Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

INFORMATIK

Algorithmen & Datenstrukturen

Höhere Sortieralgorithmen

Hansjörg Diethelm

Inhalt

- Quicksort
- Mergesort
- Datenstruktur Heap
- Heapsort
- Ordnung mit der Java Klassenbibliothek

Lernziele

Sie …

- können die behandelten höheren Sortieralgorithmen an einfachen Beispielen konkret «durchspielen».
- können anhand der Funktionsprinzipen die Laufzeitkomplexitäten der behandelten höheren Sortieralgorithmen darlegen.
- kennen die Merkmale der behandelten höheren Sortieralgorithmen.
- können das Lösungsprinzip «Teile und Herrsche» an Beispielen von höheren Sortieralgorithmen erläutern.
- können eine Heap-Datenstruktur implementieren.
- haben einen Überblick, was die Java Klassenbibliothek betreffend Sortieren bietet.

Quicksort
(C. Antony R. Hoare, 1962)

Lösungsprinzip «Teile und Herrsche»

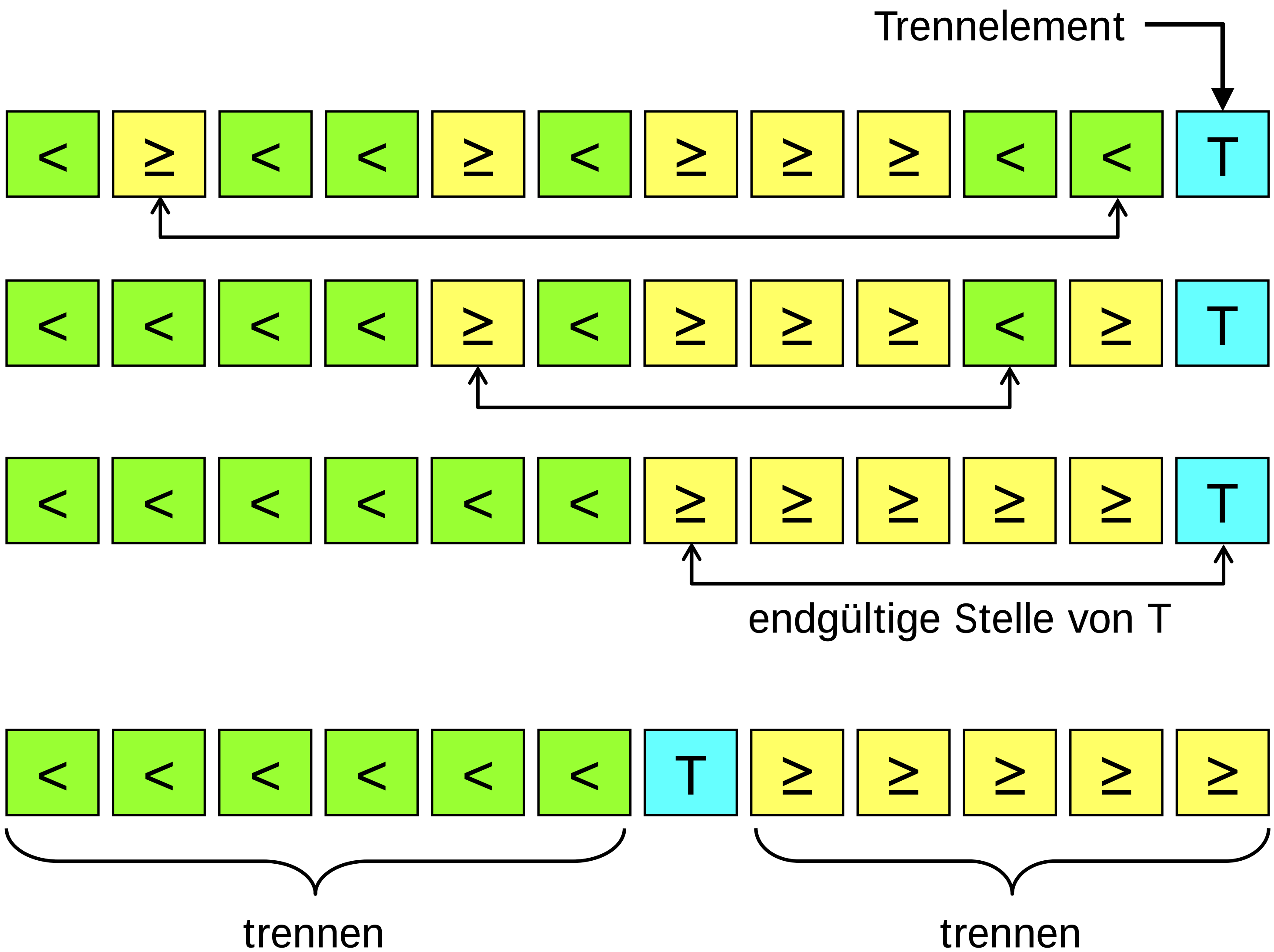
- **Ein komplexes Problem durch Zerlegen in einfachere Teilprobleme lösen:**
  1. Problem in Teilprobleme zerlegen.
  2. Teilprobleme lösen.
  3. Teillösungen zur Gesamtlösung zusammensetzen.

- Damit erreicht man häufig:
  - eine Lösungsfindung
  - einen rekursiven Lösungsansatz
  - parallelisierbare Lösungen (falls Teilprobleme unabhängig lösbar)
  - eine Reduktion des Lösungsaufwandes

Quicksort – Prinzip

- Quicksort arbeitet **instabil** und besitzt im **Average Case eine Zeitkomplexität von $O(n \cdot \log n)$**.
- Quicksort verwendet das Lösungsprinzip «Teile und Herrsche». Er wird rekursiv beschrieben:
  - **Rekursionsbasis**:
    - Eine zu sortierende Folge von **einem** Element ist sortiert.
  - **Rekursionsvorschrift**:
    - Eine zu sortierende Folge von **mehreren** Elementen wird in zwei Teilfolgen getrennt, wobei alle Elemente der ersten Teilfolge kleiner als jene der zweiten sind.
    - Gegebenenfalls werden die Teilfolgen analog weiter getrennt.

Quicksort – Prinzip des Trennens



Quicksort – Prinzip des Trennens

- Anspruch an das Trennelement:
  - **Idealerweise** liegt das Trennelement von seiner Wertigkeit her **genau in der Mitte der Folge**, d.h. halbiert diese in zwei gleich grosse Teilfolgen.

- Ziele beim Trennen:
  - Das Trennelement steht nach dem Trennen an seiner endgültigen Position.
  - Alle Elemente **links vom Trennelement** sind **kleiner** diesem.
  - Alle Elemente **rechts vom Trennelement** sind **grösser/gleich** diesem.

Quicksort – Prinzip

Quicksort – Bestimmung des Trennelementes

- Nach dem Trennen sollen **möglichst gleich grosse Teilfolgen** resultieren.
- Aber, die Bestimmung des Trennelementes soll **wenig Aufwand** verursachen!

- Zwei gängige Verfahren zur Bestimmung des Trennelementes:
  - Man wählt einfach **das letzte Element der Folge**.
  - «**Median of three**»: Man wählt aus drei Elementen der Folge dasjenige mit dem mittleren Wert. Z.B. wählt man aus `folge[anfang]`, `folge[mitte]` und `folge[ende]` das Mittlere und vertauscht es mit jenem am Ende.

Quicksort – Beispiel

```
quickSort(0/11)
EICFGFBAKHDE
DICFGFBAKHEE
DACFGFBIKHEE
DACBGFFIKHEE
DACBEFFIKHEG
```

```
quickSort(0/3)
DACBEFFIKHEG
ADCBEFFIKHEG
ABCDEFFIKHEG
```

```
quickSort(2/3)
ABCDEFFIKHEG
ABCDEFFIKHEG
```

```
quickSort(5/11)
ABCDEFFIKHEG
ABCDEFFEKHIG
ABCDEFFEGHIK
```

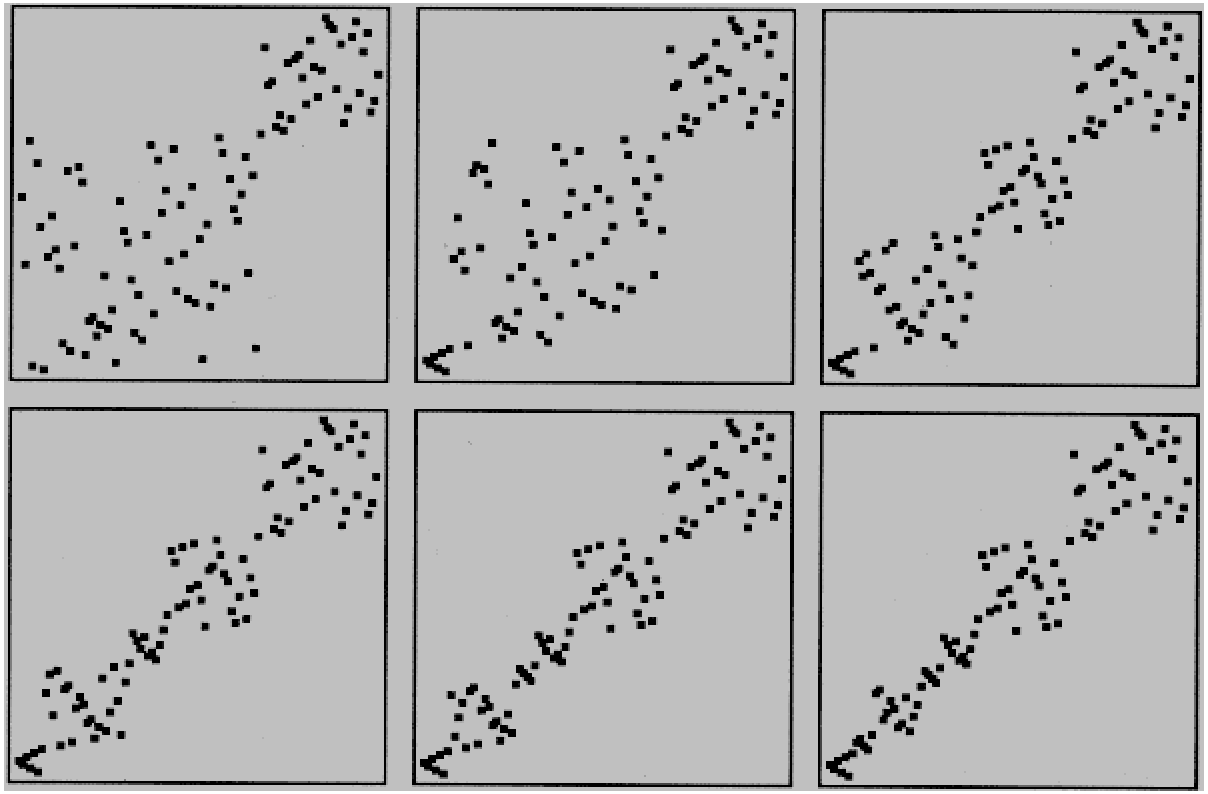
```
quickSort(5/7)
ABCDEFFEGHIK
ABCDEEFFGHIK
```

```
quickSort(6/7)
ABCDEEFFGHIK
ABCDEEFFGHIK
```

```
quickSort(9/11)
ABCDEEFFGHIK
ABCDEEFFGHIK
```

```
quickSort(9/10)
ABCDEEFFGHIK
ABCDEEFFGHIK
```

