

Softwarepraktikum

2. Frontalveranstaltung

03.11.2017



TECHNISCHE
UNIVERSITÄT
DARMSTADT



ES Real-Time Systems Lab

Prof. Dr. rer. nat. Andy Schürr

Dept. of Electrical Engineering and Information Technology

Dept. of Computer Science (adjunct Professor)

Dr. Malte Lochau

Malte.lochau@es.tu-darmstadt.de

www.es.tu-darmstadt.de

- Lösungsidee Aufgabenblock 1
- Arrays, Collections, Generics
- Comparator
- Rekursion
- Sortieralgorithmus QuickSort
- Aufgabenblock 2

Lösungsvorschlag AB1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
/**
 * Returns all ControllableProducers of the given graph in a List.
 *
 * @param graph {@link Graph}
 * @return {@link List} of {@link ControllableProducer}
 */
@Override
public List<ControllableProducer> getControllableProducers(Graph<EnergyNode, PowerLine> graph) {
    if (graph == null) {
        throw new IllegalArgumentException("Parameter is not allowed to be null.");
    }

    List<ControllableProducer> ret = new ArrayList<ControllableProducer>();
    for (EnergyNode node : graph.getNodes()) {
        if (node instanceof ControllableProducer) {
            ret.add((ControllableProducer) node);
        }
    }
    return ret;
}
```



Lösungsvorschlag AB1



```
@Override
@Test(expected = IllegalArgumentException.class)
public void testGetProducers_Parameters() throws Exception {
    sut.getProducers(null);
}
```

```
@Override
@Test
public void testGetProducers_Parameters() throws Exception {
    try {
        sut.getProducers(null);
        fail();
    } catch (IllegalArgumentException e) {
    }
}
```



Lösungsvorschlag AB1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
@Override
@Test
public void testGetControllableConsumers() throws Exception {
    assertThat("Number of ControllableConsumer is not the same!", sut.getControllableConsumers(graph1).size(), is(3));
}
```

Failure Trace

```
java.lang.AssertionError: Number of ControllableConsumer is not the same!
Expected: is <3>
but: was <0>
at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:18)
at solutions.exercise1.ScenarioUtilTest.testGetControllableConsumers(ScenarioUtilTest.java:79)
```

```
@Override
@Test
public void testGetControllableConsumers() throws Exception {
    assertThat("Number of ControllableConsumer is not the same!", sut.getControllableConsumers(graph1).size(), is(equalTo(3)));
}
```

Failure Trace

```
java.lang.AssertionError: Number of ControllableConsumer is not the same!
Expected: is <3>
but: was <0>
at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:18)
at solutions.exercise1.ScenarioUtilTest.testGetControllableConsumers(ScenarioUtilTest.java:79)
```



Lösungsvorschlag AB1



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
@Override
@Test
public void testGetControllableConsumers() throws Exception {
    assertEquals("Number of ControllableConsumer is not the same!", sut.getControllableConsumers(graph1).size(),3);
}
```

Failure Trace

```
java.lang.AssertionError: Number of ControllableConsumer is not the same! expected:<0> but was:<3>
    at solutions.exercise1.ScenarioUtilTest.testGetControllableConsumers(ScenarioUtilTest.java:79)
```

