

INF2220 - Algoritmer og datastrukturer

HØSTEN 2008

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 5:
Prioritetskø og Heap

Prioritetskø

- ▶ Kø implementasjoner vi kjenner
 - ▶ FIFO - Liste
 - ▶ LIFO - Stack

Prioritetskø

- ▶ Kø implementasjoner vi kjenner
 - ▶ FIFO - Liste
 - ▶ LIFO - Stack
- ▶ Vi ønsker ofte bedre kontroll over elementene i køen

Prioritetskø

- ▶ Kø implementasjoner vi kjenner
 - ▶ FIFO - Liste
 - ▶ LIFO - Stack
- ▶ Vi ønsker ofte bedre kontroll over elementene i køen
- ▶ Eksempel: Job scheduler (OS)
 - ▶ Jobber kan ikke kjøre ferdig før neste slipper til
 - ▶ Jobber tas typisk ut og settes inn igjen
 - ▶ Round-Robin kan bli urettferdig
 - ▶ Prioritet kan gjøre fordeling rettferdig

Prioritetskø - grensesnitt

- ▶ Prioritet er gitt ved heltall (lavt tall = høy prioritet)
- ▶ Vi kan se på prioritet som tid vi maksimalt kan vente

`insert(p, x)` sett inn element **x** med prioritet **p**

`deleteMin()` fjern element med høyest prioritet

Prioritetskø - grensesnitt

- ▶ Prioritet er gitt ved heltall (lavt tall = høy prioritet)
- ▶ Vi kan se på prioritet som tid vi maksimalt kan vente

`insert(p, x)` sett inn element x med prioritet p

`deleteMin()` fjern element med høyest prioritet

- ▶ Forskjellige datastrukturer kan implementere et slikt grensesnitt
 - ▶ Liste (sortert eller uordnet)
 - ▶ Søketre
 - ▶ Heap

Prioritetskø - Datastruktur og Kompleksitet

► Uordnet Liste

- ▶ Kompleksitet (worst case)
- ▶ Innsetting først i listen $\mathcal{O}(1)$
- ▶ Sletting kan kreve gjennomgang av hele listen $\mathcal{O}(n)$

Prioritetskø - Datastruktur og Kompleksitet

► Uordnet Liste

- ▶ Kompleksitet (worst case)
- ▶ Innsetting først i listen $\mathcal{O}(1)$
- ▶ Sletting kan kreve gjennomgang av hele listen $\mathcal{O}(n)$

► Sortert Liste

- ▶ Kompleksitet (worst case)
- ▶ Innsetting kan kreve gjennomgang av hele listen $\mathcal{O}(n)$
- ▶ Sletting fjerner første element $\mathcal{O}(1)$

Prioritetskø - Datastruktur og Kompleksitet

► Uordnet Liste

- ▶ Kompleksitet (worst case)
- ▶ Innsetting først i listen $\mathcal{O}(1)$
- ▶ Sletting kan kreve gjennomgang av hele listen $\mathcal{O}(n)$

► Sortert Liste

- ▶ Kompleksitet (worst case)
- ▶ Innsetting kan kreve gjennomgang av hele listen $\mathcal{O}(n)$
- ▶ Sletting fjerner første element $\mathcal{O}(1)$

► Søketrær

- ▶ Innsetting worst case $\mathcal{O}(n)$ average case $\mathcal{O}(\log_2(n))$
- ▶ Sletting worst case $\mathcal{O}(n)$ average case $\mathcal{O}(\log_2(n))$

Prioritetskø - Heap

- ▶ Heap er den vanligste implementasjonen av en prioritetskø
- ▶ Vi skal se på en implementasjon som kalles binær heap

Prioritetskø - Heap

- ▶ Heap er den vanligste implementasjonen av en prioritetskø
- ▶ Vi skal se på en implementasjon som kalles binær heap
- ▶ En binær heap er et binærtre med et **strukturkrav**
 - ▶ *En binær heap er et komplett binærtre*

