

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- Mergesort
- Stabile Sortiervverfahren
- Fächersortieren
- Externes 2-Wege-Mergesort
- Generische Sortiermethoden
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

Problemstellung

- Umordnung einer Folge von Datensätze (in der Regel in einem Feld abgespeichert)

$a[0], a[1], \dots, a[n-1]$

so, dass

$a[0] \leq a[1] \leq \dots \leq a[n-1]$.

- Einfachheitshalber werden in den hier vorgestellten Verfahren int-Felder sortiert.

```
public static void sort(int[] a) {  
  
    ...  
  
    assert isSorted(a);  
}  
  
private static boolean isSorted(int[] a) {  
    for (int i = 0; i < a.length-1; i++)  
        if (a[i+1] < a[i])  
            return false;  
    return true;  
}
```

Prüfung ob Sortierung korrekt.

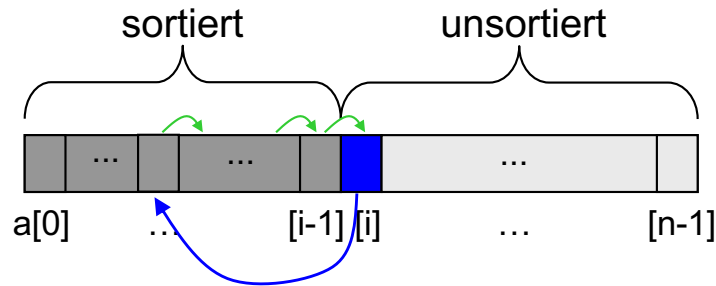
Prüfung ist optional.
Daher assert.

assert muss mit der
JVM-Option „-ea“
aktiviert werden.

Varianten

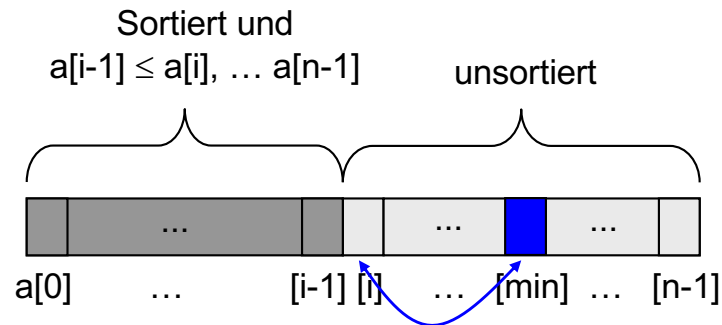
- Feld von Datensätzen nach einem Schlüssel sortieren:
 - Beispiel: Personendaten, die nach dem Familienname als Schlüssel lexikographisch sortiert werden sollen.
 - Ordnungsrelation für Datensätze notwendig.
 - Lösung mit **generischer Sortiermethode**:
`<T extends Comparable<T>> void sort(T[] a)`
`<T> void sort(T[] a, Comparator<T> cmp)`
- Sortieren von linear verketteten Listen
- Sortierung von Dateien (externe Sortierverfahren)

Sortieren durch Einfügen



```
public static void insertionSort(int[] a) {  
  
    for (int i = 1; i < a.length; i++) {  
        // Fuege a[i] in a[0]... a[i-1]  
        // an der richtigen Stelle ein:  
        int v = a[i];  
        int j = i - 1;  
        while (j >= 0 && a[j] > v) {  
            a[j+1] = a[j];  
            j--;  
        }  
        a[j+1] = v;  
    }  
}
```

Sortieren durch Auswählen



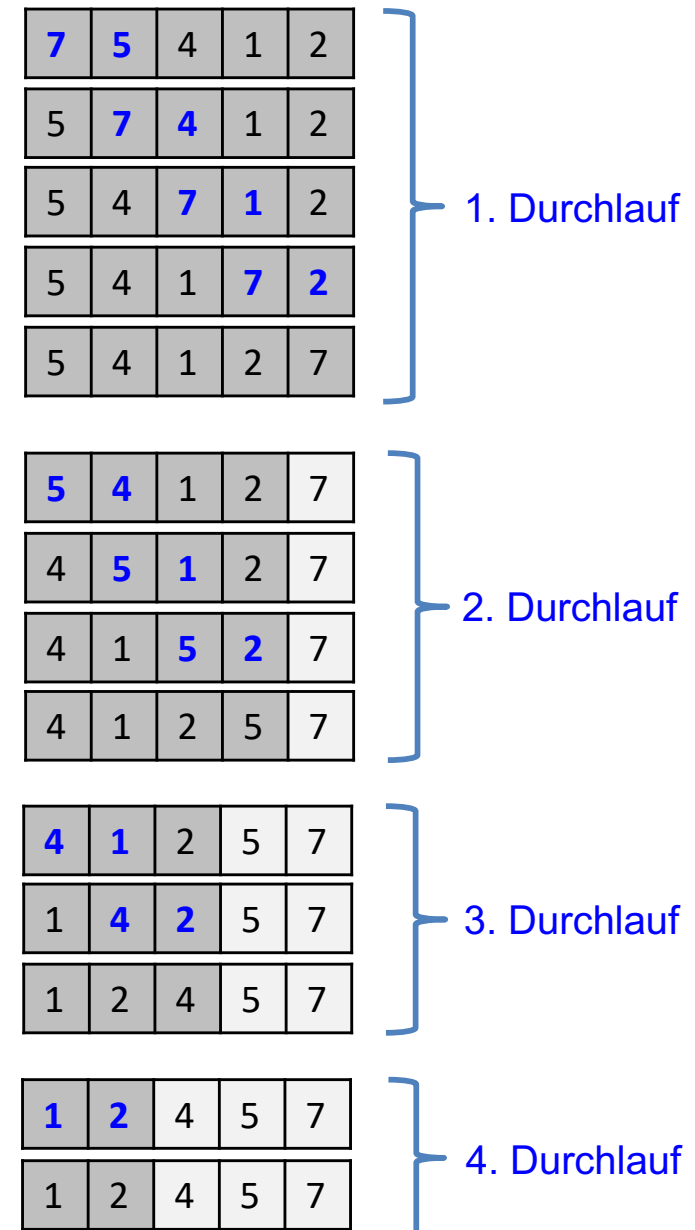
```
public static void selectionSort(int[] a) {  
    for (int i = 0; i < a.length-1; i++) {  
        // Auswahl des kleinsten Elements a[min]  
        // aus a[i], a[i+1], ... a[a.length-1]:  
        int min = i;  
        for (int j = i+1; j < a.length; j++)  
            if (a[j] < a[min])  
                min = j;  
        if (min != i)  
            swap(a, i, min);  
    }  
}
```

```
public static void swap(int[] a, int i, int j) {  
    int t = a[i];  
    a[i] = a[j];  
    a[j] = t;  
}
```

swap vertauscht $a[i]$ mit $a[j]$.
swap wird auch bei anderen
Sortierverfahren benutzt.

Sortieren durch Vertauschen (Bubble Sort)

```
public static void bubbleSort (int[] a) {  
  
    boolean vertauscht;  
    for (int i = a.length-1; i >= 1; i--) {  
        vertauscht = false;  
        // Durchlauf des Felds von 0 bis i:  
        for (int j = 0; j < i; j++) {  
            if (a[j] > a[j+1]) {  
                swap(a, j, j+1);  
                vertauscht = true;  
            }  
        }  
        if (!vertauscht)  
            break;  
    }  
}
```



Analyse der einfachen Sortiervverfahren

Laufzeiten

Sortiervverfahren	Best-Case	Average-Case	Worst-Case
insertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
selctionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$

Laufzeitmessungen

Sortierverfahren	n = 2000	n= 4000	n = 6000	n = 8000	n = 10000
insertionSort	0.77	2.63	5.86	10.41	16.24
selectionSort	4.66	18.31	41.08	72.85	114.87
bubbleSort	5.61	23.16	52.63	95.08	149.89

Zeiten in msec

Messbedingungen

- Die CPU-Zeiten sind in msec angegeben und wurden auf einem IMac 2.8 GHz Intel Core 2 Duo und NetBeans 6.8 gemessen.
- Die Zeitmessungen wurden für 30 zufällig initialisierte int-Felder mit den verschiedenen Sortierverfahren sortiert und anschließend die Zeiten gemittelt.

Aufgabe 9.1 - Quadratische Laufzeiten

- Die CPU-Zeiten bestätigen die in der Analyse ermittelten quadratischen Laufzeiten. Warum?
- Schätzen Sie die zu erwartende CPU-Zeit von BubbleSort für $n = 10^5$ und $n = 10^6$ ab.

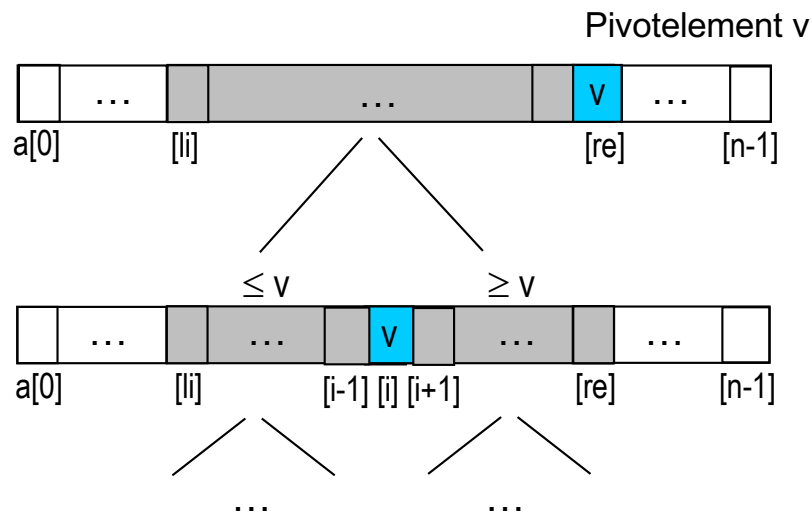
Sortier- verfahren	n = 2000	n = 4000	n = 6000	n = 8000	n = 10^4	n = 10^5	n = 10^6
insertionSort	0.77	2.63	5.86	10.41	16.24		
selectionSort	4.66	18.31	41.08	72.85	114.87		
bubbleSort	5.61	23.16	52.63	95.08	149.89		

Zeiten in msec

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- Mergesort
- Stabile Sortiervverfahren
- Fächersortieren
- Externes 2-Wege-Mergesort
- Generische Sortiermethoden
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

Quicksort ist ein Teile-und-Herrsche-Verfahren



- Aufteilung des (Teil)Felds wird durch Umordnen der Elemente erreicht.
- Nach dem Umordnen sind alle Elemente in linker bzw. rechter Hälfte kleiner gleich bzw. größer gleich dem Pivotelement v .
- QuickSort wurde 1962 von Hoare entwickelt und ist das schnellste Sortierverfahren im mittleren Fall

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

public static void quickSort(int[] a, int li, int re) {
    // Sortiert Teilfeld a[li], a[li+1], ..., a[re].

