Dagens plan

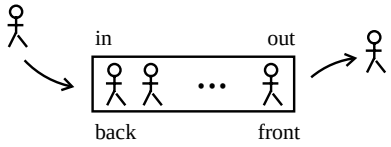
Prioritetskø ADT

- Motivasjon
- Operasjoner
- Implementasjoner og tidsforbruk
- Heap-implementasjonen
  - Strukturkravet
  - Heap-ordningskravet
  - Insert
  - DeleteMin
  - Tilleggsoperasjoner
  - Build Heap
- Anvendelser
  - Finn medianen og sortering
  - Hendelsessimulering (Event simulation)

Prioritetskø ADT

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.

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 vanlige datamaskiner har det frem til nylig bare vært en prosessor (CPU), og selv på de mest moderne er det ikke mange, så bare noen få jobber kan fysisk kjøre (utføres) på et gitt tidspunkt.
- Det er helt uakseptabelt å la hver jobb kjøre ferdig på en prosessor 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 pga. all ventingen.
- En god løsning er å senke prioriteten på jobber som allerede har kjørt lenge, slik at nye, korte jobber blir utført raskt.

Operasjoner

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

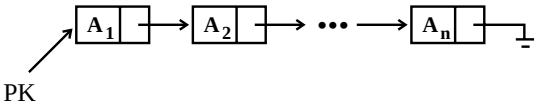
- **insert(x,p)** som setter element x inn i prioritetskøen med prioritet lik p (lite tall betyr høy prioritet, stort tall betyr liten prioritet).
- **deleteMin** som tar ut og returnerer det elementet i køen som har høyest prioritet (minst p).

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

Implementasjoner

Det er flere opplagte måter å implementere en prioritetskø på:

Enkel pekerkjede



- setter alltid inn bakerst i pekerkjeden, $\mathcal{O}(1)$
- må traversere listen for å finne det minste elementet, $\mathcal{O}(n)$

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

- **deleteMin** tar alltid ut bakerst fra listen, $\mathcal{O}(1)$
- ved innsetting må vi gå gjennom listen for å finne riktig plass, $\mathcal{O}(n)$

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 $\mathcal{O}(\log n)$ tid.
- Balanserte søketrær (f.eks. B-trær) bruker aldri mer enn $\mathcal{O}(\log n)$ tid, men slike trær er forholdsvis komplekse strukturer.

Heap-implementasjonen

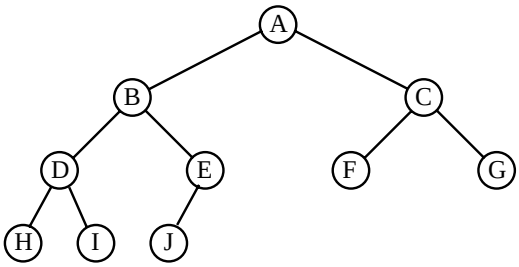
- En **heap**, også kalt **binary heap**, er et binært tre med et
  — **strukturkrav**
  (hvordan treet ser ut) og et
  — **heap-ordningskrav**
  (hvordan nodeverdiene er plassert i forhold til hverandre).
- Heap-implementasjonen er så populær at mange bruker ordet «heap» som et synonym for «prioritetskø», men vi skal skille mellom dem.
- **Insert** tar $\mathcal{O}(\log n)$ tid i verste tilfelle, men konstant tid i gjennomsnitt.
- **DeleteMin** tar $\mathcal{O}(\log n)$ både i verste tilfelle og i gjennomsnitt.
- Konstantleddene er små!

NB!
Datastrukturen heap må ikke forveksles med «heapen» i et run-time system.

Strukturkravet til en heap

Strukturkravet:

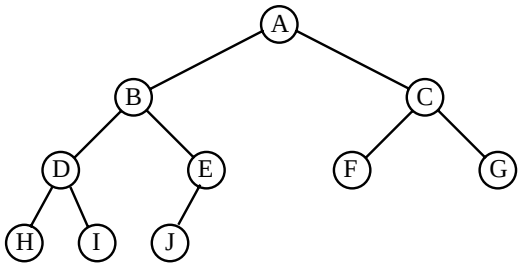
En heap er et **komplett binærtre**, dvs. et tre hvor hvert nivå i treet er helt fylt opp med noder, med unntak av det nederste nivået som er fylt opp fra venstre mot høyre.



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

- 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. $\mathcal{O}(\log n)$.
- Vi skal tegne heaper som trær, men på grunn av strukturkravet, behøver vi bare et enkelt array for å implementere en heap:



Array H:

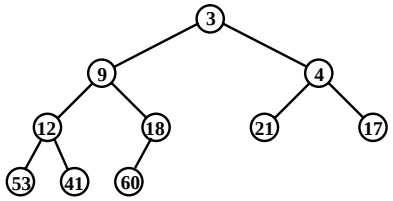
	A	B	C	D	E	F	G	H	I	J		
0	1	2	3	4	5	6	7	8	9	10	11	12

- 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 heapen på forhånd (evt. allokere mer plass når vi trenger det, men det er kostbart).

