Universität Bremen

Praktische Informatik 2
Sortieren (Forts.)

Thomas Röfer

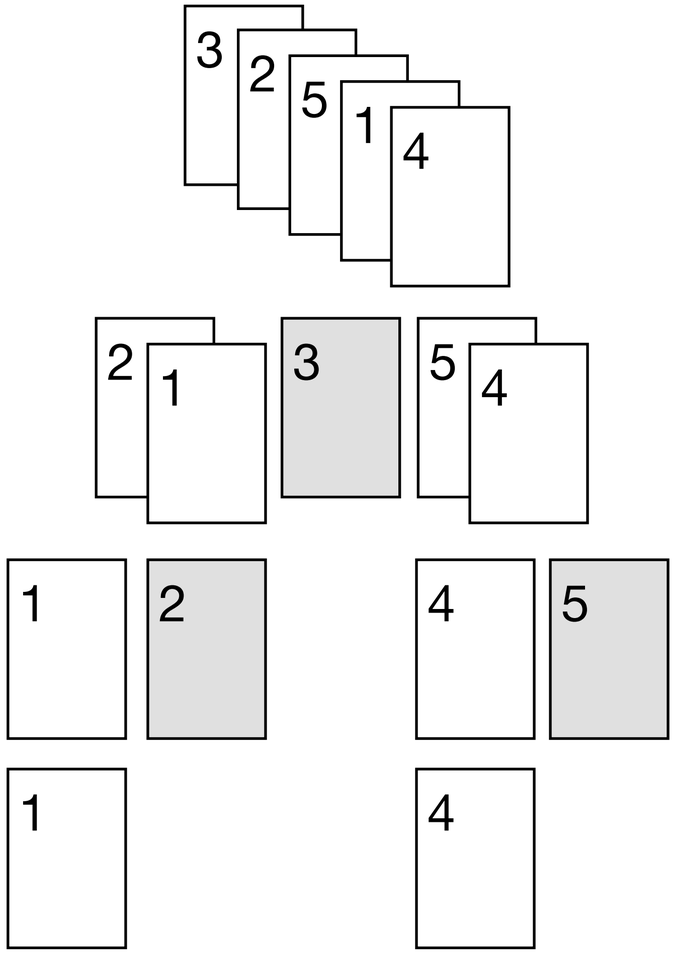
Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

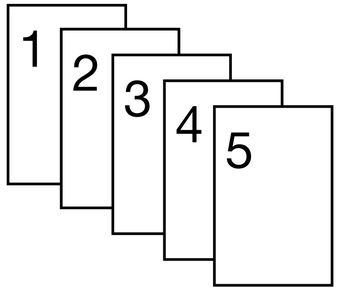
Bild: ©Michael Ihle

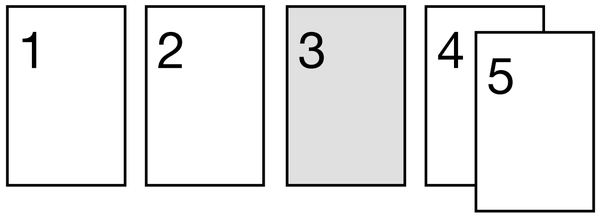
Universität Bremen

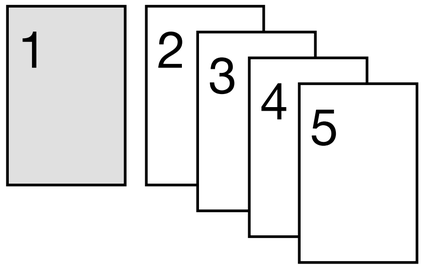
Quicksort

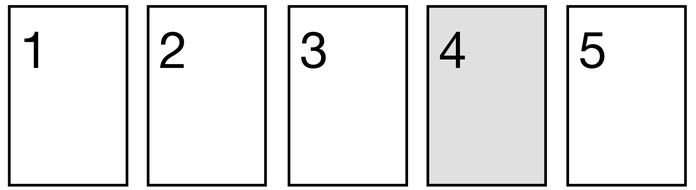


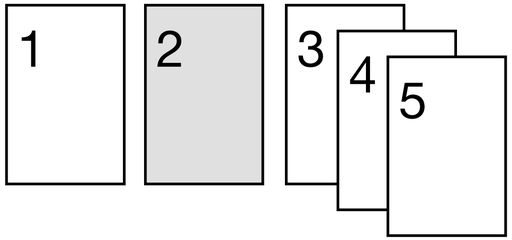
Quicksort: Ungünstige Vorsortierung

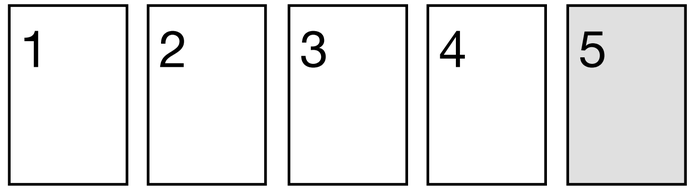












Universität Bremen

Quicksort

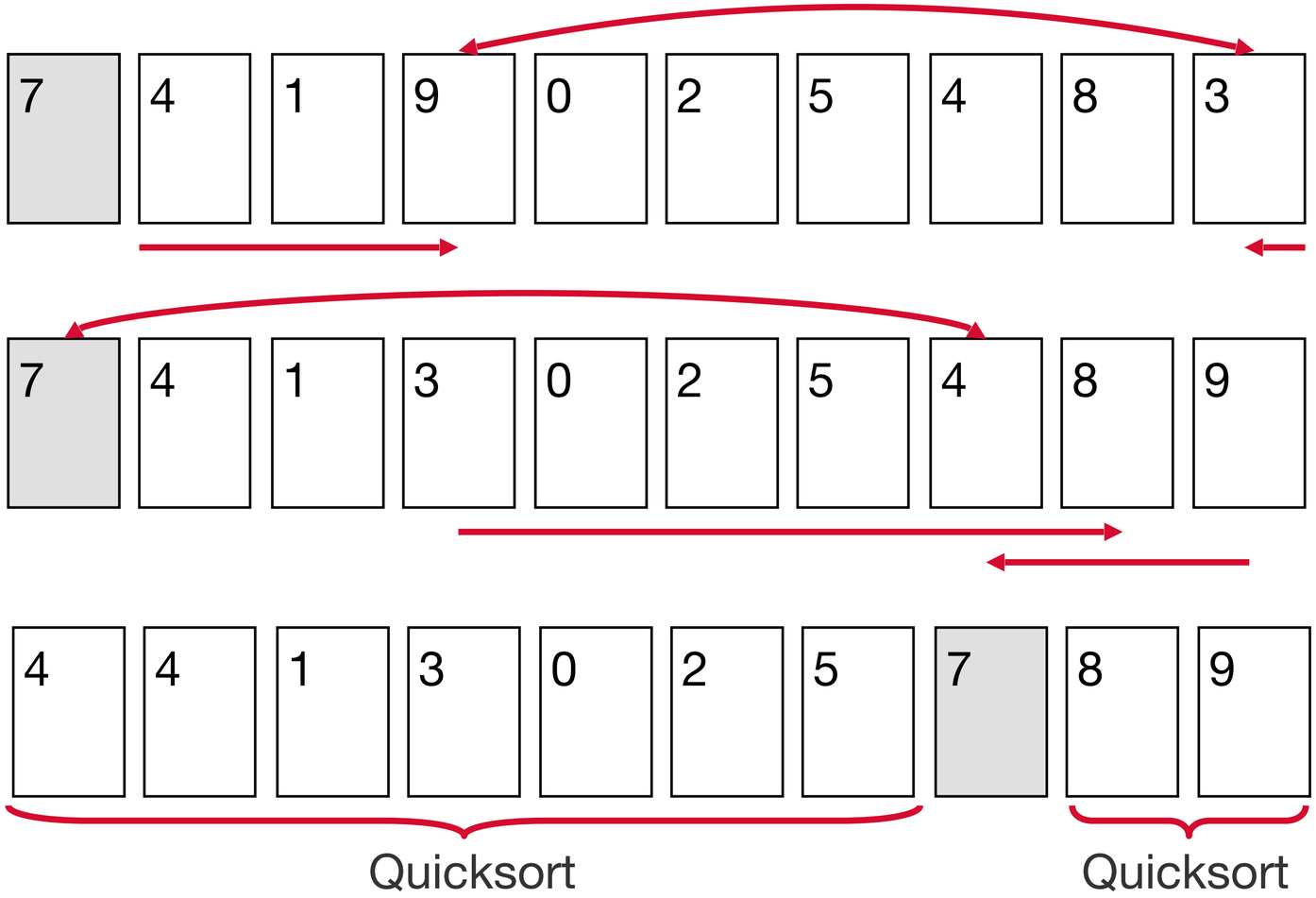
- Enthält die Folge kein oder nur ein Element, ist sie sortiert
- Ansonsten entnimm ein Element (das sog. **Pivot**-Element)
- Bilde neue Liste aus kleineren Elementen und lasse diese sortieren
- Bilde neue Liste aus größeren Elementen und lasse diese sortieren
- Hänge die sortierte Liste der kleineren Elemente, das Pivot-Element und die sortierte Liste der größeren Elemente aneinander und liefere diese zurück

