

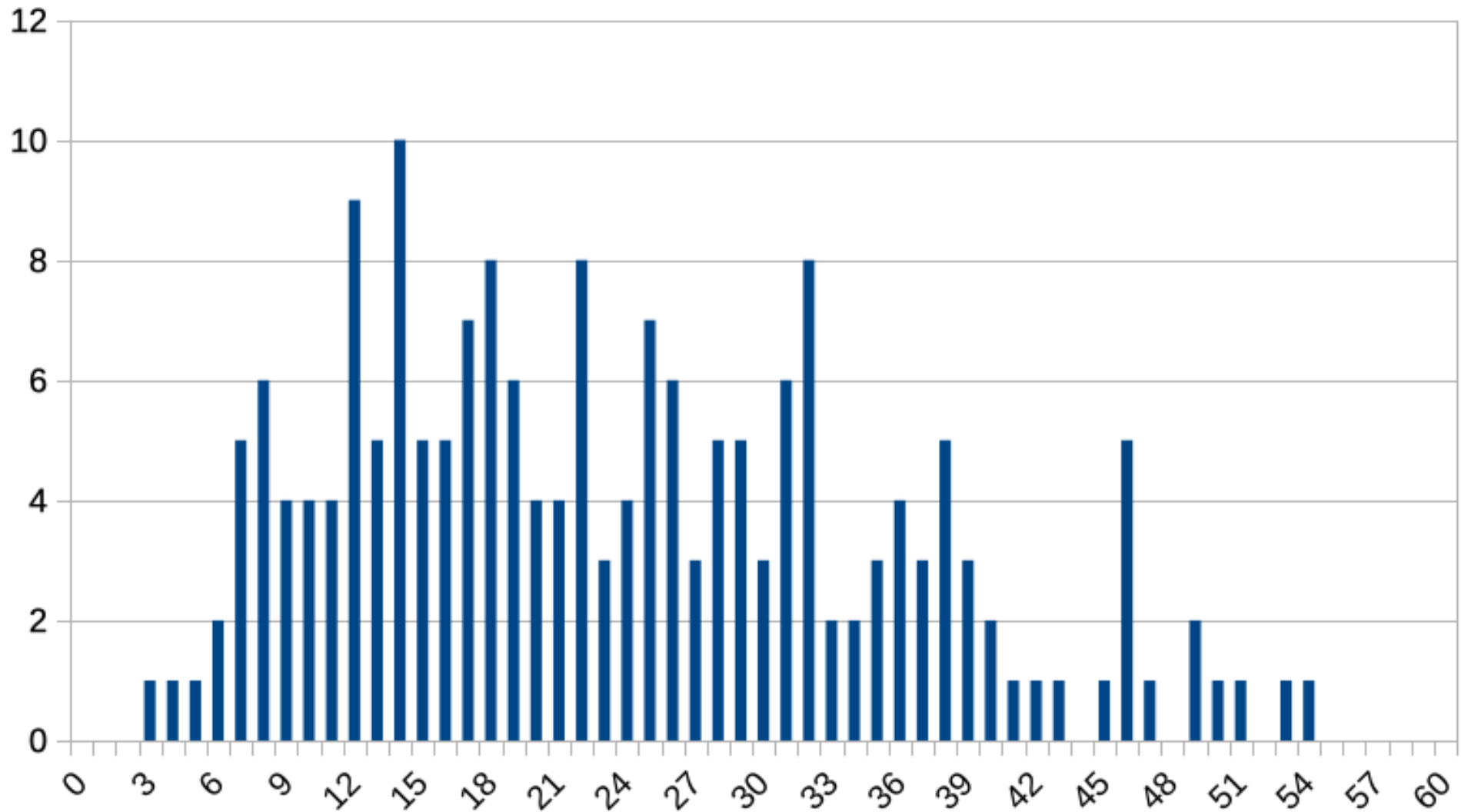
Algorithmen und Datenstrukturen

Wintersemester 2021/22

11. Vorlesung

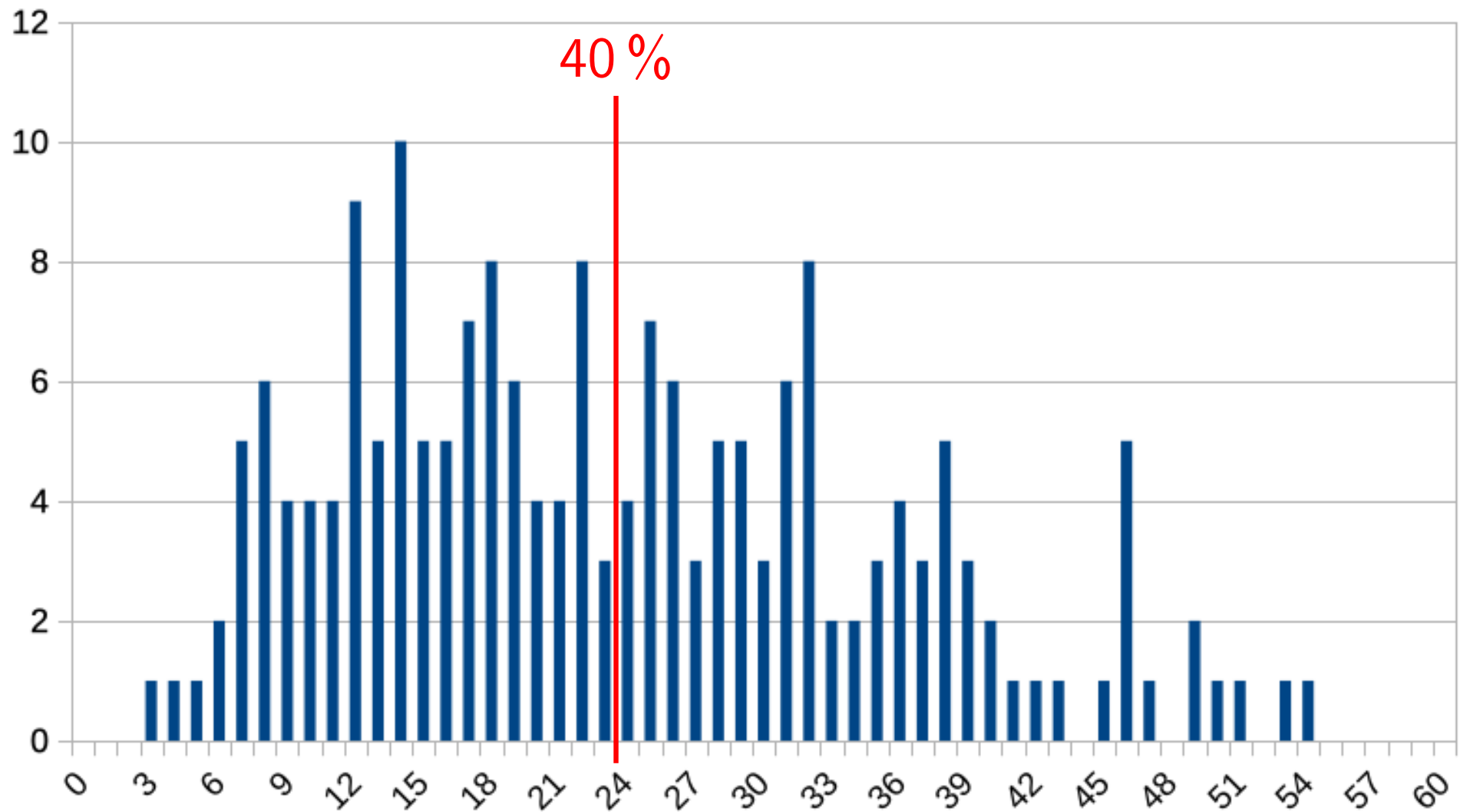
Elementare Datenstrukturen:
Stapel + Schlange + Liste

Ergebnisse 1. Zwischentest



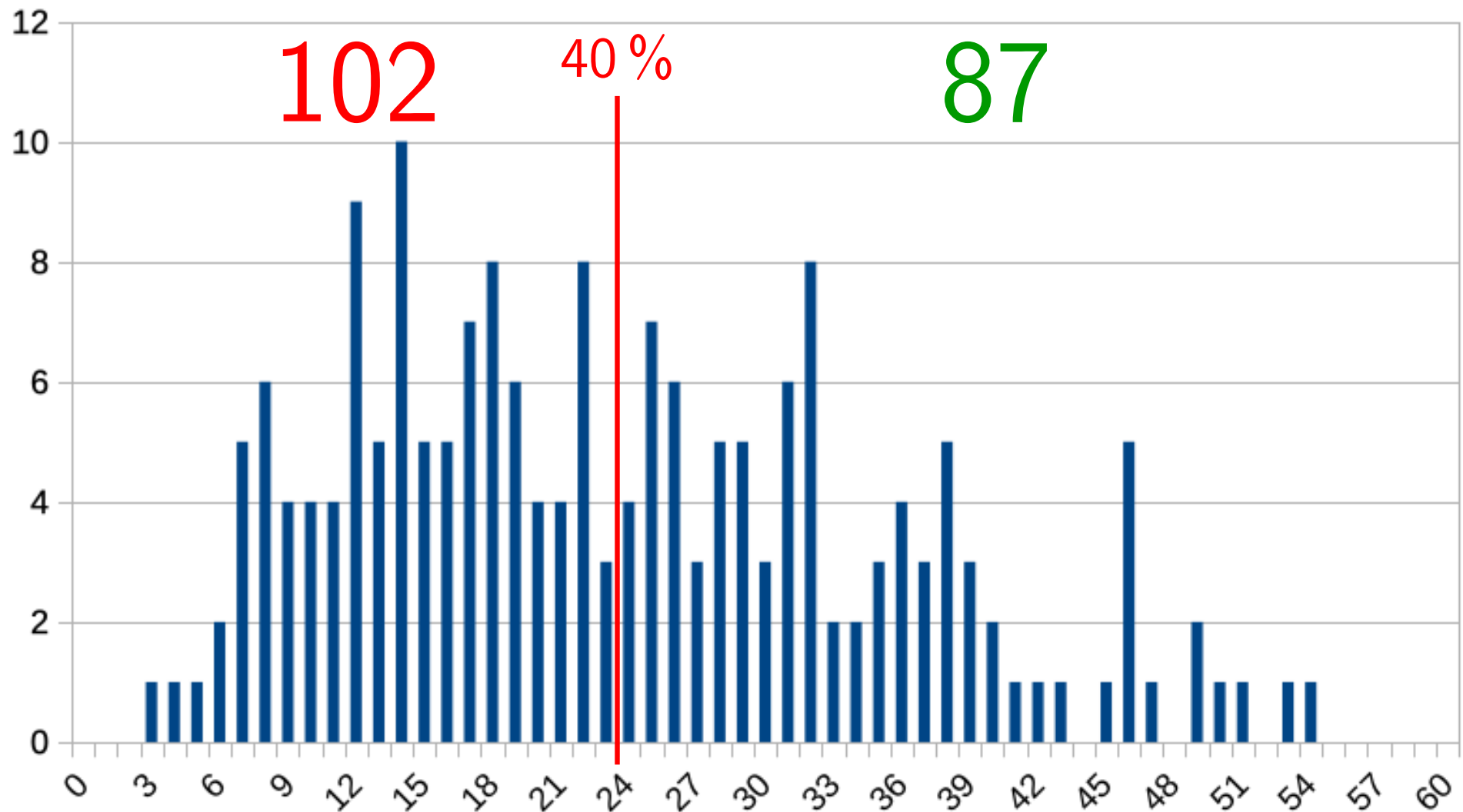
$n = 189$, Median = 21,5, Mittelwert = 23,3

