

# Algorithmen und Datenstrukturen

Wintersemester 2022/23

11. Vorlesung

Elementare Datenstrukturen:  
Stapel + Schlange + Liste

# Zur Erinnerung

## Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.



# Zur Erinnerung

## **Datenstruktur:**

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

**Abstrakter Datentyp**

**Implementierung**

# Zur Erinnerung

## **Datenstruktur:**

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

## **Abstrakter Datentyp**

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

## **Implementierung**

# Zur Erinnerung

## Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

## Abstrakter Datentyp

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

## Implementierung

wie wird die gewünschte Funktionalität realisiert:

- wie sind die Daten gespeichert (Feld, Liste, ...)?
- welche Algorithmen implementieren die Operationen?

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.



# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

**Abstrakter Datentyp**

**Implementierung 1**

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

### Implementierung 1



# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

## Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

## Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey



### Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

### Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

$O(1)$  stellt folgende Operationen bereit:  
Insert, FindMax, ExtractMax, IncreaseKey

### Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

$O(1)$  stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

### Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

$O(1)$  stellt folgende Operationen bereit:  $O(n)$   
Insert, FindMax, ExtractMax, IncreaseKey

### Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

### Implementierung 2

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

## Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

## Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps



# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

### Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

## Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

## Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

$O(1)$



### Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey

$O(1)$



### Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps

# Beispiel

## Prioritätsschlange:

verwaltet Elemente einer Menge  $M$ , wobei jedes Element  $x \in M$  eine Priorität  $x.key$  hat.

### Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey  $O(\log n)$

$O(1)$



### Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

Abstrakter Datentyp	

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

ptr Insert(key  $k$ , info  $i$ )

Delete(ptr  $x$ )

ptr Search(key  $k$ )

ptr Minimum()

ptr Maximum()

ptr Predecessor(ptr  $x$ )

ptr Successor(ptr  $x$ )

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

ptr Insert(key  $k$ , info  $i$ )

Delete(ptr  $x$ )

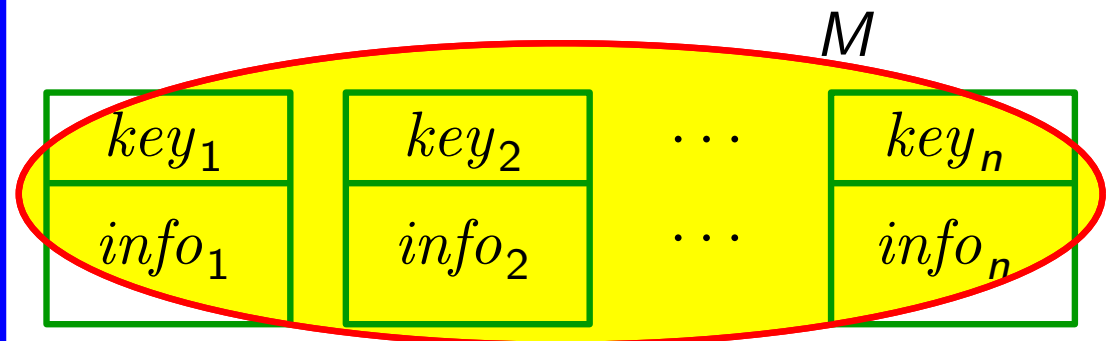
ptr Search(key  $k$ )

ptr Minimum()

ptr Maximum()

ptr Predecessor(ptr  $x$ )

ptr Successor(ptr  $x$ )





# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

ptr Insert(key  $k$ , info  $i$ )

Delete(ptr  $x$ )

ptr Search(key  $k$ )

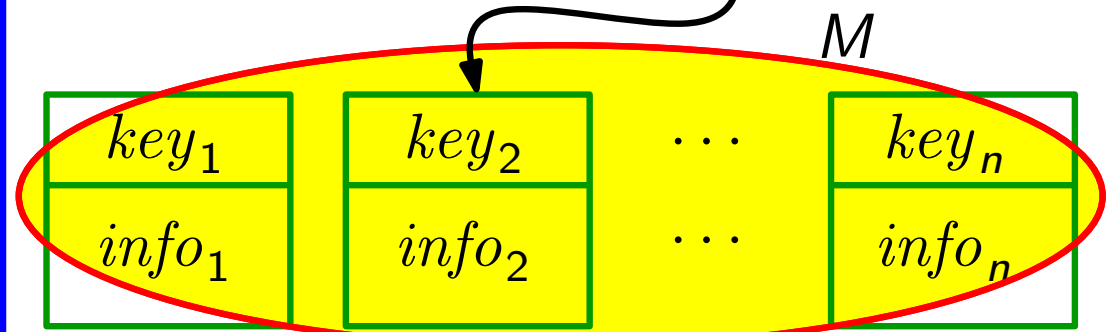
ptr Minimum()

ptr Maximum()

ptr Predecessor(ptr  $x$ )

ptr Successor(ptr  $x$ )

Zeiger (pointer, iterator)  $p$



# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

ptr Insert(key  $k$ , info  $i$ )

Delete(ptr  $x$ )

ptr Search(key  $k$ )

ptr Minimum()

ptr Maximum()

ptr Predecessor(ptr  $x$ )

ptr Successor(ptr  $x$ )

Beispiel für die Anwendung:

**Gib alle Elemente sortiert aus!**



# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

ptr Insert(key  $k$ , info  $i$ )

Delete(ptr  $x$ )

ptr Search(key  $k$ )

ptr Minimum()

ptr Maximum()

ptr Predecessor(ptr  $x$ )

ptr Successor(ptr  $x$ )

Beispiel für die Anwendung:

**Gib alle Elemente sortiert aus!**

```
ptr  $p = M.Minimum()$ 
```

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

ptr Insert(key  $k$ , info  $i$ )

Delete(ptr  $x$ )

ptr Search(key  $k$ )

ptr Minimum()

ptr Maximum()

ptr Predecessor(ptr  $x$ )

ptr Successor(ptr  $x$ )

Beispiel für die Anwendung:

**Gib alle Elemente sortiert aus!**

```
ptr  $p = M.Minimum()$ 
```

```
while  $p \neq nil$  do
```

```
└
```

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

```
ptr Insert(key  $k$ , info  $i$ )  
Delete(ptr  $x$ )  
ptr Search(key  $k$ )  
ptr Minimum()  
ptr Maximum()  
ptr Predecessor(ptr  $x$ )  
ptr Successor(ptr  $x$ )
```

Beispiel für die Anwendung:

**Gib alle Elemente sortiert aus!**

```
ptr  $p = M.Minimum()$   
while  $p \neq nil$  do  
    | gib  $p.key$  aus  
    |  $p = M.Successor(p)$ 
```

# Teil III [CLRS]

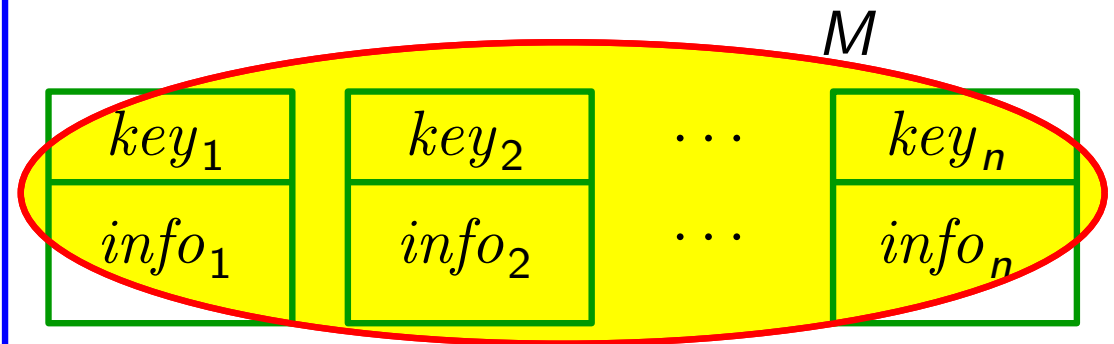


## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

### Abstrakter Datentyp

ptr Insert(key  $k$ , info  $i$ )

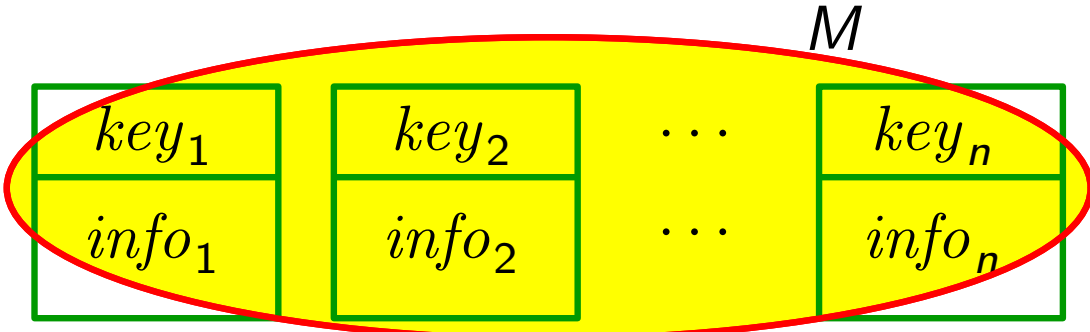


# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ )	 <p style="text-align: right;"><math>M</math></p>

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ )	<ul style="list-style-type: none"><li>● lege neuen Datensatz <math>(k, i)</math> an</li><li>● <math>M = M \cup \{(k, i)\}</math></li><li>● gib Zeiger auf <math>(k, i)</math> zurück</li></ul> <div data-bbox="946 954 2032 1284" style="text-align: center;"><math>M</math> </div>

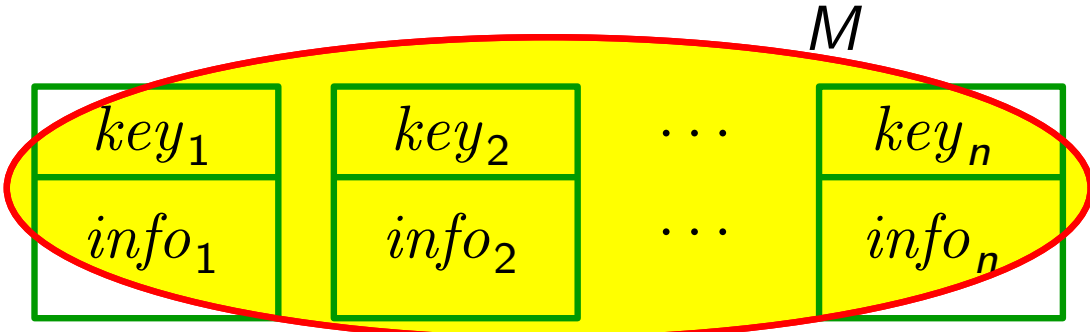


# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ )	

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>								
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ )	<ul style="list-style-type: none"><li data-bbox="953 715 1853 794">• <math>M = M \setminus \{(x.key, x.info)\}</math></li></ul> <div data-bbox="953 954 2034 1289"><p style="text-align: right;"><math>M</math></p><table border="1"><tr><td><math>key_1</math></td><td><math>key_2</math></td><td><math>\dots</math></td><td><math>key_n</math></td></tr><tr><td><math>info_1</math></td><td><math>info_2</math></td><td><math>\dots</math></td><td><math>info_n</math></td></tr></table></div>	$key_1$	$key_2$	$\dots$	$key_n$	$info_1$	$info_2$	$\dots$	$info_n$
$key_1$	$key_2$	$\dots$	$key_n$						
$info_1$	$info_2$	$\dots$	$info_n$						

# Teil III [CLRS]

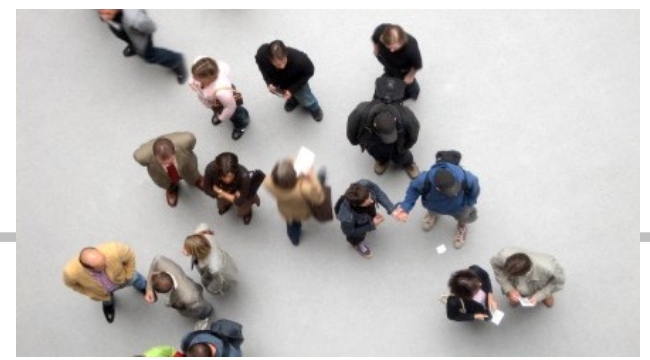


## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ )	<div data-bbox="946 954 2032 1284" style="text-align: center;"><math>M</math> <p>The diagram shows a set <math>M</math> represented as a yellow oval containing several green-bordered boxes. Each box is divided into two horizontal sections: the top section contains a key (<math>key_1, key_2, \dots, key_n</math>) and the bottom section contains information (<math>info_1, info_2, \dots, info_n</math>). Ellipses between the boxes indicate that there can be more than two elements in the set.</p></div>

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ )	<ul style="list-style-type: none"><li>• falls vorhanden, gib Zeiger <math>p</math> mit <math>p.key = k</math> zurück</li><li>• sonst gib Zeiger <math>nil</math> zurück</li></ul>

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ ) ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ ) ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ ) ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	<ul style="list-style-type: none"><li>• sei <math>M' = \{(k, i) \in M \mid k &lt; x.key\}</math></li><li>• falls <math>M' = \emptyset</math>, gib <i>nil</i> zurück,</li><li>• sonst gib Zeiger auf <math>(k^*, i^*)</math> zurück, wobei <math>k^* = \max_{(k,i) \in M'} k</math></li></ul>

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ ) ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	<ul style="list-style-type: none"><li>• sei <math>M' = \{(k, i) \in M \mid k &lt; x.key\}</math></li><li>• falls <math>M' = \emptyset</math>, gib <i>nil</i> zurück,</li><li>• sonst gib Zeiger auf <math>(k^*, i^*)</math> zurück, wobei <math>k^* = \max_{(k,i) \in M'} k</math></li></ul>



# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ ) ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ )	
ptr Search(key $k$ ) ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>
<code>ptr Insert(key <math>k</math>, info <math>i</math>)</code> <code>Delete(ptr <math>x</math>)</code>	} Änderungen
<code>ptr Search(key <math>k</math>)</code> <code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr <math>x</math>)</code> <code>ptr Successor(ptr <math>x</math>)</code>	} Anfragen

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

<b>Abstrakter Datentyp</b>	<i>Funktionalität</i>	
ptr Insert(key $k$ , info $i$ ) Delete(ptr $x$ ) ptr Search(key $k$ )	} Änderungen	} <b>Wörterbuch</b>
ptr Minimum() ptr Maximum() ptr Predecessor(ptr $x$ ) ptr Successor(ptr $x$ )	} Anfragen	

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key <math>k</math>, info <math>i</math>)</code> <code>Delete(ptr <math>x</math>)</code> <code>ptr Search(key <math>k</math>)</code>	} Änderungen
<code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr <math>x</math>)</code> <code>ptr Successor(ptr <math>x</math>)</code>	} Anfragen
	} <b>Wörterbuch</b>

**Implementierung:** je nachdem...

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

Abstrakter Datentyp	Funktionalität	
<code>ptr Insert(key <math>k</math>, info <math>i</math>)</code> <code>Delete(ptr <math>x</math>)</code> <code>ptr Search(key <math>k</math>)</code>	} Änderungen	} <b>Wörterbuch</b>
<code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr <math>x</math>)</code> <code>ptr Successor(ptr <math>x</math>)</code>	} Anfragen	

**Implementierung:** je nachdem... Drei Beispiele!

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



**Abstr. Datentyp**

**Implementierung**

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

„last-in first-out“



**Abstr. Datentyp**

**Implementierung**



# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

boolean Empty()

Push(key *k*)

key Pop()

key Top()

## Implementierung

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

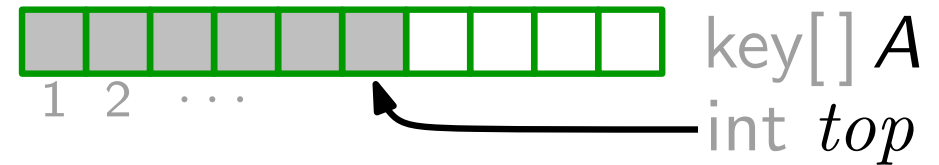
boolean Empty()

Push(key  $k$ )

key Pop()

key Top()

## Implementierung



# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

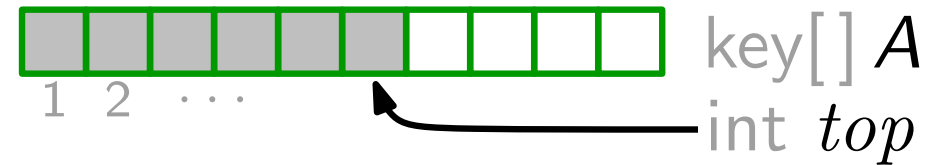
boolean Empty()

Push(key *k*)

key Pop()

key Top()

## Implementierung



**if** *top* == 0 **then return true**  
**else return false**

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

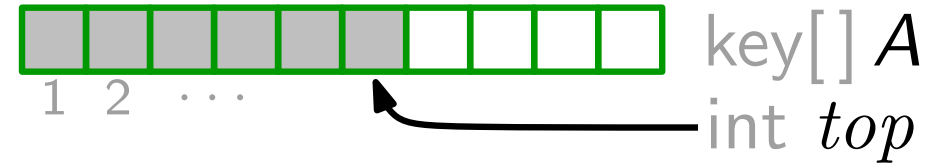
boolean Empty()

Push(key  $k$ )

key Pop()

key Top()

## Implementierung



**if**  $top == 0$  **then return** *true*  
**else return** *false*

$top = top + 1$

$A[top] = k$

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

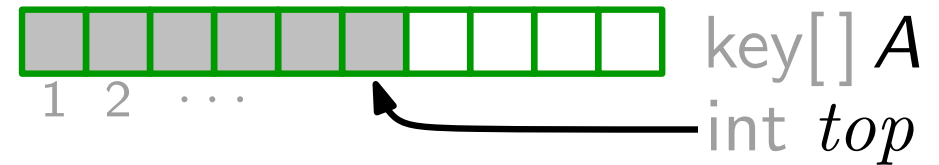
boolean Empty()

Push(key  $k$ )

key Pop()

key Top()

## Implementierung



**if**  $top == 0$  **then return** *true*  
**else return** *false*

$top = top + 1$

$A[top] = k$

### Aufgabe:

Schreiben Sie Pseudocode, der das oberste Element vom Stapel nimmt und zurückgibt!

