

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

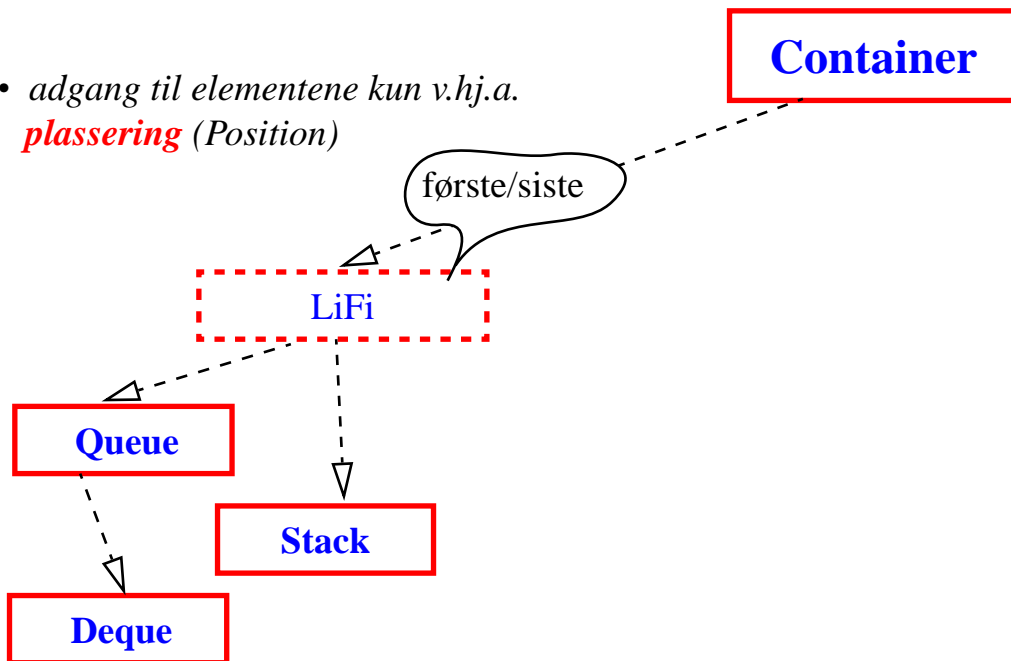
implmentasjon av Heap operasjoner

BinaryTree / Array

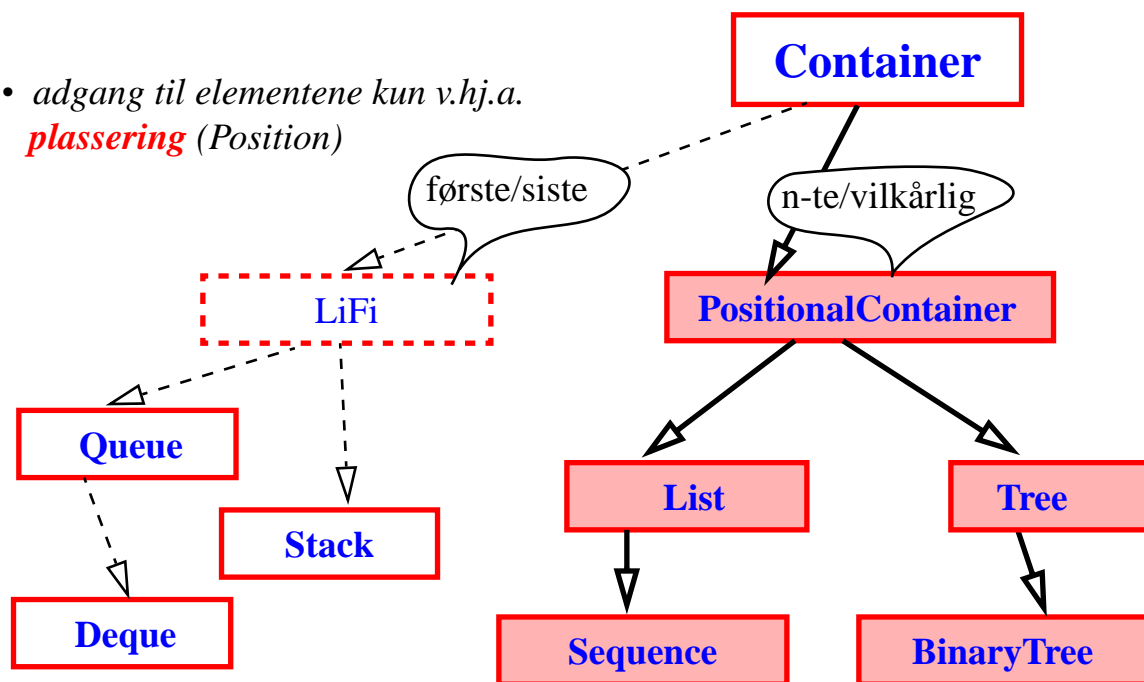
VI. LOCATOR DESIGNMØNSTER (7.4)

Kap. 7 (kursorisk: 7.4.2, 7.3.5)

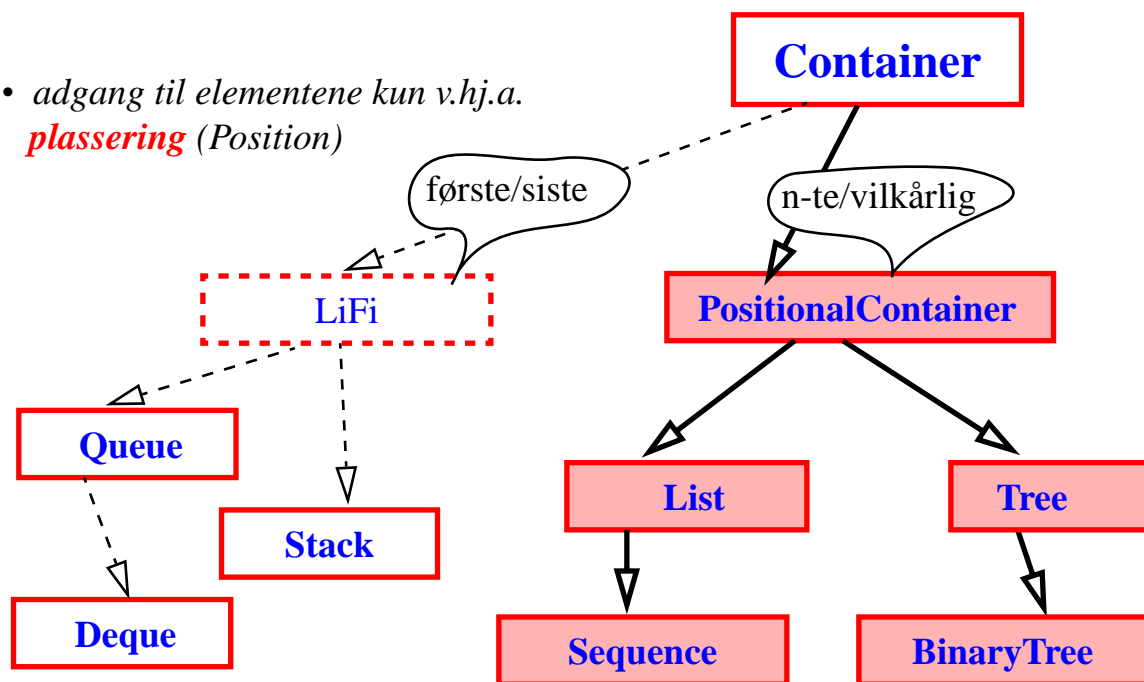
- adgang til elementene kun v.h.j.a.
plassering (Position)



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

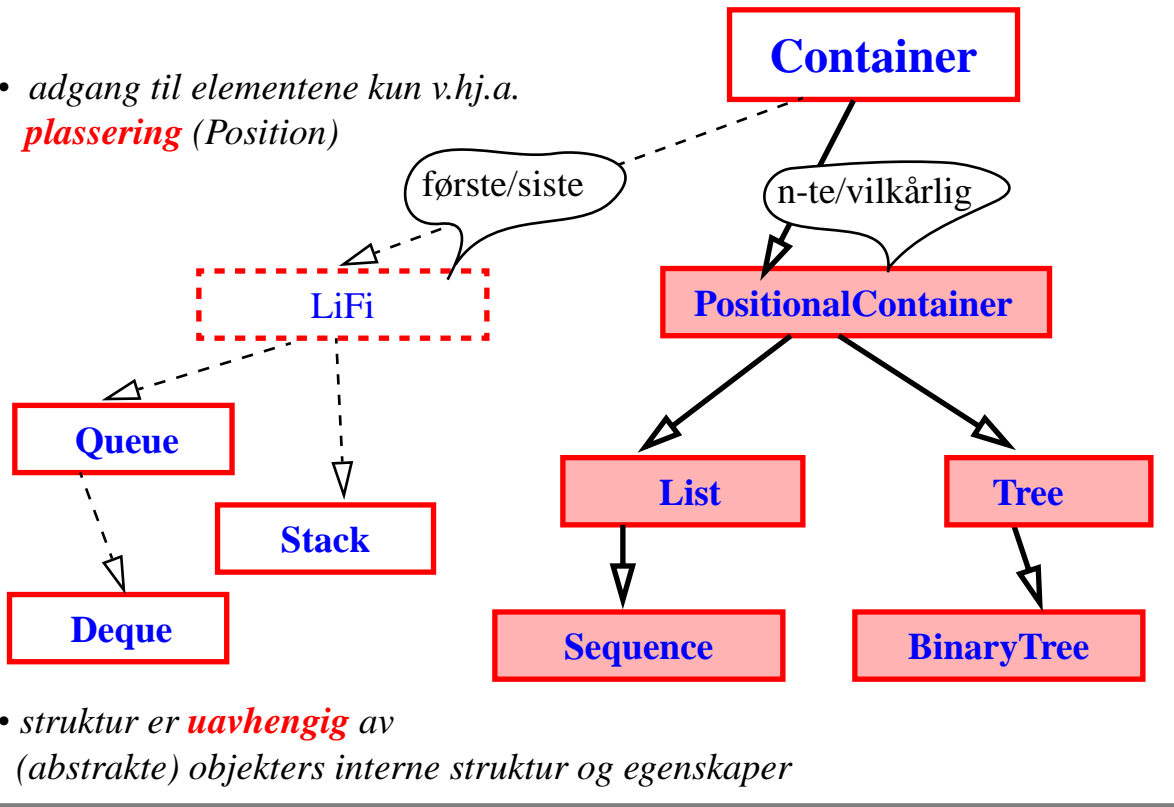


- adgang til elementene kun v.h.j.a. **plassering** (Position)



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

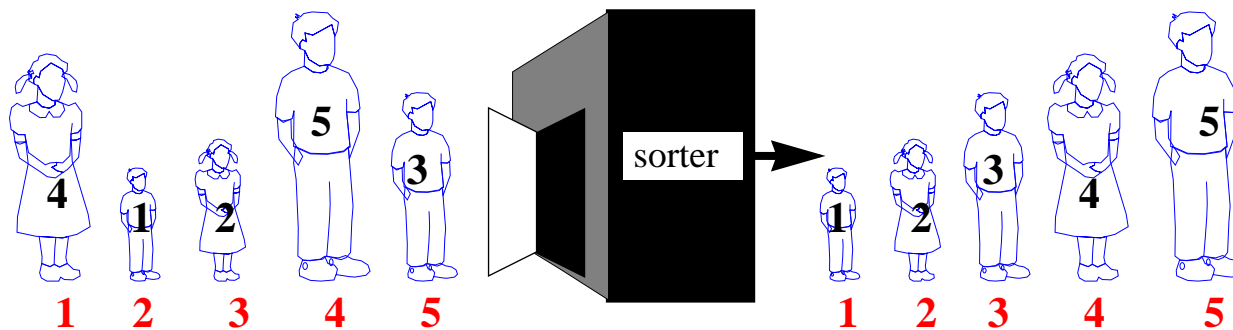
- adgang til elementene kun v.hj.a. **plassing** (Position)

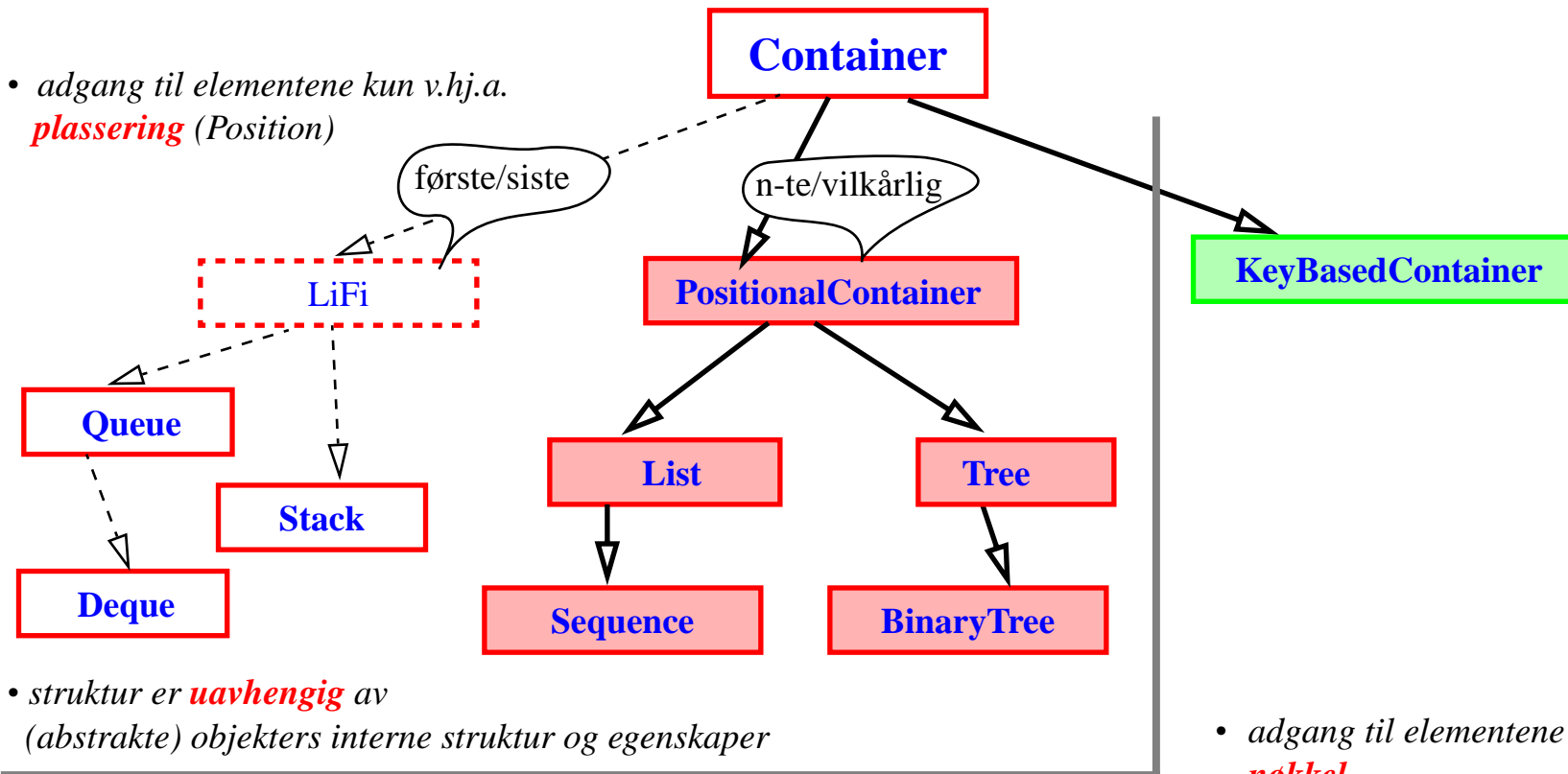


- struktur er **uavhengig** av (abstrakte) objekters interne struktur og egenskaper

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

- på strukturer med **rekkefølgen** på **Posisjoner**, og hvor
- alle **elementene** kan **sammenlignes** (mht. en **nøkkel-verdi**)

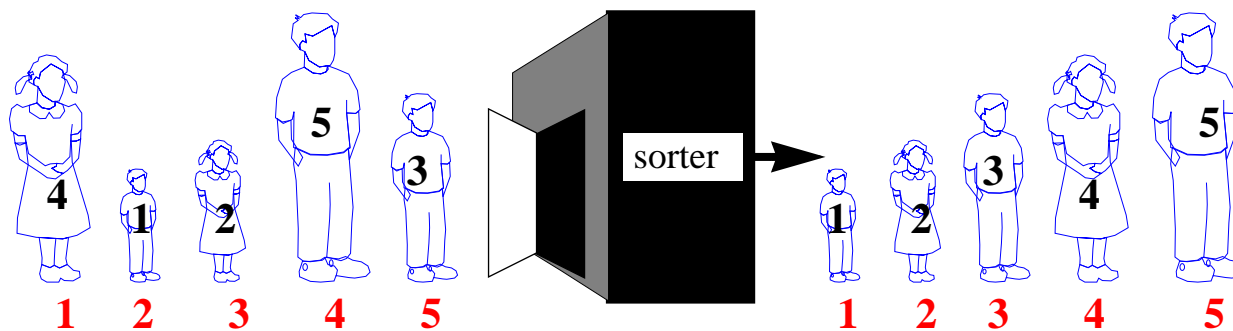


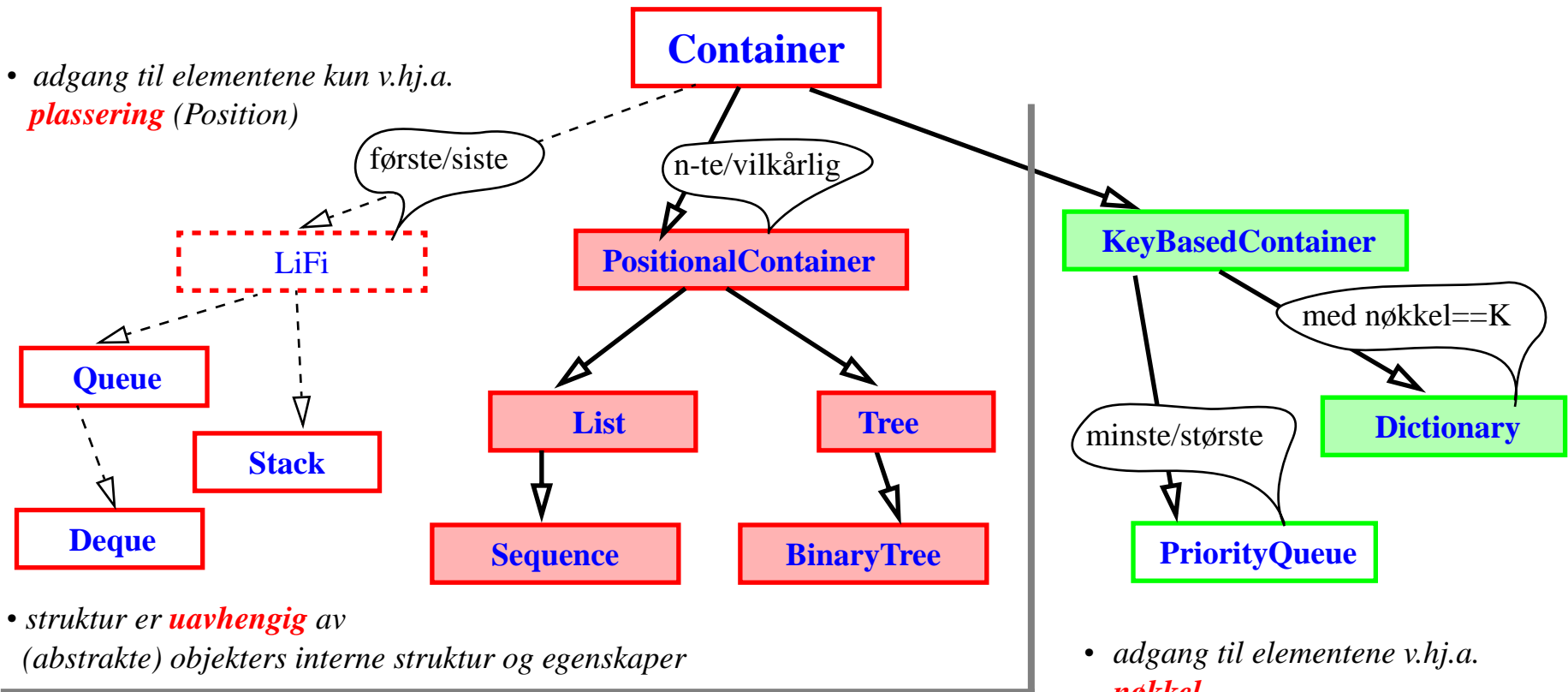


- adgang til elementene v.hj.a. **nøkkel**
- plassering i en samling **avhenger** av elementets verdi (nøkkel)

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

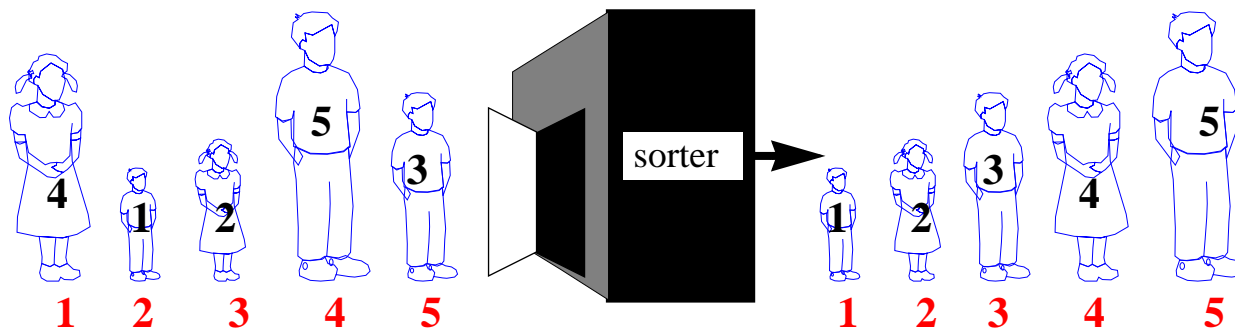
- på strukturer med **rekkefølgen** på **Posisjoner**, og hvor
- alle **elementene** kan **sammenlignes** (mht. en **nøkkel-verdi**)





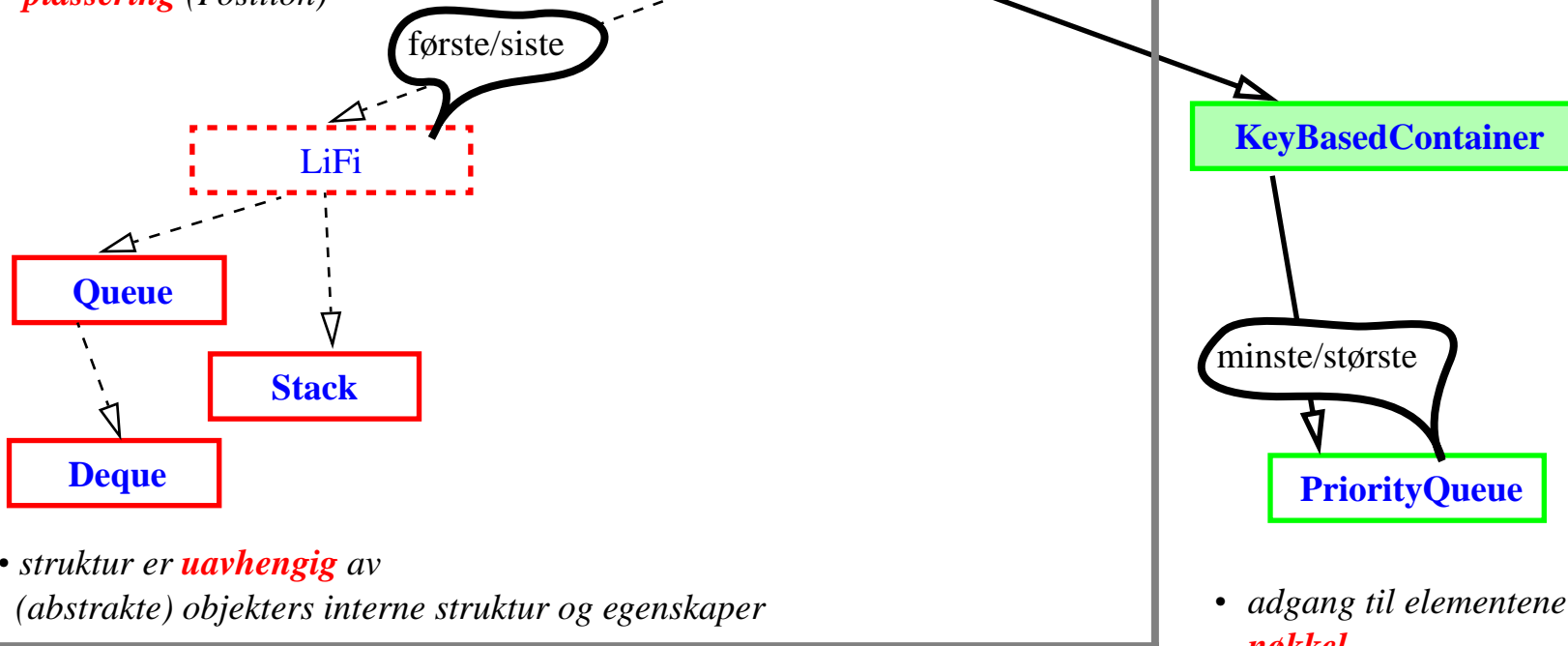
- adgang til elementene v.hj.a. **nøkkel**
- plassering i en samling **avhenger** av elementets verdi (nøkkel)

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



Container

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

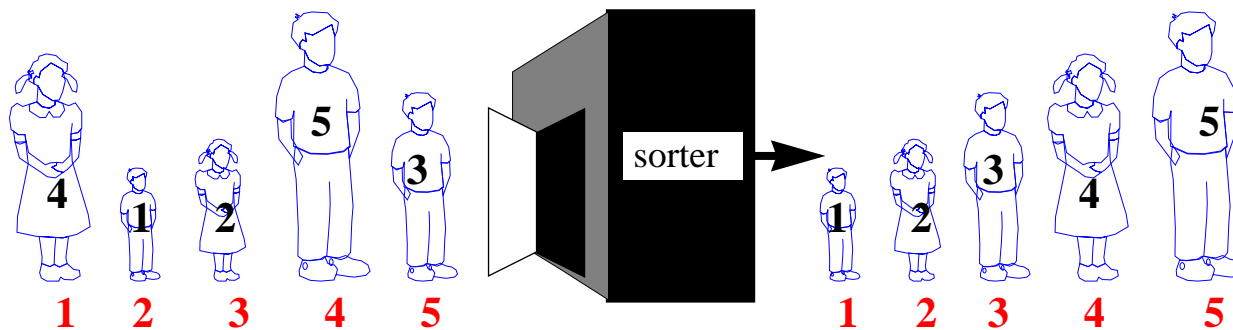


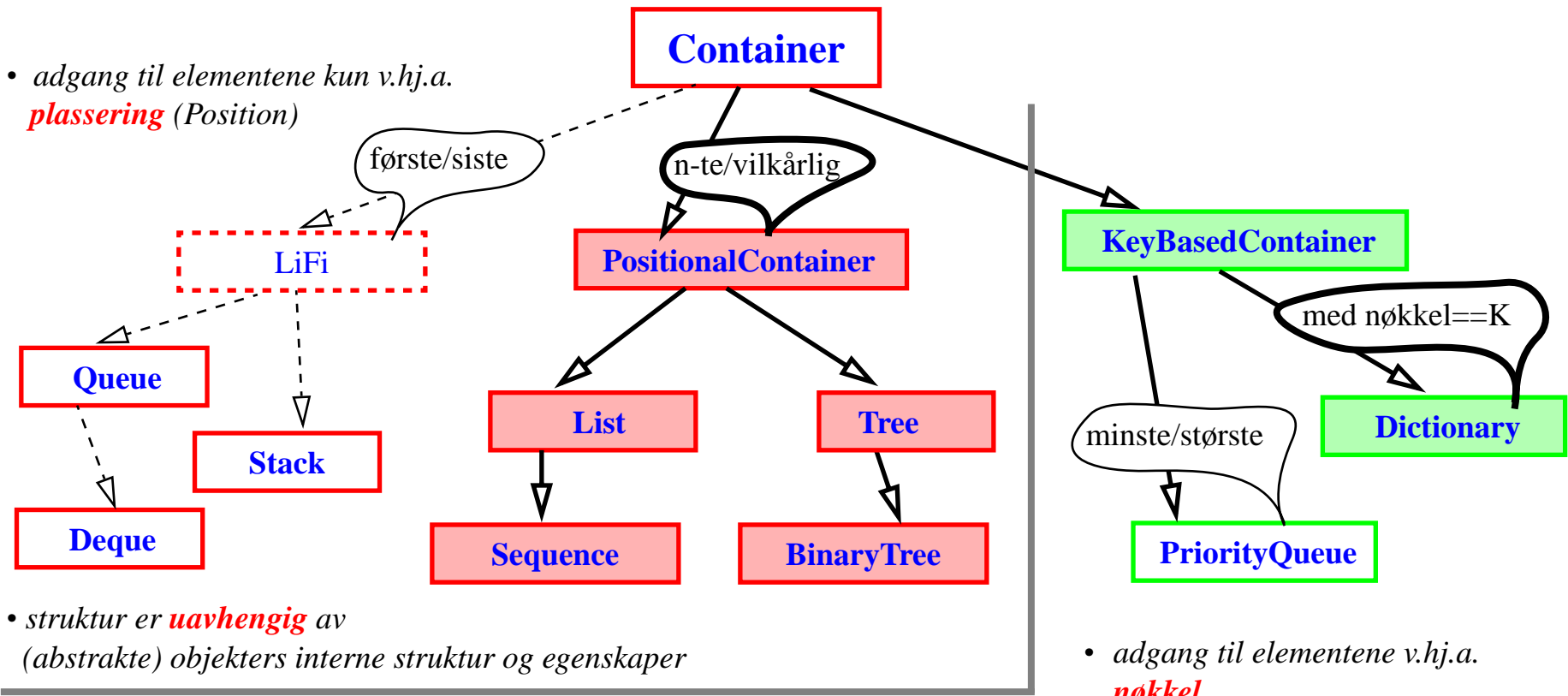
- struktur er **uavhengig** av (abstrakte) objekters interne struktur og egenskaper

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

- på strukturer med **rekkefølgen** på **Posisjoner**, og hvor
- alle **elementene** kan **sammenlignes** (mht. en **nøkkel-verdi**)

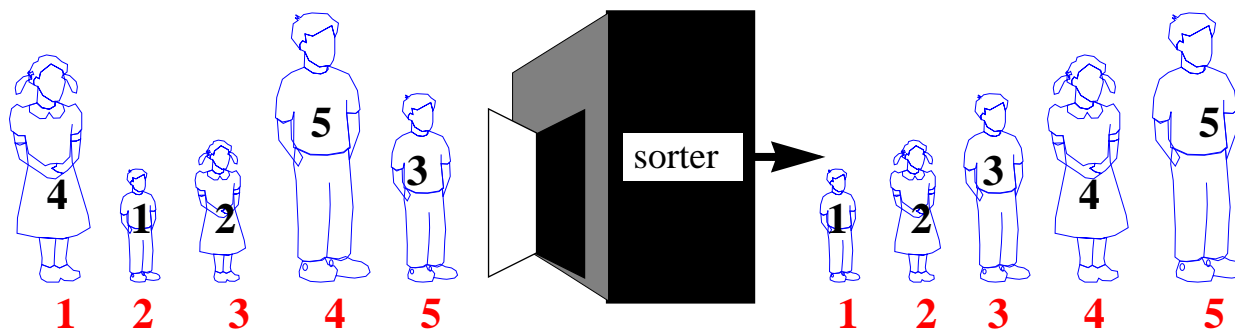
- adgang til elementene v.hj.a. **nøkkel**
- plassering i en samling **avhenger** av elementets verdi (nøkkel)





- 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 K :

1. en **binær relasjon** R på K , er en mengde av par (s,p) der s og $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$*)

<i>mengde</i> K :	$R(x,y)$
– heltallene	$x \leq y$

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 $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$*)
2. en relasjon R er en **total ordning (TO)** på K hviss den er:
 - **refleksiv** : $R(s,s)$ for alle $s \in K$
 - **transitiv** : for alle $s,p,q \in K$
hvis $R(s,p)$ & $R(p,q)$ så $R(s,q)$
 - **antisymmetrisk** : for alle $s,p \in K$
hvis $R(s,p)$ & $R(p,q)$ så $s = p$
 - **total** : for alle $s,p \in K$
enten $R(s,p)$ enten $R(p,q)$

<i>mengde</i> K :	$R(x,y)$
– heltallene	$x \leq y$

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 $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$*)
2. en relasjon R er en **total ordning (TO)** på K hvis den er:
 - **refleksiv** : $R(s,s)$ for alle $s \in K$
 - **transitiv** : for alle $s,p,q \in K$
hvis $R(s,p)$ & $R(p,q)$ så $R(s,q)$
 - **antisymmetrisk** : for alle $s,p \in K$
hvis $R(s,p)$ & $R(p,q)$ så $s = p$
 - **total** : for alle $s,p \in K$
enten $R(s,p)$ enten $R(p,q)$

<i>mengde K :</i>	<i>ordning $R(x,y)$</i>
– heltallene	$x \leq y$
– et alfabet	x kommer-ikke-etter y

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 $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$*)
2. en relasjon R er en **total ordning (TO)** på K hviss den er:
 - **refleksiv** : $R(s,s)$ for alle $s \in K$
 - **transitiv** : for alle $s,p,q \in K$
hvis $R(s,p)$ & $R(p,q)$ så $R(s,q)$
 - **antisymmetrisk** : for alle $s,p \in K$
hvis $R(s,p)$ & $R(p,q)$ så $s = p$
 - **total** : for alle $s,p \in K$
enten $R(s,p)$ enten $R(p,q)$

mengde K :

- heltallene
- et alfabet
- alle strenger (ord)
over et gitt alfabet

ordning $R(x,y)$

$x \leq y$

x kommer-ikke-etter y

leksiskografisk ordning

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 $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$*)
2. en relasjon R er en **total ordning (TO)** på K hviss den er:
 - **refleksiv** : $R(s,s)$ for alle $s \in K$
 - **transitiv** : for alle $s,p,q \in K$
hvis $R(s,p)$ & $R(p,q)$ så $R(s,q)$
 - **antisymmetrisk** : for alle $s,p \in K$
hvis $R(s,p)$ & $R(p,q)$ så $s = p$
 - **total** : for alle $s,p \in K$
enten $R(s,p)$ enten $R(p,q)$

mengde K :

ordning $R(x,y)$

- | | |
|---|---------------------------|
| – heltallene | $x \leq y$ |
| – et alfabet | x kommer-ikke-etter y |
| – alle strenger (ord)
over et gitt alfabet | leksiskografisk ordning |

Samme mengde kan TOs på forskjellige måter

- | | |
|--------------|------------|
| – heltallene | $x \geq y$ |
|--------------|------------|

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 $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$)*)
2. en relasjon R er en **total ordning (TO)** på K hviss den er:
 - **refleksiv** : $R(s,s)$ for alle $s \in K$
 - **transitiv** : for alle $s,p,q \in K$ hvis $R(s,p)$ & $R(p,q)$ så $R(s,q)$
 - **antisymmetrisk** : for alle $s,p \in K$ hvis $R(s,p)$ & $R(p,q)$ så $s = p$
 - **total** : for alle $s,p \in K$ enten $R(s,p)$ enten $R(p,q)$

<i>mengde K :</i>	<i>ordning $R(x,y)$</i>
– heltallene	$x \leq y$
– et alfabet	x kommer-ikke-etter y
– alle strenger (ord) over et gitt alfabet	leksiskografisk ordning
<i>Samme mengde kan TOs på forskjellige måter</i>	
– heltallene	$x \geq y$
– heltallene	$x < y$
	+ $x \leq y$
– et alfabet	x kommer-før y
	+ x kommer-ikke-etter y
– mennesker	x yngre-enn y
	+ x ikke-eldre-enn y

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 $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$*)
2. en relasjon R er en **total ordning (TO)** på K hviss den er:
 - **refleksiv** : $R(s,s)$ for alle $s \in K$
 - **transitiv** : for alle $s,p,q \in K$ hvis $R(s,p)$ & $R(p,q)$ så $R(s,q)$
 - **antisymmetrisk** : for alle $s,p \in K$ hvis $R(s,p)$ & $R(p,q)$ så $s = p$
 - **total** : for alle $s,p \in K$ enten $R(s,p)$ enten $R(p,q)$

<i>mengde K :</i>	<i>ordning $R(x,y)$</i>
– heltallene	$x \leq y$
– et alfabet	x kommer-ikke-etter y
– alle strenger (ord) over et gitt alfabet	leksiskografisk ordning
<i>Samme mengde kan TOs på forskjellige måter</i>	
– heltallene	$x \geq y$
– heltallene	$x < y$
	+ $x \leq y$
– et alfabet	x kommer-før y
	+ x kommer-ikke-etter y
– mennesker	x yngre-enn y
	+ x ikke-eldre-enn y

Nøkkel (for en mengde E) er en funksjon

key : $E \rightarrow K$, der

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

II. **key** er **injektiv**, dvs. slik at :

hvis $e \neq f$ så $key(e) \neq key(f)$

(forskjellige elementer har forskjellige nøkkel-verdier)

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

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 $p \in K$
(*man skriver ofte $(s,p) \in R$ som $R(s,p)$, evt. $s \leq p$)*)
2. en relasjon R er en **total ordning (TO)** på K hviss den er:
 - **refleksiv** : $R(s,s)$ for alle $s \in K$
 - **transitiv** : for alle $s,p,q \in K$ hvis $R(s,p)$ & $R(p,q)$ så $R(s,q)$
 - **antisymmetrisk** : for alle $s,p \in K$ hvis $R(s,p)$ & $R(p,q)$ så $s = p$
 - **total** : for alle $s,p \in K$ enten $R(s,p)$ enten $R(p,q)$

<i>mengde</i> K :	<i>ordning</i> $R(x,y)$
– heltallene	$x \leq y$
– et alfabet	x kommer-ikke-etter y
– alle strenger (ord) over et gitt alfabet	leksiskografisk ordning
<i>Samme mengde kan TOs på forskjellige måter</i>	
– heltallene	$x \geq y$
– heltallene	$x < y$
	+ $x \leq y$
– et alfabet	x kommer-før y
	+ x kommer-ikke-etter y
– mennesker	x yngre-enn y
	+ x ikke-eldre-enn y

Nøkkel (for en mengde E) er en funksjon

$key : E \rightarrow K$, der

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

II. key er **injektiv**, dvs. slik at :

hvis $e \neq f$ så $key(e) \neq key(f)$

(forskjellige elementer har forskjellige nøkkel-verdier)

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

E	\rightarrow	K	:	$key(e)$
– personer	\rightarrow	personnumre (heltall, \leq)	:	e 's persnr

II. Nøkler og Ordninger

Gitt en (**vilkårlig**) mengde K :

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

<i>mengde</i> K :	<i>ordning</i> $R(x,y)$
– heltallene	$x \leq y$
– et alfabet	x kommer-ikke-etter y
– alle strenger (ord) over et gitt alfabet	leksiskografisk ordning
<i>Samme mengde kan TOs på forskjellige måter</i>	
– heltallene	$x \geq y$
– heltallene	$x < y$
	+ $x \leq y$
– et alfabet	x kommer-før y
	+ x kommer-ikke-etter y
– mennesker	x yngre-enn y
	+ x ikke-eldre-enn y

Nøkkel (for en mengde E) er en funksjon

key : $E \rightarrow K$, der

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

II. **key** er **injektiv**, dvs. slik at :

hvis $e \neq f$ så **key**(e) \neq **key**(f)

(forskjellige elementer har forskjellige nøkkel-verdier)

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

E	\rightarrow	K	:	key (e)
– personer	\rightarrow	personnumre (heltall, \leq)	:	e 's persnr
– mennesker	\rightarrow	heltall, \leq	:	e 's alder ?

II. Nøkler og Ordninger

Gitt en (vilkårlig) mengde K :

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

<i>mengde</i> K :	<i>ordning</i> $R(x,y)$
– heltallene	$x \leq y$
– et alfabet	x kommer-ikke-etter y
– alle strenger (ord) over et gitt alfabet	leksiskografisk ordning
<i>Samme mengde kan TOs på forskjellige måter</i>	
– heltallene	$x \geq y$
– heltallene	$x < y$
– et alfabet	+ $x \leq y$ x kommer-før y
– mennesker	+ x kommer-ikke-etter y x yngre-enn y + x ikke-eldre-enn y

Nøkkel (for en mengde E) er en funksjon

$key : E \rightarrow K$, der

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

II. **key er injektiv, dvs. slik at :**

hvis $e \neq f$ så $key(e) \neq key(f)$

(forskjellige elementer har forskjellige nøkkel-verdier)

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

E	\rightarrow	K	:	$key(e)$
– personer	\rightarrow	personnumre (heltall, \leq)	:	e 's persnr
– mennesker	\rightarrow	heltall, \leq	:	e 's alder ?

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

Objekter med nøkler

```
class Pers {  
    int alder;  
    int pnr;  
    String adr;  
    ... }  
}
```

Objekter med nøkler

```
class Pers {  
    int alder;  
    int pnr;  
    String adr;  
    ... }  
}
```

key1: Pers → heltall, <= e.alder;

Objekter med nøkler

```
class Pers {  
    int alder;  
    int pnr;  
    String adr;  
    ... }
```

key1: Pers → heltall, <=

e.alder;

(ikke entydig)

Objekter med nøkler

```
class Pers {  
    int alder;  
    int pnr;  
    String adr;  
    ... }
```

key1: Pers → heltall, <=

e.alder;

(ikke entydig)

key2: Pers → heltall, <=

e.pnr;

(entydig)

Objekter med nøkler

```
class Pers {  
    int alder;  
    int pnr;  
    String adr;  
    ... }
```

key1: Pers	→	heltall, <=	e.alder;	(ikke entydig)
key2: Pers	→	heltall, <=	e.pnr;	(entydig)
key3: Pers	→	String, compareTo	e.adr;	(ikke entydig)

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)  
    ... }  
}
```

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

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)  
    ... }  
}
```

- 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;}  
}
```

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)  
    ... }  
}
```

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

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)  
    ... }  
}
```

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

```
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(), ...

