

Effiziente verkettete Listen

Dr. Philipp Wendler

Zentralübung zur Vorlesung
„Einführung in die Informatik: Programmierung und Softwareentwicklung“

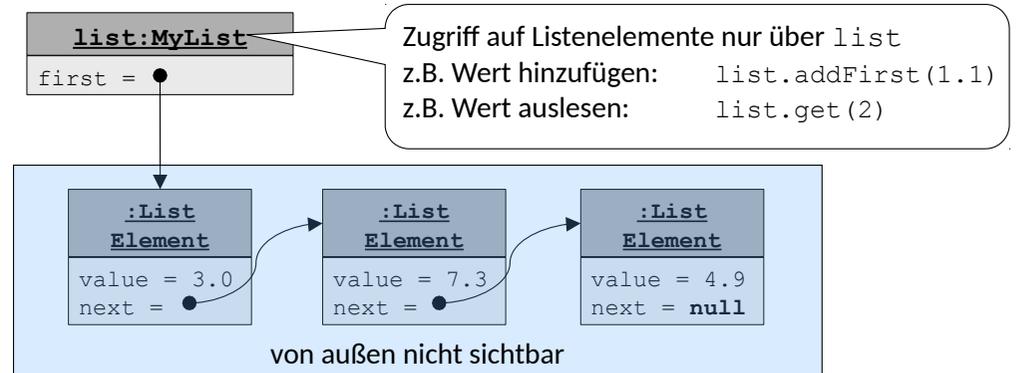
<https://www.sosy-lab.org/Teaching/2018-WS-InfoEinf/>

WS18/19

Verkettete Listen: Wiederholung

Eine verkettete Liste speichert die Listenelemente als Kette, wobei jedes Listenelement seinen Nachfolger kennt.

z.B. Repräsentation von `list=<3.0,7.3,4.9>`



Philipp Wendler:

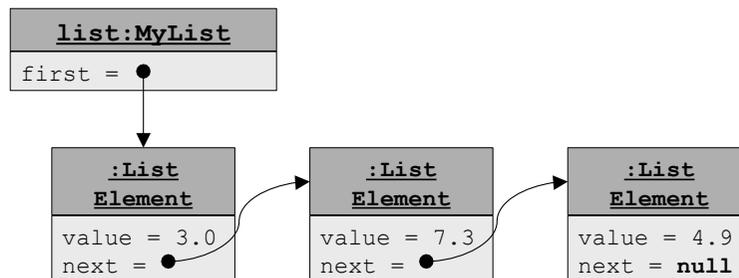
Effiziente verkettete Listen

2

Verkettete Listen: addFirst Wiederholung (I)

`list.addFirst(1.1)`

=> Hinzufügen von 1.1 am **Anfang** der Liste `list=<3.0,7.3,4.9>`



Philipp Wendler:

Effiziente verkettete Listen

3

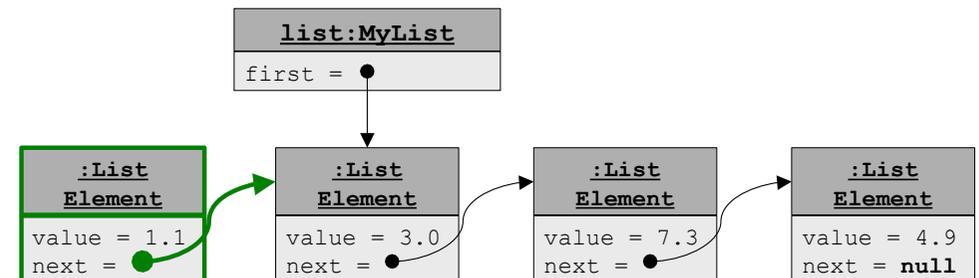
Verkettete Listen: addFirst Wiederholung (II)

`list.addFirst(1.1)`

=> Hinzufügen von 1.1 am **Anfang** der Liste `list=<3.0,7.3,4.9>`

Schritt 1: Erzeugen eines neuen `ListElement` mit

- `value = 1.1`
- `next = Zeiger auf ListElement für 3.0`



Philipp Wendler:

Effiziente verkettete Listen

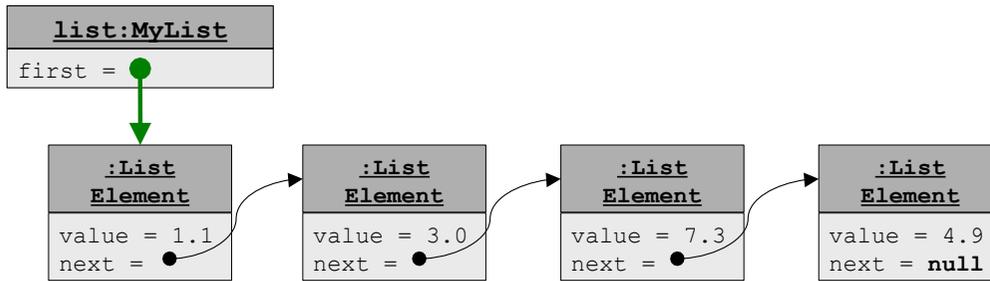
4

Verkettete Listen: addFirst Wiederholung (III)

list.addFirst(1.1)

=> Hinzufügen von 1.1 am **Anfang** der Liste list=<3.0,7.3,4.9>

Schritt 2: Speichern des neuen ListElement für 1.1 als erstes Element in der Liste, d.h. Zeiger first ändern

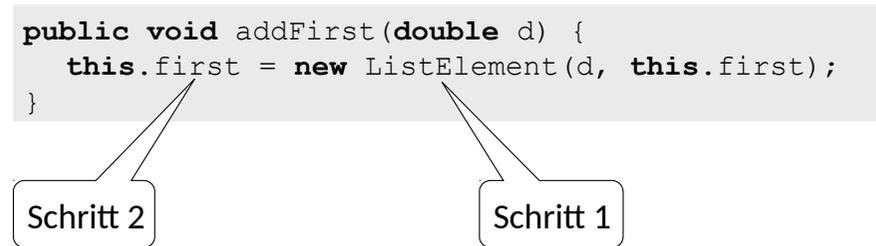


Verkettete Listen: addFirst Wiederholung (IV)

list.addFirst(1.1)

=> Hinzufügen von 1.1 am **Anfang** der Liste list=<3.0,7.3,4.9>

Implementierung von addFirst in der Klasse MyList:



Verkettete Listen: addFirst

```

public void addFirst(double d) {
    this.first = new ListElement(d, this.first);
}
    
```

Welches Problem tritt in obiger Implementierung auf?

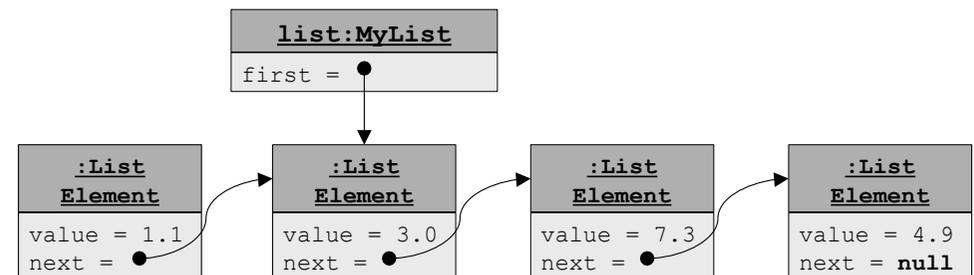
- a) Die Elemente der Liste bilden keine Kette.
- b) Das neue erste Element fehlt in der Liste.
- c) Das neue erste Element speichert den falschen Nachfolger.

Verkettete Listen: addFirst

```

public void addFirst(double d) {
    this.first = new ListElement(d, this.first);
}
    
```

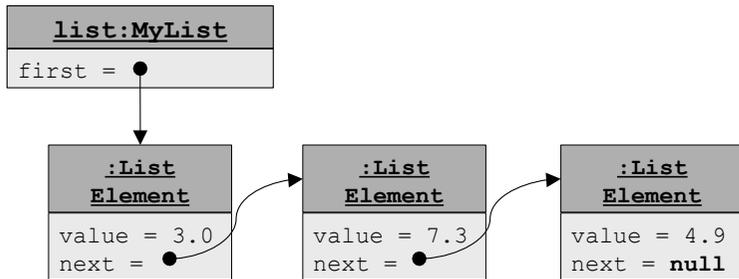
Welches Problem tritt in obiger Implementierung auf?



