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

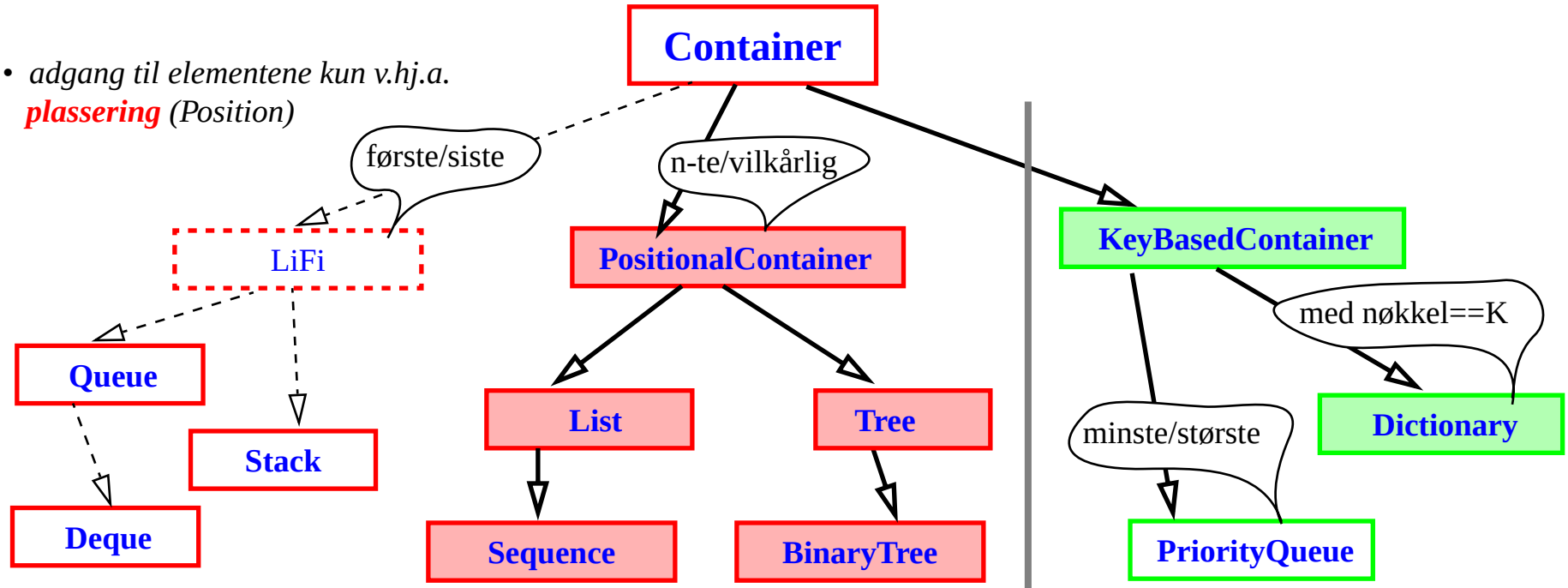
implmentasjon av Heap operasjoner

BinaryTree / Array

VI. Locator designmønster (7.4)

Kap. 7 (kursorisk: 7.4.2, 7.3.5)

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



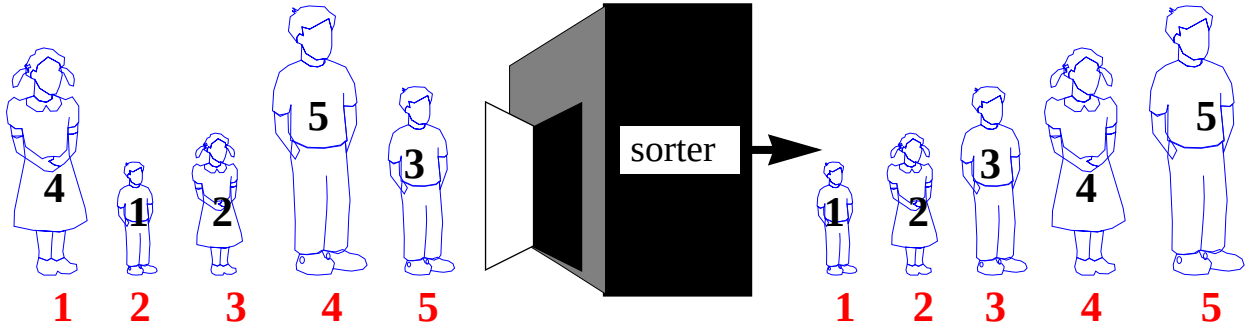
- *struktur er* ***uavhengig*** *av (abstrakte) objekters interne struktur og egenskaper*

... men f.eks. Sortering er en operasjon

– på strukturer med rekkefølgen på Posisjoner, og hvor

– alle elementene kan sammenlignes (mht. en ***nøkkel-verdi****)*

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



II. Nøkler og Ordninger

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

1. en **binær relasjon *R*** på ***K***, er en mengde av par (***s***,***p***) der ***s*** og ***p*** ε ***K***
   (*man skriver ofte* (*s*,*p*) ε *R som R*(*s*,*p*), *evt. s <= p*)
2. en relasjon ***R*** er en **total ordning** (**TO**) på ***K*** hviss den er:
   - **refleksiv** : ***R(s,s)*** for alle *s* ε ***K***
   - **transitiv** : for alle *s*,*p*,*q* ε ***K***
     hvis ***R(s,p)*** **&** ***R(p,q)*** så ***R(s,q)***
   - **antisymmetrisk** : for alle *s*,*p* ε ***K***
     hvis ***R(s,p)*** **&** ***R(p,q)*** så ***s = p***
   - **total** : for alle *s*,*p* ε ***K***
     enten ***R(s,p)*** enten ***R(p,q)***

mengde K :	***ordning 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***
– heltallene	***x*** < ***y***
	* ***x*** <= ***y***
– mennesker	***x*** *yngre-enn* ***y***
	* ***x*** *ikke-eldre-enn* ***y***