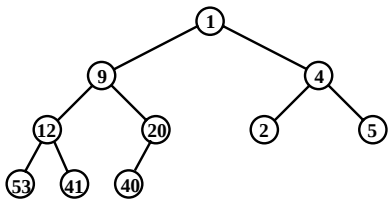
Ordningskravet til en heap

Heap-ordningskravet:

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



en heap

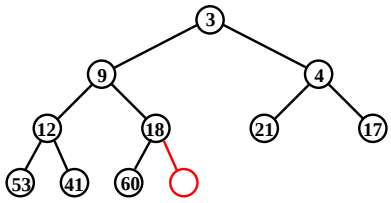


ikke en heap

- Vi antar for enkelhets skyld at det bare er prioritetsverdiene som skal lagres i heapen.
- 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.

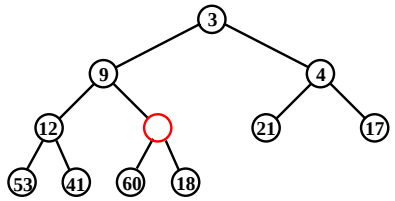


- Vi sjekker om X kan settes inn der boblen står. Hvis det vil ødelegge heap-ordningskravet, lar vi boblen flyte oppover i treet (**percolate up**) inntil den kommer til et sted hvor X kan settes inn.

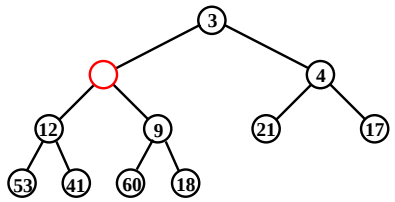
Eksempel:

Sett inn et element med prioritet 8 i heapen 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:

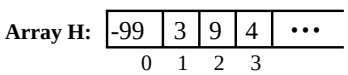


Nå kan 8 settes inn fordi $3 < 8$.

Tidsforbruk:

I verste fall må vi flytte boblen $\mathcal{O}(\log n)$ 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 (= 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:



Insert kodet i Java uten dørvakt: (side 207 i MAW)

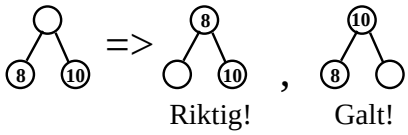
```
public void insert( AnyType x )
{
    if( currentSize == array.length - 1 )
        enlargeArray( array.length * 2 + 1 );

        // Percolate up
    int hole = ++currentSize;
    for( ; hole > 1 &&
      x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```

Med dørvakt hadde vi sluppet testen på $hole > 1$ i for-løkken.

DeleteMin

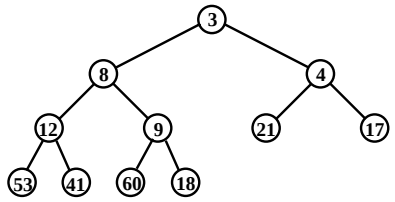
- Det er lett å finne det minste elementet, men når vi har fjernet det, er det en tom boble i roten.
- Samtidig er vi nødt til å flytte den «siste» noden X i heapen, siden treet skal krympe med et element og fortsatt være komplett.
- Hvis X kan plasseres i rotboblen uten å bryte ordningskravet, er vi ferdig.
- I motsatt fall lar vi boblen synke nedover i treet (**percolate down**) inntil X kan settes inn i boblen.
- Når boblen skal synke nedover, lar vi den bytte plass med sitt minste barn:



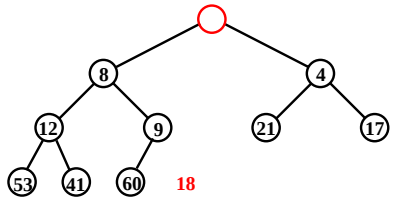
Riktig! Galt!

Eksempel:

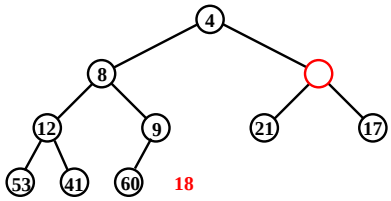
Utfør **deleteMin** på heapen nedenfor:



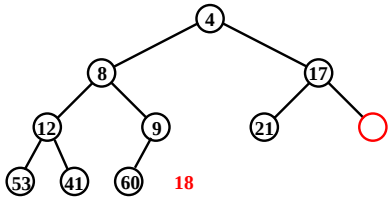
Først fjerner vi 3 fra rotboblen, og samtidig fjerner vi boblen rundt det siste elementet i heapen:



18 kan ikke settes inn i roten fordi barna har mindre verdier, så vi lar boblen synke nedover ved at den bytter plass med sitt minste barn:



Fortsatt kan ikke 18 settes inn (fordi $17 < 18$), så vi lar boblen synke videre nedover.



Nå kan 18 settes inn.

Tidsforbruk:

Det viser seg at i gjennomsnitt synker boblen nesten helt ned til bladnoden, dvs. at **deleteMin** er $\mathcal{O}(\log n)$ både i verste tilfelle og i gjennomsnitt.

Koding:

Vi må passe på det spesialtilfellet at antall elementer i heapen er like, og boblen flyter ned til elementet med bare 1 barn.

DeleteMin kodet i Java 5.0 (side 209 i MAW):

