

# **EINI**

# **LogWing/WiMa/MP**

**Einführung in die Informatik für  
Naturwissenschaftler und Ingenieure**

**Vorlesung      2 SWS      WS 18/19**

**Dr. Lars Hildebrand**  
**Fakultät für Informatik – Technische Universität Dortmund**  
**[lars.hildebrand@tu-dortmund.de](mailto:lars.hildebrand@tu-dortmund.de)**  
**<http://ls14-www.cs.tu-dortmund.de>**

## ▶ Kapitel 8 Dynamische Datenstrukturen

- Listen
- ▶ Bäume

## ▶ Unterlagen

- ▶ Dißmann, Stefan und Ernst-Erich Doberkat: *Einführung in die objektorientierte Programmierung mit Java*, 2. Auflage. München [u.a.]: Oldenbourg, 2002.  
(→ ZB oder Volltext aus Uninetz)
- ▶ Echte, Klaus und Michael Goedicke: *Lehrbuch der Programmierung mit Java*. Heidelberg: dpunkt-Verl, 2000.  
(→ ZB)

- Prolog
- Grundlagen
- Listen

## Begriffe

- ✓ Spezifikationen, Algorithmen, formale Sprachen
- ✓ Programmiersprachenkonzepte
- ✓ Grundlagen der imperativen Programmierung

- ✓ Algorithmen und Datenstrukturen
  - ✓ Felder
  - ✓ Sortieren
  - ✓ Rekursive Datenstrukturen (Baum, binärer Baum, Heap)
  - ✓ Heapsort

- Objektorientierung
  - ✓ Einführung
  - ✓ Vererbung
  - Anwendung

EINI LogWing /  
WiMa

## Kapitel 8

Dynamische  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Grundlagen
- Listen

# Grundlagen dynamischer Datenstrukturen I

- ▶ Dynamische Datenstrukturen:
  - ▶ Strukturen, die je nach Bedarf und damit **dynamisch** wachsen und schrumpfen können.
  - ▶ ≠ Felder/Arrays!
  
- ▶ **Grundidee:**
  - ▶ Dynamische Datenstrukturen bilden Mengen mit typischen Operationen ab.
  - ▶ Einzelne Elemente speichern die zu speichernden/verarbeitenden Daten.
  - ▶ Einzelne Elemente werden durch dynamische Datenstrukturen verknüpft.
  - ▶ → **Trennung von Datenstrukturierung & Nutzdaten**

- Prolog
- **Grundlagen**
- Listen

# Grundlagen dynamischer Datenstrukturen II

- ▶ Art der Elemente ist problemabhängig, variiert daher je nach Anwendung.
- ▶ Für die **Verknüpfung** existieren typische Muster:
  - ▶ Listen,
  - ▶ Bäume,
  - ▶ Graphen,
  - ▶ ...
- ▶ Objektorientierte Sicht:

Dynamische Datenstrukturen sind durch die **Art der Verknüpfung** der Elemente und die **Zugriffsmethoden** charakterisiert.

EINI LogWing /  
WiMa

## Kapitel 8

Dynamische  
Datenstrukturen

### In diesem Kapitel:

- Prolog
- **Grundlagen**
- Listen

## Wichtige dynamische Datenstrukturen:

### ▶ Listen

- ▶ lineare Listen
- ▶ doppelt verkettete Listen

### ▶ Bäume

- ▶ binäre Bäume
- ▶ binäre Suchbäume

### ▶ Graphen

- ▶ gerichtete Graphen
- ▶ ungerichtete Graphen

### ▶ Stack

### ▶ Schlangen

- Prolog
- **Grundlagen**
- Listen

# Grundlagen dynamischer Datenstrukturen IV

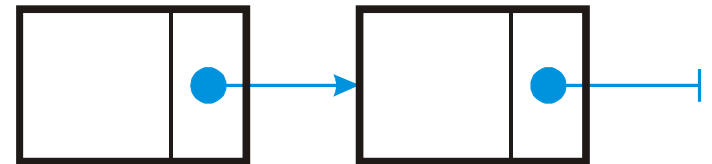
## Fragen zur Organisation der Datenstrukturen:

- ▶ Funktionen
  - ▶ Wie wird eine Instanz der Struktur initialisiert?
  - ▶ Wie werden Daten
    - eingefügt?
    - modifiziert?
    - entfernt?
  
- ▶ Wie wird in den Strukturen navigiert?
  