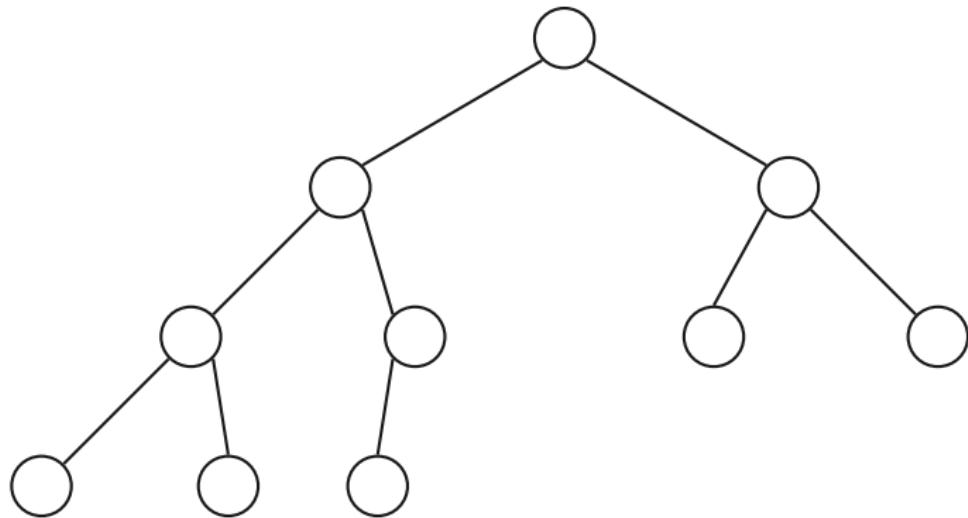
Prioritetskø - Heap

- ▶ Heap er den vanligste implementasjonen av en prioritetskø
- ▶ Vi skal se på en implementasjon som kalles binær heap
- ▶ En binær heap er et binærtre med et **strukturkrav**
 - ▶ *En binær heap er et komplett binærtre*
- ▶ Og et **ordningskrav**
 - ▶ *Barn er alltid større eller lik sine foreldre*

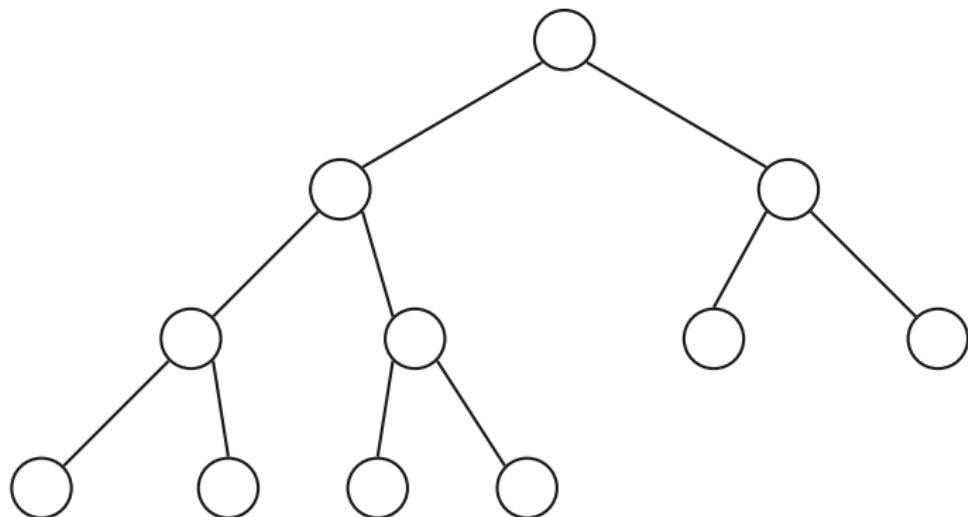
Prioritetskø - Heap

- ▶ Heap er den vanligste implementasjonen av en prioritetskø
- ▶ Vi skal se på en implementasjon som kalles binær heap
- ▶ En binær heap er et binærtre med et **strukturkrav**
 - ▶ *En binær heap er et komplett binærtre*
- ▶ Og et **ordningskrav**
 - ▶ *Barn er alltid større eller lik sine foreldre*
- ▶ Ordet **Heap** blir også brukt om dynamisk allokerert minne

