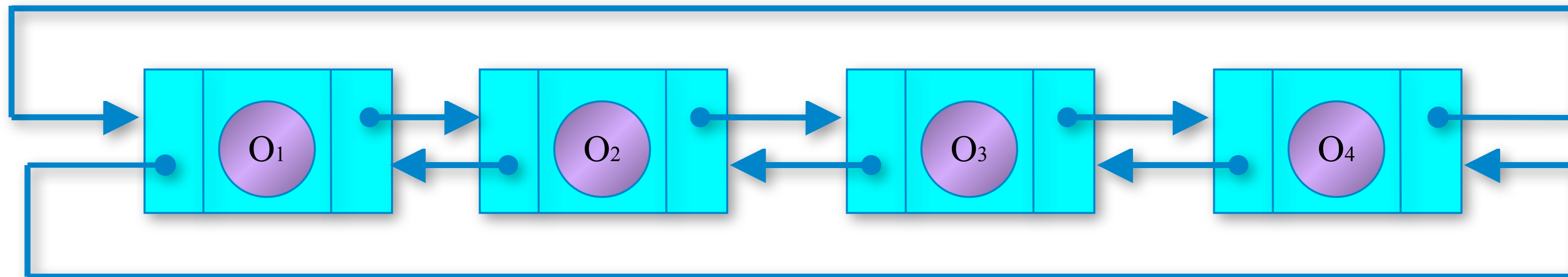
ALP II

Dynamische Datenmengen

Datenabstraktion



SS 2012

Prof. Dr. Margarita Esponda

Dynamische Datenmengen

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.

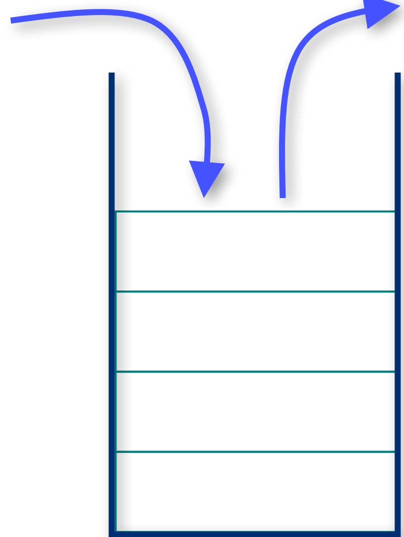
- Verkettete Listen

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"

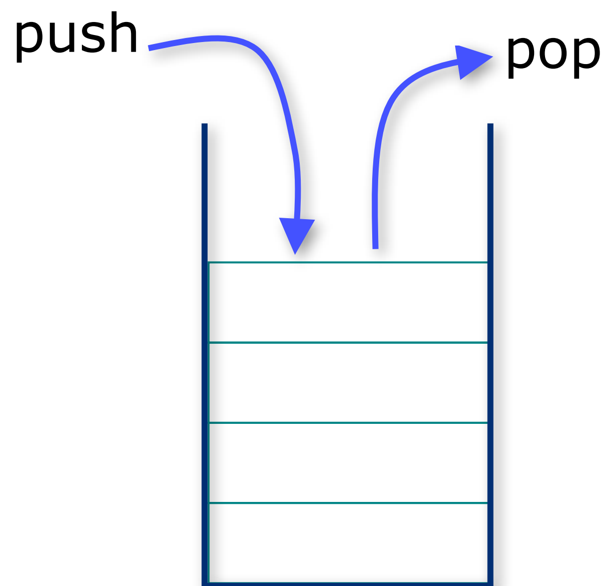


push ist der Standard-Name der Einfüge-Operation in einem Stapel (**stack**).

pop ist der Standard-Name der Lösch-Operation.

Stapel-Operationen

Zusätzlich zu den **push-** und **pop**-Operationen brauchen wir eine **peek**-Operation, die nur das nächst verfügbare Element des Stapels liest, ohne dieses Element zu entfernen und eine **empty**-Operation, die einen Wahrheitswert zurückliefert, wenn der Stapel leer ist.



Operationen
- push
- pop
- peek
- empty

Eine zusätzliche **full**-Operation kann überprüfen, ob der Stapel voll ist.

Stapel-Schnittstelle

```
public interface Stack <E> {

    public boolean empty();

    public void push( E element );

    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 die Liste leer ist.

Stapel-Schnittstelle

```
void push( E element );
```

Wenn der Stapel noch nicht voll ist, wird das Objekt **element** als oberstes Element des Stapels eingefügt, andernfalls wird ein FullStackException-Objekt erzeugt.

```
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.

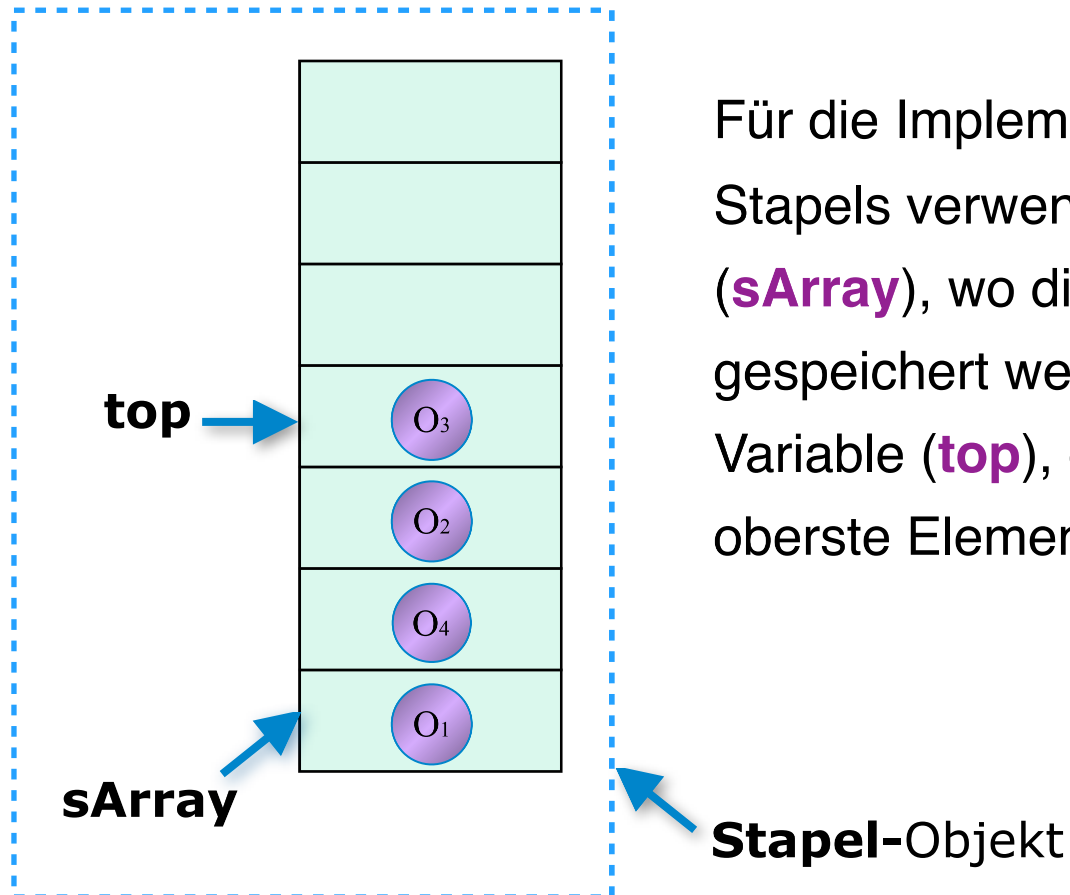
```
E peek() throws EmptyStackException;
```

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.