Ergebnisse 1. Zwischentest



$n = 189$, Median = 21,5, Mittelwert = 23,3

Ergebnisse 1. Zwischentest



$n = 189$, Median = 21,5, Mittelwert = 23,3

Zur Erinnerung

Datenstruktur:

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

A large, empty rectangular box with a blue border, intended for a diagram or additional text.A second large, empty rectangular box with a blue border, identical to the one above.

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)

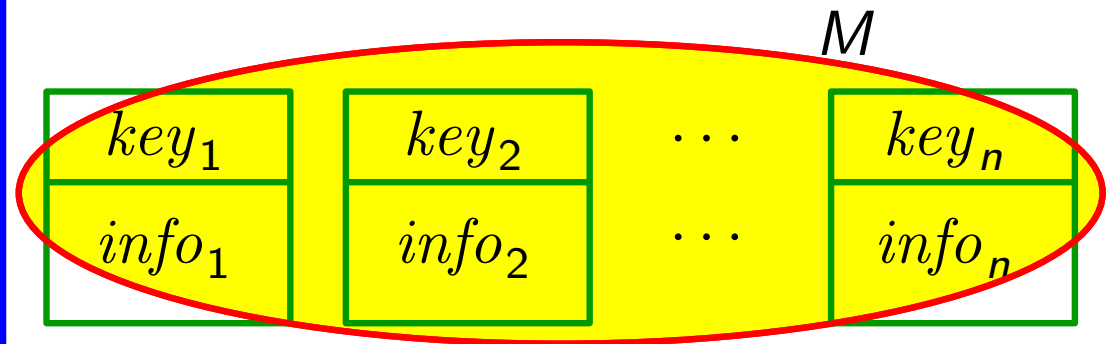
ptr Search(key k)

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  $k$ )`

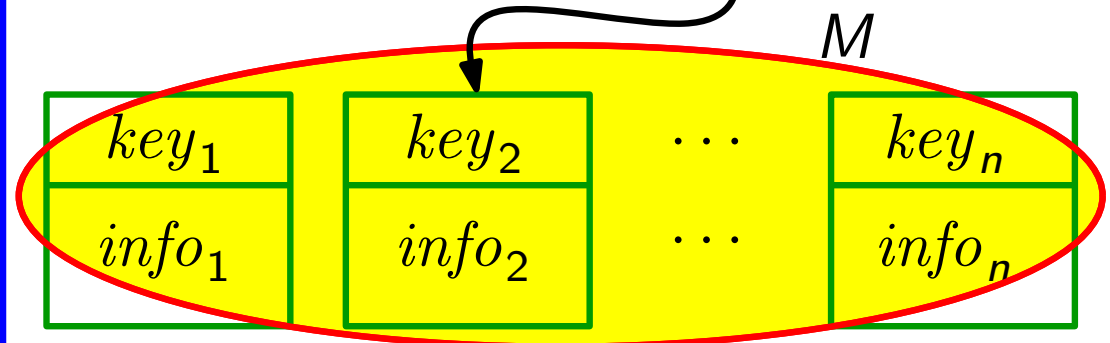
`ptr Minimum()`

`ptr Maximum()`

`ptr Predecessor(ptr  $x$ )`

`ptr Successor(ptr  $x$ )`

Zeiger (pointer, iterator) p



Teil III [CLRS]

Dynamische Menge:

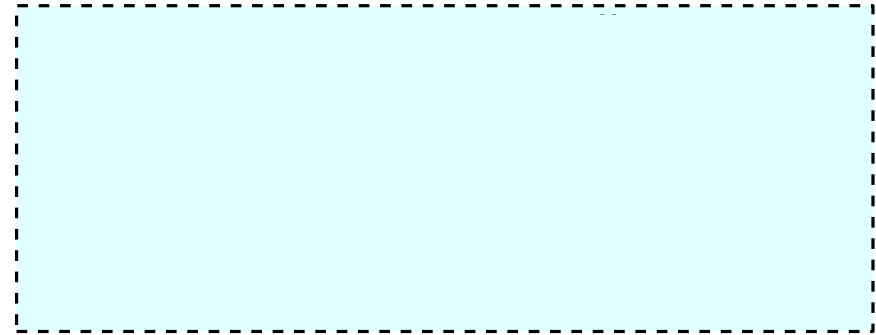
verwaltet Elemente einer sich ändernden Menge M



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]

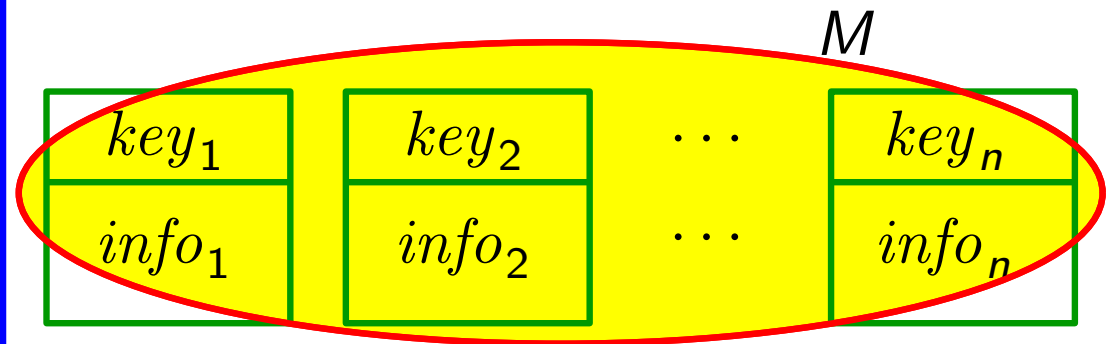
Dynamische Menge:

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

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code>	The diagram illustrates a dynamic set M. It consists of a yellow oval containing several green-bordered boxes. Each box is divided into two horizontal sections: the top section contains a key (key_1, key_2, ..., key_n) and the bottom section contains associated information ($info_1$, $info_2$, ..., $info_n$). Ellipses ($\dots$) are used between the boxes to indicate that there can be more than two elements in the set. The entire set is labeled M in the top right corner.

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code>	• lege neuen Datensatz (k, i) an • $M = M \cup \{(k, i)\}$ • gib Zeiger auf (k, i) zurück
	M The diagram illustrates a dynamic set M. It consists of a sequence of elements, each represented by a box divided into two horizontal sections. The top section of each box contains a key ($key_1, key_2, \dots, key_n$), and the bottom section contains corresponding information ($info_1, info_2, \dots, info_n$). These boxes are arranged horizontally, separated by ellipses. A large red oval encloses the entire sequence of boxes, representing the set M.

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	<i>Funktionalität</i>
ptr Insert(key k , info i) Delete(ptr x)	A diagram illustrating a dynamic set M. It consists of a yellow oval containing several green-bordered boxes. Each box is divided into two horizontal sections: the top section contains a key (key_1, key_2, ..., key_n) and the bottom section contains corresponding information ($info_1$, $info_2$, ..., $info_n$). Ellipses between the boxes indicate that there can be more than two elements in the set. The entire set is labeled with the letter M in the top right corner of the oval.

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code>	• $M = M \setminus \{(x.key, x.info)\}$ The diagram shows a yellow oval labeled M containing a table with two rows and four columns. The first column contains key_1 and $info_1$. The second column contains key_2 and $info_2$. The third column contains three dots $\dots$ in both rows. The fourth column contains key_n and $info_n$. Each cell in the table is enclosed in a green rectangular border.

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
ptr Insert(key k, info i) Delete(ptr x) ptr Search(key k)	M The diagram shows a yellow oval labeled M containing a table-like structure. It consists of three main columns. The first column has two rows: the top row contains key_1 and the bottom row contains $info_1$. The second column has two rows: the top row contains key_2 and the bottom row contains $info_2$. The third column has two rows: the top row contains key_n and the bottom row contains $info_n$. Ellipses ($\dots$) are placed between the second and third columns in both the top and bottom rows, indicating a sequence of elements. Each of the three columns is enclosed in a green rectangular border, and the entire set of columns is enclosed in a red oval.

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code>	• falls vorhanden, gib Zeiger p mit $p.key = k$ zurück • sonst gib Zeiger nil zurück

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code> <code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	• sei $M' = \{(k, i) \in M \mid k < x.key\}$ • falls $M' = \emptyset$, gib <i>nil</i> zurück, • sonst gib Zeiger auf (k^*, i^*) zurück, wobei $k^* = \max_{(k,i) \in M'} k$

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code> <code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	• sei $M' = \{(k, i) \in M \mid k < x.key\}$ • falls $M' = \emptyset$, gib <i>nil</i> zurück, • sonst gib Zeiger auf (k^*, i^*) zurück, wobei $k^* = \max_{(k,i) \in M'} k$

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Implementierung: je nachdem...

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

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

Implementierung: je nachdem... Drei Beispiele!

