Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

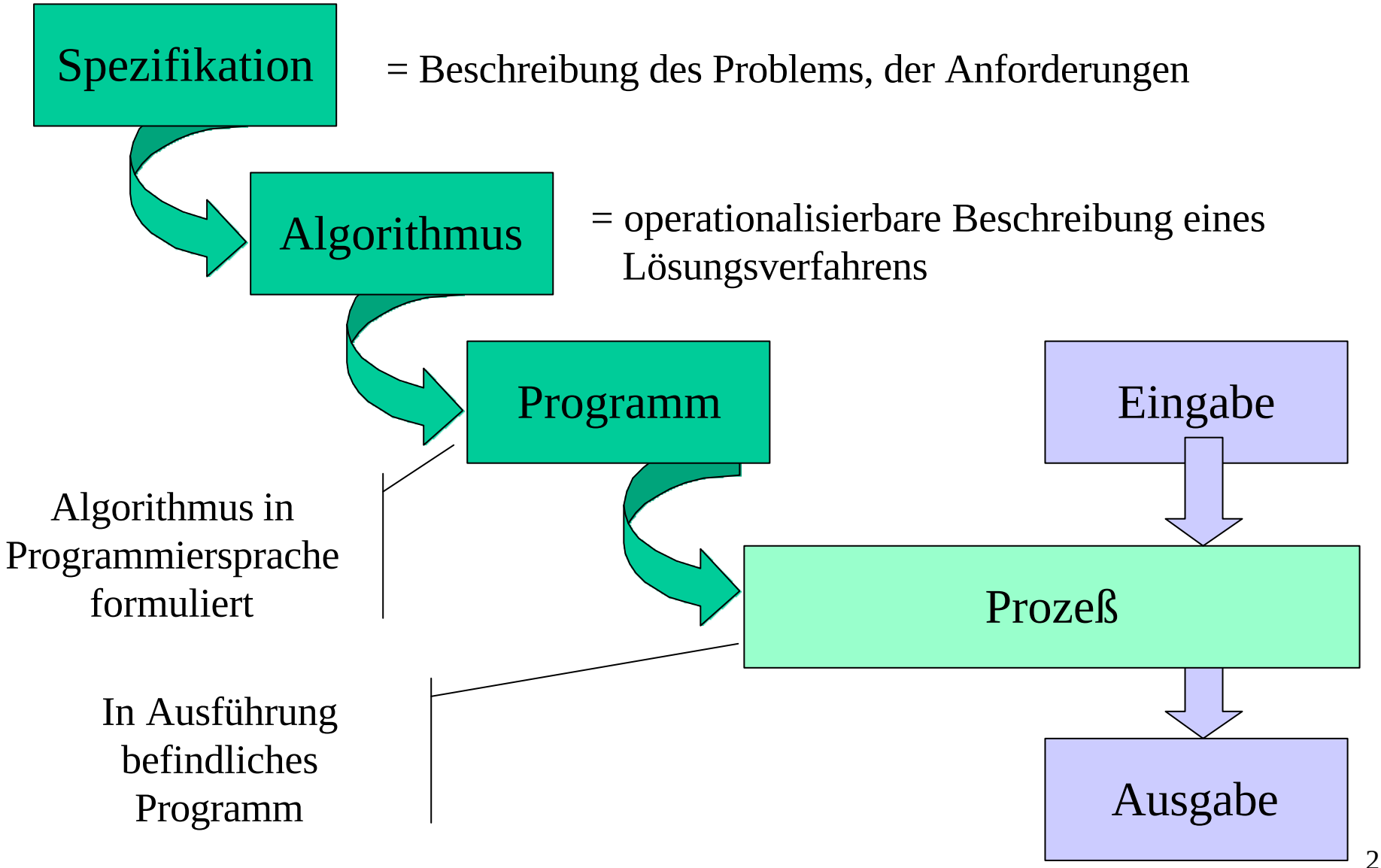
5. Vorlesungswoche

Grundlagen imperativer und objektorientierter Programmierung

Konstruktion von Datentypen: heute Arrays

Algorithmen: heute Sortieren

Stationen im Entwurf von Algorithmen und Programmen



Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Zusammenstellung gleicher Datentypen: Arrays, Felder

Motivation: Schleifen erlauben die Verarbeitung mehrerer Daten auf einen „Schlag“

Eine Entsprechung auf der Variablenseite ist die Zusammenfassung mehrerer Variablen gleichen Typs:
-> Arrays oder Felder

Beispiele:

1. Zeichenketten/Strings, Arrays aus Character/Zeichen
2. Vektoren, Matrizen: Arrays aus Integer/Float Variablen
3. Abbildung eines Lagerbestandes durch Angabe der Menge für einen Artikel and einem Lagerort

   Bei n unterschiedlichen Artikeln and m Orten:

   Bestand [1] [5] der Bestand des Artikels 1 am Ort 5

   In Java: Bestand [i] [j] Artikel mit Nummer i und Ort j

Fragen:

- Wie werden Arrays deklariert ?
- Wie werden Daten in Arrays abgelegt, verändert, ausgegeben ?
- Wie wird die Größe eines Arrays festgelegt ?

Arrays sind Variablen ...

- ... müssen daher auch deklariert werden, bevor sie benutzt werden
- ... die viele Variablen enthalten, die vom gleichen Typ sind
- ... die Anzahl der Dimensionen entspricht der Anzahl der Indexausdrücke

Deklarationen:

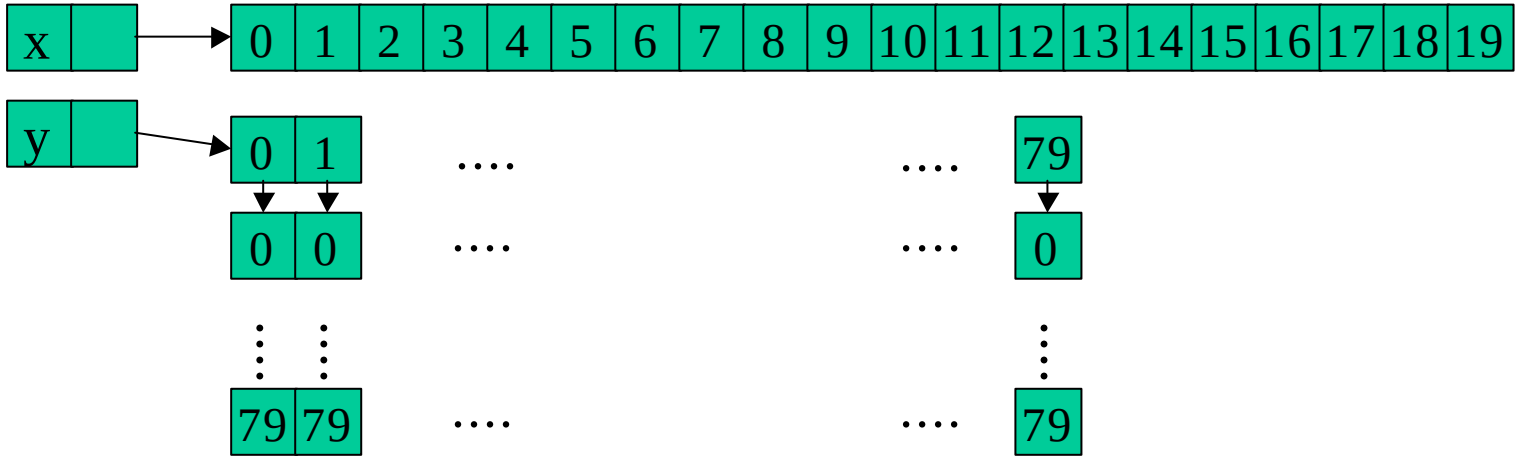
```
int [] x;    // ein-dimensional int
float [] [] y; // zwei-dimensional float
```

Legen allerdings anders als bei `int z;` noch keinen Speicherplatz fest

(©M. Goedicke, UGH Essen)

Deklarationen einer Array-Variablen legt nur einen Verweis auf eine Array fest

```
int [] x;
x = new int [20];
float [] [] y;
y = new float [80][80];
```



(©M. Goedicke, UGH Essen)

Unterscheide Zuweisung an Array - Variablen und Zuweisung an einzelnes Element

- Array-Größe ist fest nach Ausführung des new-Operators
- Veränderung der Größe nur durch Umkopieren

```
int [] a,b;
a = new int [10];
.... // Zuweisung der Elemente 0 ..9
b = new int [20];
for (int i=0; i < 10; i++) b[i] = a[i];
a = b;  // nun verweisen a und b auf das
        // gleiche Array! (siehe Strings)
...
```

Beachte: Indices immer `0..Anzahl -1`

(©M. Goedicke, UGH Essen)

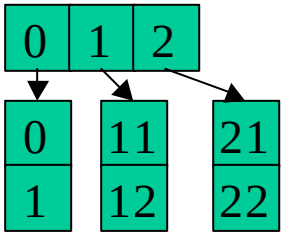
Bei der Deklaration einer Array-Variablen werden die Standardwerte zugewiesen

- Z.B.: int alles 0 ...
- Andere Variante: Belegung mit direkt angegebenen Konstanten

```
int [] m =
    {31,28,31,30,31,30,31,31,30,31,30,31};
```

- Größe und Belegung sind direkt festgelegt
  -> keine Erzeugung mittels `new` notwendig
- Funktioniert auch für mehrdimensionale Arrays:

```
int [] [] a={{0,1},{11,12},{21,22}};
```




```
a[0][0] == 0
a[0][1] == 1
a[1][0] == 11 ...
```

(©M. Goedicke, UGH Essen)

Gelegentlich ist die Größe eines Arrays nicht zum Zeitpunkt der Programmerstellung bekannt

- Die Größe könnte z.B. auch eingelesen werden
- In Java kann durch `x.length` die Anzahl (!) der Elemente dynamisch bestimmt werden:

```
int Anz = Eingabe(), Anfangsw = 0;

int [][] Matrix = new int[Anz][Anz +12];

for (int i = 0; i < Matrix.length; i++)
{
    for (int j=0; j < Matrix[0].length;j++)
    {
        Matrix[i][j] = Anfangsw;
    }
}
```

(©M. Goedicke, UGH Essen)

Beachte: Index läuft `0.. Matrix.length -1`

Bei der Zuweisung von Array-Variablen muss der Typ berücksichtigt werden

- (siehe: strenges Typssystem) ... Für einzelne Elemente eines Arrays, die selbst keine Arrays sind ist dies klar:

```
int [] a = new int [3];
a[1] = 3;
```

- Für Arrays gilt:
  Typ der Grundelemente und die Anzahl der Dimensionen muss übereinstimmen

```
int [][] a;
...
a = b; // klappt nur wenn b ebenfalls
       // 2-dimensionales int Array ist
```

(©M. Goedicke, UGH Essen)

In den eckigen Klammern sind komplexe Ausdrücke erlaubt

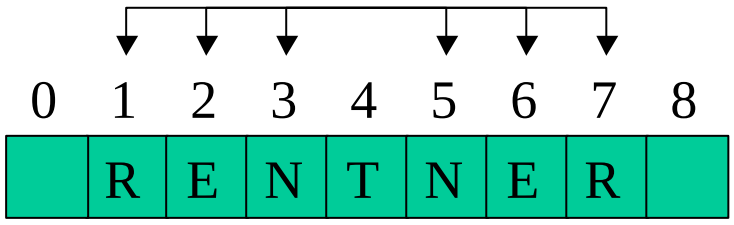
z.B.:

Bestand der Artikel 3 .. 28 am Ort 4

```
int Summe = 0;
for (int i=3; i<= 28; i++)
    Summe += Bestand[i] [4];
```

oder ein einfacher Palindrom-Test:

```
boolean Palindrom = true;
for (int i=1; i<=7/2 && Palindrom; i++)
{
    Palindrom = Zeichen[i] == Zeichen[7+1-i];
}
```



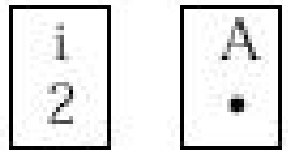
1	7
2	6
3	5

(©M. Goedicke, UGH Essen)

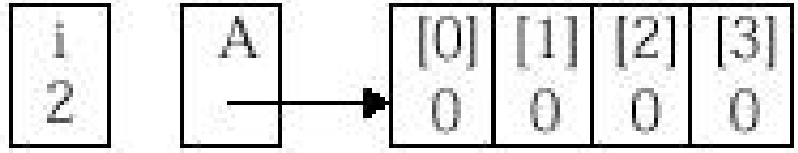
Zuweisung von Array-Variablen ... Beispiel (1)

int i = 2;

int [] A;

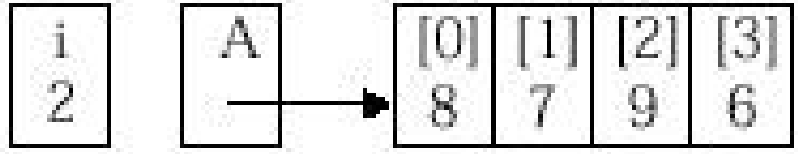


A = new int [4];



A [0] = 8; A [1] = 7;

A [i] = 9; A [3] = 6;



(©M. Goedicke, UGH Essen)

Zuweisung von Array-Variablen ... Beispiel (2)

....

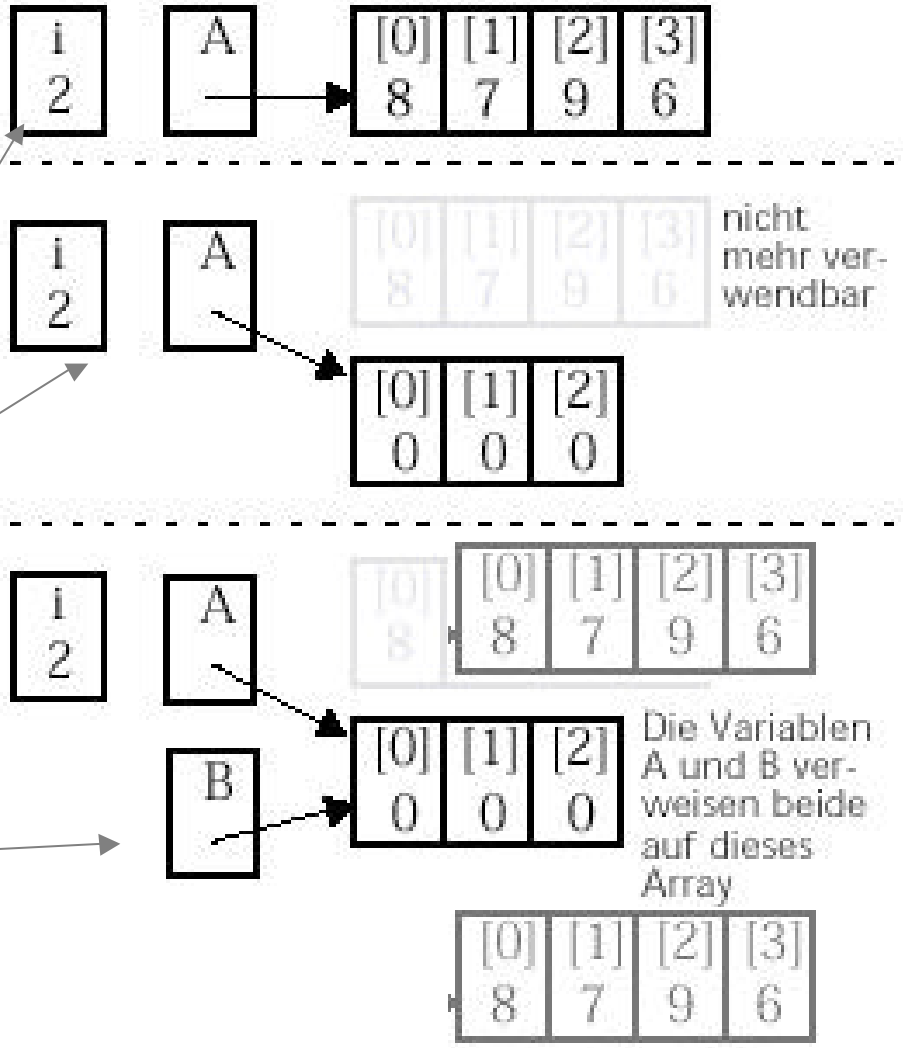
A [0] = 8; A [1] = 7;

A [i] = 9; A [3] = 6;

A = new int [3];

int [] B;

B = A;



(©M. Goedicke, UGH Essen)

Zuweisung von Array-Variablen ... Beispiel (3)

....

B = A;

A [0] = 6;

B [1] = 7;

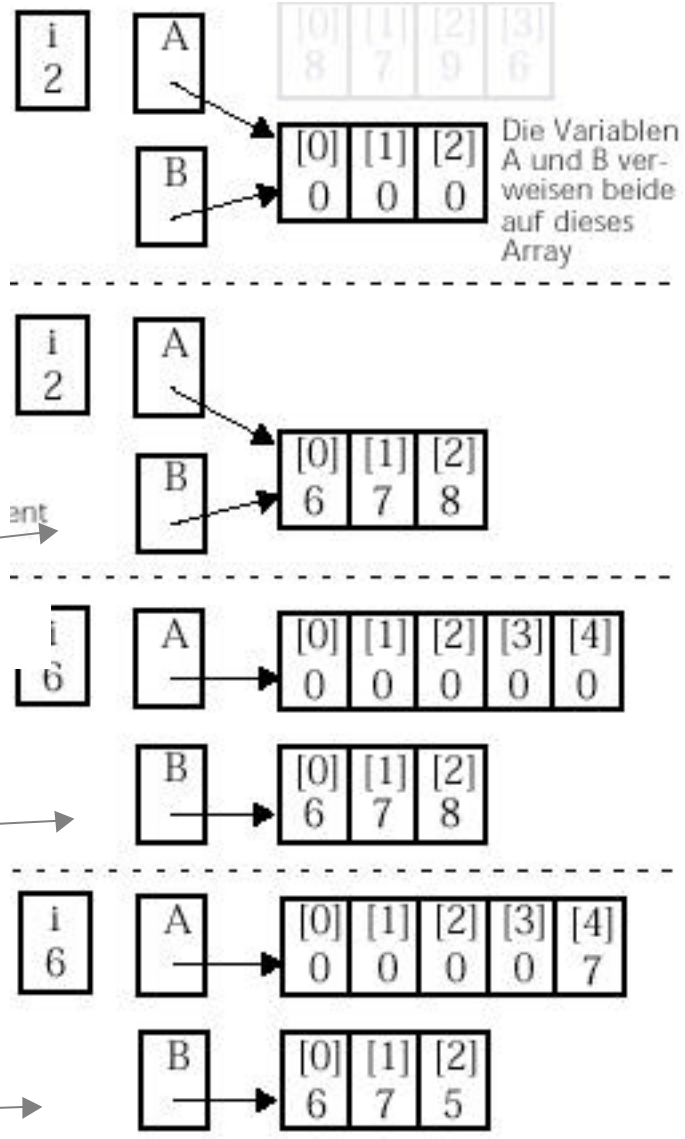
B [2] = B [0] + 2;

i = B [0];

A = new int [5];

A [i - 2] = B [1];

B [i - 4] = A.length;



(©M. Goedicke, UGH Essen)

Ein Beispiel für die Verwendung von Arrays: internes Sortieren

- Internes Sortieren bringt die Elemente einer Folge in die richtige Ordnung
- Viele Alternativen bzgl. Sortieren sind entwickelt worden
- Das einfache interne Sortieren (wie hier vorgestellt) hat geringen Speicherplatzbedarf aber hohe Laufzeit
- Verfahren:
  vertausche Elemente der Folge solange bis sie in der richtigen Reihenfolge sind
- Hier wird als Speicherungsstruktur ein Array benutzt

(©M. Goedicke, UGH Essen)

Angestrebt ist eine flexible Lösung: zunächst das Einlesen der Werte

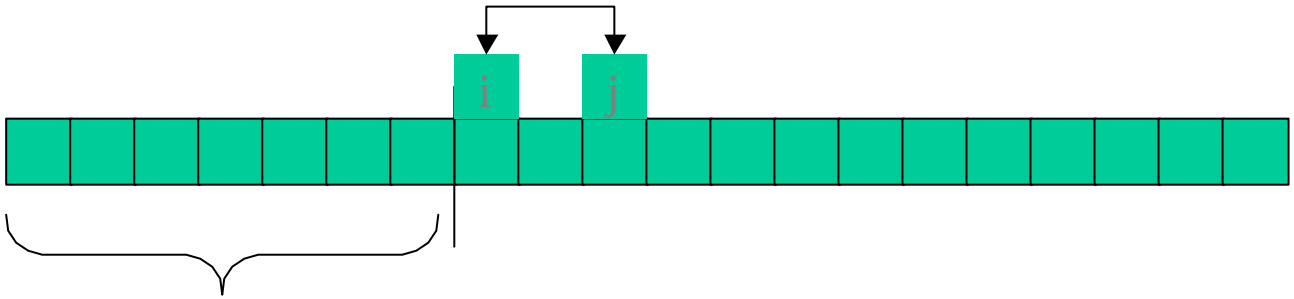
```
public class Sortierung
{ public static void main (String [] unbenutzt)
  { int i, j, z, n = Eingabe ();
    int [] a = new int [n];

    // Lies Elemente ein und gib sie dabei sofort
    // wieder aus:
    System.out.println ("Die eingegebenen Elemente:");
    for (i = 0; i < n; i++)
    {  a [i] = Eingabe ();
       System.out.print (a [i] + "  ");
    }
    System.out.println ("\n");  ...
```

(©M. Goedicke, UGH Essen)

Nun die eigentliche Sortierung ...