```
@Override
@Test
public void testGetControllableConsumers() throws Exception {
    assertTrue("Number of ControllableConsumer is not as expected", sut.getControllableConsumers(graph1).size()==3);
}
```

Failure Trace

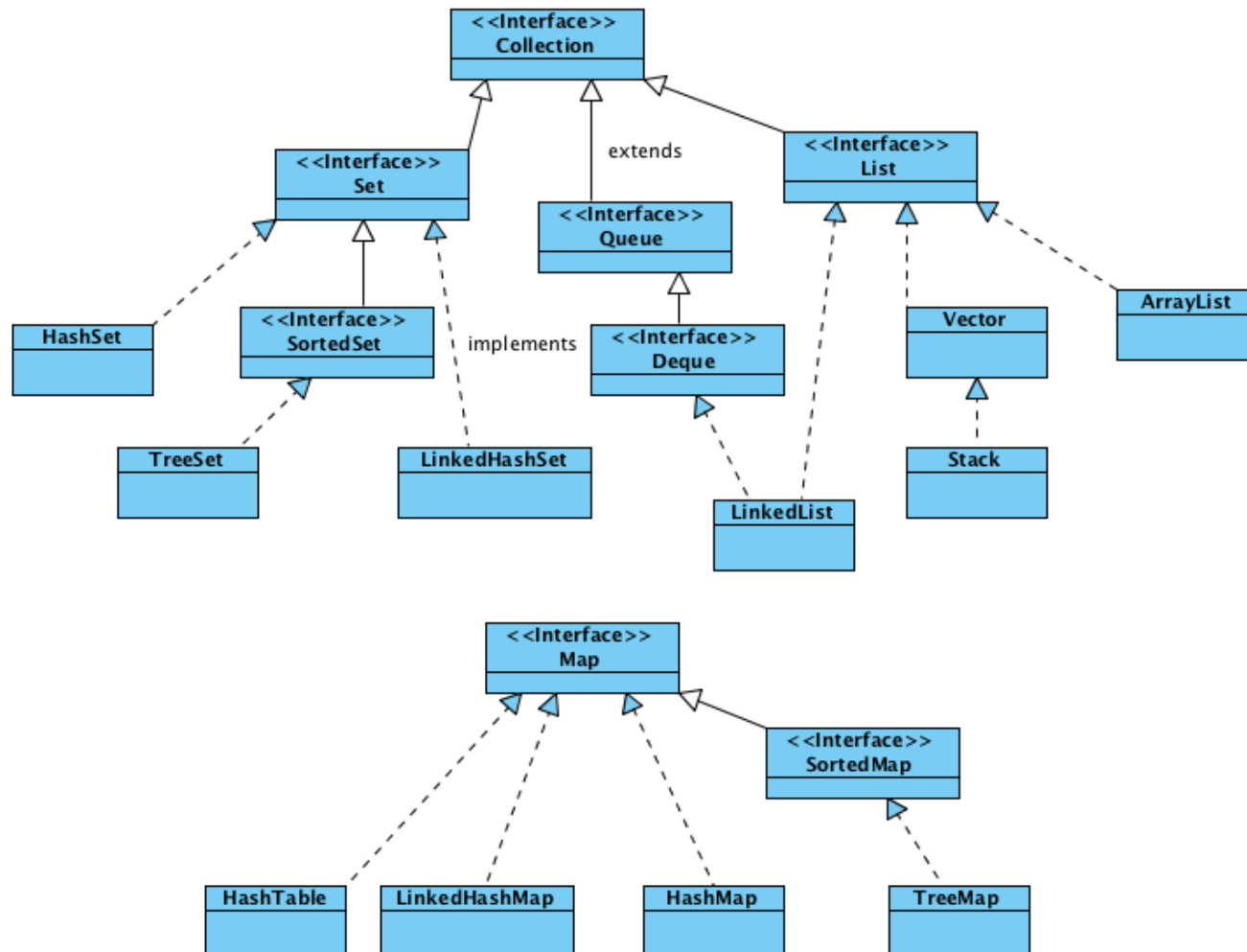
```
java.lang.AssertionError: Number of ControllableConsumer is not as expected
    at solutions.exercise1.ScenarioUtilTest.testGetControllableConsumers(ScenarioUtilTest.java:79)
```



- Lösungsidee Aufgabenblock 1
- Collections
- Comparator
- Rekursion
- Sortieralgorithmus QuickSort
- Aufgabenblock 2

- Eine Kollektion (Collection) ist ein **Objekt**, welches mehrere **gleichartige Elemente** in einer Einheit verwaltet
- Das *Java Collection Framework* ist eine Architektur zur **Repräsentation** und **Manipulation** von Kollektionen
- Das Framework enthält:
 - Schnittstellen (Interfaces)
 - Abstrakte Implementierungen
 - Implementierungen
 - Algorithmen
 - ...

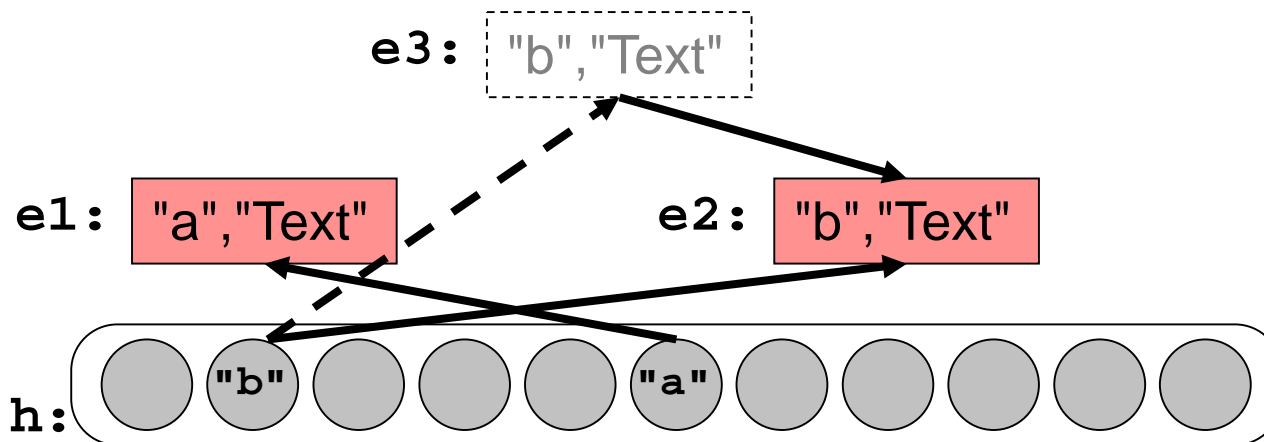
Java Collection Framework



Beispiel: HashTable

Beispiel: `java.util.HashMap` (für beliebige Objekte)

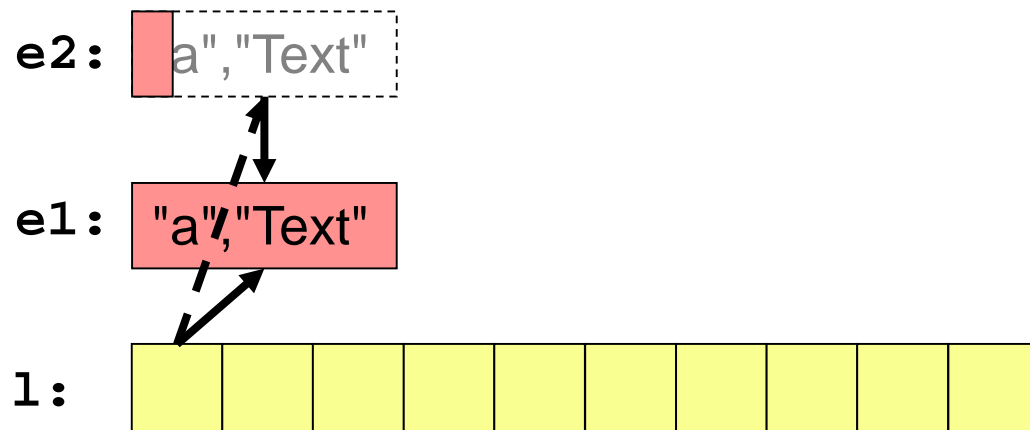
```
HashMap h = new HashMap();  
MyEntry e1 = new MyEntry("a", "Text");  
h.put(e1.getKey(), e1);  
MyEntry e2 = new MyEntry("b", "Text");  
h.put(e2.getKey(), e2);  
MyEntry e3 = (MyEntry) h.get("b");
```



Beispiel: ArrayList

Beispiel: `java.util.ArrayList` (für beliebige Objekte)

```
ArrayList l = new ArrayList();  
MyEntry e1 = new MyEntry("a", "Text");  
l.add(e1);  
MyEntry e2;  
e2 = (MyEntry) l.firstElement();  
l.remove(e1);
```



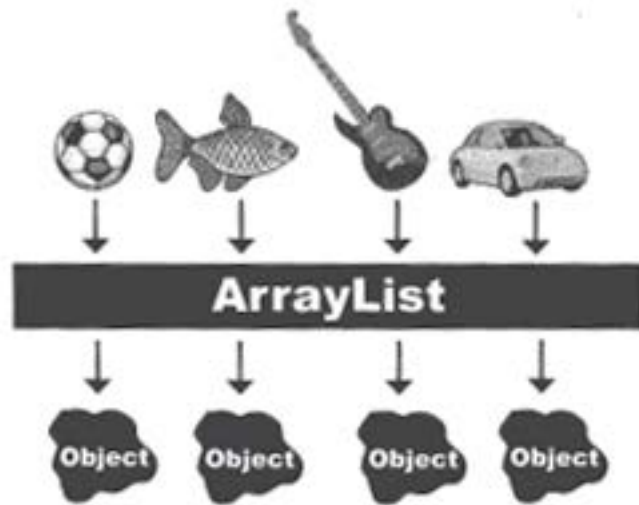
Arrays vs. Collections

```
Object[] objAr    = new String[10]; // no compile error
objAr[0]          = new Object();   // ArrayStoreException
objAr[0]          = "a";            // no Exception
```

```
LinkedList[] objLst = new LinkedList String[10]; // compile error
objLst[0].add(new Object);                        // unreachable

LinkedList[] objLst = new LinkedList[10];         // no error
```





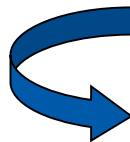
Ohne Generics:

Typunsichere Collections, Fehler treten erst zur Laufzeit auf!



Mit Generics:

Typsichere Collections, Fehler lassen sich schon zur Übersetzungszeit abfangen!



In früheren Versionen von *Java* gab es keine Möglichkeit, den Typ von Objekten einer Kollektion festzulegen.

Die Kollektionen arbeiteten mit Objekten des Typs `object`.
Dadurch war immer ein (fehleranfälliges) *casting* notwendig.