Stapel-Objekt

Implementierung der Stack-Schnittstelle

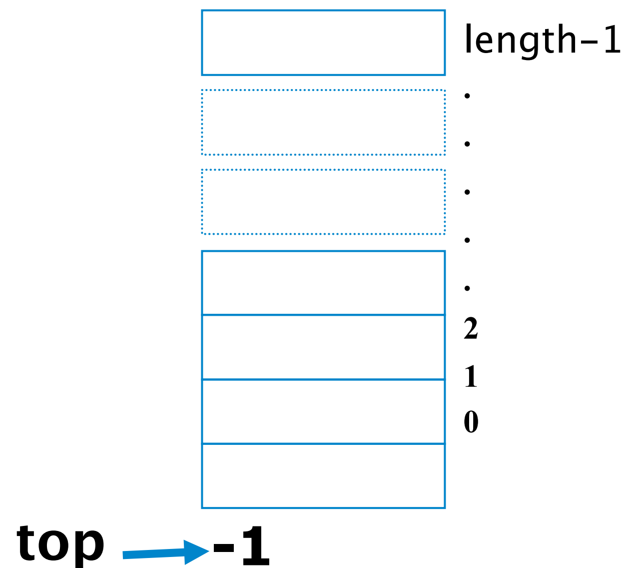
Im **sArray** werden die Stapelelemente gespeichert.

top zeigt immer auf das oberste Element des Stapels.

Drei 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(){
        top = -1;
        this.sArray = (E[]) new Object[100];
    }

    public ArrayStapel(int size){
        top = -1;
        this.sArray = (E[]) new Object[size];
    }
    . . .
}
```

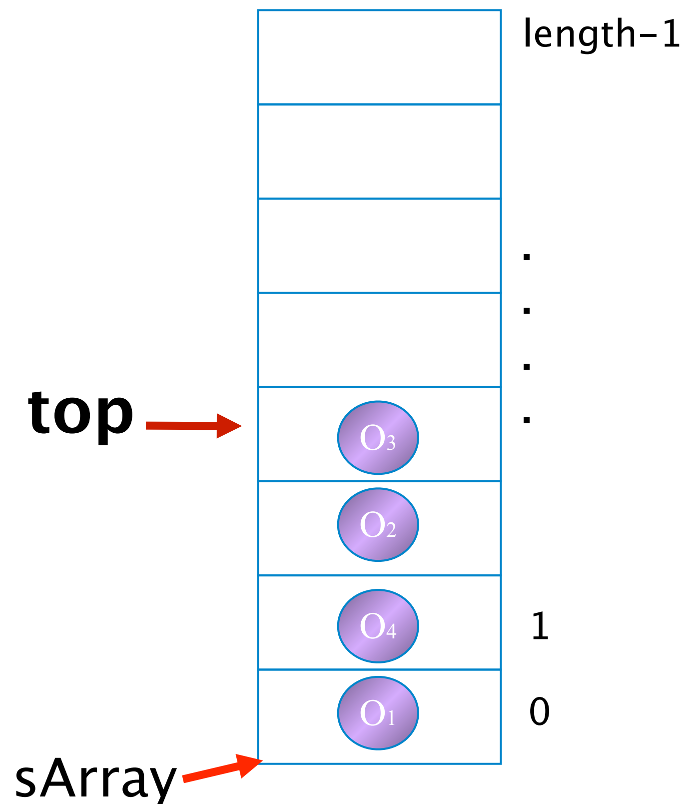
Die empty-Operation des Stapels



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

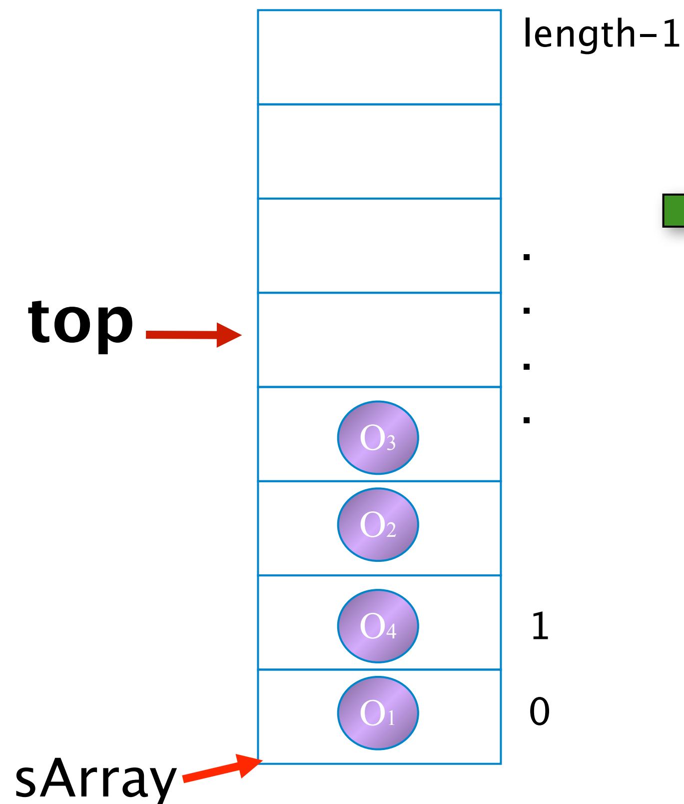
```
public boolean empty() {
    return ( top == -1 );
}
```

Die **push**-Operation



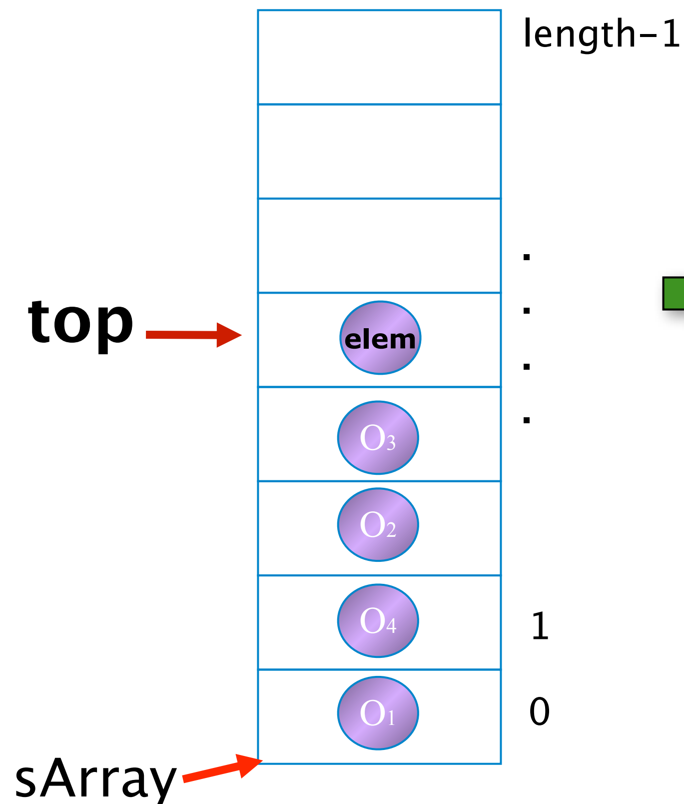

