

# Algorithmen

Patrick Gustav Blaneck

Letzte Änderung: 4. November 2021

## Inhaltsverzeichnis

<b>1</b>	<b>Grundbegriffe</b>	<b>2</b>
<b>2</b>	<b>Elementare Datenstrukturen</b>	<b>4</b>
2.1	Abstrakte Datentypen . . . . .	5
2.2	Datenstrukturen . . . . .	6
2.3	Hashing . . . . .	9
<b>3</b>	<b>Bäume</b>	<b>13</b>
3.1	Binäre Suchbäume . . . . .	14
3.2	AVL-Bäume . . . . .	18
3.3	B-Bäume . . . . .	22
3.4	Rot-Schwarz-Bäume . . . . .	26
3.5	Heaps . . . . .	28
<b>4</b>	<b>Graphen</b>	<b>32</b>
4.1	Suche . . . . .	36
4.2	Entwurfsprinzipien . . . . .	39
4.3	Graphalgorithmen . . . . .	40
<b>5</b>	<b>Formale Sprachen</b>	<b>48</b>
5.1	Textsuche . . . . .	48
5.2	Reguläre Ausdrücke . . . . .	52
<b>6</b>	<b>Sortierverfahren</b>	<b>58</b>
6.1	Elementare Sortierverfahren . . . . .	59
6.2	Höhere Sortierverfahren . . . . .	63
6.3	Spezialisierte Sortierverfahren . . . . .	67
	<b>Index</b>	<b>69</b>
	<b>Beispiele</b>	<b>71</b>

# 1 Grundbegriffe

## Definition: Eigenschaften eines Algorithmus

- *Terminierung*: Der Algorithmus bricht nach *endlichen vielen* Schritten ab.
- *Determiniertheit*: Bei vorgegebener Eingabe wird ein eindeutiges *Ergebnis* geliefert.
- *Determinismus*: Eindeutige Vorgabe der *Abfolge* der auszuführenden Schritte

## Definition: Landau-Notation

Seien  $f, g$  reellwertige Funktionen der reellen Zahlen. Dann gilt: [1]

Notation	Definition	Mathematische Definition
$f \in \mathcal{O}(g)$	obere Schranke	$\exists C > 0 \exists x_0 > 0 \forall x > x_0 :  f(x)  \leq C \cdot  g(x) $
$f \in \Omega(g)$	untere Schranke	$\exists c > 0 \exists x_0 > 0 \forall x > x_0 : c \cdot  g(x)  \leq  f(x) $
$f \in \Theta(g)$	scharfe Schranke	$\exists c > 0 \exists C > 0 \exists x_0 > 0 \forall x > x_0 : c \cdot  g(x)  \leq  f(x)  \leq C \cdot  g(x) $

Anschaulicher gilt:

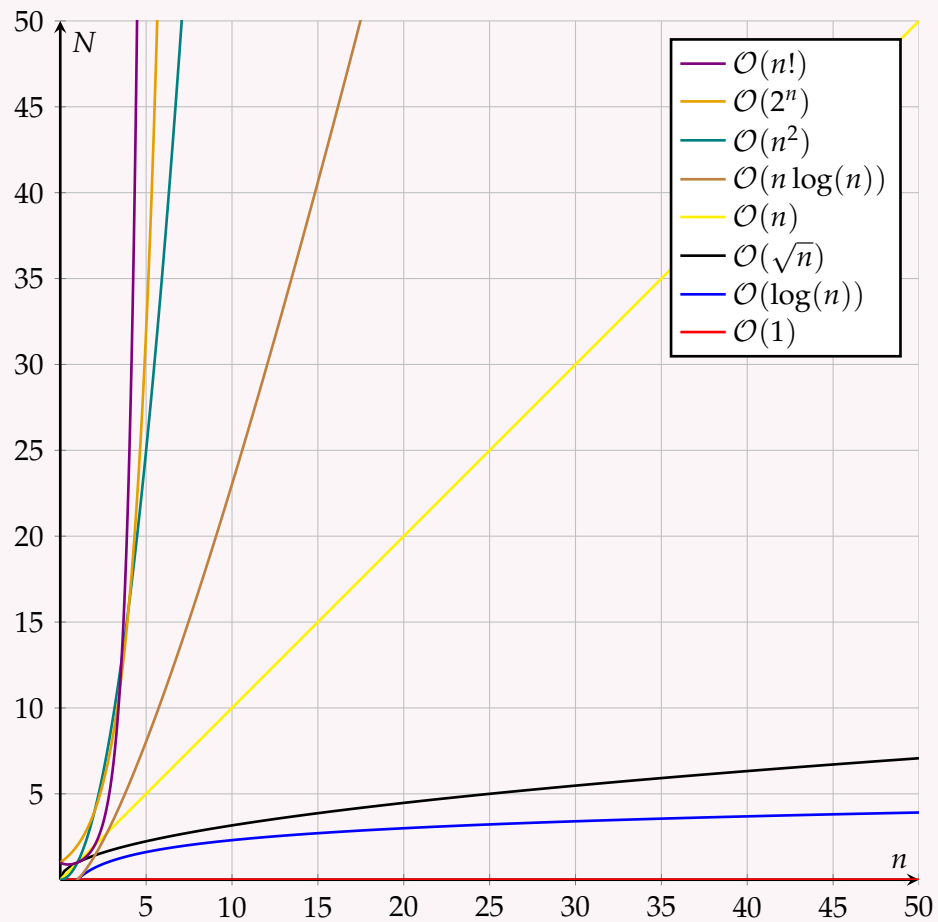
Notation	Anschauliche Bedeutung
$f \in \mathcal{O}(g)$	$f$ wächst nicht wesentlich schneller als $g$
$f \in \Omega(g)$	$f$ wächst nicht wesentlich langsamer als $g$
$f \in \Theta(g)$	$f$ wächst genauso schnell wie $g$

## Beispiel: Landau-Notation

Aus [1] :

Notation	Beispiel
$f \in \mathcal{O}(1)$	Feststellen, ob eine Binärzahl gerade ist
$f \in \mathcal{O}(\log n)$	Binäre Suche im sortierten Feld mit $n$ Einträgen
$f \in \mathcal{O}(\sqrt{n})$	Anzahl der Divisionen des naiven Primzahltests
$f \in \mathcal{O}(n)$	Suche im unsortierten Feld mit $n$ Einträgen
$f \in \mathcal{O}(n \log n)$	Mergesort, Heapsort
$f \in \mathcal{O}(n^2)$	Selectionsort
$f \in \mathcal{O}(n^m)$	
$f \in \mathcal{O}(2^{cn})$	(Backtracking)
$f \in \mathcal{O}(n!)$	Traveling Salesman Problem

### Bonus: Visualisierung Komplexitätsklassen



## 2 Elementare Datenstrukturen

### Definition: Abstrakte Datentypen (ADTs)

Anforderungen an die Definition eines Datentyps:

- *Spezifikation* eines Datentyps unabhängig von der Implementierung
- Reduzierung der von außen sichtbaren Aspekte auf die *Schnittstelle* des Datentyps

Daraus entstehen **zwei Prinzipien**:

- *Kapselung*:  
Zu einem ADT gehört eine Schnittstelle.  
Zugriffe auf den ADT erfolgen ausschließlich über die Schnittstelle.
- *Geheimnisprinzip*:  
Interne Realisierung eines ADT-Moduls bleibt verborgen.

### Definition: Homogene Datenstruktur

In einer *homogenen Datenstruktur* haben alle Komponenten den *gleichen* Datentyp.

### Definition: Heterogene Datenstruktur

In einer *heterogenen Datenstruktur* haben die Komponenten *unterschiedliche* Datentypen.

### Bonus: ADTs in Java

Viele wichtige abstrakte Datentypen werden in Java durch *Interfaces* beschrieben.

Es gibt ein oder mehrere Implementierungen dieser Interfaces mit unterschiedlichen dahinter stehenden Konzepten.

In Java: Package `java.util`

Wichtig in der Vorlesung:

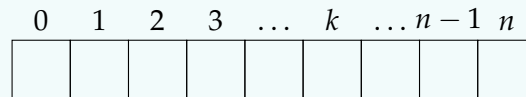
ADT	Grund-ADT/Interface	Java-Klassen
Feld		(Felder), HashMap
Liste	List	ArrayList, LinkedList
Menge	Set	HashSet, TreeSet
Prioritätswarteschlange		PriorityQueue
Stack	List	
Queue	List	
Deque	List	Deque (Interface), ArrayDeque
Map	Set	Map (Interface), HashMap, TreeMap
BidiMap	Map	BidiMap, BiMap (Interface)
MultiSet, Bag	Map	Bag, Multiset (Interface)

## 2.1 Abstrakte Datentypen

### Definition: Array

Ein *Array* hat folgende spezielle Eigenschaften:

- Feste Anzahl an Datenobjekten
- Auf jedes Objekt kann direkt lesend oder schreibend zugegriffen werden



### Performance:

Zugriff	Suche	Einf./Lösch. (Anfang)	Einf./Lösch. (Ende)	Einf./Lösch. (Mitte)
$\Theta(1)$	$\Theta(n)$	-	-	-

### Definition: Liste

Eine Liste besteht aus Elementen, die wie in einem Array linear angeordnet sind. Auf die Elemente einer Liste muss nicht wahlfrei zugegriffen werden können<sup>a</sup>.

Die Größe einer Liste ist nicht von Anfang an bekannt und sie kann beliebig verlängert bzw. verkürzt werden.

<sup>a</sup>Eine Ausnahme stellt der Listenanfang dar.

### Definition: Stack

- Daten können an einem Ende hinzugefügt oder entnommen werden.



### Definition: Queue

- Daten können an einem Ende hinzugefügt und am anderen Ende entnommen werden.



### Definition: Deque („Double ended queue“)

- Daten können an beiden Enden hinzugefügt und entnommen werden.



### Bonus: Prioritätswarteschlange

Eine *Prioritätswarteschlange* ist eine Warteschlange, deren Elemente einen Schlüssel (*Priorität*) besitzen.

### Implementierung:

In Java dient zur Implementierung die Klasse `PriorityQueue`, alternativ auch `TreeSet`.

### Definition: Menge

Eine *Menge* (*Set*) ist eine Sammlung von Elementen des gleichen Datentyps. Innerhalb der Menge sind die Elemente ungeordnet. Jedes Element kann nur einmal in der Menge vorkommen.

#### Implementierung:

In Java ist *Set* ein Interface, das unter anderem folgende Klassen implementiert:

- **TreeSet:** Basiert auf der Datenstruktur Rot-Schwarz-Baum, implementiert Erweiterung `SortedMap`.
- **HashSet:** Basiert auf der Datenstruktur Hashtabelle.

### Definition: Assoziatives Feld

Ein *assoziatives Feld* ist eine Sonderform des Feldes:

- Verwendet keinen numerischen Index zur Adressierung eines Elements.
- Verwendet zur Adressierung einen Schlüssel (z.B. `a["Meier"]`).

Assoziative Felder eignen sich dazu, Datenelemente in einer großen Datenmenge aufzufinden. Jedes Datenelement wird durch einen *eindeutigen Schlüssel* identifiziert.

#### Implementierung:

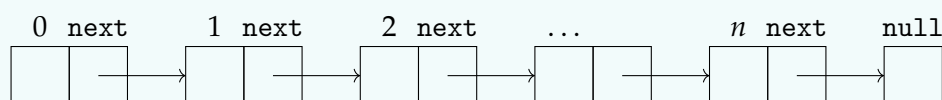
In Java entspricht ein *assoziatives Feld* dem Interface `java.util.Map`, das folgende Klassen implementiert:

- **TreeMap:** Basiert auf der Datenstruktur Rot-Schwarz-Baum, implementiert Erweiterung `SortedMap`.
- **HashMap:** Basiert auf der Datenstruktur Hashtabelle.

## 2.2 Datenstrukturen

### Definition: Einfach verkettete Liste

Im Vergleich zu einem Array kann eine *Liste* schrumpfen und wachsen.



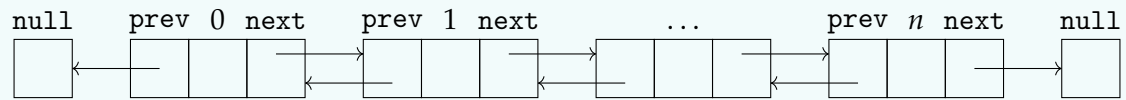
#### Performance:

Zugriff	Suche	Einf./Lösch. (Anfang)	Einf./Lösch. (Ende)	Einf./Lösch. (Mitte)
$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)/\Theta(n)^a$	Suchzeit + $\Theta(1)$

<sup>a</sup> $\Theta(1)$ , wenn das letzte Element bekannt ist,  $\Theta(n)$  sonst

### Definition: Doppelt verkettete Liste

Im Vergleich zu einer einfach verketteten Liste besitzt die *doppelt verkettete Liste* zusätzlich einen Verweis auf den Vorgänger.



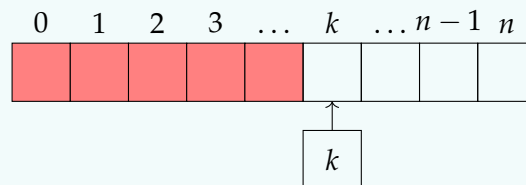
#### Performance:

Zugriff	Suche	Einf./Lösch. (Anfang)	Einf./Lösch. (Ende)	Einf./Lösch. (Mitte)
$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	Suchzeit + $\Theta(1)$

### Definition: Dynamisches Feld

Ein *dynamisches Feld* besteht aus:

- Einem normalen Feld, das nicht vollständig gefüllt ist.
- Einem Zeiger, der anzeigt, welches das erste unbesetzte Element ist.



#### Performance:

Zugriff	Suche	Einf./Lösch. (Anfang)	Einf./Lösch. (Ende)	Einf./Lösch. (Mitte)
$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)/\Theta(n)^a$	$\Theta(n)$

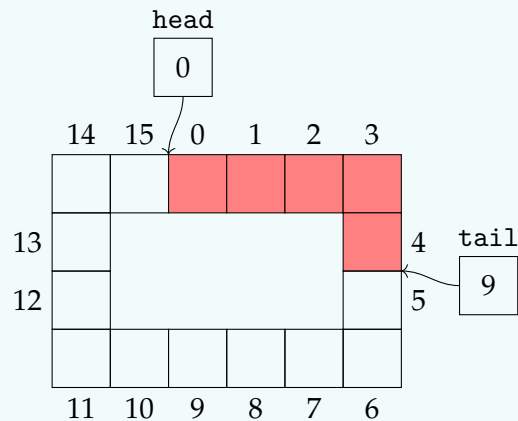
Damit ist ein dynamisches Feld gut für einen *Stack* geeignet!

<sup>a</sup>Wenn das Feld schon voll ist, muss der komplette Inhalt kopiert werden.

### Definition: Zirkuläres (dynamisches) Feld

Ein *zirkuläres Feld* besitzt einen Speicher fester Größe. Dabei speichern zwei Zeiger jeweils den Anfang (*head*) des Speichers, bzw. auf die nächste freie Speicheradresse (*tail*) im Speicher.

Wird ein Element am Anfang „abgearbeitet“, bewegt sich *head* eine Position weiter. Wird ein Element am Ende eingefügt, bewegt sich *tail* eine Position weiter.



**Performance:** (dynamisch, bei unterliegender Datenstruktur Array)

Zugriff	Suche	Einf./Lösch. (Anfang)	Einf./Lösch. (Ende)	Einf./Lösch. (Mitte)
$\Theta(1)$	$\Theta(n)$	$\Theta(1)/\Theta(n)^a$	$\Theta(1)/\Theta(n)^b$	$\Theta(n)$

Damit ist ein zirkuläres (dynamisches) Feld gut für eine *Queue/Deque* geeignet!

<sup>a</sup>Wenn das Feld schon voll ist, muss der komplette Inhalt kopiert werden.

<sup>b</sup>Siehe Fußnote a.

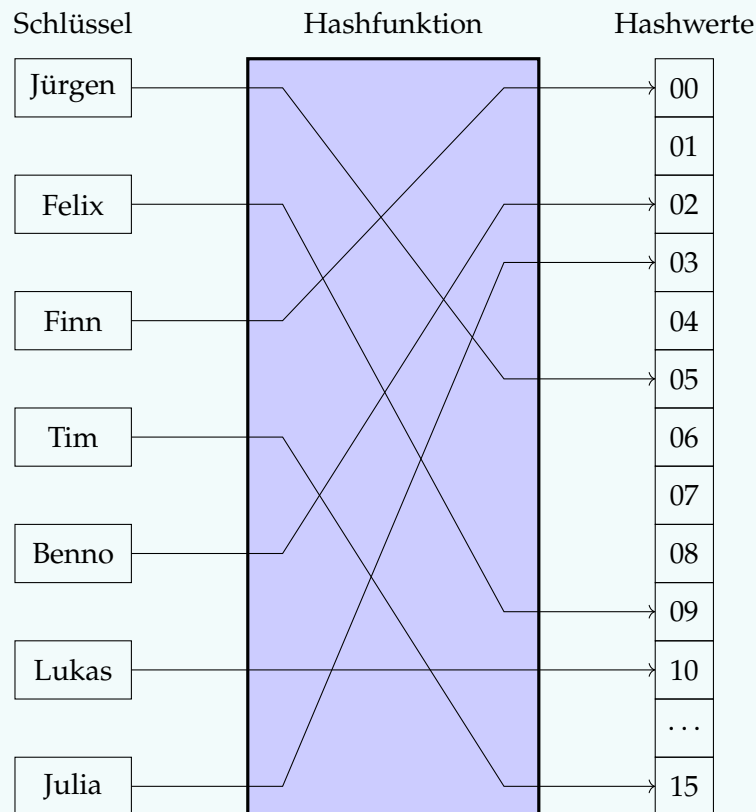


## 2.3 Hashing

### Definition: Hashfunktion

Eine Hashfunktion oder Streuwertfunktion ist eine Abbildung  $h : S \rightarrow I$ , die eine große Eingabemenge, die Schlüssel  $S$ , auf eine kleinere Zielmenge, die Hashwerte  $I$ , abbildet.<sup>a</sup>

Die Bildmenge  $h(S) \subseteq I$  bezeichnet die Menge der *Hash-Indizes*.



<sup>a</sup>Eine Hashfunktion ist daher im Allgemeinen nicht injektiv, aber surjektiv.

### Beispiel: Divisions-Hash

Die *Divisionsrest-Methode* (*Divisions-Hash*) um Integer zu hashen wird definiert durch:

$$h(x) = x \bmod N$$

Sie wird bevorzugt, wenn die Schlüsselverteilung nicht bekannt ist. Etwaige Regelmäßigkeiten in der Schlüsselverteilung sollte sich nicht in der Adressverteilung auswirken. Daher sollte  $N$  eine Primzahl sein.

### Beispiel: Hashfunktionen für verschiedene Datentypen

- Alle Datentypen: Verwenden der Speicheradresse
- Strings: ASCII/Unicode-Werte addieren (evtl. von einigen Buchstaben, evtl. gewichtet)

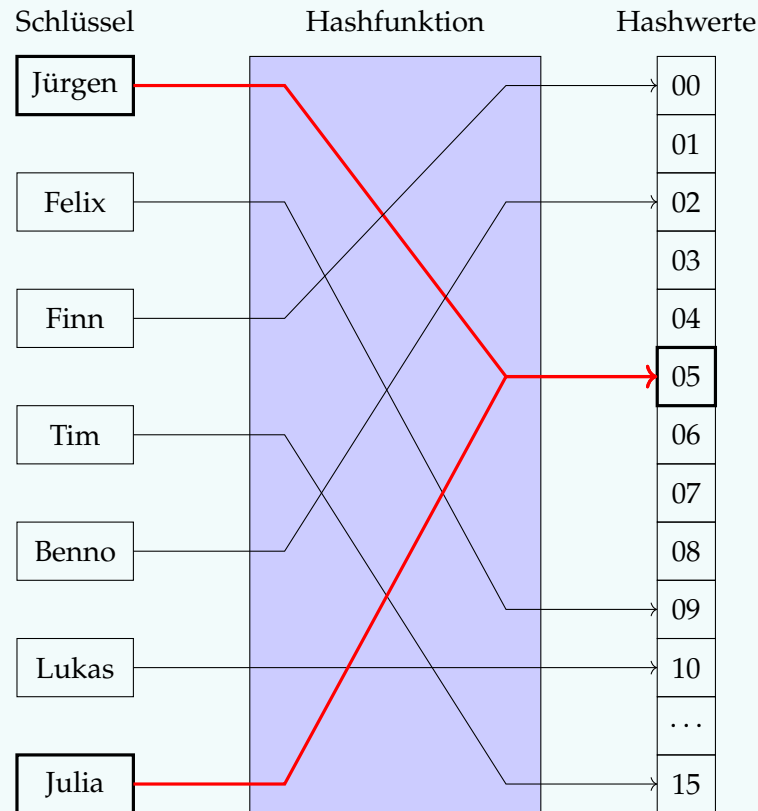
### Definition: Kollision

Sei  $S$  eine Schlüsselmenge und  $h$  eine Hashfunktion. Ist

$$s_1, s_2 \in S, s_1 \neq s_2 : h(s_1) = h(s_2)$$

so spricht man von einer *Kollision*.

Die Wahrscheinlichkeit von Kollisionen ist abhängig von der gewählten Hashfunktion. Hashfunktionen sollten also möglichst gut *streuen*, aber dennoch effizient berechenbar sein.



### Definition: Kollisionsbehandlung

Um Kollisionen zu handhaben, existieren verschiedene Strategien:

- *Hashing mit Verkettung*
  - Hashtabelle besteht aus  $N$  linearen Listen
  - $h(s)$  gibt dann an, in welche Teilliste der Datensatz gehört
  - Daten werden innerhalb der Teillisten sequentiell gespeichert
- *Hashing mit offener Adressierung*
  - Suchen einer alternativen Position innerhalb des Feldes
    1. Lineares Sondieren (Verschiebung um konstantes Intervall)
    2. Doppeltes Hashing (Nutzen einer weiteren Hashfunktion)
    3. Quadratisches Sondieren (Intervall wird quadriert)

#### Definition: Schrittzahl

Die *Schrittzahl*  $S(s)$ , die nötig ist, um den Datensatz mit Schlüssel  $s$  zu speichern bzw. wiederzufinden, setzt sich z.B. beim Hashing mit Verkettung zusammen aus:

- der Berechnung der Hash-Funktion und
- dem Aufwand für die Suche bzw. Speicherung innerhalb der Teilliste.