```
List aList = new ArrayList();
aList.add("ES");
aList.add(new Integer(18));
aList.add(new StringBuffer());

Iterator it = aList.iterator();
while (it.hasNext()) {
    String element = (String) it.next(); Laufzeitfehler
    System.out.println(element);
}
```

Beispiel: Generische ArrayList

Seit *Java 5* gibt es die Möglichkeit, den Typ von Objekten einer Kollektion festzulegen, d.h. die Kollektionen enthalten Typparameter.

Die Angabe des Typs erfolgt innerhalb `< >`.

```
List<String> aList = new ArrayList<String>();
aList.add("ES");
aList.add(new Integer(18));
aList.add(new StringBuffer());
} Fehler wird vom
  Compiler erkannt!

Iterator it = aList.iterator();
while (it.hasNext()) {
    String element = it.next();
    System.out.println(element);
}
```



Beispiel: Generische HashMap

Beispiel: Verwendung der Kollektion **Map**.

Diese Kollektion hat zwei Typparameter, da sie eine Abbildung von einem Definitionsbereich in einen Wertebereich realisiert.

```
Map<String, String> aMap;  
aMap = new HashMap<String, String>();  
  
aMap.put("one", "eins");  
aMap.put("two", "zwei");  
aMap.put("three", "drei");  
  
String german = aMap.get("one");
```

Die Werte einer **Map** können selbst auch Kollektionen sein.

```
Map<String, List<Integer>> aMap;  
aMap = new HashMap<String, List<Integer>>();
```




For-Schleife vs. Foreach-Schleife

```
int[] arr = { 1, 2, 3, 4 };

for (int i=0; i < arr.length; i++) {
    System.out.println(arr[i]);
}

for (int elem : arr) {
    System.out.println(elem);
}
```

```
HashSet<String> hs = new HashSet<String>(); hs.add("eins", "zwei", "drei", "vier");

for (Iterator<String> it = hs.iterator(); it.hasNext(); ) {
    System.out.println(it.next());
}

for (String elem : hs) {
    System.out.println(elem);
}
```



- Lösungsidee Aufgabenblock 1
- Collections
- Comparator
- Rekursion
- Sortieralgorithmus QuickSort
- Aufgabenblock 2

Vergleichen von Objekten

Sollen Objekte eines Typs miteinander verglichen werden, muss eine Ordnung (Vergleichskriterium) auf diesem Typ definiert werden.

In Java gibt es zwei Schnittstellen zur Bestimmung der Ordnung:

- **Interface Comparable**
 - Objekte einer Klasse, die das Interface **Comparable** implementiert, können sich selbst mit anderen Objekten vergleichen
 - Verwendung bei Objekten mit natürlicher Ordnung (z.B. String, Date)
- **Interface Comparator**
 - Objekte einer Klasse, die das Interface **Comparator** implementiert, können zwei Objekte gleichen Typs entgegennehmen und miteinander vergleichen
 - Verwendung bei Objekten, deren Ordnung durch mehrere komplexe Vergleichskriterien gegeben ist (z.B. PlayfieldElement, Producer)



interface java.lang.Comparable<T>

- Methode **int** `compareTo(T o)` zum Vergleich des Objektes selbst mit einem anderen Objekt
- Der Aufruf **int** `result = o1.compareTo(o2)` liefert
 - `result > 0` falls `o1` größer als `o2` ist
 - `result == 0` falls `o1` gleich `o2` ist
 - `result < 0` falls `o1` kleiner als `o2` ist
- Die Aufrufe `o1.compareTo(o2) == 0` und `o1.equals(o2)` sollen konsistente Ergebnisse liefern
- `==` vergleicht Referenzen und `equals()` ganze Objekte

interface `java.util.Comparator<T>`

- Methode **int** `compare(T o1, T o2)` zum Vergleich zweier Objekte
 - Konvention für Rückgabewert, siehe `compareTo`
- ACHTUNG: die Methode **boolean** `equals(Object obj)` prüft, ob zwei beliebige Objekte (!) gleich sind

Beispiel: Vergleich von Räumen

```
class Room {  
    int qm;  
    Room(int qm) {  
        this.qm = qm;  
    }  
}
```

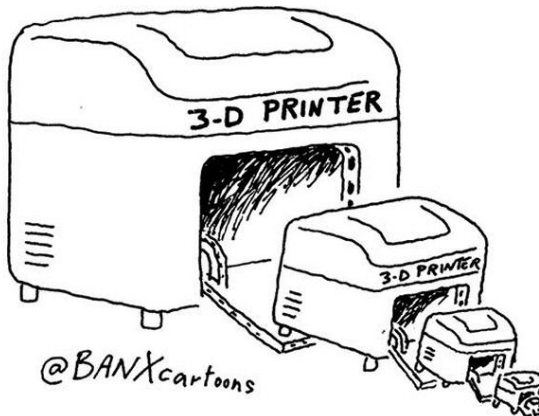
```
class RoomComparator implements Comparator<Room> {  
    @Override  
    public int compare(Room room1, Room room2) {  
        return room1.qm - room2.qm;  
    }  
}
```

- Ein **Comparator**-Objekt kann zum Sortieren einer Kollektion übergeben werden

```
List<Room> list = ...  
Collections.sort(list, new RoomComparator());
```

- Lösungsidee Aufgabenblock 1
- Collections
- Comparator
- Rekursion
- Sortieralgorithmus QuickSort
- Aufgabenblock 2

Rekursion vs. Iteration

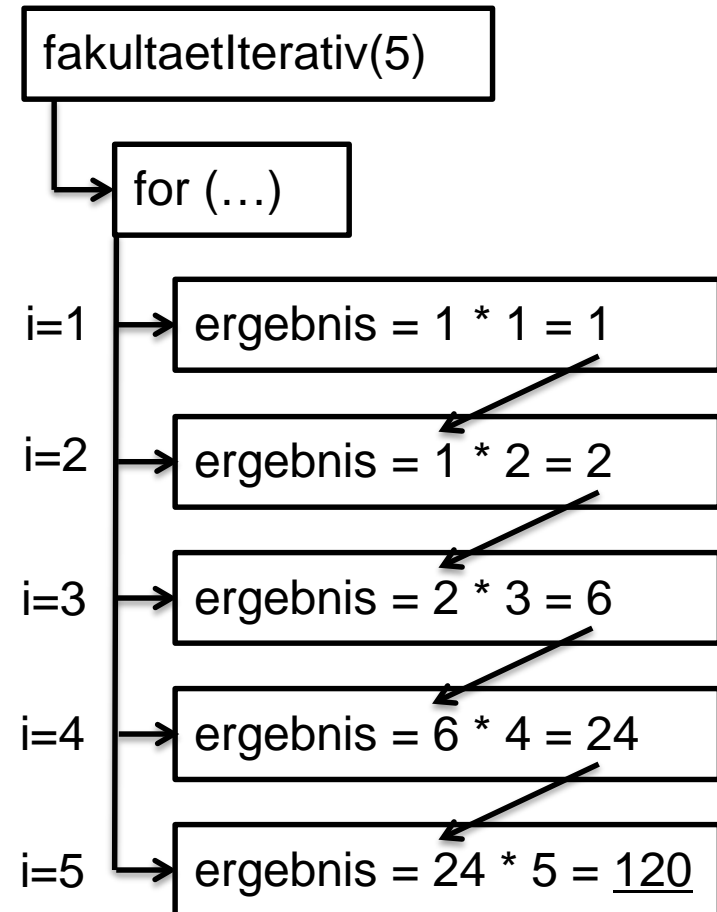


- Rekursion: Geschachtelter Aufruf einer Methode auf den Teilen eines Problems
- Iteration: Schleife über die Teile eines Problems

Beispiel: Iterative Fakultätsberechnung

Beispiel: $5! = 1 * 2 * 3 * 4 * 5 = \underline{120}$

