

Samlinger, Lister, Sekvenser

Litt om ADT-hierarki

Container ADT

Iterator – Enumeration ADT

Vektor ADT

Position ADT

Liste ADT

Sekvens ADT

Sammenlikning av implementasjoner vha array/lenket liste/2-veis lenket liste

Generisk Sortering av Sekvenser

Kap.	unntatt	kursorisk
4	4.2.3, 4.2.4, 4.5	
5		5.1.4

Container ADT (package `jdsl.core.api`;))

```
/** generisk samling av Objekter – supertype for alle samlinger */  
public interface InspectableContainer {  
    /** @return true hviss samlingen er tom */  
        boolean isEmpty();  
  
    /** @return antall elementer i samlingen */  
        int size();  
  
    /** @return iterator av alle elementer */  
        ObjectIterator elements();  
  
    /** @return true hviss Accessor a er i samlingen */  
        boolean contains(Accessor a);  
}
```

```
/** generisk samling som tillatter oppdateringer */  
public interface Container extends InspectableContainer {  
    /** @return en ny tom samling, av samme klasse */  
        Container newContainer();  
  
    /** @return elementet lagret ved a før kallet */  
        Object replaceElement(Accessor a, Object newEl);  
}
```

Container ADT (package `jdsl.core.api`;))

```
/** generisk samling av Objekter – supertype for alle samlinger */  
public interface InspectableContainer {  
    /** @return true hviss samlingen er tom */  
        boolean isEmpty();  
  
    /** @return antall elementer i samlingen */  
        int size();  
  
    /** @return iterator av alle elementer */  
        ObjectIterator elements();  
  
    /** @return true hviss Accessor a er i samlingen */  
        boolean contains(Accessor a);  
}
```

```
/** generisk samling som tillatter oppdateringer */  
public interface Container extends InspectableContainer {  
    /** @return en ny tom samling, av samme klasse */  
        Container newContainer();  
  
    /** @return elementet lagret ved a før kallet */  
        Object replaceElement(Accessor a, Object newEl);  
}
```

Container ADT (package `jdsl.core.api`;))

```
/** generisk samling av Objekter – supertype for alle samlinger */
```

```
public interface InspectableContainer {
```

```
/** @return true hviss samlingen er tom */  
boolean isEmpty();
```

```
/** @return antall elementer i samlingen */  
int size();
```

```
/** @return iterator av alle elementer */  
ObjectIterator elements();
```

```
/** @return true hviss Accessor a er i samlingen */  
boolean contains(Accessor a);
```

```
}
```

```
/** generisk “posisjon – lagringsplass” */
```

```
public interface Accessor{
```

```
/** @return elementet lagret ved denne Accessor*/  
Object element();
```

```
}
```

```
/** generisk samling som tillatter oppdateringer */
```

```
public interface Container extends InspectableContainer {
```

```
/** @return en ny tom samling, av samme klasse */  
Container newContainer();
```

```
/** @return elementet lagret ved a før kallet */  
Object replaceElement(Accessor a, Object newEl);
```

```
}
```

Container ADT (package `jdsl.core.api`;))

```
/** generisk samling av Objekter – supertype for alle samlinger */
```

```
public interface InspectableContainer {  
/** @return true hvis samlingen er tom */  
    boolean isEmpty();  
/** @return antall elementer i samlingen */  
    int size();  
/** @return iterator av alle elementer */  
    ObjectIterator elements();  
/** @return true hvis Accessor a er i samlingen */  
    boolean contains(Accessor a);  
}
```

```
/** generisk samling som tillater oppdateringer */  
public interface Container extends InspectableContainer  
/** @return en ny tom samling, av samme klasse */  
    Container newContainer();  
/** @return elementet lagret ved a før kallet */  
    Object replaceElement(Accessor a, Object newEl);  
}
```

```
/** generisk “posisjon – lagringsplass” */
```

```
public interface Accessor{  
/** @return elementet lagret ved denne Accessor*/  
    Object element();  
}
```

```
/** tillater iterasjon over samling */
```

```
public interface ObjectIterator {  
/** @return neste objektet å besøke */  
    Object nextObject();  
/** @return true hvis det er flere objekter */  
    boolean hasNext();  
/** @return objektet returnert av siste nextObject */  
    Object object();  
/** @return setter i initiell tilstand (før kallet) */  
    void reset();  
}
```

Container ADT (package `jdsl.core.api`;))

```
/** generisk samling av Objekter – supertype for alle samlinger */
```

```
public interface InspectableContainer {  
/** @return true hvis samlingen er tom */  
    boolean isEmpty();  
/** @return antall elementer i samlingen */  
    int size();  
/** @return iterator av alle elementer */  
    ObjectIterator elements();  
/** @return true hvis Accessor a er i samlingen */  
    boolean contains(Accessor a);  
}
```

```
/** generisk “posisjon – lagringsplass” */
```

```
public interface Accessor{  
/** @return elementet lagret ved denne Accessor*/  
    Object element();  
}
```

```
/** tillater iterasjon over samling */
```

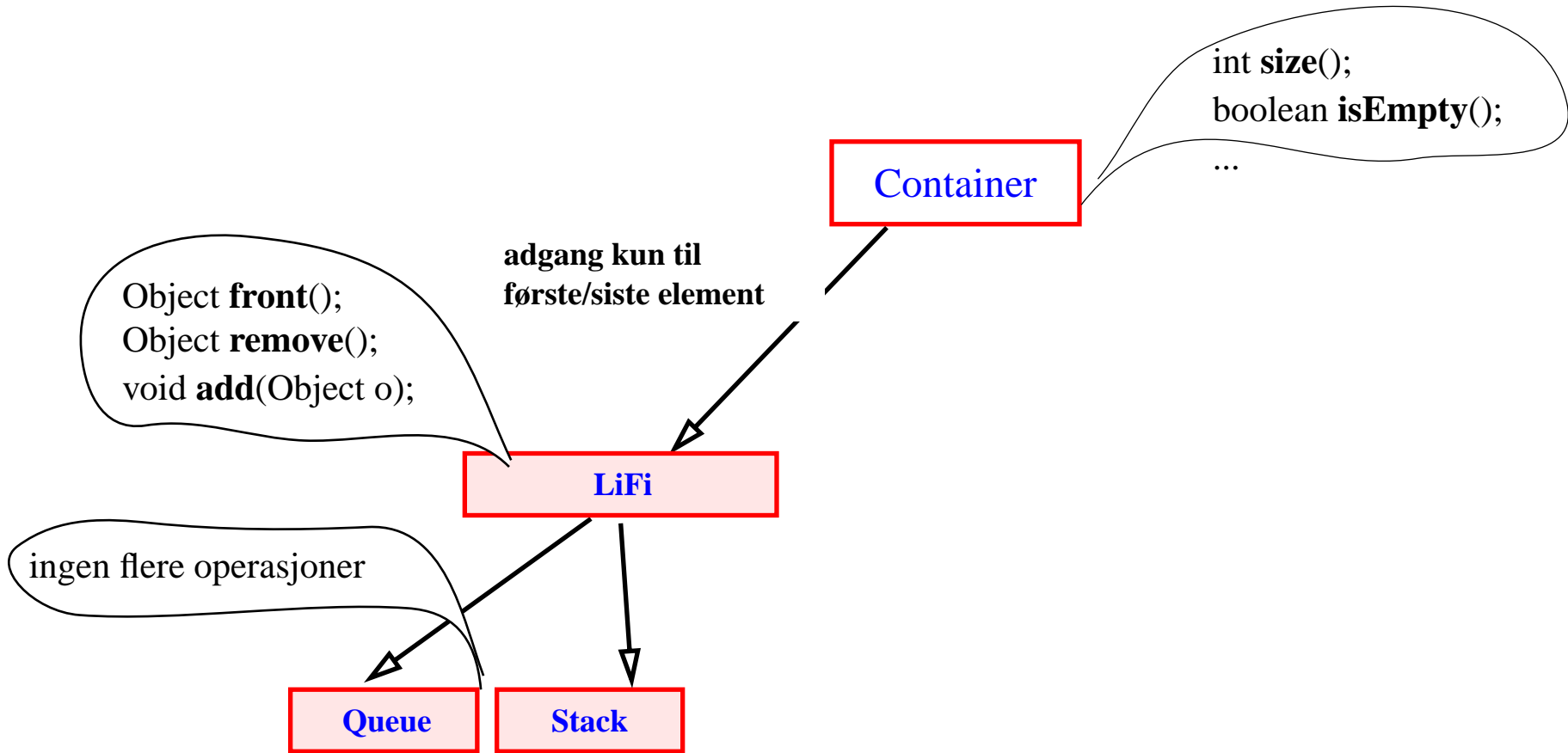
```
public interface ObjectIterator {  
/** @return neste objektet å besøke */  
    Object nextObject();  
/** @return true hvis det er flere objekter */  
    boolean hasNext();  
/** @return objektet returnert av siste nextObject */  
    Object object();  
/** @return setter i initiell tilstand (før kallet) */  
    void reset();  
}
```

```
/** generisk samling som tillater oppdateringer */
```

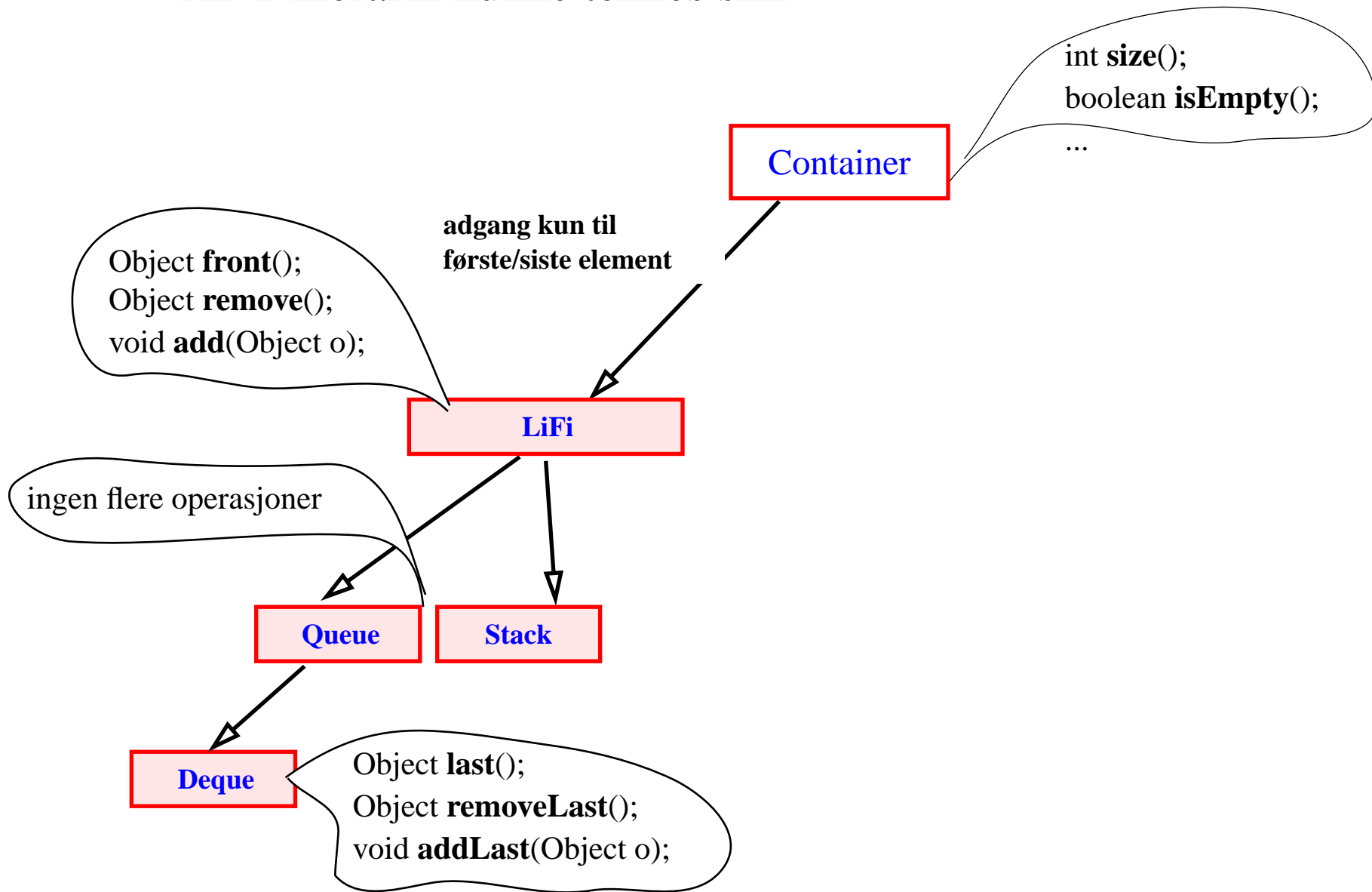
```
public interface Container extends InspectableContainer  
/** @return en ny tom samling, av samme klasse */  
    Container newContainer();  
/** @return elementet lagret ved a før kallet */  
    Object replaceElement(Accessor a, Object newEl);  
}
```

pluss Exceptions !!!!

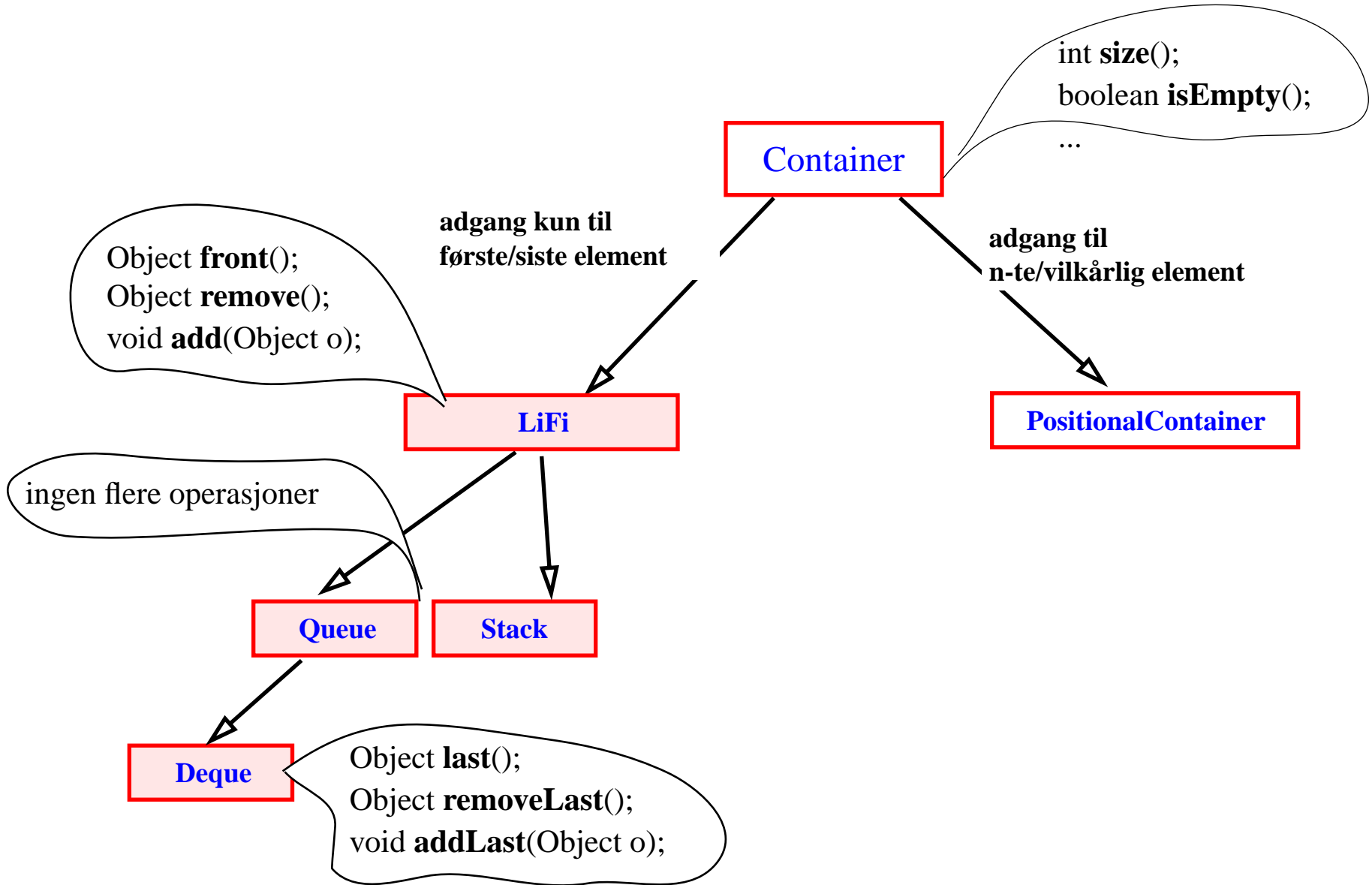
ADT hierarki kunne tenkes slik



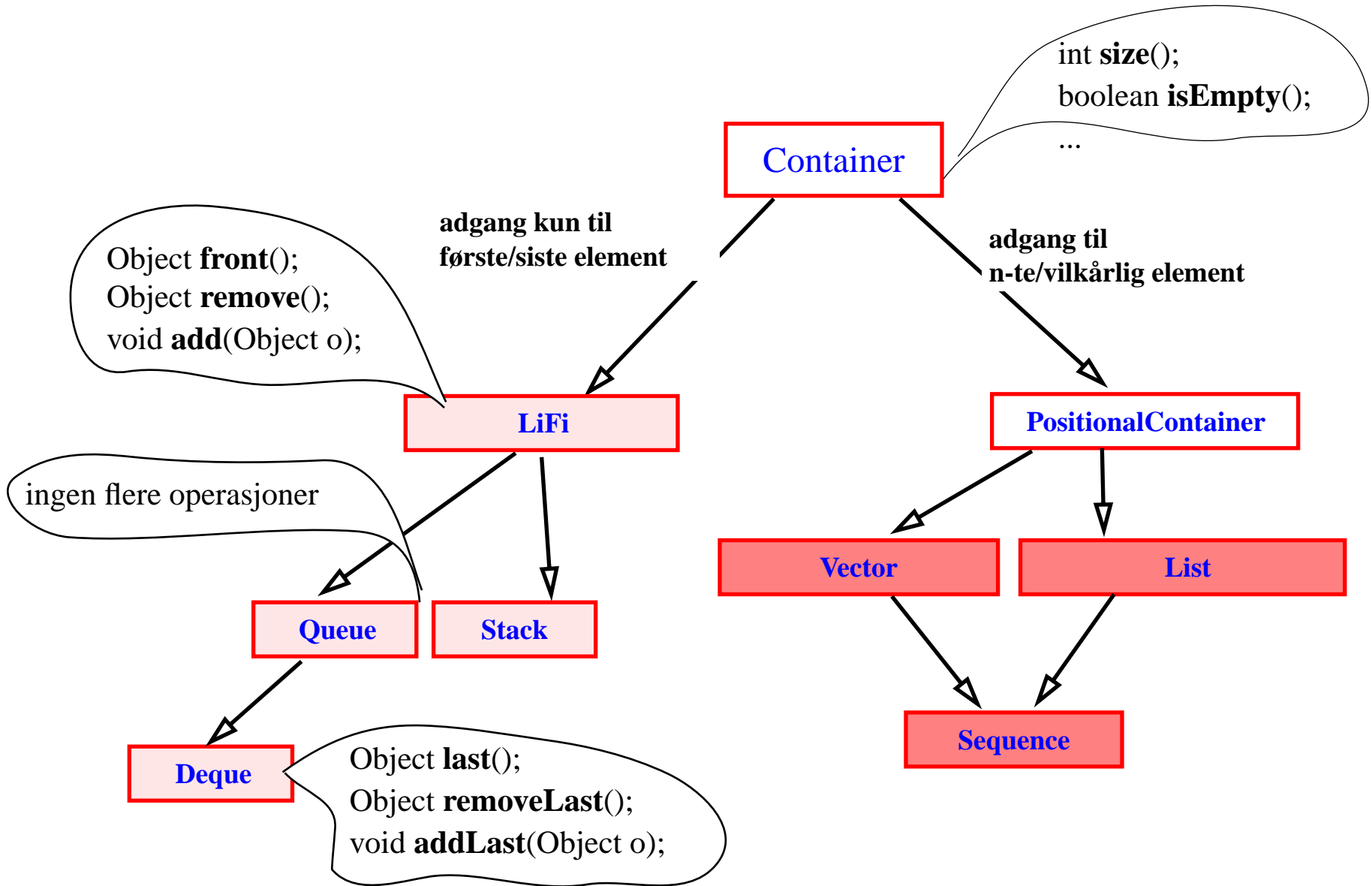
ADT hierarki kunne tenkes slik



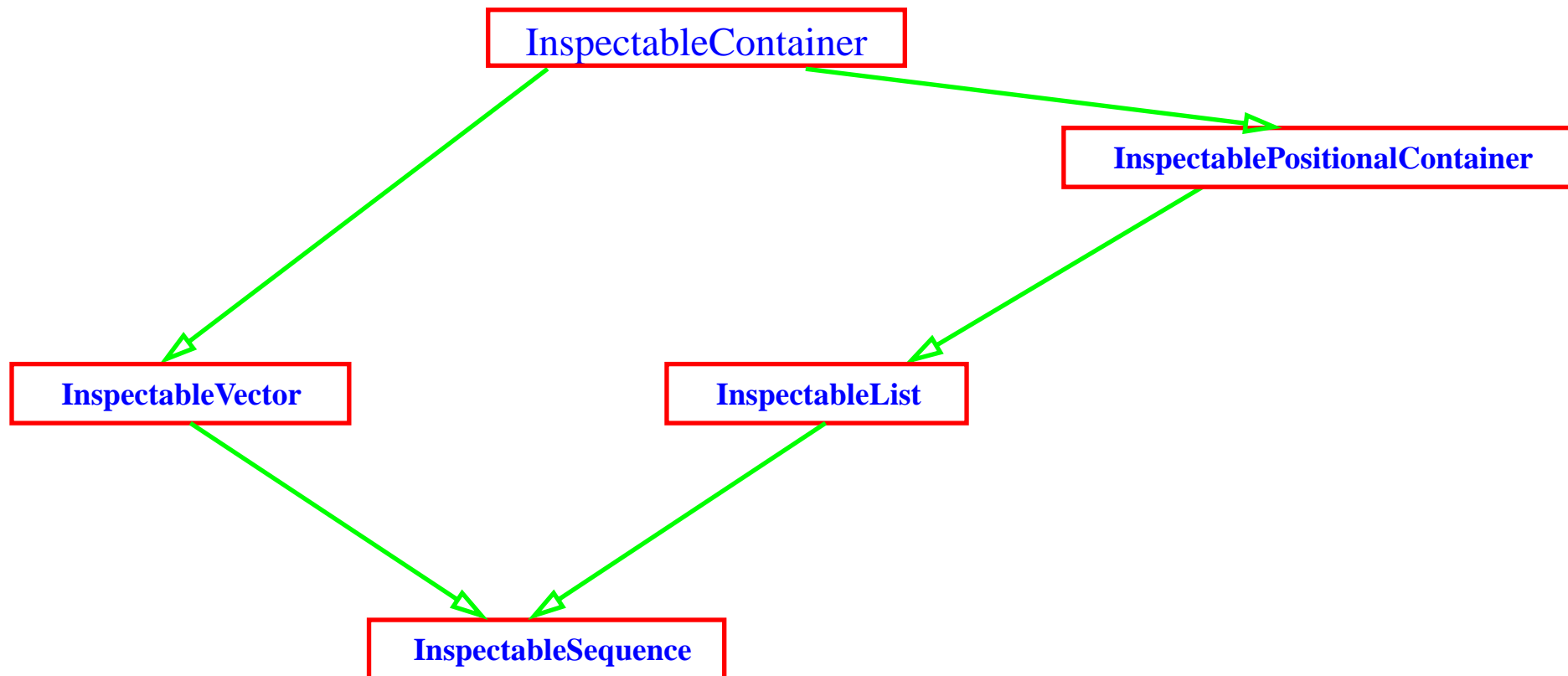
ADT hierarki kunne tenkes slik



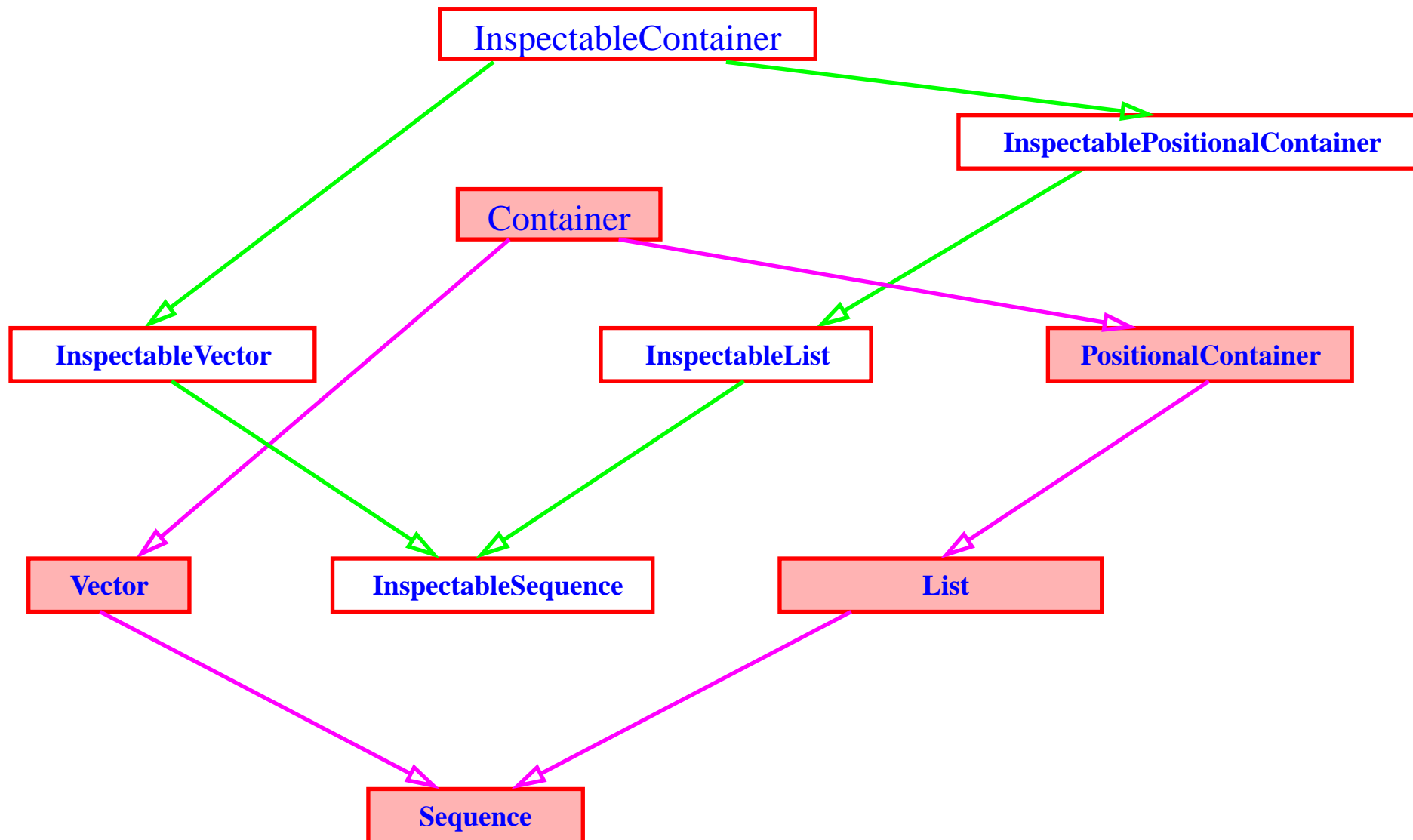
ADT hierarki kunne tenkes slik



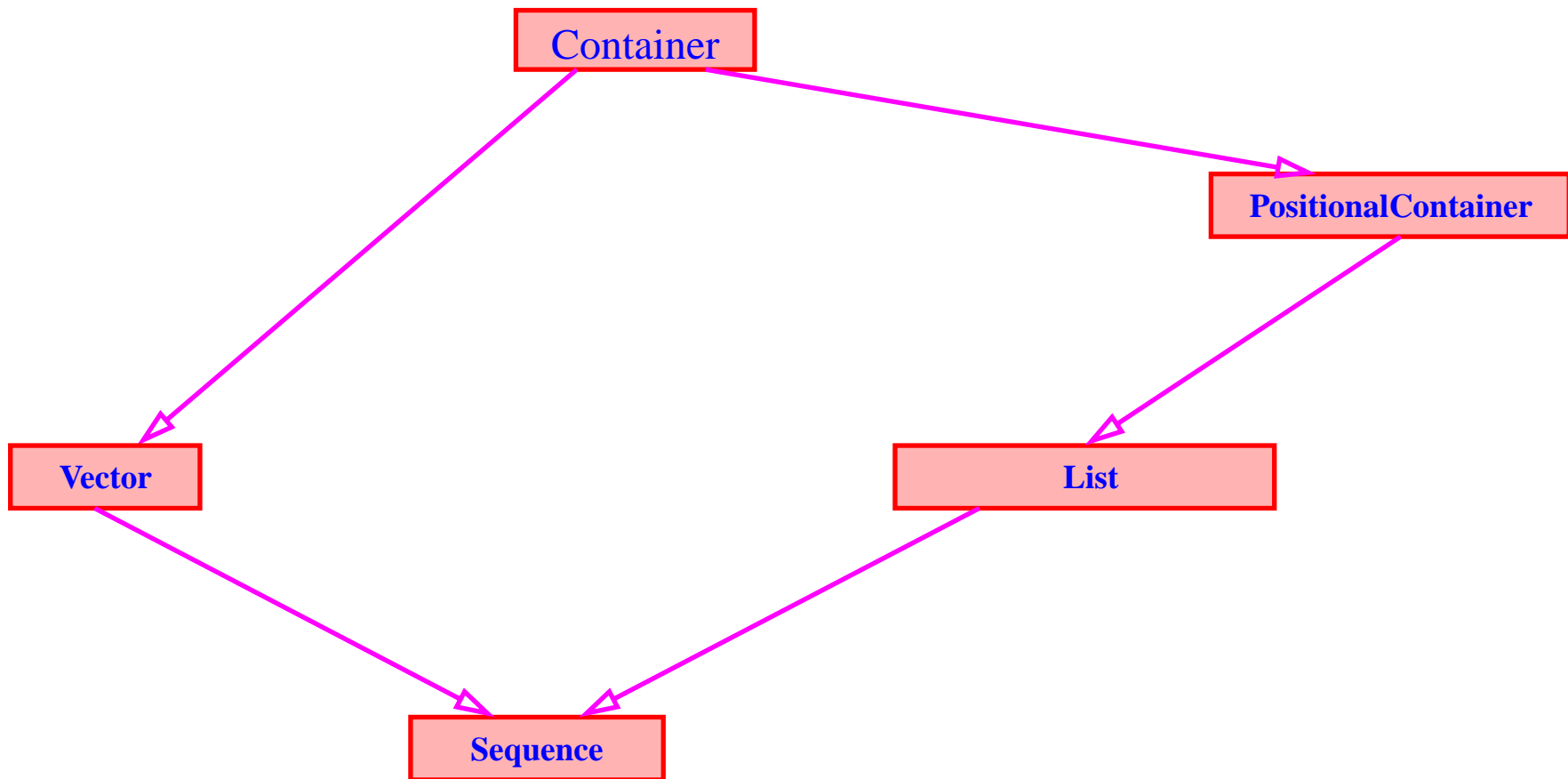
... men i JDSL.core.api er det nærmere slik



