

# Programmierung und Programmiersprachen Sommersemester 2006

Lehrstuhl für Angewandte Telematik / e-Business

Institut für Informatik  
Universität Leipzig  
Stiftungslehrstuhl der Deutschen Telekom AG

info@ebus.informatik.uni-leipzig.de  
www.lpz-ebusiness.de  
+49 (341) 97 323 30

## Message of the day

- Auf dem ursprünglichen PuPS Übungsblatt 03 hatte sich der Fehlerteufel eingeschlichen und die Teilübung 2.3 sinnentleert.
- Daher bitte neue (korrigierte) Version downloaden!



# Listen

(vgl. Skizzierung des Datentyps in VE2)

## Allgemeines zu Listen

- Listen definieren eine Reihenfolge von Elementen, die gemäß dieser Reihenfolge miteinander verknüpft sind.
- Typische Zugriffsmethoden:
  - Einfügen am Anfang
  - Einfügen an bestimmter Stelle
  - Anfügen (d.h. Einfügen am Ende)
  - Ermittlung der Länge
  - Prüfen auf Leere
  - Prüfen, ob Element in Liste vorkommt
  - Ermittlung der Position eines Elements
  - Ermittlung des ersten Elements
  - Liefern der Liste ohne erstes Element
- Die Verfügbarkeit aller dieser Methoden variiert mit dem Anwendungszweck. Nicht immer sind alle Methoden realisiert, aber man redet dennoch von Listen

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

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

- Deklaration einer Klasse `Element` mit zwei privaten Attributen und einem Konstruktor
- Ein Objekt vom Typ `Element` enthält als Attribute eine ganze Zahl und einen Zeiger auf ein weiteres Objekt des Typs `Element`
- Jedes Objekt vom Typ `Element` besitzt eine Referenz auf ein weiteres Element, man kann sie miteinander verketteten
- Die daraus entstehende Datenstruktur ist eine *Lineare Liste*.

```
class Liste {
    private Element kopf;

    Liste() {
        kopf = null;
    }

    Liste(int w) {
        kopf = new Element(w);
    }

    void fügeAn(int an) {
        ...
    }

    void fügeEin(int ein) {
        ...
    }
}
```

- Eine lineare Liste kann auf verschiedene Arten konstruiert werden:
  - Neues Element an den Anfang, in die Mitte oder an das Ende einer bereits bestehenden Liste anhängen
  - Zugriff auf die Liste wird durch eine Referenz realisiert, die auf das erste Element der Liste zeigt
  - Enthält eine Liste keine Elemente, zeigt die Referenz auf `null`
  - Besuchen eines Elements innerhalb der Liste erfordert eine Referenz von Element zu Element

```
void fügeAn(int neuerWert) {  
    Element lauf = this;  
    while (lauf.weiter != null)  
        lauf = lauf.weiter;  
    lauf.weiter = new Element(neuerWert);  
}
```

- Die Klasse `Element` wird um die Methode `fügeAn` ergänzt, die ein neues Element an das Ende einer Liste anhängt, die bereits aus wenigstens einem Element besteht.
- Die Liste wird durch die lokale Referenz `lauf` von Element zu Element durchlaufen, bis die Referenz `weiter` auf `null` verweist.
- Nun wird auf das neu erzeugte Objekt der Klasse `Element` verwiesen.
- Die Referenz `this` verweist immer auf das Objekt, für welches die Methode `fügeAn` aufgerufen wurde, daher verweist `this` nie auf den Wert `null`.

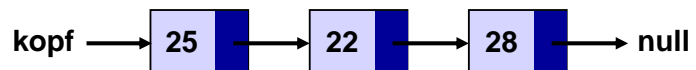
## Initiales Anlegen einer linearen Liste

- Anlegen einer linearen Liste
  - Mit dem Konstruktor von `Element` wird ein erstes Objekt geschaffen.
  - Alle weiteren Listenelemente werden durch `fügeAn` für das erste Element angefügt.

```

Element kopf;
kopf = new Element(25);
kopf.fügeAn(22);
kopf.fügeAn(28);

```



- Die Methode `fügeAn` wird für das erste Element der Liste aufgerufen.
- Jetzt wird mit `lauf` die gesamte Liste bearbeitet.