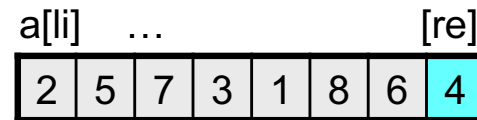
    if (re > li) {
        // Teileschritt:
        int i = partition(a, li, re);

        // Herrscheschritt:
        quickSort(a, li, i-1);
        quickSort(a, i+1, re);
    }
}

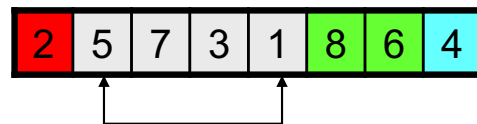
private static int partition(int[] a, int li, int re) {
    // Ordne Elemente in a[li], a[li+1], ..., a[re]
    // mit v = a[re] (Pivotelement) so um, dass gilt:
    // links von a[i] sind alle Elemente  $\leq v$  und
    // a[i] = v und
    // rechts von a[i] sind alle Elemente  $\geq v$ .
    return i;
}
```

Umordnung der Elemente im Teileschritt (1)

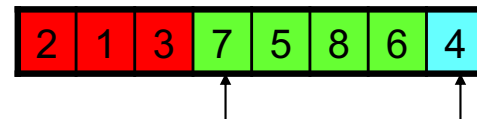
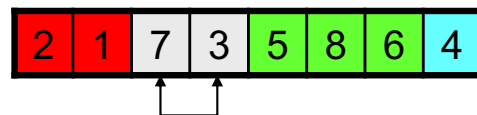
Beispiel



Pivotelement ist $v = a[re] = 4$



Vertausche



Pivotelement mit mittlerem Element vertauschen

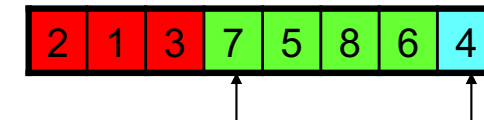
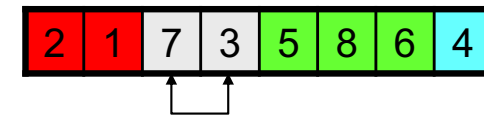
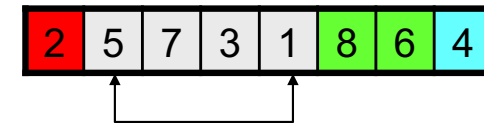
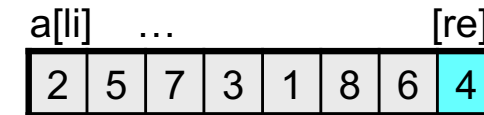


Elemente \leq Pivotelement

Elemente \geq Pivotelement

Umordnung der Elemente im Teileschritt (2)

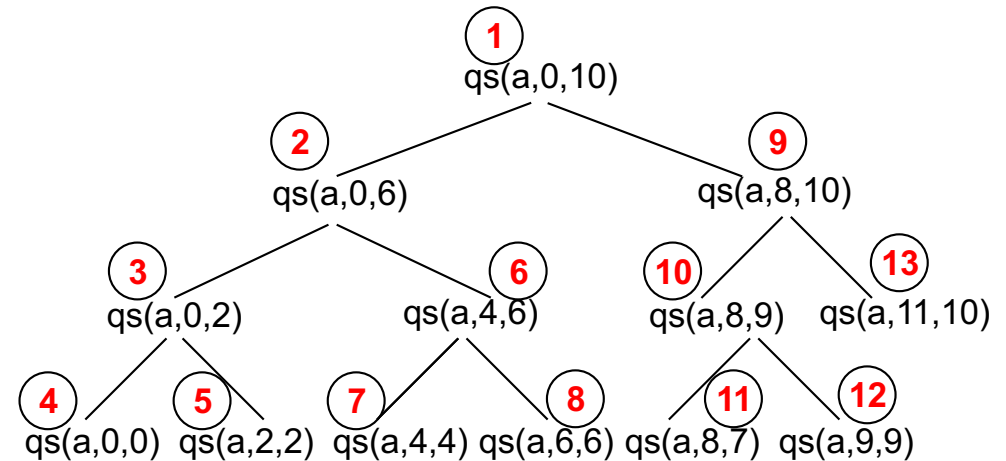
```
private static int partition(int a[], int li, int re) {  
  
    int v = a[re]; // Pivotelement  
    int i = li-1;  
    int j = re;  
  
    while (true) {  
        do i++; while (a[i] < v);  
        do j--; while (j >= li && a[j] > v);  
        if (i >= j)  
            break;  
        swap(a, i, j);  
    }  
  
    // Pivotelement v = a[re] und a[i] vertauschen:  
    a[re] = a[i]; a[i] = v;  
  
    return i;  
}
```



Aufrufbeispiel

Aufruf von QuickSort mit $a = \{3, 5, 9, 2, 10, 7, 4, 11, 1, 6, 8\}$

Zeitpunkt	Feld a [0] ... [10]			
1	3, 5, 9, 2, 10, 7, 4, 11, 1, 6, 8			
2	3, 5, 6, 2, 1, 7, 4			
3	3, 1, 2			
4	1			
5		3		
6		5, 7, 6		
7		5		
8			7	
9			10, 9, 11	
10			10, 9	
11				
12			10	
13				



qs = quicksort

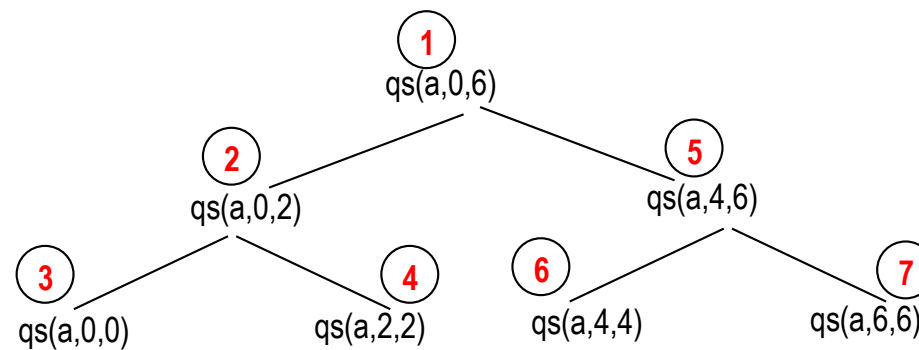
① Zeitpunkt

Zum Zeitpunkt 11 und 13 wird quicksort mit leerem Teilfeld aufgerufen

Aufrufbeispiel mit gleichen Elementen

Aufruf von QuickSort mit $a = \{5,5,5,5,5,5,5\}$

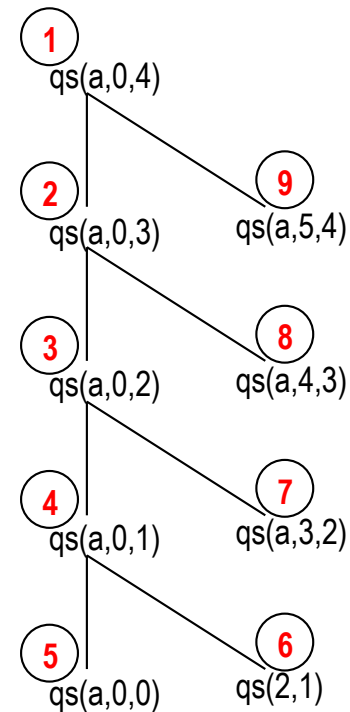
Zeitpunkt	Feld a
1	5, 5, 5, 5, 5, 5, 5 5 5 5 5 5 5 5 5
2	5, 5, 5 5 5 5 5 5
3	5
4	5
5	5, 5, 5 5 5 5 5 5
6	5
7	5



Aufrufbeispiel mit sortiertem Feld

Aufruf von QuickSort mit $a = \{1, 2, 3, 4, 5\}$

Zeitpunkt	Feld a
1	1, 2, 3, 4, 5
2	1, 2, 3, 4, 4
3	1, 2, 3, 3
4	1, 2, 2
5	1
6	
7	
8	
9	

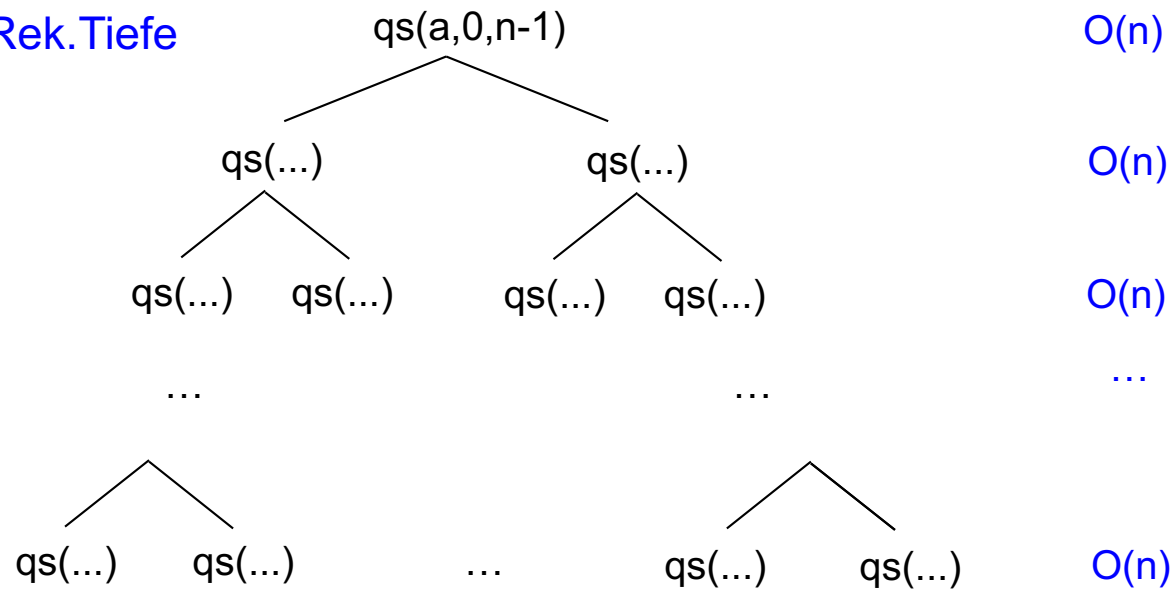


Vom Zeitpunkt 6 bis 9 wird quicksort mit einem leeren Teilfeld aufgerufen

Analyse von Quicksort – Best Case

- Feld wird immer in etwa halbiert.
- Nach Tabelle auf Seite 8-30, Fall B: $T(n) = O(n \log n)$

Maximale Rek.Tiefe
= $\log_2(n)$



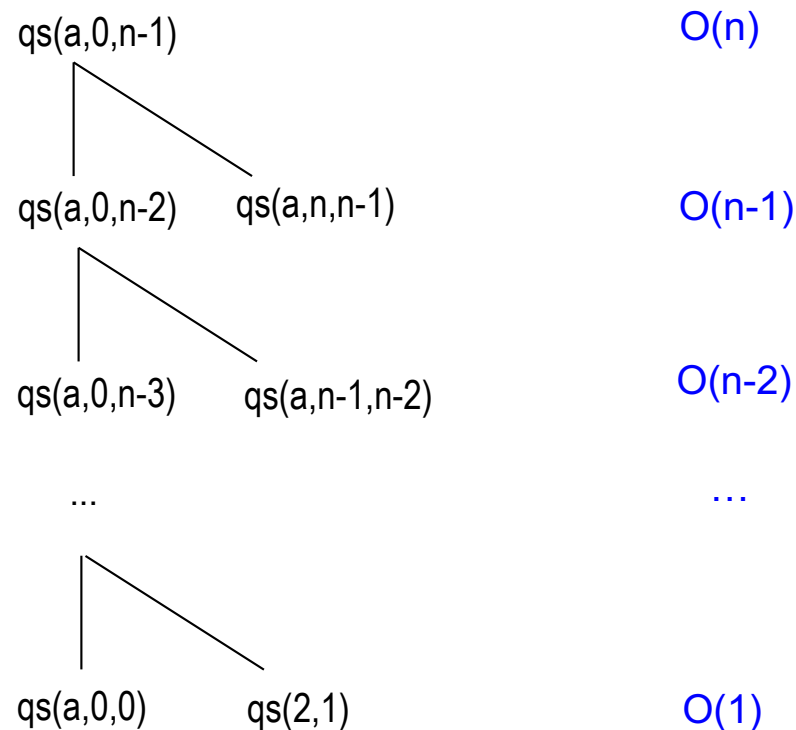
qs = quicksort

Laufzeit für partition ist $O(n)$.
Daher in jeder Rekursionstiefe
Aufwand $O(n)$.

Analyse von Quicksort – Worst Case

- Feld ist bereits sortiert!
- Max. Rekurstiefe ist daher $n-1$.
- Insgesamt: $T(n) = O(n^2)$

Maximale Rek.Tiefe
= $n-1$



Laufzeiten von Quicksort

Best-Case	Average-Case	Worst-Case
$O(n \log n)$	$O(n \log n)$	$O(n^2)$

3-Median-Strategie

Problem

Bei nahezu sortierten Feldern ist das als Pivotelement gewählte Element am rechten Rand meistens das größte (bzw. kleinste), was zu einer schlechten Aufteilung des Feldes führt.

3-Median-Strategie

- Wähle als Pivotelement den **Median** der 3 Elemente $a[li]$, $a[(li + re)/2]$ und $a[re]$.
- Definition:** Der **Median** einer Folge x von n Zahlen ist das Element, das an der mittleren Position $n/2$ liegt, falls x sortiert wäre.

Z.B. gilt für die Folge $x = 1, 9, 5, 2, 3$: $\text{Median}(x) = 3$.

Denn die sortierte Folge lautet $1, 2, 3, 4, 9$ und das Element an der mittleren Position ist 3 .