Universität Bremen

Quicksort: Aufwand

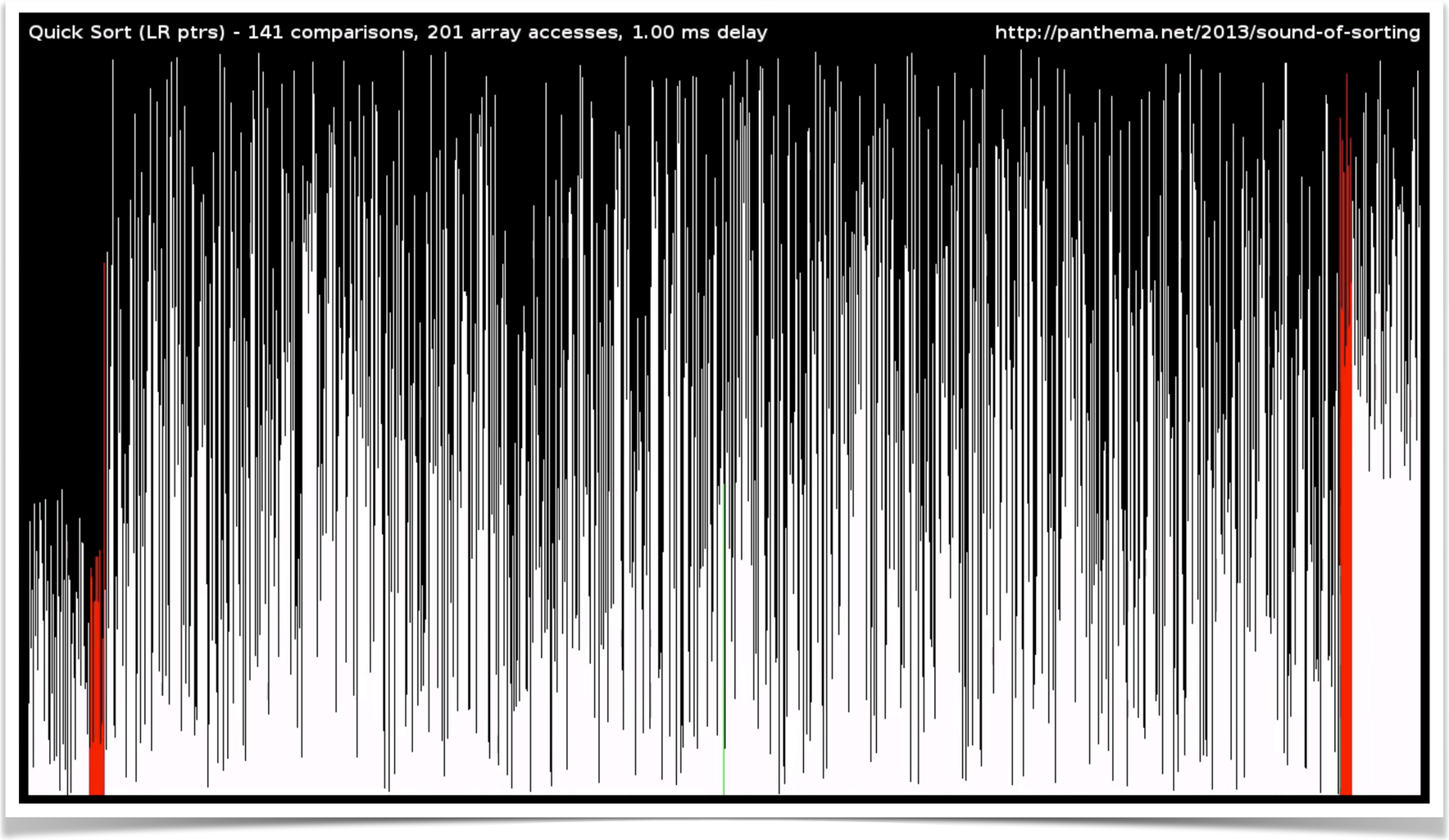
- Günstigster Fall: **O(N log N)** (Beide Teillisten sind jeweils gleich lang)
- Schlechtester Fall: **$O(N^2)$** (Eine der Teillisten ist immer leer)
  - Ist Pivot-Element immer das erste, ist dies bei Vorsortierung der Fall!
- Durchschnittlicher Fall: **O(N log N)** (Beweis siehe Ottmann / Widmayer)
- Stabilität
  - Listen: Sonderbehandlung für Elemente, die gleich dem Pivot-Element sind
  - Arrays: nicht stabil
  - Kein zusätzlicher Speicherbedarf bei Sortierung eines Arrays