Binær Heap - Strukturkrav - Komplett Binætre



Binær Heap - Strukturkrav - Komplett Binætre

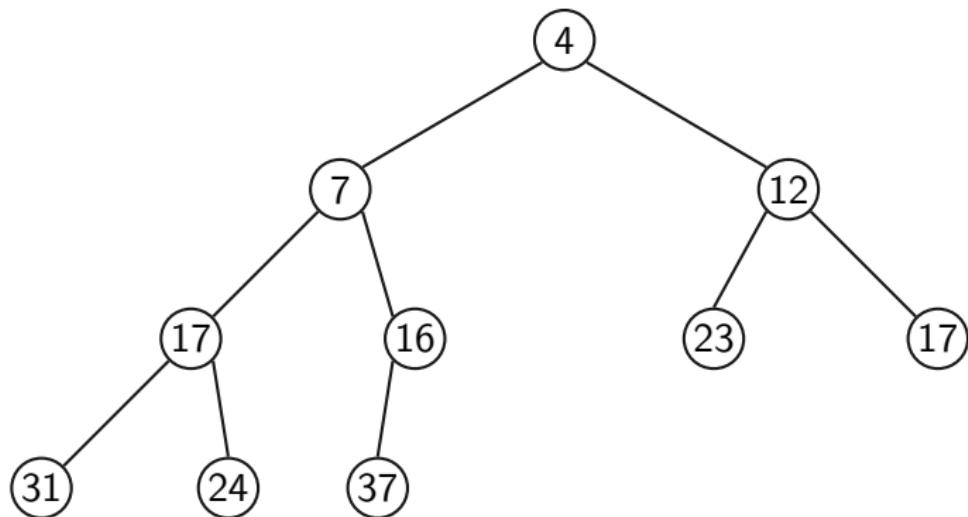


Binær Heap - Strukturkrav

Ett **komplett** binærtre har følgende egenskaper

- ▶ Treet vil være i perfekt balanse
- ▶ Bladnoder vil ha høydeforskjell på maksimalt 1
- ▶ Den maksimale høyden på treet vil være $\log_2(n)$

