

# INF2220 - Algoritmer og datastrukturer

HØSTEN 2015

Ingrid Chieh Yu  
Institutt for informatikk, Universitetet i Oslo

Forelesning 4:  
**Prioritetskø og Heap**

- ▶ M.A.W. Data Structures and Algorithm Analysis kap. 6
- ▶ 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

Forskjellige datastrukturer kan implementere et slikt grensesnitt

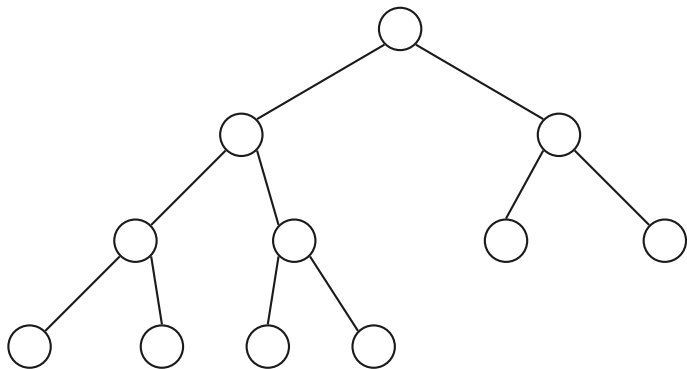
- ▶ Liste (sortert eller uordnet)
- ▶ Søketre
- ▶ Heap

	insetting	sletting
uordnet liste	$\mathcal{O}(1)$	$\mathcal{O}(n)$
sortert liste	$\mathcal{O}(n)$	$\mathcal{O}(1)$
søketrær (worst/av. case)	$\mathcal{O}(n)/\mathcal{O}(\log_2(n))$	

# Prioritetskø - Heap

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

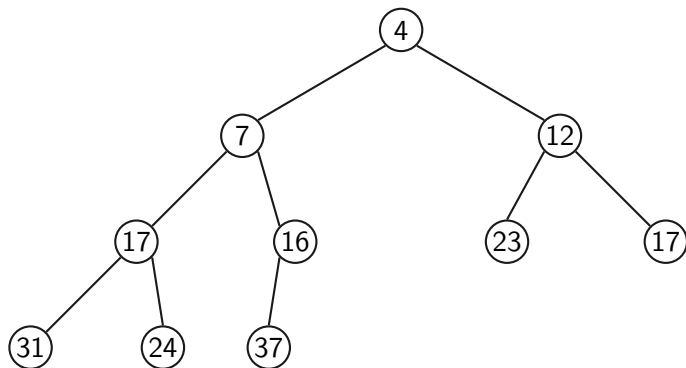
# Binær Heap - Strukturkrav - Komplette Binære



Ett **komplett** binærtre har følgende egenskaper

- ▶ Treet vil være i perfekt balanse
- ▶ Bladnoder vil ha høydeforskjell på maksimalt 1
- ▶ Treet med høyden  $h$  har mellom  $2^h$  og  $2^{h+1} - 1$  noder
- ▶ Den maksimale høyden på treet vil være  $\log_2(n)$

# Binær Heap - Ordningskrav

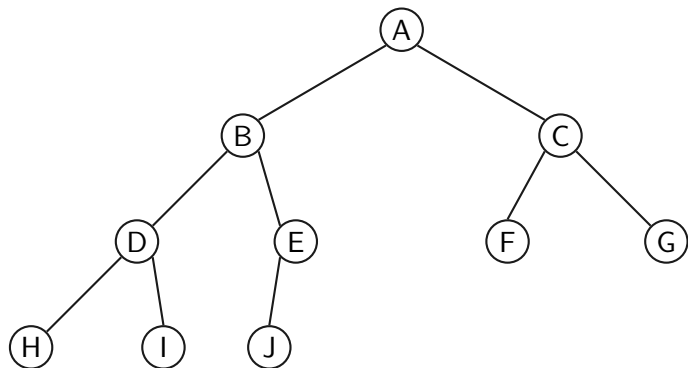




# Binær Heap - Representasjon

- ▶ 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**  $\times$  **2**
  - ▶ Høyre barn: **index**  $\times$  **2** **+1**
  - ▶ Foreldre: **(int)** **index/2**
- ▶ Vi kan risikere å måtte allokere ny array og kopiere alle elementene