#### Definition: Füllgrad

Der *Füllgrad* einer Hashtabelle ist der Quotient

$$\alpha = \frac{n}{N}$$

mit

- $N$  als Größe der Hashtabelle
- $n$  als Anzahl der gespeicherten Datensätze

#### Beispiel: Schrittzahl beim Suchen in Teillisten

Bei idealer Speicherung entfallen  $\alpha$  Elemente auf jede Teilliste. Dabei gilt:

- erfolgreiche Suche:  $c_1 + c_2 \cdot \frac{\alpha}{2}$
- erfolglose Suche:  $c_1 + c_2 \cdot \alpha$

Damit ist der Suchaufwand in  $\mathcal{O}(\alpha) = \mathcal{O}\left(\frac{n}{N}\right)$ .

Wird der Füllgrad  $\alpha$  zu groß, sollte die Hashtabelle vergrößert werden.

#### Definition: Dynamisches Hashing

Um viele Kollisionen zu vermeiden, muss die Hashtabelle ab einem gewissen Füllgrad vergrößert werden.<sup>a</sup>

Als Folge muss die gesamte Hashtabelle aber auch neu aufgebaut werden.

---

<sup>a</sup>nach Sedgewick sollte stets  $\alpha < 0.5$  gelten

### Definition: Offene Adressierung (Sondieren)

Beim Speichern wird bei *Hashing mit offener Adressierung (Sondierung)* so lang ein neuer Hashindex berechnet, bis dort ein freier Speicherplatz vorhanden ist.

Das Suchen funktioniert analog, allerdings ist das Löschen sehr aufwändig.

- Lineares Sondieren
  - Wird die Ersatzadresse bei jeder Kollision durch Erhöhen der alten Adresse um 1 berechnet, so spricht man von *linearem Sondieren (linear probing)*.
  - Die  $i$ -te Ersatzadresse für einen Schlüssel  $s$  mit Hashindex  $h(s)$  wird also wie folgt berechnet:

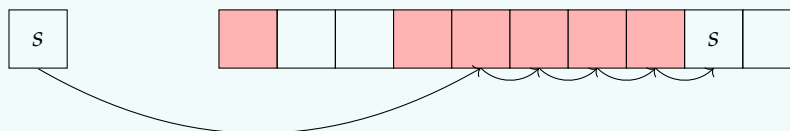
$$h_i(s) = (h(s) + i) \bmod N$$

- Doppeltes Hashing
  - Schlüssel wird nicht um 1 erhöht, sondern der Inkrement wird mit einer zweiten Hashfunktion berechnet.
  - Beseitigt praktisch die Probleme der primären und sekundären Häufung.
  - Nicht alle Felder werden durchprobiert. Im ungünstigsten Fall kann ein neues Element nicht eingefügt werden, auch wenn noch Felder frei sind.

### Definition: Primäre und sekundäre Häufung

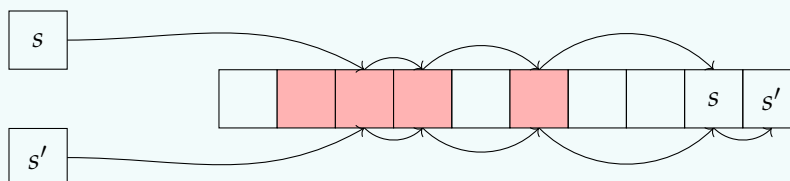
Bei der *primären Häufung (primary clustering)* ist die Wahrscheinlichkeit, dass Plätze in einem dichtbelegten Bereich eher besetzt werden, deutlich höher. Es kommt also zu Kettenbildung.

Besonders häufig tritt primäre Häufung z.B. beim linearen Sondieren auf.



Die *sekundäre Häufung (secondary clustering)* hängt von der Hashfunktion ab. Dabei durchlaufen zwei Schlüssel  $h(s)$  und  $h(s')$  stets dieselbe Sondierungsfolge. Sie behindern sich also auf den Ausweichplätzen.

Besonders häufig tritt sekundäre Häufung z.B. beim quadratischen Sondieren auf.

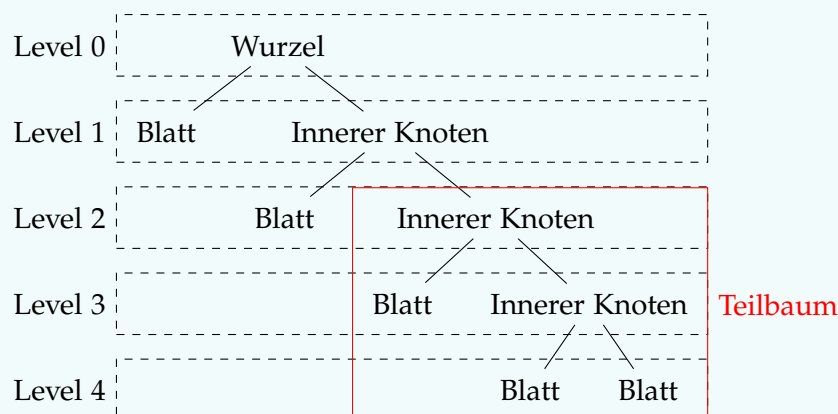


### 3 Bäume

#### Definition: Baum

Ein *Baum* ist eine hierarchische (rekursive) Datenstruktur. Es gilt:

- alle Wege gehen von einer *Wurzel* aus
- $A$  heißt *Vorgänger* von  $B$  bzw.  $B$  *Nachfolger* von  $A$ , wenn  $A$  auf einem Weg von der Wurzel zu  $B$  liegt
- $A$  heißt *Elterknoten* von  $B$ , bzw.  $B$  heißt *Kind* von  $A$ , wenn  $(A, B) \in E$
- Knoten ohne Kinder heißen *Blätter*
- Knoten mit Kindern heißen *innere Knoten*
- ein Knoten  $S$  mit allen Nachfolgern wird *Teilbaum* eines Baumes  $T$  genannt, falls  $S$  nicht Wurzel von  $T$  ist
- der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kinder



#### Definition: Binärbaum

Die Knoten eines *Binärbaums* (*binary tree*) haben höchstens den Verzweigungsgrad 2.

Bei einem *geordneten Binärbaum* ist die Reihenfolge der Kinder durch die Indizes eindeutig festgelegt:

- $T_l$ : linkes Kind, linker Teilbaum
- $T_r$ : rechtes Kind, rechter Teilbaum

Ein Binärbaum heißt *minimal* (bezogen auf die Höhe), wenn kein Binärbaum mit gleicher Knotenzahl aber kleinerer Höhe existiert.

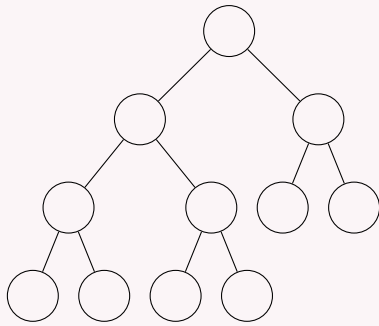
Ein *links-vollständiger Binärbaum* ist ein minimaler Binärbaum, in dem die Knoten auf dem untersten Level so weit wie möglich links stehen.

Alle Blätter eines *vollständigen Binärbaums* haben den gleichen Level und dieser ist vollbesetzt.

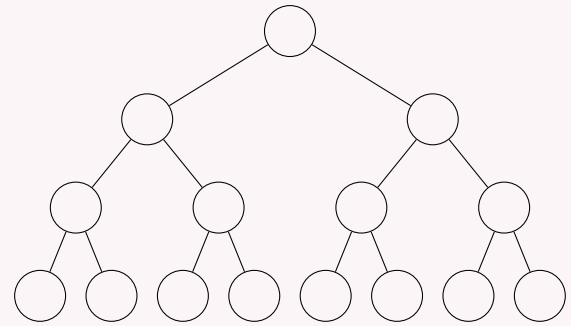
Ein vollständiger Binärbaum der Höhe  $H$  hat

$$n = 1 + 2 + 4 + \dots + 2^H = \frac{2^{H+1} - 1}{2 - 1} = 2^{H+1} - 1 \text{ Knoten}$$

Beispiel: Linksvollständiger Binärbaum



Beispiel: Vollständiger Binärbaum



### 3.1 Binäre Suchbäume

#### Definition: Binärer Suchbaum

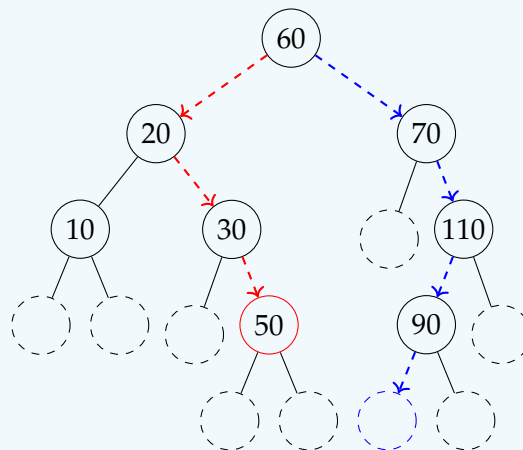
Ein *binärer Suchbaum* ist ein Binärbaum, bei dem für jeden Knoten des Baumes gilt:

Alle Schlüssel im linken Teilbaum sind kleiner, alle im rechten Teilbaum sind größer oder gleich dem Schlüssel in diesem Knoten.

#### Algorithmus: Suchen im binären Suchbaum

Suchen ist ohne Probleme durch einfaches Vergleichen ( $<$  bzw.  $\geq$ ) möglich.

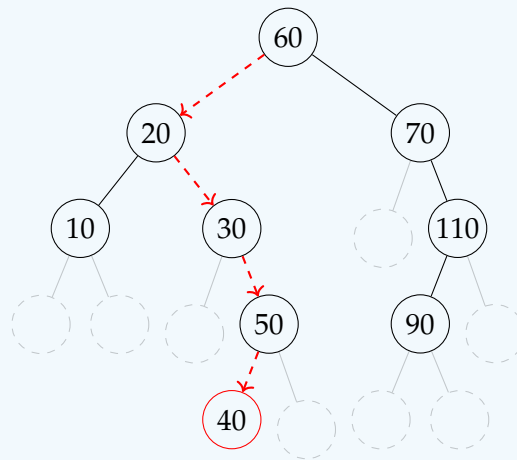
Suchen der 50 in **rot** bzw. (erfolgloses) Suchen der 80 in **blau**.



### Algorithmus: Einfügen im binären Suchbaum

Ein Knoten kann ohne Probleme hinzugefügt werden, indem man solange sucht, bis man auf einen leeren Kindknoten trifft und dort einfügt.

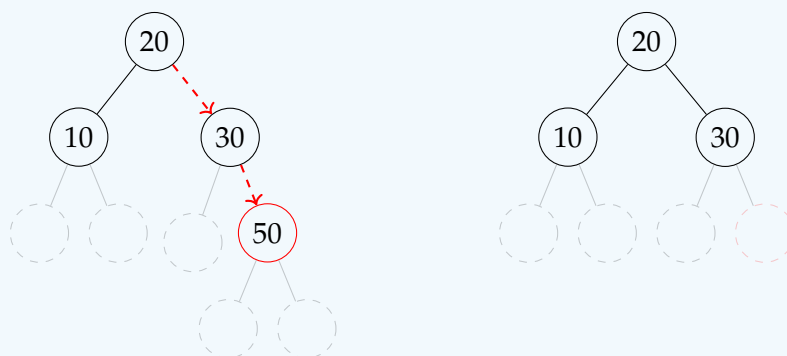
Einfügen der 40 in **rot**.



### Algorithmus: Löschen im binären Suchbaum (Blatt)

Ein Blatt kann problemlos gelöscht werden.

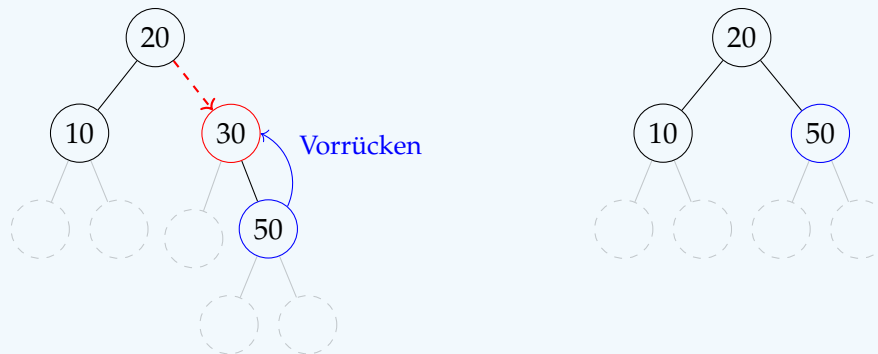
Löschen der 50 in **rot**.



#### Algorithmus: Löschen im binären Suchbaum (Innerer Knoten mit einem Kind)

Soll ein innerer Knoten mit einem Kind gelöscht werden, rückt das Kind an die Stelle des Elterknotens.

Löschen der 30 in **rot**.



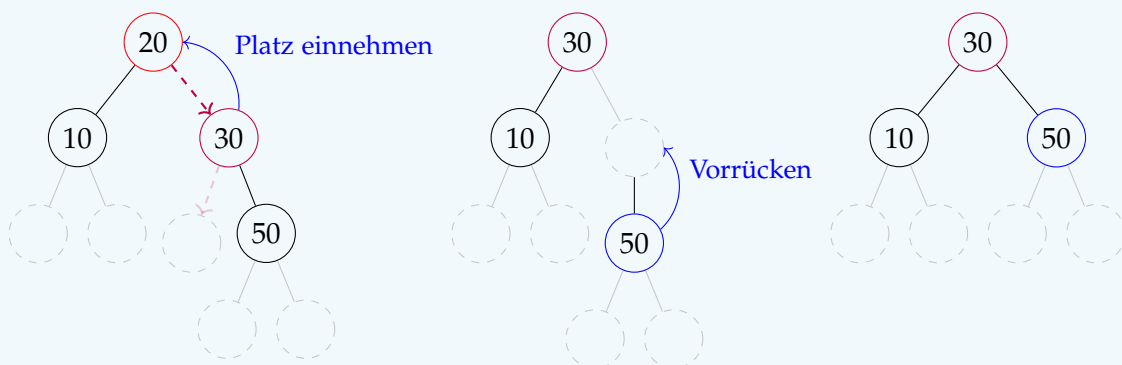
#### Algorithmus: Löschen im binären Suchbaum (Innerer Knoten mit zwei Kindern)

Soll ein innerer Knoten mit zwei Kindern gelöscht werden, nimmt der nächstgrößere Knoten seinen Platz ein.

Dieser wird wie folgt ermittelt (in **lila**):

1. Gehe einen Schritt nach rechts.
2. Gehe solange nach links, bis es kein linkes Kind mehr gibt.

Löschen der 20 in **rot**.

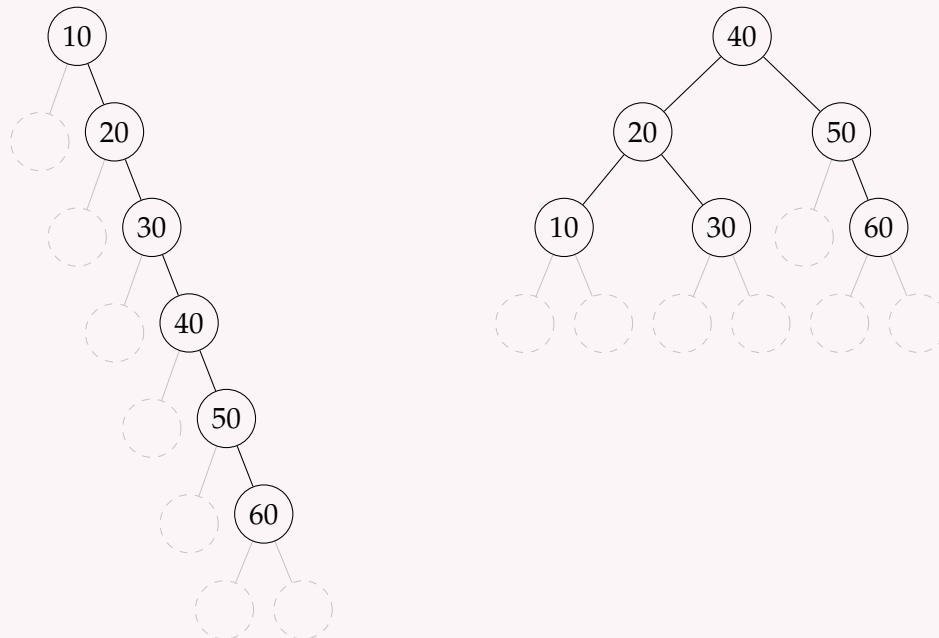




### Bonus: Komplexität beim Suchen, Löschen und Einfügen in Binärbäumen

Die Komplexität der Funktionen Suchen, Löschen und Einfügen werden durch die Komplexität des Suchens eines Elements bestimmt.

Im schlechtesten Fall ist die Anzahl der zu durchsuchenden Elemente gleich der Höhe des Baumes +1. Dabei hängt die Höhe stark von der Reihenfolge der Einfügeoperationen ab.



#### Definition: Balanciertheit

Ein Binärbaum mit  $n$  Knoten hat im besten Fall (*optimal balanciert*) die Höhe

$$H = \lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2(n) \rfloor$$

Dabei ist Suchen in  $\mathcal{O}(\log n)$ .

Ein Binärbaum mit  $n$  Knoten hat im schlechtesten Fall (*entartet/degeneriert*) die Höhe

$$H = n - 1$$

Dabei ist Suchen in  $\mathcal{O}(n)$ .

### Definition: Balance-Kriterien

1. Abgeschwächtes Kriterium für ausgeglichene Höhe
  - lokale Umordnungsoperationen reichen aus
  - z.B. *AVL-Bäume* und *Rot-Schwarz-Bäume*
2. Jeder neue Knoten wandert an die Wurzel des Baumes
  - Vorteil: Zuletzt eingefügte Elemente lassen sich schneller finden
  - durch spezielles Einfügeverfahren wird Baum zusätzlich (teilweise) ausgeglichen
  - z.B. *Splay-Bäume*
3. Unausgeglichener Verzweigungsgrad ermöglicht ausgeglichene Höhe
  - z.B. *B-Bäume*

## 3.2 AVL-Bäume

### Definition: AVL-Baum

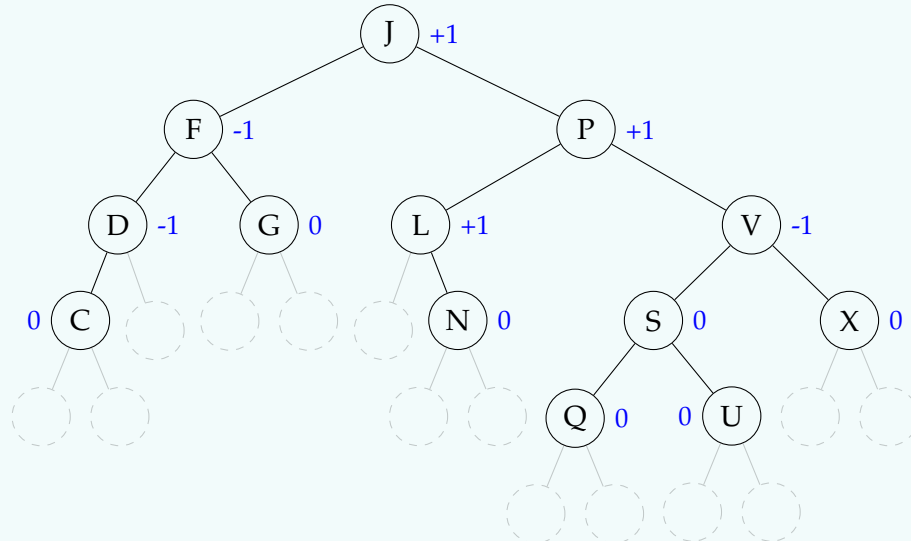
Bei einem *AVL-Baum* unterscheiden sich die Höhen zweier Teilbäume des gleichen Knotens maximal um 1.

Der sogenannte *Balance-Index* (oder *Balance-Faktor*)  $BF$  eines Knotens  $T$  ist die Differenz

$$BF(T) := H(T_r) - H(T_l)$$

Dabei gilt:

- Jeder Knoten hat einen Balance-Index.
- Er darf nur die Werte  $-1$ ,  $0$  oder  $1$  annehmen.



### Algorithmus: Einfügen in einem AVL-Baum

Beim Einfügen in einen AVL-Baum wird zu Beginn analog zu einem regulären Binärbaum eingefügt.

Anschließend wird sich der unmittelbare Elterknoten  $E$  angeschaut und es gibt für den dortigen Balance-Index  $BF(E)$  drei Fälle:

1.  $BF(E)$  wird 0:
  - $BF(E)$  war vorher  $-1$
  - man kommt von einem Kindbaum, der vorher niedriger war
  - Höhe des Knotens ändert sich nicht
  - oberhalb bleiben alle Balance-Indizes gleich
  - $\implies$  AVL-Kriterium ist für den ganzen Baum erfüllt
2.  $BF(E)$  wird  $\pm 1$ :
  - $BF(E)$  war vorher 0
  - Höhe des Teilbaums erhöht sich um 1
  - $\implies$  Überprüfung der Balance-Indizes muss beim Elternknoten von  $E$  fortgesetzt werden
3.  $BF(E)$  wird  $\pm 2$ :
  - $BF(E)$  war vorher  $\pm 1$
  - $\implies$  Teilbaum muss *rebalanciert* werden

### Algorithmus: Löschen in einem AVL-Baum

Beim Löschen in einen AVL-Baum wird zu Beginn analog zu einem regulären Binärbaum gelöscht.

