Einfach und mehrfach verkettete Listen

- Sie haben ein intuitives Verständnis vom Landau gross O-Begriff
- Sie wissen, was die einfach und mehrfach verkettete Listen sind
- Sie kennen die wichtigsten Operationen auf Listen und wissen wie die Operationen definiert sind
- Sie kennen das Konzept der Iteratoren und deren Implemention in Java
- Sie kennen die speziellen Listen: doppelt verkettet, zirkulär und sortiert
- Sie können mit den Java Collection Klassen umgehen

Aufwand, gross O Notation

Komplexität in Raum und Zeit

Beispiel

- c = Math.min(a,b);
- while ((a % c != 0) || (b % c != 0)) c--;

Welche Laufzeit hat das Programm?

- Hängt stark von Maschine, Programmiersprache und Daten ab.

Fragen:

- Welche Zeit wird der Algorithmus benötigen? Zeitkomplexität
- Wie viel Speicher wird der Algorithmus benötigen? Speicherkomplexität

nicht absolute Zahlen, sondern als Funktion von Grösse oder Anzahl Werten: n

- z.B. die Zeit die der obigen Algorithmus benötigt, wächst linear mit den Werten von a,b
  - Aufwand: O(n), aber Euklid nur $O(\log_2 n)$

O-Notation Definition

Definition: $T(n) = O(g(n)) \mid n \rightarrow \infty$

$T(n) = O(g(n))$	$\Leftrightarrow$	$\exists$ c, n_0 positive Konstanten: $\forall n \geq n_0$: $T(n) \leq c*g(n)$
	$\Leftrightarrow$	es existieren positive Konstanten c und n_0 sodass für alle $n \geq n_0$ gilt: $T(n) \leq c*g(n)$.

Bedeutung:

- Für genügend grosse n wächst T höchstens so schnell wie g.

Bemerkungen

- g ist eine "asymptotische obere Schranke" für T.
- genaue Laufzeit-Funktion T wird grob nach oben abgeschätzt durch einfachere Funktion g.
- Beispiel: für n doppelt so gross ist, braucht der Algorithmus doppelt so lange -> O(n)

… O-Notation Definition

Die Definition der **O**-Notation besagt, dass, wenn $\mathbf{T(n) = O(g(n))}$, ab irgendeinem $\mathbf{n_0}$ die Gleichung $\mathbf{T(n) \leq c \cdot g(n)}$ gilt.

Weil **T(n)** und **g(n)** Zeitfunktionen sind, ihre Werte also immer positiv sind, gilt:

$$\frac{\mathbf{T(n)}}{\mathbf{g(n)}} \leq \mathbf{c} \text{ ab irgendeinem } \mathbf{n_0}$$

Beide Funktionen können besser verglichen werden, wenn man den Grenzwert berechnet.

$$\lim_{n \to \infty} \frac{T(n)}{g(n)}$$

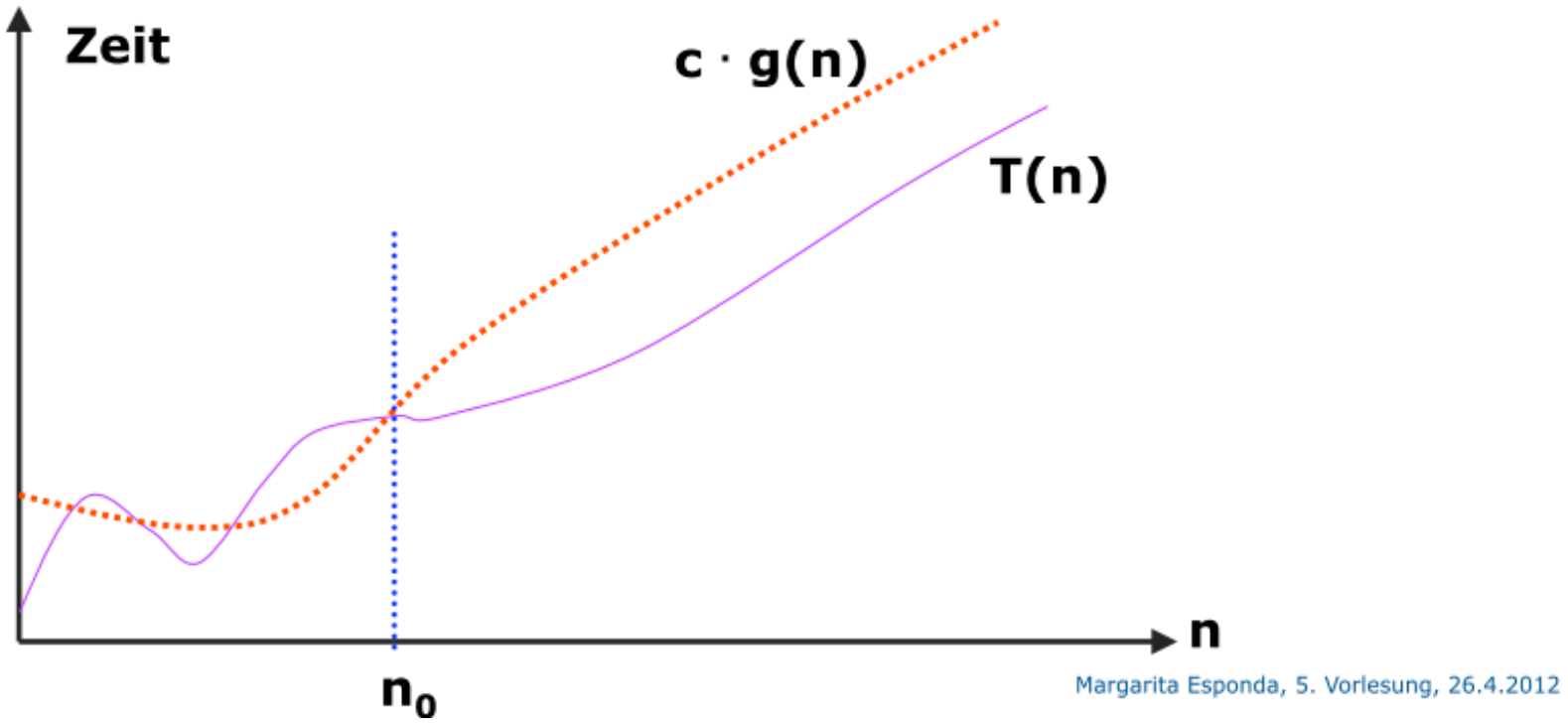
- Wenn der Grenzwert existiert, dann gilt: $\mathbf{T(n) = O(g(n))}$
- Wenn der Grenzwert gleich **0** ist, dann bedeutet dies, dass **g(n)** sogar schneller wächst als **T(n)**. Dann wäre **g(n)** eine zu große Abschätzung der Laufzeit.

Margarita Esponda, 5. Vorlesung, 26.4.2012

… O-Notation Definition

Definition:

Die Funktion $\mathbf{T(n) = O(g(n))}$, wenn es positive Konstanten **c** und $\mathbf{n_0}$ gibt, so dass $\mathbf{T(n) \leq c \cdot g(n)}$ für alle $\mathbf{n \geq n_0}$



Margarita Esponda, 5. Vorlesung, 26.4.2012

Begründung: Asymptotische Laufzeit

- Laufzeit eines Algorithmus ist besonders für grosse Eingaben interessant
- Nicht die exakte Laufzeit interessiert, sondern die Grössenordnung
- Ziel: eine Funktion f: N->N finden, sodass f(n) = a bedeutet: "Eingabe der Grösse n hat Aufwand a"
- Aufwand ist normalerweise Rechenzeit oder Speicherbedarf

Beispiel

daneben gibt es noch besten und schlechtesten Fall

Gezählt werden Anzahl Divisionen im *durchschnittlichen Fall*

n,m	ggT Linear*	Euklid*
∈ [1...10]	5	3.3
∈ [1...100]	50	6.6
∈ [1 ...1000]	500	9.9
∈ [1 ... 10000]	5000	13.3
∈ [1 ...10'000'000]	5'000'000	23.3

* Werte nur grob angenähert: Bestimmung genauerer Werte ist ein komplexes zahlentheoretisches Problem

→ ev. Bezug zur Riemannschen Vermutung

$$\zeta(s) = \prod_{p \text{ prim}} \frac{1}{1 - p^{-s}},$$

Wichtige Komplexitätsklassen

- O(1) konstanter Aufwand
- O(log n) logarithmischer Aufwand
- O(n) linearer Aufwand
- O(n * log n)
- $O(n^2)$ quadratischer Aufwand
- $O(n^k)$ für konstantes k > 1 polynomialer Aufwand
- $O(2^n)$ exponentieller Aufwand

