

*Einführung in die Informatik I*  
*Sortieren*  
*Verkettete Listen*

Prof. Bernd Brügge, Ph.D  
Technische Universität München

Wintersemester 2000/2001

15 und 16. Januar 2001

# *Überblick über diesen Vorlesungsblock*

## ❖ Themen:

- Einfügen in Reihungen
- Suchen vs. Sortieren
- Noch mehr über Reihungen (2-D und 3-D Arrays)
- Referenzvariablen (noch einmal)
- Verkettete Listen

## ❖ Ziele:

- Sie sind in der Lage, Probleme zu lösen, die Daten mit mehreren Dimensionen erfordern.
- Sie können aktiv mit verketteten Listen umgehen, insbesondere in der Implementation von Such-Algorithmen.

## *Einfügen eines Elementes*

- ❖ **Laufzeitkomplexität für nichtsortierte Reihung:  $O(1)$**

**Nichtsortierte Reihung:  
Array.html**

- ❖ **Laufzeitkomplexität für sortierte Reihung:  $O(n)$**

**Sortierte Reihung:  
Ordered.html**

# Sequentieller Suchalgorithmus für Reihenungen

**Problem:** Suche in einer Reihung nach einem Schlüsselwert (key value). Wir nehmen an, die Reihung ist nicht sortiert, und müssen daher eine *sequentielle Suche* durchführen.

```
public int sequentialSearch(int[] a, int key) {  
    for (int k = 0; k < a.length; k++)  
        if (a[k] == key)  
            return k;  
    return -1;    // Nicht gefunden  
} // sequentialSearch()
```

Sobald der Wert gefunden ist, können wir zum Aufrufer zurückkehren.

Die Suche war nicht erfolgreich.

## *Komplexität von sequentieller Suche*

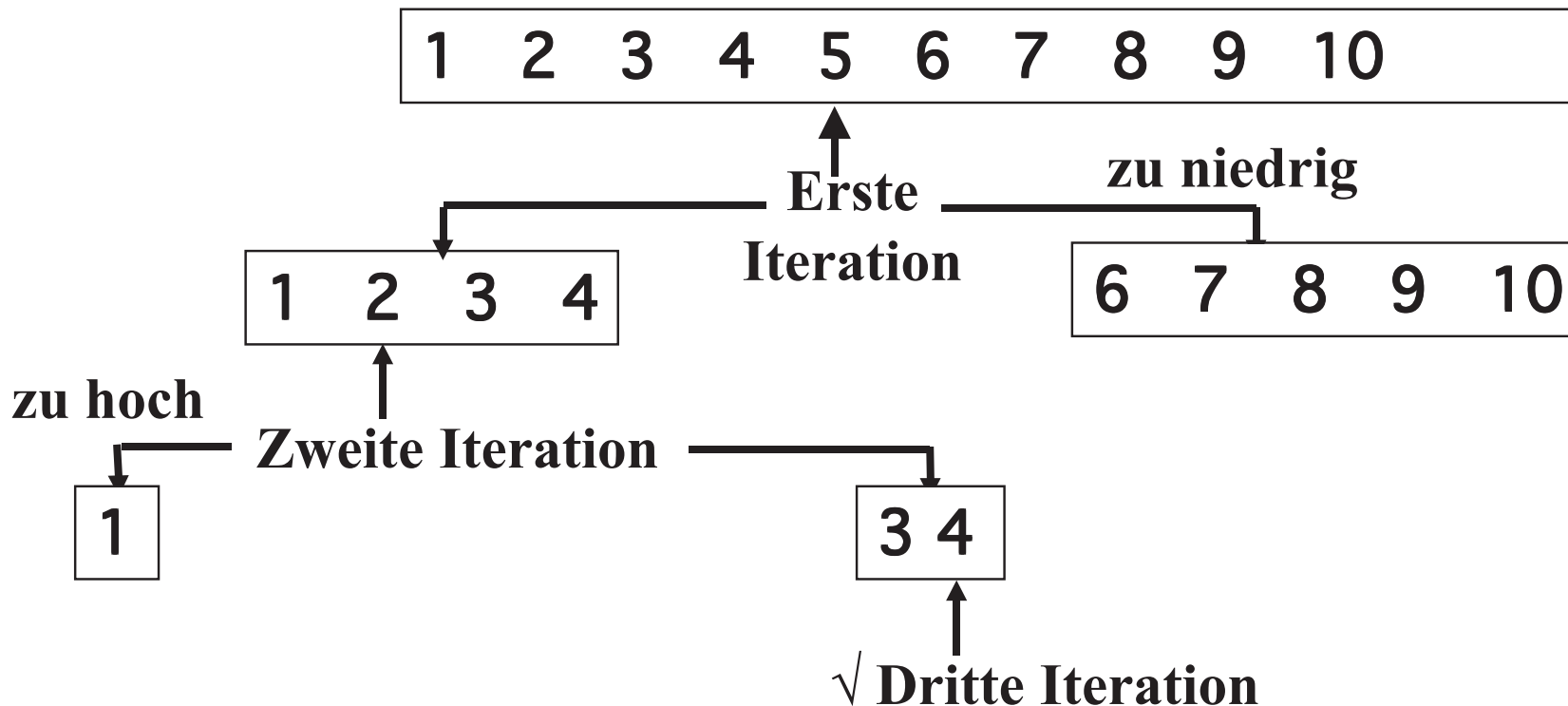
- ❖ Die Suche nach einem Element in einer ungeordneten Reihung mit  $n$  Elementen ist
  - im schlechtesten Fall:  $V(n) = n$
  - im durchschnittlichen Fall:  $V(n) = n/2$
  - im besten Fall:  $V(n) = 1$
- ❖ **Komplexität für sequentielle Suche:  $O(n)$**

# Idee des Algorithmus Binäre Suche

**Binäre Suche (binary search)** benutzt eine sog. "Teile-und-Herrsche"-Strategie (Divide-and-conquer) auf einer sortierten Reihung.

Wir teilen die Reihung bei jeder Iteration in 2 Hälften, und suchen in der Hälfte, in der der Schlüssel sein könnte.

Beispiel: Wir suchen nach der Zahl 4

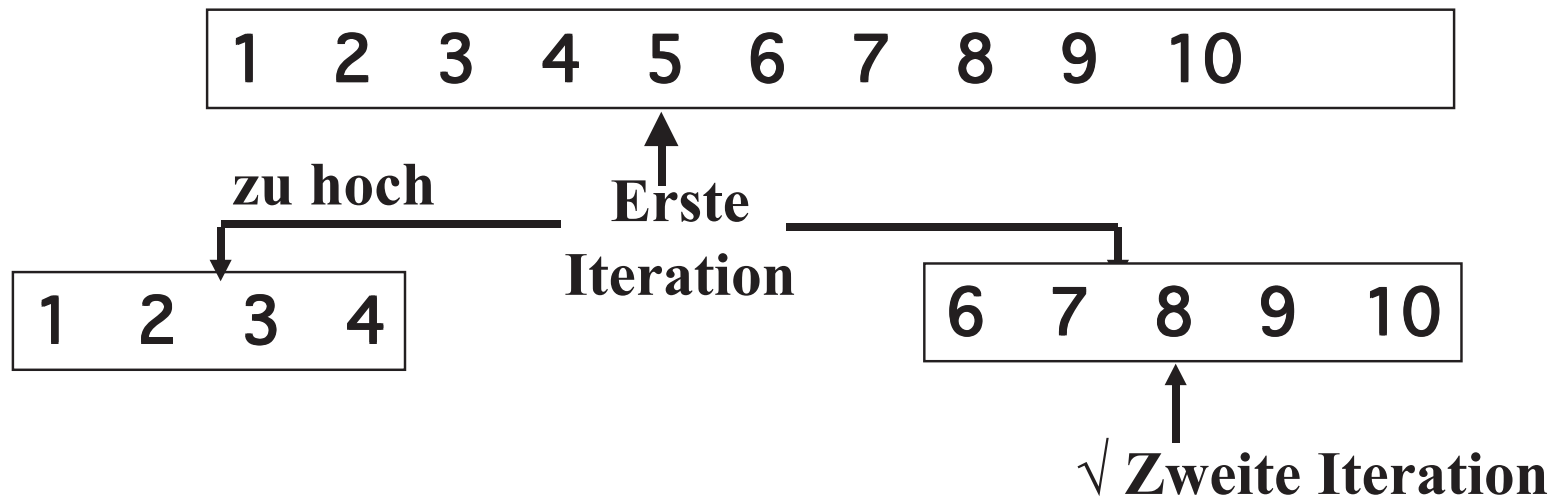


## *Idee des Algorithmus Binäre Suche*

**Binäre Suche (binary search)** benutzt eine sog. "Teile-und-Herrsche"-Strategie (Divide-and-conquer) auf einer sortierten Reihung.

Wir teilen die Reihung bei jeder Iteration in 2 Hälften, und suchen in der Hälfte, in der der Schlüssel sein könnte.

Beispiel: Wir suchen nach der Zahl 8



## Implementation von binärer Suche

Algorithmus: *low* und *high* zeigen auf das erste und letzte Element der Teilreihung (*subarray*), und *mid* zeigt immer auf den derzeitigen Mittelpunkt in der Reiheung oder Teilreihung.

```

/**
 * Vorbedingung: a ist aufsteigend sortierte Reiheung von int
 * Nachbedingung: Ergebnis ist -1 oder k, wobei a[k] == key
 */
public int binarySearch(int[] a, int key) {
    int low = 0; // Initialisierung
    int high = a.length - 1;
    while (low <= high) { // Solange wir nicht fertig sind
        int mid = (low + high) / 2;
        if (a[mid] == key) // Erfolg
            return mid;
        else if (a[mid] < key) // Suche in der oberen Hälfte
            low = mid + 1;
        else // Suche in der unteren Hälfte
            high = mid - 1;
    } // while
    return -1; // low > high: Suche erfolglos
} // binarySearch()

```

Berechnung des  
neuen  
Mittelpunktes

Wenn  $low > high$ , dann ist  
der Schlüssel nicht in der  
Reihung.

low bzw. high anpassen, um die  
Reihung in 2 Hälften zu schneiden



# Visualisierung von binärer Suche

ordered.html

Ordered

New Fill Ins Find Del

Linear  Binary Number: 594

Checking index 37, range = 25 to 49

0	0	12	237	24	496	36	699	48	955
1	7	13	293	25	518	37	708	49	997
2	9	14	307	26	526	38	714		
3	23	15	332	27	584	39	732		
4	41	16	364	28	594	40	781		
5	44	17	400	29	614	41	808		
6	49	18	433	30	621	42	809		
7	67	19	436	31	626	43	818		
8	82	20	443	32	650	44	852		
9	115	21	453	33	675	45	879		
10	134	22	480	34	677	46	910		
11	230	23	481	35	697	47	911		

Applet Loaded

## *Komplexität von binärer Suche*

- ❖ Lineare Suche läuft auf beliebigen Reihungen
- ❖ Binäre Suche braucht eine sortierte Reihung
- ❖ Bei einer Reihung von bis zu  $n$  Elementen brauchen wir höchstens  $\log_2(n)$  Vergleiche
- ❖ Die **Komplexität von binärer Suche** für sortierte Reihungen ist also  **$O(\log_2(n))$** .

max. Anzahl von Elementen $n$ in der Reihung	$2^{\log_2(n)}$	max. Anzahl von Teilungen $\log_2(n)$
1	$2^0$	0
2	$2^1$	1
4	$2^2$	2
8	$2^3$	3
16	$2^4$	4
32	$2^5$	5
64	$2^6$	6
128	$2^7$	7
256	$2^8$	8
512	$2^9$	9
513	$2^{10}$	10
1024	$2^{10}$	10

## 2-dimensionale Reihungen

**Motivation:** Berechne die durchschnittliche monatliche Regenmenge in München.

Mit einer 1-dimensionalen Reihung wäre das möglich, aber sehr kompliziert:

```
double[] rainfall = new double[365];
```

Eine *2-dimensionale Reihung* ist eine Reihung von Reihungen (an array of arrays).

Die erste Reihung sind die 12 Monate, indiziert von 0 bis 11.

