IN2010 - Algoritmer og datastrukturer

HØSTEN 2018

Ingrid Chieh Yu
Institutt for informatikk, Universitetet i Oslo

Forelesning 3:
Prioritetskø og Heap

Prioritetskø

- Algorithm design and applications kap. 5 (-5.2), 10.3
- Kø implementasjoner vi kjenner
  - FIFO - Liste
  - LIFO - Stack
  - adgang til elementene kun v.hj.a.plassering (Position)
- 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 - Nøkler og total ordning

- Fjerning av elementer er basert på nøkler (prioriteringer) som er tilordnet elementene når de settes inn i køen
- Hvert element har sin egen nøkkel, nøkler er ikke nødvendigvis unike
- Total ordning av nøkler:
  - reflexive: $k \leq k$
  - antisymmetrisk: hvis $k_1 \leq k_2$ og $k_2 \leq k_1$ da er $k_1 = k_2$
  - transitiv: hvis $k_1 \leq k_2$ og $k_2 \leq k_3$ da er $k_1 \leq k_3$

Prioritetskø - grensesnitt

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

`insert(k, e)` sett inn element **e** med nøkkel **k**
`removeMin()` returnerer og fjerne elementet med høyest prioritet (minste nøkkel)

Metode	Returverdi	innhold i kø
insert(5, A)		{ (5, A) }
insert(9, C)		{ (5, A), (9, C) }
insert(3, B))		{ (3, B). (5, A), (9, C) }
removeMin()	(3, B)	{ (5, A), (9, C) }
insert(7, D)		{ (5, A), (7, D), (9, C) }
removeMin()	(5, A)	{ (7, D), (9, C) }
removeMin()	(7, D)	{ (9, C) }
removeMin()	(9, C)	{ }
removeMin()	null	{ }

Prioritetskø - Datastruktur og Kompleksitet

Forskjellige datastrukturer kan implementere et slikt grensesnitt

- Liste (sortert eller uordnet)
- Søketre
- Heap

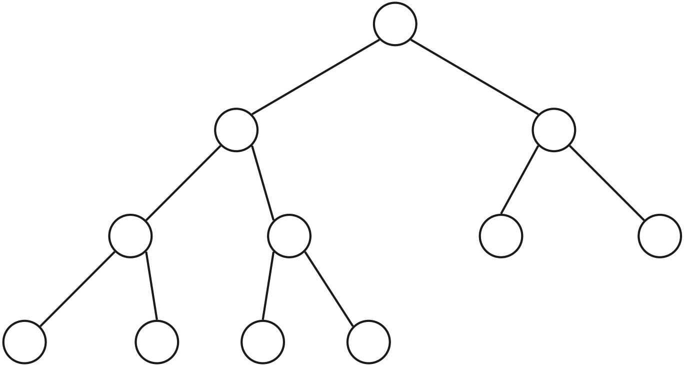
	insetting	sletting
uordnet liste	$\mathcal{O}(1)$	$\mathcal{O}(n)$
sortert liste	$\mathcal{O}(n)$	$\mathcal{O}(1)$
søketrær (worst/av. case)	$\mathcal{O}(n)/\mathcal{O}(\log_2(n))$	

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*

Binær Heap - Strukturkrav - Komplett Binætre

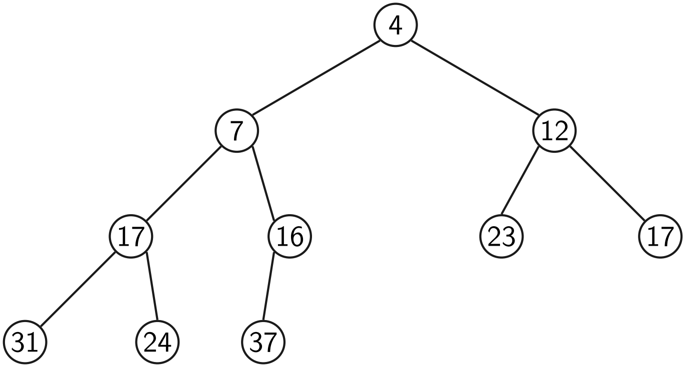


Binær Heap - Strukturkrav

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 $log_2(n)$
- hvis vi kan utføre oppdateringsoperasjoner i tid proporsjonal med høyden, vil disse operasjonene ta logaritmisk tid