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

I. ***K* 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	→	***K***	:	***key(e)***
– personer	→	personnumre (heltall,<=)	:	***e***'s persnr
– *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**: Pers	→	heltall, <=	e.alder;	*(ikke entydig)*
int pnr;	**key2**: Pers	→	heltall, <=	e.pnr;	*(entydig)*
String adr;	**key3**: Pers	→	String, compareTo	e.adr;	*(ikke entydig)*

... }

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

- *Nøkkel for et Objekt e kan være et vilkårlig Objekt k :* ***Item(k,e)***
  – en KeyBasedContainer vil samle Item's

```
public interface Item { // et par Objekt-nøkkel
  public Object key() { return key;}
  public Object element() { return el;}
}
```

- *abstrakt TO uttrykkes ved* *interface* ***Comparator***
  – som vil parametrisere enhver implementasjon av KeyBasedContainer

- *spesifikk TO er en* *class X implements* ***Comparator***

```
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 ; } ... }

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(); } ... }
```

PriorityQueue ADT

```
/** adgang gjennom minste nøkkel – i en aktuell ordning bestemt av implementasjon */

public interface PriorityQueue extends KeyBasedContainer {    % size(), isEmpty(), ...
/** 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 removeMin();

/** 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**())

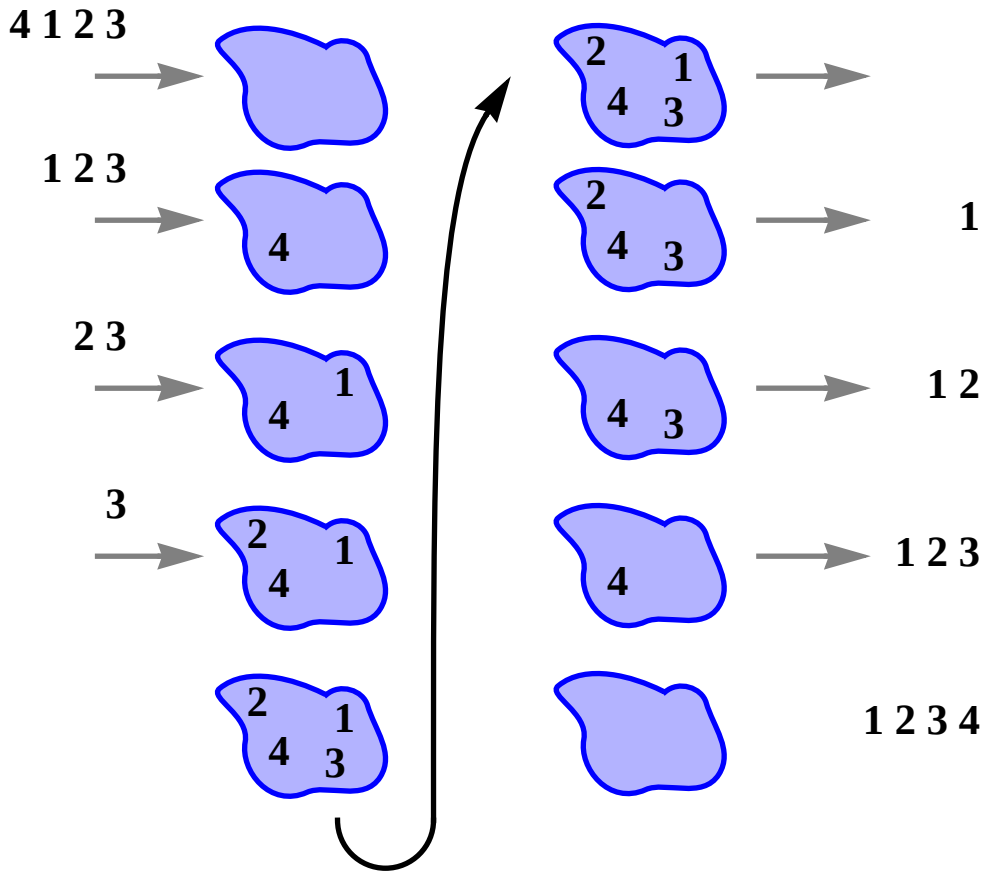
// fjern ett og ett element fra S = O(1)
e = **S.removeFirst**()

// og sett dem inn i PQ
PQ.insertItem(e,e)

b) while (! **PQ.isEmpty**())

// fjern ett og ett element fra PQ
e = **PQ.removeMin**()

// og sett dem inn på slutten av S = O(1)
S.insertLast(e)



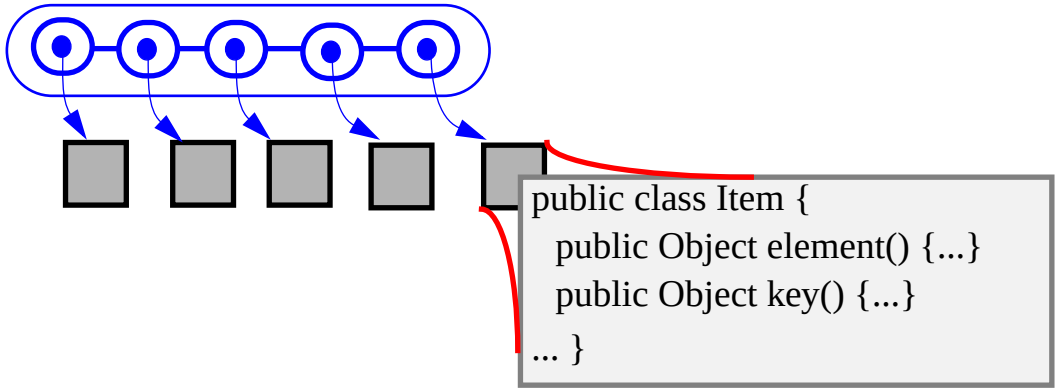
$$n + \sum_{k=1}^{n} P_k.insert(e) + \sum_{k=n}^{1} P_k.remMin() + n$$

IV. Implementasjon av PriorityQueue – *med Sequence*

I Data Representasjon

Sequence, der hver Position

lagrer en Item



II Data Struktur

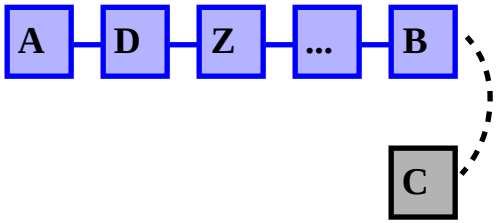
```
class PQSequence implements PriorityQueue {
    private Sequence S;
    private Comparator cp;
    /**  @param sq bør være tom sekvens
         @param c Comparator for sammenlikning av Item med passende key-objekter */
    public PQSeq(Sequence sq, Comparator c) {
       S = sq; cp = c; }
```

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

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

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

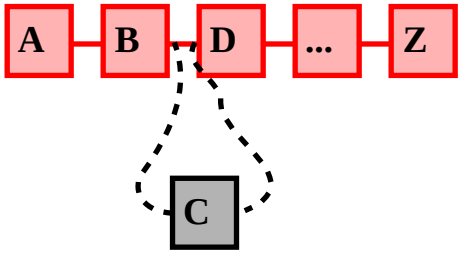
```
Position p = sq.first();
Object o = p.element();
while ( p != sq.last() )  {
    p = sq.after(p);
    if (cp.LT(p.element(), o)
        o = p.element();   }
return ((Item)o).key();
```

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*(k,o) **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()
```