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

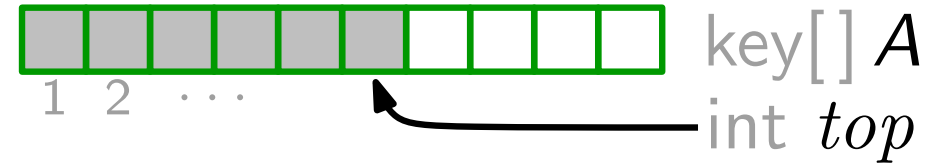
boolean Empty()

Push(key  $k$ )

key Pop()

key Top()

## Implementierung



**if**  $top == 0$  **then return** *true*  
**else return** *false*

$top = top + 1$   
 $A[top] = k$

$top = top - 1$   
**return**  $A[top + 1]$

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

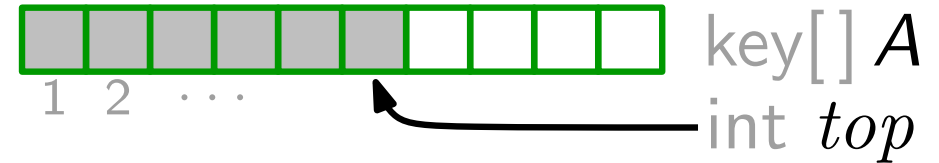
boolean Empty()

Push(key  $k$ )

key Pop()

key Top()

## Implementierung



**if**  $top == 0$  **then return** *true*  
**else return** *false*

$top = top + 1$   
 $A[top] = k$

**if** Empty() **then error** „underflow“  
**else**  
     $top = top - 1$   
    **return**  $A[top + 1]$

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

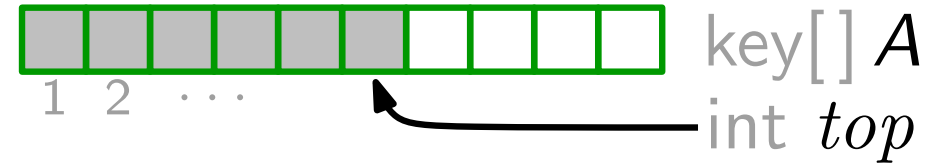
boolean Empty()

Push(key  $k$ )

key Pop()

key Top()

## Implementierung



**if**  $top == 0$  **then return** *true*  
**else return** *false*

$top = top + 1$   
 $A[top] = k$

**if** Empty() **then error** „underflow“  
**else**

$top = top - 1$   
    **return**  $A[top + 1]$

**if** Empty() **then ... else return**  $A[top]$



# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

boolean Empty()

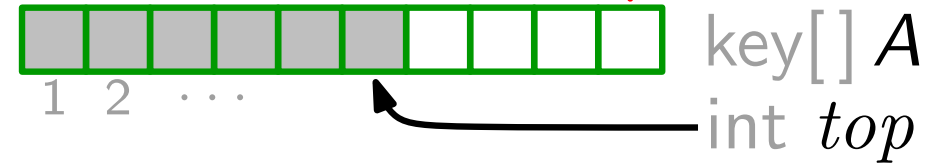
Push(key  $k$ )

key Pop()

key Top()

## Implementierung

Größe?



**if**  $top == 0$  **then return true**  
**else return false**

$top = top + 1$   
 $A[top] = k$

**if** Empty() **then error** „underflow“  
**else**

$top = top - 1$   
    **return**  $A[top + 1]$

**if** Empty() **then ... else return**  $A[top]$

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
		key[] <i>A</i> int <i>top</i>
boolean Empty()	<b>if</b> <i>top</i> == 0 <b>then return true</b> <b>else return false</b>	
Push(key <i>k</i> )	<i>top</i> = <i>top</i> + 1 <i>A</i> [ <i>top</i> ] = <i>k</i>	
key Pop()	<b>if</b> Empty() <b>then error</b> „underflow“ <b>else</b>   <i>top</i> = <i>top</i> - 1   <b>return</b> <i>A</i> [ <i>top</i> + 1]	
key Top()	<b>if</b> Empty() <b>then ... else return</b> <i>A</i> [ <i>top</i> ]	

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )		key[] $A$ int $top$
boolean Empty()	<b>if</b> $top == 0$ <b>then return</b> <i>true</i> <b>else return</b> <i>false</i>	
Push(key $k$ )	$top = top + 1$ $A[top] = k$	
key Pop()	<b>if</b> Empty() <b>then error</b> „underflow“ <b>else</b> $top = top - 1$ <b>return</b> $A[top + 1]$	
key Top()	<b>if</b> Empty() <b>then ... else return</b> $A[top]$	

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )	$A = \mathbf{new}$ key[1.. $n$ ] $top = 0$	key[] $A$ int $top$
boolean Empty()	<b>if</b> $top == 0$ <b>then return</b> <i>true</i> <b>else return</b> <i>false</i>	
Push(key $k$ )	$top = top + 1$ $A[top] = k$	
key Pop()	<b>if</b> Empty() <b>then error</b> „underflow“ <b>else</b> $top = top - 1$ <b>return</b> $A[top + 1]$	
key Top()	<b>if</b> Empty() <b>then ... else return</b> $A[top]$	

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )	$A = \mathbf{new}$ key[1.. $n$ ] $top = 0$	key[] $A$ int $top$
boolean Empty()	<b>if</b> $top == 0$ <b>then return</b> <i>true</i> <b>else return</b> <i>false</i>	
Push(key $k$ )	$top = top + 1$ { <b>if</b> $top > A.length$ <b>then</b> $A[top] = k$ { <b>error</b> „overflow“	
key Pop()	<b>if</b> Empty() <b>then error</b> „underflow“ <b>else</b>   $top = top - 1$   <b>return</b> $A[top + 1]$	
key Top()	<b>if</b> Empty() <b>then ... else return</b> $A[top]$	

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )	$A = \mathbf{new}$ key[1.. $n$ ] $top = 0$	key[] $A$ int $top$
boolean Empty()	<b>if</b> $top == 0$ <b>then return</b> <i>true</i> <b>else return</b> <i>false</i>	
Push(key $k$ )	$top = top + 1$ { <b>if</b> $top > A.length$ <b>then</b> $A[top] = k$ { <b>error</b> „overflow“	
key Pop()	<b>if</b> Empty() <b>then error</b> „underflow“ <b>else</b>   $top = top - 1$   <b>return</b> $A[top + 1]$	
key Top()	<b>if</b> Empty() <b>then ... else return</b> $A[top]$	

**Laufzeiten?**

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )	$A = \mathbf{new}^* \text{key}[1..n]$ $top = 0$	$\text{key}[]$ $A$ $\text{int } top$
boolean Empty()	<b>if</b> $top == 0$ <b>then return</b> <i>true</i> <b>else return</b> <i>false</i>	
Push(key $k$ )	$top = top + 1$ { <b>if</b> $top > A.length$ <b>then</b> $A[top] = k$ { <b>error</b> „overflow“	
key Pop()	<b>if</b> Empty() <b>then error</b> „underflow“ <b>else</b> $top = top - 1$ <b>return</b> $A[top + 1]$	
key Top()	<b>if</b> Empty() <b>then ... else return</b> $A[top]$	

**Laufzeiten?**

Alle\*  $O(1)$ ,  
d.h. konstant.

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )	key[] $A$ int $top$	
boolean Empty()		
Push(key $k$ )		
key Pop()		
key Top()		



# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )	} <i>Konstruktor</i>	Attribute { key[] $A$ int $top$
boolean Empty()		} <i>Methoden</i>
Push(key $k$ )		
key Pop()		
key Top()		

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int $n$ )	} <i>Konstruktor</i>	Attribute { key[] $A$ int $top$
boolean Empty()		} <i>Methoden</i>
Push(key $k$ )		
key Pop()		
key Top()		

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

Stack(int  $n$ )

boolean Empty()

Push(key  $k$ )

key Pop()

key Top()

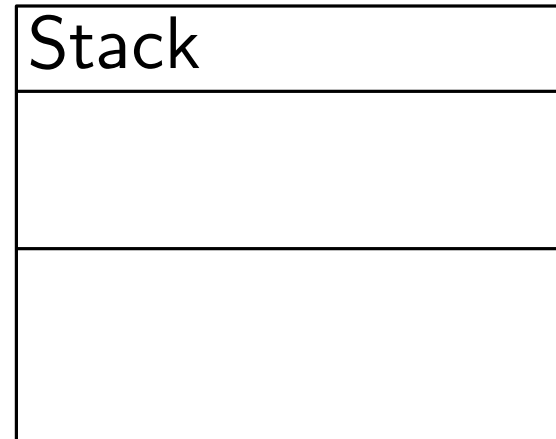
## Implementierung

$\left. \begin{array}{l} \text{Konstruktor} \\ \text{Attribute} \end{array} \right\} \begin{array}{l} \text{key[]} A \\ \text{int } top \end{array}$

$\left. \begin{array}{l} \text{Methoden} \end{array} \right\}$

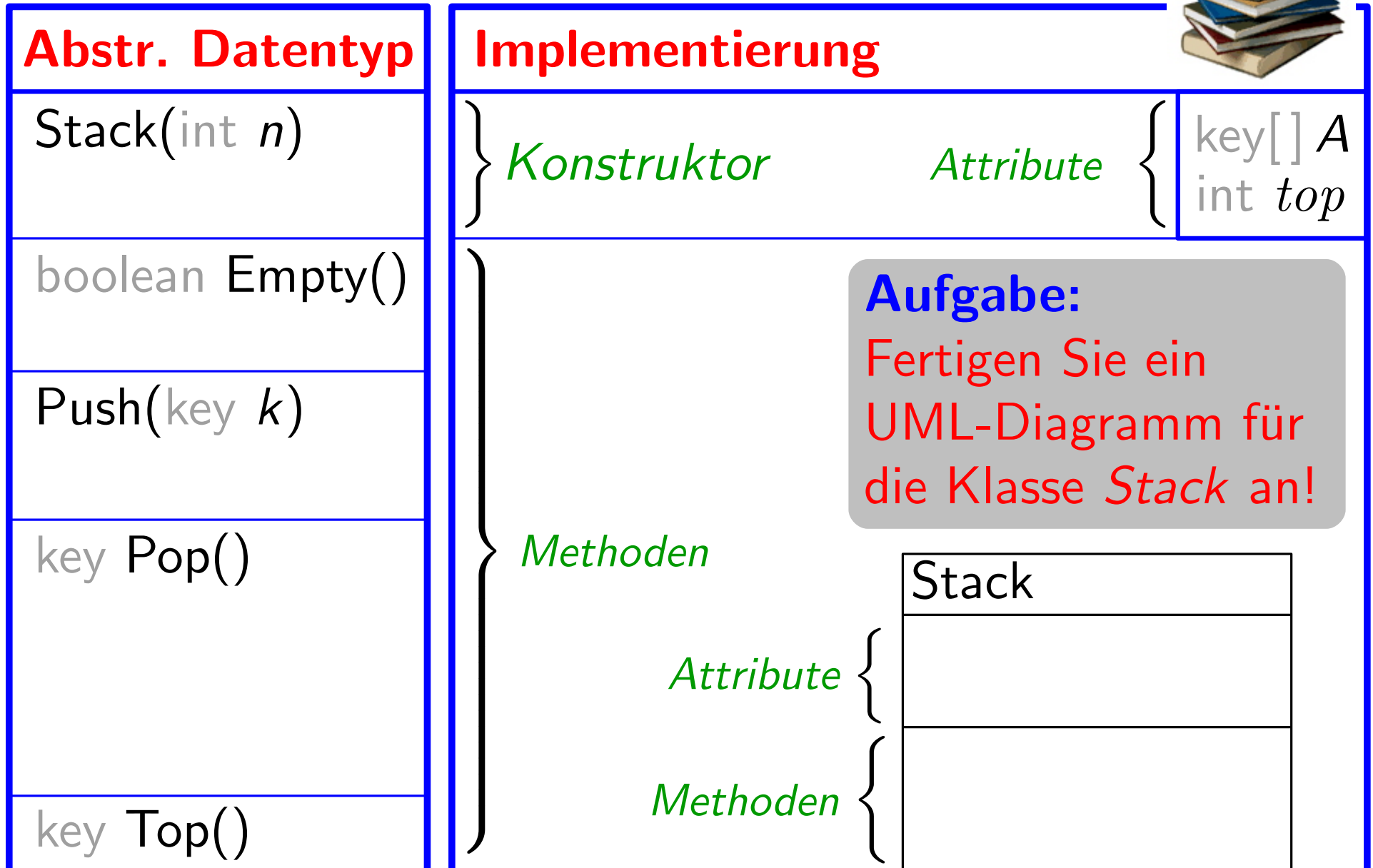
### Aufgabe:

Fertigen Sie ein UML-Diagramm für die Klasse *Stack* an!