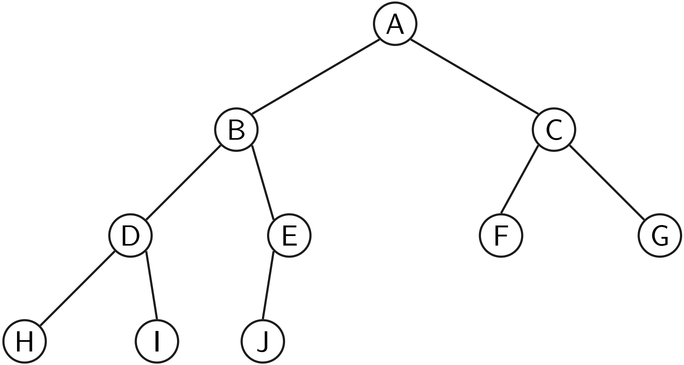
Binær Heap - Ordningskrav



Binær Heap - Representasjon

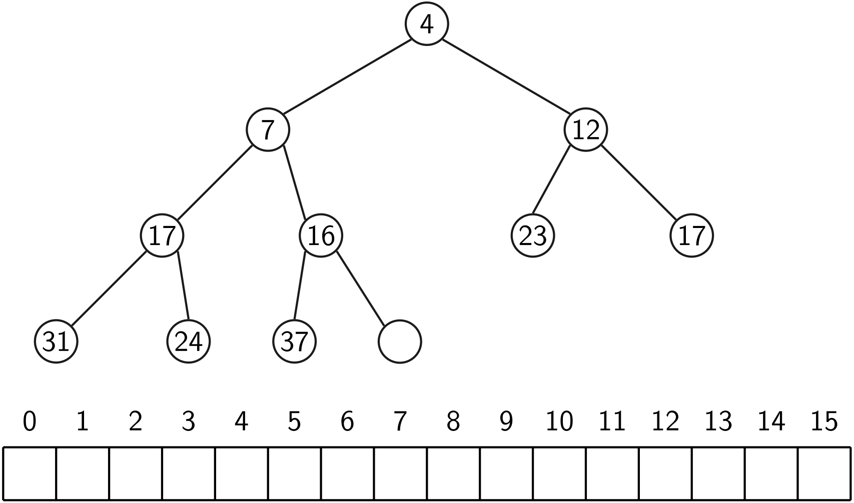
- Binærtreet er komplett så vi kan legge elementene i en array
- Vi kan enkelt finne foreldre og barn ut i fra array index
  - Venstre barn: **index** $\times$ **2**
  - Høyre barn: **index** $\times$ **2 +1**
  - Foreldre: **(int) index/2**
- Vi kan risikere å måtte allokere ny array og kopiere alle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

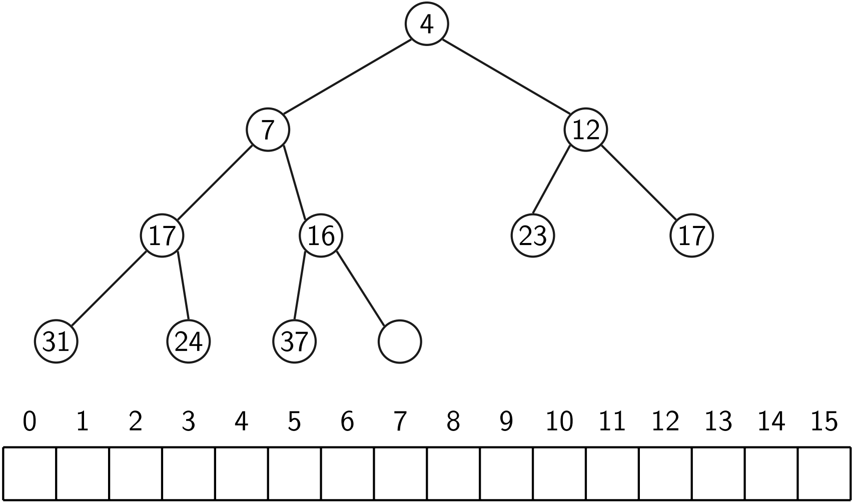
Binær Heap - `insert`



Binær Heap - `insert`

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

Binær Heap - `removeMin`



Binær Heap - `removeMin`

- 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
- Siden treet er i balanse kan vi maksimalt flyte $\mathcal{O}(log_2(n))$