```
public void push(E element) {
        if ( !full() )  {
              top++;
              sArray[top] = element;
        } else {
              resizeSArray();
              top++;
              sArray[top] = element;
        }
}
```

Die **push**-Operation



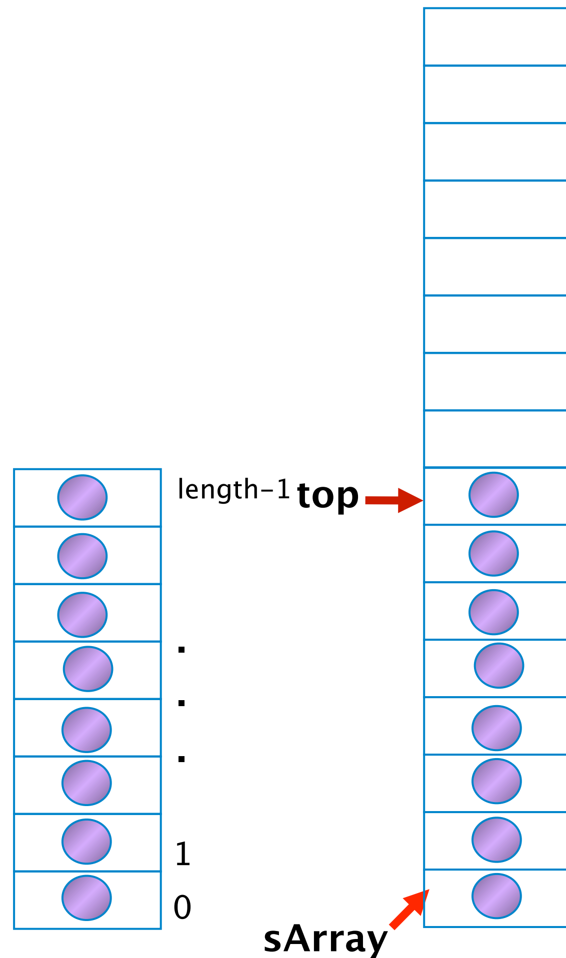

```
public void push(E element) {
        if ( !full() )  {
              top++;
              sArray[top] = element;
        } else {
              resizeSArray();
              top++;
              sArray[top] = element;
        }
}
```

Die **push**-Operation



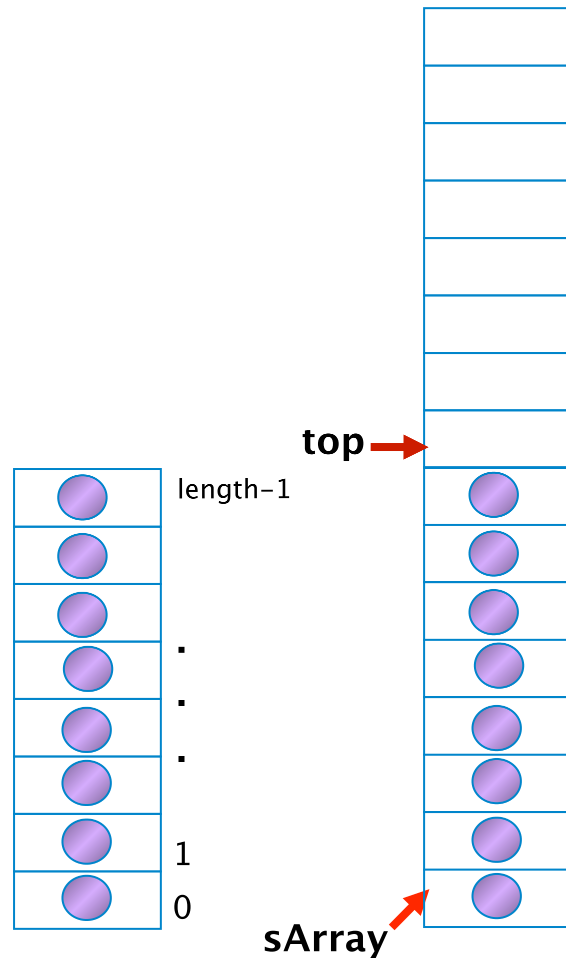
```
public void push(E element) {
        if ( !full() )  {
              top++;
              sArray[top] = element;
        } else {
              resizeSArray();
              top++;
              sArray[top] = element;
        }
}
```

Die **push**-Operation



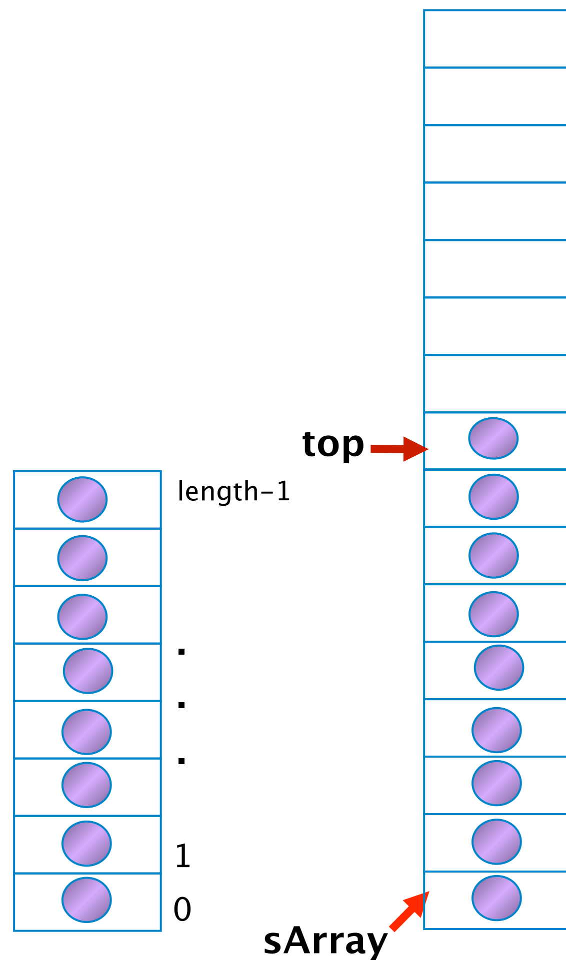

```java
public void push(E element) {
        if ( !full() )  {
              top++;
              sArray[top] = element;
        } else {
              resizeSArray();
              top++;
              sArray[top] = element;
        }
}
```

Die **push**-Operation



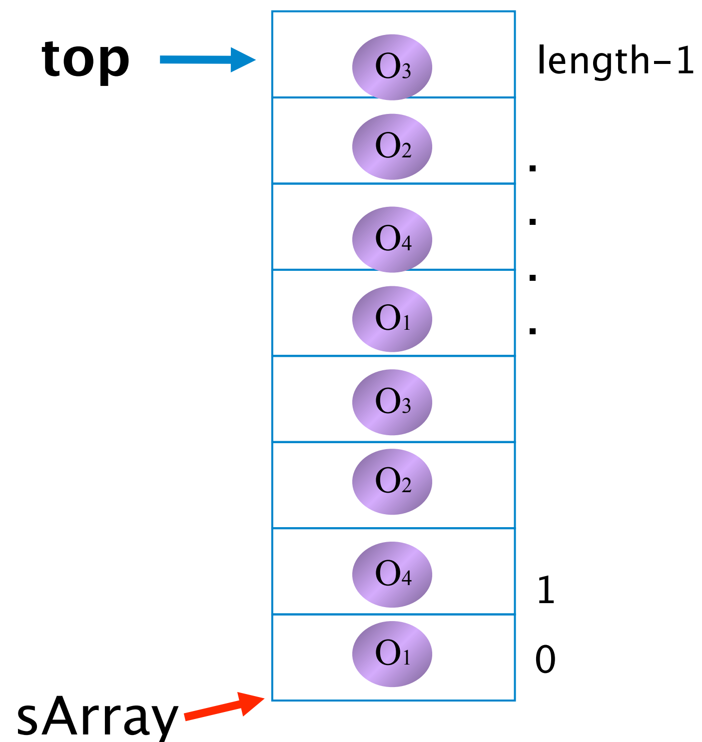

