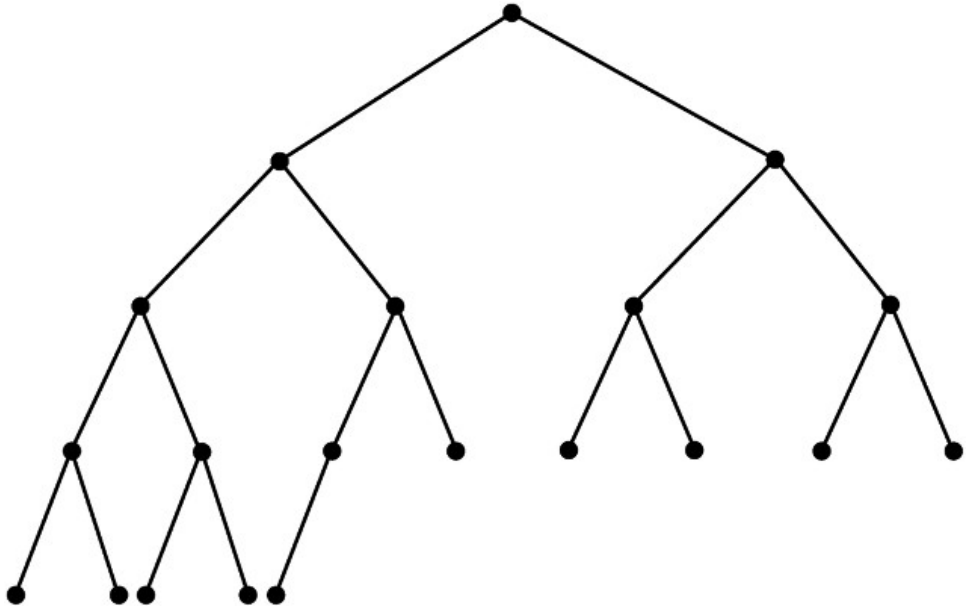
Heap og prioritetskø

Marjory the Trash Heap fra *Fraggle Rock*

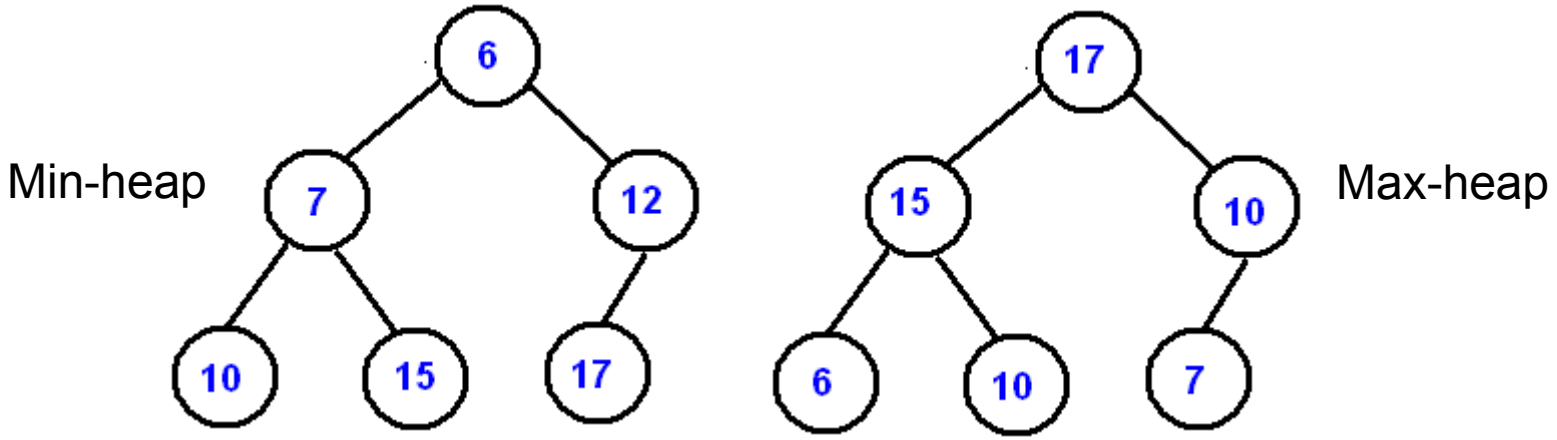
Binær heap

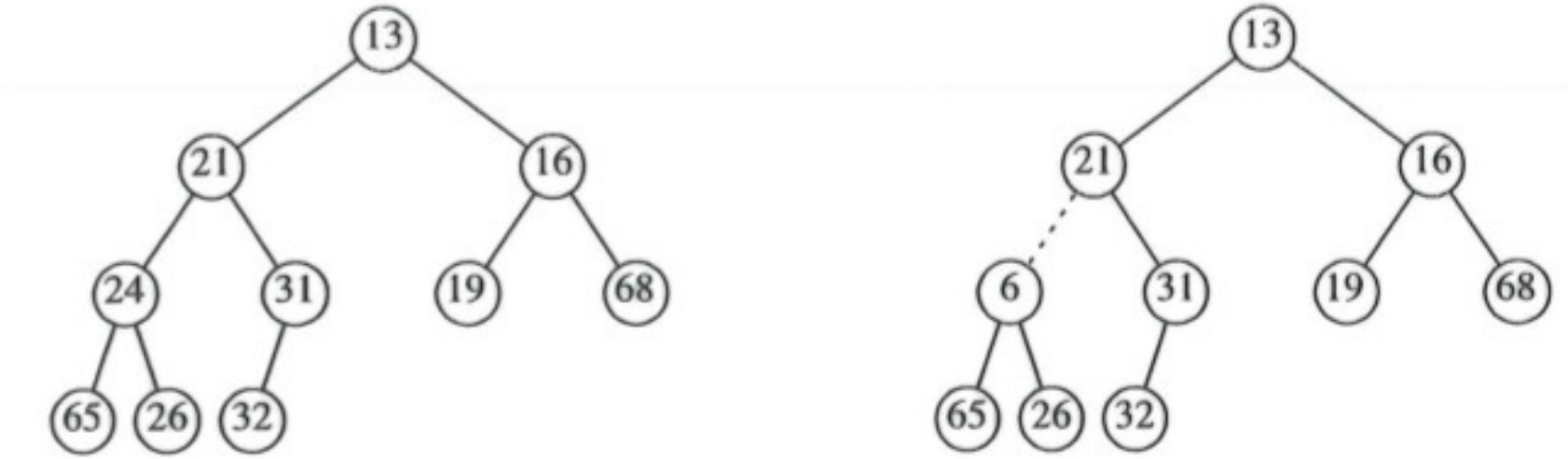
- En heap er et *komplett* binært tre:
  - Alle nivåene i treet, unntatt (muligens) det nederste, er alltid helt fylt opp med noder
  - Alle noder på nederste nivå ligger så langt til venstre som mulig



Ordningen i en heap

- Nodene i en heap er *ordnet*, men ikke sortert
- Min-heap:
  - Verdien i en node er alltid *mindre eller lik* verdien i nodens *barn*
- Max-heap:
  - Verdien i en node er alltid *større eller lik* verdien i nodens *barn*

