

Sortieralgorithmen

Themenverzeichnis

Sortieren mit Java-Werkzeugen

- Ordnungskriterien
- *interface Comparable, Comparator*

Bewertungskriterien von Algorithmen

- Zeit- und Speicherkomplexität
- Stabilität

Quadratische Algorithmen

- Einfügen (InsertionSort)
- Auswählen (SelectionSort)
- BubbleSort

Unterquadratische Algorithmen

- QuickSort
- QuickSelect
- Mischen (MergeSort)
- Shell-Sort
- Radix-Verfahren

Bäume

- Knoten, Wurzel, Kante

Binäre Bäume (BB)

- Java-Darstellung

Suchen, Einfügen und Löschen in sortierten binären Bäumen

Heap (Halde)

- HeapSort
- Darstellung des Heaps als Feld
- Aufbau des Heaps aus einer Reihung

Prioritätswarteschlangen

Optimierungen von MergeSort

- Sortierkanal
- Mischkanal
- Mehrphasen-Mischen

Sortieren mit Java-Werkzeugen

Sortieren: gut bekannter und fassbarer Begriff →

→ Objekte einer Sammlung nach bestimmten Attributen
(Sortierkriterien) aufsteigend oder absteigend ordnen.

Sortierung ist oft nützlich als Vorbereitungsschritt, der eine effiziente nachfolgende Verarbeitung ermöglicht (z.B. Binary Search, Gruppierungen, statistische Analysen u.ä.).

Jedes Sortierkriterium ist mit einem Schlüssel verbunden: einem beliebigen Objekt der Sammlung (Eintrag) **e** wird ein Schlüsselwert **key(e)** zugeordnet, wobei die Schlüsselwerte geordnet sind, d.h. man kann die Reihenfolge von zwei Schlüsselwerten mit einem Vergleich (<, >, ==) prüfen. Aus der Schlüssel-Reihenfolge ergibt sich dann die Reihenfolge der Einträge:

key(e₁) < key(e₂) ist gleichbedeutend mit **e₁ < e₂**

Bei Schlüssel-Gleichheit **key(e₁) == key(e₂)** ist die Reihenfolge von **e₁** und **e₂** unbestimmt.

Für primitive Datentypen (*int*, *double*) ist ihre Grösse ein natürliches Sortierkriterium, *String* – Objekte werden alphabetisch geordnet, d.h. nach dem numerischen Character-Wert (ASCII-Tabelle).

(In bestimmten Umgebungen kann aber das Vergleichen von Texten auch vom verwendeten Character-Set abhängig sein → siehe z.B. *Order By* - Klausel in SQL).

Für benutzerdefinierte Objekte kann ihre Ordnung z.B. in der Klassendefinition festgelegt werden: Java bietet dafür interface *Comparable* :

```
class Telefonbucheintrag implements Comparable<Telefonbucheintrag> {  
    private String name; // Schlüssel  
    private String telefonnummer; // Daten  
    public int compareTo(Telefonbucheintrag that) { // 2  
        return this.name.compareTo(that.name);  
    }  
    ...  
}
```

Wenn Elemente einer Sammlung *Comparable* implementieren, kann diese Sammlung mit entsprechenden (statischen) Bibliotheksmethoden sortiert werden:

Arrays.sort(myArray)
Collections.sort(myCollection)

Die obige Technik erlaubt allerdings nur das Sortieren nach einem festgelegten Sortierkriterium. Für einen flexiblen Einsatz von verschiedenen Kriterien steht interface *Comparator* zur Verfügung:

```
class Kunde { int kdNr; ... } // weitere Daten
class Kundenvergleich implements Comparator<Kunde>
public int compare (Kunde o1, Kunde o2) { return o1.kdNr < o2.kdNr ? -1 : 1; }
}

...
Kunde[] kartei = ...
sort(kartei, new Kundenvergleich()); // Aufruf der generischen Methode
```

Für jedes Sortierkriterium muss eine *Comparator* - implementierende Klasse definiert und beim Aufruf der Sortiermethode instanziiert werden:

```
sort(kartei, new Geburtsdatumvergleich());
```

Bewertungskriterien von Algorithmen

Die oben beschriebenen Standardwerkzeuge sind im Allgemeinfall ausreichend leistungsfähig. In bestimmten Situationen kann aber der Einsatz eines spezifischen Algorithmus von Vorteil sein.


Die Merkmale der Leistungsfähigkeit von verschiedenen Algorithmen kann man wie folgt zusammenfassen:

- Zeitkomplexität in durchschnittlichen / günstigen / ungünstigen Fällen
- Zusätzlicher Speicherbedarf, Speicherkomplexität
- Einsatzfähigkeit für Input von externen Medien, wenn Arbeitsspeicher für die Datenmenge nicht ausreicht
- Stabilität

Stabilität

Ein Sortierverfahren heißt *stabil*, wenn es die relative Reihenfolge gleicher Schlüssel in der Datei beibehält.

- Beispiel: alphabetisch geordnete Liste von Personen soll nach Alter sortiert werden → Personen mit gleichem Alter weiterhin alphabetisch geordnet

<u>Name</u>	<u>Alter</u>		<u>Name</u>	<u>Alter</u>
Endig, Martin	30		Höpfner, Hagen	24
Geist, Ingolf	28		Geist, Ingolf	28
Höpfner, Hagen	24		Schallehn, Eike	28
Schallehn, Eike	28		Endig, Martin	30

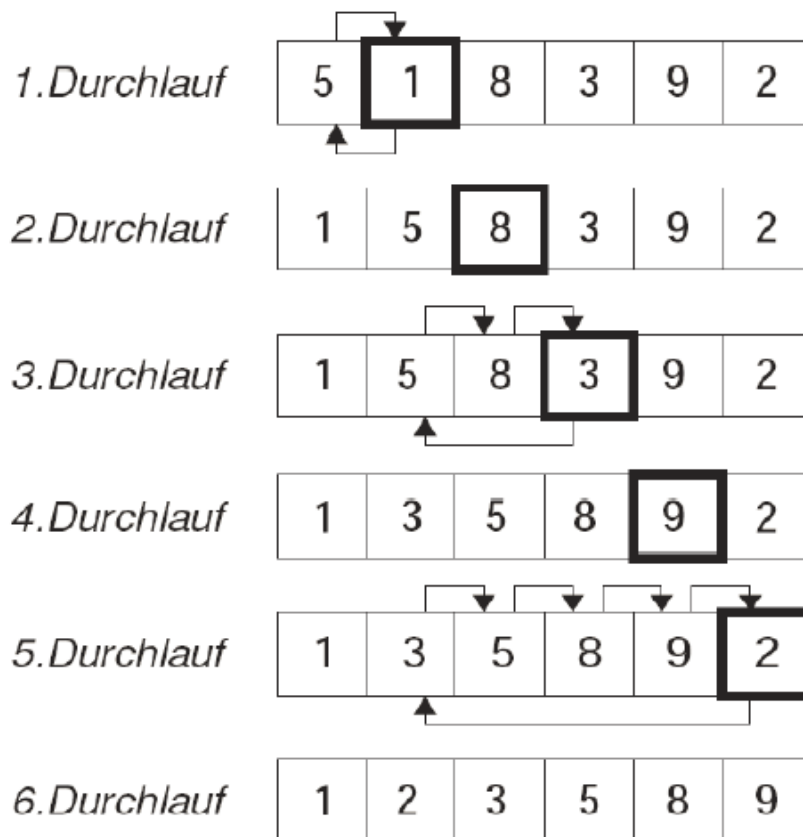
Beim Sortieren via Bibliothek-Methoden (s. oben) ist die Stabilität nicht garantiert.

Zwischenfrage: Wie könnte man die Stabilität beim Einsatz von solchen Methoden trotzdem erreichen?

Einfügen (InsertionSort)

Diese Methode entspricht dem üblichen Vorgehen beim manuellen Sortieren. Die sortierte Folge wird schrittweise (Schritt = Durchlauf) am linken Rand aufgebaut, der Anfang des unsortierten rechten Teils („Grenzstein“) ist fett markiert.

Sortieren durch Einfügen: Beispiel



Bei jedem Durchlauf nimmt man zuerst den Grenzstein heraus, dann wandert man damit nach links durch die geordnete Folge bis zur Stelle, die seinem Wert entspricht, wobei übersprungene Elemente höheren Wertes nach rechts geschoben werden. An der gefundenen (leergewordenen) Stelle wird anschliessend das herausgenommene Element eingefügt.