Anzahl Operationen

$f(n)$	$n = 2$	$2^4 = 16$	$2^8 = 256$	$2^{10} = 1024$	$2^{20} = 1048576$
$\mathrm{ld}n$	1	4	8	10	20
n	2	16	256	1024	1048576
$n \cdot \mathrm{ld}n$	2	64	1808	10240	20971520
n^2	4	256	65536	1048576	$\approx 10^{12}$
n^3	8	4096	16777200	$\approx 10^9$	$\approx 10^{18}$
2^n	4	65536	$\approx 10^{77}$	$\approx 10^{308}$	$\approx 10^{315653}$

Beispiele von Algorithmen

- Einfache Anweisungssequenz: Aufwand ist konstant

  $s_1; s_2; s_3; s_4;...;s_k$

  - O(1)

- Einfache Schleifen: Aufwand steigt Linear

```
for(int i = 0; i< n; i++) s;
```

  - wenn: s ist O(1)
  - Zeitbedarf ist O(n)

- Geschachtelte Schleifen: Aufwand steigt quadratisch

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) s;
}
```

  - Zeitbedarf ist O(n) ⊗ O(n) also $O(n^2)$

Rechnenregeln der O-Notation

Die **O**-Notation betont die dominante Größe

Beispiel: Größter Exponent

$$3n^3 + n^2 + 1000n + 500 = \mathbf{O}(n^3)$$

Ignoriert Proportionalitätskonstante

Ignoriert Teile der Funktion mit kleinerer Ordnung

Beispiel: $5n^2 + \log_2(n) = \mathbf{O}(n^2)$

Teilaufgaben des Algorithmus mit kleinem Umfang

Margarita Esponda, 5. Vorlesung, 26.4.2012

Rechnenregeln der O-Notation

- Konstanten können ignoriert werden
  - O(k*f) = O(f) | falls k > 0
- Grössere Exponenten wachsen schneller
  - $O(n^r) < O(n^s)$ | falls n > 1 und 0 <= r <= s
    - $k_1 n^3 + k_2 n^2 + n$
- der am schnellsten wachsende Term dominiert die Summe
  - O(f + g) = O(g) | falls g schneller wächst als f
  - Bsp: $O(a*n^4 + b*n^3) = O(n^4)$
    - *bei Polynomen: nur Term mit grösstem Exponenten zählt*
- Verknüpfung resp. Verschachtelung von Funktionen
  - $O(f \otimes g) = O(f) \otimes O(g)$

Rechnenregeln der O-Notation

- Bei Logarithmus-Funktionen spielt weder Basis noch Multiplikationsfaktor eine Rolle
  - $O(\log_a k * n) = O(\log_b n)$

- Exponential Funktionen wachsen *schneller* als sämtliche Polynome

  → Polynom-Anteil fällt bei Summe weg
  - Bsp $O(1.00001^n + n*1000000) = \mathbf{O(1.00001^n)}$

- Logarithmische Funktionen wachsen *langsamer* als alle Polynome

  → Log-Anteil fällt bei Summe mit Polynom weg
  - Bsp: $O(n^{1.00001} + 1000000*\log(n)) = \mathbf{O(n^{1.00001})}$

Übung Bestimmen Sie die Ordnung

- $f = 2^{n*2} + 3n + 5$
- $f = n^{1.00001} + 1000 \log(n)$
- $f = 1.00001^{n} + x * 1000\, n$
- $f = n^{-1} + n$
- $f = n\,(n-1) / 2$

Weitere Beispiele von Algorithmen

- Schleifen Index verändert sich nicht linear

```
h = 1;
while (h <= n) {
   h = 2 * h;
}
```

- h nimmt werte 1,2,4,8,… an
- es sind 1 + $\log_2 n$ Durchläufe

- Komplexität ist also O(log n)

Schleifen Indizes hängen voneinander ab

```
for (j = 0; j <n; j ++) {
    for (k = 0; k < j; k++) {
        s
    }
}
```

- die innere Schleife wird 1,2,3,4, ..n mal durchlaufen
- Zeitbedarf: $\sim \frac{n\,(n+1)}{2}$
- Also $O(n^2)$

Übung Bestimmen Sie die Ordnung

a)
```
int n = K; // was passiert wenn n eine Fliesskommazahl ist?
   while (n > 0) {
        n = n / 3;
  }
```

b)
```
for (int i = 0; i < n; i ++) {
        for (int j = i; j < n; j++) {
                foo(i,j);
        }
   }
```

c)
```
for (int i = 0; i < n; i ++) {
        for (int j = n; j > 0; j = j/2) {
                foo(j);
        }
   }
```

Liste, Listnode, Iterator

Listen

- Abstrakter Datentyp, der eine Liste von Objekten verwaltet.

- Liste ist eine der grundlegenden Datenstrukturen in der Informatik.

- Wir können mit Hilfe der Liste einen Stack implementieren.
  Schnittstelle: `java.util.List`
  Impl.: `java.util.LinkedList`

- Speichert Object oder Wert durch Typenplatzhalter bestimmt (i.e. generisch)

Minimale Operationen

Funktionskopf	Beschreibung
`void add (Object x)`	Fügt x am Schluss der Liste an
`void add (int pos, Object x)`	Fügt x an der pos in die Liste ein
`Object get(int pos)`	Gibt Element an pos zurück Vorsicht ev: O(n)
`Object remove(int pos)`	Entfernt das pos Element und gibt es als Rückgabewert zurück
`int size()`	Gibt Anzahl Element zurück
`boolean isEmpty()`	Gibt true zurück, falls die Liste leer

Wo werden Listen angewendet?

School of Engineering

Als universelle (Hilfs-)Datenstruktur

- Zur Implementierung von Stack, Queue, etc.

Eigenschaften

- Anzahl der Elemente zur Erstellungszeit unbekannt (sonst meist Array)
- Reihenfolge/Position ist relevant
- Einfügen und Löschen von Elementen ist unterstützt

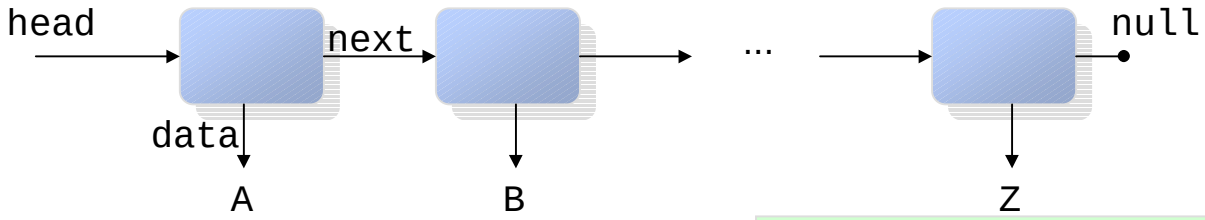
Speicherverwaltung

- Liste der belegten/freien Memoryblöcke

Betriebssysteme

- Disk-Blöcke, Prozesse, Threads, etc

Datenstruktur des Listenknotens




```
class ListNode
{
   Object data;
   ListNode next;
   ListNode(Object o) {
     data = o;
  }
}
```

Daten der Liste

Referenz auf nächsten ListNode

```
public class LinkedList implements List
{
      private ListNode head;

      void add (int pos, Object o){
      …
      }

      void add (Object o) {
      …
      }

      Object remove(int pos) {
      …
      }

}
```

Listen Element an Position: get

- Suche das Listen Element an der vorgegebenen Position

```
Object get (int pos) {
   ListNode node = this.head;
   while (pos > 0) {
      node = node.next; pos--;
   }
   return node.data;
}
```

Einfügen in eine Liste: add

- Am Schluss der Liste anhängen

```
void add (Object o) {
   if (head == null)
     add(0,o);
   else {
     ListNode n = new ListNode(o);
     ListNode f = head;
     while(f.next != null) f=f.next;
     f.next = n;
   }
```

- Am Anfang der Liste einfügen

```
void add (int pos, Object o){
 if (pos == 0) {
     ListNode n = new ListNode(o);
     n.next = head;
     head = n;
   }
  else {…}
}
```

Entfernen eines Listen Element: remove

- Entfernen eines Elements am Anfang der Liste

```
void remove (int pos){
    if (pos == 0) head = head.next;
    else {…}

}
```

- **Übung**: Schreiben Sie die Methoden remove, die ein Element am Schluss entfernt

