

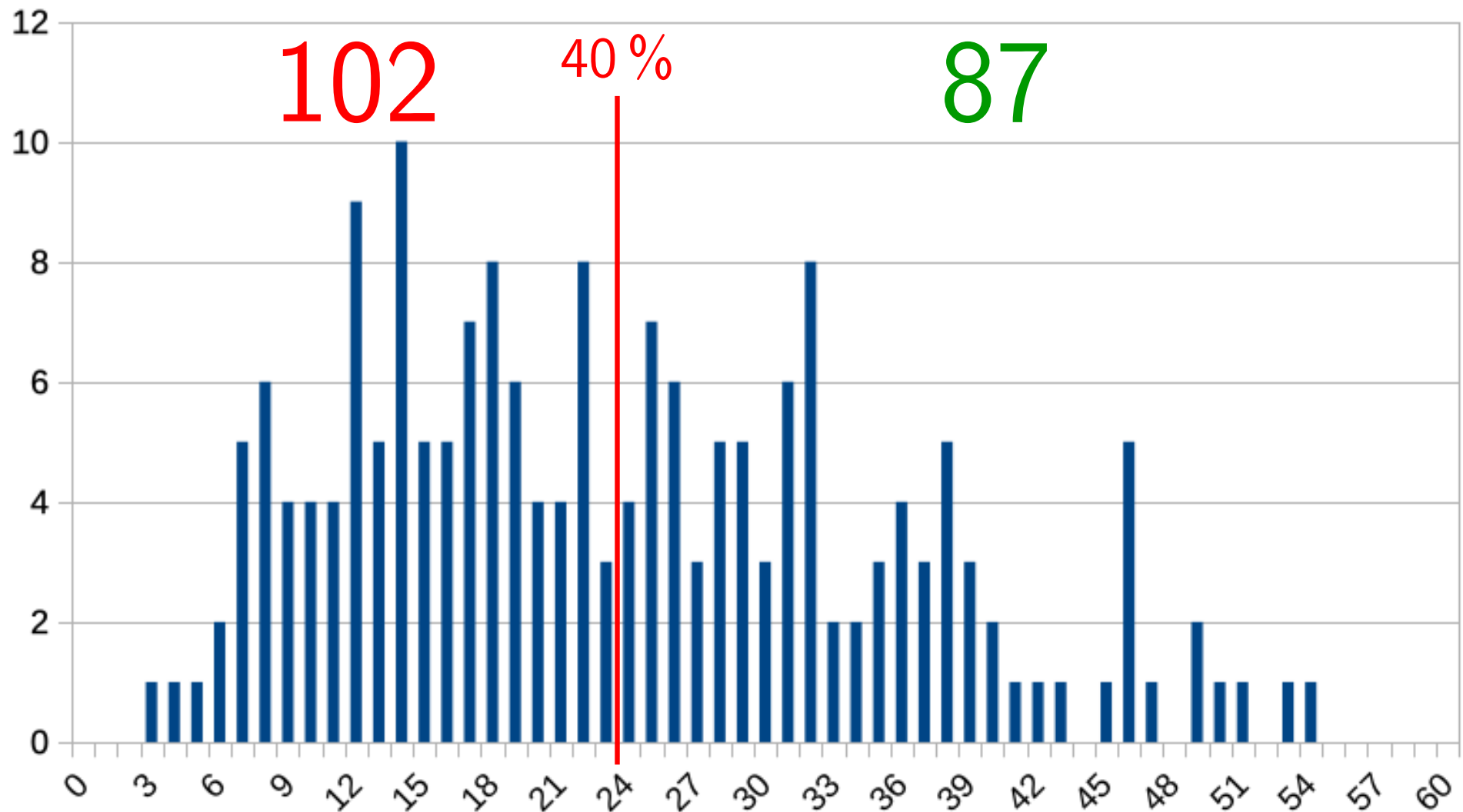
# Algorithmen und Datenstrukturen

Wintersemester 2021/22

11. Vorlesung

Elementare Datenstrukturen:  
Stapel + Schlange + Liste

# 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.

## 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.

### 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

$O(\log n)$

$O(1)$



### 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	Funktionalität
<code>ptr Insert(key <math>k</math>, info <math>i</math>)</code>	<ul style="list-style-type: none"><li>• lege neuen Datensatz <math>(k, i)</math> an</li><li>• <math>M = M \cup \{(k, i)\}</math></li><li>• gib Zeiger auf <math>(k, i)</math> zurück</li></ul>
	<p style="text-align: right;"><math>M</math></p> <p>The diagram illustrates a dynamic set <math>M</math>. 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 (<math>key_1, key_2, \dots, key_n</math>) and the bottom section contains corresponding information (<math>info_1, info_2, \dots, info_n</math>). Ellipses (<math>\dots</math>) are used between the boxes to indicate that there can be more than two elements in the set.</p>

