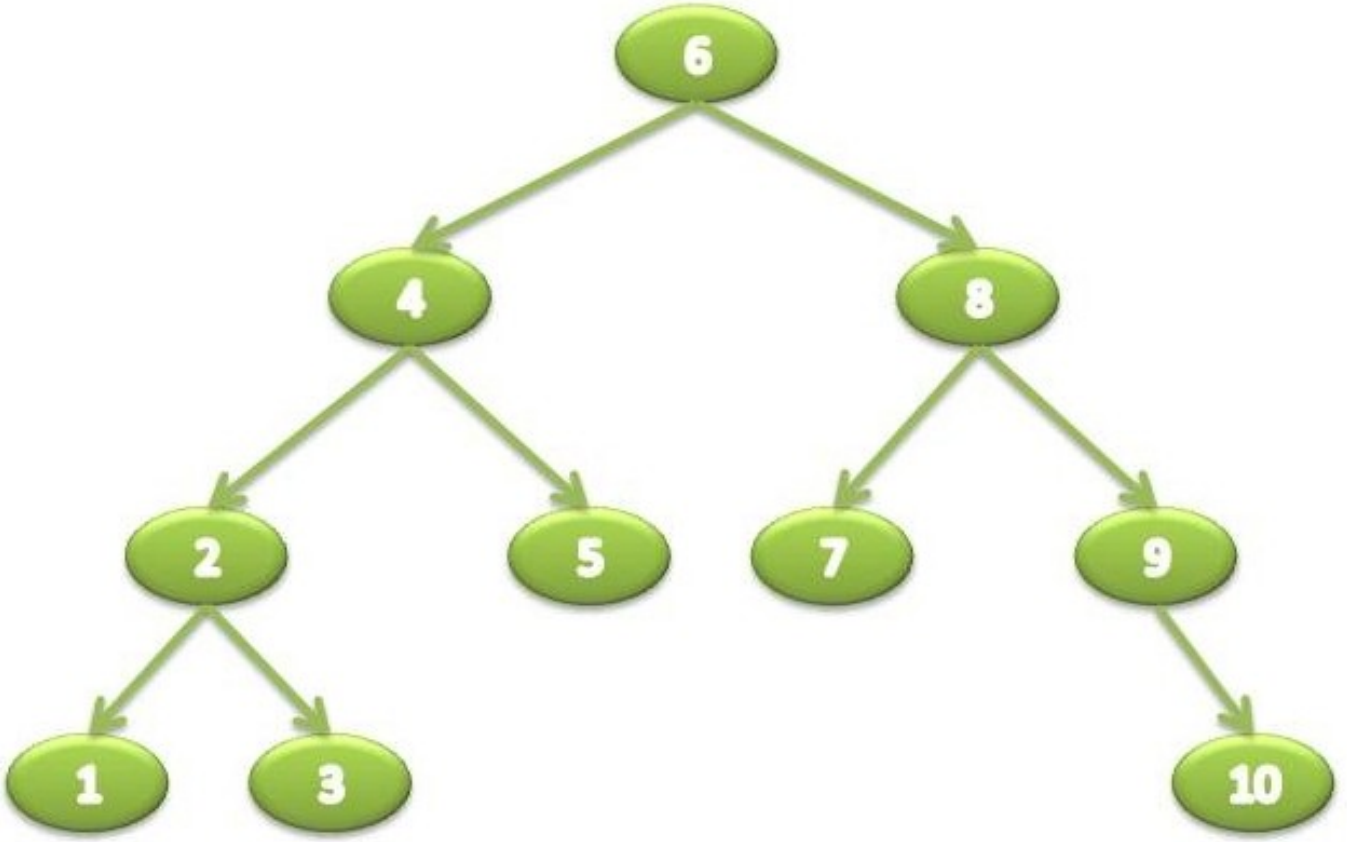
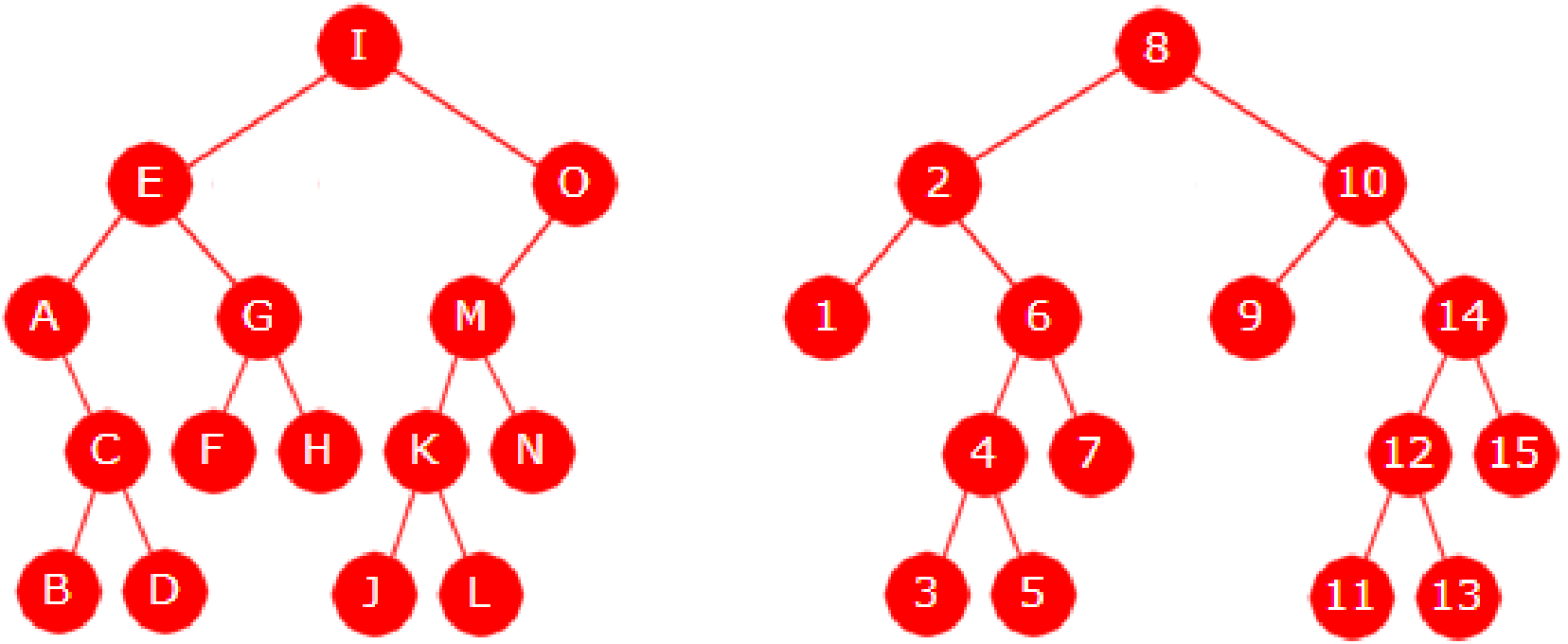
Binære søketrær

