IX. Suchen und Sortieren

1. Suchen in Feldern
2. Einfache iterative Sortierverfahren und deren Aufwand
3. Sortieren durch Teilen-und-Herrschen
4. Sortieren mit Halde (*Heapsort*)
5. Optimierende Suche – *Backtracking*

1. Suchen in Feldern

- Allgemeine Einordnung
- Suchen in ungeordneten und geordneten Feldern

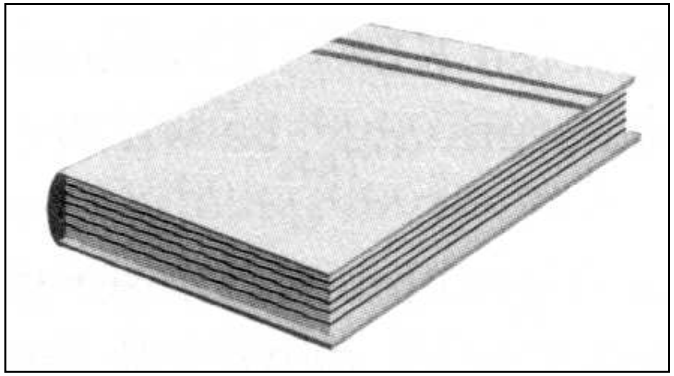
Allgemeine Einordnung

Suchverfahren

- Suchverfahren sind Algorithmen, die in einem Suchraum nach Elementen oder Mustern mit bestimmten Eigenschaften suchen
- Wir können Suchverfahren grob in zwei Kategorien unterteilen:
  - **Einfache Suchverfahren**, die – unter Verwendung bestimmter Datenstrukturen – einen Suchraum durchlaufen und nach Lösungen mit bestimmten Eigenschaften suchen, z.B. bestimmte Elemente, Konfigurationen von Elementen, optimale Wege, etc.
  - **Heuristische Suchverfahren**, die zusätzliche Informationen (Wissen) über den Suchraum und die dortige Datenverteilung zur Beschleunigung der Suche nutzen oder **optimierende Suchverfahren**, in denen Lösungen von Konfigurationsproblemen gesucht werden und dabei den Raum aller möglichen Konfigurationen geeignet absuchen
- In diesem Abschnitt werden einfache Suchverfahren vorgestellt, die auf linearen Listen bzw. Bäumen arbeiten

Daten und Schlüsselmerkmale – Suche in einem Telefonbuch

- **Eingabe**: Menge von Datensätzen



Einträge (Datensätze) der Form

Name, Vorname
Adresse
Telefonnummer

Eigenschaften: In einem Telefonbuch sind die Einträge üblicherweise alphabetisch nach dem (Nach-) Namen geordnet

- Konkrete Aufgabe: Suche nach einem (oder mehreren) Datensätzen mit einer bestimmten Eigenschaft (Merkmal), z.B.
  - Name (Name, Vorname),
  - Telefonnummer,
  - etc.
- Nach bestimmten Elementen kann gesucht werden, indem ein **Suchschlüssel** definiert und die Elemente danach durchsucht und verglichen werden

Abstrakte Formulierung des Suchproblems

- Das abstrakte Problem:
  - Gegeben ist eine Folge F von Datenobjekten
  - Es soll eine Methode realisiert werden, die ein Datenobjekt x in dieser Folge nach einem Suchschlüssel α sucht und dazu die Position p(F, x, α) eines solchen Objekts in der Folge F bestimmt

- Eine abstrakte Lösung (die Strategie wird vereinfachend mit x = α formuliert):

Der Suchschlüssel ist hier identisch mit dem Element der Folge

1. Initialisierung:
   `result = -1;`
   `S`: Menge der Suchpositionen in F
2. Basisfall:
   `if (S == ∅) return result;`
3. Reduktion:
   wähle die nächste Suchposition p und entferne p aus S
4. Rekursionsfall / iterativer Fall:
   `if (F[p] == x) return p;`
   weiter bei Schritt 2

Konkretisierungen dieses Algorithmenschemas legen u.a.
- die genaue Wahl der Suchposition sowie
- die Realisierung des Tests (S == ∅)

fest

Naive Suche – Lineare (sequenzielle) Suche

Vorgehensweise

- 1. Die Suche beginnt ganz am Anfang der Folge

  2. Die lineare Suche

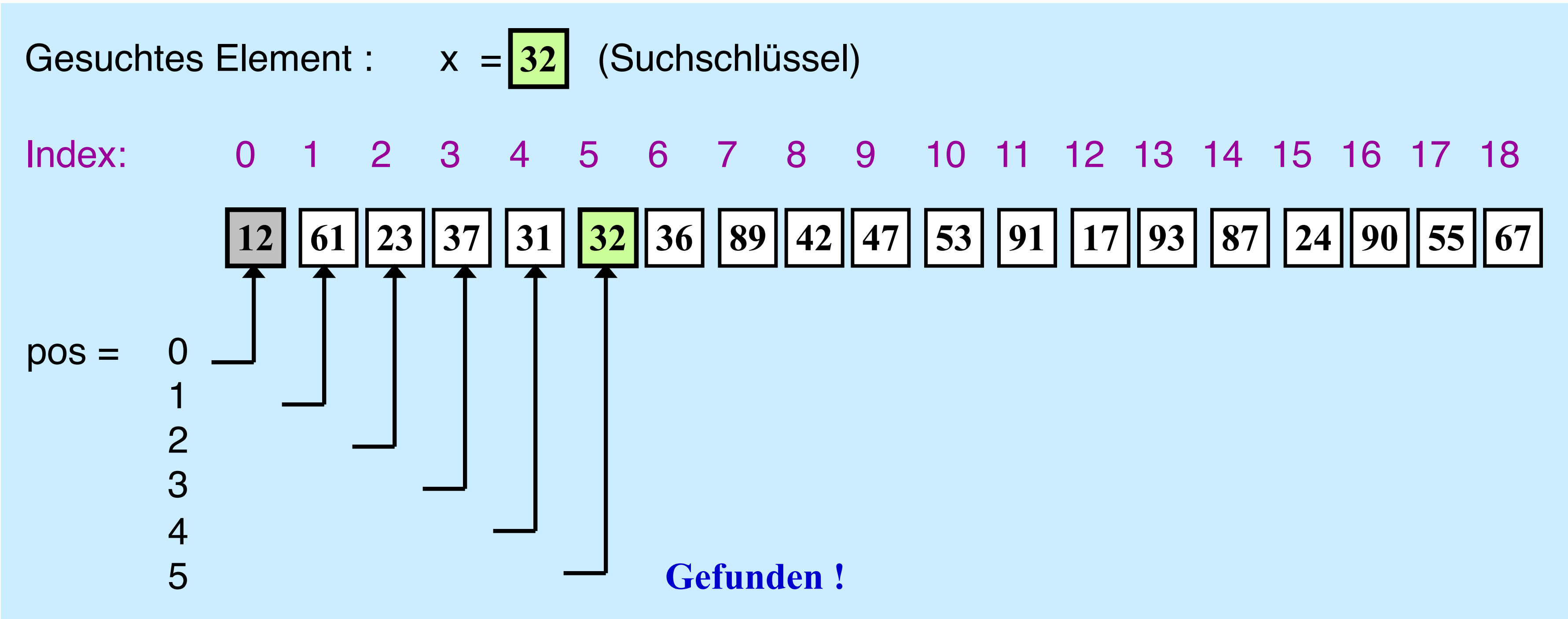
     - durchläuft die Folge sequenziell (d.h. Element für Element) und
     - testet jedes Element auf Übereinstimmung mit dem gegebenen **Suchschlüssel**

  3. Terminierung der Suche falls
     - das Element gefunden oder
     - wenn das Ende der Folge erreicht ist

- Wenn die Folge bzgl. des **Suchschlüssels** nicht geordnet ist (vgl. Suche in einen Telefonbuch nach Telefonnummern), so muss sequenziell gesucht werden – die Ordnung der Elemente kann nicht für die Organisation der Suche genutzt werden

(aus T.Seidl, J.Enderle, Binäre Suche. In B.Vöcking et al. (Hrsg.), Taschenbuch der Algorithmen, Kap.1, Springer, Berlin, 2008)

Schema der linearen Suche (in <u>ungeordneter</u> Struktur, kein Element mehrfach vorhanden)



- <u>**Beobachtung**</u>:
  - Wäre das gesuchte Element x = $\boxed{67}$, so hätte man **19** Suchschritte benötigt !
  - Der Aufwand (Laufzeit) ist proportional zur Länge n der Liste, also O(n)

Aufwand der linearen Suche bei unsortierten Folgen

- Im **günstigsten Fall** befindet sich das gesuchte Element an der ersten Stelle und die Suche terminiert nach dem ersten Vergleich; es ist 1 Vergleichsschritt notwendig
- Im **ungünstigsten Fall** befindet sich das Element an der letzten Position in der Folge **oder** ist in dieser nicht enthalten; es werden n Vergleichsschritte benötigt
- Für den **Durchschnittsfall** (ein Element kann sich an irgendeiner Position befinden – wir betrachten viele Versuche) erwarten wir, dass (n+1) / 2 Vergleichsschritte durchgeführt werden müssen

	Anzahl der Vergleiche
günstigster Fall	1
schlechtester Fall	n
Durchschnitt (erfolgreiche Suche)	(n + 1) / 2
Durchschnitt (erfolglose Suche)	n

- Im schlechtesten Fall (*worst case*) sowie im Durchschnittsfall (*average case*) ist die Laufzeit von der Ordnung O(n) (nur im günstigsten Fall, wenn das Element an der ersten Stelle steht, ist die Laufzeit (trivialerweise) konstant)

Implementierung in Java

- **Einordnung**: Hier werden Ganzzahlen als Datenobjekte verwendet; als Suchschlüssel werden ebenfalls Ganzzahlen verwendet
- Implementierung (Demo: LinearSearch.java)

```
public class LinearSearch {
    static final int NO_KEY = -1;  // ungueltige Position = Element nicht gefunden

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("usage: LinearSearch <key>");
            return;
        }

        int[] seq = new int[] { 4, 2, 7, 5, 6, 9, 8, 11};
        int   k   = Integer.parseInt(args[0]);

        System.out.println("Sequenzielle Suche (Position): " +
                           search(seq, k));
    }  // end main

    static int search(int[] sequence, int key) {
        for (int i = 0; i < sequence.length; i++)
            if (sequence[i] == key)
                return i;
        return NO_KEY;
    }  // end search
}  // end class LinearSearch
```

Aufruf: `LinearSearch 7` (Fehlermeldung bei fehlendem Argument (Schlüssel))

Konvertierung der Konsolen-Eingabe in `int`-Wert

Allgemeine Vorgehensweise zur Analyse der Komplexität

- *Wie viele Elementaroperationen werden ausgeführt?*
- Erfordert konkrete Betrachtung der jeweiligen Implementierung
- Dazu seien:

  k_1: # Initialisierungsoperationen

  k_2: # Operationen für den Test, ob der Basisfall vorliegt

  k_3: # Operationen für Reduktion und Rekursionsfall bzw. die Verwaltung der Schleifen-durchläufe

```
public class LinearSearch {
   static final int NO_KEY = -1;

   public static void main(String[] args) {
      ...
   }

   static int search(int[] sequence, int key) {
      for (int i = 0; i < sequence.length; i++)
         if (sequence[i] == key)
            return i;
      return NO_KEY;
   }  // end search
}
```

Analyse der linearen Suche in Folge F (= `seq`)

- Die **`for`**-Schleife wird höchstens n-mal ausgeführt, wenn n die Länge von F ist
- Operationen:

$$\Rightarrow \quad T_{worst}(n) = k_1 + n \cdot (k_2 + k_3)$$
$$= 1 + n \cdot (1 + 2)$$

$$\Rightarrow \quad T_{worst}(n) = O(n)$$

Binäre Suche (in Folgen)

Grundidee

- **<u>Voraussetzung</u>**: Die Folge der Datenobjekte ist nach einem Schlüssel (z.B. Name) aufsteigend sortiert
- 1. Beginn der Suche in der Mitte der Folge
  2. Fallunterscheidung für betrachteten Eintrag:

     Falls Schlüssel kleiner als Suchschlüssel:
     dann rechts (in rechter Hälfte) weitersuchen;
     Sonst: In linker Hälfte der Folge weitersuchen;
     aktuelle Hälfte := ausgewählte Hälfte (aus der Fallunterscheidung);

  3. Für aktuelle Hälfte wieder mittleren Eintrag betrachten
  4. Halbierung fortsetzen, bis
     - Suchschlüssel gefunden oder
     - keine Halbierung mehr möglich ist

- **Pseudocode:**

```
binarySearch(arr, key, min, max):
...
while (min <= max):
   middle := (min + max) / 2;  // Mitte bestimmen, Ergebnis runden
   if (arr[middle] = key) then
      <Pos. des gesuchten Elements: Mitte>
   else if (arr[middle] > key) then
      max := middle - 1;
   else (arr[middle] < key) then
      min := middle + 1;
   endif
<Element nicht gefunden>
...
```

- Beispiel: Suche nach „Nelly“ in sortiertem CD-Regal mit 501 Einträgen
  (`rack[k]` kennzeichnet die k-te CD im Regal; `k = 0 ... 500`)

```
1.  binarySearch(rack, "Nelly", 0, 500)
    - rack[250] = "Kelly Family"      ⇒ rechts weitersuchen
2.  min = 251, max = 500
    - rack[375] = "Rachmaninov"       ⇒ links weitersuchen
3.  min = 251, max = 374
    - rack[312] = "Lionel Hampton"    ⇒ rechts weitersuchen
4.  min = 313, max = 374
    - rack[343] = "Nancy Sinatra"     ⇒ rechts weitersuchen
5.  min = 344, max = 374
    - rack[359] = "Nelly"             ⇒ gefunden!
```

Schema der binären Suche (in geordneter Struktur, kein Element mehrfach vorhanden)

Gesuchtes Element : x = 32 (Suchschlüssel)

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	12	17	23	24	31	32	36	37	42	47	53	55	61	67	87	89	90	91	93

min = 0 max = 18 → m = 9 Element = 47, x < 47 → **x links**

12 17 23 24 31 32 36 37 42

min = 0 max = 8 → m = 4 Element = 31, x > 31 → **x rechts**

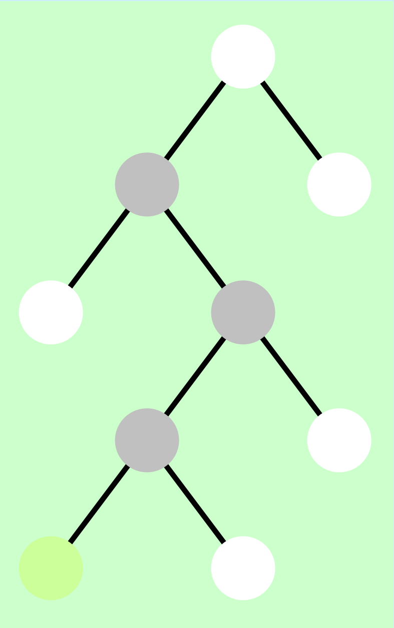
32 36 37 42

min = 5 max = 8 → m = 6 Element = 36, x < 36 → **x links**

32 36

min = 5 max = 5 → m = 5 Element = 32, x = 32 → x

Gefunden !



- **Beobachtung**: Man benötigt eine konstante Anzahl von Suchschritten, die Datenmenge wird dabei jedes mal halbiert; der Aufwand (Laufzeit) ist proportional zur Baumhöhe, also O(log n) – außer wenn ein „mittleres“ Element schon vorher das Gesuchte ist

Aufwand der binären Suche bei sortierter Folge der Länge n

- *Wie lange muss man höchstens suchen?*
  - geg.: Sortierte Folge mit n Einträgen
  - gesucht: Anzahl der maximal benötigten Suchschritte bei der Suche nach einem bestimmten Element (Suchschlüssel)

- Fragestellung anders herum …
  - geg.: k Suchschritte
  - gesucht: Anzahl der Einträge, die sich mit k Suchschritten durchsuchen lassen

    mit 1 Vergleich: 2^1 -1 Einträge durchsucht
    mit 2 Vergleichen: 2^2 -1 Einträge durchsucht
    mit 3 Vergleichen: 2^3 -1 Einträge durchsucht
    ...
    mit k Vergleichen: 2^k -1 Einträge durchsucht

  - *Wie viele Vergleiche k für n = 10.000 Einträge?* (mit $n = 2^k$)

    Antwort: $n = 10000 = 2^k \Rightarrow k = \log_2(10.000) \approx 13{,}29 \rightarrow$ 14 Vergleiche

  - Vergleichbares Problem: Ratespiel … generiere eine Zahl zwischen 0 und 100; finde mit 6 Ja/Nein-Fragen die Zahl heraus (vgl. GuessingGame.java, **Teil VI**)

Aufwand der binären Suche bei sortierten Folgen

- Die Länge der jeweils betrachteten Folge wird in jedem Schritt in etwa halbiert
- Die asymptotische Komplexität der binären Suche ist – wie vorangehend gezeigt – im schlechtesten Fall *logarithmisch* in n, d.h. in $O(\log_2 n)$
- Der Unterschied zur linearen Suche ist signifikant

	Anzahl der Vergleiche
günstigster Fall	1
schlechtester Fall	$\approx \log_2 n$
Durchschnitt (erfolgreiche Suche)	$\approx \log_2 n$
Durchschnitt (erfolglose Suche)	$\approx \log_2 n$

- Im schlechtesten Fall (*worst case*) sowie im Durchschnittsfall (*average case*) ist die Laufzeit von der Ordnung $O(\log_2 n)$ (nur im günstigsten Fall, wenn das Element in der Mitte der Folge steht, ist die Laufzeit (trivialerweise) konstant)

Implementierung in Java (Demo: BinarySearch.java)

```
public class BinarySearch {
    static final int NO_KEY = -1;  // ungueltige Position = Element nicht gefunden

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("usage: BinarySearch <key>");
            return;
        }

        int[] seq = new int[] { 2, 4, 5, 6, 7, 8, 9, 11};  // Elemente sind sortiert
        int   k   = Integer.parseInt(args[0]);  // konvertiere Konsole-Eingabe in int

        System.out.println("Binaere Suche (Position): " + search(seq, k));
    }  // end main

    static int search(int[] sequence, int key) {
        int min = 0,
            max = sequence.length - 1;

        while (min <= max) {
            int middle = (min + max) / 2;

            if (sequence[middle] == key)
                return middle;
            else if (sequence[middle] > key)  // Element in der unteren Haelfte?
                max = middle - 1;
            else                              // Element in der oberen Haelfte?
                min = middle + 1;
        }
        return NO_KEY;
    }  // end search
}  // end class BinarySearch
```

Aufruf: `BinarySearch 7`
(Fehlermeldung bei fehlendem Argument (Schlüssel))

Markierung des mittleren Elements und Halbierung der Datenmenge

Aufwand im Vergleich

- Daten sind in *Arrays* gespeichert und werden durchsucht; hier wurde als Suchschlüssel der Wert des Elements selbst verwendet (für die lineare Suche ist die Anordnung der Elemente beliebig; für die binäre Suche müssen die Elemente in aufsteigernder Ordnung gespeichert werden)

Verfahren	**Anzahl der Elemente**	**mittlerer Aufwand**	**max. Aufwand**	$O(\cdot)$
Lineare Suche	n	$n/2$	n	$O(n)$
Binäre Suche	n	$\log_2(n)$	$\log_2 n$	$O(\log n)$

- Für die Komplexitätsabschätzung bei der Bestimmung der oberen Schranke gilt wegen

$\log_2(n) = \underbrace{\log_2(10)}_{=c} \cdot \log(n)$, dass die Laufzeit in $O(\log n)$ liegt.

