# Sökträd: AVL-träd, (2,3)-träd

TDDE22, 725G97: DALG

Magnus Nielsen

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

# Sökträd är coolt!

Sökträd får till och med vara med på TV!

- CSI
- Criminal Minds
- Missing (CAN)

# Riktigt coolt!

**ACT FOUR**

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

# Balansera!

find,insert och remove i ett binärt sökträd tar $O(h)$ tid, där $h$ är höjden av trädet.

Håll sökträdet balanserat!!

# Logaritmisk höjd

Vi förutsätter ett perfekt binärt träd (ett fullt binärt träd där alla lövnoder har samma djup).

| Höjd | Noder | Logaritmberäkning |
|---|---|---|
| 0 | 1 | $\log_2(2) = 1$ |
| 1 | 3 | $\log_2(4) = 2$ |
| 2 | 7 | $\log_2(8) = 3$ |
| 3 | 15 | $\log_2(16) = 4$ |
| 4 | 31 | $\log_2(32) = 5$ |
| ... | ... | ... |
| 19 | 1 048 575 | $\log_2(1048576) = 20$ |
| ... | ... | ... |

Ett perfekt träd har höjden $h = \log_2(n+1) - 1$.
Ett balanserat träd har alltså höjden $h = O(\log_2(n))$.


OMG

LINKÖPING UNIVERSITY

# AVL-träd

- Självbalanserande BST/höjdbalanserat BST
- AVL $=$ Adelson-Velskii och Landis, 1962
- Idén: Håll reda på balansinformation i varje nod
- AVL-egenskapen
  För varje intern nod $v$ i $T$ skiljer sig höjden av barnen till $v$ med högst 1

# AVL-träd

- Självbalanserande BST/höjdbalanserat BST
- AVL = Adelson-Velskii och Landis, 1962
- Idén: Håll reda på balansinformation i varje nod
- AVL-egenskapen
  För varje intern nod $v$ i $T$ skiljer sig höjden av barnen till $v$ med högst 1
  ...eller alternativt...
  För varje intern nod $v$ i $T$ gäller att $balans(v) \in \{-1, 0, 1\}$, där $balans(v) = height(leftChild(v)) - height(rightChild(v))$

# Maximal höjd av AVL-träd

**Proposition 1** *Höjden av ett AVL-träd som lagrar $n$ poster är $O(\log n)$.*

Vilket får som följd att...

**Proposition 2** *Vi kan göra find, insert och remove i ett AVL-träd i tid $O(\log n)$ medan vi bevarar AVL-egenskapen.*

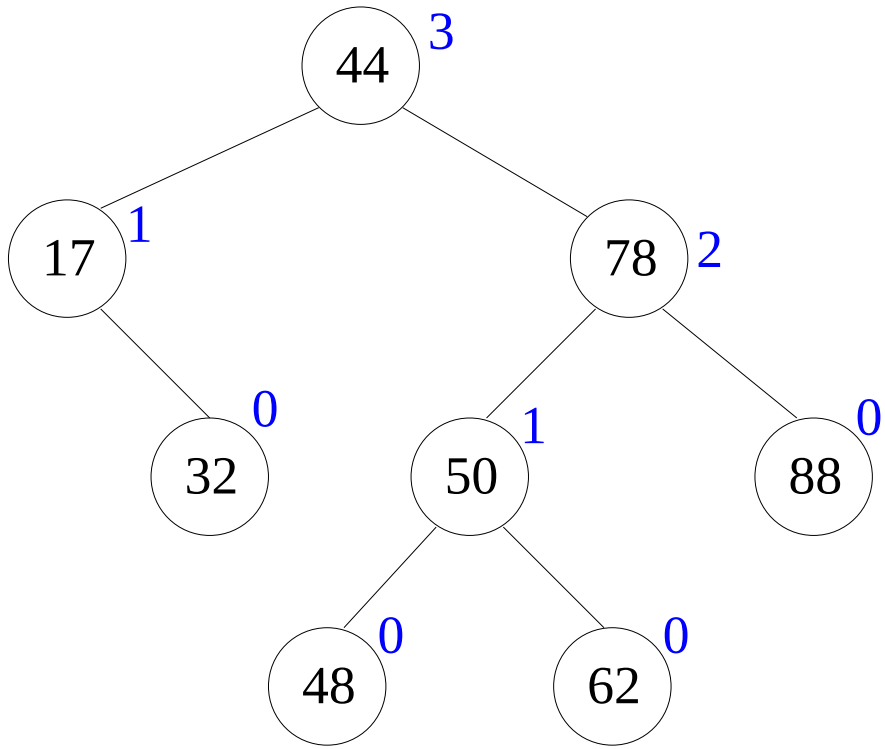
# Borttagning i ett AVL-träd

- find och remove som i ett vanligt binärt sökträd
- Uppdatera balansinformationen på väg tillbaka upp till roten
- Om för obalanserat: Strukturera om ...men...
  - När vi återställer balansen på ett ställe kan det uppstå obalans på ett annat
  - Måste upprepa balanseringen (eller kontroll av balansen) till dess vi når roten
  - Högst $O(\log n)$ ombalanseringar

# Rotationer

- Balansering måste göras både vid insert och remove
- Fyra möjliga rotationer:
  - ”Höger”eller ”vänster”
  - Enkel- eller dubbelrotation

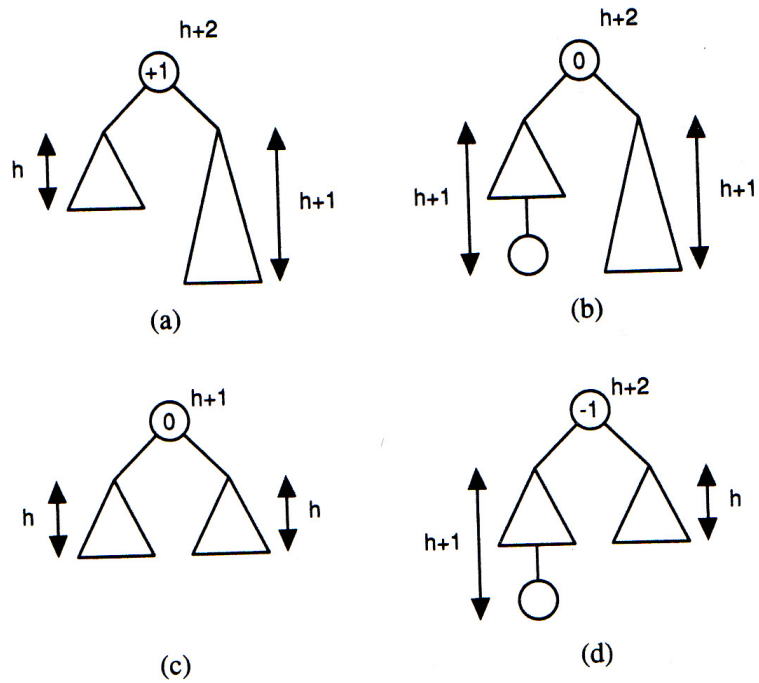
# Exempel: ett AVL-träd



# Insättning i ett AVL-träd

- Den nya noden gör att trädhöjden förändras och att trädet måste höjdbalanseras.
  - Man kan hålla reda på delträdens höjd på olika sätt:
    - Lagra höjden explicit i varje nod
    - Lagra balansfaktorn för noden
- Förändringen brukar beskrivas som en höger- eller vänsterrotation av ett delträd.
- Det räcker med en rotation för att få trädet i balans igen.

