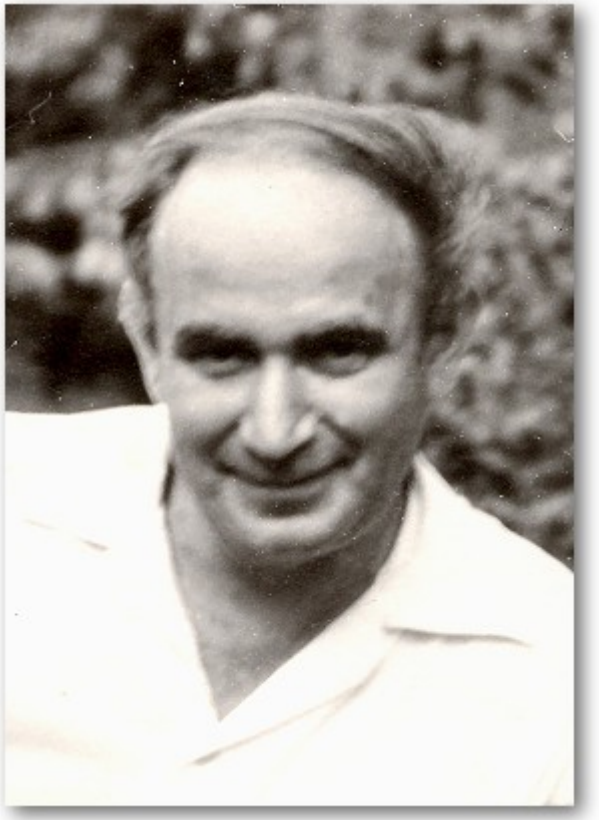
Selv-balanserende søketrær

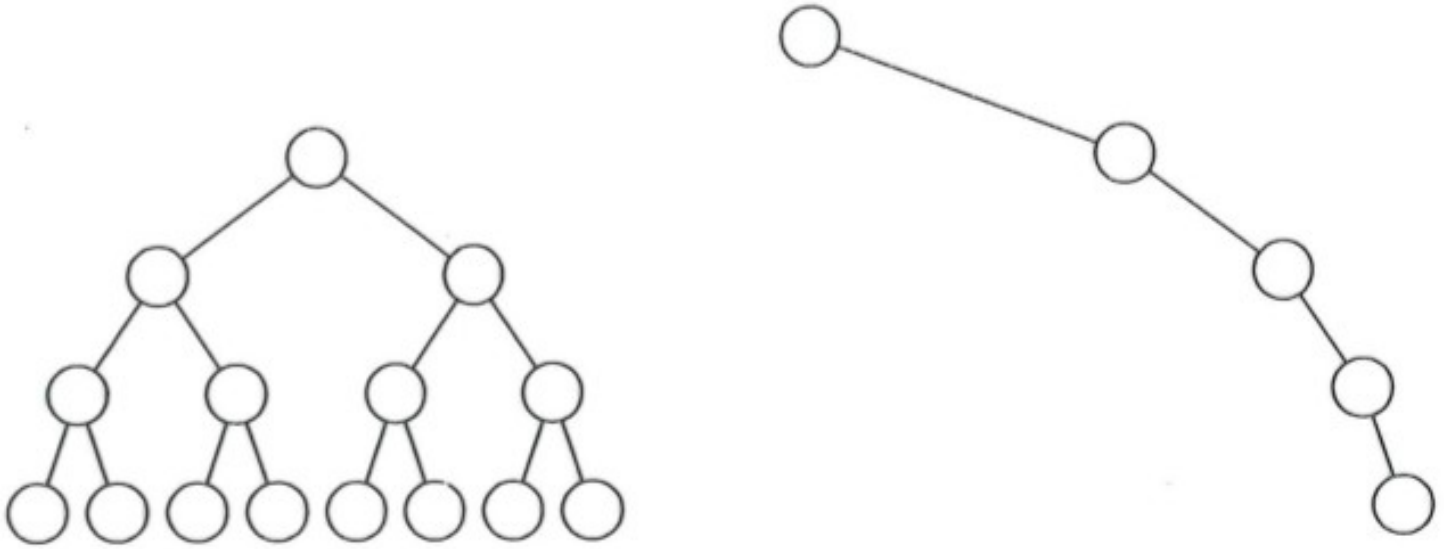
Georgy Maksimovich Adelson-Velsky



Evgenii Mikhailovich Landis

Søketrær og effektivitet

- *O*(log *n*) effektivitet av binære søketrær kan ikke *garanteres*



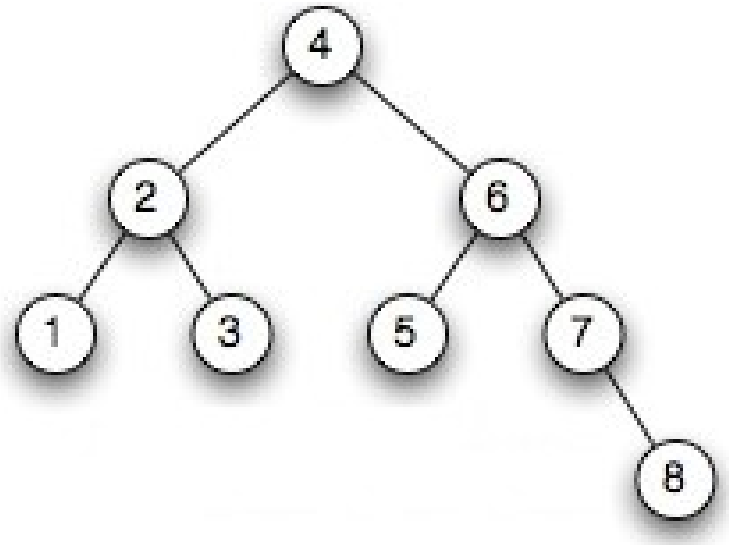
- Treet til venstre har høyde lik log *n*
- Treet til høyre har høyde lik *n*