- Für die effiziente binäre Suche muss zusätzlicher Aufwand für die Sortierung der Elemente betrieben werden

2. Einfache iterative Sortierverfahren und deren Aufwand

- Motivation, Einordnung und Begriffe
- Internes Sortieren – Schema und Notation
- *Selection sort* – Sortieren durch direkte Auswahl
- *Insertion sort* – Sortieren durch direktes Einfügen
- *Bubble sort* – Sortieren durch direktes Vertauschen
- Allgemeine Analyse der Laufzeiten
- Detailanalyse zum Aufwand von *Selection sort*

Motivation, Einordnung und Begriffe

Sortieren

- **Begriffsbestimmung**: Sortieren ist der Prozess des Ordnens einer gegebenen Menge von Objekten (Elementen) nach einem bestimmten Ordnungskriterium, wobei eine bestimmte Eigenschaft der Objekte zugrunde gelegt wird, auf der ein Suchschlüssel formuliert werden kann, z. B. Größe, Gewicht, Zahl, Zeichen etc.
- **Ziel**: Vereinfachung des späteren Suchens nach Elementen in der (dann geordneten) Menge
- Beispiele:
  - Telefonbücher … alphabetische Ordnung
  - Literaturverzeichnisse … alphabetische Ordnung oder Reihenfolge der Zitate
  - Stichwortverzeichnis / Index in Büchern … alphabetische Folge
  - Wörterbücher / Lexika
  - Adressbücher
  - …

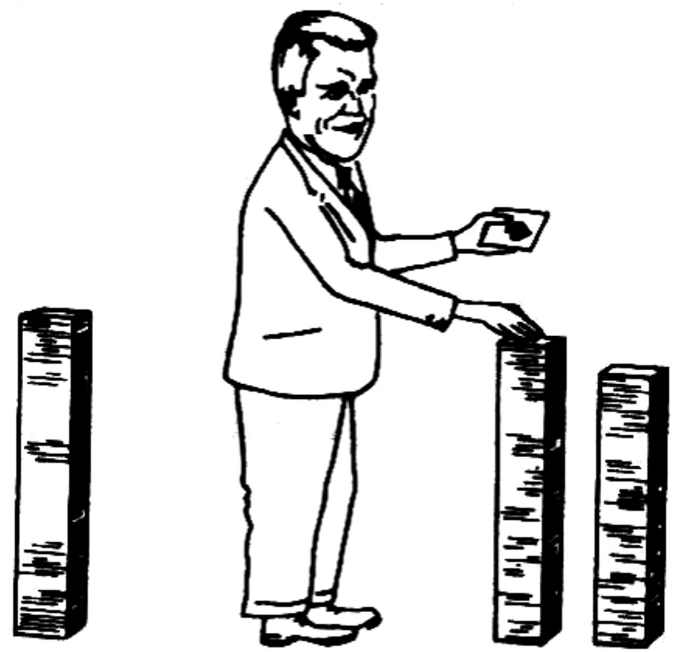
Kategorien von Sortierverfahren

Internes Sortieren

- Der Datenbestand liegt während des Sortierens im Hauptspeicher, d.h. alle Elemente sind zugreifbar
- Bsp.: Sortieren eines *Arrays*

Externes Sortierverfahren

- Der überwiegende Teil des Datenbestands lagert während des Sortierens auf Hinter-grundspeichern, d.h. von jedem Bereich (Stapel) sind nur die obersten Elemente sichtbar
- Bsp.: Sortieren von sequentiellen Verzeichnissen (*Files*)
- **Hinweis**: Das Problem lässt sich auf das „interne Sortieren" zurückführen!



(N. Wirth. Algorithmen und Datenstrukturen. Teubner, Stuttgart, 1983)

Internes Sortieren – Schema und Notation

Allgemeines

- Schema:

- Prinzip bzw. Anforderungen: Wirtschaftliche Verwendung des vorhandenen Speichers; Sortiermethoden, die Elemente **am Ort sortieren** (!) und **möglichst nicht** das Feld der Daten duplizieren!

- Methoden (Auswahl):
  - **Iterative Verfahren**: Sortieren durch Auswählen, Einfügen, Austauschen, … – *Selection sort*, *Insertion sort*, *Bubble sort*
  - **Rekursive Verfahren**: Sortieren durch Zerlegen der Daten in Teile, Sortieren durch Austausch, … – *Quicksort*, *Mergesort*, *Heap sort*

Sortierproblem

Struktur

- Eingabe: Sei `arr[0..N-1]` ein vorgegebenes *Array* mit N Elementen

  Gesucht: Genau diejenige Permutation (Anordnung vertauschter Elemente) von `arr[0..N-1]`, in der die Elemente „sortiert sind", d.h. es gilt

  `arr[i] ≤ arr[j]`

  für alle `i < j` (mit i, j $\in$ [0, N-1])

  Anders ausgedrückt: Die Anzahl der **Inversionen** in einer sortierten Reihenfolge ist Null! (Inversion bedeutet: `i < j`, aber `arr[i] > arr[j]`)

- Das Feld `arr[0..N-1]` kann als Elemente auch N unsortierte (komplexe) Datensätze enthalten;

  `arr[i]` ist dann **kein einzelnes Element**, **sondern** ein **Objekt** mit einer Reihe von Eigenschaften (oder Merkmalen); es muss eine geeignete Komponente des Objekts als (Sortier-) Schlüssel ausgewählt werden, nach dem sortiert und die Elemente geordnet werden können

- ≤ und < können eine beliebige Ordnungsrelation ausdrücken und sind nicht auf numerische Vergleiche beschränkt (z.B. Stringsortierung)

Notation

- Gegeben:
  - n Datensätze, jeder hat die Form

Sortierschlüssel (*key*)	Inhalt / Daten

    Der **Sortierschlüssel** kann aus einem oder mehreren Teilfeldern bestehen

  - Ordnungsrelation auf den Schlüsseln (vorab definiert)

- <u>Bsp</u>.:

Schlüssel

Daten

* ? $ % § ! + = X O : ; > <

Start

mit
- Elementgröße (Box) ⟷ Schlüssel (*key*)
- Buchstaben / Zeichen ⟷ Daten / Objekt-Information

Sortierung aufsteigend nach Elementgröße

O = < + % > : ! ? ; $ * % X

Ziel

Ansatzpunkte für Sortieralgorithmen

folgt aus der vorangehenden Darstellung des Sortierproblems …

- **Sukzessive Reduktion der Anzahl der Inversionen**

  Verfahren:
  - Sortieren durch Auswählen – *Selection sort* (iterativ)
  - Sortieren durch Einfügen – *Insertion sort* (iterativ)
  - Austauschen – *Bubble sort* (iterativ)

- ***Divide-and-conquer*-Idee**

  Schema: $\boxed{\text{Aufteilen der Daten}} \rightarrow \boxed{\text{Sortieren der Teile}} \rightarrow \boxed{\text{Zusammenfügen}}$

  Verfahren:
  - *Mergesort* (rekursiv)
  - *Quicksort* (rekursiv)

- **Sortieren mit „Halde“**

  Verfahren: *Heap sort* (rekursiv)

Selection sort – Sortieren durch direkte Auswahl

Methode

- **Sortieren durch direktes Auswählen**

  - Start: Feld mit N Elementen, `arr[0..N-1]`
  - Ablauf:
    - **Auswahl** eines Elements mit kleinstem Schlüssel
    - **Austausch** gegen das erste Element im Feld, `arr[0]`

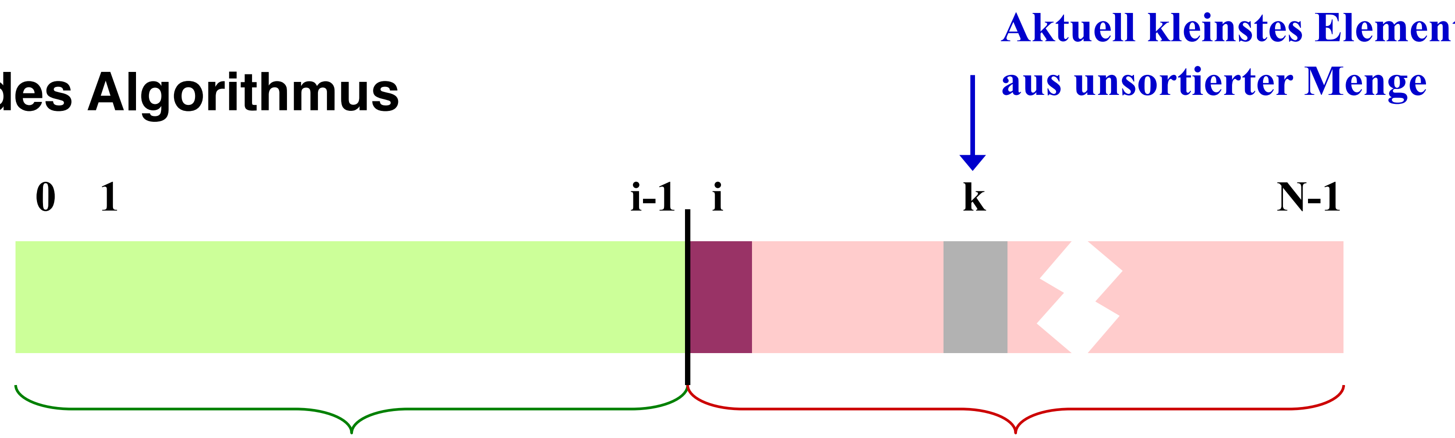
  - Wiederholung der obigen Schritte mit den

    restlichen N-1 Elementen, d.h. `arr[1..N-1]`
    restlichen N-2 Elementen, d.h. `arr[2..N-1]`
    :

- **Hinweis**: Die Reihenfolge der Sortierung geschieht von „unten-nach-oben“; **alternativ** kann auch von „oben-nach-unten“ sortiert werden!

Schema des Algorithmus

Vorher:



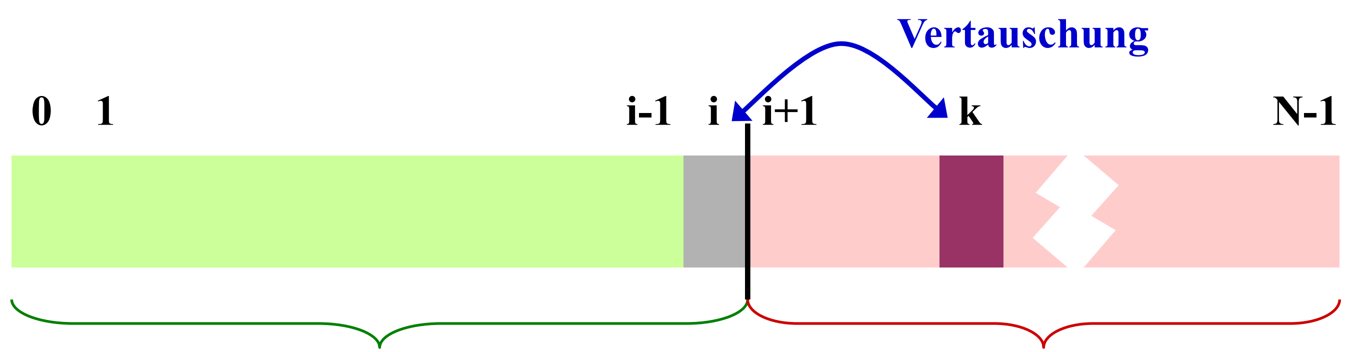
Bereits <u>endgültig</u> sortiertes Feld

$(\ \forall j=0..i\text{-}2\ )\text{: } arr[j] \leq arr[j+1]$

unsortiertes Feld

$k = \min(\ arr[i..N\text{-}1]\ ) \geq arr[i\text{-}1]$

Nachher:



Bereits <u>endgültig</u> sortiertes Feld

$arr`[i] = k$

$= \min(\ arr`[i..N\text{-}1]\ )$

$arr`[i] = \max(\ arr`[0..i]\ )$

unsortiertes Feld

$\min(\ arr`[(i+1)..N\text{-}1]\ ) \geq arr`[i]$

Jedes Element wird max. 1-mal von höherer (Index-) Position nach vorn (zur aktuellen Position) bewegt – und damit endgültig einsortiert !

Beispiel

Beispiel		44	55	12	42	94	18	06	67	Start
i = 0 :	j = 1 .. 7	44	55	12	42	94	18	06	67	
i = 1 :	j = 2 .. 7	06	55	12	42	94	18	44	67	
i = 2 :	j = 3 .. 7	06	12	55	42	94	18	44	67	
i = 3 :	j = 4 .. 7	06	12	18	42	94	55	44	67	Vertauscht mit sich selbst !
i = 4 :	j = 5 .. 7	06	12	18	42	94	55	44	67	
i = 5 :	j = 6 .. 7	06	12	18	42	44	55	94	67	Vertauscht mit sich selbst !
i = 6 :	j = 7 .. 7	06	12	18	42	44	55	94	67	
Ende		06	12	18	42	44	55	67	94	Sortierte Liste

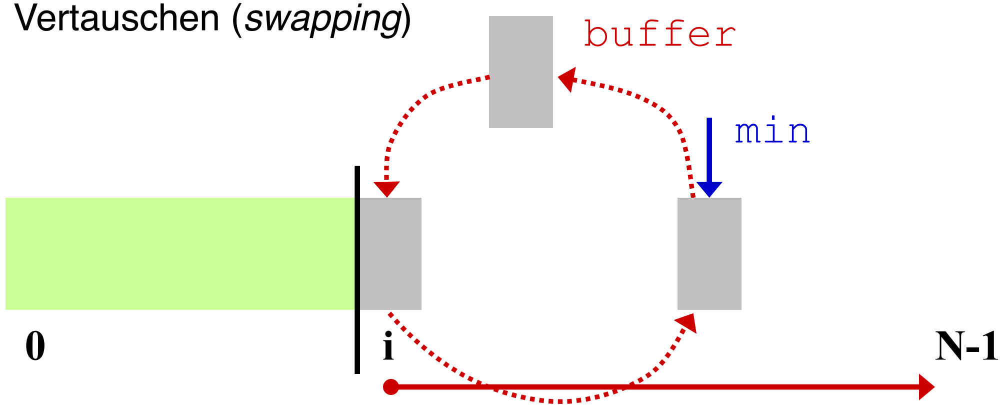
Implementierung in Java (Demo: AlgoSelectionSort.java)

```
public class AlgoSelectionSort {
      public static void main(String[] args) {
            final int N = 12;
            int[]     array = new int[N];

            for (int i = 0; i < N; i++)
                  array[i] = (int)(Math.random() * 10 * N);

            printArray(array);
            selectionSort(array);  // Aufruf 'selection sort'
            printArray(array);
      }

      public static void selectionSort(int[] arr) {
            int min;

            for (int i = 0; i < arr.length-1; i++) {  // Pos. des min. Elements
                  min = i;
                  for (int j = i+1; j < arr.length; j++) {
                        if (arr[j] < arr[min])
                              min = j;
                  }
                  swapElements(arr, i, min);  // vertausche Elemente
            }
      } // end selectionSort

      private static void swapElements(int[] arr, int pos, int minPos) {
            int buffer = arr[minPos];

            arr[minPos] = arr[pos];
            arr[pos]    = buffer;
      } // end swapElements

      private static void printArray(int[] arr) {
            final int len = arr.length;

            System.out.print("[");
            for (int i = 0; i < len-1; i++)
                  System.out.print(arr[i] + ", ");

            System.out.println(arr[len-1] + "]");
      } // end printArray
} // end class AlgoSelectionSort
```

- Suche die Position des kleinsten Elements im Bereich `[i...N-1]`
- Tausche dieses Element mit demjenigen an der Position i

Position des kleinsten Elements im unsortierten Teil, `min(arr[i...N-1])`

Vertauschen (*swapping*)



Zeitlicher Verlauf von *Selection sort* für eine zufällige Testfolge

- Visualisierung des Ablaufs der Sortierung: *http://www.sorting-algorithms.com*

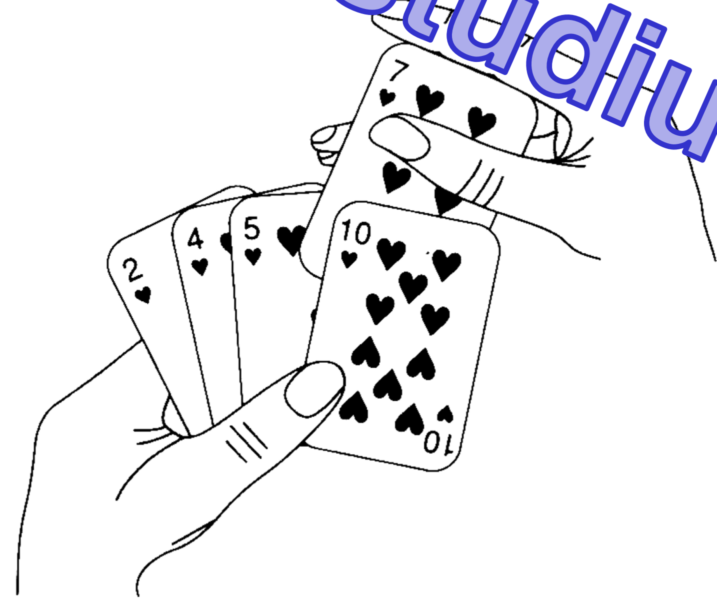
- Andere Online-Visualisierung: *https://algorithm-visualizer.org/*

Selbststudium

Insertion sort **– Sortieren durch direktes Einfügen**

Methode

- **Sortieren durch direktes Einfügen**
  - Effizientes Verfahren zur Sortierung einer kleinen Anzahl von Elementen
  - Analog zum Einfügen von Karten in einem Kartenspiel (Skat, Bridge, Rommé, etc.)



- **Strategie**

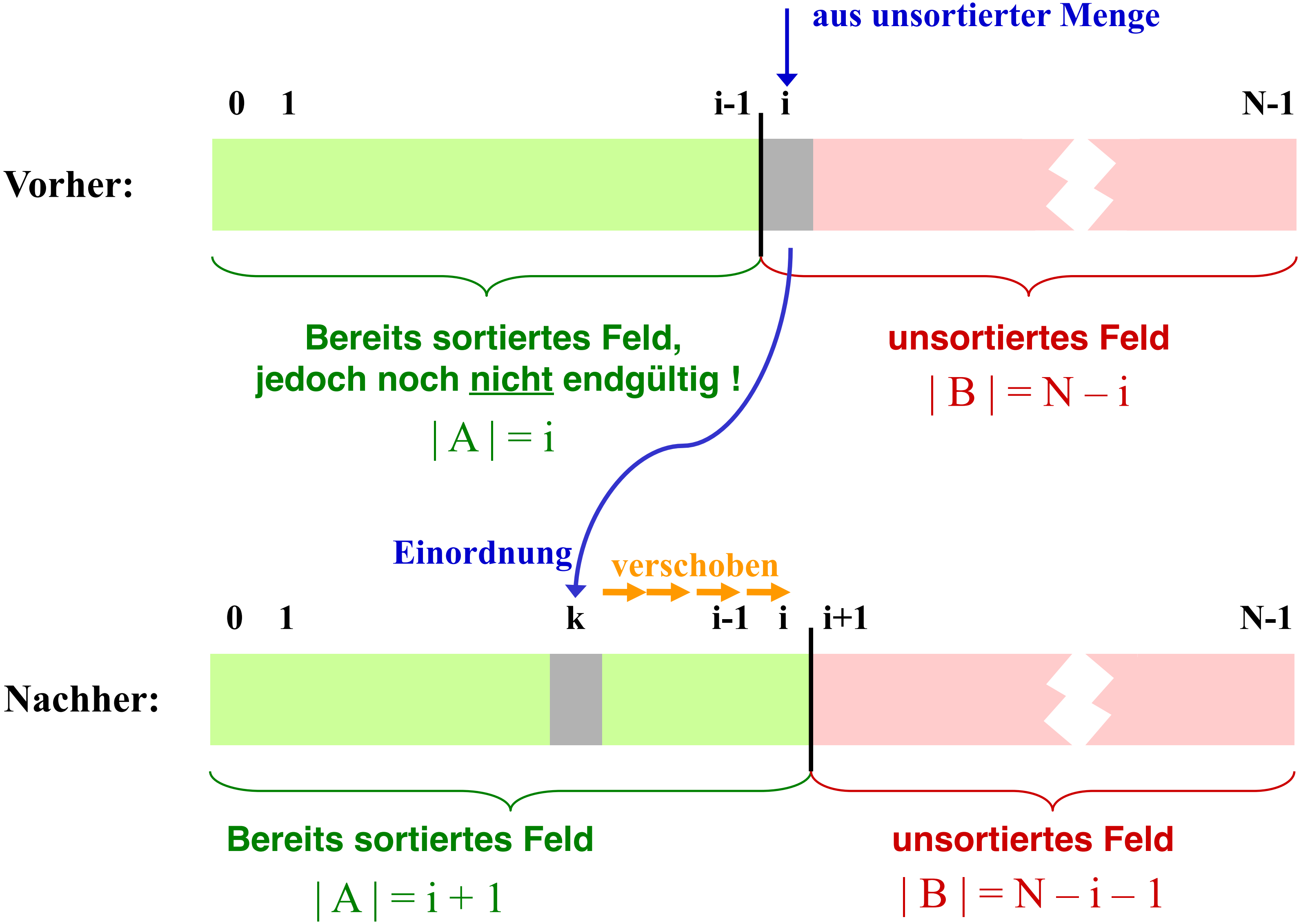
  Die Elemente (Karten) werden begrifflich aufgeteilt in eine
  - (sortierte) Ziel-Sequenz `A ≡ arr[0]..arr[i-1]` mit |A| = i
  - (unsortierte) Quellen-Sequenz `B ≡ arr[i]..arr[N-1]` mit |B| = N – i

Hinweis: Anzahl der Elemente = N;
`i` ist variabel: `[0 .. N-1]` (Beginn: `i = 0`, am Ende: `i = N-1`)

(T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to algorithms. MIT Press, 1990)

Selbststudium

Schema des Algorithmus



Selbststudium

Start

Beispiel

Beispiel	44	55	12	42	94	18	06	67
pos = 1 : i = 0 .. 0	44	55	12	42	94	18	06	67
pos = 2 : i = 1 .. 0	44	55	12	42	94	18	06	67
pos = 3 : i = 2 .. 0	12	44	55	42	94	18	06	67
pos = 4 : i = 3 .. 0	12	42	44	55	94	18	06	67
pos = 5 : i = 4 .. 0	12	42	44	55	94	18	06	67
pos = 6 : i = 5 .. 0	12	18	42	44	55	94	06	67
pos = 7 : i = 6 .. 0	06	12	18	42	44	55	94	67
Ende	06	12	18	42	44	55	67	94

