PrioritetsKøer

I. Adgang til elementer i en samling

gjennom Position

gjennom nøkkel

II. Totale ordninger og nøkler

III. PrioritetsKø ADT

IV. Implementasjon med Sequence ADT

V. Implementasjon med Heap DS

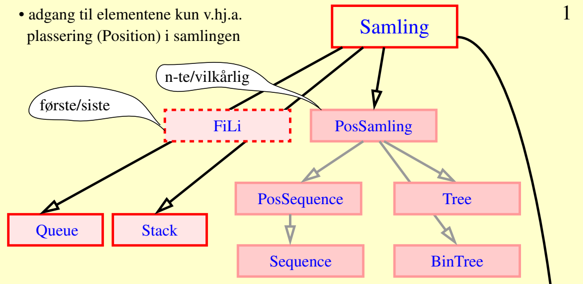
Linket Struktur

Sequence (Array)

Kap. 6 (kursorisk: 6.3.4; unntatt 6.4)

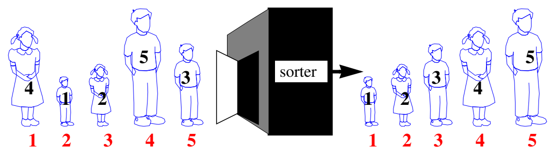
- adgang til elementene kun v.hj.a. plassering (Position) i samlingen

1



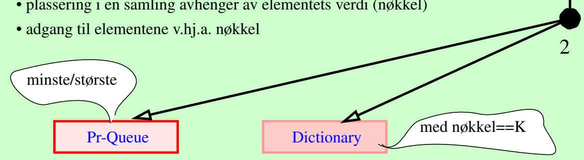
- struktur er uavhengig av (abstrakte) objekters interne struktur og egenskaper

... men f.eks. Sortering er en operasjon
1) på strukturer med rekkefølgen (total ordning) på Posisjoner, og hvor
2) alle elementene er ordnet (mht. en nøkkel-verdi)



- plassering i en samling avhenger av elementets verdi (nøkkel)
- adgang til elementene v.hj.a. nøkkel

2



Nøkler og Ordninger

Gitt en (**vilkårlig**) mengde ***S*** :

1. en **binær relasjon *R*** på *S*, er en mengde av par (***s***,***p***) der ***s*** og ***p*** ε ***S***
   (man skriver ofte (*s*,*p*) ε *R* som *R*(*s*,*p*), og hvis *R* er ≤, skriver vi $s \le p$)
2. en relasjon ≤ er en **total ordning** (**TO**) på ***S*** hviss den er:
   - **refleksiv** : $s \le s$ — for alle *s* ε *S*
   - **transitiv** : hvis $s \le p$ & $p \le q$ så $s \le q$ — for alle *s*,*p*,*q* ε *S*
   - **antisymmetrisk** : hvis $s \le p$ & $p \le s$ så $s = p$ — for alle *s*,*p* ε *S*
   (for to vilkårlige elementene $s \neq p$ ε *S* vil enten $p < s$ eller $s < p$)

TO *S* : heltallene — ***R***(***x***,***y***) : $x \le y$
S : et (latinsk, norsk, russisk) alfabet — ***R***(***x***,***y***) : ***x*** *kommer-før* ***y***
S : alle strenger (ord) over et gitt alfabet ***R*** : leksiskografisk ordning

NB! Samme mengde kan ordnes totalt på forskjellige måter, f.eks.

S : heltallene — ***R***(***x***,***y***) : ***x*** >= ***y***

ikke TO
S : heltallene — ***R***(***x***,***y***) : ***x*** < ***y***
S : pers. i 6-te rad — ***R***(***x***,***y***) : ***x*** *sitter-til-venstre-for* ***y***
* — ***R'***(***x***,***y***) : ***x*** *sitter-til-venstre-for-eller-der-hvor* ***y***
S : mennesker — ***R***(***x***,***y***) : ***x*** *yngre-enn* ***y***, ***x*** *ikke-yngre-enn* ***y***

Nøkkel (for en mengde ***E***) er en funksjon ***key*** : $E \to S$, der

I. ***S* er en (vilkårlig!) totalt ordnet mengde (med en ≤)**

II. ***key* er injektiv, dvs. slik at : hvis *e* ≠ *f* så *key*(*e*) ≠ *key*(*f*)**
(to forskjellige elementene fra *E* har forskjellige nøkkel-verdier)

tenk på *key*(*e*) som en egenskap/et attributt til *e*

	E	→	***S***	: ***key***
	stasborgere	→	personnumre (heltall,)	: ***key***(***e***) = ***e***'s persnr
*	pers i 6-te rad	→	plasser, til-venstre-eller-likt	: ***key***(***e***) = ***e***'s sitteplass
?	mennesker	→	heltall,	: ***key***(***e***) = ***e***'s alder

Ofte, oppfyller ikke nøkler II: flere elementer fra ***E*** kan ha samme nøkkel-verdi.

