

# Prioritetskøer ved »heap«, simulering

## Yet another teknik til at repræsentere mængder

Hvor hashtabellen fremviste:

- Konstant tid for finde og indsætte
- men ingen »sortering« af elementerne

har vi for »heap« eller »hob«

- konstant tid for at finde minimum □ *ikke meningsfyldt for hashing!!*
- indsætte logaritmisk
- slette minimum logaritmisk

## Også interessant fordi

- elegant og overraskende teknik for repræsentation af træer

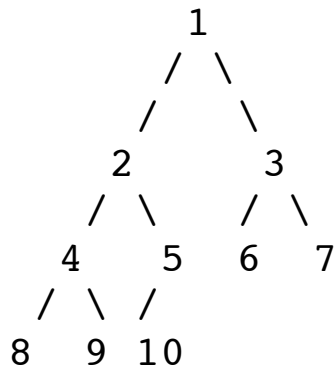
## Dagens program

- Princippet, folde binært træ sammen i array
- Indsætte og slette ved at lade elementer sive ned eller op i træet
- andre operationer
- En afledt sorteringsmetode: Smid elementerne i en hob og træk dem ud igen
- Vigtig anvendelse »discrete-event simulation«
  - kommende begivenheder sættes i kø, prioriteret efter tid

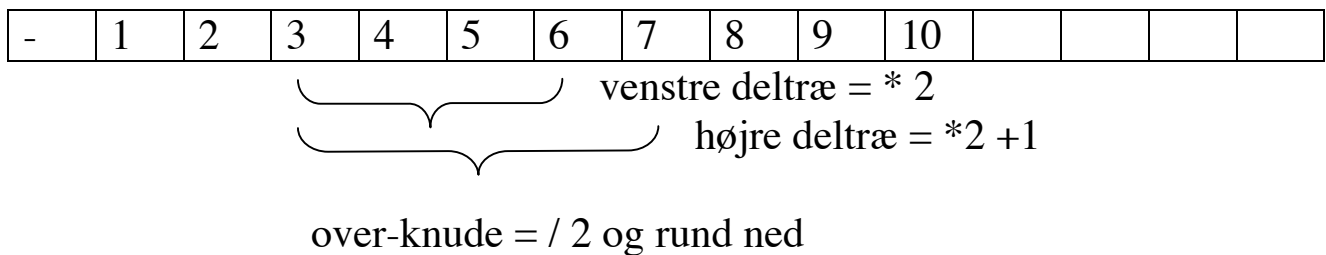
NB: »heap« bruges også om et lagerareal for dynamisk oprettelse af objekter, dvs. »new xxxx«

## Repræsentation af fuldstændigt binært træ i array

**Definition:** Et *fuldstændigt binært træ* er lige dybt over det hele bortset fra nederste etage, som er fyldt fra venstre mod højre.



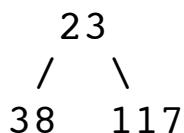
Nummerering antyder mulig placering i array:



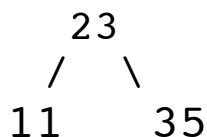
(Der er sgi' da smart)

Træet ovenfor er ikke en binær hob.

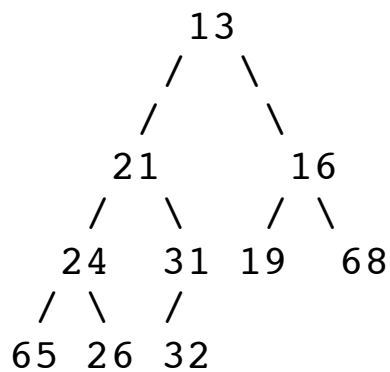
**Definition:** En *binær hob* er et fuldstændigt binært træ hvor, for hver knude ved værdi  $k$ , er enhver værdi i undertræerne  $\geq k$ .



