

# **Praktische Informatik 1**

## **Listen, Stapel, Warteschlangen**

Thomas Röfer

- Abstrakte Datentypen
- Reihung
- Stapel
- Warteschlange
- Einfach und doppelt verkettete Liste
- Musterlösung, Übungsblatt

# Einfach verkettete Liste mit Index

- Eigenschaften
  - Bildet ein Array dynamischer Größe mithilfe einer Liste nach
  - Einfügen und Löschen ohne Umkopieren (in Java nicht so relevant)
- IndexList
  - Liste erzeugen
    - $create : \rightarrow IndexList$
  - Eintrag schreiben
    - $set : IndexList \times int \times Entry \rightarrow IndexList$
  - Eintrag auslesen
    - $get : IndexList \times int \rightarrow Entry$
  - Einfügen
    - $insert : IndexList \times int \times Entry \rightarrow IndexList$
  - Löschen
    - $remove : IndexList \times int \rightarrow IndexList$
  - Auf Leere testen
    - $empty : IndexList \rightarrow boolean$

# Einfach verkettete Liste mit Index

```
class IndexList {
    ListElement first;

    IndexList() {
        first = null;
    }

    boolean empty() {
        return first == null;
    }

    ListElement at(int pos) {
        ListElement current = first;
        while (pos > 0 && current != null) {
            current = current.next;
            --pos;
        }
        return current;
    }
}
```

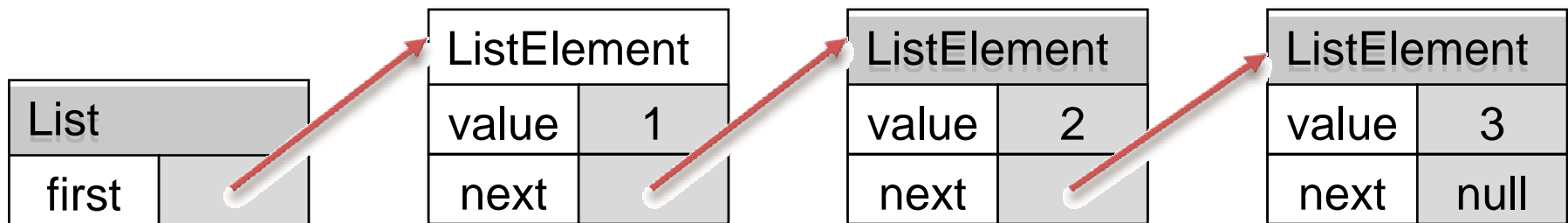
```
int get(int pos) {
    return at(pos).entry;
}

void set(int pos, int entry) {
    at(pos).entry = entry;
}

void remove(int pos) {
    if (pos == 0) {
        first = first.next;
    } else {
        ListElement current = at(pos - 1);
        current.next = current.next.next;
    }
}
```

## Einfach verkettete Liste

- Jedes Element enthält einen Zeiger auf seinen Nachfolger
- Kann nur in einer Richtung durchlaufen werden
- Einfügen immer am Anfang der Liste oder hinter einem bekannten Listenelement
- Ein Element kann nur gelöscht werden, wenn sein Vorgänger bekannt ist



# Einfach verkettete Liste mit Index

```
class IndexList {
    ListElement first;

    IndexList() {
        first = null;
    }

    boolean empty() {
        return first == null;
    }

    ListElement at(int pos) {
        ListElement current = first;
        while (pos > 0 && current != null) {
            current = current.next;
            --pos;
        }
        return current;
    }
}
```

```
int get(int pos) {
    return at(pos).entry;
}

void set(int pos, int entry) {
    at(pos).entry = entry;
}

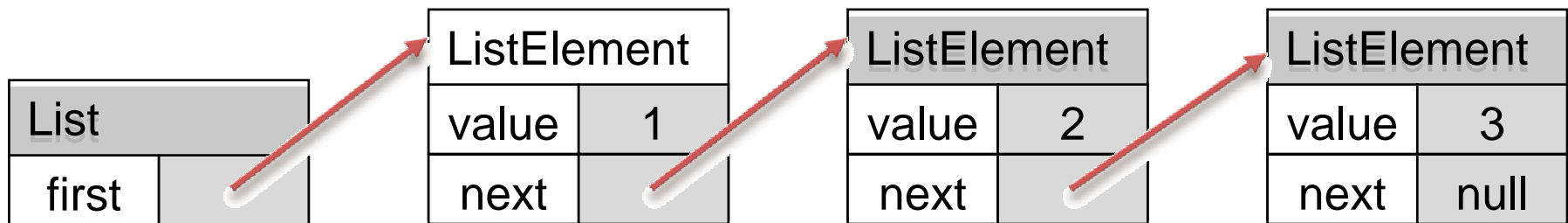
void remove(int pos) {
    if (pos == 0) {
        first = first.next;
    } else {
        ListElement current = at(pos - 1);
        current.next = current.next.next;
    }
}
```

# Einfach verkettete Liste mit Index

```
void insertBefore(int pos, int entry) {  
    if (pos == 0) {  
        first = new ListElement(entry, first);  
    } else {  
        ListElement current = at(pos - 1);  
        current.next = new ListElement(entry, current.next);  
    }  
}  
  
void insertAfter(int pos, int entry) {  
    insertBefore(pos + 1, entry);  
}  
}
```

