INF2220 - Algoritmer og datastrukturer

HØSTEN 2017

Ingrid Chieh Yu
Institutt for informatikk, Universitetet i Oslo

Forelesning 4:

Prioritetskø og Heap

Prioritetskø

- 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
  - 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 - Datastruktur og Kompleksitet

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)	$\mathcal{O}(n)/\mathcal{O}(\log_2(n))$	

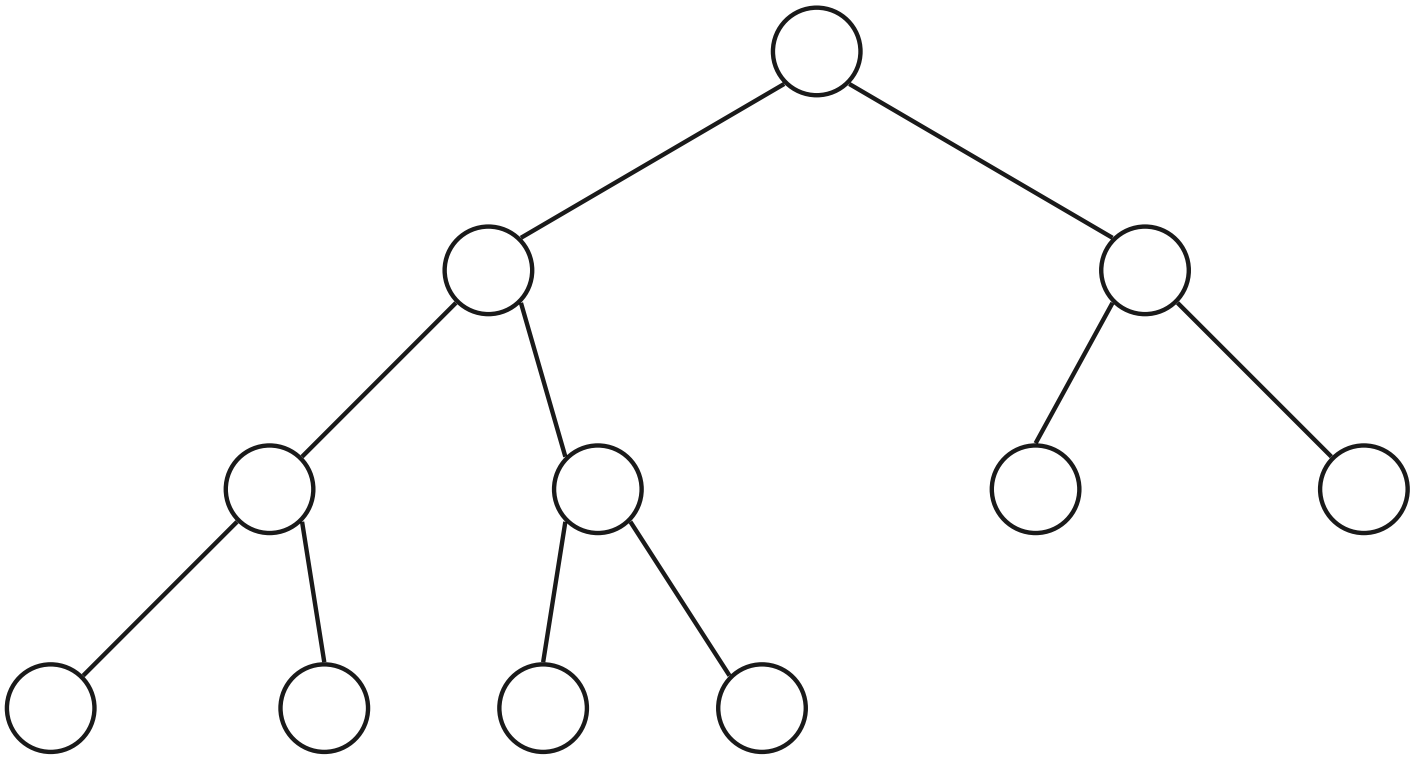
Prioritetskø - Heap

- 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 - Komplett Binætre

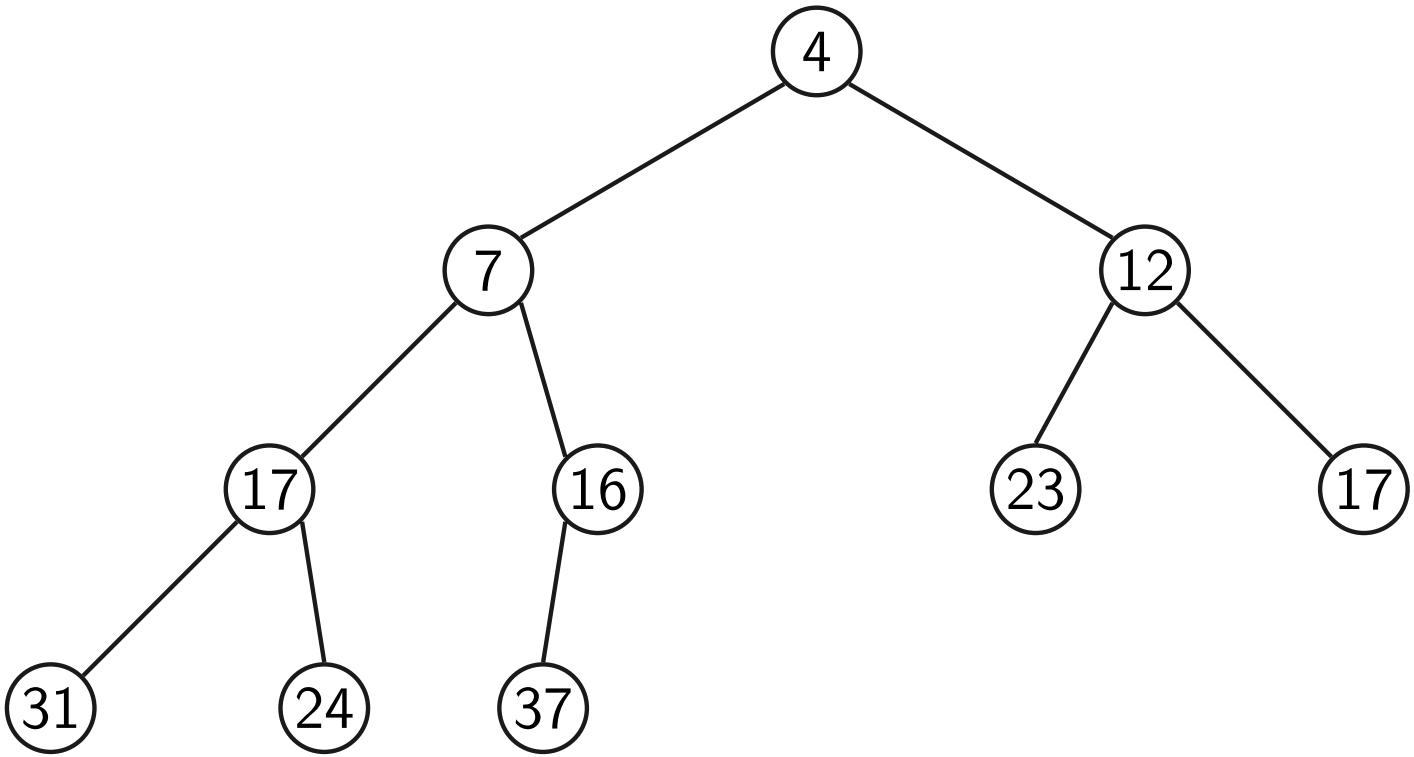


