Kapitel 3: Sortieren

Einfache Sortierverfahren
Höhere Sortierverfahren
Problemspezifische Komplexitätsschranken

DBS
Prof. Dr. Thomas Seidl

Motivation zum Sortieren

- Anwendungsbeispiele:
  - Priorisierung von Aufgaben zum Zeitmanagement
  - Packaufgaben, schwere/große Gegenstände zuerst einpacken
  - Statistische Übersicht

- Ausgangpunkt: Folge von *Datensätzen*

$$D_1, D_2, D_3, \dots, D_n$$

- Jeder Datensatz besitzt eine Schlüsselkomponente $D_i.key$
- Jeder Datensatz kann außerdem weitere Informationseinheiten enthalten (z.B. Name, Adresse, PLZ, etc.)
- Ziel: Datensätze derart anordnen, dass die Schlüsselwerte sukzessive ansteigen (oder absteigen)
- Bedingung: Schlüssel müssen vergleichbar sein!

Totale Ordnung

Sei M eine nicht leere Menge und $\leq \subseteq M \times M$ eine binäre Relation auf M.

Das Paar $(M, \leq)$ heißt genau dann eine *totale (bzw. lineare) Ordnung* auf M, wenn folgende Eigenschaften erfüllt sind:

- Reflexivität: $\forall x \in M$: $x \leq x$
- Transitivität: $\forall x, y, z \in M$: $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Antisymmetrie: $\forall x, y \in M$: $x \leq y \wedge y \leq x \Rightarrow x = y$
- Totalität: $\forall x, y \in M$: $x \leq y \vee y \leq x$

Totale Ordnung: Beispiel

- $\leq$ auf den natürlichen Zahlen bildet eine totale Ordnung.

- Die lexikographische Ordnung $\leq_{lex}$ ist eine totale Ordnung. Sei $\leq$ die totale Ordnung auf den Buchstaben $A \leq B \leq \cdots \leq Z$.

  - Für zwei endliche Wörter $u = a_1 \dots a_k u_1 \dots u_m$ und $v = a_1 \dots a_k v_1 \dots v_n$ mit $u, v \in \{A, B, \dots, Z\}^*$ und $u_1 \neq v_1$ gilt:
  $$u \leq_{lex} v \Leftrightarrow u_1 \leq v_1 \vee u_1 \text{ "leer"}$$

  - Beispiel: FRANZ $\leq_{lex}$ FRANZISKA $\leq_{lex}$ PAUL $\leq_{lex}$ PETER

- Die Teilmengenrelation $\subseteq$ auf Potenzmenge von $\mathbb{N}$ ist keine totale Ordnung.
  - Beweis: $\{1\} \nsubseteq \{2\}$ und $\{2\} \nsubseteq \{1\}$ verletzen Totalität.

Das Sortierproblem

Gegeben sei eine Folge von *Schlüsselwerten*

$$a_1, a_2, a_3, \dots, a_n$$

und eine totale Ordnung $\leq$ auf der Schlüsselmenge $\{a_1, \dots, a_n\}$.

Die Ausgabe des Sortierproblems ist dann die Folge der Schlüssel

$$a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \dots, a_{\pi(n)}$$

mit:

- $a_{\pi(i)} \leq a_{\pi(i+1)}$ für alle $1 \leq i \leq n$.
- π ist eine Permutation der Indexmenge $\{1, \dots, n\}$.

Das Sortierproblem: Beispiele

Quelle	Schlüsselelement	Ordnung
Telefonbuch	Nachname	Lexikographische Ordnung
Klausurergebnisse	Punktezahl	$\leq$ auf rationalen Zahlen
Lexikon	Stichwort	Lexikographische Ordnung
Studentenverzeichnis	Matrikelnummer	$\leq$ auf $\mathbb{N}$
Entfernungstabelle	Distanz	$\leq$ auf $\mathbb{R}$
Fahrplan	Abfahrtszeit	„früher als“

Vergleichskriterien für Sortieralgorithmen

Sortieralgorithmen können nach verschiedenen Kriterien klassifiziert werden:

- Berechnungsaufwand
  - $O(n^2)$, $O(n \log n)$, $O(n)$
- Worst Case vs. Average Case
  - Ausnutzen von Vorsortierungen
- Speicherbedarf
  - In-place / in situ
  - Kopieren
- Stabilität
  - Erhalt der Reihenfolge von gleichwertigen Schlüsseln

Vertauschungen und Vergleiche

- Die Applikation ist wichtiger Auswahlfaktor:
  - Ein Verfahren mit vielen Vertauschungen und wenig Vergleichen ist gut geeignet für teure Vergleichsoperationen.
  - Falls die Umsortierung teuer ist, sollte man ein Verfahren mit wenig Vertauschungen wählen.

Beispiel 1: Sortiere Container nach Gewicht
- Teure Vertauschung
  (Kran, erst weitere Vertauschungen nötig)
- Günstige Vergleiche
  (Container-ID, Frachtpapiere)

Beispiel 2: Sortiere Beeren nach Vitamingehalt
- Günstige Vertauschung
  (Nehmen & Ablegen)
- Teure Vergleiche
  (Labortests, Zeit- und Kostenintensiv)

Weitere Aufgaben

Viele Aufgaben sind mit dem Sortieren verwandt und können auf das Sortierproblem zurückgeführt werden:

- Bestimmung des Median (der Median ist definiert als das Element an der mittleren Position der sortierten Folge)

- Bestimmung der k kleinsten bzw. größten Elemente

Typen von Sortieralgorithmen

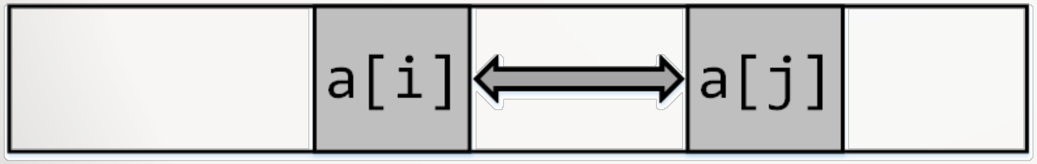
- Einfache
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - …
- Höhere
  - MergeSort
  - QuickSort
  - HeapSort
  - …
- Spezielle
  - BucketSort
  - …

Swap

Im Folgenden wird häufiger der Aufruf `swap(a, i, j)` verwendet, wobei `a` ein Array ist und `i`, `j` int-Werte.

Der Aufruf ersetzt folgende drei Zuweisungen:

```
Object temp = a[i];
a[i] = a[j];
a[j] = temp;
```



BubbleSort

- Idee:
  - Vergleiche Paare von benachbarten Schlüsseln
  - Tausche, falls linker Schlüssel größer ist als rechter
- Durch das repetitive Swapping verhalten sich Elemente wie aufsteigende Luftblasen im Wasser

```
public void bubblesort(int[] a){
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n-i-1; j++){
      if(a[j] > a[j+1])
        swap(a,j,j+1);
    }
  }
}
```

BubbleSort: Beispiel

3	7	8	6	4	2

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	0

BubbleSort: Beispiel

3	7	8	6	4	2

$3 < 7$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	0

BubbleSort: Beispiel

3	7	8	6	4	2

$7 < 8$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	1

BubbleSort: Beispiel

3	7	8	6	4	2

$$8 \nless 6$$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	2

BubbleSort: Beispiel

3	7	6	8	4	2

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	2

BubbleSort: Beispiel

3	7	6	8	4	2

$8 \nless 4$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	3

BubbleSort: Beispiel

3	7	6	4	8	2

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	3

BubbleSort: Beispiel

3	7	6	4	8	2

$8 \not< 2$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	4

BubbleSort: Beispiel

3	7	6	4	2	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	0
j	4

BubbleSort: Beispiel

3	7	6	4	2	8

$$3 < 7$$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	0

BubbleSort: Beispiel

3	7	6	4	2	8

$$7 \nless 6$$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	1

BubbleSort: Beispiel

3	6	7	4	2	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	1

BubbleSort: Beispiel

3	6	7	4	2	8

$$7 \not< 4$$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	2

BubbleSort: Beispiel

3	6	4	7	2	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	2

BubbleSort: Beispiel

3	6	4	7	2	8

$7 \nless 2$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	3

BubbleSort: Beispiel

3	6	4	2	7	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	1
j	3

BubbleSort: Beispiel

3	6	4	2	7	8

$3 < 6$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	0

BubbleSort: Beispiel

3	6	4	2	7	8

$$6 \nless 4$$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	1

BubbleSort: Beispiel

3	4	6	2	7	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	1

BubbleSort: Beispiel

3	4	6	2	7	8

$$6 \not< 2$$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	2

BubbleSort: Beispiel

3	4	2	6	7	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	2
j	2

BubbleSort: Beispiel

3	4	2	6	7	8

$$3 < 4$$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	3
j	0

BubbleSort: Beispiel

3	4	2	6	7	8

$4 \not< 2$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	3
j	1

BubbleSort: Beispiel

3	2	4	6	7	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	3
j	1

BubbleSort: Beispiel

3	2	4	6	7	8

$3 \not< 2$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	4
j	0

BubbleSort: Beispiel

2	3	4	6	7	8

Swap

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	4
j	0

BubbleSort: Beispiel

2	3	4	6	7	8

Sortiert!

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

i	4
j	0

BubbleSort: Komplexitätsanalyse Vergleiche

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

- Anzahl der Vergleiche:
  - Unabhängig von der Vorsortierung der Folge
    ⇒ worst case, average case und best case sind identisch
    (es werden stets alle Elemente der noch nicht sortierten Teilfolge miteinander verglichen)
  - Im i-ten Schleifendurchlauf werden $n - i - 1$ Elemente betrachtet und $n - i - 2$ Paare verglichen, insgesamt:

$$\sum_{i=1}^{n-1}(n - i - 2) = \sum_{i=1}^{n-1}(i - 2) = O(n^2)$$

BubbleSort: Komplexitätsanalyse Vertauschungen

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n-i-1; j++){
    if(a[j] > a[j+1])
      swap(a,j,j+1);
  }
}
```

- Anzahl der Vertauschungen:
  - Best Case: 0 Swaps (Vorsortierung)
  - Worst Case: $\frac{1}{2}(n^2 - n)$ Swaps (Invertierte Reihenfolge)
  - Average Case: $\frac{1}{2}(n^2 - n)$ Swaps
    - » Donald E. Knuth: *The Art of Computer Programming (1997)*

SelectionSort

- Idee:
  - Durchlaufe die Folge von links nach rechts mit einem Zeiger i.
  - Links von i sind die i-kleinsten Elemente sortiert.
  - Finde in der verbleibenden Menge das kleinste Element und tausche es mit i. Dann erhöhe i um eins.
  - Tausche, falls linker Schlüssel größer ist als rechter

```
public void selectionsort(int[] a){
  for(int i = 0; i < n-1; i++) {
    int min = i;
    for(int j = i+1; j < n; j++){
      if(a[j] < a[min])
        min = j;
    }
    swap(a,i,min);
  }
}
```

SelectionSort: Beispiel

3	7	8	6	4	2

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	1
min	0

SelectionSort: Beispiel

3 min	7	8	6	4	2

$3 < 7$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	1
min	0

SelectionSort: Beispiel

3 min	7	8	6	4	2

$3 < 8$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	2
min	0

SelectionSort: Beispiel

3	7	8	6	4	2

min

3 < 6

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	3
min	0

SelectionSort: Beispiel

3	7	8	6	4	2

min

$3 < 4$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	4
min	0

SelectionSort: Beispiel

3	7	8	6	4	2
					min

$3 \nless 2$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	5
min	5

SelectionSort: Beispiel

2	7	8	6	4	3

min

Swap

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	0
j	5
min	5

SelectionSort: Beispiel

2	7	8	6	4	3

min

$7 < 8$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	2
min	1

SelectionSort: Beispiel

2	7	8	6 min	4	3

$7 \nless 6$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	3
min	3

SelectionSort: Beispiel

2	7	8	6	4 min	3

$6 \nless 4$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	4
min	4

SelectionSort: Beispiel

2	7	8	6	4	3 min

$$4 \nless 3$$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	5
min	5

SelectionSort: Beispiel

2	3	8	6	4	7
	min				

Swap

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	1
j	5
min	5

SelectionSort: Beispiel

2	3	8	6 min	4	7

$8 \nless 6$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	3
min	3

SelectionSort: Beispiel

2	3	8	6	4 min	7

$$6 \nless 4$$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	4
min	4

SelectionSort: Beispiel

2	3	8	6	4	7

min

$$4 < 7$$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	5
min	4

SelectionSort: Beispiel

2	3	4	6	8	7

min

Swap

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	2
j	5
min	4

SelectionSort: Beispiel

2	3	4	6	8	7
			min		

6 < 8

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	3
j	4
min	3

SelectionSort: Beispiel

2	3	4	6	8	7

min

6 < 7

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	3
j	5
min	3

SelectionSort: Beispiel

2	3	4	6	8	7

min

$8 \nless 7$

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	4
j	5
min	5

SelectionSort: Beispiel

2	3	4	6	7	8

min

Swap

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	4
j	5
min	5

SelectionSort: Beispiel

2	3	4	6	7	8

Sortiert!

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

i	4
j	5
min	5

SelectionSort: Komplexitätsanalyse Vergleiche

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

- Anzahl der Vergleiche:
  - In jedem der n Durchläufe $n - i$ Vergleiche
$$\Rightarrow \frac{n(n-1)}{2} \in O(n^2)$$
  - Durch weitere if-Abfrage (min == i) lässt sich Anzahl der swaps im Austausch gegen n zusätzliche Vergleiche verringern

SelectionSort: Komplexitätsanalyse Vertauschungen

```
for(int i = 0; i < n-1; i++) {
  int min = i;
  for(int j = i+1; j < n; j++){
    if(a[j] < a[min])
      min = j;
  }
  swap(a,i,min);
}
```

- Anzahl der Vertauschungen:
  - Best Case: $n-1$ Swaps (0 mit zusätzlicher Abfrage)
  - Worst Case: $n-1$ Swaps
  - Average Case: $n-1$ Swaps
- Anzahl Swaps wachsen linear.
  Daher ist SelectionSort besonders für Folgen großer Objekte geeignet, deren Vertauschungen teurer sind.

InsertionSort

- Idee:
  - Halte die linke Teilfolge sortiert.
  - Füge nächstes Objekt hinzu, indem es an die korrekte Position eingefügt wird.
  - Wiederhole dies, bis Teilfolge aus der gesamten Liste besteht.

```
public void insertionsort(int[] a){
  for(int i = 1; i < n; i++) {
    int key = a[i];
    int j = i;
    while(j > 0 && a[j-1] > key){
      a[j] = a[j-1];
      j--;
    }
    a[j] = key;
  }
}
```

InsertionSort: Beispiel

3	7	8	6	4	2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	1
j	1

InsertionSort: Beispiel

3	7	8	6	4	2

$3 < 7$

7

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	1
j	1

InsertionSort: Beispiel

3	7	8	6	4	2

$7 \nless 8$ 8

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	2
j	2

InsertionSort: Beispiel

3	7	8	6	4	2

$6 \nless 8$ 6

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	3

InsertionSort: Beispiel

3	7	8	8	4	2

$7 \nless 6$

6

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	2

InsertionSort: Beispiel

3	7	7	8	4	2

$3 < 6$

6

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	1

InsertionSort: Beispiel

3	6	7	8	4	2

insert

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	3
j	1

InsertionSort: Beispiel

3	6	7	8	4	2

$4 < 8$ 4

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	4

InsertionSort: Beispiel

3	6	7	8	8	2

$4 < 7$ 4

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	3

InsertionSort: Beispiel

3	6	7	7	8	2

$4 < 6$ | 4 |

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	2

InsertionSort: Beispiel

3	6	6	7	8	2

$4 \nless 3$

4

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	1

InsertionSort: Beispiel

3	4	6	7	8	2

insert

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	4
j	1

InsertionSort: Beispiel

3	4	6	7	8	2

$2 < 8$ | 2 |

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	5

InsertionSort: Beispiel

3	4	6	7	8	8

$2 < 7$ 2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	4

InsertionSort: Beispiel

3	4	6	7	7	8

$2 < 6$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	3

InsertionSort: Beispiel

3	4	6	6	7	8

$2 < 4$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	2

InsertionSort: Beispiel

3	4	4	6	7	8

$2 < 3$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	1

InsertionSort: Beispiel

3	3	4	6	7	8

$j = 0!$

2

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	0

InsertionSort: Beispiel

2	3	4	6	7	8

insert

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	0

InsertionSort: Beispiel

2	3	4	6	7	8

Sortiert!

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

i	5
j	0

InsertionSort: Sentinel-Variante

- Erweitere das Eingabearray vorne um eine Position

$-\infty$	2	3	4	6	7	8

Wächter

- Der Code kann nun auf eine Abfrage verzichten:

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

InsertionSort: Komplexitätsanalyse Vergleiche

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

- Anzahl der Vergleiche:
  - Abhängig von Vorsortierung
  - Im Best Case $O(n)$ Vergleiche bei sortierter Folge
  - Average Case und Worst Case: $O(n^2)$

InsertionSort: Komplexitätsanalyse Verschiebeoperationen

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i;
  while(j > 0 && a[j-1] > key){
    a[j] = a[j-1];
    j--;
  }
  a[j] = key;
}
```

- Im besten Fall ist die Folge sortiert und man benötigt keine Verschiebung
- Im schlimmsten Fall ist die Folge absteigend sortiert und das j-te Element wird mit $j-1$ Operationen an den jeweiligen Anfang geschoben: $\frac{n(n-1)}{2} \in O(n^2)$
- Man kann zeigen, dass im Average Case $\frac{n(n-1)}{4} \in O(n^2)$ Operationen nötig sind.

Fazit: Einfache Sortieralgorithmen

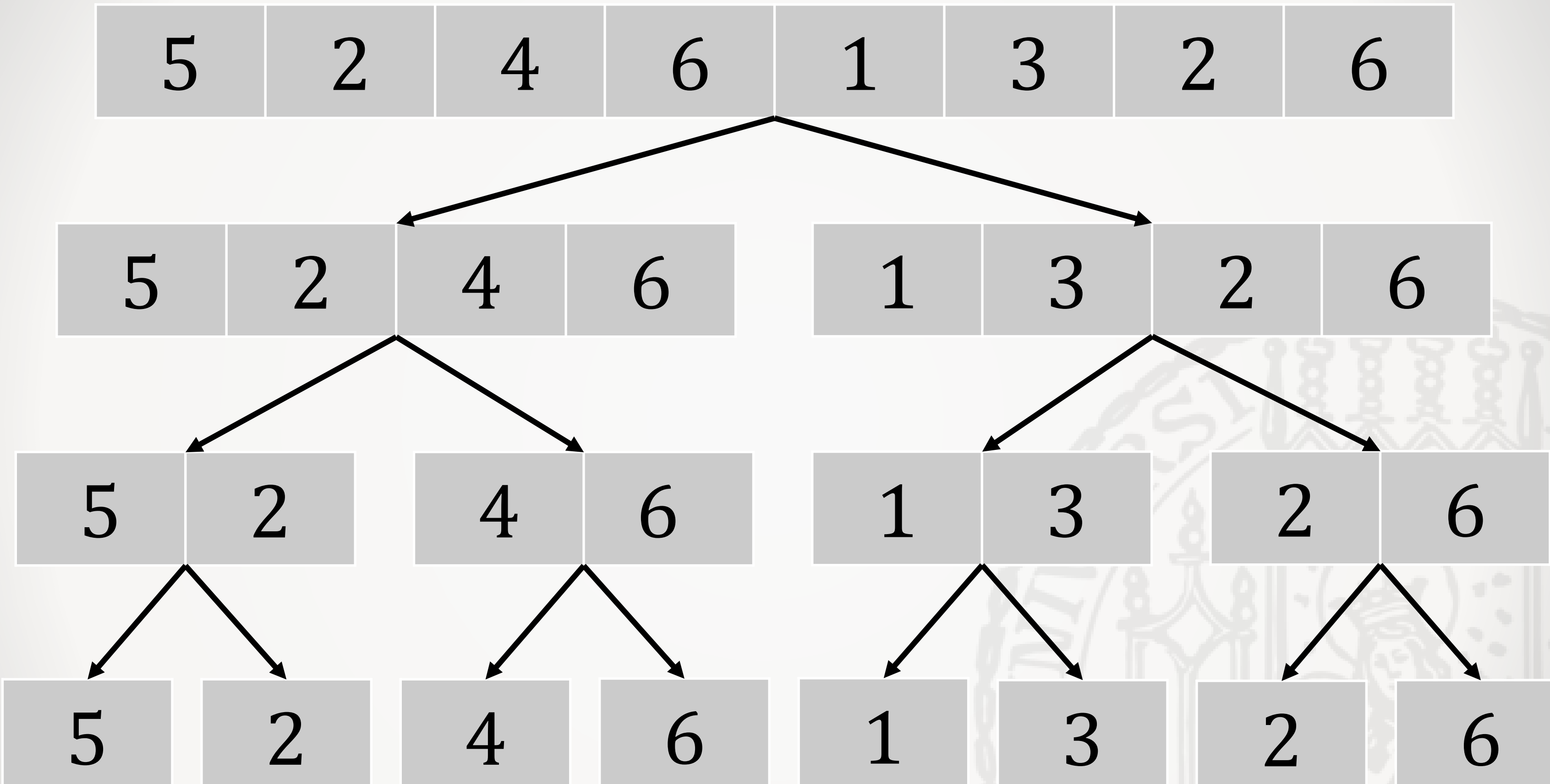
- BubbleSort:
  - Einfache Implementierung.
  - Führt auch im Best Case / Average Case viele Vergleiche aus.
- SelectionSort:
  - Benötigt durchschnittlich weniger Vertauschungen
  - Besser geeignet für teure Vertauschungen und günstige Vergleiche
- InsertionSort:
  - Mehr Vertauschungen, dafür weniger Vergleiche
  - Eignet sich gut, falls die Folge während der Ausführung erweitert wird (Datenstrom)
- Bislang benötigen alle Algorithmen $O(n^2)$ Vergleiche oder Bewegungen.

Geht es auch besser?

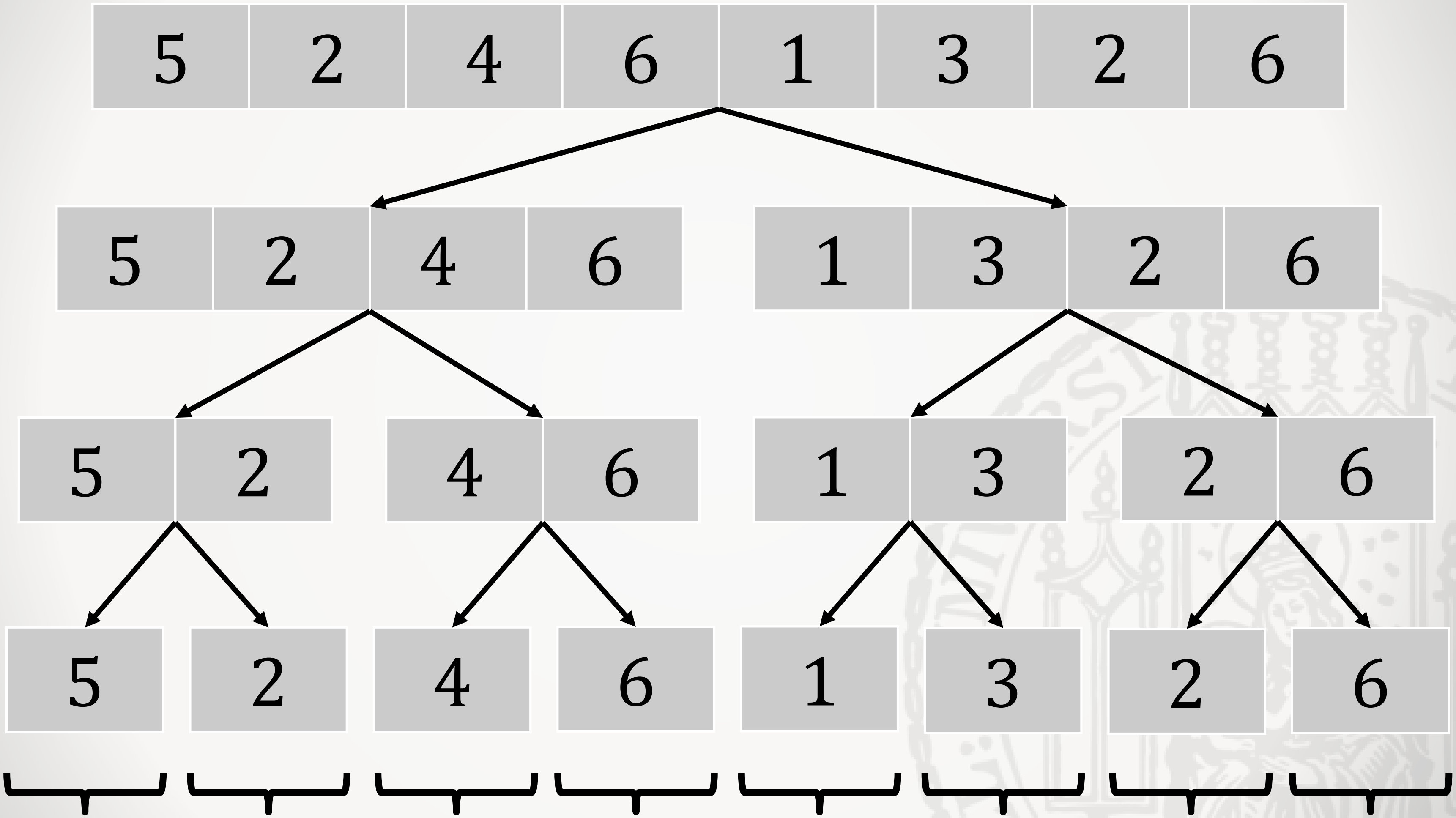
MergeSort

- Idee:
  - Zerlege die Eingabefolge in zwei gleichlange Folgen. Diese lassen sich dann unabhängig vorsortieren.
  - Anschließend werden beide Teilfolgen zusammengefügt.
  - Rekursiv können diese Schritte auch auf die Teillisten angewendet werden.
- Diese Strategie ist bekannt als Divide-and-Conquer
  - **Divide**: Zerlegen der Folge
    $$F = \left(a[1], \dots, a\left[\left\lfloor \frac{n}{2} \right\rfloor\right]\right) \circ \left(a\left[\left\lfloor \frac{n}{2} \right\rfloor + 1\right], \dots, a[n]\right) = F_L \circ F_R$$
  - **Conquer**: Sortiere F_L und F_R mittels MergeSort, falls $|F_L| > 1$ bzw. falls $|F_R| > 1$.
  - **Merge**: Verschmelze sortierte Teilfolgen F_L und F_R zu sortierter Gesamtfolge

MergeSort: Divide-Schritt

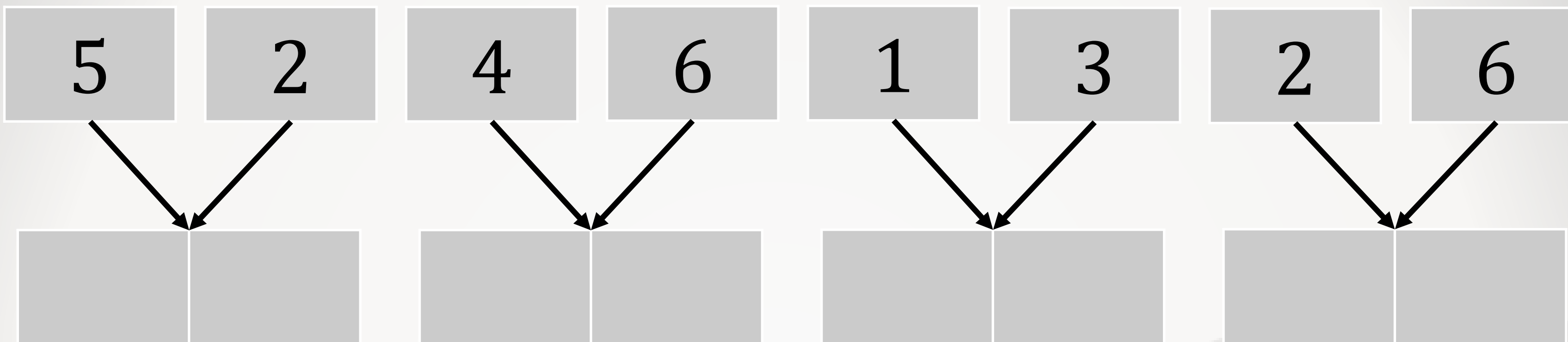


MergeSort: Conquer-Schritt

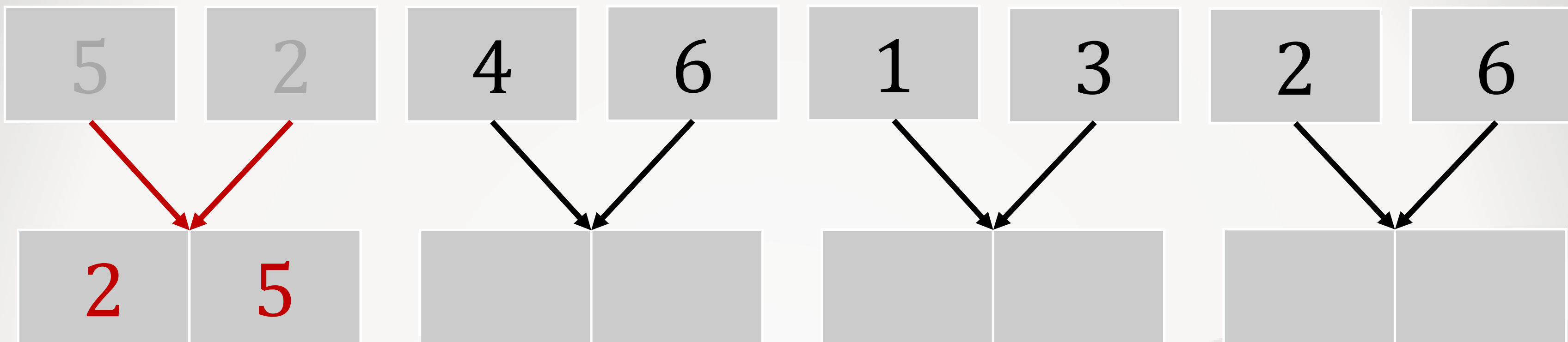


Atomare (einelementige) Folgen sind immer sortiert

MergeSort: Merge

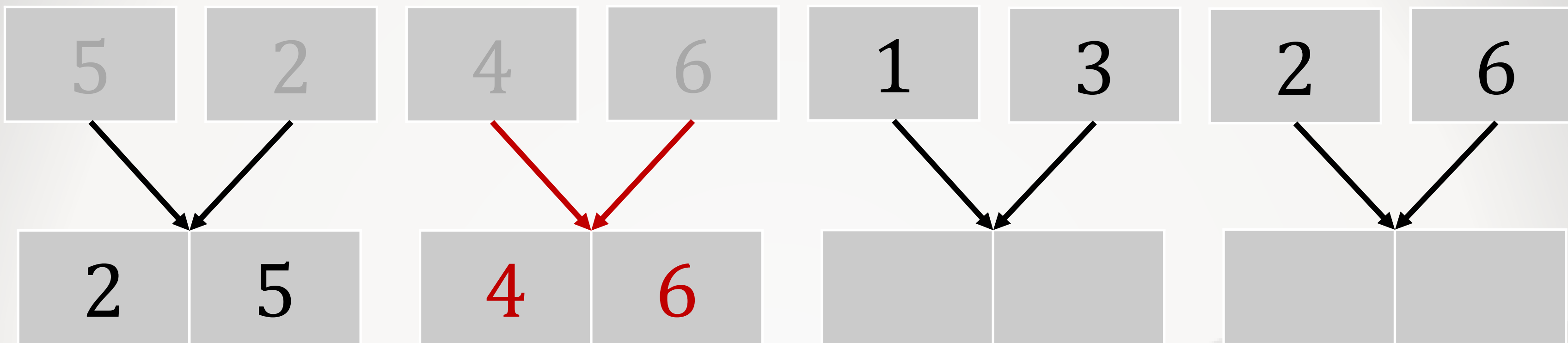


MergeSort: Merge



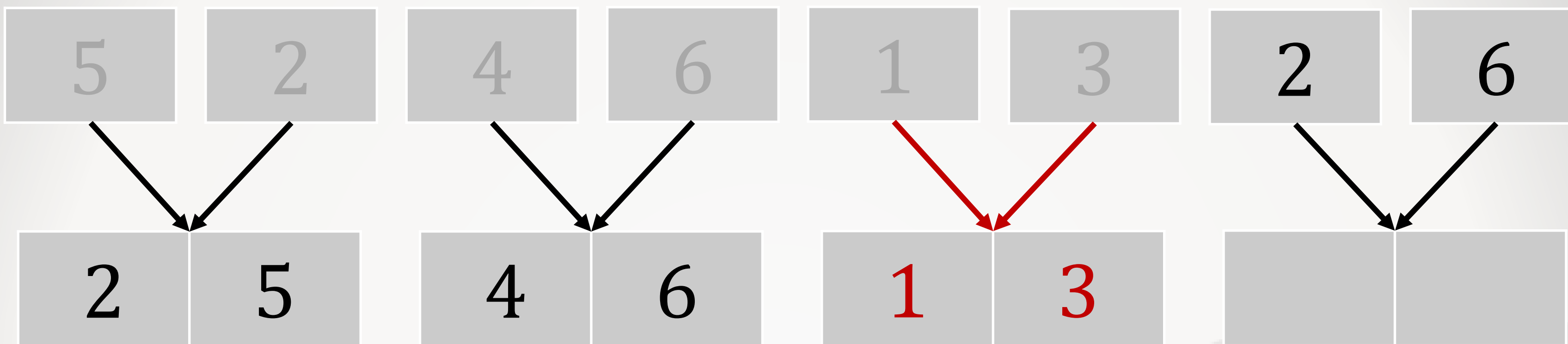
Vergleiche:
$5 < 2$?