```
void remove (int pos) {
     if (pos == size()-1) {

     }
}
```

Traversierung von Listen

- Ausgabe aller Listenelemente

```
List list = new LinkedList();
// ... (put a lot of data into the list)

// print every element of linked list
for (int i = 0; i < list.size(); i++) {
    String element = (String)list.get(i);
    System.out.println(i + ": " + element);
}
```

- Dieser Code ist sehr ineffizient, wenn die Listen gross sind
- Frage: wieso?

Problem der Position

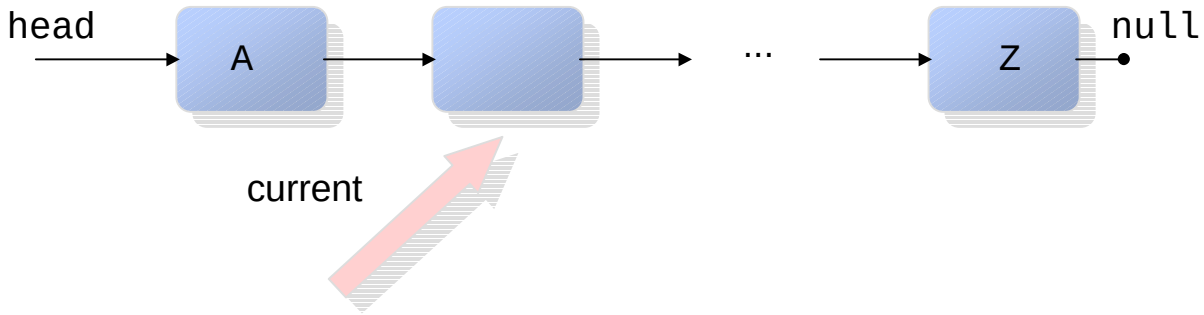
- Zum Bestimmen des i-ten Elements muss (intern) die im Schnitt die halbe Liste durchlaufen werden -> O(n)

- Besser wäre es sich die Position zu merken

- Es gibt in Java ein spezielles Objekt dafür: den Iterator

- Allgemein: spricht man auch vom Iterator-Entwurfsmuster

Das Konzept des Iterators



- der Iterator ist ein ADT, mit dem eine Datenstruktur, z.B. Liste, traversiert werden kann, ohne dass die Datenstruktur bekannt gemacht werden muss: *Information Hiding*.
- es wird ein privater (current) Zeiger auf die aktuelle Position geführt.
- Der Iterator wird im **`for (Object o : List)`** Konstrukt ebenfalls erzeugt aber versteckt

```
interface Iterator {
   boolean hasNext();
   Object next();
   void remove();
}
```

es hat noch weitere Objekte
liefere nächstes Objekt

ListIterator und Listen

```
class ListIterator implements Iterator {
   boolean hasNext();
   Object next();
   remove();

   add(Object o);


}
```

es hat noch weitere Objekte

liefere nächstes Objekt

lösche das Element, das zurückgegebenen wurde

fügt Objekt in die Liste ein, **vor** dem Element, das beim nächsten next Aufruf zurückgegeben wird.

```
class LinkedList  {
   ....
   ListIterator iterator()

   ....
}
```

liefert Iterator auf den Anfang der Liste

Iterator Verwendungsmuster

- Gehe durch die Liste durch und gebe Position und Wert aus

```
Iterator itr = list.iterator();
for (int i = 0; itr.hasNext(); i++) {
    Object element = itr.next();
    System.out.println(i + ": " + element);
}
```

- Das allgemeine Verwendungsmuster sieht folgendermassen aus

```
Iterator itr = list.iterator();
while (itr.hasNext()) {
    Object element = itr.next();
    <do something with element >;
}
```

- In foreach-Schleife versteckt