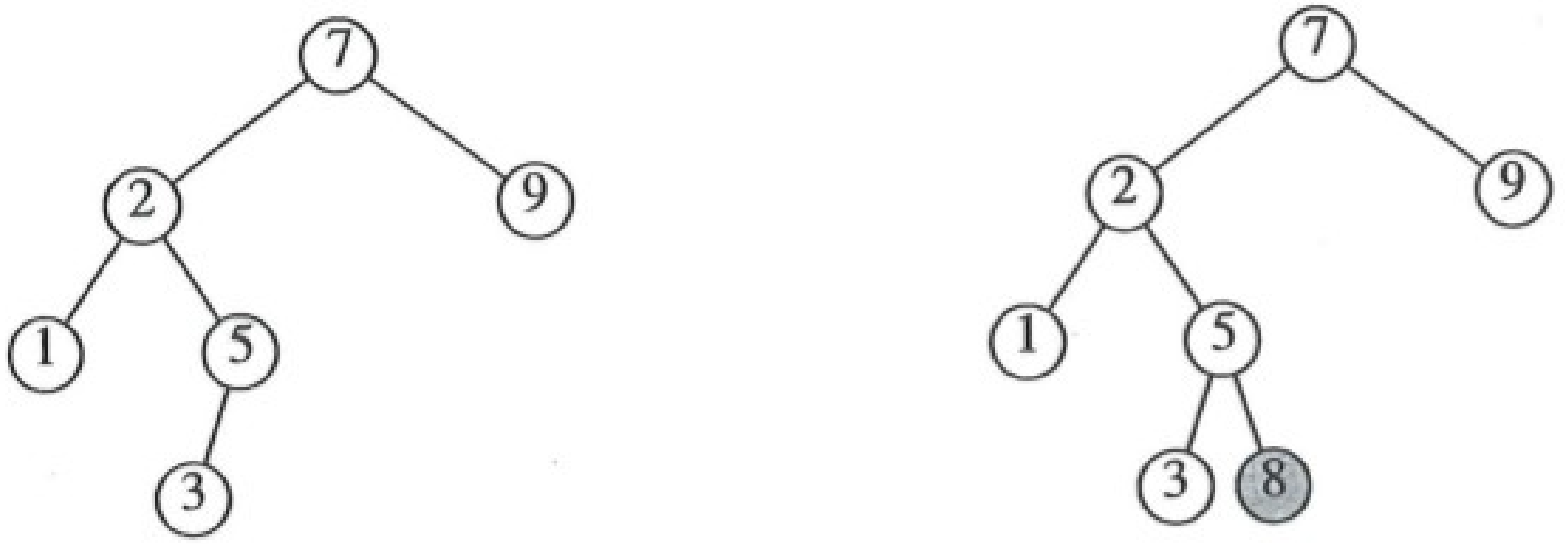


Definisjon av binært søketre

- For alle nodene i et binært søketre gjelder:
  - Alle verdiene i nodens *venstre* subtre er *mindre enn* verdien i noden
  - Alle verdiene i nodens *høyre* subtre er *større eller lik* verdien i noden
- Det binære søketreet er en *sortert* datastruktur:
  - Nodene oppsøkes i *stigende sortert* rekkefølge ved en *inorder* (venstre-roten-høyre) traversering av søketreet