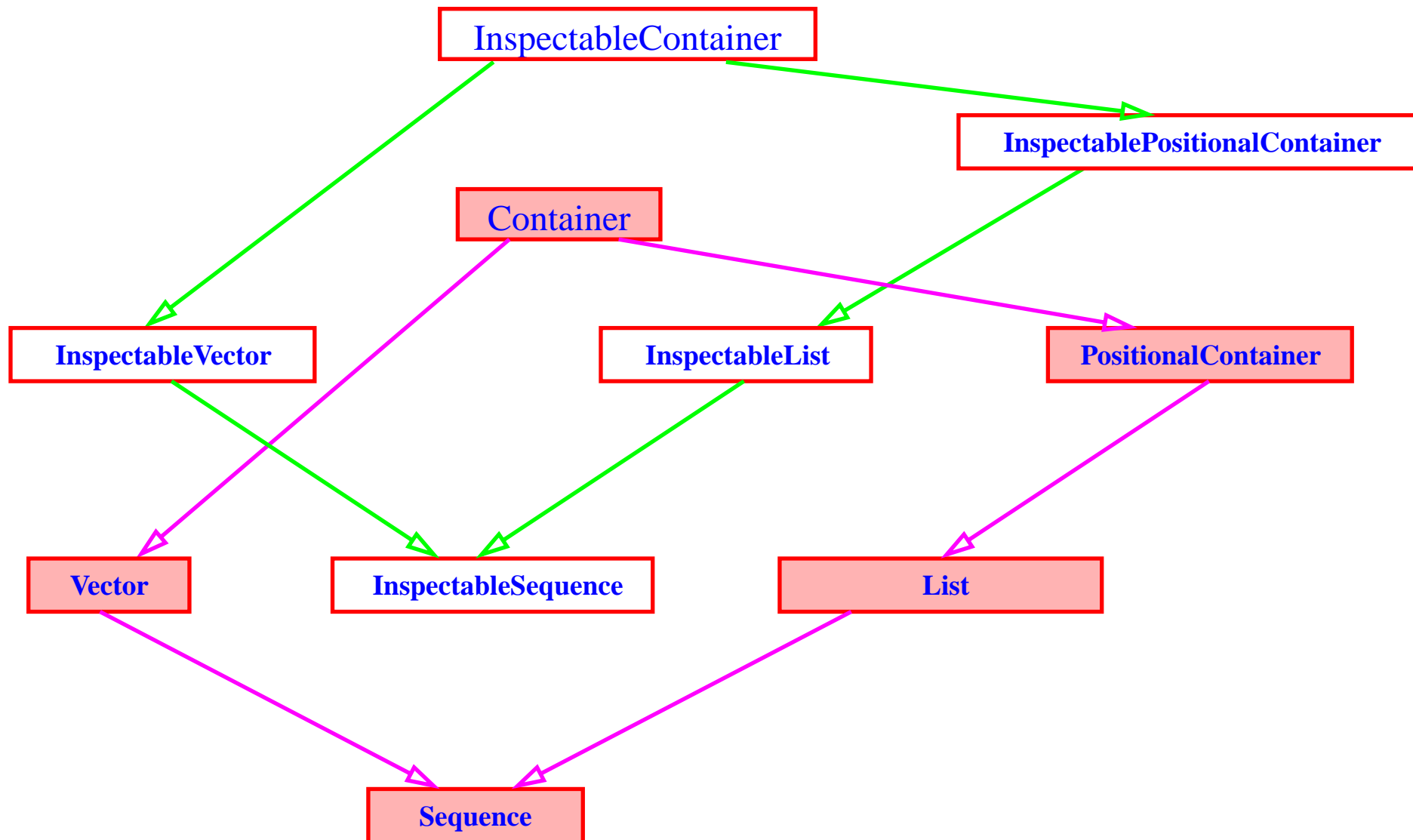
... men i JDSL.core.api er det nærmere slik



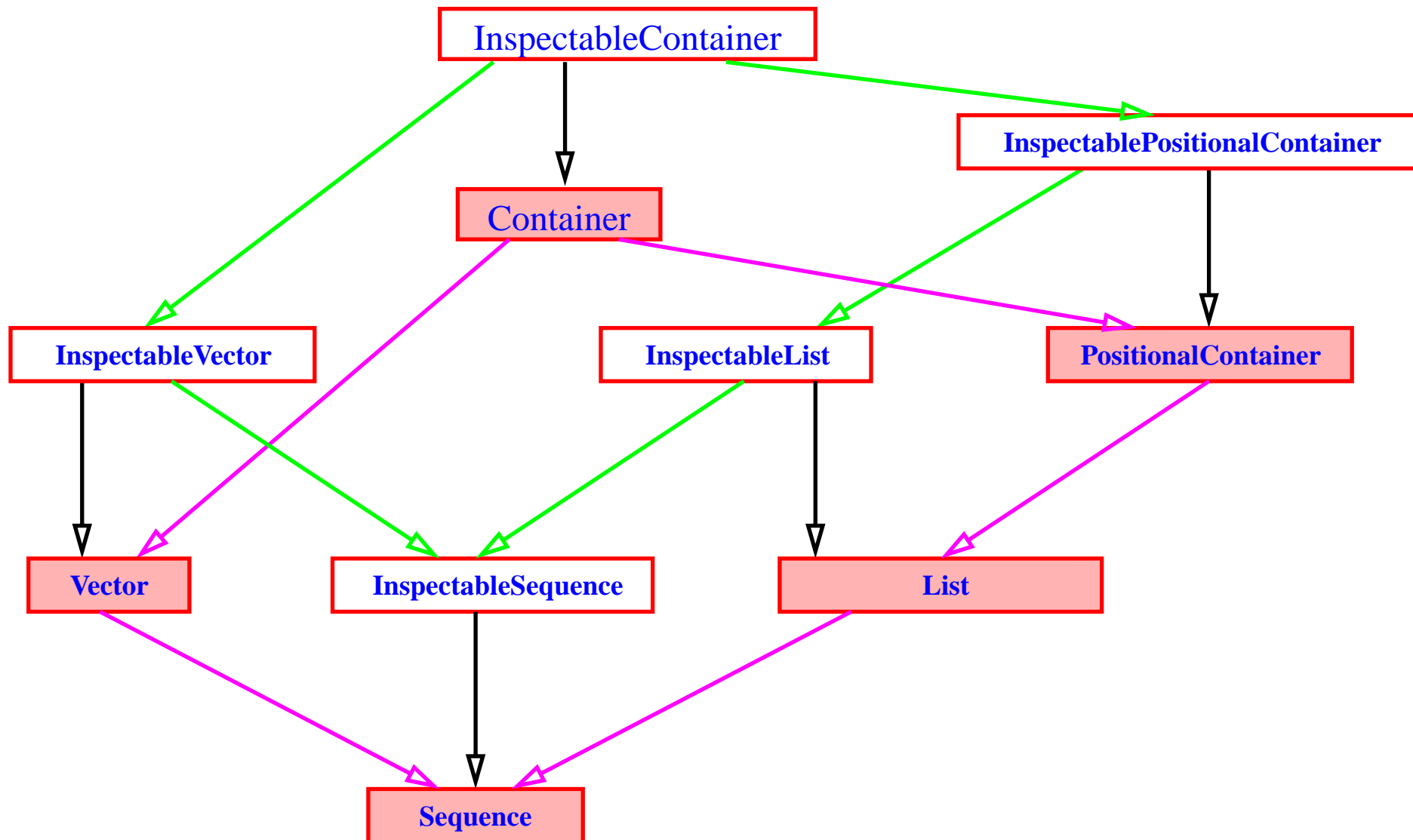
... men i JDSL.core.api er det nærmere slik



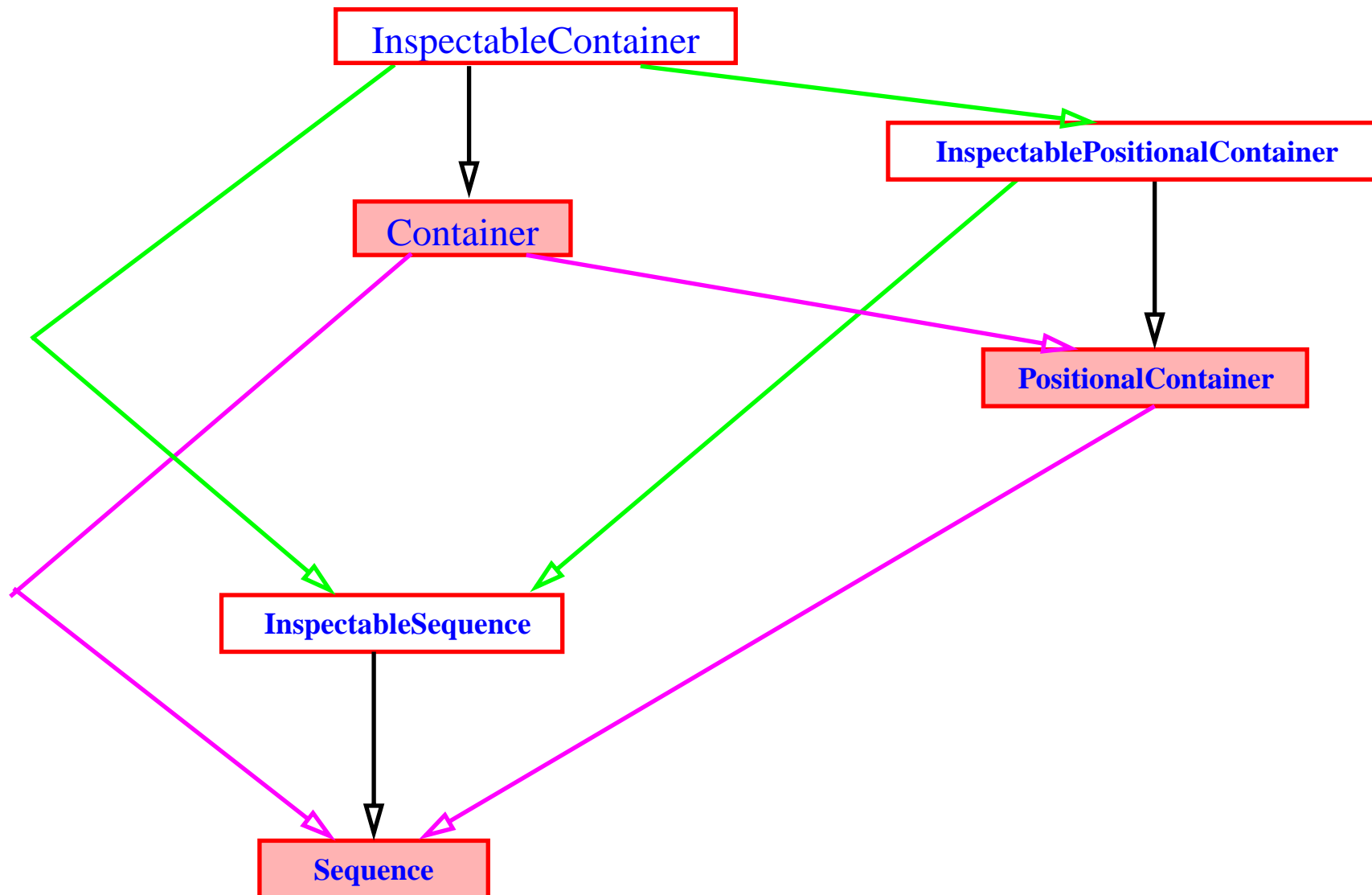
... men i JDSL.core.api er det nærmere slik



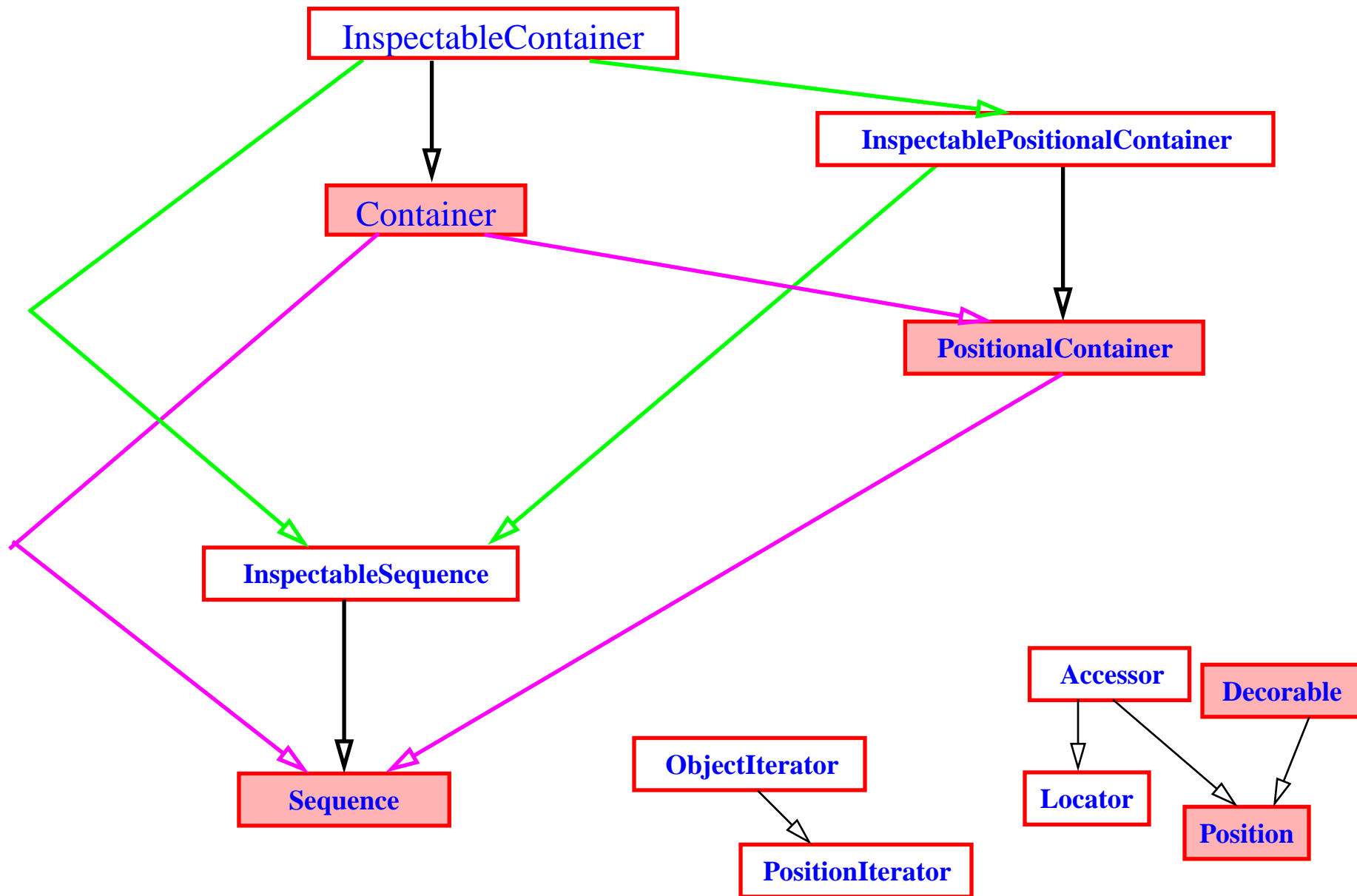
... men i JDSL.core.api er det nærmere slik



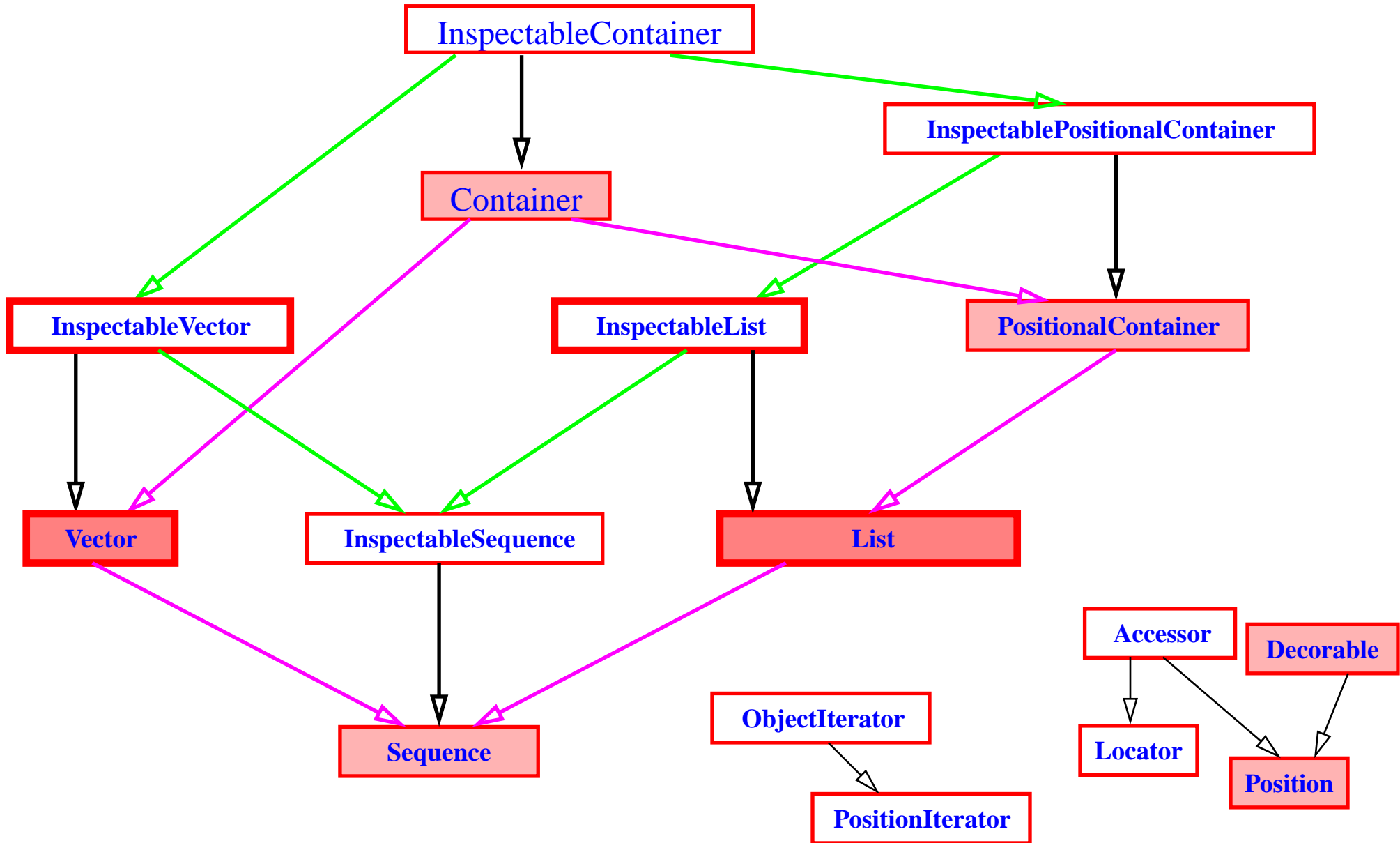
... men i JDSL.core.api er det nærmere slik



... men i JDSL.core.api er det nærmere slik

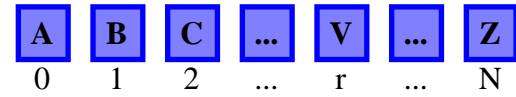


... men i JDSL.core.api er det nærmere slik

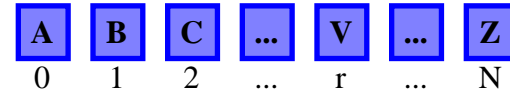


Vektor ADT

- ** adgang til/fjerning/innsetting av elementer*
- * med **vilkårlig** rank (ulik Stack og Queue)*
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */*



Vektor ADT



- /** adgang til/fjerning/innsetting av elementer*
- * med vilkårlig rank (ulik Stack og Queue)*
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */*

```
public interface Vector  
    extends Container {
```

```
    void insertAtRank(int r, Object o);
```

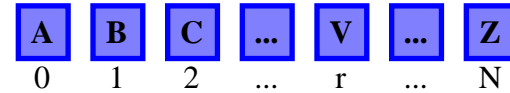
```
    Object elemAtRank(int r);
```

```
    Object replaceAtRank(int r, Object o);
```

```
    Object removeAtRank(int r);
```

```
}
```

Vektor ADT



```
/** adgang til/fjerning/innsetting av elementer  
* med vilkårlig rank (ulik Stack og Queue)  
* Rank – angis med et tall  $0 \leq r < \text{size}()$  */
```

```
public interface Vector  
    extends Container {
```

```
    void insertAtRank(int r, Object o);
```

```
/** returnerer Objektet med rank r  
* @param r et tall:  $0 \leq r < \text{size}()$   
* @return Objektet i posisjon r  
* @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()-1$   
*/ Object elemAtRank(int r);
```

```
Object replaceAtRank(int r, Object o);
```

```
Object removeAtRank(int r);
```

```
}
```

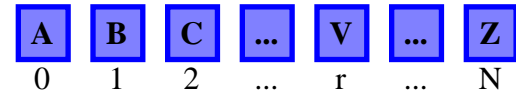
Vektor ADT

- /** adgang til/fjerning/innsetting av elementer*
- * med vilkårlig rank (ulik Stack og Queue)*
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */*

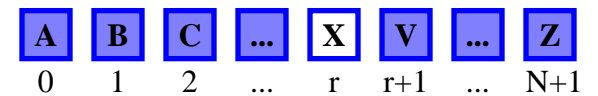
```
public interface Vector  
    extends Container {
```

```
    void insertAtRank(int r, Object o);
```

- /** returnerer Objektet med rank r*
- * @param r et tall: $0 \leq r < \text{size}()$*
- * @return Objektet i posisjon r*
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$*
- */ Object elemAtRank(int r);*



```
Object replaceAtRank(int r, Object Y);
```



```
Object removeAtRank(int r);
```

```
}
```

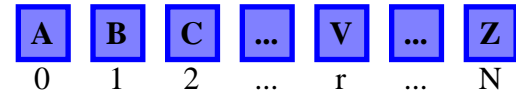
Vektor ADT

- /** adgang til/fjerning/innsetting av elementer
- * med *vilkårlig* rank (ulik Stack og Queue)
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */

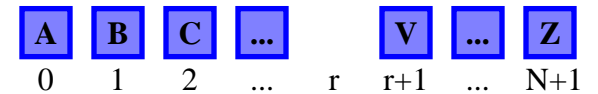
```
public interface Vector  
    extends Container {
```

```
    void insertAtRank(int r, Object o);
```

- /** returnerer Objektet med rank r
- * @param r et tall: $0 \leq r < \text{size}()$
- * @return Objektet i posisjon r
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ Object elemAtRank(int r);



```
Object replaceAtRank(int r, Object Y);
```



```
Object removeAtRank(int r);
```

```
}
```

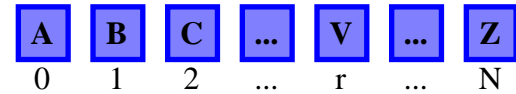
Vektor ADT

*/** adgang til/fjerning/innsetting av elementer*
** med vilkårlig rank (ulik Stack og Queue)*
** Rank – angis med et tall $0 \leq r < \text{size}()$ */*

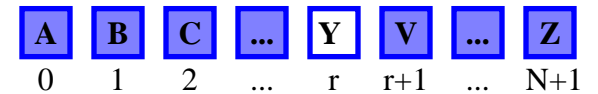
```
public interface Vector  
    extends Container {
```

```
    void insertAtRank(int r, Object o);
```

```
/** returnerer Objektet med rank r  
* @param r et tall:  $0 \leq r < \text{size}()$   
* @return Objektet i posisjon r  
* @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()-1$   
*/ Object elemAtRank(int r);
```



```
Object replaceAtRank(int r, Object Y);
```



```
Object removeAtRank(int r);
```

```
}
```

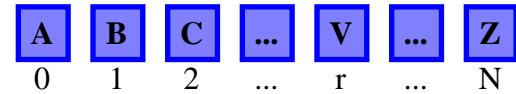

Vektor ADT

- /** adgang til/fjerning/innsetting av elementer
- * med *vilkårlig* rank (ulik Stack og Queue)
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */

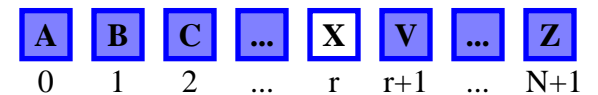
```
public interface Vector  
    extends Container {
```

```
    void insertAtRank(int r, Object o);
```

- /** returnerer Objektet med rank r
- * @param r et tall: $0 \leq r < \text{size}()$
- * @return Objektet i posisjon r
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object elemAtRank(int r);**



- /** erstatter Objektet med rank r med et nytt Objekt
- * @param r et tall: $0 \leq r < \text{size}()$
- * @param o Objektet som skal innsettes
- * @return Objektet som ble erstattet
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object replaceAtRank(int r, Object o);**



```
    Object removeAtRank(int r);
```

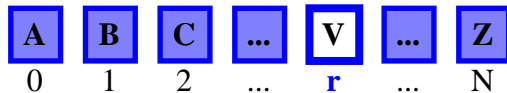
```
}
```

Vektor ADT

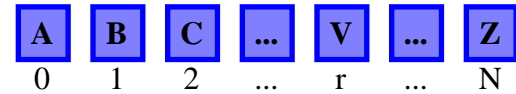
- /** adgang til/fjerning/innsetting av elementer
- * med *vilkårlig* rank (ulik Stack og Queue)
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */

```
public interface Vector  
    extends Container {
```

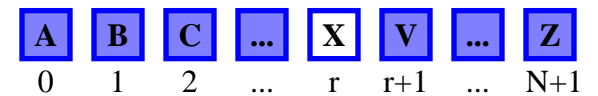
```
void insertAtRank(int r, Object o);
```



- /** returnerer Objektet med rank r
- * @param r et tall: $0 \leq r < \text{size}()$
- * @return Objektet i posisjon r
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object elemAtRank(int r);**



- /** erstatter Objektet med rank r med et nytt Objekt
- * @param r et tall: $0 \leq r < \text{size}()$
- * @param o Objektet som skal innsettes
- * @return Objektet som ble erstattet
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object replaceAtRank(int r, Object o);**



```
Object removeAtRank(int r);
```

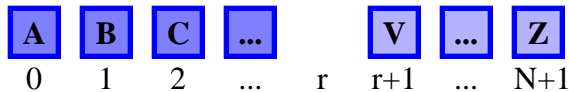
```
}
```

Vektor ADT

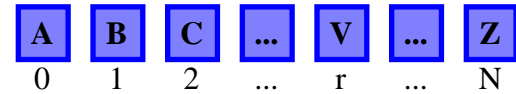
- /** adgang til/fjerning/innsetting av elementer
- * med *vilkårlig* rank (ulik Stack og Queue)
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */

```
public interface Vector  
    extends Container {
```

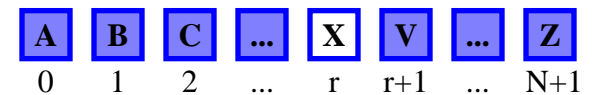
```
void insertAtRank(int r, Object o);
```



- /** returnerer Objektet med rank r
- * @param r et tall: $0 \leq r < \text{size}()$
- * @return Objektet i posisjon r
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object elemAtRank(int r);**



- /** erstatter Objektet med rank r med et nytt Objekt
- * @param r et tall: $0 \leq r < \text{size}()$
- * @param o Objektet som skal innsettes
- * @return Objektet som ble erstattet
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object replaceAtRank(int r, Object o);**



```
Object removeAtRank(int r);
```

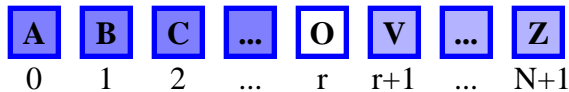
```
}
```

Vektor ADT

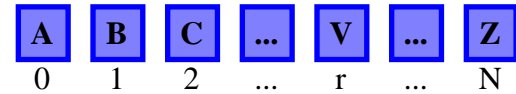
- /** adgang til/fjerning/innsetting av elementer
- * med *vilkårlig* rank (ulik Stack og Queue)
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */

```
public interface Vector  
    extends Container {
```

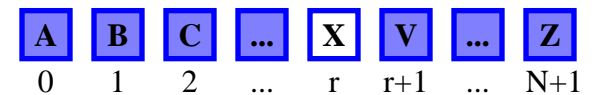
```
void insertAtRank(int r, Object o);
```



- /** returnerer Objektet med rank r
- * @param r et tall: $0 \leq r < \text{size}()$
- * @return Objektet i posisjon r
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object elemAtRank(int r);**



- /** erstatter Objektet med rank r med et nytt Objekt
- * @param r et tall: $0 \leq r < \text{size}()$
- * @param o Objektet som skal innsettes
- * @return Objektet som ble erstattet
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ **Object replaceAtRank(int r, Object o);**



```
Object removeAtRank(int r);
```

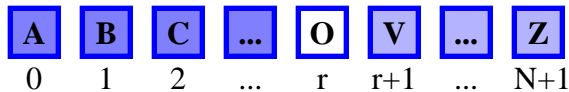
```
}
```

Vektor ADT

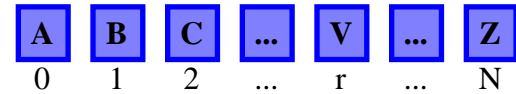
- /** adgang til/fjerning/innsetting av elementer
- * med *vilkårlig* rank (ulik Stack og Queue)
- * Rank – angis med et tall $0 \leq r < \text{size}()$ */

```
public interface Vector  
    extends Container {
```

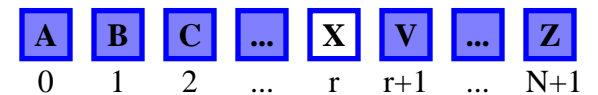
- /** setter inn Objekt o med rank r, andres rank kan endres
- * @param r et tall: $0 \leq r \leq \text{size}()$
- * @param o Objektet som skal innsettes
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()$
- */ void insertAtRank(int r, Object o);



- /** returnerer Objektet med rank r
- * @param r et tall: $0 \leq r < \text{size}()$
- * @return Objektet i posisjon r
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ Object elemAtRank(int r);



- /** erstatter Objektet med rank r med et nytt Objekt
- * @param r et tall: $0 \leq r < \text{size}()$
- * @param o Objektet som skal innsettes
- * @return Objektet som ble erstattet
- * @exception BoundaryViolationEx om $r < 0$ eller $r > \text{size}()-1$
- */ Object replaceAtRank(int r, Object o);



```
Object removeAtRank(int r);
```

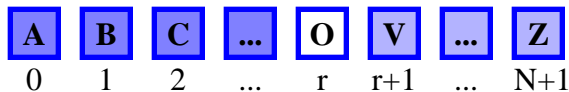
```
}
```

Vektor ADT

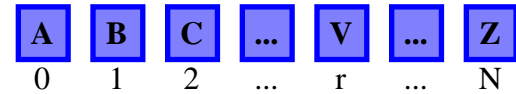
```
/** adgang til/fjerning/innsetting av elementer  
 * med vilkårlig rank (ulik Stack og Queue)  
 * Rank – angis med et tall  $0 \leq r < \text{size}()$  */
```

```
public interface Vector  
    extends Container {
```

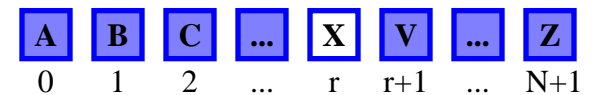
```
/** setter inn Objekt o med rank r, andres rank kan endres  
 * @param r et tall:  $0 \leq r \leq \text{size}()$   
 * @param o Objektet som skal innsettes  
 * @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()$   
 */ void insertAtRank(int r, Object o);
```



```
/** returnerer Objektet med rank r  
 * @param r et tall:  $0 \leq r < \text{size}()$   
 * @return Objektet i posisjon r  
 * @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()-1$   
 */ Object elemAtRank(int r);
```

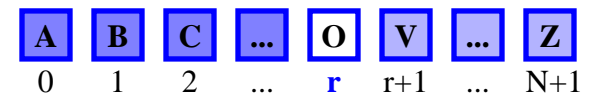


```
/** erstatter Objektet med rank r med et nytt Objekt  
 * @param r et tall:  $0 \leq r < \text{size}()$   
 * @param o Objektet som skal innsettes  
 * @return Objektet som ble erstattet  
 * @exception BoundaryViolationEx  
 * om  $r < 0$  eller  $r > \text{size}()-1$   
 */ Object replaceAtRank(int r, Object o);
```



```
Object removeAtRank(int r);
```

```
}
```

