3.2. Divide-and-Conquer-Methoden

Divide-and-Conquer-Methoden

- Einfache Sortieralgorithmen reduzieren die Größe des noch zu sortierenden Arrays pro Schritt lediglich um eins.
- **Ziel der Divide-and-Conquer Sortieralgorithmen:** Halbierung des zu sortierenden Arrays pro Schritt
- Bei Erreichung dieses Ziels: Laufzeit von $O(n \cdot \log n)$
- Allgemeiner Dive-and-Conquer-Algorithmus:

Divide-and-Conquer-Methoden

Allgemeiner Divide-and-Conquer-Algorithmus:

```
ALGORITHMUS DivideAndConquerSort;
  IF Objektmenge klein genug
     THEN löse das Problem direkt
     ELSE
        Divide:   Zerlege die Menge in Teilmengen
                  möglichst gleicher Größe

        Conquer:  Löse das Problem für jede der Teilmengen

        Merge:    Berechne aus den Teillösungen
                  die Gesamtlösung
  END;
```

MergeSort I

MergeSort:

Prinzip:

- Array wird rekursiv bis auf Elementebene zerlegt
- Jedes der Elemente ist trivialerweise sortiert
- Im Anschluss an diese Zerlegung:
  Gemäß der Zerlegungshierarchie: Verschmelzung (*Merge*) von jeweils zwei sortierten Teilarrays, so dass das Ergebnis wiederum sortiert ist.
- Nach dem letzten Merge-Schritt: gesamtes Array sortiert

MergeSort II

Allgemeiner Algorithmus:

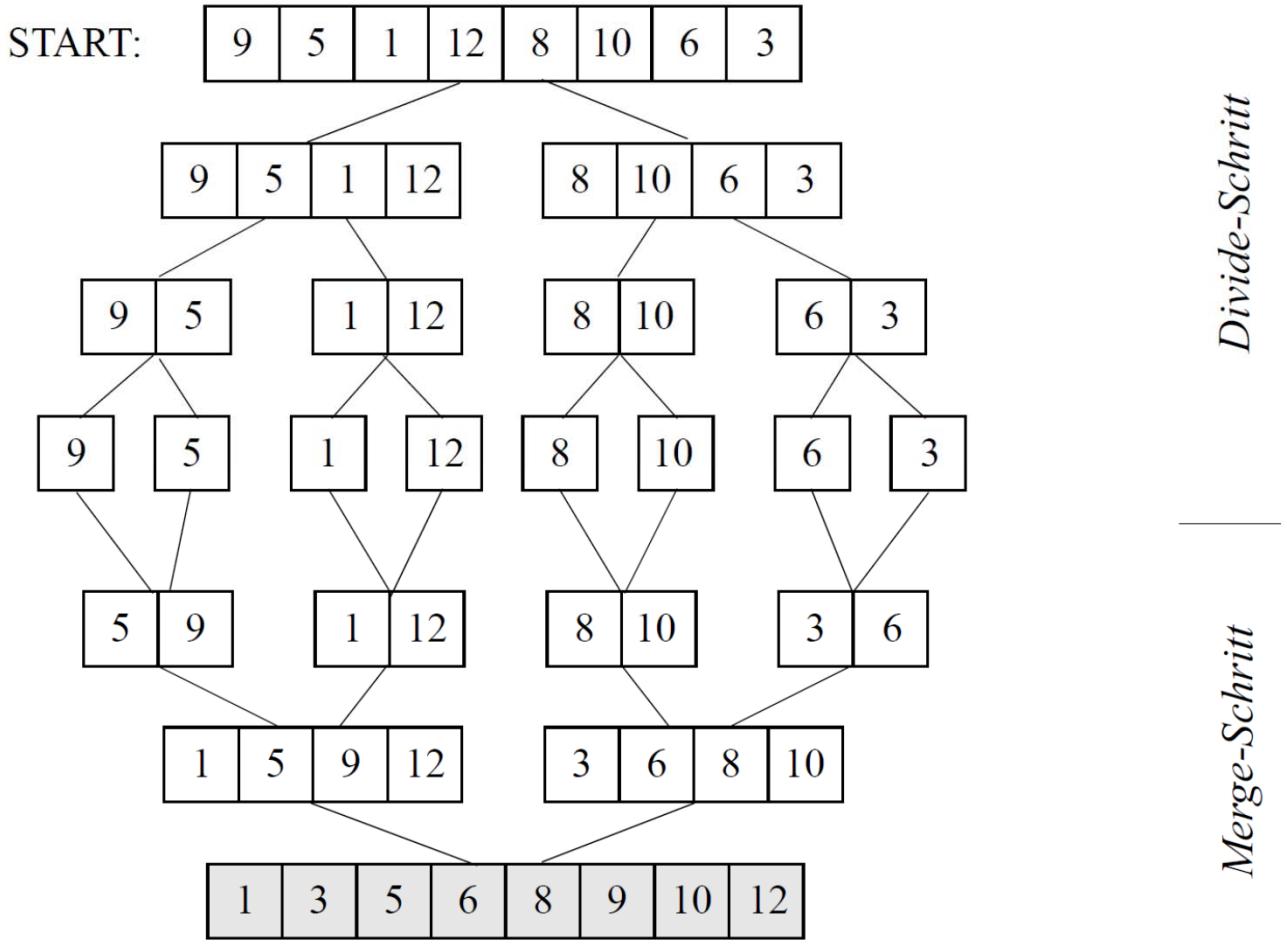
```
ALGORITHMUS MergeSort(S);
  IF |S|=1
     THEN RETURN S
     ELSE
        Divide:   S1 := a_1,…,a_⌊n/2⌋; S2 := a_⌊n/2⌋+1,…,a_n;

        Conquer:  S1‘ := MergeSort(S1);
                  S2‘ := MergeSort(S2);

        Merge:    RETURN Merge(S1‘, S2‘)
  END;
```