```
/**
 * Iterative Berechnung des Fakultätswertes.
 *
 * @param n Eingabewert
 * @return Fakultätswert von n
 */
static int fakultaetIterativ(int n) {
    int ergebnis = 1;
    for (int i = 1; i <= n; i++) {
        ergebnis = ergebnis * i;
    }
    return ergebnis;
}
```



Beispiel: Rekursive Fakultätsberechnung

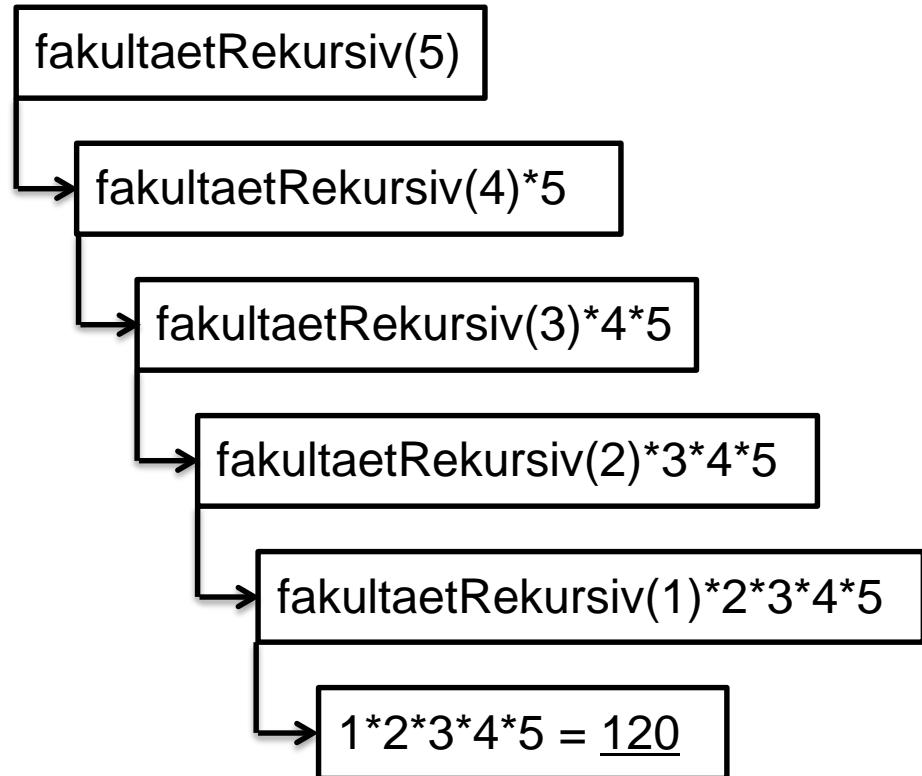


TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
/**
 * Rekursive Berechnung des Fakultätswertes.
 *
 * @param n Eingabewert
 * @return Fakultätswert von n
 */
static int fakultaetRekursiv(int n) {
    if (n <= 1)
        return 1;
    else
        return fakultaetRekursiv(n-1) * n;
}
```

Rekursionsende

Rekursionsschritt:
Aufruf der Methode
durch sich selbst



Rekursion vs. Iteration

- Rekursive Lösungen basieren auf dem Prinzip „Teile und Herrsche“, iterative Lösungen verwenden „dynamische Programmierung“
 - „Teile und Herrsche“: ein Problem wird in gleichartige Teile zerlegt, die Teile werden separat voneinander gelöst und zu einem Gesamtergebnis zusammengefügt.
 - „Dynamische Programmierung“: Ein Problem wird als Ganzes behandelt, indem die Lösung iterativ durch gleichartige Verfeinerungsschritte berechnet wird.
- Rekursive Lösungen führen meistens zu eleganterem Code, erscheinen dafür aber häufig komplizierter und benötigen mehr Speicher

Satz: Zu jeder iterativen Lösung eines Problems existiert auch eine rekursive Lösung und umgekehrt.



Gliederung

- Lösungsidee Aufgabenblock 1
- Collections
- Comparator
- Rekursion
- Sortieralgorithmus QuickSort
- Aufgabenblock 2



Sortierprobleme

Gegeben: Eine Liste Objekte gleichen Typs mit einem Vergleichskriterium

Gesucht: Geänderte Anordnung der Objekte in der Liste, sodass die „besseren“ Elemente vor den „schlechteren“ positioniert sind

Beispiel EVS:

- Die Liste aller Umspannwerke, sortiert hinsichtlich ihrer Eignung für den Bau eines Kraftwerkes
- Die Liste aller Leitungen, sortiert hinsichtlich ihrer Eignung für den Ausbau



- Verändern der Position von Elementen in einer Liste hinsichtlich eines bestimmten **Vergleichskriteriums**, das eine (totale) Ordnung auf diesen Elementen definiert.
- Sortierung kann **aufsteigend** (kleinstes Element steht an erster Position) oder **absteigend** (kleinstes Element steht an letzter Position) erfolgen.
- Beispiel für die aufsteigende Sortierung von Integer-Elementen:

Position	0	1	2	3	4	5	6	7
Daten	-5	2	9	9	18	46	110	237


- Beispiel für die absteigende Sortierung von Integer-Elementen:

Position	0	1	2	3	4	5	6	7
Daten	237	110	46	18	9	9	2	-5




- Elemente der Originalliste werden solange vertauscht, bis die Liste sortiert ist (**In-Place-Sortierung**)

Position	0	1	2	3	4	5	6	7
Daten	-5	2	9	9	110	237	18	46



- Elemente der Originalliste werden in der richtigen Reihenfolge in eine neue Liste eingefügt (**Out-of-Place-Sortierung**)

