Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität

Institut für Informatik

Algorithmen und Datenstrukturen

Wintersemester 2023/24

11. Vorlesung

Elementare Datenstrukturen: Stapel + Schlange + Liste

Prof. Dr. Alexander Wolff *Lehrstuhl für Informatik I*

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

stellt folgende Operationen bereit:

$O(1)$ Insert, FindMax, ExtractMax, IncreaseKey $O(n)$

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)	

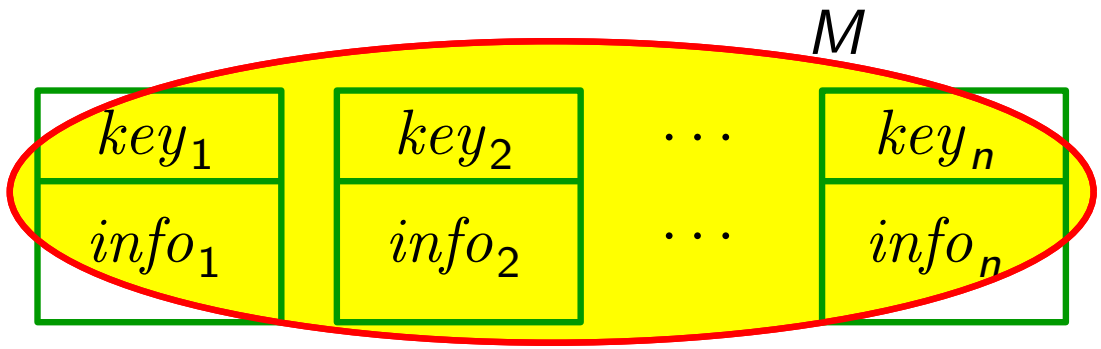
Teil III [CLRS]

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)



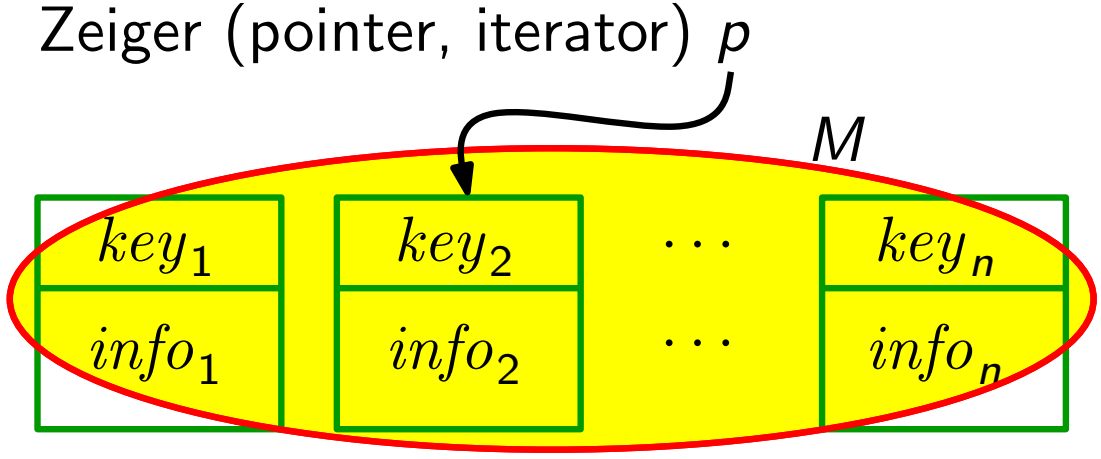
Teil III [CLRS]

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)