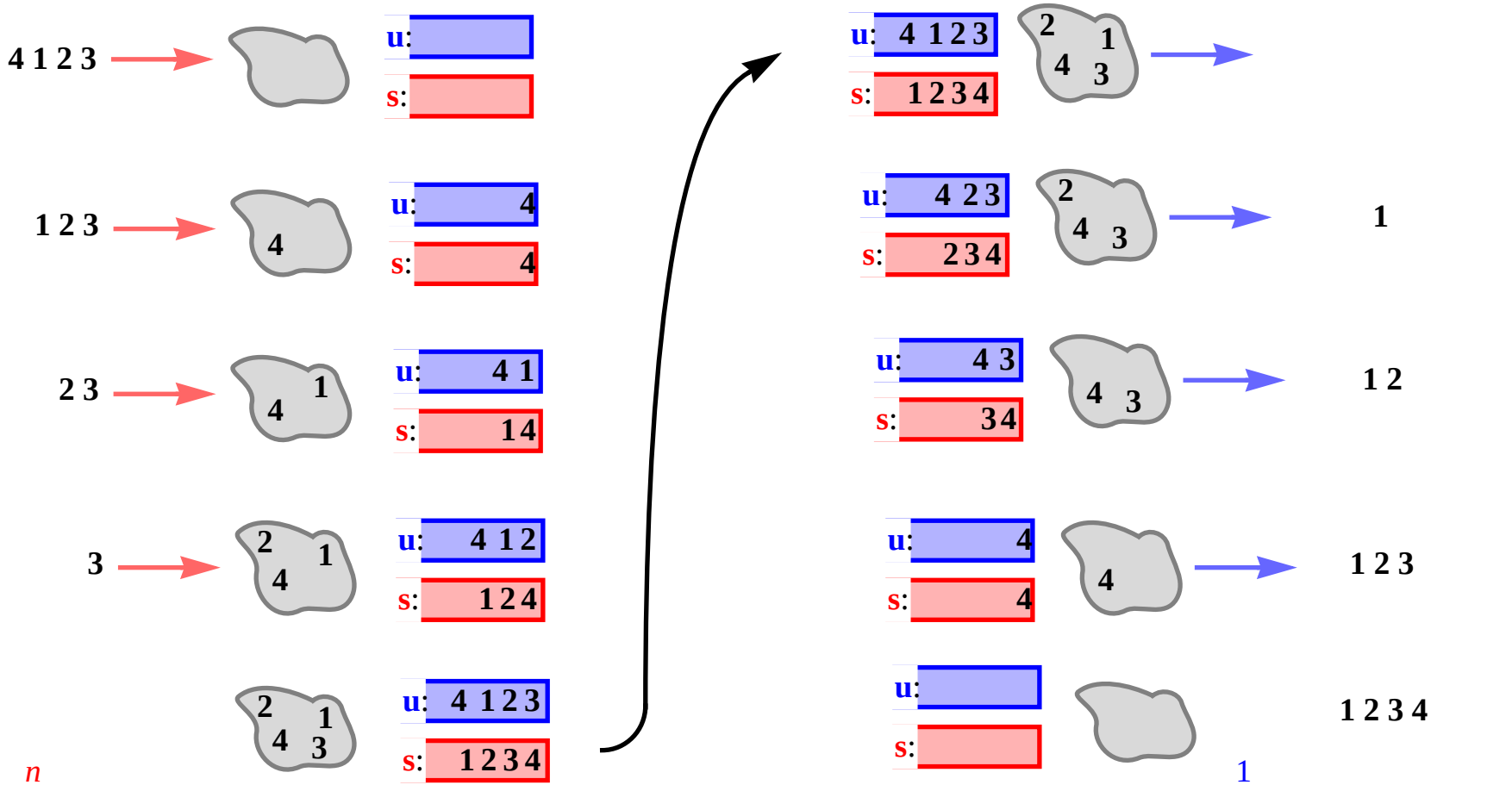
Object *minElement*() **O(1)**

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

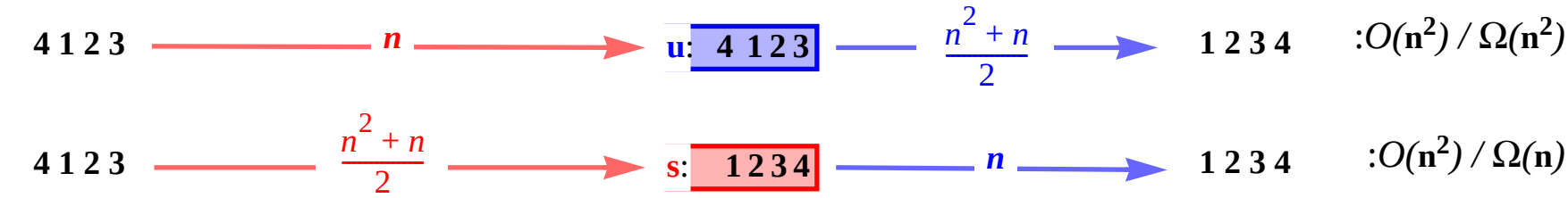
Object *removeMin*() **O(1)**

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

Insertion / Selection Sort



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



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 $=2^i$

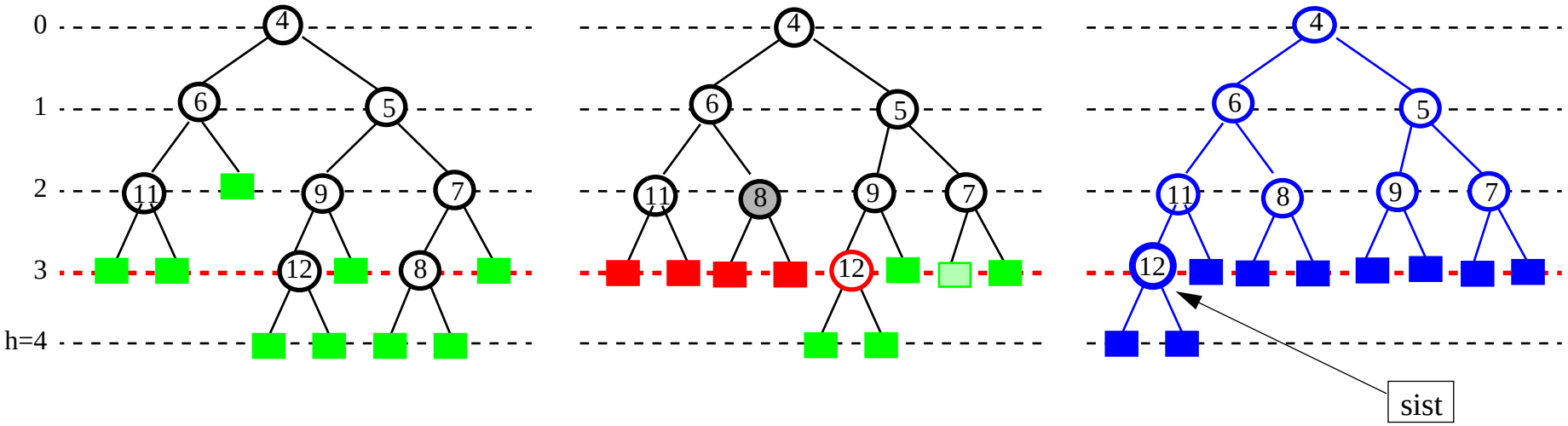