MergeSort II

Veranschaulichung:



MergeSort III

Laufzeitanalyse:

(Gilt im Wesentlichen für alle Divide-and-Conquer-Verfahren)

Laufzeit $T(n)$ für 1. Rekursionsschritt für n Elemente:

- **Laufzeit des Divide-Schrittes:**
  $\rightarrow O(1)$ (Aufteilen des Arrays $\rightarrow$ kontante Zeit)
- **Laufzeit des Conquer-Schrittes:**
  $\rightarrow 2 \cdot T\left(\frac{n}{2}\right)$ (n Elemente $\rightarrow 2 \cdot \frac{n}{2}$ Elemente)
- **Laufzeit des Merge-Schrittes:**
  $\rightarrow O(n)$ (jedes Element muss betrachtet werden)

MergeSort IV

Rekursionsgleichung für Zeitbedarf $T(n)$ des MergeSort:

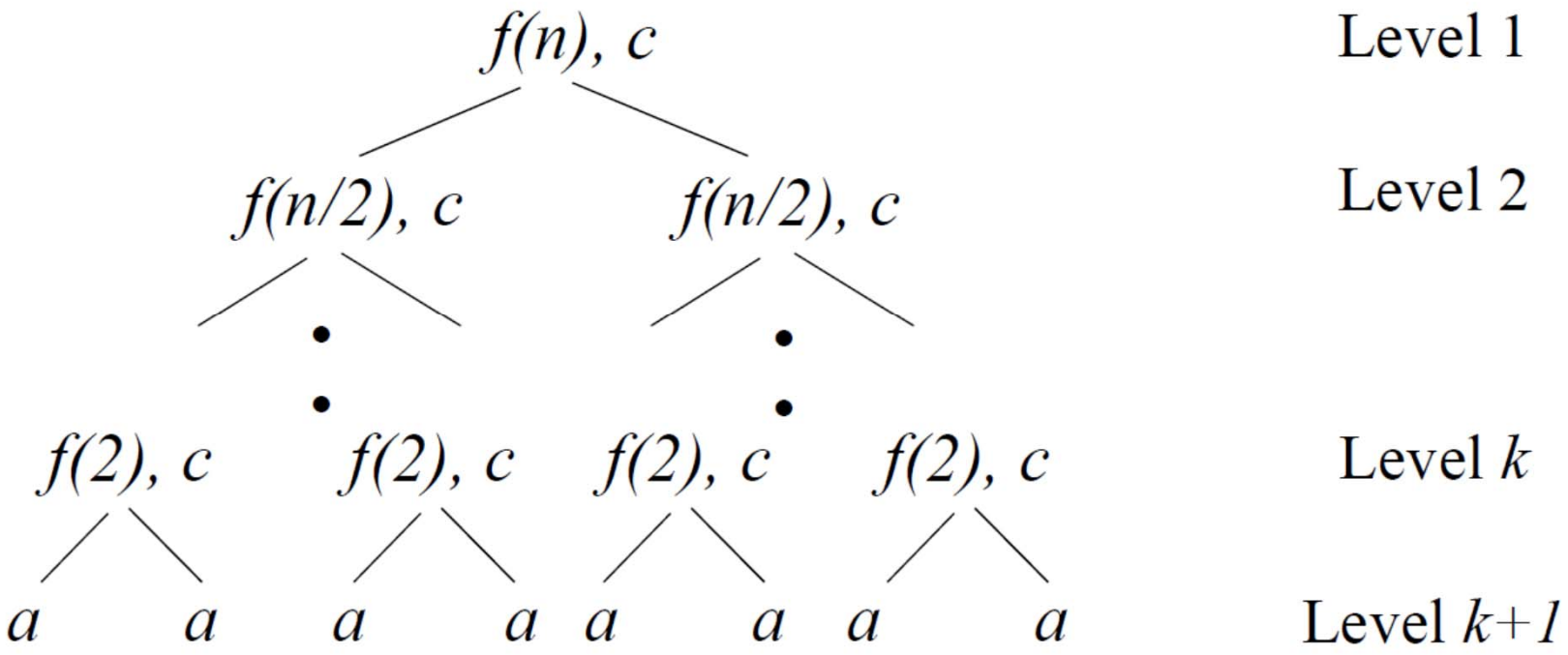
$$T(n) = \begin{cases} O(1) & falls\ n = 1 \\ O(1) + 2 \cdot T\left(\frac{n}{2}\right) + O(n) & falls\ n > 1 \end{cases}$$