- ▶ Wie werden einzelne Werte in einer Struktur wiedergefunden?
  
- ▶ Wie werden alle in einer Struktur abgelegten Werte besucht?

# Grundlagen dynamischer Datenstrukturen V

- ▶ **Aufbau** von dynamischen Datenstrukturen :
  - ▶ Klassen enthalten Attribute (hier: **weiter**), die Referenzen auf Objekte der **eigenen Klasse** darstellen.
  - ▶ Diese Attribute schaffen die Möglichkeit, ein weiteres Objekt der Klasse an eine Referenz zu binden.
  - ▶ Die einzelnen Objekte sind in der Lage, gemeinsam eine komplexe Struktur durch **aufeinander verweisende Referenzen** zu bilden.

```
class Element {  
    Element weiter;  
    ...  
}
```



- Prolog
- **Grundlagen**
- Listen





# Dynamische Datenstrukturen – Grundlagen

Artikel im EINI-Wiki:

→ **Dynamische Datenstruktur**

## Kapitel 8

Dynamische  
Datenstrukturen

### In diesem Kapitel:

- Prolog
- **Grundlagen**
- Listen

# Listen

- ▶ Listen definieren eine **Reihenfolge von Elementen**, die gemäß dieser Reihenfolge miteinander verknüpft sind.
- ▶ Typische **Zugriffsmethoden**:
  - ▶ **Anfügen** eines neuen Elementes
  - ▶ **Einfügen** eines neuen Elementes an einer bestimmten Position in der Liste
  - ▶ **Auslesen** eines beliebigen Elementes der Liste
  - ▶ **Entfernen** eines beliebigen Elementes der Liste
  - ▶ **Abfrage**, ob die Liste leer ist
  - ▶ **Leeren** der Liste
- ▶ Auch wenn nicht alle Methoden realisiert sind, wird die Struktur als Liste bezeichnet.

- Prolog
- Grundlagen
- **Listen**

# Bestandteile einer Liste – Element

```
class Element {
    private int wert;           //Nutzinformation
    private Element weiter;    //Verwaltungsinformation

    public Element(int i) {
        wert = i;
        weiter = null;
    }
}
```

- ▶ Deklaration einer Klasse **Element** mit zwei **privaten Attributen** und einem **Konstruktor**
- ▶ Ein Objekt vom Typ **Element** enthält als **Attribute** eine **ganze Zahl** und eine **Referenz** auf ein weiteres Objekt des Typs **Element**.
- ▶ **Jedes** Objekt vom Typ **Element** besitzt eine Referenz auf ein weiteres Element: Man kann sie miteinander **verketten**.