PriorityQueue ADT

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

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

void *insertItem*(Object k, Object e);

Object *minElement*();

Object *minKey*();

Object *removeMin*();

Locator *min*(); }

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);

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);

når e har et Object-attributt som nøkkel :

PQ . insertItem(e.key(), e)

men nøkkel kan bestemmes ved innsetting : PQ .

insertItem(newKey(e), e)

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();

når e har et Object-attributt som nøkkel :

PQ . insertItem(e.key(), e)

men nøkkel kan bestemmes ved innsetting : PQ .

insertItem(newKey(e), e)

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();

når e har et Object-tributt som nøkkel :

PQ . insertItem(e.key(), e)

men nøkkel kan bestemmes ved innsetting : PQ .

insertItem(newKey(e), e)

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();

når e har et Object-attributt som nøkkel :

PQ . insertItem(e.key(), e)

men nøkkel kan bestemmes ved innsetting : PQ .

insertItem(newKey(e), e)

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-attributt 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())
```

4 1 2 3

```
    e = S.removeFirst()
```

```
    PQ.insertItem(e,e)
```

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

```
    e = PQ.removeMin()
```

```
    S.insertLast(e)
```

PrioritetsKø-Sortering

```
void pqSort(Sequence S, PriorityQueue PQ)
```

```
// antar at PQ bruker Objektene som nøkler
```

```
a) while (! S.isEmpty())
```

4 1 2 3

```
    e = S.removeFirst()
```

```
    PQ.insertItem(e,e)
```

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

```
    e = PQ.removeMin()
```

```
    S.insertLast(e)
```

1 2 3 4

PrioritetsKø-Sortering

```
void pqSort(Sequence S, PriorityQueue PQ)
```

```
// antar at PQ bruker Objektene som nøkler
```

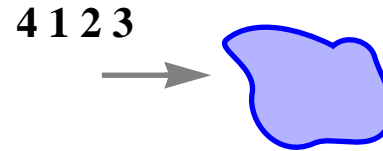
```
a) while (! S.isEmpty())
```

```
// fjern ett og ett element fra S = O(1)
```

```
    e = S.removeFirst()
```

```
// og sett dem inn i PQ
```

```
    PQ.insertItem(e,e)
```



PrioritetsKø-Sortering

```
void pqSort(Sequence S, PriorityQueue PQ)
```

```
// antar at PQ bruker Objektene som nøkler
```

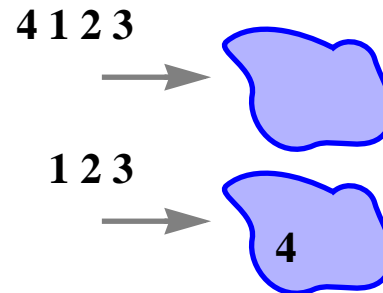
```
a) while (! S.isEmpty())
```

```
// fjern ett og ett element fra S = O(1)
```

```
    e = S.removeFirst()
```

```
// og sett dem inn i PQ
```

```
    PQ.insertItem(e,e)
```



PrioritetsKø-Sortering

```
void pqSort(Sequence S, PriorityQueue PQ)
```

```
// antar at PQ bruker Objektene som nøkler
```

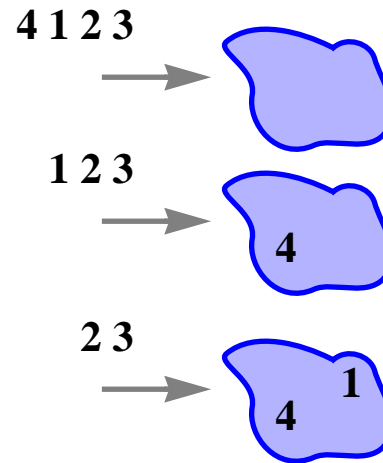
```
a) while (! S.isEmpty())
```

```
// fjern ett og ett element fra S = O(1)
```

```
    e = S.removeFirst()
```

```
// og sett dem inn i PQ
```

```
    PQ.insertItem(e,e)
```



PrioritetsKø-Sortering

```
void pqSort(Sequence S, PriorityQueue PQ)
```

```
// antar at PQ bruker Objektene som nøkler
```

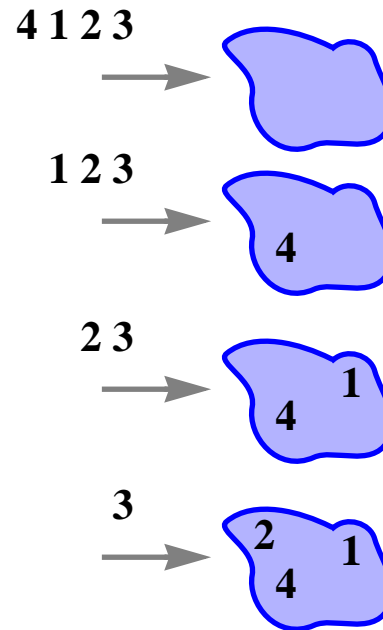
```
a) while (! S.isEmpty())
```

```
// fjern ett og ett element fra S = O(1)
```

```
    e = S.removeFirst()
```

```
// og sett dem inn i PQ
```

```
    PQ.insertItem(e,e)
```



PrioritetsKø-Sortering

```
void pqSort(Sequence S, PriorityQueue PQ)
```

```
// antar at PQ bruker Objektene som nøkler
```

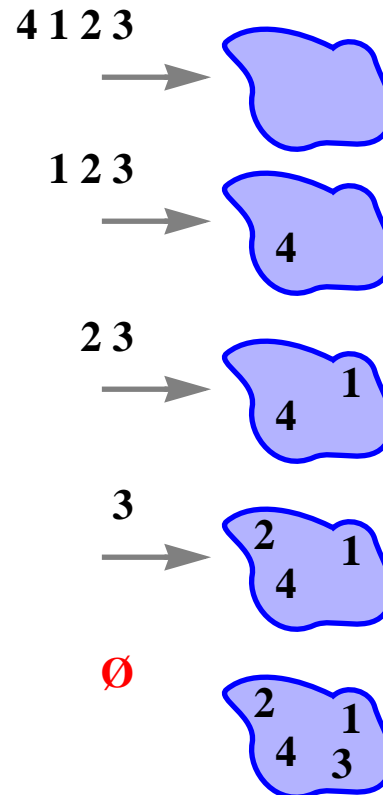
```
a) while (! S.isEmpty())
```

```
// fjern ett og ett element fra S = O(1)
```

```
    e = S.removeFirst()
```

```
// og sett dem inn i PQ
```

```
    PQ.insertItem(e,e)
```



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)
```

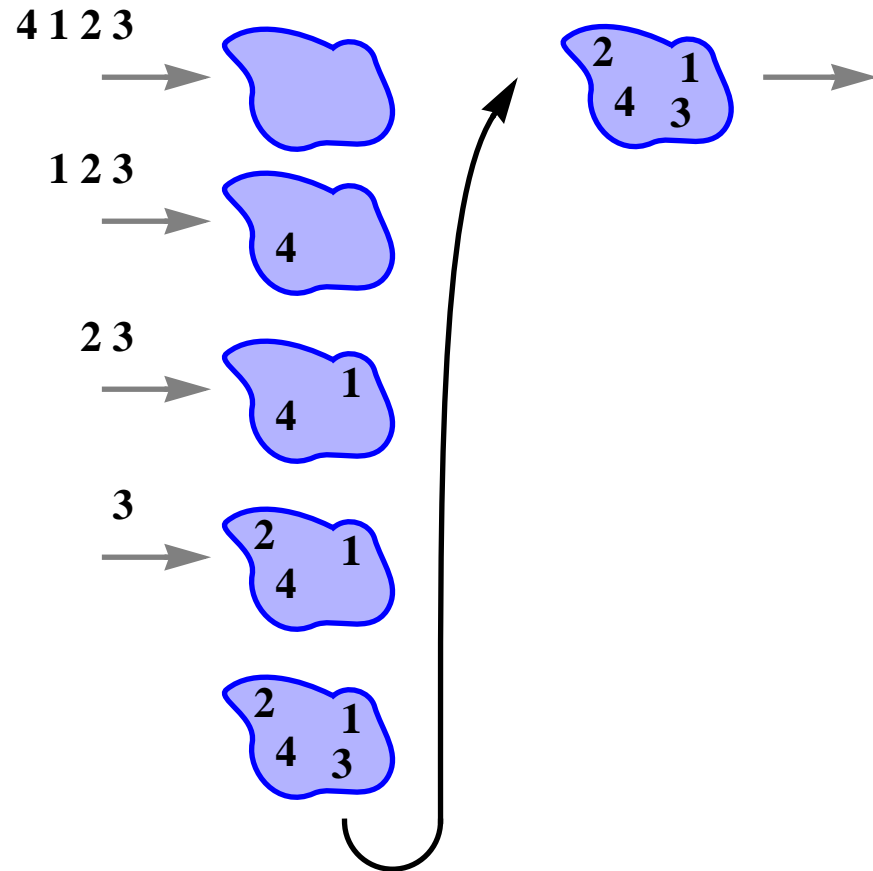
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

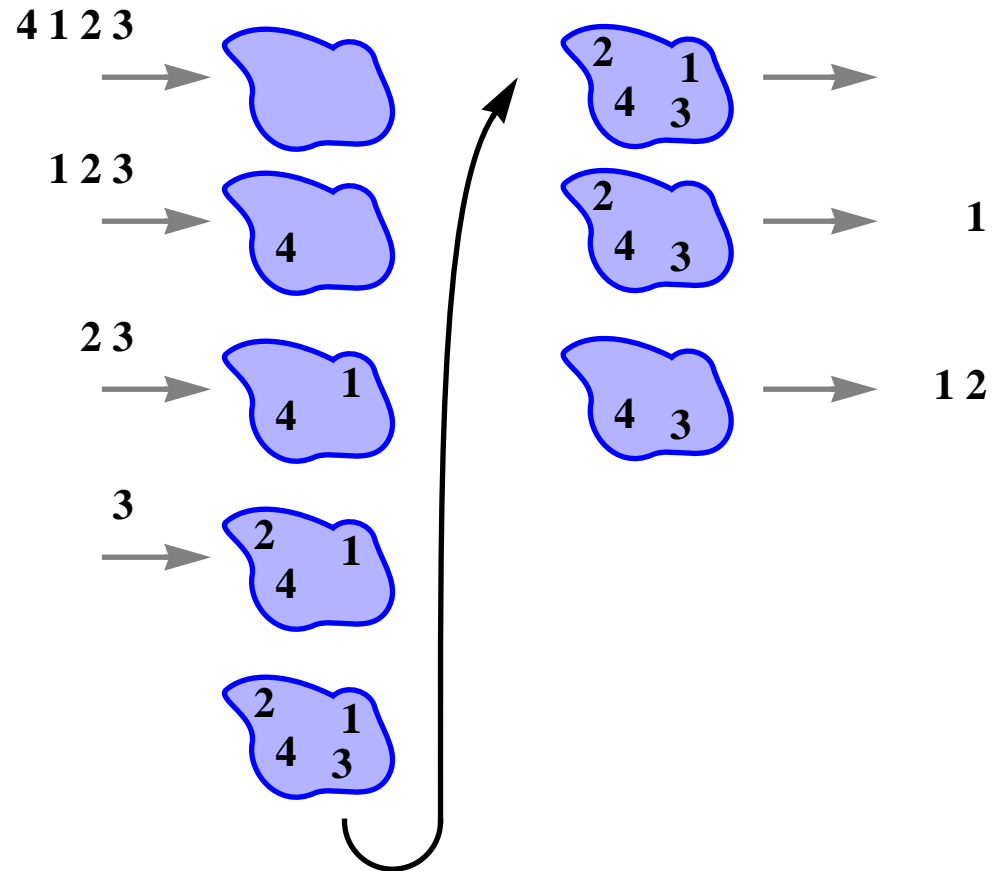
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

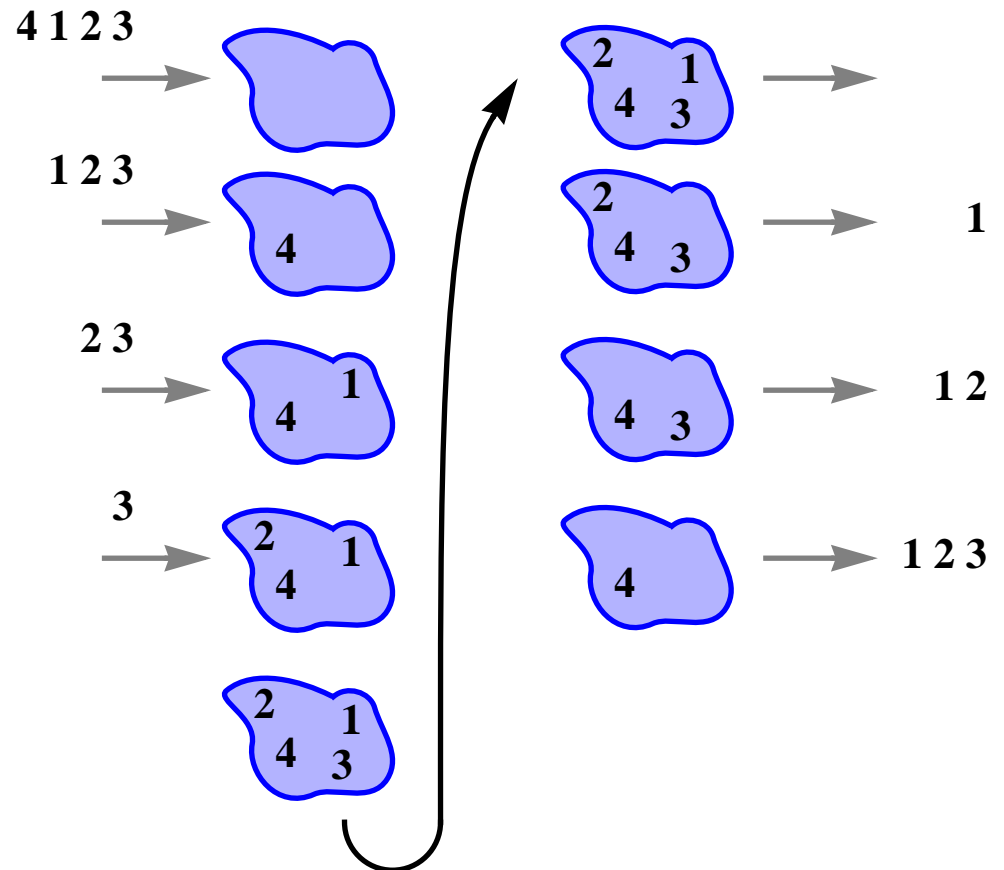
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

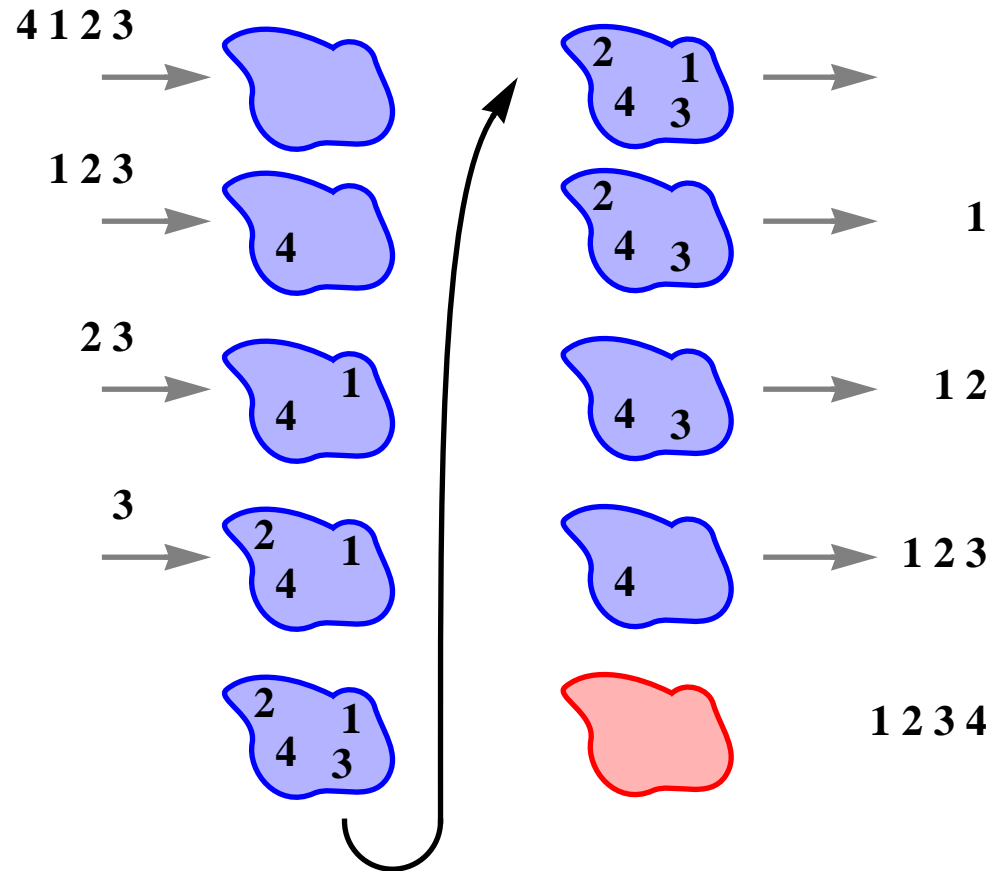
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

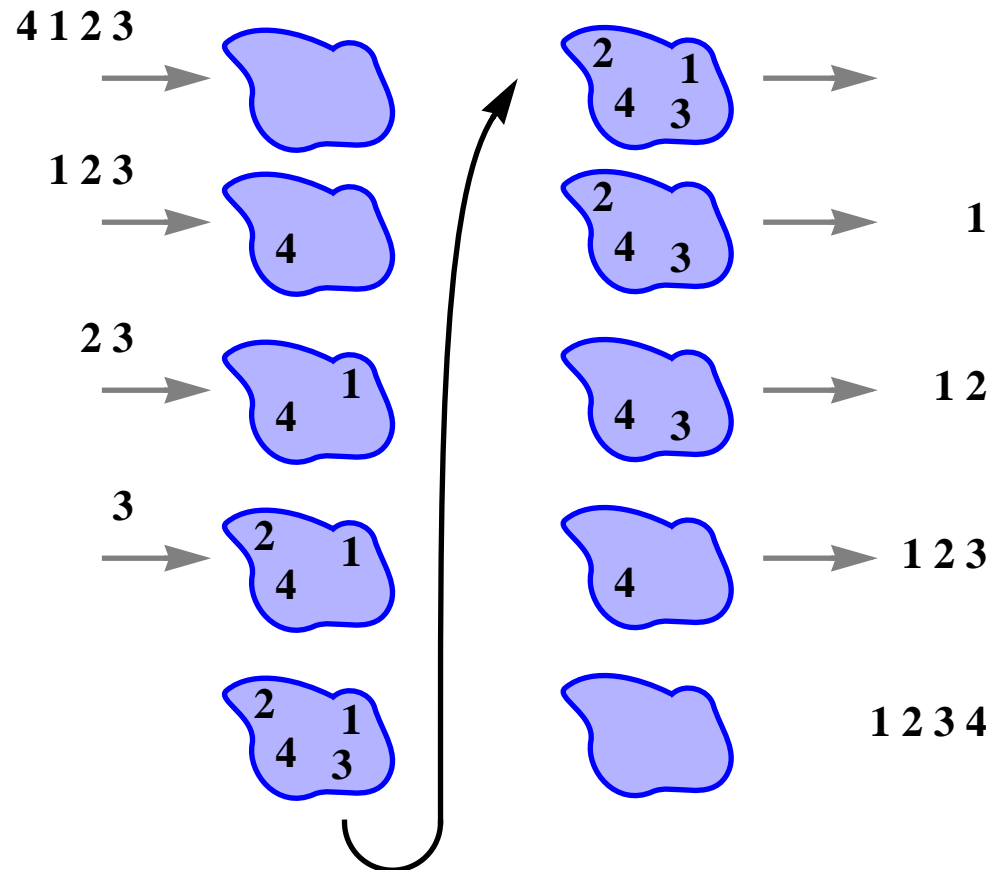
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

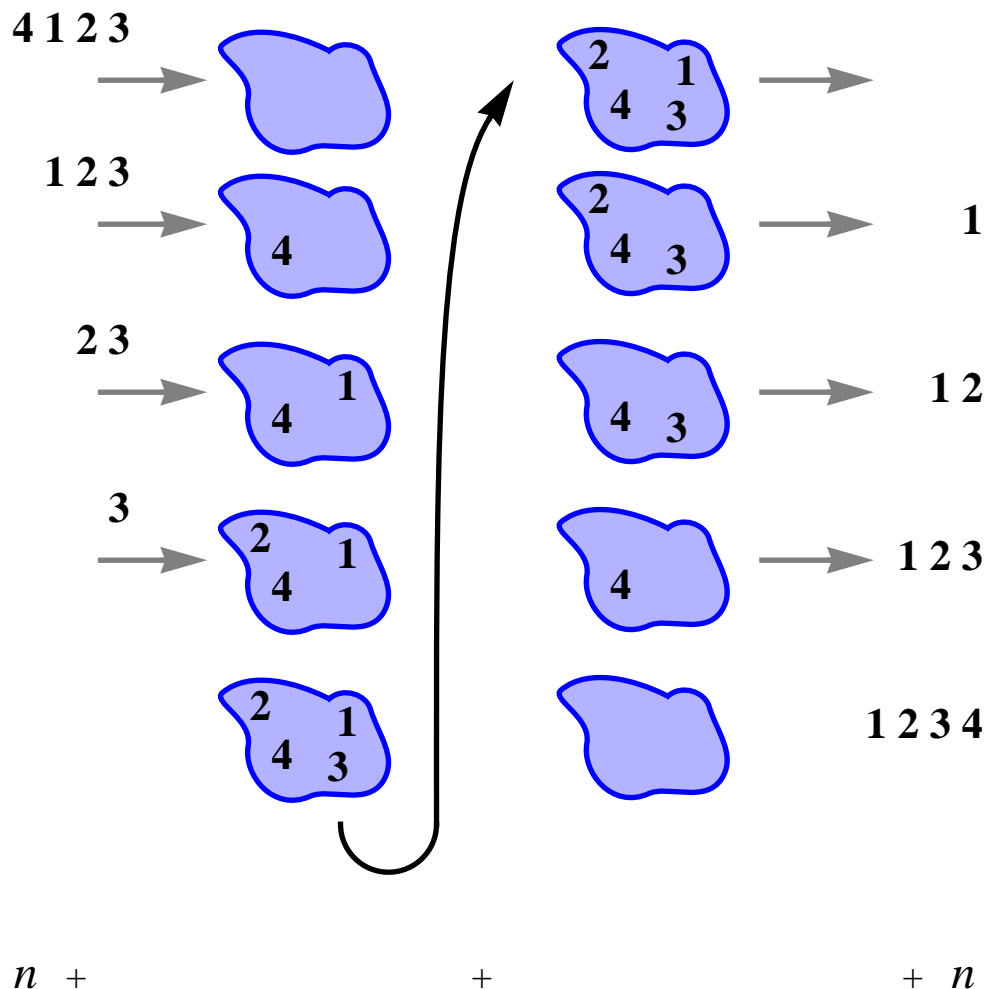
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

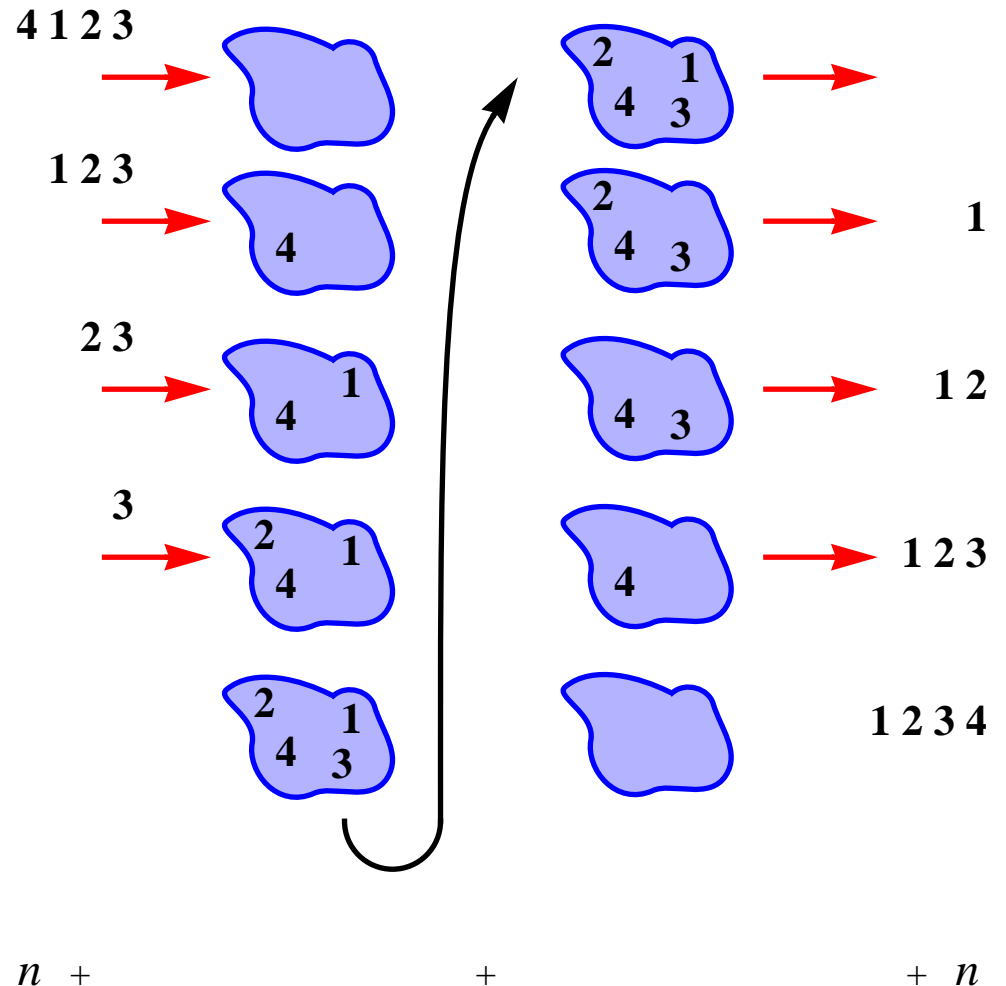
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

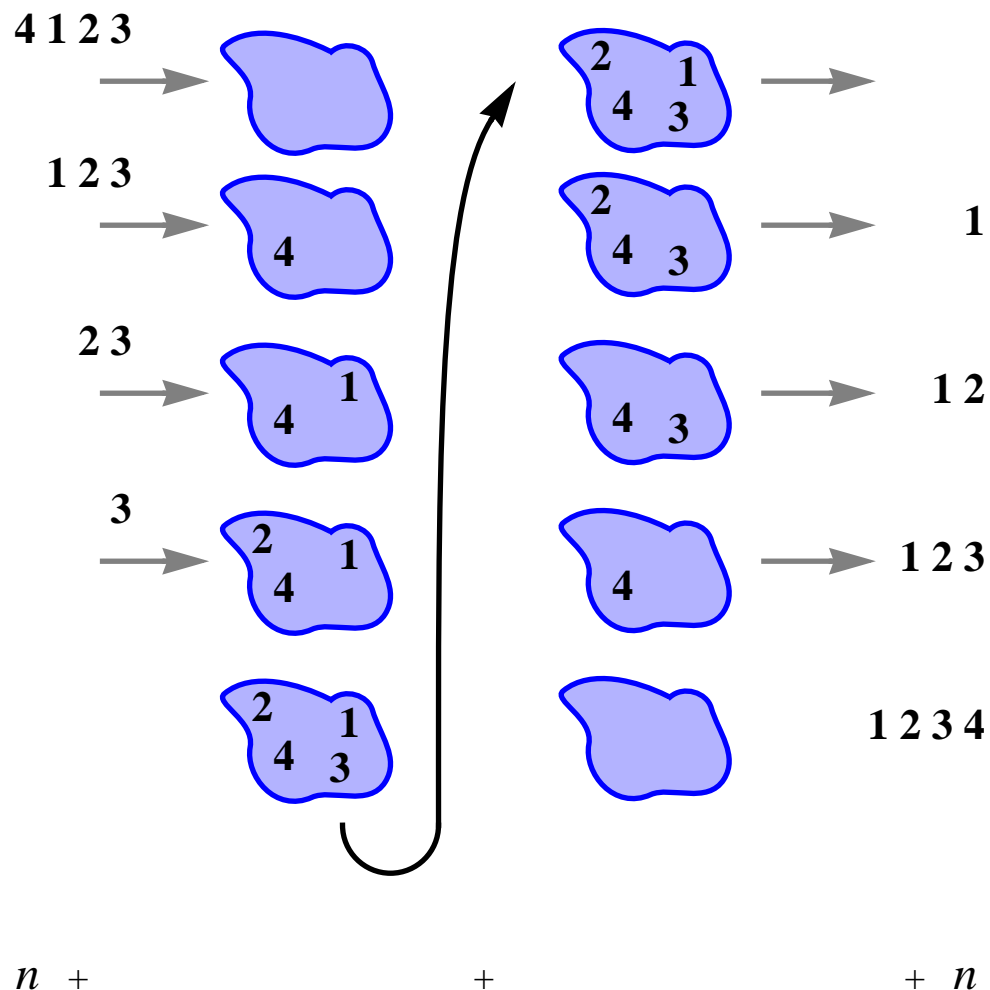
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

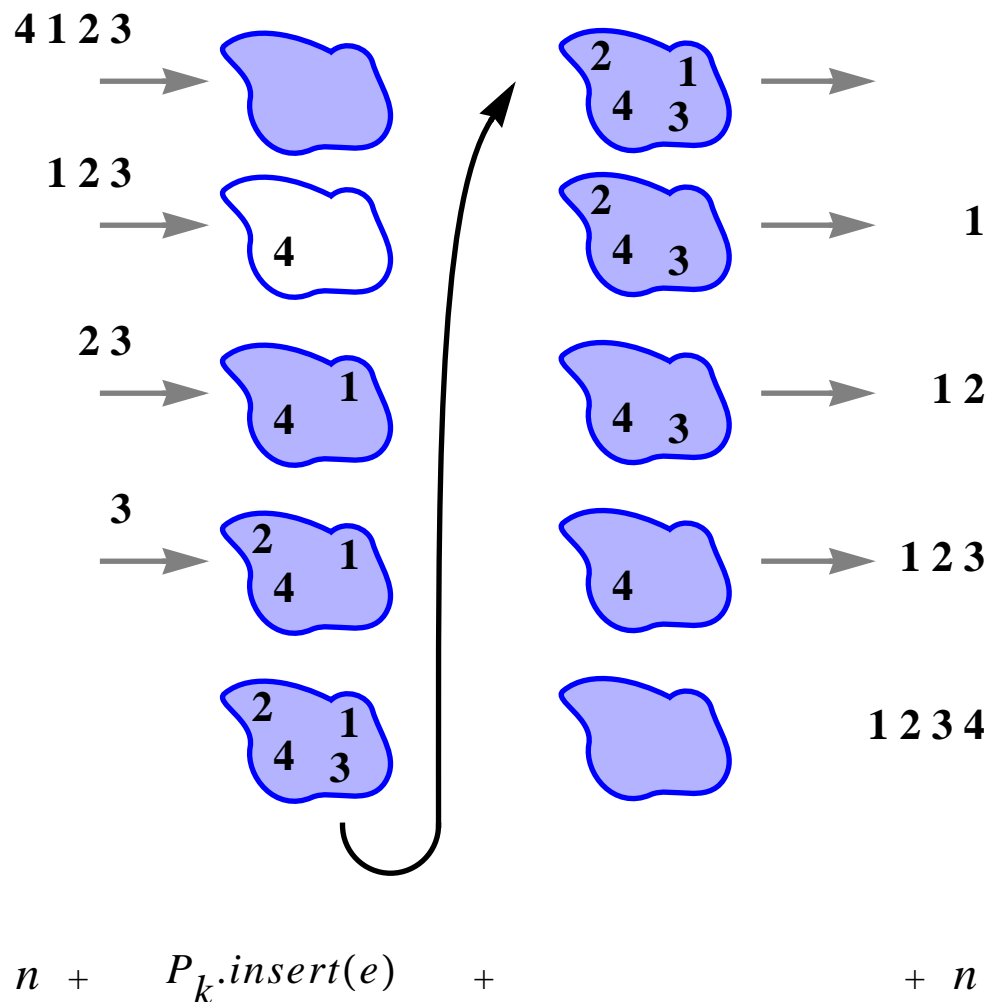
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

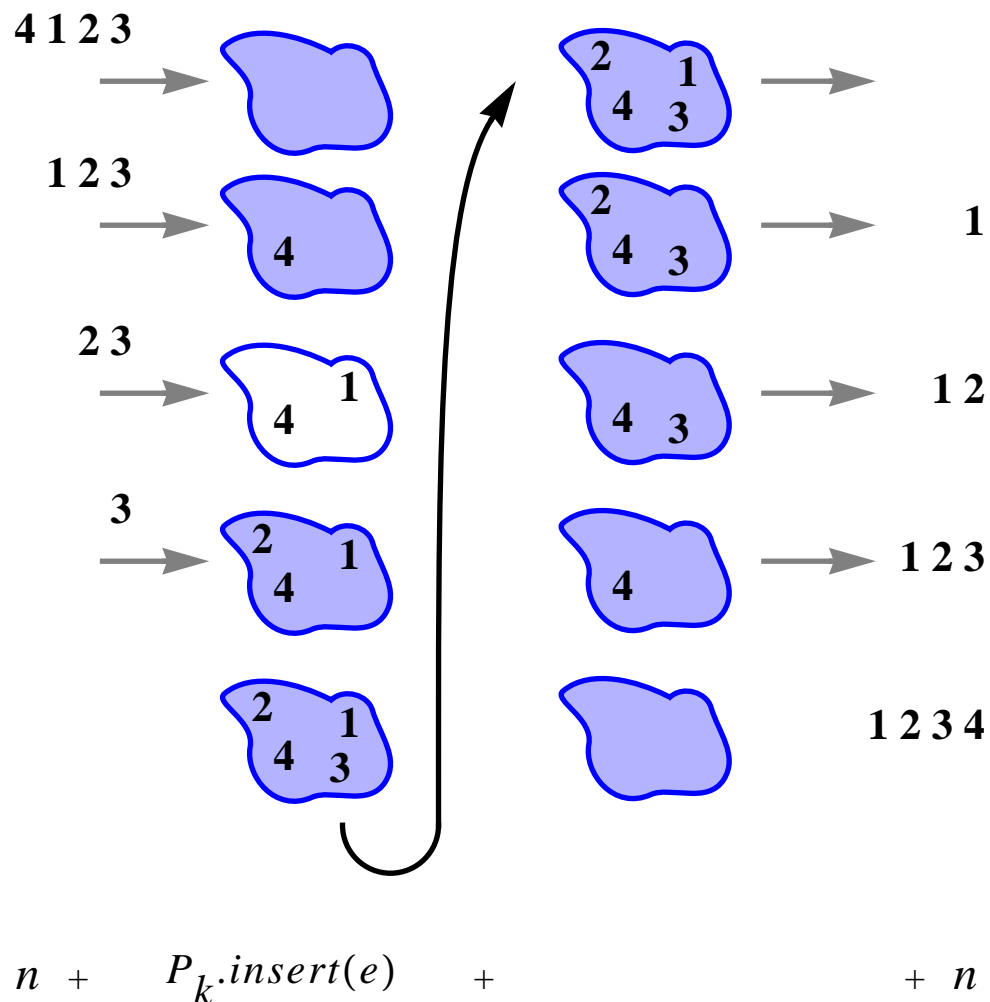
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

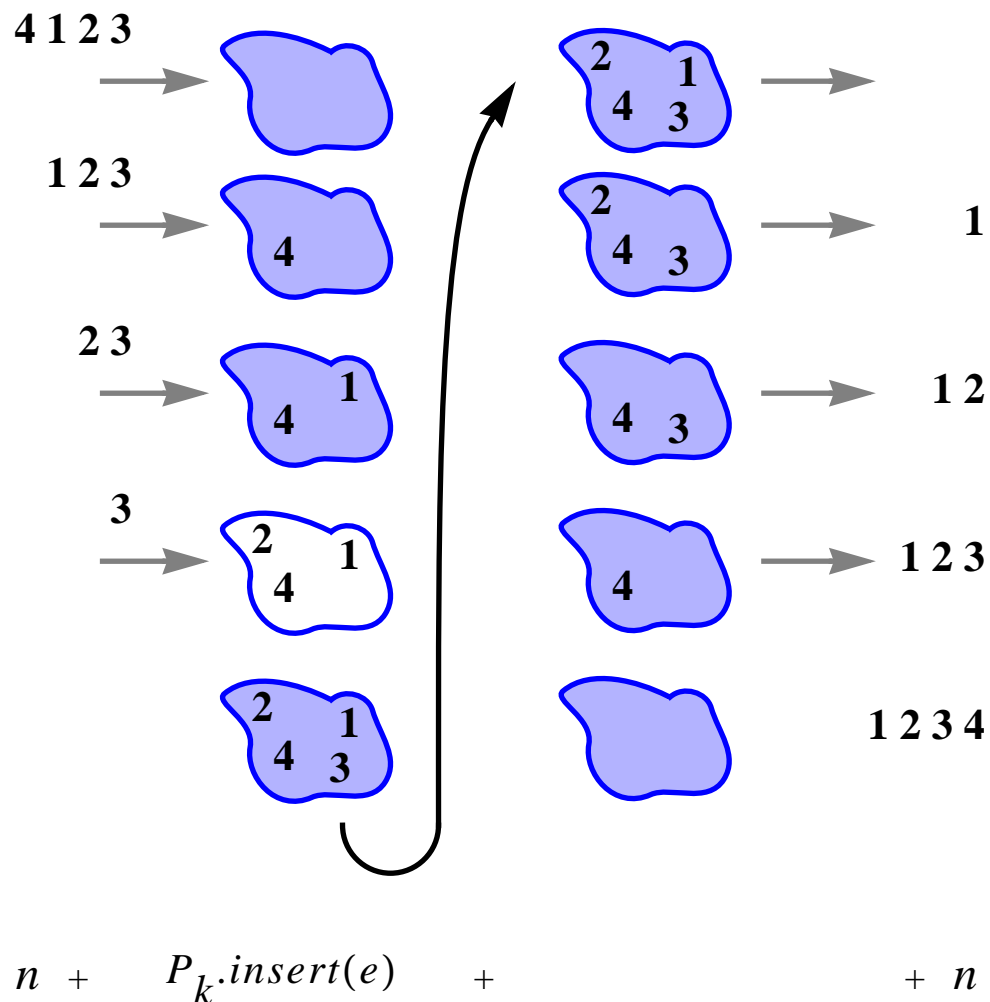
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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)
```

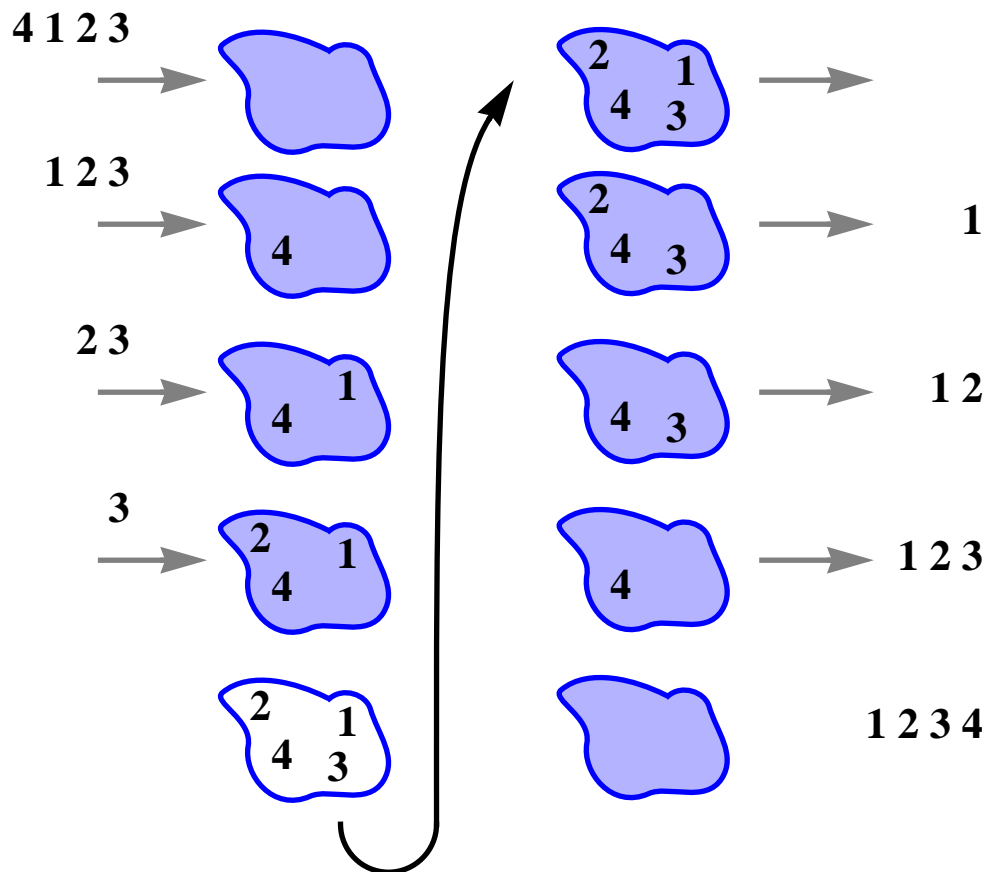
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



$$n + P_k \cdot insert(e) + n$$

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)
```