```
int partition(int a[ ], int li, int re) {  
  
    bestimme den Median aus der Folge von  
    3 Zahlen  $a[li]$ ,  $a[(li + re)/2]$ ,  $a[re]$  und  
    vertausche Median mit  $a[re]$ ;  
  
    int v =  $a[re]$ ; // Pivotelement  
    // ... der restliche Teil von partition wie bisher  
}
```

Beispiel:

Feld a						
1	2	3	4	5	6	7
			7			4
1	2	3	7	5	6	4
			4			7
1	2	3		5	6	7
		

Problem

- Bei sortierten Feldern steigt die maximale Rekursionstiefe bis $n-1$ an.
- Dies führt zu einer benötigten Größe des Systemkellers von $O(n)$. Bei großen Feldern ist daher mit einem Kellerüberlauf (stack overflow) zu rechnen, was nicht tolerierbar ist.

Lösungsschritt 1:

- Für die Sortierung ist es unerheblich, in welcher Reihenfolge die beiden rekursiven Quicksort-Aufrufe stattfinden.
- Die Reihenfolge der Aufrufe wird nun so gewählt, dass das kleinere Teilfeld zuerst sortiert wird.

```
static void quickSort(int a[ ], int li, int re) {  
    if (re > li) {  
        // Teileschritt:  
        int i = partition(a, li, re);  
  
        // Herrscheschritt:  
        if (i-li < re-i) {  
            quicksort(a, li, i-1);  
            quicksort(a, i+1, re);  
        } else {  
            quicksort(a, i+1, re);  
            quicksort(a, li, i-1);  
        }  
    }  
}
```

Lösungsschritt 2:

Sowohl der letzte rekursive Aufruf im then-Teil als auch der im else-Teil ist endrekursiv und kann mit der im Kapitel "Rekursion" besprochenen Technik beseitigt werden:

Maximale Rekursionstiefe

Da bei jedem rekursiven Aufruf das Feld wenigstens halbiert wird (warum?), ist die maximale Rekursionstiefe **höchstens** $\log_2 n$.

```
static void quickSortTailRecElim(int a[ ], int li, int re) {  
    while (re > li) {  
        // Teileschritt:  
        int i = partition(a, li, re);  
  
        // Herrscheschritt:  
        if (i-li < re-i) {  
            quickSortTailRecElim(a, li, i-1);  
            li = i+1;    // quicksortTailRecElim(a,i+1,re);  
        } else {  
            quickSortTailRecElim(a, i+1, re);  
            re = i-1;    // quicksortTailRecElim(a,li,i-1);  
        }  
    }  
}
```

Laufzeitmessungen und Vergleich zu anderen Sortierv Verfahren

Sortierv Verfahren	n = 2000	n= 4000	n = 6000	n = 8000	n = 10000
insertionSort	0.77	2.63	5.86	10.41	16.24
selectionSort	4.66	18.31	41.08	72.85	114.87
bubbleSort	5.61	23.16	52.63	95.08	149.89
quickSort	0.28	0.33	0.51	0.68	0.86
quickSort3Median	0.23	0.32	0.49	0.66	0.89
mergeSortOpt	0.23	0.48	0.78	0.99	1.27

Sortierv Verfahren	n = 20000	n= 40000	n = 60000	n = 80000	n = 100000
quickSort	2.12	3.83	5.95	8.10	10.31
quickSort3Median	1.79	3.65	5.65	7.71	9.82
mergeSortOpt	2.71	5.78	8.86	12.13	15.05

Zeiten in msec

Messbedingungen

int-Felder zufällig initialisiert. Messbedingungen wie zuvor bei den einfachen Sortierv Verfahren.

Aufgabe 9.2

- Für quickSort wurden folgende CPU-Zeiten gemessen.
(gemittelt aus 30 Messungen für zufällig generierte Felder).
Schätzen Sie die zu erwartende CPU-Zeit für $n = 10^6$ ab.

$n = 20000$	$n = 40000$	$n = 60000$	$n = 80000$	$n = 10^5$	$n = 10^6$
2.12	3.83	5.95	8.10	10.31	?

Zeiten in msec

Laufzeitmessungen für sortierte und nahezu sortierte Felder

Sortier- verfahren	n = 10000 sortiert	n = 10000 99%-sortiert	n = 10000 95%-sortiert	n = 20000 sortiert	n = 20000 99%-sortiert	n = 20000 95%-sortiert
quickSort TailRecElim	78.18	9.70	2.93	317.73	21.02	5.99
quickSort 3Median	0.28	0.35	0.32	0.51	0.51	0.59

Zeiten in msec

Vorsortierung

p%-sortiert bedeutet, dass p Prozent der Daten sortiert sind und die restlichen Daten (d.h. (100–p) % der Daten) an zufällig gewählten Stellen eingestreut worden sind.

Anmerkungen

- Standard-QuickSort führt bei sortierten Daten zu einem Stack-Overflow. Daher wurde QuickSort mit eliminierter Endrekursion (quickSortTailRecElim) eingesetzt.
- Erwartungsgemäß schneidet quickSort bei sortierten Daten schlecht ab ($O(n^2)$ -Verhalten).

Auswählen der k-kleinsten Zahl (1)

Problemstellung

Bestimme für ein Feld a mit n Zahlen die k -kleinste Zahl ($0 \leq k < n$):

$k = 0$: kleinste Zahl

$k = 1$: zweit-kleinste Zahl

$k = 2$: dritt-kleinste Zahl

...

$k = n/2$: Median

...

$k = n-1$: größte Zahl

Idee

- Naiver Ansatz:
Sortiere a mit QuickSort und liefere $a[k]$ zurück.
- Verbesserung des naiven Ansatzes:
Um $a[k]$ (im sortierten Feld) zu bestimmen, genügt es mit QuickSort nur die Hälfte des Feldes weiter rekursiv zu sortieren, in der $a[k]$ liegt.

Auswählen der k-kleinsten Zahl (2)

Algorithmus:

```
public static int quickSelect(int a[], int li, int re, int k) {  
    if (re >= li) {  
        // Teileschritt:  
        int i = partition(a, li, re);  
  