EINI LogWing /  
WiMa

Kapitel 8

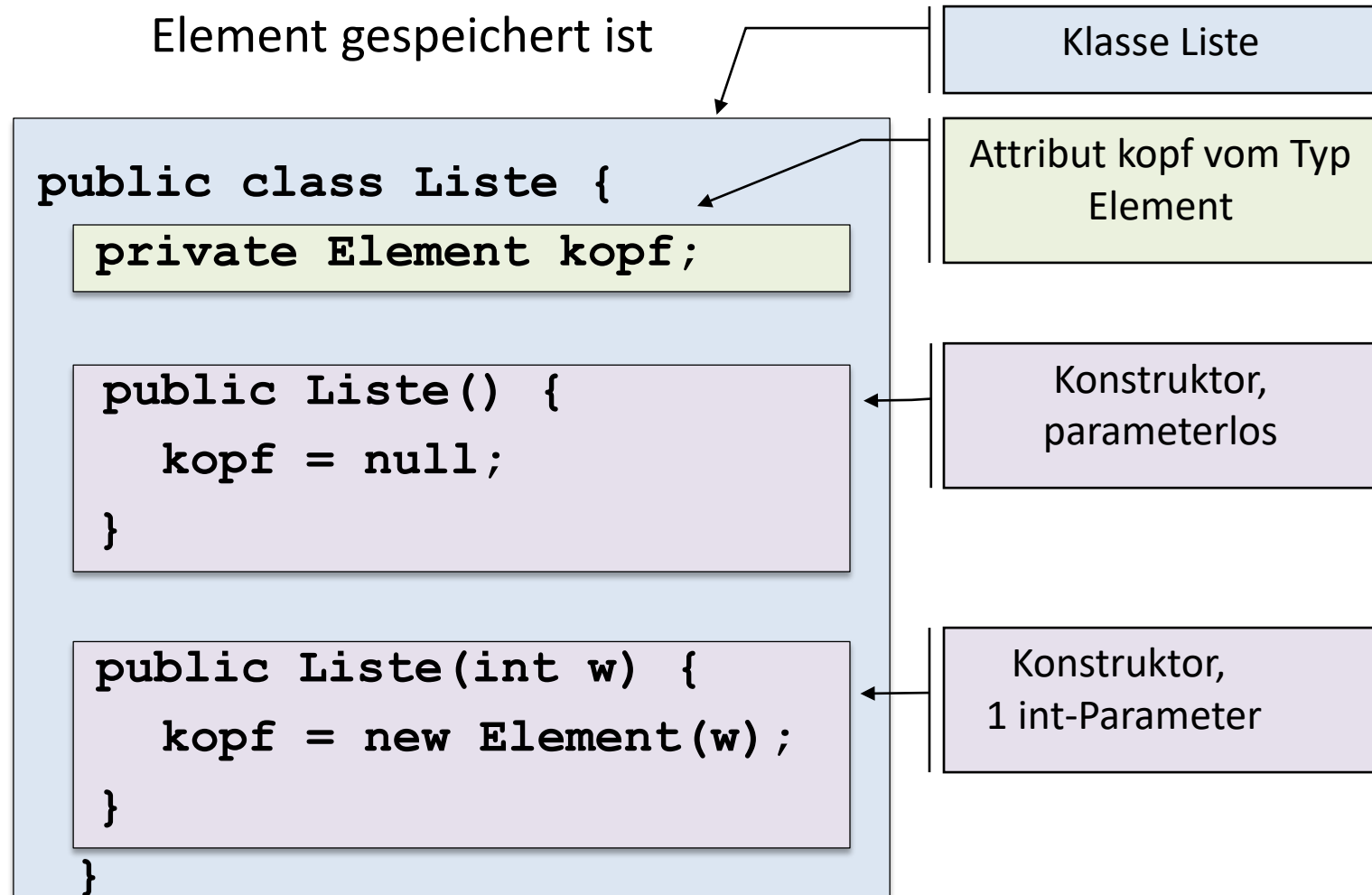
Dynamische  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Bestandteile einer Liste

- ▶ Woraus besteht eine Liste?
  - ▶ aus **Elementen**, die in der Liste gespeichert werden
  - ▶ aus der **Liste** selbst, die existiert, auch wenn kein Element gespeichert ist



# Konstruktion von Listen

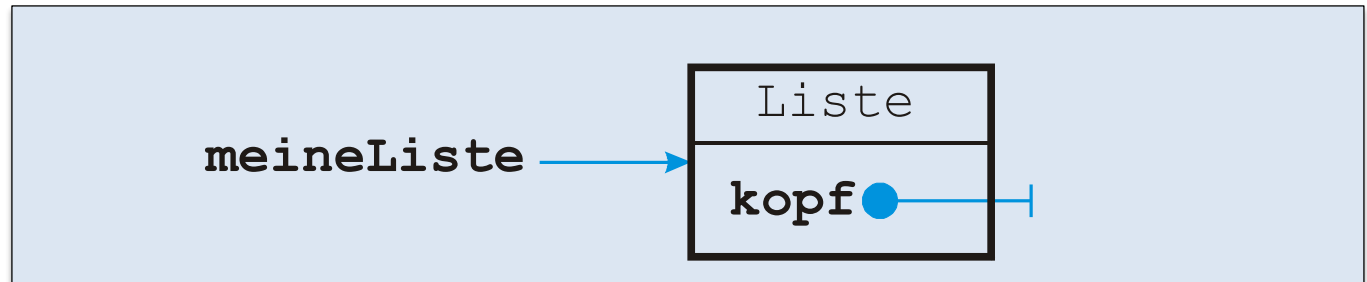
- ▶ Eine lineare Liste kann auf verschiedene Arten konstruiert werden:
  - ▶ Anhängen eines neuen Elementes
    - an den Anfang,
    - in die Mitte oder
    - an das Ende einer **bereits bestehenden** Liste.
- ▶ Zugriff auf die Liste wird durch eine Referenz realisiert,
  - ▶ die in der Klasse **Liste** realisiert ist und
  - ▶ die auf das erste Element der Liste zeigt.
  - ▶ Enthält eine Liste keine Elemente, zeigt die Referenz auf **null**.