Sortierte Liste

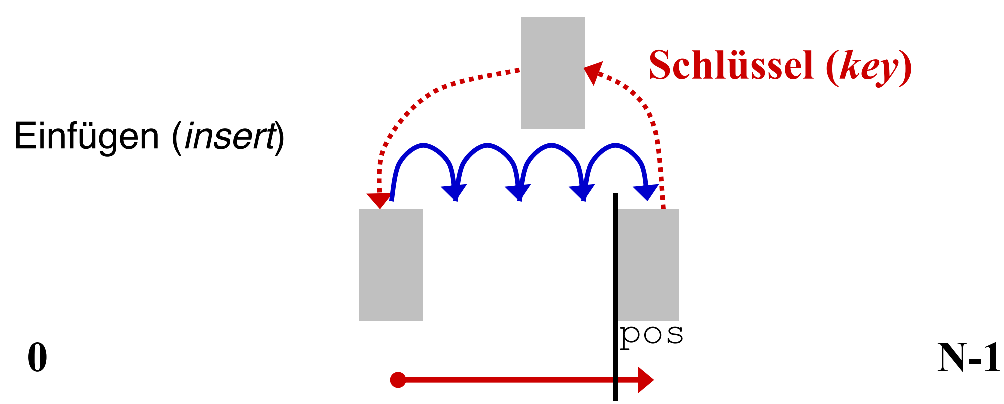
Selbststudium

Implementierung in Java (Demo: AlgoInsertionSort.java)

```
public class AlgoInsertionSort {
    public static void main(String[] args) {
        ...

        insertionSort(array);  // Aufruf 'insertion s
        ...
    }

    public static void insertionSort(int[] arr) {
        int pos,
            key;

        for (int i = 0; i < arr.length; i++) {  // bestimme Pos. des min. Elements
            pos = i - 1;
            key = arr[pos+1];  // aktuelles Element, das einsortiert werden soll
            while (pos >= 0 && arr[pos] > key) {
                arr[pos+1] = arr[pos];  // schiebe arr-Element 1 Pos. nach rechts
                pos--;
            }
            arr[pos+1] = key;
        }
    }  // end insertionSort

    ...
}  // end class AlgoInsertionSort
```

- Unterer teil-sortierter Bereich `[0...i]`
- Betrachte Element `arr[pos = i+1]` und durchlaufe Bereich von oben nach unten
- Bestimme die Position, an der das Element in den Bereich `[0...i+1]` einsortiert werden kann und verschiebe die größeren Elemente um 1 Position

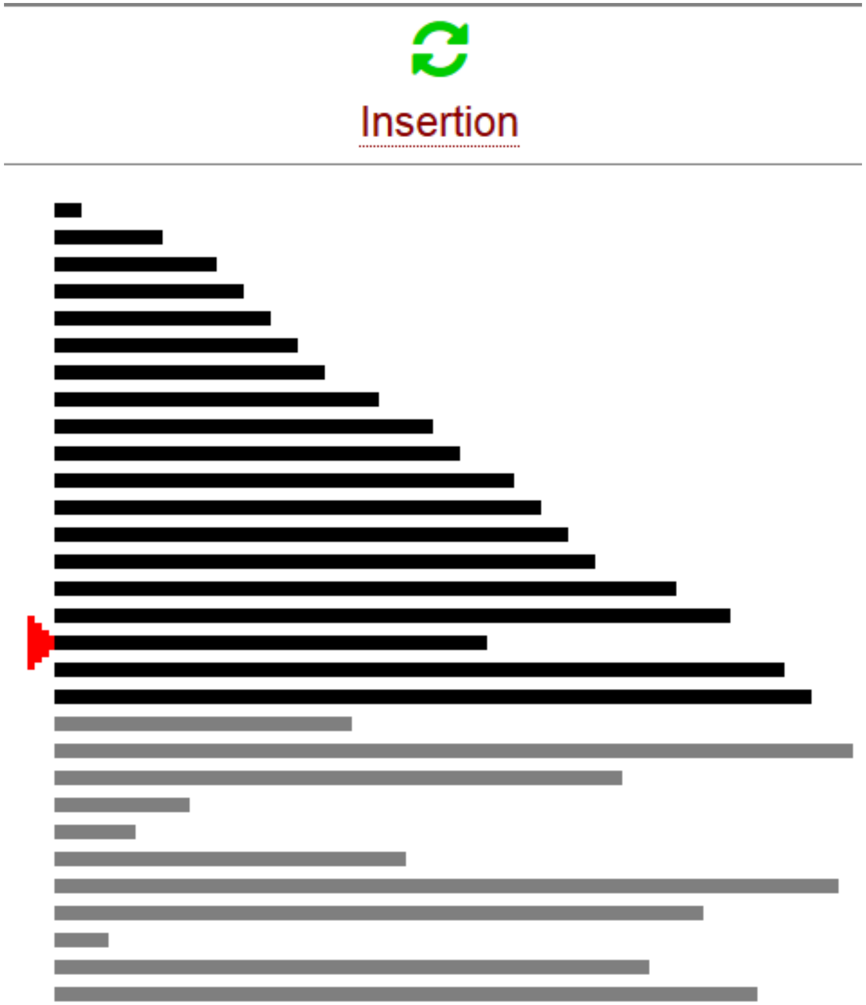
Start mit i = 0 ist redundant, da 1. Element bereits sortiert ist



Selbststudium

Zeitlicher Verlauf von *Insertion sort* für eine zufällige Testfolge

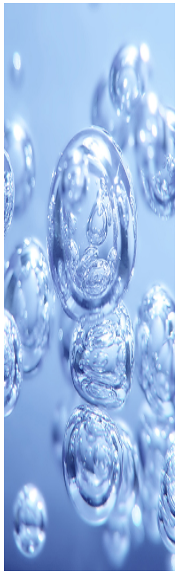
- Visualisierung des Ablaufs der Sortierung: *http://www.sorting-algorithms.com*

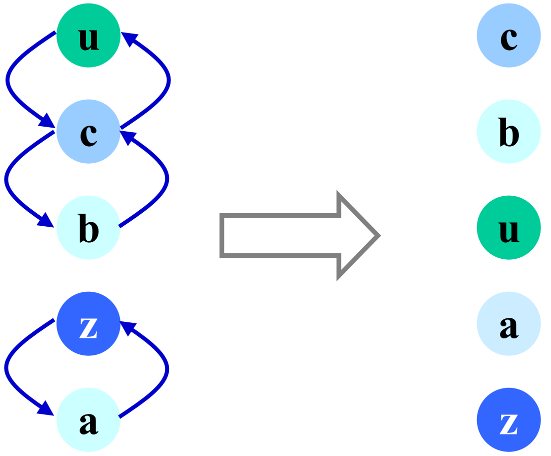


Bubble sort – Sortieren durch direktes Austauschen

Methode

- Sortierprinzip des fortgesetzten Vergleichs und ggf. Austausches von Paaren nebeneinander liegender Elemente
- Vorgehensweise
  - Die Elemente können als „Blasen“ (*bubbles*) aufgefasst werden, die bei jedem Durchlauf durch das Feld entsprechend ihres Gewichts (Schlüssel) höher aufsteigen bzw. tiefer sinken; falls eine schwerere Blase direkt über einer leichteren liegt, erfolgt beim Durchlauf ein Platzwechsel
  - In einem Durchlauf **von oben nach unten** können mehrere Blasen (nacheinander) absinken und leichtere nach oben verdrängen; dabei landet beim ersten Durchlauf die schwerste Blase ganz unten, beim zweiten Durchlauf die zweit-schwerste direkt über der schwersten, usw.

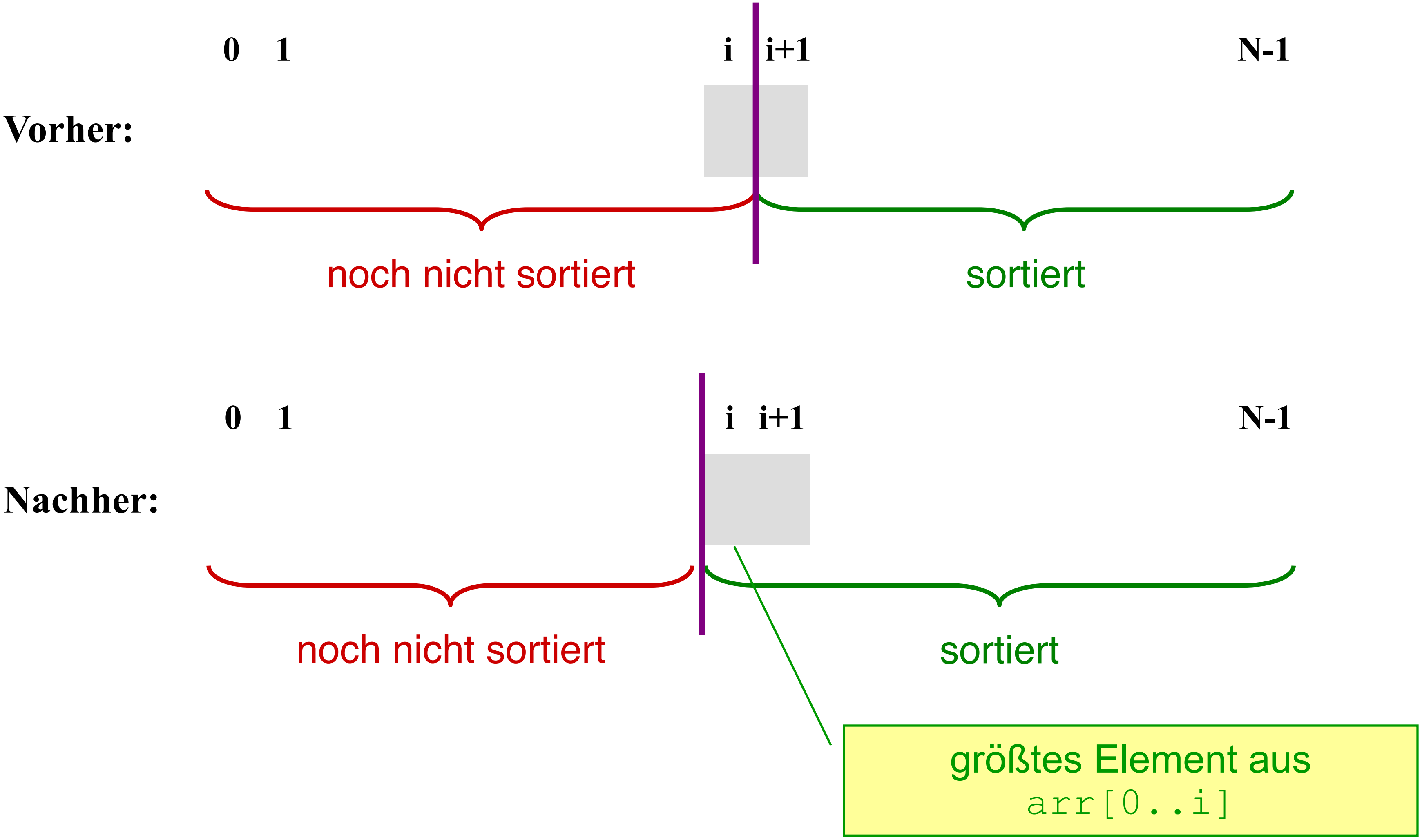




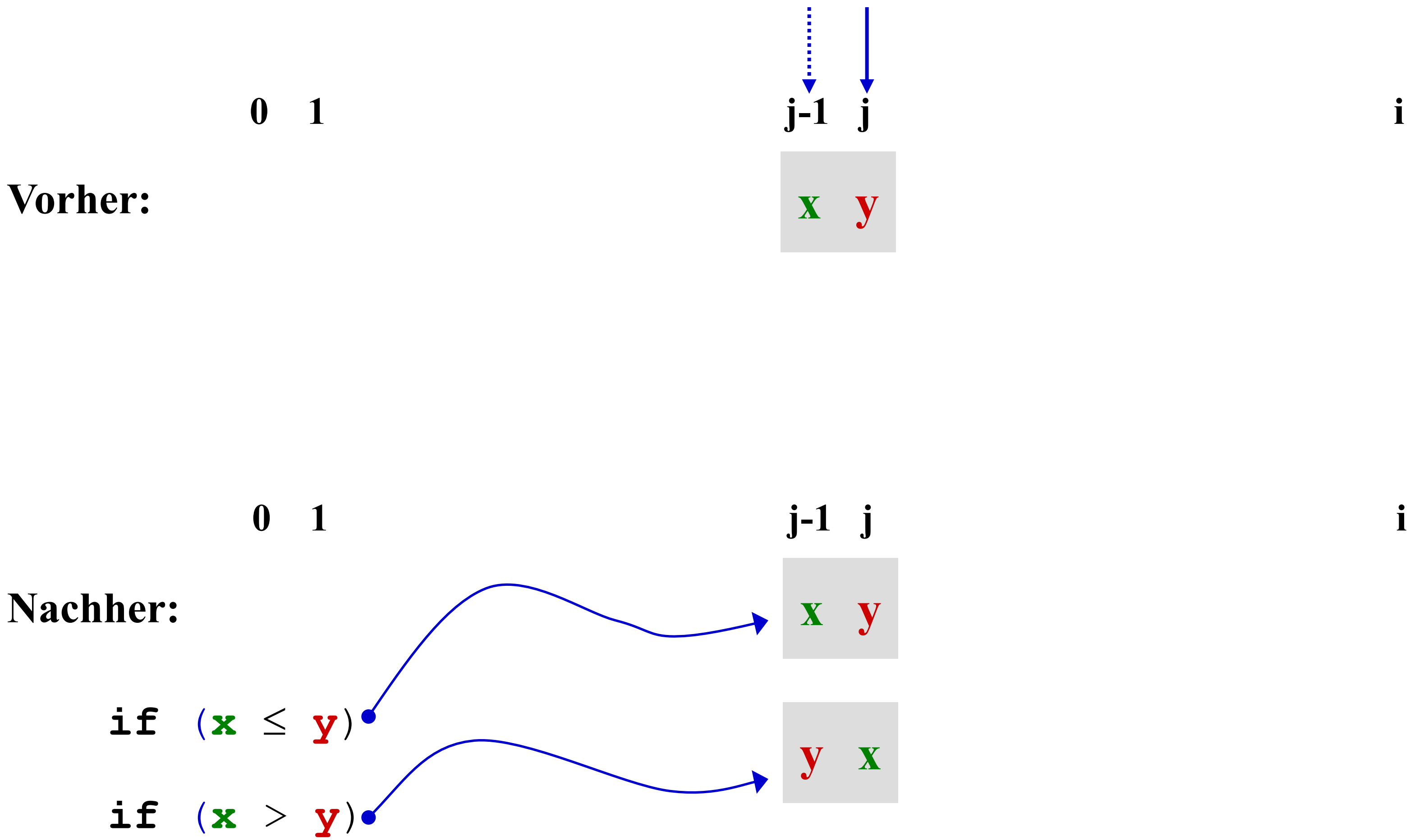
Beispiel: 1. Durchlauf von oben nach unten: u sinkt um 2 Positionen ab, bleibt vor (über) dem schwereren z hängen; dann tauscht z die Position mit dem leichteren a; z landet als schwerste Blase ganz unten

Schema des Algorithmus

Äußere Schleife (Laufindex `i: N-1 ... 1`)



Innere Schleife (Laufindex `j: 1 ... i`)



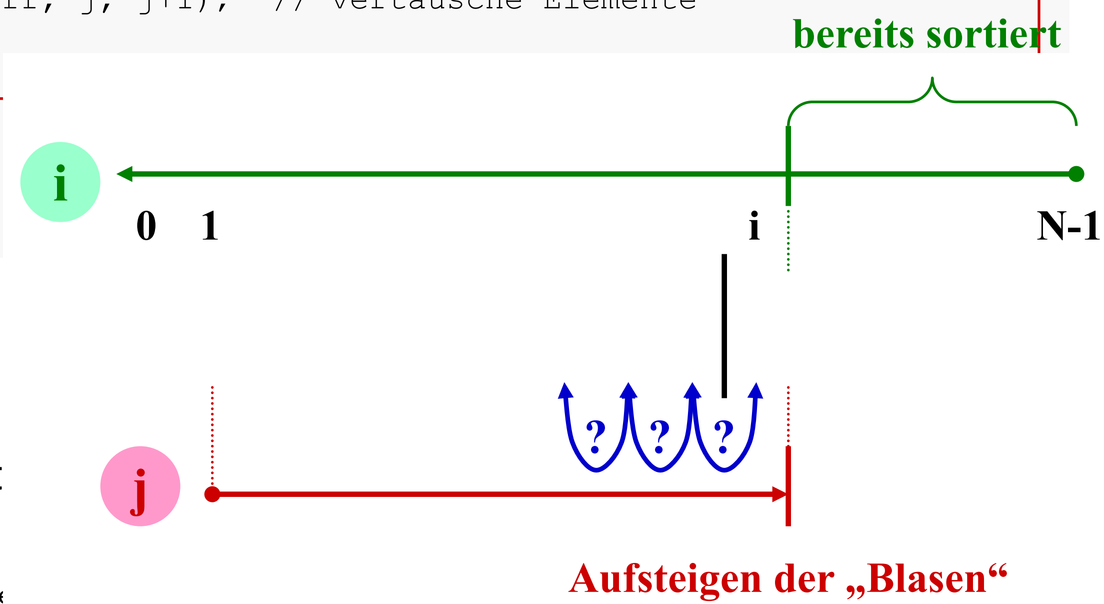
Beispiel

		44	55	12	42	94	18	06	67	Start
i = 7 :	j = 1 .. 7	44	12	42	55	18	06	67	94	
i = 6 :	j = 1 .. 6	12	42	44	18	06	55	67	94	
i = 5 :	j = 1 .. 5	12	42	18	06	44	55	67	94	
i = 4 :	j = 1 .. 4	12	18	06	42	44	55	67	94	
i = 3 :	j = 1 .. 3	12	06	18	42	44	55	67	94	
i = 2 :	j = 1 .. 2	06	12	18	42	44	55	67	94	
i = 1 :	j = 1 .. 1	06	12	18	42	44	55	67	94	
Ende		06	12	18	42	44	55	67	94	Sortierte Liste

Implementierung in Java – Sortiermethode (Demo: AlgoBubbleSort.java)

```
public class AlgoBubbleSort {
    public static void main(String[] args) {
        ...

        bubbleSort(array);  // Aufruf 'bubble sort'
        ...
    }

    public static void bubbleSort(int[] arr) {
        int min;

        for (int i = arr.length-1; i >= 1; i--) {
            for (int j = 0; j < i; j++) {  // Elemente [i..arr.length-1] sind sortiert
                if (arr[j] > arr[j+1])
                    swapElements(arr, j, j+1);  // vertausche Elemente
            }
        }
    }  // end bubbleSort

    ...
}  // end class AlgoBubbleSort
```

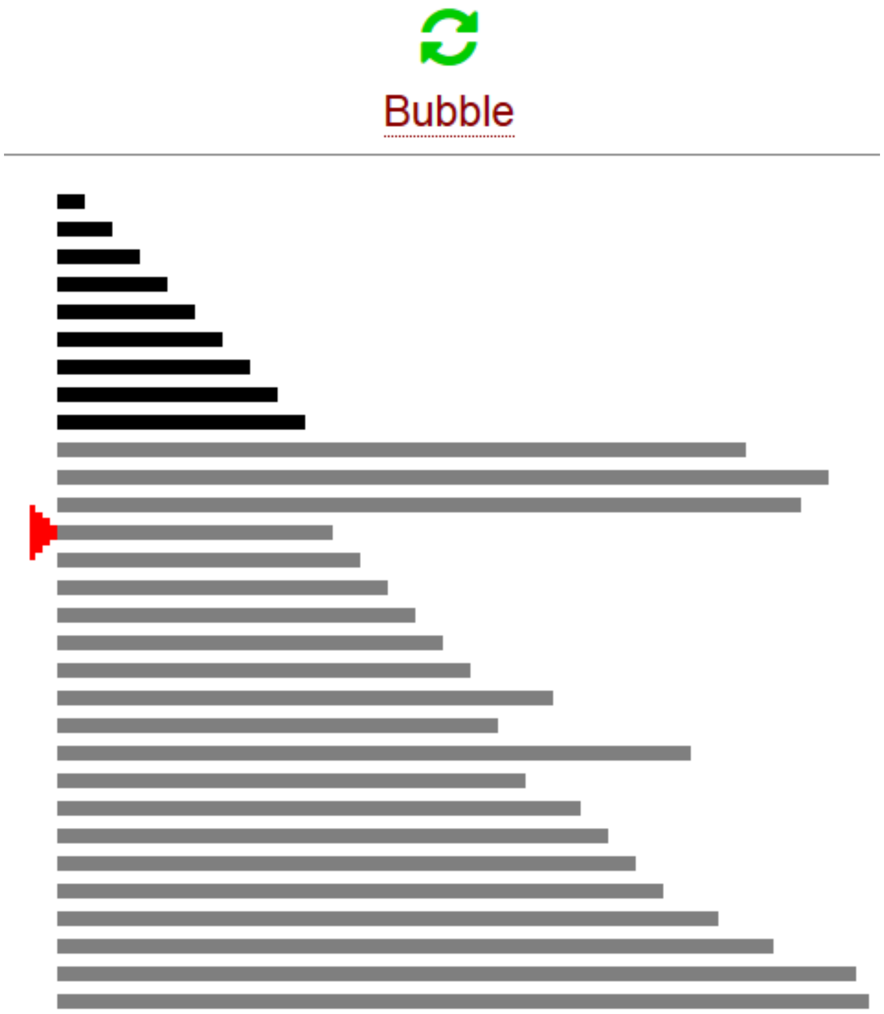
- Die Sortierung beginnt an der höchsten Position des Arrays `arr`
- In jedem Durchlauf i wird das jeweils größte Element aus der Menge `arr[j = 0..i]` an das obere Ende von `arr` verschoben
- Damit sind im Durchlauf i alle Elemente `arr[i..length-1]` sortiert



Hinweis: Unterschied zu *Selection Sort* in der Reihenfolge „von oben nach unten“; hier wird ebenfalls die `swapElements` Methode verwendet

Zeitlicher Verlauf von *Bubble sort* für eine zufällige Testfolge

- Visualisierung des Ablaufs der Sortierung: *http://www.sorting-algorithms.com*



Verbesserungen zu *Bubble sort*

- **Termination**: Man merkt sich, ob es während des Durchlaufs eine Vertauschung gegeben hat

  keine Vertauschung $\Rightarrow$ **Termination**

- **Symmetrie**:

  Bisher: Asymmetrie im Aufsteigen der „Blasen“ im schwereren / leichteren Ende des Feldes

  Bsp.: Abwechselnde Sortierrichtung …

1. Durchlauf

12	18	42	44	55	76	94	06
12	18	42	44	55	76	06	94

Im Standardverfahren würde bei jedem (weiteren) Durchlauf die 06-Blase in diesem Fall nur um (jeweils) eine Position (nach links) aufsteigen !