        // Herrscheschritt:  
        if (k < i)  
            return quickSelect (a, li, i-1, k);  
        else if (k > i)  
            return quickSelect (a, i+1, re, k);  
        else  
            return a[k];  
    }  
}
```

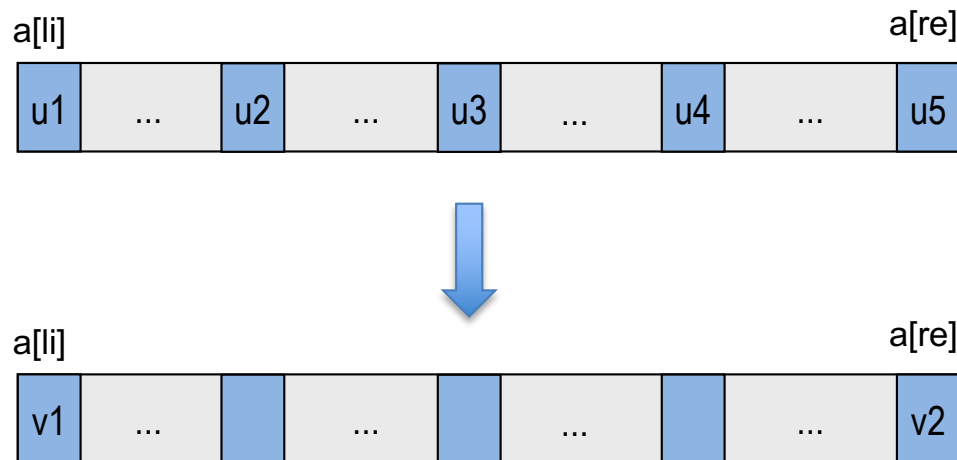
Aufgabe 9.3

- Mit welcher Größenordnung der Laufzeit (O-Notation) von quickSelect ist zu rechnen? Gehen Sie bei der Analyse davon aus, dass durch die partition-Aufrufe immer in etwa zwei gleichgroße Teilhälften entstehen.
- Welche rekursiven Aufrufe sind endrekursiv? Beseitigen Sie diese.

Dual-Pivot Quicksort: Bibliotheksfunktion aus der Java API (1)

Initialisierung:

- wähle im Bereich $a[li]$ bis $a[re]$ aus 5 Elementen u_1, u_2, u_3, u_4, u_5 zweit-kleinstes bzw. viert-kleinstes Element v_1 bzw. v_2 aus.
Bringe v_1 bzw. v_2 durch Vertauschung an den linken bzw. rechten Rand:



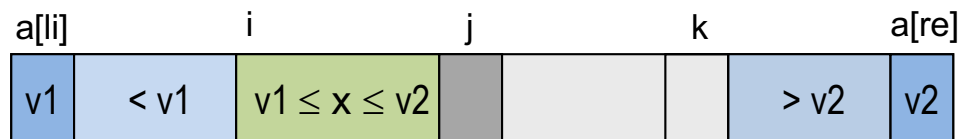
- v_1 und v_2 sind nun die Pivotelemente.
Setze Indizes $i = j = li+1$ und $k = re-1$:



Dual-Pivot Quicksort: Bibliotheksfunktion aus der Java API (2)

Umordnungsphase:

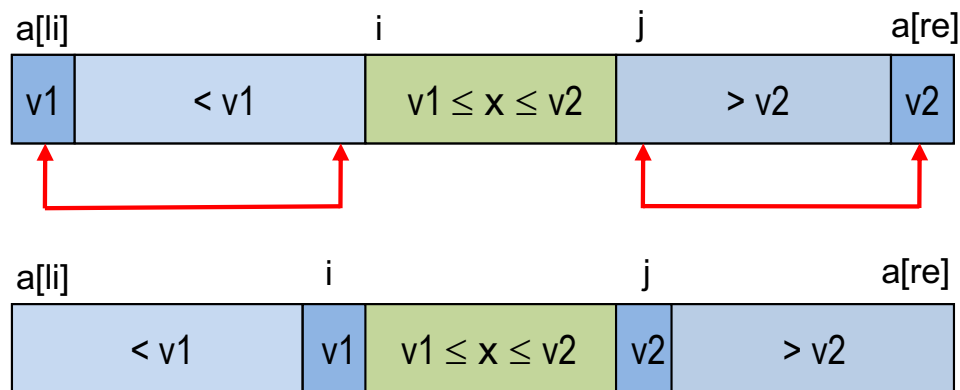
- In der Umordnungsphase werden die Elemente aus dem grauen Bereich (noch nicht betrachtete Elemente) in einen der drei Bereiche eingeordnet:
 - kleiner als $v1$
 - zwischen $v1$ und $v2$
 - größer als $v2$
- dazu wird $a[j]$ in einen der drei Bereiche eingeordnet, so lange $j \leq k$ (d.h. es gibt noch Elemente aus dem grauen Bereich):
 - $a[j] < v1$: `swap(a, i, j); i++; j++;`
 - $v1 \leq a[j] \leq v2$: `j++;`
 - $a[j] > v2$: `swap(a, j, k); k--;`



Dual-Pivot Quicksort: Bibliotheksfunktion aus der Java API (3)

Nach Umordnungsphase (grauer Bereich ist abgebaut):

- `swap(a, li, i-1); i--;`
- `swap(a, j, re);`



Sortiere 3 Teilfelder durch Rekursion weiter:

- $a[li], \dots, a[i-1]$
- $a[i+1], \dots, a[j-1]$ (muss nur sortiert werden, falls $v1 \neq v2$)
- $a[j+1], \dots, a[re]$

Beachte: Pivotelemente $a[i]$ und $a[j]$ stehen bereits an der richtigen Stelle und müssen nicht weiter betrachtet werden.

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- **Mergesort**
- Stabile Sortiervverfahren
- Fächersortieren
- Externes 2-Wege-Mergesort
- Generische Sortiermethoden
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

Wesentliche Eigenschaften

Idee

- MergeSort ist ein weiteres Teile-und-Herrsche-Verfahren.
- Im Gegensatz zu quickSort besteht bei MergeSort der Teileschritt nur aus einer Halbierung der Feldes.
- Der eigentliche Aufwand steckt bei MergeSort im Herrscheschritt. Die beiden Teilhälften werden durch rekursive Aufrufe sortiert. Die sortierten Teilhälften werden zu einer sortierten Gesamtfolge **verschmolzen** (engl. **merge**).

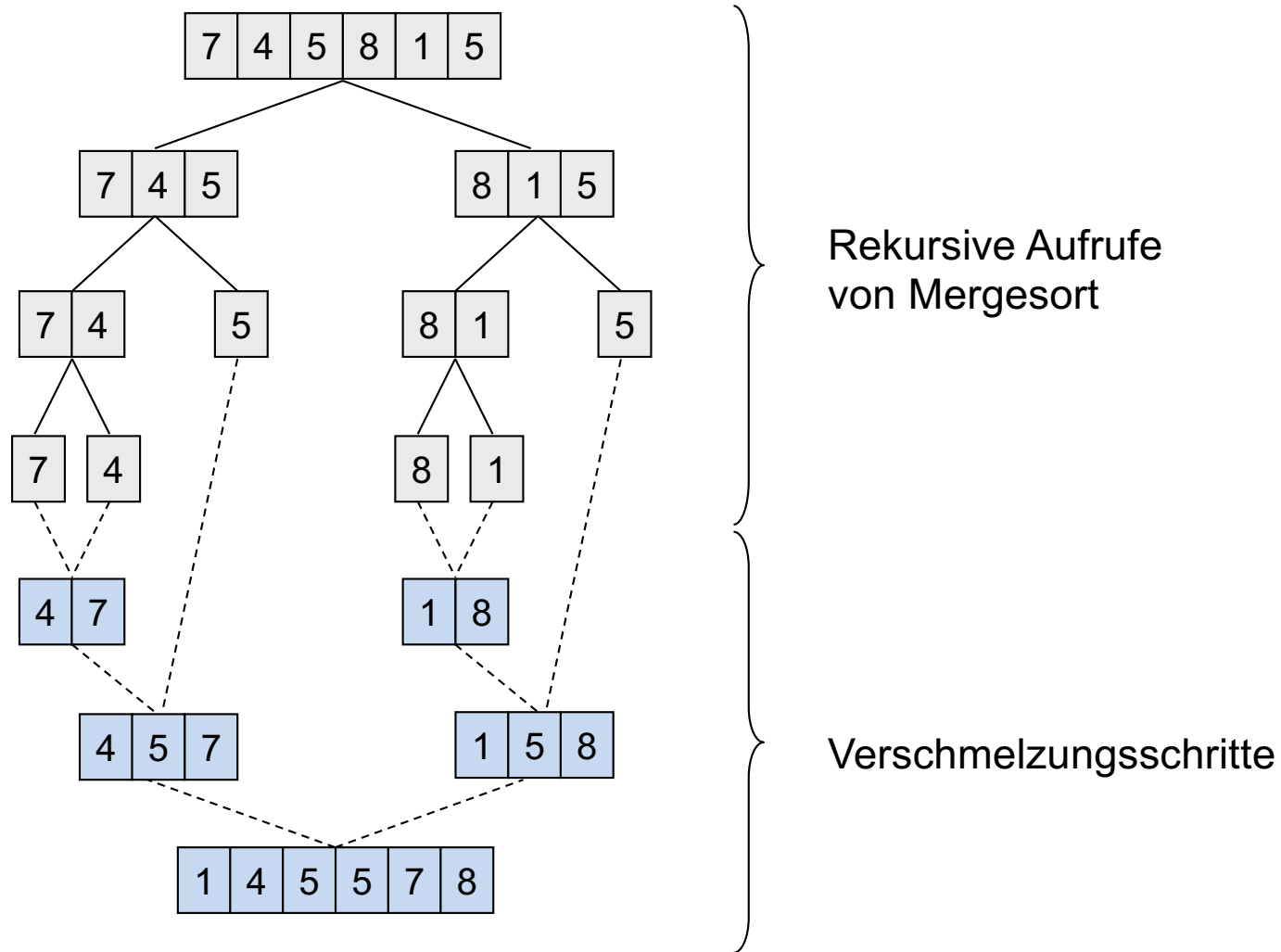
Vorteil

- Da die Felder immer genau halbiert werden, ist eine Laufzeit von $O(n \log_2 n)$ garantiert.

Nachteil

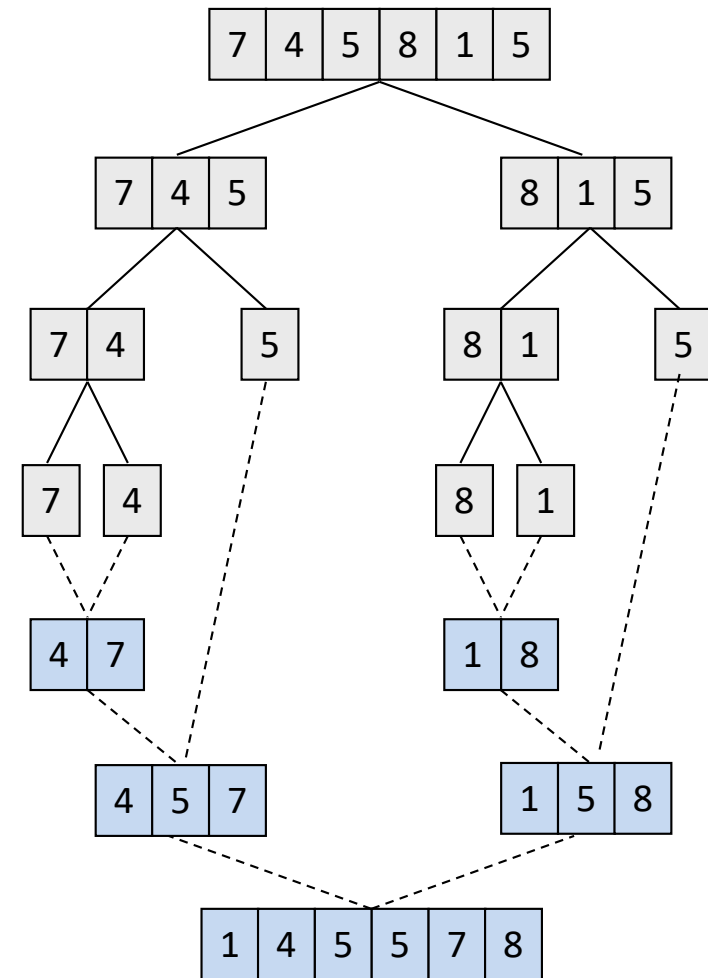
- Für die Verschmelzung ist ein zusätzliches Feld erforderlich.

Illustrierung an einem Beispiel



Grobstruktur des Algorithmus

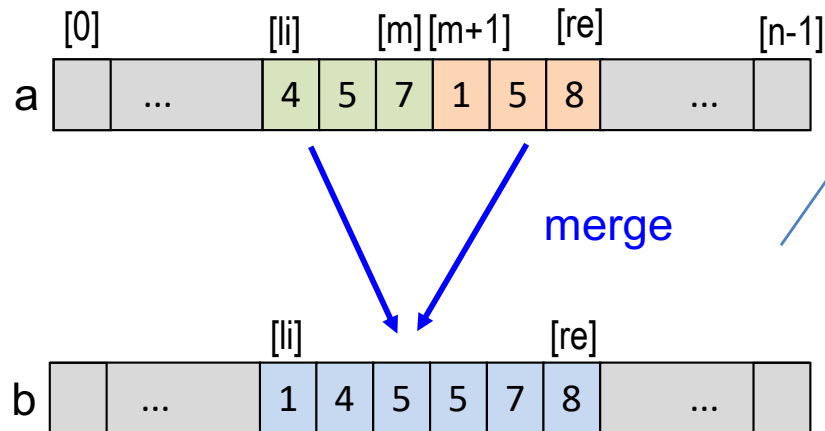
```
public static void mergeSort(int a[ ]) {  
    mergeSort(a, 0, a.length-1);  
}  
  
public static void mergeSort(int[ ] a, int li, int re) {  
    // Sortiert Teilfeld a[li], a[li+1], ..., a[re].  
    if (re > li) {  
        // Teileschritt:  
        int m = (li + re)/2;  
  
        // Herrscheschritt:  
        mergeSort(a, li, m);  
        mergeSort(a, m+1, re);  
  
        // Verschmelzung:  
        verschmelze die sortierten Teilfelder  
        a[li],...,a[m] und a[m+1],...,a[re] und  
        schreibe Ergebnis nach a[li],...,a[re];  
    }  
}
```



Einfache MergeSort-Variante

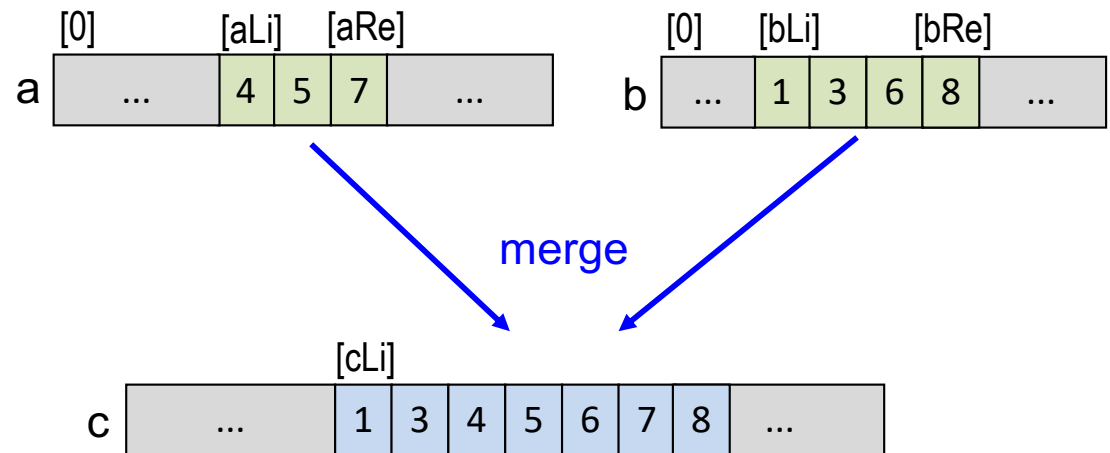
```
public static void mergeSort(int[] a) {  
  
    // Hilfsfeld b anlegen:  
    int[] b = new int[a.length];  
  
    // Aufruf der rekursiven  
    // MergeSort-Funktion:  
    mergeSort(a, 0, a.length-1, b);  
}
```

```
private static void mergeSort(int[] a, int li, int re, int[] b) {  
    if (re > li) {  
        // Teileschritt:  
        int m = (li + re)/2;  
  
        // Herrscheschritt:  
        mergeSort(a, li, m, b);  
        mergeSort(a, m+1, re, b);  
  
        // verschmelze die sortierten Teilfelder  
        // a[li],...,a[m] und a[m+1],...,a[re] und  
        // schreibe Ergebnis nach b[li],...,b[re];  
        merge(a, li, m, a, m+1, re, b, li);  
  
        // Kopiere b[li],...,b[re] zurueck nach a[li],...,a[re]:  
        System.arraycopy(b, li, a, li, re-li+1);  
    }  
}
```



Funktion merge

```
private static void merge(  
    int[ ] a, int aLi, int aRe,  
    int[ ] b, int bLi, int bRe,  
    int[ ] c, int cLi )  
{  
    int i = aLi;  
    int j = bLi;  
    int k = cLi;  
  
    while (i <= aRe && j <= bRe)  
        if (a[i] <= b[j])  
            c[k++] = a[i++];  
        else  
            c[k++] = b[j++];  
    if (j == bRe+1) // b zu Ende  
        while (i <= aRe)  
            c[k++] = a[i++];  
    else // a zu Ende  
        while (j <= bRe)  
            c[k++] = b[j++];  
}
```



Verschmelze sortierte Teilfelder
 $a[aLi], \dots, a[aRe]$ und $b[bLi], \dots, b[bRe]$
und schreibe Ergebnis nach $c[cLi], \dots$

Optimierte MergeSort-Variante

- Das Zurückkopieren von Hilfsfeld b nach a wird vermieden, Dazu werden nach jedem rekursiven Aufruf die Rollen von a und b vertauscht. (Raffinierte Lösung!)

```
public static void mergeSortOpt(int[ ] a) {  
  
    // Hilfsfeld b anlegen und  
    // mit a initialisieren  
    int[ ] b = new int[a.length];  
    System.arraycopy(a, 0, b, 0, a.length) ;  
  
    // Aufruf der rekursiven  
    // MergeSort-Funktion:  
    mergeSortOpt(a, 0, a.length-1, b);  
}
```

Kopieraufruf ist wichtig, um die Vorbedingung der privaten mergeSortOpt-Methode zu erfüllen.

