PrioritetsKøer

I. Adgang til Elementer i en Samling

gjennom Position
gjennom nøkkel

II. Totale Ordninger og Nøkler

III. PrioritetsKø ADT

PrioritetsKø-Sortering

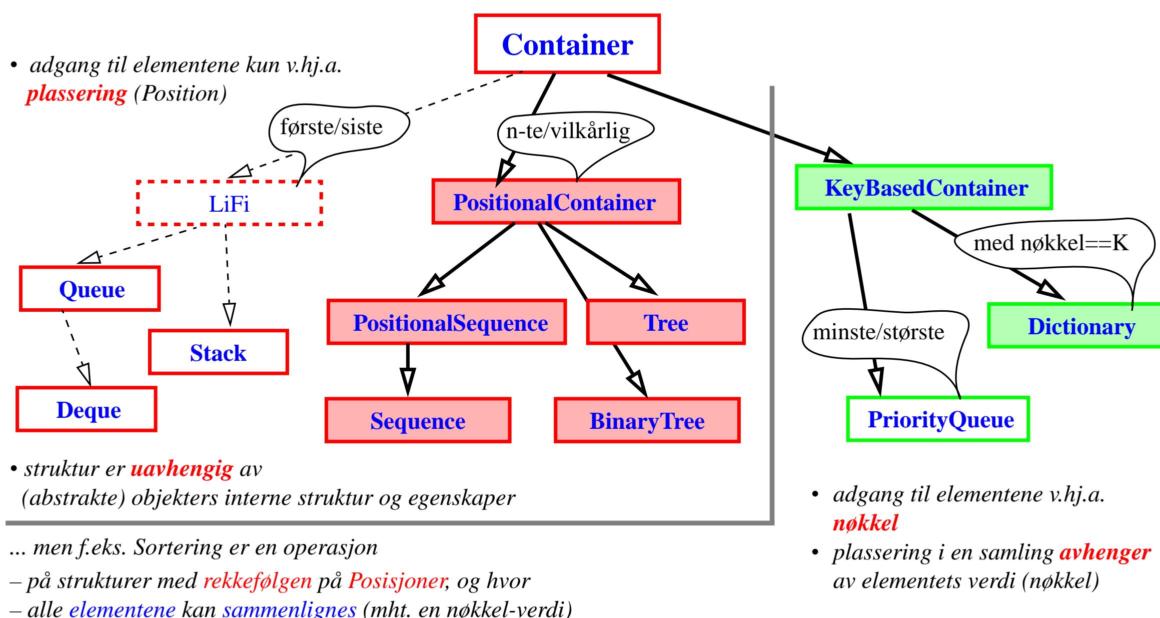
IV. Implementasjon med Sequence ADT

V. Implementasjon med Heap DS

implmentasjon av Heap operasjoner
BinaryTree / Array

VI. Locator designmønster (6.4)

Kap. 6 (kursorisk: 6.3.4)

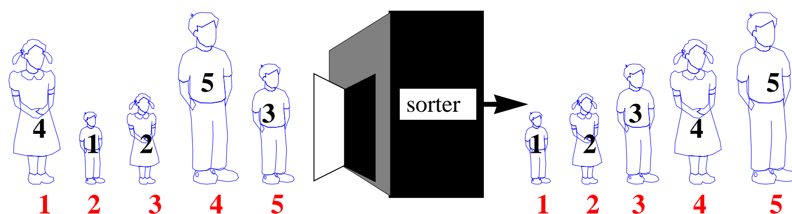


- *adgang til elementene kun v.hj.a.* ***plassering*** *(Position)*

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

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

... men f.eks. Sortering er en operasjon
– på strukturer med rekkefølgen på Posisjoner, og hvor
– alle elementene kan sammenlignes (mht. en nøkkel-verdi)



II. 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*), *evt. s* ≤ *p*)
2. en relasjon ***R*** er en **total ordning** (**TO**) på ***S*** hviss den er:
   - **refleksiv** : *R(s,s)* for alle *s* ε *S*
   - **transitiv** : for alle *s,p,q* ε *S* hvis *R(s,p)* & *R(p,q)* så *R(s,q)*
   - **antisymmetrisk** : for alle *s,p* ε *S* hvis *R(s,p)* & *R(p,q)* så *s = p*

