruker *litt* mindre minne, siden vi ikke trenger å lagre høydene
  - Rød-svarte trær gjør færre rotasjoner

Rød-svarte trær vs. AVL-trær

- Rød-svarte trær er raskere enn AVL-trær når innsetting og sletting forekommer ofte, til sammenligning med oppslag

Rød-svarte trær vs. AVL-trær

- 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

Rød-svarte trær vs. AVL-trær

- 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

Rød-svarte trær vs. AVL-trær

- 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)

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart
3. Alle tomme trær er svarte

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart
3. Alle tomme trær er svarte
4. En rød node kan ikke ha et rødt barn

Invarianter for rød-svarte trær

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

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare 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

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!

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 *bare* en konstantfaktor lenger

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 *bare* en konstantfaktor lenger
  - Husk at $\mathcal{O}(2 \cdot \log(n)) = \mathcal{O}(\log(n))$

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 *bare* en konstantfaktor lenger
  - Husk at $\mathcal{O}(2 \cdot \log(n)) = \mathcal{O}(\log(n))$
  - Så vi bevarer $\mathcal{O}(\log(n))$ på innsetting, sletting og oppslag