2. Durchlauf im *Shaker sort*

12	18	42	44	55	76	06	94
06	12	18	42	44	55	76	94

Idee für Lösungsansatz:

Änderung der Tausch-Richtung aufeinander folgender Durchläufe !

Aufsteigend – absteigend – aufsteigend – etc.

Methode: *Shaker sort*

(N. Wirth. Algorithmus und Datenstrukturen mit Modula-2. Teubner, 1983, p.87)

Aufwand für die einfachen iterativen Sortierverfahren

- *Selection sort* – allgemeine Betrachtung der notwendigen Zahl von Operationen

  Struktur:
  - das Feld `arr` wird von 0 .. N-2 durchlaufen
  - jedes mal wird das Feld zur Bestimmung des min. Elements von der aktuellen Position bis zum Ende (N-1) durchlaufen

  Aufwand: $O(N^2)$

- *Insertion sort*

  Struktur:
  - das Feld `arr` wird von 1 .. N-1 durchlaufen
  - jedes mal wird von der aktuellen Position das Feld nach unten durchlaufen, um das aktuelle Element an der richtigen Stelle einzusortieren

  Aufwand: $O(N^2)$

- *Bubble sort*

  Struktur:
  - das Feld `arr` wird von N-1 .. 0 durchlaufen
  - jedes mal wird von 2 .. aktuelle Position das Feld durchlaufen und dabei ggf. zwei nebeneinander liegende Elemente miteinander vertauscht

  Aufwand: $O(N^2)$

3. Sortieren durch Teilen-und-Herrschen

- Einordnung und Motivation
- *Mergesort*
- Aufwand für *Mergesort*
- *Quicksort* – Pivot und Algorithmus
- Aufwand für *Quicksort*

Einordnung und Motivation

Defizite der einfachen Verfahren und Alternativen

- In den bisher untersuchten iterativen Verfahren (*selection* / *insertion* / *bubble* sort) wird jedes zu sortierende Element im Prinzip mit jedem anderen verglichen; daher benötigen die auf einfachen Vergleichsstrategien beruhenden Verfahren stets einen **Aufwand** von $O(N^2)$ (alle Methoden bestehen im Kern aus zwei verschachtelten Wiederholungsschleifen mit max. n Iterationen)

- **Alternative Vorgehensweise**: Anwendung des Prinzips „Teile-und-Herrsche“ (*divide-and-conquer*) auf Sortierprobleme

- Prinzip:
  - `arr` bereits sortiert → fertig
  - `arr` noch **nicht** sortiert → Anwendung der Lösungsstrategie

Wesentliche Teilaufgaben:
- Zerlegen (1)
- Zusammenfügen (3)

Lösungsansatz:

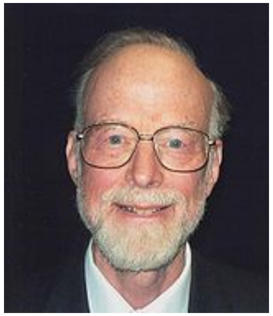
(1) `arr` in (nichtleere) Teile $\texttt{arr}_0$ .. $\texttt{arr}_i$ zerlegen ($i \geq 1$)

(2) $\texttt{arr}_0$ .. $\texttt{arr}_i$ (rekursiv) sortieren

(3) Sortierte $\texttt{arr}_0$ .. $\texttt{arr}_i$ zusammenfügen

Anwendung des *Divide-and-Conquer* Prinzips auf die Sortierung

- Nachfolgend werden zwei typische Vertreter der rekursiven *Divide-and-Conquer*-Sortiermethoden vorgestellt; beide Algorithmen teilen eine zu sortierende Folge in zwei Teilfolgen, die wiederum sortiert und zum Gesamtergebnis verknüpft werden
- Bezogen auf Sortierverfahren gibt es zwei Ausprägungen dieses Prinzips:
  - ***Hard split / easy join***:
    - Der Hauptaufwand liegt beim Teilen des Problems
    - Die Kombination der Teillösungen selbst ist trivial; <u>hier</u>: Unsortierte Folge F wird so in Teilfolgen F_1 und F_2 zerlegt, dass sich die sortierte Folge S aus F wie folgt ergibt $S = S_1\ S_2$ (S_k: Folge, die sich nach Sortierung von F_k ergibt, k = 1, 2)
    - Anwendung: ***Quicksort***-Algorithmus
  - ***Easy split / hard join***:
    - Die Aufteilung der zu sortierenden Folge F in Teilfolgen F_1 und F_2 ist trivial
    - Der Hauptaufwand liegt bei der Zusammensetzung von S aus S_1 und S_2
    - Anwendung: ***Mergesort***-Algorithmus



Quicksort – Pivot und Algorithmus

Idee des Verfahrens

- *Quicksort* (nach C.A.R. Hoare, 1962) ist ein *Divide-and-Conquer* Sortierverfahren der Ausprägung „*hard split / easy join*", d.h. der Hauptaufwand liegt in der Zerlegung der Daten in separat zu sortierende Teilfolgen
- Grundschema:
  - Teilen (*split*): Zerlegung der zu zerlegenden Folge F wird in zwei Teilfolgen $F_{<}$ und $F_{>=}$ aufgeteilt
    - Alle Elemente der einen Teilfolge $F_{<}$ sind kleiner als ein Referenzelement `key` (das sog. Pivot-Element)
    - Alle Elemente der Teilfolge $F_{>=}$ sind größer (oder gleich) dem Pivot-Element
  - Das Pivot-Element kann beliebig gewählt werden
    - Typischerweise wird das letzte Element der Folge verwendet
    - Es kann z.B. auch das erste oder mittlere Folgenelement gewählt werden
  - Sortieren: Die Teilfolgen werden jeweils durch rekursiven Aufruf von *Quicksort* sortiert; das Verfahren bricht für eine Teilfolge ab, wenn diese die Länge 0 hat

Realisierung des Teile-und-herrsche Prinzips

Grobe Skizze

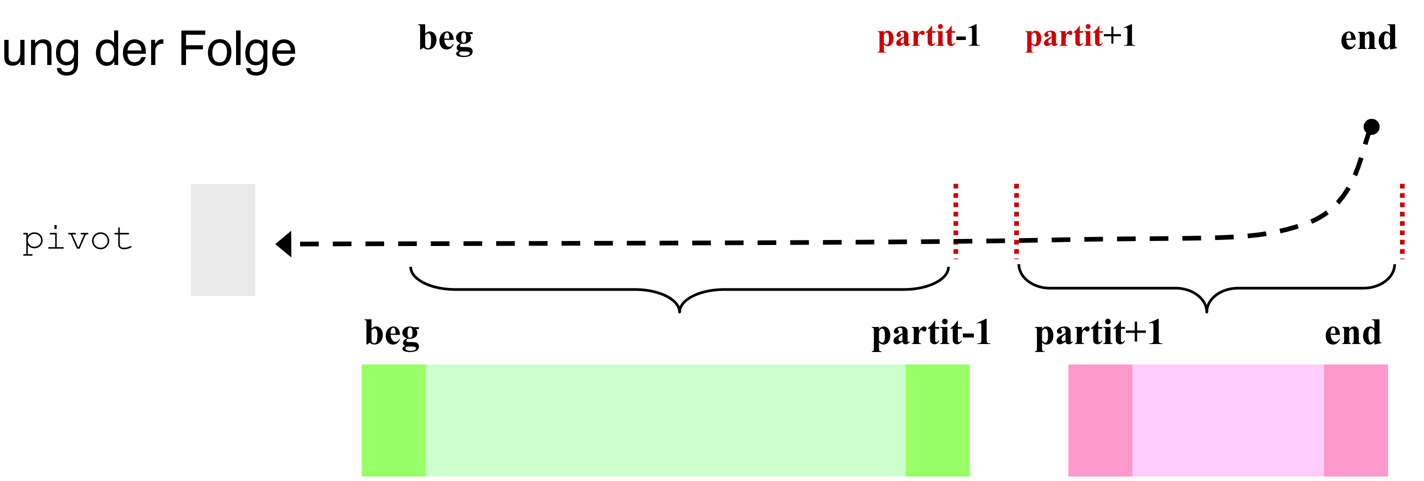
- Eingabe: Sei F die zu sortierende Folge
- Algorithmus:

> • Falls F leer ist oder nur ein Element enthält,
> `length(F) == 0 or length(F) == 1,`
> dann bleibt F unverändert
>
> • Sonst:
>
> - **Divide**: Wähle das Pivot-Element `pivot` von F und teile F in Teilfolgen $F_<$ und $F_{>=}$ auf; `pivot` wird in keine der beiden Teilfolgen eingefügt
> - **Conquer**: Sortiere $F_<$ mit *Quicksort*;
>   Sortiere $F_{>=}$ mit *Quicksort*
> - **Join**: Bilde die sortierte Folge F durch Hintereinanderhängen von
>
> $$F_< \ , \ \texttt{pivot} \ , \ F_{>=}$$

Algorithmus in Pseudocode (für ein *Array* mit der Folge)

```
procedure: quickSort(F, beg, end) → F
Eingabe: Zu sortierende Folge F,
         Indizes von unterer und oberer Grenze: beg und end
Ausgabe: Sortierte Folge F

   if beg < end then
      partit := partition(F, beg, end);
      quickSort(F, beg, partit-1);
      quickSort(F, partit+1, end);
   endif
   return F;
```

- Zerlegung der Folge



… so, dass gilt: $(\forall i \in [beg \ldots partit-1], j \in [(partit+1) \ldots end]) : arr[i] \leq arr[j]$

- Zerlegung der Folge in Pseudocode

```
procedure: partition(F, beg, end) →  partit
Eingabe: Zu zerlegende Folge F, untere und obere Grenze beg, end
Ausgabe: Position der Zerlegung, left

    pivot := F[end];
    left  := beg - 1;
    right := end;
    while (left < right)  // Vertauschen von Elementen, so dass gilt:
                          // arr[beg..pivot-1] < pivot & arr[pivot+1..end] >= pivot
        do
            left  := left + 1;
        while (F[left] < pivot);
        do
            right := right - 1;
        while (F[right] >= pivot and left < right);

        if (left < right) then
            swapElements(F[left], F[right]);
        endif
    endwhile

    swapElements(F[left], F[end]);  // vertausche Folgenelement mit Pivot

    return left;
```

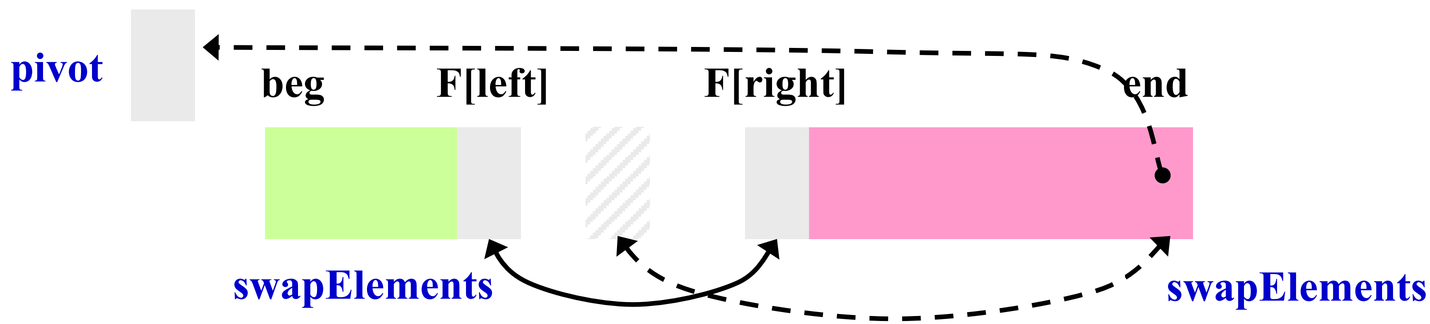
Anmerkung: Die Elemente in den Teilfolgen sind nicht geordnet

- Details zur Aufteilen der Daten

Initialisierung: Folge `F` mit Grenzen `beg`, `end`
Indexvariable `left := beg`
Indexvariable `right := end - 1`

Algorithmus zur Aufteilung:

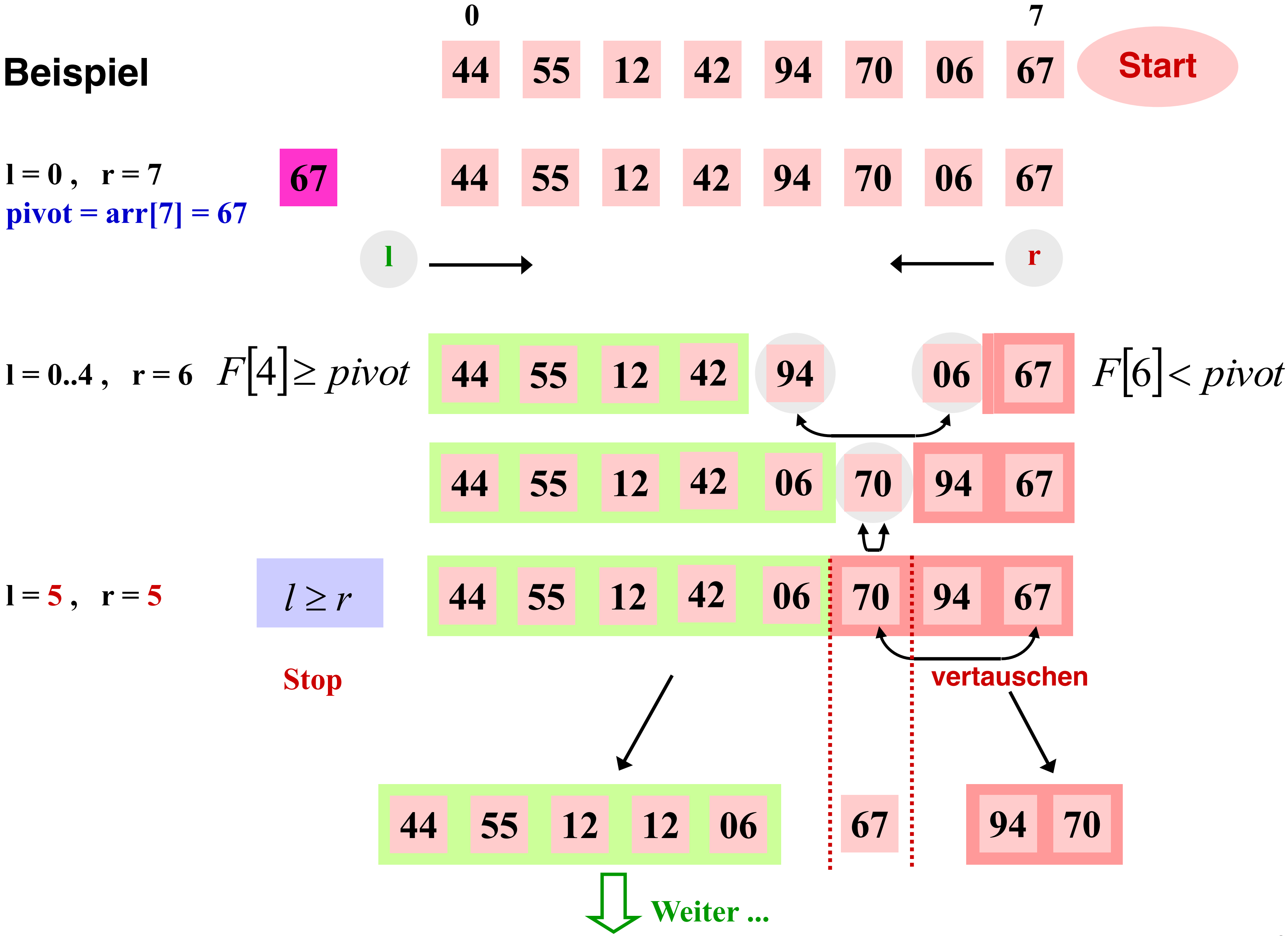
- Auswahl des Pivot-Elements **`pivot`** (dessen Wert legt die Teilungsgrenze fest)
- Solange `F[left] < pivot`, erhöhe Index `left`
- Dann: solange `F[right] >= pivot`, erniedrige Index `right`
- Wenn Positionen gefunden wurden, für die die Bedingung nicht gilt, dann vertausche *Array*-Elemente `F[left]` und `F[right]`, so dass diese im richtigen Teilfeld liegen



- Wiederholung solange, bis Index `left` und `right` sich treffen, d.h. $\underbrace{\texttt{left} \geq \texttt{right}}$

Vertauschen des Elements am Schnittpunkt mit Pivot ⟵ Bedingung definiert den **Zerlegungspunkt**

Beispiel


0
7
44 55 12 42 94 70 06 67
Start
l = 0 , r = 7
pivot = arr[7] = 67
67
44 55 12 42 94 70 06 67
l
r
l = 0..4 , r = 6
$F[4] \geq pivot$
44 55 12 42 94 06 67
$F[6] < pivot$
44 55 12 42 06 70 94 67
l = 5 , r = 5
$l \geq r$
44 55 12 42 06 70 94 67
Stop
vertauschen
44 55 12 12 06
67
94 70
Weiter ...