MergeSort: Merge



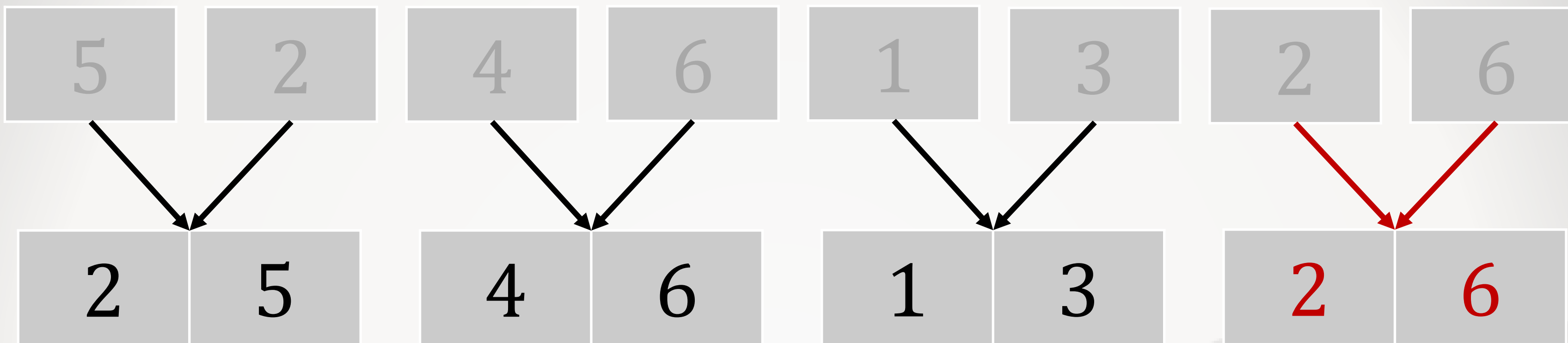
Vergleiche:
$5 < 2?$ $4 < 6?$

MergeSort: Merge



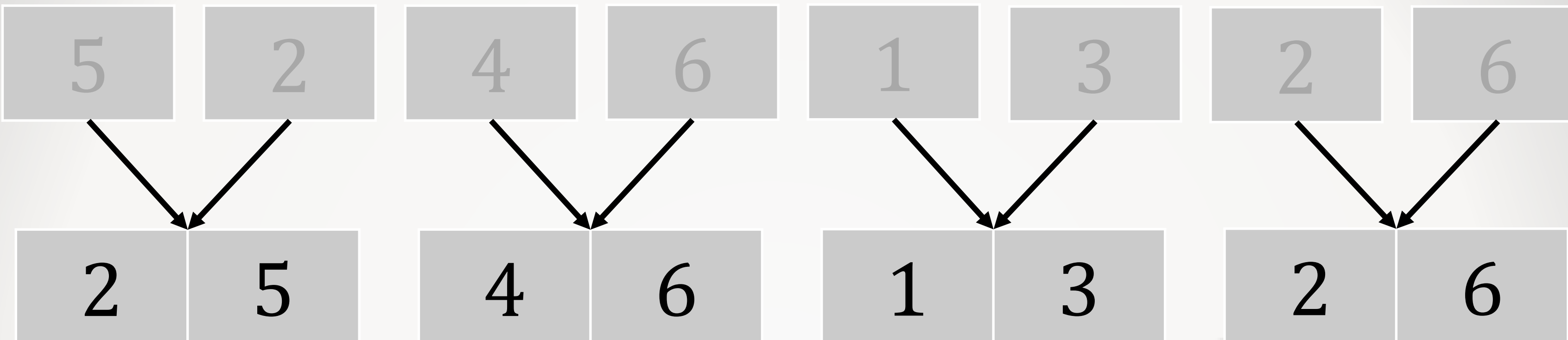
Vergleiche:
$5 < 2$? $4 < 6$? $1 < 3$?

MergeSort: Merge



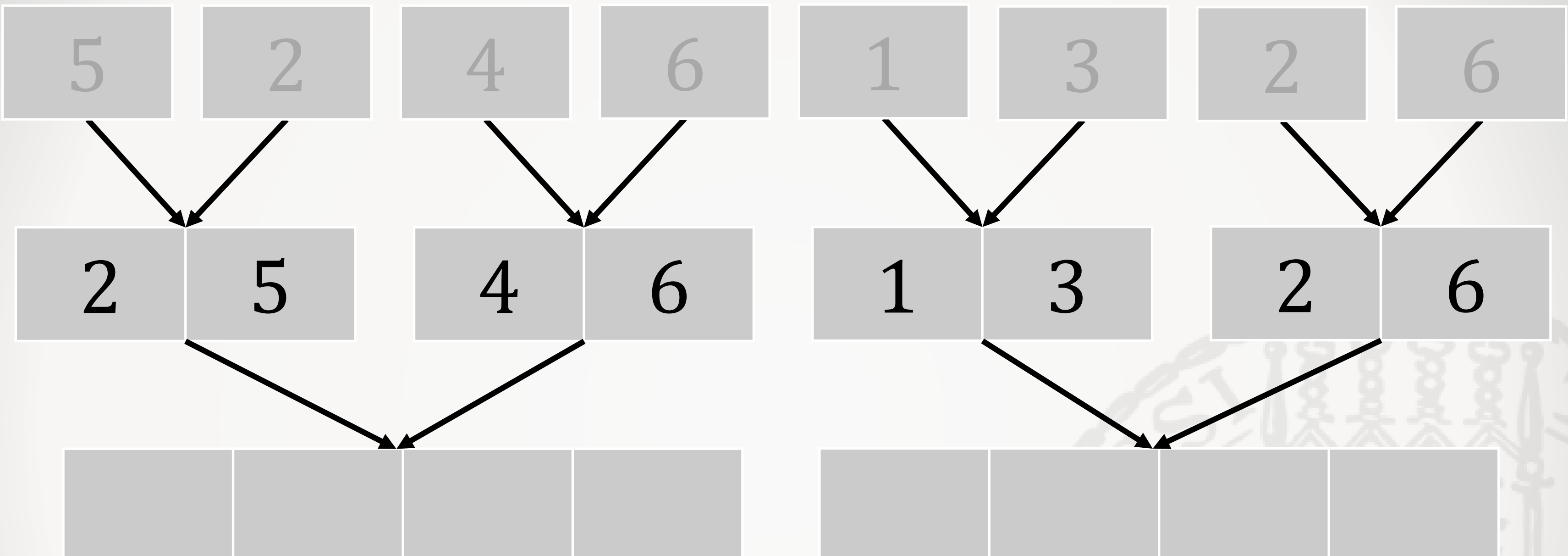
Vergleiche:
$5 < 2$? $4 < 6$? $1 < 3$? $2 < 6$?

MergeSort: Merge



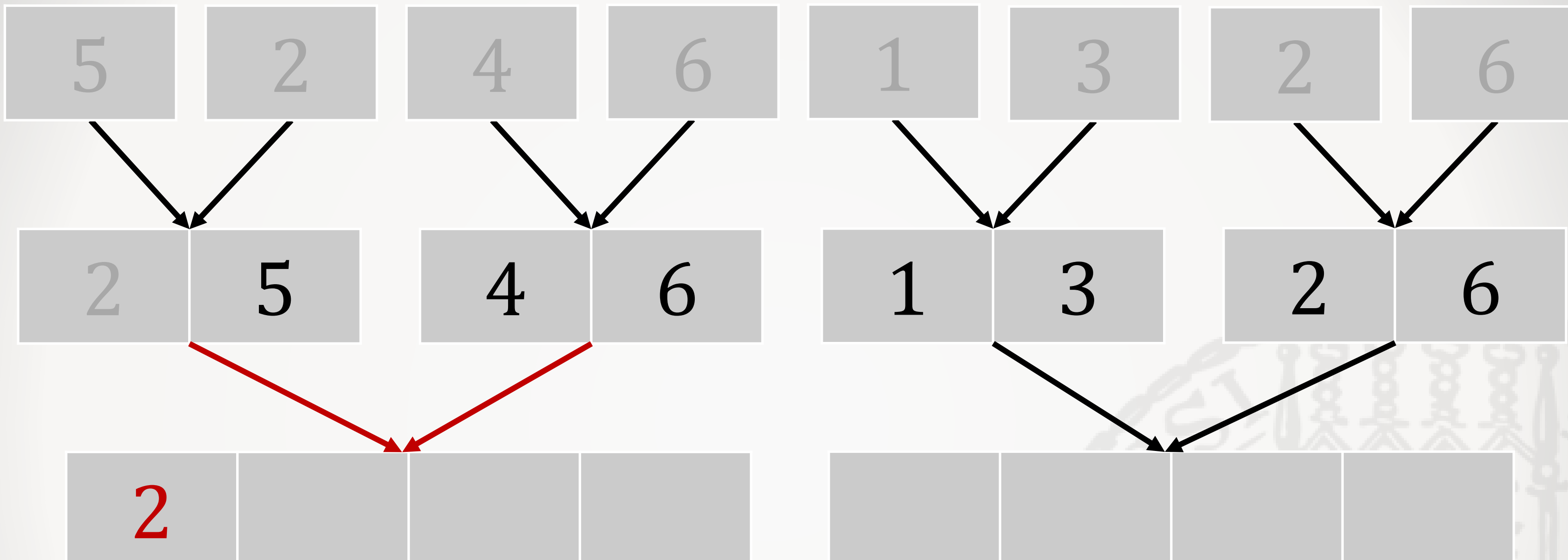
Vergleiche:
$5 < 2?\ 4 < 6?\ 1 < 3?\ 2 < 6? \Rightarrow 4$ Vergleiche

MergeSort: Merge



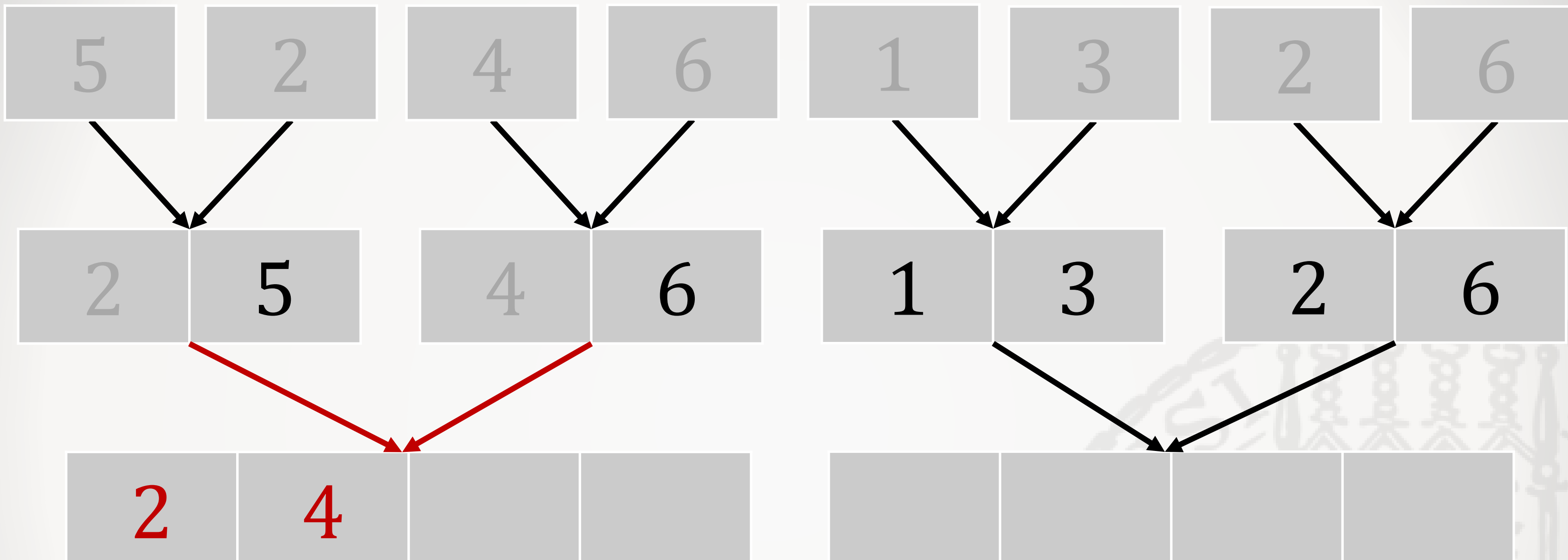
Vergleiche: 4

MergeSort: Merge



Vergleiche: 4
$2 < 4$?

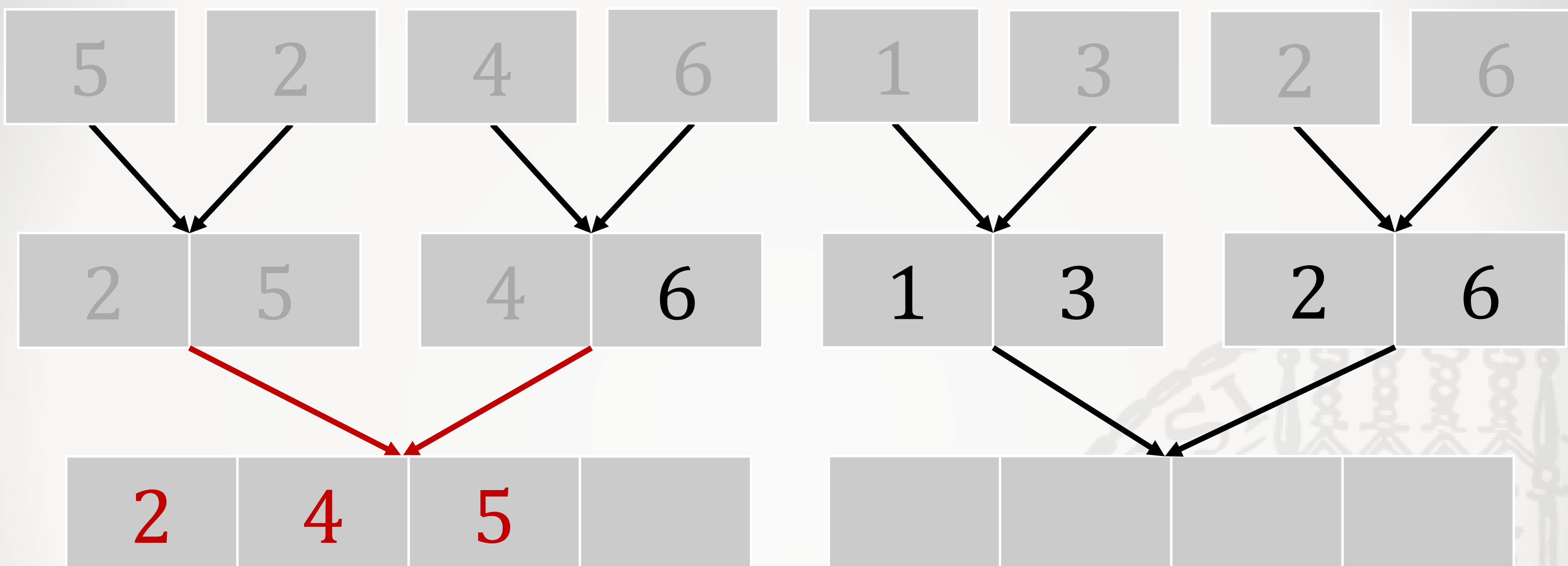
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$

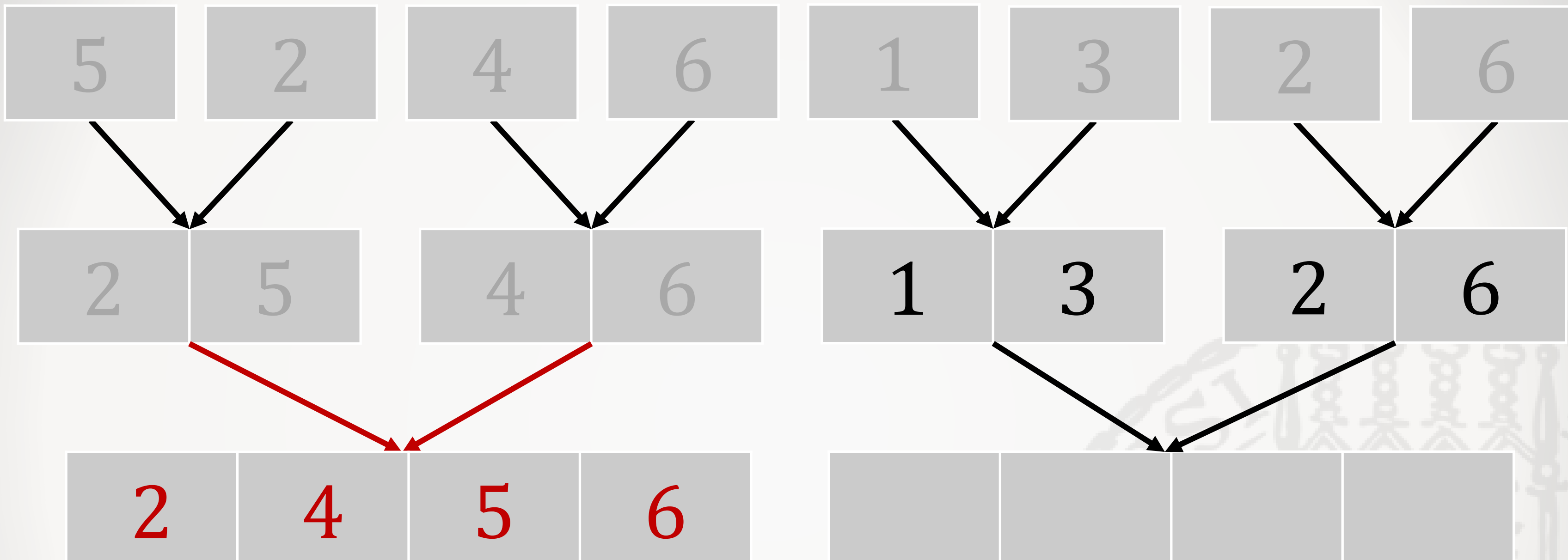
MergeSort: Merge



Vergleiche: 4

$2 < 4?\ \ 5 < 4?\ \ 5 < 6?$

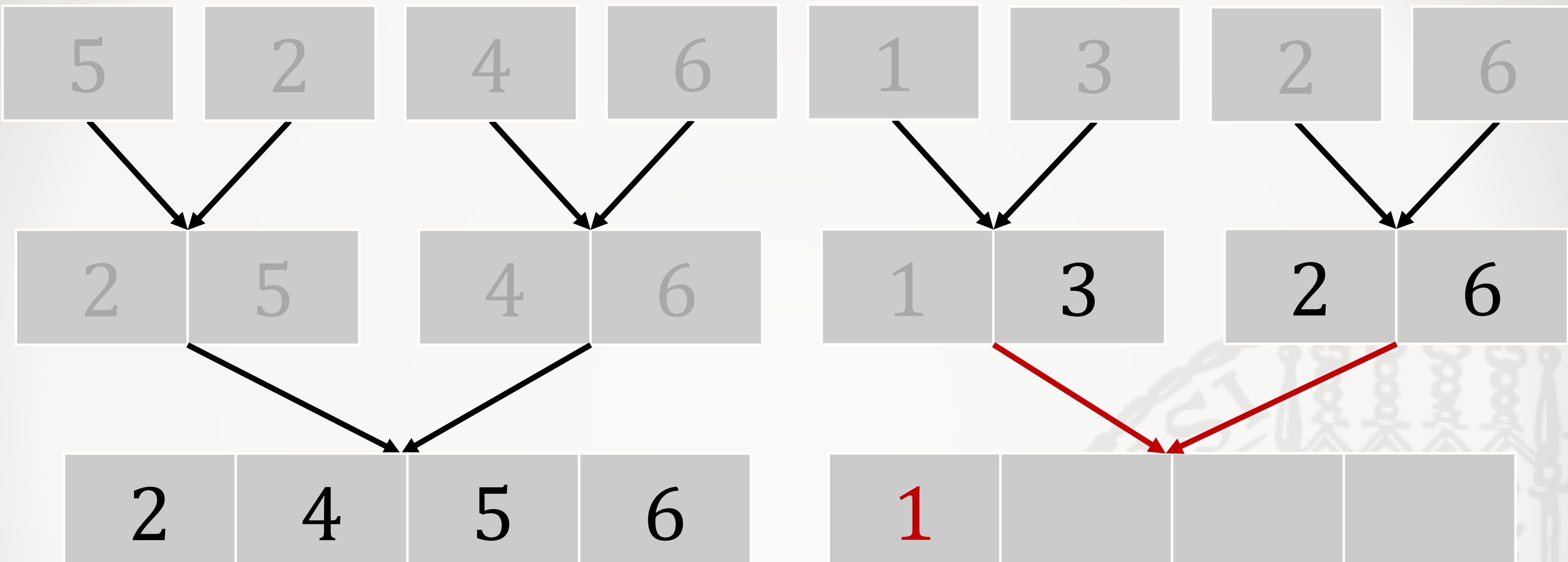
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$ $5 < 6?$

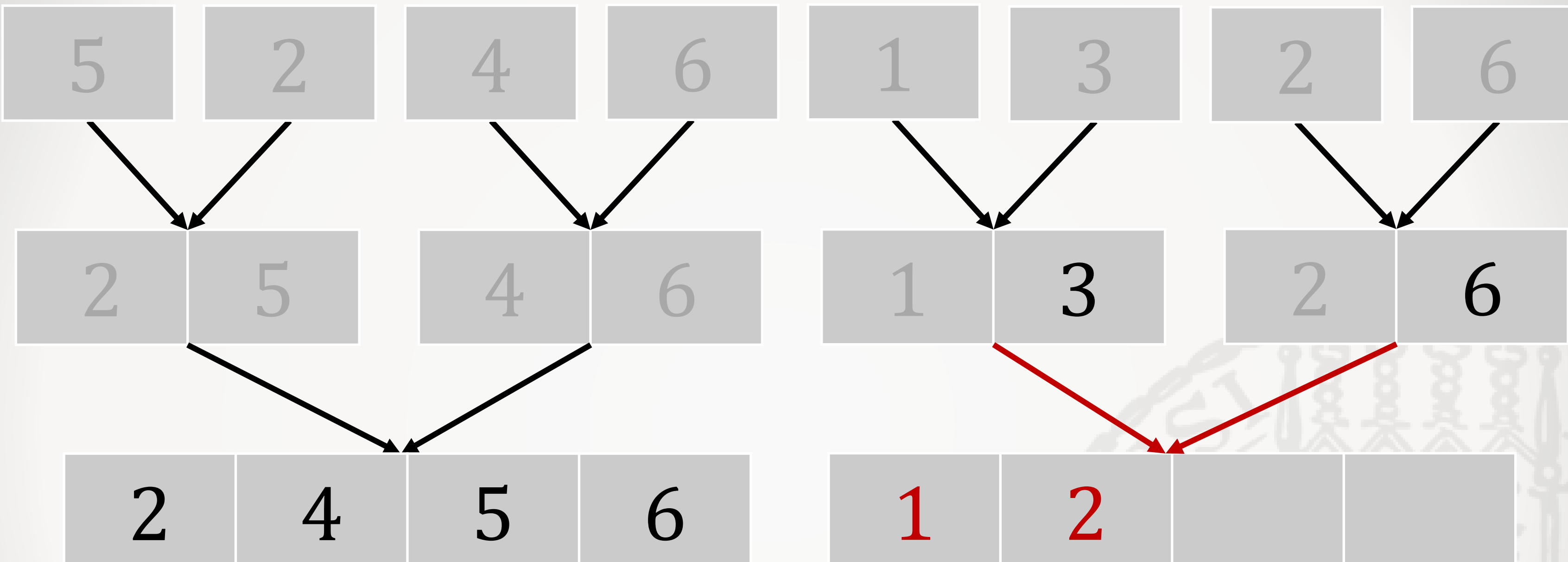
MergeSort: Merge



Vergleiche: 4

$2 < 4$? $5 < 4$? $5 < 6$? $1 < 2$?

