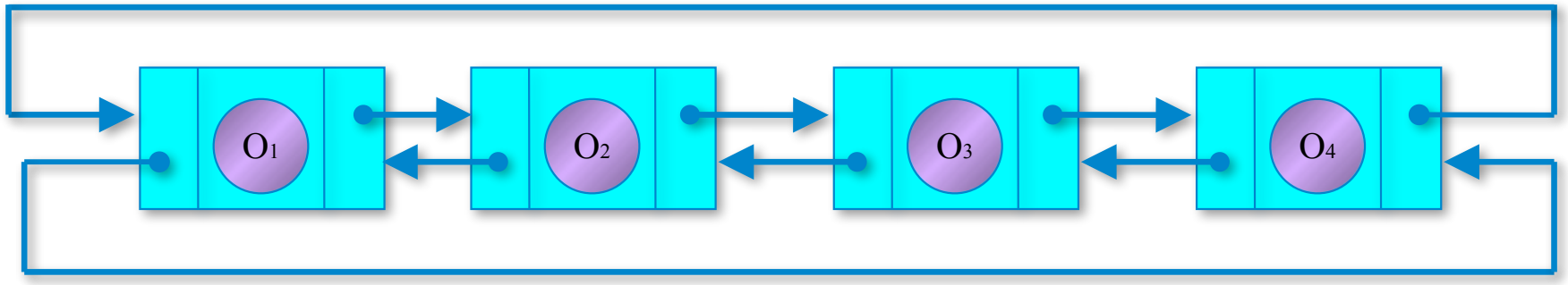
OOP
Dynamische Datenmengen
Datenabstraktion



SoSe 2020

Prof. Dr. Margarita Esponda

Datenabstraktion

Verschiedene Verständnisse von Datenabstraktion:

- **Allgemeine Abstraktion**

  Details werden hinter einer allgemeinen einfachen Idee versteckt.

- **Baukastenprinzip**

  Software-Architektur mit unabhängigen Modulen, die getrennt entwickelt und getestet werden können.

- **Datenkapselung**

  Kontrollierter Zugriff auf Daten und Methoden, um die interne Integrität von Objekten zu bewahren.

- **Schnittstellen**, die von Implementierung-Details getrennt sind und dadurch diese Information vom Rest des Systems verstecken. Die Implementierungen sind dadurch unabhängig und veränderbar.

- **Zuständigkeitstrennung**

  Einzelne Module übernehmen die Verantwortung für Features

Abstrakte Datentypen ADT

Erste Ideen:

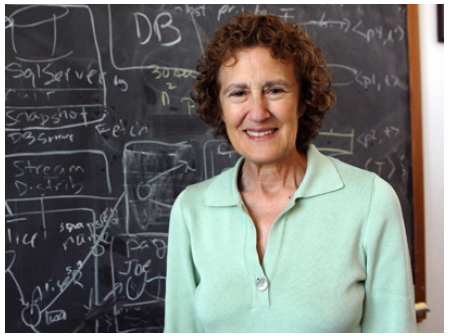
Ole-John Dahl

Erfinder der Programmiersprache Simula

Ausgereifte Idee

Barbara Liskov vom MIT

Erhielt den Turing Award für die Arbeit im Bereich ADT



ADT Operationen

Eine allgemein akzeptierte Klassifizierung ist:

Creators

erstellen neue Objekte aus anderen Objekt-Typen

Producers

produzieren Objekte aus existierenden Objekten, die den gleichen Datentyp haben

Observers

Teilobjekte eines abstrakten Datentyps werden zurückgegeben

Mutators

Verändern die Objekte

Beispiel

String Datentyp in Java

Creators

Konstruktoren

Producers

concat-, substring-, toUpperCase-Methode

Observers

charAt-Methode

Mutators

keine

Invarianten von ADT

Invarianten sind Eigenschaften oder Merkmale, die immer gelten.

Gut definierte abstrakte Datentypen haben auch Invarianten und sorgen selber dafür, das diese erhalten bleiben.

Eine wichtige Invariante vom String ist, dass die unveränderter sind (inmutable)

Eine gut definierte ADT für Heap-Datenstrukturen sorgt dafür, dass die Heap-Eigenschaft nach jeder Operation wieder hergestellt wird.

Abstrakte Datentypen

Zusammenfassung

- Abstrakte Datentypen sind durch ihre Operationen definiert.
- Operationen werden in ***creators***, ***producers***, ***observers***, and ***mutators*** klassifiziert.
- Eine gute **ADT** ist einfach und Darstellungs- bzw. Implementierung-unabhängig.
- Eine gut definierte ADT sorgt selber dafür, dass ihre vordefinierten Invarianten erhalten bleiben.
- ***creators*** und ***producers*** erstellen Objekte mit korrekten Invarianten, die von ***observers*** und ***mutators*** aufbewahrt werden.
- Gut definierte abstrakte Datentypen sollen **keine Darstellungsbelichtung** haben.

Dynamische Datenmengen als ADT

Dynamische Datenmengen können durch verschiedene **Datenstrukturen** im Rechner dargestellt werden.

Stapel und Schlangen

Stapel und Schlangen sind die einfachsten dynamischen Datenstrukturen.

Mögliche Implementierungen:

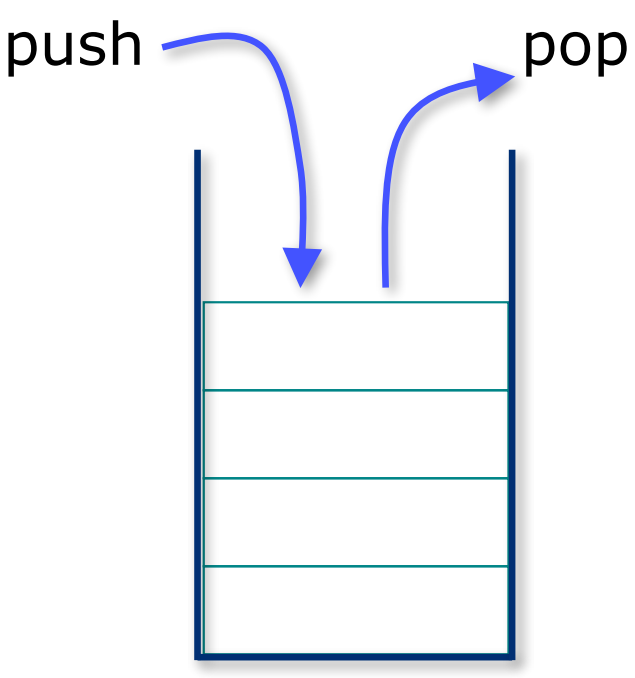
- Arrays
- "Dynamische Arrays"

  Wenn ein Feld voll ist, wird zur Laufzeit ein neues erzeugt, das doppelt so groß ist, und alle Daten des alten Feldes werden auf das neue Feld kopiert. Das Ganze wird wiederholt, wenn das Feld wieder ausgefüllt ist.
- Verkettete Listen

Stapel

In einem Stapel ("**stack**") darf nur das Element entfernt werden, das als letztes eingeführt worden ist.

LIFO - Datenstruktur "**L**ast **I**n - **F**irst **O**ut"



Mögliche Operationen:

push ist der Standard-Name der Einfüge-Operation in einem Stapel (**stack**).

pop ist der Standard-Name der Lösch-Operation.

peek liest das nächst verfügbare Element des Stapels, ohne dieses Element zu entfernen.

empty überprüft, ob der Stapel leer ist.

full überprüft, ob der Stapel voll ist.

Stapel-Schnittstelle

```
public interface Stack <E> {

    public boolean empty();

    public void push( E elem );

    public E pop()  throws EmptyStackException;

    public E peek()  throws EmptyStackException;

}
```

In dieser Implementierung werden wir eine EmptyStackException erzeugen bei dem Versuch, ein Element zu entfernen oder zu lesen (**pop**- und **peek**-Operationen), wenn der Stapel leer ist.

Stapel-Schnittstelle

```
void push( E elem );
```

Das **elem**-Objekt wird als oberstes Element des Stapels eingefügt

```
E pop() throws EmptyStackException;
```

Wenn der Stapel nicht leer ist, wird das oberste Element des Stapels entfernt und als Ergebnis zurückgegeben, andernfalls wird ein **EmptyStackException**-Objekt erzeugt.

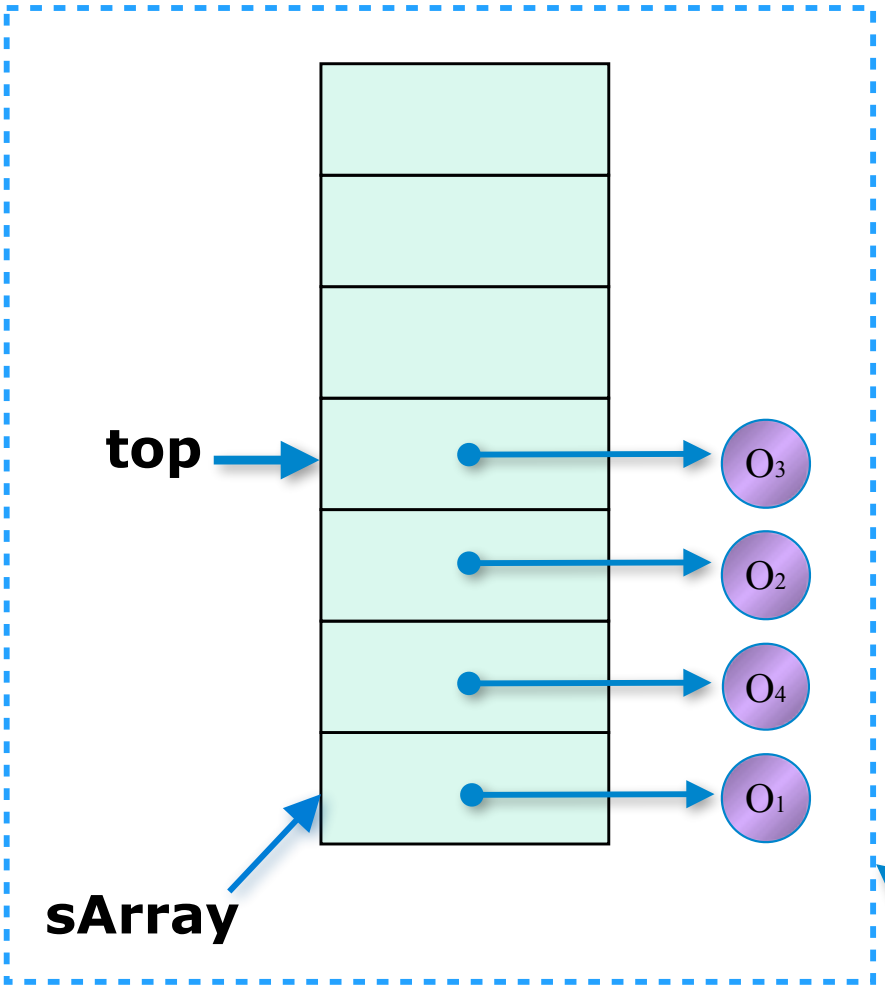
```
E peek() throws EmptyStackException;
```

