Stabler, Køer og Lister

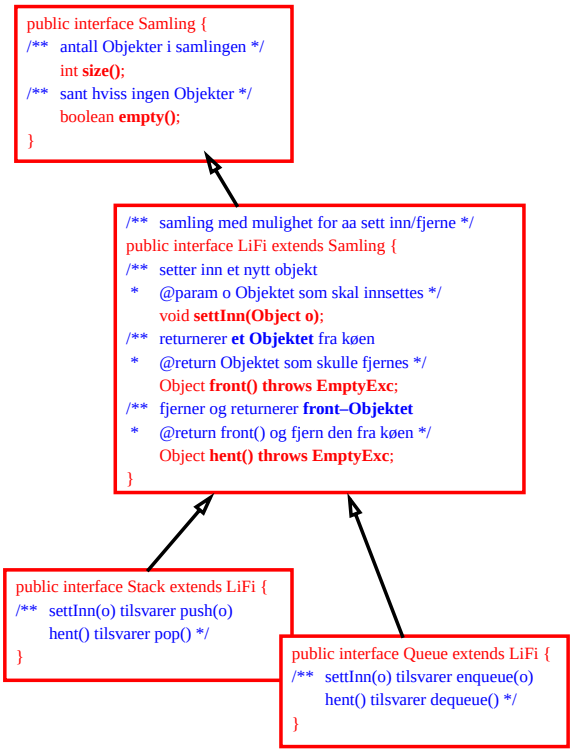
Kap. 3 (kursorisk: 3.1.3, 3.2.3, 3.4; unntatt: 3.2.4, 3.5)

ADT'er

```
/**  LIFO kø av vilkårlige Objekter
 *   S.push(o).peek() == o ;  S.push(o).pop() == o & S  */
public interface Stack {
/**  setter inn et nytt objekt på toppen
 *   @param o Objektet som skal innettes */
     void push(Object o);
/**  returnerer toppen av stabel
 *   @return Objektet som ble innsatt sist
 *   @exception EmptyExc hvis stabel er tom*/
     Object peek() throws EmptyExc;
/**  fjerner og returnerer toppen
 *   @return peek() og fjern den fra stabel
 *   @exception EmptyExc hvis stabel er tom*/
     Object pop() throws EmptyExc;
/**  sjekker om stabel er tom */
     boolean empty();
}
```

```
/**  FIFO kø av vilkårlige Objekter
 *   S.enqueue(o).front() = S.front() hvis !S.empty();ellers
 *   S.enqueue(o).dequeue()|Q = S.dequeue()|Q.enqueue(o) */
public interface Queue {
/**  setter inn et nytt objekt på slutten
 *   @param o Objektet som skal innettes */
     void enqueue(Object o);
/**  returnerer første Objektet i køen
 *   @return Objektet som har vært i køen lengst
 *   @exception EmptyExc hvis køen er tom*/
     Object front() throws EmptyExc;
/**  fjerner og returnerer første Objektet
 *   @return front() og fjern den fra køen
 *   @exception EmptyExc hvis køen er tom*/
     Object dequeue() throws EmptyExc;
/**  sjekker om køen er tom */
     boolean empty();          }
```

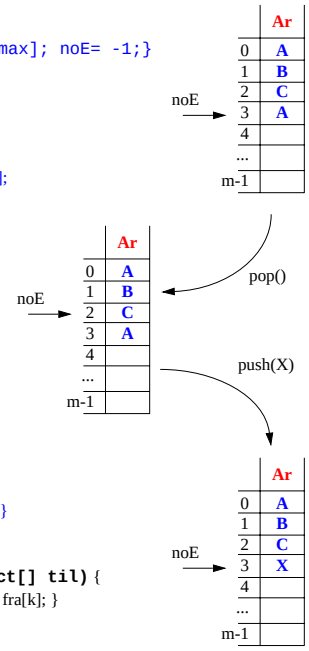
Egentlig ...