Jeder Monat ist wieder eine Reihung von 31 Tagen, indiziert von 0 bis 30.

```
double[][] rainfall = new double[12][31];
```

Monatsindex

Tagesindex

## Eine 2-D-Repräsentation

Der Regenfall am 3. Januar ist mit dieser Repräsentation `rainfall[0][2]` (schlechte Lesbarkeit durch Null-Indizierung).

Wir können die Lesbarkeit erhöhen, indem wir eine extra Reihe und Spalte erlauben und die 0-Indizes ignorieren:

Wir ignorieren  
diese Spalte

```
double[][] rainfall = new double[13][32];
```

0,0	0,1	0,2	0,3	...	0,29	0,30	0,31	
1,0	1,1	1,2	1,3	...	1,29	1,30	1,31	← Januar
2,0	2,1	2,2	2,3	...	2,29	2,30	2,31	← Februar
.								
.								
.								
11,0	11,1	11,2	11,3	...	11,29	11,30	11,31	← November
12,0	12,1	12,2	12,3	...	12,29	12,30	12,31	← Dezember

Wir ignorieren diese Reihe

3. Januar ist  
`rainfall[1][3]`

```
rainfall[1][5] = 1.15; // Regen für 5. Januar
rainfall[13][32] = 0.15 ; // Gibt es nicht!
rainfall[11][32] = 1.3; // 32. Spalte gibt es nicht!
rainfall[13][30] = 0.74; // 13. Reihe gibt es nicht!
```

## *Initialisierung einer 2-dimensionalen Reihung*

Wir benutzen 2 geschachtelte Schleifen, um die Reihung zu initialisieren:

```
/**
 * Initialisiert eine 2-D Reihung
 * Vorbedingung: rain ist nicht null
 * Nachbedingung: rain[x][y] == 0.0 für alle x,y in der Reihung
 */
public void initRain(double[][] rain) {
    for (int month = 1; month < rain.length; month++)
        for (int day = 1; day < rain[month].length; day++)
            rain[month][day] = 0.0;
} // initRain()
```

Eine 2-D Reihung als  
formaler Parameter

Zwei geschachtelte  
Zählschleifen, die  
12 x 31 iterieren

**Methodenaufruf:** Übergibt den *Namen* der Reihung an die  
Methode:

```
double[][] rainfall = new double[13][32];
initRain(rainfall); // Beispiel eines Methodenaufrufs
```

## *Berechnung des durchschnittlichen täglichen Regens*

```
public double avgDailyRain(double[][] rain) {  
    double total = 0.0;  
    for (int month = 1; month < rain.length; month++)  
        for (int day = 1; day < rain[month].length; day++)  
            total += rain[month][day];  
    return total/365;  
}
```

```
....  
....  
System.out.println("Täglicher Regen: " + avgDailyRain(rainfall));
```

Methodenaufruf innerhalb einer  
Druckanweisung.

## *Berechnung des durchschnittlichen monatlichen Regens*

```
public double avgRainForMonth(double[][] rain, int month, int nDays) {  
    double total = 0.0;  
    for (int day = 1; day <= nDays; day++)  
        total = total + rain[month][day];  
    return total/nDays;  
} // avgRainForMonth()
```

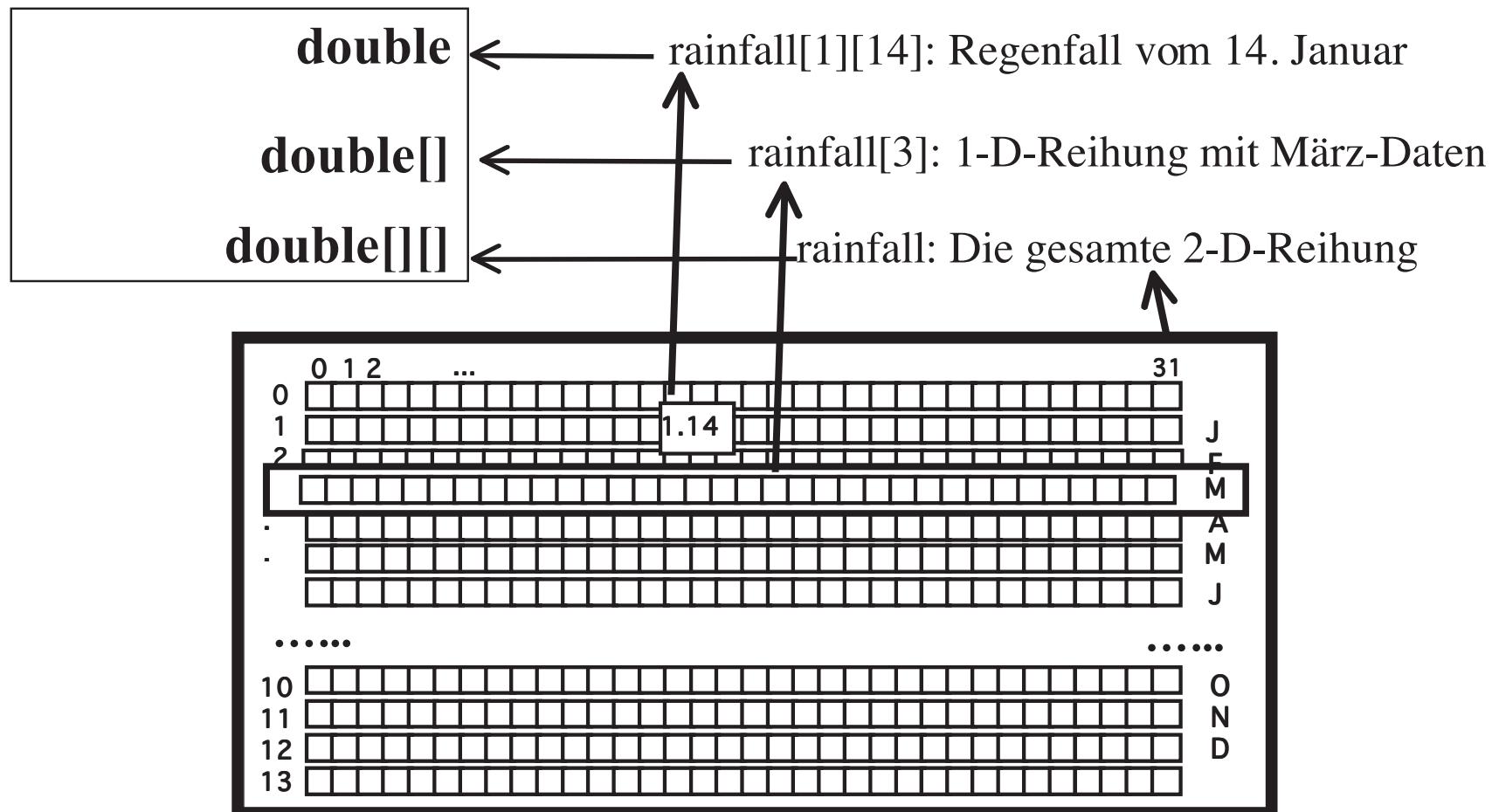
Iteration über die  
Monatsreihe

Wir müssen sagen, wieviel  
Tage im Monat sind

```
System.out.println("März Durchschnitt: " + avgRainForMonth(rainfall, 3, 31));
```

## Reihungen und Reihungswerte als Parametertypen

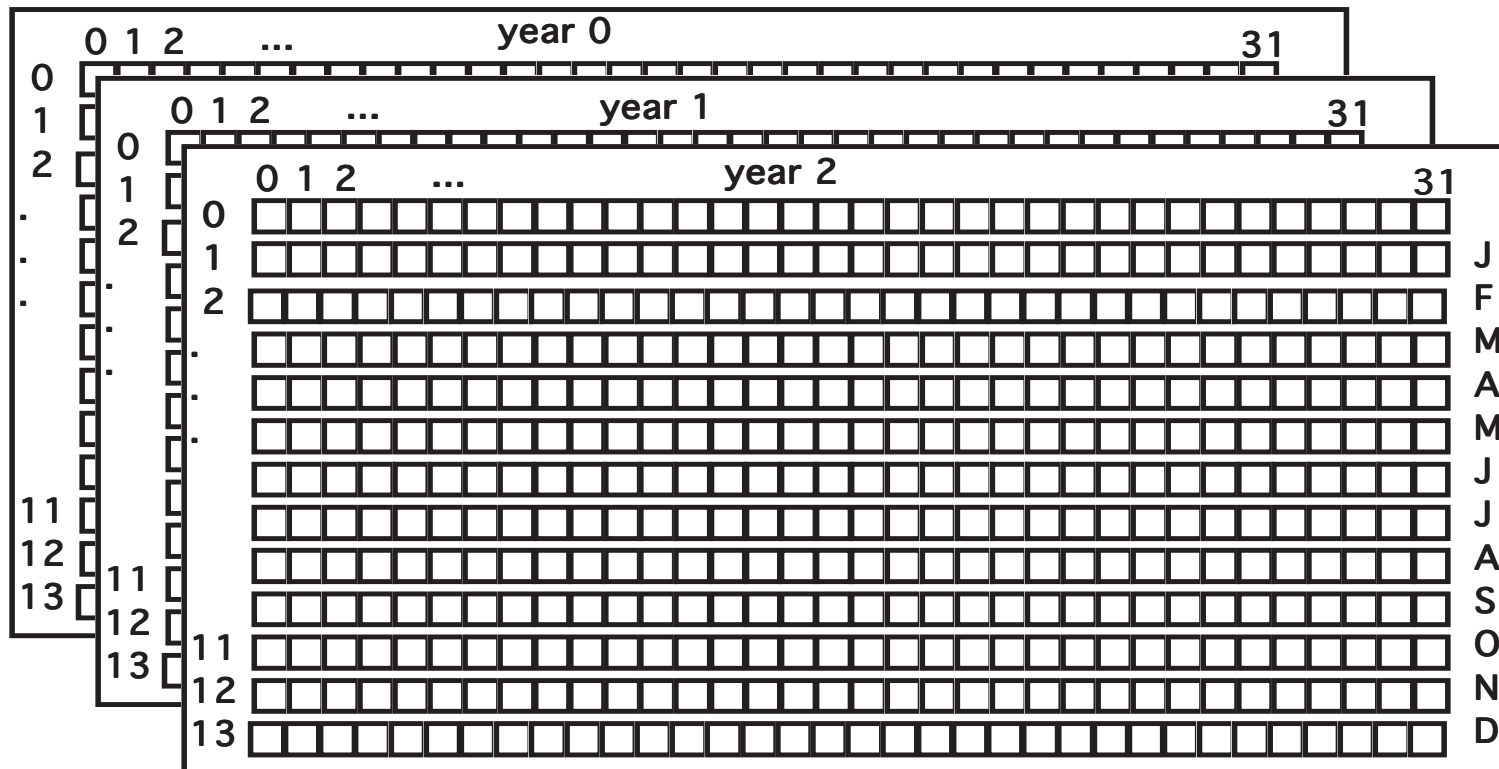
Der Typ des aktuellen Parameters bei einem Methodenaufruf muss denselben Typ wie der formale Parameter in der Methodendefinition haben. Dies gilt für alle Parameter einschließlich Reihungen:





# Multidimensionale Reihungen

Eine 3-dimensionale Reihung kann man für die Messung von Regenfällen über mehrere Jahre benutzen.



## *Eine 3-D-Reihung*

❖ Deklaration der Reihung:

```
final int NYEARS = 10;
final int NMONTHS = 13;
final int NDAYS = 32;
double[][][] rainfall = new double[NYEARS][NMONTHS][NDAYS];
```

Initialisierung mit 3 geschachtelten Schleifen:

```
for (int year = 1; year < rainfall.length; year++) {
    for (int month = 1; month < rainfall[year].length; month++) {
        for (int day = 1; day < rainfall[year][month].length; day++) {
            rainfall[year][month][day] = 0.0;
        }
    }
}
```

## *Initialisierung von Reihenungen*

- ❖ Bei kleinen Reihenungen, kann man einen Initialisierungsausdruck (initializer expression) für die Zuweisung von Anfangswerten verwenden:

2 x 3 Reihung  
von int-Werten

2 x 2 Reihung  
von char-Werten.

```
int[][] a = { {1, 2, 3}, {4, 5, 6} };  
char [][] c = { {'a', 'b'}, {'c', 'd'} };  
double [][][] d = { { {1.0, 2.0}, {3.0, 4.0} },  
                    { {5.0, 6.0}, {7.0, 8.0} },  
                    { {9.0, 10.0}, {11.0, 12.0} }  
};
```

3 x 2 x 2 Reihung  
von double-Werten.