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 ≠ 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 ≠ nil do
    gib p.key aus
    p = M.Successor(p)
```

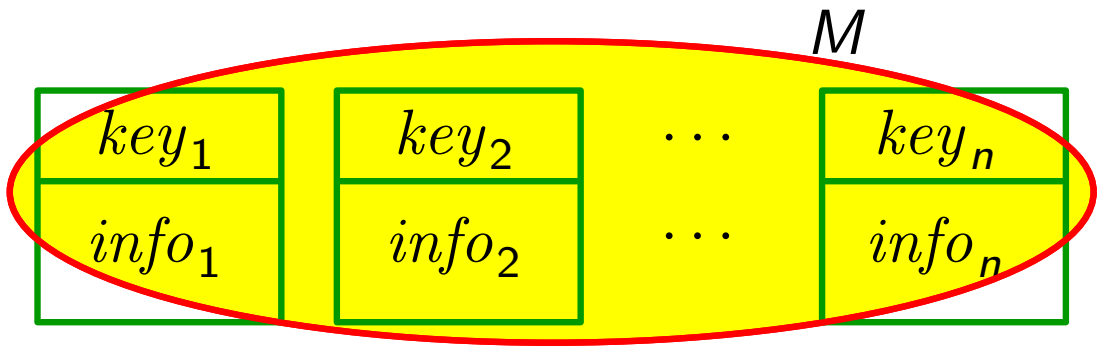
Teil III [CLRS]

Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp

ptr Insert(key k, info i)

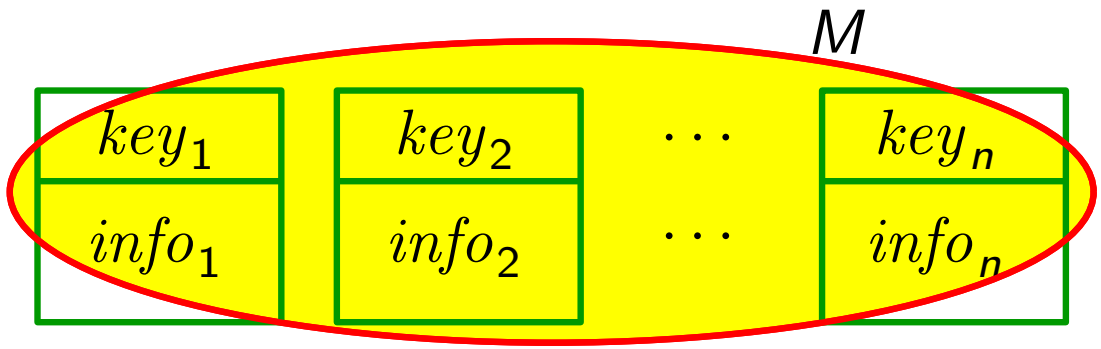


Teil III [CLRS]

Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	*Funktionalität*
ptr Insert(key k, info i)	

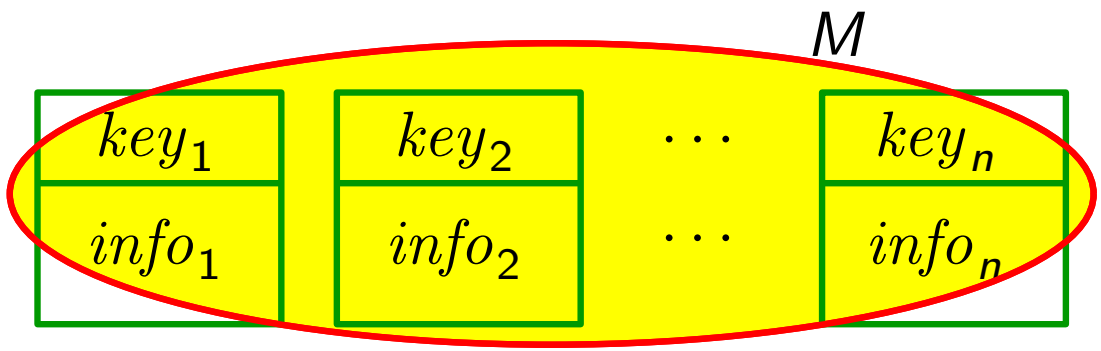


Teil III [CLRS]

Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

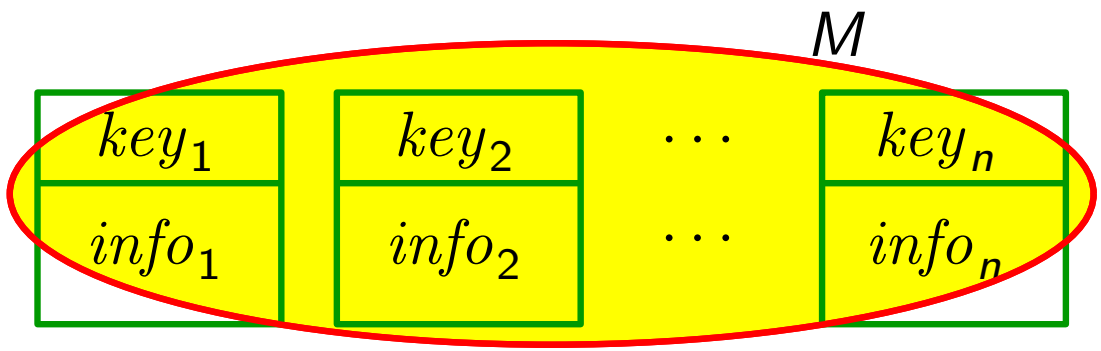
Abstrakter Datentyp	*Funktionalität*
ptr Insert(key k, info i)	• lege neuen Datensatz (k, i) an • $M = M \cup \{(k, i)\}$ • gib Zeiger auf (k, i) zurück



Teil III [CLRS]

Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

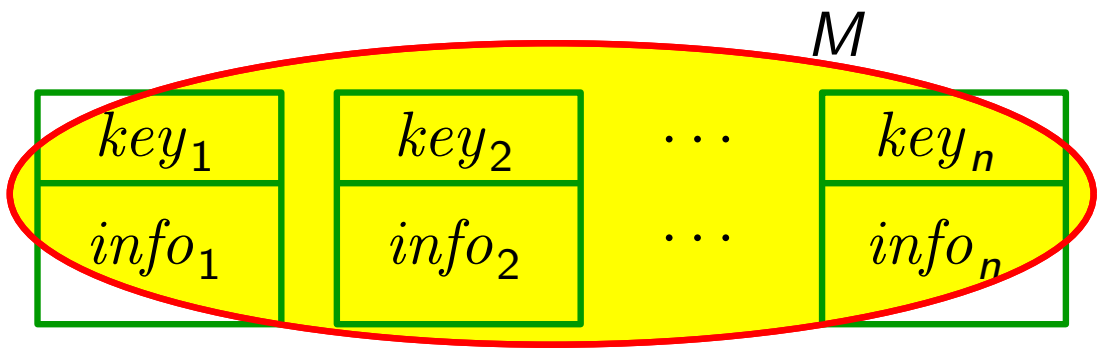
Abstrakter Datentyp	*Funktionalität*
ptr Insert(key k, info i) Delete(ptr x)	

Teil III [CLRS]

Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

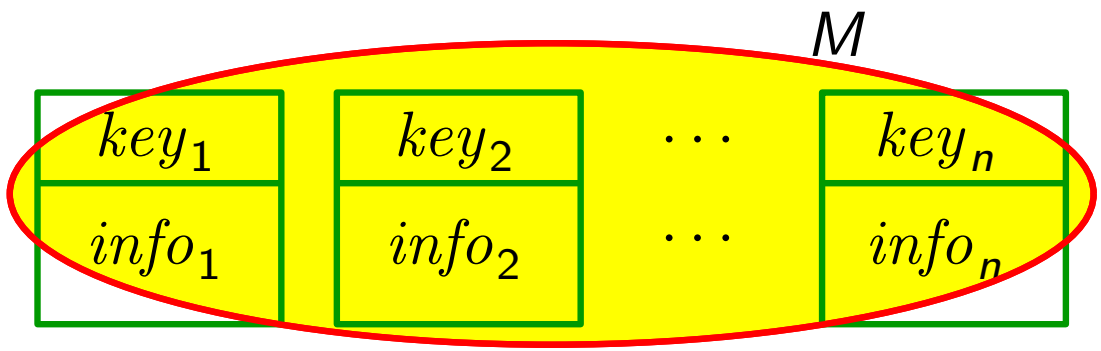
Abstrakter Datentyp	*Funktionalität*
ptr Insert(key k, info i)	
Delete(ptr x)	• $M = M \setminus \{(x.key, x.info)\}$



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)	

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

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)	
ptr Minimum()	• sei $M' = \{(k,i) \in M \mid k < x.key\}$
ptr Maximum()	• falls $M' = \emptyset$, gib nil zurück,
ptr Predecessor(ptr x)	• sonst gib Zeiger auf $(k^\star, i^\star)$ zurück, wobei $k^\star = \max_{(k,i)\in M'} k$
ptr Successor(ptr x)	

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)	
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	*Funktionalität*
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	*Funktionalität*
ptr Insert(key k, info i) Delete(ptr x)	} Änderungen
ptr Search(key k) 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*	
ptr Insert(key k, info i)	Änderungen	**Wörterbuch**
Delete(ptr x)		
ptr Search(key k)	Anfragen	
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	*Funktionalität*	
ptr Insert(key k, info i)	Änderungen	**Wörterbuch**
Delete(ptr x)		
ptr Search(key k)		
ptr Minimum()	Anfragen	
ptr Maximum()		
ptr Predecessor(ptr x)		
ptr Successor(ptr x)		

Implementierung: je nachdem...

Teil III [CLRS]

Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	*Funktionalität*	
ptr Insert(key k, info i)	Änderungen	**Wörterbuch**
Delete(ptr x)		
ptr Search(key k)		
ptr Minimum()	Anfragen	
ptr Maximum()		
ptr Predecessor(ptr x)		
ptr Successor(ptr x)		

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**

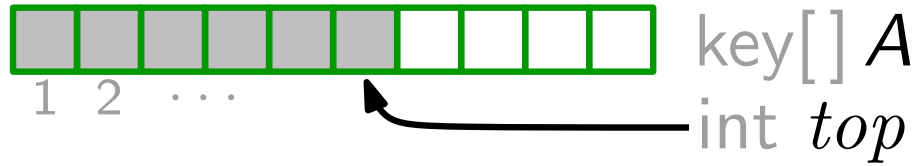
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	**Implementierung**
boolean Empty()	
Push(key k)	
key Pop()	
key Top()	

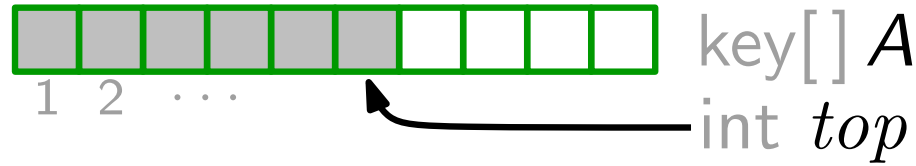
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	Implementierung
	
boolean Empty()	
Push(key k)	
key Pop()	
key Top()	

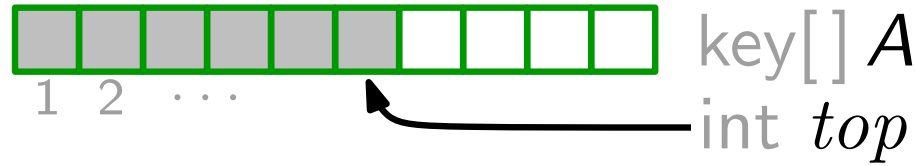
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	Implementierung
	