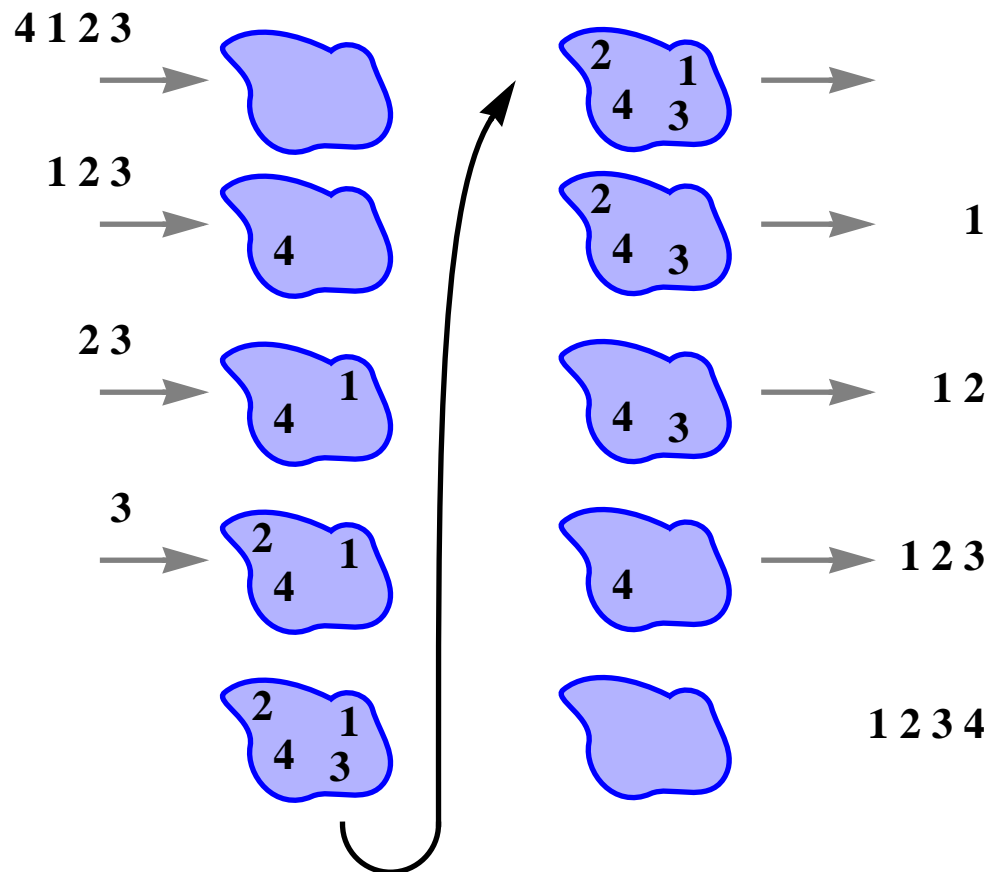
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



$$n + \sum_{k=1}^n P_k \cdot \text{insert}(e) + \dots + n$$

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)
```

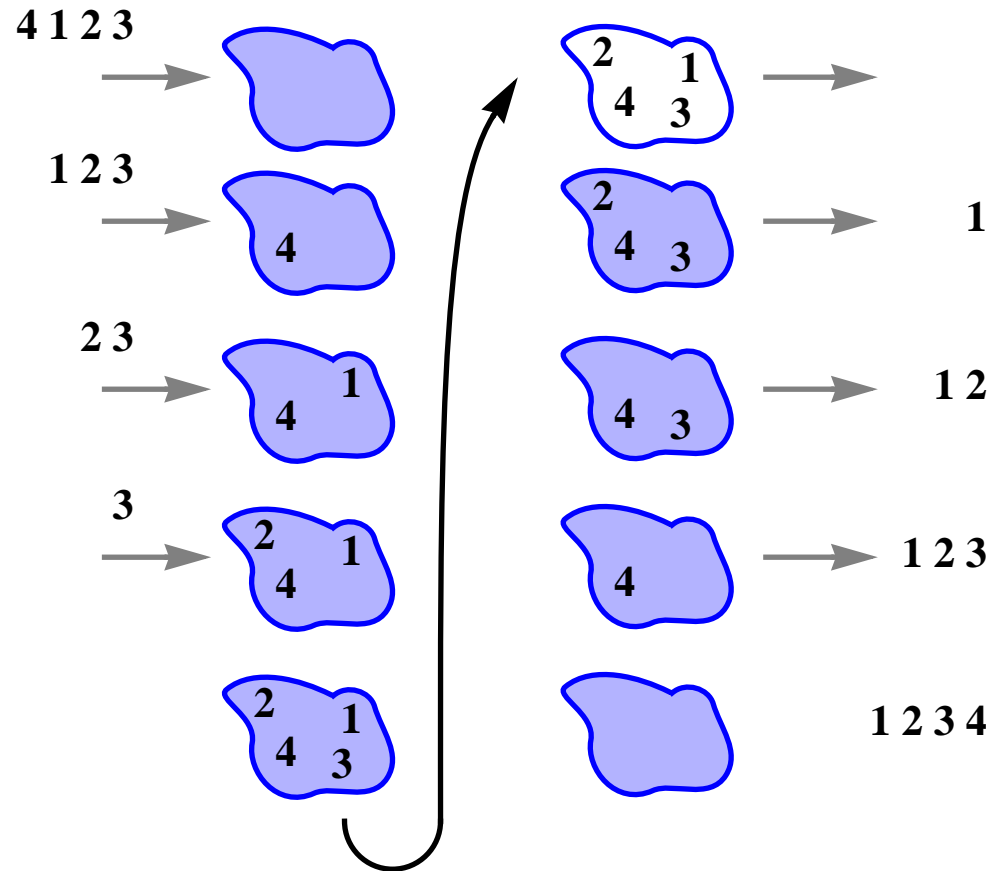
```
b) while (! PQ.isEmpty())
```

```
// 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 \cdot \text{insert}(e) + P_k \cdot \text{remMin}(\) + n$$

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)
```

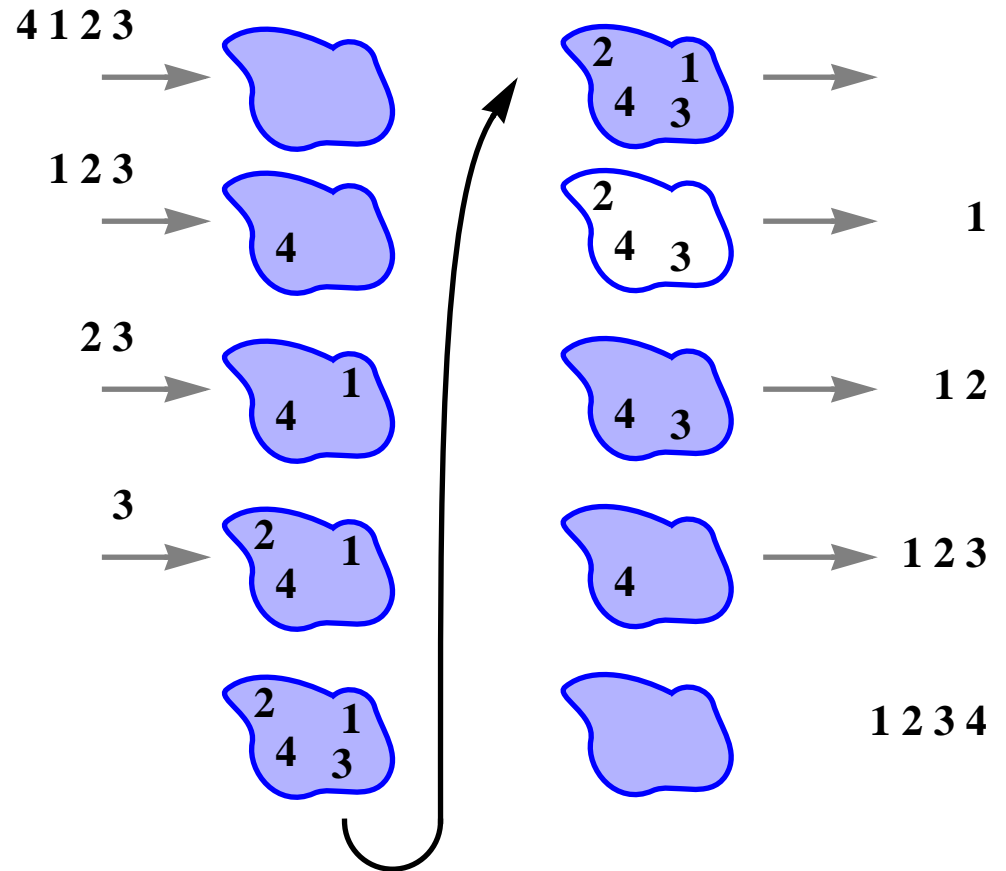
```
b) while (! PQ.isEmpty())
```

```
// 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 \cdot \text{insert}(e) + P_k \cdot \text{remMin}(\) + n$$

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)
```

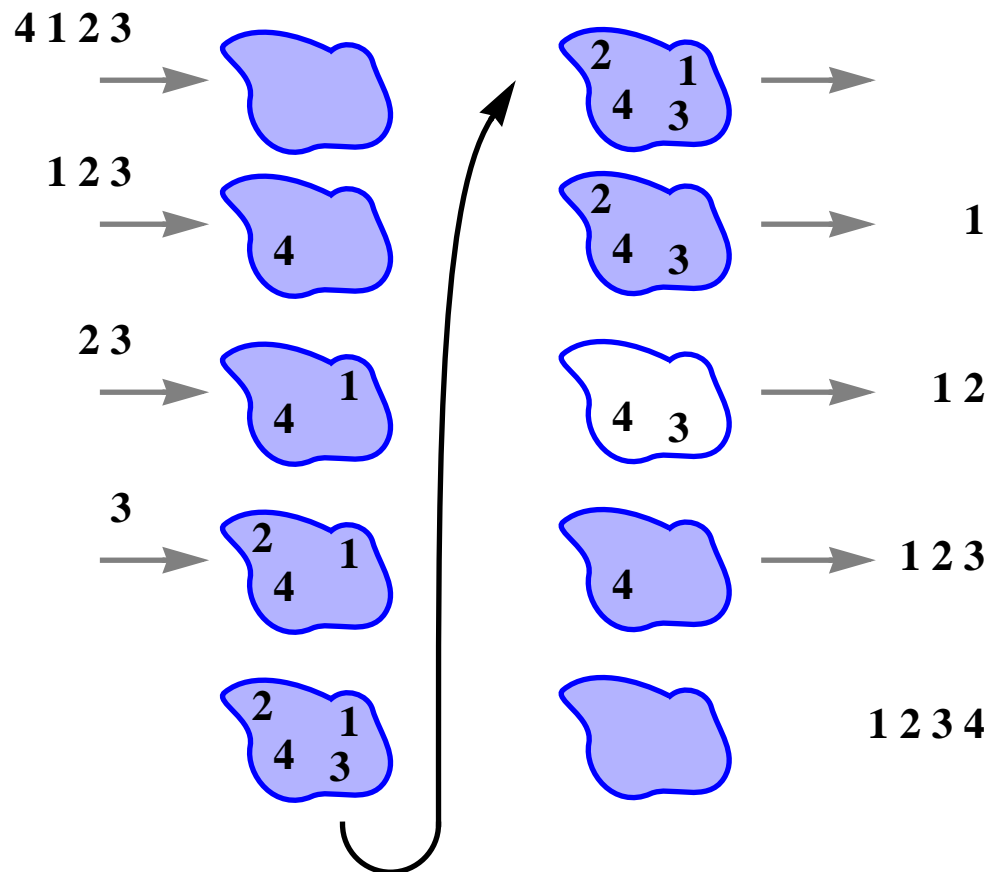
```
b) while (! PQ.isEmpty())
```

```
// 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 \cdot \text{insert}(e) + P_k \cdot \text{remMin}(\) + n$$

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)
```

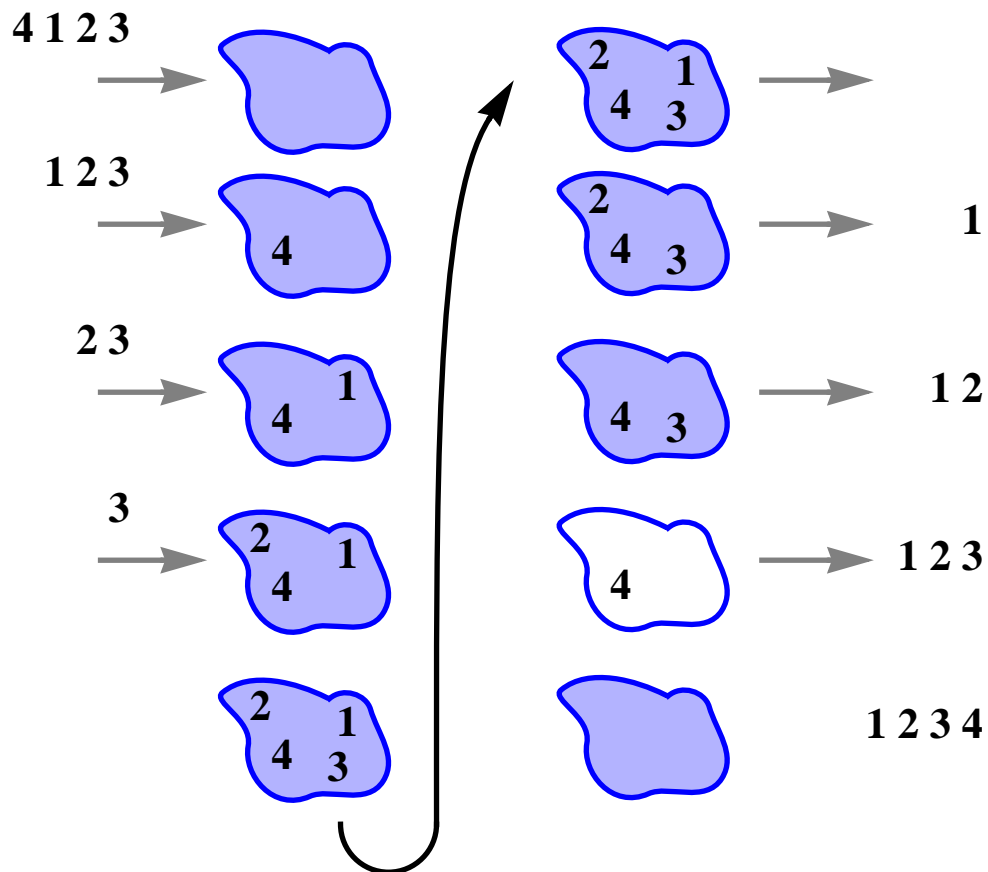
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```



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

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)
```

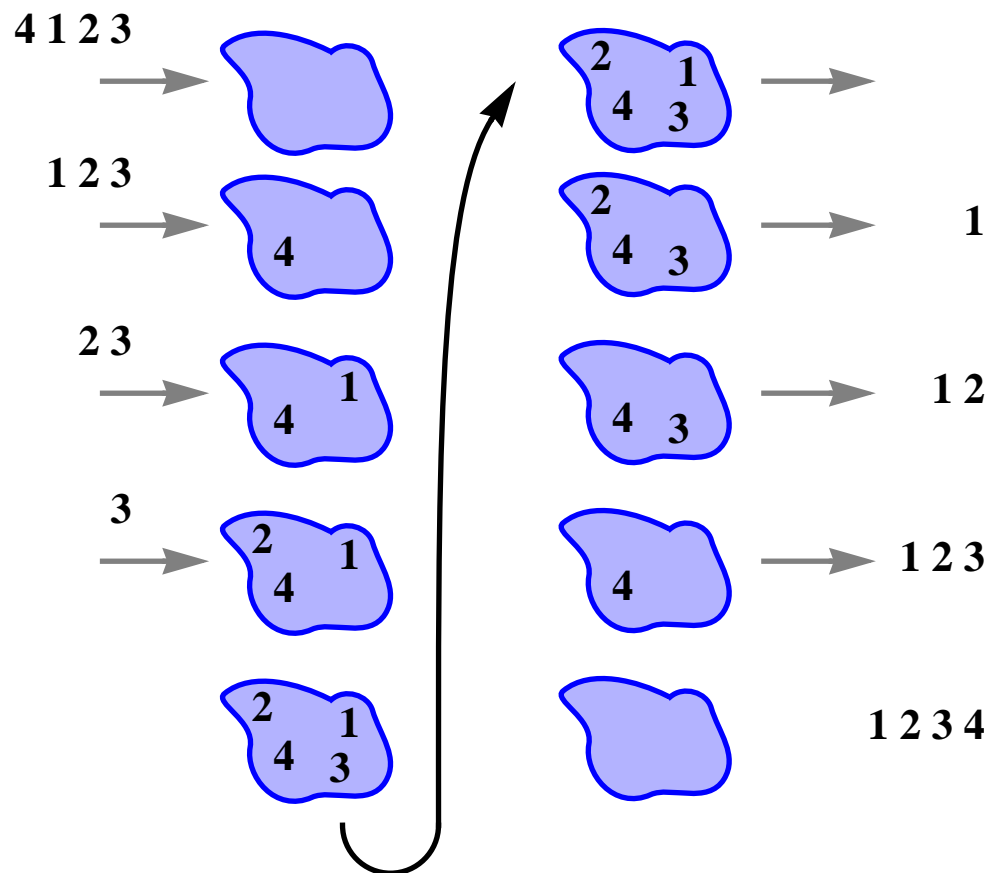
```
b) while (! PQ.isEmpty())
```

```
// fjern ett og ett element fra PQ
```

```
    e = PQ.removeMin()
```

```
// og sett dem inn på slutten av S = O(1)
```

```
    S.insertLast(e)
```

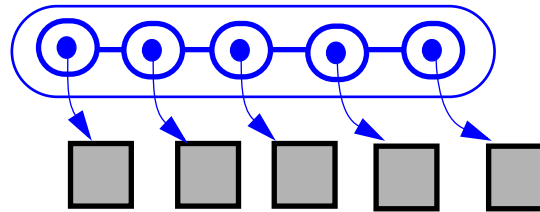


$$n + \sum_{k=1}^n P_k \cdot \text{insert}(e) + \sum_{k=n}^1 P_k \cdot \text{remMin}(\) + n$$

IV. Implementasjon av PriorityQueue – *med Sequence*

I DATA REPRESENTASJON

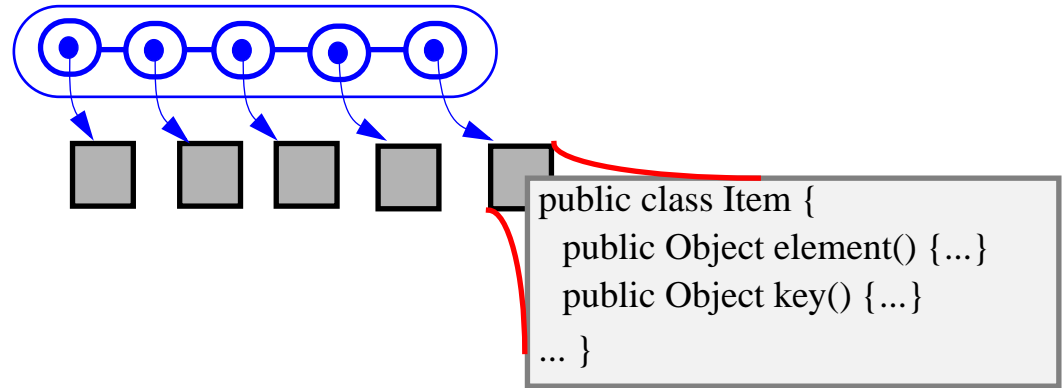
Sequence



IV. Implementasjon av PriorityQueue – *med Sequence*

I DATA REPRESENTASJON

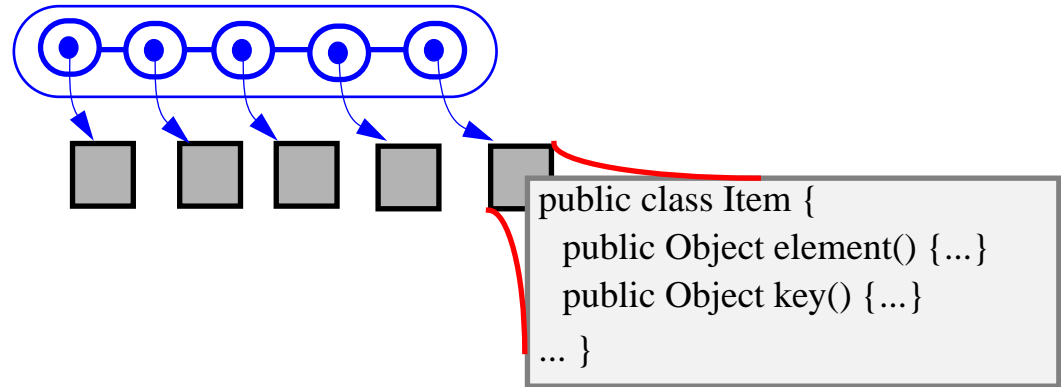
Sequence, der hver Position
lagrer en Item



IV. Implementasjon av PriorityQueue – *med Sequence*

I DATA REPRESENTASJON

Sequence, der hver Position
lagrer en Item



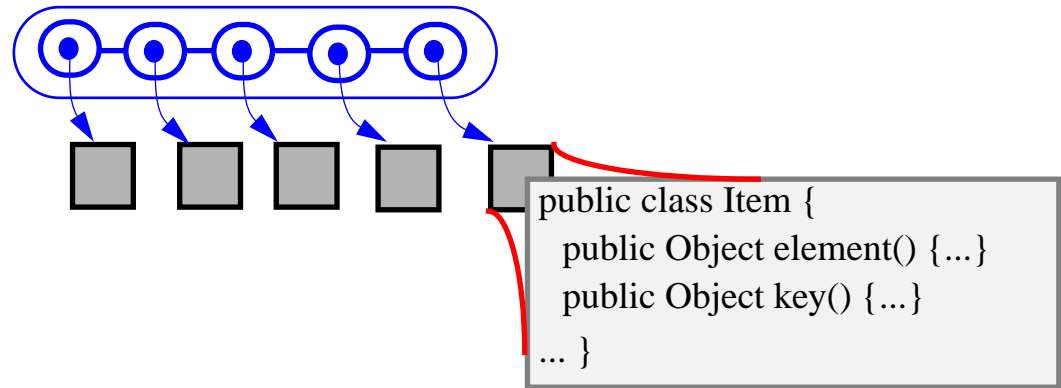
II DATA STRUKTUR

```
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; }  
}
```

IV. Implementasjon av PriorityQueue – *med Sequence*

I DATA REPRESENTASJON

Sequence, der hver Position
lagrer en Item



II DATA STRUKTUR

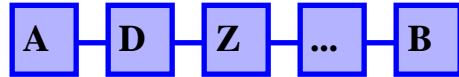
```
class PQSequence implements PriorityQueue {  
    private Sequence S;  
    private Comparator cp;  
    /** @param sq bør være tom sekvens  
        @param c Comparator for sammenlikning av Item med passende key-objekter */  
    public PQSeq(Sequence sq, Comparator c) {  
        S = sq; cp = c;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

III DATA INVARIANT

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

PriorityQueue – *med Sequence*

DATA INVARIANT: INGEN – USORTERT

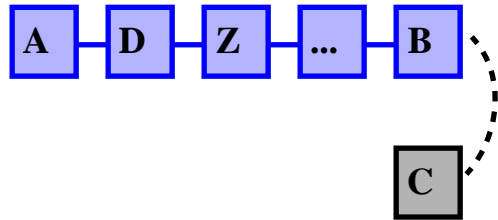


void *insertItem*(k,o)

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

PriorityQueue – *med Sequence*

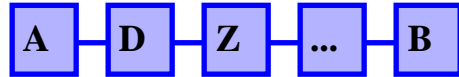
DATA INVARIANT: INGEN – USORTERT



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

PriorityQueue – *med Sequence*

DATA INVARIANT: INGEN – USORTERT



void *insertItem*(k,o)

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

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

```
Position p = sq.first();
```

```
Object o = p.element();
```

```
while ( p != sq.last() ) {
```

```
    p = sq.after(p);
```

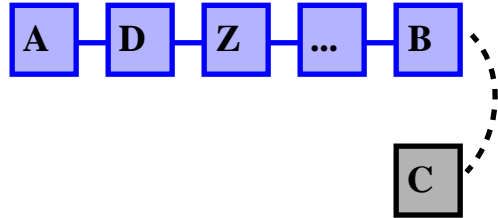
```
    if (cp.LT(p.element(), o)
```

```
        o = p.element(); }
```

```
return ((Item)o).key();
```

PriorityQueue – *med Sequence*

DATA INVARIANT: INGEN – USORTERT



void insertItem(k,o)

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

Object minKey(): finn minste ...

```
Position p = sq.first();
```

```
Object o = p.element();
```

```
while ( p != sq.last() ) {
```

```
    p = sq.after(p);
```

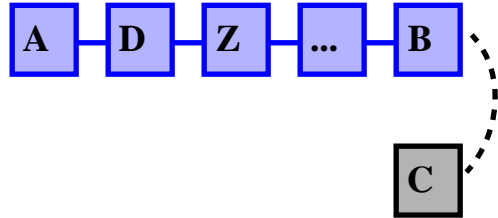
```
    if (cp.LT(p.element(), o)
```

```
        o = p.element(); } }
```

```
return ((Item)o).key();
```

PriorityQueue – *med Sequence*

DATA INVARIANT: INGEN – USORTERT



void insertItem(k,o)

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

Object minKey(): finn minste ...

```
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(); } 
```

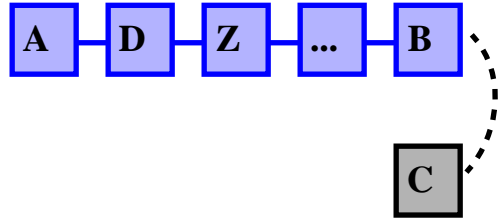
```
return ((Item)o).key();
```

Object minElement() : finn ...

Object removeMin() : finn og fjern ...

PriorityQueue – *med Sequence*

DATA INVARIANT: INGEN – USORTERT



void insertItem(k,o) $O(1)$
sq.insertLast(new Item(k,o))

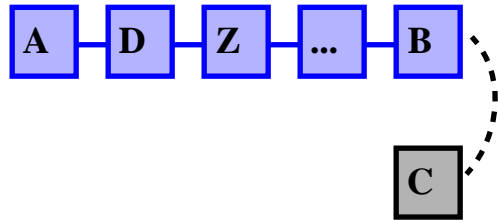
Object minKey(): finn minste ... $\Theta(n)$
Position p = sq.first();
Object o = p.element();
while (p != sq.last()) {
 p = sq.after(p);
 if (cp.LT(p.element(), o)
 o = p.element(); }
return ((Item)o).key();

Object minElement() : finn ... $\Theta(n)$

Object removeMin() : finn og fjern ... $\Theta(n)$

PriorityQueue – med Sequence

DATA INVARIANT: INGEN – USORTERT



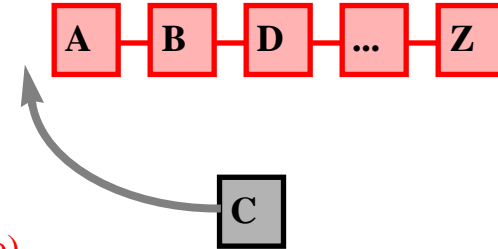
void insertItem(k,o) $O(1)$
 sq.insertLast(new Item(k,o))

Object minKey(): finn minste ... $\Theta(n)$
 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 ... $\Theta(n)$

Object removeMin() : finn og fjern ... $\Theta(n)$

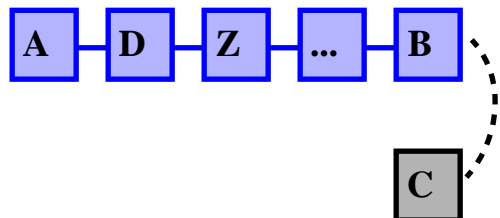
DATA INVARIANT: SORTERT STIGENDE



void insertItem(k,o)
 Item ny= new Item(k,o);
 if (sq.isEmpty()) sq.insertFirst(ny)
 else if (cp.isLessThanOrEqualTo(ny, sq.first().element()))
 sq.insertFirst(ny)
 else if (cp.isGreaterThanOrEqualTo(ny, sq.last().element()))
 sq.insertLast(ny)
 else Position c = sq.first()
 while (cp.isGreaterThanOrEqualTo(ny, c.element()) c= sq.after(c)
 sq.insertBefore(c,ny)

PriorityQueue – med Sequence

DATA INVARIANT: INGEN – USORTERT



void insertItem(k,o) $O(1)$
sq.insertLast(new Item(k,o))

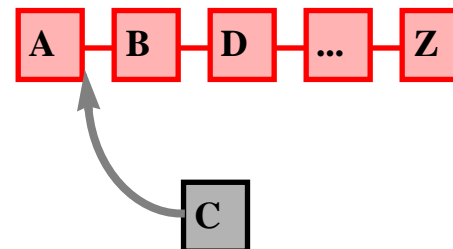
Object minKey(): finn minste ... $\Theta(n)$

```
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 ... $\Theta(n)$

Object removeMin() : finn og fjern ... $\Theta(n)$

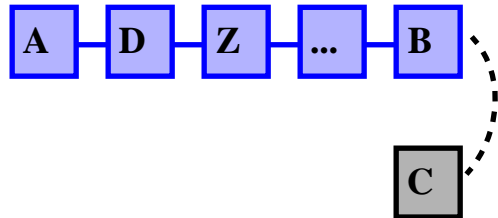
DATA INVARIANT: SORTERT STIGENDE



void insertItem(k,o)
Item ny= new Item(k,o);
if (sq.isEmpty()) sq.insertFirst(ny)
else if (cp.isLessThanOrEqualTo(ny, sq.first().element()))
 sq.insertFirst(ny)
else if (cp.isGreaterThanOrEqualTo(ny, sq.last().element()))
 sq.insertLast(ny)
else Position c = sq.first()
 while (cp.isGreaterThanOrEqualTo(ny, c.element()) c= sq.after(c)
 sq.insertBefore(c,ny)

PriorityQueue – *med Sequence*

DATA INVARIANT: INGEN – USORTERT



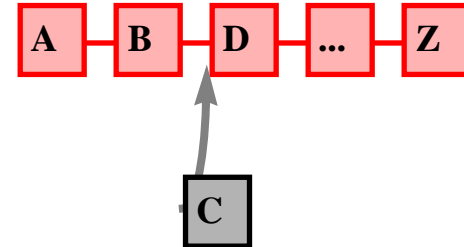
void insertItem(k,o) $O(1)$
sq.insertLast(new Item(k,o))

Object minKey(): finn minste ... $\Theta(n)$
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 ... $\Theta(n)$

Object removeMin() : finn og fjern ... $\Theta(n)$

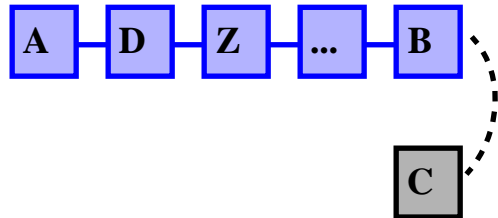
DATA INVARIANT: SORTERT STIGENDE



void insertItem(k,o)
Item ny= new Item(k,o);
if (sq.isEmpty()) sq.insertFirst(ny)
else if (cp.isLessThanOrEqualTo(ny, sq.first().element()))
 sq.insertFirst(ny)
else if (cp.isGreaterThanOrEqualTo(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)

PriorityQueue – med Sequence

DATA INVARIANT: INGEN – USORTERT



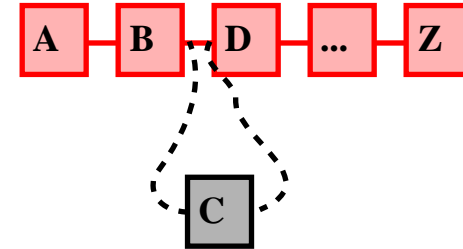
void insertItem(k,o) $O(1)$
sq.insertLast(new Item(k,o))

Object minKey(): finn minste ... $\Theta(n)$
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 ... $\Theta(n)$

Object removeMin() : finn og fjern ... $\Theta(n)$

DATA INVARIANT: SORTERT STIGENDE



void insertItem(k,o)
Item ny= new Item(k,o);
if (sq.isEmpty()) sq.insertFirst(ny)
else if (cp.isLessThanOrEqualTo(ny, sq.first().element()))
 sq.insertFirst(ny)
else if (cp.isGreaterThanOrEqualTo(ny, sq.last().element()))
 sq.insertLast(ny)
else Position c = sq.first()
 while (cp.isGreaterThanOrEqualTo(ny, c.element()) c= sq.after(c)
 sq.insertBefore(c,ny)

PriorityQueue – med Sequence

DATA INVARIANT: INGEN – USORTERT



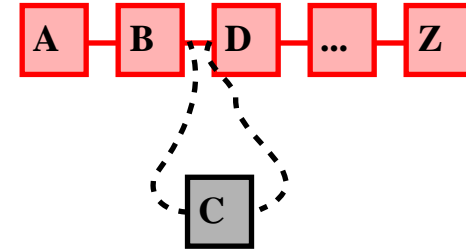
void insertItem(k,o) $O(1)$
 sq.insertLast(new Item(k,o))

Object minKey(): finn minste ... $\Theta(n)$
 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 ... $\Theta(n)$

