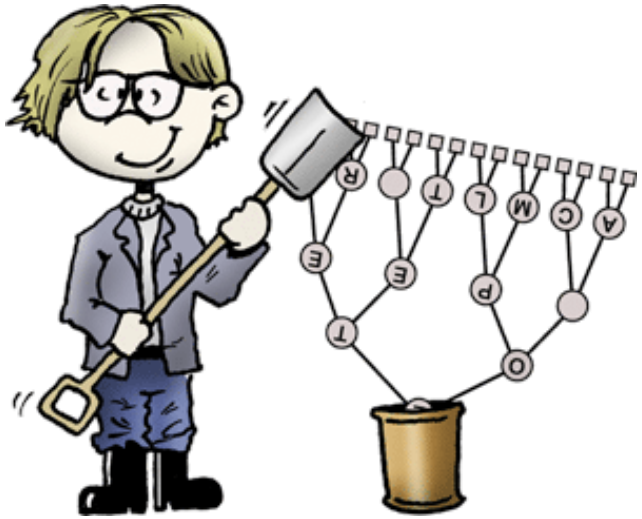
RWTH AACHEN UNIVERSITY

Algorithmen



Vorlesung im Rahmen des dualen Studiums „Scientific Programming“ (FH Aachen) / MATSE-Ausbildung

Prof. Dr. Hans Joachim Pflug

Inhalt des Kurses

1. Grundlegendes zu Algorithmen
2. Datenstrukturen
3. Algorithmen
4. Parallelisierung

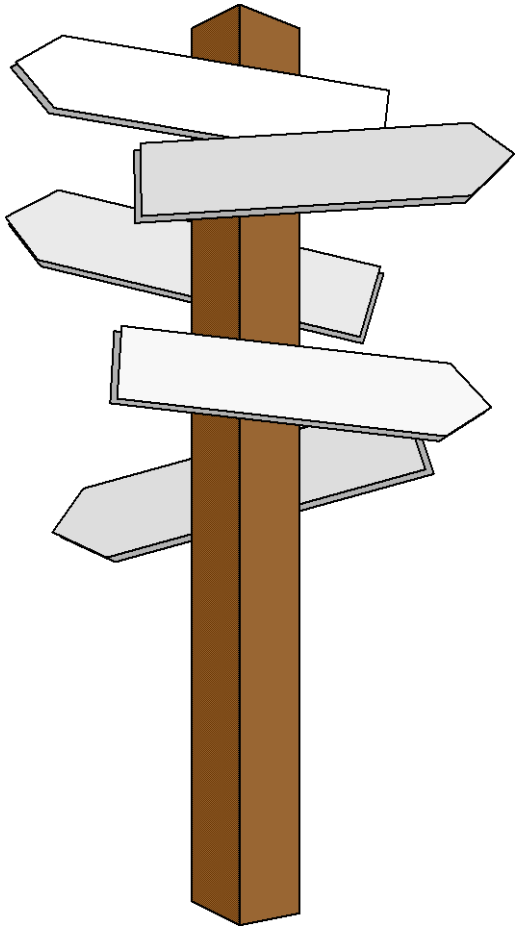
Literatur

- **Sedgewick, R.:**
  „Algorithmen in Java“
  816 Seiten
  Addison-Wesley, 3. Auflage 2003

- **Lang, H. W.:**
  „Algorithmen in Java “
  270 Seiten
  Oldenbourg, 2002

- **Wirth, N.:**
  „Algorithmen und Datenstrukturen. Pascal-Version“
  320 Seiten
  Teubner, 5. Auflage 2000

Kapitel 1: Grundlegendes zu Algorithmen



1.1 Algorithmusbegriff
1.2 Komplexitätsanalyse

Abu Ja‘far Mohammed ibn Musa *al-Khowarizmi*:

- Persischer Mathematiker, lebte um 820 in Bagdad
- Sein Name wurde in den lateinischen Übersetzungen des Mittelalters zu *Algorismi.*
- Der Name stammt eigentlich von seiner Geburtsstadt *Khowarizm*, dem heutigen *Chiwa* in Usbekistan. Die Stadt enthält zahlreiche Kulturdenkmäler und wurde 1990 komplett zum Weltkulturerbe erklärt.

Algorithmen im Alltag

- Bedienungsanleitung für Münztelefone
- Kochrezept
- Lösungsanleitung für den Zauberwürfel
- ...

Der Algorithmusbegriff

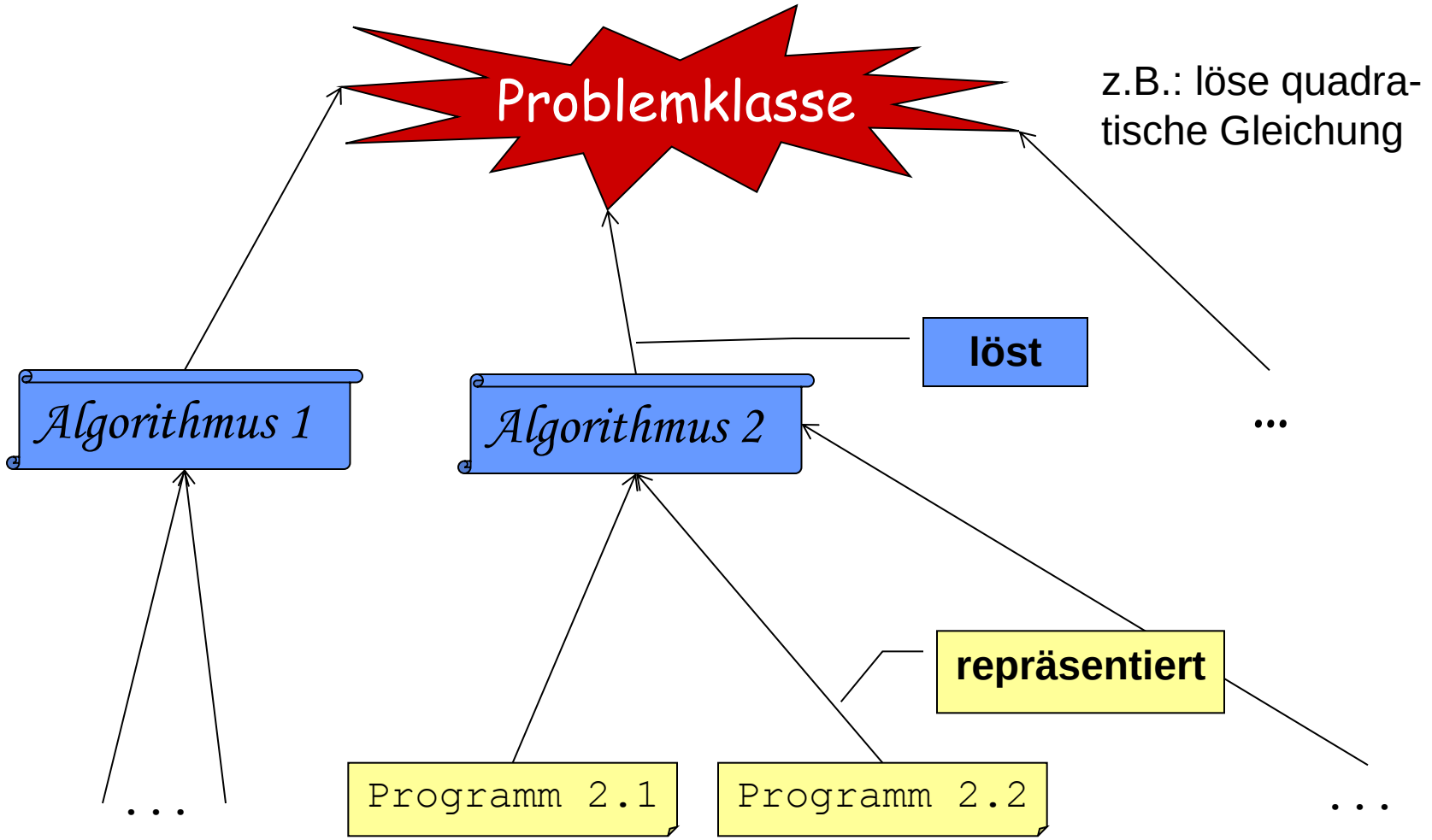
RWTH AACHEN UNIVERSITY

- Ein Algorithmus (in der EDV) ist
  - eine Lösungsvorschrift für eine Problemklasse
    - eine Problemklasse ist z.B.: „löse quadratische Gleichung“.
    - das konkrete Problem wird durch Eingabeparameter spezifiziert.
  - geeignet für Implementierung als Computerprogramm
  - endliche Folge von elementaren, ausführbaren **Instruktionen** (Verarbeitungsschritten)

- Beispiele für Instruktionen:

```
x = x + z;
if (a > 0) b = 1;
```

Problem – Algorithmus – Programm



Eigenschaften eines Algorithmus

Terminierung:

Ein Algorithmus heißt *terminierend*, wenn er (bei jeder erlaubten Eingabe) nach endlichen vielen Schritten abbricht.

Determiniertheit:

Festgelegtes Ergebnis: Bei vorgegebener Eingabe wird ein eindeutiges Ergebnis geliefert.

Determinismus:

Festgelegter Ablauf: Eindeutige Vorgabe der Abfolge der auszuführenden Schritte. ($\Rightarrow$ Determiniertheit)

Die meisten hier betrachteten Algorithmen sind deterministisch und terminierend. Sie definieren eine *Ein-/Ausgabefunktion*:

f : Eingabewerte →Ausgabewerte

Darstellung von Algorithmen

- Verbale Umschreibung (Handlungsanweisung)
- graphisch: Struktogramme, Flussdiagramme, ...
- Pseudo-Code
- Höhere Programmiersprache
- ...

- Churchsche These $\Rightarrow$ Alle Darstellungsformen von Algorithmen sind äquivalent.

1.2 Komplexitätsanalyse

- Unterschiedlichste Darstellungen für denselben Algorithmus; auch Programme sind Darstellungen ...

- Außerdem: Unterschiedliche Algorithmen für das gleiche Problem

$\Rightarrow$ Ziel: Algorithmen miteinander vergleichen (Komplexitätsanalyse)
  - Kriterien: Laufzeit und Speicherplatzbedarf
  - Komplexität wird ausgedrückt in Abhängigkeit von Menge und Größe der bearbeiteten Daten

- Komplexität von Algorithmus (bedingt) auf konkrete Implementierung übertragbar

Zusammenhang Algorithmus/Funktion

- Komplexität:
  Zur Berechnung erforderliche Aufwand an Betriebsmitteln (Speicherplatz, Rechenzeit, benötigte Geräte, usw.)
- Komplexität eines Algorithmus: erforderlicher Aufwand bei Realisierung des Algorithmus (innerhalb des Berechnungsmodells)

- Komplexität einer Funktion: Komplexität des bestmöglichen Algorithmus von allen Algorithmen, die die Funktion berechnen

- Sei A ein Algorithmus, der die Funktion f berechnet
  - Komplexität von A ≥ Komplexität von f
  - Komplexität von f ≤ Komplexität von A

- Fallen untere und obere Schranke zusammen, so hat man einen optimalen Algorithmus für das gestellte Problem.

1.2.1 Laufzeit bestimmen

Möglichkeiten für das „Messen“ der Effizienz eines Algorithmus:

1. Algorithmus auf realem Rechner implementieren;
   Zeitverbrauch messen
   Nachteil: Zu viele Einflussgrößen abgesehen von Algorithmus selbst

2. Zählen der Rechenschritte, die bei Durchführung für eine Eingabe gemacht werden $\Rightarrow$ Frage: was ist ein Rechenschritt?
   $\Rightarrow$ formales Berechnungsmodell, z.B. Turingmaschine oder RAM
   $\Rightarrow$ Algorithmus in künstlicher Programmiersprache „programmieren“;
   Operationen zählen und gewichten
   Nachteil: Aufwand; Frage der Übertragbarkeit

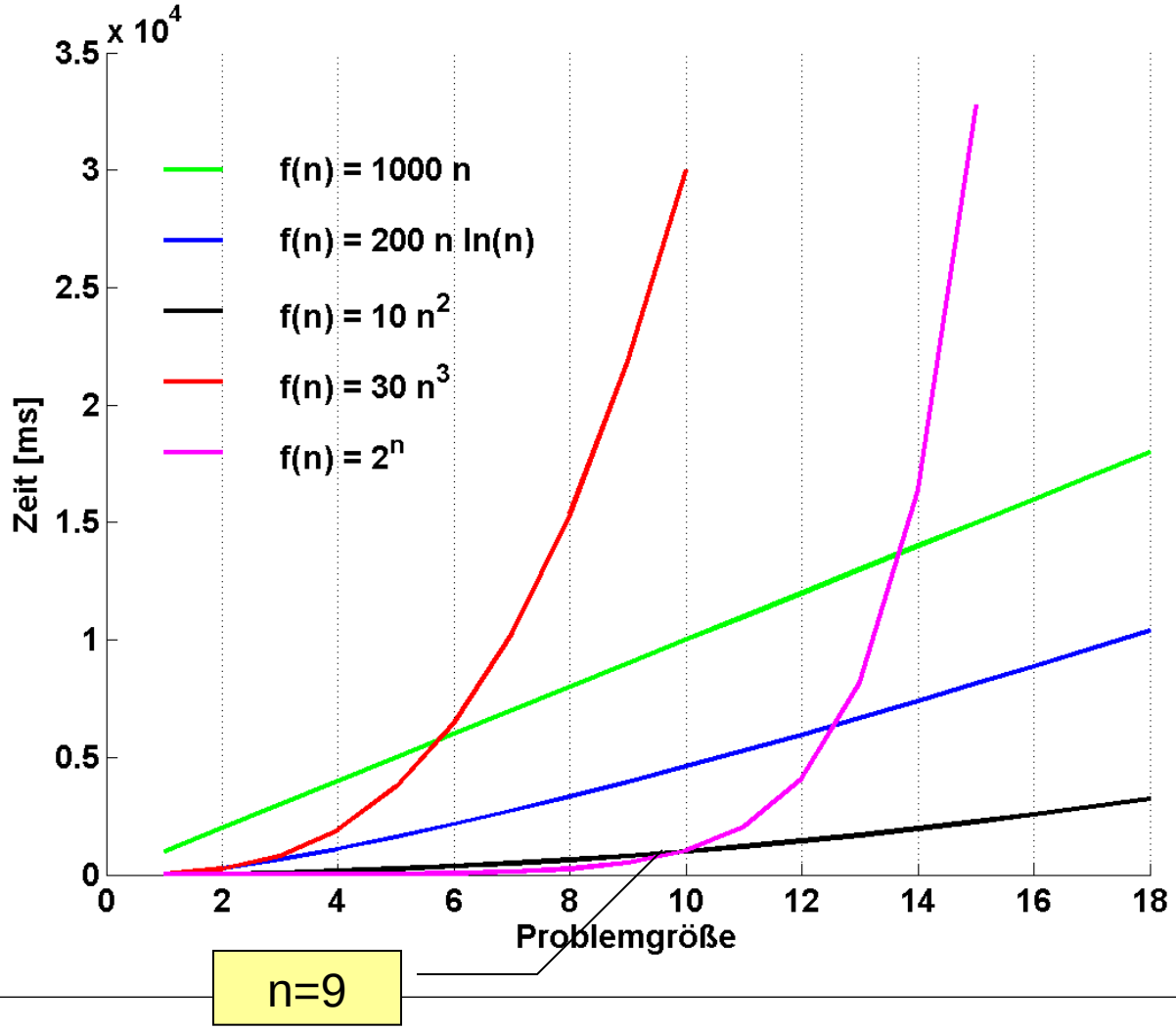
3. Operationen auf sehr hohem Niveau zählen (z.B. Anzahl der Vergleiche beim Suchen in Liste oder Anzahl der zu vertauschenden Elementpaare beim Sortieren)
   Nachteil: Grobe Abschätzung

Zeitkomplexität: Einführung

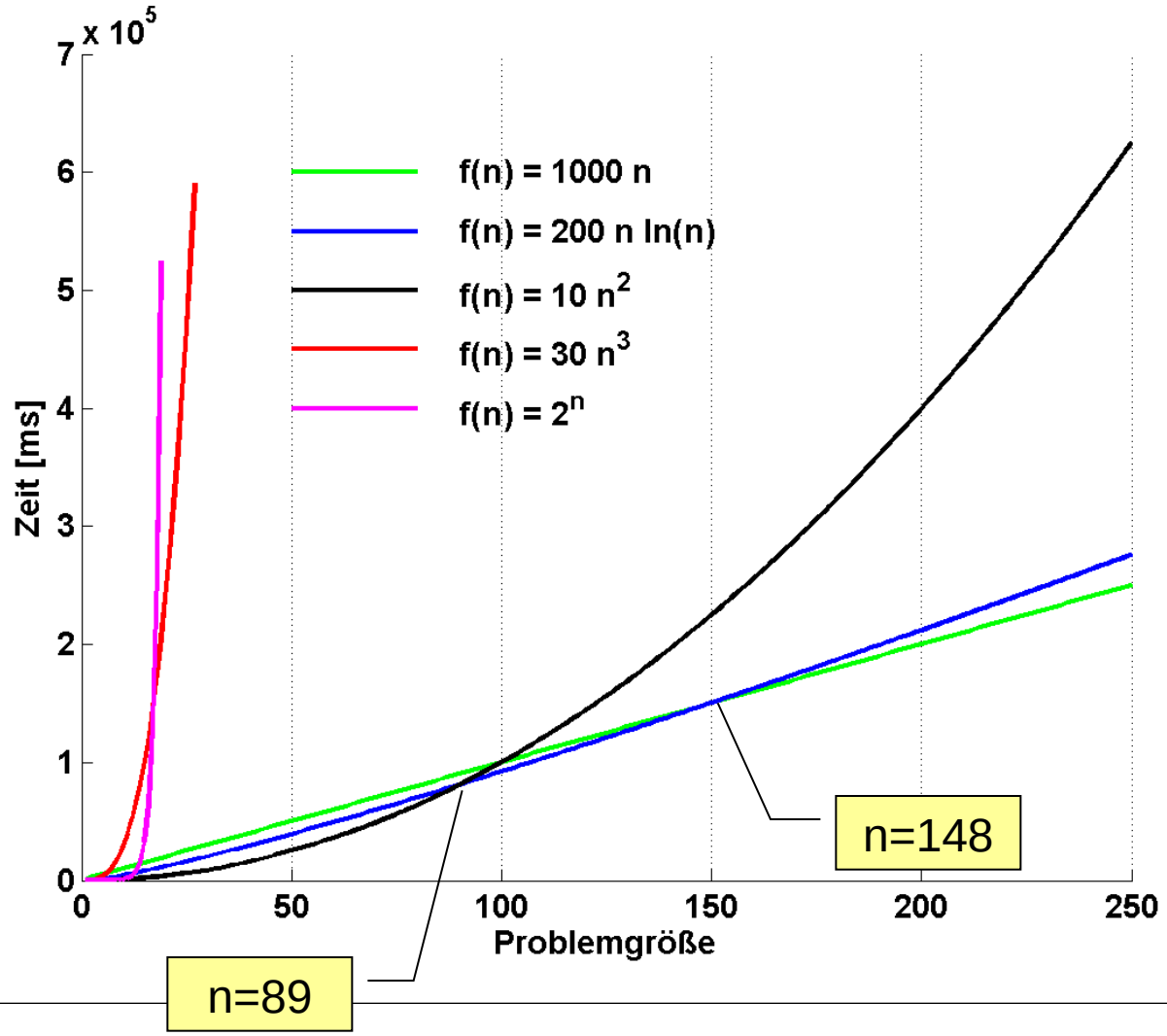
- Wir benutzen als einfach zu handhabende Messgröße den *Zeitverbrauch*.

- Gegeben ist ein Problem der Problemgröße *n*
  Beispiel: Sortieren von *n* Werten.
- Es gibt mehrere mögliche Algorithmen für das Problem.
- Welcher Algorithmus der schnellste ist, hängt von der Problemgröße *n* ab.
- Für sehr große *n* wird aber immer derjenige Algorithmus der schnellste sein, dessen Laufzeit am wenigsten mit n ansteigt.

Zeitkomplexität (bis n=18)



Zeitkomplexität (bis n=250)



Faktor in Problemgröße bei 10 mal schnellerem Rechner

10
8
3.162
2.163
1.191

Beispiel: Fehlprognose durch zu kleines n

Laufzeit: $t = n^2 + 100\ n$ (ms)

n = 5: Laufzeit 0.525 sec

n = 10: Laufzeit 1.100 sec

n = 33: Laufzeit 4.389 sec

$\Rightarrow$ IRRIGE Annahme: lineares Wachstum: $t = 110\ n$ (ms)

$\Rightarrow$ n = 200: Laufzeit: 22 sec

In Wirklichkeit sind es 420 sec = 7 min

1.2.2 Landau-Symbole

- Uns interessiert, wie ein Algorithmus für große *n* von *n* abhängt. Um das mathematisch genau fassen zu können, benutzt man die Landau-Symbole.

Edmund Georg Hermann Landau:

* 14. Februar 1877, † 19. Februar 1938

Deutscher Mathematiker, der sich um die analytische Zahlentheorie verdient gemacht hat

Definition: Landau-Symbole

In der Regel ist man nicht an exakter Anzahl v. Operationen interessiert, sondern an Komplexitätsklassen:
„Wie verändert sich der Rechenaufwand, wenn man die Eingabedaten um einen bestimmten Faktor vergrößert? Wie ist der qualitative Verlauf der Laufzeitfunktion?“

Für eine Funktion $g: \mathbb{N} \to \mathbb{R}^{>0}$ sind folgende Funktionenmengen definiert:

$$O(\mathbf{g}) = \{\mathbf{f}: \mathbb{N} \to \mathbb{R}^{>0} \mid \exists n_0 \in \mathbb{N}^{>0}, \exists c \in \mathbb{R}^{>0} \text{ mit } f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$$

$$\Omega(\mathbf{g}) = \{\mathbf{f}: \mathbb{N} \to \mathbb{R}^{>0} \mid \exists n_0 \in \mathbb{N}^{>0}, \exists c \in \mathbb{R}^{>0} \text{ mit } f(n) \geq c \cdot g(n)\ \forall n \geq n_0\}$$

Bezeichnungen für *O* (sprich: „groß O“; eigentlich „groß Omikron“):

- „Asymptotische obere Schranke“;
- „Ordnung der Funktion“;
- „Komplexitätsklasse“

Erläuterung zur O-Notation

- O(g) ist die Menge aller Funktionen, die asymptotisch höchstens so schnell wachsen wie c·g.

- z.B. ist

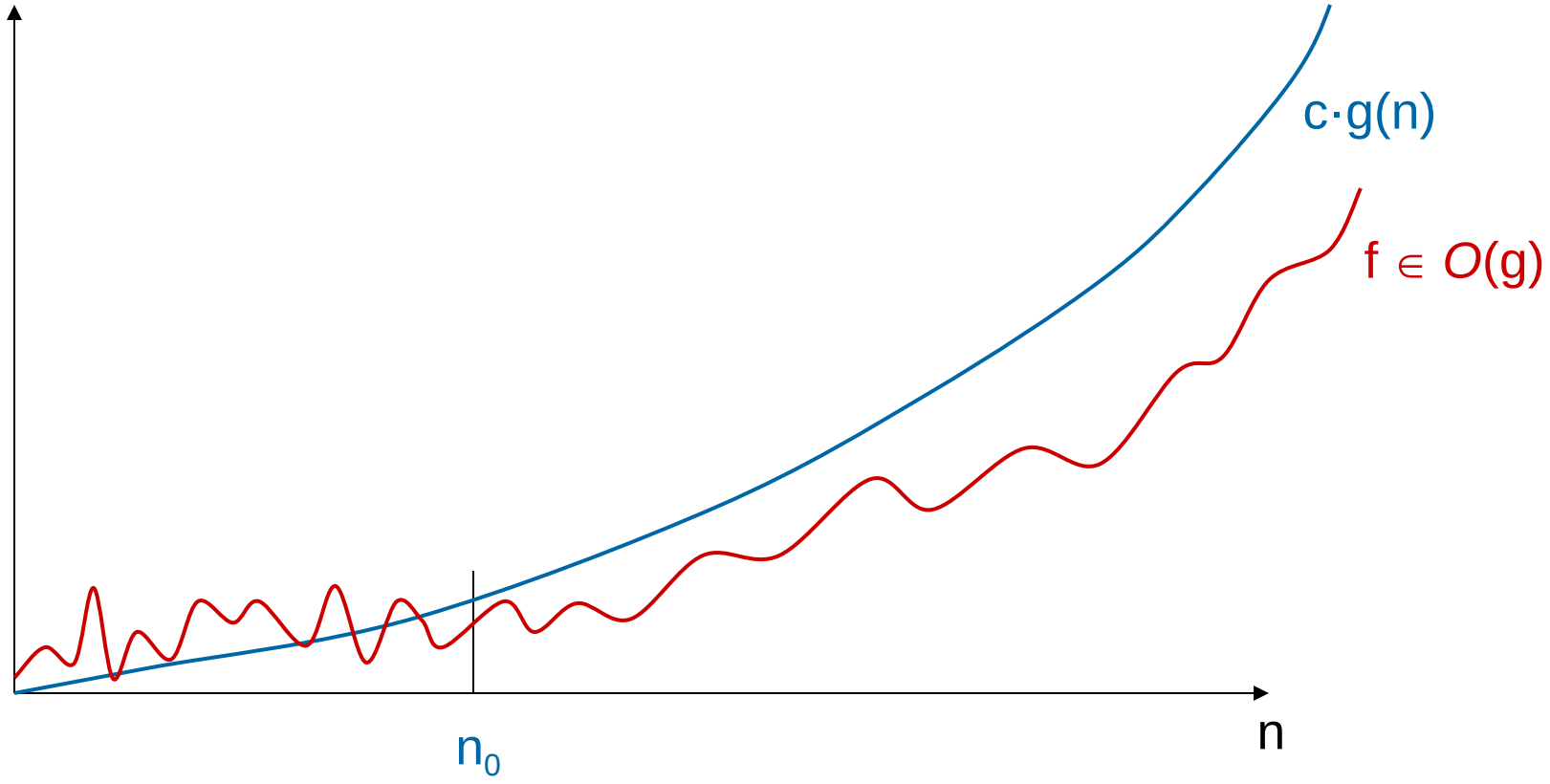
$$2n^2 + 5n + 13 \in O(n^2)$$

$$2n^2 + 5n + 13 \notin O(n)$$

$$5n + 13 \in O(n^2)$$

$$5n + 13 \in O(n)$$

Erläuterungen zur O-Notation (2)



Erläuterungen zur O-Notation (3)

- Bei Summen setzt sich der am schnellsten ansteigendeTerm durch.

Beispiel:

$f(n) = 2n^2 + 7n + 10$

steigt am schnellsten an.

$\Rightarrow f(n) \in O(n^2)$

Beweis: $2n^2+7n+10 \leq 3\,n^2$

für $n_0 = 9$

Erläuterungen zur O-Notation (4)

Man könnte auch schreiben

$$f(n) = 2n^2 + 7n + 10 \in O(2n^2 + 7n + 10)$$

Es interessiert aber nur der Vergleich mit einfachen Funktionen, wie $O(n)$, $O(n^2)$,...

Bezeichnungen typischer Klassen

Klasse	Bezeichnung	Beispiel
1	konstant	elementarer Befehl
log n	logarithmisch	Binäre Suche (2er-Logarithmus)
n	linear	Minimum einer Folge
n log n	überlinear	effiziente Sortierverfahren
n^2	quadratisch	einfache Sortierverfahren
n^3	kubisch	Matrizeninversion
n^k	polynomiell	Lineare Programmierung
2^{cn}	exponentiell	Erschöpfende Suche, Backtracking
n!	Fakultät	Permutationen, Handlungsreisender

Es gilt: $O(1) \subset O(n) \subset ... \subset O(n!)$

Faustregel für *O*-Notation

- Faustregel: Kleinstmögliche Komplexitätsklasse (in *O*-Notation) ergibt sich aus $T_A(n)$ durch:
  - Extraktion des „dominanten“ (größten) Terms und
  - Weglassen des Koeffizienten

  z.B. $T(n)=60n^2 + 4n \Rightarrow T \in O(n^2)$;
  $T(n)=ld(n) + 1 \Rightarrow T \in O(ld(n))=O(\log(n))$, wegen

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$$

Ω-Notation

- Ω(f) ist die Menge aller Funktionen, die asymptotisch mindestens ~~höchstens~~ so schnell wachsen wie c·g.

Beispiel:

$$f(n) = 2n^2 + 7n + 10$$

$$\Rightarrow f(n) \in \Omega(n^2)$$

Beweis: $2n^2+7n+10 \geq 2n^2$

für $n_0 = 1$

Θ-Notation

Ist f(n) ϵ Ω(g) und f(n) ϵ O(g), dann ist f(n) ϵ Θ(g) .

Es existiert eine obere Schranke c_1 und eine untere Schranke c_2, so dass asymptotisch (bzw. für alle $n>n_0$) gilt:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Beispiel:

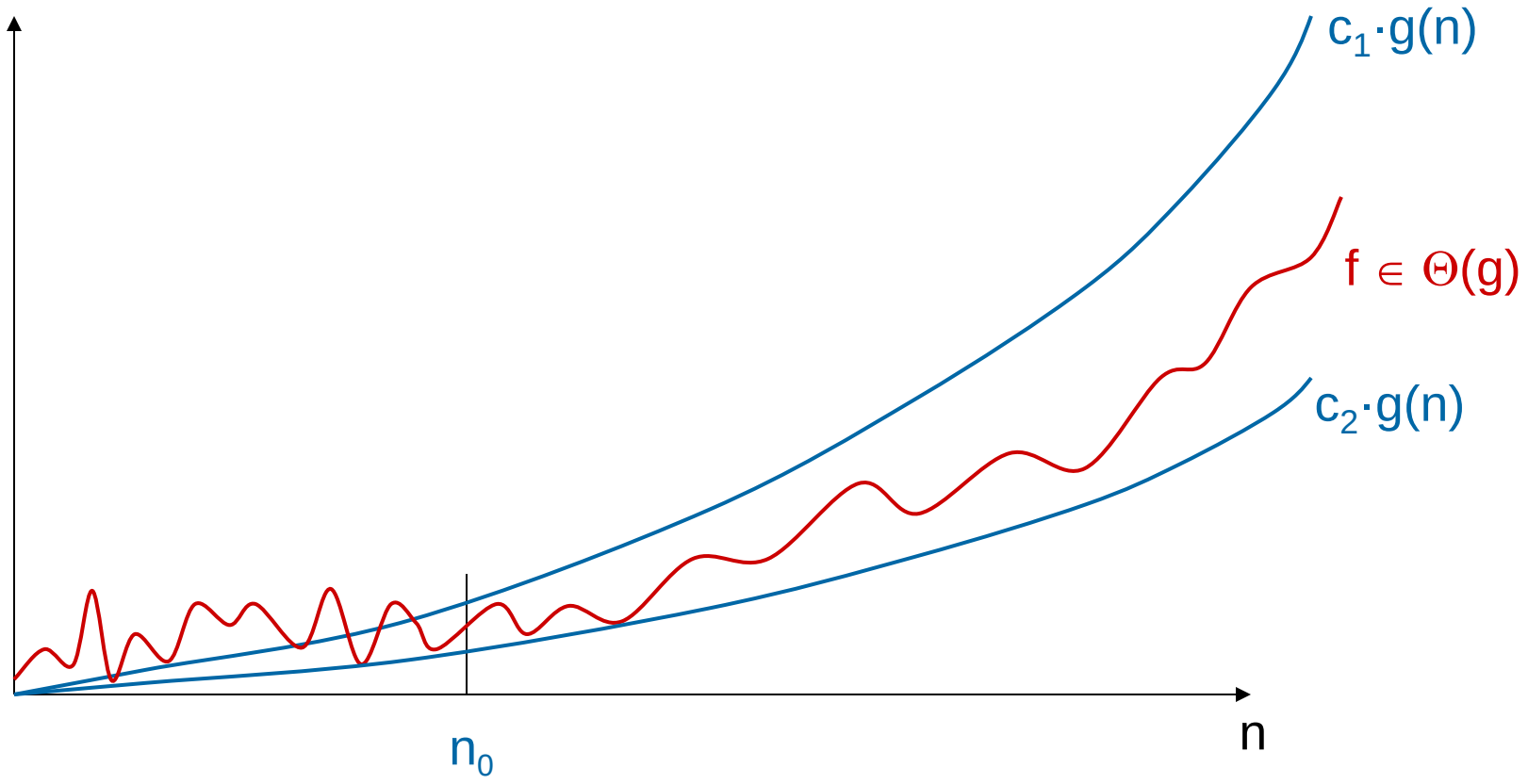
$f(n) = 2n^2 + 7n + 10$

$f(n) \in \Theta(n^2)$

Beweis: $3\,n^2 \geq 2n^2+7n+10 \geq 2\,n^2$

für $n_0 = 9$

Θ-Notation (Grafik)



Genauer hingesehen

- Ein Algorithmus habe den maximalen Zeitbedarf $T_A(n) = 3\,n^2$. Also ist
  - $T_A(n) \in O(n^2)$
  - $T_A(n) \in O(n^3)$
  - *usw.*

- Interessant ist natürlich nur das minimale O(g), hier also $O(n^2)$.
- Das Wörtchen *minimal* wird aber oft weggelassen.

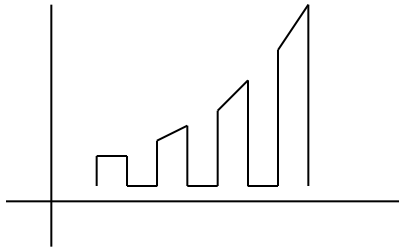
- Frage: Warum dieser Umstand mit der minimalen oberen Schranke? Könnte man nicht auch einfach die Θ-Notation benutzen?

- Antwort: Normalerweise schon, aber ....

Genauer hingesehen (2)

- Wir betrachten folgende Funktion:

```
public void sehrSeltsam(int[] array) {
  n = array.length;
  if (n % 2 == 0) {  //Feldlaenge gerade
    //Tausche nur die ersten beiden Elemente -> O(1)
  } else {           //Feldlaenge ungerade
    //Sortiere das Feld mit Bubble-Sort -> O(n^2)
  }
}
```



- Das minimale O ist $O(n^2)$
- Das maximale Ω ist $\Omega(1)$
- Der Zeitbedarf ist in keiner Menge $\Theta(g)$ vorhanden.

Beispiel

- **$T_A(n) = n(n-1)/2$**

  $T_A \in O(n^2)$, denn:

- **$T_A(n) = n^2+2n$**

  $T_A \in O(n^2)$, denn:

Also:

- Algorithmus mit $T_A(n) = n(n-1)/2$ ist zwar besser als einer mit n^2+2n(!);
- aber sie sind beide in der gleichen Komplexitätsklasse (etwa gleich gut).

Beispiel

- $T_A(n) = n(n-1)/2$

  $T_A \in O(n^2)$, denn:

  $n(n-1)/2 = 1/2\ (n^2-n) \leq n^2-n \leq n^2 \quad (n>0)$

  $\Rightarrow$ mit $n_0=1$, $c=1$ erfüllt $T_A(n)$ die Definition der *O*-Notation

- $T_A(n) = n^2+2n$

  $T_A \in O(n^2)$, denn:

  $n^2+2n \leq n^2+ 2n^2 = 3n^2$

  $\Rightarrow$ mit $n_0=1$, $c=3$ ist die Definition erfüllt

Also:

- Algorithmus mit $T_A(n) = n(n-1)/2$ ist zwar besser als einer mit n^2+2n(!);
- aber sie sind beide in der gleichen Komplexitätsklasse (etwa gleich gut).

Analyse von Algorithmen: Beispiele (1)

- Schleife

```
for (int i=1; i<=n; i++) {
      a[i] = 0;
}
```

O( )

- Nacheinanderausführen

```
for (int i=1; i<=n; i++)
      a[i] = 0;
for (int i=1; i<=n; i++)
      for (int j=1; j<=n; j++)
            a[i] = a[j] + i + j;
```

O( )

- Geschachtelte Schleife

```
for (int i=1; i<=n; i++)
      for (int j=1; j<=n; j++)
            k++;
```

O( )

- Bedingte Anweisung

```
if (x > 100) {
      y = x;
} else {
      for (int i=1; i<=n; i++)
            if (a[i] > y)
                  y = a[i];
}
```

O( )

Analyse von Algorithmen: Beispiele (2)

- Innere Schleifenbedingung hängt von Lauf ab

```
for (int i=1; i<=n; i++)
      for (int j=1; j<= i ; j++)
            k++;
```

T(n)=

$\Rightarrow$ O( )

- Aussprung

```
for (int i=1; i<=n; i++)
      if (i > 2)
            return;
```

T(n)= $\Rightarrow$ O( )

- Schleifenvariable in jedem Lauf halbiert

```
int i=n;
while (i>=1) {
      x ++;
      i /= 2;     // i wird immer halbiert
}
```

T(n)=

$\Rightarrow$ O( )

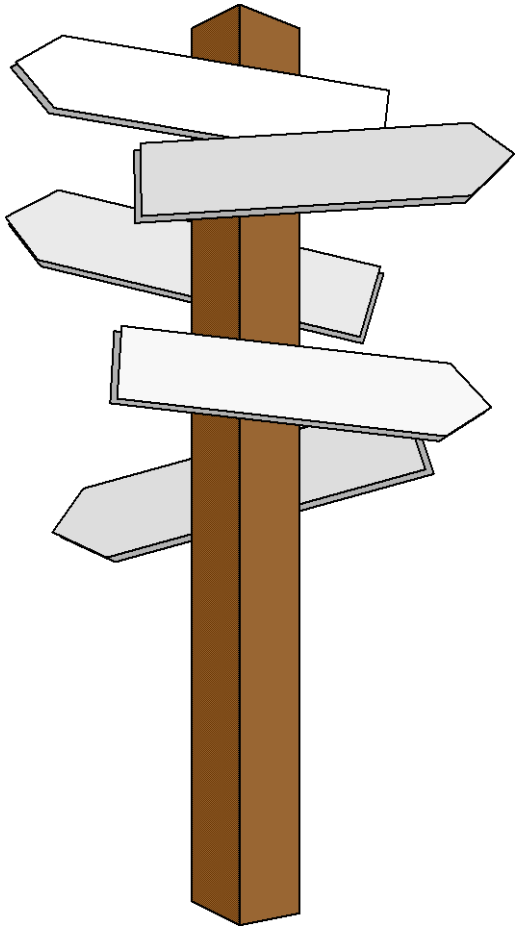
O(n) Funktionsaufrufe

```
static long fakultaet(int n) {
  if (n>0) {
    return (n * fakultaet(n-1));
  } else {
    return 1;
  }
}
```

Worst, Best, Average Case

- Man unterscheidet den Zeitbedarf im
  - im schlechtesten Fall (worst case),
  - in einem „durchschnittlichen“ Fall (average case) und
  - im besten Fall (best case).

- Der durchschnittliche Fall ist dann wichtig, wenn der beste oder der schlechteste Fall selten auftretende Sonderfälle sind.



3.1 Grundbegriffe
3.2 Einfache Datenstrukturen
Record, Set (Menge)
3.3 Lineare homogene Datenstrukturen
Array (Feld), Liste (List)
Stack, Queue, Deque
3.4 Graphen
Binäre (Such-)Bäume, B-Bäume, Heaps

- Datentyp/Datenstruktur
- Abstrakter Datentyp

Datentypen in der Informatik

Für einfache Datentypen findet sich folgende Definition

z.B.:

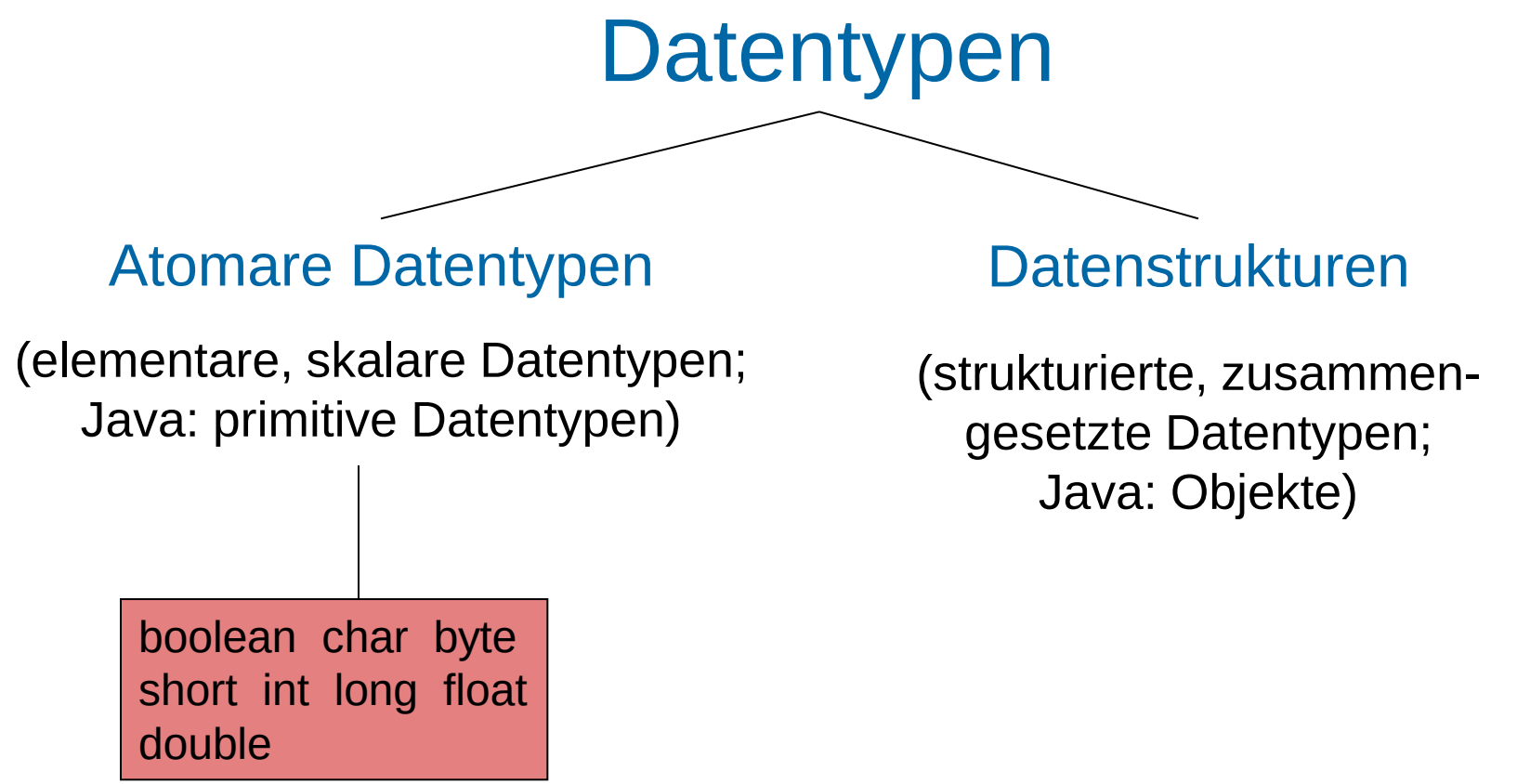
- Sedgewick: Algorithmen in Java
- Wirth: Algorithmen und Datenstrukturen
- Wikipedia

Ein ***Datentyp*** ist aus zwei Angaben festgelegt:

1. eine Menge von Daten (Werte);
2. eine Menge von Operationen auf diesen Daten

Beispiel zu dieser Definition

- Ganzzahl (‚Integer‘):
- eine ganze Zahl zwischen einer Ober- und Untergrenze und mit einer Anzahl von Standard-Operationen:
  - **+, -, *, /, %**
  - **Je nach Definition auch <, >, ==, sign, abs, odd, even, …**


Datentypen
Atomare Datentypen
(elementare, skalare Datentypen;
Java: primitive Datentypen)
boolean char byte
short int long float
double
Datenstrukturen
(strukturierte, zusammen-
gesetzte Datentypen;
Java: Objekte)

Beispiel für Datenstruktur

Vektor = Strukturierter Datentyp (Datenstruktur) mit

- Datenelementen,
- Strukturregeln
- speziellen Operationen

$$\vec{a} = \begin{pmatrix} 2.1 \\ 0.1 \\ 6.3 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 5.3 \\ 2.2 \\ 3.2 \end{pmatrix}$$

$$\vec{c} = \vec{a} + \vec{b} = \begin{pmatrix} 2.1 \\ 0.1 \\ 6.3 \end{pmatrix} + \begin{pmatrix} 5.3 \\ 2.2 \\ 3.2 \end{pmatrix} = \begin{pmatrix} 2.1+5.3 \\ 0.1+2.2 \\ 6.3+3.2 \end{pmatrix} = \begin{pmatrix} 7.4 \\ 2.3 \\ 9.5 \end{pmatrix}$$

Homogene / heterogene Datenstruktur

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

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

Abstrakte Datentypen (ADTs)

- 2 Prinzipien:
  - Kapselung: Zu einem ADT gehört eine Schnittstelle. Zugriffe auf die Datenstruktur erfolgen ausschließlich über die Schnittstelle (z.B.: addElement(...), deleteElement(...))
  - Geheimnisprinzip: Die interne Realisierung eines ADT-Moduls bei der Umsetzung bleibt verborgen.

Spezielle ADT 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.
  - **Package java.util**
- Ähnliches gilt für andere Bibliotheken, z.B. .NET:
  - **Namespace System.Collections.Generic**

Implementation abstrakter Datentypen

- Abstrakte Datentypen können unterschiedlich realisiert werden.
- Z.B. der ADT „Liste“ mit der Datenstruktur „dynamisches Feld“ oder „verkettete Liste“.
- Daher gibt es in Java zum Interface „List“ die Implementationen „ArrayList“ und „LinkedList“.

Spezielle ADTs in dieser Vorlesung

ADT	Java-Interface	Java-Klassen (Bsp.)
Record (nur kurz)	-	(einfache Klassen)
Feld (nur kurz)	-	(Felder)
Liste, Stack, Queue, Deque	List, Queue, Deque	ArrayList, LinkedList, ArrayDeque
Menge, Assoziatives Feld	Set, Map	HashSet, HashMap
Prioritätswarteschlange	-	PriorityQueue

ADT Record

Ein ***Record*** ist eine Zusammenfassung von unterschiedlichen Datenelementen zu einer Einheit.

Ein Record entspricht in Java einer Klasse

- mit mehreren public-Instanzvariablen
- und ohne Methoden

```
public class Person {
            public String name;
            public int id;
}
```

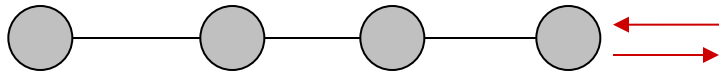
ADT Feld und Liste

- Der ADT Array(Feld) hat folgende spezielle Eigenschaften:
  - **Feste Anzahl von Datenobjekten**
  - **Auf jedes Objekt kann direkt (lesend oder schreibend) zugegriffen werden**
  - **In Java: Normales Feld**

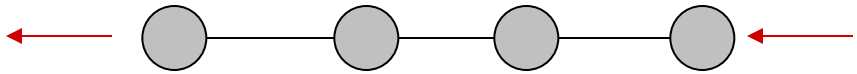
- Für die ADT Liste(List) kommt hinzu:
  - **Datenobjekte können an jeder Stelle eingefügt oder gelöscht werden.**
  - **Die Liste kann also wachsen und schrumpfen.**
  - **ADT in Java: Interface List**
  - **Bekannte Implementierung: ArrayList**

ADT Stack, Queue und Deque

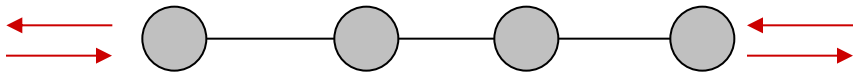
- Spezieller sind Stack, Queue und Deque
  - **Stack (Stapel, Keller): Daten können an einem Ende hinzugefügt oder entnommen werden.**



  - **Queue (Warteschlange): Daten können an einem Ende hinzugefügt und am anderen Ende entnommen werden.**



  - **Deque (Double ended queue): Daten können an beiden Enden hinzugefügt und entnommen werden.**



ADT Menge (Set)

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.

Operationen:

- **Wichtigste Operationen: Einfügen, Löschen, Testen (ob vorhanden).**

ADT Assoziatives Feld

- Englische Namen: Associative Array, Dictionary, Map

- Wichtigste Operationen
  - **Schlüssel-Wert-Paar einfügen.**
  - **Wert zum Schlüssel finden.**
- ADT: Java-Interface Map.
- Bekannte Implementierung: HashMap.

ADT Prioritätswarteschlange

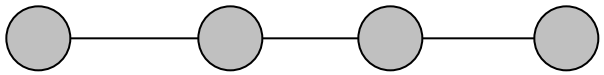
- Auch **Vorrangwarteschlange** oder **Priority Queue** genannt.
- Eine Warteschlange, deren Elemente einen Schlüssel (Priorität) besitzen.

- Wichtige Operationen bei Prioritätswarteschlangen:
  - Element in Schlange einfügen
  - Element mit der höchsten Priorität entnehmen.

- Einfach verkettete Liste
- Doppelt verkettete Liste
- Dynamisches Feld
- Anwendungen für lineare Strukturen:
  - Stack (LIFO-Speicher, Stapel, Keller)
  - Queue (FIFO-Speicher, Warteschlange)
  - Deque

Definition der abstrakten Datenstrukturen

- Linearen homogenen Datenstrukturen ist gemeinsam:
  - **mehrere gleichartigen Datenobjekte**
  - **1:1 - Relation**



Definition der abstrakten Datenstrukturen(2)

- Lineare homogene Datenstrukturen eignen sich besonders gut für
- die ADT Feld,
- die ADT Liste,
- sowie die Spezialfälle der Liste Stack, Queue und Deque.

Implementation

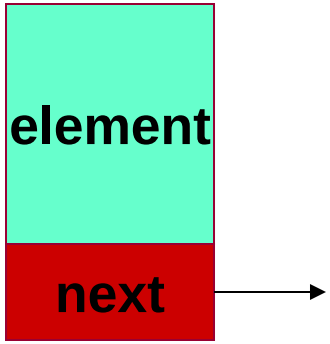
- Die wichtigsten linearen homogenen Datenstrukturen sind:

- Die verkettete Liste und das
- Feld (eventuell dynamisch und/oder zirkulär).

- Zum Beispiel in Java (.NET):
  - **Interface List (IList) (ADT)**
  - **Implementation ArrayList (List) (dynamisches Feld)**
  - **Implementation LinkedList (LinkedList) (doppelt verkettete Liste)**

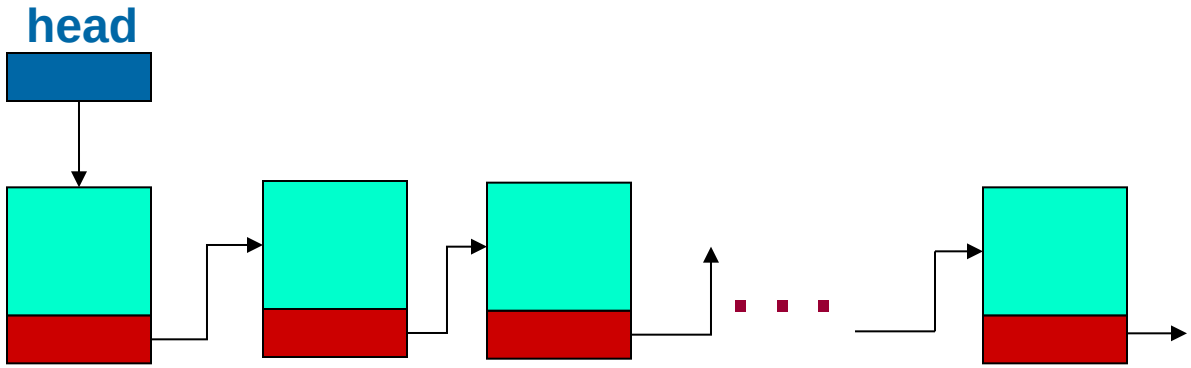
Eine **Einfach verkettete Liste** besteht aus Knoten.

```
class Node {
    Object element;  // Datenkomponente
    Node next;       // Verweis auf Restliste
}
```



Zusammenfügen von Knoten

- Durch Hintereinanderhängen von Knoten entsteht eine Liste

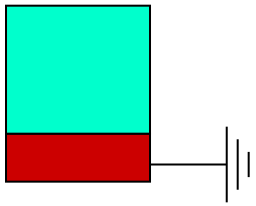


- Die Listenklasse muss sich nur den Kopf merken
- Jeder Knoten zeigt auf seinen Nachfolger

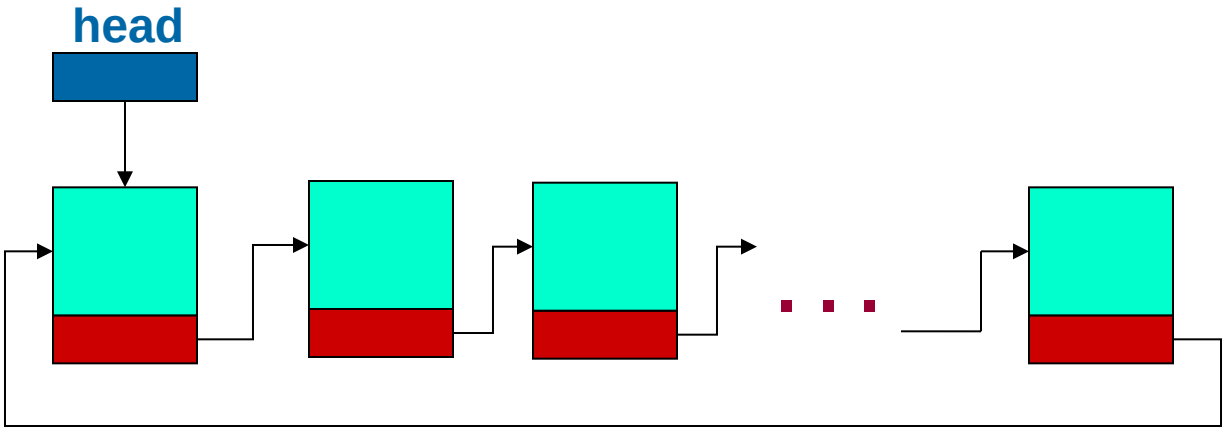
```
class MyList {
    Node head;

}
```

Listenende

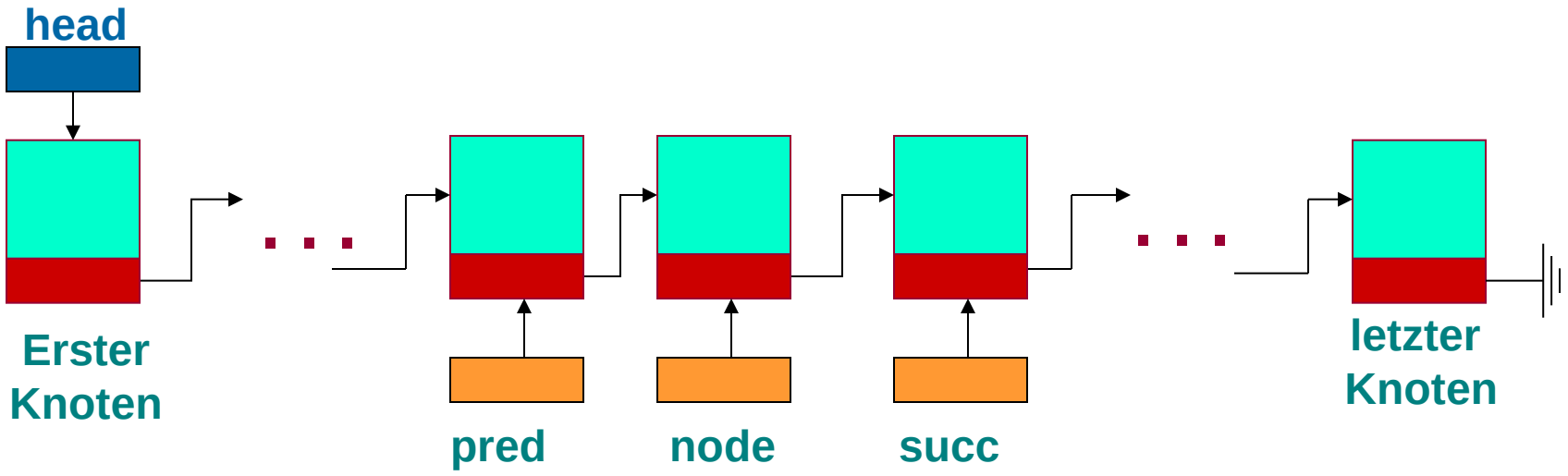
- Leerer Verweis am Listenende
  - **Für letztes Element gilt:**
    `node.next = null;`



- Alternative : Rückverweis auf Listenanfang ($\Rightarrow$ Zirkuläre Liste)



Verkettete Liste: Durchlauf



Entsprechung für Durchlauf von Arrays:

```
for (int i=0; i < a.length; i++)
    ...
```

bei Listen:

```
for (Node node=head; node != null; node = node.next)
    ...
```

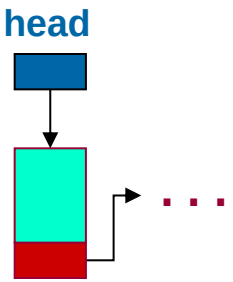
Einfügen und Löschen

- *void **insert** (int pos, Datenelement e)*
  - **Füge Element an Position *pos* ein**
- *void **delete** (int pos)*
  - **Lösche den Knoten an Position *pos***

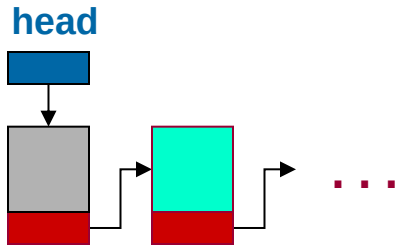
➔ Übung am Computer

Varianten für Listenanfang

Listenanfang erfordert Sonderbehandlung



Alternative: ‚Pseudo-Knoten' am Listenanfang



Komplexität

- Komplexitätsanalyse der einfach verketteten Liste:

Auslesen Anfang	O(1)
Auslesen Mitte	O(n)
Auslesen Ende	O(n)
Einfügen/Löschen Anfang	O(1)
Einfügen/Löschen Mitte	O(n)
Einfügen/Löschen Ende	O(n)

Doppelt verkettete Liste

- Verweis auf Nachfolger und Vorgänger


```
class DNode {
    Object obj;
    DNode pred; //Vorgänger
    DNode succ; //Nachfolger
}
```

```
class DList {
    DNode first; //Anfang
    DNode last;  //Ende
}
```

- Listenklasse merkt sich head und tail

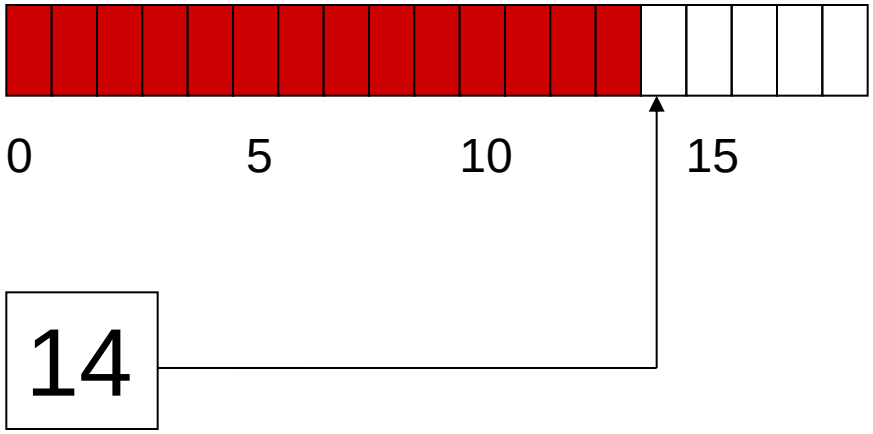
Komplexität

- Komplexitätsanalyse der einfach und doppelt verketteten Liste:

	1fach	2fach
Java-Klasse	-	Linked List
Auslesen Anfang	O(1)	O(1)
Auslesen Mitte	O(n)	O(n)
Auslesen Ende	O(n)	O(1)
Einfügen/Löschen Anfang	O(1)	O(1)
Einfügen/Löschen Mitte	O(n)	O(n)
Einfügen/Löschen Ende	O(n)	O(1)

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.**



Einfügen hinten (1)

Dynamisches Feld:

Solange das neue Element noch in das Feld passt:

```
list[n] = newEntry;
n++;
```

⇒ Einfügen eines Elements: O(1)

- Wenn das neue Element nicht mehr in das Feld passt:
  - **Ein Feld mit einer größeren Anzahl von Elementen wird angelegt.**
  - **Alle bisherigen Elemente werden umkopiert.**
  - **Das neue Element wird angehängt.**

⇒ O(???)

Einfügen hinten (2)

- Nehmen wir als Beispiel: 1000 Elemente sollen in die Liste eingefügt werden.
- Java nimmt als Anfangsgröße für das Feld 10
  - **Dies kann (und sollte auch) verändert werden, falls die maximale Größe der Liste vorher bekannt ist.**

- Immer wenn das Feld zu klein wird, wird ein neues Feld mit doppelt so vielen Elementen erzeugt und alle Elemente umkopiert:
  - **Dies ist eine Vereinfachung zur besseren Rechnung. In Wirklichkeit beträgt der Faktor:**
    - **Sun-Java: 1,5**
    - **Gnu-Java: 2**
    - **C# (Mono): 2**
    - **Python: 1,125**

Einfügen hinten (3)

Element Nr.	Neue Feldgröße	Umzukopierende Elemente
11	20	10
21	40	20
41	80	40
81	160	80
161	320	160
321	640	320
641	1280	640

Summe: s =1270

- Wird eine Liste aus *n* Elemente aufgebaut, ist die Anzahl der kopierten Elemente immer $\leq 2n$
- $\Rightarrow$ Aufbau von n Elementen: O(n)

Aufbau einer Liste von n Elementen durch Einfügen neuer Elemente von vorne:

Bei jedem Einfügen müssen alle Elemente einen Platz nach hinten kopiert werden

Das ergibt:

- 1+2+3+.....+(n-1)=(n*(n-1))/2 Kopiervorgänge
- $\Rightarrow O(n^2)$

Auslesen

- Dynamisches Feld: Elemente können generell direkt ausgelesen werden.

$\Rightarrow O(1)$

Komplexität

	1fach	2fach	Dyn. Feld
Java-Klasse	-	Linked List	Array List
Auslesen Anfang	O(1)	O(1)	O(1)
Auslesen Mitte	O(n)	O(n)	O(1)
Auslesen Ende	O(n)	O(1)	O(1)
Einfügen/Löschen Anfang	O(1)	O(1)	O(n)
Einfügen/Löschen Mitte	O(n)	O(n)	O(n)
Einfügen/Löschen Ende	O(n)	O(1)	avg.O(1) wst.O(n)

ArrayList - LinkedList

- ArrayList ist in den meisten Fällen schneller.
- ArrayList ist insbesondere schneller, wenn oft auf Elemente in der Mitte der Liste zugegriffen werden muss.

- LinkedList ist schneller, wenn oft Elemente am Anfang und am Ende der Liste eingefügt oder gelöscht werden.
  - **Wichtig für Queues oder Deques.**
  - **Hier kann auch ein „zirkuläres dynamisches Feld“ verwendet werden (siehe Queues).**
- Das Einfügen eines einzelnen Elements ist bei einer ArrayList im schlechtesten Fall O(n).

Stapel

- andere Namen: Stack, Kellerspeicher, LIFO-Liste (**L**ast-**I**n-**F**irst-**O**ut)

- Anwendungen:
  - Rücksprungadressen bei geschachtelten Funktionsaufrufen
  - Wichtige (interne) Struktur für rekursive Programme
  - Mit Hilfe eines Stacks können rekursive Programme in iterative umgewandelt werden.
  - ...

- Anwendung außerhalb des EDV-Bereichs:
  z.B. Tablettstapel in der Mensa

- Einfügen („push“) und Entfernen („pop“) am gleichen Ende („top“).

Geeignete Datenstrukturen

- In einem Stack werden nur auf einer Seite Daten angehängt bzw. entfernt.
- Sowohl verkettete Listen als auch dynamische Felder sind gut geeignet.
  - **Beide haben hier O(1).**

- In Java (java.util.Stack) als auch in .NET (System.Collections.Generic.Stack) wird ein **dynamisches Feld** benutzt (das etwas schneller ist).

ADT Stack in Java

- Es gibt in Java kein Interface Stack als Entsprechung des ADTs.
- java.util.Stack ist in Java eine konkrete Implementation des Stacks mit einer dynamischen Liste.
- Von der Namensgebung her ist das eine „Altlast“ aus Java 1.0.

Rekursion als Anwendung von Stacks

Rekursive Methode zur Berechnung der Fakultät:

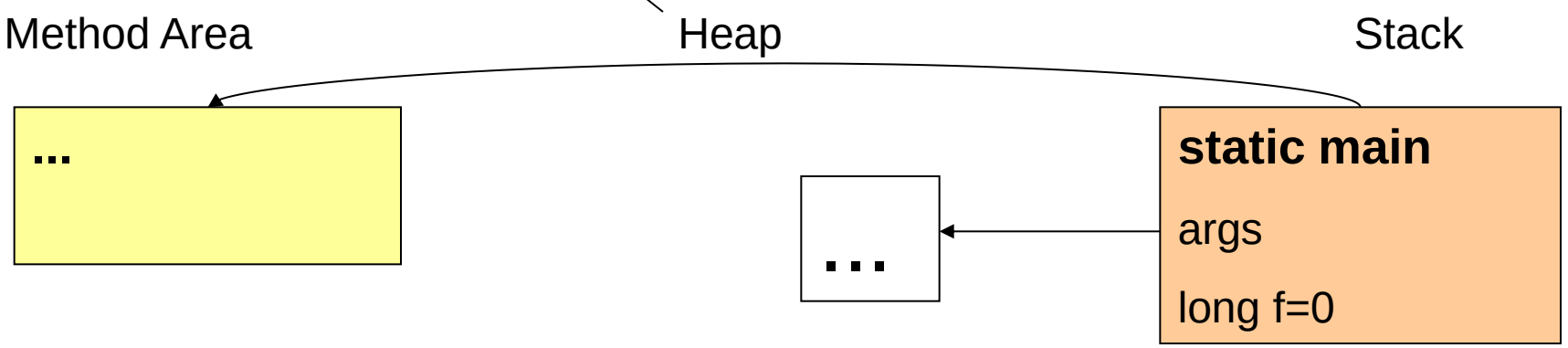
```
public static long fakultaet(int n) {
   if (n==0) {
      return 1;
   } else {
      long m = n * fakultaet(n-1);
      return m;
   }
}
```

Speicherbereich und Datenstruktur

Der Speicherbereich Heap besitzt nicht die Datenstruktur Heap.

Unterscheide:
- Speicherbereich Stack
- Datenstruktur Stack.

Der Speicherbereich Stack besitzt die Datenstruktur Stack.

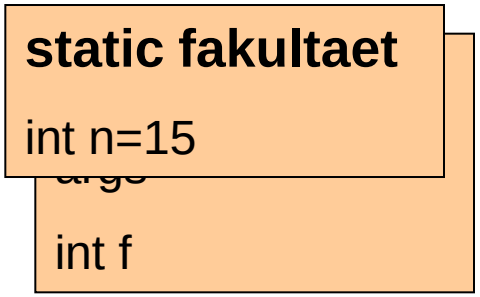


Rekursive Aufrufe (1)

```
public static long fakultaet(int n) {
  if (n==0) {
    return 1;
  } else {
    long m = n * fakultaet(n-1);
    return m;
  }
}
```

Nur Stack ist interessant.

Stack

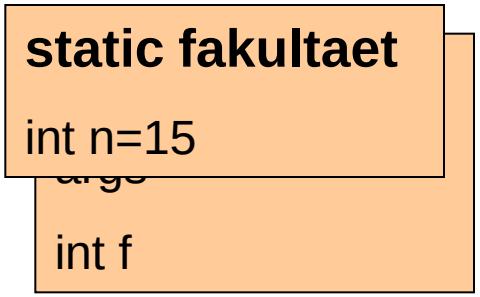


Rekursive Aufrufe (2)

```
public static long fakultaet(int n) {
  if (n==0) {
    return 1;
  } else {
    long m = n * fakultaet(n-1);
    return m;
  }
}
```

Nur Stack ist interessant.

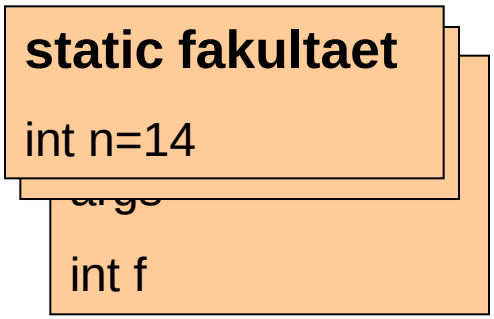
Stack



Rekursive Aufrufe (3)

```
public static long fakultaet(int n) {
  if (n==0) {
    return 1;
  } else {
    long m = n * fakultaet(n-1);
    return m;
  }
}
```

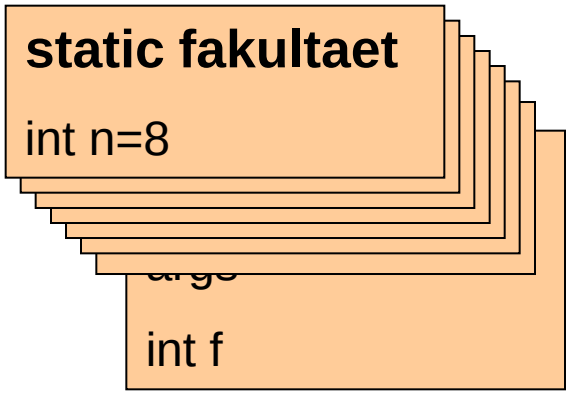
Stack



Rekursive Aufrufe (4)

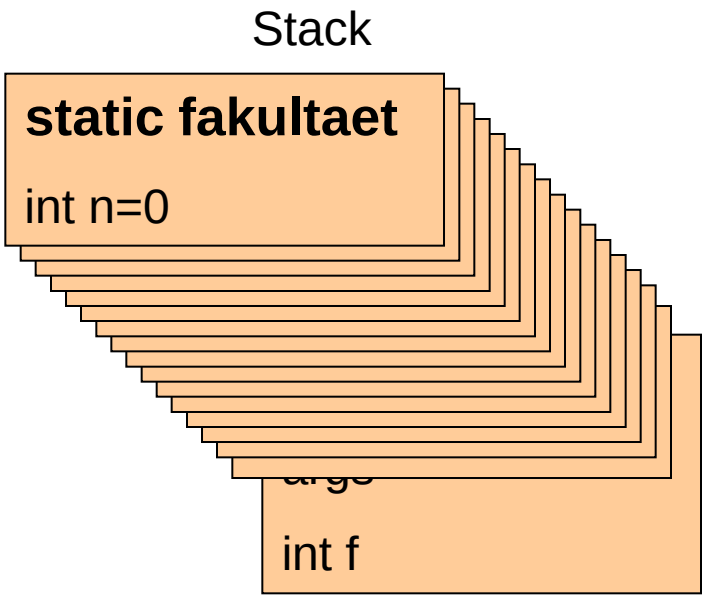
```
• public static long fakultaet(int n) {
    if (n==0) {
      return 1;
    } else {
      long m = n * fakultaet(n-1);
      return m;
    }
  }
```

Stack



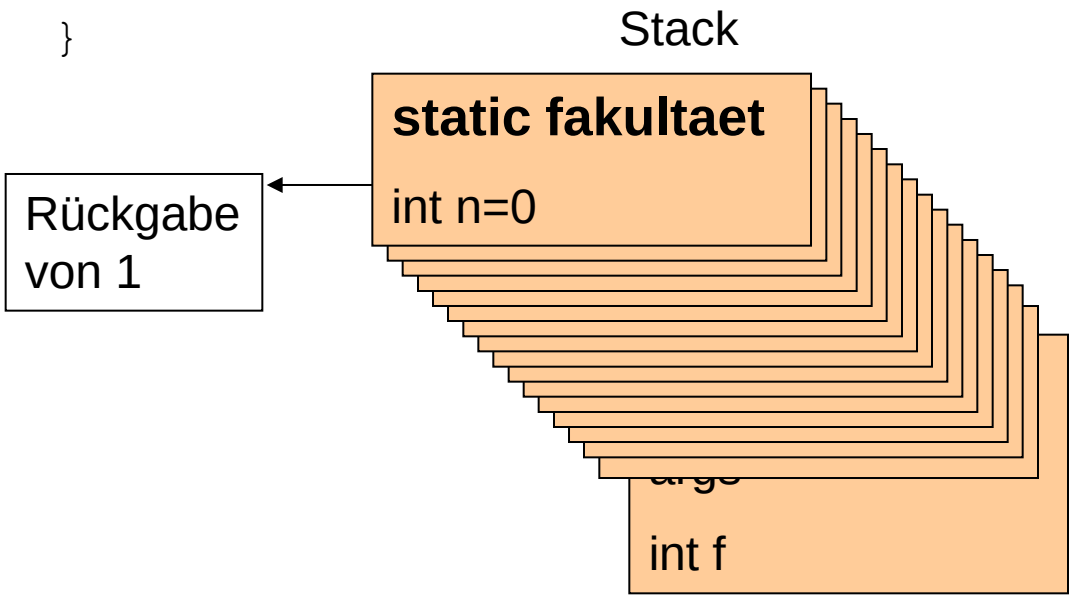
Rekursive Aufrufe (5)

```
• public static long fakultaet(int n) {
    if (n==0) {
      return 1;
    } else {
      long m = n * fakultaet(n-1);
      return m;
    }
  }
```



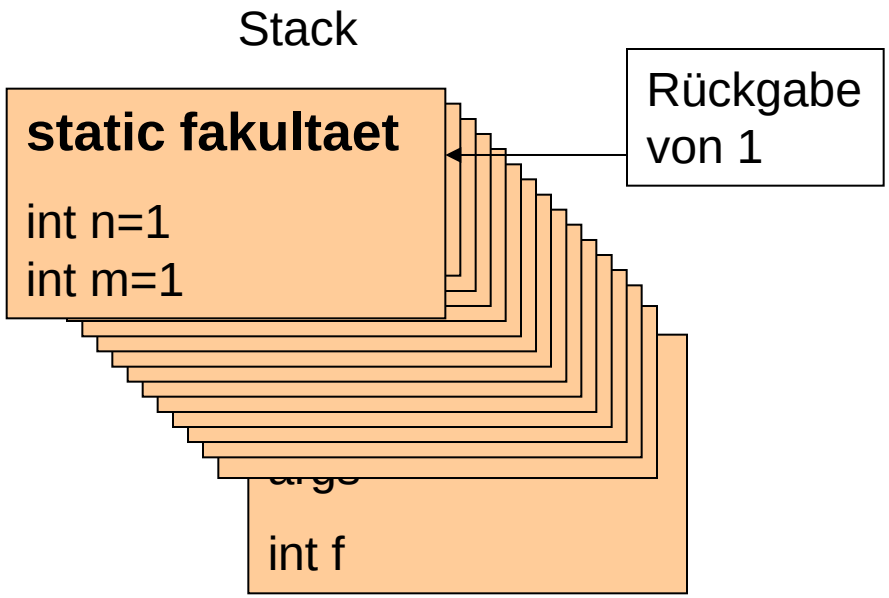
Rekursive Aufrufe (6)

```
public static long fakultaet(int n) {
     if (n==0) {
       return 1;
     } else {
       long m = n * fakultaet(n-1);
       return m;
     }
   }
```



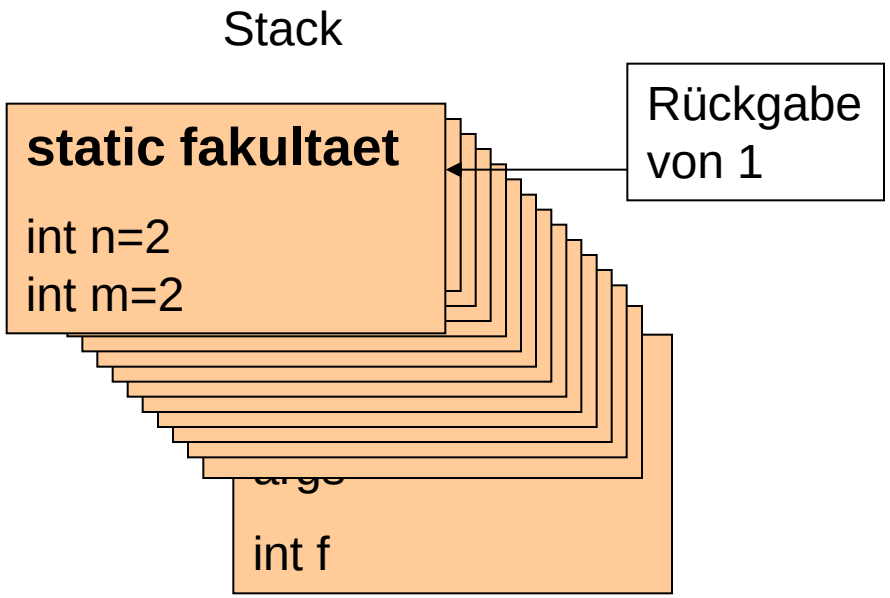
Rekursive Aufrufe (7)

```
public static long fakultaet(int n) {
     if (n==0) {
       return 1;
     } else {
       long m = n * fakultaet(n-1);
       return m;
     }
   }
```



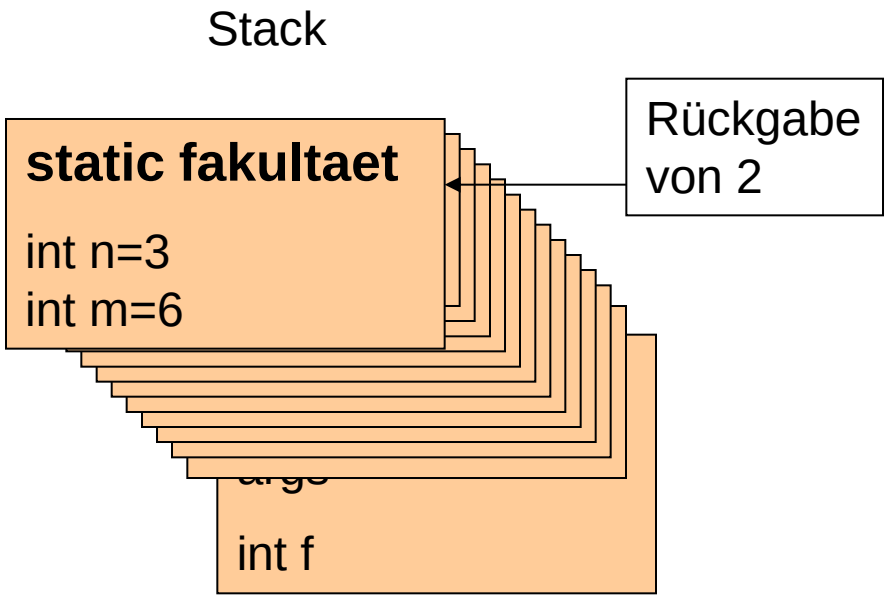
Rekursive Aufrufe (8)

```
public static long fakultaet(int n) {
     if (n==0) {
       return 1;
     } else {
       long m = n * fakultaet(n-1);
       return m;
     }
   }
```



Rekursive Aufrufe (9)

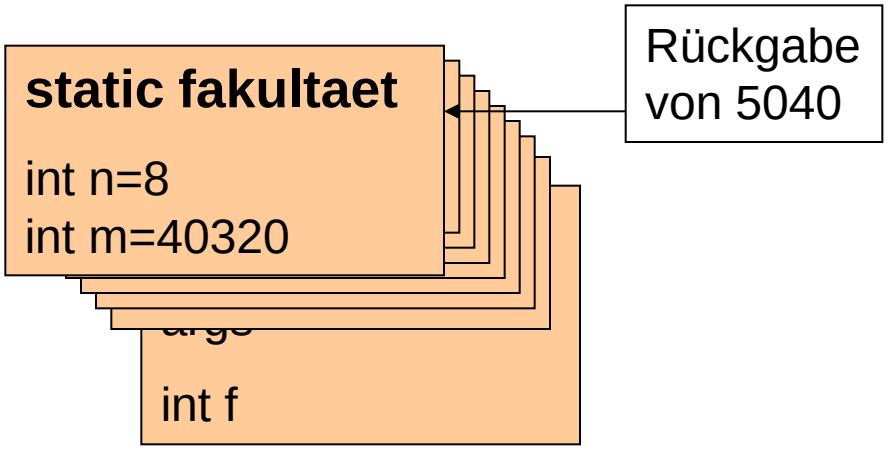
```
public static long fakultaet(int n) {
  if (n==0) {
    return 1;
  } else {
    long m = n * fakultaet(n-1);
    return m;
  }
}
```



Rekursive Aufrufe (10)

```
public static long fakultaet(int n) {
     if (n==0) {
       return 1;
     } else {
       long m = n * fakultaet(n-1);
       return m;
     }
   }
```

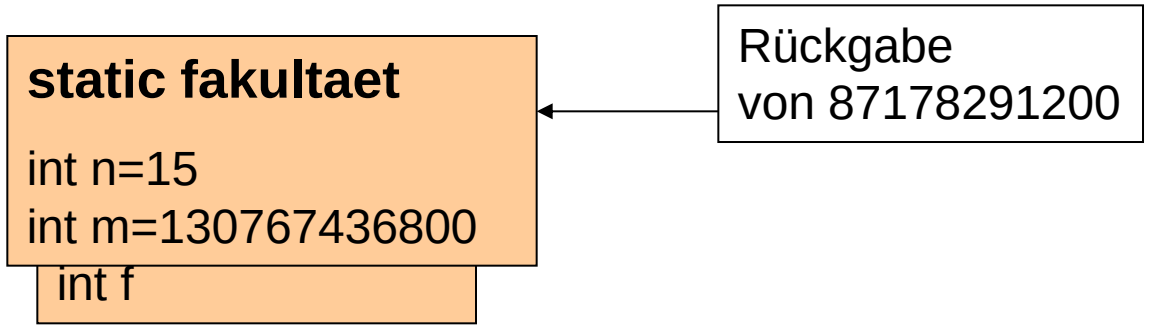
Stack



Rekursive Aufrufe (11)

```
public static long fakultaet(int n) {
     if (n==0) {
       return 1;
     } else {
       long m = n * fakultaet(n-1);
       return m;
     }
   }
```

Stack



Iterative Umsetzung mit einem Stack

```
public static Stack<Integer> stack =
                                        new Stack<Integer>();


public static long fakultaet(int n) {
  while (n>1) {
    stack.push(n);
    n--;
  }
  while (stack.isEmpty()==false) {
    n = n * stack.pop();
  }
  return n;
}
```

Warteschlange

- anderer Name: Queue, FIFO-Liste (**F**irst-**I**n-**F**irst-**O**ut)
- Anwendungen: überall wo Reihenfolge wichtig ist z.B.
  - Resourcen zuweisen im Betriebssystem
  - ...
- Anwendung außerhalb des EDV-Bereichs:
  z.B. Warten an der Mensa-Kasse
- Einfügen („put“/„enqueue“) am Ende („tail“) und
- Entfernen („get“/„dequeue“) am Anfang („head“)

Geeignete Datenstrukturen

- In einer Queue werden auf beiden Seiten Daten verschoben.
- Doppelt verkettete Listen sind gut geeignet (O(1)).
- Dynamische Felder sind schlecht geeignet. Entweder put oder get hat O(n).

