Uke 7,
Forelesning 2

- 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

Uke 4 og Uke 5

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

Uke 6

- Prioritetskøer

Uke 7.1

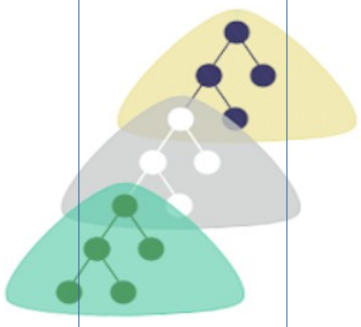
OVERSIKT – *Uke 7, Forelesning 2 (W7.L2)*

Vi fortsetter med og avslutter prioritetskøer.
Vi skal se på:

- Heap implementasjon (MAW kap. 6)
  - Programeksempler
  - Tilleggsoperasjoner
  - Bygge opp en heap

- Anvendelser
  - Finn medianen
  - Sortering
  - Hendelsessimulering (Event Simulaton)

TEMA: PRIORITETSKØER



Prioritetskøer: Heap-implementasjon og anvendelser

- Vi har sett følgende forrige gang:
  - Strukturkravet: At en heap er et komplett binært tre (et nivå er helt fylt, unntatt nederste nivå som fylles fra venstre til høyre
  - Ordningskravet: At en heap har minste verdi i roten (også for alle subtrær)
  - Insert-operasjonen med ”percolation” oppover (bobling oppover), med/uten dørvakt

Insert – kode i Java, uten dørvakt (MAW side 189):

```
public void insert( Comparable x ) throws Overflow
{  if( isFull( ) )
       throw new Overflow( );

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

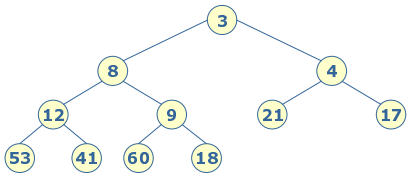
Q: Hva hvis vi hadde brukt en dørvakt?
A: Med dørvakt hadde vi sluppet ”hole > 1” testen i for-løkken.

DeleteMin: Fjerner minste elementet…

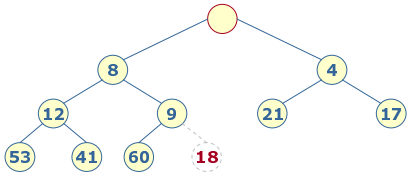
- Det er lett å finne det minste elementet, men når vi har fjernet det, er det en tom boble i roten.
- Samtidig er vi nødt til å flytte den "siste" noden X i heapen, siden treet skal krympe med et element og fortsatt være komplett.
- Hvis X kan plasseres i rotboblen uten å bryte ordningskravet, er vi 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!

DeleteMin eksempel:
Utfør **deleteMin** på heapen:



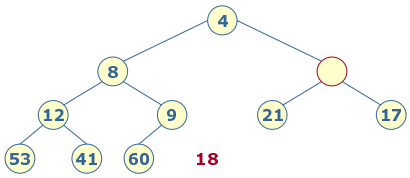
Først fjerner vi 3 fra rotboblen, og samtidig fjerner vi boblen rundt det siste elementet i heapen:



Men 18 kan ikke settes inn i roten fordi barna har mindre verdier…

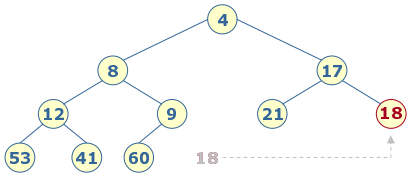
HEAP – *DeleteMin eksempel #2*

Siden 18 kan ikke settes inn i roten fordi barna har mindre verdier, lar vi 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!



HEAP – *Tidsforbruk & Koding*

Tidsforbruk:

- Det viser seg at i gjennomsnitt synker boblen nesten helt ned til bladnoden, dvs. at **deleteMin** er 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.

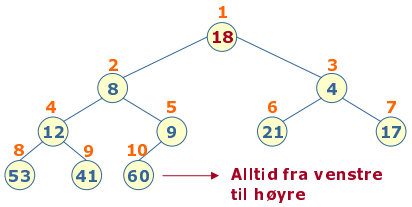
```
public Comparable deleteMin( )
{
    if( isEmpty( ) ) return null;
                                        // NB! currentSize = 11 i eksemplet.
    Comparable minItem = findMin( );    // Litt redundant. Minste er i rot-noden array[1]
    array[1] = array[currentSize--];    // Flytt siste element til roten...
                                        //          og reduser array-størrelsen med 1,
                                        //          dvs. currentSize = 10.