```
public void push(E element) {
        if ( !full() )  {
            top++;
            sArray[top] = element;
        } else {
            resizeSArray();
            top++;
            sArray[top] = element;
        }
}
```

Die **push**-Operation




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

Die **full**-Hilfsmethode



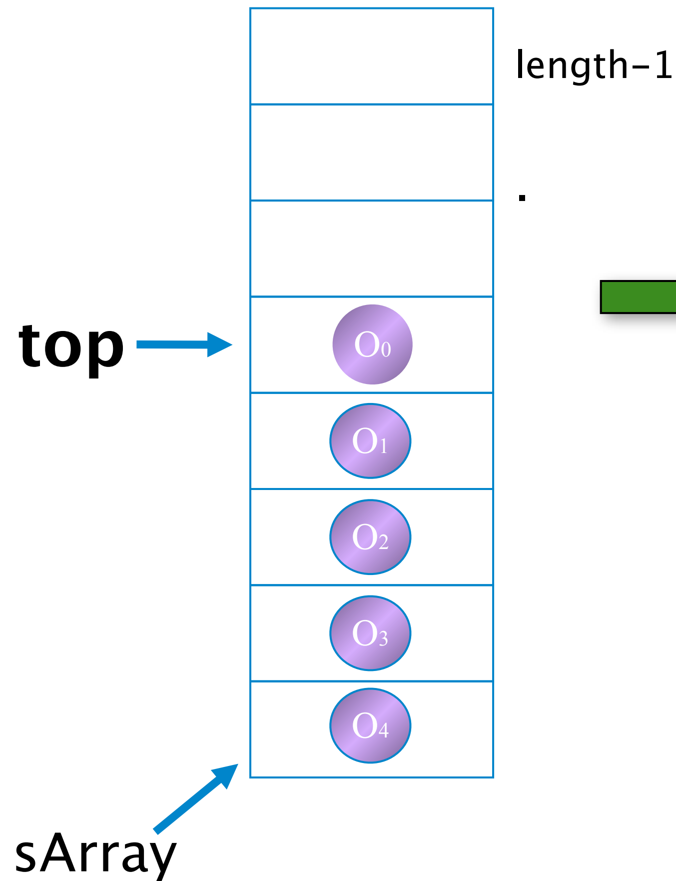
Der Stapel ist voll, wenn top gleich **stack.length-1** wird.

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

Die **resizeSArray**-Hilfsmethode

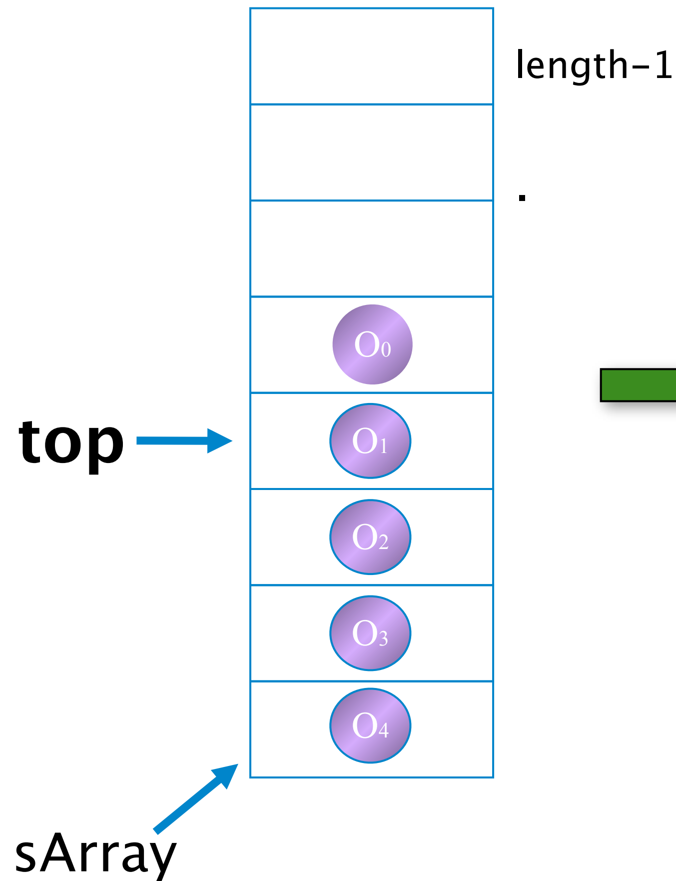
```
private void resizeSArray(){
    E[] temp = (E[]) new Object[sArray.length*2];
    for (int i=0; i<sArray.length; i++){
        temp[i] = sArray[i];
    }
    sArray = temp;
}
```

Die pop-Operation



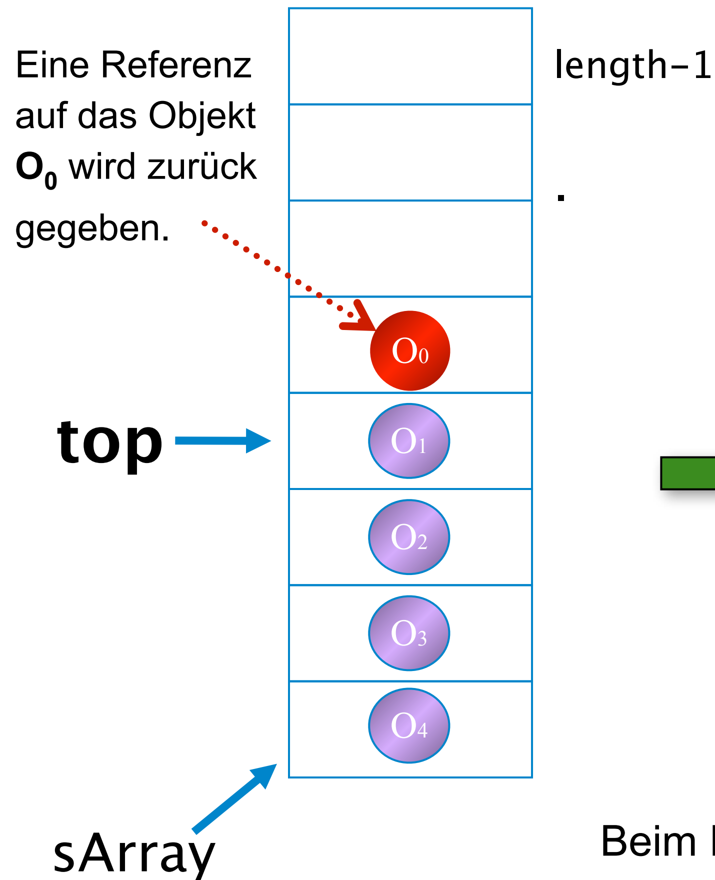