```
for ( Object element : list) {
    <do something with element >;
}
```

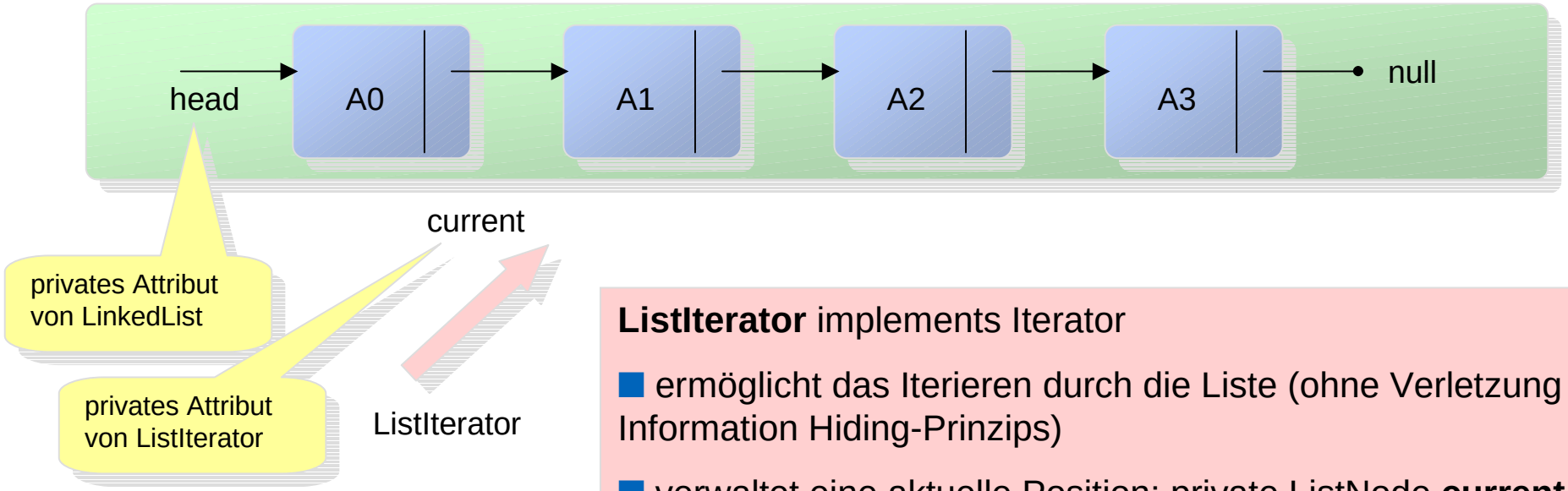
Zusammenfassung einfache Liste

Listen-Element: ListNode (privat)

- Behälter für die Objekte (Zeiger)
- Zeiger auf das nächste Element.

LinkedList implements List

definiert Operationen auf Listen wie z.B. das Einfügen, den Zugriff usw.
Zeiger auf Anfang der Liste



ListIterator implements Iterator

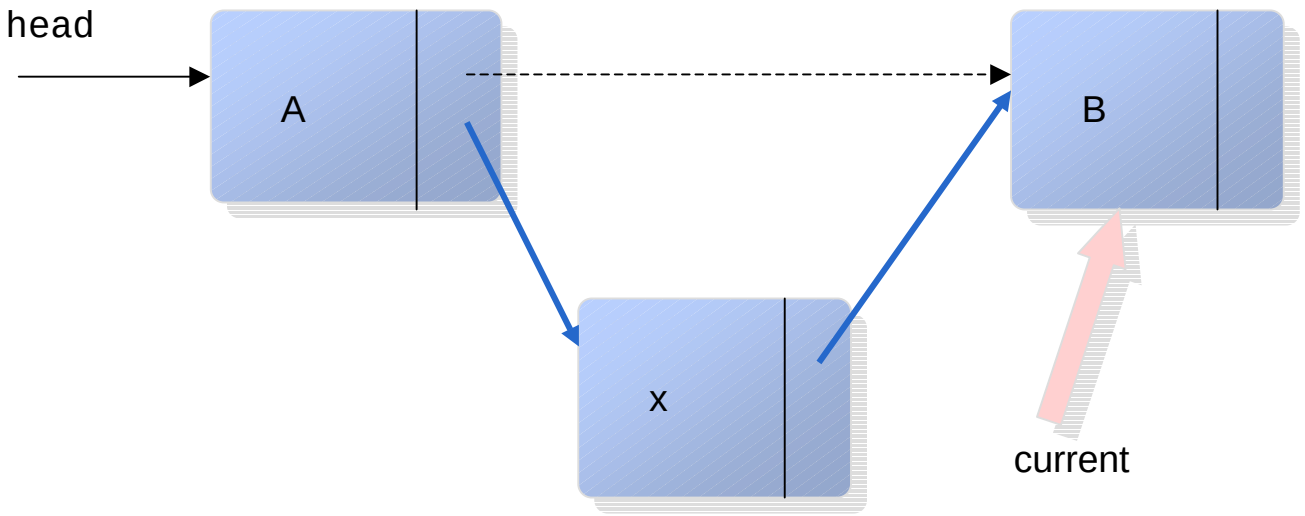
- ermöglicht das Iterieren durch die Liste (ohne Verletzung des Information Hiding-Prinzips)
- verwaltet eine aktuelle Position: private ListNode **current**
- gleichzeitig mehrere Iteratoren auf die gleiche Liste ansetzbar.

Weitere Operationen

Allgemeines Einfügen in eine Liste: add

Operationen

Methode	Beschreibung
`void add (A)`	fügt x vor `current` ein



Einfügen:

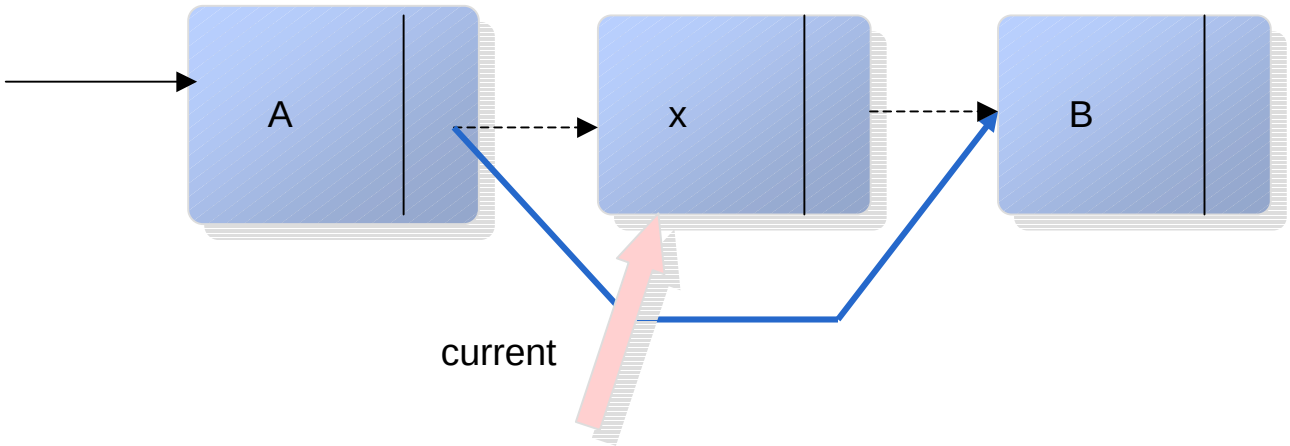
ein Element wird zwischen zwei Elemente eingefügt

Übung: Schreiben Sie die Methode `add`, so dass das neue Element vor dem `current` eingefügt wird

Allgemeines Löschen eines Objekts: remove

Operationen

Methode	Beschreibung
`void remove`	löscht Element auf das `current` zeigt

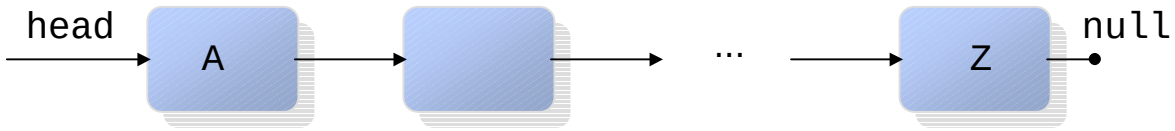


Löschen:

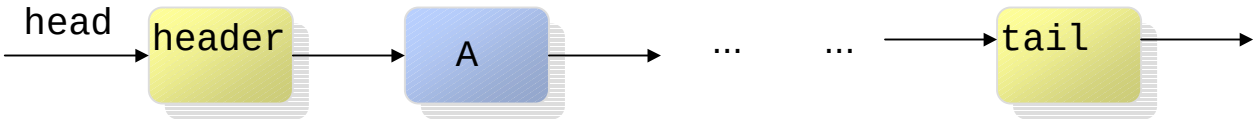
das zu löschende Element wird "umgangen"

Frage: wie findet man das Vorgänger-Objekt ?

Mitte, Anfang und Ende der Liste



- Anfang der Liste: wird meist als `head` oder root bezeichnetes
- Ende der Liste: `next` zeigt auf `null`
- Operationen müssen unterschiedlich implementiert werden, je nachdem ob sie in der Mitte, am Anfang oder am Ende der Liste angewendet werden.
- Zur Vereinfachung definiert man deshalb oft einen leeren sog. **`Anfangsknoten`** oder **`Header Node`** und einen sog. **`Schwanzknoten`** oder **`Tail Node`**



- das erste und letzte Element sind somit keine Spezialfälle mehr
  - jedes Element hat einen Vorgänger und einen Nachfolger

Doppelt verkettete Listen

Doppelt verkettete Listen

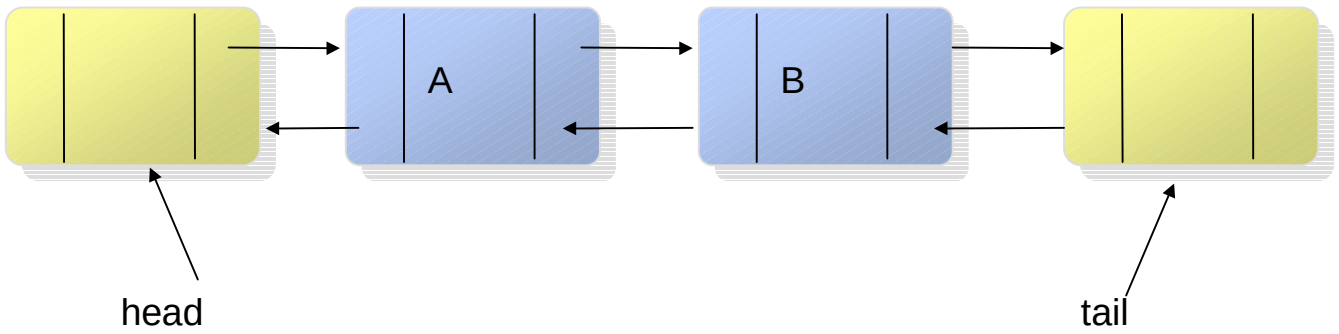
Folgende Probleme treten bei einfach verkettete Listen auf:

- Der Zugang zu Elementen in der Nähe dem Listenende kostet viel Zeit (im Vergleich mit einem Zugriff auf den Listenanfang)
- Man kann sich mit next() nur in einer Richtung effizient durch die Liste “hangeln”, die Bewegung in die andere Richtung ist ineffizient.