Die Idee:

- Ausgangspunkt: Element an der Stelle i hat den richtigen Platz,
- dann müsste eigentlich für alle Elemente an den Stellen j > i gelten, dass a[i] <= a[j]
- Wenn diese Bedingung nicht gilt: vertausche die Elemente an den Stellen i und j



Bereits sortiert,d.h.

a[k] a[k+1], f. k i-1

und a[k] a[j], f. k i-1

und j ∈ {i..n}

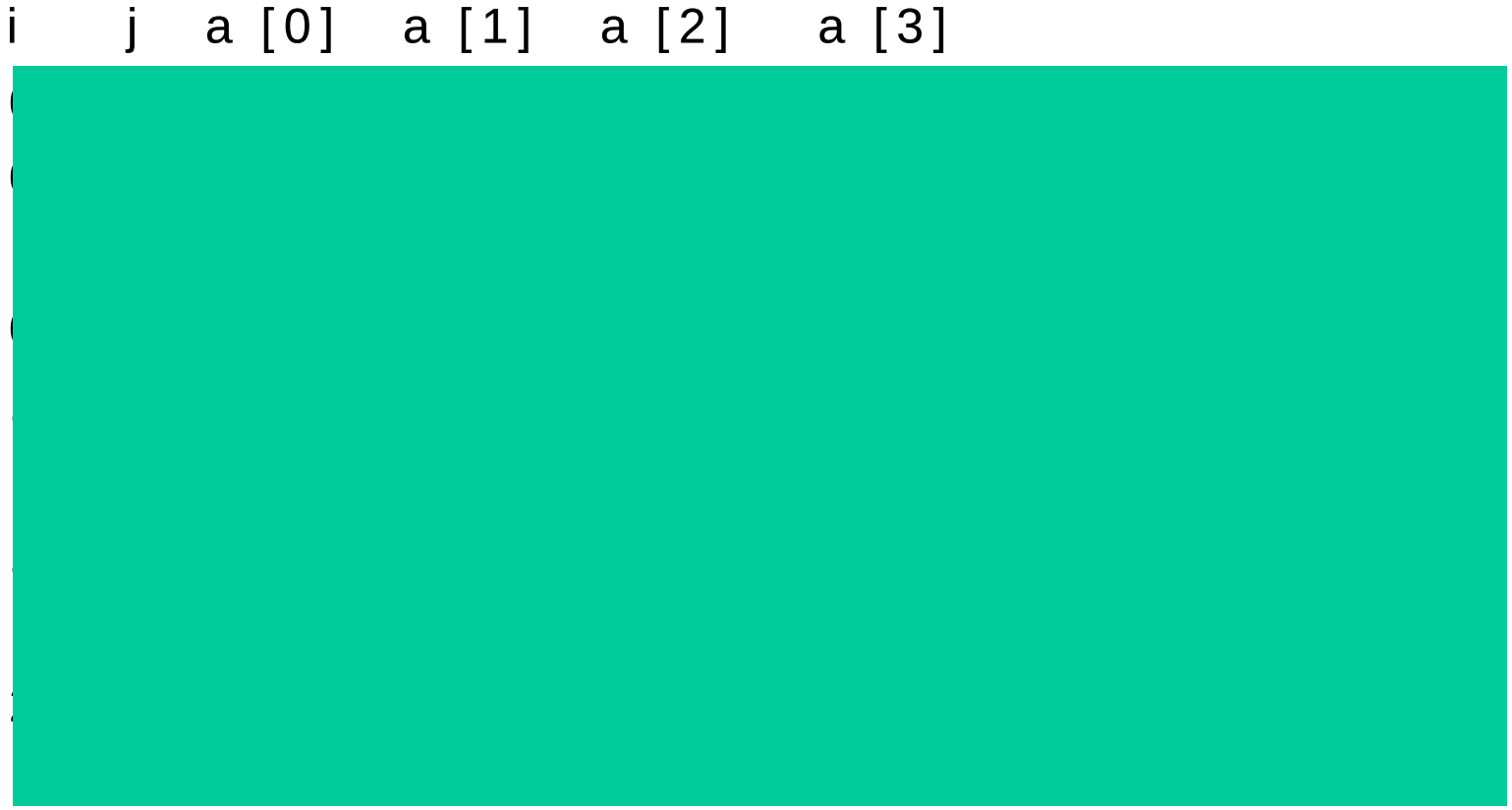
(©M. Goedicke, UGH Essen)

Diese Schritte müssen für alle Elemente im Array erledigt werden ...

```
for (i = 0; i < n - 1; i++)
{
    // Prüfe, ob a [i] Nachfolger hat, die kleiner als a [i] sind:
    for (j = i + 1; j < n; j++)
    {
      if (a [i] > a [j])
     // Ist Nachfolger a [j] kleiner als a [i] ?
      { // Vertausche a [i] mit a [j]: Ringtausch mit Hilfsvariable z
            z = a [i];
            a [i] = a [j];
            a [j] = z;
      }
    }
}
```

(©M. Goedicke, UGH Essen)

Der Ablauf noch einmal tabellarisch



(©M. Goedicke, UGH Essen)

Zum Schluss wird alles noch ausgegeben

```
// Gib sortierte Elemente aus:
   System.out.println ("Sortierte Elemente:");
   for (i = 0; i < n; i++)
     System.out.print (a [i] + "  ");

   System.out.println ("\n");
   // Zeilenvorschub.
```

Könnte man die Algorithmus Idee auch anders formulieren?
- finde Minimum x der aktuellen Menge
- positioniere x an den Anfang
- sortiere Restmenge nach Entfernen von x

Rekursive Formulierung ?

Weitere Fragen:
- Terminierung
- Korrektheit
- Aufwand, Effizienz

(©M. Goedicke, UGH Essen)

Eine Bemerkung zum Aufwand des Sortieren

- Der Aufwand wird nach Anzahl der Ausführungen von Elementaroperationen betrachtet
- Im wesentlichen sind das beim Sortieren Vergleiche und Zuweisungen
- Meist begnügt man sich mit einer vergröbernden Abschätzung
- Diese Abschätzung wird in der Regel von der Größe des Problems bestimmt: hier die Anzahl der zu sortierenden Elemente
- Obiges Sortierverfahren:
  zwei geschachtelte FOR-Schleifen, die im schlimmsten Fall über das gesamte Array der Größe n laufen
  Daher ist der Aufwand in der Größenordnung von n^2

(©M. Goedicke, UGH Essen)

Sortieren ist ein beliebtes Standardproblem der Informatik

- einfach zu verstehende Aufgabenstellung
- tritt regelmäßig auf
- Grundproblem: internes Sortieren
  - zu sortierende Menge liegt unsortiert im Speicher vor, abhängig von der Datenstruktur zur Mengendarstellung kann (im Prinzip) auf jedes Element zugegriffen werden
  - es existieren viele Algorithmen, die nach Algorithmusidee, nach Speicherplatz und Laufzeit (Berechnungsaufwand) unterschieden werden
    - Wir brauchen noch ein formales Gerüst um Speicherplatz und Berechnungsaufwand zu charakterisieren !
- Varianten:
  - externes Sortieren: Daten liegen auf externem Speichermedium mit (sequentiellem) Zugriff
  - Einfügen in sortierte Menge
  - Verschmelzen von sortierten Mengen
  - ...
- im Folgenden: effiziente Alternative zum letzten (naiven) Algorithmus: Heapsort
- Verwendung rekursiver Datenstrukturen für rekursive Algorithmen

Rekursive Datenstrukturen

Rekursion ist nicht nur ein wichtiges Hilfsmittel für die Formulierung von Algorithmen, sondern auch für die Formulierung von Datenstrukturen.

Beispiele:

Eine Liste ist ein Einzelelement gefolgt von einer Liste oder die leere Liste.

Eine Menge ist leer oder eine einelementige Menge vereinigt mit einer Menge.

Oder Bäume (dazu im folgenden mehr).

Rekursion - Bäume

Nicht-rekursive Definition von Bäumen:

Ein **Baum** B besteht aus Knoten V und Kanten E, also B = (V,E).
Knoten tragen Information, sie sind über Kanten miteinander verbunden, also E = V x V.

Kanten eines Baumes sind gerichtet, sie sind also ein Paar von Knoten (v1,v2). In einer Kante e = (v1,v2) wird v1 der **Vaterknoten** und v2 der **Sohnknoten** von e genannt.

Man kann natürlich gleichwertig von:
Elterknoten (Singular von Eltern) und Kindknoten,
Mutterknoten und Tochterknoten, ...
Englisch: parent node, child node sprechen.

Rekursion - Bäume

Im Unterschied zu den allgemeineren Graphen sind Bäume hierarchisch, das heißt, es gibt keine zyklischen Verbindungen zwischen Knotenmengen.

$$\forall\, e = (v_1,v_2) \in E :$$
$$\neg\, (\exists\, i \in \mathbb{N}, i \geq 2 \wedge v_{21}, \ldots, v_{2i} \in V \mid v_{21} = v_2,\ v_{2i} = v_1,$$
$$(v_{21},v_{22}), \ldots, (v_{2i},v_1) \in E)$$

Ein **Binärbaum (auch binärer Baum)** ist dann ein Baum, in dem jeder Knoten maximal zwei Sohnknoten hat, d.h.

$$\forall\, v \in V : |\{ v_i \in V \mid (v,v_i) \in E \}| \leq 2$$

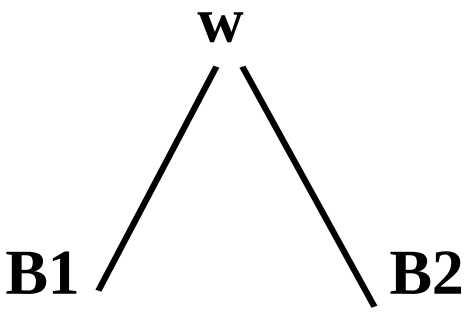
(©V. Gruhn, U Dortmund)

Rekursion - Bäume

Rekursive Definition von binären Bäumen:

Die Grundstruktur eines Baumes läßt sich leicht rekursiv definieren:

Ein **binärer Baum** ist entweder leer oder er hat die folgende Form:



wobei w ein Knoten und B1, B2 binäre Bäume sind. Zusätzlich gilt, dass ein binärer Baum nur endlich viele Knoten hat (Terminierungsbedingung!).

Rekursion - Bäume

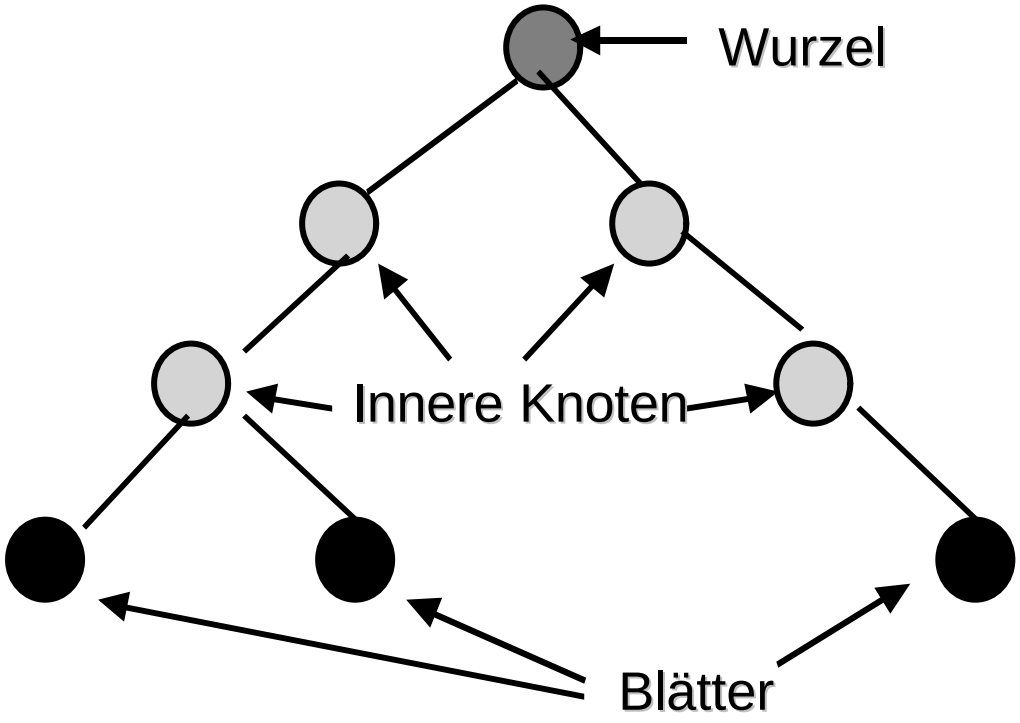
- Der Knoten *w*, mit dem B_1 und B_2 oben verbunden sind, heißt die ***Wurzel*** von *B*.