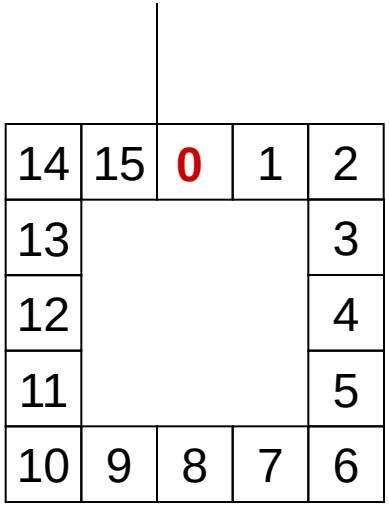
- Dieser Nachteil wird durch **zirkuläre dynamische Felder** vermieden.
  - **O(1) bei put und get.**
  - **Gewöhnlich schneller als verkettete Listen.**

ADT Queue in Java

- Interface java.util.Queue
  - *put* → **Methode offer(..)**
  - *get* → **Methoden remove() und poll()**
  - **Oberstes Element ansehen, ohne es zu entnehmen → Methoden peek() und element()**
  - *empty()* **ist nicht direkt vorhanden, kann aber mit (peek()==null) umschrieben werden.**

- Wichtigste Implementationen:
  - **java.util.ArrayDeque (zirkuläres dynamisches Feld)**
  - **Java.util.LinkedList (doppelt verkettete Liste)**
- .NET: System.Collections.Generic.Queue (zirk. dyn. Feld)

Zirkulär organisiertes Feld



Umlauf im Uhrzeigersinn:

```
i = (i+1) % data.length;
```

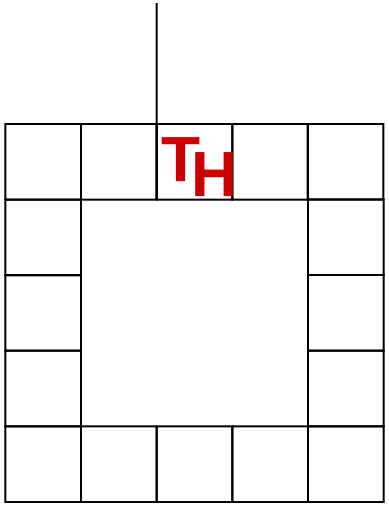
Umlauf gegen den Uhrzeigersinn:

```
i = (i-1+data.length) % data.length;
```

Queue mit Hilfe eines Feldes (1)

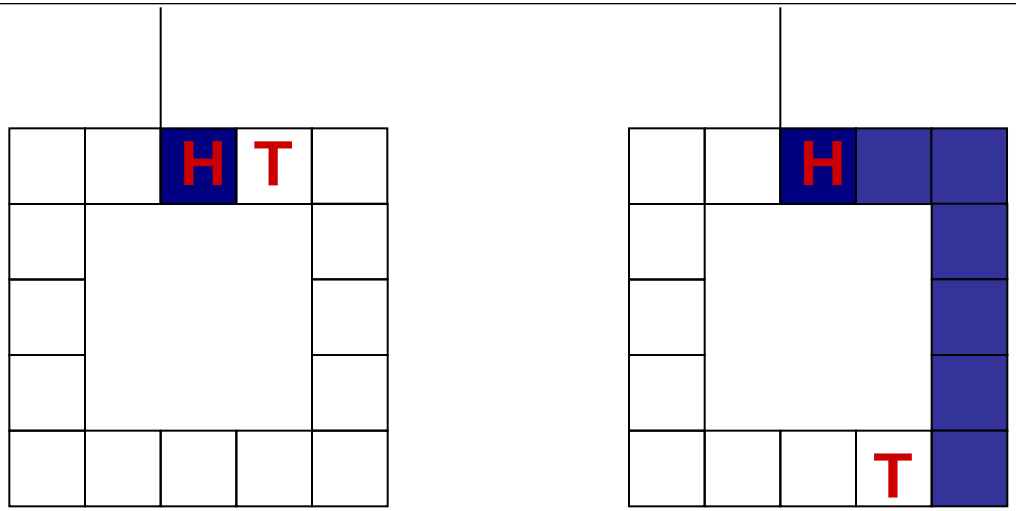
```
public class ArrayQueue {
  private int head;
  private int tail;
  private Object [] data;


...
```

Queue: Methode isEmpty()



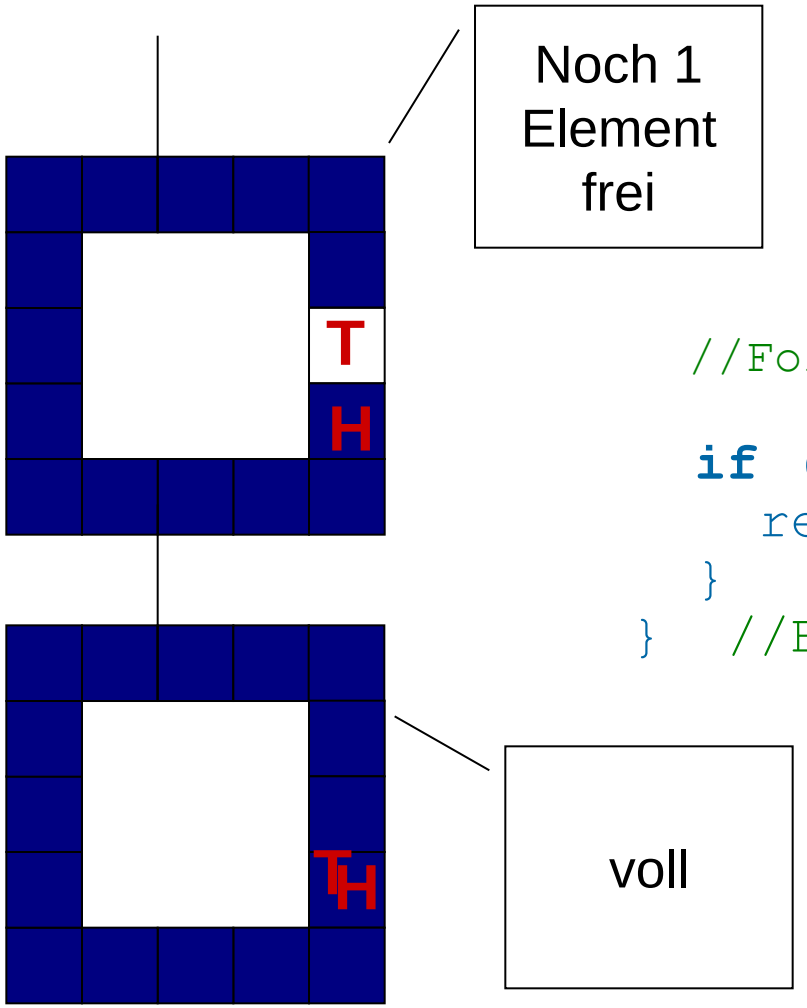

```
private void clear() {
    head = 0;
    tail = 0;
}

public boolean isEmpty(){
    return head == tail;
}
```

Queue: Methode put(..)



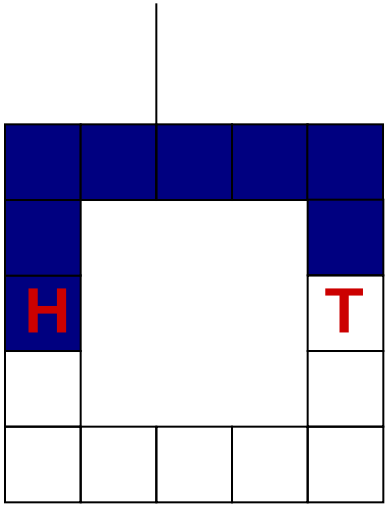

```
public void put(Object o) {
    data[tail] = o;
    tail = (tail+1) % data.length;

    //Fortsetzung naechste Folie
```

Queue: Methode put() (2)




```
        //Fortsetzung von voriger Folie

        if (head==tail) {
           resize();
        }
    }  //Ende der Methode
```

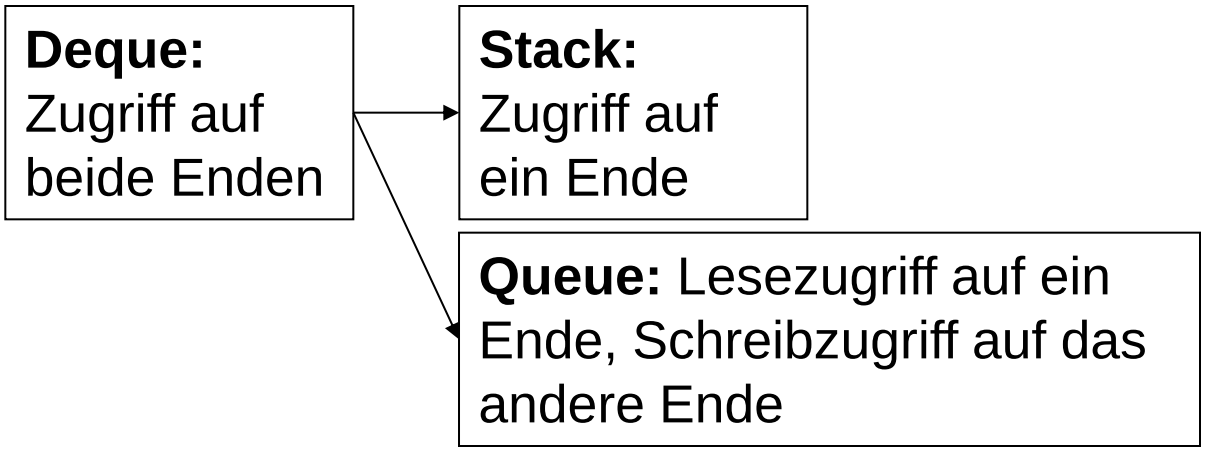
Queue: Methode get()




```
public Object get() throws QueueException {
 if (isEmpty()) {
    throw new QueueException
                    ("Queue ist leer!");
 }
 Object ret = data[head];
 head =(head+1) % data.length;

 return ret;
}
```

ADT Deque

- Deque (sprich: Deck) steht für „Double-Ended Queue“.
- Eine Deque ist eine lineare Datenstruktur, bei der die Daten an beiden Enden eingefügt oder entfernt werden können.



Geeignete Datenstrukturen

- Die geeigneten Datenstrukturen entsprechen denen der Queue.
  - **Es wird langsam dazu übergegangen, statt einer Queue gleich eine Deque zu implementieren.**

ADT Deque in Java

- Ab Java 6.
- Interface java.util.Deque

- Methoden addFirst(), addLast(), removeFirst(), removeLast().
- Es gibt eine Deque als **zirkuläres dynamisches Feld** (java.util.ArrayDeque).
- Es gibt eine Deque als **verkettete Liste** (java.util.concurrent.LinkedList).

2.4. Mengen und assoziative Felder

Abstrakter Datentyp Menge („Set“)

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.

Operationen:

- **Elementare Operationen: Einfügen, Löschen, Testen (ob vorhanden).**
- **Darauf aufbauende Operationen: Typische Operationen aus der Mengenlehre: Schnittmenge, Restmenge, …**

Datenstruktur Menge („Set“)

- Die Datenstruktur Set ist nur in wenigen Programmiersprachen direkt in der Sprache verfügbar (z.B. PASCAL).

- In Java ist *Set* ein Interface, das (unter anderem) folgende Klassen implementieren:
  - **TreeSet** (keine .NET-Entsprechung): Basiert auf der Datenstruktur Rot-Schwarz-Baum, implementiert Erweiterung SortedMap.
  - **HashSet** (.NET: HashSet): Basiert auf der Datenstruktur Hash-Tabelle.

ADT Assoziatives Feld

- Englische Namen: Associative Array, Dictionary, Map

- Sonderform des Feldes
  - **Verwendet keinen numerischen Index zur Adressierung eines Elements (wie a[1]).**
  - **Verwendet zur Adressierung einen Schlüssel (z.B. in der Form a[″Meier″])**

- Operationen
  - **Feldeintrag einfügen**
  - **Feldeintrag auslesen**
  - **Feldeintrag löschen**

Einsatz von Assoziativen Feldern

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

  daten[suchschluessel]=datenelement

- Beispiel: Telefonbuch
  - **Datenelement: (Name, Adresse, Telefonnummer)**
  - **Suchschlüssel: Name**

Assoziative Felder in Java

- ADT entspricht dem Interface java.util.Map

- 2 Implementationen
  - **TreeMap** (.NET: SortedDictionary): Basiert auf der Datenstruktur Rot-Schwarz-Baum, implementiert Erweiterung SortedMap.
  - **HashMap** (.NET: Dictionary): Basiert auf der Datenstruktur Hash-Tabelle.

- Assoziative Felder gibt es auch in vielen anderen Sprachen (gewöhnlich als Hash-Tabelle):
  - **C++ (Map), Python (Dictionary), Ruby (Hash), Perl, PHP, Visual Basic, ...**

Beispiel in Groovy und Java

- Am Beispiel: Telefonbuch als ass. Feld
- Groovy (in C# fast identisch):

```
Map tel = new HashMap();
tel["Mustermann"]=123456;
System.out.println(tel["Mustermann"]);
```

- Java

```
Map tel = new HashMap();
tel.put("Mustermann", 123456);
System.out.println(tel.get("Mustermann"));
```

Anwendungsbeispiel in Java

- Beispiel in Java (Klasse `HashMap`) .

```
...
HashMap<String, Integer> map =
                        new HashMap<String, Integer>();
map.put("Januar",1);
map.put("Februar",2);
map.put("Maerz",3);
System.out.println(map.get("Februar"));   ⇨ 2
```

Beispiel in Java (2)

- Man kann auch prüfen, ob ein String ein Monat ist

```
boolean s = map.containsKey("Januar");
```
⇨ true

Mengen und assoziative Felder

- Ein assoziatives Feld kann immer auch als Menge verwendet werden:
  - **Das Datenelement ist dann ein Dummy-Element.**

Mengen und assoziative Felder (2)

- Eine Menge kann leicht zu einem assoziatives Feld umgewandelt werden.
-
```
public class Element {
   Object key;
   Object data;

   //Zwei Elemente sind gleich, falls
   //die Schluessel gleich sind
   public boolean equals(Element b) {
     return (this.key.equals(b.key);
   }
}
```

Mengen und assoziative Felder (3)

- Diskussion der Datenstrukturen ist für beide ADT gleich.
- Im folgenden werden die Datenstrukturen immer am Beispiel der Menge vorgestellt.
  - **Die Erklärungen sind dann etwas einfacher.**
- Die Hauptanwendungen sind aber assoziative Felder.

- In Java:
  - **HashSet** ⟷ **HashMap**
  - **TreeSet** ⟷ **TreeMap**

Geeignete Datenstrukturen

- Assoziative Felder können sehr unterschiedlich verwirklicht werden:
- Einfache Lösung: Verwendung einer ArrayList:
  - **Einfügen:** `add(element)`
  - **Löschen:** `remove(element)`
  - **Prüfen:** `contains(element)`
  - **Auslesen:** `get(indexOf(element)`
- Aufwendige Lösung: Verwendung einer Datenbank
  - **MySQL, Oracle, Access, …**

Geeignete Datenstrukturen (2)

Wichtige Fragen:

- Welche Datenstrukturen werden an welcher Stelle verwendet?
- Welche Datenstrukturen sind wann geeignet?

- Folgende Datenstrukturen werden untersucht:
  - **Liste, sortierte Liste, AVL-Baum, Rot-Schwarz-Baum, B-Baum, Hashtabelle.**

2.5 Listen und sortierte Listen

Listen als Mengen

- **Einfügen:** `add(element)`
  - Es wird in der Liste gesucht, ob das Element schon vorhanden ist (O(n)). Wenn nein, wird das Element hinten angehängt (O(1)). **O(n)**
- **Löschen:** `remove(element)`
  - Element muss in der Liste gesucht werden (O(n)). Anschließend müssen eventuell die hinteren Elemente umkopiert werden (O(n)). **O(n)**
- **Prüfen:** `contains(element)`
  - Element muss in der Liste gesucht werden. **O(n)**
- **Auslesen:** `get(indexOf(element)`
  - Element muss in der Liste gesucht werden. **O(n)**

- Diese Art der Suche nennt man auch **lineare** oder **sequentielle** Suche. Durchschnittlich werden n/2 Elemente durchsucht (O(n)).

RWTH AACHEN UNIVERSITY

Definition:

Eine Liste mit n Elementen heißt (nach Schlüsseln) *sortierte Liste*, falls gilt:

$\forall\ 1 \leq i \leq n\text{-}1 : \text{Schlüssel}(i) \leq \text{Schlüssel}(i+1)$.

Der wichtigste Unterschied zur unsortierten Liste ist,
dass man ein effektiveres Suchverfahren einsetzen kann:

die **binäre Suche**.

Binäre Suche (Binary Search)

Voraussetzung:

- Liste ist sortiert

Prinzip:

- Greife zuerst auf das mittlere Element zu
- Prüfe, ob der Schlüssel des gesuchten Elements

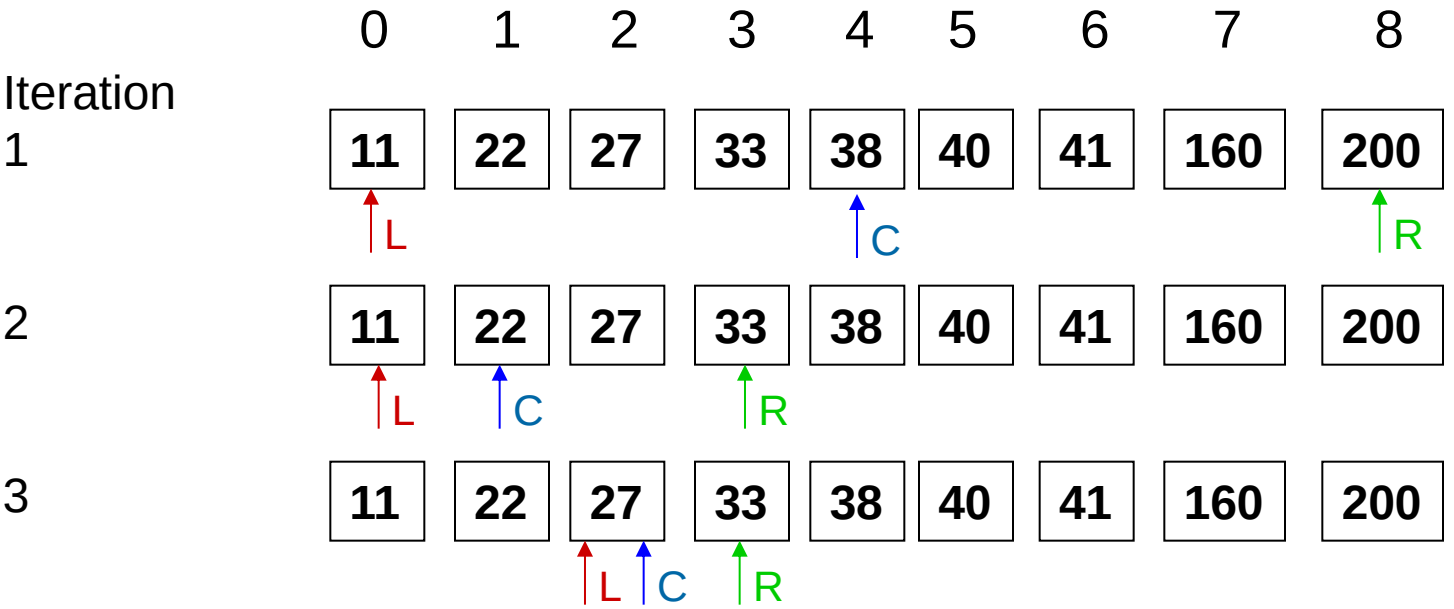
  - kleiner als der Schlüssel des betrachteten Elements ist
    $\Rightarrow$ suche genauso in linker Teilliste weiter

  - gleich dem Schlüssel des betrachteten Elements ist
    $\Rightarrow$ fertig

  - größer als der Schlüssel des betrachteten Elements ist
    $\Rightarrow$ suche genauso in rechter Teilliste weiter

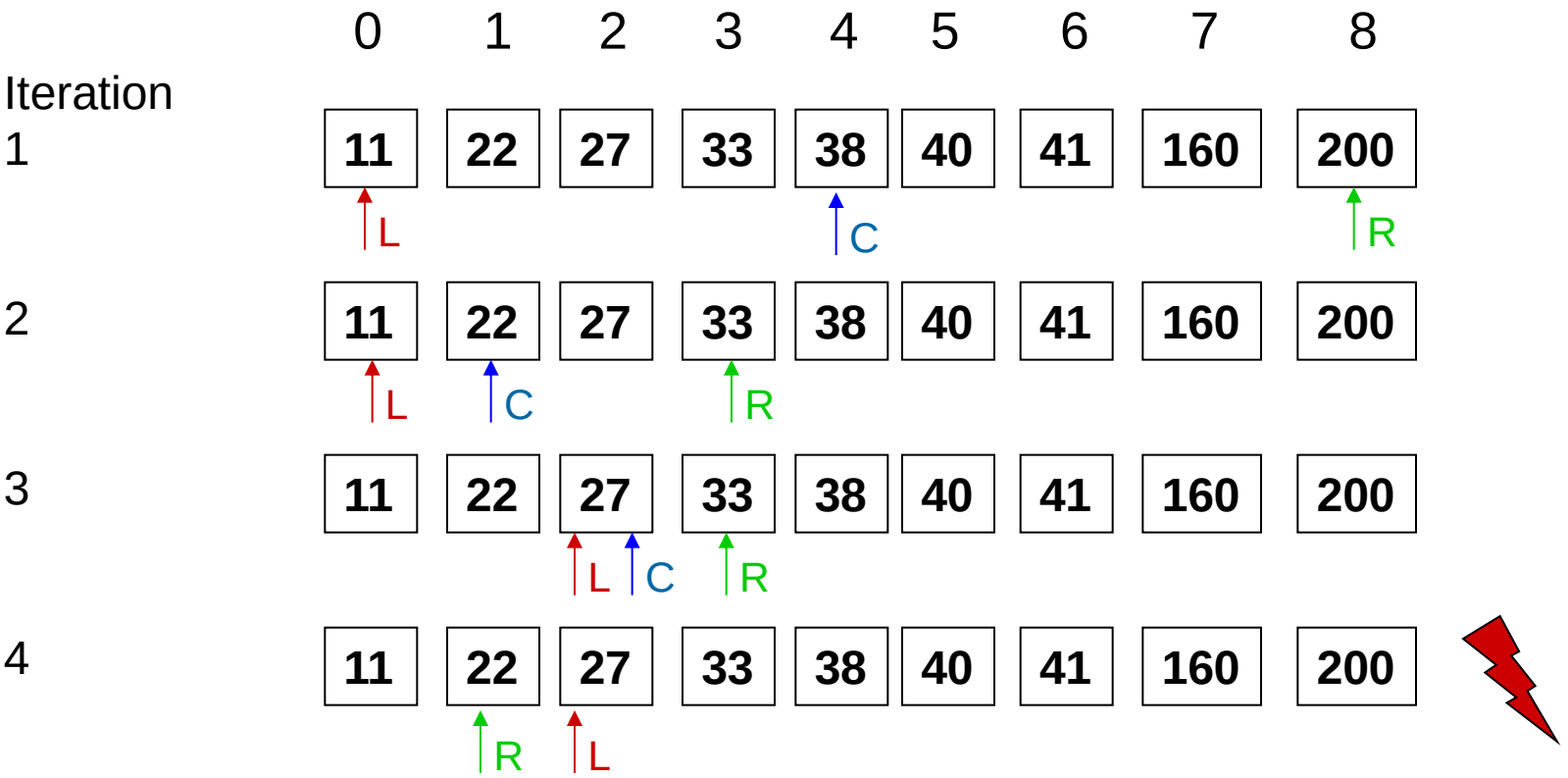
Beispiel zur binären Suche

- Suchen der 27



Beispiel zur binären Suche

- Suchen der 26



Komplexität der Binären Suche

- Bei jedem Suchschritt halbiert sich die Größe des noch zu durchsuchenden Restfelds

  $\Rightarrow$ im Worst Case $\lceil \mathrm{ld}\ (n+1) \rceil$ Suchschritte

- Mit k Suchschritten sind 2^k -1 Elemente erreichbar.

- Nach $\lceil \mathrm{ld}\ (n+1) \rceil$ -1 Schritten ist etwa die Hälfte aller Elemente erreichbar

  $\Rightarrow$ Worst Case nicht wesentlich aufwändiger als Average Case

```
java.util.Arrays.binarySearch(type[] a, type key);
```

type: (Es gibt Varianten für int, long, float, double, …, Object)

Searches the specified array for the specified value using the binary search algorithm. The array **must** be sorted (as by the sort method, above) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

a - the array to be searched.

key - the value to be searched for.

Java - API (2)

Gibt es auch für Collections:

bis Java 1.4:

```
java.util.Collections.binarySearch(List list, Object key);
```

ab Java 1.5 (Verwendung von Generic Types)

```
java.util.Collections.binarySearch(List <? extends
Comparable<? super T>> list, T key);
```

In Java 1.5 stellt die Methode von vornherein sicher, dass *key* mit den Elementen aus *list* vergleichbar ist.
Fortgeschrittene Verwendung von Generic Types, an dieser Stelle keine Erläuterung.
Verwendung wie bei Java 1.4 möglich.

Sortierte Listen als Mengen

		Unsort. Liste
– **Einfügen:**		
• **Element muss gesucht (O (log n)) und anschließend eingefügt werden (O(n)).**	**O(n)**	O(n)
– **Löschen:**		
• **Element muss gesucht (O(log n)) und anschließend gelöscht werden (O(n)).**	**O(n)**	O(n)
– **Prüfen:**		
• **Element muss in der Liste gesucht werden.**	**O(log n)**	O(n)
– **Auslesen:**		
• **Element muss in der Liste gesucht werden.**	**O(log n)**	O(n)

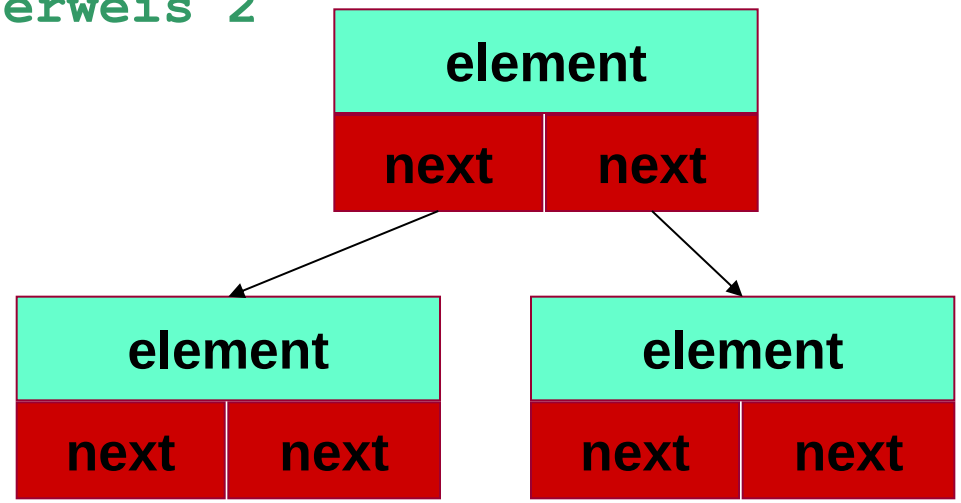
- Günstig, falls oft gesucht wird
  (aber selten eingefügt oder gelöscht)

RWTH AACHEN UNIVERSITY

2.6. Bäume

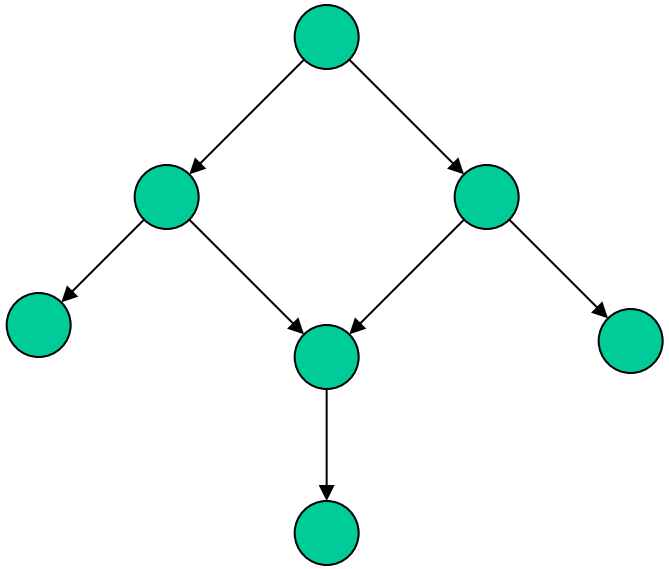
Gibt es in einem Knoten nicht einen, sondern mehrere Verweise, entstehen statt verketteten Listen **Bäume**.

```
class Node {
    Object element;   // Datenkomponente
    Node nextLeft;    // Verweis 1
    Node nextRight;   // Verweis 2

}
```



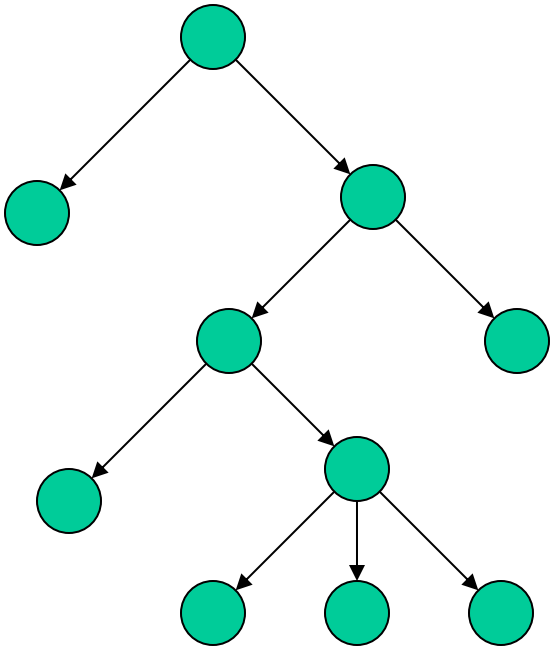
- **Bedingung für Bäume:**
  - **Zwei beliebige Knoten sind durch genau einen (einfachen) Pfad verbunden.**

Graph, aber kein Baum:



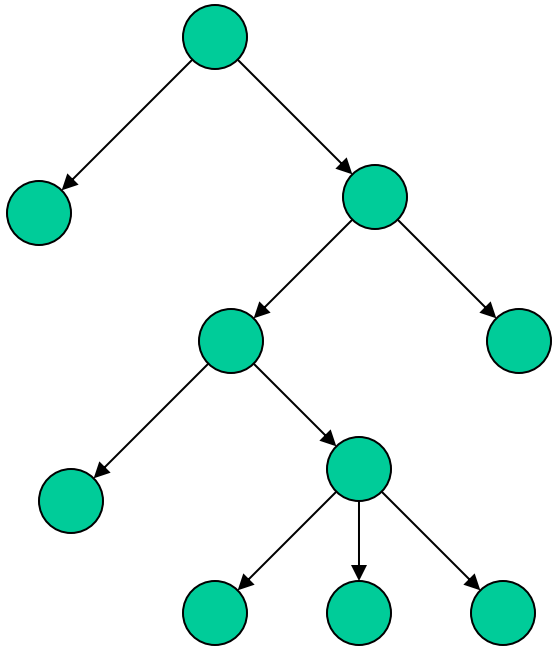
Grundbegriffe zu Bäumen (3)

- Baum = Hierarchische (rekursive) Datenstruktur
  - alle Wege gehen von einer Wurzel aus
  - A heißt Vorgänger von B, wenn A auf einem Weg von der Wurzel zu B liegt. B heißt dann Nachfolger von A.
  - A heißt Vater von B, wenn (A, B) $\in$ E, B heißt Sohn („Kind“) von A.
  - Haben B und C denselben Vater, so heißen sie Brüder.
  - Knoten ohne Söhne heißen Blätter („Terminalknoten“).
  - Knoten mit Söhnen heißen innere Knoten.

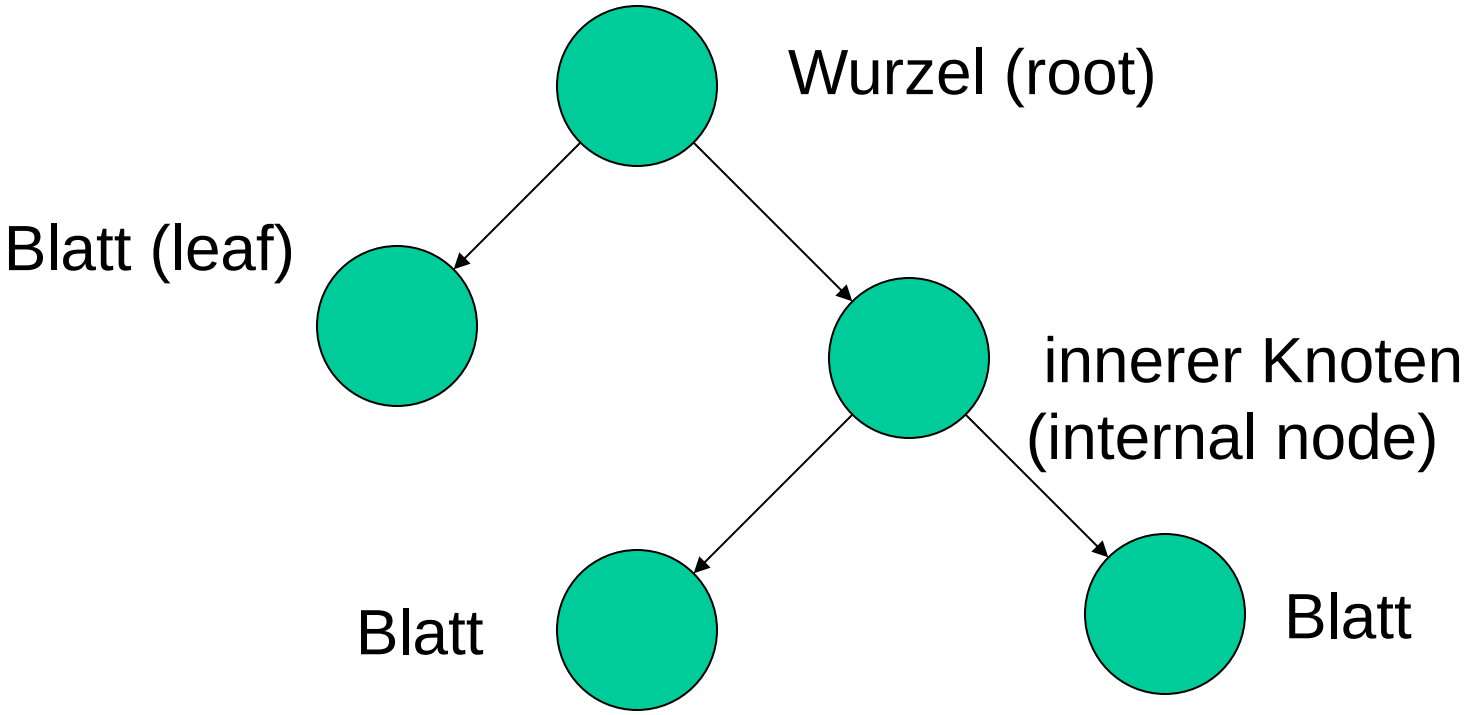


Grundbegriffe zu Bäumen (4)

- Ein Knoten S mit allen Nachfolgern wird Teilbaum eines Baumes T genannt, falls S nicht die Wurzel von T ist.
- Der Verzweigungsgrad eines Knotens ist die Anzahl seiner Kinder

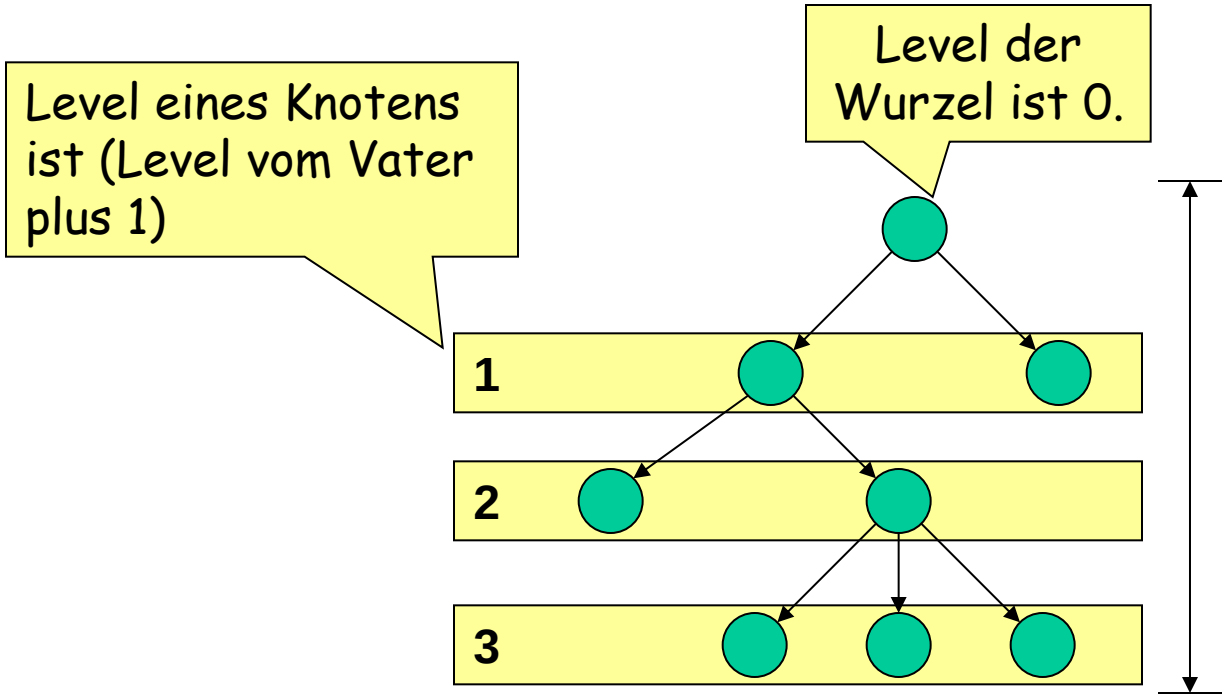


Knotenbezeichnungen an einem Beispiel



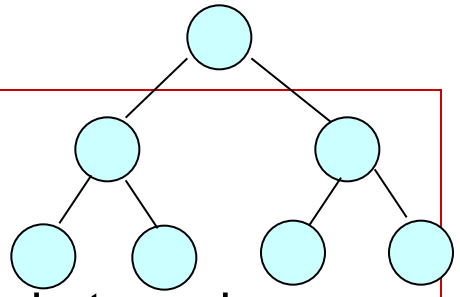
Level und Höhe

Jeder Knoten in einem Baum liegt auf einem bestimmten ***Level*** (Länge des Pfades von der Wurzel zu diesem Knoten).



Die ***Höhe*** eines Baums ist gleich dem maximalen Level eines beliebigen Knotens des Baums.

Begriffe zu Binärbäumen (1)



Definition:

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

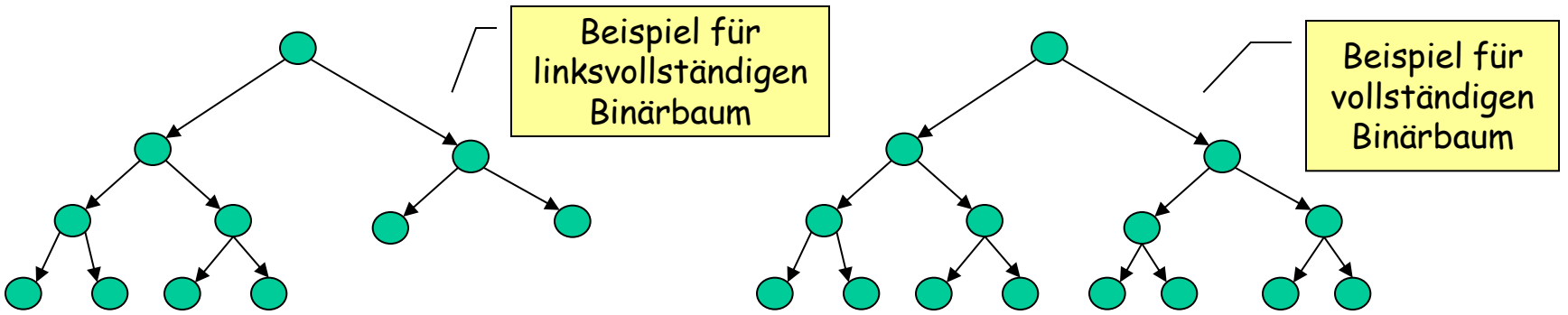
Bei einem ***geordneten Binärbaum*** ist die Reihenfolge der Söhne durch die Indizes eindeutig festgelegt (T_1 = linker Sohn, linker Teilbaum; T_2 = rechter Sohn, rechter Teilbaum).

Definition:

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.



Einführung:

http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

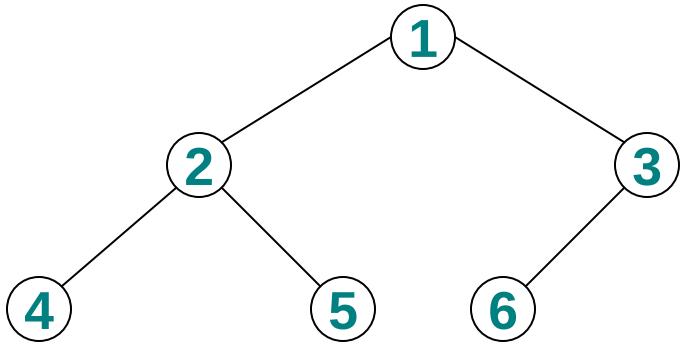
Kapitel Suchbaum (1)

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.

Array-Einbettung

Eignet sich am besten zur Darstellung links-vollständiger Binärbäume, z.B.:

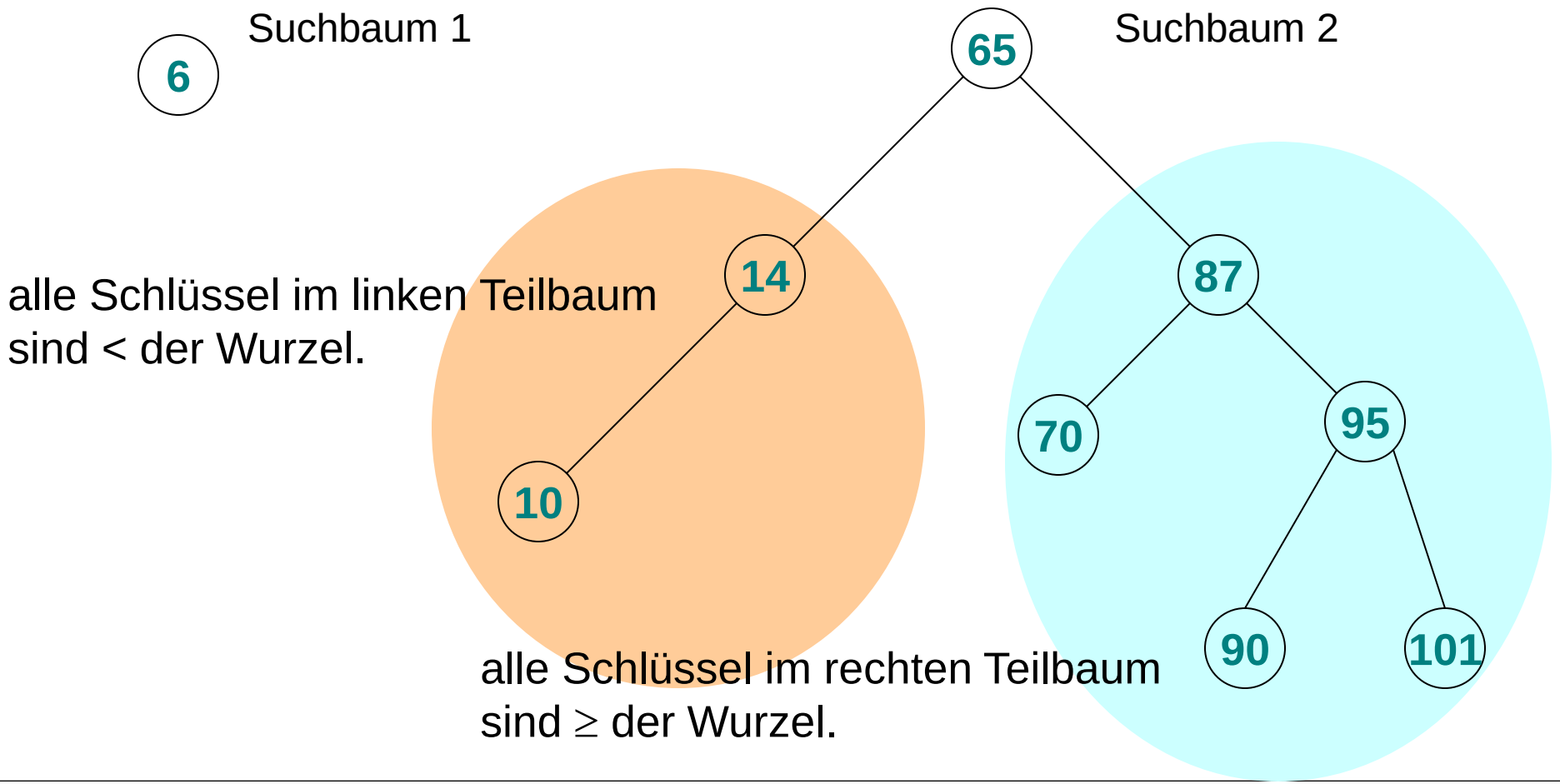


Man nummeriert die Knoten ebenenweise durch. Das Feldelement 0 bleibt frei.
Dann berechnen sich die Positionen von Vorgängern und Nachfolgern so:

$$\text{Vorgänger} : \text{Pred}(n) = \lfloor n/2 \rfloor$$

$$\text{Nachfolger} : \text{Succ}(n) = \begin{cases} 2n & \text{linker Sohn} \\ 2n+1 & \text{rechter Sohn} \end{cases} \quad (\text{falls diese existieren})$$

2.6.2. Binärer Suchbaum



Einführung:

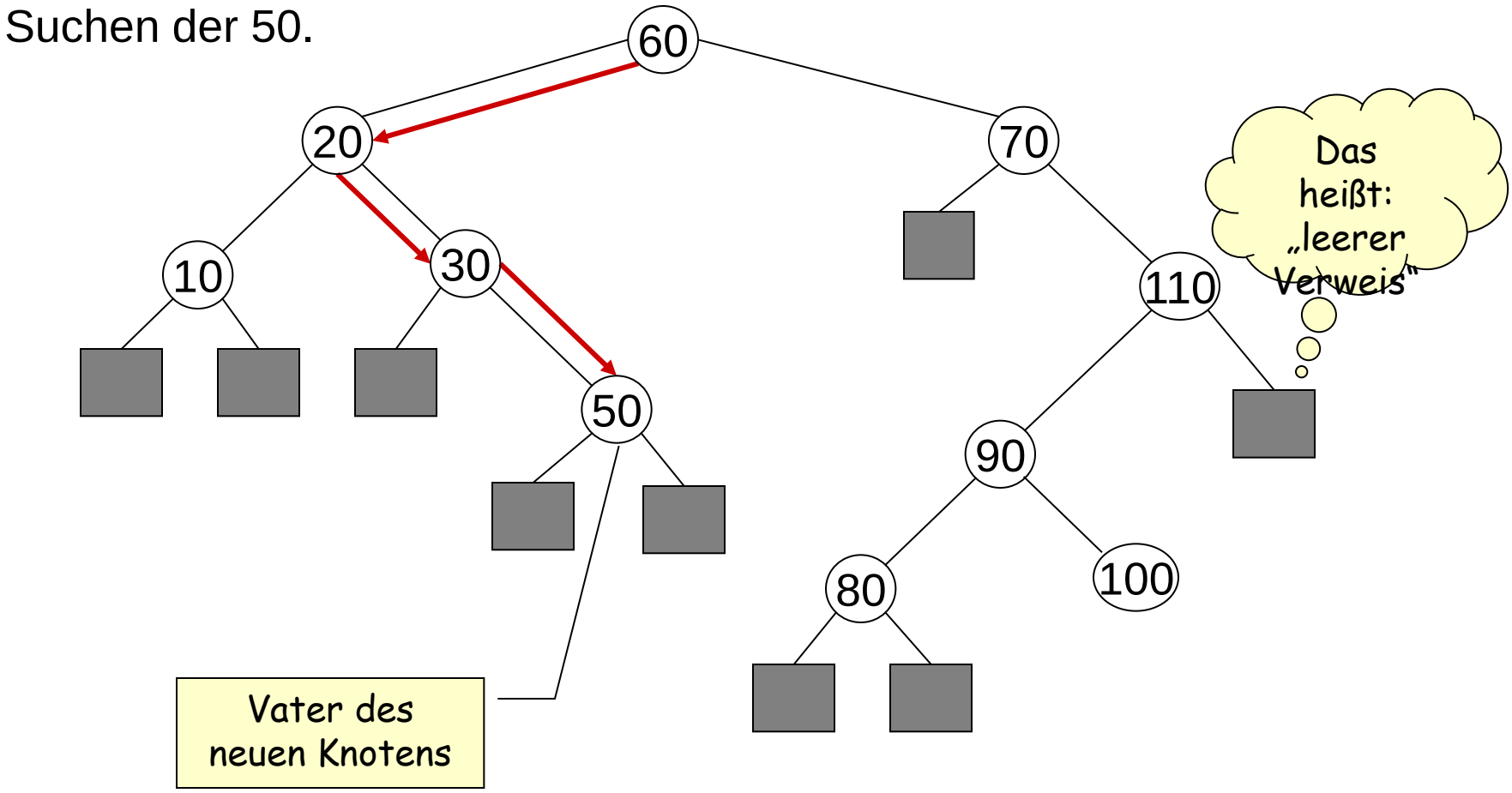
http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

Kapitel Suchbaum (2)

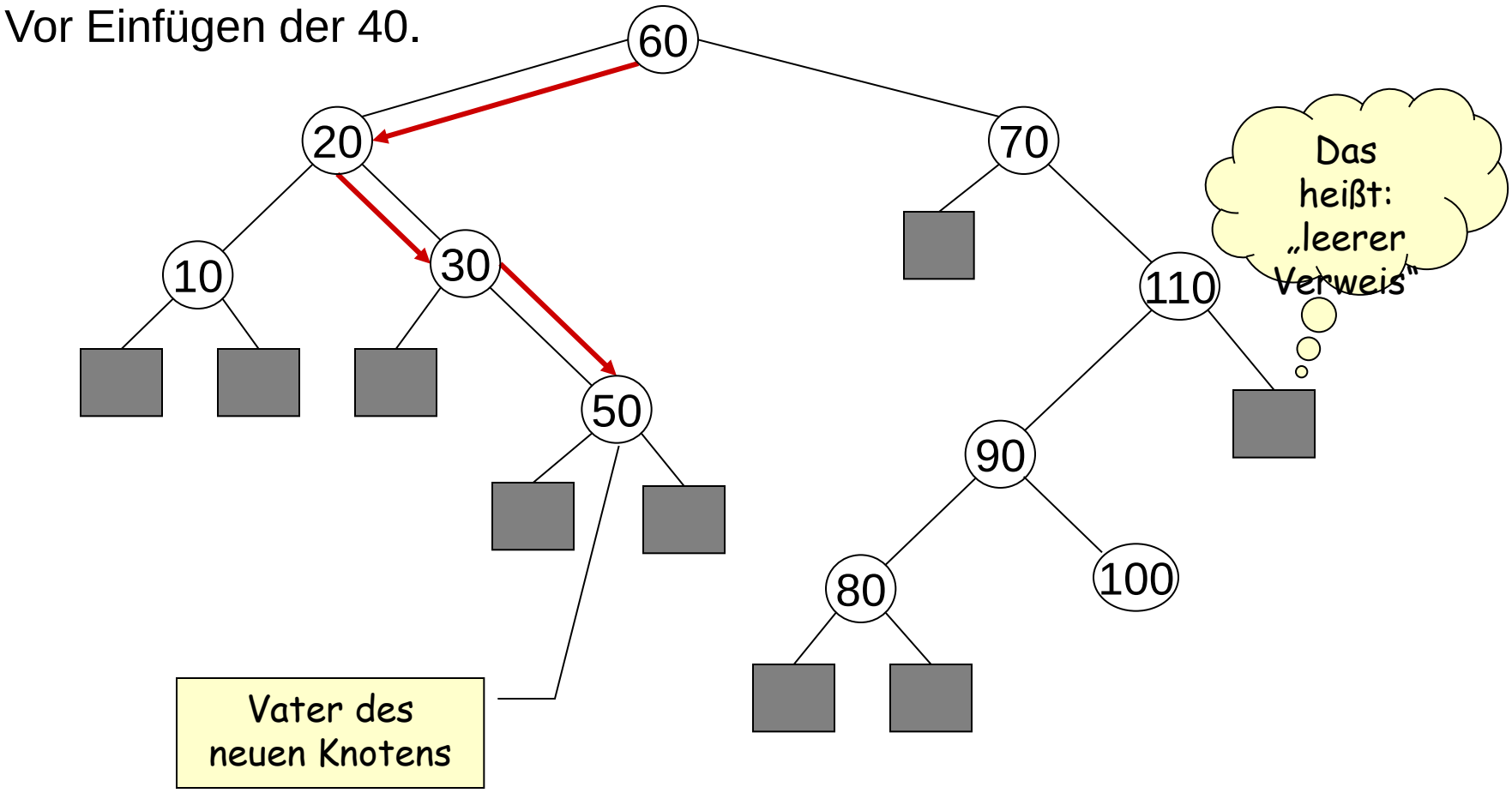
binärer Suchbaum:
Ein binärer Suchbaum ist ein Binärbaum, bei dem für jeden Knoten des Baumes gilt:

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

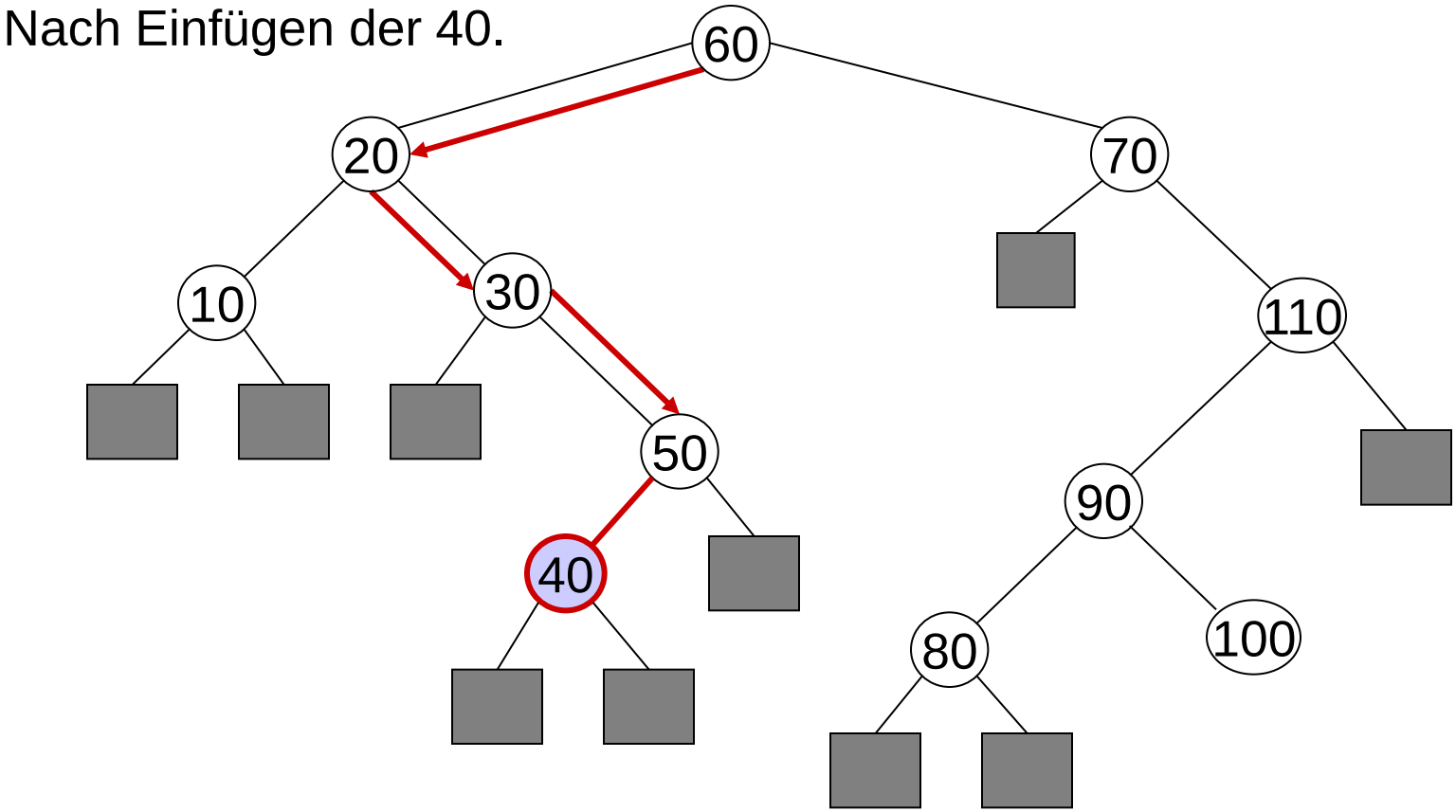
Suchen im Suchbaum



Einfügen im Suchbaum (1)

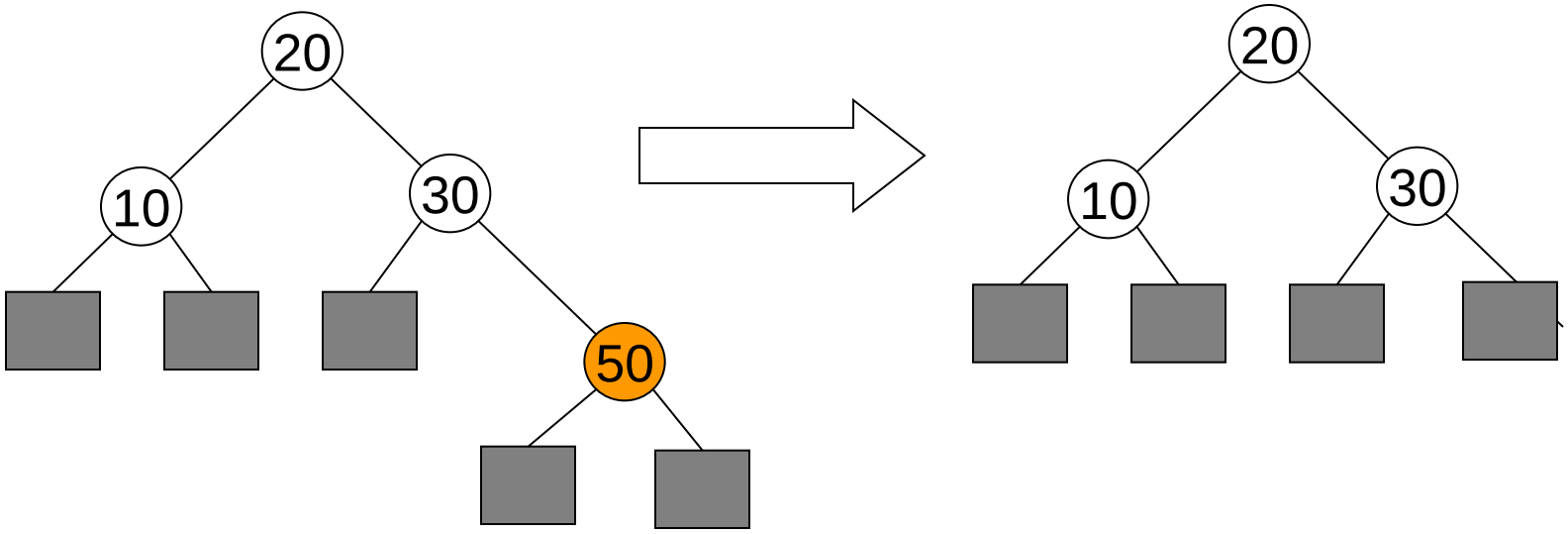


Einfügen im Suchbaum (2)



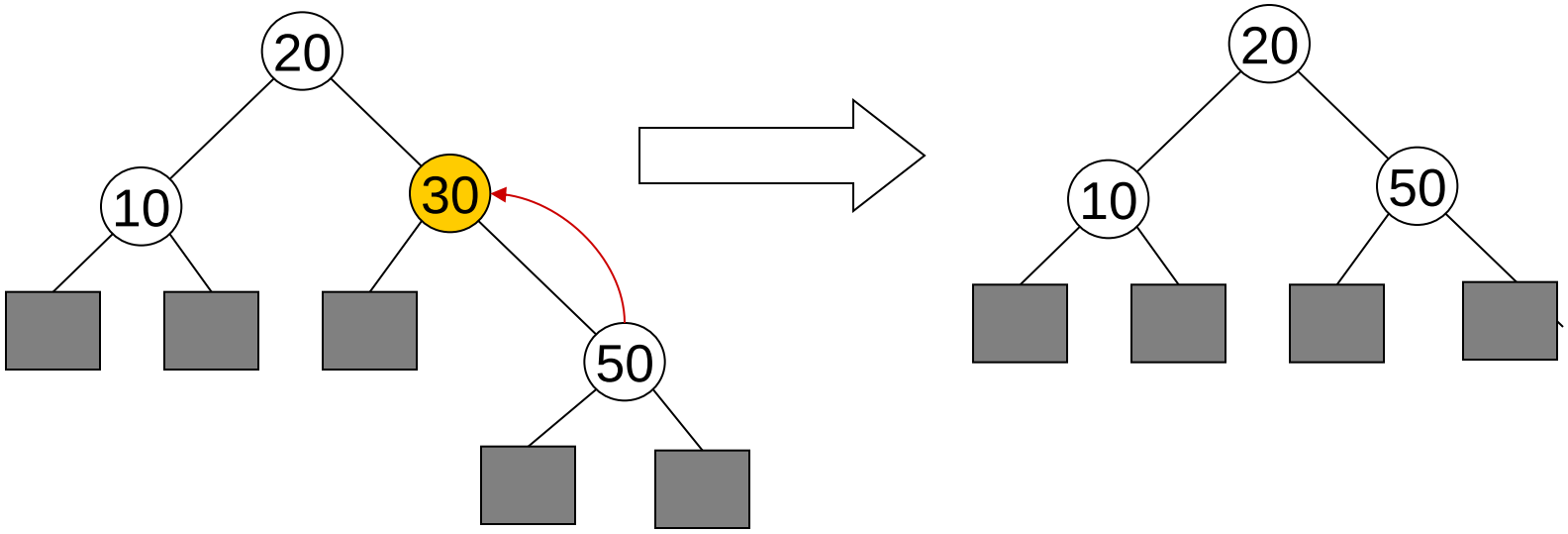
Löschen im Suchbaum (1)

- **Fall 1: zu löschender Knoten ist Blatt (hat keine Kinder)**
- Knoten kann problemlos gelöscht werden
- Beispiel: Löschen der 50



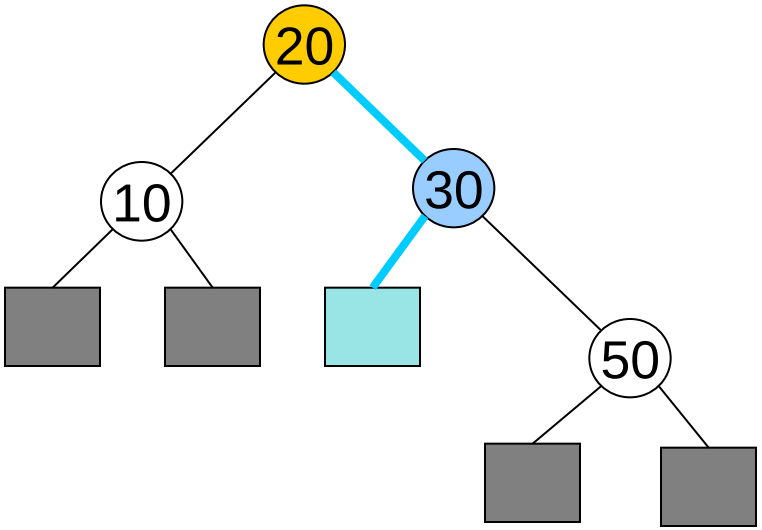
Löschen im Suchbaum (2)

- **Fall 2: zu löschender Knoten hat ein Kind**
- Kind-Knoten rückt an die Stelle des Knotens vor
- Beispiel: Löschen der 30



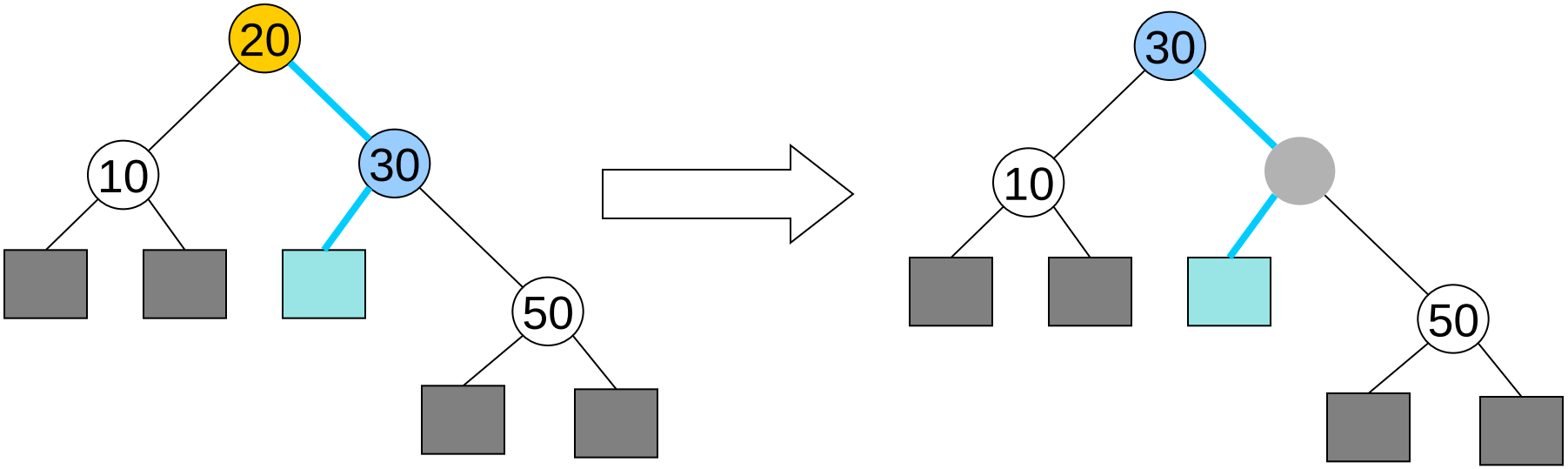
Löschen im Suchbaum (3)

- **Fall 3: zu löschender Knoten hat zwei Kinder**
- Suchen des nächstgrößeren Knoten im rechten Baum
- Erst einen Schritt nach rechts
- Dann solange nach links, bis es links nicht weitergeht.
- Beispiel: Löschen der 20 → nächstgrößerer Knoten: 30



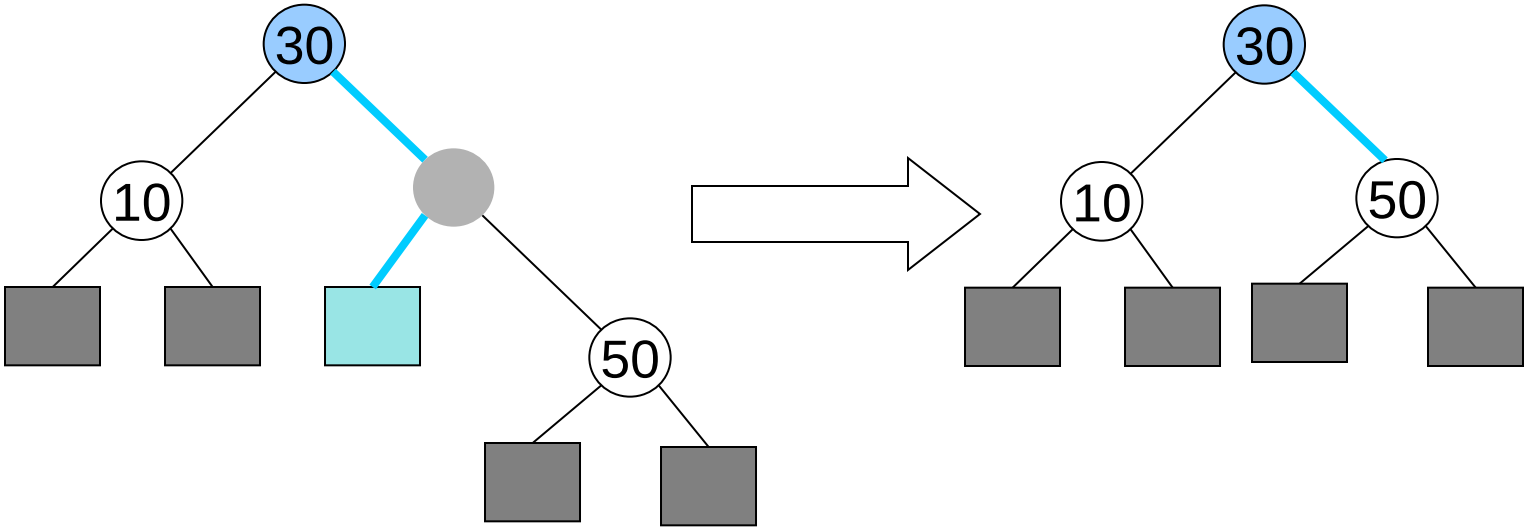
Löschen im Suchbaum (4)

- **Fortsetzung Fall 3**
- Ersetze den zu löschenden Knoten durch den nächstgrößeren.

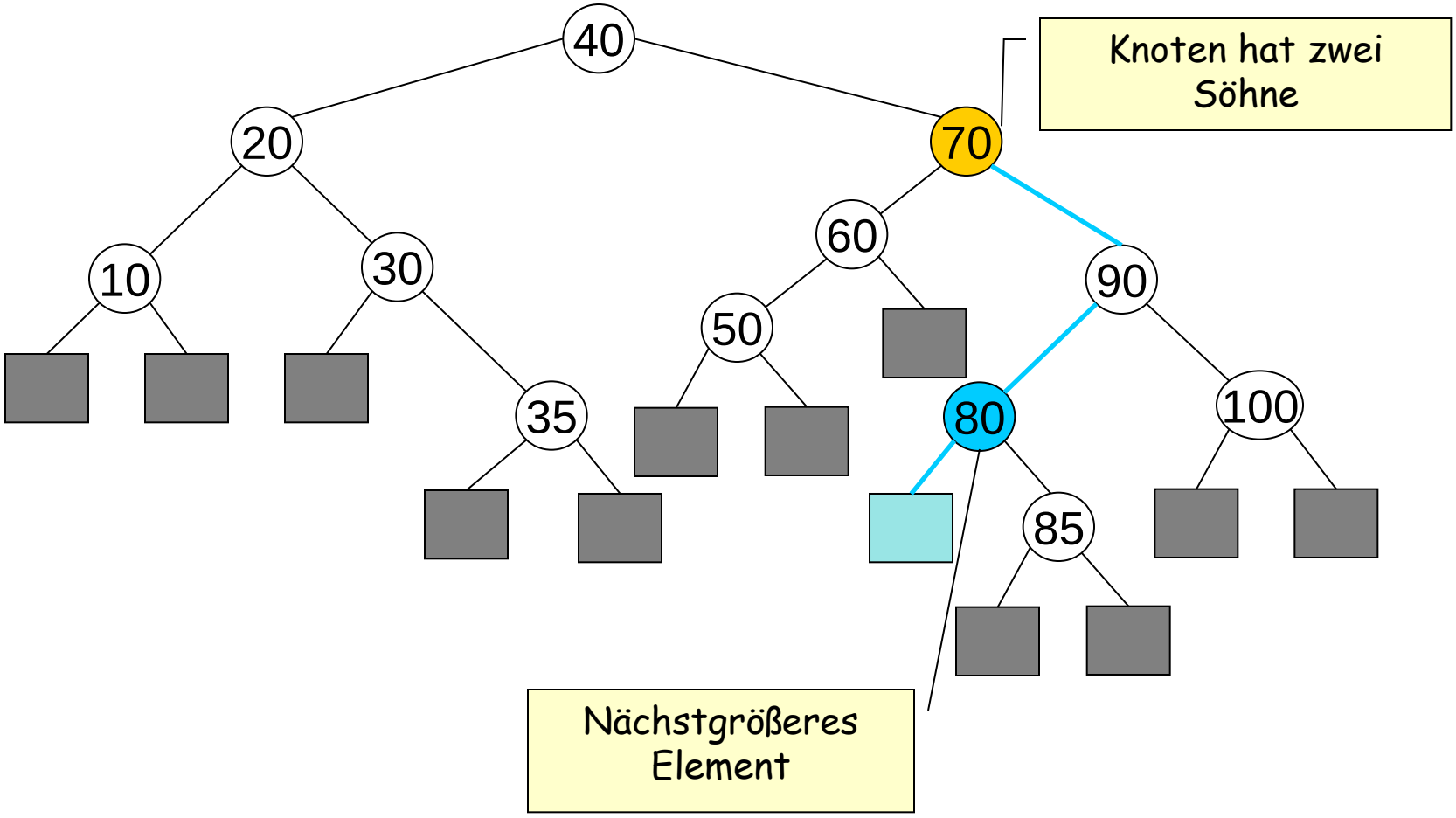


Löschen im Suchbaum (5)

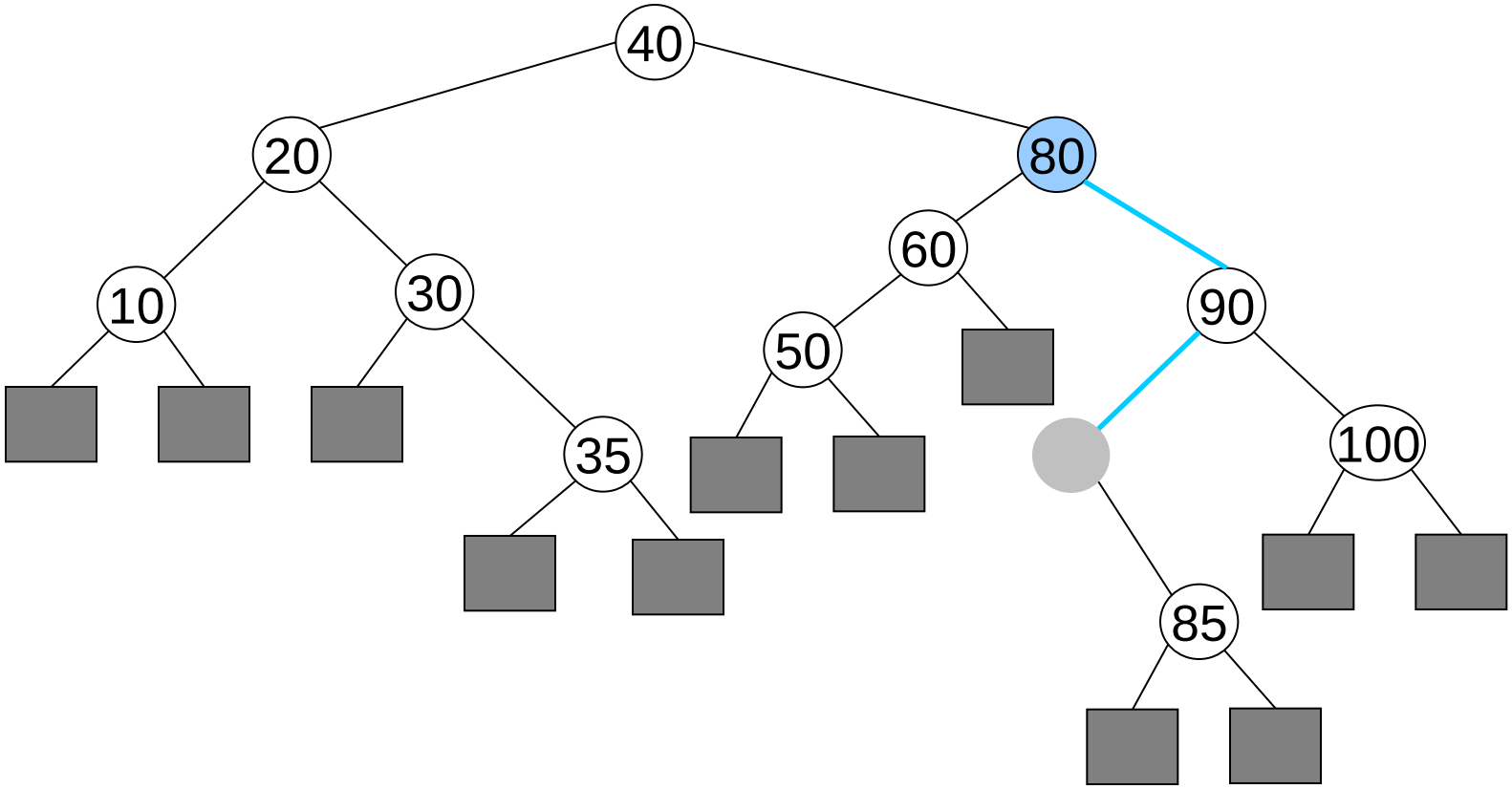
- **Fortsetzung Fall 3**
- Lasse rechte Seite des freien Platzes vorrücken.