Umsetzung:

```
public void sort(E[] sammlung) {
    for (int index = 1; index < sammlung.length; index++) { // anfangen beim 2. Element
        final E elementZumEinfuegen = sammlung[index];
        int einfuegestelle = index;
        while (einfuegestelle > 0 &&
            elementZumEinfuegen.compareTo(sammlung[einfuegestelle-1]) < 0) {
            sammlung[einfuegestelle] = sammlung[einfuegestelle-1];
            // nach oben schieben
            einfuegestelle --;
        }; // Einfuegestelle gefunden: entweder weil einfuegestelle = 0, oder
        // weil !elementZumEinfuegen.compareTo(sammlung[einfuegestelle - 1]) < 0
        sammlung[einfuegestelle] = elementZumEinfuegen;
    }
}
```

Analyse: InsertionSort

■ Aufwand:

- Anzahl der Vertauschungen
- Anzahl der Vergleiche
- Vergleiche **dominieren** Vertauschungen, d.h. es werden (wesentlich) mehr Vergleiche als Vertauschungen benötigt

■ Ausserdem Unterscheidung:

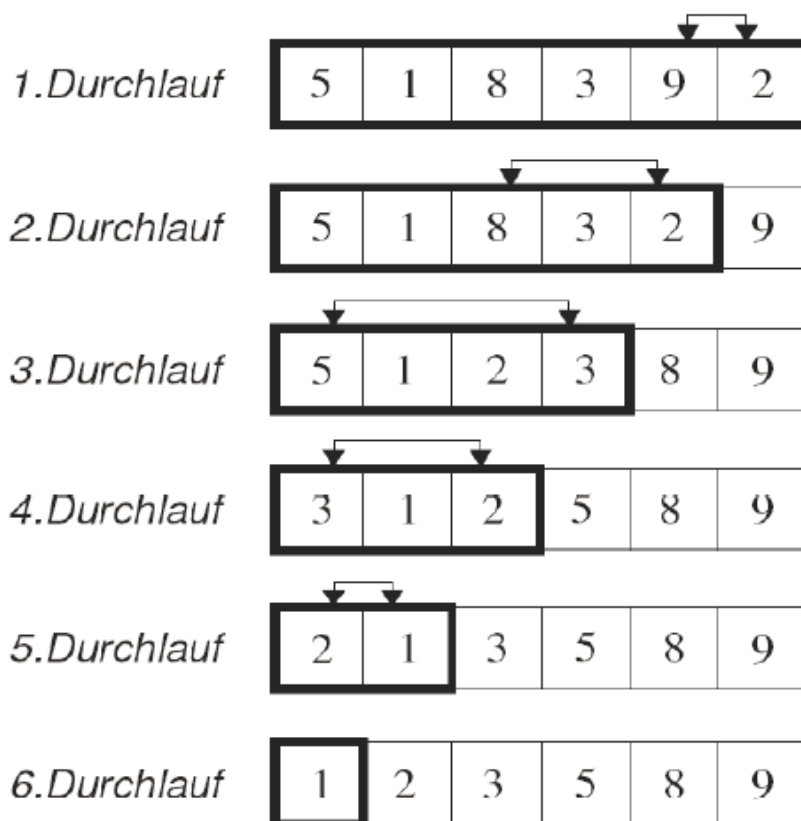
- Bester Fall: Liste ist schon sortiert
- Mittlerer (zu erwartender) Fall: Liste ist unsortiert
- Schlechtester Fall: z.B. Liste ist absteigend sortiert

Komplexitätsklasse im Durchschnittsfall: $O(n^2)$

Auswählen (SelectionSort)

Eine einfache, „gerade aus“ - Methode. Die sortierte Folge wird schrittweise (Schritt = Durchlauf) am rechten Rand aufgebaut, der unsortierte linke Teil ist fett markiert.

Sortieren durch Selektion: Beispiel



Bei erstem Durchlauf sucht man das Maximum von sämtlichen n Elementen und vertauscht es dann mit der rechten Randstelle. Dann sucht man das Maximum der unsortierten linken ($n-1$) - Subsequenz und vertauscht es mit der vorletzten Stelle rechts, usw.

Umsetzung:

```
public void sort(E[] sammlung) {
    for (int index = 0; index < sammlung.length-1; index++) {
        // bis zum vorletzten Element
        int austauschstelle = index;
        E elementZumAustauschen = sammlung[index];

        for (int kleinstes = index + 1; kleinstes < sammlung.length;
             kleinstes++) {
            if (sammlung[kleinstes].compareTo(elementZumAustauschen) < 0) {
                austauschstelle = kleinstes; // index merken
                elementZumAustauschen = sammlung[kleinstes];
            }; // Austauschstelle gefunden
        }
        if (austauschstelle != index) { // Austausch nötig
            sammlung[austauschstelle] = sammlung[index];
            sammlung[index] = elementZumAustauschen;
        }
    }
}
```

Analyse: SelectionSort

- in jedem Durchlauf das Element von $F[p]$ mit dem größten Element tauschen
- Variable p läuft von $n \dots 1$
⇒ daher n Vertauschungen
- in jedem Durchlauf das größte Element aus $1 \dots p$ ermitteln
- ⇒ $p - 1$ Vergleiche

$$(n-1) + (n-2) + (n-3) + \dots + 2 + 1 = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

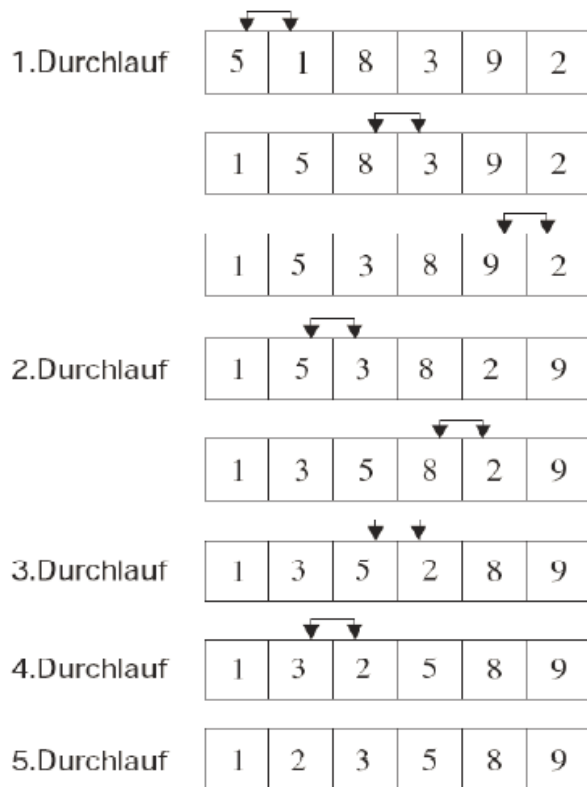
- **Anzahl Vergleiche identisch für besten, mittleren und schlechtesten Fall!**

Komplexitätsklasse: $O(n^2)$

BubbleSort

Eine weit verbreitete, sehr bekannte Methode.

BubbleSort: Beispiel



```
algorithm BubbleSort (F)
Eingabe: zu sortierende Folge  $F$  der Länge  $n$ 
do
  for  $i := 1$  to  $n - 1$  do
    if  $F[i] > F[i + 1]$  then
      Vertausche Werte von  $F[i]$  und  $F[i + 1]$ 
    fi
  od
until keine Vertauschung mehr aufgetreten
```

Bei jedem Durchlauf iteriert man durch die ganze Folge und vergleicht jedes Element mit seinem Nachfolger. Wenn das Paar nicht die gewünschte (hier aufsteigende) Reihenfolge hat, werden die zwei Elemente miteinander vertauscht.

Der Vorgang wird beendet, wenn bei einem Durchlauf keine Vertauschungen stattfinden.

Umsetzung:

```
class BubbleSort<E extends Comparable<E>> implements Sort<E> {
    public void sort(E[] sammlung) {
        for (int i = 0; i < sammlung.length; i++)
            for (int j = 0; j < sammlung.length-1; j++)
                if (sammlung[j+1].compareTo(sammlung[j]) < 0) { // vergleichen
                    final E temp = sammlung[j+1]; // austauschen
                    sammlung[j+1] = sammlung[j];
                    sammlung[j] = temp;
                }
    }
}
```

Es gibt mehrere optimierte Varianten von BubbleSort.

Beispiel:

BubbleSort: Optimierung

Beobachtung: Größte Zahl wird in jedem Durchlauf automatisch an das Ende der Liste verschoben

⇒ im Durchlauf j reicht der Vergleich bis Position $n - j$

Analyse: BubbleSort

- Bester Fall: n
- Durchschnittlicher Fall
 - Normal: n^2
 - optimiert: $n^2 / 2$
- Schlechtester Fall
 - normal: n^2
 - optimiert: $n^2 / 2$

Komplexitätsklasse im Durchschnittsfall: **$O(n^2)$**

Eine andere Optimierungsvariante (ShakerSort): es wird abwechselnd links → rechts und rechts → links iteriert.

```
public void sort(E[] sammlung) { // 1
    boolean austausch;
    int links = 1; // anfangen beim zweiten Element
    int rechts = sammlung.length-1;
    int fertig = rechts;
    do {
        austausch = false;
        for (int ab = rechts; ab >= links; ab--) // abwärts
            if (sammlung[ab].compareTo(sammlung[ab-1]) < 0) {
                austausch = true; fertig = ab;
                final E temp = sammlung[ab-1];
                sammlung[ab-1]=sammlung[ab]; sammlung[ab]=temp;
            }
        links = fertig + 1;
        for (int auf = links; auf <= rechts; auf++) // aufwärts
            if (sammlung[auf].compareTo(sammlung[auf-1]) < 0) {
                austausch = true; fertig = auf;
                final E temp = sammlung[auf-1];
                sammlung[auf-1] = sammlung[auf]; sammlung[auf] = temp;
            }
        rechts = fertig - 1;
    } while (austausch);
}
```

Damit kann „die schwerste Blase zuoberst“ in einem Durchgang zu ihrer definitiven Stellung gesenkt werden (wobei sie mit der nicht optimierten Variante viele Durchgänge benötigen würde).

Die bisher gezeigten „geradelinigen“ Verfahren haben die Zeitkomplexität **$O(n^2)$** .

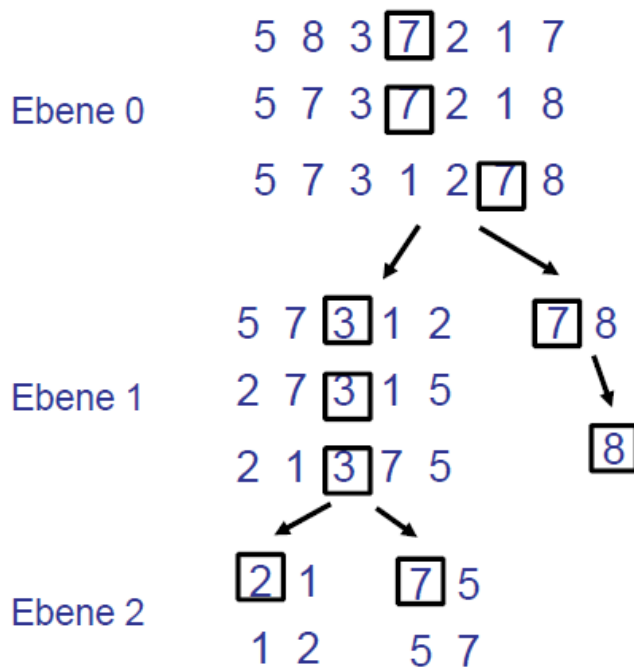
Es folgen komplexere (meistens rekursive) Ansätze, die eine bessere Zeitkomplexität erreichen. Die einfachen **$O(n^2)$** – Verfahren haben jedoch in bestimmten Situationen Vorteile, z.B.