Anschließend wird sich der unmittelbare Elterknoten  $E$  angeschaut und es gibt für den dortigen Balance-Index  $BF(E)$  drei Fälle:

1.  $BF(E)$  wird  $\pm 1$ :
  - $BF(E)$  war vorher 0
  - Höhe des Knotens ändert sich nicht
  - oberhalb bleiben alle Balance-Indizes gleich
  - $\implies$  AVL-Kriterium ist für den ganzen Baum erfüllt
2.  $BF(E)$  wird 0:
  - $BF(E)$  war vorher 0
  - Höhe des Teilbaums verringert sich um 1
  - $\implies$  Überprüfung der Balance-Indizes muss beim Elternknoten von  $E$  fortgesetzt werden
3.  $BF(E)$  wird  $\pm 2$ :
  - $BF(E)$  war vorher  $\pm 1$
  - $\implies$  Teilbaum muss *rebalanciert* werden

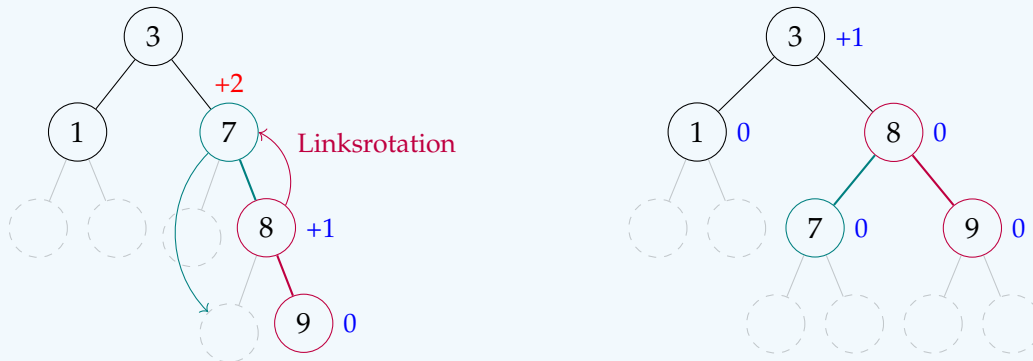
### Algorithmus: Rebalancierung

Wenn bei einer Opeation ein Höhenunterschied von mehr als 1 zwischen zwei Geschwister-Teilbäumen entsteht, ist beim Elterknoten das AVL-Kriterium verletzt. Eine entsprechende Korrektur heißt *Rebalancierung*. Als Werkzeuge eignen sich hierfür die sogenannten *Rotationen*.

## Algorithmus: Einfachrotation

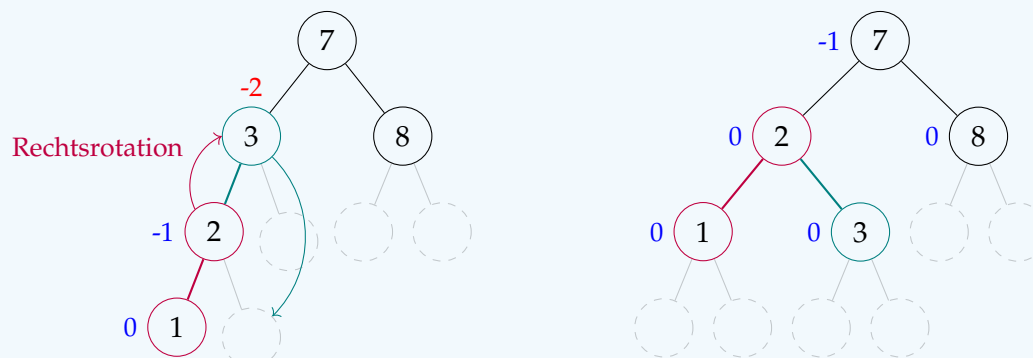
Wird ein AVL-Baum unbalanciert, wenn ein Knoten in den *rechten Teilbaum des rechten Teilbaums* eingefügt wird (Rechts-Rechts-Situation), dann wird das durch eine *Einfachrotation nach links* gelöst.

Zuletzt wurde 9 eingefügt. Linksrotation in **lila**.



Wird ein AVL-Baum unbalanciert, wenn ein Knoten in den *linken Teilbaum des linken Teilbaums* eingefügt wird (Links-Links-Situation), dann wird das durch eine *Einfachrotation nach rechts* gelöst.

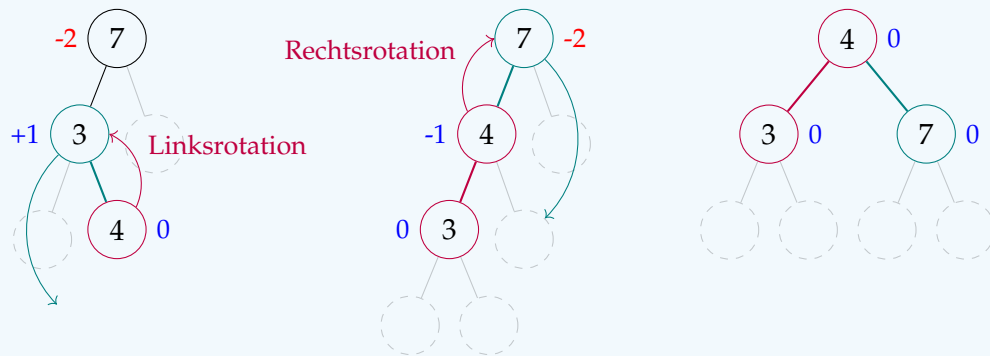
Zuletzt wurde 1 eingefügt. Rechtsrotation in **lila**.



## Algorithmus: Doppelrotation

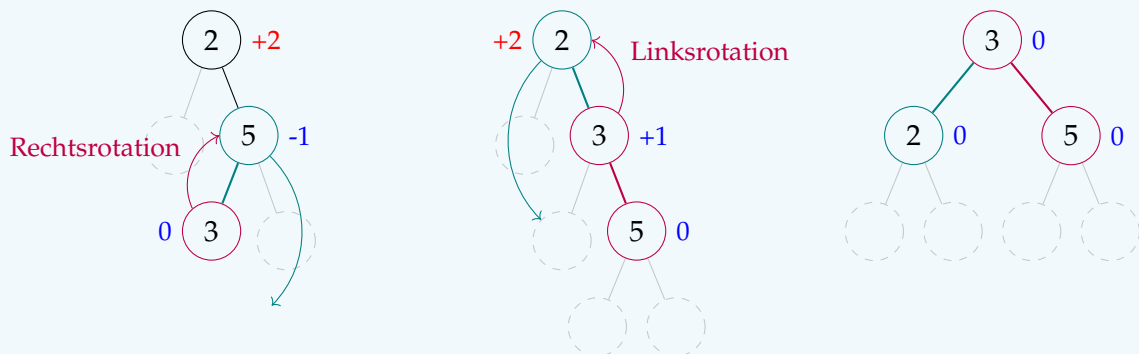
Wird ein AVL-Baum unbalanciert, wenn ein Knoten in den *rechten Teilbaum des linken Teilbaums* eingefügt wird, dann wird das durch eine *Doppelrotation* (Linksrotation, gefolgt von Rechtsrotation) gelöst.

Zuletzt wurde 4 eingefügt. Rotationen in **lila**.



Wird ein AVL-Baum unbalanciert, wenn ein Knoten in den *linken Teilbaum des rechten Teilbaums* eingefügt wird, dann wird das durch eine *Doppelrotation* (Rechtsrotation, gefolgt von Linksrotation) gelöst.

Zuletzt wurde 3 eingefügt. Rotationen in **lila**.



### Definition: Komplexität von AVL-Bäumen

- Einfügen
  - Element muss gesucht werden:  $\mathcal{O}(\log n)$
  - Element muss angehängt werden:  $\mathcal{O}(1)$
  - Baum muss ausgeglichen werden:  $\mathcal{O}(\log n)$
- Löschen
  - Element muss gesucht werden:  $\mathcal{O}(\log n)$
  - nächstgrößeres Element muss gesucht werden:  $\mathcal{O}(\log n)$
  - Elemente müssen verschoben werden:  $\mathcal{O}(1)$
  - Baum muss ausgeglichen werden:  $\mathcal{O}(\log n)^a$
- Prüfen/Auslesen
  - Element muss gesucht werden:  $\mathcal{O}(\log n)$

<sup>a</sup>Im Worst-Case muss für jede Ebene eine Doppelrotation durchgeführt werden  $\implies \mathcal{O}(\log n)$

## 3.3 B-Bäume

### Definition: B-Baum

Jeder Knoten in einem B-Baum der Ordnung  $d$  enthält  $d$  bis  $2d$  Elemente.

Die Wurzel bildet die einzige Ausnahme, sie kann 1 bis  $2d$  Elemente enthalten.

Die Elemente in einem Knoten sind aufsteigend sortiert.

Die Anzahl der Kinder in einem B-Baum ist entweder 0 (Blatt) oder um eins größer als die Anzahl der Elemente, die der Knoten enthält.

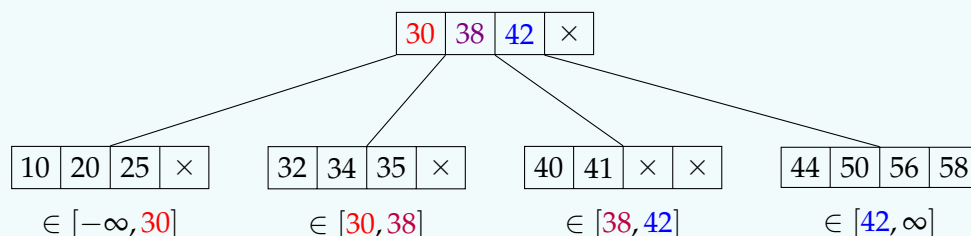
Alle Blätter liegen auf demselben Level.

- garantierte Zugriffszeiten
- bei realistischen Parametern (z.B. Ordnung 1000) sind sehr wenige ( $< 5$ ) Zugriffe auf das externe Medium nötig

B-Bäume besitzen ausgeglichene Höhe, lassen aber unausgeglichene Verzweigungsgrad und Knotenfüllgrad zu.

Der längste Weg in einem B-Baum der Ordnung  $d$  ist in  $\mathcal{O}(\log_{d+1} n)$ .

B-Baum der Ordnung 2:



## Algorithmus: Suchen in einem B-Baum

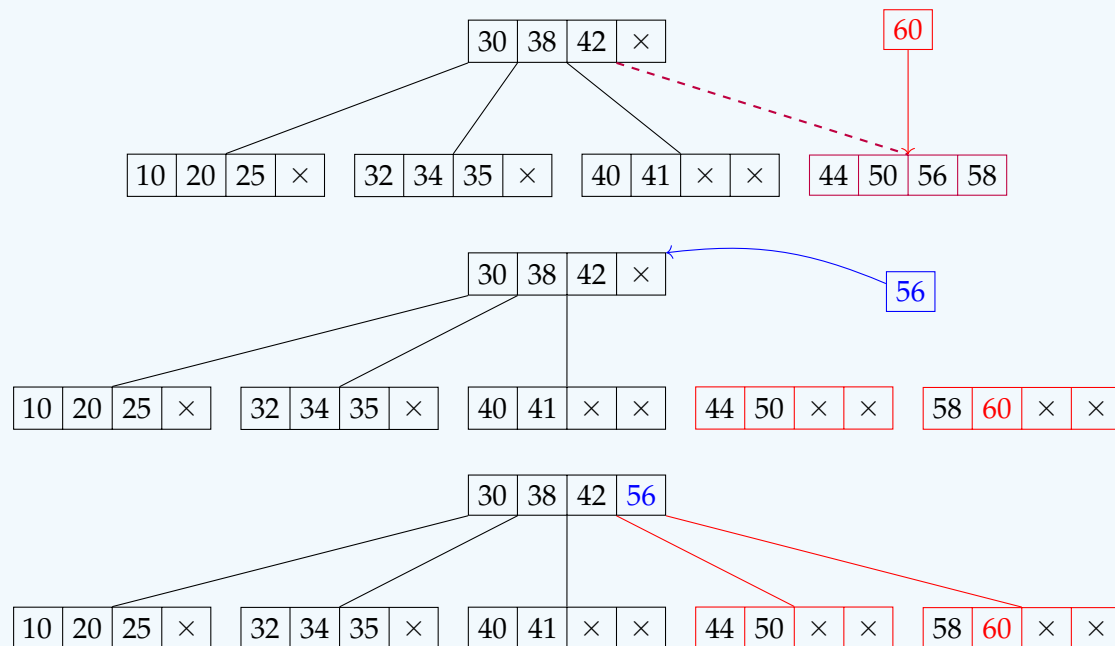
Ausgehend von der Wurzel:

1. Prüfe, ob der gerade betrachtete Knoten den gesuchten Schlüssel  $m$  enthält.  
(Suche innerhalb eines Knotens entweder linear oder binär.)
2. Falls nicht, bestimme den kleinsten Schlüssel  $k_i$ , der größer als  $m$  ist.
  - $k_i$  gefunden: Weiter bei Schritt 1 mit linkem Kind von  $k_i$  ( $p_{i-1}$ )
  - $k_i$  nicht gefunden: Weiter mit letztem Kind ( $p_n$ )

## Algorithmus: Einfügen in einem B-Baum der Ordnung $d$

1. Suche nach Schlüssel endet in einem Blatt node (in **lila**)
2. Schlüssel wird in Sortierreihenfolge eingefügt (und neuer leerer Verweis eingefügt)
3. Falls node überfüllt ist: node aufteilen  
 $k$  sei mittlerer Eintrag von node
  - a) Neuen Knoten current anlegen und mit den  $d$  größeren Schlüsseln (rechts von  $k$ ) belegen.
  - b) Die  $d$  kleineren Schlüssel (links von  $k$ ) bleiben in node.
  - c)  $k$  in Elterknoten parent von node verschieben.
  - d) Verweis rechts von  $k$  in parent mit current verbinden.
4. Falls parent nun überfüllt ist: parent aufteilen (Siehe Schritt 3)

Einfügen der 60 in **rot**.



### Algorithmus: Löschen in einem B-Baum der Ordnung $d$ (Blatt)

In einem Blatt mit Struktur

$(\text{null}, k_1, \text{null}, \dots, k_i, \text{null}, \dots, k_n, \text{null})$

(null sind hier die Kinder an der jeweiligen Stelle) wird der Wert  $x = k_i$  zusammen mit der darauf folgenden null-Referenz gelöscht.

Ein *Underflow* tritt auf, falls  $n = d$  war.

### Algorithmus: Löschen in einem B-Baum der Ordnung $d$ (Innerer Knoten)

In einem inneren Knoten mit Struktur

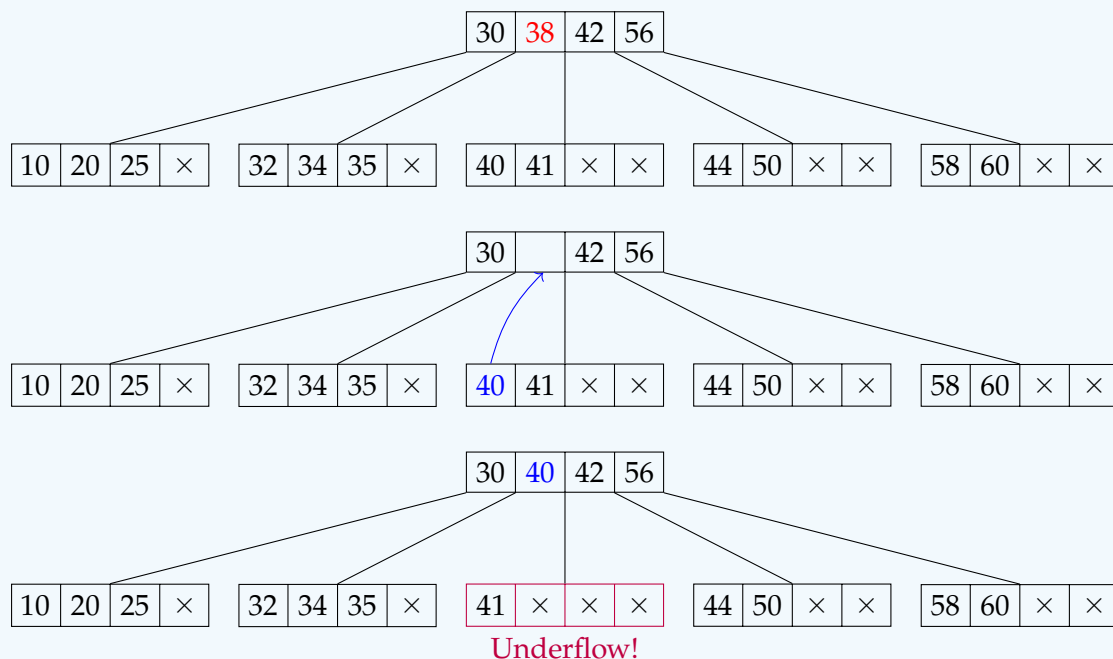
$(p_0, k_1, p_1, \dots, k_i, p_i, \dots, k_n, p_n)$

( $p_j$  sind hier die Kinder an der jeweiligen Stelle) haben alle Referenzen einen Wert ungleich null.

Das Löschen eines Wertes  $x = k_i$  funktioniert analog zum Löschen aus einem binären Suchbaum:

1. Finde kleinsten Schlüssel  $s$  im durch  $p_i$  referenzierten Teilbaum (in einem Blatt)
2. Ersetze  $k_i$  durch  $s$  und lösche  $s$  aus dem Blatt

Löschen der 38 in **rot**.



### Definition: Underflow

Ein *Underflow* tritt in einem B-Baum genau dann auf, wenn zu wenig ( $< d$ ) Schlüssel im Knoten sind.



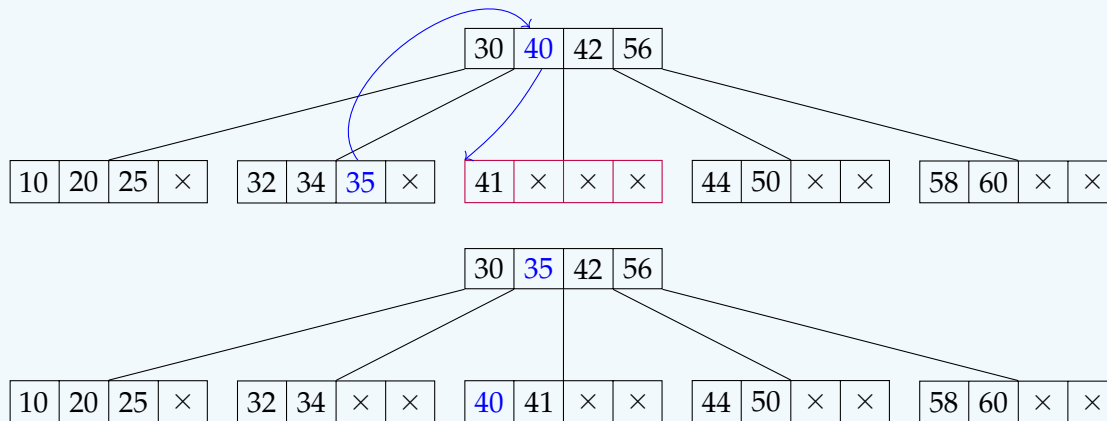
### Algorithmus: Ausgleich zwischen Geschwisterknoten

Voraussetzung: Knoten  $q$  mit Underflow hat *benachbarten* Geschwisterknoten  $p$  mit  $> d$  Schlüsseln.

Annahme:

- $p$  ist linker Geschwisterknoten von  $q$  (analog mit rechtem Geschwisterknoten)
- im Elterknoten parent (von  $p$  und  $q$ ) trennt der Schlüssel  $t$  die Verweise auf  $p$  und  $q$

Idee:  $p$  schenkt  $q$  ein Element („Umweg“ über Elterknoten)



### Algorithmus: Verschmelzen von Geschwisterknoten

Voraussetzung: Knoten  $q$  hat *benachbarten* Geschwisterknoten mit  $d$  Schlüsseln.

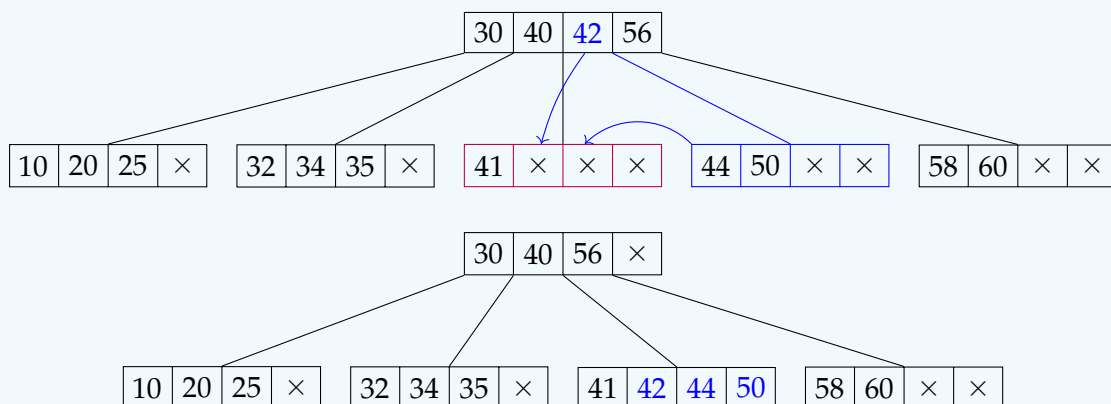
Annahme:

- $p$  ist linker Geschwisterknoten von  $q$  (analog mit rechtem Geschwisterknoten)
- im Elterknoten parent (von  $p$  und  $q$ ) trennt der Schlüssel  $t$  die Verweise auf  $p$  und  $q$

Idee:  $p$  und  $q$  mit dem trennenden Element aus parent verschmelzen.

Beachte:

- Eventueller Underflow in parent muss behandelt werden (rekursiv)
- Falls letzter Schlüssel der Wurzel gelöscht wird, wird der einzige Nachfolger der Wurzel die neue Wurzel (Höhe des B-Baums wird um 1 verringert).



#### Definition: B+-Baum

Im Unterschied zu B-Bäumen speichern *B+-Bäume* ihre Datensätze ausschließlich in den Blättern.