oder anders ausgedrückt (mit Konstanten a und c und einer (mindestens) linear wachsenden Funktion f):

$$T(n) = \begin{cases} a & falls\ n = 1 \\ c + 2 \cdot T\left(\frac{n}{2}\right) + f(n) & falls\ n > 1 \end{cases}$$

MergeSort V

Zur Veranschaulichung:
Baum der Knoten der Rekursionshierarchie für $n = 2^k$:



MergeSort VI

- Aufwand für alle c's und a's: $(n-1) \cdot c + n \cdot a = O(n)$.
- Da f (mindestens) linear wächst, gilt: $i \cdot f\left(\frac{n}{i}\right) \leq f(n)$
- Hieraus ergibt sich folgende Abschätzung, wobei $k = \log n$:

Level 1: $f(n)$

Level 2: $2 \cdot f\left(\frac{n}{2}\right) \leq f(n)$

…

Level $k-1$: $\frac{n}{4} \cdot f(4) \leq f(n)$

Level k: $\frac{n}{2} \cdot f(2) \leq f(n)$

$\rightarrow T(n) \leq f(n) \cdot \log n \; = \; O(n \cdot \log n)$

MergeSort VII

Allgemeines Ergebnis:

- Jedes Divide-and-Conquer-Sortierverfahren, dessen Divide- und Merge-Schritt in $O(n)$ durchgeführt werden kann und das eine balancierte Unterteilung des Problems garantiert, besitzt eine Laufzeit von $O(n \cdot \log n)$.

QuickSort I

QuickSort (Hoare 1962)**:**

Prinzip:

- Auswahl eines (beliebigen) Schlüssels x aus dem Array
- **Divide-Schritt:**
  Zerlegung des Arrays in Schlüssel $\geq x$ und Schlüssel $< x$
- **Conquer-Schritt:**
  Die beiden resultierenden Teilarrays werden rekursiv bis auf Elementebene in gleicher Weise behandelt.
- **Kein Merge-Schritt** durch entsprechende Speicherung der Teilarrays innerhalb des Arrays.

QuickSort II

Allgemeiner Algorithmus:

```
ALGORITHMUS QuickSort(S);
  IF |S|=1
     THEN RETURN S
     ELSE
        Divide:  Wähle irgendeinen Schlüsselwert x=s_j.key
                 aus S aus. Berechne eine Teilfolge S1
                 aus S mit den Elementen, deren
                 Schlüsselwert < x ist und eine Teilfolge
                 S2 mit Elementen >= x.

        Conquer: S1‘ := QuickSort(S1);
                 S2‘ := QuickSort(S2);

        Merge:   RETURN Concat(S1‘, S2‘)
  END;
```