## Einfach verkettete Liste

- Jedes Element enthält einen Zeiger auf seinen Nachfolger
- Kann nur in einer Richtung durchlaufen werden
- Einfügen immer am Anfang der Liste oder hinter einem bekannten Listenelement
- Ein Element kann nur gelöscht werden, wenn sein Vorgänger bekannt ist





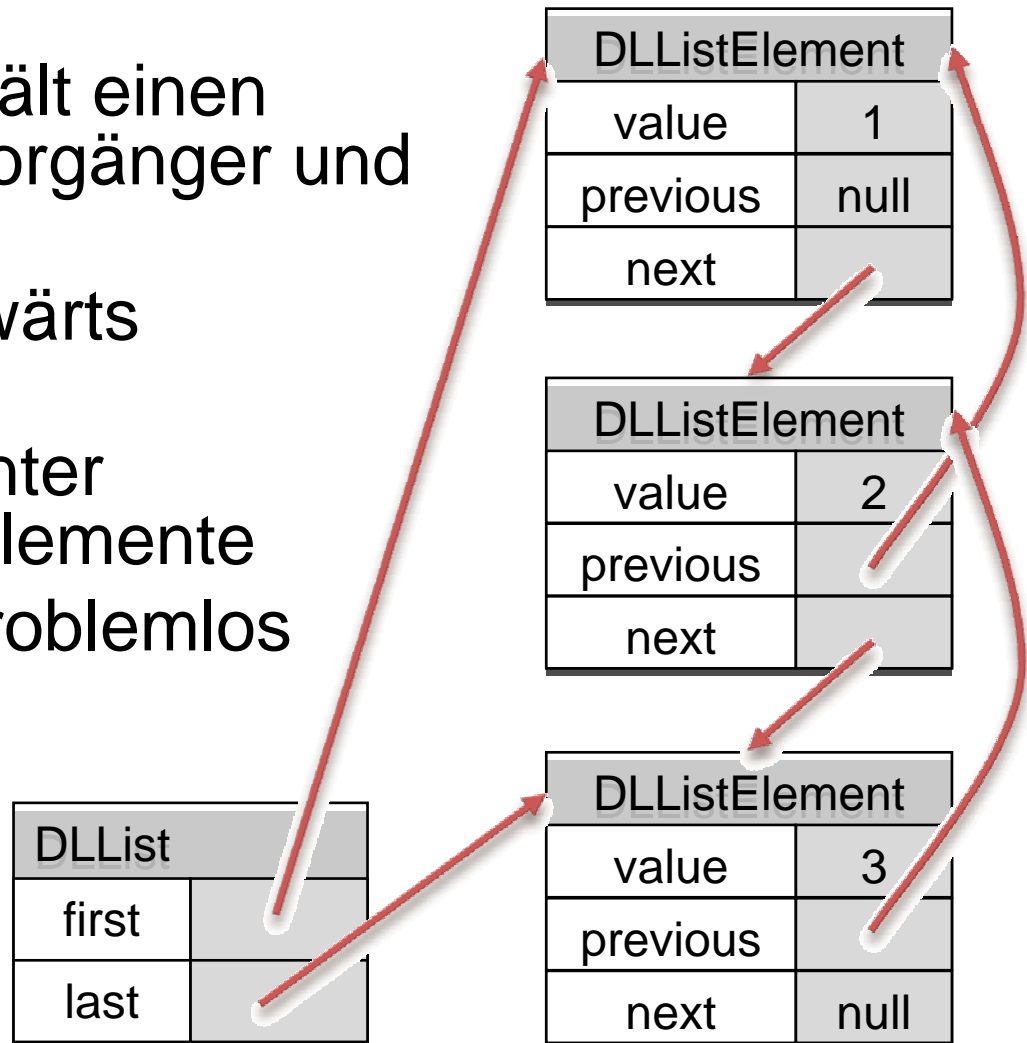
# Einfach verkettete Liste mit Index

```
void insertBefore(int pos, int entry) {  
    if (pos == 0) {  
        first = new ListElement(entry, first);  
    } else {  
        ListElement current = at(pos - 1);  
        current.next = new ListElement(entry, current.next);  
    }  
}  
  
void insertAfter(int pos, int entry) {  
    insertBefore(pos + 1, entry);  
}  
}
```



# Doppelt verkettete Liste

- Jedes Element enthält einen Zeiger auf seinen Vorgänger und seinen Nachfolger
- Kann vor- und rückwärts durchlaufen werden
- Einfügen vor und hinter existierende Listenelemente
- Elemente können problemlos gelöscht werden