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung

I. Stapel

„last-in first-out“

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



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

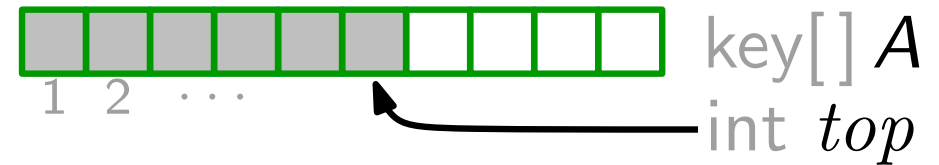
boolean Empty()

Push(key *k*)

key Pop()

key Top()

Implementierung



I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

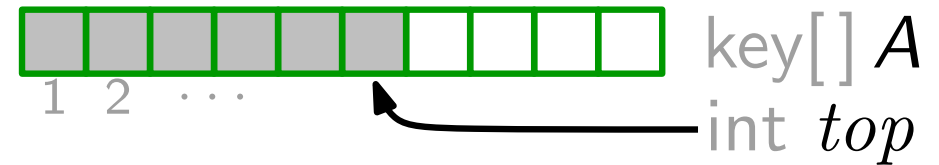
boolean Empty()

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

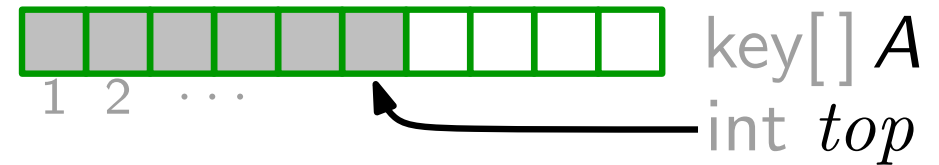
boolean Empty()

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

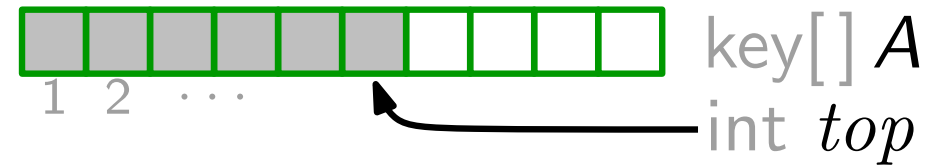
boolean Empty()

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

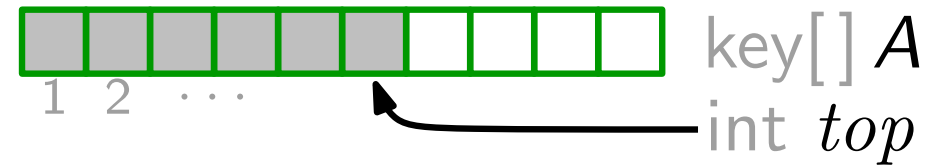
boolean Empty()

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

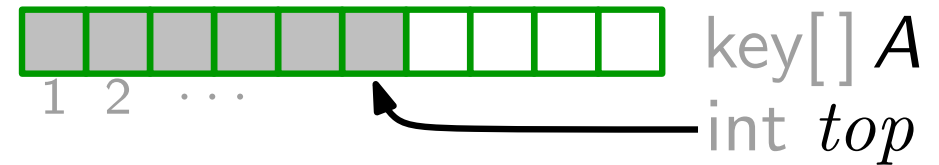
boolean Empty()

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

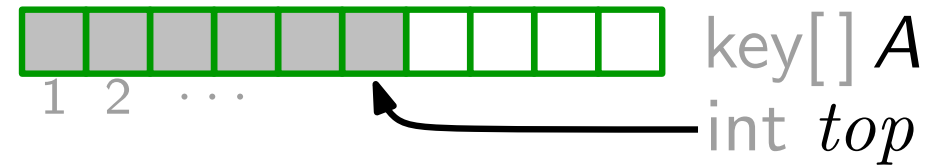
boolean Empty()

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()

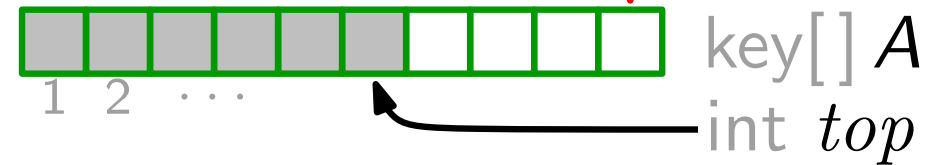
Push(key k)

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

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stack(int n)

boolean Empty()

Push(key k)

key Pop()

key Top()

Implementierung

$A = \text{new key}[1..n]$
 $top = 0$

key[] A
 int top

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

Stack(int n)

boolean Empty()

Push(key k)

key Pop()

key Top()

Implementierung

$A = \text{new key}[1..n]$
 $top = 0$

key[] A
 int top

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

$top = top + 1$ { **if** $top > A.length$ **then**
 $A[top] = k$ { **error** „overflow“

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

Stack(int n)

boolean Empty()

Push(key k)

key Pop()

key Top()

Implementierung

$A = \text{new key}[1..n]$
 $top = 0$

key[] A
 int top

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

$top = top + 1$ { **if** $top > A.length$ **then**
 $A[top] = k$ { **error** „overflow“

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

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

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

Laufzeiten?

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stack(int n)

boolean Empty()

Push(key k)

key Pop()

key Top()

Implementierung

$A = \text{new}^* \text{key}[1..n]$
 $top = 0$

key[] A
 int top

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

$top = top + 1$ { **if** $top > A.length$ **then**
 $A[top] = k$ { **error** „overflow“

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

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

Laufzeiten?

Alle^{*} $O(1)$,
 d.h. konstant.

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

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stack(int n)

boolean Empty()

Push(key k)

key Pop()

key Top()

Implementierung

key[] A
int top

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



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

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung
Stack(int <i>n</i>)	Konstruktor Attribute key[] <i>A</i> int <i>top</i>
boolean Empty()	Methoden Aufgabe: Fertigen Sie ein UML-Diagramm für die Klasse <i>Stack</i> an!
Push(key <i>k</i>)	
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()

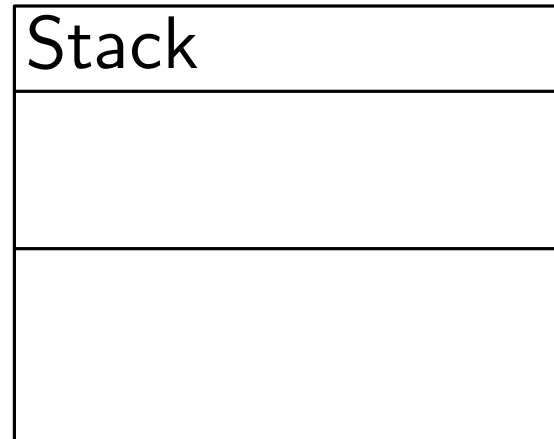
Implementierung

}
Konstruktor
Attribute
{
key[] *A*
int *top*

}
Methoden

Aufgabe:

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



I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stack(int *n*)

boolean Empty()

Push(key *k*)

key Pop()

key Top()

Implementierung

}

Konstruktor

Attribute
{

key[] *A*
int *top*

}

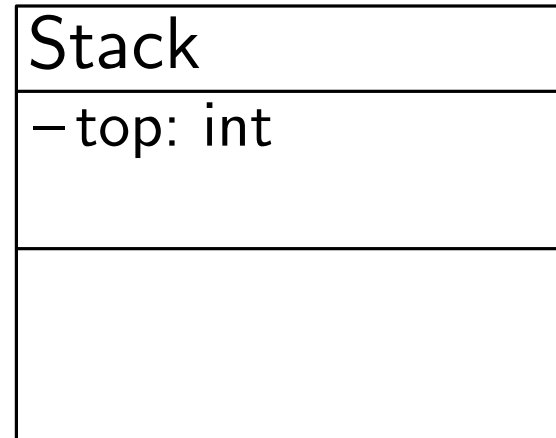
Methoden

Aufgabe:

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

Attribute

Methoden



I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stack(int *n*)

boolean Empty()

Push(key *k*)

key Pop()

key Top()

Implementierung

}
{
Konstruktor
Attribute
{
key[] *A*
int *top*
}

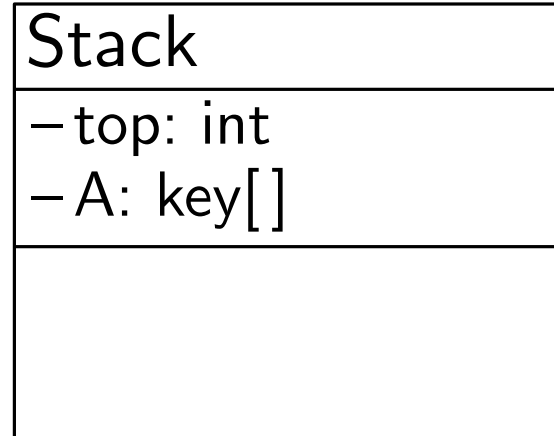
}
{
Methoden

Aufgabe:

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

Attribute

Methoden



I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stack(int *n*)

boolean Empty()

Push(key *k*)

key Pop()

key Top()

Implementierung

}
{
Konstruktor
Attribute
{
key[] *A*
int *top*
}

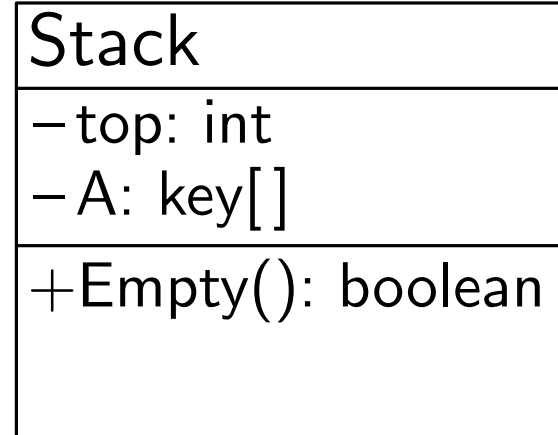
{
Methoden

Aufgabe:

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

Attribute

Methoden



I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

Stack(int *n*)

boolean Empty()

Push(key *k*)

key Pop()

key Top()

Implementierung

}
{
Konstruktor
Attribute
{
key[] *A*
int *top*
}

{
Methoden

Aufgabe:

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

Attribute

Methoden

Stack

– top: int

– A: key[]

+ Empty(): boolean

+ Push(key)

...