    percolateDown(1);                   // Så får siste elementet som nå er i roten
                                        //          synke ned til rett plass.

    return minItem;
}
```


HUSK-

- Array implementasjon



ALTSÅ: Hvis mor er på array index **i** (f. eks. **i = 3**) … 4

Da er venstre barn på **2×i** (**= 6**), 21

og høyre barn på **2×i + 1** (**= 3**), 17

-	18	8	4	12	9	21	17	53	41	60			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
private void percolateDown( int hole )      // F. eks. hole = 1 ved inngang, verdi = 18
{   int child;
    Comparable tmp = array[hole];           // tmp 'husker' array[1] = 18
    // hole inneholder verdien som skal synke ned (verdien er mellomlagret i tmp)
    // currentSize = 10

    for( ; hole * 2 <= currentSize; hole = child )      // Så lenge venstre barn er i heap'en;
                                                        // og child blir ny hole i neste omgang.
    {   child = hole * 2;                       // child er venstre barn til hole.
                                                // f. eks. child = 2

        // Velg minste barn: Hvis child (venstre barn) ikke er den siste, og hvis høyre
        // barn (=4) er mindre, velg høyre barn. Ellers behold venstre barn.
        if( child != currentSize && array[child+1].compareTo(array[child])<0 ) child++;

        // NB! Nå er child = 3, høyre barn = array[3] = 4 og tmp = 18.
        if( array[child].compareTo(tmp) < 0 )       // Ja, 4 < 18.
            array[hole] = array[child];             // Altså array[1] = array[3] = 4
        else break;                                 // og for-løkken fortsetter med hole=3
    }
    array[ hole ] = tmp;                    // NB! Siste 'hole' får verdien 18.
}
```

-	18	8	4	12	9	21	17	53	41	60	18		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- 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?)
- Men … siden cirka halvparten av nodene i et komplett binært tre 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 heap-posisjonen til et element, **i**, er kjent, kan følgende operasjoner utføres i O (log n) verstetid: **DecreaseKey(i, Δ)**, **IncreaseKey(i, Δ)** og **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.

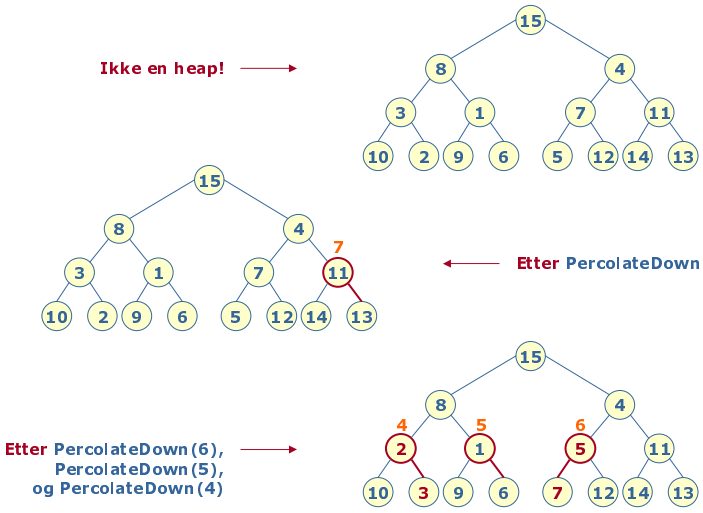
Delete

- **Delete** fjerner elementet i posisjon '**i**' fra heapen. Det kan gjøres ved først å utføre **DecreaseKey(i,1)** og deretter **DeleteMin**.
- **Eks (OS):** Når en bruker dreper en prosess (kill), må den fjernes fra prioritetskøen.