2.b) på nivå $h-1$ alle interne noder er ‘‘til venstre for’’ alle eksterne

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

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

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

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



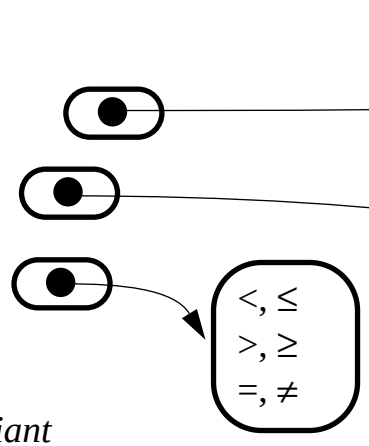
Implementasjon av PriorityQueue *med Heap*

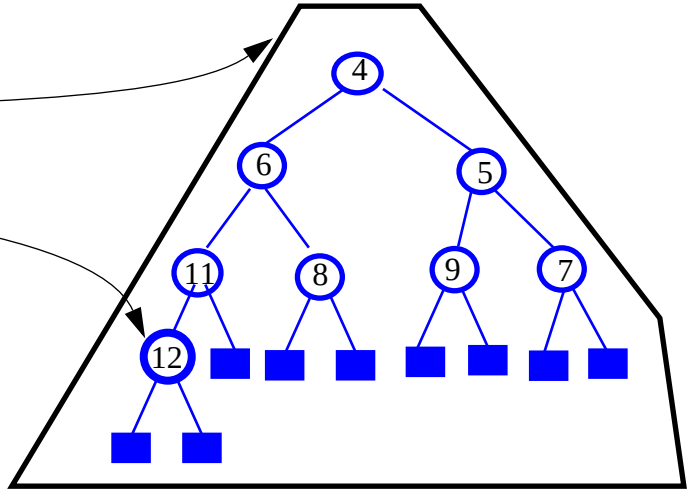
DATASTRUKTUR

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()