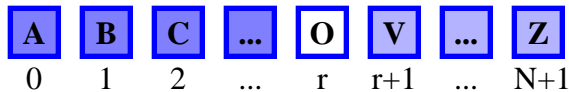


Vektor ADT

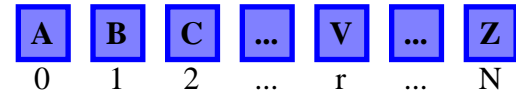
```
/** adgang til/fjerning/innsetting av elementer
 * med vilkårlig rank (ulik Stack og Queue)
 * Rank – angis med et tall  $0 \leq r < \text{size}()$  */
```

```
public interface Vector
    extends Container {
```

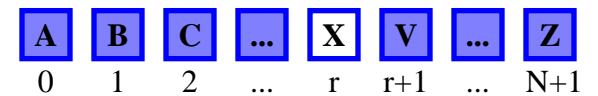
```
/** setter inn Objekt o med rank r, andres rank kan endres
 * @param r et tall:  $0 \leq r \leq \text{size}()$ 
 * @param o Objektet som skal innsettes
 * @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()$ 
 */ void insertAtRank(int r, Object o);
```



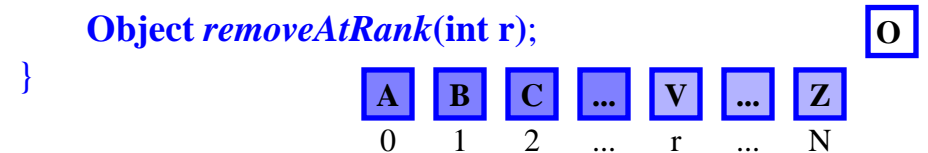
```
/** returnerer Objektet med rank r
 * @param r et tall:  $0 \leq r < \text{size}()$ 
 * @return Objektet i posisjon r
 * @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()-1$ 
 */ Object elemAtRank(int r);
```



```
/** erstatter Objektet med rank r med et nytt Objekt
 * @param r et tall:  $0 \leq r < \text{size}()$ 
 * @param o Objektet som skal innsettes
 * @return Objektet som ble erstattet
 * @exception BoundaryViolationEx
 * om  $r < 0$  eller  $r > \text{size}()-1$ 
 */ Object replaceAtRank(int r, Object o);
```



```
Object removeAtRank(int r);
```

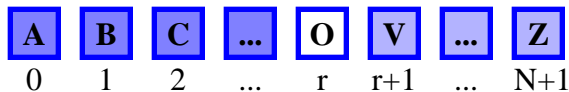


Vektor ADT

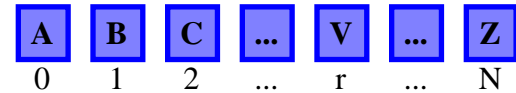
```
/** adgang til/fjerning/innsetting av elementer
 * med vilkårlig rank (ulik Stack og Queue)
 * Rank – angis med et tall  $0 \leq r < \text{size}()$  */
```

```
public interface Vector
    extends Container {
```

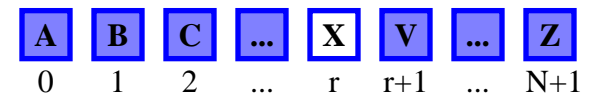
```
/** setter inn Objekt o med rank r, andres rank kan endres
 * @param r et tall:  $0 \leq r \leq \text{size}()$ 
 * @param o Objektet som skal innsettes
 * @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()$ 
 */ void insertAtRank(int r, Object o);
```



```
/** returnerer Objektet med rank r
 * @param r et tall:  $0 \leq r < \text{size}()$ 
 * @return Objektet i posisjon r
 * @exception BoundaryViolationEx om  $r < 0$  eller  $r > \text{size}()-1$ 
 */ Object elemAtRank(int r);
```



```
/** erstatter Objektet med rank r med et nytt Objekt
 * @param r et tall:  $0 \leq r < \text{size}()$ 
 * @param o Objektet som skal innsettes
 * @return Objektet som ble erstattet
 * @exception BoundaryViolationEx
 * om  $r < 0$  eller  $r > \text{size}()-1$ 
 */ Object replaceAtRank(int r, Object o);
```



```
/** fjerner og returnerer Objektet med rank r
 * @param r et tall:  $0 \leq r < \text{size}()$ 
 * @return Objektet som ble fjernet
 * @exception BoundaryViolationEx
 * hvis  $r < 0$  eller  $r > \text{size}()-1$ 
 */ Object removeAtRank(int r);
```



En ADT kan implementere en annen ADT

```
/** gir aksess til FØRSTE og SISTE elementet */  
public interface Dequeue {  
    /** @return første/siste objektet */  
    Object first();  
    Object last();  
  
    /** sett inn i første/siste posisjon */  
    void insertFirst(Object o);  
    void insertLast(Object o);  
  
    /** @return første/siste objektet - som fjernes! */  
    Object removeFirst();  
    Object removeLast();  
  
    int size();  
    boolean isEmpty();  
}
```

En ADT kan implementere en annen ADT

```
/** gir aksess til FØRSTE og SISTE elementet */  
public interface Dequeue {  
    /** @return første/siste objektet */  
    Object first();  
    Object last();  
  
    /** sett inn i første/siste posisjon */  
    void insertFirst(Object o);  
    void insertLast(Object o);  
  
    /** @return første/siste objektet - som fjernes! */  
    Object removeFirst();  
    Object removeLast();  
  
    int size();  
    boolean isEmpty();  
}
```

```
public class VecDeq implements Dequeue {  
    private Vector v;  
    public VecDeq() { v = new VectorImplementasjon(); }
```

En ADT kan implementere en annen ADT

```
/** gir aksess til FØRSTE og SISTE elementet */  
public interface Dequeue {  
    /** @return første/siste objektet */  
    Object first();  
    Object last();  
  
    /** sett inn i første/siste posisjon */  
    void insertFirst(Object o);  
    void insertLast(Object o);  
  
    /** @return første/siste objektet - som fjernes! */  
    Object removeFirst();  
    Object removeLast();  
  
    int size();  
    boolean isEmpty();  
}
```

```
public class VecDeq implements Dequeue {  
    private Vector v;  
    public VecDeq() { v = new VectorImplementasjon(); }  
  
    Object first() { return v. elemAtRank(0); }  
    Object last() { return v. elemAtRank(v.size()); }  
  
    void insertFirst(Object o) { v. insertAtRank(0, o); }  
    void insertLast(Object o) { v. insertAtRank(v. size(), o); }  
  
    Object removeFirst();  
        { return v. removeAtRank(0); }  
    Object removeLast();  
        { return v. removeAtRank(v. size()); }  
  
    int size() { ... }  
    boolean isEmpty() { ... }  
}
```

Implementasjon av Vector vha. java.util.Vector

med java.util.Vector

```
package java.util;  
  
public class Vector {  
    ...  
    void ensureCapacity(  
        int minCapacity) { ... }  
    void insertElementAt(  
        Object o, int i) { ... }  
    void setElementAt(  
        Object o, int i) { ... }  
    void remove(int i) { ... }  
    Object elementAt(int i) { ... }  
    Object lastElement() { ... }  
    Object firstElement() { ... }  
    int capacity() { ... }  
    int size() { ... }  
}
```

Implementasjon av Vector vha. java.util.Vector

med *java.util.Vector*

```
package java.util;

public class Vector {
    ...
    void ensureCapacity(
        int minCapacity) { ... }
    void insertElementAt(
        Object o, int i) { ... }
    void setElementAt(
        Object o, int i) { ... }
    void remove(int i) { ... }
    Object elementAt(int i) { ... }
    Object lastElement() { ... }
    Object firstElement() { ... }
    int capacity() { ... }
    int size() { ... }
}
```

```
import java.util.Vector;
public class RSvector implements Vector {
    private Vector v;
    public RSvector() { v= new Vector(); }

    public void insertAtRank(int r, Object o)
        throws BoundaryViolationException {
        try{ v.insertElementAt(o, r); } // sjekker rank r
        catch(ArrayIndexOutOfBoundsException e) {
            throw new BoundaryViolationException ; } }

    public void elemAtRank(int r) { // sjekk rank r
        return v.elementAt(r); }

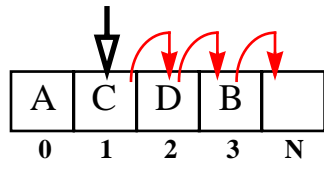
    public Object removeAtRank(int r) { //sjekk rank r
        return v.remove(r); }

    public Object replaceAtRank(int r, Object o) { //sjekk rank r
        return v.set(r,O); }

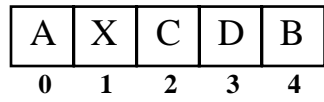
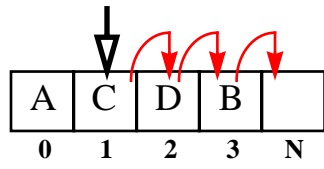
    public int size() { return v.size(); }
    public boolean isEmpty() { return v.size() == 0; }
}
```

A	C	D	B	
0	1	2	3	N

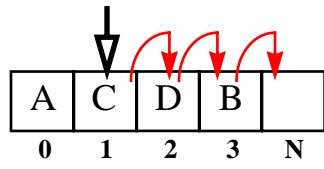
insertAtRank(1,X)



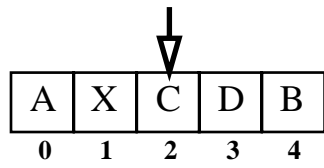
insertAtRank(1,X)



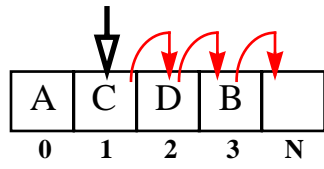
insertAtRank(1,X)



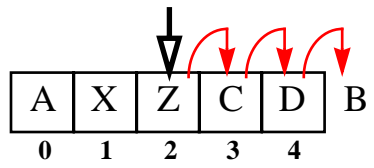
insertAtRank(2,Z)



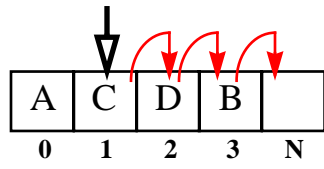
insertAtRank(1,X)



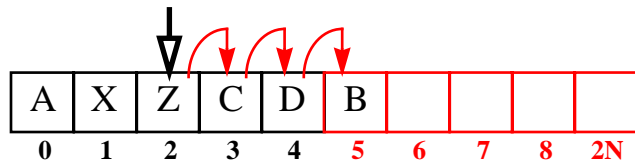
insertAtRank(2,Z)



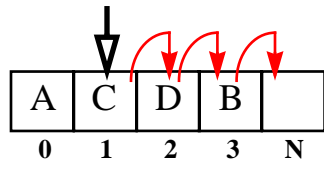
insertAtRank(1,X)



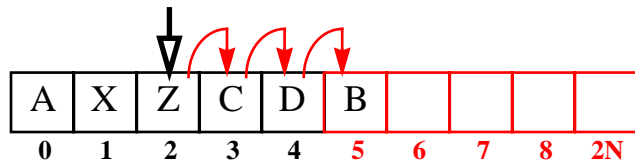
insertAtRank(2,Z)



insertAtRank(1,X)

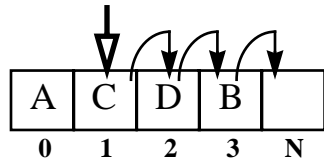


insertAtRank(2,Z)

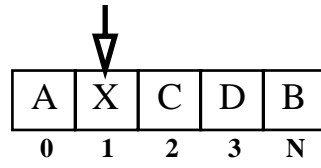


$O(n)$

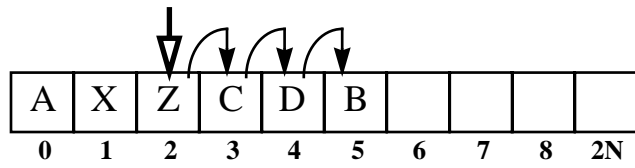
insertAtRank(1,X)



removeAtRank(1)

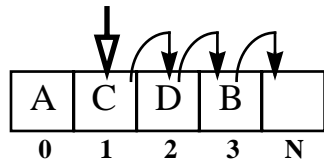


insertAtRank(2,Z)

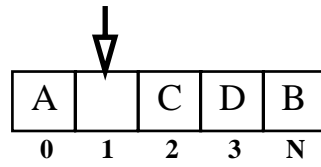


$O(n)$

insertAtRank(1,X)

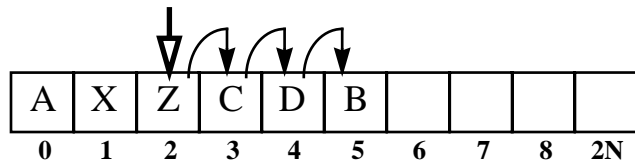


removeAtRank(1)



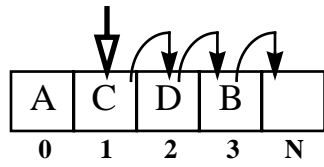
X

insertAtRank(2,Z)

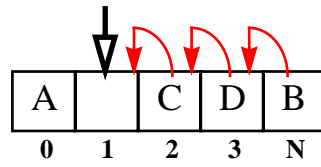


$O(n)$

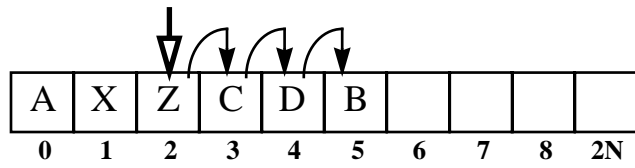
insertAtRank(1,X)



removeAtRank(1)

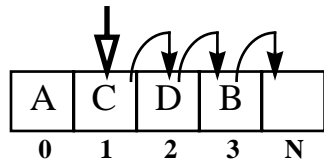


insertAtRank(2,Z)

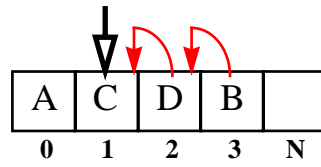


$O(n)$

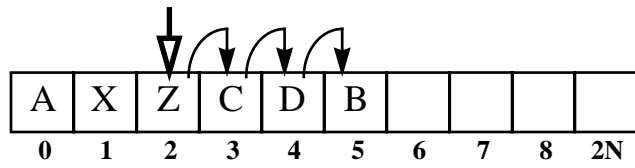
insertAtRank(1,X)



removeAtRank(1)



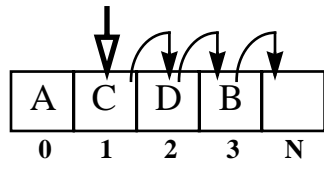
insertAtRank(2,Z)



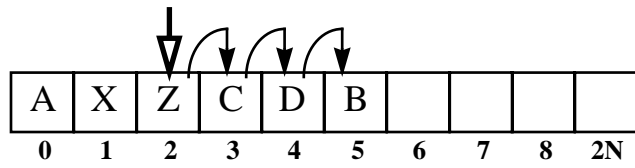
$O(n)$

$O(n)$

insertAtRank(1,X)

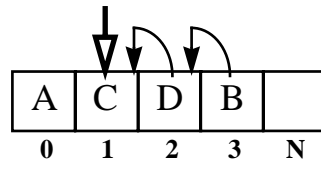


insertAtRank(2,Z)



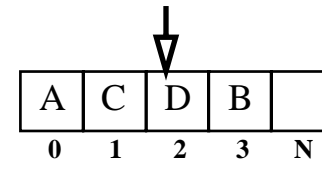
$O(n)$

removeAtRank(1)



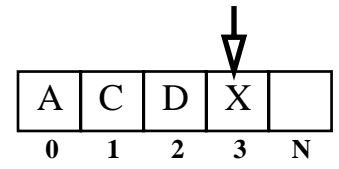
$O(n)$

elemAtRank(2)



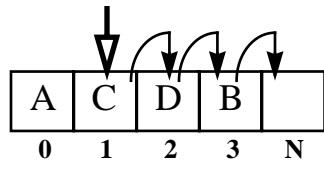
$O(1)$

replaceAtRank(3,X)

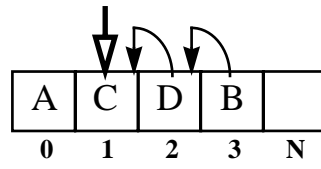


$O(1)$

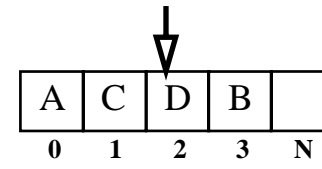
insertAtRank(1,X)



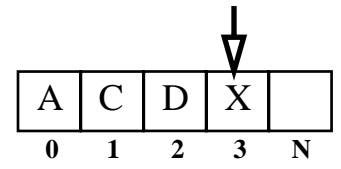
removeAtRank(1)



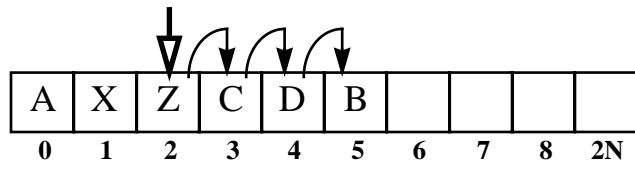
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



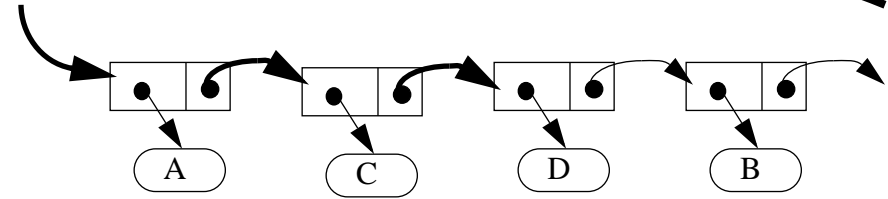
$O(1)$

$O(1)$

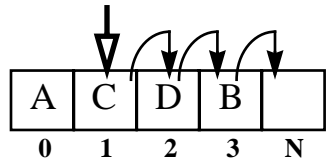
$O(n)$

$O(n)$

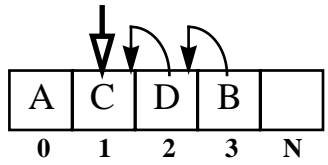
elemAtRank(2)



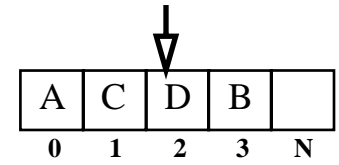
insertAtRank(1,X)



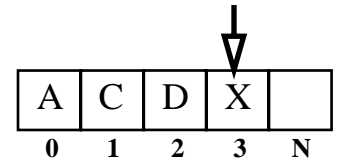
removeAtRank(1)



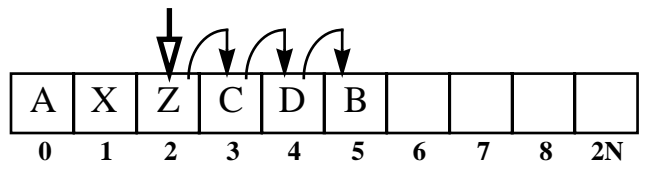
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



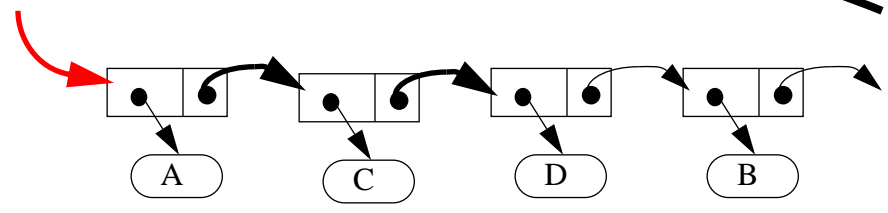
O(n)

O(n)

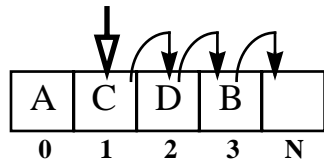
O(1)

O(1)

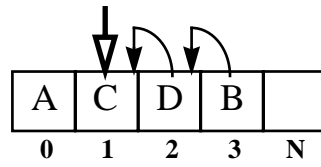
elemAtRank(2)



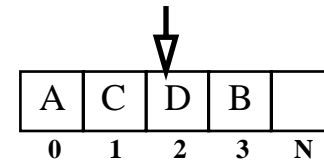
insertAtRank(1,X)



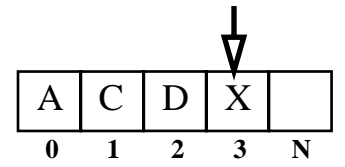
removeAtRank(1)



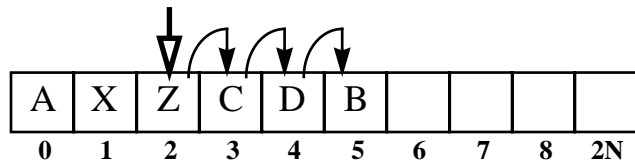
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



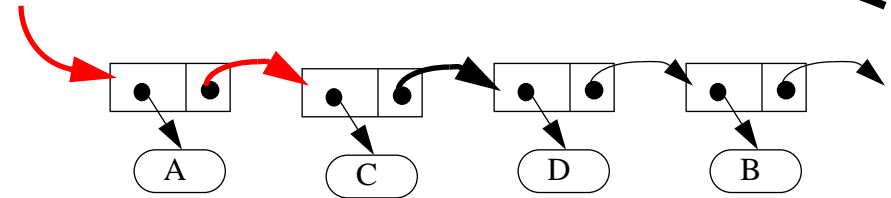
$O(n)$

$O(n)$

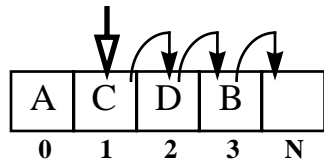
$O(1)$

$O(1)$

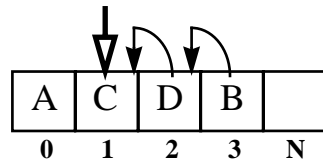
elemAtRank(2)



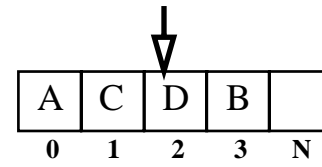
insertAtRank(1,X)



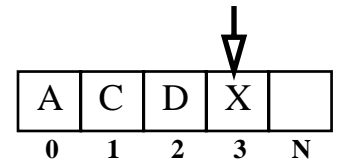
removeAtRank(1)



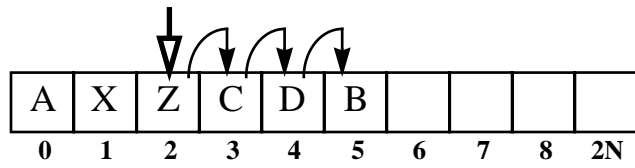
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



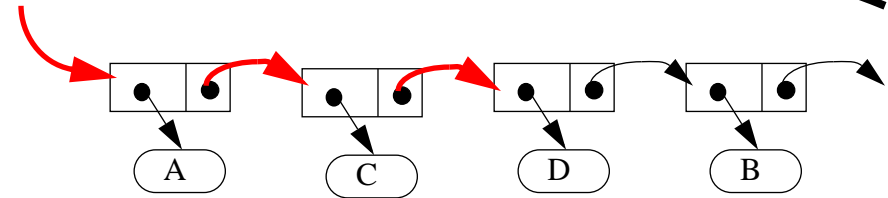
$O(n)$

$O(n)$

$O(1)$

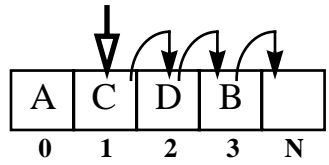
$O(1)$

elemAtRank(2)

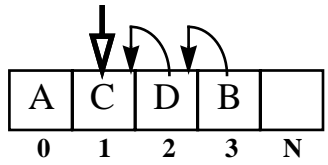


$O(n)$

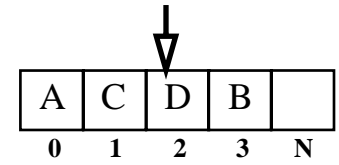
insertAtRank(1,X)



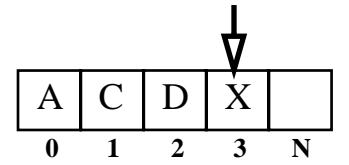
removeAtRank(1)



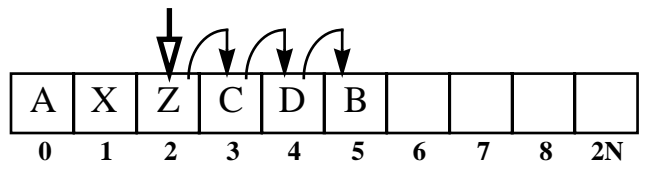
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



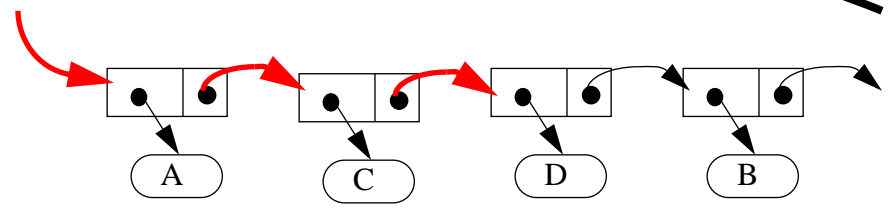
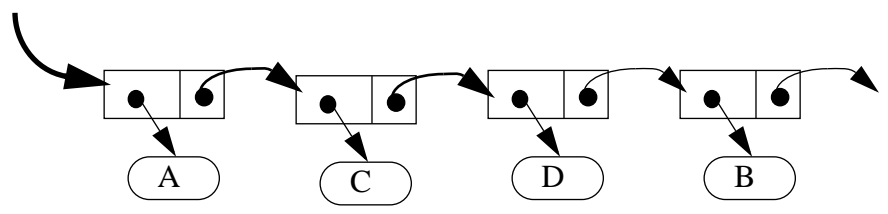
O(n)

O(n)

O(1)

O(1)

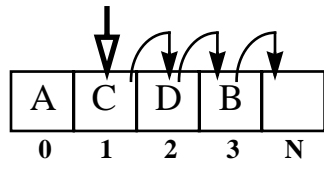
elemAtRank(2)



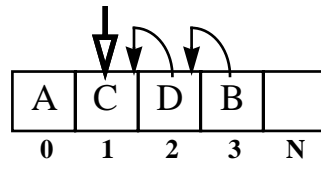
insertAtRank(3,Z)

O(n)

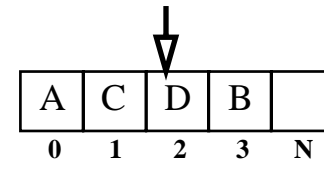
insertAtRank(1,X)



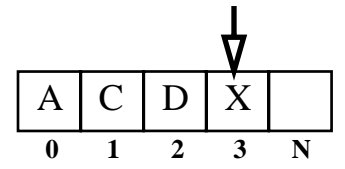
removeAtRank(1)



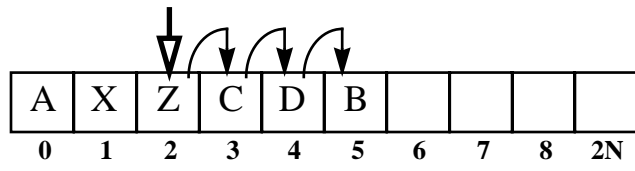
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



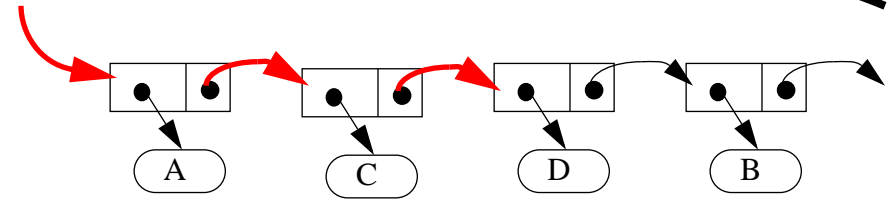
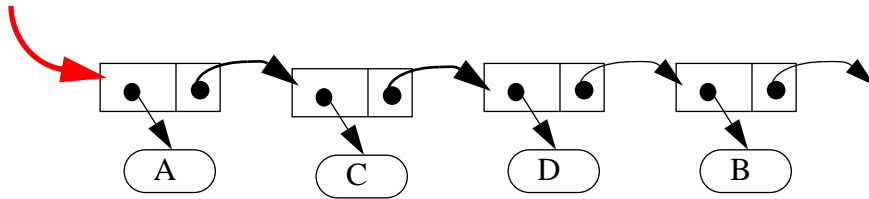
$O(n)$

$O(n)$

$O(1)$

$O(1)$

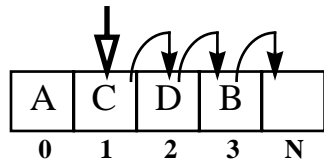
elemAtRank(2)



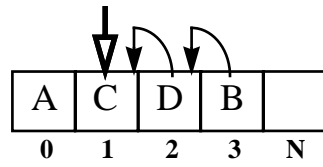
insertAtRank(3,Z)

$O(n)$

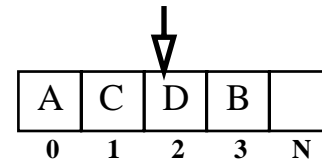
insertAtRank(1,X)



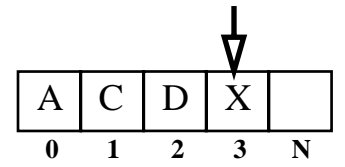
removeAtRank(1)



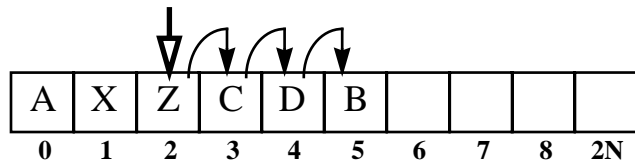
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



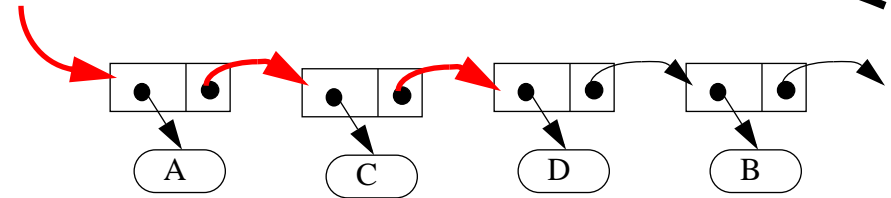
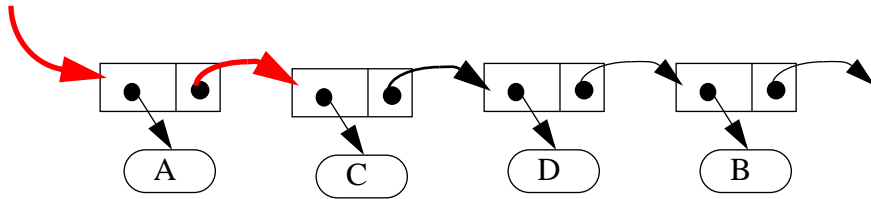
$O(n)$

$O(n)$

$O(1)$

$O(1)$

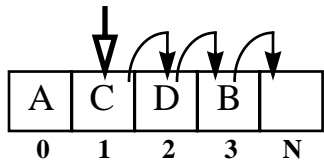
elemAtRank(2)



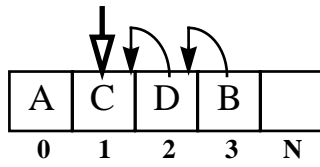
insertAtRank(3,Z)

$O(n)$

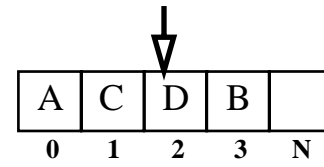
insertAtRank(1,X)



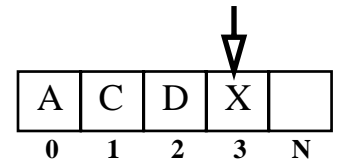
removeAtRank(1)



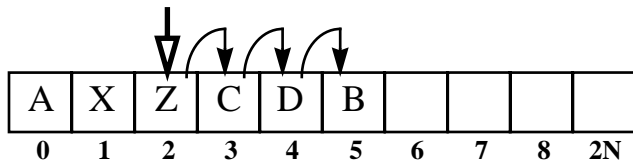
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