Stabel med Array

noE – Ar[noE] er toppen av stabel – hvis noE+1 = antall elementer > 0
max = Ar.length: max > noE >= -1

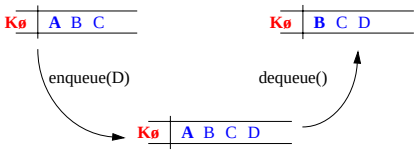
```
public class StabA
  IMPLEMENTS STACK {

  private Object[] Ar;
  private int noE, max=10;

  public StabA() { Ar= new Object[max]; noE= -1;}

  public void push(Object o) {
     if (noE=max-1) {
        Object[] temp= new Object[max];
        Copy(Ar, tab);
        max= 2*max; Ar= new Object[max];
        Copy(tab,Ar);  }
     noE++;
     Ar[noE]= o;
  }

  public Object pop() throws EmptyExc {
     if (empty()) throw new EmptyExc();
     else { noE--; return Ar[noE+1]; }
  }

  public Object peek() throws EmptyExc {
     if (empty()) throw new EmptyExc();
     else return Ar[noE];
  }

  public boolean empty() { return noE < 0; }

  public int size() { return noE+1; }

  private Copy(Object[] fra, Object[] til) {
     for (int k=0; k<fra.length; k++)  til[k]= fra[k]; }
}
```



O(1)

Queue med Array

```
public interface Queue {
    void enqueue(Object o);
    Object dequeue();
    Object front();
    boolean empty();
}
```



I. DATA REPRESENTASJON

en array med

– en peker for neste innsetting

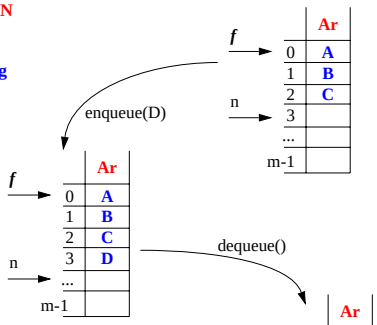
– en peker tilsvarende første

II. DATA STRUKTUR

Object[] Ar

int n, *f*

int max



III. DATA INVARIANT

n/f – indeks til neste / første

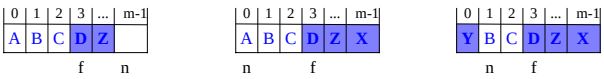
tom – hviss n==f (initielt begge er 0)

Når er den full? Etter max-innsettinger...n==f !

full – når #elem == max–1

#elem = (max – f + n) mod max < max

kø tilsv. A[f, f+1%max, f+2%max, ..., n–1]



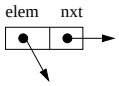

```
public class KøA
  IMPLEMENTS QUEUE {

private  Object[] Ar;
private int n, f, max=10;

   public KøA() {
      Ar= new Object[max];          // verifiser DI & O()
      n=0; f=0;
   }

   public void enqueue(Object o) {
      if (size() < max-1) {
          Ar[n] = o;
          n = (n + 1) % max;  }
      else { ??? }
   }

   public Object dequeue() throws EmptyExc {
      if (empty()) throw new EmptyExc("Tom ved dequeue()");
      else { tmp= Ar[f];
             f = (f +1) % max;
             return tmp; }
   }

   public Object front() throws EmptyExc {
      if (empty()) throw new EmptyExc("Tom ved front()");
      else return Ar[f];
   }

   public boolean empty() { return f == n; }

   public int size() { return (max – f + n) % max; }
}
```

n/f – indeks til neste / første
tom – hviss n==f
#elem = (max – f + n) mod max < max
full – hviss #elem == max–1
kø tilsv. A[f, f+1%max, f+2%max, ..., n-1]

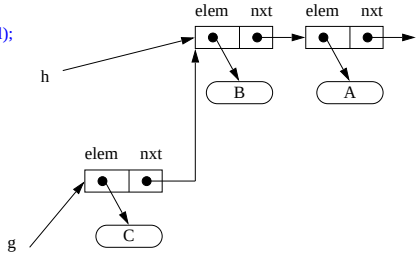
O(1)