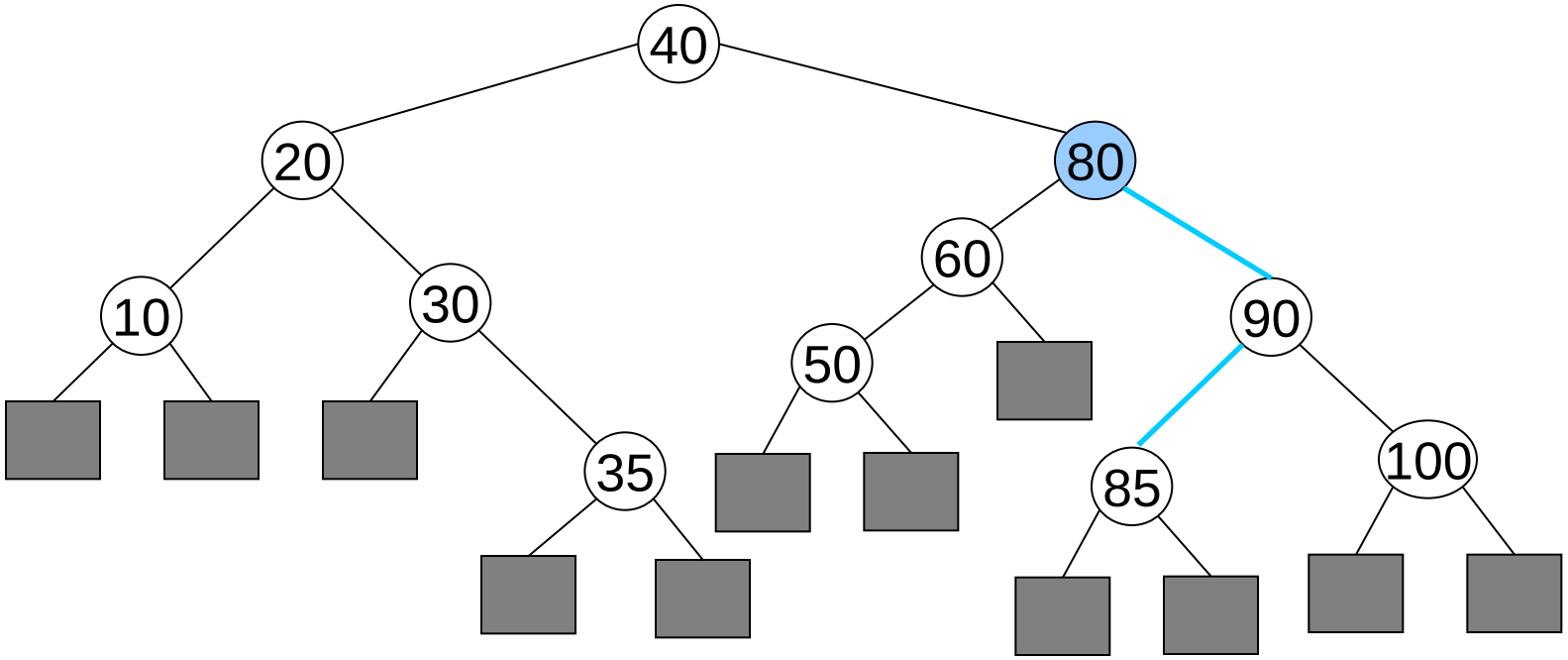
Löschen der „70“ (1)



Löschen der „70“ (2)



Löschen der „70“ (3)



Applet zum Löschen im Suchbaum

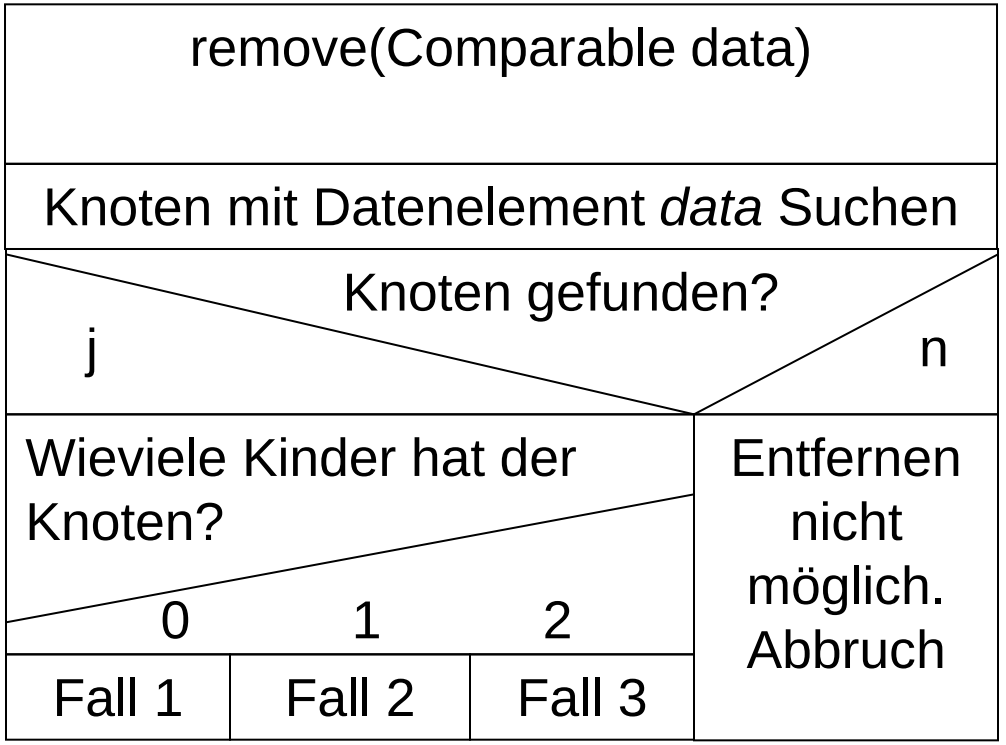
http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

Kapitel Suchbaum (3)

Struktogramm zum Löschen

Datenstruktur:

```
class Node {
  Node parent;
  Node right;
  Node left;
  Comparable data;
}
```



Unterroutinen zum Löschen (1)

Datenstruktur:

```
class Node {
  Node parent;
  Node right;
  Node left;
  Comparable data;
}
```

CopyNode: Kopiert Daten und Referenzen zu den Nachfolgern von Node n1 in Node n2

```
public void copyNode
                (Node n1, Node n2) {
    n2.data = n1.data;
    n2.left = n1.left;
    n2.right = n1.right;

}
```

copyData:
Kopiert *data* von Node n1 in Node n2

```
public void copyData(Node n1, Node n2) {
  n2.data = n1.data;
}
```

Unterroutinen zum Löschen (2)

clearNode(Node n):
Löscht n aus dem Baum. Darf nur aufgerufen werden, wenn n ein Blatt ist.

```
public void clearNode(Node n) {
  if (n==root) {
    root = null;
    return;
  }
  if (n.parent.left == n) {
    n.parent.left = null;
  } else {
    n.parent.right = null;
  }
}
```

Löschen: Fall 1 und 2

- Zu löschender Node: Node n
- **Fall 1: keine Kinder**

```
clearNode(n)
```

- **Fall 2: ein Kind**

```
if (n.left==null) {
  copyNode(n.right, n);
} else {
  copyNode(n.left, n);
}
```

Löschen (Fall 3)

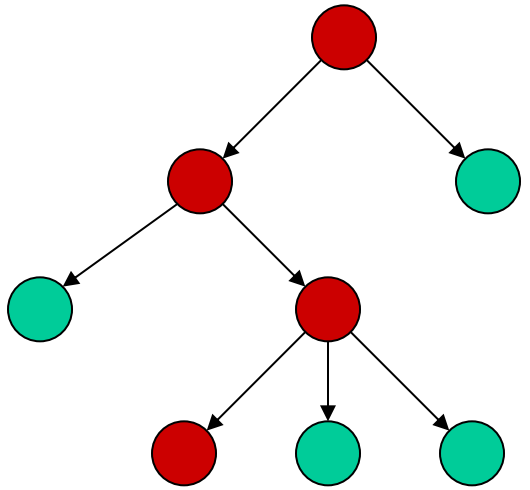
- **Fall 3: zwei Kinder**

```
//minimales Element auf der rechten Seite suchen
Node minR = n.right;
while (minR.left != null) {
  minR = minR.left;
}
//n durch minR ersetzen
copyData(minR, n);
//rechte Seite von minR nach minR schieben
if (minR.right!=null) {
  copyNode(minR.right, minR);
} else {
  clearNode(minR);
}
```

Komplexität

- 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 Baums+1.

- Die Höhe hängt stark von der Reihenfolge der Einfüge-Operationen ab.



Applet zur Höhe von Suchbäumen

- Einführendes Applet:

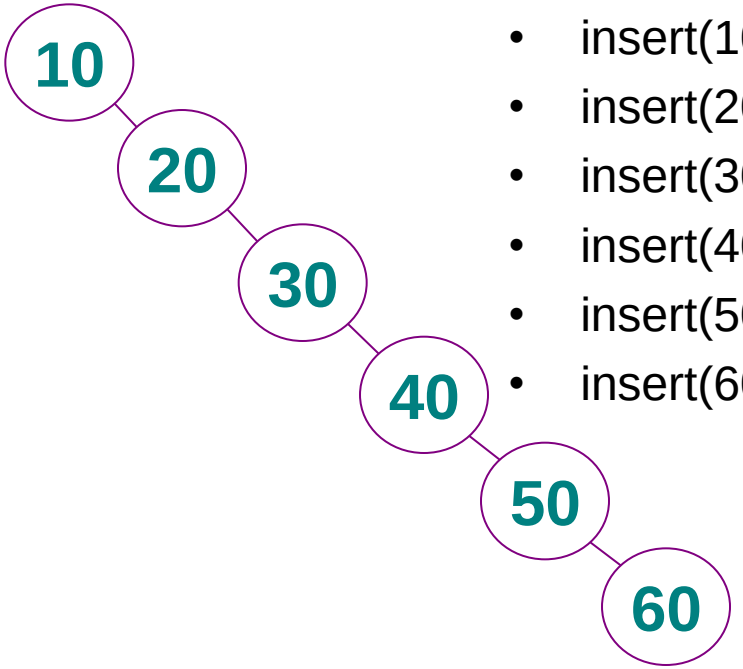
http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

Kapitel AVL (1)

Einfügen in einen binären Suchbaum

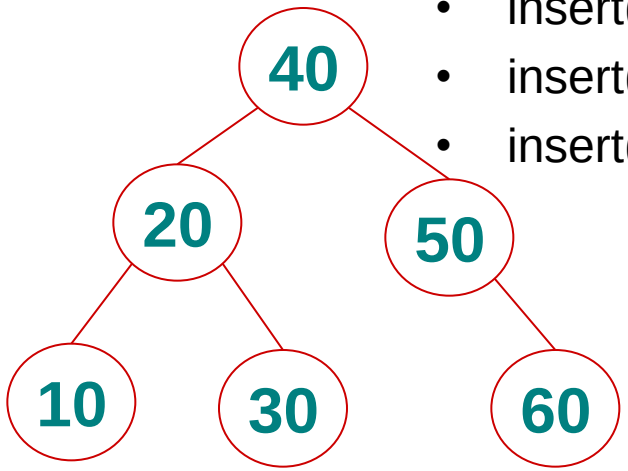
Beispiel 1:

- create()
- insert(10)
- insert(20)
- insert(30)
- insert(40)
- insert(50)
- insert(60)



Beispiel 2:

- create()
- insert(40)
- insert(20)
- insert(50)
- insert(10)
- insert(30)
- insert(60)



- Balanciertheit (Ausgeglichenheit) der Knoten-Verteilung nicht garantiert
- bei ungünstiger Reihenfolge der Einfügeoperationen können zu linearen Listen entartete Bäume entstehen

Minimale und maximale Höhe

- Ein vollständig gefüllter Binärbaum (voller Baum, in dem die letzte Ebene voll besetzt ist) der **Höhe H** hat
  - n = 1+2+4+…+ 2^H = (**2^{H+1}-1**)/(2-1) **Knoten**, (geometrische Reihe)
- Ein Binärbaum mit n Knoten hat im **besten Fall** (optimal balanciert) die Höhe $\lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2(n) \rfloor$

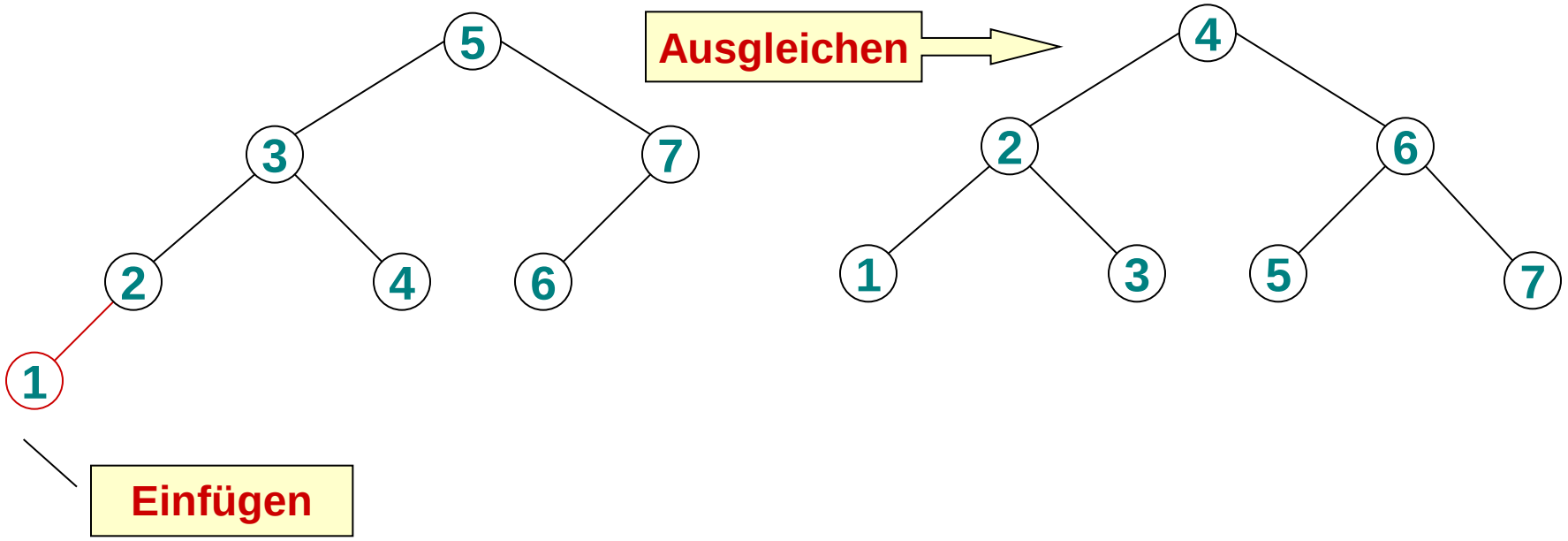
  ⇒ **Suchen ist O(log n)**
- Ein Binärbaum mit n Knoten hat im **schlechtesten Fall** („entarteter“/„degenerierter“ Baum) die Höhe n-1.

  ⇒ **Suchen ist O(n)**

Wie kann Entartung verhindert werden?

- Eine Möglichkeit: Baum nach jedem insert/remove durch Umstrukturierung ausgleichen.

  $\Rightarrow$ kann aufwändig sein, z.B.:



Balance-Kriterien (1)

(mindestens) 3 Lösungsideen (Balance-Kriterien):

1. abgeschwächtes Kriterium für ausgeglichene Höhe
   $\Rightarrow$ **lokale Umordnungsoperationen reichen aus.**

- **Verschiedene Varianten, die jeweils unterschiedliche Kriterien haben:**
  - **AVL-Bäume: Werden ausführlich behandelt.**
  - **Rot-Schwarz-Bäume: Werden in der Java-Klassenbibliothek benutzt. Ähnlichkeiten mit B-Bäumen (siehe 3. Lösungsidee). Kurze Vorstellung im Anschluss an B-Bäume.**

Balance-Kriterien (2)

2. Jeder neue Knoten wandert an die Wurzel des Baums.
   - **Vorteil: Zuletzt eingefügte Elemente lassen sich schneller finden.**
   - **Durch ein spezielles Einfügeverfahren wird der Baum zusätzlich (teilweise) ausgeglichen.**
- **Splay-Bäume**: Werden in der Vorlesung nicht weiter behandelt.

Balance-Kriterien (3)

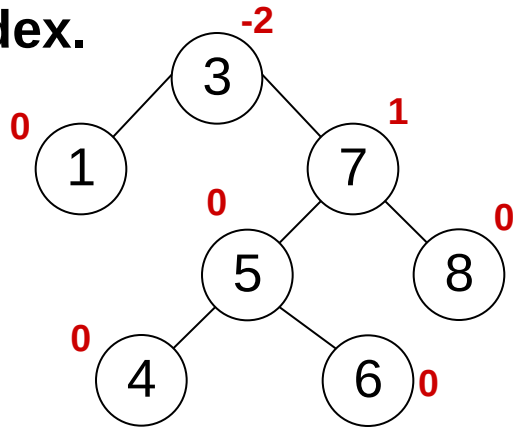
3. Unausgeglichener Verzweigungsgrad ermöglicht ausgeglichene Höhe.

- **B-Bäume**:
  - B-Bäume besitzen **ausgeglichene Höhe**, lassen aber **unausgeglichenen Verzweigungsgrad** zu.
  - Varianten von B-Bäumen werden speziell in Datenbanksystemen als Indexstrukturen eingesetzt.
  - Werden ausführlich behandelt.

- Bei einem AVL-Baum **unterscheiden sich die Höhen zweier Teilbäume des gleichen Knotens maximal um 1**.
  - **Der sogennante Balance-Index ist die Differenz**

    Höhe(linker Teilbaum) – Höhe(rechter Teilbaum)

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



Applet zum Balance-Index

- http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

  Kapitel AVL (2)

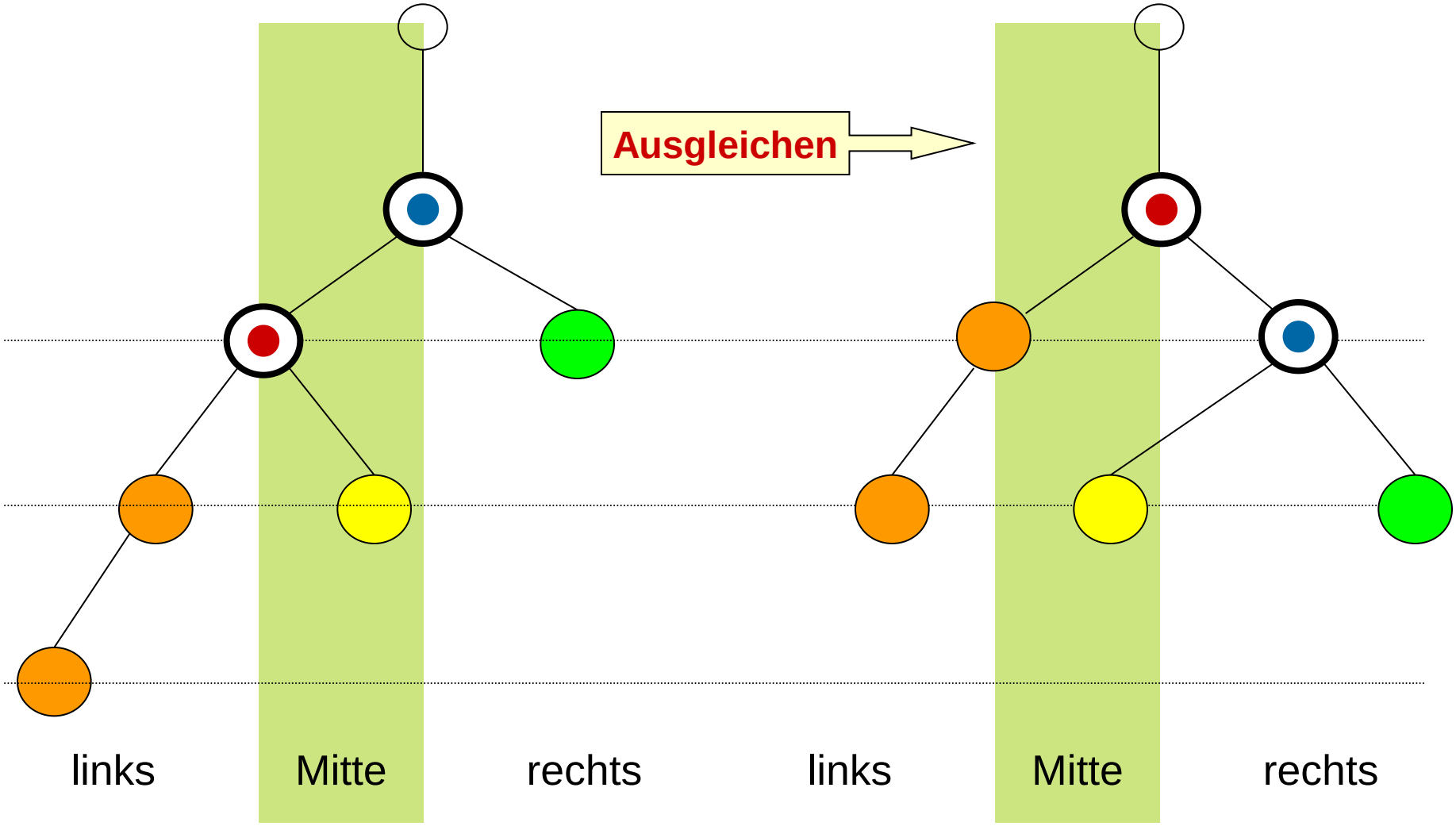
Ausgleichen im AVL-Baum

Wenn durch eine Einfüge- oder Lösche-Operation die AVL-Bedingung verletzt wird, muss (mit lokalen Operationen) rebalanciert (ausgeglichen) werden.

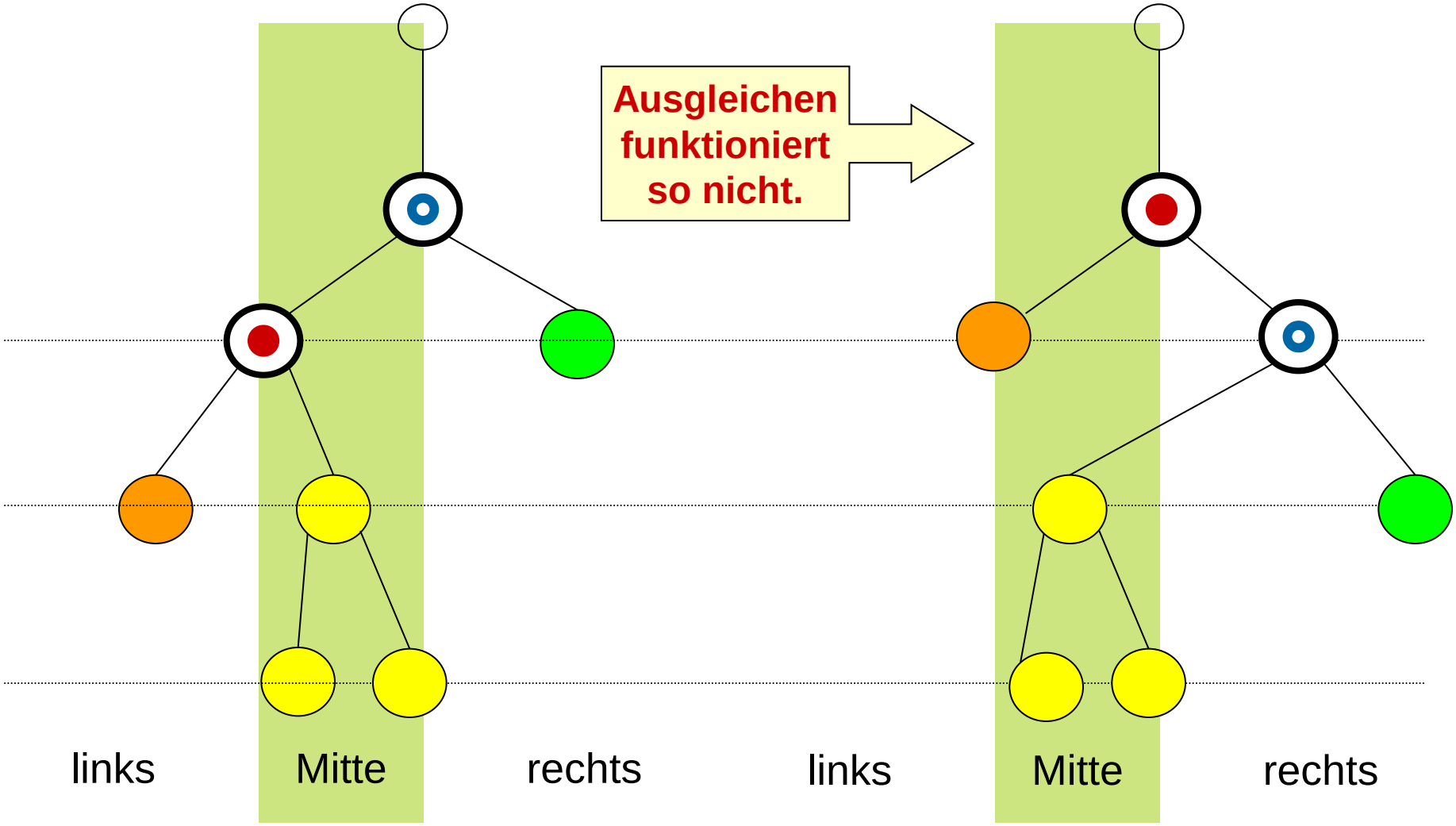
Je nach Situation wendet man dazu entweder eine Rotation oder eine Doppelrotation an.

- **Wie die Rotationen aussehen, wird auf den folgenden Folien erklärt, ist aber nicht klausurrelevant.**

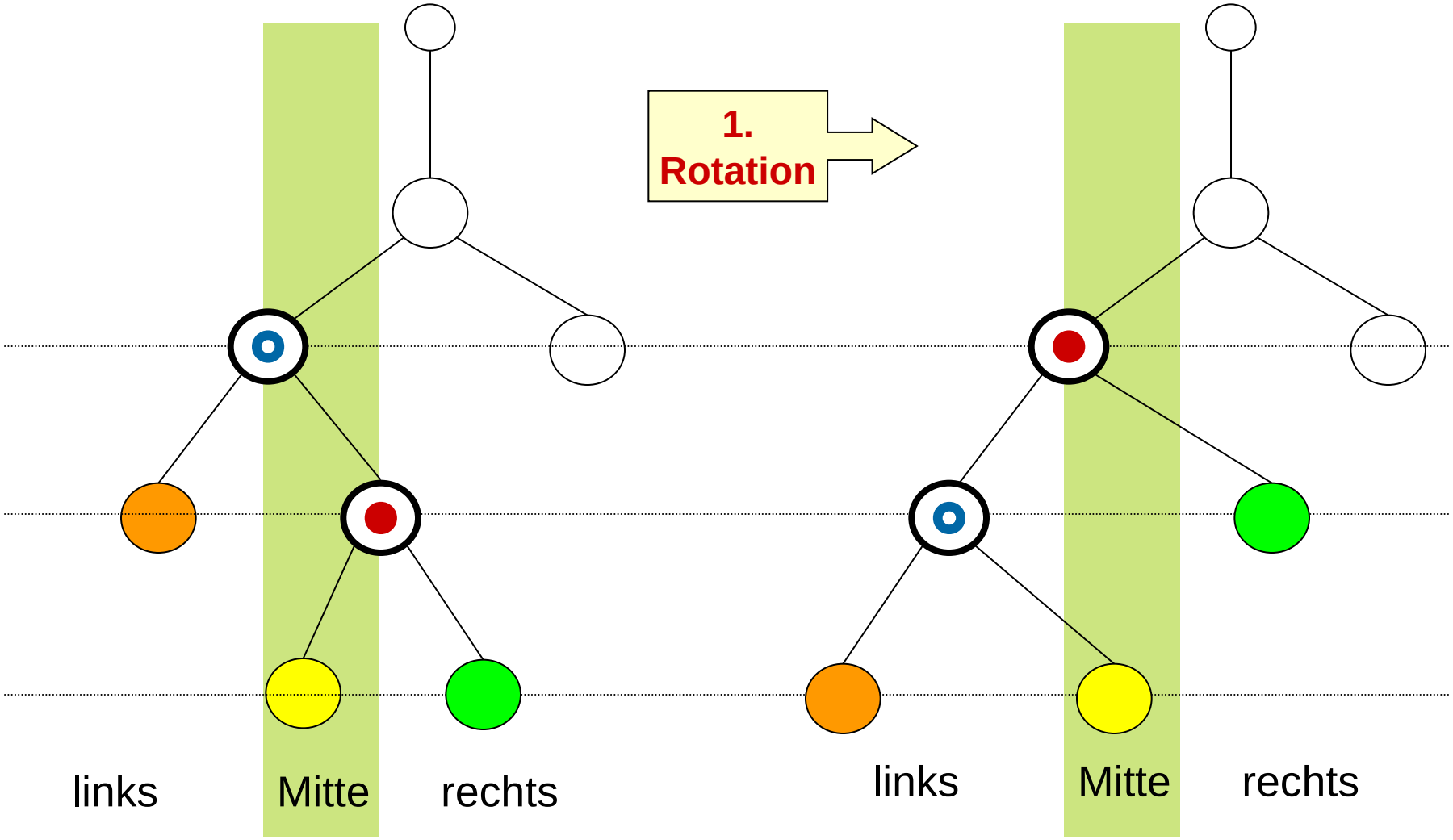
Ausgleichen im AVL-Baum (eine Rotation)



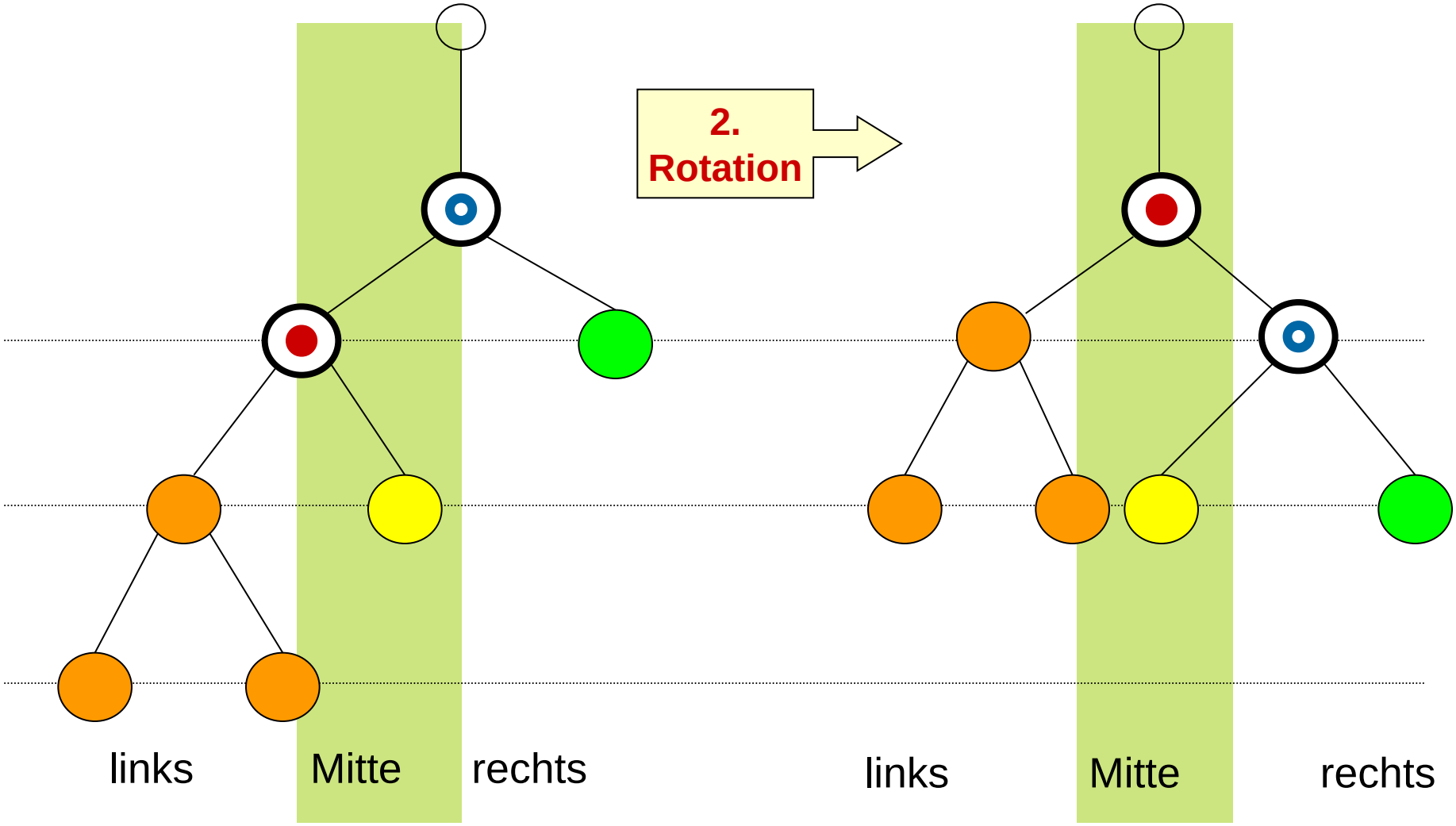
Ausgleichen im AVL-Baum (eine Rotation)



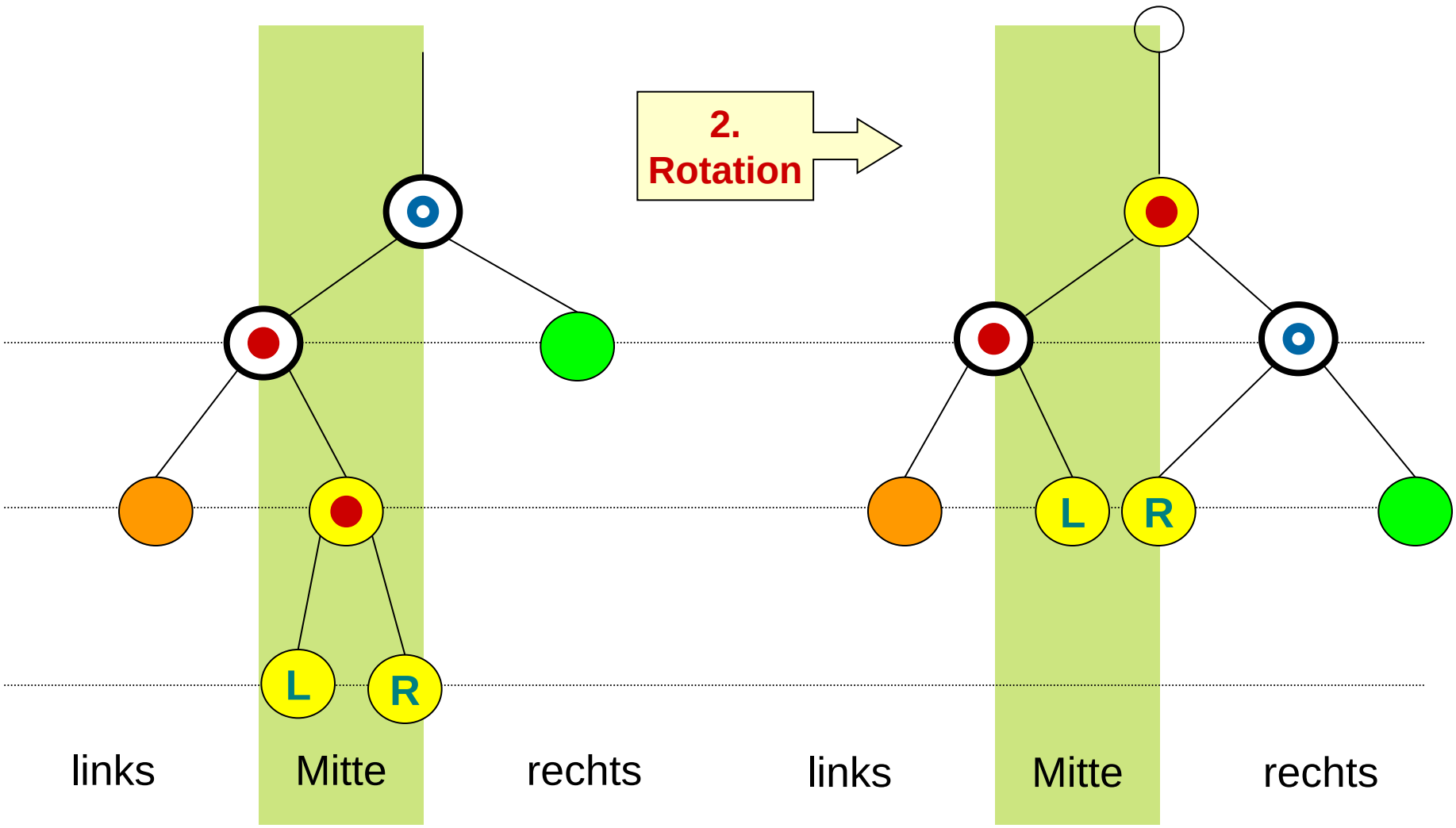
Ausgleichen im AVL-Baum (Doppelrotation)



Ausgleichen im AVL-Baum (Doppelrotation)

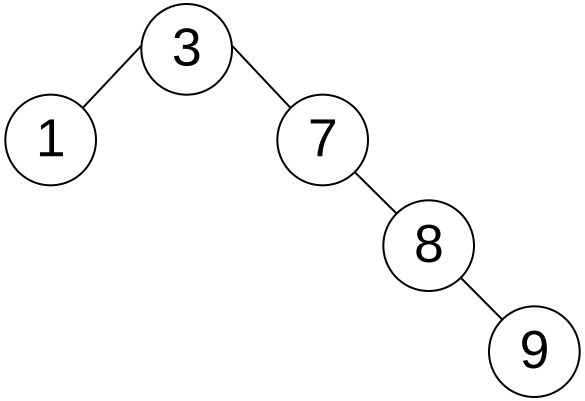


Ausgleichen im AVL-Baum (Doppelrotation)

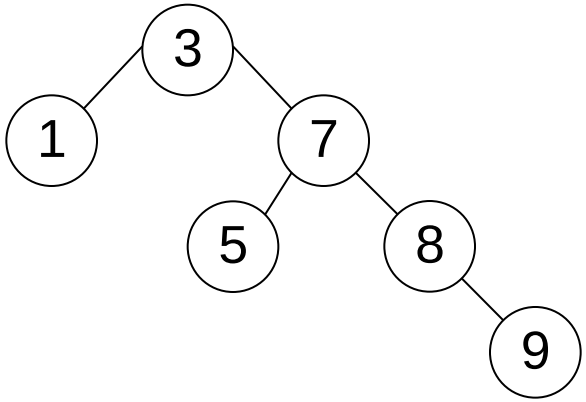


Gleiche die folgenden Bäume durch Rotationen aus

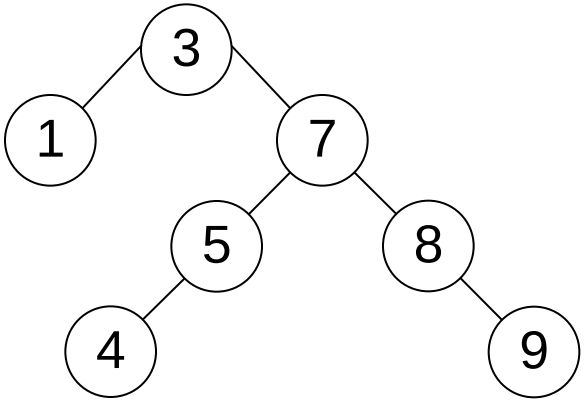
a)



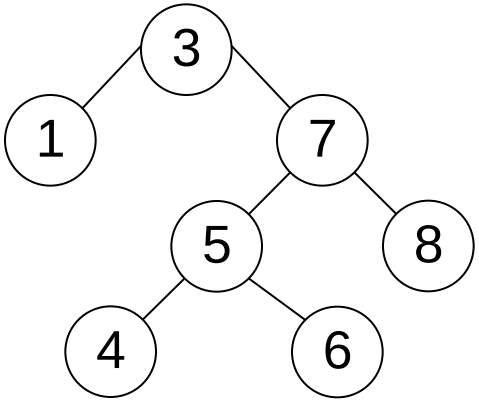
b)



c)



d)



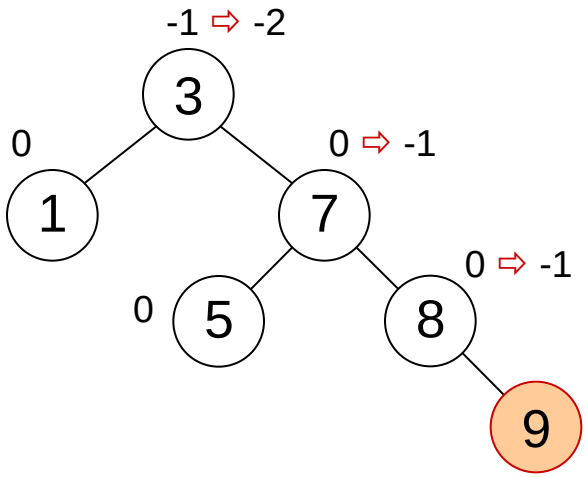
Applets zum Rotieren von AVL-Bäumen

- http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

  Kapitel AVL (3) und AVL (4)

Komplexität

- Die Höhe eines AVL-Baums ist um maximal 45% höher als die eines optimal balancierten Baums.
  - **Suchoperation hat O(log n).**
- Nach dem Einfügen wird maximal um alle Knoten zwischen der Wurzel und dem eingefügten Knoten rotiert.
  - **Ähnlich beim Löschen.**
  - **Ausgleichen nach dem Einfügen oder Löschen hat O(log n).**



AVL-Bäume als Mengen

- **Einfügen:**
  - **Element muss gesucht werden (O(log n)).**
    Element muss angehängt werden (O(1)).
    Baum muss ausgeglichen werden (O(log n)). **O(log n)**
- **Löschen:**
  - **Element muss gesucht werden (O(log n)).**
    Das nächstgrößere Element muss gesucht werden (im worst case O(log n)).
    Elemente müssen umkopiert werden (O(1)).
    Baum muss ausgeglichen werden (O(log n)). **O(log n)**
- **Prüfen/Auslesen:**
  - **Element muss im Baum gesucht werden.** **O(log n)**

- B-Bäume wurden 1972 (1969?) von Rudolf Bayer und Edward M. McCreight eingeführt

- Ziel: effiziente Indexstrukturen für riesige Datenbestände (z.B. bei Datenbanken), die überwiegend auf externen Datenträgern (z.B. auf Festplatten) gespeichert sind.
- Viele Datenbanken (z.B. MySQL, Oracle, Access) benutzen (als Default) B-Bäume (bzw. eine Abart davon).

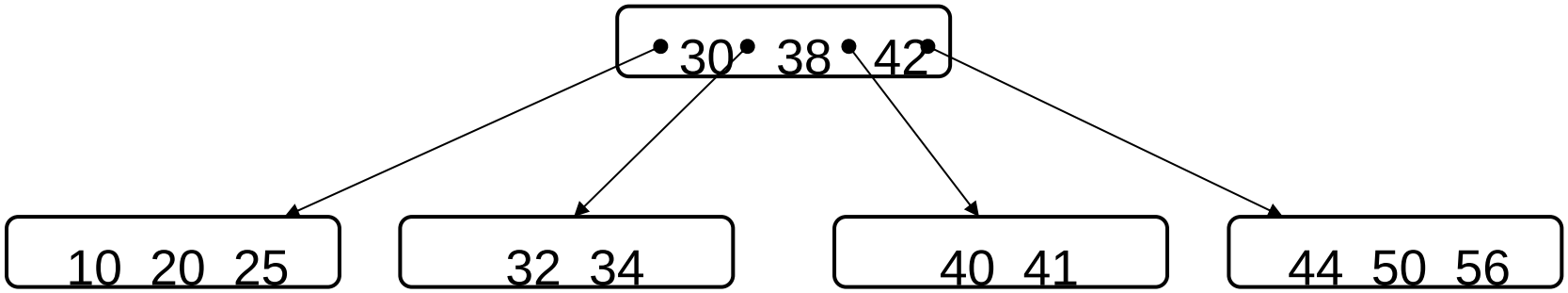
B-Baum (2)

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

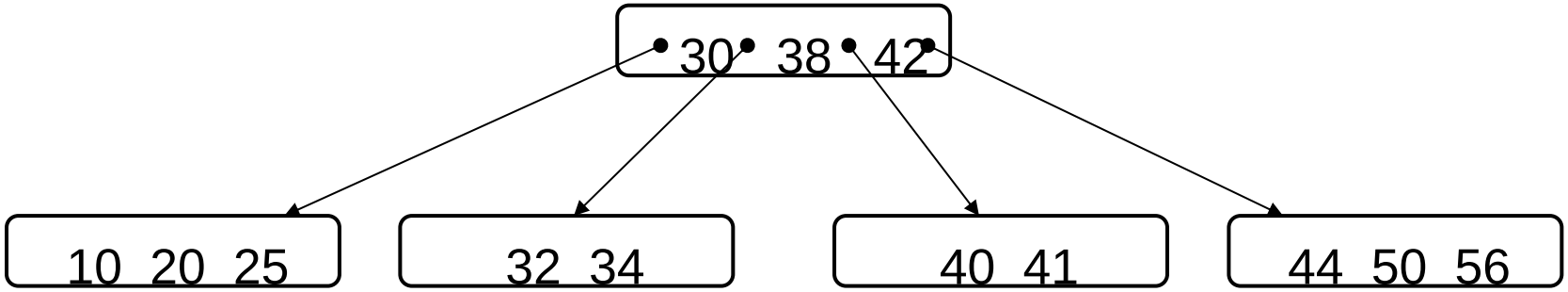


B-Baum (3)

Alle Blätter liegen auf demselben Level.

- Garantierte Zugriffszeiten.
- Bei realistischen Parametern (z.B. Ordnung 1000) sind nur sehr wenige (<5) Zugriffe auf das externe Medium nötig.

B-Bäume besitzen ausgeglichene Höhe, lassen aber unausgeglichenen Verzweigungsgrad und Knotenfüllgrad zu.
Der längste Weg in einem B-Baum der Ordnung d ist in O(logd+1 n) (ohne Rechnung).

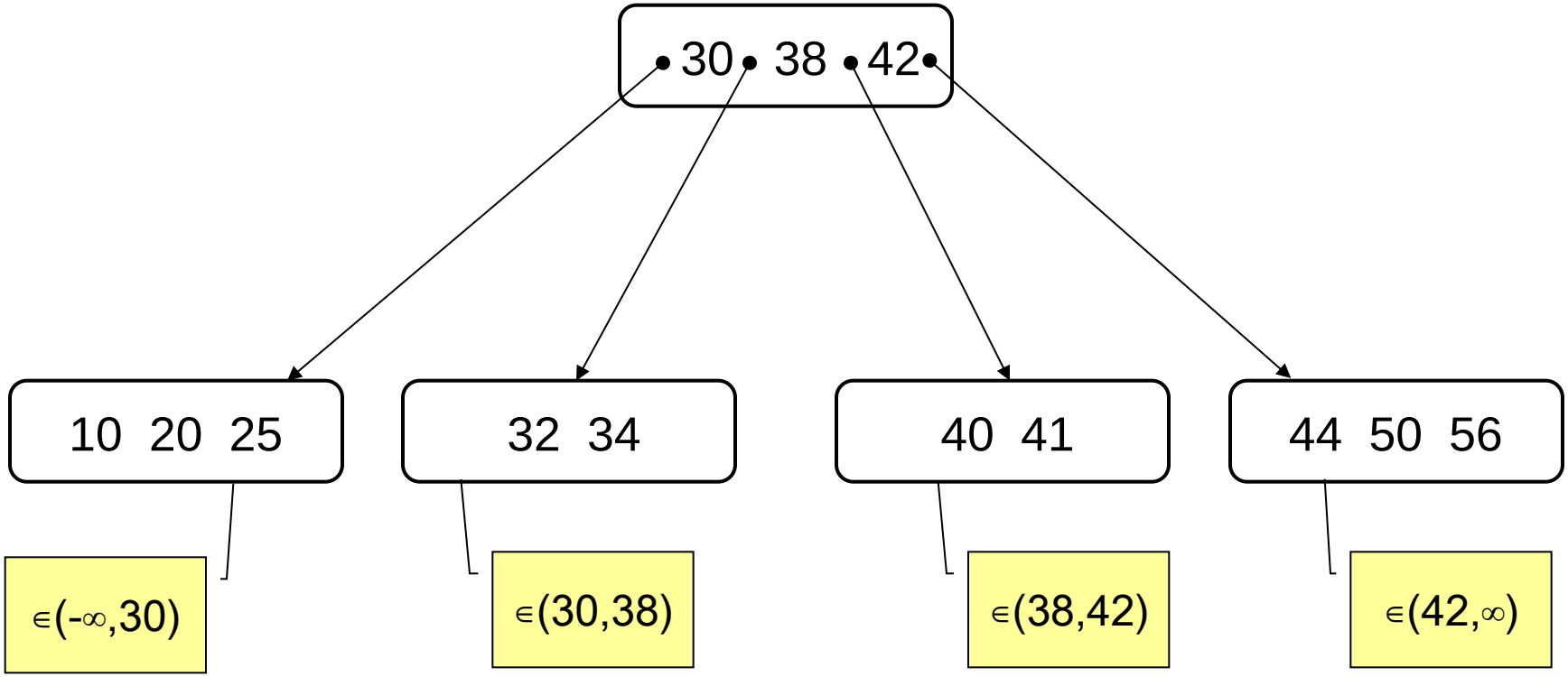


Andere Variante

- Bisherige Definition: Jeder Knoten in einem B-Baum der Ordnung d enthält ***d bis 2d*** Elemente.

- Es gibt eine weitere Variante des B-Baums. Jeder Knoten in einem B-Baum der Ordnung d enthält hier ***d-1 bis 2d-1*** Elemente.
- Wir behandeln nur die erste Variante. Bei der zweiten Variante ist das Einfügen geringfügig anders. Das bekannteste Beispiel der 2. Variante sind die (2,3,4)-Bäume (mit 1,2 oder 3 Element-Knoten).

B-Baum der Ordnung 2



Suchen in einem B-Baum

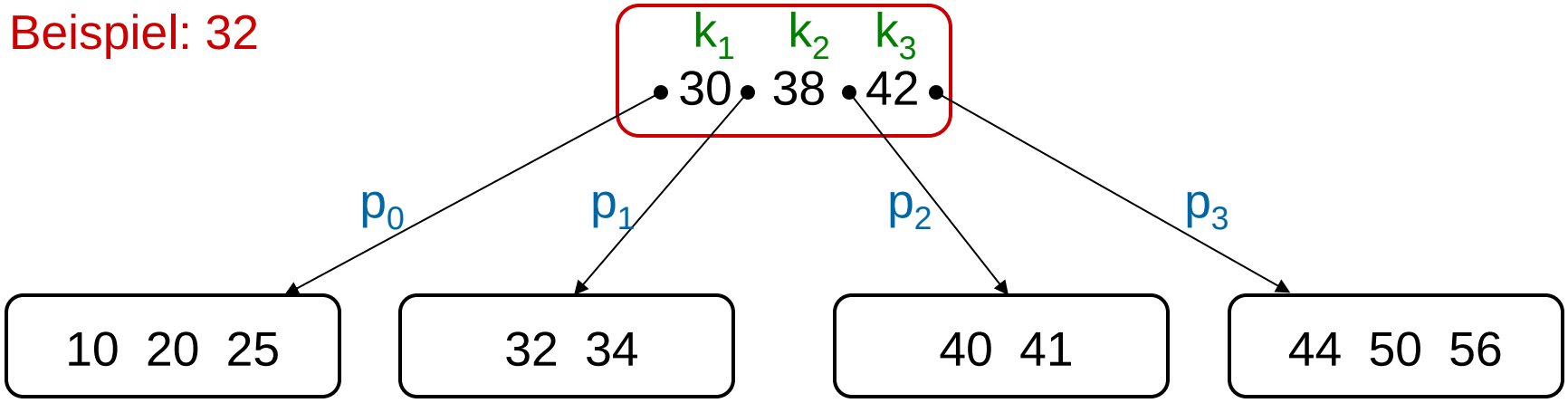
Ausgehend von der Wurzel:

1. Prüfe ob der gerade betrachtete Knoten den gesuchten Schlüssel m enthält

   Suche innerhalb eines Knotens: linear oder binär.

   (Nicht wichtig, weil Laufzeit hauptsächlich von Anzahl der Zugriffe auf Hintergrundspeicher (Festplatte) abhängt.)

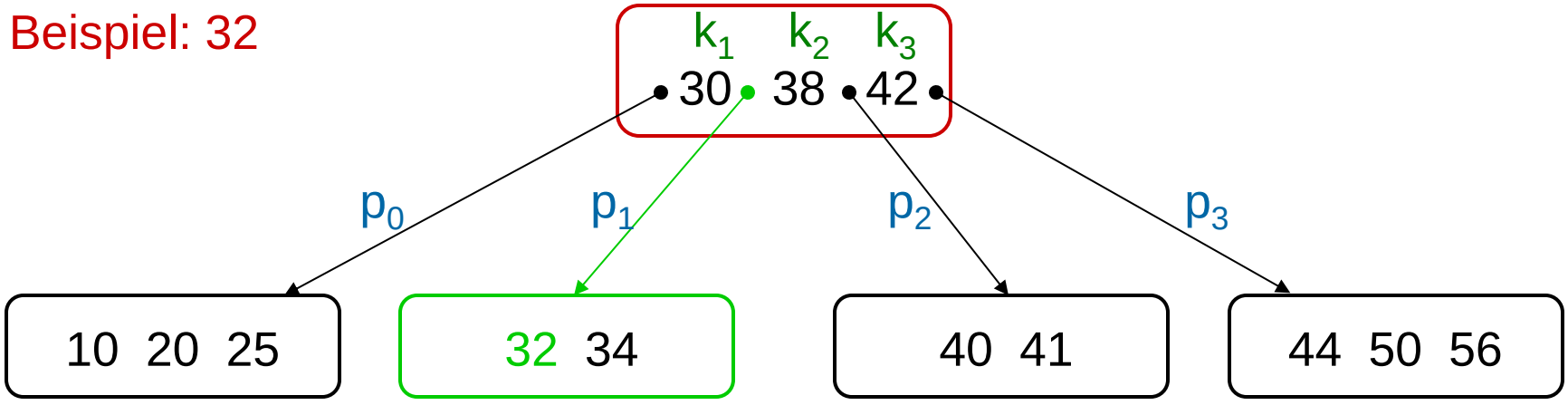
Beispiel: 32



Suchen in einem B-Baum (2)

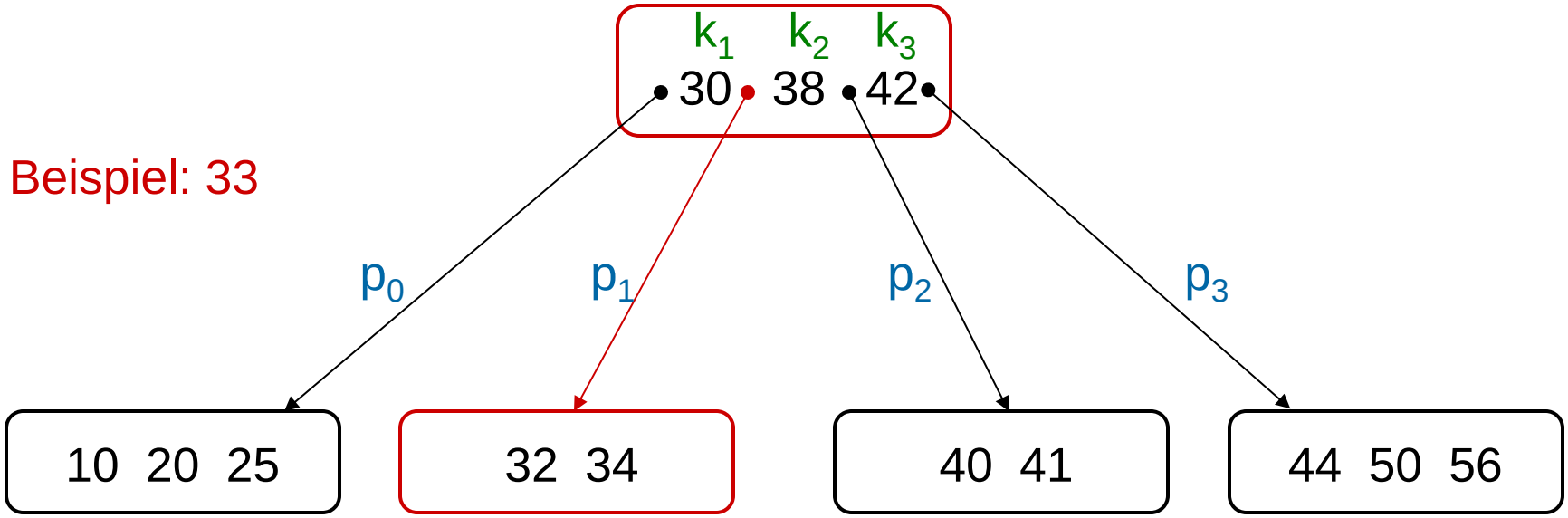
2. Falls nicht, bestimme den kleinsten Schlüssel k_i, der größer als m ist.
   - k_i gefunden: Weiter bei Schritt 1. mit Sohn links von k_i: p_{i-1}.
   - k_i nicht gefunden: Weiter bei Schritt 1. mit letztem Sohn: p_n.

Beispiel: 32



Suchen in einem B-Baum (3)

Erfolglose Suche endet in einem Blatt.



Einfügen in einen B-Baum der Ordnung d

1. Suche nach Schlüssel endet (scheitert) in einem Blattknoten „*node*“.

2. Schlüssel in *node* in Sortierreihenfolge einfügen (und neuen leeren Verweis einfügen).

3. Falls *node* nun überfüllt ist (2d+1 Elemente): *node* aufteilen

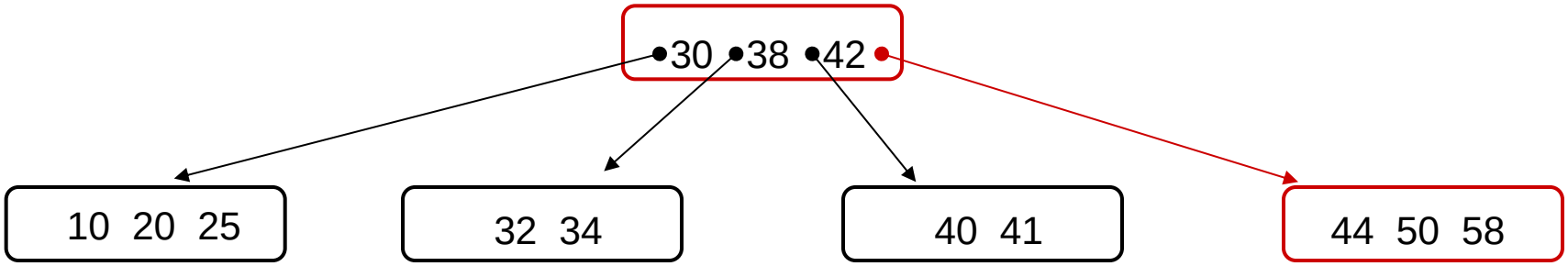
   k sei mittlerer Eintrag von *node*.

   1. Neuen Knoten „*current*“ anlegen und mit den d größeren Schlüsseln (rechts von *k*) belegen. Die d kleineren Schlüssel (links von *k*) bleiben in *node*.
   2. Falls node Wurzelknoten war:
      - neue Wurzel „*parent*“ anlegen.
      - Verweis ganz links in *parent* mit *node* verbinden.
   3. *k* in Vaterknoten „*parent*“ von *node* verschieben.
   4. Verweis rechts von *k* in *parent* mit *current* verbinden.

4. Falls *parent* nun überfüllt ist: Schritt 3. mit *parent* wiederholen.

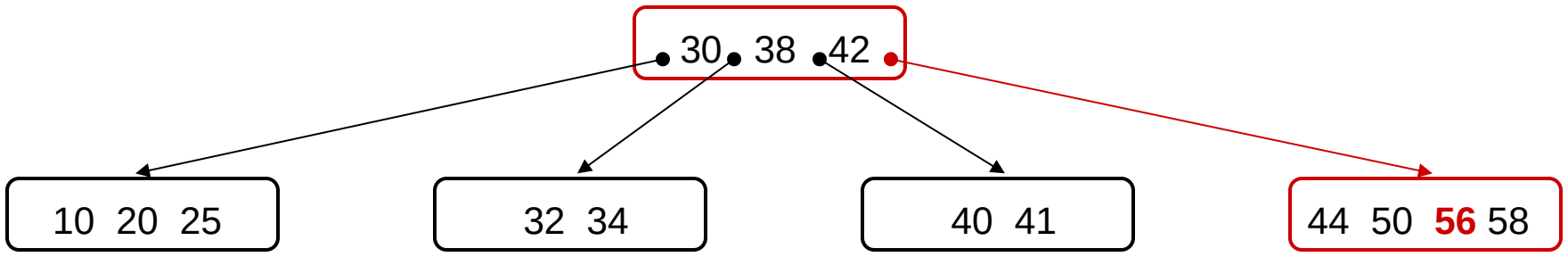
Beispiel: Einfügen von 56.

1. Suche nach Schlüssel endet (scheitert) in einem Blattknoten „*node*“.



Beispiel: Einfügen von 56.

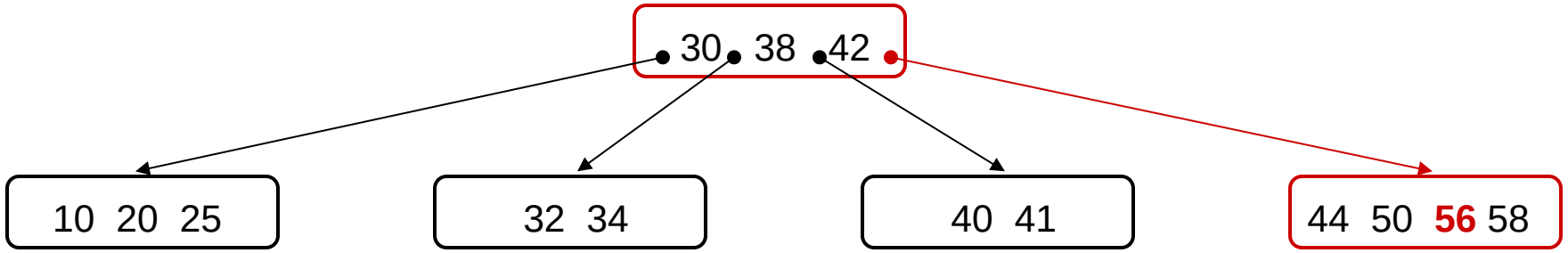
2. Schlüssel in *node* in Sortierreihenfolge einfügen (und neuen leeren Verweis einfügen).



Beispiel: Einfügen von 56.

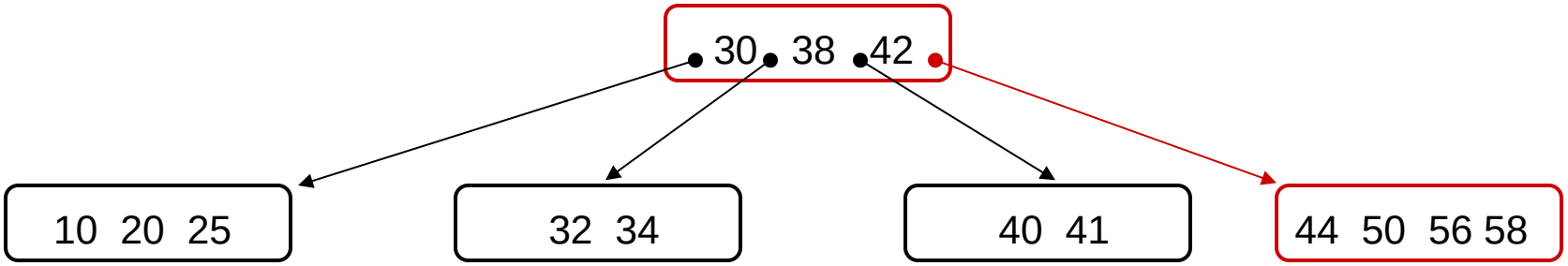
3. Falls *node* nun überfüllt ist (2d+1 Elemente):

→ ist nicht der Fall
→ Ende



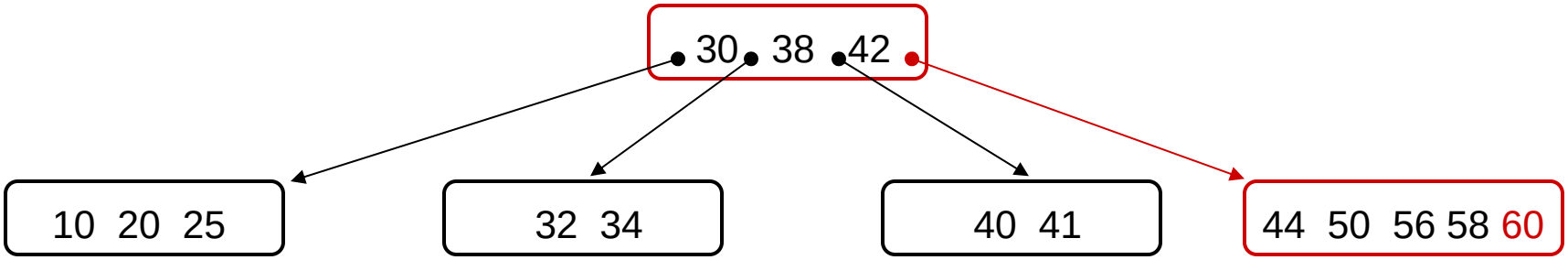
Beispiel: Einfügen von 60.

1. Suche nach Schlüssel endet (scheitert) in einem Blattknoten „*node*“.



Beispiel: Einfügen von 60.

2. Schlüssel in *node* in Sortierreihenfolge einfügen (und neuen leeren Verweis einfügen).

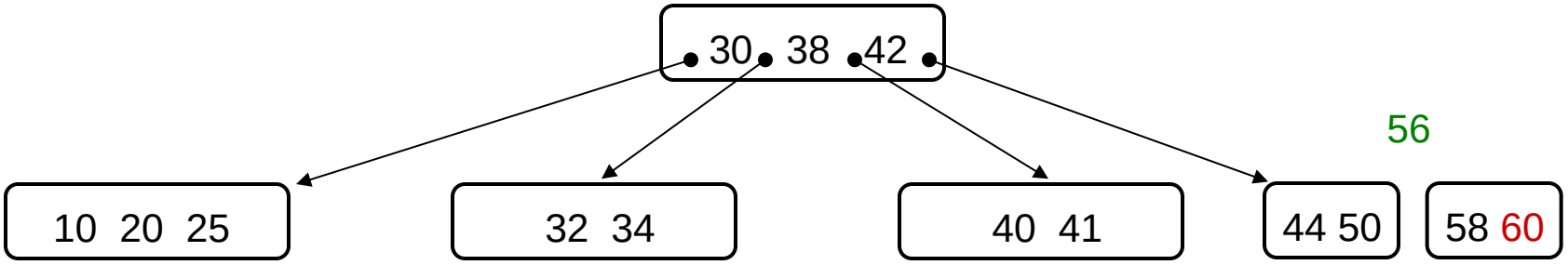


Beispiel: Einfügen von 60.

3. Falls *node* nun überfüllt ist (2d+1 Elemente): *node* aufteilen

   k sei mittlerer Eintrag von *node*.

   1. Neuen Knoten „*current*“ anlegen und mit den d größeren Schlüsseln (rechts von *k*) belegen. Die d kleineren Schlüssel (links von *k*) bleiben in *node*.
   2. Falls node Wurzelknoten war:
      - neue Wurzel „*parent*“ anlegen.
      - Verweis ganz links in *parent* mit *node* verbinden.
   3. *k* in Vaterknoten „*parent*“ von *node* verschieben.
   4. Verweis rechts von *k* in *parent* mit *current* verbinden.

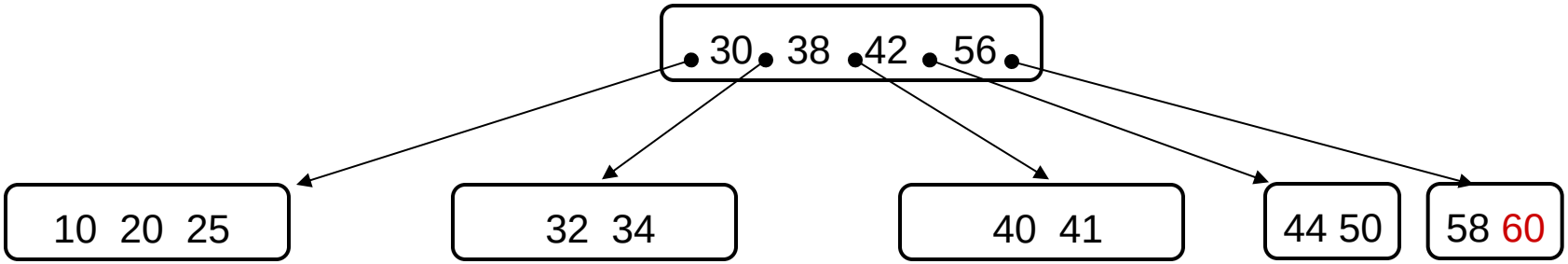


Beispiel: Einfügen von 60.

3. Falls *node* nun überfüllt ist (2d+1 Elemente): *node* aufteilen

   k sei mittlerer Eintrag von *node*.

   1. Neuen Knoten „*current*“ anlegen und mit den d größeren Schlüsseln (rechts von *k*) belegen. Die d kleineren Schlüssel (links von *k*) bleiben in *node*.
   2. Falls node Wurzelknoten war:
      - neue Wurzel „*parent*“ anlegen.
      - Verweis ganz links in *parent* mit *node* verbinden.
   3. *k* in Vaterknoten „*parent*“ von *node* verschieben.
   4. Verweis rechts von *k* in *parent* mit *current* verbinden.

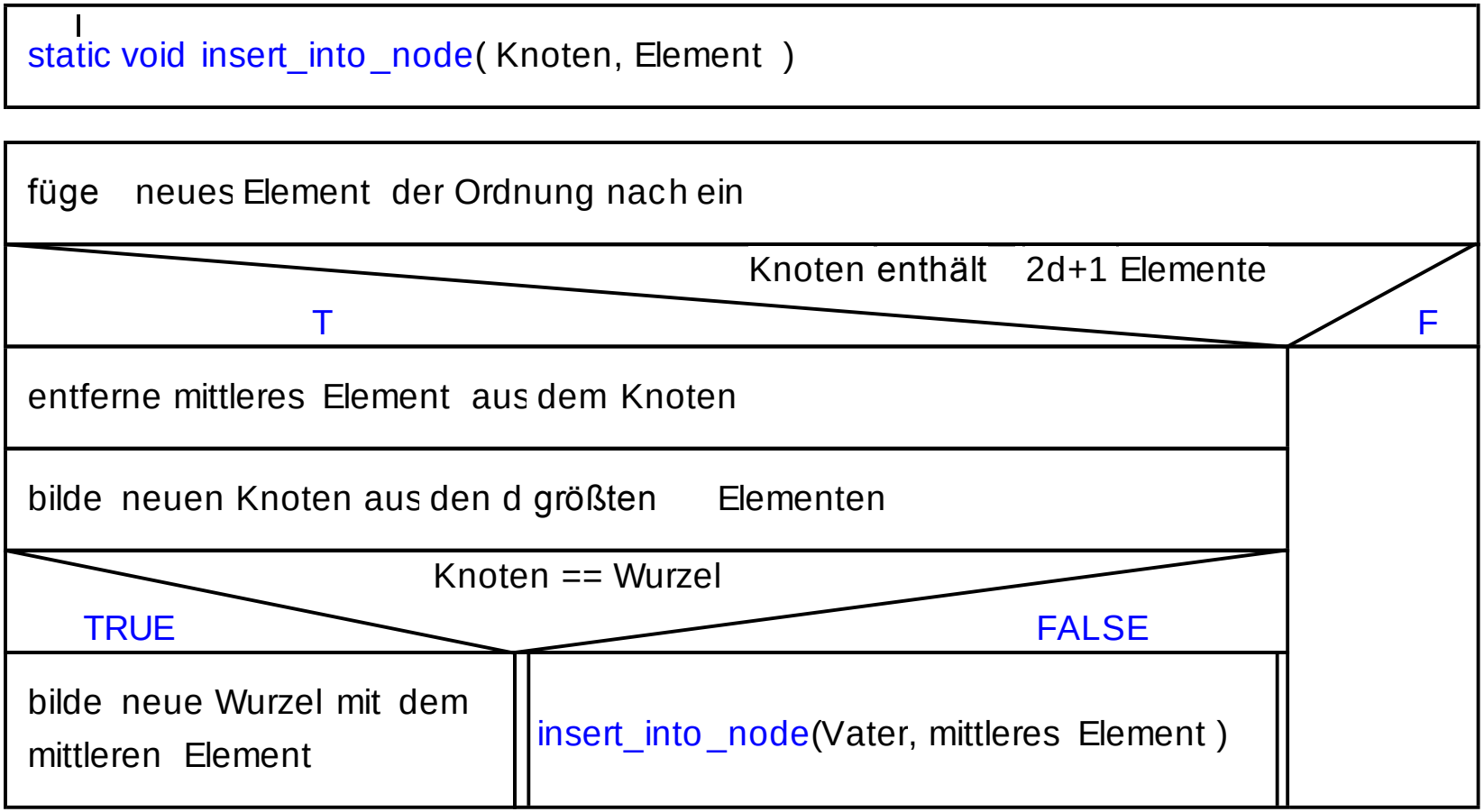


Beispiel Einfügen (8)

3. Falls *parent* nun überfüllt ist: Schritt 3. mit *parent* wiederholen.

   → in diesem Beispiel nicht nötig.

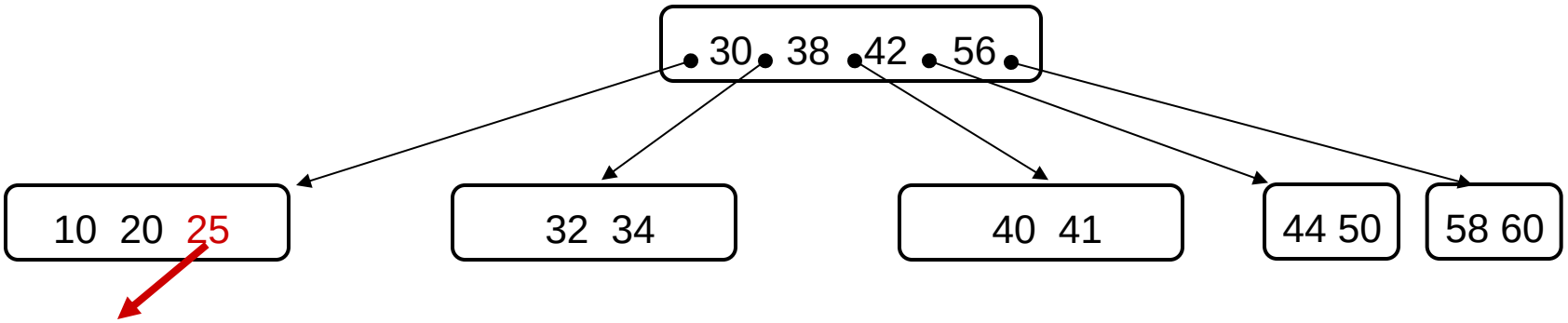
Element in Knoten einfügen



Löschen aus einem Blatt

- Unterscheidung: der zu löschende Schlüssel $x=k_i$ liegt

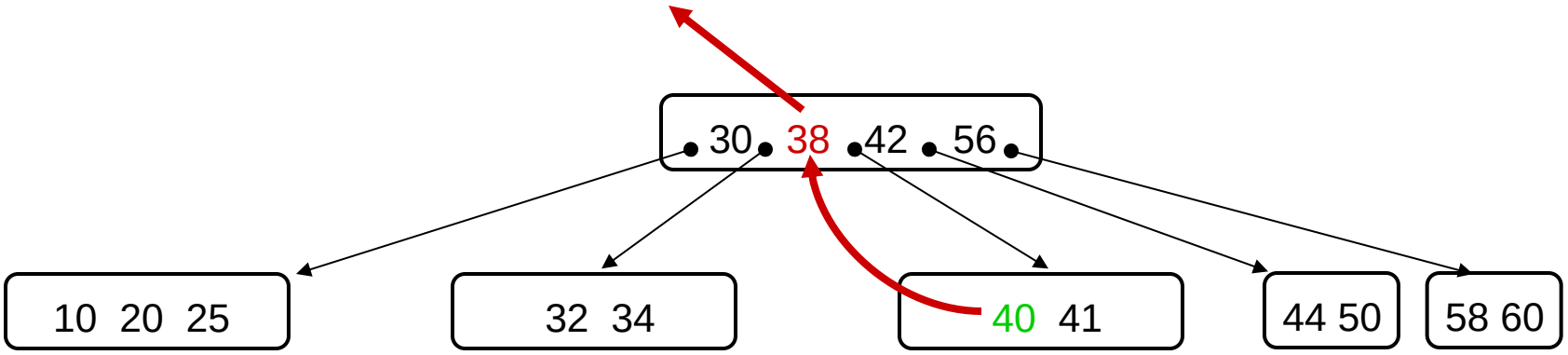
  a) in einem Blatt mit Struktur (null, k_1,..., null, k_i, null,..., k_n, null):
  Entfernen von $x=k_i$ und der darauf folgenden null-Referenz.
  Ein → Underflow tritt auf, falls n=d war.