   (*for to vilkårlige s* ≠ *p* ε *S vil enten R(p,s) eller R(s,p)*)

S :		***R***(***x***,***y***)
– heltallene		***x*** ***y***
– et alfabet		***x*** *kommer-ikke-etter* ***y***
– alle strenger (ord) over et gitt alfabet		leksiskografisk ordning
Samme mengde kan TOs på forskjellige måter		
– heltallene		***x*** ***y***, (|*x*||*y*| *el.* |*x*|=|*y*| *og x*<*0*)
– heltallene		***x*** < ***y***
– pers. i 6-te rad		***x*** *sitter-til-venstre-for* ***y***
	*	***x*** *sitter-til-venstre-for -eller-der-hvor* ***y***
– mennesker		***x*** *yngre-enn* ***y***
	*	***x*** *ikke-yngre-enn* ***y***

Nøkkel (for en mengde ***E***) er en funksjon
key : ***E*** → ***S***, der

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

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

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

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

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

Objekter med nøkler

```
class Pers {
      int alder;        key1: E → heltall, ≤
      int pnr;          key2: E → heltall, ≤
      String adr;       key3: E → String, compareTo
... }
```

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

e.alder;	(*ikke entydig*)
e.pnr;	(*entydig*)
e.adr;	(*ikke entydig*)

- *Nøkkel for et Objekt e kan være et vilkårlig Objekt k :* ***Item(k,e)***
  – en KeyBasedContainer vil samle Item's
- *abstrakt TO uttrykkes ved* *interface* ***Comparator***
  – som vil parametrisere enhver implementasjon av KeyBasedContainer
- *spesifikk TO er en* *class X implements* ***Comparator***

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

```
class intKeyComp implements Comparator { // antar Item ( key:Integer, ?)
  boolean isLessThan(Object a, Object b) {  Item aa = (Item)a; Item bb= (Item)b;
     return (( (Integer) aa.key() ) . intValue() < ( (Integer) bb.key() ) . intValue(); } ... }

class stringKeyComp implements Comparator { // antar Item ( key:String, ?)
  boolean isLessThan(Object a, Object b) {  Item aa = (Item)a; Item bb= (Item)b;
     return ( (String) aa.key() ) . compareTo ( (String) bb.key() )  < 0 ; } ... }
```

PriorityQueue ADT

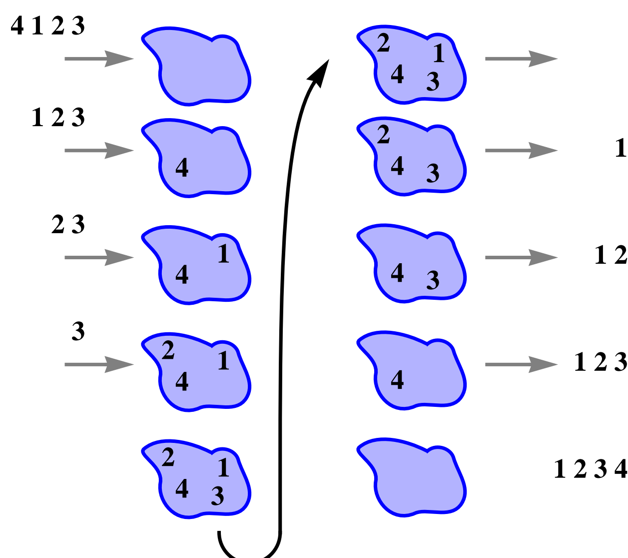
```
/** adgang gjennom minste nøkkel – i en aktuell ordning bestemt av implementasjon */
public interface PriorityQueue extends KeyBasedContainer {
/** sett inn et nytt element med angitt nøkkel
 *  @param k nøkkel til det nye elementet
 *  @param e elementet som skal settes inn */
    void insertItem(Object k, Object e);
/** returner Objektet med minste nøkkel
 *  @return   Objektet med minste nøkkel
 *  @exception EmptyContainerException  hvis isEmpty() */
    Object minElement();
/** returner minste nøkkel i køen
 *  @return   minste nøkkel
 *  @exception EmptyContainerException  hvis isEmpty() */
    Object minKey();
/** fjern og returnerer elementet med minste nøkkel
 *  @return   elementet med minste nøkkel
 *  @exception EmptyContainerException  hvis isEmpty() */
    Object removeMinElement();
/** returner lokator til minste elementet  */
    Locator min();          }
```

når e har et Object-atributt som nøkkel :
PQ . insertItem(e.key(), e)
men nøkkel kan bestemmes ved innsetting : PQ . insertItem(newKey(e), e)