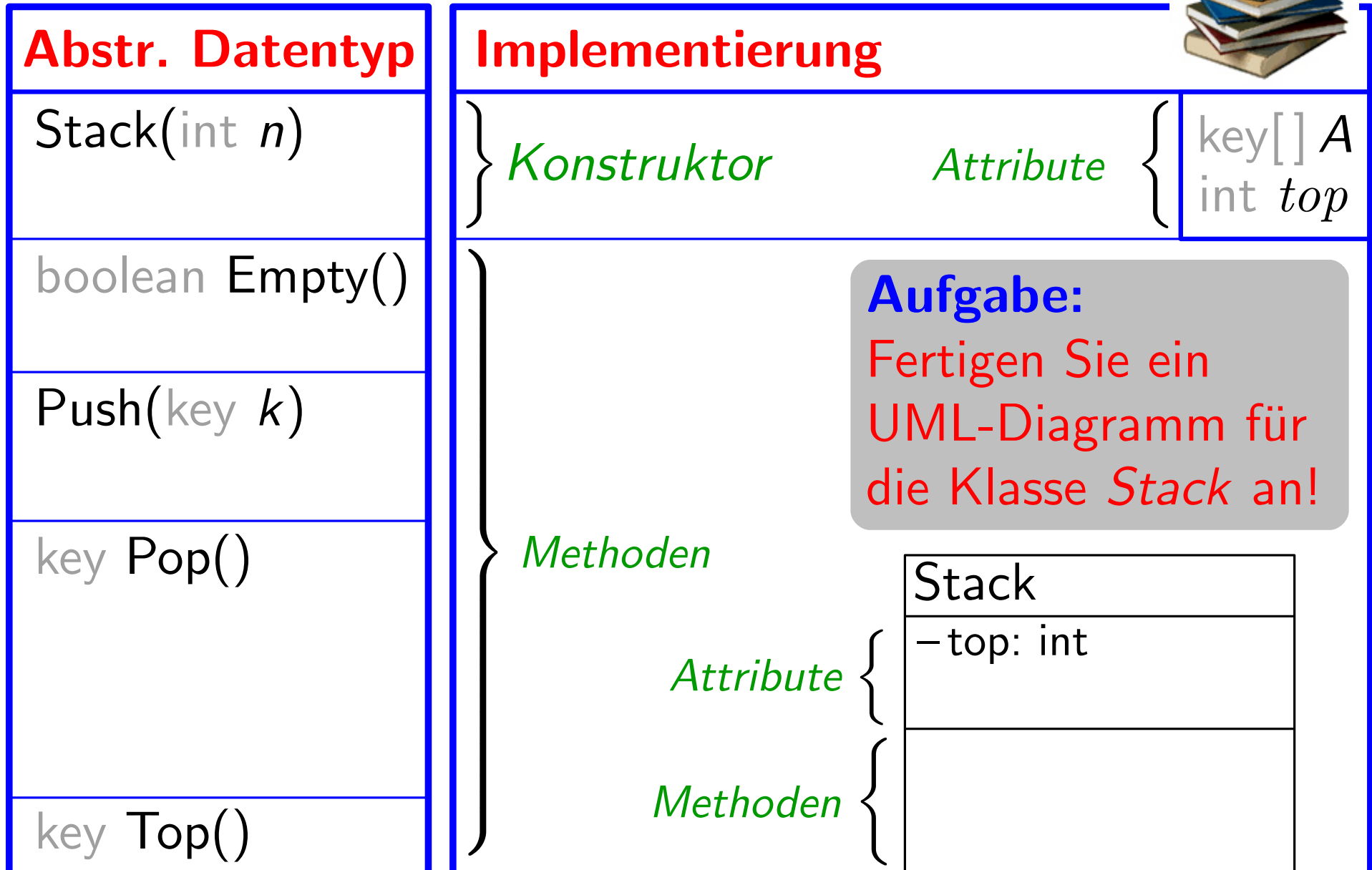
# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



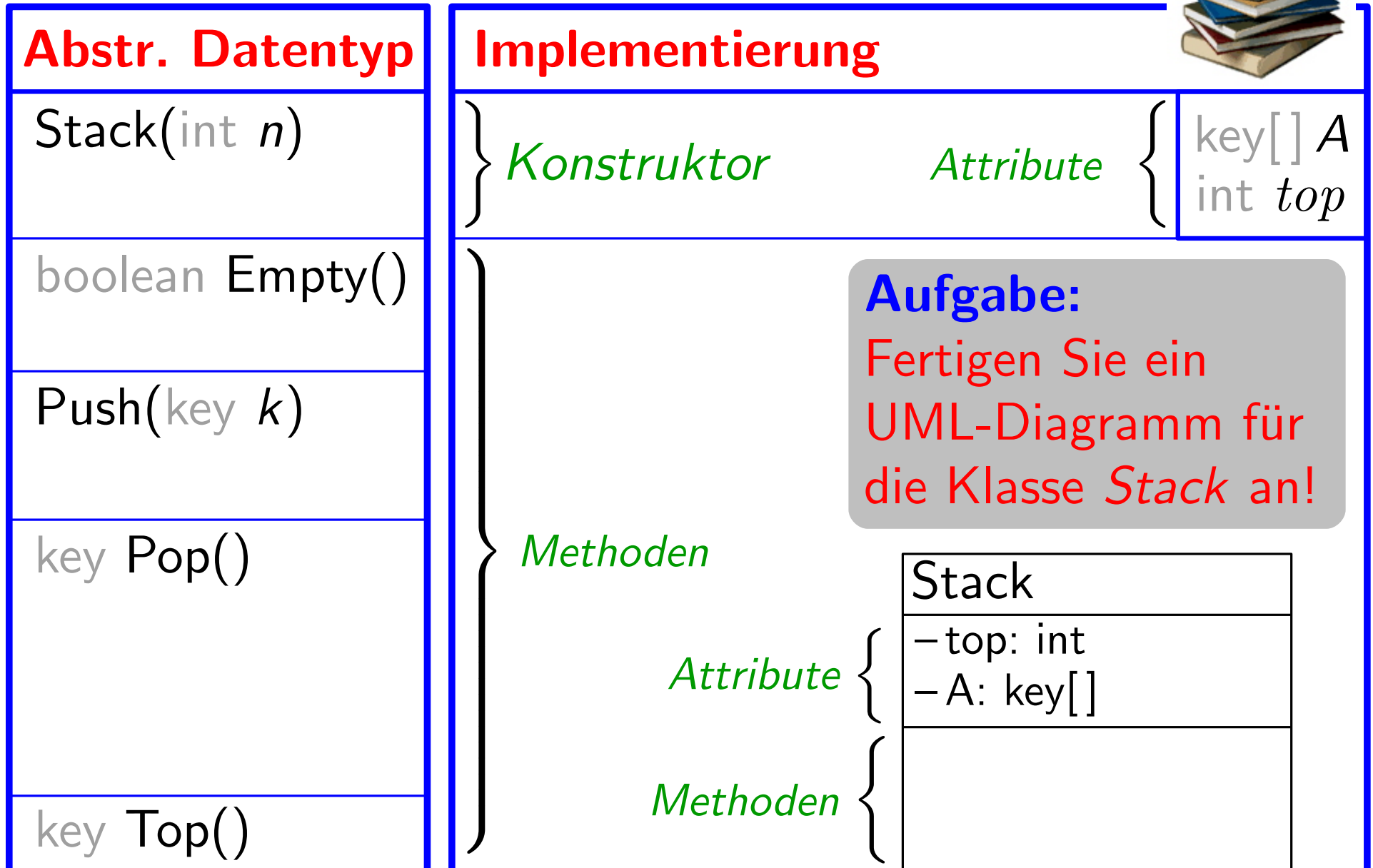
# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



# I. Stapel

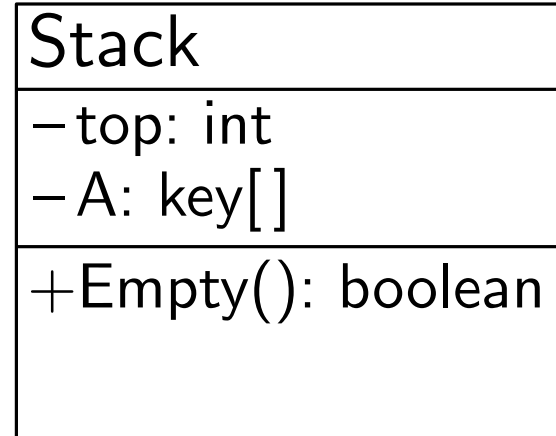
verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung									
Stack(int <i>n</i> )	} <i>Konstruktor</i>	<table border="0"> <tr> <td data-bbox="1400 440 1783 643"><i>Attribute</i></td> <td data-bbox="1783 440 2051 643"> <table border="0"> <tr> <td data-bbox="1783 440 1825 643">{</td> <td data-bbox="1825 440 2051 643">key[] <i>A</i> int <i>top</i></td> </tr> </table> </td> </tr> </table>	<i>Attribute</i>	<table border="0"> <tr> <td data-bbox="1783 440 1825 643">{</td> <td data-bbox="1825 440 2051 643">key[] <i>A</i> int <i>top</i></td> </tr> </table>	{	key[] <i>A</i> int <i>top</i>				
<i>Attribute</i>	<table border="0"> <tr> <td data-bbox="1783 440 1825 643">{</td> <td data-bbox="1825 440 2051 643">key[] <i>A</i> int <i>top</i></td> </tr> </table>	{	key[] <i>A</i> int <i>top</i>							
{	key[] <i>A</i> int <i>top</i>									
boolean Empty()	<table border="0"> <tr> <td data-bbox="768 643 1400 829">}</td> <td data-bbox="1400 643 2051 829"><i>Methoden</i></td> </tr> </table>		}	<i>Methoden</i>						
}			<i>Methoden</i>							
Push(key <i>k</i> )										
key Pop()										
key Top()	}	<table border="0"> <tr> <td data-bbox="1400 1422 1783 1524"><i>Attribute</i></td> <td data-bbox="1783 1422 2051 1524"> <table border="0"> <tr> <td data-bbox="1783 1422 1825 1524">{</td> <td data-bbox="1825 1422 2051 1524">-top: int -A: key[]</td> </tr> </table> </td> </tr> <tr> <td data-bbox="1400 1422 1783 1524"><i>Methoden</i></td> <td data-bbox="1783 1422 2051 1524"> <table border="0"> <tr> <td data-bbox="1783 1422 1825 1524">{</td> <td data-bbox="1825 1422 2051 1524">+Empty(): boolean</td> </tr> </table> </td> </tr> </table>	<i>Attribute</i>	<table border="0"> <tr> <td data-bbox="1783 1422 1825 1524">{</td> <td data-bbox="1825 1422 2051 1524">-top: int -A: key[]</td> </tr> </table>	{	-top: int -A: key[]	<i>Methoden</i>	<table border="0"> <tr> <td data-bbox="1783 1422 1825 1524">{</td> <td data-bbox="1825 1422 2051 1524">+Empty(): boolean</td> </tr> </table>	{	+Empty(): boolean
<i>Attribute</i>	<table border="0"> <tr> <td data-bbox="1783 1422 1825 1524">{</td> <td data-bbox="1825 1422 2051 1524">-top: int -A: key[]</td> </tr> </table>	{	-top: int -A: key[]							
{	-top: int -A: key[]									
<i>Methoden</i>	<table border="0"> <tr> <td data-bbox="1783 1422 1825 1524">{</td> <td data-bbox="1825 1422 2051 1524">+Empty(): boolean</td> </tr> </table>	{	+Empty(): boolean							
{	+Empty(): boolean									

## Aufgabe:

Fertigen Sie ein UML-Diagramm für die Klasse *Stack* an!



# I. Stapel

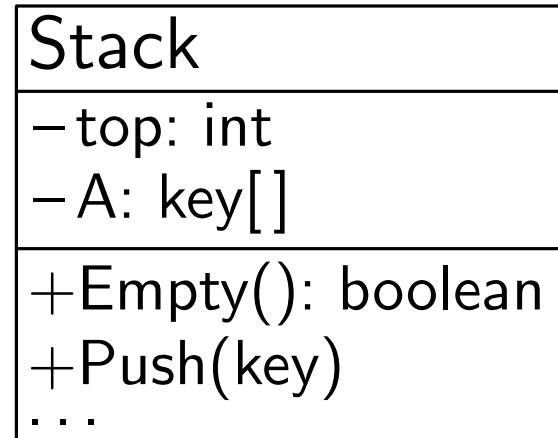
verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung
Stack(int <i>n</i> )	<div style="display: flex; align-items: center;"> <div style="margin-right: 20px;">} <i>Konstruktor</i></div> <div style="margin-right: 20px;"><i>Attribute</i></div> <div style="font-size: 2em;">}</div> <div style="border-left: 1px solid black; padding-left: 5px;">           key[] <i>A</i>            int <i>top</i> </div> </div>
boolean Empty()	<div style="display: flex; align-items: center;"> <div style="font-size: 4em; margin-right: 10px;">}</div> <div style="margin-right: 20px;"><i>Methoden</i></div> </div>
Push(key <i>k</i> )	
key Pop()	
key Top()	<div style="display: flex; align-items: center;"> <div style="margin-right: 20px;"><i>Attribute</i></div> <div style="font-size: 2em; margin-right: 10px;">}</div> <div style="border-left: 1px solid black; padding-left: 5px;">           - top: int            - A: key[]         </div> </div> <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="margin-right: 20px;"><i>Methoden</i></div> <div style="font-size: 2em; margin-right: 10px;">}</div> <div style="border-left: 1px solid black; padding-left: 5px;">           + Empty(): boolean            + Push(key)            ...         </div> </div>

## Aufgabe:

Fertigen Sie ein UML-Diagramm für die Klasse *Stack* an!





## II. Schlange

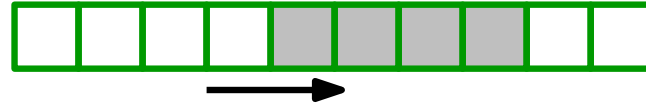
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



**Abs. Datentyp**

**Implementierung**

## II. Schlange



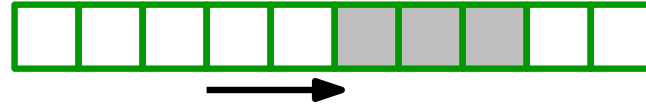
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



**Abs. Datentyp**

**Implementierung**

## II. Schlange



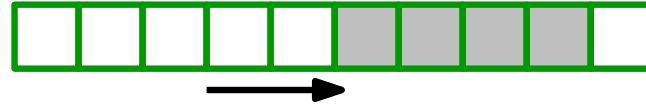
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



**Abs. Datentyp**

**Implementierung**

## II. Schlange



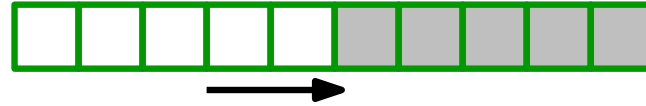
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



**Abs. Datentyp**

**Implementierung**

## II. Schlange



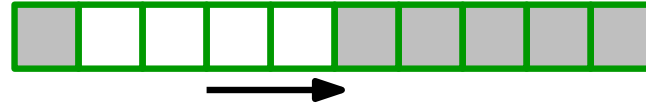
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



**Abs. Datentyp**

**Implementierung**

## II. Schlange



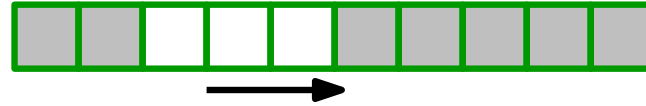
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



**Abs. Datentyp**

**Implementierung**

## II. Schlange



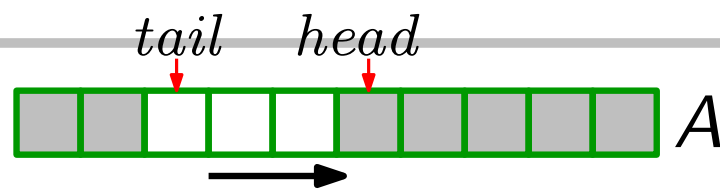
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



**Abs. Datentyp**

**Implementierung**

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



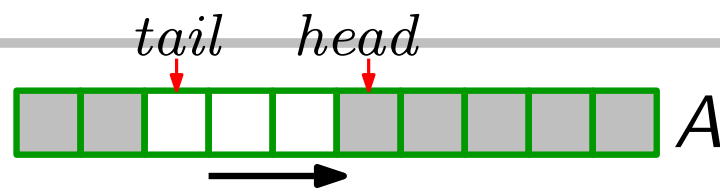
### Abs. Datentyp

### Implementierung

```
key[] A  
int tail  
int head
```



## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



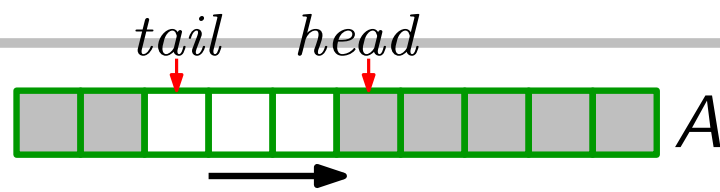
### Abs. Datentyp

Queue(int *n*)