Wenn der Stapel nicht leer ist, wird das oberste Element des Stapels gelesen und als Ergebnis zurückgegeben, andernfalls wird ein **EmptyStackException**-Objekt erzeugt.

```
boolean empty();
```

Überprüft, ob der Stapel leer ist.

Implementierung der Stapel-Schnittstelle



Für die Implementierung unseres Stapels verwenden wir ein Array (**sArray**), wo die Stapelelemente gespeichert werden und eine **int**-Variable (**top**), die immer auf das oberste Element des Stapels zeigt.

Implementierung der Stack-Schnittstelle

Im **sArray** werden die Stapelelemente gespeichert.

top zeigt immer auf das oberste Element des Stapels.

Zwei Konstruktoren werden definiert, die das Array mit einer Anfangsgröße initialisieren, und den **top**-Zeiger mit –1 (für leere Stapel) initialisieren.

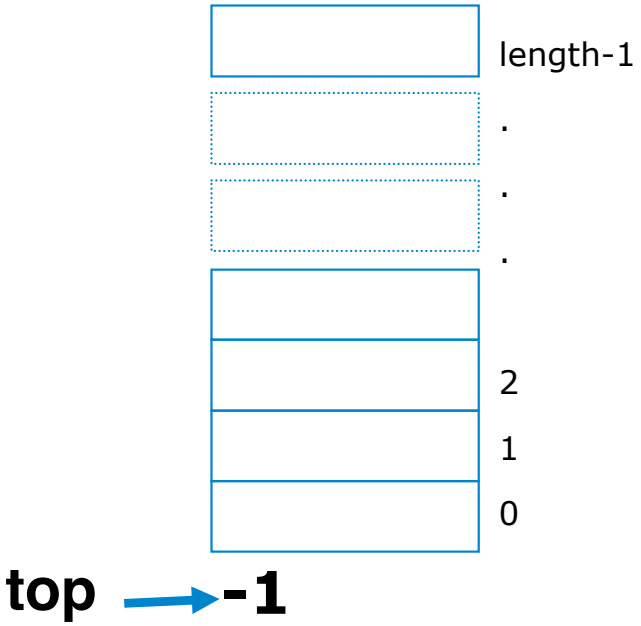
```
public class ArrayStapel<E> implements Stack<E> {

    private E[] sArray;
    private int top;

    public ArrayStapel(E[] sArray){
        top = -1;
        this.sArray = sArray;
    }

    public ArrayStapel(){
        this( (E[]) new Object[100]);
    }
    ...
}
```

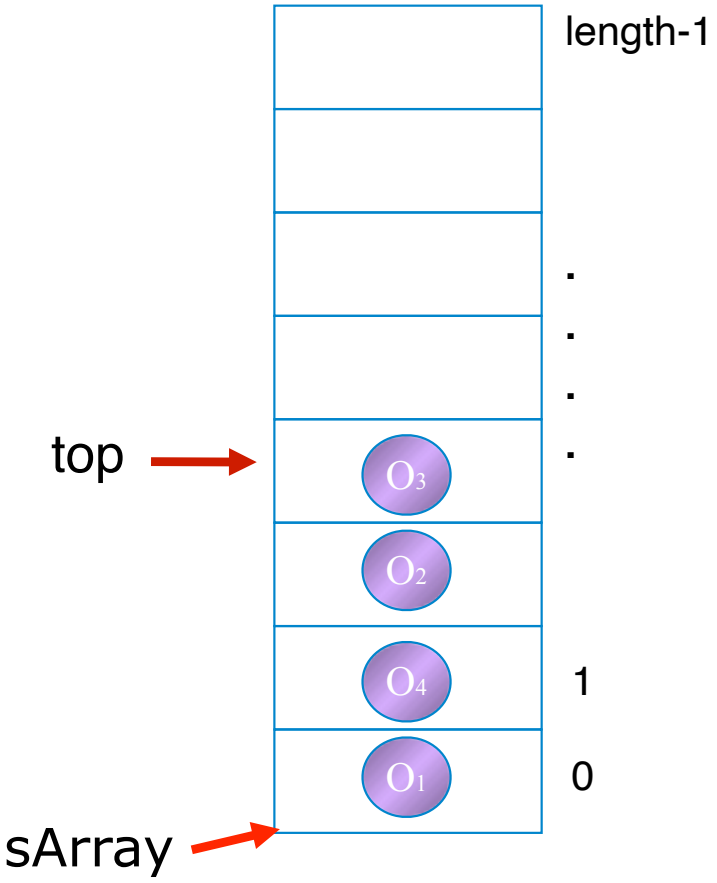
Die **empty**-Operation des Stapels



Der Stapel ist leer, wenn **top** auf keinen gültigen Stapelplatz zeigt.

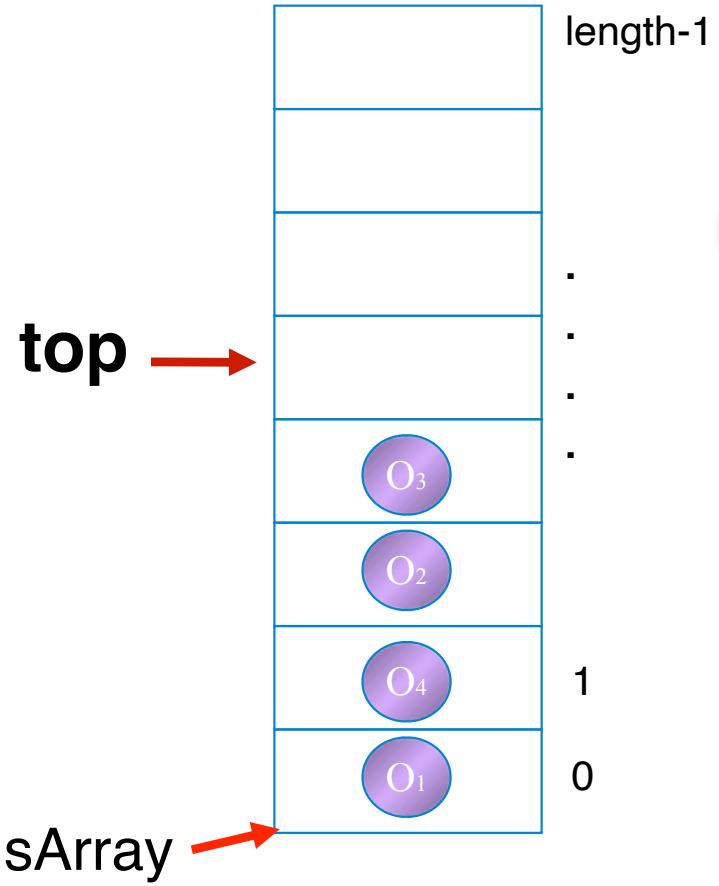
```
public boolean empty() {
    return   top == -1;
}
```

Die **push**-Operation



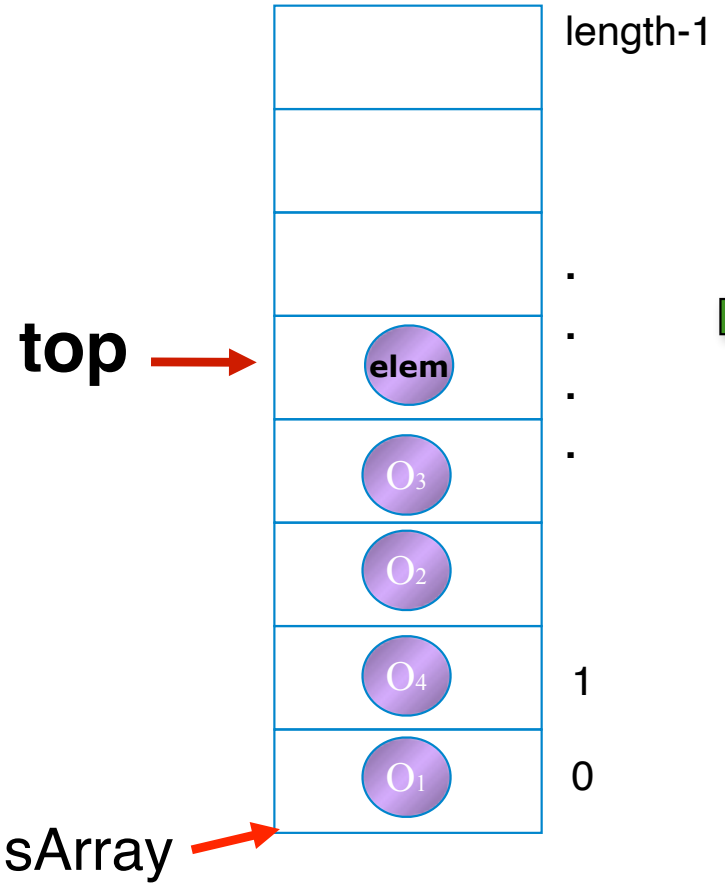

```
public void push(E elem) {
        if ( !full() )  {
            top++;
            sArray[top] = elem;
        } else {
            resizeSArray();
            top++;
            sArray[top] = elem;
        }
}
```