Verkettete Listen: addLast (I)

`list.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0,7.3,4.9>`



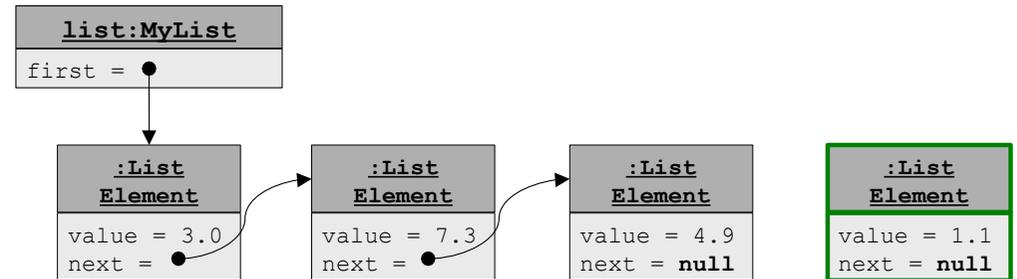
Verkettete Listen: addLast (II)

`list.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0,7.3,4.9>`

Schritt 1: Erzeugen eines neuen `ListElement` mit

- `value = 1.1`
- `next = null`

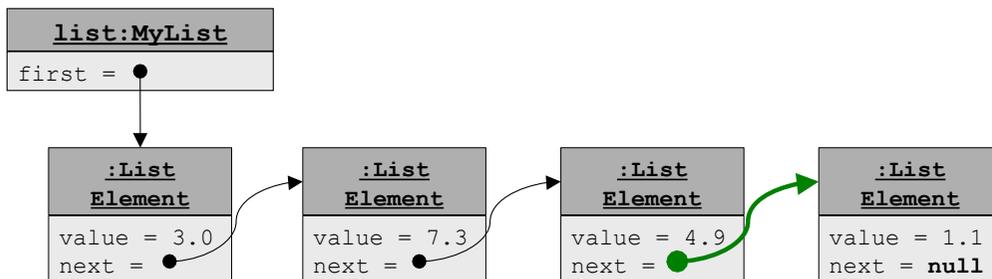


Verkettete Listen: addLast (III)

`list.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0,7.3,4.9>`

Schritt 2: Speichern des neuen `ListElement` für 1.1 als letztes Element in der Liste, d.h. die **gesamte** Liste bis zum letzten Element durchlaufen und dessen Zeiger `next` ändern



Verkettete Listen: addLast (IV)

`list.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0,7.3,4.9>`

Implementierung von `addLast` in der Klasse `MyList`:

```
public void addLast(double d) {
    ListElement newElement = new ListElement(d);

    ListElement e = this.first;
    while (e.getNext() != null) {
        e = e.getNext();
    }
    e.setNext(newElement);
}
```

Schritt 1

Schritt 2

Verkettete Listen: addLast

```
public void addLast(double d) {
    ListElement newElement = new ListElement(d);
}
```

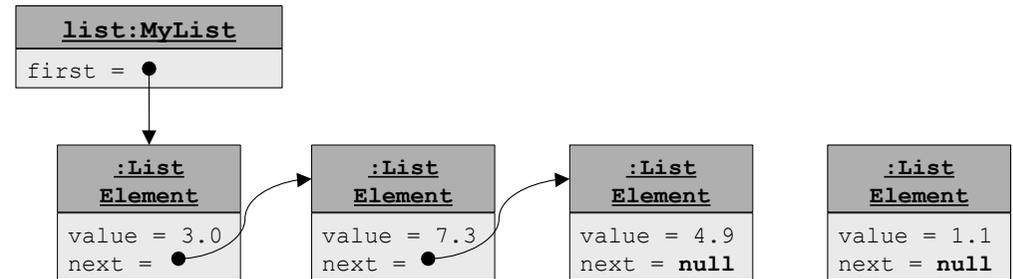
Welches Problem tritt in obiger Implementierung auf (ohne Schleife von Folie 12)?

- a) Die Elemente der Liste bilden keine Kette.
- b) Das neue letzte Element fehlt in der Liste.
- c) Das neue letzte Element speichert den falschen Nachfolger.

Verkettete Listen: addLast

```
public void addLast(double d) {
    ListElement newElement = new ListElement(d);
}
```

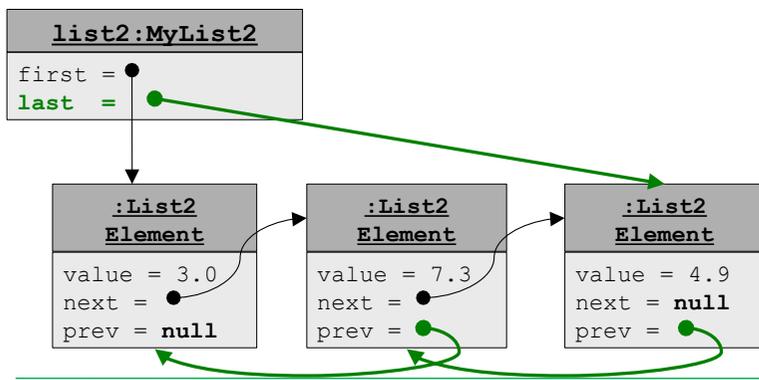
Welches Problem tritt in obiger Implementierung auf (ohne Schleife von Folie 12)?