Objekter med nøkler

```
class E {
    int alder;
    int pnr;
    Object adr;
... }
```

key1: E →	heltall, ≤	e.alder;
key2: E →	heltall, ≤	e.pnr;
key3: E →	Object, ?	e.adr;

e= new E(21,333,"Oslo")

- Nøkkel for et Objekt e kan være et vilkårlig Objekt k : (e,k)
- For det meste (om ikke alltid) bruker vi (totalt) ordnede nøkler
- Ideen av TO kan uttrykkes abstrakt v.hj.a.

Designmønster (pattern)

```
public interface Comparator {
    boolean lt(Object a, Object b);
    boolean leq(Object a, Object b);
    boolean gt(Object a, Object b);
    boolean geq(Object a, Object b);
    boolean eq(Object a, Object b);
    ... }
```

- Algoritmer skal bruke bare dette mønsteret som parameter
- Dette mønstret må implementeres i hvert tilfelle – for å fange den spesifikke TO'en : avhengig av de aktuelle Objektene og slik applikasjonen krever

/** **generisk int-Comp** : sammenligner Integer objekter **/

```
class intComp implements Comparator {
  public boolean lt(Object a, Object b) { return
    (((Integer)a).intValue() < ((Integer)a).intValue()); }
  public boolean leq(Object a, Object b) { return
    (((Integer)a).intValue() <= ((Integer)a).intValue()); }
...}
```

```
boolean yngre_enn(E e, E g) { return
   new intComp().lt(new Integer(e.alder), new Integer(g.alder)) ; }
```

Spesifikke Comparator

implementerer både nøkkel-funksjonen og sammenlikning

- Algoritmer skal bruke bare Comparator mønsteret som parameter
- Dette mønstret må implementeres i hvert tilfelle – for å fange den spesifikke TO'en : avhengig av de aktuelle Objektene og slik applikasjonen krever

F.eks. en sorterings metode programmeres som

```
Object[] sort(Object[] A, Comparator C) { ... }
```

utelukkende ved å bruke Comparator-operasjoner.

Dersom vi nå skal sortere array med objekter av

```
class Eks { int alder; int pnr; Object adr; ... }
```

mht. alder, pnr, etc. lager vi en Comparator for hvert tilfelle

```
class aldComp implements Comparator { // antar Objekter Eks
  boolean lt(Object a, Object b) {
    return (((Eks)a).alder < ((Eks)b).alder); }
...}
```

```
class pnrComp implements Comparator { // antar Objekter Eks
  boolean lt(Object a, Object b) {
    return (((Eks)a).pnr < ((Eks)b).pnr); }
...}
```

```
class adrComp implements Comparator { // antar Objekter Eks
  boolean lt(Object a, Object b) {
    return (((Eks)a).adr ?? ((Eks)b).adr); }
...}
```

```
sort(A, new aldComp());

sort(A, new pnrComp());

sort(A, new adrComp());
```

NB!
en Comparator skal brukes **kun med de spesifikke Objektene** den er designet for

Vi skal bruke...

```
public class Par { // et par Objekt-nøkkel
  private Object el, key;
  public Par(Object k, Object e) { setElem(e); setKey(k); }
  public Object key() { return key;}
  public Object elem() { return el;}
  public void setKey(Object k) {key=k;}
  public void setElem(Object e) {el=e;}
}
```

```
class ParComp implements Comparator {
  public boolean lt(Object a, Object b) {
    Par aa = (Par)a; Par bb = (Par)b;
    return aa.key() ??? bb.key() ; // passende sammenlikning
  }
... }
```


PrioritetsKø

en samling som gir oss **adgang til elementet med den minste nøkkelen** (i en aktuell total ordning)

```
public interface Samling
{  int size();
   boolean isEmpty();
   Enumeration elements();
}
```

```
public interface PrQueue
           extends Samling {
/**  sett inn et nytt element
   * @param e objektet som skal innsettes
   * @param k noekkel objekt */
   void insert(Object e, Object k);
/**  return - og fjern - minste
   * @return elementet med minste noekkel */
   Object remMin();
/**  return - uten aa fjerne - minste
   * @return elementet med minste noekkel */
   Object min();
/**  return minste noekkel-verdi
   * @return minste noekkel */
   Object minKey();
}
```

NB!!