Dies ist bei der Anwendung für Datenbanken naheliegend und sinnvoll.

### 3.4 Rot-Schwarz-Bäume

#### Definition: Rot-Schwarz-Baum

Ein *Rot-Schwarz-Baum* ist ein balancierter binärer Suchbaum, in dem jeder innere Knoten zwei Kinder hat.

Jeder innere Knoten hat eine Farbe, so dass gilt:

- Die Wurzel ist schwarz.
- Alle Blätter (null-Knoten) sind schwarz.
- Für jeden Knoten gilt, dass jeder Pfad zu den Blättern die gleiche Anzahl an schwarzen Knoten hat. (Schwarz-Tiefe)
- Beide Kinder eines roten Knotens sind schwarz.

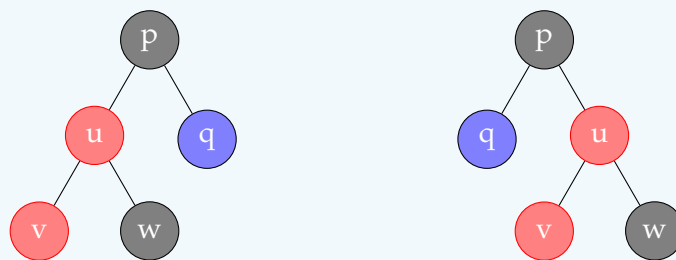
Rot-Schwarz-Bäume sind eine gängige Alternative zu AVL-Bäumen.

#### Algorithmus: Einfügen in einen Rot-Schwarz-Baum

Zuerst wird wie in einem normalen Binärbaum eingefügt, danach werden die Rot-Schwarz-Bedingungen repariert.

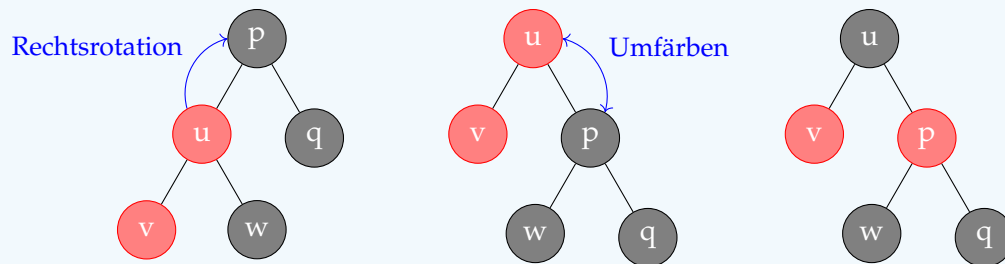
Annahmen:

- eingefügter Knoten  $v$  ist rot
- Elterknoten  $u$  von  $v$  ist rot (sonst fertig)
- $v$  ist linkes Kind von  $u$  (anderer Fall symmetrisch)
- Geschwisterknoten  $w$  (rechtes Kind von  $u$ ) ist schwarz
- Alle roten Knoten außer  $u$  haben 2 schwarze Kinder

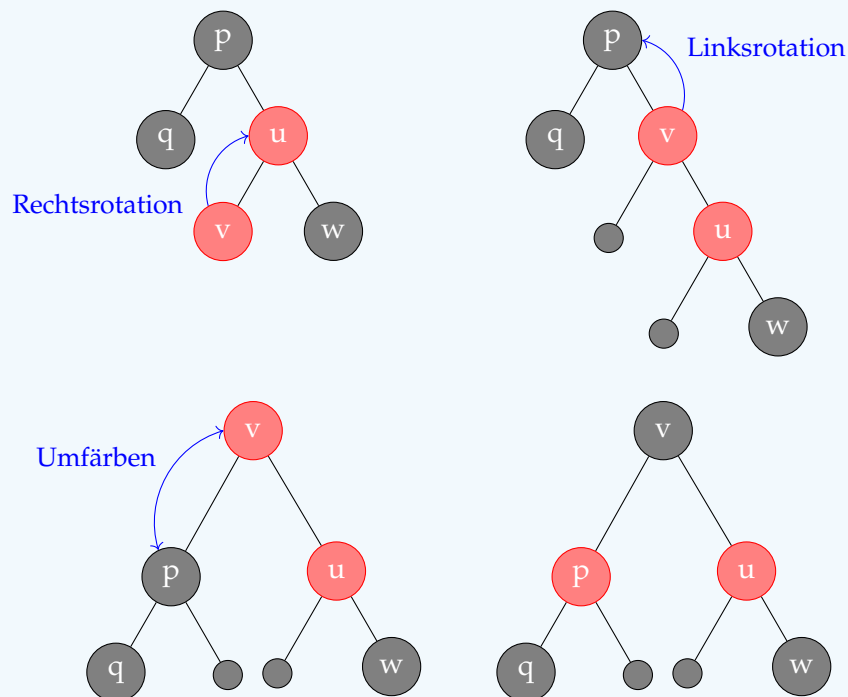


**Fall 1: Onkelknoten  $q$  von  $v$  ist schwarz**

Fall 1a:  $u$  ist linkes Kind von  $p$  ( $v-u-p$  bilden eine Linie)



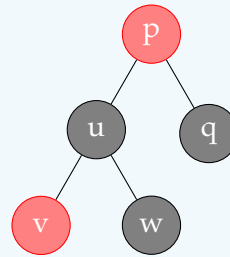
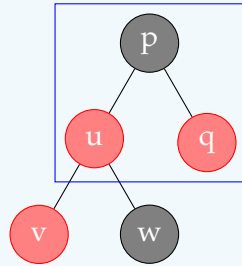
Fall 1b:  $u$  ist rechtes Kind von  $p$  ( $v-u-p$  bilden ein Dreieck)



## Algorithmus: Einfügen in einen Rot-Schwarz-Baum (Fall 2)

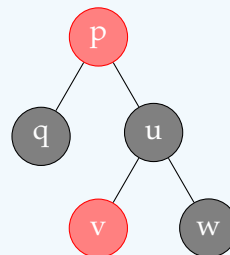
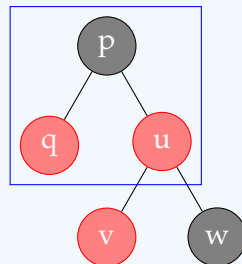
Fall 2: Geschwisterknoten  $q$  von  $u$  ist rot

Umfärben



beziehungsweise

Umfärben



- Falls der Elterknoten von  $p$  schwarz ist, sind wir fertig
- Falls  $p$  die Wurzel ist, färbe  $p$  schwarz
- Sonst behandle  $p$  wie  $v$  und wiederhole

## Algorithmus: Löschen in einem Rot-Schwarz-Baum

Das Löschen in einem Rot-Schwarz-Baum kann schnell sehr schwierig zu visualisieren werden.

Ich will hier gern einmal folgende Videos empfehlen:

- <https://youtu.be/e03GzpCCUSg> (ausführliche Beispiele, englisch)
- <https://youtu.be/bDT1woMULVw> (ausführliche Erklärung, Pseudocode, deutsch)

## 3.5 Heaps

### Bonus: Heap (Wortbedeutungen)

Das Wort *Heap* hat zwei Bedeutungen:

- Besonderer Speicherbereich, in dem Objekte und Klassen gespeichert werden.
- Datenstruktur zur effizienten Implementierung einer Prioritätswarteschlange.

### Definition: Heap

Ein *Heap* ist ein Binärbaum mit folgenden Eigenschaften:

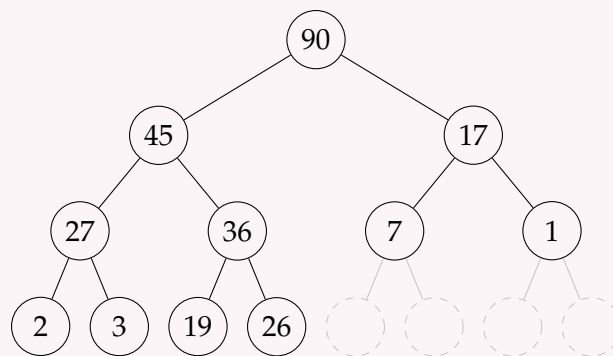
- linksvollständig
- Kinder eines Knotens höchstens so groß wie der Knoten selbst (Max-Heap)
- größtes Element befindet sich an der Wurzel (Max-Heap)
- entlang jedes Pfades von einem Knoten zur Wurzel sind Knoteninhalte aufsteigend sortiert

Ein Heap lässt sich insbesondere als Array sehr leicht speichern. Dabei gilt:

- $\text{heap}[0]$  ist die Wurzel
- $\text{heap}[(i-1)/2]$  ist der Elterknoten des Knoten  $i$
- $\text{heap}[(2*i)+1]$  ist das linke Kind des Knoten  $i$
- $\text{heap}[(2*i)+2]$  ist das rechte Kind des Knoten  $i$

Das entspricht einem Level-Order-Baumdurchlauf.

### Beispiel: Heap als Array



Für den Heap oben gilt die Array-Darstellung:

[90, 45, 17, 27, 36, 7, 1, 2, 3, 19, 26]

### Algorithmus: Einfügen in einem Heap

Das Einfügen eines Elements in den Heap erfolgt, indem das neue Element an das Ende des Heaps gesetzt wird.

Weil das neu eingesetzte Element die Eigenschaften des Heaps verzerren kann, wird die Operation *Up-Heapify* durchgeführt, um die Eigenschaften des Heaps in einem Bottom-up-Ansatz zu erhalten.

### Algorithmus: Löschen in einem Heap

Das Entfernen eines Elements erfolgt, indem das gelöschte Element durch das letzte Element im Heap ersetzt wird. Dann wird das letzte Element aus dem Heap gelöscht. Nun wird das letzte Element an einer Stelle im Heap platziert.

Es kann die Heap-Bedingung nicht erfüllen, sodass die Operation *Down-Heapify* durchgeführt wird, um die Eigenschaften des Heaps aufrechtzuerhalten.

### Definition: Heapify

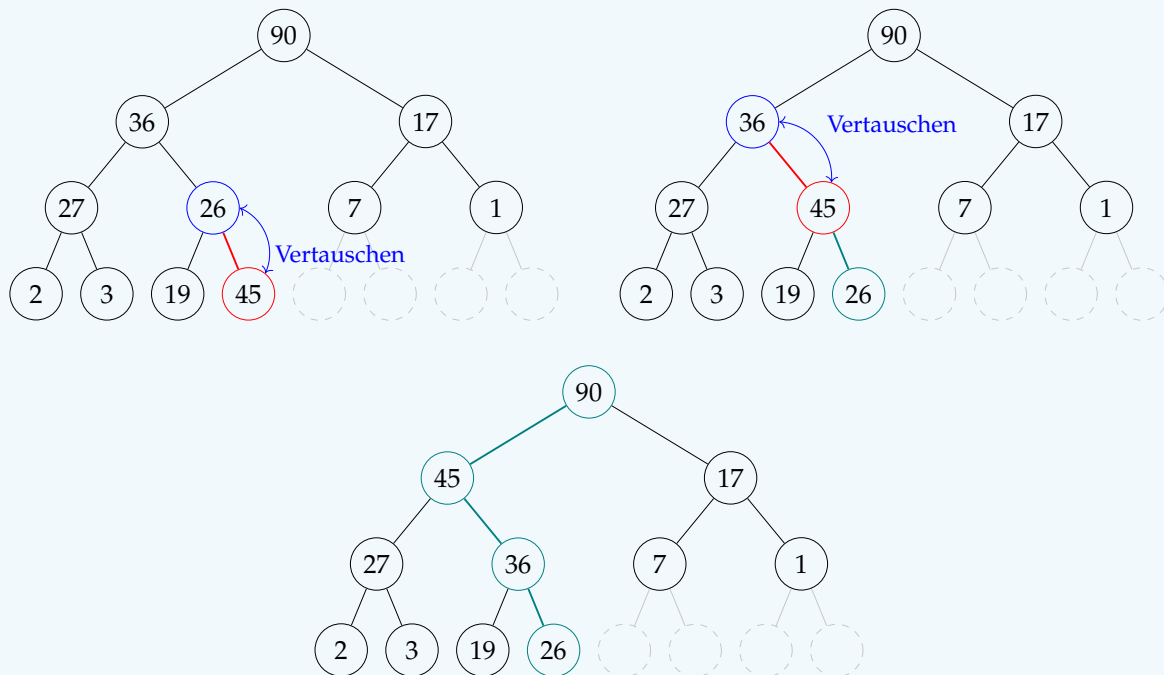
Heapify ist eine Operation, um die Elemente des Heaps neu anzuordnen, um die Heap-Bedingung aufrechtzuerhalten.

Die Heapify kann in zwei Methoden erfolgen:

- Up-Heapify (erfolgt beim Einfügen)
- Down-Heapify (erfolgt beim Löschen)

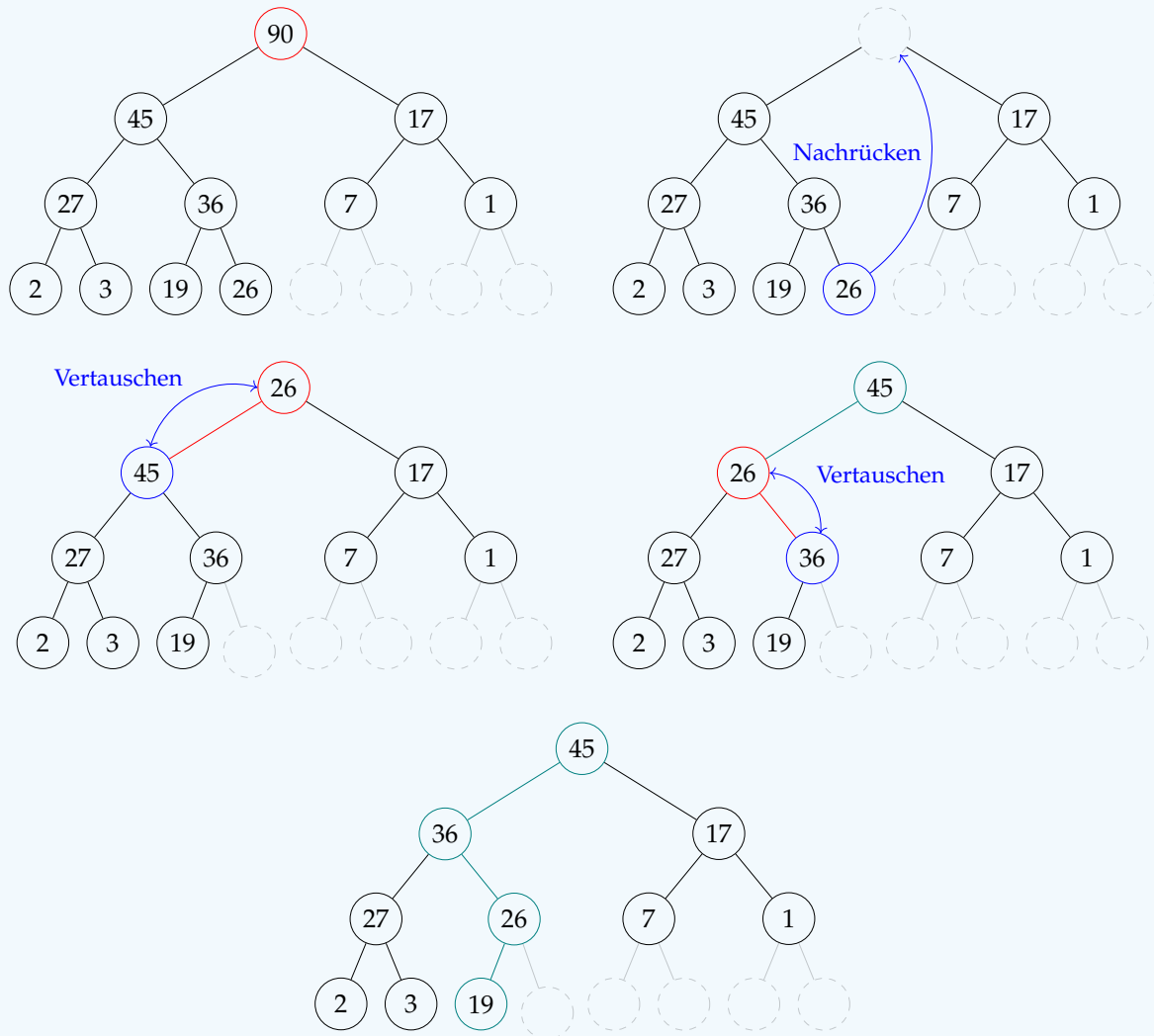
### Algorithmus: Up-Heapify (Einfügen)

Einfügen der 45 in **rot**.



## Algorithmus: Down-Heapify (Löschen)

Löschen der 90 in **rot**.



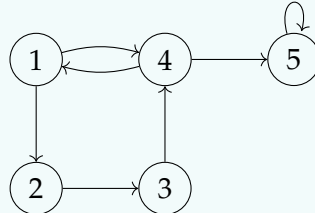
## 4 Graphen

### Definition: Gerichteter Graph

Ein *gerichteter Graph*  $G = (V, E)$  besteht aus

- einer endlichen, nicht leeren Menge  $V = \{v_1, \dots, v_n\}$  von *Knoten (vertices)* und
- einer Relation  $E \subseteq V \times V$  von geordneten Paaren  $e = (u, v)$  den *Kanten (edges)*.

Jede Kante  $(u, v) \in E$  hat einen Anfangsknoten  $u$  und einen Endknoten  $v$  und damit eine Richtung von  $u$  nach  $v$  ( $u = v$  ist möglich).

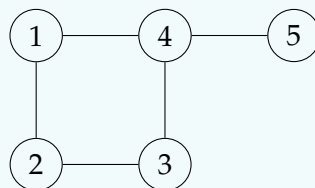


### Definition: Ungerichteter Graph

Ein *ungerichteter Graph*  $G = (V, E)$  besteht aus

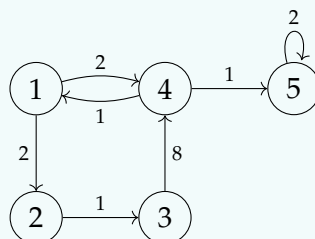
- einer endlichen, nicht leeren Menge  $V = \{v_1, \dots, v_n\}$  von *Knoten (vertices)* und
- einer symmetrischen Relation  $E \subseteq V \times V$  von geordneten Paaren  $e = (u, v) \iff (v, u)$  den *Kanten (edges)*.

Jede Kante  $(u, v) \in E$  hat einen Anfangsknoten  $u$  und einen Endknoten  $v$  und damit eine Richtung von  $u$  nach  $v$  ( $u = v$  ist möglich).



### Definition: Gewichteter Graph

Ein Graph heißt *gewichtet*, wenn jeder Kante ein Wert als *Gewicht* zugeordnet ist (z.B. Transportkosten, Entfernung).



### Definition: Teilgraph

$G' = (V', E')$  heißt *Teilgraph* von  $G = (V, E)$ , wenn gilt:

$$V' \subseteq V \quad \text{und} \quad E' \subseteq E$$



### Definition: Weg

Sei  $G = (V, E)$  ein Graph.

Eine Folge von Knoten

$$W := (v_1, v_2, \dots, v_n)$$

heißt *Weg* oder *Pfad* in  $G$ , falls gilt:

$$\forall 1 \leq i \leq n-1 : (v_i, v_{i+1}) \in E$$

(also eine Folge von zusammenhängenden Kanten)

$\alpha(W) := v_1$  heißt *Anfangsknoten* des Weges  $W$ .

$\omega(W) := v_n$  heißt *Endknoten* des Weges  $W$ .

$\forall v_i \in V : (v_i)$  heißt *trivialer Weg* und ist stets ein Weg in  $G$ .

Die *Länge eines Weges* ist  $l(W) := n - 1$ , falls  $n$  Knoten auf diesem Weg besucht werden.

Ein Weg heißt *einfacher Weg*, wenn kein Knoten (ausgenommen Start- und Endknoten) mehr als einmal vorkommt.

Ein *Zykel* oder *Kreis* ist ein nicht-trivialer einfacher Weg mit der Bedingung  $\alpha(W) = \omega(W)$ .

### Definition: Adjazenz

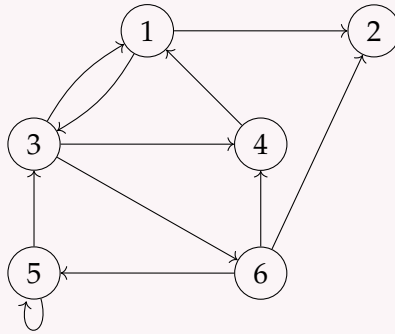
Zwei Knoten heißen *adjazent* (*benachbart*), wenn sie eine Kante verbindet.