QuickSort III

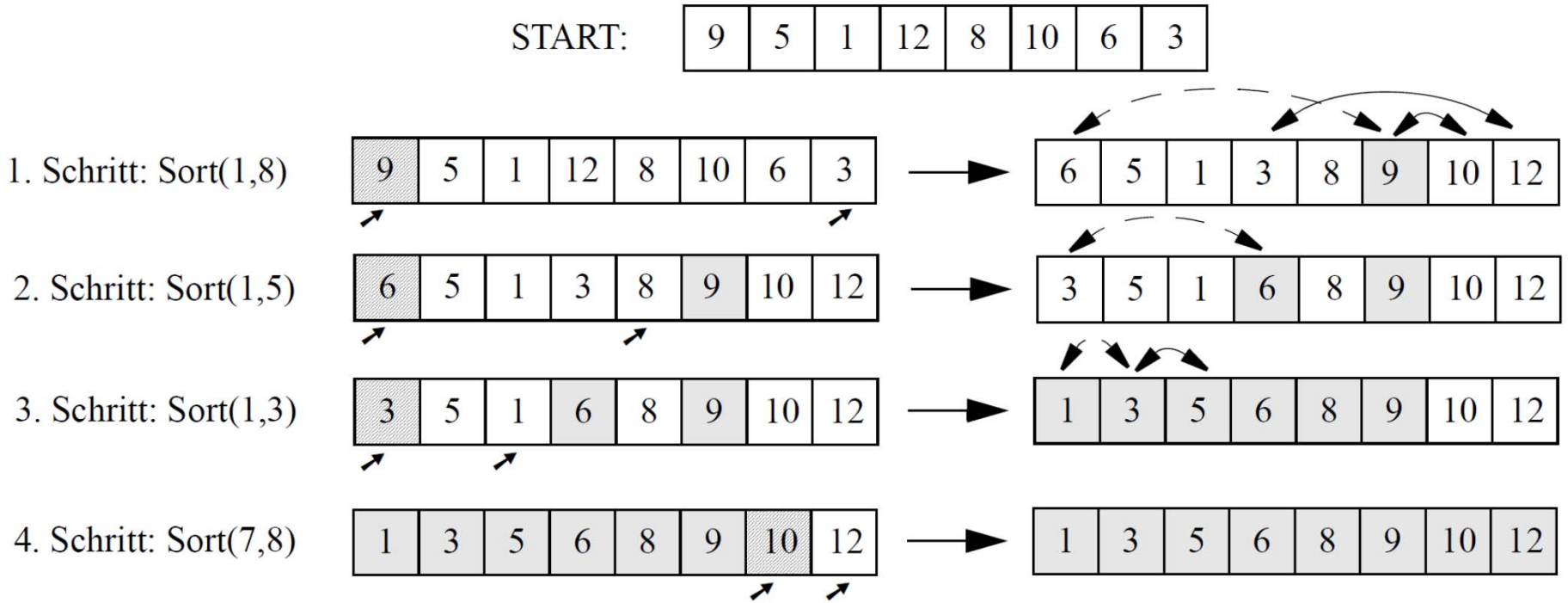
Algorithmus:

```
void quickSort() {
   sort(1, n);
}
```

```
void sort(int li, int r) {
   int i = li + 1;
   int j = r;
   Entry x = a[li]; // hier: Auswahl des linken Randelementes
   boolean flag = false;
   do { // Aufteilung in Elemente > x, bzw. Elemente < x
      while ((a[i].key < x.key) && (i < r)) i = i+1;
      while (a[j].key > x.key) j = j-1;
      if (i<j) {
         vertausche(i, j);
         i = i+1; j = j-1;
      } else flag=true;
   } while (!flag);
   vertausche(li, j); // x an richtige Stelle bringen
   if (li < j-1) sort(li, j-1);
   if (j+1 < r) sort(j+1, r);
}
```

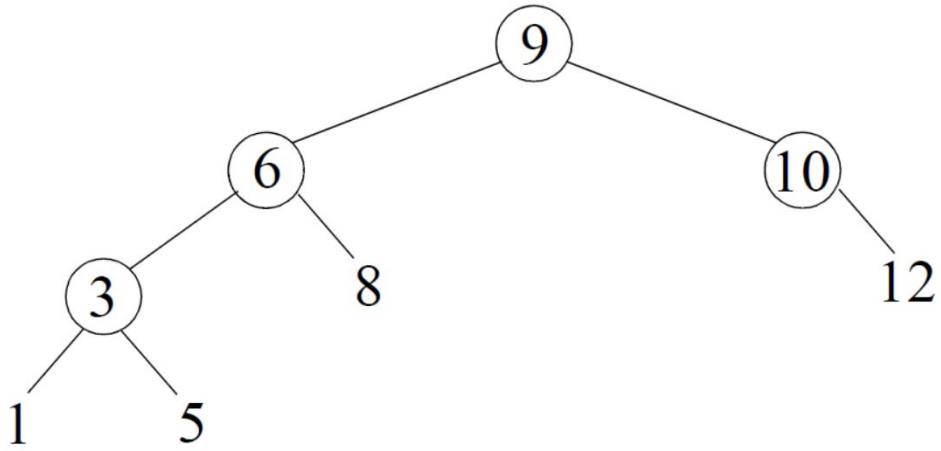
QuickSort IV

Veranschaulichung:



QuickSort V

- Rekursive Unterteilung des Arrays bezüglich des ausgewählten Elementes
  → binärer Baum über den Arrayelementen



- Höhe des Baumes → Rekursionstiefe des Verfahrens erwünscht:
  balancierter binärer Baum → Rekusionstiefe: $O(\log n)$