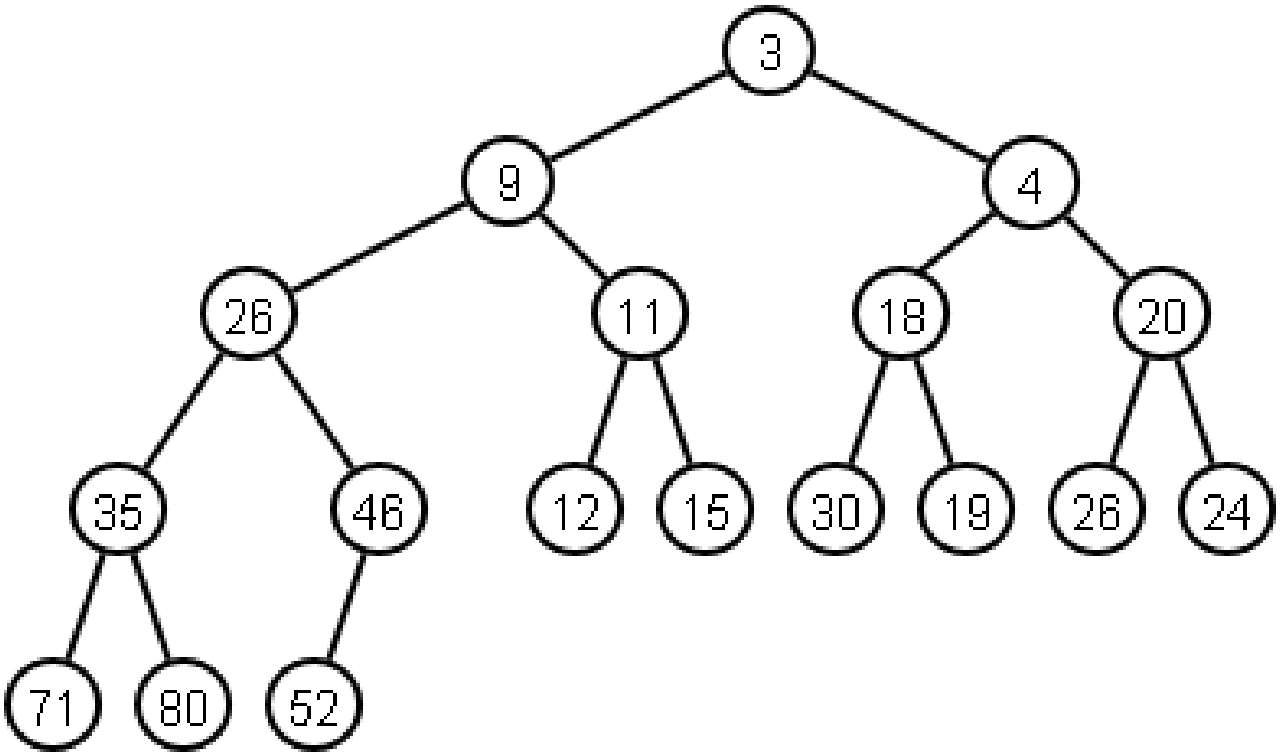




To komplette binære trær – bare treet til venstre er en heap

Heap: Monotont binært tre

- Min-heap:
  - Verdiene *øker* når vi følger en vei fra roten til et blad i treet – *monoton* stigning
  - *Minste* verdi i en min-heap er alltid i *roten* av treet
- Max-heap:
  - Verdiene *avtar* når vi følger en vei fra roten til et blad
  - *Største* verdi i en max-heap er alltid i *roten* av treet
- Alle subtrær i en heap er også en heap

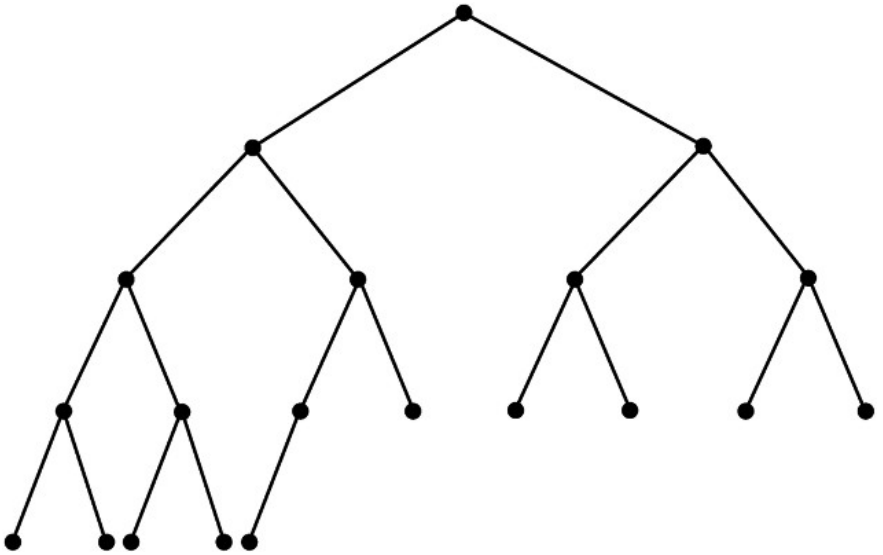


- Verdiene stiger langs alle veier fra roten
- Alle subtrær er også en heap

Høyden *h* av en heap med *n* noder

$n = 1$	$h = 0 = \log 1$
$2 \le n < 4$	$h = 1 = \log 2$
$4 \le n < 8$	$h = 2 = \log 4$
$8 \le n < 16$	$h = 3 = \log 8$
$16 \le n < 32$	$h = 4 = \log 16$ ...