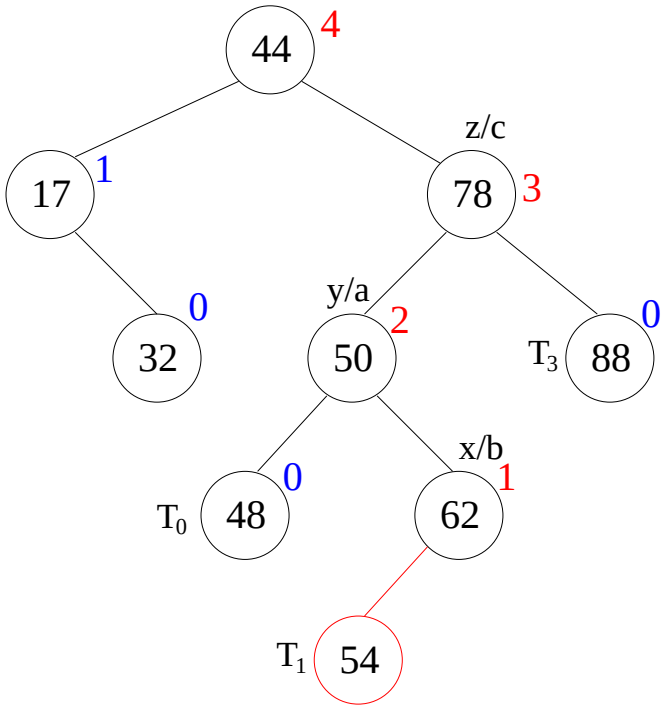
# Insättning i AVL-träd (enkla fall)



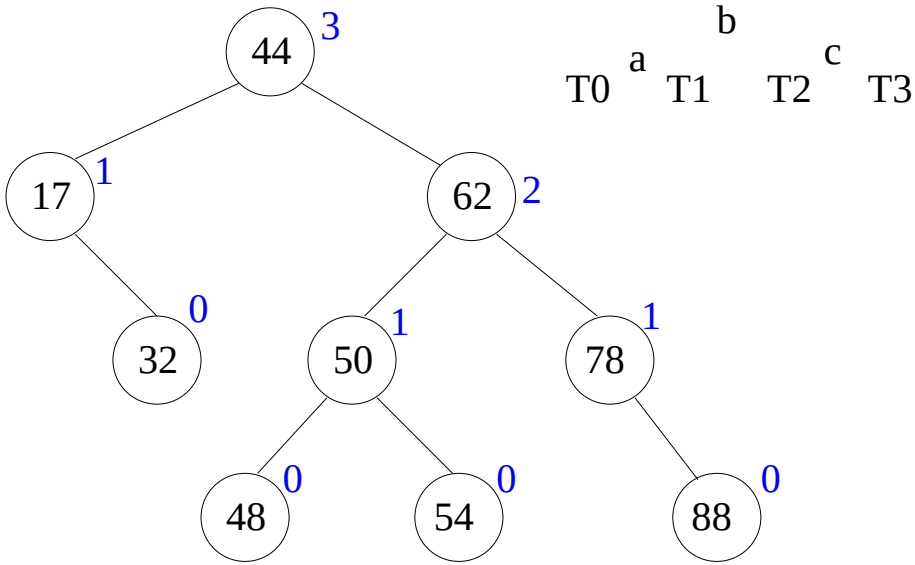
# Insättningsalgoritm

- Starta från den nya noden och leta uppåt tills man hittar en nod $x$ s.a. dess "grandparent" $z$ är obalanserad. Markera $x$:s förälder med $y$.
- Gör en rekonstruering av trädet så här:
  - Döp om $x, y, z$ till $a, b, c$ baserat på deras inorder-ordning.
  - Låt $T_0, T_1, T_2, T_3$ vara en uppräkning i inorder av delträden till $x$, $y$ och $z$. (Inget av delträden får ha $x$, $y$ eller $z$ som rot.)
  - $z$ byts mot $b$, dess barn är nu $a$ och $c$.
  - $T_0$ och $T_1$ är barn till $a$ och $T_2$ och $T_3$ är barn till $c$.