Quicksort – Prinzip



Quicksort – Implementation (1)

```java
/**
 * Vertauscht zwei bestimmte Zeichen im Array.
 *
 * @param a Zeichen-Array
 * @param firstIndex Index des ersten Zeichens
 * @param secondIndex Index des zweiten Zeichens
 */
private static final void exchange(final char[] a,
        final int firstIndex,
        final int secondIndex) {
    char tmp;
    tmp = a[firstIndex];
    a[firstIndex] = a[secondIndex];
    a[secondIndex] = tmp;
}
```

Quicksort – Implementation (2)

```
public static final void quickSort(final char[] a, final int left, final int right) {
    int up = left;                  // linke Grenze
    int down = right - 1;           // rechte Grenze (ohne Trennelement)
    char t = a[right];              // rechtes Element als Trennelement
    boolean allChecked = false;
    do {
        while (a[up] < t) {
            up++;                   // suche grösseres (>=) Element von links an
        }
        while ((a[down] >= t) && (down > up)) {
            down--;                 // suche echt kleineres(<) Element von rechts an
        }
        if (down > up) {            // solange keine Überschneidung
            exchange(a, up, down);
            up++; down--;           // linke und rechte Grenze verschieben
        } else {
            allChecked = true;      // Austauschen beendet
        }
    } while (!allChecked);
    exchange(a, up, right);         // Trennelement an endgültige Position (a[up])
    if (left < (up - 1)) quickSort(a, left, (up - 1));    // linke Hälfte
    if ((up + 1) < right) quickSort(a, (up + 1), right); // rechte Hälfte, ohne T’Elt.
}
```

Quicksort Optimierungen

Quicksort – gleicher Elemente

- Oben wurde Quicksort so eingeführt, dass beim Trennen **Elemente** < **T** links und **Elemente** $\geq$ **T** rechts vom Trennelement T zu liegen kommen, z.B.

<	$=^1$	>	$=^2$	<	>	$=^3$	>	<	$=^4$	>	>	T
<	<	<	T	>	>	$=^3$	>	$=^1$	$=^4$	>	>	$=^2$

- Sind **Elemente gleich**, so werden diese gegebenenfalls nach rechts vertauscht (vgl. 1 und 2), so dass schlussendlich alle gleichen Elemente wie gefordert rechts von T zu liegen kommen.

- Bei **vielen** gleichen Elementen führt dies dazu, dass **T weit links zu liegen kommt** bzw. die Trennung asymmetrisch erfolgt und damit das Sortieren viele rekursive Methodenaufrufe erfordert bzw. das **Sortieren «lange» dauert**.

Quicksort – Behandlung gleicher Elemente

- Passt man Quicksort so an, dass beim Trennen auch **Elemente = T** die Seite wechseln, so kann man erreichen, dass das Trennen symmetrischer erfolgt bzw. **eher gleich lange Teilfolgen** resultieren, also

$<$	$=^{1}$	$>$	$=^{2}$	$<$	$>$	$=^{3}$	$>$	$<$	$=^{4}$	$>$	$>$	T
$<$	$=^{4}$	$<$	$=^{3}$	$<$	T	$=^{2}$	$>$	$>$	$=^{1}$	$>$	$>$	$>$

- Damit gilt nach dem Trennen:
  - links: **Elemente $\leq$ T**
  - rechts: **Elemente $\geq$ T**
- Insgesamt sind damit zwar etwas mehr Element-Vertauschungen notwendig, was aber weniger rekursive Methodenaufruf mehr als wettmachen!

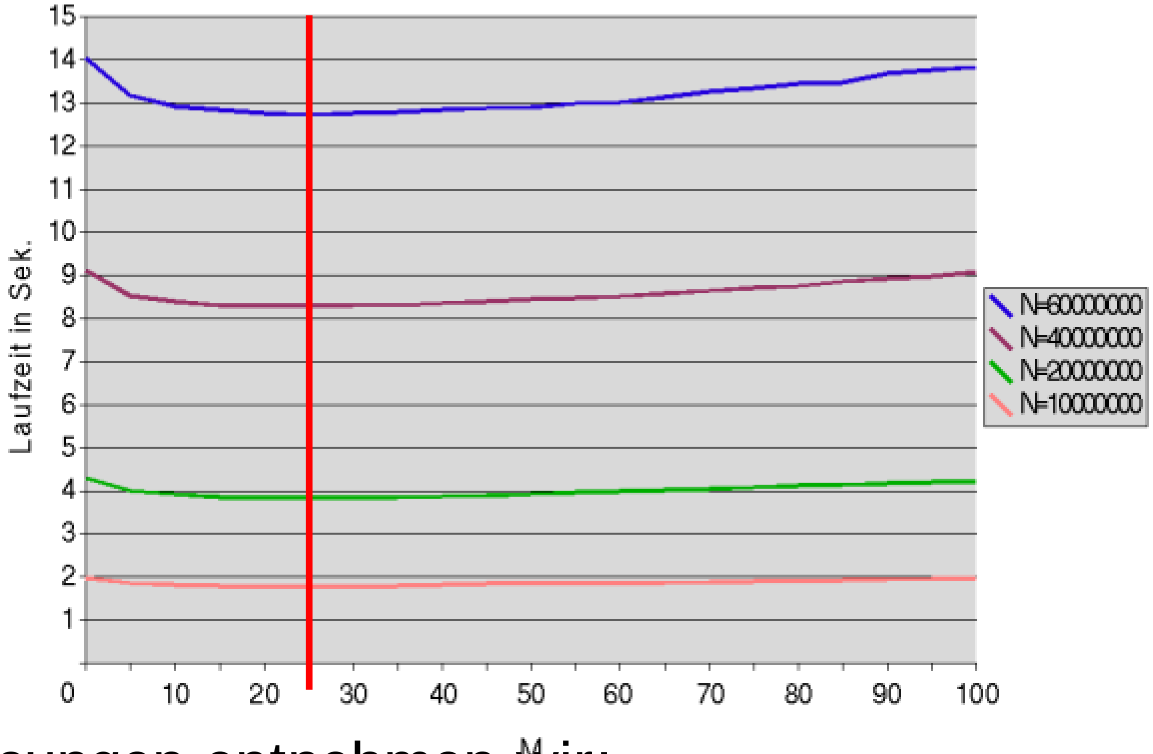
Quick-Insertion-Sort – Behandlung kurzer Teilfolgen

- Der klassische Quicksort zerlegt eine Folge weiter in Teilfolgen, auch wenn diese nur noch aus 2 Elementen besteht!
- Die damit verbundenen rekursiven Methodenaufrufe benötigen «**viel Zeit**» (und «Speicher»).

- **Schneller** geht's, wenn eine **Folge mit weniger als M Elementen** z.B. mit «**Insertion Sort**» zu Ende sortiert wird.
- Damit lassen sich in Abhängigkeit von der Wahl von M Laufzeitverbesserungen von 20% und mehr beobachten.

Quick-Insertion-Sort – Wahl von M

- Für **welches M** ist die Laufzeit = f(M) minimal?
  und zwar bei N = 10, 20, 40 und 60 Mio. Elementen



- Den Messungen entnehmen wir:
  - M ist praktisch unabhängig von N.
  - **kürzeste Laufzeiten bei M = 25**

Quick-Insertion-Sort – Beispiel

quickSort(0/11)

EICFGFBAKHDE
DICFGFBAKHEE
DACFGFBIKHEE
DACBGFFIKHEE
DACBEFFIKHEG

insertionSort(0/3)

DACBEFFIKHEG
ABCDEFFIKHEG

quickSort(5/11)

ABCDEFFIKHEG
ABCDEFFEKHIG
ABCDEFFEGHIK

insertionSort(5/7)

ABCDEFFEGHIK
ABCDEEFFGHIK

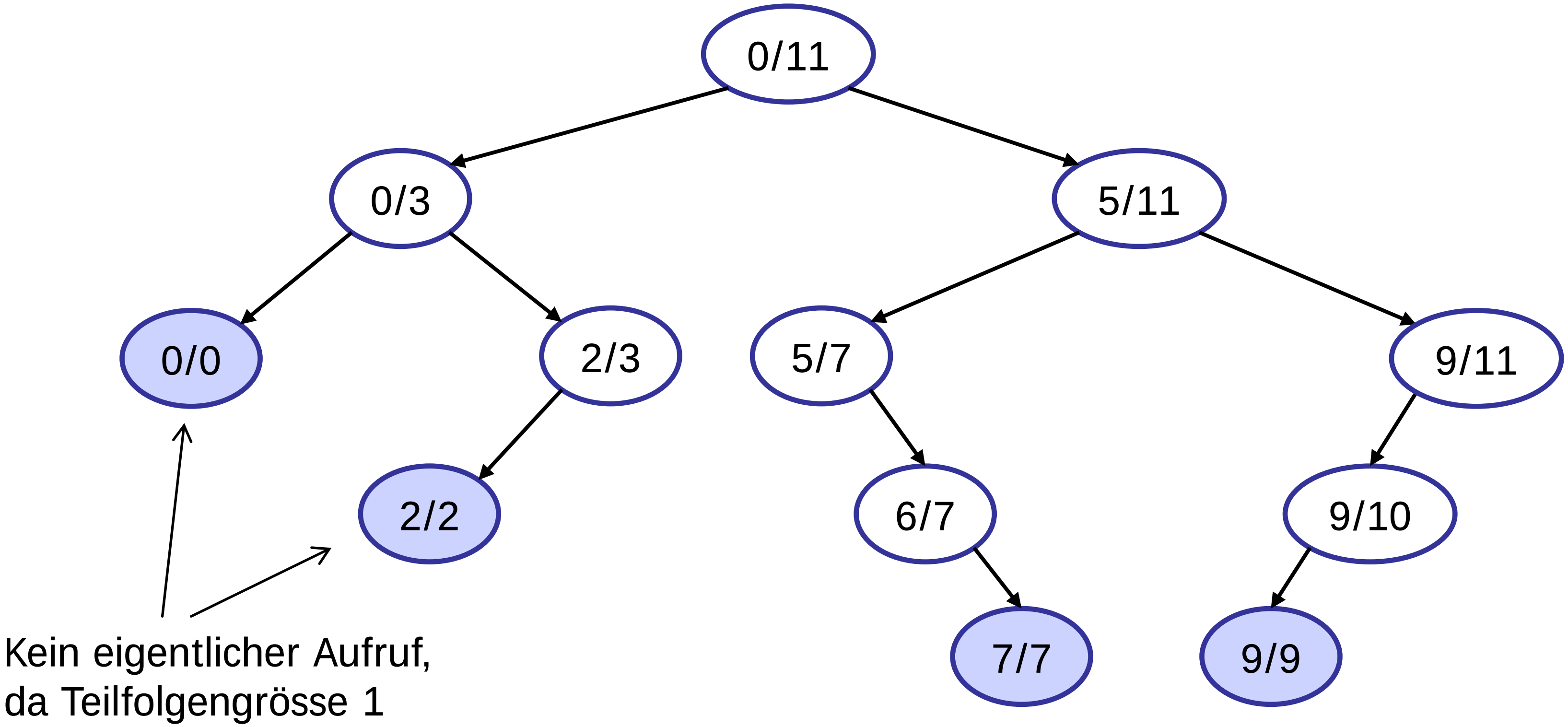
insertionSort(9/11)