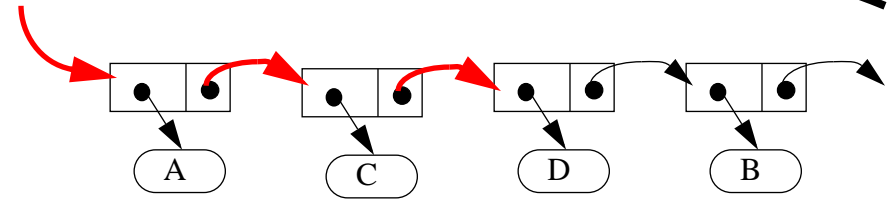
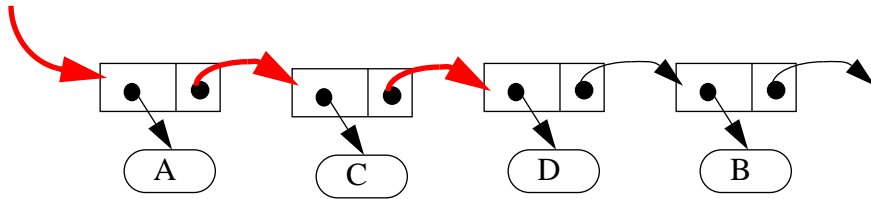
$O(n)$

$O(n)$

$O(1)$

$O(1)$

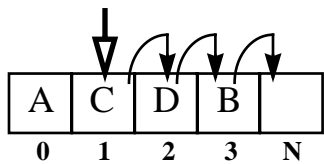
elemAtRank(2)



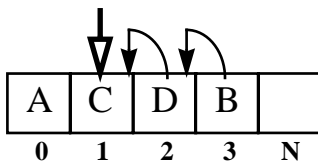
insertAtRank(3,Z)

$O(n)$

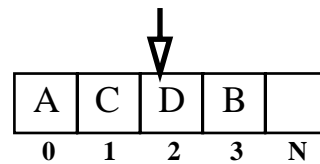
insertAtRank(1,X)



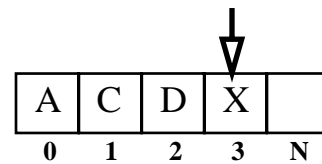
removeAtRank(1)



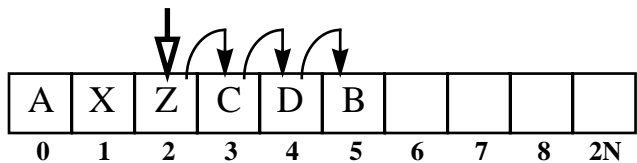
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



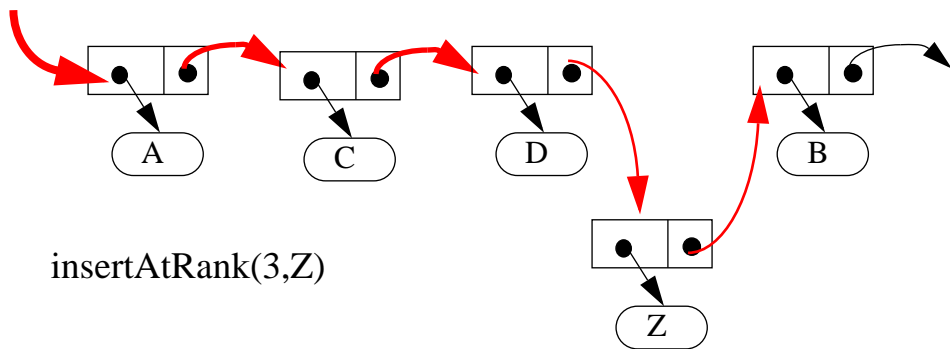
$O(n)$

$O(n)$

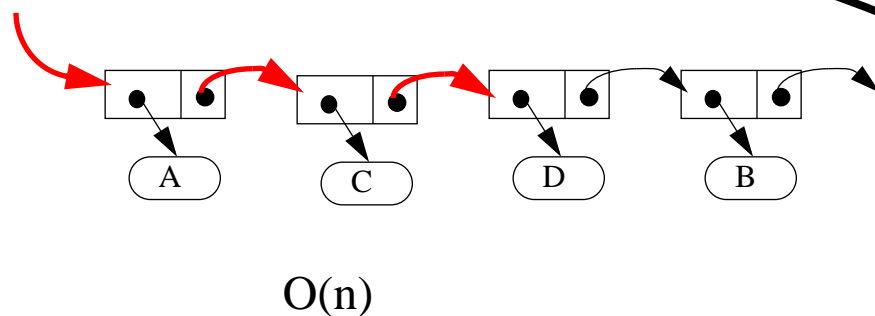
$O(1)$

$O(1)$

elemAtRank(2)

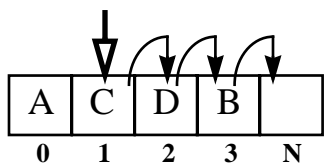


$O(n)$

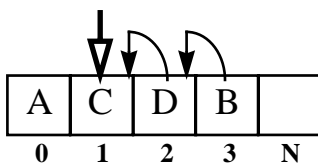


$O(n)$

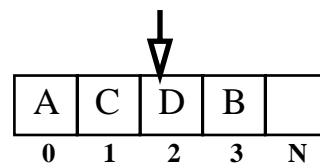
insertAtRank(1,X)



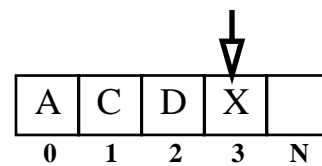
removeAtRank(1)



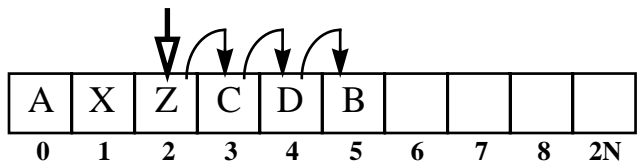
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



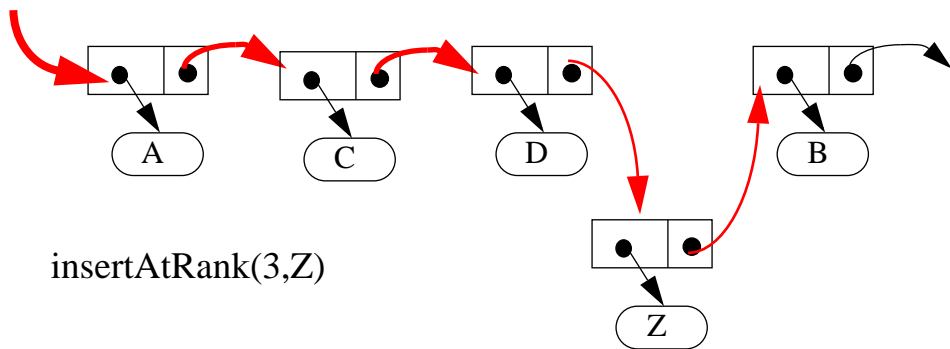
$O(n)$

$O(n)$

$O(1)$

$O(1)$

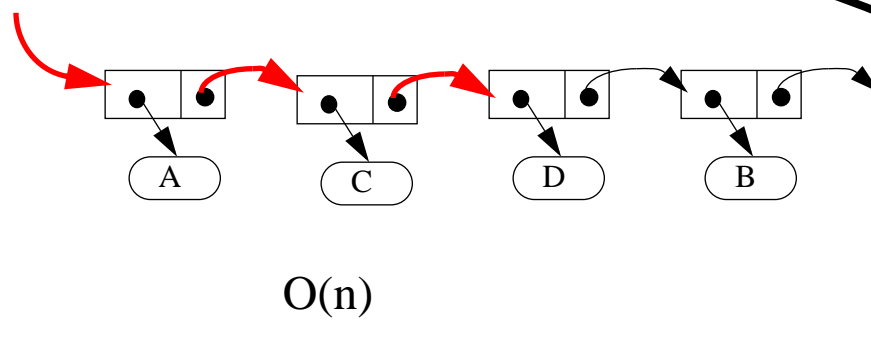
elemAtRank(2)



insertAtRank(3,Z)

$O(n)$

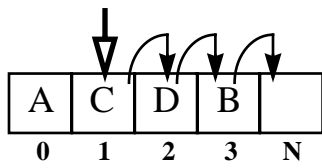
$O(n)$



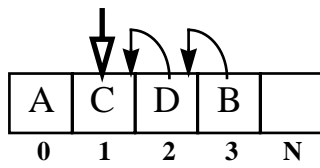
$O(n)$

$O(n)$

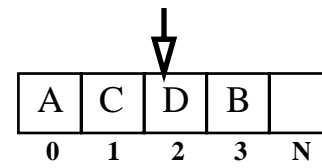
insertAtRank(1,X)



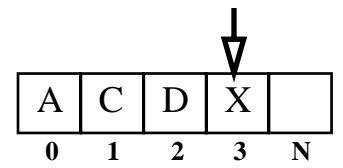
removeAtRank(1)



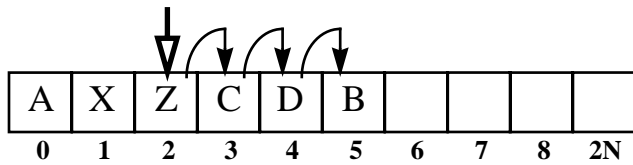
elemAtRank(2)



replaceAtRank(3,X)



insertAtRank(2,Z)



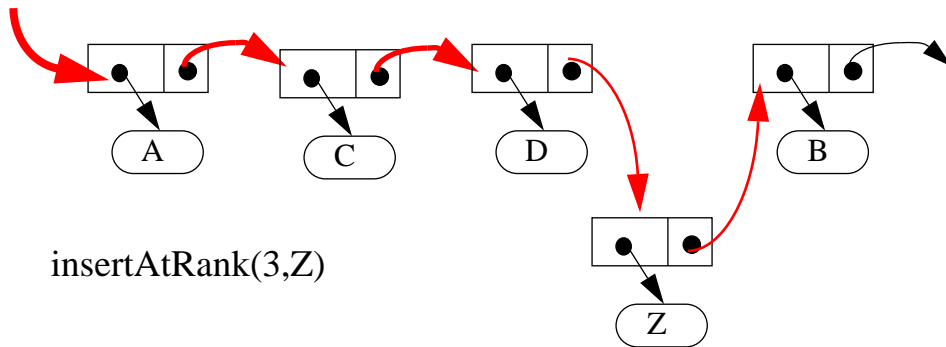
$O(n)$

$O(n)$

$O(1)$

$O(1)$

elemAtRank(2)



insertAtRank(3,Z)

$O(n)$

$O(n)$

$O(n)$

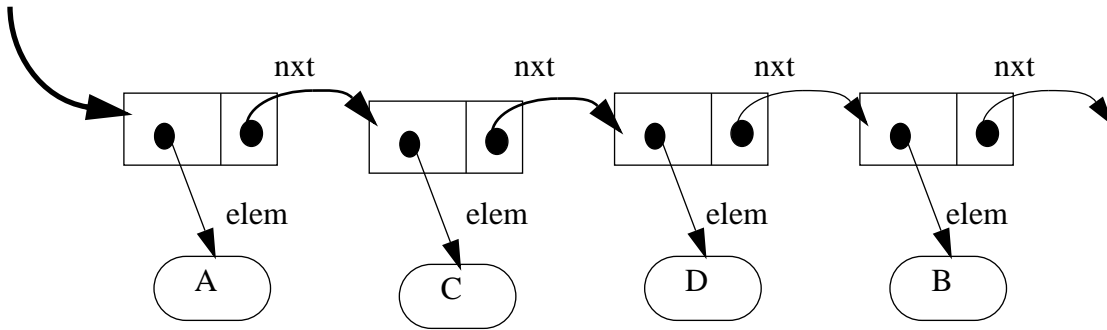
To-veis liste vil ha samme problemet med å finne elemAtRank – $O(n)$

$O(n)$

Implementasjoner av Vektor ADT

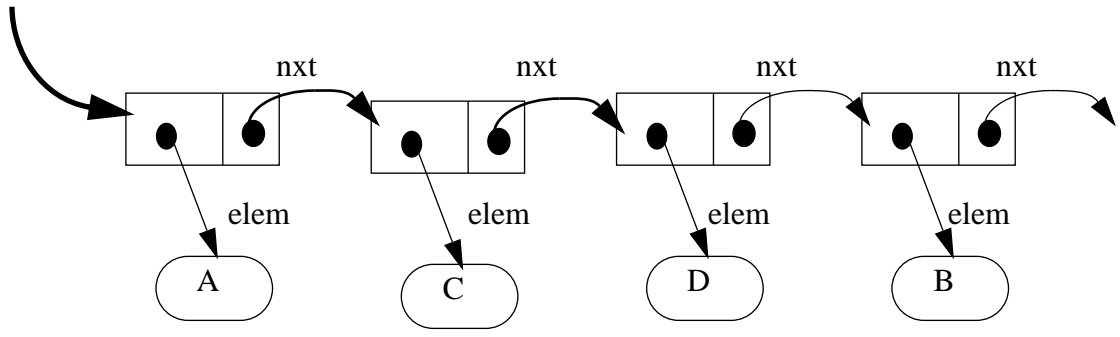
kompleksitet operasjon	Vector <i>relativt!!!</i>	Array <i>(extendable)</i>	en-veis liste	to-veis liste
Container :				
size	<i>1 Vector-op</i>	1	1	1
isEmpty	<i>1 Vector-op</i>	1	1	1
Vector :				
elemAtRank	<i>1 Vector-op</i>	1	n	n
insertAtRank	<i>1 Vector-op</i>	n	n	n
removeAtRank	<i>1 Vector-op</i>	n	n	n
replaceAtRank	<i>1 Vector-op</i>	1	n	n

Position ADT

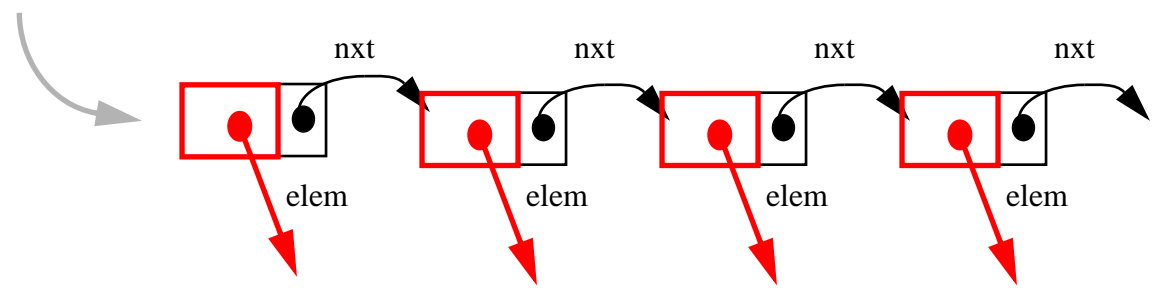


```
class Node {  
    Node nxt;  
    Object elem;  
    ...  
}
```

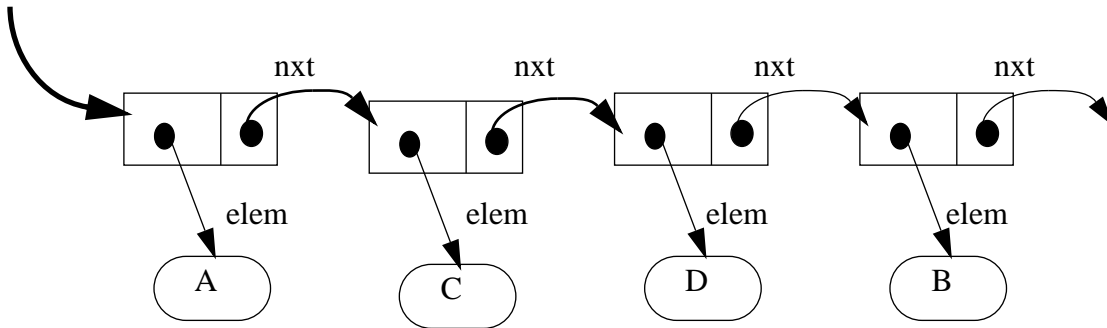
Position ADT



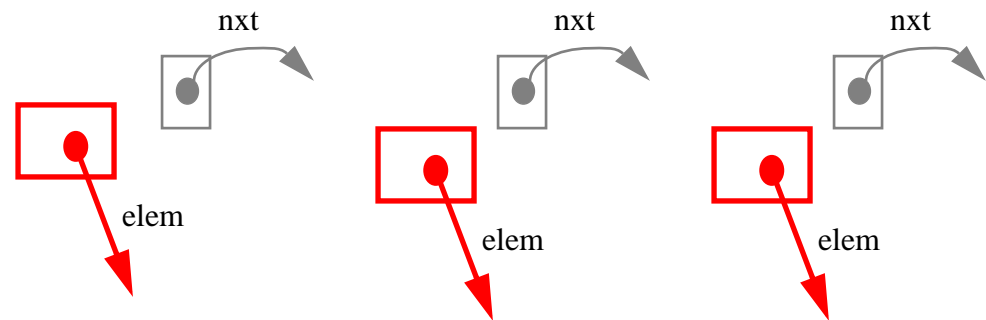
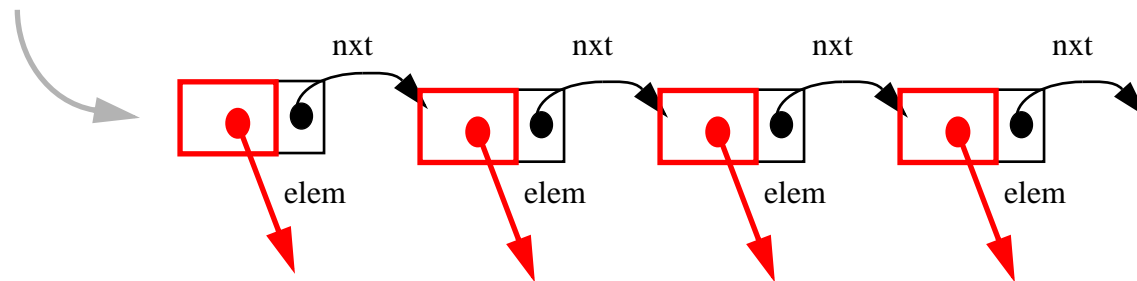
```
class Node {  
    Node nxt;  
    Object elem;  
    ...  
}
```



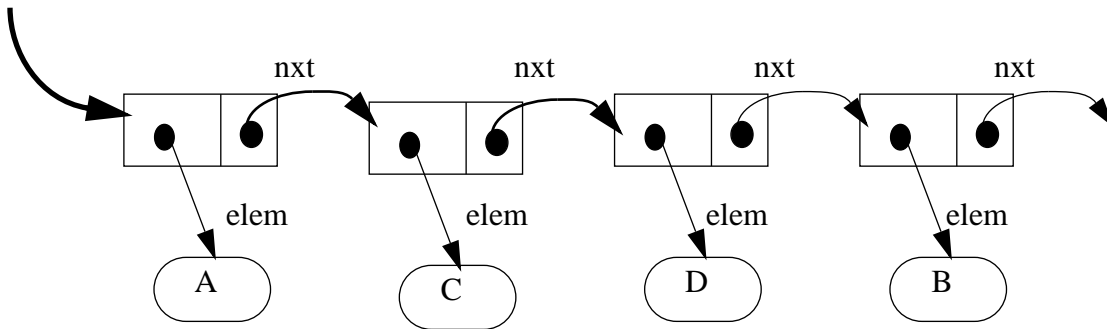
Position ADT



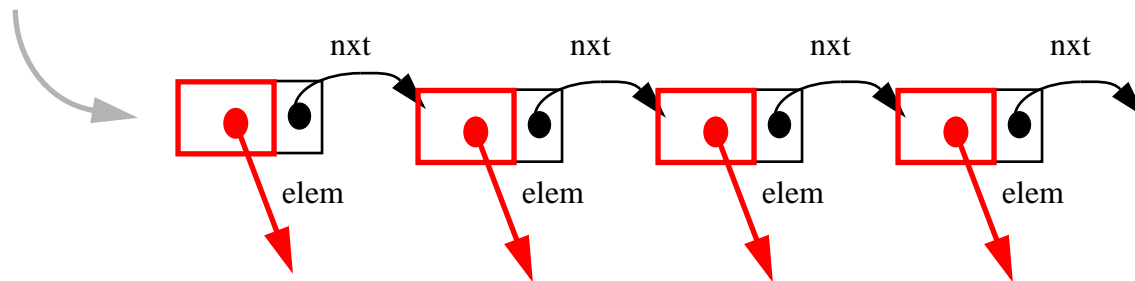
```
class Node {  
    Node nxt;  
    Object elem;  
    ...  
}
```



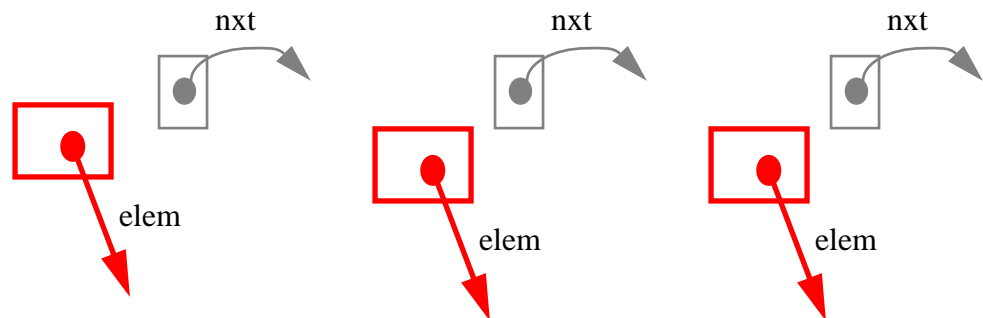
Position ADT



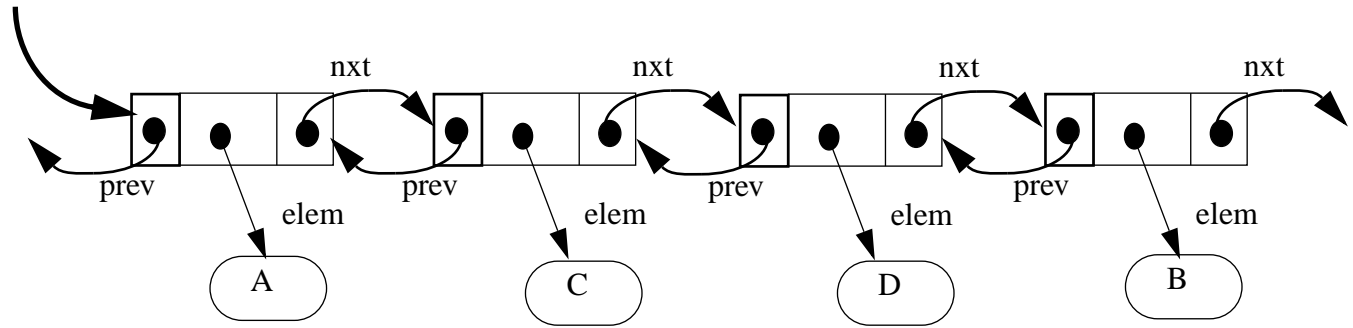
```
class Node {
    Node nxt;
    Object elem;
    ...
}
```



```
interface Position {
    Object element();
}
```

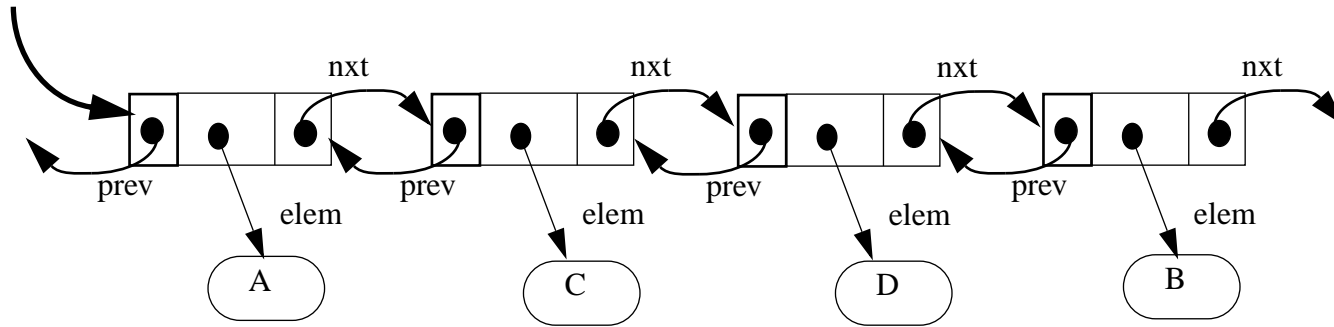


Position ADT

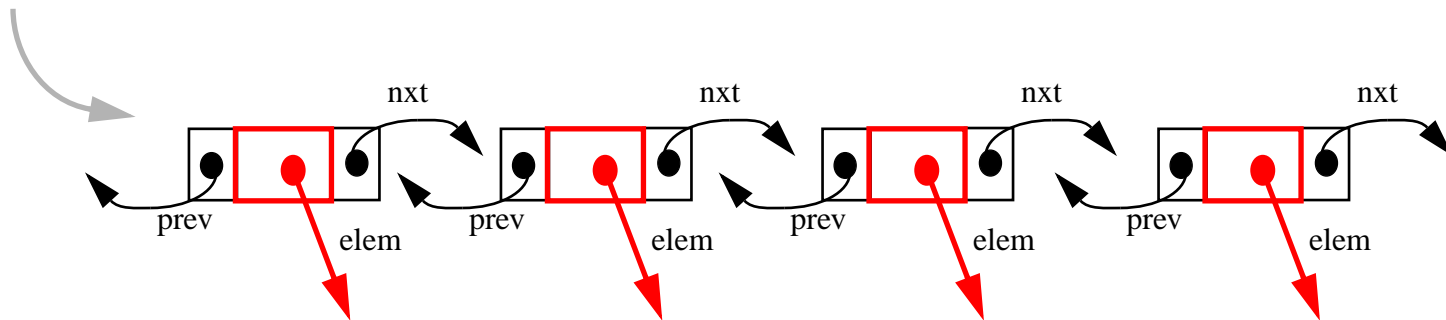


```
class Node {  
    Node nxt;  
    Node prev;  
    Object elem;  
    ...  
}
```

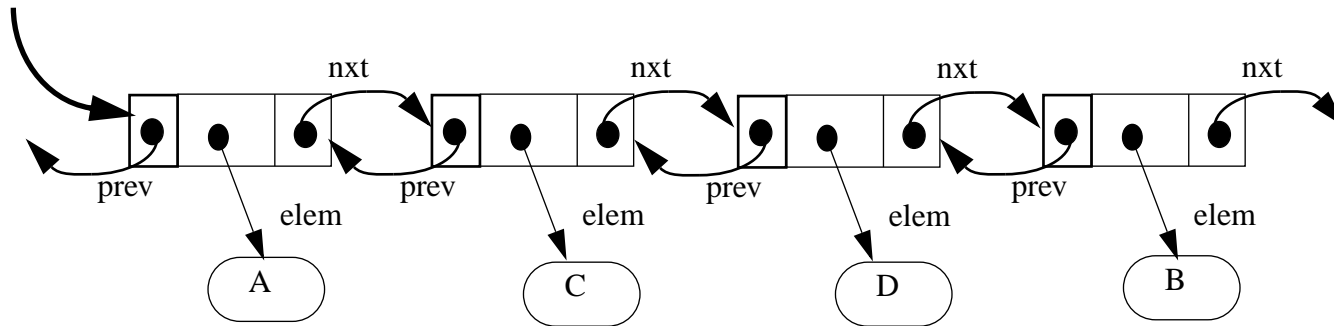
Position ADT



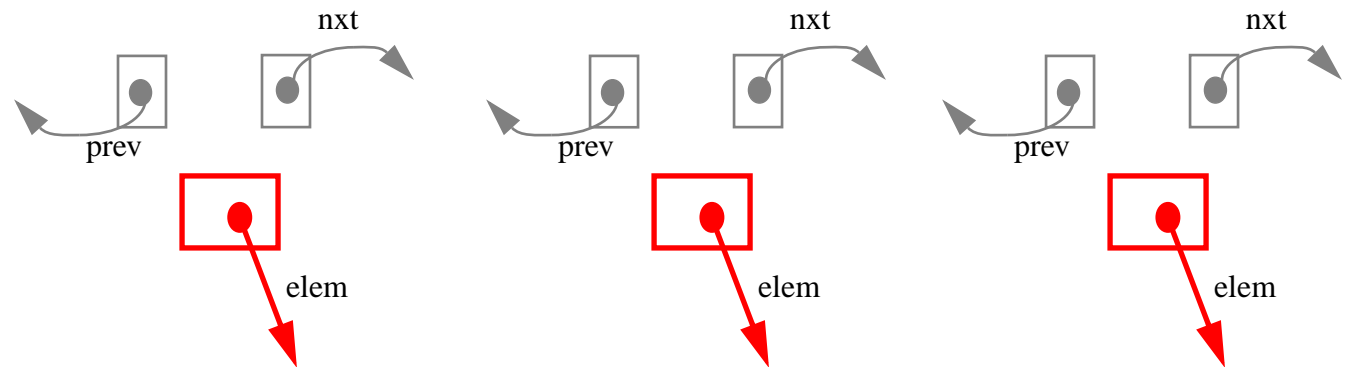
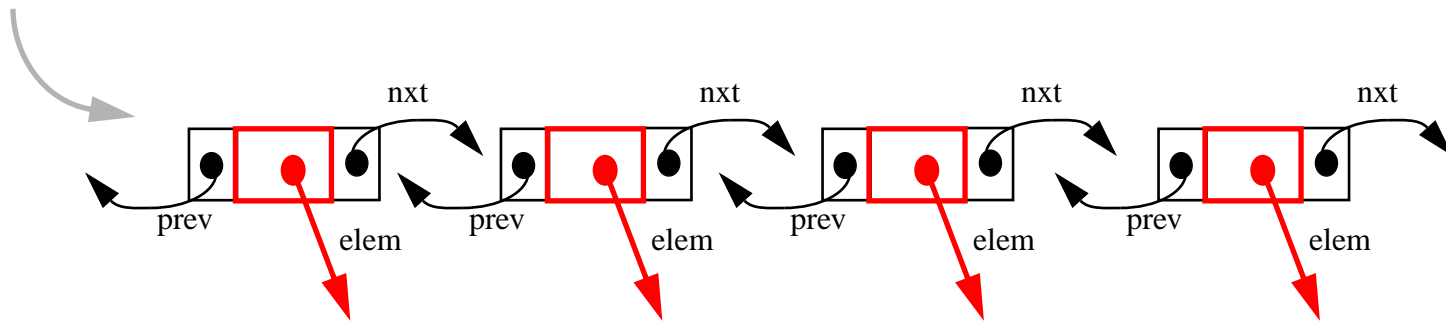
```
class Node {  
    Node nxt;  
    Node prev;  
    Object elem;  
    ...  
}
```