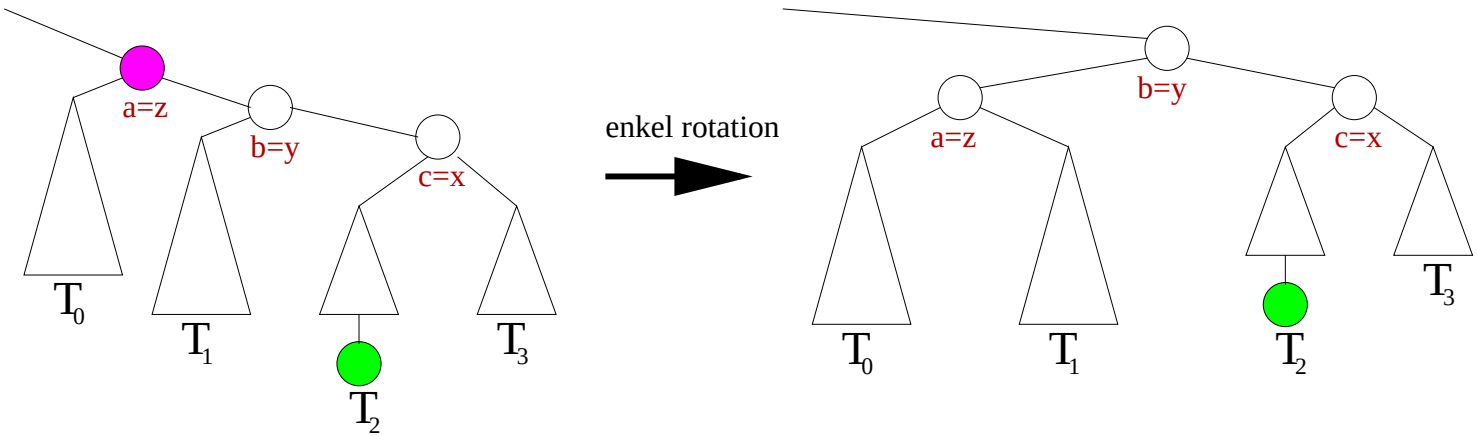
# Exempel: insättning i ett AVL-träd



# Exempel: insättning i ett AVL-träd

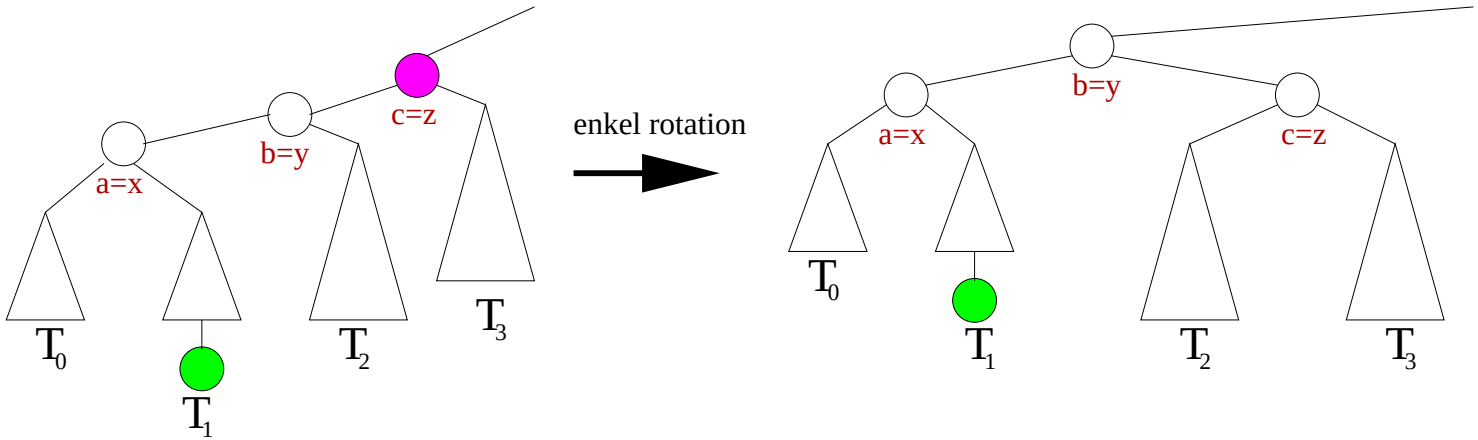


# Fyra olika rotationer



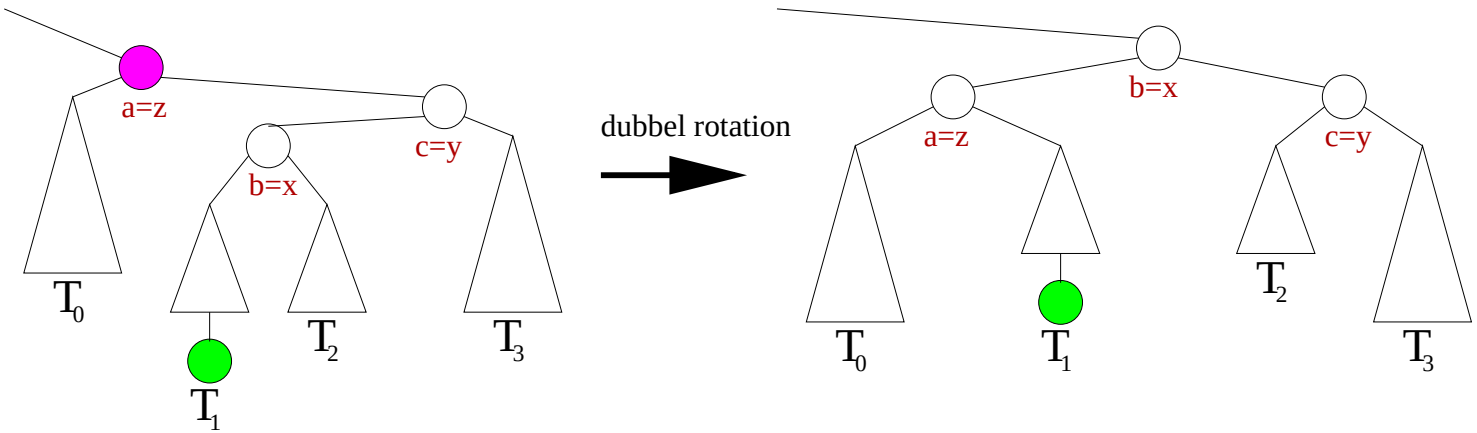

Om $b = y$ kallas det en enkel rotation.
"Rotera upp $y$ över $z$"

# Fyra olika rotationer



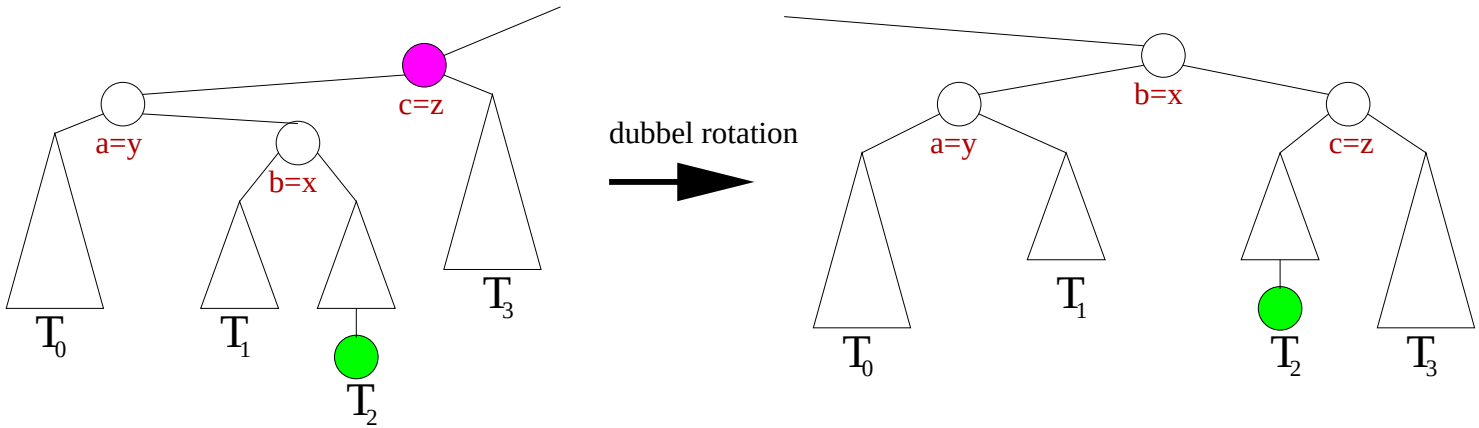

Om $b = y$ kallas det en enkel rotation.
"Rotera upp $y$ över $z$"

# Fyra olika rotationer



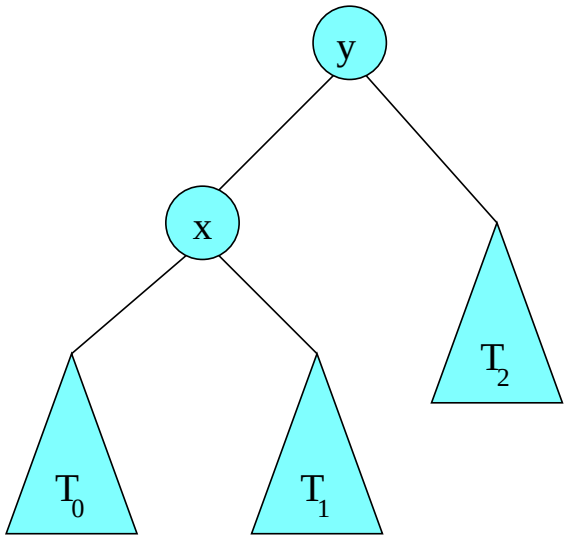

Om $b = x$ kallas det en dubbel rotation.
"Rotera upp $x$ över $y$ och sedan över $z$"

# Fyra olika rotationer



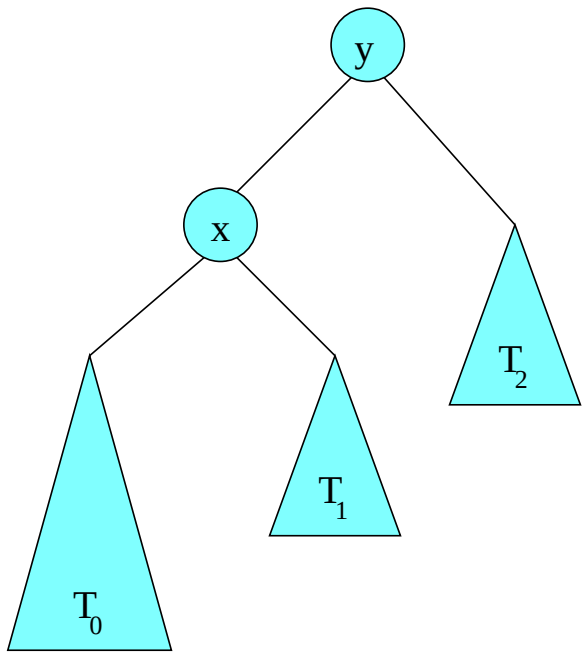

Om $b = x$ kallas det en dubbel rotation.
"Rotera upp $x$ över $y$ och sedan över $z$"

# Ett annat sätt att beskriva det på



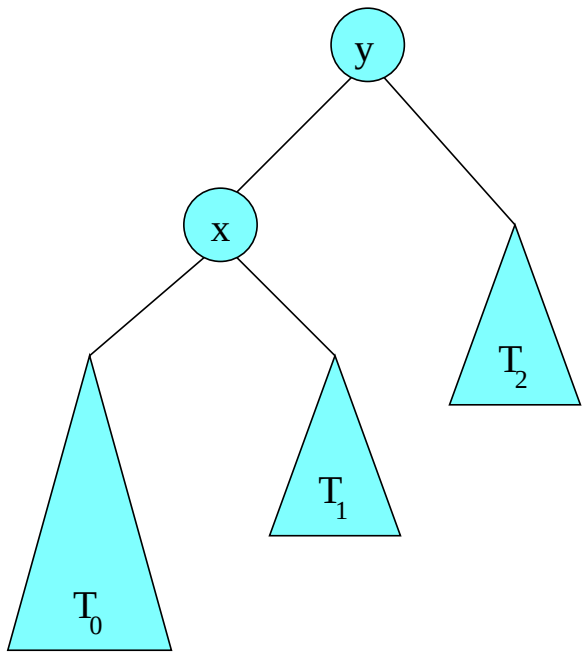

Antag att vi har balans...