Binær Heap - Strukturkrav

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 $log_2(n)$

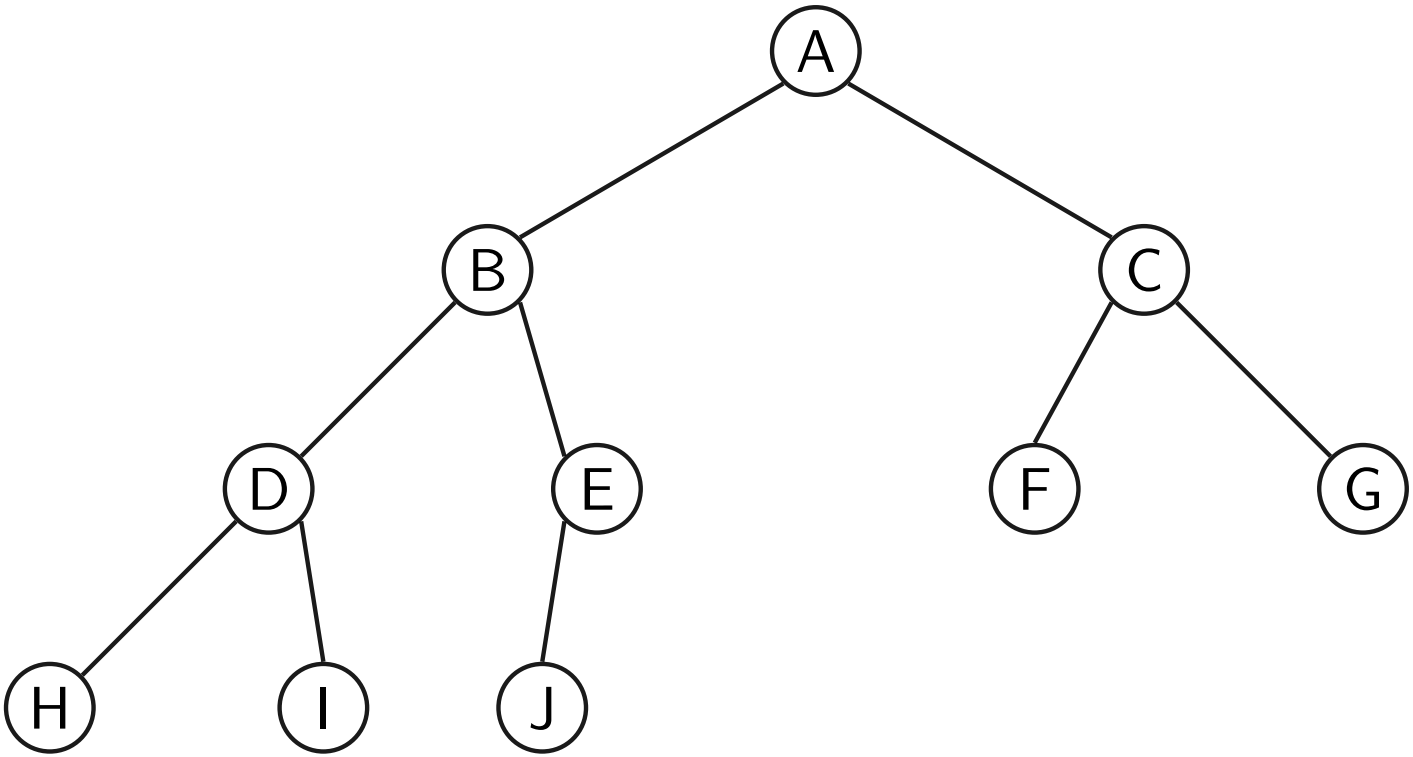
Binær Heap - Ordningskrav



Binær Heap - Representasjon

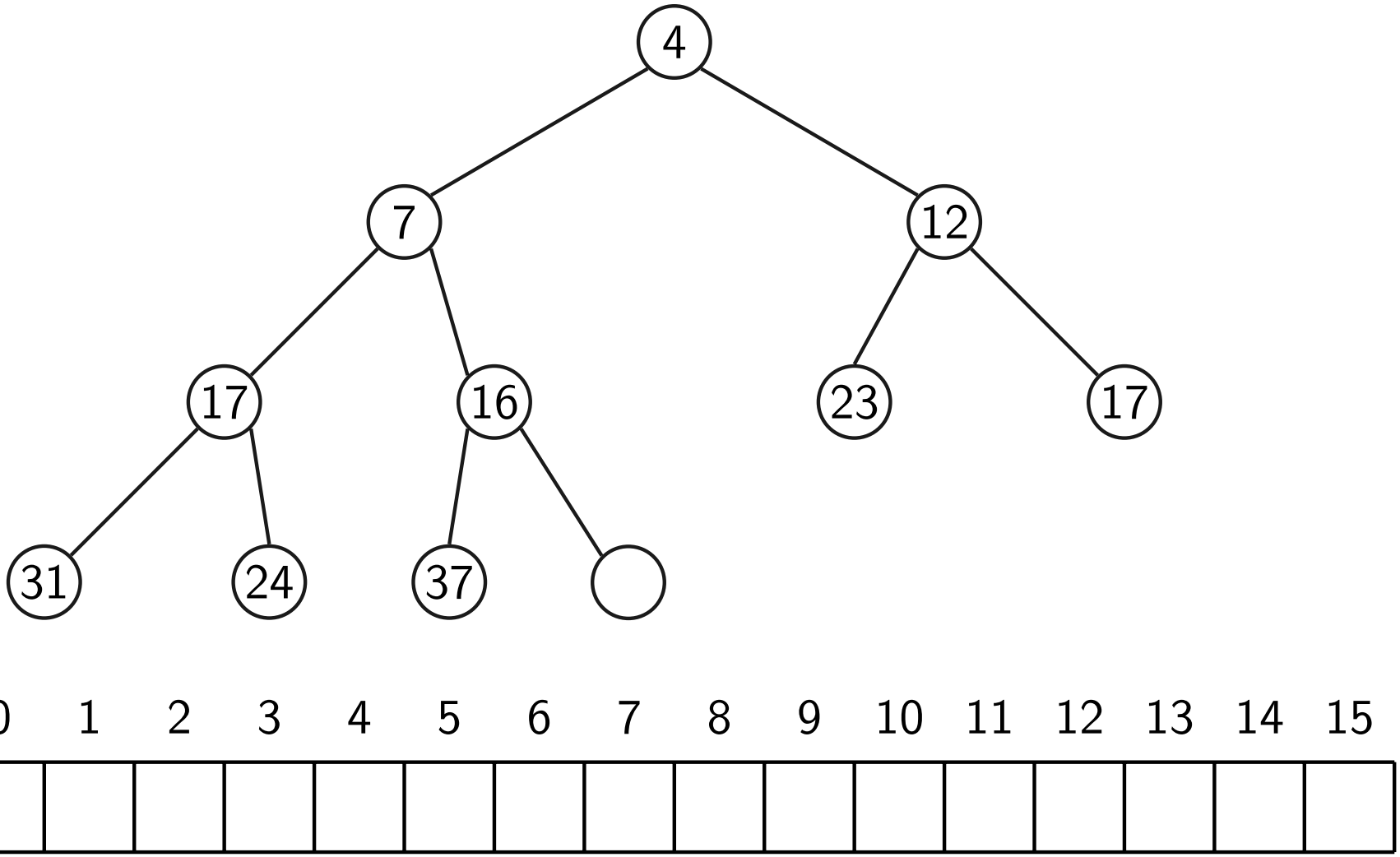
- Binærtreet er komplett så vi kan legge elementene i en array
- Vi kan enkelt finne foreldre og barn ut i fra array index
  - Venstre barn: **index** $\times$ **2**
  - Høyre barn: **index** $\times$ **2 +1**
  - Foreldre: **(int) index/2**
- Vi kan risikere å måtte allokere ny array og kopiere alle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