- Prolog
- Grundlagen
- **Listen**

# Schema der Klasse `Liste`

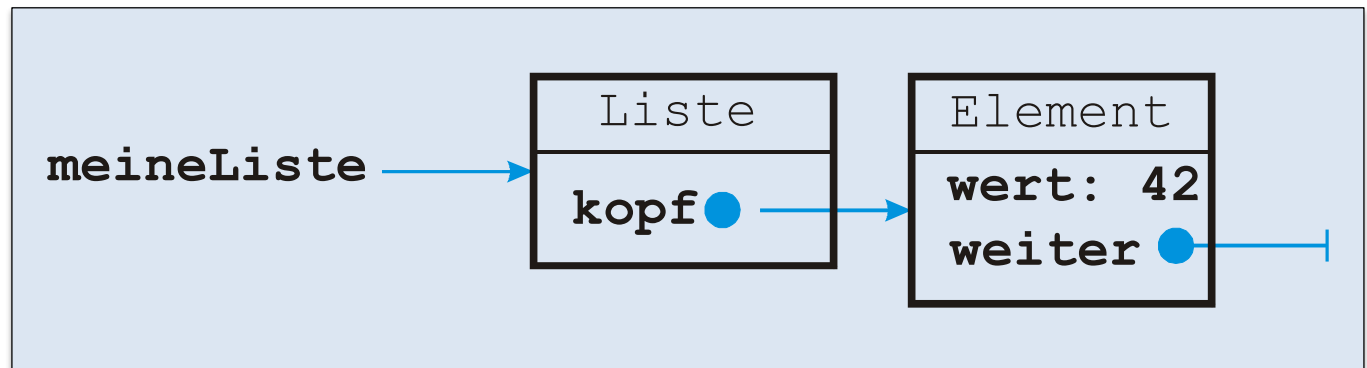
- ▶ Eine **leere Liste** erzeugen:

```
Liste meineListe = new Liste();
```



- ▶ Eine **Liste mit einem Element** erzeugen:

```
Liste meineListe = new Liste(42);
```



## Was fehlt noch?

- ▶ Typische **Zugriffsmethoden**:
  - ▶ **Anfügen** eines neuen Elementes
  - ▶ **Einfügen** eines neuen Elementes an einer bestimmten Position in der Liste
  - ▶ **Auslesen** eines beliebigen Elementes der Liste
  - ▶ **Entfernen** eines beliebigen Elementes der Liste
  - ▶ **Abfrage**, ob die Liste leer ist
  - ▶ **Leeren** der Liste

- Prolog
- Grundlagen
- **Listen**

# Klasse Element: Vollständige Implementierung I

```
class Element {
```

```
    private int wert;  
    private Element weiter;
```

```
    public Element(int i) {  
        wert = i; weiter = null;  
    }
```

```
    public Element(int i, Element e) {  
        wert = i; weiter = e;  
    }
```

```
    public void SetzeWert(int i) {wert = i;}
```

```
    public int HoleWert() {return wert;}
```

```
    public void SetzeWeiter(Element e) {weiter = e;}
```

```
    public Element HoleWeiter() {return weiter;}
```

```
}
```

EINI LogWing /  
WiMa

## Kapitel 8

Dynamische  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**



## Anmerkungen

- ▶ Es existiert ein zusätzlicher Konstruktor, der das neue Element **vor** ein bestehendes Element einreicht.
- ▶ Implementierung von Get- und Set-Methoden für den Zugriff auf die privaten Attribute:
  - ▶ `public void SetzeWert(int i)`
  - ▶ `public int HoleWert()`
  - ▶ `public void SetzeWeiter(Element e)`
  - ▶ `public Element HoleWeiter()`

# An den Anfang der Liste einfügen I

```
class Liste {  
    ...  
    public void FuegeEin(int neuerWert) {  
        kopf = new Element(neuerWert, kopf);  
    }  
}
```