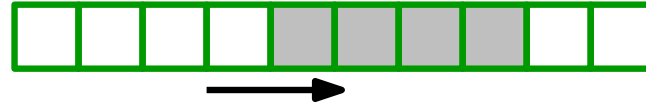
II. Schlange

verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung

II. Schlange

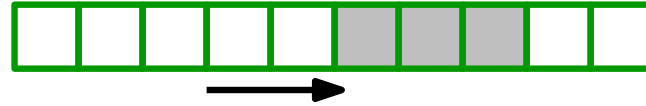


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung

II. Schlange



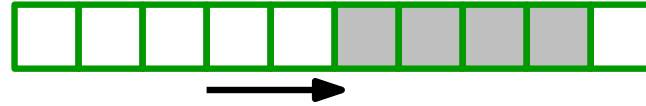
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp

Implementierung

II. Schlange

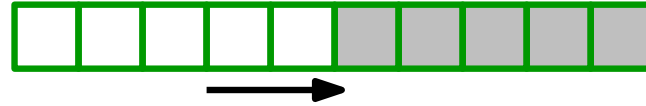


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung

II. Schlange



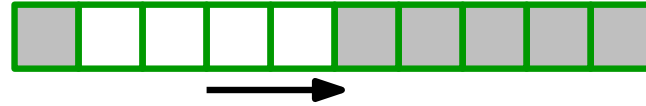
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp

Implementierung

II. Schlange

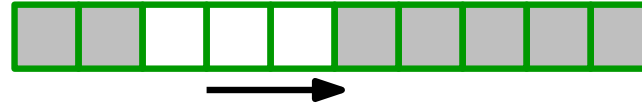


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung

II. Schlange

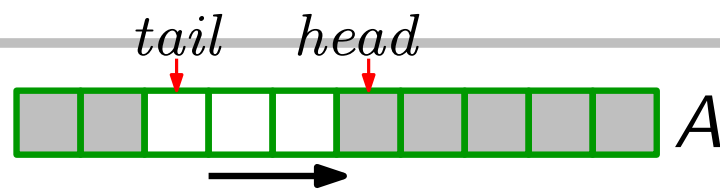


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp	Implementierung

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

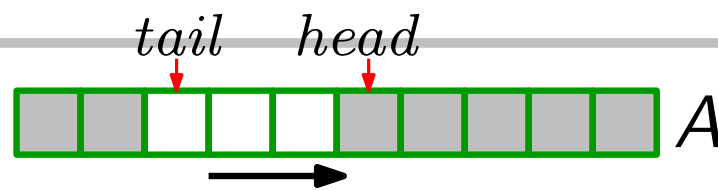


Abs. Datentyp

Implementierung

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

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



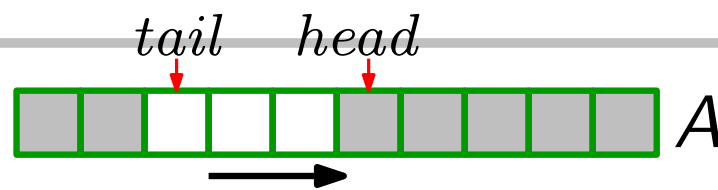
Abs. Datentyp

Queue(int *n*)

Implementierung

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

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp

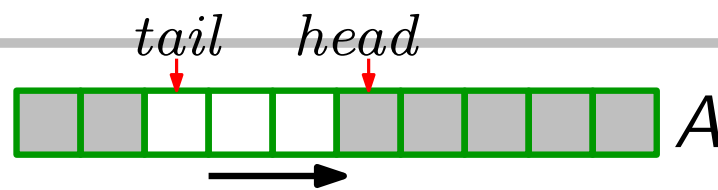
Queue(int *n*)

Implementierung

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

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

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp

Queue(int *n*)

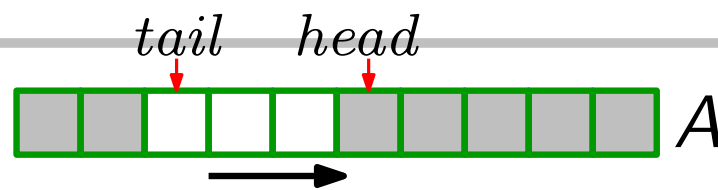
boolean Empty()

Implementierung

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

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

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Abs. Datentyp

Queue(int *n*)

boolean Empty()

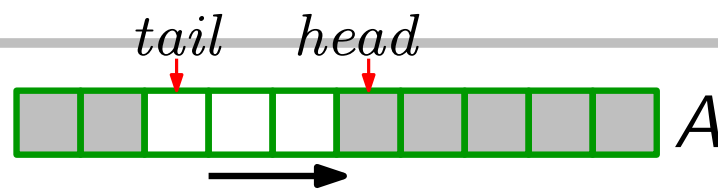
Implementierung

A = **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*

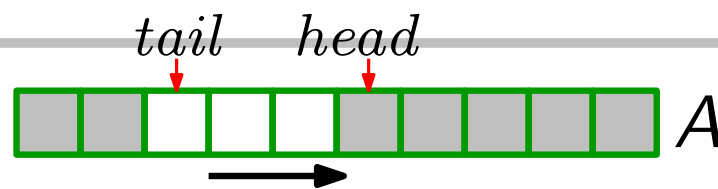
Implementierung

A = **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

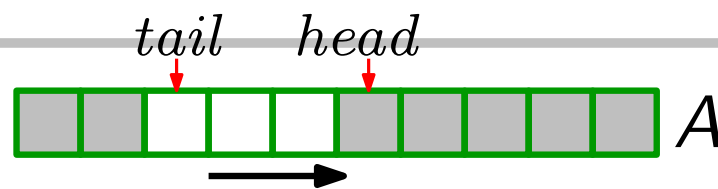
$A = \text{new key}[1..n]$
 $tail = head = 1$

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

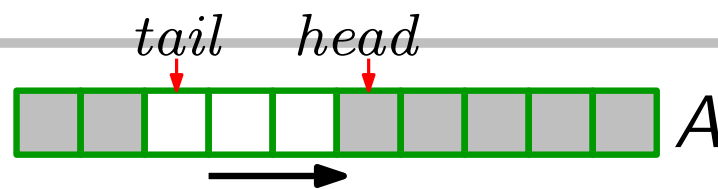
A = **new** key[1..*n*]
tail = *head* = 1

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

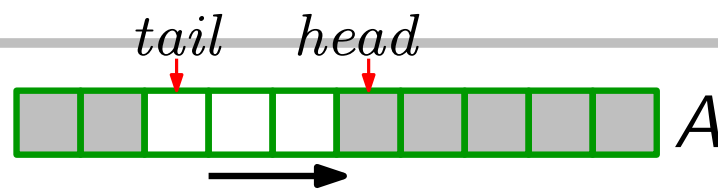
key[] *A*
int *tail*
int *head*

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

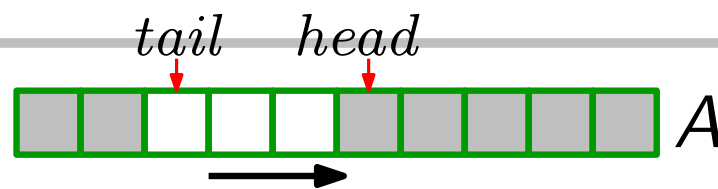
k = *A*[*head*]

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

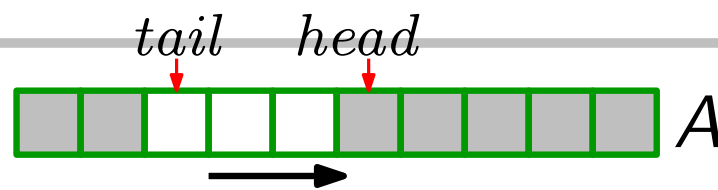
A[*tail*] = *k*

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

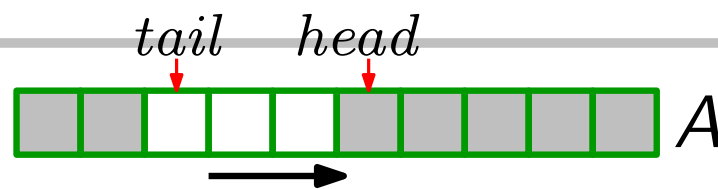


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



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

II. Schlange

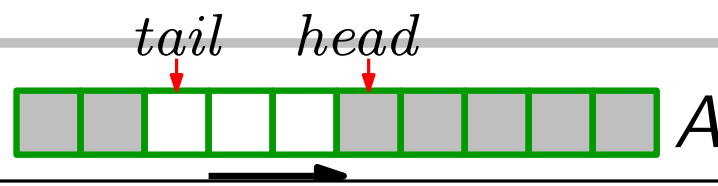


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



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

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

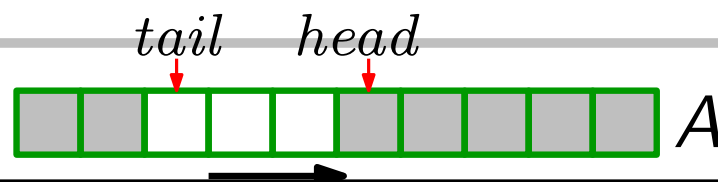
... was er selbst ausführt



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

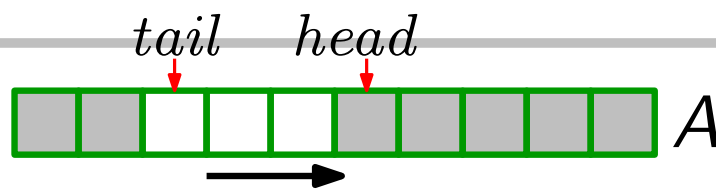
... was er selbst ausführt



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

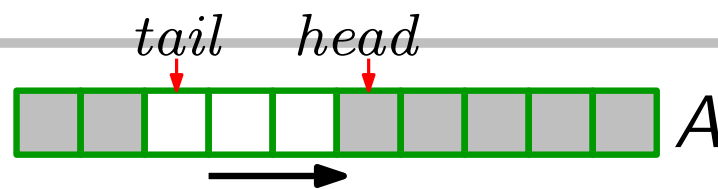


verwaltet sich ändernde Menge nach *FIFO-Prinzip*



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

II. Schlange



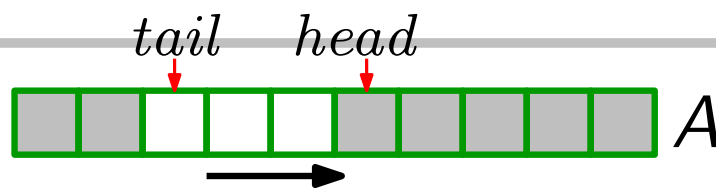
verwaltet sich ändernde Menge nach *FIFO-Prinzip*



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

Laufzeiten?