# Ett annat sätt att beskriva det på



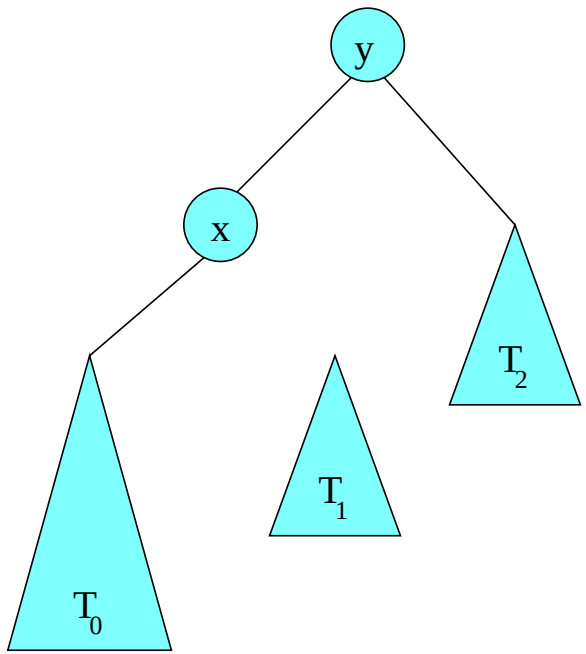

...och sedan stoppar in något som förstör den

# Ett annat sätt att beskriva det på



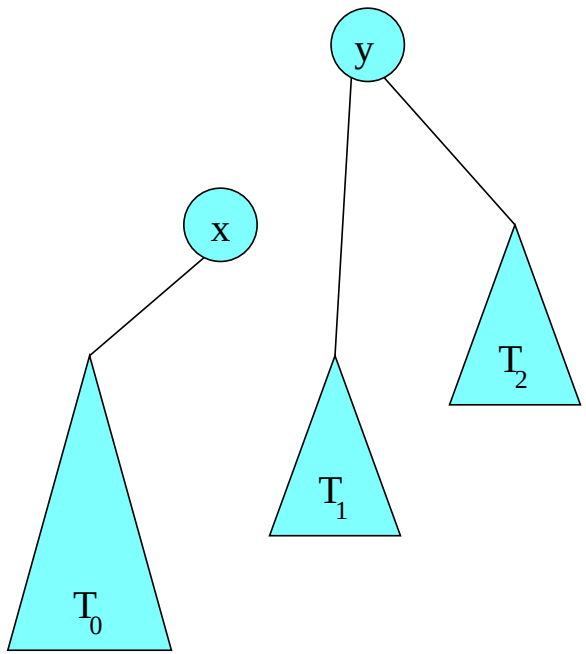

Gör en enkel rotation

# Ett annat sätt att beskriva det på



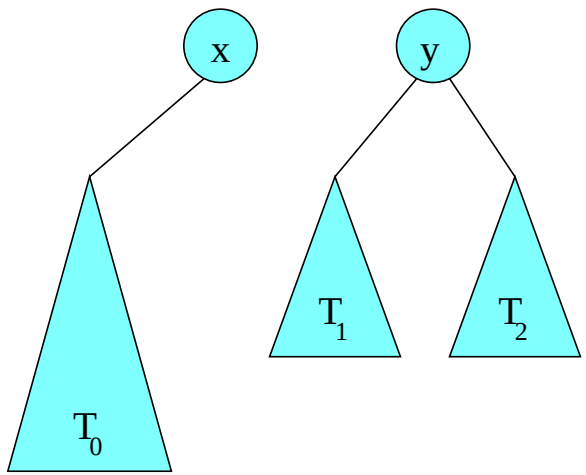

Gör en enkel rotation

# Ett annat sätt att beskriva det på



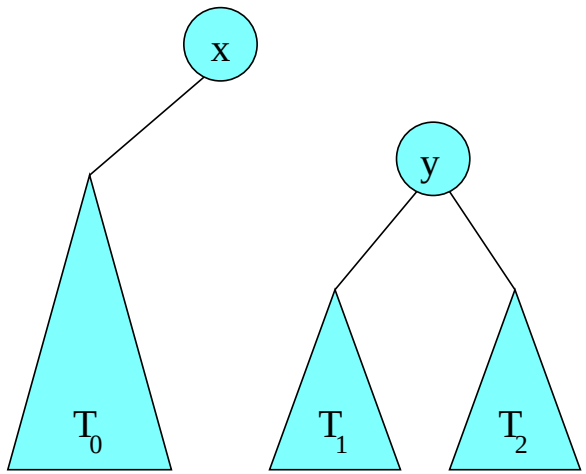

Gör en enkel rotation

# Ett annat sätt att beskriva det på



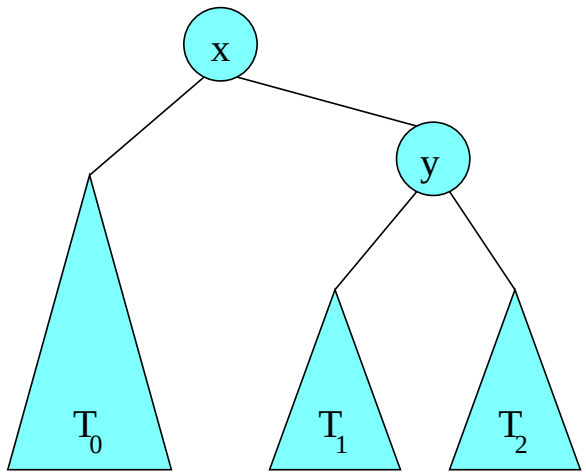

Gör en enkel rotation

# Ett annat sätt att beskriva det på



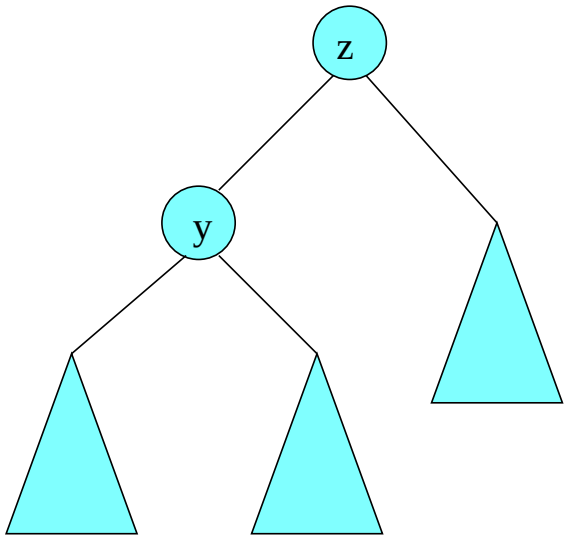

Gör en enkel rotation

# Ett annat sätt att beskriva det på



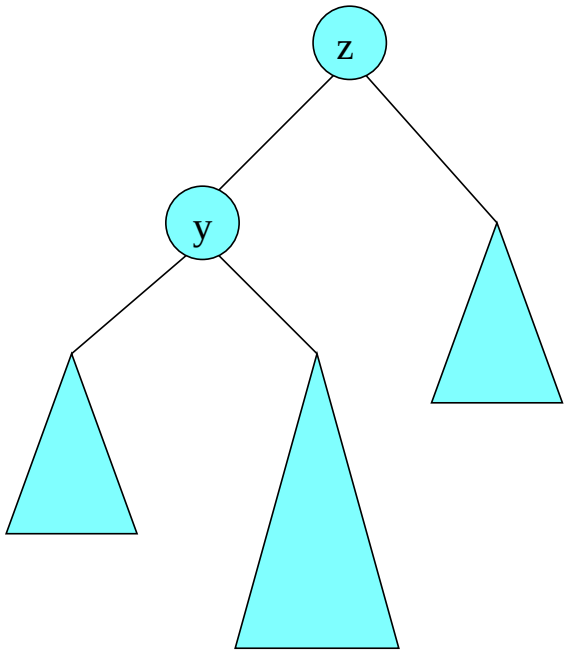

Klart!