EINI LogWing /  
WiMa

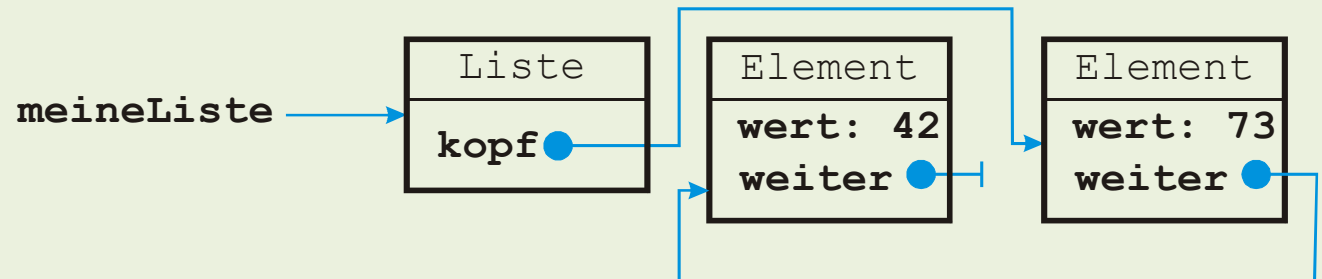
## Kapitel 8

Dynamische  
Datenstrukturen

```
Liste meineListe = new Liste(42);
```



```
meineListe.FuegeEin(73);
```



In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# An den Anfang der Liste einfügen II

## Anmerkungen zu FuegeEin()

- ▶ Ein neues Element wird erzeugt:
  - ▶ `kopf = new Element(neuerWert, kopf) ;`
  - ▶ `neuerWert` enthält die Nutzungsinformation.
  - ▶ `kopf` enthält die Referenz auf das alte erste Element.
  - ▶ Das neue Element referenziert das alte erste Element.
- ▶ Das neue Element wird zum neuen Kopf der Liste:
  - ▶ `kopf = new Element(neuerWert, kopf) ;`
- ▶ Wichtig: Die Referenz auf das alte erste Element darf nicht verloren gehen!

- Prolog
- Grundlagen
- **Listen**

# Ausgeben der Liste I

```
public class Liste {  
    ...  
    public void ZeigeListe() {  
        Element aktuellesElement = this.kopf;  
        while (aktuellesElement != null) {  
            System.out.println(aktuellesElement.HoleWert());  
            aktuellesElement = aktuellesElement.HoleWeiter();  
        }  
    }  
}
```

```
Liste meineListe = new Liste(42);  
meineListe.FuegeEin(73);  
meineListe.ZeigeListe();
```

```
> run TestListe
```

```
73
```

```
42
```

EINI LogWing /  
WiMa

Kapitel 8

Dynamische  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Ausgeben der Liste II

## Anmerkungen zu ZeigeListe()

- ▶ Anzahl der Elemente variabel, daher Programmierung einer Schleife notwendig
- ▶ Start ist das Element, auf das **kopf** verweist:

```
Element aktuellesElement = this.kopf;
```

- ▶ Ist kein Element in der Liste gespeichert, verweist **kopf** auf **null**.
- ▶ Solange das aktuelle Element **!= null** gilt:
  - ▶ Der Wert des aktuellen Elementes wird ausgegeben:  

```
System.out.println(aktuellesElement.HoleWert());
```
  - ▶ Das aktuelle Element wird auf das nächste gesetzt:  

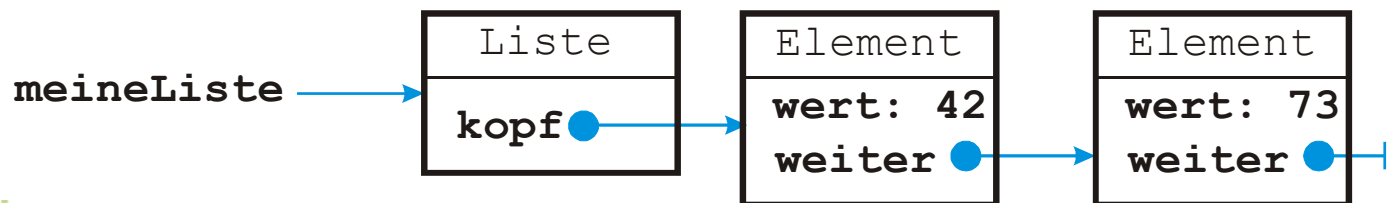
```
aktuellesElement = aktuellesElement.HoleWeiter();
```

- Prolog
- Grundlagen
- **Listen**

# An das Ende der Liste anfügen I

```
public class Liste {  
    ...  
    public void FuegeAn(int neuerWert) {  
        Element aktElement = this.kopf;  
        if (aktElement == null)  
            FuegeEin(neuerWert);  
        else {  
            while (aktElement.HoleWeiter() != null) {  
                aktElement = aktElement.HoleWeiter();  
            }  
            aktElement.SetzeWeiter(new Element(neuerWert));  
        }  
    }  
}
```

```
Liste meineListe = new Liste(42);  
meineListe.FuegeAn(73);
```



# An das Ende der Liste anfügen II

## Anmerkungen zu FuegeAn()

▶ Anzahl der Elemente variabel, daher Programmierung einer Schleife notwendig

▶ Start ist das Element, auf das **kopf** verweist:

```
Element aktElement = this.kopf;
```

▶ Ist **kein Element** in der Liste gespeichert, kann das neue Element mit **FuegeEin()** eingetragen werden.