Löschen aus einem inneren Knoten

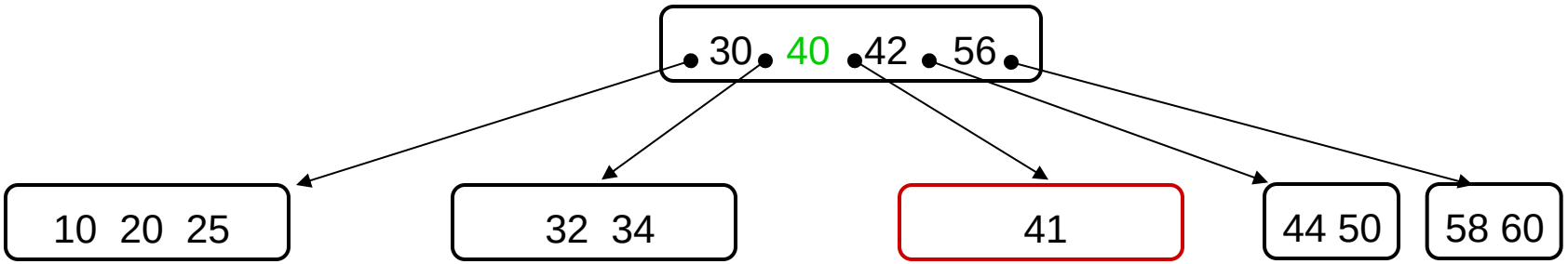
b) in einem inneren Knoten (p_0, k_1, ..., k_i, p_i,..., k_n, p_n):

- alle Referenzen haben einen Wert ungleich *null*.
- analog zu Löschen aus binären Suchbäumen: Finde kleinsten Schlüssel s im durch p_i referenzierten Teilbaum (in einem Blatt).
- Ersetze k_i durch s und lösche s aus dem Blatt (Fall a).



Underflow-Problem

- Underflow-Problem: zu wenig (d-1) Schlüssel im Knoten



Behandlung des Underflows

- Bei der Behandlung des Underflows sind wieder verschiedene Fälle möglich.

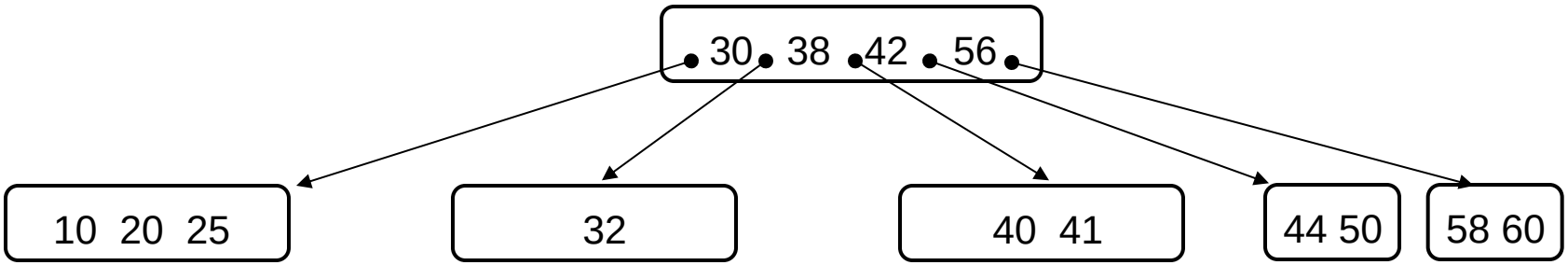
- Wir benötigen zwei Operationen:
  1. Ausgleichen (zwischen zwei Bruder-Knoten)
  2. Verschmelzen (von zwei Bruder-Knoten)

Ausgleich zwischen Bruder-Knoten

- Voraussetzung für Ausgleichsoperation bei Underflow in Knoten q: q hat benachbarten Bruder-Knoten p mit mehr als d Schlüsseln

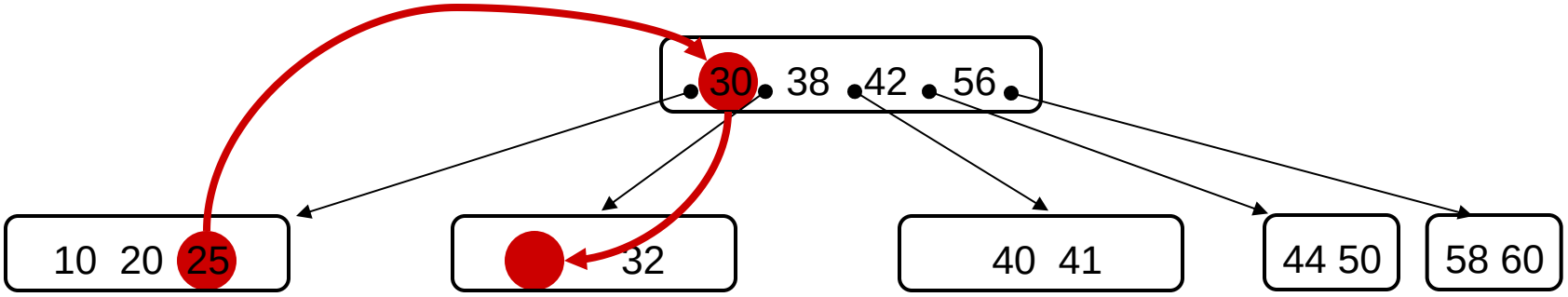
- Annahme:
  - p ist linker Bruder von q.
  - im Vater parent (von p und q) trennt der Schlüssel t die Verweise auf p und q.

Falls vorhanden und ausreichend besetzt. Sonst analog mit rechtem Bruder.



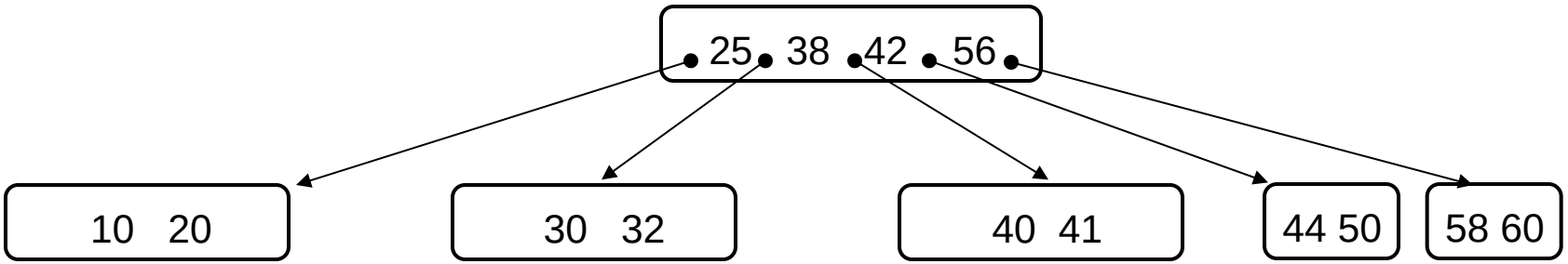
Ausgleich zwischen Bruder-Knoten (2)

- Idee: p schenkt q ein Element („Umweg“ über Vater)

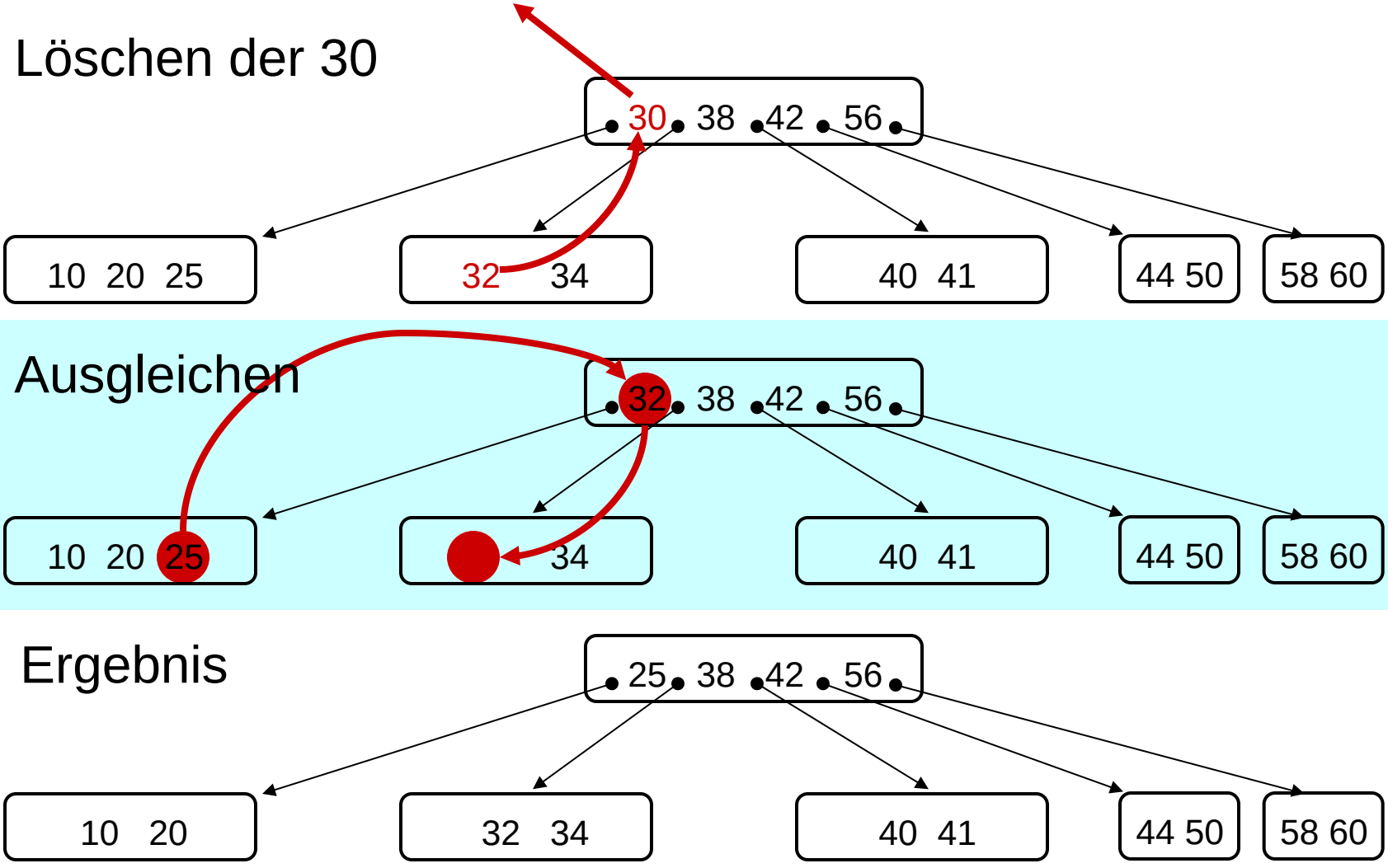


Ausgleich zwischen Bruder-Knoten (3)

- Ergebnis:

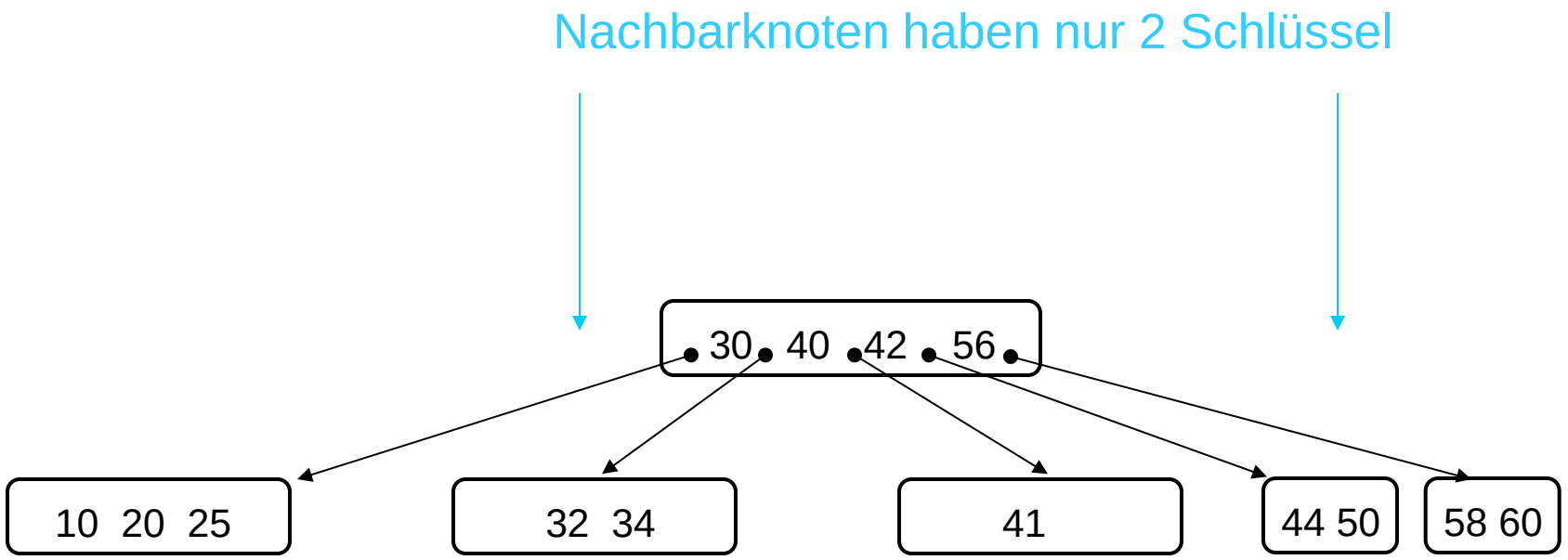


Beispiel: Löschen aus innerem Knoten



Verschmelzen von Bruder-Knoten (1)

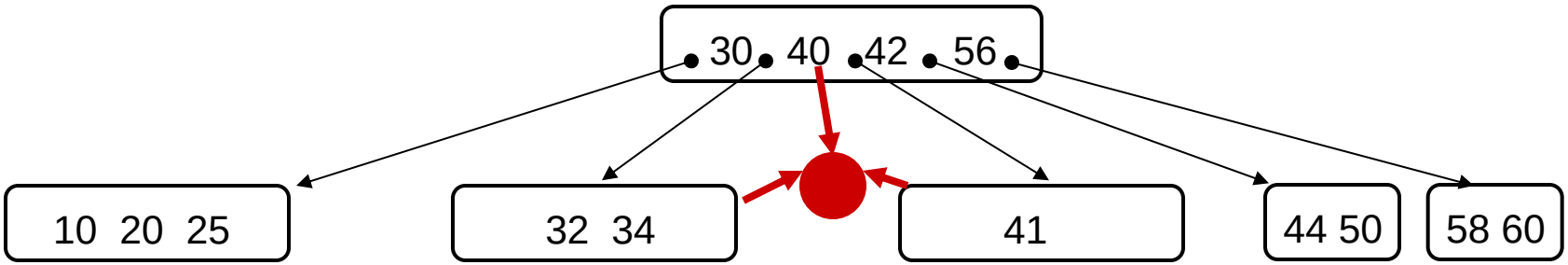
- Verschmelzung mit Bruder-Knoten p bei Underflow in Knoten q: Alle benachbarten Brüder von q haben nur d Schlüssel.



Verschmelzen von Bruder-Knoten (2)

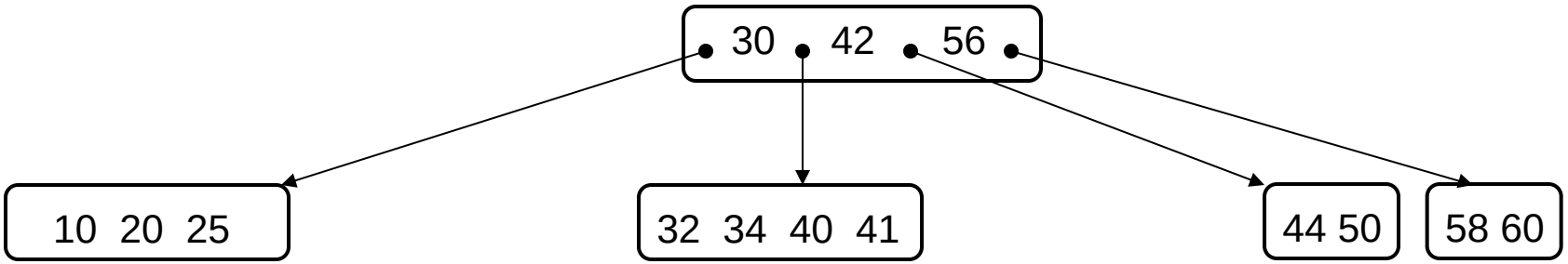
- Annahme:
  - p ist linker Bruder von q.
  - im Vater parent (von p und q) trennt der Schlüssel t die Verweise auf p und q.
- Idee: p und q mit dem trennenden Element aus parent verschmelzen.

Falls vorhanden. Sonst analog mit rechtem Bruder.



Verschmelzen von Bruder-Knoten (3)

- **Ergebnis:**



Verschmelzen von Bruder-Knoten (4)

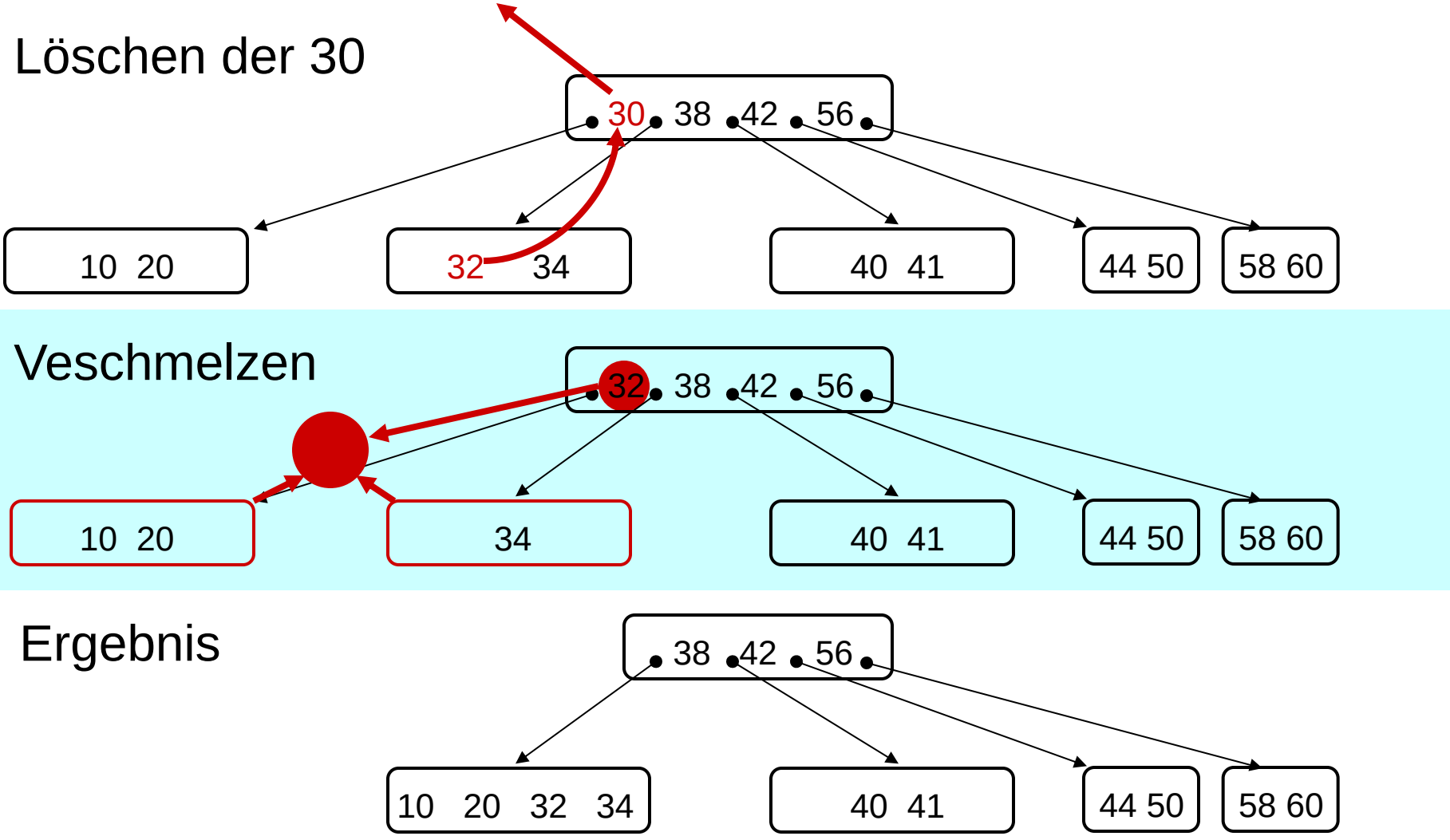
- Jetzt eventuellen Underflow in parent behandeln...
- Underflow-Behandlung rekursiv nach oben fortsetzten.

- Die Wurzel darf auch einen einzigen Schlüssel enthalten.
- Falls zuletzt der letzte Schlüssel aus der Wurzel gelöscht wird (bei Verschmelzung der beiden letzten Söhne der Wurzel zu einem einzigen Knoten):

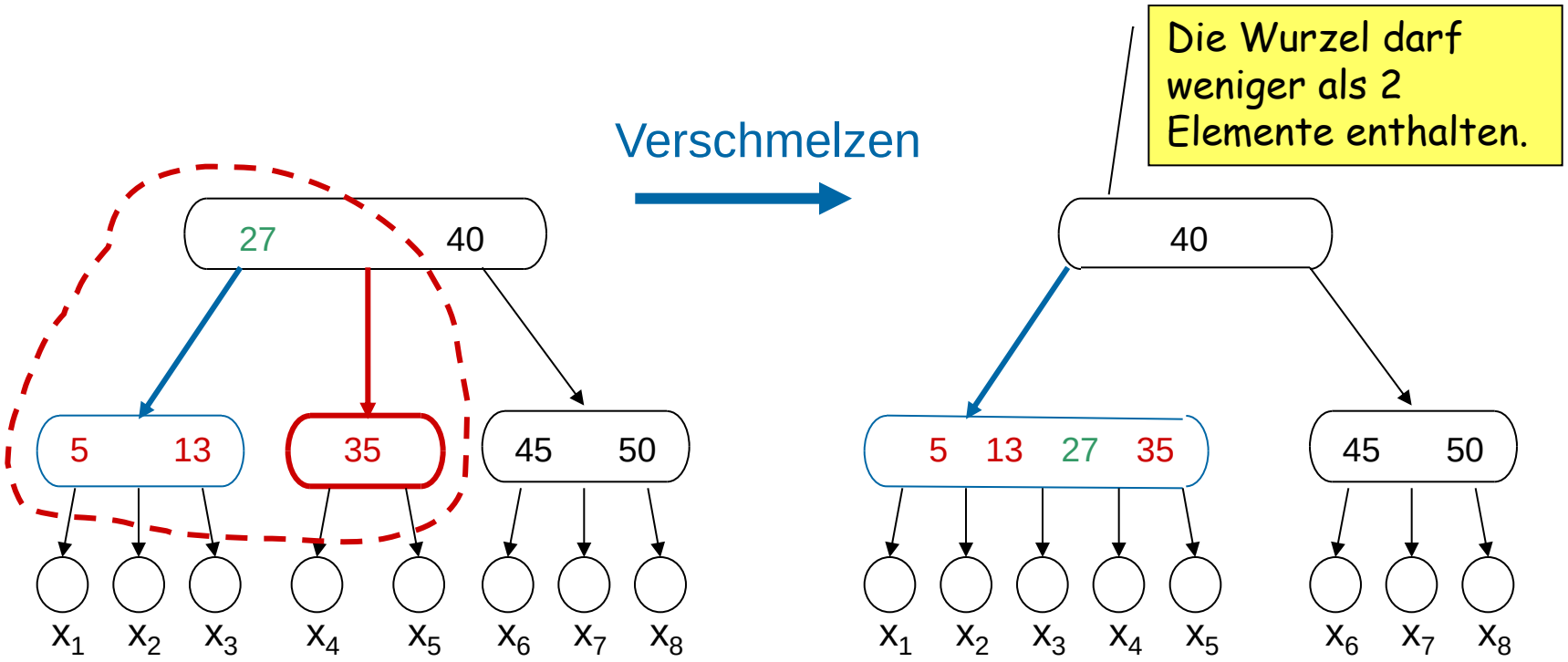
  Einziger Nachfolger der Wurzel wird neue Wurzel.

  $\Rightarrow$ **Höhe des B-Baums um 1 verringert**.

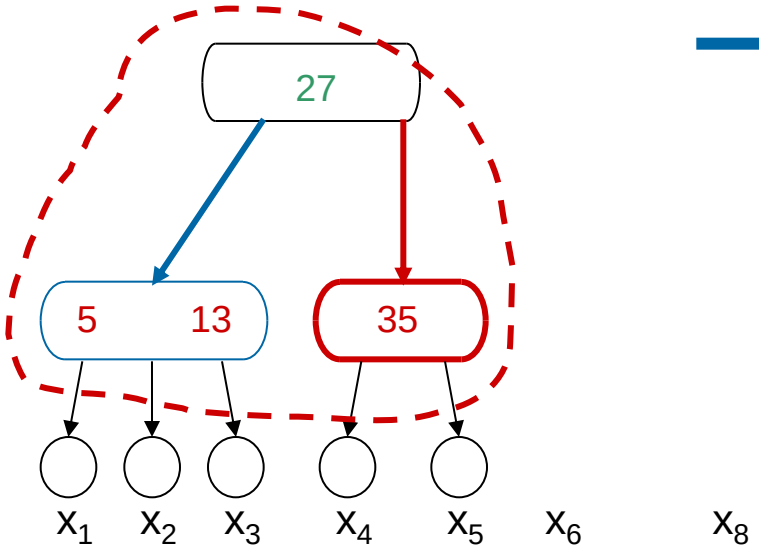
Beispiel: Löschen aus innerem Knoten



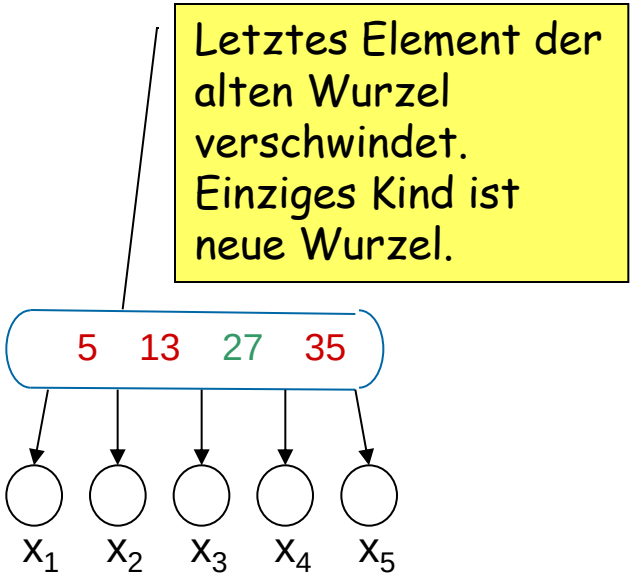
Spezialfall Wurzel (1)



Spezialfall Wurzel (2)



Verschmelzen



B+-Bäume

- Die wichtigsten Unterschiede zwischen B-Bäumen und B+-Bäumen sind:
- B-Bäume
  - **trennen Datensätze nicht von Schlüsseln.**
- B+ -Bäume
  - **speichern in den inneren Knoten nur Schlüssel.**
  - **Die eigentlichen Datensätze befinden sich ausschließlich in den Blättern.**
  - **Dies ist bei der Anwendung für Datenbanken naheliegend und sinnvoll.**

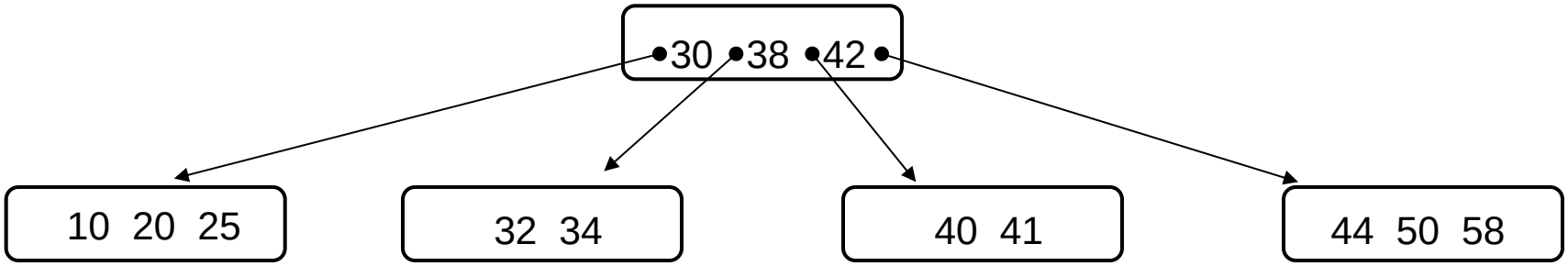
Beispiel-Applet zu B+-Bäumen

- http://www.seanster.com/BplusTree/BplusTree.html

B-Bäume als Mengen

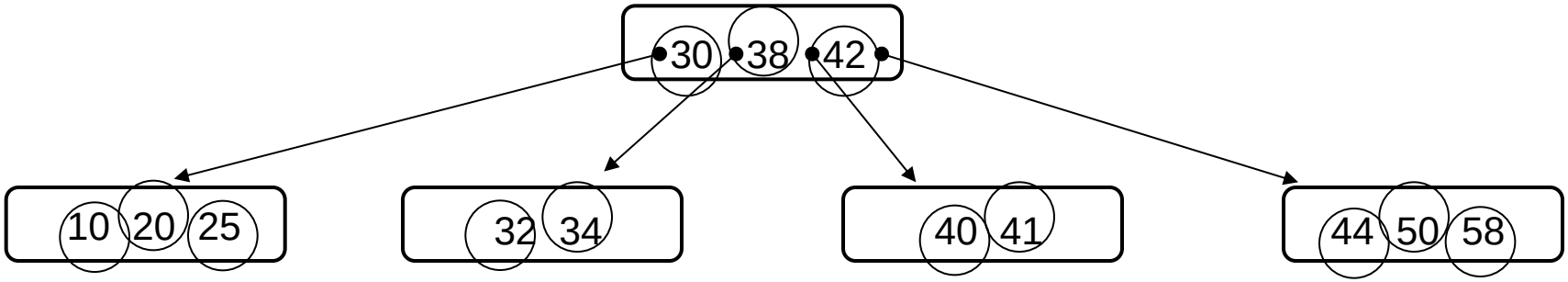
- Einfügen, Löschen, Prüfen und Auslesen sind auch hier O(log n).
- Vorteile haben B-Bäume bei sehr großen Datenmengen, die auf der Festplatte liegen.
  - **Es sei die Ordnung d=1024**
  - **Bei einer Höhe von h=4 können bereits $1024^4-1 \approx 10^{12}$ Schlüssel gespeichert werden.**
  - **Dabei müssen für jede Suchanfrage maximal 5 Baumknoten inspiziert werden.**
- Bei kleinen Datenmengen werden eher Hashtabellen verwendet.

- Rot-Schwarz-Bäume kombinieren (2,3,4)-Bäume (eine Variante der B-Bäume) mit Binärbäumen.
- Gehen wir zunächst vom untenstehenden (2,3,4)-Baum aus.



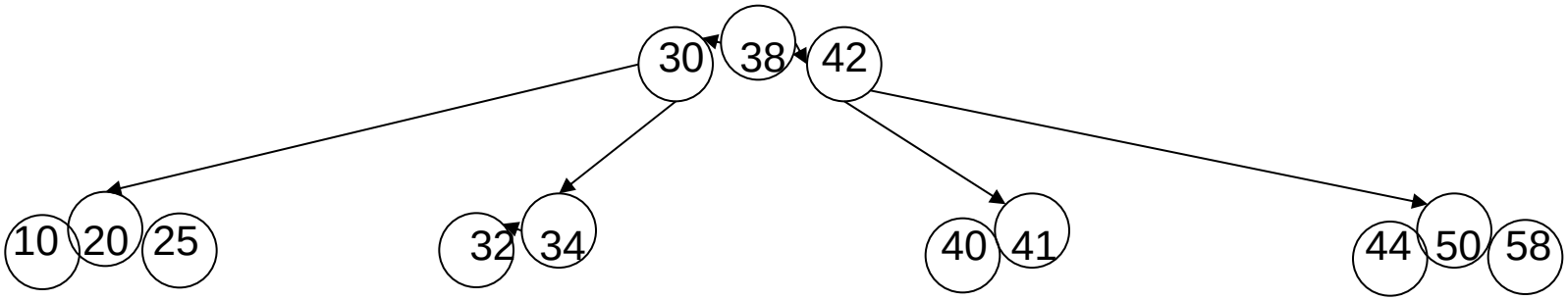
Rot-Schwarz-Bäume (2)

- Die einzelnen B-Baum-Knoten werden als kleine Binärbäume betrachtet, so dass jeder Knoten wieder nur 1 Element besitzt.



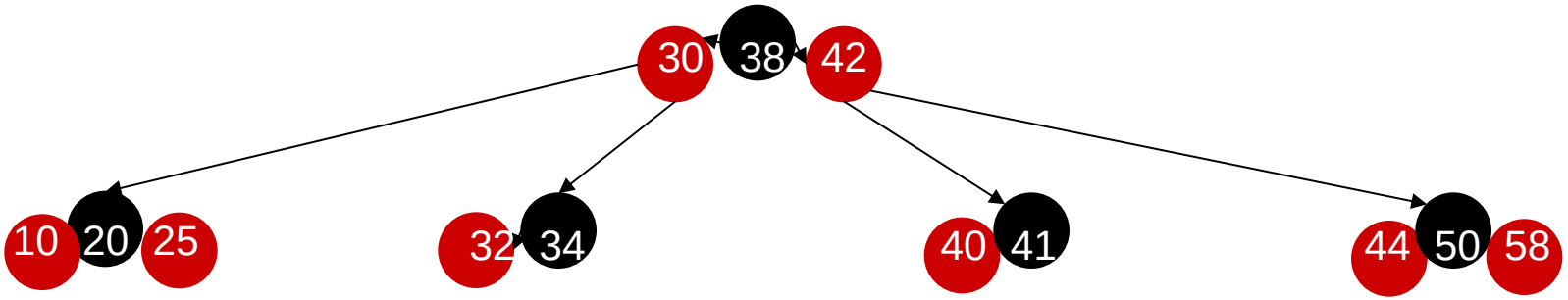
Rot-Schwarz-Bäume (3)

- Die einzelnen B-Baum-Knoten werden als kleine Binärbäume betrachtet, so dass jeder Knoten wieder nur 1 Element besitzt.



Rot-Schwarz-Bäume (4)

- Jeder Knoten erhält einen zusätzliches boolean-Attribut, das sagt, ob der Knoten „rot“ oder „schwarz“ ist.
- Die Wurzeln der kleinen Binärbäume sind schwarz; die anderen Knoten rot.



Rot-Schwarz-Bäume (5)

- Einfügen und Löschen erfolgt wie bei B-Bäumen.
- Zusätzliche Regel: Enthält eine „kleiner“ Binärbaum 3 Elemente, dann wird das mittlere Element automatisch zur Wurzel.

- Die binären Suchbäume in der Java-Bibliothek sind Rot-Schwarz-Bäume
  - **Z.B. java.util.TreeMap, java.util.TreeSet.**

Rot-Schwarz-Bäume als Mengen

- Das Laufzeitverhalten ist ähnlich wie bei AVL-Bäumen.

- Die Höhe von Rot-Schwarz-Bäumen ist etwa 15-30% größer als bei AVL-Bäumen.
  - **AVL-Bäume sind also etwas kompakter.**
  - **Dadurch verringert sich bei Rot-Schwarz-Bäumen die Zeit zum Einfügen/Löschen leicht.**
  - **Die Zeit zum Suchen erhöht sich leicht.**

- Trotz des sehr ähnlichen Verhaltens findet man Rot-Schwarz-Bäume weit häufiger als AVL-Bäume.
  - **AVL-Bäume sind meist Lehrbeispiel.**

Vorteil von Suchbäumen

- Ein Vorteil von Suchbäumen ist, dass durch einen In-Order-Durchlauf (siehe späteres Kapitel) die Daten leicht in einer sortierten Reihenfolge durchlaufen werden können.
  - **Mit einigen Modifikationen könnte man einen binären Suchbaum auch als Liste verwenden:**
  - **Lesen, Schreiben, Löschen und Einfügen mit O(log n).**
  - **Dies ist aber unüblich.**

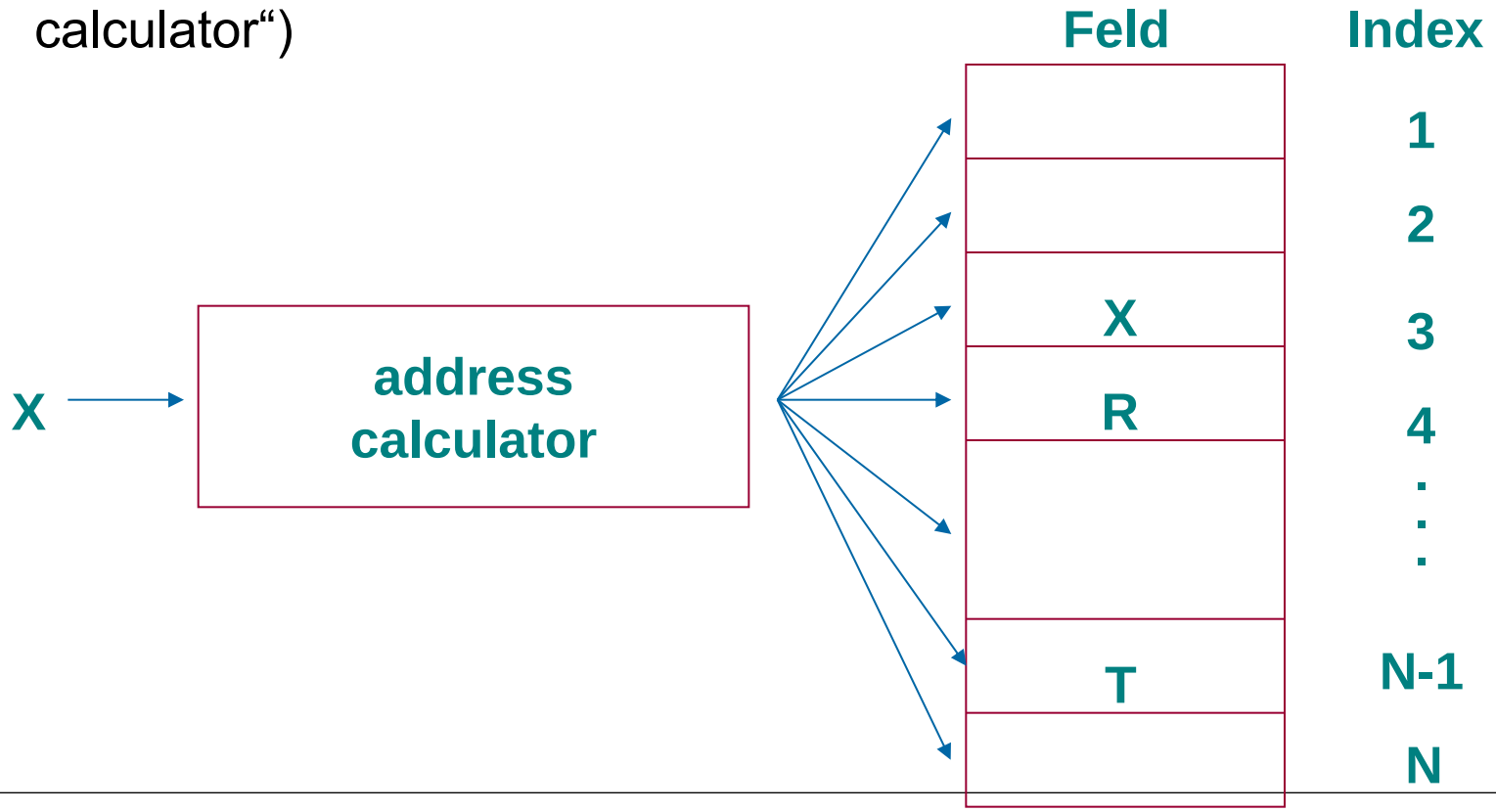
- Dies ist mit Hash-Tabellen nicht möglich.

(Hashtables / Streuwerttabellen)

- Einführung: Hashfunktion und Kollision
- Hashing in Teillisten
- Offene Adressierung (Sondieren)

2.7.1. Grundprinzip einer Hash-Tabelle

- Elemente werden in einem großen Feld gespeichert.
- In welchem Feldelement ein bestimmter Eintrag gespeichert wird, berechnet eine Hashfunktion h(X) aus dem Schlüssel X („address calculator“)



Definitionen zu Hash-Funktion

S sei eine Schlüsselmenge und I eine Adressmenge im weitesten Sinn.

Dann heißt $h : S \rightarrow I$ ***Hash - Funktion***.

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

$|I|$ = Größe N der Hash-Tabelle

Bemerkung:

Die Schlüsselmenge ist im allgemeinen *sehr viel* größer als die Adressmenge.

Deshalb wird eine Hash-Funktion surjektiv ($h(S) = I$), Alle Adressen werden erreicht.

aber **nicht** injektiv sein ($h(s)=h(t) \not\Rightarrow s=t$) !

Hash-Funktionen für verschiedene Datentypen

- Hash-Funktionen hängen ab von
  - Datentypen der Schlüssel
  - Anwendung

- Integer: Divisionsrest-Methode („Divisions-Hash“):

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

  - Bevorzugtes Verfahren, wenn die Schlüsselverteilung nicht bekannt ist.
  - Etwaige Regelmäßigkeiten in der Schlüsselverteilung sollten sich nicht in der Adressverteilung auswirken.
  - Daher sollte *N* eine Primzahl sein.

  - Andere Methoden: Faltung, Mid-Square, ...

Hash-Funktionen für verschiedene Datentypen

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

Beispiel für Hash-Funktion: *h(s)* = s mod 10

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

Einfügen von 69 führt zu **Kollision**

Beispiele für Hashfunktionen (2)

h3(s):

```
int index = ord(s.charAt(0));
index += ord(s.charAt(1));
index += ord(s.charAt(2));
index = index % 17;
```

0:
1:
2:
3: Mai, September
4:
5: Januar
6: Juli
7: März
8: Juni
9: August, Oktober
10: Februar
11:
12:
13:
14: November
15: April, Dezember
16:

Hashing in Java (1)

- Collections Framework unterstützt Hashtabellen, z.B. mit Klasse **`HashMap`**

- **`java.lang.Object`** definiert Methode

  `int hashCode()`

  - berechnet Hash-Wert für Objekt (u.U. aus Speicheradresse)
  - **`java.lang.Integer`** und **`java.lang.String`** liefern eigene Implementierungen
  - kann in selbstgeschriebenen Klassen überschrieben werden

- **`hashCode()`** dient als Basis für Hashfunktionen ...

RWTH AACHEN UNIVERSITY

Definition:

Sei S Schlüsselmenge, h Hash-Funktion.

Ist für $s_1 \neq s_2$ (mit $s_i \in S$)

$$h(s_1) = h(s_2),$$

so spricht man von einer ***Kollision*.**

- Wahrscheinlichkeit von Kollisionen ist abhängig von Hashfunktion

$\Rightarrow$ Hash-Funktionen sollten möglichst gut **streuen**!

- Außerdem: Hash-Funktion muss effizient berechenbar sein.
- Wahl einer guten Hash-Funktion schwierig (→Wikipedia)

Wahrscheinlichkeit von Kollisionen

Annahme:

ideale Hash-Funktion, d.h. gleichmäßige Verteilung über die Hash-Tabelle

„Geburtstagproblem“:

Wie groß ist die Wahrscheinlichkeit, dass mindestens 2 von n Leuten auf einer Party am gleichen Tag Geburtstag haben?

Analogie zum Geburtstagsproblem:

- m = 365 Tage = Größe Hash-Tabelle (bisher N genannt)
- n Personen = Zahl Elemente

Wahrscheinlichkeit von Kollisionen

p(i;m) := Wahrscheinlichkeit, dass i-ter Schlüssel auf freien Platz abgebildet wird (i=1,...,n), wie alle Schlüssel vorher:

$$p(1;m) = \frac{m-0}{m} = (1-\frac{0}{m}) \quad \text{(alle Plätze frei)}$$

$$p(2;m) = \frac{m-1}{m} = (1-\frac{1}{m}) \quad \text{(ein Platz belegt, } m\text{-1 Plätze frei)}$$

$$\vdots$$

$$p(i;m) = \frac{m-i+1}{m} = (1-\frac{i-1}{m}) \quad (i\text{-1 Plätze belegt, } m\text{-}i+1 \text{ Plätze frei})$$

Wahrscheinlichkeit für „keine Kollision“:

$$P(NoKol \mid n,m) = \prod_{i=1}^{n} p(i;m) = \prod_{i=0}^{n-1} (1-\frac{i}{m})$$

Wahrscheinlichkeit für „mindestens eine Kollision“:

$$P(Kol \mid n,m) = 1 - P(NoKol \mid n,m)$$

Tabelle zum Geburtstagsproblem (m=365)

n	*Pr(Kol|n,m)*
10	0,117
20	0,411
...	...
22	0,476
23	0,507
24	0,538
...	...
30	0,706
40	0,891
50	0,970

Schon bei 23 Gästen ist die Wahrscheinlichkeit für „Geburtstagspärchen" > 0,5! („Geburtstagsparadoxon")

Bei 50 Gästen ist „Kollision" fast sicher

Kollisionsbehandlung

Situation: zwei Einträge werden auf die gleiche Feldadresse abgebildet, d.h. $h(x_1) = h(x_2)$

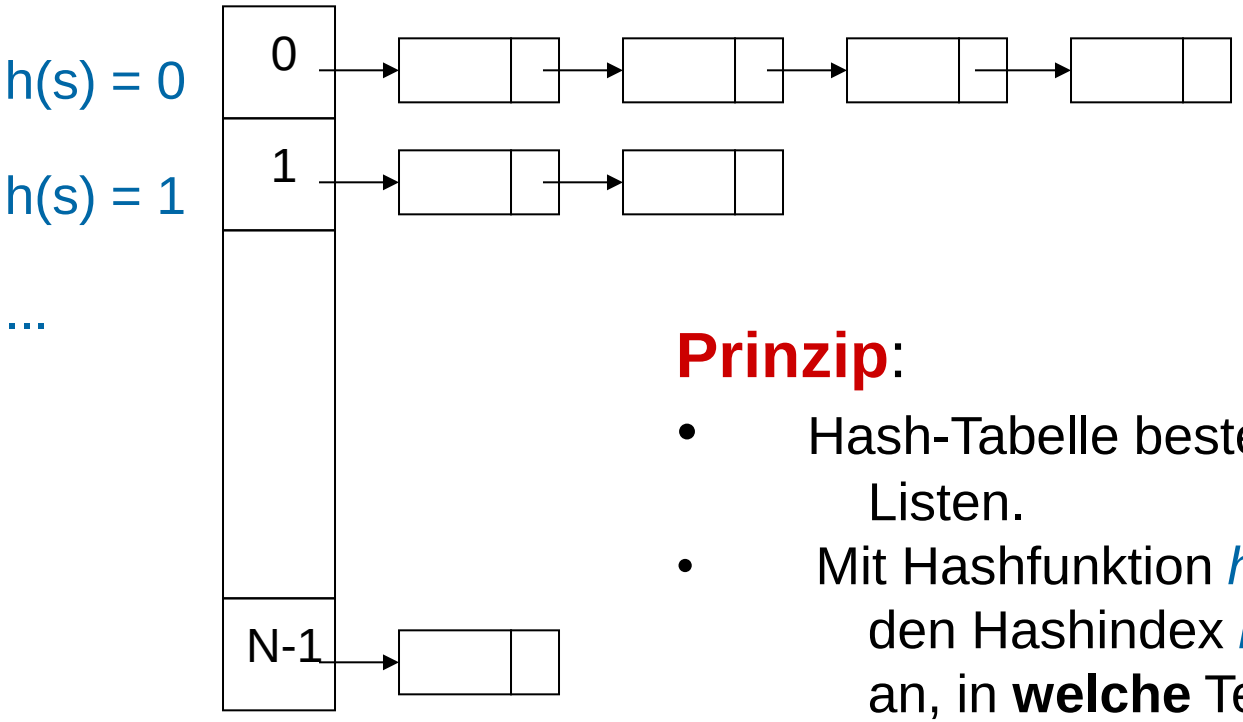
Strategien bei Kollisionsbehandlung (u.a.):

a) „Verkettung der Überläufer“, „Hash in Teillisten“:
Liste für alle Elemente, die die gleiche Position belegen.

b) „Sondieren“, „Hashing mit offener Adressierung“:
Suchen einer alternativen Position innerhalb des Feldes. Wir betrachten:

1. Lineares Sondieren
2. Doppeltes Hashing
3. Quadratisches Sondieren

2.7.2. Hash in Teillisten

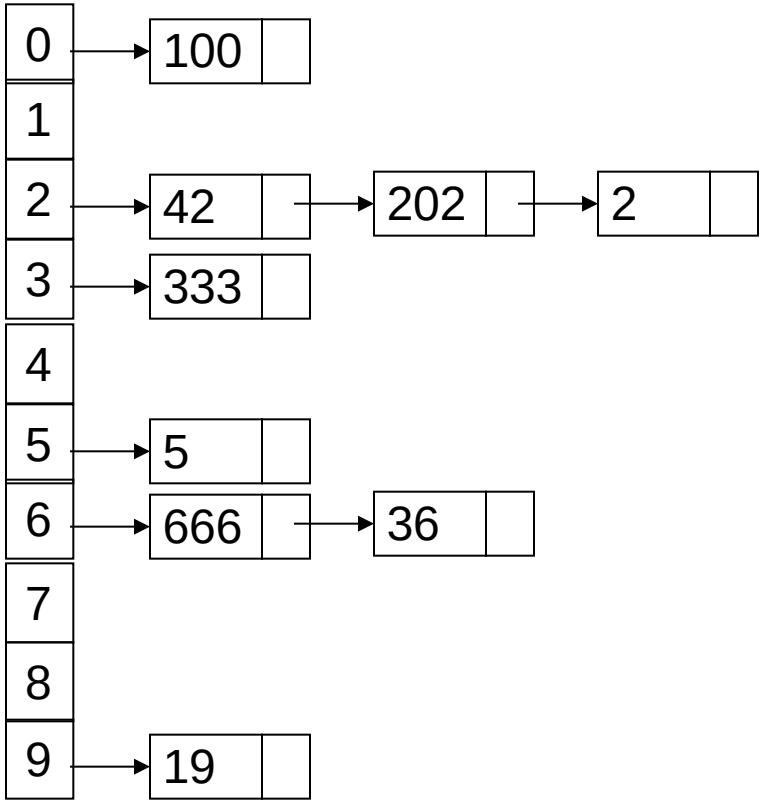


Prinzip:

- Hash-Tabelle besteht aus *N* linearen Listen.
- Mit Hashfunktion *h* aus Schlüssel *s* den Hashindex *h(s)* berechnen (gibt an, in **welche** Teilliste der Datensatz gehört).
- Dann **innerhalb** der Teillisten sequentiell speichern.

Beispiel zu Verkettung mit linearen Listen

$h(i) = i \bmod 10$



Definitionen: Schrittzahl und Füllgrad

Die ***Schrittzahl*** *S(s)*, die nötig ist, um den Datensatz mit Schlüssel *s* zu speichern bzw. wiederzufinden, setzt sich bei Hashing in Teillisten zusammen aus

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

Der ***Füllgrad*** einer Hash-Tabelle ist der Quotient

$$\alpha = n/N,$$

mit

- N := Größe der Hash-Tabelle (# Adressen) und
- n := # gespeicherte Datensätze (normalerweise ist $N \geq n$).

Schrittzahl beim Suchen in Teillisten

N = # Teillisten; n = # gespeicherte Datensätze

Füllgrad der Hashtabelle: $\alpha = n/N$

- Bei idealer Speicherung entfallen α Elemente auf jede Teilliste.
- Schrittzahl im Mittel: Hash-Index berechnen.
  - erfolgreiche Suche: $c_1 + c_2 \cdot \alpha/2$
  - erfolglose Suche: $c_1 + c_2 \cdot \alpha$ — Lineare Suche in Teilliste.

$\Rightarrow$ Suchaufwand: $O(\alpha) = O(n/N)$

$\Rightarrow$ wird der Füllgrad zu groß, sollte die Hashtabelle vergrößert werden (dynamisches Hashing).

Dynamisches Hashing

- Um zu viele Kollisionen zu vermeiden, muss die Hash-Tabelle ab einem gewissen Füllgrad **vergrößert** werden →**dynamisches Hashing**

- Folge einer Vergrößerung: Die gesamte Hashtabelle muss neu aufgebaut werden.
  - **Sowohl beim Hash in Teillisten als auch beim Hash mit offener Adressierung.**
  - **Es sollte stets gelten: $\alpha < 1/2$ (nach Sedgewick).**

2.7.3. Offene Adressierung (Sondieren)

- Speichern:
  - Hashindex mit Hashfunktion aus Schlüssel des Datensatzes berechnen.
    Falls Speicherplatz frei: dort speichern
  - Bei Kollision: Ersatzadresse berechnen und Speicherversuch wiederholen
    Falls berechneter Speicherplatz erneut belegt: Neue Ersatzadresse berechnen; solange bis
    - freier Platz gefunden oder
    - verfügbarer Speicher ganz durchlaufen (↯)
- Suchen: analog zum Ablauf beim Speichern
- Löschen ist aufwändig!

Definition:

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 Hash-Index *h(s)* wird also wie folgt berechnet:

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

($h_0(s) = h(s)$: Hashindex der Hashfunktion selbst)

Beispiel zum Sondieren (1)

- Aufgabenstellung:
  - Eine Firma mit zur Zeit 60 Mitarbeitern vergibt Personalnummern. Da Nummern von ehemaligen Mitarbeitern nicht neu vergeben werden, hat man maximal vierstellige Nummern vorgesehen. (Schlüsselmenge: S = {1...9999}).
  - 100 Speicherplätze für Datensätze (von 0 an nummeriert) zur Speicherung der Personaldaten (Hash-Indizes: I = {0,...,99}).

- Hash-Funktion: $h : S \rightarrow I$ mit $h(s) = s \bmod 100$

- Bei Kollision:
  - berechneten Hashindex solange um 1 (modulo 100) erhöhen, bis freier Platz (Speichern) bzw. gesuchter Eintrag (Suchen) gefunden wird.

Beispiel zum Sondieren (2)

Einfügen:

Schlüssel	Name	h(s)	
1233	Müller	33	
2034	Meier	34	
9539	Schulze	39	
3433	Schmitz	33	1.Kollision, Ersatzadr. 34 2.Kollision, Ersatzadr. 35

Wiederfinden:

Schlüssel	Name	h(s)	
2034	Meier	34	Datensatz gefunden
3433	Schmitz	33	nicht gefunden; Ersatzadr. 34 nicht gefunden; Ersatzadr. 35 Datensatz gefunden
7334	Huber	34	nicht gefunden; Ersatzadr. 35 nicht gefunden; Ersatzadr. 36 leer: nicht gefunden

Primäre und Sekundäre Häufung

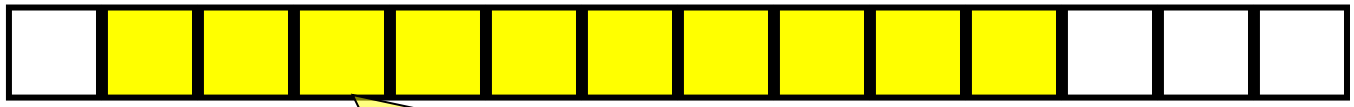
Nachteil des Divisions-Hash mit linearem Sondieren:

- es bilden sich Ketten belegter Plätze. Formal:

$$h_0(s) \neq h_0(t) \wedge h_i(s) = h_k(t) \Rightarrow h_{i+1}(s) = h_{k+1}(t)$$

$\Rightarrow$ erhöht Wahrscheinlichkeit für weitere Kollisionen in diesem Bereich.

– Diesen Effekt nennt man **primäre Häufung** („primary clustering“).



Alle Hashindizes, die auf die Häufung treffen, landen auf dem ersten Element nach der Häufung und tragen zu ihrer Vergrößerung bei.

Primäre und Sekundäre Häufung

- Ähnlich: **Sekundäre Häufung** („secondary clustering“) (hängt von Hashfunktion ab):

$$h_0(s) = h_0(t) \Rightarrow h_i(s) = h_i(t)$$

- Wenn primäre Häufungen durch bessere Verfahren vermieden werden, können immer noch sekundäre Häufungen entstehen, wenn viele (ursprüngliche) Hash-Adressen gleich sind.

Doppeltes Hashing

- Ähnlich wie lineares Sondieren.
- Der 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 ungünstigen Fall kann ein neues Element nicht eingefügt werden, auch wenn noch Felder frei sind.

- Bei der Verwendung von Speicheradressen als Hash-Index können die unteren Bits als Hash-Index und die mittleren Bits als Inkrement benutzt werden.

Quadratisches Sondieren

Definition:

Die Strategie, bei der die Funktion

$$h_i(s) = (h(s) + i^2) \bmod N$$

zur Berechnung der *i-ten Ersatzadresse* gewählt wird, heißt ***quadratisches Sondieren***.

Ersatzadressen bei Quadratischem Sondieren

(Beispiel für N = 11, d.h. 11 Speicheradressen)

i	0	1	2	3	4	5	6	7	8	9	10
i^2	0	1	4	9	16	25	36	49	64	81	100
i^2 mod N	0	1	4	9	5	3	3	5	9	4	1

Bemerkungen:

- Keine primäre Häufung mehr (aber noch sekundäre Häufung)
- Nachteil: $h_i(s) = h_{N-i}(s)$,
  d.h. nicht alle zur Verfügung stehenden Adressen werden erreicht.
  Im Beispiel werden nicht erreicht:
  $(h(s) + k) \bmod N$ mit $k \in \{2, 6, 7, 8\}$

Bedingungen für Abdeckung der Tabelle (1)

Satz: (ohne Beweis)

Ist N eine Primzahl, so sind die Zahlen

$i^2 \bmod N$ für $0 \le i \le N/2$ paarweise verschieden.

Hiermit lässt sich also bei geeigneter Wahl der Tabellengröße immerhin die halbe Tabelle überdecken.

Vergleich der Hash-Verfahren

- Für Mengen bzw. assoziative Felder sind Hash-Tabellen anderen Datenstrukturen überlegen (außer in Spezialfällen).
- Die Unterschiede zwischen den Hash-Verfahren sind vergleichsweise gering.
- Gewöhnlich wird gewählt:
  - **Hashing in Teillisten oder doppeltes Hashing**
  - **generell in Kombination mit dynamischem Hashing.**

Implementierungen von Hashtabellen

Sprache	Klasse	Variante	Max. Füllgrad
Sun-Java	HashMap, Hashtable	Hash mit Teillisten	3/4
Gnu-Java (gcj)	HashMap, Hashtable	Hash mit Teillisten	3/4
C# / Mono	Dictionary	Doppeltes Hashing	3/4
Python	Dictionary	Mehrfaches Hashing	2/3
Ruby	Hash	Hash mit Teillisten	5

Hashtabellen als Mengen

- Beispielhaft für dynamische Hashtabellen mit Teillisten:
- Suchen:
  - **Durchschnittlich werden α Elemente durchsucht. Bei dynamischen Tabellen gilt: $\alpha<c$ (c fest).** **O(1)**
- Einfügen:
  - **Suchen (O(1))**
  - **Einfügen (O(1))**
  - **Eventuell Neuaufbau der Tabelle. Dies ist durchschnittlich O(1) aus den gleichen Gründen wie bei der dynamischen Liste** **O(1)**
- Löschen
  - **Suchen (O(1))**
  - **Löschen (O(1))** **O(1)**

Hashtabellen als Mengen (2)

- Bei ungünstiger Verteilung der Hashwerte (alle Elemente haben denselben Hashwert) ist der Suchaufwand O(n).

- Muss die Hashtabelle vergößert werden, müssen alle Hashwerte neu berechnet werden.
  - **Für einzelne Elemente sehr lange Funktionszeiten möglich (worst case O(n)).**

Komplexität von Mengen

	Unsortierte Liste	Sortierte Liste	Baum	(Dyn.) Hashtabelle
Suchen	O(n)	O(log n)	O(log n)	O(1)
Einfügen	O(n)	O(n)	O(log n)	O(1) wst: O(n)
Löschen	O(n)	O(n)	O(log n)	O(1)

RWTH AACHEN UNIVERSITY

2.8. Tries

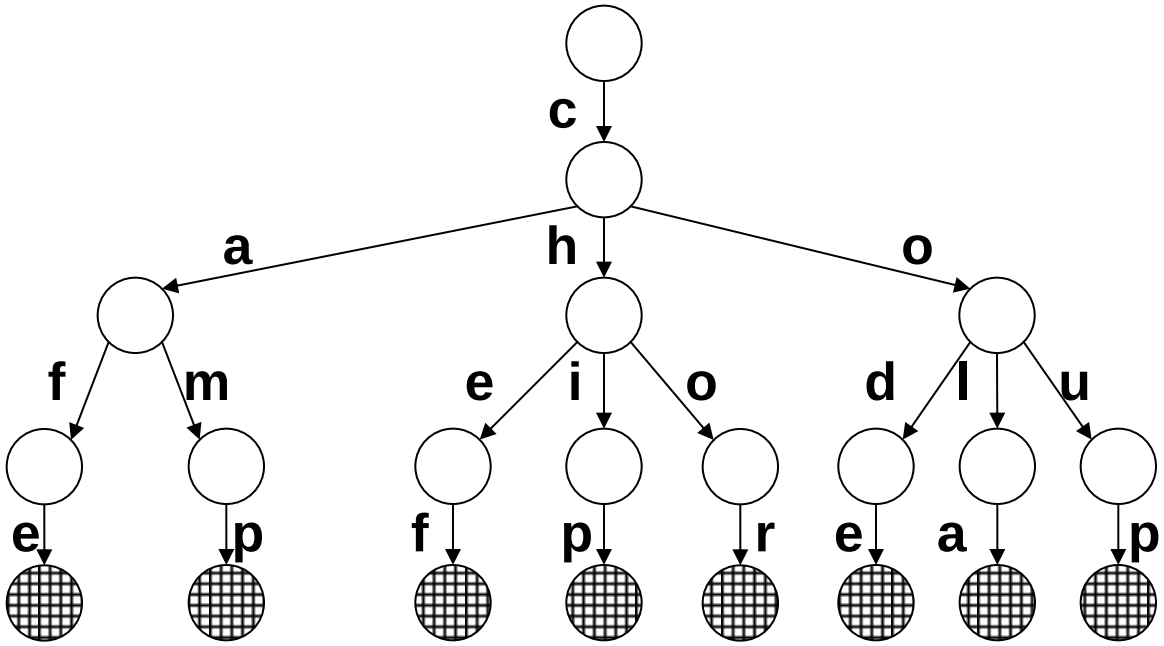
Trie (1)

- Eine weitere Art von Bäumen, die speziell zur Speicherung von Strings geeignet ist.
- Der Name (er spricht sich wie das englische „try“) stammt von dem Wort „retrieval“ (Suche), weil Tries meist zur Textsuche verwendet werden.
- Platzsparende Speicherung von vielen kurzen Strings.
- Einfache Such- Einfüge- und Löschoperationen.

- Es gibt auch Tries, die statt Strings Binärzahlen verwenden (binäre Tries).

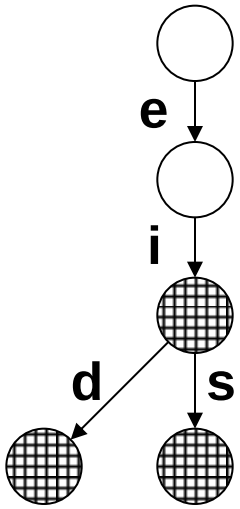
Trie (2)

- Jeder Verweis besitzt einen Buchstaben.
- Jeder Weg von der Wurzel bis zu einem ausgefüllten Knoten entspricht einem Wort



Trie (3)

- In jedem Knoten gibt es einen Binärwert, der sagt, ob ein Knoten der Endknoten eines Wortes ist.



RWTH AACHEN UNIVERSITY

2.9. Prioritätswarteschlangen

ADT Prioritätswarteschlange

- Auch **Vorrangwarteschlange** oder **Priority Queue** genannt.
- Eine Warteschlange, deren Elemente einen Schlüssel (Priorität) besitzen.

- Wichtige Operationen bei Prioritätswarteschlangen:
  - Element in Schlange einfügen
  - Element mit der höchsten Priorität entnehmen.
    - Dies ist gewöhnlich das Element mit dem kleinsten Schlüssel,
    - Manchmal ist es auch das Element mit dem größten Schlüssel.

Anwendungen

- Ereignissimulation
  - **Die Schlüssel sind die Zeitpunkte von Ereignissen, die in chronologischer Reihenfolge zu verarbeiten sind.**
- Verteilung der Rechenzeit auf mehrere Prozesse
- Graphalgorithmen
  - **Dijkstra, Kap. 4.5**
- Sortierverfahren
  - **Alle Elemente in Prioritätswarteschlange einfügen**
  - **Nach der Reihe die größten Elemente entnehmen**
  - **Heap-Sort, Kap. 4.3**

Anwendung außerhalb des EDV-Bereichs

- Wartezimmer eines Arztes
- Reihenfolge des Aufrufs wird durch Prioritätswarteschlange bestimmt.
- Die Priorität wird ermittelt aufgrund:
  - **Ankunftszeit**
  - **Termin / kein Termin**
  - **Privatpatient / Kassenpatient**
  - **Notfall / kein Notfall**

Datenstrukturen für Pr.-Warteschlange

- Einfache Implementierung mit Array (Feld).
- Effiziente Implementierung mit AVL-Baum.
- Andere effiziente Implementierung: Partiell geordneter Baum (**Heap**).

Prioritätswarteschlangen in Java

- Kein Interface
- Nur eine einzige Implementation einer Prioritätswarteschlange: **`java.util.PriorityQueue`**
  - (wird auch in `java.util.concurrent.PriorityBlockingQueue` benutzt).

- Die zugrundeliegende Datenstruktur ist ein Heap.

Heap

- Das Wort Heap (Halde) hat zwei Bedeutungen:

1. **Heap:** Besonderer Speicherbereich, in dem Objekte und Klassen gespeichert werden.
2. **Heap:** Datenstruktur zur effizienten Implementierung einer Prioritätswarteschlange.

- Beide Bedeutungen haben nichts miteinander zu tun. In folgenden Kapitel widmen wir uns ausschließlich der zweiten Bedeutung.
- Dabei betrachten wir ausschließlich den **binären Heap**. Es gibt z.B. noch den Binominial-Heap und den Fibonacci-Heap.

Einführung:

http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html

Kapitel Heap (1+2)

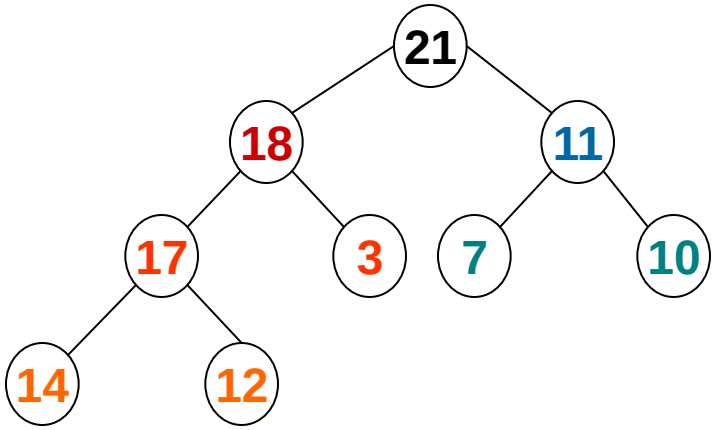
Definition:

Ein ***Heap*** ist ein Binärbaum mit folgenden Eigenschaften:

- Er ist links-vollständig
- Die Kinder eines Knotens sind höchstens so groß wie der Knoten selbst.

⇒ das größte Element befindet sich an der Wurzel des Heaps

Achtung: In der Literatur gibt es auch die umgekehrte Definition



Heap-Eigenschaften

Ein Heap ist also ein Binärbaum mit den beiden Eigenschaften:

Form :

(linksvollständig)

und

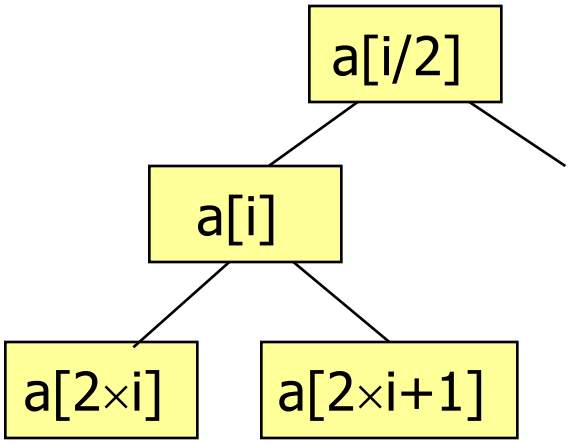
Ordnung:

Entlang jedes Pfades von einem Knoten zur Wurzel sind die Knoteninhalte aufsteigend sortiert.

Wiederholung: Feld-Einbettung

Vorteil der Linksvollständigkeit: Feld-Einbettung leicht möglich.

Vater und Söhne zu a[i]:



Anmerkung: In Java werden Felder beginnend mit 0 indiziert.

⇒ entweder:

- der k-te Knoten wird in dem Feld-Element a[k-1] gespeichert...
- oder Feld-Index 0 bleibt unbenutzt!

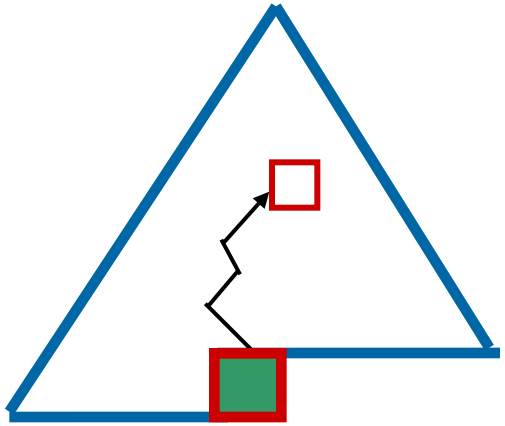
Element einfügen: upHeap(„swim“)

- neues Element in Heap einfügen
  geht (wegen Linksvollständigkeit)
  nur an genau einer Position.

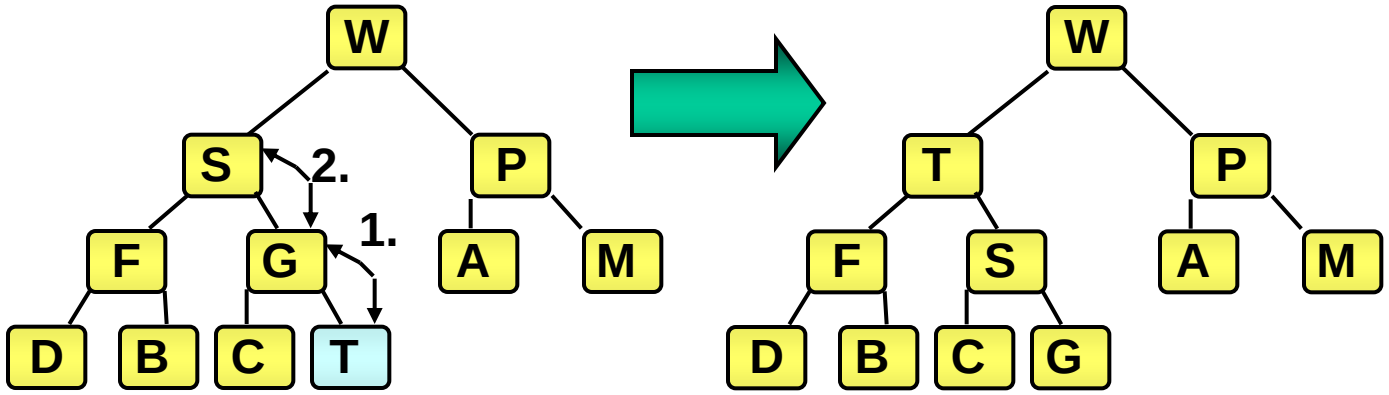
- Ordnungseigenschaft kann dadurch verletzt werden.

- Algorithmus zum Wiederherstellen der Ordnung: upHeap.

- Idee:
  - Knoten mit Vaterknoten vergleichen und ggf. vertauschen.
  - Dies setzt sich nach oben fort (notfalls bis zur Wurzel).



upHeap: Beispiel



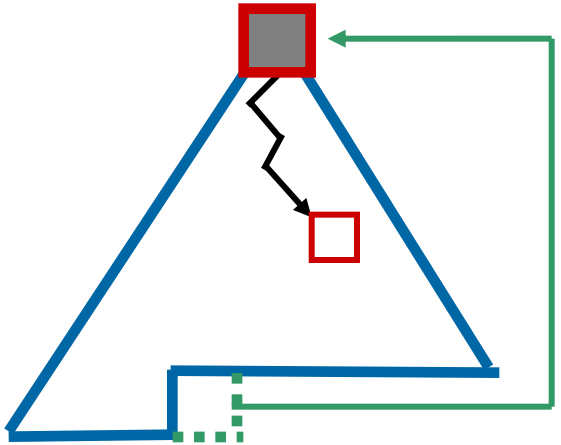
Wurzel entfernen: downHeap („sink“)

- Nach Entfernen der Wurzel eines Heaps wird am weitesten rechts stehender Blattknoten in der untersten Ebene die neue Wurzel.
  $\Rightarrow$ Form wiederhergestellt.

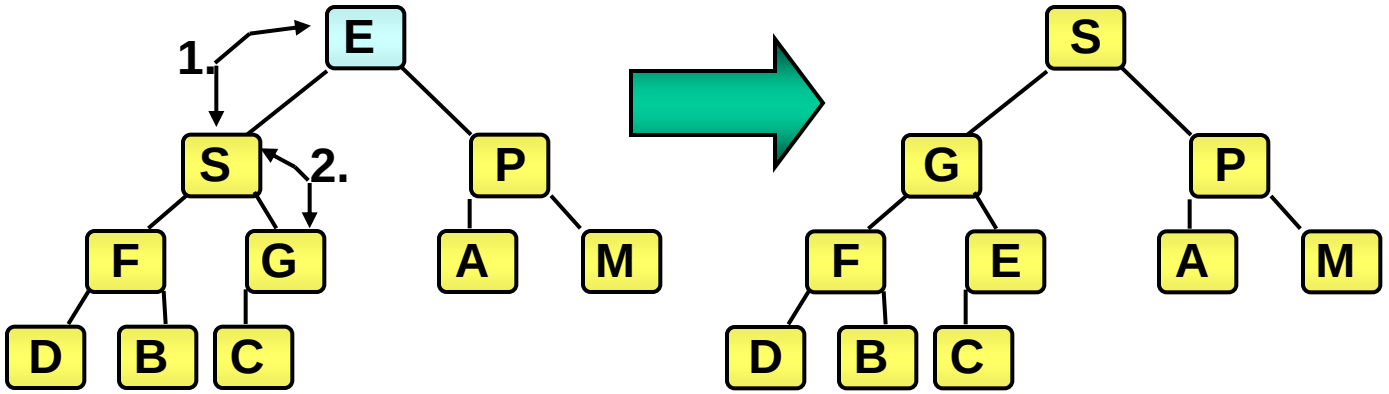
- Ordnungseigenschaft kann dadurch verletzt werden.

- Algorithmus zum Wiederherstellen der Ordnung: downHeap.

- Idee:
  - Neues Wurzel-Element wandert nach unten („versickert“).
  - Dabei wird es jeweils falls nötig mit dem größeren Sohn vertauscht.



downHeap: Beispiel (1)



Aufbau eines Heaps

- Einfache Methode:
  - **Alle Elemente der Reihe nach einfügen und aufsteigen lassen.**

- Falls ein bestehendes (unsortiertes) Array in einen Heap umgewandelt werden soll, ist das **Bottom-up-Verfahren** effizienter:
  - **Die erste Hälfte der Elemente wird nach unten versickert (downHeap).**
  - **Auf dieses Verfahren wird in der Vorlesung nicht weiter eingegangen.**

Zeitkomplexität für Heap-Aufbau

- sink folgt dem Weg von der Wurzel eines links-vollständigen Binärbaums bis maximal zu seinen Blättern, d.h. es werden höchstens

$$\lceil \log_2(n+1) \rceil$$

Knoten besucht. (minimale Höhe +1) $\Rightarrow$ O(log n)

- Aufbau eines Heaps (mit Bottom-up) ruft sink für n/2 Elemente auf, also:

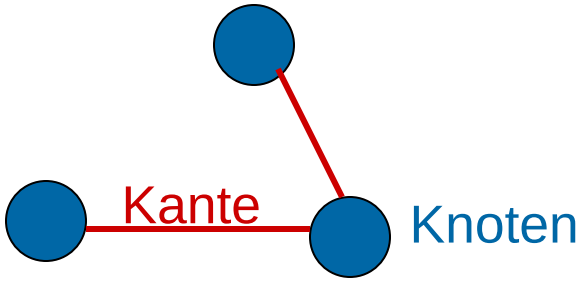
T^{av}heapcreate(n) = T^{w} heapcreate(n) =

$$n/2 \cdot \lceil \log_2(n+1) \rceil \in O(n \log n)$$

2.10. Graphen

Graphen

- sehr allgemeine Datenstruktur
- bestehen aus Knoten und Kanten, die die Knoten verbinden



- Modellierung vieler praktischer Probleme mit Graphen
  - **Z.B. Telefonnetze, Flugpläne, …**
- viele Algorithmen für Graphen bekannt

Formale Definition für gerichtete Graphen

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

  – aus einer endlichen, nicht leeren Menge $V = \{v_1, \ldots, v_n\}$ von Knoten (Engl.: „vertices“) und

  – einer Menge $E \subseteq V \times V$ von geordneten Paaren $e = (u,v)$, den Kanten (Engl.: „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).