```
private static void mergeSortOpt(int[ ] a, int li, int re, int[ ] b) {  
    // Sortiert a[li], ..., a[re].  
    // Vorbedingung: a[li], ..., a[re] und b[li], ..., b[re] sind  
    // die gleichen Zahlenfolgen.  
    // Nachbedingung: a[li], ..., a[re] sind sortiert.  
  
    if (re > li) {  
        // Teileschritt:  
        int m = (li + re)/2;  
  
        // Herrscheschritt: Sortiere b[li],...,b[m] und b[m+1], ..., b[re]  
        mergeSortOpt(b, li, m, a);  
        mergeSortOpt(b, m+1, re, a);  
  
        // verschmelze die sortierten Teilfelder b[li],...,b[m] und  
        // b[m+1],...,b[re] und schreibe Ergebnis nach a[li],...,a[re]:  
        merge(b, li, m, b, m+1, re, a, li);  
    }  
}
```

Analyse von Mergesort

CPU-Zeiten:

Sortiervverfahren	n = 20000	n= 40000	n = 60000	n = 80000	n = 100000
quickSort3Median	1.79	3.65	5.65	7.71	9.82
mergeSort (einfache Variante)	3.30	6.69	10.24	13.93	17.38
mergeSortOpt	2.71	5.78	8.86	12.13	15.05

gemittelte Zeiten in msec

Größenordnungen der Laufzeiten:

Best-Case	Average-Case	Worst-Case
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

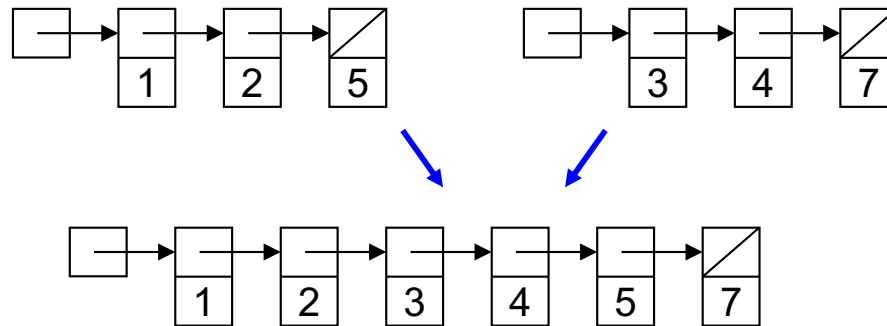
Aufgabe 9.4

Bestätigen Sie die Laufzeit von Mergesort im Best- und Worst-Case.

Verwenden Sie dabei die im Kapitel Komplexitätsanalyse hergeleiteten Ergebnisse zu Teile-und-Herrsche-Verfahren.

Sortieren von linear verketteten Listen

Verschmelzen (merge) von sortierten linear, verketteten Listen:



Beachte:

Verschmelzung von sortierten, linear verketteten Listen lässt sich im Gegensatz zu Feldern ohne Hilfsdatenstrukturen durchführen.

Aufgabe 9.5

Schreiben Sie ein MergeSort-Verfahren zum Sortieren von linear, verketteten Listen mit n Knoten:

Node mergeSort(Node p, int n);

Beachte: der Parameter n ist nicht unbedingt notwendig, lässt jedoch eine etwas effizientere Implementierung zu.

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- Mergesort
- **Stabile Sortiervverfahren**
- Fächersortieren
- Externes 2-Wege-Mergesort
- Generische Sortiermethoden
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

Stabile Sortiervverfahren (1)

- Ein Sortiervverfahren heißt **stabil**, falls sich die Reihenfolge von Elementen mit gleichem Schlüssel nicht ändert.
- Beispielsweise wird die Folge von Personendaten

Anton, 35 Jahre
Hans, 32 Jahre
Markus, 35 Jahre
Petra, 28 Jahre
Sylvia, 35 Jahre

durch Sortierung nach dem Schlüssel Alter mit einem stabilen Sortiervverfahren in die folgende Reihenfolge gebracht:

Petra, 28 Jahre
Hans, 32 Jahre
Anton, 35 Jahre
Markus, 35 Jahre
Sylvia, 35 Jahre

Vorteil

- Die Stabilität ist wichtig, wenn die Datensätze nach einem anderen Schlüssel bereits vorsortiert sind.
- Zum Beispiel sind die Daten oben bereits nach dem Namen alphabetisch vorsortiert. Nach der stabilen Sortierung nach dem Alter bleiben daher die Daten in einer Altersstufe alphabetisch sortiert.

Stabile Sortiervverfahren (2)

Aufgabe 9.6

Welche der vorgestellten Sortiervverfahren sind stabil?

Sortiervverfahren	stabil	Begründung
InsertionSort		
SelectionSort		
BubbleSort		
QuickSort		
MergeSort		

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- Mergesort
- Stabile Sortiervverfahren
- **Fächersortieren**
- Externes 2-Wege-Mergesort
- Generische Sortiermethoden
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

BucketSort

Idee

- Definiere für jeden Schlüsselwert ein **Fach (Bucket)** und füge jeden zu sortierenden Datensatz in das entsprechende Fach ein.
- Die Anzahl M der unterschiedlichen Schlüssel sollte dabei verhältnismäßig klein sein (Beispiele: Noten: 1, 2, 3, 4, 5; Geburtsdaten: 1.1, ..., 31.12).

Algorithmus

```
private static void bucketSort(int[] a) {  
  
    M Fächer als Feld bucket[M] anlegen;  
  
    for (int i = 0; i < a.length; i++)  
        füge a[i] in bucket[a[i]] ein;  
  
    kopiere bucket nach a zurück;  
}
```

Laufzeit

$T(n) = O(n)$.

Implementierungsvarianten

- Jedes Fach `bucket[v]` ist eine linear verkettete Liste
- Alle Fächer werden in einem einzigen Feld der Größe n abgespeichert. Dazu ist ein extra Zählerdurchgang durch alle Daten notwendig, um die Fächergrößen und damit die Indexgrenzen zu ermitteln.

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- Mergesort
- Stabile Sortiervverfahren
- Fächersortieren
- **Externes 2-Wege-Mergesort**
- Generische Sortiermethoden
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

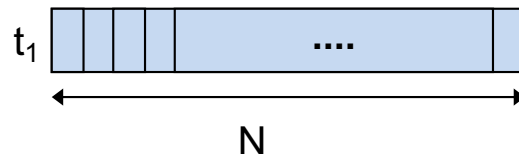
Externes 2-Wege-Mergesort

Problemstellung

- Datensätze, die sortiert werden sollen, befinden sich in einer Datei.
- Datei passt nicht komplett in Hauptspeicher.
- Auf die Daten kann nur sequentiell zugegriffen werden.
- L = Anzahl der Daten, die in den Hauptspeicher passen.
- N = Anzahl Daten, die insgesamt sortiert werden sollen.

Verfahren

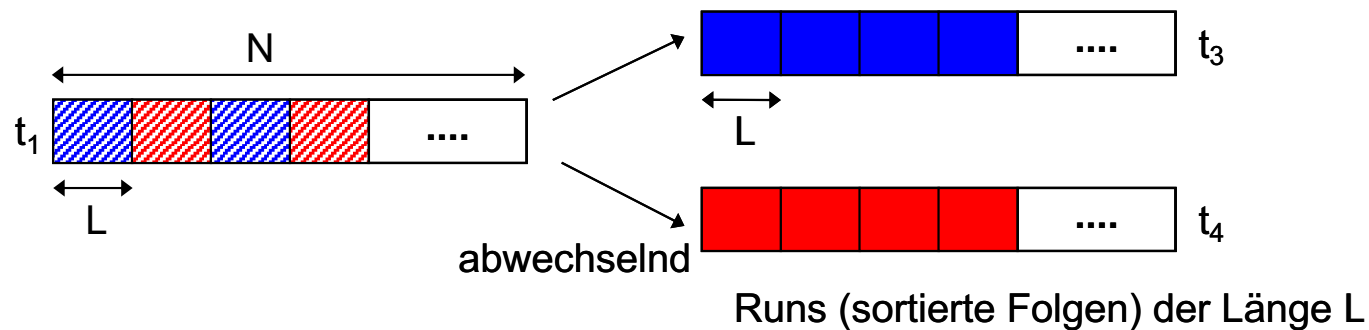
- Die zu sortierende N Datensätze befinden sich anfangs in einer Datei t_1 .



- Es stehen 3 weitere Hilfsdateien t_2 , t_3 , und t_4 zur Verfügung.
- Das Verfahren besteht aus 2 Phasen:
 - Sortierphase
 - Verschmelzungsphase

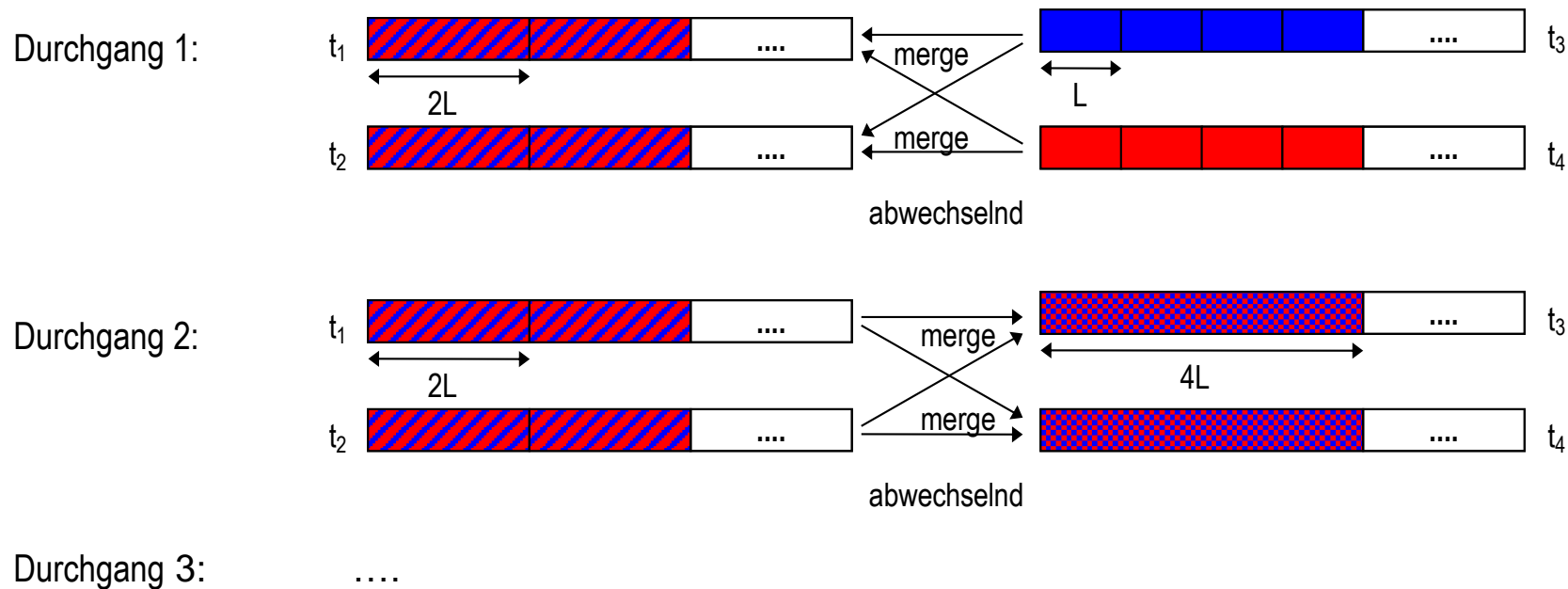
Sortierphase

- Es werden jeweils L Datensätze von t_1 gelesen, intern sortiert und abwechselnd auf t_3 und t_4 geschrieben.
- Diese sortierten Folgen werden **Runs** genannt.
- Nach dem Sortierdurchgang haben die Runs die Länge L . Sie sind in der folgenden Abbildung mit gefüllter Farbe dargestellt.



Verschmelzungsphase

- Die Verschmelzungsphase besteht aus mehreren Durchgängen.
- In jedem Durchgang gibt es 2 Eingabe- und 2 Ausgabedateien. Im ersten Durchgang sind t_3 und t_4 die Eingabe- und t_1 und t_2 die Ausgabedateien. Nach jedem weiteren Durchgang werden die Rollen vertauscht.
- In jedem Durchgang werden von den beiden Eingabedateien jeweils ein Run gelesen und zu einem doppelt so langen Run verschmolzen. Die verschmolzenen Runs werden abwechselnd auf eine der beiden Ausgabedateien geschrieben.
- Die Verschmelzungsphase ist beendet, sobald sich genau ein Run der Länge N ergibt.



Analyse

Ziel:

- Bei einem Durchgang (engl. pass) werden alle N Datensätze genau einmal gelesen und geschrieben.
- Ermittle die Anzahl der Durchgänge.

Analyse:

- Die Sortierphase besteht aus genau einem Durchgang. Es entstehen dabei N/L viele Runs.
- Nach jedem Durchgang in der Verschmelzungsphase halbiert sich die Anzahl der Runs.
- Um genau 1 Run zu erhalten, sind $P(N)$ viele Durchgänge notwendig:

$$P(N) = \left\lceil \log_2 \frac{N}{L} \right\rceil + 1$$

Rechenbeispiel:

- Für $N = 10^9$ und $L = 10^6$ ergibt sich $P(N) = 11$.

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- Mergesort
- Stabile Sortiervverfahren
- Fächersortieren
- Externes 2-Wege-Mergesort
- **Generische Sortiermethoden**
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

Generische Sortiermethode mit Comparable-Beschränkung (1)

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

```
public static <T extends Comparable<T>> void insertionSort(T[] a) {  
  
    for (int i = 1; i < a.length; i++) {  
        T v = a[i];  
        int j = i - 1;  
        while (j >= 0 && a[j].compareTo(v) > 0) {  
            a[j+1] = a[j];  
            j--;  
        }  
        a[j+1] = v;  
    }  
}
```

Hier insertionSort.
Andere Sortiervverfahren analog.