- Ofte lagrer vi **Objekter der nøkkel er et attributt** – i så fall må vi lagre de med et kall `PQ.insert(o, o.key())`
- Men her kan vi også samle **Objekter uten noen nøkkel** – en nøkkel tilordnes ved innsetting: `PQ.insert(o, new Key(2))`

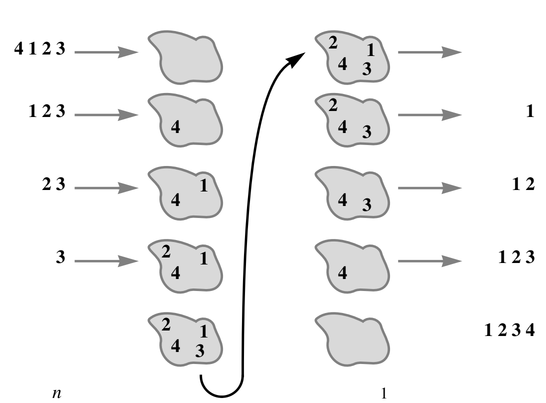
PrQ-Sortering mønster

Sorter en gitt Sequence S mht. en bestemt ordning (Comparator C)
– bruker en PrQueue P (med samme Comparator C)

trykkfeil i boken s.207

```
void sort(Sequence S, PrQueue P)
  a)  while (! S.isEmpty())
          e = S.removeFirst()     fjern ett og ett element fra S     O(1)
          P.insert(e,e)           og sett dem inn i P
  b)  while (! P.isEmpty())
          e = P.remMin()          fjern ett og ett element fra P
          S.insertLast(e)         og sett dem inn på slutten av S    O(1)
```

a) b)

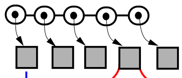


$$O\left(\sum_{k=1}^{n} P_k.insert(e)\right) + O\left(\sum_{k=n}^{1} P_k.remMin(\,)\right)$$

PrQueue >> Sequence

```
class PQSeq implements PrQueue {
  protected Comparator cp;
  protected Sequence sq=new SeqDL();
  public PQSeq(Comparator c)
     { cp= c; }
... }
```

```
ev.
  protected Sequence sq ;
  public PQSeq(Comparator c, Sequence s)
     { cp= c; sq= s; }
```

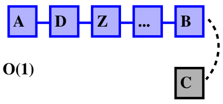


```
public class Par {
  private Object el, key;
  public Par(Object k, Object e) {
    setElem(e); setKey(k); }
  public Object key() { return key;}
  public Object elem() { return el;}
  public void setKey(Object k) {key=k;}
  public void setElem(Object e) {el=e;}
}
```

```
class CoPar implements Comparator {
  public boolean lt(Object a, Object b) {
    Par aa = (Par)a; Par bb = (Par)b;
    return aa.key() < bb.key() ;
  }
... }
```

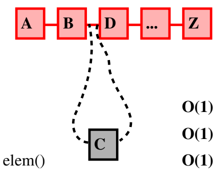

PrQueue >> Sequence >> (LL,DL,Array)

I. DATA INVARIANT : INGEN – USORTERT




```
insert(o,k) : sq.insertLast(new Par(k,o))          O(1)
  minKey() : finn ...
             Position p = sq.first();
             Object o = p.elem();
             while (p != sq.last() )
                p = sq.after(p);
                if (cp.lt(p.elem(), o)
                   o = p.elem();
             return ((Par)o).elem().key();         Θ(n)
     min() : finn ...                              Θ(n)
  remMin() : finn og fjern ...                     Θ(n)
       ... : ...                                   O(1)
```

II. DATA INVARIANT : SORTERT – MINSTE STÅR FORAN




```
  minKey() : ((Par) sq.first().elem()) . key()                 O(1)
     min() : ((Par) sq.first().elem()) . elem()                O(1)
  remMin() : ((Par) (sq.removeFirst(.elem()) ) . elem()        O(1)
insert(o,k) : Par ny= new Par(k,o);
             if ( sq.isEmpty() ) sq.insertFirst(ny)
             else if ( cp.leq(ny, sq.first().elem()) ) sq.insertFirst(ny)
             else if ( cp.geq(ny, sq.last().elem()) ) sq.insertLast(ny)
             else   Position c = sq.first()
                    while (cp.gt(ny, c.elem() )  c= sq.after(c)
                    sq.insertBefore(c,ny)                      O(n)
       ... : ...                                               O(1)
```