### Implementierung

key[] *A*  
int *tail*  
int *head*

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

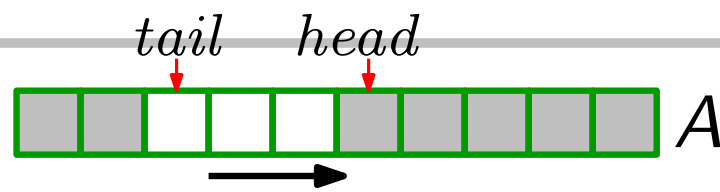
Queue(int *n*)

### Implementierung

$A = \mathbf{new}$  key[1..*n*]  
 $tail = head = 1$

key[] *A*  
int *tail*  
int *head*

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int *n*)

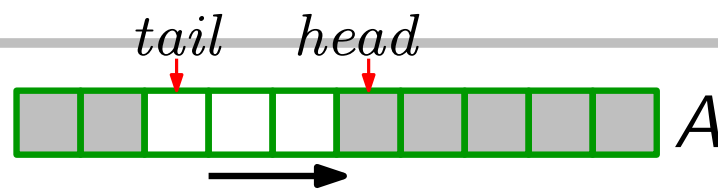
boolean Empty()

### Implementierung

$A = \mathbf{new\ key}[1..n]$   
 $tail = head = 1$

key[] *A*  
int *tail*  
int *head*

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int *n*)

boolean Empty()

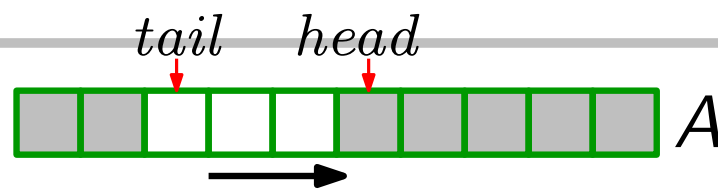
### Implementierung

$A = \text{new key}[1..n]$   
 $tail = head = 1$

key[] *A*  
int *tail*  
int *head*

**if**  $head == tail$  **then return true**  
**else return false**

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int *n*)

boolean Empty()

Enqueue(key *k*)  
*stell neues Element  
an den Schwanz der  
Schlange an*

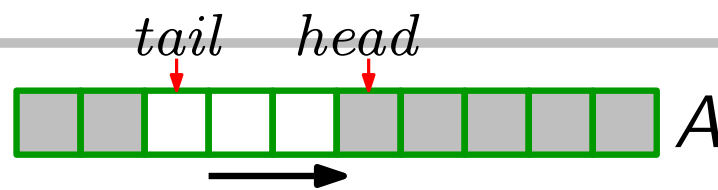
### Implementierung

$A = \text{new key}[1..n]$   
 $tail = head = 1$

key[] *A*  
int *tail*  
int *head*

**if** *head* == *tail* **then return true**  
**else return false**

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int *n*)

boolean Empty()

Enqueue(key *k*)  
*stell neues Element  
an den Schwanz der  
Schlange an*

### Implementierung

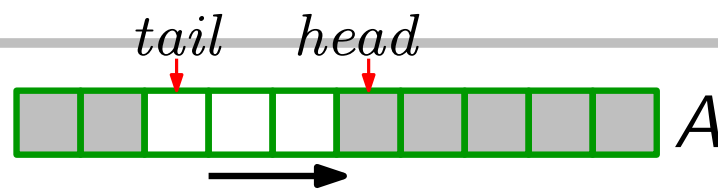
$A = \text{new key}[1..n]$   
 $tail = head = 1$

key[] *A*  
int *tail*  
int *head*

**if**  $head == tail$  **then return true**  
**else return false**

$A[tail] = k$   
**if**  $tail == A.length$  **then**  $tail = 1$   
**else**  $tail = tail + 1$

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int  $n$ )

boolean Empty()

Enqueue(key  $k$ )

*stell neues Element  
an den Schwanz der  
Schlange an*

key Dequeue()

*entnimmt Element am  
Kopf der Schlange*

### Implementierung

$A = \text{new key}[1..n]$   
 $tail = head = 1$

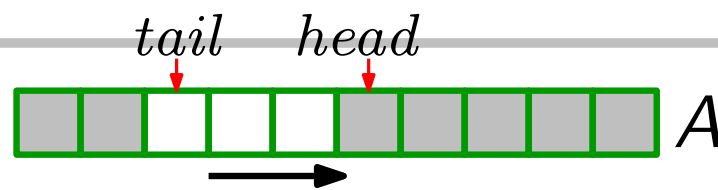
key[]  $A$   
int  $tail$   
int  $head$

**if**  $head == tail$  **then return true**  
**else return false**

$A[tail] = k$

**if**  $tail == A.length$  **then**  $tail = 1$   
**else**  $tail = tail + 1$

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int  $n$ )

boolean Empty()

Enqueue(key  $k$ )

*stell neues Element  
an den Schwanz der  
Schlange an*

key Dequeue()

*entnimmt Element am  
Kopf der Schlange*

### Implementierung

```
A = new key[1..n]
tail = head = 1
```

```
key[] A
int tail
int head
```

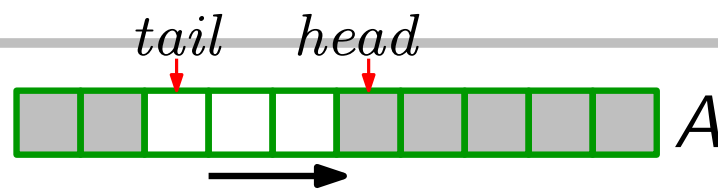
```
if head == tail then return true
else return false
```

```
A[tail] = k
if tail == A.length then tail = 1
else tail = tail + 1
```

```
k = A[head]
if head == A.length then head = 1
else head = head + 1
return k
```



## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int *n*)

boolean Empty()

Enqueue(key *k*)  
*stell neues Element  
 an den Schwanz der  
 Schlange an*

key Dequeue()  
*entnimmt Element am  
 Kopf der Schlange*

### Implementierung

```
A = new key[1..n]
tail = head = 1
```

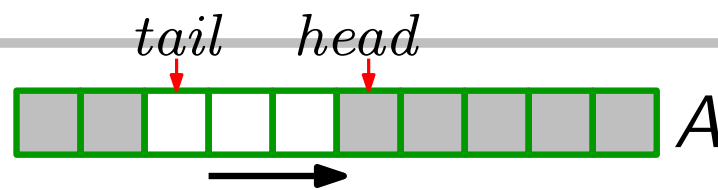
```
key[] A
int tail
int head
```

```
if head == tail then return true
else return false
```

```
A[tail] = k
if tail == A.length then tail = 1
else tail = tail + 1
```

```
k = A[head]
if head == A.length then head = 1
else head = head + 1
return k
```

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



### Abs. Datentyp

Queue(int  $n$ )

boolean Empty()

Enqueue(key  $k$ )

*stell neues Element  
an den Schwanz der  
Schlange an*

key Dequeue()

*entnimmt Element am  
Kopf der Schlange*

### Implementierung

```
A = new key[1..n]
tail = head = 1
```

```
key[] A
int tail
int head
```

```
if head == tail then return true
else return false
```

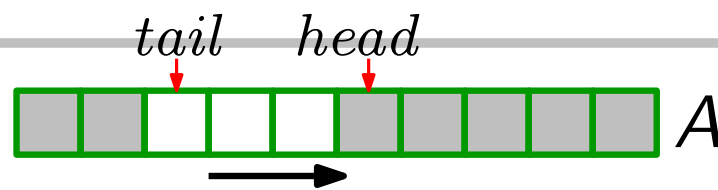
```
A[tail] = k
```

```
if tail == A.length then tail = 1
else tail = tail + 1
```

```
k = A[head]
```

```
if head == A.length then head = 1
else head = head + 1
return k
```

## II. Schlange

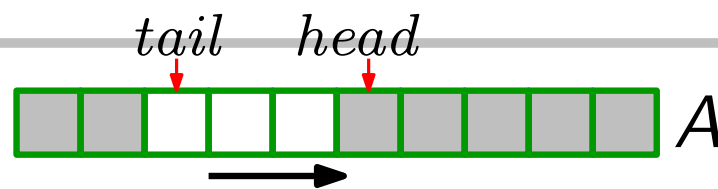


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung	
Queue(int <i>n</i> )	<pre>A = new key[1..n] tail = head = 1</pre>	<pre>key[] A int tail int head</pre>
<b>Aufgabe:</b>		
boolean Empty()	<pre>if head == tail then return true else return false</pre>	
Enqueue(key <i>k</i> ) <i>stell neues Element an den Schwanz der Schlange an</i>	<pre>A[tail] = k if tail == A.length then tail = 1 else tail = tail + 1</pre>	
key Dequeue() <i>entnimmt Element am Kopf der Schlange</i>	<pre>k = A[head] if head == A.length then head = 1 else head = head + 1 return k</pre>	

## II. Schlange

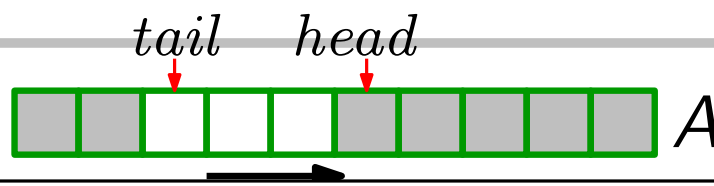


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung	
Queue(int <i>n</i> )	<pre>A = new key[1..n] tail = head = 1</pre>	<pre>key[] A int tail</pre>
<b>Aufgabe:</b> Fangen Sie underflow & overflow ab!		
boolean Empty()	<pre>if head == tail then return true else return false</pre>	
Enqueue(key <i>k</i> ) <i>stell neues Element an den Schwanz der Schlange an</i>	<pre>A[tail] = k if tail == A.length then tail = 1 else tail = tail + 1</pre>	
key Dequeue() <i>entnimmt Element am Kopf der Schlange</i>	<pre>k = A[head] if head == A.length then head = 1 else head = head + 1 return k</pre>	

# II. Schlange



## Die sinnliche Spur der Erinnerung

Wer lernen will, muss vor allem reden und be-greifen

Der Mensch behält von dem ...

... was er liest



10 Prozent

... was er hört



20

... was er sieht



30

... was er sieht und hört



50

... worüber wir selbst sprechen



70

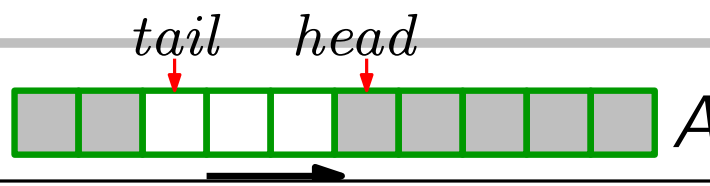
... was er selbst ausführt



90

**HANDLUNGSORIENTIERTES LERNEN** ist am effektivsten. Diese Einsicht ist seit fast 20 Jahren bekannt. Damals erschien diese Studie der American Audiovisual Society. Ergebnis: Von dem, was wir mit eigenen Händen tun, behalten wir 90 Prozent im Gedächtnis, von dem, worüber wir selbst sprechen, immerhin noch 70 Prozent. Von der reinen Lektüre eines Buches erinnern wir später nur noch 10 Prozent

# II. Schlange



## Die sinnliche Spur der Erinnerung

Wer lernen will, muss vor allem reden und be-greifen

Der Mensch behält von dem ...

... was er liest



10 Prozent

... was er hört



20

... was er sieht



30

... was er sieht und hört



50

... worüber wir selbst sprechen



70

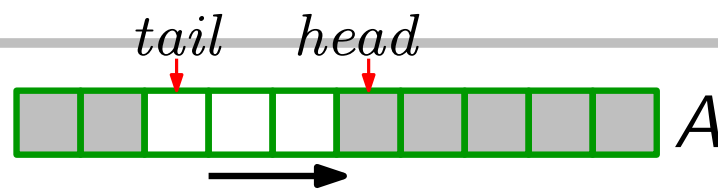
... was er selbst ausführt



90

**HANDLUNGSORIENTIERTES LERNEN** ist am effektivsten. Diese Einsicht ist seit fast 20 Jahren bekannt. Damals erschien diese Studie der American Audiovisual Society. Ergebnis: Von dem, was wir mit eigenen Händen tun, behalten wir 90 Prozent im Gedächtnis, von dem, worüber wir selbst sprechen, immerhin noch 70 Prozent. Von der reinen Lektüre eines Buches erinnern wir später nur noch 10 Prozent

## II. Schlange

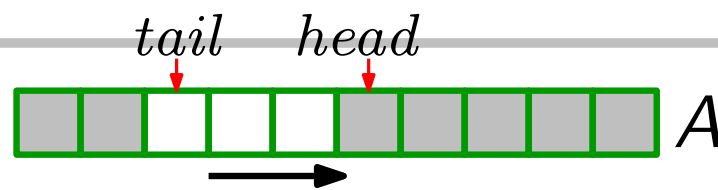


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung	
Queue(int <i>n</i> )	<pre>A = new key[1..n] tail = head = 1</pre>	<pre>key[] A int tail</pre>
<b>Aufgabe:</b> Fangen Sie underflow & overflow ab!		
boolean Empty()	<pre>if head == tail then return true else return false</pre>	
Enqueue(key <i>k</i> ) <i>stell neues Element an den Schwanz der Schlange an</i>	<pre>A[tail] = k if tail == A.length then tail = 1 else tail = tail + 1</pre>	
key Dequeue() <i>entnimmt Element am Kopf der Schlange</i>	<pre>k = A[head] if head == A.length then head = 1 else head = head + 1 return k</pre>	

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

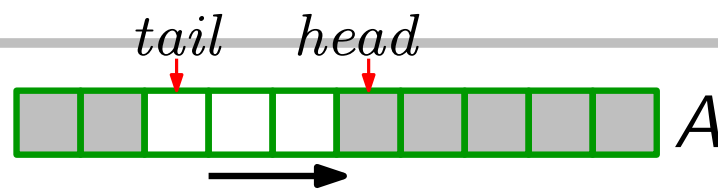


Abs. Datentyp	Implementierung	
Queue(int <i>n</i> )	$A = \text{new key}[1..n]$ $tail = head = 1$	$key[]$ A int <i>tail</i> int <i>head</i>
<b>Aufgabe:</b> Fangen Sie underflow & overflow ab!		
boolean Empty()	<b>if</b> $head == tail$ <b>then return true</b> <b>else return false</b>	
Enqueue(key <i>k</i> ) <i>stell neues Element an den Schwanz der Schlange an</i>	$A[tail] = k$ <b>if</b> $tail == A.length$ <b>then</b> $tail = 1$ <b>else</b> $tail = tail + 1$	
key Dequeue() <i>entnimmt Element am Kopf der Schlange</i>	$k = A[head]$ <b>if</b> $head == A.length$ <b>then</b> $head = 1$ <b>else</b> $head = head + 1$ <b>return k</b>	

**Laufzeiten?**



## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



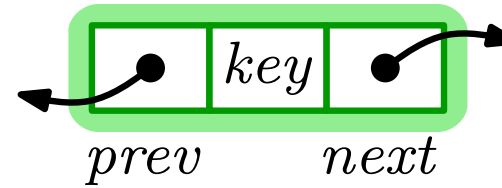
Abs. Datentyp	Implementierung	
Queue(int <i>n</i> )	$A = \text{new}^* \text{key}[1..n]$ $\text{tail} = \text{head} = 1$	$\text{key}[]$ <i>A</i> $\text{int tail}$ $\text{int head}$
<b>Aufgabe:</b> Fangen Sie underflow & overflow ab!		
boolean Empty()	<b>if</b> $\text{head} == \text{tail}$ <b>then return true</b> <b>else return false</b>	
Enqueue(key <i>k</i> ) <i>stell neues Element an den Schwanz der Schlange an</i>	$A[\text{tail}] = k$ <b>if</b> $\text{tail} == A.\text{length}$ <b>then</b> $\text{tail} = 1$ <b>else</b> $\text{tail} = \text{tail} + 1$	
key Dequeue() <i>entnimmt Element am Kopf der Schlange</i>	$k = A[\text{head}]$ <b>if</b> $\text{head} == A.\text{length}$ <b>then</b> $\text{head} = 1$ <b>else</b> $\text{head} = \text{head} + 1$ <b>return</b> $k$	