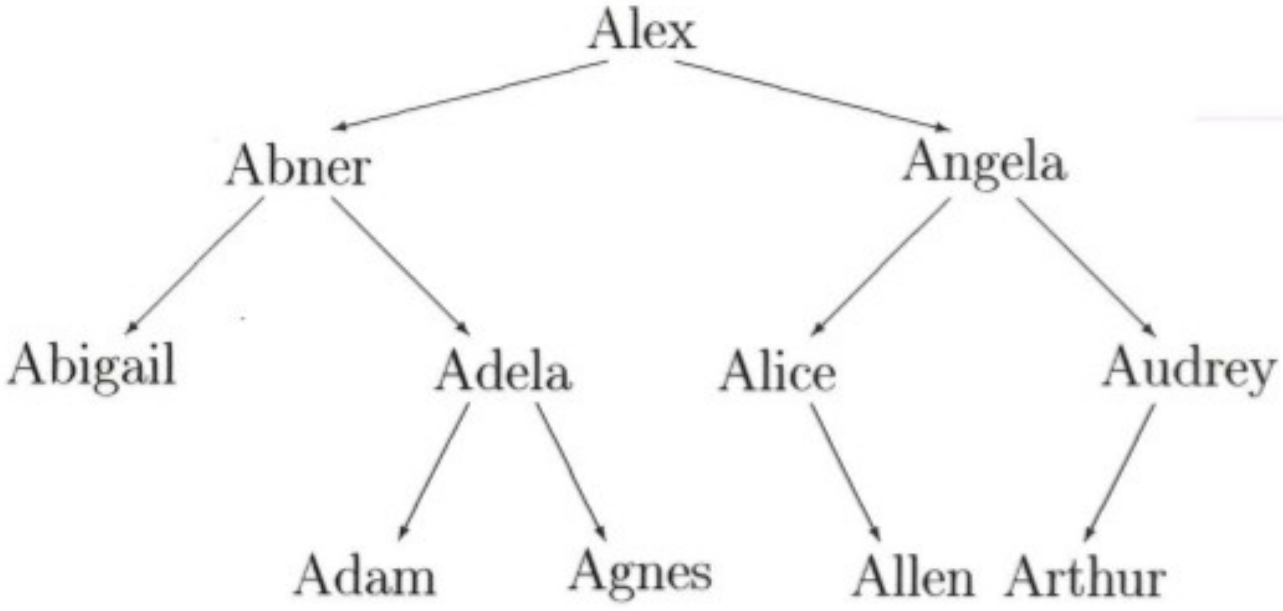
To binære søketrær





Two binary trees (only the left tree is a search tree)

Alfabetisk sortert søketre



- Inorder traversering:

  Abigail, Abner, Adam, Adela, Agnes, Alex, Alice, Allen, Angela, Arthur, Audrey

Forenklinger i forhold til læreboka

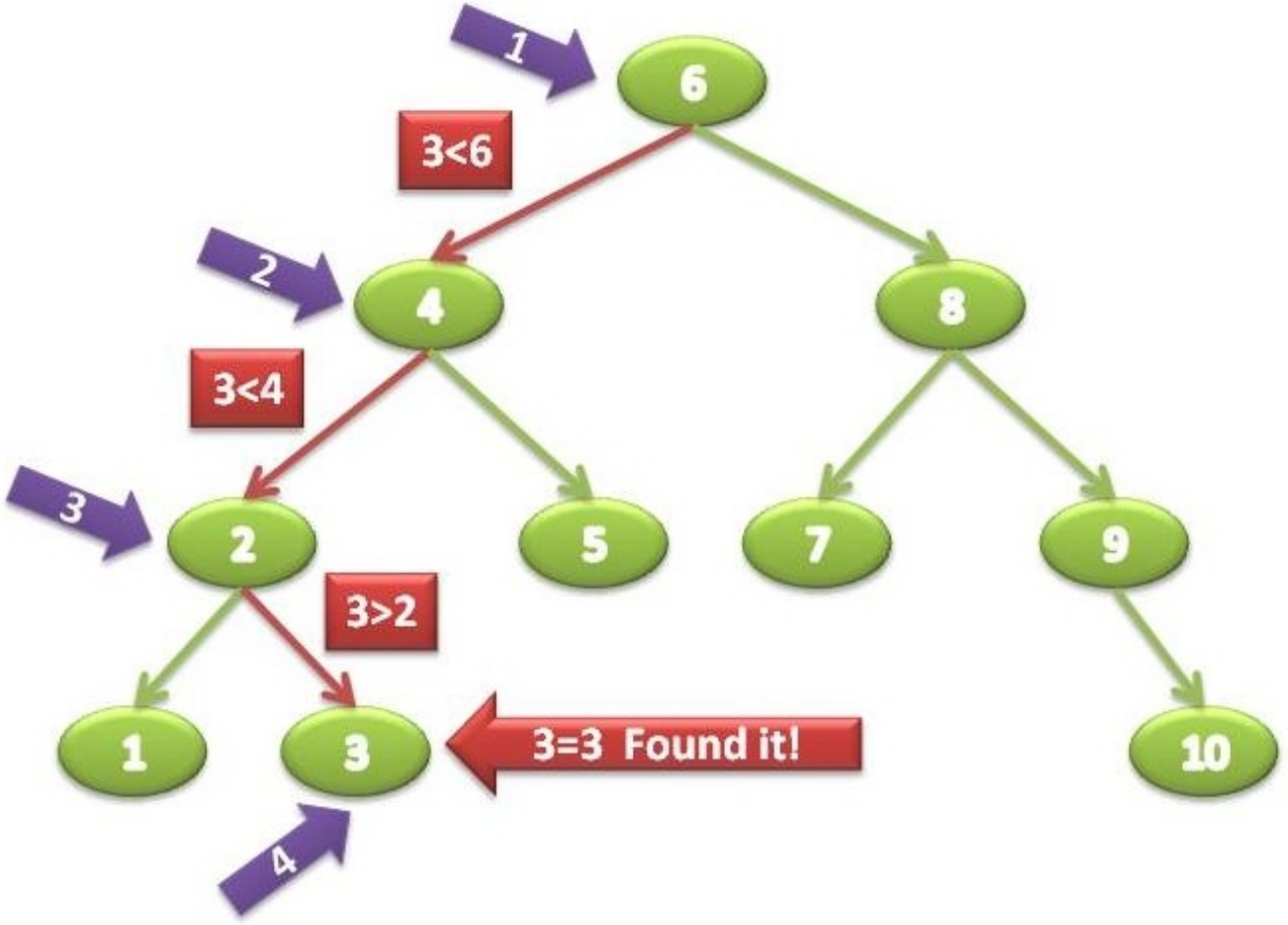
- Læreboka implementerer binære søketrær med:
  - Generisk ADT som kan lagre 'alt'
  - Både array og pekere
- På forelesning forenkler vi til:
  - Bare implementasjon med pekere
  - Ingen ADT, «skreddersyr» i stedet koden for eksemplene
  - Noder som bare inneholder enkle data og referanser til venstre og høyre barn
  - Eksempel på søketre med enkle tegn som data:

    `binarySearchTree.Java`

Søking i binært søketre

- Starter i roten av treet
- Følger en *vei* gjennom treet inntil:
  - Verdien vi søker etter er funnet, eller
  - Vi kommer til en bladnode uten å ha funnet verdien det søkes etter
- Maksimalt antall steg i søking blir lik *høyden* av treet (lengden av lengste vei fra roten frem til et blad)

Eksempel: Søk etter verdien $x = 3$



Søking i binært søketre: Implementasjon

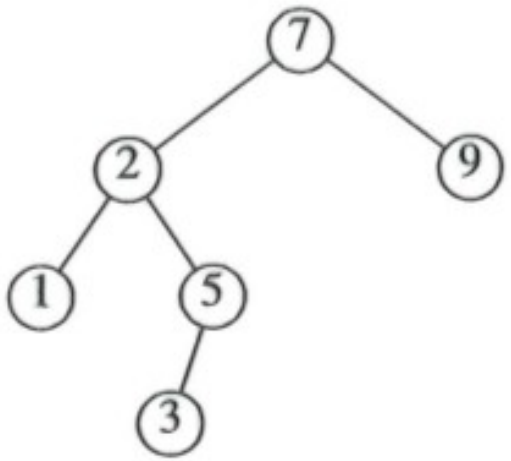
- Implementeres mest effektivt med iterasjon:
  - En while-løkke der vi i hvert steg enten går til høyre eller venstre i treet
- Enkelt eksempel:
  - Søketre med data som er enkle tegn
  - Java-kode
- Kan også kodes rekursivt:
  - Enklere(?) kode, men langsommere enn iterasjon
  - Java-kode