- bei einer kleinen Anzahl von Objekten sind sie schnell
- sie benötigen keinen zusätzlichen Speicherplatz
- InsertionSort ist für „beinahe sortierte“ Sammlungen sehr effizient

QuickSort

Sehr effektive Methode, wird häufig eingesetzt.

QuickSort: Beispiel



```
algorithm QuickSort (F, u, o)
Eingabe: eine zu sortierende Folge F,
die untere und obere Grenze u, o
```

```
if o > u then
  Zerlege F[u...o] in zwei Partitionen
    F[u...r] und F[l...o];
  QuickSort (F, u, r);
  QuickSort (F, l, o);
fi
```

```
algorithm Zerlege (F, u, o)
Eingabe: F, untere/obere Grenze u, o,
Ausgabe: Positionen l, r der Zerlegung
p := F[(u+o)/2];
l := u; r := o;
while l <= r do
  l := Index des ersten Elementes
    aus l...o mit F[l] >= p
  r := Index des ersten Elementes
    aus u...r mit F[r] <= p
  if l <= r then
    Tausche F[l] und F[r];
    l := l+1; r := r-1;
  fi
od
```

Ein Element wird als „Pivot“ bestimmt (gewöhnlich das Element in der Mitte, es ist aber nicht zwingend). Dann wird die Folge mittels Vertauschen von Elementen so in zwei Teile zerlegt, dass im linken Teil alle Elemente \leq Pivot und im rechten Teil alle \geq Pivot sind.

Anschliessend werden beide Teile rekursiv weitergeteilt bis zur Folgenlänge 1, womit dann die Folge vollständig sortiert ist.

Das Teilungsverfahren kann unterschiedlich realisiert werden, eine Variante ist der obige *Zerlege* – Algorithmus.

Umsetzung:

```
class QuickSort<E extends Comparable<E>> implements Sort<E> {
    private void sort(E[] sammlung, int links, int rechts) {
        int auf = links; // linke Grenze
        int ab = rechts; // rechte Grenze
        final E ausgewaehlt = sammlung[(links + rechts) / 2];
        // ausgewähltes Element
        do {
            while (sammlung[auf].compareTo(ausgewaehlt) < 0)
                auf ++; // suchen größeres Element von links an
            while (ausgewaehlt.compareTo(sammlung[ab]) < 0)
                ab --; // suchen kleineres Element von rechts an
            if (auf <= ab) { // austauschen auf und ab:
                final E temp = sammlung[auf];
                sammlung[auf] = sammlung[ab];
                sammlung[ab] = temp;
                auf ++; // linke und rechte Grenze verschieben:
                ab --;
            };
        } while (auf <= ab); // Überschneidung
        if (links < ab)
            sort(sammlung, links, ab); // linke Hälfte sortieren
        if (auf < rechts)
            sort(sammlung, auf, rechts); // rechte Hälfte sortieren
    }

    public void sort(E[] sammlung) {
        sort(sammlung, 0, sammlung.length-1); // die ganze Reihung sortieren
    }
}
```

Eine verfeinerte QuickSort- Version:

```

Procedure  $qSort(a : \text{Array of Element}; \ell, r : \mathbb{N})$  // Sortiere das Teilarray  $a[\ell..r]$ .
  while  $r - \ell + 1 > n_0$  do // Benutze Teile-und-Herrsche.
     $j := \text{pickPivotPos}(a, \ell, r)$  // Wähle Pivotelement
     $\text{swap}(a[\ell], a[j])$  // und schaffe es an die erste Stelle.
     $p := a[\ell]$  //  $p$  ist jetzt das Pivotelement.
     $i := \ell; j := r$ 
    repeat //  $a: \boxed{\ell \quad i \rightarrow \leftarrow j \quad r}$ 
      while  $a[i] < p$  do  $i++$  // Überspringe Einträge,
      while  $a[j] > p$  do  $j--$  // die schon im richtigen Teilarray stehen.
      if  $i \leq j$  then // Wenn die Partitionierung noch nicht fertig ist,
         $\text{swap}(a[i], a[j]); i++; j--$  // (*) vertausche falsch positionierte Einträge.
      until  $i > j$  // Partitionierung ist fertig.
      if  $i < (\ell + r)/2$  then  $qSort(a, \ell, j); \ell := i$  // Rekursiver Aufruf für kleineres
      else  $qSort(a, i, r); r := j$  // der beiden Teilarrays.
    endwhile
   $\text{insertionSort}(a[\ell..r])$  // schneller für kleine  $r - \ell$ 

```

Abb. 5.7. Verfeinertes Quicksort für Arrays.

[illegible]

Abb. 5.8. Ausführung von *qSort* (Abb. 5.7) auf $\langle 3, 6, 8, 1, 0, 7, 2, 4, 5, 9 \rangle$, wobei stets der erste Eintrag als Pivot dient und $n_0 = 1$ gewählt wurde. Die *linke Seite* stellt den ersten Partitionierungsschritt dar. (Eben vertauschte Einträge sind grau hinterlegt.) Die *rechte Seite* zeigt die Ergebnisse aller Partitionierungsschritte, sowohl in den rekursiven Aufrufen als auch in der *while*-Schleife.

Optimierungen:

- (i) Pivot wird vor der Zerlegung zum linken Rand verschoben ==> Verkürzung der Iteration.
- (ii) Nach der Zerlegung wird nur die kürzere Teilfolge rekursiv weiter sortiert, die längere wird zuerst nochmal zerlegt ==> Minimierung von aufwendigen rekursiven Aufrufen.
- (iii) Kurze Teilfolgen (Länge $\leq n_0$) werden per InsertionSort sortiert, weil diese Methode für kurze Folgen schneller ist.

Analyse QuickSort:

- Zeitaufwand
 - Bester Fall → $n * \log n$
 - Durchschnittsfall (Ermittlung komplex) → $1.38 * n * \log n$
 - Schlechtester Fall (selten) → $n^2 * \log n$

Komplexitätsklasse im Durchschnittsfall: **$O(n * \log n)$**

QuickSelect

Bei bestimmten Aufgabenstellungen soll der **k**-te kleinste Eintrag ermittelt werden.

Beispiel: In einer Serie von **15** Messresultaten wird der Median als der achte kleinste Messwert bestimmt, die erste und dritte Quartile ergeben sich aus dem vierten resp. zwölften kleinsten Messwert.

Ermittlungen dieser Art sind trivial, wenn die untersuchte Sammlung vorausgehend sortiert wird, können aber auch mit einem geringeren Aufwand realisiert werden.

Zu diesem Zweck kann z.B. das QuickSelect – Verfahren benutzt werden, das durch eine Reduktion des QuickSort-Schema's erhalten wird:

//Finde den Eintrag mit Rang k

Function *select*(*s* : Sequence of Element; *k* : \mathbb{N}) : Element

assert $|s| \geq k$

wähle $p \in s$ rein zufällig

$a := \langle e \in s : e < p \rangle$

if $|a| \geq k$ then return *select*(*a*, *k*)

$b := \langle e \in s : e = p \rangle$

if $|a| + |b| \geq k$ then return *p*

$c := \langle e \in s : e > p \rangle$

return *select*(*c*, $k - |a| - |b|$)

// Pivoteintrag

k
//

<i>a</i>

k
//

<i>a</i>	<i>b</i> = $\langle p, \dots, p \rangle$
----------	--

k
//

<i>a</i>	<i>b</i>	<i>c</i>
----------	----------	----------

Abb. 5.9. Quickselect.

Anders als bei QuickSort werden die Iterationen nicht überlappend mit den gleichen Teilen der Folge durchgeführt, woraus sich eine Zeitkomplexität **O(n)** ergibt.

Mischen (MergeSort)

Dieses Vorgehen kann auch dann verwendet werden, wenn Arbeitsspeicher für die Datenmenge nicht ausreicht → Input wird von externen Medien bezogen.

Der Ansatz folgt dem folgenden Schema:

Zusammenfügen von zwei Folgen (Algorithmus)

```
procedure Merge ( $F_1, F_2$ ) →  $F$   
Eingabe: zwei zu sortierende Folgen  $F_1, F_2$   
Ausgabe: eine sortierte Folge  $F$   
  
   $F :=$  leere Folge;  
  while  $F_1$  oder  $F_2$  nicht leer do  
    Entferne das kleinere der Anfangselemente  
      aus  $F_1$  bzw.  $F_2$ ;  
    Füge dieses Element an  $F$  an  
  od;  
  Füge die verbliebene nichtleere Folge  $F_1$   
    oder  $F_2$  an  $F$  an;  
  return  $F$ 
```

Nach dem gleichen Prinzip können auch mehrere sortierte Folgen (z.B. Files) gemischt werden.

Die gesamte Datenmenge muss zuerst in Segmente zerlegt werden, die im Arbeitsspeicher sortiert werden können und die Resultate nachher gemischt wie oben beschrieben.