# Doppelt verkettete Liste

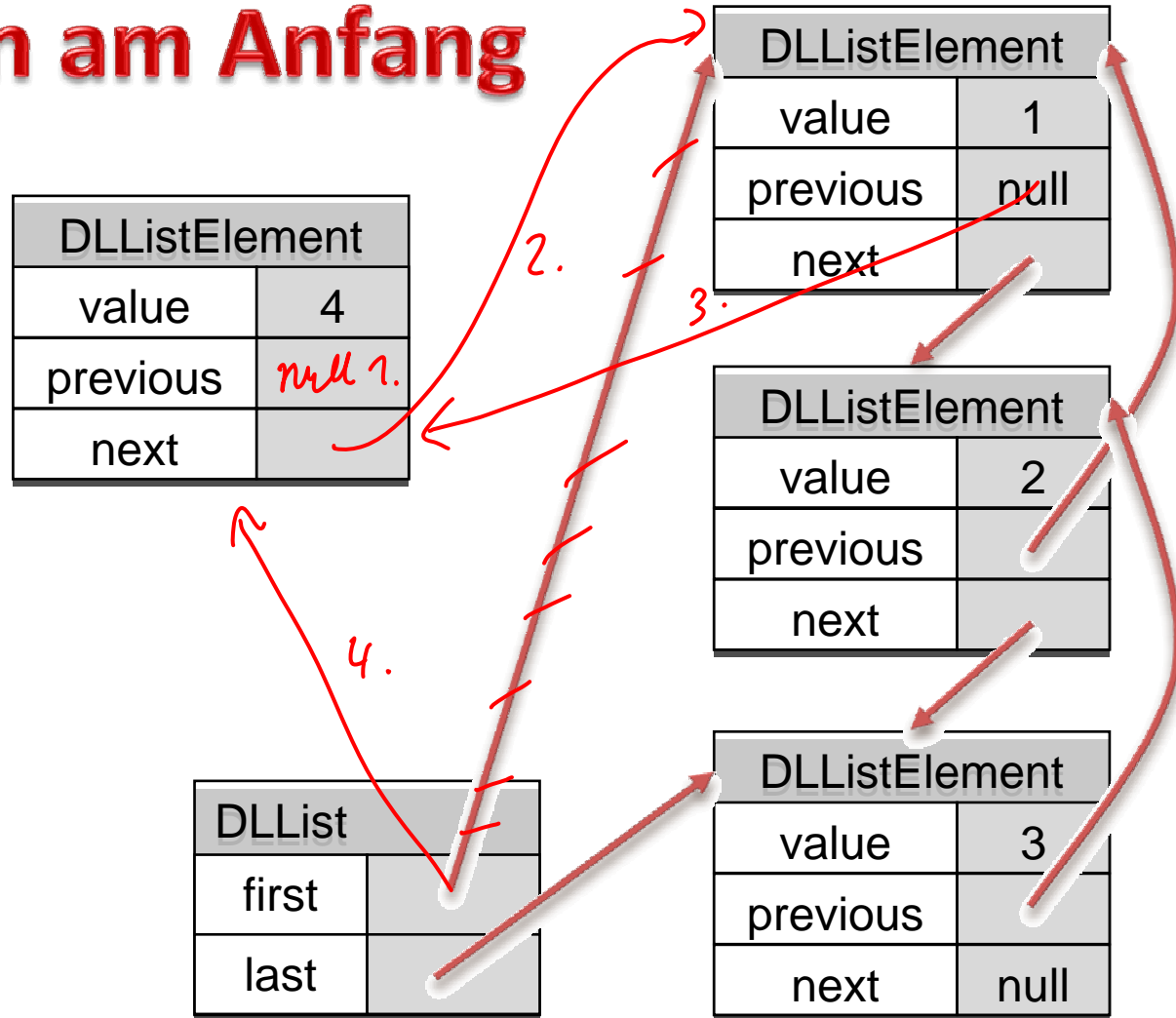
- Datentyp der Einträge:  
*int*

```
class DLListElement {  
    int entry;  
    DLListElement previous;  
    DLListElement next;  
  
    DLListElement(int entry,  
        DLListElement previous,  
        DLListElement next) {  
        this.entry = entry;  
        this.previous = previous;  
        this.next = next;  
    }  
}
```

```
class DLList {  
    DLListElement first;  
    DLListElement last;  
  
    DLList() {  
        first = last = null;  
    }  
  
    boolean empty() {  
        return first == null;  
    }  
}
```