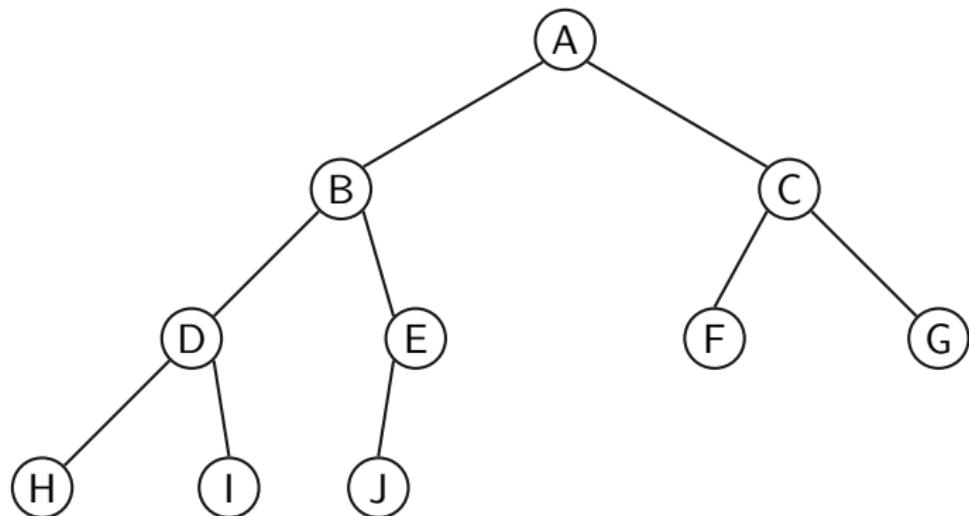
Binær Heap - Ordningskrav



Binær Heap - Representasjon

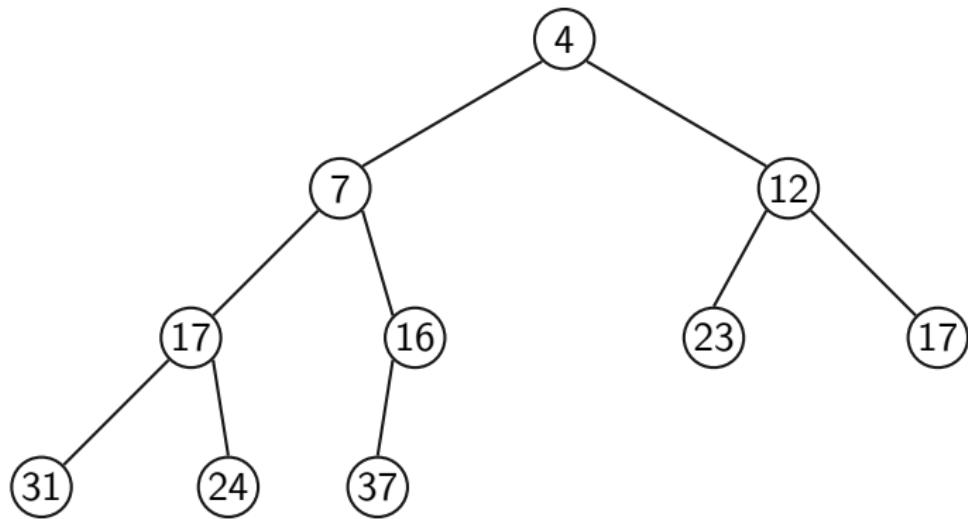
- ▶ Binærtreet er **komplett** så vi kan legge elementene i en **array**
- ▶ Vi kan enkelt finne foreldre og barn ut i fra array **index**
 - ▶ Venstre barn: **index × 2**
 - ▶ Høyre barn: **index × 2 +1**
 - ▶ Foreldre: **(int) index/2**
- ▶ Vi kan risikere å måtte allokkere ny array og kopiere alle elementene