ABCDEEFFGHIK
ABCDEEFFGHIK

Quicksort
Laufzeitbetrachtungen

Quicksort – Methodenaufrufe als Binärbaum

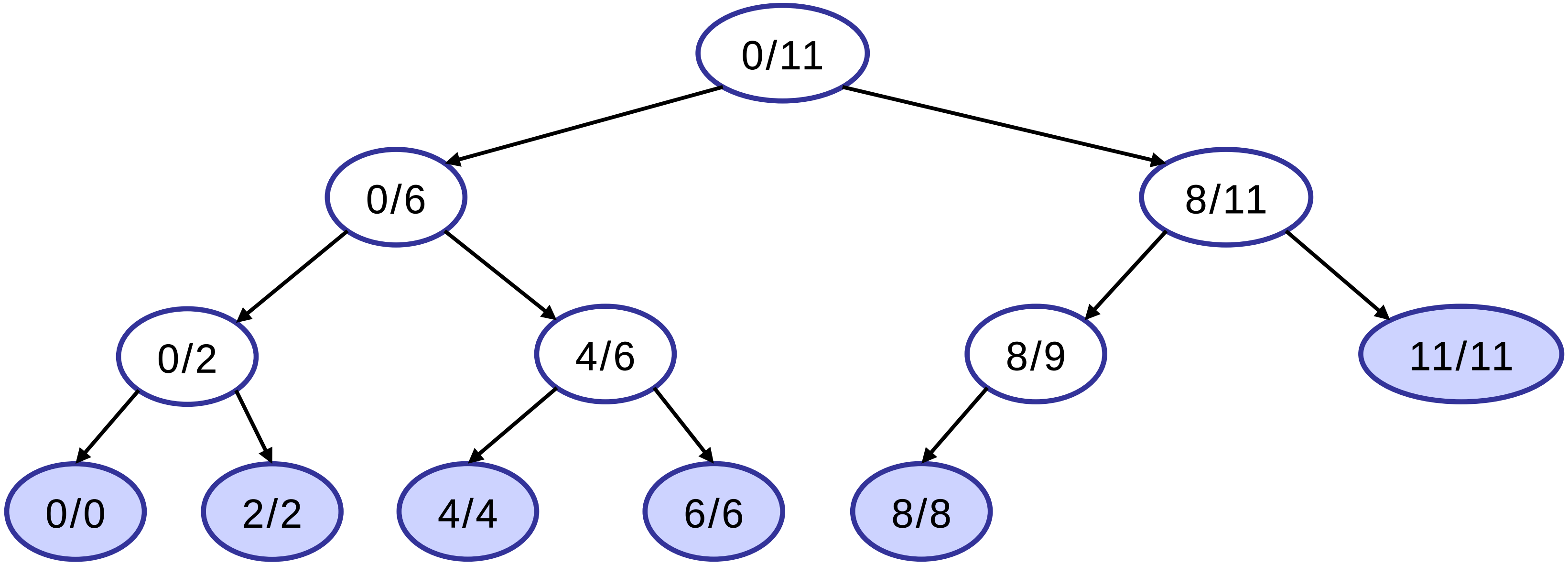
- Das Abarbeiten von Quicksort lässt sich **als Binärbaum** darstellen.
- Jeder Methodenaufruf von Quicksort ist ein Knoten (left/right).
- Für das Beispiel auf Slide 11:



Quicksort – «Best Case»

- Das gewählte **Trennelement** kommt **immer in die Mitte** der Folge zu liegen:
  - Es resultiert ein Binärbaum mit $\mathbf{log_2\, n}$ Niveaus.
  - Pro Niveau braucht es insgesamt immer rund **n** Vergleiche.

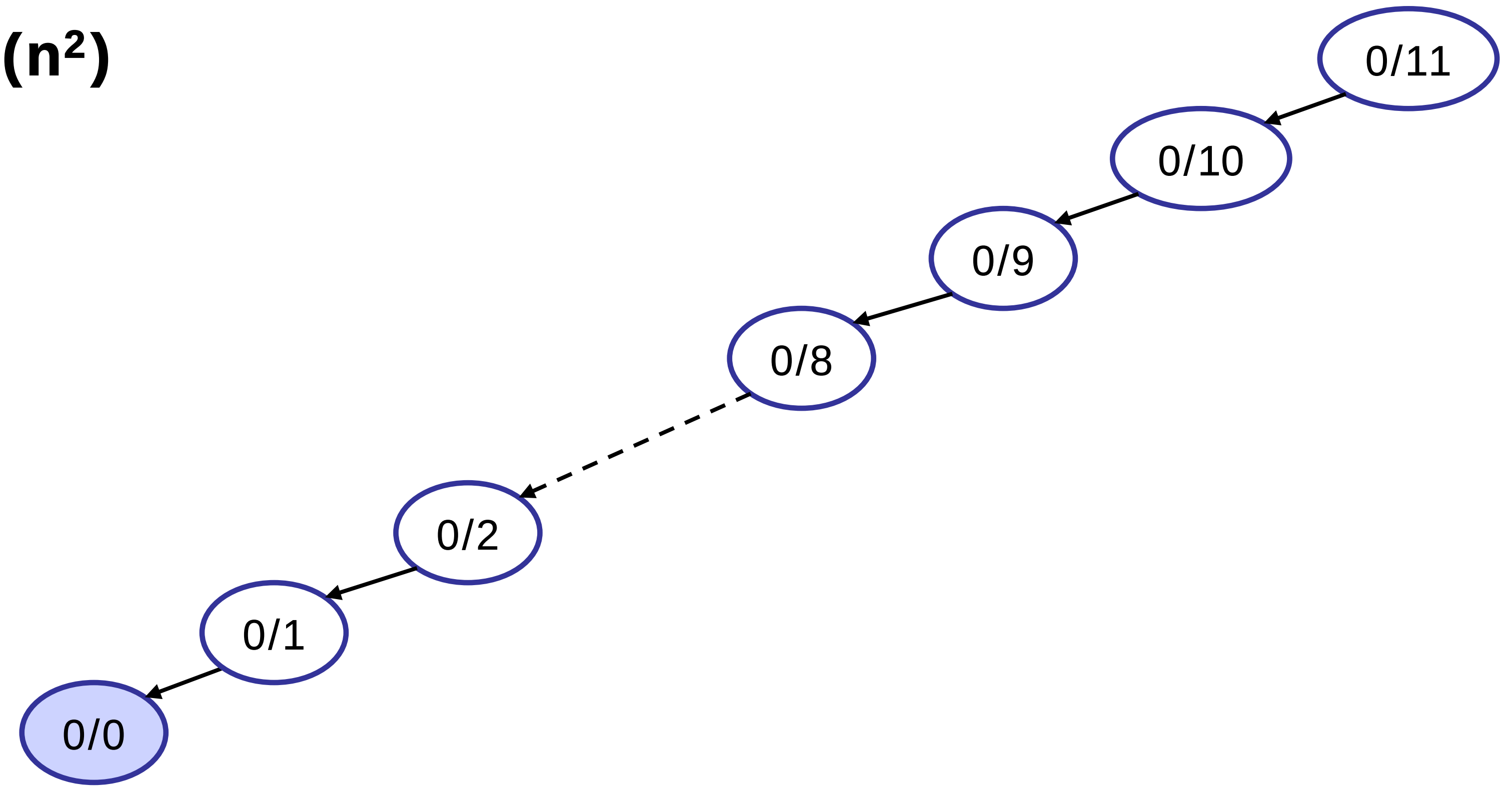
➔ $\mathbf{O(n \cdot log_2\, n)}$



Quicksort – «Worst Case»

- Das gewählte **Trennelement** kommt **immer an den Rand** der Folge zu liegen:
  - Es resultiert ein degenerierter Binärbaum mit **n** Niveaus.
  - Pro Niveau braucht es durchschnittlich rund **n / 2** Vergleiche.

➔ **$O(n^2)$**



Quicksort – «Average Case»

- Bei zufälligen Permutationen ist der Quicksort-Algorithmus **im Mittel nicht viel schlechter als im «Best Case»**.
- Es lässt sich zeigen, dass die Anzahl Niveaus des Binärbaumes im Mittel rund 39% grösser ist wie im «Best Case», d.h. Quicksort arbeitet im «Average Case» nur rund 1.4 Mal langsamer.

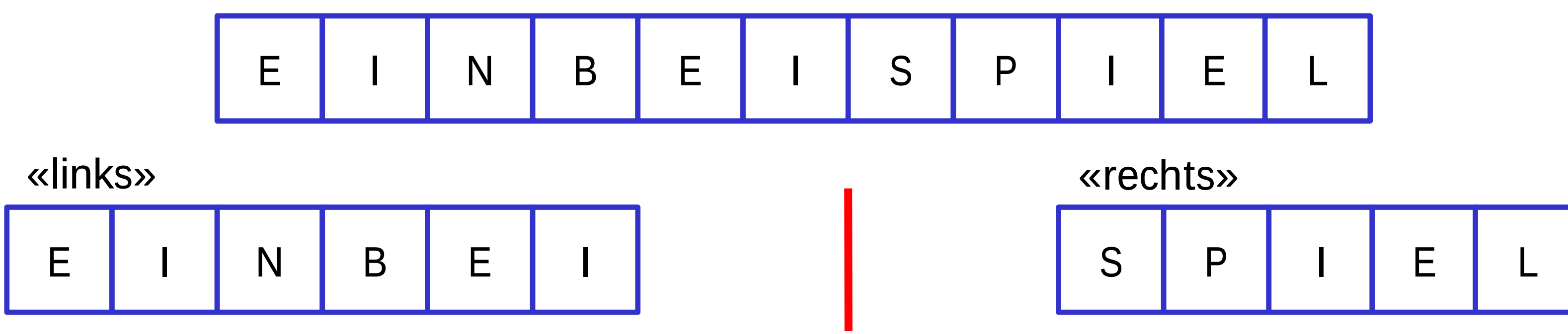
➔ **O(n · log n)**

- Dieser gute «Average Case» ist einer der Hauptgründe, weshalb sich Quicksort in der Praxis als eines der beliebtesten Sortier-verfahren etabliert hat.