boolean Empty()	**if** $top == 0$ **then return** *true* **else return** *false*
Push(key k)	
key Pop()	
key Top()	

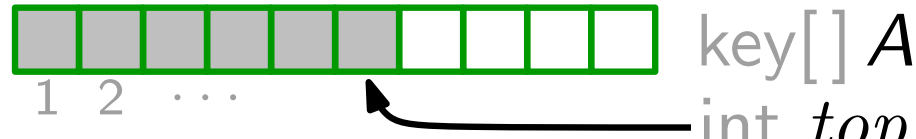
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	Implementierung
	
boolean Empty()	**if** $top == 0$ **then return** *true* **else return** *false*
Push(key k)	$top = top + 1$ $A[top] = k$
key Pop()	
key Top()	

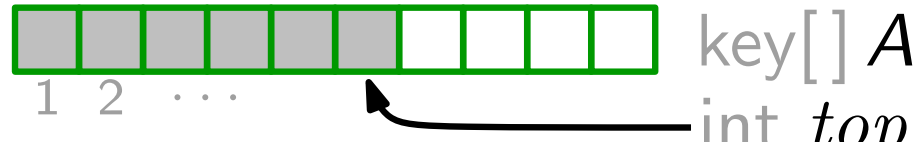
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	Implementierung
	
boolean Empty()	**if** $top == 0$ **then return** *true* **else return** *false*
Push(key k)	$top = top + 1$ $A[top] = k$
key Pop()	**Aufgabe:** Schreiben Sie Pseudocode, der das oberste Element vom Stapel nimmt und zurückgibt!
key Top()	

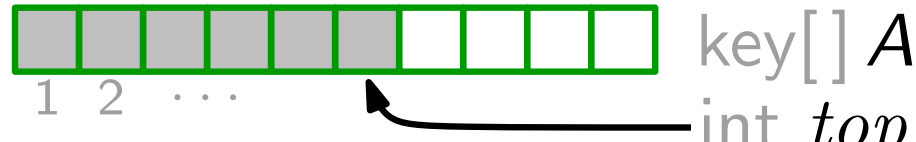
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	Implementierung
	
boolean Empty()	**if** $top == 0$ **then return** *true* **else return** *false*
Push(key k)	$top = top + 1$ $A[top] = k$
key Pop()	$top = top - 1$ **return** $A[top + 1]$
key Top()	

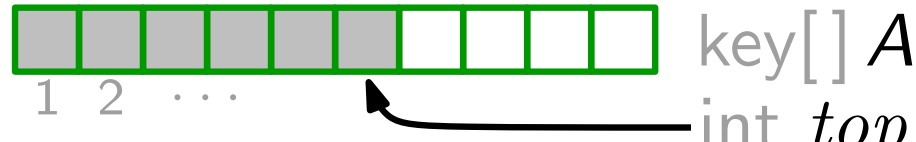
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	Implementierung
	
boolean Empty()	**if** $top == 0$ **then return** *true* **else return** *false*
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()	

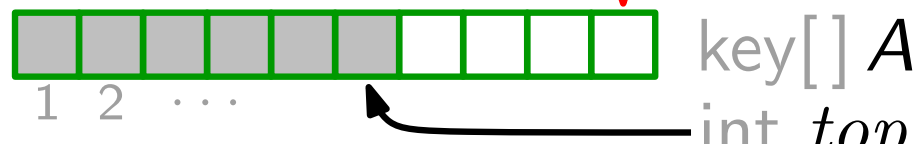
I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

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

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	Implementierung	
Stack(int n)	$A =$ **new*** key$[1..n]$ $top = 0$	key[ ] A int top
boolean Empty()	**if** $top == 0$ **then return** *true* **else return** *false*	
Push(key k)	$top = top + 1$ $A[top] = k$ **if** $top > A.length$ **then** **error** „overflow“	
key Pop()	**if** Empty() **then error** „underflow“ **else** $top = top - 1$ **return** $A[top + 1]$	**Laufzeiten?** Alle* $O(1)$, d.h. konstant.
key Top()	**if** Empty() **then** ... **else return** $A[top]$	

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)	*Konstruktor* — *Attribute*: key[] A, int top
boolean Empty()	*Methoden*
Push(key k)	
key Pop()	
key Top()	

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp	**Implementierung**
Stack(int *n*)	*Konstruktor* — *Attribute*: key[] *A*, int *top*
boolean Empty()	*Methoden*
Push(key *k*)	
key Pop()	
key Top()	

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()

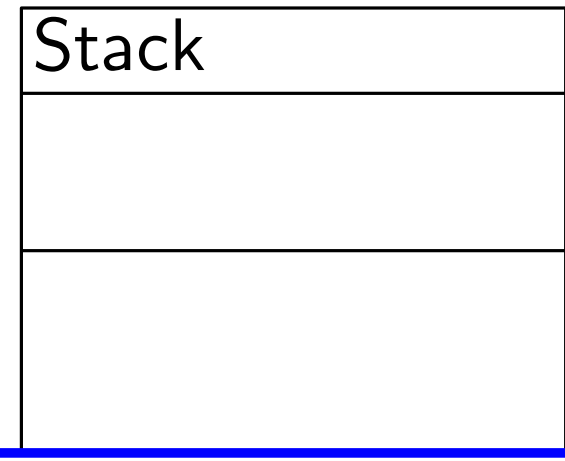
Implementierung

Konstruktor

Attribute: key[] *A*, int *top*

Methoden

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



I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp
Stack(int *n*)
boolean Empty()
Push(key *k*)
key Pop()
key Top()

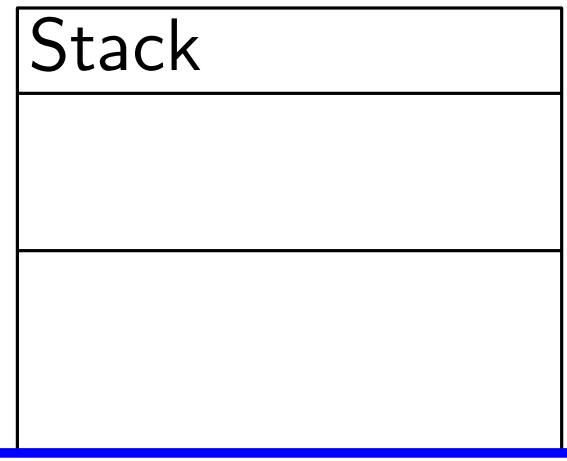
Implementierung

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()

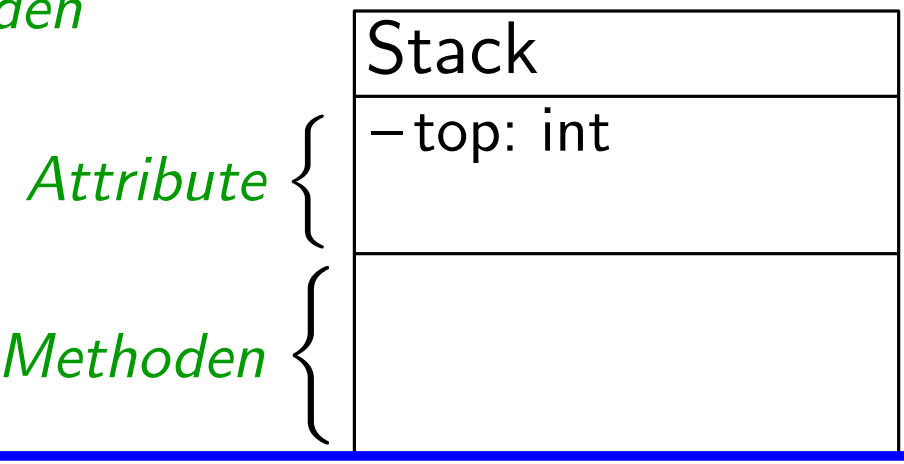
Implementierung

Konstruktor

Attribute: key[] *A*, int *top*

Methoden

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



I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp
Stack(int *n*)
boolean Empty()
Push(key *k*)
key Pop()
key Top()

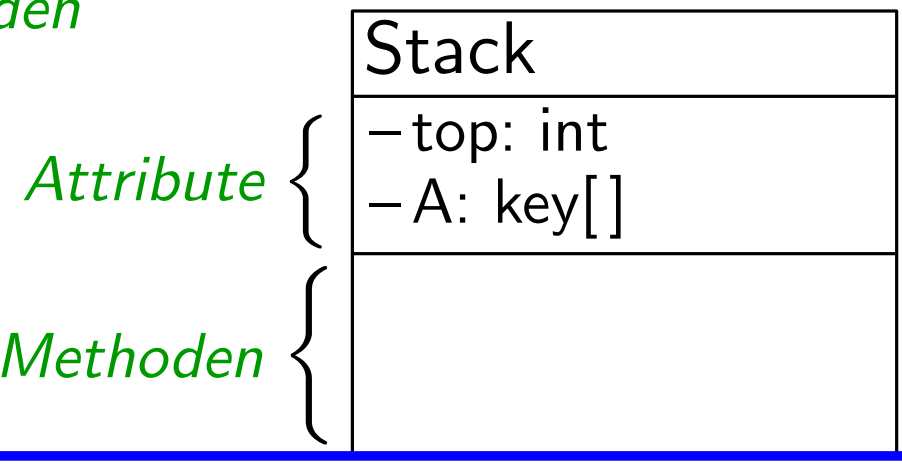
Implementierung