# Doppelt verkettete Liste

## Einfügen am Anfang



# Doppelt verkettete Liste

## Einfügen am Ende

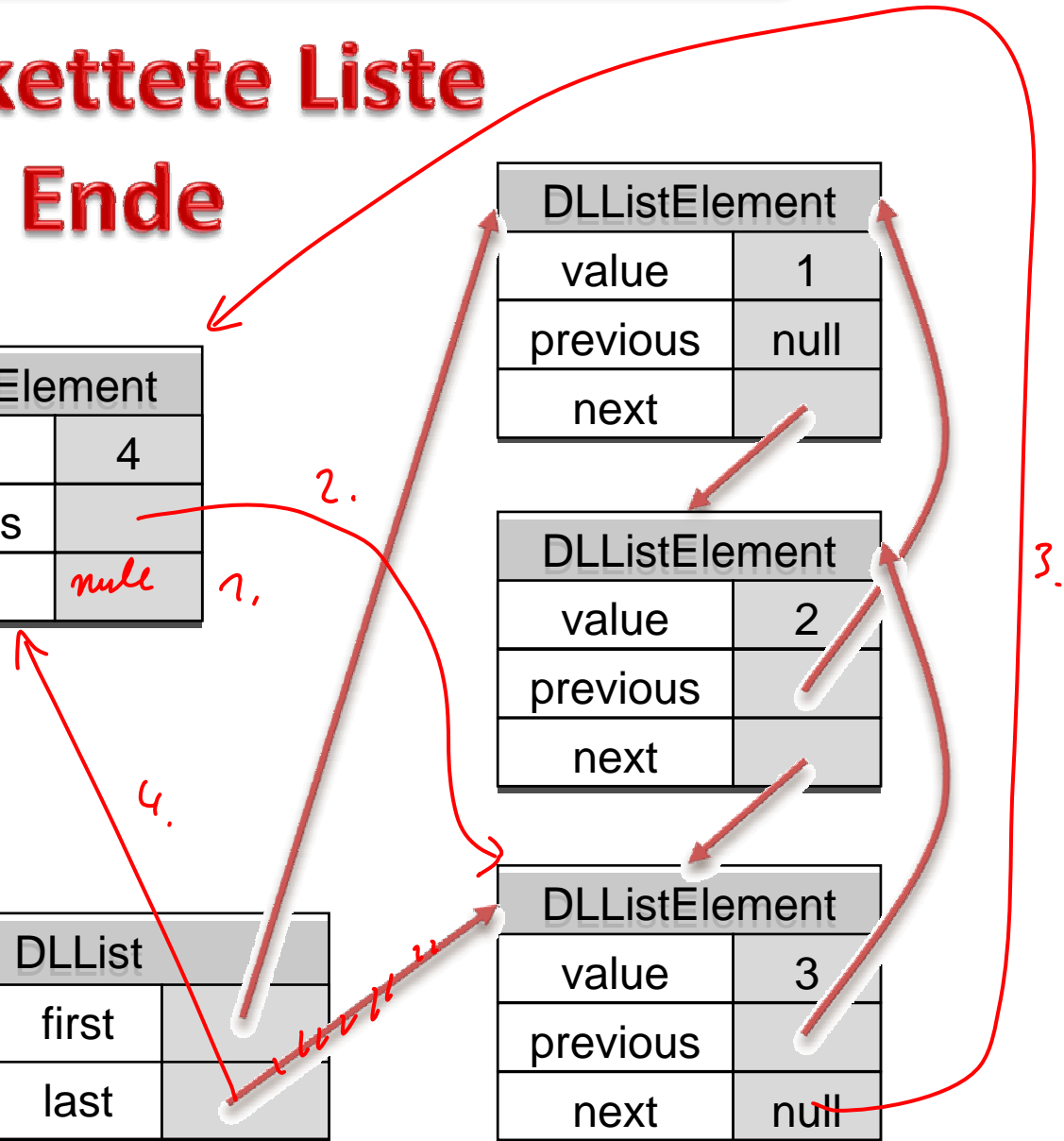
DLListElement	
value	4
previous	
next	<i>null</i>