Mergesort
(John von Neumann, 1945)

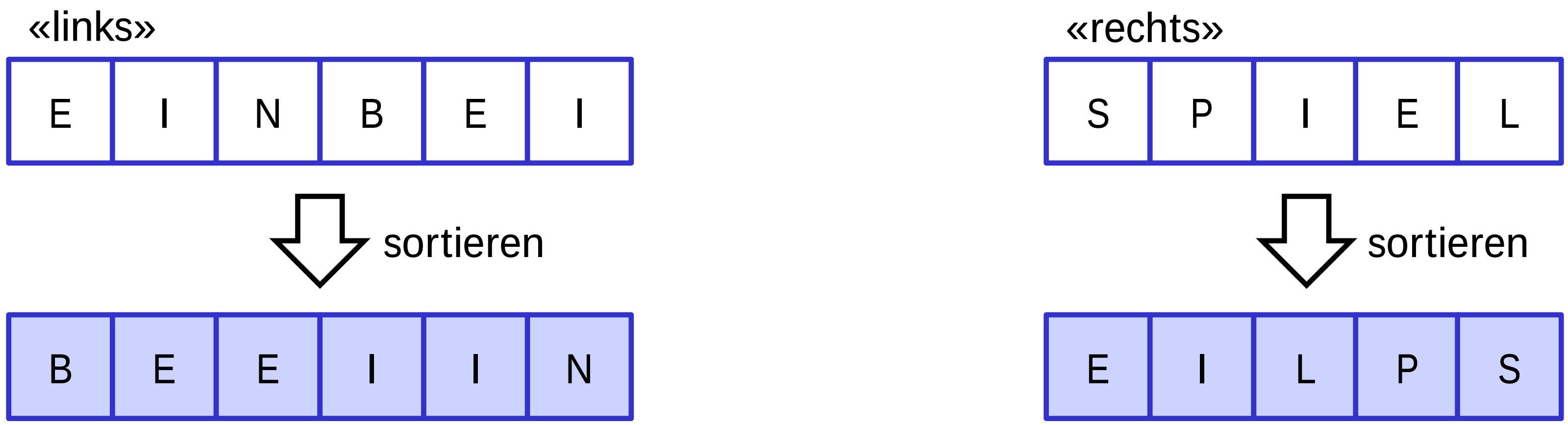
Mergesort – Prinzip

- Mergesort arbeitet **stabil** und besitzt **generell eine Zeitkomplexität von O(n · log n)**.
- Mergesort verwendet wie Quicksort das Lösungsprinzip «Teile und Herrsche». Er wird rekursiv beschrieben:
  - **Rekursionsbasis**:
    - Eine zu sortierende Folge von **einem** Element ist sortiert.
  - **Rekursionsvorschrift**:
    1. Die zu sortierende Folge von **mehreren** Elementen wird in zwei Hälften halbiert, z.B.



Mergesort – Prinzip

2. Sortiere die linke und die rechte Hälfte.
   - D.h. auf zwei einfachere Sortierprobleme zurückgeführt.
   - Weil unabhängig, gegebenenfalls sogar parallel sortierbar.



3. Füge die beiden sortierten Hälften zu der sortierten Folge zusammen, und zwar mit dem «Reissverschlussverfahren» bzw. durch «Mischen» (to merge).

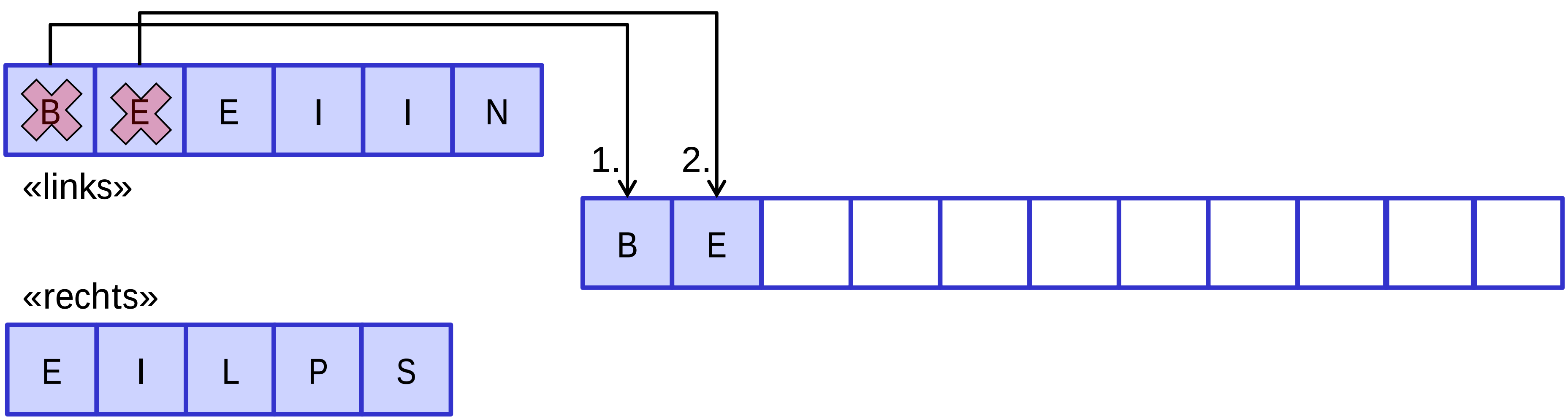
Mergesort – Prinzip

«Mischen»:

a) Vergleiche die beiden ersten Elemente der sortierten Hälften.

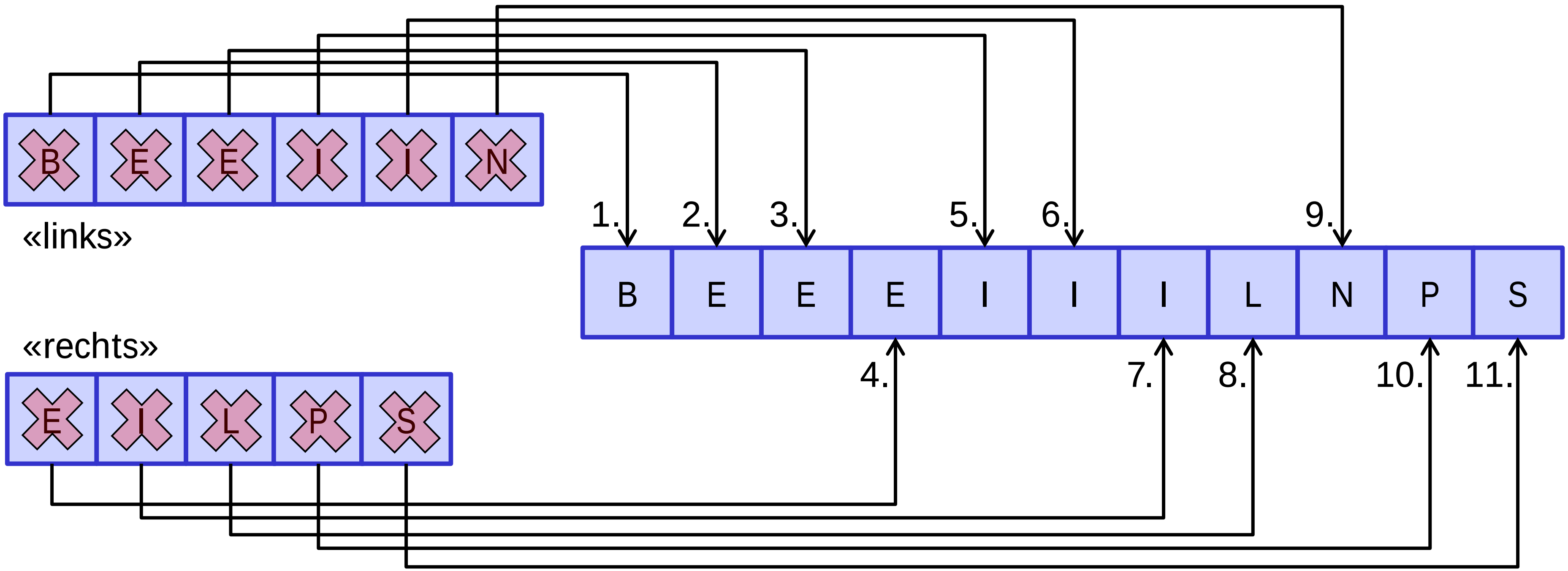
b) Kopiere das Kleinere und füge es dem Resultat hinzu. (Falls die Elemente gleich sind, so kopiere das Element von «links».)

c) Gehe zu a), so lange nicht beide Hälften abgearbeitet sind.