▶ Ansonsten muss das **Ende der Liste** gesucht werden:

```
while (aktElement.HoleWeiter() != null) {  
    aktElement = aktElement.HoleWeiter();  
}
```

▶ Dann kann dort das neue Element mit dem letzten Element **verbunden** werden:

```
aktElement.SetzeWeiter(new Element(neuerWert));
```

# Etwas testen...

```
public class TestListe {  
    public static void main(String[] args) {  
        Liste meineListe = new Liste(42);  
        meineListe.FuegeEin(73);  
        meineListe.ZeigeListe();  
  
        meineListe = new Liste();  
        meineListe.ZeigeListe();  
        meineListe.FuegeEin(42);  
        meineListe.FuegeAn(73);  
        meineListe.ZeigeListe();  
    }  
}
```

```
> run TestListe  
73  
42  
42  
73
```

EINI LogWing /  
WiMa

Kapitel 8  
Dynamische  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**



# Effizienz der Klasse `Lineare Liste`

## Aufwand

- ▶ Erzeugen einer Liste
  - ▶ eine Instanziierung
- ▶ **FuegeEin ()**
  - ▶ unabhängig von der Anzahl der gespeicherten Elemente
- ▶ **ZeigeListe ()**
  - ▶ abhängig von der Anzahl der gespeicherten Elemente
- ▶ **FuegeAn ()**
  - ▶ Erfordert bei jedem Aufruf ein vollständiges Durchlaufen der Liste.
  
  - ▶ Eine sehr viel **effizientere Realisierung** dieser Listenoperation wäre möglich, wenn
    - neben dem ersten Element
    - auch das letzte Element der Liste unmittelbar erreichbar wäre.

- Prolog
- Grundlagen
- **Listen**

# Klasse EffizienteListe I

## Änderungen an den Attributen & Konstruktoren

- ▶ Neues Attribut **fuss**, das das letzte Element der Liste referenziert
- ▶ Setzen von **fuss** in die Konstruktoren

```
public class EffizienteListe {  
  
    private Element kopf;  
    private Element fuss;  
  
    public EffizienteListe() {  
        kopf = null;  
        fuss = null;  
    }  
  
    public EffizienteListe(int w) {  
        kopf = new Element(w);  
        fuss = kopf;  
    }  
}
```

- Prolog
- Grundlagen
- **Listen**

# Klasse `EffizienteListe` II

## Änderungen an der Methode `FuegeAn()`:

- ▶ kein Suchen nach dem Ende der Liste
- ▶ Das letzte Element der Liste ist immer in **fuss** gespeichert.
- ▶ Direkter Zugriff auf das letzte Element ist möglich.

```
public void FuegeAn(int neuerWert) {
    Element neuesElement = new Element(neuerWert);

    if (fuss == null) {
        kopf = neuesElement;
        fuss = neuesElement;
    } else {
        fuss.SetzeWeiter(neuesElement);
        fuss = neuesElement;
    }
}
```

### In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Klasse `EffizienteListe` III

## Änderungen an der Methode `ZeigeListe()`:

- ▶ keine

```
public void ZeigeListe() {
    Element aktuellesElement = this.kopf;
    while (aktuellesElement != null) {
        System.out.println( aktuellesElement.HoleWert() );
        aktuellesElement = aktuellesElement.HoleWeiter();
    }
}
```

### In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Klasse `EffizienteListe` IV

## Änderungen an der Methode `FuegeEin()`

- ▶ Sonderfall: Nur falls noch kein Element in der Liste ist, muss die `fuss`-Referenz berücksichtigt werden!

```
public void FuegeEin(int neuerWert) {
    kopf = new Element(neuerWert, kopf);
    if (fuss == null)
        fuss = kopf;
}
```

### In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Klasse `EffizienteListe V`

## Änderungen an der Klasse `Element`

- ▶ keine

## Beobachtungen:

- ▶ Alle Änderungen betreffen die **Verwaltungsinformationen** der Klasse `Liste`.
- ▶ Die Klasse `Element` bleibt völlig **unbeeinflusst**.
- ▶ Die Methode `fuegeAn()` hat nun eine konstante Laufzeit.