PrioritetsKø-Sortering

```
void pqSort(Sequence S, PriorityQueue PQ)
// antar at PQ bruker Objektene som nøkler
 a) while (! S.isEmpty())
// fjern ett og ett element fra S = O(1)
      e = S.removeFirst()
// og sett dem inn i PQ
      PQ.insertItem(e,e)
 b) while (! PQ.isEmpty())
// fjern ett og ett element fra PQ
      e = PQ.removeMinElement()
// og sett dem inn på slutten av S = O(1)
      S.insertLast(e)
```

trykkfeil i boken s.207



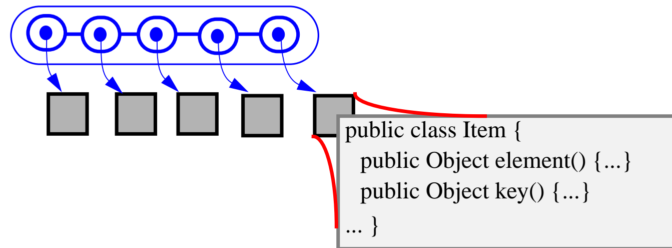
$$n + \sum_{k=1}^{n} P_k.insert(e) + \sum_{k=n}^{1} P_k.remMin(\,) + n$$

IV. Implementasjon av PriorityQueue – *med Sequence*

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en Item



II DATA STRUKTUR

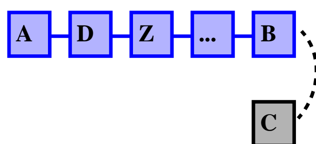
```
class PQSequence implements PriorityQueue {
    private Sequence S;
    private Comparator cp;
    /**  @param sq bør være tom sekvens
         @param c Comparator for sammenlikning av Item med passende key-objekter */
    public PQSeq(Sequence sq, Comparator c) {
       S = sq; cp = c; }
```

III DATA INVARIANT

- Item i S kan sammenlignes med cp
- S er usortert
- S er sortert

PriorityQueue – *med Sequence*

DATA INVARIANT: INGEN – USORTERT



void *insertItem*(o,k) **O**(1)

```
sq.insertLast(new Item(k,o))
```

Object *minKey*(): finn minste ... **Θ(n)**

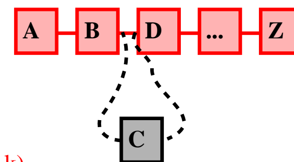
```
Position p = sq.first();
Object o = p.element();
while ( p != sq.last() ) {
   p = sq.after(p);
   if (cp.LT(p.element(), o)
       o = p.element();   }
return ((Item)o).key();
```

Object *minElement*() : finn ... **Θ(n)**

Object *removeMin*() : finn og fjern ... **Θ(n)**

- *Invarianten skal velges avhengig av forvented hyppighet av operasjoner*
- *Kompleksitet er relativ til implementasjon av Sequence (LL/DL/Array)*

DATA INVARIANT: SORTERT STIGENDE



void *insertItem*(o,k) **O(n)**

```
Item ny= new Item(k,o);
if ( sq.isEmpty() ) sq.insertFirst(ny)
else if ( cp.isLessThanOrEqual(ny, sq.first().element()) )
   sq.insertFirst(ny)
else if ( cp.isGreaterThanOrEqual(ny, sq.last().element()) )
   sq.insertLast(ny)
else Position c = sq.first()
   while (cp.isGreaterThan(ny, c.element() )  c= sq.after(c)
   sq.insertBefore(c,ny)
```