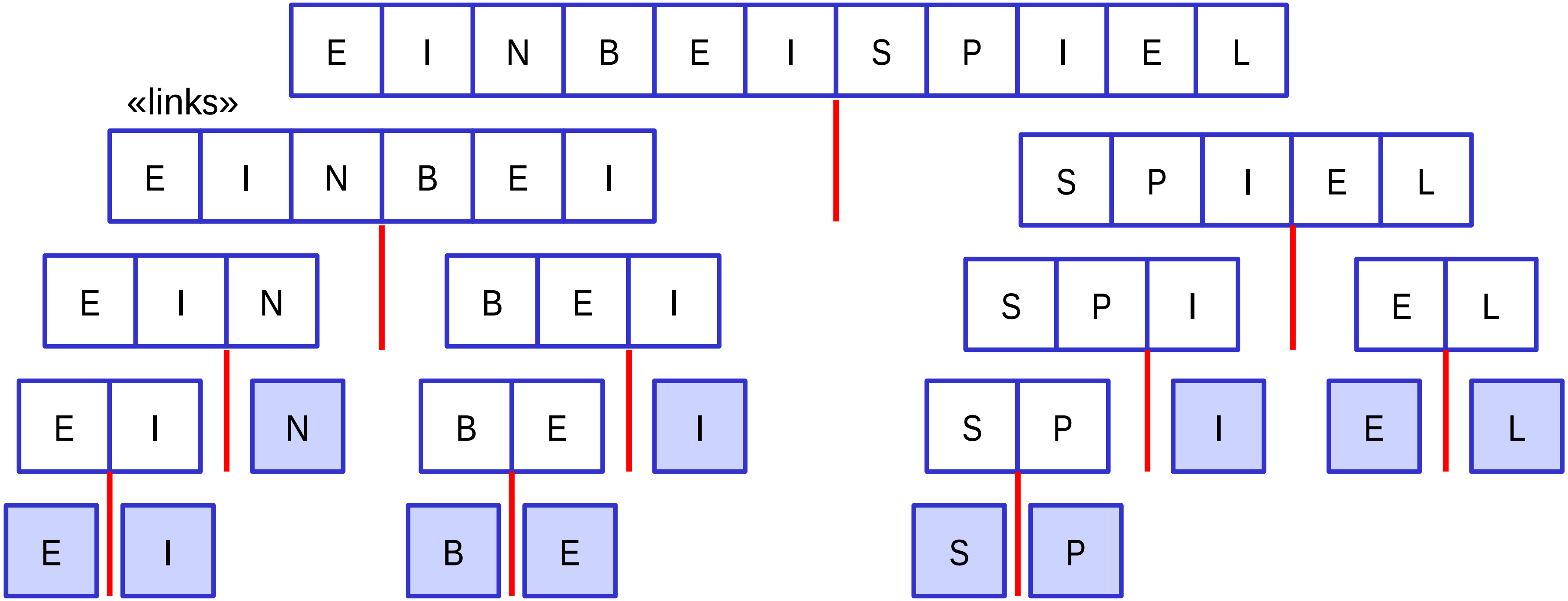


Mergesort – Prinzip

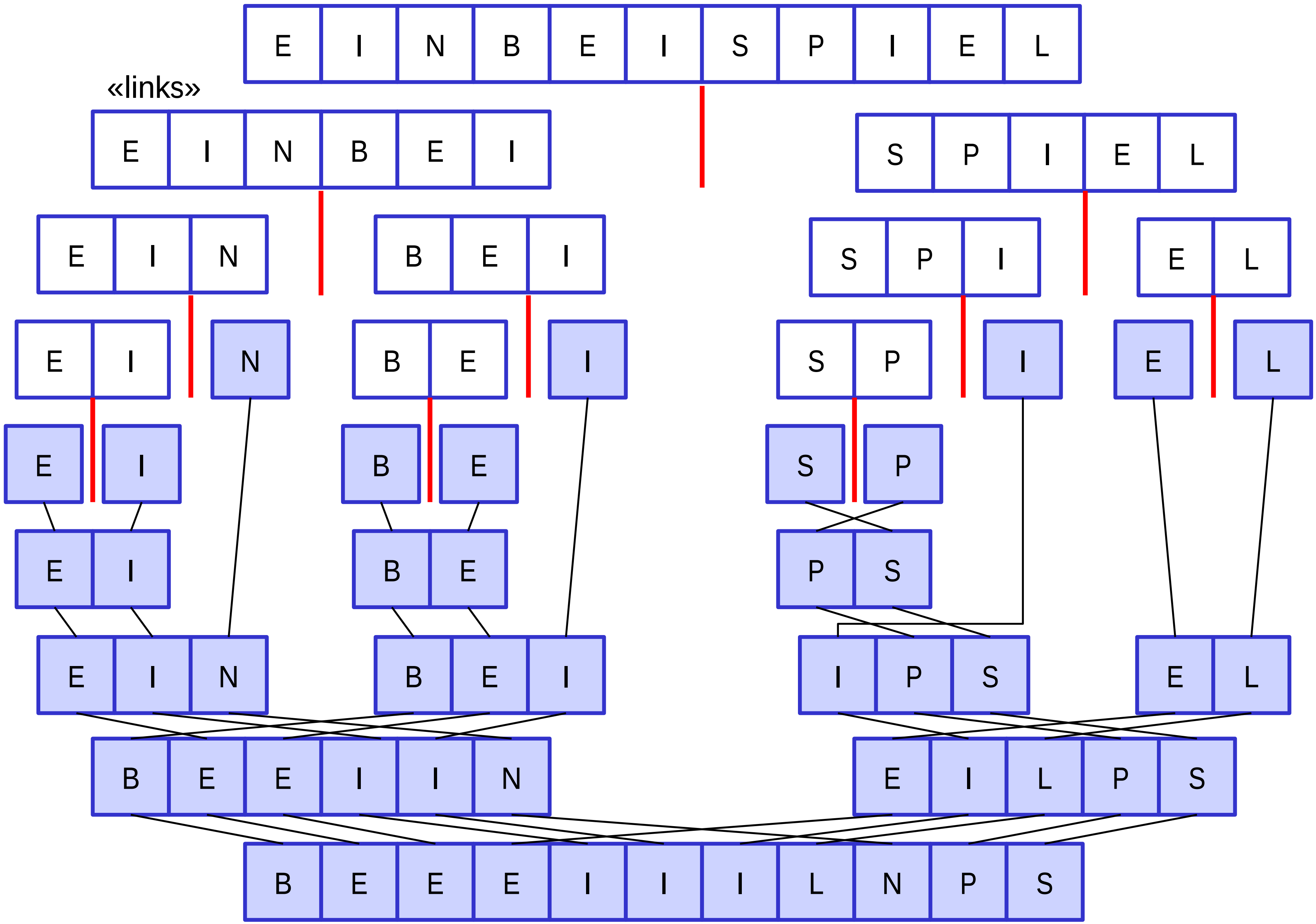
«Mischen»:



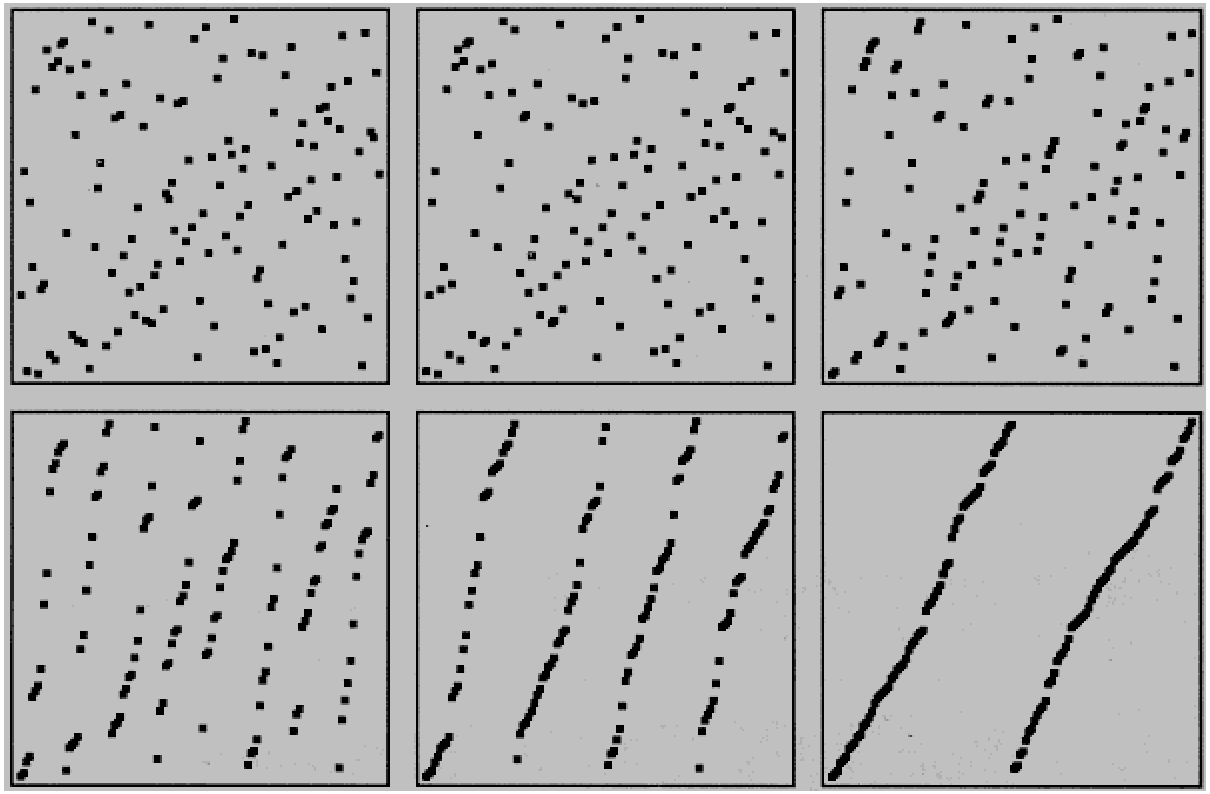
Mergesort – 1. Halbieren



Mergesort – 2. Mischen



Mergesort – Prinzip



Mergesort – Komplexität

- Beim Halbieren resultieren gleich grosse Teilfolgen (+/- 1 Elt.):
  - Es resultiert garantiert ein Binärbaum mit $\mathbf{\log_2 n}$ Niveaus.
  - Pro Niveau braucht es insgesamt immer rund **n** Vergleiche.

➔ **Zeitkomplexität $O(n \cdot \log_2 n)$**

- Mergesort benötigt aber **für das Mischen zusätzlichen Speicherplatz**, und zwar für **n** Elemente (vgl. Slide 30).

➔ **Speicherkomplexität $O(n)$**

Mergesort – Implementation (1)

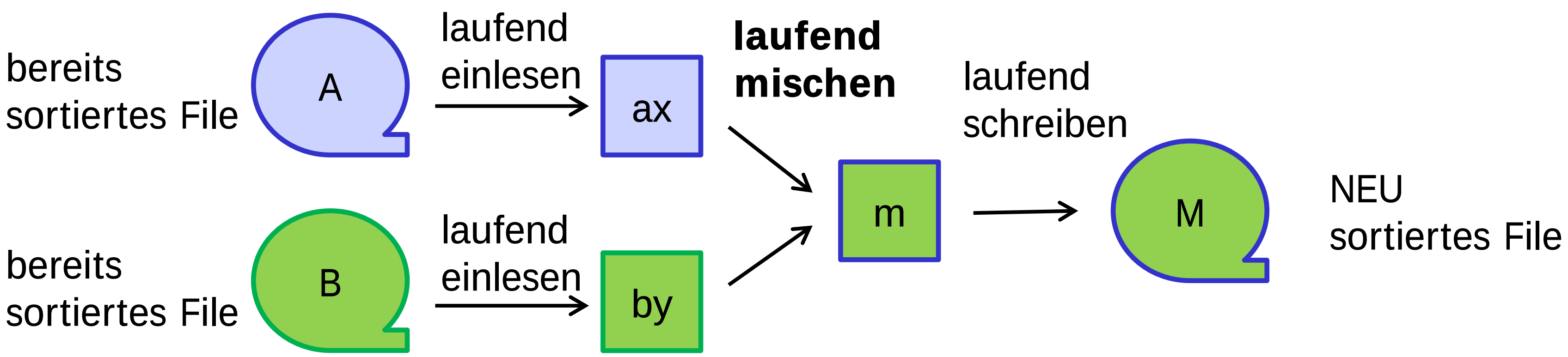
```java
private static char[] b;
/**
 * Sortiert ein Zeichen-Array mit dem Mergesort-Algorithmus.
 * @param a Zeichen-Array zum Sortieren
 */
public static void mergeSort(final char[] a) {
    b = new char[a.length];  // zusätzlicher Speicher fürs Mergen
    mergeSort(a, 0, a.length - 1);
}

/**
 * Rekursiver Mergesort-Algorithmus.
 * @param a Zeichen-Array zum Sortieren
 * @param left linke Grenze, zu Beginn 0
 * @param right rechte Grenze, zu Beginn a.length - 1
 */
private static void mergeSort(final char a[], final int left,
        final int right) {
    ... ;
}
```