In Java kann jede Teilreihung einer mehr-dimensionalen Reihung eine unterschiedliche Länge haben!

## *Dynamische Reihungen: java.util.Vector*

- ❖ Reguläre Reihungen sind auf die zum Zeitpunkt der Initialisierung (*statisch*) vereinbarte Größe beschränkt. Sie können später nicht mehr wachsen oder kleiner werden.
- ❖ Die `java.util.Vector` Klasse implementiert eine Reihung von *Objekten* (keine Grundtypwerte als Elemente!), die auch nach der Initialisierung (*dynamisch*) wachsen kann.

```
public class Vector {
    public Vector(int size);    // Konstruktor: Dynamisches Array mit Anfangsgröße
    public Vector();           // Konstruktor: Dynamisches Array ohne Anfangsgröße
    // Öffentliche Methoden
    public void addElement(Object obj);    // Füge obj zum Array dazu
    public Object elementAt(int index);    // Hole das bei index gespeicherte Element
    public void insertElementAt(Object o, int index); // Speicher o bei index
    public int indexOf(Object obj);        // Ermittle den index von obj
    public int lastIndexOf(Object obj);
    public void removeElementAt(int index); // Entferne ein Element bei Index
    public int size();                      // Länge des dynamischen Arrays
}
```

# *Listen*

- ❖ Reihungen haben den Vorteil des direkten Zugriffes (random access) auf Elemente: Wenn der Index bekannt ist, dann ist die Zugriffszeit  $O(1)$ .
- ❖ Reihungen haben allerdings einige Nachteile:
  - In einer unsortierten Reihung ist das Einfügen eines Elementes  $O(1)$ , also schnell, aber die Suche nur  $O(n)$ .
  - In einer sortierten Reihung ist das Einfügen eines Elementes nur  $O(n)$ , aber die Suche schnell:  $O(\log_2(n))$ .
  - Das Entfernen von Elementen ist in beiden Fällen  $O(n)$ , also langsam.
- ❖ Wir schauen uns nun eine Datenstruktur namens **Verkettete Liste** an, die diese Probleme nicht hat.
- ❖ Verkettete Listen können überall dort besser eingesetzt werden, wo nicht häufig direkter Zugriff notwendig ist.

# *Typen von Listen*

- ❖ Es gibt verschiedene Typen von Listen:
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
  - Kompakte Listen
  - Symmetrische Listen
  - Sortierte Listen
  - Listen mit Iteratoren

## *Beispiel einer Liste*

### ❖ Beispiel:

– Wir sind auf einer Party, auf der auch Andreas, Helmut, Selma, Opa und Barbie sind.

– Andreas weiß, wo Helmut ist.

– Helmut weiß, wo Selma ist.

– Selma weiß, wo Opa ist

– Opa weiß, wo Barbie ist.

❖ Um Selma etwas zu sagen, muss ich es Andreas sagen. Der sagt es dann Helmut und der Selma.

❖ Um Barbie zu finden, muss ich....

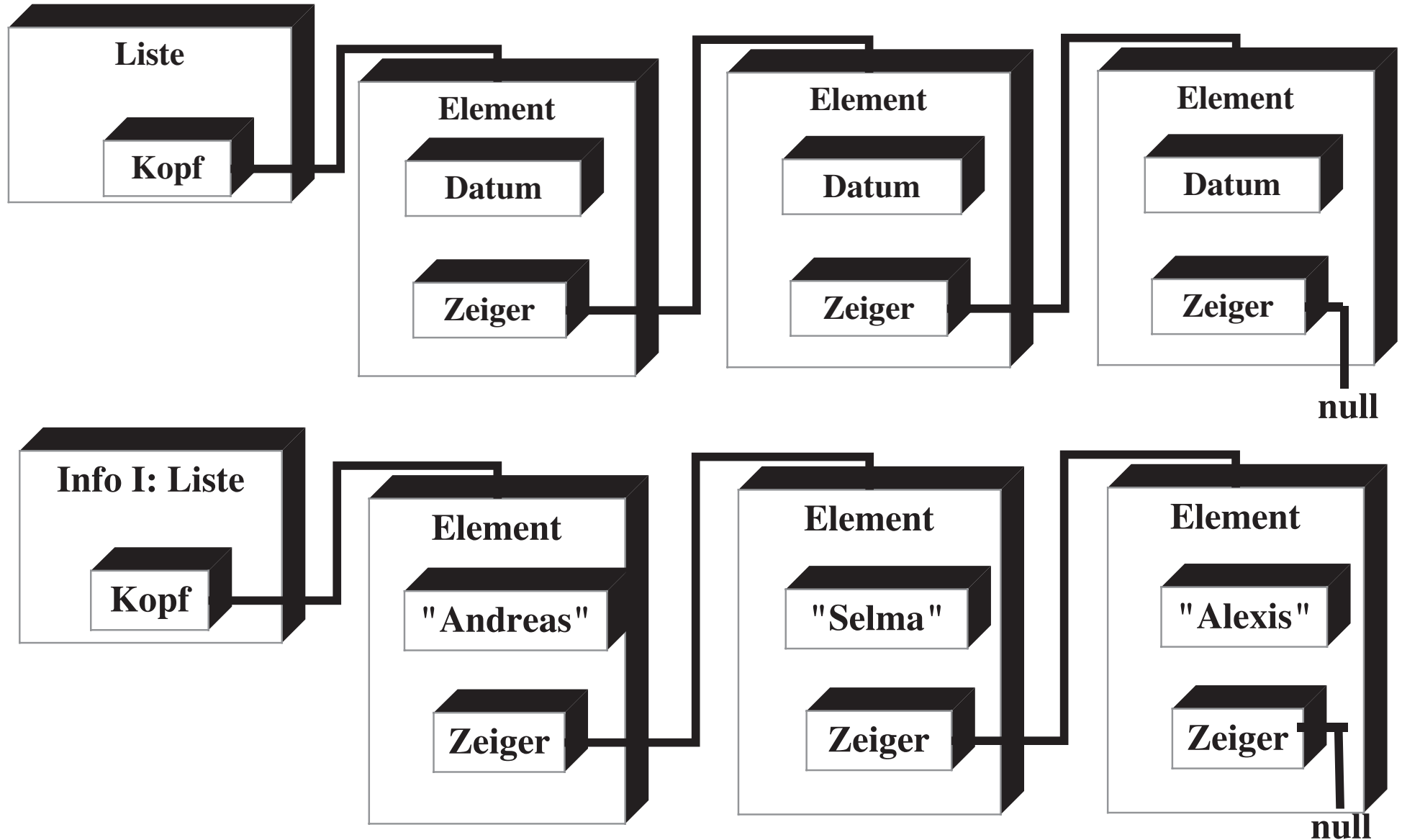


## *Modellierung von Listen*

- ❖ Grundbaustein einer verketteten Liste ist das Listenelement (link).
  - Ein Listenelement enthält immer zwei Attribute:
    - Eine Referenz auf das nächste Listenelement (next)
    - Applikationsspezifische Daten (data)
- ❖ Beispiel:
  - Das Studentenverzeichnis aller Studenten in Info I
  - Die Immatrikulation hat gerade begonnen.
    - Das Studentenverzeichnis besteht aus 3 Studenten:
      - Andreas, Selma, Alexis
- ❖ Wir können das Verzeichnis als verkettete Liste modellieren.
  - Andreas, Selma und Alexis sind dann die applikationsspezifischen Daten (vom Typ Student).

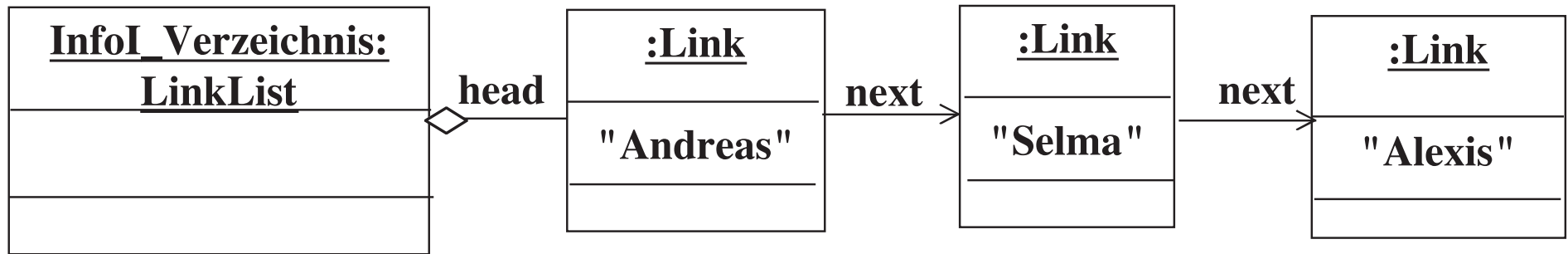


# *Eine Liste hat einen Kopf und Listenelemente*

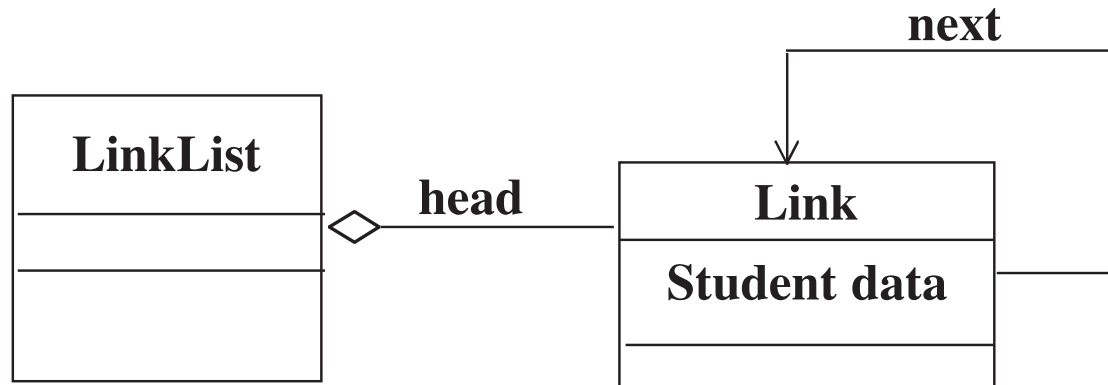


# Liste als Modell (Erste Iteration)

❖ Instanzdiagramm:



❖ Klassendiagramm:



## *Implementation des Listenelementes in Java*

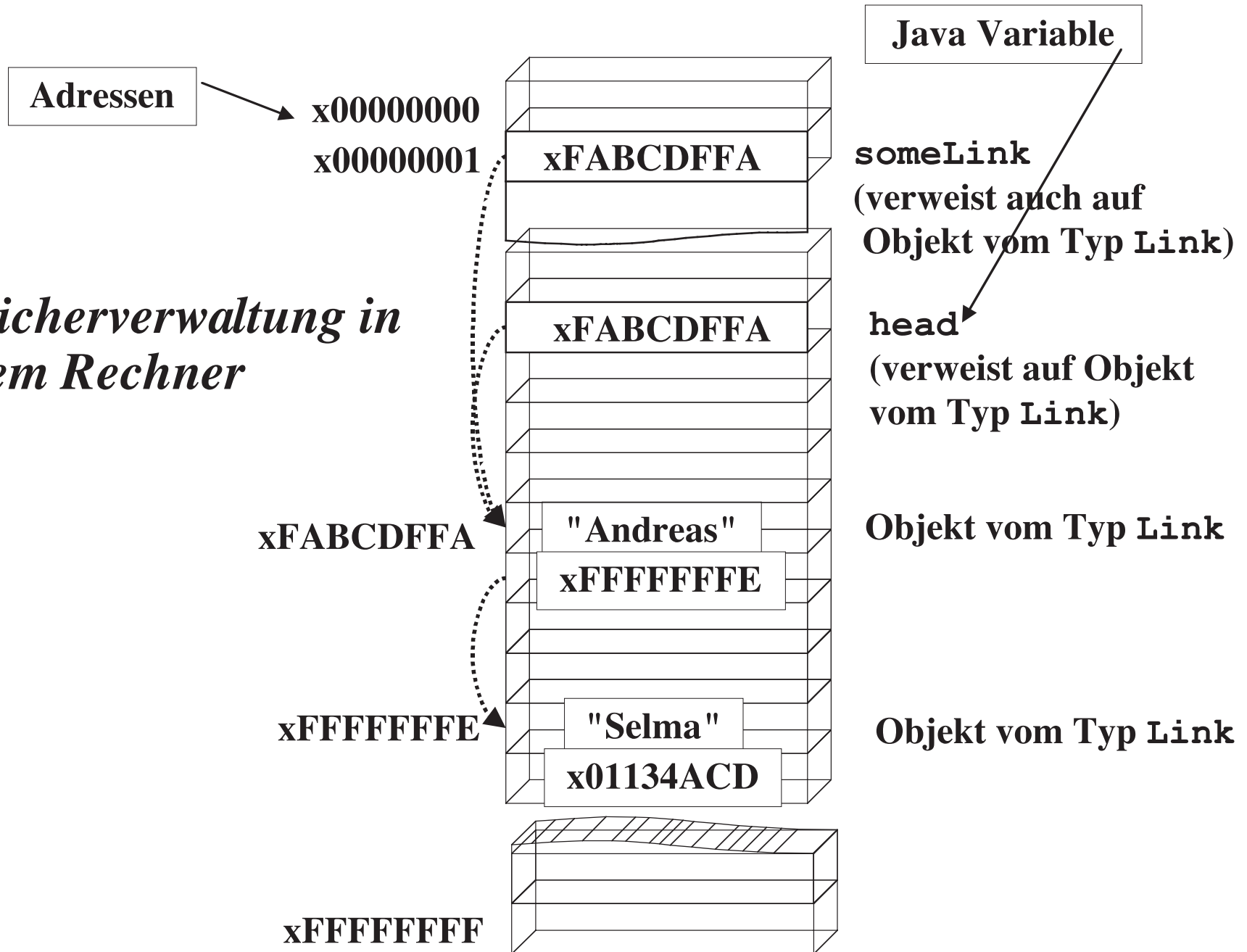
```
class Link {  
    private Link next;  
    private Student data;  
  
    ....  
}
```

Eine derartige Klassendefinition heißt auch rekursiv (self-referential), da sie ein Attribut enthält (in diesem Fall namens next), das von demselben Typ ist wie die Klasse selbst.

## *Verweisvariablen und einfache Variablen in Java*

- ❖ Kann man ein Attribut vom Typ Link in einer Klassendefinition vom Typ Link verwenden?
  - Wird dadurch das Listenelement nicht unendlich groß?
- ❖ Das Link-Attribut enthält kein weiteres Link-Objekt, auch wenn es so aussieht. Das Link-Attribut ist eine Verweisvariable:
  - Es enthält als Wert eine Referenz auf ein Objekt vom Typ Link.
  - Referenzen sind Adressen im Speicher des Rechners.
    - Alle Adressen in einem Rechner haben die gleiche Größe.
    - Eine Verweisvariable belegt nur soviel Speicher, wie zur Speicherung einer Adresse benötigt wird.
    - Daher ist es kein Problem für den Java-Compiler, die Größe von rekursiv definierten Listenelementen zu berechnen.

*Speicherverwaltung in einem Rechner*



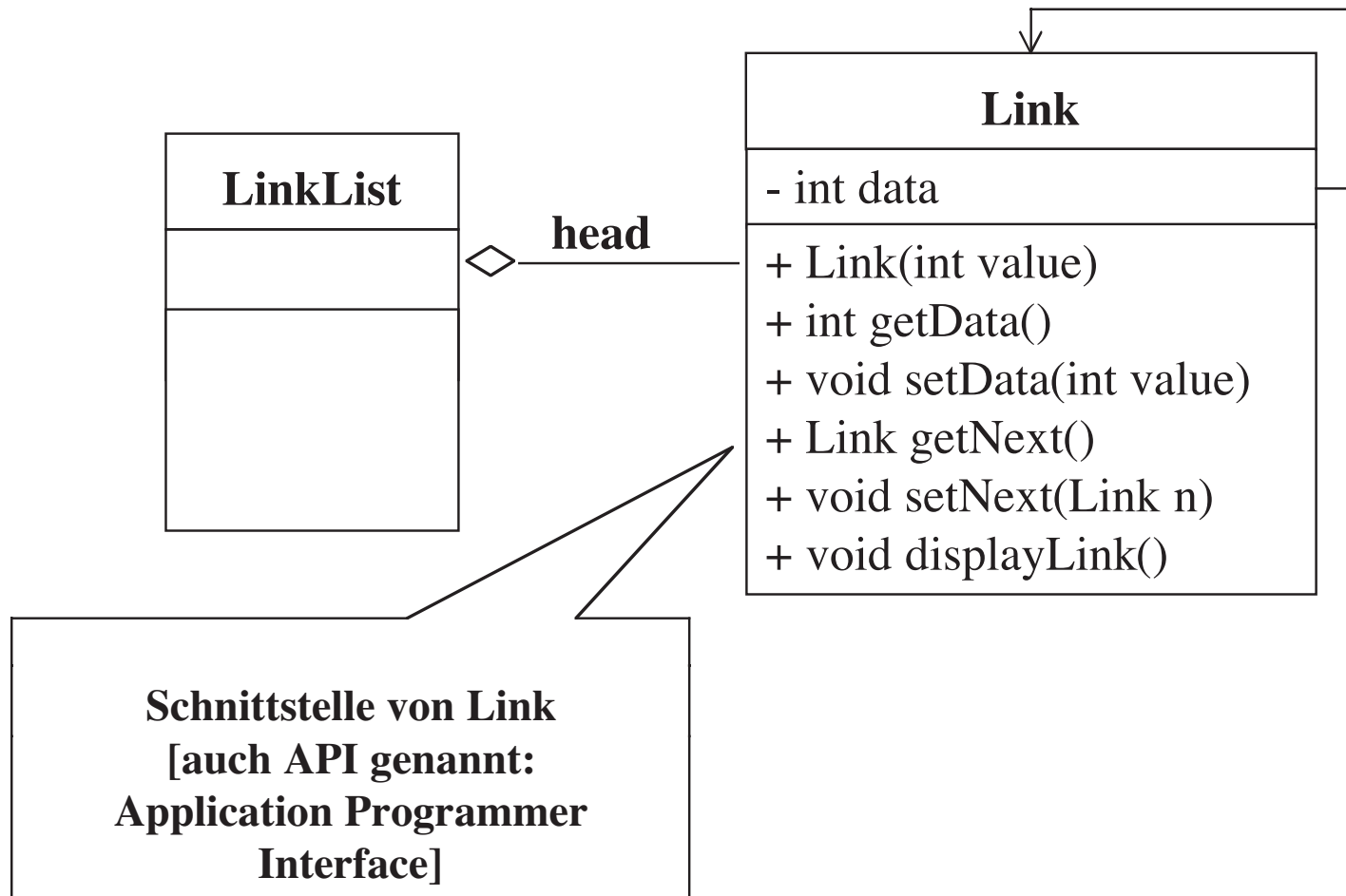
## *Java-Implementation von **int**-Listenelementen*

```
class Link {  
    private int data;    // Datum  
    private Link next; // Nächster Eintrag  
                        // in Liste  
  
    public Link (int value) {  
        data = value; // Initialisiere Datum  
        next = null;  // Initialisiere next  
    }  
  
    public int getData () {  
        return data;  
    }  
  
    public void setData (int value) {  
        data = value;  
    }  
}
```

```
    public Link getNext () {  
        return next;  
    }  
  
    public void setNext (Link n) {  
        next = n;  
    }  
  
    public void displayLink () {  
        System.out.print("{ " + data + " } ");  
    }  
} // end class Link
```

# Modell der Implementation von Link

next



## *Java-Implementation einer Verketteten Liste*

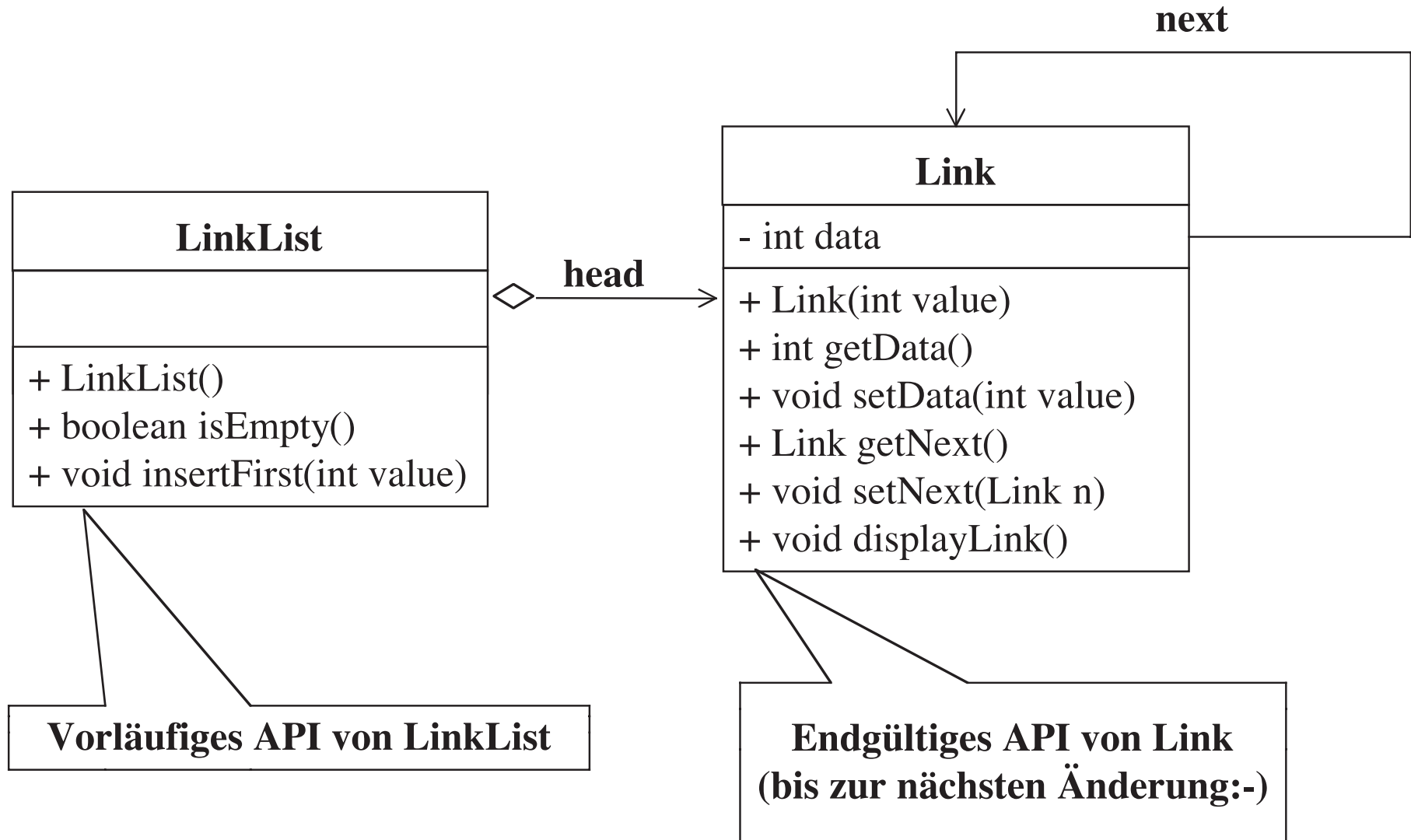
```
class LinkedList {  
    private Link head;           // Referenz auf das erste Element in der Liste  
  
    public LinkedList () {      // Konstruktor  
        head = null;           // Noch keine Elemente in der Liste  
    }  
}
```