# Ett annat sätt att beskriva det på



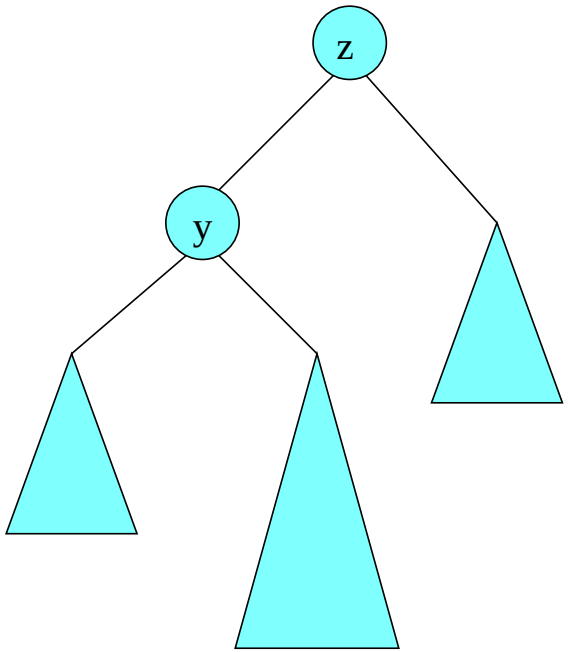

Ett nytt exempel...

# Ett annat sätt att beskriva det på



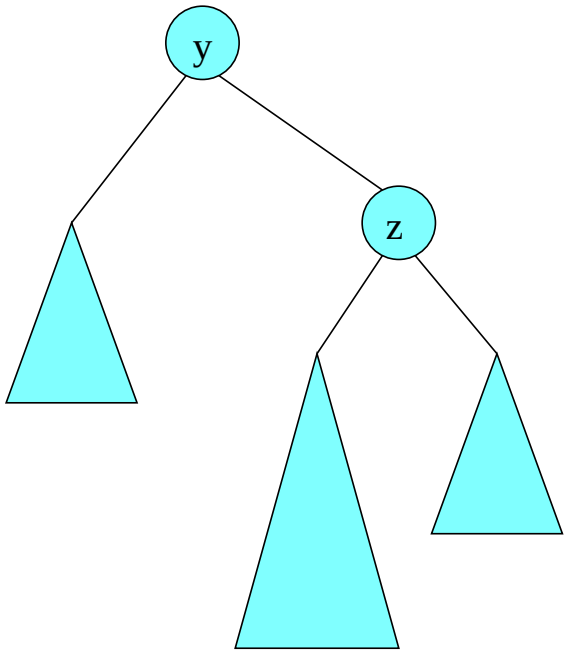

...den här gången stoppar vi in något på ett annat ställe

# Ett annat sätt att beskriva det på



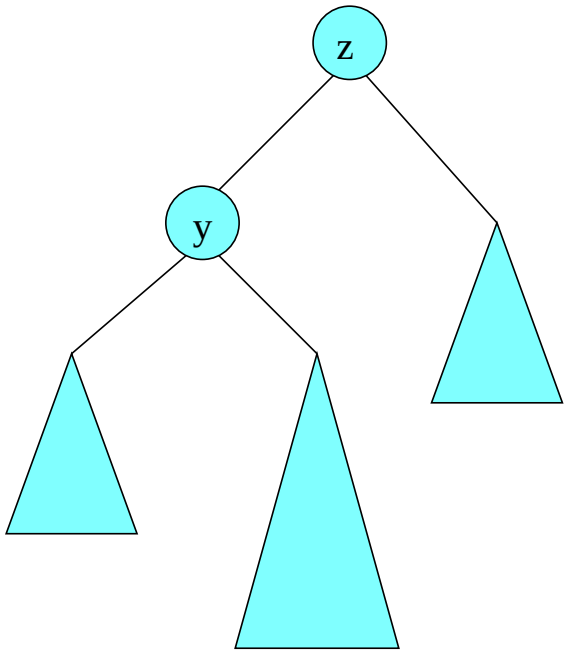

Prova en enkel rotation igen...