QuickSort VI

Laufzeitanalyse:

Zunächst Untersuchung des Zerlegungsschrittes:

Annahme:

Wahl des k-ten Elementes der Sortierungsfolge für den Zerlegungsschritt und Positionierung dieses Elementes an Position k:

- Divide-Schritt:
  In jedem Schritt wird jedes Elementes betrachtet

  $\rightarrow n$ Vergleiche

QuickSort VII

- Anzahl der Vertauschungsoperationen abhängig von k:
  - (Anzahl der Elemente $< k$) · (Wahrscheinlichkeit für Vertauschung)
    (= Wahrscheinlichkeit, dass das Element im rechten Teil war)
    $= (k-1) \cdot \frac{n-(k-1)}{n}$
  - Mittelwert $\overleftarrow{M(n)}$ der Vertauschungsoperationen über alle $k = 1,2,\dots,n$ für den ersten Zerlegungsschritt

$$\overleftarrow{M(n)} = \frac{1}{n} \cdot \sum_{k=1}^{n} (k-1) \cdot \frac{n-(k-1)}{n} = \frac{1}{n^2} \cdot \sum_{k=0}^{n-1} k \cdot (n-k) =$$

$$= \frac{n \cdot (n-1)}{2 \cdot n} - \frac{2 \cdot n^2 - 3 \cdot n + 1}{6 \cdot n} = \frac{n - \frac{1}{n}}{6} = O(n)$$

→ der Divide-Schritt benötigt stets $O(n)$ Operationen.

QuickSort VIII

Gesamtzeitbedarf:

- **Günstigster Fall:**
  Array wird jeweils in zwei Teilarrays etwa gleicher Größe geteilt

$$T_{best}(n) = c \cdot n + 2T\left(\frac{n}{2}\right) = c \cdot n + 2\left(c \cdot \frac{n}{2}\right) + 4T\left(\frac{n}{4}\right) =$$

$$= 2cn + 4T\left(\frac{n}{4}\right) = \cdots = \log n \cdot cn + n \cdot T(1) = O(n \cdot \log n)$$

- **Durchschnittlicher Fall:**
  Array wird jeweils bezüglich eines zufällig ausgewählten Elements zerlegt.

  Nach Analysen:
  Nur um den Faktor $2 \cdot \ln(2) \approx 1{,}39$ (ca. 40%) schlechter

  $\rightarrow T_{\emptyset}(n) = O(n \cdot \log n)$

QuickSort IX

- **Schlechtester Fall:**
  Degenerierung (lineare Liste); stets wird das größte/kleinste Element gewählt. (für die obige Implementierung im Falle eines bereits sortierten Elementes)

  $$T_{worst}(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n \cdot (n-1)}{2} = O(n^2)$$

  Beachte hierbei auch die hohe Rekursionstiefe n.

QuickSort X

Bemerkungen:

- Austauschen von Schlüsseln über große Distanzen
  → schnell ‚nahezu' sortiert
- QuickSort schlecht für kleine n (wie alle komplexen Verfahren)
  → Verbesserung durch direktes Sortieren kleiner Teilarrays
- Methoden, um die Wahrscheinlichkeit des schlechtesten Falls zu vermindern:

  → Zerlegungselement x als Median aus 3 oder 5 Elementen bestimmt

  → ‚Clever QuickSort'

  dennoch: QuickSort bleibt immer eine Art ‚Glücksspiel', denn es bleibt:

  $$T_{worst}(n) = O(n^2)$$

- QuickSort war lange Zeit das im Durchschnittsverhalten beste bekannte Sortierverfahren.

3.3. Sortieren mit Hilfe von Bäumen

Sortieren mit Hilfe von Bäumen

Sortieren durch Auswählen:

- n-maliges Extrahieren des Minimums/Maximums einer Folge von n Schlüsseln
- Verwenden einer geeigneten Baumstruktur, die diese Operation effizient unterstützt

Heapsort I

HeapSort (Williams 1964)

Definition: Heap:

Sei $K = \{k_1, k_2, \ldots, k_n\}$ eine Menge von Schlüsseln und T ein binärer Baum, dessen Knoten die Schlüssel aus K speichern.