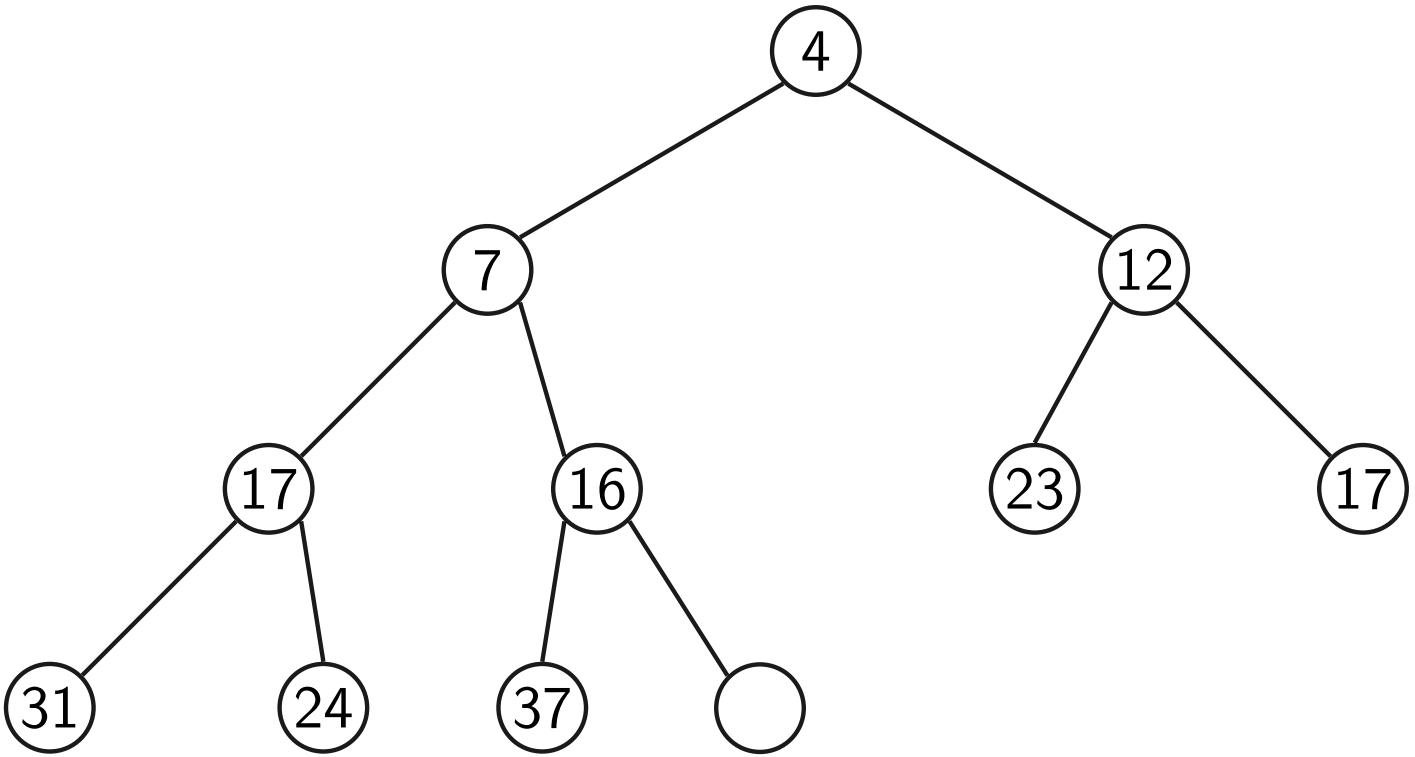
Binær Heap - `insert`

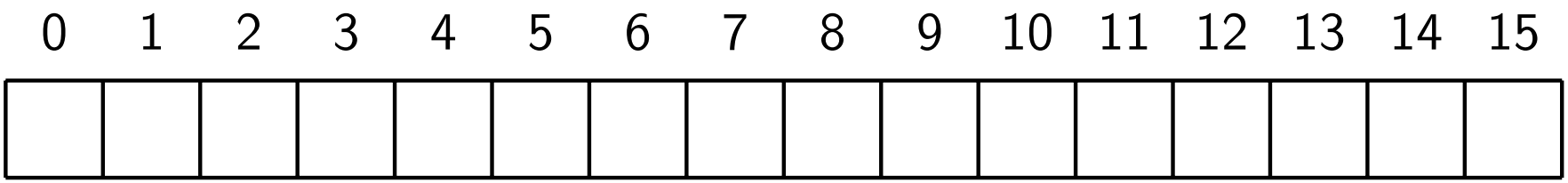


Binær Heap - `insert`

- 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`





Binær Heap - `deleteMin`

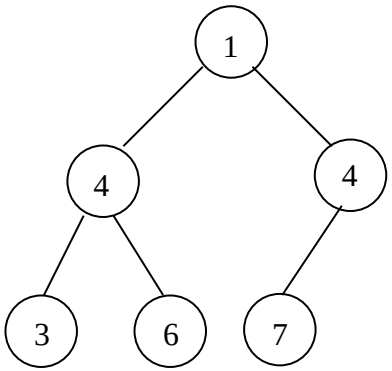
- Vi fjerner rot elementet fra heapen
- Vi lar det siste elementet bli ny rot
- Vi lar den nye rota flyte ned til riktig posisjon
- Dette kalles **percolate down** i læreboka
- Siden treet er i balanse kan vi maksimalt flyte $\mathcal{O}(log_2(n))$

Binær Heap - Andre Operasjoner

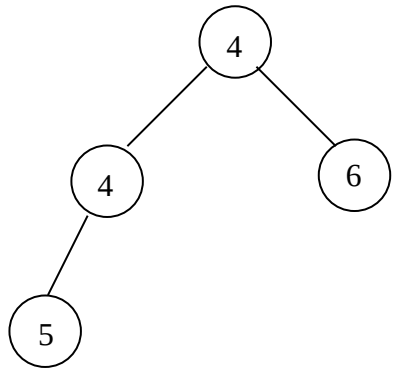
- `findMin` kan gjøres i konstant tid
- `delete` fjern vilkårlig element fra heapen
- Vi kan også endre prioritet på elementer i heap