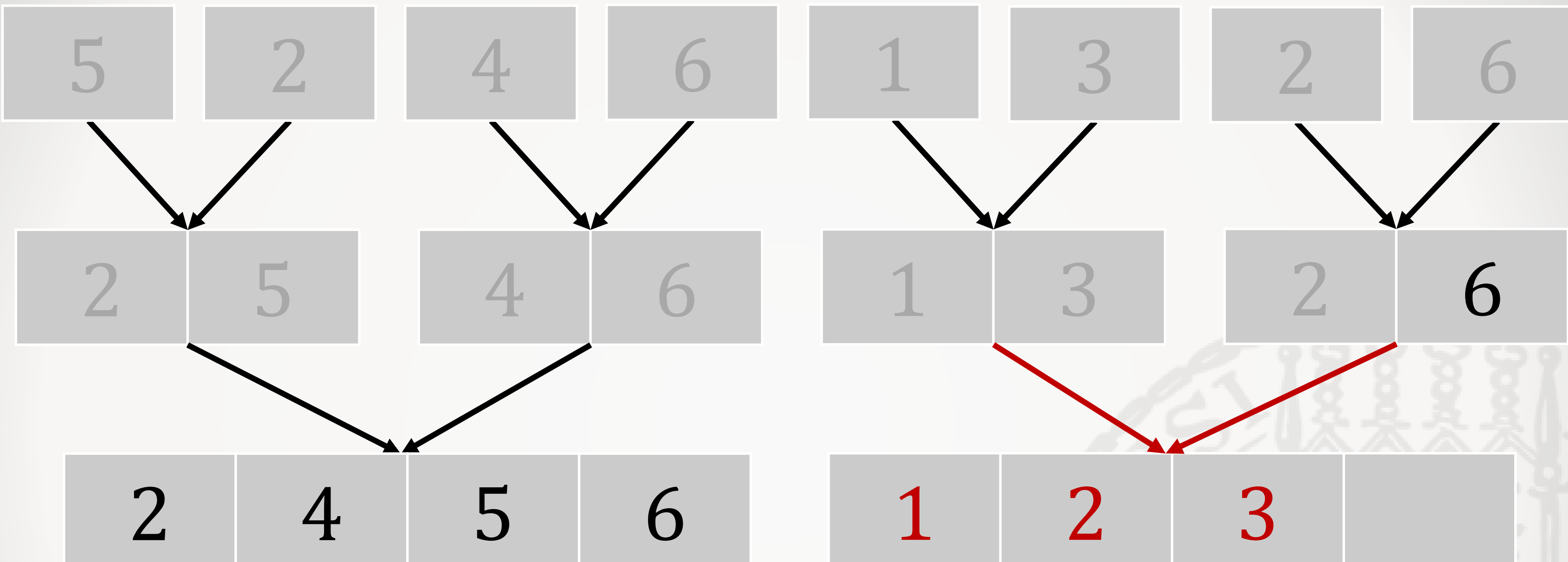
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$ $5 < 6?$ $1 < 2?$ $3 < 2?$

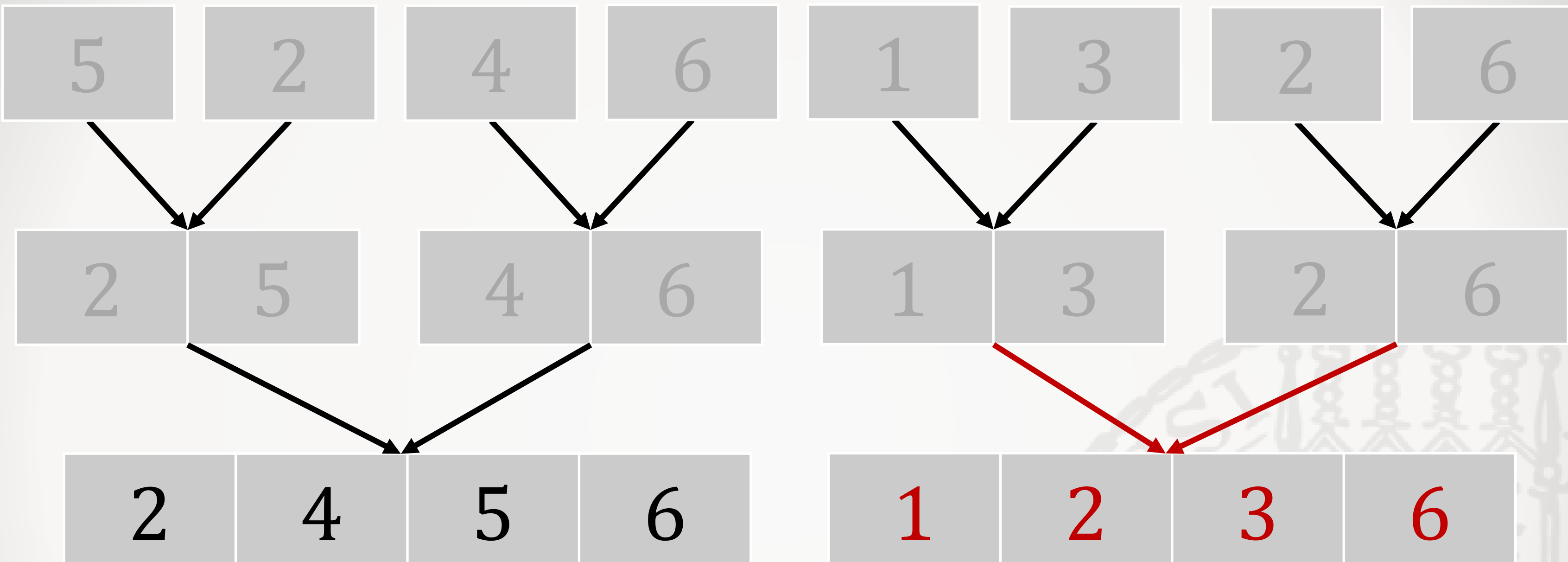
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$ $5 < 6?$ $1 < 2?$ $3 < 2?$ $3 < 6?$

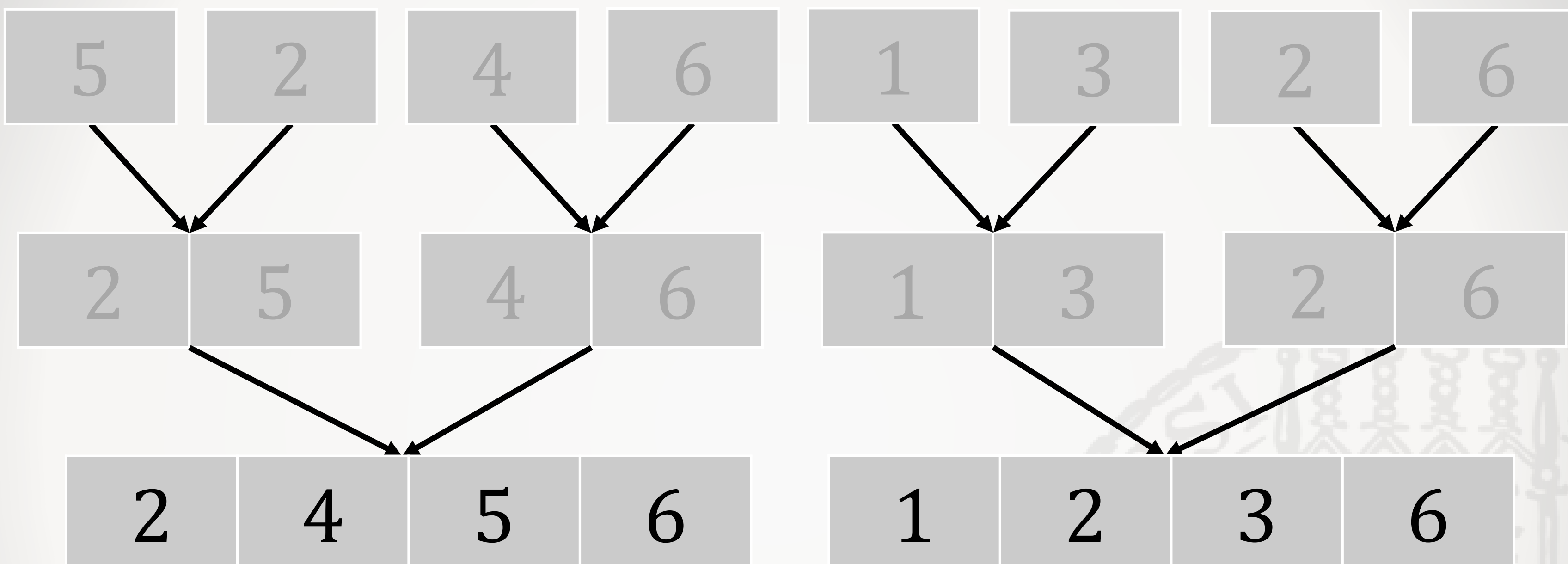
MergeSort: Merge



Vergleiche: 4

$2 < 4?$ $5 < 4?$ $5 < 6?$ $1 < 2?$ $3 < 2?$ $3 < 6?$

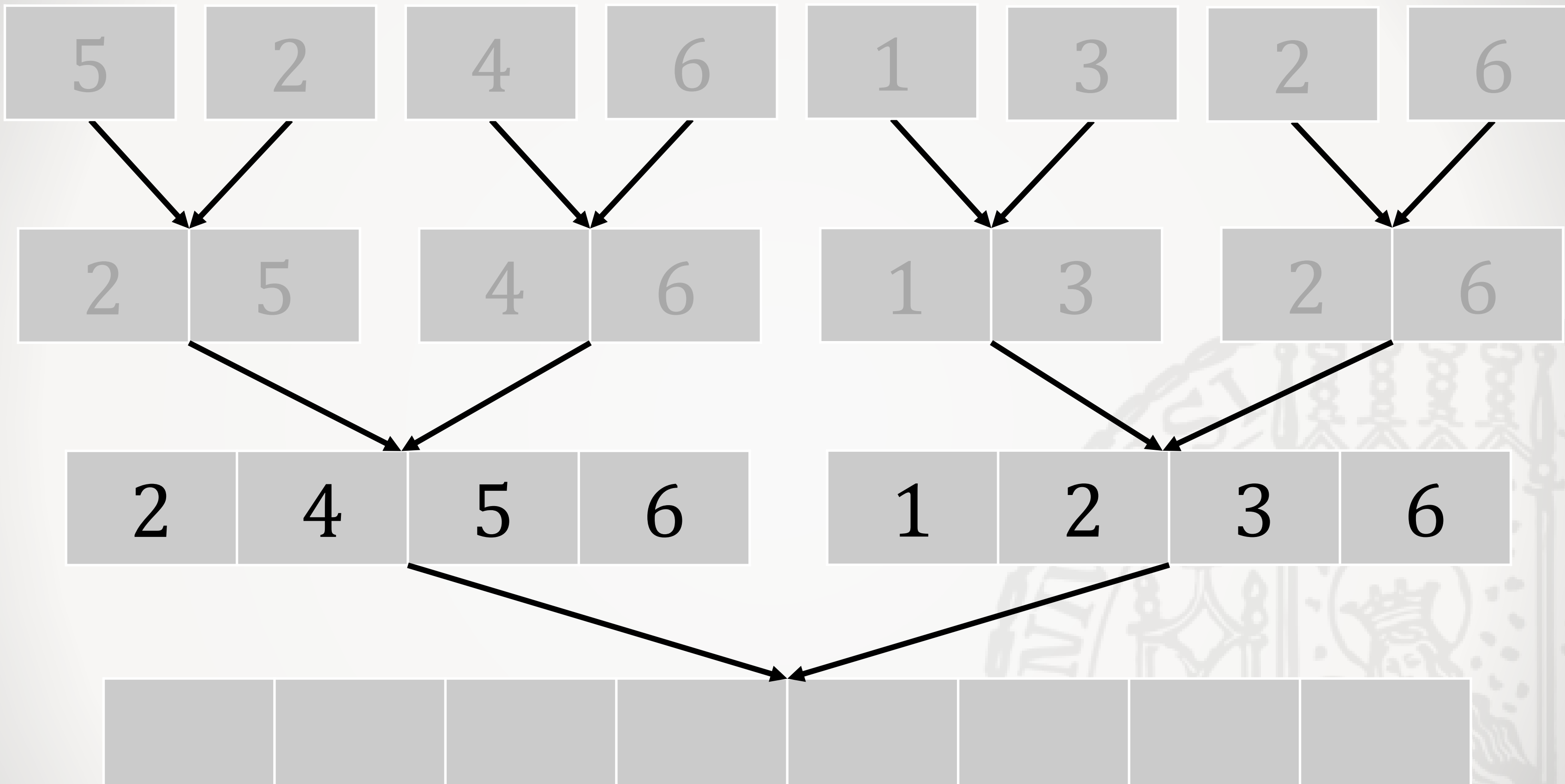
MergeSort: Merge



Vergleiche: $4 + 6$

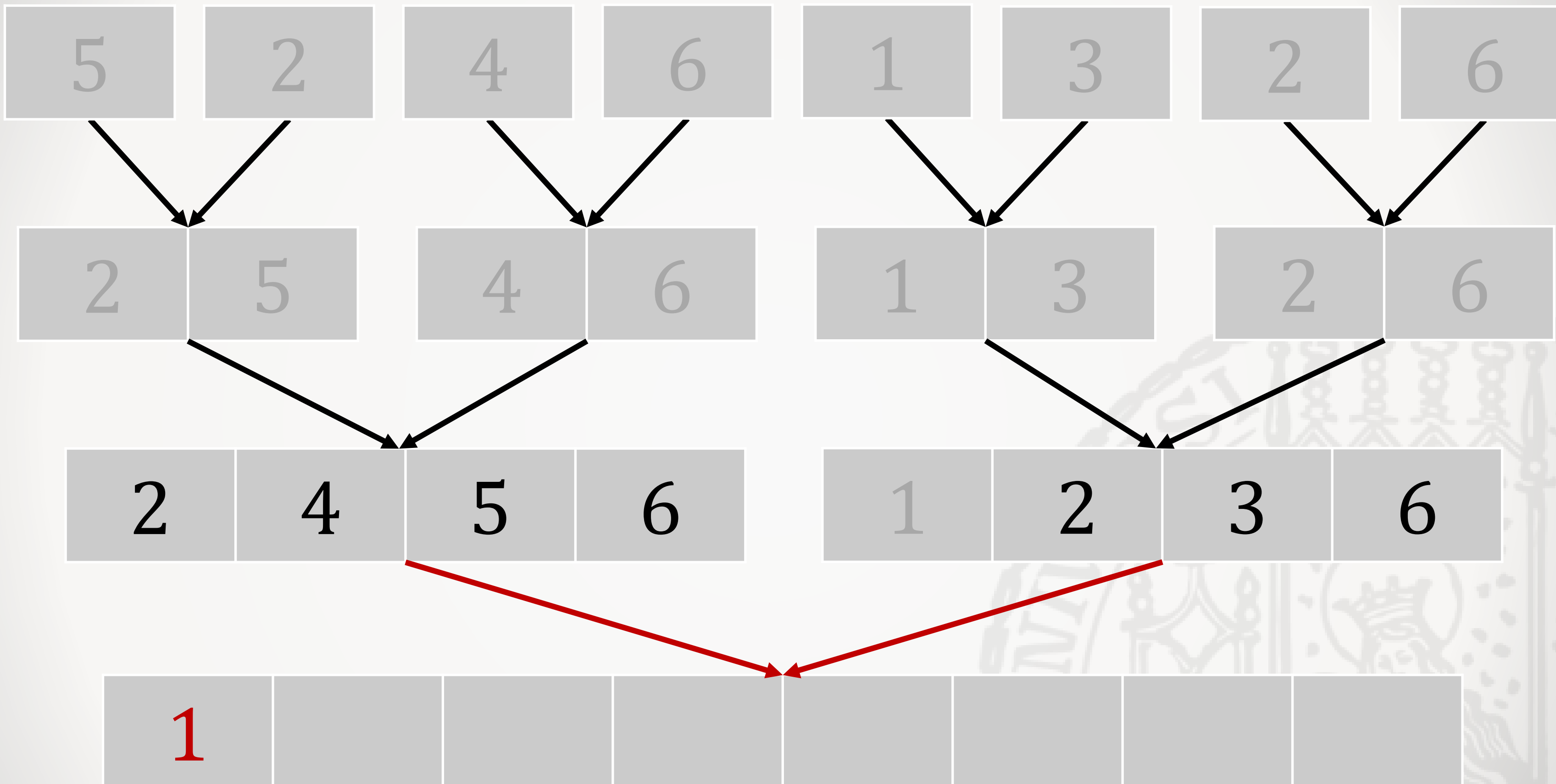
$2 < 4?$ $5 < 4?$ $5 < 6?$ $1 < 2?$ $3 < 2?$ $3 < 6? \Rightarrow 6$ Vergleiche

MergeSort: Merge



Vergleiche: 4 + 6

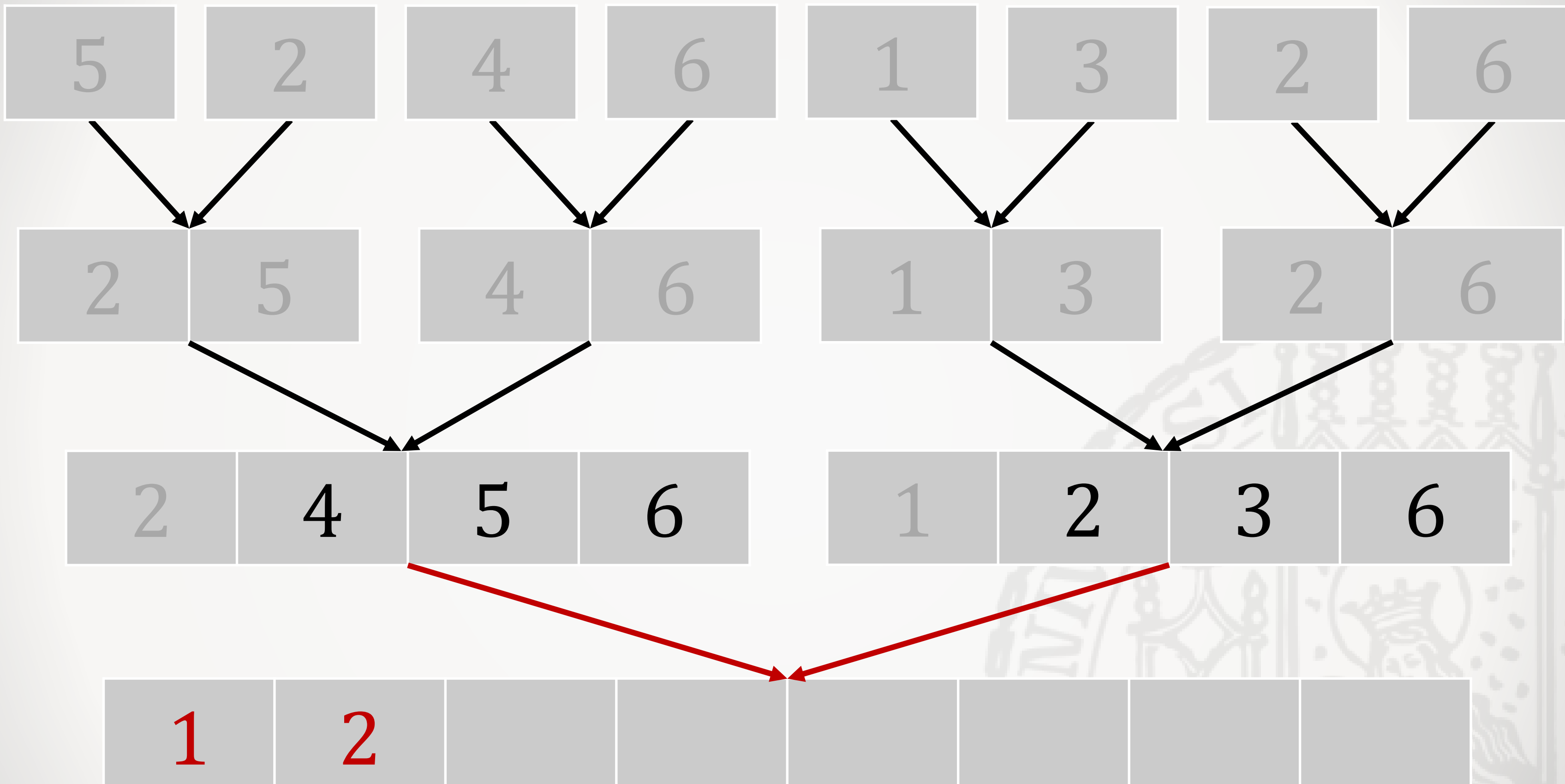
MergeSort: Merge



Vergleiche: 4 + 6

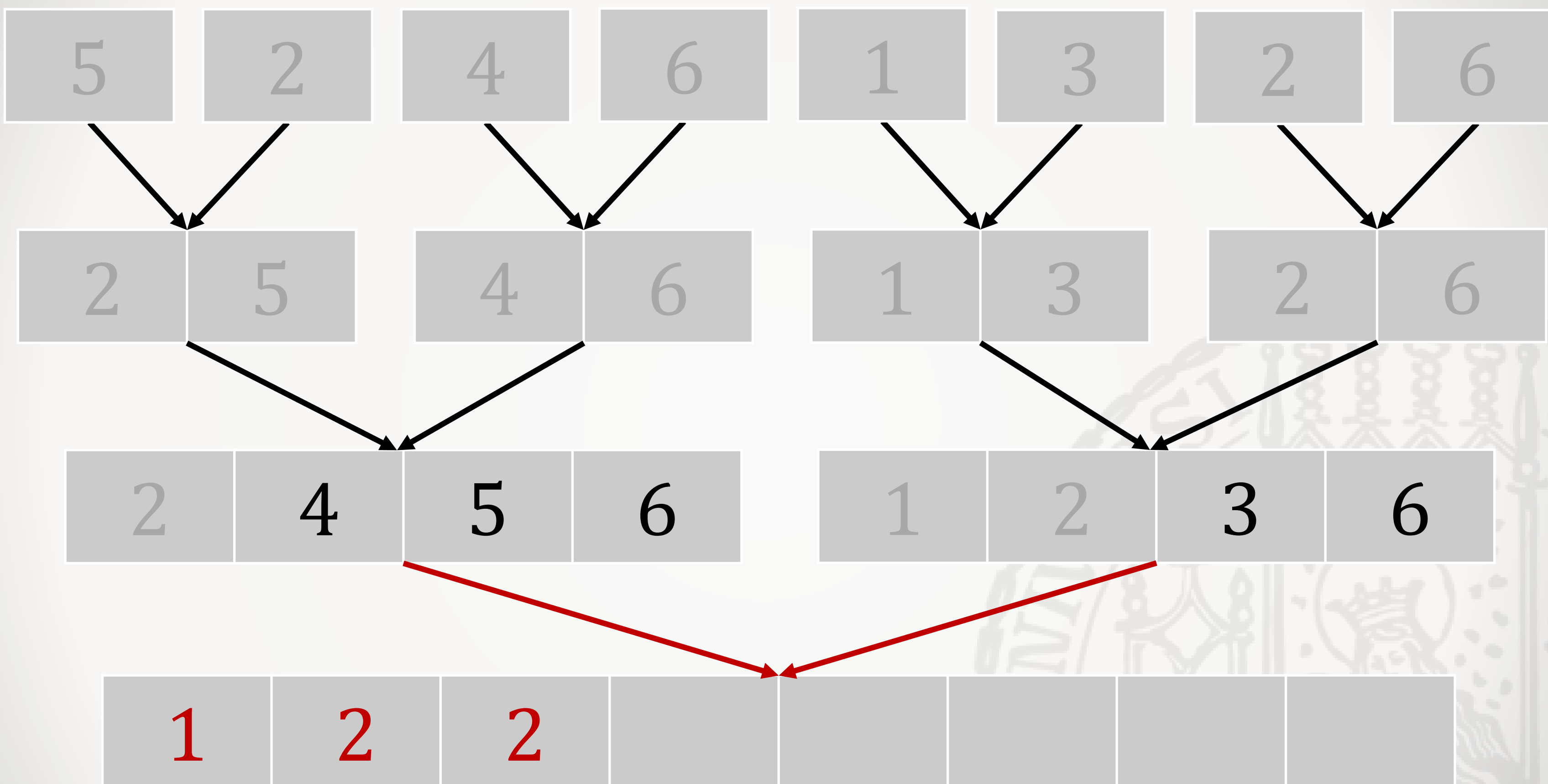
2 < 1?

MergeSort: Merge



Vergleiche: 4 + 6
$2 < 1?$ $2 < 2?$

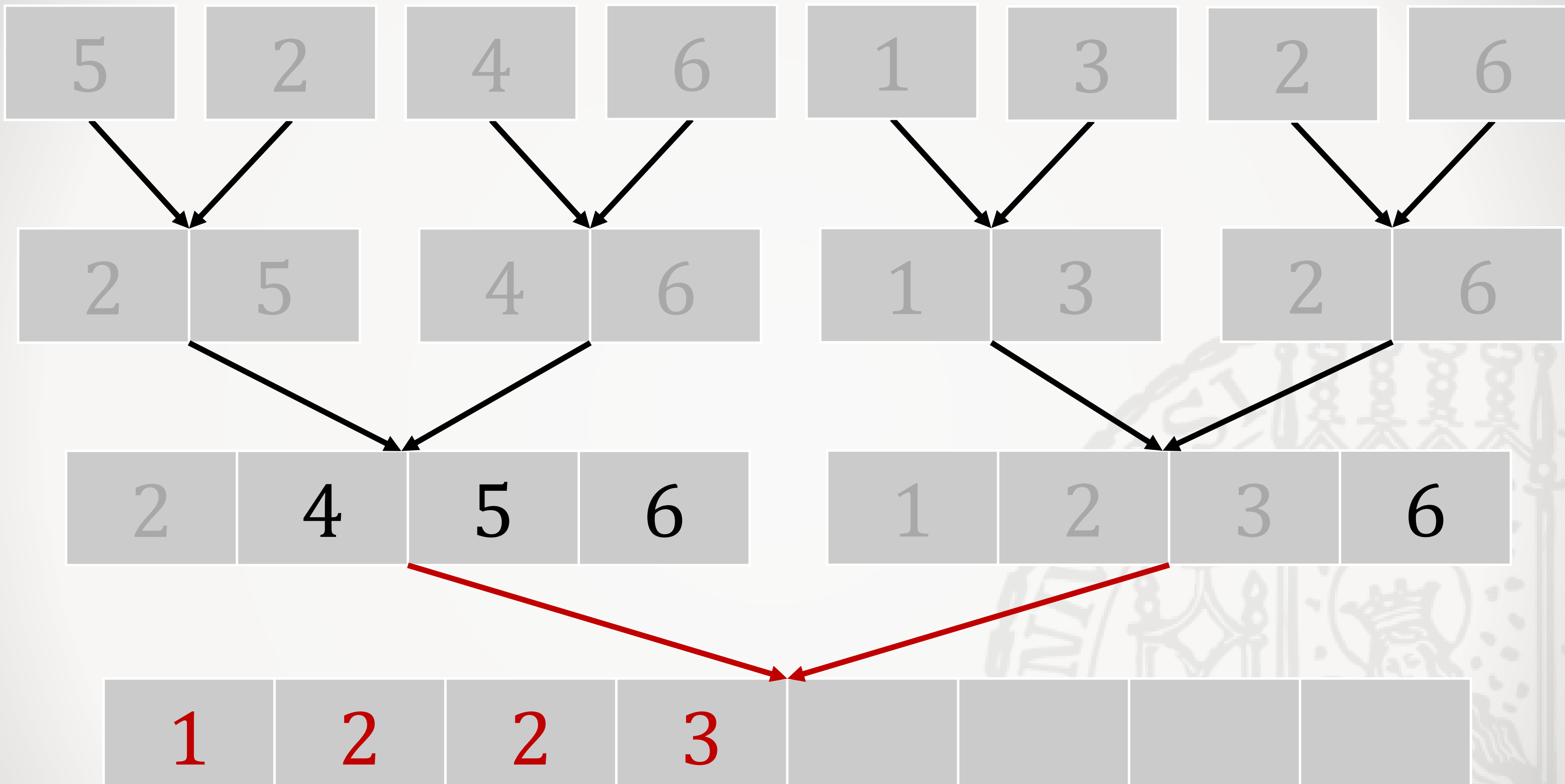
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?$ $2 < 2?$ $2 < 4?$

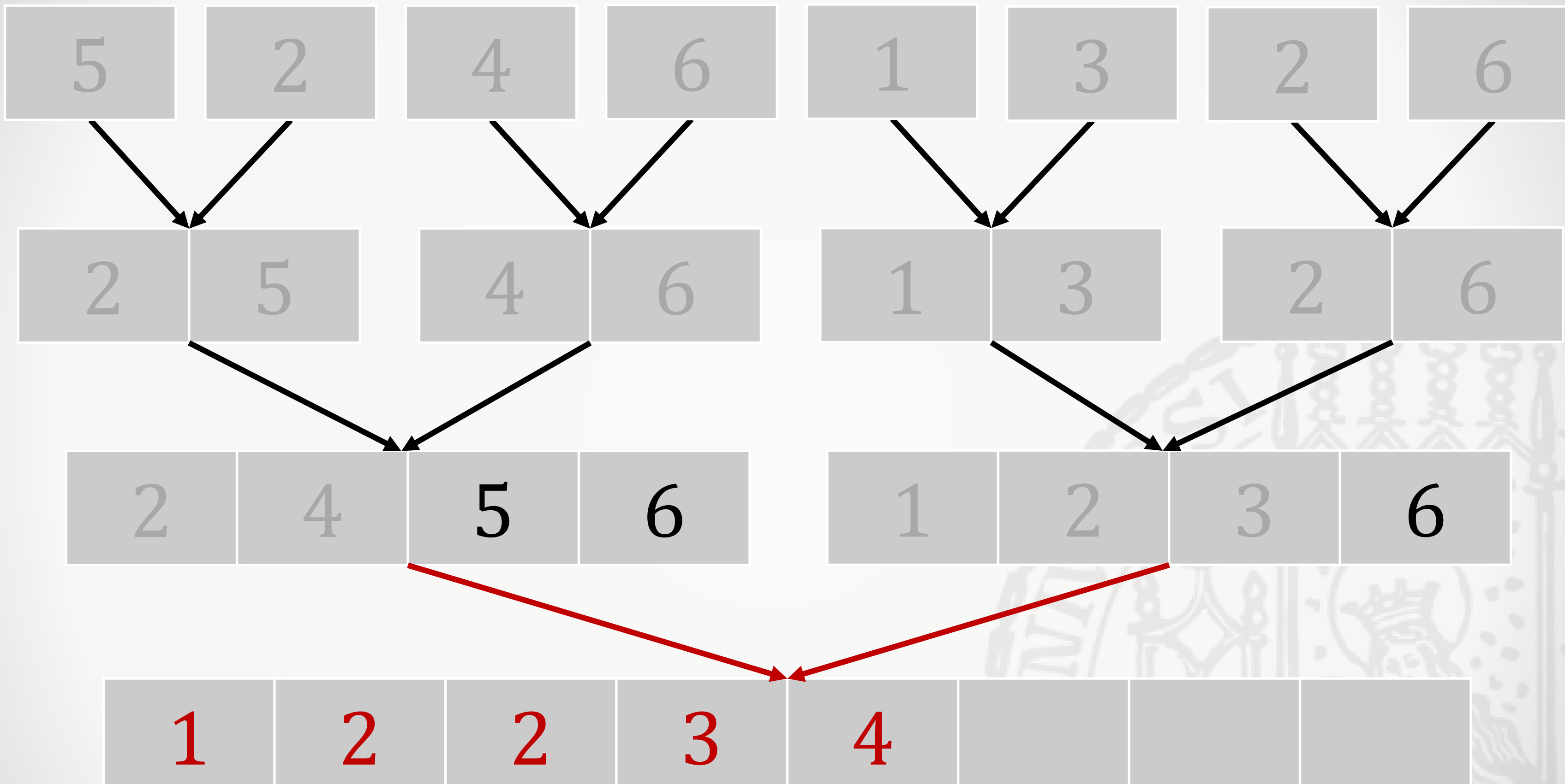
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?$ $2 < 2?$ $2 < 4?$ $4 < 3?$