Doppelt verkettete Listen

Eine doppelt verkettete Liste speichert die Listenelemente als Kette, wobei jedes Listenelement seinen Nachfolger **und Vorgänger** kennt.

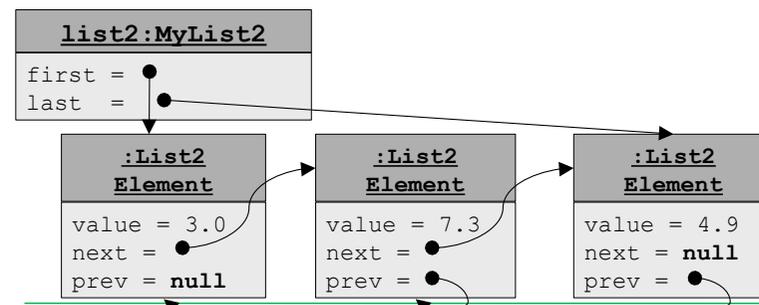
z.B. Repräsentation von `list2=<3.0, 7.3, 4.9>`



Doppelt verkettete Listen: addLast (I)

`list2.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list2=<3.0, 7.3, 4.9>`



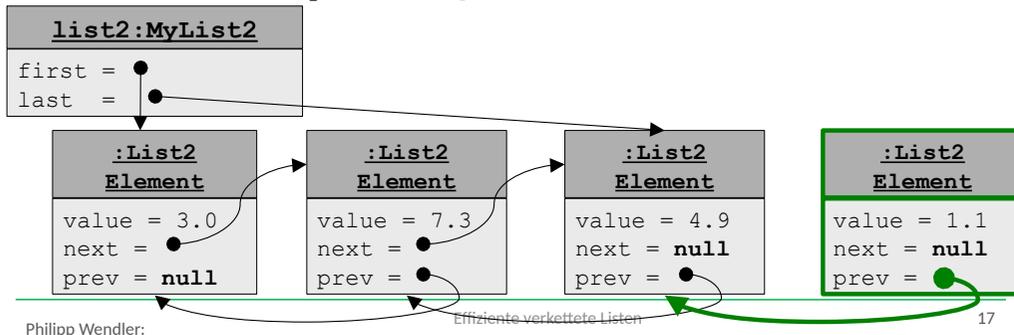
Doppelt verkettete Listen: addLast (II)

list2.addLast(1.1)

=> Hinzufügen von 1.1 am **Ende** der Liste list2=<3.0,7.3,4.9>

Schritt 1: Erzeugen eines neuen List2Element mit

- value = 1.1
- next = null, prev = Zeiger auf List2Element für 4.9



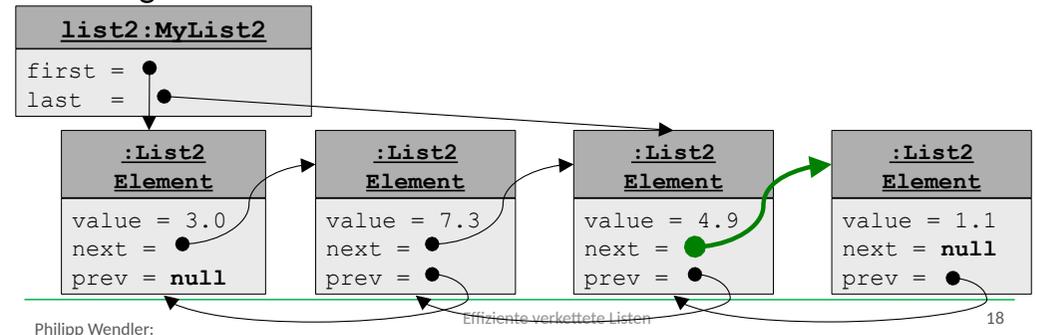
Doppelt verkettete Listen: addLast (III)

list2.addLast(1.1)