Mergesort – Implementation (2)

```java
private static void mergeSort(final char a[], final int left, final int right) {
    int i, j, k, m;
    if (right > left) {
        m = (right + left) / 2;         // Mitte ermitteln
        mergeSort(a, left, m);          // linke Hälfte sortieren
        mergeSort(a, m + 1, right);     // rechte Hälfte sortieren
        // "Mergen"
        for (i = left; i <= m; i++) {   // linke Hälfte in Hilfsarray kopieren
            b[i] = a[i];
        }
        for (j = m; j < right; j++) {   // rechte Hälfte umgekehrt in Hilfsa. kopieren
            b[right + m - j] = a[j + 1];
        }
        i = left; j = right;            // Index für linke und rechte Hälfte
        for (k = left; k <= right; k++) {    // füge sortiert in a ein
            if (b[i] <= b[j]) {
                a[k] = b[i]; i++;
            } else {
                a[k] = b[j]; j--;
            }
        }
    }
}
```

Mergesort – auch geeignet zum externen Sortieren

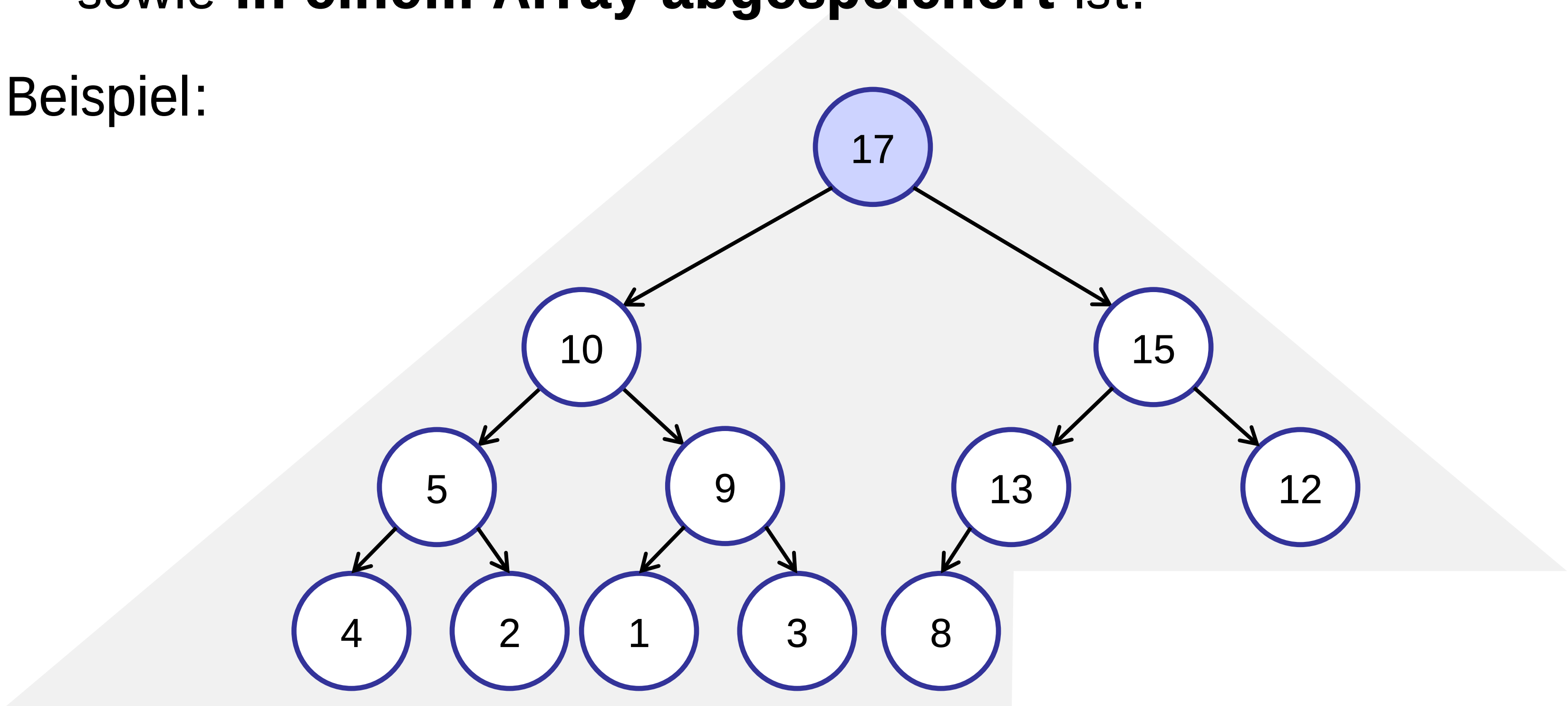
- Externes Sortieren mit Mergesort:
  1. Daten so aufteilen, dass sie mit einem internen Sortierverfahren sortiert werden können. Die sortierten Daten als Files speichern (A, B, …).
  2. Zwei Files A und B parallel einlesen und in ein drittes File M mischen/schreiben.



Datenstruktur Heap
für Heapsort
(ungleich Heap bei der JVM, nur gleiche Namen!)

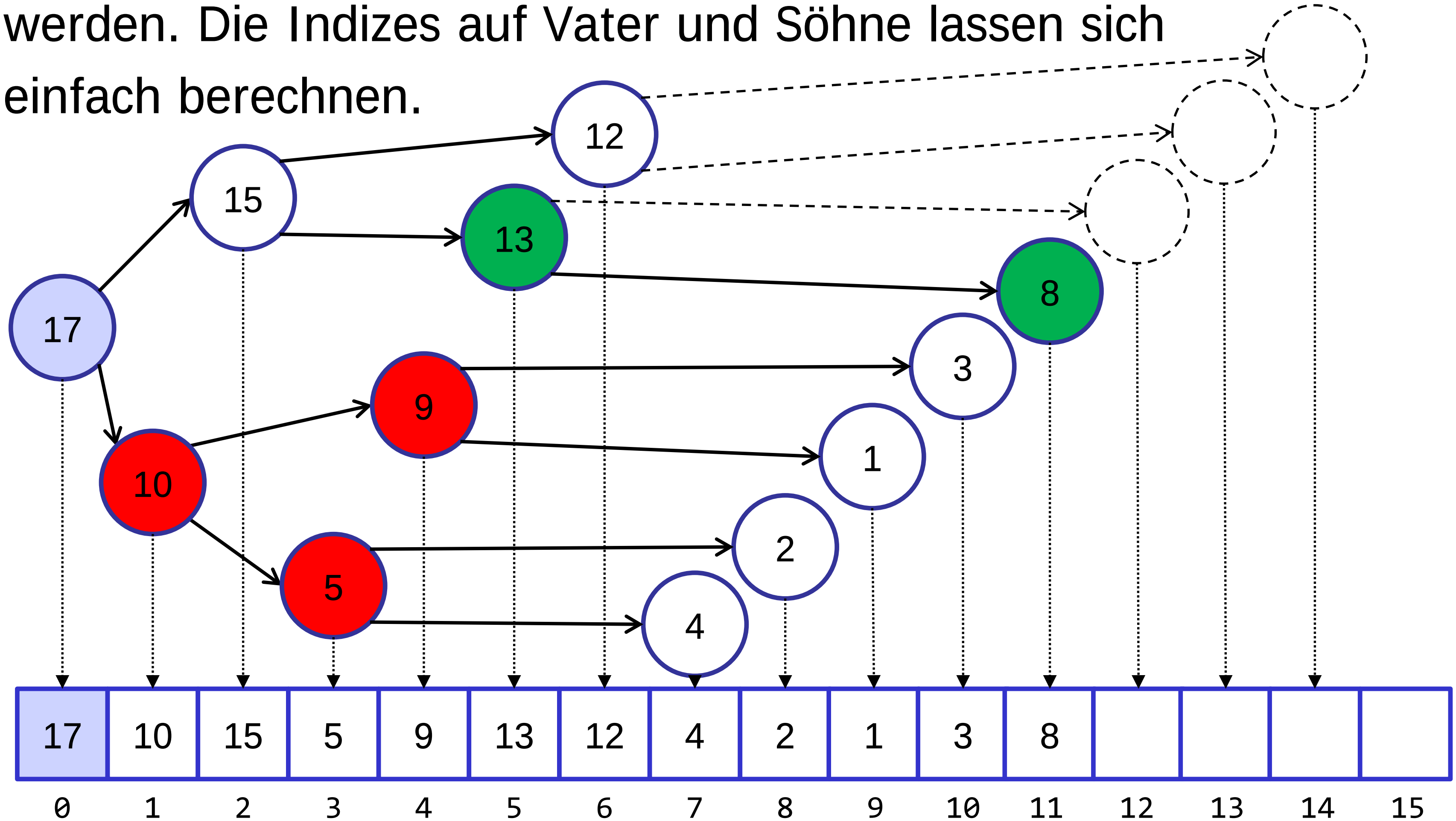
Datenstruktur Heap – Definition

- Ein Heap ist ein **binärer Baum**, der …
  - eine **strukturelle Bedingung** erfüllt, d.h. **voll** ist (➔ D21_IP_Bäume) UND
  - eine **inhaltliche Bedingung** erfüllt, d.h. jeder **innere Knoten** ≥ **als seine Söhne** ist,
  - sowie **in einem Array abgespeichert** ist.
- Beispiel:



Datenstruktur Heap – Abbildung in ein Array

Der volle binäre Baum kann einfach in einem Array abgespeichert werden. Die Indizes auf Vater und Söhne lassen sich einfach berechnen.

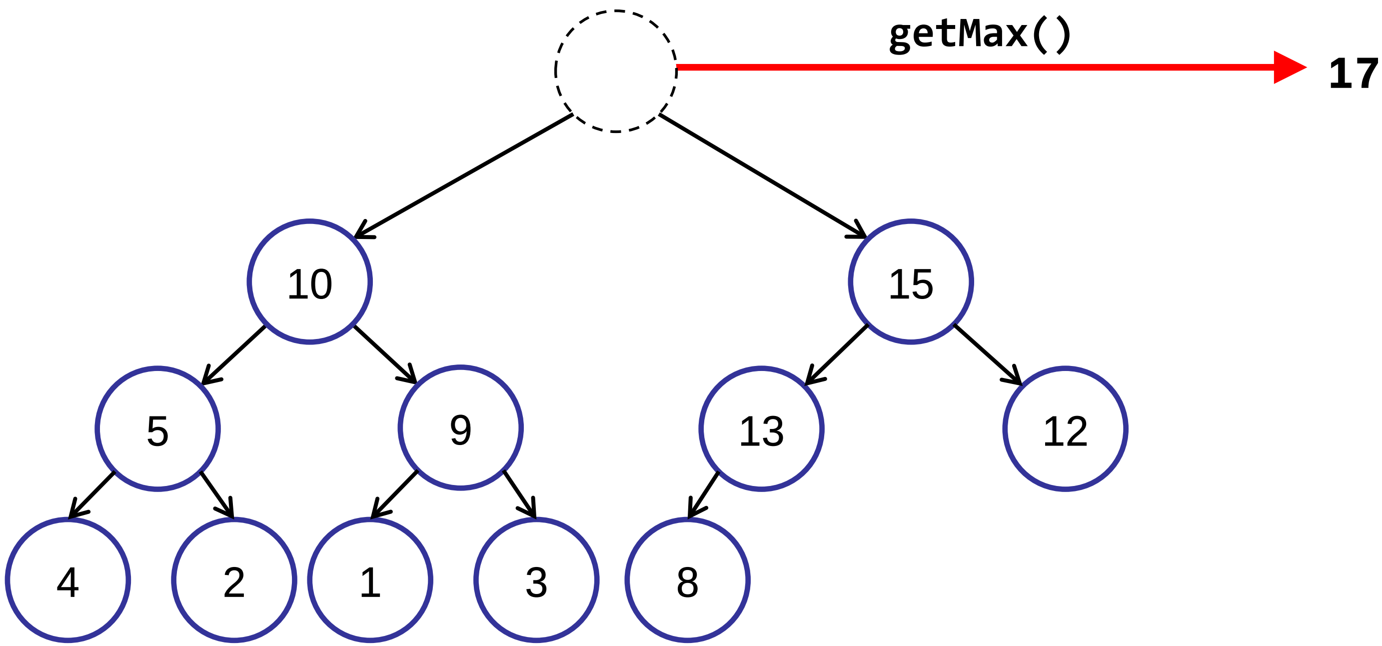


Vater mit Index `(j-1)/2` ⬅ Sohn mit Index `j=11`

Vater mit Index `i=1` ➡ linker Sohn mit Index (2*i)+1 | rechter Sohn mit Index 2*(i+1)