  - Senking av prioritet kalles ofte `increaseKey`
  - Øking av prioritet kalles ofte `decreaseKey`
- Både `increaseKey` og `decreaseKey` gjøres typisk ved å:
  - Lokalisere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

Oppgave

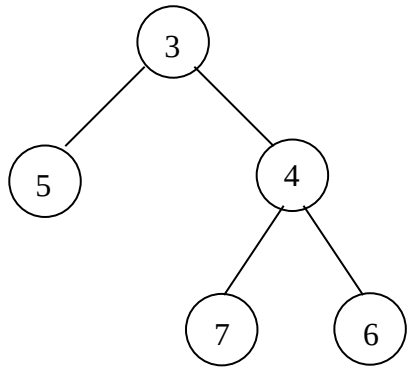
Hvilken av følgende BT er ikke en heap?:



(A)



(B)



(C)

1. A, B og C
2. B og C
3. A og C

Oppgave

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

1. $\mathcal{O}(n)$
2. $\mathcal{O}(n^2)$
3. $\mathcal{O}(n \log n)$
4. $\mathcal{O}(\log n)$

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**)

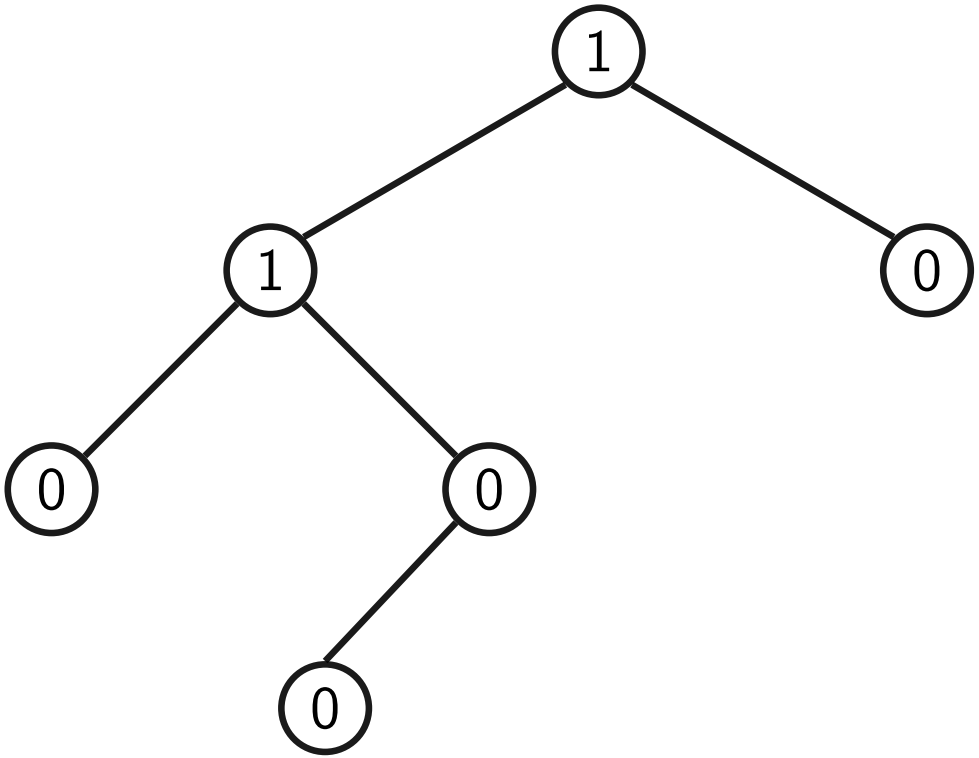
Oppgave

Gitt en array av Integer, hvordan sjekker man om den representerer en min-heap?