Binær Heap - Representasjon

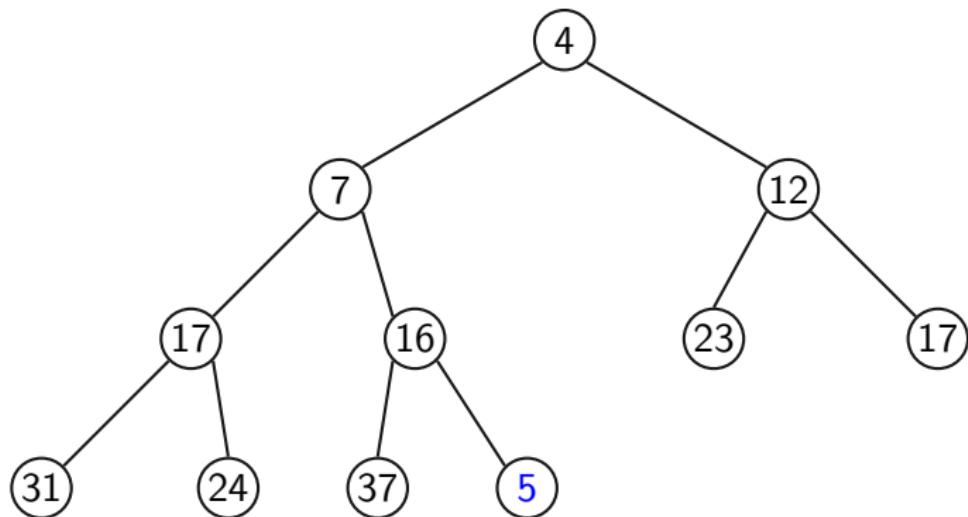


	A	B	C	D	E	F	G	H	I	J					
--	---	---	---	---	---	---	---	---	---	---	--	--	--	--	--

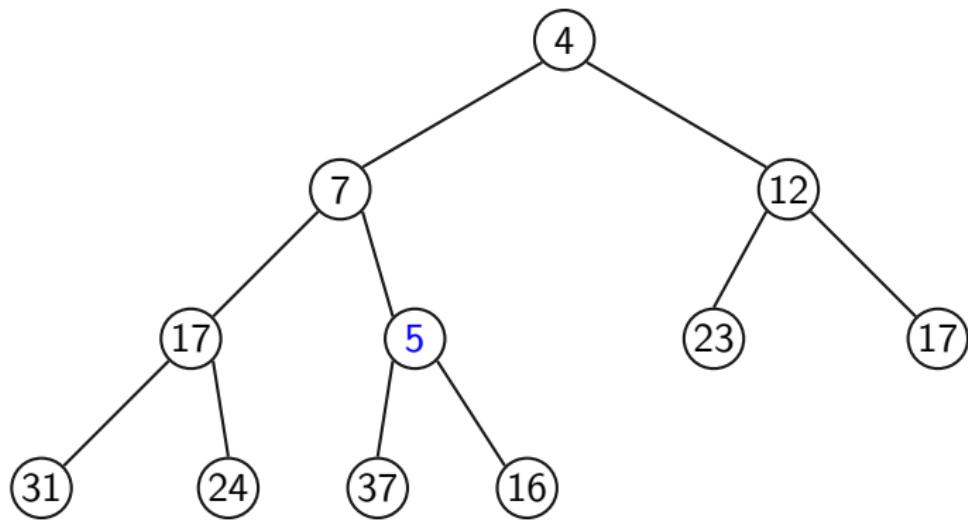
Binær Heap - insert



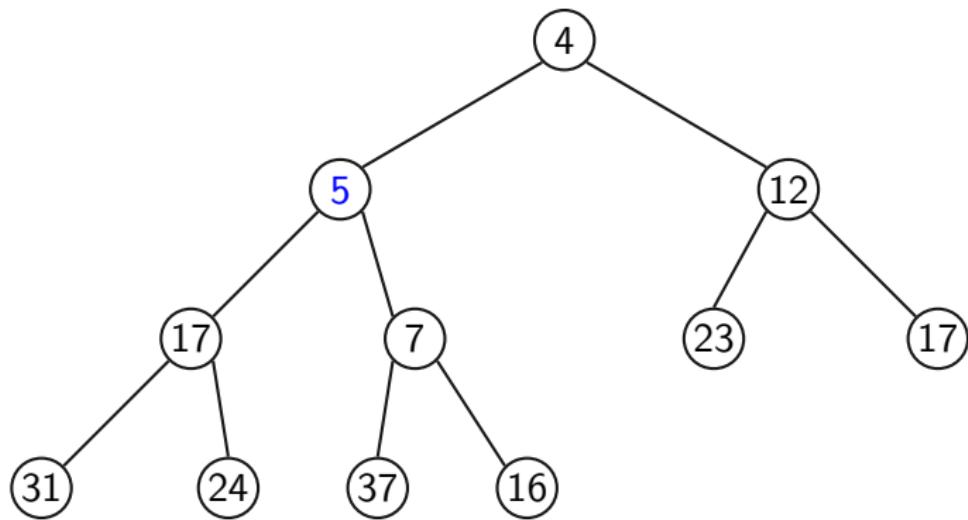
Binær Heap - insert



Binær Heap - insert



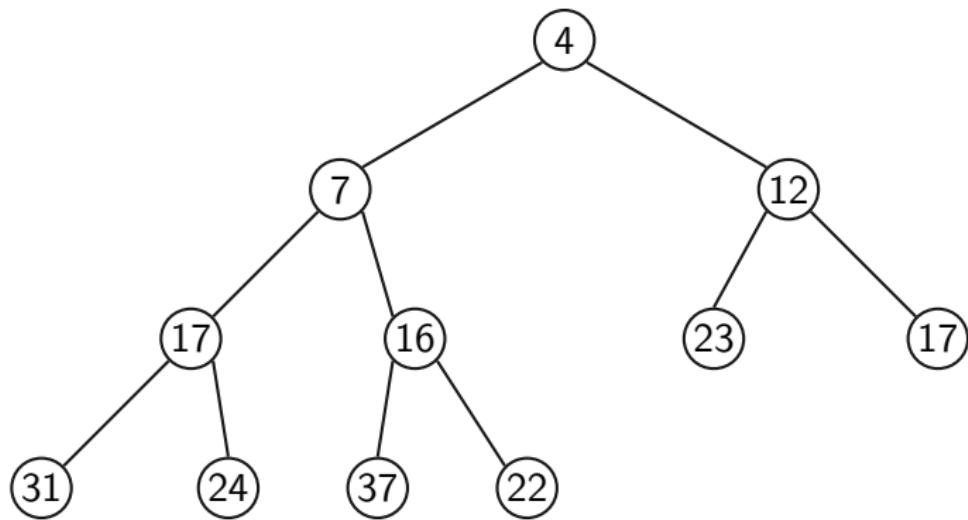
Binær Heap - insert