Die **push**-Operation



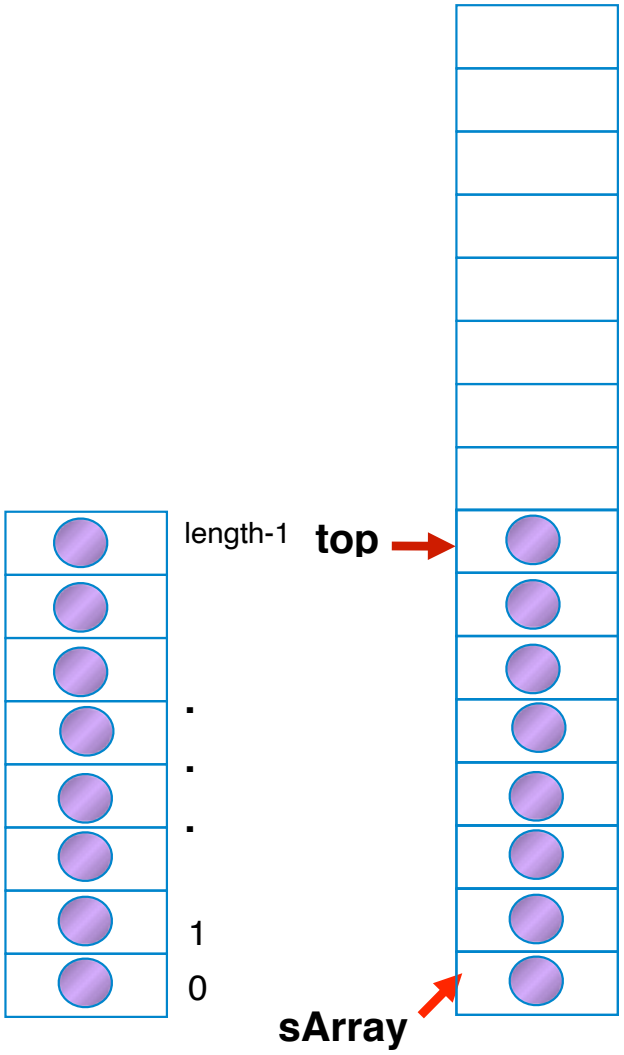

```
public void push(E elem) {
        if ( !full() )  {
            top++;
            sArray[top] = elem;
        } else {
            resizeSArray();
            top++;
            sArray[top] = elem;
        }
}
```

Die **push**-Operation



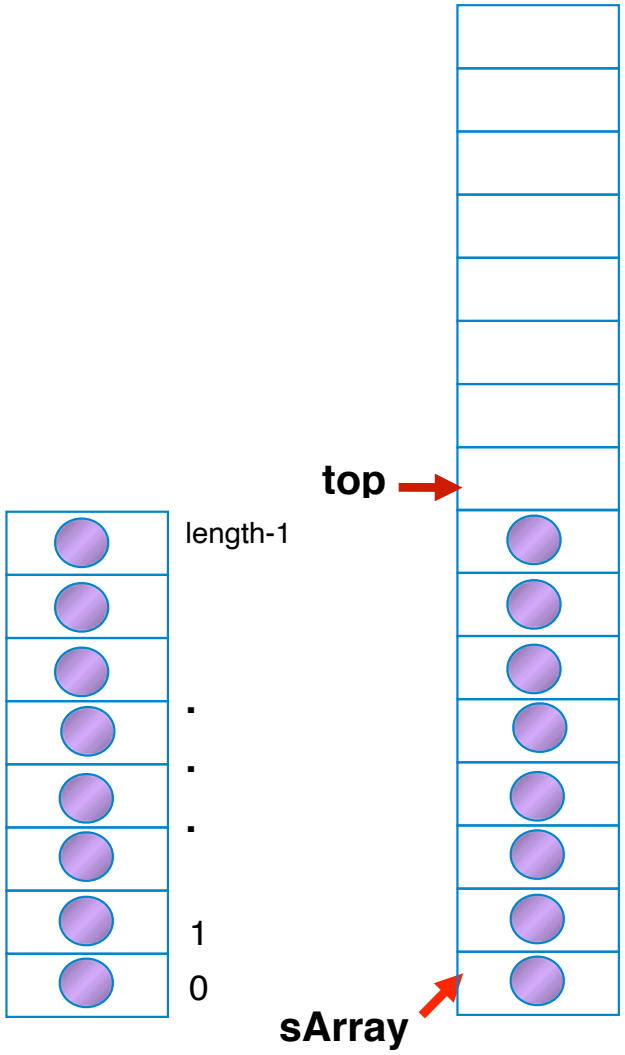

```
public void push(E elem) {
        if ( !full() )  {
            top++;
            sArray[top] = elem;
        } else {
            resizeSArray();
            top++;
            sArray[top] = elem;
        }
}
```

Die **push**-Operation



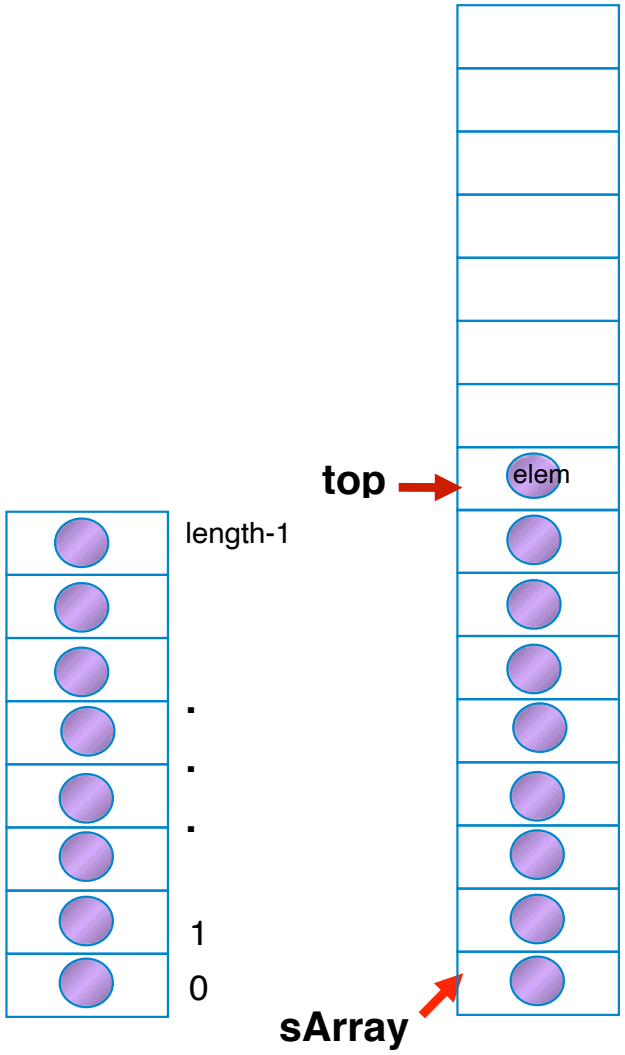

```
public void push(E elem) {
        if ( !full() )  {
            top++;
            sArray[top] = elem;
        } else {
            resizeSArray();
            top++;
            sArray[top] = elem;
        }
}
```

Die **push**-Operation



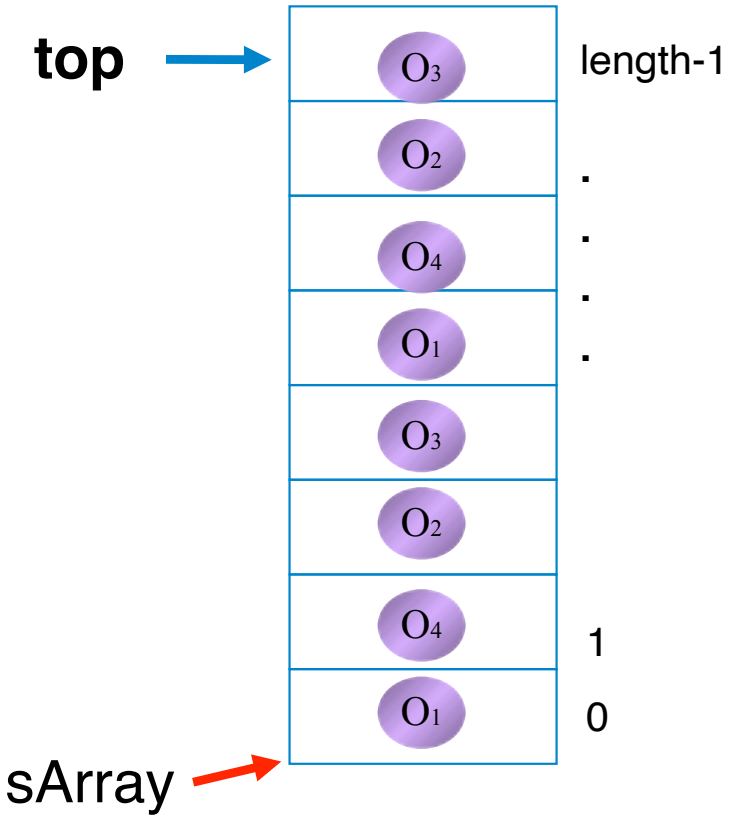

```
public void push(E elem) {
        if ( !full() )  {
            top++;
            sArray[top] = elem;
        } else {
            resizeSArray();
            top++;
            sArray[top] = elem;
        }
}
```