Object removeMin() : finn og fjern ... $\Theta(n)$

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.isLessThanOrEqualTo(ny, sq.first().element() )
    sq.insertFirst(ny)
else if ( cp.isGreaterThanOrEqualTo(ny, sq.last().element() )
    sq.insertLast(ny)
else Position c = sq.first()
    while (cp.isGreaterThanOrEqualTo(ny, c.element() ) c= sq.after(c)
    sq.insertBefore(c,ny)
```

Object minKey() $O(1)$

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

Object minElement() $O(1)$

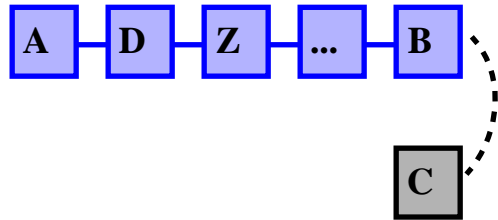
```
return ((Item) sq.first().element() ) . element()
```

Object removeMin() $O(1)$

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

PriorityQueue – med Sequence

DATA INVARIANT: INGEN – USORTERT



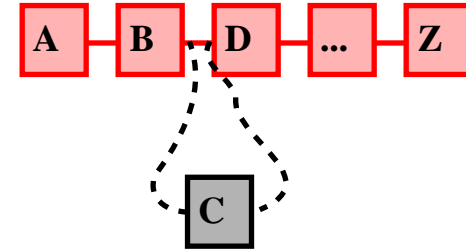
void insertItem(k,o) $O(1)$
 sq.insertLast(new Item(k,o))

Object minKey(): finn minste ... $\Theta(n)$
 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 ... $\Theta(n)$

Object removeMin() : finn og fjern ... $\Theta(n)$

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.isLessThanOrEqualTo(ny, sq.first().element() ) )
    sq.insertFirst(ny)
else if ( cp.isGreaterThanOrEqualTo(ny, sq.last().element() ) )
    sq.insertLast(ny)
else Position c = sq.first()
    while (cp.isGreaterThanOrEqualTo(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()

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

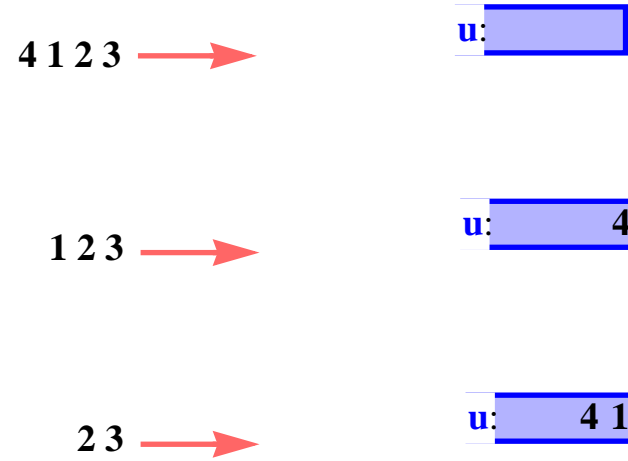
Usortert Sekvens



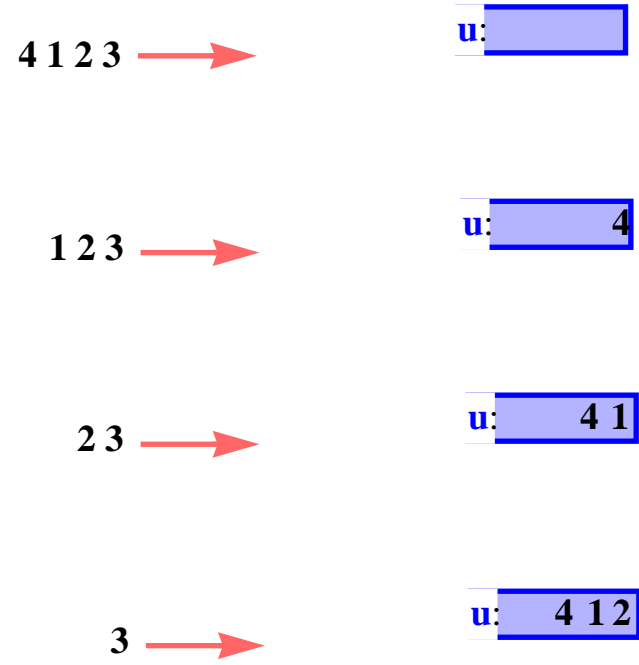
Usortert Sekvens



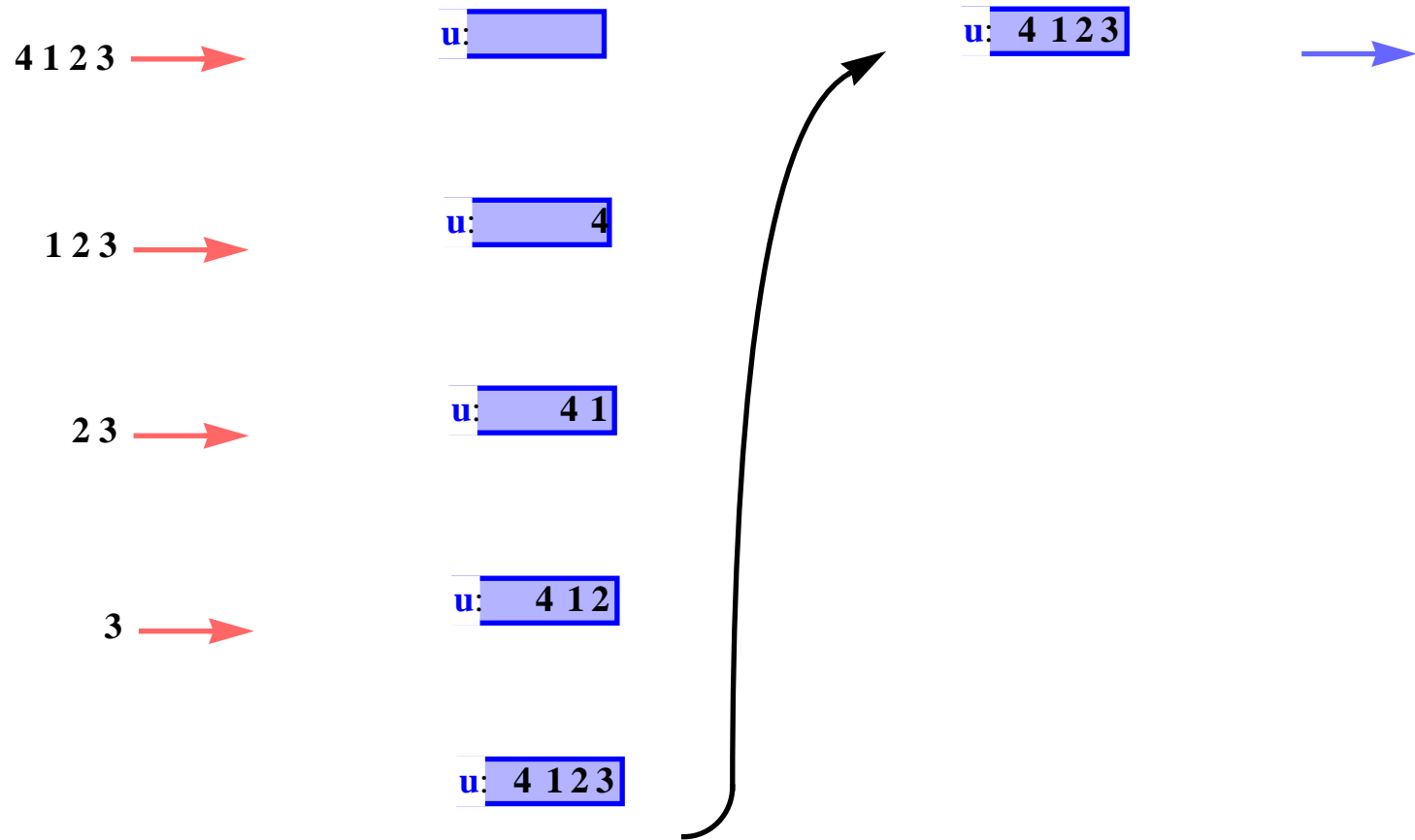
Usortert Sekvens



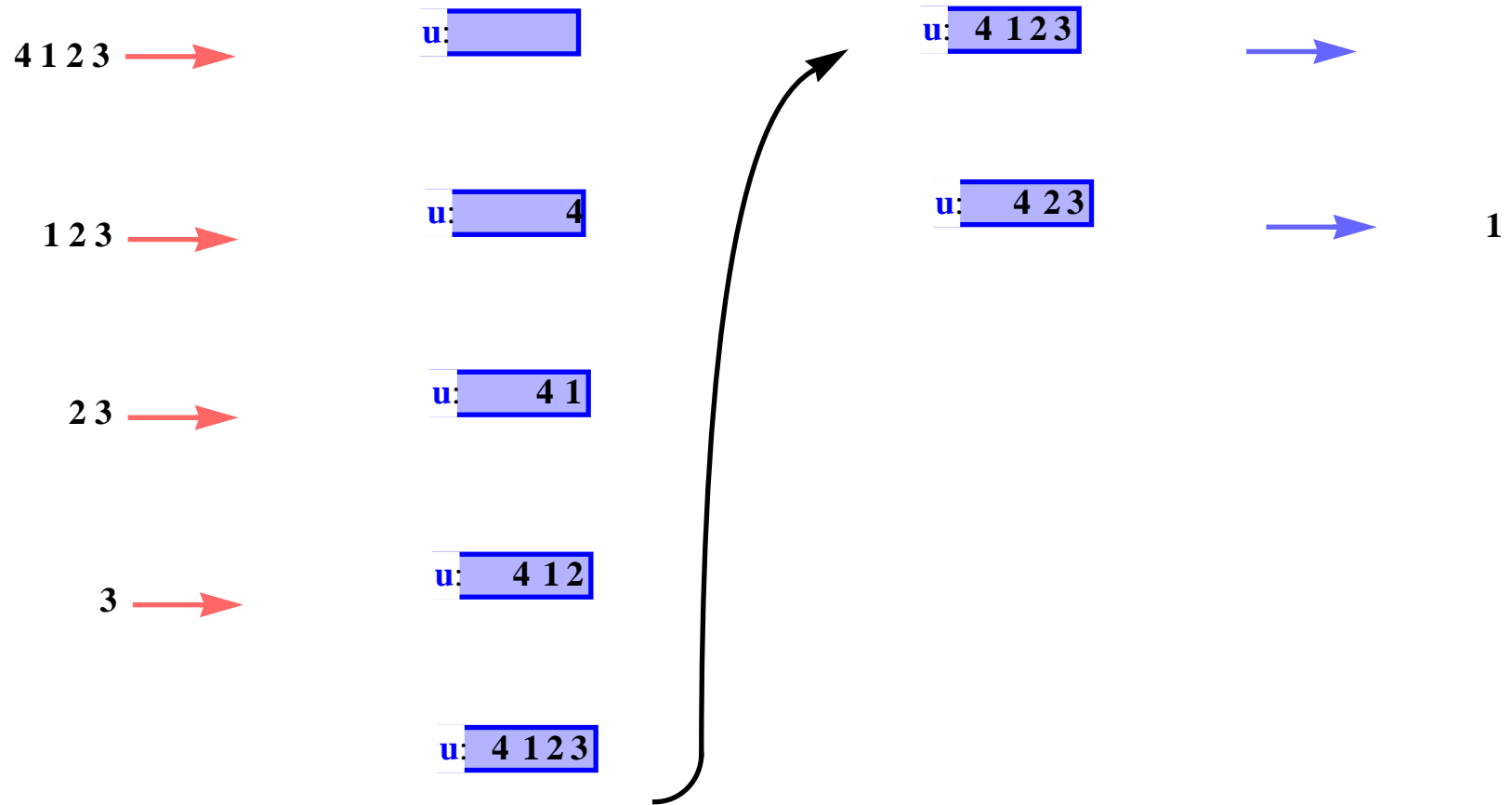
Usortert Sekvens



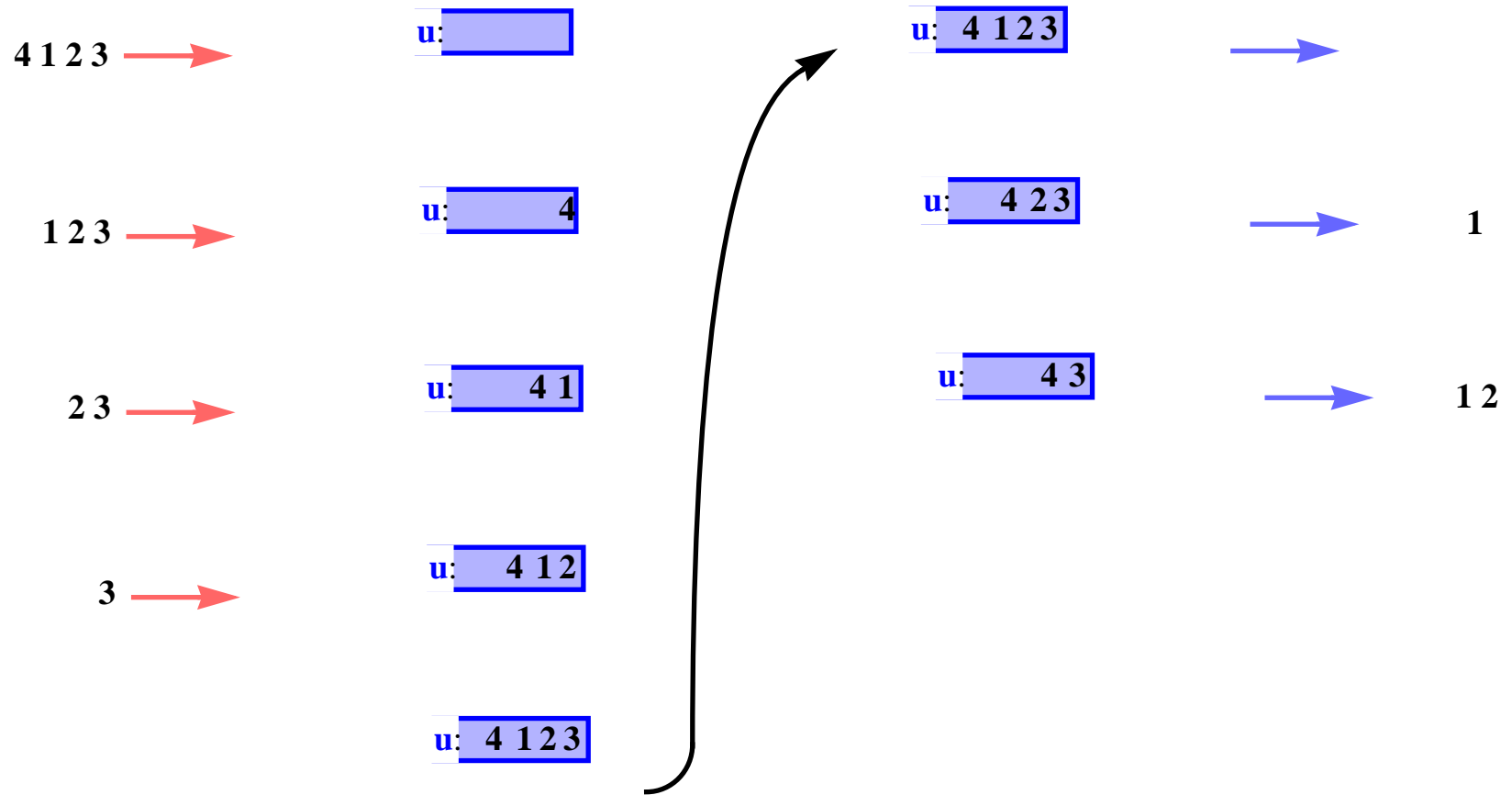
Usortert Sekvens



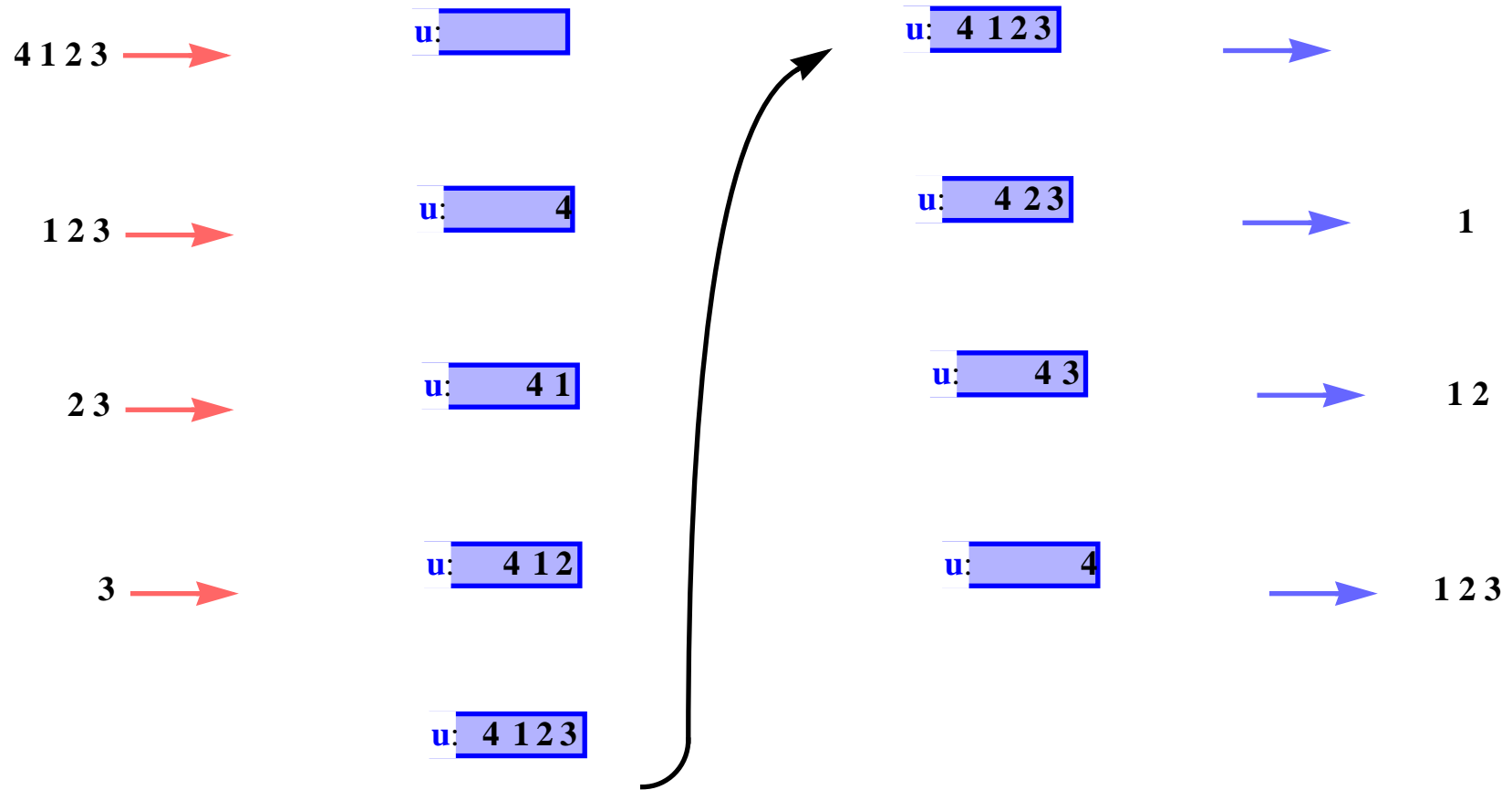
Usortert Sekvens



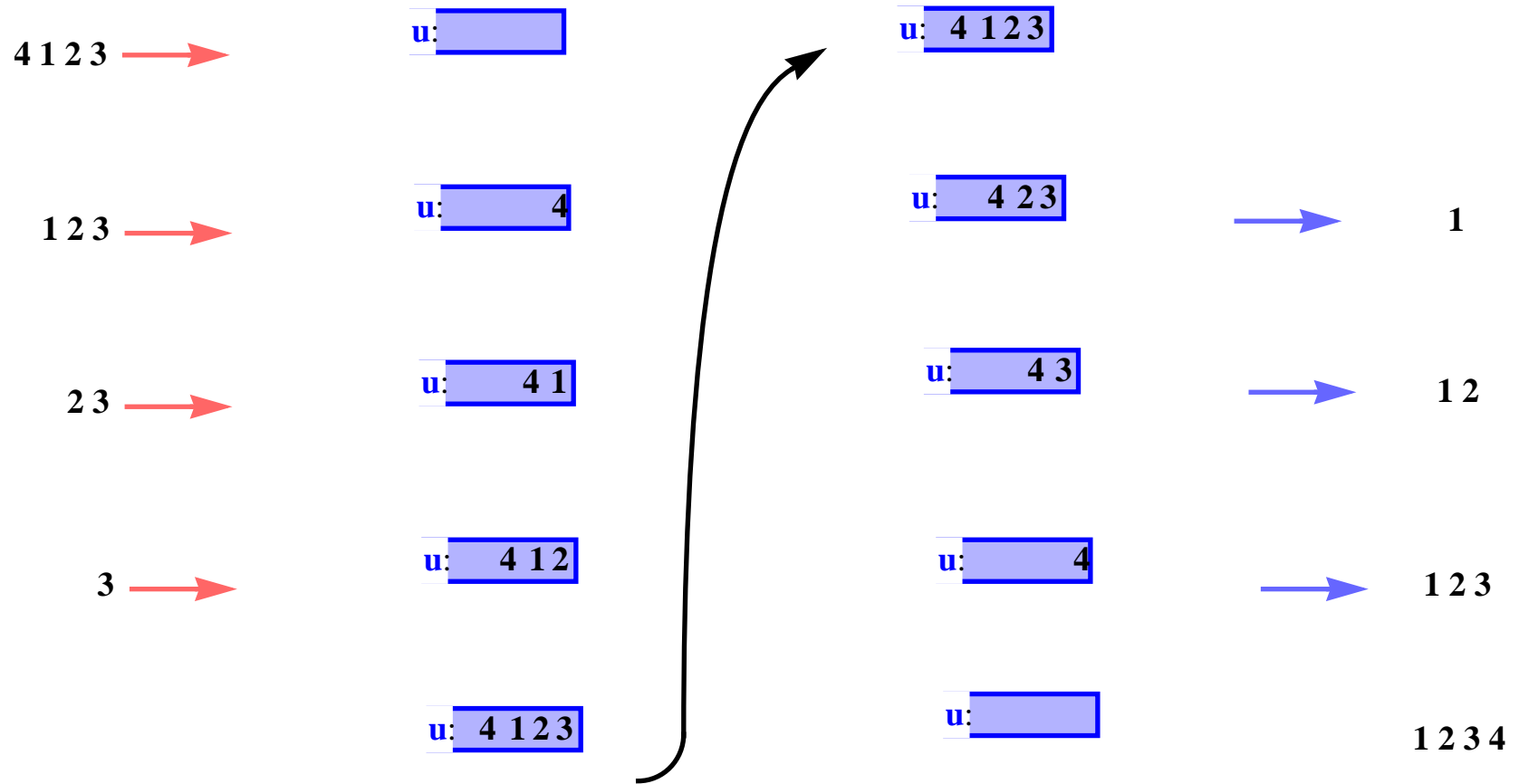
Usortert Sekvens



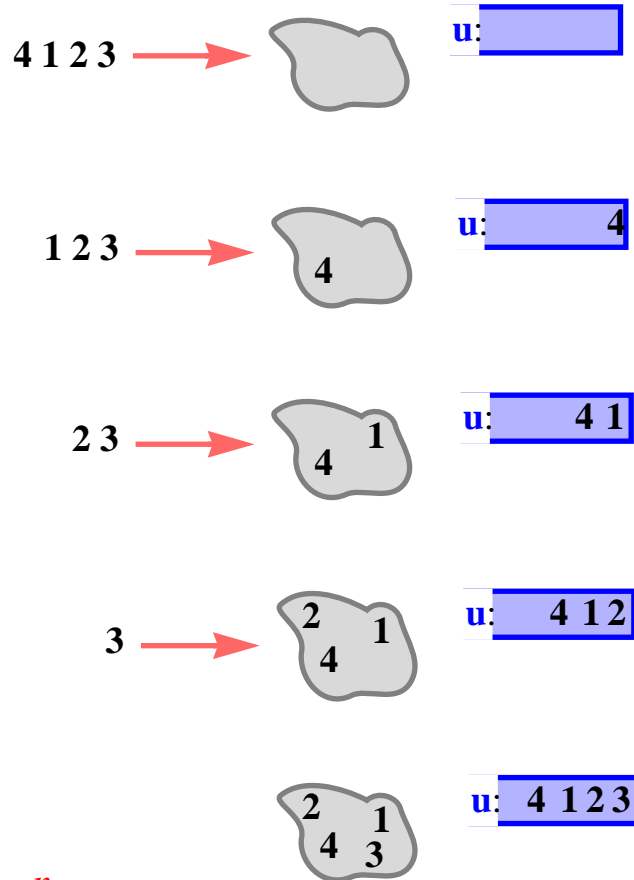
Usortert Sekvens



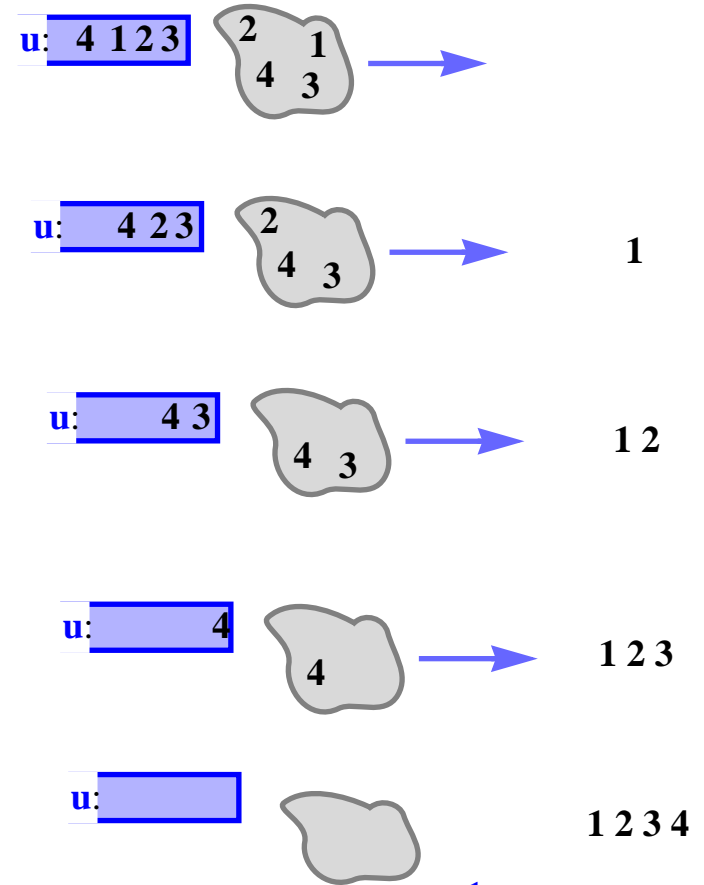
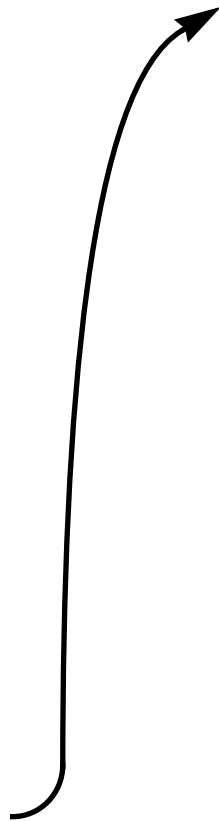
Usortert Sekvens



Usortert Sekvens



$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$



+

$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$

Usortert Sekvens

4 1 2 3 → u: []

1 2 3 → u: [4]

2 3 → u: [4 1]

3 → u: [4 1 2]

u: [4 1 2 3]

$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

Usortert Sekvens

4 1 2 3 → u: []

1 2 3 → u: [4]

2 3 → u: [4 1]

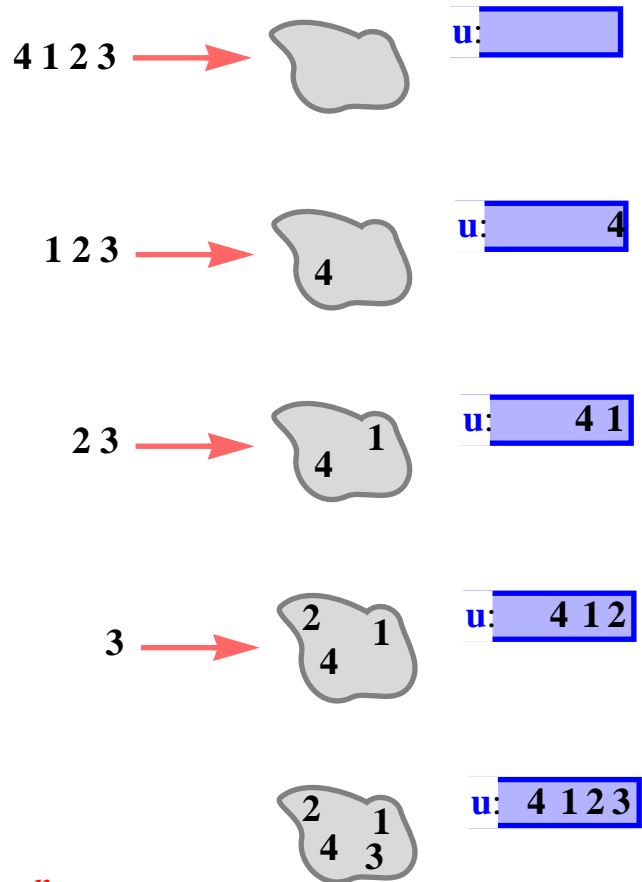
3 → u: [4 1 2]

u: [4 1 2 3]

$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

4 1 2 3 → n → u: [4 1 2 3]

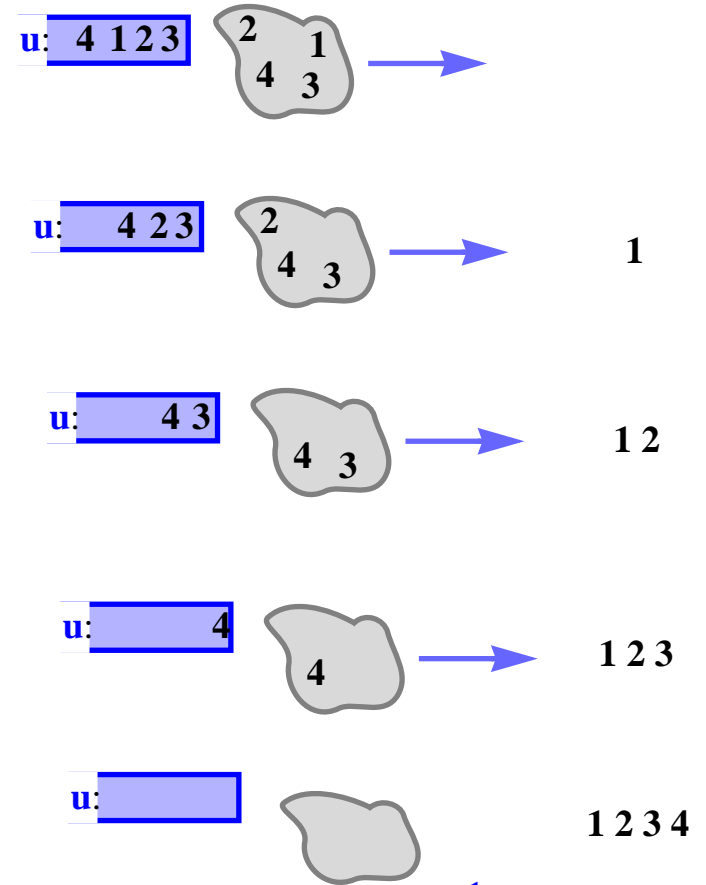
Usortert Sekvens



$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$



+



$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$

Usortert Sekvens

u: 4 1 2 3



u: 4 2 3



1

u: 4 3



1 2

u: 4



1 2 3

u:

1 2 3 4

$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

+

$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$

4 1 2 3



u: 4 1 2 3



1 2 3 4

Usortert Sekvens

u: 4 1 2 3



u: 4 2 3



1

u: 4 3



1 2

u: 4



1 2 3

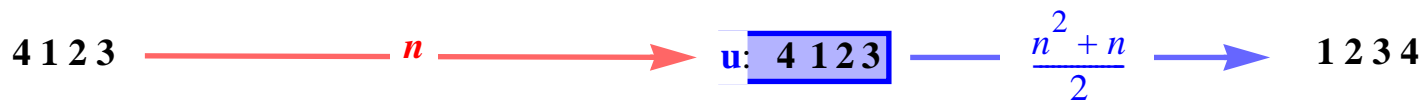
u:

1 2 3 4

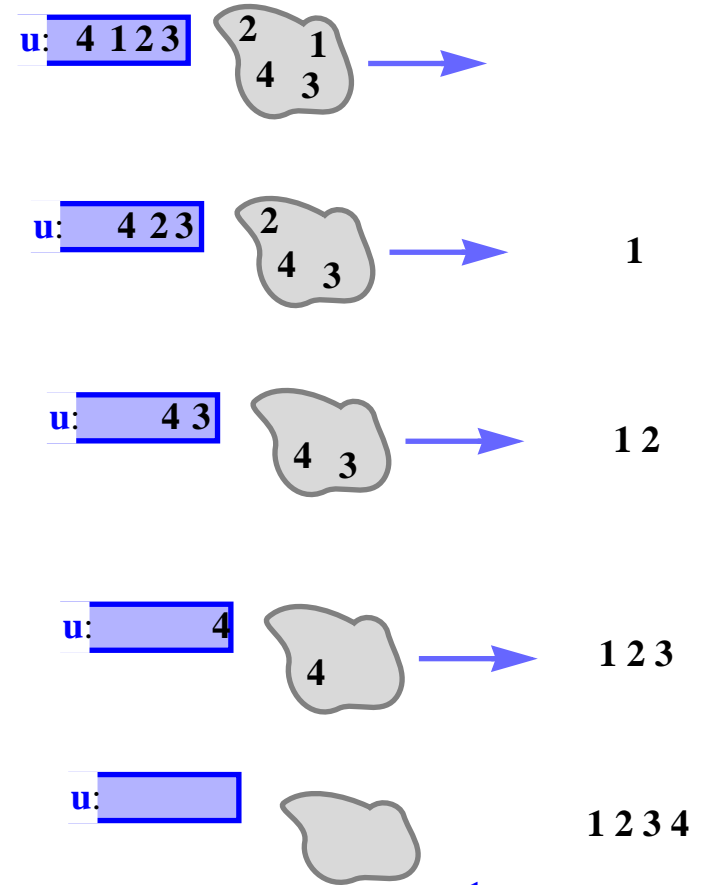
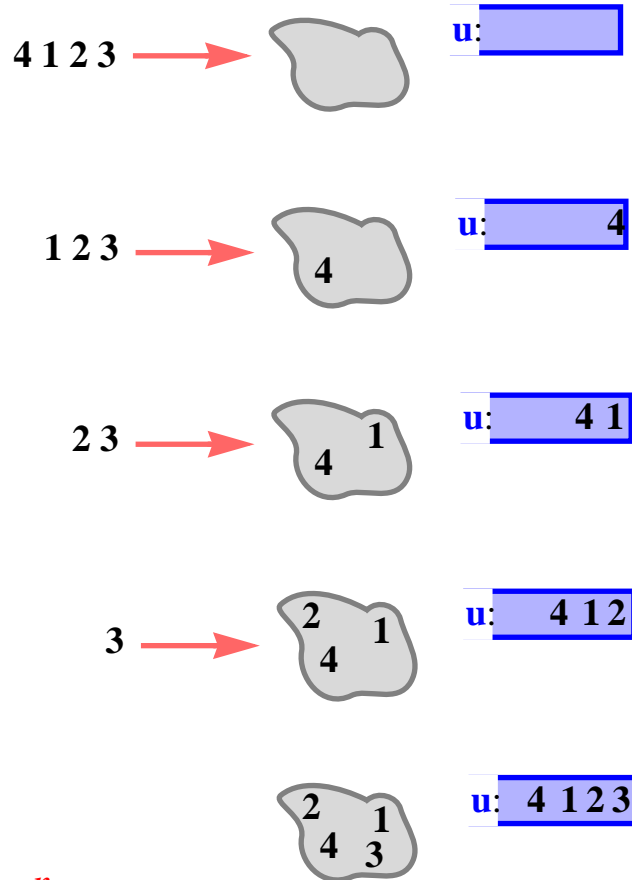
$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

+

$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$



Selection Sort



$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

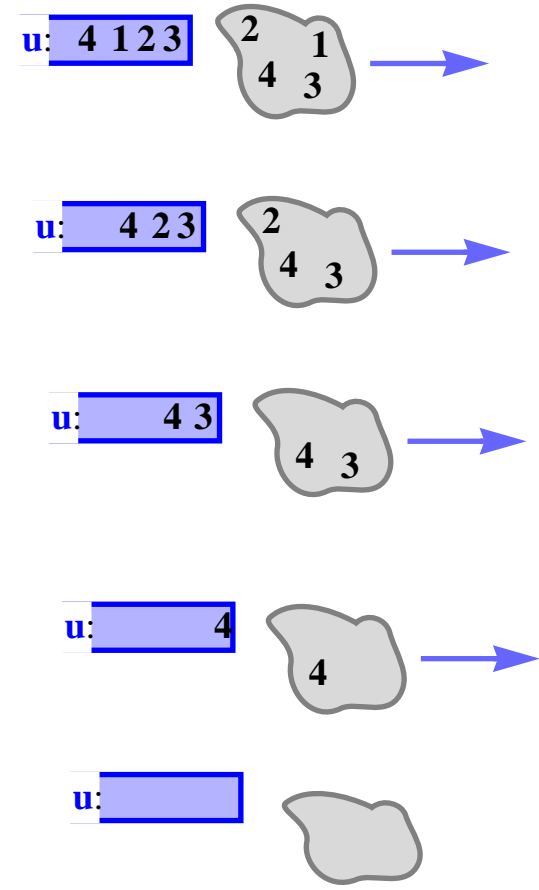
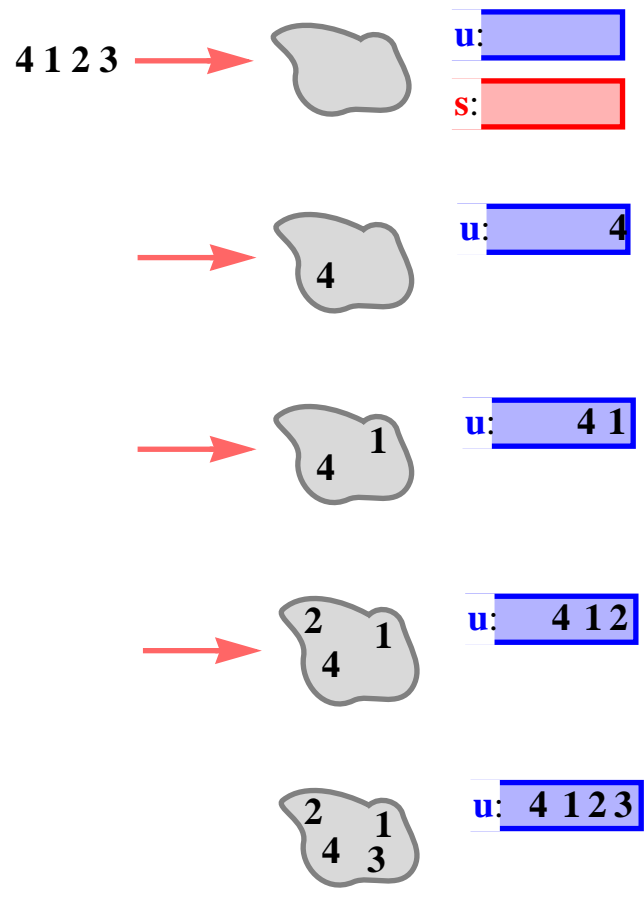
+

$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$

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

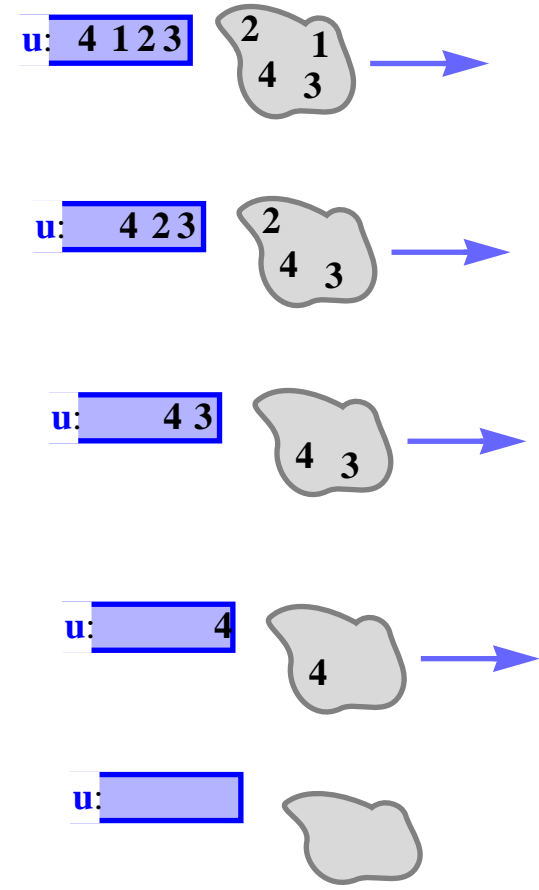
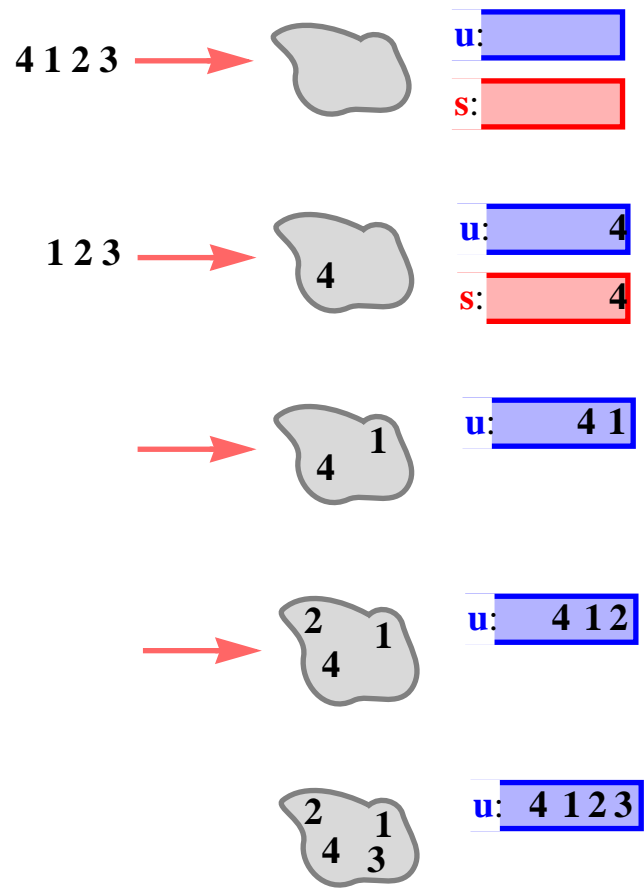
/

Selection Sort



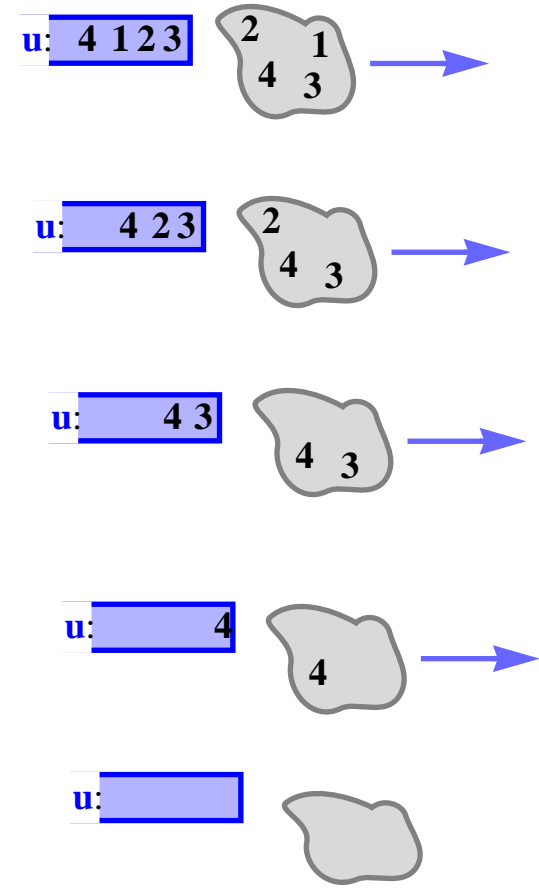
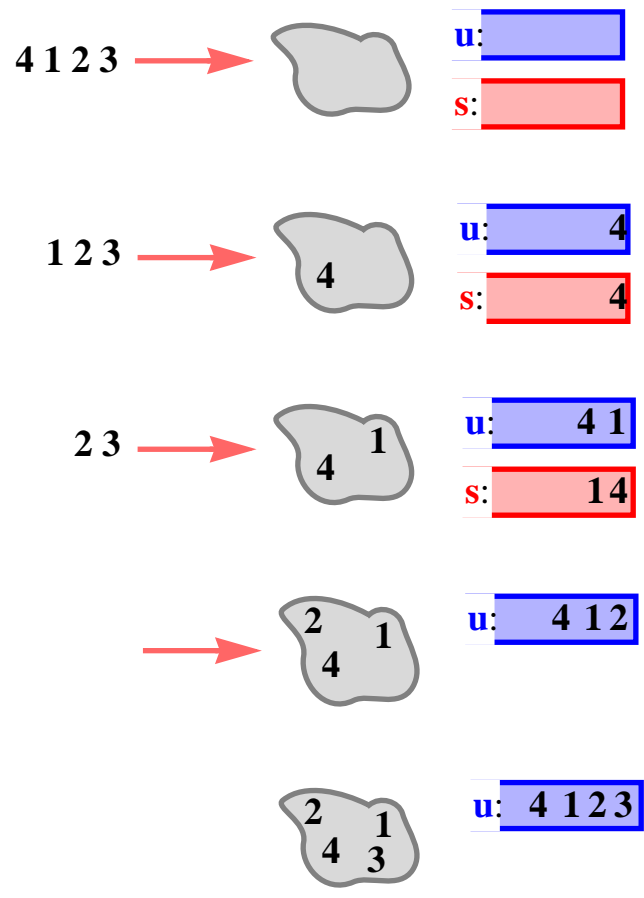
/

Selection Sort



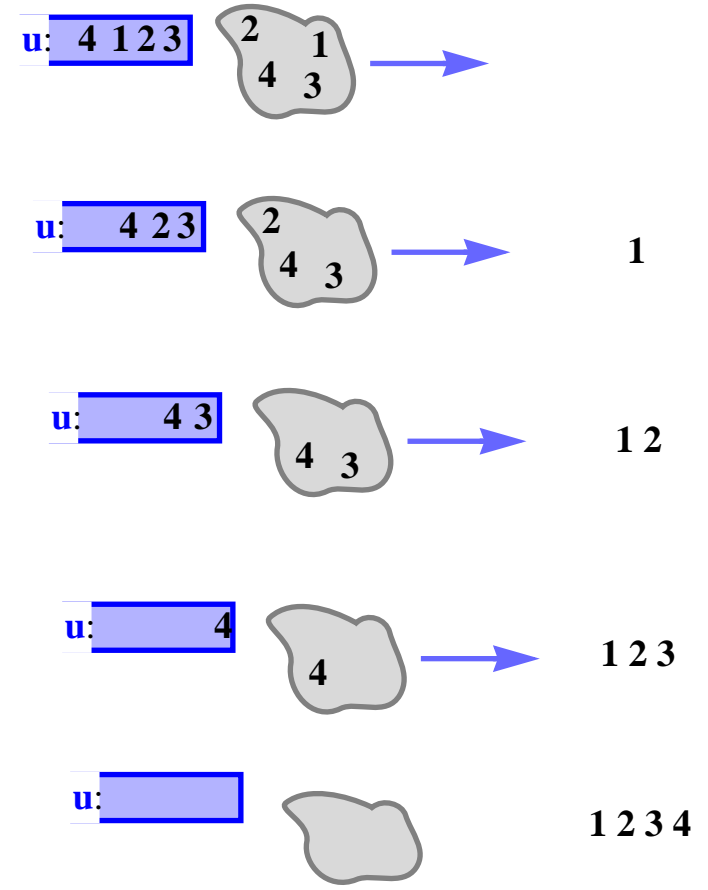
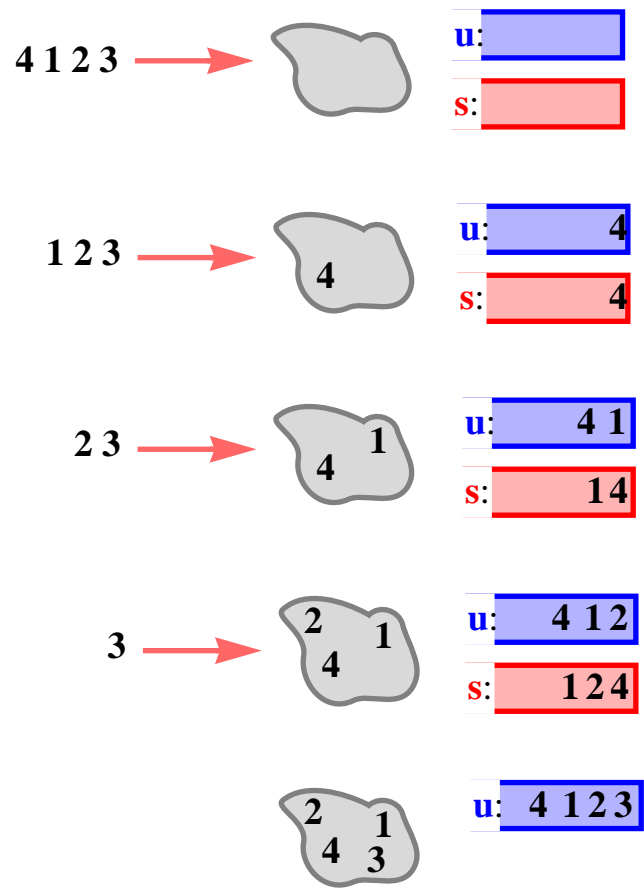
/

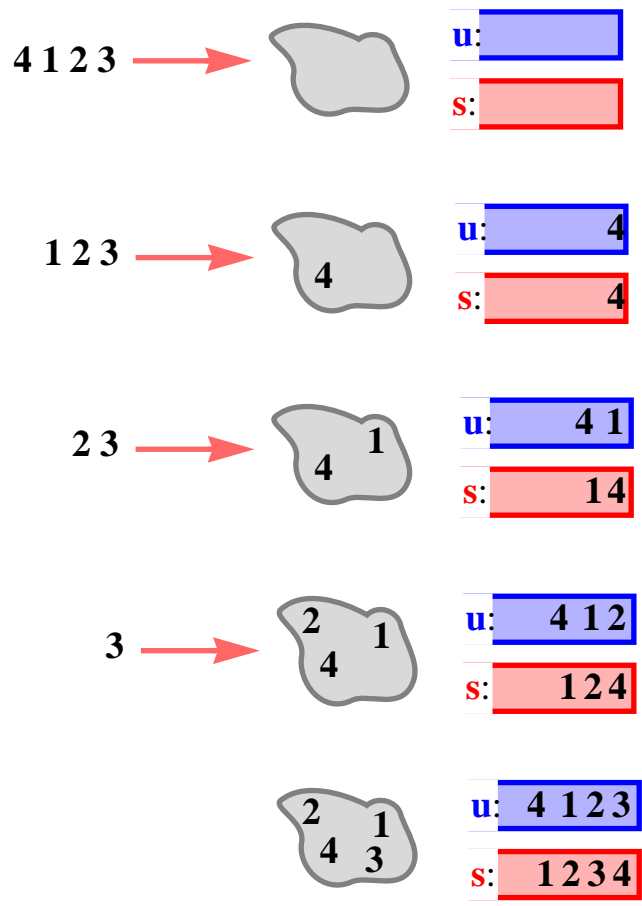
Selection Sort



/

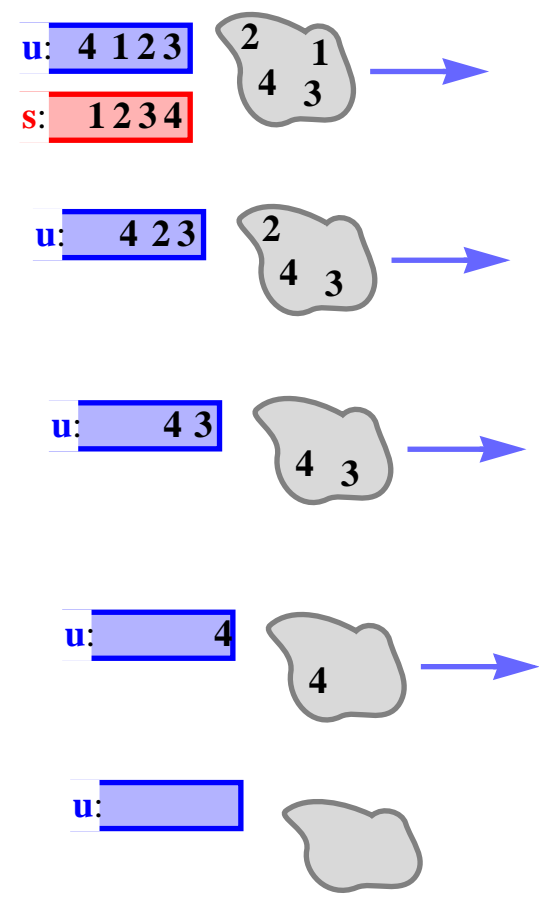
Selection Sort

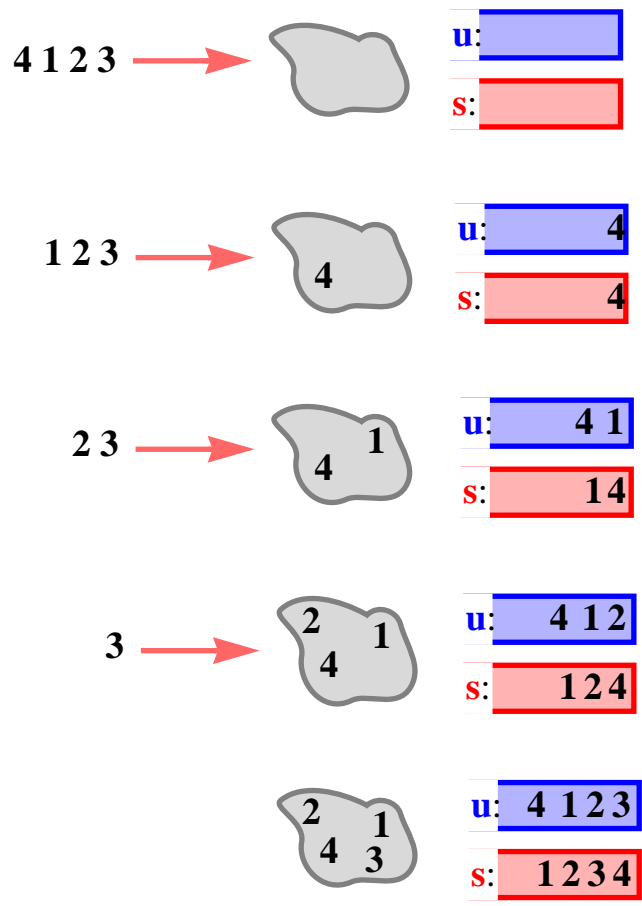




/

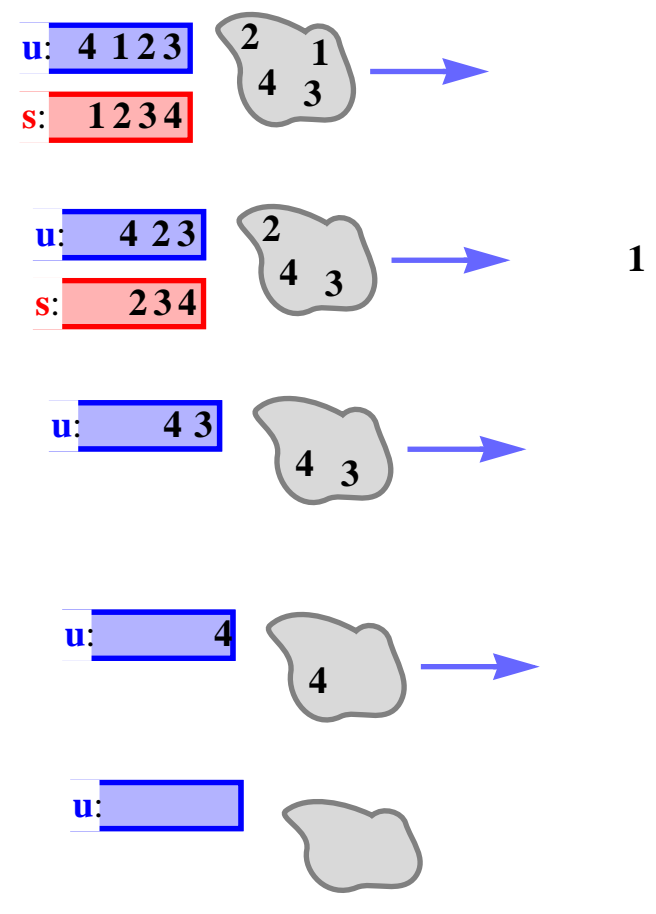
Selection Sort

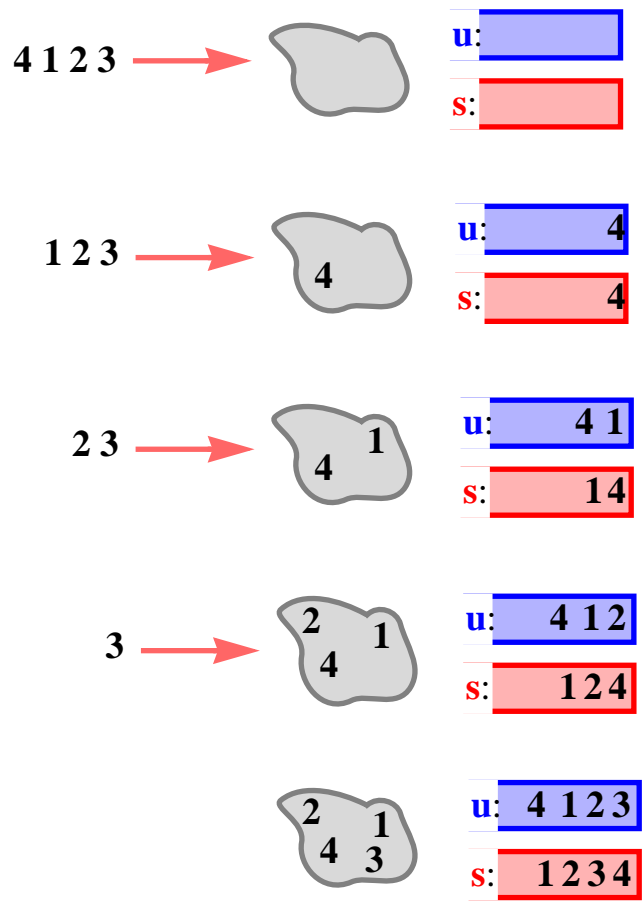




/

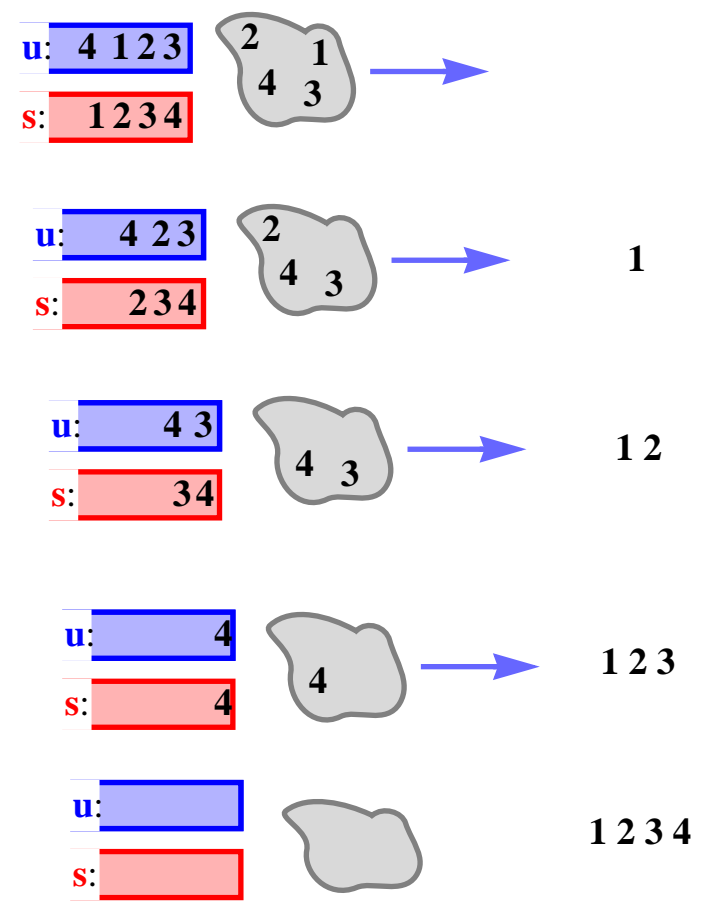
Selection Sort

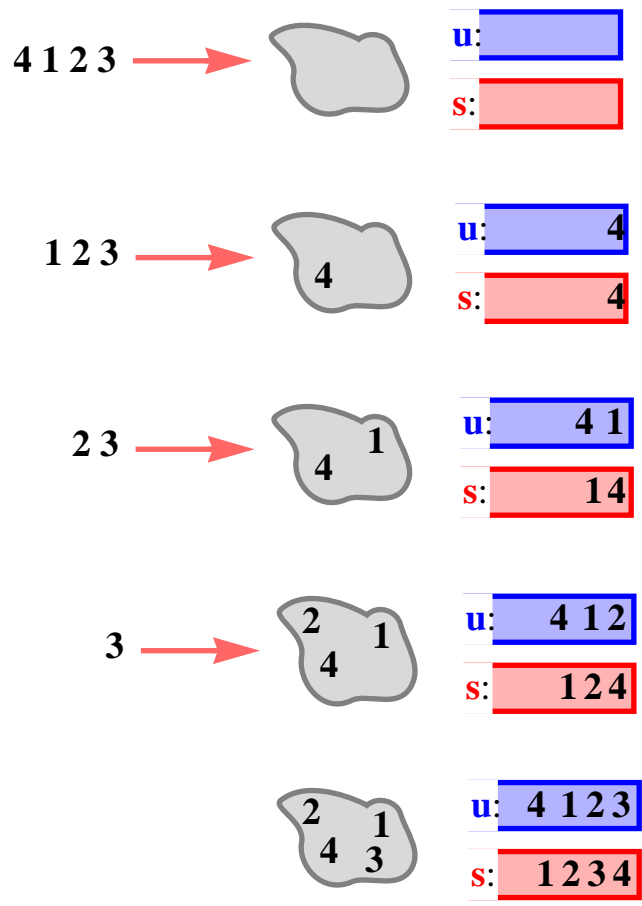




/

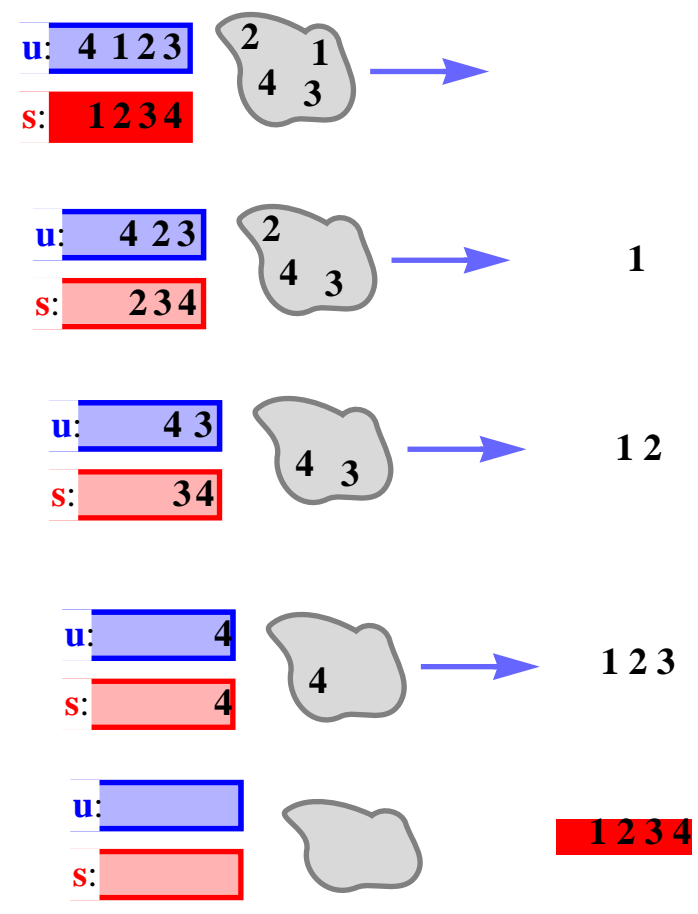
Selection Sort

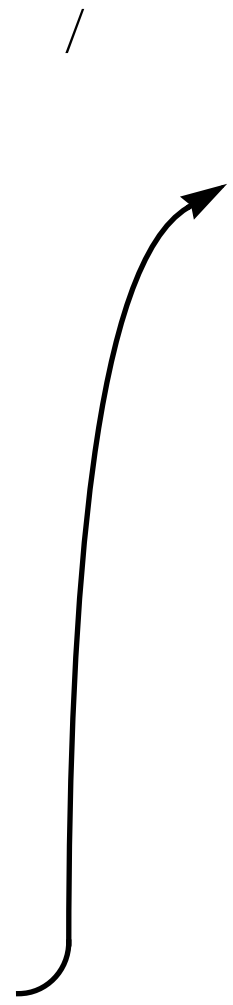
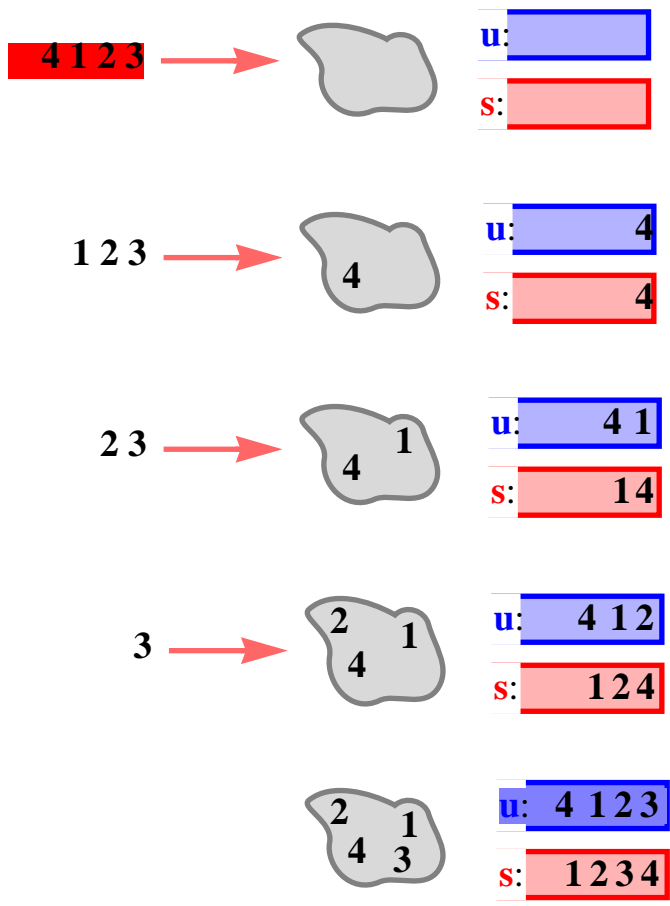




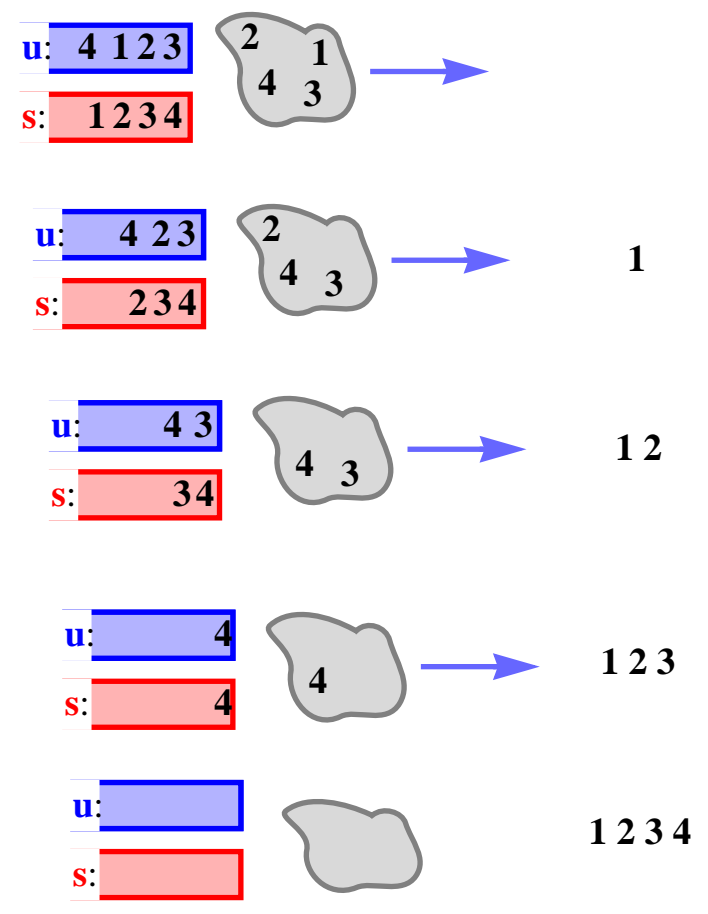
/

Selection Sort

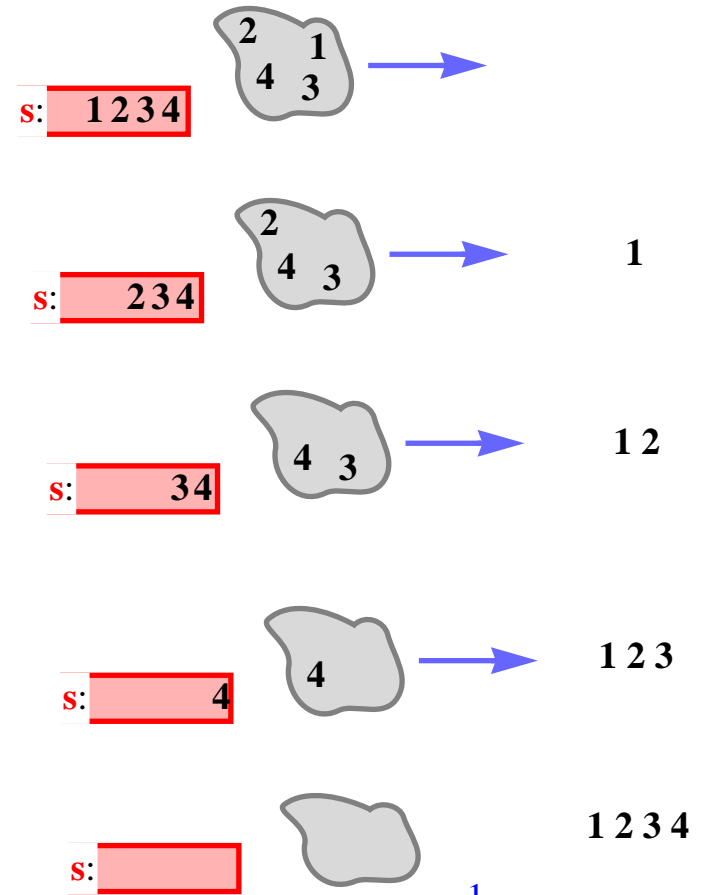