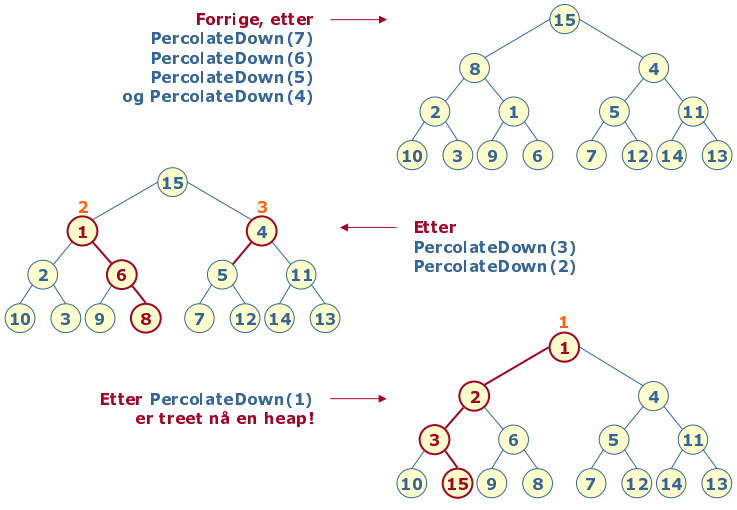
BuildHeap

- Anta at du har N elementer som du ønsker å sette inn i en tom heap.
- En enkel innsetting tar **O (1)** tid i gjennomsnitt og **O (log n)** tid i verste fall.
- Dermed kan vi bygge heapen i **O (n)** gjennomsnittlig tid og **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:
  - Sett de N elementene inn i et komplett binært tre, men uten å tenke på om heap-ordingskravet er oppfylt.
  - for i= N div 2 downto 1 PercolateDown(i);
- **PercolateDown(i)** lar elementet i posisjon '**i**' synke ned i treet.

HEAP – *Eksempel: Bygg en Heap #1*

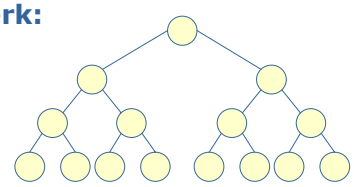


HEAP – *Eksempel: Bygg en Heap #2*



- Vi ønsker å vise at bygge-algoritmen har verstetid ***O* (*n*)**.
- Vi observerer at i eksemplet over er arbeidet (antall flyttinger/tester) proporsjonalt med antallet kanter for hver perkolering (de røde kantene).
- Antall slike (røde) kanter kan maksimalt være lik summen av høydene til alle nodene i heapen.
- Altså:
  Hvis vi kan bevise at denne summen alltid er ***O* (*n*)**, så er algoritmen også ***O* (*n*)**. Det er det vi skal forsøke...

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



— 2^0 noder med høyde h-0
— 2^1 noder med høyde h-1
— 2^2 noder med høyde h-2
— 2^3 noder med høyde h-3

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

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

BEVIS:

- Vi bruker et triks. Vi trekker summen fra den dobbelte summen $S = 2 \cdot S - S$

$$
\begin{aligned}
2 \cdot S = \quad & 2 \cdot h + 2^2 \cdot (h-1) + 2^3 \cdot (h-2) + \ldots + 2^{(h-1)} \cdot 2 + 2^h \\
- \quad S = \quad & h + 2 \cdot (h-1) + 2^2 \cdot (h-2) + 2^3 \cdot (h-3) + \ldots + 2^{(h-1)} \cdot 1 \\
\hline
S = 2 \cdot S - S = \quad & -h + 2 + 4 + 8 + \ldots + 2^{(h-1)} + 2^h \quad +1-1 \\
= \quad & -h + (1 + 2 + 4 + 8 + \ldots + 2^{(h-1)} + 2^h) - 1 \\
= \quad & (2^{h+1} - 1) - (h+1)
\end{aligned}
$$

  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 $O(n)$.

Sortering

- Anta at vi skal sortere n elementer.
- Ved først å bruke **BuildHeap** og deretter **DeleteMin** n ganger, har vi en rask $O(n + n \log n) = O(n \log n)$ 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 $O(n \log n)$-algoritme for å finne **medianen** ($k = n/2$).

Hendelser og simulering ved hjelp av en heap

- 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.

Disjunkt mengde ADT (Disjoint Sets, MAW kap. 8.1-8.5)

- Relasjoner aRb
- Ekvivalens relasjoner,
  ekvivalens klasser,
  det dynamiske ekvivalensproblemet
- Operasjoner
- Implementasjon