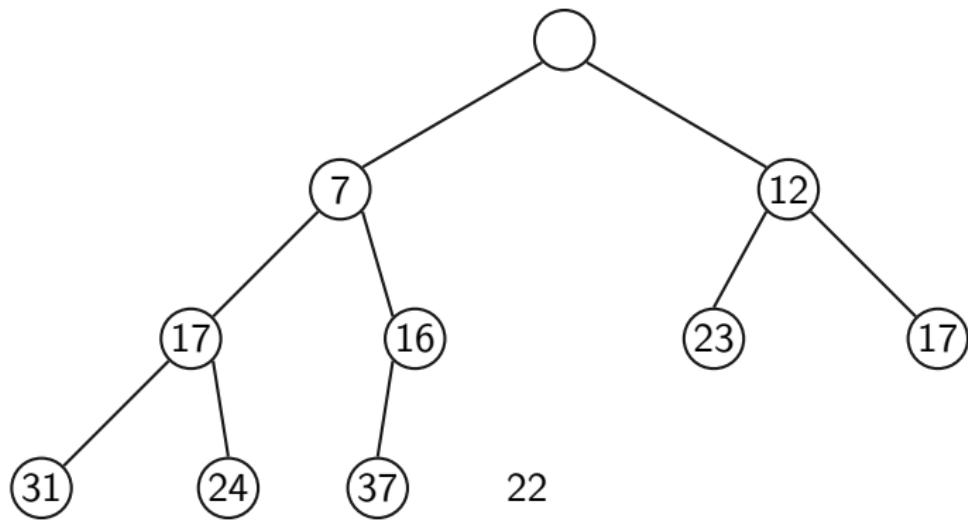
Binær Heap - insert

- ▶ Legg det nye elementet på neste ledige plass i heapen
- ▶ La det nye elementet **flyte** opp til riktig posisjon
- ▶ Dette kalles **up-heap bubbling** i læreboka
 - ▶ Kalles ofte **percolate up** i annen litteratur
- ▶ Siden treet er i balanse kan vi maksimalt flyte $\mathcal{O}(\log_2(n))$