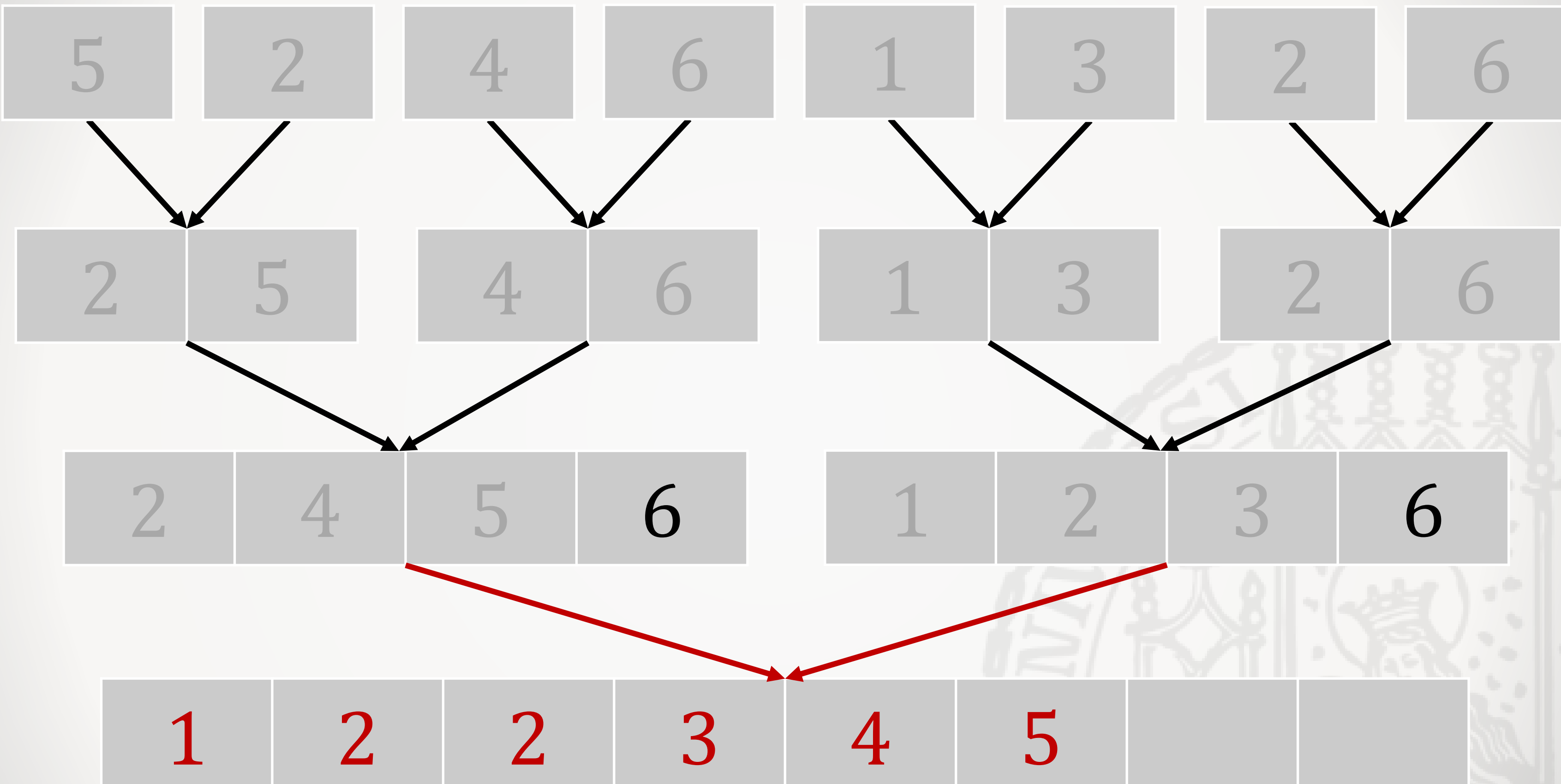
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?$ $2 < 2?$ $2 < 4?$ $4 < 3?$ $4 < 6?$

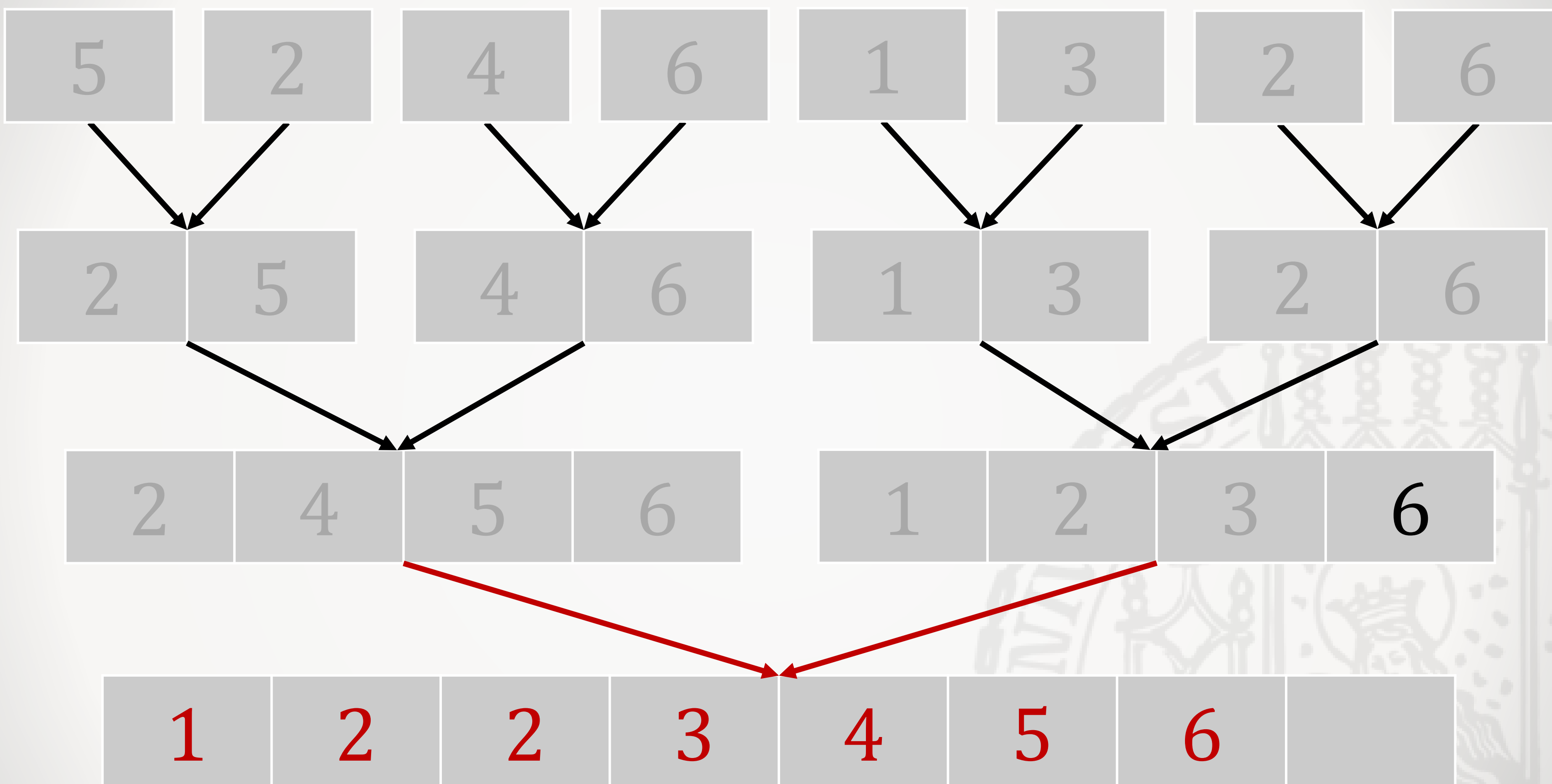
MergeSort: Merge



Vergleiche: $4 + 6$

$2 < 1?$ $2 < 2?$ $2 < 4?$ $4 < 3?$ $4 < 6?$ $5 < 6?$

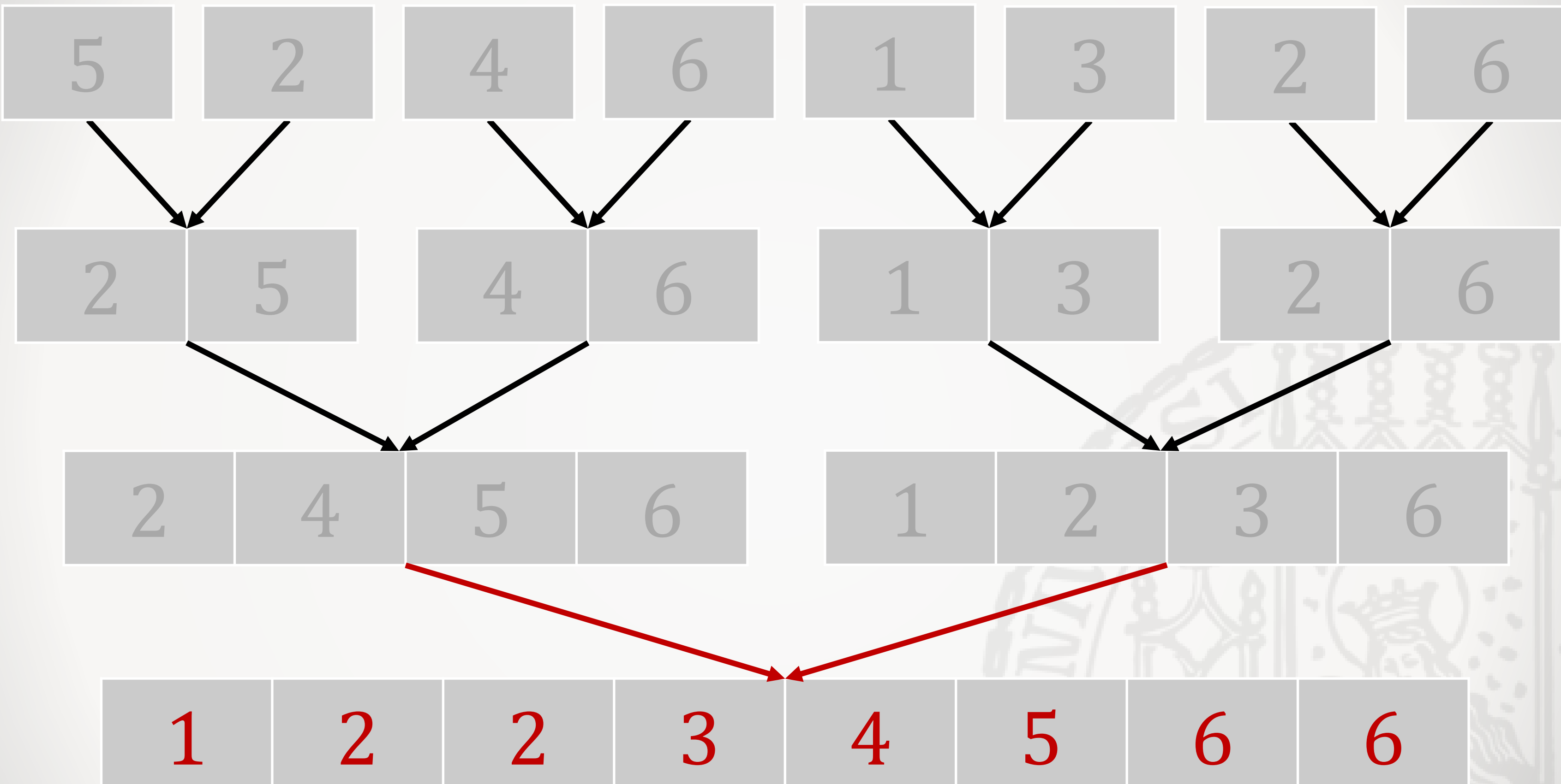
MergeSort: Merge



Vergleiche: 4 + 6

2 < 1? 2 < 2? 2 < 4? 4 < 3? 4 < 6? 5 < 6? 6 < 6?

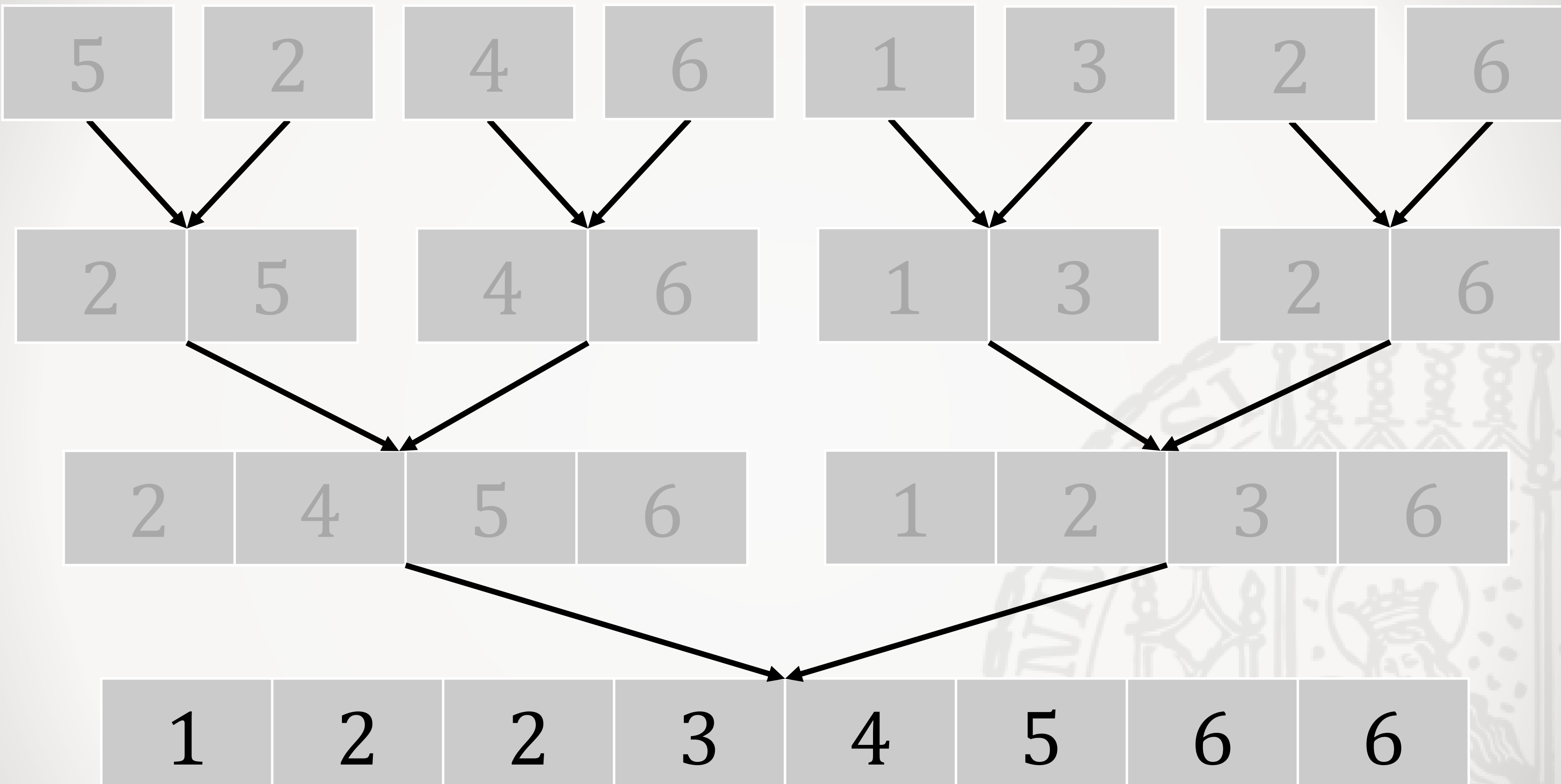
MergeSort: Merge



Vergleiche: 4 + 6

$2 < 1?$ $2 < 2?$ $2 < 4?$ $4 < 3?$ $4 < 6?$ $5 < 6?$ $6 < 6?$

MergeSort: Merge



Vergleiche: $4 + 6 + 7 = 17$ (InsertionSort: 18)

$2 < 1?\ 2 < 2?\ 2 < 4?\ 4 < 3?\ 4 < 6?\ 5 < 6?\ 6 < 6? \Rightarrow$ 7 Vergleiche

MergeSort: Implementierung mit Listen

- Implementierung meist mit temporärem Hilfsarray oder mit Listen

```
void mergesort(int[] a) {
  mergesort(a, 0, a.length);
}

void mergesort(int[] a, int l, int r) {
  if (l < r) {
    int q = (l + r) / 2;
    mergesort(a, l, q);
    mergesort(a, q + 1, r);
    merge(a, l, q, r);
    }
}
```

```
void merge(int[] a, int l, int q, int r) {
  int[] arr = new int[a.length];
  int i, j;
  for (i = l; i <= q; i++)
    arr[i] = a[i];
  for (j = q + 1; j <= r; j++)
    arr[r + q + 1 - j] = a[j];

  i = l;
  j = r;
  for (int k = l; k <= r; k++) {
    if (arr[i] <= arr[j]) {
      a[k] = arr[i];
      i++;
    } else {
      a[k] = arr[j];
      j--;
    }
  }
}
```

MergeSort: Implementierung mit Hilfsarray

- Implementierung in-place möglich, aber nicht trivial
- Stattdessen oft mit temporären Hilfsarray oder meist mit Listen

```
void mergesort(int[] a) {
  mergesort(a, 0, a.length);
}


void mergesort(int[] a, int l, int r) {
  if (l < r) {
    int q = (l + r) / 2;
    mergesort(a, l, q);
    mergesort(a, q + 1, r);
    merge(a, l, q, r);
    }
}
```

Doppelter Speicheraufwand skaliert immer noch linear, da $2n \in O(n)$

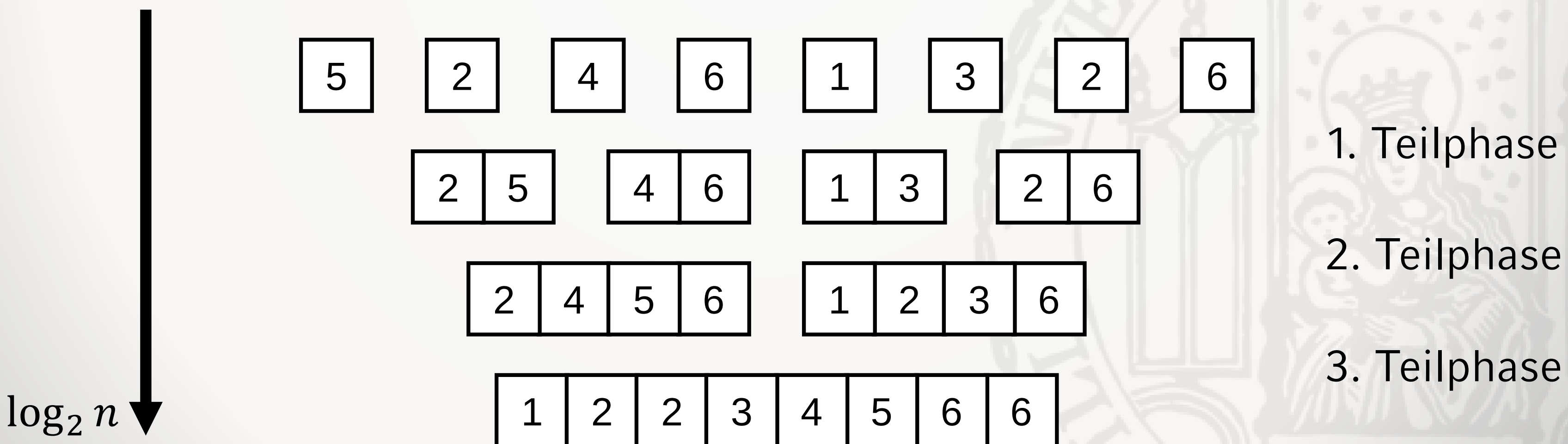
```
void merge(int[] a, int l, int q, int r) {
  int[] arr = new int[a.length];
  int i, j;
  for (i = l; i <= q; i++)
    arr[i] = a[i];
  for (j = q + 1; j <= r; j++)
    arr[r + q + 1 - j] = a[j];

  i = l;
  j = r;
  for (int k = l; k <= r; k++) {
    if (arr[i] <= arr[j]) {
      a[k] = arr[i];
      i++;
    } else {
      a[k] = arr[j];
      j--;
    }
  }
}
```

MergeSort: Komplexitätsanalyse

Erinnerung:
Binärbaum mit n Knoten hat maximal Höhe $h = \log_2 n$.

- Wieviele Vergleiche werden benötigt:
  - In der Divide-Phase wird nichts verglichen
  - In der Merge-Phase werden maximal $\log_2 n$ Teilphasen (Baumebenen im Beispiel) durchgeführt.
  - Die i-te Teilphase besteht maximal aus $\frac{n}{2^{i+1}}$ Mergeoperationen
  - In jeder Mergeoperation kann es zu $2^i - 1$ Vergleichen kommen (Bei „Verzahnung“, z.B.: [1,3,5,7] [2,4,6,8])



QuickSort

- Idee:
  - Wähle ein Element der Folge p (Pivotelement) aus.
  - Zerlege die Folge in zwei Teilfolgen:
    - Teilfolge 1: Enthält alle Elemente $x \leq p$
    - Teilfolge 2: Enthält alle Elemente $x > p$
  - Sortiere rekursiv auf den Teillisten mit Quicksort.
  - Einelementige Listen sind schon sortiert.
  - Im Gegensatz zu Mergesort:
    - Aufwand liegt im Divide-Schritt.
    - Merge-Schritt ist lediglich Konkatenation.

QuickSort: Beispiel

Pivot:
Wir benutzen hier das erste Listenelement als Pivot!

5	2	4	6	1	3	2	6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das erste Listenelement als Pivot!

5	2	4	6	1	3	2	6

2	4	1	3	2

5

6	6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das erste Listenelement als Pivot!

5	2	4	6	1	3	2	6

2	4	1	3	2		5		6	6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das erste Listenelement als Pivot!

5 2 4 6 1 3 2 6

2 4 1 3 2 | 5 | 6 6

1 2 | 2 | 4 3 | 5 | 6 | 6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das erste Listenelement als Pivot!

5	2	4	6	1	3	2	6

2	4	1	3	2	5	6	6

1	2	2	4	3	5	6	6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das erste Listenelement als Pivot!

5	2	4	6	1	3	2	6
2	4	1	3	2	5	6	6
1	2	2	4	3	5	6	6
1	2	2	3	4	5	6	6

QuickSort: Beispiel

Pivot:
Wir benutzen hier das erste Listenelement als Pivot!

5 2 4 6 1 3 2 6

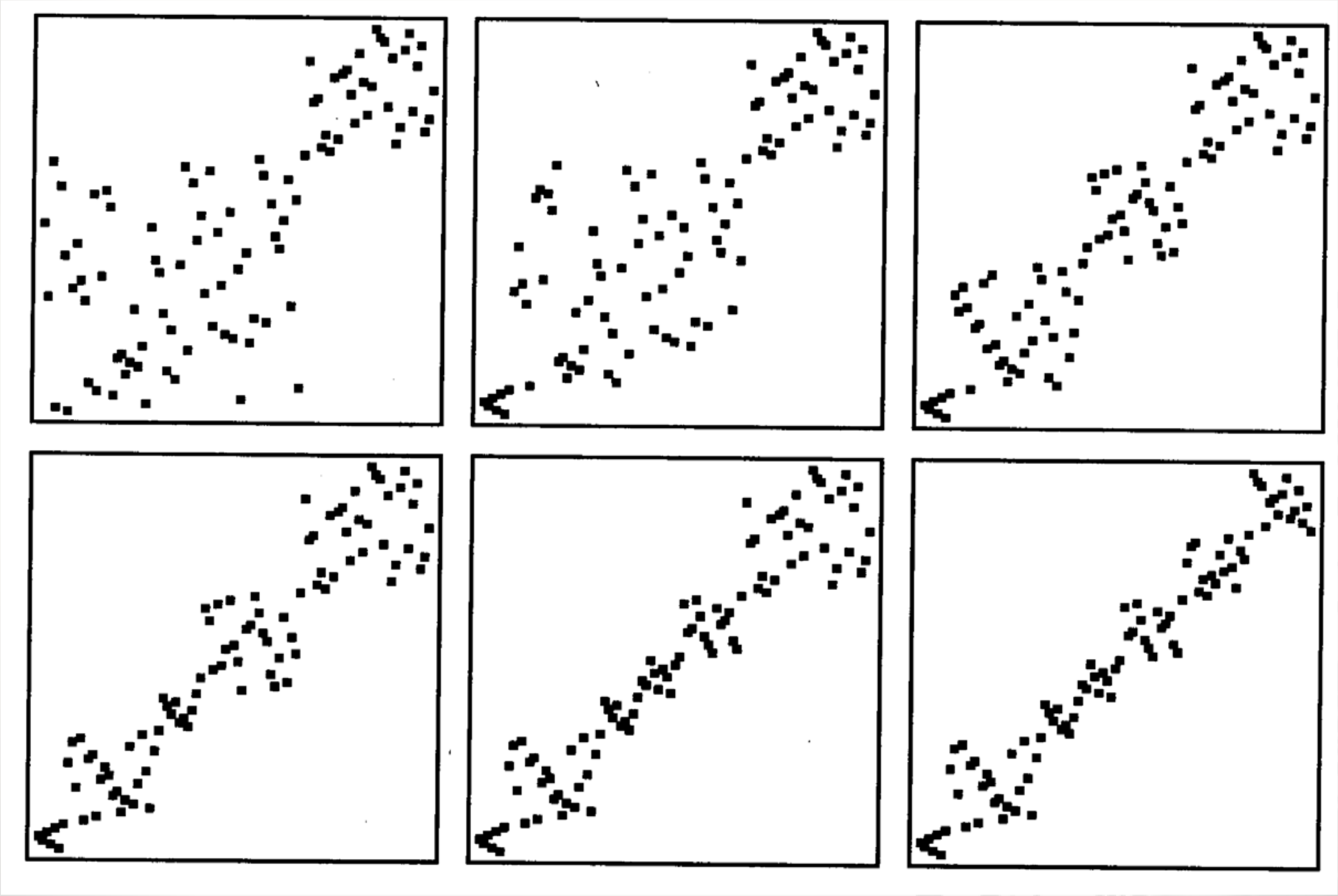
2 4 1 3 2   5   6 6

1 2   2   4 3   5   6   6

1   2   2   3   4   5   6   6

1 2 2 3 4 5 6 6

QuickSort: Visualisierung



QuickSort: Implementierung

- Wir wählen das mittlere Element als Pivot (sonst vertauschen).

```
void quicksort(int[] a) {
  quicksort(a, 0, a.length - 1);
}

void quicksort(int[] a, int left, int right) {
  int i = left;
  int j = right;
  int pivot = a[i + (j - i)/2];
  while(i <= j) {
    while(a[i] < pivot) i++;
    while(a[j] > pivot) j--;
    if(i <= j) {
      swap(a, i, j);
      i++;
      j--;
    }
  }
  if(left < j)  quicksort(a, left, j);
  if(i < right) quicksort(a, i, right);
}
```

← Pivot

Zwei Iteratoren laufen gegenläufig in Richtung Mitte/Pivot

Falls linkes Element kleiner und rechtes Element größer als Pivot ist, tausche beide Elemente

← ← Rekursionen

QuickSort: Komplexitätsanalyse Best-Case

- Wir betrachten die Anzahl der Vergleichsoperationen.
- Best-Case: Die Folge zerfällt immer in zwei gleich große Teile.
- Das Pivot wird mit allen übrigen Elementen verglichen.
- Die beiden Teilfolgen enthalten nicht mehr das Pivot.
- Beide Teilfolgen werden ebenfalls sortiert.
- Wenn eine Folge nur noch aus einem Element besteht, werden keine Vergleiche mehr benötigt.

$$T(n) = \begin{cases} 2 \cdot T\left(\frac{n-1}{2}\right) + n - 1 & , n > 1 \\ 0 & , n = 1 \end{cases}$$

Mit Mastertheorem gilt $T(n) \in O(n \log n)$.

QuickSort: Komplexitätsanalyse Worst-Case

- Die Folge zerfällt schlimmstenfalls immer in
  - eine leere Folge
  - und eine Folge der Länge $n-1$.
- Der Aufrufbaum ist somit ein entarteter Baum (ein linearer Zweig)

$$T(n) = \begin{cases} T(n-1) + n - 1 & , n > 1 \\ 0 & , n = 1 \end{cases}$$

Mit Mastertheorem gilt $T(n) \in O(n^2)$.

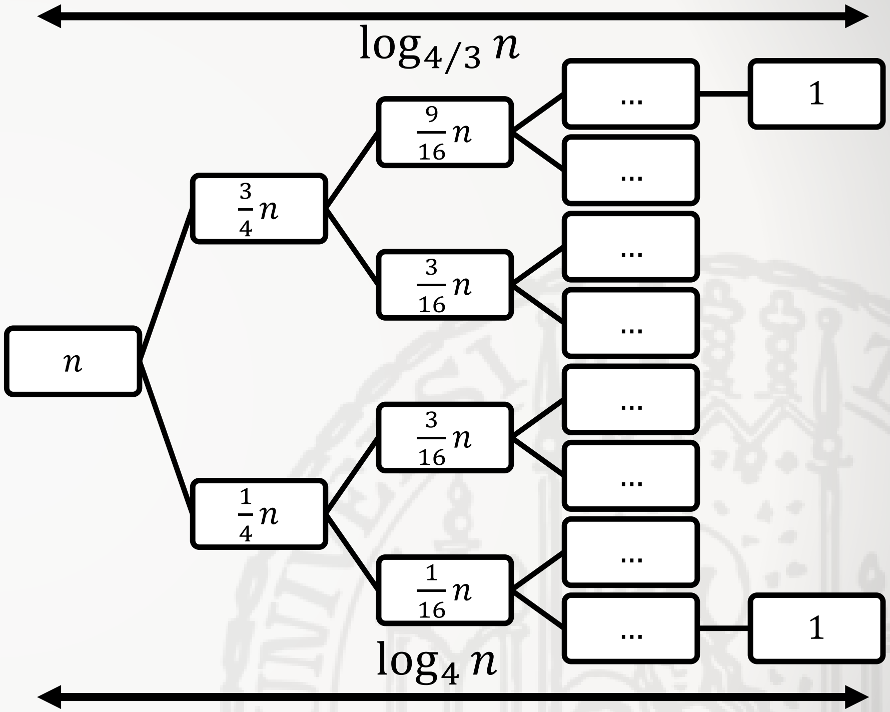
- Im schlimmsten Fall ist QuickSort damit genauso langsam wie BubbleSort / SelectionSort / InsertionSort

QuickSort: Komplexitätsanalyse Average-Case (1/2)

- Annahmen:
  - Alle n Schlüssel seien verschieden.
  - Die Wahrscheinlichkeit der Eingabeanordnung sei $\frac{1}{n!}$.
  - Jede der $n!$ Permutationen ist damit gleich wahrscheinlich.
- Bei der Partitionierung entstehen zwei Teilfolgen, die Längen im Interval $(0, n-1)$ annehmen können. Jede Länge ist gleich wahrscheinlich. Der Möglichkeitsraum der Längenpaare ist daher
$$\{(0, n-1), (1, n-2), \dots, (n-2,1), (n-1,0)\}$$
- Im Durchschnitt ist daher die Länge der größeren Teilfolge:
$$L_1 = \frac{2}{n} \sum_{i=\frac{n}{2}}^{n-1} i = \frac{3}{4} n - \frac{3}{2}$$
- Wir können die Länge zu $L_1 = \frac{3}{4} n$ abschätzen, da es dadurch nur langsamer wird.