*Bemærk: en anden egenskab end det binære søgetræ*



## Eksempel på binær hob og repræsentation i array



-	13	21	16	24	31	19	68	65	26	32				
---	----	----	----	----	----	----	----	----	----	----	--	--	--	--

Tydeligvis ikke sorteret — men en svagere egenskab, som er nemmere at vedligeholde

### Operationer:

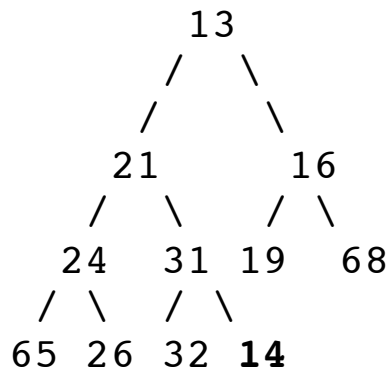
findMin()  $\approx$  return array[1] ;)

deleteMin()  $\approx$  slette array[1] og ryste strukturen så  
egenskaben kommer på plads igen

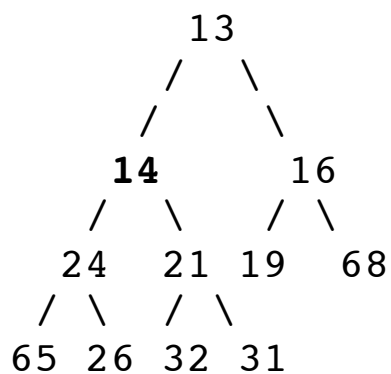
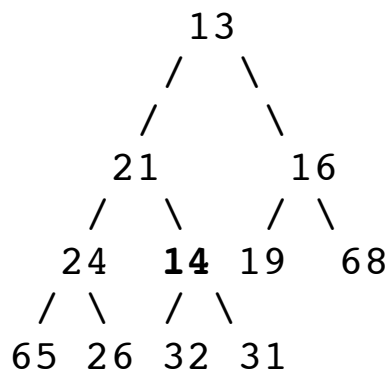
insert(X)  $\approx$  placere i bunden og skubbe op indtil pengene passer

## Indsættelse i binær hob

1) Indsætte på næste ledige plads



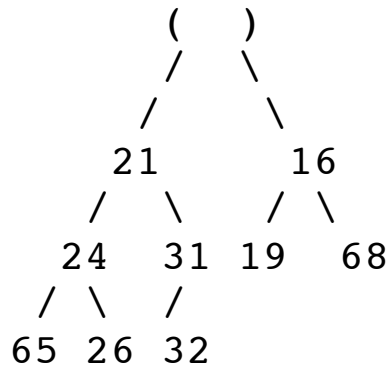
2) Så længe der er knas med overboen, byt med ham:



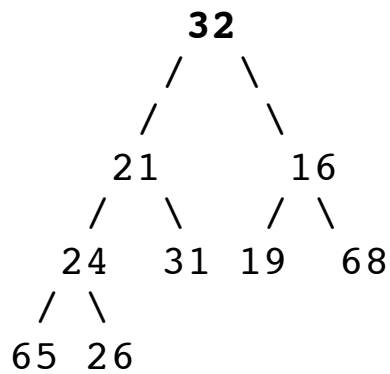
Det var da nemt  $\approx$  indlysende logaritmisk tid