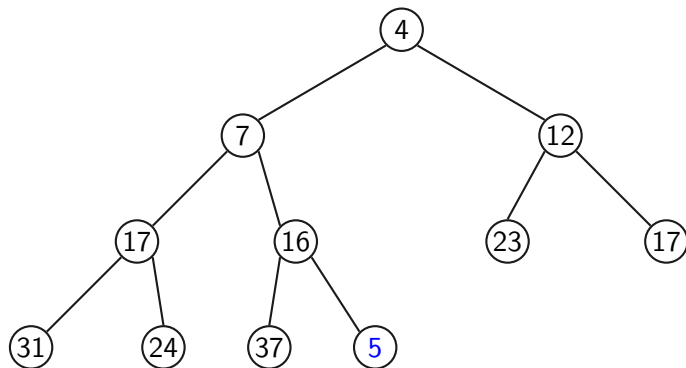
# Binær Heap - Representasjon



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

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

# Binær Heap - insert



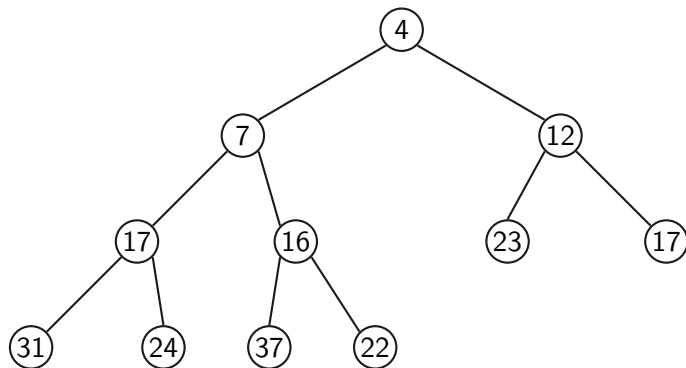
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



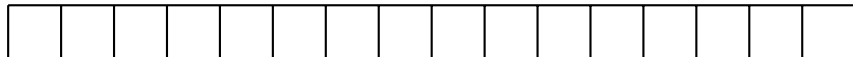
## Binær Heap - insert

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

# Binær Heap - deleteMin



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

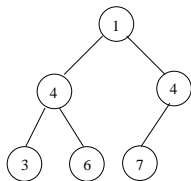


- ▶ Vi fjerner rot elementet fra heapen
- ▶ Vi lar det siste elementet bli ny rot
- ▶ Vi lar den nye rota **flyte** ned til riktig posisjon
- ▶ Dette kalles **percolate down** i læreboka
- ▶ Siden treet er i balanse kan vi maksimalt flyte  $\mathcal{O}(\log_2(n))$

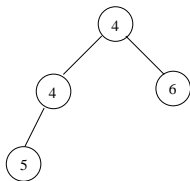
# Binær Heap - Andre Operasjoner

- ▶ `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 `increaseKey`
  - ▶ Øking av prioritet kalles ofte `decreaseKey`
- ▶ 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`

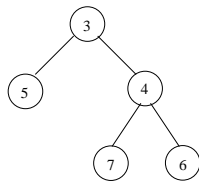
Hvilken av følgende BT er ikke en heap?:



(A)



(B)



(C)