=> Hinzufügen von 1.1 am **Ende** der Liste list2=<3.0,7.3,4.9>

Schritt 2: Speichern des neuen List2Element für 1.1 als Nachfolger des List2Element für 4.9

d.h. Zeiger next des List2Element für 4.9 ändern

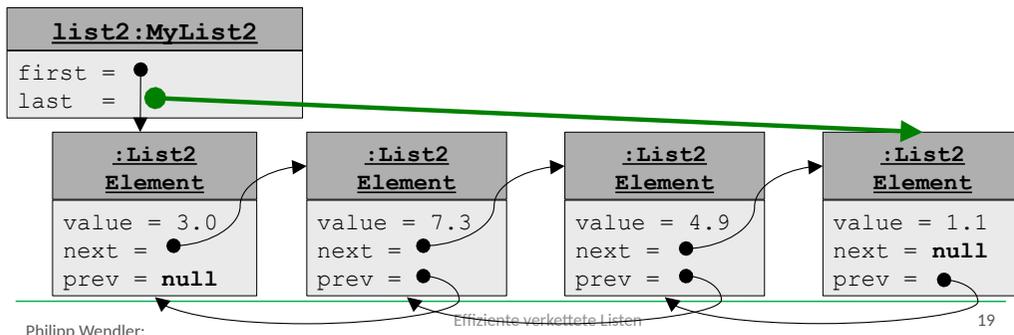


Doppelt verkettete Listen: addLast (IV)

list2.addLast(1.1)

=> Hinzufügen von 1.1 am **Ende** der Liste list2=<3.0,7.3,4.9>

Schritt 3: Speichern des neuen List2Element für 1.1 als letztes Element in der Liste, d.h. Zeiger last ändern



Doppelt verkettete Listen: addLast (V)

list2.addLast(1.1)

=> Hinzufügen von 1.1 am **Ende** der Liste list2=<3.0,7.3,4.9>

Implementierung von addLast in der Klasse MyList2:

```
public void addLast(double d) {
    List2Element newElement =
        new List2Element(d);
    newElement.setPrev(this.last);

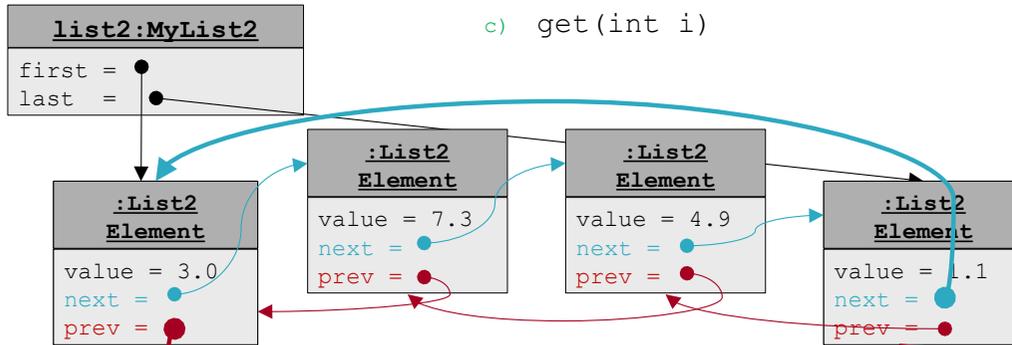
    this.last.setNext(newElement);
    this.last = newElement;
}
```

Schritt 1: new List2Element(d);
 Schritt 2: this.last.setNext(newElement);
 Schritt 3: this.last = newElement;

Variante einer doppelt verketteten Liste:

Welche Methode ist wegen des Zyklus schwierig zu implementieren?

- addFirst
- addLast
- get(int i)



Verkettete Listen: Zeitkomplexität

Zeitkomplexität von `addFirst`

- für einfach verkettete Listen: $O(1)$
- für doppelt verkettete Listen: $O(1)$

Zeitkomplexität von `addLast`

- für einfach verkettete Listen: $O(n)$ mit n = Länge der Liste
- für doppelt verkettete Listen: $O(1)$

Verkettete Listen: Zeitkomplexität

Bei welcher Anwendung sollte eine verkettete Liste einem Array vorgezogen werden?

- Einkaufsliste
- Preisliste aller Produkte eines Supermarkts
- Ergebnisliste eines Wettkampfs
- Einwohnerliste einer Stadt

Wiederholung aus der Vorlesung:

Arrays eignen sich zur Behandlung von Folgen mit fester Anzahl von Elementen, während verkettete Listen besser bei dynamischen Folgen sind.