Die **push**-Operation




```
public void push(E elem) {
        if ( !full() )  {
            top++;
            sArray[top] = elem;
        } else {
            resizeSArray();
            top++;
            sArray[top] = elem;
        }
}
```

Die **full**-Hilfsmethode



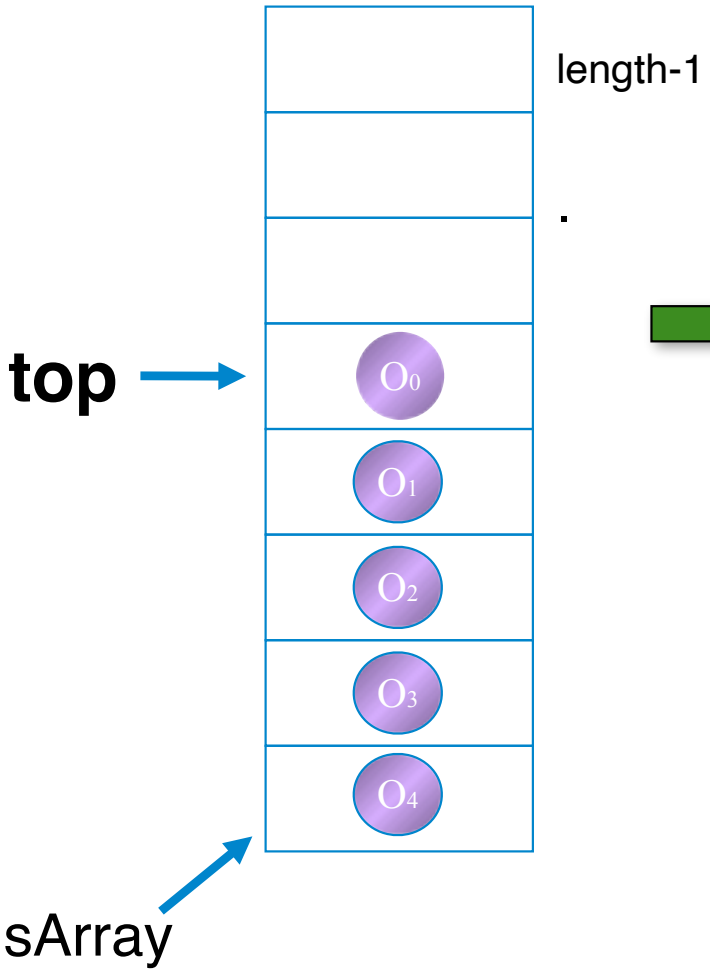
Der Stapel ist voll, wenn **top** gleich **stack.length-1** wird.

```
private boolean full() {
    return  !( top < sArray.length-1 );
}
```

Die **resizeSArray**-Hilfsmethode

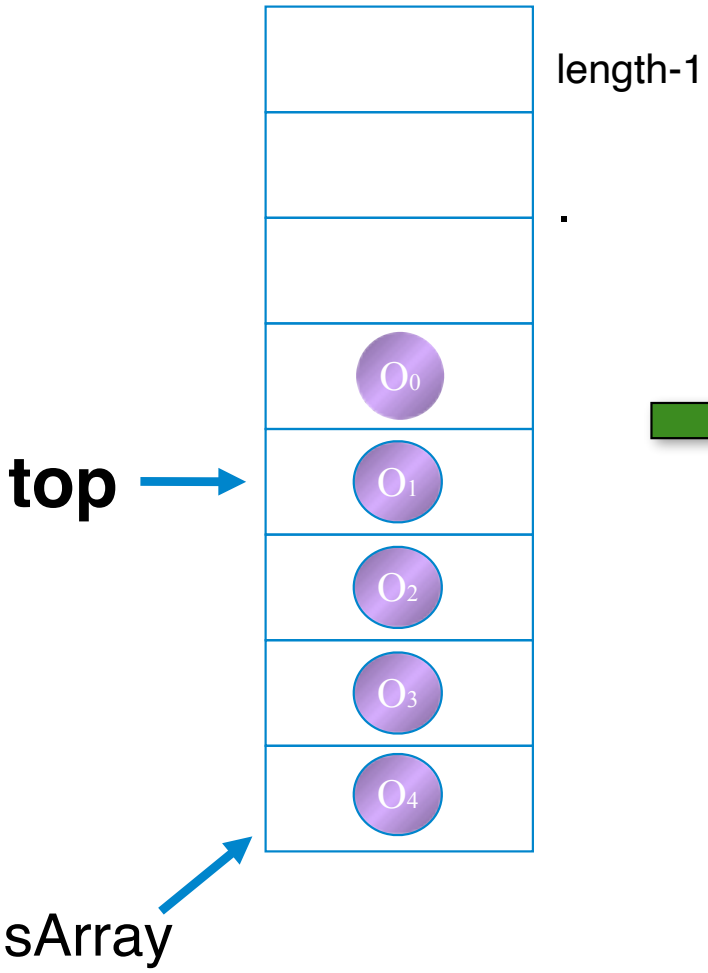
```
private void resizeSArray(){
    E[] temp = (E[]) new Object[sArray.length*2];
    for (int i=0; i<sArray.length; i++){
        temp[i] = sArray[i];
    }
    sArray = temp;
}
```

Die **pop**-Operation



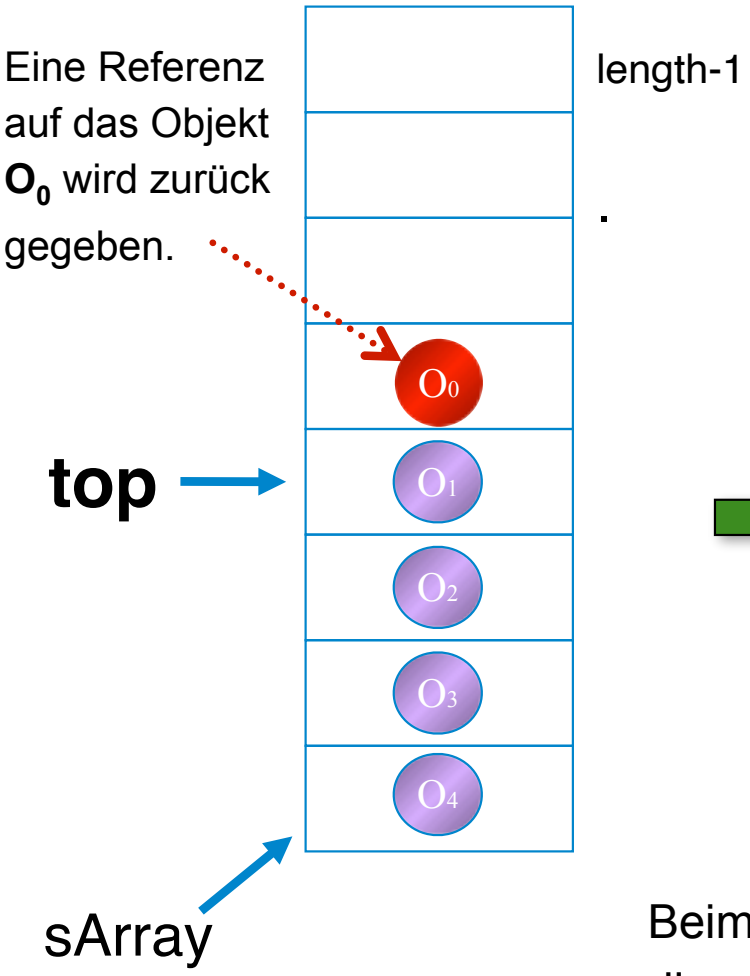

```
public E pop ()
        throws EmptyStackException {
    if ( !empty() ) {
        top--;
        return sArray [ top+1 ];
    } else
        throw new EmptyStackException();
}
```

Die **pop**-Operation




```
public E pop ()
        throws EmptyStackException {
   if ( !empty() ) {
        top--;
        return sArray [ top+1 ];
   } else
        throw new EmptyStackException();
}
```

Die **pop**-Operation

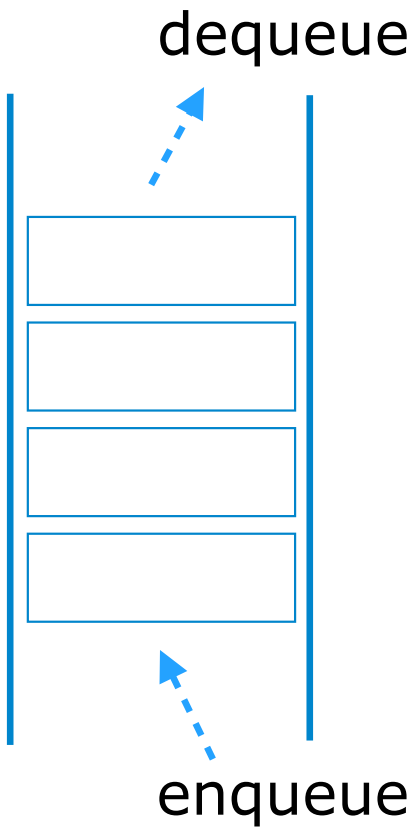



```
public E pop ()
        throws EmptyStackException {
    if ( !empty() ) {
        top--;
        return sArray [ top+1 ];
    } else
        throw new EmptyStackException();
}
```

Beim Einfügen eines neuen Elements im **sArray** wird die alte Referenz einfach überschrieben.

Implementierung einer Warteschlange als Array

Warteschlangen implementieren eine **FIFO**-Strategie. D.h. das erste Element, das eingeführt worden ist, ist das erste, das später entfernt wird.