Object *minKey*() **O(1)**

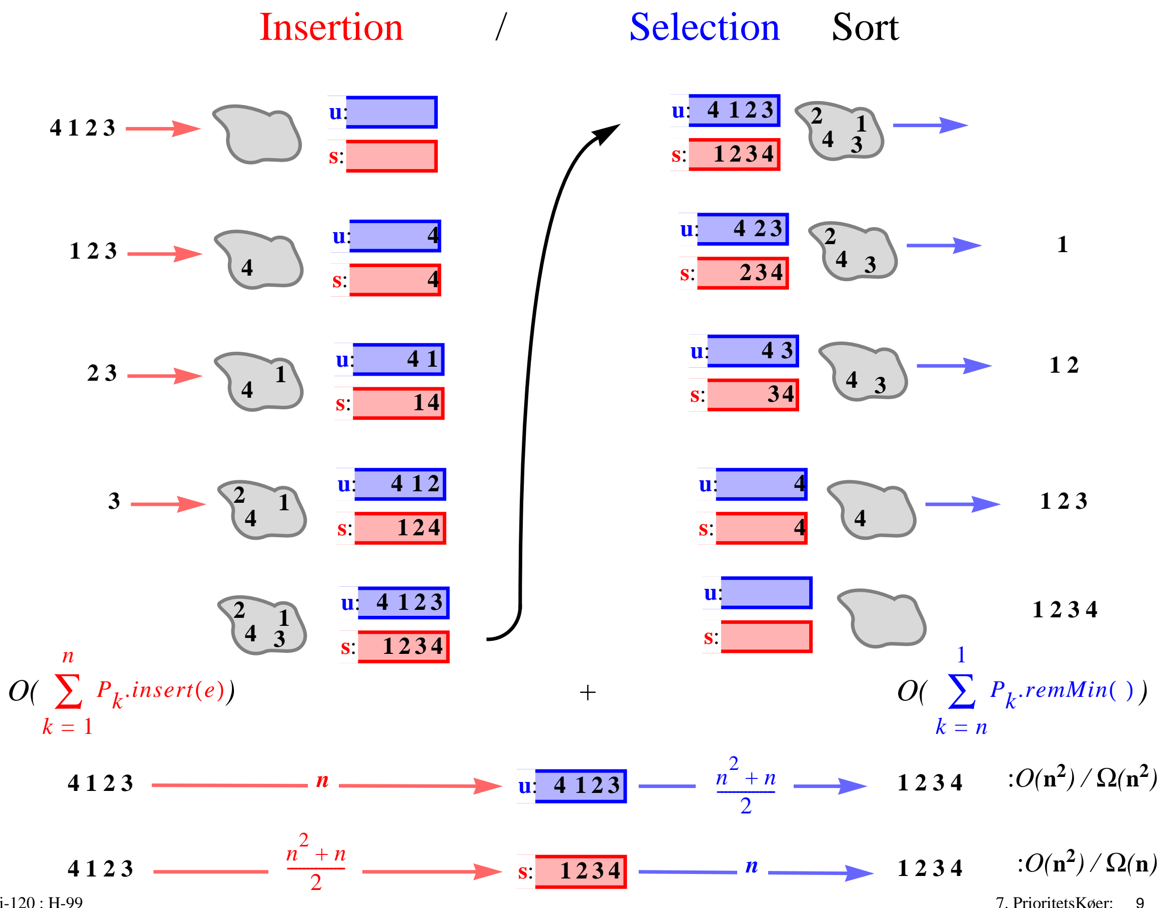
```
return ((Item) sq.first().element()) . key()
```

Object *minElement*() **O(1)**

```
return ((Item) sq.first().element()) . element()
```

Object *removeMin*() **O(1)**

```
return ( (Item)sq.remove(sq.first()) ) . element()
```



V. Heap DataStruktur

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

1. Heap-Ordering ("relasjonell")

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

2. Komplett Binært Tre ("strukturell")

T med høyde h:

2.a) alle nivåene i=0,1,...,h-1 har maks. no. noder $=2^i$

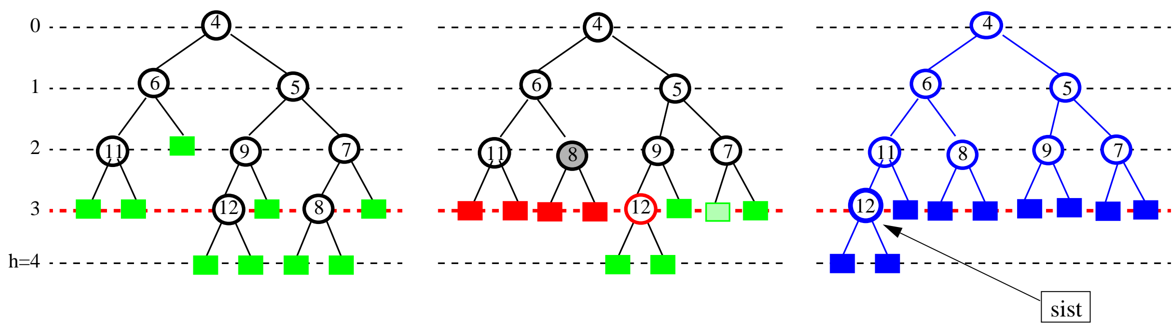
2.b) på nivå h-1 alle interne noder er "til venstre for" alle eksterne

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

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

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

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



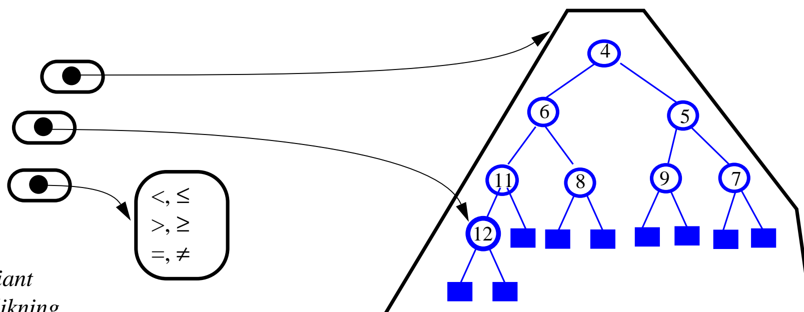
Implementasjon av PriorityQueue *med Heap*

DATASTRUKTUR

private **BinaryTree** heap

private **Position** sist

private **Comparator** cp

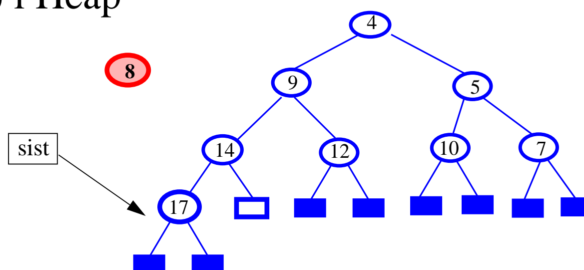


DATAINVARIANT

heap oppfyller Heap-Invariant
med hensyn til cp-sammenlikning
og sist er "siste" posisjon i heap: ==null hviss heap.isEmpty()

```
public PQhp(Comparator c) {
    cp = c ; heap = new BinaryTreeIMP(); }  // sist==null hviss heap.isEmpty()
private boolean DI() {
    traverser Heap og sjekk at enhver node v har key(v)  key(parent(v)) :
        cp . isGreaterThanOrEqualTo ( (Item)v.element() , (Item) heap.parent(v).element() )
    Komplett BinTree er vanskeligere }
public Object minElement() throws EmptyContainerException {
    if (heap.isEmpty()) throw new EmptyContainerException("");
    return ((Item) Heap.root().element()) . element() ; }
public Object minKey() throws EmptyContainerException {
    if (heap.isEmpty()) throw new EmptyContainerException("");
    return ((Item) Heap.root().element()) . key() ; }
```

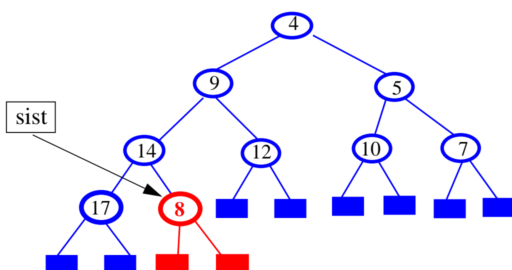

A. *insertItem*(o,k) i Heap



1. ***Bevar*** **KOMPLETT BINÆRT TRE INVARIANT**:

Finn innsettingsnode – "til høyre" for siste
– utvid bladet til en intern node og sett inn
det nye elementet

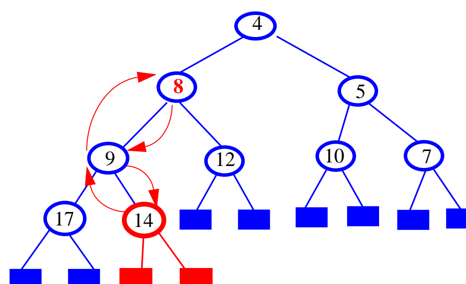
gitt sist:**O(1)** el. **O(log n)**



2. ***Gjenopprett*** **HEAP-ORDERING INVARIANT** :

Flytt det nye elementet "opp" inntil dets
forelder har mindre nøkkel

$\leq h = \lceil \log(n+1) \rceil = \mathbf{O(\log n)}$



A.1. Finn innsettingsnode

1. *Gitt* **sist**, *finn innsettingsnode* **u**
   – *avhengig av implementasjon av* **BinaryTree**

- **Sequence** (array):
  sist = *n*; **u** = **sist** +1 **O(1)**

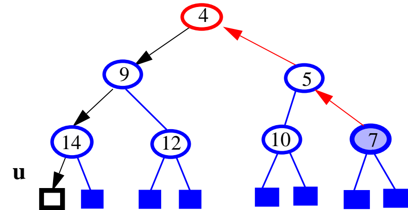
```
u = sist
while ( u != root()
        && u != leftChild(parent(u)) )
    u = parent(u)
if ( u != root)
    u = rightChild(parent(u))
while ( ! isExternal(u) )
    u = leftChild(u)
return u                          O(log n)
```

- **BinaryTree** interface
  (vilkårlig implementasjon)

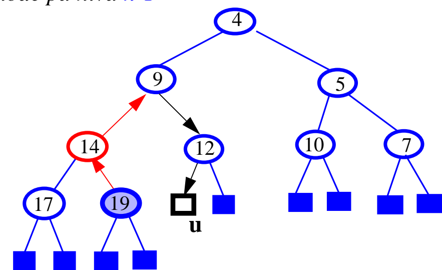
  Avhengig av hvem **sist** *er har vi tre tilfeller:*