## Rekursives Anfügen

- Rekursives Vorgehen:

```

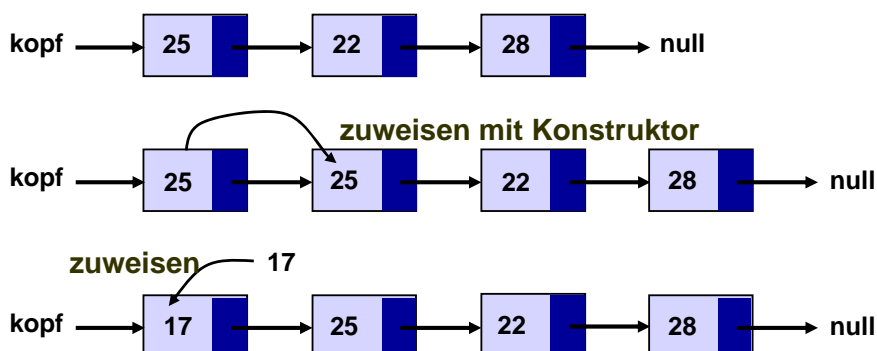
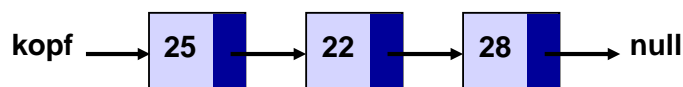
void rekFügeAn(int neuerWert) {
    if (weiter != null)
        weiter.rekFügeAn(neuerWert);
    else
        weiter = new Element(neuerWert);
}

```

- Beachte:
  - Die Ausführung der Methode `rekFügeAn` ist aufwendiger, als die bisher vorgestellten Methoden, da alle Aufrufe von `rekFügeAn` ineinander geschaltet sind und erst dann beendet werden, wenn das neue Element angefügt ist.

```
void fügeEin(int neuerWert) {  
    Element neuesElement = new Element(wert);  
    neuesElement.weiter = weiter;  
    weiter = neuesElement;  
    wert = neuerWert;  
}
```

- `kopf.fügeEin(17)` wird nun auf die Liste angewendet



- Alles andere würde bedeuten, dass **kopf** umgesetzt wird.
- Das aber würde bedeuten, dass ein Element den Zugriff auf sich abtritt.
- Das geht nicht, also bleibt nur der Weg über das Einfügen an der zunächst zweiten Stelle.

- Umsetzen von linearen Listen mit der Klasse **Element** und den Methoden **fügeAn** und **fügeEin**:
- **Aufbau einer Liste aus einer „leeren“ Liste:**
  - Da die Methoden **fügeAn** und **fügeEin** Bestandteile der Objekte der Klasse **Element** sind, können sie nur dann aufgerufen werden, wenn auch ein solches Element besteht.
  - Daher muss das erste Element einer Liste immer über einen Konstruktor erzeugt werden, alle weiteren können dann über die Methoden hinzugefügt werden.
  - Innerhalb eines Programms muss daher vor jedem Aufruf von **fügeAn** oder **fügeEin** überprüft werden, ob die Referenz auf die Liste auf **null** verweist.
  - Übersichtlicher wäre es hingegen, in allen Situationen Elemente durch die Methoden **fügeAn** und **fügeEin** hinzufügen zu können.

- **Änderung des ersten Elements:**
  - Die Implementierung der Methode **fügeEin** hat gezeigt, dass das Hinzufügen eines neuen Objektes als erstes Element der Liste nicht möglich ist; hierfür muss auf entsprechende Zuweisungen zurückgegriffen werden.
  - Ebenso problematisch ist das Löschen des ersten Elements einer Liste.
  - Insbesondere das letzte Element einer Liste lässt sich nicht durch eine Methode der Klasse **Element** entfernen.
- **Effizienz der Methode **fügeAn**:**
  - Die Methode **fügeAn** 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.

- Die Idee, eine Liste mit einer Referenz auf ihr erstes Element gleichzusetzen, wird dem Umgang mit der entstehenden Datenstruktur nicht gerecht.
- Der Wertebereich einer Klasse, die lineare Listen implementiert, sollte auch die leere Liste beinhalten und für diese eine korrekte Anwendung der Methoden garantieren.
- Die Ausführung von Methoden sollte durch zusätzliche Referenzen auf ausgewählte Elemente der Liste unterstützt werden.
- Als Ergebnis entstehen die folgenden, modifizierten Klassen `Element` und `Liste`

## Klasse `Element` (verbessert)

```
class Element {
    private int wert; private Element weiter;

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

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

    void setzeWert(int i) {
        wert = i;
    }

    int gibWert() {
        return wert;
    }

    void setzeWeiter(Element e) {
        weiter = e;
    }

    Element gibWeiter() {
        return weiter;
    }
}
```