Quicksort: Beispiel



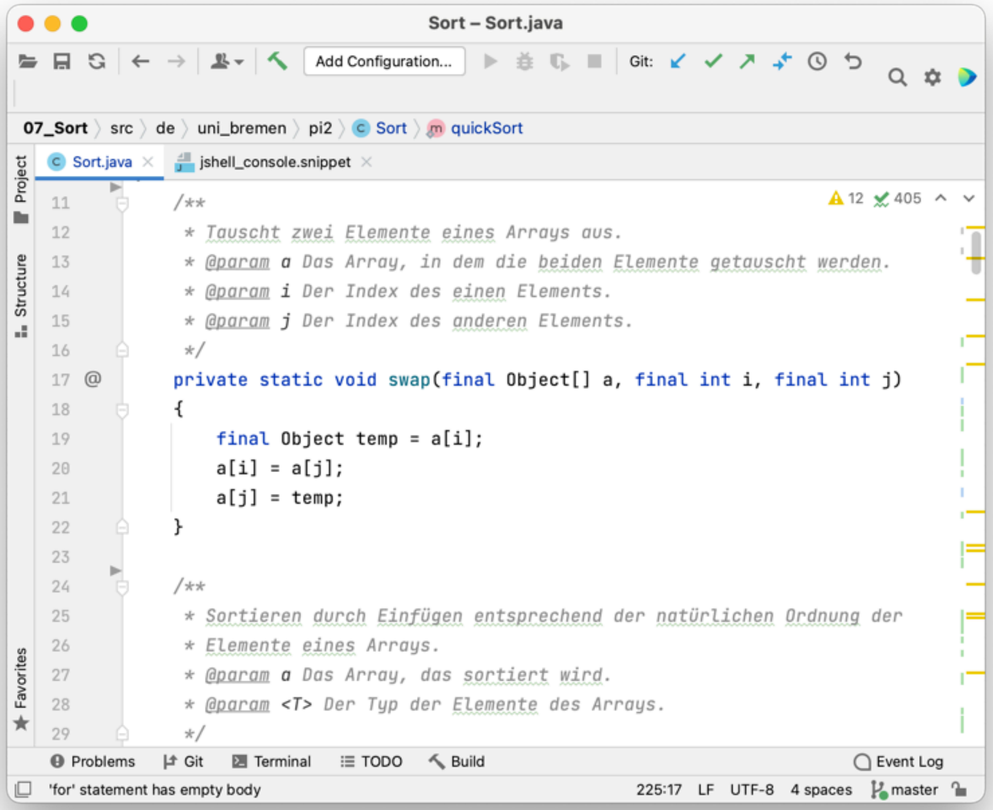
Universität Bremen

Quicksort: Demo



Universität Bremen

Quicksort: Demo

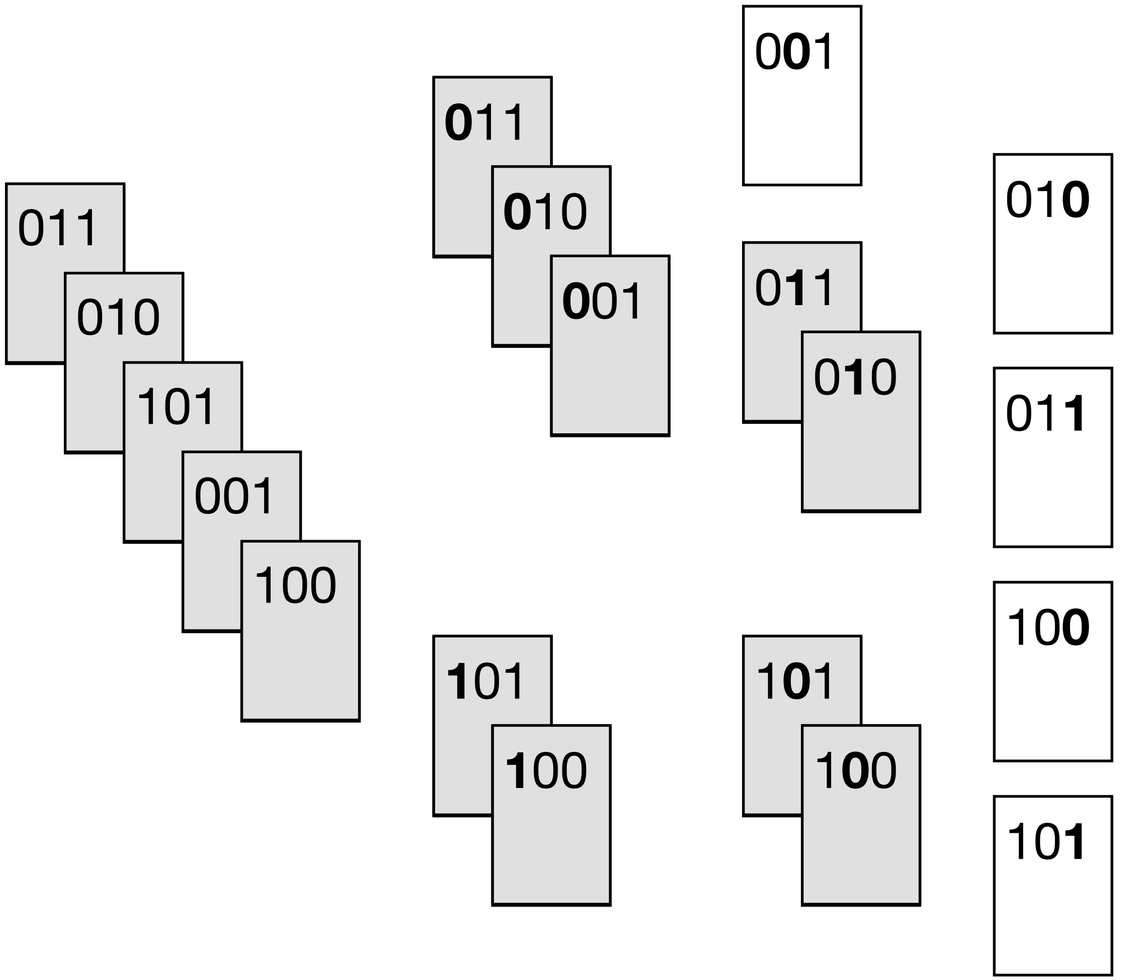


Universität Bremen

Quicksort: Auswahlstrategien für Pivot-Element

- Beliebige Elemente könnten an Position des Pivot-Elements getauscht werden: **swap(a, bottom, pos)**
- Zufallsstrategie: **pivot = a[bottom + (top - bottom) × random(0…1)]**
  - Schließt Standardfälle aus (z.B. Vorsortierung)
  - Wahrscheinlichkeit des schlechtesten Falls bleibt aber gleich
  - Keine reproduzierbare Laufzeit
- 3-Median-Strategie: **pivot = median(a[bottom], a[bottom + (top - bottom) / 2], a[top – 1])**
  - Senkt die Wahrscheinlichkeit des schlechtesten Falls
  - Kann mit Zufallsstrategie kombiniert werden

Binärer Radixsort



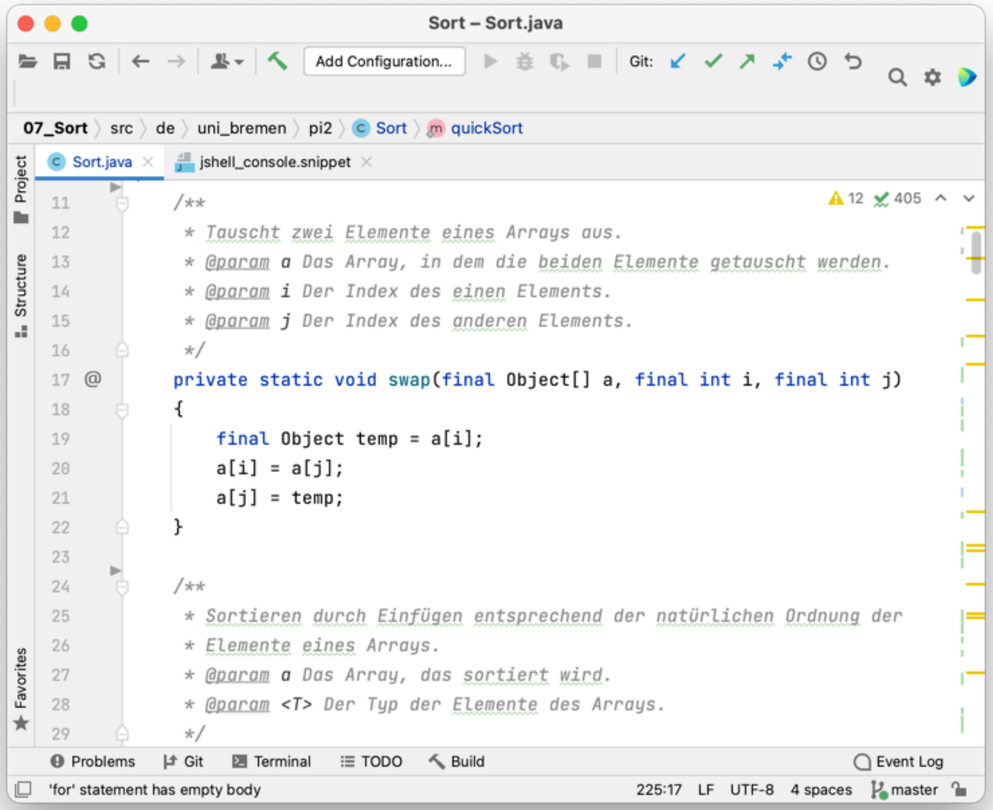
Universität Bremen

(Binärer) Radixsort

- Betrachtet Sortierschlüssel als Folge von Stellen (z.B. Ziffern) und sortiert sie Stelle für Stelle
- Radix: Anzahl der verschiedenen Werte, die jede Stelle annehmen kann
- Es gibt Radix-viele Teilfolgen, in die jeweils eingetragen wird (anstatt z.B. in 2 bei Quicksort für **< Pivot** und **≥ Pivot**)
- Sortieren am Ort ist möglich und besonders einfach mit Radix 2, d.h. die Schlüssel werden bit-weise durchlaufen (Binärer Radix Sort bzw. auch Binärer Quicksort)
  - Dann kann analog zu Quicksort sortiert werden
- Aufwand ist **$O(N \cdot b)$**, wobei **b** die Anzahl der Stellen ist

Universität Bremen

Binärer Radixsort: Demo



Heap

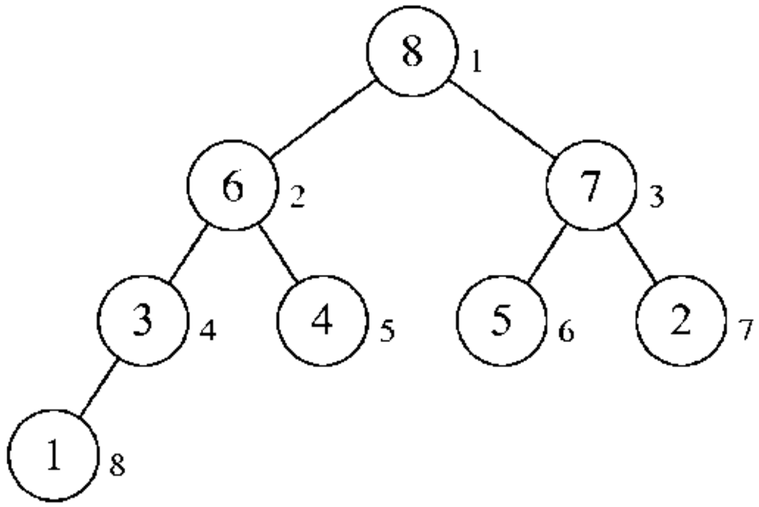
- Eine Folge **$F = k_1, k_2, \ldots, k_N$** von Schlüsseln nennen wir einen (Max-)Heap (Halde), wenn **$k_i \leq k_{\lfloor i/2 \rfloor}$** für **$2 \leq i \leq N$** gilt (Min-Heap: **$\leq \rightarrow \geq$**)