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



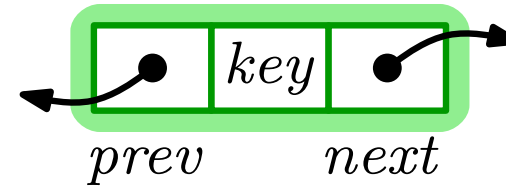
Abs. Datentyp	Implementierung	
Queue(int n)	$A = \text{new}^* \text{key}[1..n]$ $tail = head = 1$	$\text{key}[]$ A $\text{int } tail$ $\text{int } head$
Aufgabe: Fangen Sie underflow & overflow ab!		
boolean Empty()	if $head == tail$ then return true else return false	
Enqueue(key k) <i>stell neues Element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

Laufzeiten?
 Alle^{*} $O(1)$.

III. Liste

Abs. Datentyp	Implementierung

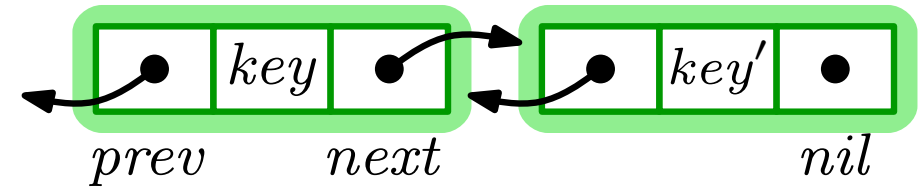
III. Liste



Abs. Datentyp

Implementierung

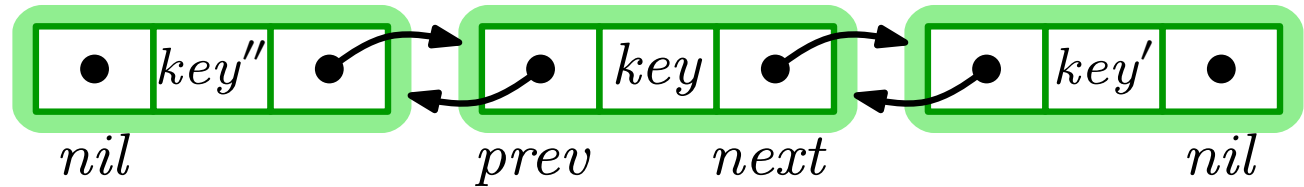
III. Liste



Abs. Datentyp

Implementierung

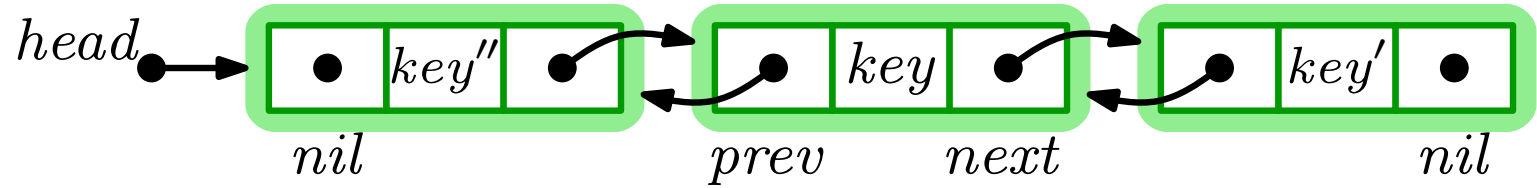
III. Liste



Abs. Datentyp

Implementierung

III. Liste

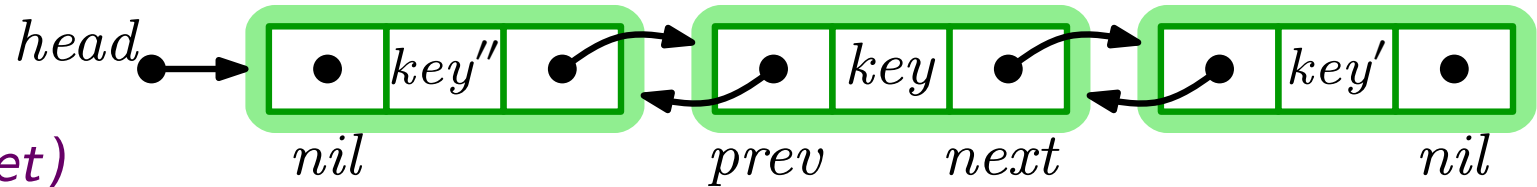


Abs. Datentyp

Implementierung

III. Liste

(doppelt verkettet)

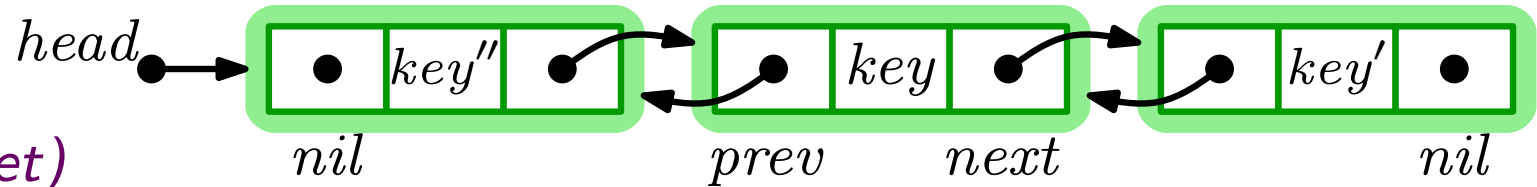


Abs. Datentyp

Implementierung

III. Liste

(doppelt verkettet)



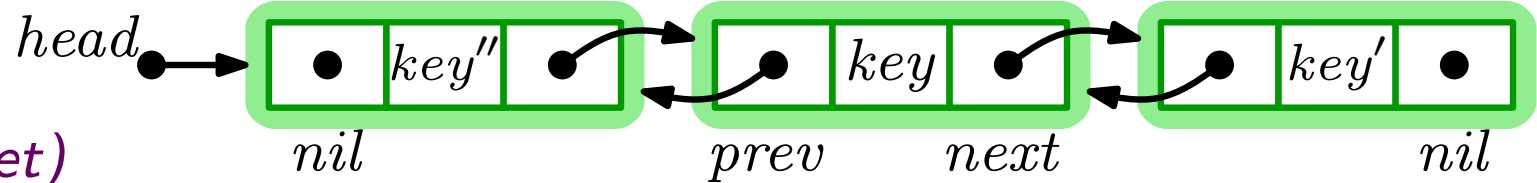
Abs. Datentyp

Implementierung

ptr $head$

III. Liste

(doppelt verkettet)



Abs. Datentyp

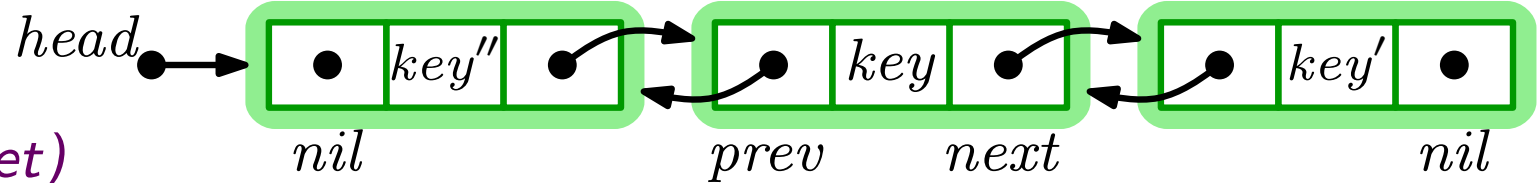
Implementierung

key	key
ptr	prev
ptr	next

ptr	head
-----	------

III. Liste

(doppelt verkettet)