## At fjerne mindste element i binær hob

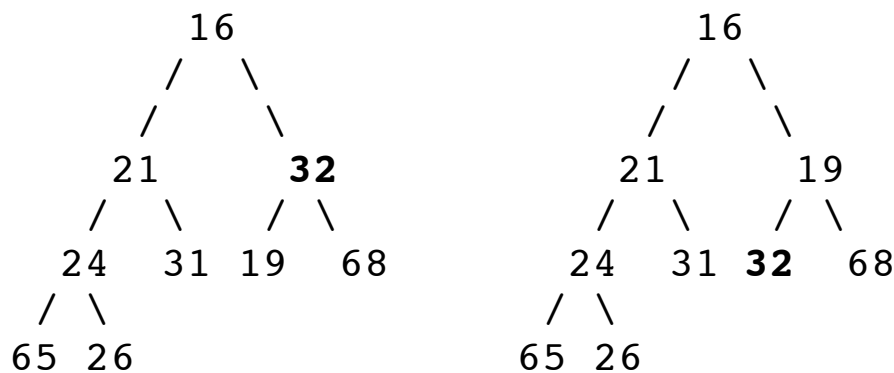
1) Fjern det



2) Tag gamle »sidste« element og placer i toppen



2) Så længe der er knas med underboerne, byt med den mindste af dem



Det var da også nemt  $\approx$  indlysende logaritmisk tid

## At oversætte dette til Java

I hvilken kontekst?

Som implementation af en *generisk* prioritetskø, **ergo**:

```
public interface PriorityQueue
{
    ....
    Comparable findMin();

    Comparable deleteMin();

    void insert( Comparable x ); //Forenklet i forhold til bogen

    boolean isEmpty();

    void makeEmpty();

    int size(); }

-----
```

```
public class BinaryHeap implements PriorityQueue
{ // metoder og konstruktorer...

    private static final int DEFAULT_CAPACITY = 100;

    private int currentSize; // Number of elements in heap

    private Comparable [ ] array; // The heap array
}
```

## Indsættelse i binær hob - i Java

```
public void insert( Comparable x ) //forenklet i forhold til bogen
{ if( currentSize + 1 == array.length )
    doubleArray( );
  int hole = ++currentSize;
  array[ 0 ] = x; // uden betydning, men forenkler løkken:

  for( ; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
    array[ hole ] = array[ hole / 2 ];
  array[ hole ] = x; }
```

Obs: Vi venter med at placere det nye element før vi ved hvor det skal ende.

## At fjerne det mindste element

```
public Comparable findMin( )
{ if( isEmpty( ) ) throw new UnderflowException( "Empty binary heap" );
  return array[ 1 ]; }

public Comparable deleteMin( )
{ Comparable minItem = findMin( );
  array[ 1 ] = array[ currentSize-- ];
  percolateDown( 1 );
  return minItem; }

private void percolateDown( int hole )
{ int child;
  Comparable tmp = array[ hole ];

  for( ; hole * 2 <= currentSize; hole = child )
  { child = hole * 2;
    // find underbo med mindste værdi:
    if( child != currentSize &&
        array[ child + 1 ].compareTo( array[ child ] ) < 0 )
      child++;
    // hvis knas med den underbo, simuler et bytning
    if( array[ child ].compareTo( tmp ) < 0 )
      array[ hole ] = array[ child ];
    else
      break; }
  array[ hole ] = tmp; }
```