```
public E pop ()
        throws EmptyStackException {
    if ( !empty() ) {
        top--;
        return sArray [ top+1 ];
    } else
        throw new EmptyStackException();
}
```

Die pop-Operation




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

Die **pop**-Operation

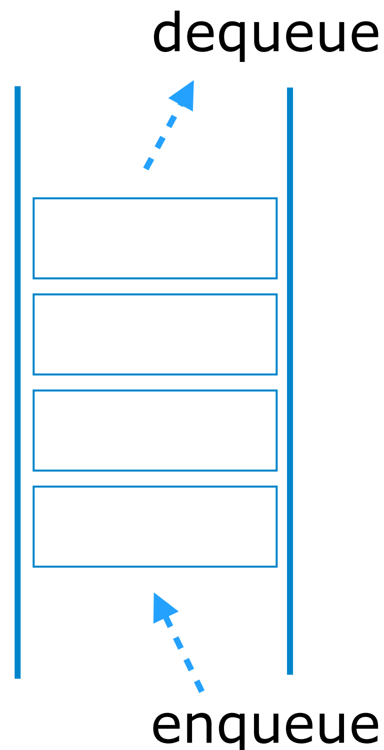



```
public E pop ()
        throws EmptyStackException {
    if ( !empty() ) {
        top--;
        return sArray [ top+1 ];
    } else
        throw new EmptyStackException();
}
```

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

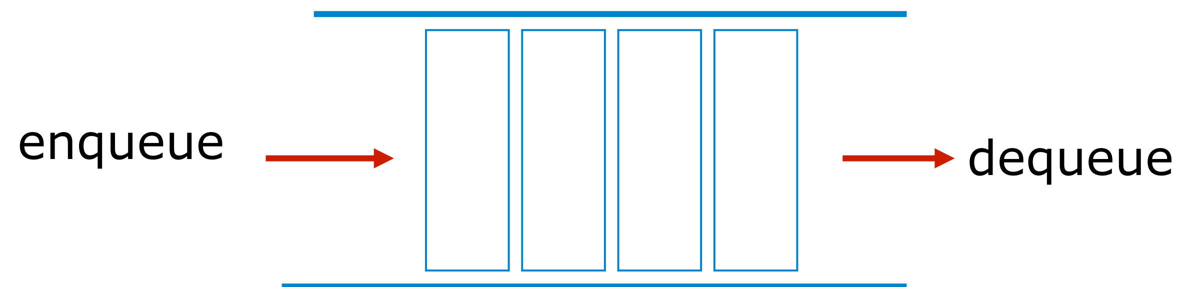
"**F**irst **I**n - **F**irst **O**ut"

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 einer Warteschlange

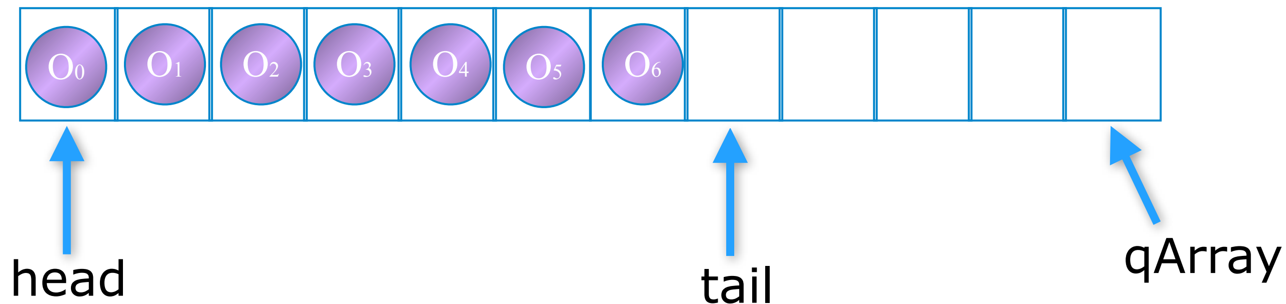
Implementierung einer Warteschlange

Implementierung einer Warteschlange

alle Personen bewegen sich!

keine gute Idee für die Implementierung!

Implementierung der Warteschlange



Erstes Element der Warteschlange

Erster freier Platz in der Warteschlange

Für die Implementierung unserer Warteschlange brauchen wir:

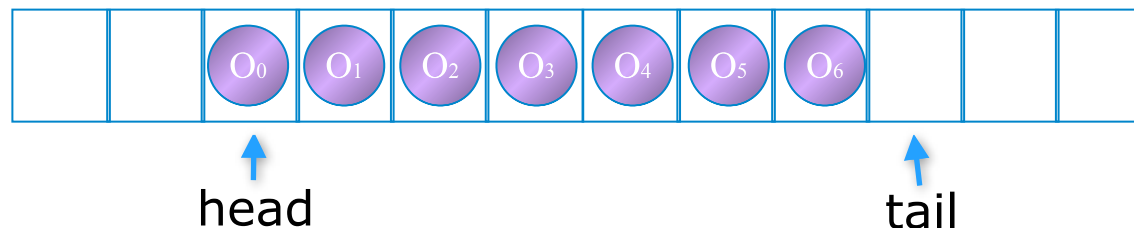
- ein Array (**queue**)
- einen Zeiger (**head**), der immer auf das erste Element zeigt
- einen Zeiger (**tail**), der immer auf den ersten frei verfügbaren Platz der Warteschlange zeigt.

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.

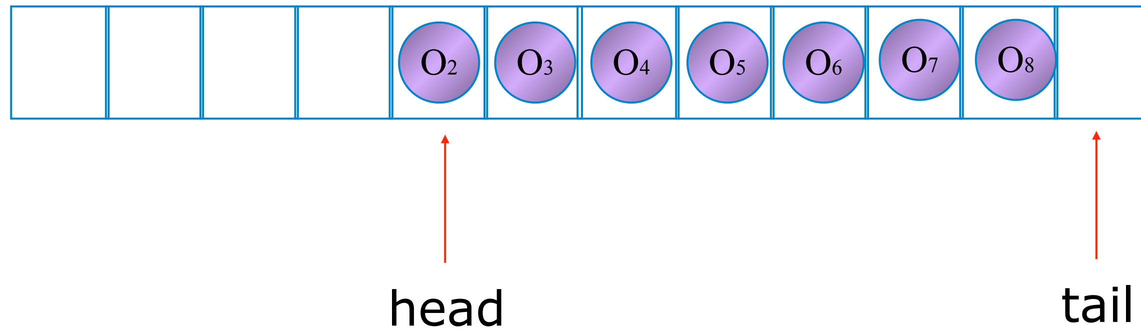
Neue Elemente in der Warteschlange werden in der Position eingefügt, die von **tail** angezeigt wird, und **tail** wird um eine Position nach rechts verschoben.

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



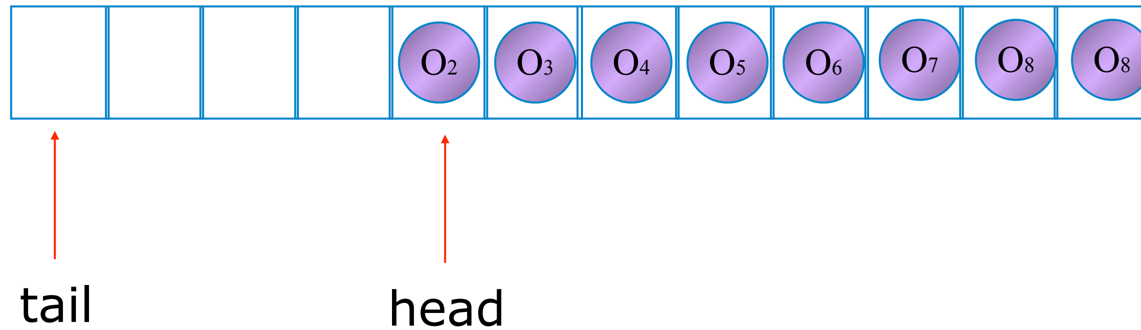
Wraparound-Strategie

enqueue



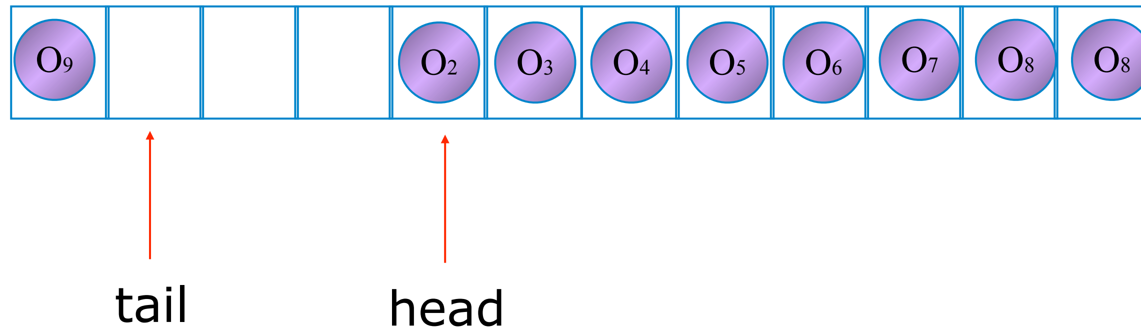
Wraparound-Strategie

enqueue



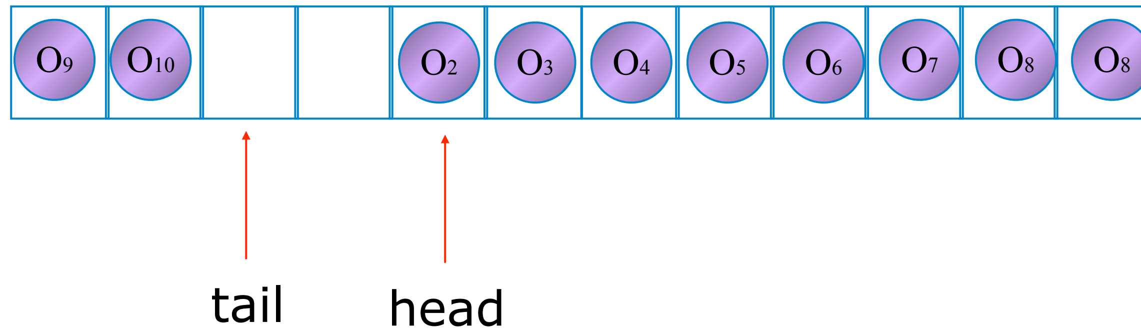
Wraparound-Strategie

enqueue



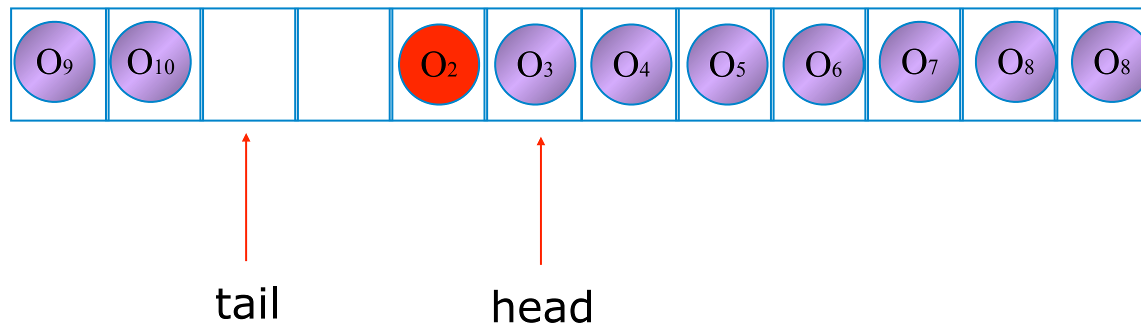
Wraparound-Strategie

enqueue



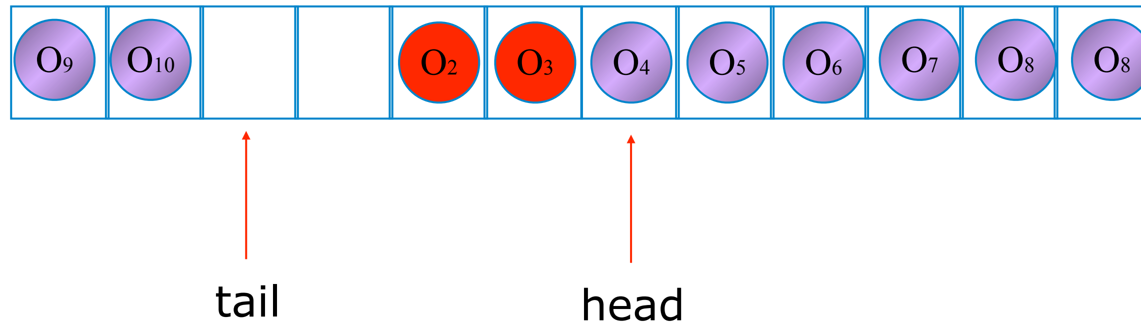
Wraparound-Strategie

dequeue



Wraparound-Strategie

dequeue



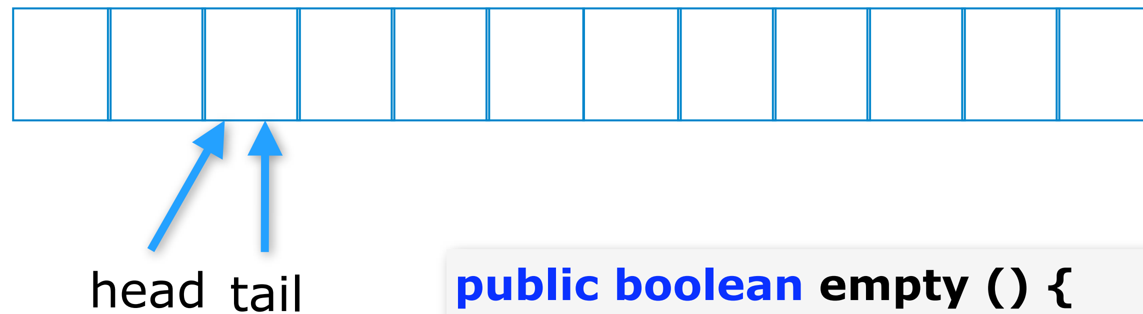
Die Warteschlangen-Schnittstelle

```
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 feste Große!