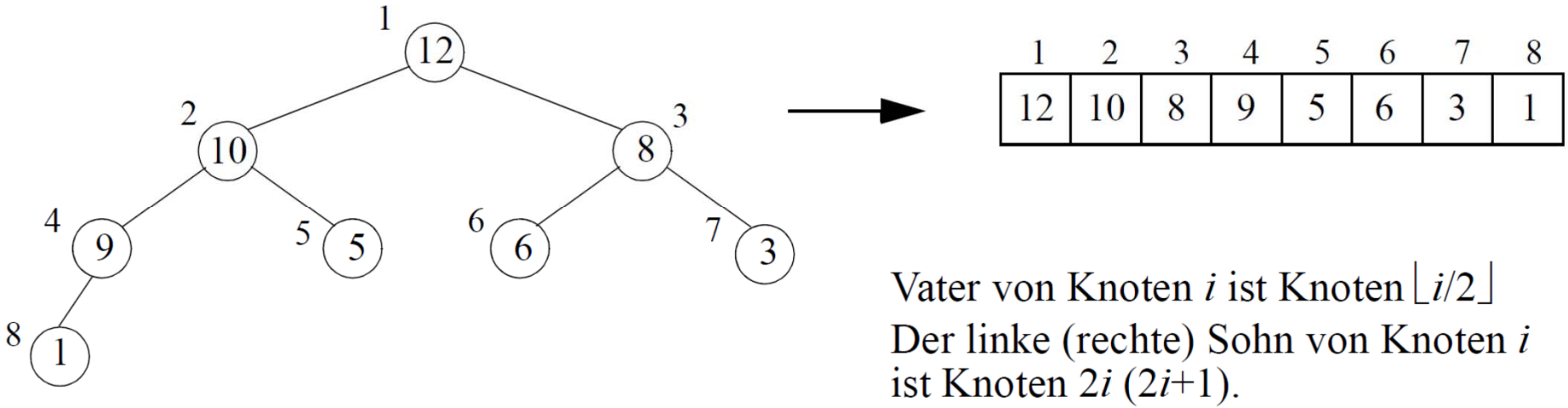
T ist genau dann ein Heap, wenn gilt:

- T ist vollständig ausgeglichen und
- Der Schlüssel in jedem Knoten ist größer oder gleich den Schlüsseln in den Söhnen des Knotens (*partielle Ordnung*).

Heapsort II

Ein Heap ist auf sehr einfache Art mit Hilfe eines sequentiellen Arrays implementierbar.

Beispiel:



Vater von Knoten i ist Knoten $\lfloor i/2 \rfloor$
Der linke (rechte) Sohn von Knoten i ist Knoten $2i$ $(2i+1)$.

Heapsort III

- Eine Folge von Schlüsseln $k_1, k_2, \ldots, k_n$ bildet einen sequentiellen Heap, wenn gilt:

$$k_{\lfloor \frac{j}{2} \rfloor} \geq k_j \quad \text{für} \quad 1 \leq \left\lfloor \frac{j}{2} \right\rfloor < j \leq n$$

- In diesem Fall gilt: $k_1 = \max_{1 \leq i \leq n}(k_i)$, d.h. das Maximum steht in der Wurzel.

- Weiterhin erfüllen alle Blätter $k_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, k_n$ für sich genommen bereits die Heap-Eigenschaft.

Heapsort IV

Prinzip:

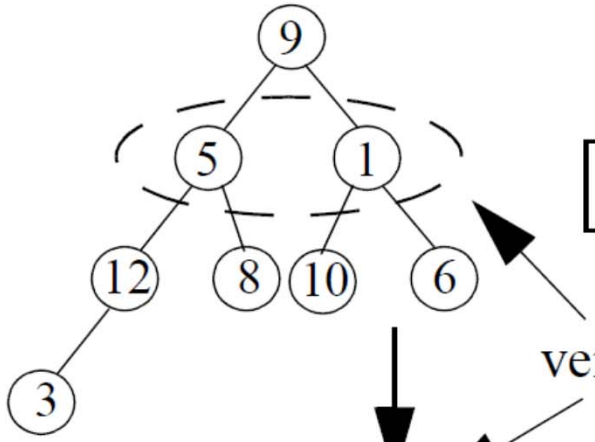
- Die zu sortierende Schlüsselfolge wird in einem Vorbereitungsschritt in eine sequentielle Heap-Struktur überführt.
- Aus dieser Struktur wird in jedem Schritt das Maximum (die Wurzel) entnommen und der Rest wiederum in Heap-Struktur gebracht.
- Eine sukzessive n-malige Entnahme des Maximums führt zur Sortierung der Folge.