FIFO - Datenstruktur

"**F**irst **I**n - **F**irst **O**ut"

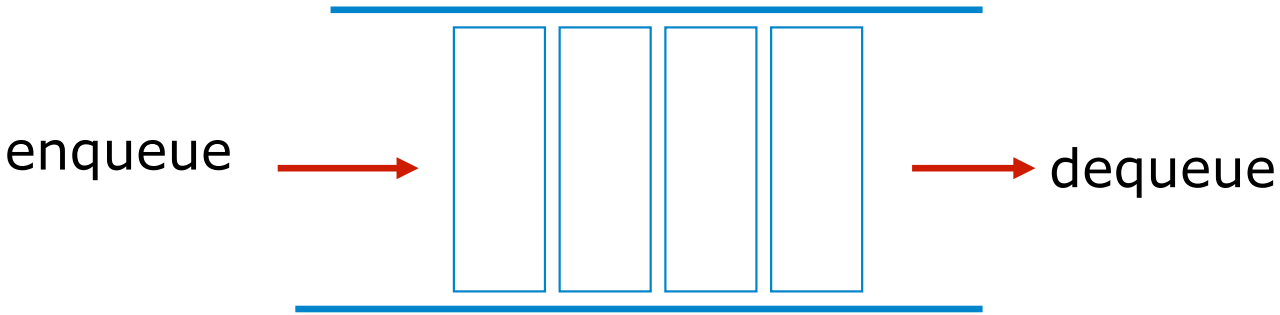
enqueue ist der Standard-Name der Einfüge-Operation.

dequeue ist der Standard-Name der Lösch-Operation.

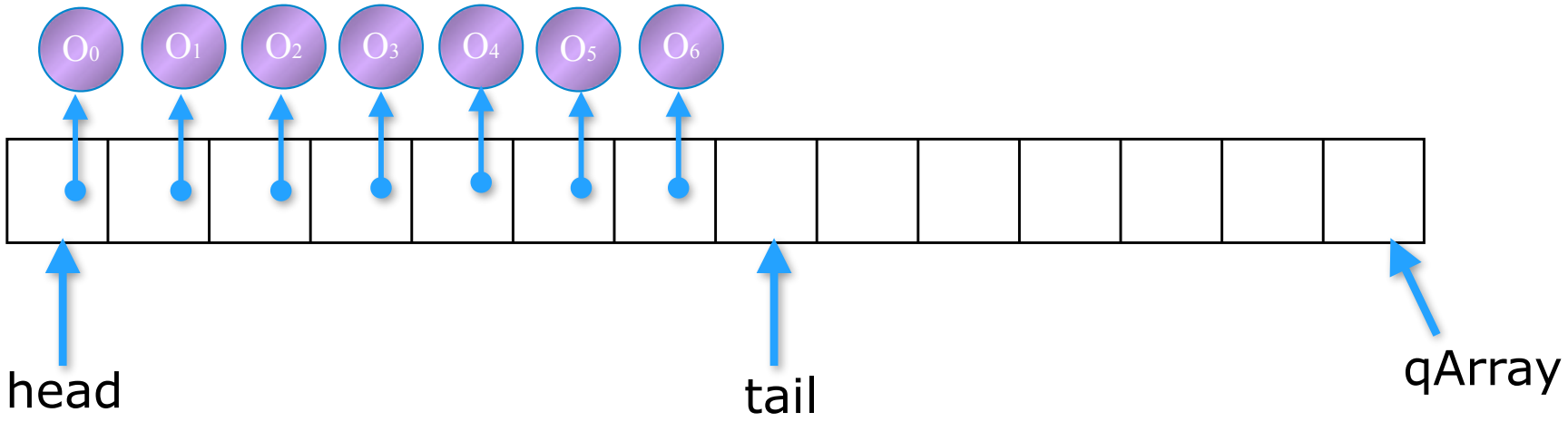
Warteschlangen-Operationen

Operationen

- enqueue
- dequeue
- head
- empty
- full



Implementierung der Warteschlange



Erstes Element der Warteschlange

Erster freier Platz in der Warteschlange

Implementierung einer Warteschlange

Implementierung einer Warteschlange

Implementierung einer Warteschlange

alle Personen bewegen sich!

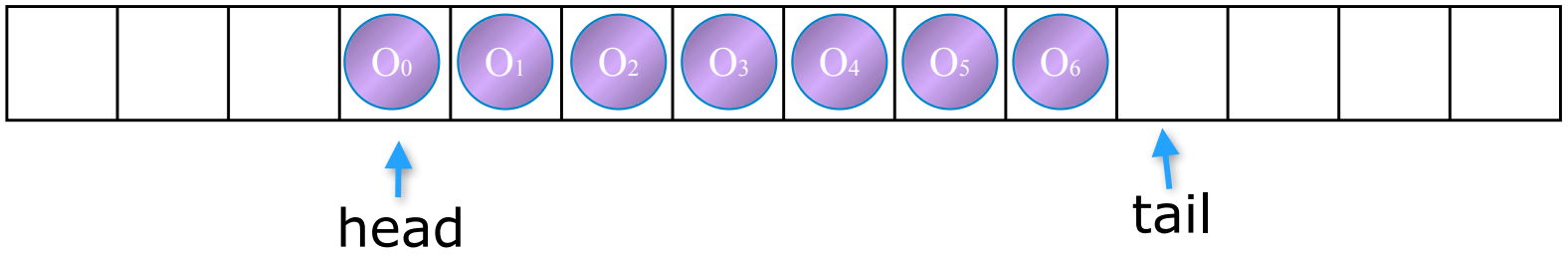
keine gute Idee für die Implementierung!

Wraparound-Strategie

Um die **Einfüge-** und **Lösch-**Operation in unserer Warteschlange in einer konstanten Zeit **O(1)** zu realisieren, wird das Feld in unserer Warteschlange als eine zirkulare Struktur betrachtet.

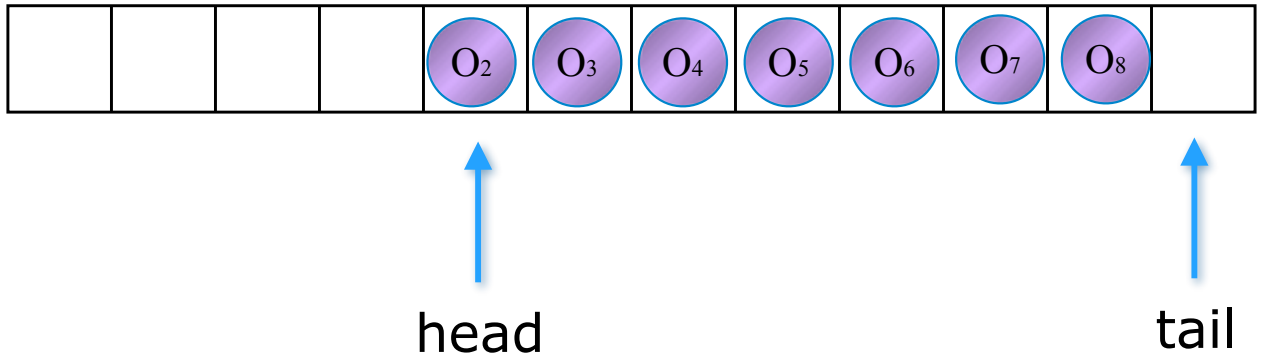
Neue Elemente in der Warteschlange werden in der Position eingefügt, die von **tail** angezeigt wird, und **tail** wird um eine Position nach rechts verschoben.

Wenn ein Element aus der Warteschlange entfernt wird, wird der **head-**Zeiger einfach um eine Position nach rechts bewegt.