Generelt: Høyden er lik største heltall mindre eller lik $\log_2 n$



Bruk av heap

- En heap tilbyr bare to operasjoner:
  - Fjern den minste (eller største) verdien i datastrukturen
  - Sett inn et nytt element i datastrukturen
  - Garanterer $O(\log n)$ effektivitet av operasjonene
- Brukes i problemer der det er viktig å ha rask tilgang til det minste (eller største) av dataelementene
- Kan også brukes til å lage en svært effektiv "in-house" sorteringsalgoritme, heapsort, som alltid er $O(n \log n)$

Arrayimplementasjon av heap

- Rotnoden lagres på indeks 0
- *Barna* til noden på indeks *i* lagres alltid på indeksene:

  $2 \cdot i + 1$ og $2 \cdot i + 2$

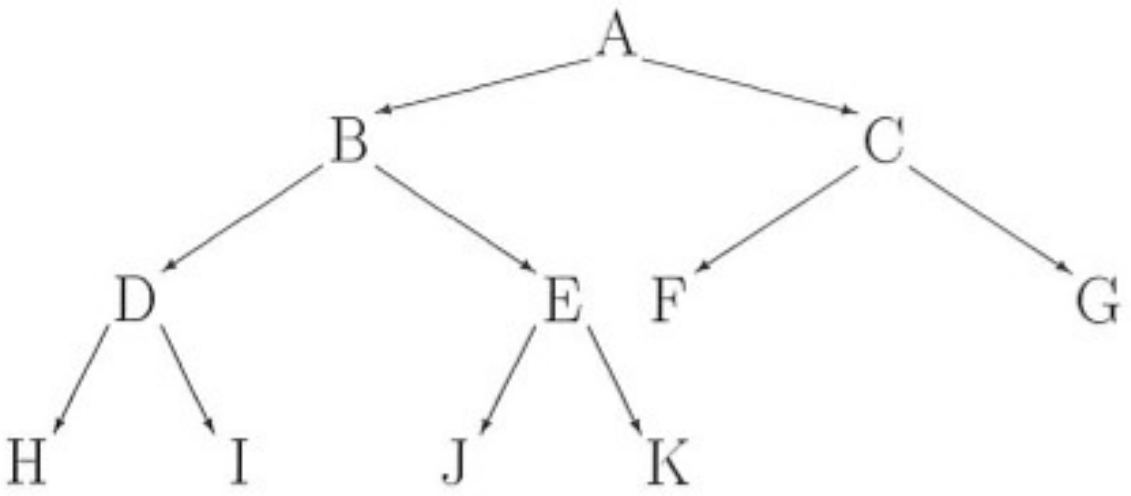
- *Forelder* til noden på indeks *i* (unntatt roten) lagres alltid på indeks:

  $(i - 1) / 2$ (heltallsdivisjon)

- Arrayimplementasjon er enkelt og svært effektivt, fordi en heap alltid er et komplett binært tre.
- Trenger bare å holde rede på antall noder i heapen

Arrayimplementasjon, eksempel

- En heap pakkes effektivt inn i en array, fra venstre mot høyre og fra roten og nedover, uten “hull” i arrayen