Die **empty**-Operation

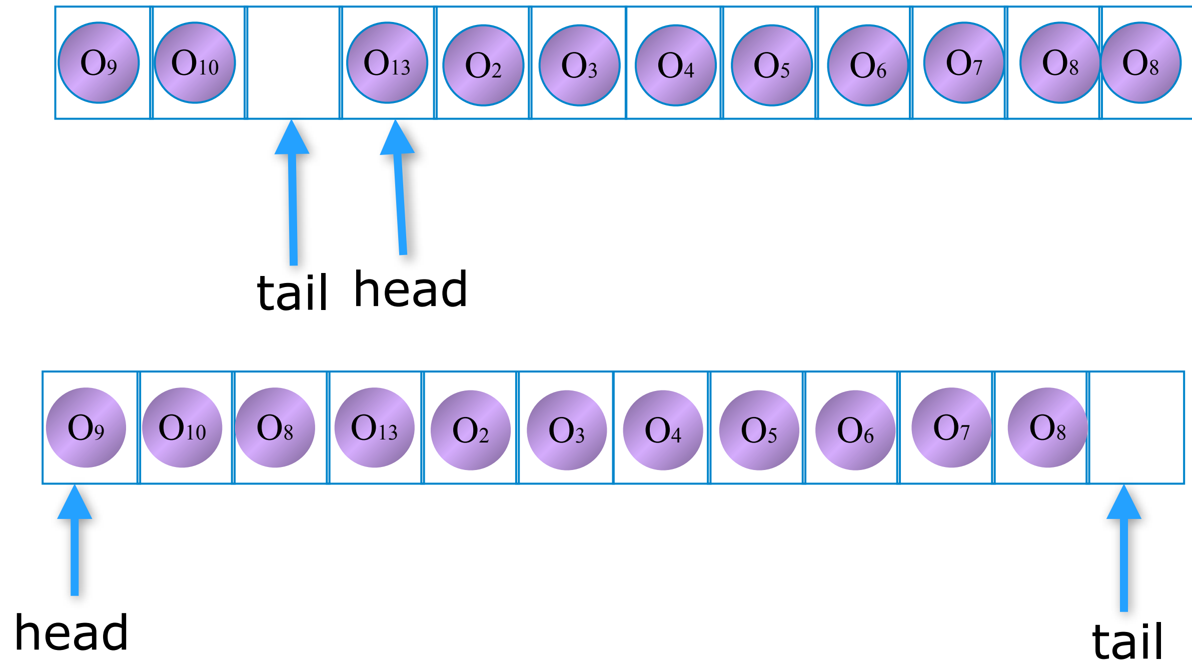
Unsere Warteschlange ist leer, wenn **head** und **tail** auf die gleiche Position des Feldes zeigen.




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

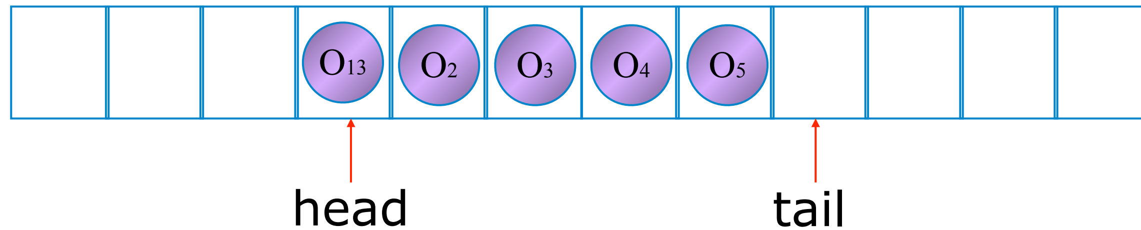
Die **full**-Operation

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 )) ;
}
```

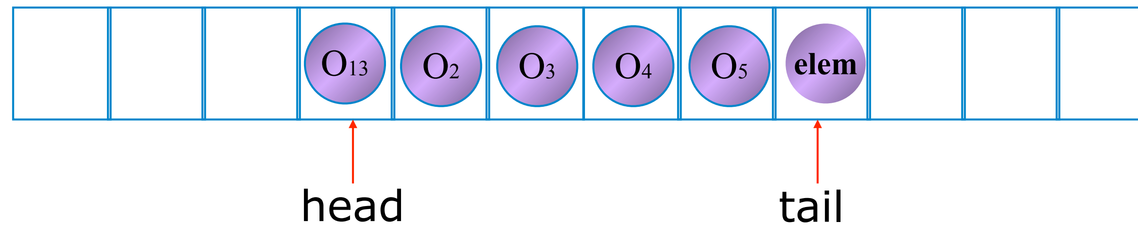
Die enqueue-Operation



Wenn die Warte-schlange nicht voll ist

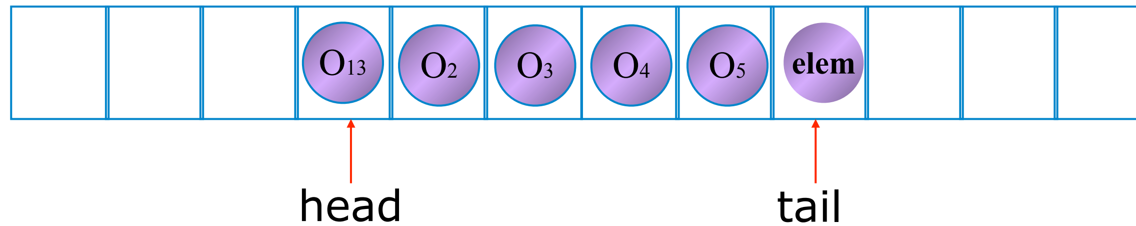
```
public void enqueue ( E elem ) throws FullQueueException {
    if ( !full() ) {
        queue[tail] = elem;
        if ( tail == (queue.length-1) )
            tail = 0;
        else  tail++;
    } else
        throw new FullQueueException();
}
```

Die **enqueue**-Operation

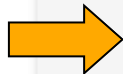



```
public void enqueue ( E elem ) throws FullQueueException {
        if ( !full() ) {
            queue[tail] = elem;
            if ( tail == (queue.length-1) )
                tail = 0;
            else  tail++;
        } else
            throw new FullQueueException();
}
```

Die **enqueue**-Operation

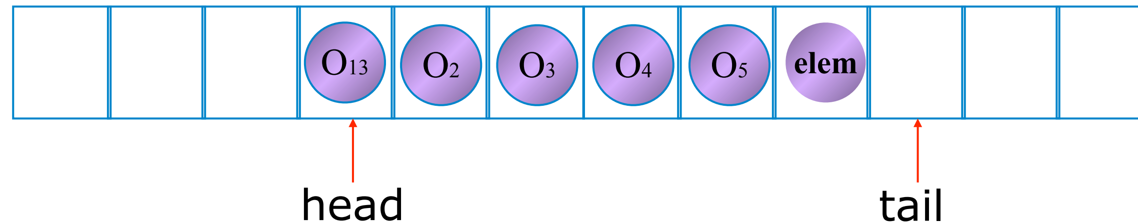


Hier wird geprüft, ob **tail** am Ende des Feldes ist



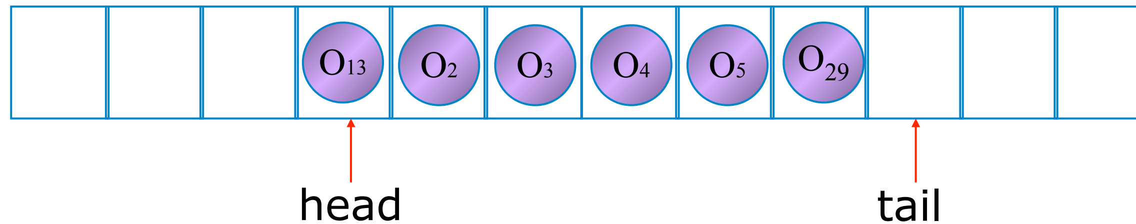
```
public void enqueue ( E elem ) throws FullQueueException {
        if ( !full() ) {
            queue[tail] = elem;
            if ( tail == (queue.length-1) )
                tail = 0;
            else  tail++;
        } else
            throw new FullQueueException();
}
```

Die **enqueue**-Operation

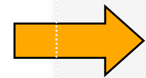