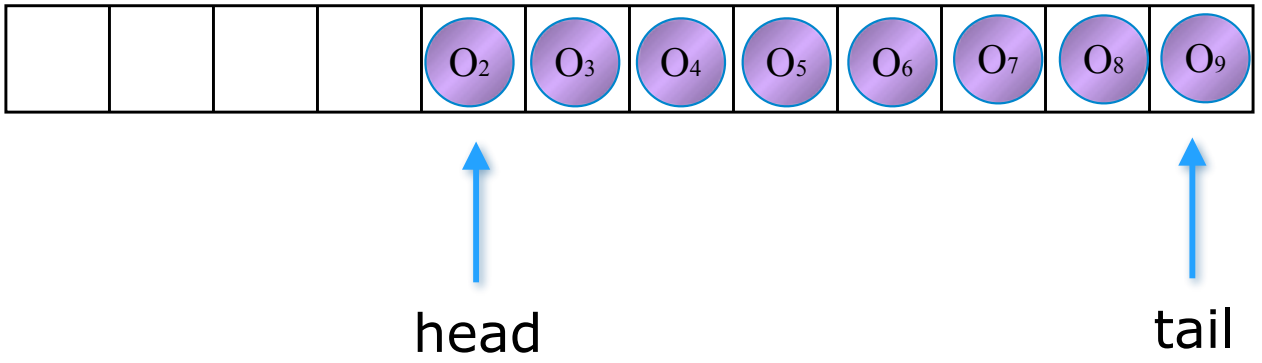
Wraparound-Strategie

enqueue

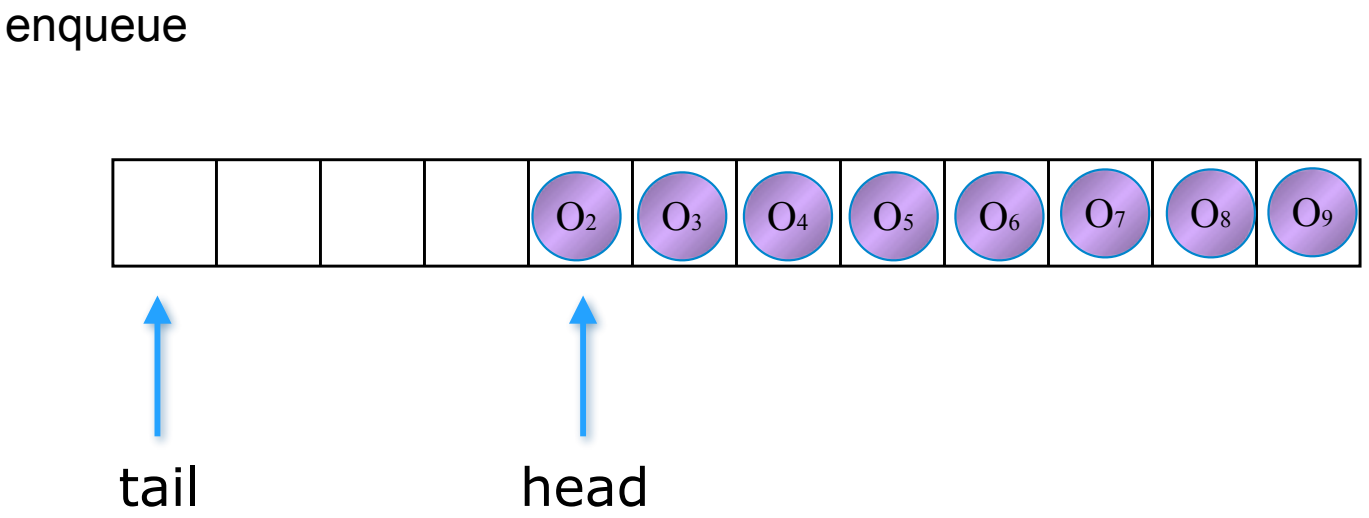


Wraparound-Strategie

enqueue

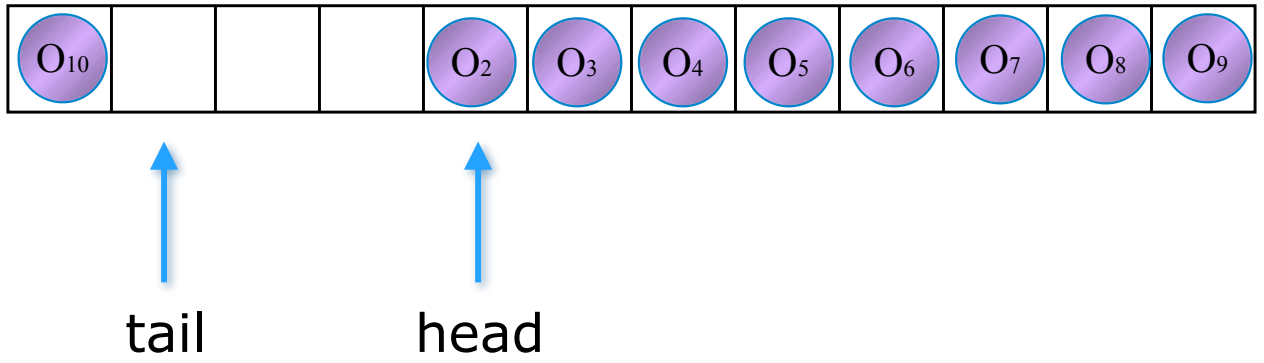


Wraparound-Strategie



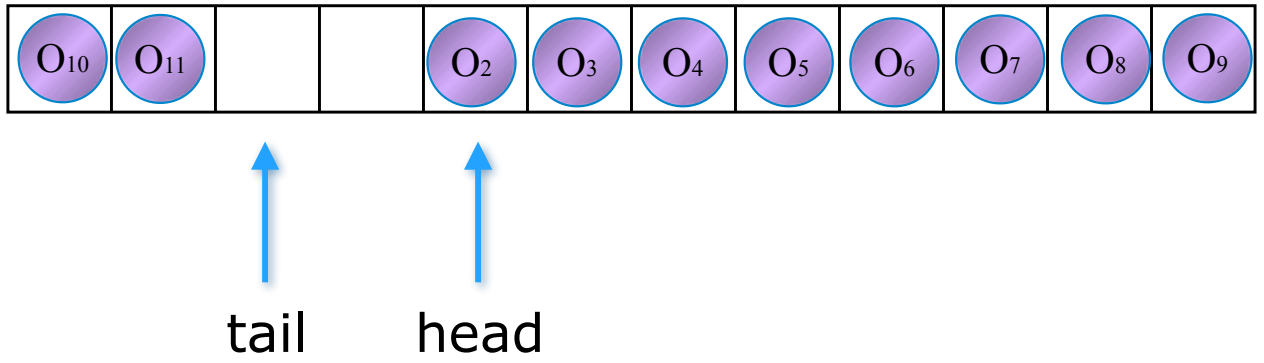
Wraparound-Strategie

enqueue



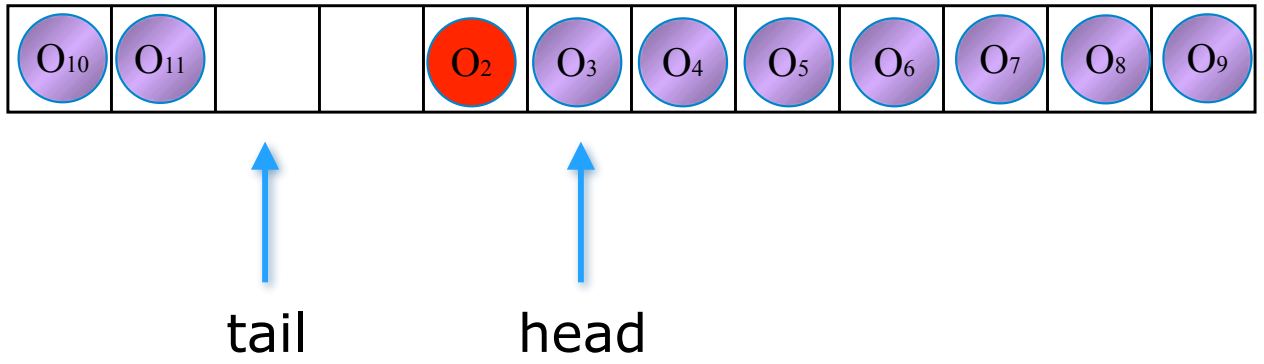
Wraparound-Strategie

enqueue



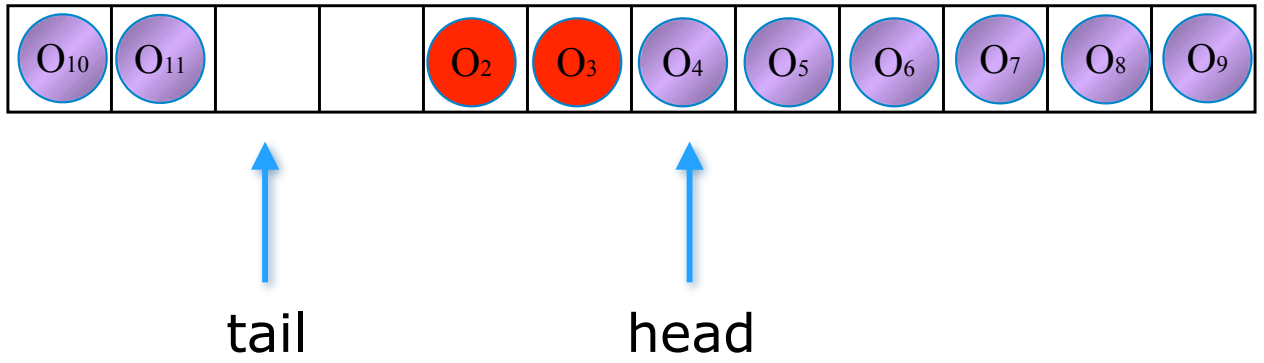
Wraparound-Strategie

dequeue



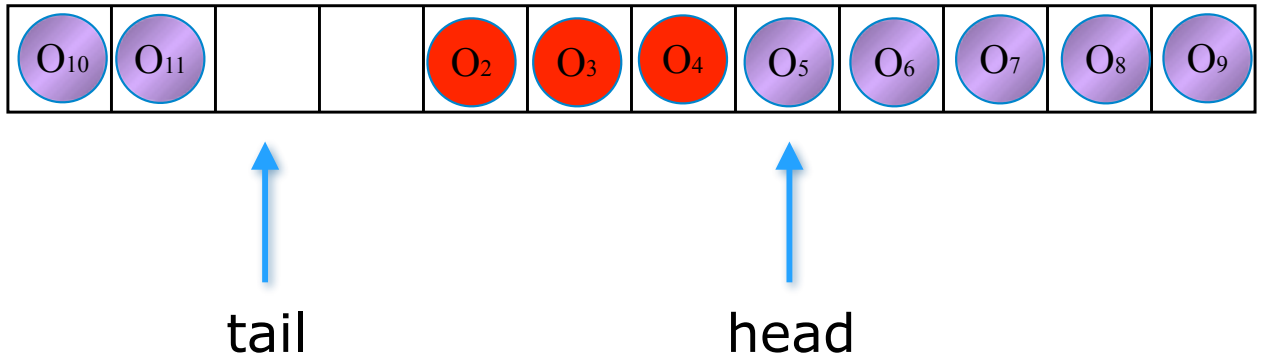
Wraparound-Strategie

dequeue



Wraparound-Strategie

dequeue



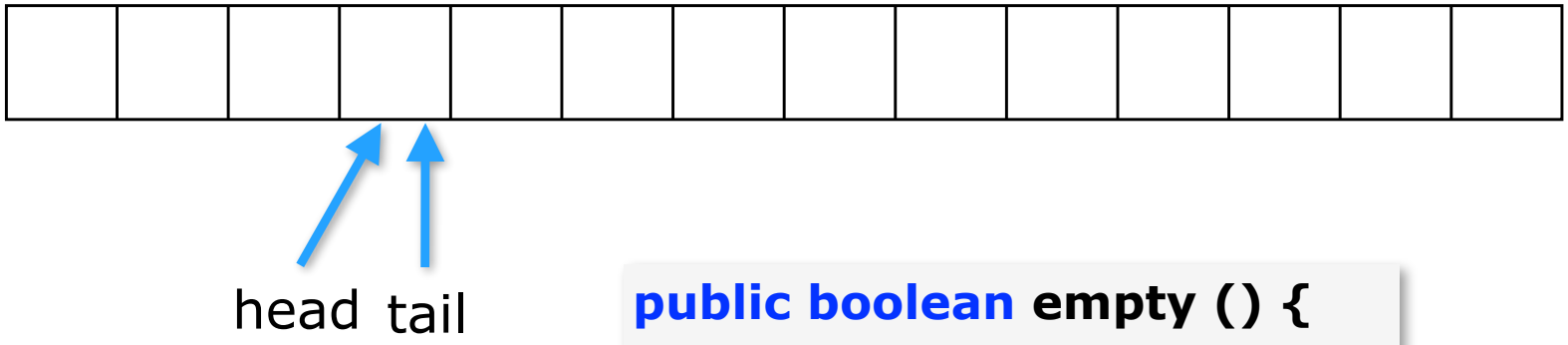
Die Warteschlangen-Schnittstelle

```
public interface Queue <E> {
    public void enqueue( E elem )  throws FullQueueException;
    public E dequeue() throws EmptyQueueException;
    public E head() throws EmptyQueueException;
    public boolean empty();
    public boolean full();
    public void toString();
}
```

Warteschlange mit fester Größe!

Die **empty**-Operation

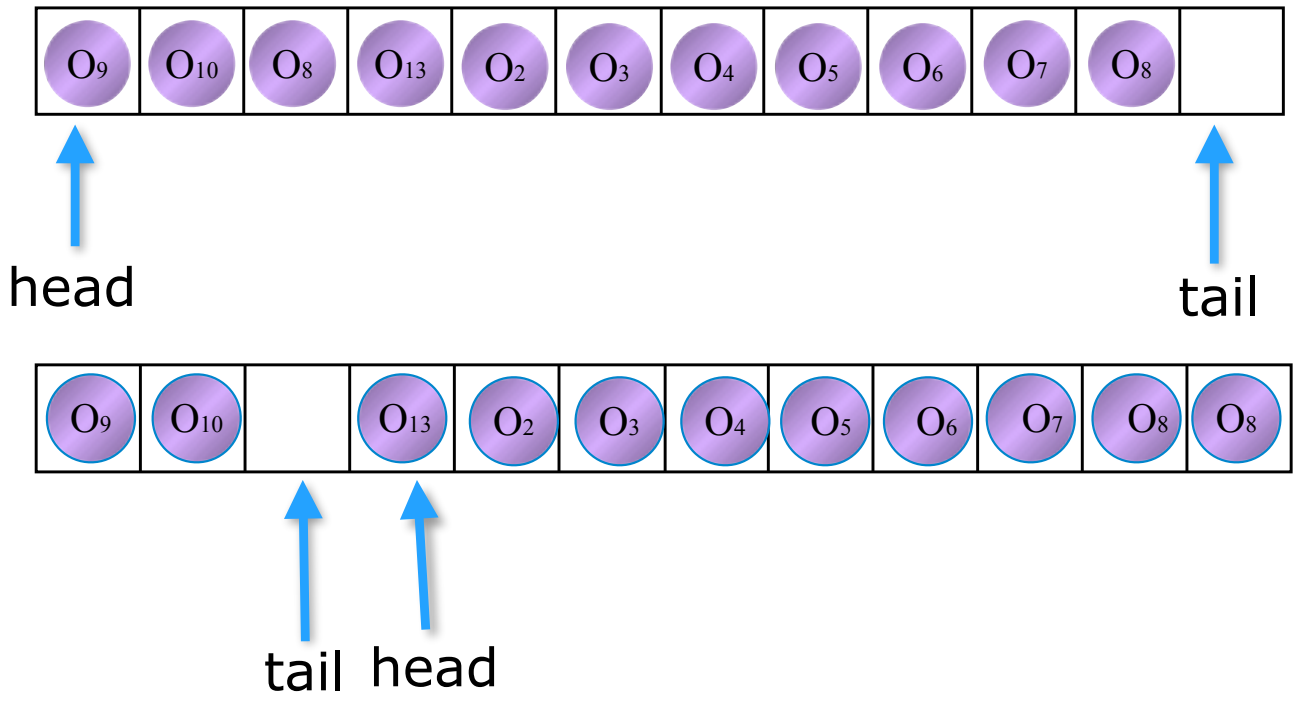
Unsere Warteschlange ist leer, wenn **head** und **tail** auf die gleiche Position des Feldes zeigen.




```
public boolean empty () {
        return head == tail;
}
```

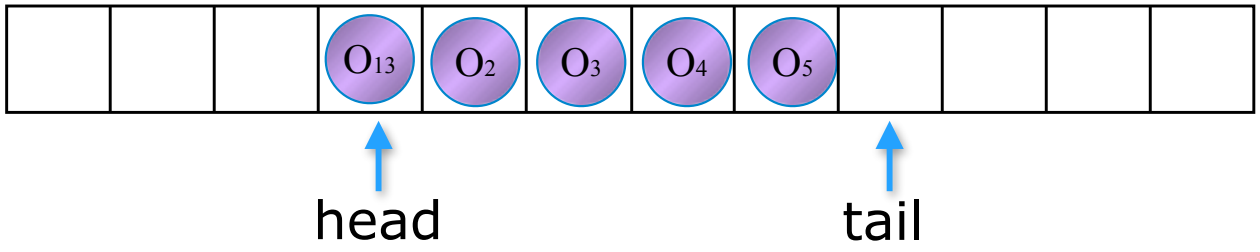
Die **full**-Operation