Oppdater aldri objekter i en Samling!

fordi nøkkel-verdi avhenger, typisk, av Objektets attributter

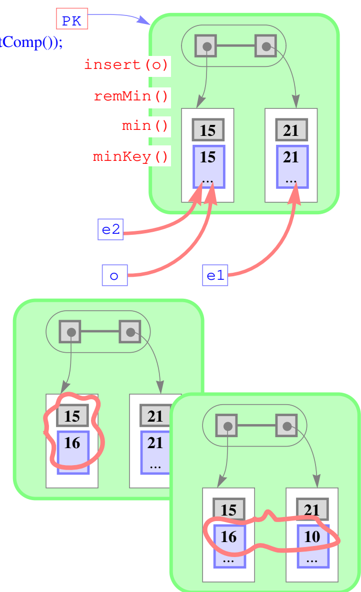
```
class El {
      private int alder;
      private int pnr;
      public Object key() { return new Integer(alder); }
      public void setAlder(int a) { alder=a; }
... }

PrQueue PK = new PQSeq(new intComp());
El e1= new El(21);
El e2= new El(15);
PK.insert(e1,e1.key());
PK.insert(e2,e2.key());

El o = (El) PK.min();

o.setAlder(16);

e1.setAlder(10);
```



Et godt - metodologisk - råd !!!

I. IMPLEMENETER DATA INVARIANT

```
/**   DI:  sequence av Par (Key,Object)
 *         sortert mht. Comparator cp
 *         Comparator cp sammenlikner Par mht. Key-verdi
 */
class PQSeq implements PrQueue {
  protected Comparator cp;
  protected Sequence sq=new SeqDL();
  public PQSeq(Comparator c) { cp= c; }

  private boolean DI() {                       // sjekk om sq er sortert
     boolean b= true;
     Position c= sq.first(), s= sq.last();
     while (c != s && b )
     {   if ( cp.gt( (Par)c.elem(), (Par)sq.after(c).elem() ) )  b= false;
         else c= sq.after(c); }
     return b;
  }
  public Object min() throws DIException {
     if (! DI()) throw DIException("DI feil: Ikke sortert");
     else return ((Par)sq.first().elem()).elem()
... }
```

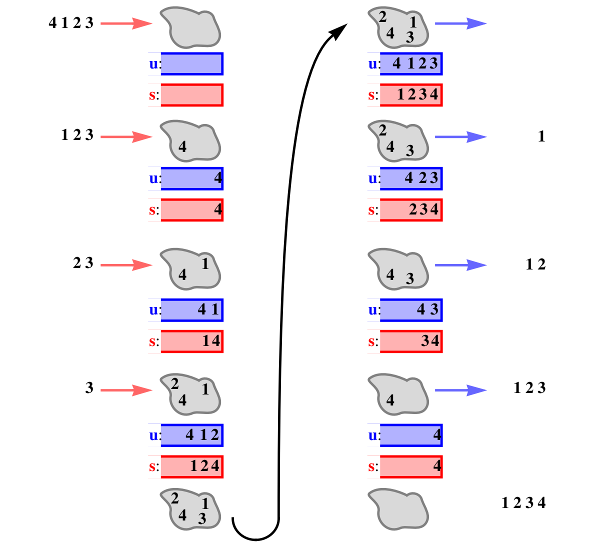
II. FOR Å OPPDATERE ET OBJEKT I EN SAMLING :

El o = (El) PK.removeMin();	**1. fjern fra Samlingen**
o.setAlder(16);	**2. oppdater**
PK.insert(o, o.key());	**3. sett inn i Samlingen**

Dette kan virke noe kostbart (spesielt når vi oppdaterer attributter som ikke påvirker nøkkel-verdier) men :

1. Hvilke attributter påvirker nøkkel kan variere og være uklart
2. Kostnaden øker vanligvis ikke algoritmers kompleksistet
3. Resulterende kode er betydelig sikrere