```
      public AnyType deleteMin( )
      {
        if( isEmpty( ) )
            throw new UnderflowException( );
        AnyType minItem = findMin( );
        array[1] = array[currentSize--];
// Siste element i treet flyttes til den ledige roten
// og får så synke ned til rett plass
        percolateDown(1);
        return minItem;
      }

      private void percolateDown( int hole )
      {
        int child;
        AnyType tmp = array[hole];
// hole inneholder verdien som skal synke ned (tmp)
        for( ; hole * 2 <= currentSize; hole = child )
        {
          child = hole * 2;
// child er venstre barn av hole
          if( child != currentSize &&
// sjekk at hole har et høyre barn
// og velg minste barn å sammenligne med
              array[child+1].compareTo(array[child])<0 )
              child++;
// flytt evt. opp minste barn og fortsett å synke
          if( array[child].compareTo(tmp) < 0 )
              array[hole] = array[child];
          else
              break;
        }
        array[ hole ] = tmp;
      }
```


Andre heap-operasjoner

- Selv om **findMin** er lett, så er heap-ordningskravet ikke til hjelp hvis vi ønsker å implementere tilleggsfunksjonen **findMax**.
- Det eneste vi vet om det største elementet, er at det befinner seg i en bladnode (hvorfor vet vi det?)
- Siden cirka halvparten av nodene i et komplett binærtre er bladnoder, er ikke det til noe særlig hjelp.
- Hvis det er viktig å vite hvor i heapen et gitt element er, må vi bruke en annen datastruktur i tillegg, f.eks. en hash.
- Hvis vi kjenner heap-posisjonen, i, til et element, kan følgende 3 operasjoner utføres i $\mathcal{O}(\log n)$ verstetid:
  - **DecreaseKey(i,Δ)**
  - **IncreaseKey(i,Δ)**
  - **Delete(i)**

DecreaseKey(i,Δ)

- **DecreaseKey** minsker prioriteten til elementet i posisjon 'i' med verdien Δ ($H[i] = H[i] - \Delta$ i en heltalls-heap).
- Heap-ordningen kan bli ødelagt, så vi lar elementet flyte oppover i heapen inntil den er kommet til en «lovlig» plass.
- **Eks (OS):** En systemadministrator kan øke prioriteten til en viktig jobb.

IncreaseKey(i,Δ)

- **IncreaseKey** øker prioriteten til elementet i posisjon 'i' med verdien Δ ($H[i] = H[i] + \Delta$).
- Heap-ordningen kan bli ødelagt, så vi lar elementet synke nedover i heapen inntil det er kommet på riktig plass.
- **Eks (OS):** Mange jobbfordelere senker automatisk prioriteten på jobber som har kjørt lenge.

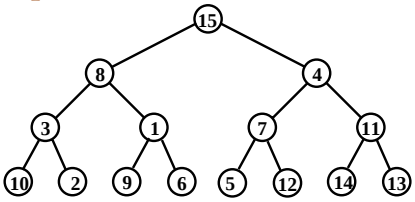
Delete

- **Delete** fjerner elementet i posisjon i fra heapen. Det kan gjøres ved først å utføre **DecreaseKey(i,∞)** og deretter **DeleteMin**.
- **Eks (OS):** Når en bruker dreper en prosess (kill), må den fjernes fra prioritetskøen.

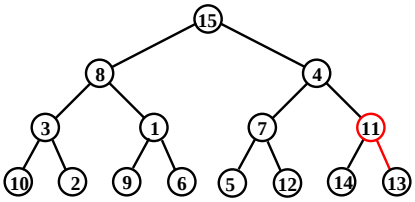
BuildHeap

- Anta at du har N elementer som du ønsker å sette inn i en tom heap.
- En enkel innsetting tar $\mathcal{O}(1)$ tid i gjennomsnitt og $\mathcal{O}(\log n)$ tid i verste fall.