- B_1 und B_2 heißen ***linker*** bzw. ***rechter Unterbaum*** der Wurzel *w*.

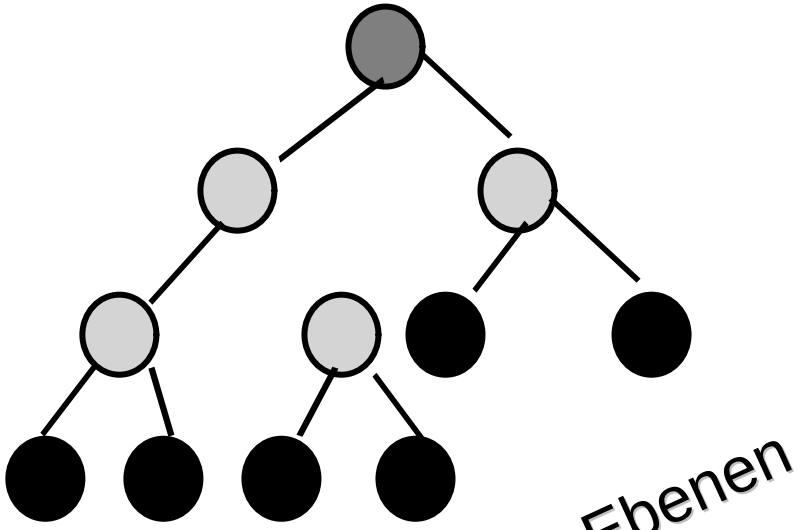
- Falls vorhanden, heißen die Wurzeln von B_1 und B_2 ***linker*** bzw. ***rechter Sohn*** der Wurzel von *B*.

- Besitzt ein Knoten weder einen linken noch einen rechten Sohn, so wird er als ***Blatt*** bezeichnet.

Rekursion - Bäume



Blätter auf einer Ebene



Blätter auf mehreren Ebenen

(©V. Gruhn, U Dortmund)

Rekursion - Bäume

Ein ***Blatt*** ist dadurch ausgezeichnet, dass der linke und der rechte Unterbaum leer sind.

Ein ***innerer Knoten*** hat die Eigenschaft, dass der rechte oder der linke Unterbaum nicht leer ist und dass es sich nicht um die Wurzel handelt.

Die ***Wurzel*** ist dadurch charakterisiert, dass sie kein Sohn eines anderen Knotens ist.

Bäume - Formalisierung der rekursiven Def.

Der leere Baum wird durch die leere Menge Ø dargestellt, seine Knotenmenge ist leer, also

$$Knoten(\emptyset) := \emptyset.$$

Seien B_1 und B_2 binäre Bäume, so dass ihre Knotenmengen disjunkt sind, es gilt also

$$Knoten(B_1) \cap Knoten(B_2) = \emptyset.$$

Sei weiterhin w ein Knoten mit $w \notin Knoten(B_1) \cup Knoten(B_2)$, dann heißt

$$B := (w, B_1, B_2)$$

ein binärer Baum mit der Knotenmenge

$$Knoten(B) := \{w\} \cup Knoten(B_1) \cup Knoten(B_2),$$

der Wurzel w, linkem Unterbaum B_1 und rechtem Unterbaum B_2.

Innere Knoten und Blätter lassen sich in dieser Formulierung ebenfalls mathematisch präzise beschreiben.

Rekursion - Auf dem Weg zu Heapsort

Problem: Gegeben sei ein Feld **a** mit **n** Werten, die sortiert werden müssen.

Ausgangssituation: Werte im Feld sind unsortiert

Ziel: sortiertes Feld

Idee 1: kleinsten Wert suchen, diesen mit dem ersten Wert tauschen, fast immer sehr langsam

Idee 2: benachbarte Werte tauschen, wenn der zweite Wert größer ist. Solange bis alles sortiert ist. Langsam, wenn von vorne nach hinten durchgetauscht werden muß.

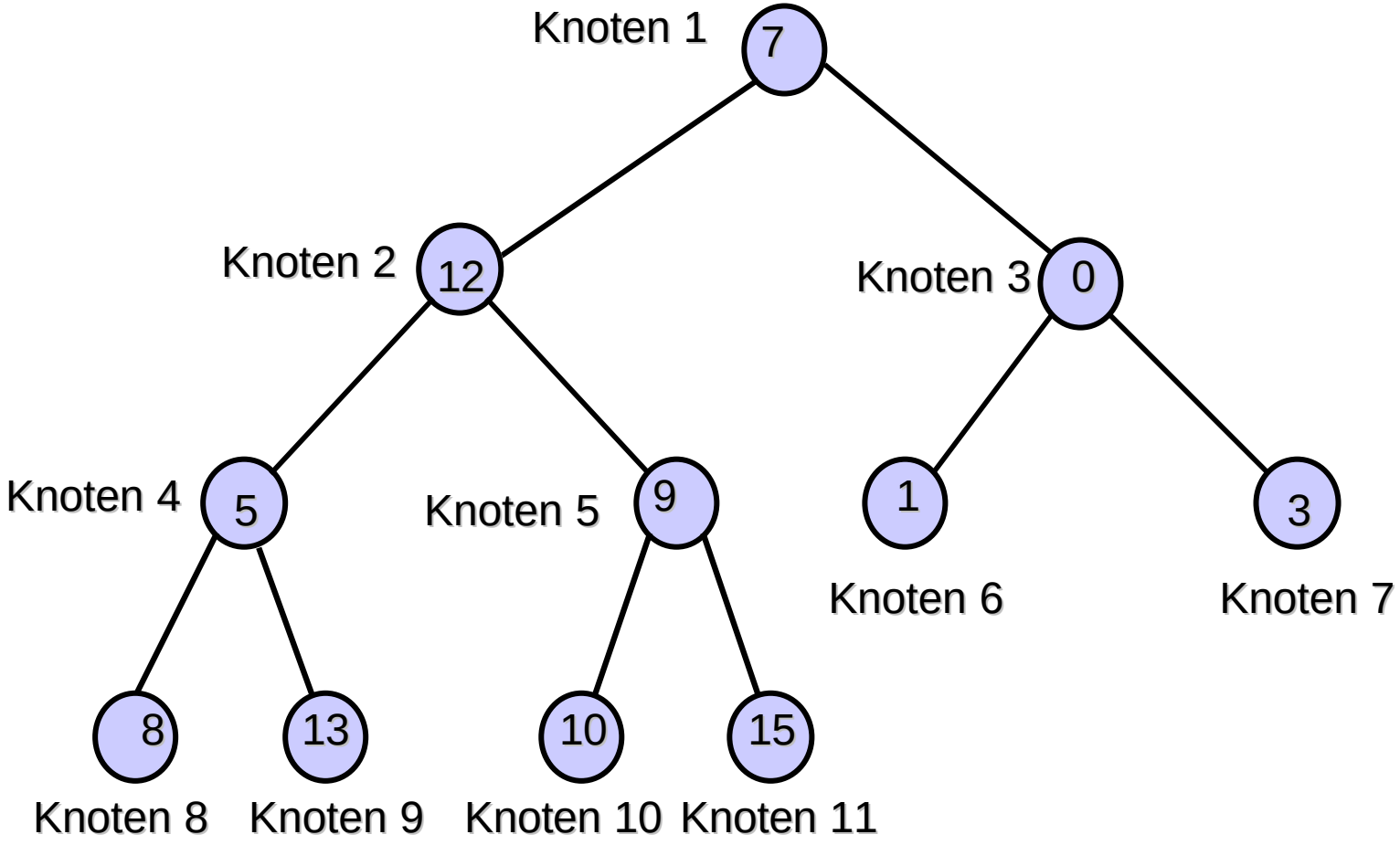
Idee 3: Verwendung eines Binärbaums (Heapsort)

Rekursion - Auf dem Weg zu Heapsort

Wir verwenden die Zahlen **`1 ... n`** zur Nummerierung der Knoten eines binären Baums: **`1`** bildet die Wurzel, und der Knoten **`i > 1`** hat **`i/2`** als Vater, so dass der linke Sohn eines Knotens gerade und der rechte Sohn eines Knotens ungerade ist, sofern der Knoten diese Söhne hat.

Damit können wir ein Feld ebenfalls als Baum darstellen: Ist das Feld **`a`** mit **`n`** Komponenten gegeben, so beschriften wir den Knoten **`i`** mit dem Feldelement **`a[i].`** Im folgenden Beispiel haben wir das Feld mit den Elementen **`7, 12, 0, 5, 9, 1, 3, 8, 13, 10, 15`** als binären Baum dargestellt. Die Knoten **`i`** werden mit ihrer Beschriftung **`a[i]`** angegeben.

Rekursion - Auf dem Weg zu HeapSort



(©V. Gruhn, U Dortmund)

Rekursion - Auf dem Weg zu Heapsort

Ein Feld genügt der *Heap-Bedingung* im Knoten `i`, falls im Unterbaum mit Wurzel `i` jeder Knoten eine kleinere Beschriftung als seine Söhne trägt.