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

Abstrakter Datentyp	Funktionalität
<code>ptr Insert(key <math>k</math>, info <math>i</math>)</code> <code>Delete(ptr <math>x</math>)</code>	<ul style="list-style-type: none"><li>• <math>M = M \setminus \{(x.key, x.info)\}</math></li></ul> <div><p>The diagram shows a yellow oval labeled <math>M</math> containing a table of key-value pairs. The table has two rows and four columns. The first column contains <math>key_1</math> and <math>info_1</math>. The second column contains <math>key_2</math> and <math>info_2</math>. The third column contains three dots <math>\dots</math> in both rows. The fourth column contains <math>key_n</math> and <math>info_n</math>. Each cell is enclosed in a green rectangular border.</p></div>

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

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



# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

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

# Teil III [CLRS]



## Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge  $M$

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

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

# I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



## Abstr. Datentyp

boolean Empty()

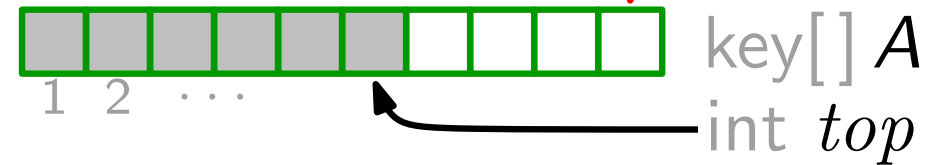
Push(key  $k$ )

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

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<sup>\*</sup>  $O(1)$ ,  
 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

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

}
{
*Methoden*

### Aufgabe:

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

*Attribute*

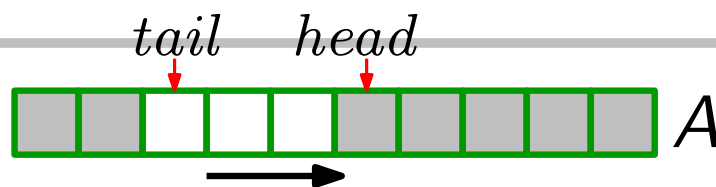
*Methoden*

Stack

- top: int  
 - A: key[]

+ Empty(): boolean  
 + Push(key)  
 ...

## II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*

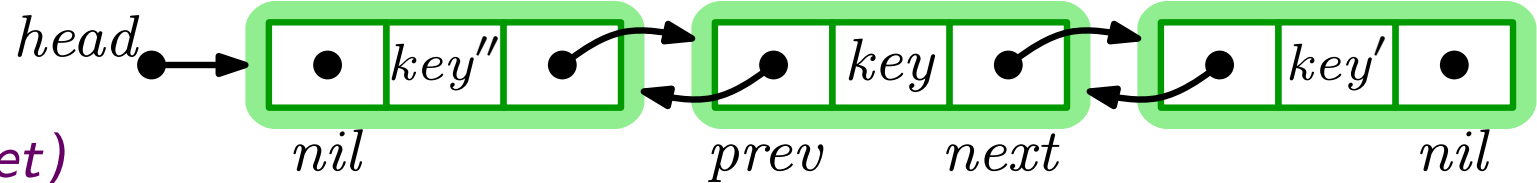


Abs. Datentyp	Implementierung	
Queue(int $n$ )	$A = \text{new}^* \text{key}[1..n]$ $tail = head = 1$	$\text{key}[]$ A $\text{int } tail$ $\text{int } head$
<b>Aufgabe:</b> Fangen Sie underflow & overflow ab!		
boolean Empty()	<b>if</b> $head == tail$ <b>then return true</b> <b>else return false</b>	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