Position	0	1	2	3	4	5	6	7
Daten	-5	2	9	9	110	237	18	46



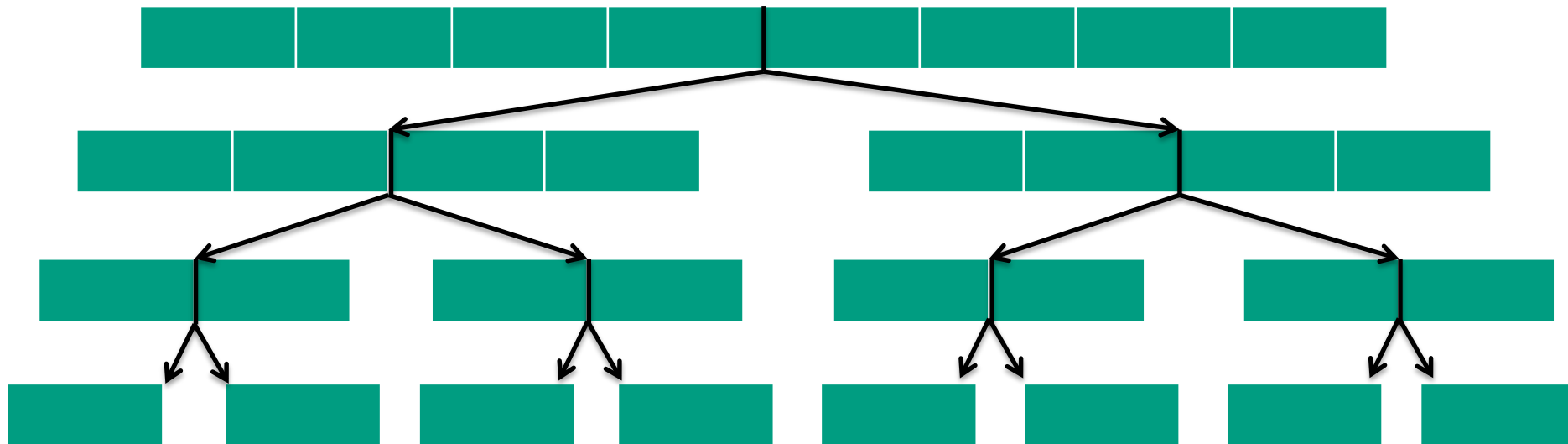
Position	0	1	2	3	4	5	6	7
Daten	-5	2	9	9	18	46	110	237

QuickSort – Grundlagen

- Ca. 1960 entwickelt
- In-Place Sortieralgorithmus
- Prinzip des „Teile & Herrsche“