Linket Liste DT

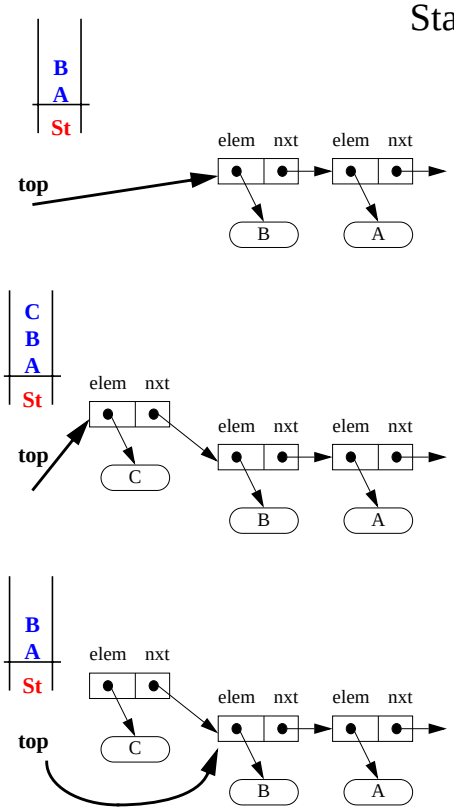
```
public class Node {
   private  Object elem;
   private Node next;
   public Node(Object o, Node n) {
      elem= o; next= n;}
   public Node() {
      this(null, null); }
   Object getElem() { return elem; }
   Node getNext() { return next; }
   void setElem(Object o) { elem= o; }
   void setNext(Node n) { next= n; }
}
```

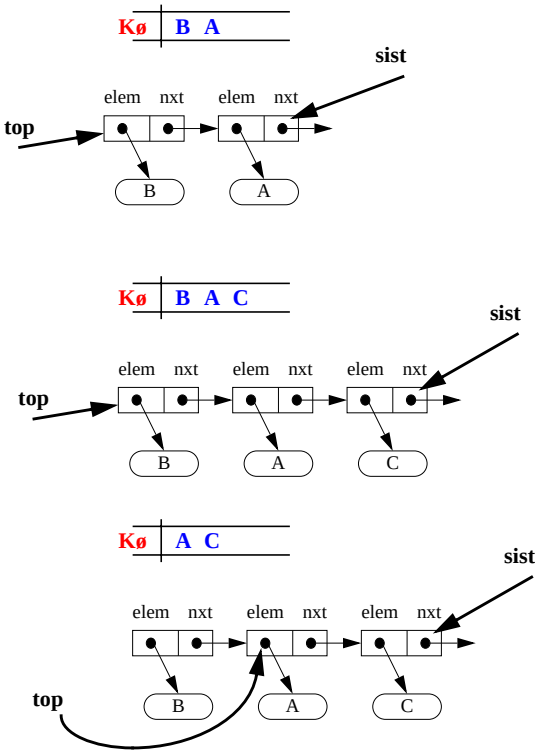



```
Node h= new Node(A,null);
h= new Node(B,h);

Node g= new Node(C,h)
```