# Ett annat sätt att beskriva det på



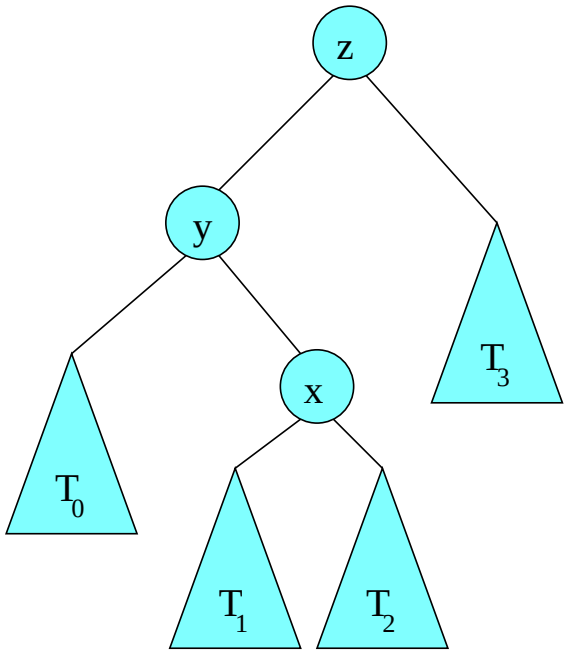

…hmm, vi har inte fått balans

# Ett annat sätt att beskriva det på



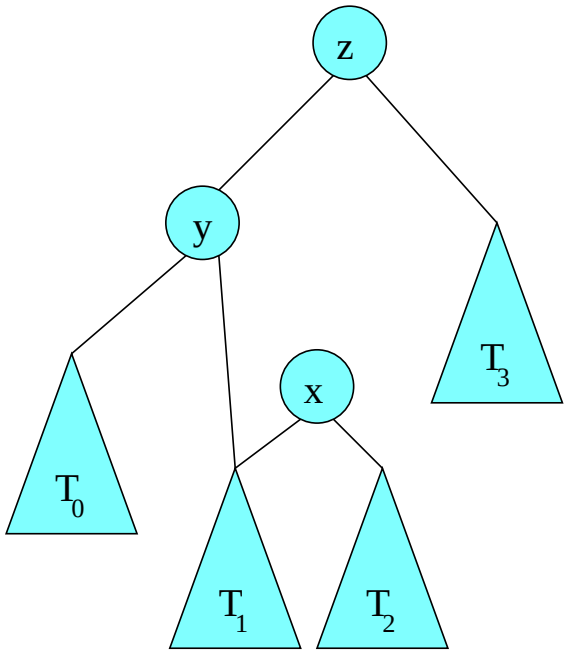

Börja om från början...och titta på strukturen i $y$

# Ett annat sätt att beskriva det på



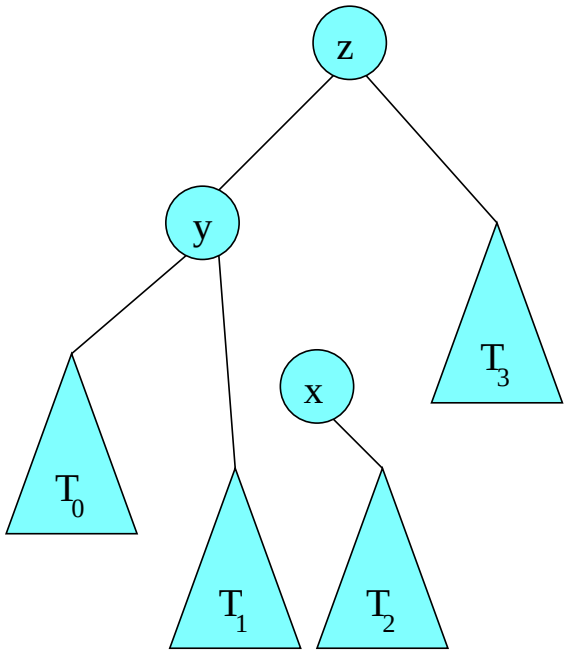

Vi får lov att göra en dubbel rotation

# Ett annat sätt att beskriva det på



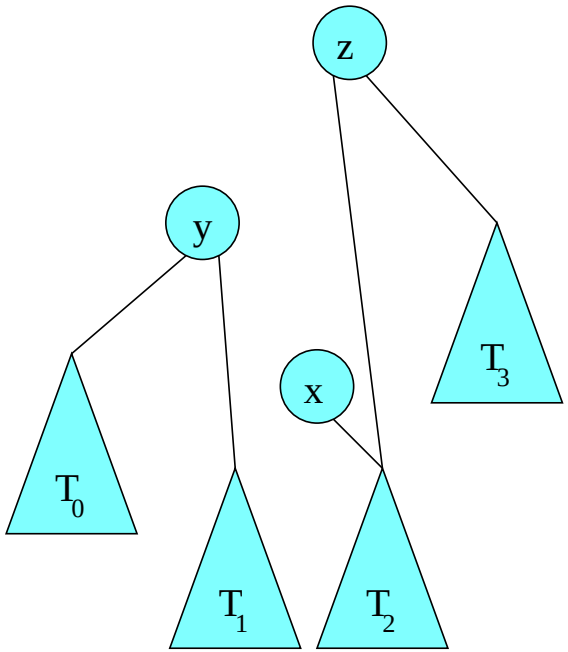

Vi får lov att göra en dubbel rotation

# Ett annat sätt att beskriva det på



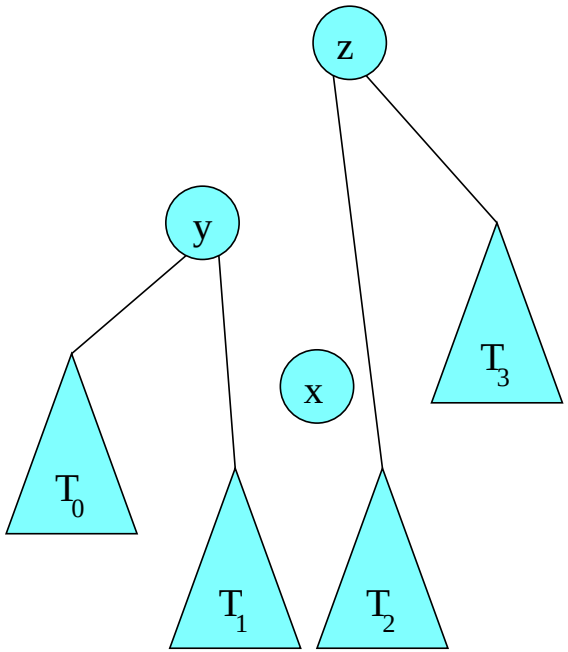

Vi får lov att göra en dubbel rotation

# Ett annat sätt att beskriva det på



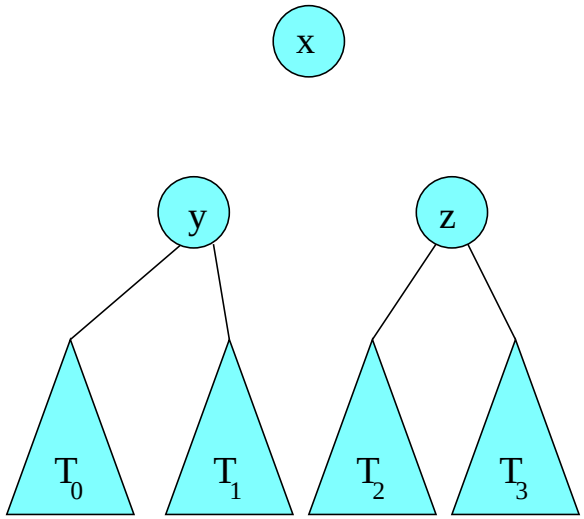

Vi får lov att göra en dubbel rotation

# Ett annat sätt att beskriva det på



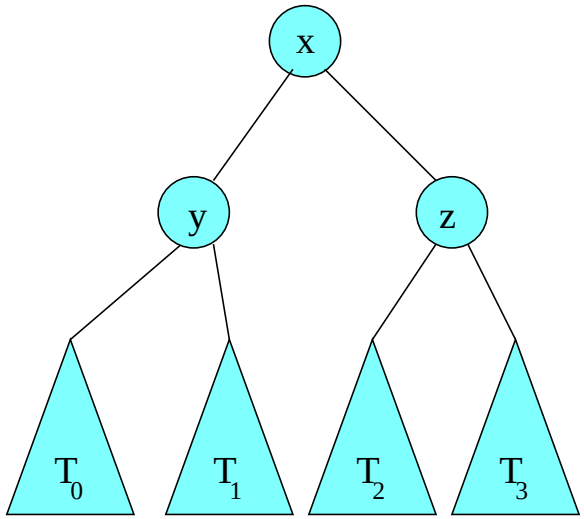

Vi får lov att göra en dubbel rotation

# Ett annat sätt att beskriva det på




Vi får lov att göra en dubbel rotation

# Ett annat sätt att beskriva det på

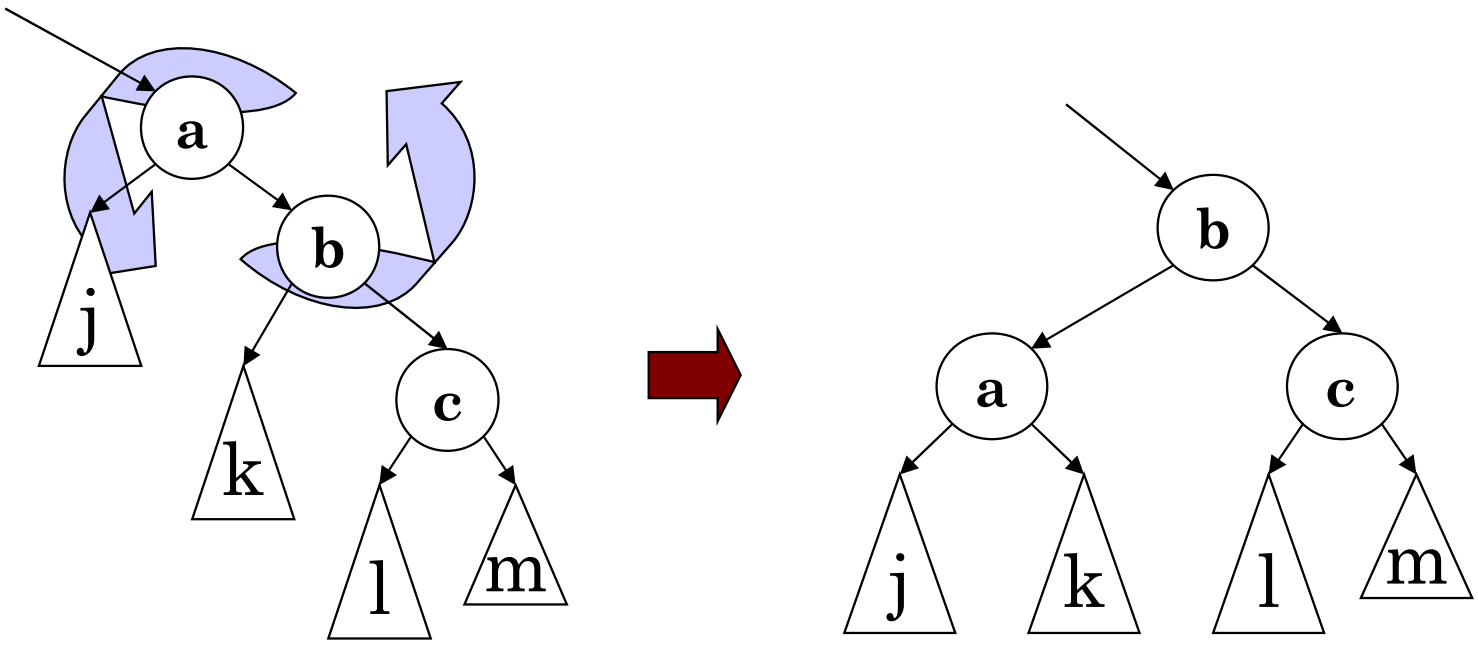



Klart!