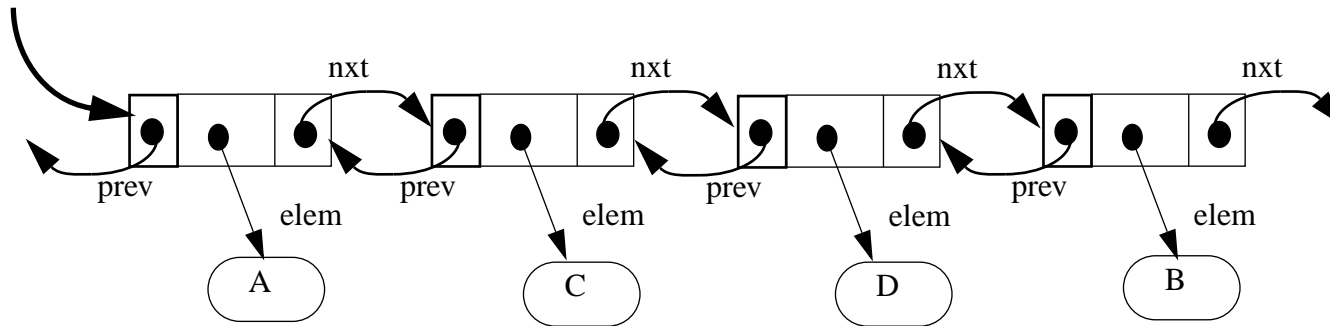
Position ADT



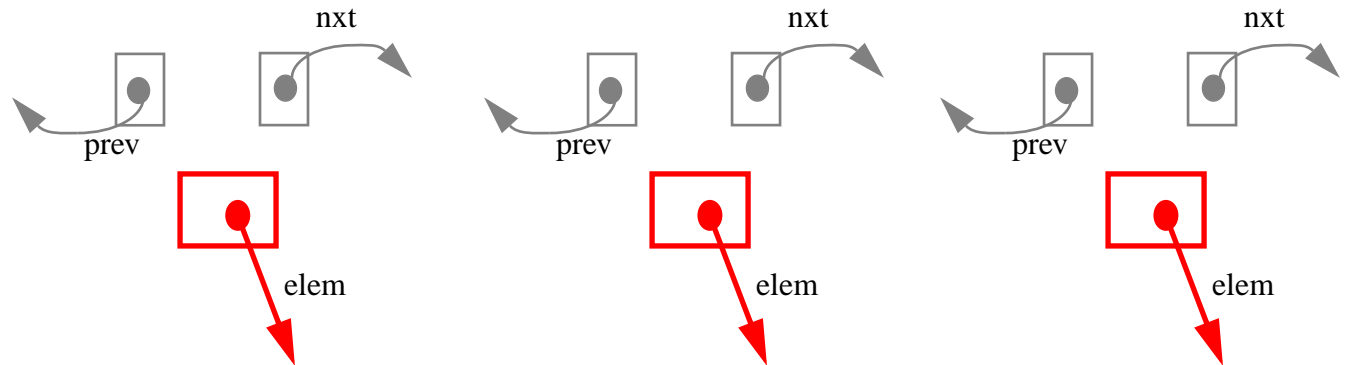
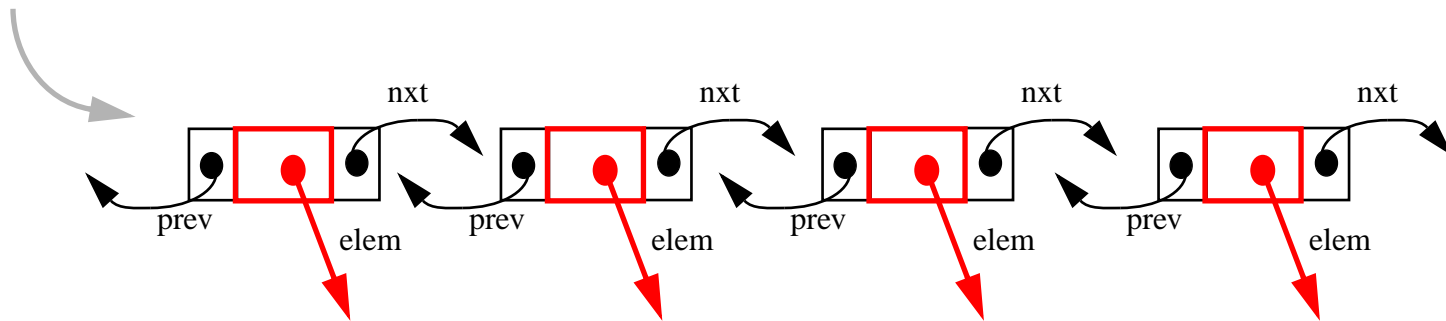
```
class Node {  
    Node nxt;  
    Node prev;  
    Object elem;  
    ...  
}
```



Position ADT



```
class Node {
    Node nxt;
    Node prev;
    Object elem;
    ...
}
```



```
interface Position {
    Object element();
}
```

Position ADT

*= abstraksjon av “et sted”
som kan lagre noe*

```
// jdsl.core.api.Accessor!!!  
public interface Position {  
    Object element();  
// void setElement(Object o);  
}
```

Position ADT

= *abstraksjon av “et sted”
som kan lagre noe*

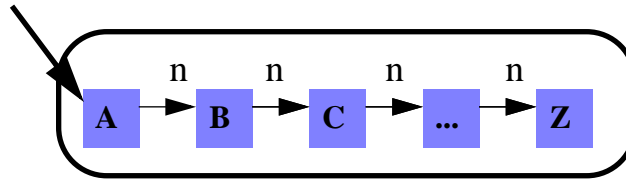
```
// jdsl.core.api.Accessor!!!  
public interface Position {  
    Object element();  
// void setElement(Object o);  
}
```

“*Rå struktur*”

= *aksess-metoder til
en samling av Position'er*

Position ADT

= *abstraksjon av “et sted”
som kan lagre noe*



```
// jdsl.core.api.Accessor!!!
```

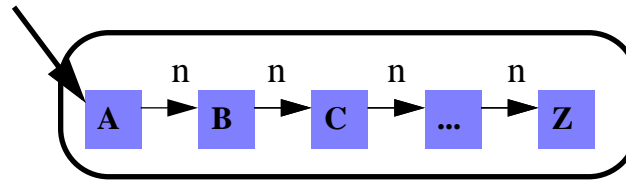
```
public interface Position {  
    Object element();  
// void setElement(Object o);  
}
```

“Rå struktur”

= *aksess-metoder til
en samling av Position'er*

Position ADT

= *abstraksjon av “et sted”
som kan lagre noe*



Position first()
Position next(**Position** p)

```
// jdsl.core.api.Accessor!!!
```

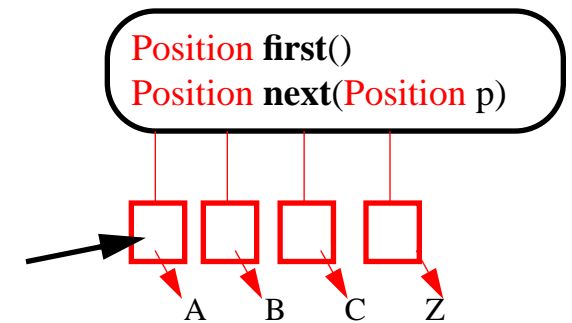
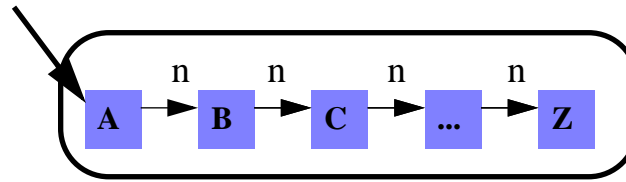
```
public interface Position {  
    Object element();  
// void setElement(Object o);  
}
```

“Rå struktur”

= *aksess-metoder til
en samling av Position'er*

Position ADT

= *abstraksjon av "et sted"*
som kan lagre noe



```
// jdsl.core.api.Accessor!!!
```

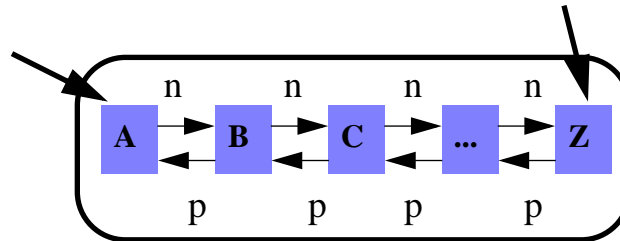
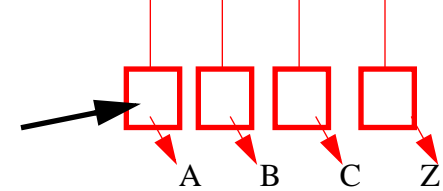
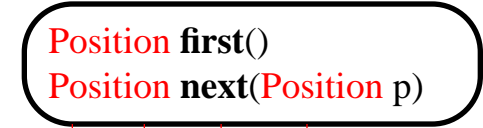
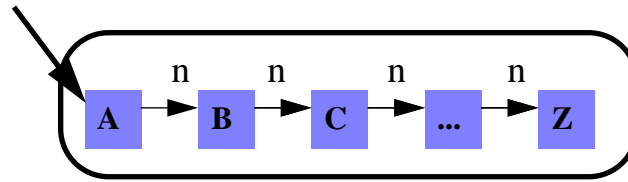
```
public interface Position {  
    Object element();  
    // void setElement(Object o);  
}
```

“Rå struktur”

= *aksess-metoder til*
en samling av Position'er

Position ADT

= *abstraksjon av "et sted"*
som kan lagre noe



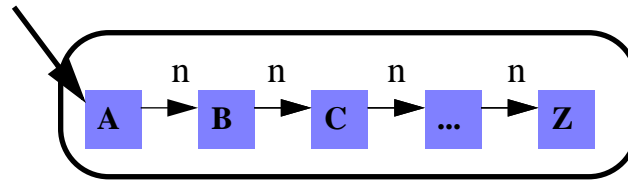
```
// jdsl.core.api.Accessor!!!  
public interface Position {  
    Object element();  
    // void setElement(Object o);  
}
```

“Rå struktur”

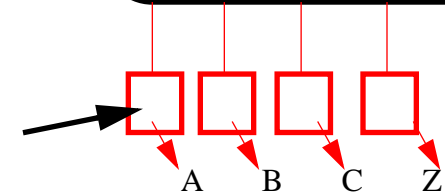
= *aksess-metoder til*
en samling av Position'er

Position ADT

= *abstraksjon av "et sted"*
som kan lagre noe



Position first()
Position next(**Position** p)



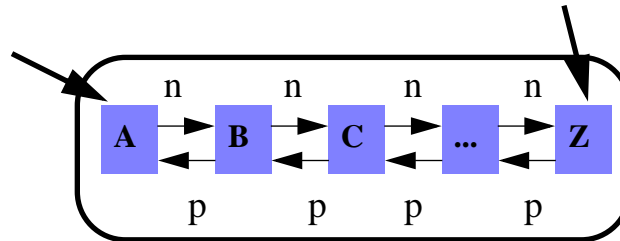
```
// jdsl.core.api.Accessor!!!
```

```
public interface Position {
```

```
    Object element();
```

```
// void setElement(Object o);
```

```
}
```



Position first(), last()
Position next(**Position** p)
Position prev(**Position** p)

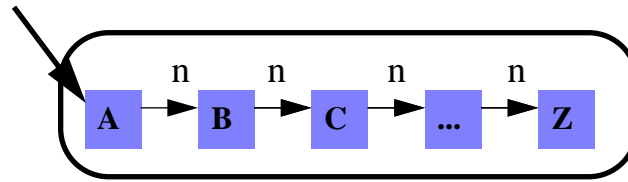
"Rå struktur"

= *aksess-metoder til*

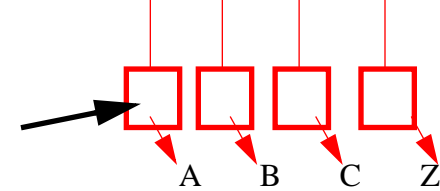
en samling av Position'er

Position ADT

= *abstraksjon av "et sted"*
som kan lagre noe



Position first()
Position next(**Position** p)



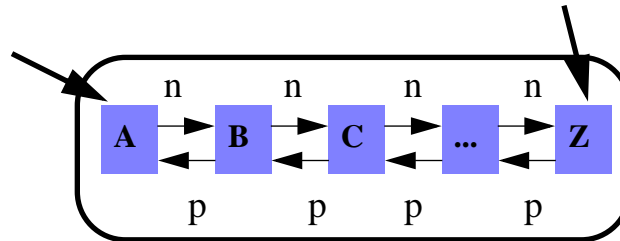
```
// jdsl.core.api.Accessor!!!
```

```
public interface Position {
```

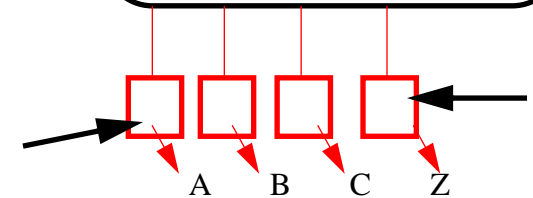
```
    Object element();
```

```
// void setElement(Object o);
```

```
}
```



Position first(), last()
Position next(**Position** p)
Position prev(**Position** p)



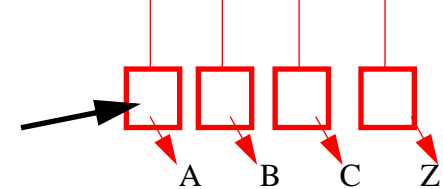
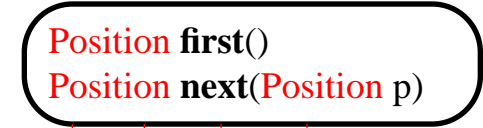
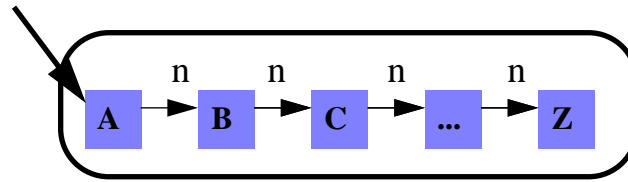
"Rå struktur"

= *aksess-metoder til*

en samling av Position'er

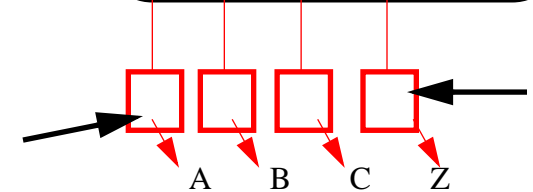
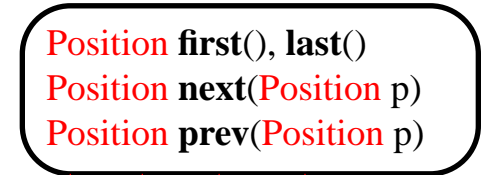
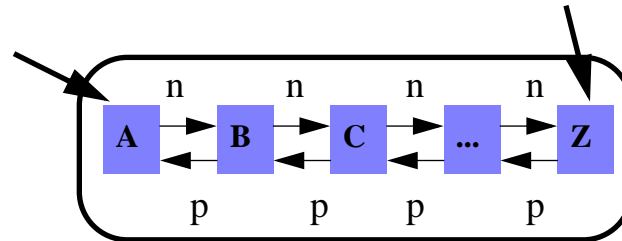
Position ADT

= *abstraksjon av "et sted"*
som kan lagre noe



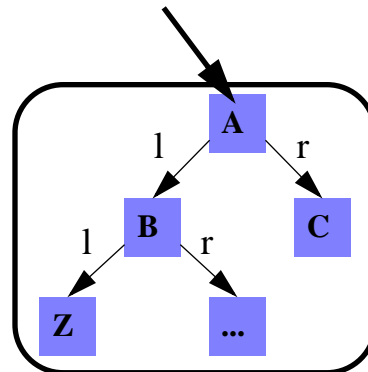
// jdsl.core.api.Accessor!!!

```
public interface Position {
    Object element();
    // void setElement(Object o);
}
```



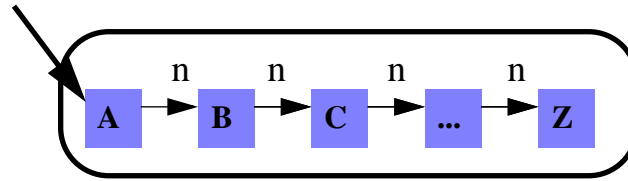
“Rå struktur”

= *aksess-metoder til*
en samling av Position'er

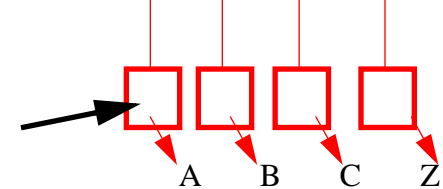


Position ADT

= *abstraksjon av "et sted"*
som kan lagre noe

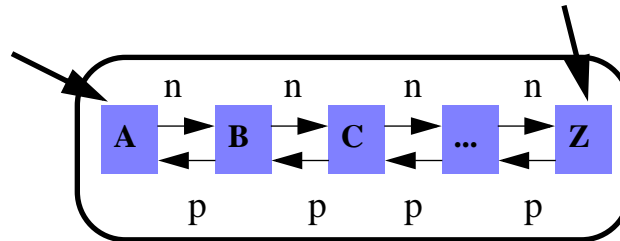


Position first()
Position next(**Position** p)

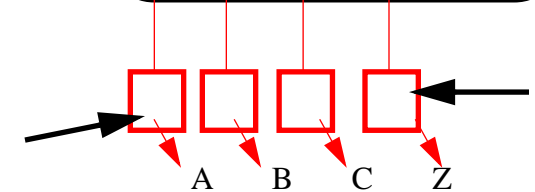


// jdsl.core.api.Accessor!!!

```
public interface Position {
    Object element();
    // void setElement(Object o);
}
```

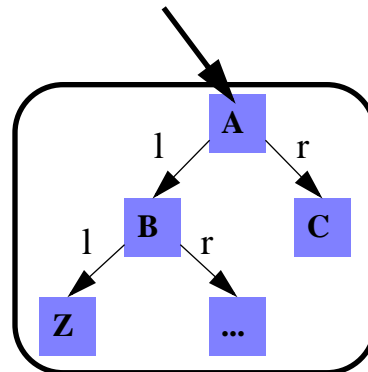


Position first(), last()
Position next(**Position** p)
Position prev(**Position** p)



“Rå struktur”

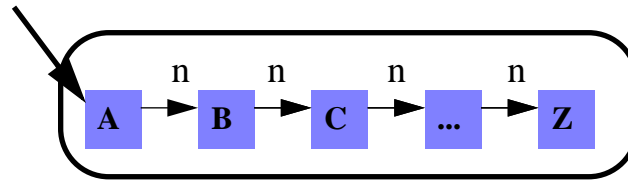
= *aksess-metoder til*
en samling av Position'er



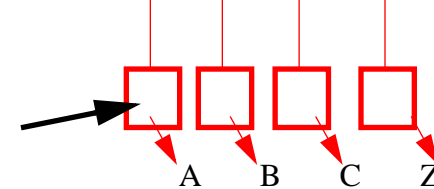
Position root()
Position left(**Position** p)
Position right(**Position** p)

Position ADT

= *abstraksjon av "et sted"*
som kan lagre noe

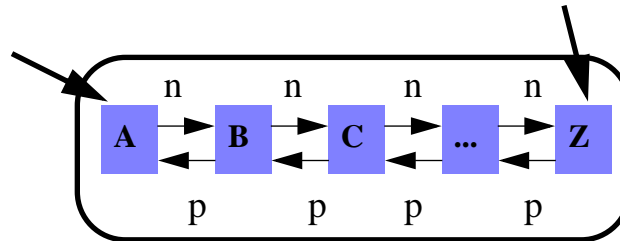


Position first()
Position next(**Position** p)

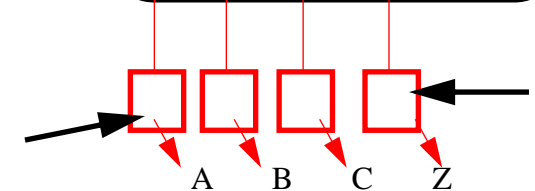


// jdsl.core.api.Accessor!!!

```
public interface Position {
    Object element();
    // void setElement(Object o);
}
```

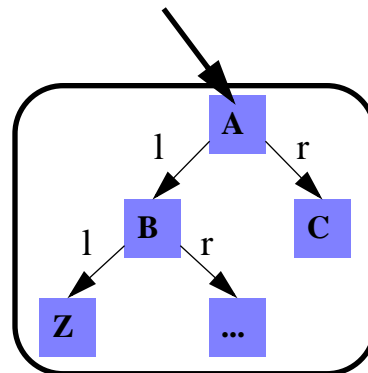


Position first(), last()
Position next(**Position** p)
Position prev(**Position** p)

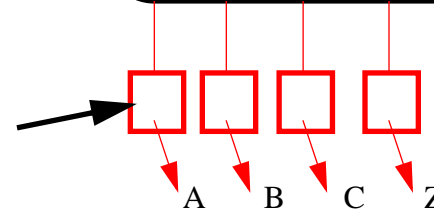


“Rå struktur”

= *aksess-metoder til*
en samling av Position'er



Position root()
Position left(**Position** p)
Position right(**Position** p)



interface PositionalContainer i JDSL

```
package jdsl.core.api;
```

```
/** generisk samling av Objekter som aksesseres vha posisjon */
```

```
public interface PositionalContainer extends Container, InspectablePositionalContainer {
```

```
/** @return enumerering av posisjonene fra samlingen */
```

```
PositionIterator positions(); // (fra InspectablePosCont.)
```

interface PositionalContainer i JDSL

```
package jdsl.core.api;
```

```
/** generisk samling av Objekter som aksesseres vha posisjon */
```

```
public interface PositionalContainer extends Container, InspectablePositionalContainer {
```

```
/** @return enumerering av posisjonene fra samlingen */
```

```
PositionIterator positions(); // (fra InspectablePosCont.)
```

```
/** erstatt Objektet ved posisjon p med Objektet o (fra Container)
```

```
* @param p posisjon der Objektet skal erstattes
```

```
* @param o det nye Objektet som skal plasseres ved p
```

```
* @return det gamle Objektet lagret opprinnelig ved p
```

```
* @exception InvalidAccessorException hvis p ikke er i denne samlingen */
```

```
Object replaceElement(Position p, Object o) throws InvalidAccessorException;
```

interface PositionalContainer i JDSL

```
package jdsl.core.api;
```

```
/** generisk samling av Objekter som aksesseres vha posisjon */  
public interface PositionalContainer extends Container, InspectablePositionalContainer {  
  
/** @return enumerering av posisjonene fra samlingen */  
PositionIterator positions(); // (fra InspectablePosCont.)  
  
/** erstatt Objektet ved posisjon p med Objektet o (fra Container)  
* @param p posisjon der Objektet skal erstattes  
* @param o det nye Objektet som skal plasseres ved p  
* @return det gamle Objektet lagret opprinnelig ved p  
* @exception InvalidAccessorException hvis p ikke er i denne samlingen */  
Object replaceElement(Position p, Object o) throws InvalidAccessorException;  
  
.....  
  
/** bytt Objektene lagret ved argumentposisjoner  
* @param p, q posisjoner i samlingen  
* @exception InvalidAccessorException hvis p eller q ikke er i denne samlingen */  
void swapElements(Position p, Position q) throws InvalidAccessorException;  
}
```

List ADT

```
public interface List extends  
    PositionalContainer, InspectableList {
```

```
// Position akses _____ .
```

```
    Position first();
```

```
    Position last();
```

```
    Position before(Position v);
```

```
    Position after(Position v);
```

```
    boolean isFirst(Position v);
```

```
    boolean isLast(Position v);
```

List ADT

```
public interface List extends
    PositionalContainer, InspectableList {
    // Position aksess _____

    /** @return posisjonen til første elementet i sekvensen
     *  * første elem fåes ved first().element() */
    Position first();

    /** @return posisjon til siste elementet i sekvensen */
    Position last();

    /** @param v en posisjon i sekvensen
     *  * @return posisjon til elementet rett før v i sekvensen
     *  * @exception BoundaryViolationException om v==first() */
    Position before(Position v);

    /** @param v en posisjon i sekvensen
     *  * @return posisjon til elementet rett etter v i sekvensen
     *  * @exception BoundaryViolationException om v==last() */
    Position after(Position v);

    boolean isFirst(Position v);
    boolean isLast(Position v);
}
```

List ADT

```
public interface List extends
    PositionalContainer, InspectableList {
    // Position aksess _____
    /** @return posisjonen til første elementet i sekvensen
     *  første elem fåes ved first().element() */
    Position first();
    /** @return posisjon til siste elementet i sekvensen */
    Position last();
    /** @param v en posisjon i sekvensen
     *  @return posisjon til elementet rett før v i sekvensen
     *  @exception BoundaryViolationException om v==first() */
    Position before(Position v);
    /** @param v en posisjon i sekvensen
     *  @return posisjon til elementet rett etter v i sekvensen
     *  @exception BoundaryViolationException om v==last() */
    Position after(Position v);
    boolean isFirst(Position v);
    boolean isLast(Position v);
```

```
// element håndtering _____
Position insertFirst(Object e);
Position insertLast(Object e);
Position insertBefore(Position v, Object e);
Position insertAfter(Position v, Object e);
Object remove(Position v);
```

List ADT

```
public interface List extends
    PositionalContainer, InspectableList {
    // Position aksess _____
    /** @return posisjonen til første elementet i sekvensen
     * første elem fåes ved first().element() */
    Position first();
    /** @return posisjon til siste elementet i sekvensen */
    Position last();
    /** @param v en posisjon i sekvensen
     * @return posisjon til elementet rett før v i sekvensen
     * @exception BoundaryViolationException om v==first() */
    Position before(Position v);
    /** @param v en posisjon i sekvensen
     * @return posisjon til elementet rett etter v i sekvensen
     * @exception BoundaryViolationException om v==last() */
    Position after(Position v);
    boolean isFirst(Position v);
    boolean isLast(Position v);
```

```
// element håndtering _____
    Position insertFirst(Object e);
    Position insertLast(Object e);
    Position insertBefore(Position v, Object e);
    Position insertAfter(Position v, Object e);
    Object remove(Position v);
    // int size(); boolean isEmpty();
    // ObjectIterator elements();
    // Container newContainer();
    // PositionIterator positions();
    // void swapElements(Position v, Position w);
    // Object replaceElement(Position v, Object e);
}
```

List ADT

public interface List extends

PositionalContainer, InspectableList {

```
// Position aksess _____  
  
/** @return posisjonen til første elementet i sekvensen  
 * første elem fåes ved first().element() */  
Position first();  
  
/** @return posisjon til siste elementet i sekvensen */  
Position last();  
  
/** @param v en posisjon i sekvensen  
 * @return posisjon til elementet rett før v i sekvensen  
 * @exception BoundaryViolationException om v==first() */  
Position before(Position v);  
  
/** @param v en posisjon i sekvensen  
 * @return posisjon til elementet rett etter v i sekvensen  
 * @exception BoundaryViolationException om v==last() */  
Position after(Position v);  
  
boolean isFirst(Position v);  
boolean isLast(Position v);
```

```
// element håndtering _____  
  
/** @return posisjon til det innsatte elementet */  
Position insertFirst(Object e);  
  
/** @return posisjon til det innsatte elementet */  
Position insertLast(Object e);  
  
/** sett inn elt o i sekvensen rett foran det i v */  
Position insertBefore(Position v, Object e);  
  
/** sett inn elt o i sekvensen rett etter det i v */  
Position insertAfter(Position v, Object e);  
  
/** @return Objektet som ble fjernet */  
Object remove(Position v);  
  
// int size(); boolean isEmpty();  
  
// ObjectIterator elements();  
// Container newContainer();  
  
// PositionIterator positions();  
// void swapElements(Position v, Position w);  
// Object replaceElement(Position v, Object e);  
}
```


List ADT

public interface List extends

PositionalContainer, InspectableList {

// Position aksess _____ implem.

/** @return posisjonen til første elementet i sekvensen
* første elem fåes ved first().element() */

Position **first**(); $O(1)$

/** @return posisjon til siste elementet i sekvensen */

Position **last**(); $O(1)$

/** @param v en posisjon i sekvensen

* @return posisjon til elementet rett før v i sekvensen
* @exception BoundaryViolationException om $v == \text{first}()$ */

Position **before**(Position v); $O(1)$

/** @param v en posisjon i sekvensen

* @return posisjon til elementet rett etter v i sekvensen
* @exception BoundaryViolationException om $v == \text{last}()$ */

Position **after**(Position v); $O(1)$

boolean **isFirst**(Position v); $O(1)$

boolean **isLast**(Position v); $O(1)$

// element håndtering _____ implem.

/** @return posisjon til det innsatte elementet */

Position **insertFirst**(Object e); $O(1)$

/** @return posisjon til det innsatte elementet */

Position **insertLast**(Object e); $O(1)$

/** sett inn elt o i sekvensen rett foran det i v */

Position **insertBefore**(Position v, Object e); $O(1)$

/** sett inn elt o i sekvensen rett etter det i v */

Position **insertAfter**(Position v, Object e); $O(1)$

/** @return Objektet som ble fjernet */

Object **remove**(Position v); $O(1)$

// int **size**(); boolean **isEmpty**(); $O(1)$

// ObjectIterator **elements**(); $O(n)$

// Container **newContainer**(); $O(1)$

// PositionIterator **positions**(); $O(n)$

// void **swapElements**(Position v, Position w); $O(1)$

// Object **replaceElement**(Position v, Object e); $O(1)$

}

List ADT

public interface List extends

PositionalContainer, InspectableList {

// Position aksess _____ implem.

/** @return posisjonen til første elementet i sekvensen
* første elem fåes ved first().element() */

Position **first**(); $O(1)$

/** @return posisjon til siste elementet i sekvensen */

Position **last**(); $O(1)$

/** @param v en posisjon i sekvensen

* @return posisjon til elementet rett før v i sekvensen
* @exception BoundaryViolationException om $v == \text{first}()$ */

Position **before**(Position v); $O(1)$

/** @param v en posisjon i sekvensen

* @return posisjon til elementet rett etter v i sekvensen
* @exception BoundaryViolationException om $v == \text{last}()$ */

Position **after**(Position v); $O(1)$

boolean **isFirst**(Position v); $O(1)$

boolean **isLast**(Position v); $O(1)$

// element håndtering _____ implem.