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!

Stack
Attribute: −top: int −A: key[]
Methoden: +Empty(): boolean

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Abstr. Datentyp
Stack(int *n*)
boolean Empty()
Push(key *k*)
key Pop()
key Top()

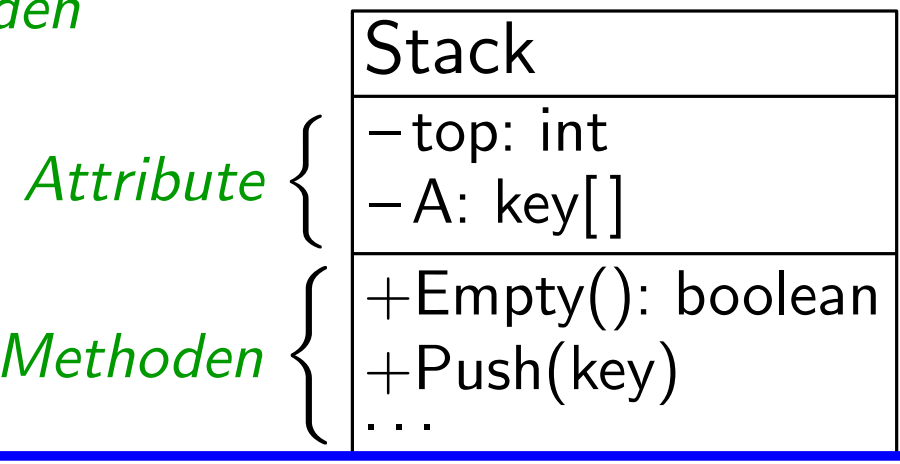
Implementierung

Konstruktor

Attribute: key[] *A*, int *top*

Methoden

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

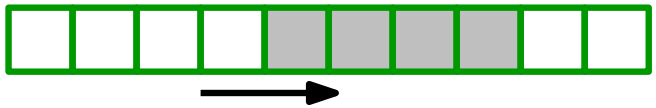


II. Schlange

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung

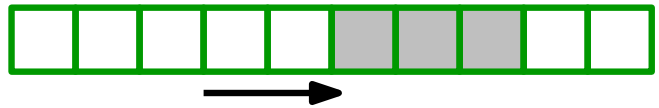
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung

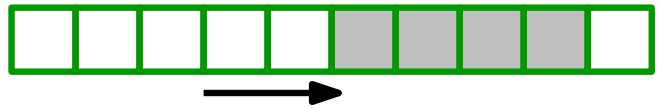
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung

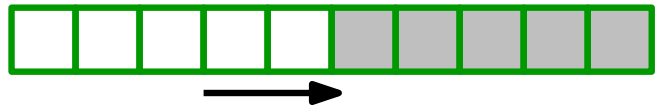
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung

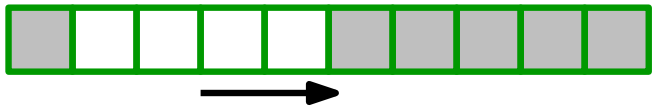
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung

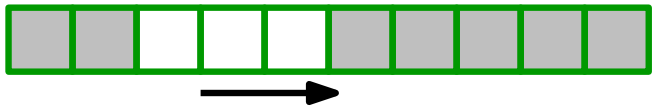
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung

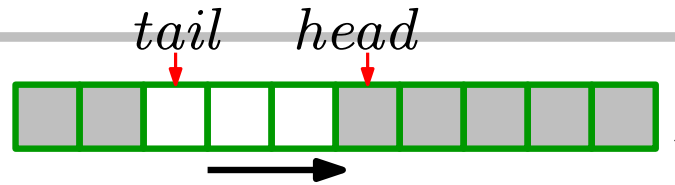
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung

II. Schlange



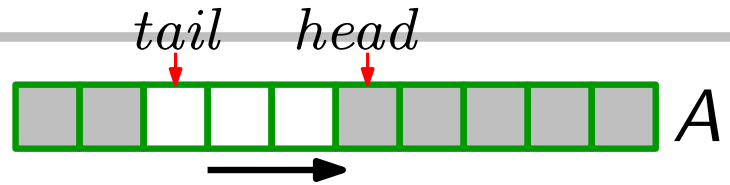
verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp

Implementierung

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

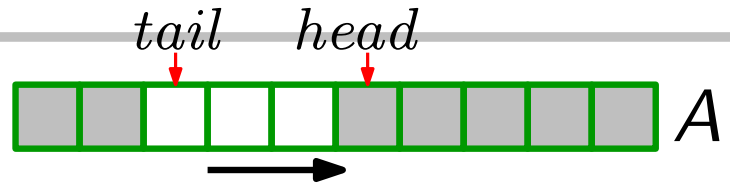
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	**Implementierung**	
Queue(int n)		key[] A int $tail$ int $head$

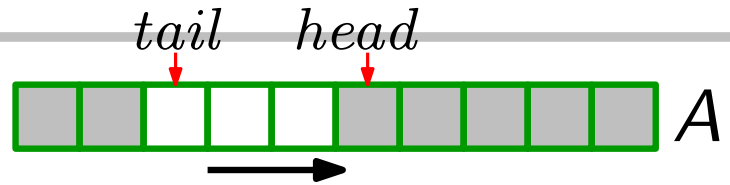
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	**Implementierung**	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[] A int $tail$ int $head$

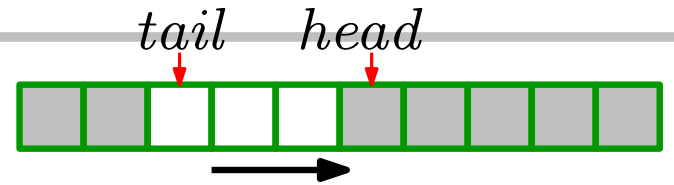
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	**Implementierung**	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[] A int $tail$ int $head$
boolean Empty()		

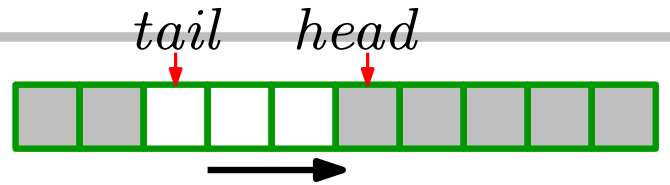
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[ ] A int $tail$ int $head$
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	

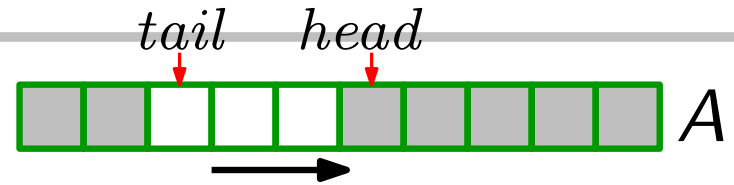
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[] A int $tail$ int $head$
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	
Enqueue(key k) *stell neues Element an den Schwanz der Schlange an*		

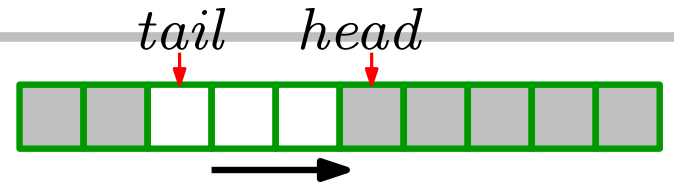
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[] A int $tail$ int $head$
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	
Enqueue(key k) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	

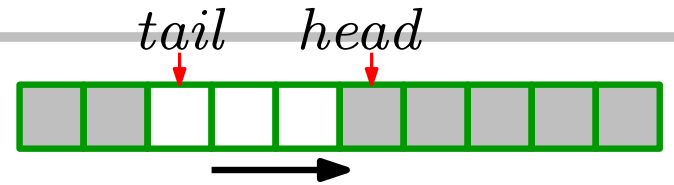
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[] A int $tail$ int $head$
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	
Enqueue(key k) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	
key Dequeue() *entnimm Element am Kopf der Schlange*		

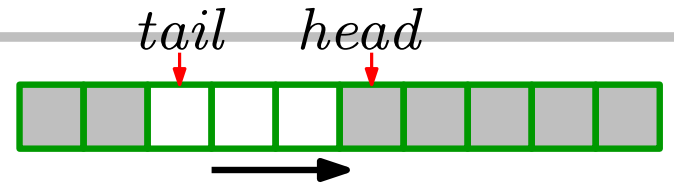
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[] A int $tail$ int $head$
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	
Enqueue(key k) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	