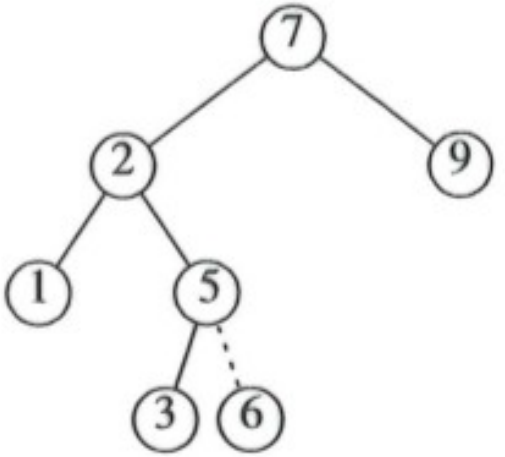
Søking i binært søketre: Effektivitet

- Verste tilfelle:
  - Verdien vi søker finnes ikke i treet
  - Søket går langs den lengste veien i treet, fra roten helt ut til et blad
  - Antall steg i worst-case blir lik høyden av treet
- Konklusjon:
  - Søking i et binært søketre med n noder er $O(\log n)$ hvis treet er *balansert*
  - Men: Søking kan bli $O(n)$ hvis treet degenerer til en skjev «nesten-liste»

Innsetting av ny verdi i binært søketre

- Starter i roten av treet
- Følger en *vei* gjennom treet inntil:
  - Vi kommer til posisjonen der verdien *kunne* ha ligget som en bladnode *hvis* den var i treet
  - Setter inn den nye noden som en bladnode på denne posisjonen:





- Animasjon av innsetting

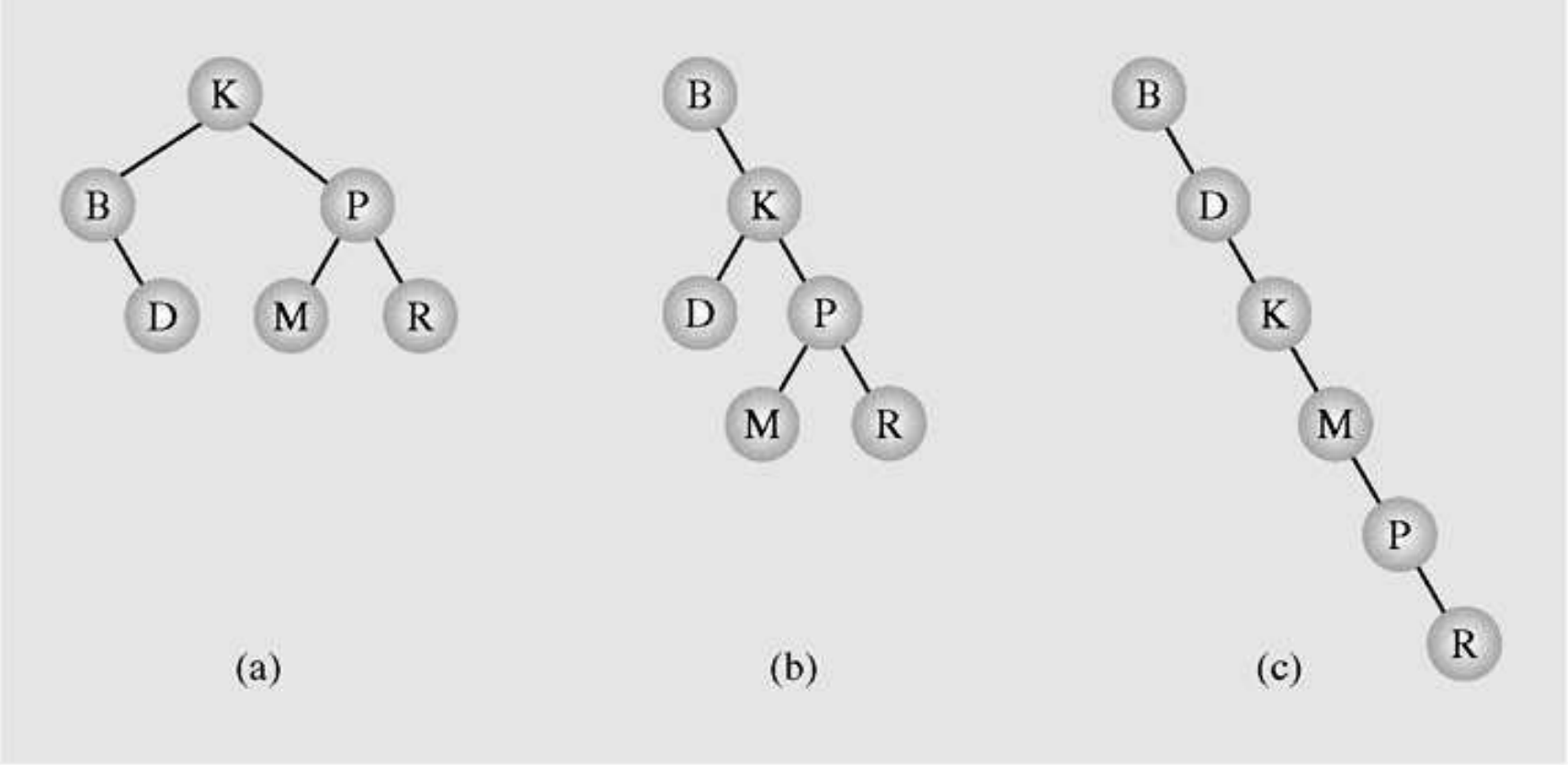
Innsetting: Implementasjon

- Iterativt:
  - Med en while-løkke der vi i hvert steg enten går til høyre eller venstre i treet
  - Setter inn ny node som et blad, som blir venstre eller høyre barn til sist oppsøkte node i treet
  - Java-kode
- Rekursivt:
  - Et rekursivt kall for hvert nivå i treet
  - Bunn i rekursjonen når neste node på søkeveien er null, da opprettes ny node og settes inn som et blad
  - Java-kode