# Trenodsrekonstruering = rotationer...

Vissa författare använder vänster- och högerrotationer: Enkel vänsterrotation:

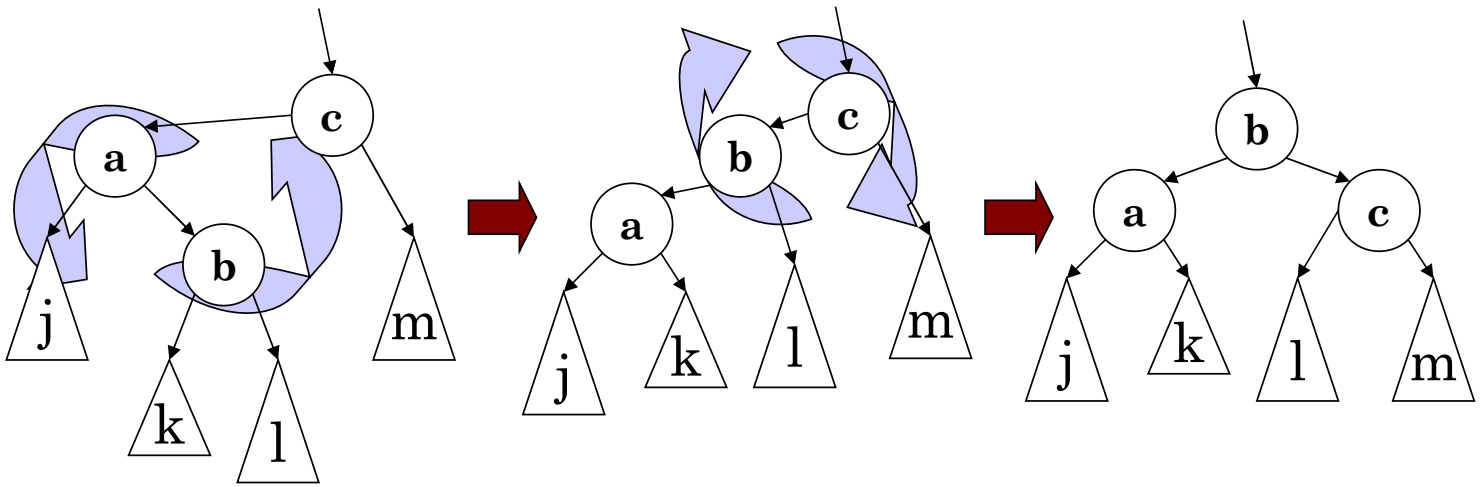
- vänstra delen av delträdet ($a$ och $j$) sänks ner
- vi har "roterat (upp) $b$ över $a$"



# Dubbla rotationer...

Två rotationer behövs när noderna som ska balanseras om är placerade i ett sicksackmönster.

- Rotera upp $b$ över $a$
- Rotera upp $b$ över $c$



# Borttagning i ett AVL-träd

- find och remove som i ett vanligt binärt sökträd
- Uppdatera balansinformationen på väg tillbaka upp till roten
- Om för obalanserat: Strukturera om ...men...
  - När vi återställer balansen på ett ställe kan det uppstå obalans på ett annat
  - Måste upprepa balanseringen (eller kontroll av balansen) till dess vi når roten
  - Högst $O(\log n)$ ombalanseringar

# Ny approach: släpp på något av kraven

- AVL-träd: *binärt* träd, accepterar viss (liten) obalans...
- Kom ihåg:
  Fullt binärt träd: icke-tomt; graden är antingen 0 eller 2 för varje nod
  Perfekt binärt träd: fullt, alla löv har samma djup
- Kan vi bygga och underhålla ett perfekt träd (om vi struntar i ”binärt”)?
  Då skulle vi alltid känna till söktiden i värsta fall exakt!

# $(2, 3)$-träd

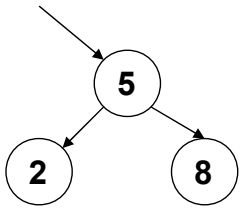
Förut:

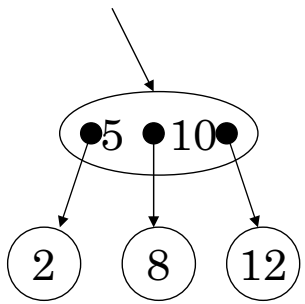
- Ett "pivotelement"
- Om större letar vi till höger
- Om mindre letar vi till vänster

Nu:

- Tillåt flera (nämligen 1–2) "pivotelement"
- Antalet barn till en intern nod är antalet pivotelement + 1 (dvs 2–3)

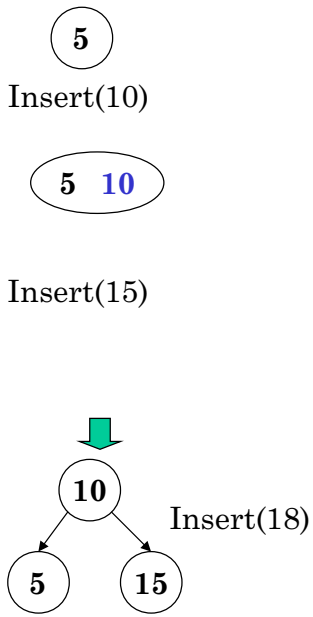
# $(2, 3)$-träd



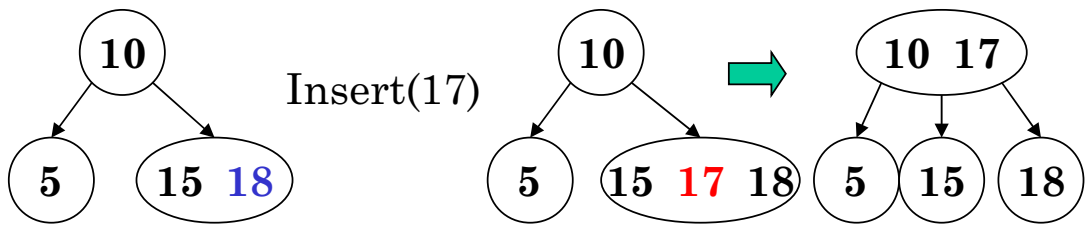
# Insättning i ett $(a, b)$-träd med $a = 2$ och $b = 3$

- Så länge det finns plats i barnet vi hittar, lägg till elementet i det barnet...