Die Heap-Bedingung ist in den Knoten 3, 4, 5, 6, 7, 8, 9, 10, 11 erfüllt, nicht jedoch im Knoten 2, da

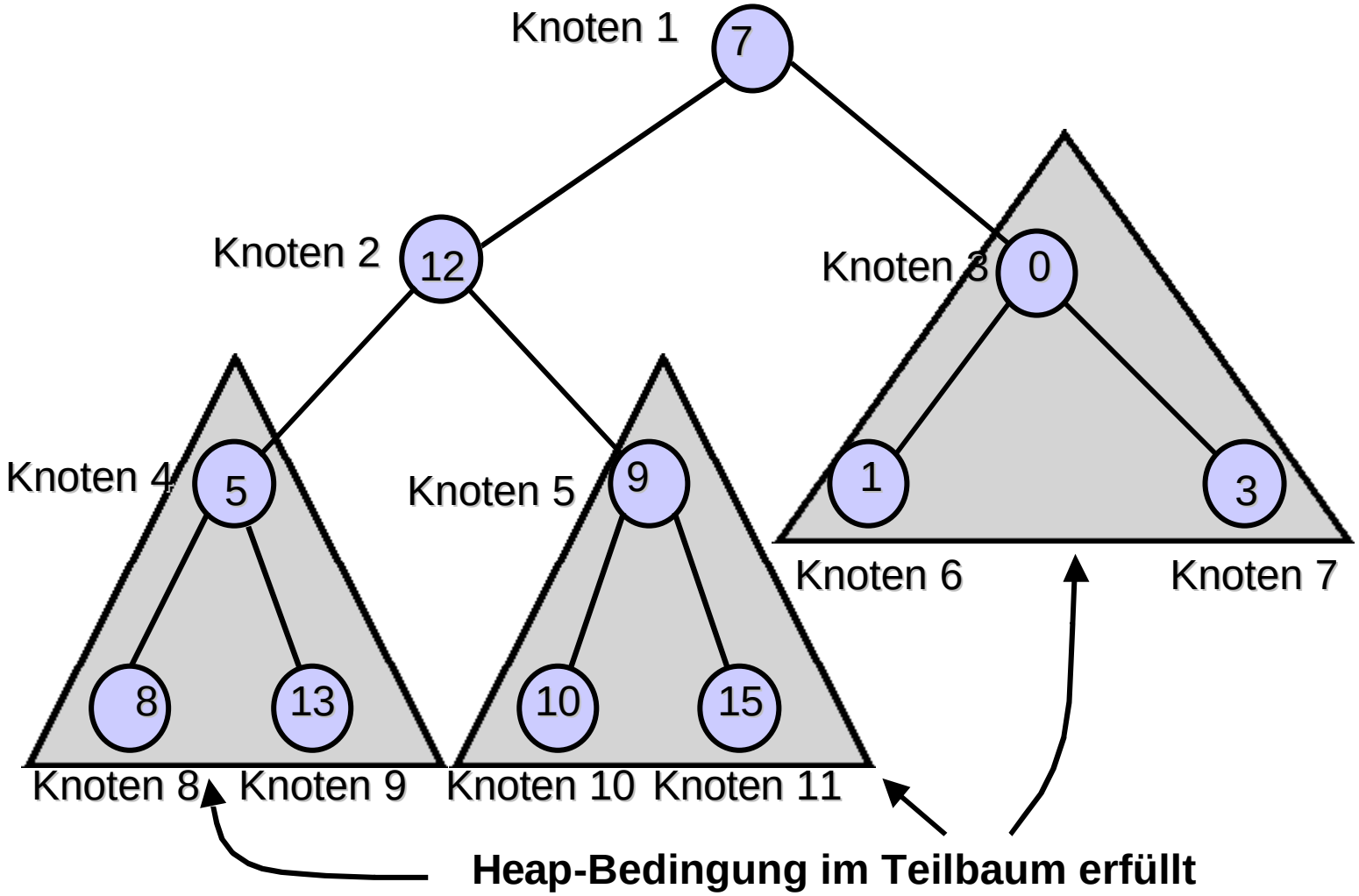
`a[2] = 12`, **`a[4] = 5`** und **`a[5] = 9`**

aber **`12 > 5`** und **`12 > 9`**,

und auch nicht im Knoten 1, da

`a[3] = 0` und **`7 > 0`**.

Lokale Gültigkeit der Heap-Bedingung



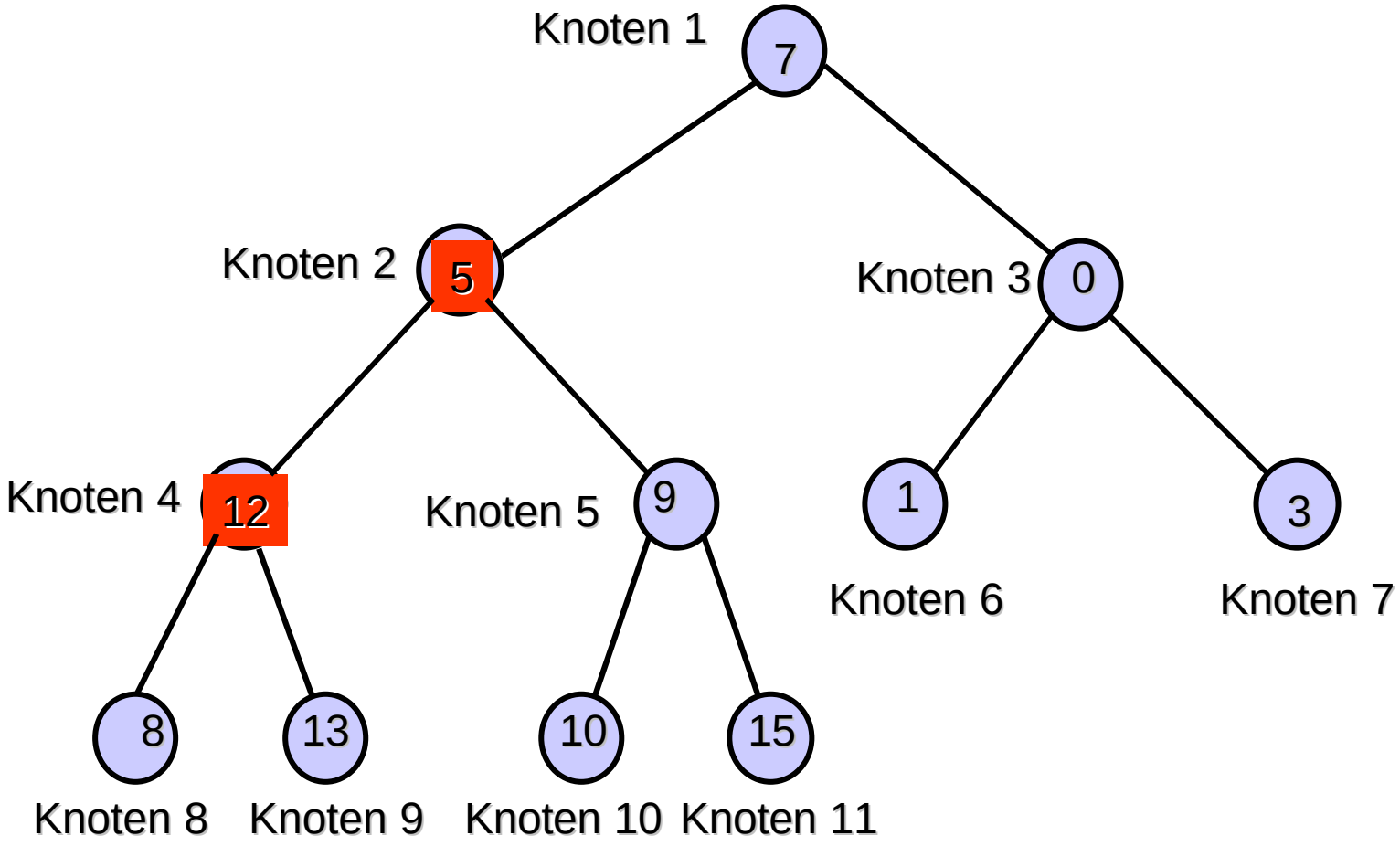
(©V. Gruhn, U Dortmund)

Rekursion - Auf dem Weg zu Heapsort

Die Heap-Bedingung kann jedoch im Knoten 2 wie folgt hergestellt werden:

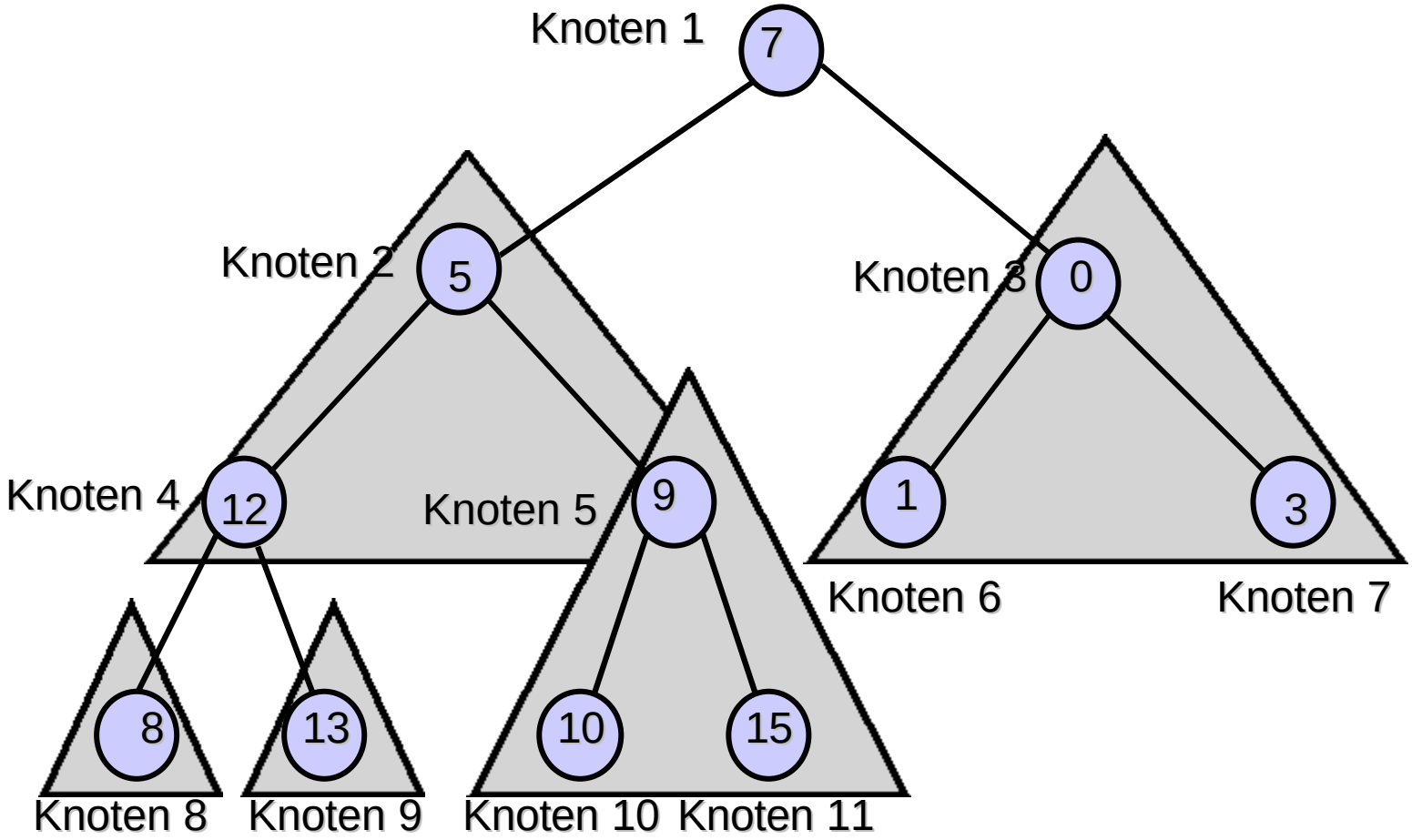
> Wir vertauschen die Beschriftung im Knoten 2 mit der Beschriftung desjenigen Sohnes, der die kleinere Beschriftung trägt. In diesem Fall handelt es sich um den Knoten 4. Wir nehmen hierzu die kleinere der Beschriftungen der Söhne, damit im Knoten 2 die Heap-Bedingung *lokal* erfüllt ist: Die Beschriftung des Knotens 2 ist kleiner als die Beschriftungen seiner Söhne (Ergebnis vgl. folgende Folie).