QuickSort: Komplexitätsanalyse Average-Case (2/2)

- $L_1 = \frac{3}{4}n, L_2 = \frac{1}{4}n$
- Der längste Rekursionspfad besteht aus $\log_{4/3} n$ Knoten.
- Jedes Level des Aufrufbaums nutzt $O(n)$ Vergleiche
- Damit ergeben sich $O(n \log n)$ Vergleiche im durchschnittlichen Fall.



QuickSort: Pivotelement

- Pivot (frz.): Dreh-/Angelpunkt
- Die Wahl des Pivots beeinflusst stark die Ausführung und Rekursionstiefe des Algorithmus.
  - Beispiel: Sortiere 7-elem. Liste und nutze 1. Element als Pivot.

4	2	1	3	6	7	5
2	1	3	4	6	7	5
1	2	3	4	5	6	7
1	2	3	4	5	6	7

1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

- Wie Pivot wählen, damit Worst-Case vermieden wird?

QuickSort: Pivotelement einfache Auswahlstrategien

- „Wähle immer erstes Element“ $p = x_1$
  - Einfach zu implementieren
  - (Teilweise) vorsortierte Listen führen zu schlechter Laufzeit
- „Wähle immer letztes Element“ $p = x_n$
  - Gleiche Eigenschaften wie „ Wähle immer erstes Element“
- „Wähle immer mittleres Element“ $p = x_{[n/2]}$
  - Ebenfalls einfach zu implementieren
  - (Teilweise) vorsortierte Listen nicht kritisch
  - Auch hier lassen sich Worst-Case-Szenarien erstellen
    - Sortiere 6421357

6	4	2	1	3	5	7
1	6	4	2	3	5	7
1	2	6	4	3	5	7
1	2	3	6	4	5	7
1	2	3	4	6	5	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

QuickSort: Pivotelement Randomized

- Bei deterministischer Wahl eines bestimmten Elements lässt sich stets ein Worst-Case finden.

- Lösung 1: Nicht-deterministisches QuickSort
  - Wähle zufälliges Element als Pivot
  - In der Praxis weit verbreitet (einfach und schnell)
  - Wahrscheinlichkeit für Worst-Case sehr unwahrscheinlich

- Wie sieht eine deterministische optimale Strategie aus?
  - Beide Teilfolgen sollten gleich groß sein.
  - Optimales Pivotelement ist damit der Median.

QuickSort: Pivotelement Median

- Naiver Algorithmus $median$
  - Gegeben ist eine Folge $X = x_1 x_2 \dots x_n$
  - Sortiere X
  - Gib Element an Position $\lceil n/2 \rceil$ aus

- Aufwand: Worst-Case $O(n \log n)$ bis $O(n^2)$ wegen Sortieren
  - QuickSort hätte dann $O(n \log^2 n)$.

- Geht das besser?
  Können wir das k kleinste Element in linearer Zeit finden?

SELECT - Algorithmus

- Bestimme Median-of-Medians (MoM):
  - Partitioniere die Folge in Subfolgen der Länge 5 $\Rightarrow O(n)$.
  - Bestimme in allen 5er-Blöcken den Median $\Rightarrow O(1)$.
  - Bestimme rekursiv den Median $\Rightarrow T(n) \geq T(n/5)$.
- Arrangiere die Folge so, dass alle kleineren Elemente links und alle größeren Elemente rechts vom MoM stehen $\Rightarrow O(n)$.
- Falls der Index des MoM gleich $\lceil n/2 \rceil$ ist, dann haben wir den Median gefunden $\Rightarrow O(1)$.
- Falls der Index des MoM kleiner als $\lceil n/2 \rceil$ ist, dann befindet sich der Median im rechten Feld. Wiederhole den Algorithmus dort.
- Falls der Index größer als $\lceil n/2 \rceil$ ist, dann befindet sich der Median im linken Feld. Wiederhole den Algorithmus dort $\Rightarrow T(n) = ?$.

SELECT - Algorithmus: Beispiel

- Median von [5, 2, 17, 13, 14, 7, 16, 4, 2, 18, 10, 8, 4, 11]
- 5er Blöcke:
  - [5, 2, 17, 13, 14] [7, 16, 4, 2, 18] [10, 8, 4, 11]
  - Mediane: 13, 7, 8
  - Medians-of-Medians: 7
- Neu anordnen: [5, 2, 4, 2, 4, 7, 8, 17, 13, 14, 16, 18, 10, 11]
- Echter Median müsste an Index [14/2] stehen, daher kann 7 nicht der Median sein (steht bei Index 6).
  Suche also weiter in rechter Folge:

$$[8, 17, 13, 14, 16, 18, 10, 11]$$

SELECT - Algorithmus: Beispiel (2)

- Median von [8, 17, 13, 14, 16, 18, 10, 11]
- 5er Blöcke:
  - [8, 17, 13, 14, 16] [18, 10, 11]
  - Mediane: $14, 10$
  - Medians-of-Medians: 10
- Neu anordnen: [8, 10, 17, 13, 14, 16, 18, 11]
- In der gesamten Liste:

  $$[5, 2, 4, 2, 4, 7, 8, 10, 17, 13, 14, 16, 18, 11]$$

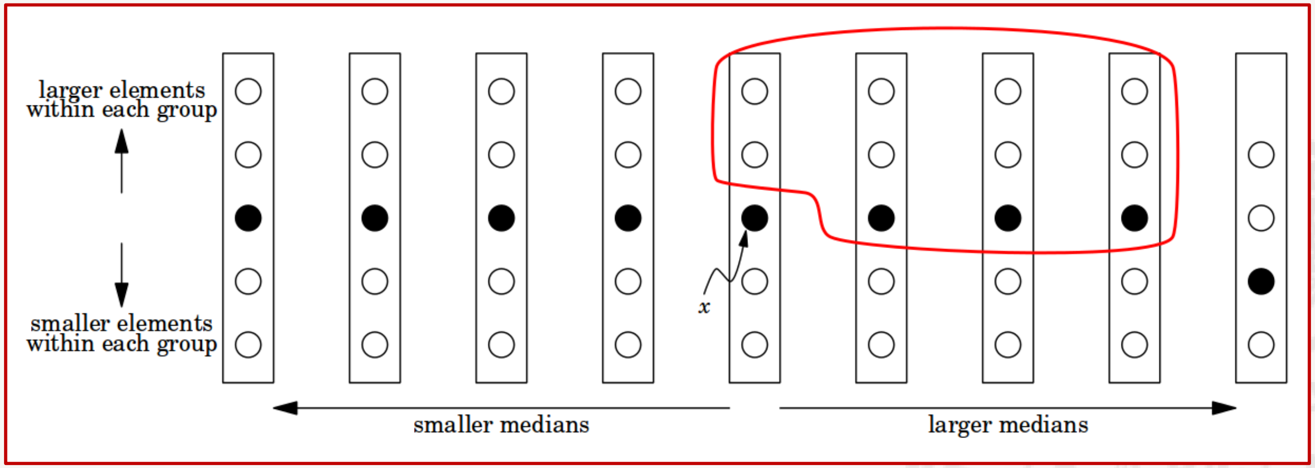
- Echter Median müsste an Index $\lfloor 14/2 \rfloor$ stehen, daher kann 10 nicht der Median sein (steht bei Index 8).
  Suche also weiter in linker Folge:

  $$[8]$$

- $[5, 2, 4, 2, 4, 7, 8, 10, 17, 13, 14, 16, 18, 11]$
  Die 8 ist an der richtigen Stelle und damit das Ergebnis.

SELECT - Algorithmus: Komplexität

- Wieviele Elemente sind beim Divide-Schritt nach der Median-Kandidatbestimmung schlimmstenfalls kleiner (oder größer) als der Kandidat x?



- Von den $n/5$ Gruppen enthält eine Hälfte mindestens 3 Elemente, die größer oder gleich x sind.
- Die Ausnahmen sind zwei Gruppen:
  - Die Gruppe, die x enthält
  - Die letzte evtl. unvollständige Gruppe

SELECT - Algorithmus: Komplexität (2)

- Daher sind mindestens $3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 2 \right\rceil\right) \geq \frac{3n}{10} - 6$ Elemente größer als x (und analog auch genauso viele Elemente kleiner).
- Im schlimmsten Fall ergibt sich damit für die noch ungeklärte Komplexität des letzten Schritts des Algorithmus:

$$T(n) = n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$$

- Insgesamt:

$$T(n) = \begin{cases} O(1), & n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n), & n \geq 140 \end{cases}$$

SELECT - Algorithmus: Komplexität (3)

$$T(n) = \begin{cases} O(1), & n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n), & n \geq 140 \end{cases}$$

- Angenommen, $T(n) \leq cn$ für eine sehr große Konstante c und alle $n < 140$. Eine solche Konstante kann stets gefunden werden.
- Sei a eine passende Konstante, sodass $O(n) \leq an$ für alle $n \in \mathbb{N}$.
- Nun kommt der Induktionsschritt für $n \geq 140$:

$$T(n) \leq c\left\lceil\frac{n}{5}\right\rceil + c\left(\frac{7n}{10} + 6\right) + an$$

$$\leq \frac{cn}{5} + c + \frac{7cn}{10} + 6c + an = cn + \left(-\frac{1}{10}cn + 7c + an\right)$$

$$-\frac{1}{10}cn + 7c + an \leq 0 \Leftrightarrow c \geq 10a\left(\frac{n}{n-70}\right)$$

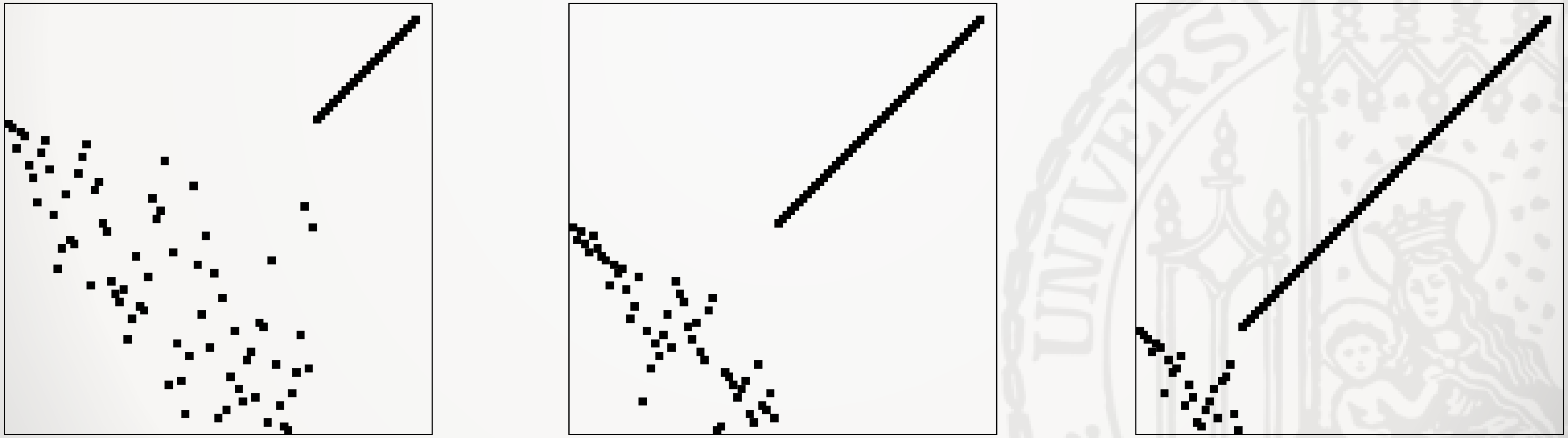
- Da $n \geq 140$, gilt $\frac{n}{n-70} \leq 2$ und wir können $c \geq 20a$ wählen.
- Somit gilt $T(n) \leq cn$ und SELECT ist in $O(n)$

QuickSort: Pivotelement Median

- Nun wissen wir:
  Es geht mit geeigneter Pivot-Auswahlstrategie schneller als $O(n^2)$.
- Suchen wir immer den Median, verhindern wir stets den Worst-Case. QuickSort ist dann in $O(n \log n)$.
- In der Praxis ist dies jedoch unüblich. Der Worst-Case ist statistisch zu unwahrscheinlich.
- Median-of-three ist die am häufigsten angewendete Strategie.
  - Verwende drei Elemente (z.B. $x_1, x_{\lceil n/2 \rceil}, x_n$).
  - Noch besser: Verwende den Median dreier zufälliger Elemente.

HeapSort

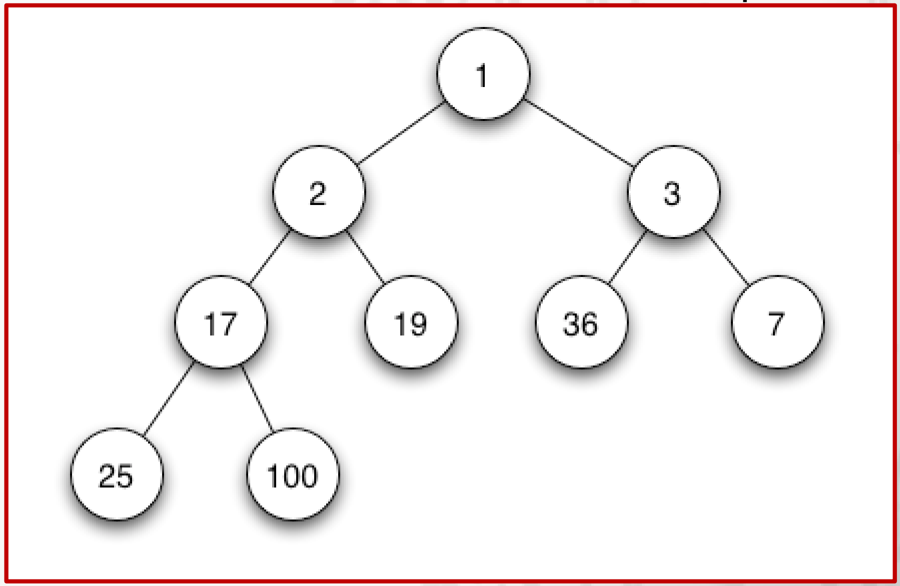
- SelectionSort: Wähle n-mal das jeweils nächste Maximum in $O(n)$
- Falls das immer in $O(\log n)$ ginge, hätten wir ein $O(n \log n)$ Verfahren
- Idee: Stelle das jeweils nächste Maximum in $O(\log n)$ ganz nach vorne.



Heap-Struktur

- Ein Heap ist ein Binärbaum, der folgende Eigenschaften erfüllt:
  - Er ist links-vollständig, d.h. jeder Knoten ist entweder ein Blatt oder hat einen linken Kindknoten.
  - Die Knoten enthalten eine Menge von Schlüsselwerten, auf der eine Totalordnung definiert ist.
  - Für jeden Knoten gilt je nach Heap-Variante:
    - Der Vaterknoten hat einen größeren Schlüsselwert (Max-Heap)
    - Der Vaterknoten hat einen kleineren Schlüsselwert (Min-Heap)

- Ein Heap wird daher auch als partiell geordneter Baum bezeichnet.



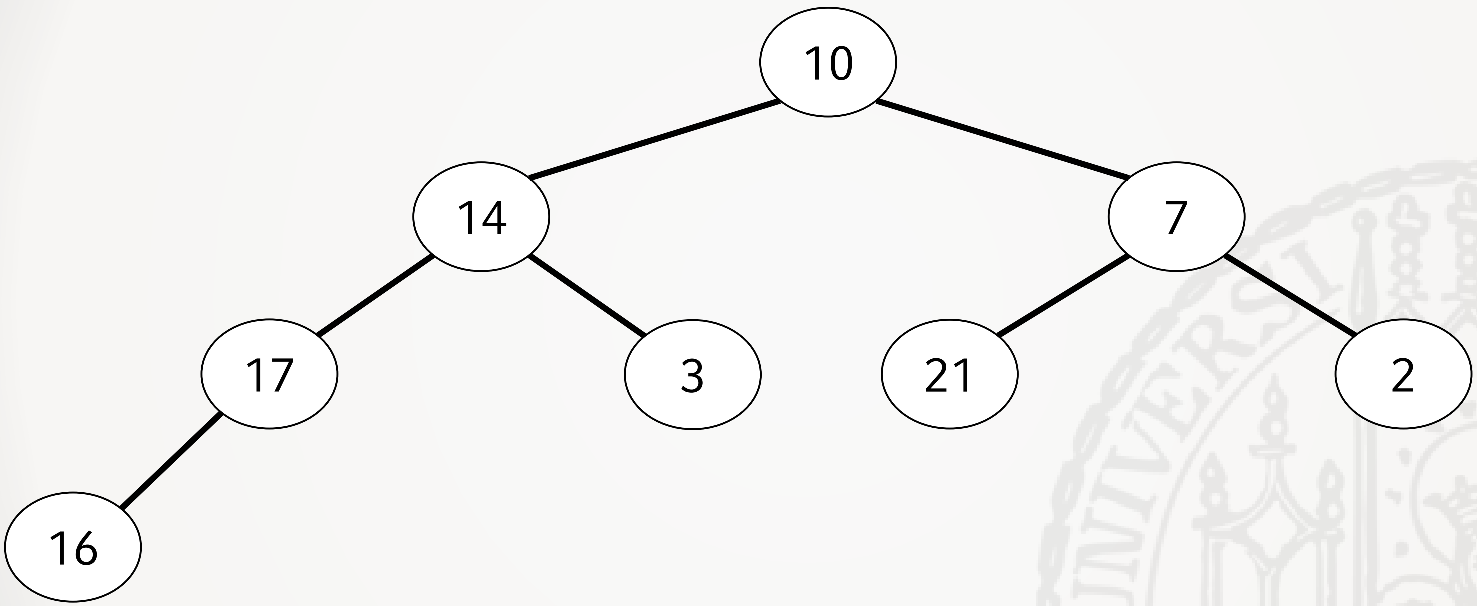
HeapSort: Ablauf

- Füge alle Folgenelemente in einen Binärbaum ein.
- Stelle die Max-Heap-Eigenschaft im Baum her.
  - Von Hinten (Blätter) nach Vorne (Wurzel):
    Prüfe Heap-Eigenschaft; Falls nicht erfüllt, tausche das Vaterelement solange mit dem größeren Kind, bis das Element die Heapeigenschaft erfüllt.
- Für alle Elemente x_i in $(x_0, \dots, x_{n-1})$:
  - Tausche Wurzel (größtes Element) mit dem $(\mathrm{n} - i)$-ten Element.
  - Stelle für alle Elemente kleiner x_i die Heapeigenschaft wieder her (Versickern)

- Gesamter Algorithmus möglich in-place als Arrayeinbettung.

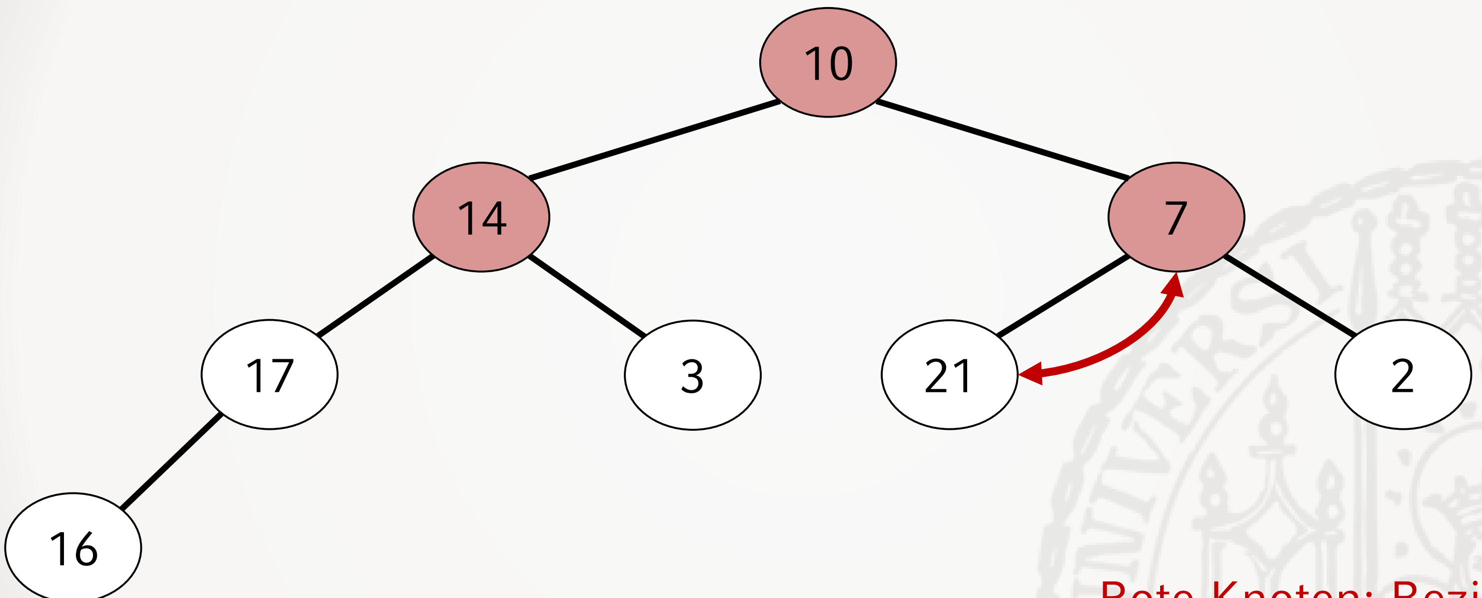
HeapSort: Beispiel

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Einfügen in Binärbaum



HeapSort: Beispiel

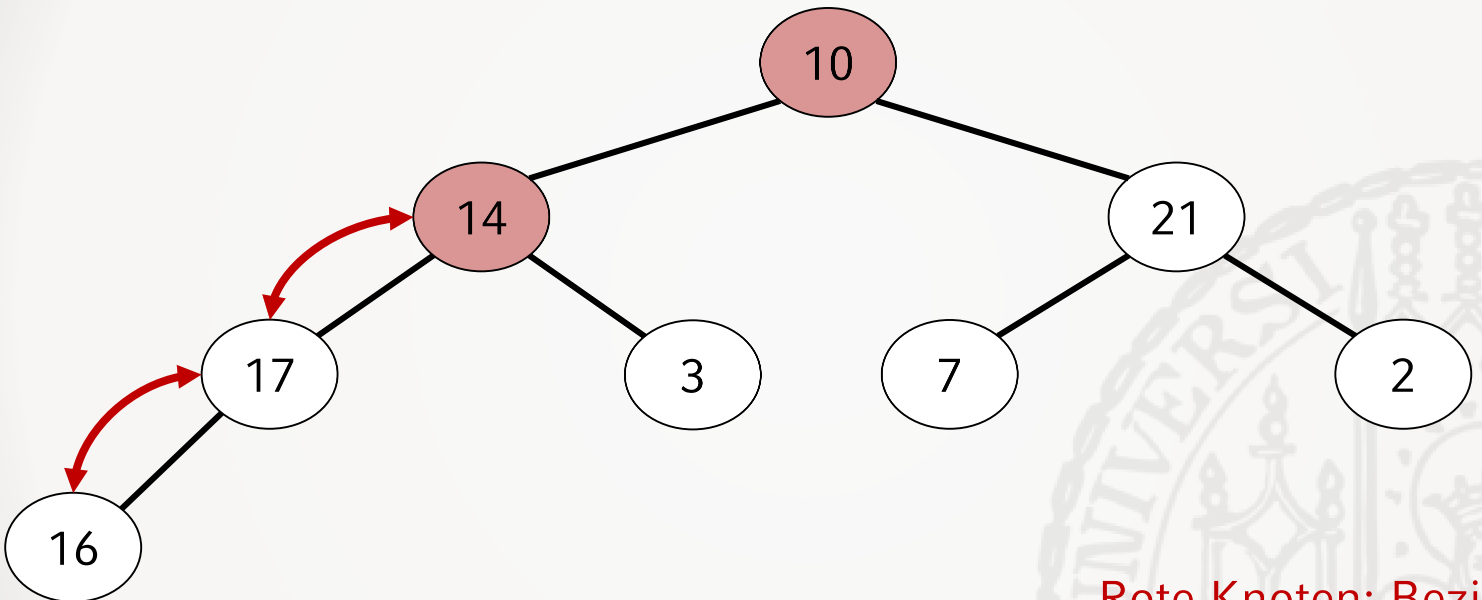
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft herstellen (Heapify)



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

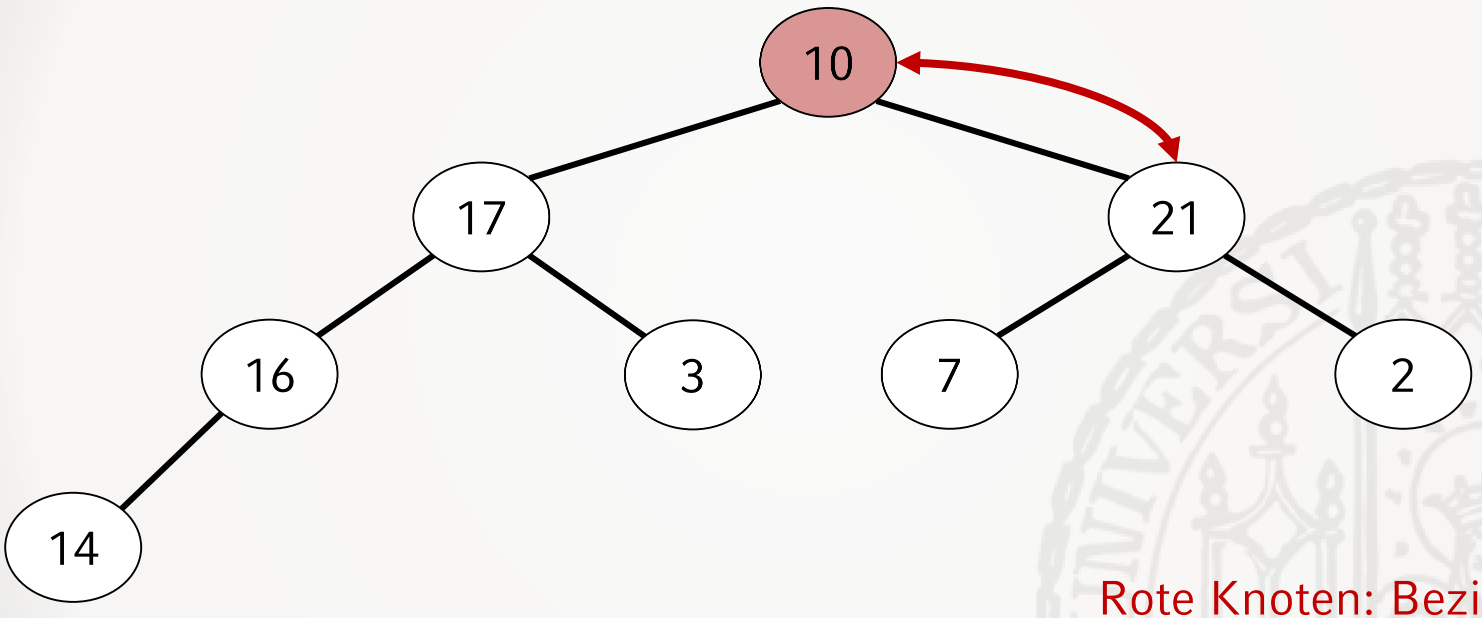
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft herstellen (Heapify)



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

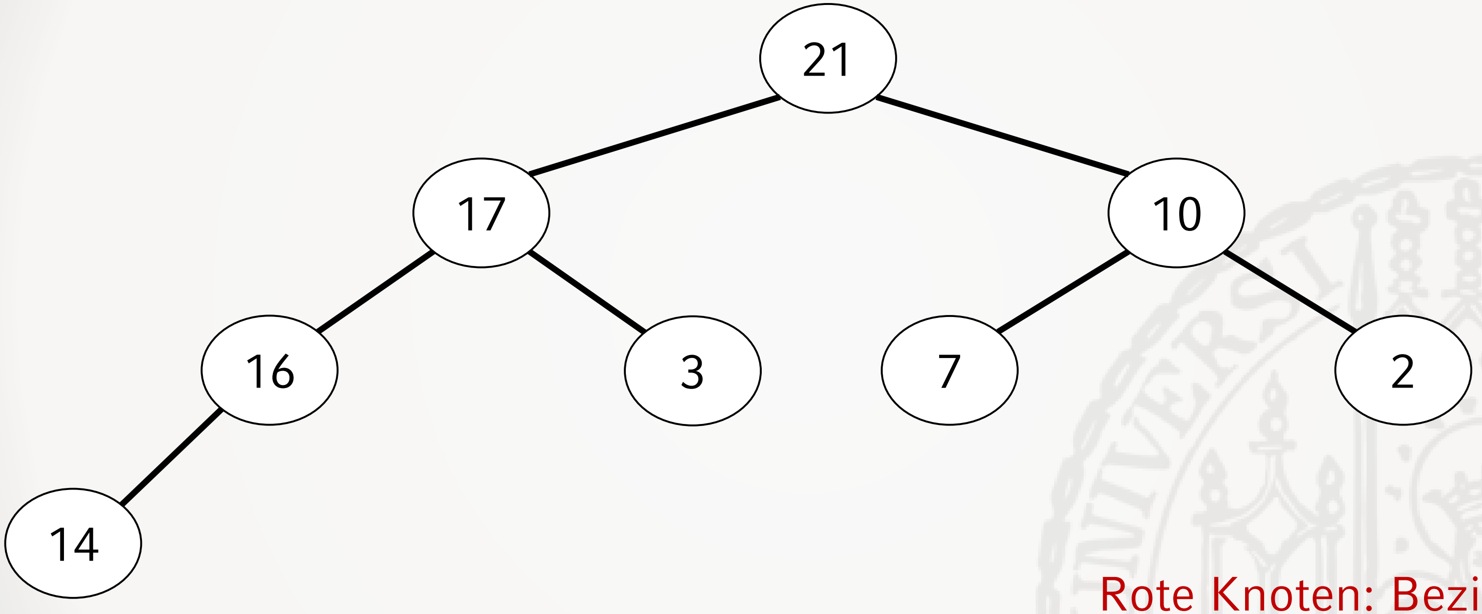
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft herstellen (Heapify)



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

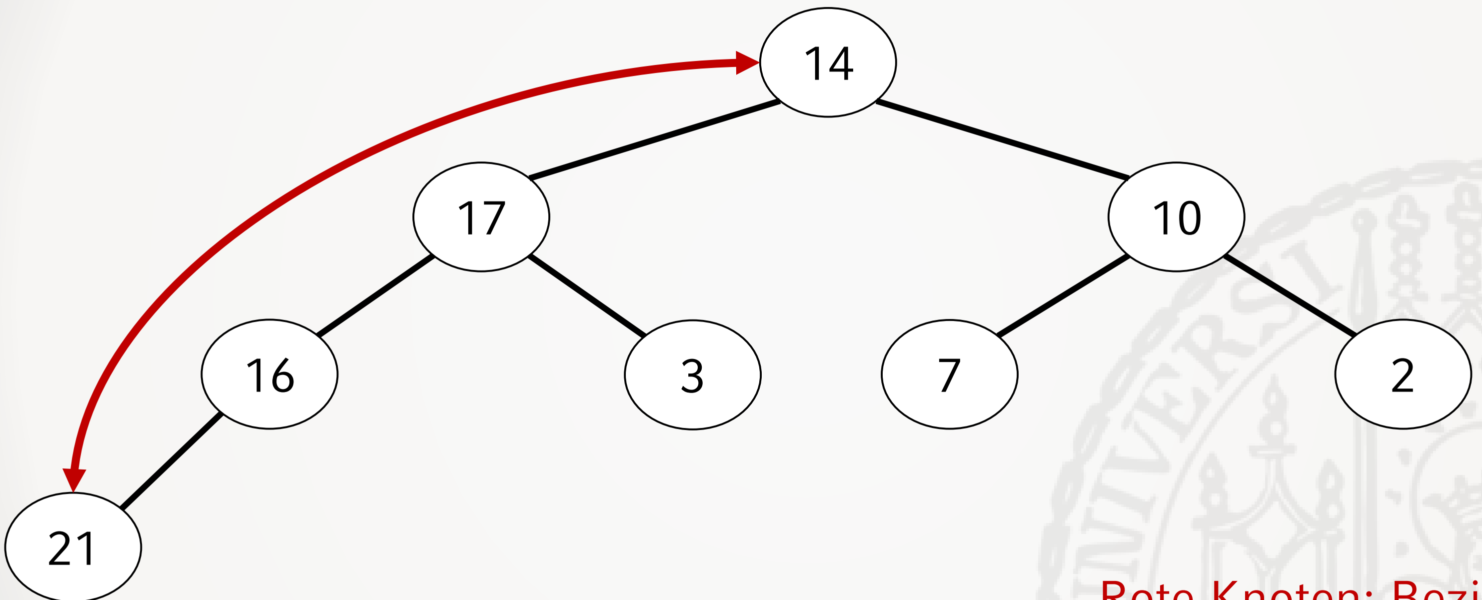
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft hergestellt!