Insertion /Selection Sort



$O(\sum_{k=1}^{n} P_k.insert(e))$ + $O(\sum_{k=n}^{1} P_k.remMin())$

4 1 2 3 —n→ u: 4 1 2 3 —$\frac{n^2+n}{2}$→ 1 2 3 4 $O(n^2)$ $\Omega(n^2)$

4 1 2 3 —$\frac{n^2+n}{2}$→ s: 1 2 3 4 —n→ 1 2 3 4 $O(n^2)$ $\Omega(n)$

III. Heap DS

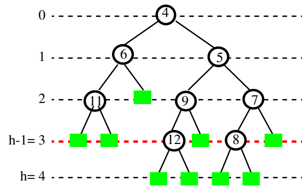
er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler) som tilfredstiller **DATA INVARIANT**:

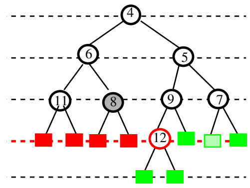
Heap-Ordering (“relasjonell”)

1. for enhver node v (unntatt roten): key(v) ≥ key(parent(v))

Komplett Binært Tre (“strukturell”)

2. T med høyde h:
   - 2.a) alle nivåene $i=0,1,...,h-1$ har maks. no. noder $=2^i$
   - 2.b) på nivå $h-1$ alle interne noder er “til venstre for” alle eksterne



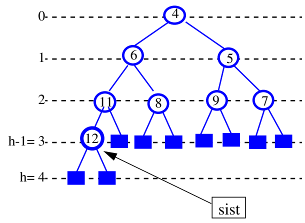


Heap med har høyde: ***n* (interne) noder** $h = \lceil \log(n+1) \rceil$

$1 + 2 + ... + 2^{h-2} + 1 = 2^{h-1} \le n$

$n \le 1 + 2 + ... + 2^{h-2} + 2^{h-1} = 2^h - 1$

$h \le \log(n) + 1$ & $\log(n+1) \le h$



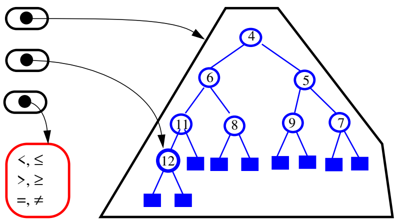
III. Implementasjon av PrQueue Heap

public class HPK implements PrQueue {

private BinTree Heap

private Position sist

private Comparator cp



private boolean **DI**()
{ traverser Heap og sjekk at enhver node v har
key(v) ley(parent(v))
Komplett BinTree er vanskeligere }

public Object **min**()
{ return ((Par) Heap.root().elem()) . elem() ; }

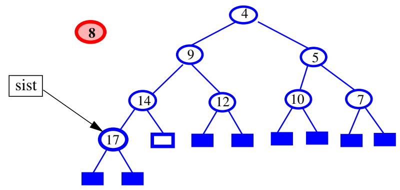
public Object **minKey**()
{ return ((Par) Heap.root().elem()) . key() ; }

public Object **remMin**()
{ ... }

public void **insert**(Object k, Object e)
{ ... }

...

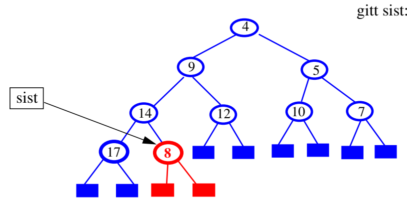
A. Insert i Heap



Bevar **KOMPLETT BINÆRT TRE INVARIANT**:

1. Finn innsettingsnode – “til høyre” for siste utvid bladet til en intern node og sett inn det nye elementet

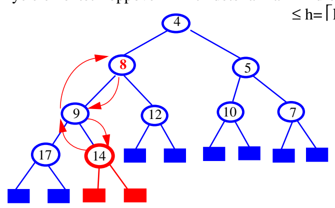
gitt sist:**O(1)** el. **O(log n)**



Gjenopprett **HEAP-ORDERING INVARIANT** :

2. Flytt det nye elementet “oppover” inntil dets far har mindre nøkkel

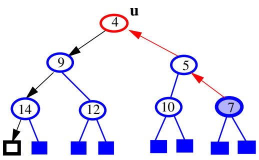
$\le$ h=$\lceil$log(n+1)$\rceil$ = **O(log n)**