Implementierung in Java (Demo: AlgoQuickSort.java)

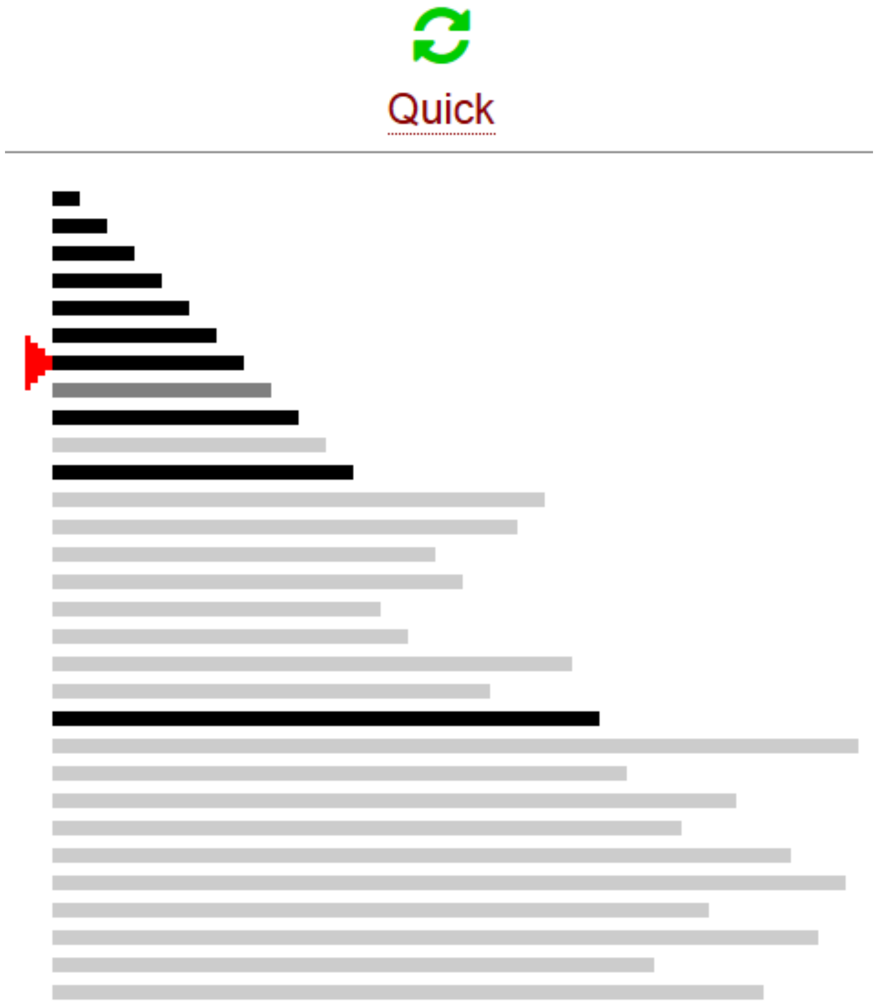
Es werden die Kern-Methoden vorgestellt, ohne weitere Hilfsmethoden zum Vertauschen von Elementen und der Ausgabe der Elemente

```
public class AlgoQuickSort {
    public static void main(String[] args) {
        final int N = 18;
        int[]     array = new int[N];

        for (int i = 0; i < N; i++)
            array[i] = (int)(Math.random() * 10 * N);

        printArray(array);
        quickSort(array);  // Aufruf 'quicksort'
        printArray(array);
    }

    public static void quickSort(int[] arr) {
        sort(arr, 0, arr.length-1);
    }

    static void sort(int[] arr, int start, int end) {
        if (start < end) {
            int divider = partition(arr, start, end);

            sort(arr, start, divider-1);  // Aufteilung ohne Pivot (à la Sedgewick)
            sort(arr, divider+1, end);
        }
    }

    // Fortsetzung ...
```

```
    // hier geht's weiter:

    static int partition(int[] arr, int beg, int end) {
        int pivot = arr[end],
            left  = beg - 1,
            right = end;

        while (left < right) {
            do {
                left++;
            } while (arr[left] < pivot);
            do {
                right--;
            } while (arr[right] >= pivot && left < right);

            if (left < right)
                swapElements(arr, left, right);
        }
        swapElements(arr, left, end);
        return left;
    }

    private static void swapElements(int[] arr, int low, int high) {...}

    private static void printArray(int[] arr) {...}

}
```

Zeitlicher Verlauf von *Quicksort* für eine zufällige Testfolge

- Visualisierung des Ablaufs der Sortierung: *http://www.sorting-algorithms.com*



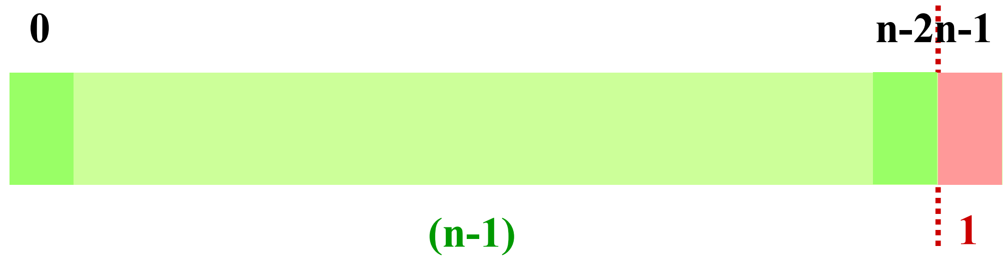
Aufwand für *Quicksort*

Laufzeiten bei einer „schlechten“ Zerlegung

- **Allgemeine Bewertung**: Die Laufzeit hängt davon ab, ob die Zerlegung des Felds **un**balanciert oder balanciert ist; letzteres hängt wiederum von dem Wert des Pivot-Elements `pivot` und damit dem resultierenden Zerlegungspunkt ab (**Hinweis**: Für die Analyse wird die Variante des Algorithmus‘ betrachtet, in der das Pivot-Element in der unteren Teilfolge enthalten ist)

- **Schlechte Zerlegung**

  Zerlegung des *Arrays*:



  Annahme: Bei jedem Schritt tritt eine unbalancierte Zerlegung auf, z.B. bei bereits vollständig sortierter Sequenz:

  - bei jeder Teilung entsteht ein 1-elementiges Teilfeld und
  - ein Teilfeld mit den restlichen Elementen

- Kosten: Die Zerlegung kostet O(n) und die Rekursion T(1) = O(1)

- Baum der rekursiven Aufrufe bei schlechter Zerlegung

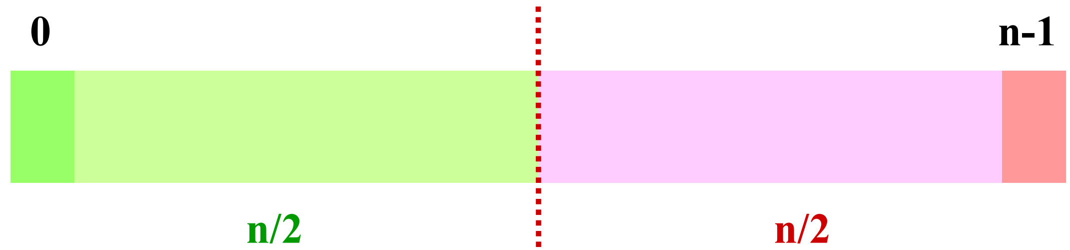
$$T(n) = T(n-1) + O(n) = \sum_{k=1}^{n} O(k) = O\left(\sum_{k=1}^{n} k\right) = O(n^2)$$

$$
\begin{array}{c|ccccccc|c}
 & & n & & & & & & n \\
 & 1 & & n-1 & & & & & n \\
 & & 1 & & n-2 & & & & n-1 \\
n & & & 1 & & n-3 & & & n-2 \\
 & & & & 1 & & & & \vdots \\
 & & & & & & 2 & & 3 \\
 & & & & & 1 & & 1 & 2 \\
 & & & & & & & & O(n^2)
\end{array}
$$

- **Ergebnis**: Die Zeitkomplexität ist im **ungünstigsten Fall** O(n^2) – dann, wenn die Eingabesequenz bereits vollständig sortiert ist

Laufzeiten der „besten“ Zerlegung (optimal balancierte Zerlegung)

- Zerlegung des *Arrays*:



- Baum der rekursiven Aufrufe bei balancierter Zerlegung

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \cdot \log_2 n)$$

$$n \qquad\qquad n$$

$$n/2 \quad n/2 \qquad\qquad n$$

$$n/4 \quad n/4 \quad n/4 \quad n/4 \qquad\qquad n$$

$$\log_2 n \qquad n/8 \quad n/8 \quad n/8 \quad n/8 \quad n/8 \quad n/8 \quad n/8 \quad n/8$$

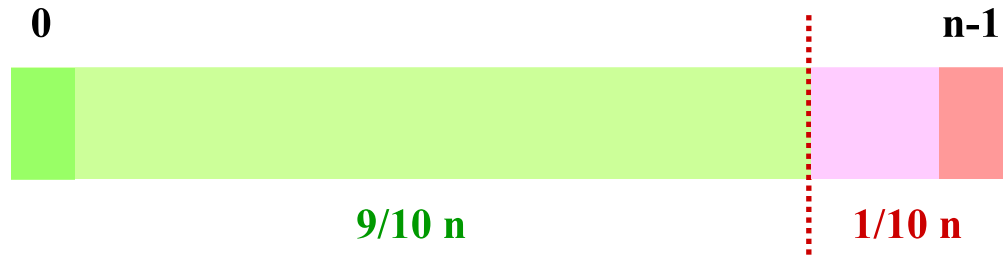
$$\vdots$$

$$1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \; 1 \qquad\qquad n$$

$$O(n \cdot \log_2 n)$$

Laufzeiten einer balancierten Zerlegung

- Beispiel-Zerlegung (9:1):



- Rekursion für die Ausführung von *Quicksort* bei balancierter Zerlegung

$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + O(n) = O(n \cdot \log_2 n)$$

Analyse und Erläuterungen:

- Für jede Ebene trägt der Zerlegungsbaum die Kosten n bei, bis die Blätter der Ebene $\log_{10} n = O(\log_2 n)$ erreicht sind
- Die weiteren Ebenen haben geringere Kosten als n; der Baum terminiert bei einer Tiefe von $\log_{10/9} n = O(\log_2 n)$

- Baum der rekursiven Aufrufe bei balancierter Zerlegung

n — n

$\frac{1}{10}n$ $\quad$ $\frac{9}{10}n$ — n

$\log_{10} n$

$\frac{1}{100}n$ $\quad$ $\frac{9}{100}n$ $\quad$ $\frac{9}{100}n$ $\quad$ $\frac{81}{100}n$ — n

$\log_{10/9} n$

1 $\quad$ $\frac{81}{1000}n$ $\quad$ $\frac{729}{1000}n$ — n

$\leq n$

1 — $\leq n$

$$O(n \cdot \log_2 n)$$

- **Ergebnis**: Der Aufwand ist $T(n) = O(n \cdot \log_2 n)$ (gilt z.B. auch für 99:1); damit ist im Mittel der Aufwand von ähnlicher Größenordnung wie für die beste Zerlegung

Selbststudium

Mergesort

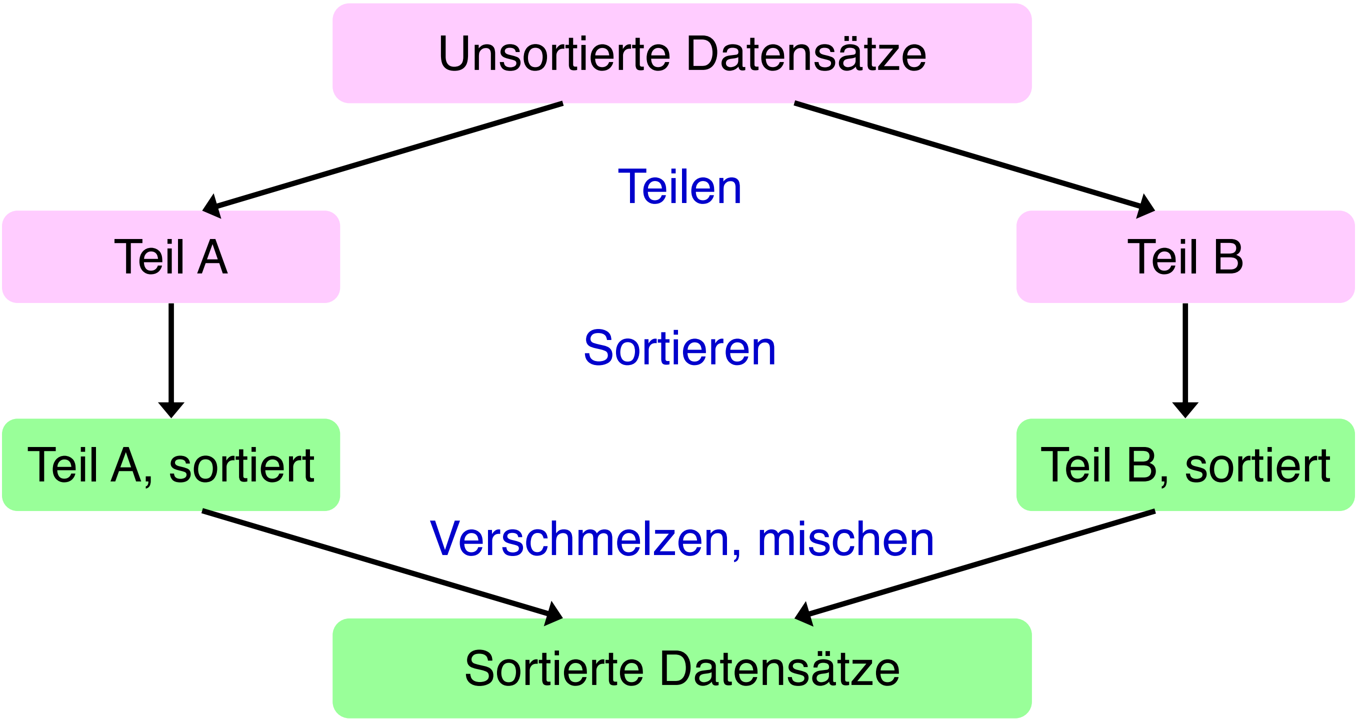
Idee des Verfahrens

- *Mergesort* ist ein *Divide-and-Conquer* Sortierverfahren der Ausprägung „*easy split / hard join*“, d.h. der Hauptaufwand liegt in der Zusammenführung der Resultate der einzeln sortierten Teilfolgen

- Grundschema:
  - Teilen (*split*): Die zu sortierende Folge F wird in zwei etwa gleich große Teilfolgen F_1 und F_2 aufgeteilt
  - Sortieren: Diese Teilfolgen werden durch rekursive Anwendung von *Mergesort* sortiert und anschließend gemischt; man gelangt zu sortierten Teilfolgen, bis die zu sortierenden Teilfolgen nur noch aus einem Element bestehen (das anschließende Mischen liefert dann die sortierte Teilfolge)
  - Verschmelzen (*merge*, *join*): Die sortierten Teilfelder werden zusammen gemischt

Selbststudium

- Schematischer Ablauf für die Zusammenführung (*merge*, *join*):



- Mischen (*join*) der sortierten Teilfolgen (Pseudocode)
  - Es wird zunächst vereinfachend angenommen, dass Folgen ein einfaches Anhängen bzw. Entfernen von Elementen aus der Menge erlauben
  - Dies muss dann für die konkrete Realisierung, z.B. auf *Arrays*, in Form von Einfüge- und Lösch-Operationen umgesetzt werden

Selbststudium

Pseudocode:

```
procedure: merge(F₁, F₂) → F
Eingabe: Zwei zu mischende (sortierte) Teilfolgen F₁, F₂
Ausgabe: Sortierte Folge

    F := leere Folge;
    while F₁ nicht leer and F₂ nicht leer:
        Entferne das kleinere der beiden Anfangselemente aus F₁, F₂;
        Fuege dieses Element an F an (d.h. an das Ende von F);
    endwhile
    Fuege die verbliebene nichtleere Teilfolge F₁ oder F₂ an F an;
    return F;
```

- Rekursive Realisierung der eigentlichen Sortier-Operation (Pseudocode)

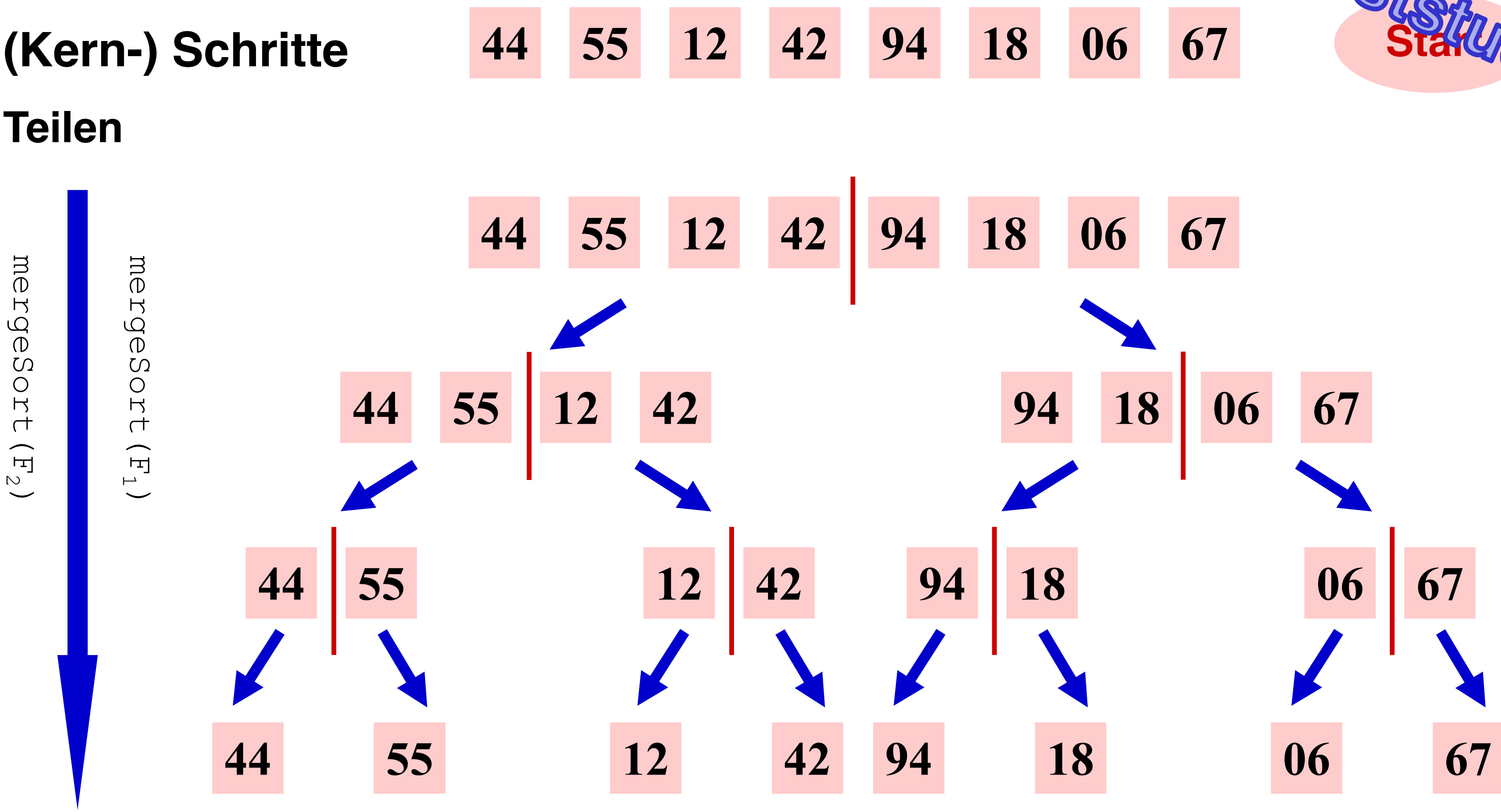
```
procedure: mergeSort(F) → S
Eingabe: Eine zu sortierende Folge F
Ausgabe: Sortierte Folge S

    if length(F) = 1 then                              Basisfall
        return F;
    else
        Teile F in F₁ und F₂;                          Divide
        S₁ := mergeSort(F₁);   // Sortierung von F₁    Conquer
        S₂ := mergeSort(F₂);   // Sortierung von F₂
        return merge(S₁, S₂);  // mischen              Join
    endif
```

Selbststudium

Start

(Kern-) Schritte

Teilen

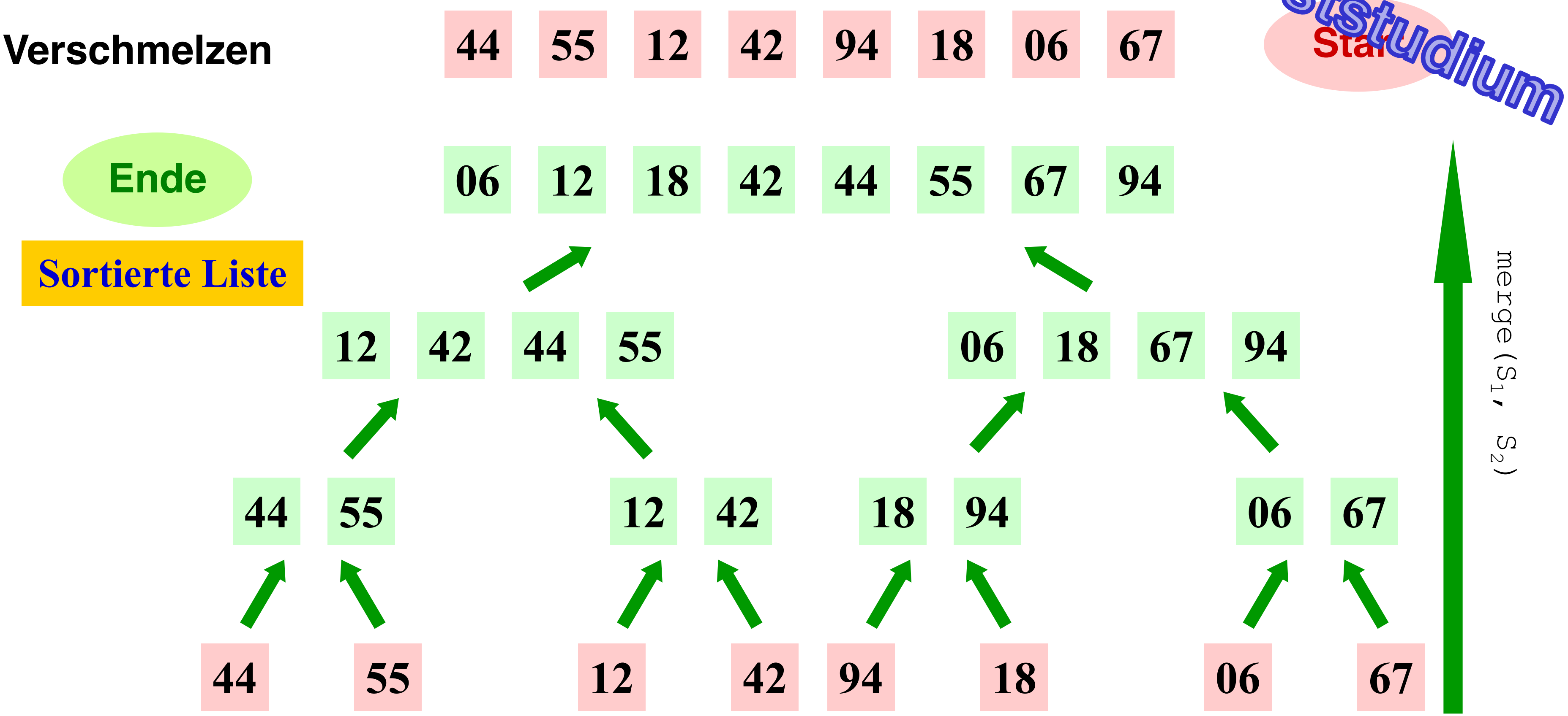


Ergebnis der Zerlegung der Daten

Anmerkung: Die Zerlegung erfolgt stets durch Halbierung der Datenmenge (Rekursionsbaum) bis auf die unterste Ebene mit nur jeweils einem Datenelement