Binær Heap - Andre Operasjoner

- `min` 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` ∞ + `removeMin`

Binær Heap - Sortering

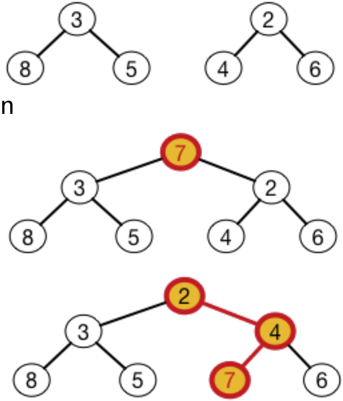
- Vi kan bruke en binær heap til å sortere
- Vi kan bygge en binær heap (`insert`) på $\mathcal{O}(n \cdot log_2(n))$
- Vi kan ta ut alle elementene (`removeMin`) på $\mathcal{O}(n \cdot log_2(n))$
- $2 \cdot \mathcal{O}(n \cdot log_2(n)) = \mathcal{O}(n \cdot log_2(n))$ (**worst case**)

Oppgave

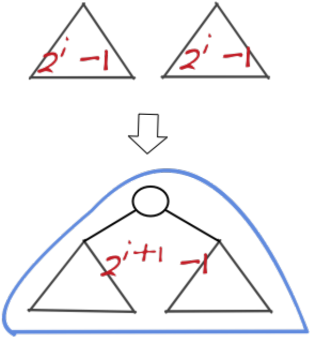
Gitt en array av Integer, hvordan sjekker man om den representerer en min-heap?

Flette to heaper

- gitt to heaper og nøkkel k
- vi lager en ny heap med k i rotnoden og med de to heaper som subtrær
- utfører down-heap og gjenopprette ordningskravet



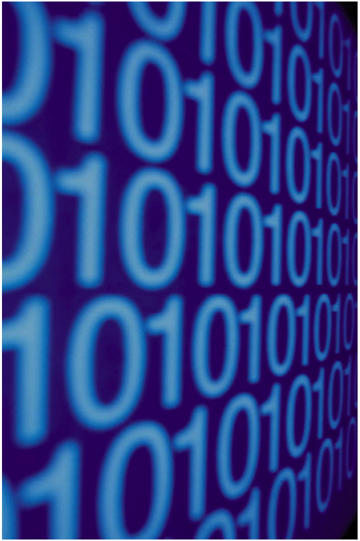
Bottom-Up Heap Construction



- gitt en liste av n nøkkel-element par, konstruerer heap $O(n)$ tid
- hvert rekursivt call returnere en subtre som er en heap

```
Algorithm BottomUpHeap(S):
  Input: A list S storing n = 2^h - 1 keys
  Output: A heap T storing the keys in S

  if S is empty then
      return an empty heap
  Remove the first key, k, form S
  Split S into two lists, S_1 and S_2, each of size (n-1)/2
  T_1 <- BottomUpHeap(S_1)
  T_2 <- BottomUpHeap(S_2)
  create binary tree T with root r string k, left subtree T_1
  and right subtree T_2.
  down-heap bubbling from r
  return T
```



Huffman-koding

Motivasjon

- Store tekstfiler tar stor plass, og tar lang tid å overføre over en kanal med lav båndbredde
- Enheter med begrenset lagringsplass og vi ønsker å maksimere antall dokumenter vi lagrer
- f.eks.,fixed-length encoding: ASCII 7-bits, Unicode 16-bits
- Unicode gir minst dobbelt så store filer som ASCII
- Det er behov for tekstkompresjon og variable-length encoding !