**Die Java Konstante null ist der Wert einer  
auf nichts zeigenden Verweisvariable**

```
public boolean isEmpty () {    // wahr, wenn Liste leer ist  
    return (head == null);  
}  
  
public void insertFirst (int value) { // Erzeuge ein neues Element am Anfang  
    newLink = new Link(value);  
    newLink.setNext(head);         // newLink zeigt auf den alten Wert von head  
    head = newLink;               // head zeigt auf das neue Element  
}
```

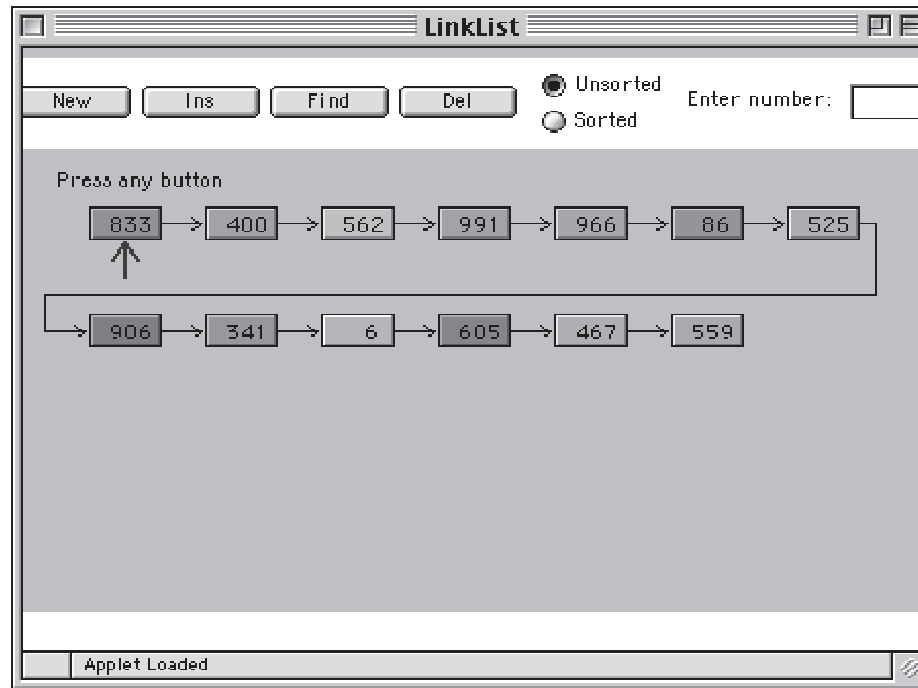


# Modell der Implementation von LinkList



# Verkettete Liste: Einfügen eines Elementes

LinkList.html



## *Verkettete Liste: Löschen des ersten Elementes*


- ❖ Diese Methode nimmt an, dass die Liste nicht leer ist.
- ❖ Das aus der Liste entfernte Element wird als Resultat zurückgegeben (erlaubt Überprüfung, ob ein Element entfernt wurde, und wenn ja, welches)

```
public Link deleteFirst () {  
    Link temp = head;           // sichere die Referenz auf das erste Element  
    head = head.getNext();     // Löschen: head zeigt jetzt auf das nächste Element  
    return temp;               // ehemaliges erstes Element wird zurückgegeben  
}
```

## *Verkettete Liste: Finden eines Elementes*

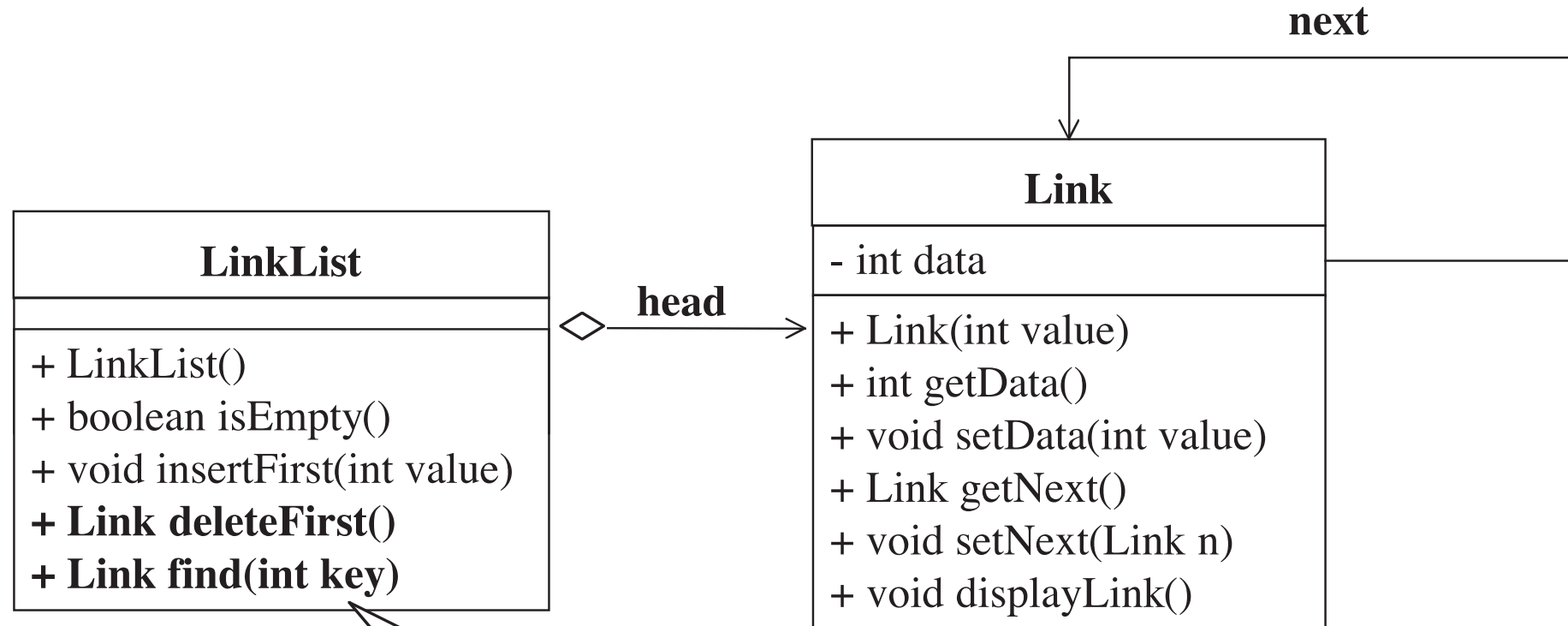
- ❖ Finde ein Listenelement mit dem Wert key
- ❖ Wenn ein Element gefunden wird, wird es als Resultat zurückgegeben, sonst ist das Resultat null

```
public Link find(int key) {  
    Link current = head;           // wir starten am Kopf der Liste  
    while ((current != null) && (current.getData() != key)) {  
                                   // solange wir noch nicht am Listenende  
                                   // sind und noch nichts gefunden haben,  
        current = current.getNext(); // holen wir uns das nächste Element  
    }  
    return current;                // entweder haben wir das Element gefunden,  
                                   // oder current ist null!  
}
```



Wenn next public wäre, könnten wir auch schreiben:  
current = current.next

## *Neues Modell ("Nächste Iteration")*



**deleteFirst() und find() sind neu  
im API von LinkList**

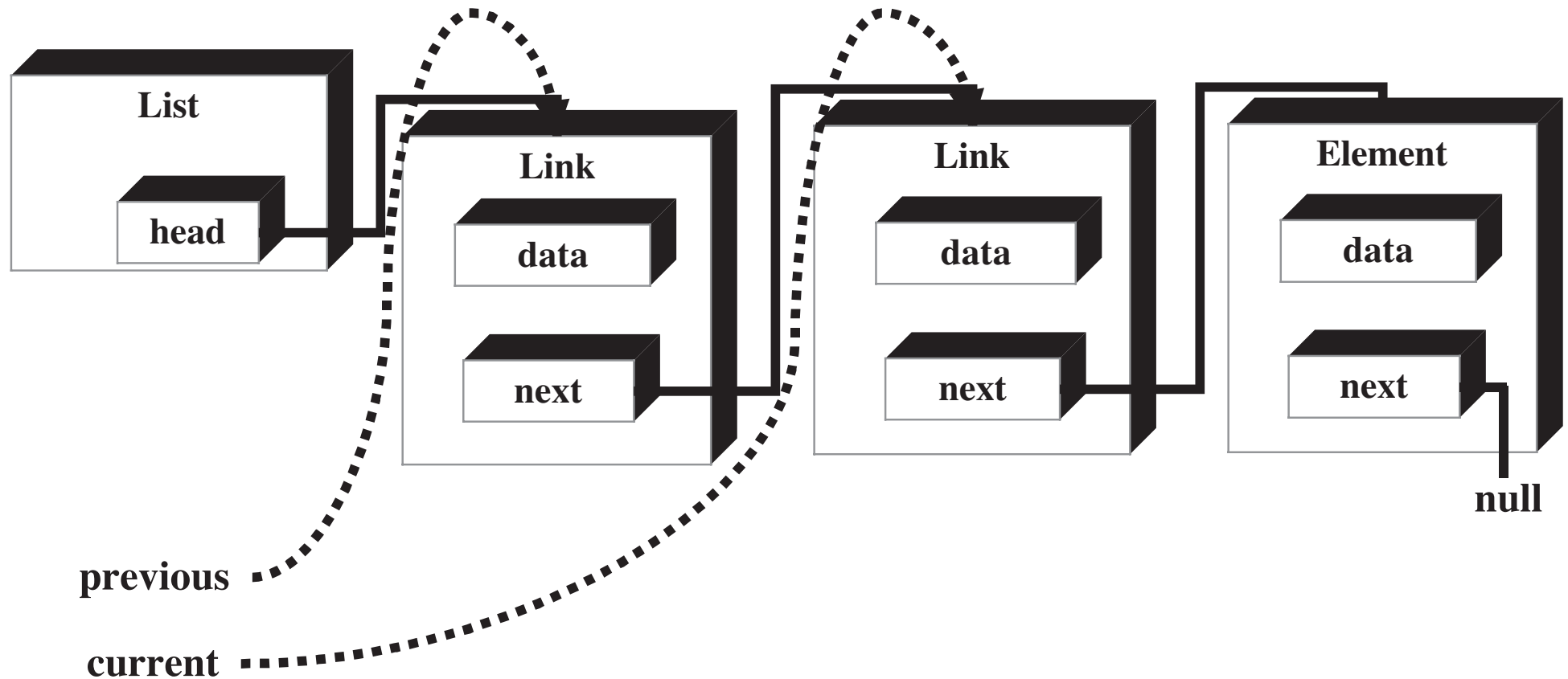
## *Verkettete Liste: Löschen eines beliebigen Elementes*

```
public Link delete (int key) { // Annahme: Liste nicht leer LinkedList.html
    Link current = head; // erst suchen wir nach dem Element
    Link previous = head;
    while (current.getData() != key) { // solange wir es nicht gefunden haben
        if (current.getNext() == null)
            return null;
        else {
            previous = current; // wir merken uns dieses Element
                                // für eventuellen "Bypass"
            current = current.getNext(); // und gehen jetzt zum nächsten Element
        }
    } // wir haben das gesuchte Element gefunden
    if (current == head) // wenn es das erste Element war,
        head = head.getNext(); // löschen wir es.
    else // sonst ...
        previous.setNext(current.getNext()); // ... machen wir einen "Bypass"
    return current;
}
```

