Uke 7, Forelesning 1

- Forutsetninger for og essensen i faget
- Metodekall, rekursjon, permutasjoner
- Analyse av algoritmer
- Introduksjon til ADT'er
- De første ADT'er: Lister, stabler og køer

Uke 1, Uke 2 og Uke 3

- Flere ADT'er: Generelle trær, binære trær og binære søketrær
- Oblig. 1 og "dronning konkurransen"

Uke 4 og Uke 5

- Flere ADT'er: Hashing, hash-tabeller
- ADT'er for disk-datastrukturer introduseres: Utvidbar hashing, B-Trær

Uke 6

Vi avslutter utvidbar hashing fra forrige gang.

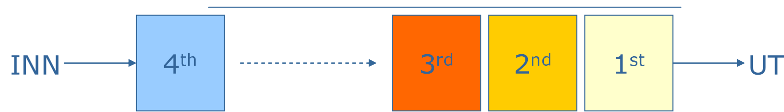
Vi ser på B-trær (så vidt nevnt) forrige gang:
- Operasjoner
- Optimal nodestørrelse

Vi introduserer prioritetskø ADT'en:
- Motivasjon
- Operasjoner
- Implementasjoner og tidsforbruk
- Innledende om heap-implementasjonen (mer neste uke)

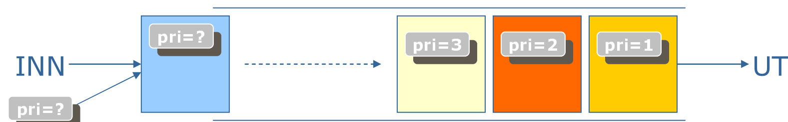
Introduksjon til prioritetskøer...

PRIORITETSKØ – *Motivasjon*

- I en vanlig kø (kapittel 3 i MAW) setter vi inn elementer i en ende av køen og tar ut elementer i den andre enden (First In First Out – FIFO).



- Ofte har vi bruk for å **prioritere** jobbene som vi legger i køen: Noe må gjøres med en gang, mens andre ting kan vente litt.



PRIORITETSKØ – *Eksempel*

Eksempel:
Jobbfordeleren (scheduler) i operativsystemer

- En Unix-server har som oftest mange brukere tilkoblet samtidig, og hver bruker har mange jobber som kjører på en gang (Netscape, Emacs, Java-kompilering, klokke, osv.).

- På en vanlig datamaskin i dag er det bare en prosessor (CPU), så bare en jobb kan fysisk kjøre (utføres) på et gitt tidspunkt.

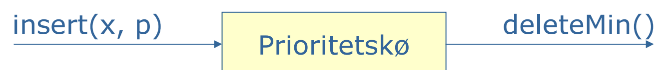
- Det er helt uakseptabelt å la hver jobb kjøre ferdig før neste jobb får slippe til.

Løsning

- En mulig løsning er å plassere jobbene i en kø.
- Når en jobb tas ut av køen, får den lov å kjøre en liten stund (noen millisekunder) før den legges bakerst i køen igjen. På den måten ser det ut som om alle jobbene kjører samtidig.
- Problemet er at korte jobber tar uforholdsmessig lang tid p.g.a. all ventingen.
- En god løsning er å senke prioriteten på jobber som allerede har kjørt lenge, slik at nye, korte jobber blir utført raskt.

En **prioritetskø** må minst støtte følgende to operasjoner:

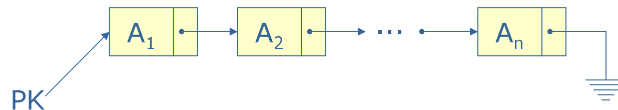
- **insert(x,p)** som setter element x inn i prioritetskøen med prioritet lik p (lite tall betyr høy prioritet, stort tall betyr liten priotiet).
- **deleteMin()** som tar ut og returnerer det elementet i køen som har høyest priortet (minst p).



- Vi skal senere se litt på noen andre operasjoner som en mer avansert prioritetskø kan tenkes å støtte.