### Definition: Speicherung von Graphen

- Kantenorientiert
  - Index für Kanten
  - für jede Kante speichern: Vorgänger-, Nachfolgerknoten (Markierung, Gewicht)
  - meist statische Darstellung, z.B. Kantenliste
- Knotenorientiert
  - gebräuchlicher als kantenorientiert
  - in vielen Ausprägungen, z.B. Knotenliste, Adjazenzmatrix, Adjazenliste
  - für Adjazenzmatrix gilt:

$$A_{ij} = \begin{cases} 1 & , \text{ falls } (i, j) \in E \\ 0 & , \text{ sonst} \end{cases}$$

### Beispiel: Kantenliste



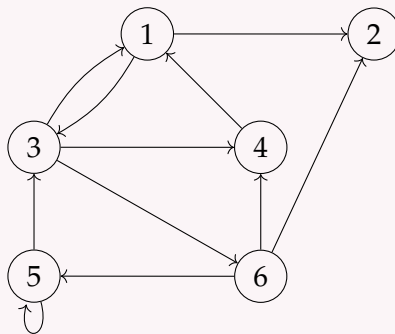
Für eine Kantenliste gilt:

- Position 0: Anzahl der Knoten
- Position 1: Anzahl der Kanten
- Danach für jede Kante  $c = (i, j)$ :
  - Startknoten  $i$
  - Endknoten  $j$

Für den Graphen  $G$  oben gilt die Kantenliste:

[6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4]

### Beispiel: Knotenliste



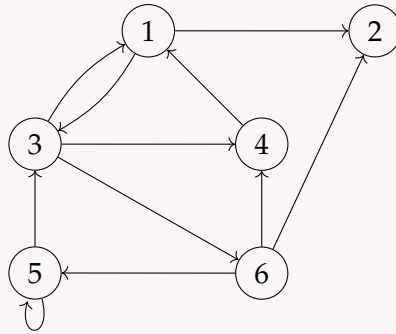
Für eine Knoten gilt:

- Position 0: Anzahl der Knoten
- Position 1: Anzahl der Kanten
- Danach für jeden Knoten  $i$ :
  - Ausgangsgrad des Knotens  $i$  (Anzahl der ausgehenden Kanten)
  - Alle Knoten  $j$  für die gilt  $(i, j) \in E$

Für den Graphen  $G$  oben gilt die Knotenliste:

[6, 11, 2, 2, 3, 0, 3, 1, 4, 6, 1, 1, 2, 3, 5, 3, 2, 4, 5]

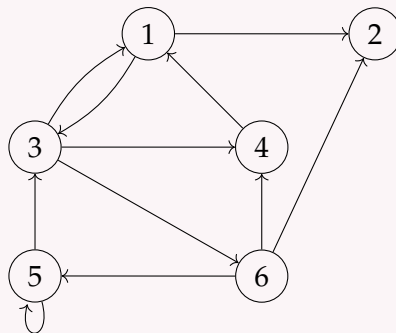
### Beispiel: Adjazenzmatrix



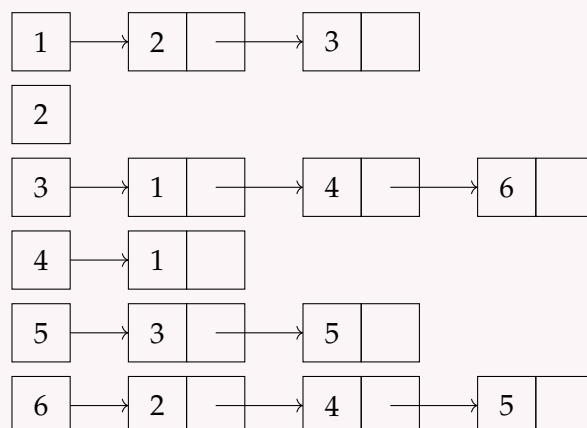
Für den Graphen G oben gilt die Adjazenzmatrix:

$$A(G) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

### Beispiel: Adjazenzliste



Für den Graphen G oben gilt die Adjazenzliste:

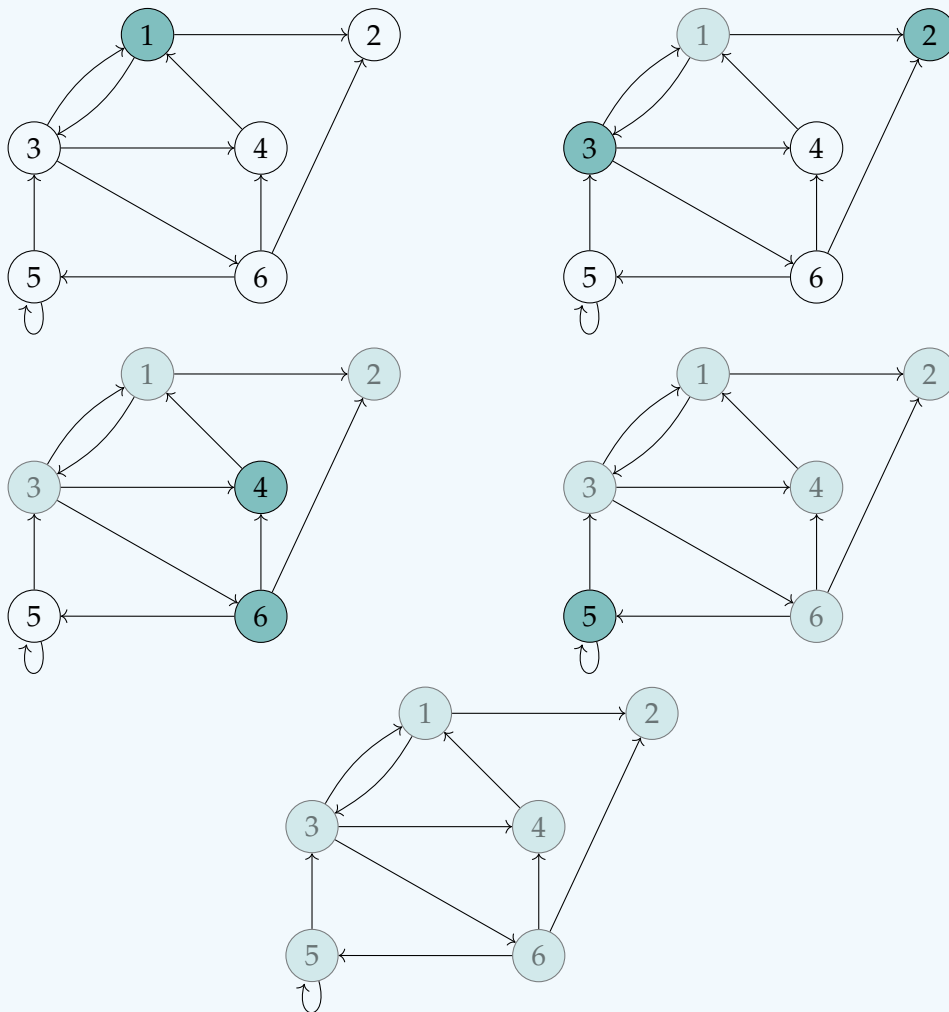


## 4.1 Suche

### Algorithmus: Breitensuche

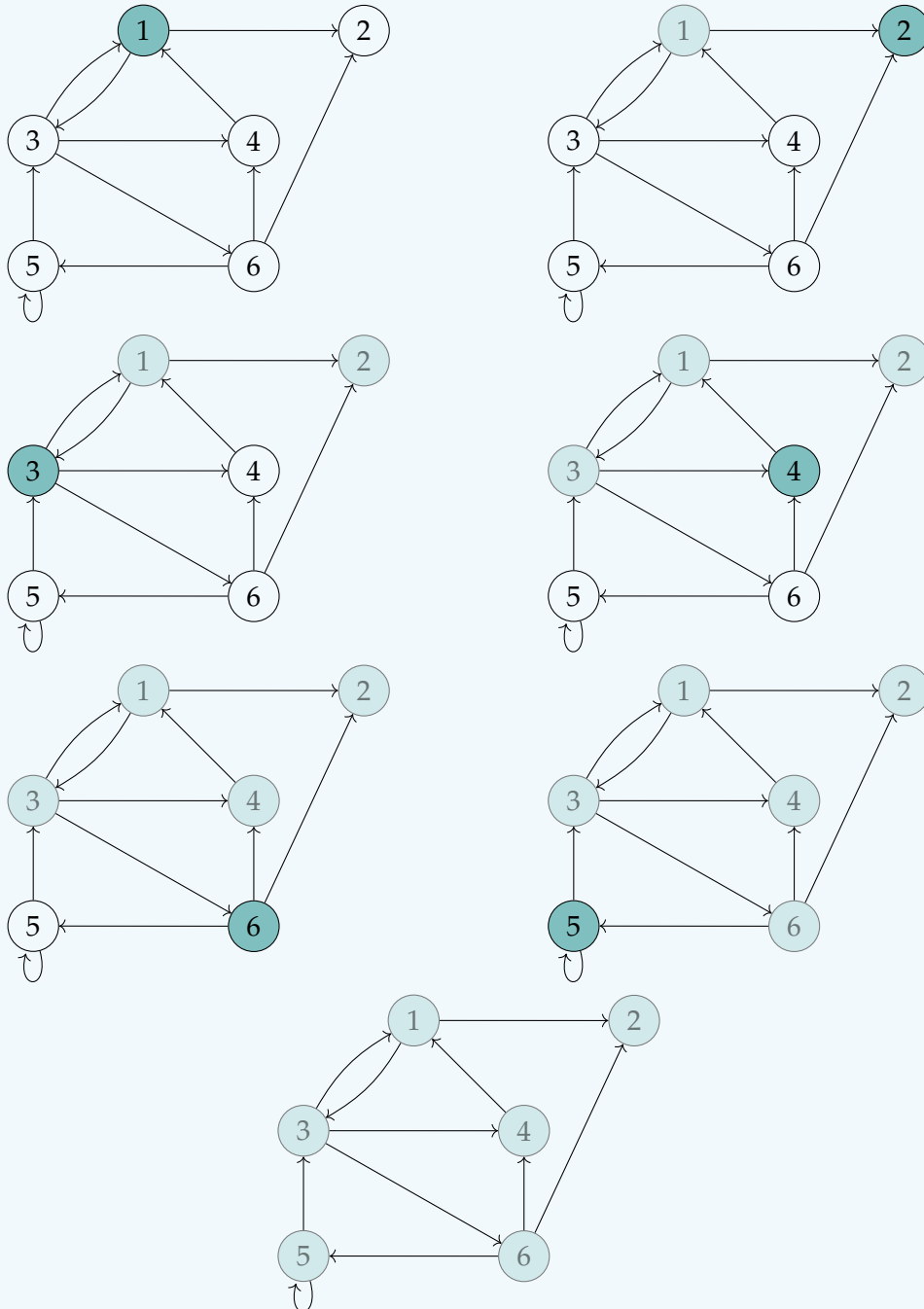
1. Zunächst werden alle Knoten als „noch nicht besucht“ markiert.
2. Startpunkt  $v$  wählen und als „besucht“ markieren.
3. Jetzt:
  - a) alle von  $v$  aus direkt erreichbaren (nicht besuchten) Knoten „besuchen“
  - b) alle von  $v$  über zwei Kanten erreichbaren (nicht besuchten) Knoten „besuchen“
  - c) ...
4. Wenn noch nicht alle Knoten besucht worden sind, wähle einen neuen unbesuchten Startpunkt  $v$  und beginne bei Schritt 3

Start bei Knoten 1.



1. Zunächst alle Knoten als „noch nicht besucht“ markieren.
2. Startpunkt  $v$  wählen und als „besucht“ markieren.
3. Von dort aus möglichst langen Pfad entlang gehen; dabei nur bisher nicht besuchte Knoten „besuchen“
4. Wenn dann noch nicht alle Knoten besucht worden sind, wähle einen neuen unbesuchten Startpunkt  $v$  und beginne bei Schritt 3

Start bei Knoten 1.



#### Definition: Level-Order-Baumdurchlauf

Führt man eine Breitensuche bei Bäumen aus, stellt man fest:

- Es ist nicht nötig, die Knoten zu markieren.
- Der Baum wird Ebene für Ebene durchlaufen,

Diesen Durchlauf nennt man *Level-Order-Durchlauf*.

#### Definition: Pre-Order-Baumdurchlauf

Führt man eine Tiefensuche bei Bäumen aus, nennt man diesen Durchlauf *Pre-Order-Durchlauf*.

Dabei wird stets der linke Teilbaum zuerst durchlaufen.

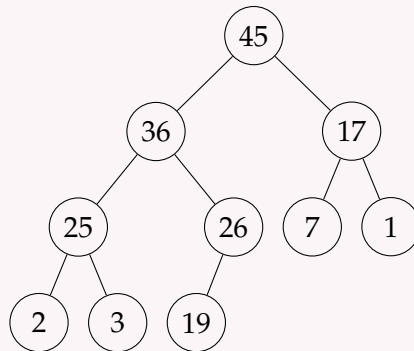
#### Definition: Post-Order-Baumdurchlauf

1. Durchlaufe linken Teilbaum
2. Durchlaufe rechten Teilbaum
3. Betrachte die Wurzel

#### Definition: In-Order-Baumdurchlauf

1. Durchlaufe linken Teilbaum
2. Betrachte die Wurzel
3. Durchlaufe rechten Teilbaum

#### Beispiel: Baumdurchläufe



Für den gegebenen Baum gelten folgende Baumdurchläufe:

Pre-Order	45	36	25	2	3	26	19	17	7	1
In-Order	2	25	3	19	26	36	45	7	17	1
Post-Order	2	3	25	19	26	36	7	1	17	45
Level-Order	45	36	17	25	26	7	1	2	3	19

## 4.2 Entwurfsprinzipien

### Definition: Traveling-Salesman-Problem

Das *Traveling-Salesman-Problem* ist ein Optimierungsproblem. Die Aufgabe besteht darin, eine Reihenfolge für den Besuch mehrerer Orte so zu wählen, dass:

- kein Ort doppelt besucht wird und
- die Reisstrecke minimal ist.

### Definition: Greedy

Wähle immer den Schritt, dessen unmittelbarer Folgezustand das beste Ergebnis liefert. Denke nicht an die weiteren Schritte (greedy = gierig).

Ob ein Greedy-Algorithmus die optimale Lösung liefert oder nicht, ist vom konkreten Problem abhängig.

Für das Traveling-Salesman-Problem ist eine Greedy-Lösung z.B. die *Nearest-Neighbour-Lösung*:

- Gehe in jedem Schritt zur nächstliegenden Stadt, die noch nicht besucht ist.
- $\mathcal{O}(n^2)$
- liefert nicht die optimale Lösung (kann sogar beliebig schlecht werden)
- arbeitet meistens einigermaßen gut

### Bonus: Random Insertion

Wähle immer einen zufälligen Knoten.

- $\mathcal{O}(n^2)$
- keine Garantie, dass die Lösung irgendeinem Gütekriterium genügt
- im Allgemeinen ist Lösung ziemlich gut
- Vorteil: Durch den Zufallsfaktor kann man das Verfahren beliebig oft wiederholen und sich die beste Lösung herausuchen.

### Bonus: Lösung mit minimalem Spannbaum

1. Konstruktion eines minimalen Spannbaums nach Prim
2. Durchlaufen mit Tiefensuche
3. jede Kante wird zweimal durchlaufen
4. summierte Gewichte des minimalen Spannbaums müssen kleiner sein als das optimale TSP-Ergebnis
5. Optimierung:
  - Knoten schon einmal besucht: beim nächsten Mal überspringen
  - bei einem Schritt mehrere Möglichkeiten: zuerst kürzeren Weg wählen (greedy)

#### Definition: Backtracking

Durchlauf eines Lösungsbaum in Pre-Order-Reihenfolge.

Situation: Mehrere Alternativen sind in bestimmten Schritten des Algorithmus möglich.

Lösung mit *Backtracking*:

1. Wähle eine Alternative und verfolge dieses Weg weiter.
2. Falls man so eine Lösung des Problems findet, ist man fertig.
3. Ansonsten gehe einen Schritt zurück und verfolge rekursiv eine andere (nicht versuchte) Alternative in diesem Schritt.
4. Falls alle Alternativen erfolglos probiert wurden, einen Schritt zurückgehen ...

Backtracking-Algorithmen können exponentiellen Aufwand haben.

Tipps:

- Durch Einführung von Zusatzbedingungen möglichst viele Sackgassen ausschließen.
- Symmetriebedingungen ausnutzen.

#### Definition: Branch and Bound

1. Ermitteln einer oberen Schranke (greedy)
2. Falls dieser Wert überschritten wird, kann man die Suche auf diesem Pfad abbrechen.
3. Bessere Lösungen nutzen, um die Schranke zu verbessern.

### 4.3 Graphalgorithmen

#### Definition: Bipartiter Graph

Ein Graph  $G = (V, E)$  heißt *bipartit*, wenn man  $V$  in disjunkte Mengen  $U$  und  $W$  zerlegen kann, so dass alle Kanten zwischen  $U$  und  $W$  verlaufen.

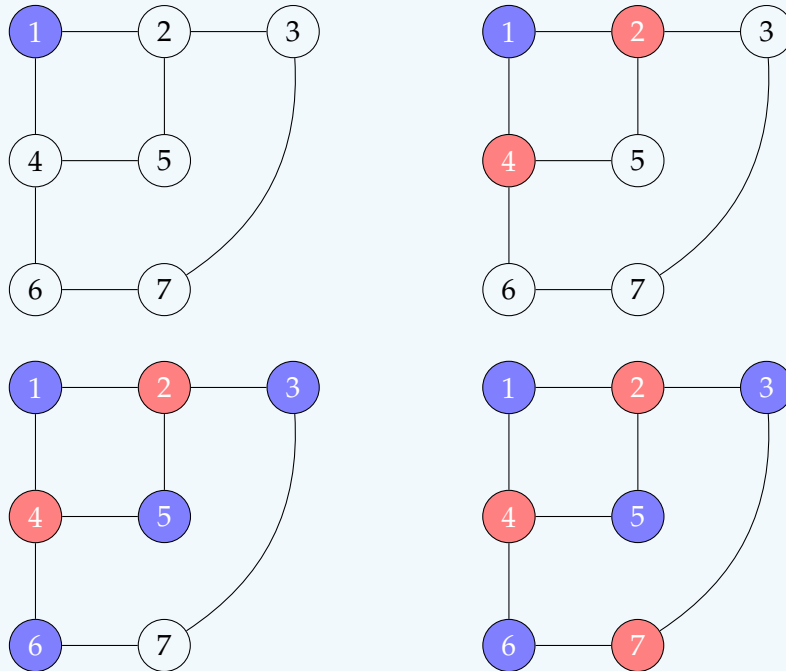
Die Knoten eines bipartiten Graphen lassen sich mit zwei Farben so einfärben, dass zwei Nachbarknoten immer unterschiedlich eingefärbt sind.



### Algorithmus: Prüfung ob ein Graph bipartit ist

1. Einfärben des 1. Knotens
2. Durchlauf mit Breitensuche
3. Abwechselndes Färben

Start bei Knoten 1.



### Definition: Spannbaum

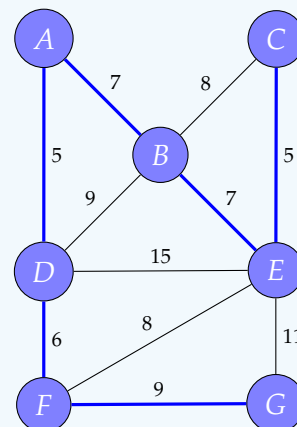
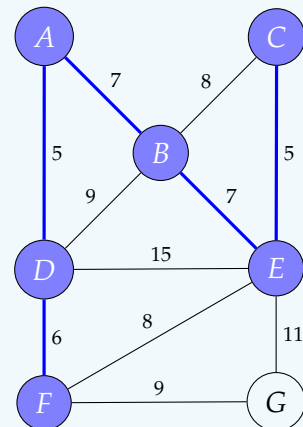
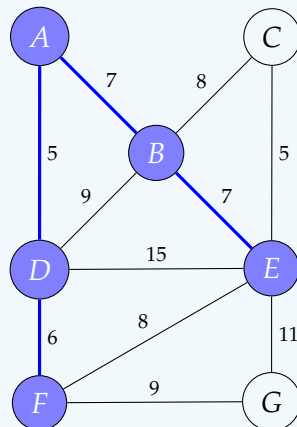
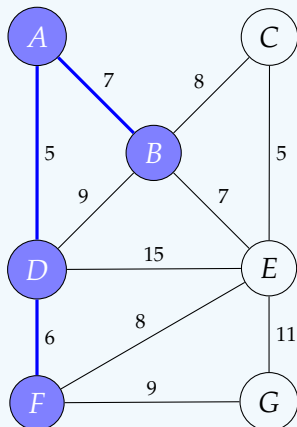
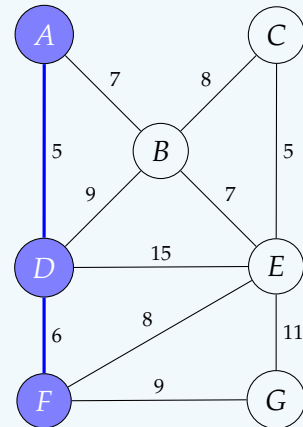
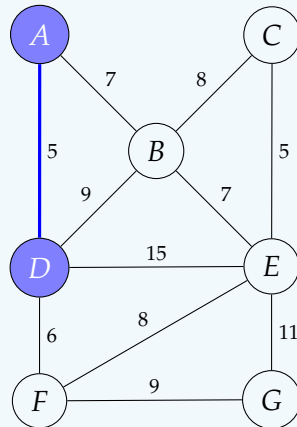
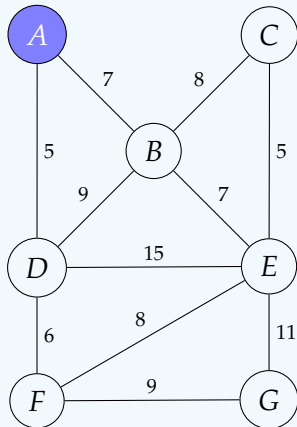
Ein *Spannbaum* verbindet alle Knoten eines ungerichteten Graphen miteinander, hat jedoch keine Kreise.

Der *minimale Spannbaum* ist der Spannbaum, dessen Kanten das kleinste summierte Gewicht haben.

## Algorithmus: Prim-Algorithmus

1. Wähle einen beliebigen Startknoten für den minimalen Spannbaum  $T$
2. Solange  $T$  noch nicht alle Knoten enthält:
  - Wähle eine Kante  $e$  mit minimalem Gewicht aus, die einen noch nicht in  $T$  enthaltenen Knoten  $v$  mit  $T$  verbindet.
  - Füge  $e$  und  $v$  dem Graphen  $T$  hinzu.

Start bei Knoten A.



### Definition: Shortest-Path-Probleme

Eigentlich: Suche nach *günstigsten Wegen* in gewichteten Graphen: Gewichte  $\simeq$  Kosten

Bei Anwendung auf ungewichtete Graphen ergibt sich: *kürzeste Wege*.