## Klasse lineare Liste (verbessert)

```

class Liste {
    private Element kopf, fuß;
    Liste() { kopf = fuß = null; }

    Liste(int w) { kopf = fuß = new Element(w); }

    void fügeAn(int an) {
        Element neu = new Element(an);
        if (fuß != null) {
            fuß.setzeWeiter(neu);
            fuß = neu;
        } else
            kopf = fuß = neu;
    }

    void fügeEin(int ein) {
        kopf = new Element(ein, kopf);
        if (fuß == null)
            fuß = kopf;
    }
}

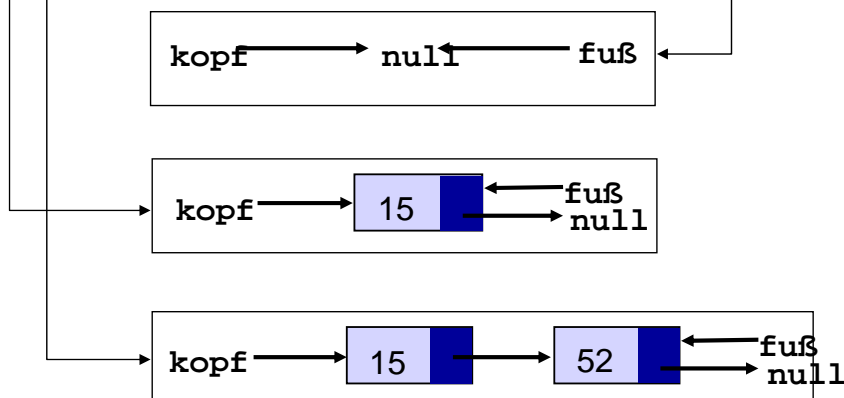
```

## Beispiel Anfügen und Einfügen

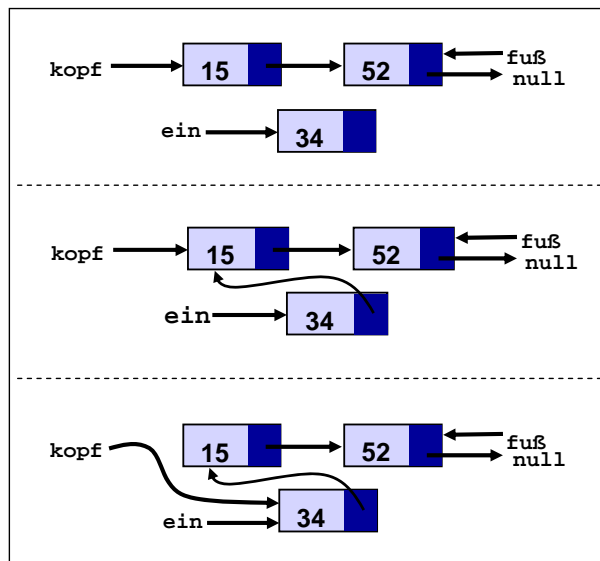
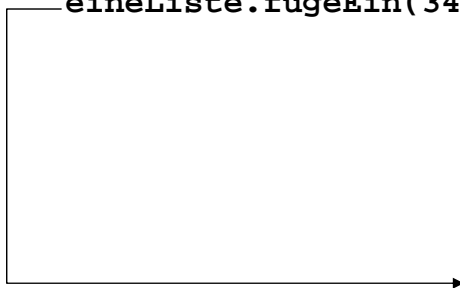
```

Liste eineListe = new Liste();
eineListe.fügeAn(15);
eineListe.fügeAn(52);
eineListe.fügeEin(34)

```



```
Liste eineListe = new Liste();  
eineListe.fügeAn(15);  
eineListe.fügeAn(52);  
eineListe.fügeEin(34)
```



## Anwendungsbeispiel

- Einordnen eines Werts in eine bereits aufsteigend geordnete Liste
- Keine zwei Elemente haben die identische Belegung des Attributs `wert`
- Der Algorithmus ist auf natürliche Weise rekursiv
- **Die Idee:**
  - Ist eine ganze Zahl  $x$  gegeben, sodass  $x$  kleiner als das erste Element der Liste ist, so füge man  $x$  am Anfang der Liste ein.
  - Muss  $x$  in der Mitte der Liste eingefügt werden, so suche man die entsprechende Position, spalte dort die Liste in einen Anfangs- und einen Endteil auf und füge  $x$  am Anfang des Listenrestes ein. Schließlich verbinde man das Ende des Anfangsteils der Liste mit der so entstandenen neuen Liste.
  - Ist  $x$  größer als das letzte Element der Liste, so bildet das neue Element allein den neuen Rest:  $x$  wird angefügt.