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

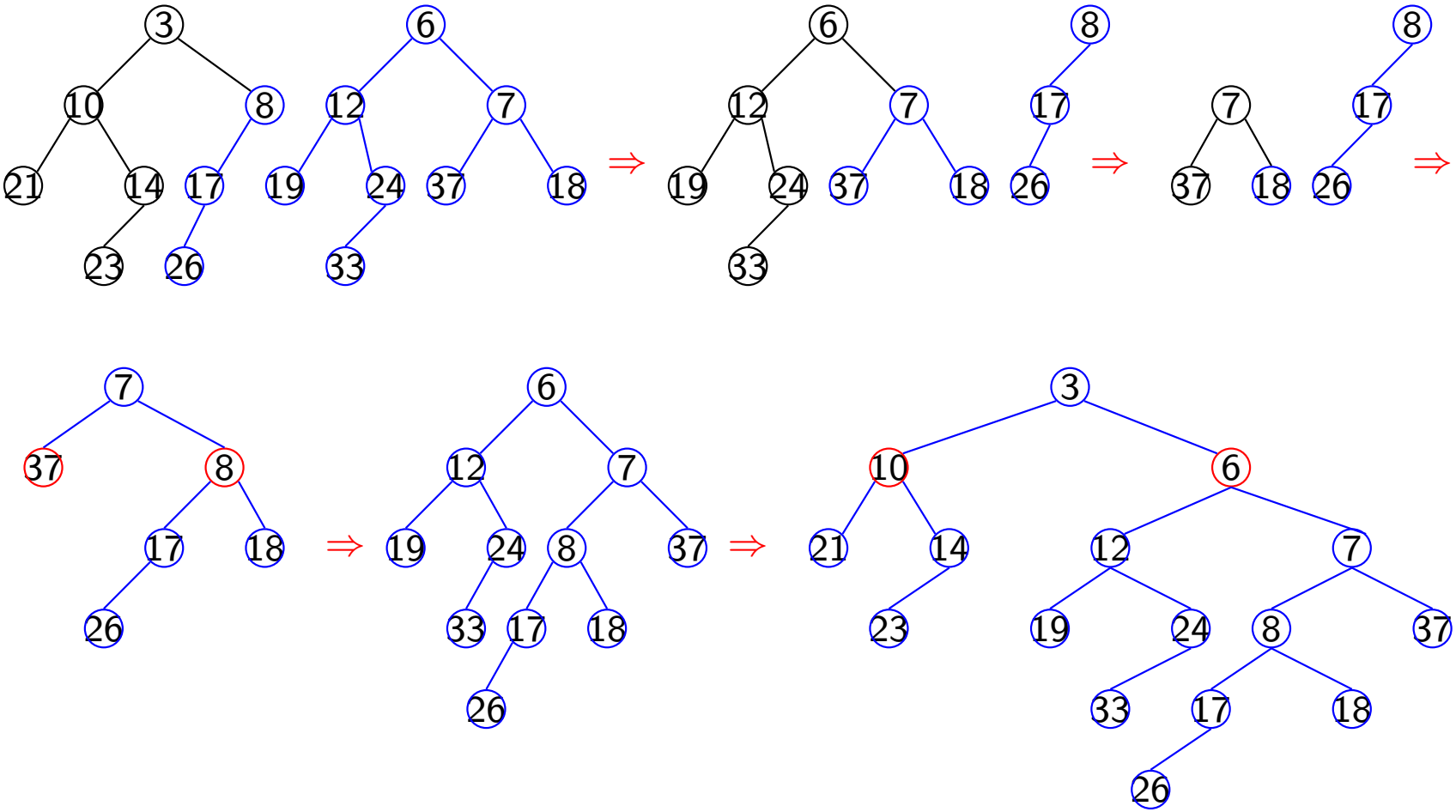
Venstreorientert Heap - Strukturkrav



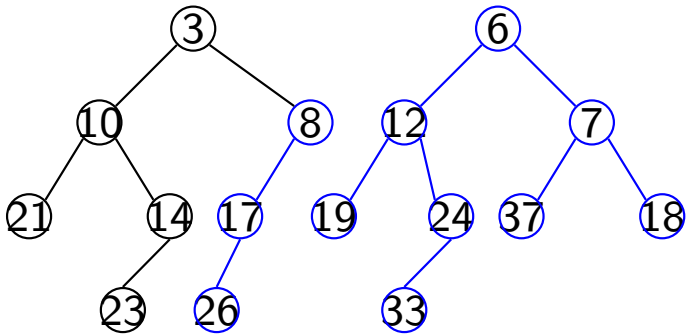
La **H1** og **H2** være to heaper.
`Merge` kan gjøres rekursivt:

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

Merge - Eksempel

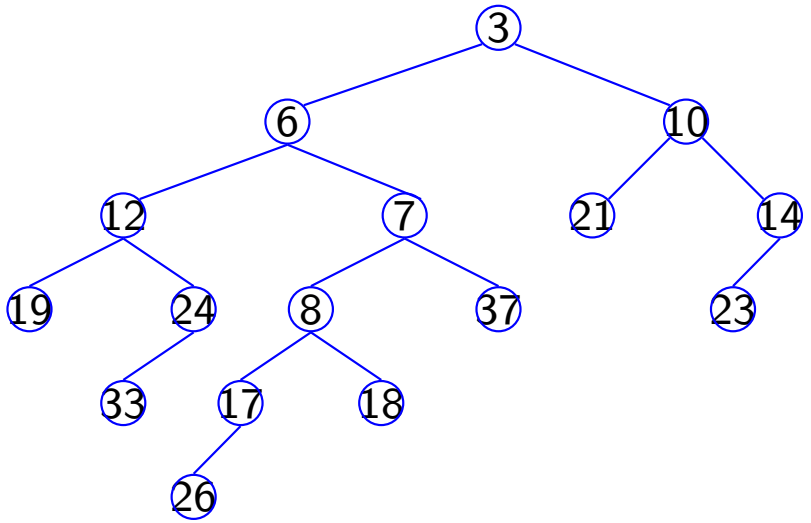


Merge - Eksempel



$\Rightarrow$ ...