Beispiele:

- Single-Source-Shortest-Path
  - Dijkstra-Algorithmus (nicht-negative Kantengewichte)
  - Bellman-Ford-Algorithmus (keine negativen Zyklen)
- All-Pairs-Shortest-Path
  - Floyd-Warshall-Algorithmus
- One-Pair
  - A\*-Algorithmus

### Algorithmus: Dijkstra-Algorithmus

Gegeben: Graph  $G = (V, E)$ , dessen Bewertungsfunktion die Eigenschaften hat:

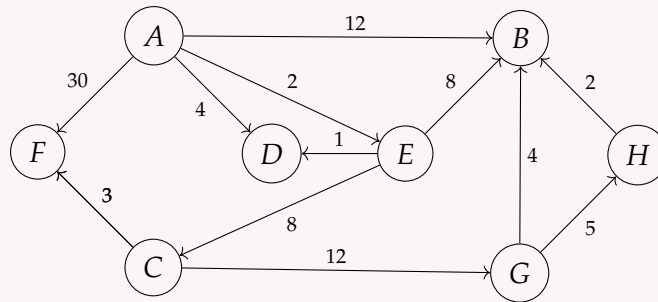
- Jede Kante von  $v_i$  nach  $v_j$  hat nicht-negative Kosten:  $C(i, j) \geq 0$
- Falls keine Kante zwischen  $v_i$  und  $v_j$ :  $C(i, j) = \infty$
- Diagonalelemente:  $C(i, i) = 0$

Menge  $S$ : die Knoten, deren günstigste Wegekosten von der vorgegebenen Quelle (Startknoten) bereits bekannt sind.

1. Initialisierung:  $S = \{\text{Startknoten}\}$
2. Beginnend mit Quelle alle ausgehenden Kanten betrachten (analog Breitensuche). Nachfolgerknoten  $v$  mit günstigster Kante zu  $S$  hinzunehmen.
3. Jetzt: Berechnen, ob die Knoten in  $V \setminus S$  günstiger über  $v$  als Zwischenweg erreichbar sind, als ohne Umweg über  $v$ .
4. Danach: Denjenigen Knoten  $v'$  zu  $S$  hinzunehmen, der nun am günstigsten zu erreichen ist. Bei zwei gleich günstigen Knoten wird ein beliebiger davon ausgewählt.
5. Ab Schritt 3 wiederholen, bis alle Knoten in  $S$  sind.

Zeitkomplexität (bei Speicherung des Graphen mit Adjazenzmatrix):  $\mathcal{O}(|V|^2)$

## Beispiel: Dijkstra-Algorithmus



$v_i$	$d[2]$	$d[3]$	$d[4]$	$d[5]$	$d[6]$	$d[7]$	$d[8]$	$p[2]$	$p[3]$	$p[4]$	$p[5]$	$p[6]$	$p[7]$	$p[8]$
	B	C	D	E	F	G	H	B	C	D	E	F	G	H
A	12		4	2	30			A		A	A	A		
E	10	10	3	2	30			E	E	E	A	A		
D	10	10	3	2	30			E	E	E	A	A		
B	10	10	3	2	30			E	E	E	A	A		
C	10	10	3	2	13	22		E	E	E	A	C	C	
F	10	10	3	2	13	22		E	E	E	A	C	C	
G	10	10	3	2	13	22	27	E	E	E	A	C	C	G
H	10	10	3	2	13	22	27	E	E	E	A	C	C	G

<https://youtu.be/4pBP2hbnGso> (Herleitung und Erklärung)

## Algorithmus: Bellman-Ford-Algorithmus

Gegeben: Graph  $G = (V, E)$ , dessen Bewertungsfunktion die Eigenschaften hat:

- Falls keine Kante zwischen  $v_i$  und  $v_j$ :  $C(i, j) = \infty$
- Diagonalelemente:  $C(i, i) = 0$

1. Initialisierung: Startknoten  $s$ , Distanz zu  $V \setminus \{s\}$  auf  $\infty$  setzen

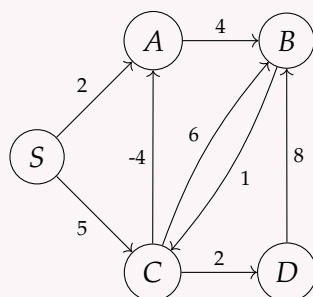
2. Für jede Kante  $(i, j)$ :

- Falls Distanz zu  $v_i$  bekannt:

Falls  $d(v_i) + C(i, j) < d(v_j)$ , setze  $d(v_j)$  auf  $d(v_i) + C(i, j)$

Zeitkomplexität:  $\mathcal{O}(|V| \cdot |E|)$

## Beispiel: Bellman-Ford-Algorithmus



	S	A	B	C	D
0	0	$\infty$	$\infty$	$\infty$	$\infty$
1	0	2	$\infty$	5	$\infty$
2	0	1	6	5	7
3	0	1	5	5	7
4	0	1	5	5	7

Keine Änderungen nach Schritt 3  $\implies$  Fertig!

## Algorithmus: Floyd-Algorithmus

Gegeben: Graph  $G = (V, E)$ , dessen Bewertungsfunktion die Eigenschaften hat:

- Jede Kante von  $v_i$  nach  $v_j$  hat nicht-negative Kosten:  $C(i, j) \geq 0$
- Falls keine Kante zwischen  $v_i$  und  $v_j$ :  $C(i, j) = \infty$
- Diagonalelemente:  $C(i, i) = 0$

Im Kontrast zum Dijkstra-Algorithmus bestimmt der *Floyd-Algorithmus* für alle geordneten Paare  $(v, w)$  den kürzesten Weg von  $v$  nach  $w$ .

$|V|$  Iterationen:

1. Vergleiche Kosten von
  - direkter Verbindung von Knoten  $i$  zu Knoten  $j$
  - Umweg über Knoten 1 (also: von  $i$  nach 1 + von 1 nach  $j$ )
  - Falls Umweg günstiger: alten Weg durch Umweg ersetzen
2. Umwege über Knoten 2 betrachten.
- $k$ . Umwege über Knoten  $k$  betrachten.

Der Floyd-Algorithmus nutzt eine  $|V| \times |V|$ -Matrix, um die Kosten der günstigsten Wege zu speichern:

$A_k[i][j] :=$  minimale Kosten, um in Schritt  $k$  über irgendwelche der Knoten in  $V$  vom Knoten  $i$  zum Knoten  $j$  zu gelangen

- Initialisierung:  $A_0[i][j] = C(i, j)$
- $|V|$  Iterationen mit „dynamischer Programmierung“:  
Iterationsformel zur Aktualisierung von  $A[i][j]$ :

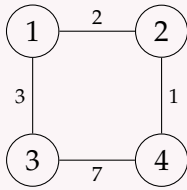
$$A_k[i][j] = \min\{A_{k-1}[i][j], A_{k-1}[i][k] + A_{k-1}[k][j]\}$$

Der Floyd-Algorithmus ist ein wichtiges Beispiel für dynamische Programmierung.

## Code: Floyd-Algorithmus

```
for(int i = 0; i < a.length; i++) {
    for(int j = 0; j < a.length; j++) {
        for(int k = 0; k < a.length; k++) {
            if(a[j][k] > a[j][i] + a[i][k]) {
                a[j][k] = a[j][i] + a[i][k];
            }
        }
    }
}
```

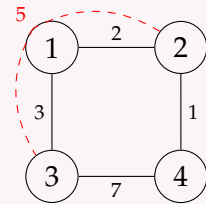
$A_0$  (direkter Weg):



	1	2	3	4
1	0	2	3	$\infty$
2	2	0	$\infty$	1
3	3	$\infty$	0	7
4	$\infty$	1	7	0

	1	2	3	4
1	-	-	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-

$A_1$  (Umweg über 1):



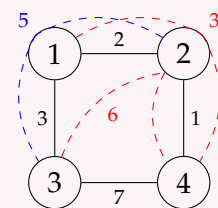
	1	2	3	4
1	0	2	3	$\infty$
2	2	0	5	1
3	3	5	0	7
4	$\infty$	1	7	0

	1	2	3	4
1	-	-	-	-
2	-	-	1	-
3	-	1	-	-
4	-	-	-	-

$$A_0[2][3] > A_0[3][1] + A_0[1][2] \implies A_1[2][3] = A_0[3][1] + A_0[1][2] = 3 + 2 = 5$$

$$A_0[3][2] > A_0[2][1] + A_0[1][3] \implies A_1[3][2] = A_0[2][1] + A_0[1][3] = 2 + 3 = 5$$

$A_2$  (Umweg über 2):



	1	2	3	4
1	0	2	3	3
2	2	0	5	1
3	3	5	0	6
4	3	1	6	0

	1	2	3	4
1	-	-	-	2
2	-	-	1	-
3	-	1	-	2
4	2	-	2	-

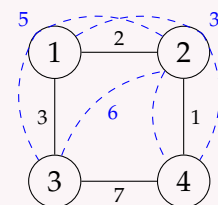
$$A_1[1][4] > A_1[1][2] + A_1[2][4] \implies A_2[1][4] = A_1[1][2] + A_1[2][4] = 2 + 1 = 3$$

$$A_1[4][1] > A_1[4][2] + A_1[2][1] \implies A_2[3][2] = A_1[4][2] + A_1[2][1] = 1 + 2 = 3$$

$$A_1[3][4] > A_1[3][2] + A_1[2][4] \implies A_2[3][4] = A_1[3][2] + A_1[2][4] = 5 + 1 = 6$$

$$A_1[4][3] > A_1[4][2] + A_1[2][3] \implies A_2[4][3] = A_1[4][2] + A_1[2][3] = 1 + 5 = 6$$

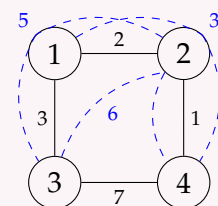
$A_3$  (Umweg über 3):



	1	2	3	4
1	0	2	3	3
2	2	0	5	1
3	3	5	0	6
4	3	1	6	0

	1	2	3	4
1	-	-	-	2
2	-	-	1	-
3	-	1	-	2
4	2	-	2	-

$A_4$  (Umweg über 4):



	1	2	3	4
1	0	2	3	3
2	2	0	5	1
3	3	5	0	6
4	3	1	6	0

	1	2	3	4
1	-	-	-	2
2	-	-	1	-
3	-	1	-	2
4	2	-	2	-

### Algorithmus: Warshall-Algorithmus

Gegeben: Graph  $G = (V, E)$ , egal ob gerichtet oder nicht.

Aufgabenstellung: Prüfe für alle geordneten Paare  $(v_i, v_j)$ , ob ein Weg von  $v_i$  nach  $v_j$  existiert.

Ziel: Berechne Adjazenzmatrix  $A$  zu *transitivem Abschluss* von  $G$ :

$A[i][j] = \text{true} \iff$  es existiert ein nicht-trivialer Weg (Länge  $> 0$ ) von  $v_i$  nach  $v_j$

Iterationsformel zur Aktualisierung von  $A[i][j]$ :

$$A_k[i][j] = A_{k-1}[i][j] \vee (A_{k-1}[i][k] \wedge A_{k-1}[k][j])$$

Der Algorithmus funktioniert also analog zum Floyd-Algorithmus.

### Code: Warshall-Algorithmus

```
for(int i = 0; i < a.length; i++) {  
    for(int j = 0; j < a.length; j++) {  
        for(int k = 0; k < a.length; k++) {  
            a[j][k] = a[j][k] || (a[j][i] && a[i][k]);  
        }  
    }  
}
```

## 5 Formale Sprachen

### 5.1 Textsuche

#### Bonus: Textsucheverfahren

- Naiver, grober, oder brute-force-Algorithmus  $\in \mathcal{O}(n \cdot m)$ 
  - für kleine Texte am schnellsten
- Knuth-Morris-Pratt, Rabin-Karp  $\in \Theta(n + m)$
- Boyer-Moore, (Boyer-Moore)-Sunday, (Boyer-Moore)-Horsepool  $\in \mathcal{O}(n + m)$ 
  - für große Texte am schnellsten

#### Algorithmus: Naive Textsuche

Bei der *naiven Textsuche* wird an allen Positionen  $i$  des Textes nach dem Muster geprüft.

Die möglichen Positionen reichen von  $i = 0$  (Muster linksbündig mit dem Text) bis  $i = n - m$  (Muster rechtsbündig mit dem Text).

Das Muster wird an der jeweiligen Position zeichenweise von links nach rechts mit dem Text verglichen.

Bei einem *Mismatch* oder bei vollständiger Übereinstimmung (*Match*) wird das Muster um eine Position weitergeschoben und an dieser Position verglichen.

#### Beispiel: Naive Textsuche

**Aufgabe:** Finde das Muster SINN im Text DASISTSINNLOSERTEXT mithilfe naiver Textsuche.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
D	A	S	I	S	T	S	I	N	N	L	O	S	E	R	T	E	X	T
S	I	N	N															
	S	I	N	N														
		S	I	N	N													
			S	I	N	N												
				S	I	N	N											
					S	I	N	N										
						S	I	N	N									
							S	I	N	N								
								S	I	N	N							

$i=0; j=0 \Rightarrow \text{Mismatch!}$

$i=1; j=0 \Rightarrow \text{Mismatch!}$

$i=2; j=2 \Rightarrow \text{Mismatch!}$

$i=3; j=0 \Rightarrow \text{Mismatch!}$

$i=4; j=1 \Rightarrow \text{Mismatch!}$

$i=5; j=0 \Rightarrow \text{Mismatch!}$

$i=6; j=4 \Rightarrow \text{Match!}$



## Algorithmus: Knuth-Morris-Pratt-Algorithmus

Der *Knuth-Morris-Pratt-Algorithmus* baut auf der naiven Textsuche auf.

Der Unterschied liegt darin, dass bei einem Mismatch das Muster nicht nur um eine Position verschoben werden kann, sondern um mehrere gleichzeitig. Dabei gilt:

- **Fall 1:** Wenn die letzten überprüften Buchstaben gleich dem Anfang des Patterns sind, verschiebt man das Muster entsprechend und macht beim anschließenden Zeichen weiter.
- **Fall 2:** Wenn die letzten überprüften Buchstaben nicht gleich dem Anfang des Musters sind, verschiebt man das Muster so, dass das erste Zeichen auf dem Mismatch zu liegen kommt.

### Beispiel: Knuth-Morris-Pratt-Algorithmus

**Aufgabe:** Finde das Muster UNGLEICHUNGEN im Text UNGLEICHUNGSTEIL... mithilfe des Knuth-Morris-Pratt-Algorithmus.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
U	N	G	L	E	I	C	H	U	N	G	S	T	E	I	L	...
U	N	G	L	E	I	C	H	U	N	G	E	N	⇒ Fall 1			
								U	N	G	L	E	I	C	H	...

usw.

**Aufgabe:** Finde das Muster UNGLEICHER im Text UNGLEICHUNGSTEIL... mithilfe des Knuth-Morris-Pratt-Algorithmus.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
U	N	G	L	E	I	C	H	U	N	G	S	T	E	I	L	...
U	N	G	L	E	I	C	H	E	R	⇒ Fall 2						
								U	N	G	L	E	I	C	H	...

usw.

## Algorithmus: Boyer-Moore-Sunday-Algorithmus

Beim *Boyer-Moore-Sunday-Algorithmus* wird nach einem Mismatch das Zeichen betrachtet, das hinter dem Muster liegt.

Dabei kann das Muster so weit nach vorne geschoben werden, bis ein Buchstabe des Musters mit diesem Buchstaben übereinstimmt.<sup>a</sup>

Kommt der folgende Buchstabe im Muster nicht vor, wird das Muster über den Buchstaben hinweggeschoben, da alle Positionen vorher sowieso zwecklos sind.

<sup>a</sup>Taucht der Buchstabe mehrfach im Wort auf, wird um die geringste Distanz nach vorne geschoben - also an den Buchstaben, der im Muster am weitesten hinten liegt.

## Beispiel: Boyer-Moore-Sunday-Algorithmus

**Aufgabe:** Finde das Muster ACBABCBA im Text AABBACCBACDACBABCBA mithilfe des Boyer-Moore-Sunday-Algorithmus.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	A	B	B	A	C	C	B	A	C	D	A	C	B	A	B	C	B	A
A	C	B	A	B	C	B	A											
	A	C	B	A	B	C	B	A										
		A	C	B	A	B	C	B	A									
			A	C	B	A	B	C	B	A								
				A	C	B	A	B	C	B	A							
					A	C	B	A	B	C	B	A						
						A	C	B	A	B	C	B	A					
							A	C	B	A	B	C	B	A				
								A	C	B	A	B	C	B	A			
									A	C	B	A	B	C	B	A		
										A	C	B	A	B	C	B	A	
											A	C	B	A	B	C	B	A

### Definition: last-Tabelle

Die *last-Tabelle* enthält zu jedem Zeichen des Zeichensatzes die Position des letzten Vorkommens im Muster (oder  $-1$ , falls es nicht vorkommt).

Die Implementierung erfolgt z.B. als Array indiziert mit (Unicode-)Zeichensatz:

- A auf Index 65
- B auf Index 66
- ...
- a auf Index 97
- b auf Index 98
- ...

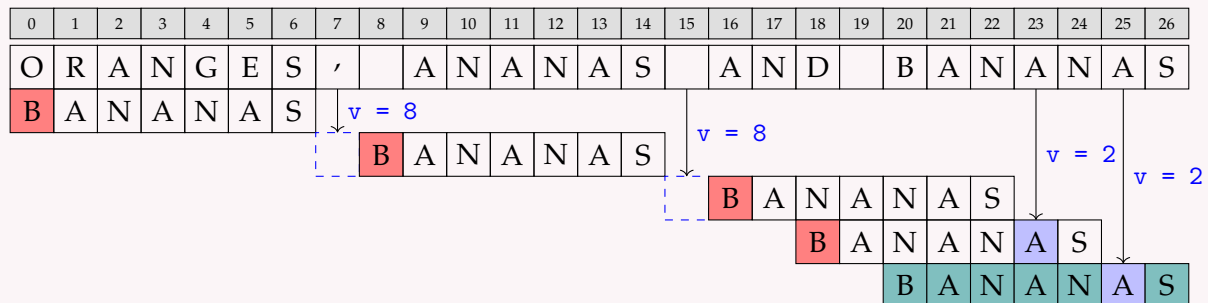
Dabei gilt:

- Alte Position des Patterns:  $i$  (das Pattern reicht bis  $i + m - 1$ )
- Verschiebedistanz:  $v = m - \text{last}[\text{text}[i+m]]$
- Neue Position des Patterns:  $i + v$  (das Pattern reicht bis  $i + v + m - 1$ )

### Beispiel: Boyer-Moore-Sunday-Algorithmus mit last-Tabelle

**Aufgabe:** Finde das Muster BANANAS im Text ORANGES, ANANAS AND BANANAS mithilfe des Boyer-Moore-Sunday-Algorithmus und einer last-Tabelle.

Pattern	B	A	N	S	sonst
last-Wert	0	5	4	6	-1



### Algorithmus: Rabin-Karp-Algorithmus

Der *Rabin-Karp-Algorithmus* funktioniert sehr ähnlich zur naiver Textsuche. Im Gegensatz wird hier aber der Hashwert des jeweiligen Textfensters mit dem Hashwert des Musters verglichen. Nur wenn beide Hashwerte gleich sind, werden die beiden zeichenweise verglichen. Sind die Hashwerte verschieden, rückt das Textfenster einen Schritt weiter nach rechts.

### Beispiel: Rabin-Karp-Algorithmus

**Aufgabe:** Finde das Muster ANE im Text BENANE mithilfe des Rabin-Karp-Algorithmus. Die Hashfunktion  $h(x)$  sei gegeben durch die jeweilige Position des Buchstaben  $x$  im Alphabet.

i	Textfenster	Hashwerte	Überprüfung?	Ergebnis
0	B E N A N E A N E	$h(BEN) = 21$ $h(ANE) = 20$	⚡	
1	B E N A N E A N E	$h(ENA) = 20$ $h(ANE) = 20$	✓	Mismatch
2	B E N A N E A N E	$h(NAN) = 29$ $h(ANE) = 20$	⚡	
3	B E N A N E A N E	$h(ANE) = 20$ $h(ANE) = 20$	✓	Match

## 5.2 Reguläre Ausdrücke

### Bonus: Regulärer Ausdruck (Begriffe)

- Zeichen
  - z.B. Buchstabe, Ziffer
- Alphabet
  - endliche Menge von Zeichen
  - z.B.  $\Sigma = \{a, b, c\}$
- Wort über Alphabet  $\Sigma$ 
  - endliche Folge von Zeichen aus  $\Sigma$
  - z.B.  $abcb$
  - Spezialfall: leeres Wort  $\varepsilon$
- $\Sigma^*$ 
  - Menge aller Wörter über  $\Sigma^*$
  - z.B.  $\Sigma = \{a, b\} \implies \Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
- Sprache  $L$  über Alphabet  $\Sigma$ 
  - Teilmenge  $L \subseteq \Sigma^*$

### Definition: Regulärer Ausdruck

Ein *regulärer Ausdruck* ist eine Formel, die eine Sprache beschreibt, d.h. eine Teilmenge aller möglichen Worte definiert.

### Definition: Verkettung

Bei der *Verkettung* (Concatenation) werden zwei oder mehrere Buchstaben durch diese Operation aneinandergehängt, z.B.  $ab$ .

Der Operator wird nicht mitgeschrieben.

### Definition: Alternative

Die *Alternative* erlaubt die Angabe alternativer Zeichen im Muster.

Schreibweise:

- $L((A \mid B)(A \mid B)) = \{AA, AB, BA, BB\}$
- $L((A \mid C)((B \mid C)D)) = \{ABD, CBD, ACD, CCD\}$
- $L(C(AC \mid B)D) = \{CACD, CBD\}$

### Definition: Hüllenbildung

Die *Hüllenbildung* (Closure) erlaubt es, Teile des Musters beliebig oft zu wiederholen.

Schreibweise: Hinter den zu wiederholenden Buchstaben wird ein Stern  $*$  gesetzt. Sind mehrere Buchstaben zu wiederholen, müssen sie in Klammern gesetzt werden.

- $L(A^*) = \{\varepsilon, A, AA, AAA, \dots\}$
- $L((ABC)^*) = \{\varepsilon, ABC, ABCABC, ABCABCABC, \dots\}$
- $L(DA^*B) = \{DB, DAB, DAAB, DAAAB, \dots\}$

### Definition: Perl Compatible Regular Expressions

*Perl Compatible Regular Expressions* (PCRE) ist eine Bibliothek zur Auswertung und Anwendung von regulären Ausdrücken.