Abs. Datentyp

Implementierung

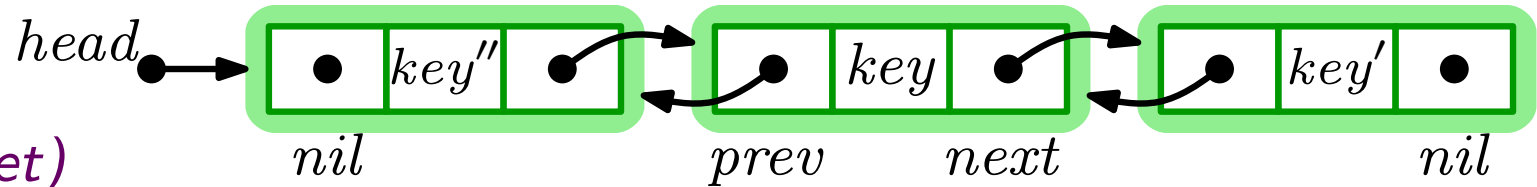
Item

key	key
ptr	prev
ptr	next

ptr head

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

Implementierung

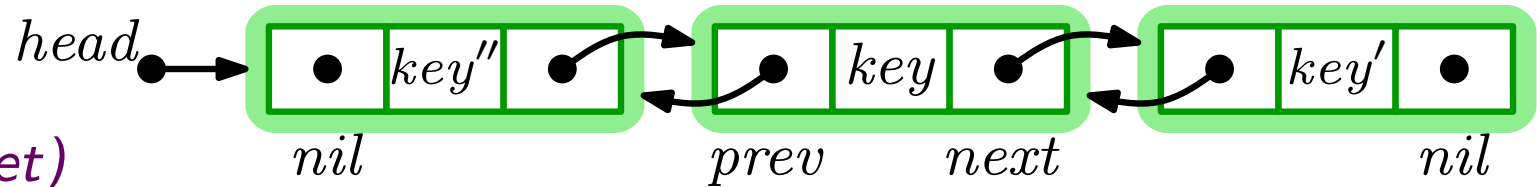
Item

key	key
ptr	prev
ptr	next

ptr head

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

Implementierung

head = nil

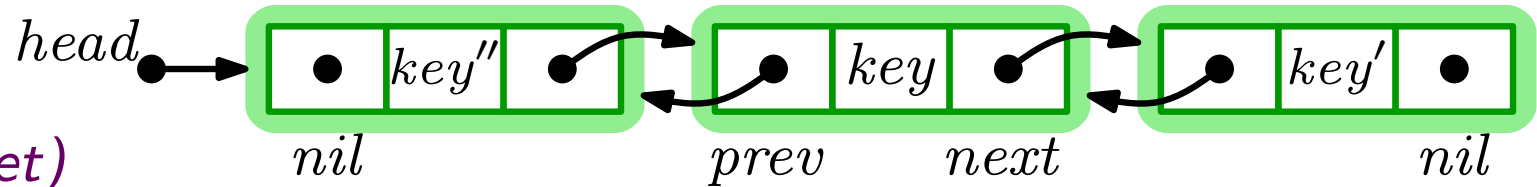
Item

key	key
ptr	prev
ptr	next

ptr *head*

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

Implementierung

$head = nil$

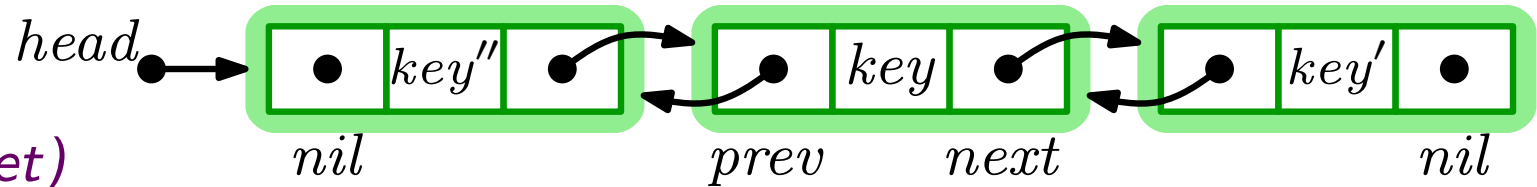
Item

key	key
ptr	prev
ptr	next

ptr head

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

Implementierung

$head = nil$

Item

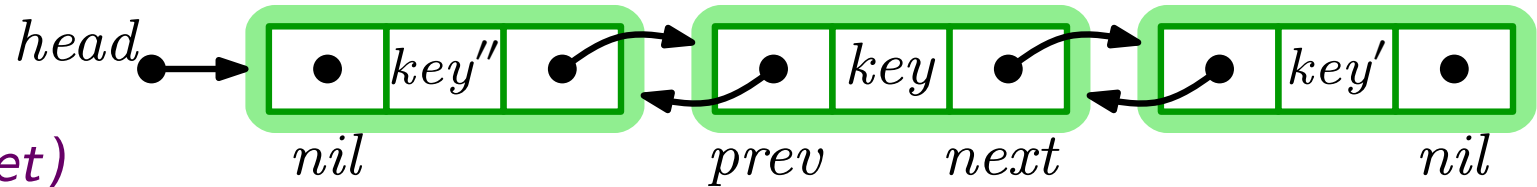
key	key
ptr	prev
ptr	next

ptr $head$

$x = head$

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

Implementierung

$head = nil$

Item

key	key
ptr	prev
ptr	next

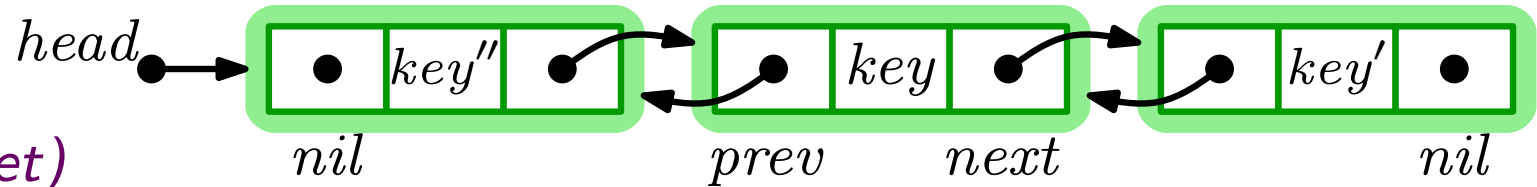
ptr head

$x = head$

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

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

Implementierung

$head = nil$

Item

key	key
ptr	prev
ptr	next

ptr head

$x = head$

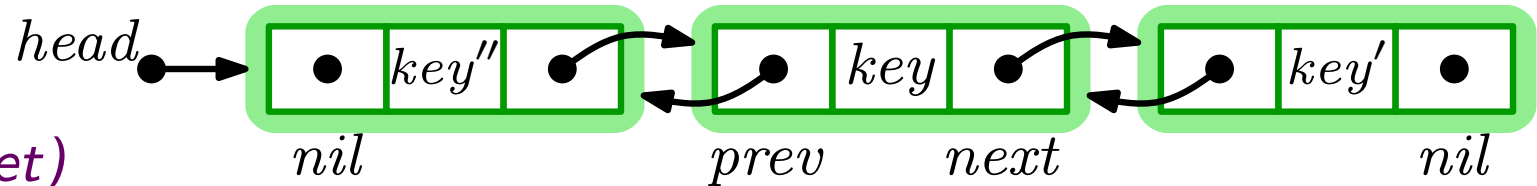
while $x \neq nil$ **and** $x.key \neq k$ **do**

$x = x.next$

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

$x = head$

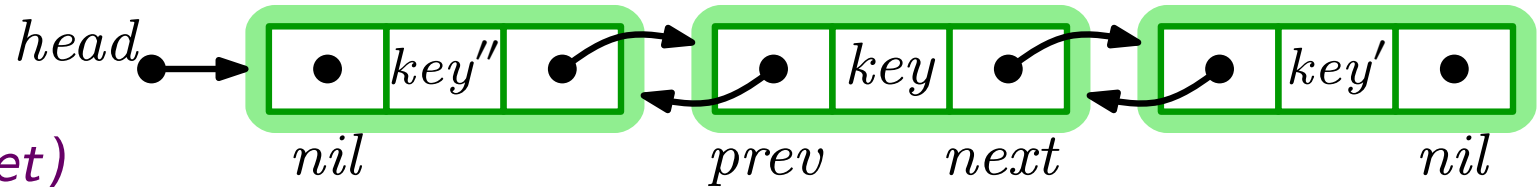
while $x \neq nil$ **and** $x.key \neq k$ **do**

$x = x.next$

return x

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()

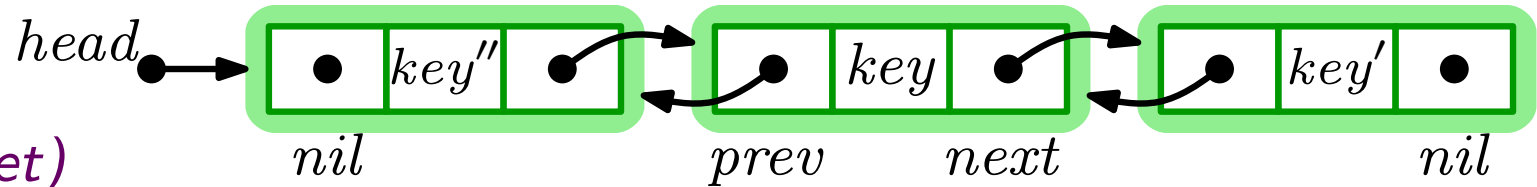
$x.key = k$; $x.prev = nil$; $x.next = head$

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()

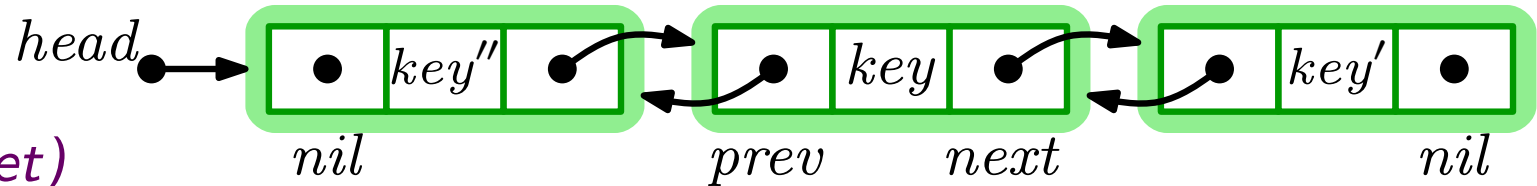
$x.key = k; x.prev = nil; x.next = head$

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()

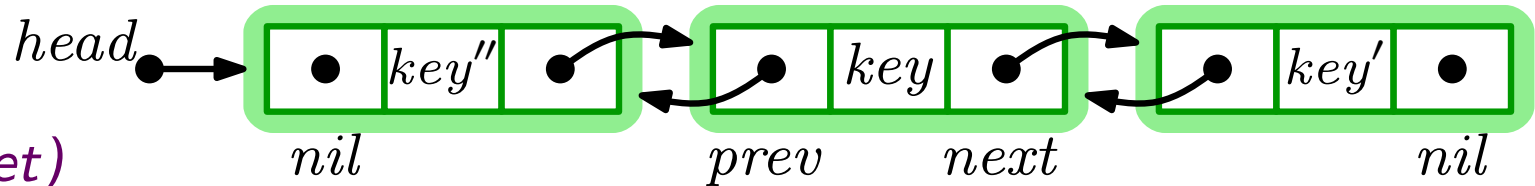
$x.key = k; x.prev = nil; x.next = head$

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()

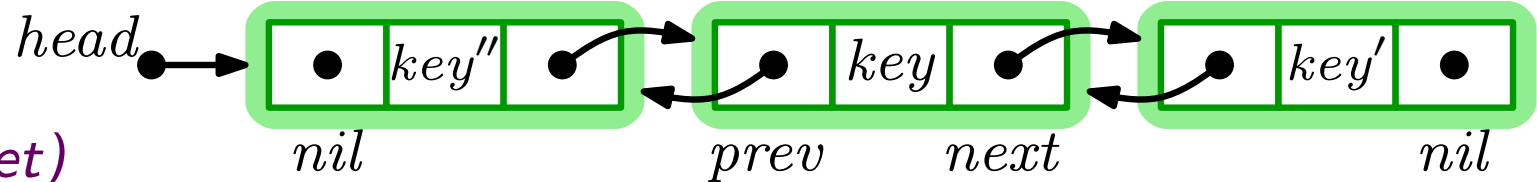
$x.key = k; x.prev = nil; x.next = head$