Selv-balanserende søketrær

- Implementerer innsetting og fjerning av verdier slik at ubalanse i treet *automatisk rettes opp* hvis nødvendig
- Har *alltid* O(log *n*) effektivitet
- Krever mer komplisert programkode med mye «pekerfikling»
- Noen varianter av selv-balanserende trær:
  - AVL-tre
  - Red-black tree (Java Collections, læreboken)
  - Splay-tree
  - Scapegoat tree

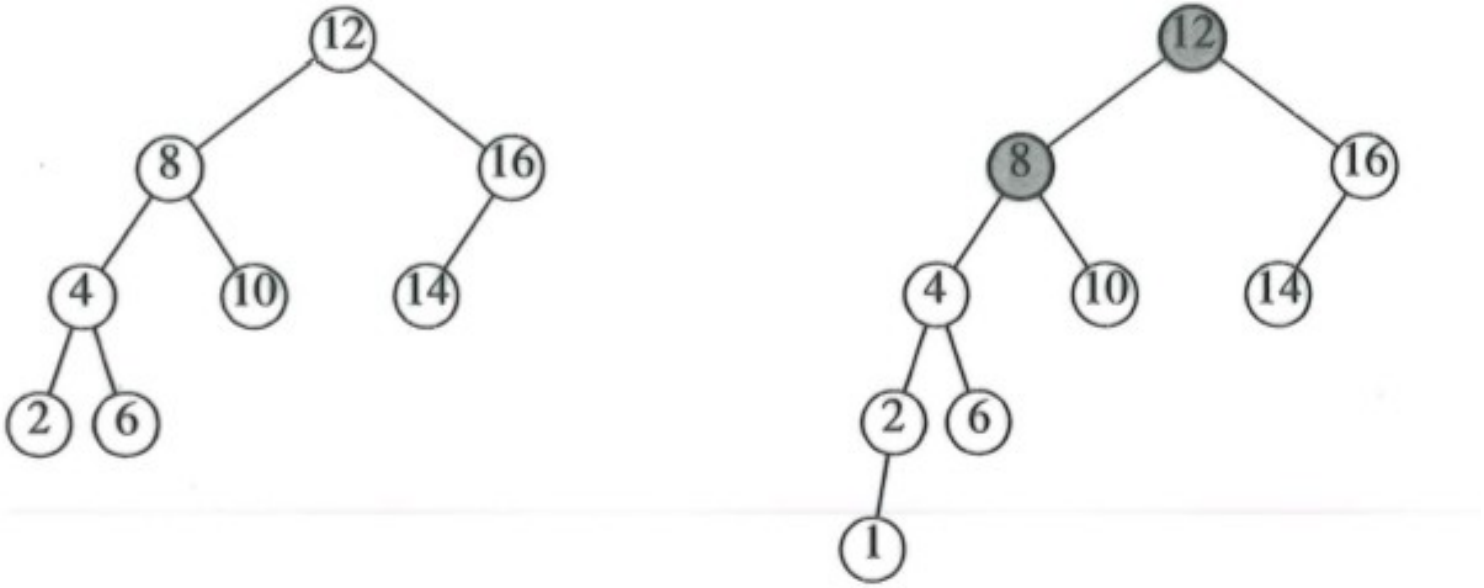
AVL-trær

- Første publiserte selv-balanserende binære søketre, oppkalt etter de to oppfinnerne *
- Er sortert på samme måte som et søketre
- Krav til balanse:
  - For alle noder i treet skal forskjellen i høyde på nodens venstre og høyre subtre maksimalt være lik 1



- Kan bevises at høyden av et AVL-tre *alltid* er $O(\log n)$

* Georgy M. **A**delson-**V**elsky.; Evgenii M. **L**andis: "An algorithm for the organization of information", *Proceedings of the USSR Academy of Sciences*, 1962



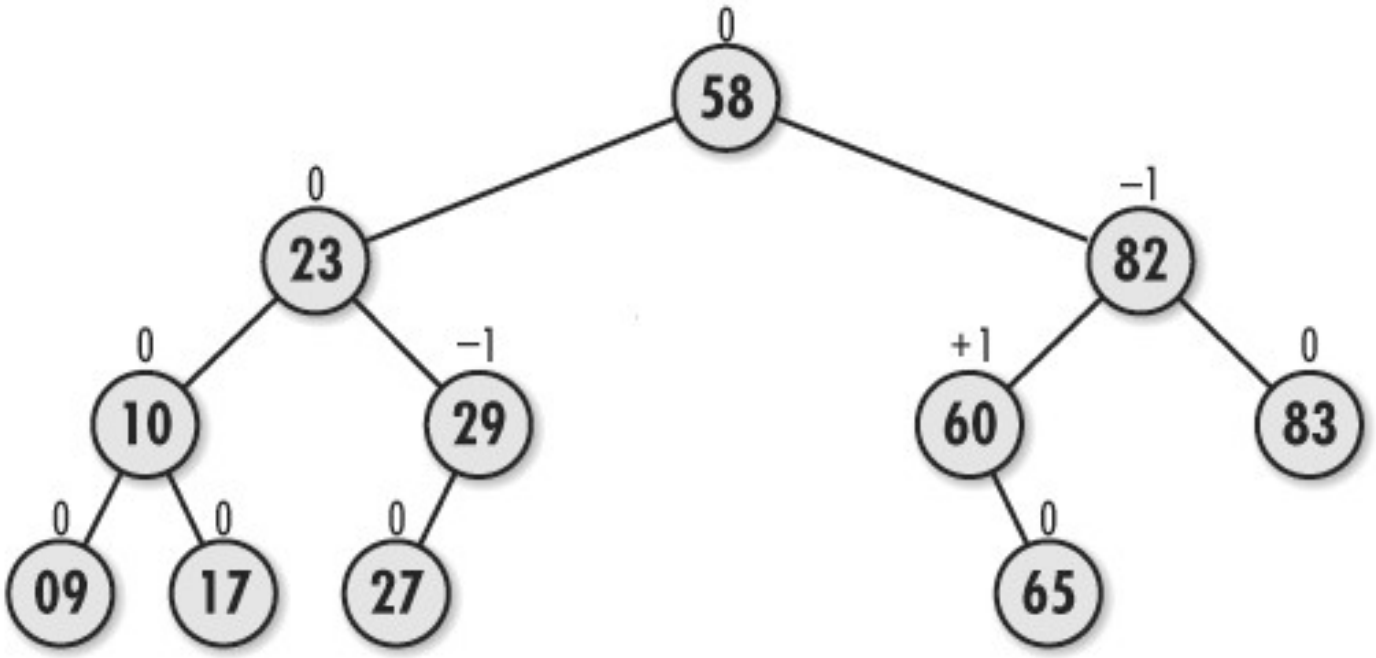
- Treet til venstre er et AVL-tre, alle nodene er i balanse
- Treet til høyre er ikke et AVL-tre, de to mørke nodene bryter med balansekravet
- Eksempel som tester om et søketre også er et AVL-tre:

  Javakode med testprogram

Balansefaktorer

- Hver node i et AVL-tre lagrer et heltall som kalles en *balansefaktor,* i tillegg til data og de to pekerne til høyre og venstre subtre
- Balansefaktoren er *forskjellen i høyde* mellom nodens venstre og høyre subtre:
  - Lik 0: Subtrærne er like høye
  - Negativ: Venstre subtre er høyest
  - Positiv: Høyre subtre er høyest

AVL-tre med balansefaktorer



Alle nodene i et AVL-tre *skal* ha balansefaktor lik -1, 0 eller 1

Søking, innsetting og fjerning i AVL-tre

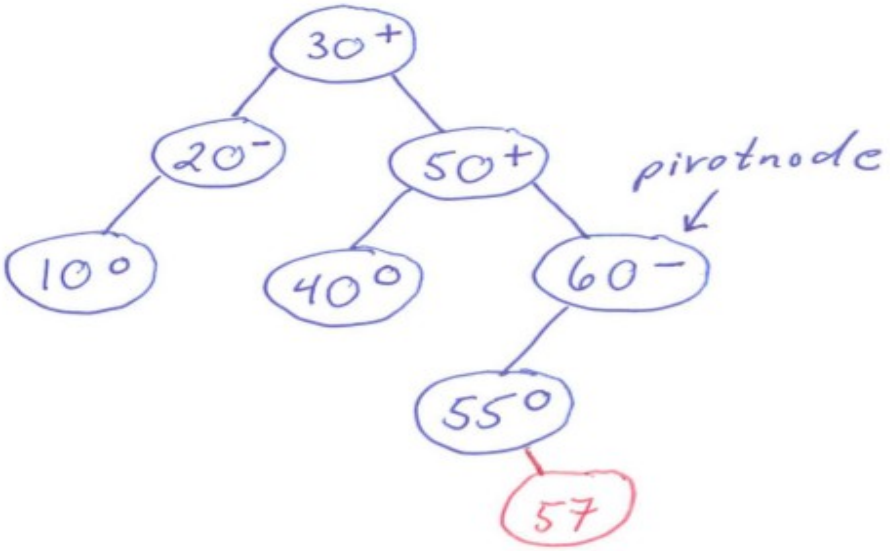
- Søking i et AVL-tre gjøres på samme måte som søking i et vanlig binært søketre
- Innsetting og fjerning i AVL-tre:
  - Gjøres først på samme måte som innsetting og fjerning i et vanlig binære søketre
  - Retter deretter opp evt. ubalanse i treet som kan ha oppstått ved innsetting/fjerning av en node
- Vi skal bare se på hvorledes innsetting i AVL-tre gjøres

Innsetting i AVL-tre

- Sett inn ny node som i et vanlig søketre
- Justér balansefaktoren i alle nodene på søkeveien fra roten ned til ny node, slik at den fortsatt er korrekt etter innsetting
- Hvis en node får balansefaktor lik -2 eller 2, er treet kommet i ubalanse etter innsettingen
- Ved ubalanse flyttes noder rundt i treet slik at balansen gjenopprettes