**Laufzeiten?**  
 Alle\*  $O(1)$ .

# III. Liste

<b>Abs. Datentyp</b>	<b>Implementierung</b>

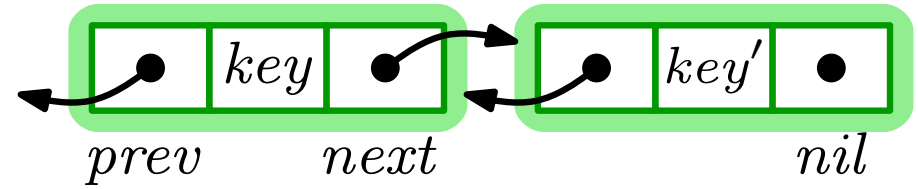
# III. Liste



**Abs. Datentyp**

**Implementierung**

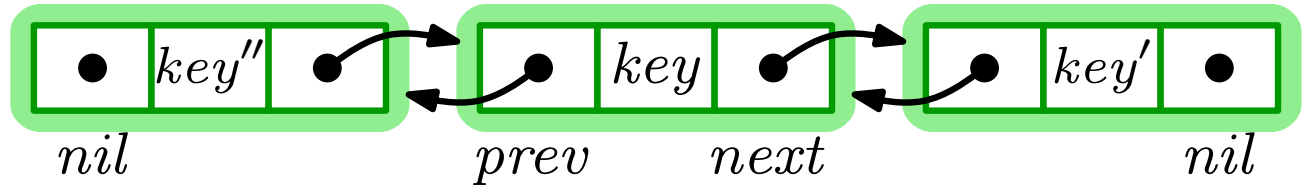
# III. Liste



**Abs. Datentyp**

**Implementierung**

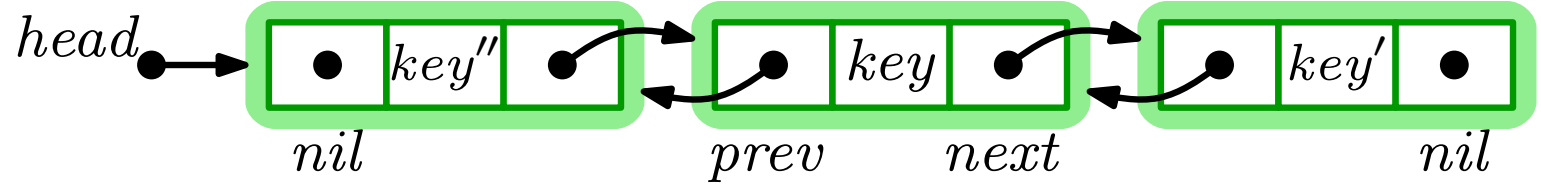
# III. Liste



**Abs. Datentyp**

**Implementierung**

# III. Liste

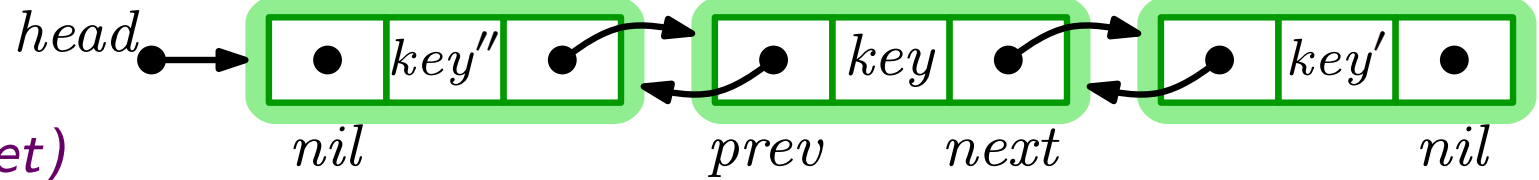


**Abs. Datentyp**

**Implementierung**

# III. Liste

(doppelt verkettet)

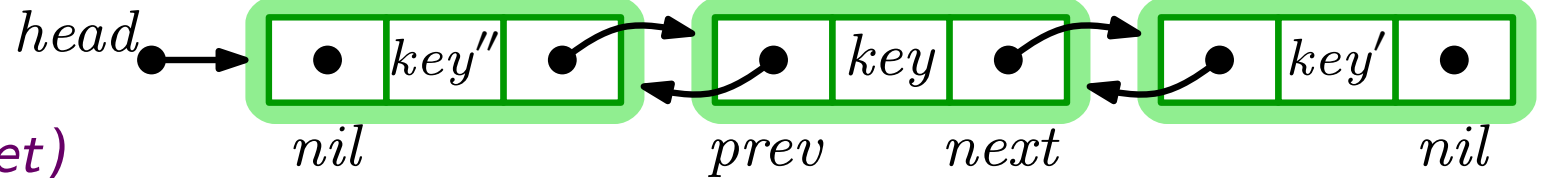


**Abs. Datentyp**

**Implementierung**

# III. Liste

(doppelt verkettet)



**Abs. Datentyp**

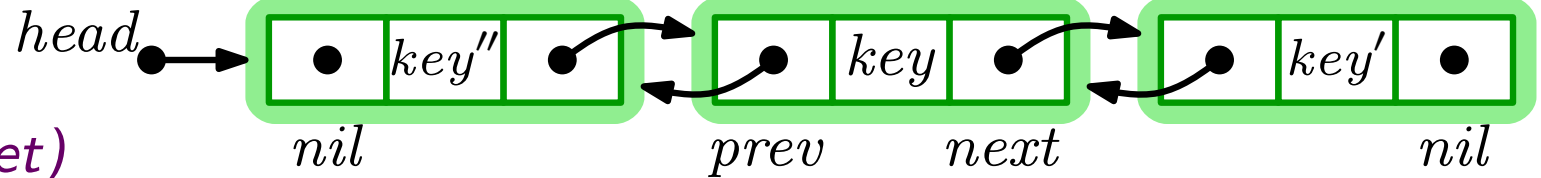
**Implementierung**

ptr *head*



# III. Liste

(doppelt verkettet)



**Abs. Datentyp**

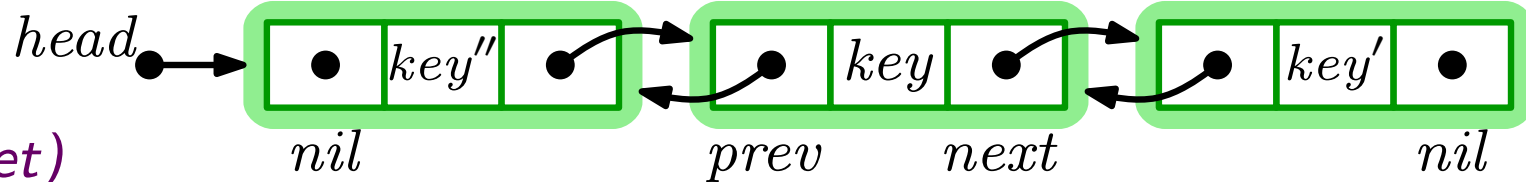
**Implementierung**

```
key key  
ptr prev  
ptr next
```

```
ptr head
```

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

## Implementierung

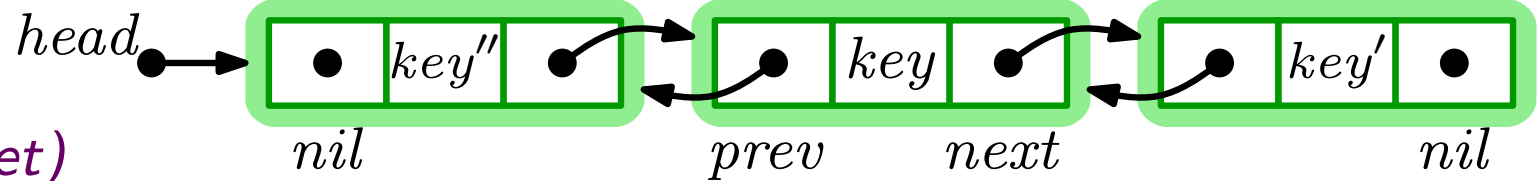
Item

key	key
ptr	prev
ptr	next

ptr head

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

## Implementierung

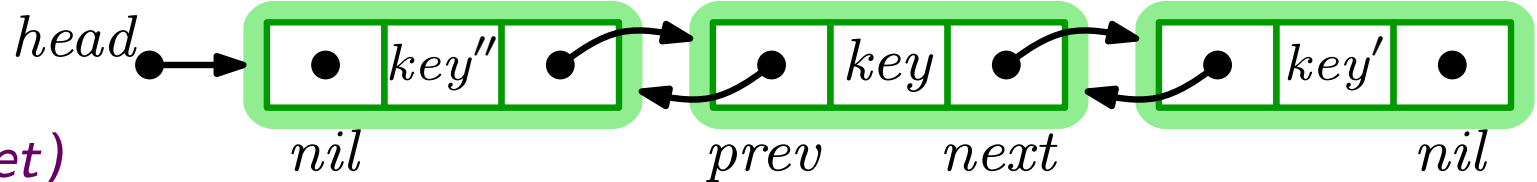
Item

key	key
ptr	prev
ptr	next

ptr head

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

## Implementierung

$head = nil$

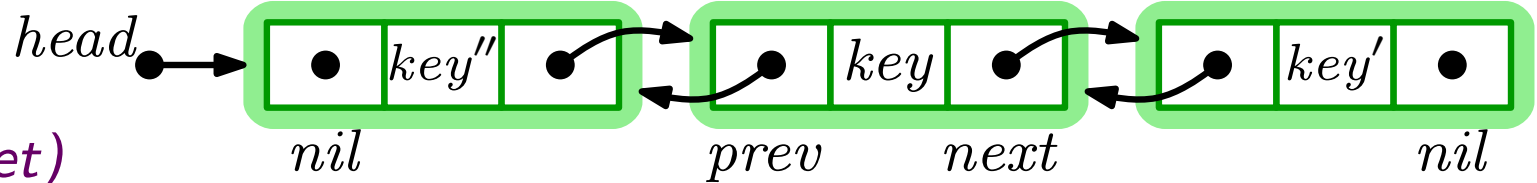
Item

key	key
ptr	prev
ptr	next

ptr head

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

## Implementierung

*head = nil*

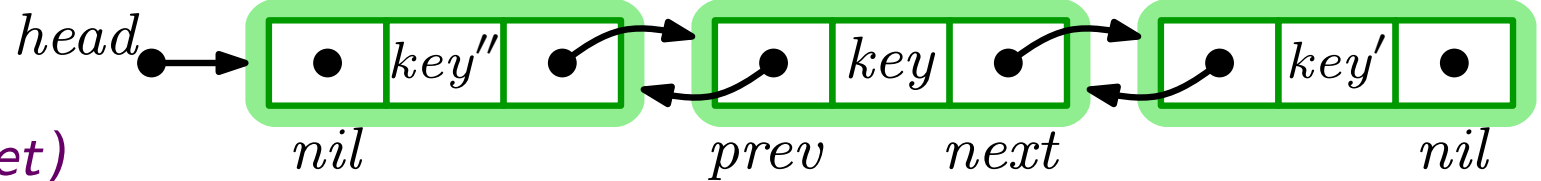
Item

key	key
ptr	prev
ptr	next

ptr *head*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

## Implementierung

*head = nil*

*x = head*

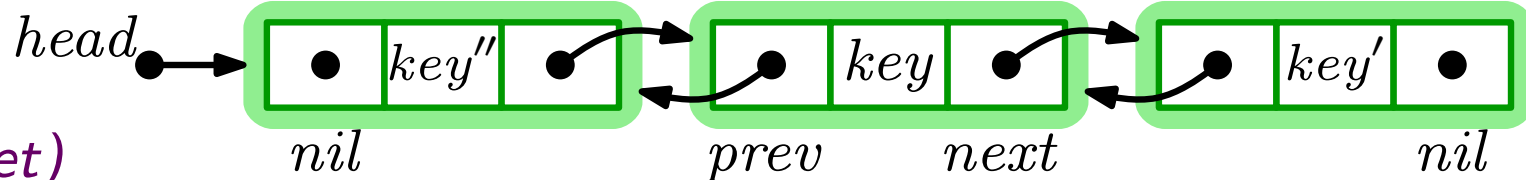
Item

key	<i>key</i>
ptr	<i>prev</i>
ptr	<i>next</i>

ptr *head*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

## Implementierung

*head* = *nil*

Item

key	<i>key</i>
ptr	<i>prev</i>
ptr	<i>next</i>

ptr *head*

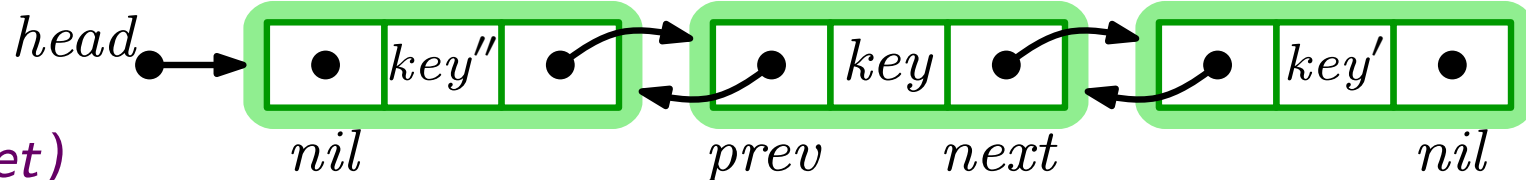
*x* = *head*

**while** *x* ≠ *nil* **and** *x.key* ≠ *k* **do**

└ *x* = *x.next*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

## Implementierung

*head* = *nil*

Item

key	<i>key</i>
ptr	<i>prev</i>
ptr	<i>next</i>

ptr *head*

*x* = *head*

**while** *x* ≠ *nil* **and** *x.key* ≠ *k* **do**

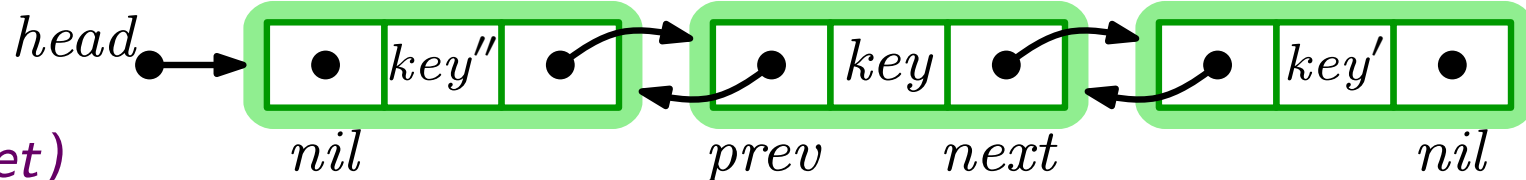
└ *x* = *x.next*

**return** *x*



# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

ptr Insert(key  $k$ )

## Implementierung

$head = nil$

Item

key	key
ptr	prev
ptr	next

ptr head

$x = head$

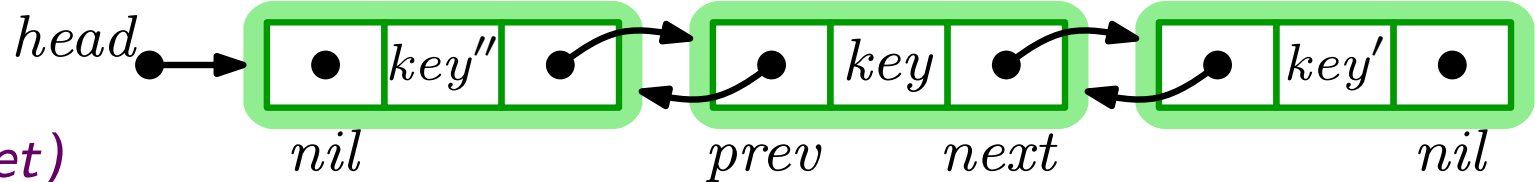
**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**