Selbststudium

Verschmelzen



Zur Methode `merge(...)`

- Die Folgen S_1 und S_2 sind bereits sortiert
- Abhängig von der Repräsentation der Daten benötigt das Mischen der Folgen (auf jeder Stufe) jeweils eine **Folge zur Zwischenspeicherung** des sortierten Abschnitts und anschließender Ersetzung der Folge – dies erfordert einen Aufwand von **O(n)**

Selbststudium

Implementierung basierend auf *Arrays* in Java

- Für die *Array*-basierte Implementierung des *Mergesort*-Verfahrens sind die im Pseudo-code verwendeten Operationen wie “Entfernen” und “Anfügen” (S.57) nicht gut geeignet
- Während das Zerlegen noch auf einfache Weise durch Zeiger auf den Anfang (`beg`) und das Ende (`end`) des jeweiligen Teilfeldes realisiert werden kann, erfordert das Mischen ein **Hilfsfeld**, in das die neu sortierte Folge eingetragen wird
- Implementierung in Java (Demo: AlgoMergeSort.java)

```
public class AlgoMergeSort {
     public static void main(String[] args) {
          final int N = 18;
          int[]     array = new int[N];

          for (int i = 0; i < N; i++)
               array[i] = (int)(Math.random() * 10 * N);

          printArray(array);
          mergeSort(array);  // Aufruf 'mergesort'
          printArray(array);
     }

     public static void mergeSort(int[] arr) {
          sort(arr, 0, arr.length-1);
     }

     static void sort(int[] arr, int start, int end) {
          if (start < end) {
               int middle = (start + end) / 2;

               sort(arr, start, middle);
               sort(arr, middle+1, end);
               merge(arr, start, middle, end);
          }
     }
```

Selbststudium

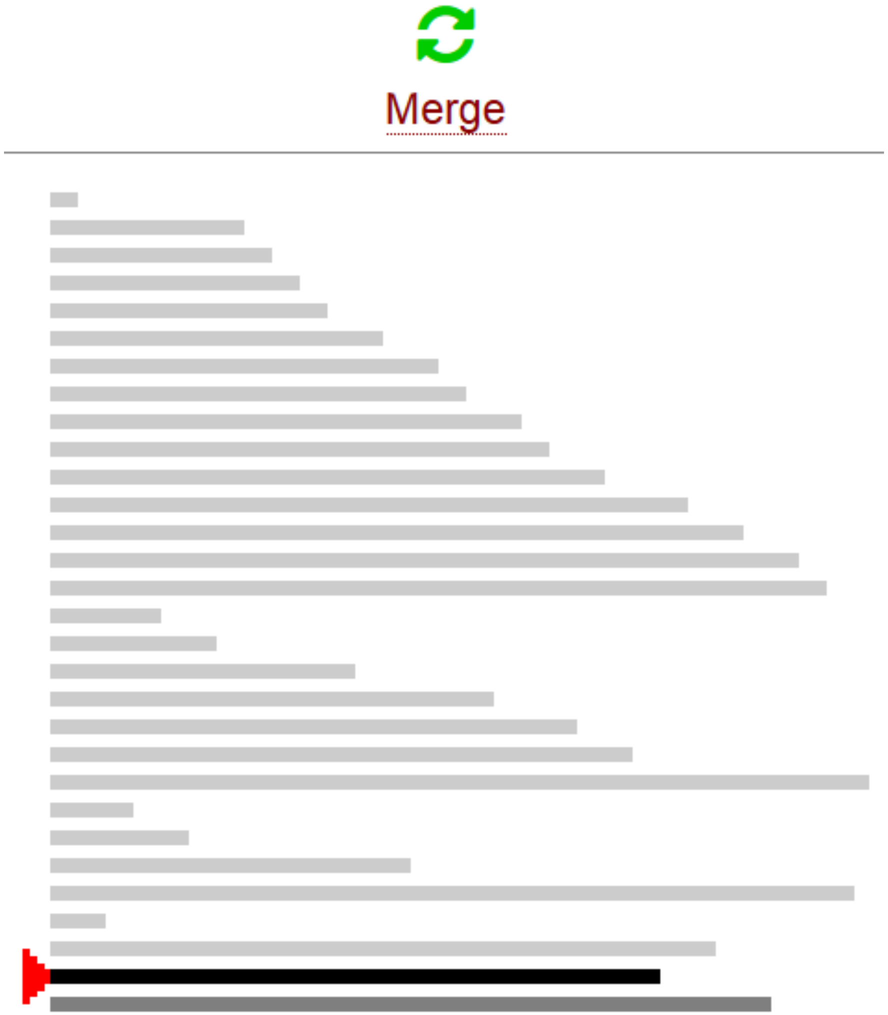
- Mischen der Teilfolgen („*hard join*“)

```
    static void merge(int[] arr, int start, int middle, int end) {
        if (end - start >= 1) {
            int   a = 0,
                  b = 0,
                  na = middle - start + 1,
                  nb = end - middle;
            int[] buf = new int[na+nb];

            while (a < na && b < nb) {
                if (arr[start+a] <= arr[middle+1+b]) {
                    buf[a+b] = arr[start+a];
                    a++;
                }
                else {
                    buf[a+b] = arr[middle+1+b];
                    b++;
                }
            }
            while (a < na) {
                buf[a+b] = arr[start+a];
                a++;
            }
            while (b < nb) {
                buf[a+b] = arr[middle+1+b];
                b++;
            }

            for (int i = 0; i < na+nb; ++i) {
                arr[start+i] = buf[i];
            }
        }
    }

    ...
}
```

Selbststudium

Zeitlicher Verlauf von *Mergesort* für eine zufällige Testfolge

- Visualisierung des Ablaufs der Sortierung: *http://www.sorting-algorithms.com*



4. Sortieren mit Halde (*Heap sort*)

- Prinzip der Sortierung mit *Heap sort*
- Struktur des *Heaps* und seine Eigenschaften
- Entwicklung eines Algorithmus
- Aufwand von *Heap sort* und Gesamtbewertung

Prinzip der Sortierung mit *Heap sort*

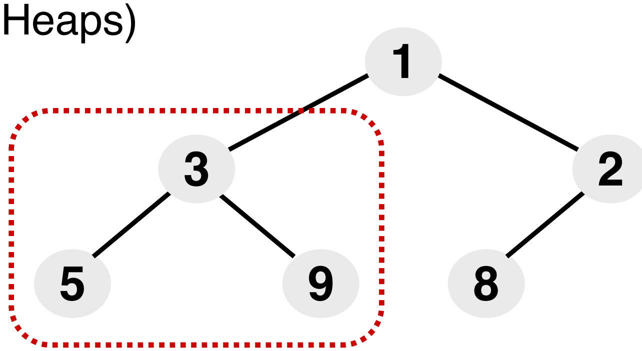
Motivation und Eigenschaften von *Heap sort*

- *Heap sort* ist ein Sortierverfahren, das durch Verwendung einer Baumstruktur als Repräsentation effizient realisiert werden kann
- Grundlegend ist die Datenstruktur ***Heap*** (Halde), die einen binären Baum mit bestimmten Eigenschaften bezeichnet
  - Der Baum ist **vollständig**, d.h. die Blattebene ist von links nach rechts gefüllt **!**
  - Der Schlüssel eines jeden Knotens ist bei einem sog. *Min-Heap* kleiner oder gleich dem Schlüssel seiner Kinder, d.h.

    `node.item ≤ node.left.item & node.item ≤ node.right.item;`

    diese partielle Ordnung wird als ***Heap*-Eigenschaft** bezeichnet
    (*Max-Heap* ist analog definiert, mit „größer oder gleich“ statt „kleiner oder gleich“; wir betrachten Min-Heaps)

$3 \leq 5$ **&** $3 \leq 9$

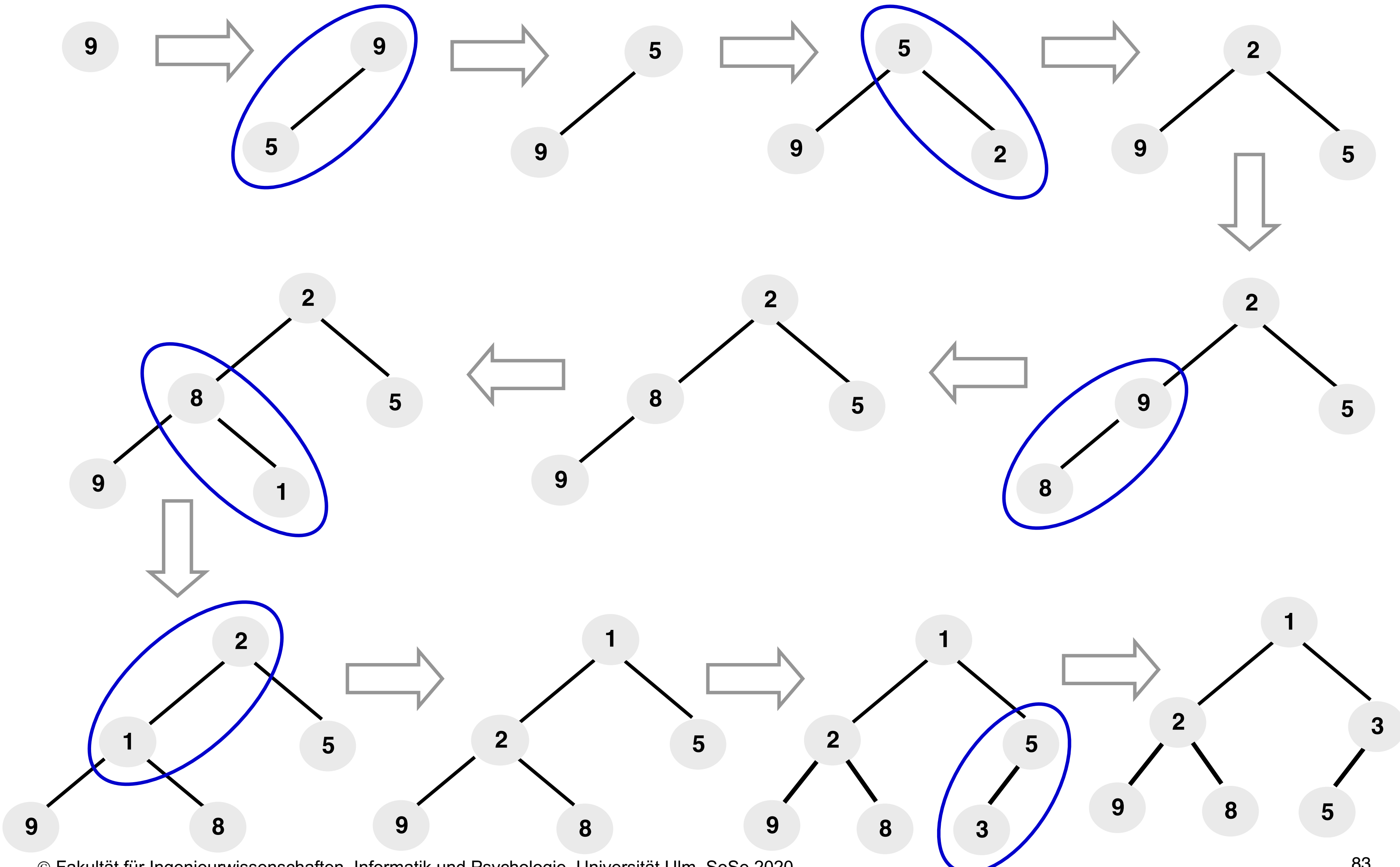


Aufbau (Füllen) eines Heaps

Regeln:

- Es wird stets immer erst eine Ebene vollständig von links nach rechts gefüllt, bevor mit dem Füllen der nächsten Ebene (wieder von links nach rechts) begonnen wird;
  d.h. das neue Element wird zunächst „am Ende“ des Baumes eingefügt.
- Das neu eingefügte Element wird dann mit seinem Vaterknoten verglichen: Ist $\text{Wert}_{\text{Kindknoten}} < \text{Wert}_{\text{Vaterknoten}}$, werden Kind- und Vaterknoten vertauscht.
- Dieser Vergleichs- und Tauschvorgang wird solange von unten nach oben fortgesetzt, bis entweder die Bedingung nicht mehr gilt oder der neue Knoten zum Wurzelknoten gemacht wurde.

Eingabe: 9, 5, 2, 8, 1, 3



Löschen des Wurzelelements eines Heaps

Regeln:

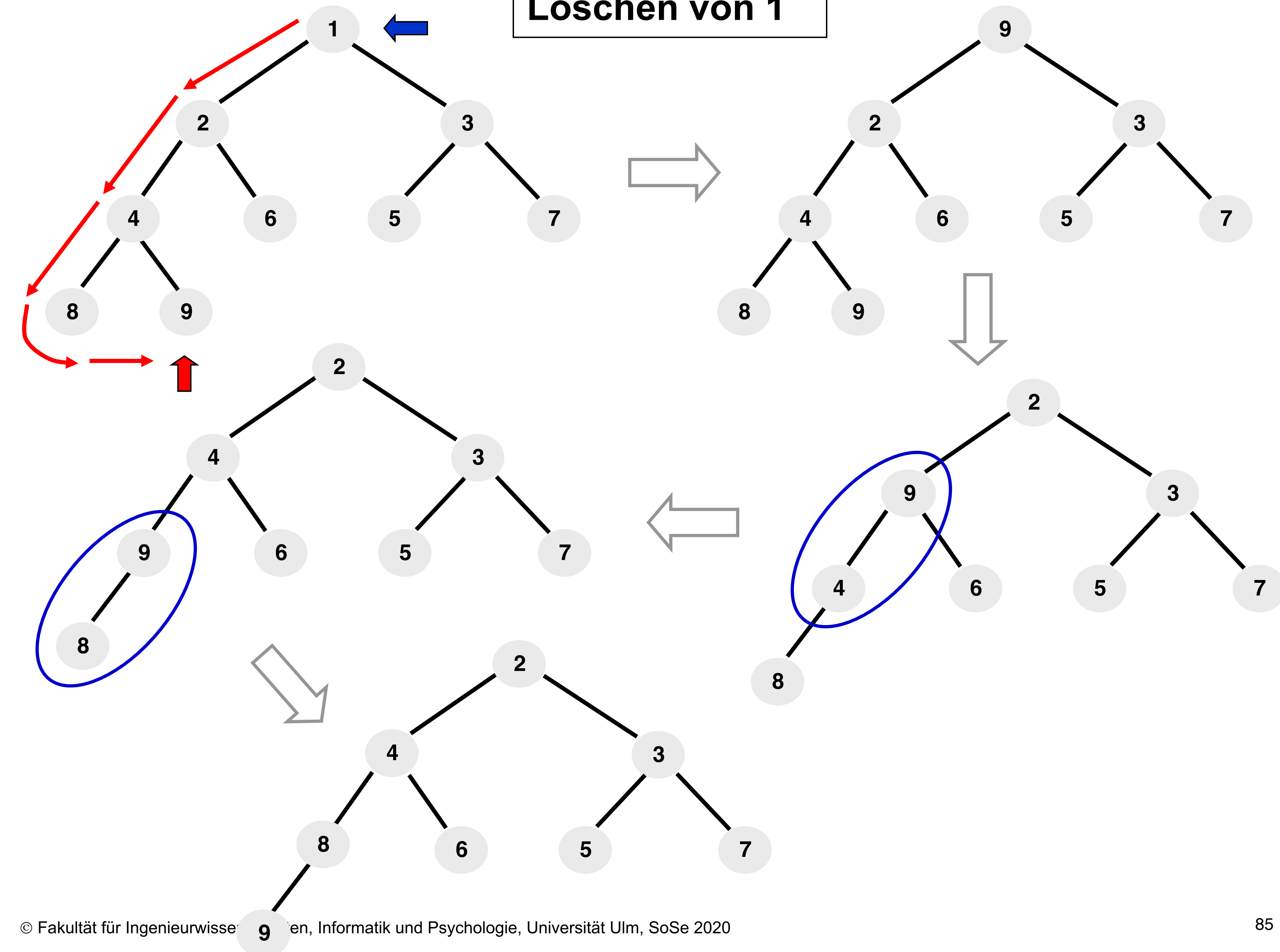
- Das Schrumpfen eines Heaps ist das Gegenstück zum Wachsen …,
  er schrumpft immer von hinten (von „rechts unten").

- Das Wurzelelement wird gelöscht, in dem es durch das letzte Element (E_{Ersatz}) auf Blattebene ersetzt wird (Blattebene, „ganz rechts").

- Falls E_{Ersatz} größer ist als mind. eines seiner Kindknoten, „versickert" es (durch Vertauschen, wie vorher) in Richtung des (jeweils) kleineren Kindknotens.

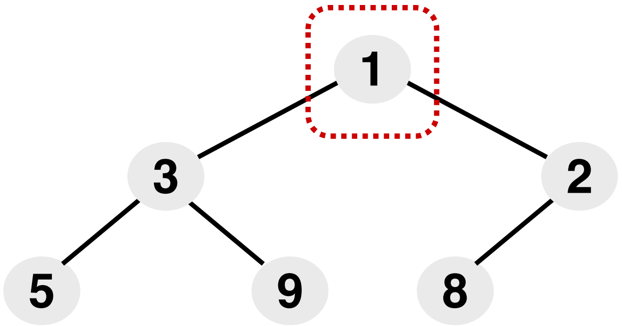
- Der Vergleichs- und Tauschvorgang wird solange fortgesetzt, bis die Bedingung nicht mehr gilt oder die Blattebene erreicht wurde.