/** @return posisjon til det innsatte elementet */

Position **insertFirst**(Object e); $O(1)$

/** @return posisjon til det innsatte elementet */

Position **insertLast**(Object e); $O(1)$

/** sett inn elt o i sekvensen rett foran det i v */

Position **insertBefore**(Position v, Object e); $O(1)$

/** sett inn elt o i sekvensen rett etter det i v */

Position **insertAfter**(Position v, Object e); $O(1)$

/** @return Objektet som ble fjernet */

Object **remove**(Position v); $O(1)$

// int **size**(); boolean **isEmpty**(); $O(1)$

// ObjectIterator **elements**(); $O(n)$

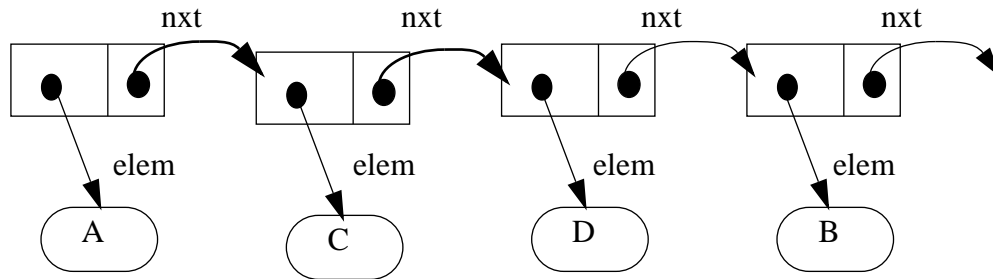
// Container **newContainer**(); $O(1)$

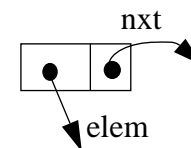
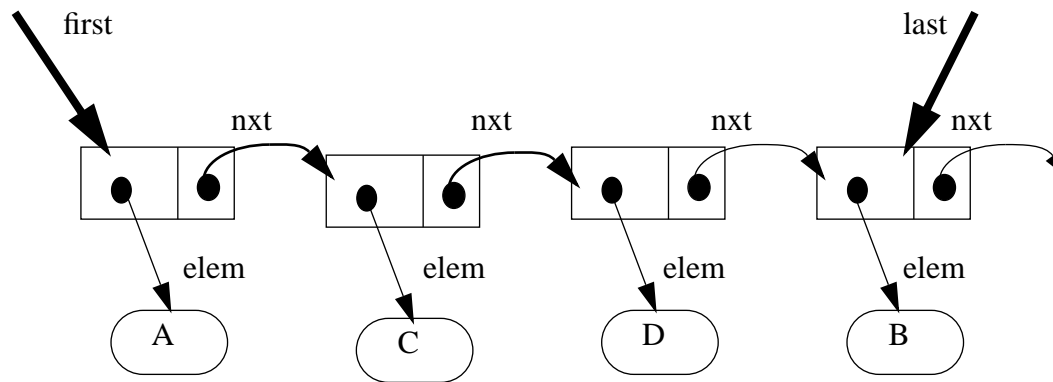
// PositionIterator **positions**(); $O(n)$

// void **swapElements**(Position v, Position w); $O(1)$

// Object **replaceElement**(Position v, Object e); $O(1)$

<i>implementasjon med</i>	<i>krever at</i>
<i>en-/to-veis liste</i>	Node implements Position
<i>array</i>	"indeks" implements Position





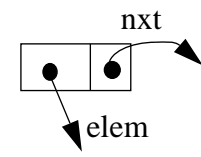
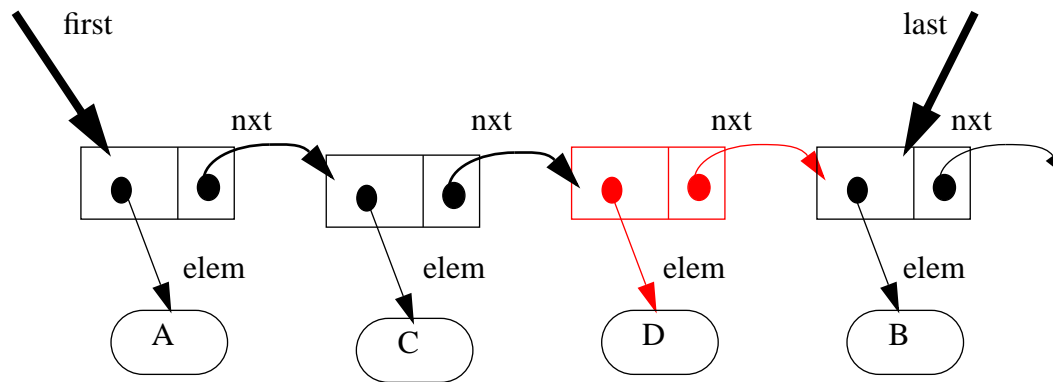
```

EL:
DataStruktur:

class Node
    implements Position

Node first, last

int size
  
```



first()

last()

after(p)

before(p)

insertFirst(o)

insertLast(o)

insertAfter(p,o)

insertBefore(p,o)

remove(p)

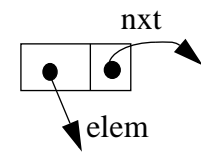
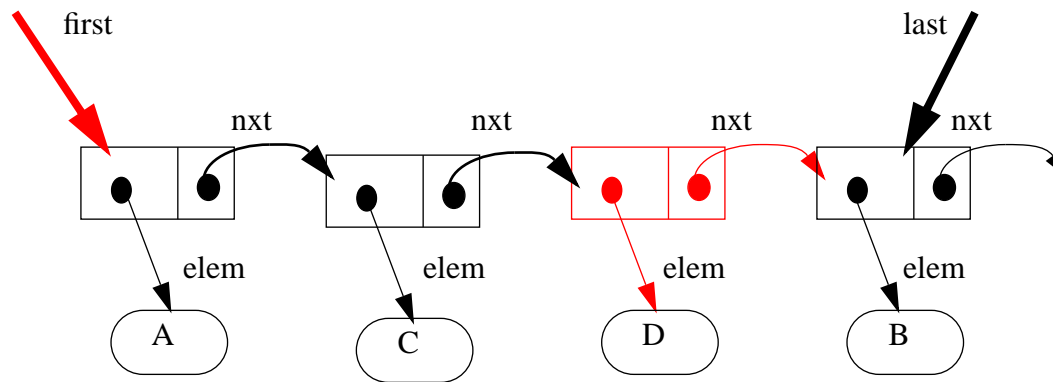
```

EL:
DataStruktur:

class Node
    implements Position

Node first, last

int size
  
```



first()

last()

after(p)

before(p)

insertFirst(o)

insertLast(o)

insertAfter(p,o)

insertBefore(p,o)

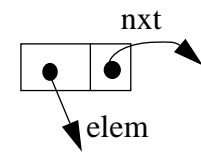
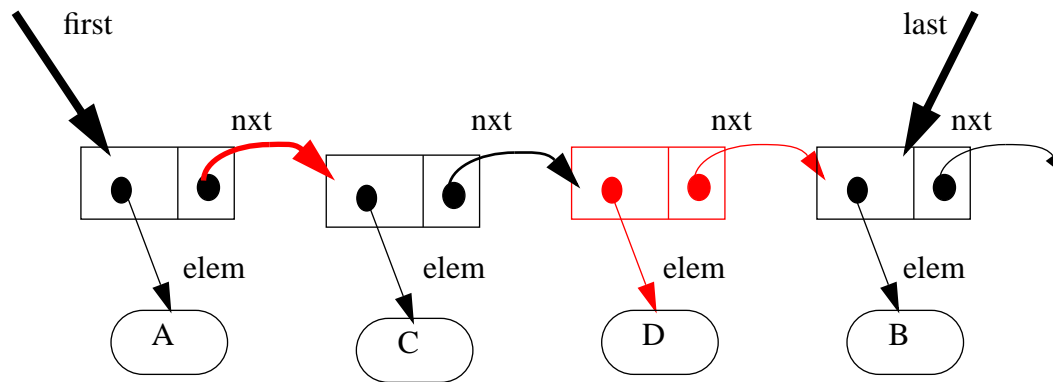
remove(p)

EL:
DataStruktur:

```
class Node
    implements Position
```

Node first, last

int size



first()

last()

after(p)

before(p)

insertFirst(o)

insertLast(o)

insertAfter(p,o)

insertBefore(p,o)

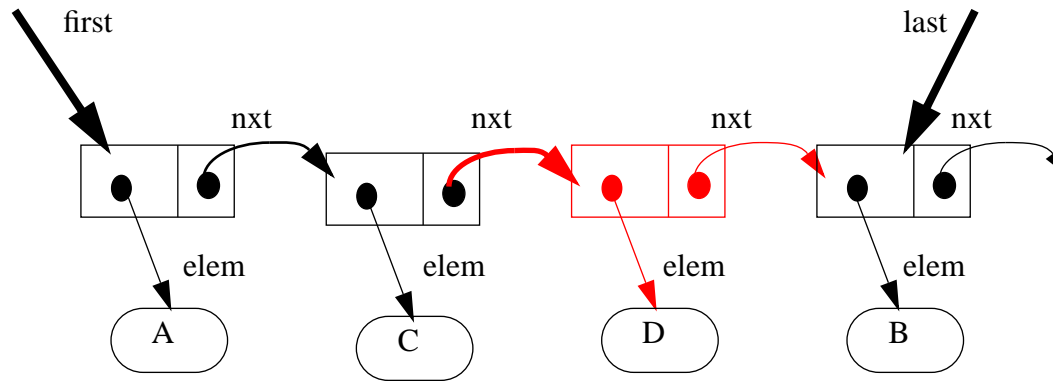
remove(p)

EL:
DataStruktur:

```
class Node
    implements Position
```

Node first, last

int size



first()

last()

after(p)

before(p)

insertFirst(o)

insertLast(o)

insertAfter(p,o)

insertBefore(p,o)

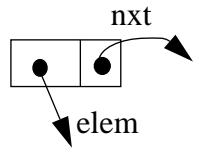
remove(p)

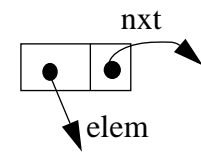
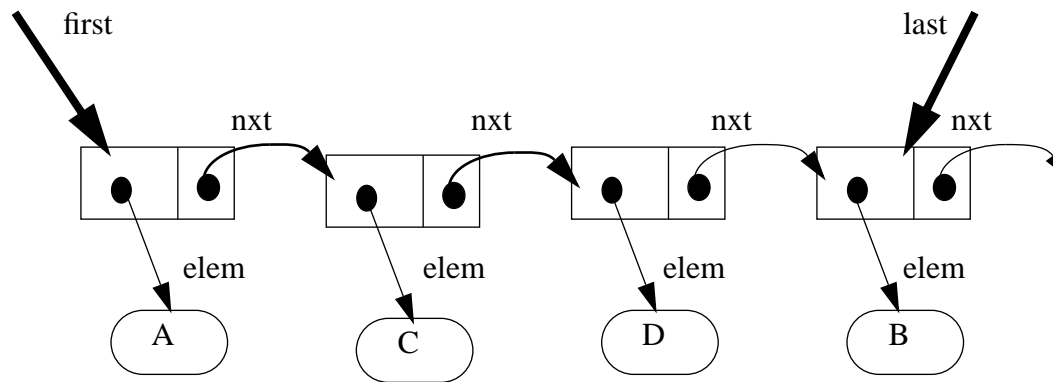
EL:
DataStruktur:

```
class Node
    implements Position
```

Node first, last

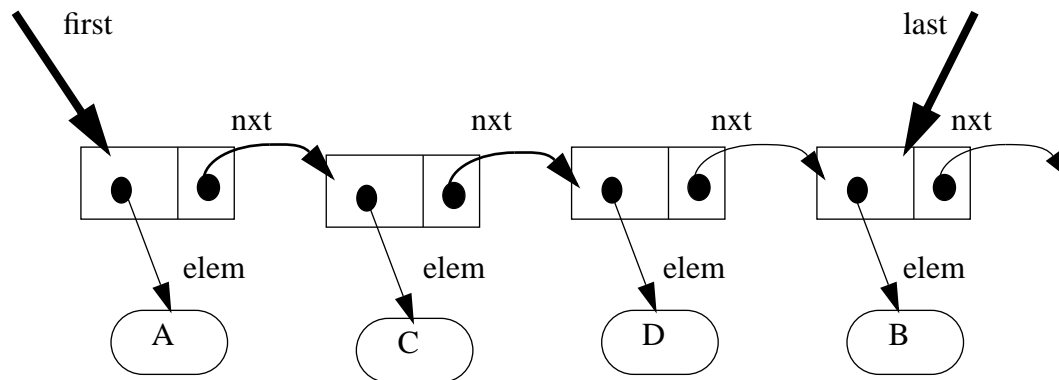
int size





	EL	
first()	1	
last()	1	
after(p)	1	
before(p)	n	
insertFirst(o)	1	
insertLast(o)	n	
insertAfter(p,o)	1	
insertBefore(p,o)	n	
remove(p)	1	

EL:
 DataStruktur:
 class Node
 implements Position
 Node first, last
 int size



EL

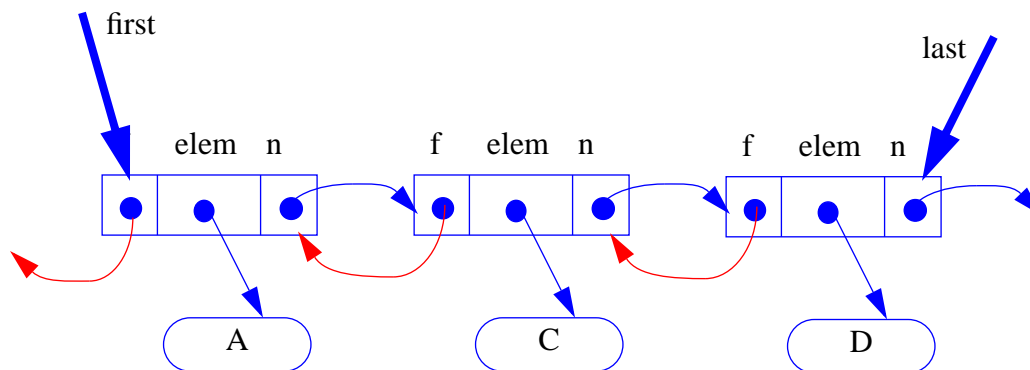
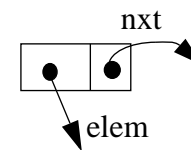
- first() 1
- last() 1
- after(p) 1
- before(p) n**
- insertFirst(o) 1
- insertLast(o) n**
- insertAfter(p,o) 1
- insertBefore(p,o) n**
- remove(p) 1

EL:
DataStruktur:

class Node
implements Position

Node first, last

int size

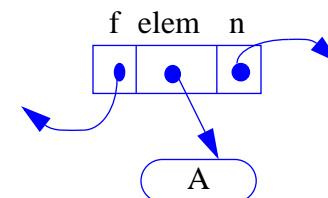


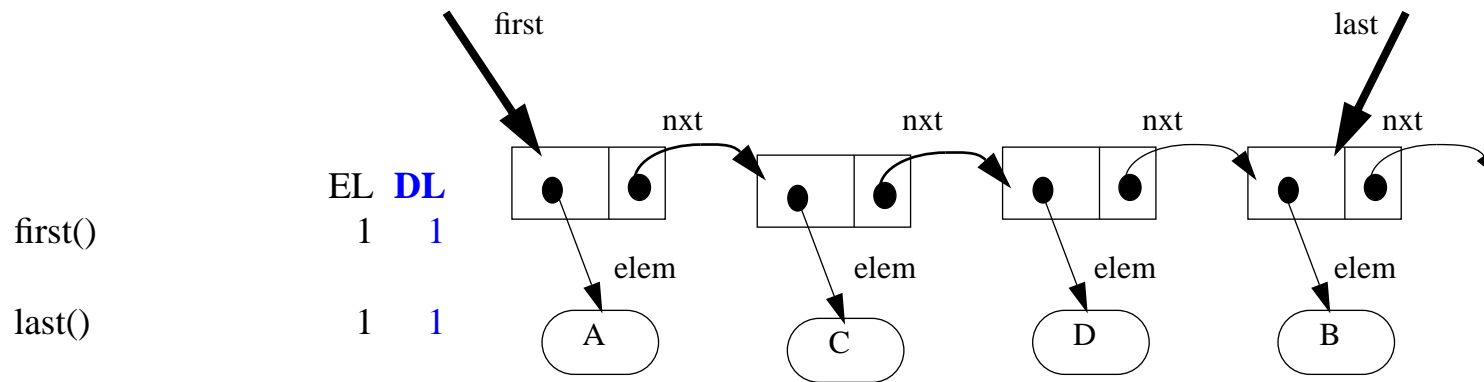
DL:
DataStruktur:

class DLNode
implements Position

Node first, last

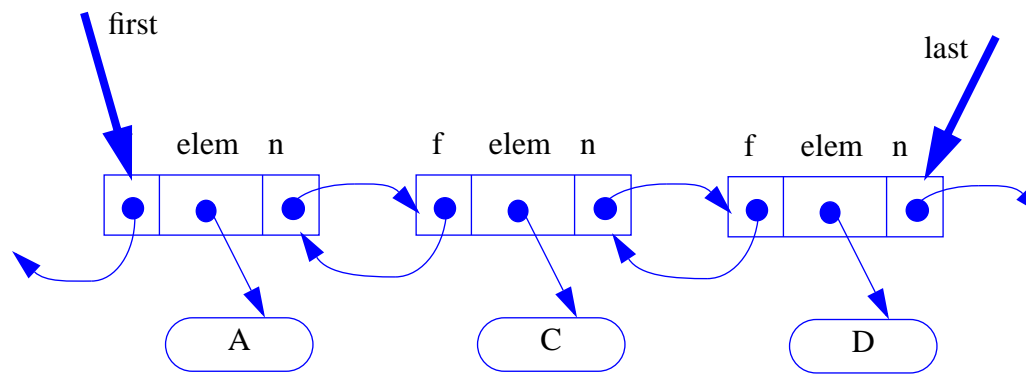
int size





	EL	DL
first()	1	1
last()	1	1
after(p)	1	1
before(p)	n	1
insertFirst(o)	1	1
insertLast(o)	n	1
insertAfter(p,o)	1	1
insertBefore(p,o)	n	1
remove(p)	1	1

EL:
 DataStruktur:
 class Node
 implements Position
 Node first, last
 int size



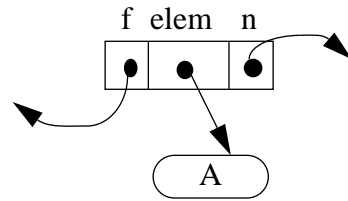
DL:
 DataStruktur:
 class DLNode
 implements Position
 Node first, last
 int size

List med to-veis liste

```
public class DLNode implements Position
{ private DLNode f, n;
  private Object elem;
  public DLNode(Object o, DLNode ff, DLNode nn)
  { elem= o; f= ff; n= nn; }
  public Object element() { return elem; }
  public void setElement(Object o)
  { elem = o; }
  public DLNode getNext() { return n; }
  public DLNode getPrev() { return f; }
  public void setNext(DLNode d) { n= d; }
  public void setPrev(DLNode d) { f= d; }
}
```

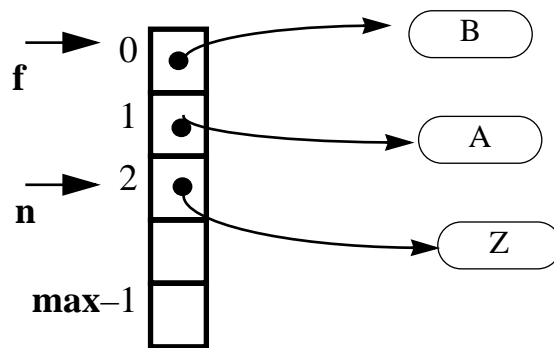
DataStruktur

```
private DLNode first, last;
// antall elementer size =
private int n;
```



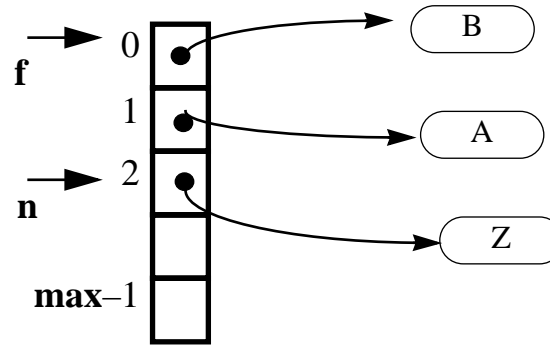
```
public class PSdl implements List {
  public Position last() { return last; }
  public Position before(Position p) {
    return ((DLNode)p).getPrev(); }
  public Position insertFirst(Object o) {
    first = new DLNode(o, null, first);
    n++;
    if (last == null) last = first;
    return first ; }
  public Position insertBefore(Position p, Object o) {
    DLNode pp = (DLNode)p;
    DLNode ny = new DLNode(o, pp.getPrev(), pp);
    n++;
    if isFirst(p) first = ny;
    return ny; }
  ...
}
```

List med array (ikke sirkulær)



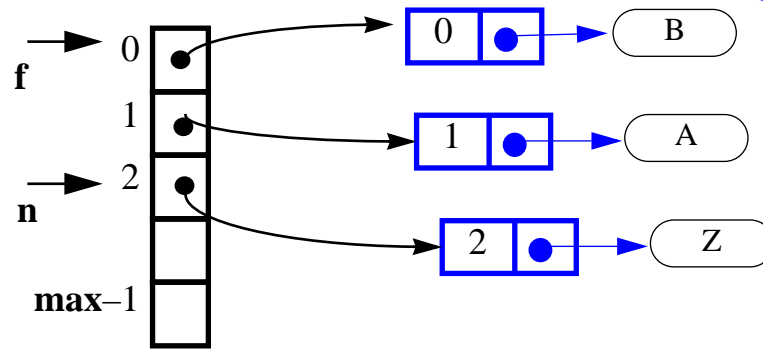
List med array (ikke sirkulær)

```
p = first().after();  
p = insertBefore(p, new A(Z));  
p.element();
```



List med array (ikke sirkulær)

```
p = first().after();  
p = insertBefore(p, new A(Z));  
p.element();
```

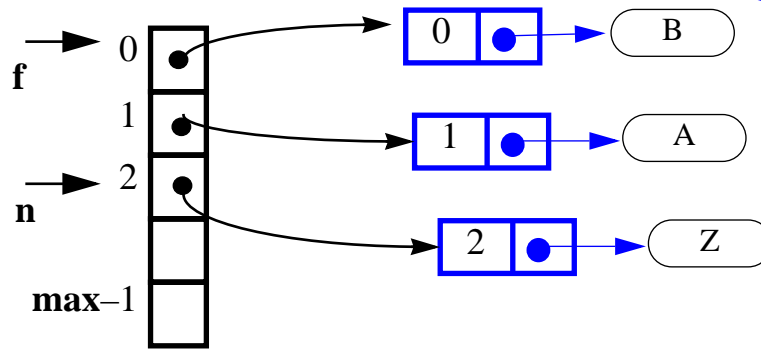


List med array (ikke sirkulær)

Data Representasjon

```
public class ArPos
implements Position
{ private int r;
  private Object obj;
  public ArPos(
    int i, Object o)
  { r= i; obj= o; }
  public element() { return obj; }
  public rank() { return r; }
  public setElement(Object o) { obj= o; }
  public setRank(int i) { r= i; }
}
```

```
p = first().after();
p = insertBefore(p, new A(Z));
p.element();
```



List med array (ikke sirkulær)

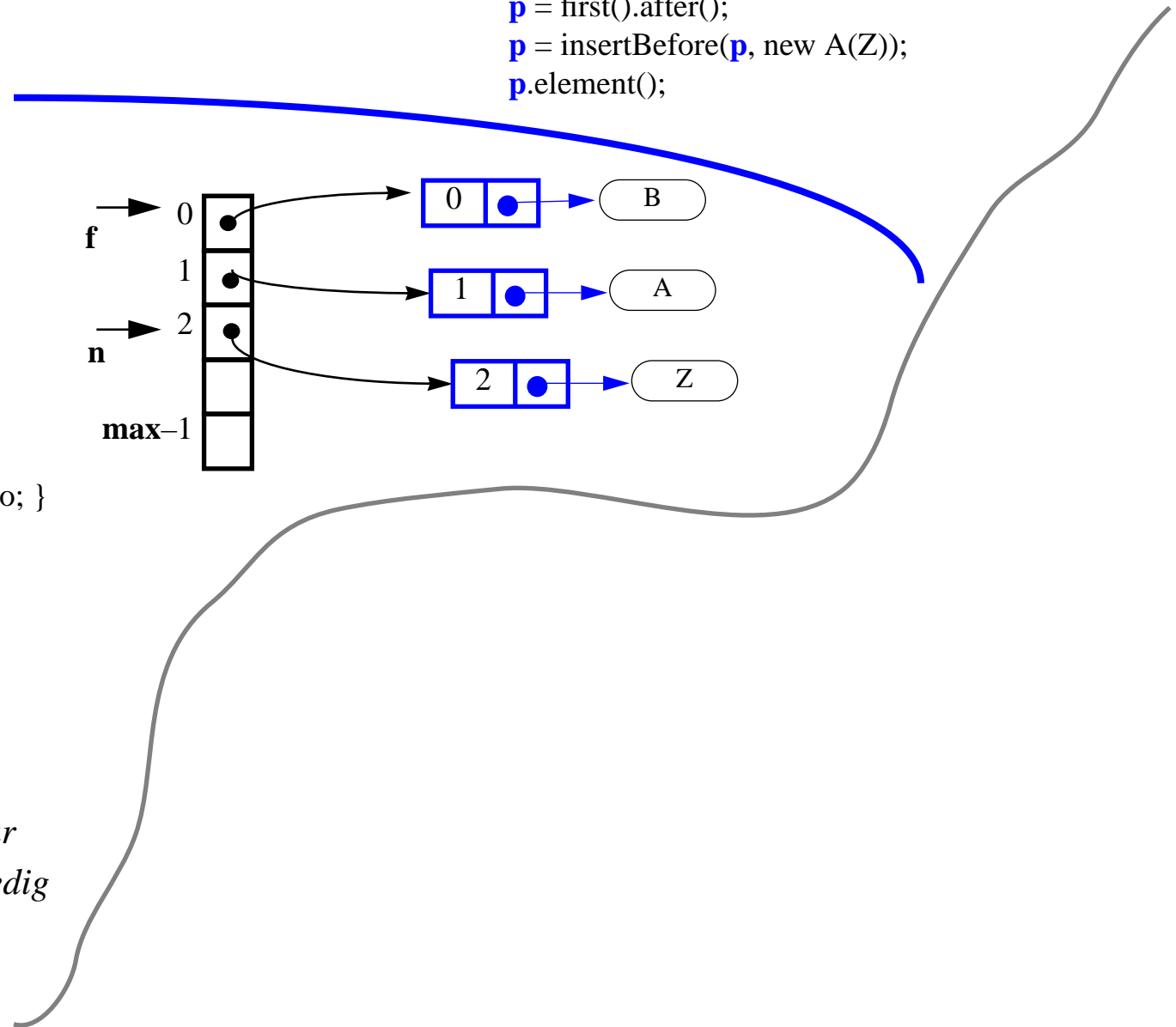
Data Representasjon

```
public class ArPos
implements Position
{ private int r;
  private Object obj;
  public ArPos(
    int i, Object o)
  { r= i; obj= o; }
  public element() { return obj; }
  public rank() { return r; }
  public setElement(Object o) { obj= o; }
  public setRank(int i) { r= i; }
}
```

Data Struktur

```
private ArPos[N] ar;
private int max; // størrelse til ar
private int f, n; // første–neste ledig
```

```
p = first().after();
p = insertBefore(p, new A(Z));
p.element();
```



List med array (ikke sirkulær)

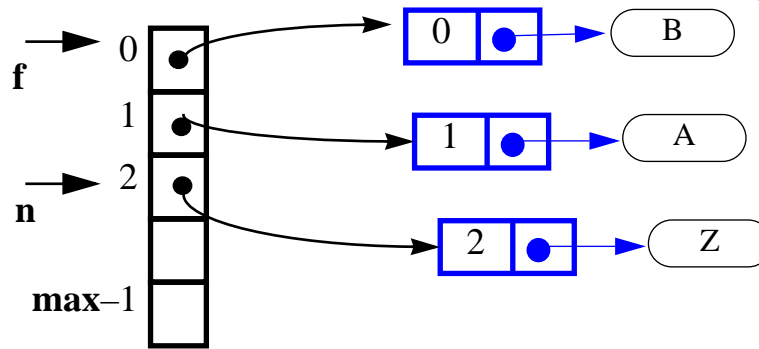
Data Representasjon

```
public class ArPos
implements Position
{ private int r;
  private Object obj;
  public ArPos(
    int i, Object o)
  { r= i; obj= o; }
  public element() { return obj; }
  public rank() { return r; }
  public setElement(Object o) { obj= o; }
  public setRank(int i) { r= i; }
}
```

Data Struktur

```
private ArPos[N] ar;
private int max; // størrelse til ar
private int f, n; // første-neste ledig
```

```
p = first().after();
p = insertBefore(p, new A(Z));
p.element();
```



DataInvariant

$f = 0$

$f \leq n < max$

isEmpty hviss $f == n$;

size = n

ArPos[f..n-1] er fylt med alle elementene

List med array (ikke sirkulær)

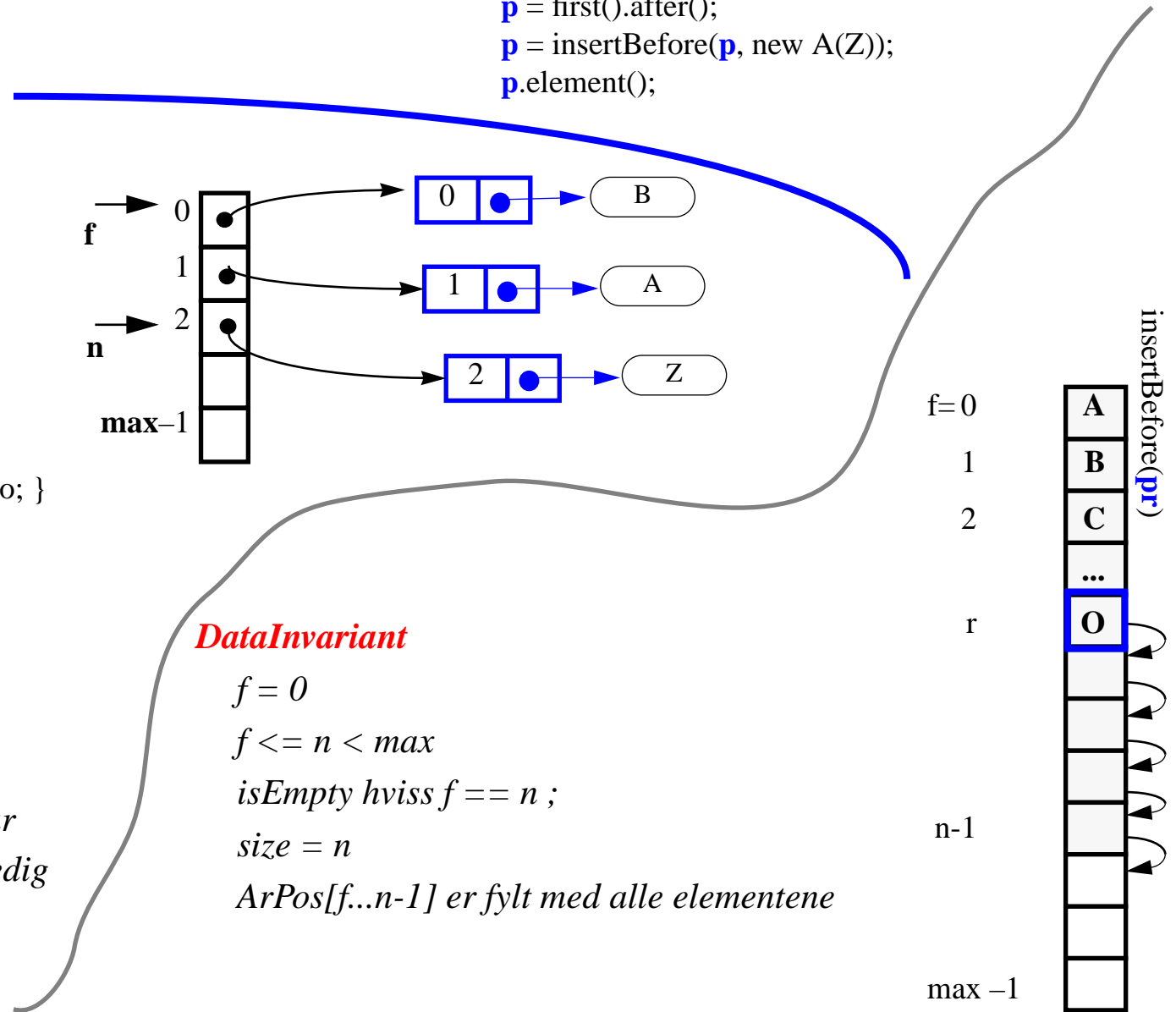
Data Representasjon

```
public class ArPos
implements Position
{ private int r;
  private Object obj;
  public ArPos(
    int i, Object o)
  { r= i; obj= o; }
  public element() { return obj; }
  public rank() { return r; }
  public setElement(Object o) { obj= o; }
  public setRank(int i) { r= i; }
}
```

Data Struktur

```
private ArPos[N] ar;
private int max; // størrelse til ar
private int f, n; // første-neste ledig
```

```
p = first().after();
p = insertBefore(p, new A(Z));
p.element();
```



DataInvariant

- $f = 0$
- $f \leq n < max$
- isEmpty hviss $f == n$;
- size = n
- ArPos[f..n-1] er fylt med alle elementene

List med array (ikke sirkulær)

Data Representasjon

```

public class ArPos
implements Position
{ private int r;
  private Object obj;
  public ArPos(
    int i, Object o)
  { r= i; obj= o; }
  public element() { return obj; }
  public rank() { return r; }
  public setElement(Object o) { obj= o; }
  public setRank(int i) { r= i; }
}
    
```