- Graphische Darstellung:

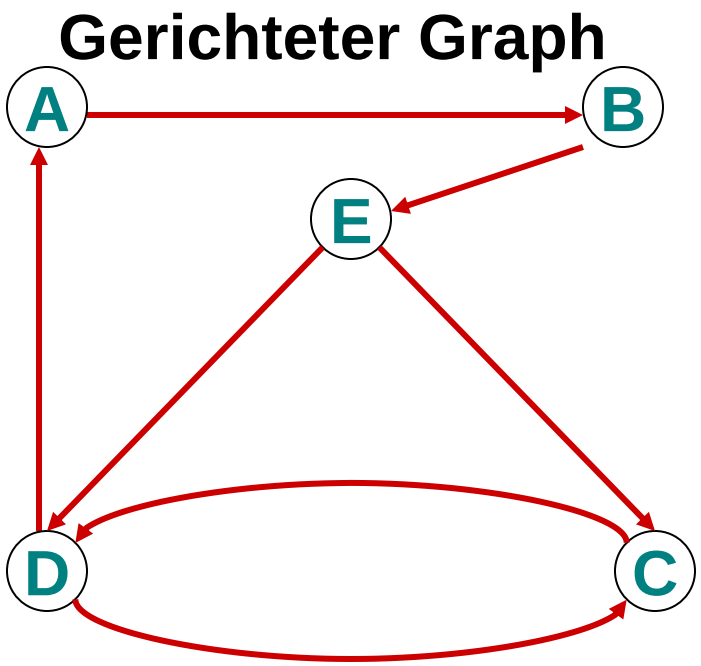
Definition für ungerichtete Graphen

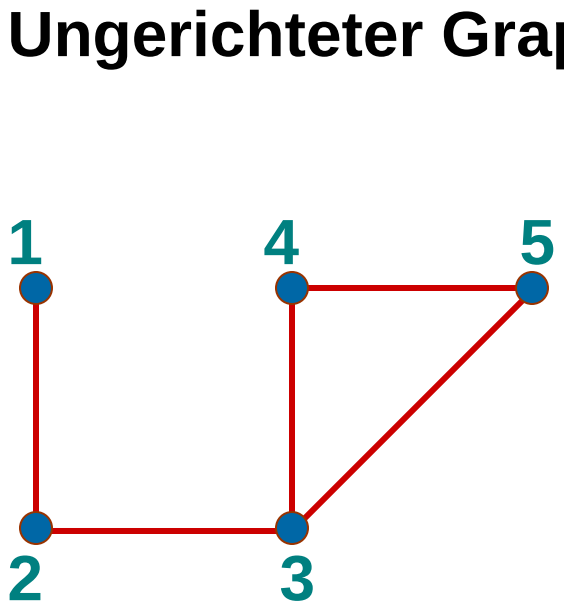
- Ein *ungerichteter Graph* G = (V,E) ist ein gerichteter Graph, bei dem die Relation E symmetrisch ist:

$$(u,v) \in E \Leftrightarrow (v,u) \in E$$

- Graphische Darstellung: 

Gerichteter / Ungerichteter Graph

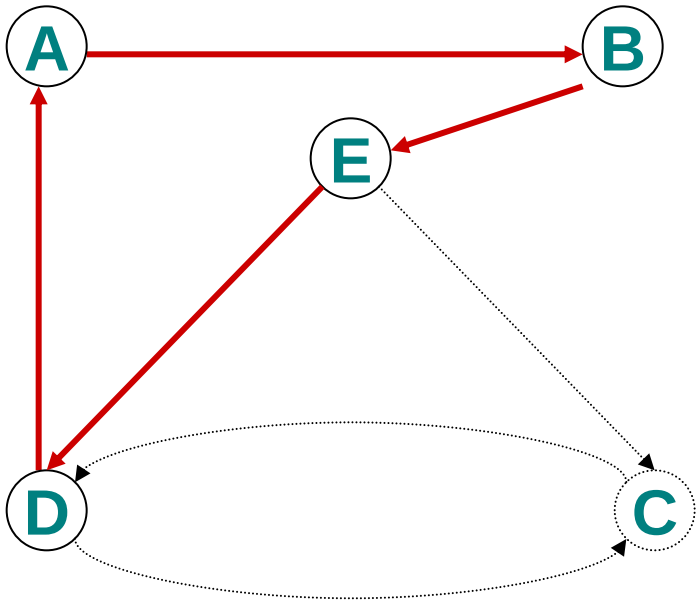




Weitere Begriffe aus der Graphentheorie

- G‘ = (V‘, E‘) heißt **Teilgraph** von G = (V, E), wenn gilt:

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



Weitere Begriffe aus der Graphentheorie

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

- Bei einem gerichteten Graphen mit der Kante u→v ist u **Vorgänger** von v; v ist **Nachfolger** von u.

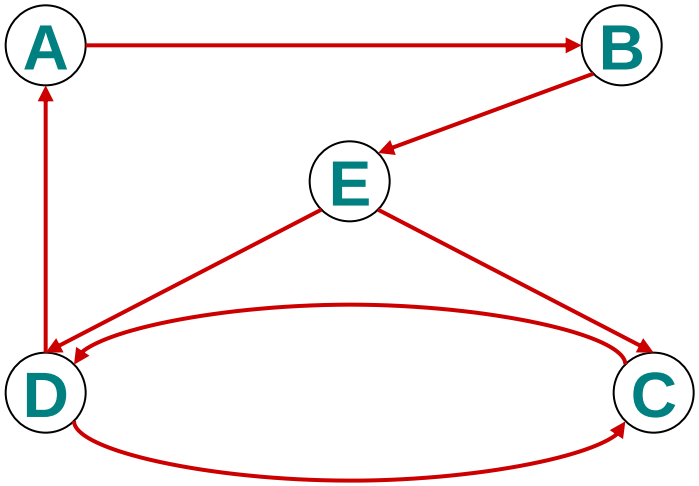
Wege in Graphen

Sei G = (V, E) ein (gerichteter oder ungerichteter) Graph.

- Eine Folge von Knoten W := (v_1, v_2, v_3, ... , v_n) heißt **Weg** (oder **Pfad**) in G, falls gilt:

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

(also eine Folge von „zusammenhängenden“ Kanten)



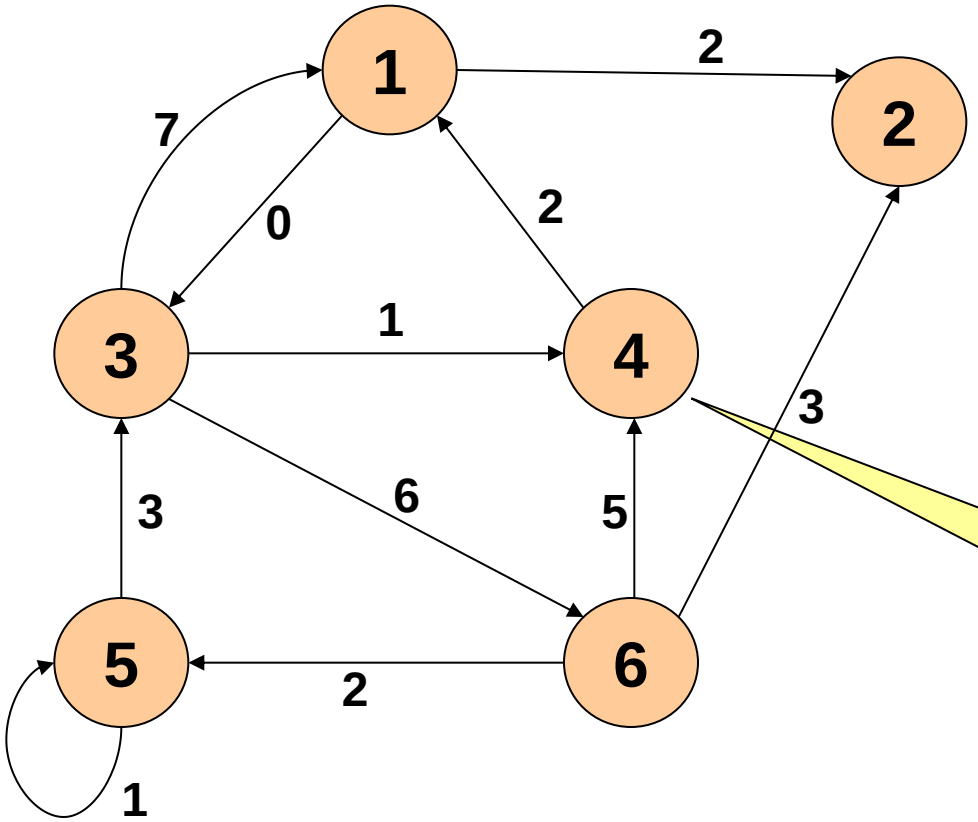
Beispielwege:

(A, B)
(B, E, D, C)
(D, C, D, C, D, C)
(A, B, E, D, A)

Weitere Definitionen

- v_1 =: $\alpha(W)$ heißt **Anfangsknoten** des Weges W
- v_n =: $\omega(W)$ 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 (außer eventuell dem ersten/letzten – siehe Zykel) mehr als einmal vorkommt.
- Ein **Zykel** (Kreis) ist ein (nicht-trivialer) einfacher Weg mit der Bedingung $\alpha(W) = \omega(W)$.

Gewichteter Graph



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

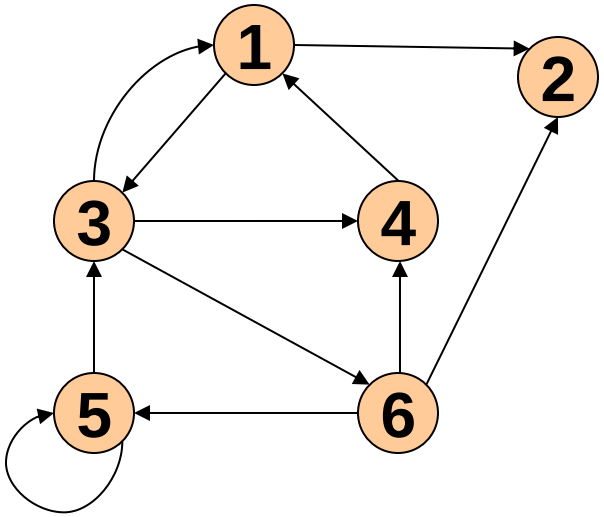
Um von 3 nach 1 zu kommen, ist der Weg über 4 billiger als der direkte Weg.

Je nach Zielsetzung:

- **Kantenorientiert**
  - Prinzip:
    - Index für Kanten
    - Für jede Kante speichern: Vorgängerknoten, Nachfolgerknoten, ggfs.: Markierung, Gewicht
  - Meist statische Darstellung, z.B. Kantenliste

- **Knotenorientiert**
  - gebräuchlicher als kantenorientiert
  - in vielen Ausprägungen, z.B.:
    - Knotenliste
    - Adjazenzmatrix
    - Adjazenzliste

Kantenliste für gerichteten Graphen



Platzbedarf: 2+2·|E|

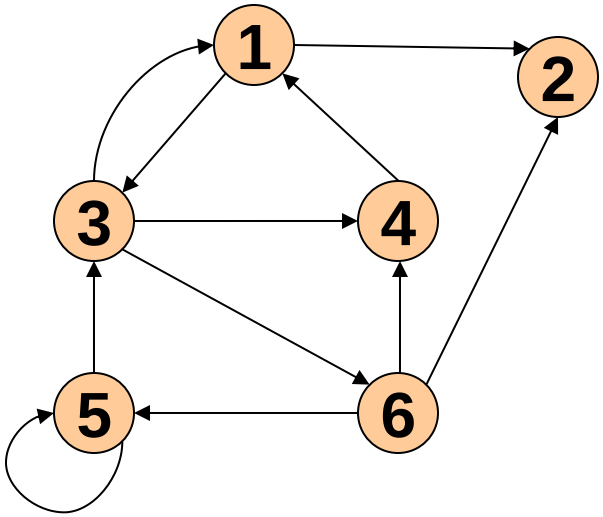
Anzahl der Knoten

Anzahl der Kanten

Für jede Kante: Vorgänger und Nachfolger

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

Knotenliste für gerichteten Graphen



Platzbedarf: 2+|V|+|E|
(normalerweise weniger als Kantenliste)

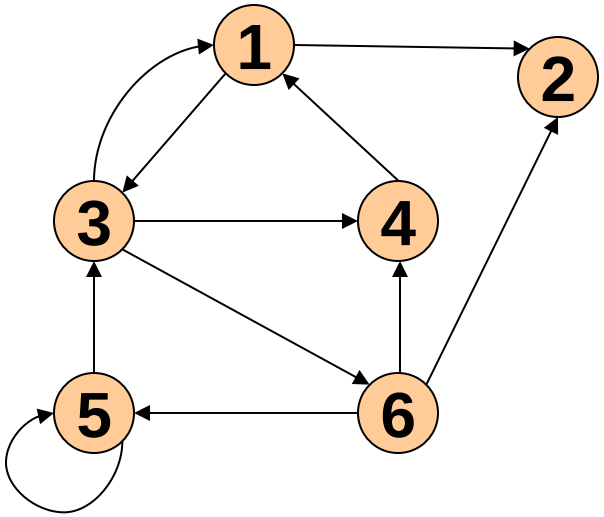
Anzahl der Knoten

Anzahl der Kanten

Für jeden Knoten:
Ausgangsgrad und
und Liste der Nachfolger

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

Adjazenzmatrix für gerichteten Graphen



Platzbedarf: $|V|^2$

Adjazenzmatrix für ungerichtete Graphen ist symmetrisch

Gewichte können anstelle boolescher Werte gespeichert werden

$$A_{ij} = \begin{cases} true, & falls\ (i,j) \in E \\ false, & andernfalls \end{cases}$$

nach

von

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Vor- und Nachteile der Adjazenzmatrizen

Vorteile:

+ Entscheidung, ob $(i,j) \in E$, in Zeit $O(1)$
+ erweiterbar für Kantenmarkierung und Gewichte

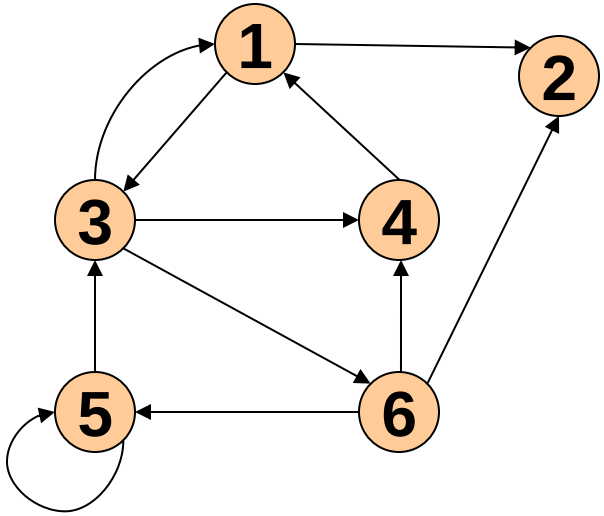
Nachteile:

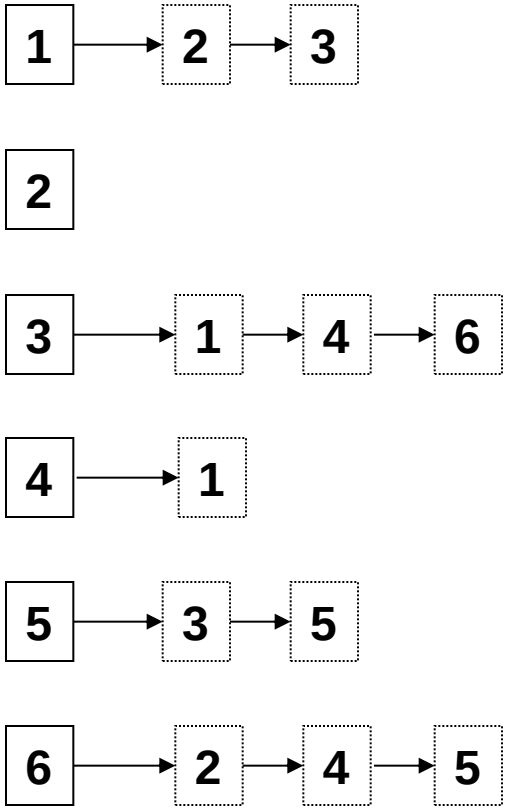
– Platzbedarf stets $O(|V|^2) \Rightarrow$ ineffizient falls $|E| \ll |V|^2$

– Initialisierung benötigt Zeit $O(|V|^2)$

„viel kleiner als“

Adjazenzliste für gerichteten Graphen





Platzbedarf:
- |V| Listen
- |E| Listenelemente

Vor- und Nachteile der Adjazenzlisten

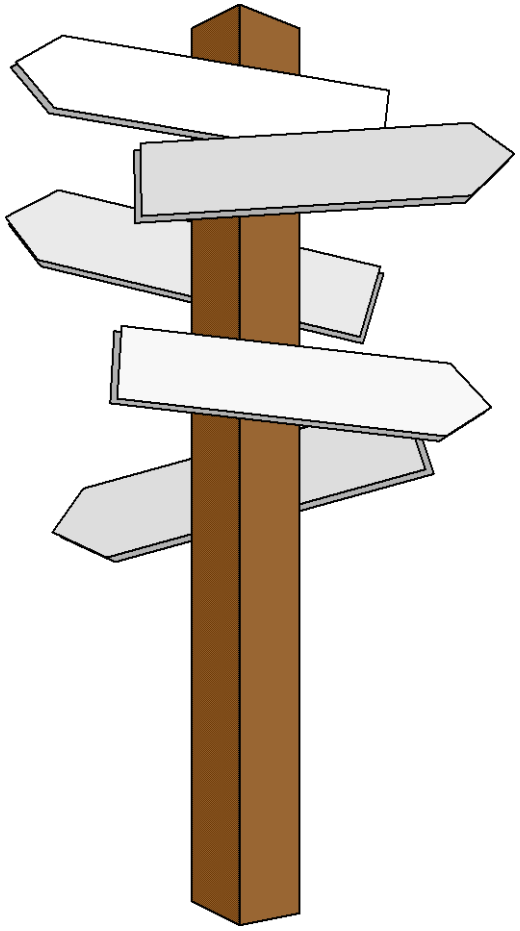
Vorteil:

+ Geringer Platzbedarf von $O(|V| + |E|)$

Nachteil:

– Entscheidung, ob $(i,j) \in E$ in Zeit $O(|E|)$ im schlimmsten Fall

Kapitel 3: Spezielle Algorithmen



RWTH AACHEN UNIVERSITY

3.1.1 Baumdurchläufe
- **Preorder**
- **Inorder**
- **Postorder**
- **Levelorder**

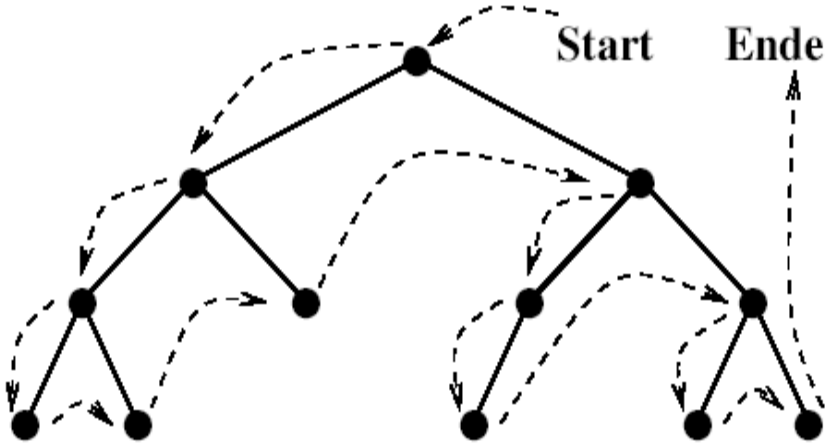
3.1.2 Graphdurchläufe
- **Tiefensuche**
- **Breitensuche**

3.1.3 Anwendungen
- **Arithmetische Ausdrücke**
- **Backtracking**

3.1.4 Branch-and-Bound

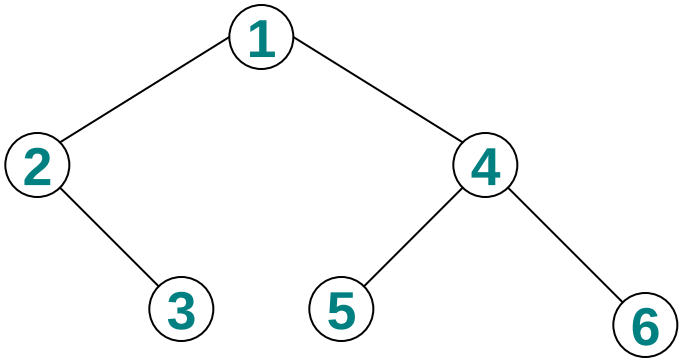
Baumdurchläufe: Preorder

1. Betrachte die Wurzel (und führe eine Operation auf ihr aus)
2. Durchlaufe linken Teilbaum
3. Durchlaufe rechten Teilbaum



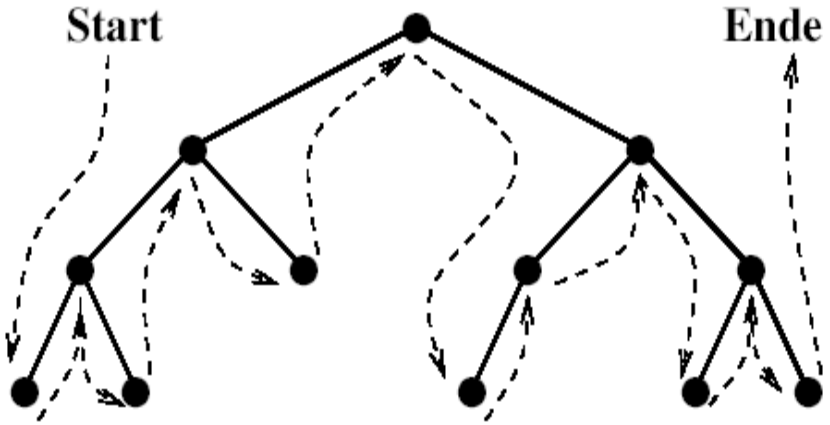
Preorder (wurzel)

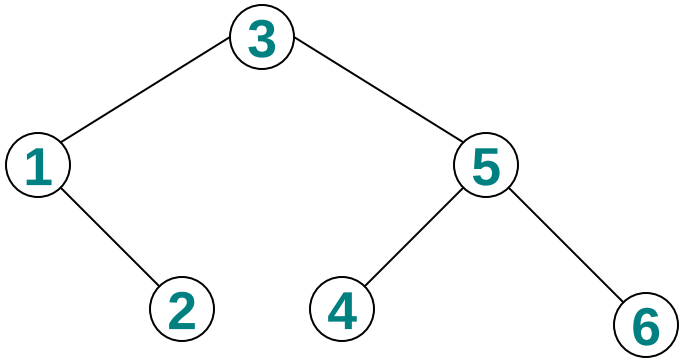
Besuche Wurzel
Preorder (linker Sohn)
Preorder (rechter Sohn)



Baumdurchläufe: Inorder

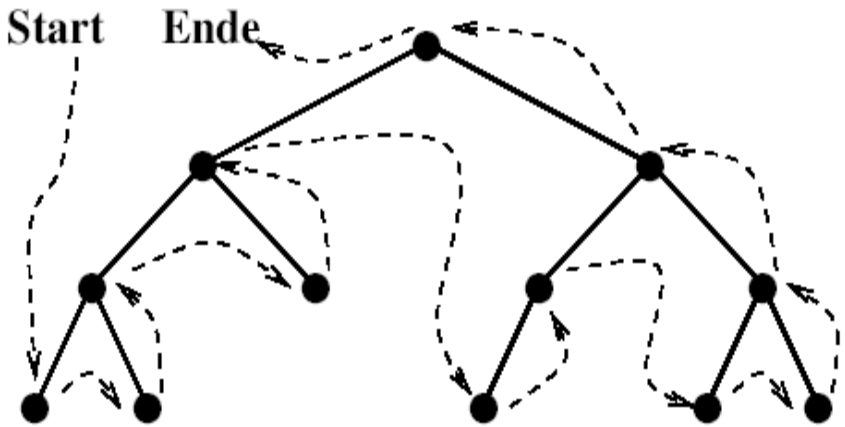
1. Durchlaufe linken Teilbaum
2. Betrachte die Wurzel (und führe eine Operation auf ihr aus)
3. Durchlaufe rechten Teilbaum

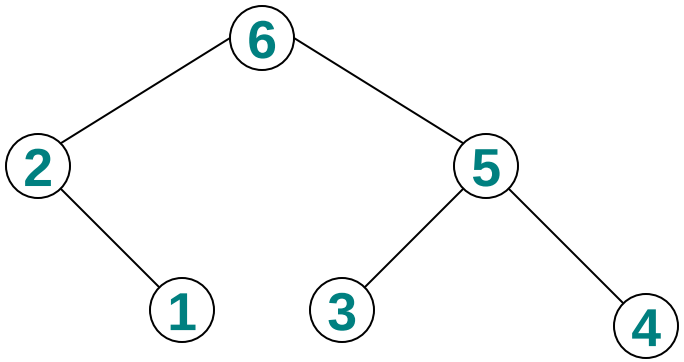




Baumdurchläufe: Postorder

1. Durchlaufe linken Teilbaum
2. Durchlaufe rechten Teilbaum
3. Betrachte die Wurzel (und führe eine Operation auf ihr aus)

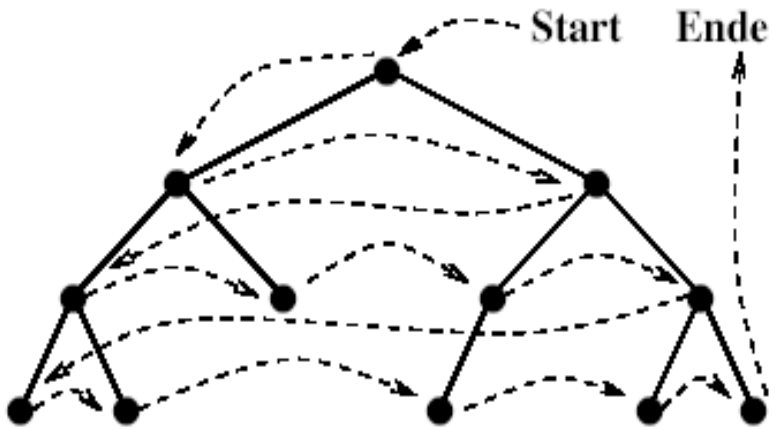




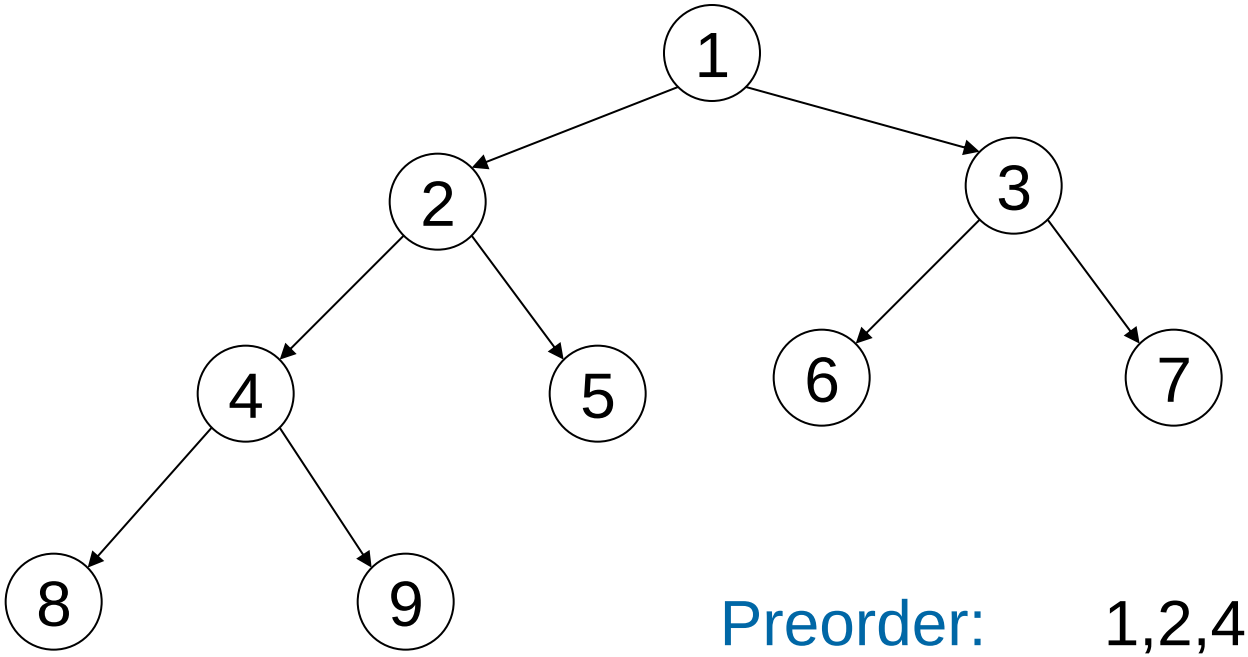
Baumdurchläufe: Levelorder

Durchlaufen eines Baumes Schicht für Schicht:

- Starte bei der Wurzel (Ebene 0)
- Bis die Höhe des Baumes erreicht ist, setze die Ebene um eins höher und gehe von links nach rechts durch alle Knoten dieser Ebene



Beispiel



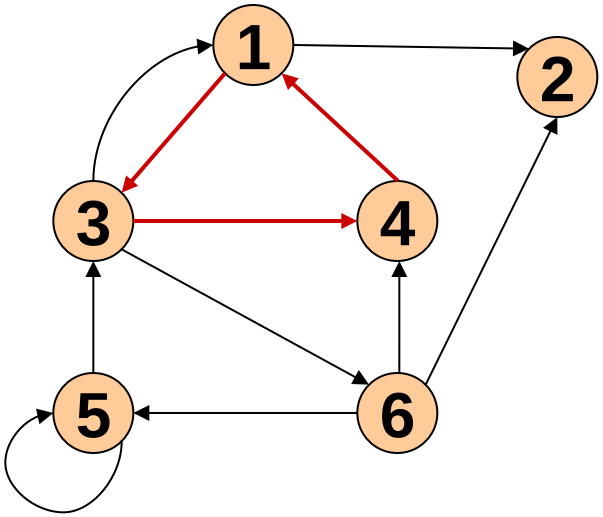
Preorder:	1,2,4,8,9,5,3,6,7
Inorder:	8,4,9,2,5,1,6,3,7
Postorder:	8,9,4,5,2,6,7,3,1
Levelorder:	1,2,3,4,5,6,7,8,9

Struktogramm Levelorder

Durchlaufe Levelorder(v)

```
Initialisiere leere queue
queue.put(v)
Solange queue nicht leer
    u = queue.get()
    Besuche Knoten „u“
    Für jeden Kindknoten „w“
        queue.put(w)
```

- Graphdurchläufe sind prinzipiell ähnlich wie Baumdurchläufe.
- Allerdings müssen die durchlaufenen Knoten markiert werden, damit man auf zyklischen Wegen nicht in Endlosschleifen gerät.



Pre-Order → Tiefensuche
Level-Order → Breitensuche

Initialisierung für DFS und BFS

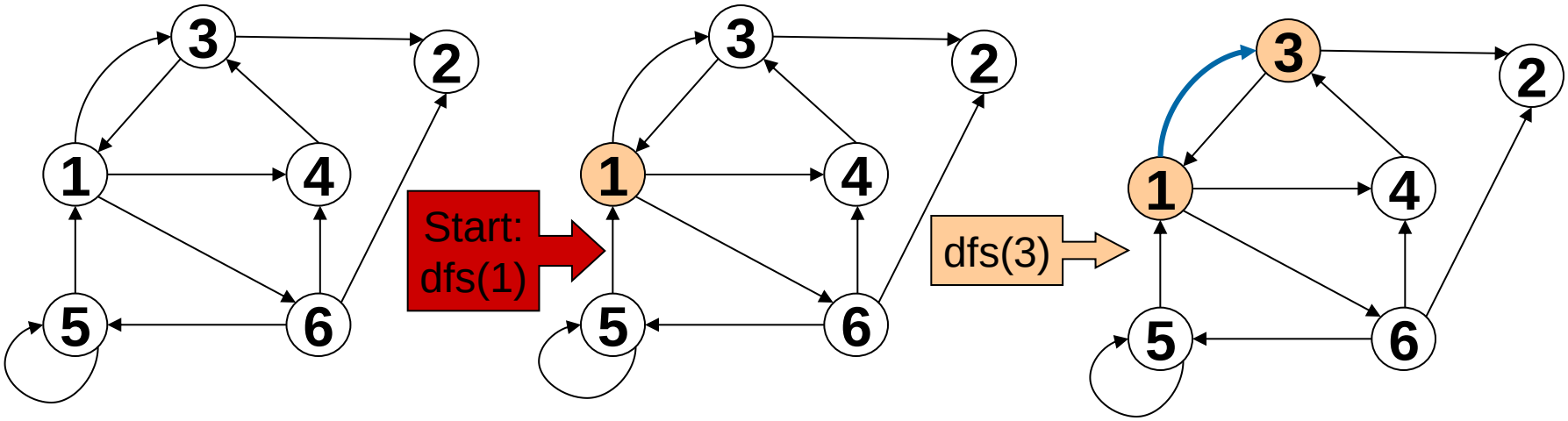
Zu Beginn muss dafür gesorgt sein, dass alle Knoten als „noch nicht besucht“ markiert sind:

```
Für alle Knoten 'v'
    markiere Knoten 'v' als 'unbesucht'
```

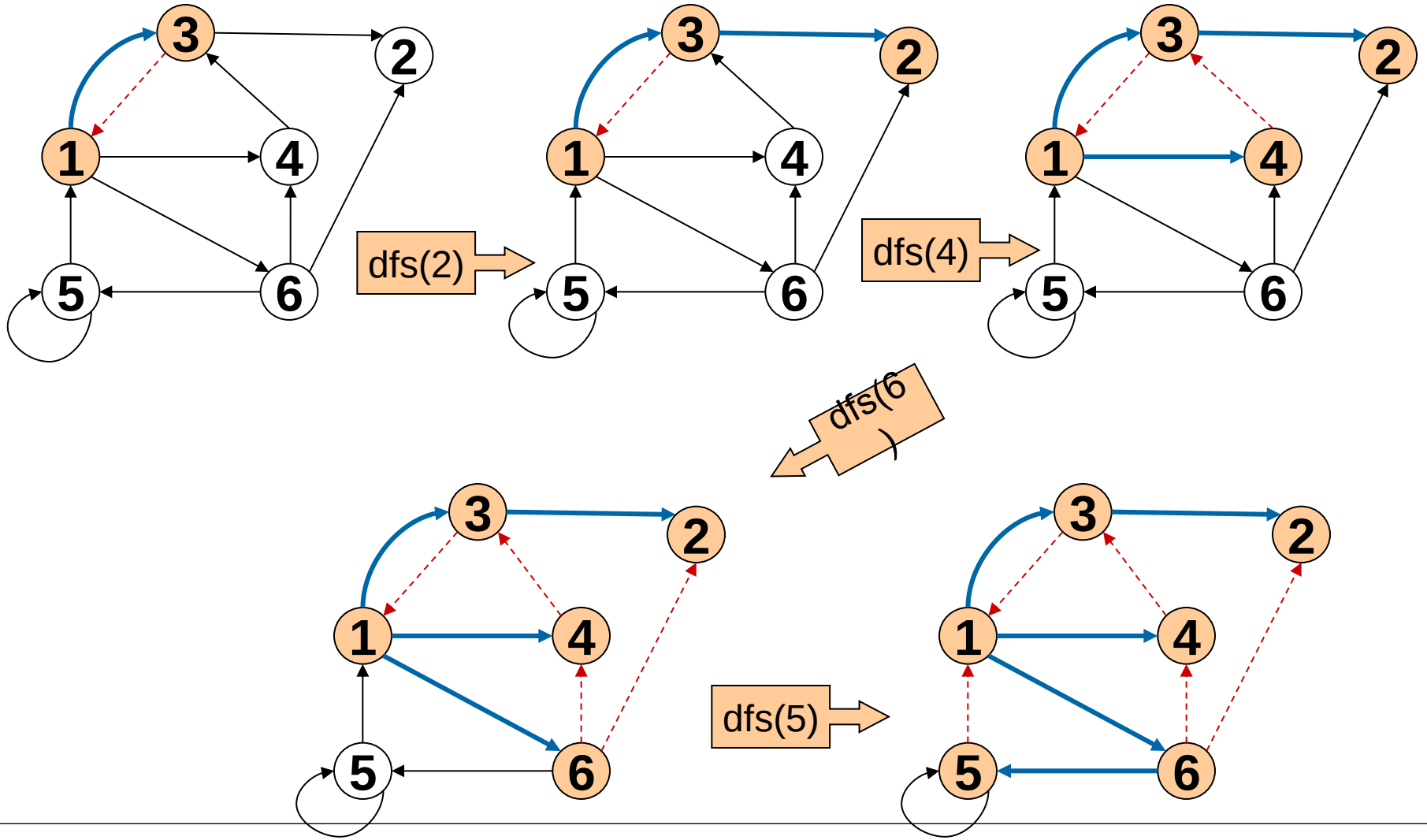
Entspricht Preorder-Durchlauf bei Baum:

1. zunächst alle Knoten als „noch nicht besucht“ markieren
2. Startpunkt wählen (und als „besucht“ markieren)
3. von dort aus möglichst langen Pfad entlang gehen; dabei nur bisher „unbesuchte“ Knoten „besuchen“ (Implementierung rekursiv oder mit explizitem Stack).
4. wenn dann noch nicht alle Knoten besucht worden sind: einen unbesuchten Knoten als neuen Startpunkt wählen und von 3. beginnen

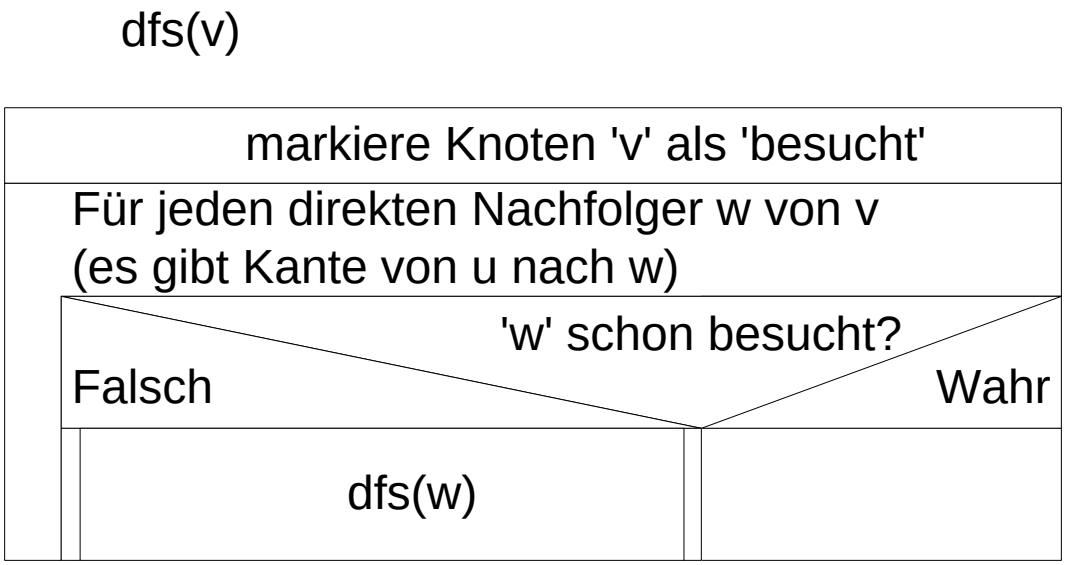
Beispiel zu DFS (1)



Beispiel zu DFS (2)



Tiefensuche (DFS)



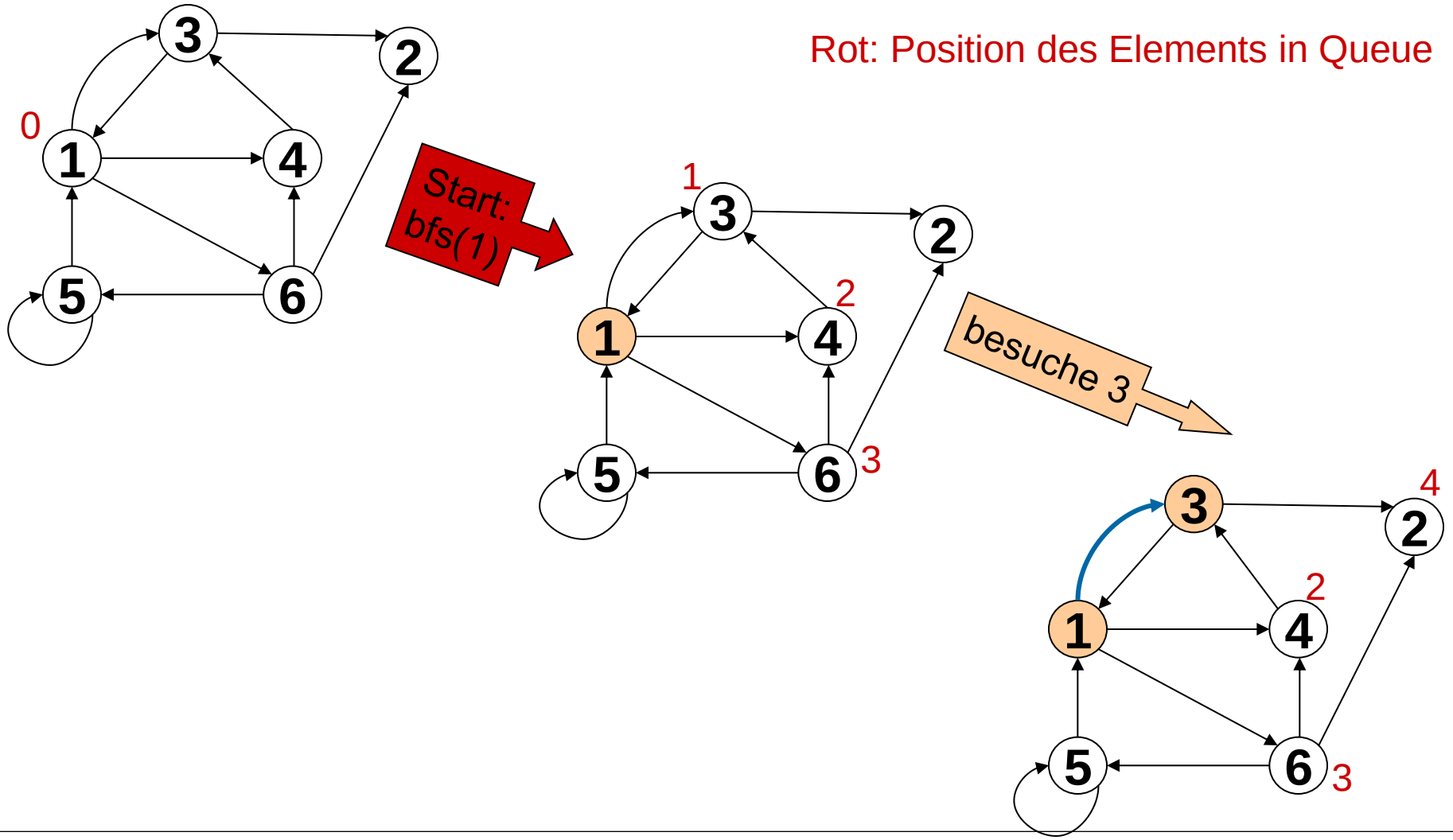
Breitensuche/Breadth-First-Search (BFS)

Entspricht Levelorder-Durchlauf bei Bäumen:

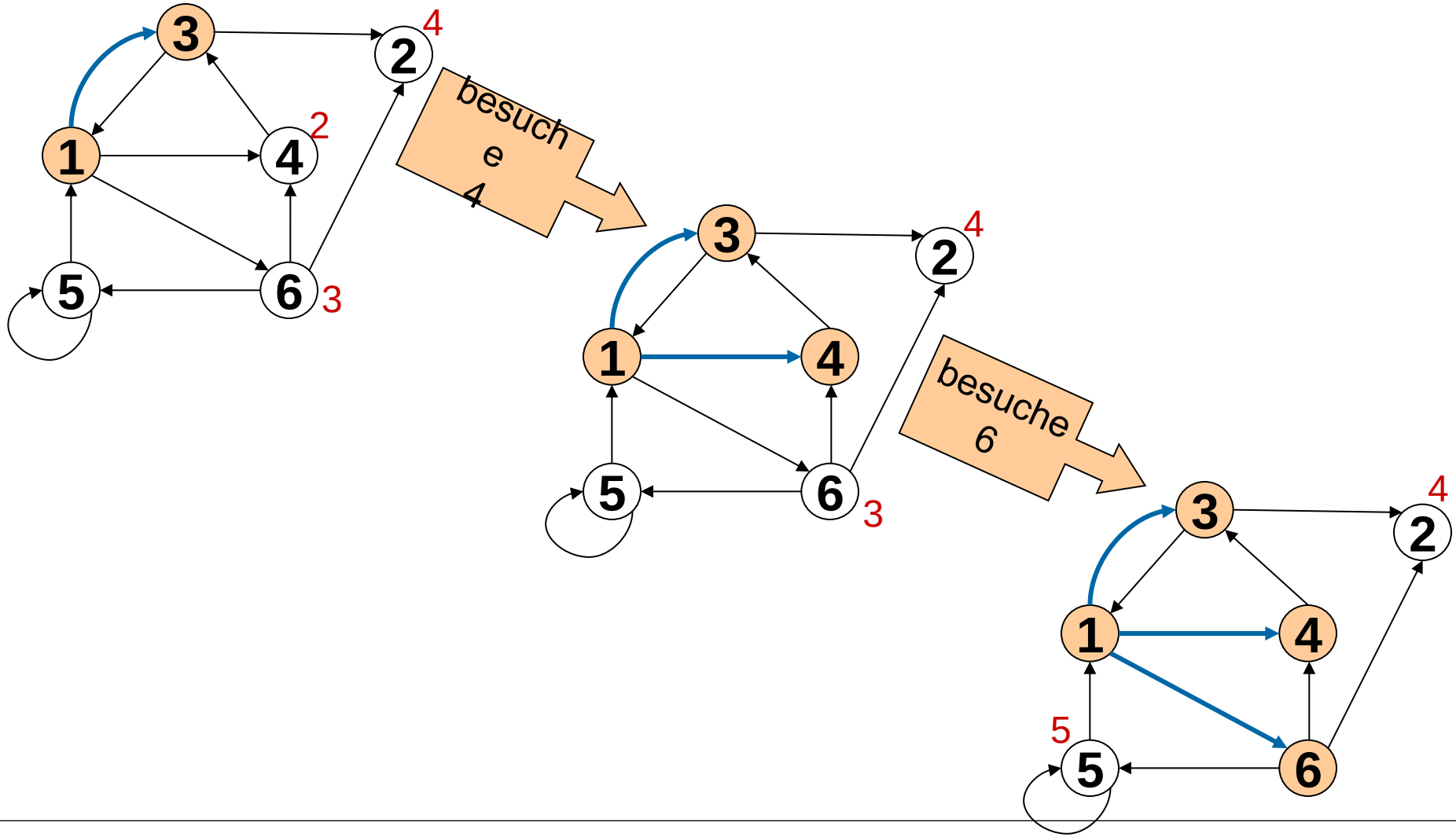
1. zunächst alle Knoten als „noch nicht besucht“ markieren
2. Startpunkt v wählen (und als „besucht“ markieren)
3. Jetzt:
   i. alle von v aus direkt erreichbaren (noch nicht besuchten) Knoten „besuchen“
   ii. alle von v aus über zwei Kanten erreichbaren Knoten „besuchen“
   iii. ... über n Kanten ...

   Implementierung mit einer Queue.
4. wenn dann noch nicht alle Knoten besucht worden sind: einen unbesuchten Knoten als neuen Startpunkt wählen und von 3. beginnen

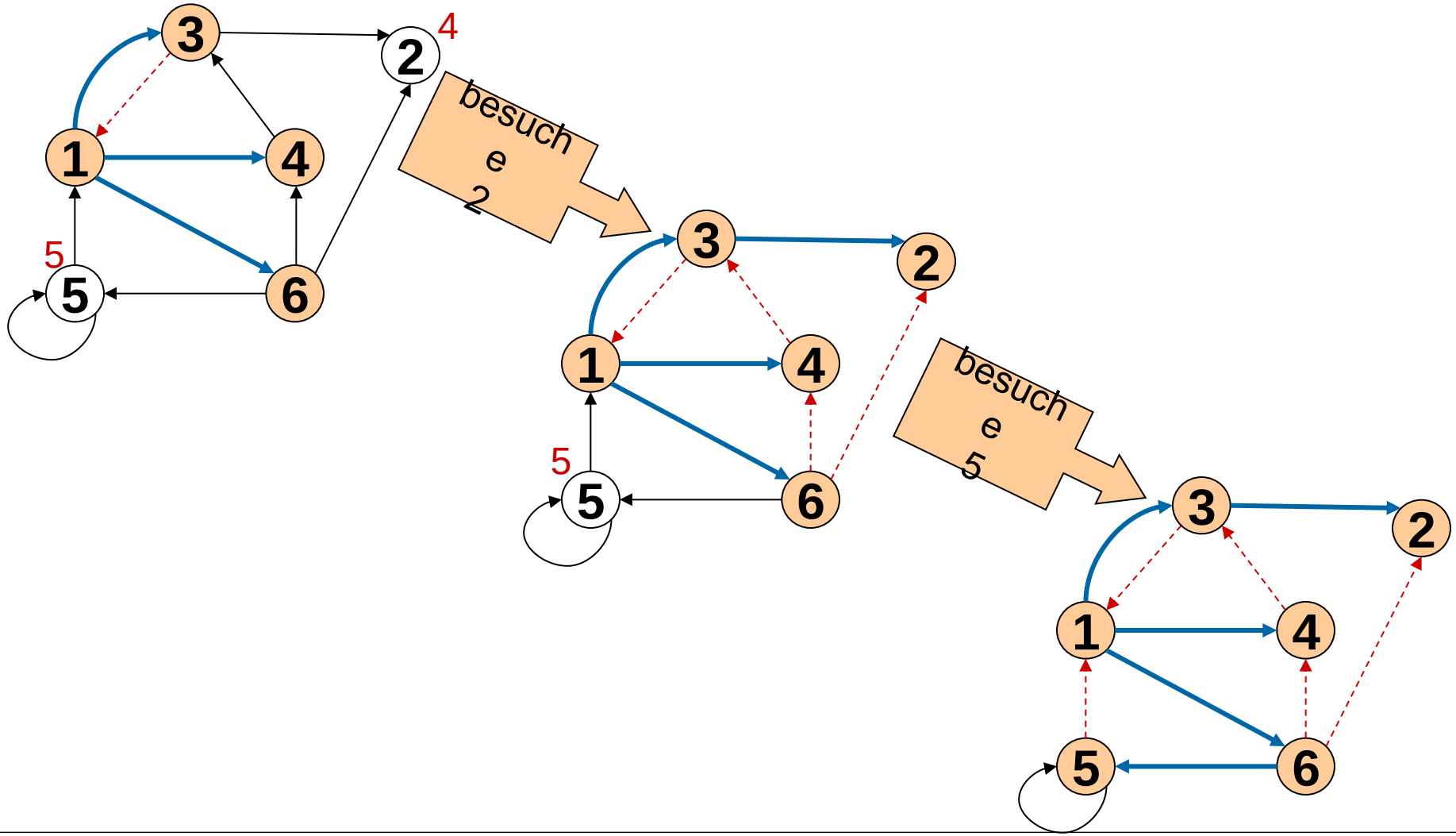
Beispiel zu BFS (1)



Beispiel zu BFS (2)



Beispiel zu BFS (3)



Breitensuche (BFS)

bfs(v)

Initialisiere leere queue
queue.put(v)
Markiere Knoten ’v’ als besucht
Solange queue nicht leer
u = queue.get()
Für jeden direkten Nachfolger w von u (es gibt Kante von u nach w)

'w' schon besucht? Falsch	Wahr
queue.put(w)	
Markiere Knoten ’w’ als besucht	

Bemerkungen zu DFS und BFS

- Zeitkomplexität von DFS und BFS:
  - Bei Speicherung des Graphen mit Adjazenzlisten:
    - DFS: $O(|V| + |E|)$
    - BFS: $O(|V| + |E|)$
  - Bei Speicherung des Graphen mit Adjazenzmatrix:
    - DFS: $O(|V|^2)$
    - BFS: $O(|V|^2)$

- Warum Bezeichnungen Tiefen/Breiten-Suche?
  - Eine Anwendung ist die Suche im Graphen nach einem bestimmten Knoten:
    - Vergleich des Knoteninhalts mit gesuchtem Inhalt und Abbruch bei Gleichheit.
    - Worst-Case-Komplexität bleibt gleich.

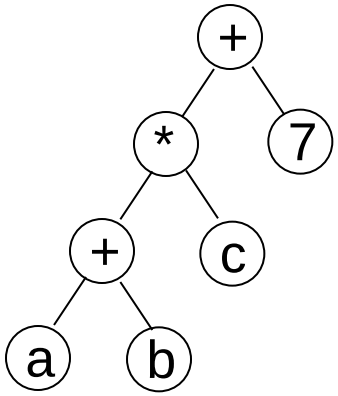
3.1.3 Anwendungen

Arithmet. Ausdrücke in Pre-/Postfix-Notation

- **(a + b) * c + 7**

  Prefix-Notation: + * + a b c 7

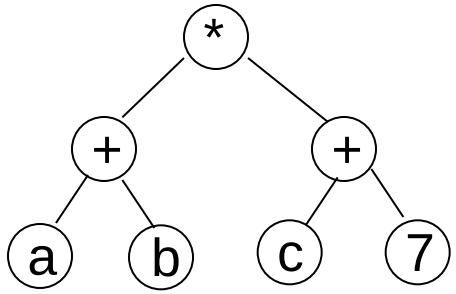
  Postfix-Notation: a b + c * 7 +



- **(a + b) * (c + 7)**

  Prefix-Notation: * + a b + c 7

  Postfix-Notation: a b + c 7 + *



Bemerkung: Bei der Infix-Notation sind i.allg. Klammern notwendig, um die Baumstruktur eindeutig wiedergeben zu können.

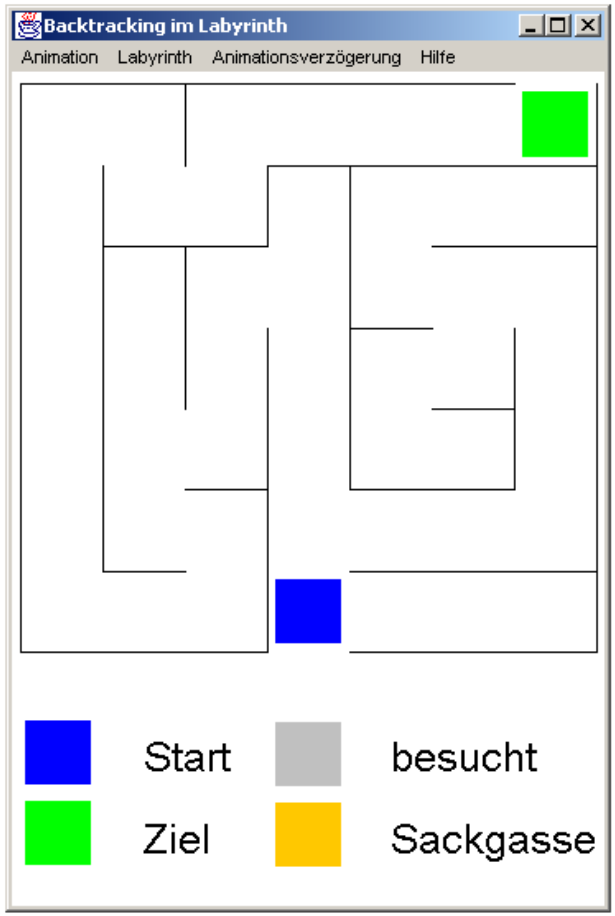
Backtracking

- Situation: Mehrere Alternativen sind in bestimmten Schritten des Algorithmus möglich
  - z.B. 4 Richtungen bei Suche in einem Labyrinth: $\rightarrow$, $\downarrow$, $\uparrow$, $\leftarrow$

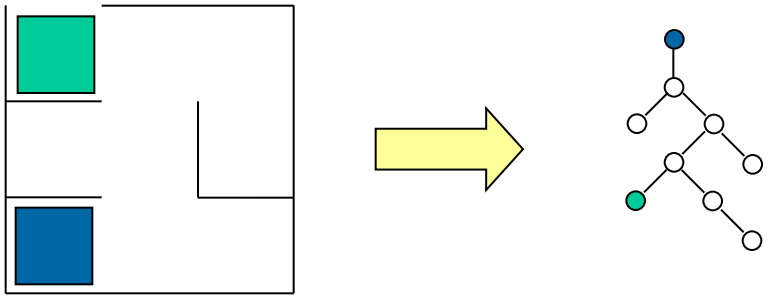
- Lösung mit Backtracking:
  - Wähle eine Alternative und verfolge diesen Weg weiter.
  - Falls man so eine Lösung des Problems findet, ist man fertig.
  - Ansonsten gehe einen Schritt zurück und verfolge rekursiv eine andere (bisher noch nicht probierte) Alternative in diesem Schritt.
  - Falls alle Alternativen erfolglos probiert wurden: Einen Schritt zurückgehen ...

Beispiel: Labyrinthsuche

- „Wie kommt die (virtuelle) Maus zum Käse?“
- Bei Abzweigungen durchsucht sie erst die eine Richtung. Führt sie in eine Sackgasse, kehrt sie an die Abzweigung zurück (Backtracking)und wählt eine andere Richtung.
- Rekursiver Algorithmus.
- Bild aus: http://www.swisseduc.ch/informatik/vortraege/backtracking/index.html



Baumdurchlauf / Backtracking

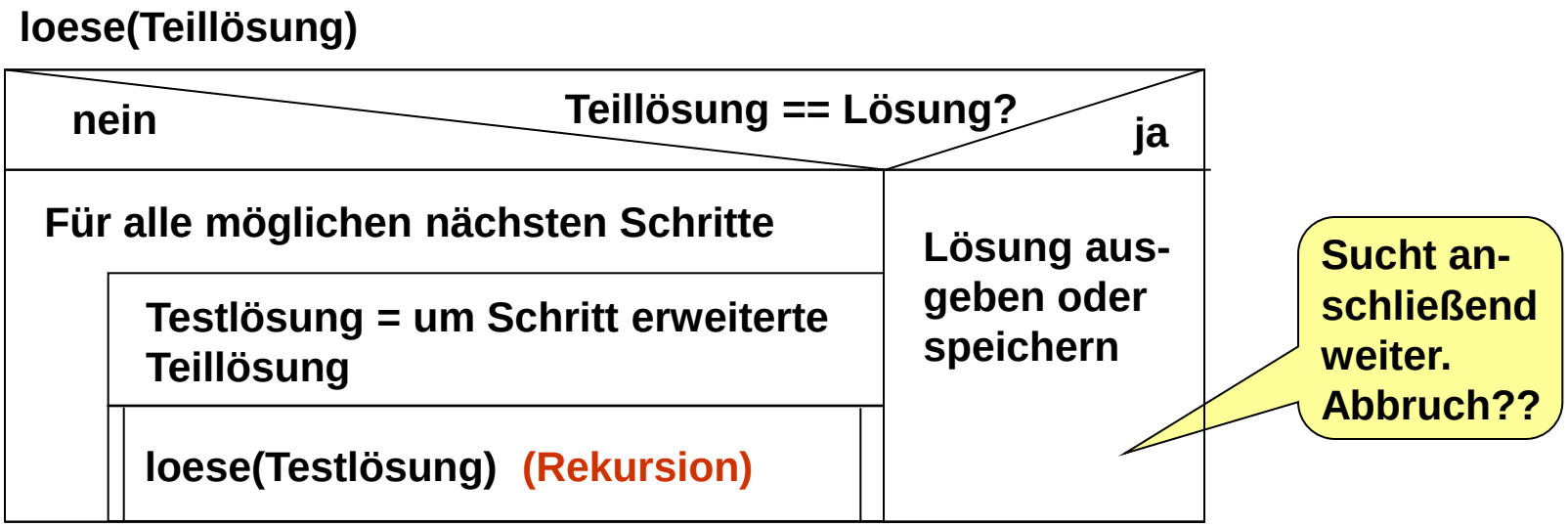


Baumdurchlauf (mit Pre-Order) und Suchen eines bestimmten Knotens im Baum.

Das „Hochlaufen“ des Baumes (nach Beendigung eines Rekursionsaufrufs) entspricht dem Zurückgehen eines Schritts im Labyrinth.

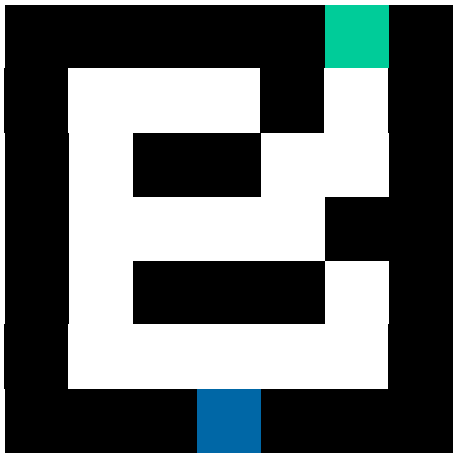
Daher hat diese Suche den Namen **Backtracking**.

Struktogramm



Labyrinthsuche: Spezifikation

- Wände bestehen aus einem Block
- Ränder sind Wände (außer Start und Ziel)
  $\Rightarrow$ Erster Schritt immer eindeutig
- Abbruch der Suche nach Erreichen des Ziels



- Klarheit des Codes geht vor Performance
- Objektorientiertheit

Labyrinthsuche: Objektorientierung

RWTH AACHEN UNIVERSITY

Location

- x, y: int
+Location(x: int, y: int)
+add(loc: Location): Location
+equals(loc: Location): boolean

Backtracking

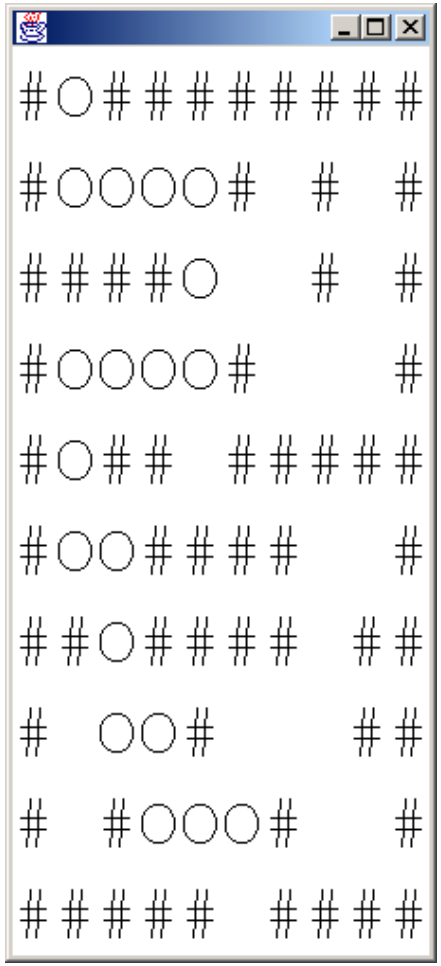
+Backtracking(m: Maze)
+solve(loc: Location,
lastLoc: Location): boolean
+static main(String[] args): int

Maze

+Maze(int width, int height)
+isFree(loc: Location): boolean
+isEnd(loc: Location): boolean
+getStart(): Location
+getFirstStep(): Location
+markTrack(loc: Location,
vis: boolean): void
+print(): void

(Code ist auf Kursseite verfügbar.)

Labyrinthsuche: Grafische Ausgabe

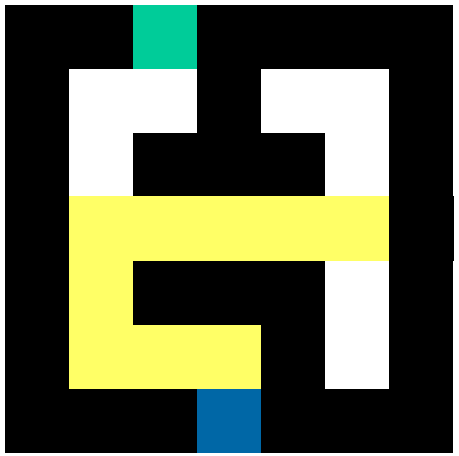


Mehr Intelligenz (Heuristik)

```
Location[] directions = new Location[] {up, right, down,
   left};
```

```
for (int i=0; i<4; i++) {
  test = loc.add(directions[i]);
```

Es ist bekannt, dass das Ziel oben liegt



Bei manchen Wegen ist es sinnlos, sie weiterzuvefolgen.

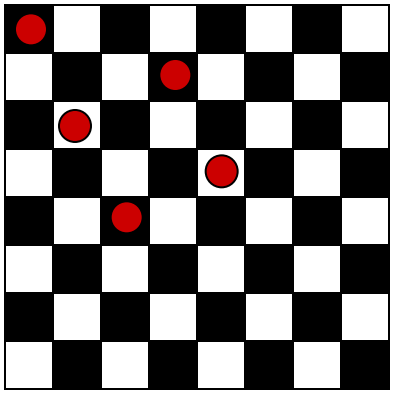
Bsp. zu Backtracking: 8-Damen-Problem

Aufgabenstellung: 8 Damen auf Schachbrett so aufstellen, dass sie sich gegenseitig nicht schlagen können (In keiner Diagonalen, Waagerechten oder Senkrechten darf mehr als eine Dame stehen)

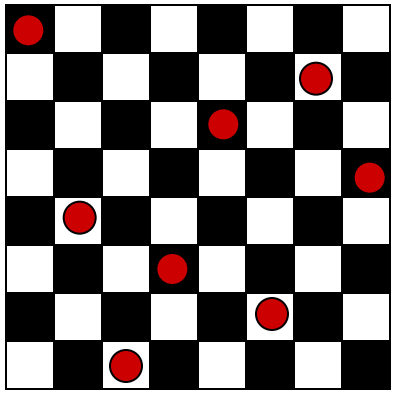
Eng verwandt: "Springerproblem"

Sackgasse

?



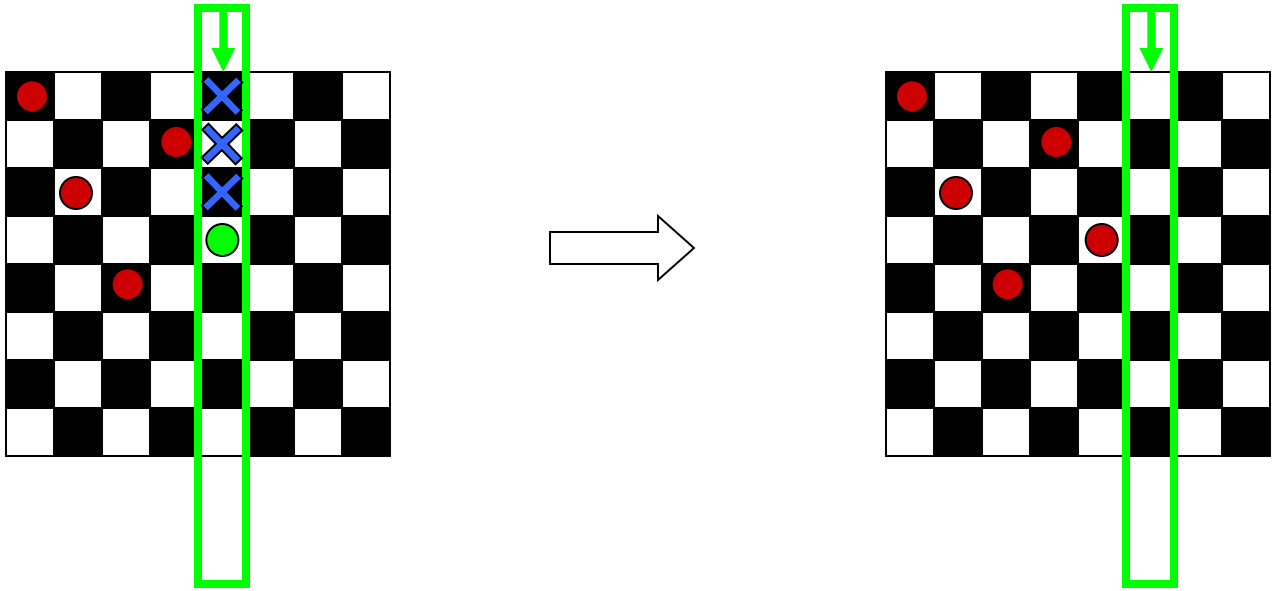
eine Lösung



Lösungsansatz: 8-Damen-Problem

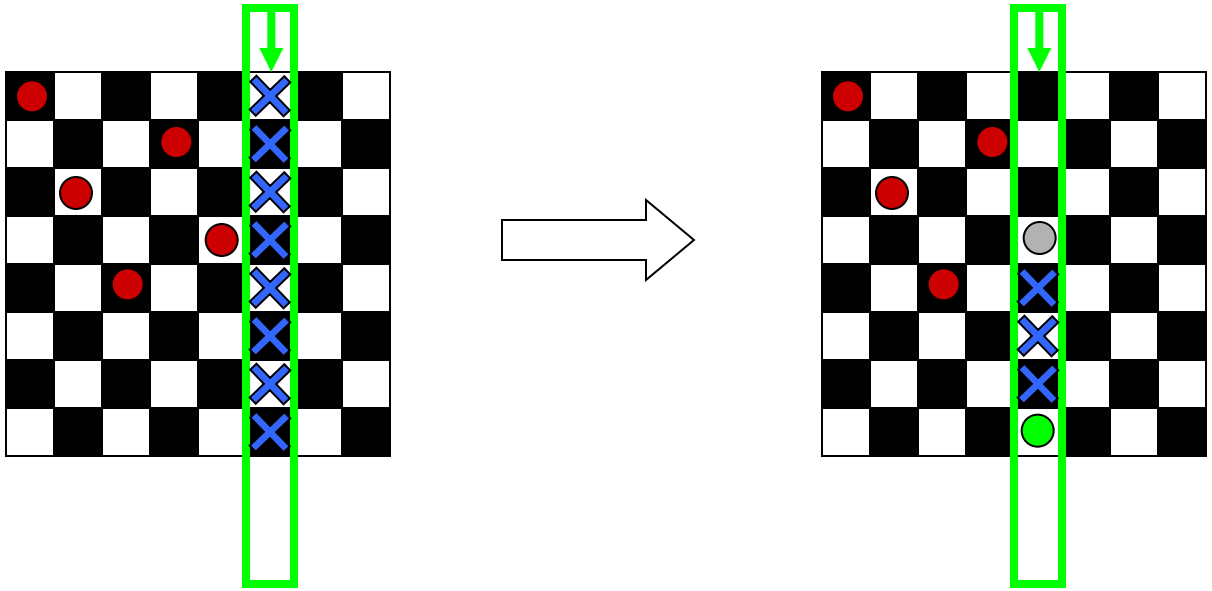
Teile Problem in 8 Schritte auf:

- In Schritt i: eine Dame in i-ter Spalte platzieren unter Berücksichtigung der bereits platzierten i-1 Damen.
  Dazu: Platzierung in Zeilen j=1-8 ausprobieren: Erste Zeile wählen, in der die i-te Dame nicht „bedroht“ ist.
- Falls eine sichere Position gefunden: (i+1)-te Dame genauso behandeln.



Lösungsansatz: 8-Damen-Problem (2)

- Falls sichere Platzierung in keiner Zeile möglich: einen Schritt zurück gehen und alternative Position für (i-1)-te Dame wählen.
- Verfahren funktioniert rekursiv



Realisierung: 8-Damen-Problem

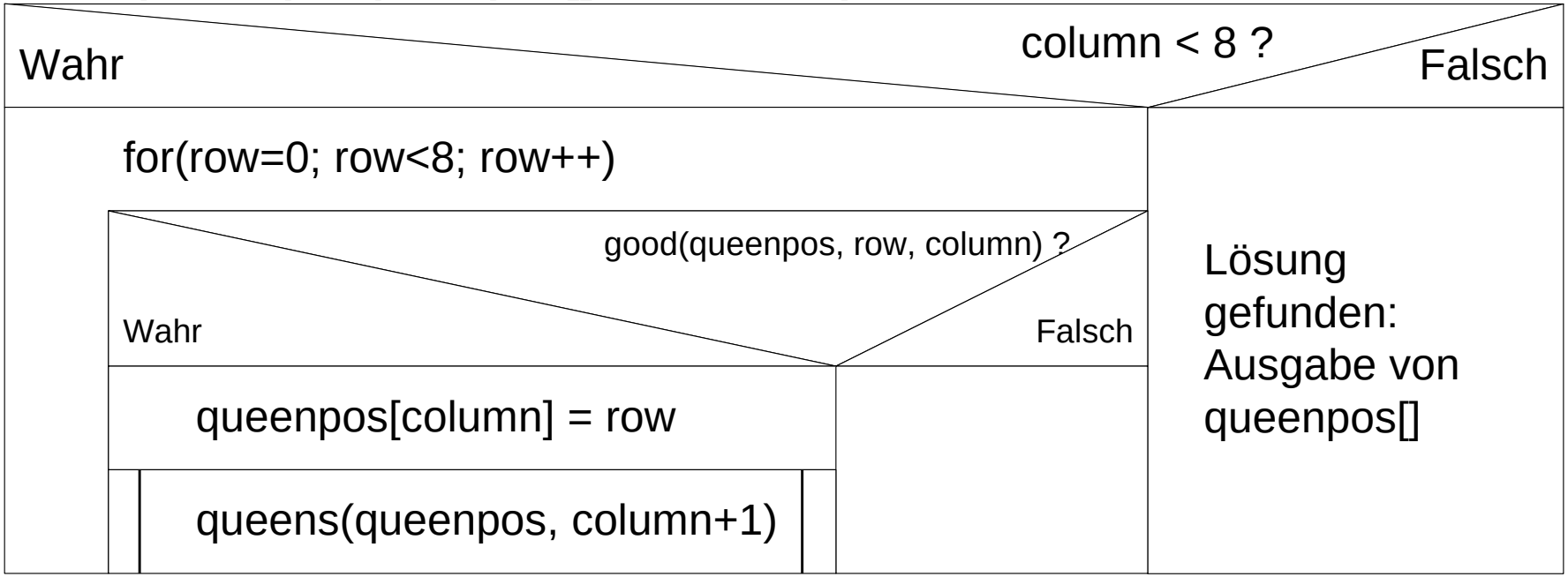
Array queenpos[8]: Positionen der Damen, d.h.

queenpos[i] = Nummer der Zeile, in der i-te Dame in Spalte i platziert wird.

Realisierung: 8-Damen-Problem

Aufruf: queens(queenpos,0)

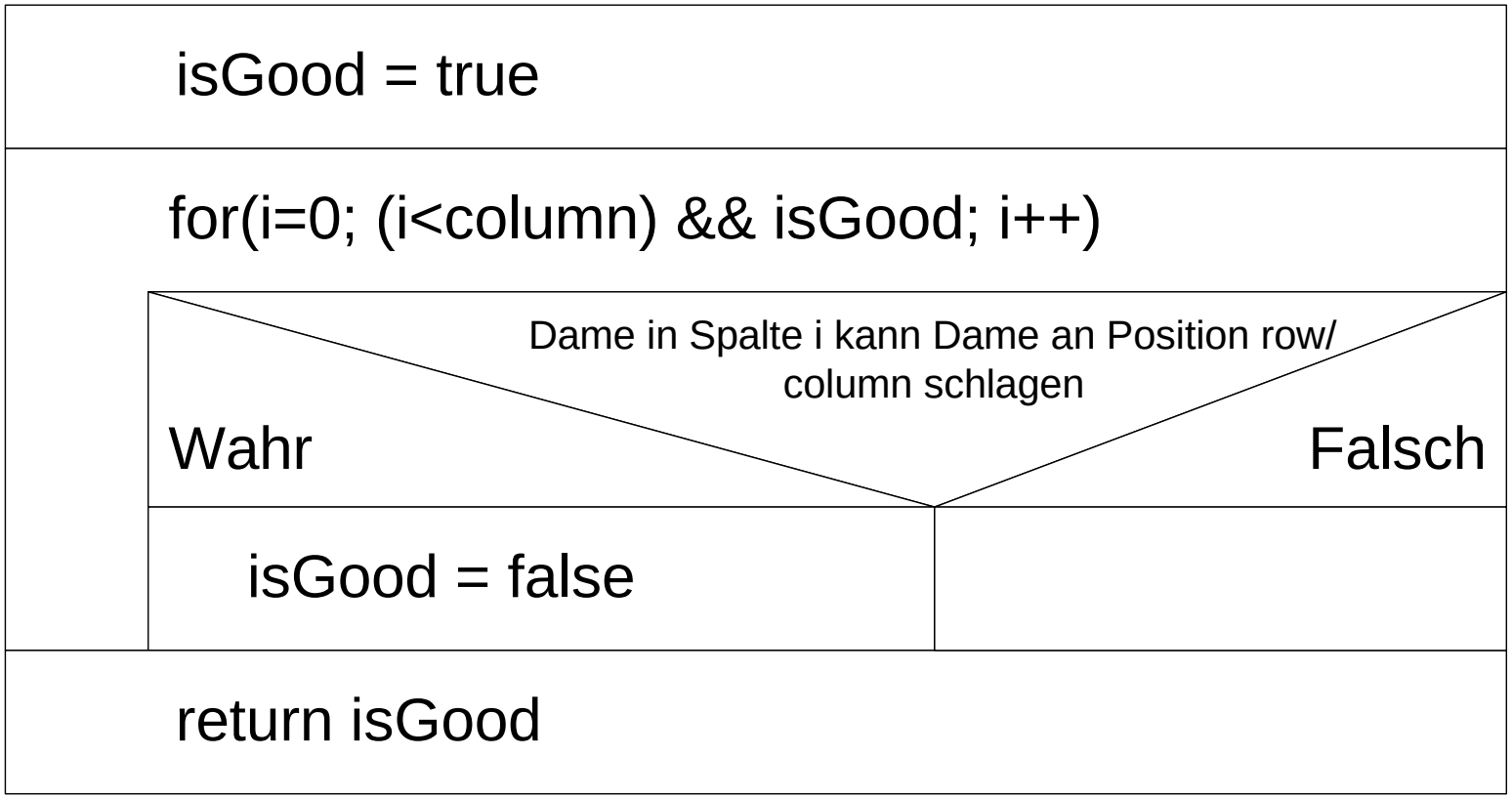
queens(int queenpos[], int column):



Bemerkung: Verfahren findet alle Lösungen

Test, ob Position geschlagen wird

boolean good(int queenpos[], int row, int column):



Bemerkungen zu Backtracking

- Backtracking-Algorithmen können exponentiellen Aufwand haben!
- Durch Einführung von Zusatzbedingungen möglichst viele Sackgassen ausschließen.
- Symmetriebedingungen ausnutzen.

Gegeben: Es gibt n Gegenstände. Jeder Gegenstand i hat ein Gewicht g_i und einen Wert w_i.

In einen Rucksack passen Gegenstände bis zum Gesamtgewicht G.