Stabel / Kø med LL





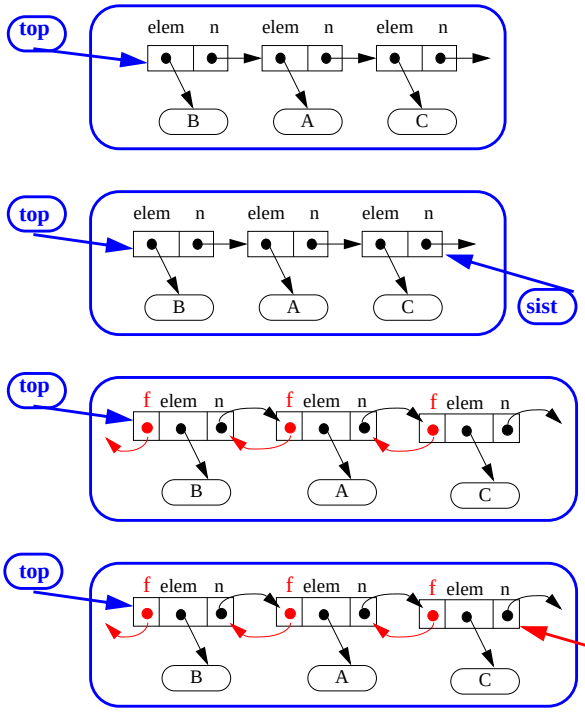
Stabel / Kø med LL

```
public class StabL
 IMPLEMENTS STACK {
 private Node top;
 private int s;

public StabL() { top= null;
 s= 0; }


public void push(Object o) {
 top= new Node(o,top); s++;
}



public Object pop() throws EmptyExc {
 if (empty()) throw new EmptyExc();
 else {
    Object u= top.getElem();
    top= top.getNext(); s--;
    return u;
 } }

public Object peek() throws EmptyExc {
 if (empty()) throw new EmptyExc() ;
 else return top.getElem();
}

public boolean empty()
   { return (top==null); }

public int size() { return s; }
}
```

```
public class KøL
 IMPLEMENTS QUEUE {
 private Node top, sist;
 private int s;

public KøL() {top= null;
 sist= null; s= 0; }


public void enqueue(Object o) {
 Node l= new Node(o,null);
 if (sist != null)
    sist.setNext(l);
 if (top == null) top= l;
 sist= l; s++;
}

public Object dequeue() throws EmptyExc{
 if (empty()) throw new EmptyExc();
 else {
    Object u= top.getElem();
    top= top.getNext(); s--;
    return u;
} }

public Object front() throws EmptyExc {
 if (empty()) throw new EmptyExc();
 else return top.getElem();
 }

public boolean empty()
   { return (top==null); }

public int size() { return s; }
}
```

O(1)

Lister og Listebasertre DataStrukturer



	travers.	fra	innsetting	fjerning
EN-VEIS				
	neste	første	første	første (Stabel)
	neste	første, siste	første siste	første (Kø)
TO-VEIS				
	neste, forrige	første	første vilkårlig	første vilkårlig
	neste, forrige	første, siste	første, vilkårlig, siste	første, vilkårlig, siste

interface DQueue

```
/**  en samling - kø - der objektene kan innsettes/fjernes i begge ender */
 extends Samling {
/**  setter inn et nytt objekt i starten
 *    @param o Objektet som skal innsettes */
      void insertFirst(Object o);

/**  setter inn et nytt objekt på slutten
 *    @param o Objektet som skal innsettes */
      void insertLast(Object o);

/**  fjerner objekt fra starten
 *    @return Objektet som fjernes
 *    @exception EmptyExc dersom køen er tom */
      void removeFirst(Object o) throws EmptyExc;

/**  fjerner objekt fra slutten
 *    @return Objektet som fjernes
 *    @exception EmptyExc dersom køen er tom */
      void removeLast(Object o) throws EmptyExc;

/**  returnerer første Objektet i køen
 *    @return første objektet i køen
 *    @exception EmptyExc dersom køen er tom */
      Object first() throws EmptyExc;

/**  returner siste Objektet
 *    @return siste objektet i køen
 *    @exception EmptyExc dersom køen er tom */
      Object last() throws EmptyExc;
}
```

implementasjon med toveis-liste med top og sist gir alle operasjonene $O(1)$

implementasjon
av en ADT med en annen ADT

```
class StackDQ implements Stack {
  private DQueue d;

  public StackDQ(DQueue e) {
    d= e; }

  public void push(Object o) {
    d.insertFirst(o); }

  public Object pop()
      throws EmptyExc {
    return d.removeFirst(); }

  public Object peek()
      throws EmptyExc {
    return d.first(); }

  public boolean empty() {
    returnd.empty(); }

  public int size() {
    return d.size(); }

}
```

```
class KøDQ implements Queue {
  private DQueue d;

  public KøDQ(DQueue e) {
    d= e; }

  public void enqueue(Object o) {
    d.insertFirst(o); }

  public Object dequeue()
      throws EmptyExc {
    return d.removeLast(); }

  public Object front()
      throws EmptyExc {
    return d.first(); }

  public boolean empty() {
    returnd.empty(); }

  public int size() {
    return d.size(); }

}
```

Bruker bestemmer hvilken implementasjon av DQueue han vil bruke når han oppretter et nytt objekt – kaller **... new StackDQ(???)**, **... new KøDQ(???)**