- Anders ausgedrückt: **$k_i \geq k_{2i}$** und **$k_i \geq k_{2i+1}$**, sofern **$2i \leq N$** bzw. **$2i+1 \leq N$**

- Beispiel

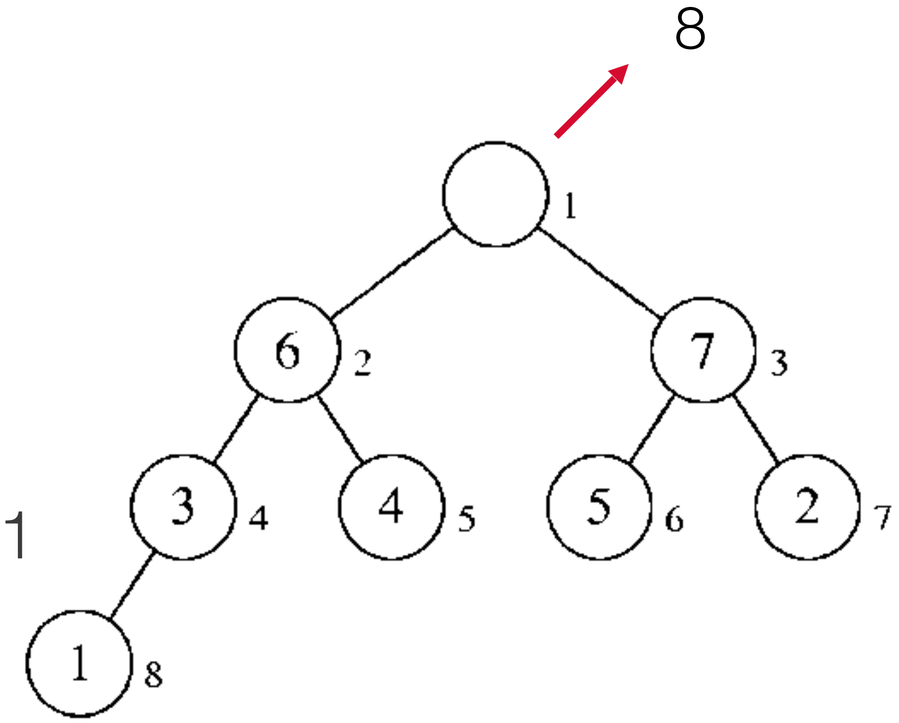
  - **$F = 8, 6, 7, 3, 4, 5, 2, 1$** genügt der Heap-Bedingung, weil **$8 \geq 6, 8 \geq 7, 6 \geq 3, 6 \geq 4, 7 \geq 5, 7 \geq 2, 3 \geq 1$**

- Hat **nichts** mit dem Java-Heap zu tun!

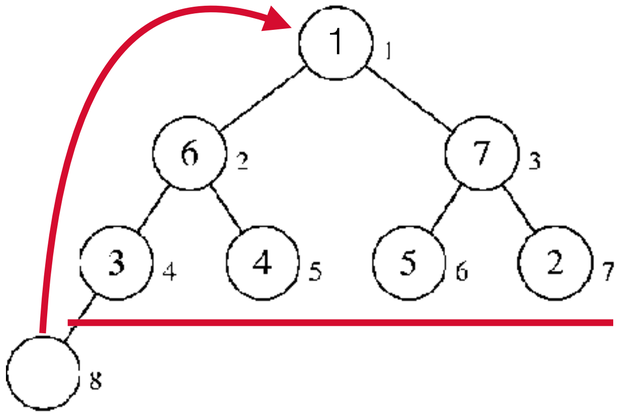


Heap: Ausgabe in absteigender Reihenfolge

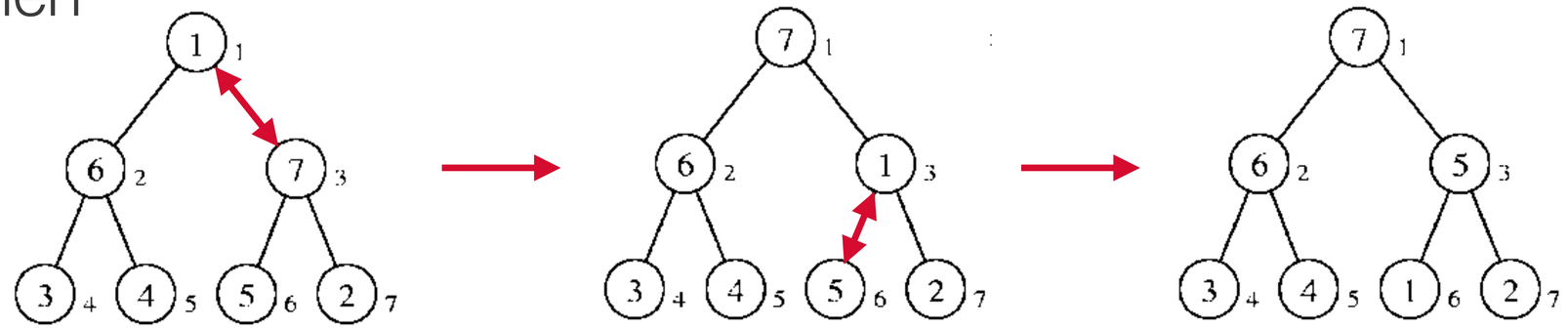
- Wenn Heap leer, dann fertig
- Ansonsten gib $\mathbf{k_1}$ aus
- Entferne $\mathbf{k_1}$ aus dem Heap
- Stelle **Heap-Bedingung** für restliche Schlüssel wieder her, so dass die neue Wurzel an Position 1 steht



Wiederherstellen der Heap-Bedingung



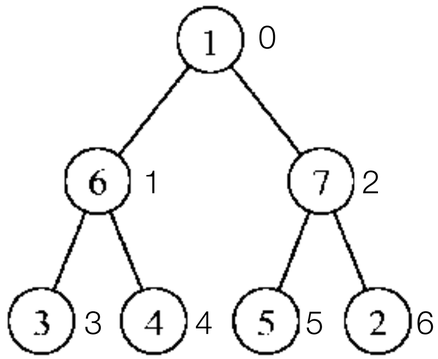
- Nach Entfernen der Wurzel sind zwei Teil-Heaps übrig
- Vereinige beide, indem der Schlüssel mit dem **höchsten Index** als Wurzel eingesetzt wird
- **k_1** versickern lassen (**sift down**), indem so lange mit dem jeweils größeren Nachfolger getauscht wird, bis beide Nachfolger kleiner/gleich sind oder unten angekommen



Universität Bremen

Implementierung des Versickerns

- Heap kann in einem Array implementiert werden
- Aufwand für das Versickern ist **O(log N)**