Heapsort IV

Aufbau der Heap-Struktur:

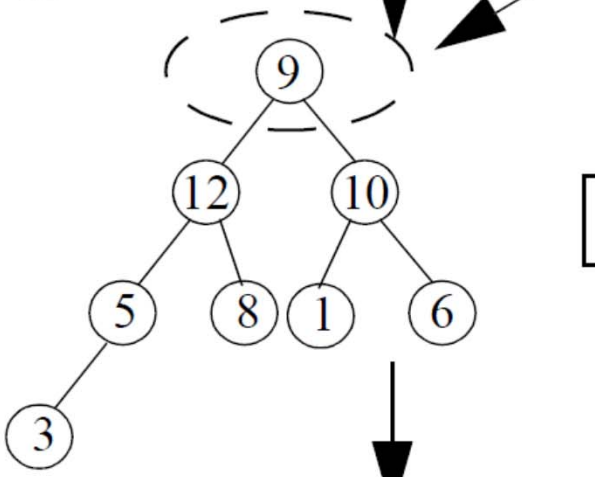
Ausgangspunkt: unsortiertes Array

- Die $n \; DIV \; 2$ Elemente in den Blättern erfüllen die Heap-Eigenschaft.
- ‚Absenken' aller inneren Schlüssel, die diese Eigenschaft verletzen
  - Erfolgt durch Vertauschen des Schlüssels mit dem größeren seiner Söhne (*sukressive über alle Level*).
- **Ergebnis:** ‚Anfangsheap'
  Der maximale Schlüssel befindet sich in der Wurzel

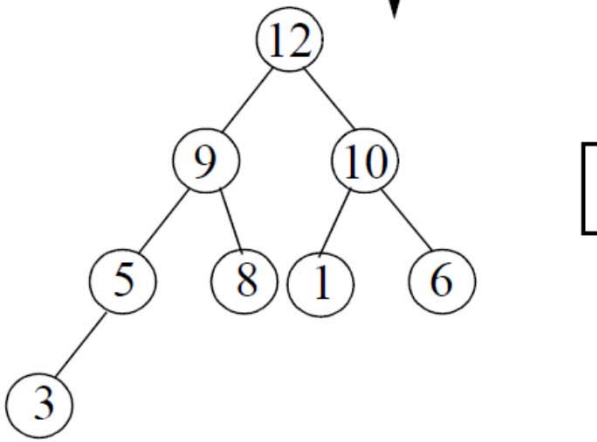


9	5	1	12	8	10	6	3

verletzen die Heap-Eigenschaft.



9	12	10	5	8	1	6	3



12	9	10	5	8	1	6	3

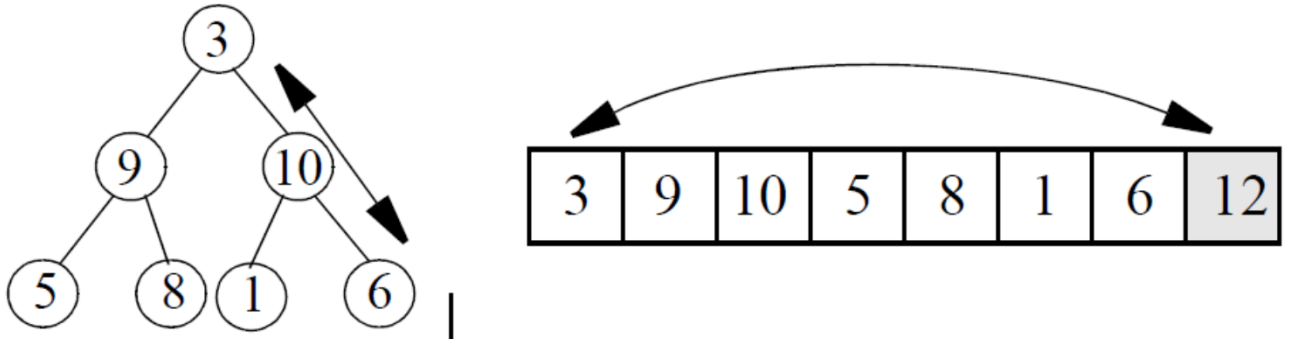
Heapsort VI

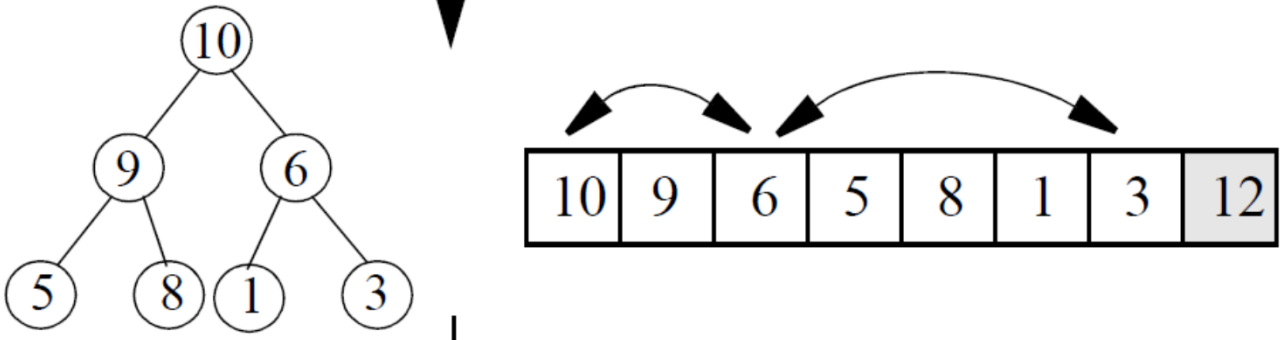
Sortierschritt:

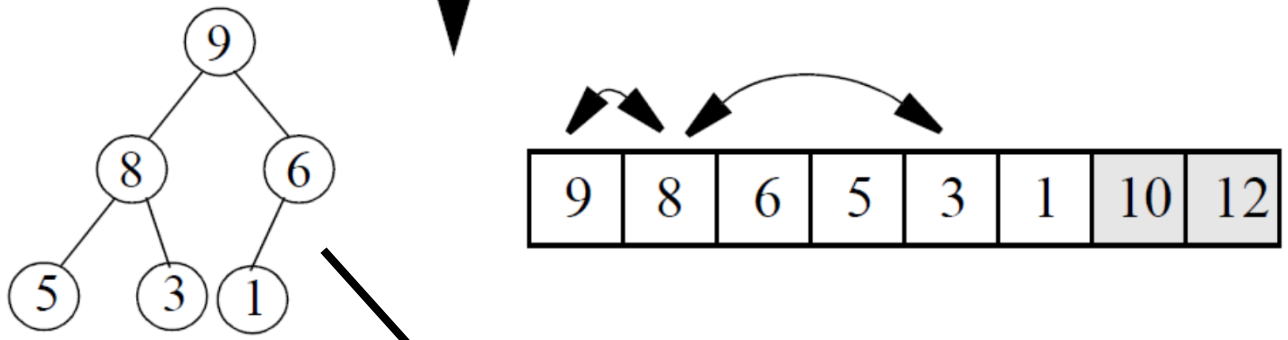
Vertausche: Maximum ↔ letztes Element

- ‚Absenken', um Heap-Eigenschaft zu gewährleisten
- Wiederum steht das Maximum in der Wurzel

Iteration dieses Schrittes, bis das Array sortiert ist.

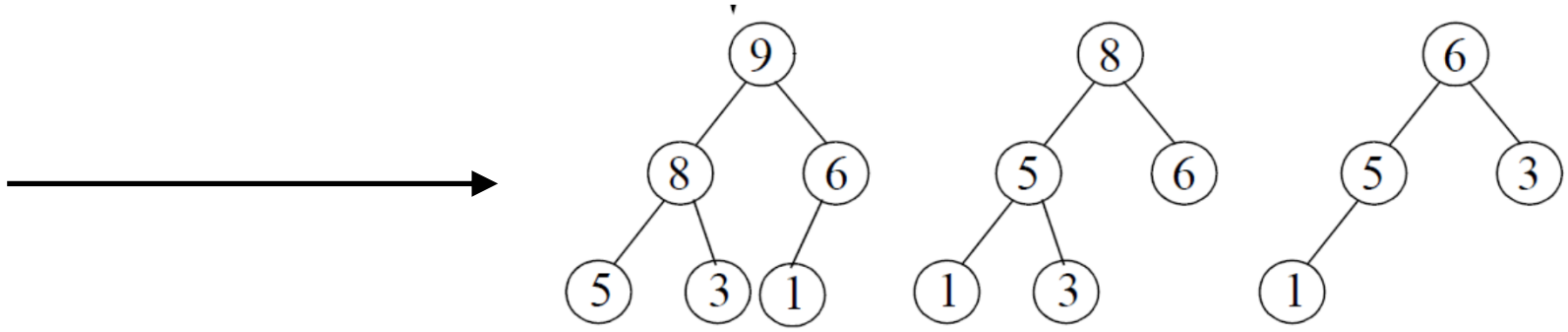




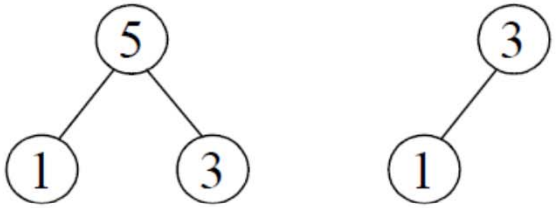


nächste Folie

Heapsort VIII



sortiert:	10,12	9,10,12	8,9,10,12
Heapgröße:	6	5	4



6,8,9,10,12	5,6,8,9,10,12	3,5,6,8,9,10,12	1,3,5,6,8,9,10,12
3	2	1	0