$x = x.next$

**return**  $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

ptr Insert(key  $k$ )

## Implementierung

$head = nil$

Item

key	key
ptr	prev
ptr	next

ptr head

$x = head$

**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**  
     $x = x.next$

**return**  $x$

$x = \mathbf{new}$  Item()

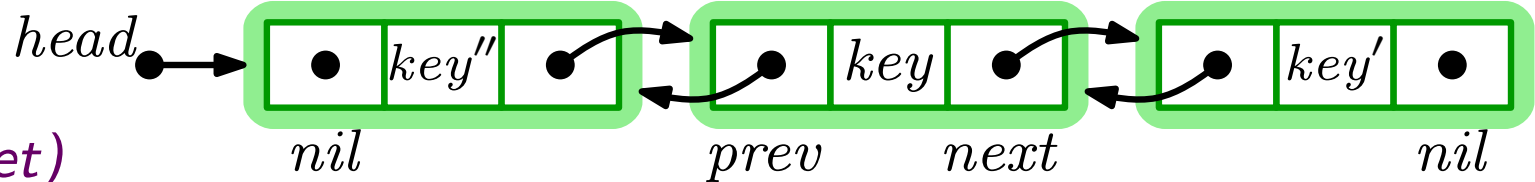
$x.key = k$ ;  $x.prev = nil$ ;  $x.next = head$

**if**  $head \neq nil$  **then**  $head.prev = x$

$head = x$ ; **return**  $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

ptr Insert(key  $k$ )

## Implementierung

$head = nil$

Item

key	key
ptr	prev
ptr	next

ptr head

$x = head$

**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**  
     $x = x.next$

**return**  $x$

$x = \mathbf{new}$  Item()

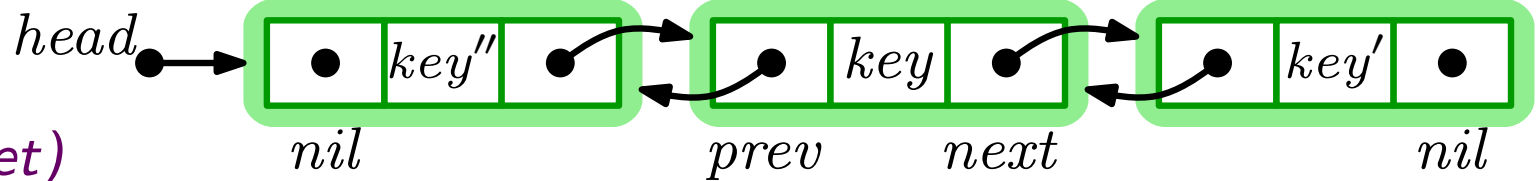
$x.key = k; x.prev = nil; x.next = head$

**if**  $head \neq nil$  **then**  $head.prev = x$

$head = x; \mathbf{return}$   $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

ptr Insert(key  $k$ )

## Implementierung

$head = nil$

Item(key  $k$ , ptr  $p$ )

Item

key	key
ptr	prev
ptr	next

ptr  $head$

$x = head$

**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**

$x = x.next$

**return**  $x$

$x = \mathbf{new}$  Item()

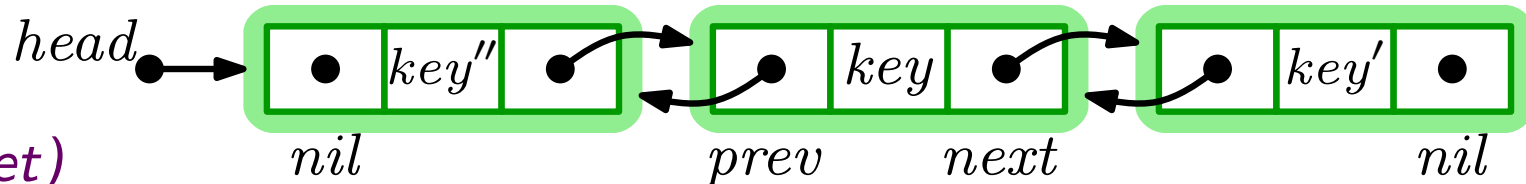
$x.key = k; x.prev = nil; x.next = head$

**if**  $head \neq nil$  **then**  $head.prev = x$

$head = x; \mathbf{return}$   $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

ptr Insert(key  $k$ )

## Implementierung

$head = nil$

Item(key  $k$ , ptr  $p$ )

$key = k$

$next = p$

$prev = nil$

Item

key key

ptr prev

ptr next

ptr head

$x = head$

**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**

$x = x.next$

**return**  $x$

$x = \mathbf{new}$  Item()

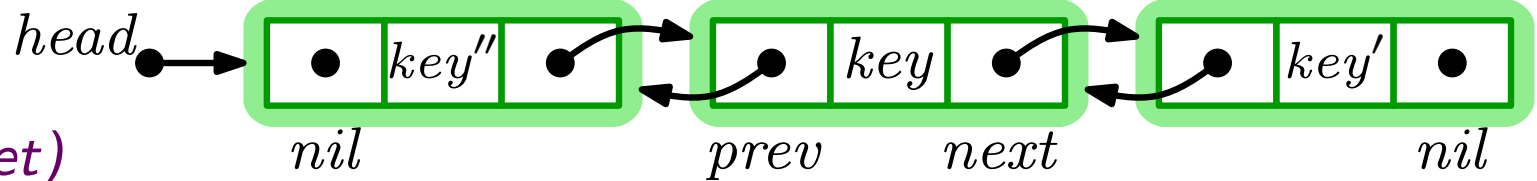
$x.key = k; x.prev = nil; x.next = head$

**if**  $head \neq nil$  **then**  $head.prev = x$

$head = x; \mathbf{return}$   $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

ptr Insert(key  $k$ )

## Implementierung

$head = nil$

Item(key  $k$ , ptr  $p$ )

$key = k$

$next = p$

$prev = nil$

Item

key key

ptr prev

ptr next

ptr head

$x = head$

**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**

$x = x.next$

**return**  $x$

$x = \mathbf{new}$  Item( $\rightarrow k, head$ )

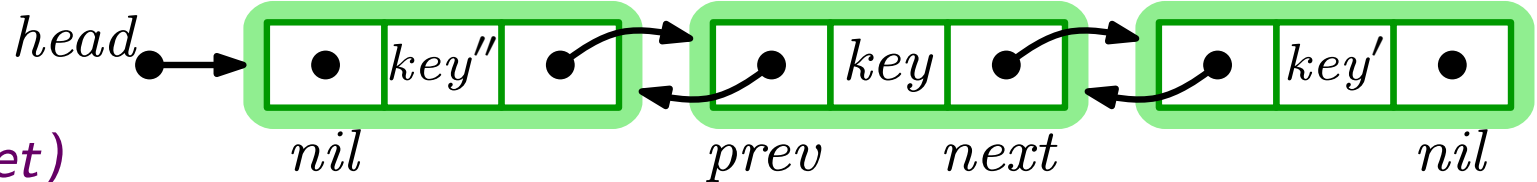
$x.key = k; x.prev = nil; x.next = head$

**if**  $head \neq nil$  **then**  $head.prev = x$

$head = x; \mathbf{return}$   $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

ptr Insert(key  $k$ )

## Implementierung

$head = nil$

Item(key  $k$ , ptr  $p$ )

$key = k$

$next = p$

$prev = nil$

Item

key key

ptr prev

ptr next

ptr head

$x = head$

**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**

$x = x.next$

**return**  $x$

$x = \mathbf{new}$  Item( $\rightarrow k, head$ )

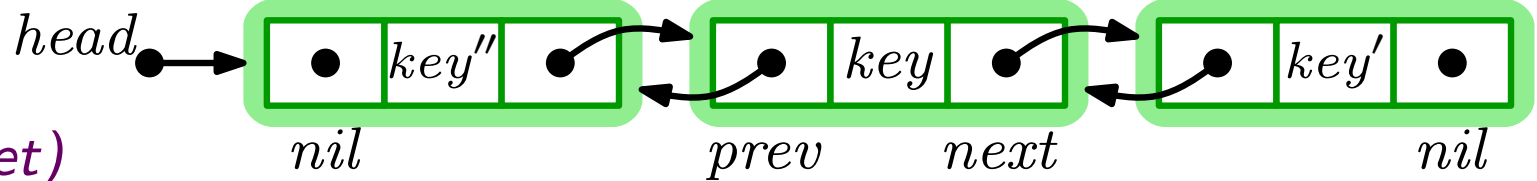
~~$x.key = k, x.prev = nil; x.next = head$~~

**if**  $head \neq nil$  **then**  $head.prev = x$

$head = x$ ; **return**  $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

ptr Insert(key *k*)

### Aufgabe:

Implementieren Sie  
Delete(ptr *x*)

## Implementierung

*head* = *nil*

Item(key *k*, ptr *p*)

*key* = *k*

*next* = *p*

*prev* = *nil*

Item

key *key*

ptr *prev*

ptr *next*

ptr *head*

*x* = *head*

**while** *x* ≠ *nil* **and** *x.key* ≠ *k* **do**

└ *x* = *x.next*

**return** *x*

*x* = **new** Item(*k*, *head*)

~~*x.key* = *k*, *x.prev* = *nil*; *x.next* = *head*~~

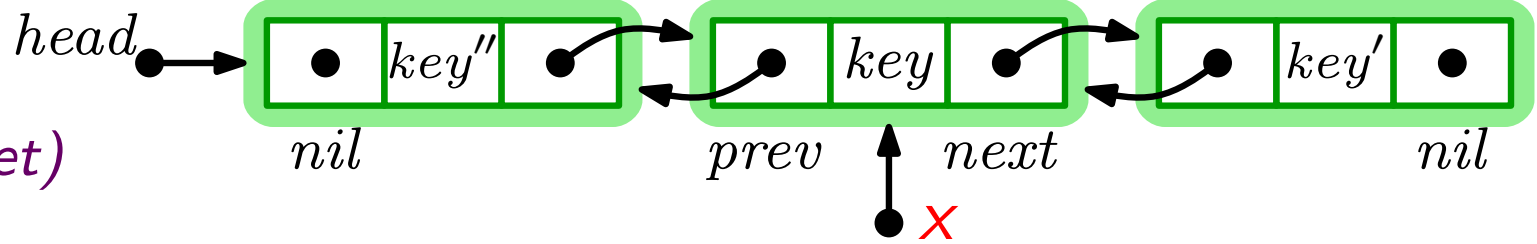
**if** *head* ≠ *nil* **then** *head.prev* = *x*

*head* = *x*; **return** *x*



# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

ptr Insert(key *k*)

### Aufgabe:

Implementieren Sie  
Delete(ptr *x*)

## Implementierung

*head* = *nil*

Item(key *k*, ptr *p*)

*key* = *k*

*next* = *p*

*prev* = *nil*

Item

key *key*

ptr *prev*

ptr *next*

ptr *head*

*x* = *head*

**while** *x* ≠ *nil* **and** *x.key* ≠ *k* **do**

└ *x* = *x.next*

**return** *x*

*x* = **new** Item(*k*, *head*)

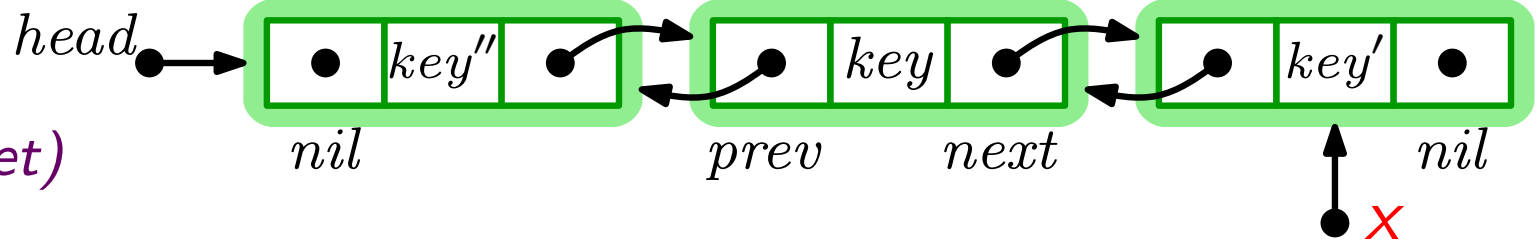
~~*x.key* = *k*, *x.prev* = *nil*; *x.next* = *head*~~

**if** *head* ≠ *nil* **then** *head.prev* = *x*

*head* = *x*; **return** *x*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

ptr Insert(key *k*)

### Aufgabe:

Implementieren Sie Delete(ptr *x*)

## Implementierung

*head* = *nil*

Item(key *k*, ptr *p*)

*key* = *k*

*next* = *p*

*prev* = *nil*

Item

key *key*

ptr *prev*

ptr *next*

ptr *head*

*x* = *head*

**while** *x* ≠ *nil* **and** *x.key* ≠ *k* **do**

*x* = *x.next*

**return** *x*

*x* = **new** Item(*k*, *head*)

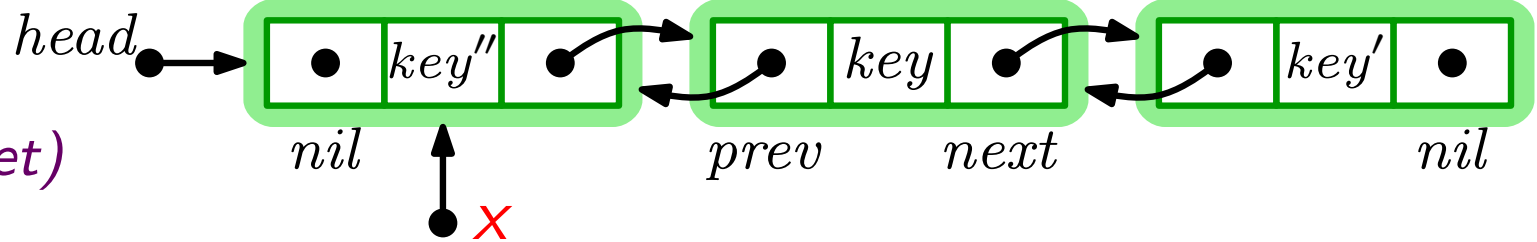
~~*x.key* = *k*, *x.prev* = *nil*; *x.next* = *head*~~

**if** *head* ≠ *nil* **then** *head.prev* = *x*

*head* = *x*; **return** *x*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

ptr Insert(key *k*)

### Aufgabe:

Implementieren Sie  
Delete(ptr *x*)

## Implementierung

*head* = nil

Item(key *k*, ptr *p*)

key = *k*

next = *p*

prev = nil

Item

key key

ptr prev

ptr next

ptr *head*

*x* = *head*

**while** *x* ≠ nil **and** *x*.key ≠ *k* **do**

└ *x* = *x*.next

**return** *x*

*x* = **new** Item(*k*, *head*)

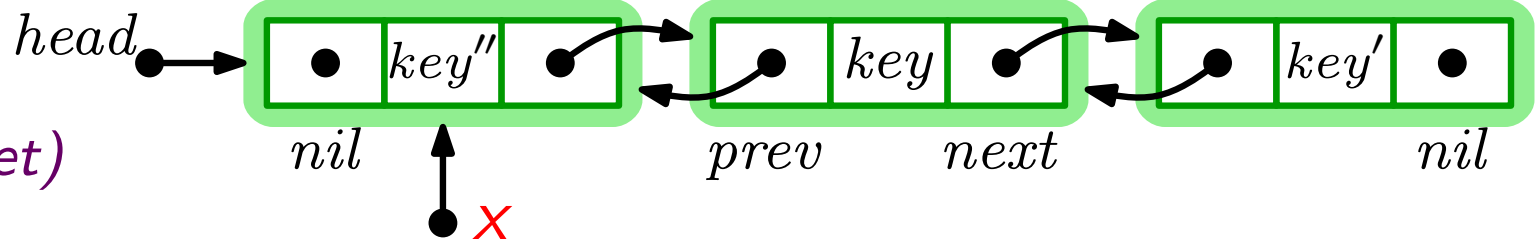
~~*x*.key = *k*, *x*.prev = nil; *x*.next = *head*~~