**Laufzeiten?**  
 Alle<sup>\*</sup>  $O(1)$ .

# 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()

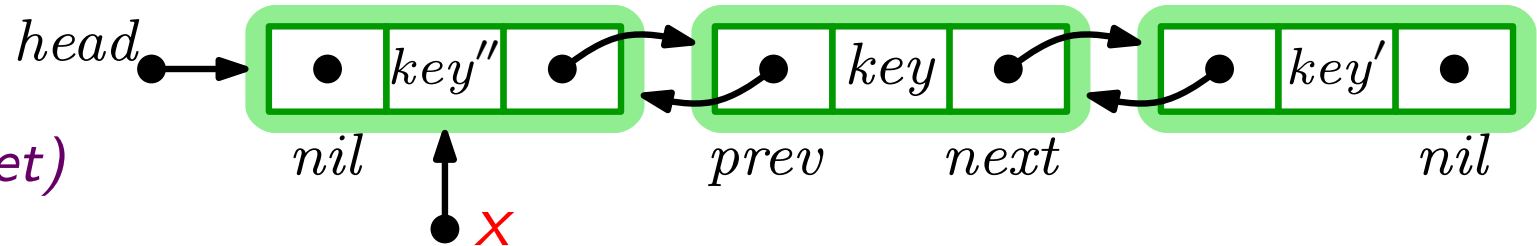
$x.key = k$ ;  $x.prev = nil$ ;  $x.next = head$

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

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

# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()

ptr Search(key  $k$ )

**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**  $x$

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

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

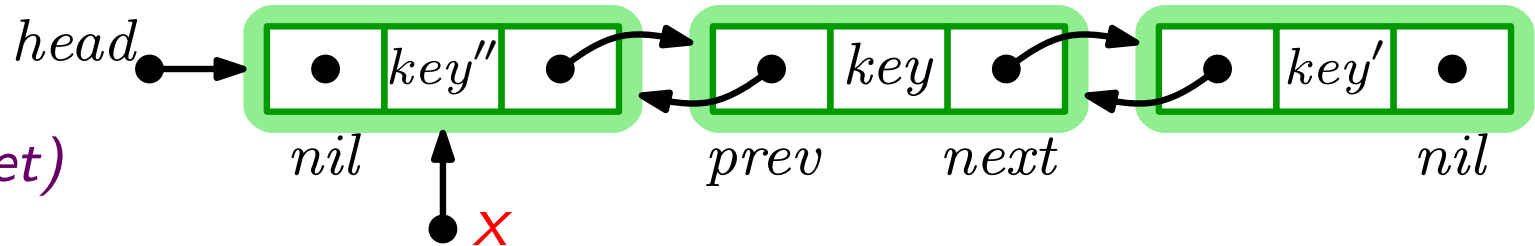
**if**  $head \neq nil$  **then**  $head.prev = x$

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



# III. Liste

(doppelt verkettet)



## Abs. Datentyp

List()  $O(1)$

## 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  $x$ 

```

```

 $x = \text{new Item}(k, head)$ 
 $x.key = k, x.prev = nil; x.next = head$ 
if  $head \neq nil$  then  $head.prev = x$ 
 $head = x$ ; return  $x$ 

```

# Von Pseudocode zu Javacode: (1) Item

```
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

```
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
```



```
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**  $x.next.prev = x.prev$



```
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) {  
        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);  
    }  
}
```

```
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;  
}
```

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

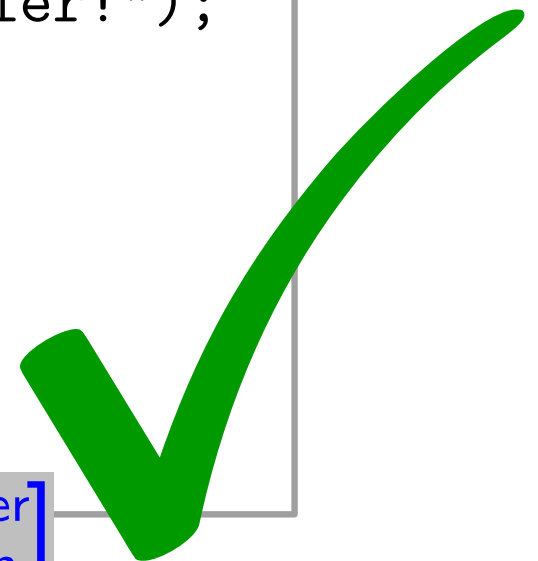
```
public Item search(Object k) {
    Item x = head;    !k.equals(x.getKey())
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

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

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

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