key Dequeue() *entnimm Element am Kopf der Schlange*	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k	

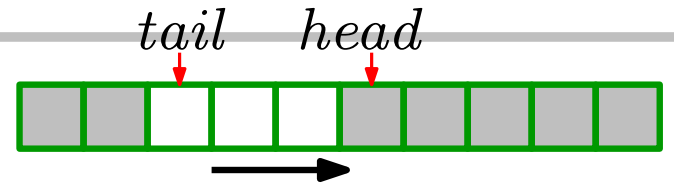
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[] A int $tail$ int $head$
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	
Enqueue(key k) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	
key Dequeue() *entnimm Element am Kopf der Schlange*	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k	

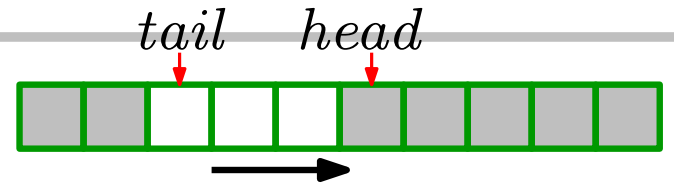
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **new** key[1..n] $tail = head = 1$	key[ ] A int $tail$ int $head$
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	
Enqueue(key k) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	
key Dequeue() *entnimm Element am Kopf der Schlange*	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k	

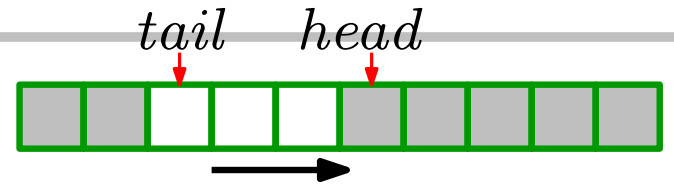
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

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

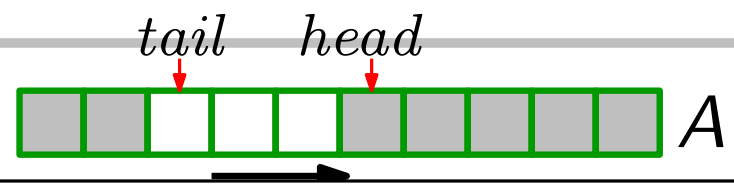
II. Schlange

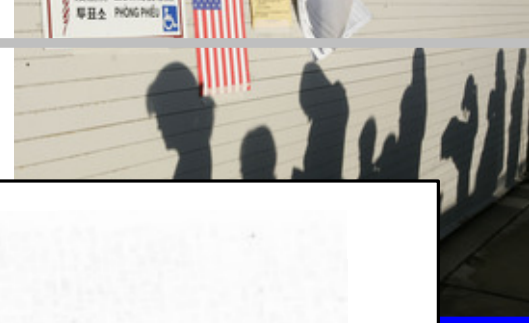


verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **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) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	
key Dequeue() *entnimm Element am Kopf der Schlange*	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k	

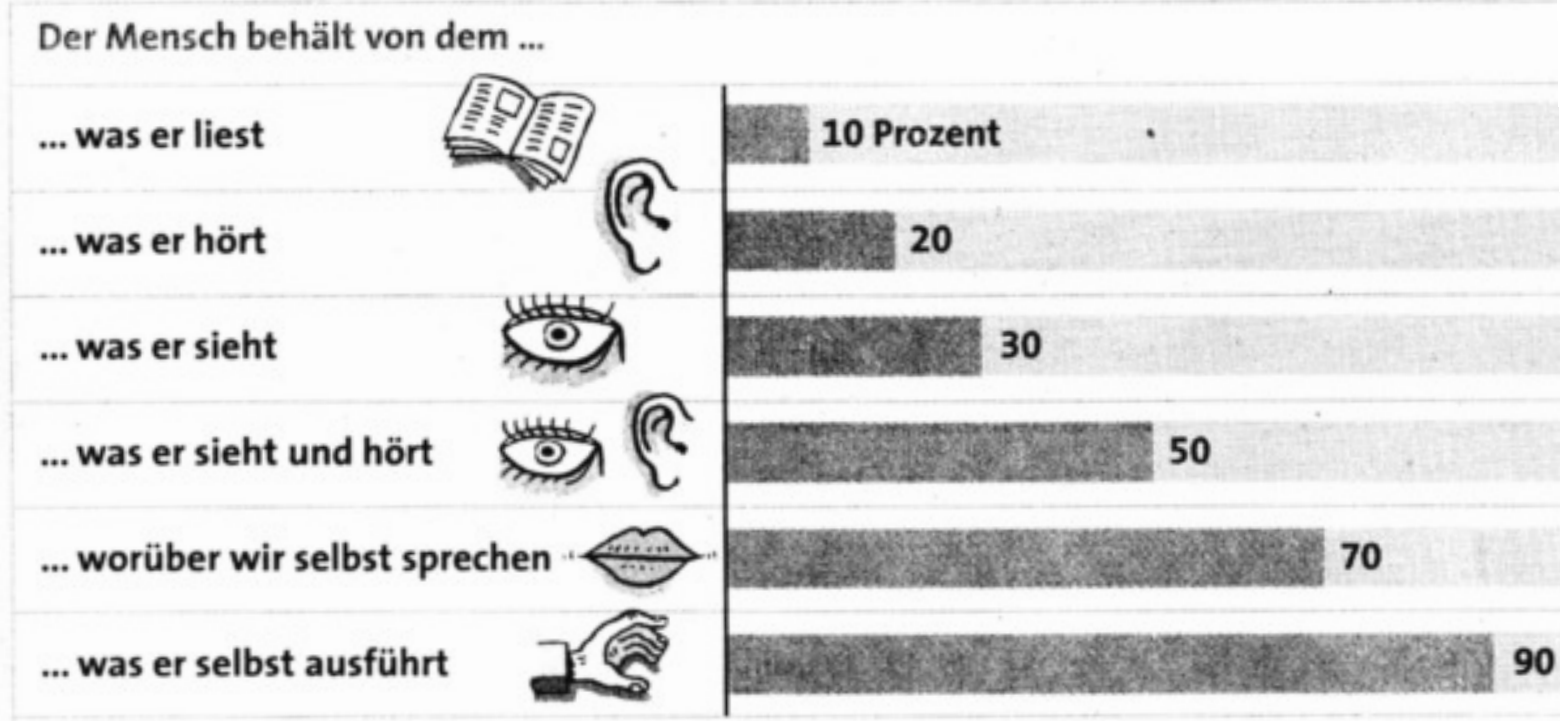
II. Schlange





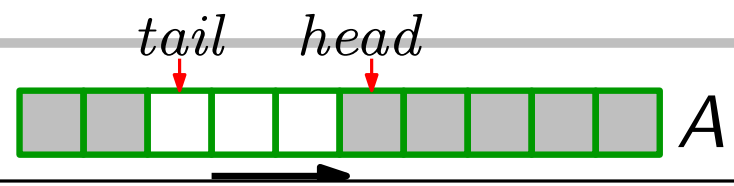
Die sinnliche Spur der Erinnerung

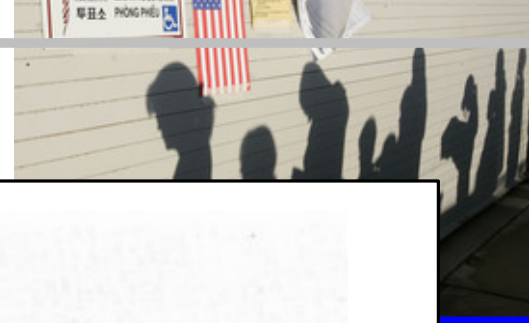
Wer lernen will, muss vor allem reden und be-greifen



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

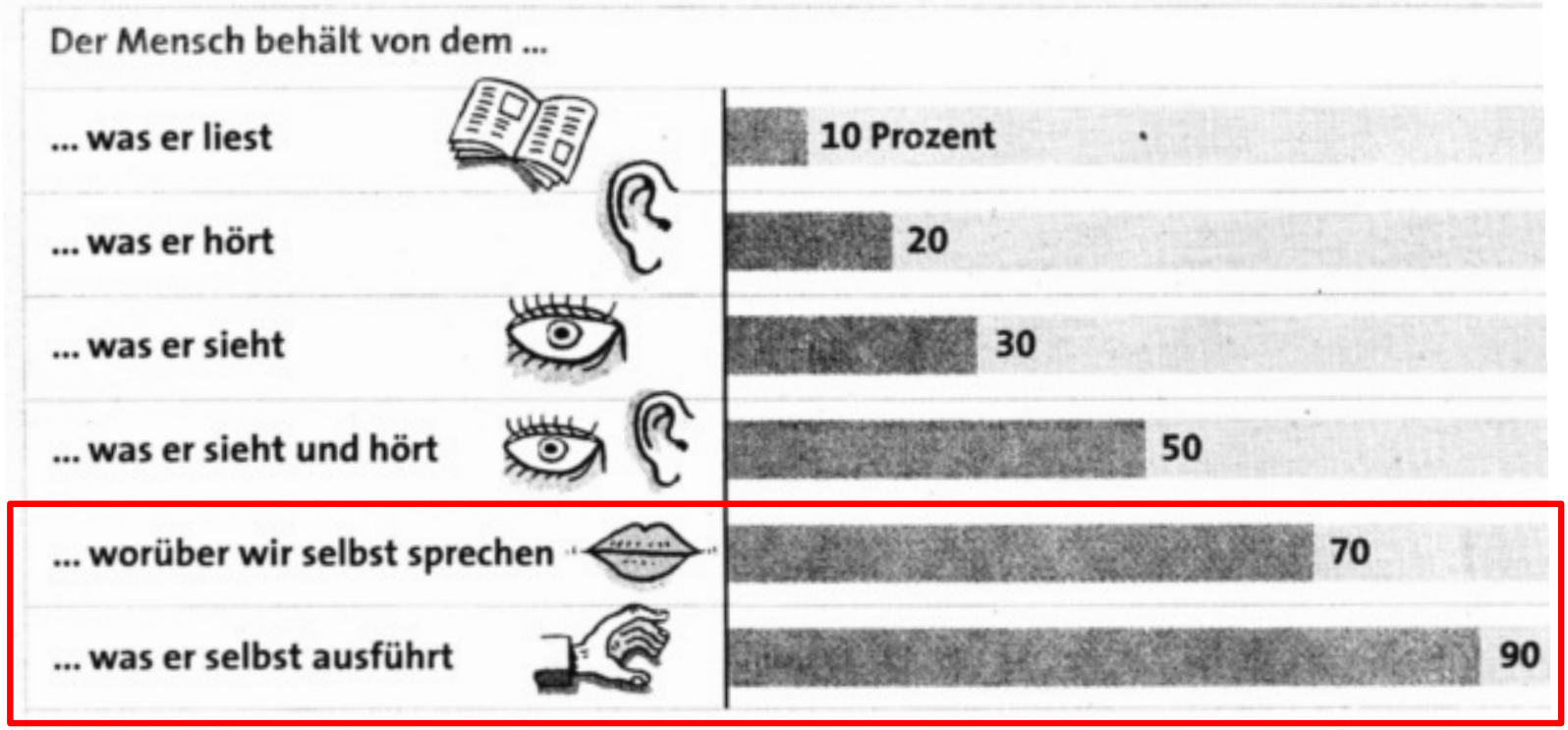
II. Schlange





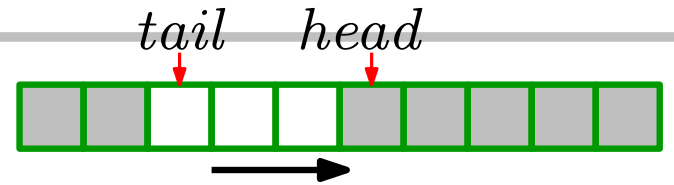
Die sinnliche Spur der Erinnerung

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



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

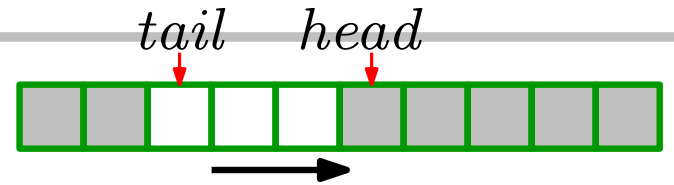
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	Implementierung	
Queue(int n)	$A =$ **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) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	
key Dequeue() *entnimm Element am Kopf der Schlange*	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k	

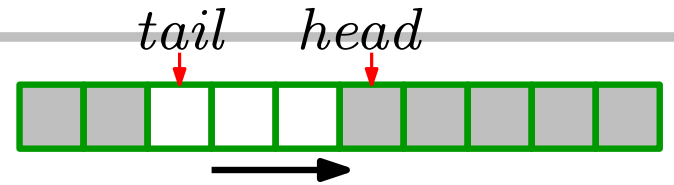
II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Abs. Datentyp	**Implementierung**	
Queue(int n)	$A =$ **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) *stell neues Element an den Schwanz der Schlange an*	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$	
key Dequeue() *entnimm Element am Kopf der Schlange*	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k	**Laufzeiten?**

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

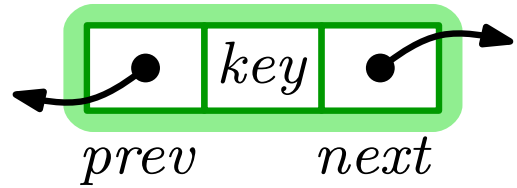
Abs. Datentyp	**Implementierung**	
Queue(int n)	$A =$ **new*** key$[1..n]$ $tail = head = 1$	key[] A int $tail$ int $head$
Aufgabe: Fangen Sie underflow & overflow ab!		
boolean Empty()	**if** $head == tail$ **then return** *true* **else return** *false*	
Enqueue(key k) *stell neues Element an den Schwanz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