**previous.next = current.next;**

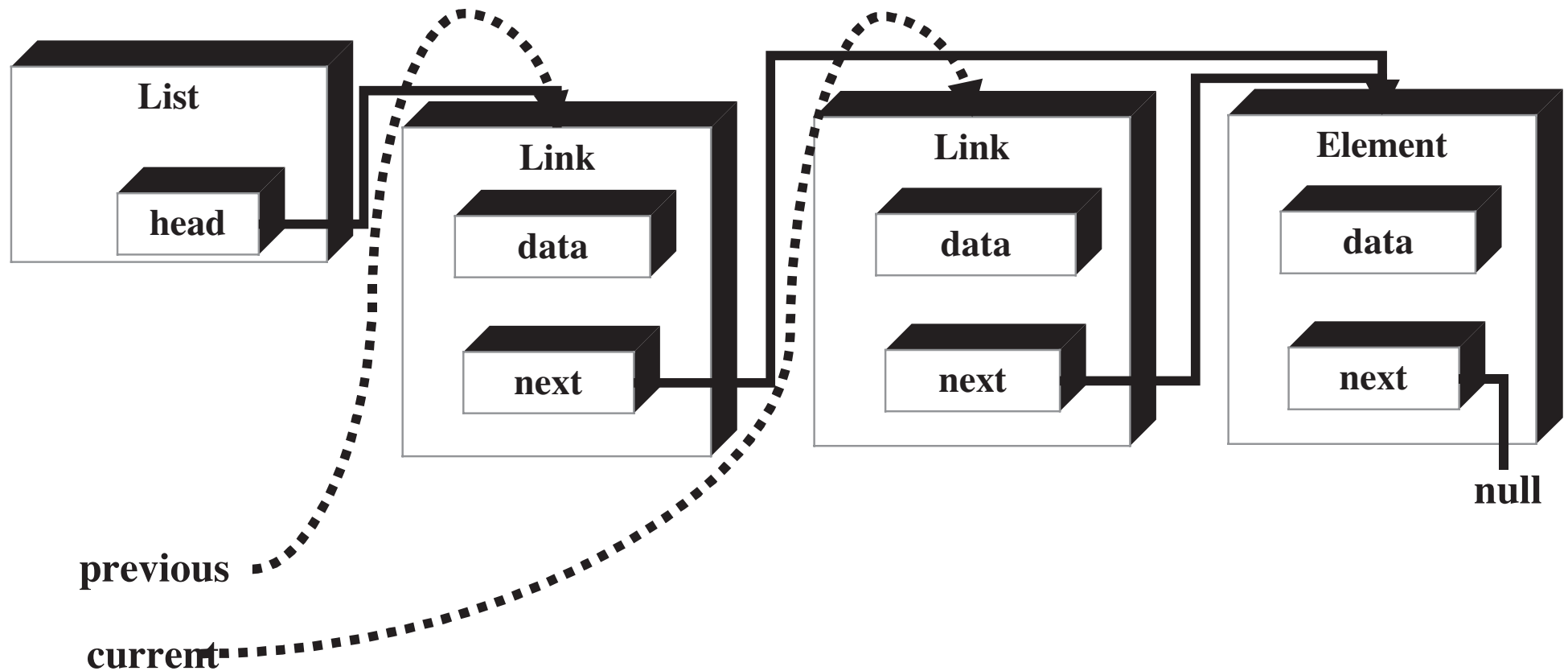
# Die "Bypass"-Operation unter dem Mikroskop

```
previous = current;           // Wir merken uns dieses Element  
current = current.getNext(); // für eventuellen "Bypass"
```



# Die "Bypass"-Operation unter dem Mikroskop

❖ `previous.setNext(current.getNext())`



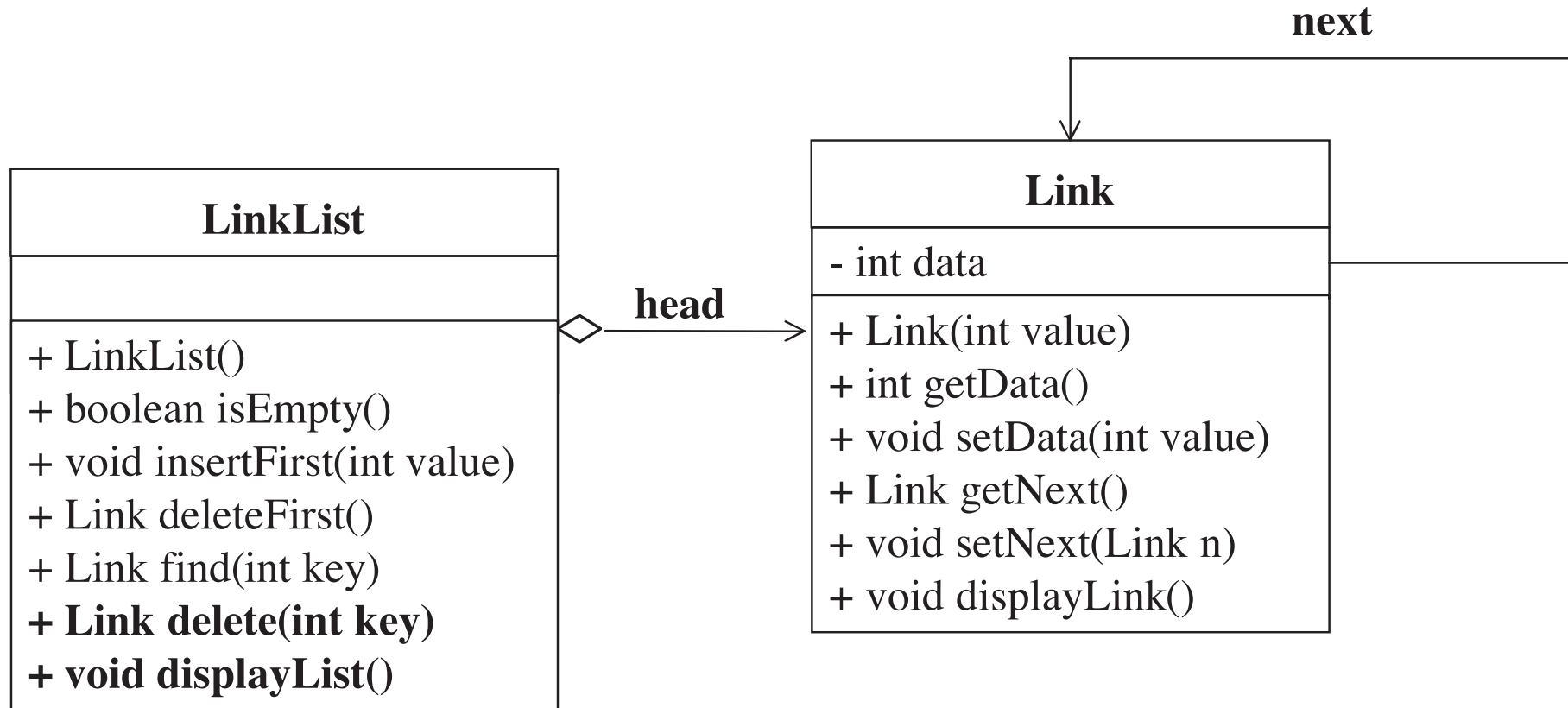


## *Drucken einer verketteten Liste*

Diese Methode wird immer nach einer Änderung der Listen aufgerufen (d.h. nach Aufrufen der Methoden insertFirst, deleteFirst, delete)

```
public void displayList() {  
    System.out.print("Liste: ");  
    Link current = head;           // Start am Anfang der Liste  
    while(current != null) {       // bis zum Ende  
        current.displayLink();     // drucke einzelnes Listenelement  
        current = current.getNext(); // auf zum nächsten Element  
    }  
    System.out.println();  
}
```

## Revidiertes Modell ("Neueste Iteration")



## *Sortierte Liste*

- ❖ In einer sortierten Listen werden Listenelemente in ansteigender Reihenfolge arrangiert; gewöhnlich wird dabei ein Schlüsselwert (key value) innerhalb des Datums als Sortierkriterium benutzt.
- ❖ Vorteile von sortierten Listen gegenüber sortierten Reihungen:
  - Sehr schnelles Einfügen von neuen Elementen, da die anderen Elemente nicht bewegt werden müssen.
  - Eine Liste kann beliebig expandieren, während eine Reihung eine feste Größe hat.
- ❖ Nachteil:
  - Sortieren von Listen ist schwieriger zu implementieren als das Sortieren von Reihungen.

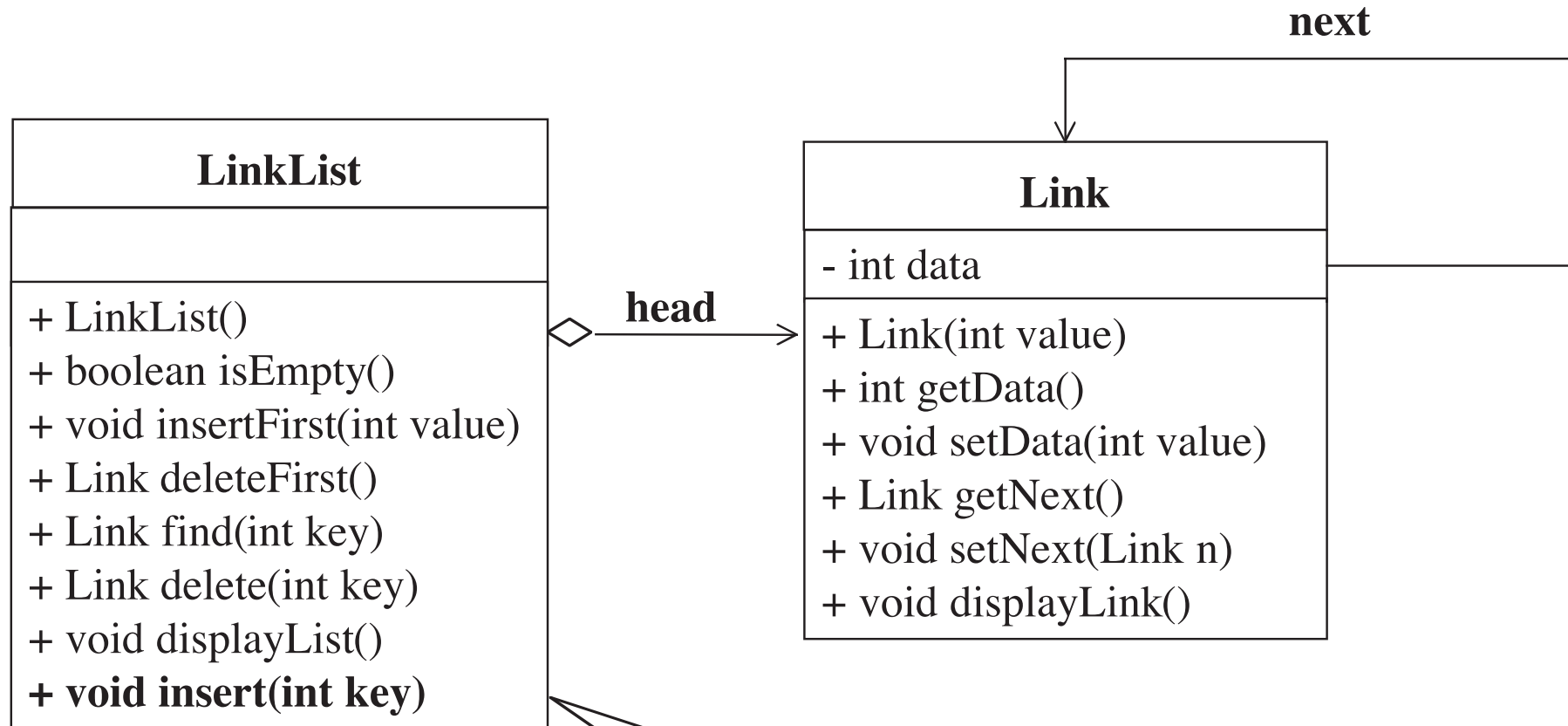
## *Java-Implementation: Einfügen in eine sortierte Liste*

```
public void insert (int key) {
    Link newLink = new Link(key);    // neues Element erzeugen
    Link previous = null;
    Link current = head;              // fang beim ersten Element an

    while((current != null) && (key > current.getData())) { // bis zum Ende der Liste
                                                                    // oder bis key < current

        previous = current;
        current = current.getNext();    // gehe zum nächsten Element
    }
    if (previous == null)                // wenn wir am Anfang der Liste stehen,
        head = newLink;                 // dann newLink als erstes Element einfügen
    else                                  // wir sind nicht am Anfang
        previous.setNext(newLink);      // Einschieben des neuen Elementes:
                                        // prev --> newLink
    newLink.setNext(current);           // newLink --> current
}
```

# Endgültiges Modell für verkettete Liste (LinkedList) und Listenelement (Link)



**Endgültiges API von LinkedList  
(bis zur nächsten Iteration:-)**

## *Komplexitätsvergleich: Verkettete Liste vs. Reihung*

- ❖ **Einfügen und Löschen am Anfang einer verketteten Liste:  $O(1)$ .**
- ❖ **Finden, Einfügen und Löschen** erfordern die Suche nach einem Element, was im durchschnittlichen Fall die Hälfte der Elemente involviert:  **$O(n)$ .**
- ❖ **Eine Reihung hat auch  $O(n)$**  für diese Operationen, aber
  - die verkettete Liste ist beim Einfügen und Löschen schneller, da keine Elemente verschoben werden müssen.
  - Geschwindigkeitsunterschied signifikant, wenn eine Kopie wesentlich länger dauert als eine Vergleichsoperation.
- ❖ **Listen nutzen Speicherplatz effizienter:**
  - Verbrauchen genau soviel Speicherplatz wie nötig.
  - Allerdings brauchen wir pro Element einen **next**-Verweis.
  - Die Größe einer Reihung muss bei der Initialisierung spezifiziert werden (evtl. mehr als benötigt).