0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J	K

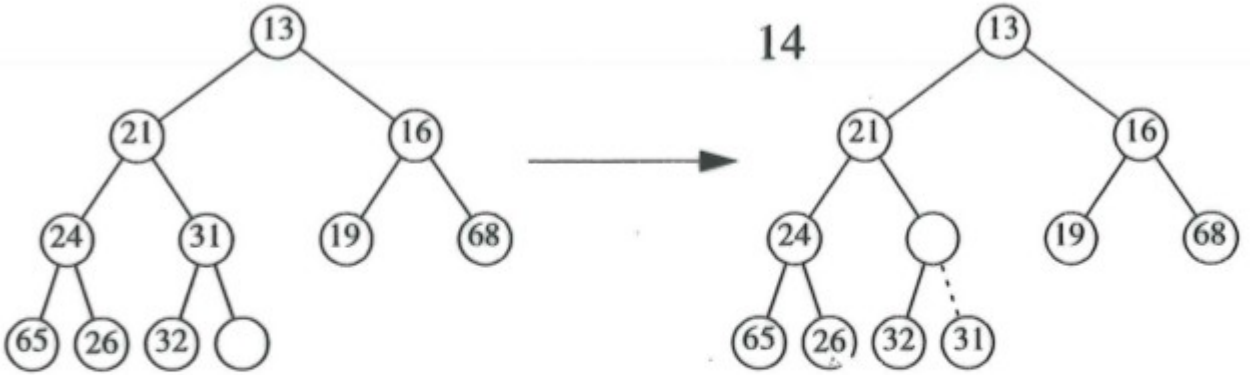
Forenklinger i forhold til læreboken

- Læreboken implementerer heap med:
  - Generisk ADT som kan lagre 'alt'
  - Både array og pekere
- For å holde fokus på prinsippene og algoritmene, og ikke på Java, forenkler vi til:
  - Bare implementasjon med array
  - Ingen ADT, bare noder som inneholder enkle data som heltall eller tegn
- Enkel min-heap med heltall: `intHeap.java`

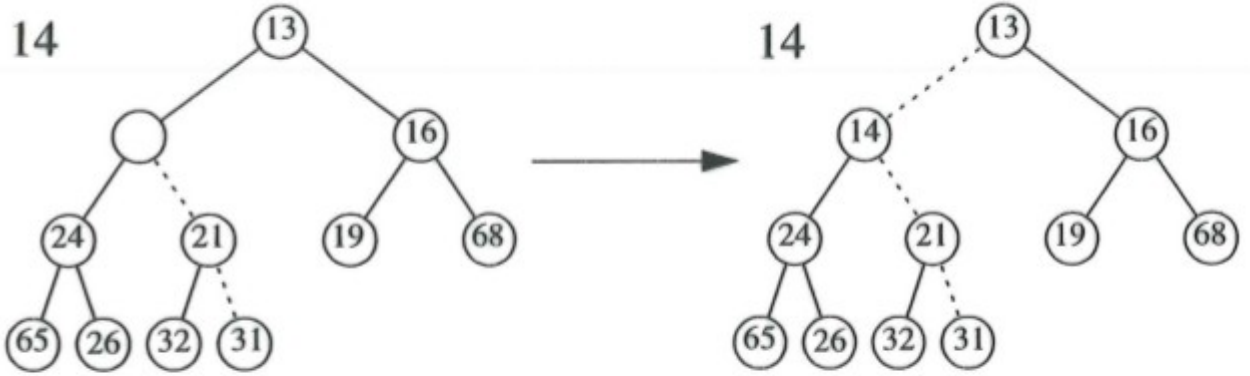
Innsetting av ny verdi i en min-heap

1. Sett inn ny node med ny verdi *sist* i heap'en, på første ledige plass i nederste nivå
2. Hvis ny verdi er *mindre enn* verdi i foreldernoden, bytt om ny node og forelder
3. Fortsett bytteprosessen *oppover* i treet inntil ny node står riktig (>= forelder eller i roten)

Den nye verdien vil “flyte” oppover i treet inntil den står på riktig plass (“percolate upwards”, “siftup”, “bubble up”)



Attempt to insert 14, creating the hole and bubbling the hole up



The remaining two steps to insert 14 in previous heap

Innsetting i min-heap lagret i array

Samme eksempel som på forrige side:

0	1	2	3	4	5	6	7	8	9	10
13	**21**	**16**	**24**	**31**	**19**	**68**	**65**	**26**	**32**	**14**
13	**21**	**16**	**24**	**14**	**19**	**68**	**65**	**26**	**32**	**31**
13	**14**	**16**	**24**	**21**	**19**	**68**	**65**	**26**	**32**	**31**
13	**14**	**16**	**24**	**21**	**19**	**68**	**65**	**26**	**32**	**31**