Aufgabenstellung: Packe Gegenstände so in den Rucksack, dass der Gesamtwert maximal wird.

$$\left(\sum c_i g_i < G\right) \wedge \left(\sum c_i w_i \text{ maximal}\right)$$

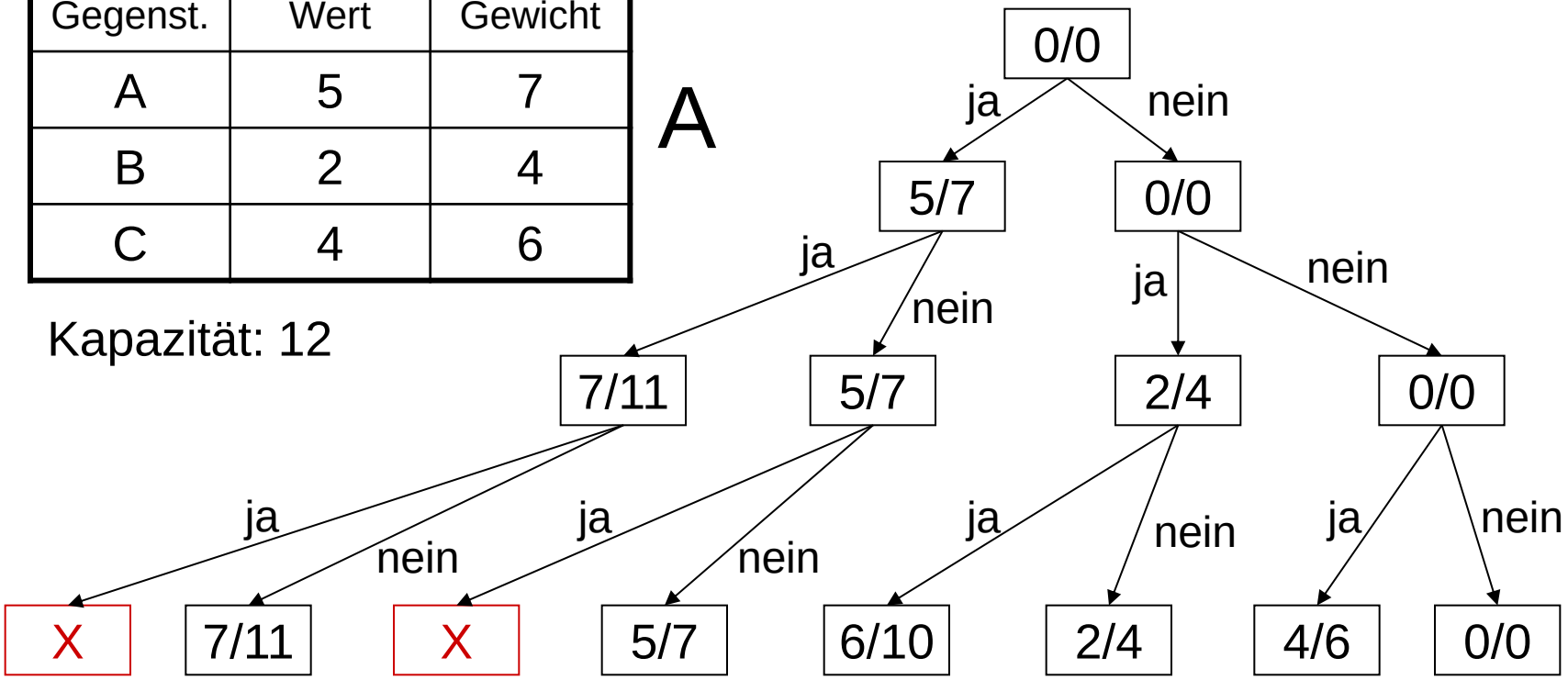
$c_i \in \{0,1\}$

Komplexität: mit Binärbedingung: 2^n

Beispiel

Gegenst.	Wert	Gewicht
A	5	7
B	2	4
C	4	6

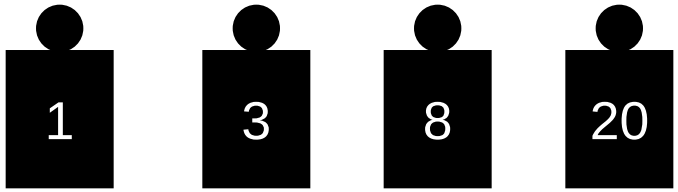
Kapazität: 12



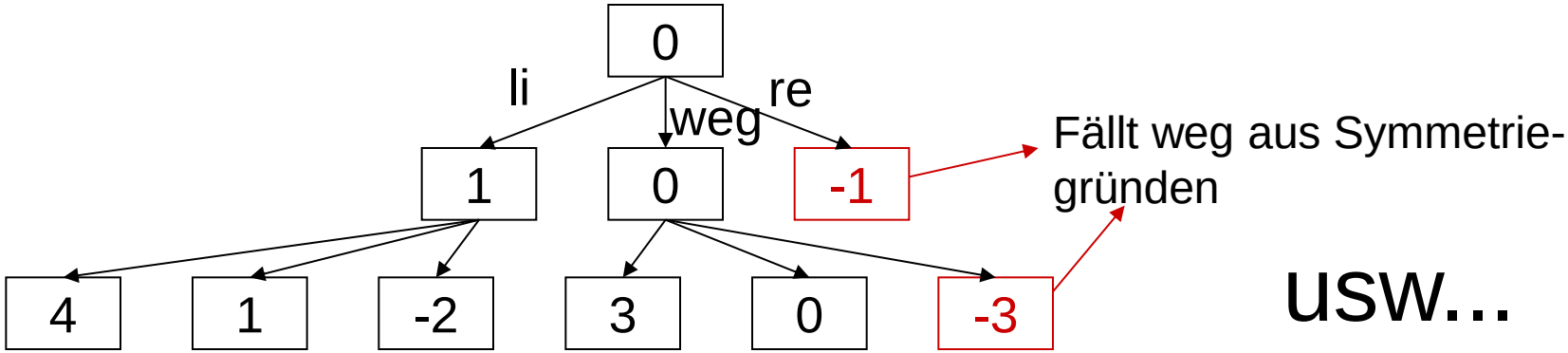
Suche des Blattes mit dem größten Wert

Ähnliches Beispiel: Waage

- Zur Verfügung steht eine Balkenwaage und 4 Gewichte.



- Welche Gewichte kann man ausmessen?



- Günstigste Wege von einer Quelle: Dijkstra-Algorithmus
- Günstigste Wege zwischen allen Knotenpaaren: Floyd-Algorithmus
- Existiert ein Weg zwischen zwei Knoten? Warshall–Algorithmus

Kürzeste-Wege-Probleme

- Eigentlich: Suche nach günstigsten Wegen in *gewichteten* Graphen: Gewichte ≅ Kosten;
- Bei Anwendung auf *ungewichtete* Graphen (Gewichte=1) ergibt sich: „kürzeste“ Wege.

- Praktische Bedeutung für Transport- und Kommunikationsnetzwerke

- (einige) Problemarten:

  1. Bestimme günstigsten Weg von festem Knoten („Quelle“) zu allen anderen Knoten („Single-source shortest-path problem“).

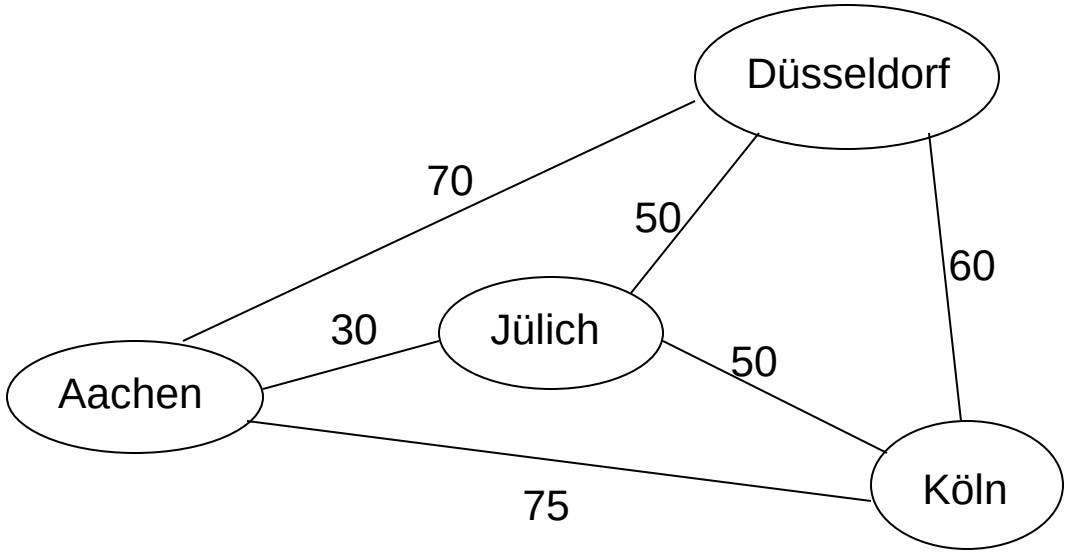
  2. Bestimme günstigste Wege zwischen allen Knotenpaaren („All-pairs shortest-path problem“).

  3. Bestimme günstigsten Weg zwischen zwei gegebenen Knoten.

  4. Existiert ein Weg zwischen zwei Knoten?

Anwendungsbeispiel

Spedition fährt täglich im Dreieck Aachen-Düsseldorf-Köln und möchte die Kosten minimieren.
Kantenkosten im Graphen sind z.B. Kilometer oder benötigte Zeit.



1) Kostenminimaler Weg von Aachen zu allen anderen Orten?
2) Kostenminimaler Weg zwischen allen Orten?
3) Kostenminimaler Weg zwischen Aachen und Köln?

Algorithmus von Dijkstra (1959):

Gegeben: Graph $G = (V,E)$ dessen Bewertungsfunktion folgende 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: $C(i,i) = 0$ (spezielle Konvention, falls nötig)
- Bei ungewichtetem Graph (Gewichte=1):
  Ergebnis = Länge der kürzesten Wege

Idee: Iterative Erweiterung einer Menge von günstig erreichbaren Knoten

Algorithmus von Dijkstra (1959) (2)

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

1. Initialisierung: $S = \{$ Startknoten $\}$

2. Beginnend mit Quelle alle ausgehenden Kanten betrachten (analog BFS). Nachfolgerknoten v mit günstigster Kante zu S hinzunehmen.

3. Jetzt: berechnen, ob die Knoten in $V \setminus S$ günstiger über v als „Zwischenstation“ erreichbar ist, als ohne Umweg über v.

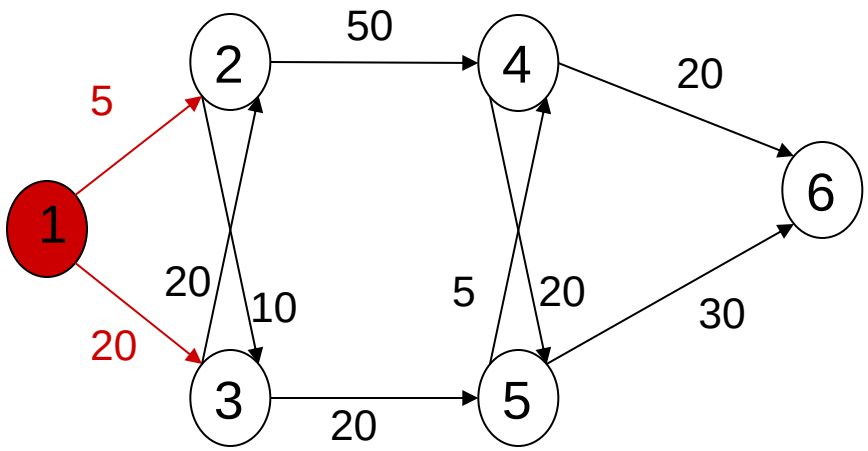
4. Danach: denjenigen Knoten v' zu S hinzunehmen, der nun am günstigsten zu erreichen ist. Bei zwei gleich günstigen Knoten wird ein beliebiger davon ausgewählt.

Neuberechnen der Wegekosten (3.) und Erweitern von S (4.) wiederholen, bis alle Knoten in S sind.

Beispiel (1)

Initialisierung: $S = \{$ Startknoten $\}$

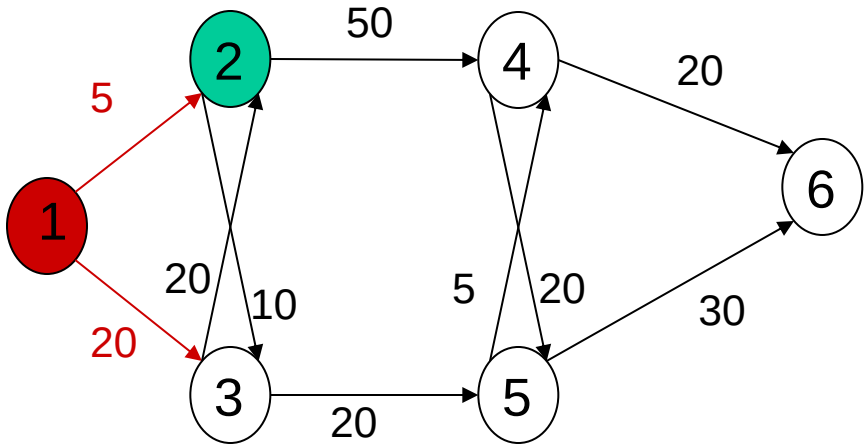
Beginnend mit Quelle alle ausgehenden Kanten betrachten (analog BFS).



Knoten	Abstand
1	0
2	5
3	20
4	
5	
6	

Beispiel (2)

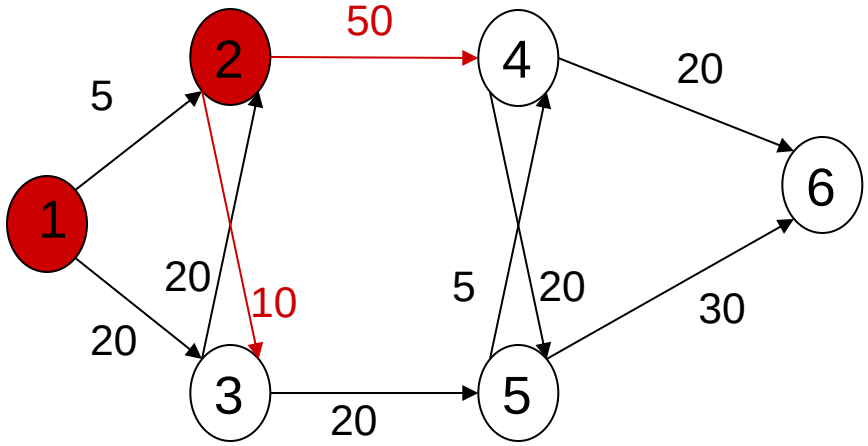
Nachfolgerknoten *v* mit günstigster Kante zu *S* hinzunehmen.



Knoten	Abstand
1	0
2	5
3	20
4	
5	
6	

Beispiel (3)

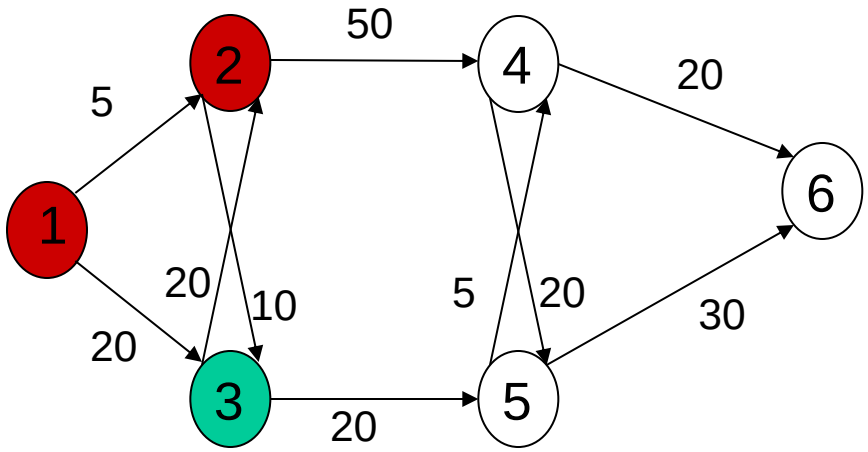
3. Jetzt: berechnen, ob die Knoten in $V \setminus S$ günstiger über v als „Zwischenstation“ erreichbar ist, als ohne Umweg über v.



Knoten	Abstand
1	0
2	5
3	~~20~~ 15
4	55
5	
6	

Beispiel (4)

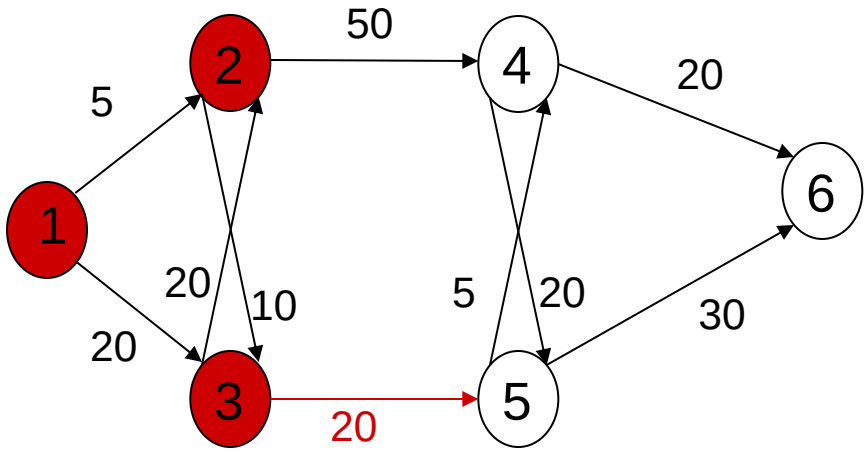
4. Danach: denjenigen Knoten *v'* zu *S* hinzunehmen, der nun am günstigsten zu erreichen ist.



Knoten	Abstand
1	0
2	5
3	15
4	55
5	
6	

Beispiel (5)

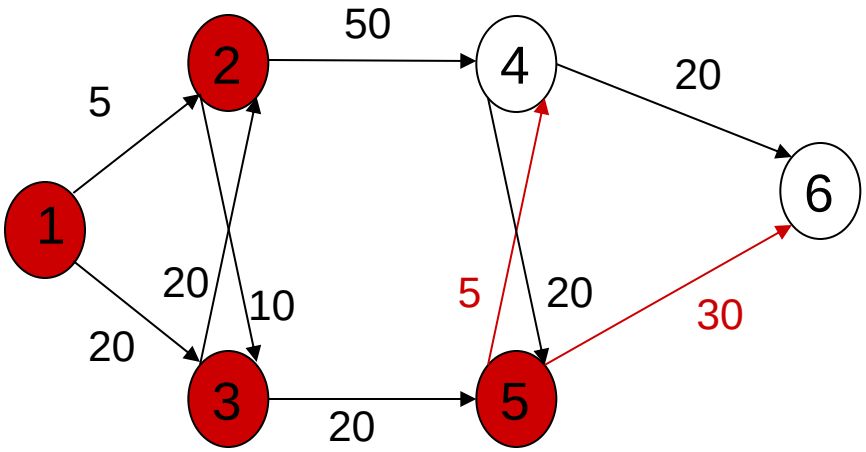
Neuberechnen der Wegekosten.



Knoten	Abstand
1	0
2	5
3	15
4	55
5	35
6	

Beispiel (6)

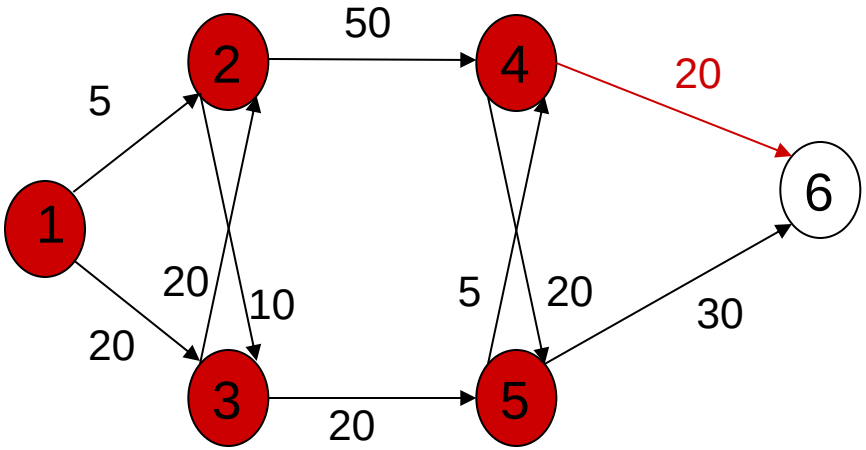
- wiederholen, bis alle Knoten in *S* sind.



Knoten	Abstand
1	0
2	5
3	15
4	~~55~~ 40
5	35
6	65

Beispiel (7)

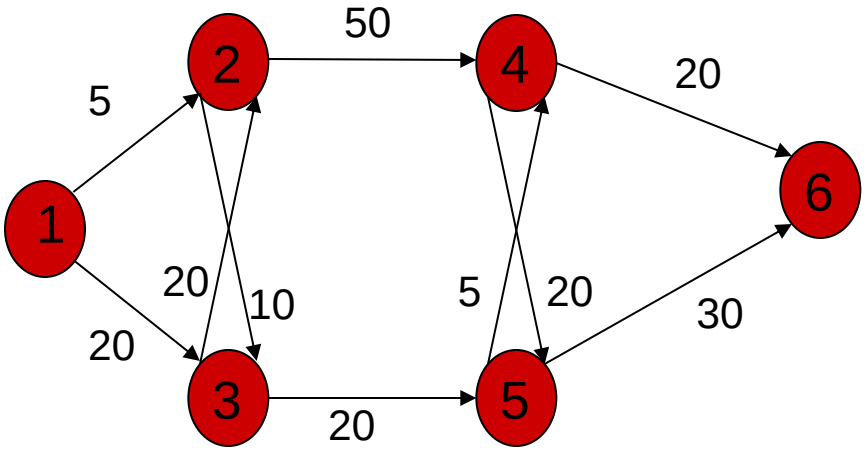
- wiederholen, bis alle Knoten in *S* sind.



Knoten	Abstand
1	0
2	5
3	15
4	40
5	35
6	~~65~~ 60

Beispiel (8)

- wiederholen, bis alle Knoten in *S* sind.



Knoten	Abstand
1	0
2	5
3	15
4	40
5	35
6	60

Dijkstra-Algorithmus: Datenstrukturen

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

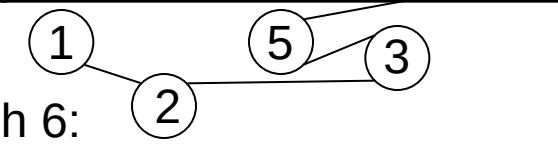
$d[i]$: aktuell günstigste Kosten irgendeines Weges von Quelle zu v_i

$p[i]$: Wenn zusätzlich zu günstigsten Kosten auch nach günstigstem Weg zu jedem Knoten gefragt ist:

- Speichere direkten Vorgänger von v_i auf günstigstem Weg in $p[i]$
- $p[i]$ wird mit Index der Quelle initialisiert.
- Nach Ablauf des Algorithmus kann günstigster Weg zu jedem Knoten mit Hilfe des Feldes p zurückverfolgt werden.

Beispiel zum Ablauf bei Dijkstra

Iteration	S	v_i	d[2]	d[3]	d[4]	d[5]	d[6]	p[2]	p[3]	p[4]	p[5]	p[6]
Init.	{1}	1	5	20				1	1			
1	{1,2}	2	5	15	55			1	2	2		
2	{1,2,3}	3	5	15	55	35		1	2	2	3	
3	{1,2,3,5}	5	5	15	40	35	65	1	2	5	3	5
4	{1,2,3,4,5}	4	5	15	40	35	60	1	2	5	3	4
5	{1,2,3,4,5,6}	6	5	15	40	35	60	1	2	5	3	4



Rückverfolgen des günstigsten Weges z.B. von 1 nach 6:

- Vorgänger von 6 = 4
- Vorgänger von 4 = 5
- Vorgänger von 5 = 3, ...

d.h. günstigster Weg ist 1-2-3-5-4-6 (mit Kosten 60)

Algorithmus von Dijkstra (1): Initialisieren

Quelle (Index '1') zur Menge 'S' hinzufügen

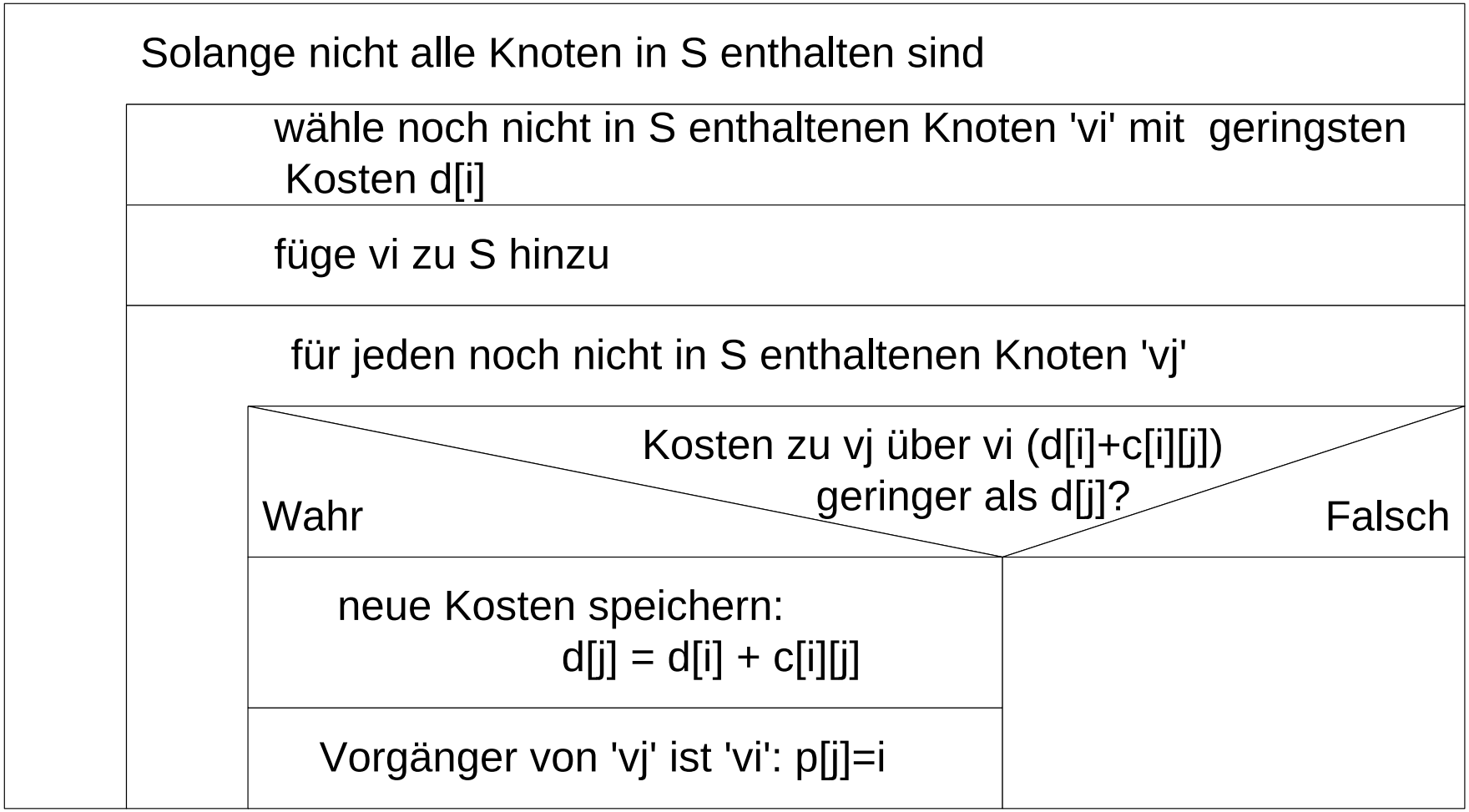
für alle anderen Knoten 'vi'

Kantenkosten zwischen Quelle und 'vi' in Feld 'd' speichern (d[i]=c[1][i])

Vorgänger von 'vi' ist Quelle '1': p[i]=1

falls Kante zwischen vi und Quelle besteht

Algorithmus von Dijkstra (2)



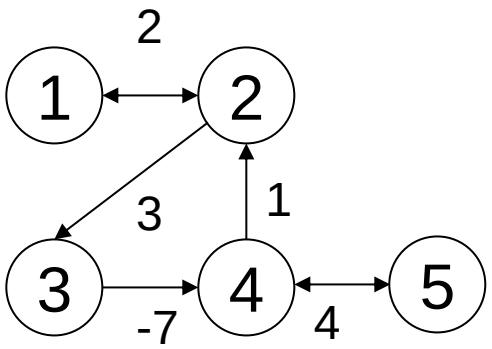
Dijkstra-Algorithmus: Bemerkungen

- Zeitkomplexität des Dijkstra-Algorithmus:

  Annahme: Speicherung des Graphen mit Adjazenzmatrix

$$T_{Dijkstra}(|V|) \in O(|V|^2)$$

- Nachteil: Keine negativen Gewichte möglich
  - Ausweichen auf den Bellman-Ford-Algorithmus
  - Negative Zykel: Kein günstigster Weg möglich

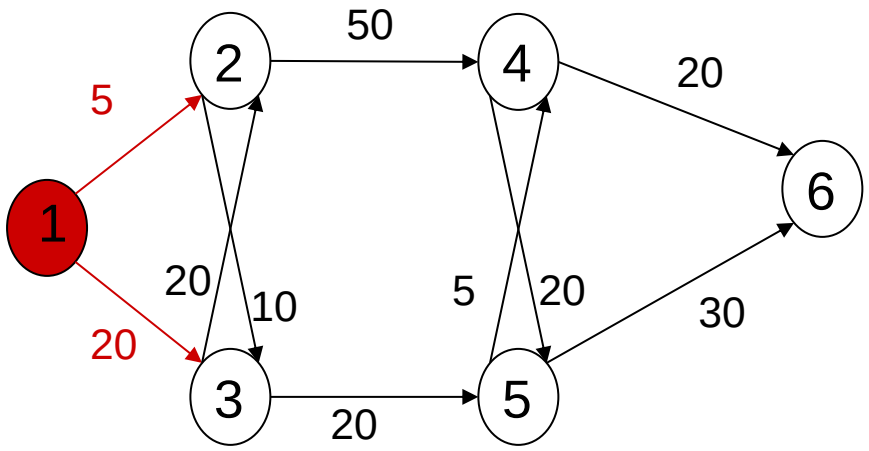


Negativer Zykel 2-3-4-2
Die Kosten 1→5 können beliebig klein werden.

Bellman-Ford-Algorithmus

- Dynamische Programmierung
- Nachteil O(v·e) (statt $O(v^2)$ bei Dijkstra)
- Vorteil: negative Kosten möglich (keine neg. Zykel).

Liste der Kanten aufstellen



1 → 2	5
1 → 3	20
2 → 3	10
2 → 4	50
3 → 2	20
3 → 5	20
4 → 5	20
4 → 6	20
5 → 4	5
5 → 6	30

Beispiel (1)

Berechnung des Abstands d_x zwischen Knoten 1 und Knoten x

- Schritt 1: $d_1 = 0$
- Schritt 2: Für jede Kante a → e:
  - Falls d_a bekannt
    - Falls $d_a + g < d_e$ (g: Gewicht der Kante) oder d_e unbekannt:
      - Setze d_e auf $d_a + g$.

1 → 2	5
1 → 3	20
2 → 3	10
2 → 4	50
3 → 2	20
3 → 5	20
4 → 5	20
4 → 6	20
5 → 4	5
5 → 6	30

Knoten	Abstand
1	0
2	5
3	~~20~~ 15
4	~~55~~ 40
5	35
6	~~75~~ 65

- Optimaler Wert für alle Knoten, die optimal mit 1 Schritt erreicht werden können.

Beispiel (2)

Berechnung des Abstands d_x zwischen Knoten 1 und Knoten x

- Schritt 3: Wiederhole Schritt 2

1 → 2	5
1 → 3	20
2 → 3	10
2 → 4	50
3 → 2	20
3 → 5	20
4 → 5	20
4 → 6	20
5 → 4	5
5 → 6	30

Knoten	Abstand
1	0
2	5
3	15
4	55
5	35
6	~~65~~ 60

- Optimaler Wert für alle Knoten, die optimal mit 2 Schritten erreicht werden können.

Beispiel (3)

Berechnung des Abstands d_x zwischen Knoten 1 und Knoten x

- Schritt 4: Wiederhole Schritt 2
  - Keine Änderung mehr →fertig

- Spätestens nach v-1 Wiederholungen sind alle Abstände bestimmt (mehr als v-1 Schritte benötigt keine optimale Wegstrecke).

Knoten	Abstand
1	0
2	5
3	15
4	55
5	35
6	65

Beispiel (2)

Berechnung des Abstands d_x zwischen Knoten 1 und Knoten x

- Wiederhole Schritt 2

1 → 2	5
1 → 3	20
2 → 3	10
2 → 4	50
3 → 2	20
3 → 5	20
4 → 5	20
4 → 6	20
5 → 4	5
5 → 6	30

Knoten	Abstand
1	0
2	5
3	~~20~~ 15
4	45
5	55
6	

- Optimaler Wert für alle Knoten, die mit 2 Schritten erreicht werden können.

Günstigste Wege zwischen allen Knotenpaaren

Gegeben (ähnlich wie für Dijkstra-Algorithmus):
Graph G = (V,E) mit folgender Bewertungsfunktion:

- 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 : $C(i,j) = \infty$

Aufgabenstellung: Bestimme für alle geordneten Paare (v,w) den kürzesten Weg von v nach w.

Lösungsmöglichkeiten:

1. Wende *Dijkstra* -Algorithmus für alle Knoten v als Quelle an
   $\Rightarrow$ Zeitkomplexität $O(|V|^3)$.
2. Verwende den *Floyd* -Algorithmus.

Grundidee zum Algorithmus von Floyd

|*V*| Iterationen:

- 1. Schritt: 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. Schritt: zusätzlich Umwege über Knoten 2 betrachten.

- k-ter Schritt: Umwege über Knoten k betrachten, usw.

Floyd-Algorithmus nutzt eine |*V*|x|*V*| Matrix *A[i][j]* , um Kosten der günstigsten Wege zu speichern (Subscript nur zum Bezeichnen der Iteration):

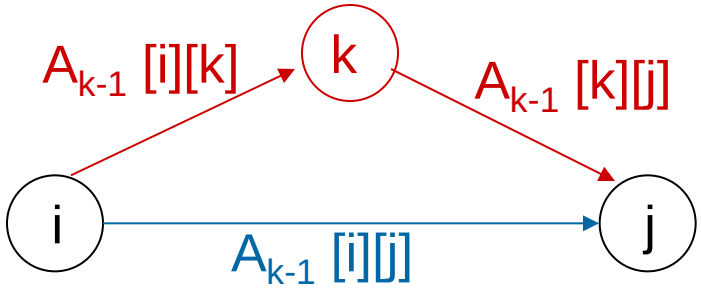
A_k[i][j] := minimale Kosten, um über irgendwelche der Knoten in {1,...,k} vom Knoten i zum Knoten j zu gelangen

Algorithmus von Floyd

A[i][j]= Kosten des aktuell günstigsten Wegs :

- Initialisierung: $A_0[i][j] = C(i,j) \quad \forall\ i \neq j$; Diagonalelemente $A_0[i][i] = 0$
- |V| Iterationen mit „dynamischer Programmierung“:
  Iterationsformel zur Aktualisierung von A[i][j] :

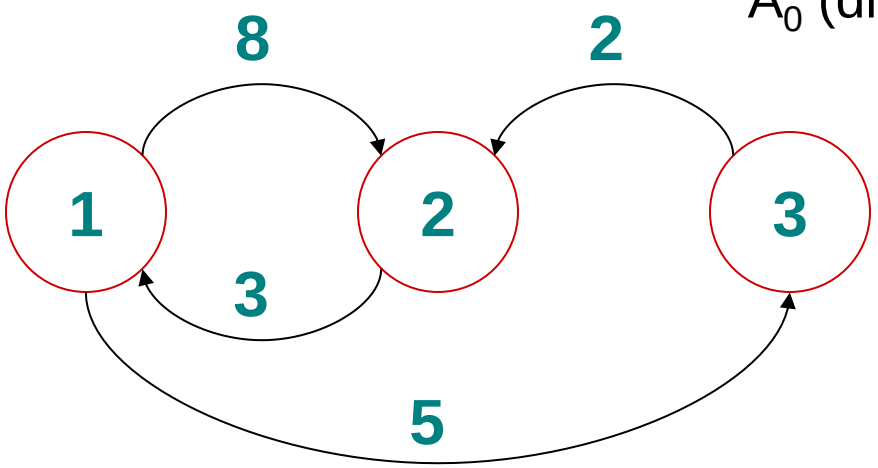
$$A_k[i][j] = \min \{ A_{k-1}[i][j], A_{k-1}[i][k]+A_{k-1}[k][j] \}$$



- Zum Schluss: $A_{|V|}[i][j]$ = Kosten des günstigsten Wegs von i nach j.

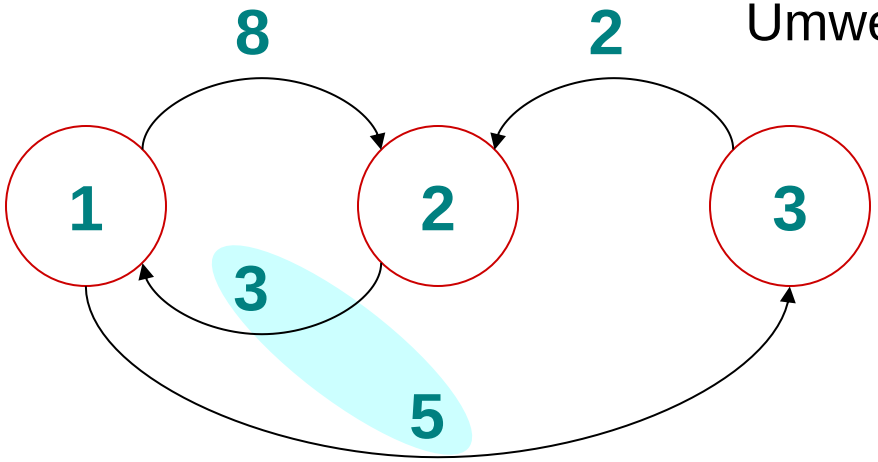
Bemerkung: $A_k[i][k] = A_{k-1}[i][k]$ und $A_k[k][j] = A_{k-1}[k][j]$

Beispiel zum Floyd-Algorithmus (1)



A_0 (direkter Weg)

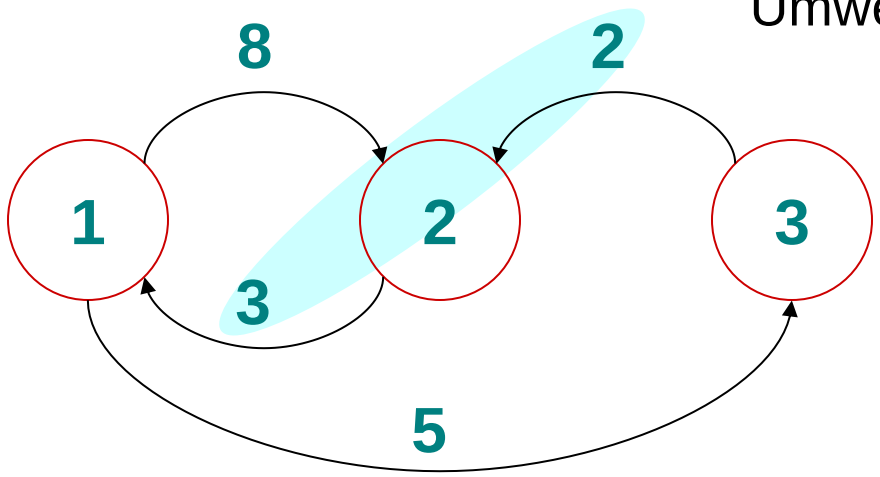
von \ nach	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0



Umweg über 1

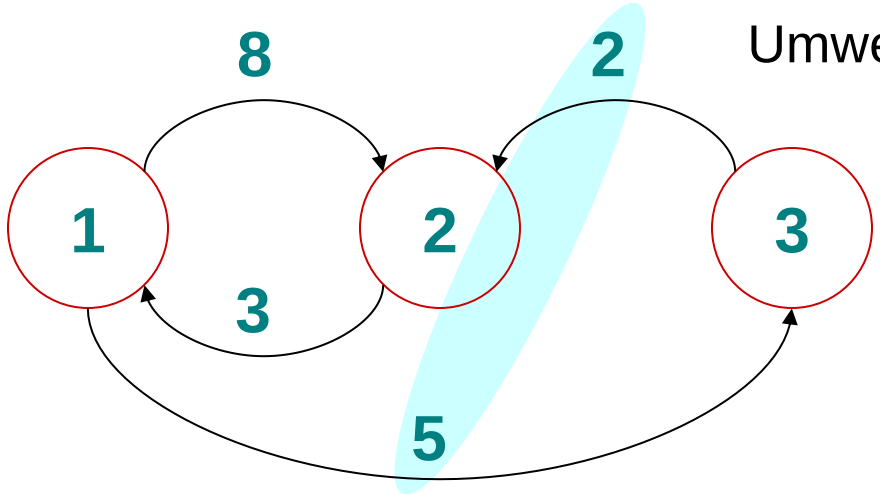
von \ nach	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

Beispiel zum Floyd-Algorithmus (1)



Umweg über 2

von \ nach	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0



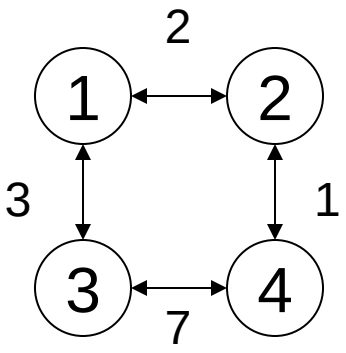
Umweg über 3

von \ nach	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

Beispiel zum Floyd-Algorithmus (2)

$$P_3 = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{bmatrix}$$

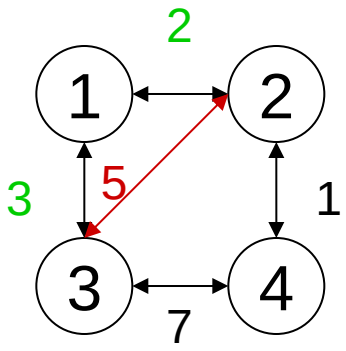
2. Beispiel zum Floyd-Algorithmus (1)



A_0 (direkter Weg)

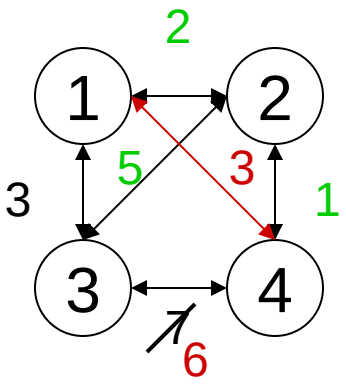
von \ nach	1	2	3	4
1	0	2	3	∞
2	2	0	∞	1
3	3	∞	0	7
4	∞	1	7	0

Umweg über 1



von \ nach	1	2	3	4
1	0	2	3	∞
2	2	0	5	1
3	3	5	0	7
4	∞	1	7	0

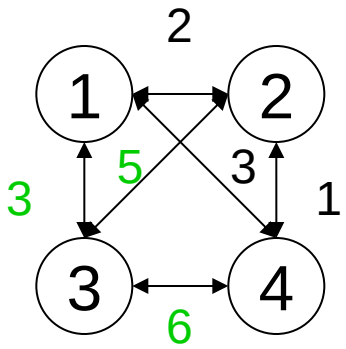
2. Beispiel zum Floyd-Algorithmus (2)



Umweg über 2

von \ nach	1	2	3	4
1	0	2	3	3
2	2	0	5	1
3	3	5	0	6
4	3	1	6	0

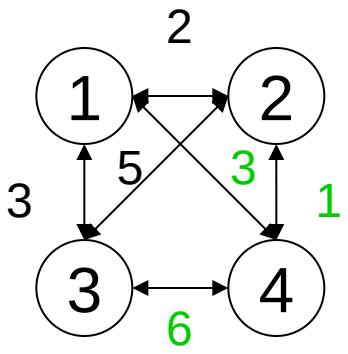
Umweg über 3



von \ nach	1	2	3	4
1	0	2	3	3
2	2	0	5	1
3	3	5	0	6
4	3	1	6	0

2. Beispiel zum Floyd-Algorithmus (3)

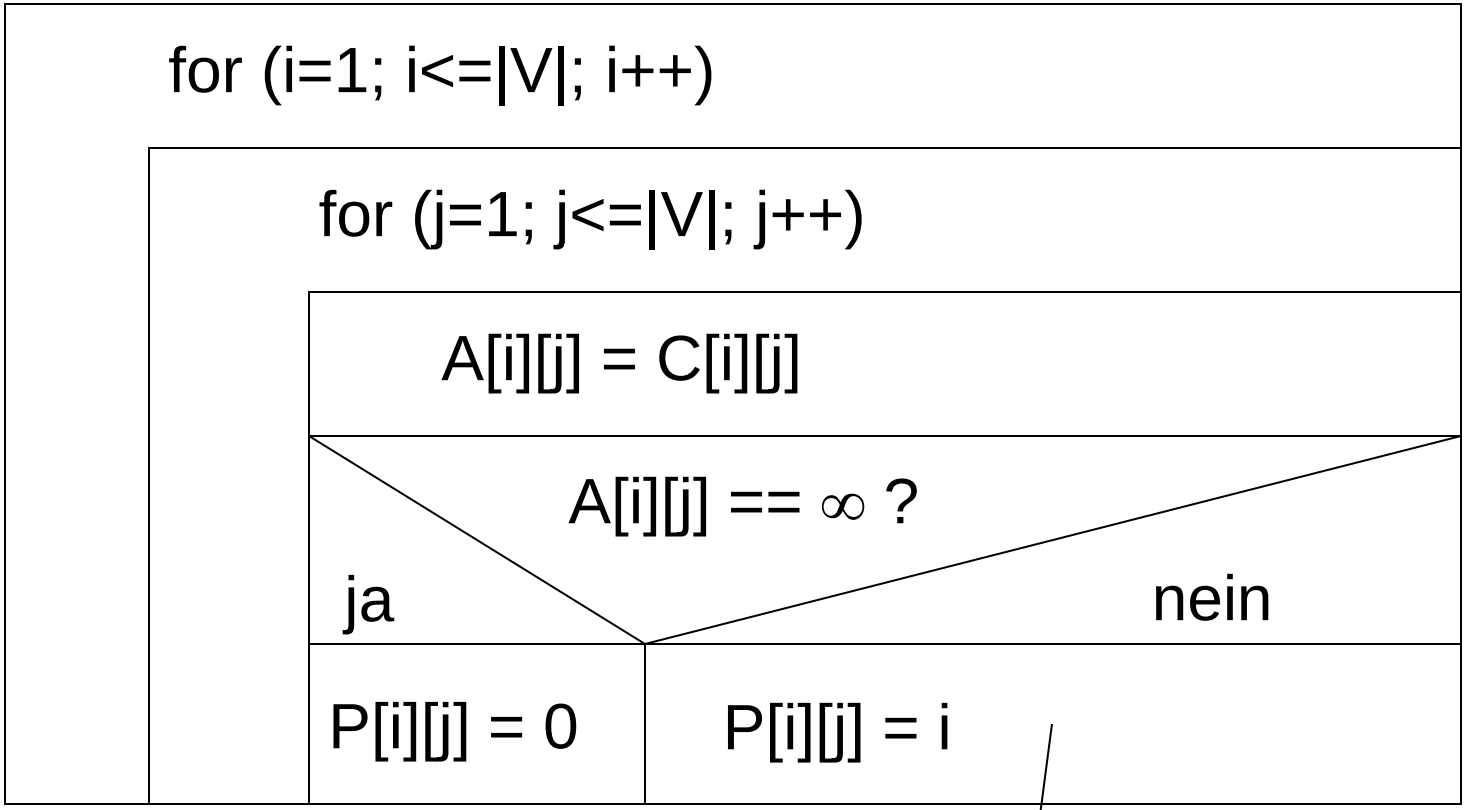
Umweg über 4



von \ nach	1	2	3	4
1	0	2	3	3
2	2	0	5	1
3	3	5	0	6
4	3	1	6	0

Floyd-Algorithmus (1)

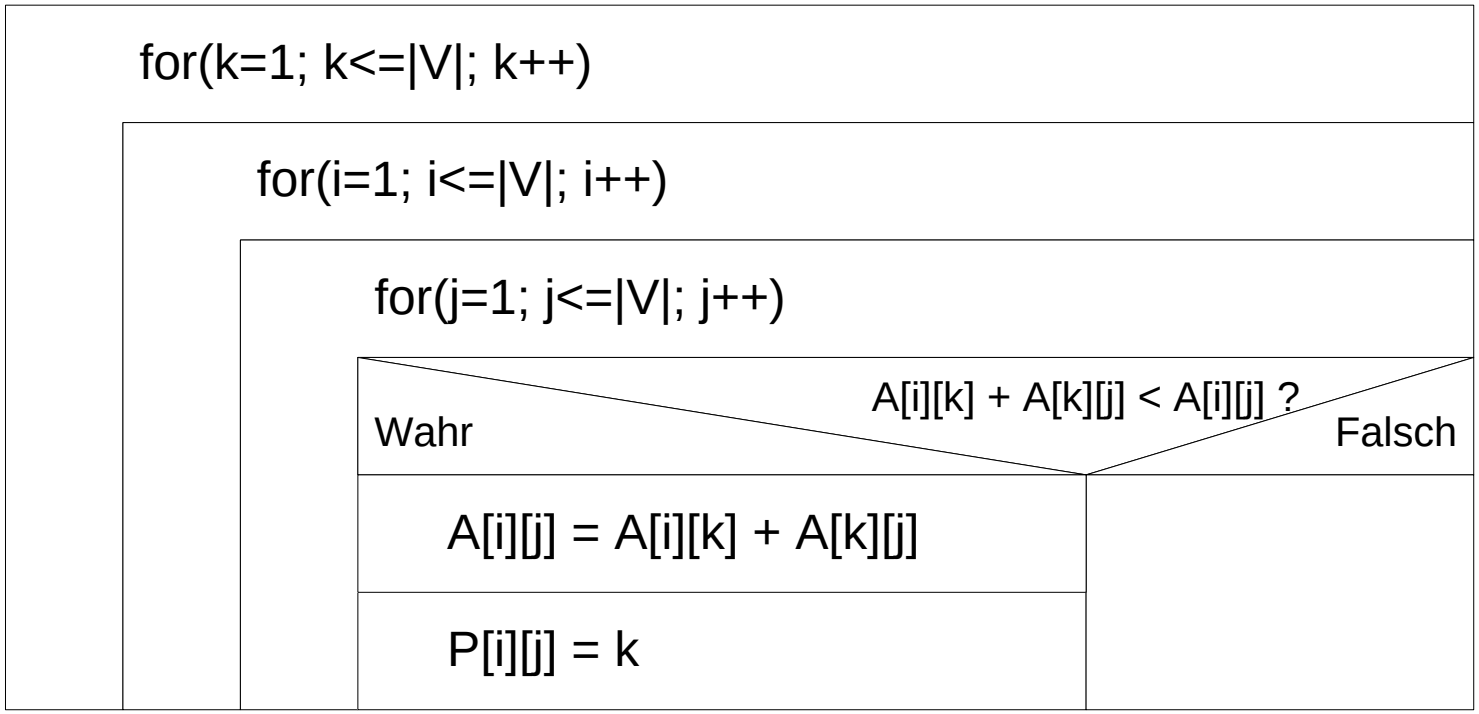
Floyd-Algorithmus (Initialisierung):



enthält zum Schluss einen Vorgänger auf günstigstem Weg von v_i nach v_j.

Floyd-Algorithmus (2)

Floyd-Algorithmus:



Zeitkomplexität des Floyd-Algorithmus

$T_{Floyd}(|V|) \in O(|V|^3)$ (asymptotisch nicht schneller als |V| mal Dijkstra)

Bemerkung:

Algorithmus hat sehr einfache Struktur.

$\Rightarrow$ Compiler erzeugt effizienten Code.

Dadurch wird Floyd doch schneller als |V| mal Dijkstra sein.

Existiert ein Weg zwischen zwei Knoten?

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

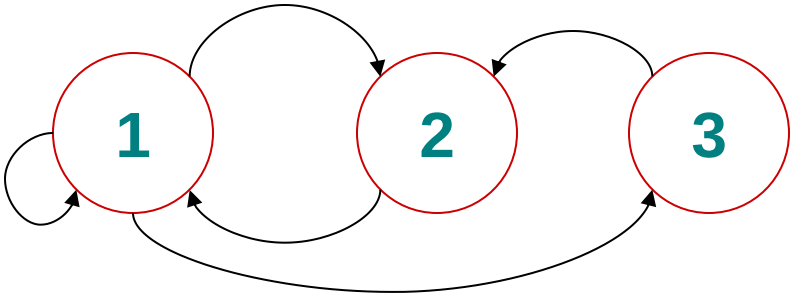
Aufgabenstellung: Prüfe für alle geordneten Paare (v_i,v_j), ob ein Weg (beliebiger Länge) von v_i nach v_j existiert.

Ziel: Berechne „Adjazenzmatrix A zu transitivem Abschluss“ von G:
$A[i][j]$ = true $\Leftrightarrow$ es existiert ein nicht-trivialer Weg (Länge>0) von v_i nach v_j

Lösung: Modifikation von Floyd–Algorithmus
$\Rightarrow$ Warshall-Algorithmus

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

Beispiel zum Warshall-Algorithmus



Anfang

Floyd:

0	8	5
3	0	∞
∞	2	0

Warshall:

$A_{Original}$ =

T	T	T
T	T	F
F	T	T

Ende

Floyd:

0	7	5
3	0	8
5	2	0

Warshall:

$A_{Abschluss}$ =

T	T	T
T	T	T
T	T	T

Warshall-Algorithmus

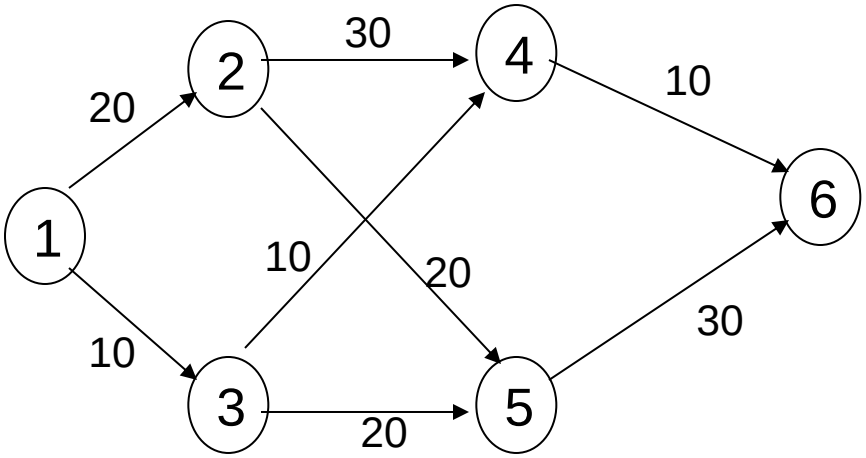
Warshall-Algorithmus:

```
for(i=1; i<=|V|; i++)
    for(j=1; j<=|V|; j++)
        A[i][j] = true, falls (i,j) Element von E;
                  sonst false
for(k=1; k<=|V|; k++)
    for(i=1; i<=|V|; i++)
        for(j=1;j<=|V|;j++)
            A[i][j] = A[i][j] || (A[i][k]  && A[k][j])
```

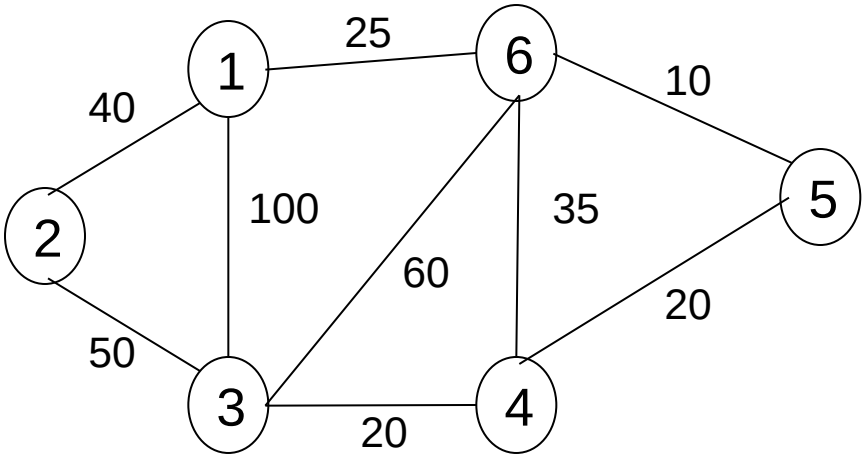
Zeitkomplexität des Warshall-Algorithmus

$$T_{warshall}(|V|) \in O(|V|^3)$$

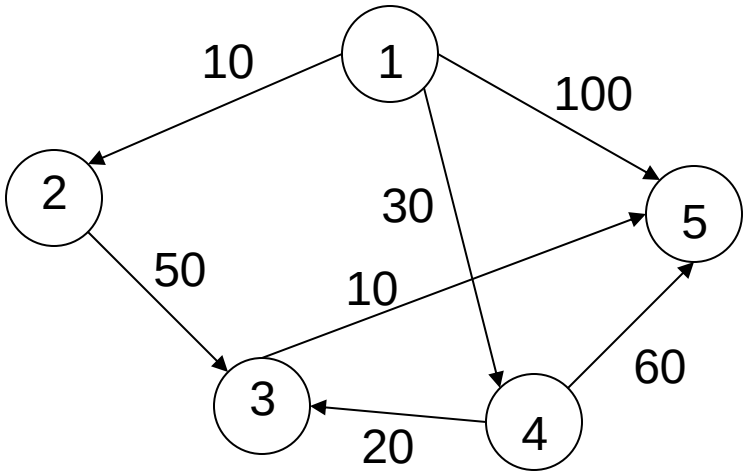
Beispielgraphen (1)



Beispielgraphen (2)



Beispielgraphen (3)



Ausblick zu Graph-Algorithmen

- Teile dieser Algorithmen finden sich als 'Skelett' in vielen interessanten Lösungen zu Problemen aus dem wirklichen Leben:
  - Netzplantechnik
  - Compiler (Vektorisierung)
  - Problem des Handlungsreisenden
  - Produktionssteuerung, etc.

- Es gibt eine große Anzahl weiterer Probleme und Algorithmen
  - Zusammenhangskomponenten (Erreichbarkeit)
  - Flüsse auf Netzwerken (maximale, minimale)
  - Transportprobleme
    - Euler-Kreis: Kreis, in dem jede Kante einmal durchlaufen wird (einfaches Problem)
    - Hamilton-Kreis: Kreis, in dem jeder Knoten einmal durchlaufen wird (sehr schwieriges Problem)
  - Färbungsprobleme

- Einführung
- Naives Verfahren
- Knuth/Morris/Pratt
- Boyer/Moore
- Mustererkennung mit endlichen Automaten

3.4.1 Einführung in Textsuche

Problem: Suche erstes Vorkommen von

Muster-Zeichenfolge `pattern[0...m-1]` in

Text-Zeichenfolge `text[0...n-1]`

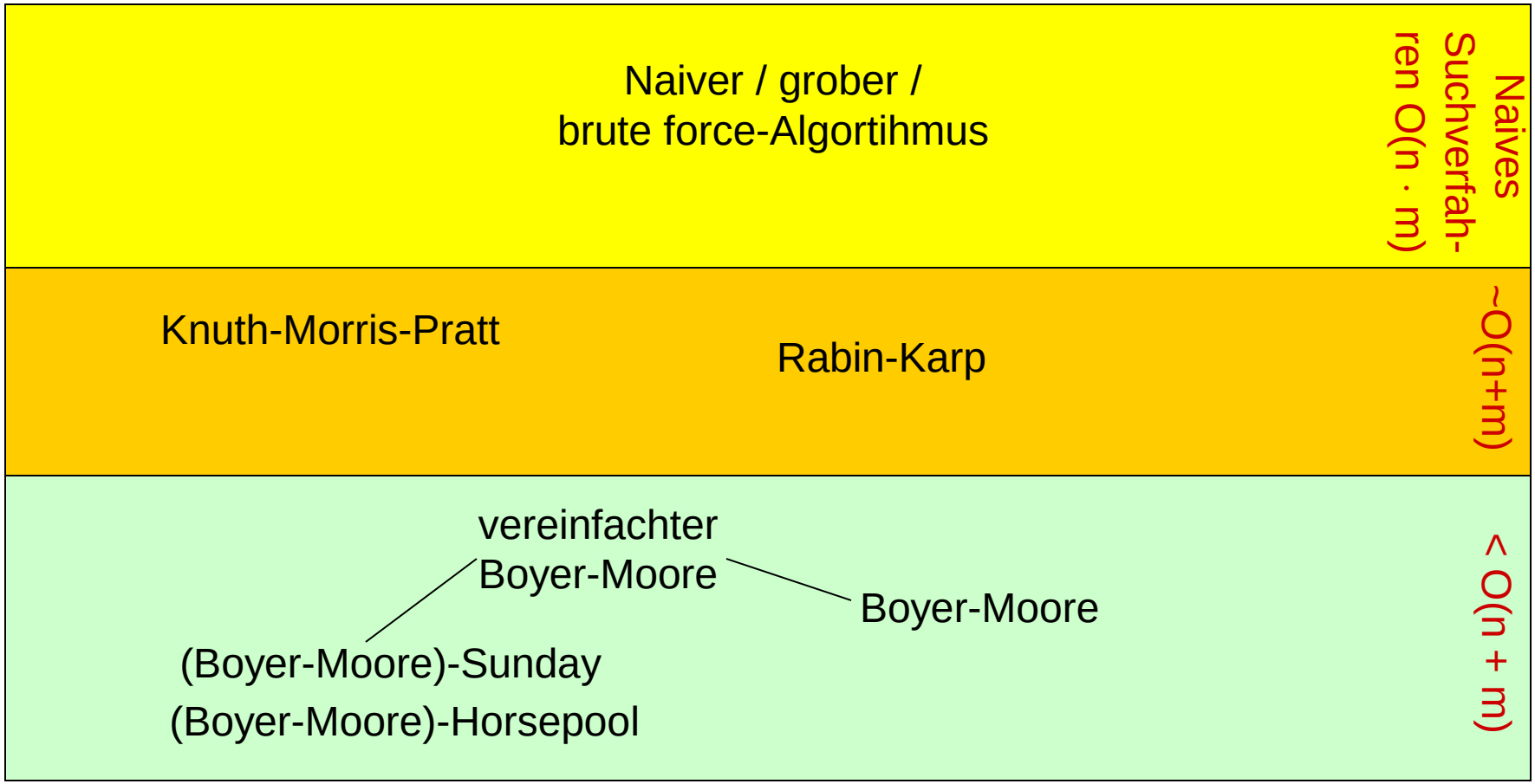
Anwendungen:

- Finden von Mustern in Text-Dateien, z.B.
  Unix-Kommando **`grep`**; Suchfunktion im Editor; Web-Suchmaschine
- Algorithmen funktionieren auch bei anderen „Alphabeten“,
  z.B. DNS-Sequenzen oder Bitfolgen

Verwandtes Problem: Muster ist regulärer Ausdruck

⇒ „Pattern Matching“ („Mustererkennung“) mit endlichen Automaten

Übersicht über die Textsuchverfahren



Einteilung

- Die Textsuchverfahren „**Naiv**“, „**Knuth-Morris-Pratt**“, „**Boyer-Moore-Sunday**“ sind verwandt.
  - **Für kleine Texte ist „Naiv“ am schnellsten.**
  - **Für große Texte ist „Boyer-Moore-Sunday“ am schnellsten.**
  - **Knuth-Morris-Pratt hat vor allem historische Bedeutung.**

- „**Rabin-Karp**“ (hier nicht erklärt) benutzt ein anderes Prinzip.

- Zunächst wird die **grundsätzliche Problemstellung** bei „Naiv“, „Knuth-Morris-Pratt“ und „Boyer-Moore-Sunday“ erläutert.

Algorithmusskizze Naiv, KMP, BMS

Zu Beginn setzt man das Muster an den Beginn des Textes

text `U N G L E I C H U N G S T E I L...`

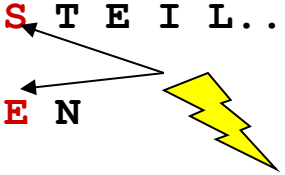
pattern `U N G L E I C H U N G E N`

Dann wird der Text Zeichen für Zeichen durchgegangen, bis entweder

- eine komplette Übereinstimmung festgestellt wird
- oder ein Zeichen nicht übereinstimmt.

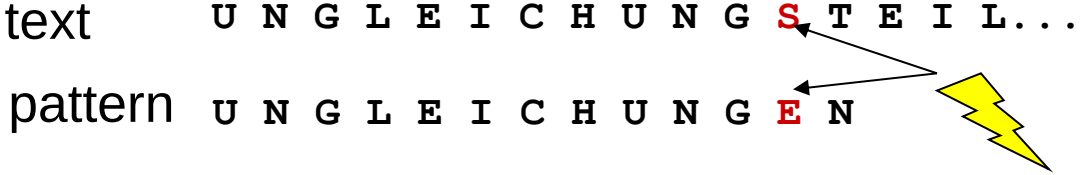
text `U N G L E I C H U N G S T E I L...`

pattern `U N G L E I C H U N G E N`



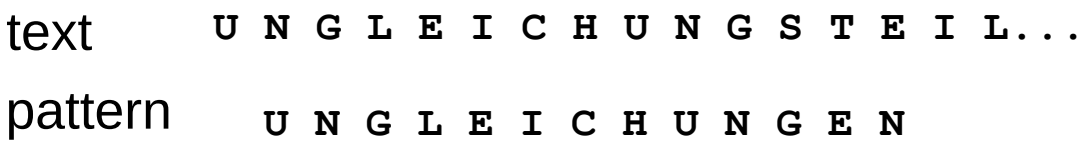
Wenn ein Zeichen nicht übereinstimmt, muss das Muster neu angesetzt werden. Aber wohin?
Das kommt auf das Verfahren an.

Naives Verfahren



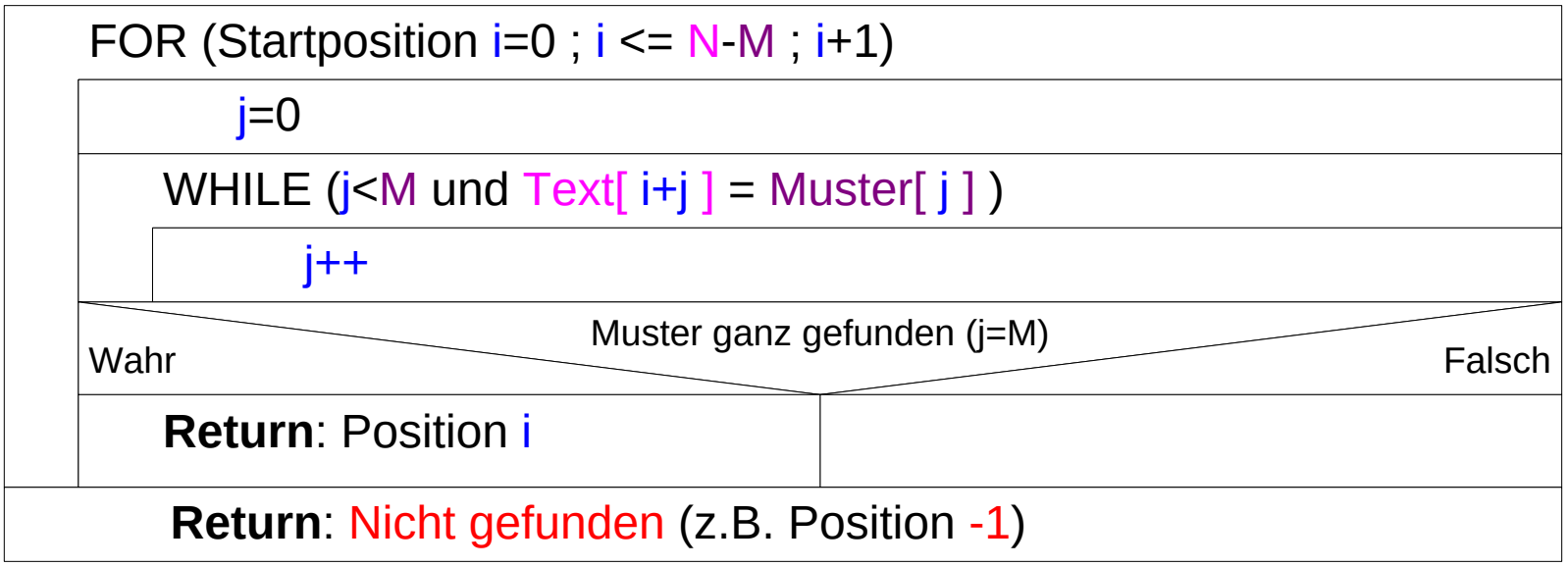
Naives Verfahren:

Das Muster wird einen Buchstaben weiter gesetzt.



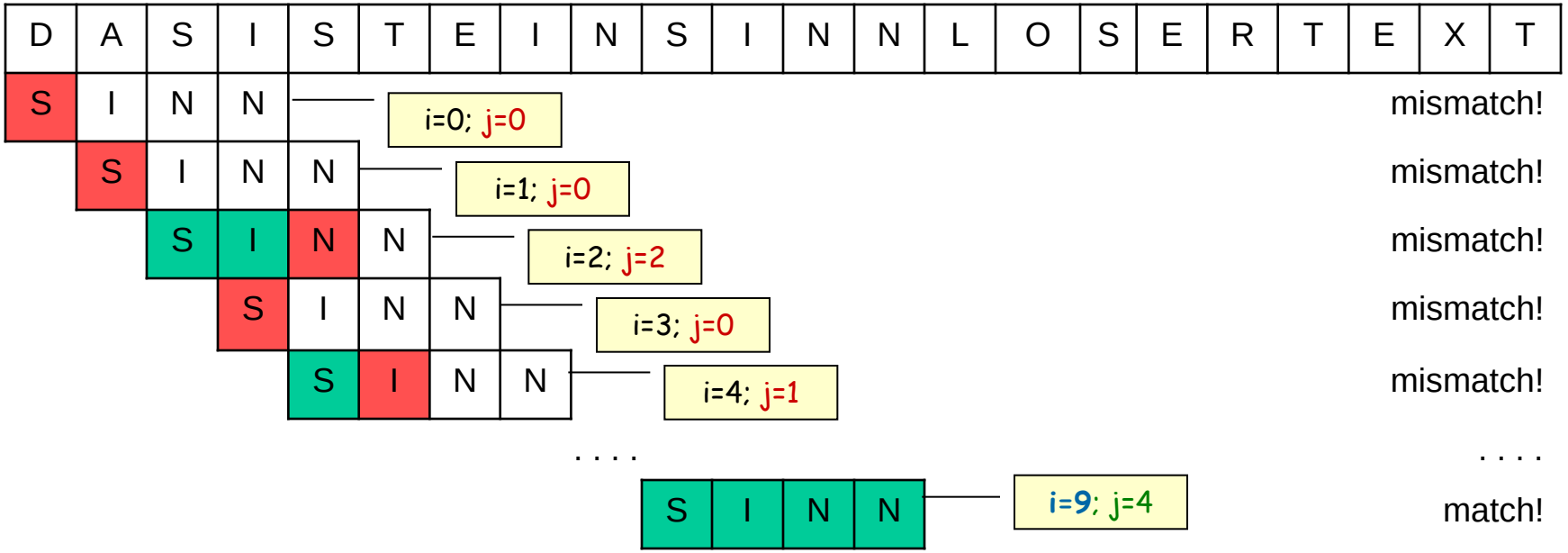
Naives Verfahren: Struktogramm

Ab jeder Position `i` ∈ [0;n-m] prüfen, ob `text[i...i+(m-1)]=pattern`



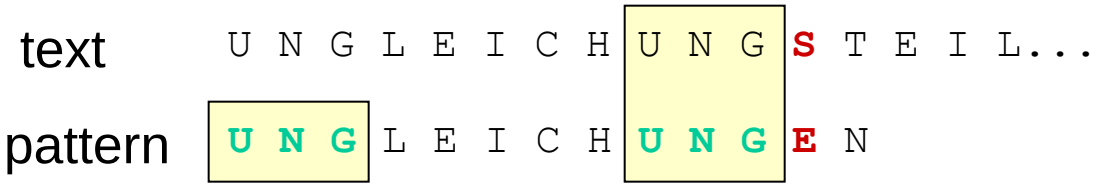
- Worst Case: *m(n-m)* Vergleiche; für *n*>>*m* (Normalfall): *O(n·m)*
- Best Case (realistisch): *O(n)* bei Nicht-Finden; *O(m)* bei Finden;
- Zusätzlicher Platzbedarf: *O(1)*

Mustersuche mit naivem Algorithmus (Bsp.)

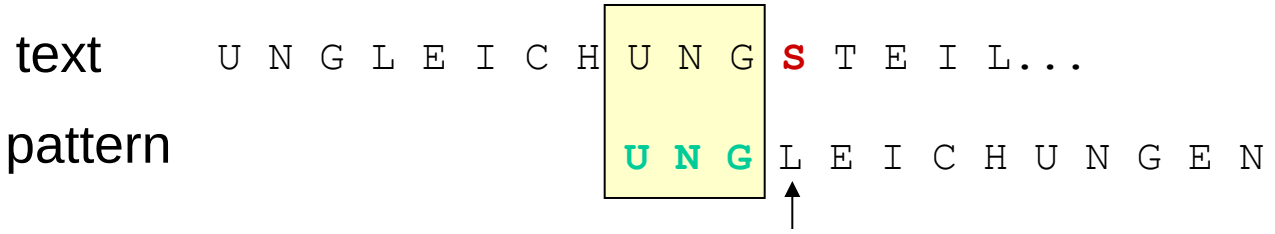


Ein „mismatch“ liegt bei Nichtübereinstimmung vor.
„match“ bedeutet „komplette Übereinstimmung“.

Jetzt muss das Muster neu angesetzt werden. Aber wo?
Regel: Wenn die letzten überprüften Buchstaben gleich dem Anfang des Patterns sind, verschiebt man das Muster entsprechend.



und macht beim anschließenden Zeichen weiter.



Algorithmusskizze KMP (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.

text `U N G L E I C H U N G S T E I L...`

pattern `U N G L E I C H E R`

ergibt

text `U N G L E I C H U N G S T E I L...`

pattern `U N G L E I C H E R`

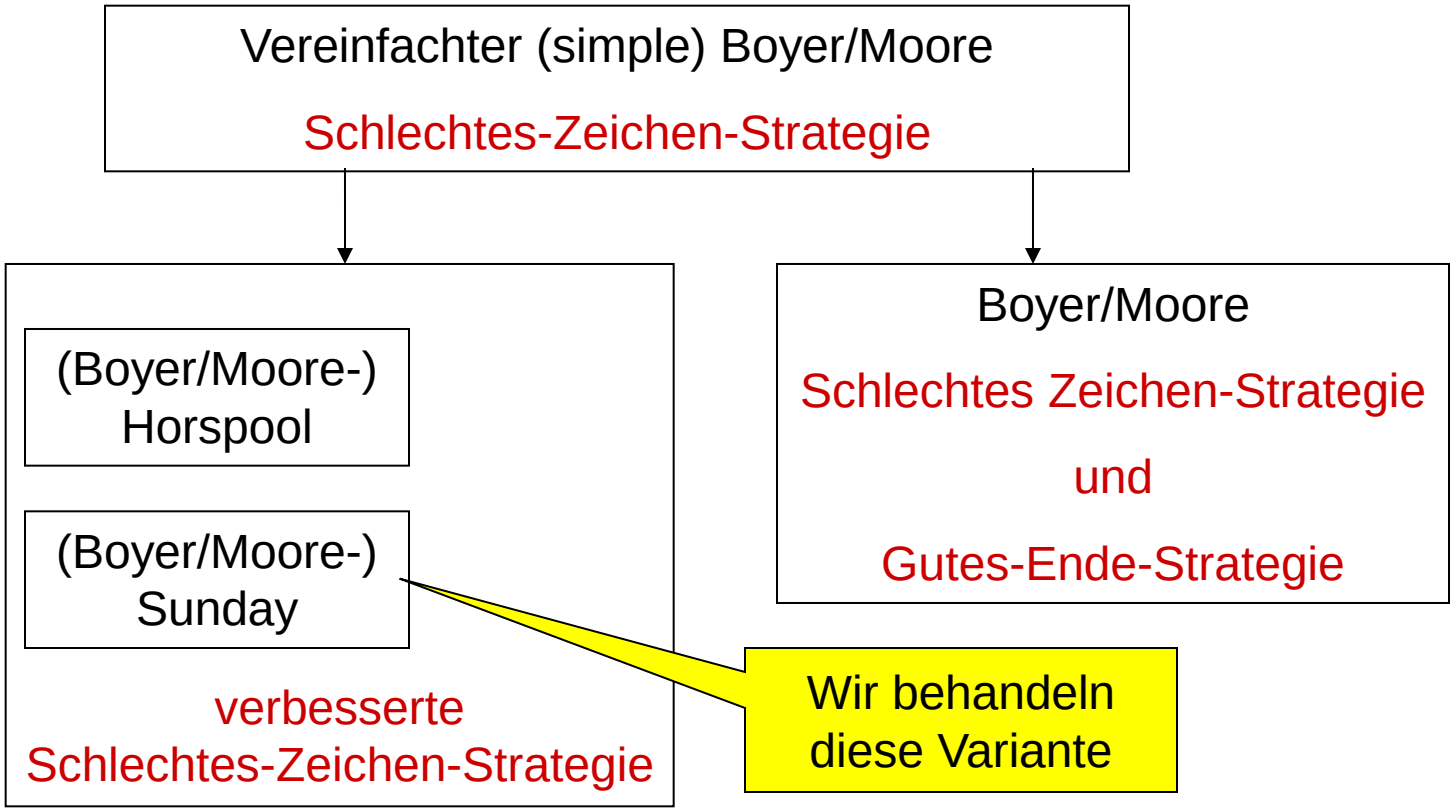
Wichtig ist zu prüfen, welcher von beiden Fällen eintritt.

Komplexität von KMP-Textsuche

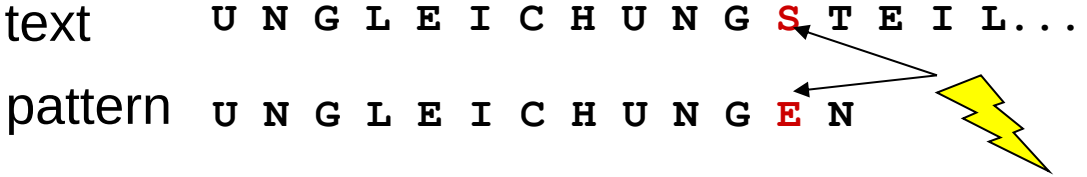
- Der Algorithmus wird in der Vorlesung nicht genauer erklärt.
- Es muss eine next-Tabelle initialisiert werden: Aufwand *O(m)*
- Dann beginnt die Such-Phase: in jedem Schritt
  - entweder im Text um 1 Zeichen weitergehen (`++i`),
  - oder Muster um mindestens 1 Zeichen weiter rechts „anlegen“ (`j=next[j]`).

  $\Rightarrow$ Aufwand *O(n)*
- KMP-Algorithmus insgesamt: *O(n + m)*
- Zusätzlicher Platzbedarf: *O(m)*

3.4.3 Boyer/Moore



Boyer-Moore-Sunday



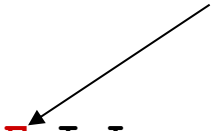
Wenn ein Zeichen nicht übereinstimmt, muss das Pattern neu angesetzt werden. Aber wohin?