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

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

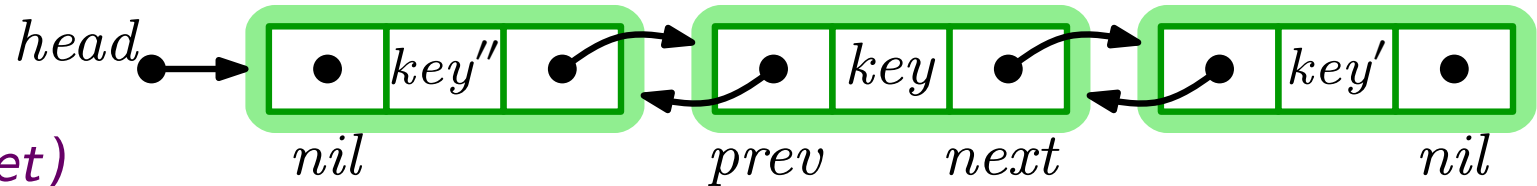
$x.key = k; x.prev = nil; x.next = head$

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

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

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

└ *x* = *x.next*

return *x*

x = **new** Item(~~↗~~ *k*, *head*)

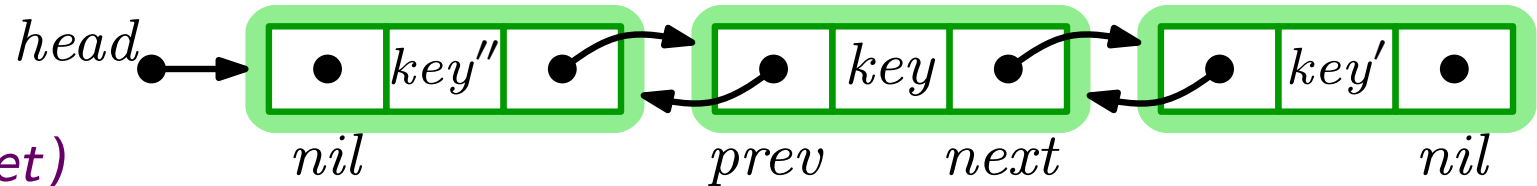
~~*x.key* = *k*, *x.prev* = *nil*; *x.next* = *head*~~

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

$x = x.next$

return x

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

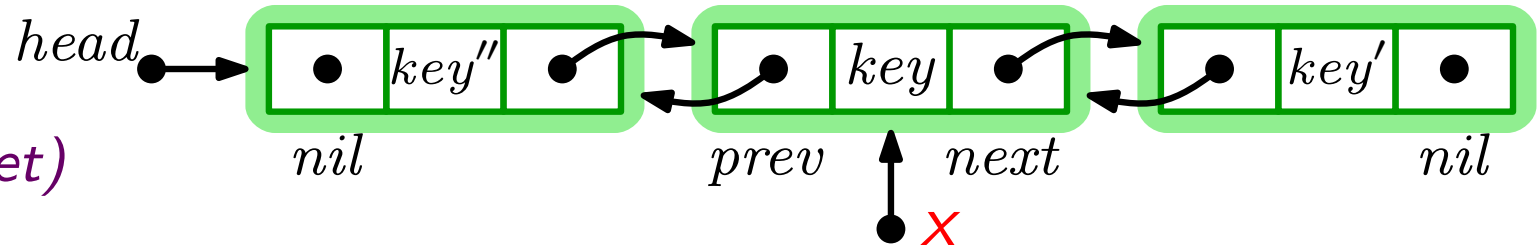
~~$x.key = k, x.prev = nil; x.next = head$~~

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*)

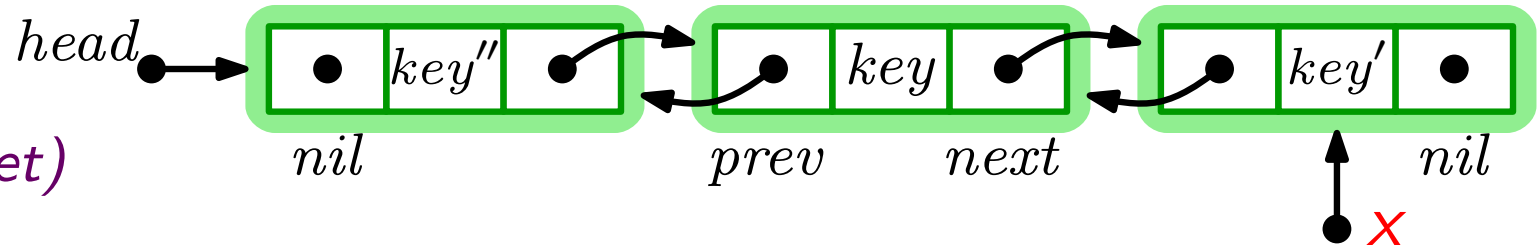
~~*x.key* = *k*, *x.prev* = *nil*; *x.next* = *head*~~

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

$x = x.next$

return x

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

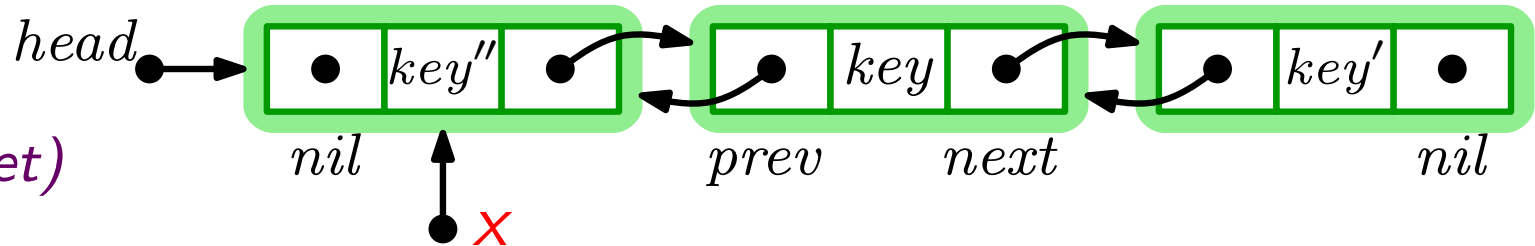
~~$x.key = k, x.prev = nil; x.next = head$~~

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

$x = x.next$

return x

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

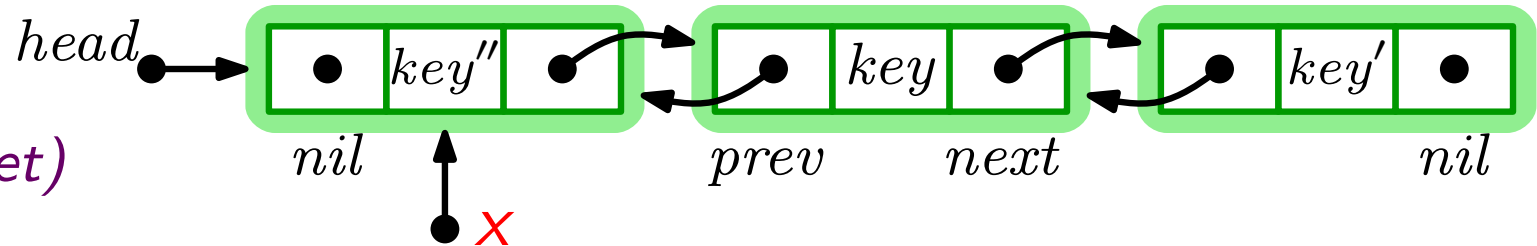
~~$x.key = k, x.prev = nil; x.next = head$~~

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

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

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

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

└ $x = x.next$

return x

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

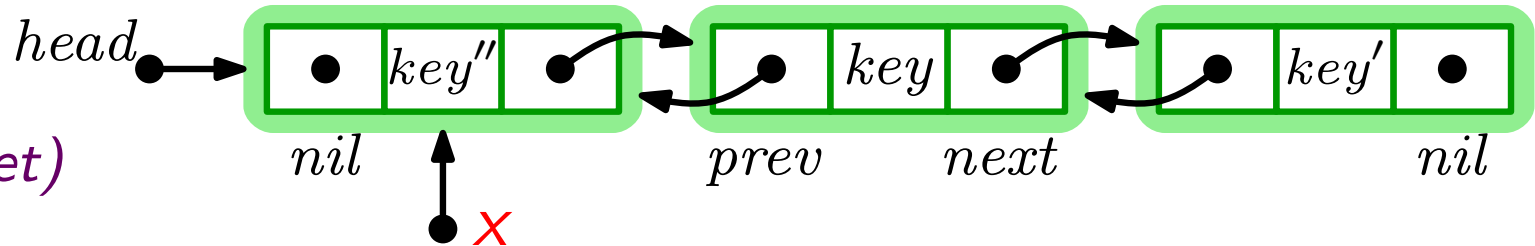
~~$x.key = k, x.prev = nil; x.next = head$~~

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

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

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

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

$x = x.next$

return x

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

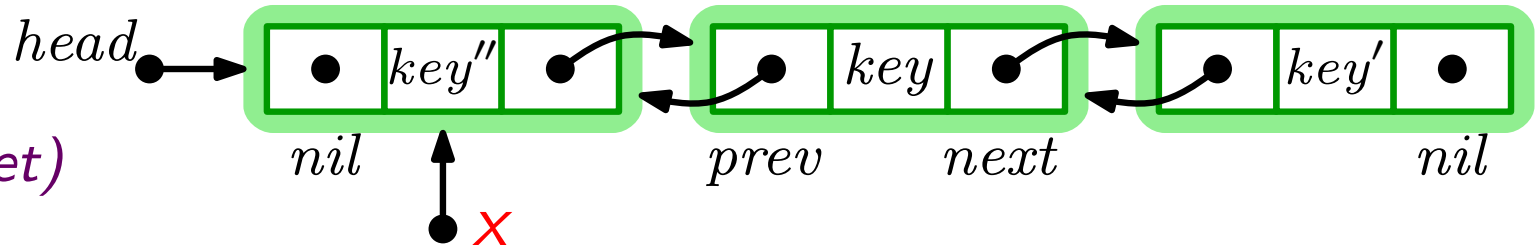
~~$x.key = k, x.prev = nil; x.next = head$~~

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

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

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

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*)