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

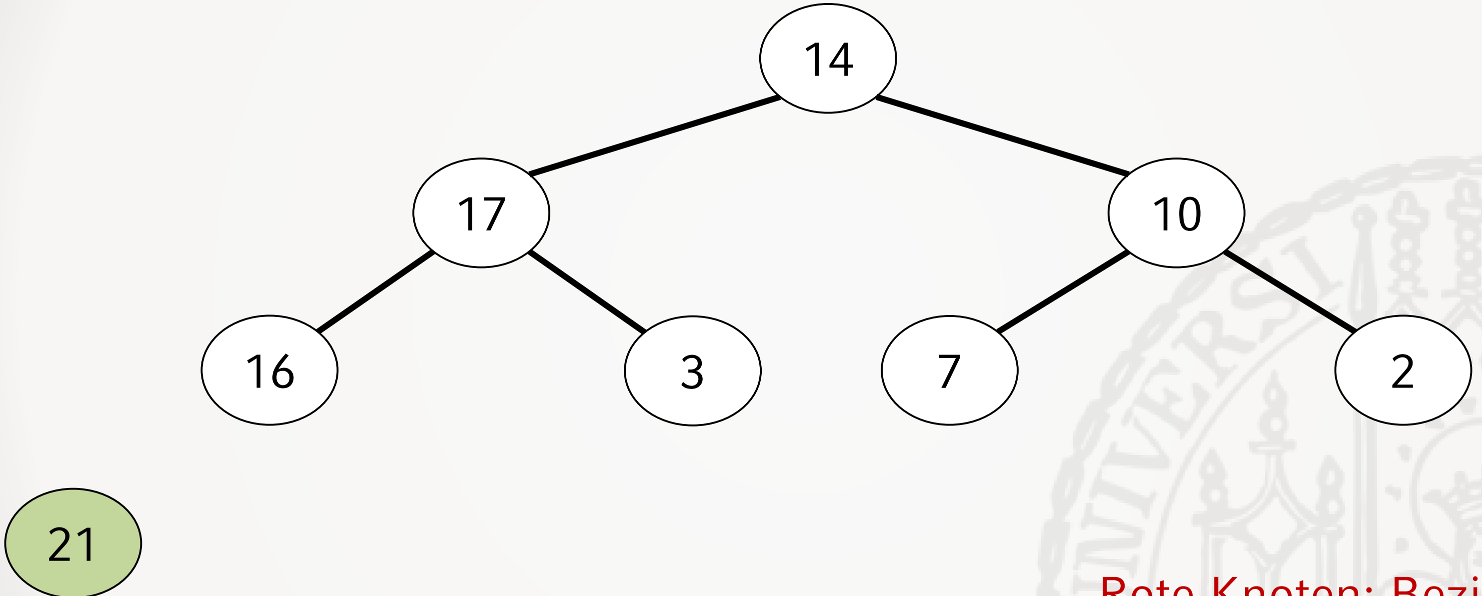
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

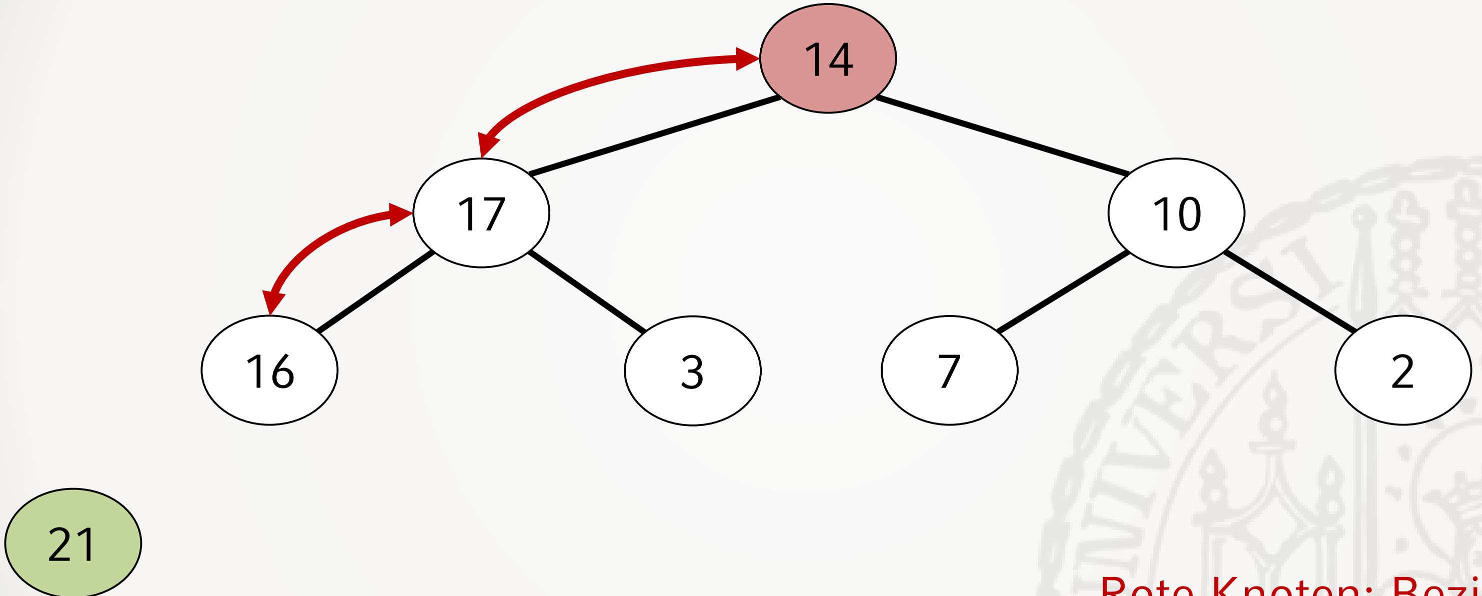
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

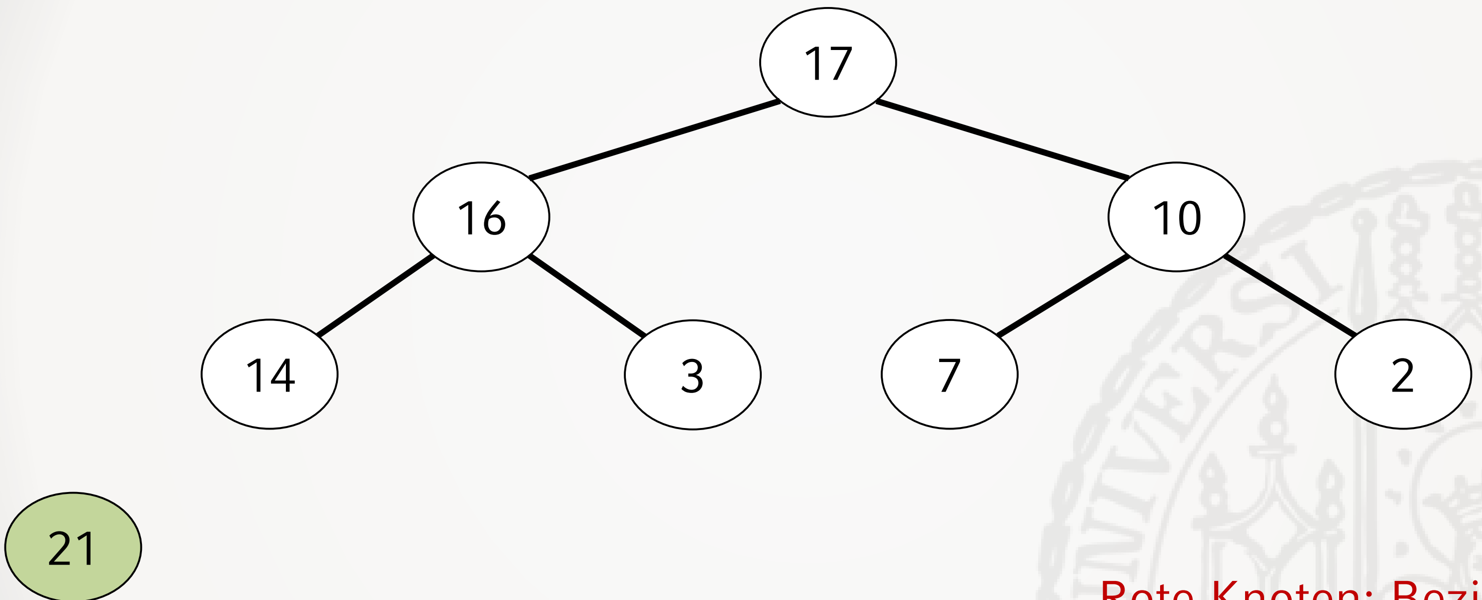
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

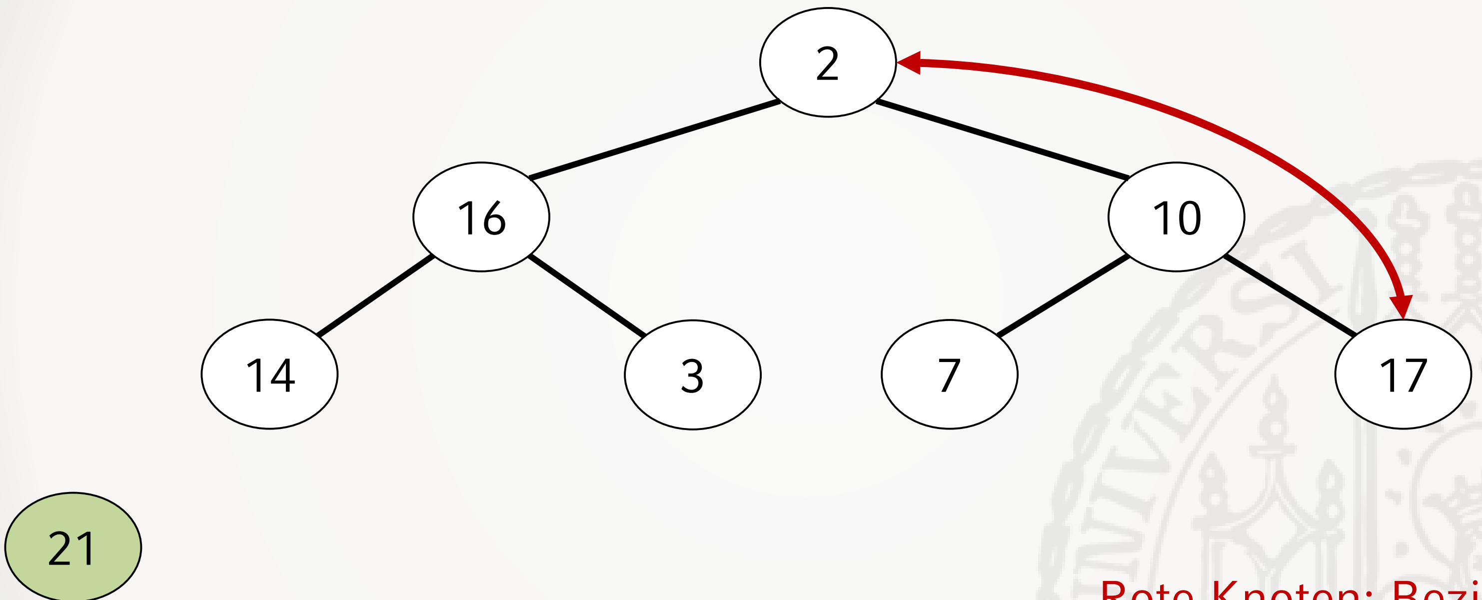
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

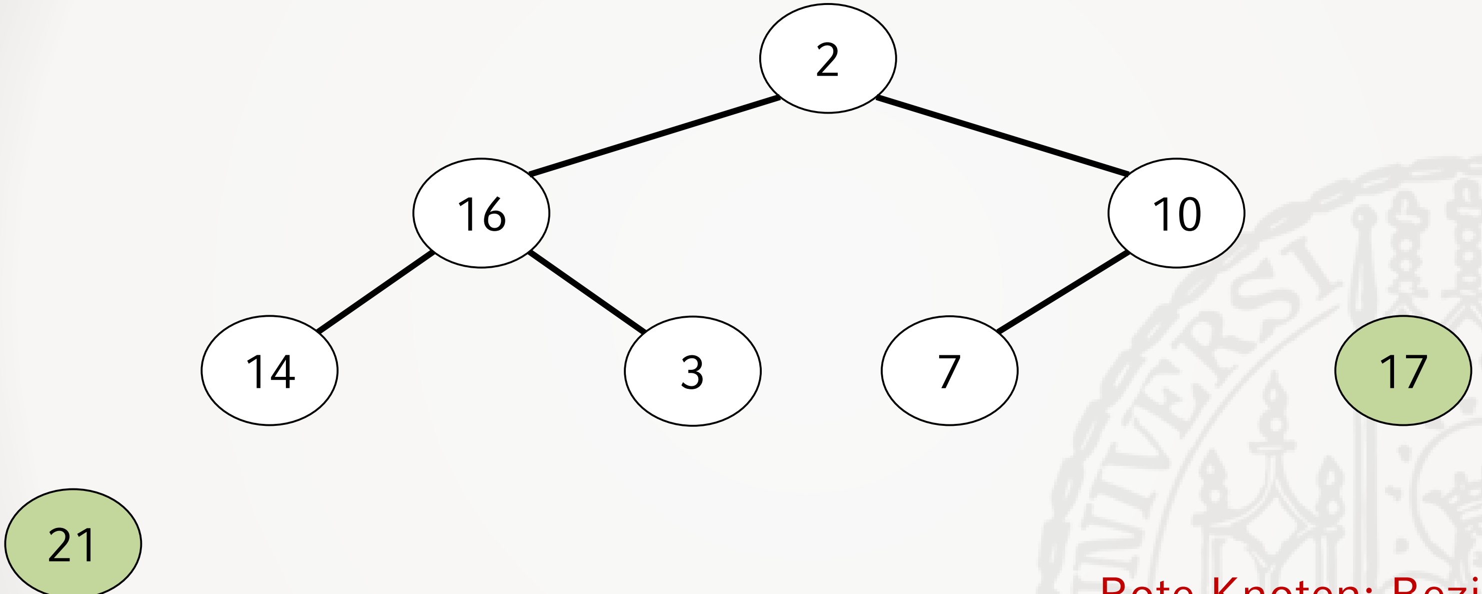
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

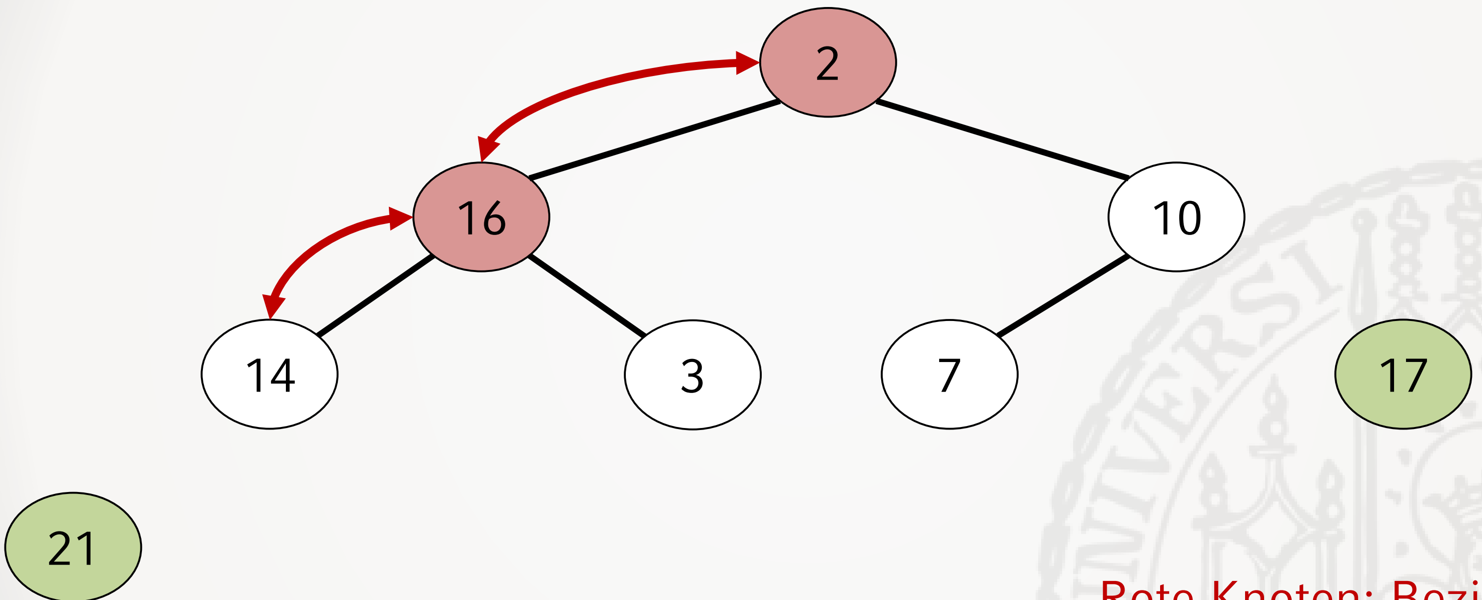
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

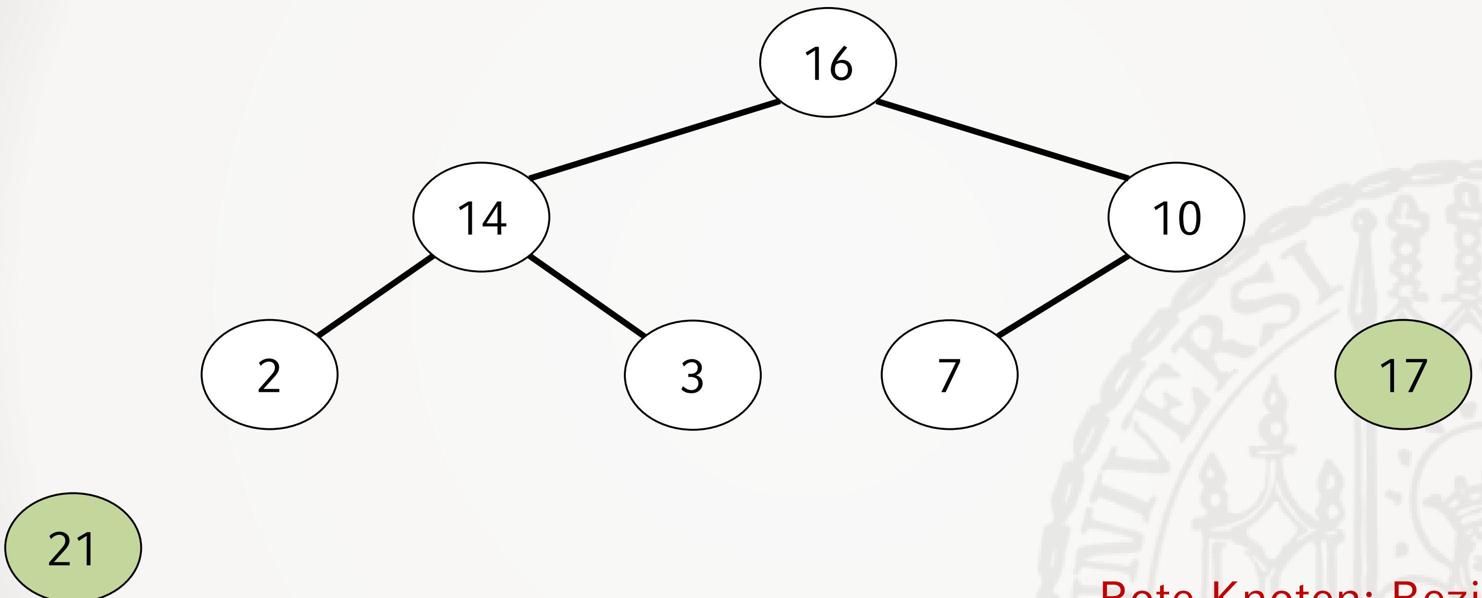
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

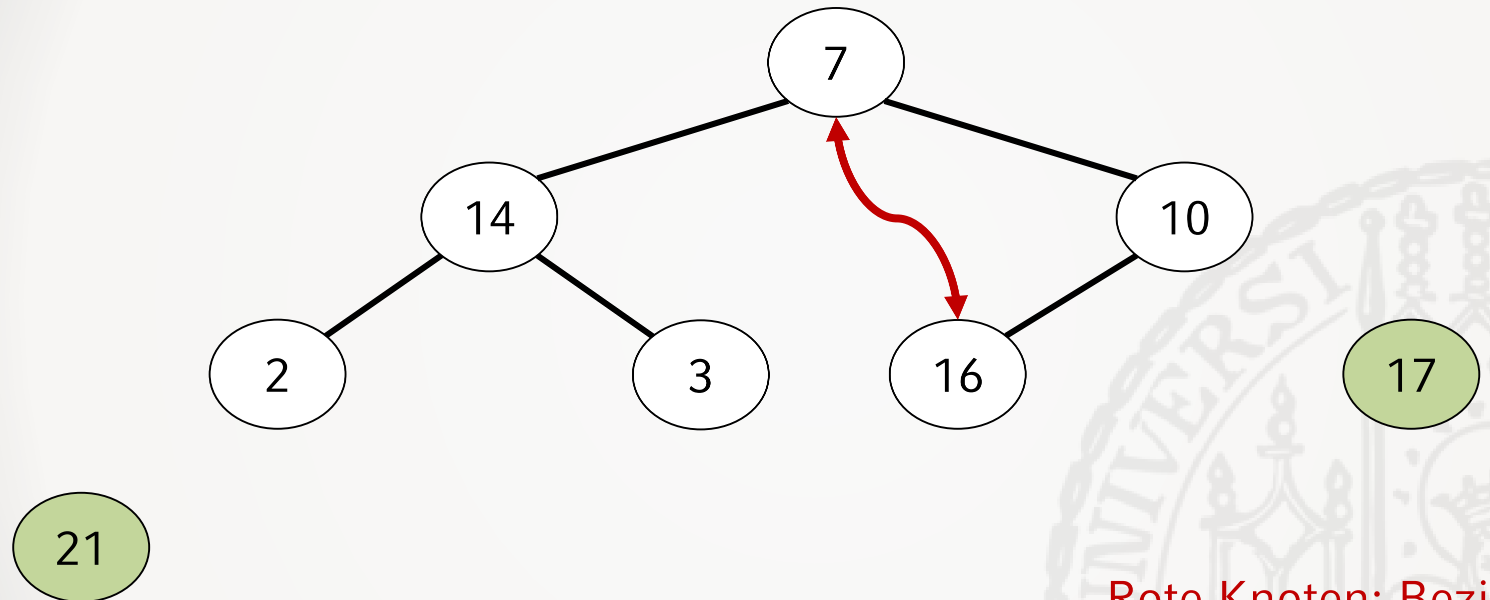
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

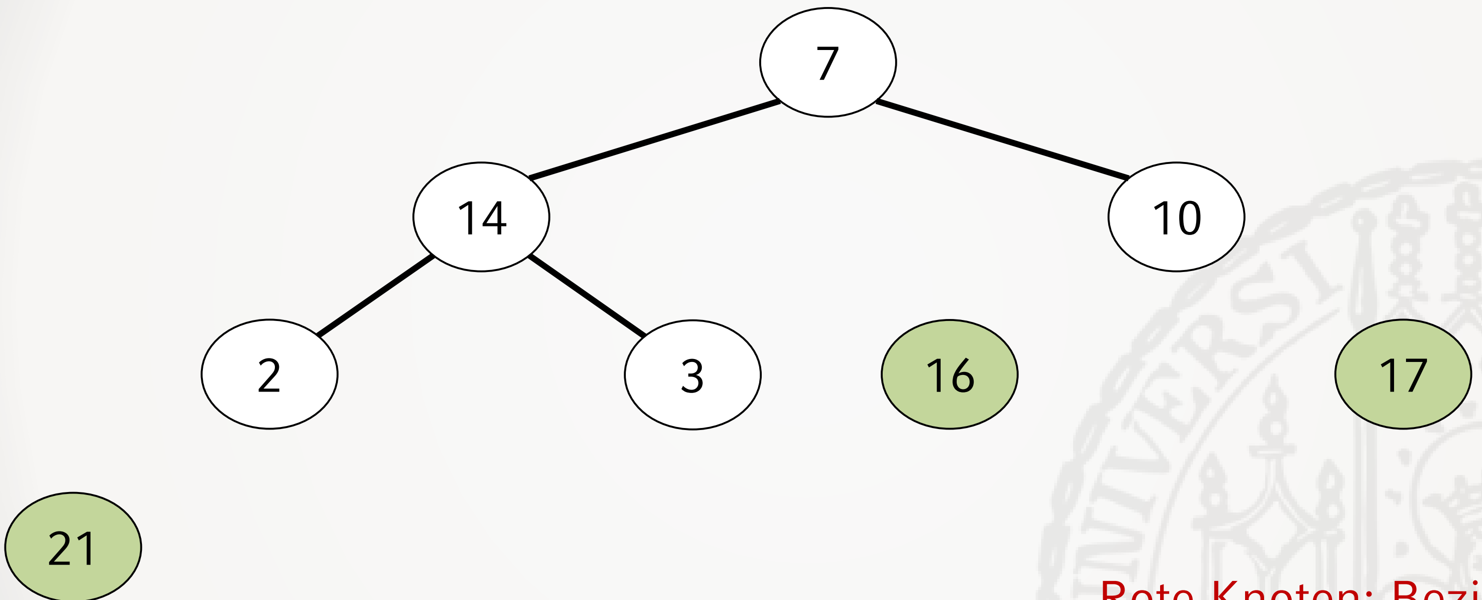
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

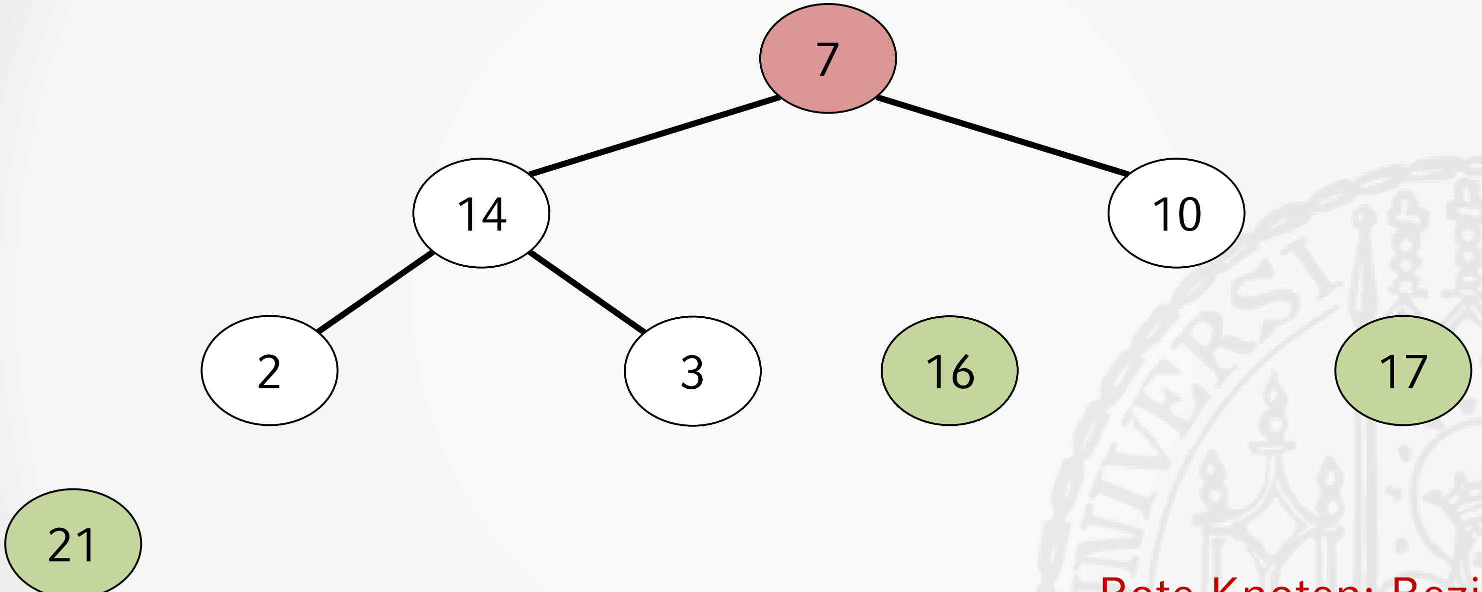
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