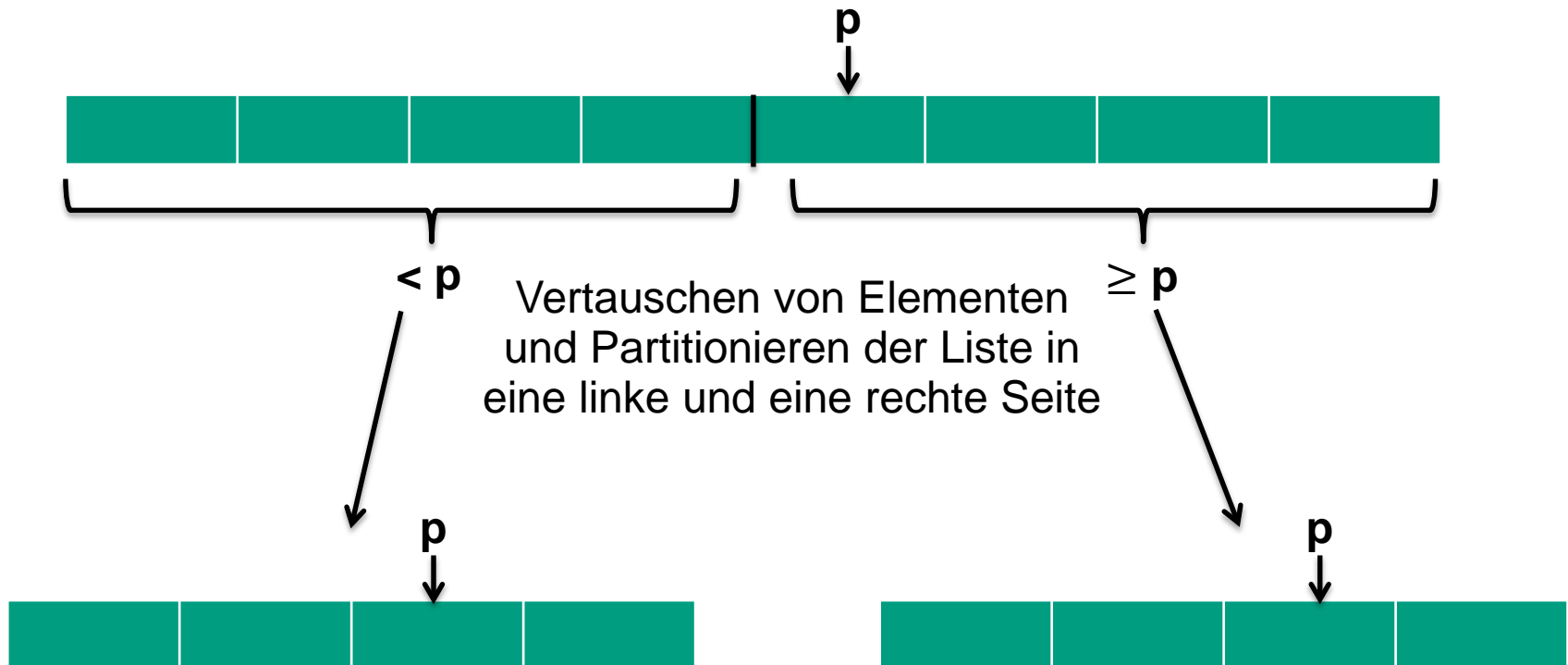


C.A.R. Hoare



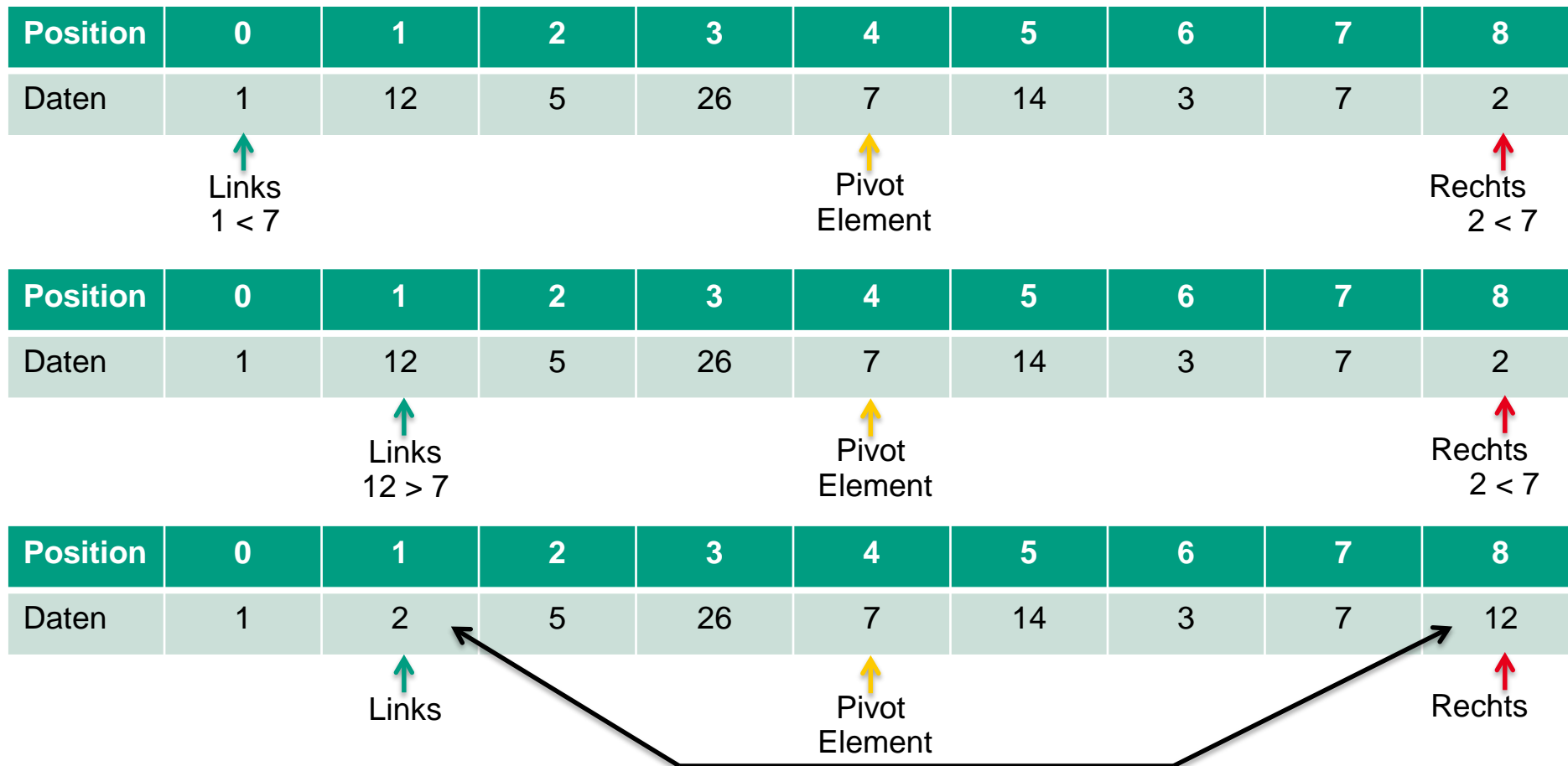
QuickSort – Grundprinzip

Wahl eines beliebigen Pivot-
Elements aus der Liste

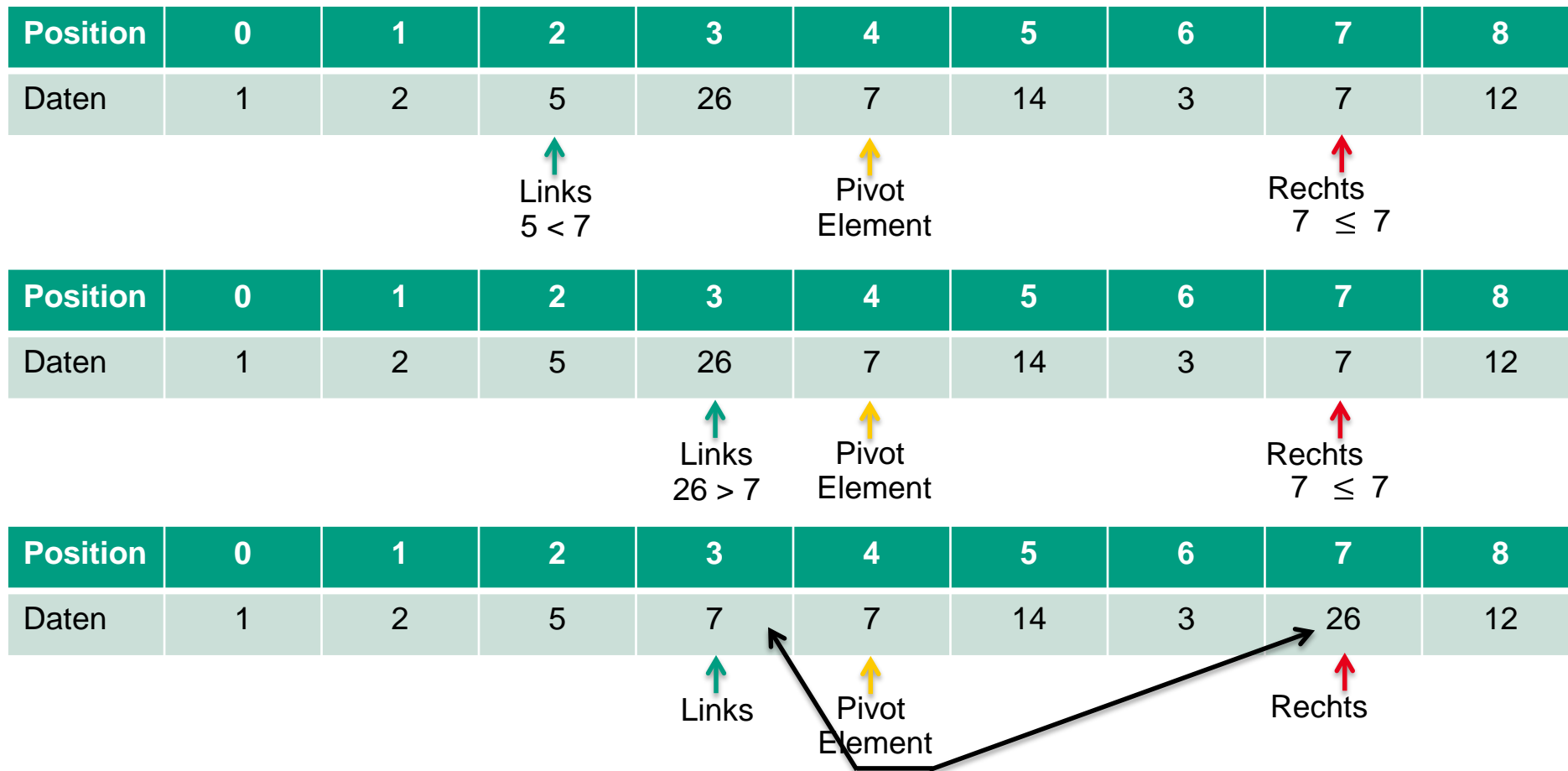


Rekursiver Aufruf dieser Prozedur auf beiden Teillisten,
bis Teillisten nur noch ein Element enthalten

QuickSort – Beispiel



QuickSort – Beispiel



QuickSort – Beispiel

Position	0	1	2	3	4	5	6	7	8
Daten	1	2	5	7	7	14	3	26	12

Diagram illustrating the initial partitioning step of QuickSort. The array is [1, 2, 5, 7, 7, 14, 3, 26, 12]. The pivot element is 7 at index 4. The left pointer (Links) is at index 4, and the right pointer (Rechts) is at index 6. The comparison $3 < 7$ is shown, indicating that the element at index 6 is less than the pivot.