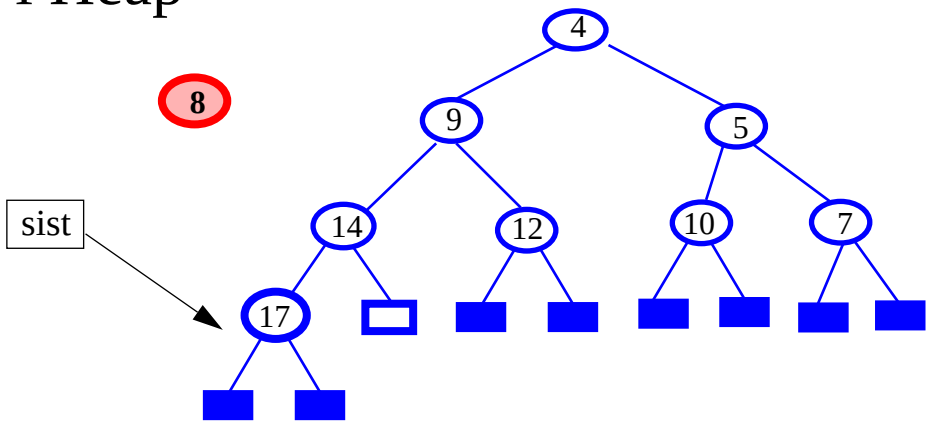
public ***PQhp***(**Comparator** c) {
cp = c ; heap = new BinaryTreeIMP(); } // sist==null hviss heap.isEmpty()

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()** ; }

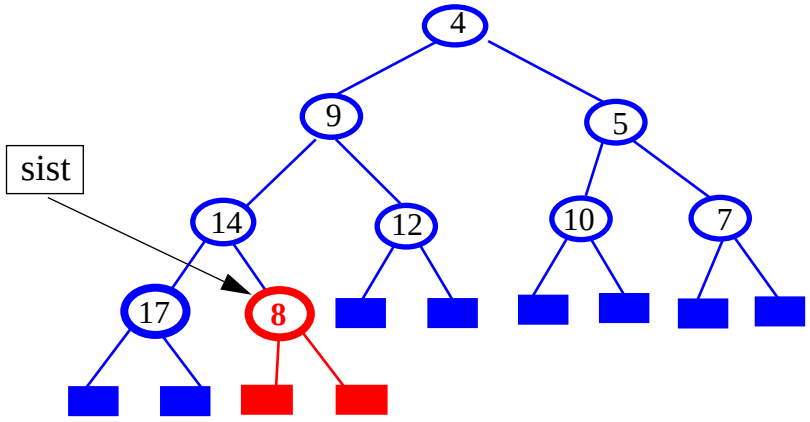
A. *insertItem*(k,o) i Heap



1. *Bevar* KOMPLETT BINÆRT TRE INVARIANT:

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

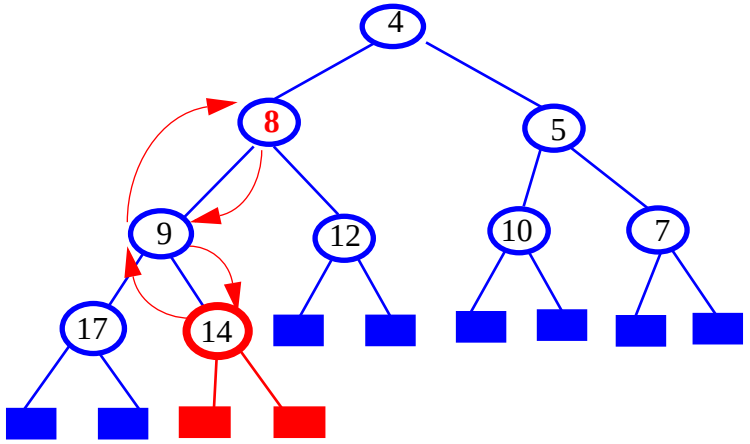
gitt sist:**O(1)** el. **O(log n)**



2. *Gjenopprett* HEAP-ORDERING INVARIANT :

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

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

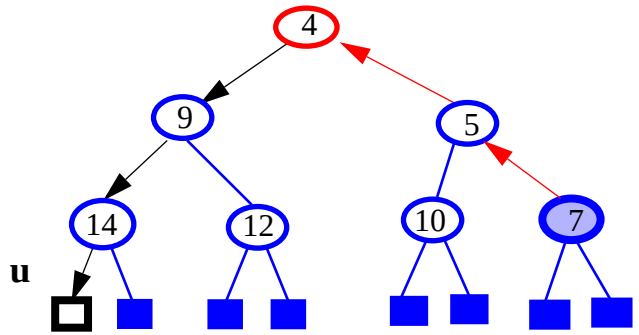
```
u = sist
while ( u != root()
        && u != leftChild(parent(u)) )
    u = parent(u) ;
if ( u != root)
    u = rightChild(parent(u)) ;
while ( ! isExternal(u) )
    u = leftChild(u) ;
return u                                  O(log n)
```

- **BinaryTree** interface
  (vilkårlig implementasjon)

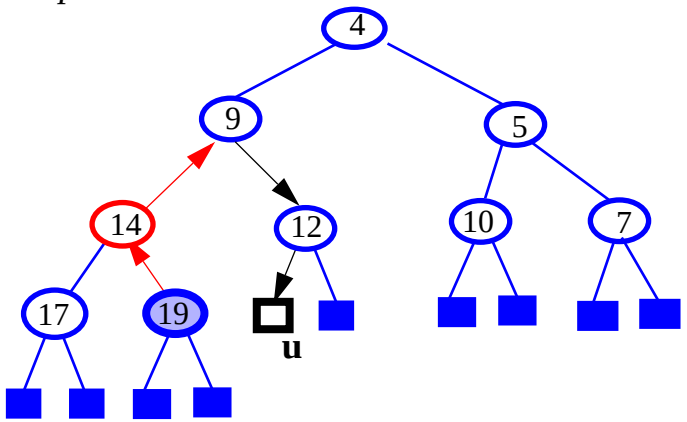
  Avhengig av hvem **sist** *er har vi tre tilfeller:*