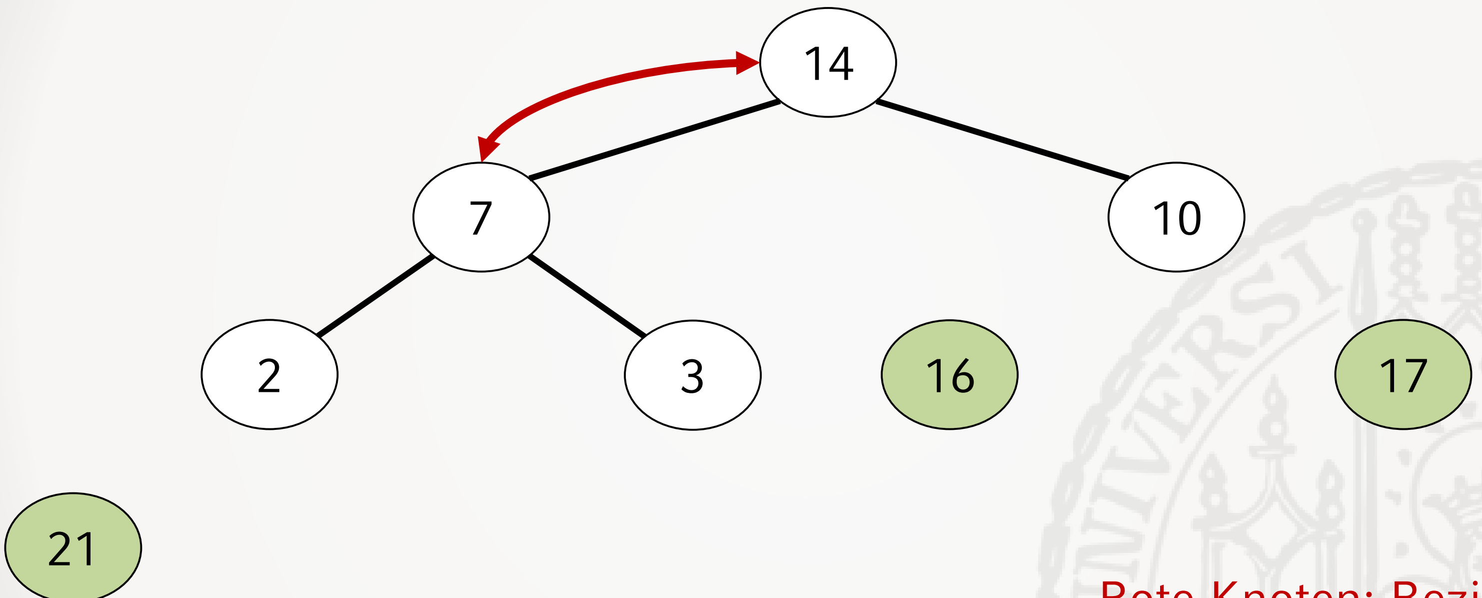
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

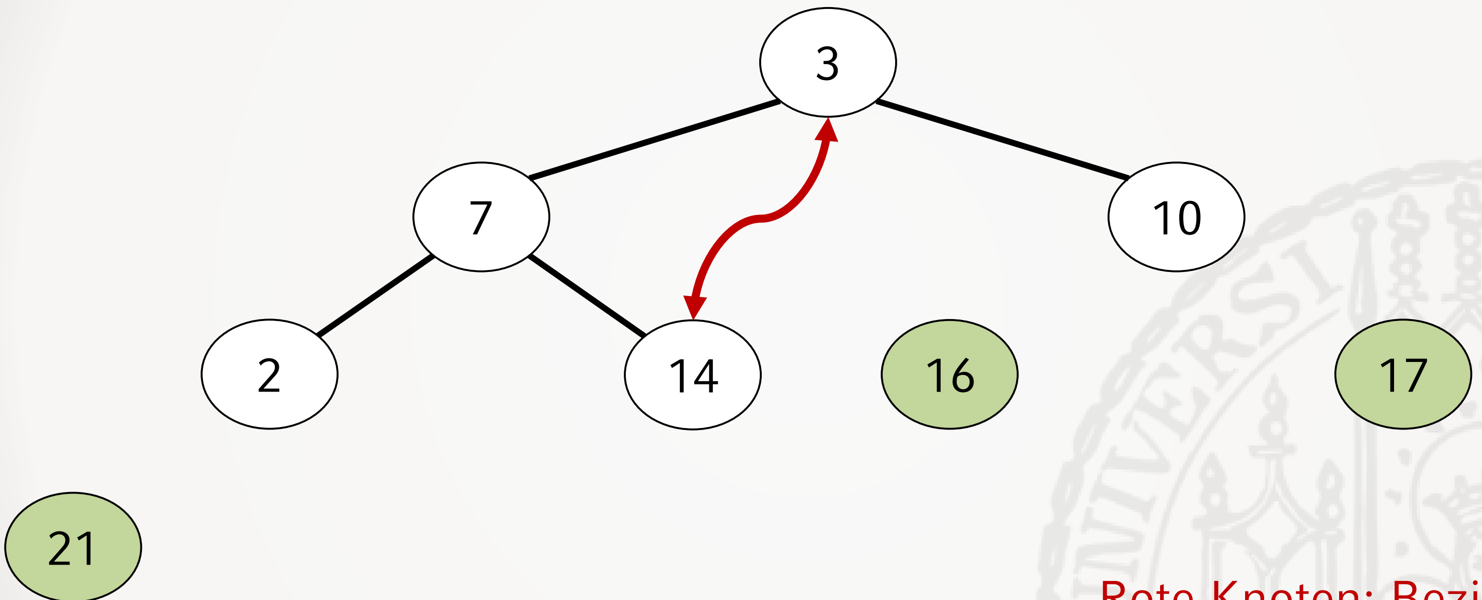
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

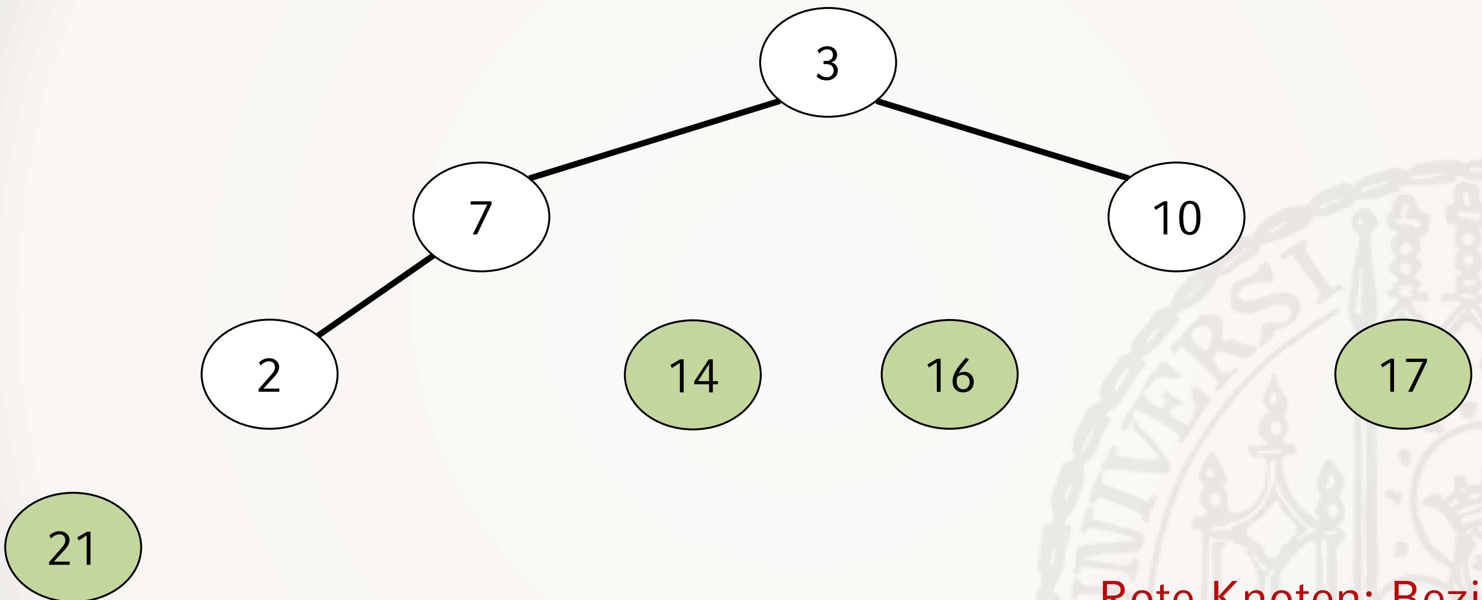
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

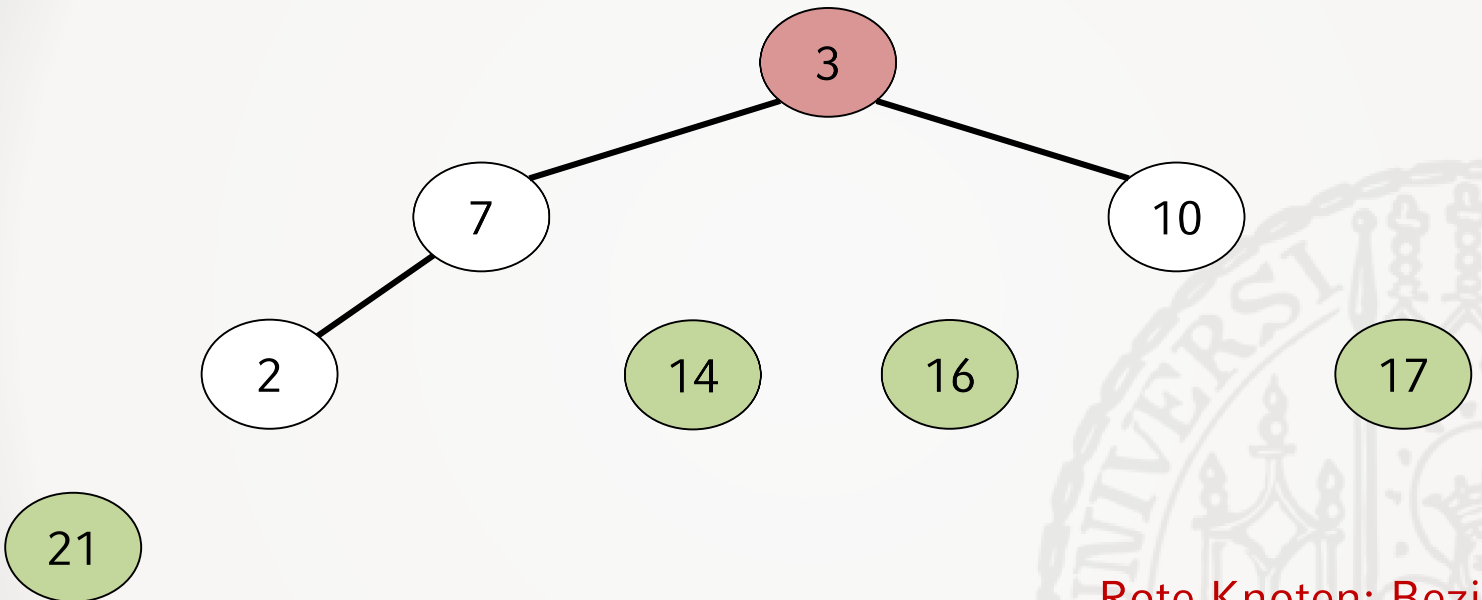
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

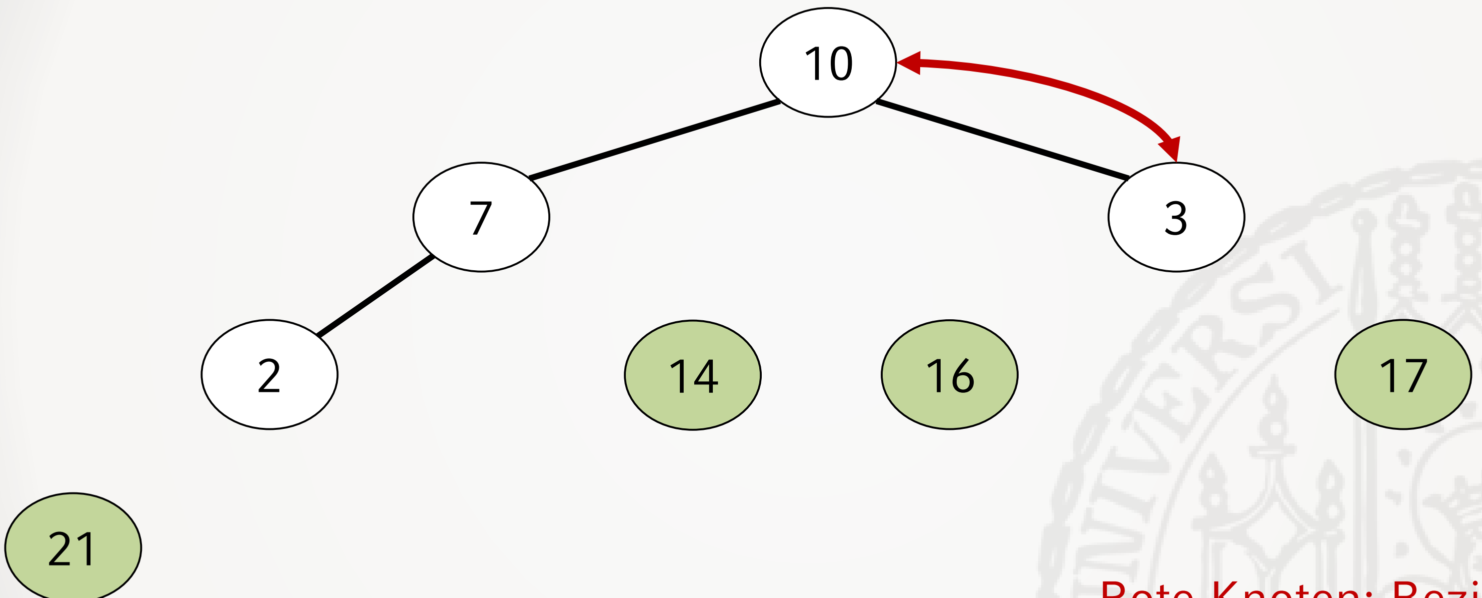
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

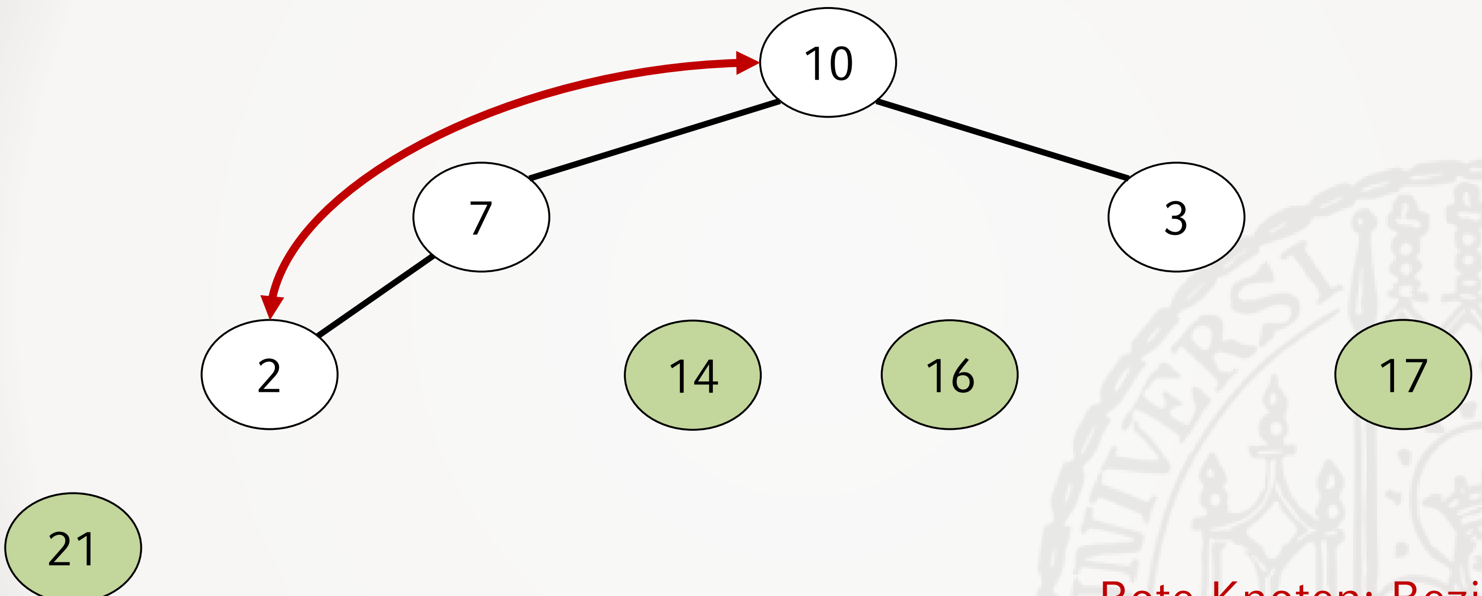
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

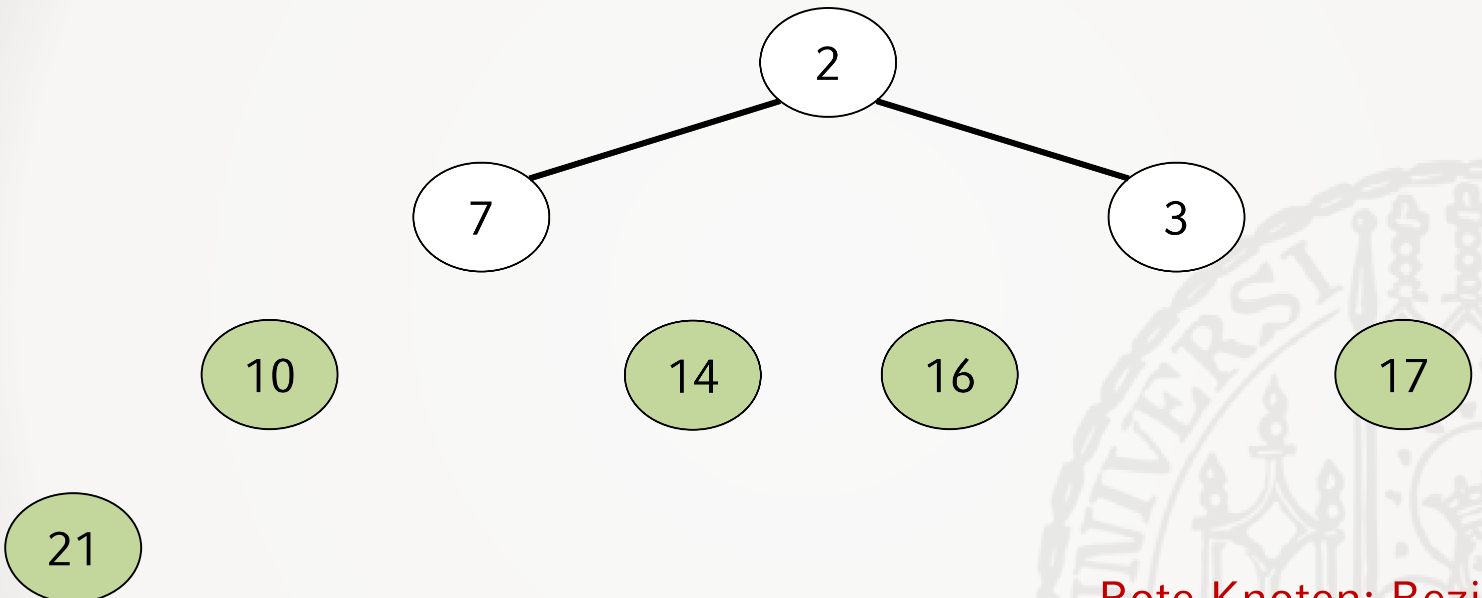
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

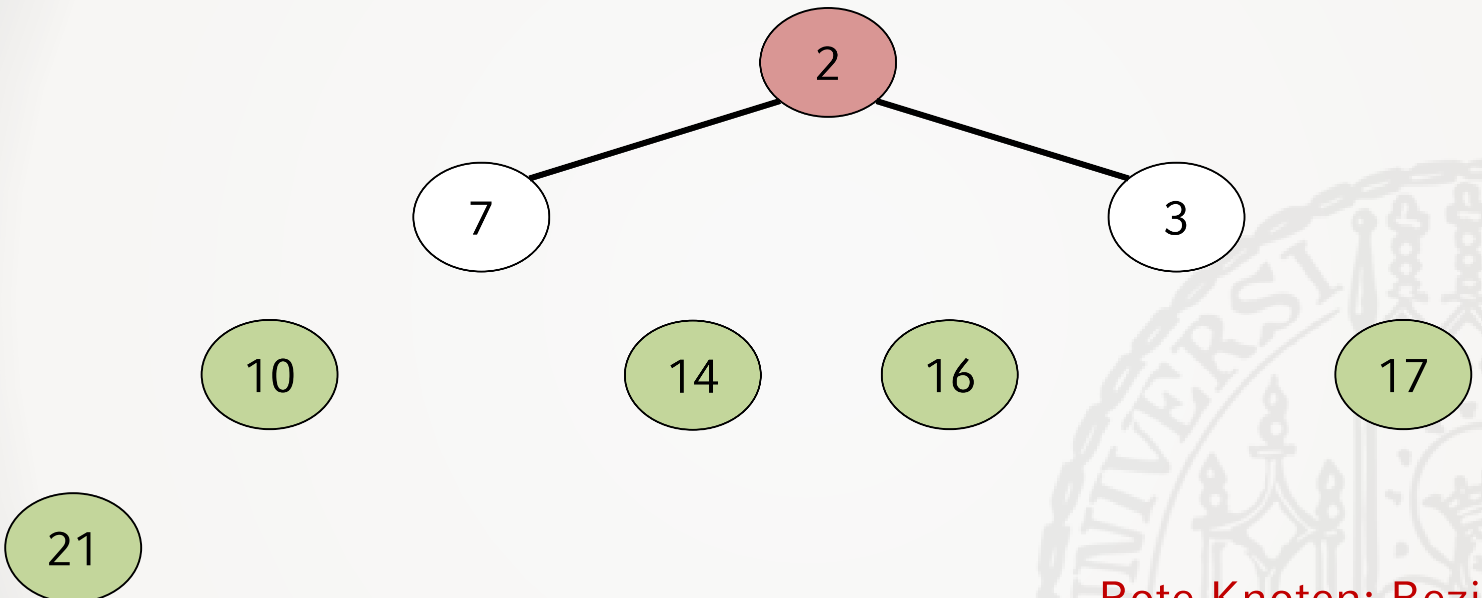
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

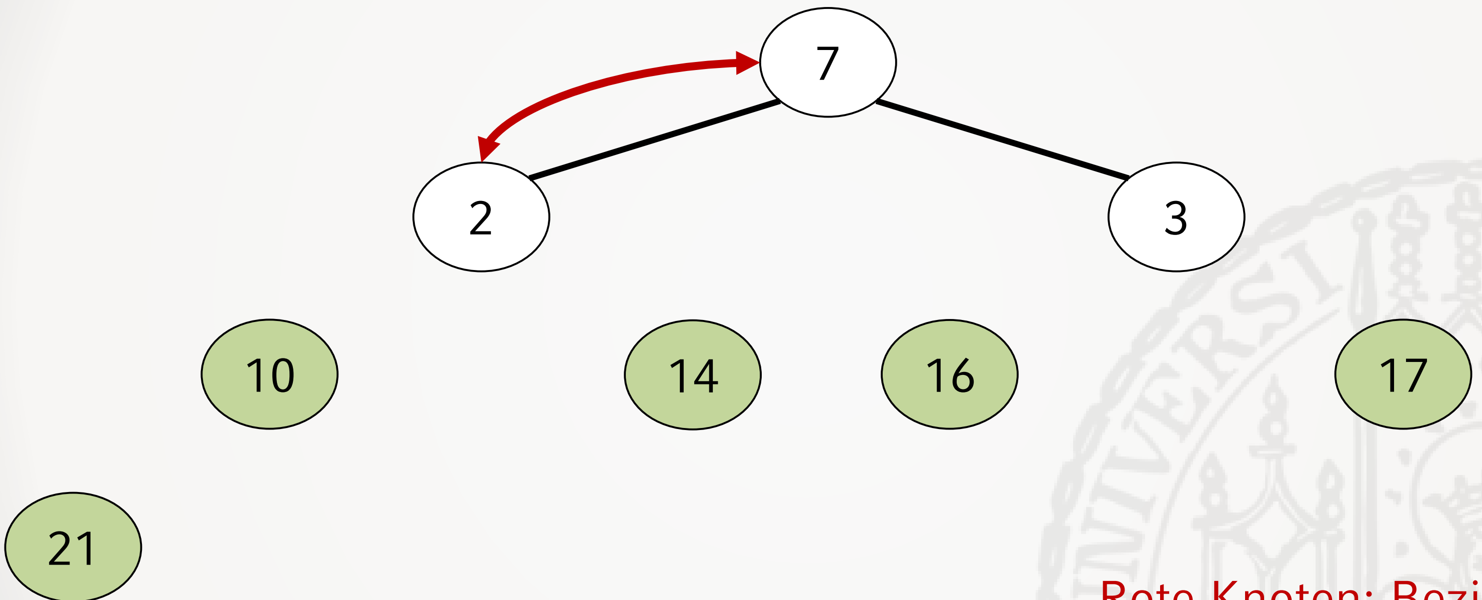
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

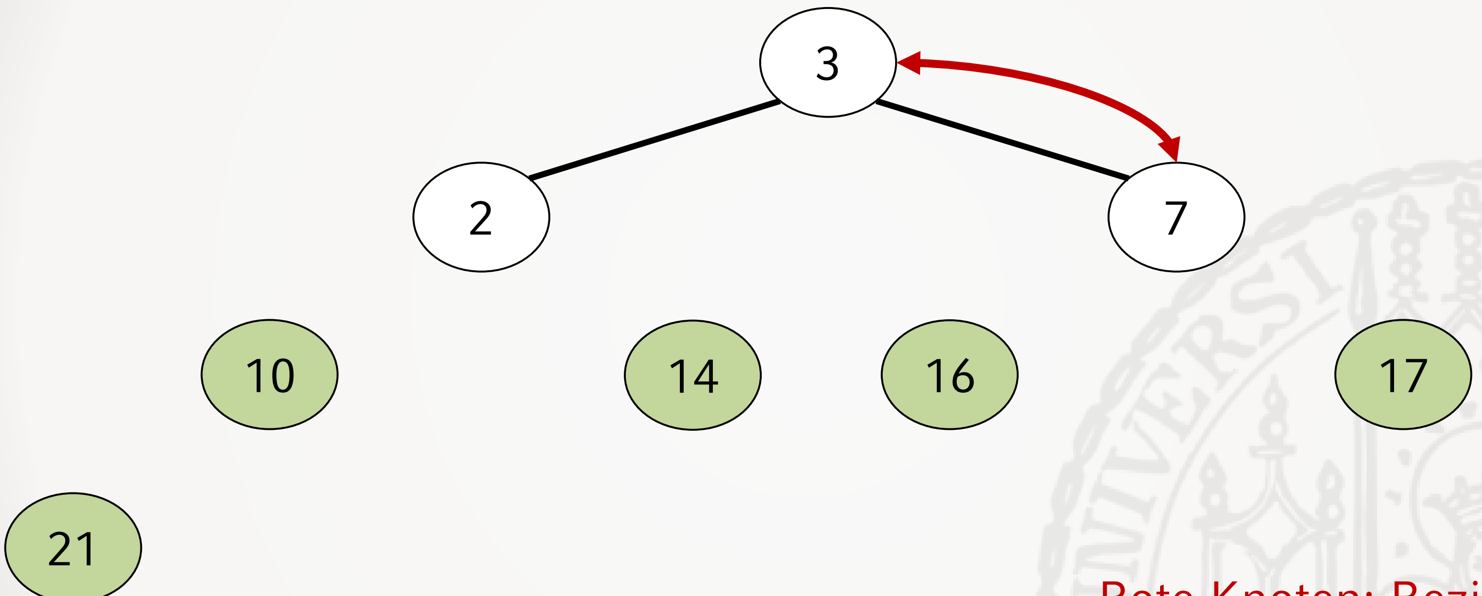
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

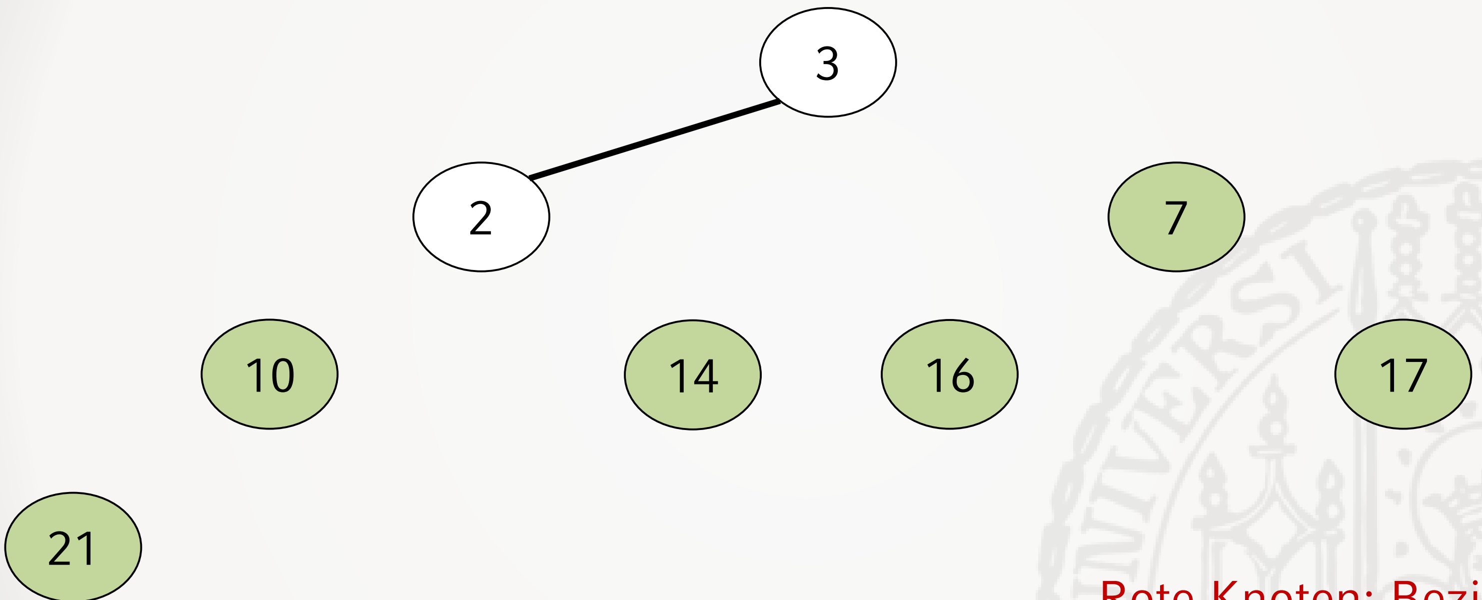
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