Verwendung von `LinkedList`

Normalerweise implementiert man verkettete Listen nicht selbst, sondern verwendet die Standard-Implementierung `LinkedList`, die als doppelt verkettete Liste umgesetzt ist.

- Konstruktor:** z.B.

```
LinkedList<Double> list = new LinkedList<Double>();
```
- Methoden:**
 - `list.addFirst(...)`, `list.removeFirst(...)`
 - `list.addLast(...)`, `list.removeLast(...)`
 - `list.size()`
 - `list.get(i)`
 - ... und viele mehr (siehe Java API)

Angeben des Typs von Listenelementen (I)

Eine Liste soll nur Werte des gleichen Typs speichern können:

- bei eigener Implementierung:
 - z.B. fester Typ `double` für das Attribut `value` in `ListElement`
 - z.B. fester Typ `Figur` für das Attribut `value` in `FigurListElement`
- bei Verwendung von `LinkedList`:
 - z.B. `LinkedList<Double> doubleListe = new LinkedList<Double>();`
 - z.B. `LinkedList<Figur> figurListe = new LinkedList<Figur>();`

Innerhalb der spitzen Klammern dürfen nur Klassennamen stehen
(keine Grunddatentypen wie `double`)!

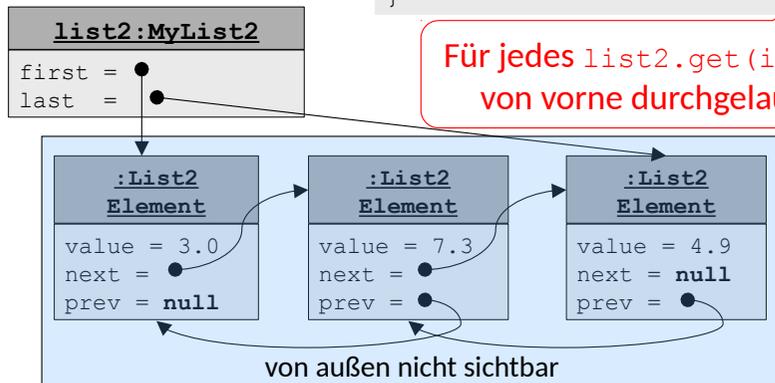
Angeben des Typs von Listenelementen (II)

- Alle Methoden sind für die Liste `doubleListe` auf Werte vom Typ `Double` festgelegt:
 - z.B. **richtig:** `doubleListe.addFirst(1.0);`
 - z.B. **falsch:** `doubleListe.addFirst(1);`
`doubleListe.addFirst("text");`
 - z.B. **richtig:** `Double d = doubleListe.get(0);`
 - z.B. **falsch:** `Figur f = doubleListe.get(0);`
- Alle Methoden sind für die Liste `figurListe` auf Werte vom Typ `Figur` festgelegt:
 - z.B. **richtig:** `figurListe.addFirst(new Figur(...));`
 - z.B. **falsch:** `figurListe.addFirst("text");`
 - z.B. **richtig:** `Figur f = figurListe.get(0);`
 - z.B. **falsch:** `Double d = figurListe.get(0);`

Durchlaufen einer verketteten Liste: Beispiel

```
double summe = 0.0;
for (int i = 0; i < list2.size(); i++) {
    summe = summe + list2.get(i);
}
```

Für jedes `list2.get(i)` muss die Liste von vorne durchgelaufen werden!



Durchlaufen einer LinkedList: Beispiel

```
LinkedList<Double> list = new LinkedList<Double> ();
... // Hinzufügen der Elemente
```

```
double summe = 0.0;
Iterator<Double> iterator = list.iterator();
while (iterator.hasNext()) {
    summe = summe + iterator.next();
}
```

Kurzform:

```
double summe = 0.0;
for (Double d : list) {
    summe = summe + d;
}
```

Der `Iterator` merkt sich, welches Element in der Liste er als nächstes zurückgeben muss.

Verkettete Listen: Zeitkomplexität

```
LinkedList<Double> list = new LinkedList<Double> ();  
... // Hinzufügen der Elemente  
  
double summe = 0.0;  
Iterator<Double> iterator = list.iterator();  
while (iterator.hasNext()) {  
    summe = summe + iterator.next();  
}
```

Was ist die Zeitkomplexität der Schleife?

- a) $O(1)$
- b) $O(n)$
- c) $O(n^2)$