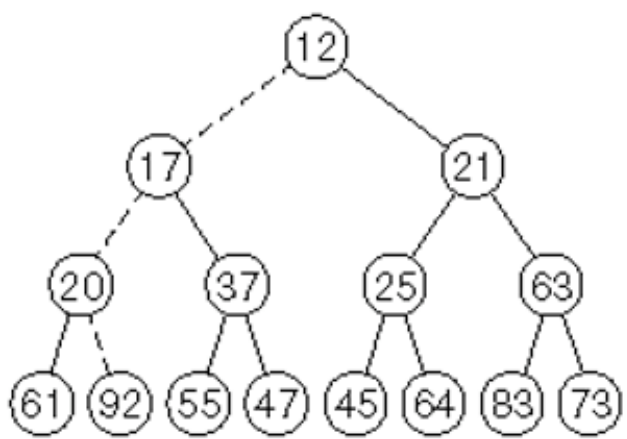
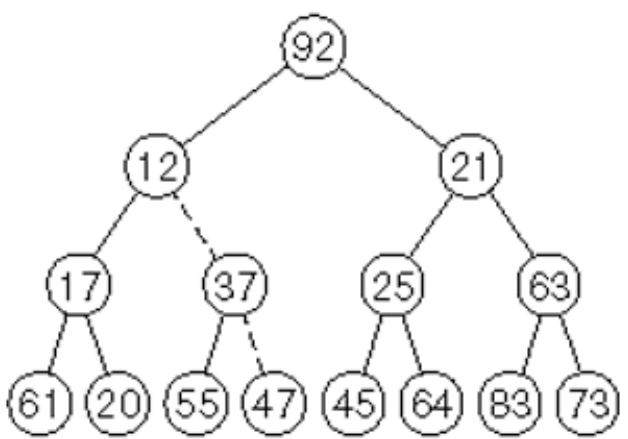
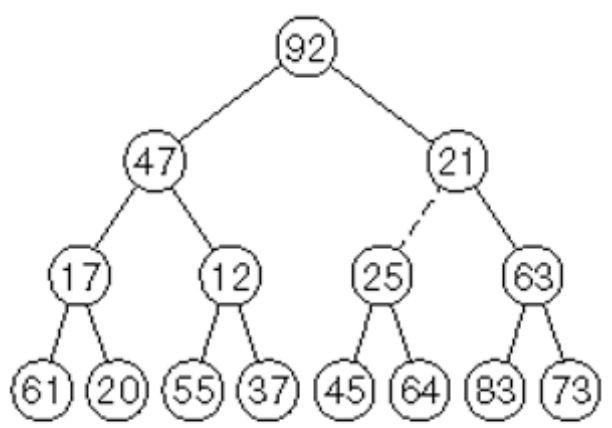
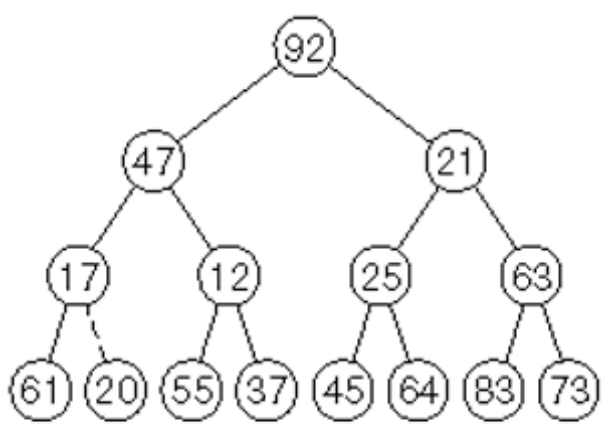
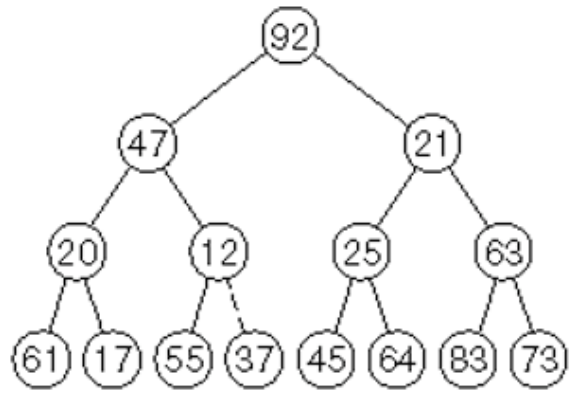
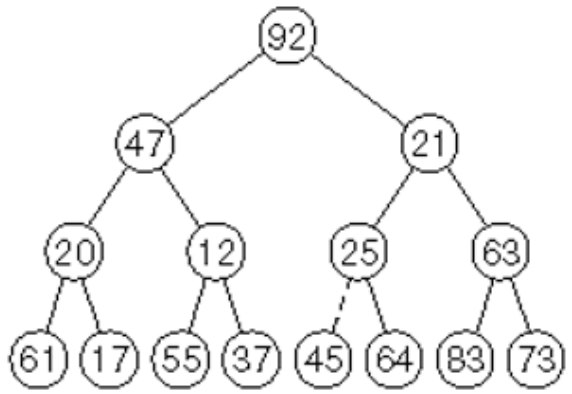
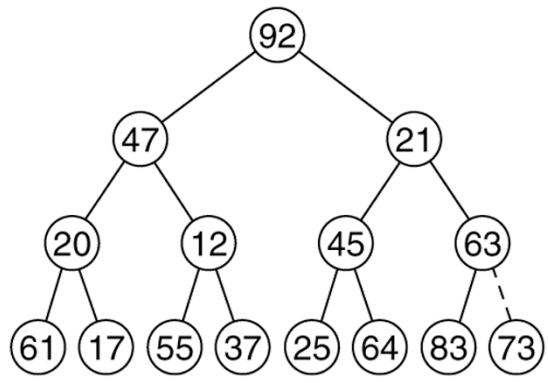
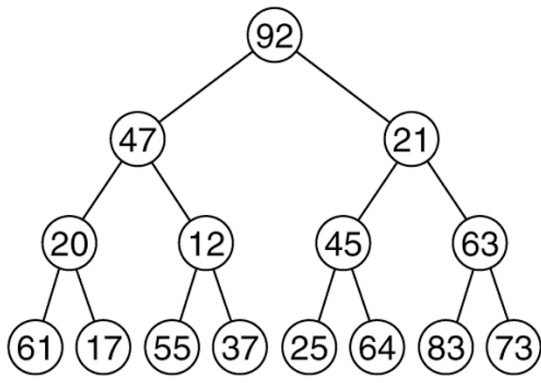
OBS: Det nedsivende element placeres først i array når vi ved, hvor det skal ende!