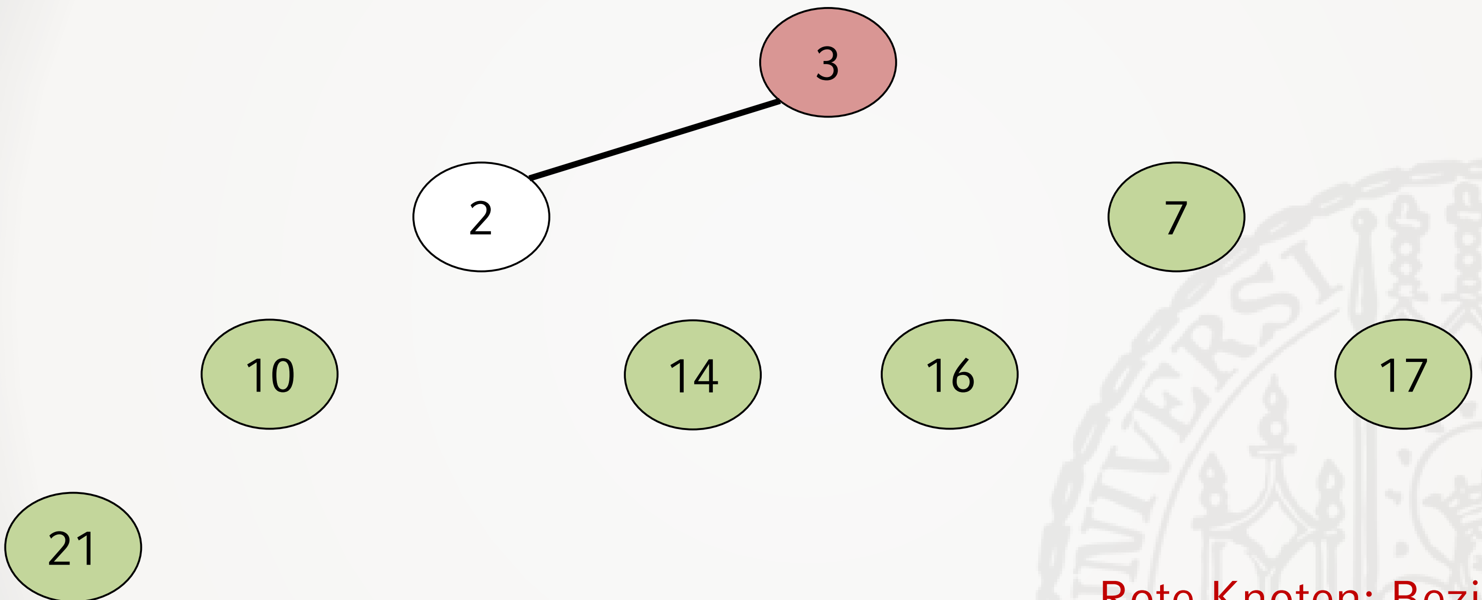
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

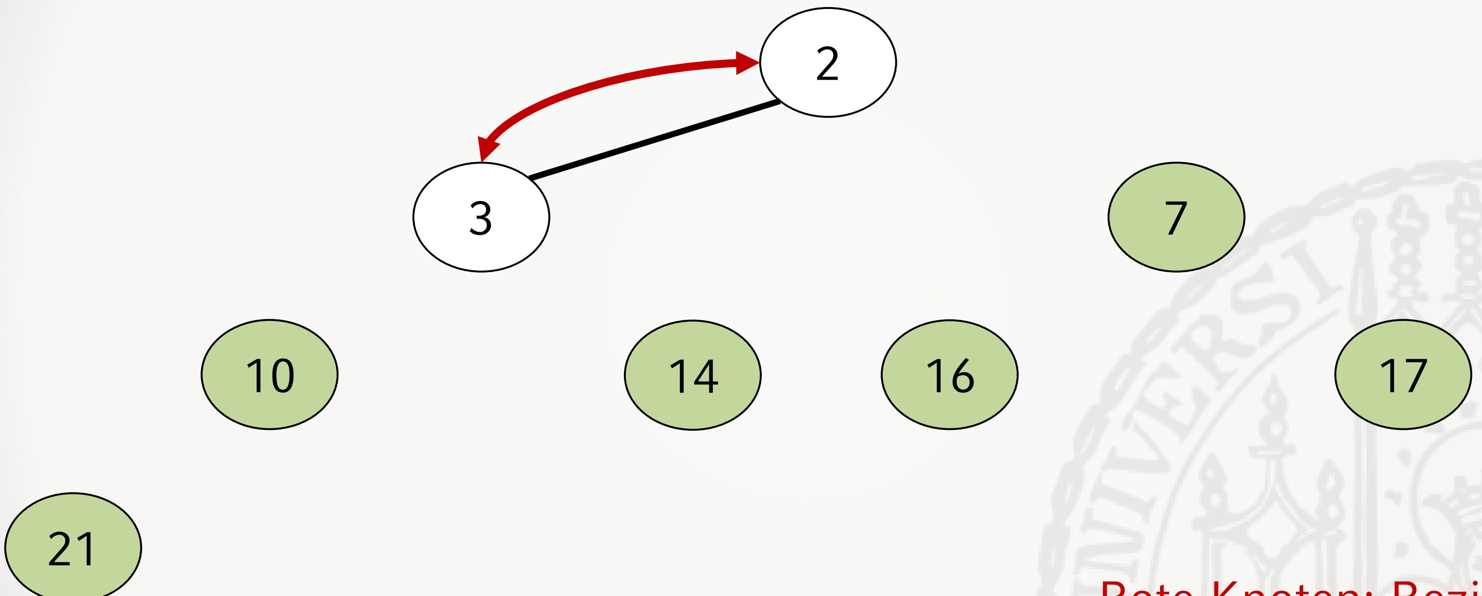
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Heap-Eigenschaft wiederherstellen



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

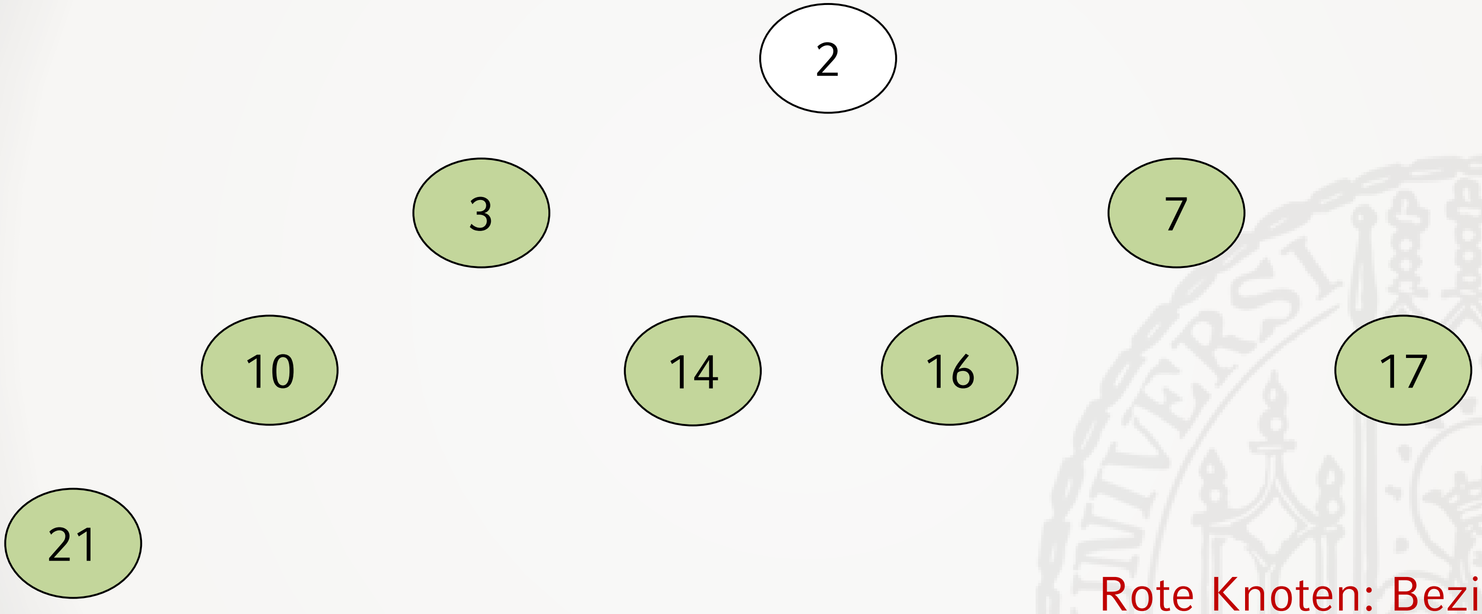
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Tausche Wurzel mit letztem Element



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

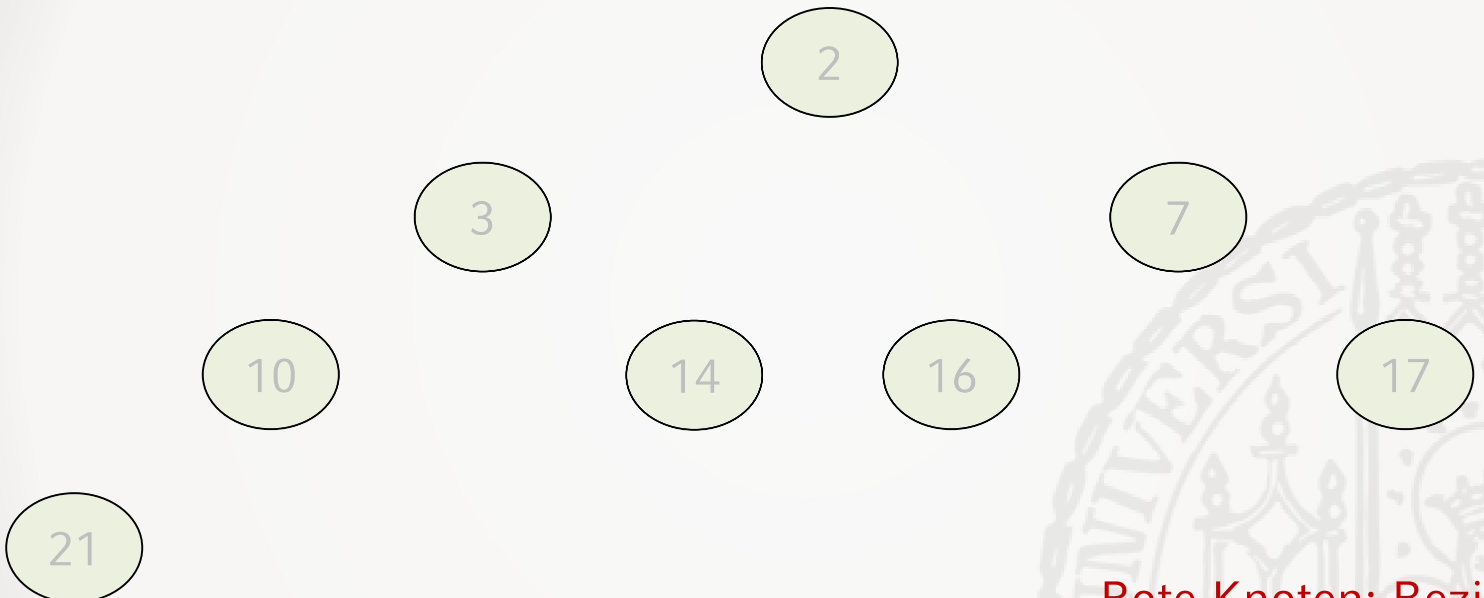
- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Entferne letzten Knoten aus der Struktur



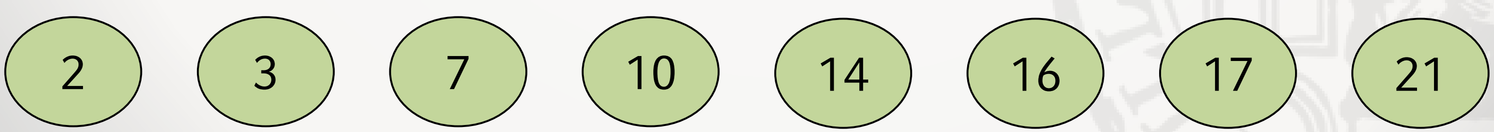
Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft

HeapSort: Beispiel

- Sortiere $10, 14, 7, 17, 3, 21, 2, 16$
- Alle Knoten abgearbeitet und Sortieren beendet.



Rote Knoten: Beziehung zu Kindknoten entspricht nicht der Heap-Eigenschaft



HeapSort: Beispiel mit Arrayeinbettung

10	14	7	17	3	21	2	16
10	14	21	17	3	7	2	16
10	17	21	16	3	7	2	14
21	17	10	16	3	7	2	14
14	17	10	16	3	7	2	21
17	16	10	14	3	7	2	21
2	16	10	14	3	7	17	21
16	14	10	2	3	7	17	21
7	14	10	2	3	16	17	21
14	7	10	2	3	16	17	21
3	7	10	2	14	16	17	21
10	7	3	2	14	16	17	21
2	7	3	10	14	16	17	21
7	2	3	10	14	16	17	21
3	2	7	10	14	16	17	21
2	3	7	10	14	16	17	21

HeapSort: Komplexität

- Überführen der Elemente in einen Binärbaum: $O(n)$
- Herstellen der Heap-Eigenschaft (Heapify)
  - Jeder Knoten könnte versickern.
  - Versickern geht maximal $\log_2 n$ tief.
  - Damit naiv und grob: $O(\log n\, n)$
- Für jeden Knoten (weil jeder mal Wurzel wird)
  - Tauschen mit dem letzten Knoten $O(1)$
  - Heapeigenschaft reparieren, Vertauschungen entlang eines Pfades im Baum: $O(\log n)$
- Insgesamt also $O(n \log n)$

HeapSort: Komplexität Heapaufbau

- Benötigt die Herstellung eines Heaps überhaupt $O(n \log n)$?
- Betrachten wir einen vollständigen Binärbaum mit Höhe h und damit $n = 2^{h+1} - 1$ Knoten.
- Wir bestimmen den Aufwand jedes Knotens von den Blättern an bis zur Wurzel.
- Ein Blatt kann nicht versickern und erfüllt immer die Heapeigenschaft $\Rightarrow$ 0 Vertauschungen für 2^h Knoten.
- Auf der nächsten Ebene gibt es 2^{h-1} Knoten, die maximal einmal versickern können.
- Die Wurzel kann schließlich h-mal versickern.

$$T(n) = \sum_{i=0}^{h} \left(i \cdot 2^{h-i}\right) = 2^h \sum_{i=0}^{h} \frac{i}{2^i}$$

HeapSort: Komplexität Heapaufbau (2)

- Kleiner Einschub: Geometrische Reihe ($|x| < 1$)

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

- Beide Seiten ableiten und multiplizieren mit x:

$$\sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

- Für $x = \frac{1}{2}$ erhalten wir:

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = \sum_{i=0}^{\infty} i \left(\frac{1}{2}\right)^i = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$$

HeapSort: Komplexität Heapaufbau (3)

- Nun können wir unsere Formel

$$T(n) = \sum_{i=0}^{h} \left(i \cdot 2^{h-i}\right) = 2^h \sum_{i=0}^{h} \frac{i}{2^i}$$

abschätzen, indem wir noch ein paar (unendlich viele) Summanden hinzufügen:

$$2^h \sum_{i=0}^{h} \frac{i}{2^i} \leq 2^h \sum_{i=0}^{\infty} \frac{i}{2^i} \leq 2^h 2$$

Damit gilt mit $n = 2^{h+1} - 1$:

$$T(n) \leq 2^{h+1} = n + 1 \in O(n)$$

Schranken des Sortierproblems

- Bisher: Sortieralgorithmen haben Komplexität
- Können wir einen Sortieralgorithmus finden, der besser ist als die gezeigten?
- Jetzt: Frage nach der Komplexität des Problems

- Wie groß ist die Anzahl der Schlüsselvergleiche $T_{min}(n)$, um eine n-elementige Folge von Schlüsselelementen mit einem beliebigen (effizientesten) Algorithmus zu sortieren?

- Existiert $T_{min}(n)$, sodass $T_{min}(n) \leq T_A(n)$ für alle Algorithmen A? D.h. jeder denkbare Algorithmus braucht in diesem Fall mindestens $T_{min}(n)$ viele Vergleiche.

Obere Schranke

- Betrachten wir beispielsweise MergeSort, so wissen wir bereits:

$$T_{min}(n) \leq n\lceil\log_2 n\rceil - n + 1 \in O(n \log n)$$

Untere asymptotische Schranke

- Wir benötigen zusätzlich ein Komplexitätsmaß für eine untere asymptotische Schranke.
- Erinnerung:
  $O(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$

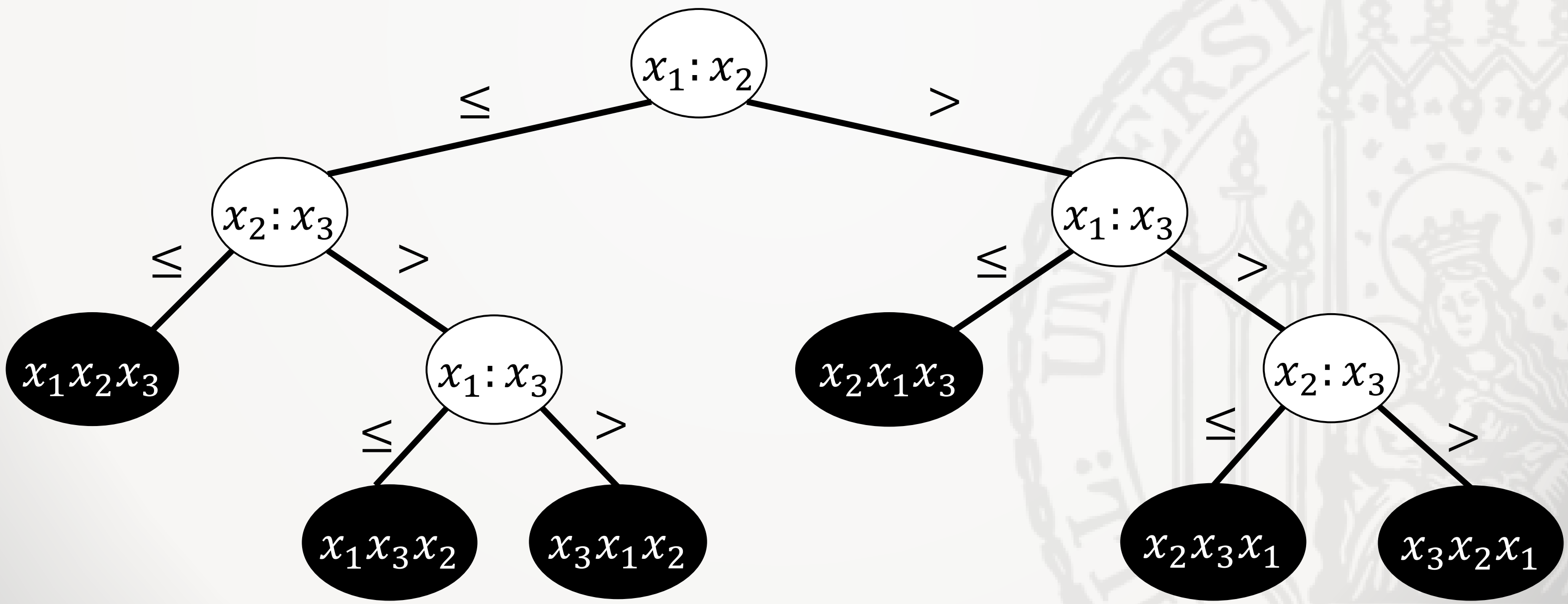
- Analog zu $O(f)$ definieren wir:
  $\Omega(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0: |g(n)| \geq c \cdot |f(n)|\}$

Permutationen

- Gegeben sei eine n-elementige Folge
$$x_1, x_2, \dots, x_n$$
- Sei $X = \{x_1, x_2, \dots, x_n\}$. Eine bijektive Abbildung $\pi: X \to X$ ist eine Permutation.
- Zum Beispiel ist 3,6,2 eine Permutation von 6,2,3 mit $\pi(3) = 6, \pi(6) = 2$ und $\pi(2) = 3$.
- Es gibt $n!$ viele Permutationen auf X.
- Sortieren ist die Auswahl einer Permutation der Folge mit
$$\pi(x_1) \leq \pi(x_2) \leq \cdots \leq \pi(x_n)$$

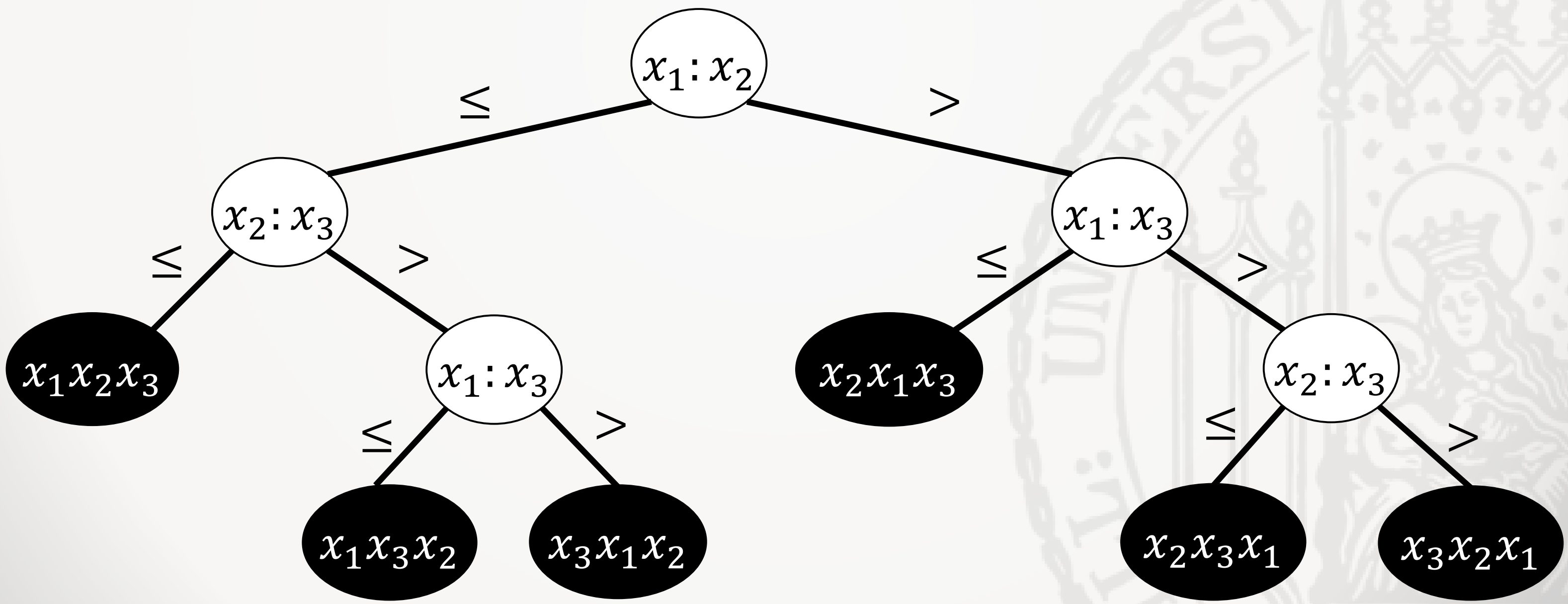
Entscheidungsbaum

- Der Ablauf eines nur auf Vergleichen basierenden Sortieralgorithmus kann durch einen Entscheidungsbaum dargestellt werden.
- Der folgende binäre Entscheidungsbaum „sortiert" eine 3-elementige Folge $x_1 x_2 x_3$.
  Da $3! = 6$, muss der Entscheidungsbaum 6 Blätter besitzen.



Entscheidungsreihenfolge von Sortieralgorithmen

- Das Entscheidungsbaummodell stellt unabhängig vom Algorithmus alle möglichen Ausgänge dar.
- Sortieralgorithmen müssen auf dieser Abstraktion das Ergebnis (= das Blatt mit der gesuchten Permutation) erreichen.
- Sie unterscheiden sich durch die Suchstrategie und damit dem Pfad von untersuchten Knoten in diesem Baum bis zu einem Blatt.



Untere Schranke für den Worst-Case

- Die Anzahl der Vergleiche im Worst-Case eines Algorithmus entspricht der Länge des längsten Weges im Entscheidungsbaum.
- Dies wiederum ist die Höhe des Entscheidungsbaums.
- Der Entscheidungsbaum habe Höhe h und b Blätter.
- Wir wissen:
  - Jede Permutation ist als Blatt erreichbar $\Rightarrow n! \leq b$
  - Ein Baum der Höhe h hat maximal 2^h Blätter $\Rightarrow b \leq 2^h$

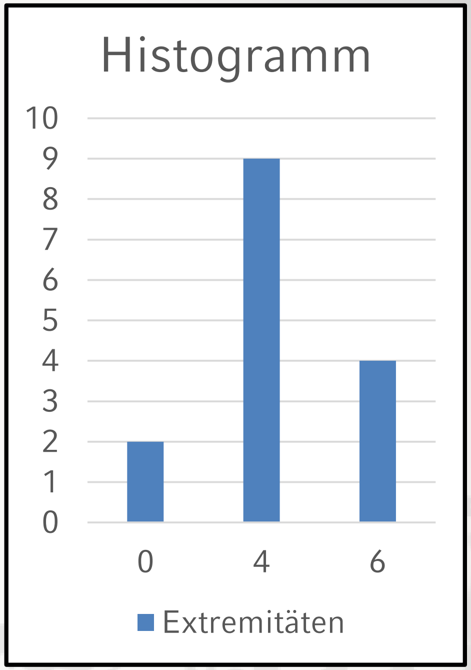
$$2^h \geq b \geq n! \Rightarrow h \geq \log_2 n! \geq \log_2 \left(\frac{n}{e}\right)^n$$
$$= n \log_2 n - n \log_2 e \in \Omega(n \log n)$$

- Damit folgt der zentrale Satz:
  Jedes vergleichsbasierte Sortierverfahren benötigt $\Omega(n \log n)$ Vergleiche im Worst-Case.

CountingSort

- Angenommen, die Schlüssel sind darstellbar als ganzzahlige Werte im Bereich $0, \dots, m-1$.
- Beispiel: Sortieren aller eingeschriebener Studenten anhand Studiensemesters $1 \dots 20$.
- Wir können ein Histogramm erstellen und zählen für jeden Schlüsselwert seine Häufigkeit.
- Diese Häufigkeit liefert die Position für jeden Record.
- Bewege nun die Records an ihre errechnete Position.

CountingSort: Beispiel



- Beispiel: Sortiere Tiere anhand der Anzahl ihrer Extremitäten.
  Ameise, **B**är, **D**achs, **E**lefant, **F**liege, **G**iraffe, **H**ase, **K**arpfen, **L**öwe, **M**istkäfer, **S**chlange, **T**iger, **U**hu, **W**espe, **Z**ebra
- Mögliche Kategorien: 0, 4, 6
- Damit liegen die Bereiche im Zielarray fest:

0	4	6

- Ein weiterer Durchlauf reicht, um die einzelnen Schlüssel an die passende Position zu schreiben:

K	**S**	**B**	**D**	**E**	**G**	**H**	**L**	**T**	**U**	**Z**	**A**	**F**	**M**	**W**

CountingSort: Komplexität

- Es werden n Schlüssel sortiert mit m verschiedenen Schlüsselwerten
- Initialisierung des Histogramms → $O(m)$
- Zählen der Schlüssel, Schleife über alle Schlüssel → $O(n)$
- Bestimmung der Adressen für alle m Werte → $O(m)$
- Zusammenstellung des finalen sortierten Arrays $O(n)$

- Insgesamt: $O(m+n) = O(\max(m, n))$

- Für kleine m ist CountingSort besser als vorherige Verfahren?
  - Untere Schranke $O(n \log n)$ beruht auf der Anzahl der Vergleiche.
  - CountingSort ist nicht vergleichsbasiert, sondern adressbasiert.

Modernes effizientes Sortieren

- Java: Arrays.sort() nutzt „Dual-Pivot Quicksort“
  - Wähle zwei Pivots und splitte in drei Teile
- Vergleichsbasierte Methoden oft als Hybridverfahren
  - IntroSort sortiert wie QuickSort, außer eine Bewertung im Teilschritt prognostiziert den Worst-Case, dann wird HeapSort genutzt
- Python: Timsort
  - Hybrid aus MergeSort und InsertionSort
  - Findet vorsortierte Teilstücke und sortiert bloß dazwischen
- Oft anwendungsspezifisch:
  - Bei großer Wahrscheinlichkeit von Vorsortierung ist Best-Case wichtiger als Worst-Case
- Paralleles Sortieren ebenfalls möglich. Keine Änderung der Komplexität, aber Beschleunigung durch zeitgleiches Rechnen.

Fazit: Sortieren

- Einfache Sortieralgorithmen (BubbleSort, SelectionSort, InsertionSort):
  - Einfache Implementierung
  - Im schlimmsten Fall unnötig viele Vergleiche $\Rightarrow O(n^2)$
- Bessere Suchverfahren (QuickSort, MergeSort, HeapSort):
  - Komplexere Implementierung
  - Nutzen Datenstrukturen zur Verwaltung von Wissen über vorherige Vergleiche
  - Daher kann Sortieren in $O(n \log n)$ durchgeführt werden.
- $O(n \log n)$ viele Vergleiche werden mindestens benötigt bei vergleichsbasierten Verfahren.