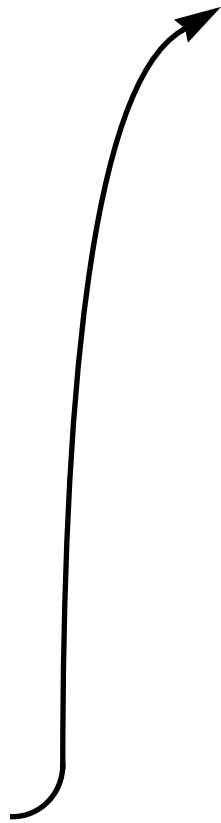
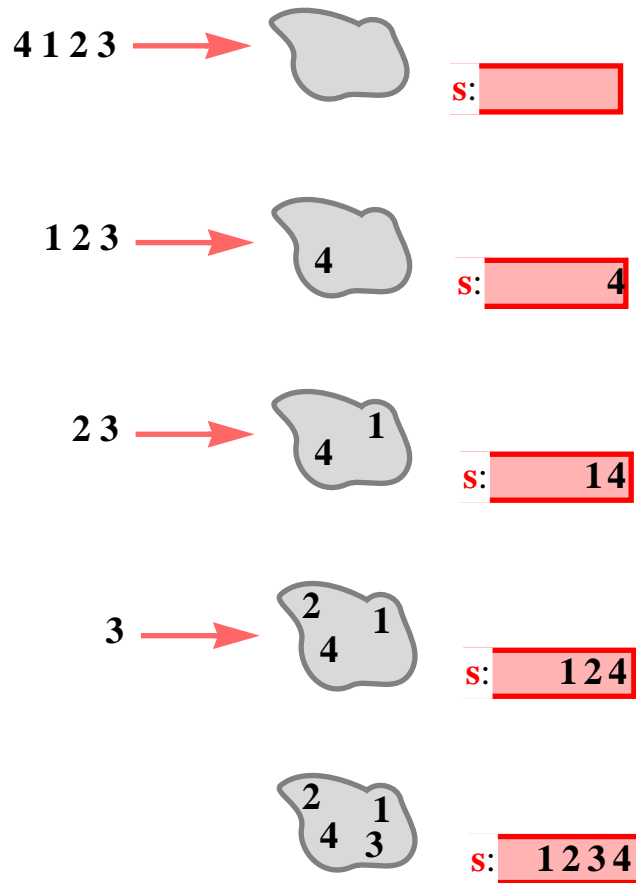




Selection Sort



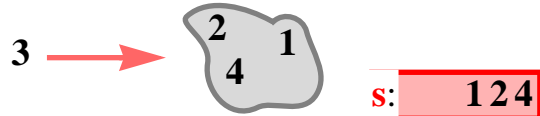
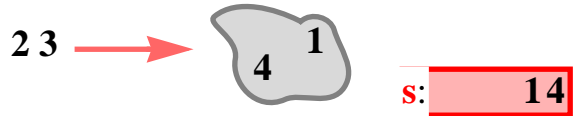
/ Selection Sort



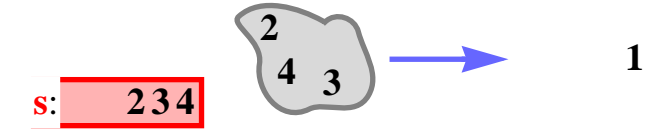
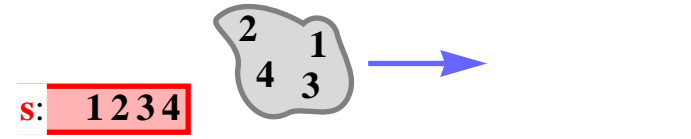
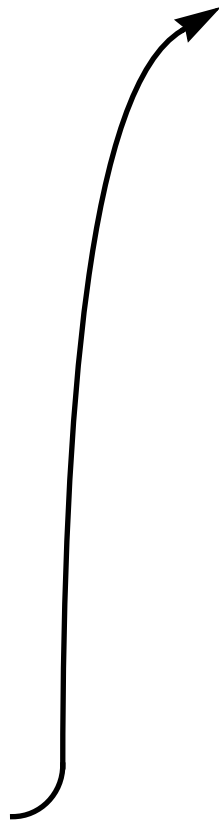
$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$



/ Selection Sort



$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

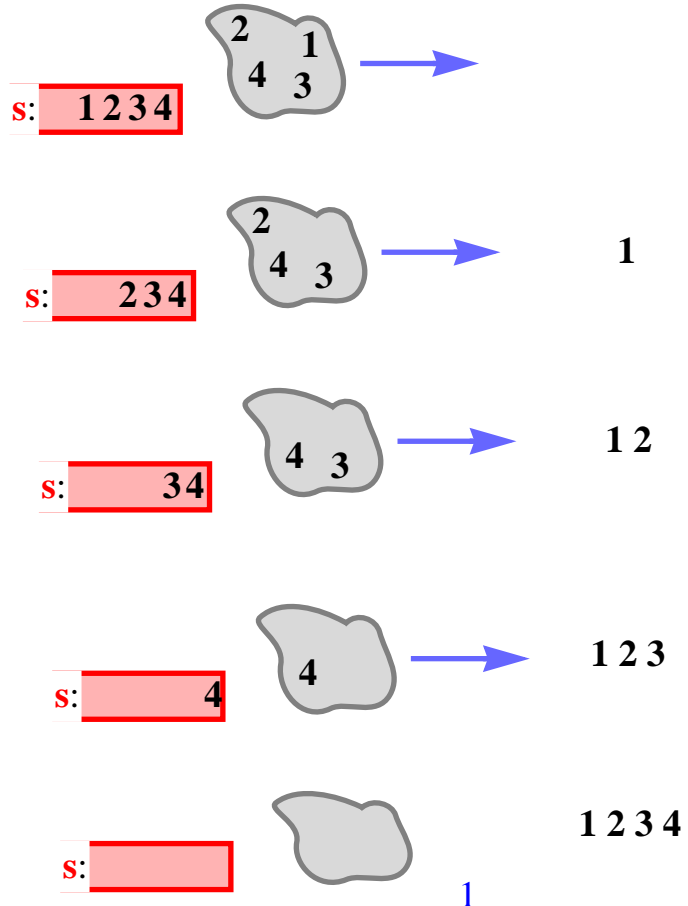
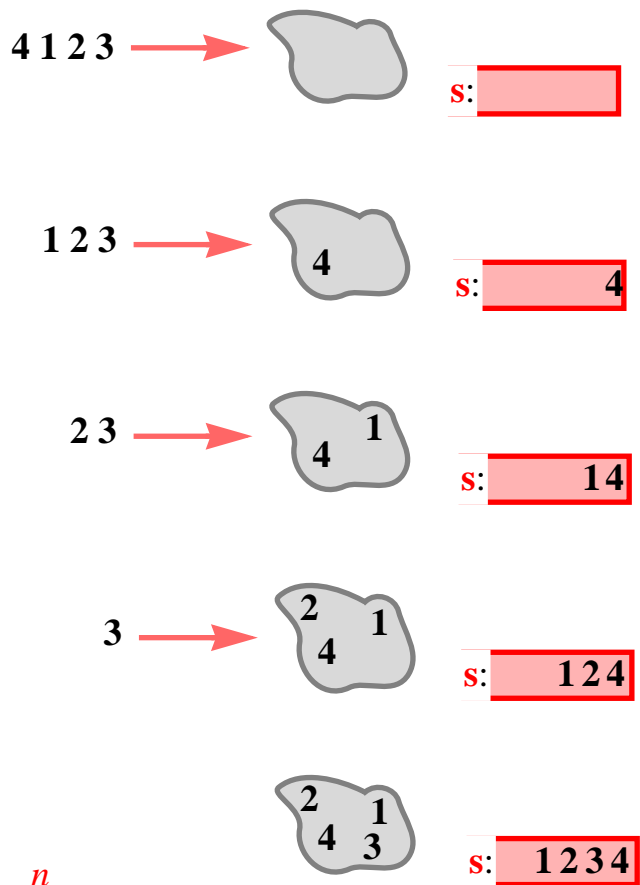


$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$



Insertion

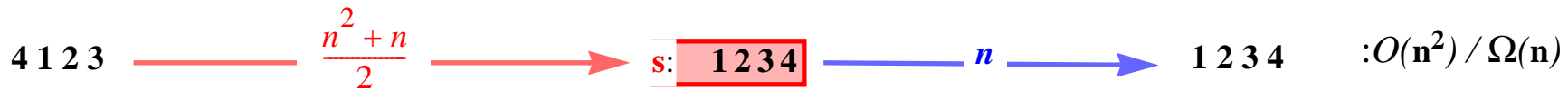
Selection Sort



$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

+

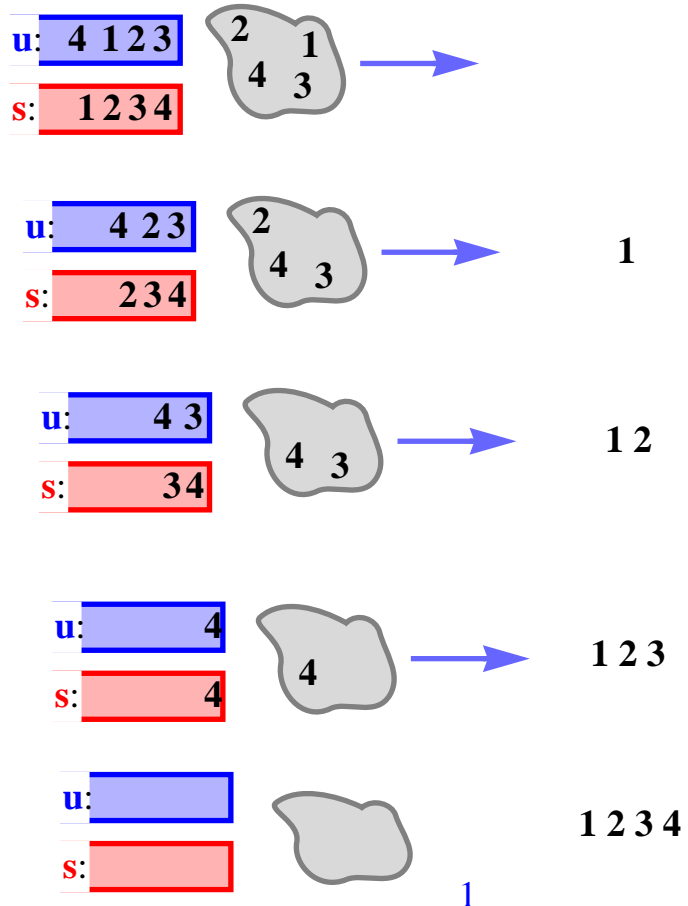
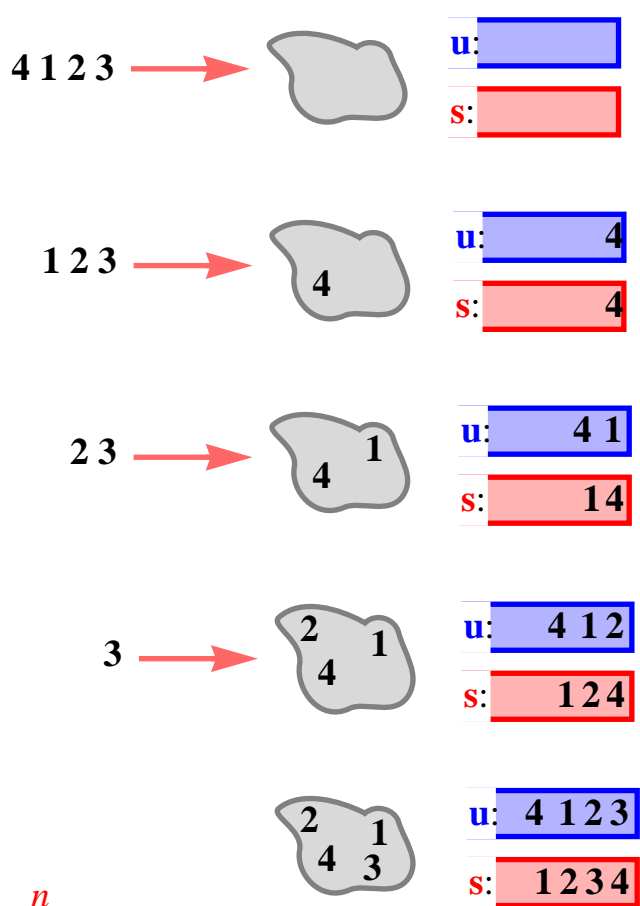
$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$



Insertion

/

Selection Sort



$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

+

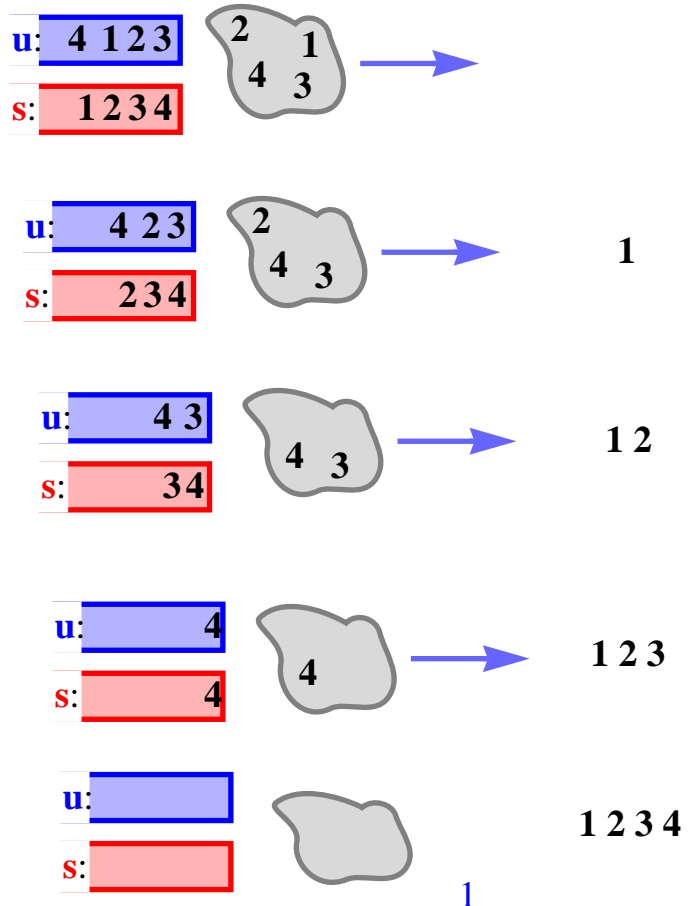
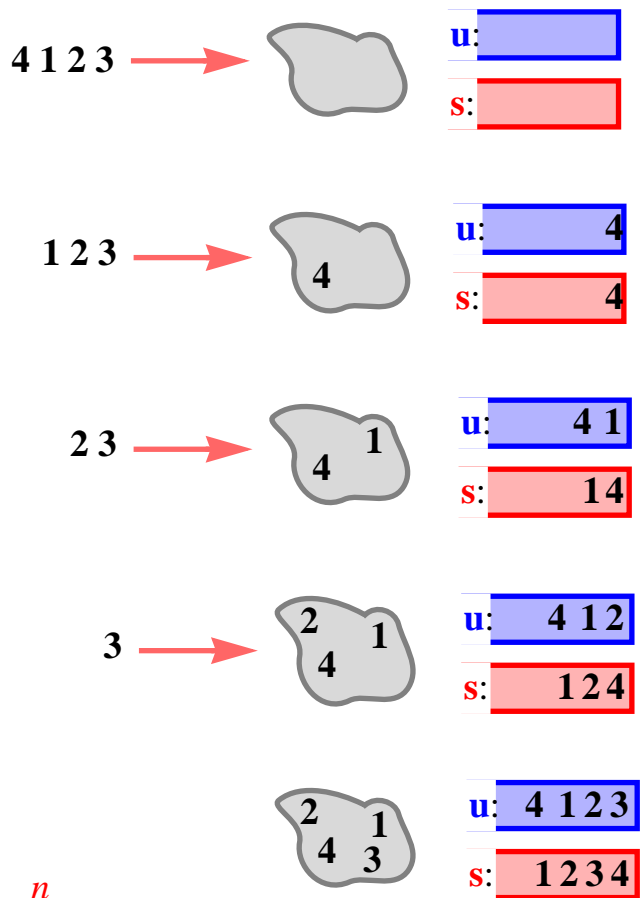
$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$

$$4\ 1\ 2\ 3 \xrightarrow{n} [u: 4\ 1\ 2\ 3] \xrightarrow{\frac{n^2+n}{2}} 1\ 2\ 3\ 4 \quad :O(n^2) / \Omega(n^2)$$

$$4\ 1\ 2\ 3 \xrightarrow{\frac{n^2+n}{2}} [s: 1\ 2\ 3\ 4] \xrightarrow{n} 1\ 2\ 3\ 4 \quad :O(n^2) / \Omega(n)$$

Insertion

Selection Sort



$$O\left(\sum_{k=1}^n P_k \cdot \text{insert}(e)\right)$$

+

$$O\left(\sum_{k=n}^1 P_k \cdot \text{remMin}(\)\right)$$

$$4\ 1\ 2\ 3 \xrightarrow{n} \text{u: } 4\ 1\ 2\ 3 \xrightarrow{\frac{n^2+n}{2}} 1\ 2\ 3\ 4 \quad :O(n^2) / \Omega(n^2)$$

$$4\ 1\ 2\ 3 \xrightarrow{\frac{n^2+n}{2}} \text{s: } 1\ 2\ 3\ 4 \xrightarrow{n} 1\ 2\ 3\ 4 \quad :O(n^2) / \Omega(n)$$

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): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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

V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

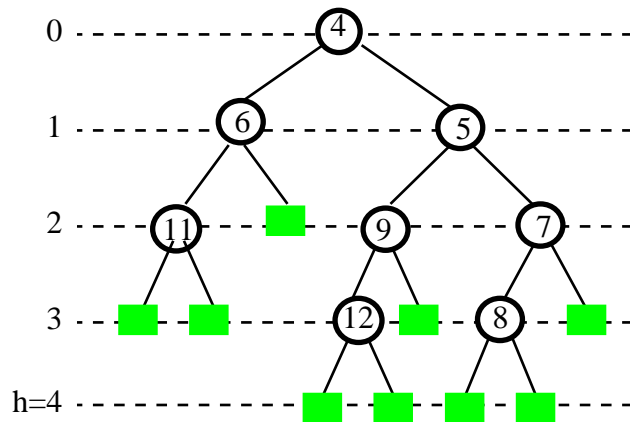
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

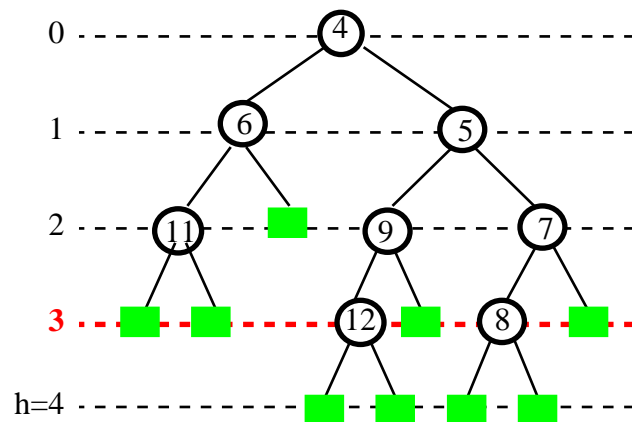
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

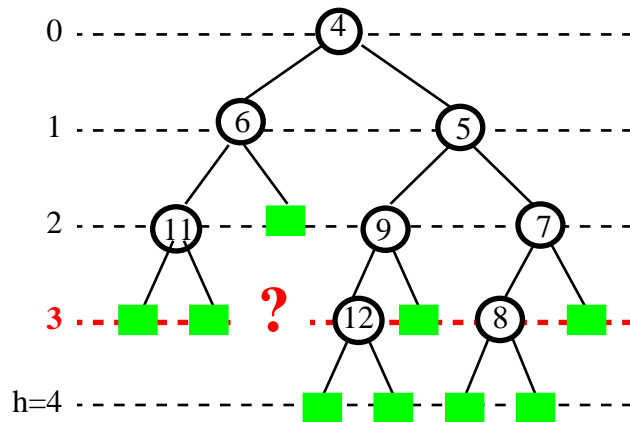
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

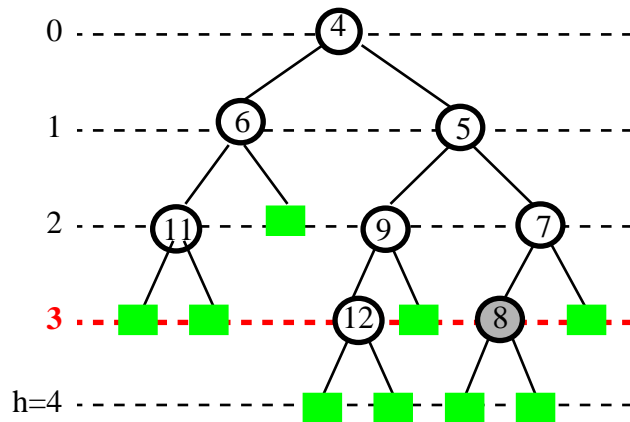
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

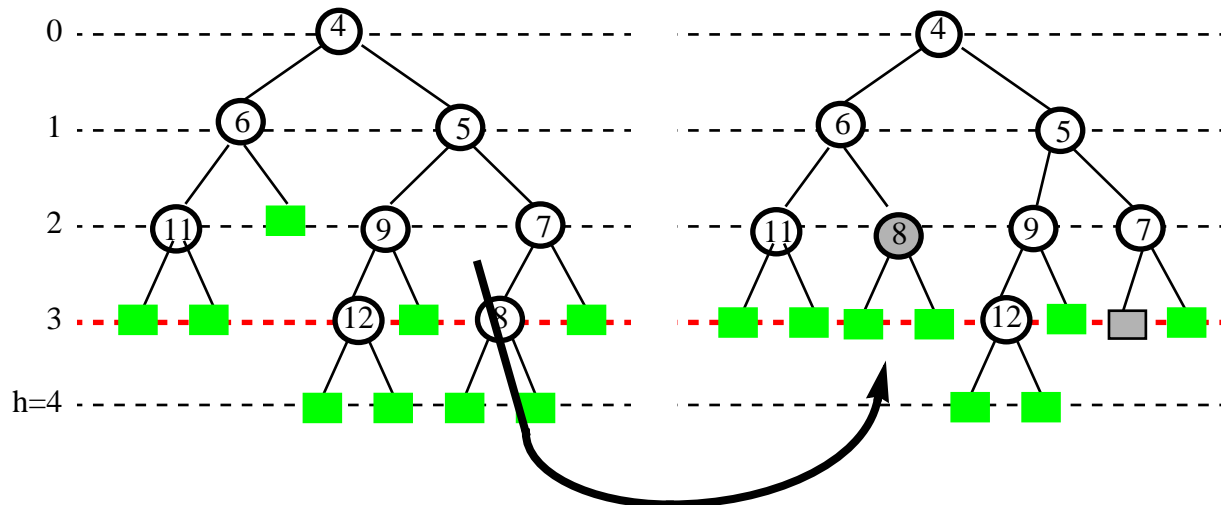
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

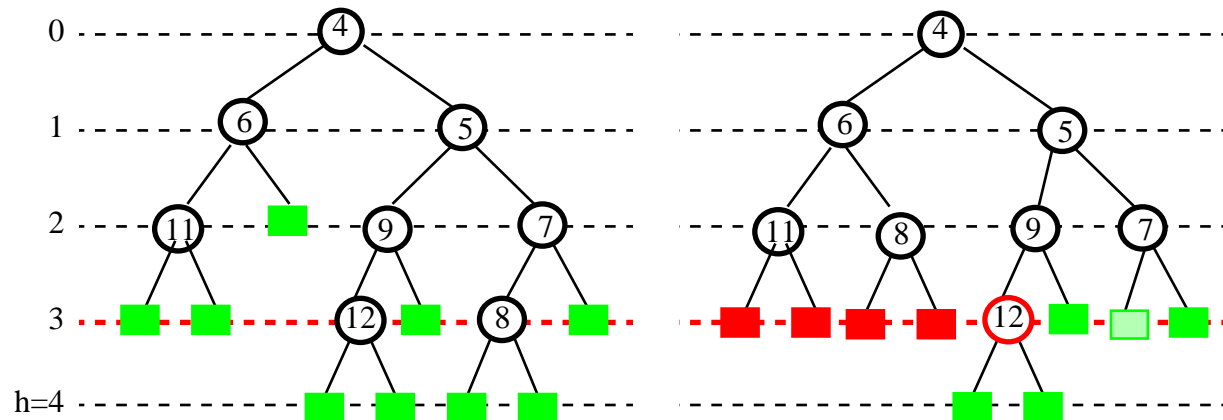
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

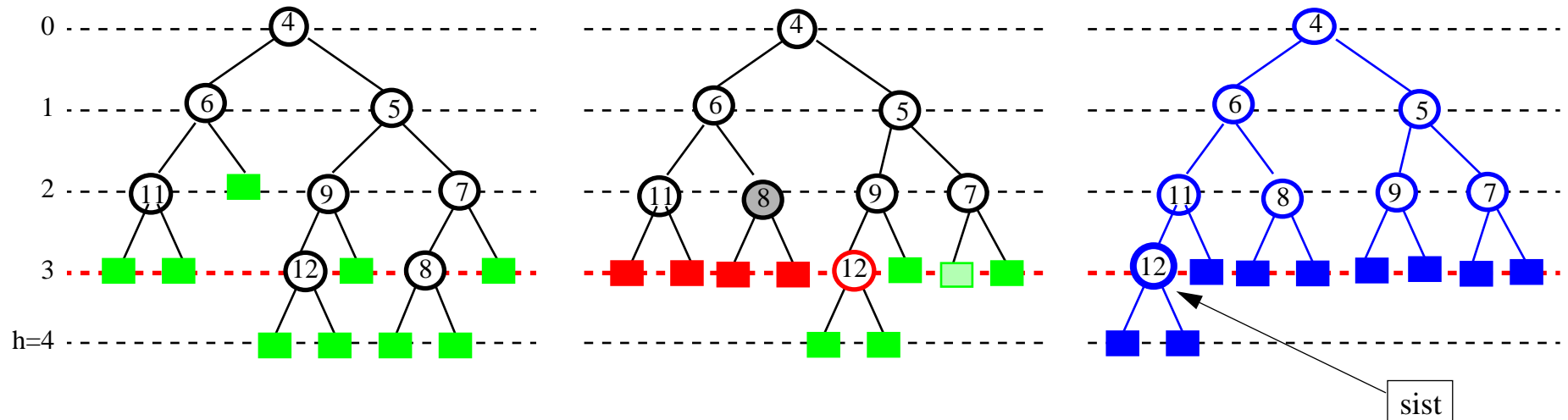
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



V. Heap DataStruktur

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

1. Heap-Ordering (“relasjonell”)

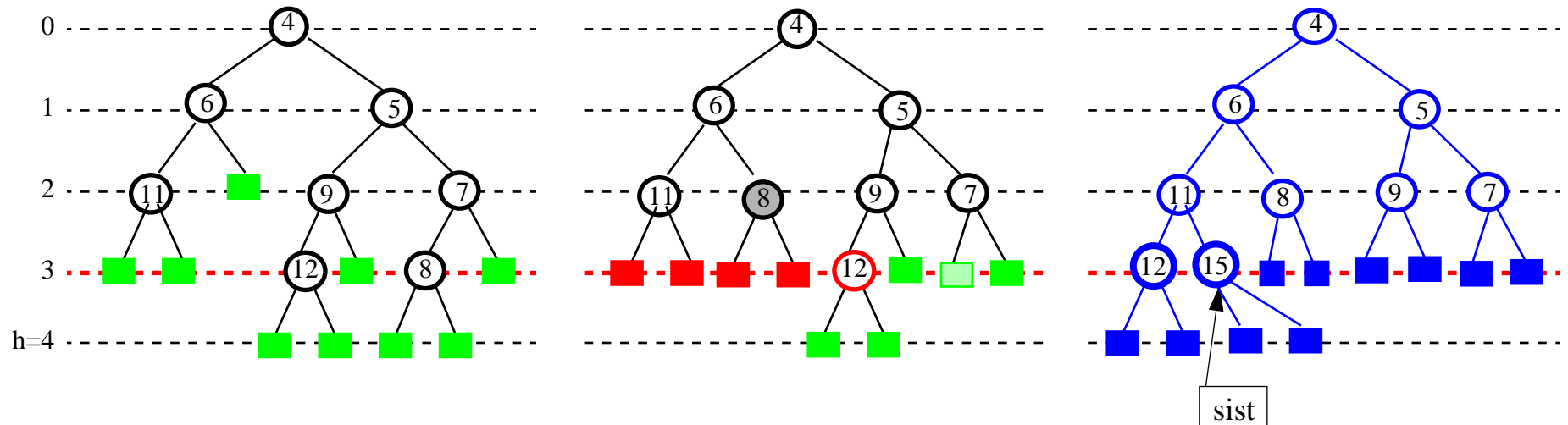
for enhver node v (unntatt roten): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

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



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): $\text{key}(v) \geq \text{key}(\text{parent}(v))$

2. Komplett Binært Tre (“strukturell”)

T med høyde h :

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

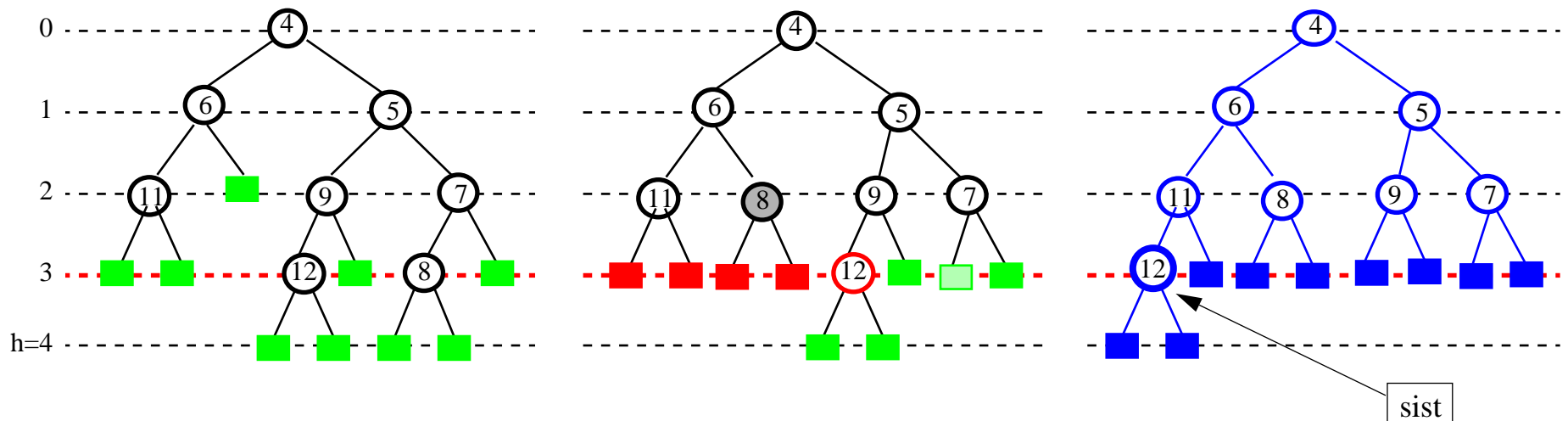
2.b) på nivå $h-1$ alle interne noder er “til venstre for” alle eksterne

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

$$1 + 2 + \dots + 2^{h-2} + 1 = 2^{h-1} \leq n$$

$$n \leq 1 + 2 + \dots + 2^{h-2} + 2^{h-1} = 2^h - 1$$

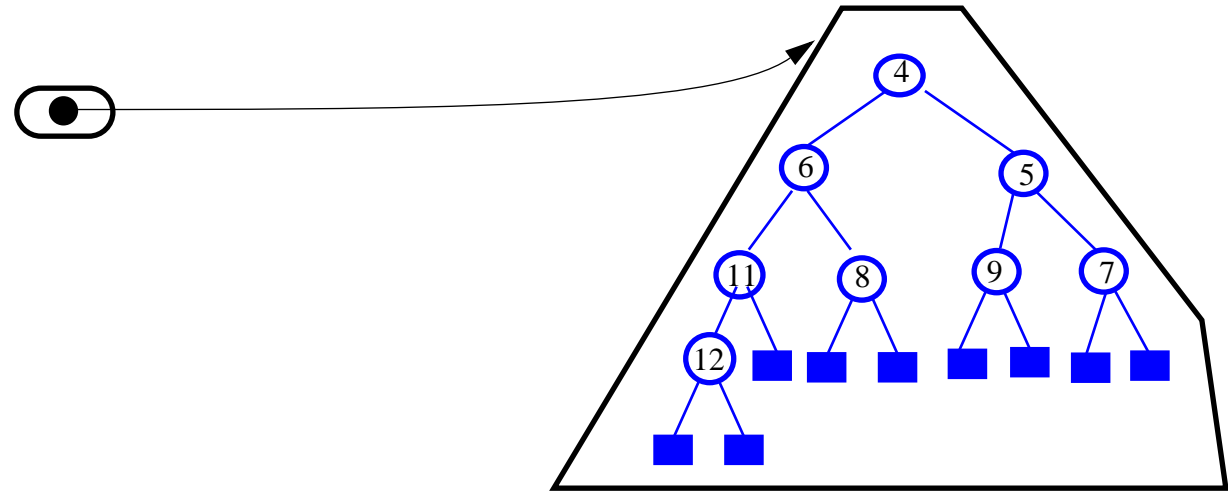
$$h \leq \log(n) + 1 \ \& \ \log(n+1) \leq h$$



Implementasjon av PriorityQueue *med Heap*

DATASTRUKTUR

private **BinaryTree** heap

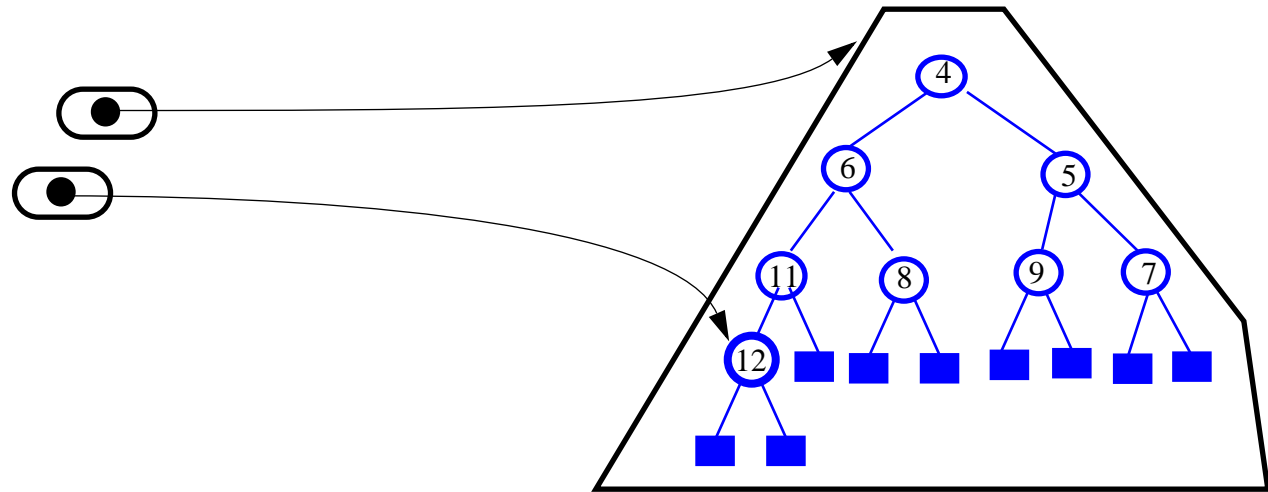


Implementasjon av PriorityQueue *med Heap*

DATASTRUKTUR

private **BinaryTree** heap

private **Position** sist



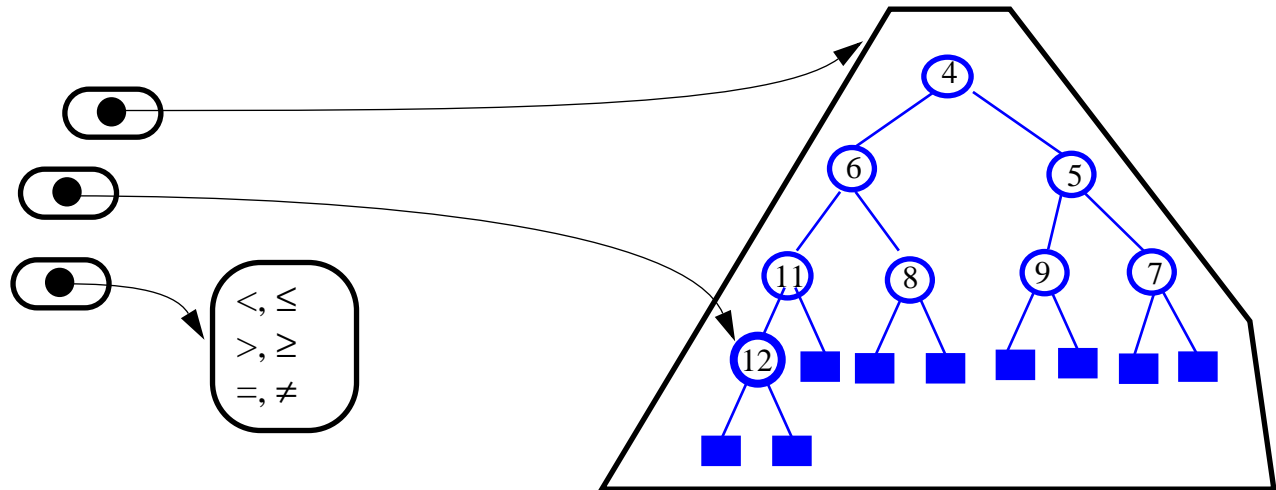
Implementasjon av PriorityQueue *med Heap*

DATASTRUKTUR

private **BinaryTree** heap

private **Position** sist

private **Comparator** cp



Implementasjon av PriorityQueue *med Heap*

DATASTRUKTUR

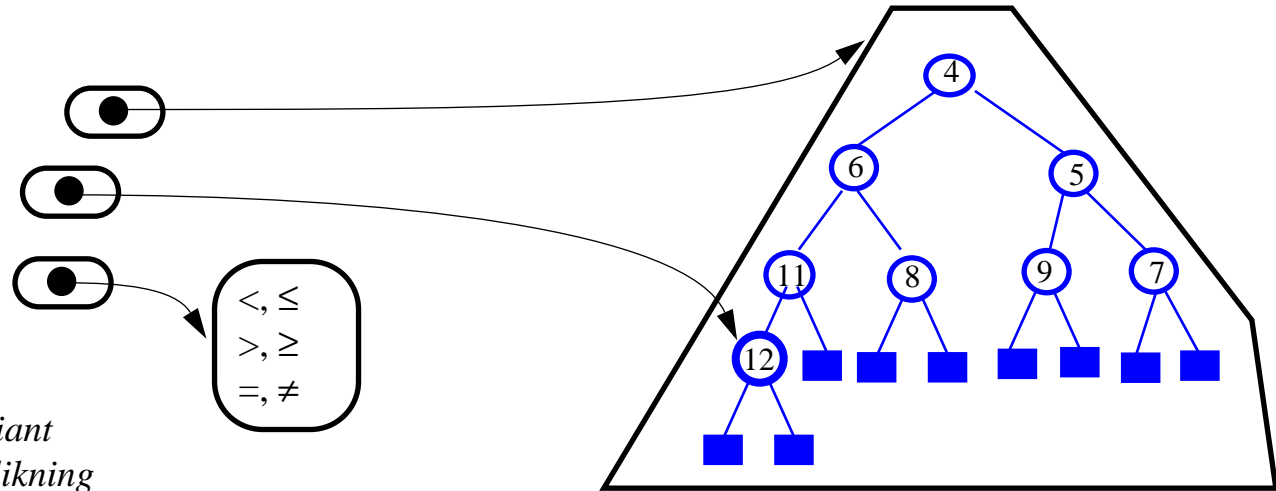
private **BinaryTree** heap

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()*



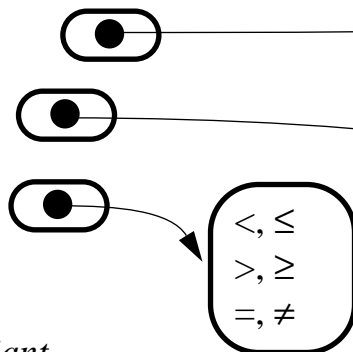
Implementasjon av PriorityQueue *med Heap*

DATASTRUKTUR

private **BinaryTree** heap

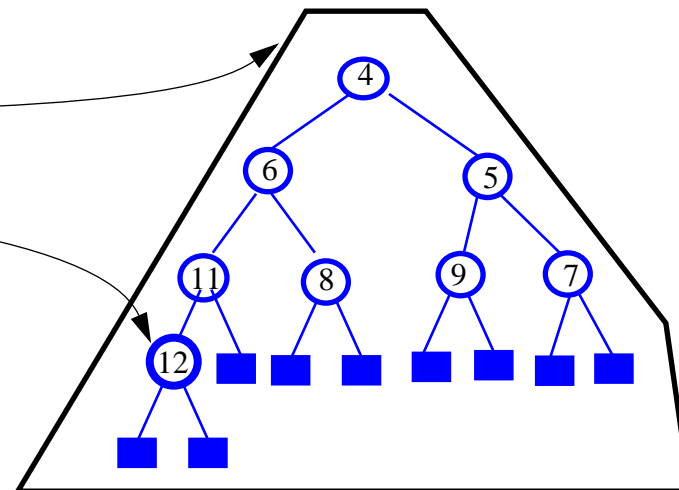
private **Position** sist

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 (heap.positions()) 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 }  
}
```