Löschen von 1



Verwendung und Eigenschaften

- *Heap*-Strukturen sind geeignet, wenn keine vollständige Sortierung aller Elemente festgelegt ist, sondern nur jeweils das kleinste Element (bzw. das größte bei einem Max-Heap) ausgewählt werden soll



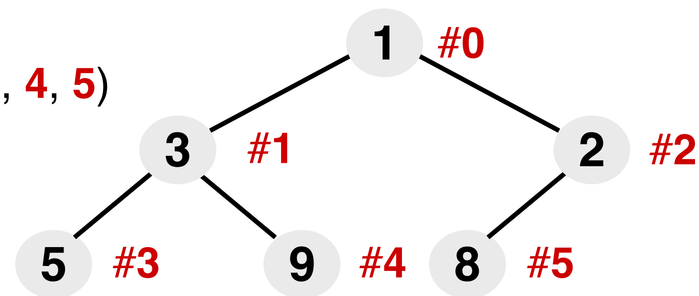
- Das kleinste Element der Folge befindet sich immer in der Wurzel

- **<u>Anwendung</u>**: Prioritätswarteschlangen (Betriebssysteme), bei denen immer das Element mit der höchsten Priorität entnommen wird, während neue Elemente eingefügt werden

Struktur des *Heaps* und seine Eigenschaften

Realisierung

- Die *Heap*-Struktur kann in der Implementierung durch ein *Array* repräsentiert werden und so effiziente Tausch-Operationen für die Elemente ermöglichen
- Aufbau:
  - Wurzel (= Position **0**)
  - Kinder der Wurzel (= Positionen **1** und **2**)
  - Knoten der nächsten Ebene (Positionen **3**, **4**, **5**)
- Nummeriert man die Knoten des *Heaps* in *Level order*, so werden …
  - die Kinder des *k*-ten Knotens auf den Positionen 2*k*+1 (linkes Kind) und 2*k*+2 (rechtes Kind) abgelegt und
  - der Eltern-Knoten eines Knotens k ist an der Position $\lceil k/2 \rceil$ - 1 zu finden



Index	0	1	2	3	4	5
Schlüssel-Elemente	1	3	2	5	9	8

- Da der *Heap* ein vollständiger Baum ist, ergeben sich keine Lücken im *Array*
- Durch die Darstellung als Baum können Elementfolgen sortiert werden, ohne dass zusätzliche Datenstrukturen aufgebaut werden

Heap-Eigenschaft und Entfernen des Wurzelelements

- Ein *Array* `arr[0, ..., n-1]` erfüllt die ***Heap*-Eigenschaft**, wenn gilt:

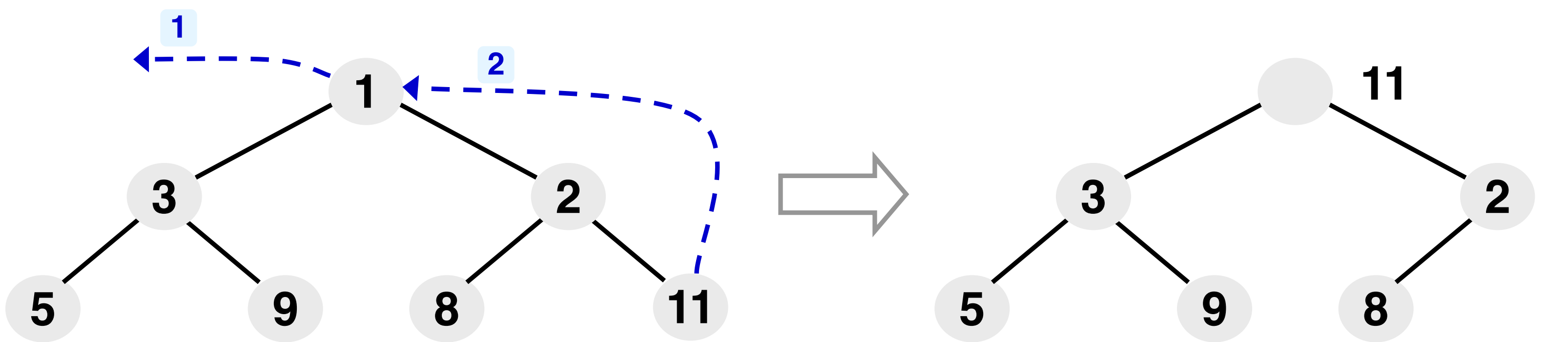
  `arr[k]` $\leq$ `arr[2k+1]` und `arr[k]` $\leq$ `arr[2k+2]` , $\forall\ k \geq 0 \wedge 2k+2 < n$

- In einem *Heap* enthält die Wurzel immer das kleinste Element; man kann die Elemente des *Heaps* in sortierter Reihenfolge auslesen, indem jeweils das Wurzelelement aus dem Baum entnommen wird

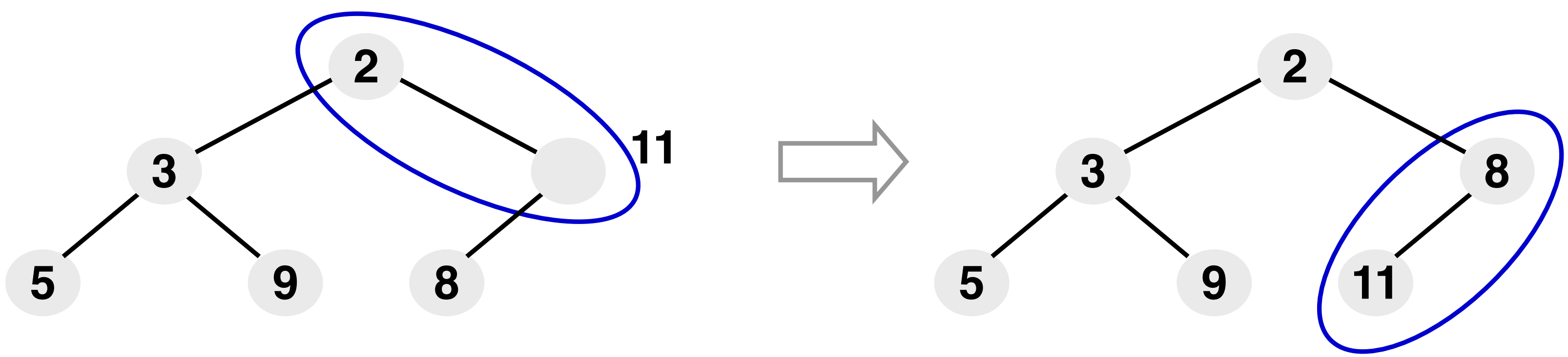
  Problem: Durch Entfernen der Wurzel (mit dem kleinsten Element) entsteht ein „Loch“ im Baum – wodurch die *Heap*-Eigenschaft verletzt wird

- **Lösung**: Wiederherstellung der *Heap*-Eigenschaft
  - Entfernen des sich im Baum am weitesten rechts-unten befindlichen Objekts (hier: Position 5 mit Schlüsselelement/Wert 8)
  - Kopieren des Schlüsselelements dieses Knotens (hier: 8) in den **Wurzelknoten**
  - Anschließendes **„Versickern“ dieses Elements** im Baum – das Element wird solange nach unten bewegt, bis die *Heap*-Eigenschaft wieder erfüllt wird

- Ablauf – Wiederherstellung der *Heap*-Eigenschaft nach Entfernung der Wurzel
  - Entfernen des Elements 0 führt zu einem „Loch“ und Bewegen des letzten Elements zur Wurzel



  - Vergleich mit Nachfolgeelementen und „versickern“ falls ein Nachfolger kleiner ist: das Element wird mit dem kleinsten Nachfolgeelement vertauscht, damit die Heap-Eigenschaft an diesem Knoten erfüllt ist. Ein weiteres Versickern ist u. U. nötig.



Heap-Eigenschaft ist wieder erfüllt

Entwicklung eines Algorithmus

Grundidee für eine Realisierung mit *Arrays*

- **Grundidee**:
  - „Entfernen" des aktuell kleinsten Elements aus dem *Heap* und anschließende Wiederherstellung der *Heap*-Eigenschaft mittels „Versickern"
  - Dieser Schritt wird für den verbleibenden Rest-*Heap* solange wiederholt, bis alle Elemente in sortierter Reihenfolge vorliegen
- Für eine Verwendung im Rahmen eines Sortierverfahrens sollte dies ohne zusätzlichen Speicherbedarf realisiert werden
- Wir nutzen dazu aus, dass ein *Heap* durch ein ***Array*** repräsentiert werden kann: Das *Array*-Element 0 entspricht dem aktuell kleinsten *Heap*-Element (Wurzel), während das letzte *Array*-Element jeweils dem letzten *Heap*-Element entspricht
- Wir vertauschen jeweils das erste mit dem aktuell letzten *Array*-Element und stellen für den um 1 verkleinerten Rest-*Heap* die *Heap*-Eigenschaft mittels Versickern wieder her

  Beachte: Die bereits richtig sortierten Elemente werden am Ende des *Arrays* (in den zur Repräsentation des jeweiligen Rest-*Heaps* nicht mehr benötigten Elementen) eingeordnet

Algorithmus und seine Implementierung

Ablauf mit elementaren Phasen

- Pseudocode

```
procedure: heapSort(F) →  F_H
Eingabe: zu sortierende Folge F
Ausgabe: Permutation von F, absteigend sortiert

   Überführe F in einen Heap;

   len := length(F);
   while len > 1:
      Vertausche F[0] und F[len - 1];  // swap(Wurzel, letztes)
      Versickere F[0] im Rest-Heap F[0, ..., len - 2];
      Dekrementiere len;  // letztes Element entfernt
   endwhile
   return F;
```

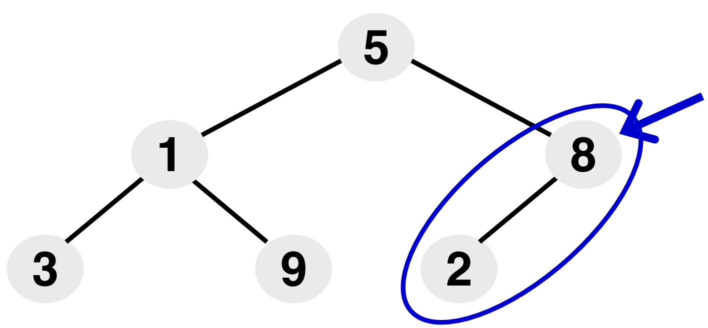
Phase 1

Phase 2

- **Ergebnis**: Die sortierte Folge ist nach Beendigung im *Array* F (in umkehrter Reihenfolge) gespeichert

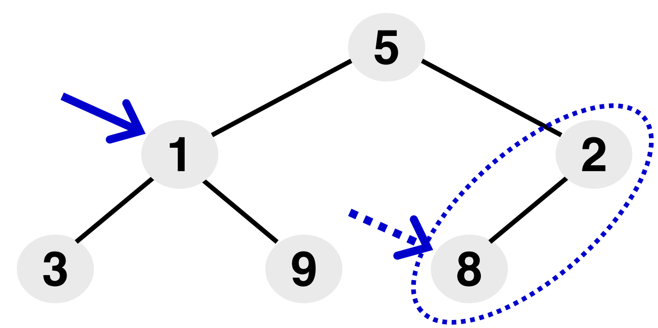
Phase 1: Aufbau eines *Heaps*

- In Phase 1 des Algorithmus wird das *Array* in eine *Heap*-Struktur überführt
- Hierzu wird das Prinzip des Versickerns von Elementen angewendet:
  - Beginnend beim letzten Element, wird versucht, dieses weiter im Baum nach unten zu bewegen. Dann schrittweise vorherige Elemente prüfen.
  - Aufwandsreduzierung: Es kann ausgenutzt werden, dass die Blätter des Baumes keine Kinder haben und deshalb nicht betrachtet werden müssen: Für ein Feld mit n Elementen müssen so nur die ersten n/2 Elemente berücksichtigt werden
- Beispiel-*Array* und die Überführung in eine *Heap*-Struktur
  - Erstes betrachtetes Element: 8; *Heap*-Eigenschaft verletzt



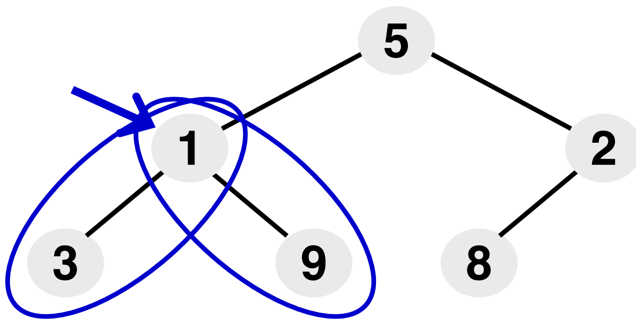
5	1	$\boxed{8}$	3	9	2

  - *Heap* nach Versickern von 8; nächstes betrachtetes Element ist 1



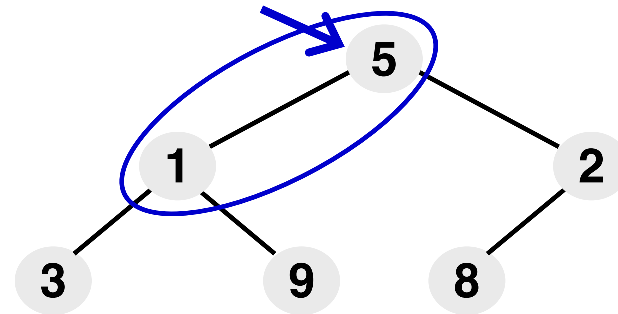
5	$\boxed{1}$	2	3	9	8

- Für Element 1 ist nichts zu tun, da *Heap*-Eigenschaft erhalten ist



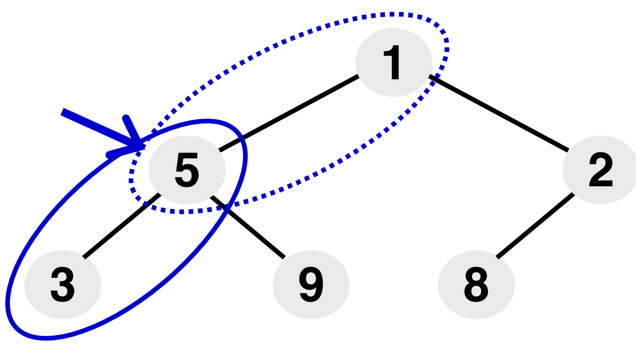
5	1	2	3	9	8

- Betrachtetes Element ist 5; *Heap*-Eigenschaft ist verletzt



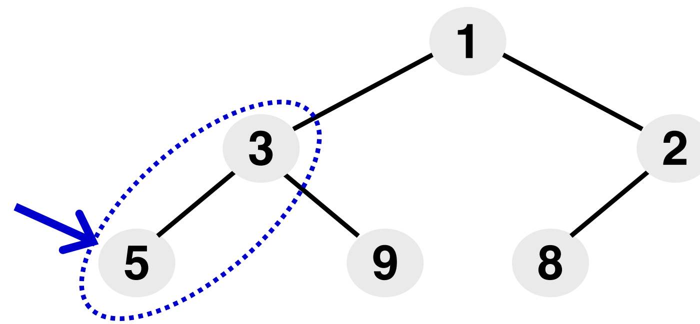
5	1	2	3	9	8

- Versickern von Element 5 (1. Schritt): vertausche 5 und 1; *Heap*-Eigenschaft ist weiterhin verletzt



1	5	2	3	9	8

- Versickern von Element 5 (2. Schritt); Vertausche 5 und 3

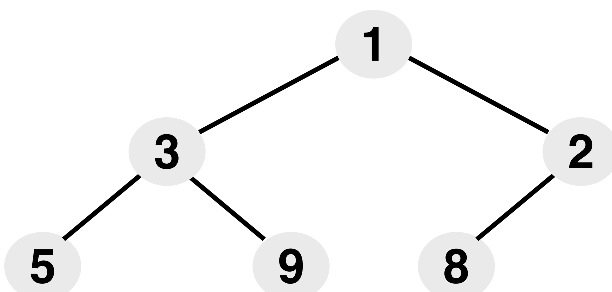


1	3	2	5	9	8

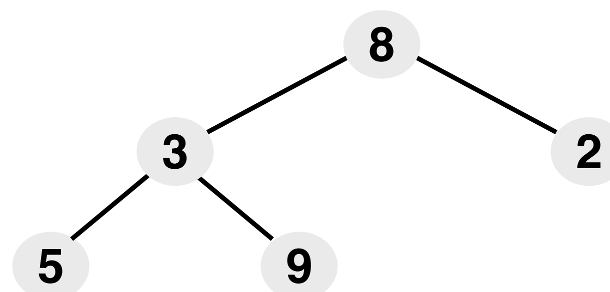
***Heap*-Eigenschaft ist hergestellt**

Phase 2: Sortierung des *Heaps*

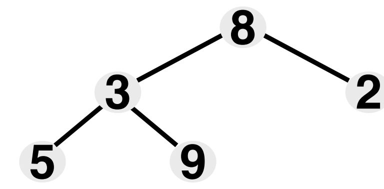
- In Phase 2 des Algorithmus wird der *Heap* sortiert
- Die Sortierung erfolgt durch schrittweises Entfernen der Wurzel mit anschließender Wiederherstellung der *Heap*-Eigenschaft (*heapify*)
- Beispiel des Ablaufs anhand des in Phase 1 aufgebauten *Heaps*
  - Start-Situation vor Phase 2
  - Löschen des Wurzelelements (**1**); ersetze dies durch das letzte Array-Element (**8**); füge dieses als letztes Element im *Array* ein



1	3	2	5	9	8

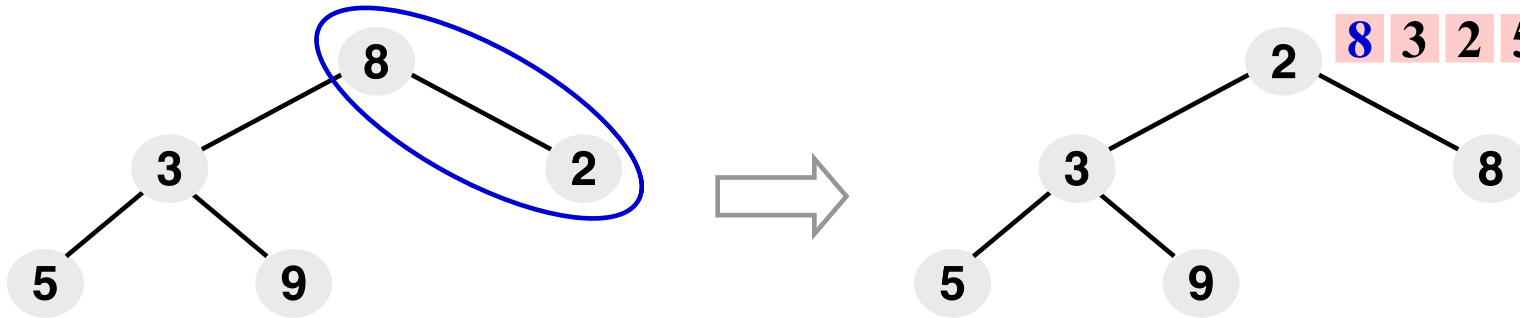


8	3	2	5	9	1



8	3	2	5	9	1

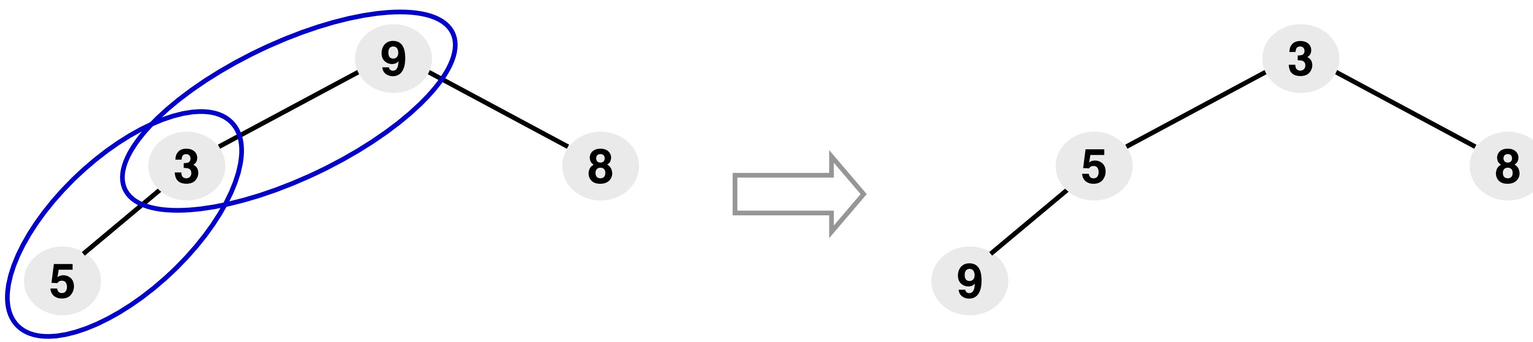
- *Heap*-Eigenschaft wieder herstellen



8	3	2	5	9	1

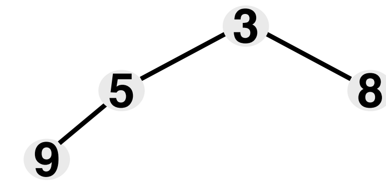
2	3	8	5	9	1

- Wurzelelement löschen (2; in *Array* an der frei werdenden Stelle einfügen); *Heap*-Eigenschaft wieder herstellen



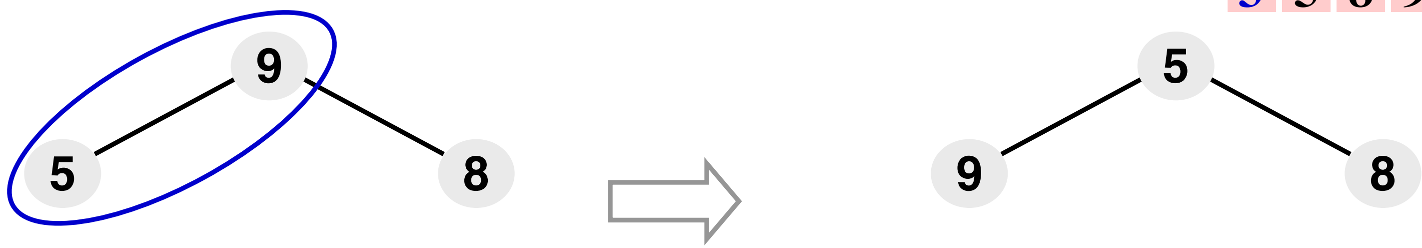
9	3	8	5	2	1

3	5	8	9	2	1



3	5	8	9	2	1

- Wurzelelement löschen (**3**), letztes Element an Wurzel (**9**); *Heap*-Eigenschaft wieder herstellen



9	5	8	3	2	1

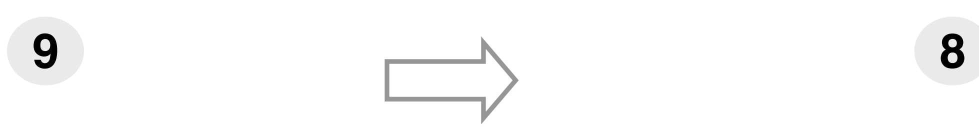
5	9	8	3	2	1

- Wurzelelement löschen (**5**); letztes Element an Wurzel (**8**); *Heap*-Eigenschaft wieder herstellen

8	9	5	3	2	1

8	9	5	3	2	1

- Wurzelelement löschen (**8**); letztes Element an Wurzel (**9**)



Fertig!