```
public static void main(String[] args) {  
  
    Integer[] intArr = {5,3,1,7,6,4,8};  
    insertionSort(intArr);  
  
    String[] strArr = {"ein", "zwei", "drei"};  
    insertionSort(strArr);  
}
```

Generische Sortiermethode mit Comparable-Beschränkung (2)

- Damit beispielsweise ein Feld mit Circle-Objekten mit der generischen sort-Methode sortiert werden kann, muss die Klasse Circle das Interface Comparable implementieren.

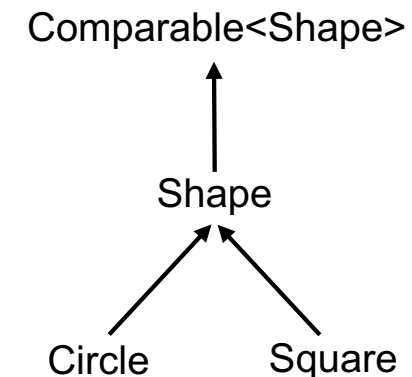
```
class Circle implements Comparable<Circle> {  
  
    private double radius;  
  
    public Circle(double r) {radius = r;}  
  
    public int compareTo(Circle c) {  
        if (radius < c.radius)  
            return -1;  
        else if (radius == c.radius)  
            return 0;  
        return +1;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Circle[] cArr = {new Circle(1), new Circle(5), new Circle(3)};  
    insertionSort(cArr);  
}
```

Nun: Sortieren von Elementen aus einer Typhierarchie

- Gegeben: Hierarchie von Typen, die Comparable als gemeinsamen Supertyp haben
- Ziel: Sortiermethode für Elemente aus dieser Typhierarchie

```
public abstract class Shape
    implements Comparable<Shape> {
    abstract double area();
    public int compareTo(Shape s){
        if (area() < s.area()) return -1;
        else if (area() == s.area()) return 0;
        else return +1;
    }
}
```



```
public class Circle extends Shape {
    private double radius;

    public Circle(double r){
        radius=r;
    }

    public double area() {
        return PI*radius*radius;
    }
}
```

```
public class Square extends Shape {
    private double width;

    public Square(double a){
        width=a;
    }

    public double area() {
        return width*width;
    }
}
```

Sortieren von Feldern mit Elementen aus einer Typhierarchie

- Felder sind **kovariant** und daher flexibel genug.
- Typbeschränkung kann so bleiben wie bisher.

```
public static <T extends Comparable<T>> void insertionSort(T[] a) {  
    // ...  
}
```

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

```
    Shape[] shapeArr = {new Circle(1), new Square(2), new Circle(3)};  
    Circle[] circleArr = {new Circle(1), new Circle(5), new Circle(3)};
```

```
    insertionSort(shapeArr);
```

```
    insertionSort(circleArr);
```

```
}
```

OK.

Da Shape ein Subtyp von Comparable<Shape> ist, darf T = Shape in der generischen Sortiermethode eingesetzt werden.

Auch OK !!!

Da Shape ein Subtyp von Comparable<Shape> und Circle[] ein Subtyp von Shape[] ist, ist der Aufruf der generischen Sortiermethode mit T = Shape korrekt.

Sortieren von Listen mit Elementen aus einer Typhierarchie

- Parameterisierte Listen sind nur mit **wildcards** flexibel genug.

```
public static <T extends Comparable<? super T> > void sort(List<T> list) {  
    // ...  
}  
  
public static void main(String[ ] args) {  
  
    List<Shape> shapeList = List.of(new Circle(1), new Square(2), new Circle(3));  
    List<Circle> circleList = List.of( new Circle(1), new Circle(5), new Circle(3));  
  
    sort(shapeList);  
  
    sort(circleList);  
}
```

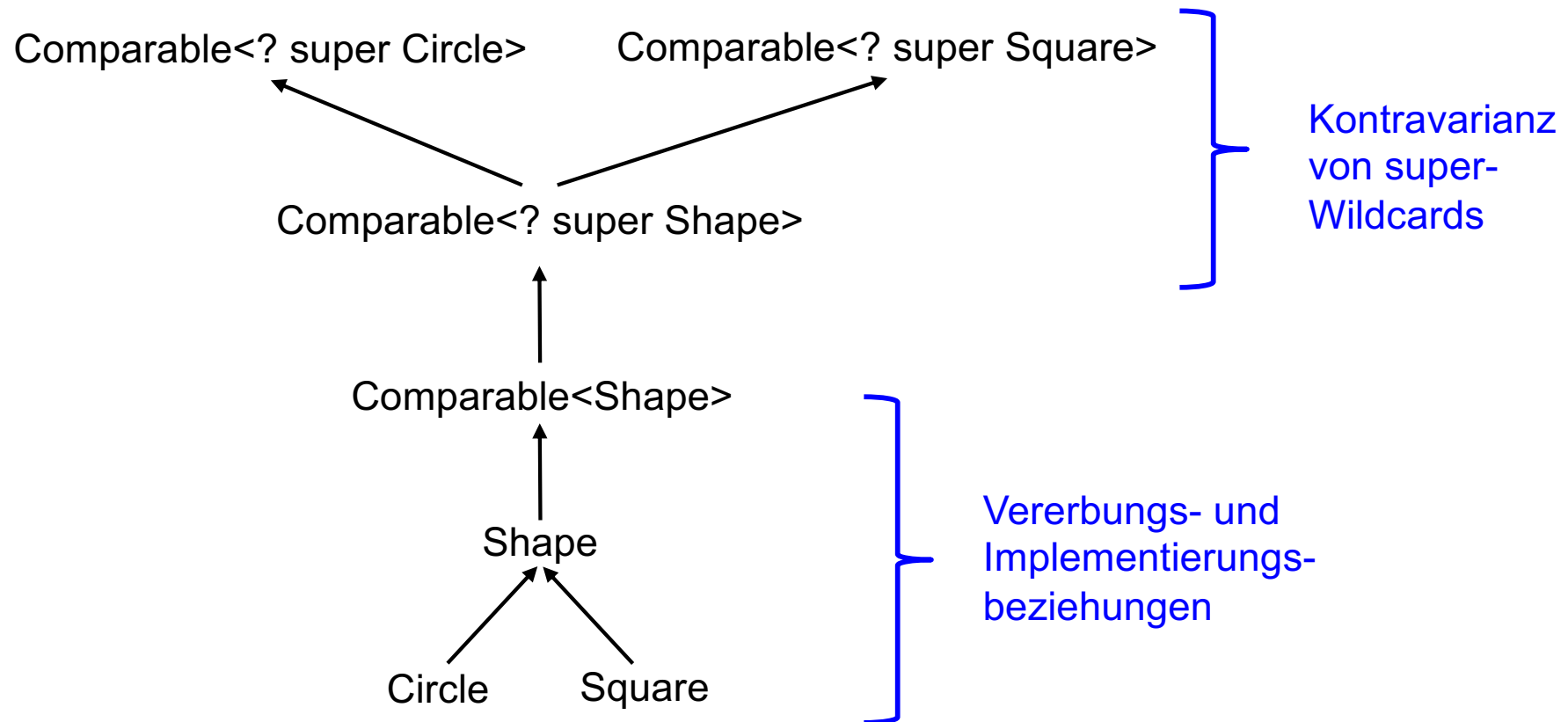
OK.

Da Shape ein Subtyp von Comparable<? super Shape> ist.

Auch OK !!!

Da Circle ein Subtyp von Comparable<? super Circle> ist.
(siehe nächste Seite)

Subtyp-Beziehungen von Circle, Square, Shape zu Comparable mit super-Wildcards



Problem: feste Compare-Funktionalität

```
public static <T extends Comparable<T>> void insertionSort(T[] a) {  
    ...  
    while (j >= 0 && a[j].compareTo(v) > 0) { ... }  
    ...  
}
```

```
class Circle implements Comparable<Circle> {  
    public Circle(double r) {radius = r;}  
    private double radius;  
    public int compareTo(Circle c) {...}  
}
```

```
public static void main(String[] args) {  
    Circle[] cArr = {new Circle(1), new Circle(5), new Circle(3)};  
    insertionSort(cArr);  
}
```

Problem:

- Indem die Klasse Circle das Interface Comparable implementiert, ist die Vergleichsoperation für Circle für immer festgelegt.
- Soll nun ein Circle-Feld unterschiedlich sortiert werden (absteigend, aufsteigend, nach Flächeninhalt, nach Lage, etc.), dann ist dieser Ansatz zu unflexibel.
- Daher: Vergleichsoperation als Parameter.

Generische Sortiermethode mit Comparator-Parameter (1)

- Mit Hilfe des Interface `java.lang.Comparator` können an Methoden - wie z.B. Sortiermethoden - Vergleichsoperationen als Parameter übergeben werden.

```
public interface Comparator<T> {  
    int compare(T x, T y);  
}
```

```
public static <T> insertionSort(T[] a, Comparator<T> c) {  
    for (int i = 1; i < a.length; i++) {  
        T v = a[i];  
        int j = i - 1;  
        while (j >= 0 && c.compare(a[j], v) > 0) {  
            a[j+1] = a[j];  
            j--;  
        }  
        a[j+1] = v;  
    }  
}
```

a[j] ist größer als v

- Die Spezifikation für `Comparator` in der Java API ist einzuhalten.
- Insbesondere soll `compare(x,y)` eine negative Zahl, 0, bzw. eine positive Zahl zurückliefern, falls x kleiner, gleich bzw. größer als y ist.
- Es wird empfohlen, dass `compare(x,y)==0` und `x.equals(y)` denselben Wert zurückliefern.

Generische Sortiermethode mit Comparator-Parameter (2)

- Im Elementtyp lassen sich verschiedene Vergleichsoperationen zur Verfügung stellen.

```
class Circle {  
    private double radius;  
    public Circle(double r) {radius = r;}  
  
    private static class NaturalOrderComparator implements Comparator<Circle> {  
        public int compare(Circle c1, Circle c2) {  
            if (c1.radius < c2.radius)  
                return -1;  
            else if (c1.radius == c2.radius)  
                return 0;  
            else  
                return +1;  
        }  
    }  
  
    public static Comparator<Circle> naturalOrder() {  
        return new NaturalOrderComparator();  
    }  
  
    // ... naechste Folie
```

Generische Sortiermethode mit Comparator-Parameter (3)

```
// ...

private static class ReverseOrderComparator implements Comparator<Circle> {
    public int compare(Circle c1, Circle c2) {
        NaturalOrderComparator cmp = new NaturalOrderComparator();
        return cmp.compare(c2, c1);
    }
}

public static Comparator<Circle> reverseOrder() {
    return new ReverseOrderComparator();
}

}
```

reverseOrder ist invers zu NaturalOrder

```
public static void main(String[] args) {
    Circle[] cArr = {new Circle(1), new Circle(5), new Circle(3)};

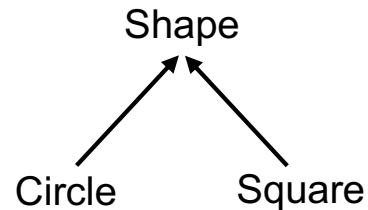
    insertionSort(cArr, Circle.naturalOrder());
    insertionSort(cArr, Circle.reverseOrder());
}
```

aufsteigend sortieren

absteigend sortieren

Generische Sortiermethode mit Comparator-Parameter (4)

- Sollen Elemente aus einer Typhierarchie sortiert werden, kann die Flexibilität ebenfalls mit Wildcards erhöht werden.