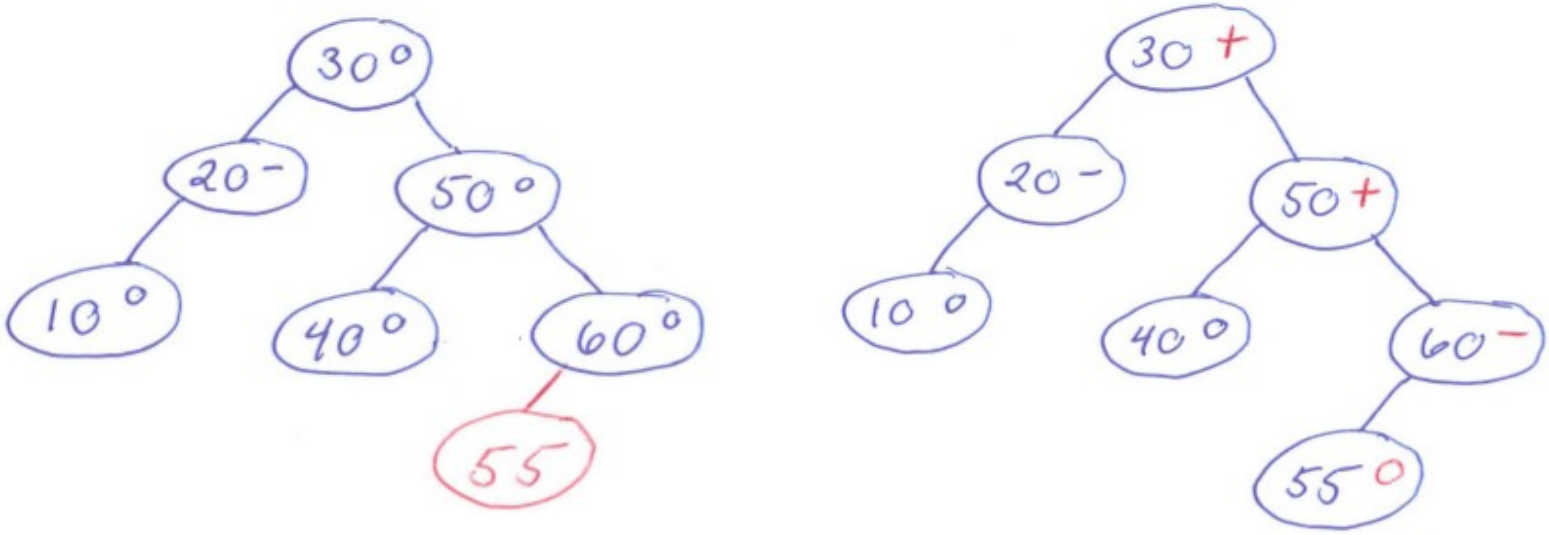
Innsetting i AVL-tre: Pivotnoden

- Pivotnode:
  - *Siste* node på veien ned til ny node som har balanse-faktor som *ikke* er lik 0
  - Ved ubalanse er det *alltid* tilstrekkelig at vi bare justerer balansen i subtreet som har rot i pivotnoden
  - Justeringen gjøres ved å flytte noder i treet med *rotasjoner* rundt pivotnoden
  - Flere ulike tilfeller som må behandles separat



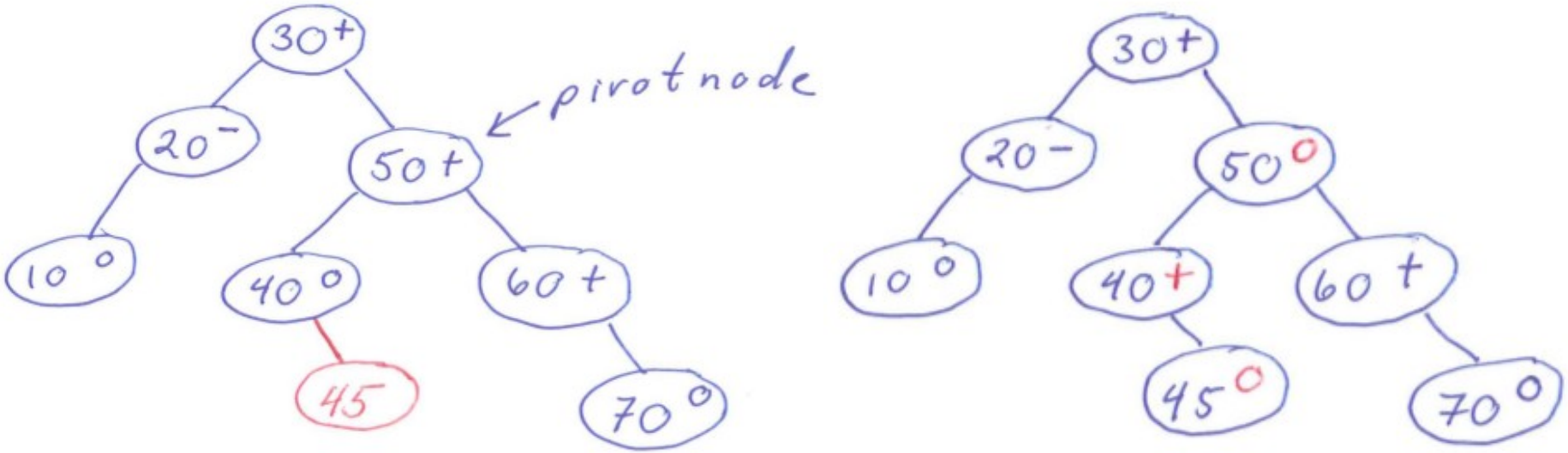
Tilfelle 1: Det finnes ingen pivotnode

- Alle noder på søkeveien ned til den nye noden som er satt inn har balansefaktor lik 0
- Innsettingen kan ikke forårsake ubalanse, ingen flytting av noder er nødvendig
- Trenger bare å justere balansefaktoren for alle nodene som ligger på søkeveien



Tilfelle 2: Ny node i pivotnodes minste subtre

- Det finnes en pivotnode på veien ned til ny node
- Ny node er satt inn i pivotnodens *korteste* subtre
- Kan ikke forårsake ubalanse i treet, trenger ikke flytte noder
- Justerer balansefaktor for alle noder fra og med pivotnoden og ned til den nye noden som er satt inn