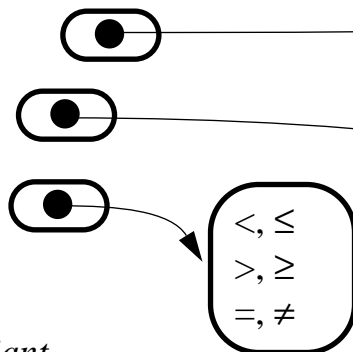
Implementasjon av PriorityQueue *med Heap*

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()

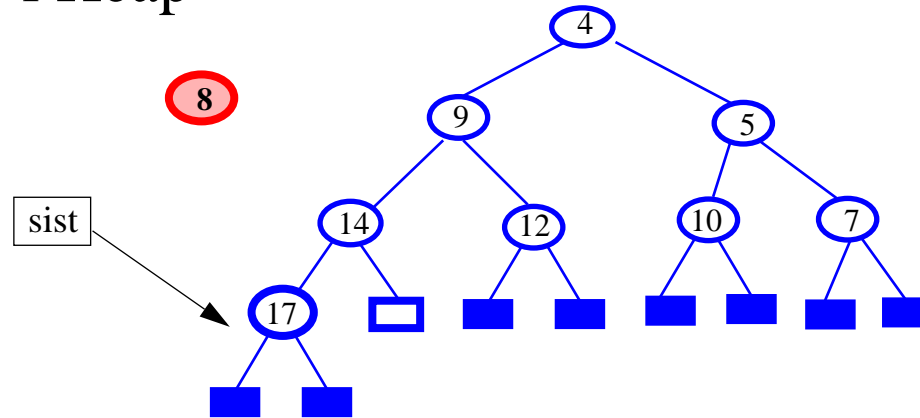
```
public PQhp(Comparator c) {
    cp = c ; heap = new BinaryTreeIMP(); } // sist==null hviss heap.isEmpty()

private boolean DI() {
    traverser Heap (heap.positions()) og sjekk at enhver node v har key(v) >= key(parent(v)) :
        cp . isGreaterThanOrEqualTo ( (Item)v.element() , (Item) heap.parent(v).element() )
    Komplette BinTree er vanskeligere }

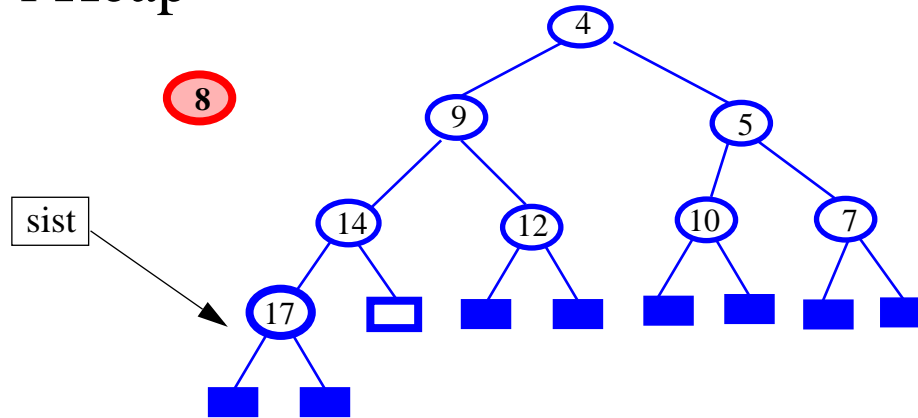
public Object minElement() throws EmptyContainerException {
    if (heap.isEmpty()) throw new EmptyContainerException("");
    return ((Item) Heap.root().element()) . element() ; }

public Object minKey() throws EmptyContainerException {
    if (heap.isEmpty()) throw new EmptyContainerException("");
    return ((Item) Heap.root().element()) . key() ; }
```

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



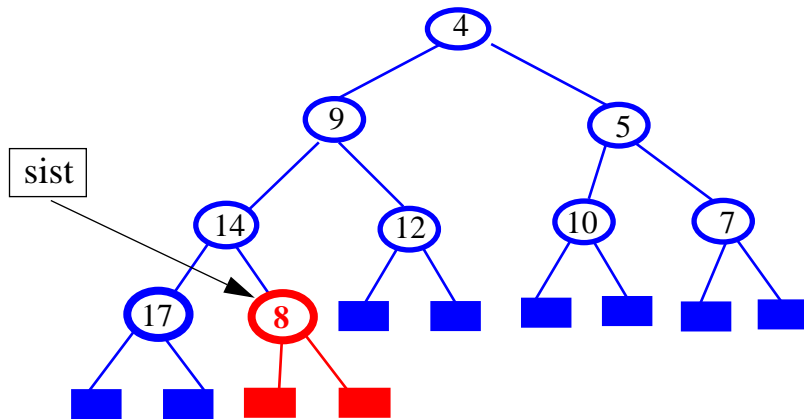
A. *insertItem(k,o)* i Heap



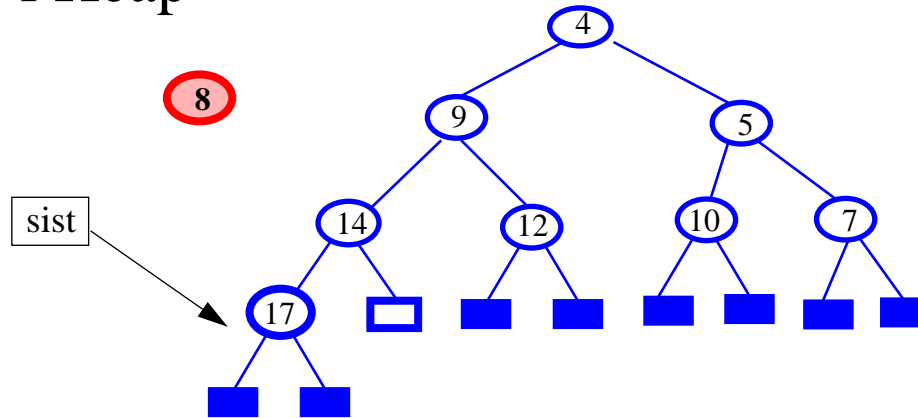
1. Bevar **KOMPLETT BINÆRT TRE INVARIANT**:

*Finn innsetningsnode – ‘til høyre’ for siste
– utvid bladet til en intern node og sett inn
det nye elementet*

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



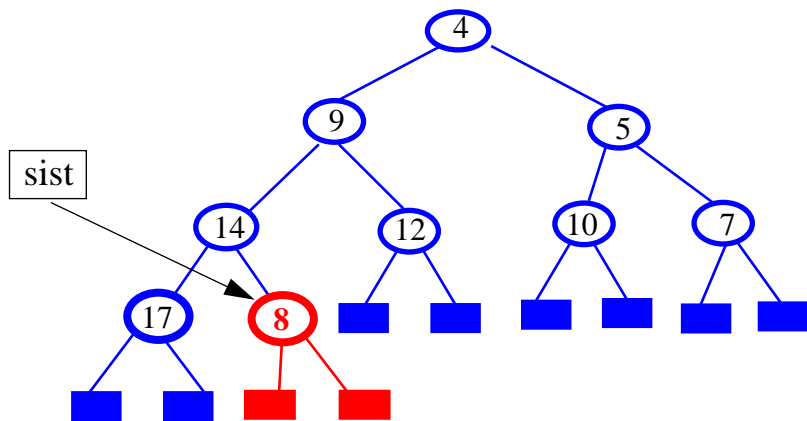
A. insertItem(k,o) i Heap



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

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

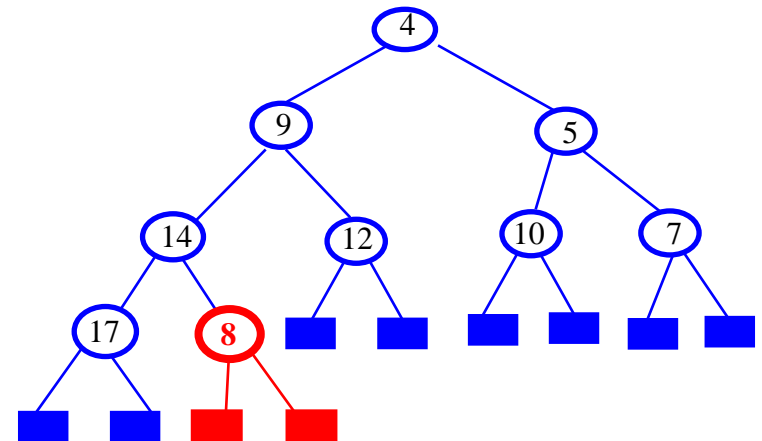
gitt sist: $O(1)$ el. $O(\log n)$



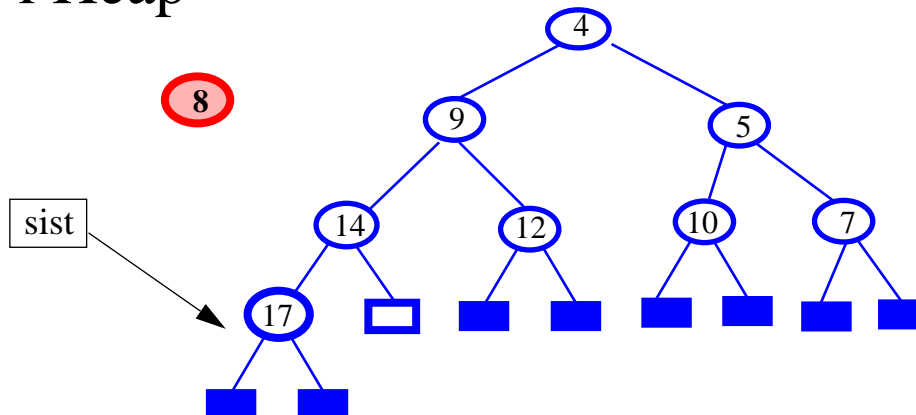
2. Gjenopprett **HEAP-ORDERING INVARIANT** :

Flytt det nye elementet “opp” inntil dets
 forelder har mindre nøkkel

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



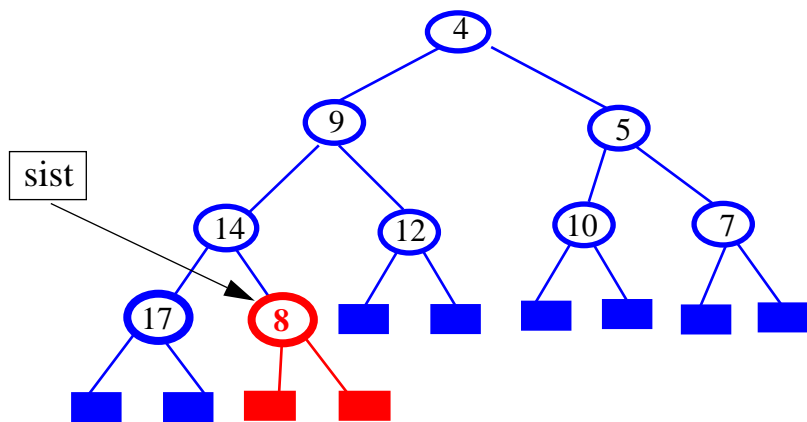
A. insertItem(k,o) i Heap



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

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

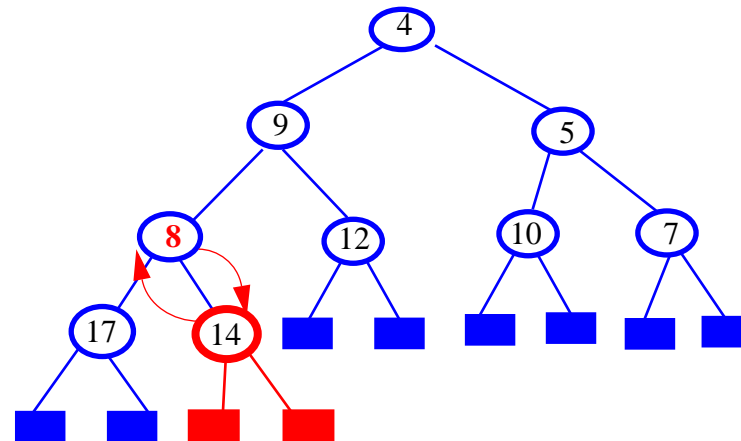
gitt sist: $O(1)$ el. $O(\log n)$



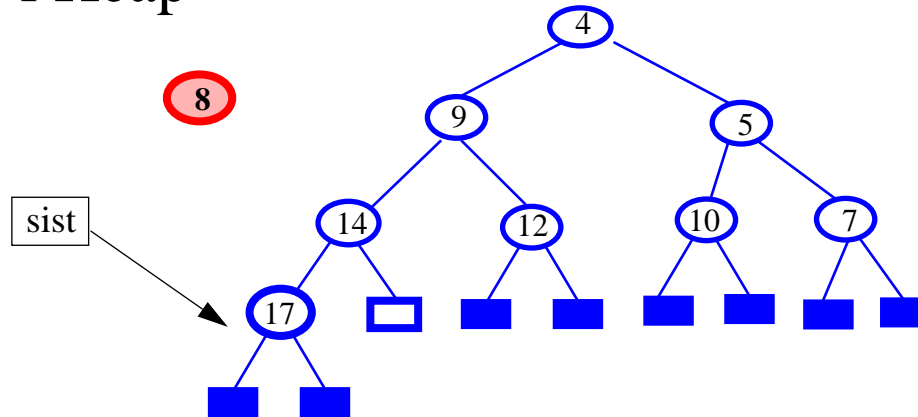
2. Gjenopprett **HEAP-ORDERING INVARIANT** :

Flytt det nye elementet “opp” inntil dets
 forelder har mindre nøkkel

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



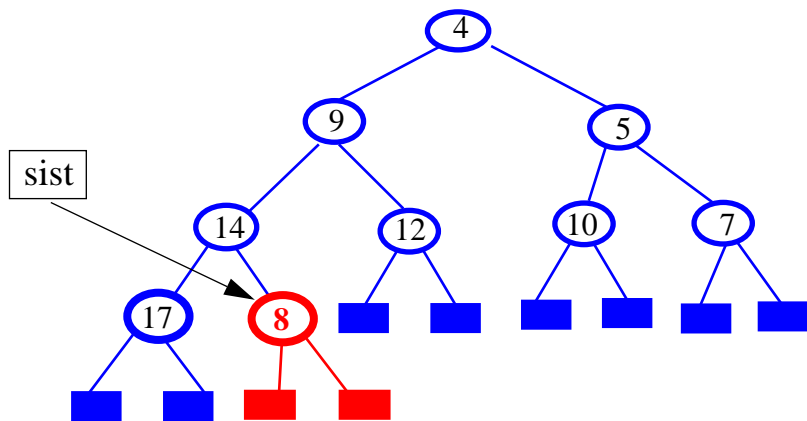
A. insertItem(k,o) i Heap



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

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

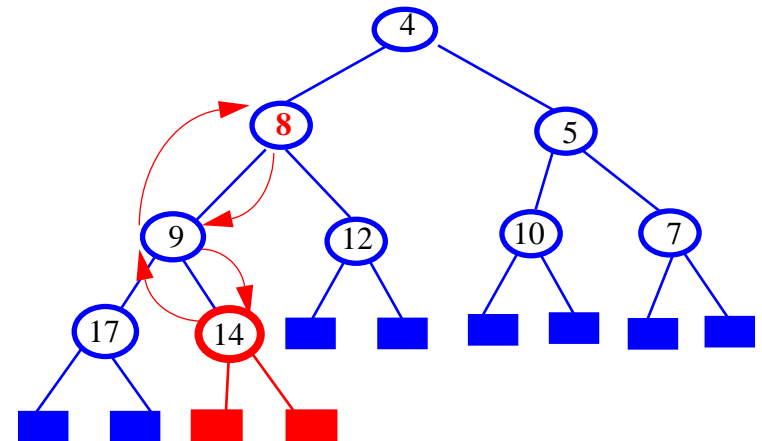
gitt sist: $O(1)$ el. $O(\log n)$



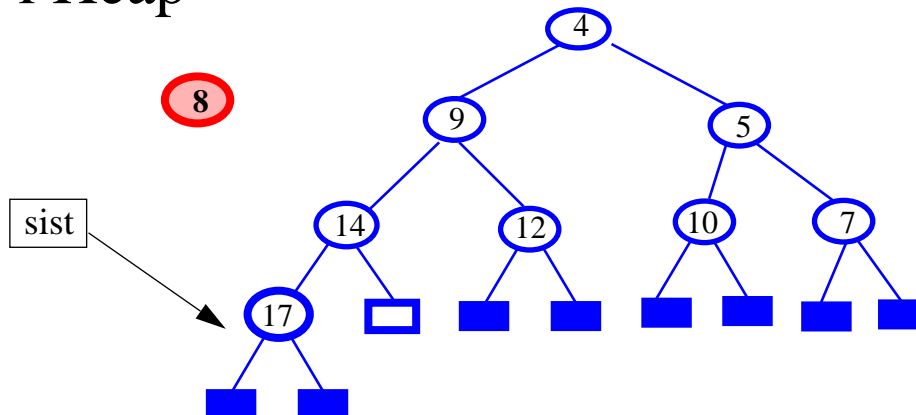
2. Gjenopprett **HEAP-ORDERING INVARIANT** :

Flytt det nye elementet “opp” inntil dets
 forelder har mindre nøkkel

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



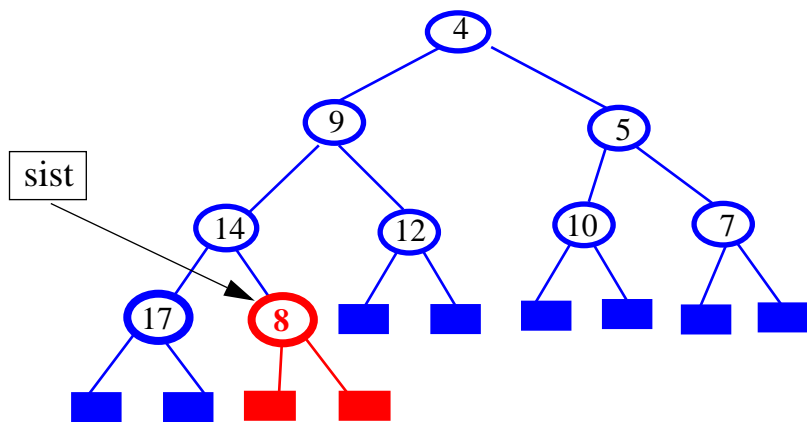
A. insertItem(k,o) i Heap



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

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

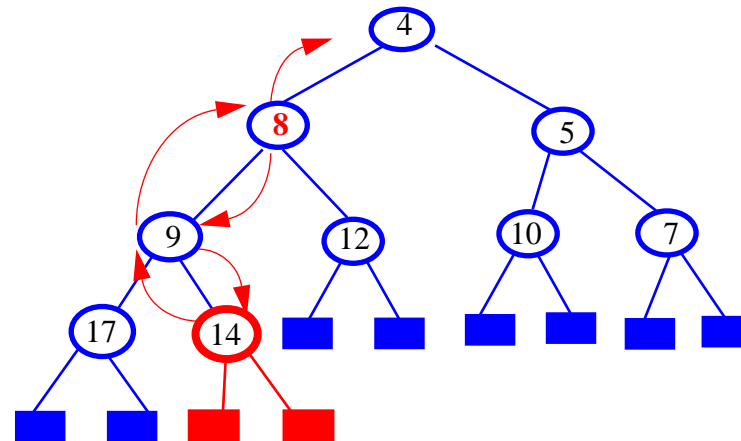
gitt sist: $O(1)$ el. $O(\log n)$



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 innsetningsnode

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

- **Sequence** (array):

sist = n ; **u** = **sist** + 1

O(1)

A.1. Finn innsetningsnode

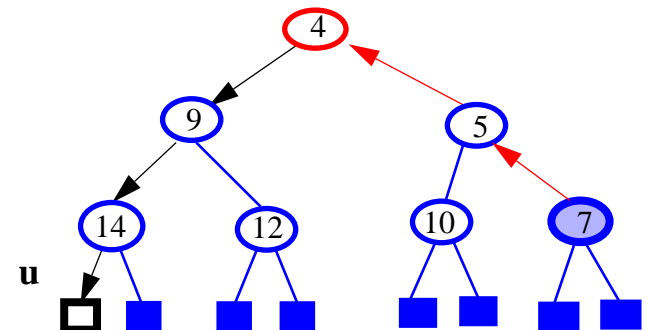
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

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

- **BinaryTree** interface
(vilkaarlig implementasjon)

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

- a) *$T.isEmpty()$: **u** = $T.root()$*
- b) *ytterste noden på nivå $h-1$*



A.1. Finn innsetningsnode

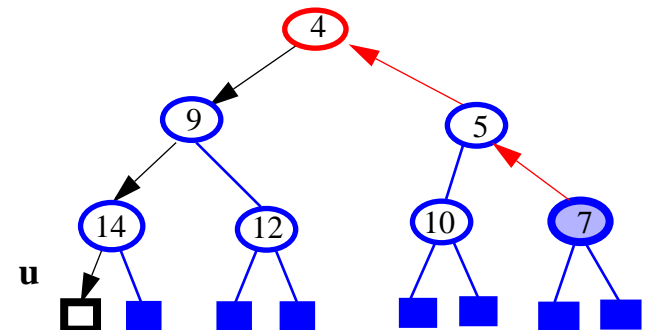
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

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

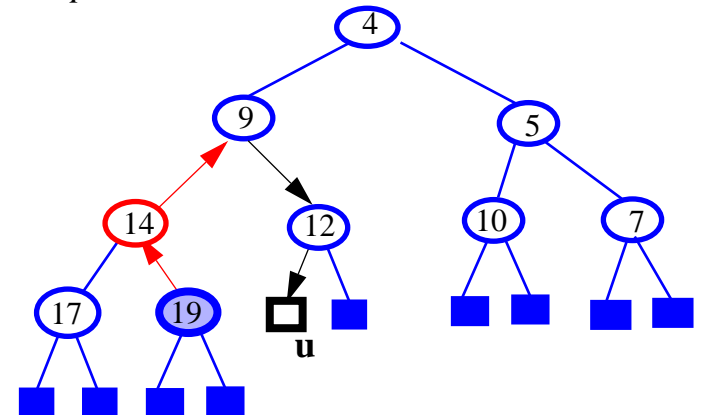
- **BinaryTree** interface
(vilka'rlig implementasjon)

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

- a) *T.isEmpty()* : **u** = *T.root()*
- b) ytterste noden p'a niv'ao $h-1$



- c) en mellomnode p'ao niv'ao $h-1$



A.1. Finn innsetningsnode

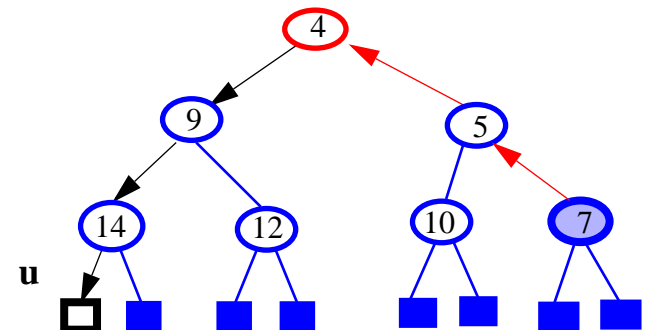
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