Wenn in diese Felder das letzte Element eingefügt wird, werden **tail** und **head** auch gleich sein und die Warteschlange wäre voll. D.h., wir würden nicht eine leere von einer vollen Warteschlange unterscheiden können. Deshalb werden wir nie unsere Warteschlange ausfüllen und den Zustand **full** definieren, wenn **tail** eine Position von **head** entfernt liegt.




```
public boolean full () {
    return   (( tail == queue.length-1 ) && ( head == 0 ))
             || ( head == ( tail+1 )) ;
}
```

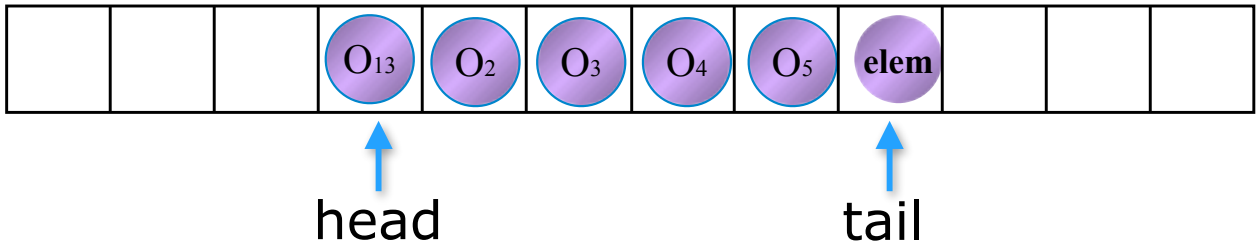
Die **enqueue**-Operation



Wenn die Warte-schlange nicht voll ist

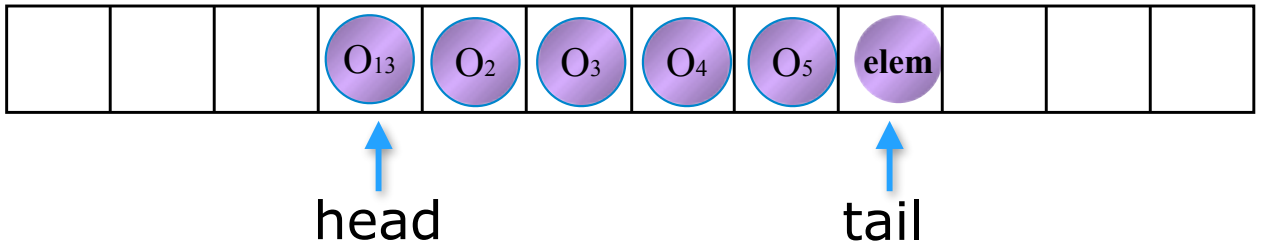
```
public void enqueue ( E elem ) throws FullQueueException {
    if ( !full() ) {
        queue[tail] = elem;
        if ( tail == (queue.length-1) )
            tail = 0;
        else  tail++;
    } else
        throw new FullQueueException();
}
```

Die **enqueue**-Operation




```
public void enqueue ( E elem ) throws FullQueueException {
        if ( !full() ) {
            queue[tail] = elem;
            if ( tail == (queue.length-1) )
                tail = 0;
            else  tail++;
        } else
            throw new FullQueueException();
}
```