Position	0	1	2	3	4	5	6	7	8
Daten	1	2	5	7	3	14	7	26	12

Diagram illustrating the partitioning step of QuickSort. The array is [1, 2, 5, 7, 3, 14, 7, 26, 12]. The pivot element is 3 at index 4. The left pointer (Links) is at index 4, and the right pointer (Rechts) is at index 6. The comparison $7 \geq 3$ is shown, indicating that the element at index 6 is greater than or equal to the pivot. Arrows indicate the movement of elements: the element 7 at index 6 is moved to index 5, and the element 3 at index 4 is moved to index 6.

QuickSort – Beispiel

Position	0	1	2	3	4	5	6	7	8
Daten	1	2	5	7	3	14	7	26	12

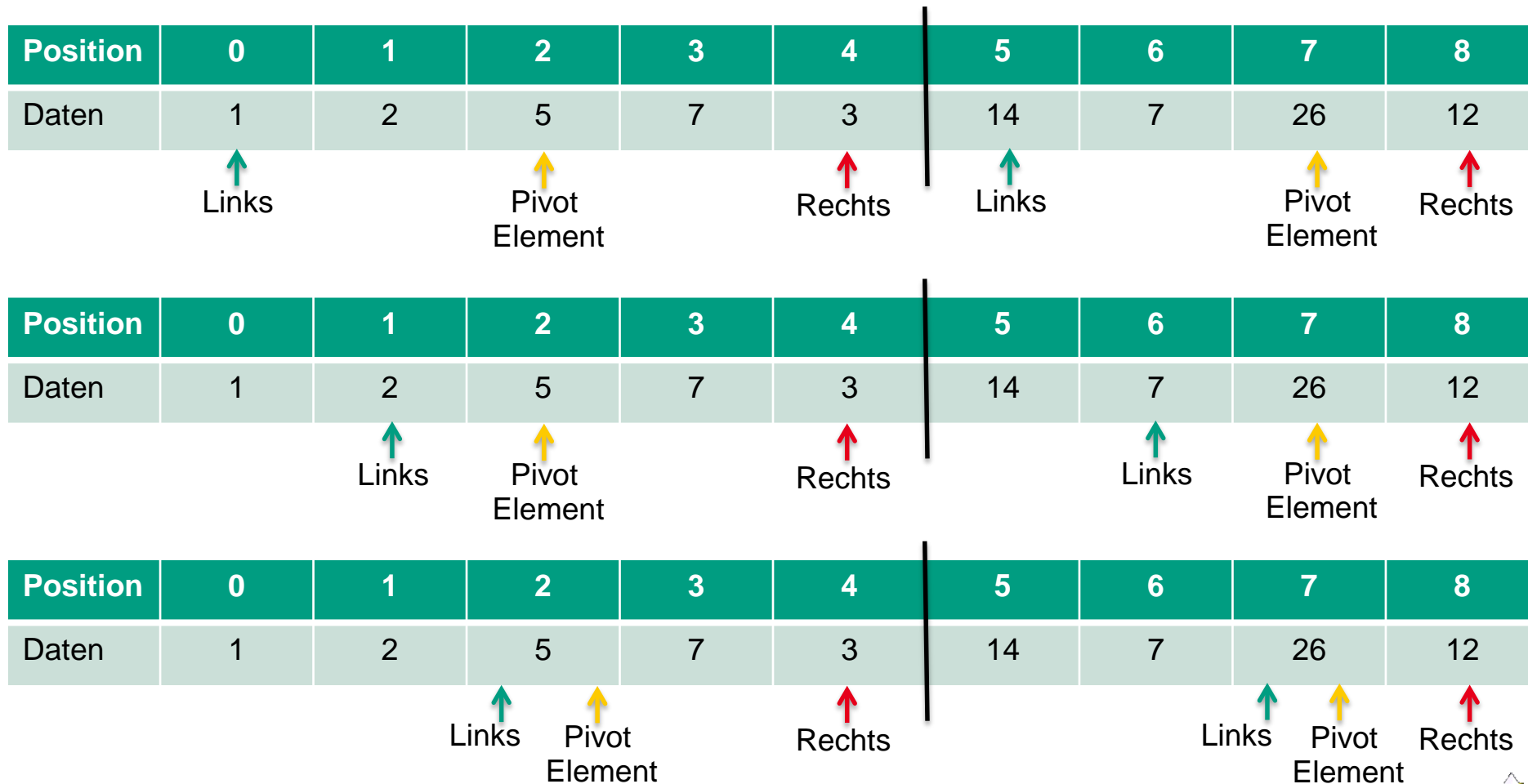
↑ Links ↑ Rechts ↑ Pivot Element

$3 < 7$ $14 > 7$

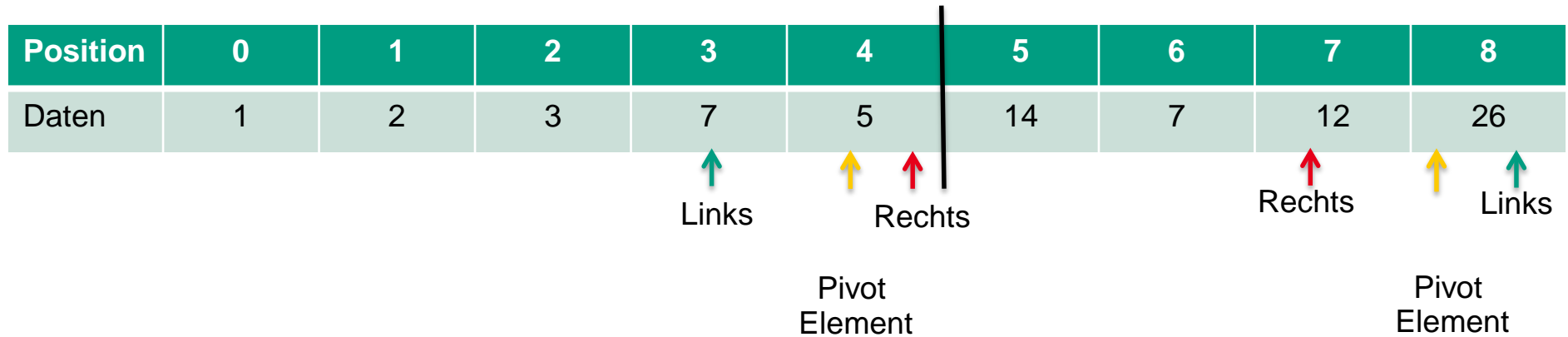
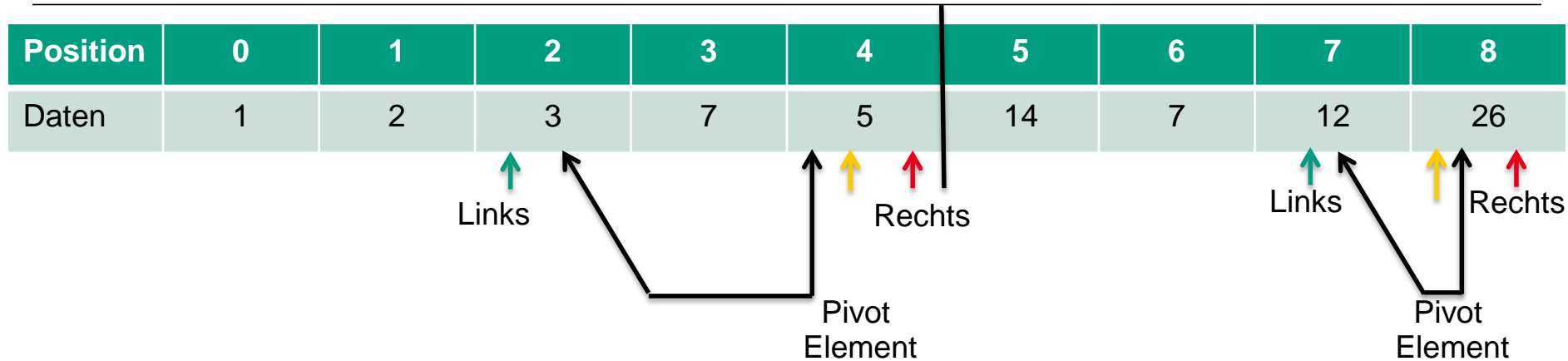
Position	0	1	2	3	4	5	6	7	8
Daten	1	2	5	7	3	14	7	26	12

↑ Rechts < ↑ Links ↑ Pivot Element

QuickSort – Beispiel



QuickSort – Beispiel



QuickSort – Beispiel

Position	0	1	2	3	4	5	6	7	8
Daten	1	2	3	5	7	14	7	12	26

Diagram illustrating the partitioning step of QuickSort. A vertical line separates the array into two parts. The pivot element is 7 at position 4. Elements less than the pivot (5, 3, 2, 1) are moved to the left. Elements greater than the pivot (14, 12, 26) are moved to the right. The array is partitioned into two sub-arrays: [1, 2, 3, 5] and [14, 7, 12, 26].

Links: 5, 3, 2, 1
 Rechts: 14, 7, 12, 26
 Pivot Element: 7

Position	0	1	2	3	4	5	6	7	8
Daten	1	2	3	5	7	14	7	12	26

The array is partitioned into two sub-arrays: [1, 2, 3, 5] and [14, 7, 12, 26]. The pivot element 7 is at position 4. The array is partitioned into two sub-arrays: [1, 2, 3, 5] and [14, 7, 12, 26].