Beispiel:

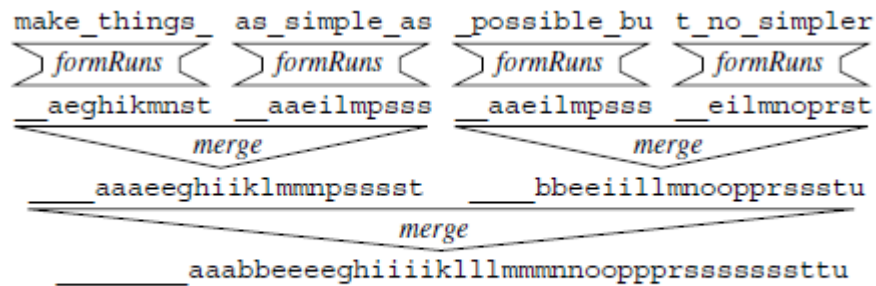
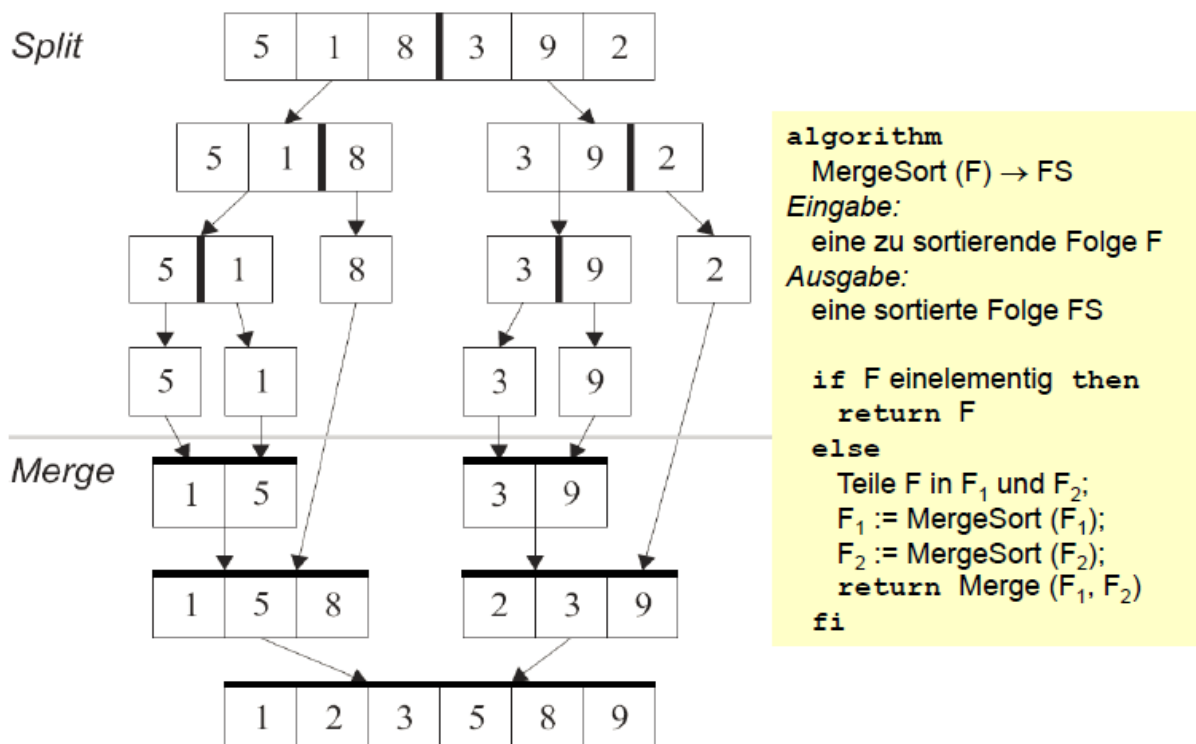


Abb. 5.14. Ein Beispiel für Zweiweg-Mergesort, ausgehend von Läufen der Länge 12.

Eine Variante von MergeSort kann auch für Datenfolgen im Arbeitsspeicher sehr effektiv eingesetzt werden.

Dabei wird die Folge rekursiv zweigeteilt bis zur Folgenlänge **1** und die Fragmente anschliessend durch Mischung zusammengefügt:

MergeSort: Beispiel



Analyse MergeSort (rekursive Variante):

- Zeitaufwand
 - Unempfindlich auf die Datenbeschaffenheit → $n * \log n$
- Speicherbedarf
 - Zusätzlicher Speicher notwendig für Resultate der Mischung

Komplexitätsklasse: $O(n * \log n)$

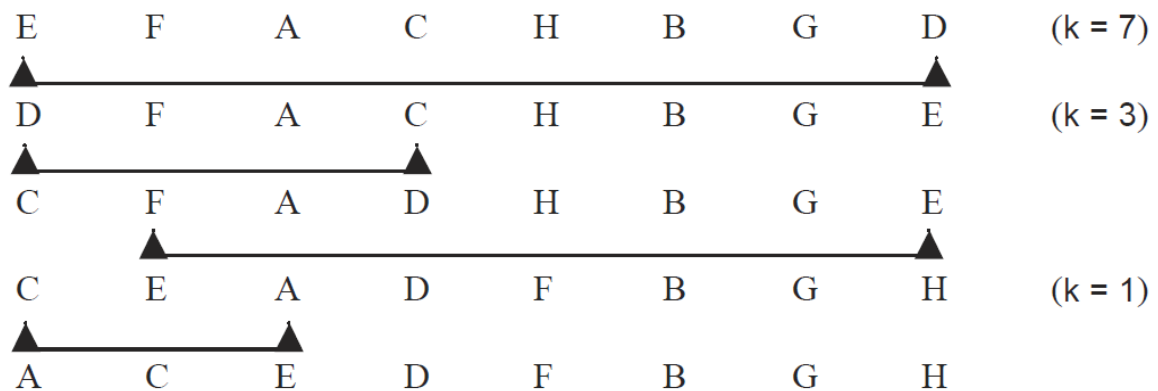
Sortierverfahren im Vergleich

Verfahren	Stabilität	Vergleiche (im Mittel)
SelectionSort	instabil	$\approx n^2 / 2$
InsertionSort	stabil	$\approx n^2 / 4$
BubbleSort	stabil	$\approx n^2 / 2$
MergeSort	stabil	$\approx n \log_2 n$
QuickSort	instabil	$\approx 1.38n \log_2 n$

ShellSort

Eine Variante von InsertionSort, bei der zuerst grössere „Sprünge“ durchs Vertauschen von Elementen ausgeführt werden, um das mehrfache schieben um nur eine Stelle zu ersetzen.

Die aufgebaute Teilfolge am linken Rand ist nicht vollständig sortiert → in jedem Durchlauf werden nur Elemente mit bestimmten Abständen (Schrittweiten) durchs Vertauschen geordnet.



... usw. mit dem einfachen Einfügen

Komplexitätsklasse: $O(n^{1.2})$

Radix-Verfahren

Alle bisher behandelten Sortierv Verfahren waren „vergleichsbasiert“: die Schlüssel-Instanzen konnten miteinander nur verglichen werden, hatten aber keine sonstige Eigenschaften, die beim Sortieren behilflich sein könnten. Man kann formal beweisen, dass mit dieser Einschränkung $O(n * \log n)$ die bestmögliche Zeitkomplexität ist, welche erreicht werden kann ($\rightarrow O(n * \log n)$ ist die *untere Schranke* für vergleichsbasierte Sortieralgorithmen).

RadixSort ist ein Beispiel des Sortierv Verfahrens, das eine bessere Zeitkomplexität erreicht, indem es numerische Schlüssel bekannter Länge m (d.h. aus m Ziffern bestehend) benutzt.

Es wird zuerst nach dem letzten Zeichen des Schlüssels sortiert, nachher nach dem vorletzten usw. („kleine Ordnung vor der grossen Ordnung“).

Beispiel:

Unsortierte Folge \rightarrow **170, 45, 75, 90, 802, 2, 24, 66**

Zuerst werden Schlüssel ergänzt zu einer einheitlichen Länge von drei Zeichen

\Rightarrow **170, 045, 075, 090, 802, 002, 024, 066**

Sortiert nach der letzten Ziffer

\Rightarrow **170, 090, 802, 002, 024, 045, 075, 066**

(das Paar **802, 002** behält dabei seine vorherige Reihenfolge, ebenfalls die Paare **170, 090** und **045, 075**).

Sortiert nach der mittleren Ziffer

\Rightarrow **802, 002, 024, 045, 066, 170, 075, 090**

Sortiert nach der ersten Ziffer

\Rightarrow **002, 024, 045, 066, 075, 090, 170, 802**

Komplexitätsklasse: $O(m * n)$

Bäume

Bäume → Systeme von Knoten, die mit Kanten verbunden sind.

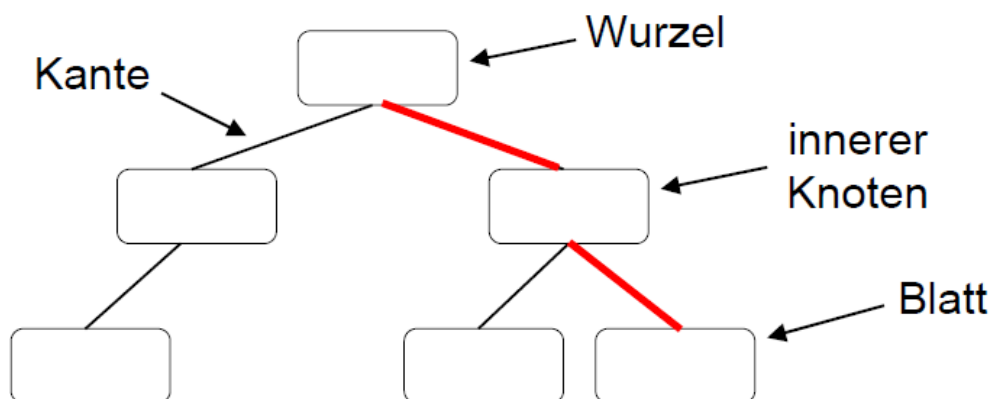
Dabei stellt jede Kante eine Vater ↔ Kind – Beziehung dar.

Ein Vaterknoten kann mehrere Kinderknoten besitzen, ein Kinderknoten gehört höchstens einem Vaterknoten.