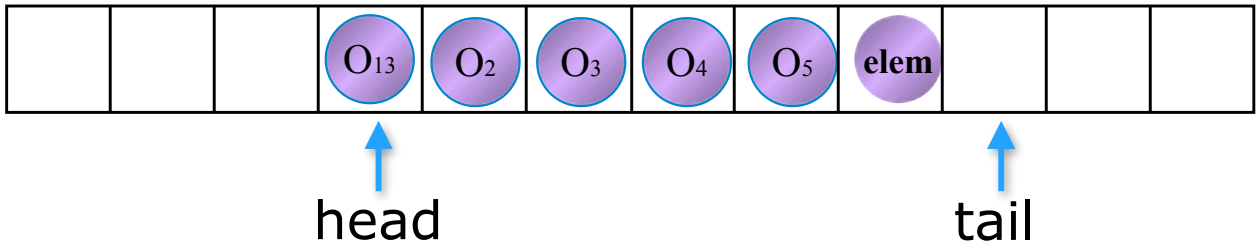
Die **enqueue**-Operation



Hier wird geprüft, ob **tail** am Ende des Feldes ist

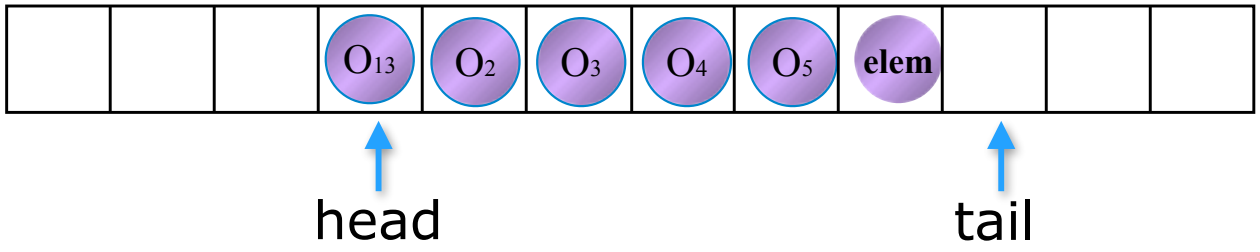
```
public void enqueue ( E elem ) throws FullQueueException {
        if ( !full() ) {
            queue[tail] = elem;
            if ( tail == (queue.length-1) )
                    tail = 0;
            else  tail++;
        } else
            throw new FullQueueException();
}
```

Die **enqueue**-Operation




```
public void enqueue ( E elem ) throws FullQueueException {
        if ( !full() ) {
            queue[tail] = elem;
            if ( tail == (queue.length-1) )
                tail = 0;
            else  tail++;
        } else
            throw new FullQueueException();
}
```

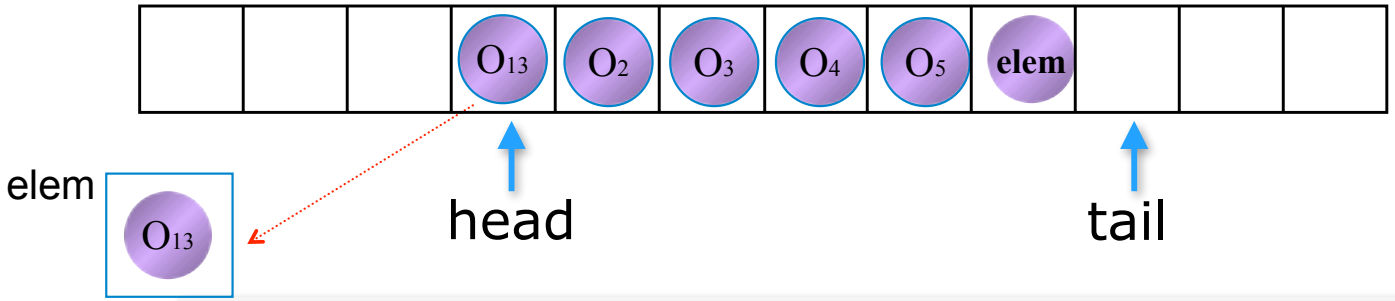
Die **dequeue**-Operation



Wenn die Warteschlange nicht leer ist.

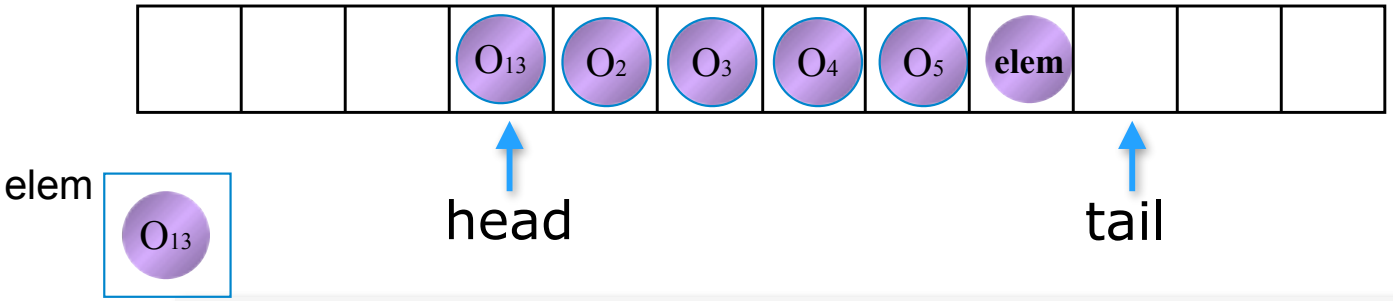
```
public E dequeue() throws EmptyQueueException {
    if ( !empty() ) {
        E elem = queue[head];
        if ( head == (queue.length-1) )
            head = 0;
        else  head++;
        return elem;
    } else
        throw new EmptyQueueException();
}
```

Die **dequeue**-Operation




```
public E dequeue() throws EmptyQueueException {
    if ( !empty() ) {
        E elem = queue[head];
        if ( head == (queue.length-1) )
            head = 0;
        else  head++;
        return elem;
    } else
        throw new EmptyQueueException();
}
```

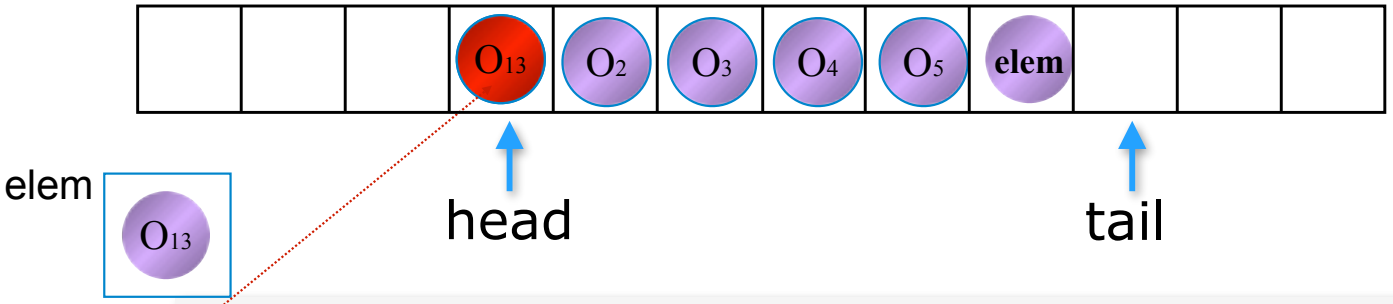
Die **dequeue**-Operation



Wenn **head** am Ende des Feldes ist.

```
public E dequeue() throws EmptyQueueException {
    if ( !empty() ) {
        E elem = queue[head];
        if ( head == (queue.length-1) )
            head = 0;
        else  head++;
        return elem;
    } else
        throw new EmptyQueueException();
}
```

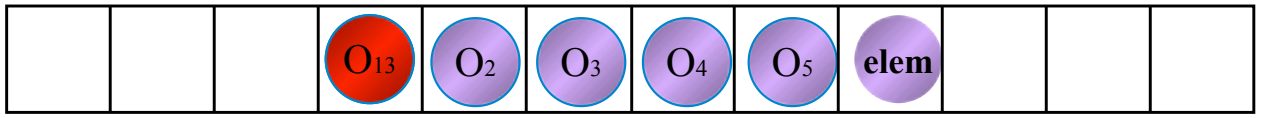
Die **dequeue**-Operation

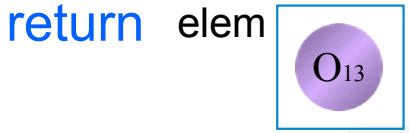


Die verbliebene Objekt-Referenz wird später überschrieben.

```
public E dequeue() throws EmptyQueueException {
    if ( !empty() ) {
        E elem = queue[head];
        if ( head == (queue.length-1) )
            head = 0;
        else  head++;
        return elem;
    } else
        throw new EmptyQueueException();
}
```

Die **dequeue**-Operation



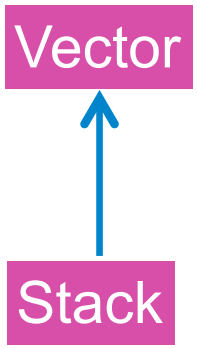


head

tail

```
public E dequeue() throws EmptyQueueException {
      if ( !empty() ) {
          E elem = queue[head];
          if ( head == (queue.length-1) )
                head = 0;
          else  head++;
          return elem;
      } else
          throw new EmptyQueueException();
}
```

Java-Klassen für Stapel und Warteschlangen




```
public class Stack<E> extends Vector <E> {
    . . .
}
```

```
public interface Queue <E> extends Collection <E>
```

Einige Implementierungen sind:

- AbstractQueue
- PriorityBlockingQueue
- ConcurrentLinkedQueue
- DelayQueue
- LinkedBlockingQueue
- ArrayBlockingQueue
- PriorityQueue
- SynchronousQueue
- LinkedList

Zusammenfassung

Stapel und Schlangen sind dynamische Datenstrukturen, doch die Implementierung mit Hilfe von Feldern hat die Einschränkung, dass die maximal erreichbare Größe vorher bekannt sein muss.

Eine gute Lösung dieses Problems sind „Dynamische Arrays“. Wenn ein Feld voll ist, wird zur Laufzeit ein neues erzeugt, das doppelt so groß ist, und alle Daten des alten Feldes werden auf das neue Feld kopiert. Das Ganze wird wiederholt, wenn das Feld wieder ausgefüllt ist.

Eine zweite Lösung ist, von Anfang an echte dynamische Datenstrukturen zu verwenden (wie Listen, Bäume, usw.).