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

b) *ytterste noden på nivå h-1*



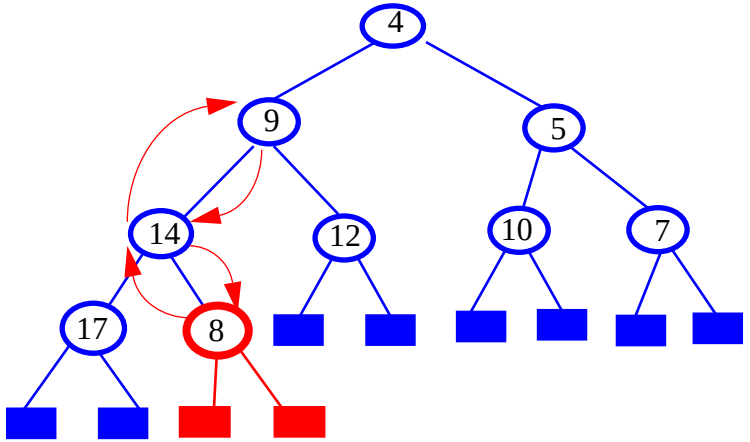
c) *en mellomnode på nivå h-1*



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)
```

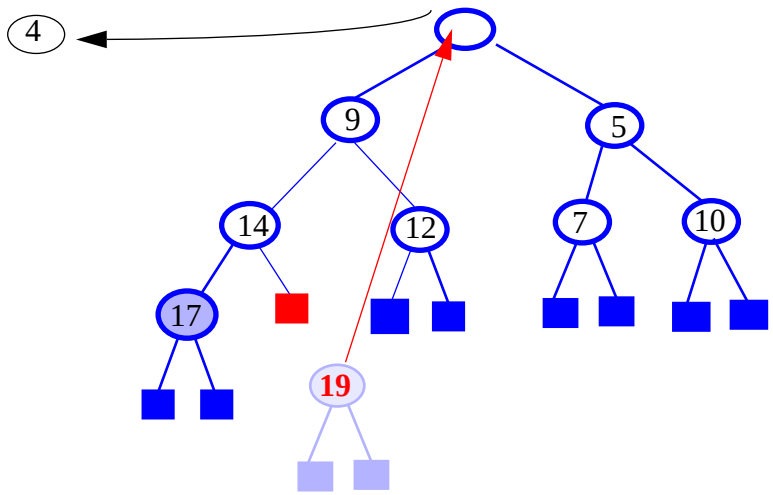
```
expandExternal(u) ;
u.setElement(ny) ;
```




```
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()

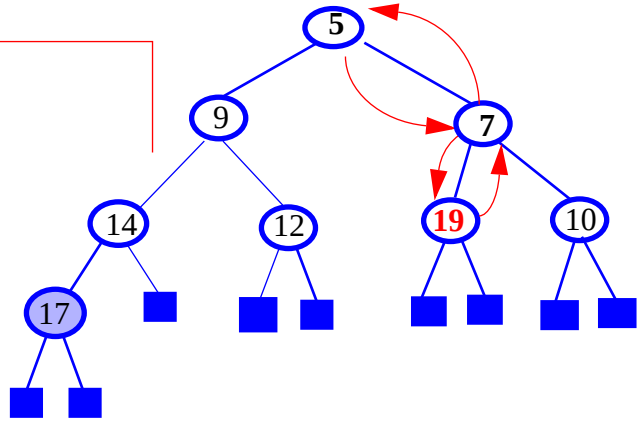


2. ***Bevar* KOMPLETT BINARYTREE INVARIANT** :
– *plasser sist.element() i rot-posisjon*
– *og fjern sist-posisjon = sett inn et nytt blad*
(BinTree: removeAboveExternal(leftChild(sist)))
– ***oppdater sist*** **O(1 + log n)**