- Dermed kan vi bygge heapen i $\mathcal{O}(n)$ gjennomsnittlig tid og $\mathcal{O}(n \log n)$ verstetid ved å utføre **Insert** N ganger.
- Er det mulig å klare å få lineær tid også i verste tilfelle?
- En smart **BuildHeap**-algoritme:
  1. Sett de N elementene inn i et komplett binærtre, men uten å tenke på om heap-ordingskravet er oppfylt.
  2. **for** i= N div 2 **downto** 1
     PercolateDown(i);
- **PercolateDown(i)** lar elementet i posisjon 'i' synke ned i treet.

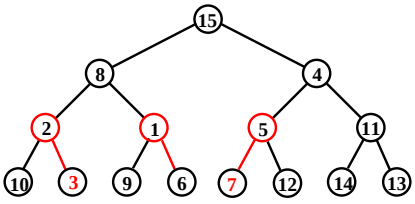
Eksempel



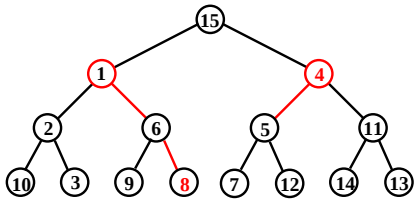
ikke heap



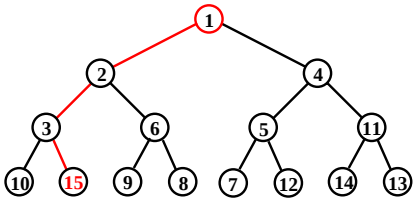
Etter PercolateDown(7)



PercolateDown 6,5 og 4



PercolateDown indeks 3 og 2



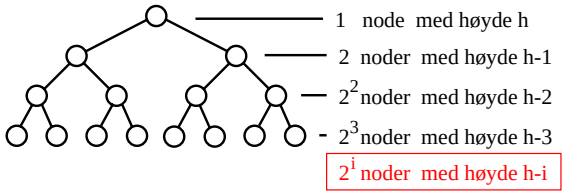
PercolateDown(1), nå er treet en heap

- Vi ønsker å vise at algoritmen har verstetid $\mathcal{O}(n)$.
- Vi observerer at i eksemplet over er arbeidet (antall flyttinger/tester) proporsjonalt med antallet røde kanter.
- Antall røde kanter kan maksimalt være lik summen av høydene til alle nodene i heapen.

- Hvis vi kan bevise at denne summen alltid er $\mathcal{O}(n)$, så er algoritmen også $\mathcal{O}(n)$.

Teorem 1 *I et komplett binætre med høyde h og med $2^{h+1}-1$ noder, er summen av høydene til alle nodene lik $2^{h+1}-1-(h+1)$.*

Bevis:



Altså er summen

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$S = h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1} \cdot 1$$

Vi bruker et triks: Vi trekker summen fra den dobbelte summen ($S = 2S - S$):

$$\begin{aligned} 2S &= \quad 2h \quad + 2^2(h-1) + \cdots + 2^{h-1} \cdot 2 + 2^h \cdot 1 \\ S &= h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1} \cdot 1) \end{aligned}$$

$$\begin{aligned} 2S - S &= -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h \\ &= -h + (1 + 2 + 4 + \cdots + 2^{h-1} + 2^h) - 1 \\ &= (2^{h+1} - 1) - (h+1) \end{aligned}$$

som var det vi skulle vise. ■

- Dette er en øvre grense.
- Siden en heap med høyde h har mellom 2^h og $2^{h+1}-1$ noder, så er summen $\mathcal{O}(n)$.

Anvendelser

Sortering

- Anta at vi skal sortere n elementer.
- Ved først å bruke **BuildHeap** og deretter **deleteMin** n ganger, har vi en rask $\mathcal{O}(n + n\log n) = \mathcal{O}(n\log n)$ sorteringsalgoritme.

Å finne medianen

- Vi kan også finne det k'te minste elementet ved å bruke **BuildHeap** og deretter **deleteMin** k ganger.
- Vi har dermed en $\mathcal{O}(n\log n)$-algoritme for å finne **medianen** ($k = n/2$).

Hendelsesimulering

- Ved simulering (f.eks. av kassakøer i et supermarked) deler man tiden inn i diskrete biter kalt "ticks".
- I stedet for for hvert tick å sjekke om det er registrert en hendelse på dette tidspunktet, legger man hendelsene inn i en prioritetskø ordnet etter hendelsestidspunktet.
- Neste hendelse kan da taes ut fra toppen av heapen.
- Dette sparer mye prosesseringstid, spesielt ved høy tidsoppløselighet.