I: A, B og C

II: B og C

III: A og C



**Basert på prioritetskø, hvor fort kan vi sortere  $n$  tall ?:**

I:  $\mathcal{O}(n)$

II:  $\mathcal{O}(n^2)$

III:  $\mathcal{O}(n \log n)$

IV:  $\mathcal{O}(\log n)$

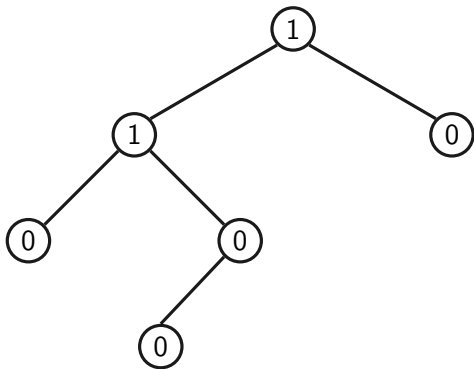
# Binær Heap - Sortering

- ▶ 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)`

# Venstreorientert Heap

- ▶ En **venstreorientert heap** er en prioritetskø implementert som en variant av binær heap
- ▶ **Ordningskrav**: samme som ordningskravet til binær heap
- ▶ **Strukturkrav**:
  - ▶ La **null path length**  $npl(x)$  være *lengden av den korteste veien* fra  $x$  til en node uten to barn.
  - ▶  $npl(l) \geq npl(r)$  hvor  $l$  og  $r$  er venstre og høyre barnet til  $x$
  - ▶ forsøker å være ubalansert!
- ▶ Å flette to binære heaper (**merge**) tar  $\Theta(N)$  for heaper med like størrelser
- ▶ **Venstreorientert Heap** støtter merge i  $\mathcal{O}(\log n)$

# Venstreorientert Heap - Strukturkrav

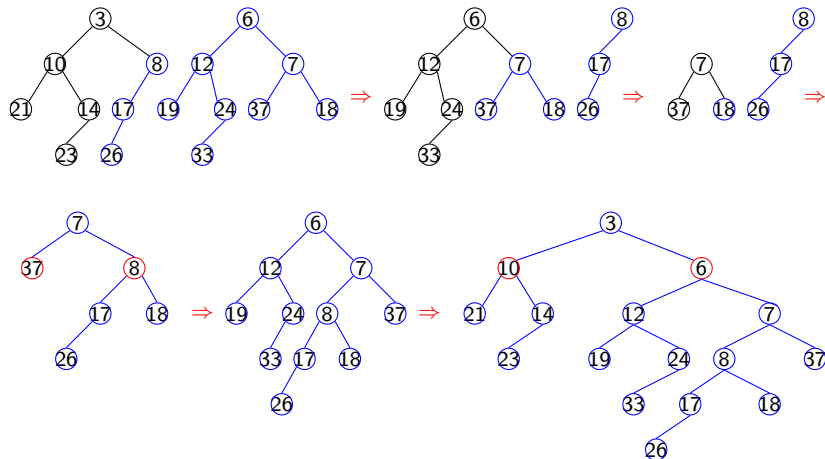


La **H1** og **H2** være to heaper.

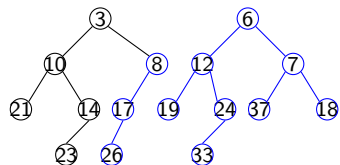
Merge kan gjøres rekursivt:

- ▶ sammenligner H1.rot med H2.rot. Antar nå at H1.rot er minst.
- ▶ la den høyre subheaven til H1 være heaven som man får ved å merge H1.høyre med H2
- ▶ bevare strukturkravet ved å bytte ut (swap) rotens høyre og ventre barn.

# Merge - Eksempel



# Merge - Eksempel

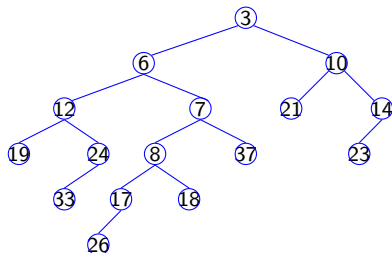


⇒

...

...

⇒



- ▶ **insert**: utføre merge med en en-noders heap
- ▶ **deleteMin**: fjerne rotnoden og merge høyre og venstre subheap
- ▶  $\mathcal{O}(\log n)$



**Neste Forelesning: 17. september**  
GRAFER