DLList	
first	
last	

DLListElement	
value	1
previous	null
next	

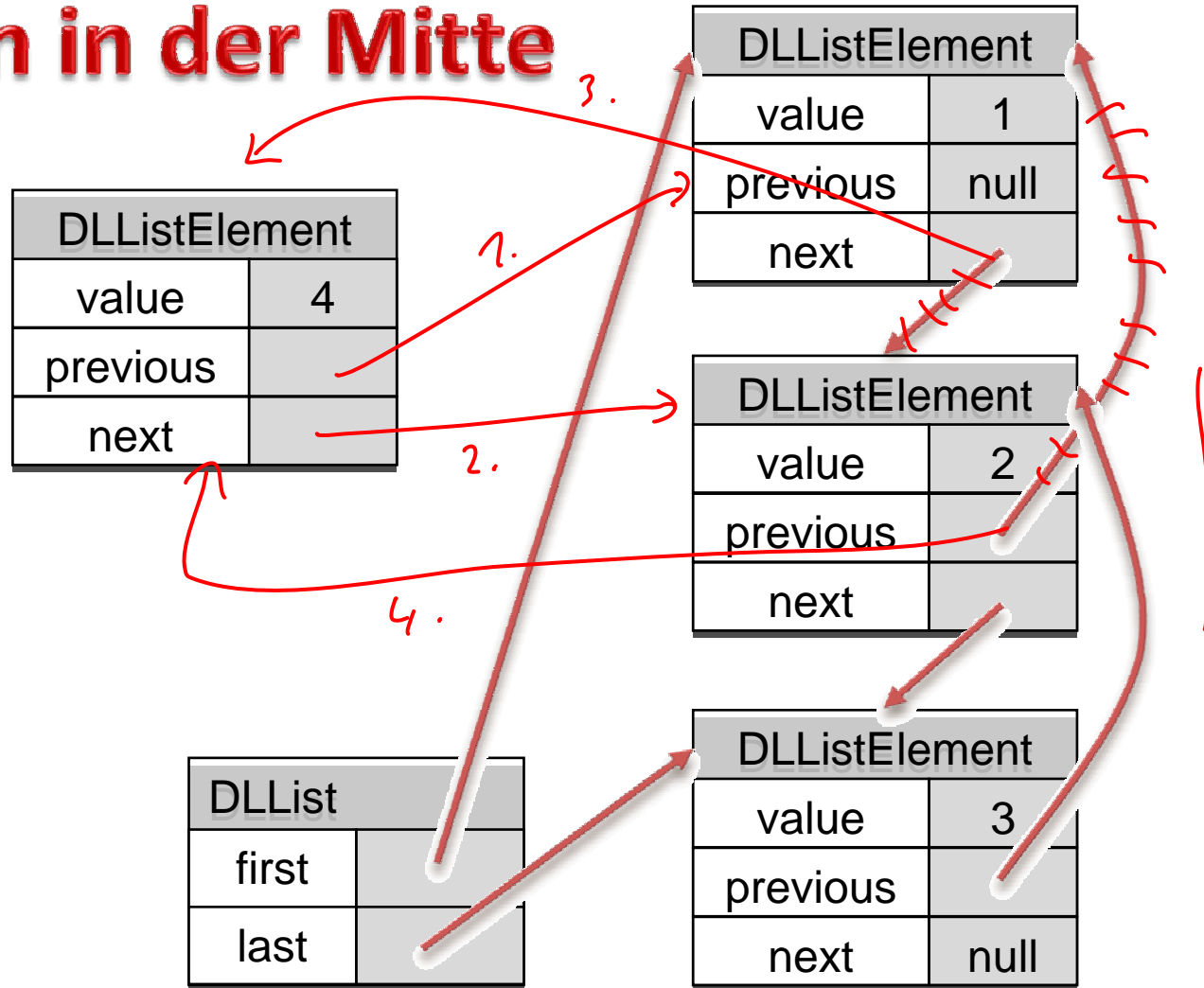
DLListElement	
value	2
previous	
next	

DLListElement	
value	3
previous	
next	null



# Doppelt verkettete Liste

## Einfügen in der Mitte



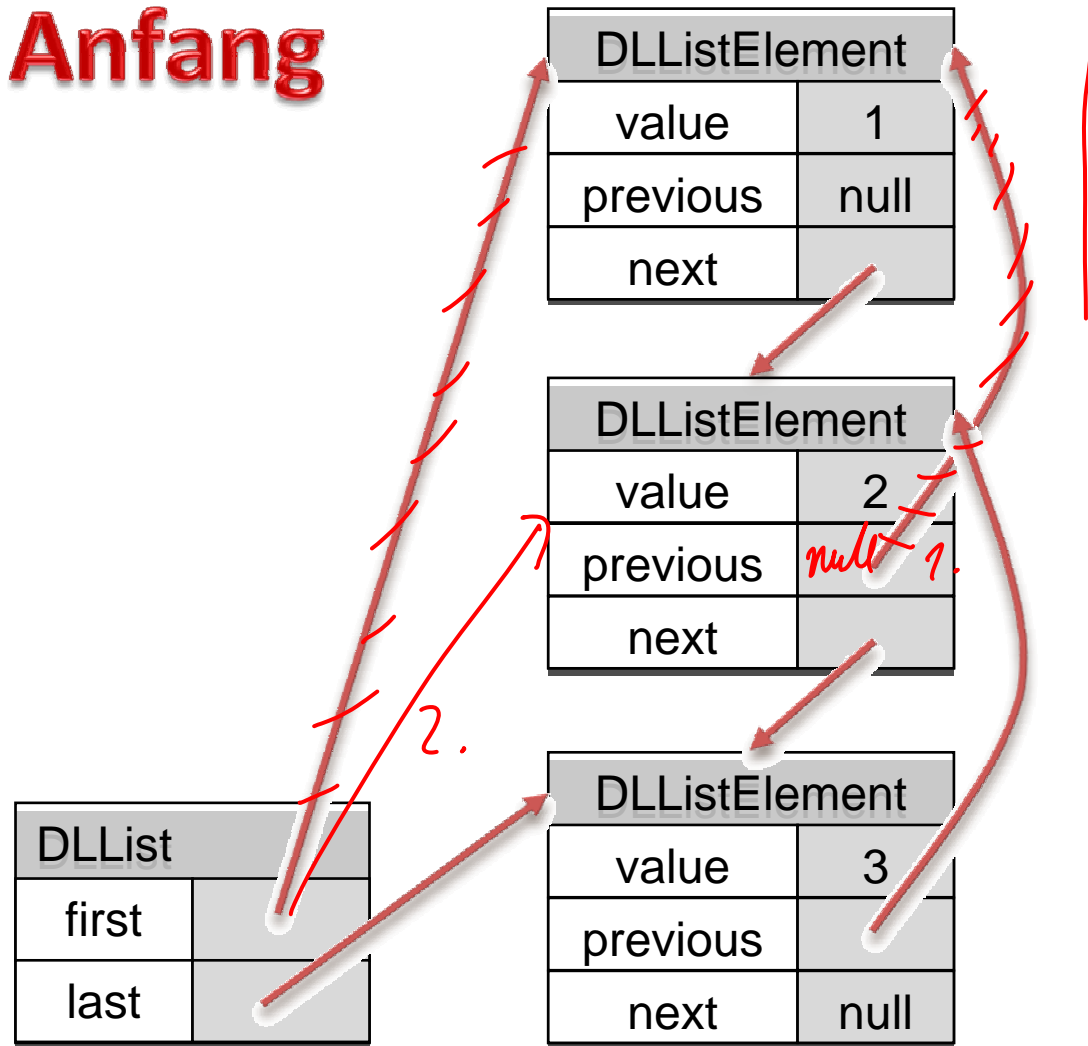
# Doppelt verkettete Liste

## Einfügen

```
void insertBefore(int entry, DLListElement anchor) {  
    DLListElement newElem;  
    if (anchor == null) {  
        newElem = new DLListElement(entry, last, null);  
        last = newElem;  
    } else {  
        newElem = new DLListElement(entry, anchor.previous, anchor);  
        anchor.previous = newElem;  
    }  
    if (newElem.previous == null) {  
        first = newElem;  
    } else {  
        newElem.previous.next = newElem;  
    }  
}
```