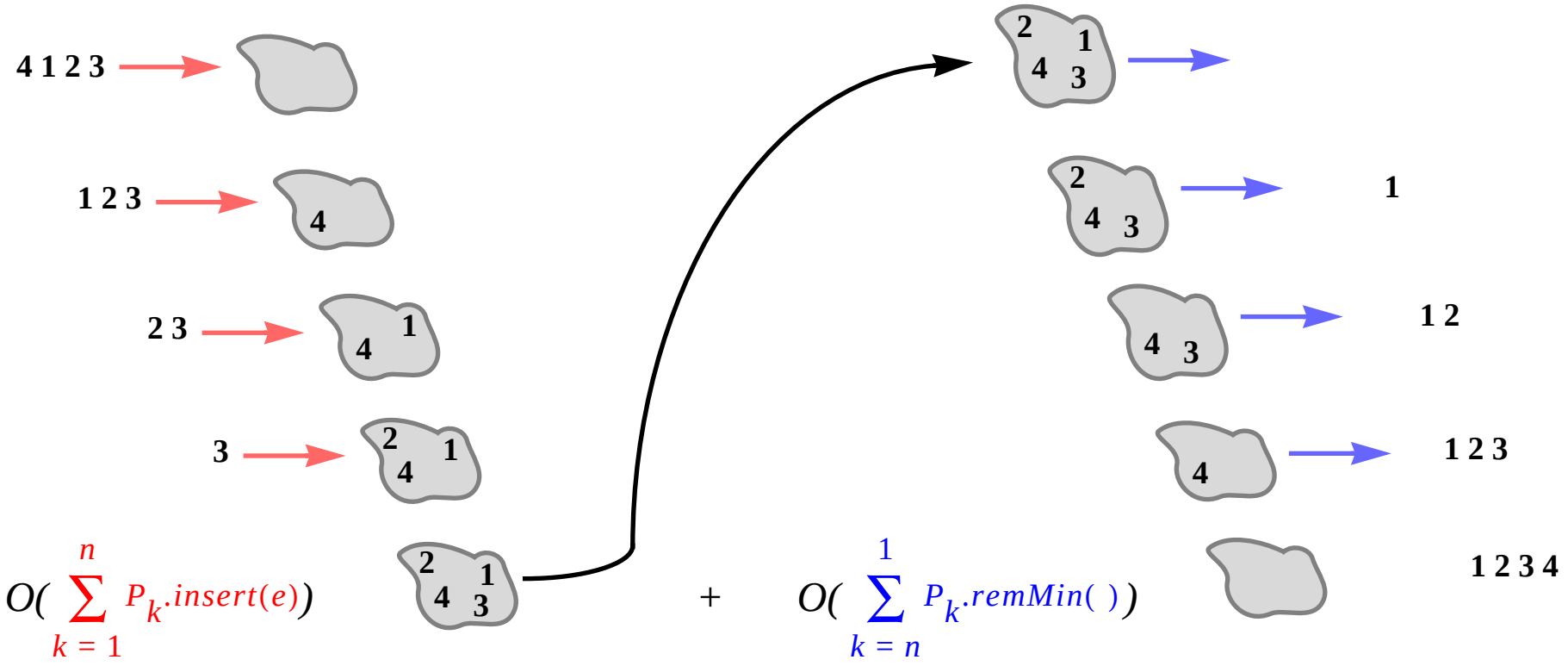
3. ***Gjenopprett* HEAP-ORDERING INVARIANT**:
– *flytt det nye rot-elementet "ned" til passende posisjon ("nedover bobling")*

```
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 $:O(n^2)$
Insertion Sort	4 1 2 3	$\frac{n^2+n}{2}$	**s:** 1 2 3 4	n	1 2 3 4 $:O(n^2)$
Heap Sort	4 1 2 3	$n \log(n)$	**h:**	$n \log(n)$	1 2 3 4 $:O(n \log(n))$

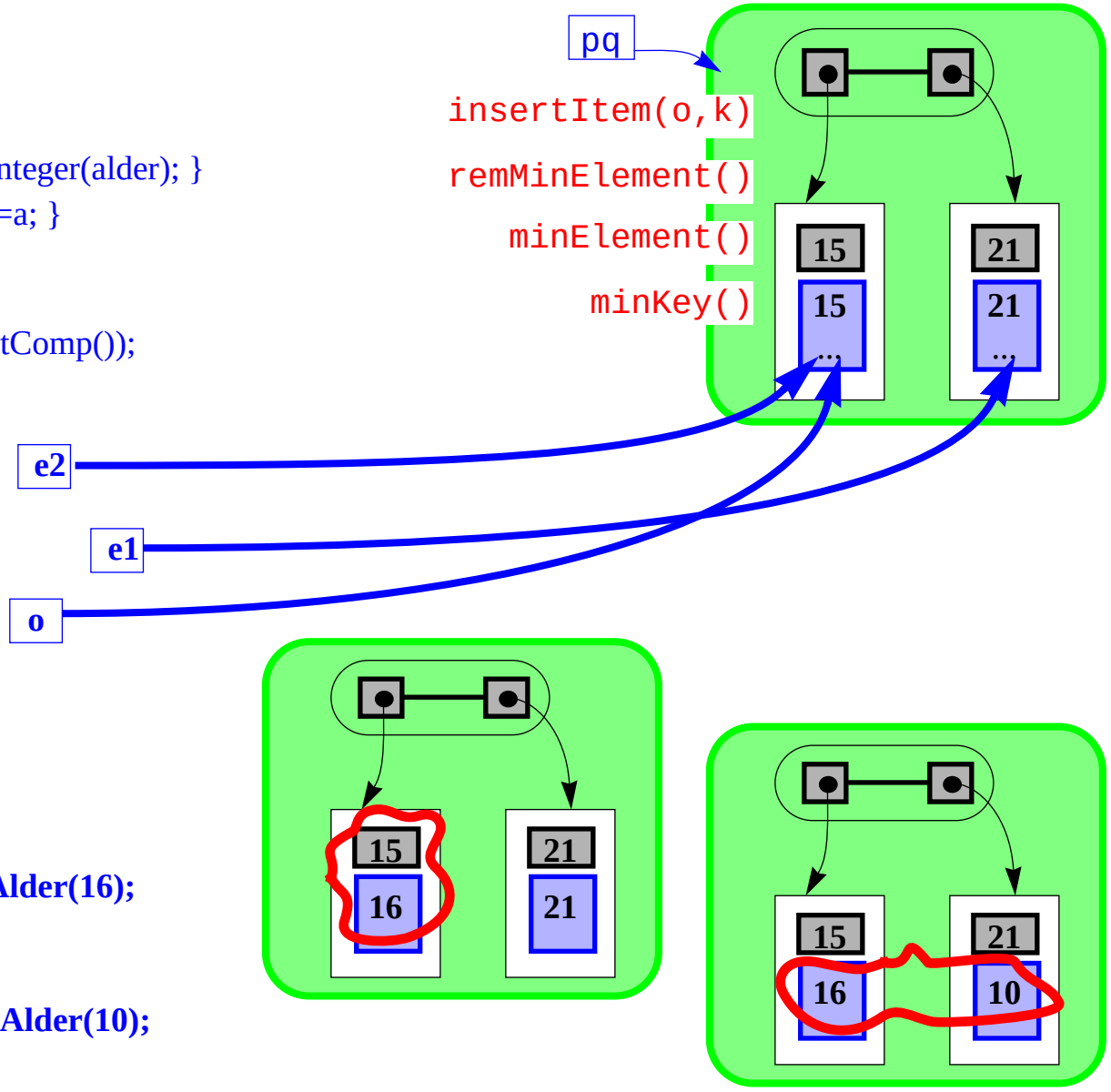
VI. Oppdater aldri objekter i en Samling!

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 ...

VI. *Locator* designmønster

Position – men for samlinger der Objekter lokaliseres v.hj.a. nøkkel – "posisjon for et objekt med en gitt nøkkel"