Idé og regler

- Hovedidé:
  Husk "Morse"
  Tegn som forekommer ofte $\Rightarrow$ korte koder, sjeldne tegn $\Rightarrow$ 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

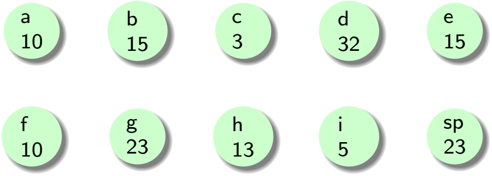
- Eksempel på regel 2:
  Dersom 011001 er (binær)kode for et tegn, kan hverken 0, 01, 011, 0110 eller 01100 være kode for noe tegn

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

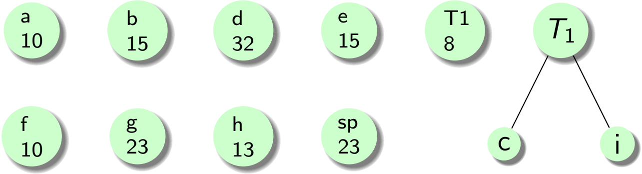
Eksempel

Initiell prioritetskø:

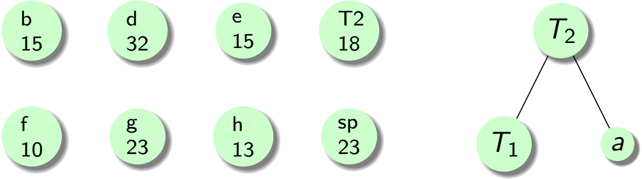


Bygging av treet

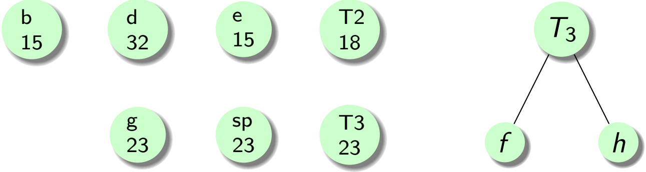
De 2 minste (sjeldnest forekommende) nodene er **c** og **i**.
Disse taes ut og erstattes med T_1 (vekt 8):



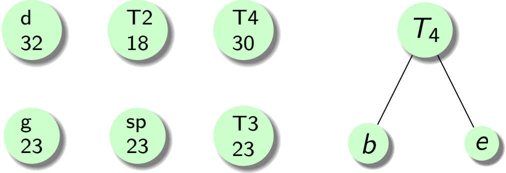
Dernest erstattes T_1 og a med T_2 (vekt 18):



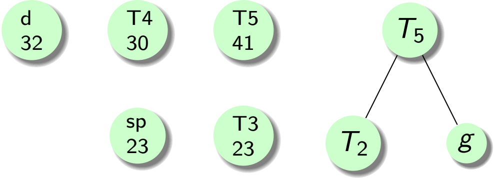
Så går f og h ut. De erstattes av T_3 (vekt 23):



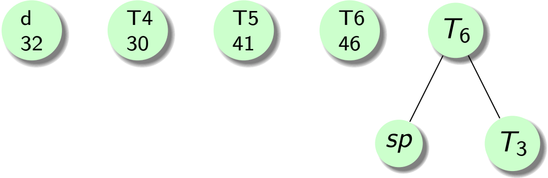
Så erstattes b og e med T_4 (vekt 30):



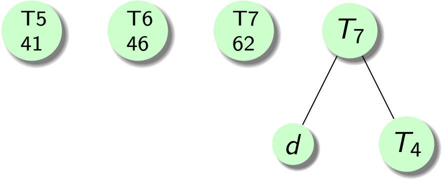
Dernest T_2 og g med T_5 (vekt 41):



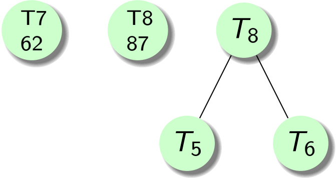
Så byttes sp og T_3 ut med T_6 (vekt 46):



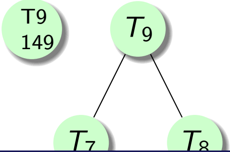
Så d og T_4 med T_7 (vekt 62):