Links: 5, 3, 2, 1
 Rechts: 14, 7, 12, 26
 Pivot Element: 7



QuickSort – Implementierung

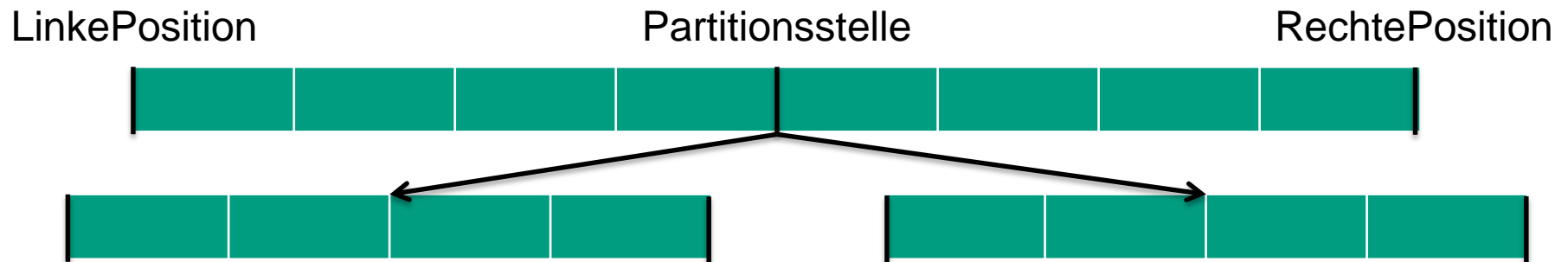
Methode Quicksort(LinkePosition, RechtePosition)

falls LinkePosition < RechtePosition

 Partitionsstelle := Partitioniere(LinkePosition, RechtePosition)

 Quicksort(LinkePosition, Partitionsstelle-1)

 Quicksort(Partitionsstelle, RechtePosition)



QuickSort – Implementierung

Methode Partitioniere(LinkePosition, RechtePosition)

Links := LinkePosition

Rechts := RechtePosition

Pivot-Position := (LinkePosition+RechtePosition)/2

Pivot := Daten[Pivot-Position]

solange Links <= Rechts

solange Daten[Links] < Pivot
 Links+1

solange Daten[Rechts] > Pivot
 Rechts-1

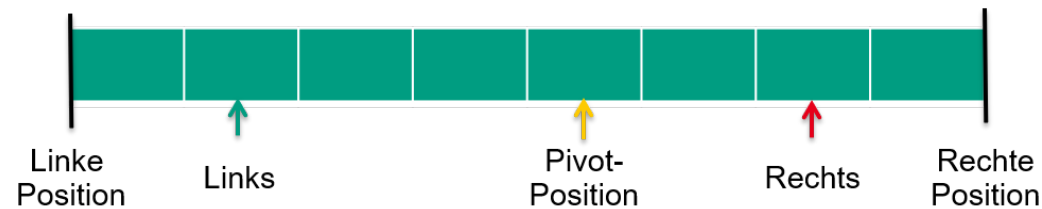
falls Links <= Rechts

 Tausche Daten[Links] mit Daten[Rechts]

 Links+1

 Rechts-1

Partitionsstelle := Links



QuickSort – Eigenschaften

Die Laufzeit des Algorithmus wird im Wesentlichen bestimmt durch die Anzahl n der Listenelemente und die Wahl des Pivot-Elementes

- Worst-Case Komplexität: $O(n^2)$
- Best/Average-Case Komplexität: $\Omega(n \log n)$

Ein interaktives Beispiel zu verschiedenen Implementierungen des QuickSort-Algorithmus finden Sie [hier](#)



- Lösungsidee Aufgabenblock 1
- Collections
- Comparator
- Rekursion
- Sortieralgorithmus QuickSort
- Aufgabenblock 2

Aufgabenblock 2

- 2.1. **ConstructionCostCalculator** implementieren
- 2.2. **PlayFieldElementComparator** implementieren
- 2.3. **QuickSort** implementieren
- 2.4. **QuickSort** testen