## At skabe en hob ud fra et samling elementer:

```
public BinaryHeap( Comparable [ ] items )
{   currentSize = items.length;
    array = new Comparable[ items.length + 1 ];

    for( int i = 0; i < items.length; i++ ) array[ i + 1 ] = items[ i ];
    buildHeap( );   }

private void buildHeap( )
{   for( int i = currentSize / 2; i > 0; i-- )
    percolateDown( i ); }
```



(a)

(b)

**En god matematiker kan vise dette liniært i antal knuder!**

# Heapsort $\approx$ garanteret $O(n \log n) \sim$ gennemsnit og værste fald

Quicksort gennemsnitlig  $O(n \log n)$

— — — værste fald  $O(n^2)$ , men i praksis hurtigere end Heapsort

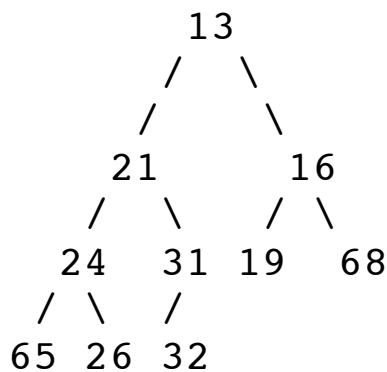
En version som sorterer i omvendt rækkefølge og undgår celle 0:

```
public static void heapsort( Comparable [ ] a )
{ BinaryHeap b = new BinaryHeap(a);
  for( int i = a.length - 1; i > 1; i-- )
    a[i] = b.deleteMin(); }
```

a:

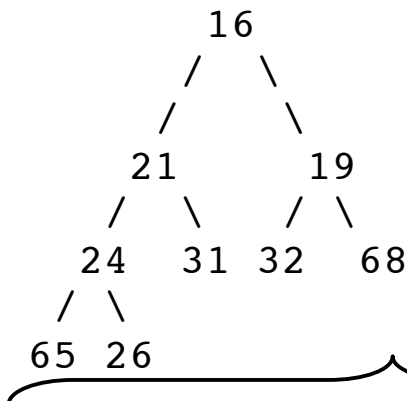
32	19	24	65	21	16	13	31	26	68
----	----	----	----	----	----	----	----	----	----

b umiddelbart efter new:



-	13	21	16	24	31	19	68	65	26	32
---	----	----	----	----	----	----	----	----	----	----

b umiddelbart efter første iteration:



-	16	21	19	24	31	32	68	65	26	-
---	----	----	----	----	----	----	----	----	----	---

## Rigtig sortering med heapsort:

- Benyt oprindelig array
- Juster overknode/underknode beregninger så de arbejder med roden i celle 0
- I stedet for en »minimumsheap« benyt en »maximumsheap«
- Indret nedsivningsmetoden efter det
  - test  $>$  og ikke  $<$
  - lidt + og  $- 1...$

## Eksempel på anvendelse af prioritetskø: »discrete-event simulation«

*Tid* simuleret ved eksplicit tal

*Begivenhed*  $\approx$

- finder sted på et bestemt *tidspunkt*;
- kan ændre tilstandsvariable
- kan generere nye, fremtidig begivenheder

*Elementerne* i køen:

Begivenheder som vil ske i den nærmeste fremtid

*Algoritme*

Initialiser variable;

BinaryHeap begivenhedsKø = new BinaryHeap;

begivenhedsKø.insert( new Begivenhed(noget-som-starter-det-hele));

while(!begivenhedsKø.empty() && !vi-gider-ikke-mere)

{ Begivenhed B = begivenhedsKø.deleteMin();

//udfør B:

if( B er xxxx) { .... }

else if(B er yyyy) {...}

....

else {...} }

**Eksempel: Posthus**

Tilstandsvariable: Antal kunder i kø, antal kasser, antal ledige kasser

Begivenheder: Ny kunde ankommer

Antal kunder i kø tælles én op;

vha tilfældighedsgenerator genereres næste

kundeankomst (lidt ude i fremtiden)

Kundeekspedition starter

Antal-ledige kasser tælles én ned;

vha tilfældighedsgenerator genereres tidspunkt

for begivenhed at ekspedition afslutter

Kundeekspedition afslutter .....

## Eksempel: Simulering af jernbanedrift

Jernbanenettet opdeles i diskrete delstrækninger svarende til f.eks. 200m skinne.

Begivenheder:

Tog X kører ind på strækning Y med fart Z kl. T

Skiftespor skiftes sådan og sådan kl. T

Tog X passerer skiftespor Y kl. T

Tog X standser på station Y kl. T

Tog X starter fra station Y i retning Z kl. T

Hvad kan undersøges ved simulering:

Sikkerhed:            Katastrofe defineres hvis to tog befinder sig på samme delstrækning

Robusthed for driftsforstyrrelser:

Hvis et givet tog går i stå i Valby 10 min i morgemyldretiden, hvordan forplanter det sig til resten af togdriften?

## Eksempel fra bogen: En samling modems som studerende ringer op til for at komme på nettet

- Studerende ringer op tilfældigt
- Hvis der er ledigt modem, får de et; ellers dutes optaget

Input til simulering:

- Antal modems ialt
- Sandsynlighedsfordeling for opkaldsforsøg
- Sandsynlighedsfordeling for hvor lang tid opkaldene varer
- Hvor lang tid simuleringen kører

Eksempel på udskrift:

User 0 dials in at time 0 and connects for 1 minute  
User 0 hangs up at time 1  
User 1 dials in at time 1 and connects for 5 minutes  
User 2 dials in at time 2 and connects for 4 minutes  
User 3 dials in at time 3 and connects for 11 minutes  
User 4 dials in at time 4 but gets busy signal  
User 5 dials in at time 5 but gets busy signal  
User 6 dials in at time 6 but gets busy signal  
User 1 hangs up at time 6  
User 2 hangs up at time 6  
User 7 dials in at time 7 and connects for 8 minutes  
User 8 dials in at time 8 and connects for 6 minutes  
....