# Doppelt verkettete Liste

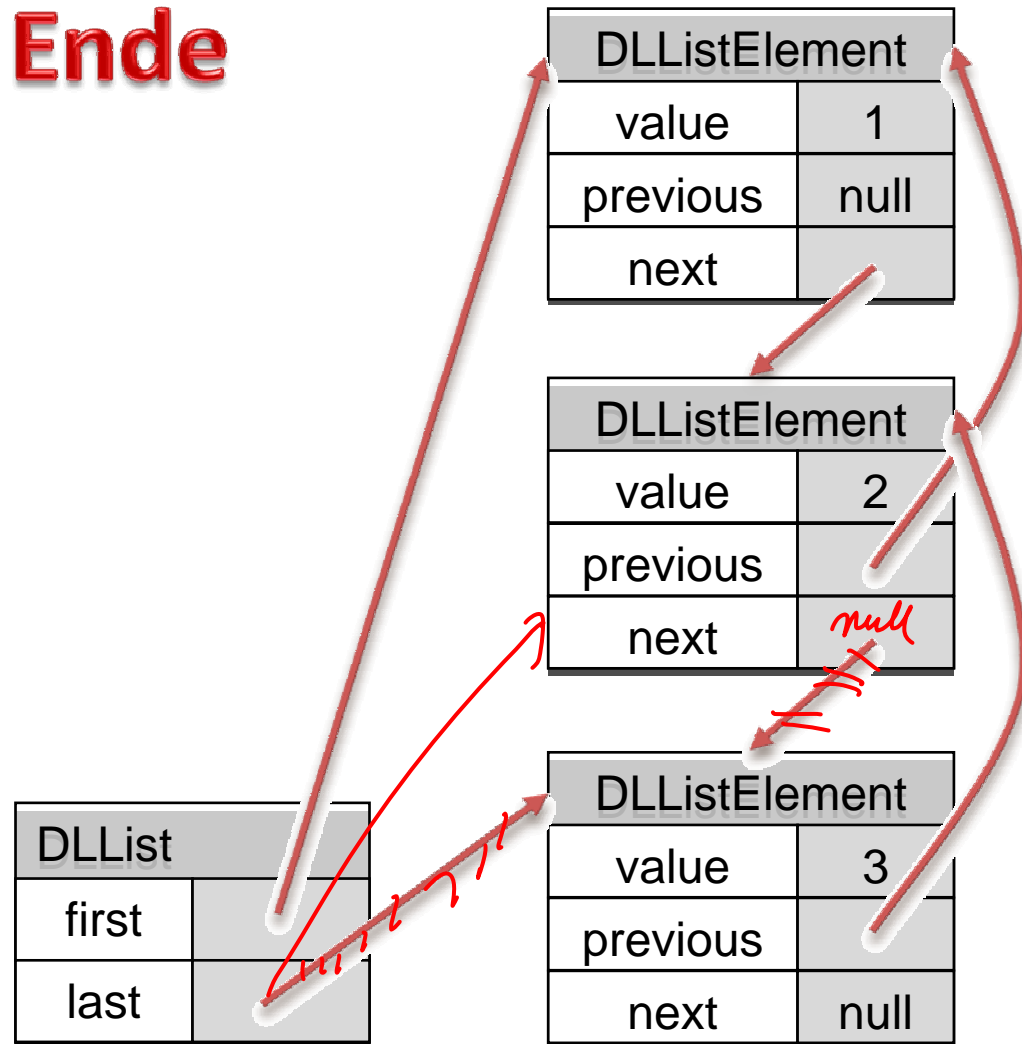
## Löschen am Anfang





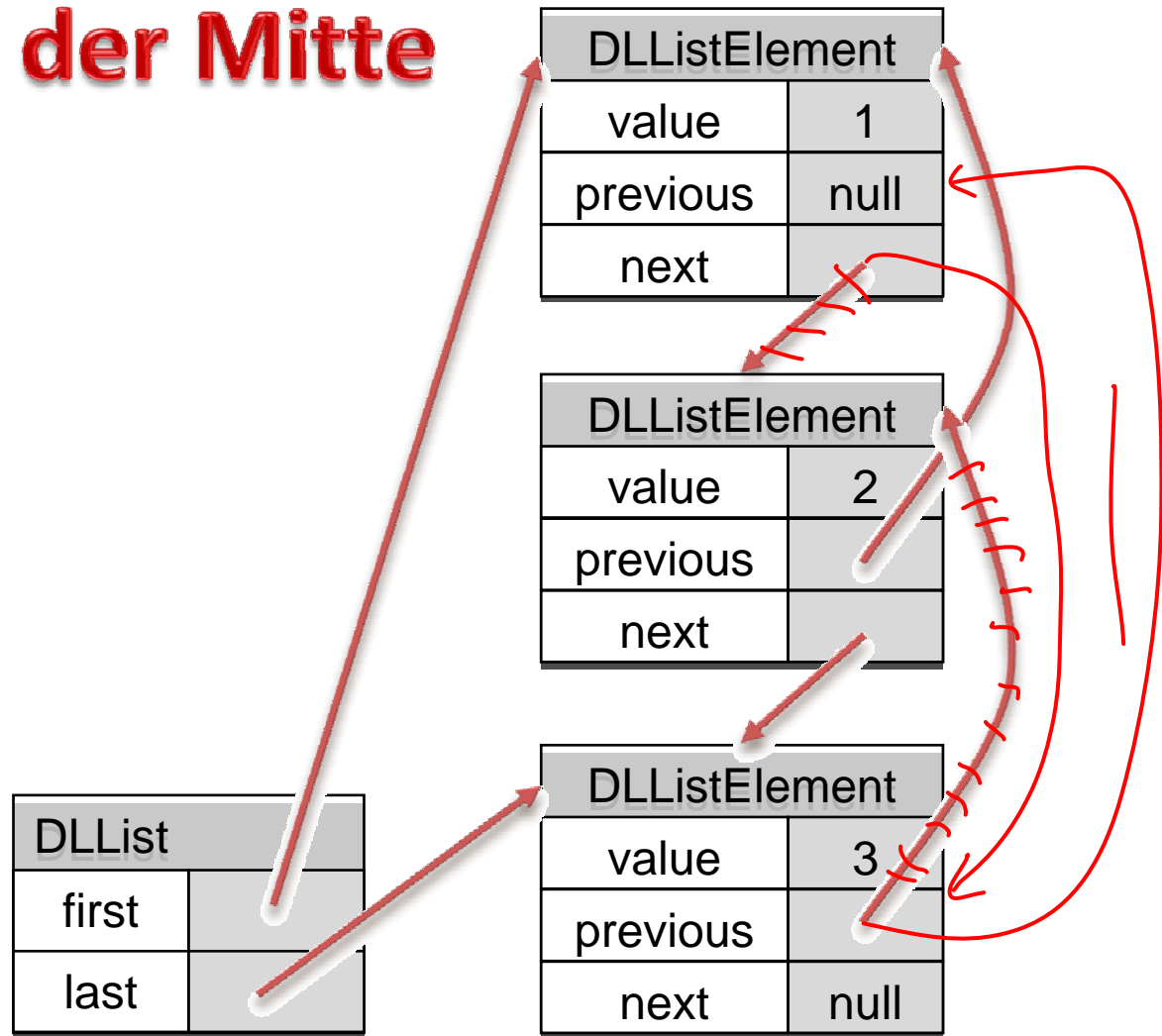
# Doppelt verkettete Liste

## Löschen am Ende



# Doppelt verkettete Liste

## Löschen aus der Mitte



# Doppelt verkettete Liste

## Löschen

```
void remove(DLListElement element) {  
    if (element.previous == null) {  
        first = element.next;  
    } else {  
        element.previous.next = element.next;  
    }  
    if (element.next == null) {  
        last = element.previous;  
    } else {  
        element.next.previous = element.previous;  
    }  
}
```

# Musterlösung zu Übungsblatt 8

- Aufgabe 1
  - *if* → *while*
    - Bedingung darf wegen der Seiteneffekte nicht 2x ausgewertet werden, sowohl aus der Bedingung selbst, als auch aus dem Ausdruck im *if*-Zweig
    - *break* gab's auch nicht mehr
  - *while* → *if*
    - Geht durch Rekursion mit Einschränkungen
  - Begründungen!
- Aufgabe 2
  - Wer *longs* mit / und % zerlegt hat, lief in Probleme wegen des Vorzeichens

Praktische Informatik I WS 2007/08

## Übungsblatt 8

Musterlösung

### Aufgabe 1 Der Grinch hat die Anweisungen geklaut (30%)

*Der Grinch mag nicht nur kein Weihnachten, sondern auch kein Java. Daher hat er fast alle Java-Kontrollanweisungen entwendet (do ...while, switch usw., aber auch den ?:-Operator). Übrig geblieben sind nur while und if-else. Eine Anweisung von beiden könnt ihr verstecken, um sie seinem diebischen Zugriff zu entziehen und somit auch weiterhin programmieren zu können. Zeigt, dass sich eine der beiden Anweisungen vollständig durch die andere ersetzen lässt. Diskutiert auch, warum die Ersetzung in der anderen Richtung im Allgemeinen nicht möglich ist. Die Anweisung if-else kann vollständig durch while ersetzt werden. Die allgemeine if-Anweisung ist von der Form:*

```
1 if (condition) statement1 else statement2
```