Sie beinhaltet standardisierte Regeln zur Erzeugung regulärer Ausdrücke für die Textsuche.

Wichtige Regeln in PCRE sind:

Verknüpfungen	
AB	Zeichenfolge AB
A B	A oder B
[AB]	Zeichenklasse A oder B
Quantoren	
A{n}	A kommt genau n-mal vor
A{min,}	A kommt mindestens min-mal vor
A{min,max}	A kommt mindestens min und höchstens max-mal vor
Abkürzungen für Quantoren	
A?	entspricht A{0,1}
A*	entspricht A{0,}
A+	entspricht A{1,}
A	entspricht A{1}
Zeichenklassen	
\w	Buchstaben (word)
\d	Zahlen (digit)
.	Alles außer Zeilenvorschub
Referenzen	
()	Gruppierung
\x oder \$x	x-te Rückwärtsreferenz
Greedy	
(default)	Greedy
?	Reluctant, non-greedy

### Definition: Endlicher Automat

Ein *endlicher Automat* ist ein abstraktes Maschinenmodell.

Seine Aufgabe ist zu entscheiden, ob ein Wort zu einer Sprache gehört, die durch einen regulären Ausdruck beschrieben ist (Akzeptoren).

Am Anfang ist der Automat im *Anfangszustand*.


In jedem Schritt wird eine *Eingabesymbol*  $\sigma$  gelesen und abhängig von  $\sigma$  geht die Maschine von einem *Zustand* in einen bestimmten anderen über.

Wenn nach Leses des letzten Zeichens ein *Endzustand* erreicht ist, ist das Muster *erkannt* und gehört damit zur Sprache.

### Definition: Konstruktionsverfahren nach Kleene

Das *Konstruktionsverfahren nach Kleene* erzeugt aus einer gegebenen regulären Sprache einen *nichtdeterministischen endlichen Automaten* (NEA).<sup>a</sup>

Die Grundlage ist das Zustands-Übergangs-Diagramm mit folgenden Elementen:

start  $\rightarrow$   Anfangszustand

 (Zwischen-)Zustand

 Endzustand

$\xrightarrow{a}$  Zustandsübergang bei einem gegebenen Symbol

$\xrightarrow{\varepsilon}$   $\varepsilon$ -Übergang

---

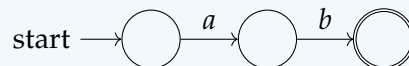
<sup>a</sup>Es existiert zu jedem NEA ein DEA.

### Algorithmus: Konstruktion eines Automaten nach Kleene

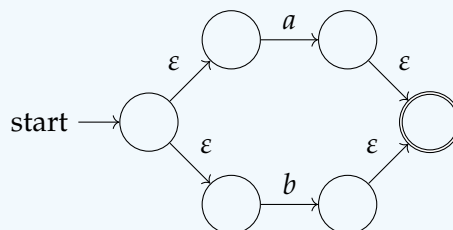
Einzelnes Symbol: Regulärer Ausdruck  $a$



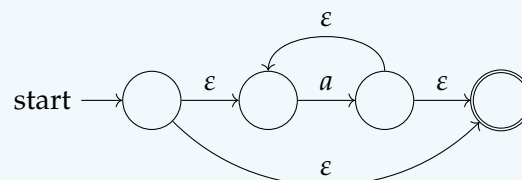
Verkettung: Regulärer Ausdruck  $ab$



Alternative: Regulärer Ausdruck  $a|b$

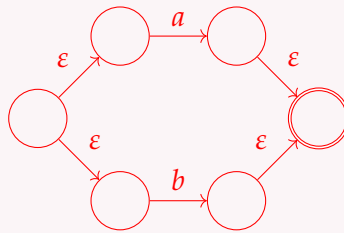


Hüllenbildung: Regulärer Ausdruck  $a^*$

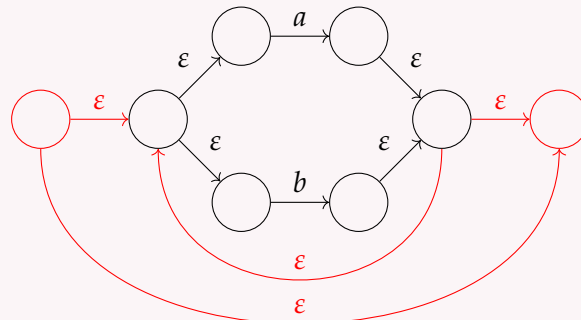


Aufbau des Automaten für den regulären Ausdruck  $a|a(a|b)^*a$

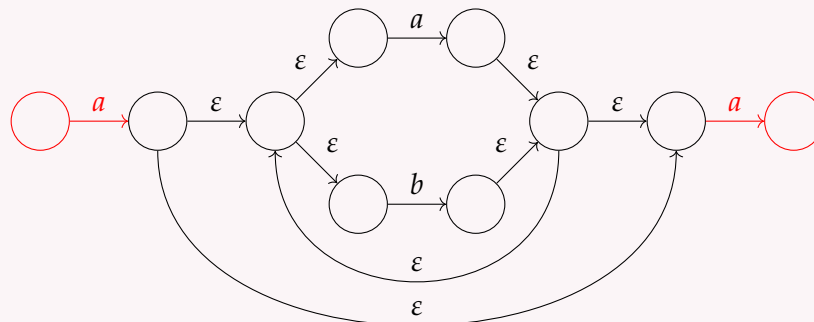
1. Schritt:  $a|b$



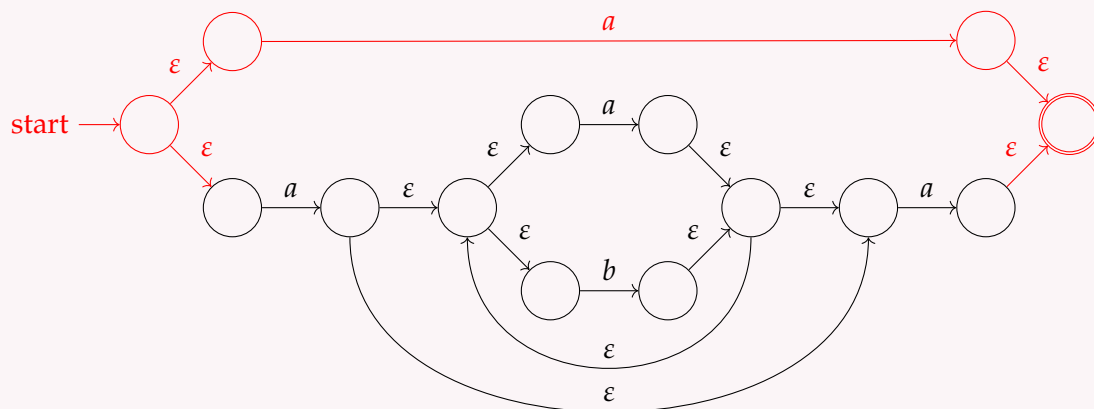
2. Schritt:  $(a|b)^*$



3. Schritt:  $a(a|b)^*a$



4. Schritt:  $a|(a|b)^*a$



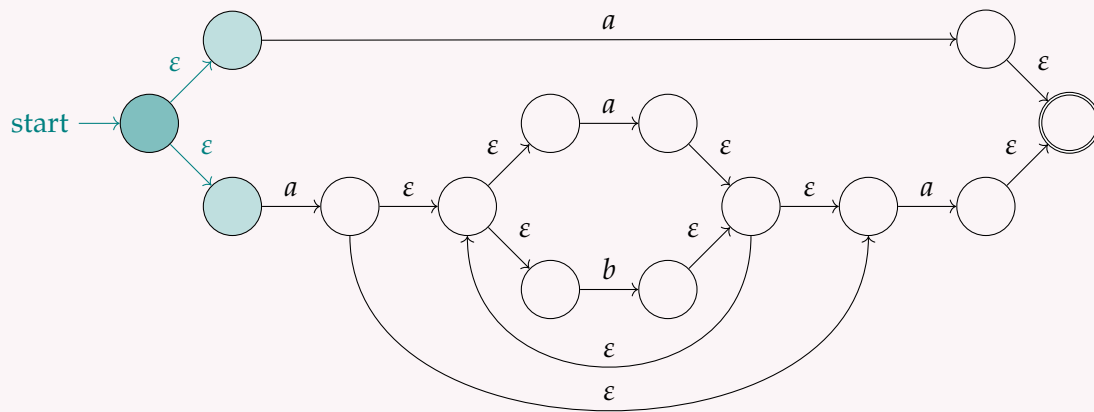
## Algorithmus: Ablaufregeln der Simulation nach Kleene

1. Initialisierung
  - a) Markiere den Anfangszustand
  - b) Markiere alle Zustände, die durch  $\epsilon$ -Übergänge erreichbar sind.
2. Für jedes gelesene Eingabesymbol
  - a) Markiere alle Zustände, die durch dieses Eingabesymbol erreichbar sind.
  - b) Lösche alle anderen Zustände.
  - c) Markiere alle Zustände, die jetzt durch  $\epsilon$ -Übergänge erreichbar sind.

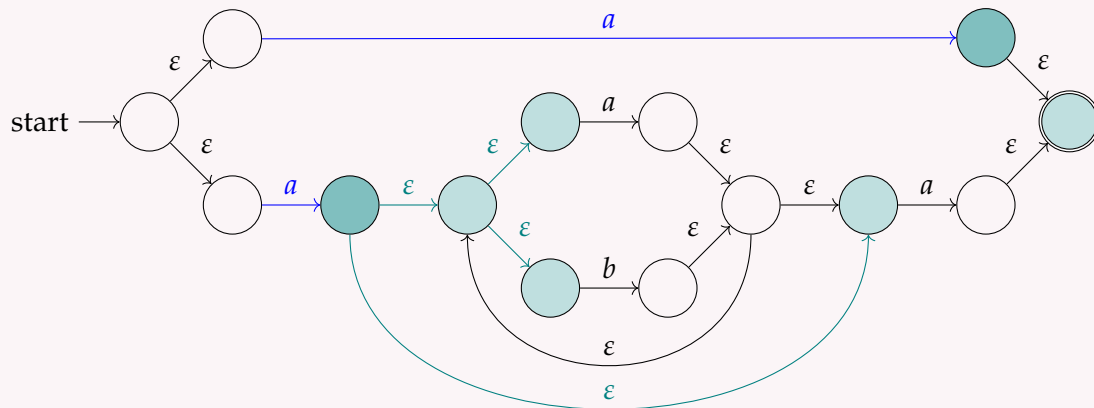
## Beispiel: Ablauf der Simulation nach Kleene

**Aufgabe:** Prüfen Sie, ob das Wort abba von dem Automaten der regulären Sprache  $a|a(a|b)^*a$  akzeptiert wird.

Anfangszustand:

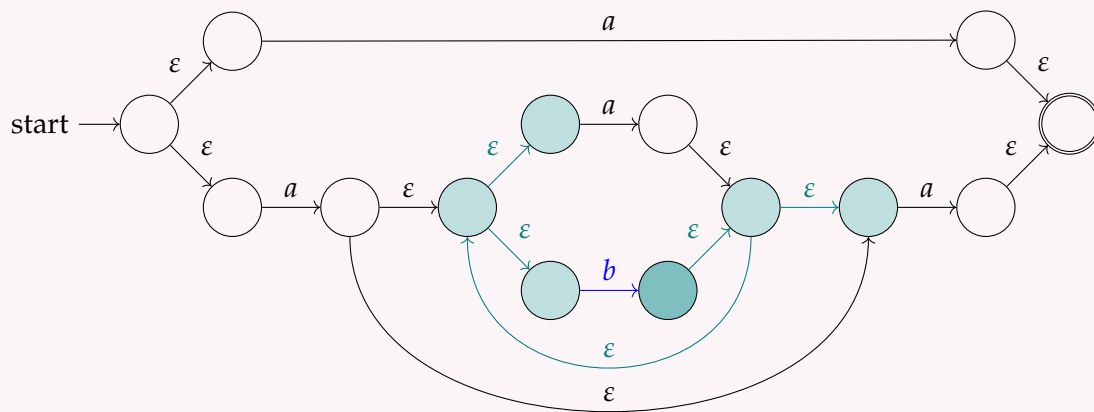


1. Buchstabe: a

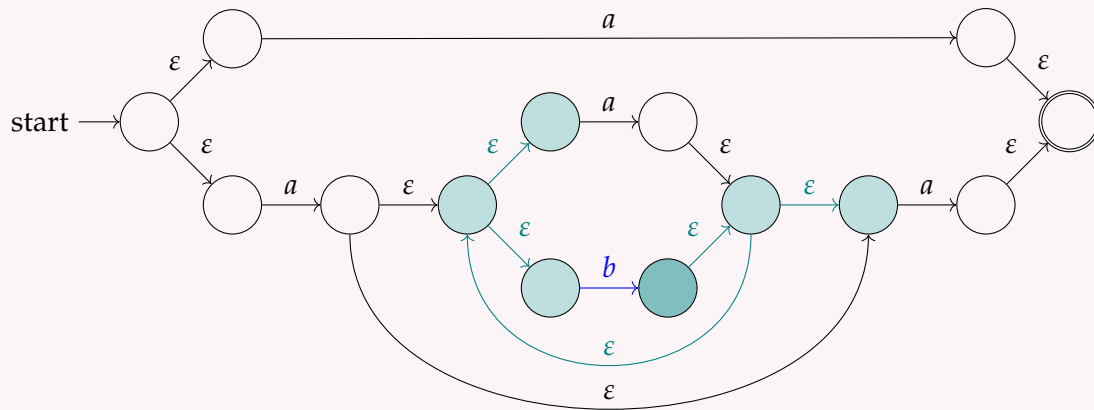




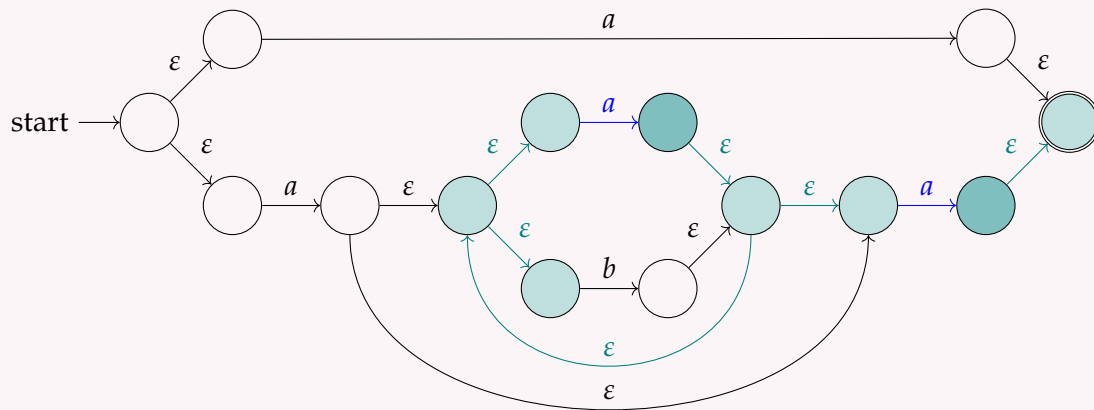
2. Buchstabe: b



3. Buchstabe: b



4. Buchstabe: a



Damit wird abba durch den regulären Ausdruck  $a|a(a|b)^*a$  beschrieben.

## 6 Sortierverfahren

### Definition: Klassifikationskriterien für Sortieralgorithmen

- Effizienz
  - Schlechter als  $\mathcal{O}(n^2)$ : nicht ganz ernst gemeinte Verfahren
  - $\mathcal{O}(n^2)$ : Elementare Sortierverfahren
    - \* Bubble-Sort
    - \* Insertion-Sort
    - \* Selection-Sort
  - Zwischen  $\mathcal{O}(n^2)$  und  $\mathcal{O}(n \log n)$ 
    - \* Shell-Sort
  - $\mathcal{O}(n \log n)$ : Höhere Sortierverfahren
    - \* Heap-Sort
    - \* Merge-Sort
    - \* Quick-Sort
  - Besser als  $\mathcal{O}(n \log n)$ : Spezialisierte Sortierverfahren
    - \* Radix-Sort
- Speicherverbrauch
- Intern vs. Extern
- Stabil vs. Instabil
- allgemein vs. spezialisiert

### Bonus: Beste Sortierverfahren

- normalerweise Quick-Sort
- Merge-Sort, falls:
  - die Datenmenge zu groß für den Hauptspeicher ist.
  - die Daten als verkettete Liste vorliegen.
  - ein stabiles Verfahren nötig ist.
- Insertion-Sort, falls:
  - wenige Elemente u sortieren sind.
  - die Daten schon vorsortiert sind.
- Radix-Sort, falls:
  - sich ein hoher Programmieraufwand für ein sehr schnelles Verfahren lohnt.

## 6.1 Elementare Sortierverfahren

### Algorithmus: Simple-Sort

Das Suchen und Vertauschen wird bei *Simple-Sort* wie folgt realisiert:

1. Gehe vom Element *i* aus nach rechts
2. Jedes Mal, wenn ein kleineres Element als das auf Position *i* auftaucht, vertausche es mit dem Element *i*
3. Wiederholen bis das Array sortiert ist

Simple-Sort ergibt einen besonders einfachen Code, ist aber langsamer als z.B. Selection-Sort. Dann zu nutzen, wenn:

- Sie keine Zeit oder Lust zum Nachdenken haben.
- die Felder so klein sind, dass der Algorithmus nicht effektiv sein muss.
- niemand sonst Ihren Code zu sehen bekommt.

### Code: Simple-Sort

```
public void simpleSort (int[] a) {  
    for(int i = 0; i < a.length; i++) {  
        for(int j = i+1; j < a.length; j++) {  
            if (a[i] > a[j]) {  
                swap(a, i, j);  
            }  
        }  
    }  
}
```

### Algorithmus: Selection-Sort

*Selection-Sort* funktioniert ähnlich wie *Simple-Sort*:

1. Suchen des kleinsten Elements im unsortierten Bereich
2. Vertauschen mit Position *i*
3. Wiederholen bis das Array sortiert ist

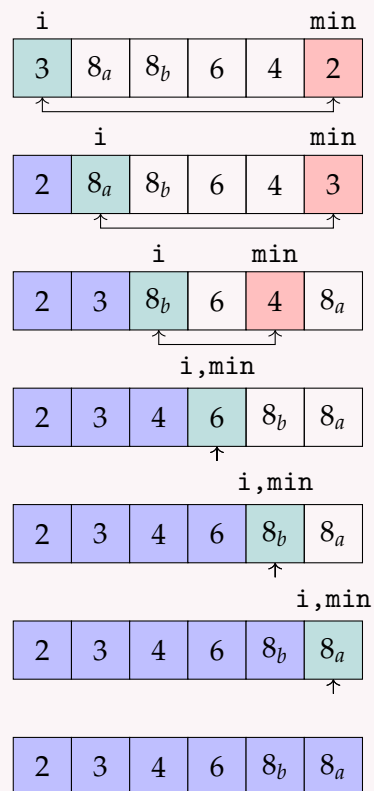
Selection-Sort ist eines der wichtigeren elementaren Verfahren, ist aber meistens etwas langsamer als Insertion-Sort.

Selection-Sort ist instabil und bietet keine Vorteile, wenn das Feld schon vorsortiert ist.

## Code: Selection-Sort

```
public void selectionSort (int[] a) {
    for(int i = 0; i < a.length; i++) {
        int small = i;
        for(int j = i+1; j < a.length; j++) {
            if(a[small] > a[j]) {
                small = j;
            }
        }
        swap(a, i, small);
    }
}
```

## Beispiel: Selection-Sort



## Algorithmus: Insertion-Sort

Die Grundidee ist:

1. Starte mit einem Wert im Array (meist Position 0)
2. Nimm jeweils den nächsten Eintrag im Array und füge ihn an der richtigen Stelle im sortierten Bereich ein
3. Wiederholen bis das Array sortiert ist

Insertion-Sort ist in den meisten Fällen der schnellste elementare Suchalgorithmus - auch, weil er potentielle Vorsortierung ausnutzt.

Insertion-Sort ist stabil.

## Code: Insertion-Sort

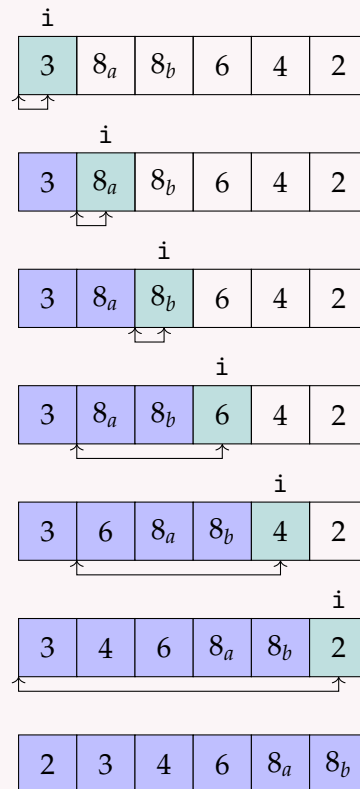
```
public void insertionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        int m = a[i];

        // fuer alle Elemente links vom aktuellen Element
        for(int j = i; j > 0; j--) {
            if (a[j-1] <= m) {
                break;
            }

            // groessere Elemente nach hinten schieben
            a[j] = a[j-1];
        }

        // m an freiem Platz einfuegen
        a[j] = m;
    }
}
```

## Beispiel: Insertion-Sort



## Bonus: Bewertung elementarer Sortierverfahren

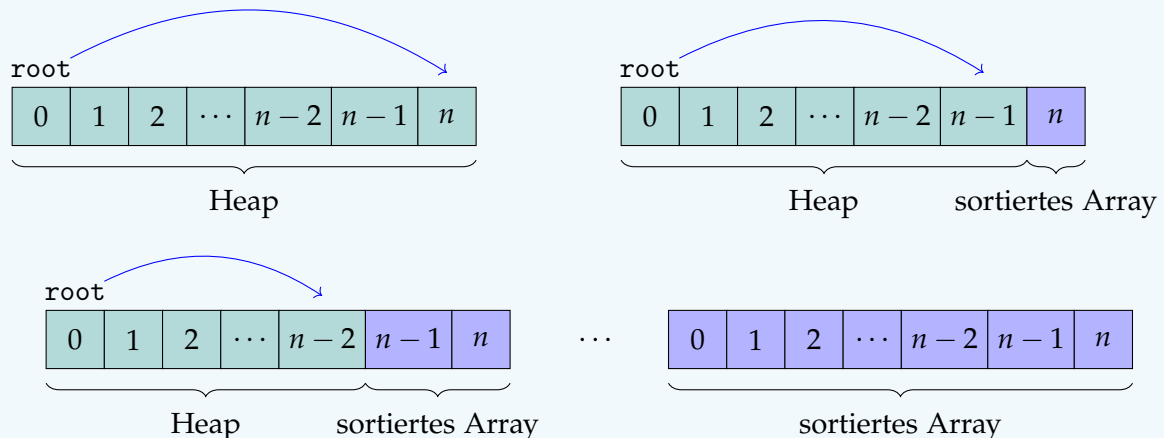
- Simple-Sort
  - Einfach zu implementieren
  - Langsam
- Selection-Sort
  - Aufwand unabhängig von Vorsortierung
  - nie mehr als  $\mathcal{O}(n)$  Vertauschungen nötig
- Bubble-Sort
  - Stabil
  - Vorsortierung wird ausgenutzt
- Insertion-Sort
  - Stabil
  - Vorsortierung wird ausgenutzt
  - schnell für ein elementares Suchverfahren ( $\mathcal{O}(n^2)$ )

## 6.2 Höhere Sortierverfahren

### Algorithmus: Heap-Sort

Der Heap ist Grundlage für das Sortierverfahren *Heap-Sort*.