Allgemein bekannte Beispiele von Bäumen:

- Klassendiagramme
- Systematik von Pflanzen und Tieren in der Biologie
- Dateisysteme

Begriffe im Baum

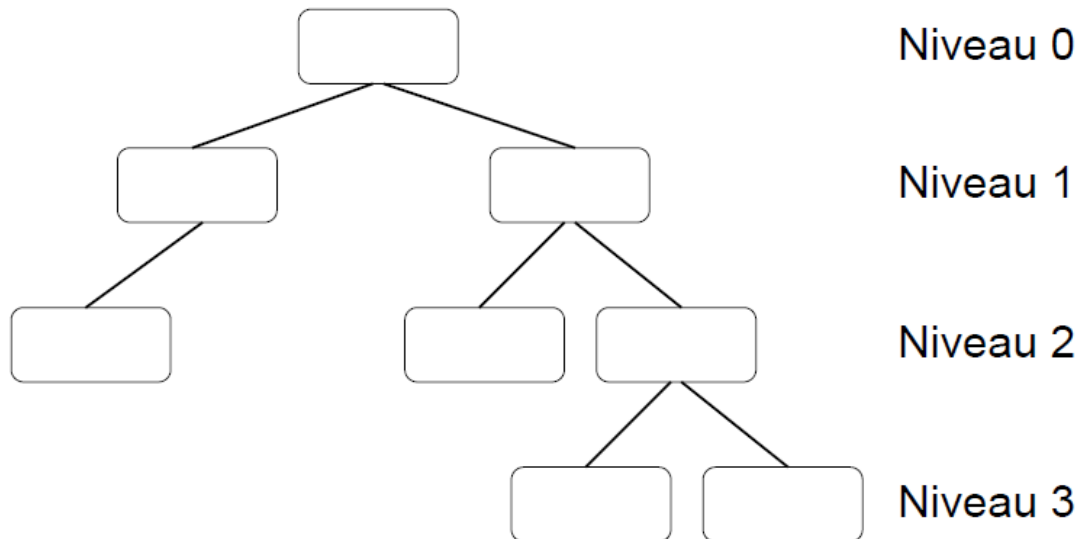


■ **Pfad:** Folgen von durch Kanten verbundene Knoten

■ zu jedem Knoten existiert genau ein Pfad von der Wurzel

Niveau's (Ebenen, Level's), Höhe und Gewicht eines Baumes:

Niveau und Höhe



■ $\text{Höhe} := \text{größtes Niveau} + 1$

Gewicht → Anzahl von Knoten

Binäre Bäume (BB)

⇒ eine besonders wichtige Kategorie von Bäumen.

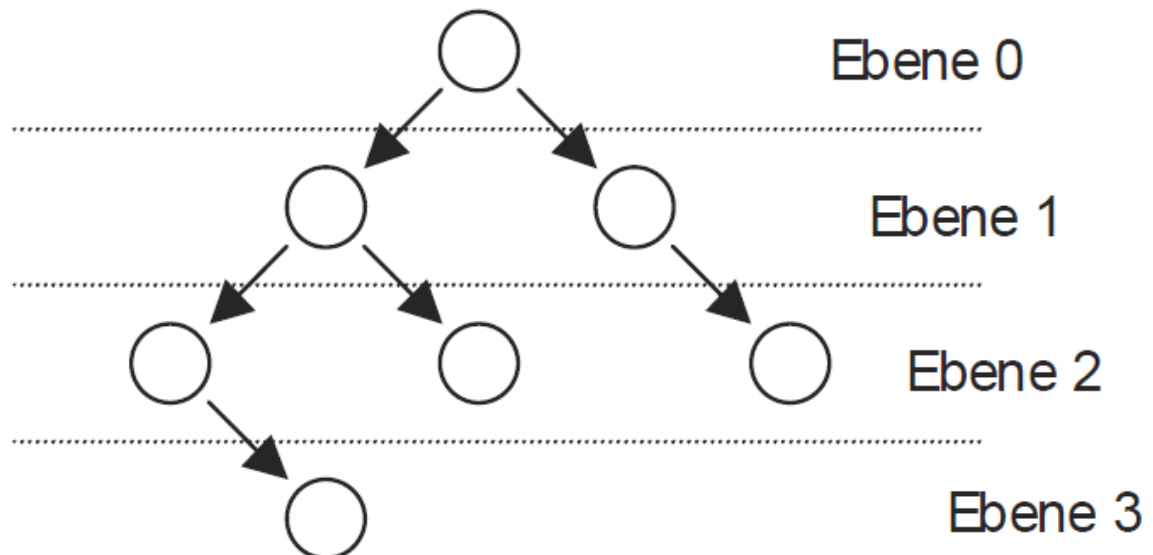
In einem binären Baum hat jeder Knoten maximal zwei Kinder.

Für binäre Bäume können die folgenden Eigenschaften definiert werden:

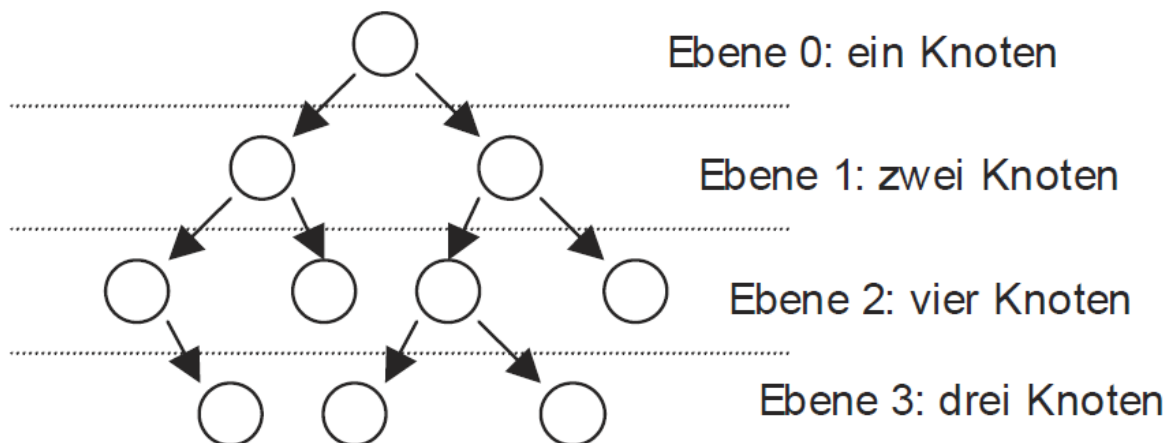
- BB ist **voll** → jede Ebene ausser der untersten ist voll, d.h. enthält 2^k Knoten ($k = 0$ -basierte Ebene)
- BB ist **komplett** → ist voll und die unterste Ebene ist linksbündig und dicht
- BB ist **sortiert** → jeder Knoten besitzt einen Schlüssel, wobei im linken Unterbaum sämtliche Schlüssel \leq Knotenschlüssel und im rechten Unterbaum sämtliche Schlüssel \geq Knotenschlüssel sind
- BB ist **streng sortiert** → jeder Knoten besitzt einen Schlüssel, wobei im linken Unterbaum sämtliche Schlüssel $<$ Knotenschlüssel und im rechten Unterbaum sämtliche Schlüssel \geq Knotenschlüssel sind

Beispiele von binären Bäumen

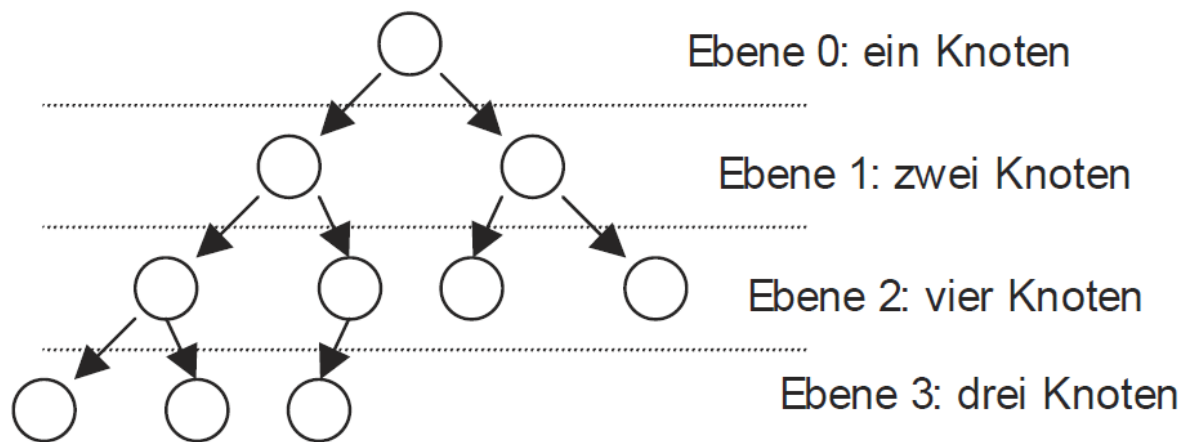
Baum nicht voll:



Baum voll, nicht komplett:



Baum komplett:



Die Java-Darstellung von binären Bäumen ist ähnlich wie bei verketteten Listen.

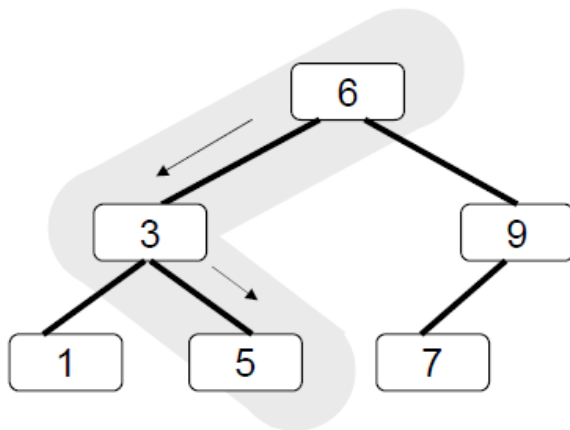
Ein Knoten wird durch die innere Klasse *Knoten* dargestellt, die eine Variable *inhalt* und zwei Referenzen auf Unterbäume besitzt.