Boyer-Moore-Sunday:
Man betrachtet den Buchstaben, der hinter dem Muster liegt:

Boyer-Moore-Sunday

Boyer-Moore-Sunday:

Man betrachtet den Buchstaben, der hinter dem Muster liegt:



text `U N G L E I C H U N G S T E I L...`

pattern `U N G L E I C H U N G E N`

Das Muster kann so weit nach vorne geschoben werden, bis ein Buchstabe des Musters mit diesem Buchstaben übereinstimmt.

text `U N G L E I C H U N G S T E I L...`

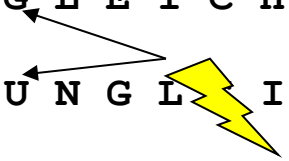
pattern `U N G L E I C H U N G E N`

Boyer-Moore-Sunday

Anschließend wird Text und Muster wieder verglichen.

text U N G L E I C H U N G S T E I L...

pattern U N G L I C H U N G E N



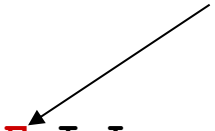
Bei Ungleichheit wird wieder der erste Buchstabe hinter dem Muster betrachtet und das Muster entsprechend vorgeschoben.

text U N G L E I C H U N G S T E I L...

pattern U N G L E I C H U N G E N

Fragen zu BMS

Der auf das Muster folgende Buchstabe kommt im Muster mehrmals vor. Wo wird das Muster angelegt?



text `U N G L E I C H U N G S T E I L...`

pattern `U N G L E I C H U N G E N`

Antwort: An den Buchstaben, der im Muster am weitesten hinten liegt.
Das entspricht dem kleinsten der möglichen Sprünge.

text `U N G L E I C H U N G S T E I L...`

pattern `U N G L E I C H U N G E N`

Fragen zu BMS (2)

Der auf das Muster folgende Buchstabe kommt im Muster **nicht** vor.
Wo wird das Muster angelegt?

text `U N G L E I C H U N G S - T E I L...`

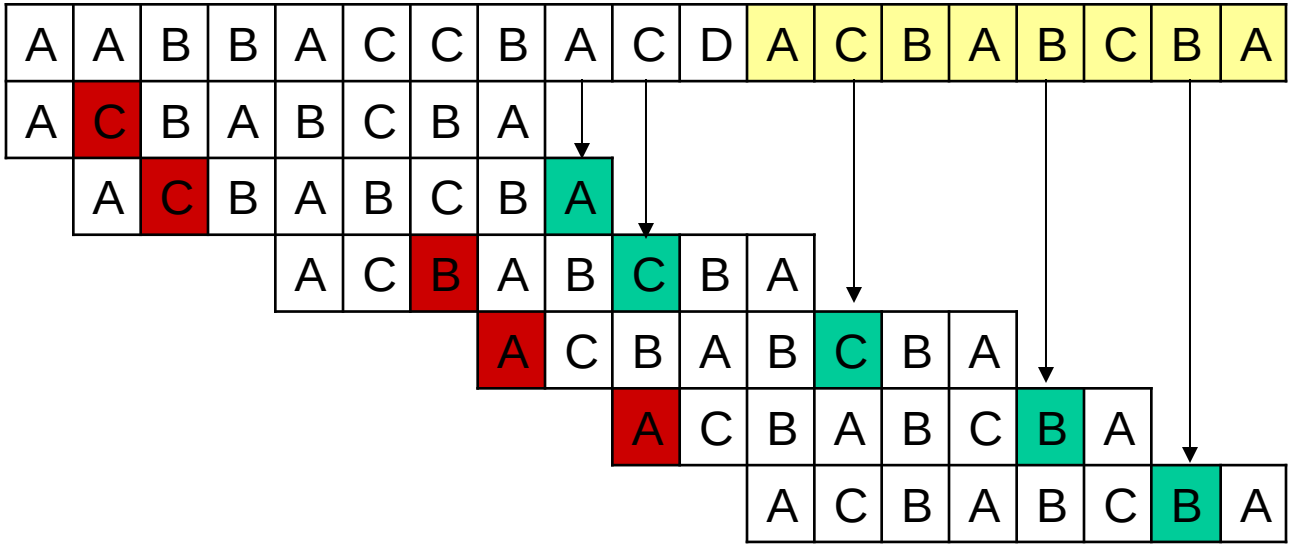
pattern `U N G L E I C H U N G E N`

Antwort: Das Muster wird über den Buchstaben hinweggeschoben.
Alle Positionen vorher sind zwecklos, da der Buchstabe ja im Muster nicht vorkommt.

text `U N G L E I C H U N G S - T E I L...`

pattern `U N G L E I C H U N G E N`

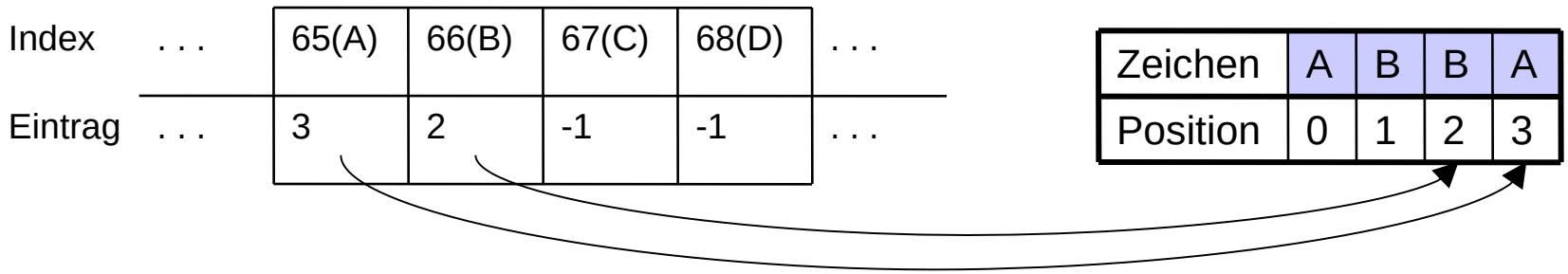
Beispiel 1


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

Die “last-Tabelle”

last-Tabelle
...enthält zu jedem Zeichen des Zeichensatzes die Position des letzten Vorkommens im Muster (oder ‘-1’, falls es nicht vorkommt). Implementierung z.B. als Array indiziert mit (Unicode)-Zeichensatz: ‘A’ auf Index 65, ‘B’ auf 66 usw., ‘a’ auf Index 97, ‘b’ auf 98 usw..

Auszug aus last-Tabelle am Beispielpattern: "ABBA"



Beispiel 2: Last-Tabelle

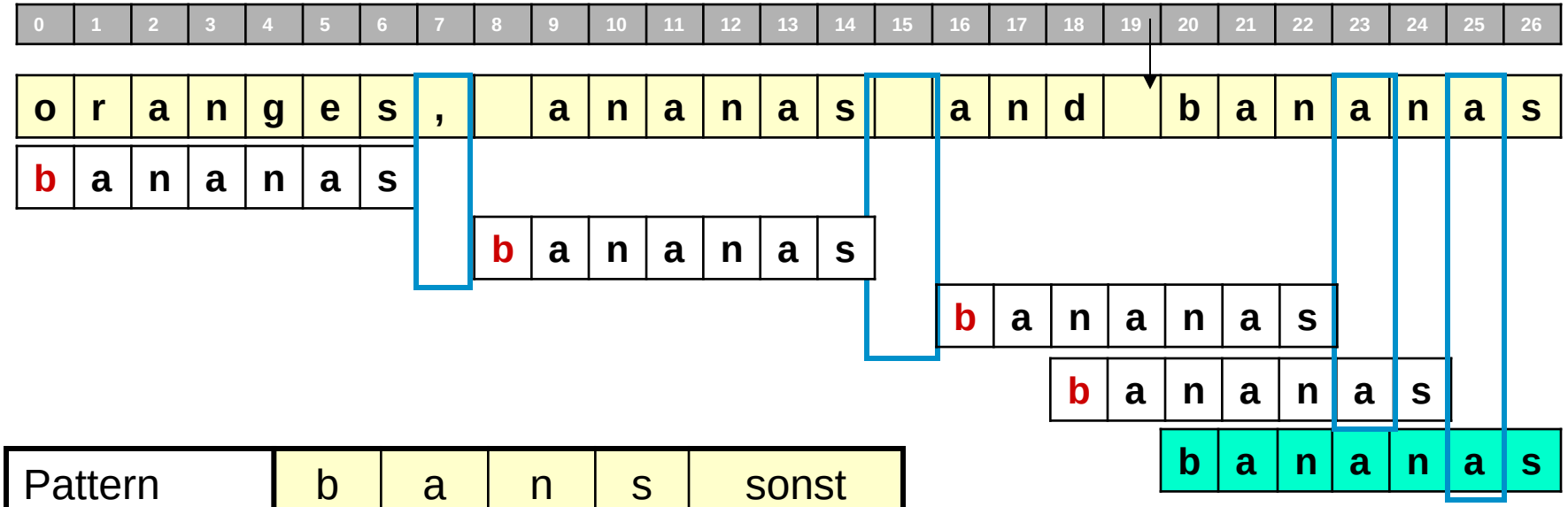
Last-Tabelle des Musters „bananas“:

Pattern	b	a	n	a	n	a	s
Index im Muster	0	1	2	3	4	5	6
Unicode-Index	98	97	110	97	110	97	115
Last-Wert	0	5	4	5	4	5	6

...	a	b	...	n	...	s	...
0-96	97	98	99-109	110	111-114	115	116-127
-1	5	0	-1	4	-1	6	-1

Boyer/Moore/Sunday (Beispiel 2)

Suche nach Pattern ‚bananas‘



Pattern	b	a	n	s	sonst
Last-Wert	0	5	4	6	-1

Alte Position des Patterns: i ... i+m-1
Verschiebedistanz v = m-last[text[i+m]]
Neue Position des Patterns: i+v ... i+v+m-1

Boyer/Moore/Sunday

- Laufzeit dieser Variante (unter Annahme n>>m):
  - *O(n·m)* im schlimmsten Fall
    Bsp. f. ungünstige Text/Pattern-Kombination: Text: a* Muster: aaaaaba
  - Im Normalfall (Text und Muster sind sich sehr unähnlich) geht es aber viel schneller. Wenn Alphabet des Textes groß im Vergleich zu m: nur etwa *O(n/m)* Vergleiche

- Nur sinnvoll, wenn das Alphabet groß ist (z.B. ASCII/Unicode). Für Bitstrings ist dieses Verfahren weniger gut geeignet.

- *String.indexOf(String str)* verwendet naive Textsuche.
  - **Python ebenso**
  - **Ruby verwendet Rabin/Karp (hier nicht behandelt)**

3.4.4 „Mustererkennung“ mit endlichen Automaten

- Bisher gesucht: konkrete Zeichenfolge
- Jetzt: Suche nach allgemeinerem Muster
- Dazu benötigt:
  - Notation, um allgemeine Muster zu beschreiben
    $\Rightarrow$ reguläre Ausdrücke
  - Mechanismus, um solche Muster zu „erkennen“
    $\Rightarrow$ endliche Automaten

Musterbeschreibung

- Die Notation, in der die Muster beschrieben werden, heißt „reguläre Ausdrücke“.
  - **Die Bezeichnung „regulärer Ausdruck“ kann man etwa so interpretieren: „Eine Menge von Ausdrücken, die durch (einfache) Regeln beschrieben wird“.**
- Jeder reguläre Ausdruck beschreibt eine Menge von Zeichenfolgen, nämlich alle, die dem Muster entsprechen.

Begriffe: Alphabet, Wörter, Sprache

Zeichen - z.B. Buchstabe, Ziffer

Alphabet - endliche Menge von Zeichen, z. B. $\Sigma = \{a,b,c\}$

Wort über Alphabet Σ

- endliche Folge von Zeichen aus Σ, z.B. abcb
- Spezialfall: "leeres Wort" ε

Σ^* - Menge aller Wörter über Σ,
z. B. $\Sigma = \{a,b\} \Rightarrow \Sigma^* = \{ \varepsilon, a,b,aa,ab,ba,bb,aaa,...\}$

Sprache *L* über Alphabet Σ

- Teilmenge: $L \subseteq \Sigma^*$ (auch $\{\varepsilon\}$ ist Sprache über Σ).

Sonderfall:
→*reguläre* Sprachen

Definition: Reguläre Ausdrücke

Regulärer Ausdruck (in der theoretischen Informatik):

- **Ein regulärer Ausdruck ist eine „Formel“, die eine Sprache beschreibt, d.h. eine Teilmenge alle möglichen Worte definiert.**

- Der Begriff Regulärer Ausdruck hat (vor allem in der Unix-Welt) noch eine andere, verwandte Bedeutung.
  - **wird am Ende des Kapitels erklärt.**

Regeln für reguläre Ausdrücke

- Die Regeln für reguläre Ausdrücke setzen sich aus 3 grundlegenden Operationen zusammen.
  - **Verkettung**
  - **Oder**
  - **Hüllenbildung**

Verkettung

- Die erste Operation heißt „Verkettung“ (Concatenation)
- Zwei oder mehr Buchstaben werden durch diese Operation aneinandergehängt, z.B. **AB**. Der Operator wird nicht mitgeschrieben.
- Das heißt schlicht und einfach, dass die Buchstaben im Text direkt hintereinander stehen müssen.

- Würde man nur diese eine Operation zulassen, wäre man bei der „einfachen“ Textsuche.

Oder

- Die zweite Operation heißt „Oder“ (Or)
- Sie erlaubt die Angabe von Alternativen im Muster
- Schreibweise: (A|B) für „entweder A oder B“.
- Beispiele:
  - **(A|B)(A|B) bedeutet: AA, AB, BA oder BB.**
  - **(A|C)((B|C)D) bedeutet: ABD, CBD, ACD oder CCD.**
  - **C(AC|B)D bedeutet: CACD oder CBD.**

Hüllenbildung

- Die dritte Operation heißt „Hüllenbildung“ (Closure).
- Sie erlaubt es, Teile des Musters beliebig oft zu wiederholen.
- Erlaubt ist auch 0 mal Wiederholung
- Schreibweise: Hinter den zu wiederholenden Buchstaben wird ein Stern (*) gesetzt.
- Sind mehrere Buchstaben zu wiederholen, müssen sie in Klammern gesetzt werden.
- Beispiele:
  - **A* bedeutet: ε, A, AA, AAA, AAAA, AAAAA, ...**
    - **ε bedeutet „Leerstring“**
  - **(ABC)* bedeutet: ε, ABC, ABCABC, ABCABCABC, ...**
  - **DA*B bedeutet: DB, DAB, DAAB, DAAAB, ...**

Operatorhierarchie

- Klammern binden am stärksten.
- Es folgt die Hüllenbildung (*).
- Dann folgt die Verkettung.
- Am schwächsten ist der oder-Operator (|).

Ziele

Zuerst werden wir mit regulären Ausdrücken ermitteln:

- ob ein Textstring einem regulären Ausdruck entspricht.

Später werden wir:

- einen regulären Ausdruck in einem längeren Text suchen.

Beispiel (1)

- Beispiele:

- a|a(a|b)*a bedeutet:
  - **a | a(a|b)*a** **entweder ein einfaches a oder**
  - **a | a(a|b)*a** **ein führendes a, gefolgt von**
  - **a | a(a|b)*a** **einer beliebigen Kombination von a und b (z.B. abbab)**
  - **a | a(a|b)*a** **und einem abschließenden a**
- Zusammengefasst: Alle Zeichenketten aus a und b, die am Anfang und am Ende ein a enthalten.

Beispiel (2)

- (1+01)*(0+1) bedeutet:
  - **(1+01)*(0+1)** **Eine beliebige Abfolge aus den Elementen „1“ und „01“**
  - **(1+01)*(0+1)** **gefolgt von einer abschließenden 0 oder 1.**

- Zusammengefasst: Alle Kombinationen aus 1 und 0, in denen nicht mehrere 0 aufeinander folgen.

Endliche Automaten

Endlicher Automat (finiter Automat): abstraktes Maschinenmodell

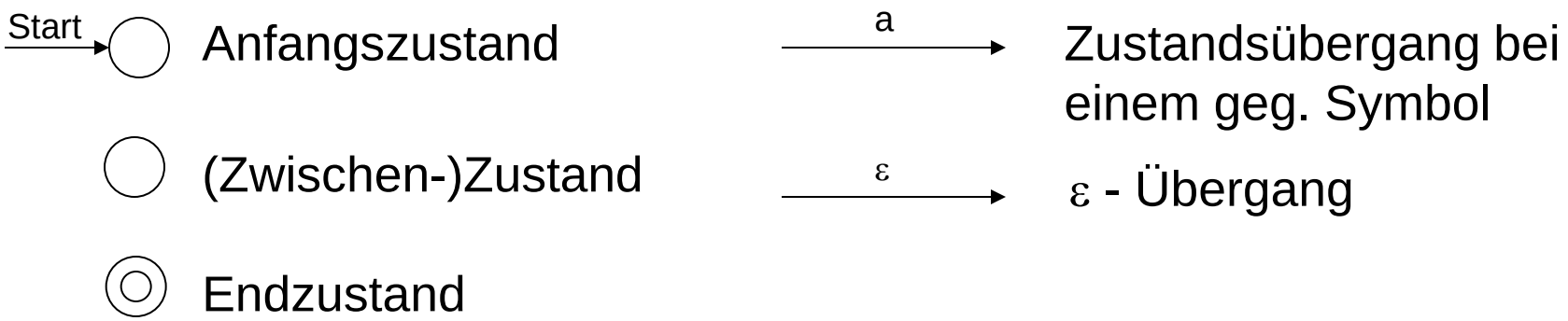
- Aufgabe: Entscheiden, ob Wort zur Sprache gehört, die durch regulären Ausdruck beschrieben ist (Akzeptoren).
- Anfangs: Maschine ist in „Anfangszustand“
- In jedem Schritt wird ein Eingabesymbol σ „gelesen“. Abhängig von σ geht die Maschine von einem Zustand in einen bestimmten anderen über.
- Wenn nach Lesen des letzten Zeichens ein „Endzustand“ erreicht ist, ist das Muster ist „erkannt“ (Wort gehört zu Sprache).

Graphische Simulation

- Es ist möglich, einen endlichen Automaten graphisch zu simulieren.
- Besonders einfach ist das Konstruktions-Verfahren nach Kleene, das zu einem nichtdeterministischen finiten Automaten (NFA) führt.
- Es gibt immer auch einen entsprechenden deterministischen finiten Automaten (DFA). Dort gibt es keine ε-Übergänge.

Graphische Simulation (2)

- Grundlage ist das Zustands-Übergangs-Diagramm mit folgenden Elementen.

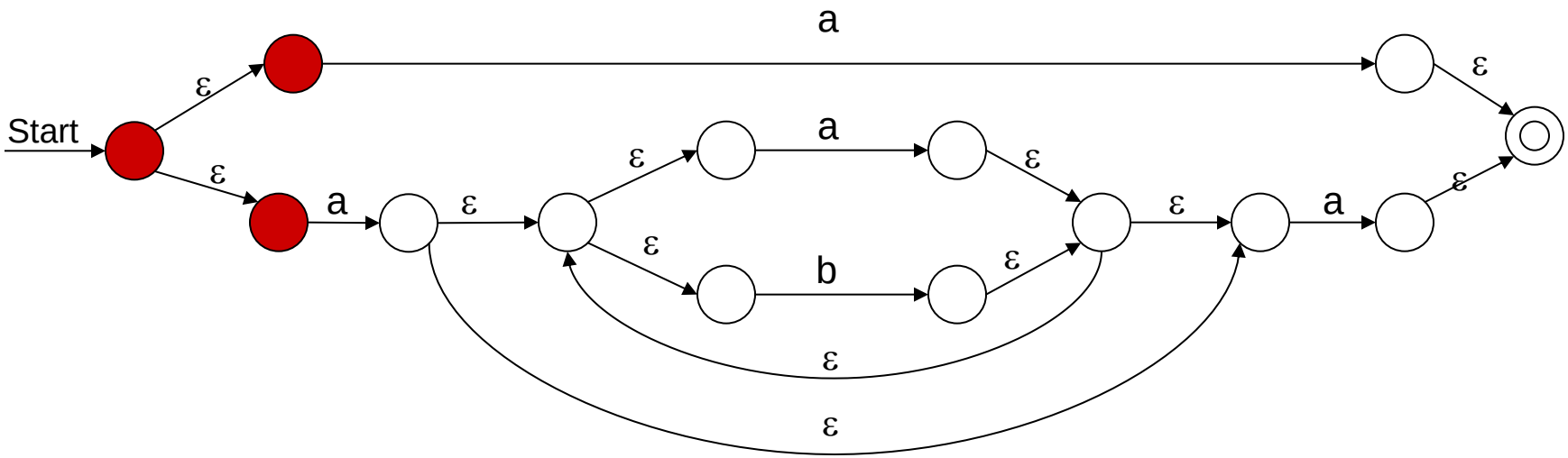


Ablaufregeln der Simulation

1. Initialisierung
   1. **Markiere den Anfangszustand**
   2. **Markiere alle Zustände, die durch ε-Übergänge erreichbar sind.**
2. Für jedes gelesene Eingabesymbol
   1. **Markiere alle Zustände, die durch dieses Eingabesymbol erreichbar sind.**
   2. **Lösche alle anderen Zustände.**
   3. **Markiere alle Zustände, die jetzt durch ε-Übergänge erreichbar sind.**

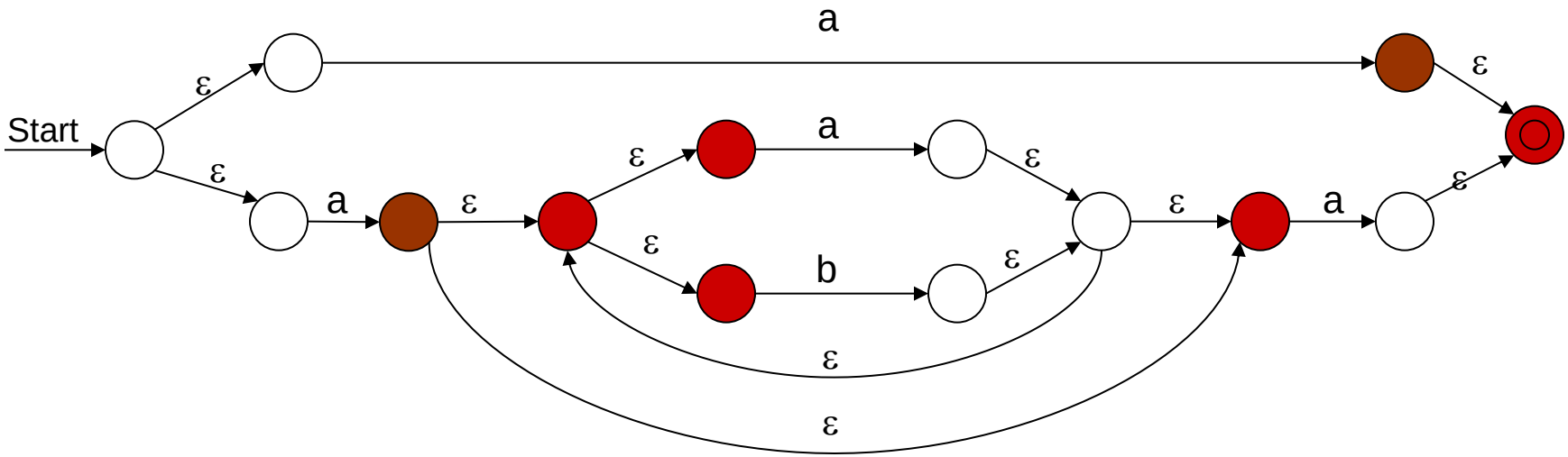
Ablauf der Simulation (1)

- Text „ABBA“
- Automat „A|A(A|B)*A“ (Konstruktion später)
- Anfangszustand



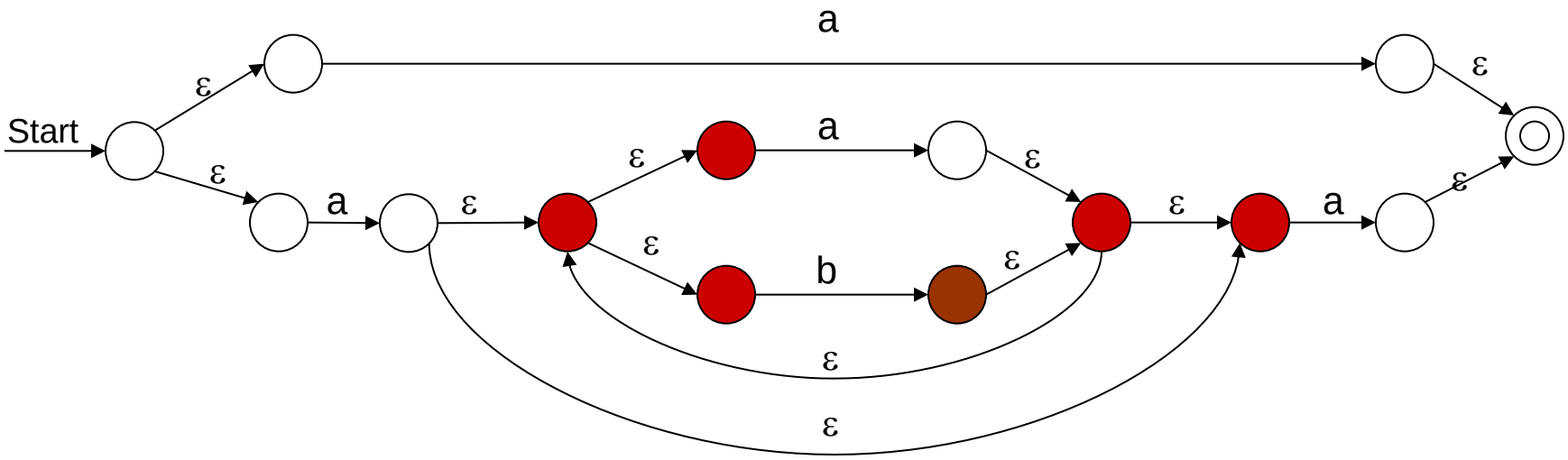
Ablauf der Simulation (2)

- Text „ABBA“
- 1. Buchstabe: A



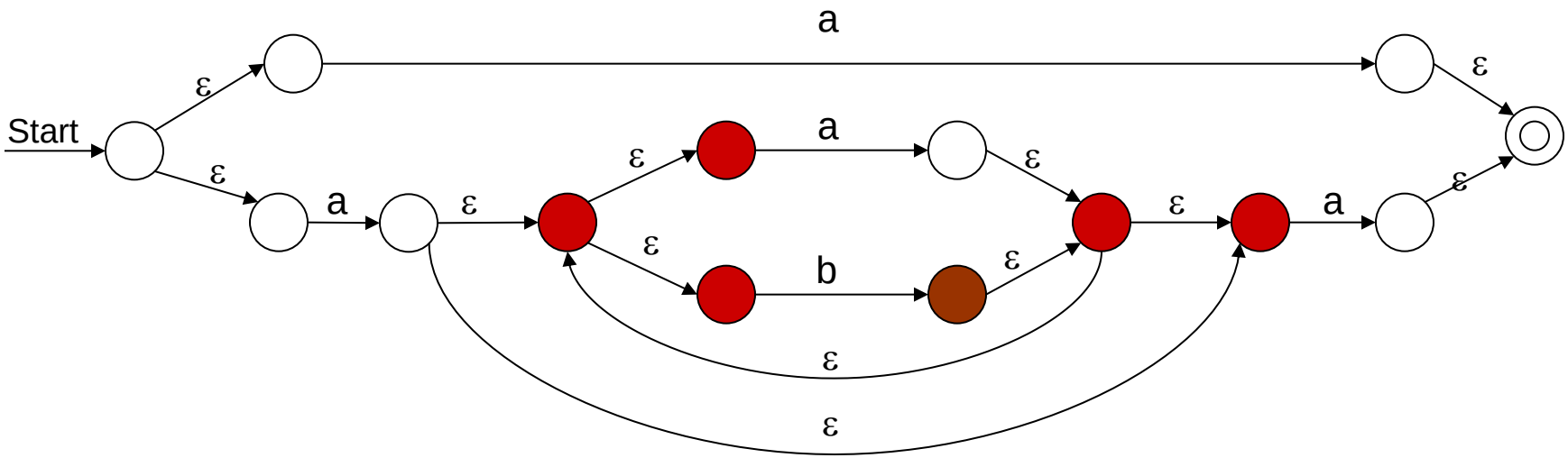
Ablauf der Simulation (3)

- Text „ABBA“
- 2. Buchstabe: B



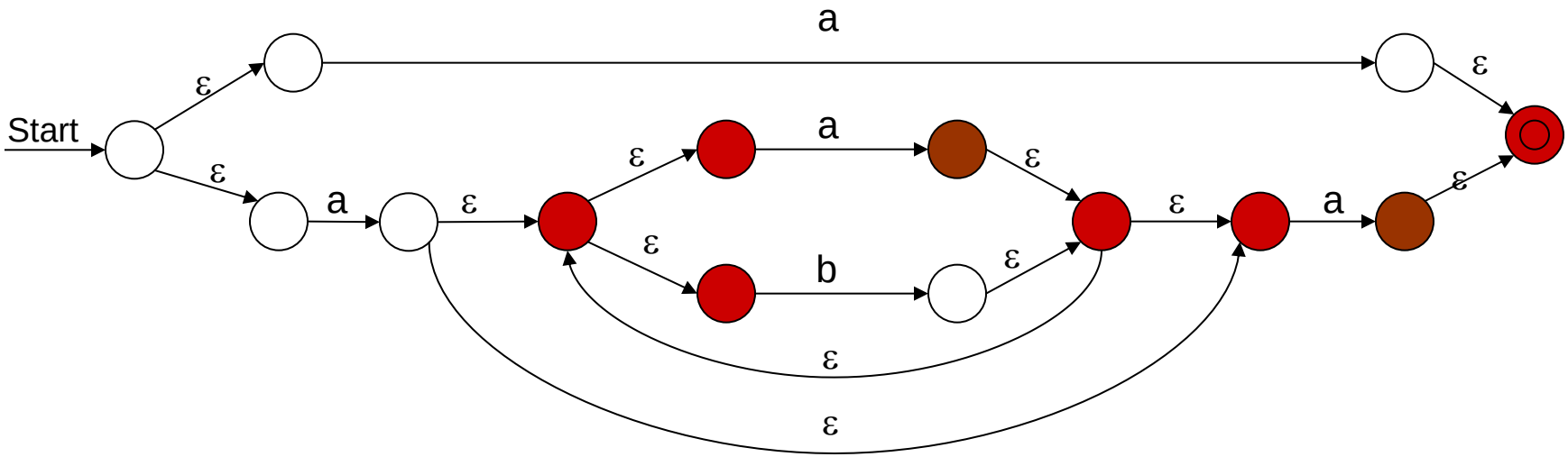
Ablauf der Simulation (4)

- Text „ABBA“
- 3. Buchstabe: B



Ablauf der Simulation (5)

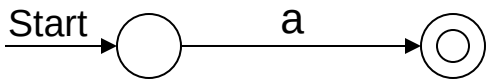
- Text „ABBA“
- 4. Buchstabe: A



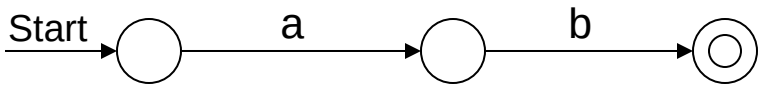
ABBA wird durch den regulären Ausdruck A|A(A|B)*A beschrieben.

Konstruktion des Automaten

- Einzelnes Symbol: Regulärer Ausdruck „a“:

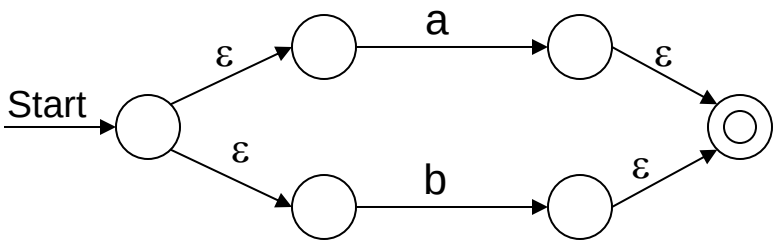


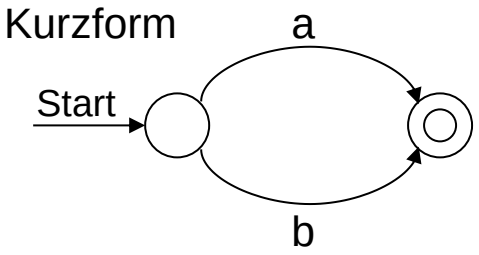
- Verkettung: Regulärer Ausdruck „ab“:



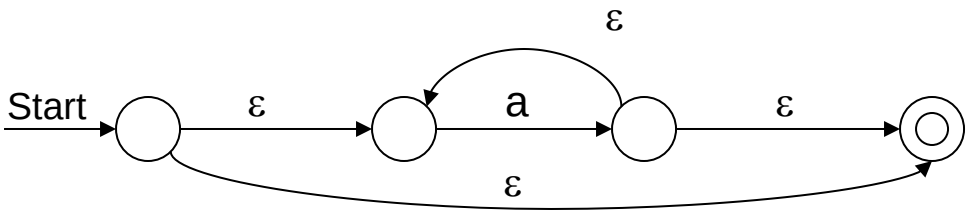
Konstruktion des Automaten (2)

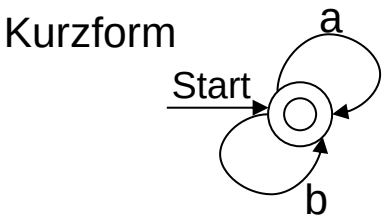
- Oder-Veknüpfung a|b





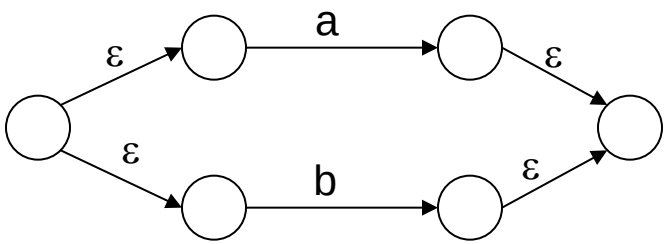
- Hüllenbildung a*





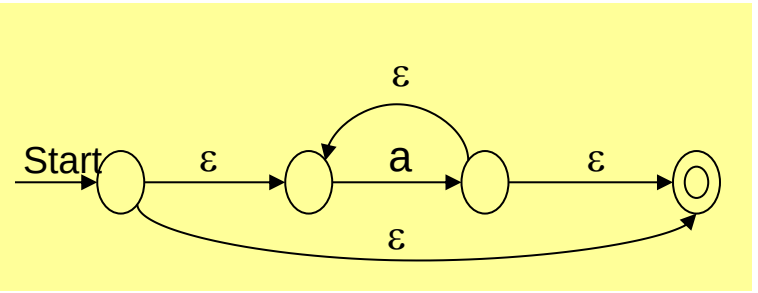
Kombination der Elemente

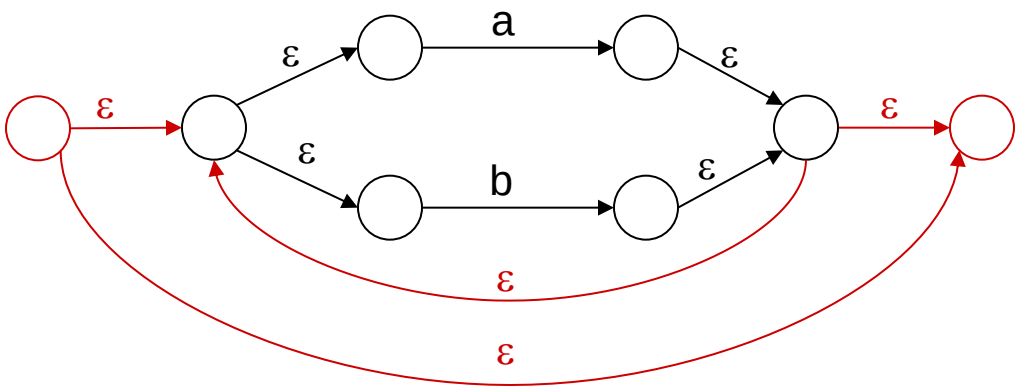
- Aufbau des Automaten für: a|a(a|b)*a
- 1. Schritt: a|b



Kombination der Elemente

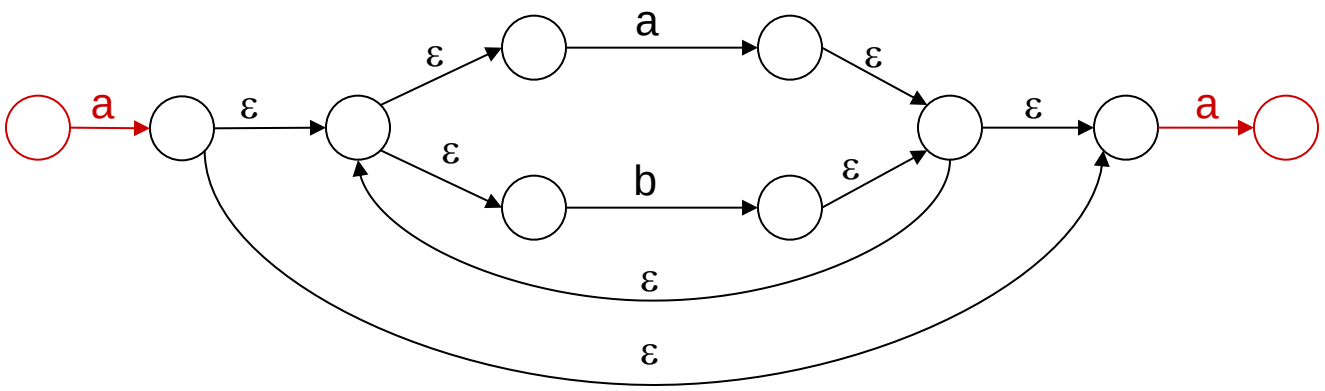
- Aufbau des Automaten für: a|a(a|b)*a
- 2. Schritt: (a|b)*





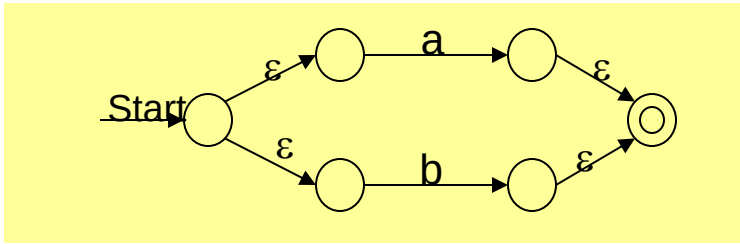
Kombination der Elemente

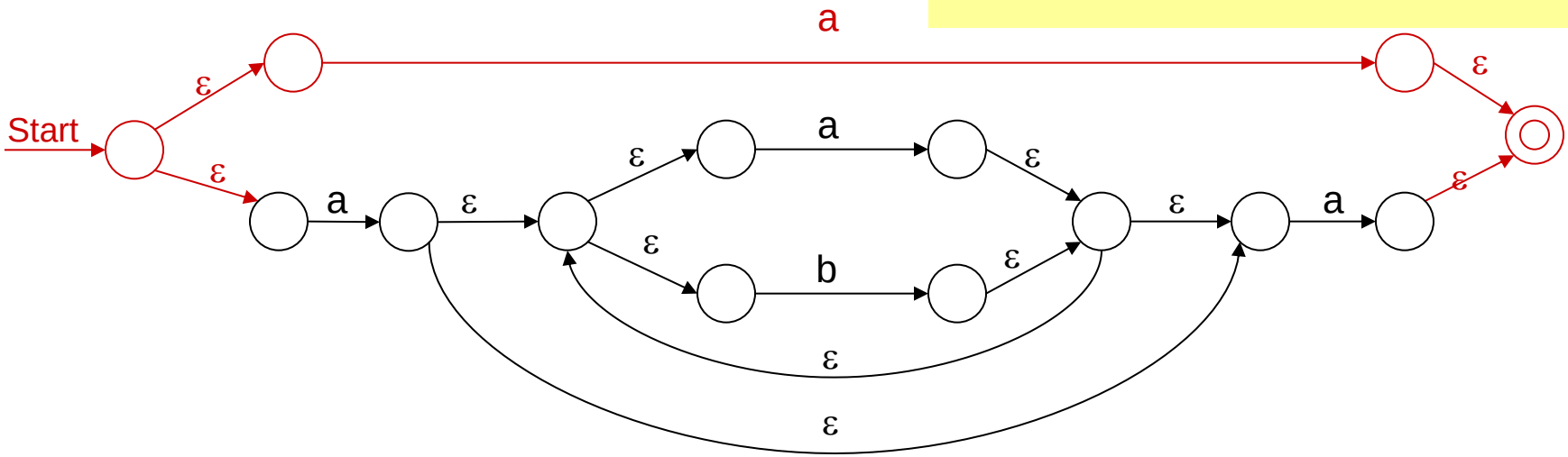
- Aufbau des Automaten für: a|a(a|b)*a
- 3. Schritt: a(a|b)*a



Kombination der Elemente

- Aufbau des Automaten für: a|a(a|b)*a
- 4. Schritt: a|a(a|b)*a

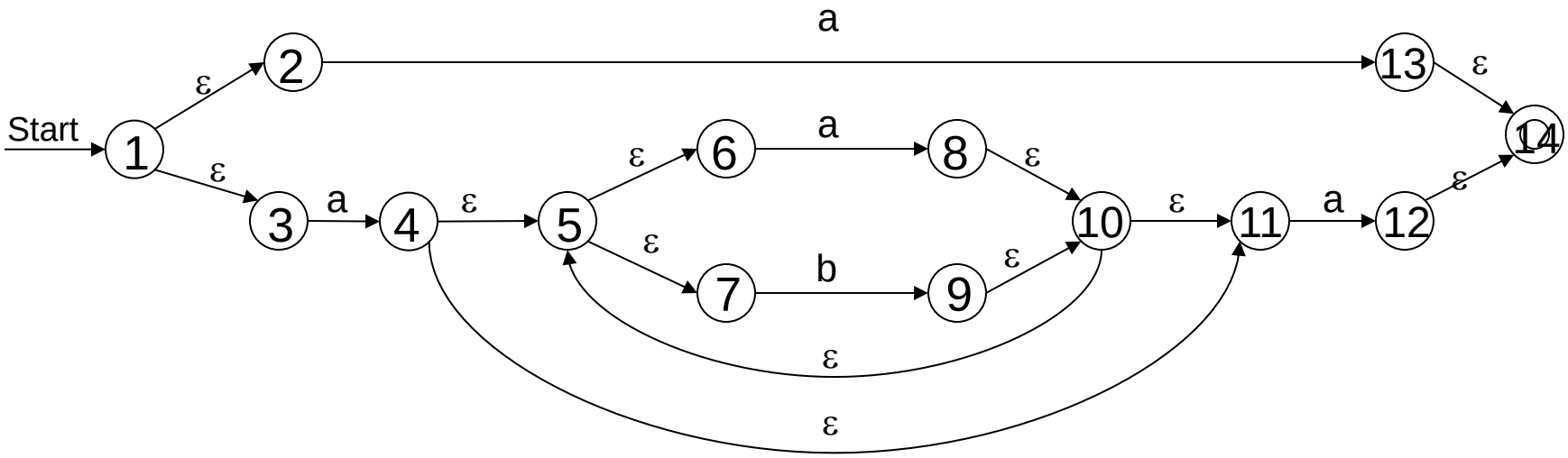




Umsetzung auf den Computer

- Zustands-Übergangs-Diagramm ist ein Graph
- Speicherung in Tabellenform (z.B. Adjazenzliste)
- Neuer Zustand lässt sich aus Tabelle ermitteln.

1	ε2, ε3		8	ε10
2	a13		9	ε10
3	a4		10	ε11, ε5
4	ε5, ε11		11	a12
5	ε6, ε7		12	ε14
6	a8		13	ε14
7	b9		14	



Letzter Zustandsübergang (Wiederholung)

1 (a)

1	ε2, ε3		8	ε10
2	a13		9	ε10
3	a4		10	ε11, ε5
4	ε5, ε11		11	a12
5	ε6, ε7		12	ε14
6	a8		13	ε14
7	b9		14	

2 (ε)

1	ε2, ε3		8	ε10
2	a13		9	ε10
3	a4		10	ε11, ε5
4	ε5, ε11		11	a12
5	ε6, ε7		12	ε14
6	a8		13	ε14
7	b9		14	

3 (ε)

1	ε2, ε3		8	ε10
2	a13		9	ε10
3	a4		10	ε11, ε5
4	ε5, ε11		11	a12
5	ε6, ε7		12	ε14
6	a8		13	ε14
7	b9		14	

4 (ε)

1	ε2, ε3		8	ε10
2	a13		9	ε10
3	a4		10	ε11, ε5
4	ε5, ε11		11	a12
5	ε6, ε7		12	ε14
6	a8		13	ε14
7	b9		14	

Letzter Zustandsübergang (2)

5
(ε)

1	ε2, ε3		8	ε10
2	a13		9	ε10
3	a4		10	ε11, ε5
4	ε5, ε11		11	a12
5	ε6, ε7		12	ε14
6	a8		13	ε14
7	b9		14	

Suchen nach regulären Ausdrücken

- **Beispiel**

- Text: `AABABBAABBCBA`

- Gesucht wird das Pattern: `AB(A|B)*A`

- Regel: **First, longest**

- Ansetzen des Patterns mit nichtdeterministischem endlichen Automaten:
  - **+: Endzustand ist markiert.**
  - **o: Endzustand ist nicht markiert, aber Zwischenzustände sind markiert.**
  - **–: Kein Zustand ist markiert. Hier kann abgebrochen werden.**

Beispiel: Suchen nach reg. Ausdrücken

- Text: `AABABBAABBCBA` Pattern: `AB(A|B)*A`

- Ansatz am ersten Buchstaben → Pattern passt nicht
  `AABABBAABBCBA`
  `o-`
- Ansatz am zweiten Buchstaben
  `AABABBAABBCBA`
  `oo+oo++oo-`

  - **Wenn das „–“ erreicht ist, nimmt man die Position des letzten „+“: AABABBAA → First, longest**
  - **First, shortest (erstes „+“; ABA) kann man auch ermitteln, dies wird aber selten gebraucht.**

- Weitersuchen: Gewöhnlich setzt man hinter dem gefundenen Pattern wieder an.

Reguläre Ausdrücke in Unix/Linux

- In der theoretischen Informatik sind die Regeln für reguläre Ausdrücke:
  - **XY: Verkettung**
  - **X|Y: Oder**
  - **X*: Hüllenbildung**

- In Unix wurden diese Regeln in einigen Werkzeugen (ed, grep) umgesetzt.
  - **Es wurden aber der Bequemlichkeit halber viele weitere Operatoren hinzugefügt.**
- Daraus entstanden die Perl Compatible Regular Expressions (PCRE), die meistens einfach „reguläre Ausdrücke“ genannt werden.

Perl Compatible Regular Expressions (PCRE)

- Standardisierte Regeln zur Erzeugung regulärer Ausdrücke für die Textsuche.
- Sind in vielen Editoren und Programmiersprachen vorhanden.
  - **Z.B. Eclipse: Im Dialogfeld „Find/Replace“ das Auswahlfeld „Regular expressions“ anklicken.**
  - **Z.B. OpenOffice: Im Dialogfeld „Find & Replace“ „More Options“ auswählen und anschließend „Regular expressions“ anklicken.**
  - **Z.B. Java, C#, Python, Ruby, Perl, ...**

PCRE und NFA

- Einige Erweiterungen der PCRE gehen über die Grenzen von regulären Ausdrücken hinaus.
  - **Unterstützung von Rückwärtsreferenzen (sehr praktische Möglichkeit).**
- PCREs werden nicht mit endlichen Automaten untersucht.
  - **Das Suchverfahren heißt zwar NFA, ist aber mathematisch kein NFA.**
  - **Es werden Suchbäume aufgebaut, die mit Backtracking durchsucht werden.**
  - **Wissen über die genaue Funktionsweise ist für Gebrauch wichtig.**

Wichtige Regeln in PCRE

- 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}`

Wichtige Regeln in PCRE

- Zeichenklassen

`\w`	Buchstaben (Word)
`\d`	Zahlen (Digit)
`.`	Alles außer Zeilenvorschub

- Referenzen

`()`	Gruppierung
`\x`	x-te Rückwärtsreferenz (je nach Implementierung auch `$x`).

- Greedy

(default)	Greedy (gierig)
`?`	Reluctant, non-greedy (genügsam)

Oder-Operator / Zeichenklassen

- „A oder B“ lässt sich auf zwei Arten ausdrücken:
  - **Mit Hilfe des oder-Operators** `A|B`
  - **Mit Hilfe von Zeichenklassen** `[AB]`

- Vorteile des oder-Operators:
  - **Die Alternativen können komplexe Ausdrücke sein:** `A|(BC)*`

- Vorteile von Zeichenklassen:
  - **Die Auswertung geht schneller (siehe nachfolgende Folien).**

Auswertung von PCRE

- Nur für Suche in Texten
- Methode: Backtracking
  - **Keine nichtdeterministischen endlichen Automaten**

- Buchstabe für Buchstabe des Textes wird durchlaufen.
- Es wird geprüft, welcher Teil des Patterns den Textbuchstaben „schlucken“ kann.
- Manchmal gibt es mehrere Alternativen.
  - **Die Alternativen kann man sich als Verzweigungen in einem Baum vorstellen.**
  - **Es wird (nach bestimmten Regeln) erst ein Zweig durchlaufen. Kann auf diesem Weg das komplette Pattern nicht geschluckt werden, wird anschließend der andere Zweig genommen.**
- Ist das komplette Pattern „abgefüttert“, ist das Pattern im Text gefunden.

Verzweigungen bei der Auswertung

- Beispiel: Text = ABBA

- Pattern = A|AB
  - **Das erste A im Text will geschluckt werden.**
  - **Beide As des Patterns könnten zuschlagen.**
  - **Regel: Zuerst kommt der linke Teil zum Zug.**
  - **Der gefundene Text ist also A**

- Pattern = AB|A
  - **Gefundener Text: AB**

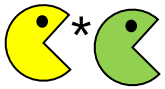
Verzweigungen bei der Auswertung

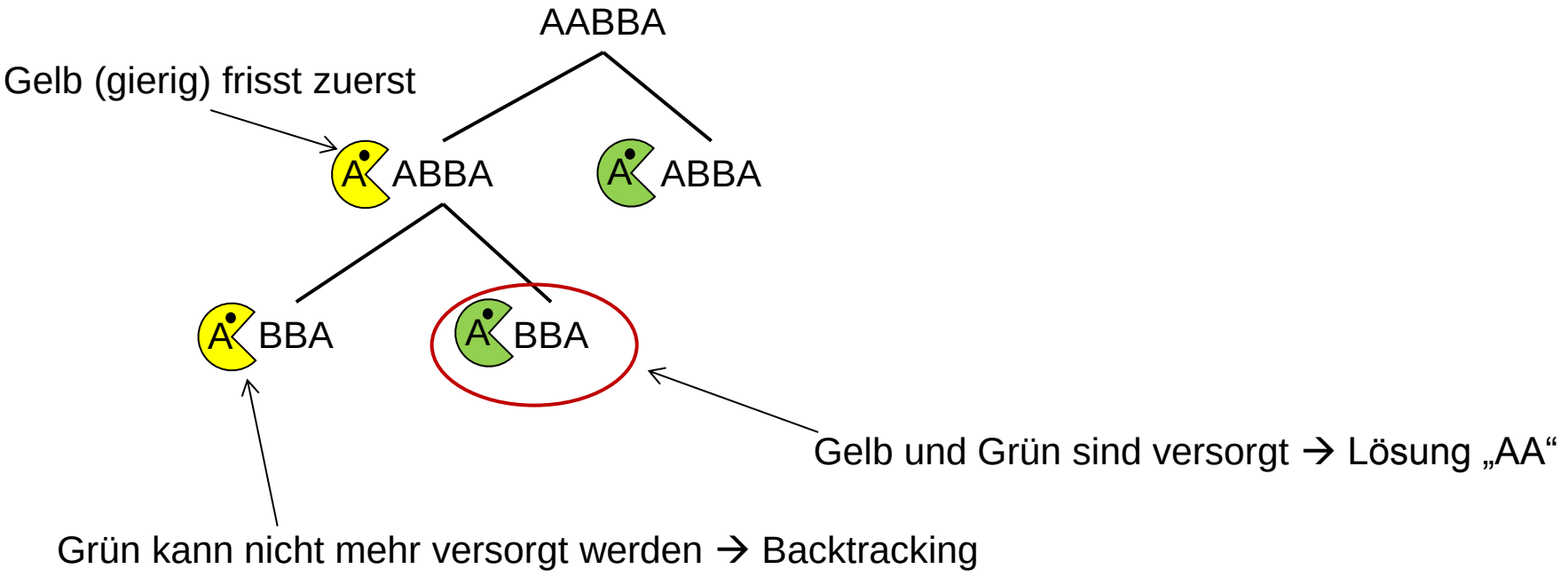
- Beispiel: Text = AABBA

- Pattern = A*A
  - **Das erste A im Text will geschluckt werden.**
  - **Beide As des Patterns könnten zuschlagen.**
  - **Regel: Der * ist „greedy“ (gierig) und schluckt erst einmal, soviel er kriegen kann.**
  - **Der gefundene Text ist AA (genauere Erläuterung später).**

- Pattern = A*?A
  - **Regel: *? ist „reluctant“ (genügsam) und nimmt nur soviel, wie unbedingt nötig.**
  - **Das A*? Überlässt das Text-A dem zweiten A des Patterns.**
  - **Gefundener Text: A**

Backtracking bei PCRE

- Beispiel: Text = AABBA; Pattern = A*A
- Darstellung des Patterns zur Verdeutlichung: 
- Text wird geschluckt: Baum zum Backtracking
  - **Baum wird in Preorder-Reihenfolge durchlaufen**



Greedy und Reluctant

Greedy	Reluctant (non-greedy)
A*	A?*
A+	A?+
A?	A??

- `[AB]` ergibt **keine** Verzweigung (kein Backtracking, schneller)

PCRE in Java

- Auch in Java kann man PCRE gebrauchen, z.B. in String.replaceFirst(..), String.split(..), String.matches(..)
  - **Siehe auch das nachfolgende Beispiel**

Beispiel

- Finden eines regulären Ausdrucks in Java

```
import java.util.*;
import java.io.*;
import java.util.regex.*;
..
public void testRegex() {
  //Testdatei laden
  Scanner sc = new Scanner(new File("pangalak.txt"));
  String txt = "";
  while (sc.hasNextLine()) {
    txt = txt + sc.nextLine()+"\n";
  }
```

Beispiel (2)

```
  //Vorbereiten des regulaeren Ausdrucks
  Pattern p = Pattern.compile("n[a-z]*e");
  Matcher m = p.matcher(txt);
  //Suchen
  while (m.find()) {
    System.out.println("Start: "+m.start()+" String:"
                       +m.group());
  }
}
```

- **Ausgabe**

```
Start: 17, String: ngalaktische
Start: 33, String: nnergurgle
Start: 51, String: nehme
...
```

Referenzen

- Klammern fangen Referenzen, die intern durchnummeriert werden.
- Die Referenzen können mit Hilfe der Nummern umgesetzt werden.
- Beispiel in Java:
  - **Ersetzen aller Doppelbuchstaben durch einfache Buchstaben:**

```
s.replaceAll("(\\w)\\1","$1");
```

- (\w) fängt einen Buchstaben und legt ihn unter der Referenz 1 ab.
- Im Pattern selbst kann der Buchstabe mit \1 referenziert werden.
- Im Replace-String benötigt man dazu $1 (Java-Eigenart).

Andere Regeln

- Andere Varianten von regulären Ausdrücken:
  - **Basic Regular Expressions (BRE) (POSIX-Standard)**
  - **Extended Regular Expressions (ERE) (POSIX-Standard)**

- Vereinfachte (unvollständige) Varianten:
  - **Windows (DOS-) Wildcards**
  - **MS Office-Platzhalterzeichen**

- Vorbemerkungen
- Nicht ernstzunehmende Verfahren
- Elementare Sortierverfahren
  - Insertion-Sort
- Höhere Sortierverfahren
  - Quick-Sort
  - Merge-Sort (Mischen)
- Spezialisierte Sortierverfahren
  - Radix-Sort

Relevanz von Sortierverfahren

- Sortierverfahren sind Bestandteil vieler Anwendungen
- Laut Statistik: ca. 25% der kommerziell verbrauchten Rechenzeit entfällt auf Sortiervorgänge.
  (Seite 63 in T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen. 1996, Spektrum, Akademischer Verlag, Heidelberg, Berlin)
- Sortierte Datensätze können
  - viel effizienter durchsucht werden (siehe Binäre Suche)
  - leichter auf Duplikate geprüft werden
  - von Menschen leichter gelesen werden
- Auch andere praxisrelevante Aufgaben können auf Sortierproblem zurückgeführt werden, z.B.
  - Median bestimmen
  - Bestimmung der k kleinsten Elemente

Aufzählung der Sortierverfahren

- **Große Anzahl von Sortierverfahren:**

  Binary-Tree-Sort, Bogo-Sort, Bubble-Sort, Bucket-Sort, Comb-Sort, Counting-Sort, Gnome-Sort, Heap-Sort, Insertion-Sort, Intro-Sort, Merge-Sort, OET-Sort, Quick-Sort, Radix-Sort, Selection-Sort, Shaker-Sort, Shear-Sort, Shell-Sort, Simple-Sort, Slow-Sort, Smooth-Sort, Stooge-Sort

Klassifikationskriterien

- Effizienz
- Speicherverbrauch
- Intern / Extern
- Stabil / Instabil
- allgemein / spezialisiert

Klassifikation: Effizienz

- Das wichtigste Klassifikationskriterium ist die **Effizienz**.
- Einteilung in:

- Schlechter als $O(n^2)$: nicht ganz ernst gemeinte Verfahren

- $O(n^2)$: Elementare Sortierverfahren
- Zwischen $O(n^2)$ und $O(n \cdot \log n)$
- $O(n \cdot \log n)$: Höhere Sortierverfahren

- Besser als $O(n \cdot \log n)$: Spezialisierte Verfahren.

Klassifikation: Effizienz (2)

- Von diesen Verfahren sollten Sie die Effizienz kennen:

- $O(n^2)$: Elementare Sortierverfahren
  - **Bubble-Sort, Insertion-Sort, Selection-Sort**
- Zwischen $O(n^2)$ und $O(n \cdot \log n)$
  - **Shell-Sort**
- $O(n \cdot \log n)$: Höhere Sortierverfahren
  - **Heap-Sort, Merge-Sort, Quick-Sort**
- Besser als $O(n \cdot \log n)$: Spezialisierte Verfahren.
  - **Radix-Sort**

Klassifikation: Effizienz (Spezialfälle)

- Spezialfälle sind:
  - **Datenmenge sehr klein (<50)**
  - **Datenmenge sehr groß (passt nicht in Hauptspeicher)**
  - **Daten sind schon „vorsortiert“ (nur wenige Elemente sind nicht am richtigen Platz).**
  - **Daten stehen nicht in einem Feld sondern in einer verketteten Liste.**

Weitere Klassifikationskriterien

- Speicherverbrauch
- Rein vergleichsbasiertes Verfahren (sehr universell einsetzbar) oder spezialisiertes Verfahren (muss an jeweiliges Problem angepasst werden):

- Stabil / Instabil

  (behalten Datensätze mit gleichen Schlüsseln relative Reihenfolge?)
  Relevant bei mehrmaligem Sortieren nach verschiedenen Schlüsseln

Beispiel für stabiles Sortieren

- Namen nach Vor- und Nachname sortieren:
  „Meier, Martin“, „Meier, Ulla“, „Schmitz, Heinz“, „Ernst, Eva“

1. Sortieren zuerst nach Vornamen:
   „Ernst, Eva“, „Schmitz, Heinz“, „Meier, Martin“, „Meier, Ulla“

2. Dann Sortieren nach Nachnamen:
   - Stabil: „Ernst, Eva“, „Meier, Martin“, „Meier, Ulla“, „Schmitz, Heinz“
   - Instabil, z.B. „Ernst, Eva“, „Meier, Ulla“, „Meier, Martin“, „Schmitz, Heinz“

- Das wichtigste stabile Verfahren ist Merge-Sort. Außerdem sind Radix-Sort, Insertion-Sort und Bubble-Sort stabil.

Beste Sortierverfahren

- Normalerweise **Quicksort.**

- 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 zu sortieren sind.**
  - **die Daten schon vorsortiert sind.**

- Radix-Sort, falls
  - **sich ein hoher Programmieraufwand für ein sehr schnelles Verfahren lohnt.**

Hilfsmethoden zum Sortieren (Java)

```
static void swap(int array[], int index1, int index2) {
      int temp;
      temp = array[index1];
      array[index1] = array[index2];
      array[index2] = temp;
}
```

Für Klassen, die das Interface List implementieren (ArrayList, …) geht auch:

```
Collections.swap(List<?> list, int i, int j)
```

3.5.1 Nicht ganz ernstzunehmende Verfahren

O(n·n!): Bogo-Sort

- „The archetypal perversly awful algorithm“ (Wikipedia).
- Würfelt solange alle Elemente durcheinander, bis das Feld (zufällig) sortiert ist.
- Zitat aus dem Internet:
  - **Looking at a program and seeing a dumb algorithm, one might say „Oh, I see, this program uses bogo-sort.“**

```
public void sort(List a) {
  while (isSorted(a)==false) {
    Collections.shuffle(a);
  }
}
```

```
public boolean isSorted(List a) {
  for (int i=1; i<a.size(); i++) {
    if isLastSmaller(a.get(i-1),
                     a.get(i)) {
      return false;
    }
  }
  return true;
}
```

$O(n^{\log(n)/2})$: Slow-Sort

- Vorgehensweise wird hier nicht erklärt
  - **-> Wikipedia**
- Versucht, die Arbeit soweit wie möglich zu vervielfachen (multiply and surrender).
- Das Ergebnis wird erst dann berechnet, wenn die Lösung nicht weiter hinausgezögert werden kann.
- Ziel: Auch im besten Fall ineffizienter als alle anderen Sortierverfahren.
- Ist aber trotzdem ein echtes Sortierverfahren.

O(n³): Stupid-Sort

- Sehr kurzer Code.
- Variante von Bubble-Sort.
- Kann auch rekursiv implementiert werden („*with recursion, stupid-sort can be made even more stupid*“ (Wikipedia)).

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

3.5.2 O(n²): Einfache Sortierverfahren

Übersicht

- Die bekannten einfachen Sortierverfahren sind:

Bubble-Sort **Selection-Sort** **Insertion-Sort**

- Das beste Verfahren der 3 ist **Insertion-Sort**. Es wird ausführlich vorgestellt.
- Selection-Sort wird kurz angesprochen.
- Bubble-Sort wird nicht weiter erklärt.

- Dafür wird ein besonders einfacher Algorithmus namens **Simple-Sort** behandelt.

Simple-Sort / Selection-Sort

- Simple-Sort
  - **$O(n^2)$, mit hohem Vorfaktor.**
  - **Leicht zu merken.**
- Wenden Sie es an:
  - **wenn Sie keine Zeit oder Lust zum Nachdenken haben.**
  - **wenn die Felder so klein sind, dass der Algorithmus nicht effektiv sein muss.**
  - **wenn niemand sonst Ihren Code zu sehen bekommt.**
- Selection-Sort
  - **Eines der wichtigeren elementaren Verfahren.**
  - **Wird ein elementares Verfahren benötigt, wird aber meistens das etwas schnellere InsertionSort verwendet.**

Prinzip: Simple-Sort und Selection-Sort

- Das Grundprinzip ist für beide Sortierverfahren gleich
- SimpleSort ergibt einen besonders einfachen Code, ist aber langsamer.
- Grundprinzip:

for (int i=0; i<array.length-1; i++)	
	Suche das kleinste Element zwischen i und dem rechten Feldende.
	Vertausche dieses Element mit dem Element i.

Simple-Sort (Code)

- Das Suchen und Vertauschen wird bei SimpleSort auf ganz spezielle Weise gemacht:
  - **Gehe vom i. Element aus nach rechts.**
  - **Jedes Mal, wenn ein kleineres Element als das auf Position i auftaucht, dann vertausche es mit dem i. Element.**

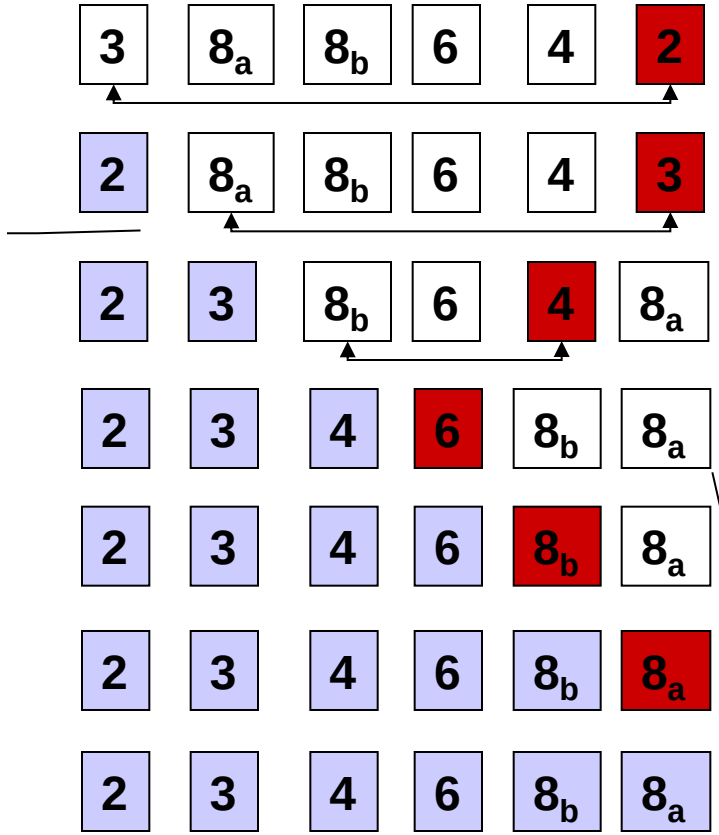
```
public void simpleSort (int[] a) {

  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);
      }
    }
  }
}
```

Selection-Sort (Code)

- „Normale“ Vorgehensweise: Erst kleinstes Element suchen, dann mit Element i vertauschen.

```
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 zu Selection-Sort



SelectionSort ist instabil:

Sortierte Teilfolge

Kleinstes Element

Keine Vorteile, wenn Feld schon sortiert.

Sortieren durch Einfügen: InsertionSort

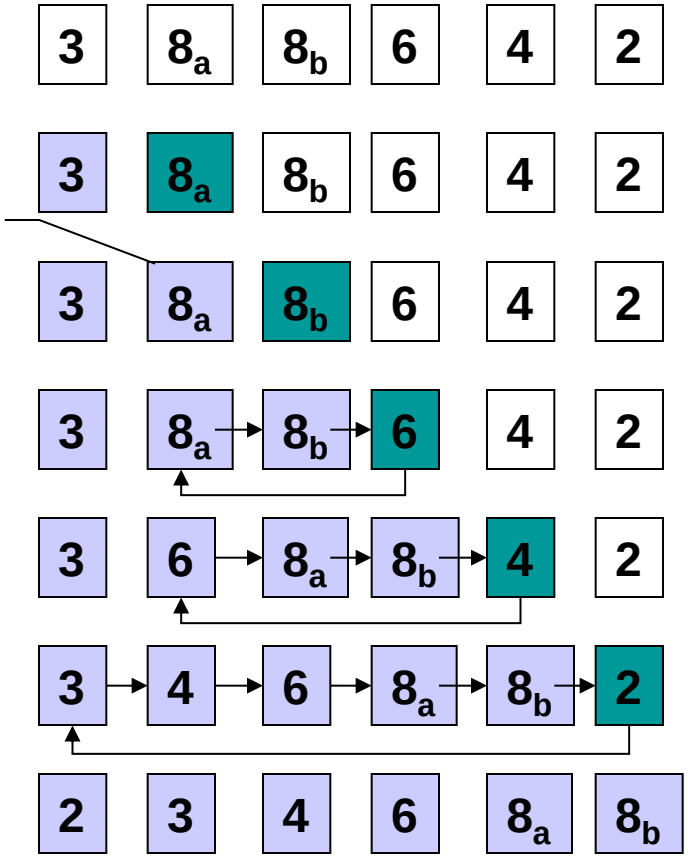
- In den meisten Fällen der schnellste elementare Suchalgorithmus.

Grundidee (am Beispiel der Sortierung eines Kartenstapels)

1. **Starte mit der ersten Karte einen neuen Stapel.**
2. **Nimm jeweils die nächste Karte des Originalstapels und füge diesen an der richtigen Stelle in den neuen Stapel ein.**

Beispiel zu InsertionSort

InsertionSort ist stabil: Vertauschung nur bei Ungleichheit



Sortierte Teilfolge

Einzusortierendes Element

InsertionSort in Java

```
public static void insertionSort(int[] array){

    for (int i=1; i < array.length; i++) {
        int m = array[i];

        // fuer alle Elemente links von aktuellem Element
        int j;
        for (j=i; j>0; j--) {
            if (array[j-1]<m) {
                break;
            }
            // größere Elemente nach hinten schieben
            array[j] = array[j-1];
        }