Laufzeiten?
Alle* $O(1)$.

III. Liste

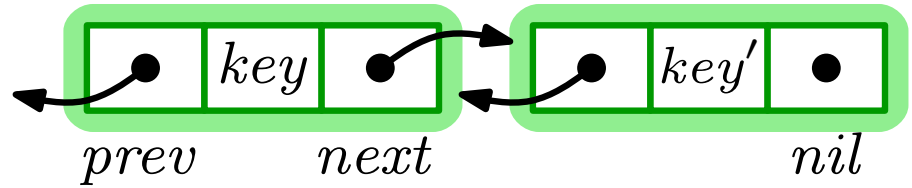
Abs. Datentyp	Implementierung

III. Liste



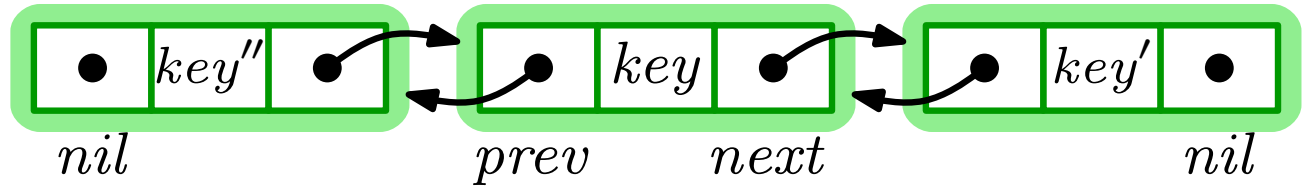
Abs. Datentyp	Implementierung

III. Liste



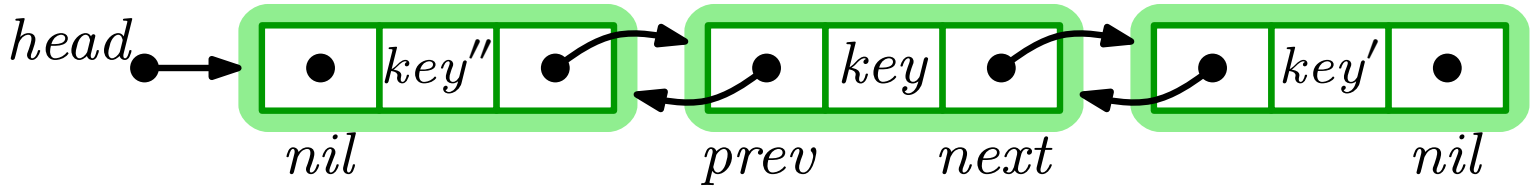
Abs. Datentyp	Implementierung

III. Liste



Abs. Datentyp	Implementierung

III. Liste

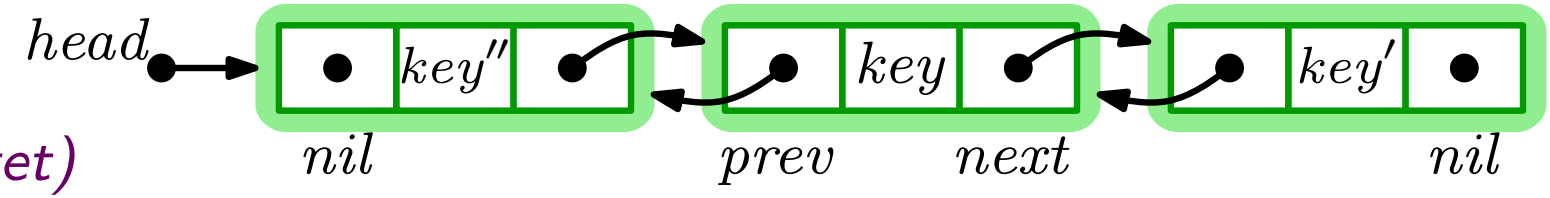


Abs. Datentyp

Implementierung

III. Liste

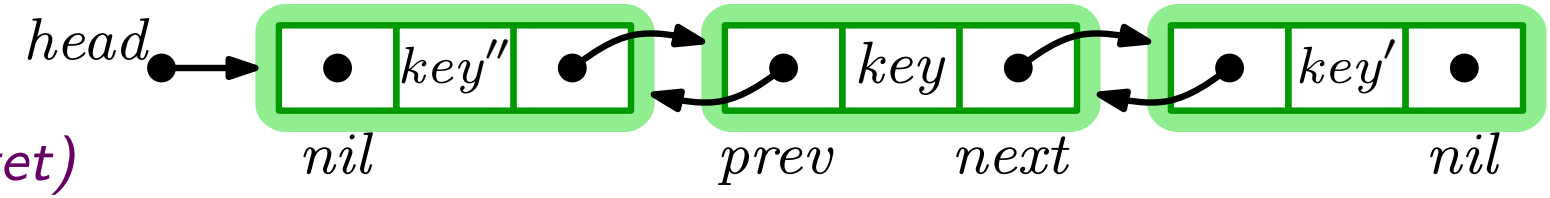
(doppelt verkettet)



Abs. Datentyp	Implementierung

III. Liste

(doppelt verkettet)



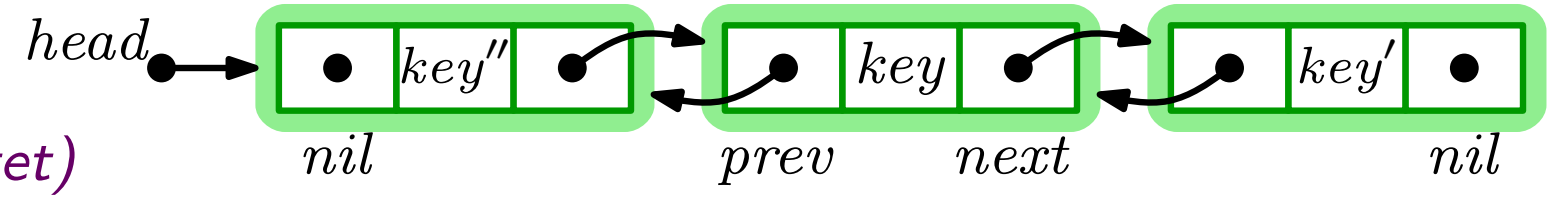
Abs. Datentyp

Implementierung

ptr *head*

III. Liste

(doppelt verkettet)



Abs. Datentyp

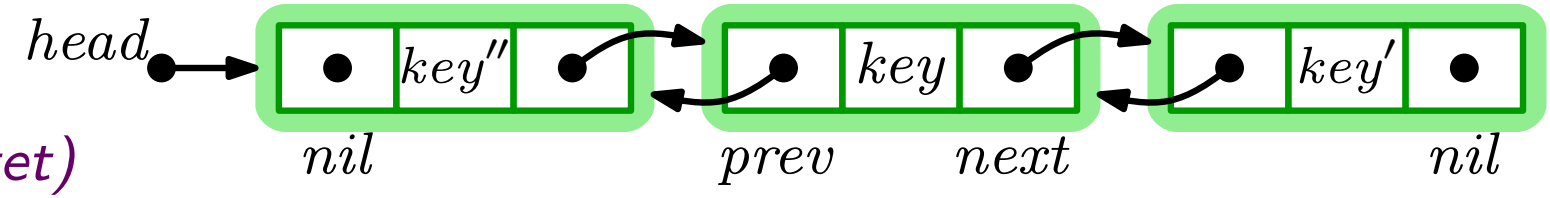
Implementierung

key *key*
ptr *prev*
ptr *next*

ptr *head*

III. Liste

(doppelt verkettet)



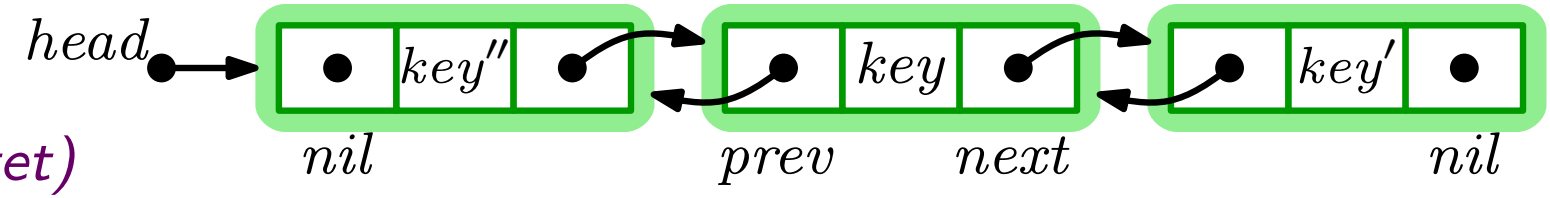
Abs. Datentyp

Implementierung

Item
key *key*
ptr *prev*
ptr *next*
ptr *head*

III. Liste

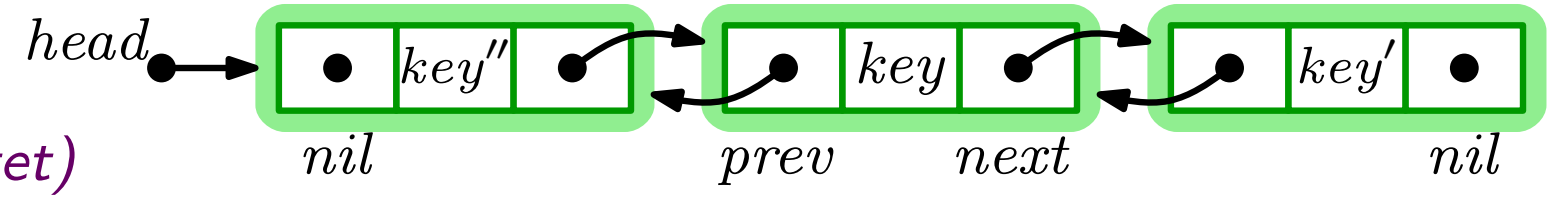
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	Item key *key* ptr *prev* ptr *next* ptr *head*

III. Liste

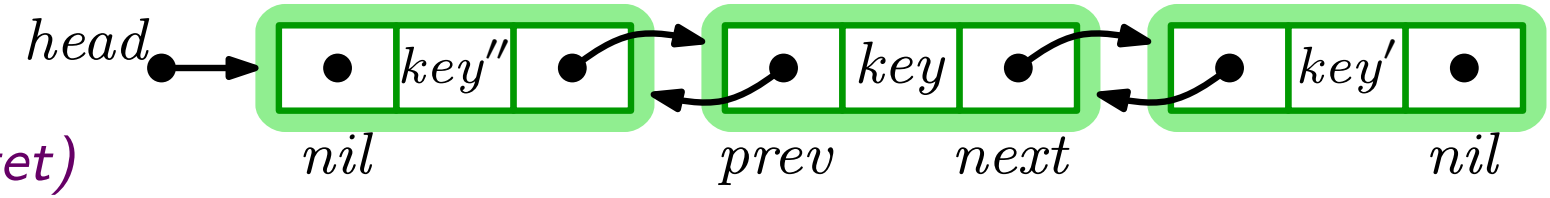
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$
	Item key key ptr $prev$ ptr $next$ ptr $head$

III. Liste

(doppelt verkettet)



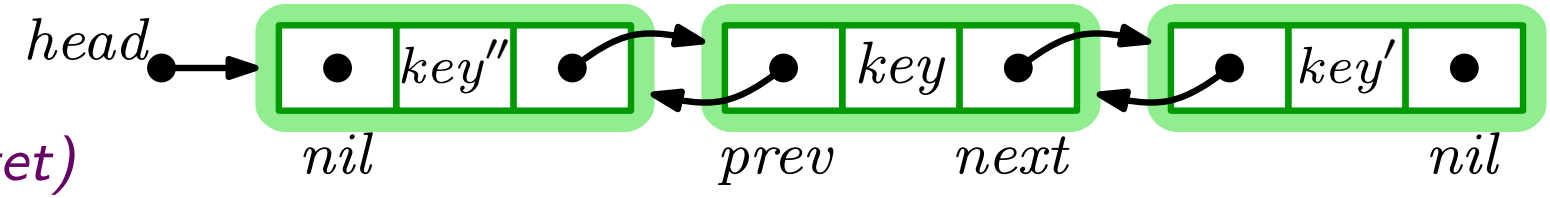
Abs. Datentyp	Implementierung
List()	$head = nil$
ptr Search(key k)	

Item
- key key
- ptr $prev$
- ptr $next$

ptr $head$

III. Liste

(doppelt verkettet)

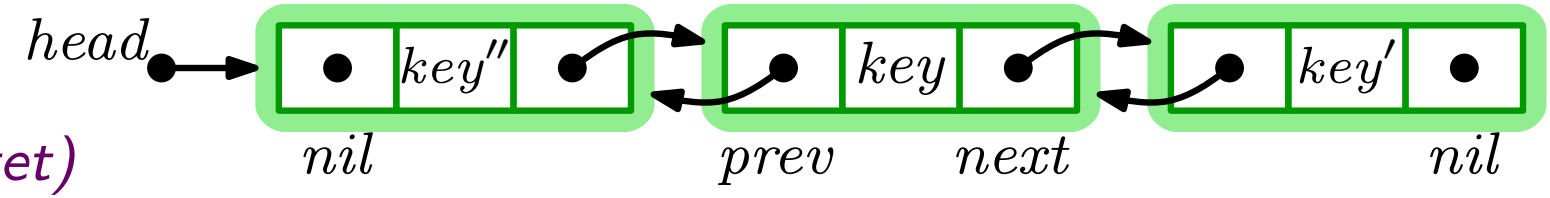


Abs. Datentyp	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$

Item
key key
ptr $prev$
ptr $next$
ptr $head$

III. Liste

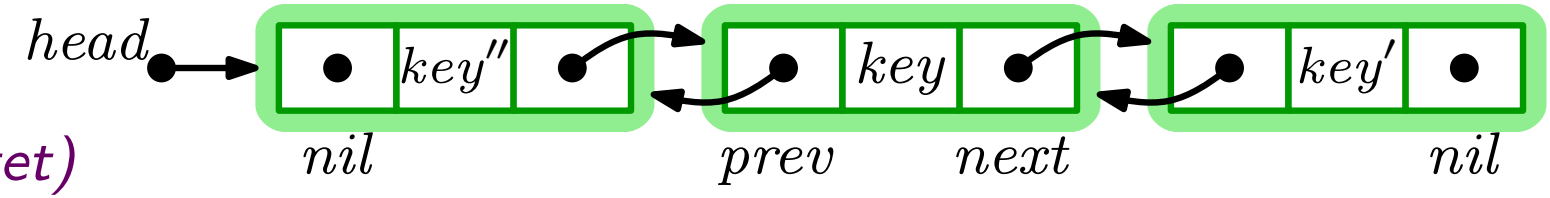
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$

III. Liste

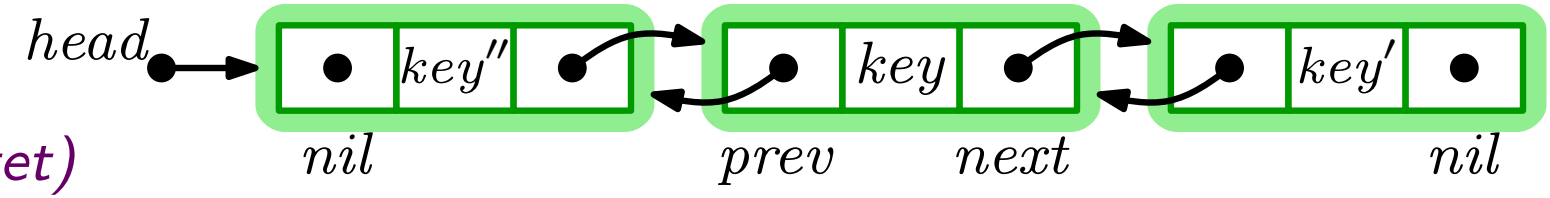
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x

III. Liste

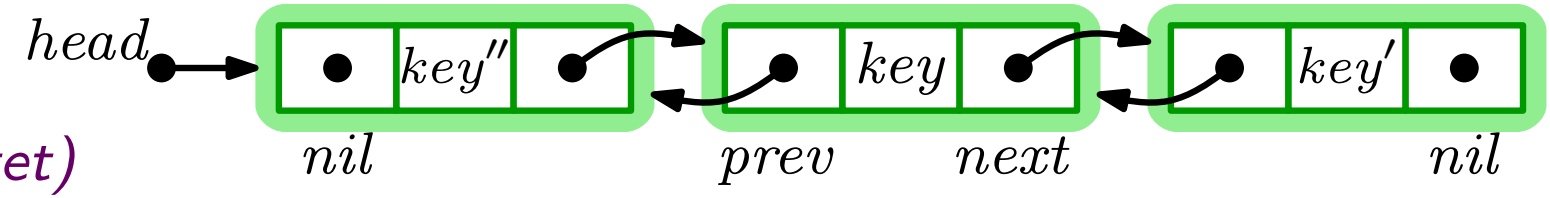
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	

III. Liste

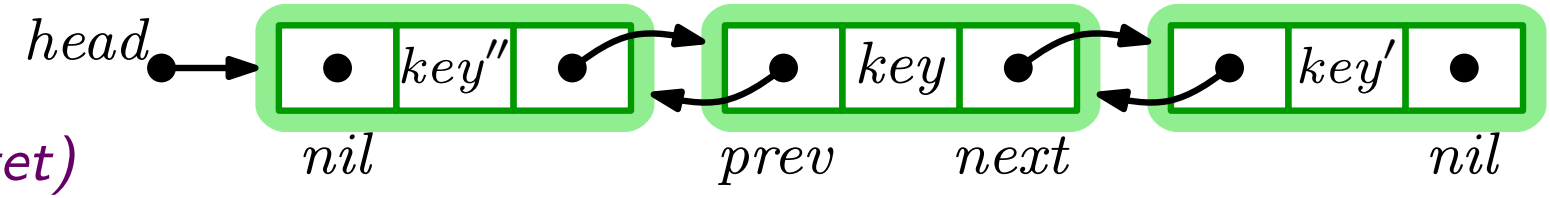
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\llcorner\ x = x.next$ **return** x
ptr Insert(key k)	$x =$ **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

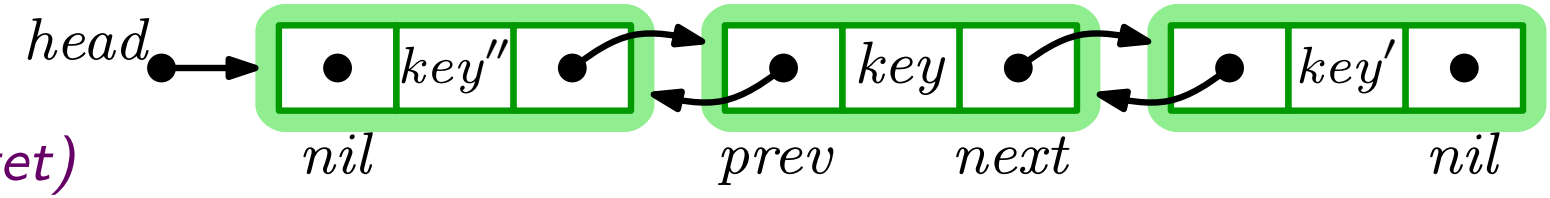
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	$x =$ **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

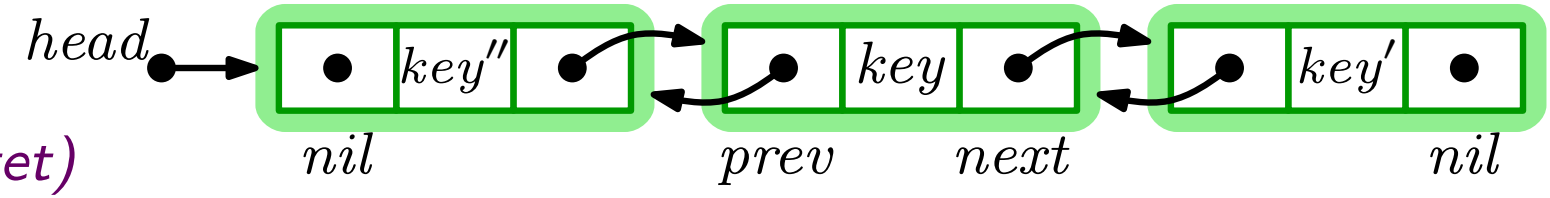
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item(key k, ptr p) Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\llcorner\ x = x.next$ **return** x
ptr Insert(key k)	$x =$ **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

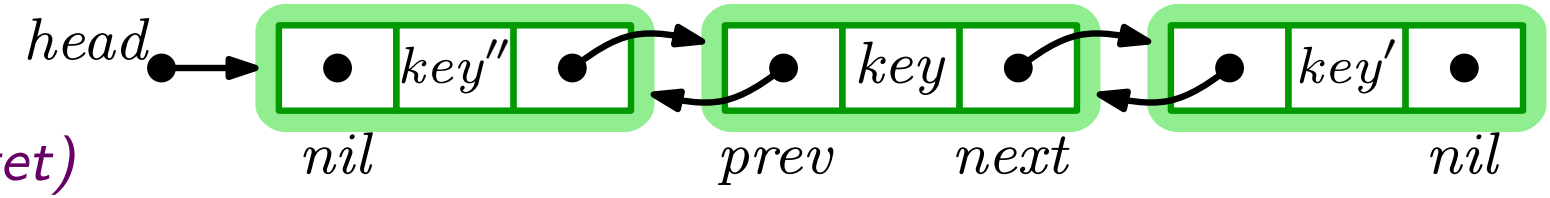
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item(key k, ptr p) $key = k$ $next = p$ $prev = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	$x =$ **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

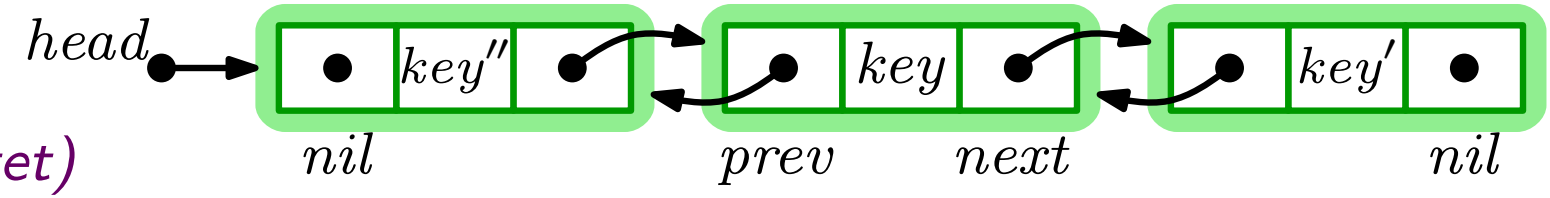