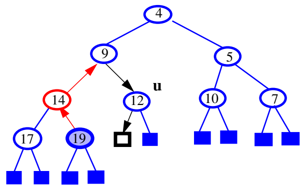
A.1. Finn innsettingsnode

1. Gitt **siste**, finn innsettingsnode **u**
   – avhengig av implementasjon av BinTree

- Sequence (array): **siste** = *n*; **u = siste** +1 **O(1)**
- BinTree interface (Linket Struktur)
  Avhengig av hvem **siste** er har vi tre tilfeller:
  a) *T.isEmpty()* : **u** = *T.root()*
  b) ytterste noden i nivå *h-1*



  c) en mellomnode i nivå *h-1*

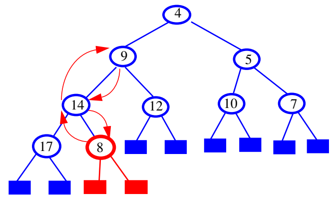



```
u = siste
while ( u != root() && u != leftChild(parent(u)) )
      u = parent(u)
if ( u != root) u = rightChild(parent(u))
while ( ! isExternal(u) ) u = leftChild(u)
return u                                              O(log n)
```


A.2. "Oppover bobling"

```
u = siste                                             A.1
while ( u != root() && u != leftChild(parent(u)) )
      u = parent(u)
if ( u != root) u = rightChild(parent(u))
while ( ! isExternal(u) ) u = leftChild(u)
return u                                              O(log n)
```

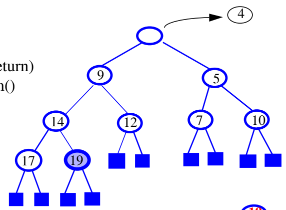
```
expandExternal(u) ;
u.setElem(ny) ;
```



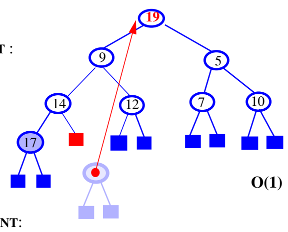

```
while ( u != root() && cp.lt( u.elem(), parent(u).elem() ) )      A.2
      swap(u , parent(u))
      u = parent(u);
                                                                  O(log n)
```


B. Fjerning min() fra Heap

1. hold root-Objektet (til return)
   ((Par)root().elem()).elem()



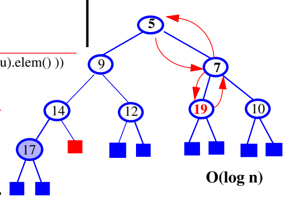
2. Bevar **KOMPLETT BINTREE INVARIANT** :
   – plasser sist.elem() i rot-posisjon
   – og fjern sist-posisjon
   sett inn et nytt blad
   (BinTree: removeAboveExt(leftChild(sist)))



O(1)

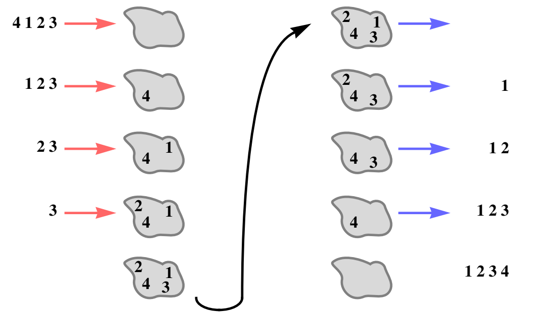
3. Gjenopprett **HEAP-ORDERING INVARIANT**:
   – flytt det nye rot-elementet "nedover" til en passende posisjon ("nedover bobling")

```
u = root();  done = false;
while ( ! done )
  if ( isExternal(leftChild(u)) && isExternal(rightChild(u)) )
        done = true
  else
  {   if ( isExternal(rightChild(u)) )
            n = leftChild(u)
      else if (cp.lt( leftChild(u).elem(), rightChild(u).elem() ))
            n = leftChild(u)
      else n = rightChild(u);
      if ( cp.gt( u.elem(), n.elem() ))
             swap(u,n);  u = n
      else done= true
} }
```



O(log n)

PrioritetsKø Sortering



$$O\left(\sum_{k=1}^{n} P_k.insert(e)\right) + O\left(\sum_{k=n}^{1} P_k.remMin(\,)\right)$$

Selection Sort: $O(\mathbf{n}^2)$

4 1 2 3 —— n ——→ **u: 4 1 2 3** —— $\frac{n^2+n}{2}$ ——→ **1 2 3 4**

Insertion Sort: $O(\mathbf{n}^2)$

4 1 2 3 —— $\frac{n^2+n}{2}$ ——→ **s: 1 2 3 4** —— n ——→ **1 2 3 4**

Heap Sort: $O(\mathbf{n}\log(\mathbf{n}))$

4 1 2 3 —— $n\log(n)$ ——→ **h: .. ..** —— $n\log(n)$ ——→ **1 2 3 4**