- Zu Beginn: Unsortiertes Feld
- Phase 1:
  - Alle Elemente werden nacheinander in einen Heap eingefügt
  - Resultat ist ein Heap, der in ein Feld eingebettet ist
- Phase 2:
  - Die Elemente werden in absteigender Reihenfolge entfernt (Wurzel!)
  - Heap schrumpft immer weiter



### Algorithmus: Quick-Sort

*Quick-Sort* ist ein rekursiver Algorithmus, der nach dem Prinzip von „divide-and-conquer“ („teile und herrsche“) arbeitet:

1. Müssen 0 oder 1 Elemente sortiert werden: Rekursionsabbruch
2. Wähle ein Element als *Pivot-Element* aus
3. Teile das Feld in zwei Teile:<sup>a</sup>
  - Ein Teil mit den Elementen größer als das Pivot.
  - Ein Teil mit den Elementen kleiner als das Pivot.
4. Wiederhole rekursiv für beide Teilfelder.

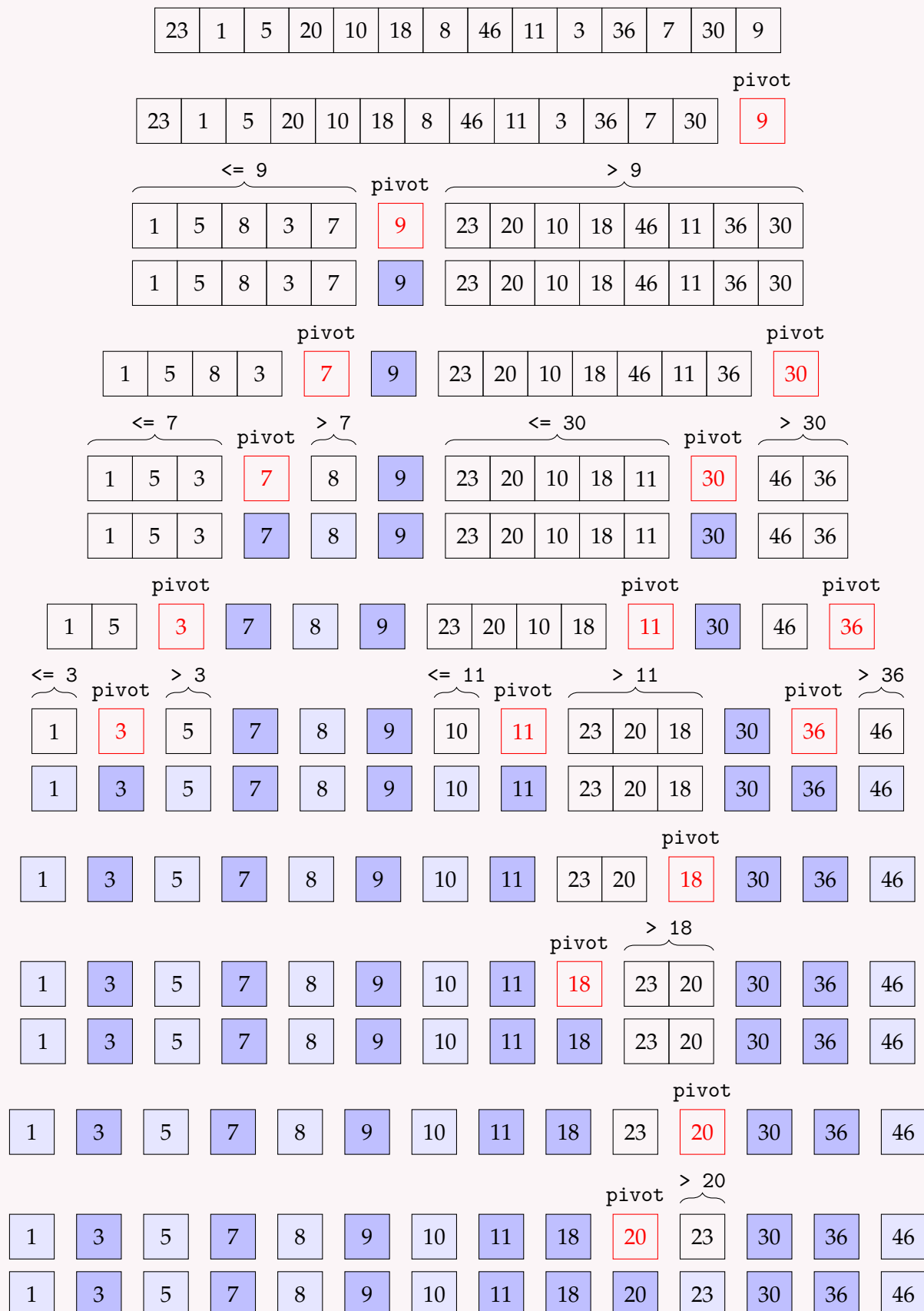
Die überwiegende Mehrheit der Programmbibliotheken benutzt Quick-Sort. In fast allen Fällen sind zwei Optimierungen eingebaut:

- „Median of three“
- „Behandlung kleiner Teilfelder“

<sup>a</sup>Beachte: Die Elemente werden *in-place* getauscht. Siehe Beispiel.

## Beispiel: Quick-Sort

ACHTUNG: Das Beispiel muss in Rücksprache mit Herrn Pflug noch angepasst werden. Der in der Vorlesung benutzte - und damit klausurrelevante - Quick-Sort nutzt ein *in-place*-Verfahren zum Aufteilen der Elemente. Das Beispiel unten (noch) nicht.





### Definition: Zeitkomplexität von Quick-Sort

- Best-Case:
  - Pivot-Wert ist immer Median der Teilliste  $\implies$  Teillisten werden stets halbiert
  - $\log n$  Stufen nötig mit  $n$  Elementen
  - $\mathcal{O}(n \log n)$
- Worst-Case:
  - Pivot-Wert ist immer größtes oder kleinstes Element der Teilliste
  - $n - 1$  Stufen nötig mit  $n - i$  Elementen pro Stufe
  - $\mathcal{O}(n^2)$
- Average-Case:
  - Berechnung ziemlich aufwändig
  - $\mathcal{O}(n \log n)$

### Bonus: Standard-Optimierung von Pivot-Elementen

Wählt man z.B. bei einem vorsortierten Array stets das letzte Element, hat man genau den Worst-Case.

*Median-of-three-Methode* zum Auswählen des Pivots:

- Es werden drei Elemente als Referenz-Elemente (Anfang, Mitte, Ende) gewählt, von denen dann der Median als Pivot verwendet wird.
- Kann auf mehr als drei Elemente ausgebaut werden.

### Bonus: Standard-Optimierung des Rekursionsabbruchs

Beim „einfachen“ Rekursionsabbruch (Teilliste enthält 0 oder 1 Elemente) sind die letzten Rekursionsdurchgänge nicht mehr effektiv.

Daher kann die Rekursion z.B. schon früher abgebrochen werden und die dann vorhandene Teilliste mit Insertion-Sort sortiert werden. Die Grenze dafür ist nicht klar festgelegt (Empfehlung von Knuth und Sedgewick liegt bei 9, im Internet aber oft zwischen 3 und 32).

### Definition: Dual-Pivot Quick-Sort

Der *Dual-Pivot Quick-Sort* funktioniert analog zum normalen Quick-Sort.

Hier werden jedoch zwei Pivot-Elemente verwendet, die dann die entsprechende Teilliste in drei Bereiche aufteilen.

### Definition: Internes und externes Sortieren

Bis jetzt wurde vorausgesetzt, dass schneller Zugriff auf einen beliebigen Datensatz (*wahlfreier Zugriff*) möglich ist. Das ist zumeist möglich, wenn der Datensatz im Hauptspeicher liegt. Ist dies der Fall, spricht man von *internem Sortieren*.

Bei sehr großen Datenbeständen ist das aber oft nicht mehr möglich, da sie z.B. auf Hintergrundspeicher ausgelagert werden müssen. Hier hat man dann lediglich *sequentiellen Zugriff*<sup>a</sup> und man spricht von *externem Sortieren*.

<sup>a</sup>das Gleiche gilt auch für verkettete Listen

## Algorithmus: Merge-Sort

*Merge-Sort* kann sowohl iterativ als auch rekursiv implementiert werden, wobei die rekursive Variante etwas schneller ist.

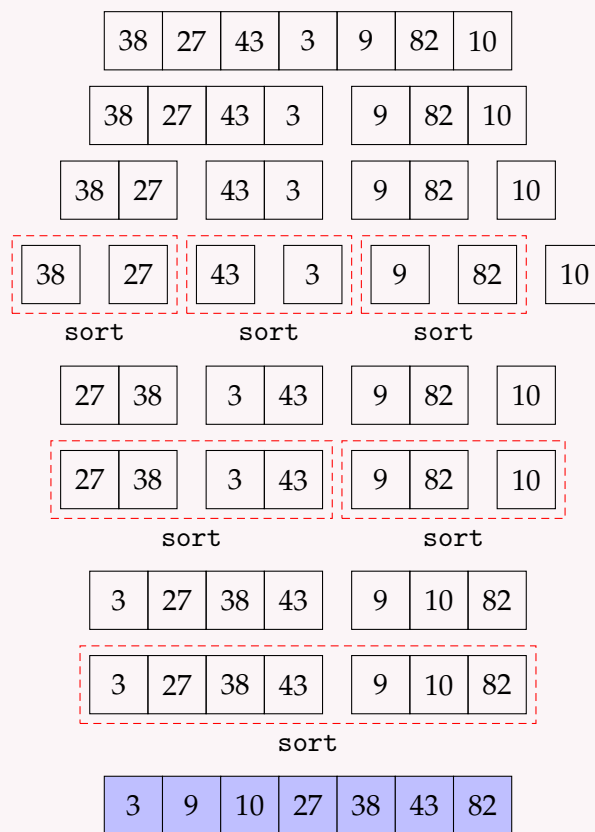
Der rekursive Ablauf ist:

1. Teile die Daten in zwei Hälften
2. Unterteilung wird fortgesetzt, bis nur noch ein Element in einer Menge vorhanden ist
3. Teilstücke werden für sich sortiert
4. sortierte Teilstücke werden zusammengeführt

Merge-Sort ist stabil.

Die Komplexität von Merge-Sort liegt in  $\mathcal{O}(n \log n)$ .

### Beispiel: Merge-Sort (rekursiv)



#### Definition: Optimierung von Merge-Sort

- Felduntergrenze
  - Feld wird nicht in Einzelelemente geteilt, sondern in Gruppen zu  $n$  Elementen, die dann mit Insertion-Sort sortiert werden
  - gcj-Java nimmt 6 Elemente als Grenze
  - Python nimmt 64 Elemente als Grenze
- Trivialfall
  - ist das kleinste Element der einen Teilfolge größer als das größte Element der anderen Teilfolge  $\implies$  Zusammenfügen beschränkt sich auf Hintereinandersetzen der beiden Teilfolgen
  - genutzt von z.B. Java
  - Vorsortierung wird ausgenutzt
- Natürlicher Merge-Sort
  - weitergehende Ausnutzung der Vorsortierung
  - jede Zahlenfolge besteht aus Teilstücken, die abwechselnd monoton steigend und monoton fallend sind
  - Idee ist, diese bereits sortierten Teilstücke als Ausgangsbasis des Merge-Sorts zu nehmen
  - bei nahezu sortierten Feldern werden die Teilstücke sehr groß und as Verfahren sehr schnell
  - Best-Case in  $\mathcal{O}(n)$

### 6.3 Spezialisierte Sortierverfahren

#### Definition: Radix-Sort

*Radix-Sort* heißt eine Gruppe von Sortierverfahren mit folgenden Eigenschaften:

- Sortiervorgang erfolgt mehrstufig
- zunächst wird Grobsortierung vorgenommen, zu der nur ein Teil des Schlüssels (z.B. erstes Zeichen) herangezogen wird
- grob sortierte Bereiche dann feinsortiert, wobei schrittweise der restliche Teil des Schlüssels verwendet wird

#### Bonus: MSD-Radix-Sort

- MSD = most significant digits
- In der 1. Stufe wird die erste Ziffer betrachtet.
- Benutzt Bucket-Sort mit Teillisten.
- Bucket-Sort wird für die einzelnen Buckets weiter genutzt, bis alle Werte sortiert sind

#### Bonus: LSD-Radix-Sort

- LSD = least significant digits
- In der 1. Stufe wird die letzte Ziffer betrachtet.
- Benutzt Bucket-Sort mit schlüsselindiziertem Zählen.

## Algorithmus: Bucket-Sort

*Bucket-Sort* sortiert für bestimmte Werte-Verteilungen einen Datensatz in linearer Zeit.

Der Algorithmus ist in drei Phasen eingeteilt:

1. Verteilung der Elemente auf die Buckets
2. Jeder Bucket wird mit einem weiteren Sortiertverfahren sortiert .
3. Der Inhalt der sortierten Buckets wird konkateniert.

Zu viele oder zu wenige Eimer können zu großen Laufzeitproblemen führen. Empfohlen werden:

- Sedgewick: Für 64-bit Schlüssel (long)  $2^{16}$  Eimer
- Linux-related: Für 32-bit Schlüssel (int)  $2^{11}$  Eimer

Der Einsatz von Bucket-Sort lohnt sich nur, wenn die Anzahl der zu sortierenden Werte deutlich größer ist, als die Anzahl der Eimer.

## Index

- last-Tabelle, 50
- Ablaufregeln der Simulation nach Kleene, 55
- Abstrakte Datentypen (ADTs), 4
- Adjazenz, 33
- ADTs in Java, 4
- Alternative, 52
- Array, 5
- Assoziatives Feld, 6
- Ausgleich zwischen Geschwisterknoten, 24
- AVL-Baum, 18
- 
- B+-Baum, 25
- B-Baum, 22
- Backtracking, 39
- Balance-Kriterien, 17
- Balanciertheit, 17
- Baum, 13
- Bellman-Ford-Algorithmus, 44
- Beste Sortiervverfahren, 58
- Bewertung elementarer Sortiervverfahren, 62
- Binärbaum, 13
- Binärer Suchbaum, 14
- Bipartiter Graph, 40
- Boyer-Moore-Sunday-Algorithmus, 49
- Branch and Bound, 40
- Breitensuche, 36
- Bucket-Sort, 67
- 
- Deque („Double ended queue“), 5
- Dijkstra-Algorithmus, 43
- Doppelrotation, 20
- Doppelt verkettete Liste, 6
- Down-Heapify (Löschen), 30
- Dual-Pivot Quick-Sort, 65
- Dynamisches Feld, 7
- Dynamisches Hashing, 11
- 
- Eigenschaften eines Algorithmus, 2
- Einfach verkettete Liste, 6
- Einfachrotation, 19
- Einfügen im binären Suchbaum, 14
- Einfügen in einem AVL-Baum, 18
- Einfügen in einem B-Baum der Ordnung  $d$ , 23
- Einfügen in einem Heap, 29
- Einfügen in einen Rot-Schwarz-Baum, 26
- Einfügen in einen Rot-Schwarz-Baum (Fall 1), 26
- Einfügen in einen Rot-Schwarz-Baum (Fall 2), 27
- Endlicher Automat, 53
- 
- Floyd-Algorithmus, 44, 45
- Füllgrad, 11
- 
- Gerichteter Graph, 32
- Gewichteter Graph, 32
- Greedy, 39
- 
- Hashfunktion, 9
- Heap, 28
- Heap (Wortbedeutungen), 28
- Heap-Sort, 63
- Heapify, 29
- Heterogene Datenstruktur, 4
- Homogene Datenstruktur, 4
- Hüllenbildung, 52
- 
- In-Order-Baumdurchlauf, 38
- Insertion-Sort, 60, 61
- Internes und externes Sortieren, 65
- 
- Klassifikationskriterien für  
    Sortieralgorithmen, 58
- Knuth-Morris-Pratt-Algorithmus, 48
- Kollision, 9
- Kollisionsbehandlung, 10
- Komplexität beim Suchen, Löschen und  
    Einfügen in Binärbäumen, 16
- Komplexität von AVL-Bäumen, 21
- Konstruktion eines Automaten nach Kleene, 54
- Konstruktionsverfahren nach Kleene, 53
- 
- Landau-Notation, 2
- Level-Order-Baumdurchlauf, 37
- Liste, 5
- LSD-Radix-Sort, 67
- Löschen im binären Suchbaum (Blatt), 15
- Löschen im binären Suchbaum (Innerer  
    Knoten mit einem Kind), 15
- Löschen im binären Suchbaum (Innerer  
    Knoten mit zwei Kindern), 16

Löschen in einem AVL-Baum, 19  
 Löschen in einem B-Baum der Ordnung  $d$   
 (Blatt), 23  
 Löschen in einem B-Baum der Ordnung  $d$   
 (Innerer Knoten), 24  
 Löschen in einem Heap, 29  
 Löschen in einem Rot-Schwarz-Baum, 28  
 Lösung mit minimalem Spannbaum, 39  
  
 Menge, 5  
 Merge-Sort, 65  
 MSD-Radix-Sort, 67  
  
 Naive Textsuche, 48  
  
 Offene Adressierung (Sondieren), 11  
 Optimierung von Merge-Sort, 66  
  
 Perl Compatible Regular Expressions, 52  
 Post-Order-Baumdurchlauf, 38  
 Pre-Order-Baumdurchlauf, 38  
 Prim-Algorithmus, 41  
 Primäre und sekundäre Häufung, 12  
 Prioritätswarteschlange, 5  
 Prüfung ob ein Graph bipartit ist, 40  
  
 Queue, 5  
 Quick-Sort, 63  
  
 Rabin-Karp-Algorithmus, 51  
 Radix-Sort, 67  
 Random Insertion, 39  
 Rebalancierung, 19  
 Regulärer Ausdruck, 52  
 Regulärer Ausdruck (Begriffe), 52

Rot-Schwarz-Baum, 26  
  
 Schrittzahl, 10  
 Selection-Sort, 59  
 Shortest-Path-Probleme, 42  
 Simple-Sort, 59  
 Spannbaum, 41  
 Speicherung von Graphen, 33  
 Stack, 5  
 Standard-Optimierung des  
 Rekursionsabbruchs, 65  
 Standard-Optimierung von Pivot-Elementen,  
 65  
 Suchen im binären Suchbaum, 14  
 Suchen in einem B-Baum, 22  
  
 Teilgraph, 32  
 Textsucheverfahren, 48  
 Tiefensuche, 36  
 Traveling-Salesman-Problem, 39  
  
 Underflow, 24  
 Ungerichteter Graph, 32  
 Up-Heapify (Einfügen), 30  
  
 Verkettung, 52  
 Verschmelzen von Geschwisterknoten, 25  
 Visualisierung Komplexitätsklassen, 2  
  
 Warshall-Algorithmus, 46, 47  
 Weg, 32  
  
 Zeitkomplexität von Quick-Sort, 64  
 Zirkuläres (dynamisches) Feld, 7

## Beispiele

Ablauf der Simulation nach Kleene, 56  
Ablauf der Simulation nach Kleene  
(Fortsetzung), 56  
Adjazenzliste, 35  
Adjazenzmatrix, 34  
  
Baumdurchläufe, 38  
Bellman-Ford-Algorithmus, 44  
Boyer-Moore-Sunday-Algorithmus, 49  
Boyer-Moore-Sunday-Algorithmus mit  
last-Tabelle, 50  
  
Dijkstra-Algorithmus, 43  
Divisions-Hash, 9  
  
Floyd-Algorithmus, 45  
  
Hashfunktionen für verschiedene Datentypen,  
9  
Heap als Array, 29  
Insertion-Sort, 61  
  
Kantenliste, 33  
Knotenliste, 34  
Knuth-Morris-Pratt-Algorithmus, 49  
Konstruktion eines Automaten nach Kleene,  
54  
  
Landau-Notation, 2  
Linksvollständiger Binärbaum, 14  
  
Merge-Sort (rekursiv), 66  
  
Naive Textsuche, 48  
  
Quick-Sort, 63  
  
Rabin-Karp-Algorithmus, 51  
  
Schrittzahl beim Suchen in Teillisten, 11  
Selection-Sort, 60  
  
Vollständiger Binärbaum, 14

## Literatur

- [1] Wikipedia. *Landau-Symbole* — *Wikipedia, The Free Encyclopedia*. <http://de.wikipedia.org/w/index.php?title=Landau-Symbole&oldid=212674123>. [Online; accessed 05-June-2021]. 2021.