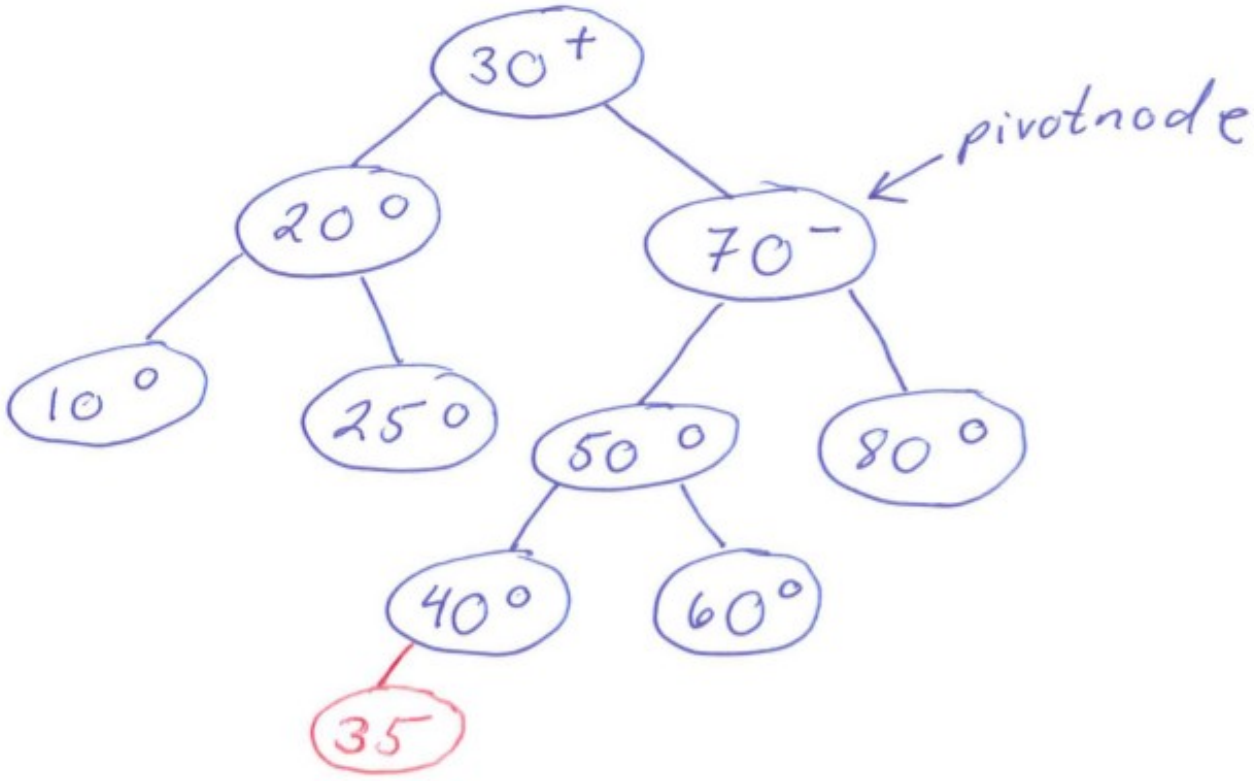


Tilfelle 3:
Ny node i pivotnodes høyeste subtre

- Det finnes en pivotnode på veien ned til ny node
- Ny node er satt inn i pivotnodens *høyeste* subtre
- Pivotnoden blir ubalansert
- Treet må omstruktureres for å opprettholde AVL-egenskapene
- To (fire) tilfeller av omstrukturering:
  - Enkel rotasjon (to symmetriske tilfeller)
  - Dobbel rotasjon (to symmetriske tilfeller)