... $\Rightarrow$



Huffman-koding

Motivasjon

- Store tekstfiler tar stor plass, og tar lang tid å overføre over nettet
- *Konklusjon*
  Det er behov for datakompresjon!

Idé og regler

- Hovedidé:
  Husk "Morse"
  Tegn som forekommer ofte ⇒ korte koder, sjeldne tegn ⇒ lange koder

- Regel 1:
  Hvert tegn som forekommer i filen, skal ha sin egen entydige kode
- Regel 2:
  Ingen kode er prefiks i en annen kode

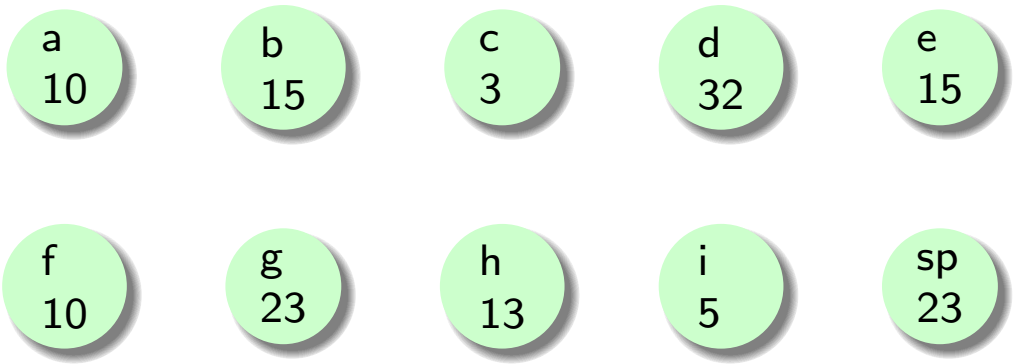
- Eksempel på regel 2:
  Dersom 011001 er (binær)kode for et tegn, kan hverken 0, 01, 011, 0110 eller 01100 være kode for noe tegn

Algoritme

- Lag en frekvenstabell for alle tegn som forekommer i datafilen
- Betrakt hvert tegn som en node, og legg dem inn i en prioritetskø P med frekvensen som vekt
- Mens P har mer enn ett element
  - Ta ut de to minste nodene fra P
  - Gi dem en felles foreldrenode med vekt lik summen av de to nodenes vekter
  - Legg foreldrenoden inn i P
- Huffmankoden til et tegn (bladnode) får vi ved å gå fra roten og gi en '0' når vi går til venstre og '1' når vi går til høyre
- Resultatfilen består av to deler:
  - En tabell over Huffmankoder med tilhørende tegn
  - Den Huffmankodede datafilen

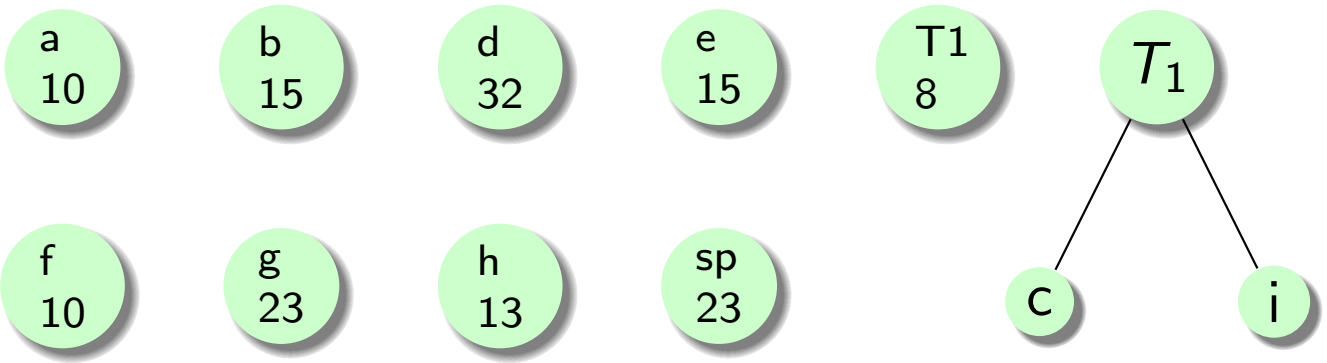
Eksempel

Initiell prioritetskø:

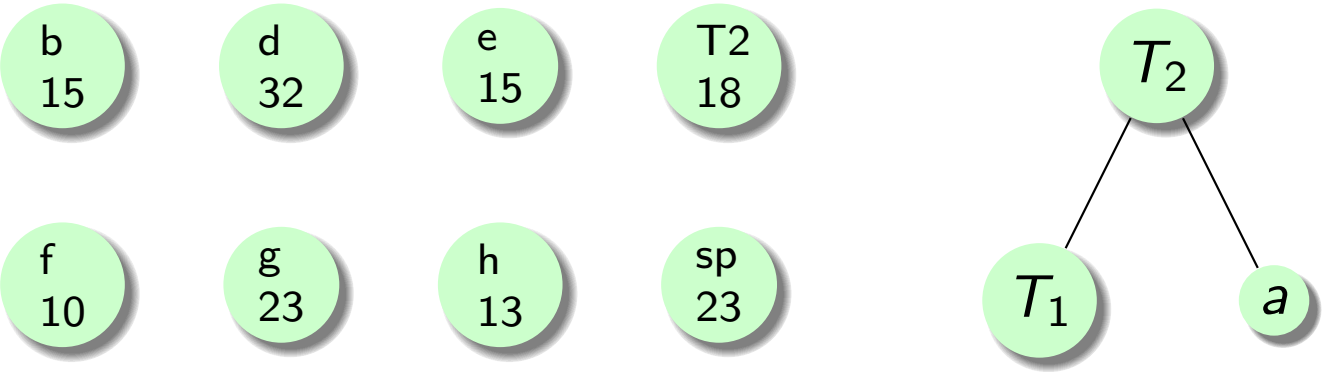


Bygging av treet

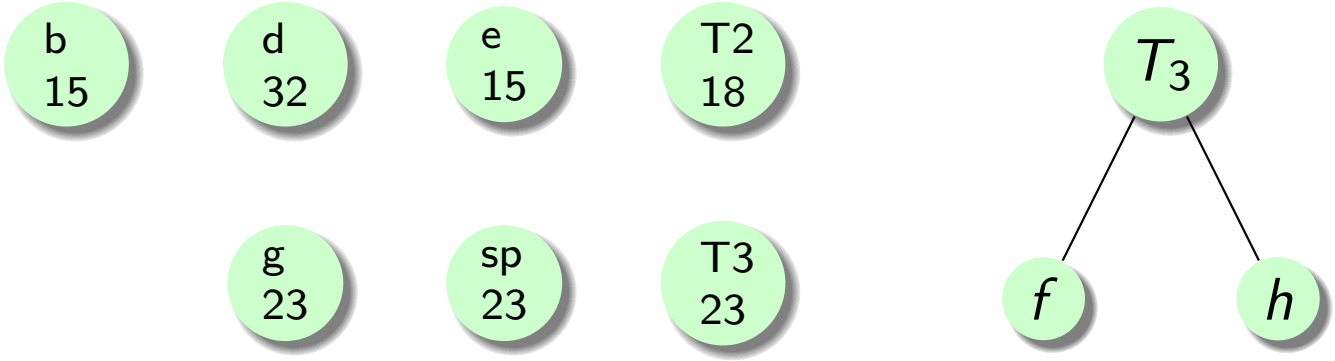
De 2 minste (sjeldnest forekommende) nodene er **c** og **i**.
Disse taes ut og erstattes med T_1 (vekt 8):



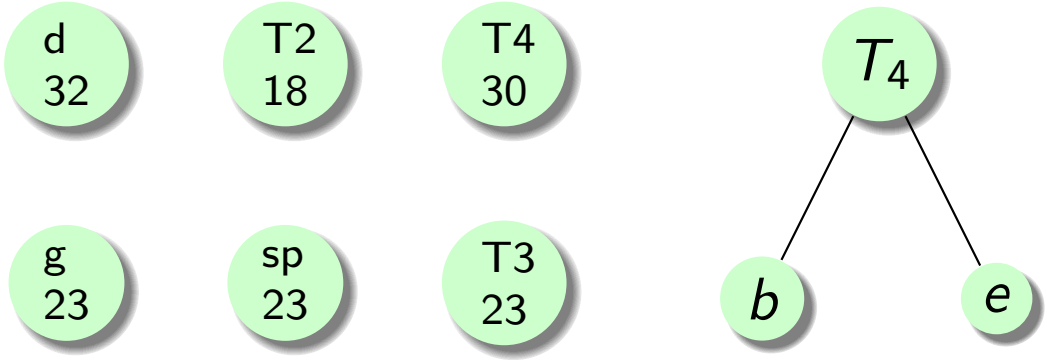
Dernest erstattes T_1 og a med T_2 (vekt 18):



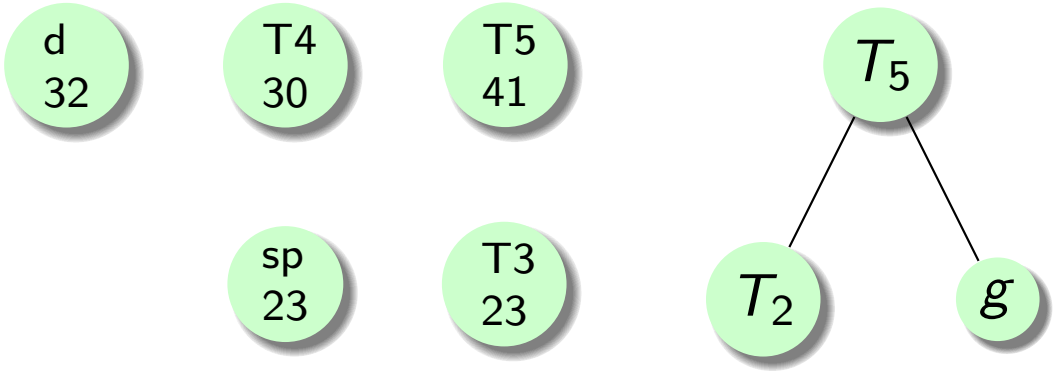
Så går f og h ut. De erstattes av T_3 (vekt 23):



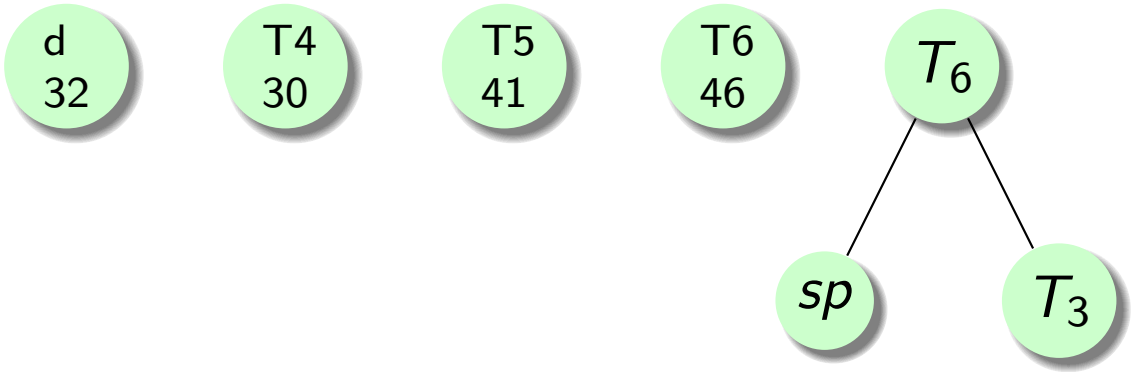
Så erstattes b og e med T_4 (vekt 30):



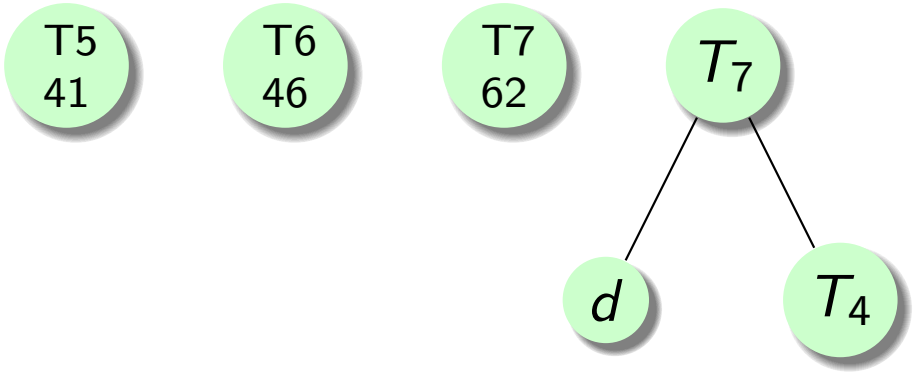
Dernest T_2 og g med T_5 (vekt 41):