a) *T.isEmpty()* : **u** = *T.root()*

b) *ytterste noden på nivå h-1*



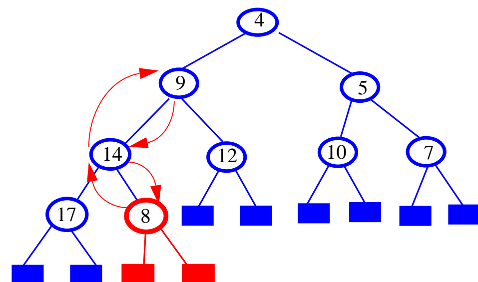
c) *en mellomnode på nivå h-1*



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)
```

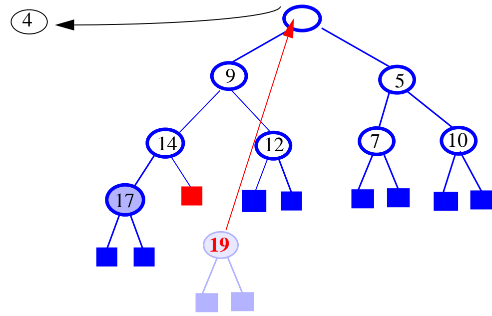
```
expandExternal(u) ;
u.setElement(ny) ;
```




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

B. *removeMinElement*() fra Heap

1. *hold root-Objektet (til return)*
((Item)root().element()).element()



2. ***Bevar* KOMPLETT BINARYTREE INVARIANT** :

– *plasser sist.element() i rot-posisjon*

– *og fjern sist-posisjon = sett inn et nytt blad*

(BinTree: removeAboveExternal(leftChild(sist)))

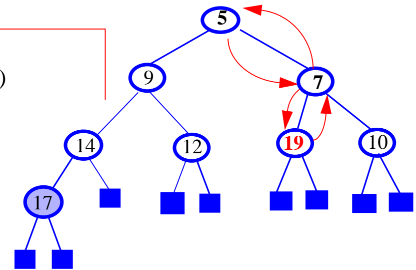
– ***oppdater sist*** **O(**1 + **log n)**

3. ***Gjenopprett* HEAP-ORDERING INVARIANT**:

– *flytt det nye rot-elementet "ned" til passende posisjon ("nedover bobling")*

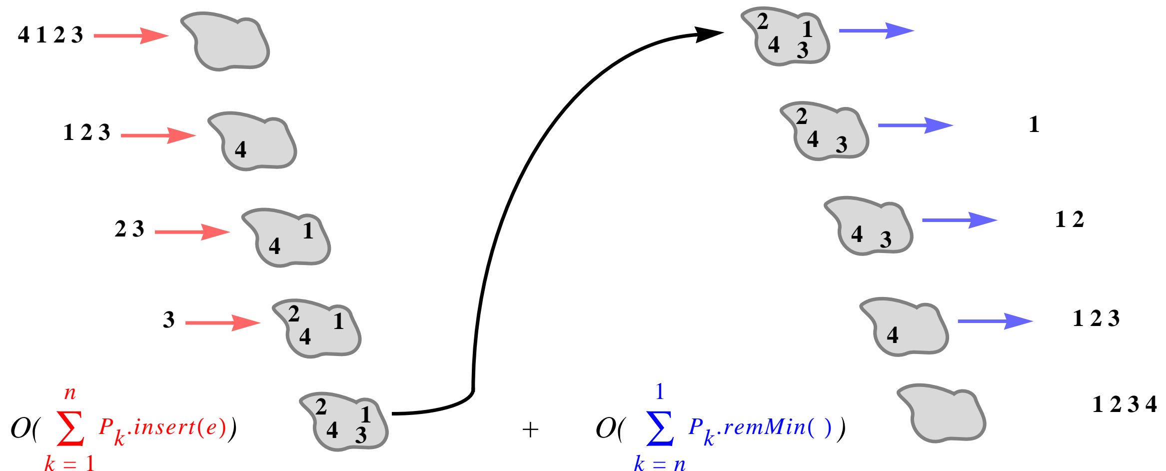
```
u = root();  done = false;
while ( ! done ) {
  if ( isExternal(leftChild(u)) && isExternal(rightChild(u)) )  done = true
  else
  {  if ( isExternal(rightChild(u)) )  neste = leftChild(u)
     else if (cp.isLessThan( leftChild(u).element(), rightChild(u).element() ))
          neste = leftChild(u)
     else neste = rightChild(u);