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item(key k, ptr p) $key = k$ $next = p$ $prev = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(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	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) ~~$x.key = k$; $x.prev = nil$; $x.next = head$~~ **if** $head \neq nil$ **then** $head.prev = x$ $head = x$; **return** x

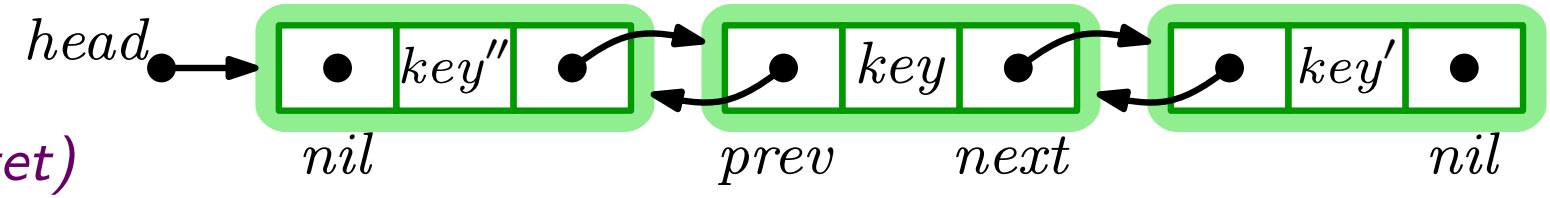
Item(key k, ptr p)
$key = k$
$next = p$
$prev = nil$

Item
key key
ptr $prev$
ptr $next$

ptr $head$

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 $next$

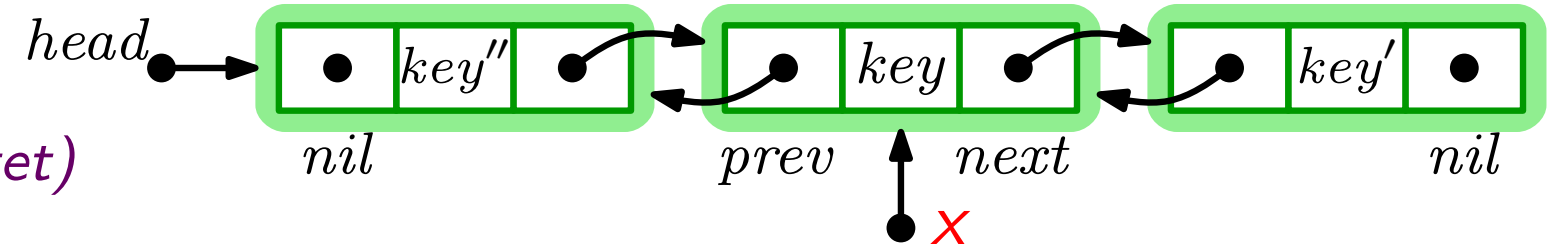
ptr $head$

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

$x =$ **new** Item(~~)~~ 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

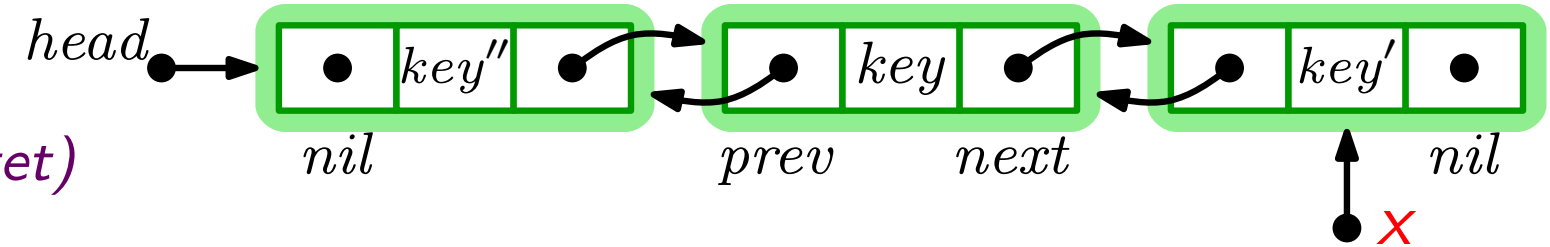
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item(key k, ptr p) $key = k$ $next = p$ $prev = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k) **Aufgabe:** Implementieren Sie Delete(ptr x)	$x =$ **new** Item(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

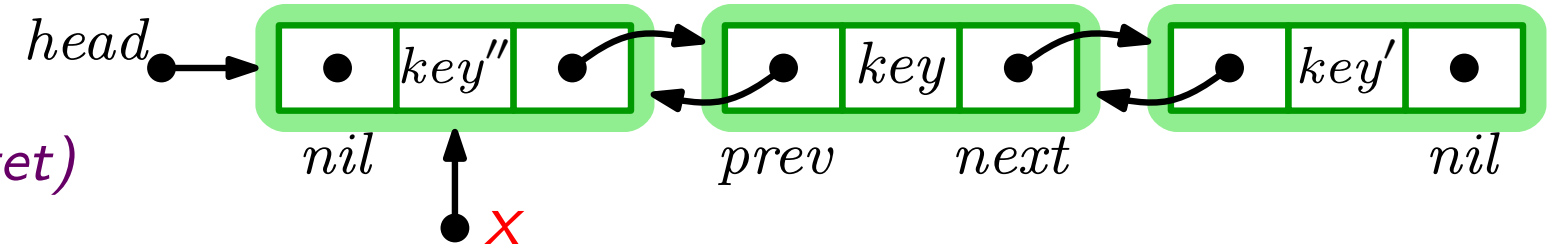
(doppelt verkettet)



Abs. Datentyp	Implementierung
List()	$head = nil$ Item(key k, ptr p) $key = k$ $next = p$ $prev = nil$ Item key key ptr $prev$ ptr $next$ ptr $head$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k) **Aufgabe:** Implementieren Sie Delete(ptr x)	$x =$ **new** Item(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 $next$

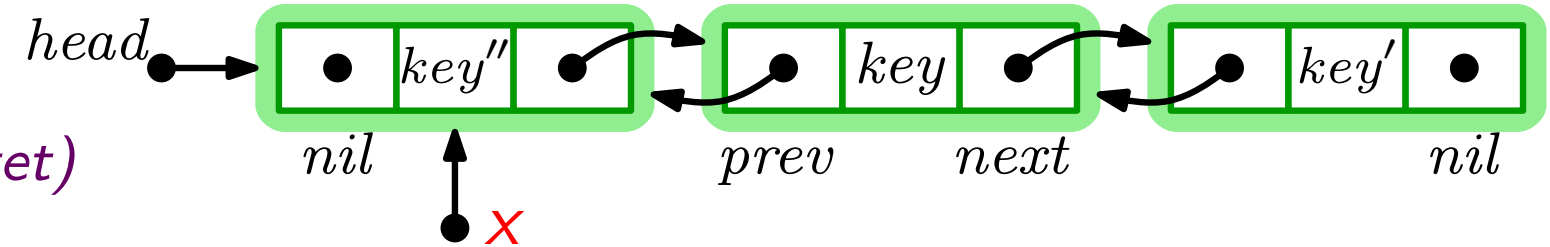
ptr $head$

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

$x =$ **new** Item(~~)~~ 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 $next$

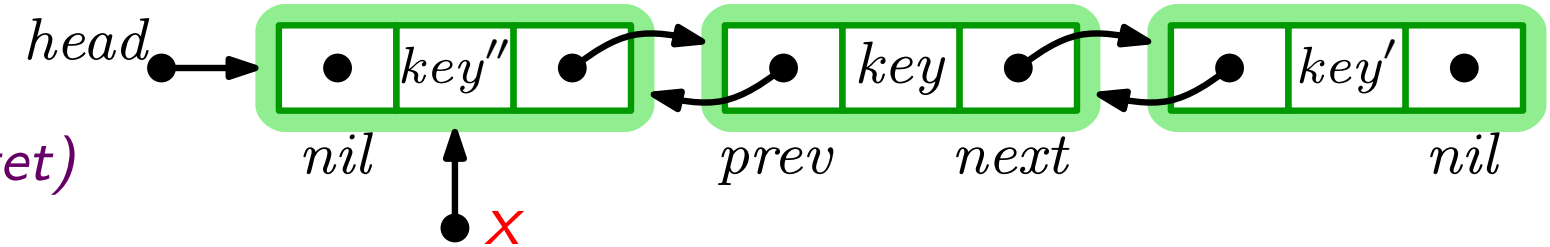
ptr $head$

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

$x =$ **new** Item(~~)~~ 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 $next$

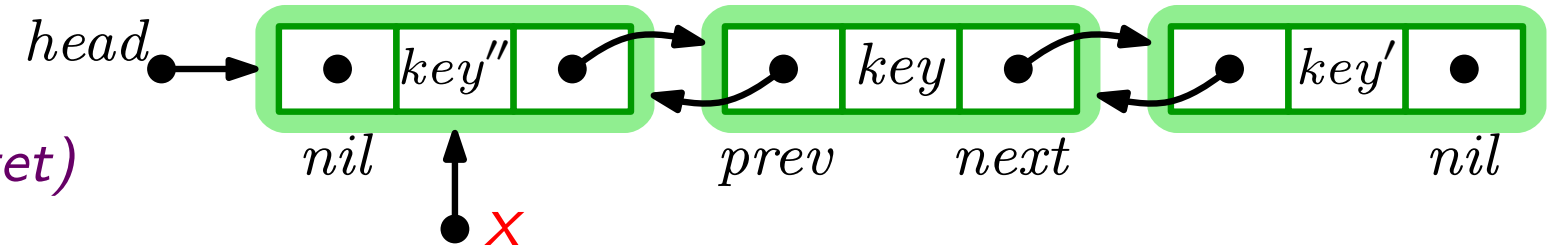
ptr $head$

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

$x =$ **new** Item(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 $next$

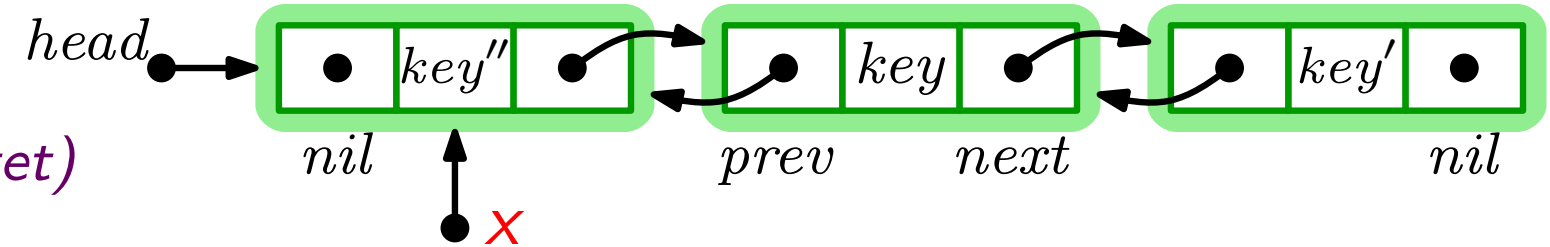
ptr $head$

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

$x =$ **new** Item(~~)~~ 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)

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 $next$

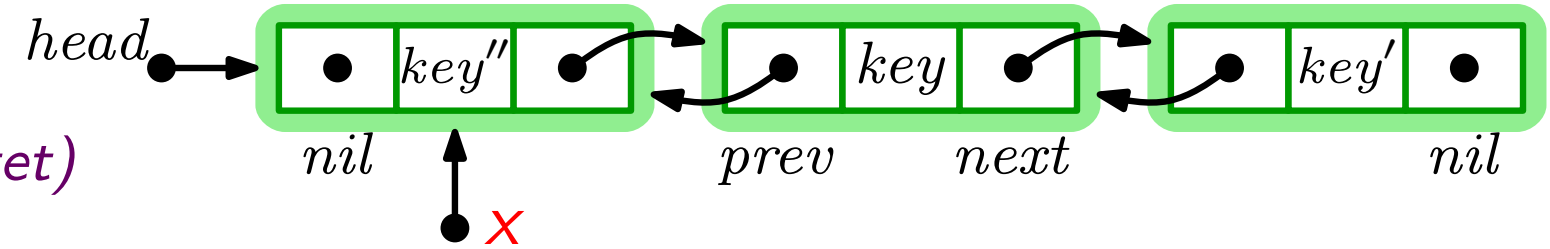
ptr $head$

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

$x =$ **new** Item(~~)~~ 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 key
ptr $prev$
ptr $next$

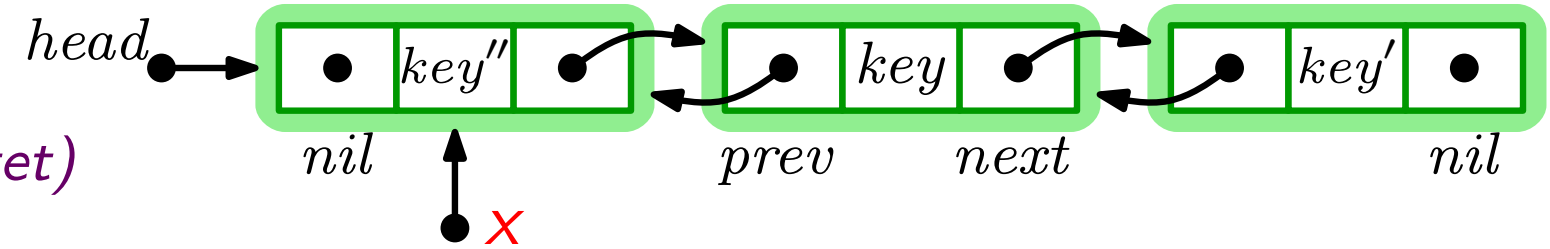
ptr $head$

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

$x =$ **new** Item(~~)~~ 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 $next$

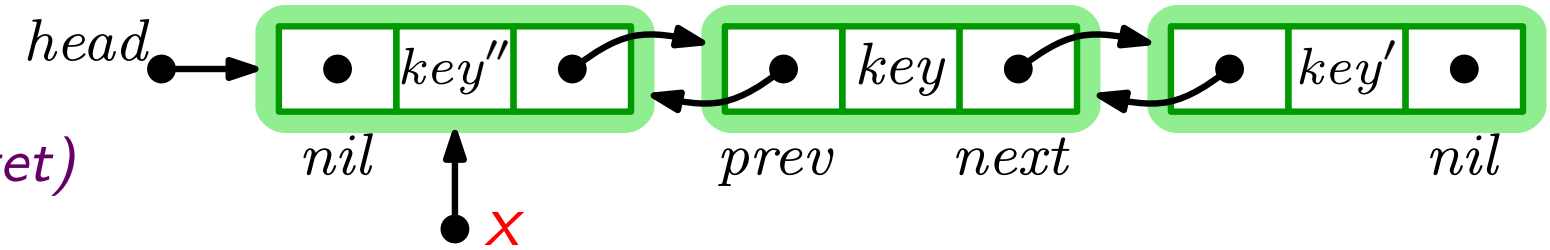
ptr $head$

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

$x =$ **new** Item(~~)~~ 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 $next$

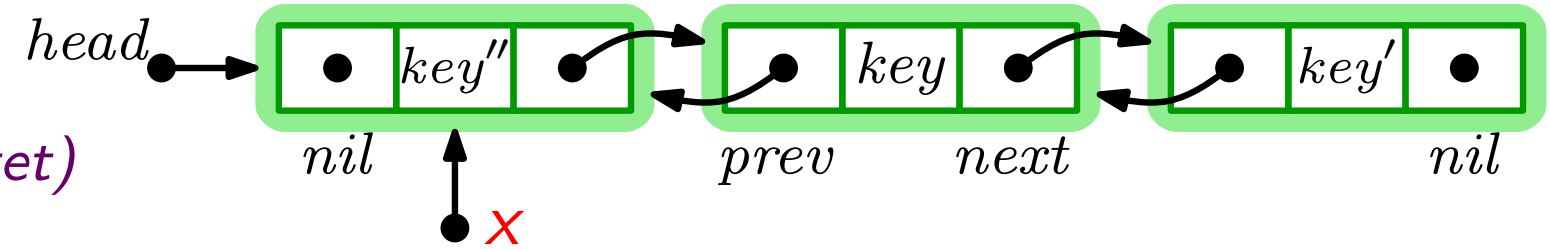
ptr $head$

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

$x =$ **new** Item(~~)~~ 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

$x =$ **new** Item(~~)~~ k, $head$)
~~$x.key = k$; $x.prev = nil$; $x.next = head$~~
if $head \neq nil$ **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
```