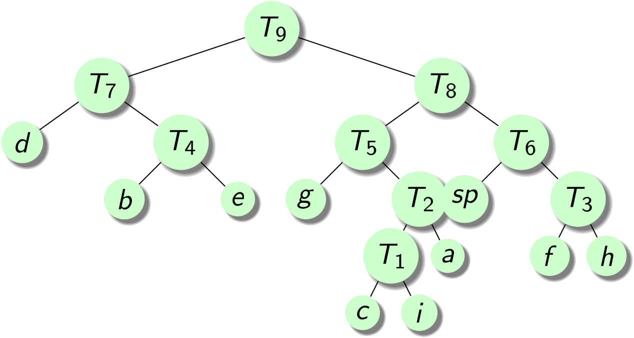
Så T_5 og T_6 med T_8 (vekt 87):



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:

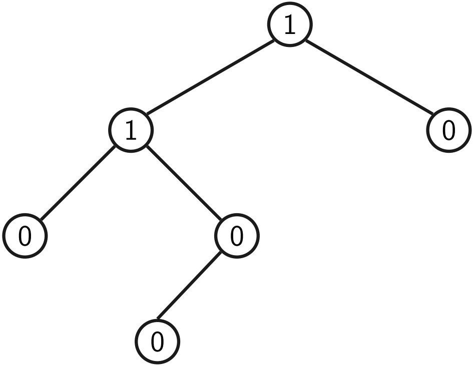
(venstre 0, høyre 1)

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

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

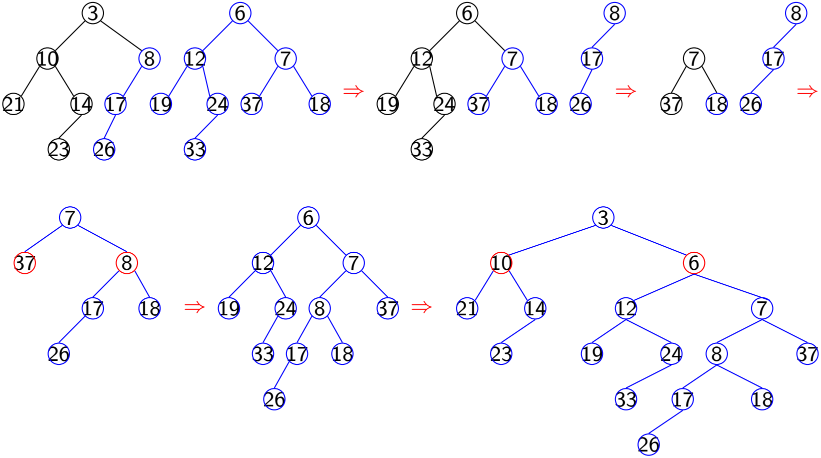
Venstreorientert Heap - Strukturkrav



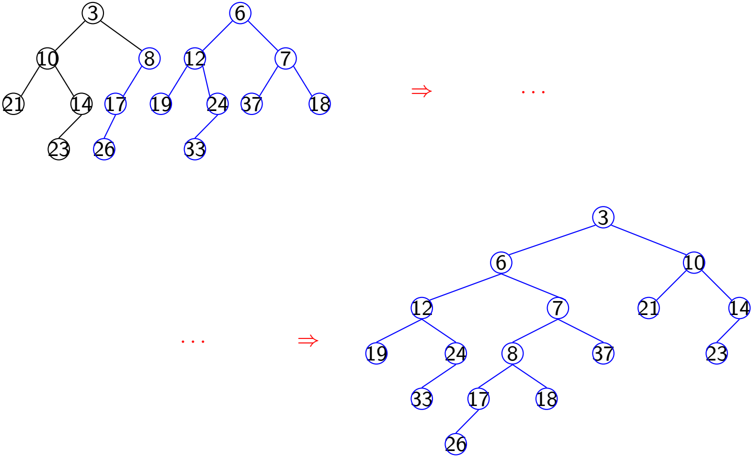
La **H1** og **H2** være to heaper.
`Merge` kan gjøres rekursivt:

- 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



Neste Forelesning: 14. september

GRAFER