     if ( cp.isGreaterThan( u.element(), neste.element() ))
          swap(u,neste);  u = neste;
     else done= true
} }
```



O(log n)

PrioritetsKø Sortering



Selection Sort	**4 1 2 3**	n →	**u: 4 1 2 3**	$\frac{n^2+n}{2}$ → **1 2 3 4**	$:O(\mathbf{n^2})$
Insertion Sort	**4 1 2 3**	$\frac{n^2+n}{2}$ →	**s: 1 2 3 4**	n → **1 2 3 4**	$:O(\mathbf{n^2})$
Heap Sort	**4 1 2 3**	$n \log(n)$ →	**h: ....**	$n \log(n)$ → **1 2 3 4**	$:O(\mathbf{n \log(n)})$

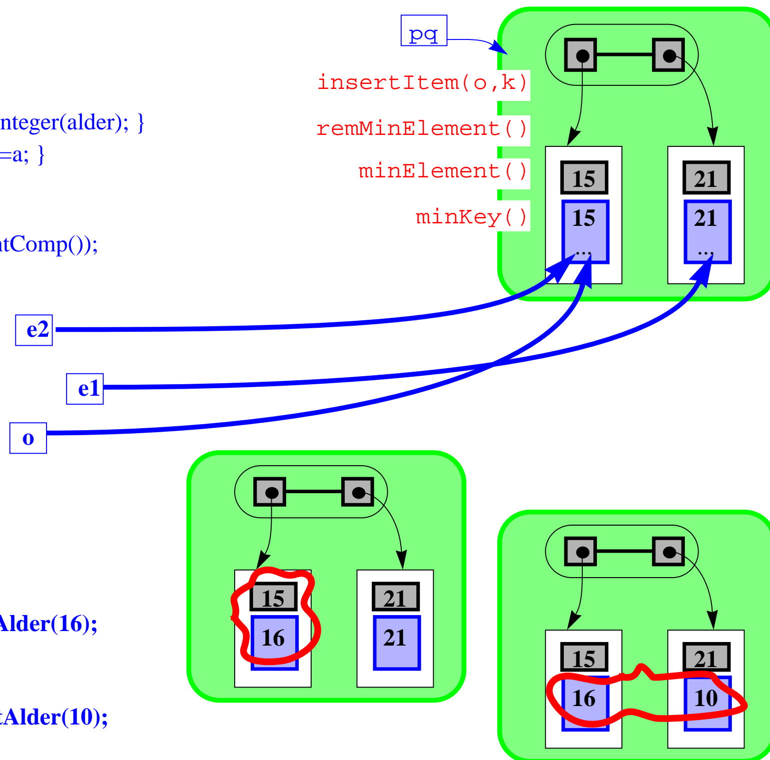
Oppdater aldri objekter i en Samling!

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; }
... }

PriorityQueue pq = new PQSeq(new intComp());
El e1= new El(21);
El e2= new El(15);
pq.insertItem(e2, e2.key());
pq.insertItem(e1, e1.key());

El o = (El) pq.minElement();
```



o . setAlder(16);

e1 . setAlder(10);

Et godt - metodologisk - råd !!!

1. Implemeneter Data Invariant

I eksemplet over: – *for hver* Item i *har vi at:* i.element().key() == i.key()
– *sekvens er sortert*

2. Unngå oppdatering av objekter i Nøkkel-Baserte-Samlinger

Må du oppdatere slike objekter:

3. Fjern før oppdatering:

```
El o = (El) pq.removeMinElement();
o.setAlder(16);
pq.insertItem(o, o.key());
```

1. fjern fra Samlingen
2. oppdater
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*

4. Bruk mer sofistikert Grensesnitt ...

Locator designmønster

tilsvarer Position for samlinger der Objekter lokaliseres v.hj.a. nøkkel (KeyBasedContainer)

```
public interface Locator {
    Object element()
    Object key()
    Container container()
    boolean isContained()
}
```

```
public interface KeyBasedContainer extends Container {
  /**  Inserts a Locator into this Container. */
     void insert(Locator)
  /**  Inserts a <key, element> pair into this Container. */
     Locator insert(Object k, Object o)
  /**  Enumeration of all the Locators within this Container. */
     Enumeration locators()
  /**  Enumeration of all of the keys of all the locators in the Container. */
     Enumeration keys()
  /**  When you need a locator that can be inserted into this KeyBasedContainer but don't want to insert it quite yet. */
     Locator makeLocator(Object k, Object o)
  /**  Removes an element from this Container. */
     void remove(Locator)
     Object replaceElement(Locator l, Object o)
  /**  Changes the mapping of a Locator's element to a new key.
       @return  the old key to which the Locator's element was mapped */
     Object replaceKey(Locator, Object)
}
```

implementasjon vil kreve en god del manipulering med datastrukturen

Oppsummering

Typer av Samling

- *LiFi (Stack, Queue)*
- *PositionalContainer*
- *KeyBasedContainer*

Nøkkel:

- *Totale Ordninger*
- *objekter med nøkler (Item)*

Prioritetskø ADT :	***implementasjon*** *og*	***sorteringsmønster***
	usortert sekvens	*seleksjonsort*
	sortert sekvens	*innstikksort*
	heap	*heapsort*

Heap datastruktur:

- *Binært tre + datainvariant*
- *innsetting / fjerning ! opprettholdelse av datainvarianten*

Locator designmønster:

- *overtar rollen av Position for KeyBasedContainer*
- *nyttig når objekter/nøkler må oppdateres mens de er i en samling*