PRIORITETSKØ – *Implementasjoner #1*

- Det er flere opplagte måter å implementere en prioritetskø på: **Enkel pekerkjede**, for eksempel



- setter alltid inn bakerst i pekerkjeden, *O* (1)
- må traversere listen for å finne det minste elementet, *O* (*n*)

PRIORITETSKØ – *Implementasjoner #2*

Sortert pekerkjede ($A_1 < A_2 < \quad < A_n$)

- **deleteMin** tar alltid ut bakerst fra listen, *O* (1)
- ved innsetting må vi gå gjennom listen for å finne riktig plass, *O* (*n*)
- Enkel pekerkjede er sannsynligvis bedre enn sortert pekerkjede fordi vi aldri kan ta ut flere elementer enn vi har satt inn.

Søketrær

- I alle søketrær bruker både **insert** og **deleteMin** i snitt *O*(log *n*) tid.
- Balanserte søketrær (f.eks. B-trær) bruker aldri mer enn *O* (log *n*) tid, men slike trær er forholdsvis komplekse strukturer.

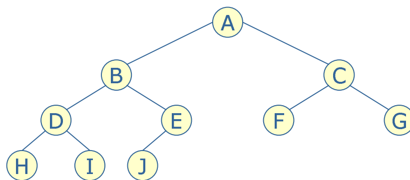
PRIORITETSKØ – *Heap implementasjon*

- En **heap**, også kalt **binary heap**, er et binært tre med et
  - **strukturkrav** dvs. hvordan treet ser ut, og et
  - **heap-ordningskrav** dvs. hvordan node-verdiene er plassert i forhold til hverandre.
- Heap-implementasjonen er så populær at mange bruker ordet 'heap' som et synonym for 'prioritetskø'.
- NB! Datastrukturen heap må ikke forveksles med "heapen" i et run-time system.
- **Insert** tar O (log n) tid i verste tilfelle, men konstant tid i gjennomsnitt.
- **DeleteMin** tar O (log n) både i verste tilfelle og i gjennomsnitt.
- Konstantleddene er små!

PRIORITETSKØ – *Strukturkravet til en heap #1*

Strukturkravet:

- En heap er et **komplett binærtre**, dvs. et tre hvor hvert nivå i treet er helt fylt opp med noder, med unntak av det nederste nivået som er fylt opp fra venstre mot høyre.

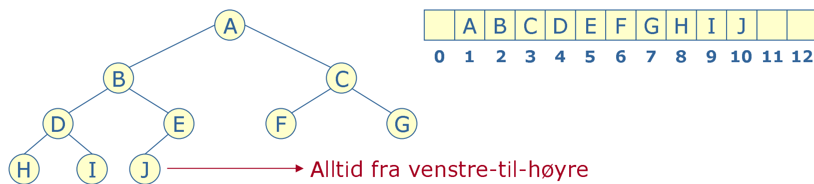


- Det er lett å se at alle heap'er må ha mellom $2h$ og $2^{h+1} - 1$ noder, der h er høyden til treet (NB! $1 + 2 + 4 + \ldots + 2h = 2^{h+1} - 1$).
- Dermed er høyden i treet (uttrykt ved n) lik $\lceil \log n \rceil$.

PRIORITETSKØ – *Strukturkravet til en heap #2*

- Innsetting og sletting består i å la en verdi henholdsvis flyte opp og synke ned i treet.
- Siden høyden er logaritmisk, vil operasjonene ikke bruke mer enn logaritmisk tid, dvs. $O(\log n)$.

- Vi skal tegne heap'er som trær, men på grunn av strukturkravet, behøver vi bare en enkelt array for å implementere en heap:

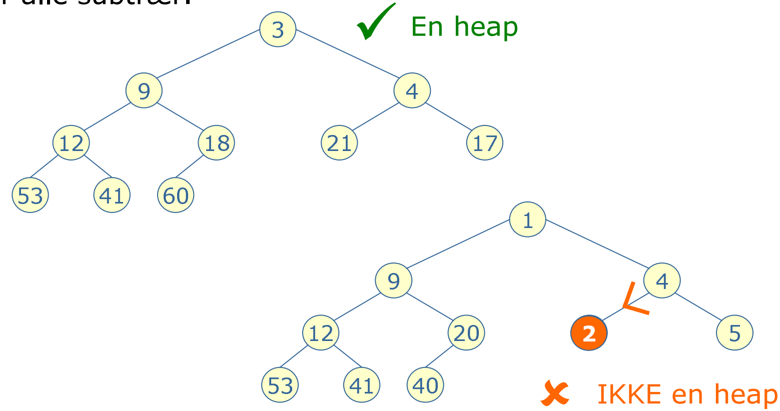


PRIORITETSKØ – *Strukturkravet til en heap #3*

- Noden i posisjon $H(i)$ har sitt venstre barn i $H(2i)$, sitt høyre barn i $H(2i + 1)$ og forelderen i $H(\lfloor i/2 \rfloor)$.
- Dermed blir operasjonene som traverserer treet ekstremt hurtige.
- Vi må bestemme en maksimal størrelse på heap'en på forhånd (evt. allokere mer plass når vi trenger det, men det er kostbart).

Heap-ordningskravet:

- Den minste verdien i treet er i roten. Dette skal også gjelde for alle subtrær.

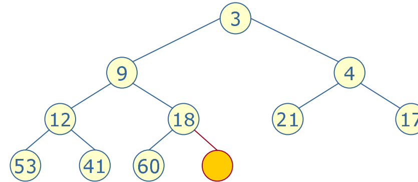


- Vi antar for enkelhets skyld at det bare er prioritetsverdiene som skal lagres i heap'en.
- Ordningskravet gjør at tilleggsoperasjonen **findMin()** kan utføres i konstant tid.

- Hvis man ønsker en "max-heap", kan man kreve at det største elementet skal være i roten av treet (og alle subtrær).

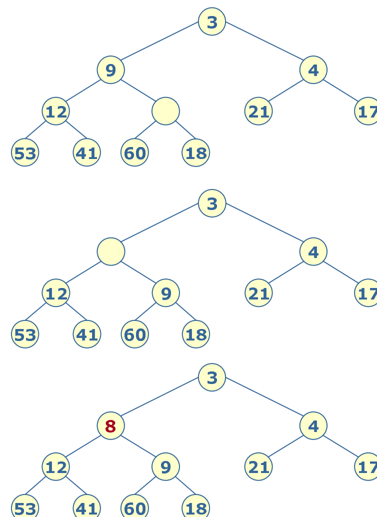
Insert

- Vi må sikre oss at heap-kravene er oppfylt etter at innsettingen er ferdig.
- Når vi skal sette inn et element X, må vi lage en "boble" (tom node) i den neste ledige plassen, ellers vil ikke treet være komplett.



- Vi sjekker om X kan settes inn der boblen står. Hvis det vil ødelegge heap-ordningskravet, lar vi boblen flyte oppover i treet (**percolate up**) inntil den kommer til et sted hvor X kan settes inn.

- **Eksempel:** Sett inn et element med prioritet 8 i heap'en på forrige lysark.
- 8 kan ikke settes inn i den nye boblen fordi forelderen i så fall vil ha større verdi enn barnet (18 > 8). Vi lar derfor boblen bytte plass med 18
- Elementet kan fortsatt ikke settes inn fordi 9 > 8, så vi lar boblen flyte oppover enda et nivå
- Nå kan 8 settes inn fordi 3 < 8.



PRIORITETSKØ – *Tidsforbruk*

- I verste fall må vi flytte boblen $O(\log n)$ ganger, men i gjennomsnitt flyter boblen opp bare 1,607 nivåer, dvs. at innsetting tar konstant tid i gjennomsnitt.
- Hvis elementet vi setter inn er den nye minsteverdien i heapen, vil boblen flyte helt opp til roten.
- For å slippe å teste på om vi er i roten (dvs. plass nr 1 i arrayen) hver gang vi flytter boblen, kan vi bruke en **dørvakt** (sentinel) i posisjon 0 i arrayen.
- Dørvakten må ha en verdi som **garantert** er mindre enn alle lovlige prioritetsverdier:

Array H:

-99	3	9	4	...	...
0	**1**	**2**	**3**	**4**	**5**

NESTE GANG – *Oppsummering*

Mer om prioritetskøer: Heap implementasjon (MAW kap. 6)

- Programeksempler
- Tilleggsoperasjoner
- Build Heap

Anvendelser

- Finn medianen
- Sortering
- Hendelsessimulering (Event Simulaton)