**if** *head* ≠ nil **then** *head*.prev = *x*

*head* = *x*; **return** *x*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key *k*)

**Hausaufgabe:**

Benutzen Sie  
Stopper!

ptr Insert(key *k*)

**Aufgabe:**

Implementieren Sie  
Delete(ptr *x*)

## Implementierung

*head* = nil

Item(key *k*, ptr *p*)

key = *k*

next = *p*

prev = nil

Item

key key

ptr prev

ptr next

ptr *head*

*x* = *head*

**while** *x* ≠ nil **and** *x*.key ≠ *k* **do**

└ *x* = *x*.next

**return** *x*

*x* = **new** Item(*k*, *head*)

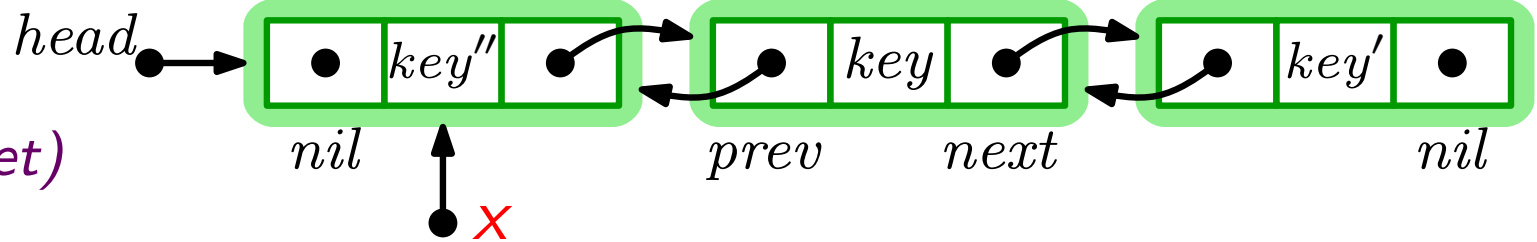
~~*x*.key = *k*, *x*.prev = nil; *x*.next = *head*~~

**if** *head* ≠ nil **then** *head*.prev = *x*

*head* = *x*; **return** *x*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

**Laufzeiten?**

ptr Search(key *k*)

ptr Insert(key *k*)

## Implementierung

*head* = *nil*

Item(key *k*, ptr *p*)

*key* = *k*

*next* = *p*

*prev* = *nil*

Item

key *key*

ptr *prev*

ptr *next*

ptr *head*

*x* = *head*

**while** *x* ≠ *nil* **and** *x.key* ≠ *k* **do**

*x* = *x.next*

**return** *x*

*x* = **new** Item(*k*, *head*)

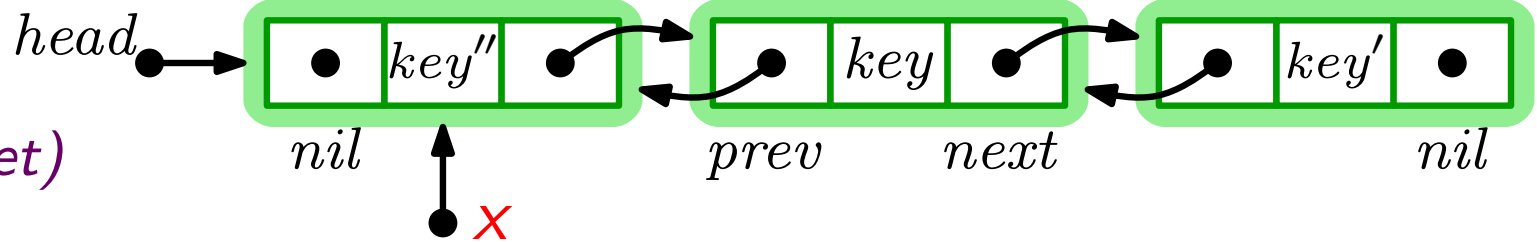
~~*x.key* = *k*, *x.prev* = *nil*; *x.next* = *head*~~

**if** *head* ≠ *nil* **then** *head.prev* = *x*

*head* = *x*; **return** *x*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

**Laufzeiten?**

ptr Search(key k)

ptr Insert(key k)

Delete(ptr x)

## Implementierung

head = nil

Item(key k, ptr p)

key = k

next = p

prev = nil

Item

key key

ptr prev

ptr next

ptr head

x = head

**while** x ≠ nil **and** x.key ≠ k **do**

└ x = x.next

**return** x

x = **new** Item(~~key~~ k, head)

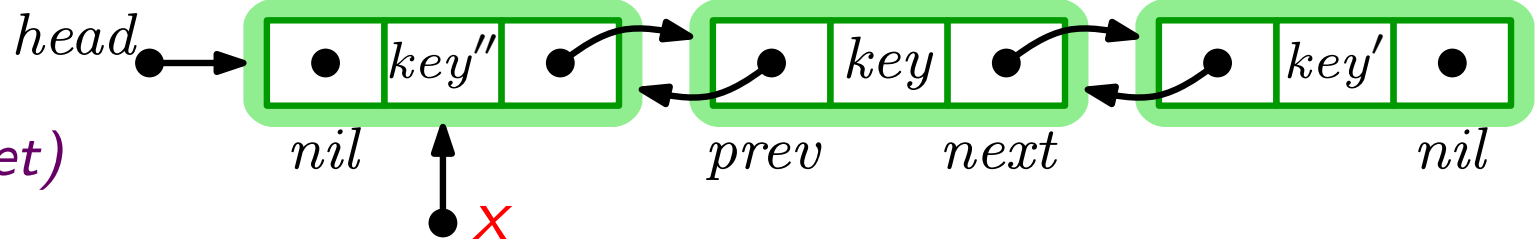
~~x.key = k, x.prev = nil; x.next = head~~

**if** head ≠ nil **then** head.prev = x

head = x; **return** x

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List() O(1)

## Laufzeiten?

ptr Search(key k)  

ptr Insert(key k)  

Delete(ptr x)  

## Implementierung

*head = nil*

Item(key k, ptr p)

*key = k*

*next = p*

*prev = nil*

Item

key key

ptr prev

ptr next

ptr head

*x = head*

**while** *x ≠ nil* **and** *x.key ≠ k* **do**

*x = x.next*

**return** *x*

*x = new* Item(↗ *k*, *head*)

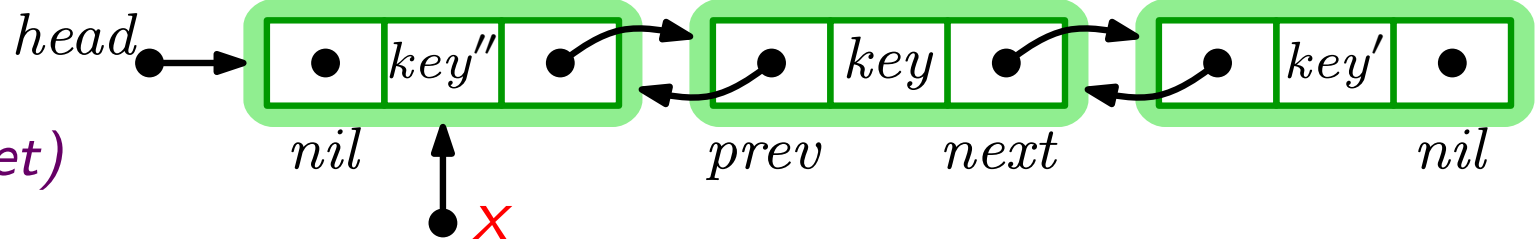
~~*x.key = k, x.prev = nil; x.next = head*~~

**if** *head ≠ nil* **then** *head.prev = x*

*head = x; return* *x*

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()  $O(1)$

**Laufzeiten?**

ptr Search(key  $k$ )  $O(n)$

ptr Insert(key  $k$ )

Delete(ptr  $x$ )

## Implementierung

$head = nil$

```

Item(key  $k$ , ptr  $p$ )
  key =  $k$ 
  next =  $p$ 
  prev =  $nil$ 

```

```

Item
  key key
  ptr prev
  ptr next

```

ptr  $head$

$x = head$

```

while  $x \neq nil$  and  $x.key \neq k$  do
   $x = x.next$ 

```

return  $x$

$x = \mathbf{new}$  Item( $\rightarrow k, head$ )

~~$x.key = k, x.prev = nil; x.next = head$~~

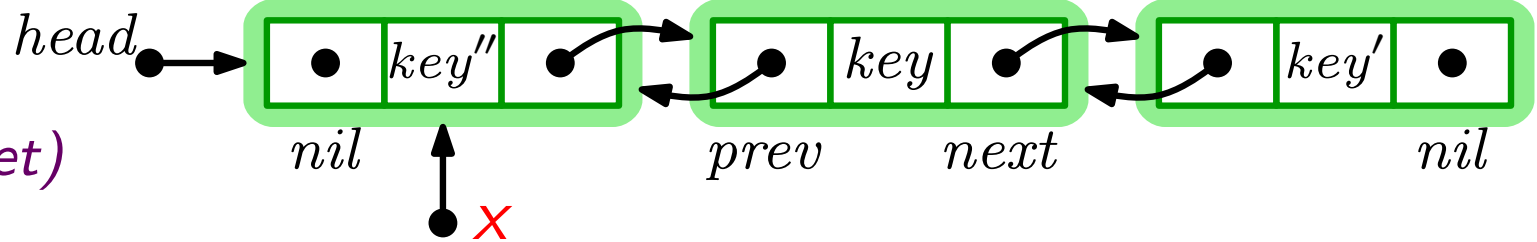
if  $head \neq nil$  then  $head.prev = x$

$head = x$ ; return  $x$



# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()  $O(1)$

## Laufzeiten?

ptr Search(key  $k$ )  $O(n)$

ptr Insert(key  $k$ )  $O(1)$

Delete(ptr  $x$ )

## Implementierung

$head = nil$

```
Item(key  $k$ , ptr  $p$ )
  key =  $k$ 
  next =  $p$ 
  prev =  $nil$ 
```

Item

```
key key
ptr prev
ptr next
```

ptr  $head$

$x = head$

**while**  $x \neq nil$  **and**  $x.key \neq k$  **do**  
 $\quad \perp x = x.next$

**return**  $x$

$x = \mathbf{new}$  Item( $\rightarrow k, head$ )

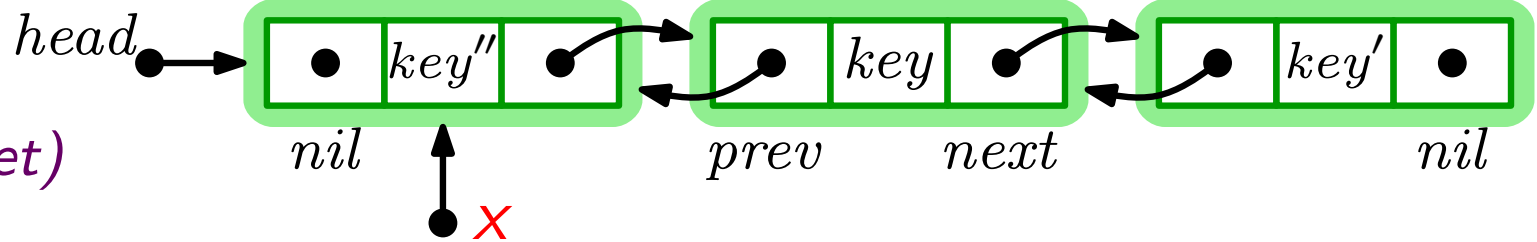
~~$x.key = k, x.prev = nil; x.next = head$~~

**if**  $head \neq nil$  **then**  $head.prev = x$

$head = x$ ; **return**  $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()  $O(1)$

## Laufzeiten?

ptr Search(key  $k$ )  $O(n)$

ptr Insert(key  $k$ )  $O(1)$

Delete(ptr  $x$ )  $O(1)$

## Implementierung

$head = nil$

```

Item(key  $k$ , ptr  $p$ )
  key =  $k$ 
  next =  $p$ 
  prev =  $nil$ 

```

```

Item
  key key
  ptr prev
  ptr next

```

ptr  $head$

$x = head$

```

while  $x \neq nil$  and  $x.key \neq k$  do
   $x = x.next$ 

```

return  $x$

$x = \mathbf{new}$  Item( $\rightarrow k, head$ )

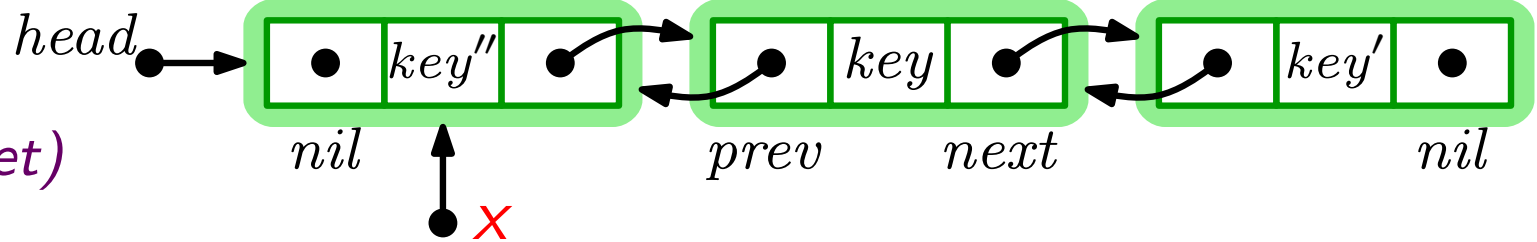
~~$x.key = k, x.prev = nil; x.next = head$~~

if  $head \neq nil$  then  $head.prev = x$

$head = x$ ; return  $x$

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()  $O(1)$

## Laufzeiten?

ptr Search(key  $k$ )  $O(n)$

ptr Insert(key  $k$ )  $O(1)$

$O(1)$   
Delete(ptr  $x$ )

## Implementierung

$head = nil$	Item(key $k$ , ptr $p$ ) key = $k$ next = $p$ prev = $nil$	Item key key ptr prev ptr next
		ptr head

```

x = head
while x ≠ nil and x.key ≠ k do
  x = x.next
return x

```

```

x = new Item( $k$ , head)
x.key = k, x.prev = nil; x.next = head
if head ≠ nil then head.prev = x
head = x; return x

```

# Von Pseudocode zu Javacode: (1) Item

Item(key $k$ , ptr $p$ ) $key = k$ $next = p$ $prev = nil$	Item key $key$ ptr $prev$ ptr $next$
---	---

# Von Pseudocode zu Javacode: (1) Item

Item(key $k$ , ptr $p$ ) $key = k$ $next = p$ $prev = nil$	Item key $key$ ptr $prev$ ptr $next$
---	---

# Von Pseudocode zu Javacode: (1) Item

```
public class Item {
```

```
    private Object key;  
    private Item prev;  
    private Item next;
```

```
Item(key k, ptr p)
```

```
    key = k
```

```
    next = p
```

```
    prev = nil
```

```
Item
```

```
    key key
```

```
    ptr prev
```

```
    ptr next
```

```
}
```

# Von Pseudocode zu Javacode: (1) Item

```
public class Item {
```

```
    private Object key;  
    private Item prev;  
    private Item next;
```

```
    public Item(Object k, Item p) {  
        key = k;  
        next = p;  
        prev = null;  
    }
```

```
Item(key k, ptr p)
```

```
    key = k
```

```
    next = p
```

```
    prev = nil
```

```
Item
```

```
key key
```

```
ptr prev
```

```
ptr next
```

```
}
```

# Von Pseudocode zu Javacode: (1) Item

```
public class Item {
```

```
  private Object key;
  private Item prev;
  private Item next;
```

```
  public Item(Object k, Item p) {
    key = k;
    next = p;
    prev = null;
  }
```

```
  public void setPrev(Item p) { prev = p; }
  public void setNext(Item p) { next = p; }

  public Item getPrev() { return prev; }
  public Item getNext() { return next; }
  public Object getKey() { return key; }
```

```
}
```

```
Item(key k, ptr p)
  key = k
  next = p
  prev = nil
```

```
Item
```

```
key key
ptr prev
ptr next
```