## *Komplexität: Sortierte Liste*

- ❖ Einfügen, Suchen und Löschen von beliebigen Listenelementen erfordern im durchschnittlichen Fall  $n/2$  Vergleiche, d.h. die Komplexität ist  $O(n)$ .
- ❖ Das kleinste Element kann in  $O(1)$  eingefügt, gesucht und gelöscht werden (denn es ist immer am Anfang der Liste).
  - Wenn eine Anwendung oft das kleinste Element braucht, und schnelles Einfügen nicht so oft benötigt wird, dann ist die sortierte Liste eine sehr gute Wahl.

## *Insertionsort mit sortierten Listen*

- ❖ Sortierte Listen kann man zum Sortieren von Reihungen benutzen.
  - Nimm jedes Element der Reihung und füge es in eine sortierte Liste ein. Dadurch werden sie automatisch sortiert.
  - Entferne die Elemente aus der Liste eins nach dem anderen und kopiere sie zurück in die Reihung.
  - Damit ist die Reihung sortiert!
- ❖ **Komplexität ist  $O(n^2)$ , aber effizienter als Insertionsort:**
  - **Vergleichsoperationen  $O(n^2)$ :** Einfügen der Elemente in die sortierte Liste erfordert durchschnittlich  $n^2/4$  Vergleichsoperationen, denn jedes einzusortierende Elemente muss im Schnitt mit der Hälfte der Elemente in der Liste verglichen werden.
  - **Kopieroperationen  $O(n)$ :** Jedes Element wird nur 2mal kopiert: einmal von der Reihung in die Liste, und dann von der Liste wieder zurück in die Reihung.  
2\*n Kopien sind besser als  $O(n^2)$  Kopien im Insertionsort.

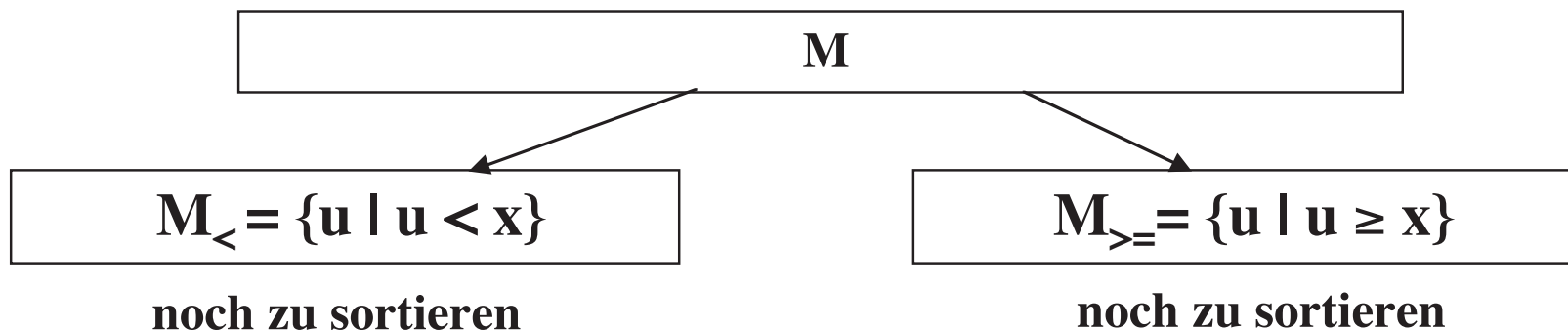


## *Vergleichstabelle: Komplexität von Operationen auf verschiedenen Datenstrukturen*

<b>Daten Struktur</b>	<b>Suchen</b>	<b>Einfügen</b>	<b>Löschen</b>	<b>Sortieren</b>
<b>Unsortierte Reihung</b>	<b><math>O(n)</math></b>	<b><math>O(1)</math></b>	<b><math>O(n)</math></b>	<b><math>O(n \log_2(n))</math></b>
<b>Sortierte Reihung</b>	<b><math>O(\log_2(n))</math></b>	<b><math>O(n)</math></b>	<b><math>O(n)</math></b>	<b>---</b>
<b>Verkettete Liste</b>	<b><math>O(n)</math></b>	<b><math>O(1)</math></b>	<b><math>O(n)</math></b>	<b><math>O(n \log_2(n))</math></b>
<b>Sortierte Liste</b>	<b><math>O(n)</math></b>	<b><math>O(n)</math></b>	<b><math>O(n)</math></b>	<b>---</b>

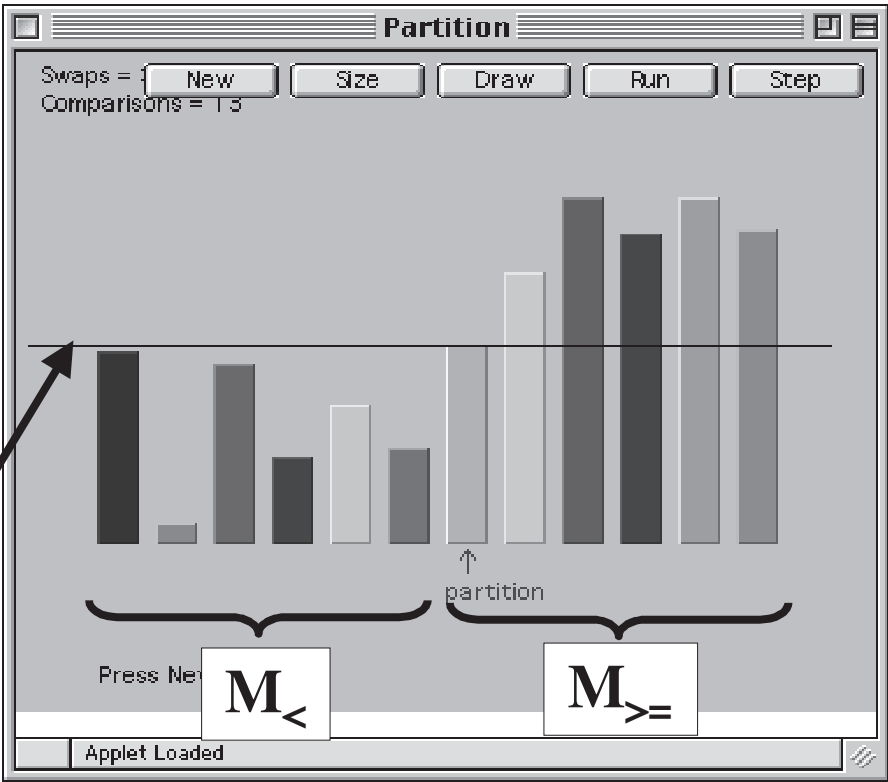
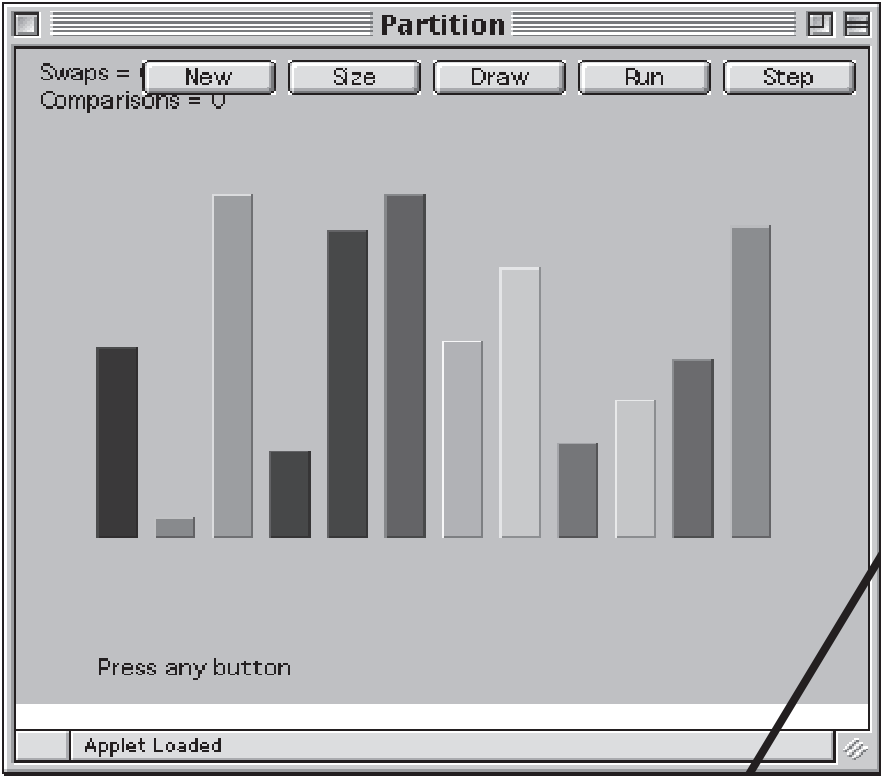
## *Ein schneller Sortieralgorithmus: Quicksort*

- ❖ Sortieren durch Zerlegen (Quicksort) wurde von C.A.R. Hoare in 1962 entdeckt.
- ❖ Sehr populär:
  - In vielen Fällen schnellster Algorithmus mit  $O(n * \log_2(n))$ .
  - Allerdings kann er in einigen Fällen zu  $O(n^2)$  degenerieren
- ❖ Quicksort benutzt die Partitionierung einer Menge  $M$  mit Hilfe eines Pivotwertes  $x$ :
  - Alle Werte kleiner als der Pivotwert kommen in eine Teilmenge  $M_{<} = \{u \mid u < x\}$ , alle Werte größer oder gleich dem Pivotwert kommen in eine Teilmenge  $M_{\geq} = \{u \mid u \geq x\}$ .