```
public void enqueue ( E elem ) throws FullQueueException {
    if ( !full() ) {
        queue[tail] = elem;
        if ( tail == (queue.length-1) )
            tail = 0;
        else  tail++;
    } else
        throw new FullQueueException();
}
```

Die **dequeue**-Operation

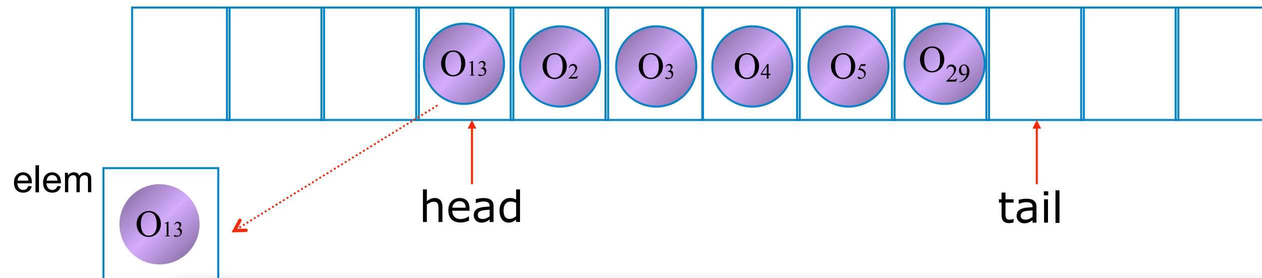


Wenn die Warteschlange nicht leer ist.



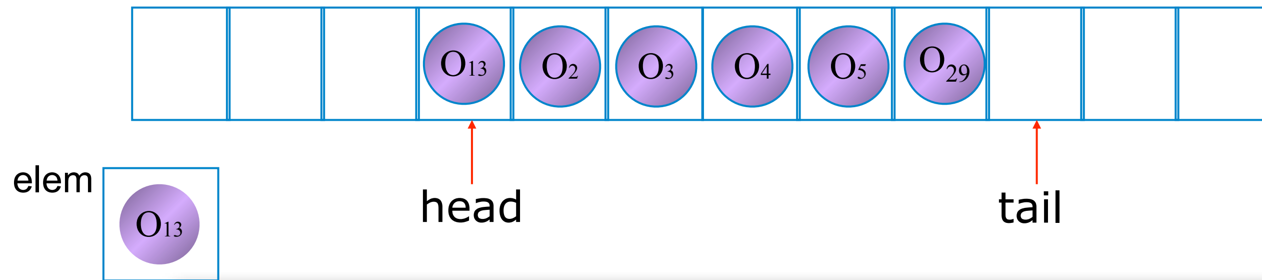
```
public E dequeue() throws EmptyQueueException {
    if ( !empty() ) {
        E elem = queue[head];
        if ( head == (queue.length-1) )
            head = 0;
        else  head++;
        return elem;
    } else
        throw new EmptyQueueException();
}
```

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

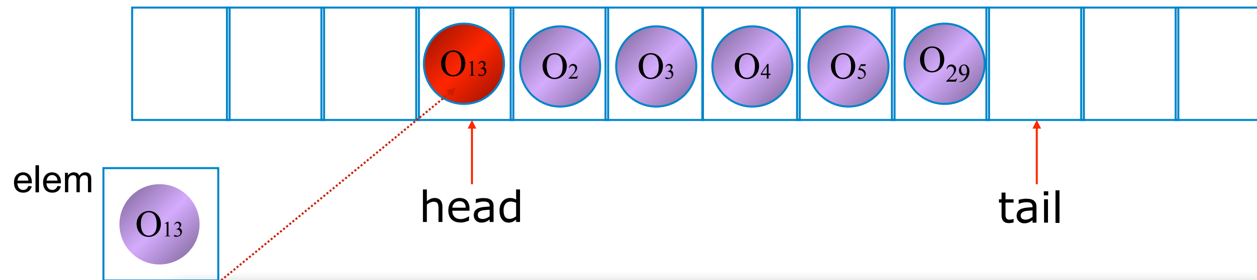


elem

```
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();
}
```

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

Die **dequeue**-Operation

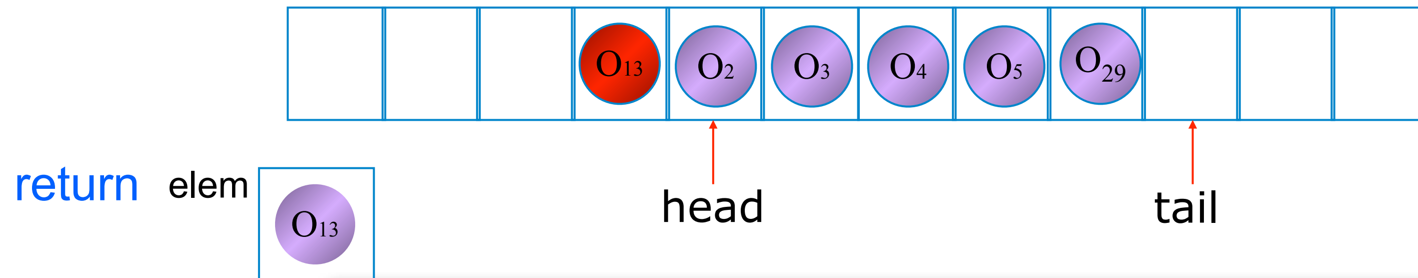


elem

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




```
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

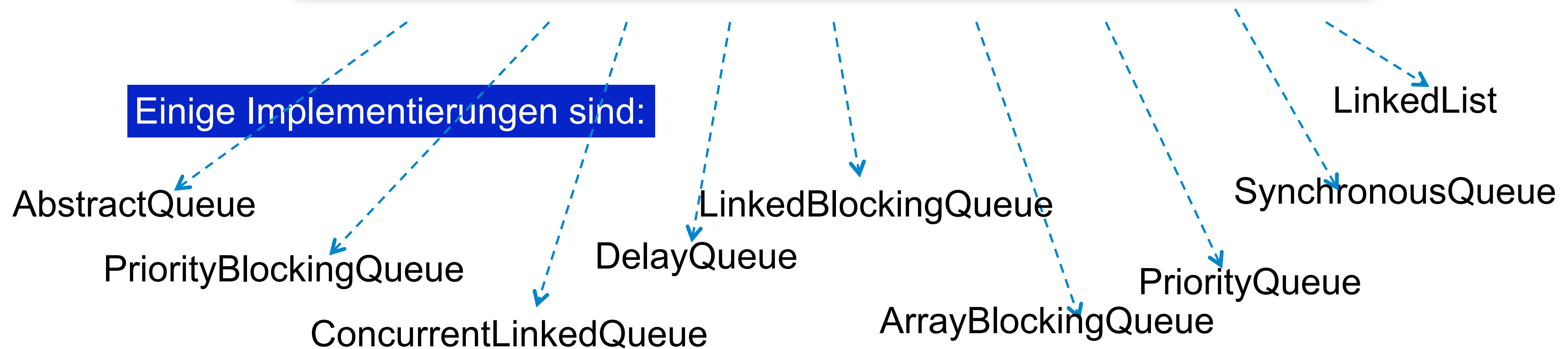
Vector

Stack

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

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

Einige Implementierungen sind:



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 mögliche 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.).