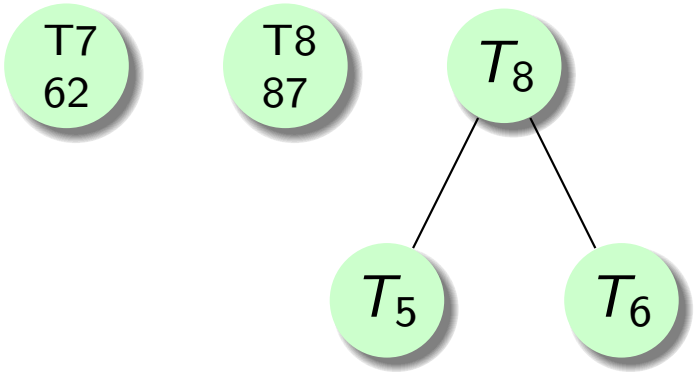
Så byttes sp og T_3 ut med T_6 (vekt 46):



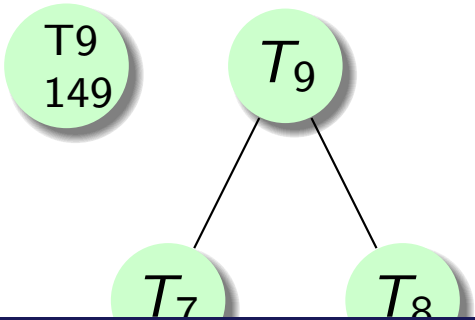
Så d og T_4 med T_7 (vekt 62):



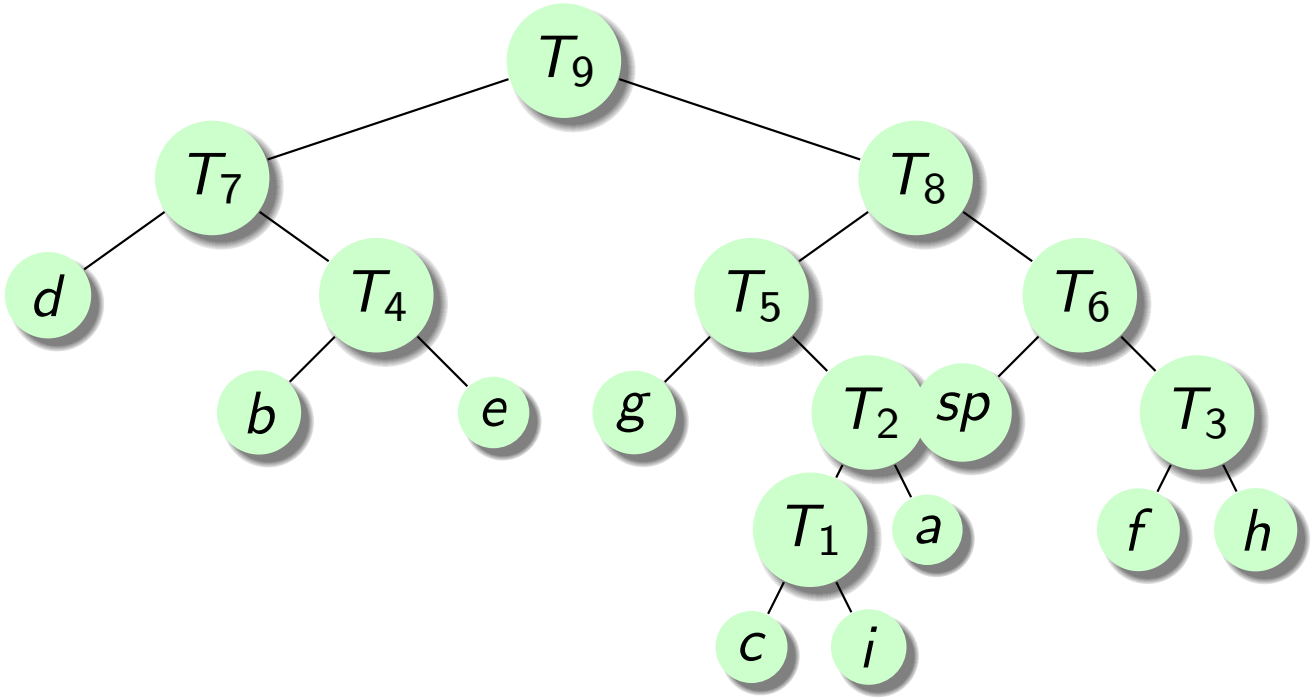
Så T_5 og T_6 med T_8 (vekt 87):



Endelig taes T_7 og T_8 ut av køen. Disse blir barn av rotnoden T_9:



Det ferdige kodetreet



ser slik ut:

Det gir denne kodetabellen:

(venstre 0, høyre 1)

a	–	1011	f	–	1110
b	–	010	g	–	100
c	–	10100	h	–	1111
d	–	00	i	–	10101
e	–	011	sp	–	110

Oppgave

Når vi komprimerer en fil med Huffmankoding så lagrer vi binærsekvensene som representerer bokstavene uten noe form for skilletegn. Hvordan kan vi vite når et tegn slutter og når neste begynner?

1. Det kan vi fordi alle sekvensene er like lange
2. Det kan vi siden det fins ikke to bit-sekvenser med samme start hvor den ene er lenger enn den andre
3. Det kan vi ved å bruke f.eks ASCII eller UTF-8 tabellene
4. Det er litt tilfeldig

Oppgave

Anta at en binærheap har N elementer, og høyde M. Hvor mange elementer må vi sette inn i heapen for at vi med sikkerhet kan si at den vil få høyde $M + 1$?

1. minst N+1
2. minst log(N)
3. minst N
4. minst 1

Neste Forelesning: 21. september

GRAFER + oblig 2