# Von Pseudocode zu Javacode: (1) Item

```
public class Item {
```

```
private Object key;
private Item prev;
private Item next;
```

```
Item(key k, ptr p)
```

```
key = k
next = p
prev = nil
```

```
Item
```

```
key key
ptr prev
ptr next
```

```
public Item(Object k, Item p) {
    key = k;
    next = p;
    prev = null;
}
```

```
public void setPrev(Item p) { prev = p; }
public void setNext(Item p) { next = p; }
public Item getPrev() { return prev; }
public Item getNext() { return next; }
public Object getKey() { return key; }
```

setter-  
und  
getter-  
Methoden

```
}
```

# Von Pseudocode zu Javacode: (2) List

```
ptr head
```

```
List()
```

```
head = nil
```

```
ptr Insert(key k)
```

```
x = new Item(k, head)
```

```
if head  $\neq$  nil then
```

```
└ head.prev = x
```

```
head = x
```


```
return x
```

# Von Pseudocode zu Javacode: (2) List

```
public class List {
```

```
    private Item head;
```

```
ptr head
```



```
List()
```

```
    head = nil
```

```
ptr Insert(key k)
```

```
    x = new Item(k, head)
```

```
    if head  $\neq$  nil then
```

```
        | head.prev = x
```

```
    head = x
```

```
    return x
```

# Von Pseudocode zu Javacode: (2) List

```
public class List {
    private Item head;
    public List() {
        head = null;
    }
}
```

*ptr head*

List()  
*head = nil*

*ptr Insert(key k)*

```
x = new Item(k, head)
if head  $\neq$  nil then
    | head.prev = x
head = x
return x
```

# Von Pseudocode zu Javacode: (2) List

```
public class List {
```

```
private Item head;
```

```
ptr head
```

```
public List() {
    head = null;
}
```

```
List()
    head = nil
```

```
public Item insert(Object k) {
    Item x = new Item(k, head);
    if (head != null) {
        head.setPrev(x);
    }
    head = x;
    return x;
}
```

```
ptr Insert(key k)
```

```
x = new Item(k, head)
if head  $\neq$  nil then
    | head.prev = x
head = x
return x
```

# Von Pseudocode zu Javacode: (2) List

```
public class List {
```

```
private Item head;
```

```
ptr head
```

```
public List() {
    head = null;
}
```

```
List()
    head = nil
```

```
public Item insert(Object k) {
    Item x = new Item(k, head);
    if (head != null) {
        head.setPrev(x);
    }
    head = x;
    return x;
}
```

```
ptr Insert(key k)
```

```
x = new Item(k, head)
if head ≠ nil then
    | head.prev = x
head = x
return x
```

```
public Item getHead() { return head; }
```

# Von Pseudocode zu Javacode: (2) List

```
ptr Search(key k)
```

```
  x = head
```

```
  while x ≠ nil and x.key ≠ k do
```

```
    ⊥ x = x.next
```

```
  return x
```

# Von Pseudocode zu Javacode: (2) List

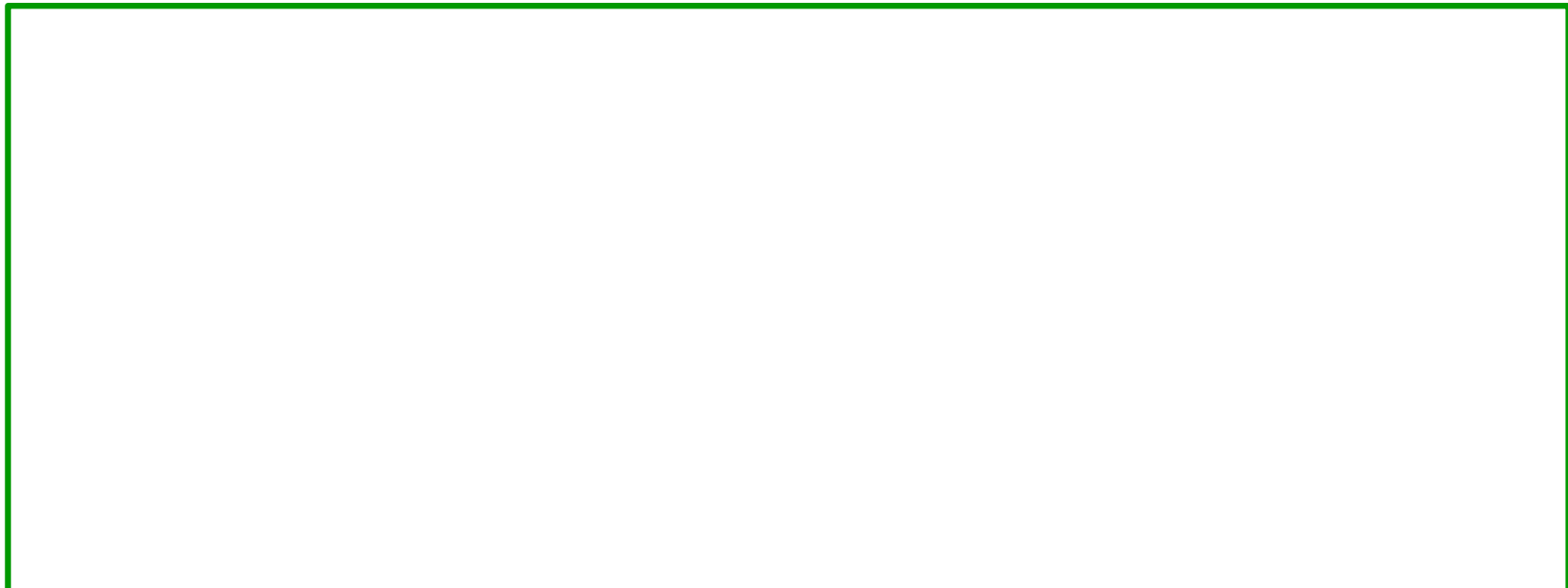
```
ptr Search(key k)
```

```
  x = head
```

```
  while x ≠ nil and x.key ≠ k do
```

```
    ⊥ x = x.next
```

```
  return x
```





# Von Pseudocode zu Javacode: (2) List

```
ptr Search(key k)
```

```
  x = head
```

```
  while x ≠ nil and x.key ≠ k do
```

```
    ⊥ x = x.next
```

```
  return x
```



```
public Item search(Object k) {
```

```
  Item x = head;
```

```
  while (x != null && x.getKey() != k) {
```

```
    x = x.getNext();
```

```
  }
```

```
  return x;
```

```
}
```

# Von Pseudocode zu Javacode: (2) List

Delete(ptr x)

**if**  $x.prev \neq nil$  **then**  $x.prev.next = x.next$

**else**  $head = x.next$

**if**  $x.next \neq nil$  **then**  $x.next.prev = x.prev$

# Von Pseudocode zu Javacode: (2) List

Delete(ptr x)

**if**  $x.prev \neq nil$  **then**  $x.prev.next = x.next$

**else**  $head = x.next$

**if**  $x.next \neq nil$  **then**  $x.next.prev = x.prev$



# Von Pseudocode zu Javacode: (2) List

```
Delete(ptr x)
```

```
  if  $x.prev \neq nil$  then  $x.prev.next = x.next$   
  else  $head = x.next$   
  if  $x.next \neq nil$  then  $x.next.prev = x.prev$ 
```



```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

```
}
```

## Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {
```

```
    }  
}
```

# Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
  
    }  
}
```

## Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
    }  
}
```

## Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
  
        System.out.println("Die Liste enthaelt:");  
        for (Item it = myList.getHead(); it != null;  
            it = it.getNext()) {  
            System.out.println((Integer) it.getKey());  
        }  
  
    }  
}
```



## Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
  
        System.out.println("Die Liste enthaelt:");  
        for (Item it = myList.getHead(); it != null;  
            it = it.getNext()) {  
            System.out.println((Integer) it.getKey());  
        } Was wird hier ausgegeben?  
    }  
}
```

## Javacode: (3) Main

```
public class Listentest {
    public static void main(String[] args) {
        List myList = new List();
        myList.insert(new Integer(10));
        myList.insert(new Integer(16));
        System.out.println("Die Liste enthaelt:");
        for (Item it = myList.getHead(); it != null;
            it = it.getNext()) {
            System.out.println((Integer) it.getKey());
        }
        Item it = myList.search(new Integer(16));
        myList.delete(it);
    }
}
```

## Javacode: (3) Main

```
public class Listentest {
    public static void main(String[] args) {
        List myList = new List();
        myList.insert(new Integer(10));
        myList.insert(new Integer(16));
        System.out.println("Die Liste enthaelt:");
        for (Item it = myList.getHead(); it != null;
            it = it.getNext()) {
            System.out.println((Integer) it.getKey());
        }
        Item it = myList.search(new Integer(16));
        myList.delete(it);
    }
}
```

## Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
        System.out.println("Die Liste enthaelt:");  
        for (Item it = myList.getHead(); it != null;  
            it = it.getNext()) {  
            System.out.println((Integer) it.getKey());  
        }  
        Item it = myList.search(new Integer(16));  
        myList.delete(it);  
    }  
}
```

```
Die Liste enthaelt:  
16  
10  
Fehler!
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

List.java

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

List.java

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

Listentest.java

```
myList.insert(new Integer(16));  
...  
Item it = myList.search(new Integer(16));  
myList.delete(it);
```

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

Listentest.java

```
myList.insert(new Integer(16));  
...  
Item it = myList.search(new Integer(16));  
myList.delete(it);
```

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```



# Warum "Fehler!" ?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

Listentest.java

```
myList.insert(new Integer(16));  
...  
Item it = myList.search(new Integer(16));  
myList.delete(it);
```

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

Listentest.java

```
myList.insert(new Integer(16));  
...  
Item it = myList.search(new Integer(16));  
myList.delete(it);
```

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {
    Item x = head;
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

Listentest.java

```
myList.insert(new Integer(16));
... gleiche Zahlen, aber verschiedene Objekte!
Item it = myList.search(new Integer(16));
myList.delete(it);
```

```
public void delete(Item x) {
    if (x == null) System.out.println("Fehler!");
    Item prev = x.getPrev();
    Item next = x.getNext();
    if (prev != null) prev.setNext(next);
    else head = next;
    if (next != null) next.setPrev(prev);
}
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {
    Item x = head;    !k.equals(x.getKey())
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

Listentest.java

```
myList.insert(new Integer(16));
... gleiche Zahlen, aber verschiedene Objekte!
Item it = myList.search(new Integer(16));
myList.delete(it);
```

```
public void delete(Item x) {
    if (x == null) System.out.println("Fehler!");
    Item prev = x.getPrev();
    Item next = x.getNext();
    if (prev != null) prev.setNext(next);
    else head = next;
    if (next != null) next.setPrev(prev);
}
```

# Warum "Fehler!"?

Item.java

```
public Item search(Object k) {
    Item x = head;    !k.equals(x.getKey())
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

Listentest.java

```
myList.insert(new Integer(16));
... gleiche Zahlen, aber verschiedene Objekte!
Item it = myList.search(new Integer(16));
myList.delete(it);
```

```
public void delete(Item x) {
    if (x == null) System.out.println("Fehler!");
    Item prev = x.getPrev();
    Item next = x.getNext();
    if (prev != null) prev.setNext(next);
    else head = next;
    if (next != null) next.setPrev(prev);
}
```



# Warum "Fehler!" ?

Item.java

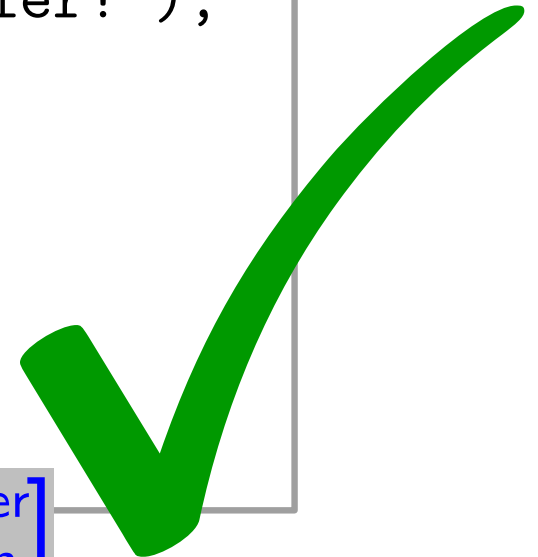
```
public Item search(Object k) {
    Item x = head;    !k.equals(x.getKey())
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

Listentest.java

```
myList.insert(new Integer(16));
... gleiche Zahlen, aber verschiedene Objekte!
Item it = myList.search(new Integer(16));
myList.delete(it);
```

```
public void delete(Item x) {
    if (x == null) System.out.println("Fehler!");
    Item prev = x.getPrev();
    Item next = x.getNext();
    if (prev != null) prev.setNext(next);
    else head = next;
    if (next != null) next.setPrev(prev);
}
```

**[Unschön: Klasse Item muss public sein, so dass Anwender  
und Bibliotheksklasse List darüber kommunizieren können.]**



# Übersicht Elementare Datenstrukturen

---

Operationen

Stapel

Schlange

Liste

---

Einfügen

Entfernen

weitere Oper.

(außer Konstruktor  
und Empty())

---

# Übersicht Elementare Datenstrukturen

---

Operationen

---

Stapel

Schlange

Liste

---

Einfügen

Push()

Entfernen

Pop()

weitere Oper.

(außer Konstruktor  
und Empty())

Top()

---



# Übersicht Elementare Datenstrukturen

---

Operationen

---

Stapel

Schlange

Liste

---

Einfügen

Push()

Enqueue()

Entfernen

Pop()

Dequeue()

weitere Oper.

(außer Konstruktor  
und Empty())

Top()

---

# Übersicht Elementare Datenstrukturen

---

Operationen

---

Stapel

Schlange

Liste

---

Einfügen

Push()

Enqueue()

Entfernen

Pop()

Dequeue()

weitere Oper.

(außer Konstruktor  
und Empty())

Top()

Head()

Tail()

---

# Übersicht Elementare Datenstrukturen

---

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	Insert()
Entfernen	Pop()	Dequeue()	Delete()
weitere Oper. (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

---

# Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
<b>Einfügen</b>	Push()	Enqueue()	Insert()
– Einschränkung			
<b>Entfernen</b>	Pop()	Dequeue()	Delete()
– Einschränkung			
<b>weitere Oper.</b> (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

# Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
<b>Einfügen</b>	Push()	Enqueue()	Insert()
– Einschränkung	nur oben		
<b>Entfernen</b>	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben		
<b>weitere Oper.</b> (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

# Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
<b>Einfügen</b>	Push()	Enqueue()	Insert()
– Einschränkung	nur oben	nur hinten	
<b>Entfernen</b>	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben	nur vorne	
<b>weitere Oper.</b> (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

# Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
<b>Einfügen</b>	Push()	Enqueue()	Insert()
– Einschränkung	nur oben	nur hinten	nur vorne
<b>Entfernen</b>	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben	nur vorne	beliebig
<b>weitere Oper.</b> (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

# Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
<b>Einfügen</b>	Push()	Enqueue()	Insert()
– Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
<b>Entfernen</b>	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben	nur vorne	beliebig
<b>weitere Oper.</b> (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()



# Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
<b>Einfügen</b>	Push()	Enqueue()	Insert()
– Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
<b>Entfernen</b>	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben	nur vorne	beliebig
<b>weitere Oper.</b> (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

*Alle hier aufgelisteten Operationen außer Search() laufen in  $O(1)$  Zeit!*

# Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
<b>Einfügen</b>	Push()	Enqueue()	Insert()
– Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
<b>Entfernen</b>	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben	nur vorne	beliebig
<b>weitere Oper.</b> (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

*Alle hier aufgelisteten Operationen außer Search() laufen in  $O(1)$  Zeit!*

Listen sind mächtiger als Stapel/Schlangen. Wozu also Stapel/Schlangen?