### In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Einfügen in geordnete Liste I

EINI LogWing /  
WiMa

## Kapitel 8

Dynamische  
Datenstrukturen

- ▶ Problem: Einordnen eines Werts in eine aufsteigend geordnete Liste
- ▶ Keine zwei Elemente haben die identische Belegung des Attributs **wert**.
- ▶ Der Algorithmus ist auf natürliche Weise **rekursiv**.

### In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Einfügen in geordnete Liste II

## ▶ Idee:

- ▶ Sei  $x$  der einzufügende Wert
- ▶ 1. Fall:  $x$  kleiner als 1. Element  $\Rightarrow$  Einfügen am Anfang
- ▶ 2. Fall:  $x$  größer als 1. Element:
  - Suche passende Position in der Liste
  - Trenne Liste in Anfangs- und Endteil auf
  - Setze Element an den Anfang des Endteils und verbinde Teillisten
  - Falls  $x$  größer als das letzte Element der Liste, ist  $x$  des neue Ende der Liste.

- Prolog
- Grundlagen
- **Listen**



# Einfügen in geordnete Liste III

## Präzisierung des Algorithmus

Folgende Fälle sind zu unterscheiden:

### ▶ **kopf == null**

- ▶ Einen Sonderfall bildet die Situation, dass die Liste leer ist, also noch kein Element enthält.
- ▶ Es muss ein erstes Element angelegt werden, das sicherlich eine geordnete, einelementige Liste bildet.

### ▶ **kopf != null:**

- ▶ Wir definieren eine private Methode **Positioniere**, die als Parameter den einzuordnenden Wert und eine Referenz auf den Anfang einer Teilliste übergeben bekommt.
- ▶ Als Ergebnis gibt **Positioniere** eine Referenz auf **Element** zurück, die auf die Teilliste verweist, in die  $x$  einsortiert ist.

# Einfügen in geordnete Liste IV

- ▶ Sei **anfang** die an **Positioniere** übergebene Teilliste und gelte:
  - ▶ **x < anfang.wert**:
    - Erzeuge ein neues Element und füge es am Anfang der bei **anfang** beginnenden Teilliste ein.
  - ▶ **x > anfang.wert**:
    - Füge x in die mit **anfang.weiter** beginnende Restliste ein, indem hierfür **Positioniere** mit den entsprechenden Parametern erneut aufgerufen wird.

- Prolog
- Grundlagen
- **Listen**

# Einordnen mit Hilfe von Positioniere

```
class Liste {
    ...

    void OrdneEin(int i) {
        kopf      = Positioniere(kopf, i);
    }

    private Element Positioniere(Element einElement, int i) {

        if (einElement == null)
            einElement = new Element(i);

        else {

            if (i < einElement.HoleWert()) {
                einElement = new Element(i, einElement);
            }

            if (i > einElement.HoleWert()) {
                einElement.SetzeWeiter(
                    Positioniere(einElement.HoleWeiter(), i));
            }
        }

        return einElement;
    }
}
```

EINI LogWing /  
WiMa

## Kapitel 8

Dynamische  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Durchlaufen einer Struktur

EINI LogWing /  
WiMa

Kapitel 8  
Dynamische  
Datenstrukturen

- ▶ In vielen Anwendungen, die auf dynamischen Datenstrukturen basieren, besteht die Notwendigkeit, alle Elemente der Struktur genau einmal zu besuchen.
- ▶ Dies gilt für Listen wie für andere dynamische Strukturen.
- ▶ Dieses möglichst nur einmalige Besuchen aller Elemente nennt man **Durchlaufen einer Struktur**.
- ▶ Anwendungsbeispiele: Prüfen auf Vorhandensein, Einsortieren, aber auch Ausgabe

In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Durchlaufen einer Liste I

```
class Liste {
```

```
    ...
```

```
    public void ZeigeListe() {  
        Element aktElement = this.kopf;  
        while (aktElement != null) {  
            System.out.println(aktElement.HoleWert());  
            aktElement = aktElement.HoleWeiter();  
        }  
    }
```

```
    public void ZeigeListeRekursiv() {  
        ZeigeListeRekursiv(kopf);  
    }  
  
    private void ZeigeListeRekursiv(Element aktElement) {  
        if (aktElement != null) {  
            System.out.println(aktElement.HoleWert());  
            ZeigeListeRekursiv(aktElement.HoleWeiter());  
        }  
    }  
}
```