```
public interface KeyBasedContainer extends Container {

   /**  @return the Locator of the inserted pair <k,o>. */
     Locator insert(Object k , Object o) ;

   /**  Removes the element stored at the locator L */
     void remove(Locator l) ;

   /**  @return the element stoered previously at L */
     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 L , Object o) ;

     ObjectIterator keys() ;

     LocatorIterator locators() ;
}
```

```
public interface Locator {
    Object element()
    Object key()
}
```

for en Locator-basert PrioritetsKø, kunne vi også ha:

```
   /**  @return the Locator of the item with the least key. */
     Locator min() ;
```

implementasjon vil kreve en god del manipulering med datastrukturen

f. eks.:

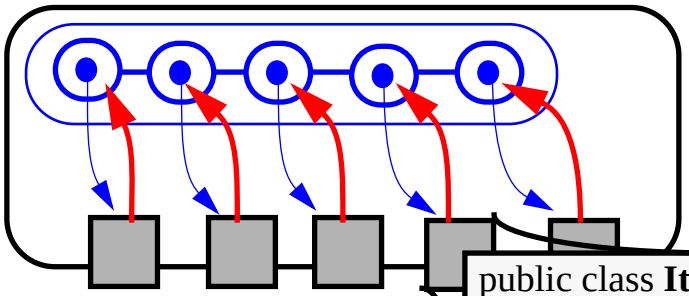
operasjon	output	PQ (sortert sekvens)
PQ . insert(5,A)	L1	<5,A>
PQ . insert(3,B)	L2	<3,B> <5,A>
PQ . insert(7,C)	L3	<3,B> <5,A> <7,C>
PQ . min()	L2	
L2 . key()	3	
PQ . remove(L1)		<3,B> <7,C>
PQ . replaceKey(L2,9)	3	<7,C> <9,B>
L2 . key()	9	
PQ . replaceElement(L3,D)	C	<7,D> <9,B>
PQ . min().element()	D	
PQ . remove(PQ.min()) = PQ . removeMin()		<9,B>

Implementasjon av PriorityQueue – *med Locator*

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en **Locator** Item



```
public class Item implements Locator {
  public Object element() {...}
  public Object key() {...}
  protected Position pos() {...}
  protected void setPos(Position p) {...}
  protected void setKey(Object k) {...}
  protected void setElement(Object o) {...}
... }
```

II DATA STRUKTUR

class PQ-L-Sequence extends PQSequence

implements PriorityQueue {

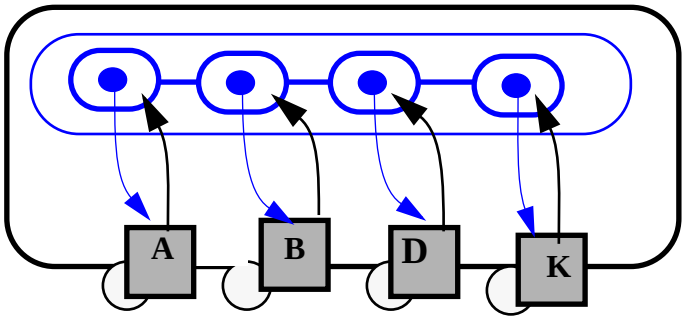
III DATA INVARIANT

- Item (i Locator) kan sammenlignes med cp
- For hver posisjon **p** i **S**:

  ((Locator) **p.element()**) . **pos()** == **p**
- S er sortert/usortert

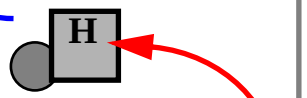
Sortert Sequence implementasjon med Locator

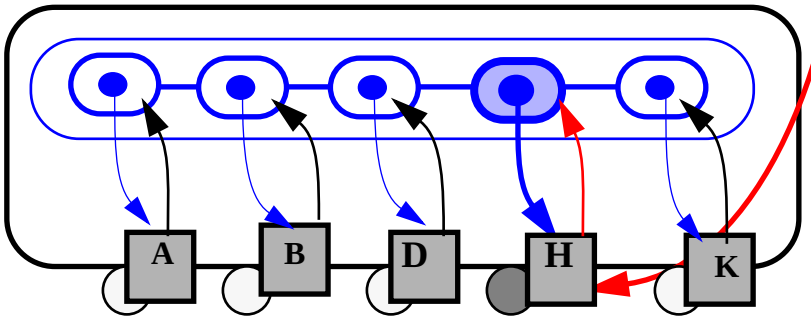
S:

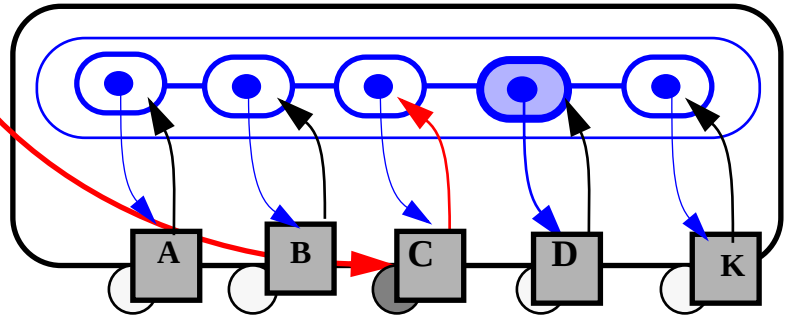


insert (“H”, o)
-- lag en **L** = Locator (“H”,o)

locInsert (**L**)
-- sett den inn i S (returnerer Position **p**)
-- **L** . setPos(**p**)
-- returner **L**







remove (**L**) :
-- S . remove(((Item)**L**) . pos())

removeMin ():
-- S . remove(S.first())

replaceKey (**L**, “C”)
-- Object oldKey = **L** . key()
-- S . remove(((Item)**L**) . pos())
-- **L** . setKey(“C”)
-- locInsert(**L**)
-- return oldKey

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*