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

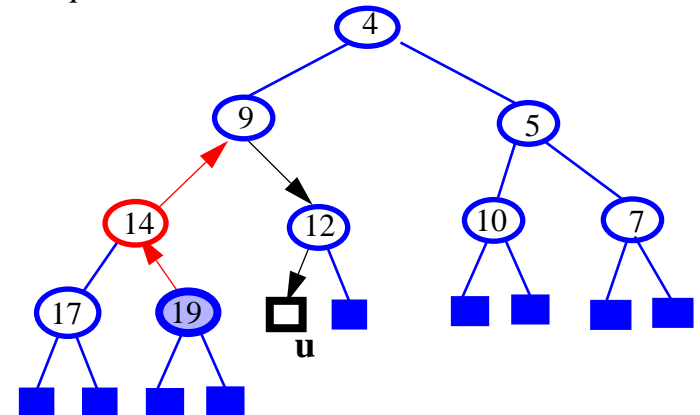
```
u = sist
while ( u != root()
      && u != leftChild(parent(u)) )
    u = parent(u);
if ( u != root()
    u = rightChild(parent(u));
while ( ! isExternal(u) )
    u = leftChild(u);
return u O(log n)
```

- **BinaryTree** interface
(vilkaarlig 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.1. Finn innsetningsnode

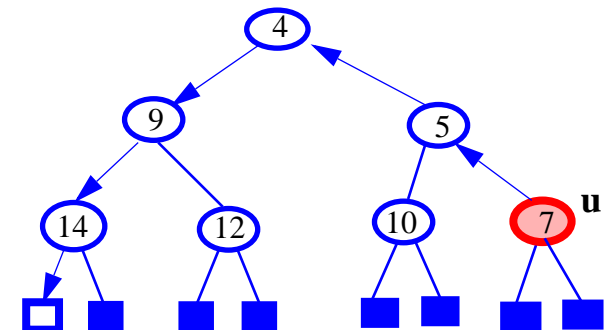
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

- **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
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:

 - a) *T.isEmpty()* : **u** = *T.root()*
 - b) ytterste noden på nivå $h-1$



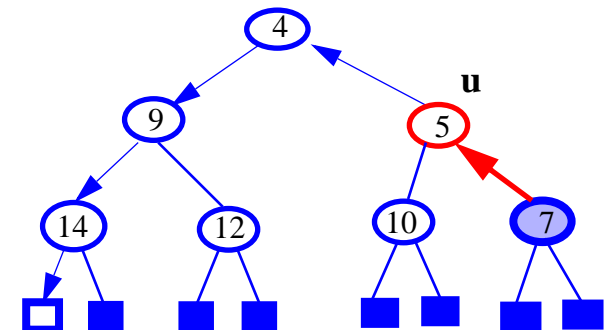
A.1. Finn innsetningsnode

1. Gitt **sist**, finn innsetningsnode **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
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:
- a) *T.isEmpty()* : **u** = *T.root()*
 - b) ytterste noden på nivå $h-1$



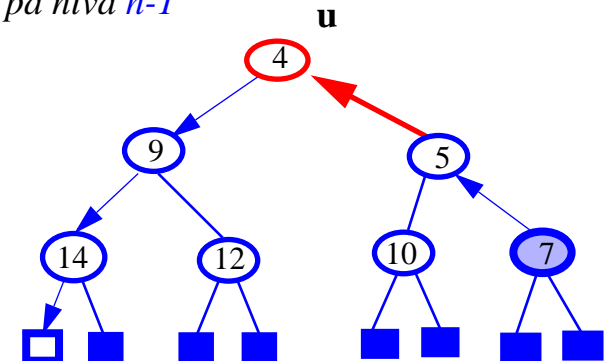
A.1. Finn innsetningsnode

1. Gitt **sist**, finn innsetningsnode **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
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:
- a) *T.isEmpty()* : **u** = *T.root()*
 - b) ytterste noden på nivå $h-1$



A.1. Finn innsetningsnode

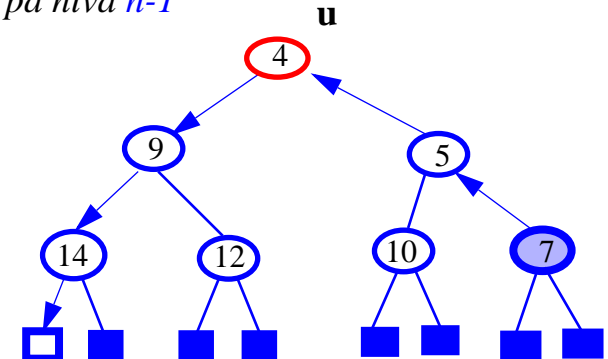
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

- **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
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:

 - a) *T.isEmpty()* : **u** = *T.root()*
 - b) ytterste noden på nivå $h-1$



A.1. Finn innsetningsnode

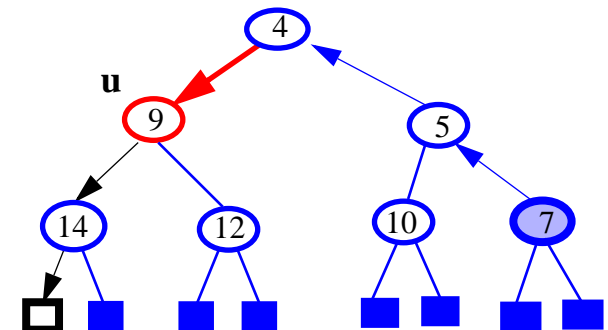
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

- **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
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:

 - a) *T.isEmpty()* : **u** = *T.root()*
 - b) ytterste noden på nivå *h-1*



A.1. Finn innsetningsnode

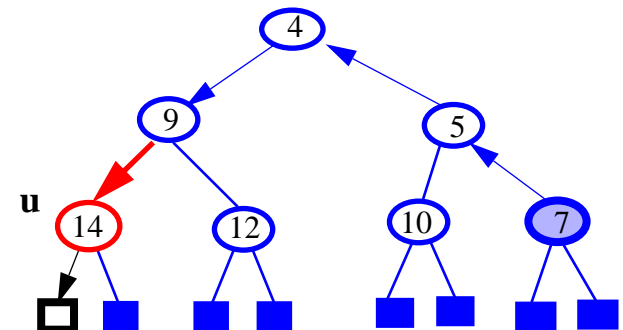
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

- **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
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:

 - a) *T.isEmpty()* : **u** = *T.root()*
 - b) ytterste noden på nivå *h-1*



A.1. Finn innsetningsnode

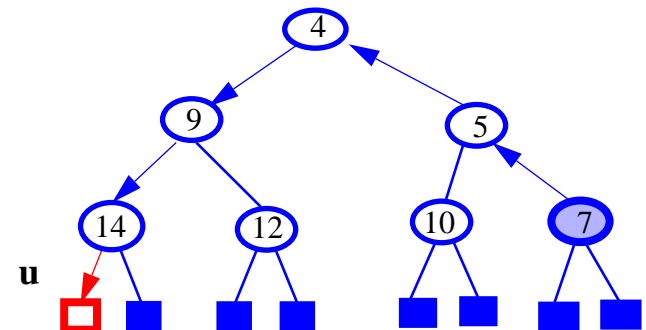
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

- **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
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:

 - a) *T.isEmpty()* : **u** = *T.root()*
 - b) ytterste noden på nivå *h-1*



A.1. Finn innsetningsnode

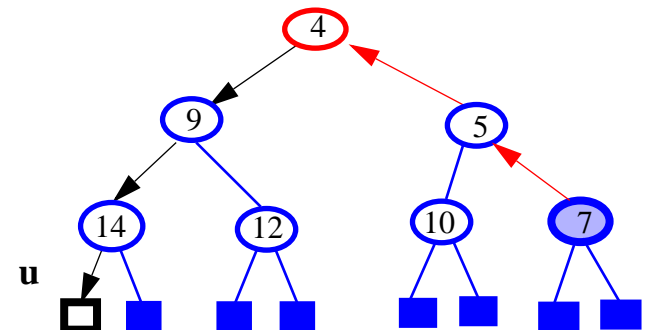
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

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

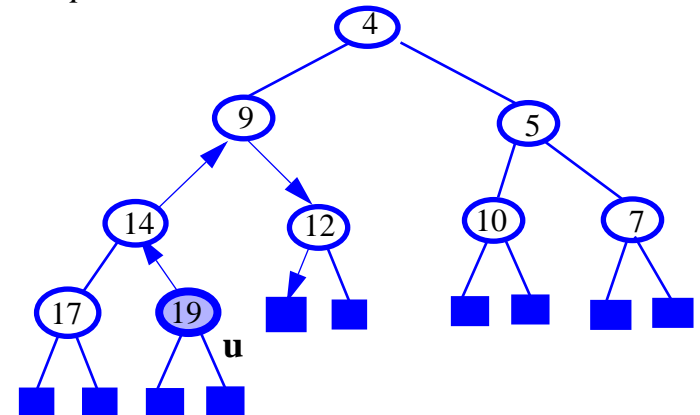
```
u = sist
while ( u != root()
      && u != leftChild(parent(u)) )
    u = parent(u) ;
if ( u != root )
    u = rightChild(parent(u)) ;
while ( ! isExternal(u) )
    u = leftChild(u) ;
return u O(log n)
```

- **BinaryTree** interface
(vilkaarlig 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.1. Finn innsetningsnode

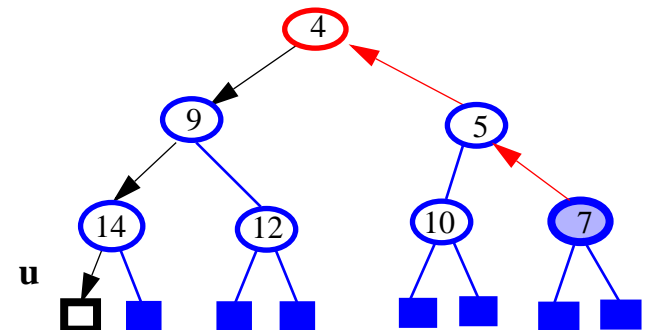
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

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

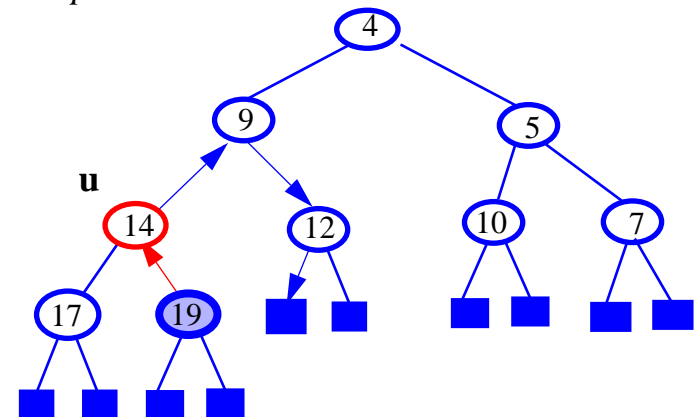
```
u = sist
while ( u != root()
      && u != leftChild(parent(u)) )
    u = parent(u) ;
if ( u != root )
    u = rightChild(parent(u)) ;
while ( ! isExternal(u) )
    u = leftChild(u) ;
return u O(log n)
```

- **BinaryTree** interface
(vilkaarlig 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.1. Finn innsetningsnode

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

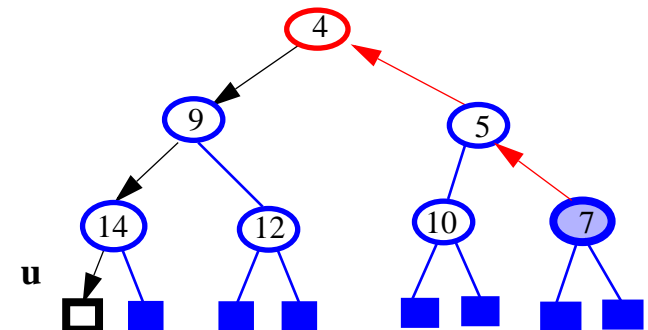
- **Sequence** (array):
sist = *n*; **u** = **sist** + 1 **O(1)**

```
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)
```

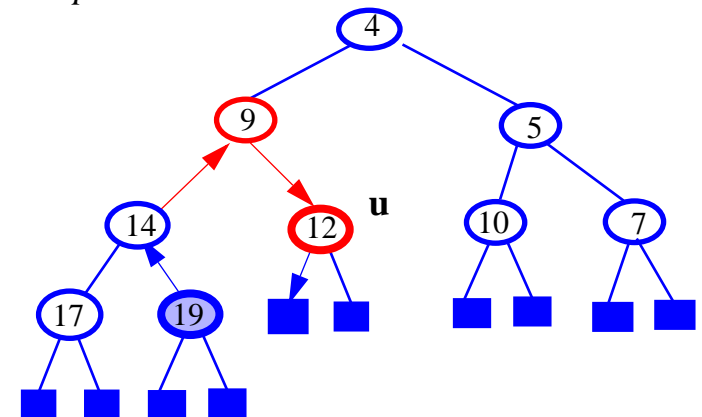
- **BinaryTree** interface
(vilkaarlig implementasjon)

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.1. Finn innsetningsnode

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

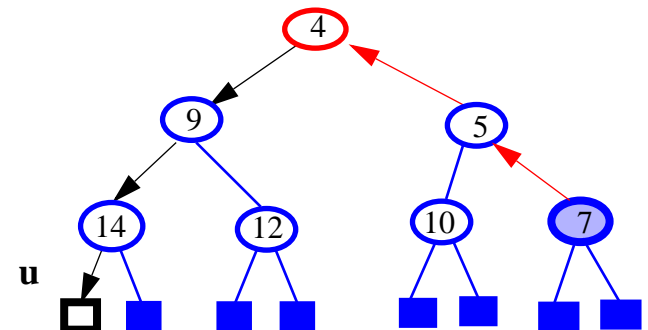
- **Sequence** (array):
`sist = n; u = sist + 1` **O(1)**

```

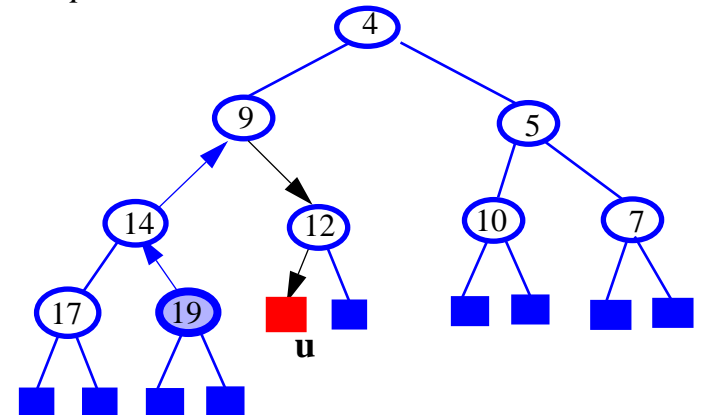
u = sist
while ( u != root()
    && u != leftChild(parent(u)) )
    u = parent(u);
if ( u != root )
    u = rightChild(parent(u));
while ( ! isExternal(u) )
    u = leftChild(u);
return u
O(log n)
    
```

- **BinaryTree** interface
 (vilkaarlig implementasjon)
 Avhengig av hvem **sist** er har vi tre tilfeller:

- T.isEmpty()* : **u** = *T.root()*
- ytterste noden på nivå *h-1*



c) en mellomnode på nivå *h-1*



A.1. Finn innsetningsnode

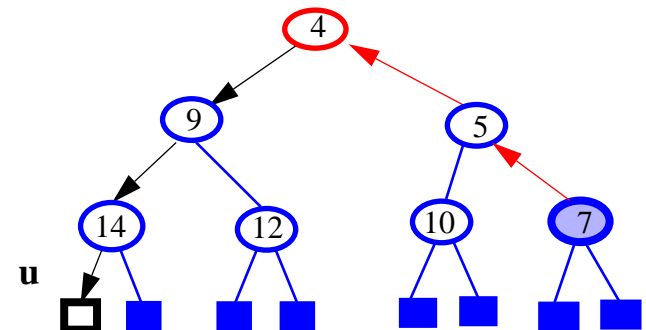
1. Gitt **sist**, finn innsetningsnode **u**
– avhengig av implementasjon av **BinaryTree**

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

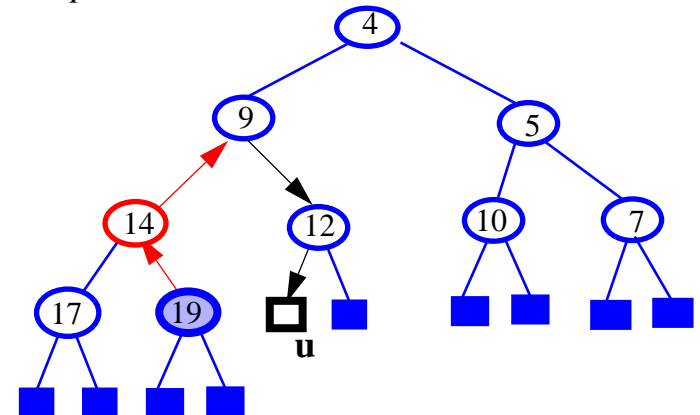
```
u = sist
while ( u != root()
      && u != leftChild(parent(u)) )
    u = parent(u);
if ( u != root()
    u = rightChild(parent(u));
while ( ! isExternal(u) )
    u = leftChild(u);
return u O(log n)
```

- **BinaryTree** interface
(vilkaarlig implementasjon)
Avhengig av hvem **sist** er har vi tre tilfeller:

- a) *T.isEmpty()* : **u** = *T.root()*
b) ytterste noden på nivå *h-1*

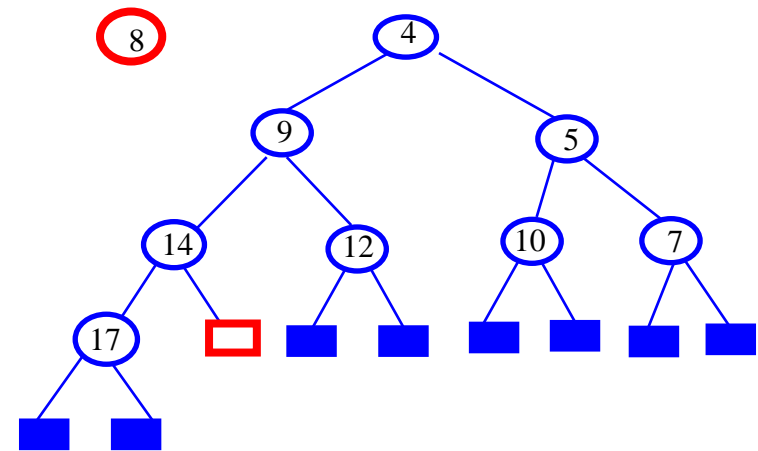


- c) en mellomnode på nivå *h-1*



A.2. “Oppover bobling”

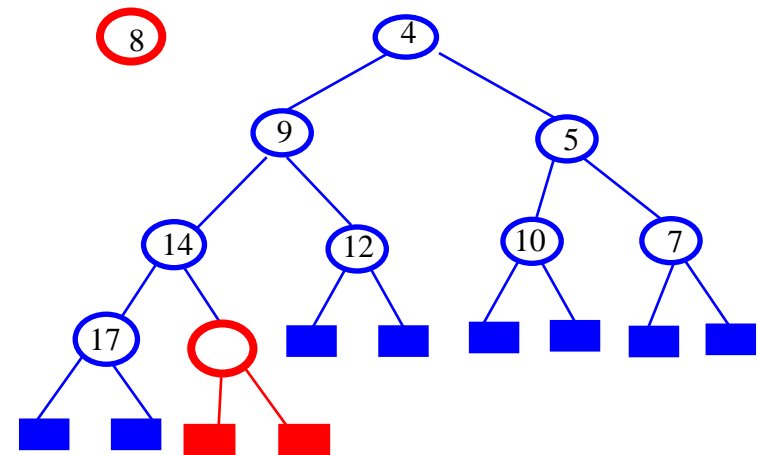
```
u = siste A.1  
while ( u != root() && u != leftChild(parent(u)) )  
    u = parent(u)  
if ( u != root() u = rightChild(parent(u))  
while ( ! isExternal(u) ) u = leftChild(u)  
return u O(log n)
```



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)
```

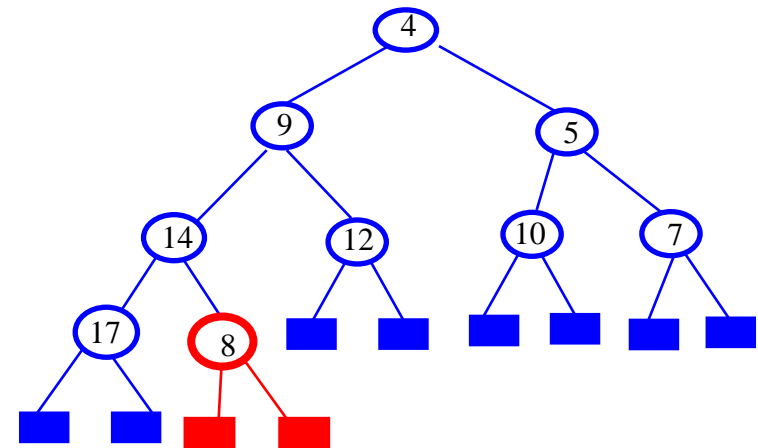
```
expandExternal(u) ;  
u.setElement(ny) ;
```



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)
```

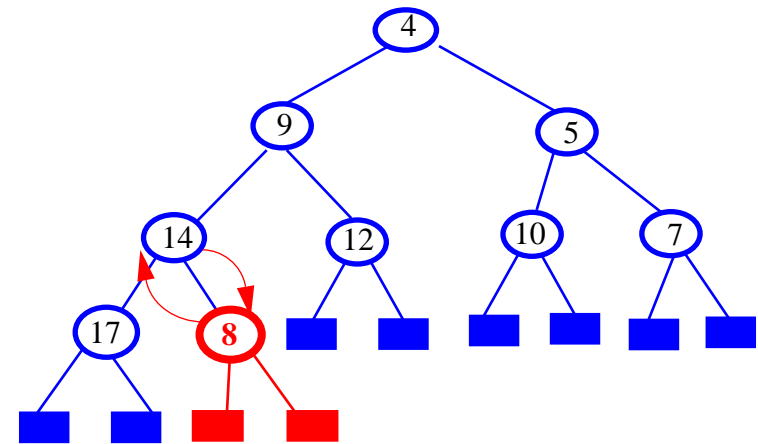
```
expandExternal(u) ;  
u.setElement(ny) ;
```



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) ;
```

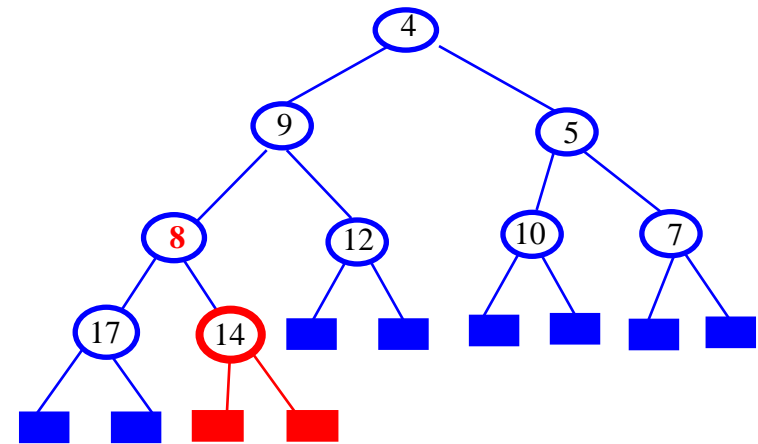


```
while ( u != root() && cp.isLessThan( u.element(), parent(u).element() ) ) A.2  
    swap(u , parent(u))  
    u = parent(u); O(log n)
```

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.setElement(ny) ;
```

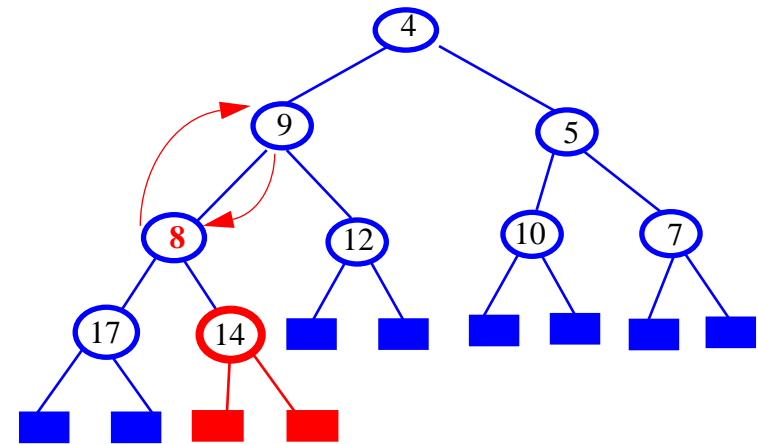


```
while ( u != root() && cp.isLessThan( u.element(), parent(u).element() ) ) A.2  
    swap(u , parent(u))  
    u = parent(u); O(log n)
```

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.setElement(ny) ;
```

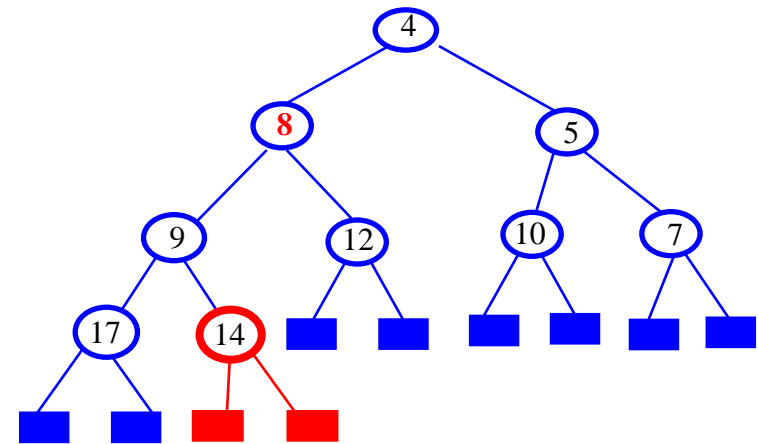


```
while ( u != root() && cp.isLessThan( u.element(), parent(u).element() ) ) A.2  
    swap(u , parent(u))  
    u = parent(u); O(log n)
```


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)
```

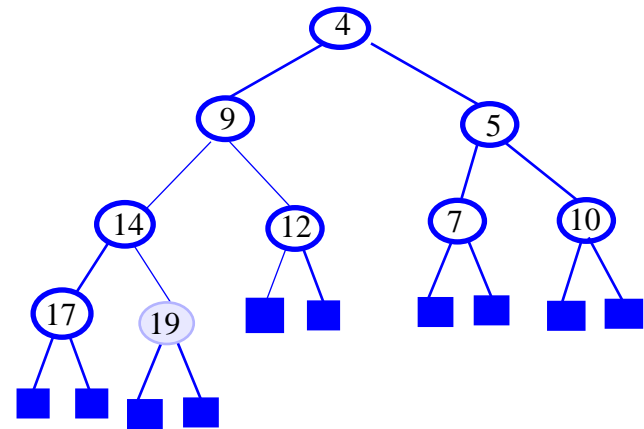
```
expandExternal(u) ;  
u.setElement(ny) ;
```



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

B. *removeMinElement()* fra Heap

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



B. *removeMinElement()* fra Heap

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

2. **Bevar KOMPLETT BINARYTREE INVARIANT :**

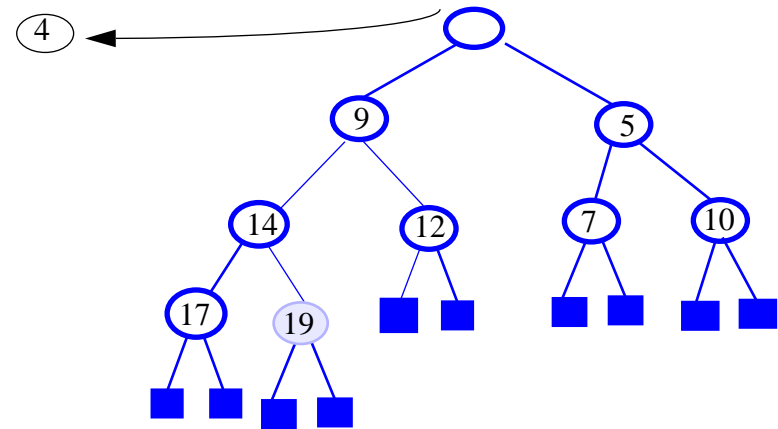
– *plasser sist.element() i rot-posisjon*

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

(BinTree: `removeAboveExternal(leftChild(sist))`)

– *oppdater sist*

$O(1 + \log n)$



B. *removeMinElement()* fra Heap

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

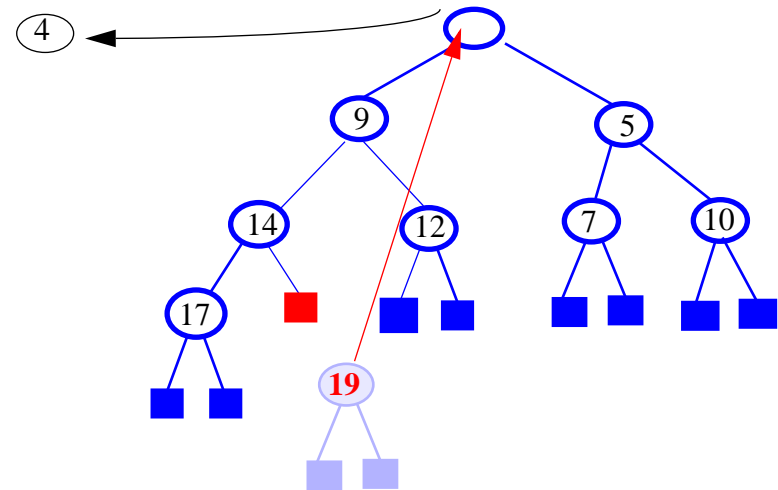
2. **Bevar KOMPLETT BINARYTREE INVARIANT :**

- *plasser sist.element() i rot-posisjon*
- *og fjern sist-posisjon = sett inn et nytt blad*

(BinTree: `removeAboveExternal(leftChild(sist))`)

– *oppdater sist*

$O(1 + \log n)$



B. *removeMinElement()* fra Heap

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

2. **Bevar KOMPLETT BINARYTREE INVARIANT :**

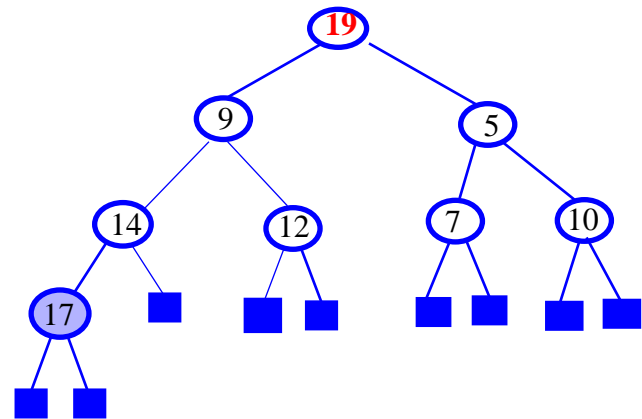
- *plasser `sist.element()` i rot-posisjon*
- *og fjern `sist-posisjon` = sett inn et nytt blad*

(BinTree: `removeAboveExternal(leftChild(sist))`)

– *oppdater `sist`*

$O(1 + \log n)$

④



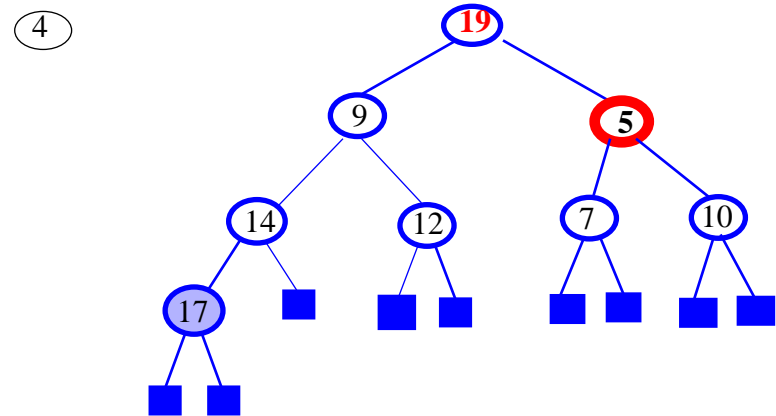
B. *removeMinElement()* fra Heap

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”)

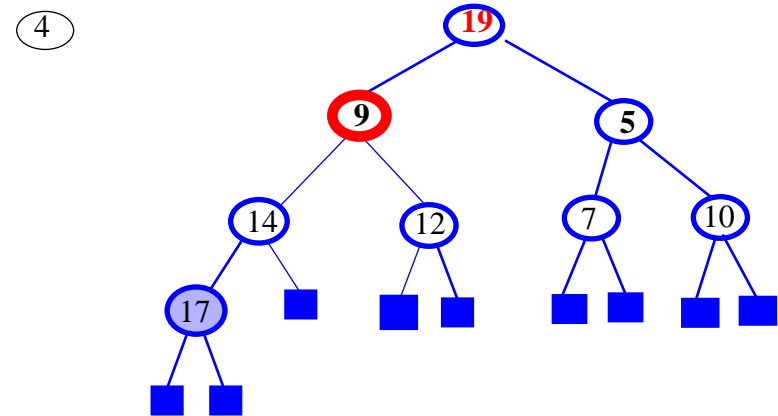
B. *removeMinElement()* fra Heap

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”)*

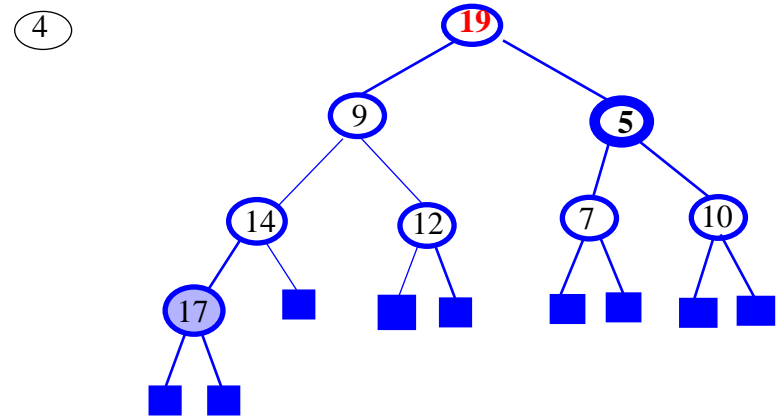
B. *removeMinElement()* fra Heap

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”)*

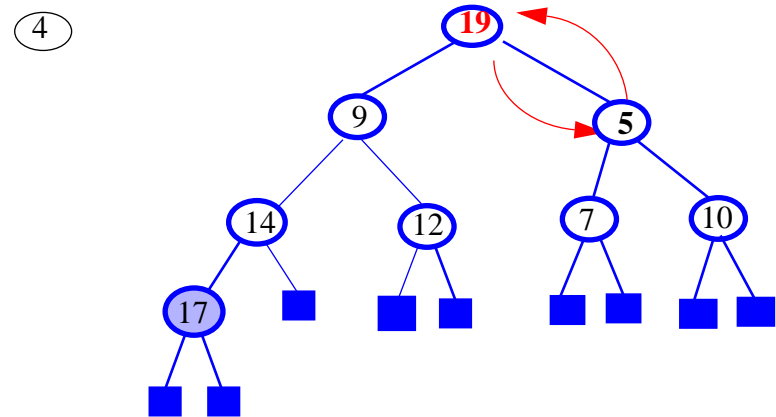
B. *removeMinElement()* fra Heap

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. **Gjenoppsett 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
  }
}

```

$O(\log n)$

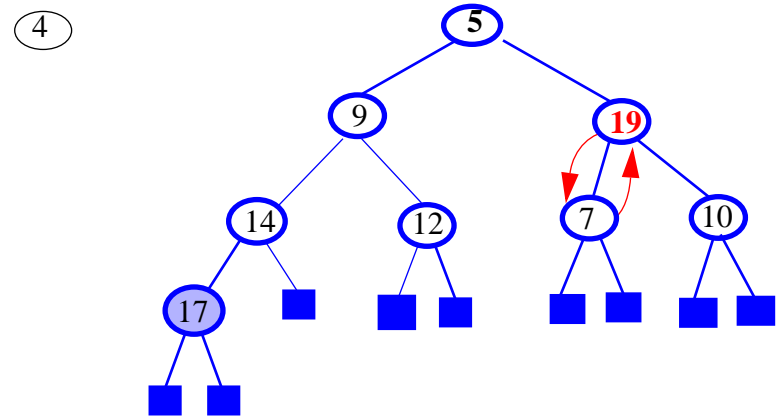
B. *removeMinElement()* fra Heap

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. Gjenoppsett **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
  }
}

```

$O(\log n)$

B. *removeMinElement()* fra Heap

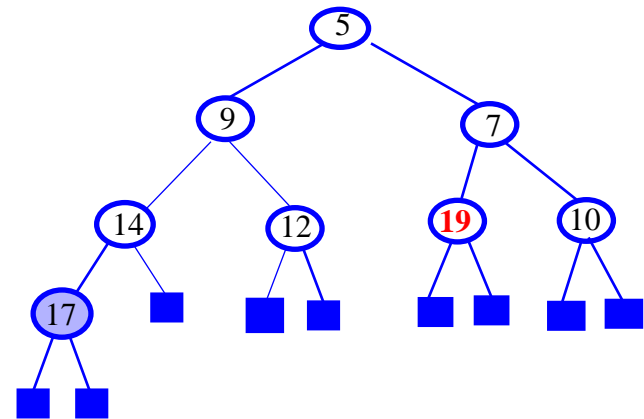
1. hold root-Objektet (til return)
`((Item)root().element()).element()`

(4)

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. Gjenoppsett **HEAP-ORDERING INVARIANT**:

- flytt det nye rot-elementet “ned” til passende posisjon (“nedover bobling”)

```

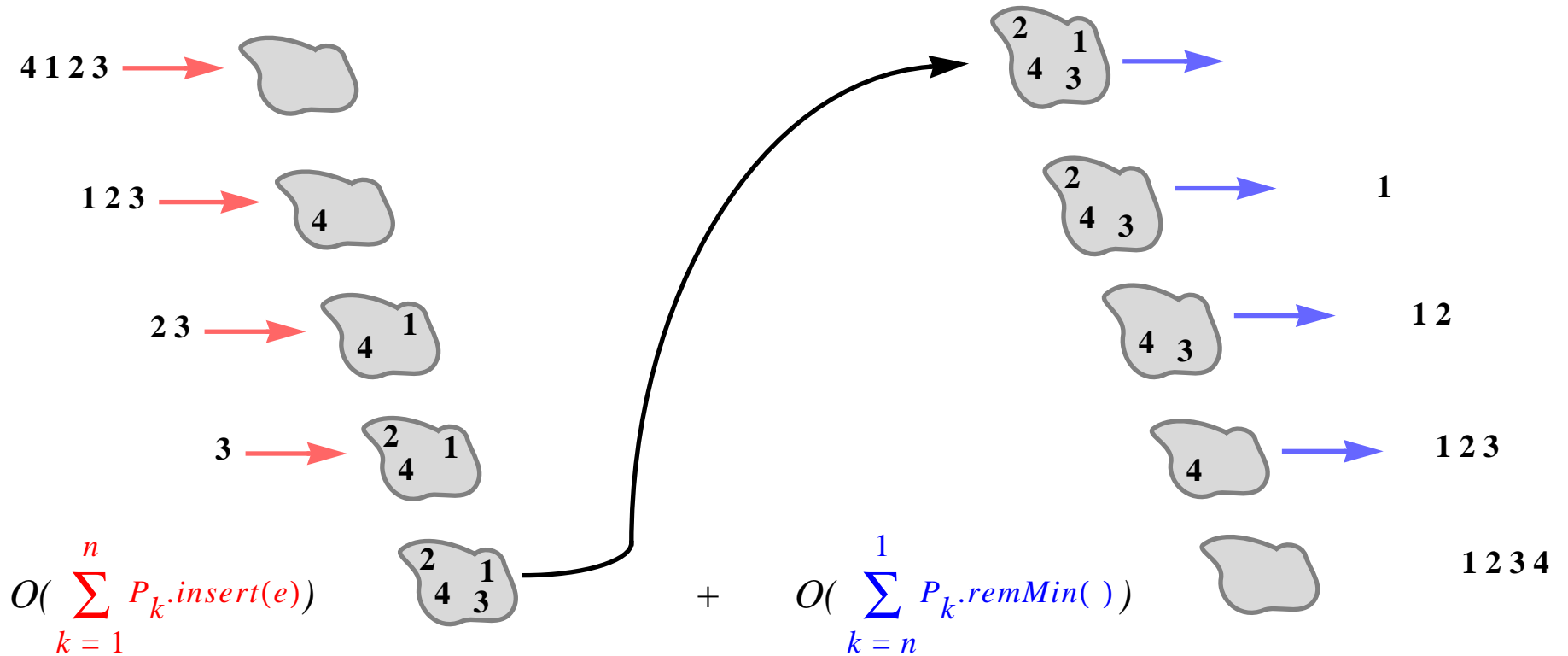
u = root(); done = false;
while ( ! done ) {
  if ( isExternal(leftChild(u)) && isExternal(rightChild(u)) ) done = true
  else
  {
    if ( isExternal(rightChild(u)) ) neste = leftChild(u)
    else if ( cp.isLessThan( leftChild(u).element(), rightChild(u).element() ) )
      neste = leftChild(u)
    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 \xrightarrow{n} **u:** 4 1 2 3 $\xrightarrow{\frac{n^2+n}{2}}$ 1 2 3 4 $:O(n^2)$

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

Heap Sort 4 1 2 3 $\xrightarrow{n \log(n)}$ **h:** $\xrightarrow{n \log(n)}$ 1 2 3 4 $:O(n \log(n))$

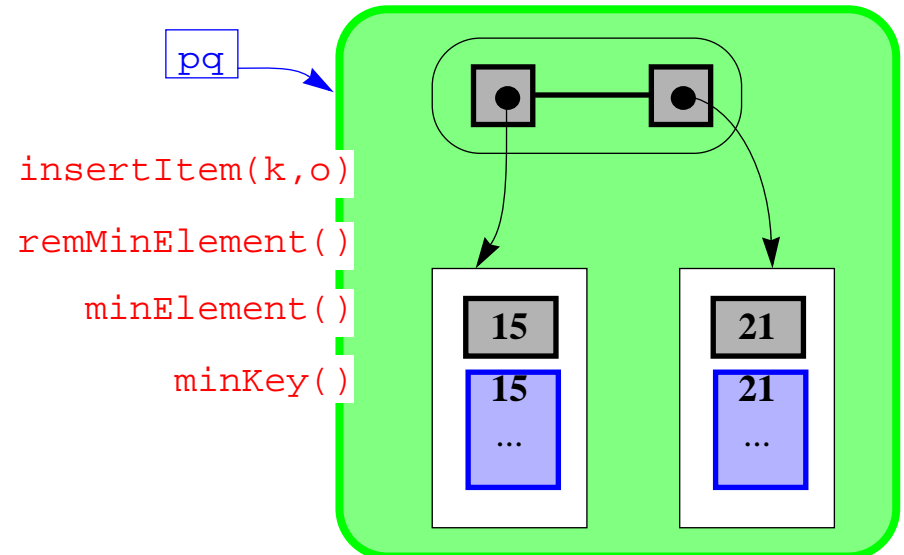
VI. Oppdater aldri objekter i en Samling!

fordi nøkkel-verdi avhenger, typisk, av Objektets attributter

VI. Oppdater aldri objekter i en Samling!

fordi nøkkel-verdi avhenger, typisk, av Objektets attributter

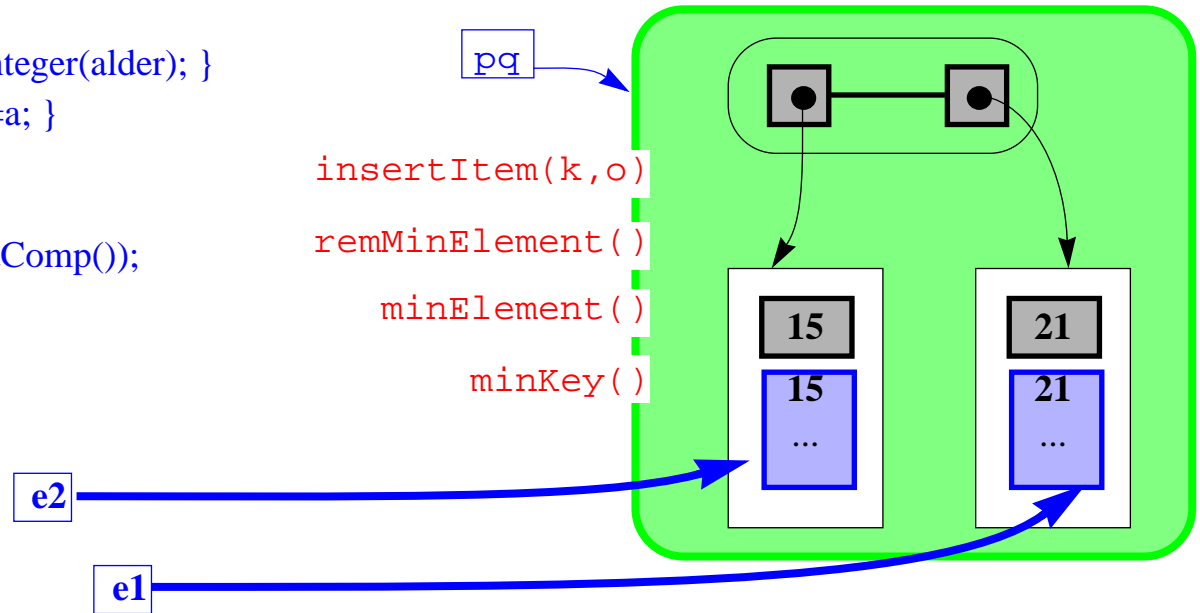
```
class El {  
    private int alder;  
    private int pnr;  
    public Object key() { return new Integer(alder); }  
    public void setAlder(int a) { alder=a; }  
    ... }  
  