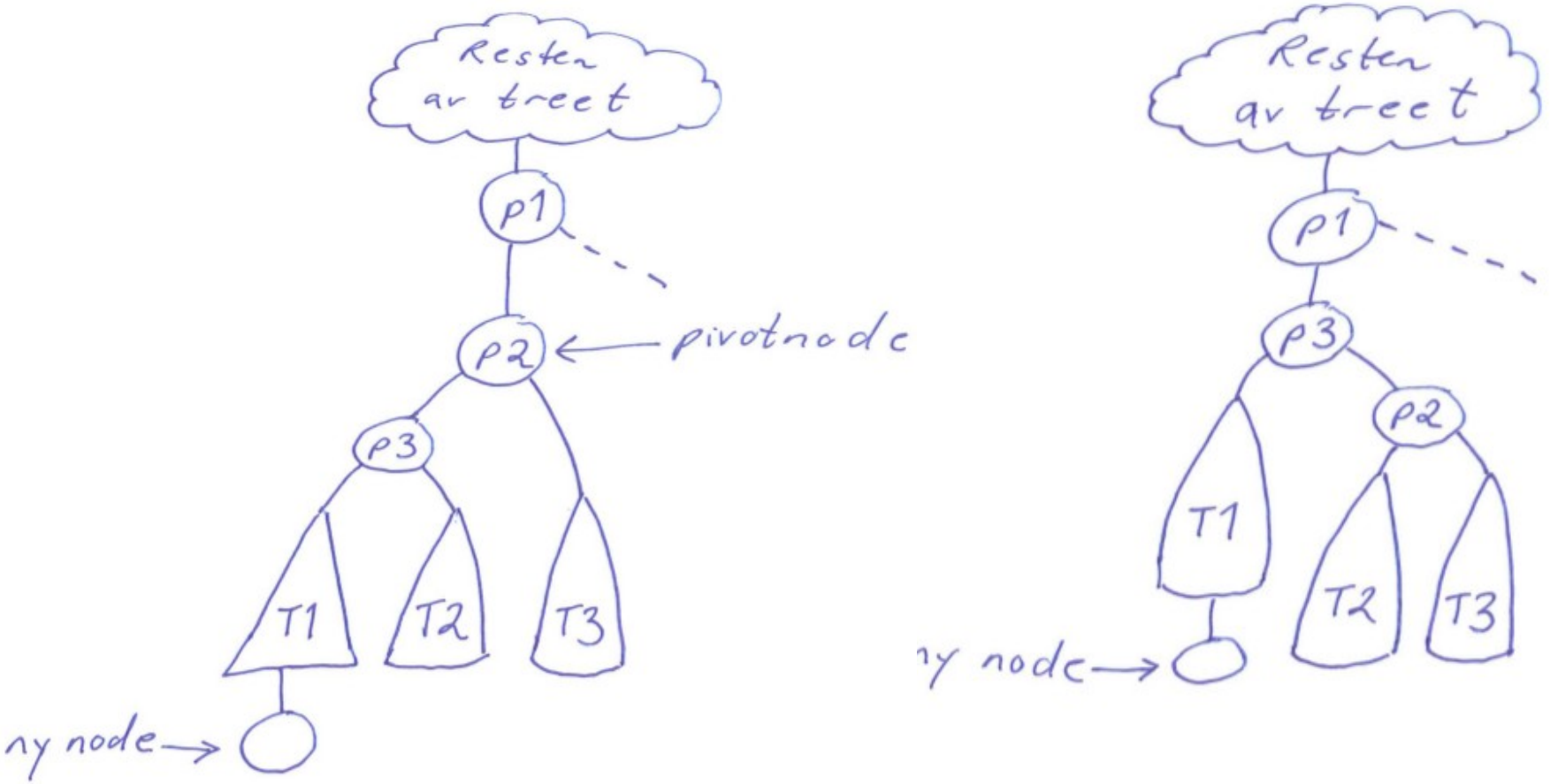
Eksempel som krever enkel rotasjon

- Ny node settes inn i *venstre* subtre til pivotnodens *etterfølger* på søkestien:

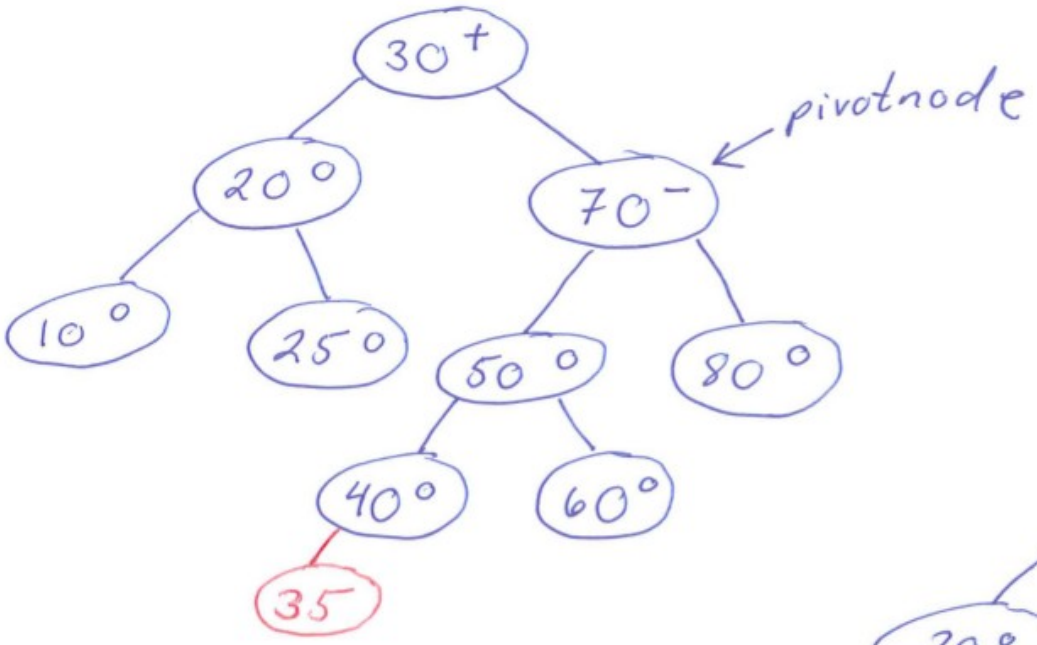


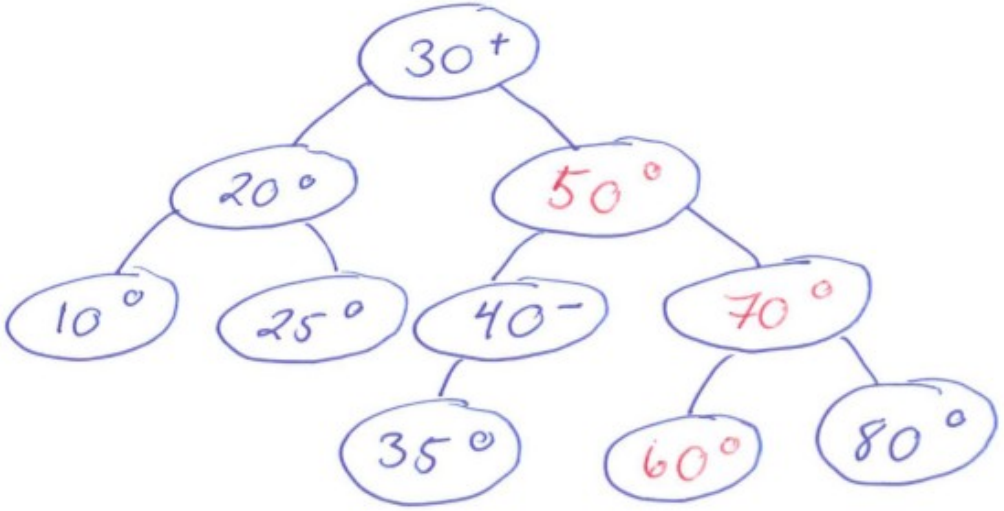
Enkel rotasjon

(symmetrisk for noden **p3** til høyre)