Die Klasse *Binärbaum* besitzt die Variable *wurzel*.

```
class Binaerbaum<E extends Comparable<E>> {  
    protected static class Knoten<E extends Comparable<E>> {  
        private E inhalt;  
        private Knoten<E> links, rechts;  
        public Knoten(E inhalt) { this.inhalt = inhalt; }  
    }  
    protected Knoten<E> wurzel;  
    ... // Zugriffsmethoden  
}
```

Suchen, Einfügen und Löschen in sortierten binären Bäumen

Die Suche nach einem bestimmten Schlüssel ist in einem sortierten BB sehr einfach:



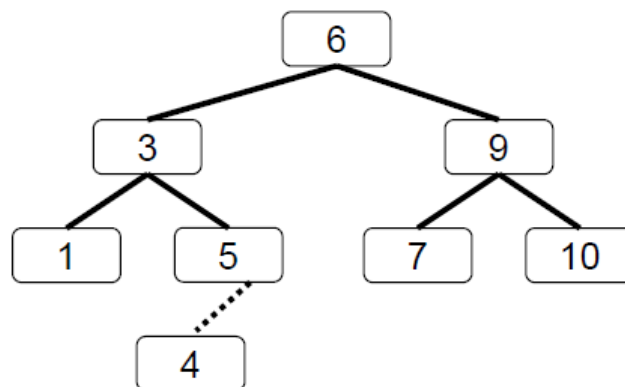
1. Vergleich des Suchschlüssels mit Schlüssel der Wurzel
2. wenn **kleiner**, dann in **linkem** Teilbaum weitersuchen
3. wenn **größer**, dann in **rechtem** Teilbaum weitersuchen
4. sonst ~> gefunden

Aufgrund der Eignung zum Suchen nennt man sortierte BB auch „Suchbäume“.

Einfügen → die Einfügen-Position kann mit dem gleichen Verfahren wie bei der obigen Suche ermittelt werden. Beim Endknoten des Suchpfades wird eine der zwei folgenden Alternativen auftreten:

Finden der Einfügeposition

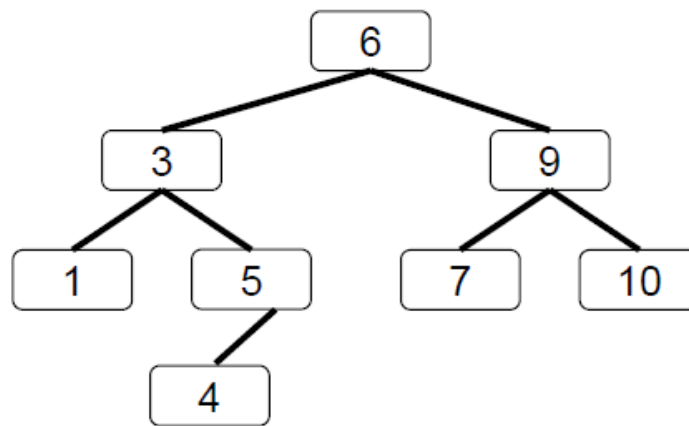
- Knoten, dessen Schlüsselwert **größer** als der einzufügende Schlüssel ist und der **keinen linken** Nachfolger hat
- Knoten, dessen Schlüsselwert **kleiner** als der einzufügende Schlüssel ist und der **keinen rechten** Nachfolger hat



Die so gefundene Position ist nicht die einzige Möglichkeit, **4** könnte auch rechts unter **3**, d.h. zwischen **3** und **5** eingehängt werden. Das oben beschriebene Vorgehen ist allerdings am einfachsten.

Verfahren beim Löschen:

- zu löschendes Element suchen: Knoten k
- drei Fälle
 1. k ist Blatt: Löschen
 2. k hat einen Sohn: Sohn hochziehen
 3. k hat zwei Söhne: Tausche mit **weitest links** stehendem Knoten des **rechten** Teilbaums
 - entferne diesen nach den Regeln 1. oder 2.



Der Zeitaufwand von den obigen Algorithmen hängt stark davon ab, wie ausgeglichen der Baum ist (diese Eigenschaft wird in der nächsten PVA thematisiert).

Heap (Halde)

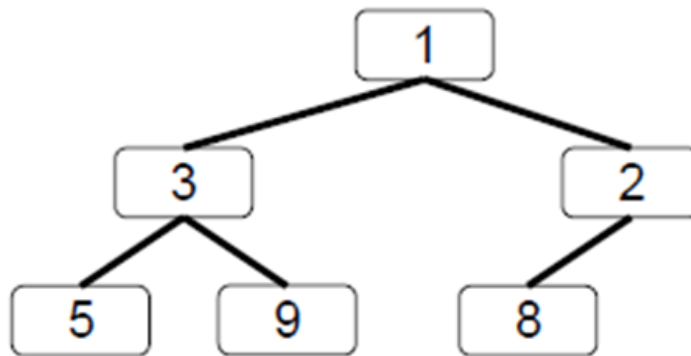
Eine andere sehr interessante Anwendung von Bäumen für Sortierzwecke.

Ein *Heap* (auch *Binärheap*) ist ein binärer Baum

- (i) der komplett aber nicht zwingend sortiert ist
- (ii) in dem der Schlüssel jedes Kindes grösser (oder gleich) ist als der Schlüssel des Vaters

Daraus ergibt sich → die Wurzel hat den kleinsten Schlüssel.

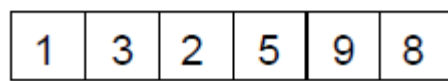
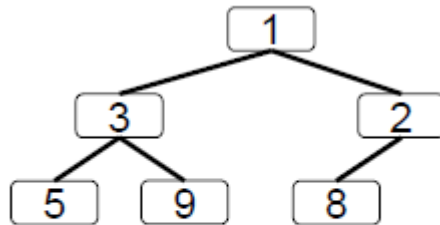
Beispiel:



Anmerkung: Es gibt auch eine alternative Heap-Definition, welche die umgekehrte Bedingung „Vater \geq Sohn“ festlegt, so dass die Wurzel den grössten Schlüssel besitzt.

Ein solches Heap bezeichnet man als *max-Prioritätswarteschlange* (*max-priority-queue*). Sämtliche diesbezügliche Prozeduren und Aussagen lassen sich von der erstenen Heap-Art ableiten, indem die Zeichen \geq und \leq ausgetauscht werden.

Heap kann auch als Feld (Reihung) dargestellt werden
(\rightarrow „verflacht in Levelorder“):



[1] [2] [3] [4] [5] [6] Position in der Reihung

Mit **1**-basierter Indexierung gilt: Vater $i \rightarrow$ Kinder $2i, 2i + 1$
Sohn $j \rightarrow$ Vater $j/2$

Wenn ein Heap als Feld dargestellt ist, sind die Verknüpfungen *Vater* \rightarrow *Sohn* in den Knoten überflüssig, weil sie sich aus den obigen Index-Beziehungen ergeben.

Anmerkung: Die gleiche Darstellungsart mit nacheinander folgenden Ebenen in einem Feld ist bei jedem kompletten Binärbaum anwendbar. Wenn die Beziehung „Vater \leq Sohn“ durchgehend erfüllt ist, bezeichnet man das Feld als *heapperecht*.

HeapSort

Diese Methode nutzt die Tatsache aus, dass die Wurzel eines Heaps immer den niedrigsten Schlüssel besitzt.

Daraus ergibt sich das folgende Vorgehen:

- die Wurzel herausnehmen und in die Zielfolge einreihen
- den Rest wieder zu einem Heap konsolidieren
- die neue Wurzel herausnehmen und in die Zielfolge einreihen
- den Rest wieder zu einem Heap konsolidieren
-

Die Zielfolge braucht dabei keinen zusätzlichen Speicher, wenn das Heap als Reihung (siehe oben) dargestellt wird → die Halde schrumpft vorne, wodurch hinten Platz für die sortierte Folge frei wird.

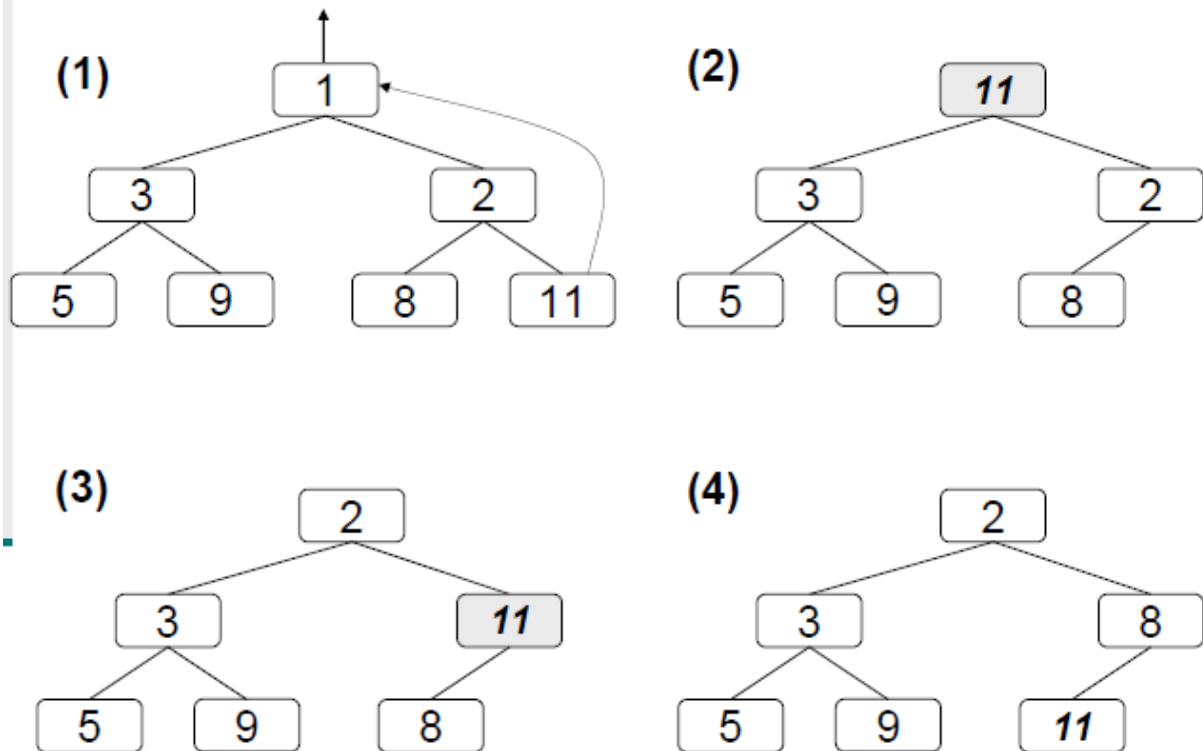
Das Vorgehen bei **Wurzel entfernen / Halde konsolidieren** in Detail:

Entfernen aus Heap

1. Entnimm die Wurzel
2. Schiebe „letztes“ Element des Feldes an die Wurzelposition
3. Lasse es „Durchsickern“ (eventuell bis auf Blattebene)
 - Sickern jeweils in Richtung des „kleineren“ Elementes um Heap-Eigenschaft zu erhalten

Beispiel:

Entfernen der Wurzel im Heap



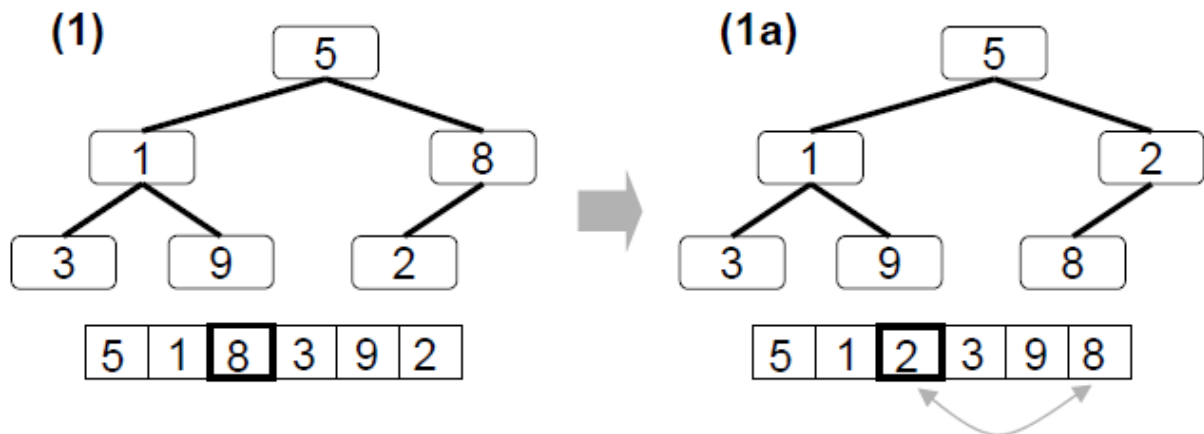
Vor einer Anwendung des obigen Vorgangs muss aus der unsortierten Datenmenge (Reihung) zuerst ein Heap aufgebaut werden.

Der Aufbau erfolgt in folgenden Schritten:

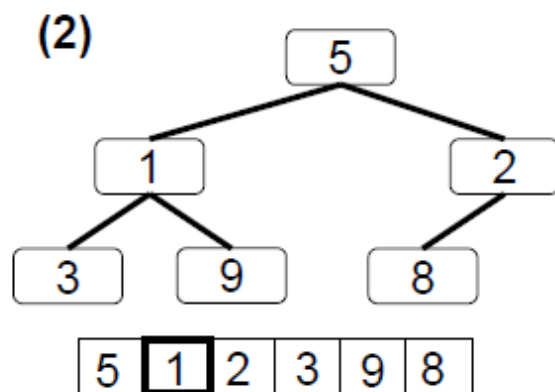
- Die Reihung wird aufgeteilt in Ebenen eines kompletten Baumes (siehe oben)
- Die Knoten werden von hinten durchgewandert und bei jedem die Schlüssel-Ungleichheit mit seinen Kindern (Vater $i \rightarrow$ Kinder $2i, 2i + 1$) geprüft (bei den Blättern in der untersten Ebene erübrigt sich das \rightarrow keine Kinder)
- Bei einer Verletzung der Haldenbedingung (Vater \leq Kind) wird Vater in Richtung des niedrigeren Schlüssels „versenkt“ (*siftDown*)

Beispiel:

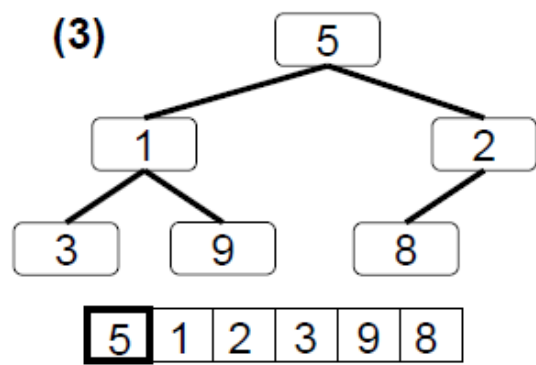
Aufbau eines Heaps



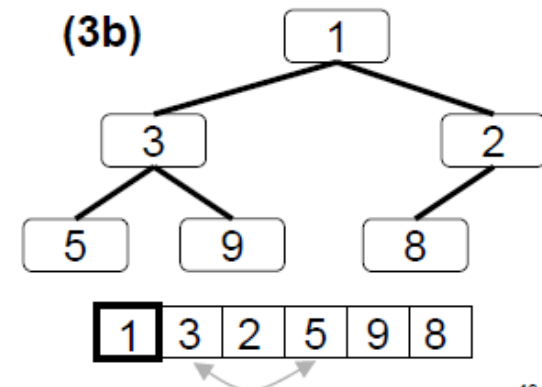
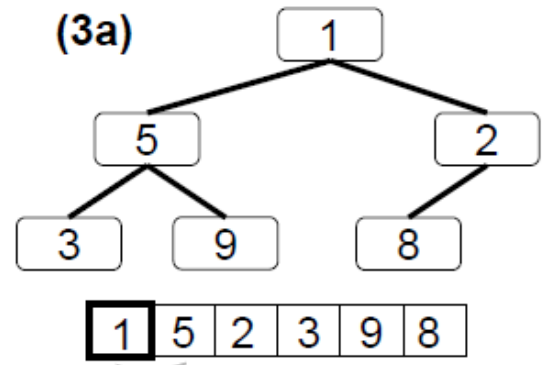
→ „8“ muss nach unten „sickern“



→ keine Änderung notwendig



→ „5“ muss nach unten „sickern“



Prioritätswarteschlangen

Heaps gehören zu einer allgemeineren Kategorie von Sammlungen – sog. *Prioritätswarteschlangen*. Ihre Glieder sind mit Prioritätsstufen versehen, die als Schlüssel verwendet werden → an der Spitze (Wurzel) der Warteschlange befindet sich jederzeit das Glied mit der höchsten Priorität (üblicherweise entspricht sie dem kleinsten Schlüsselwert), das in einer typischen Verarbeitung mit Vorrang von der Warteschlange entfernt und konsumiert wird. Somit ersetzt diese Prioritätsregel das FIFO-Prinzip bei gewöhnlichen Warteschlangen.

Man unterscheidet *nicht-adressierbare* und *adressierbare* Prioritätswarteschlangen.

Die nicht-adressierbare Variante verfügt über die folgende Schnittstelle:

- $Q.\text{build}(\{e_1, \dots, e_n\})$: $Q := \{e_1, \dots, e_n\}$.
- $Q.\text{insert}(e)$: $Q := Q \cup \{e\}$.
- $Q.\text{min}$: $\text{return min } Q$ (ein Eintrag mit minimalem Schlüssel).
- $Q.\text{deleteMin}$: $e := \text{min } Q$; $Q := Q \setminus \{e\}$; $\text{return } e$.

Somit können neue Glieder hinzugefügt und die Wurzel entfernt werden, sonstige Manipulationen der übrigen Glieder sind nicht vorgesehen. Die vorher beschriebenen Binärheap-Techniken eignen sich gut zur Implementierung dieser Variante.

In einer adressierbaren Prioritätswarteschlange besitzt jedes Glied eine Identifikation („Griff“ h), welche die zusätzliche Erweiterung der Schnittstelle ermöglicht:

- $Q.\text{insert}$: wie oben, aber es wird ein Griff für den neuen Eintrag zurückgegeben.
- $\text{remove}(h)$: entferne den durch den Griff h angegebenen Eintrag.
- $\text{decreaseKey}(h, k)$: verringere den Schlüssel des durch den Griff h angegebenen Eintrags auf k .
- $Q.\text{merge}(Q')$: $Q := Q \cup Q'$; $Q' := \emptyset$.

Diese Variante erlaubt dementsprechend Manipulationen sämtlicher Glieder der Sammlung. Auch diese Struktur könnte grundsätzlich mit einem Binärheap realisiert werden, eine effizientere Lösung ist jedoch mit einem System von getrennten Heaps (Wald) erreichbar, das mit der untenstehenden Klasse *AdressablePQ* per Pseudocode dargestellt wird (die Heaps sind nicht zwingend binär).