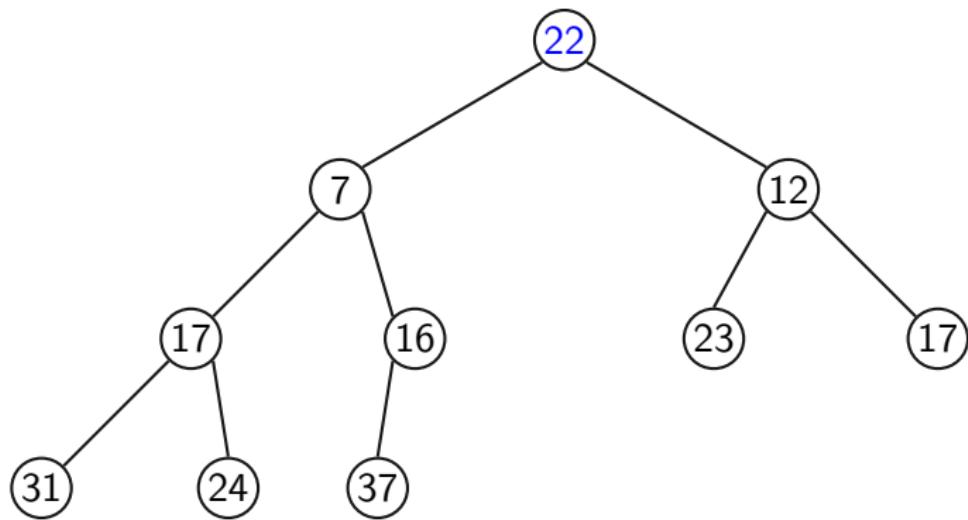
Binær Heap - deleteMin



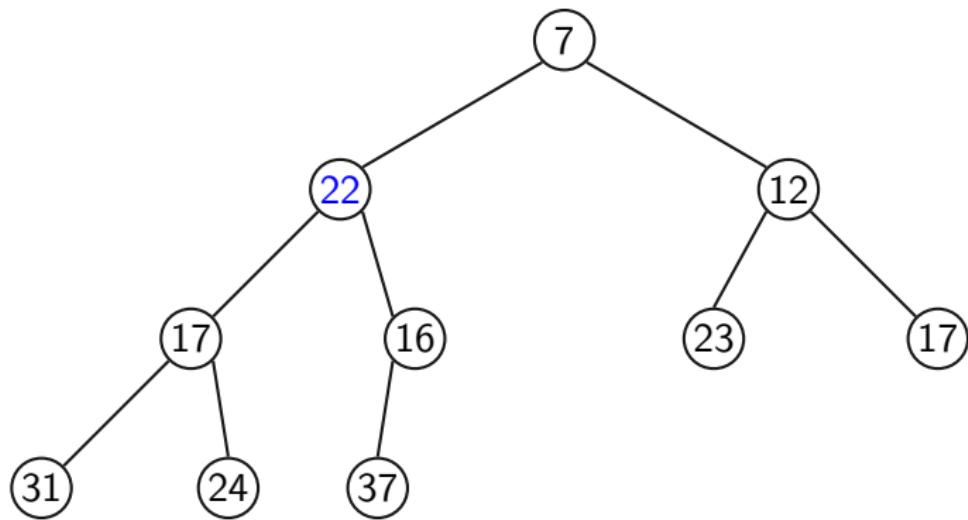
Binær Heap - deleteMin



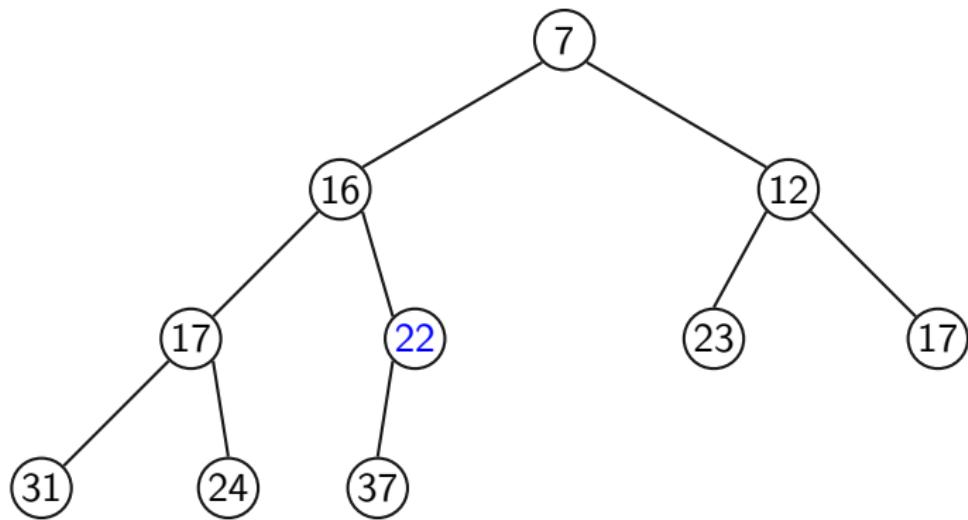
Binær Heap - deleteMin



Binær Heap - deleteMin



Binær Heap - deleteMin



Binær Heap - deleteMin

- ▶ Vi fjerner rot elementet fra heapen
- ▶ Vi lar det siste elementet bli ny rot
- ▶ Vi lar den nye rota **flyte** ned til riktig posisjon
- ▶ Dette kalles **down-heap bubbling** i læreboka
 - ▶ Kalles ofte **percolate down** i annen litteratur
- ▶ Siden treet er i balanse kan vi maksimalt flyte $\mathcal{O}(\log_2(n))$

Binær Heap - Andre Operasjoner

- ▶ `findMin` kan gjøres i konstant tid
- ▶ `delete` fjern vilkårlig element fra heapen
- ▶ Vi kan også endre prioritet på elementer i heap
 - ▶ Senking av prioritet kalles ofte `decreaseKey`
 - ▶ Øking av prioritet kalles ofte `increaseKey`
- ▶ Både `increaseKey` og `decreaseKey` gjøres typisk ved å:
 - ▶ Lokalisere element i heapen
 - ▶ Øk eller senk prioritet
 - ▶ La elementet *flyte* opp eller ned avhengig av operasjon
- ▶ `delete` kan typisk gjøres ved `decreaseKey ∞ + deleteMin`

Binær Heap - Sortering

- ▶ Vi kan bruke en binær heap til å sortere
- ▶ Vi kan bygge en binær heap (`insert`) på $\mathcal{O}(n \cdot \log_2(n))$
- ▶ Vi kan ta ut alle elementene (`deleteMin`) på $\mathcal{O}(n \cdot \log_2(n))$
- ▶ $2 \cdot \mathcal{O}(n \cdot \log_2(n)) = \mathcal{O}(n \cdot \log_2(n))$ (**worst case**)
- ▶ Oppbygging av heap kan også effektiviseres
 - ▶ Legg alle elementene på heapen (inn i arrayen)
 - ▶ `foreach` node som ikke er bladnode:
 - ▶ `percolateDown(node)`