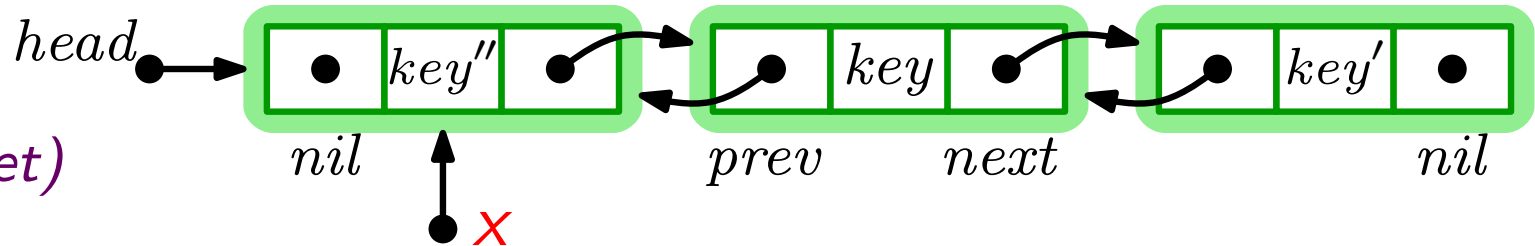
~~*x.key* = *k*, *x.prev* = nil; *x.next* = *head*~~

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)

ptr Insert(key k)

Delete(ptr x)

Implementierung

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

```

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

```

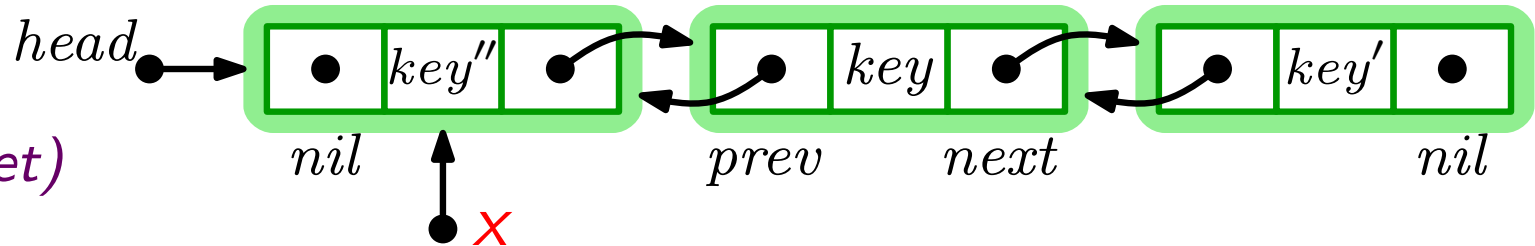
```

 $x = \text{new Item}(\textcolor{red}{k}, head)$ 
 $x.key = k, x.prev = nil; x.next = head$ 
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)

Delete(ptr x)

Implementierung

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

```

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

```

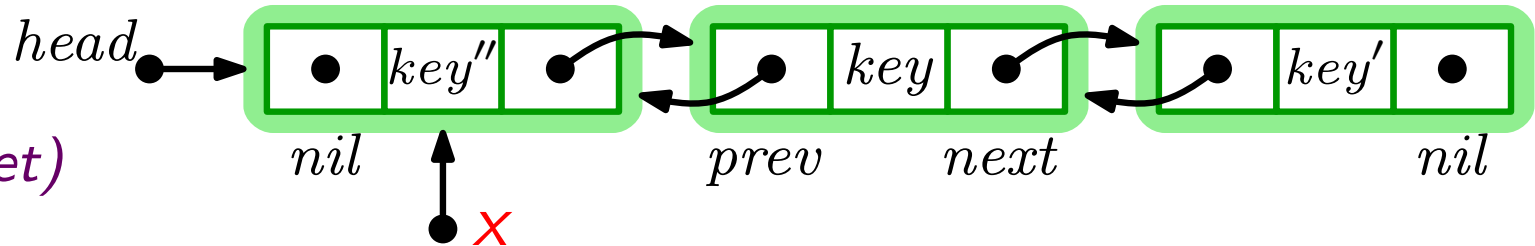
```

 $x = \text{new Item}(\textcolor{red}{k}, head)$ 
 $x.key = k, x.prev = nil; x.next = head$ 
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 ptr $prev$ ptr $next$
ptr $head$		

```

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

```

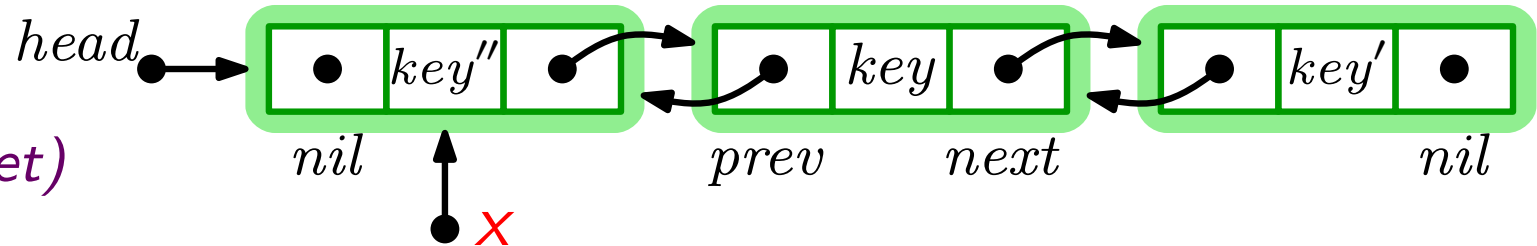
```

 $x = \text{new Item}(\textcolor{red}{k}, head)$ 
 $x.key = k, x.prev = nil; x.next = head$ 
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)	Item
	key = k	key key
	next = p	ptr prev
	prev = nil	ptr next
		ptr head

```

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

```

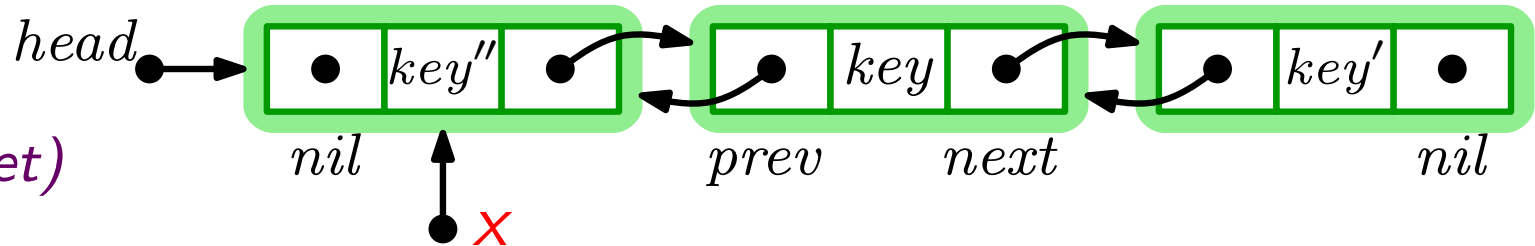
```

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) $O(n)$

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

Delete(ptr x)

Implementierung

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

```

Von Pseudocode zu Javacode: (1) Item

Item(key <i>k</i> , ptr <i>p</i>) <i>key</i> = <i>k</i> <i>next</i> = <i>p</i> <i>prev</i> = <i>nil</i>	Item key <i>key</i> ptr <i>prev</i> ptr <i>next</i>
---	--

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  $\neq$  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  $\neq$  nil and x.key  $\neq$  k do
```

```
         $\sqsubset$  x = x.next
```

```
    return x
```

Von Pseudocode zu Javacode: (2) List

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

```
    x = head
```

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

```
         $\sqsubset$  x = x.next
```

```
    return x
```



Von Pseudocode zu Javacode: (2) List

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

```
    x = head
```

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

```
         $\sqsubset$  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!"?

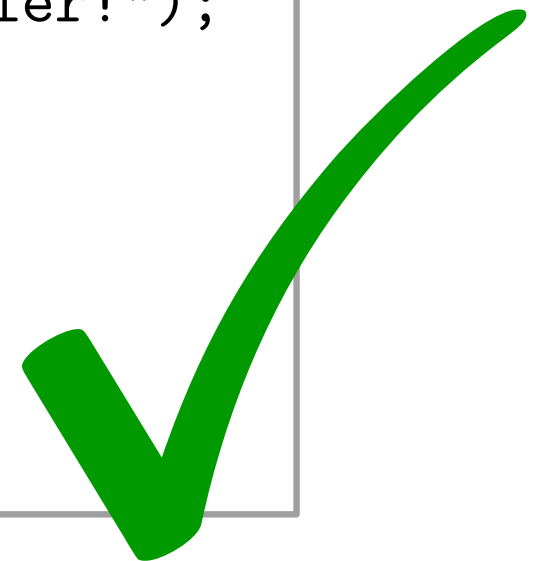
Item.java

```
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

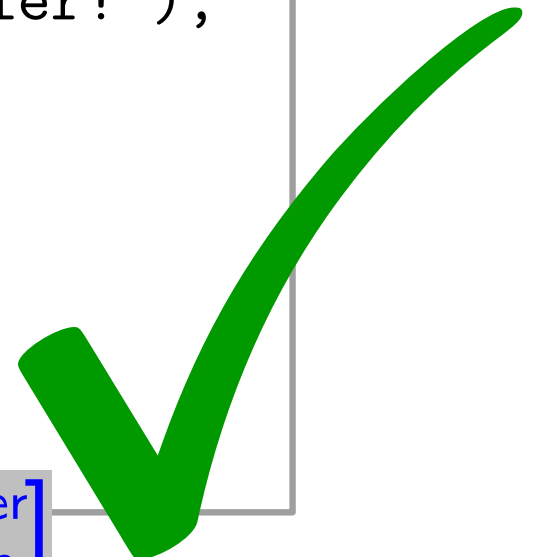
```
public Item search(Object k) {
    Item x = head;    !k.equals(x.getKey())
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

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
Einfügen	Push()	Enqueue()	Insert()
– Einschränkung			
Entfernen	Pop()	Dequeue()	Delete()
– Einschränkung			
weitere Oper. (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

Übersicht Elementare Datenstrukturen

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

Übersicht Elementare Datenstrukturen

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

Übersicht Elementare Datenstrukturen

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

Übersicht Elementare Datenstrukturen

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

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	Insert()
– Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
Entfernen	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben	nur vorne	beliebig
weitere Oper. (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
Einfügen	Push()	Enqueue()	Insert()
– Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
Entfernen	Pop()	Dequeue()	Delete()
– Einschränkung	nur oben	nur vorne	beliebig
weitere Oper. (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?