9	8	5	3	2	1

9	8	5	3	2	1

Realisierung in Java (Demo: AlgoHeapSort.java)

- Für die Implementierung des Algorithmus in Java benötigen wir zwei Hilfsmethoden:
  - `swapElements`

    Vertauschen zweier Feldelemente (gleiche Methode wie in den Implementierungen der iterativen Sortierverfahren)
  - `percolate`

    Versickern eines Elementes in einer durch ein *Array* repräsentierten *Heap*-Struktur
- Die Sortiermethode `heapSort` in nachfolgendem Programm ist dann nur noch eine direkte Umsetzung des vorangehend in Pseudocode dargestellten Algorithmus

- **Einordnung**: Es werden die Kern-Methoden vorgestellt, ohne weitere Hilfsmethoden zum Vertauschen von Elementen und der Ausgabe der Elemente
- *Heap sort* Methode

```
public class AlgoHeapSort {
    public static void main(String[] args) {
        final int N = 18;
        int[]     array = new int[N];

        for (int i = 0; i < N; i++)
            array[i] = (int)(Math.random() * 10 * N);

        printArray(array);
        heapSort(array);  // Aufruf 'heap sort'
        printArray(array);
    }

    public static void heapSort(int[] arr) {
        for (int i = arr.length/2; i >= 0; i--)
            percolate(arr, i, arr.length-1);

        for (int i = arr.length-1; i > 0; i--) {
            swapElements(arr, 0, i); // tausche jeweils letztes mit erstem Element
            percolate(arr, 0, i-1); // 'heapify' von Position 0 bis i-1
        }
    }

    // Fortsetzung...
```

- Versickern der Elemente zur Erhaltung der *Heap*-Eigenschaft

```
    // Hier geht’s weiter:

    private static void percolate(int[] arr, int parent, int n) {
        int child1 = 2 * parent + 1,  // Indizierung beginnt bei 0
            child2 = child1 + 1;

        if (child2 <= n) {
            int i = (arr[child1] < arr[child2]) ? child1 : child2;

            if (arr[i] < arr[parent]) {
                swapElements(arr, parent, i);
                percolate(arr, i, n);
            }
        }
        else if (child1 <= n) {
            if (arr[child1] < arr[parent])
                swapElements(arr, child1, parent);
        }
    }

    ...

}
```

Zeitlicher Verlauf von *Heap sort* für eine zufällige Testfolge

- Visualisierung des Ablaufs der Sortierung: *http://www.sorting-algorithms.com*

Aufwand von *Heap sort* und Gesamtbewertung

Laufzeiten der Kern-Methoden und Eigenschaften des Verfahrens

- Komplexität von `percolate`
  - Das jeweils zu versickernde Element rutscht in jedem Schritt eine Ebene in dem binären Baum nach unten
  - Komplexität: O(log n)
- Komplexität von `heapSort`
  - Das Versickern wird in der 1. Phase n/2-mal ausgeführt; in der 2. Phase wird das Versickern (n-2)-mal aufgerufen; etc.
  - Insgesamt ergibt sich daraus die Komplexität O(n·log n)
- **Nachteil** von *Heap sort*: Kein stabiles Sortierverfahren, d.h. bei gleichen Schlüsseln wird deren Reihenfolge geändert

Bsp.: Gegeben sei eine Adress-Datenbank; der Nachname sei Schlüssel-Element

Es existieren 2 Datensätze mit Schlüssel-Element „Maier“, wobei sich beide Datensätze in Nicht-Schlüssel-Elementen (Vorname, Adresse, ...) unterscheiden

Beim ***Heap sort*** kann es passieren, dass auch die ursprünglich (vor der Sortierung) vorgegebene Reihenfolge der beiden „Maier“-Datensätze geändert wird

Kriterien für die Auswahl von Sortierverfahren

- **Anzahl der zu sortierenden Elemente**:

  Für kleines n sind die einfach zu implementierenden Algorithmen (*SelectionSort*, *InsertSort*, *BubbleSort*) durchaus zufriedenstellend

- **Anordnung der Elemente in Eingabefolge**

  Es liegt eine unterschiedliche Eignung der Algorithmen bei vorgeordneten Datenfolgen vor, die die Laufzeit beeinflusst

- **Art der Implementierung:**

  Je nachdem, ob für die Implementierung
  - ein *Array*,
  - eine Liste oder
  - eine andere Datenstruktur

  verwendet wird, können die tatsächlichen Ausführungszeiten – und damit der jeweilige Aufwand – variieren

- Sortierverfahren und ihre Komplexität – Gesamtübersicht

Sortierverfahren	*worst case*	*average case*	zusätzlicher Speicher
Selection Sort	n^2	n^2	1
Insertion Sort	n^2	n^2	1
Bubble Sort	n^2	n^2	1
Mergesort	$n \cdot \log n$	$n \cdot \log n$	n
Quicksort	n^2	$n \cdot \log n$	$\log n$
Heap Sort	$n \cdot \log n$	$n \cdot \log n$	1

Verwalten der rekursiven Aufrufe

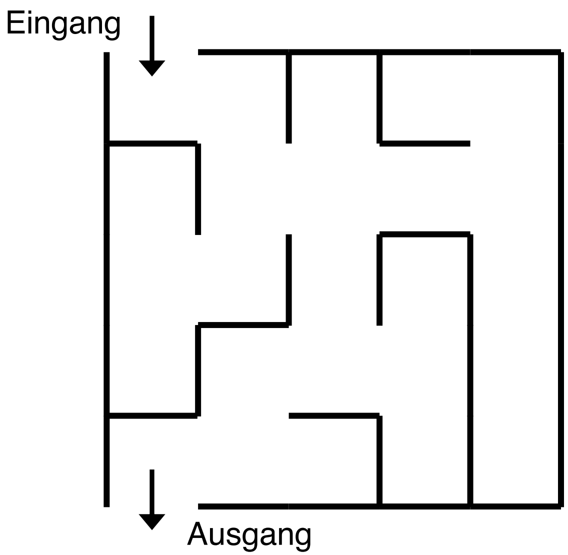
5. Optimierende Suche – *Backtracking*

- Einordnung und Vorgehensweise
- Durchqueren eines Labyrinths
- ~~8-Damen Problem (*Eight queens problem*)~~

Einordnung und Vorgehensweise

Strategie im Allgemeinen

- *Backtracking* ist eine (rekursive) Problemlösungsstrategie, die ein Problem durch sukzessive Erweiterung einer Teillösung bis zum Erreichen einer Gesamtlösung bearbeitet
- Die Vorgehensweise basiert auf dem Prinzip „Versuch-und-Irrtum" (*trial & error*): Wenn ein Lösungspfad nicht zum Ziel führt, werden die Schritte zur Teillösung zurück genommen und an deren Stelle alternative Lösungswege gewählt
- *Backtracking* wird meist rekursiv implementiert
- Anwendungsbeispiele sind …
    - Springerproblem (Wege eines Springers auf einem Schachbrett, Kombinatorik)
    - 8-Damen Problem (zur kollisionsfreien Belegung eines Schachbretts)
    - Rucksackproblem
    - Sudoku
    - Suche eines Weges in einem Labyrinth
    - Wegesuche von A nach B in einem Graphen
    - …



Durchqueren eines Labyrinths

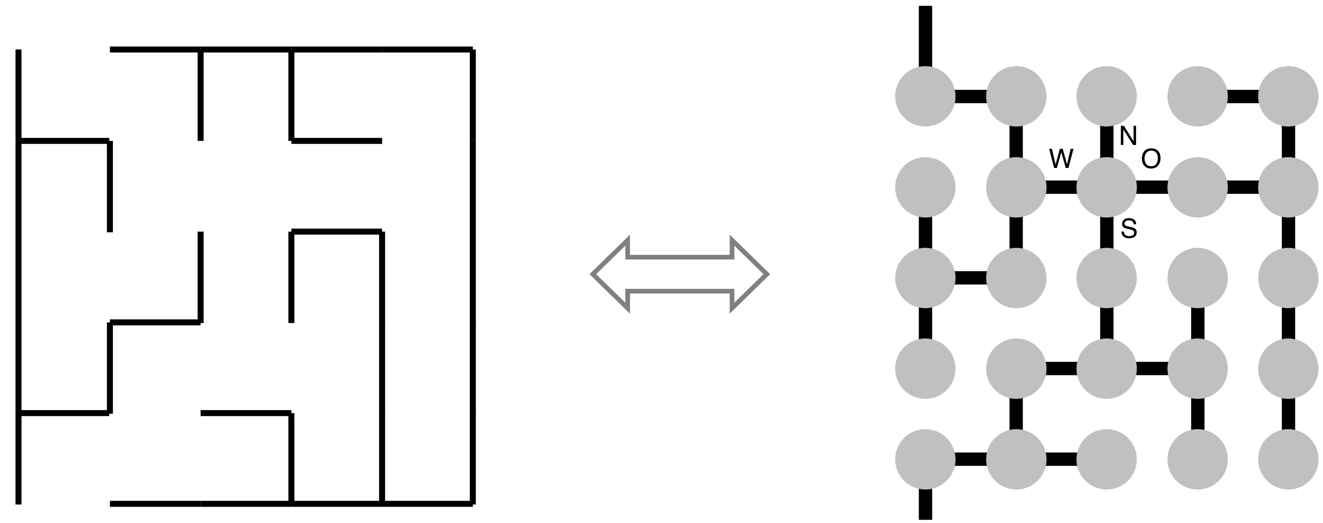
Aufgabenstellung und Wegfindung

- Auf einem Raster mit Feldern sind Wände und offene Durchgangsbereiche zwischen den Rasterflächen (Schachbrett); es gibt einen Eingang (das Quadrat oben links ist der Startpunkt) und einen Ausgang

  Problem: Finde den Ausgang aus einem Labyrinth

- **Repräsentation**: Die Wege auf dem Schachbrett sind horizontal oder vertikal von einem Schachbrett nach links/rechts oder oben/unten, je nachdem ob der Weg frei oder durch eine Wand versperrt ist;

  das Feld lässt sich als Graph mit Knoten (für die Schachbrettposition) und (ungerichteten) Verbindungen als Kanten darstellen (Graphen als Datenstrukturen werden in **Teil XII** diskutiert)



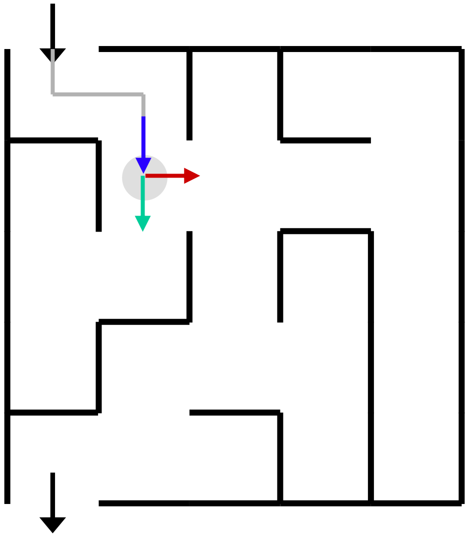
- **Wegfindung**: An jeder Kreuzung stellt man sich die Frage, welchen Weg man weiter geht

  Strategie: Man wählt, wenn es verschiedene Wege gibt, immer möglichst die Variante rechter Hand (in Laufrichtung); wenn es nicht mehr weiter geht, läuft man wieder zurück und nimmt die nächste (noch nicht besuchte) rechte Abzweigung

  Anmerkung: Das Suchproblem durch ein Labyrinth entspricht einer Wegsuche von A nach B in einem Graph

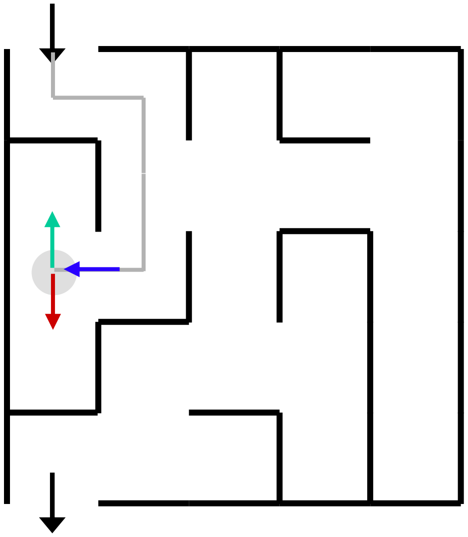
Beispiel für das gegebene Labyrinth

1



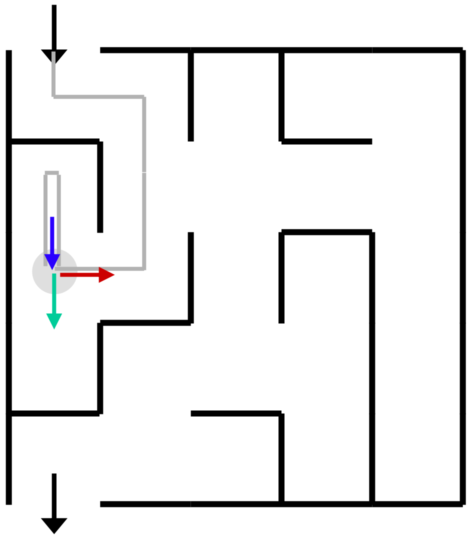
Erste Kreuzung:
rechts abbiegen
(gerade aus)

2



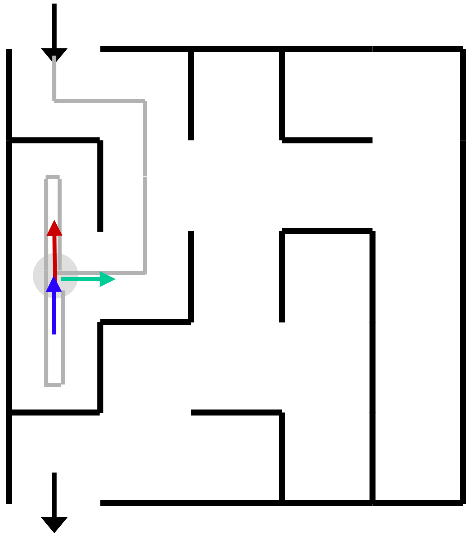
Zweite Kreuzung:
rechts abbiegen

3



Nach der Sackgasse kommen wir zum zweiten Mal an die Kreuzung und wählen wieder die rechte Alternative (gerade aus)

4

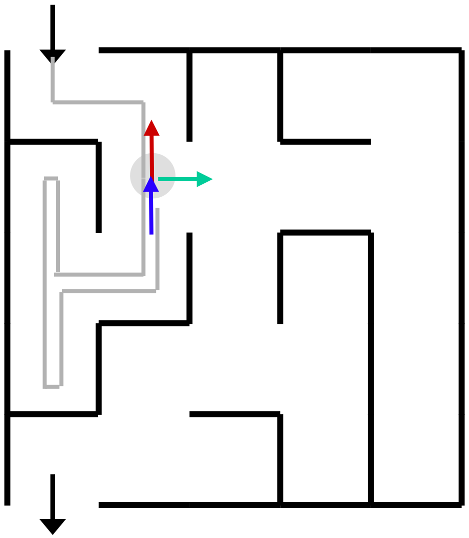


Nach der Sackgasse kommen wir zum dritten Mal an die Kreuzung und wählen wieder die rechte Alternative (rechts)

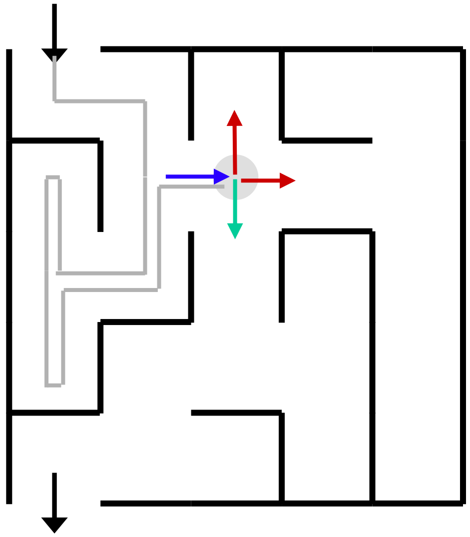
Nach der Sackgasse kommen wir zum zweiten Mal an diese Kreuzung und wählen wieder die rechte Alternative (rechts);

ohne unsere Strategie zu ändern haben wir die „richtige“ Fortsetzung gefunden – wir sind, weil es hier keine Lösung gab, wieder zurück gelaufen (*backtracking*-Schritt).

5

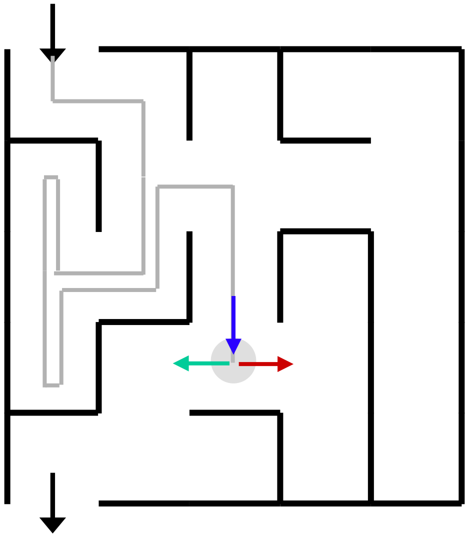


6



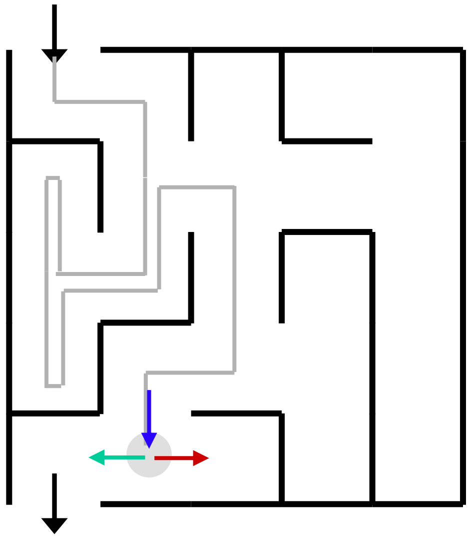
Auch an dieser Kreuzung wählen wir die rechte Abzweigung

7



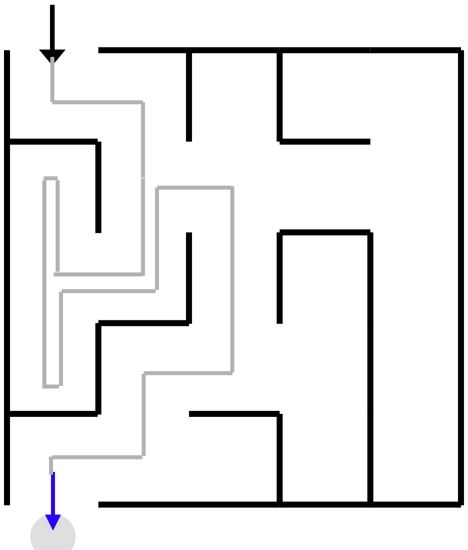
Noch einmal eine Kreuzung: wieder rechts

8



Wieder eine Kreuzung: wie immer zuerst rechts …

9



Geschafft … Ausgang!

Prinzip des *Backtracking* (Rücksetzungsverfahren)

- **Konzept**: Manche Probleme erfordern Lösungen, die aus *n* Komponenten bestehen; dabei bietet jede der Lösungskomponenten evtl. mehrere Auswahlmöglichkeiten

  Eine Gesamtlösung wird dadurch erreicht, dass eine Teillösung ausgewählt und diese dann schrittweise erweitert wird; stellt man fest, dass eine Erweiterung nicht mehr zu einer Lösung führt, geht man einen Schritt zurück (*backtracking*) und wählt für die vorhergehende Lösungskomponente die nächste Alternative

  Dies wird solange wiederholt, bis man entweder eine Lösung gefunden hat, oder keine weiteren Auswahlmöglichkeiten mehr vorhanden sind

- **Eigenschaften**:
  - Strategie ist gut mit rekursiven Methoden lösbar
  - Konzept ist für eine Vielzahl von Problemen einsetzbar
  - Es werden alle Möglichkeiten durchprobiert
  - Der Ansatz definiert ein *trial-and-error*-Verfahren; ist nicht sehr effizient (exponentielle Laufzeit)

Implementierung der Labyrinthaufgabe (Demo: Labyrinth.java)

```
public class Labyrinth {
      static final int START_NODE =  1;
      static final int EXIT_NODE  = 21;

      static boolean[] isVisitedNode = new boolean[26];
      static int[][]   edge          = new int[][] {
            {},
            { 2         }, { 1,  7    }, { 8            }, { 5         }, { 4, 10    },
            {11         }, { 2, 12,  8}, { 3,  9, 13,  7}, { 8, 10     }, { 5,  9, 15},
            { 6, 16, 12}, { 7, 11     }, { 8, 18        }, {19         }, {10, 20    },
            {11         }, {18, 22    }, {13, 17, 19    }, {14, 18, 24}, {15, 25    },
            {22         }, {17, 21, 23}, {22            }, {19         }, {20        }};

      public static void main(String[] args) {
            searchPath(START_NODE);
      }  // end main

      static boolean searchPath(int node) {
            if (isVisitedNode[node])  // dieser Knoten wurde bereits ueberprueft
                  return false;
            else {
                  isVisitedNode[node] = true;

                  if (node == EXIT_NODE) {
                        System.out.println("Ausgang (exit) erreicht");
                        System.out.print(node + " ");
                        return true;              // Ziel erreicht; starte 'Aufstieg'
                  }
                  else {
                        int[] nextNodes = edge[node];

                        for (int i = 0; i < nextNodes.length; i++) {
                              if (searchPath(nextNodes[i])) {
                                    System.out.print(node + " ");
                                    return true;  // Ziel erreicht; 'Aufstieg'
                              }
                        }
                        return false;         // Sackgasse ==> Backtracking
                  }
            }
      }  // end searchPath
}  // end class Labyrinth
```