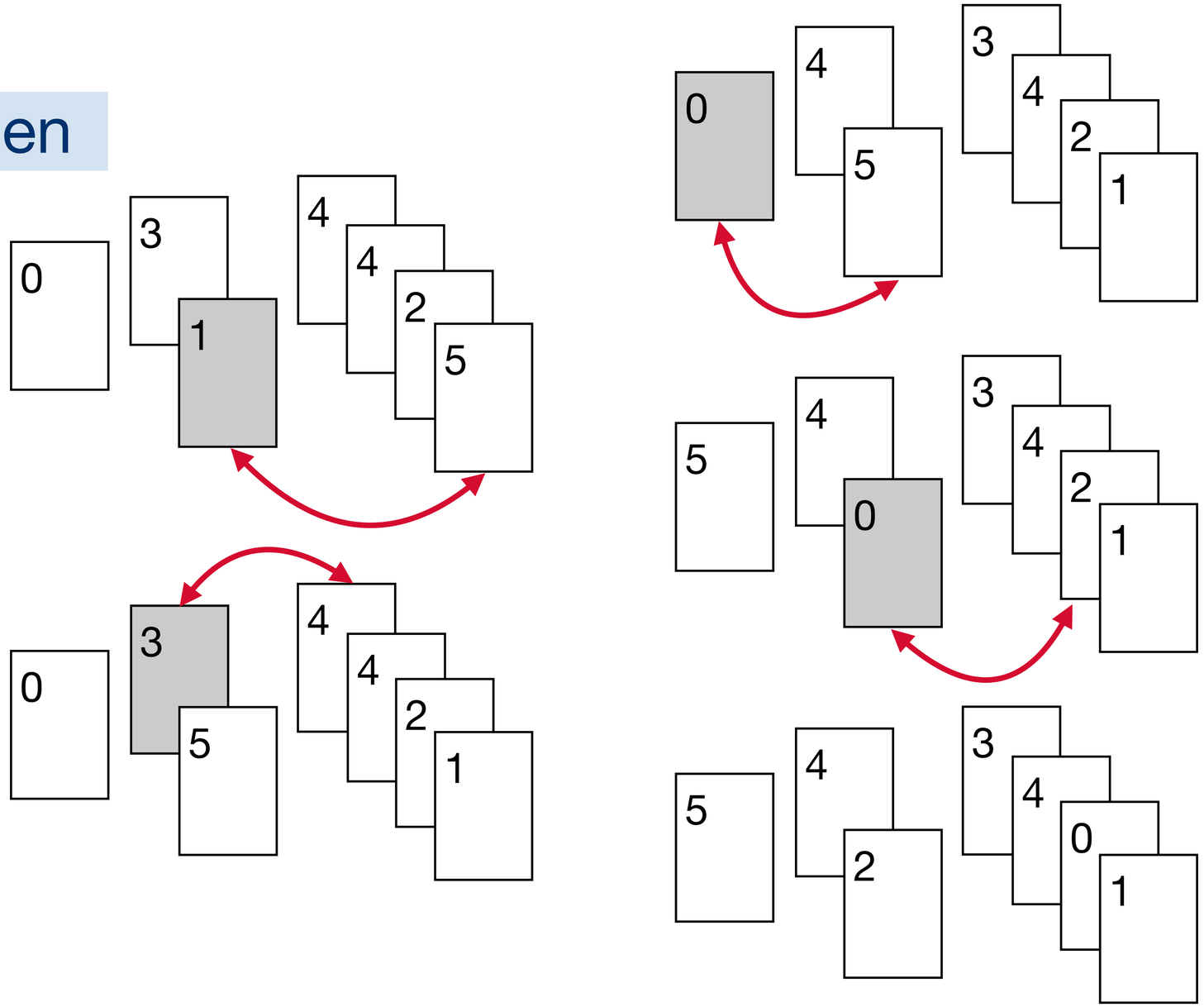

```
private static <T extends Comparable<T>>
        void siftDown(final T[ ] a, int i, final int n)
{
    while (i < n / 2) {
        int j = 2 * i + 1;
        if (j + 1 < n && a[j].compareTo(a[j + 1]) < 0) {
            ++j;
        }
        if (a[i].compareTo(a[j]) < 0) {
            swap(a, i, j);
            i = j;
        }
        else {
            break;
        }
    }
}
```

Universität Bremen

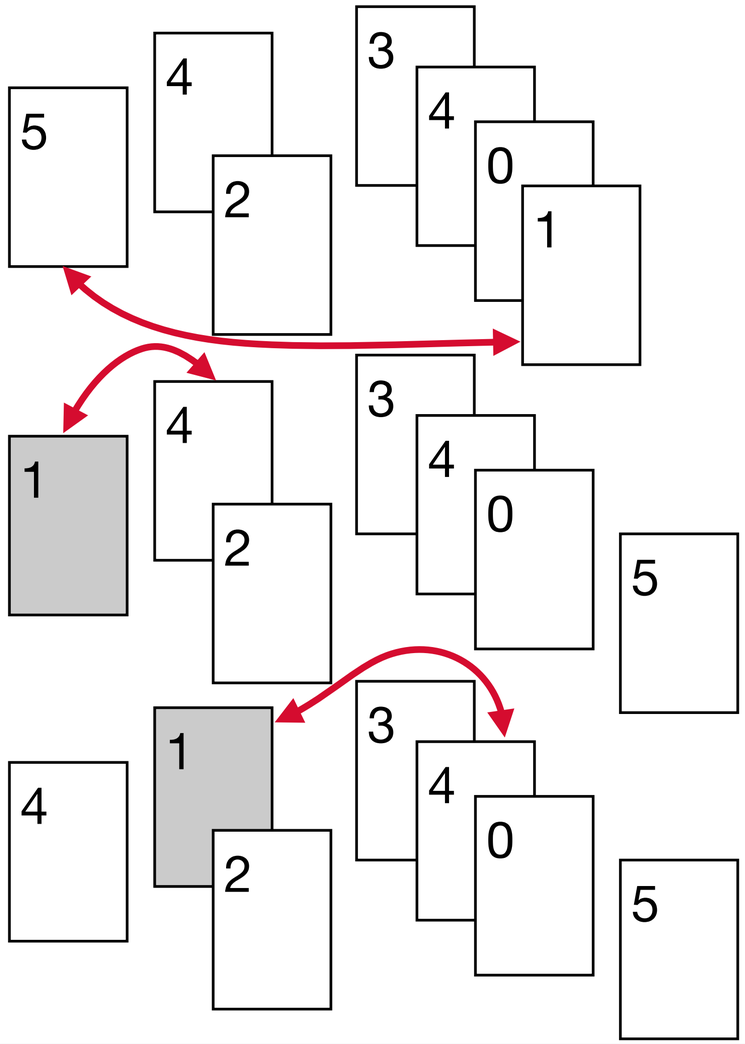
Heapsort

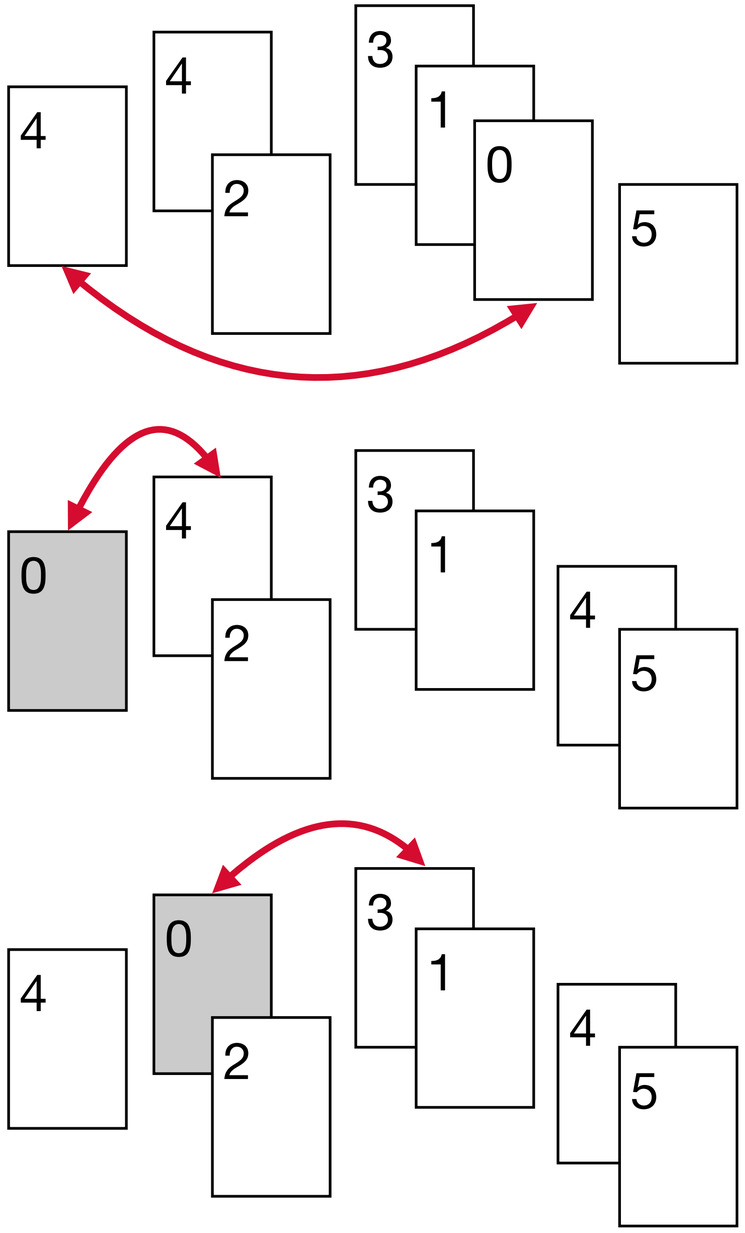
- Zuerst wird ein Heap aufgebaut, indem der Reihe nach vorne ein Element vorangestellt wird und dieses dann versickert
  - Das Voranstellen passiert nicht wirklich, stattdessen wächst der Heap einfach vom Ende des Arrays her
- Dann wird der Heap schrittweise geleert, wodurch jeweils das größte Element entnommen wird
  - Nach dem Entfernen des jeweils größten Elements ist der Heap am Ende um ein Element kürzer. Dort kann das soeben entnommene Element eingetragen werden