Class *Handle* = Pointer to *PQItem*

Class *AddressablePQ*

minPtr : *Handle* // Wurzel, die den minimalen Schlüssel enthält

roots : Set of *Handle* // Zeiger auf Wurzeln

Function *min* return Eintrag bei *minPtr*

Procedure *link*(*a, b* : *Handle*)

assert $a \leq b$

entnehme *b* aus *roots*

mache *a* zum Vorgänger von *b*

Procedure *combine*(*a, b* : *Handle*)

assert *a* und *b* sind Wurzeln

if $a \leq b$ then *link*(*a, b*) else *link*(*b, a*)

Procedure *newTree*(*h* : *Handle*)

roots := *roots* \cup {*h*}

if $*h < min$ then *minPtr* := *h*

Procedure *cut*(*h* : *Handle*)

hänge den Unterbaum mit Wurzel *h*

von seinem Vorgängerknoten ab

newTree(*h*)

Function *insert*(*e* : *Element*) : *Handle*

i := ein Griff für ein neues *PQItem*, das *e* enthält

newTree(*i*)

return *i*

Function *deleteMin* : *Element*

e := Eintrag bei *minPtr*

foreach Kind *h* der Wurzel bei *minPtr* do *cut*(*h*)

dispose *minPtr*

Rebalancierung; Aktualisierung von *minPtr*

return *e*

Procedure *decreaseKey*(*h* : *Handle*, *k* : *Key*)

assert $k \leq *h$

ändere den Schlüssel von *h* auf *k*

if *h* ist keine Wurzel then

cut(*h*); führe eventuell Aktionen zur Rebalancierung aus

Procedure *remove*(*h* : *Handle*) *decreaseKey*(*h*, $-\infty$); *deleteMin*

Procedure *merge*(*o* : *AddressablePQ*)

if $*minPtr > *(o.minPtr)$ then *minPtr* := *o.minPtr*

roots := *roots* \cup *o.roots*

o.roots := \emptyset ; eventuell Rebalancierung

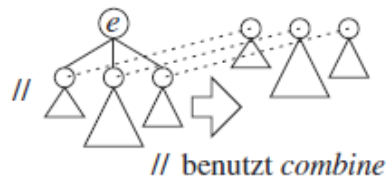
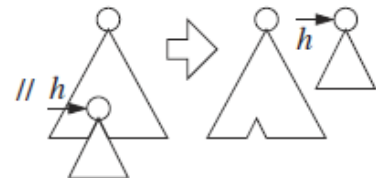
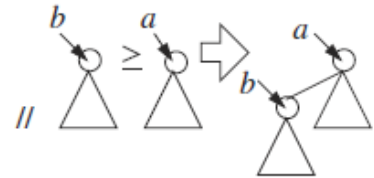
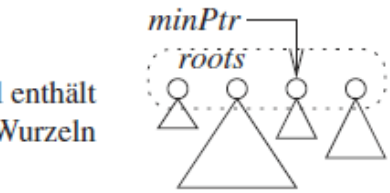


Abb. 6.4. Adressierbare Prioritätswarteschlangen

Für die Effizienz der Struktur sind ausschlaggebend die Prozeduren/Funktionen, welche nach einem ersten schnellen ($O(1)$ -) Schritt eine Rebalancierung des Waldes vornehmen müssen. Dieser zweite aufwendigere Schritt vereinigt hauptsächlich einzelne Heaps zu umfangreicheren Bäumen. Dieser Vorgang wird mit verschiedenen Ansätzen gehandhabt (siehe *Pairing Heaps* und *Fibonacci-Heaps* unten).

Pairing Heaps

Das Zusammenfügen von einzelnen Heap-Bäumen zu einem Heap erfolgt durch paarweises Vereinigen in zwei Pässen:

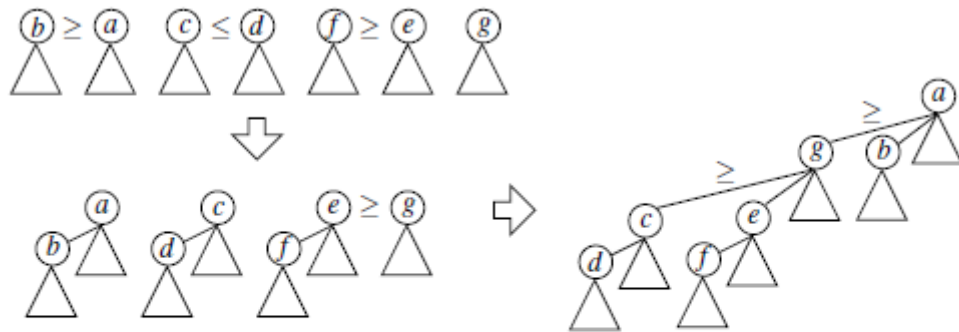
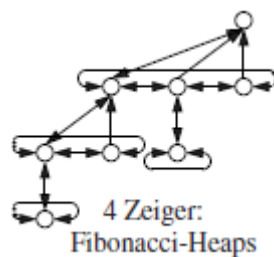


Abb. 6.5. Die Operation *deleteMin* für Pairing-Heaps fügt die durch Löschen der Wurzel entstehenden Bäume zuerst paarweise zusammen, um sie danach in einem Durchlauf von rechts nach links wieder zu einem Baum zusammenzubauen: Wurzel e wird Kind von g , dann wird Wurzel c Kind von g , dann wird Wurzel g Kind von a .

Die theoretische Zeitkomplexität dieser Rebalancierung ist nicht bekannt.

Fibonacci-Heaps

Jeder Knoten hat vier Zeiger, die auf einen Vorgänger, auf ein Kind und auf zwei Geschwister zeigen, wobei die Kinder eines Knotens als doppelt verkettete zirkuläre Liste organisiert sind:



Die Rebalancierung hat eine amortisiert logarithmische Zeitkomplexität.

Optimierungen von MergeSort

Beim MergeSort auf externen Medien ist die Effizienz stark davon abhängig, wie lange (im Arbeitsspeicher) sortierte Fragmente (*Sequenzen*) am Anfang für das Mischen zur Verfügung stehen.

Bei einer konventionellen Sortierung ist die Sequenzlänge durch die Speicherkapazität begrenzt (→ max. n Datensätze).

Es gibt allerdings Möglichkeiten, mit dem gleichen Speicher auch längere Sequenzen zu erzeugen.

Sortierkanal

Vorgehen:

- Mit den ersten n Datensätzen vom Input wird ein Heap erstellt
- Der nächste Datensatz D wird vom Input gelesen
 - $D > \text{Wurzel}$ → die Wurzel wird in die aufgebaute Sequenz verschoben und D in das Heap versenkt
 - $D \leq \text{Wurzel}$ aber $D \geq \text{der letzte Datensatz der Sequenz}$ oder die Sequenz ist noch leer ist → D wird direkt in die Sequenz verschoben
 - $D < \text{der letzte Datensatz der Sequenz}$ → die Wurzel wird in die Sequenz verschoben und D in ein zweites Heap integriert, welche für die nächste Sequenz reserviert ist und hinter dem aktuellen (schrumpfenden) Heap wächst
- Wenn das aktuelle Heap leer ist, wird die Sequenz abgeschlossen und das „Reserve-Heap“ dient anschliessend als aktuelles Heap für die nächste Sequenz

Dieses Verfahren liefert Sequenzen mit durchschnittlich $2n$ Datensätzen.

Mischkanal

Das obige Verfahren (*Sortierkanal*) wird wiederholt, wobei die aufgebaute Sequenz (Länge $2n$ in *Datei2*) mit Output des Sortierkanals laufend gemischt und das Ergebnis in *Datei3* geschrieben wird. Die in *Datei3* erhaltenes Resultat (Länge $4n$) wird anschliessend mit Sortierkanal-Output zu einer $6n$ langen Sequenz in *Datei2* gemischt usw.:

	Anfang	Nach 1. Durchlauf	Nach 2. Durchlauf	Nach 3. Durchlauf
Datei1 (Input)	N	$N - 3n$	$N - 5n$	$N - 7n$
Datei2	0	$2n$	0	$6n$
Datei3	0	0	$4n$	0
Reserve-Halde	0	n	n	n

Die erzeugte Sequenz wächst bei jedem Durchlauf um $2n$, nach dem letzten Durchlauf sind die Daten vollständig sortiert.

Vorteil: Während dem Mischvorgang auf externen Medien wird gleichzeitig auch die Arbeitsspeicher-Kapazität durchgehend genutzt.

Nachteil: Ist effektiv nur für weniger als ca. 5 Durchläufe (d.h. für $N < 10n$), weil die gleichen Daten immer wieder angefasst werden müssen (der Kopieraufwand wächst bei weiteren Durchläufen stark auf Kosten des Sortierens).

Mehrphasen-Mischen

Eine Technik, die auch für sehr grosse Datenmengen verhältnismässig effizient bleibt.

Im ersten Schritt werden mit dem obigen Verfahren (*Mischkanal*) Sequenzen erzeugt (typische Länge **4n**), auf eine Anzahl von Dateien verteilt (6 im folgenden Beispiel, wobei eine leer bleibt) und anschliessend in mehreren Durchläufen miteinander gemischt (Output wird in die jeweils leere Datei geschrieben):

<i>Durchlauf:</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>
1. Datei	16	8	4	2	1	0
2. Datei	15	7	3	1	0	1
3. Datei	14	6	2	0	1	0
4. Datei	12	4	0	2	1	0
5. Datei	8	0	4	2	1	0
6. Datei	0	8	4	2	1	0
Σ	65	33	17	9	5	1

Die Sequenzlänge wächst bei jedem Durchlauf, die Zahl der Sequenzen sinkt. Nach dem letzten Durchlauf sind die Daten vollständig sortiert.

Bei einer optimal gewählten Verteilung von Sequenzen auf die vorhandenen Dateien bildet die Abfolge von Sequenzzahlen nach einzelnen Durchläufen (letzte Zeile der obigen Tabelle) annähernd eine Fibonacci-Folge (→ bei diesem Ansatz wird die Methode auch als Fibonacci-Mischen bezeichnet).