EINI LogWing /  
WiMa

Kapitel 8

Dynamische  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Durchlaufen einer Liste II

- ▶ Durchlauf einer Liste in **umgekehrter** Reihenfolge:
  - ▶ Referenz **fuss** verweist zwar auf das letzte Element einer Liste, kann jedoch nicht von dort zum vorletzten Element gelangen.
  - ▶ Für eine umgekehrte Ausgabe müssen alle Listenelemente gemerkt werden, während die Liste vom Anfang zum Ende durchläuft.
  - ▶ Erst nach einmaligem Durchlaufen kann vom letzten bis zum ersten Element gedruckt werden.
- ▶ Großer Aufwand?

## In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

# Durchlaufen einer Liste III

## Einsatz der rekursiven Variante

- ▶ analog zu `ZeigeListeRekursiv()`
- ▶ aber: rekursiver Aufruf und Ausgabe vertauscht

```
public void ZeigeListeUmgekehrt() {
    ZeigeListeUmgekehrt(kopf);
}

private void ZeigeListeUmgekehrt(Element aktElement) {
    if (aktElement != null) {
        ZeigeListeUmgekehrt(aktElement.HoleWeiter());
        System.out.println(aktElement.HoleWert());
    }
}
```

```
private void ZeigeListeRekursiv(Element aktElement) {
    if (aktElement != null) {
        System.out.println(aktElement.HoleWert());
        ZeigeListeRekursiv(aktElement.HoleWeiter());
    }
}
```

- Prolog
- Grundlagen
- **Listen**

# Etwas testen...

```
public class TestListe {  
    public static void main(String[] args) {  
  
        Liste meineListe = new Liste(42);  
        meineListe.OrdneEin(7);  
        meineListe.OrdneEin(73);  
        meineListe.OrdneEin(1);  
        meineListe.OrdneEin(50);  
  
        meineListe.ZeigeListeRekursiv();  
        meineListe.ZeigeListeUmgekehrt();  
    }  
}
```

```
> run TestListe  
1  
7  
42  
50  
73  
73  
50  
42  
7  
1
```

EINI LogWing /  
WiMa

## Kapitel 8

Dynamische  
Datenstrukturen

### In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**

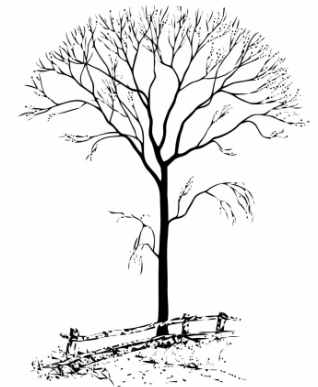


# Doppelt verkettete Listen

- ▶ Wird der Durchlauf vom Ende einer Liste zu ihrem Anfang häufig benötigt, dann ist die lineare Verkettung von vorne nach hinten nicht der ideale Navigationspfad.
- ▶ Besser wäre es dann, auch eine **Rückwärtsverkettung** zu haben.
- ▶ Aufgrund dieser Überlegung kommt man zur **doppelt verketteten Liste**.
- ▶ Die lokale Klasse **Element** enthält eine zweite Referenz **voran**, die genau entgegengesetzt zu **weiter** gerichtet ist und somit für jedes Element innerhalb der Liste auf seinen direkten Vorgänger verweist.

- Prolog
- Grundlagen
- **Listen**

- ▶ Dynamische Datenstrukturen
  - ▶ können zur Laufzeit wachsen und schrumpfen.
  - ▶ werden durch Nutzungsinformation und Verwaltungsinformation realisiert.
  - ▶ Verwaltungsinformation ist eine Referenz auf eigene Klasse.
  
- ▶ Listen
  - ▶ Einfache Listen
  - ▶ Einfache Listen mit Referenz auf das letzte Element
  - ▶ Sortierte Listen
  - ▶ Doppelt verkettete Listen
  
- ▶ Wie geht es weiter?



In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**



# Listen

---

Artikel im EINI-Wiki:

→ **Liste**

## Kapitel 8

Dynamische  
Datenstrukturen

### In diesem Kapitel:

- Prolog
- Grundlagen
- **Listen**



**Vielen Dank für Ihre Aufmerksamkeit!**

## Nächste Termine

- ▶ Nächste Vorlesung – WiMa 31.01.2019, 08:15
- ▶ Nächste Vorlesung – LogWing 01.02.2019, 08:15