Heap aufbauen

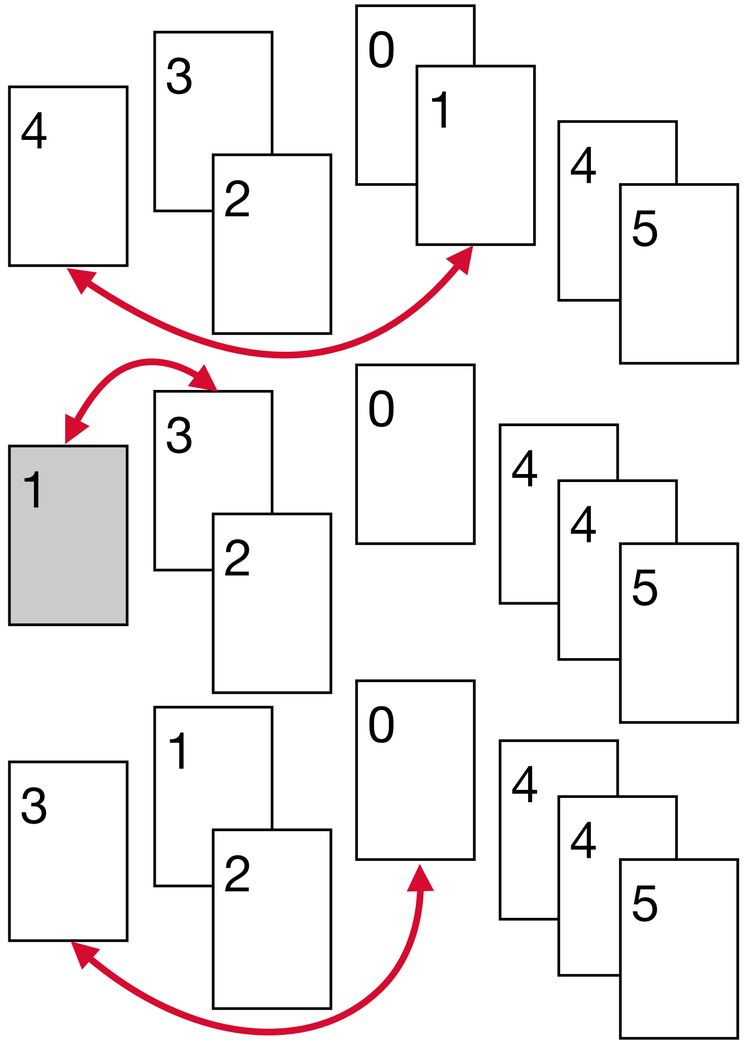


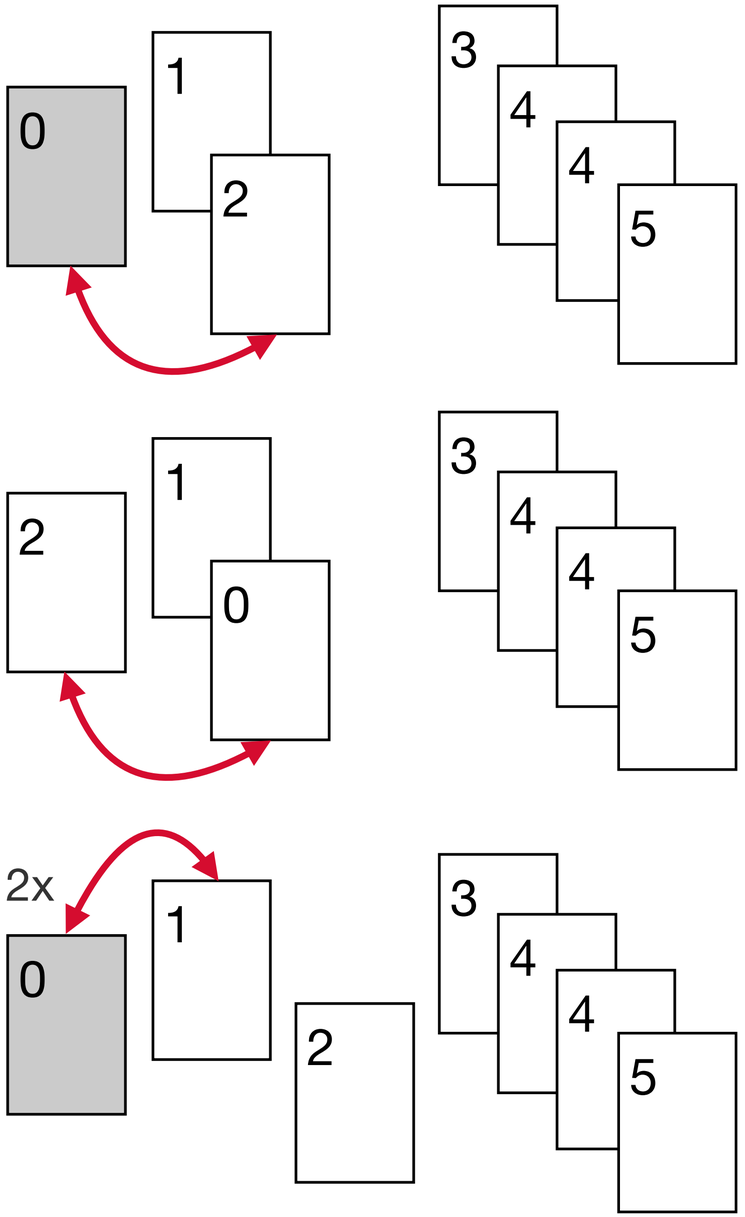
Heap abbauen / sortieren





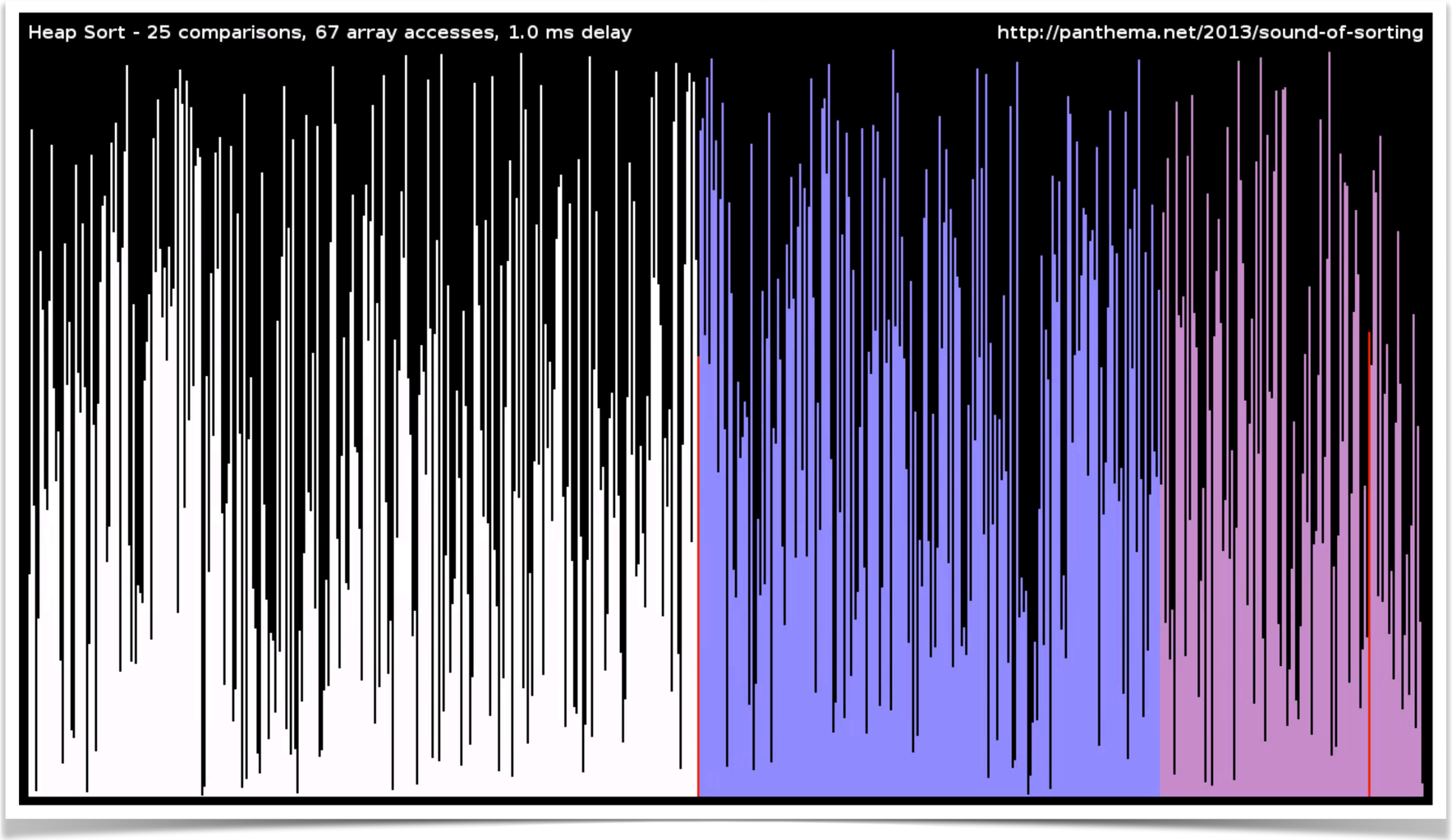
Heap abbauen / sortieren





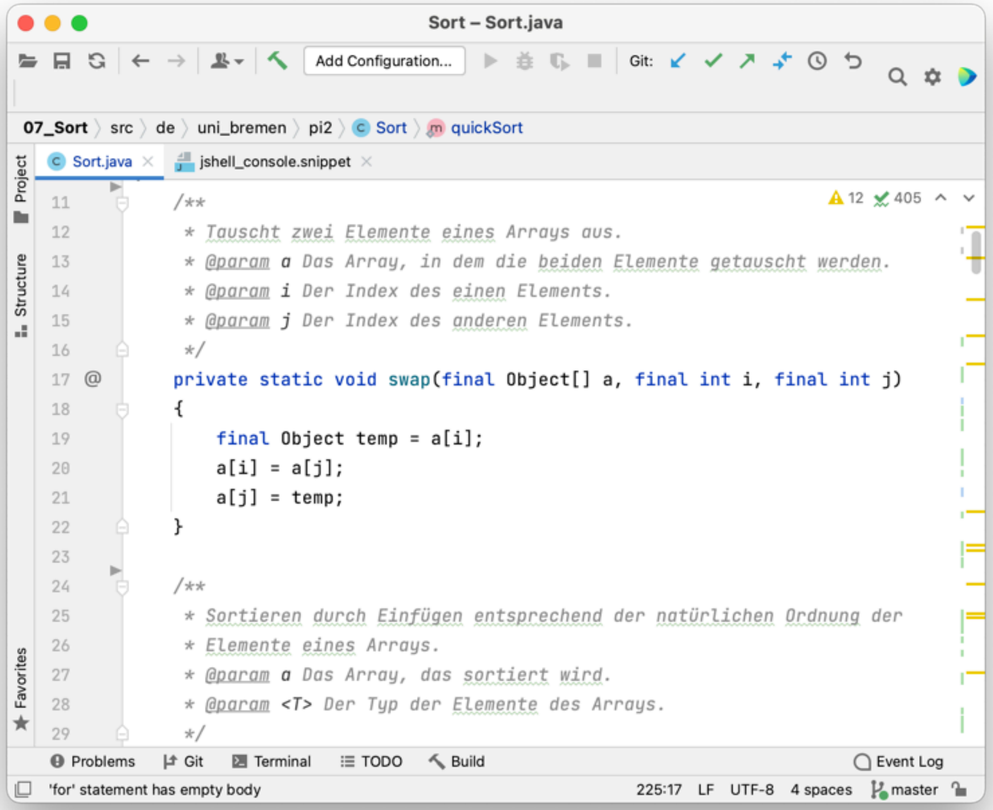
Universität Bremen

Heapsort: Demo



Universität Bremen

Heapsort: Demo



Universität Bremen

Heapsort: Aufwand

- **½N**-mal Versickern für Aufbauen des Heaps: **O(½ N log N)**
- **N**-mal Versickern für das Auslesen des Heaps: **O(N log N)**
- Schlechtester und durchschnittlicher Fall: **O(N log N)**
- Bester Fall: **O(N)** (Alle Elemente sind gleich)
- **Vorteil**: Speicherkomplexität ist **O(N)**
- **Nachteil**: Heapsort ist nicht stabil

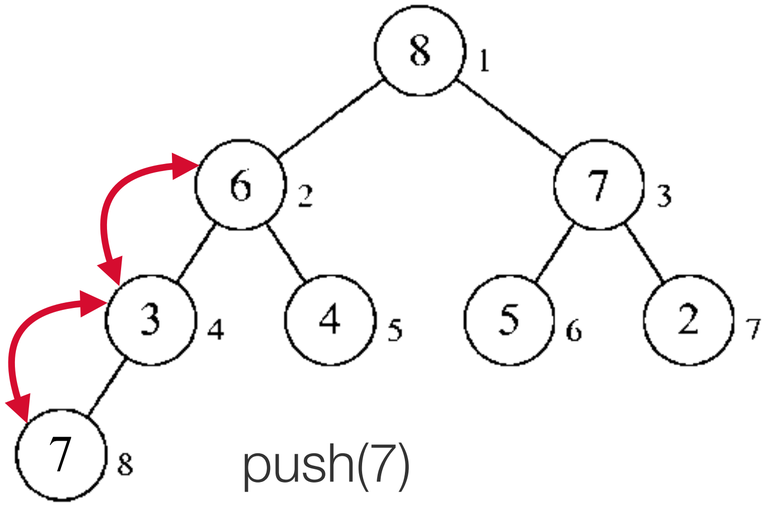
Universität Bremen

Vorrangwarteschlange

- Eine Vorrangwarteschlange (**Priority Queue**) ist eine Warteschlange, die die in ihr gespeicherten Werte entsprechend einer Ordnung (**Priorität**) wieder ausliefert