Modellen forenklet, ingen tilfældighed på opkaldshyppighed, ét hvert minut.

## At beskrive simuleringsklasse med events i Java:

```
public class ModemSim
{
    private static class Event implements Comparable
    { static final int DIAL_IN = 1;
      static final int HANG_UP = 2;

      public Event( ) { this( 0, 0, DIAL_IN );}

      public Event( int name, int tm, int type )
      { who = name; time = tm; what = type;}

      public int compareTo( Object rhs )
      { Event other = (Event) rhs;
        return time - other.time; }

      int who;      // the number of the user
      int time;     // when the event will occur
      int what;     // DIAL_IN or HANG_UP }

    private Random r;           // A random source
    private PriorityQueue eventSet; // Pending events
    private int freeModems;     // Number of modems unused
    private double avgCallLen;  // Length of a call
    private int freqOfCalls;    // Interval between calls

    // Konstruktør at initialisere tilstand

    public ModemSim( int modems, double avgLen, int callIntrvl )
    { eventSet = new BinaryHeap( );
      freeModems = modems;
      avgCallLen = avgLen;
      freqOfCalls = callIntrvl;
      r = new Random( );
      nextCall( freqOfCalls ); // Schedule first call
    }
}
```

**//En lille hjælper: Generel et opkald, sæt i kø op gør parat til næste**

```
private int userNum = 0;
private int nextCallTime = 0;

private void nextCall( int delta )
{ Event ev = new Event( userNum++, nextCallTime, Event.DIAL_IN );
  eventSet.insert( ev );
  nextCallTime += delta; }
```

**// og den der gør arbejdet:**

```
public void runSim( long stoppingTime )
{ Event e = null;
  int howLong;
  while( !eventSet.isEmpty( ) )
  { e = (Event) eventSet.deleteMin( );
    if( e.time > stoppingTime ) break;
    if( e.what == Event.HANG_UP ) // HANG_UP
    { freeModems++;
      System.out.println("User "+e.who+" hangs up at time "+e.time);}
    else // DIAL_IN
    { System.out.print("User "+e.who+" dials in at time "+e.time+" " );
      if( freeModems > 0 )
      { freeModems--;
        howLong = r.nextPoisson( avgCallLen );
        System.out.println("and connects for "+howLong+" minutes");
        e.time += howLong;
        e.what = Event.HANG_UP;
        eventSet.insert( e ); }
      else
        System.out.println( "but gets busy signal" );

      nextCall( freqOfCalls ); }}}
```

```
public static void main( String [ ] args )
{ ModemSim s = new ModemSim( 3, 5.0, 1 ); s.runSim( 20 ); }
```

## Opsummering for idag:

- Prioritetskø implementerer ved »binary heap«
- Elegant måde at pakke træer ned i arrays — undgå alle de referencer
- Effektiv tilgang: op-ned »\*« og »/« med 2; højre-venstre  $\pm 1$

## Anvendelser

- Yet-another-sorteringsalgoritme
- Simulering: Kø  $\approx$  fremtidige begivenheder som venter på at blive udført
- Alt hvad der handler om jobafvikling: Printjobs, processer i et operativsystem, ...

## Slutbemærkninger om Java-delen af kurset

- Nu har vi været rundt om et udvalg af datastrukturer og algoritmer; dækkende for de vigtigste arter
- Hvordan det set ud i et objektorienteret, specifik Java
- Hvordan pakker af ting man skal bruge igen-og-igen struktureres i Javas API
- En idé om hvordan det er at skrive store systemer i Java.

## Vi slutter af med et par gange om noget helt andet

Prolog  $\approx$  et programmeringssprog som aldeles ikke er objektorienteret!

- Kompakt og langt større udtrykskraft
- Laboratoriesprog mere end til udvikling af programmer til uovervåget drift