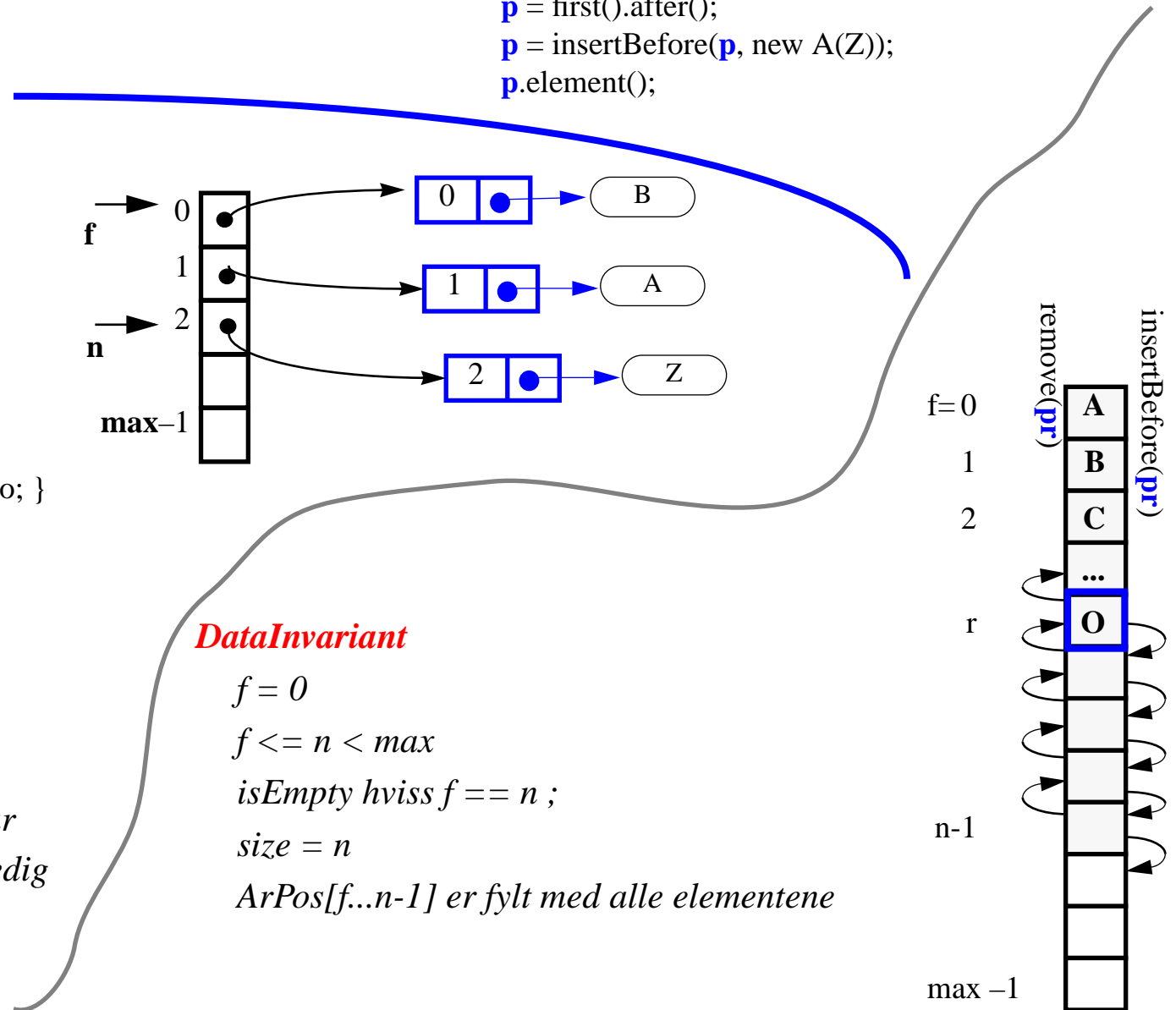
Data Struktur

```

private ArPos[N] ar;
private int max; // størrelse til ar
private int f, n; // første-neste ledig
    
```

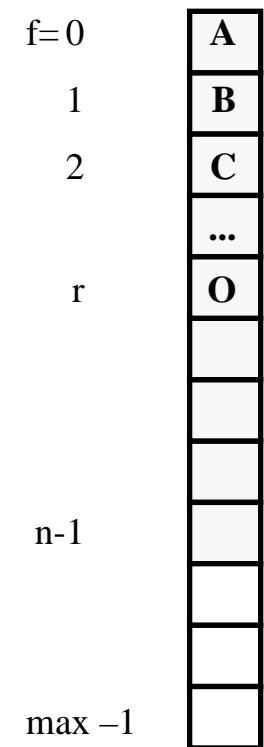
```

p = first().after();
p = insertBefore(p, new A(Z));
p.element();
    
```



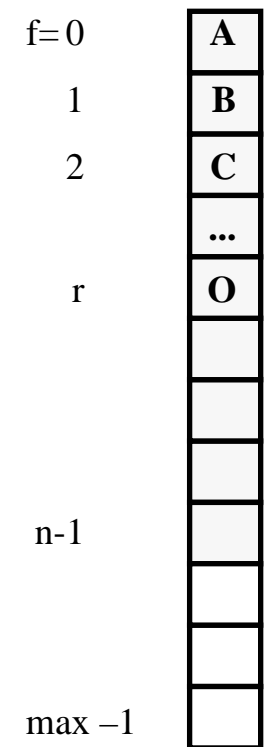
List med array (ikke sirkulær)

```
public class ListAr implements List {  
    public Position last() throws EmptyContainer{  
        if (n > f) return ar[n-1];  
        else throw new EmptyContainer(); }  
}
```



List med array (ikke sirkulær)

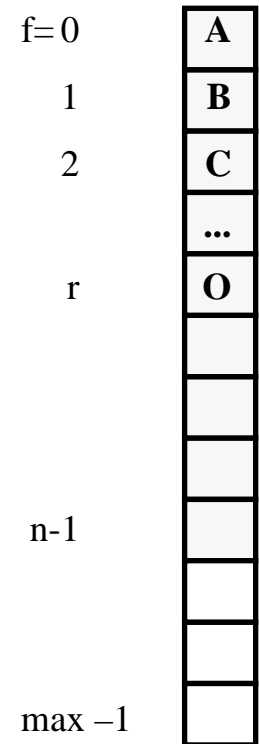
```
public class ListAr implements List {  
    public Position last() throws EmptyContainer{  
        if (n > f) return ar[n-1];  
        else throw new EmptyContainer(); }  
  
    public Position before(Position p) throws InvalidAccessorExc  
    { if ( check(p).rank()==0) throw new ...Exception();  
      return ar[ check(p).rank()-1 ] ; }  
}
```



List med array (ikke sirkulær)

```
public class ListAr implements List {  
    public Position last() throws EmptyContainer{  
        if (n > f) return ar[n-1];  
        else throw new EmptyContainer(); }  
  
    public Position before(Position p) throws InvalidAccessorExc  
    { if ( check(p).rank()==0) throw new ...Exception();  
      return ar[ check(p).rank()-1 ] ; }  
}
```

```
private ArPos check(Position p) throws InvalidAccessorExc {  
    if (! p instanceof ArPos) throw new InvalidAccessorExc();  
    ArPos a = (ArPos)p;  
    if (p.rank() >= 0 && p.rank() <= n) return a;  
    else throw new InvalidAccessorExc(); }  
}
```



List med array (ikke sirkulær)

```

public class ListAr implements List {
    public Position last() throws EmptyContainer{
        if (n > f) return ar[n-1];
        else throw new EmptyContainer(); }

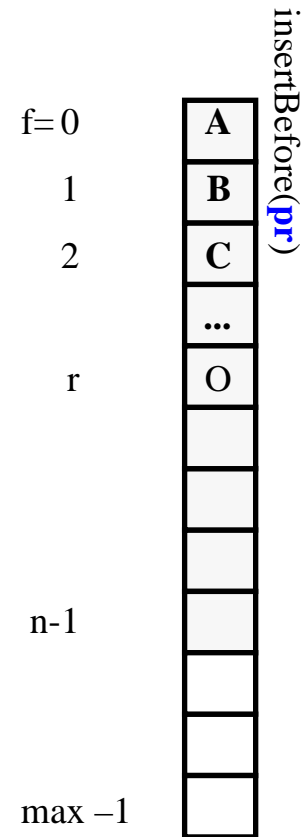
    public Position before(Position p) throws InvalidAccessorExc
    { if ( check(p).rank()==0) throw new ...Exception();
      return ar[ check(p).rank()-1 ] ; }

    public Position insertLast(Object o) {
        ar[n] = new ArPos(n,o); n++; }

    public Position insertBefore(Position pr, Object X)
        throws InvalidAccessorExc {
        int r = check(p).rank();
        for (int k= n; k>r; k--) {
            ar[k]= ar[k-1]; ar[k].setRank(k); }
        ar[r]= new ArPos(r,X);
        n++ ; return ar[r]; } }

    private ArPos check(Position p) throws InvalidAccessorExc {
        if (! p instanceof ArPos) throw new InvalidAccessorExc();
        ArPos a = (ArPos)p;
        if (p.rank() >= 0 && p.rank() <= n) return a;
        else throw new InvalidAccessorExc(); }

```



List med array (ikke sirkulær)

```

public class ListAr implements List {
    public Position last() throws EmptyContainer{
        if (n > f) return ar[n-1];
        else throw new EmptyContainer(); }

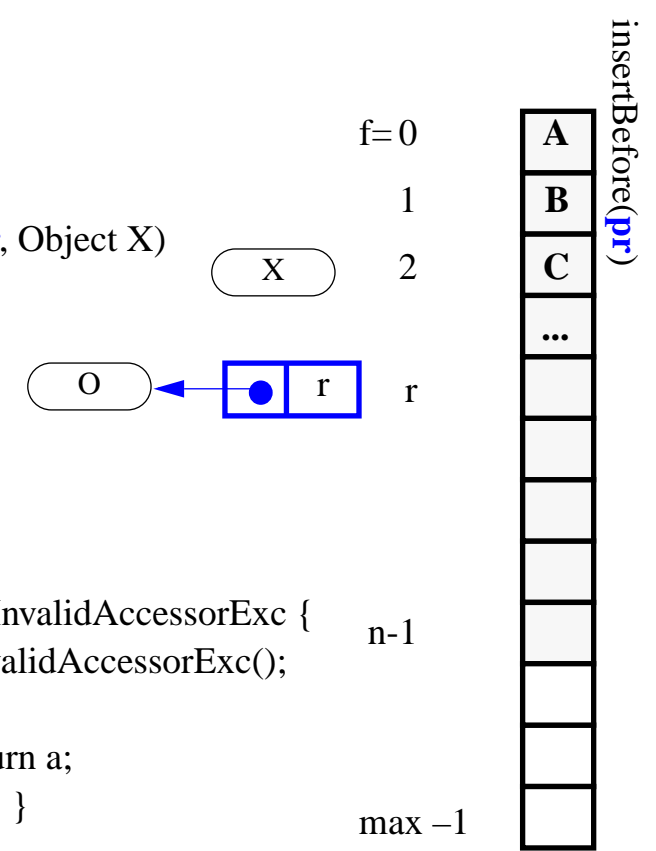
    public Position before(Position p) throws InvalidAccessorExc
    { if ( check(p).rank()==0) throw new ...Exception();
      return ar[ check(p).rank()-1 ] ; }

    public Position insertLast(Object o) {
        ar[n] = new ArPos(n,o); n++; }

    public Position insertBefore(Position pr, Object X)
        throws InvalidAccessorExc {
        int r = check(p).rank();
        for (int k= n; k>r; k--) {
            ar[k]= ar[k-1]; ar[k].setRank(k); }
        ar[r]= new ArPos(r,X);
        n++ ; return ar[r]; } }

    private ArPos check(Position p) throws InvalidAccessorExc {
        if (! p instanceof ArPos) throw new InvalidAccessorExc();
        ArPos a = (ArPos)p;
        if (p.rank() >= 0 && p.rank() <= n) return a;
        else throw new InvalidAccessorExc(); }

```



List med array (ikke sirkulær)

```

public class ListAr implements List {
    public Position last() throws EmptyContainer{
        if (n > f) return ar[n-1];
        else throw new EmptyContainer(); }

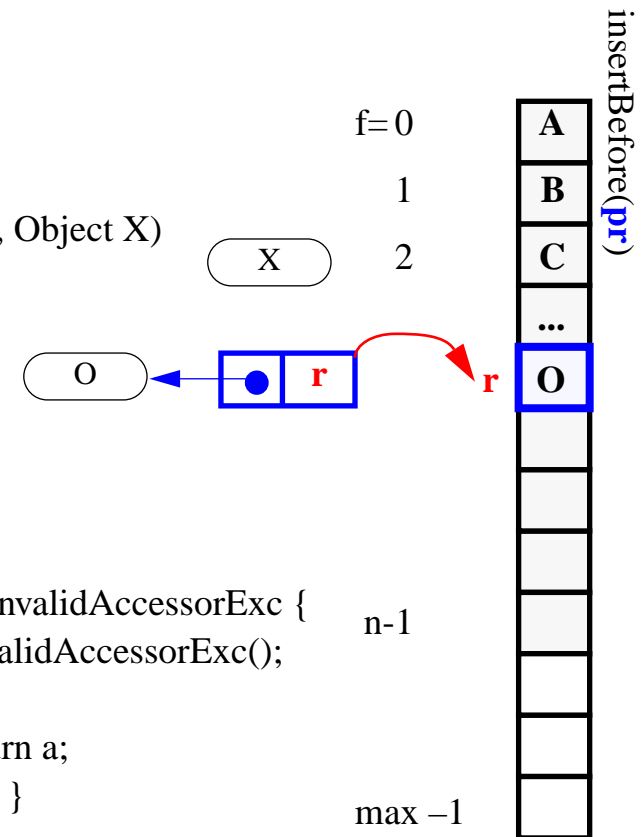
    public Position before(Position p) throws InvalidAccessorExc
    { if ( check(p).rank()==0) throw new ...Exception();
      return ar[ check(p).rank()-1 ] ; }

    public Position insertLast(Object o) {
        ar[n] = new ArPos(n,o); n++; }

    public Position insertBefore(Position pr, Object X)
        throws InvalidAccessorExc {
        int r = check(p).rank();
        for (int k= n; k>r; k--) {
            ar[k]= ar[k-1]; ar[k].setRank(k); }
        ar[r]= new ArPos(r,X);
        n++ ; return ar[r]; } }

    private ArPos check(Position p) throws InvalidAccessorExc {
        if (! p instanceof ArPos) throw new InvalidAccessorExc();
        ArPos a = (ArPos)p;
        if (p.rank() >= 0 && p.rank() <= n) return a;
        else throw new InvalidAccessorExc(); }
}

```



List med array (ikke sirkulær)

```

public class ListAr implements List {
    public Position last() throws EmptyContainer{
        if (n > f) return ar[n-1];
        else throw new EmptyContainer(); }

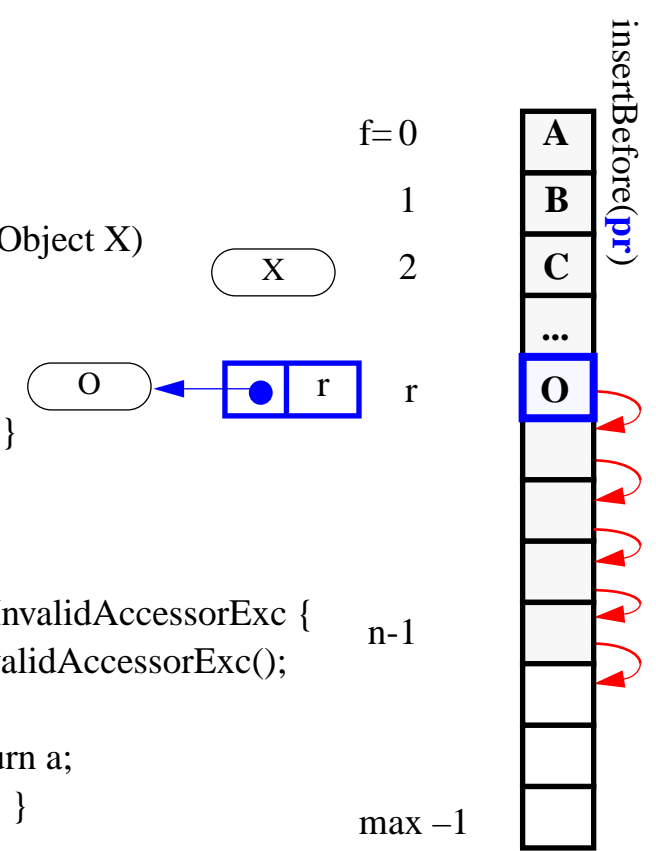
    public Position before(Position p) throws InvalidAccessorExc
    { if ( check(p).rank()==0) throw new ...Exception();
      return ar[ check(p).rank()-1 ] ; }

    public Position insertLast(Object o) {
        ar[n] = new ArPos(n,o); n++; }

    public Position insertBefore(Position p, Object X)
        throws InvalidAccessorExc {
        int r = check(p).rank();
        for (int k= n; k>r; k--) {
            ar[k]= ar[k-1]; ar[k].setRank(k); }
        ar[r]= new ArPos(r,X);
        n++ ; return ar[r]; } }

    private ArPos check(Position p) throws InvalidAccessorExc {
        if (! p instanceof ArPos) throw new InvalidAccessorExc();
        ArPos a = (ArPos)p;
        if (p.rank() >= 0 && p.rank() <= n) return a;
        else throw new InvalidAccessorExc(); }

```



List med array (ikke sirkulær)

```

public class ListAr implements List {
    public Position last() throws EmptyContainer{
        if (n > f) return ar[n-1];
        else throw new EmptyContainer(); }

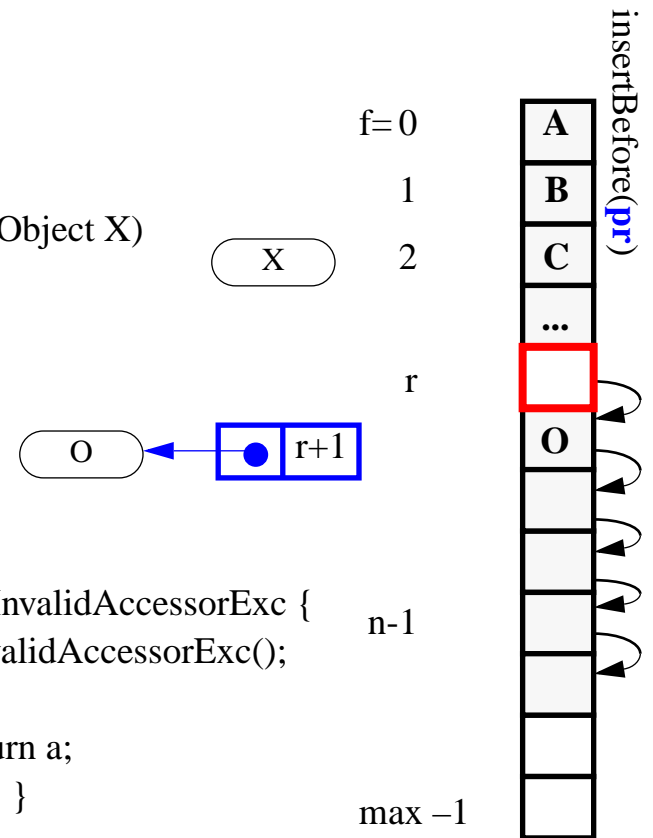
    public Position before(Position p) throws InvalidAccessorExc
    { if ( check(p).rank()==0) throw new ...Exception();
      return ar[ check(p).rank()-1 ] ; }

    public Position insertLast(Object o) {
        ar[n] = new ArPos(n,o); n++; }

    public Position insertBefore(Position p, Object X)
        throws InvalidAccessorExc {
        int r = check(p).rank();
        for (int k= n; k>r; k--) {
            ar[k]= ar[k-1]; ar[k].setRank(k); }
        ar[r]= new ArPos(r,X);
        n++ ; return ar[r]; } }

    private ArPos check(Position p) throws InvalidAccessorExc {
        if (! p instanceof ArPos) throw new InvalidAccessorExc();
        ArPos a = (ArPos)p;
        if (p.rank() >= 0 && p.rank() <= n) return a;
        else throw new InvalidAccessorExc(); }

```



List med array (ikke sirkulær)

```

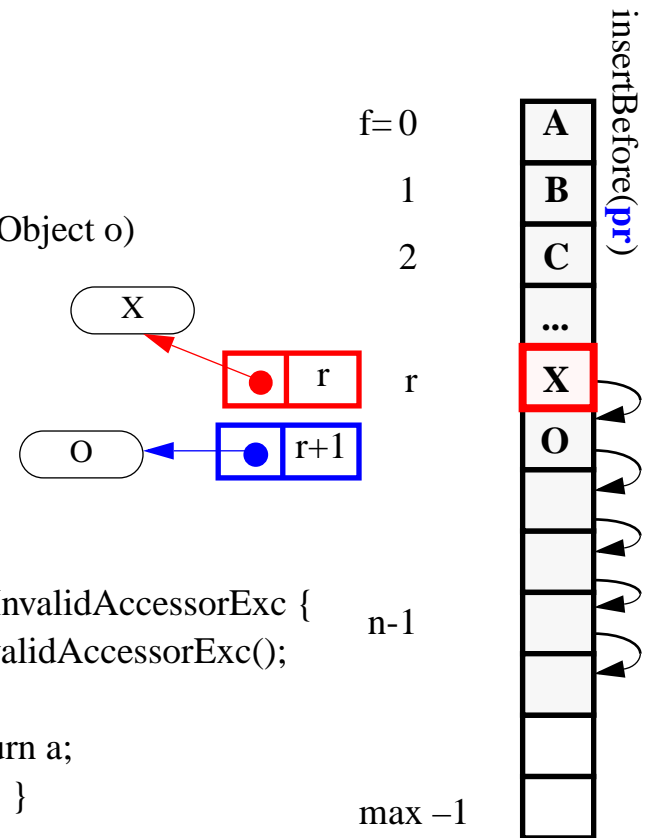
public class ListAr implements List {
    public Position last() throws EmptyContainer{
        if (n > f) return ar[n-1];
        else throw new EmptyContainer(); }

    public Position before(Position p) throws InvalidAccessorExc
    { if ( check(p).rank()==0) throw new ...Exception();
      return ar[ check(p).rank()-1 ] ; }

    public Position insertLast(Object o) {
        ar[n] = new ArPos(n,o); n++; }

    public Position insertBefore(Position p, Object o)
        throws InvalidAccessorExc {
        int r = check(p).rank();
        for (int k= n; k>r; k--) {
            ar[k]= ar[k-1]; ar[k].setRank(k); }
        ar[r]= new ArPos(r,X)
        n++ ; return ar[r]; } }

    private ArPos check(Position p) throws InvalidAccessorExc {
        if (! p instanceof ArPos) throw new InvalidAccessorExc();
        ArPos a = (ArPos)p;
        if (p.rank() >= 0 && p.rank() <= n) return a;
        else throw new InvalidAccessorExc(); }
    
```



List med array (ikke sirkulær)

```

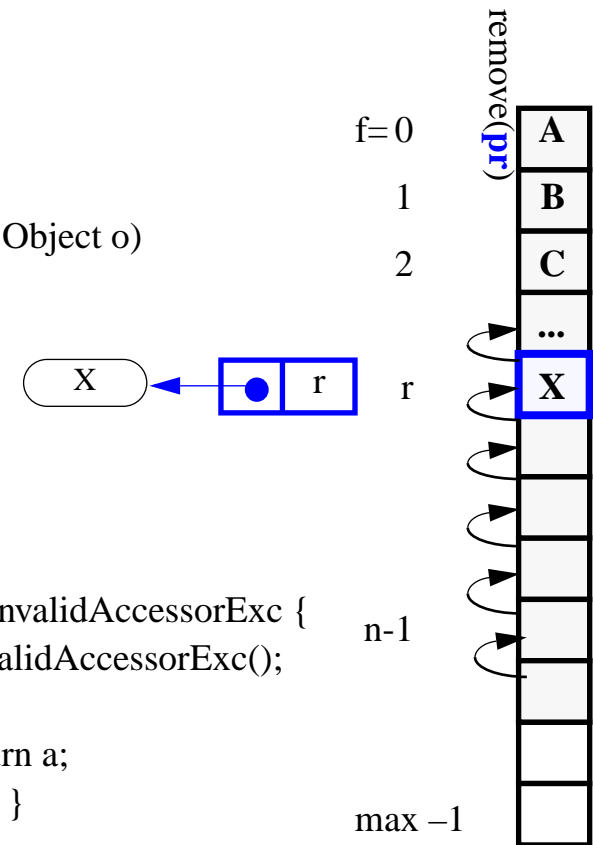
public class ListAr implements List {
    public Position last() throws EmptyContainer{
        if (n > f) return ar[n-1];
        else throw new EmptyContainer(); }

    public Position before(Position p) throws InvalidAccessorExc
    { if ( check(p).rank()==0) throw new ...Exception();
      return ar[ check(p).rank()-1 ] ; }

    public Position insertLast(Object o) {
        ar[n] = new ArPos(n,o); n++; }

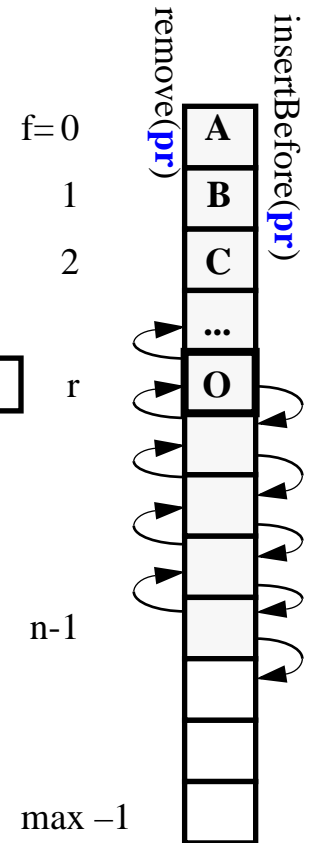
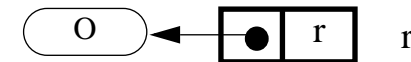
    public Position insertBefore(Position pr, Object o)
        throws InvalidAccessorExc {
        int r = check(p).rank();
        for (int k= n; k>r; k--) {
            ar[k]= ar[k-1]; ar[k].setRank(k); }
        ar[r]= new ArPos(r,o);
        n++ ; return ar[r]; } }

    private ArPos check(Position p) throws InvalidAccessorExc {
        if (! p instanceof ArPos) throw new InvalidAccessorExc();
        ArPos a = (ArPos)p;
        if (p.rank() >= 0 && p.rank() <= n) return a;
        else throw new InvalidAccessorExc(); }
    
```



List med array (ikke sirkulær)

		public class <i>ListAr</i> implements List {
		public Position last() throws EmptyContainer{
first()	ArPos 1	if (n > f) return ar[n-1];
		else throw new EmptyContainer(); }
last()	1	public Position before (Position p) throws InvalidAccessorExc
		{ if (check(p).rank()==0) throw new ...Exception();
after(p)	1	return ar[check(p).rank()-1] ; }
before(p)	1	public Position insertLast (Object o) {
		ar[n] = new ArPos(n,o); n++; }
insertFirst(o)	n/1	public Position insertBefore (Position p, Object o)
		throws InvalidAccessorExc {
insertLast(o)	1	int r = check(p).rank();
insertAfter(p,o)	n	for (int k= n; k>r; k--) {
		ar[k]= ar[k-1]; ar[k].setRank(k); }
insertBefore(p,o)	n	ar[r]= new ArPos(r,o);
		n++ ; return ar[r]; } }
remove(p)	n	private ArPos check (Position p) throws InvalidAccessorExc {
		if (! p instanceof ArPos) throw new InvalidAccessorExc();
		ArPos a = (ArPos)p;
		if (p.rank() >= 0 && p.rank() <= n) return a;
		else throw new InvalidAccessorExc(); }



Implementasjoner av List

kompleksitet operasjon	Array	en-veis liste	to-veis liste
size, isEmpty	1	1	1
newContainer	1	1	1
elements, positions	n	n	n
replace, swap	1	1	1
first	1	1	1
last	1	n/1 (sist peker)	1
before	1	n	1
after	1	1	1
insertFirst	n/1(sirkulær)	1	1
insertLast	1	n/1 (sist peker)	1
insertBefore	n	n	1
insertAfter	n	1	1
remove	n	n	1

Sequence ADT

size, isEmpty
elements, positions
newContainer
replace, swap

first, last
before, after
insertFirst/-Last
insertBefore/-After
remove

elemAtRank
replaceAtRank
insertAtRank
removeAtRank

List

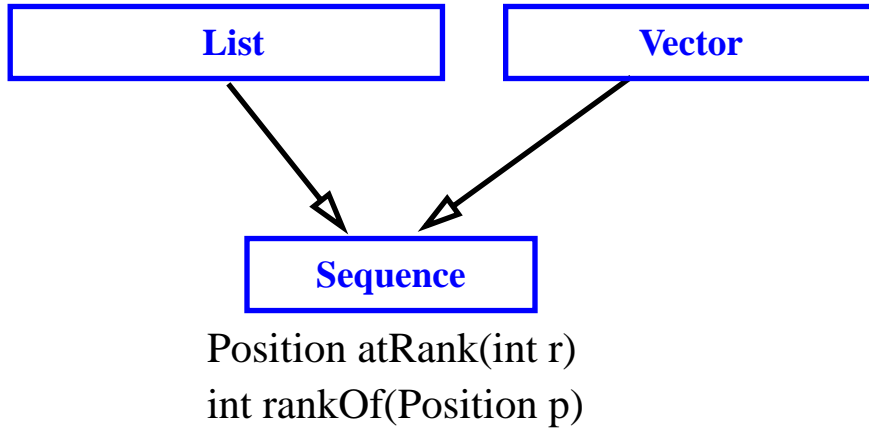
Vector

Sequence ADT

size, isEmpty
elements, positions
newContainer
replace, swap

first, last
before, after
insertFirst/-Last
insertBefore/-After
remove

elemAtRank
replaceAtRank
insertAtRank
removeAtRank

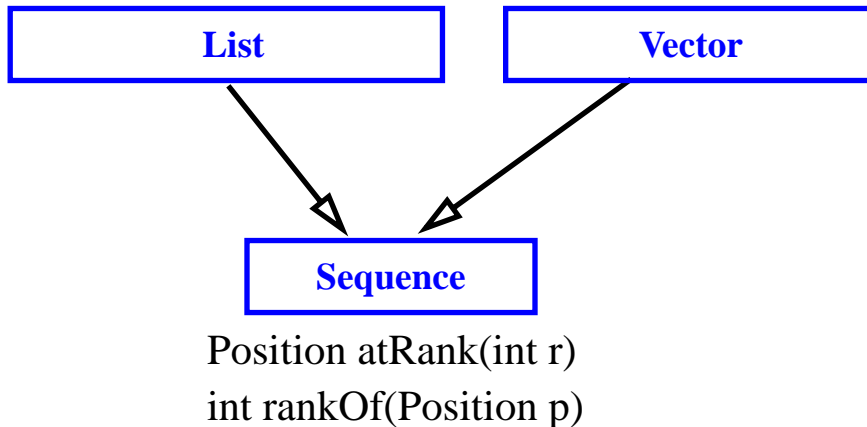


Sequence ADT

size, isEmpty
elements, positions
newContainer
replace, swap

first, last
before, after
insertFirst/-Last
insertBefore/-After
remove

elemAtRank
replaceAtRank
insertAtRank
removeAtRank



```
public interface Sequence
    extends List, Vector {
```

```
/** @param r rank
    @return posisjonen til elementet med rank r
    @exception BoundaryViolationException
    Position atRank(int r)
```

```
/** @param p posisjon
    @return rank til elementet i posisjonen p
    @exception InvalidAccessorException
    int rankOf(Position p)
}
```

Implementasjoner av Sequence

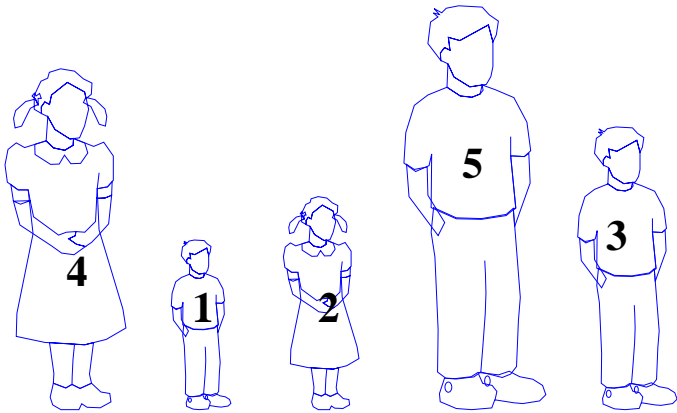
	kompleksitet operasjon	Array	en-veis liste	to-veis liste
Container	size, isEmpty, newContainer	1	1	1
PositionalContainer	elements, positions	n	n	n
	replace, swap	1	1	1
List	first	1	1	1
	last	1	n/1	1
	before	1	n	1
	after	1	1	1
	insertFirst	n/1 (sirkulær)	1	1
	insertLast	1	n/1	1
	insertBefore	n	n	1
	insertAfter	n	1	1
	remove	n	n	1
Vector	elemAtRank, replaceAtRank	1	(n)	(n)
	insertAtRank, removeAtRank	n	(n)	(n)
Sequence	rankOf	1	1	1
	atRank	1	n	n