Rekursion - Auf dem Weg zu Heapsort



(©V. Gruhn, U Dortmund)

Herstellung der lokalen Heap-Bedingung im Knoten 2



(©V. Gruhn, U Dortmund)

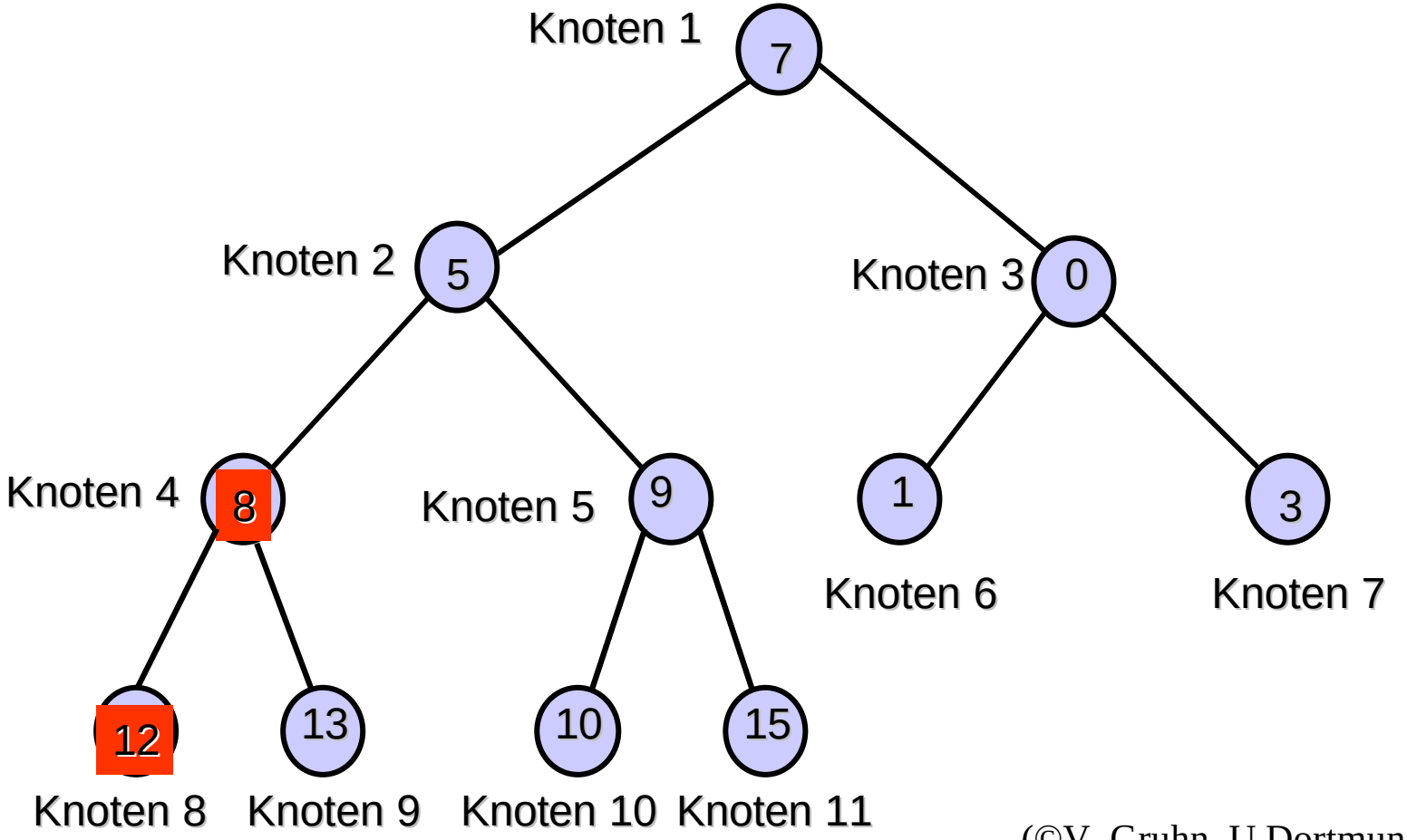
Rekursion - Auf dem Weg zu Heapsort

Nun ist zwar die Heap-Bedingung im Knoten 2 lokal hergestellt haben. Als Folge ist sie aber im Knoten 4 verletzt ist!

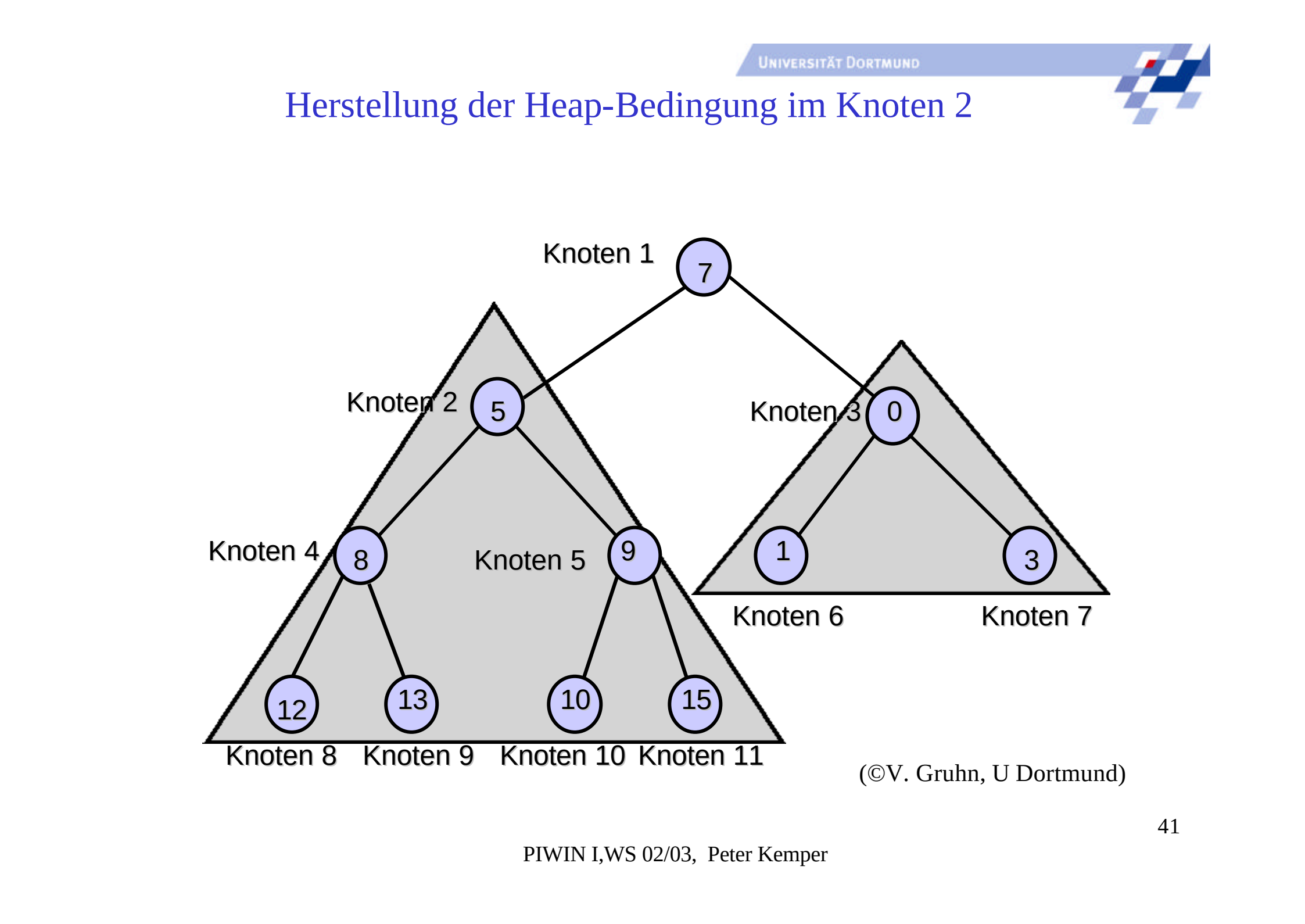
Die Vertauschung der Beschriftung eines Knotens mit der kleineren Beschriftung eines seiner Söhne, lässt sich an dieser Stelle wiederholen, so dass die Beschriftung 12 in ein Blatt, den Knoten 8, wandert.

Trivialerweise ist die Heap-Bedingung in einem Blatt erfüllt, so dass wir durch das beschriebene Vorgehen schrittweise dafür gesorgt haben, dass die Heap-Bedingung im Knoten 2 hergestellt wurde.

Rekursion - Auf dem Weg zu Heapsort



(©V. Gruhn, U Dortmund)



Herstellung der Heap-Bedingung im Knoten 2

(©V. Gruhn, U Dortmund)

Rekursion - Auf dem Weg zu Heapsort

Die Heap-Bedingung gilt aber immer noch nicht für den Knoten 1, da die Beschriftungen beider Söhne, der Knoten 2 und 3 kleiner sind als die Beschriftung der Wurzel des Baumes.

Rekursion - Heapsort

Definition: Das Feld **`a[1], ..., a[n]`** heißt ein ***Heap***, falls die Heap-Bedingung im Knoten 1 erfüllt ist. Hieraus ergibt sich unmittelbar als Konsequenz, dass das kleinste Element in einem Heap immer in der Wurzel, d.h. in **`a[1],`** steht.

Der populäre Sortieralgorithmus ***Heapsort*** arbeitet in zwei Phasen:

Es ist ein Feld **a** mit **n** Elementen gegeben.

In der ersten Phase wird dieses Feld in einen Heap umgestaltet, d.h. es wird dafür gesorgt, daß der entsprechende binäre Baum die Heap-Bedingung im Knoten 1 erfüllt.

In der zweiten Phase wird aus dem zuvor aufgebauten Heap systematisch ein geordnetes Feld erzeugt.

Rekursion - Heapsort

1. Heap-Aufbau:

Idee: Sei in **`i`** die Heap-Bedingung nicht erfüllt (**`i`** ist also kein Blatt), wohl aber in allen Söhnen.

Es werden folgende Vertauschungen vorgenommen:

`falls 2*i < n, 2*i + 1 > n`:
vertausche **`a[i]`** mit **`a[2*i]`.** Da **`i`** nur einen Sohn hat, ist die Heap-Bedingung in diesem Knoten erfüllt.

`falls 2*i < n, 2*i + 1 < n`:

suche den Sohn **`k, k Î {2i, 2i+1},`** so daß gilt:
`a[k] = min {a[2*i], a[2*i + 1]},`
vertausche **`a[i]`** mit **`a[k]`** und wende rekursiv diese Idee auf **`k`** an.