- **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.

- 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` beginnenden Restliste ein, indem hierfür `positioniere` mit den entsprechenden Parametern erneut aufgerufen wird.

- **Beachte:**
  - Keine doppelten Einträge werden zugelassen.
  - Die Referenz `fuß` verweist auch nach dem Einsortieren auf das letzte Element.
  - Wenn `positioniere` die leere Referenz `null` als Wert für den Parameter `anfang` übergeben bekommt, muss `fuß` korrigiert werden und auf das neu eingeordnete letzte Objekt gesetzt werden.

## Einordnen mit Hilfe von `positioniere`

```
class Liste {
    private Element kopf, fuß;

    Liste() {
        kopf = fuß = null;
    }

    Liste(int w) {
        kopf = fuß = new Element(w);
    }

    ...

    void ordneEin(int i) {
        kopf = positioniere(kopf, i);
    }
}
```

```
private Element positioniere(Element anfang, int i) {
    if (anfang == null) {
        fuß = anfang = new Element(i);
    } else {
        if (i < anfang.gibWert()) {
            anfang = new Element(i, anfang);
        }

        if (i > anfang.gibWert()) {
            anfang.setzeWeiter(
                positioniere(anfang.gibWeiter(), i));
        }
    }

    return anfang;
}
...
}
```

## Durchlaufen einer Struktur

- 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.
- Anwendungsbeispiele:
  - Prüfen auf Vorhandensein
  - Einsortieren
  - Ausdrucken
- Konkrete Ausprägungen dieses Problems spielen in der Informatik eine wichtige Rolle (z.B. Travelling Salesman Problem).

- Rekursive Methode zum Drucken einer linearen Liste:

```
class Liste {
    ...
    void rekDrucke() {
        rekDrucke(kopf);
    }

    private void rekDrucke(Element aktuell) {
        if (aktuell != null) {
            System.out.println(aktuell.gibWert());
            rekDrucke(aktuell.gibWeiter());
        }
    }
    ...
}
```

- Iterative Methode zum Drucken einer linearen Liste:

```
void iterDrucke() {
    Element aktuell = kopf;
    while (aktuell != null) {
        System.out.println(aktuell.gibWert());
        aktuell = aktuell.gibWeiter();
    }
}
```

- 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.

- Ausgabe einer Liste in umgekehrter Reihenfolge:

```
void reversivDrucke() {
    reversivDrucke(kopf);
}

private void reversivDrucke(Element aktuell) {
    if (aktuell != null) {
        reversivDrucke(aktuell.gibWeiter());
        System.out.println(aktuell.gibWert());
    }
}
```

- Ist 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.
- Auf Grund dieser Überlegung kommt man zu doppelt verketteten Liste (einmal von vorne nach hinten, einmal umgekehrt).

- 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.
- Mit doppelt verketteten Listen kann in beide Richtungen einer Liste navigiert werden und deshalb komplexe Operationen auf einer Liste unterstützt werden.
- In Java: **LinkedList** (Unterklasse von **AbstractSequentialList** in `java.util`)

```
class DElement {
    ...
    // bekannte Deklarationen der linearen Liste

    private DElement voran, weiter;
    void setzeVoran(DElement e) {
        voran = e;
    }

    Element gibVoran() {
        return voran;
    }
}
```

```
class DListe {
    ...
    // bekannte Deklarationen der linearen Liste

    void fügeAn(int an) {
        DElement neu;
        neu = new DElement(an);
        if (fuß != null) {
            fuß.setzeWeiter(neu);
            neu.setzeVoran(fuß);
            fuß = neu;
        } else {
            kopf = fuß = neu;
        }
    }

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

```
private DElement positioniere(DElement anfang, int i) {
    if (anfang == null) {
        anfang = new DElement(i);
        anfang.setzeVoran(fuß);
        fuß = anfang;
    } else {
        if (i < anfang.gibWert()) {
            DElement neu = new DElement(i, anfang);
            neu.setzeVoran(anfang.gibVoran());
            anfang.setzeVoran(neu);
            if (neu.gibVoran() != null)
                neu.gibVoran().setzeWeiter(neu);
            anfang = neu;
        }
        if (i > anfang.gibWert())
            anfang.setzeWeiter(
                positioniere(anfang.gibWeiter(), i));
    }
    return anfang;
}
```

```
void reversivDrucke() {
    DElement aktuell = fuß;
    while (aktuell != null) {
        System.out.println(aktuell.gibWert());
        aktuell = aktuell.gibVoran();
    }
}
```