- Zugriff auf nächstes Element in **O(1)**, Einfügen und Entfernen in **O(log n)**
- Kann mit Heap implementiert werden
  - **top**: Erstes Element
  - **pop**: Ende an Anfang tauschen und **siftDown**, bisherigen Anfang zurückgeben
  - **push**: Neues Element ans Ende und **siftUp**

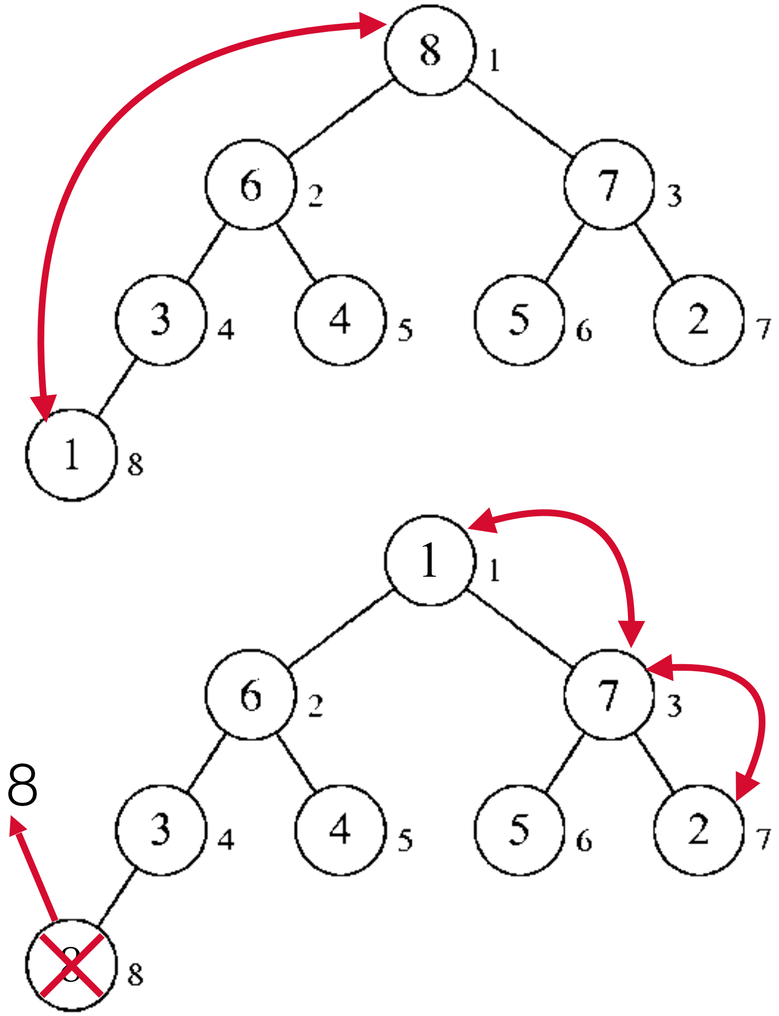
Vorrangwarteschlange: Beispiel

```
public class PriorityQueue<T extends Comparable<T>>
{
    private final List<T> heap = new ArrayList<T>();

    public boolean empty()
    {
        return heap.isEmpty();
    }

    public T top()
    {
        return heap.get(0);
    }
```




```
public void push(final T entry)
{
    heap.add(entry);
    for (int i = heap.size() - 1, j = (i - 1) / 2;
            i > 0 && heap.get(j).compareTo(heap.get(i)) < 0;
            i = j, j = (i - 1) / 2) {
        Collections.swap(heap, i, j);
    }
}
```