Rekursion - Heapsort

Dies führt zum Algorithmus **`Heapify`**, der rückwärtsgehend für jeden Knoten **`k = n/2, ..., 1`** die Heap-Bedingung erfüllt.

Rückwärtslaufen ist nötig, denn der Algorithmus sorgt dafür, dass aus kleineren Heaps größere aufgebaut werden.

Wenn wir in einem Knoten sind, dessen Unterbäume beide die Heapbedingung erfüllen, so wird durch die Bewegung des Knotenelements ja lediglich einer der Unterbäume modifiziert, der andere Unterbaum bleibt in diesem Durchlauf unverändert.

Rekursion - Heapsort

Methode **`Heapify`** hatn als Eingabe den Knoten **`dieserKnoten`** und die aktuelle **`heapGröße`** des Feldes.

Im ersten Schritt werden zunächst der linke und der rechte Sohn bestimmt. Danach findet eine Fallunterscheidung statt.

1. Fall: Der linke Knoten liegt noch im Feld, aber der rechte Knoten liegt nicht mehr im Feld: Es muss lediglich getestet werden, ob die Beschriftung des Knotens größer ist als die Beschriftung seines einzigen Sohnes.

2. Fall: Rechter und linker Sohnknoten liegen im Feld:

Der Knoten mit der kleineren Beschriftung muss bestimmt werden und die Beschriftung dieses Knotens mit der Beschriftung des eingegebenen Knotens verglichen werden. Falls es sich hierbei herausstellt, dass die Heap-Bedingung verletzt ist, wird mit Hilfe der elementaren Methode **`Tausche`** der Inhalt dieses Knotens mit dem seines Sohnes vertauscht und **`Heapify`** mit dem Sohn und der Heapgröße erneut aufgerufen.

Rekursion - Heapsort

```
void Heapify(int dieserKnoten, int heapGröße) {
   int links = 2 * dieserKnoten,
       rechts = links + 1,
       derSohn;
   if (links <= heapGröße && rechts > heapGröße) {
      if (a[dieserKnoten] > a[links])
         Tausche(dieserKnoten, links);
   }
   else {
      if (rechts <= heapGröße) {
         derSohn = (a[links]< a[rechts])? links: rechts;
         if (a[dieserKnoten] > a[derSohn] ) {
            Tausche(dieserKnoten, derSohn);
            Heapify(derSohn, heapGröße);
         }
      }
   }
}
```

(©V. Gruhn, U Dortmund)

Rekursion - HeapSort

Nachdem **`Heapify`** in der beschriebenen Art aufgerufen worden ist, stellt das Feld a einen Heap dar, der jetzt dazu herangezogen werden kann, das Feld zu sortieren.

Wir hatten gerade festgestellt, dass das kleinste Element in der Wurzel liegt. Also vertauschen wir zunächst **`a[1]`** mit **`a[n].`**

Damit haben wir bereits ein Element des Feldes, das kleinste, sortiert. Um das zweitkleinste Element zu bestimmen, sorgen wir nun dafür, daß die restlichen **`n-1`** Elemente wieder einen Heap bilden.

Anschließend können wir dessen Wurzel und **`a[n-1]`** miteinander vertauschen und haben bereits zwei Elemente des Feldes sortiert, und so geht es weiter, bis alle Elemente sortiert sind.

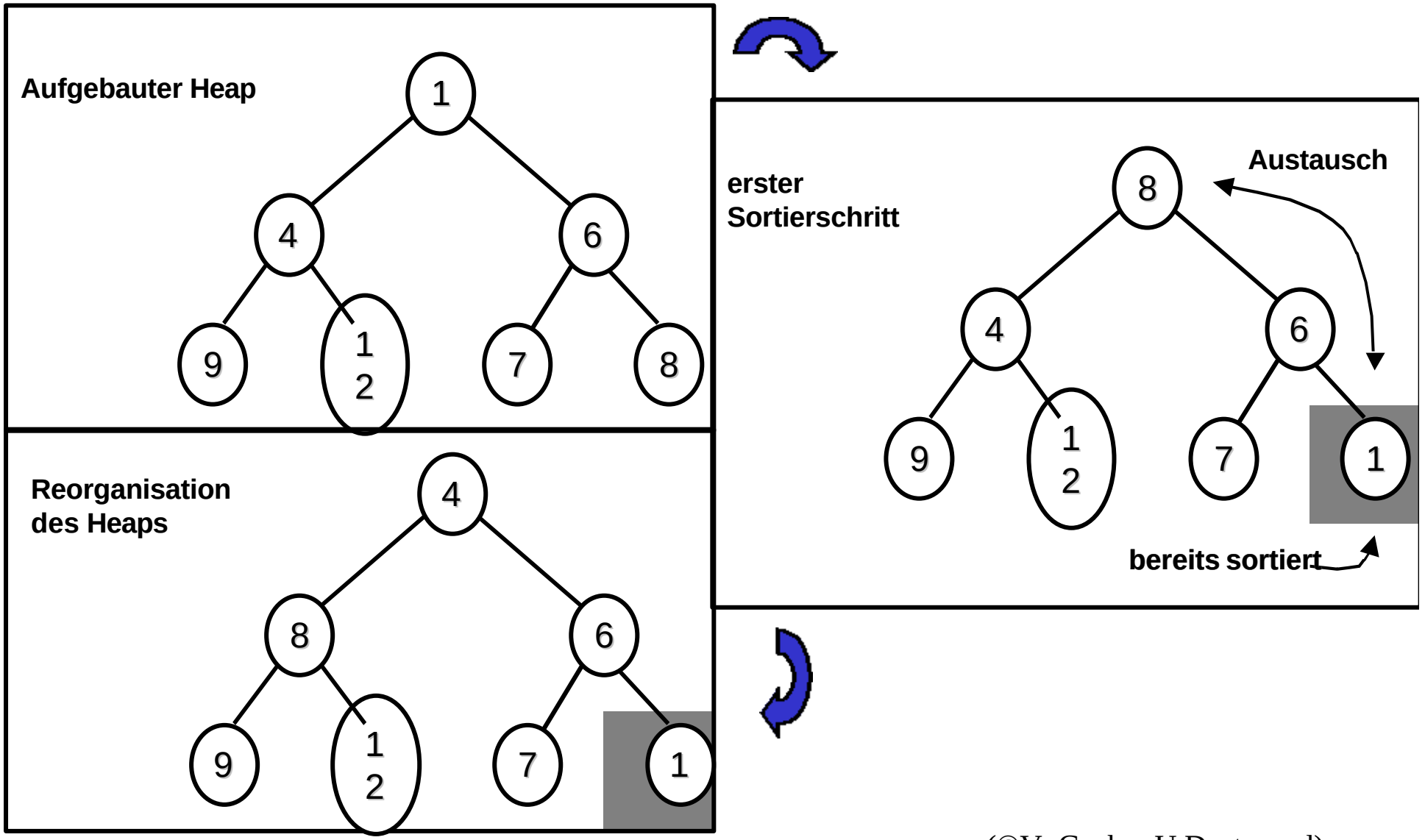
Rekursion - HeapSort

Auf der folgenden Folie wird dieser erste Sortierschritt an einem Beispiel gezeigt.

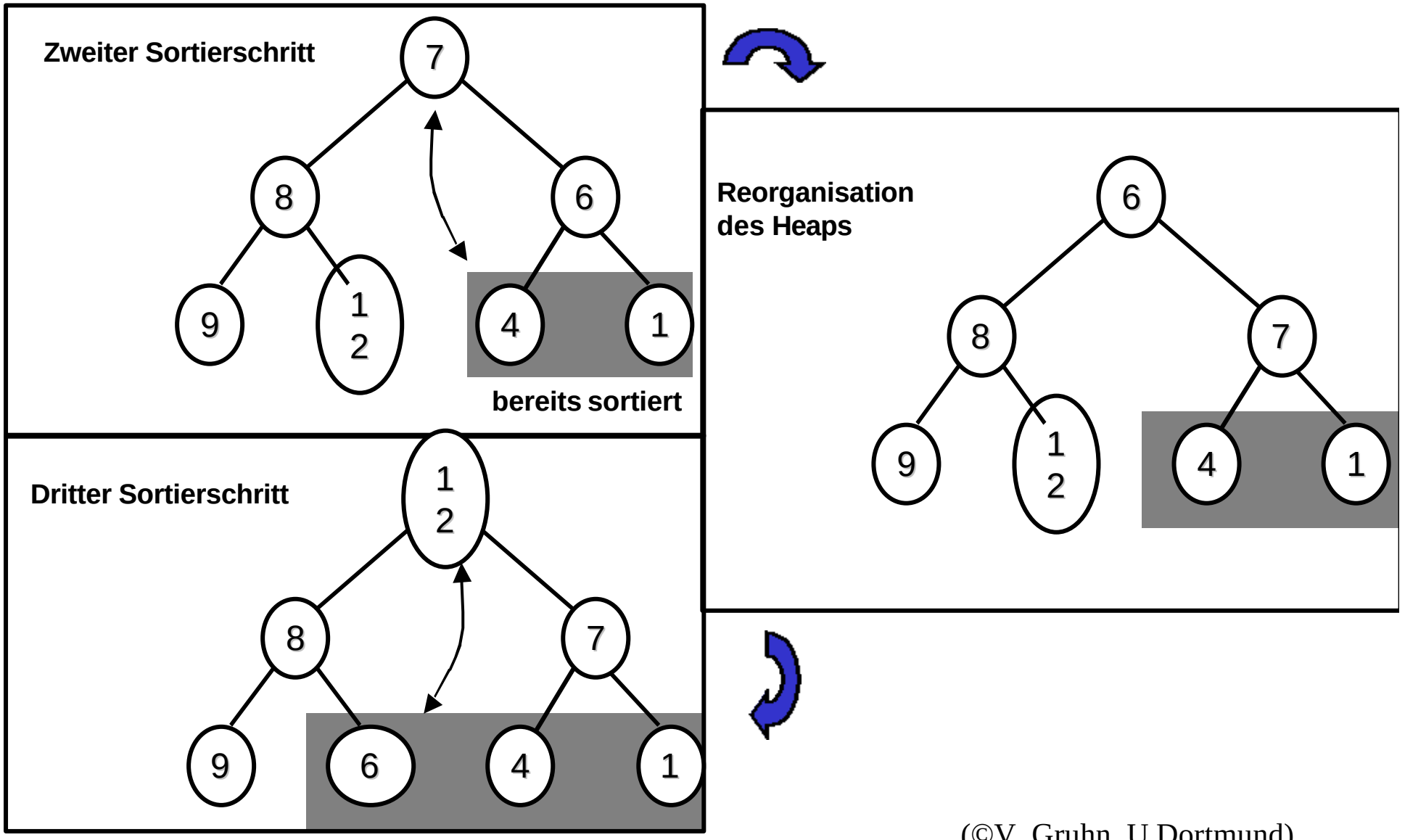
Die übernächste Folie demonstriert dann die nächsten beiden Sortierschritte, nach deren Ausführung bereits die drei letzten Elemente des Feldes sortiert sind, so dass nun nur noch n-3 Elemente betrachtet werden müssen.

Das Feld ist vollständig sortiert, wenn der im nächsten Sortierschritt noch zu betrachtende Heap nur noch aus der Wurzel besteht.