```
class ListNode
{
   Object data;
   ListNode next,prev;
}
```

symmetrisch aufbauen:

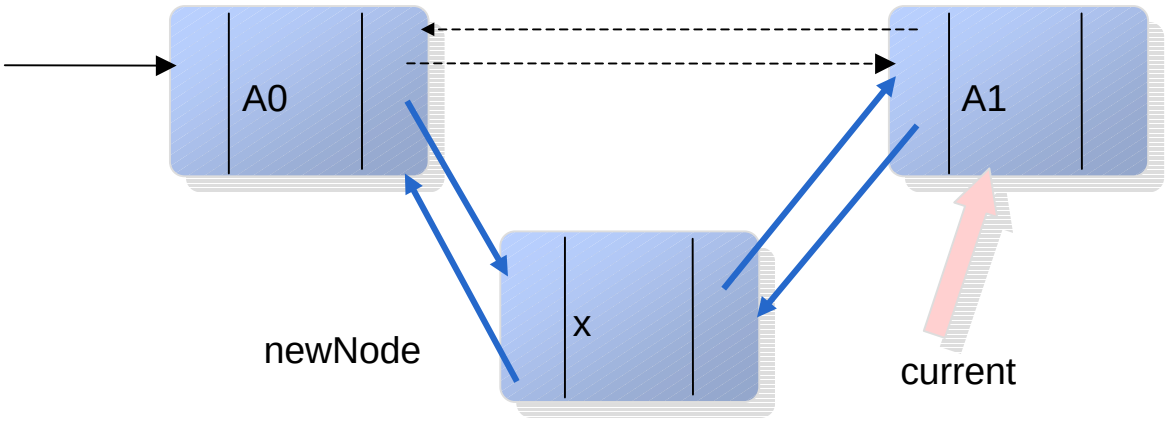
jeder Knoten hat zwei Referenzen next und previous



Add bei doppelt verketteten Listen

Operation

Methode	Beschreibung
`void add (x)`	fügt `x` ein



Nachteil (gegenüber einfach verkettet): mehr Anweisungen

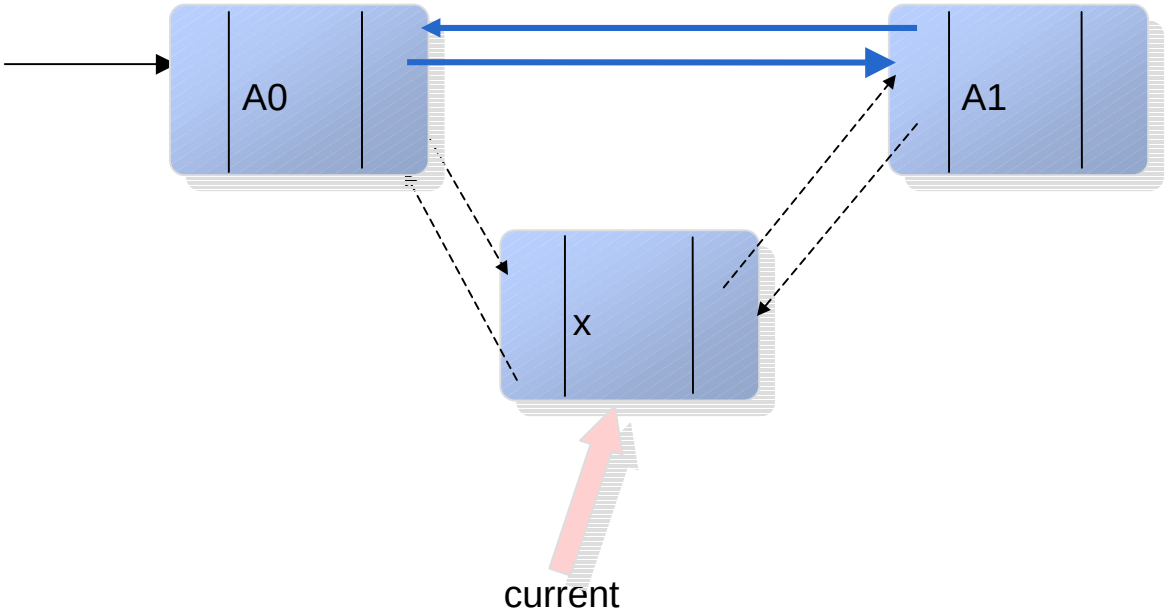
Vorteil:
add-Operation ist an jeder Stelle einfach möglich

```
newNode.next = current;
newNode.prev = current.prev;
current.prev.next = newNode;
current.prev = newNode;
```

Remove bei doppelt verketteten Listen

Operation

Methode	Beschreibung
`void remove (x)`	löscht `x`



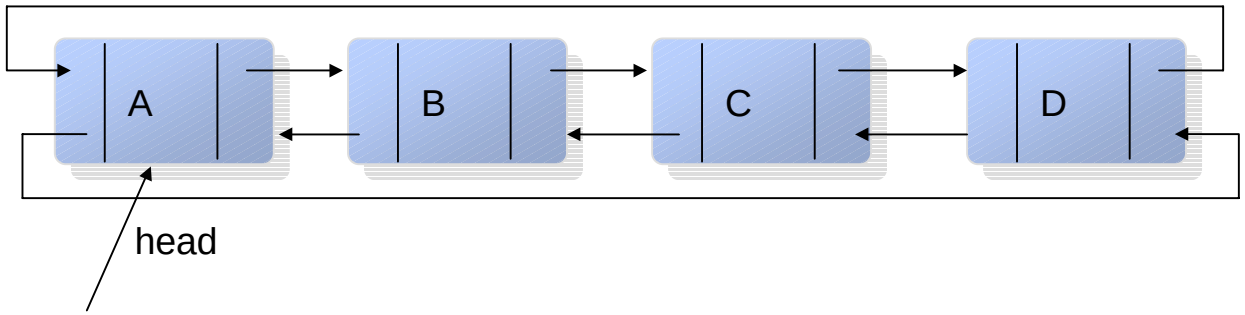

```
current.prev.next = current.next;
current.next.prev = current.prev;
```

Vorteil:

Remove-Operation ist nun sehr einfach

Zirkuläre doppelt verkettete Listen

Head und Tail Knoten wurden eingeführt, um sicherzustellen, dass jeder Knoten einen Vorgänger und Nachfolger hat.



Idee: wieso nicht einfach das erste Element wieder auf das letzte zeigen lassen

-> zirkulär verkettete Liste

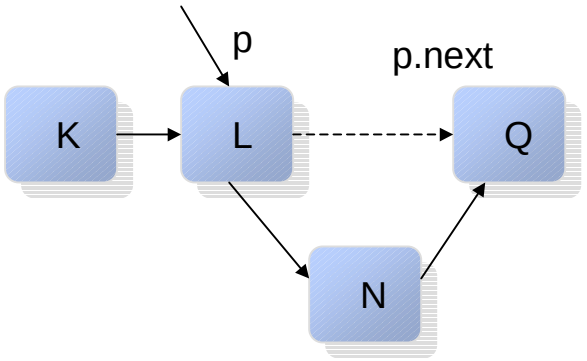
einzig die leere Liste zum Spezialfall (wird separat behandelt)

Sortierte Listen

Sortierte Listen

- Wie der Name sagt: *Die Elemente in der Liste sind (immer) sortier*t.
- Hauptunterschied:
  - **insert()** Methode fügt die Elemente *sortiert* in die Liste ein.
- Implementation
  - suche vor dem Einfügen die richtige Position

```
while (p.next.data < n.data) p = p.next;
```



- Anwendung:
  - überall wo sortierte Datenbestände verwendet werden, z.B. PriorityQueue

Sortierte Listen - Comparable Interface

- Problem: Wie vergleicht man Objekte miteinander?

  ⇒ Es muss etwas geben, das eine Beurteilung von 2 Elementen bezüglich `>, =, <` ermöglicht …

- Lösung: Das Interface `java.lang.Comparable` ist vorgesehen, zum Bestimmen der relativen (natürlichen) Reihenfolge von Objekten.

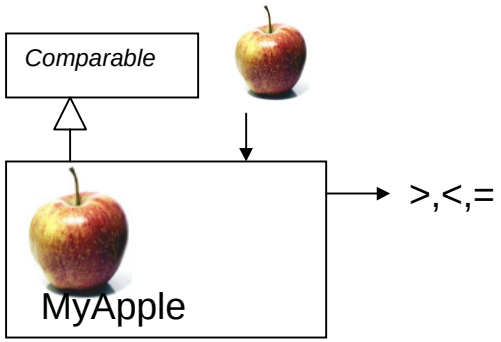
Bsp: `x.compareTo(y)`

// if x < y ⇒ negative Zahl

// if x = y ⇒ 0

// if x > y ⇒ positive Zahl

```
public interface Comparable {
   int compareTo(Object o);
}
```



Beispiel Comparable

```
class MyApple implements Comparable {
   int value;

   int compareTo(Object o) {
       Apple a= (Apple)o;
       if (this.value < a.value) return -1;
       else if (this.value > a.value) return 1;
       else return 0;
   }
}
```

return this.value - ((Apple)o).value; falls es nicht zu Werteüberlauf kommen kann.
Sonst Integer.compare(this.value,a.value).

Sortierte Listen

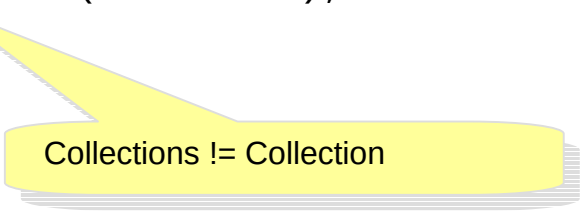
- Bei geordneten Listen muss eine Ordnung bezüglich der Elemente definiert sein.

- Das Interface **java.lang.Comparable** wird von folgenden Klassen implementiert:

```
Byte, Character, Double, File, Float, Long,
ObjectStreamField, Short, String, Integer, BigInteger,
BigDecimal, Date
```

Lösung 1: Listen, die aus Objekten bestehen, welche dieses Interface implementieren, können mit **Collections.sort** (statische Methode) automatisch sortiert werden.