Vorrangwarteschlange: Beispiel




```
public T pop()
{
    Collections.swap(heap, 0, heap.size() - 1);
    int i = 0;
    while (i < heap.size() / 2 - 1) {
        int j = 2 * i + 1;
        if (j + 1 < heap.size() - 1 && heap.get(j)
                .compareTo(heap.get(j + 1)) < 0) {
            ++j;
        }
        if (heap.get(i).compareTo(heap.get(j)) < 0) {
            Collections.swap(heap, i, j);
            i = j;
        } else {
            break;
        }
    }
    return heap.remove(heap.size() - 1);
}
```

Universität Bremen

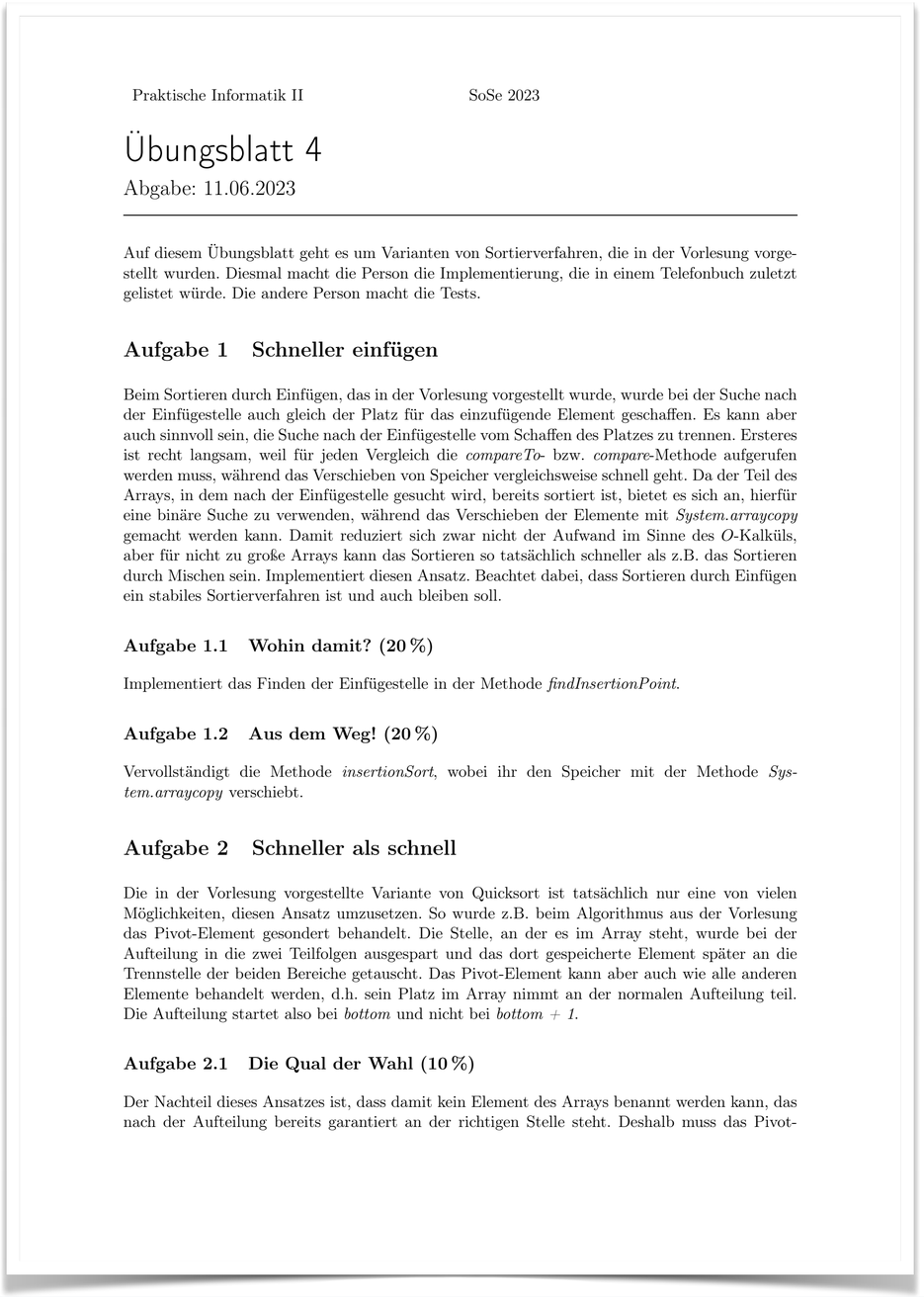
Zusammenfassung der Konzepte

- **Greedy**-Sortierverfahren haben mittleren Aufwand von **$O(N^2)$**
- **Divide and Conquer**-Sortierverfahren haben mittleren Aufwand von **O(N log N)**
- **Mergesort** sortiert beim **rekursiven Aufstieg**
  - **Doppelter Platzaufwand, stabil**
- **Quicksort** sortiert beim **rekursiven Abstieg** (ebenso **Radixsort**)
  - Aufwand im schlechtesten Fall bei **Quicksort**: **$O(N^2)$**, auf Arrays **nicht stabil**
- **Heapsort** baut **Heap** auf und wieder ab
  - **O(N log N)** im schlechtesten Fall, **O(N)** im besten Fall, **nicht stabil**

Universität Bremen

Übungsblatt 4

- Aufgabe 1: Sortieren durch Einfügen mit binärer Suche nach Einfügestelle
- Aufgabe 2: Quicksort ohne Sonderbehandlung für Pivot-Element bei Aufteilung
  - Warum muss Pivot-Element anders gewählt werden?
  - Implementierung der anderen Pivot-Wahl
  - Warum kann bei Aufteilung auf Bereichsprüfung verzichtet werden?
  - Implementierung der anderen Aufteilung



Praktische Informatik II SoSe 2023

Übungsblatt 4

Abgabe: 11.06.2023

Auf diesem Übungsblatt geht es um Varianten von Sortierverfahren, die in der Vorlesung vorgestellt wurden. Diesmal macht die Person die Implementierung, die in einem Telefonbuch zuletzt gelistet würde. Die andere Person macht die Tests.

Aufgabe 1 Schneller einfügen

Beim Sortieren durch Einfügen, das in der Vorlesung vorgestellt wurde, wurde bei der Suche nach der Einfügestelle auch gleich der Platz für das einzufügende Element geschaffen. Es kann aber auch sinnvoll sein, die Suche nach der Einfügestelle vom Schaffen des Platzes zu trennen. Ersteres ist recht langsam, weil für jeden Vergleich die *compareTo*- bzw. *compare*-Methode aufgerufen werden muss, während das Verschieben von Speicher vergleichsweise schnell geht. Da der Teil des Arrays, in dem nach der Einfügestelle gesucht wird, bereits sortiert ist, bietet es sich an, hierfür eine binäre Suche zu verwenden, während das Verschieben der Elemente mit *System.arraycopy* gemacht werden kann. Damit reduziert sich zwar nicht der Aufwand im Sinne des O-Kalküls, aber für nicht zu große Arrays kann das Sortieren so tatsächlich schneller als z.B. das Sortieren durch Mischen sein. Implementiert diesen Ansatz. Beachtet dabei, dass Sortieren durch Einfügen ein stabiles Sortierverfahren ist und auch bleiben soll.

Aufgabe 1.1 Wohin damit? (20 %)

Implementiert das Finden der Einfügestelle in der Methode *findInsertionPoint*.

Aufgabe 1.2 Aus dem Weg! (20 %)

Vervollständigt die Methode *insertionSort*, wobei ihr den Speicher mit der Methode *System.arraycopy* verschiebt.

Aufgabe 2 Schneller als schnell

Die in der Vorlesung vorgestellte Variante von Quicksort ist tatsächlich nur eine von vielen Möglichkeiten, diesen Ansatz umzusetzen. So wurde z.B. beim Algorithmus aus der Vorlesung das Pivot-Element gesondert behandelt. Die Stelle, an der es im Array steht, wurde bei der Aufteilung in die zwei Teilfolgen ausgespart und das dort gespeicherte Element später an die Trennstelle der beiden Bereiche getauscht. Das Pivot-Element kann aber auch wie alle anderen Elemente behandelt werden, d.h. sein Platz im Array nimmt an der normalen Aufteilung teil. Die Aufteilung startet also bei *bottom* und nicht bei *bottom + 1*.

Aufgabe 2.1 Die Qual der Wahl (10 %)

Der Nachteil dieses Ansatzes ist, dass damit kein Element des Arrays benannt werden kann, das nach der Aufteilung bereits garantiert an der richtigen Stelle steht. Deshalb muss das Pivot-