```
public static <T> insertionSort(T[] a, Comparator<? super T> c {  
    // ...  
}
```

```
public class Shape {  
    ...  
    public static Comparator<Shape> naturalOrder() {  
        return new Comparator<Shape>() {....};  
    }  
}
```

```
public static void main(String[] args) {  
    Circle[] cArr = {new Circle(1), new Circle(5), new Circle(3)};  
  
    insertionSort(cArr, Shape.naturalOrder());  
}
```

Vorgriff: Lambda-Ausdruck als Comparator

```
Integer[ ] aArr = {5, 2, 7, 8, 9, 1, 4, 3, 6, 10};  
insertionSort(cArr(a, new Comparator<Integer>() {  
    public int compare(Integer x, Integer y) {  
        return y.compareTo(x);  
    }  
}));
```

Herkömmliches
Comparator-
Objekt

```
Integer[ ] aArr = {5, 2, 7, 8, 9, 1, 4, 3, 6, 10};  
insertionSort(cArr(a, (x,y) -> y.compareTo(x) ));
```

Lambda-Ausdruck als
Comparator

Kapitel 9: Sortiervverfahren

- Problemstellung
- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Vertauschen (Bubble-Sort)
- Quicksort
- Mergesort
- Stabile Sortiervverfahren
- Fächersortieren
- Externes 2-Wege-Mergesort
- Generische Sortiermethoden
- Sortieren mit der Java-API:
Arrays, Collections, List und Streams

Suchen und Sortieren mit java.util.Arrays (1)

- Die Klasse `Arrays` besteht ausschließlich aus statischen Methoden zum Bearbeiten von Feldern.
- Es gibt u.a. Methoden zum Suchen und Sortieren. Dabei kann auch nur ein Teil des Feldes [fromIndex, toIndex) durchsucht bzw. sortiert werden.
- Es gibt Methoden für Felder mit Basisdatentypen, für Object-Felder und für generische Felder.
- binarySearch für Basisdatentypen führt eine binäre Suche durch und setzt daher voraus, dass das Feld bereits sortiert ist.

```
public static int binarySearch(int[ ] a, int key);  
public static int binarySearch(double[ ] a, double key);  
public static int binarySearch(int[ ] a, int fromIndex, int toIndex, int key);  
public static int binarySearch(double[ ] a, int fromIndex, int toIndex, double key);  
...
```


Suchen und Sortieren mit java.util.Arrays (2)

- sort für Basisdatentypen ist ein modifiziertes QuickSort mit 2 Pivotelementen (Dual-Pivot Quicksort) und einer Partitionierung in 3 Teilen.
- Das Sortierverfahren ist nicht stabil.

```
public static void sort(int[ ] a);  
public static void sort(int[ ] a, int fromIndex, int toIndex);  
  
public static void sort(double[ ] a);  
public static void sort(double[ ] a, int fromIndex, int toIndex);  
...
```

- sort für ein Object-Feld ist eine modifizierte MergeSort-Variante.
- Die Elemente müssen vom Typ Comparable sein.
- Das Sortierverfahren ist stabil.

```
public static void sort(Object[ ] a);  
public static void sort(Object[ ] a, int fromIndex, int toIndex);
```

Suchen und Sortieren mit java.util.Arrays (3)

- binarySearch und sort gibt es auch als generische Methoden.
- sort ist eine modifizierte MergeSort-Variante und ist daher stabil (wie das sort für Object-Felder).
- Die Vergleichsoperation wird dabei als Comparator-Objekt verpackt und als Parameter übergeben.

```
static <T> int binarySearch(T[] a, T key, Comparator<? super T> c);  
static <T> int binarySearch(T[] a, int fromIndex, int toIndex, T key, Comparator<? super T> c);
```

```
static <T> void sort(T[] a, Comparator<? super T> c);  
static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c);
```

- Die Vergleichsoperation *c* muss wenigstens Elemente vom Typ *T* vergleichen können. Daher: `Comparator<? super T>`.
- Es darf *c* == null sein. Dann wird die Standard-Ordnung verwendet. Dazu muss aber *T* Subtyp von `Comparable<? super T>` sein.

Klasse java.util.Collections

- Die Klasse Collections besteht ausschließlich aus statischen, generischen Methoden, die auf Collections arbeiten bzw. Collections zurückliefern.
- Methoden zum Ändern der Reihenfolge von Elementen:
reverse, rotate, shuffle, sort, swap
- Methoden zum Ändern der Inhalte eines Containers:
copy, fill, replaceAll
- Methoden zum Suchen von Elementen:
min, max, binarySearch, ...
- Methoden zum Erzeugen und Verpacken von Collections:
emptyList, singletonList, unmodifiableList, ...

Sortieren und Suchen

- Die **Sortiervverfahren** sind eine mergeSort-Variante und sind damit stabil.

```
public static <T extends Comparable<? super T>> void sort(List<T> list);  
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

- Die **binarySearch-Methoden** führen eine **binäre Suche** durch und setzen daher voraus, dass die Liste bereits sortiert ist.
- Nur bei einer ArrayList-Liste wird eine $O(\log n)$ -Laufzeit garantiert.
- Bei einer LinkedList-Liste kommt das Verfahren zwar mit $O(\log n)$ Vergleichen aus, benötigt aber aufgrund der Traversierung durch die linear verkettete Liste $O(n)$ -Laufzeit.

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);  
public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c);
```

Beispiel: Korrekte bzw. nicht-korrekte Parametertypen für `binarySearch(list, key)`

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

	list	List<Circle>	List<Square>	List<Shape>	List<Integer>
key					
Circle	+	+	+	+	-
Shape	+	+	+	+	-
Square	+	+	+	+	-
Integer	-	-	-	-	+

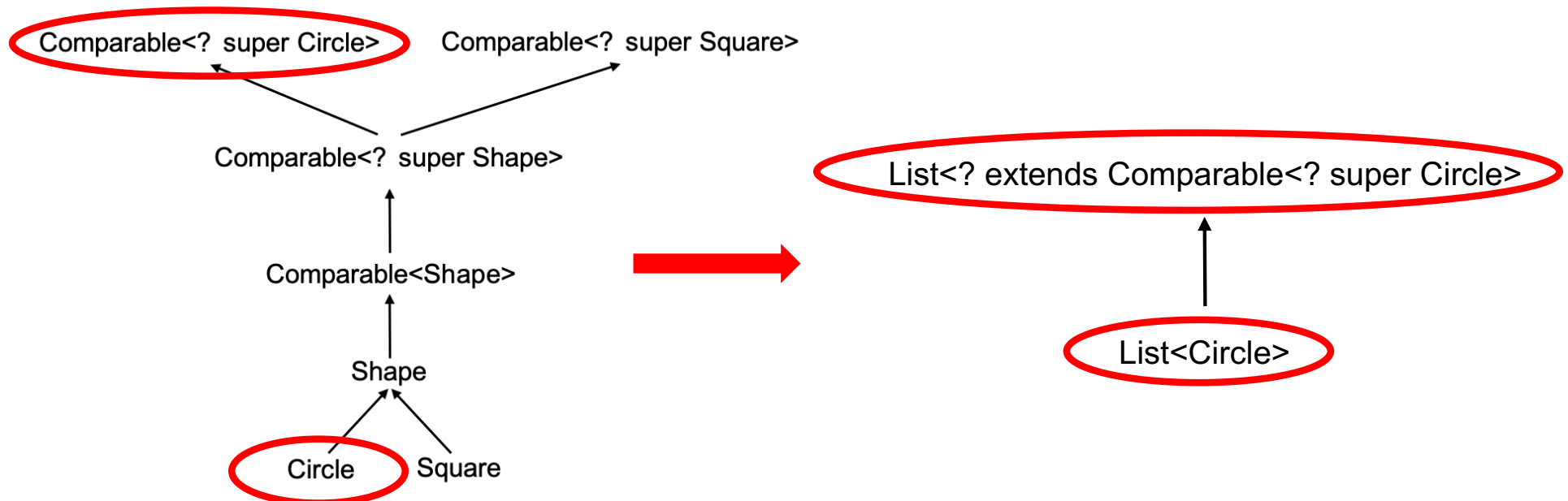
Für welche Kombination von list und key ist folgender Aufruf korrekt?