```
Collections.sort(List list);
```

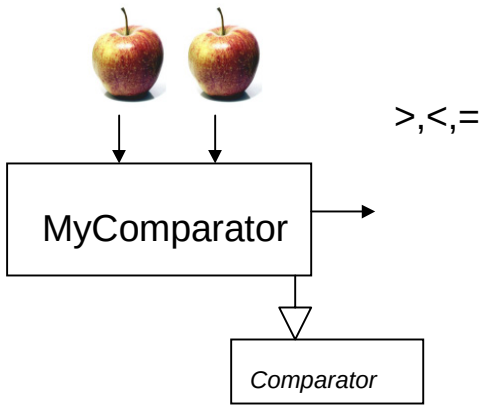


Sortierte Listen

- Was macht man, wenn nach anderem Kriterium verglichen werden soll

- **Lösung 2**: Das Interface ***java.util.Comparator*** ist vorgesehen für Objekte wenn Objekte nach unterschiedlichen Kriterien sortiert werden sollen
- Es können sogar unterschiedliche Objekte sein, solange sie vergleichbar sind.
- z.B.Anzahl Würmer

```
public interface Comparator {
      public int compare(Object o1, Object o2);
}
```



- Ein Comparator-Objekt (Objekt von Klasse welche Comparator implementiert) wird beim Methodenaufruf übergeben

```
      Collections.sort(List list, Comparator comp);
```

Beispiel Comparator

- Verwendet man den Raw Type* Comparator (Typ nicht durch Typenplatzhalter bestimmt) dann kann mittels einem Comparator ein gemeinsames Kriterium zweier beliebiger Objekte verglichen werden (wird aber eher selten verwendet).

```
class MyComparator implements Comparator {
```

```
   int compare(Object o1, Object o2) {
        Apple a= (Apple)o1;
        Pear p = (Pear)o2);
        if (a.value < p.value) return -1;
        else if (a.value > p.value) return 1;
        else return 0;
   }
}
```

* wird in der Generic Vorlesung noch genauer erklärt

Arrays und Listen

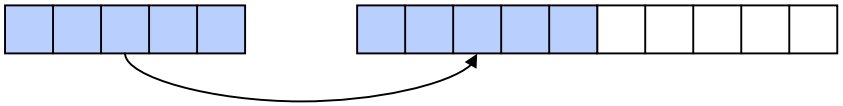
Vergleich Liste und Array

- **Array**: Teil der Java Sprache
  - Benutzung sehr einfach
  - alle eingebauten Typen und beliebige Objekte
  - **Anzahl Elemente muss zur Erstellungszeit bekannt sein**: `new A[10];`
  - Operationen:
    - *Indizierter Zugriff sehr **effizient**: a[i]*
    - *Anfügen von Elementen, Ersetzen und Vertauschen von Elementen*
    - *Einfügen und Löschen mit Kopieren verbunden: **ineffizient***
- **Liste**: Klassen in Java Bibliothek: LinkedList
  - nicht ganz so einfach in der Benutzung
  - nur Referenztypen können in Listen verwaltet werden
  - **Anzahl Elemente zur Erstellungszeit nicht bekannt** : `new LinkedList();`
  - Operationen:
    - *Indizierter Zugriff möglich aber **ineffizient**: `list.get(i)`*
    - *Anfügen, Ersetzen, Vertauschen und **Einfügen** und **Löschen** von Elementen*

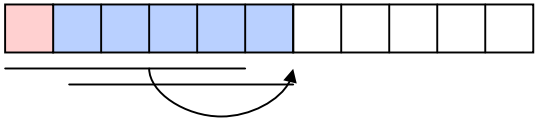
Array Implementation der Liste

- Wenn mehr Elemente gespeichert werden sollen als im Array Platz haben, muss ein neuer Array erstellt werden und es müssen die Elemente umkopiert werden (gute Strategie Länge * 2).

```
aNeu = new Object[a.length *2];
System.arraycopy(a,0,aNeu,0,a.length,);
a = aNeu;
```



- Beim Einfügen am Anfang der Liste müssen alle nachfolgenden Elemente im Array umkopiert werden
- `System.arraycopy(a,0,a,1,a.length-1);`



- Array Implementation der Liste: **`java.util.ArrayList`**
- langsam bei Einfügen und Löschen
- schneller bei Zugriff auf beliebiges Element

… Array Implementation der Liste, Vergleich

School of Engineering

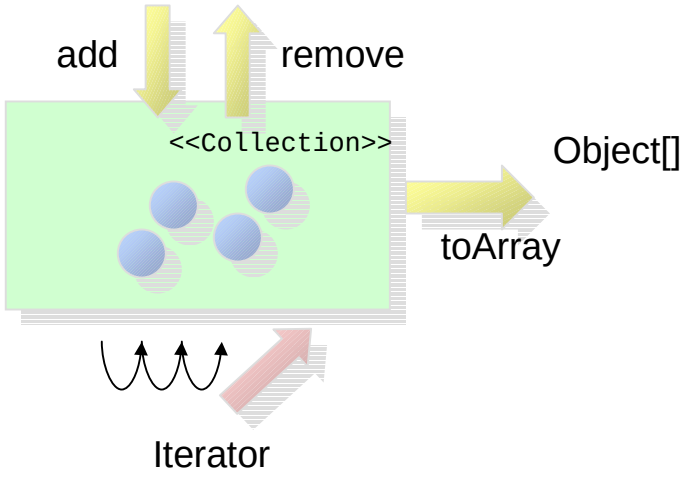
- LinkedList
  - schneller für Mutationen, langsam bei direktem Zugriff
  - non-synchronized Aufrufe
- ArrayList
  - Implementation als Array
  - -> direkter Zugriff schnell, Mutationen langsam
  - non-synchronized Aufrufe
- Vector deprecated
  - ab JDK 1.0 vorhanden, alt
  - z.T. redundante Methoden
  - relativ langsam
  - synchronized Aufrufe

Das java.util.Collection Interface

- Gemeinsames Interface für Sammlungen (Collections) von Objekten - Ausnahme Array - leider.

Abstrakter Datentyp für beliebige Objektbehälter.

Die add Methode fügt ein Element an der "natürlichen" Position hinzu.



Operationen

Funktionskopf	Beschreibung
`void add(Object x)`	Fügt x hinzu
`boolean remove (Object x)`	löscht das Element x
`void removeAll()`	löscht ganze Collection
`Object[] toArray()`	wandelt Collection in Array um
`Iterator iterator()`	gibt Iterator auf Collection zurück
`int size()`	Gibt Anzahl Element zurück
`boolean isEmpty()`	Gibt true zurück, falls die Collection leer ist

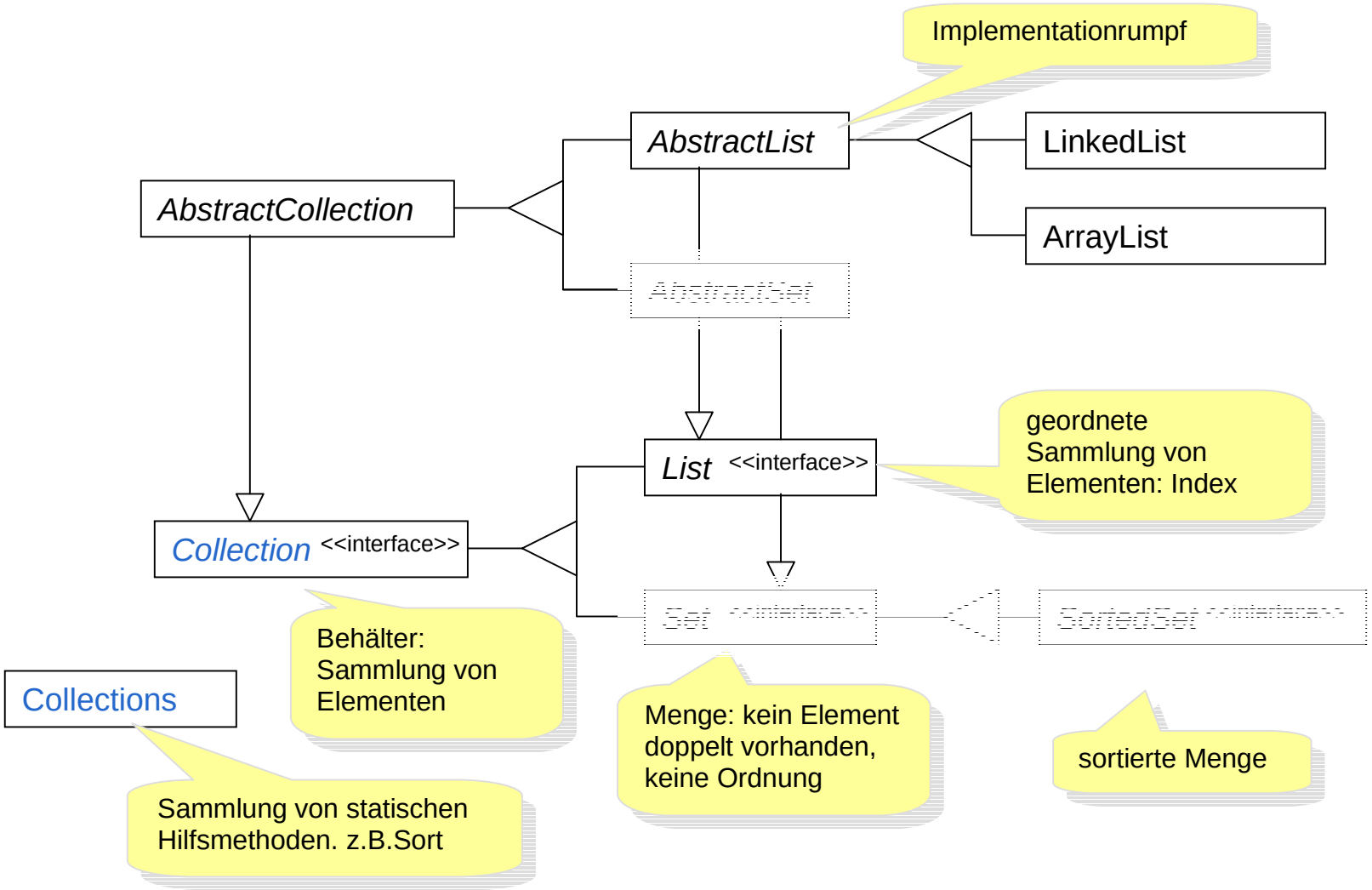
Die java.util.Collections class

School of Engineering

- Folgende Statische Methoden von `Collections` können angewendet werden
  - Example:
    ```
    Collections.replaceAll(list, "hello", "goodbye");
    ```

Method name	Description
`binarySearch(`*list*`, `*value*`)`	searches a sorted list for a value and returns its index
`copy(`*dest*`, `*source*`)`	copies all elements from one list to another
`fill(`*list*`, `*value*`)`	replaces all values in the list with the given value
`max(`*list*`)`	returns largest value in the list
`min(`*list*`)`	returns smallest value in the list
`replaceAll(`*list*`, `*oldValue*`, `*newValue*`)`	replaces all occurrences of *oldValue* with *newValue*
`reverse(`*list*`)`	reverses the order of elements in the list
`rotate(`*list*`, `*distance*`)`	shifts every element's index by the given distance
`sort(`*list*`)`	places the list's elements into natural sorted order
`swap(`*list*`, `*index1*`, `*index2*`)`	switches element values at the given two indexes

Collection Klassen im JDK



Wrapper Klassen

Thread-Safety und Synchronized

Problem

- wenn mehrere Threads gleichzeitig z.B. gleiches Element entfernen passiert ein Unglück, i.e. Listen-Datenstruktur können inkonsistent werden
- Thread-Safe
  - mehrere Threads können gleichzeitig auf z.B. remove Methode zugreifen
  - in Java einfach mit synchronized vor z.B. remove Methode-> nur ein Thread darf gleichzeitig in der Methode sein
  - Nachteil:
    - *meist nicht nötig*
    - *andere Threads werden u.U. behindert*
    - *synchronized kostet was*
- **Neue Collection Klassen sind alle non-synchronized**

- Müssen mit `Collections.synchronizedList()` bei Bedarf Thread-Safe gemacht werden:

```
List list = new LinkedList()
list = Collections.synchronizedList (list );
```

Not Implemented und Read-Only

Problem

- Listen sollen vor unbeabsichtigter Veränderung geschützt werden

Lösung

- können mit `Collections.unmodifiableList()` unveränderbar gemacht werden.

```
List list = new LinkedList()
List = Collections.unmodifiableList(list );
```

Bemerkung: analoge Methoden für `Set, SortedSet, Map, SortedMap`

Problem

- Das List Interface ist gross und einige Methoden machen für gewisse Implementation keinen Sinn

Lösung

- Es wird die `UnsupportedOperationException` von diesen Methoden geworfen

Zusammenfassung

School of Engineering

- Einfach verkettete Liste
  - Operationen: add, remove
  - Iterator: zum Traversieren der Liste
- Doppelt verkettete Listen
- Zirkuläre, doppelt verkette Liste
- Sortierte Listen
  - Das Comparable Interface, die Comparator Klasse
- Klassenhierarchie der Collection Klassen
- Spezialfälle
  - Thread-Safe, Read-Only
  - Vollständigkeit der List-Interfaces

Anhang

Sets

Anwendungsbeispiel

- 1) Alle Wörter eines Texts sollen gesammelt werden um dann später die Liste der Wörter aufzulisten.

- Geht mit Listen
  - Vor dem Einfügen überprüfen ob das Wort schon in der Liste vorhanden ist.
  - `if (!list.contains(s)) list.add(s)`

- 2) Frage: Was sind die gemeinsamen Wörter von zwei Texten?

Weiterer ADT: Set

- **set**: ist eine ungeordnete Menge ohne Duplikate
  - wichtigste Operation `contains` um herauszufinden ob ein Objekt Teil der Menge ist.

- Interface: **set**
  - Es fehlen folgende Methoden
    - *get(*index*)*
    - *add(*index*,* value*)*
    - *remove(*index*)*

- Implementation durch
  - **Hashset** bei grossen Datenbeständen (etwas) effizienter
  - **TreeSet** speichert die Elemente in alphabetischer (geordneter) Folge

- Beide Implementationen sehr effizient

Anwendungsbeispiel

- Folgendes Beispiel zeigt die Anwendung

```
Set stooges = new HashSet();
stooges.add("Larry");
stooges.add("Moe");
stooges.add("Curly");
stooges.add("Moe");    // duplicate, won't be added
stooges.add("Shemp");
stooges.add("Moe");    // duplicate, won't be added
System.out.println(stooges);
```

- Output:

```
[Moe, Shemp, Larry, Curly]
```

  - Die Reihenfolge ist zufällig (später mehr)

- Falls `TreeSet` verwendet wird:

```
Set stooges = new TreeSet();
```

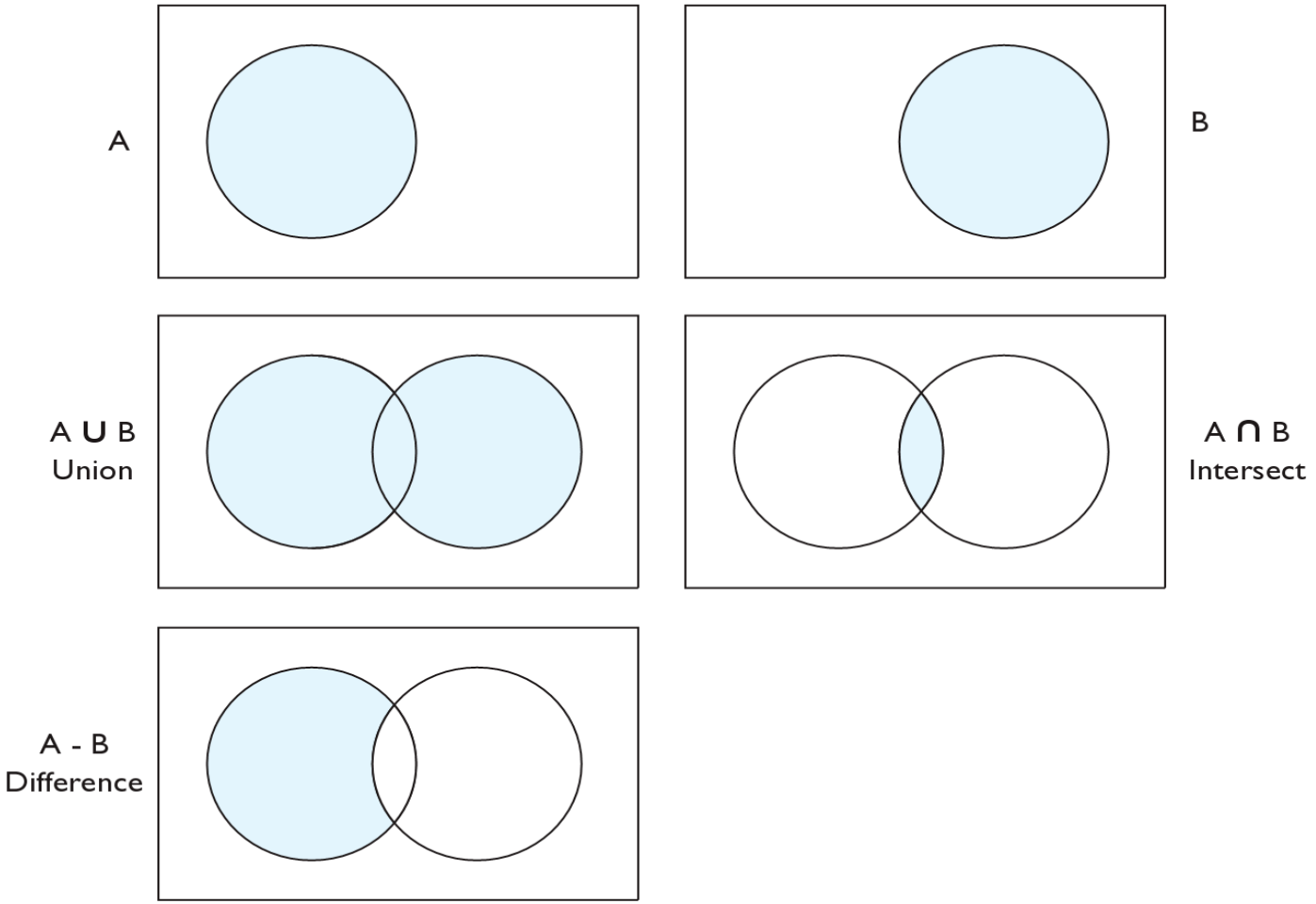
  - Die Reihenfolge ist alphabetisch

- Output:

```
[Curly, Larry, Moe, Shemp]
```

Set Operation

- Sets unterstützen die gängigen Mengenoperationen



Beispiele

- Um zwei Sets miteinander zu vergleichen:
  - **subset**: S1 is a *subset* of S2 if S2 contains every element from S1.
    - *`containsAll` tests for a subset relationship.*
- um zwei Sets zusammenzufügen
  - **union**: S1 *union* S2 contains all elements that are in S1 or S2.
    - *`addAll` performs set union.*
  - **intersection**: S1 *intersect* S2 contains only the elements that are in *both* S1 and S2.
    - *`retainAll` performs set intersection.*
  - **difference**: S1 *difference* S2 contains the elements that are in S1 that are *not* in S2.
    - *`removeAll` performs set difference.*

Collections im JDK

Das List Interface

Method Summary	
void	**add**(int index, Object element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**add**(Object o) Appends the specified element to the end of this list (optional operation).
boolean	**addAll**(Collection c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection c) Returns true if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
Object	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.
ListIterator	**listIterator**() Returns a list iterator of the elements in this list (in proper sequence).
ListIterator	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
Object	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence in this list of the specified element (optional operation).
boolean	**removeAll**(Collection c) Removes from this list all the elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection c) Retains only the elements in this list that are contained in the specified collection (optional operation).
Object	**set**(int index, Object element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

© K. Rege, ZHAW

Das ListIterator Interface

Method Summary	
void	**add**(Object o) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
Object	**next**() Returns the next element in the list.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to next.
Object	**previous**() Returns the previous element in the list.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to previous.
void	**remove**() Removes from the list the last element that was returned by next or previous (optional operation).
void	**set**(Object o) Replaces the last element returned by next or previous with the specified element (optional operation).

Funktionen

- Elementeinfügung
- Austausch
- bidirektionalen Zugriff .

`ListIterator` wird von `Liste` erzeugt:

```
public ListIterator
      listIterator(int index)
```

Achtung: Index bezeichnet das erste Element welches beim erstmaligen Aufruf von **`next`** zurückgegeben wird.

Ist der Index ausserhalb des zulässigen Bereichs (index < 0 || index > size()) wird eine **`IndexOutOfBoundsException`** geworfen.

Wert-Typen und Referenz-Typen

Fragen

- Gegeben folgende Java Zeilen

```
int[] a = {3,1,2};
int[] b;
b = a;
b[0] = 1;
```

- welchen Wert hat a[0]?

- Ist folgende Klassendeklaration korrekt?

```
public class C {
     public C p;
}

C c = new C();
c.p = c;
```

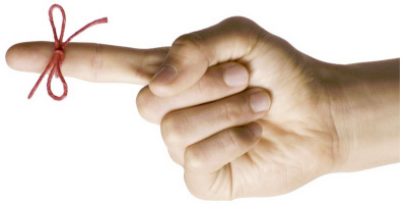
Wert-Typen und Referenz-Typen

- einfache Wert-Typen in Java sind:
  - byte, short, int, long
  - float, double
  - char
  - boolean

  Beispiel
  - `int a, b;`
  - `a = 3; b = a;`
- eingebaute Referenz-Typen in Java sind
  - Arrays
  - von Object abgeleitet -> alle Klassen
  - String (verhält sich bei Operationen wie ein Wert-Typ, `s = s + "aa" + "b"`)

  Beispiel
  - `int[] a = {3,1,2};`
  - `int[] b;`
  - `b = a;`
  - `b[0] = 1; // a[0] = 1`

Wert-Typen und Referenz-Typen



- Objekt-Variablen sind lediglich Referenzen (Zeiger) auf Objekte
- `int[] a= {3,1,2};`
- `int[] b;`

- am Anfang zeigen diese nirgendwo hin: `null`
- `int[] a,b;`

```
a[0] = 3; // Fehler
```

- können mittels der Zuweisung gesetzt:

```
a = new int[3];
b = a;
```

- oder wieder zu null gesetzt

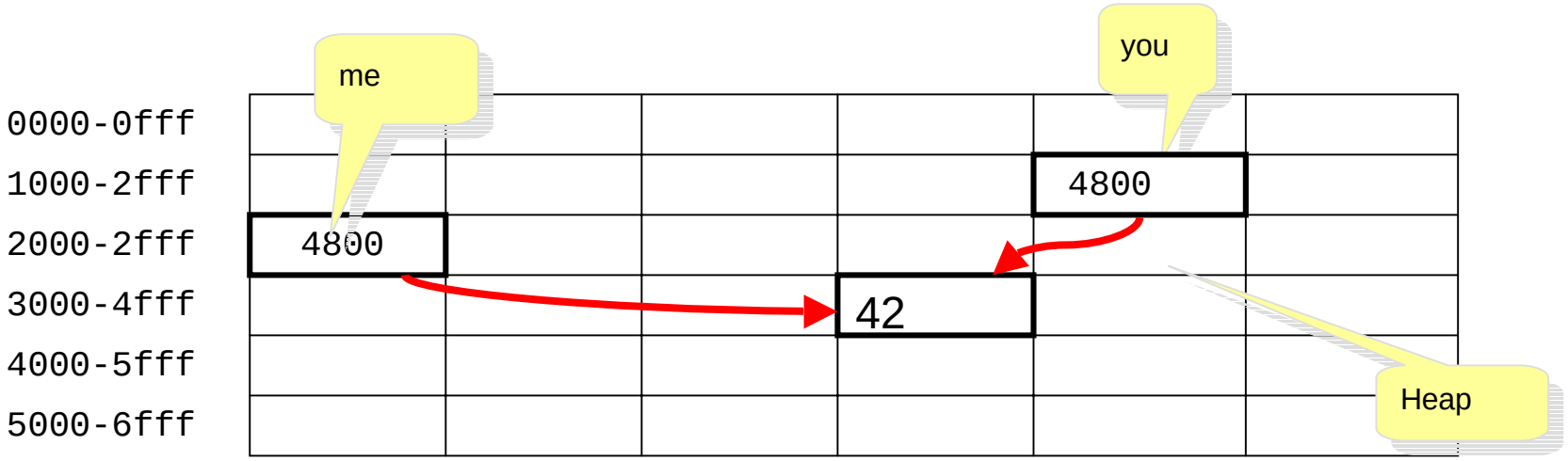
```
a = null;
```

Wert-Typen und Referenz-Typen

0000-0fff						
1000-2fff		radius				
2000-2fff	42					
3000-4fff						
4000-5fff						
5000-6fff						

- Begriffe:
  - Variable: benannter Speicherplatz für einen Wert
    - *statt Adresse 2000 können wir radius schreiben*
  - Wert: "Nutz"-Daten
    - *42*
  - Zuweisung: Deponieren eines Wertes in einer Variablen
    - *radius = 42*
  - Ausdruck: Liefert einen Wert: radius = 6*7;

Wert-Typen und Referenz-Typen



- Zeiger (Referenzen) sind lediglich Verweise auf den eigentlichen Wert

```
class MyClass {
      int val;
}
MyClass me = new MyClass(); me.val = 42;
MyClass you = me;
```