Sortering

- utføres på Elementer som kan sammenliknes:

total ordning: for alle Elementer a, b :

$a < b$ **eller** $a > b$ **eller** $a == b$

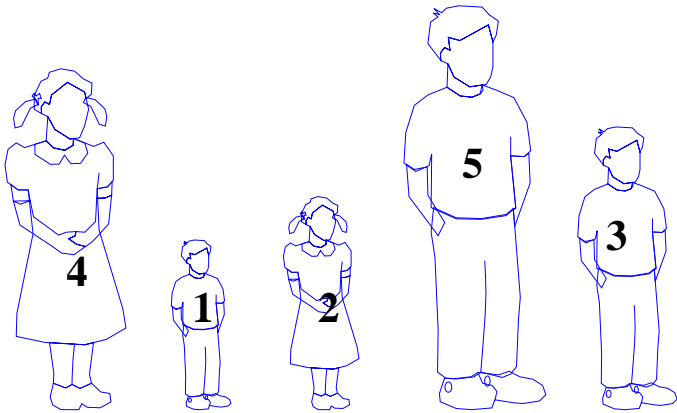


Sortering

- utføres på Elementer som kan sammenliknes:

total ordning: for alle Elementer a, b :

a < b eller a > b eller a == b



public interface Comparator

```
{    /** @return true iff a < b; false otherwise */
    boolean isLessThan (Object a, Object b)
        /** @return true iff a > b; false otherwise */
    boolean isGreaterThan (Object a, Object b)
        /** @return true iff a == b; false otherwise */
    boolean isEqualTo (Object a, Object b)
        /** @return true iff a <= b; false otherwise */
    boolean isLessThanOrEqualTo (Object a, Object b)
        /** @return true iff a >= b; false otherwise */
    boolean isGreaterThanOrEqualTo (Object a, Object b)
}
```

Sortering

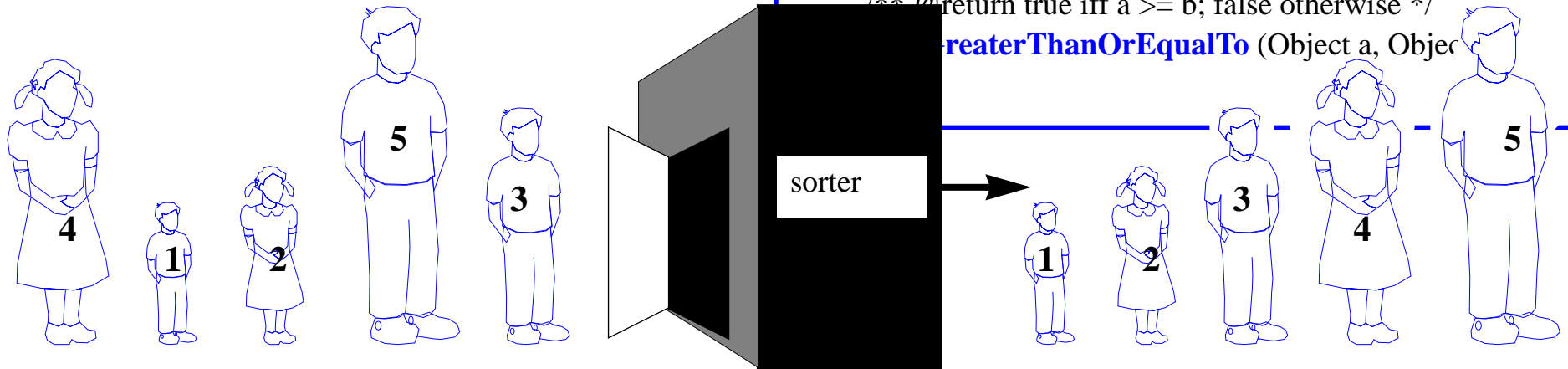
- utføres på Elementer som kan sammenliknes:

total ordning: for alle Elementer a, b :

$a < b$ **eller** $a > b$ **eller** $a == b$

public interface Comparator

```
{    /** @return true iff a < b; false otherwise */  
    boolean isLessThan (Object a, Object b)  
        /** @return true iff a > b; false otherwise */  
    boolean isGreaterThan (Object a, Object b)  
        /** @return true iff a == b; false otherwise */  
    boolean isEqualTo (Object a, Object b)  
        /** @return true iff a <= b; false otherwise */  
    boolean isLessThanOrEqualTo (Object a, Object b)  
        /** @return true iff a >= b; false otherwise */  
    boolean isGreaterThanOrEqualTo (Object a, Object b)
```



Sortering

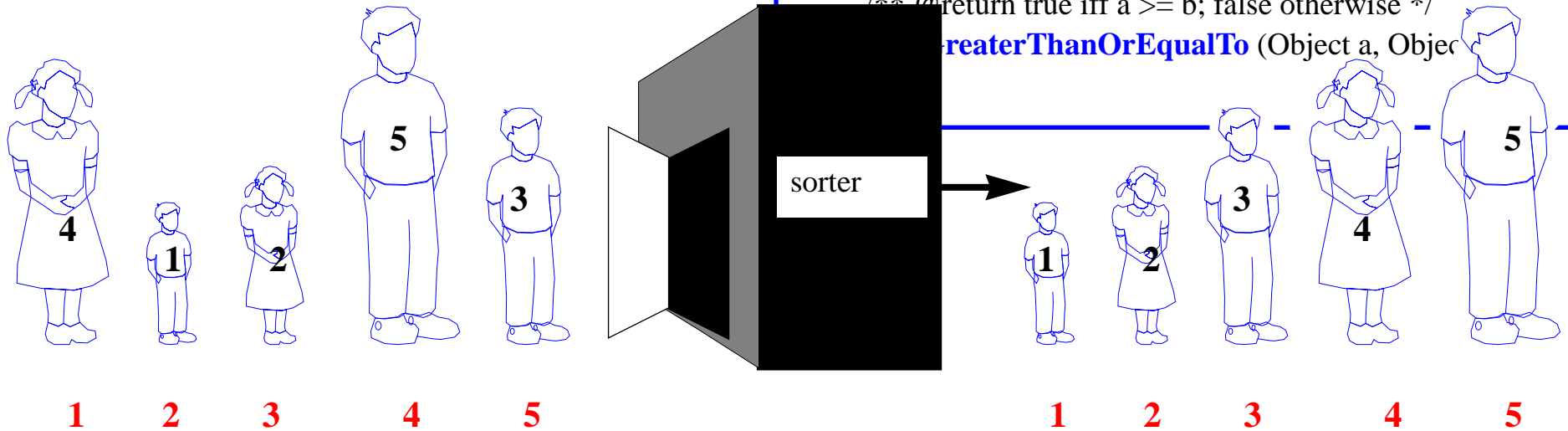
- utføres på Elementer som kan sammenliknes:

total ordning: for alle Elementer a, b :

$a < b$ **eller** $a > b$ **eller** $a == b$

public interface Comparator

```
{    /** @return true iff a < b; false otherwise */  
    boolean isLessThan (Object a, Object b)  
        /** @return true iff a > b; false otherwise */  
    boolean isGreaterThan (Object a, Object b)  
        /** @return true iff a == b; false otherwise */  
    boolean isEqualTo (Object a, Object b)  
        /** @return true iff a <= b; false otherwise */  
    boolean isLessThanOrEqualTo (Object a, Object b)  
        /** @return true iff a >= b; false otherwise */  
    boolean isGreaterThanOrEqualTo (Object a, Object b)
```

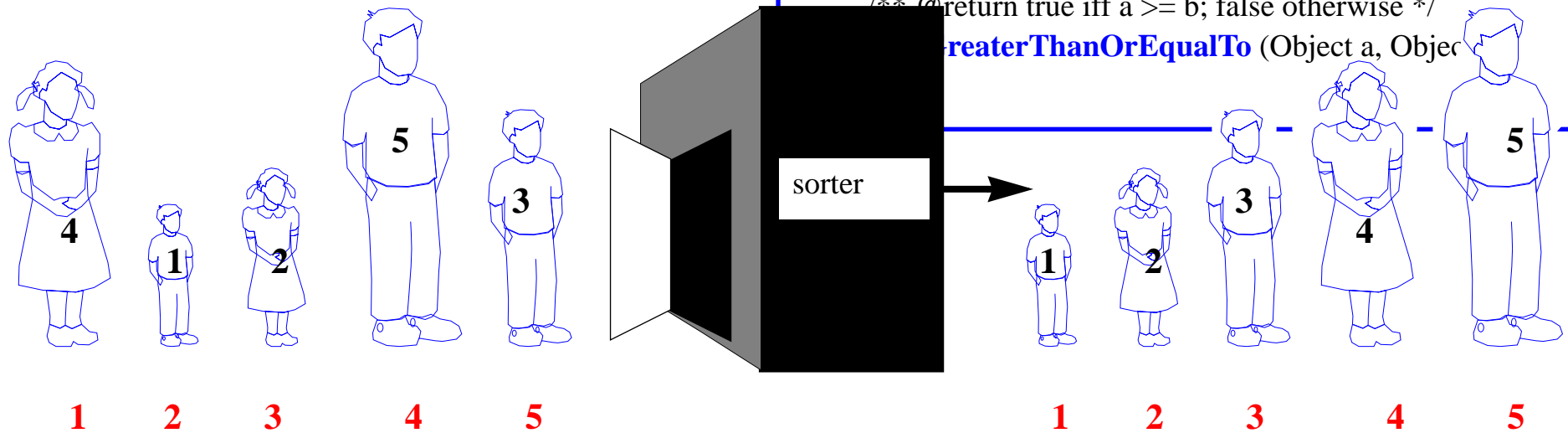


Sortering

- utføres på Elementer som kan sammenliknes:

total ordning: for alle Elementer a, b :

$a < b$ eller $a > b$ eller $a == b$



- som står samlet i en struktur der Posisjoner er ordnet:

total ordning: for alle Posisjoner p, q :

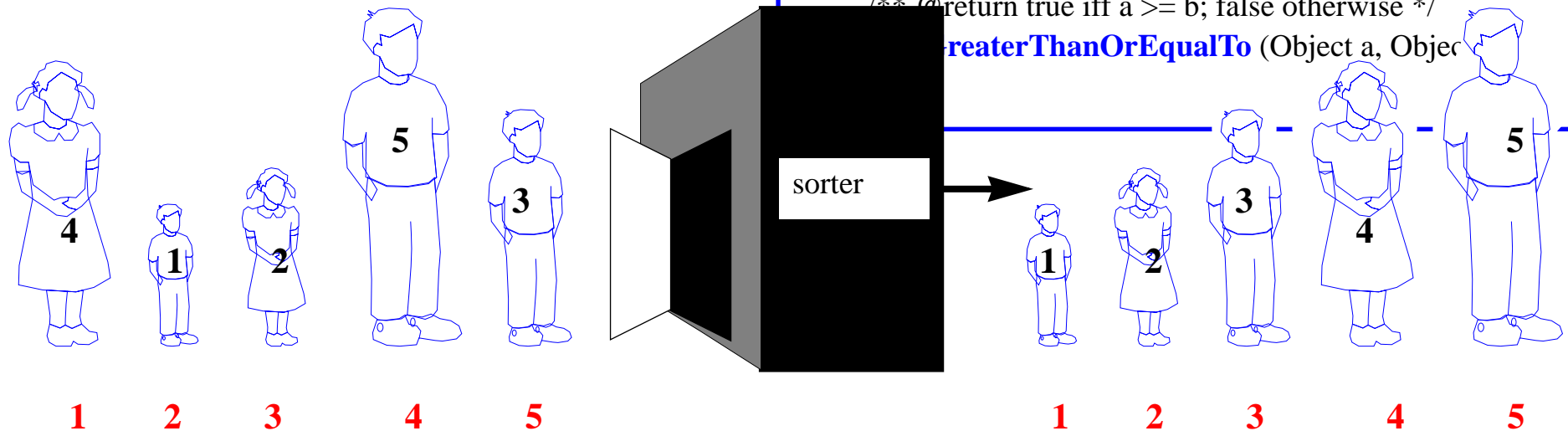
$p < q$ eller $p > q$ eller $p == q$

Sortering

- utføres på Elementer som kan sammenliknes:

total ordning: for alle Elementer a, b :

$a < b$ eller $a > b$ eller $a == b$



- som står samlet i en struktur der Posisjoner er ordnet:

total ordning: for alle Posisjoner p, q :

$p < q$ eller $p > q$ eller $p == q$

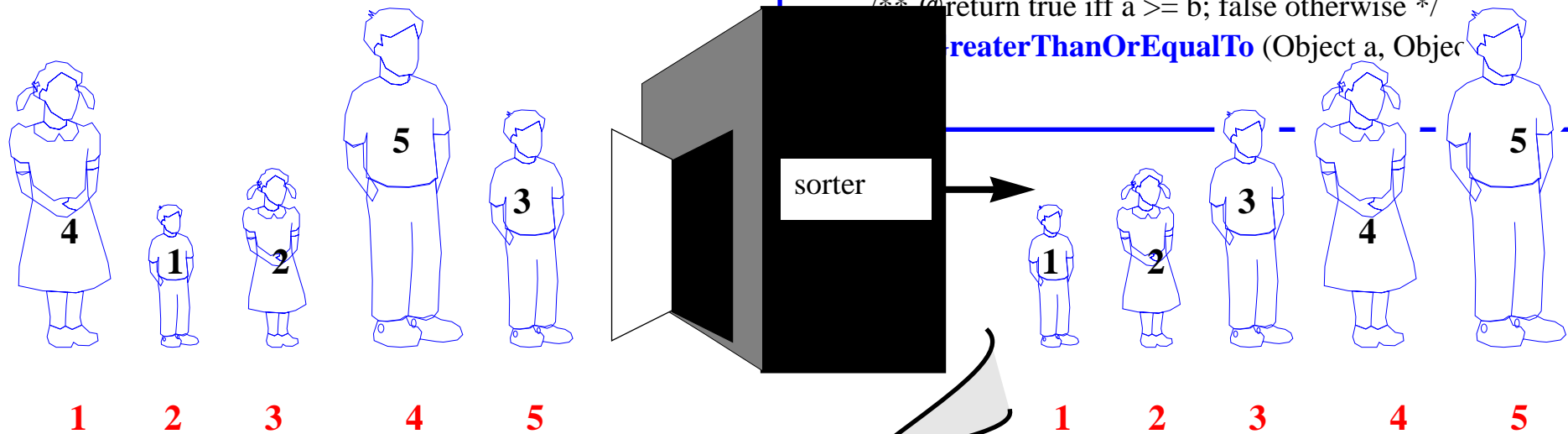
*I List ADT, uttrykkes denne ordningen med operasjoner **before/after**:*

Sortering

- utføres på Elementer som kan sammenliknes:

total ordning: for alle Elementer a, b :

$a < b$ **eller** $a > b$ **eller** $a == b$



- som står samlet i en struktur der Posisjoner er ordnet:

total ordning: for alle Posisjoner p, q :

$p < q$ **eller** $p > q$ **eller** $p == q$

public interface Comparator

```
{    /** @return true iff a < b; false otherwise */  
    boolean isLessThan (Object a, Object b)  
        /** @return true iff a > b; false otherwise */  
    boolean isGreaterThan (Object a, Object b)  
        /** @return true iff a == b; false otherwise */  
    boolean isEqualTo (Object a, Object b)  
        /** @return true iff a <= b; false otherwise */  
    boolean isLessThanOrEqualTo (Object a, Object b)  
        /** @return true iff a >= b; false otherwise */  
    boolean isGreaterThanOrEqualTo (Object a, Object b)
```

*I List ADT, uttrykkes denne ordningen med operasjoner **before/after**:*

p **before** q \rightarrow `p.element()`
`isLessThanOrEqualTo` `q.element()`

Generisk seleksjon-sort av Sequence

```
for (k=0,1,2...< n) { // n= S.size()
  int m = k;
  for (j= k+1...< n)
    if (S[j] < S[m]) m = j;
  S.swap( m, k )
}
```

$$1+2+\dots+(n-1)+n = O(n^2)$$

*en algoritme er generisk hviss alle
klasse-parametre er ADTer : interface
eller Object*

```
void SS1(Object[] S, Comparator C) {
```

Generisk seleksjon-sort av Sequence

```
for (k=0,1,2...< n) { // n= S.size()
  int m = k;
  for (j= k+1...< n)
    if (S[j] < S[m]) m = j;
  S.swap( m, k )
}
```

$$1+2+\dots+(n-1)+n = \mathcal{O}(n^2)$$

*en algoritme er generisk hviss alle
klasse-parametre er ADTer : interface
eller Object*

```
void SS1(Sequence S, Comparator C) {
```

Generisk seleksjon-sort av Sequence

```
for (k=0,1,2...< n) { // n= S.size()
    int m = k;
    for (j= k+1...< n)
        if (S[j] < S[m]) m = j;
    S.swap( m, k )
}
```

$$1+2+\dots+(n-1)+n = O(n^2)$$

en algoritme er generisk hvis alle klasse-parametre er ADTer : interface eller Object

```
void SS1(Sequence S, Comparator C) {
    int k, j, min;
    int n = S.size() ;
    for (k=0; k<n; k++) {
        min = k ;
        em = S.elemAtRank(min);
        for (j= k+1; j<n; j++)
            if (C.isLessThan(
                S.elemAtRank(j), em ))
                { min = j;
                  em = S.elemAtRank(min); }
        S.swap( S.atRank(min), S.atRank(k) );
    }
}
```

Generisk seleksjon-sort av Sequence

```
for (k=0,1,2...< n) { // n= S.size()
    int m = k;
    for (j= k+1...< n)
        if (S[j] < S[m]) m = j;
    S.swap( m, k )
}
```

$$1+2+\dots+(n-1)+n = O(n^2)$$

en algoritme er generisk hviss alle klasse-parametre er ADTer : interface eller Object

```
void SS1(Sequence S, Comparator C) {
    int k, j, min;
    int n = S.size() ;
    for (k=0; k<n; k++) {
        min = k ;
        em = S.elemAtRank(min);
        for (j= k+1; j<n; j++)
            if (C.isLessThan(
                S.elemAtRank(j), em ))
                { min = j;
                  em = S.elemAtRank(min); }
        S.swap( S.atRank(min), S.atRank(k) );
    }
}
```

kan brukes kun når **atRank(r)** er $O(1)$;

er den $O(n)$ får vi $SS1 = O(n^3)$!!!

Generisk seleksjon-sort av Sequence

```
for (k=0,1,2...< n) { // n= S.size()
  int m = k;
  for (j= k+1...< n)
    if (S[j] < S[m]) m = j;
  S.swap( m, k )
}
```

$$1+2+\dots+(n-1)+n = O(n^2)$$

en algoritme er generisk hvis alle klasse-parametre er ADTer : interface eller Object

```
void SS1(Sequence S, Comparator C) {
  int k, j, min;
  int n = S.size() ;
  for (k=0; k<n; k++) {
    min = k ;
    em = S.elemAtRank(min);
    for (j= k+1; j<n; j++)
      if (C.isLessThan(
        S.elemAtRank(j), em ))
        { min = j;
          em = S.elemAtRank(min); }
    S.swap( S.atRank(min), S.atRank(k) );
  }
}
```

kan brukes kun når **atRank(r)** er $O(1)$;

er den $O(n)$ får vi $SS1 = O(n^3)$!!!

```
void SS2(Sequence S, Comparator C)
  Position k, j, min;
  k= S.first();
  while ( k != S.last() ) {
    min = k; j = k;
    while ( j != S.last() ) {
      j = S.after(j);
      if (C.isLessThan(j.element(), m.element()))
        min = j;
    }
    S.swap(min, k) ;
    k = S.after(k) ;
  } }
```

first(), **last(r)**, **after(p)** kan alltid fåes $O(1)$;

$SS2 = O(n^2)$!!!

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                     S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                         S[0...n-1] er sortert
                        //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k=5 5 4 2 1 3

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 5 2 1 3

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 5 1 3

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```



Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

Generisk **boble-sortering** av Sequence

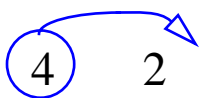
```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 4  2 1 3 5

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S


    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 4 2 4 1 3 5



Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

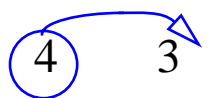
    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 4 2 1 4 3 5



Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 4 2 1 3 4 5

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 3 2 1 3 4 5

k= 2 2 1 3 4 5



Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 3 2 1 3 4 5

k= 2 1 (2) 3 4 5

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 3 2 1 3 4 5

k= 2 1 2 3 4 5

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 3 2 1 3 4 5

k= 3 1 2 3 4 5

k= 2 ① 2 3 4 5

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 3 2 1 3 4 5

k= 3 1 2 3 4 5

k= 2 1 2 3 4 5

Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                     har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }                   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}                       S[0...n-1] er sortert
                       //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                   n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }                   S[k] er størst i S[0...k]
}                       S[1...n] er sortert
                       // og har n største elem. fra S
```

k= 5 4 2 1 3 5

k= 3 2 1 3 4 5

k= 3 1 2 3 4 5

k= 2 1 2 3 4 5

k= 1 ① 2 3 4 5

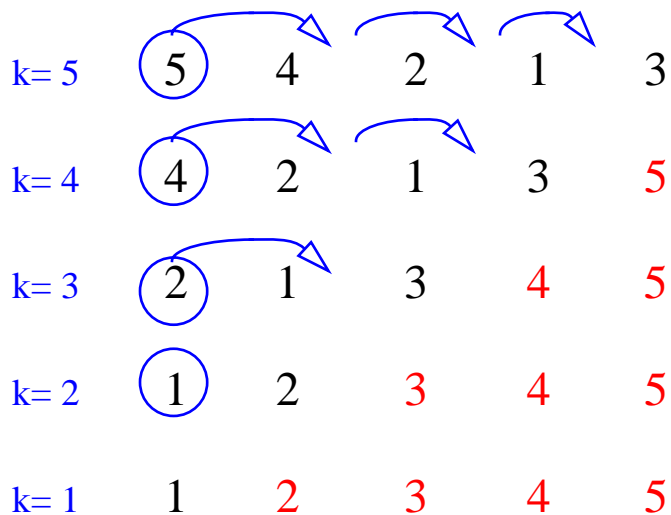
Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                      har k minste elem. fra S

  int m = k;
  for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
    if (S[j] < S[m])
      m = j;
  }   S[m] er minst i S[k...n-1]
  S.swap( m, k )
}   S[0...n-1] er sortert
    //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                    n-k største elem. fra S

  for (j= 0...<k) {   S[j] er størst i S[0...j]
    if (S[j+1] < S[j])
      S.swap( j, j+1 )
  }   S[k] er størst i S[0...k]
}   S[1...n] er sortert
    og har n største elem. fra S
```



```
void BubbleSort(Sequence S, Comparator C) {
  Position p1, p2 ;
  for ( k = S.size()-1; k>0; k--) {
    p1 = S.first() ;
    for ( j = 0; j<k; j++) {
      p2 = S.after(p);
      if (C.isLessThan(p2.element(), p1.element()))
        S.swap(p1,p2);
      p = a;
    } } }
}
```

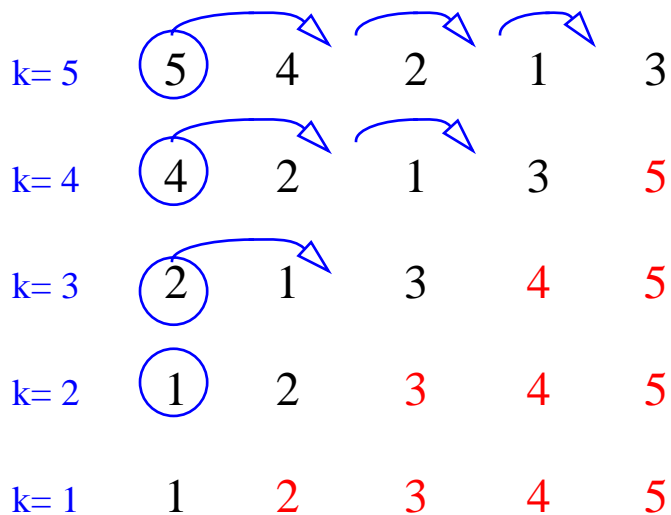
Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                      har k minste elem. fra S

    int m = k;
    for (j= k+1...< n) {   S[m] er minst i S[k...j-1]
        if (S[j] < S[m])
            m = j;
    }   S[m] er minst i S[k...n-1]
    S.swap( m, k )
}   S[0...n-1] er sortert
    //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                    n-k største elem. fra S

    for (j= 0...<k) {   S[j] er størst i S[0...j]
        if (S[j+1] < S[j])
            S.swap( j, j+1 )
    }   S[k] er størst i S[0...k]
}   S[1...n] er sortert
    og har n største elem. fra S
```



```
void BubbleSort(Sequence S, Comparator C) {
    Position p1, p2 ;
    for ( k = S.size()-1; k>0; k--) {
        p1 = S.first() ;
        for ( j = 0; j<k; j++) {
            p2 = S.after(p);
            if (C.isLessThan(p2.element(), p1.element()))
                S.swap(p1,p2);
            p = a;
        }
    }
}
```

$O(n^2)$

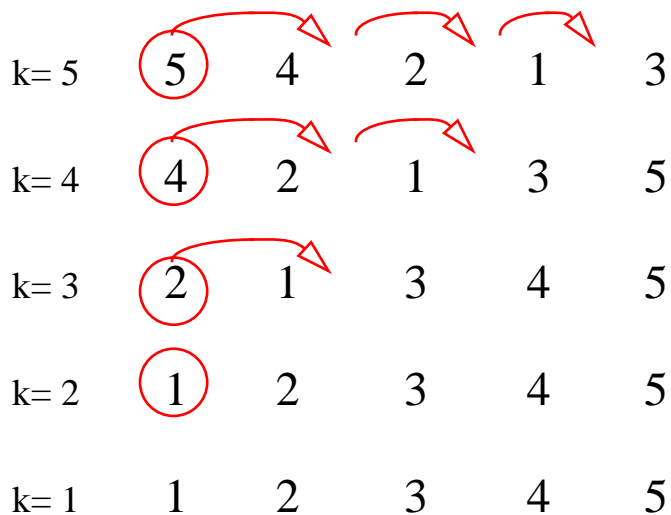
Generisk **boble-sortering** av Sequence

```
// n= S.size()
for (k=0,1...< n) {   S[0...k-1] er sortert og
                      har k minste elem. fra S

  int m = k;
  for (j= k+1...< n) {  S[m] er minst i S[k...j-1]
    if (S[j] < S[m])
      m = j;
  }                    S[m] er minst i S[k...n-1]
  S.swap( m, k )
}                      S[0...n-1] er sortert
                      //og har n minste elem. fra S
```

```
// n= S.size() - 1
for (k=n...> 0) {   S[k+1...n] er sortert og har
                    n-k største elem. fra S

  for (j= 0...<k) {  S[j] er størst i S[0...j]
    if (S[j+1] < S[j])
      S.swap(j, j+1 )
  }                  S[k] er størst i S[0...k]
}                    S[1...n] er sortert
                    og har n største elem. fra S
```



```
void BubbleSort(Sequence S, Comparator C) {
  Position p1, p2 ;
  for ( k = S.size()-1; k>0; k--) {
    p1 = S.first() ;
    for ( j = 0; j<k; j++) {
      p2 = S.after(p);
      if (C.isLessThan(p2.element(), p1.element()))
        S.swap(p1,p2);
      p = a;
    }
  }
}
```

$O(n^2)$

Korrekthet av boble-sortering

BubbleSort(int[] A) { // n er største indeks i tabellen

INN1: $k=n : A[n+1..n] \dots 0 \leq r < s \leq n \ \& \ s > n \dots$ ingen slik s

1: **for** ($k = n, k > 0, k--$) {

LI1: $0 \leq r < s \leq n \ \& \ s > k \rightarrow A[r] \leq A[s]$

dvs. $A[k+1..n]$ er sortert og har $n-k$ største elementer

INN2: $j=0 : A[0]$ er størst i $A[0..0]$

2: **for** ($j = 0, j < k, j++$) {

LI2: $0 \leq t < j \leq n \rightarrow A[t] \leq A[j]$

dvs. $A[j]$ er størst i $A[0..j]$

if ($A[j] > A[j+1]$) **swap**($A[j], A[j+1]$);

$j'=j+1$: er $A[j']$ størst i $A[0..j']$?

hvis $A[j] \leq A[j+1] \ \& \ LI2 \rightarrow LI2'$

ellers $A[j'] = A[j] > A[j+1] = A[j'-1] \ \& \ LI2 \rightarrow LI2'$

LI2': $j'=j+1: 0 \leq t < j' \leq n \rightarrow A[t] \leq A[j']$

}

UTG2: $LI2 \ \& \ j=k$ gir: $0 \leq t < k \leq n \rightarrow A[t] \leq A[k]$

dvs. $A[k]$ er størst bland $A[0..k]$

UTG2 & LI1 gir: $A[0..k-1] \leq A[k] \leq A[k+1..n] \rightarrow$

LI1': $k'=k-1: 1 \leq r < s \leq n, s > k' \rightarrow A[r] \leq A[s]$

dvs. $A[k'..n]$ er sortert og har $n-k'$ største elementer

}

UTG1: **LI1 & $k=0$** gir $0 \leq r < s \leq n \ \& \ s > 0 \rightarrow A[r] \leq A[s]$

dvs. : $A[1..n]$ er sortert og har $n-1$ største elementer,

men da har $A[0]$ minste element $\leq A[r]$ for $0 < r \leq n$

(spesielt, for $r=1 : A[0] \leq A[1]$)

Oppsummering

- *Container, PositionalContainer, Position, List, Sequence,*
- *organisering av ADTer*
 - *hierarki av interface burde avspeile alle relasjoner mellom begrepene (ingen entydig arrangement)*
- *riktige abstraksjoner kan være vanskelig å finne*
 - *Position (Locator, Accessor)*

Oppsummering

- *Container, PositionalContainer, Position, List, Sequence,*
- *organisering av ADTer*
 - *hierarki av interface burde avspeile alle relasjoner mellom begrepene (ingen entydig arrangement)*
- *riktige abstraksjoner kan være vanskelig å finne*
 - *Position (Locator, Accessor)*
- *ADT-2 kan brukes for en “generisk” implementasjon av en annen ADT-1*
 - *relativ kompleksitet, dvs kompleksitet til ADT-1 avhengig av implementasjon av ADT-2*
- *forskjellige implementasjoner av samme ADT*
 - *vurdering av implementasjoner opp mot hverandre*

Oppsummering

- *Container, PositionalContainer, Position, List, Sequence,*
- *organisering av ADTer*
 - *hierarki av interface burde avspeile alle relasjoner mellom begrepene (ingen entydig arrangement)*
- *riktige abstraksjoner kan være vanskelig å finne*
 - *Position (Locator, Accessor)*
- *ADT-2 kan brukes for en “generisk” implementasjon av en annen ADT-1*
 - *relativ kompleksitet, dvs kompleksitet til ADT-1 avhengig av implementasjon av ADT-2*
- *forskjellige implementasjoner av samme ADT*
 - *vurdering av implementasjoner opp mot hverandre*
- *“generisk” algoritme bruker kun ADTer*
 - *grensesnitt informasjon om parametre*
 - *og derfor kan brukes med vilkårlige implementasjoner*

Oppsummering

- *Container, PositionalContainer, Position, List, Sequence,*
- *organisering av ADTer*
 - *hierarki av interface burde avspeile alle relasjoner mellom begrepene (ingen entydig arrangement)*
- *riktige abstraksjoner kan være vanskelig å finne*
 - *Position (Locator, Accessor)*
- *ADT-2 kan brukes for en “generisk” implementasjon av en annen ADT-1*
 - *relativ kompleksitet, dvs kompleksitet til ADT-1 avhengig av implementasjon av ADT-2*
- *forskjellige implementasjoner av samme ADT*
 - *vurdering av implementasjoner opp mot hverandre*
- *“generisk” algoritme bruker kun ADTer*
 - *grensesnitt informasjon om parametre*
 - *og derfor kan brukes med vilkårlige implementasjoner*
- *sorterings algoritmer*
 - *seleksjon-, merge-, bobble-sortering*