Enkel rotasjon: Eksempel

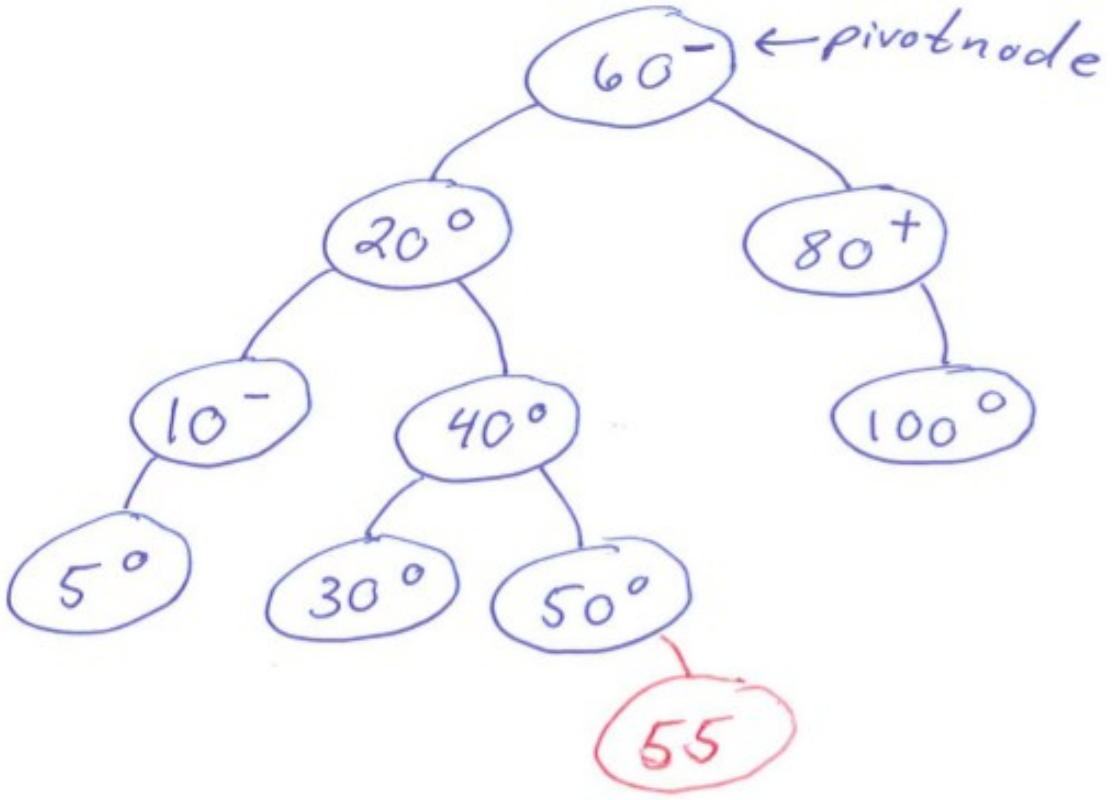




Animasjon av AVL-tre

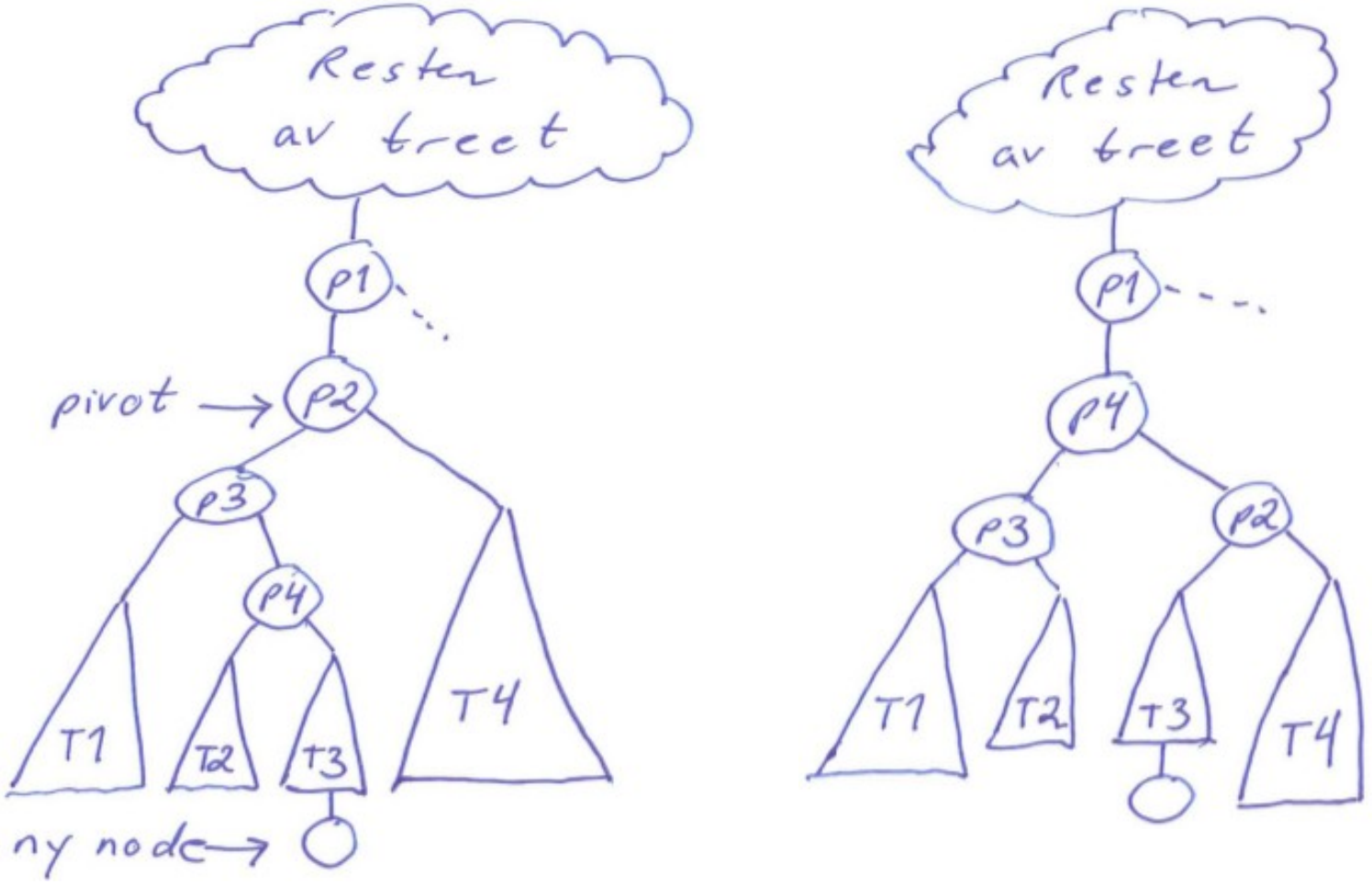
Eksempel som krever dobbel rotasjon

- Ny node settes inn i *høyre* subtre til pivotnodens *etterfølger* på søkestien:

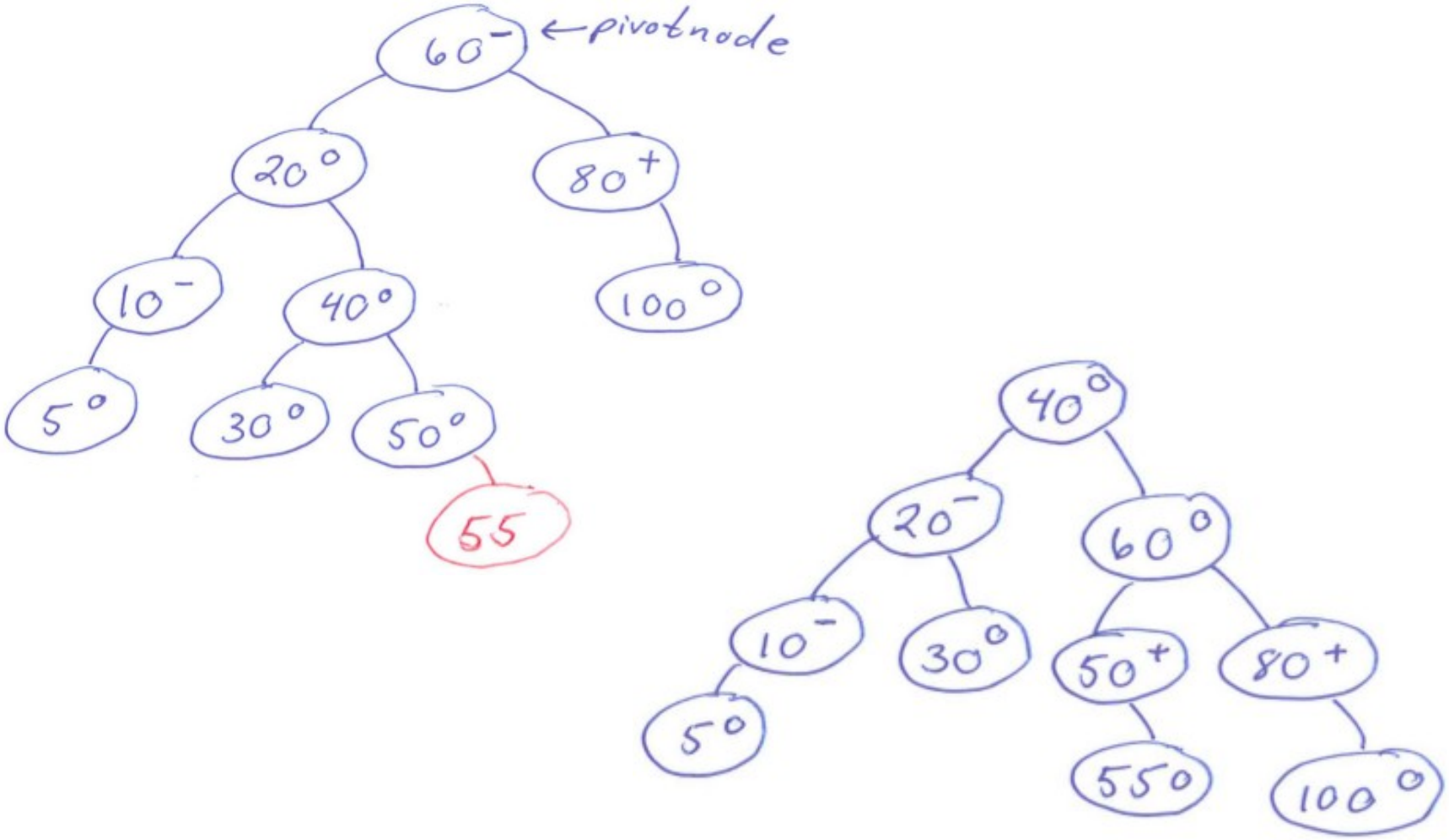


Dobbel rotasjon

(symmetrisk for noden **p3** til høyre)



Dobbel rotasjon: Eksempel



Animasjon av AVL-tre

AVL-trær: Implementasjon og effektivitet

- Innsetting i AVL-trær krever relativt «fiklete» kode
- Fjerning med rebalansering er enda verre...
- Men: Rebalansering er i verste fall en $O(\log n)$ operasjon
- AVL-trær garanterer derfor alltid $O(\log n)$ effektivitet for søking, innsetting og fjerning
- Hvis vi *vet* at dataene kommer i random rekkefølge (som f.eks. i oblig. 4), er det tilstrekkelig å bruke vanlig binært søketre

Test: Selvbalanserende vs.vanlig binært søketre

- Test av vanlig søketre:
  - Innsetting av opp til ti millioner tall
  - Setter først inn i tilfeldig rekkefølge og deretter sortert
  - Bruker et egenprogrammert søketre med heltallsdata
  - Java-kode: testBSTRandomSorted.java
- Test av selvbalanserende søketre:
  - Tester på samme måte som for det vanlige søketreet, med først random og deretter sortert input
  - Bruker Java-klassen TreeMap som implementerer selvbalanserende red-black trær
  - Java-kode: testBalancedBSTRandomSorted.java