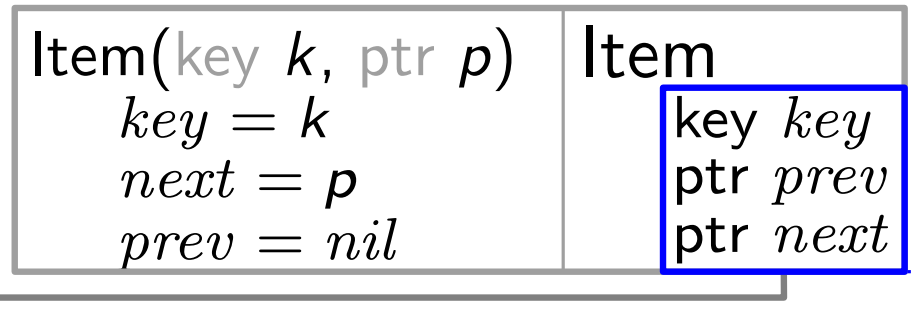
```
Item
    key key
    ptr prev
    ptr next
```

Von Pseudocode zu Javacode: (1) Item

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

Von Pseudocode zu Javacode: (1) Item

```
public class Item {
    private Object key;
    private Item prev;
    private Item 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

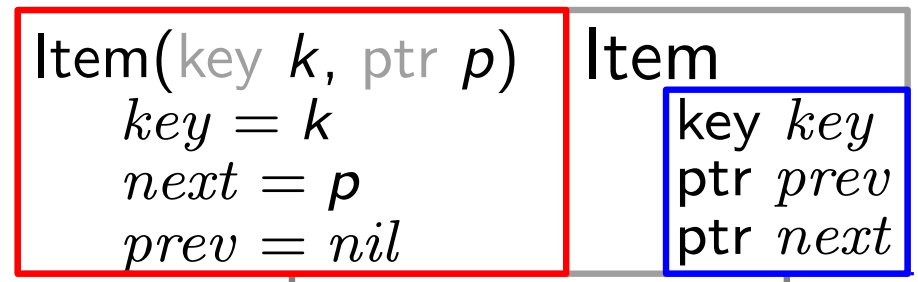
```
public class Item {
    private Object key;
    private Item prev;
    private Item next;

    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; }
}
```



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 $next$

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

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

```
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

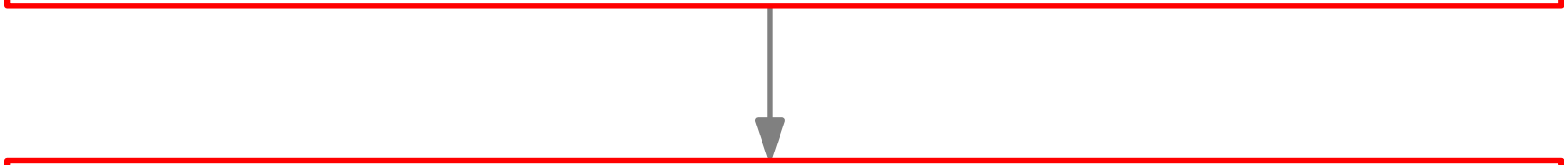
Delete(ptr x)

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;
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

!k.equals(x.getKey())

Listentest.java

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

gleiche Zahlen, aber verschiedene Objekte!

```
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;
}
```

!k.equals(x.getKey())

Listentest.java

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

gleiche Zahlen, aber verschiedene Objekte!

```
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

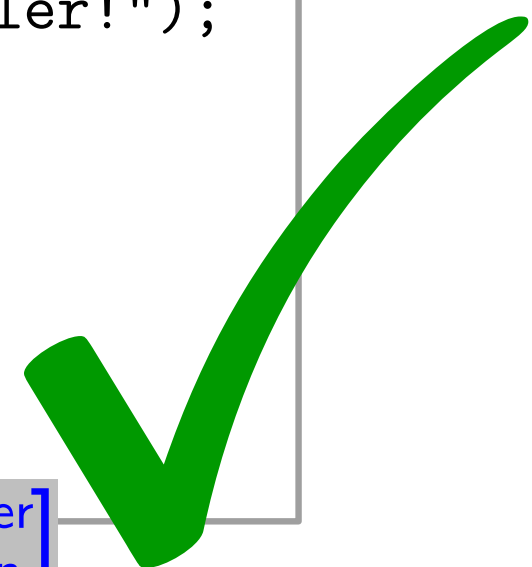
```
public Item search(Object k) {
    Item x = head;
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

!k.equals(x.getKey())

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?