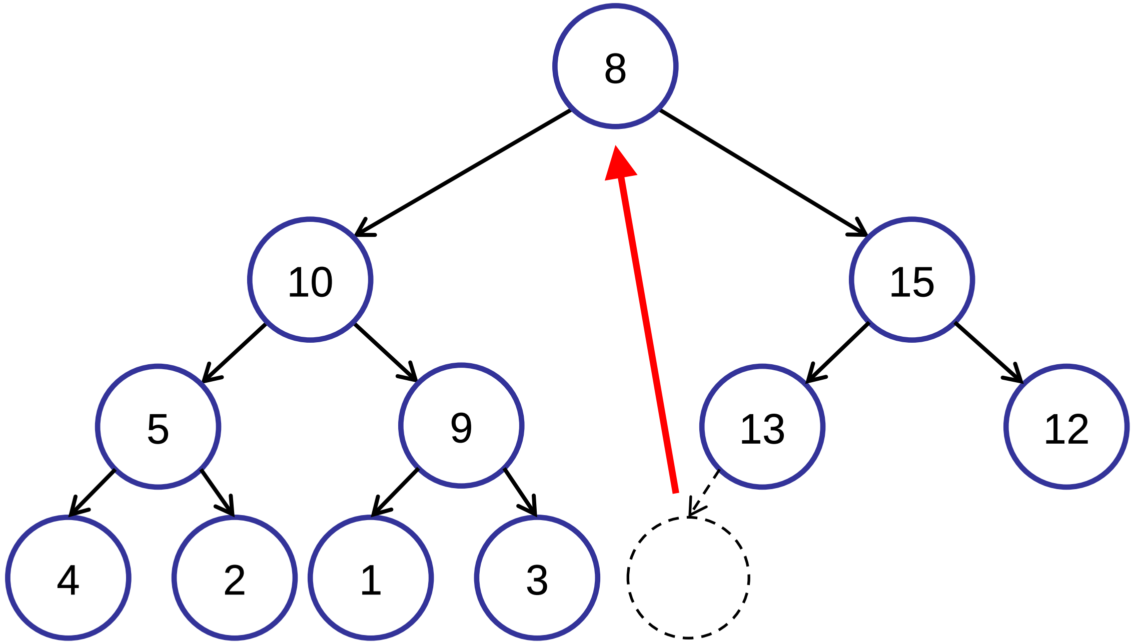
Datenstruktur Heap – `getMax()`

- Nach dem **Entfernen des Wurzel-Elementes** mit **`getMax()`** muss der Baum **reorganisiert** werden, damit die strukturelle und die inhaltliche Bedingung für den Heap wieder erfüllt sind:
  1. Wurzelelement entfernen: O(1)



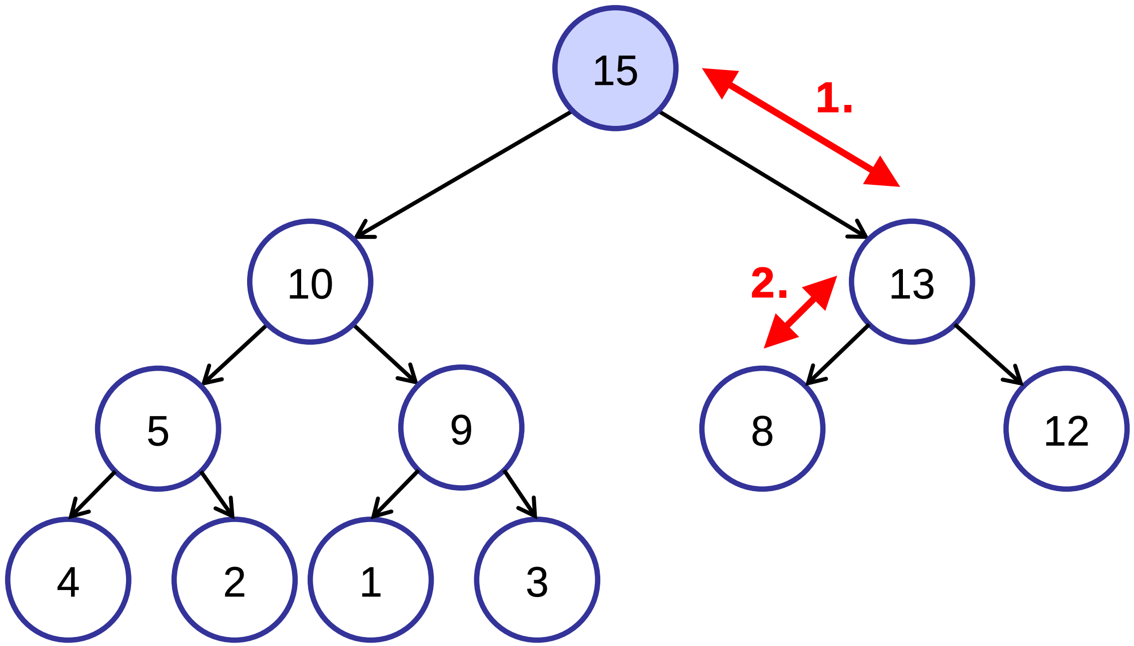
Datenstruktur Heap – `getMax()`

2. Blatt unten rechts zur Wurzel hoch verschieben bzw. **strukturelle Bedingung** sicherstellen: O(1)



Datenstruktur Heap – `getMax()`

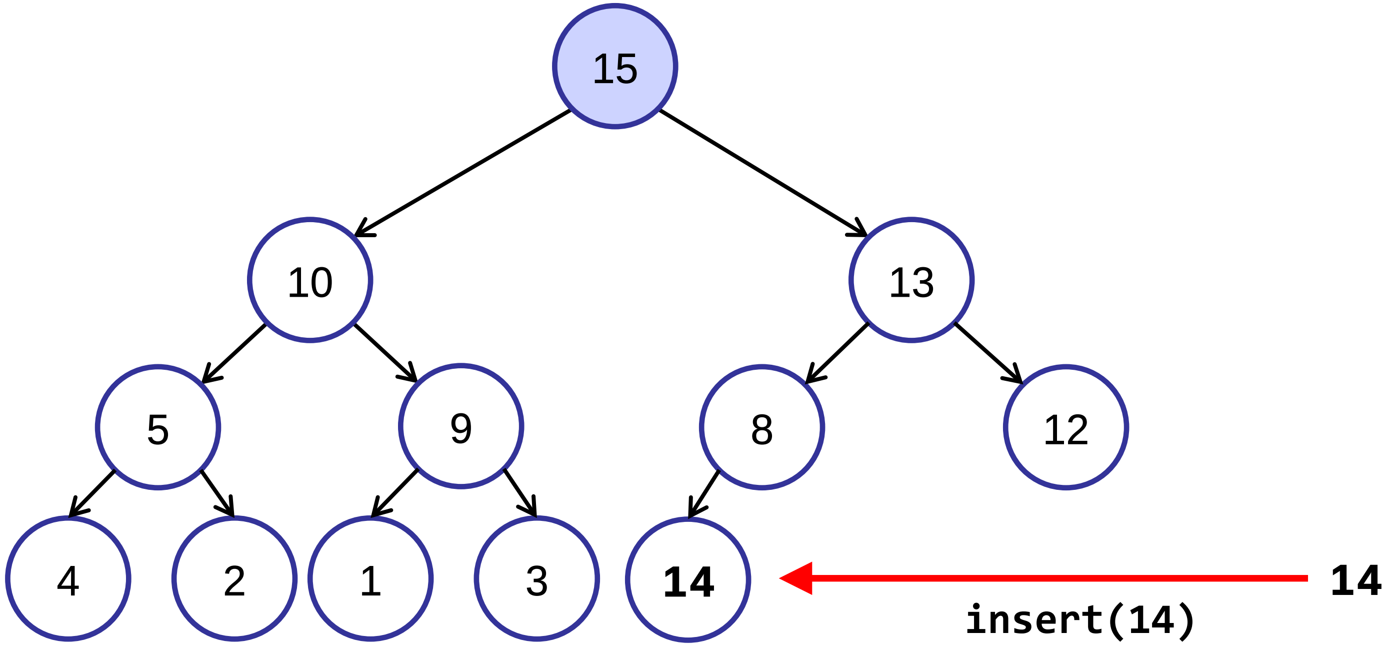
3. Sinkprozess durchführen bzw. **inhaltliche Bedingung** sicherstellen (grösserer Sohn steigt auf): $O(\log_2 n)$



- Damit resultiert diese **Zeitkomplexität** für **`getMax()`**:
  $O(1) + O(1) + O(\log_2 n)$ ➔ **$O(\log_2 n)$**