```
binarySearch(list, key);
```

Beispiel: Korrekte bzw. nicht-korrekte Parametertypen für `binarySearch(list,key)`

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

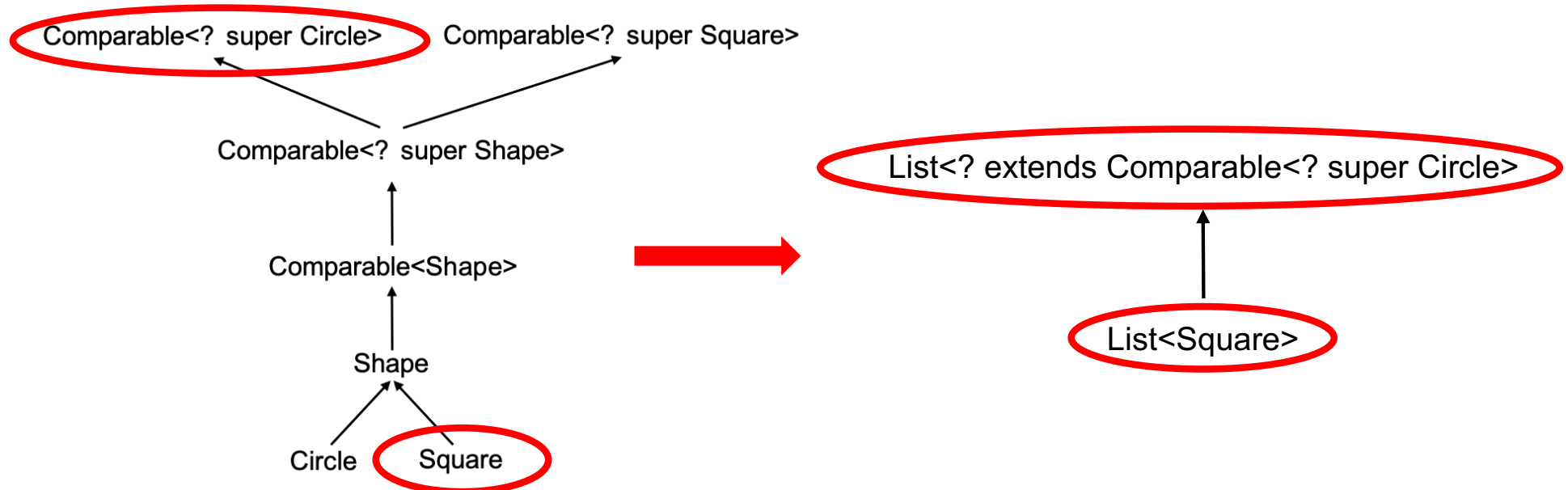
list	List<Circle>	List<Square>	List<Shape>	List<Integer>
key				
Circle	+	+	+	-
Shape	+	+	+	-
Square	+	+	+	-
Integer	-	-	-	+



Beispiel: Korrekte bzw. nicht-korrekte Parametertypen für `binarySearch(list,key)`

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

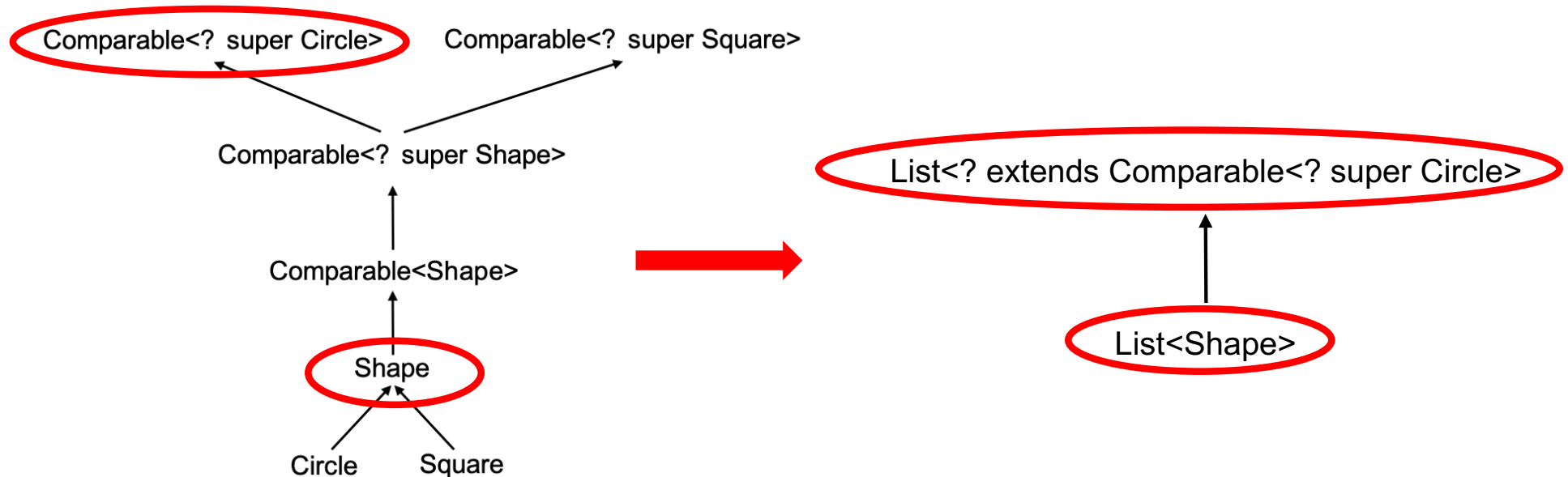
list key	List<Circle>	List<Square>	List<Shape>	List<Integer>
Circle	+	+	+	-
Shape	+	+	+	-
Square	+	+	+	-
Integer	-	-	-	+



Beispiel: Korrekte bzw. nicht-korrekte Parametertypen für `binarySearch(list,key)`

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

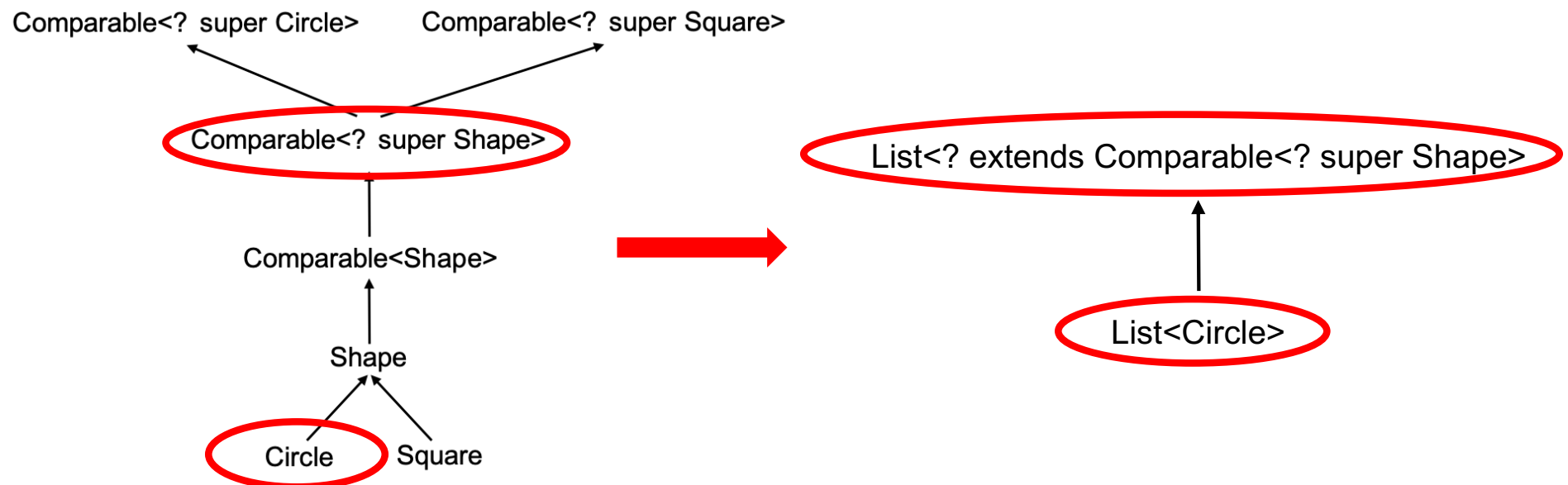
list key	List<Circle>	List<Square>	List<Shape>	List<Integer>
Circle	+	+	+	-
Shape	+	+	+	-
Square	+	+	+	-
Integer	-	-	-	+



Beispiel: Korrekte bzw. nicht-korrekte Parametertypen für `binarySearch(list,key)`

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

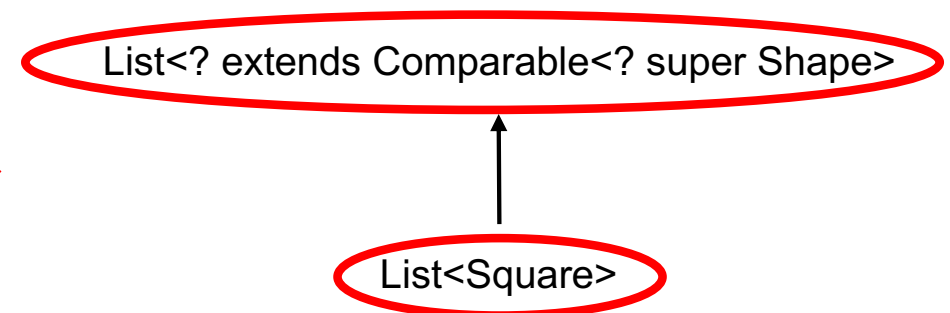
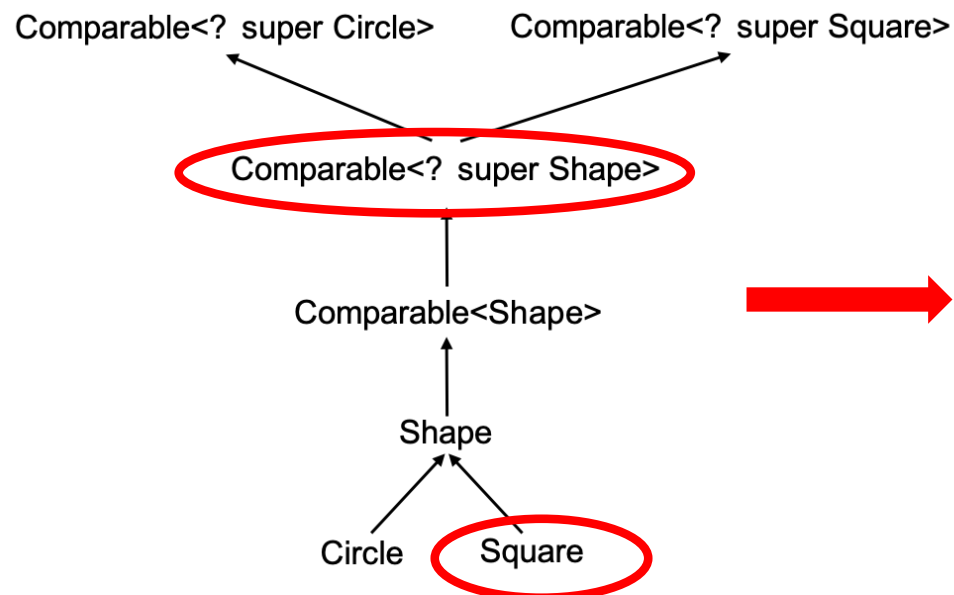
list	key	List<Circle>	List<Square>	List<Shape>	List<Integer>
Circle		+	+	+	-
Shape		+	+	+	-
Square		+	+	+	-
Integer		-	-	-	+



Beispiel: Korrekte bzw. nicht-korrekte Parametertypen für `binarySearch(list,key)`

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

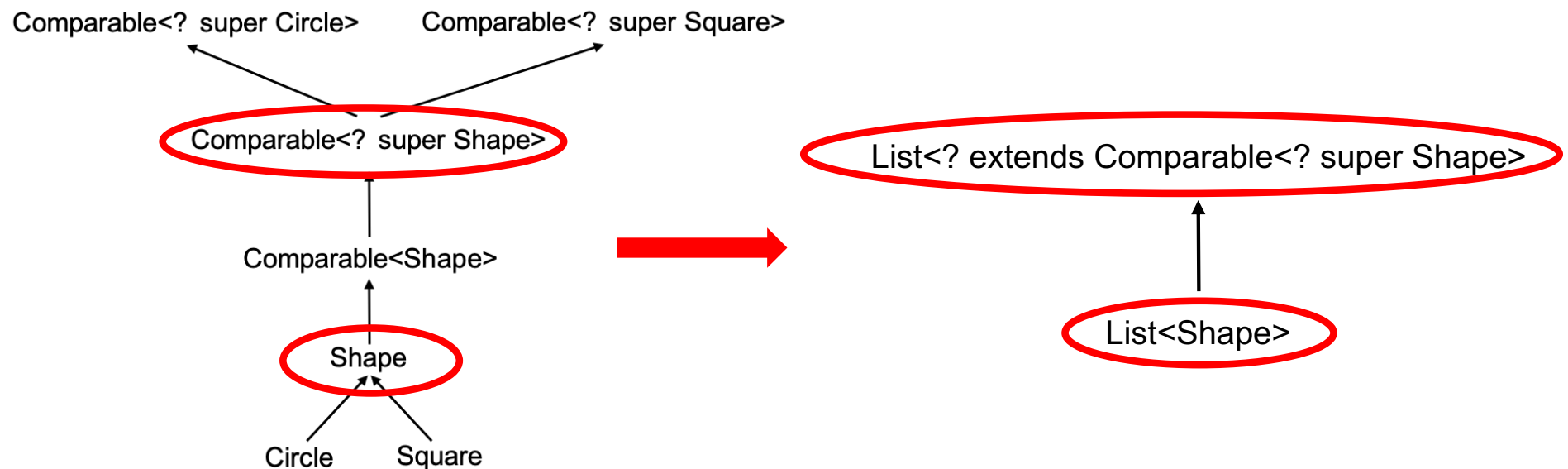
list key	List<Circle>	List<Square>	List<Shape>	List<Integer>
Circle	+	+	+	-
Shape	+	+	+	-
Square	+	+	+	-
Integer	-	-	-	+



Beispiel: Korrekte bzw. nicht-korrekte Parametertypen für `binarySearch(list,key)`

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

list key	List<Circle>	List<Square>	List<Shape>	List<Integer>
Circle	+	+	+	-
Shape	+	+	+	-
Square	+	+	+	-
Integer	-	-	-	+



Sortieren mit dem Interface List

```
interface List {  
    // ...  
    default void sort(Comparator<? super E> c)  
}
```

- Das Java Interface List hat eine default Methode `sort(cmp)`.
- Das Sortierverfahren ist eine mergeSort-Variante und ist damit **stabil**.

```
record Person(String name, int geb) { }  
  
List<Person> persList = new LinkedList<>();  
persList.add(new Person("Klaus", 1961));  
persList.add(new Person("Peter", 1959));  
persList.add(new Person("Maria", 1959));  
persList.add(new Person("Petra", 1961));  
persList.add(new Person("Albert", 1959));  
persList.add(new Person("Anton", 1961));  
persList.add(new Person("Iris", 1959));  
  
persList.sort( (p1, p2) -> p1.name().compareTo(p2.name()) );
```

Alphabetische Sortierung nach dem Namen

Vorgriff: Sortieren mit Streams

```
record Person(String name, int geb) { }  
List<Person> persList = new LinkedList<>();  
persList.add(new Person("Klaus", 1961));  
persList.add(new Person("Peter", 1959));  
persList.add(new Person("Maria", 1959));  
persList.add(new Person("Petra", 1961));  
persList.add(new Person("Albert", 1959));  
persList.add(new Person("Anton", 1961));  
persList.add(new Person("Iris", 1959));  
  
persList.stream()  
    .sorted((p1, p2) -> p1.name().compareTo(p2.name()))  
    .forEach(System.out::println);
```