Dabei ist *condition* ein boolescher Ausdruck. Wir definieren zwei Variablen von Typ Boolean: *ifCondition* und *elseCondition*. Dann wird die allgemeine if-Form durch folgende while-Form ersetzt:

```
1 boolean ifCondition = condition;
2 boolean elseCondition = !ifCondition;
3 while (ifCondition) {
4     statement1
5     ifCondition = false;
6 }
7 while (elseCondition) {
8     statement2
9     elseCondition = false;
10 }
```

Prinzipiell kann die *while* durch *if-else* in Kombination mit Rekursion ersetzen, d.h.

```
1 while (condition) statement
```

lässt sich durch folgende Form nachbilden:

```
1 void whileFn() {
2     if (condition) {
3         statement
4         whileFn();
5     }
6 }
```

Das Problem ist allerdings, dass nun alle in *statement* und *condition* verwendeten Variablen global (d.h. Objekt- oder Klassenattribute) sein müssen, denn ansonsten hat *statement* keine dauerhaften Auswirkungen, auch nicht auf *condition*. Zudem ist die Anzahl der Schleifendurchläufe nun durch die Größe des Java-Stacks eingeschränkt. Diese Art der Ersetzung ist also mit starken Einschränkungen versehen.

# Übungsblatt 10

- Aufgabe 1
  - *Stack* und *Queue* mit selbstgemachten Listen implementieren
  - Signaturen beibehalten
  - Möglichst in Aufgabe 2 verwenden
- Aufgabe 2
  - Code verstehen
  - Den Code erweitern
  - Vollständigen Rechner implementieren
  - Fehler in der Eingabe erkennen
  - JavaDoc

Praktische Informatik I WS 2007/08

## Übungsblatt 10

Abgabe: 23.01.08

### Aufgabe 1 Warteschlangen unbeschränkt stapeln (40%)

In dem auf der PI-I Webseite bereitgestellten Paket *uebung10.zip* sind Implementierungen für einen beschränkten Stapel und eine beschränkte Warteschlange enthalten. Ersetzt beide durch unbeschränkte Varianten, die auf einfach verketteten Listen basieren. Die Schnittstelle soll dabei erhalten bleiben, d.h. die Methoden *push*, *pop*, *top* und *empty* sollen ihre Signaturen behalten.

### Aufgabe 2 Mit Schlangen und Stapeln rechnen (60%)

In dem bereitgestellten Paket befindet sich auch die Klasse *Calc*, in der die Methode *calc* Ausdrücke der folgenden Syntax auswertet:

```
expression = number { ( '+' | '-' ) number }
number = digit { digit }
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Erweitert die Klasse so, dass Ausdrücke dieser Syntax ausgewertet werden:

```
expression = term { ( '+' | '-' ) term }
term = factor { ( '*' | '/' ) factor }
factor = '(' expression ')' | '-' factor | number
number = digit { digit }
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Meldet Fehler mithilfe von Ausnahmen.

Dokumentiert die Lösungen beider Aufgaben mit JavaDoc.

## Aufgabe 2

$\text{expression} = \text{term} \{ ( '+' | '-' ) \text{term} \}$   
 $\text{term} = \text{factor} \{ ( '*' | '/' ) \text{factor} \}$   
 $\text{factor} = '(' \text{expression} ') ' | '-' \text{factor} | \text{number}$   
 $\text{number} = \text{digit} \{ \text{digit} \}$   
 $\text{digit} = '0' \dots '9'$

•  $e = \underline{1} + 2^* - (3 + 4) + 5$

$t = 1 + \dots$      $t = 2 \times - (3 + 4) + 5$

$f = 1 + \dots$      $f = 2 \times - (3 + 4) + 5$

$n = \underline{1} + \dots$      $n = \underline{2} \times \dots$

$f = - (3 + 4) + 5$

$f = \underline{(3 + 4)} + 5$

$e = 3 + 4 + 5$

$t = 3 + 4 + 5$

$f = 3 + 4 + 5$

$n = \underline{3} + 4 + 5$

$t = 5$

$f = 5$

$n = \underline{5}$

$t = 4 + 5$

$f = 4 + 5$

$n = \underline{4} + 5$

$$\begin{array}{l}
 4 \\
 3 \\
 2 \\
 1
 \end{array}
 \left|
 \begin{array}{l}
 7 \\
 7 \\
 7 \\
 7
 \end{array}
 \right|
 \sim
 \begin{array}{l}
 7 \\
 7 \\
 7 \\
 7
 \end{array}
 \left|
 \begin{array}{l}
 -14 \\
 -14 \\
 -14 \\
 -14
 \end{array}
 \right|
 \begin{array}{l}
 5 \\
 5 \\
 5 \\
 5
 \end{array}
 \left|
 \begin{array}{l}
 -8 \\
 -8 \\
 -8 \\
 -8
 \end{array}
 \right|$$

## Aufgabe 2

expression = term { ( '+' | '-' ) term }  
term = factor { ( '\*' | '/' ) factor }  
factor = '(' expression ')' | '-' factor | number  
number = digit { digit }  
digit = '0' ... '9'

+2      ✗

5++4      ✗

)      ✗

(3+4      ✗

(3+4a      ✗

5--2      ✓