Innsetting: Effektivitet

- Verste tilfelle:
  - Går den lengste veien som finnes i treet fra roten til et blad
  - Antall steg i worst-case er lik høyden av treet
- Konklusjon:
  - Innsetting i et søketre med n noder er $O(\log n)$ hvis treet er *balansert*
  - Innsetting blir $O(n)$ hvis treet degenererer til en «nesten-liste»

Innsetting kan gi *ubalanse*

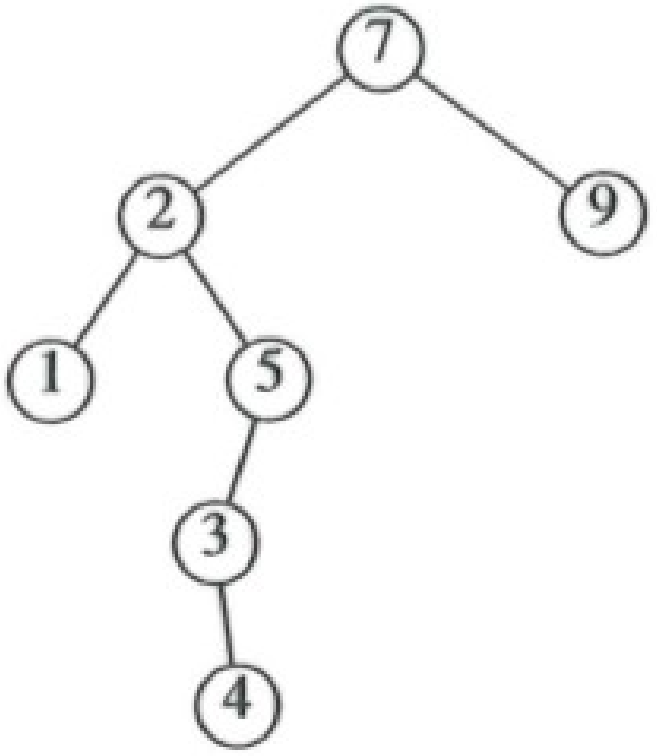


Binært søketre etter innsetting av:

(a) K B P D M R (b) B K D P M R (c) B D K M P R

Fjerning av en verdi fra binært søketre

- Starter i roten av treet
- Følger en *vei* gjennom treet inntil vi finner noden som skal fjernes
- Problem: Fjerning av f.eks. verdien 2 i figuren vil gi et «hull» i treet
- Kan hende vi må *flytte* på en annen node slik at den fyller et evt. hull som oppstår når en node skal fjernes

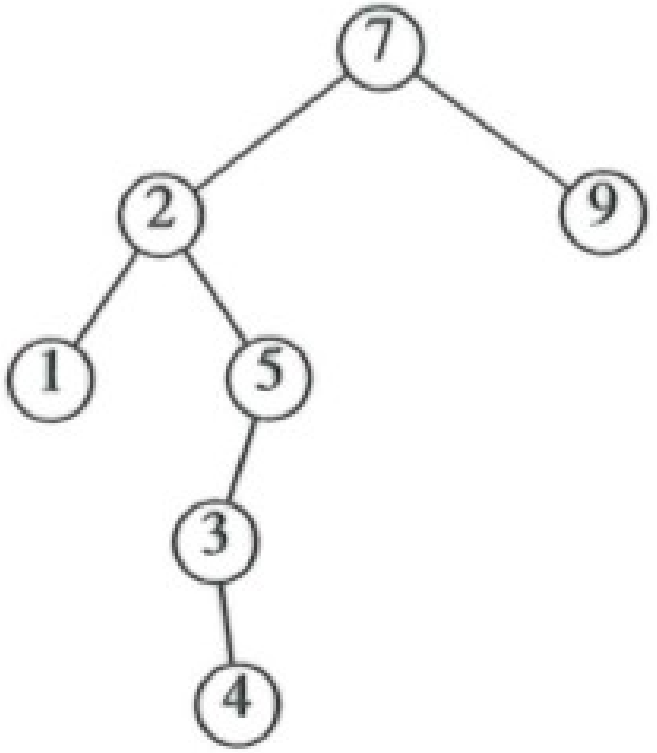


Fjerning av node i binært søketre: Må skille på tre ulike tilfeller

1. Noden som skal fjernes er en bladnode, har hverken venstre eller høyre subtre (nodene med verdier 1, 4 og 9 i figuren)