PriorityQueue pq = new PQSeq(new intComp());  
El e1= new El(21);  
El e2= new El(15);  
pq.insertItem( e2.key(), e2 );  
pq.insertItem( e1.key(), e1 );
```



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.key(), e2 );  
pq.insertItem( e1.key(), e1 );
```



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());
```

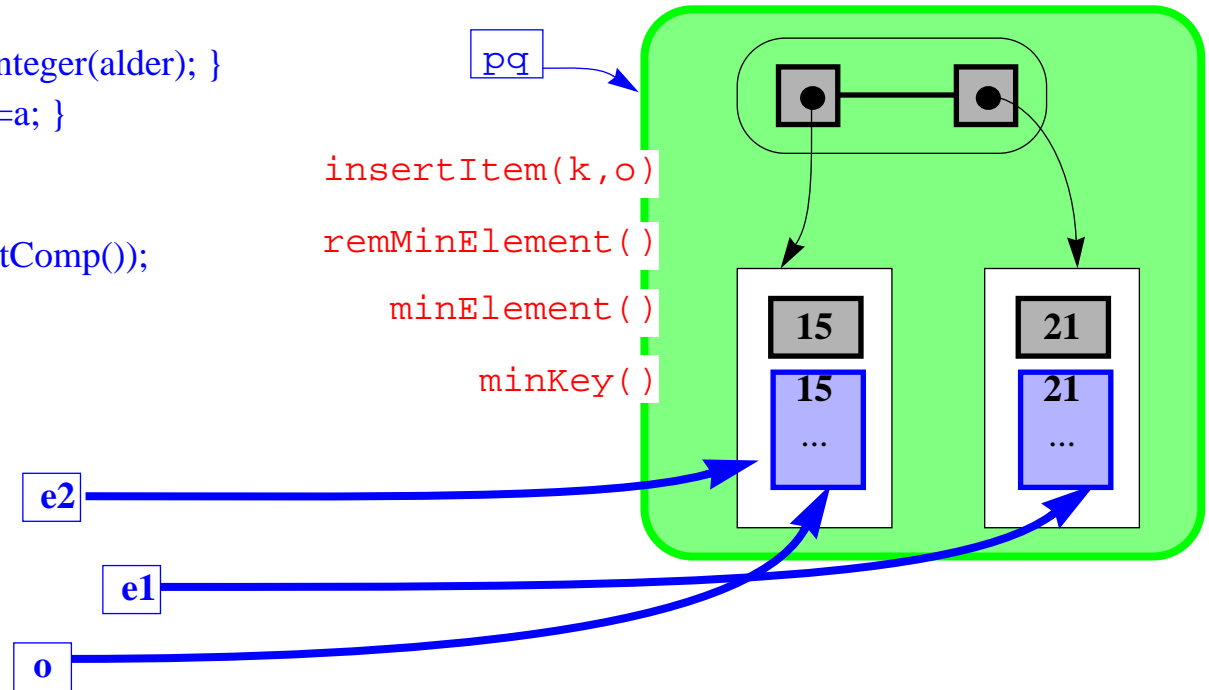
```
El e1= new El(21);
```

```
El e2= new El(15);
```

```
pq.insertItem( e2.key(), e2 );
```

```
pq.insertItem( e1.key(), e1 );
```

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



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());
```

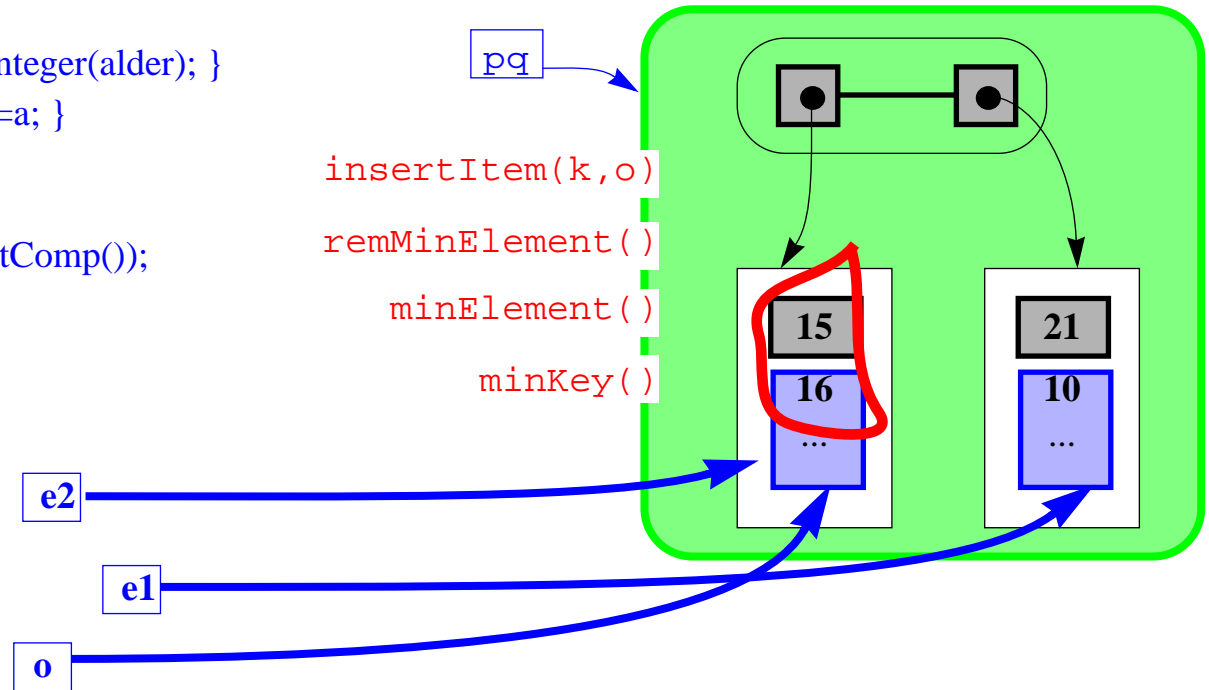
```
El e1= new El(21);
```

```
El e2= new El(15);
```

```
pq.insertItem( e2.key(), e2 );
```

```
pq.insertItem( e1.key(), e1 );
```

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



```
o . setAlder(16);
```

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());
```

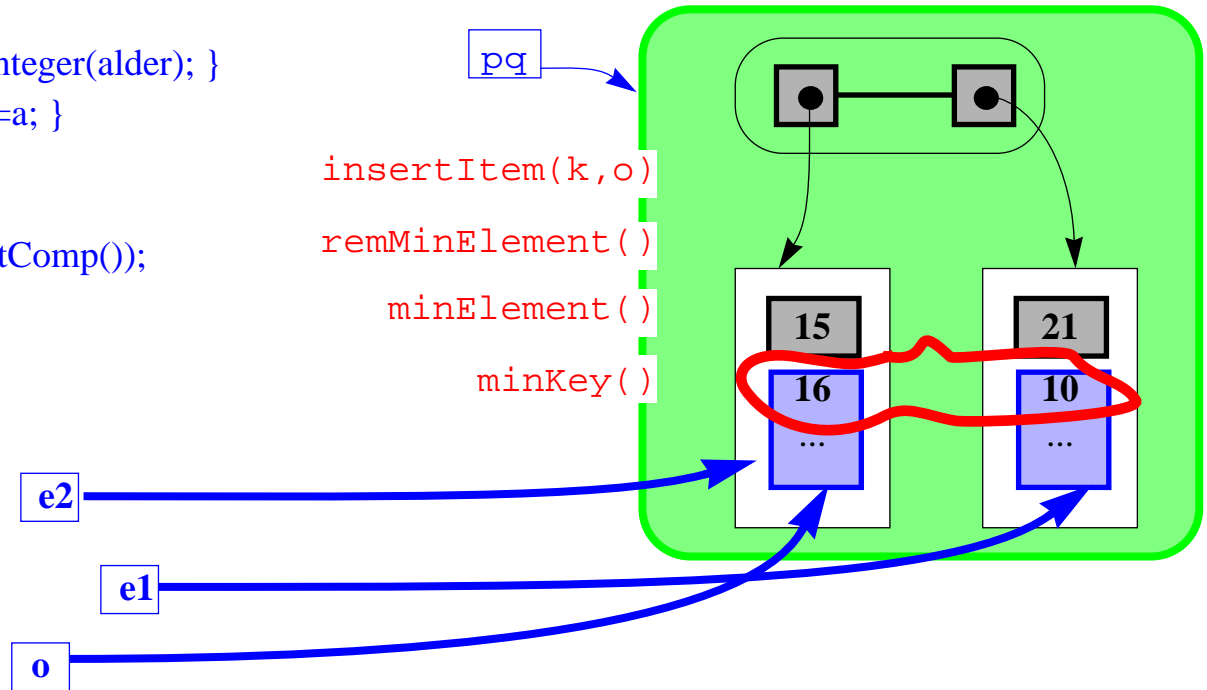
```
El e1= new El(21);
```

```
El e2= new El(15);
```

```
pq.insertItem( e2.key(), e2 );
```

```
pq.insertItem( e1.key(), e1 );
```

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



```
o . setAlder(16);
```

```
e1 . setAlder(10);
```

Et godt - metodologisk - råd !!!

1. IMPLEMENTER DATA INVARIANT

I eksemplet over:

- for hver Item i har vi at: $i.element().key() == i.key()$
- sekvens er sortert

Et godt - metodologisk - råd !!!

1. IMPLEMENTER 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:

Et godt - metodologisk - råd !!!

1. IMPLEMENTER 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

Et godt - metodologisk - råd !!!

1. IMPLEMENTER 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 kompleksitet*
3. *Resulterende kode er betydelig sikrere*

Et godt - metodologisk - råd !!!

1. IMPLEMENTER 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 kompleksitet*
3. *Resulterende kode er betydelig sikrere*

4. BRUK MER SOFISTIKERT GRENSESNIITT ...

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 {
```

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


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) ;
```

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

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) ;
```

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

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 stored previously at L */
```

```
    Object replaceElement(Locator L , Object o) ;
```

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

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 stored 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) ;
```

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

VI. *Locator* designmønster

Position – men for samlinger der Objekter lokaliseres v.h.j.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 stored 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) ;
```

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

implementasjon vil kreve en god del manipulering med datastrukturen

VI. *Locator* designmønster

Position – men for samlinger der Objekter lokaliseres v.h.j.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 stored 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()  
}
```

implementasjon vil kreve en god del manipulering med datastrukturen

VI. *Locator* designmønster

Position – men for samlinger der Objekter lokaliseres v.h.j.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 stored 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() ;  
}
```

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

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

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

implementasjon vil kreve en god del manipulering med datastrukturen

f. eks.:

operasjon	output	PQ (sortert sekvens)
PQ . insert(5,A)	L1	<5,A>

f. eks.:

operasjon	output	PQ (sortert sekvens)
PQ . insert(5,A)	L1	<5,A>
PQ . insert(3,B)	L2	<3,B> <5,A>

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>

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	

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	

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>

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>

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	

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>

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	

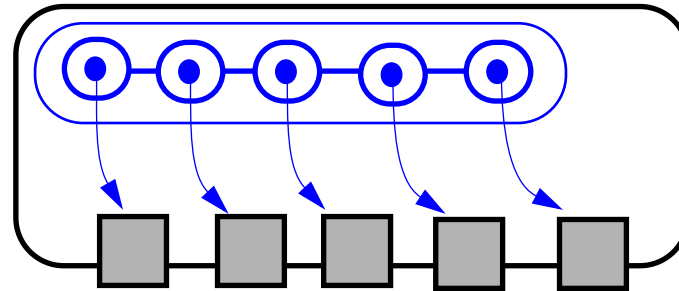
f. eks.:

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

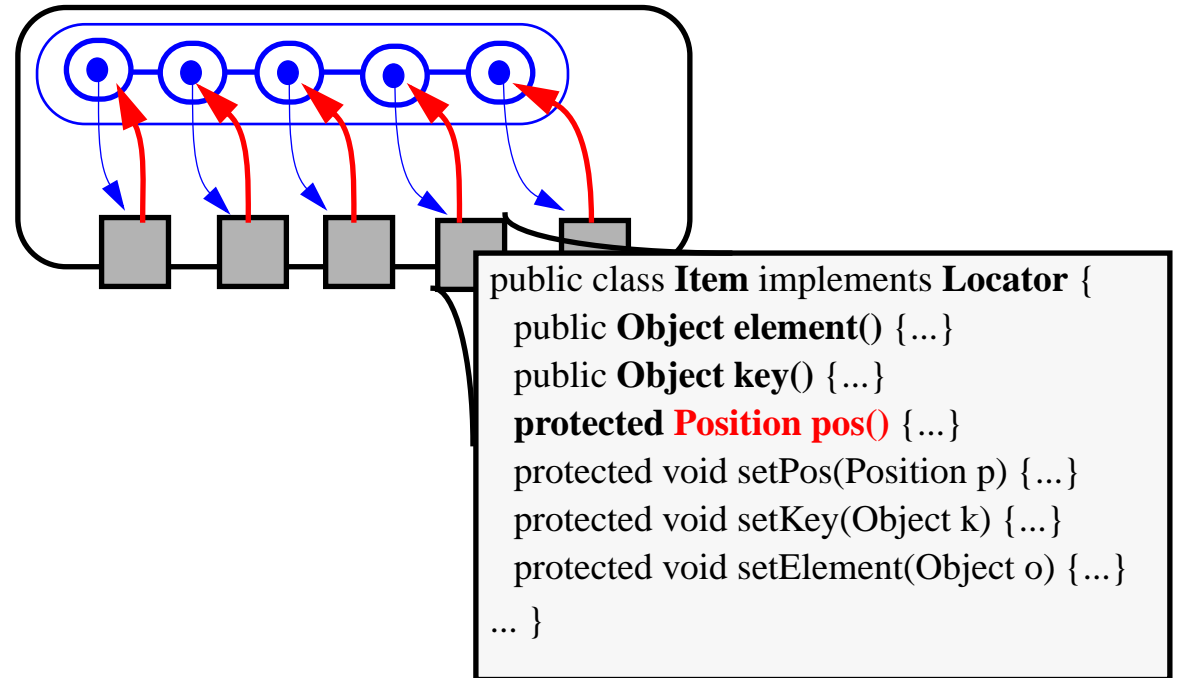


Implementasjon av PriorityQueue – med Locator

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en **Locator** Item

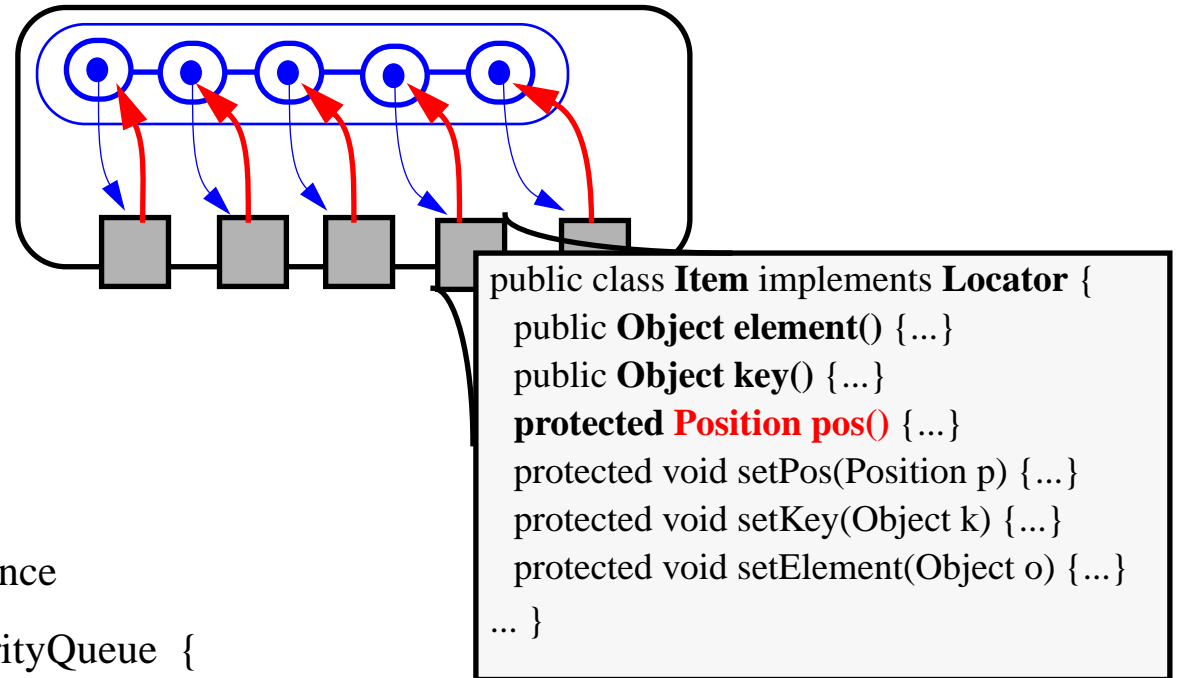


Implementasjon av PriorityQueue – med Locator

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en **Locator** Item



II DATA STRUKTUR

class PQ-L-Sequence extends PQSequence

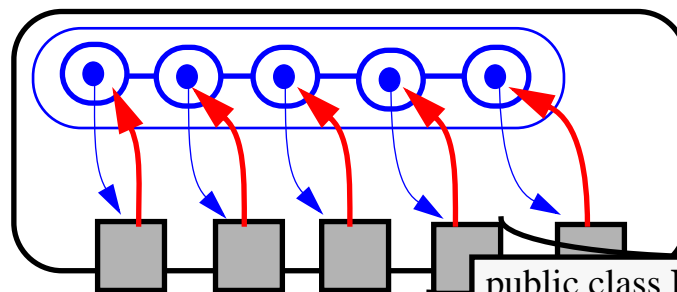
implements PriorityQueue {

Implementasjon av PriorityQueue – med Locator

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en **Locator** Item



II DATA STRUKTUR

```
class PQ-L-Sequence extends PQSequence
    implements PriorityQueue {
```

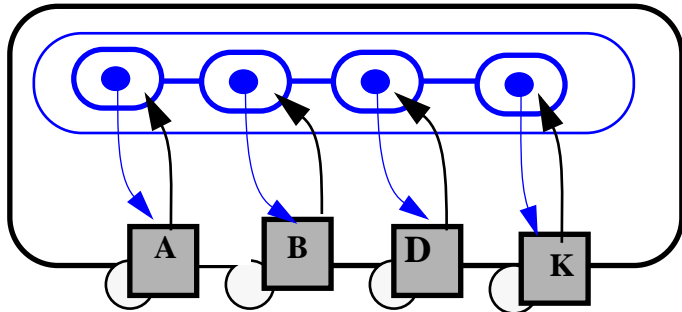
```
public class Item implements Locator {
    public Object element() {...}
    public Object key() {...}
    protected Position pos() {...}
    protected void setPos(Position p) {...}
    protected void setKey(Object k) {...}
    protected void setElement(Object o) {...}
    ... }
```

III DATA INVARIANT

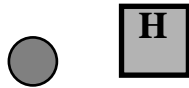
- Item (i Locator) kan sammenlignes med cp
- For hver posisjon **p** i **S**:
((Locator) **p.element()**) . **pos()** == **p**
- S er sortert/usortert

Sortert Sequence implementasjon med Locator

S:

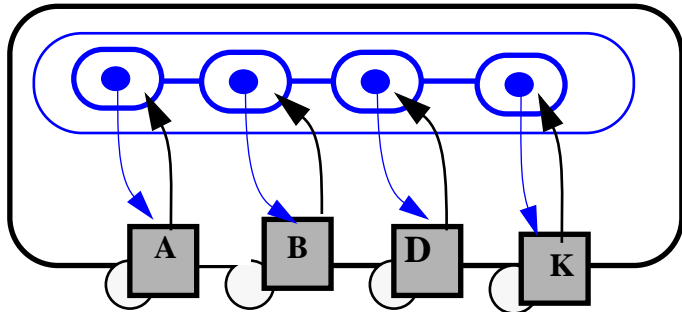


insert ("H", o)

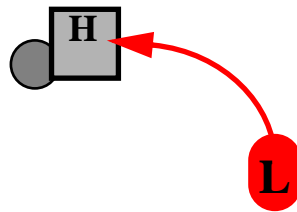


Sortert Sequence implementasjon med Locator

S:

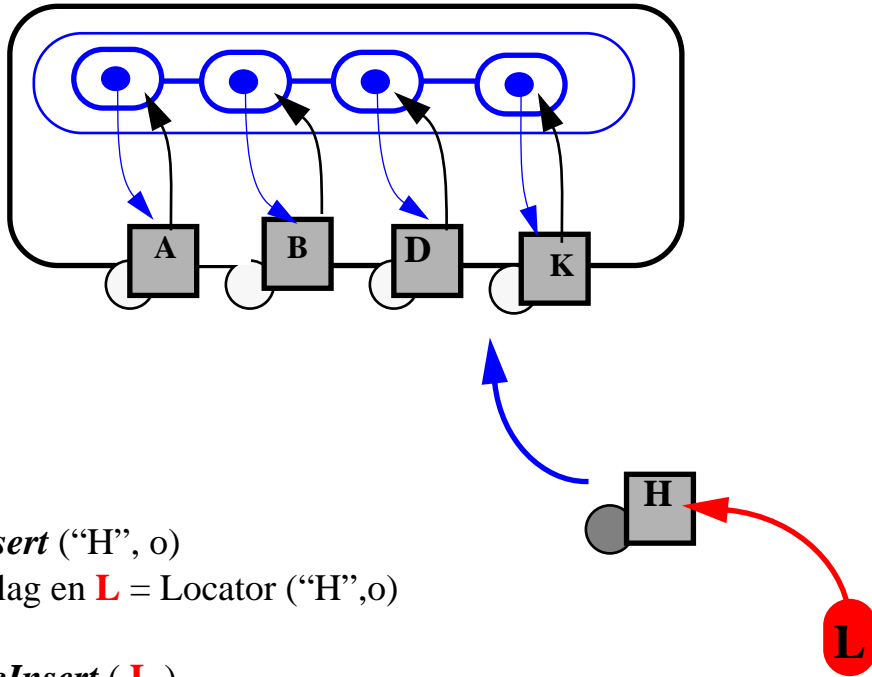


insert ("H", o)
-- lag en **L** = Locator ("H",o)



Sortert Sequence implementasjon med Locator

S:



insert ("H", o)

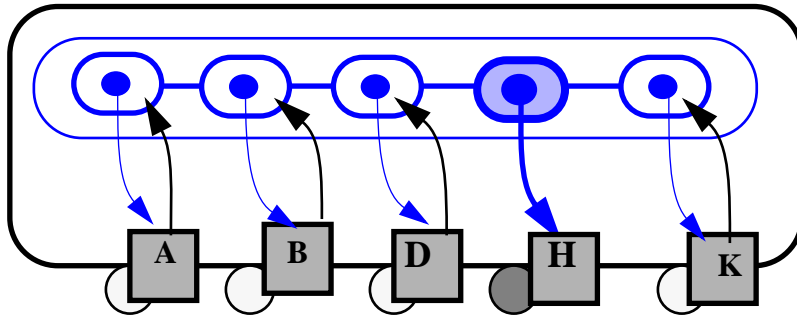
-- lag en **L** = Locator ("H",o)

locInsert (**L**)

-- sett den inn i S (returnerer **Position p**)

Sortert Sequence implementasjon med Locator

S:



insert ("H", o)

-- lag en **L** = Locator ("H",o)

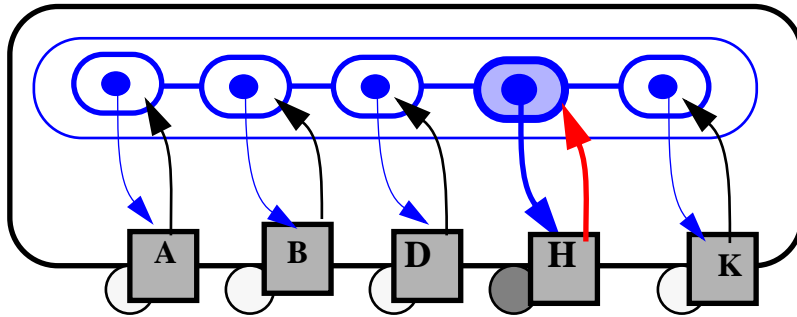
locInsert (**L**)

-- sett den inn i S (returnerer **Position p**)

L

Sortert Sequence implementasjon med Locator

S:



insert ("H", o)

-- lag en **L** = Locator ("H",o)

locInsert (**L**)

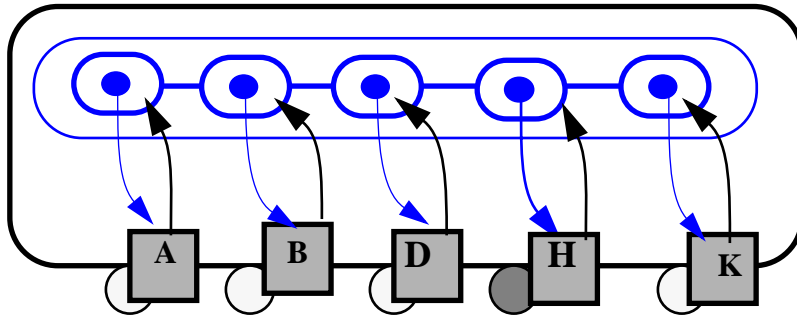
-- sett den inn i S (returnerer **Position p**)

-- **L** . *setPos*(**p**)

L

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**)

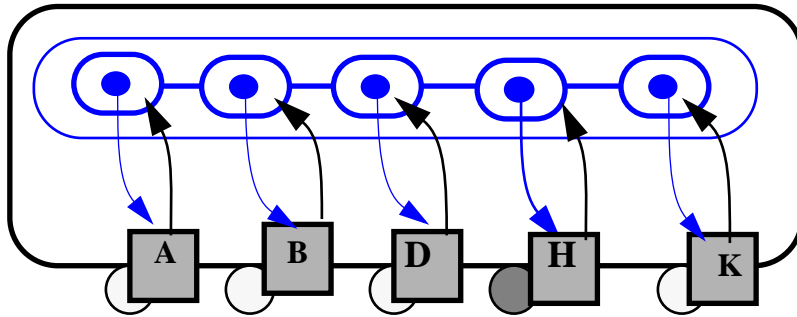
-- **L** . setPos(**p**)

-- returner **L**

L

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**

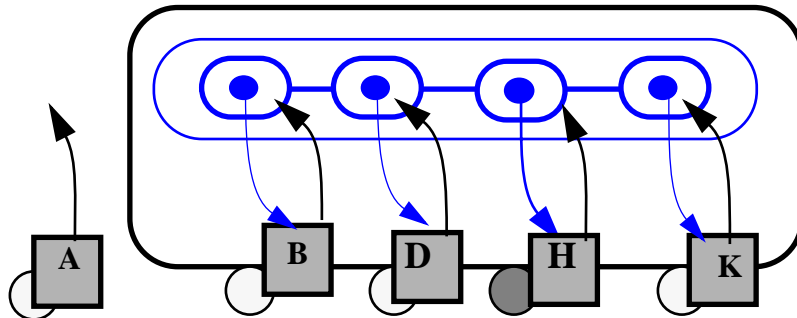
removeMin ():

-- S . remove(S.first())

L

Sortert Sequence implementasjon med Locator

S:



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

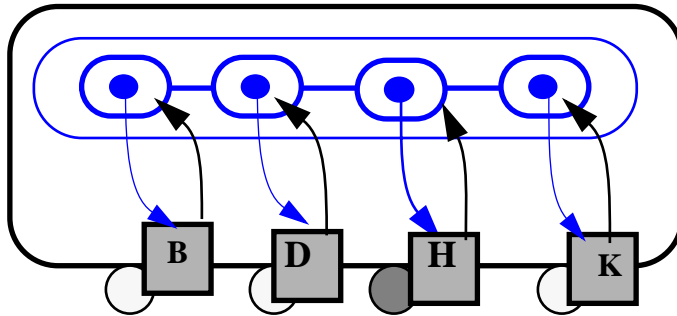
locInsert (**L**)
-- sett den inn i S (returnerer Position **p**)
-- **L** . setPos(**p**)
-- returner **L**

removeMin ():

-- S . remove(S.first())

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**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()

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

-- **L** . setKey("C")

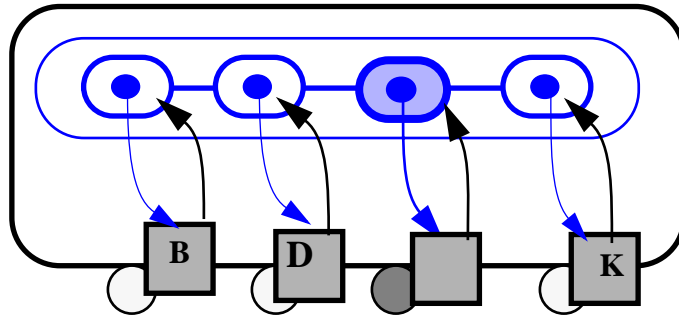
-- locInsert(**L**)

-- return oldKey

L

Sortert Sequence implementasjon med Locator

S:



H

insert ("H", o)

-- lag en **L** = Locator ("H",o)

locInsert (**L**)

-- sett den inn i S (returnerer Position **p**)

-- **L** . setPos(**p**)

-- returner **L**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object **oldKey** = **L** . key()

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

-- **L** . setKey("C")

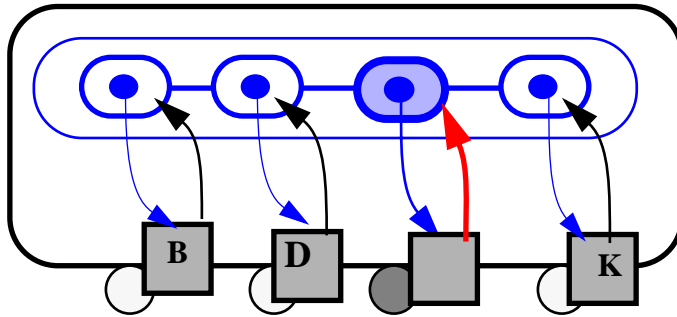
-- locInsert(**L**)

-- return oldKey

L

Sortert Sequence implementasjon med Locator

S:



H

insert ("H", o)

-- lag en **L** = Locator ("H",o)

locInsert (**L**)

-- sett den inn i S (returnerer Position **p**)

-- **L** . setPos(**p**)

-- returner **L**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()

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

-- **L** . setKey("C")

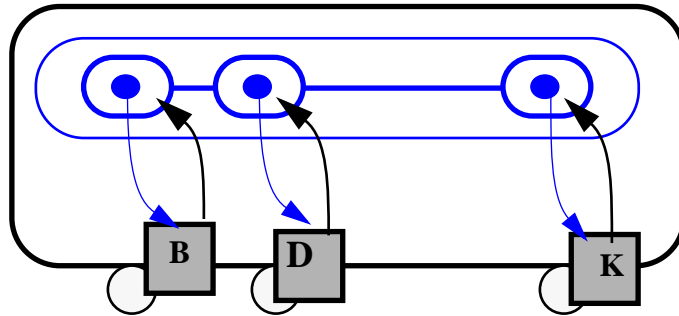
-- locInsert(**L**)

-- return oldKey

L

Sortert Sequence implementasjon med Locator

S:



H

insert ("H", o)

-- lag en **L** = Locator ("H",o)

locInsert (**L**)

-- sett den inn i S (returnerer Position **p**)

-- **L** . setPos(**p**)

-- returner **L**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()

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

-- **L** . setKey("C")

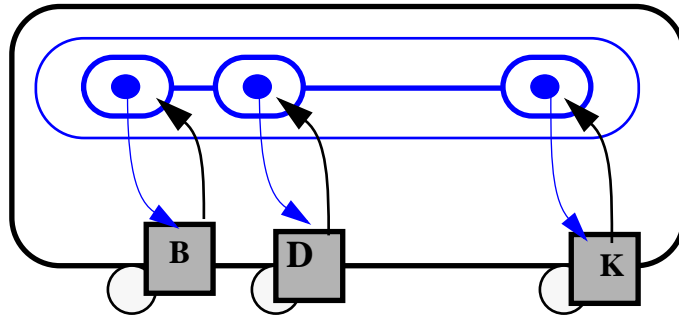
-- locInsert(**L**)

-- return oldKey

L

Sortert Sequence implementasjon med Locator

S:



H

insert ("H", o)

-- lag en **L** = Locator ("H",o)

locInsert (**L**)

-- sett den inn i S (returnerer Position **p**)

-- **L** . setPos(**p**)

-- returner **L**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()

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

-- **L** . **setKey**("C")

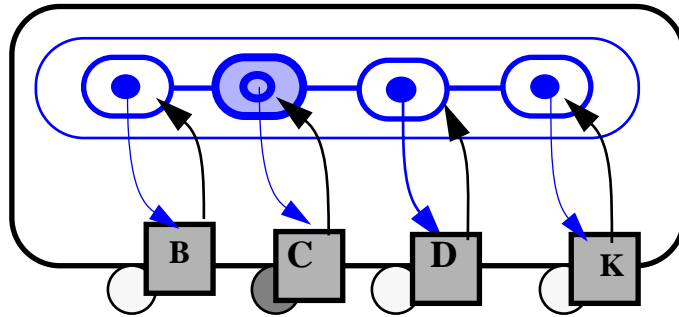
-- locInsert(**L**)

-- return oldKey

L

Sortert Sequence implementasjon med Locator

S:



H

insert ("H", o)

-- lag en **L** = Locator ("H",o)

locInsert (**L**)

-- sett den inn i S (returnerer Position **p**)

-- **L** . setPos(**p**)

-- returner **L**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()

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

-- **L** . setKey("C")

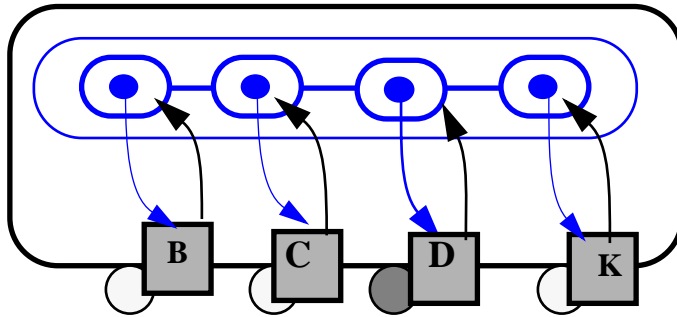
-- **locInsert**(**L**)

-- return oldKey

L

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**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()
-- S . remove(((Item)**L**) . pos())
-- **L** . setKey("C")
-- locInsert(**L**)
-- return oldKey

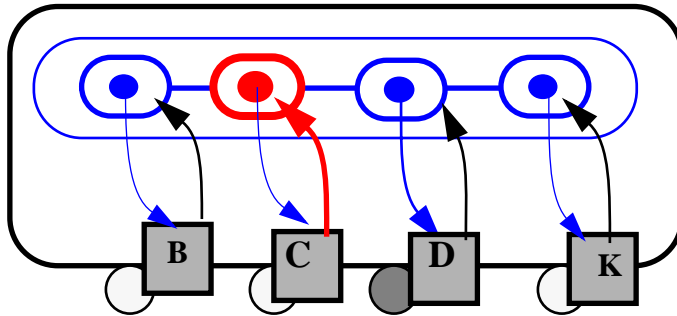
remove (**L**) :

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

L

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**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()
-- S . remove(((Item)**L**) . pos())
-- **L** . setKey("C")
-- locInsert(**L**)
-- return oldKey

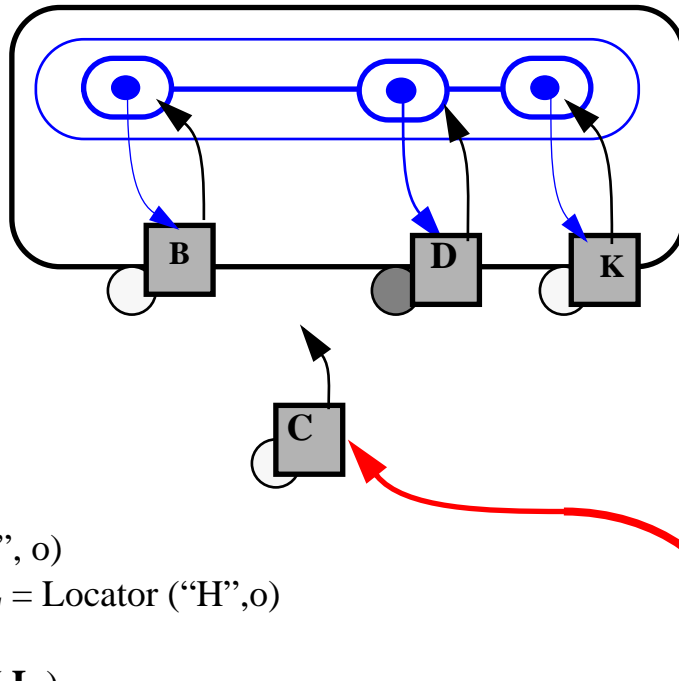
remove (**L**) :

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

L

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**

removeMin ():

-- S . remove(S.first())

replaceKey (**L**, "C")

-- Object oldKey = **L** . key()
-- S . remove(((Item)**L**) . pos())
-- **L** . setKey("C")
-- locInsert(**L**)
-- return oldKey

remove (**L**) :

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

Oppsummering

Typer av Samling

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

Oppsummering

Typer av Samling

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

Nøkkel:

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

Oppsummering

Typer av Samling

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

Nøkkel:

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

<i>Prioritetskø ADT :</i>	<i>implementasjon</i> og	<i>sorteringsmønster</i>
	<i>usortert sekvens</i>	<i>seleksjonsort</i>
	<i>sortert sekvens</i>	<i>innstikksort</i>
	<i>heap</i>	<i>heapsort</i>

Oppsummering

Typer av Samling

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

Nøkkel:

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

<i>Prioritetskø ADT :</i>	<i>implementasjon</i> og	<i>sorteringsmønster</i>
	<i>usortert sekvens</i>	<i>seleksjonsort</i>
	<i>sortert sekvens</i>	<i>innstikksort</i>
	<i>heap</i>	<i>heapsort</i>

Heap datastruktur:

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

Oppsummering

Typer av Samling

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

Nøkkel:

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

Prioritetskø ADT :	<i>implementasjon</i> og	<i>sorteringsmønster</i>
	<i>usortert sekvens</i>	<i>seleksjonsort</i>
	<i>sortert sekvens</i>	<i>innstikksort</i>
	<i>heap</i>	<i>heapsort</i>

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*