# Insättning i ett $(a, b)$-träd med $a = 2$ och $b = 3$

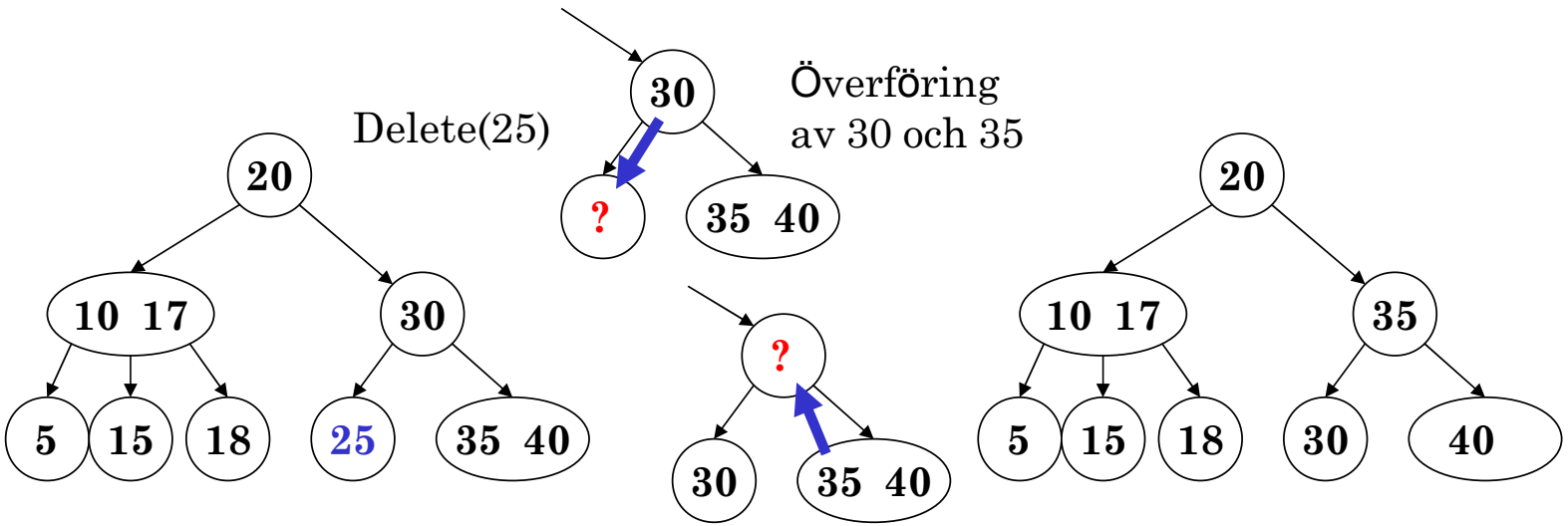
- Om fullt, dela upp och tryck det utvalda pivotelementet uppåt...
  ...detta kan hända upprepade gånger



# Borttagning i $(2, 3)$-träd

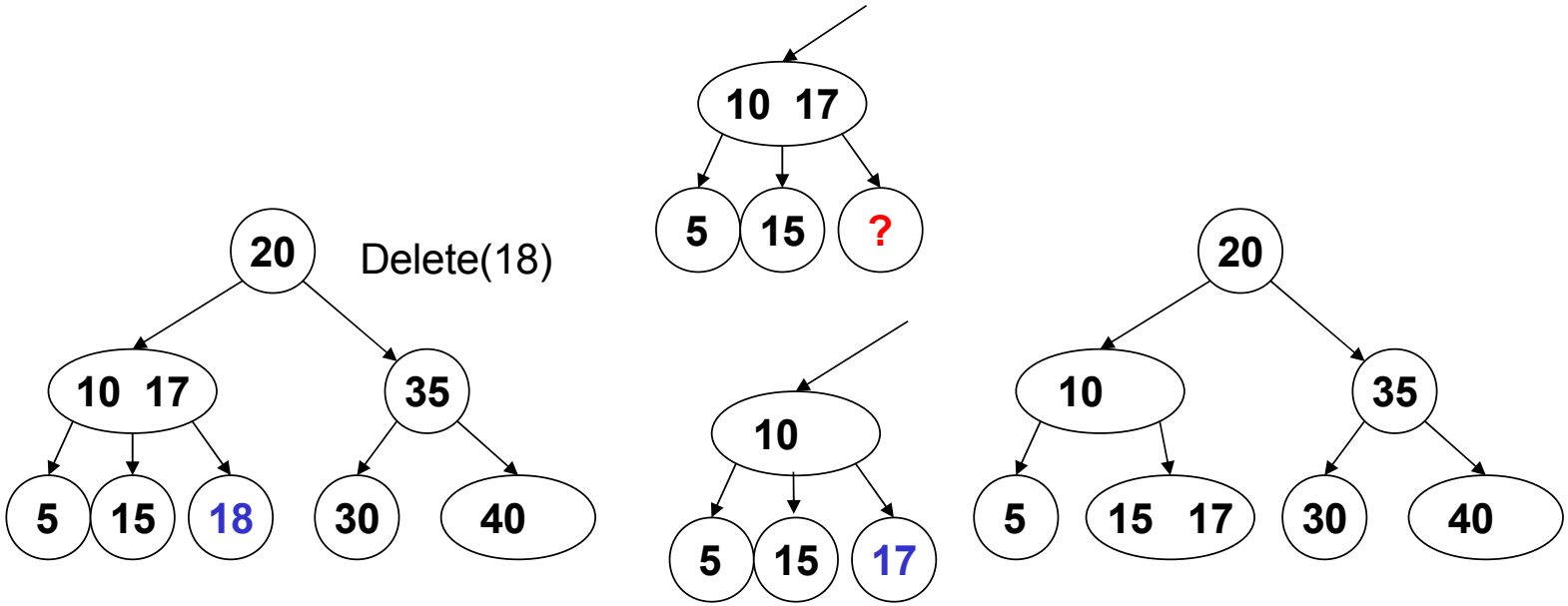
Tre fall:

- Inga villkor bryts genom borttagning
- Ett löv tas bort (blir tomt)
  För då över någon annan nyckel till det lövet,
  ...ok om vi har syskon med 2+ element



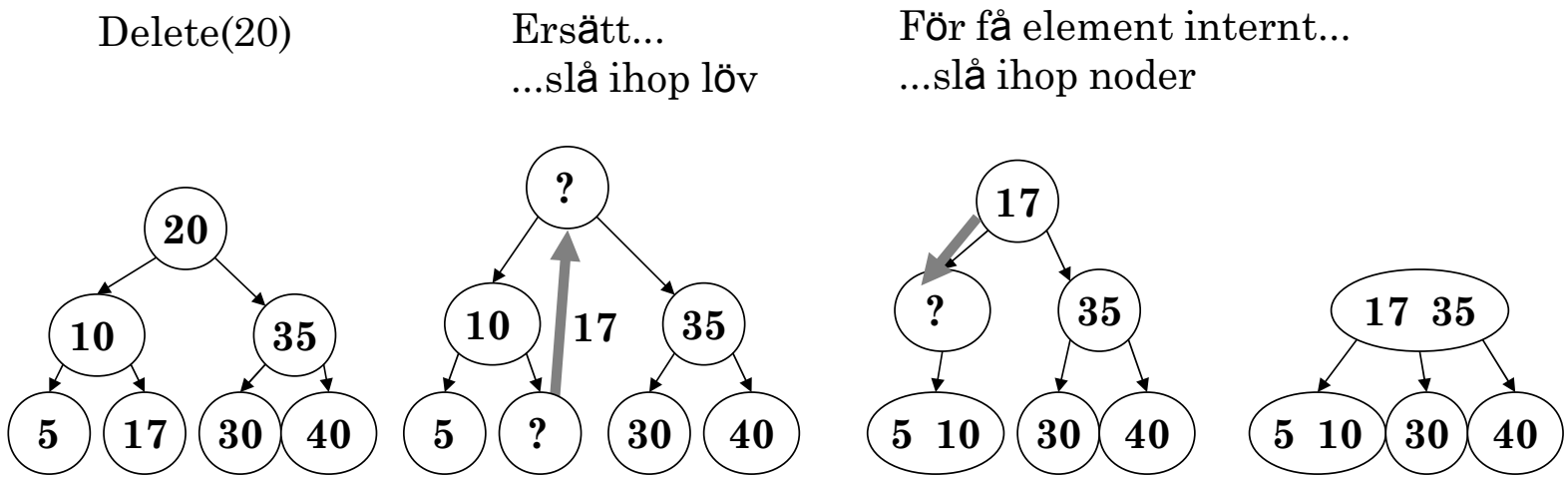
# Borttagning i $(2, 3)$-träd

- Om ett löv tas bort (blir tomt)
- För då över någon annan nyckel till det lövet, eller
- Slå ihop det med en granne



# Borttagning i $(2, 3)$-träd

- En intern nod blir tom
  Roten: ersätt med föregångare eller efterföljare i inorder
  Reparera sedan inkonsistenser med lämpliga ihopslagningar och överföringar...



# Nästa gång:
# Prioritetsköer och heapar

www.liu.se

LINKÖPING UNIVERSITY