Demo: Animasjon av innsetting i min-heap

Innsetting: Effektivitet og implementasjon

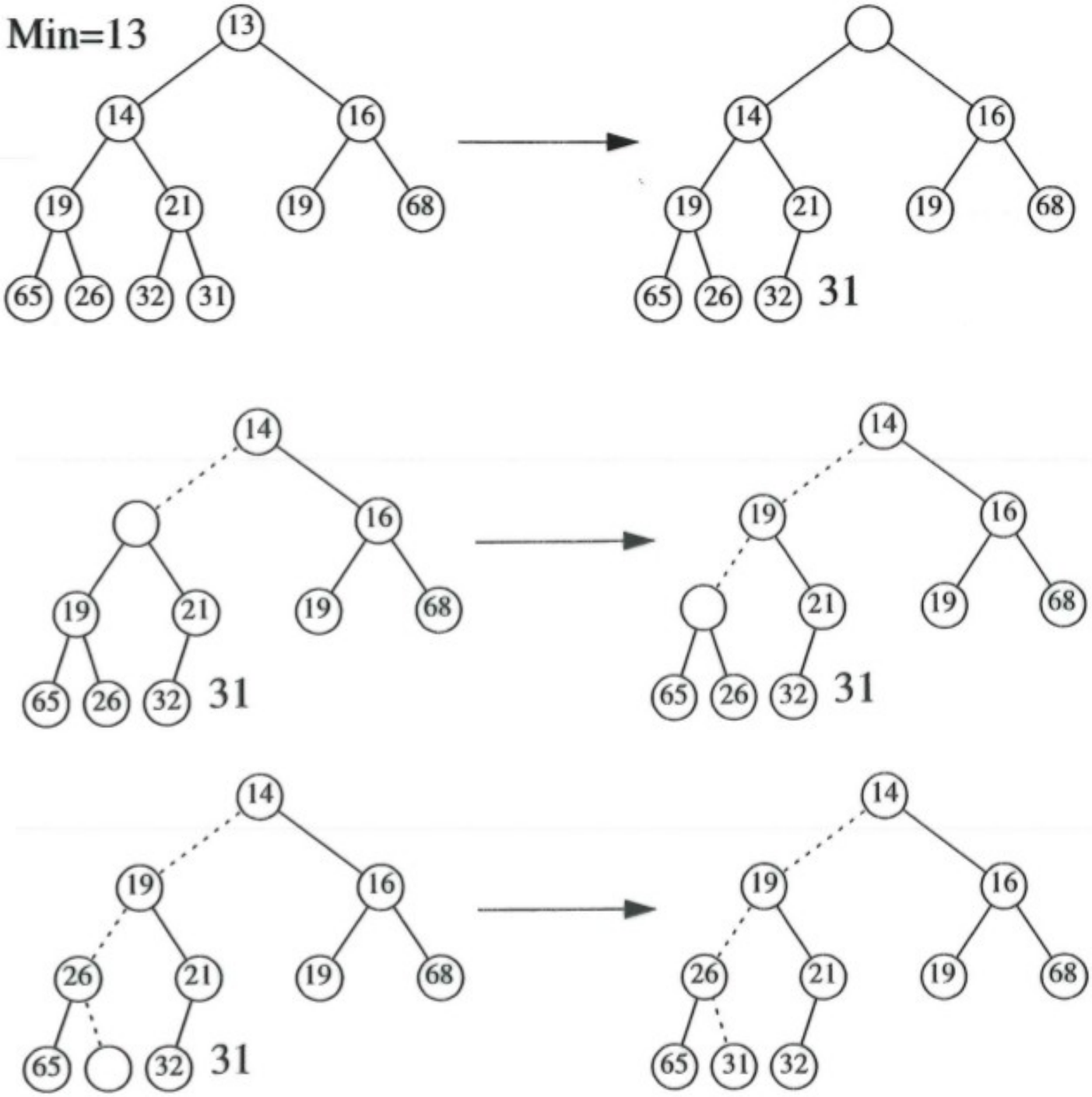
- Implementeres med en enkel while-løkke som starter fra siste posisjon i heap og bytter med foreldernode inntil ny verdi står riktig
- Finner foreldernode ved en enkel indeksberegning, raskt og effektivt
- Verste tilfelle:
  - Ny node beveger seg helt opp til roten
  - Er *alltid* $O(\log n)$ for heap med n noder
- Java-kode for min-heap med heltall: intHeap.java