# Partitionierung einer Reihung mit einem Pivotwert

partition.html



Pivotwert

## *Klasse für Quicksort-Algorithmus*

```
class Sort {  
    private double[] m;           // die zu sortierende Reihung  
    private int nElems;          // Anzahl von Elementen in m  
  
    public Sort (int max) {       // Konstruktor  
        m = new double[max];     // erzeugt die Reihung  
        nElems = 0;              // enthält noch keine Elemente  
    }  
  
    // Implementation des Partitions-Algorithmus  
    private int partitions (int left, int right) {...}  
  
    // Implementation des Quicksort-Algorithmus  
    public void quickSort (int left, int right) {...}  
} // end class Sort
```

# Implementation des Partitions-Algorithmus

```
private int partitions (int left, int right) {
    int leftScan = left;           // links auf das erste Element
    int rightScan = right;        // rechts auf das letzte Element
    double pivot = m[right];     // wir wählen das letzte Element
                                // der Reihung als Pivot-Element

    while (leftScan < rightScan) {
        while ((leftScan < rightScan) && (m[leftScan] < pivot))
            leftScan++;           // eins weiter nach rechts
        while ((leftScan < rightScan) && (m[rightScan] >= pivot))
            rightScan--;          // eins weiter nach links
        if (leftScan < rightScan) // wenn leftScan und rightScan sich nicht kreuzen,
            swap(leftScan, rightScan); // müssen wir die beiden Elemente vertauschen
        // ansonsten sind wir mit der Partitionierung fertig
    } // end while
    if (leftScan < right)        // wenn  $M_{\geq}$  mehr als ein Element enthält,
        swap(leftScan, right);   // Pivot-Element an erste Stelle in  $M_{\geq}$  bringen
    return leftScan;             // leftScan zeigt jetzt auf 1.Element (Pivot-Element) von  $M_{\geq}$ 
} // end partitionIt()
```

- Der Algorithmus benutzt 2 lokale Variablen leftScan und rightScan als Zeiger
- Jeder dieser Zeiger zeigt auf ein Ende der Reihung
- leftScan bewegt sich nach rechts, rightScan bewegt sich nach links

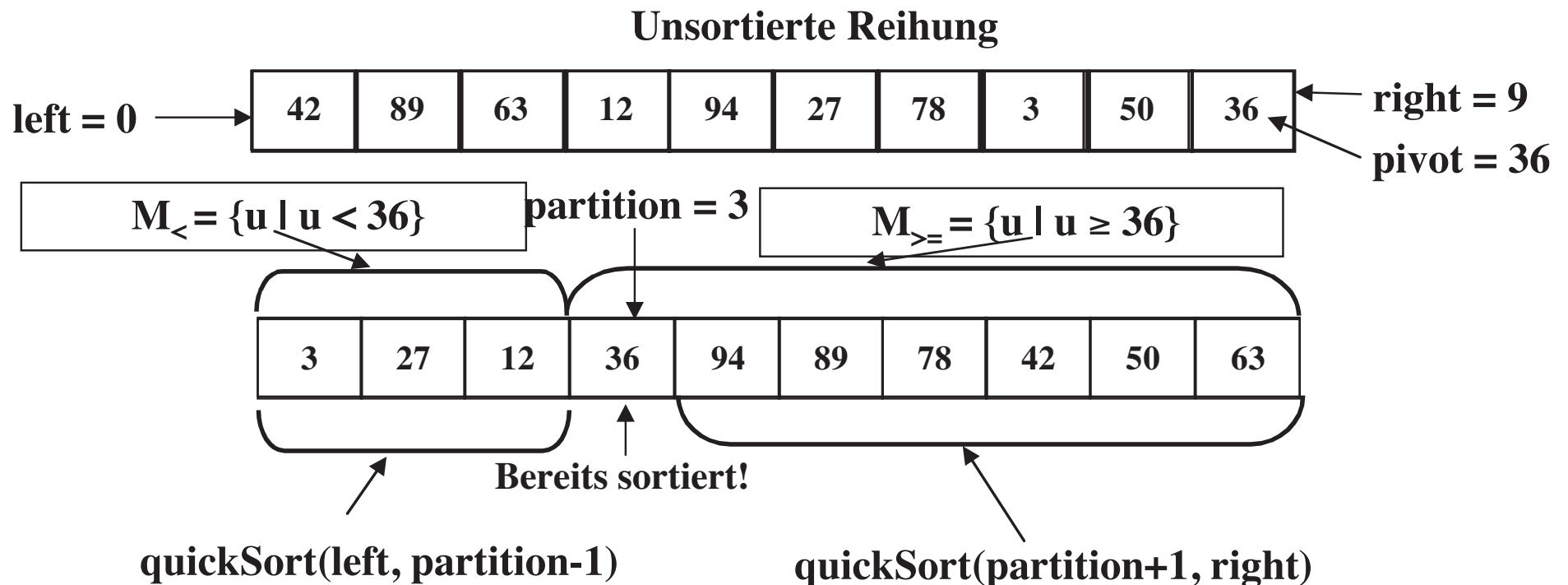
- Wenn leftScan ein Element  $\geq$  pivot sieht, dann stoppt seine while-Schleife
- Wenn rightScan ein Element  $<$  pivot sieht, dann stoppt seine while-Schleife

## *Komplexität des Partitions-Algorithmus*

- ❖ Die **Komplexität des Partitions-Algorithmus** ist  $O(n)$ .
  - Die 2 Zeiger leftScan und rightScan fangen an entgegengesetzten Enden der Reihung an, **vergleichen** Elemente mit dem Pivotwert und bewegen sich aufeinander zu, wobei sie manchmal Elemente **austauschen**. Wenn sie sich treffen, ist die Partitionierung beendet.
- ❖ **Anzahl der Element-Vergleiche  $V(n)$** :  $n$  Vergleiche mit Pivotwert
- ❖ **Anzahl der Vertauschoperationen  $T(n)$** :  $T(n) \leq n/2$ 
  - abhängig von der Wahl des Pivotwerts:
    - Anzahl der Vertauschungen  $\leq$  Länge der kleineren Partition
    - „Idealfall“: Pivot-Element = Median der Reihung (erzeugt 2 gleichgroße Partitionen)  
Reihung sortiert: letztes Element als Pivot-Element ungünstig
  - abhängig von der Elementreihenfolge („Sortierungsgrad“):
    - schlechtester Fall: Reihung invers sortiert  
Anzahl der Vertauschungen = Länge der kleineren Partition

## Idee von Quicksort

- ❖ Unsortierte Reihung mit Indizes **left = 0** und **right = 9**
- ❖ Wähle einen Pivotwert (rechtes Element der Reihung → **36**).
- ❖ **Partitioniere Reihung in 2 Teilreihungen mit Partitions-Algorithmus:**  
 $M_{<} = \{3, 27, 12\}$ ,  $M_{\geq} = \{63, 94, 89, 78, 42, 50, 36\}$
- ❖ Setze Pivot-Element an erste Stelle von  $M_{\geq}$  (Indexposition **3**)
- ❖ **Rufe Quicksort auf den Teilreihungen rekursiv auf.**



# Implementation von Quicksort

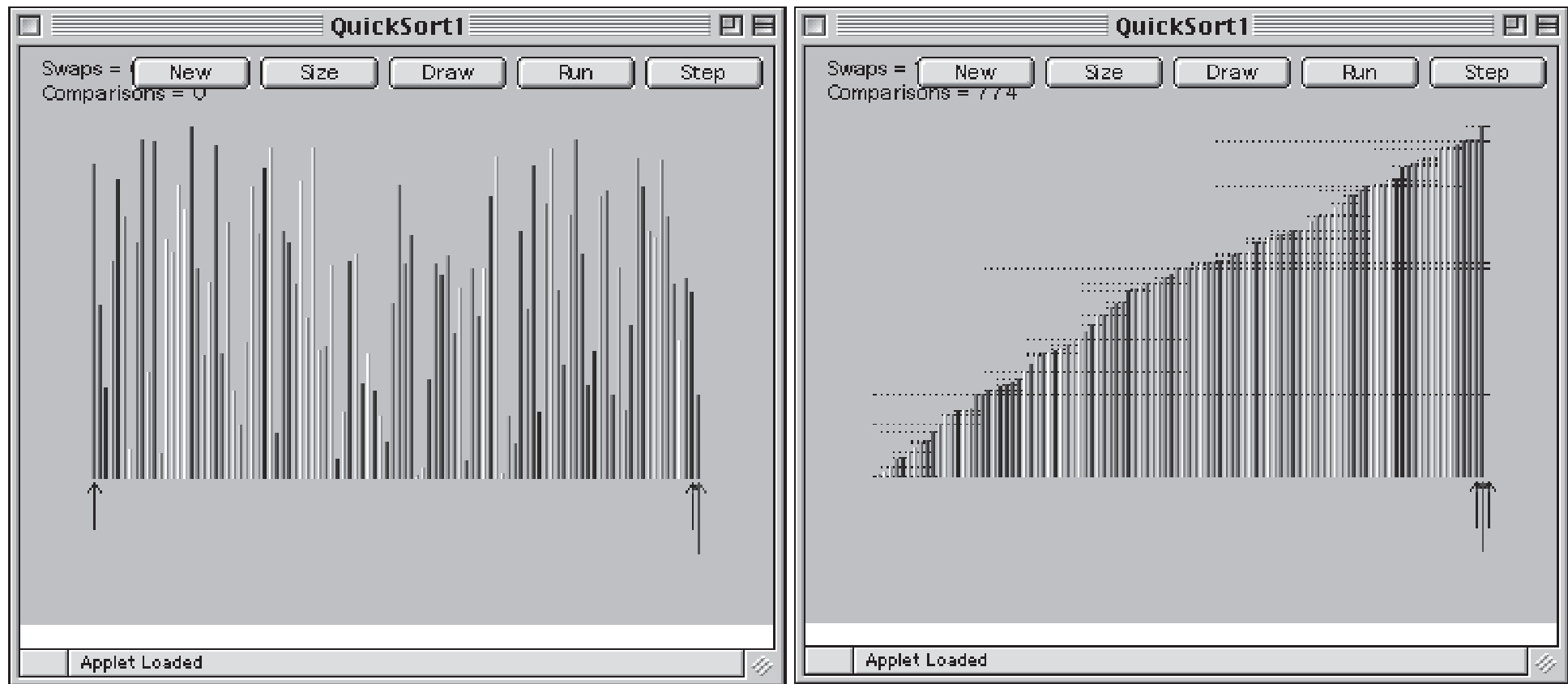
```
class Sort {  
    private double[] m;  
    private int nElems;  
    ...
```



```
    public void quickSort (int left, int right) {  
        if (right-left <= 0) // Terminierungsfall: Wenn die zu sortierende  
                             // Teilreihung höchstens ein Element hat,  
            return; // ist sie bereits sortiert  
        else { // Teilreihung enthält mindestens 2 Elemente  
            int partition = partitions(left, right); // partitioniere die Teilreihung  
            quickSort(left, partition-1); // sortiere  $M_{<}$  (linke Partition)  
            quickSort(partition+1, right); // sortiere  $M_{\geq}$  (rechte Partition)  
        }  
    } // end quickSort()  
} // end class Sort
```



# Visualisierung von Quicksort



## *Informelle Analyse der Komplexität von Quicksort*

- ❖ Um die Komplexität von Quicksort zu berechnen, bestimmen wir die Anzahl der Elementvergleiche  $V(n)$  für eine  $n$ -elementige Reihung.
- ❖ Quicksort benutzt den Partitions-Algorithmus mit **Komplexität  $O(n)$** , um 2 Teilreihungen zu erzeugen. Jede Teilreihung wird dann durch einen rekursiven Aufruf von Quicksort bearbeitet.
- ❖ Analyse für **günstigsten Fall**:
  - Nehmen wir an, dass jeder Aufruf des Partitions-Algorithmus die Reihung immer in 2 gleichgroße Teilreihungen partitioniert, und dass wir ein  $k$  haben, so dass  $n = 2^k$ .
  - Beispiel: Eine Reihung mit 128 Elementen, d.h.  $n = 128 = 2^8$ 
    - Die Längen der Teilreihungen sind dann jeweils 64, 32, 16, 8, 4, 2, 1, d.h. nach  $k = 8 = \log_2(n)$  Rekursionsebenen terminiert der Quicksort-Algorithmus.

# Quicksort: Komplexität im besten Fall (Informelle Analyse)

❖ Anzahl der Vergleiche  $V(n)$  als rekursive Gleichung geschrieben:

$$\begin{aligned}
 V(n) &= n + (2 * V(2^{k-1})) \\
 &= n + (2 * n/2) + (4 * V(2^{k-2})) \\
 &= n + n + (4 * V(2^{k-2})) \\
 &= \underbrace{n + n + \dots + n}_{k\text{-mal}} + (2^k * V(1))
 \end{aligned}$$

**k-mal**

$$= n * k + 2^k * V(1)$$

$$= n * k + 2^k * c$$

$$= n * \log_2(n) + n * c$$

$$\in O(n \log_2(n))$$

Anzahl von Vergleichen im ersten Aufruf von partitions

Komplexität von 2 Quicksorts mit  $2^{k-1}$  elementigen Teilreihungen

Anzahl von Vergleichen in partitions für 2 Quicksorts

Komplexität von 4 Quicksorts mit  $2^{k-2}$  elementigen Teilreihungen

$2^k$  Aufrufe von Quicksort mit 1-elementigen Teilreihungen

Anzahl von Vergleichen in einer 1-elementigen Teilreihung ist konstant

Annahme:  $n = 2^k$ , also  $k = \log_2(n)$

## *Komplexitätsheuristik*

- ❖ Quicksort ist ein sog. „**Teile-und-Herrsche**“-Algorithmus („**Divide and Conquer**“).
  - Derartige Algorithmen haben im allgemeinen die Laufzeitkomplexität  $O(n * \log_2(n))$