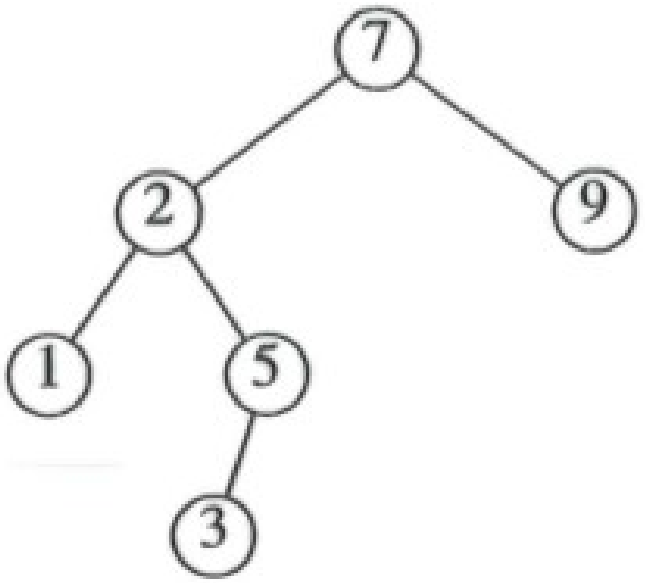
2. Noden som skal fjernes har ett subtre som er tomt og ett som ikke er tomt (3 og 5)

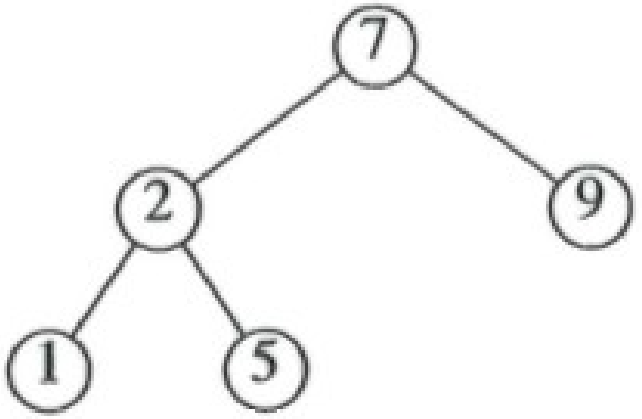
3. Noden som skal fjernes har noder i både høyre og venstre subtre (2 og 7)



Tilfelle 1: Fjerning av bladnode

- Sett forgjengernoden til å peke på null
- Fjerning av verdien 3:

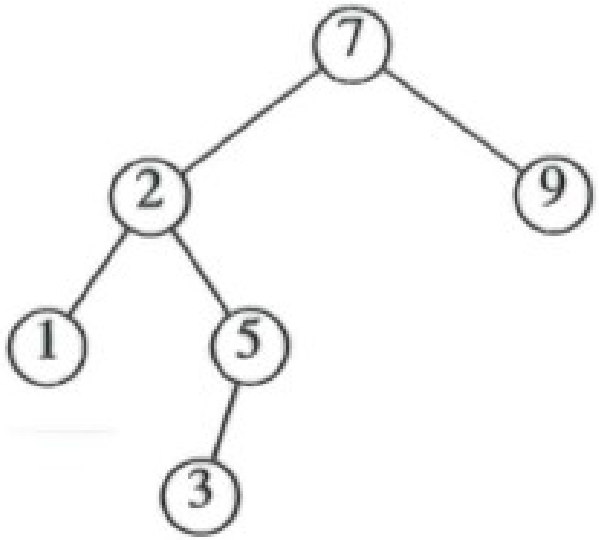


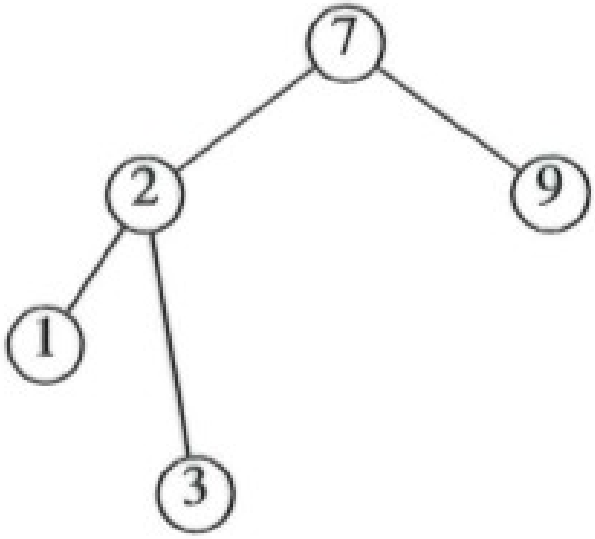


- Animasjon

Tilfelle 2:
Fjerne node med ett tomt subtre

- Sett forgjengernoden til å peke på fjernet nodes etterfølger
- Fjerning av verdien 5:

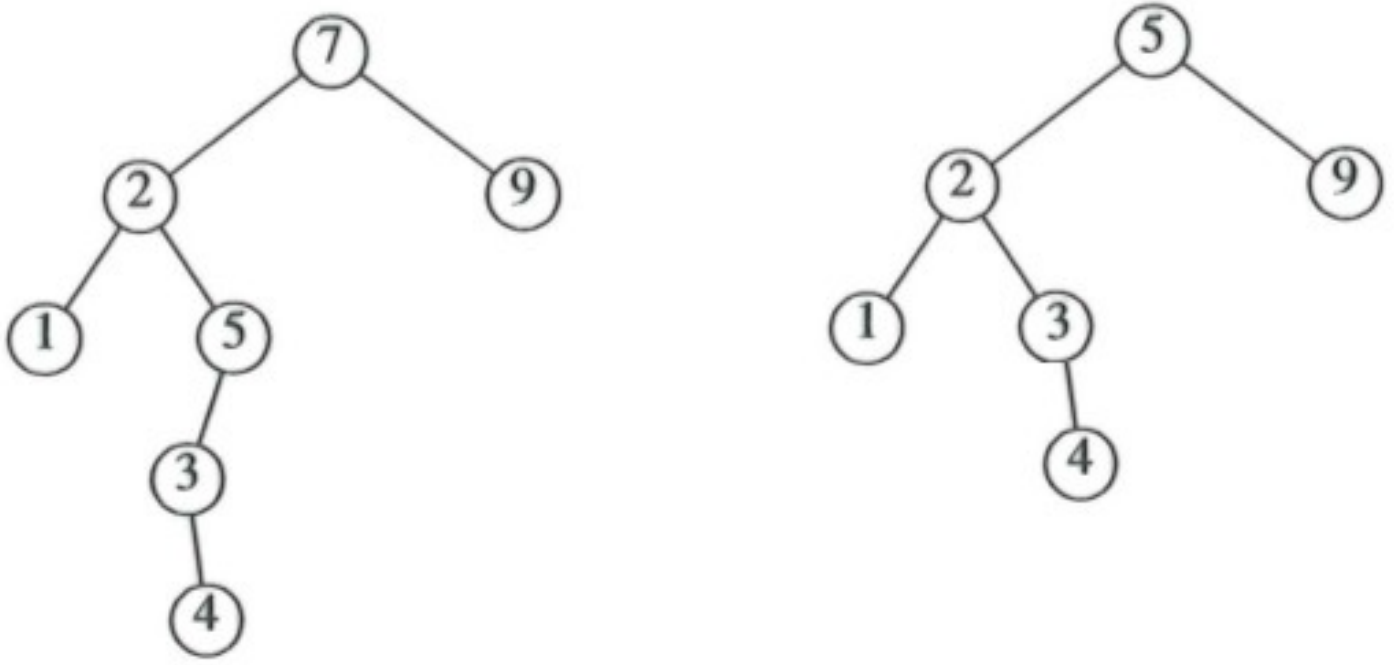




- Animasjon

Tilfelle 3: Fjerne node med to subtrær

- Finn største node i venstre subtre (lengst til høyre i subtreet), og flytt denne til posisjonen der noden som skal fjernes står
- Krever oppdateringer av pekere, flere spesialtilfeller
- Fjerning av verdien 7, noden med verdi 5 flyttes:



- Animasjon

Fjerning: Implementasjon

- Programmeres enklest med iterasjon
- Vanlig å dele opp i to metoder:
  1. Finn noden som skal fjernes, *og* dennes foreldernode, med et vanlig venstre-høyre søk i det binære treet
  2. Behandle de tre ulike tilfellene av fjerning i en egen metode, som returnerer en peker til noden (eller null) som nå står på fjernet nodes plass i treet
- Fjerningen medfører en del «pekerfikling» for å håndtere alle spesialtilfellene
- Eksempel der data er enkle tegn: Java-kode

Fjerning: Effektivitet

- Verste tilfelle:
  - Søkingen etter noden som fjernes, og deretter denne nodens evt. «erstatter», går den lengste veien fra roten til et blad i treet
- «Pekerfiklingen» *etter* at vi har funnet de to nodene avhenger ikke av antall noder i treet, er alltid $O(1)$
- Konklusjon:
  - Fjerning i et tre med n noder er $O(\log n)$ hvis treet er balansert, men kan bli $O(n)$ hvis treet degenerer
- Merk at fjerning *også* kan ødelegge balansen i treet

BST vs. sortert array/usortert lenket liste

Datastruktur	*Søking*	*Innsetting*	*Fjerning*
Lenket liste	$O(n)$	$O(1)$	$O(n)$
Sortert array	$O(\log n)$	$O(n)$	$O(n)$
Binært søketre	$O(\log n)$	$O(\log n)$	$O(\log n)$

- Testprogram som sammenligner effektivitet