Fjerning av minste verdi (roten) i min-heap

1. Ta ut rotnoden
2. Fyll igjen “hullet” ved å flytte den *siste* noden i heap'en inn på rotens posisjon
3. Hvis flyttet node er større enn et av barna:

   Bytt om det *minste* av barna med flyttet node
4. Fortsett å bytte flyttet node nedover i treet inntil den står riktig (<= begge barn eller blad)

Noden som flyttes opp til roten vil “flyte” nedover i treet inntil den står riktig (“percolate downwards”, “siftdown”)


Min=13
13
14
16
19
21
19
68
65
26
32
31
14
16
19
21
19
68
65
26
32
31
14
16
19
21
19
68
65
26
32
31
14
19
16
21
19
68
65
26
32
31
14
19
16
26
21
19
68
65
32
31
14
19
16
26
21
19
68
65
31
32

Fjerning av minste verdi i min-heap

Samme eksempel som på forrige side, med array :

0	1	2	3	4	5	6	7	8	9	10
13	14	16	19	21	19	68	65	26	32	31
31	14	16	19	21	19	68	65	26	32	
14	31	16	19	21	19	68	65	26	32	
14	19	16	31	21	19	68	65	26	32	
14	19	16	26	21	19	68	65	31	32	
14	19	16	26	21	19	68	65	31	32	

Demo: Animasjon av fjerning i min-heap

Fjerning av minste verdi: Effektivitet og implementasjon

- Implementeres med en enkel while-løkke som starter med “ny rot” og bytter med minste barn inntil verdi står riktig
- Finner barnenodene ved enkel indeksberegning, raskt og effektivt
- Verste tilfelle:
  - Ny rotnode beveger seg helt ned til et blad
  - Er *alltid* $O(\log n)$ for heap med n noder
- Java-kode for min-heap med heltall: intHeap.java

Prioritetskø

- En kø som *ikke* er First-In-First-Out
- Alle elementene i køen har en verdi som angir en *prioritet*
- Det er alltid elementet med *størst* (evt. minst) prioritet som *først* skal tas ut av køen
- `deQueue`:
  - Ta ut elementet med størst/minst prioritet
- `enQueue`:
  - Sett inn et element med en gitt prioritet

Implementasjon av prioritetskø

- Med sortert array:
  - Fjerning av elementet i køen med størst prioritet, `deQueue`, blir $O(1)$
  - Innsetting av nytt element med gitt prioritet, `enQueue`, blir $O(n)$, for langsomt for store datasett
- Med binært søketre / AVL-tre:
  - Både `deQueue` og `enQueue` blir $O(\log n)$
  - Unødvendig komplisert for en prioritetskø, trenger ikke å ha en full sortering av elementene i køen
- Beste alternativ: Heap

Prioritetskø og heap

- En heap *implementerer* en prioritetskø:

```
insert    = enQueue
removeMin = deQueue
```

- Innsetting og fjerning blir begge $O(\log n)$
- Java-implementasjon: Se koden fra læreboka

Anvendelser av prioritetskøer

- I operativsystemer:
  - Scheduling av prosesser
  - Køer til delte ressurser som f.eks. printere
- Simulering av køsystemer:
  - Objektene som skal håndteres står i en prioritetskø
  - Eksempel: Flyplass-simulering der flyene som skal lande prioriteres etter gjenværende mengde drivstoff
- Handlingsdrevet simulering

Handlingsdrevet simulering

- Discrete event simulation / Event-driven simulation
- Handlingene (events) som skal skje i systemet ligger i en prioritetskø, ordnet på *tidspunkt* for handlingen
- Simuleringen drives fremover ved at vi i hvert steg tar ut og utfører handlingen som skal utføres *først* (har lavest "tidsstempel")
- Utførelse av en handling kan føre til at *nye* handlinger legges inn i prioritetskøen