Sortieren mit einem Heap - erster Sortierschritt



Sortieren mit einem Heap - 2. / 3. Sortierschritt



Rekursion - Heapsort

Der Konstruktor für die Klasse Heap allokiert das Feld **`a`**. Da **`a[0]`** nicht benutzt wird, benötigen wir ein Feld mit **`k+1`** Elementen, um **`k`** Zahlen zu speichern.

Die Initialisierung von **`a`** passiert über die Methode **`Setze`**.

Neben der bereits bekannten, privaten Methode **`Heapify`** werden die parameterlosen Methoden **`BaueHeap`** und **`Sortiere`** implementiert. Zusammen realisieren sie den Algorithmus *Heapsort*.

Rekursion - HeapSort

Der Aufbau eines ersten Heaps aus einem gegebenen Feld erfolgt durch den Aufruf der Methode **`BaueHeap`**.

Da jedes Blatt des Baumes für sich bereits einen Heap bildet, beginnen wir in der Mitte des Feldes mit dem Aufruf von **`Heapify`** und bauen so immer größere Heaps auf.

Anschließend führt die Methode **`Sortiere`** das bereits vorgestellte Vorgehen mit Vertauschen und Reorganisieren durch.

Das Auslesen eines sortierten Feldes kann durch die mehrfache Ausführung der Methode **`Gib`** erfolgen

Rekursion - Klasse Heap

```
class Heap {
   private int[] a;
   Heap(int k) {
         a = new int[k+1];
   }
   void Setze(int i, int x) { a[i] = x; }

   int Gib(int i) { return a[i]; }

   private void Tausche(int eins, int zwei) {
         int t = a[eins];
         a[eins] = a[zwei];
         a[zwei] = t;
   }
```

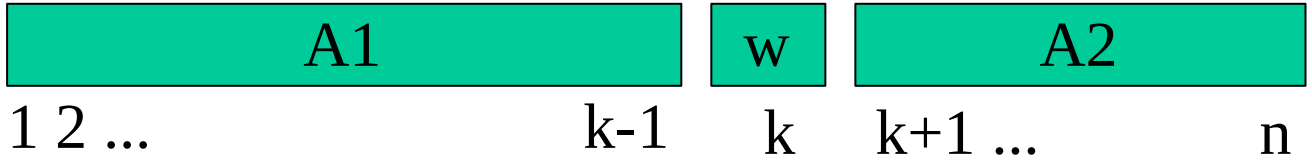
Rekursion - Auf dem Weg zu Heapsort

```
private void Heapify(int dieserKnoten, int heapGröße)
```
{wie vorne definiert }

```
void BaueHeap() {
        for (int i = (a.length-1)/2; i >= 1; i--)
              Heapify(i, a.length-1);
}

void Sortiere() {
        BaueHeap();
        for (int i = a.length-1; i > 1; i--) {
              Tausche(1, i);
              Heapify(1, i-1);
        }
}
```

Quicksort (nach C.A. Hoare)

- gutes Bsp, dass bessere Algorithmen mehr bringen können als ausgefeilte, systemnahe Programmierung
- Strategie: divide et impera, teile & herrsche, divide & conquer
- Idee:
  - wähle einen Wert w des Sortierfeldes aus
  - partitioniere Elemente in 2 Teilmengen, die selbst noch unsortiert sind
    - A1, deren Elemente <= w sind,
    - A2, deren Elemente >= w sind
  - sortiere die Teilmengen A1 und A2 (getrennt, rekursiv)
- Besonderheit: Partitionierung des Sortierfeldes nach w



Quicksort

Teilalgorithmus zur Partitionierung, (vereinfacht)

1. wähle w aus dem Array
2. suche Eintrag an einer Position i von links mit a[i] >= w
3. suche Eintrag an einer Position j von rechts mit a[i] <= w
4. vertausche a[i] mit a[j]
5. wiederhole ab 2. bis i >= j erreicht ist

Verfahren ist etwas einfacher, weil w untersortiert wird

Auswahl von w hat Einfluss auf die Güte des Verfahrens

warum ?

Auswahlverfahren:

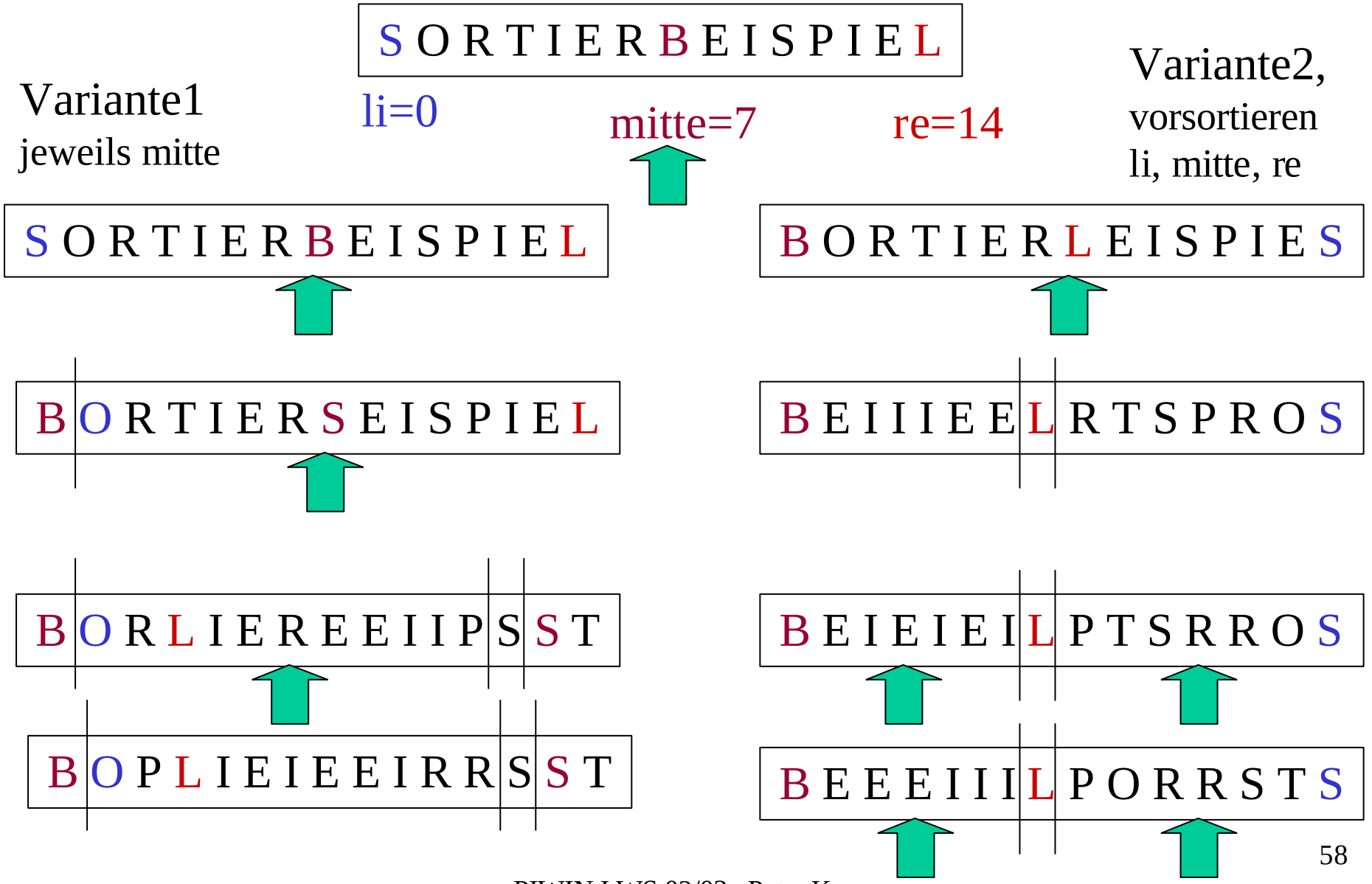
- betrachte a[links], a[(links+rechts)/2], a[rechts]
- Variante 1: wähle eines der drei Elemente
- Variante 2: wähle das wertmäßig mittlere Element

Erfahrungen:

Variante 1 ist bei zufällig verteilten Daten gut

Variante 2 ist bei ungünstig verteilten Daten gut

Quicksort


S O R T I E R B E I S P I E L
li=0
mitte=7
re=14
Variante1
jeweils mitte
Variante2,
vorsortieren
li, mitte, re
S O R T I E R B E I S P I E L
B O R T I E R L E I S P I E S
B O R T I E R S E I S P I E L
B E I I I E E L R T S P R O S
B O R L I E R E E I I P S S T
B E I E I E I L P T S R R O S
B O P L I E I E E I R R S S T
B E E E I I I L P O R R S T S

Quicksort

```
void quicksort(char [] A){
int high = A.length-1 ; rekquicksort(A,0,high) ;
}
```

```
void rekquicksort(char[] A, int low, int high){
int li = low, re = high, mid = (li+re)/2 ;
if (A[li] > A[mid]) tausche(A,li,mid) ;
if (A[mid] > A[re]) tausche(A,mid,re) ;
if (A[li] > A[mid]) tausche(A,li,mid) ;
if ((re-li) > 2) {
    char w = A[mid] ;
    do {
        while (A[li]<w) li++;
        while (w < A[re]) re-- ;
        if (li <= re) { tausche(A,li,re) ; li++ ; re-- ; }
    while (li <= re) ;
    if (lo < re) rekquicksort(A,lo,re) ;
    if (li < hi) rekquicksort(A,li,hi) ;
}}
```

Vorsortieren

mehr als 3Werte

w ist mittlerer Wert

suche von links und rechts Tauschkandidaten

Testfragen

- Quicksort ist ein rekursiver Algorithmus
  - Von welcher Art ist die Rekursion ?
  - Lässt sich die Anzahl rekursiver Aufrufe wie bei Fibonacci durch akkumulierende Parameter reduzieren ?
- Vergleich der Algorithmen
  - Was ist die Idee/Strategie beim naiven 1. Algorithmus, bei Heapsort, Quicksort?
  - Warum terminiert der 1. Algorithmus, Heapsort, Quicksort ?
  - Warum ist der 1. Algorithmus, Heapsort, Quicksort korrekt ?
  - Welcher Algorithmus ist besser ?
    - Was spricht für den 1. Algorithmus ?
    - Was spricht für Heapsort ?
    - Was spricht für Quicksort ?

Zusammenfassung

- Arrays
  - Datenstruktur zur Abbildung gleichartiger Daten
  - Deklaration
  - Dimensionierung und Zuordnung von Speicher zur Laufzeit
  - Zuweisung: ganzes Array, Werte einzelner Elemente
  - Speicherfreigabe durch Garbage Collection (Java) oder explizit (C/C++)
- Algorithmen auf Arrays: Beispiel Sortieren
  - naives Verfahren: Minimum bestimmen, entfernen, Restmenge sortieren
  - Heapsort: ähnlich, nur mit Binärbaum über Indexstruktur
  - Quicksort: divide&conquer, zerlegen in 2 Teilmengen anhand eines Pivotelementes
- Offen: Welches Verfahren ist besser ?