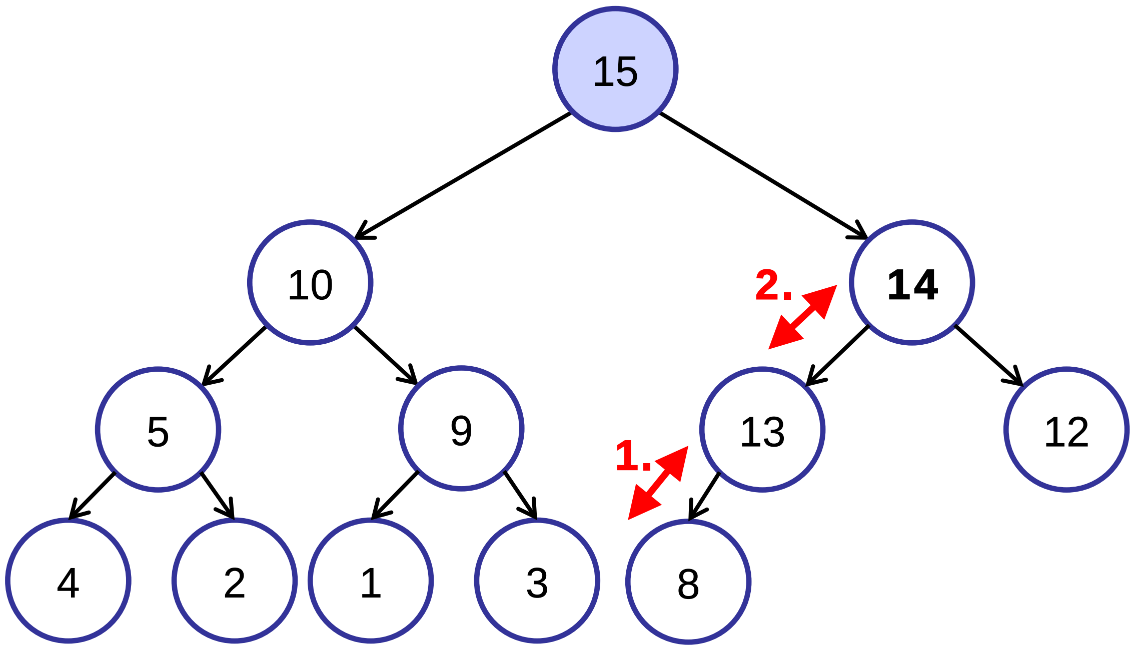
Datenstruktur Heap – `insert()`

- Auch nach dem Einfügen eines neuen Elementes mit **`insert()`**, z.B. `insert(14)` muss der Baum **reorganisiert** werden, damit die Bedingungen für den Heap wieder erfüllt sind:
    1. Neues Element als Blatt unten rechts einfügen bzw. **strukturelle Bedingung** sicherstellen: O(1)



Datenstruktur Heap – `insert()`

2. Steigprozess durchführen bzw. **inhaltliche Bedingung** sicherstellen (vertauschen, falls Vater kleiner): $O(\log_2 n)$



- Damit resultiert diese **Zeitkomplexität** für **`insert()`**:
  $O(1) + O(\log_2 n)$ ➔ $\mathbf{O(\log_2 n)}$

Heapsort
(John William Joseph Williams, 1964)

Heapsort – Prinzip

- Heapsort arbeitet **instabil** und besitzt **generell eine Zeitkomplexität von O(n · log n)**.
- Zur Erinnerung:
  - Beim «direkten Auswählen» sucht man jeweils im unsortierten Bereich nach dem kleinsten (alternativ grössten) Datenelement.
  - Dieses fügt man anschliessend an der richtigen Stelle im bereits sortierten Bereich ein.
- **Heapsort arbeitet im Prinzip gleich**:
  - Das Suchen im unsortierten Bereich beschleunigt man aber wesentlich durch Wahl einer geschickten Datenstruktur (**Heap**).
  - Mit `getMax()` erfordert das **Suchen nur noch O($\log_2$ n)**.
  - Heapsort zeigt das Zusammenspiel von Algorithmen und Datenstrukturen sehr schön auf!

Heapsort – Prinzip

- Algorithmus:

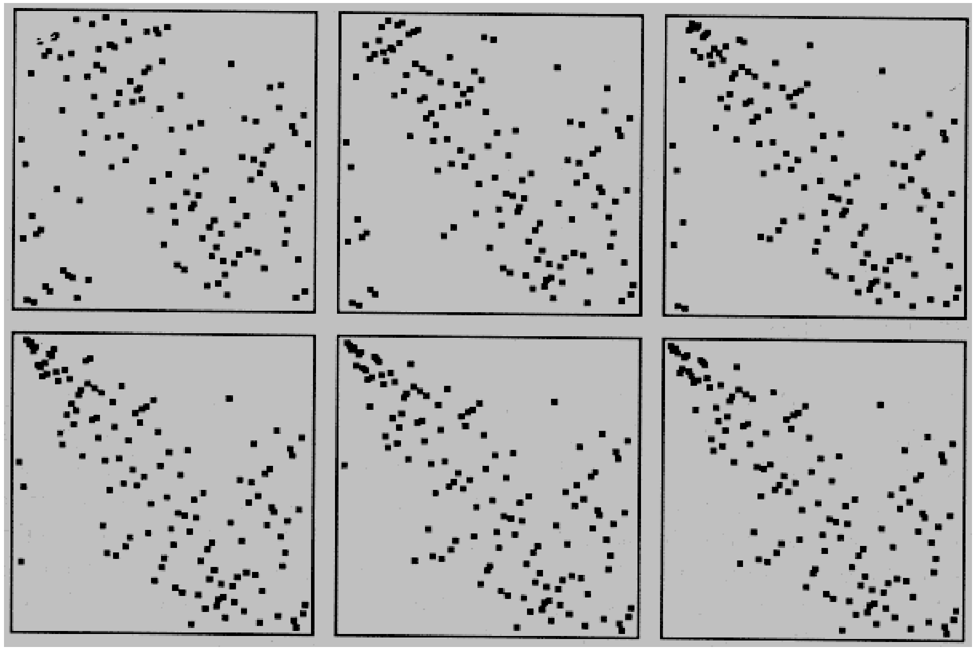
1. Heap **aufbauen**: Man fügt (n-1) Mal mit **`insert()`** ein Element in den Heap ein. ➔ $O(n \cdot \log n)$
2. Elemente **sortieren**: Man entnimmt (n-1) Mal mit **`getMax()`** aus dem schwindenden Heap das grösste Element. Damit resultiert die Sortierung. ➔ $O(n \cdot \log n)$

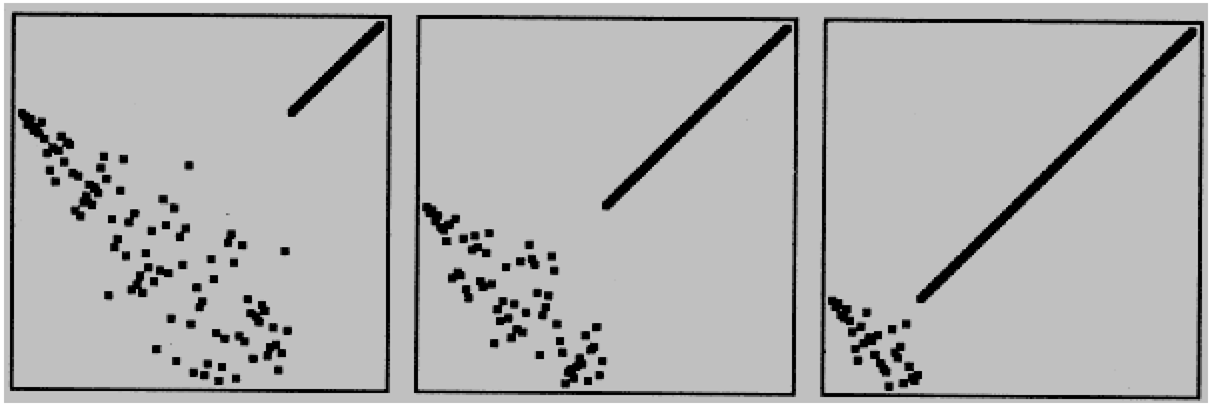
➔ **Zeitkomplexität $O(n \cdot \log n)$**

- Durch geschickte Implementation braucht Heapsort keinen zusätzlichen Speicherplatz.

➔ **Speicherkomplexität $O(1)$**

Heapsort – Prinzip





Ordnung mit der Java Klassenbibliothek

Sortieren vs. geordnete Collections

- Ordnung ermöglicht effizienten Daten-Zugriff:
  - Ordnung schafft man **durch Sortieren**,
  - oder **durch geordnete Datenstrukturen bzw. Collections**.
- Wichtige, typisch zu überschreibende Methoden von `Object`:
  - `boolean` **`equals`**`(Object obj)`
  - `int` **`hashCode`**`()` passend zu `equals(...)`
    - gleiche Objekte → gleicher HashCode
    - gleicher HashCode → nicht zwingend gleiche Objekte!
- Natürliche und spezielle Ordnung:
  - Interface **`Comparable`**`<T>`
    - `int` **`compareTo`**`(T o)` passend zu `equals(...)`
  - Interface **`Comparator`**`<T>`
    - `int` **`compare`**`(T o1, T o2)`

Arrays und Listen sortieren

- **Arrays sortieren:**
  - Klasse `java.util.`**`Arrays`**
    - `static void` **`sort`**`(int[] a)`
    - `static void` **`sort`**`(Object[] a)`
    - `static <T> void` **`sort`**`(T[] a, Comparator<? super T> c)`

- **Listen sortieren:**
  - Klasse `java.util.`**`Collections`**
    - `static <T extends Comparable<? super T>> void` **`sort`**`(List<T> list)`
    - `static <T> void` **`sort`**`(List<T> list, Comparator<? super T> c)`

Nützliche Hilfsmethoden

- **Collection in Array umwandeln:**
  - Interface `java.util.`**`Collection`**`<E>`
    - `Object[ ]` **`toArray`**`()`
    - `<T> T[ ]` **`toArray`**`(T[] a)`

- **Array in Liste umwandeln:**
  - Klasse `java.util.`**`Arrays`**
    - `static <T> List<T>` **`asList`**`(T... a)`

Geordnete Collections bzw. geordnete binäre Bäume ...

- **für ungleiche Objekte**, deshalb Set:
  - Klasse `java.util.`**`TreeSet`**`<E>`
    - **`TreeSet`**`()`
    - **`TreeSet`**`(Comparator<? super E> comparator)`
  - auch Klasse `java.util.`**`Collections`**
    - `static <T> SortedSet<T>` **`synchronizedSortedSet`**`(SortedSet<T> s)`

- **für Key / Value-Paare**, deshalb Map:
  - Klasse `java.util.`**`TreeMap`**`<K,V>`
    - **`TreeMap`**`()`
    - **`TreeMap`**`(Comparator<? super K> comparator)`
  - auch Klasse `java.util.`**`Collections`**
    - `static <K,V> SortedMap<K,V>` **`synchronizedSortedMap`**`(SortedMap<K,V> m)`

Zusammenfassung

- «Teile und Herrsche» ist ein zentrales Lösungsprinzip.
- Quicksort mit seiner Average Case Zeitkomplexität von $O(n \cdot \log n)$ und Varianten haben sich in der Praxis sehr bewährt. «Median of Three», geschickte Behandlung gleicher Datenelemente und geschickte Kombination mit anderen Sortieralgorithmen sind einfache Optimierungen von Quicksort.
- Mergesort ist beinahe das ideale Sortierverfahren: garantierte Zeitkomplexität von $O(n \cdot \log n)$, stabil, parallelisierbar, für internes und externes Sortieren geeignet; aber etwas mehr Speicher.
- Heapsort garantiert ebenfalls eine Zeitkomplexität von $O(n \cdot \log n)$ und lässt sich ohne Rekursion implementieren.
- Grösste oder auch kleinste Datenelemente lassen sich effektiv mit einem Heap verwalten. Zugriffe benötigen $O(\log n)$.

Fragen?