        // m an freiem Platz einfügen
        array[j] = m;
    }
}
```

Komplexitätsanalyse von InsertionSort

- Wir zählen die Anzahl der Vergleiche (Anzahl der „Bewegungen“ ist ungefähr gleich).

- ***n-1* Durchläufe**: Im Durchlauf mit Nummer *k* (*k* $\in$ *[2;n]*):
  - höchstens *k-1* Vergleiche.
  - mindestens 1 Vergleich (und keine Bewegung).

- **Best Case** - vollständig **sortierte** Folge: *n-1* Vergleiche, keine Bewegungen ➔ O(n).

- **Average Case** - jedes Element wandert **etwa in Mitte des sortierten Teils**:

$$\sum_{k=2}^{n} k/2 = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right) \approx \frac{n^2}{4}$$ Vergleiche ➔ O(n^2).

Bewertung

SimpleSort:

- Einfach zu implementieren.
- Langsam.

SelectionSort:

- Aufwand ist unabhängig von der Eingangsverteilung (Vorsortierung).
- Es werden nie mehr als O(n) Vertauschungen benötigt.

BubbleSort:

- Stabil
- Vorsortierung wird ausgenutzt.
- Langsam

InsertionSort:

- Stabil
- Vorsortierung wird ausgenutzt
- Für ein elementares Suchverfahren ($O(n^2)$) schnell.

Analyse elementarer Sortierverfahren

Verfahren	Laufzeitmessungen (nach Wirth, Sedgewick)	Vosortierung ausnutzen	Stabil
SimpleSort	*330**		
SelectionSort	*120-200*		
InsertionSort	*100*	*X*	*X*
BubbleSort	*250-400*	*X*	*X*

* Eigene Messung

InsertionSort:

- Stabil
- Vorsortierung wird ausgenutzt
- Für ein elementares Suchverfahren ($O(n^2)$) schnell.

3.5.3 O(n· log n): Höhere Sortierverfahren

Übersicht

- Die bekannten höheren Sortierverfahren sind:

Quick-Sort	**Merge-Sort**	**Heap-Sort**

- Das meist beste Verfahren der 3 ist **Quick-Sort**. Es wird ausführlich vorgestellt.
- Ebenso **Merge-Sort**, der in einigen Fällen Vorteile hat.

- Heap-Sort wird kurz angesprochen.
- Gute Beschreibung aller 3 Verfahren in:
  - http://www.linux-related.de/coding/sort/sort_main.htm

Quick-Sort: Verbreitung

- Die überwiegende Mehrheit der Programmbibliotheken benutzt Quick-Sort.
  - **z.B. Sun-Java, Gnu-Java, C#/.NET (nach Wikipedia), C#/Mono, Ruby, NAG, ...**
- In fast allen Fällen sind zwei Optimierungen eingebaut: „Median of three“ und „Behandlung kleiner Teilfelder“.
- Gnu-C++ benutzt Intro-Sort (Quick-Sort-Variante).

- Bei Objekten kann Stabilität wichtig sein. Hier verwendet Java (wie Python) Merge-Sort.

QuickSort: Prinzip

Prinzip: *divide-and-conquer* (‚teile' und ‚herrsche')[1]

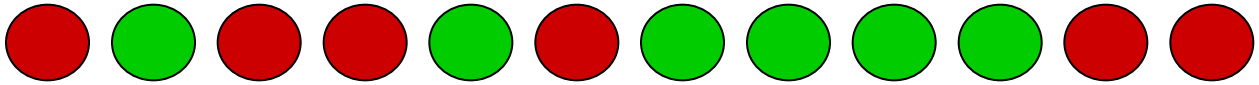
Rekursiver Algorithmus

- Müssen 0 oder 1 Elemente sortiert werden $\Rightarrow$ Rekursionsabbruch
- Wähle ein Element als „Pivot-Element" aus.
- Teile das Feld in 2 Teile:
  - **Ein Teil mit den Elementen größer als das Pivot.**
  - **Ein Teil mit den Elementen kleiner als das Pivot.**
- Wende den Algorithmus rekursiv für beide Teilfelder an.

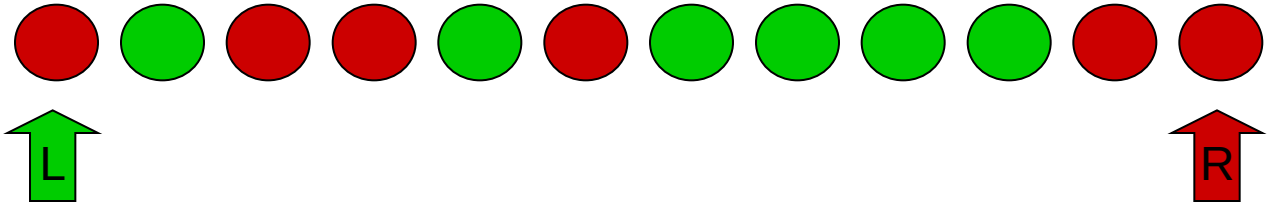
1: Gegenteiliges Prinzip: *multiply-and-surrender (Problem vergrößern, bis man aufgibt).*

Zerlegung bei Quicksort

- Beispiel des Vorgehensweise bei der Zerlegung:
  - **Ein Feld besteht aus roten und grünen Elementen.**
  - **Die roten Elemente sollen auf die rechte Seite, die grünen auf die linke Seite.**

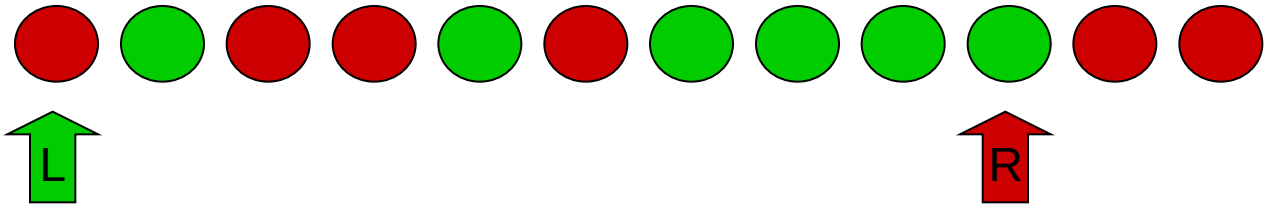


  - **Zwei Zeiger werden eingeführt.**
    - **Ein linker Zeiger. Links von ihm stehen nur grüne Elemente.**
    - **Ein rechter Zeiger. Rechts von ihm stehen nur rote Elemente.**

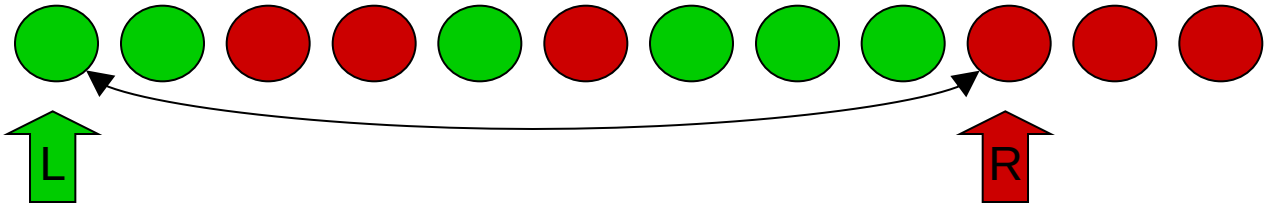


Zerlegung bei Quicksort (2)

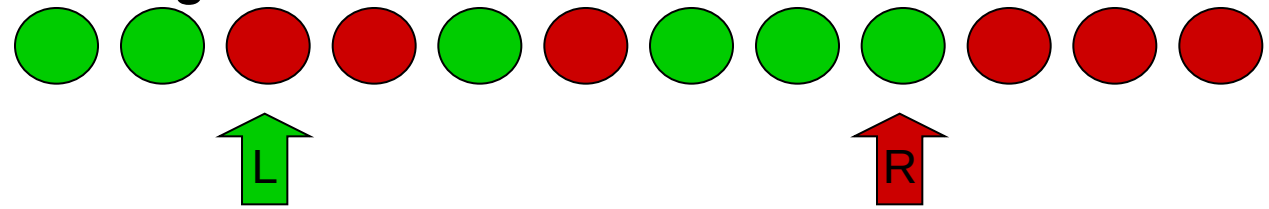
– **Die Zeiger rücken so weit vor, wie es geht. Sie stehen jetzt über einem Element der gegenteiligen Farbe.**



– **Diese Elemente werden getauscht**

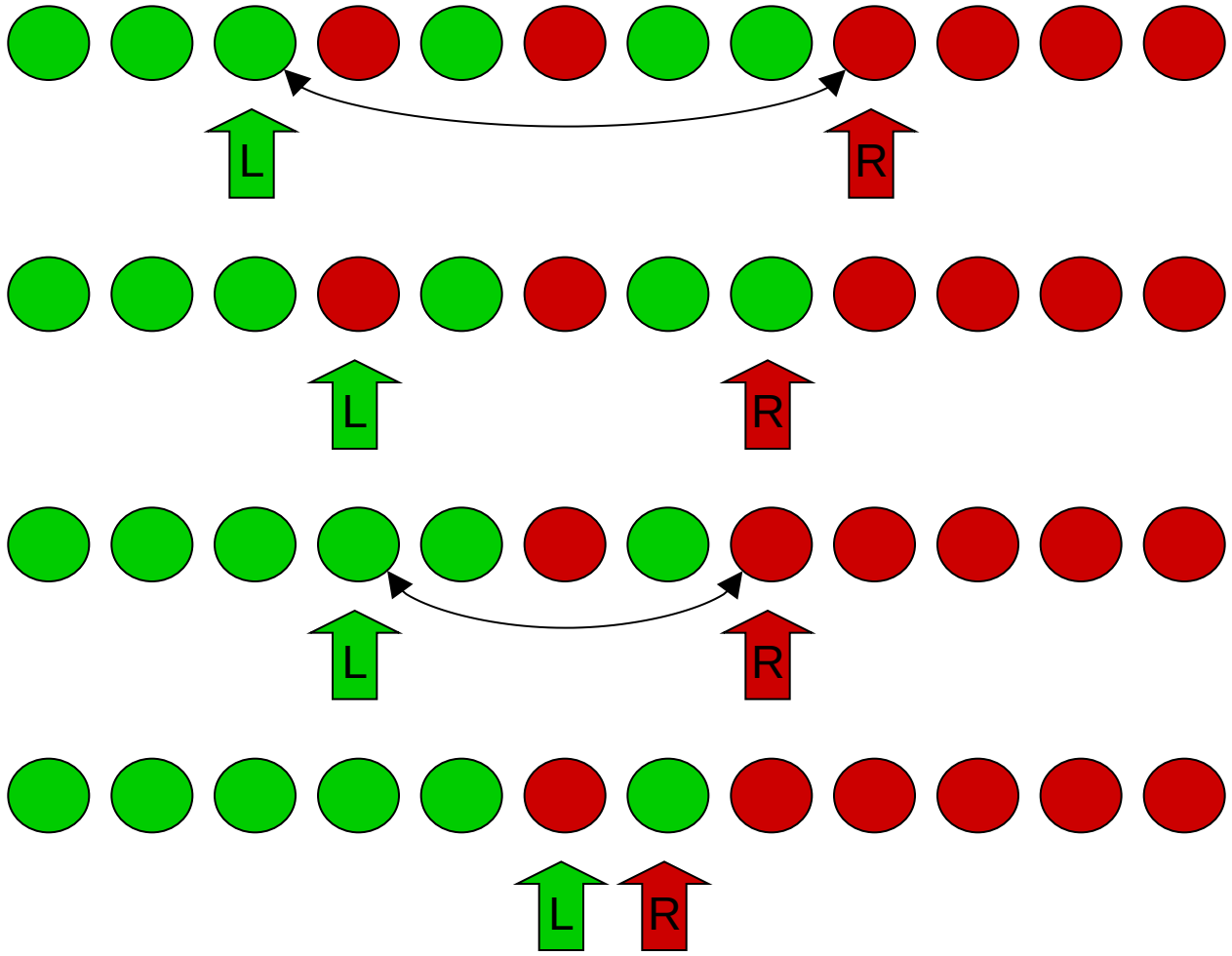


– **Die Zeiger können weiter vorrücken**



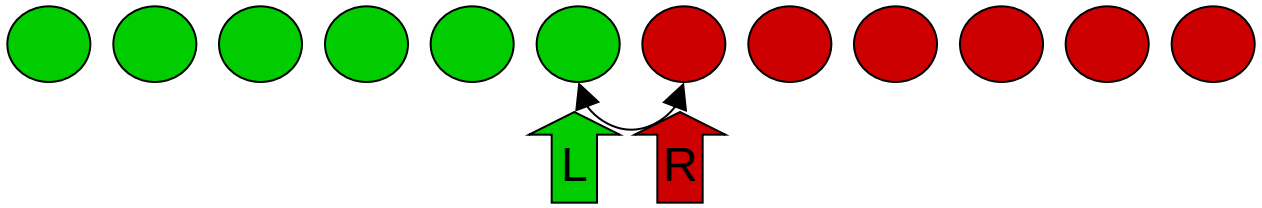
Zerlegung bei Quicksort (3)

- Tauschen
- Vorrücken
- Tauschen
- Vorrücken

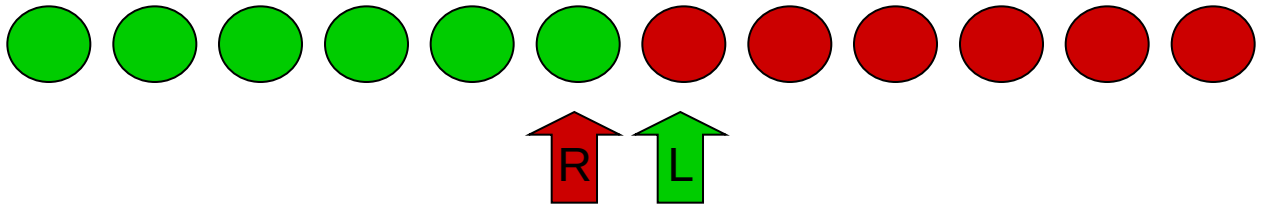


Zerlegung bei Quicksort (4)

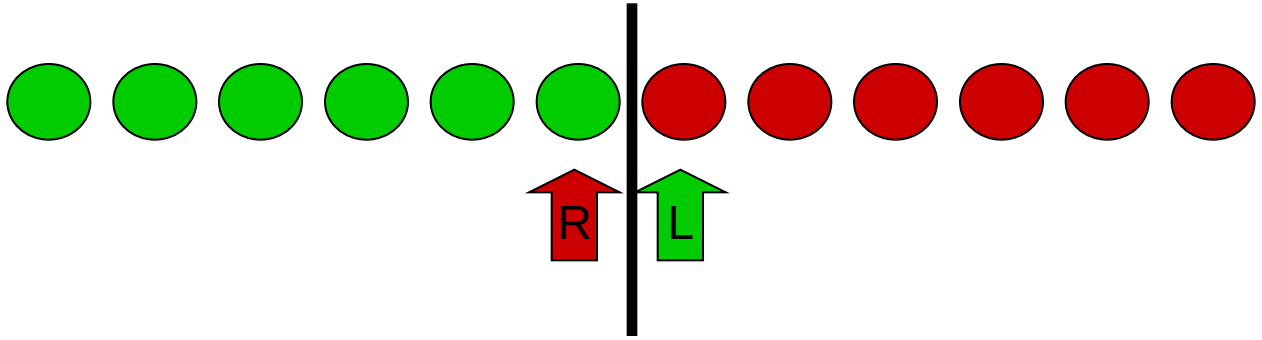
- Tauschen



- Vorrücken



- Der linke und der rechte Zeiger sind jetzt vertauscht. Das bedeutet, dass die Elemente getrennt sind.



Pivot-Element

- Wahl des Pivot-Werts `pivot` =*a[k]*:
  - z.B. mittleres *(k=n/2)*, erstes *(k=1)*, oder letztes *(k=n)* Element.
  - in den folgenden Beispielen immer mittleres Element

- Ab jetzt ist nur der Wert des Pivots relevant, nicht die Stelle, wo es steht.

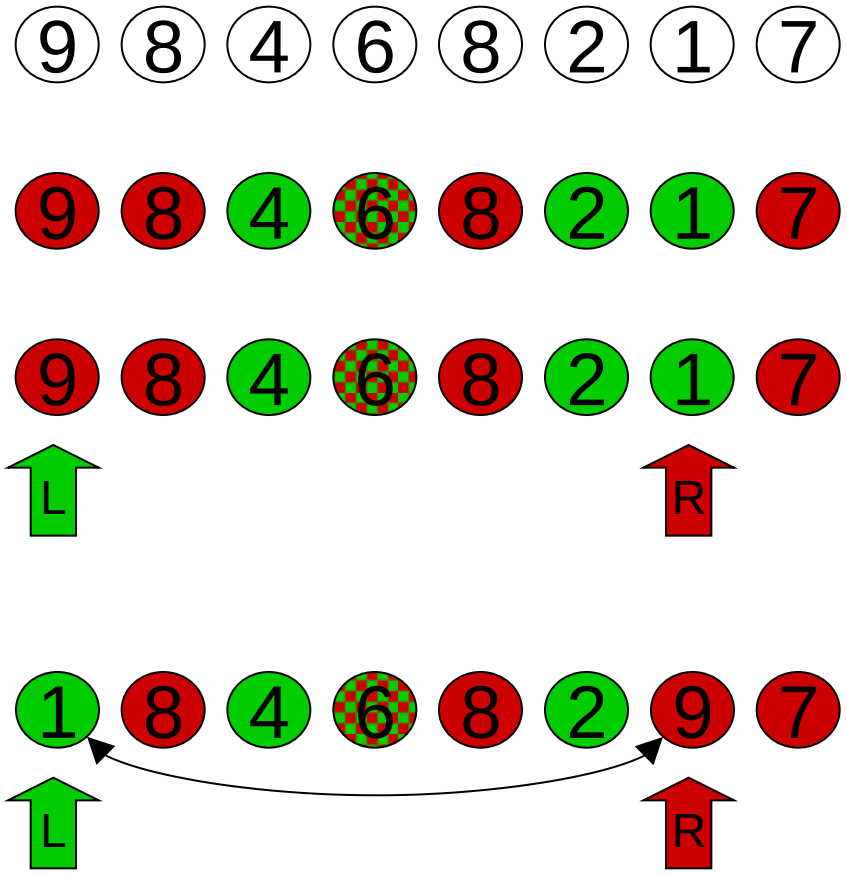
- Übertragen auf das vorige Schema:
  - **Rote Elemente: Größer als Pivot**
  - **Grüne Elemente: Kleiner als Pivot.**

Pivot-Element (2)

- Durch das Pivot-Element gibt es beim Tauschen einige Sonderregeln:
  - **Alle Elemente gleich dem Pivot werden auf jeden Fall getauscht. Sie sind also sowohl rot als auch grün.**
    - Darum bewegen sich die Pivot-Elemente (sehr wahrscheinlich) an eine andere Stelle.
  - **Nach einem Tausch bewegen sich die Zeiger grundsätzlich ein Element weiter.**
    - So wird verhindert, dass ein Pivot-Element zweimal getauscht wird.

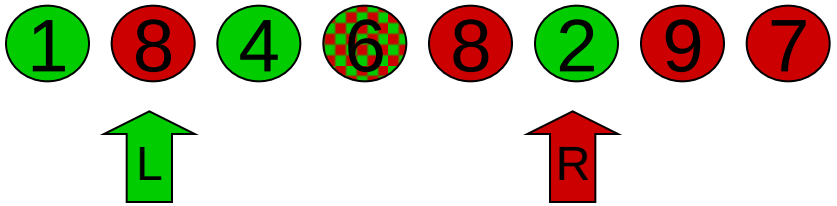
Beispiel zu Quick-Sort (1)

- Unsortiertes Feld
- Pivot wählen: 6
- Vorrücken
- Tauschen

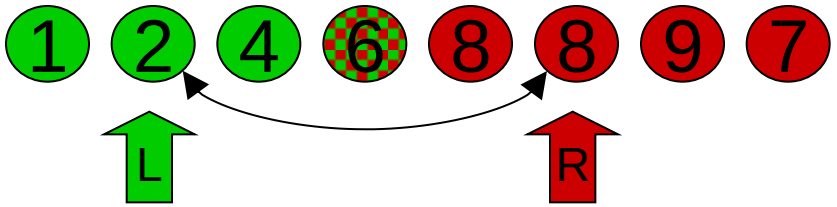


Beispiel zu Quick-Sort (2)

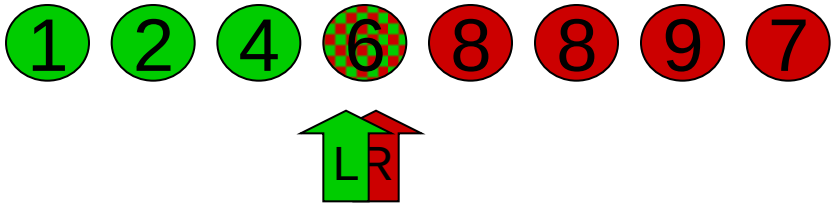
- Vorrücken



- Tauschen

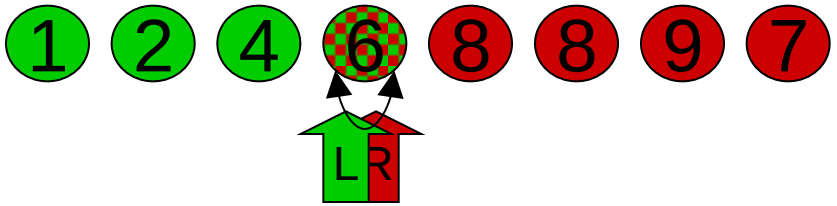


- Vorrücken

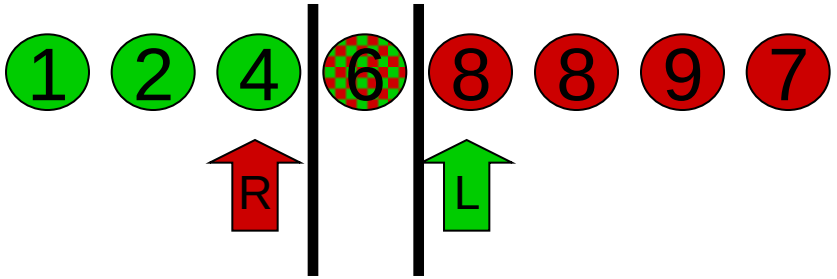


Beispiel zu Quick-Sort (3)

- Tauschen (nur formal)



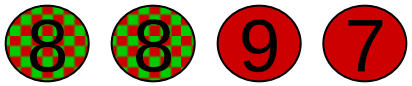
- Nach jeder Vertauschung rücken die Zeiger grundsätzlich weiter.
- Vorrücken und trennen



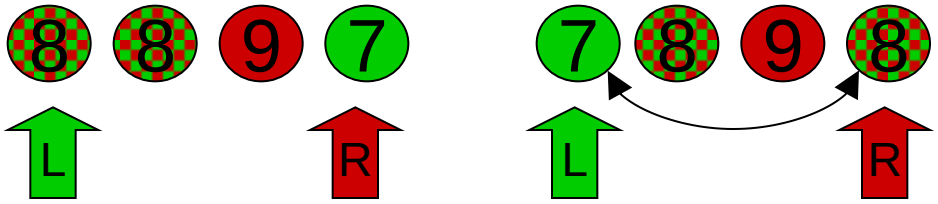
- Das rechte und das linke Teilfeld müssen noch sortiert werden.
- Das Pivot-Element in der Mitte ist bereits am richtigen Platz.

Beispiel zu Quick-Sort (4)

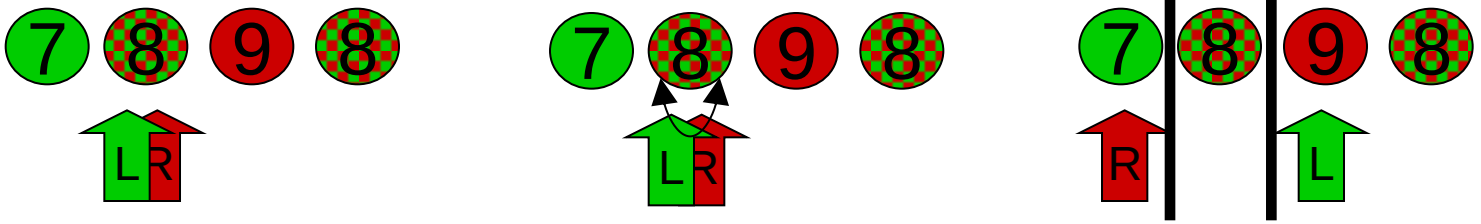
- Rechte Seite: Pivot wählen.



  - **Andere Elemente mit dem gleichen Wert, wie das Pivot-Element werden auch zum Pivot-Element**
- Sortieren



- Nach jeder Vertauschung rücken die Zeiger grundsätzlich weiter.



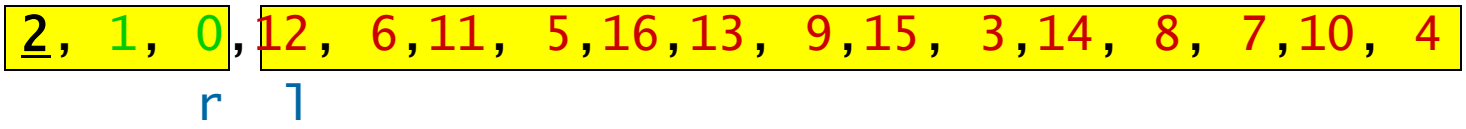
Beispiel zu Quick-Sort (5)

- usw. usf. bis alle Teilfelder nur noch aus einem Element bestehen (Übung).

Warum Quick-Sort funktioniert

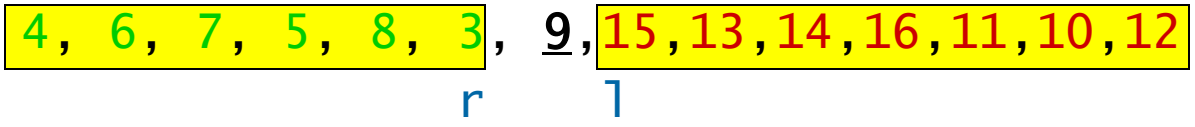
Fallunterscheidung:

– Entweder vollständige Trennung in zwei Teillisten:



– oder es gibt zusätzlich zu den beiden Teillisten noch einen mittleren Bereich mit einem Pivot-Element. Da alle Elemente links davon kleiner oder gleich, alle Elemente rechts davon größer oder gleich sind, steht dieses Element schon an der richtigen Stelle.

–



Zeitkomplexität von Quick-Sort: Best Case

Pivot-Wert ist immer Median der Teilliste ⇒ Teillisten werden stets halbiert.

Stufe 1

1 n

n Elemente werden mit dem Pivotwert verglichen und maximal n/2 Paare vertauscht.

Stufe 2

1 n/2 n

In jeder der beiden Teillisten werden n/2 Elemente (also insgesamt n) mit dem jeweiligen Pivotwert verglichen und maximal n/4 Paare vertauscht.

Allgemein: In jeder Stufe werden n Elemente betrachtet. Abbruch bei Teillisten der Länge $n \leq 1 \Rightarrow \lfloor \mathrm{ld}\, n \rfloor$ Stufen (mit Halbierung der Teillisten).

Also: $T^b_{quicksort}(n) \in \boldsymbol{O(n \log n)}$

Zeitkomplexität v. Quick-Sort: Worst Case

Als Pivot-Wert wird stets das größte oder kleinste Element der Teilliste ausgewählt.

Dann gilt:

- Die Länge der längsten Teilliste ist (n-1) bei Stufe 1,
  (n-2) bei Stufe 2, etc.
  Allgemein : (n-i) bei Stufe i.
- Es sind (n-1) Stufen nötig.
  In jeder Stufe i werden n-i Elemente betrachtet.
- Also: $T^{w}_{quicksort}(n) \in \boldsymbol{O(n^2)}$

Zeitkomplexität v. Quick-Sort: Average Case

Aufwand im Mittel:

Genaue Analyse ist aufwändig. Resultat: $T^{av}_{quicksort}(n) \in \boldsymbol{O(n \log n)}$

Genauer:
#Vergleiche im mittleren Fall ist nur um etwa 39% größer als im Besten Fall. QuickSort ist also auch im Mittleren Fall eine sehr gute Wahl.

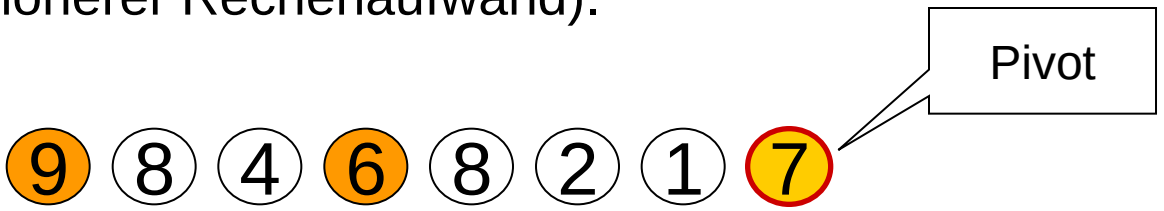
Hauptziel ist Vermeidung des schlimmsten Falls.

Probleme von Quick-Sort: Pivot-Elemente

- Wenn die Pivot-Elemente unglücklich gewählt sind, erhält man das wesentlich schlechtere Zeitverhalten $O(n^2)$.
- Dazu Anmerkung:
  Wenn man beim Zerlegen das erste oder letzte Feldelement als Pivot wählt, dann liegt der Worst Case bei sortierten Feldern vor.
  - **Auch nahezu sortierte Felder haben schon $O(n^2)$.**
- Pivot geschickter wählen.
  - **Einfachste Möglichkeit: Mittleres Element als Pivot wählen. Dann liegt der Best case bei sortierten Felder vor.**
  - **Man will den Quick-Sort aber noch sicherer machen.**

Optimierung: Pivot-Elemente

- *Median-of-three-Methode* zum Auswählen des Pivots:
  - Es werden drei Elemente als Referenz-Elemente, z.B. vom Listenanfang, vom Listenende und aus der Mitte gewählt. Das Element mit dem mittleren Schlüsselwert wird als Referenz-Element gewählt.
  - Kann auf mehr als drei Elemente ausgebaut werden.
  - Die Zahl der Referenz-Elemente kann von der Anzahl der Sortier-Elemente abhängig gemacht werden.
  - Die Referenz-Elemente können auch zufällig ausgewählt werden (dann aber höherer Rechenaufwand).



Optimierung: Pivot-Elemente (2)

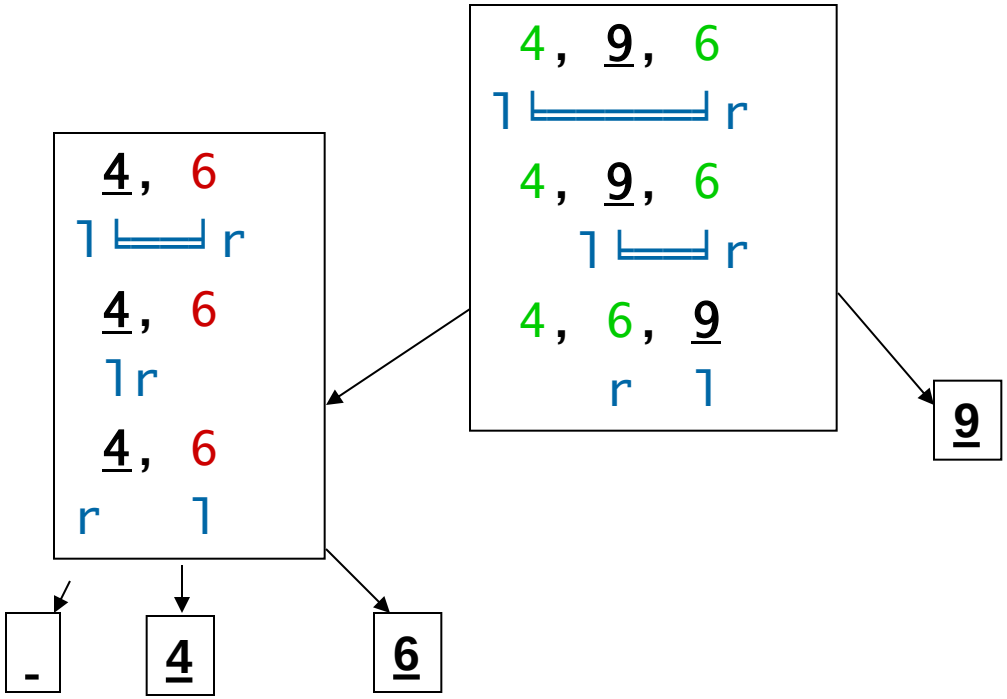
Sprache	Bibl.-Aufruf	Referenz-Elemente (r) für Sortier-Elemente (s)
Java (Sun)	java.util.Arrays.sort	$s=7 \Rightarrow r=1$ $7<s<40 \Rightarrow r=3$ $s \geq 40 \Rightarrow r=9$
Java (Gnu)	java.util.Arrays.sort	$7<s<40 \Rightarrow r=3$ $s \geq 40 \Rightarrow r=9$
C# / Mono	System.Collections.ArrayList.sort	r=3
Ruby	Array.sort	$s<60 \Rightarrow r=1$ $60 \leq s<200 \Rightarrow r=3$ $s \geq 200 \Rightarrow r=7$

Optimierte Variante: Intro-Sort

- Optimierte Quicksort-Variante
  - **Z.B. in Gnu C++**
- Ab einer gewissen Rekursionstiefe wird zu Heap-Sort gewechselt.
  - **Heap-Sort hat O(n log n) auch im schlechtesten Fall. Damit kann $O(n^2)$ nicht auftreten.**
  - **Allgemein ist Heap-Sort aber langsamer als Quick-Sort.**
  - **Große Rekursionstiefe deutet bei Quick-Sort auf Worst-Case-Probleme. Daher wechselt man hier zu Heap-Sort.**

Probleme von Quick-Sort: Rekursionsabbruch

- Einfache Lösung: Rekursionsabbruch, wenn die Teilliste 0 oder 1 Element enthält.
- Aber: Letzte Rekursionsdurchgänge nicht mehr effektiv.



Optimierung: Rekursionsabbruch

- **Kleine Teilliste mit InsertionSort sortieren**:
  - Die Grenze für eine kleine Teilliste ist nicht klar festgelegt.
    - Die Standardwerke (Knuth, Sedgewick) empfehlen 9.
    - Im Internet findet man aber auch andere Werte zwischen 3 und 32.

Sprache	Bibl.-Aufruf	Rekursionsabbruch bei max. n Elementen
Java (Sun)	java.util.Arrays.sort	6
Java (Gnu)	java.util.Arrays.sort	7
C# / Mono	System.Collections.ArrayList.sort	3
Ruby	Array.sort	2

Optimierung: Doppelte Schlüssel

- Oft kommen viele Sortierschlüssel mehrfach im Feld vor.
  - **Beispiel: 4,6,8,2,4,6,2,8,4,6,1,9,5,4,8,2,6,4**
- Für genau diesen Fall gibt es eine weitere Optimierung:

Quick-Sort mit Dreiwegezerlegen

- Keine Standardoptimierung, deshalb hier keine nähere Erklärung.
  - **Nähere Erklärung z.B. in Sedgewick, Algorithmen in Java**
- Sun-Java hat diese Optimierung (als eine der wenigen Bibliotheken) eingebaut.

Bewertung

Quicksort:

- Schnellster der oft verwendeten allgemeinen Algorithmen.
- Programmbibliotheken implementieren ganz überwiegend QuickSort.
- Nutzt Vorsortierung nicht aus.
- Im Worst Case nur $O(n^2)$.
- Schlecht für kleine n (<20).
- Zwei Standard-Optimierungen

Standard-Optimierung: Rekursionsabbruch für kleine n:

- Verwendung von InsertionSort
- Grenze liegt zwischen 3 und 32.

Standard-Optimierung: Pivot ist Median mehrerer Elemente:

- 3 oder mehr Pivot-Elemente.
- Dadurch Worst Case sehr unwahrscheinlich.
- Langsamer aufgrund zusätzlicher Rechnenschritte.

Internes / Externes Sortieren

- Voraussetzung für bisher behandelte Verfahren:
  Schneller Zugriff auf einen beliebigen Datensatz (wahlfreiem Zugriff).
  - Bezeichnung „Internes Sortieren“

- Dies ist in manchen Fällen nicht möglich:
  - bei sehr großen Datenbeständen z.B. auf Hintergundspeichern (externen Speichern) mit sequentiellem Zugriff.
  - bei verketteten Listen.

- Hier werden Verfahren verwendet, die lediglich sequentiellen Zugriff benötigen.
  - Bezeichnung „Externes Sortieren“.
  - Wichtigstes Verfahren: Merge-Sort.

Sortieren durch Mischen: Merge-Sort

- **Prinzip:**
  - Sortiere die Daten paarweise

C: 5 1 8_A 7 3 8_B 12 2

C: 1 5 7 8_A 3 8_B 2 12

  - Füge jeweils zwei Paare zu sortierten Viererfolgen zusammen (Mischen)

C: 1 5 7 8_A 2 3 8_B 12

  - Füge zwei Vierfolgen zu einer sortierten Achterfolge zusammen (Mischen)

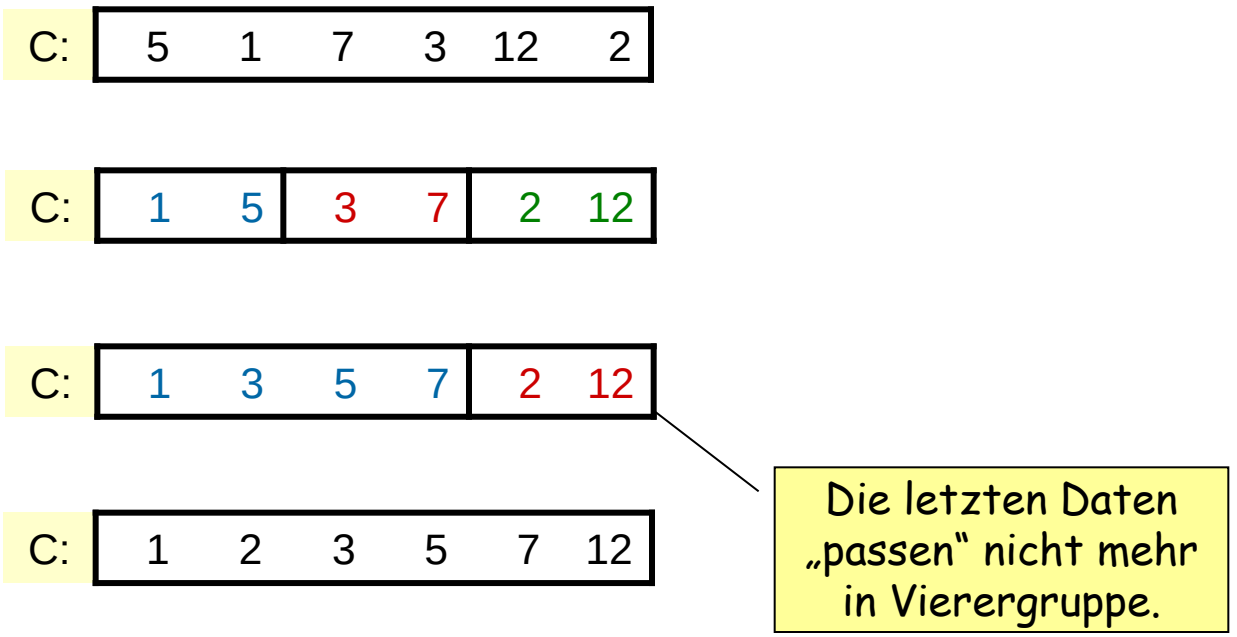
C: 1 2 3 5 7 **8_A** **8_B** 12

  - Usw. bei größeren Datensätzen.
  - Der eigentliche Aufwand liegt in den Mischvorgängen.

8 aus A hat Vorrang vor 8 aus B ⇒ MergeSort ist stabil.

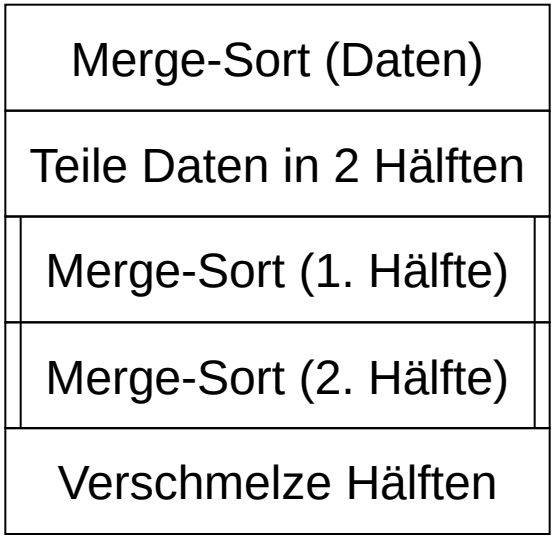
Krumme Anzahl von Datensätzen

- **Prinzip (iterative Variante):**



Iterativer / Rekursiver MergeSort

- Außer der bisher behandelten iterativen Variante gibt es auch eine rekursive Variante.
- Krumme Zahlen werden dabei gleichmäßiger aufgeteilt, wodurch die rekursive Variante etwas schneller ist.



MergeSort: Merge

Merge

Eingabe: nA sortierte Gruppen in Sequenz A,
nB sortierte Gruppen in Sequenz B,
Gruppenlänge len mit (nA+nB)*len = N

Ausgabe: N Elemente in Sequenz C
in sortierten Gruppen der Länge 2*len

Über alle Paare gA, gB von Gruppen der Länge len aus Sequenz A bzw. aus Sequenz B

Solange noch Elemente sowohl in gA als auch in gB

a = kleinstes Element aus gA, b = kleinstes Element aus gB

a <= b

Wahr	Falsch
Entnehme a aus gA und hänge es an C an	Entnehme b aus gB und hänge es an C an

Verschiebe restliche Elemente von gA nach C

Verschiebe restliche Elemente von gB nach C

Komplexitätsanalyse von MergeSort

- Sequenz C wird $k = (ld\ n)$-mal zerlegt und dann wieder zusammengemischt.

- Verteilen und Mischen erfordern jeweils $O(n)$ Operationen.

$\Rightarrow T_{mergesort}(n) \in O(n \log n)$ (auch im worst case)

Optimierung 1: Felduntergrenze

- Viele sinnvolle Optimierungsmöglichkeiten.

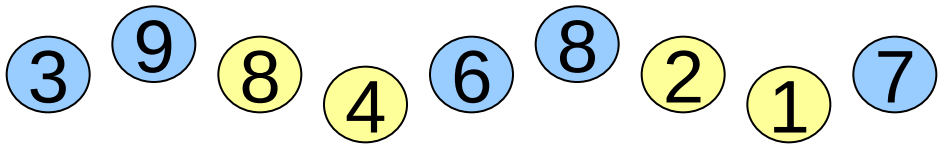
1. Feld wird nicht in Einzelelemente geteilt, sondern in Gruppen zu n Elementen, die im 1. Schritt mit InsertionSort sortiert werden.
   - **Ähnlich wie bei QuickSort.**
   - **gcj-Java nimmt 6 Elemente als Grenze.**
   - **Python nimmt 64 Elemente als Grenze (variiert aber manchmal).**
   - **Sun-Java hat diese Optimierung nicht.**

Optimierung 2: Trivialfall

- Java überprüft beim Zusammenfügen:
  - **ist das kleinste Element der einen Teilfolge größer ist als das größte Element der anderen Teilfolge?**
  - **wenn ja, beschränkt sich das Zusammenfügen auf das Hintereinandersetzen der beiden Teilfolgen.**
  - **Damit wird eine Vorsortierung ausgenutzt.**

Optimierung 3: Natürlicher Mergesort

- Weitergehende Ausnutzung der Vorsortierung.
- Jede Zahlenfolge besteht aus Teilstücken, die abwechselnd monoton steigend und monoton fallend sind.



Die Idee ist, diese bereits sortierten Teilstücke als Ausgangsbasis des Merge-Sorts zu nehmen.

- Bei nahezu sortierten Feldern werden die Teilstücke sehr groß und das Verfahren sehr schnell (O(n) im Best case).

Zusammenfassung der Optimierungen

Optimierung	Java (Sun)	Java (gcj)	Python
Felduntergrenze	-	6	64
Trivialfall	x	x	x
Natürlicher Mergesort	-	-	x

Merge-Sort auf der Festplatte

- Merge-Sort auf externen Laufwerken:
  - **Teile die Daten in zwei gleich große Dateien A und B.**
  - **Lese jeweils ein Datum von A und B. Schreibe das sortierte Paar abwechselnd in zwei neue Dateien C und D.**
  - **Lese jeweils ein Paar aus C und D. Verschmelze die Paare und schreibe die Vierergruppen abwechselnd in A und B.**
  - **usw.**
  - **Wiederhole, bis in einer Datei die komplette sortierte Folge sthet.**

MergeSort:

- O(n log n), auch im Worst case
- MergeSort kann auch externe Daten sortieren.
- Stabil
- Benötigt zusätzlichen Speicherplatz.
- Durchschnittlich langsamer als Quicksort.
- Nutzt Vorsortierung nicht aus

Optimierung: Natural MergeSort

- Nutzt Vorsortierung aus.
- Bei vorsortierten Daten häufig schneller als O(n log n).

RWTH AACHEN UNIVERSITY

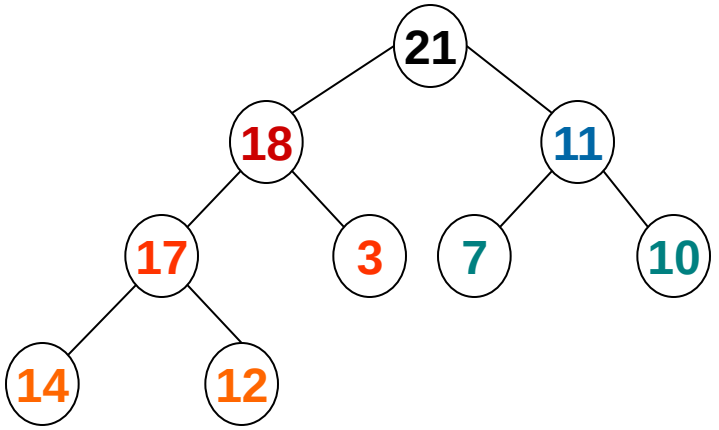
Definition:

Ein ***Heap*** ist ein Binärbaum mit folgenden Eigenschaften:

- Er ist links-vollständig
- Die Kinder eines Knotens sind höchstens so groß wie der Knoten selbst.

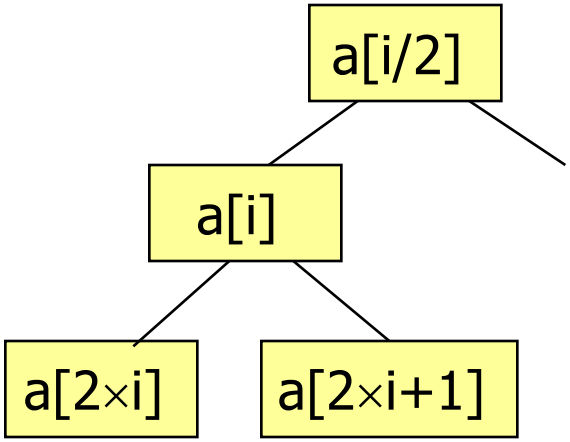
⇒ das größte Element befindet sich an der Wurzel des Heaps

Achtung: In der Literatur gibt es auch die umgekehrte Definition



Wiederholung: Array-Einbettung

Vater und Söhne zu a[i]:



Anmerkung: In Java werden Arrays beginnend mit 0 indiziert.

⇒ entweder:

- der k-te Knoten wird in dem Array-Element a[k-1] gespeichert...
- oder Array-Index 0 bleibt unbenutzt!

Heap-Sort

- Zu Beginn: Unsortiertes Feld

- Phase 1: Aufbau des Heaps
  - **nach dem Bottom-up-Verfahren**
  - **Resultat ist ein Heap, der in ein Feld eingebettet ist.**

- Phase 2: Der Heap wird geleert, indem immer wieder die Wurzel (d.h. das größte Element) entfernt wird.
  - **Die Elemente werden in absteigender Reihenfolge entfernt.**
  - **Der Heap schrumpft immer weiter.**

Trick beim Heap-Sort

- Zunächst wird das Feld komplett in einen Heap umgewandelt:

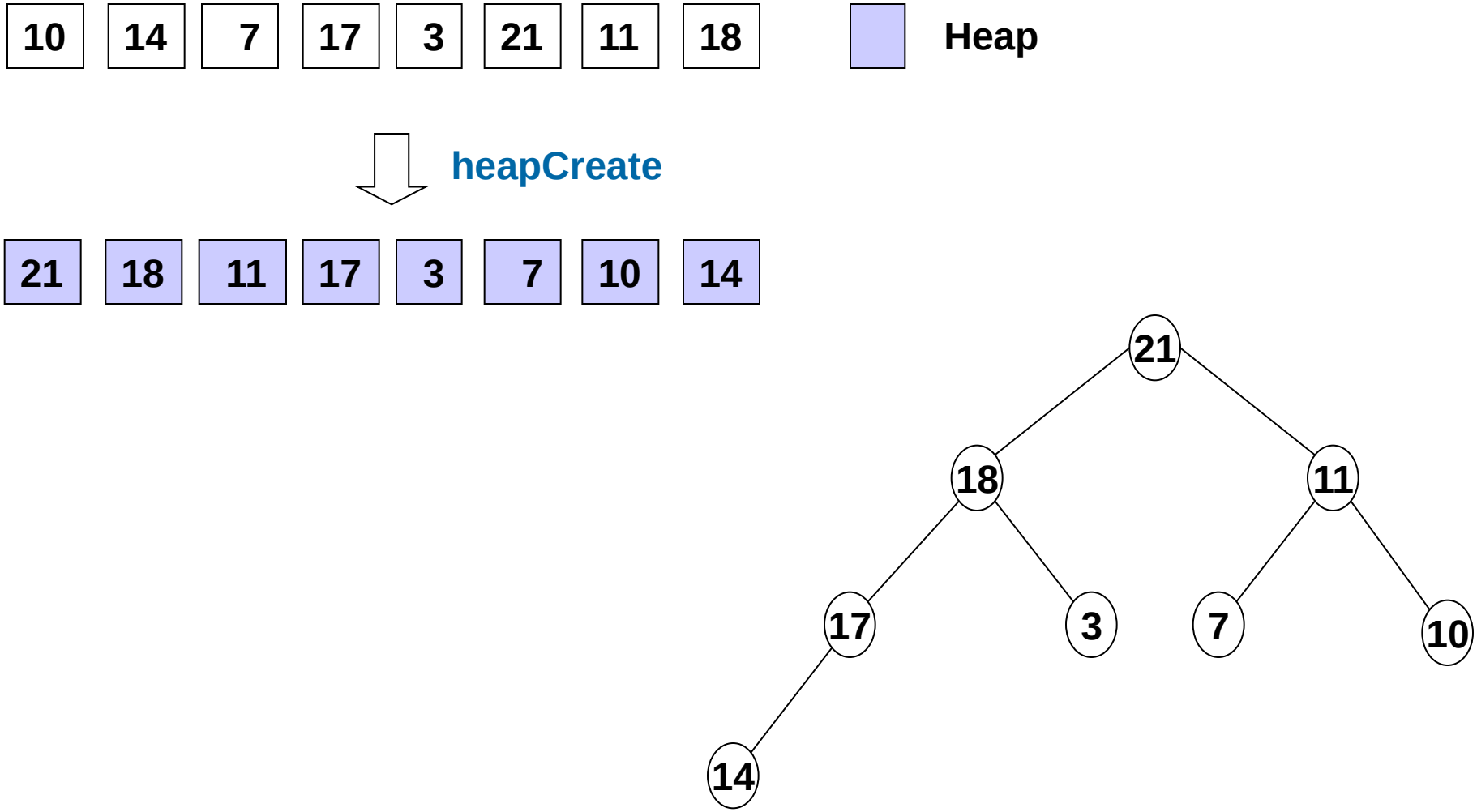
- Wenn die Wurzel entfernt wird, schrumpft der Heap um 1 Element und das letzte Feldelement gehört nicht mehr zum Heap.
- Hier kann das 1. sortierte Element untergebracht werden

Trick beim Heap-Sort (2)

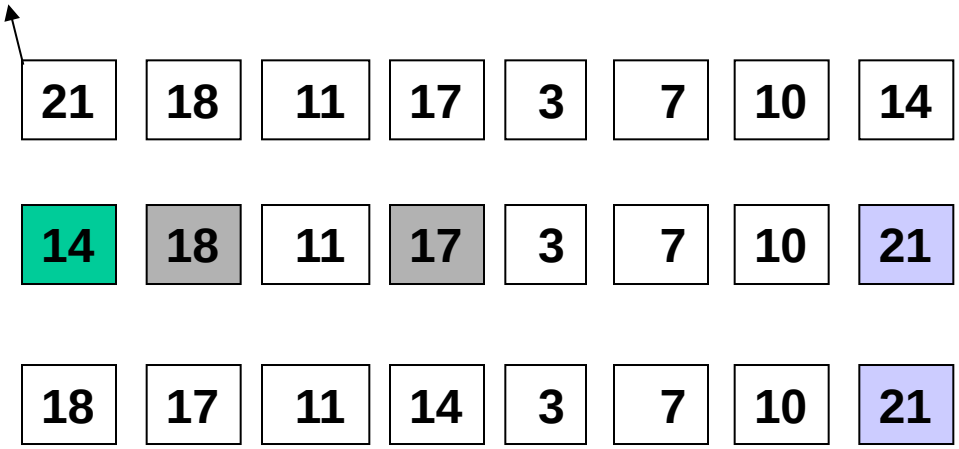
- Wenn weitere Male die Wurzel entfernt wird, schrumpft der Heap immer weiter. Die freiwerdenden Stellen werden mit den entnommenen Werten besetzt, die jetzt aufsteigend sortiert sind.

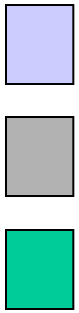
- Zuletzt ist der Heap auf 0 geschrumpft und das Feld ist sortiert.

Beispiel für HeapSort



Beispiel für Phase 2 von HeapSort

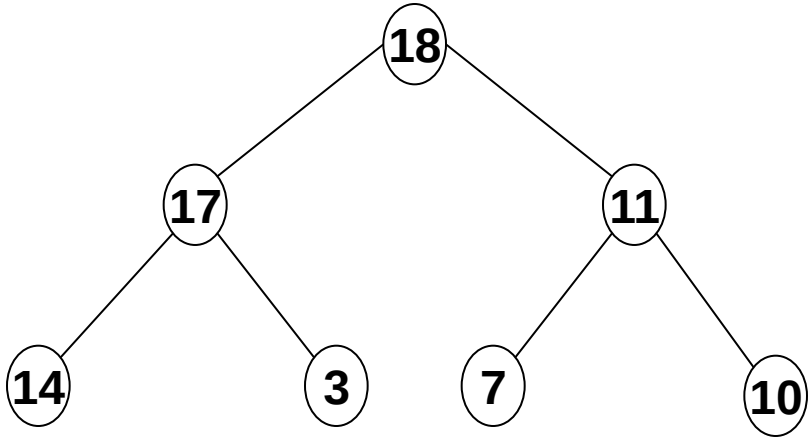




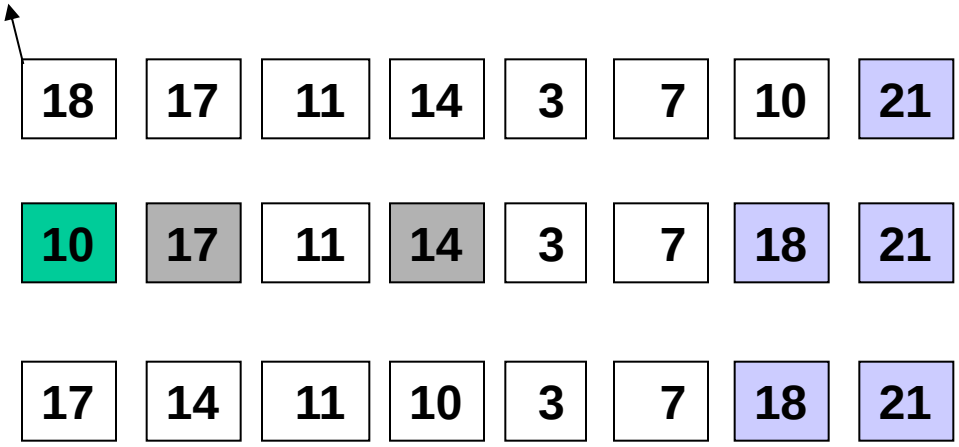
Sortierte Teilfolge

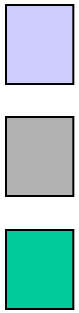
Verschobene Elemente

versickerndes Element



Beispiel für Phase 2 von HeapSort

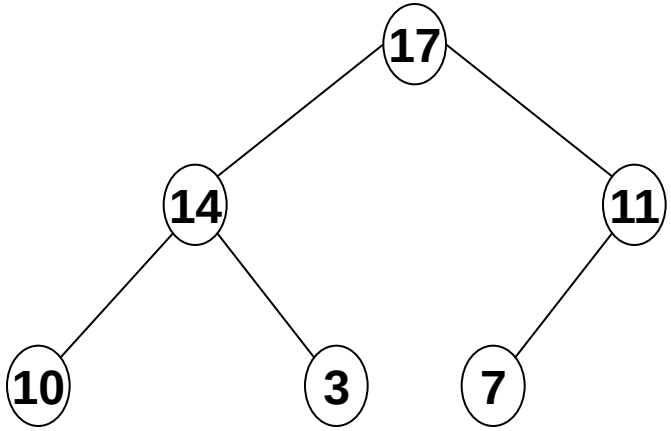




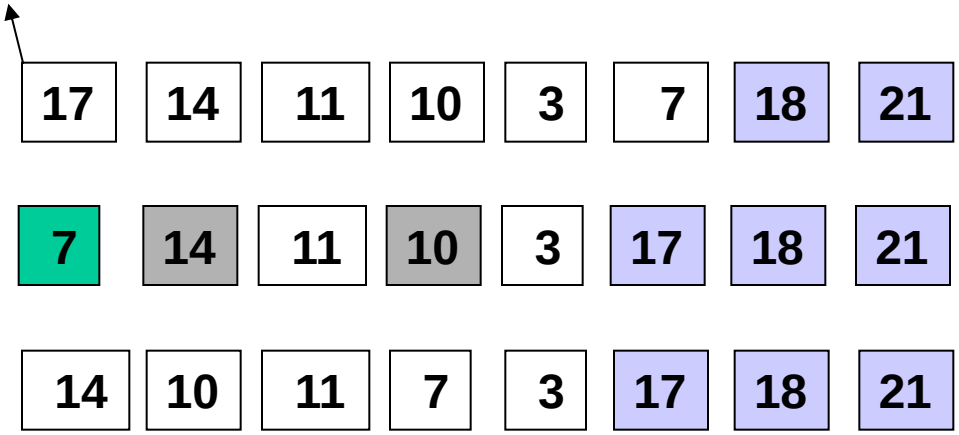
Sortierte Teilfolge

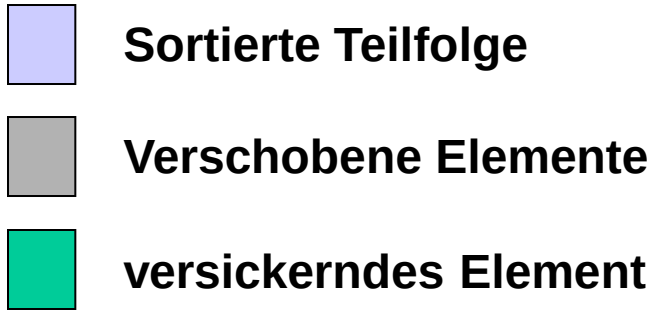
Verschobene Elemente

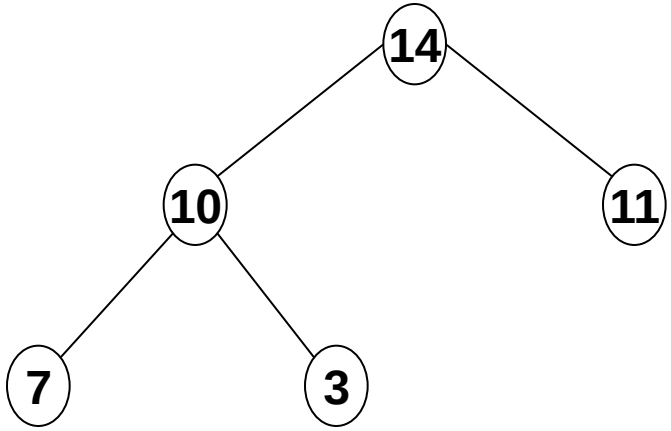
versickerndes Element



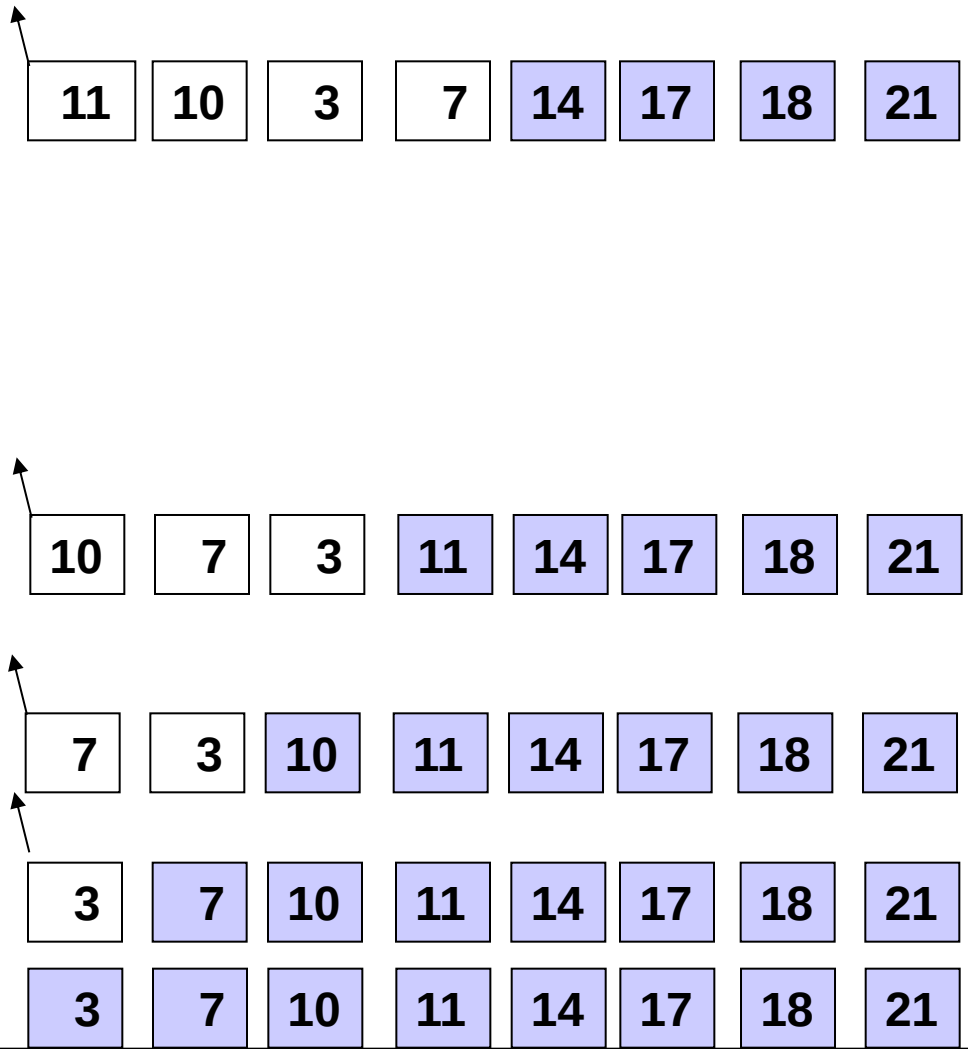
Beispiel für Phase 2 von HeapSort

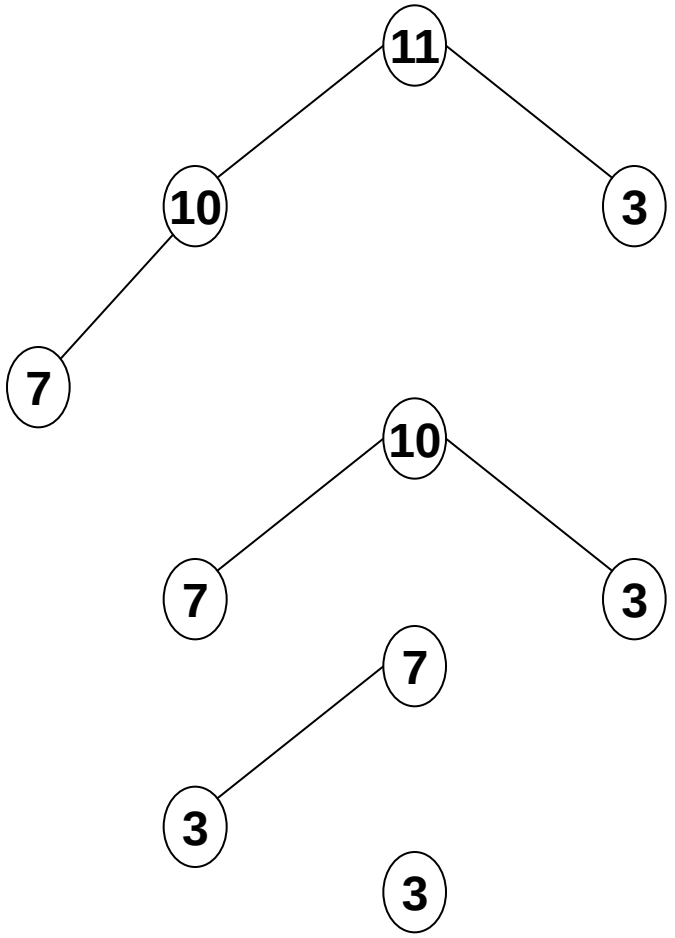






Beispiel für Phase 2 von HeapSort





Komplexität von HeapSort im Worst Case

1. Aufbau des Heap (heapCreate):
   Für n/2 Elemente wird sink ausgeführt.
   Maximal werden dabei log(n+1) Vergleiche durchgeführt.

   $\Rightarrow T^{w}_{heapCreate}(n) \leq n/2 \log(n+1) \in O(n \log n)$

2. Beim sortierten Entnehmen der Elemente mit Wiederherstellen des Heap wird sink für n-1 Elemente ausgeführt.

   $\Rightarrow T_{scndPhase}(n) \leq (n-1) \log(n+1)$

$\Rightarrow$ Für HeapSort insgesamt gilt dann:

$T^{w}_{heapSort}(n) \leq 3/2 \cdot n \cdot \log(n+1) \in O(n \log n)$

RWTH AACHEN UNIVERSITY

HeapSort:

- O(n log n), auch im worst case
  - **Hauptvorteil gegenüber Quick-Sort**
  - **Wird daher in Quick-Sort-Optimierung „Intro-Sort“ verwendet.**
- Kein zusätzlicher Speicher nötig.
- Nicht stabil.
- Vorsortierung wird nicht ausgenutzt.
- Im Normalfall langsamer als QuickSort.

Analyse schneller Sortierverfahren

Verfahren	Laufzeitmessungen (nach Wirth, Sedgewick)	Vorsortierung ausnutzen	Worst Case ok	Zus. Speicher	Stabil
QuickSort	*100*			$\Phi(\log n)$	
HeapSort	*150-200*		*X*	$\Phi(1)$	
MergeSort	*150-200*	*X*	*X*	$\Phi(n)$	*X*

Vergleich elementare - schnelle Verfahren

- Faustformeln
- Die O(n log n)-Verfahren schlagen InsertionSort ab einer Länge von 50 deutlich.
- Die O(n log n)-Verfahren schlagen ShellSort ab einer Länge von 5.000.000 deutlich.

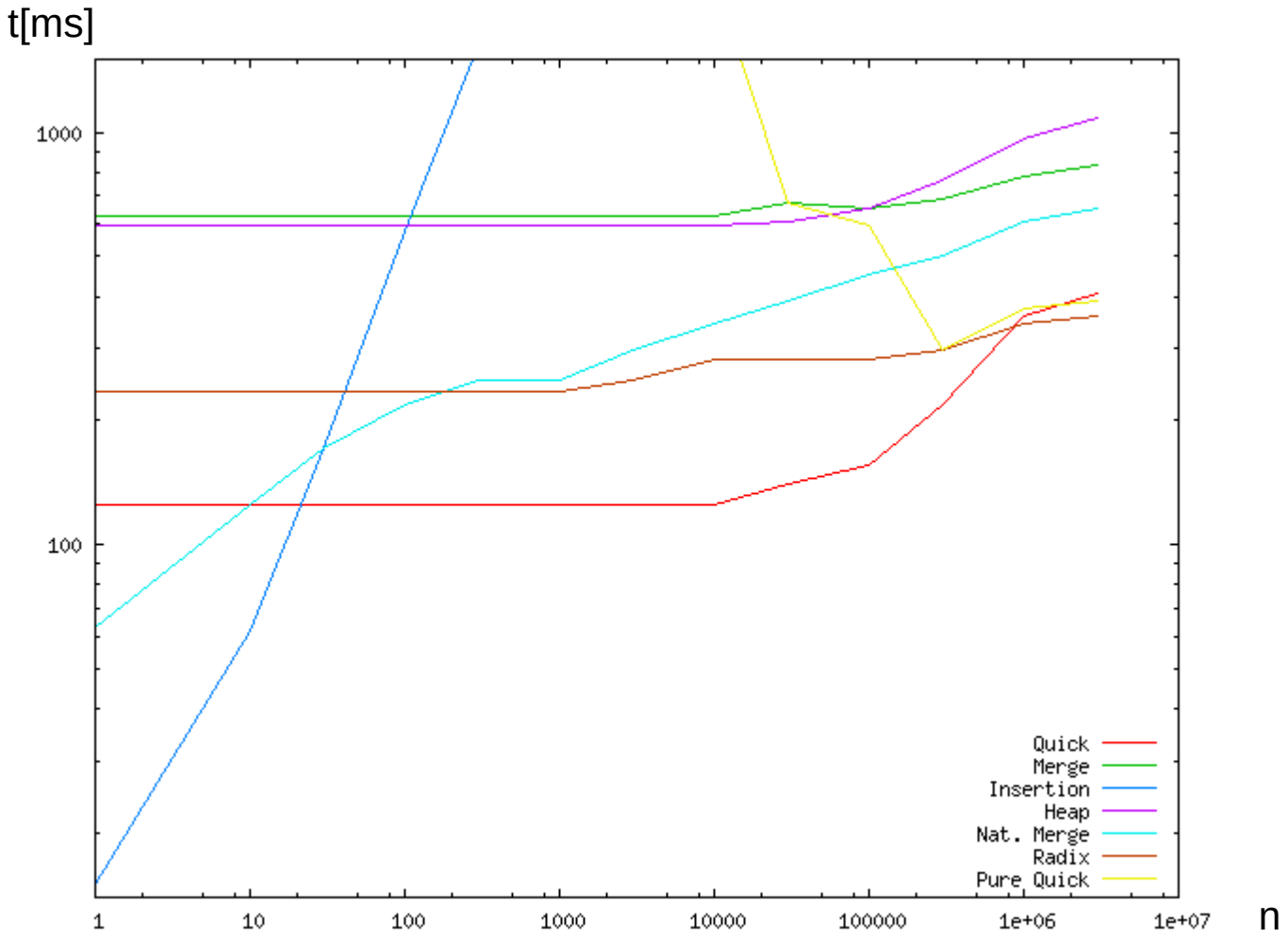
Vorsortierte Felder

- Welches Verfahren ist für nahezu sortierte Felder am besten?

- Experiment
  - **2.000.000 Elemente**
  - **Feldinhalt = Feldindex**
  - **n Elemente werden paarweise vertauscht**
  - **n variiert zwischen 0 (sortiert) und 3.000.000 (unsortiert).**

Sortierver-fahren	Sortiertes Feld	Unsortier-tes Feld
Bubble	O(n)	$O(n^2)$
Insertion	O(n)	$O(n^2)$
Quick („Pur“)	$O(n^2)$	O(n log n)
Heap, Merge, Quick	O(n log n)	O(n log n)
Nat. Merge	O(n)	O(n log n)
Radix	O(n)	O(n)

Vorsortierte Felder (2)



3.5.4. Besser als O(n log n): Spezialisierte Sortierverfahren

O(n): Radix-Sort

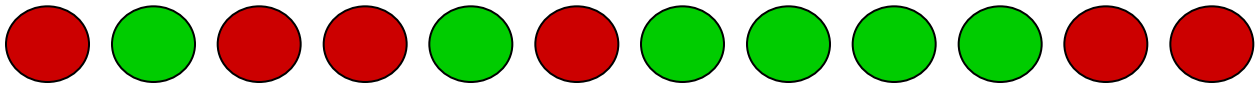
- Radix-Sort heißt eine Gruppe von Sortierverfahren mit folgender Eigenschaft:
  - **Der Sortiervorgang erfolgt mehrstufig.**
  - **Zunächst wird eine Grobsortierung vorgenommen, zu der nur ein Teil des Schlüssels (z.B. der erste Buchstabe) herangezogen wird.**
  - **Die grob sortierten Bereiche werden dann feinsortiert, wobei schrittweise der restliche Teil des Schlüssels verwendet wird.**
- Spezialisiert heißt hier, dass die Aufteilung in Grob- und Feinstufen auf das Problem angepasst sein muss.

Radix-Sort-Varianten

- Radix-Sort baut auf einer Sortierung in einzelnen Stufen auf.
  - **Diese Stufen können unterschiedlich sein.**
- Beispiel: Zweistellige Integer-Werte sollen sortiert werden.

Binärer Quick-Sort

- Möglichkeit 1: In der 1. Stufe werden die Werte ≥50 von den Werten <50 getrennt.
  - **Innerhalb jeder Stufe Aufteilung in nur 2 Gruppen.**
  - **Große Ähnlichkeit mit Quick-Sort.**
  - **Namen:**
    Radix-Quick-Sort, Binärer Quick-Sort, Radix-Exchange-Sort
  - **Leider auch nur so schnell wie Quicksort (O(n log n)).**
  - **Wird nicht weiter behandelt.**



MSD- und LSD-Radix-Sort

Die anderen Varianten benutzen Bucket-Sort (nächste Folien).

- Möglichkeit 2: In der 1. Stufe wird die erste Ziffer betrachtet.
  - **Benutzt Bucket-Sort mit Teillisten.**
  - **Namen:**
    MSD-Radix-Sort (MSD = most significant digits)
- Möglichkeit 3: In der 1. Stufe wird die letzte Ziffer betrachtet.
  - **Benutzt Bucket-Sort mit schlüsselindiziertem Zählen.**
  - **Namen:**
    LSD-Radix-Sort, Straight Radix Sort
    (LSD = least significant digits)

O(n): BucketSort

- Eignet sich für den Fall, dass der Wertebereich der Schlüssel eng begrenzt ist.
- Beispiel: Sortieren von Fussballspielern nach Toren:

20	Altintop
12	Amanatidis
14	Ballack
21	Berbatov
15	Klasnic
12	Klimowicz
25	Klose
12	Marcelinho
17	Makaay
12	Podolski
13	Smolarek
12	Thurk
16	Vittek

- Datensätze in Eimer werfen.
- Inhalt der Eimer der Reihe nach in Hauptliste einfügen.
- Datensätze sind nach Toren sortiert.

Bucket-Sort mit Teillisten

- Die Eimer sind Array-Lists

- 1. Schritt (Partitionierungsphase): Verteilung der Daten auf die Teillisten (Eimer).
- 2. Schritt (Sammelphase): Die Daten werden nach der Reihe aus den Eimern wieder in die Liste kopiert.

MSD-Radix-Sort

- Dieses Verfahren wird mehrstufig durchgeführt. Der Inhalt eines Eimers wird noch einmal mit Bucket-Sort sortiert.

1

0-10 10-20 20-30 30-40

2

0 1 2 9

Optimierung: Kleine Teilfelder

- Wie bei Quick-Sort bricht man auch bei MSB-Radix-Sort ab, wenn die Teilfelder (Eimer-Inhalte) zu klein werden.
- Zum letzten Schritt benutzt man auch hier Insertion-Sort

Bucket-Sort mit schlüsselindiziertem Zählen

Unsortiertes Feld:

Index	Tore	Spieler
0	20	Altintop
1	12	Amanatidis
2	14	Ballack
3	21	Berbatov
4	15	Klasnic
5	12	Klimowicz
6	25	Klose
7	12	Marcelinho
8	17	Makaay
9	12	Podolski
10	13	Smolarek
11	12	Thurk
12	16	Vittek

2 Hilfsfelder sind nötig:

- eines mit gleicher Größe wie das unsortierte Feld
- eines, dessen Größe der Anzahl der Eimer entspricht.

Tore	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Anzahl																													

Beispiel (2)

- Im 2. Hilfsfeld wird gezählt, welcher Wert wie oft vorkommt.

Tore	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Anzahl	-	-	-	-	-	-	-	-	-	-	-	-	5	1	1	1	1	1	-	-	1	1	-	-	-	1	-	-	-

- Jetzt wird in jedes Feld i die Summe der Felder 0..i-1 eingetragen.

Tore	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Position	0	0	0	0	0	0	0	0	0	0	0	0	0	5	6	7	8	9	10	10	10	11	12	12	12	12	13	13	13

Beispiel (3)

- Jetzt werden die Datensätze nach-einander in das 1. Hilfsfeld einsortiert.
- Der 1. Datensatz ist (20 – Altintop)
- Im Hilfsfeld 2 wird die Position ermittelt: 10
- Der Datensatz wird an Position 10 eingetragen
- Im Hilfsfeld 2 wird der Wert unter „20“ um 1 erhöht

Index	Tore	Spieler
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10	20	Altintop
11		
12		

Tore	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Position	0	0	0	0	0	0	0	0	0	0	0	0	0	5	6	7	8	9	10	10	~~10~~ 11	11	12	12	12	12	13	13	13

Beispiel (4)

- 2. Datensatz: (12 – Amanatidis)
- 3. Datensatz: (14 – Ballack)
- 4. Datensatz: (21 – Berbatov)
- 5. Datensatz: (15 – Klasnic)

Index	Tore	Spieler
0	12	Amanatidis
1		
2		
3		
4		
5		
6	14	Ballack
7	15	Klasnic
8		
9		
10	20	Altintop
11	21	Berbatov
12		

Tore	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Position	0	0	0	0	0	0	0	0	0	0	0	0	0/1	5	6/7	7/8	8	9	10	10	11	11/12	12	12	12	12	13	13	13

12

Beispiel (5)

- 6. Datensatz: (12 – Klimowicz)

Index	Tore	Spieler
0	12	Amanatidis
1	12	Klimowicz
2		
3		
4		
5		
6	14	Ballack
7	15	Klasnic
8		
9		
10	20	Altintop
11	21	Berbatov
12		

Tore	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Position	0	0	0	0	0	0	0	0	0	0	0	0	1 2	5	7	8	8	9	10	10	11	12	12	12	12	12	13	13	13

Beispiel (6)

- usw. usf.
- Die Datensätze sind jetzt im Hilfsfeld 1 sortiert.

- Dieses Sortierverfahren ist **stabil**.

Index	Tore	Spieler
0	12	Amanatidis
1	12	Klimowicz
2	12	Marcelinho
3	12	Podolski
4	12	Thurk
5	13	Smolarek
6	14	Ballack
7	15	Klasnic
8	16	Vittek
9	17	Makaay
10	20	Altintop
11	21	Berbatov
12	25	Klose

Tore	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Position	0	0	0	0	0	0	0	0	0	0	0	0	5	6	7	8	9	10	10	10	11	12	12	12	12	13	13	13	13

LSD-Radix-Sort

- Dieses Verfahren kann auch mehrmals hintereinander angewandt werden.
- Man muss aber jetzt mit der **letzten** Ziffer beginnen.

Beispiel (2)

- Beginn mit der letzten Ziffer
- Unsortiertes Feld
- Hilfstabelle

Index	Tore	Spieler
0	20	Altintop
1	12	Amanatidis
2	14	Ballack
3	21	Berbatov
4	15	Klasnic
5	12	Klimowicz
6	25	Klose
7	12	Marcelinho
8	17	Makaay
9	12	Podolski
10	13	Smolarek
11	12	Thurk
12	16	Vittek

Tore (2. Ziffer)	0	1	2	3	4	5	6	7	8	9
Anzahl	1	1	5	1	1	2	1	1	-	-

Tore (2. Ziffer)	0	1	2	3	4	5	6	7	8	9
Position	0	1	2	7	8	9	11	12	13	13

Beispiel (3)

- Beginn mit der letzten Ziffer
- Sortiertes Feld (Schritt 1):
- Hilfstabelle

Index	Tore	Spieler
0	20	Altintop
1	21	Berbatov
2	12	Amanatidis
3	12	Klimowicz
4	12	Marcelinho
5	12	Podolski
6	12	Thurk
7	13	Smolarek
8	14	Ballack
9	15	Klasnic
10	25	Klose
11	16	Vittek
12	17	Makaay

Tore (2. Ziffer)	0	1	2	3	4	5	6	7	8	9
Position	0	1	2	7	8	9	11	12	13	13

Beispiel (4)

- Erste Ziffer
- Unsortiertes Feld (Schritt 2):
- Hilfstabelle

Index	Tore	Spieler
0	20	Altintop
1	21	Berbatov
2	12	Amanatidis
3	12	Klimowicz
4	12	Marcelinho
5	12	Podolski
6	12	Thurk
7	13	Smolarek
8	14	Ballack
9	15	Klasnic
10	25	Klose
11	16	Vittek
12	17	Makaay

Tore (1.Ziffer)	0	1	2
Anzahl	0	10	3

Tore (1. Ziffer)	0	1	2
Position	0	0	10

Beispiel (5)

- Erste Ziffer
- Sortiertes Feld (Schritt 2):
- Hilfstabelle

Index	Tore	Spieler
0	12	Amanatidis
1	12	Klimowicz
2	12	Marcelinho
3	12	Podolski
4	12	Thurk
5	13	Smolarek
6	14	Ballack
7	15	Klasnic
8	16	Vittek
9	17	Makaay
10	20	Altintop
11	21	Berbatov
12	25	Klose

Tore (1. Ziffer)	0	1	2
Position	0	10	13

- Funktioniert, weil Bucket-Sort **stabil** ist.
- Alle Datensätze mit gleicher 1. Ziffer sind nach voriger Sortierung (2. Ziffer) sortiert.
- Geht auch mit mehr als 2 Stufen.

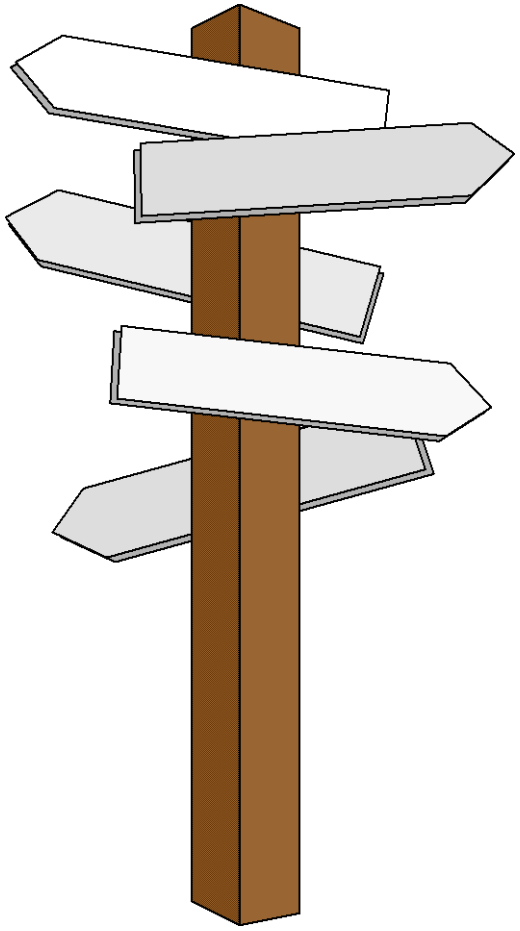
Wieviel Eimer pro Schritt?

- In der Praxis nimmt man nicht 10 Eimer (Dezimalsystem) sondern richtet sich nach dem Binärsystem, d.h. 2^n Eimer.
- Wie groß sollte man n wählen?
  - **Zu wenige Eimer sind schlecht, weil man dann zu viele Schritte braucht.**
  - **Zu viele Eimer sind auch schlecht, weil man dann zu viele leere Eimer durchsuchen muss.**
- Es wird vorgeschlagen:
  - **Sedgewick: Für 64-bit-Schlüssel (long) 2^{16} (65536) Eimer.**
  - **Linux-Related: Für 32-bit-Schlüssel (int) 2^{11} (2048) Eimer.**
- Einsatz lohnt sich nur, wenn die Anzahl der zu sortierenden Werte deutlich größer ist, als die Anzahl der Eimer.

Vergleich schneller und spezieller Sortierverfahren

Verfahren	Laufzeitmessungen (nach Wirth, Sedgewick)	Vorsortierung ausnutzen	Worst Case ok	Zus. Speicher	Stabil
QuickSort	*100*			$\Phi(\log n)$	
HeapSort	*150-200*		*X*	$\Phi(1)$	
MergeSort	*150-200*	*X*	*X*	$\Phi(n)$	*X*
MSB-Radix-Sort	*85 (bei 100.000 El.)*		*X*	$\Phi(n)$	*X*

Kapitel 4: Entwurfsprinzipien



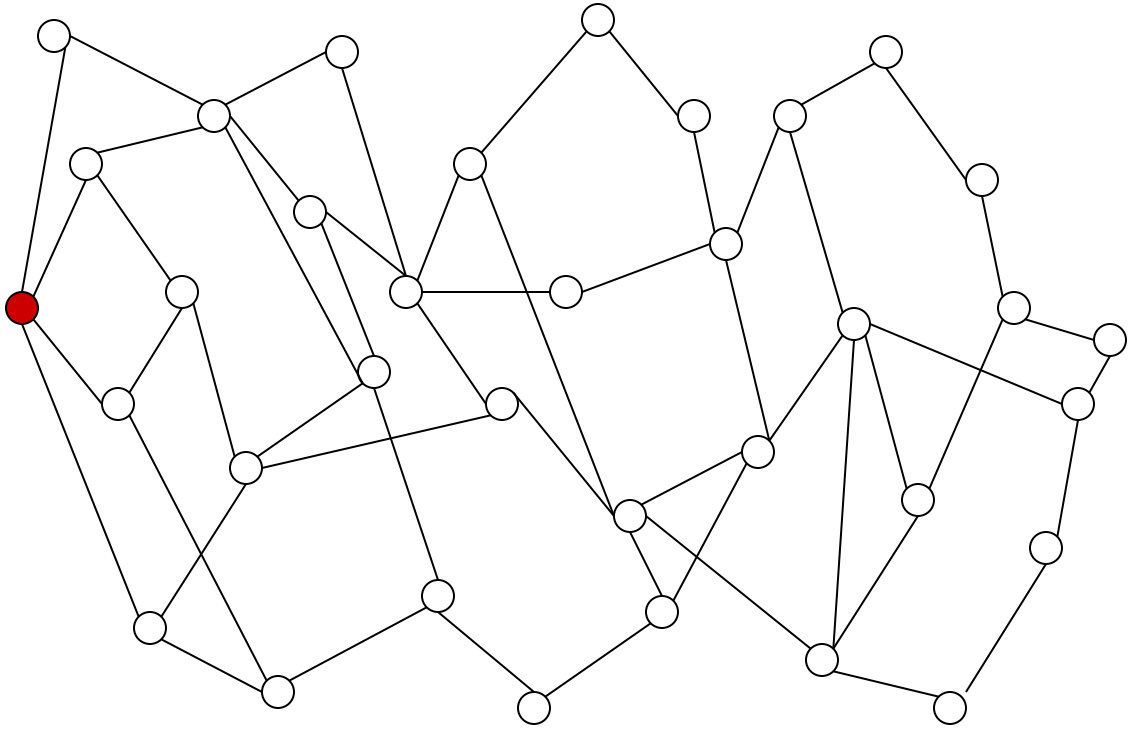
Entwurfsprinzipien für Algorithmen

- allgemeine Prinzipien für den Entwurf von Algorithmen
- viele komplexe Problemstellungen lassen sich mit diesen Prinzipien angehen
- oft werden mehrere dieser Prinzipien kombiniert.

- „greedy“ = Englisch für „gierig“
- Grundidee:
  Nächsten Schritt stets in erfolgversprechendste Richtung gehen (lokales Optimum).
- Lokale Sichtweise, nicht vorausschauend.
- Im Allgemeinen nur Näherungslösung.
- Bei manchen Problemklassen führt diese Strategie immer zum globalen Optimum.

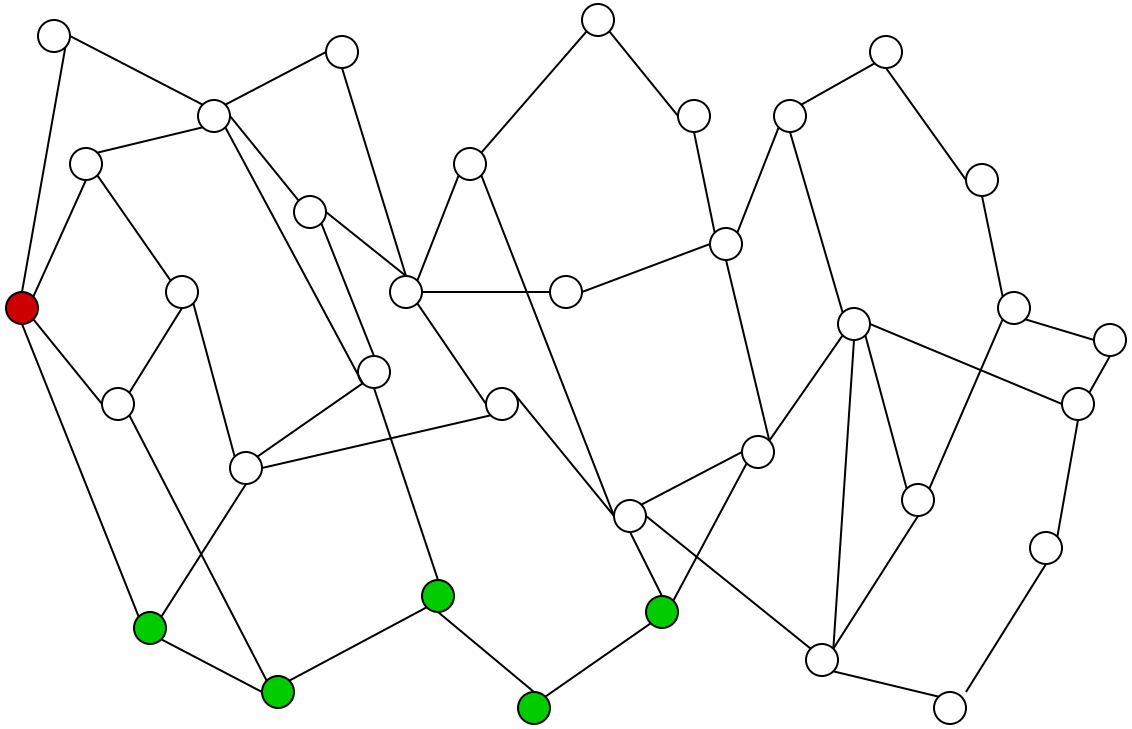
Problemstellung

- Versuche, mit 5 Schritten so weit wie möglich nach rechts zu kommen.



Näherungslösung mit Greedy

- Mache immer den größten Schritt.
  - **Gute Laufzeitkomplexität.**
  - **Lösung nicht optimal, aber (je nach Problemstellung) möglicherweise schon ausreichend.**



Andere Problemstellungen

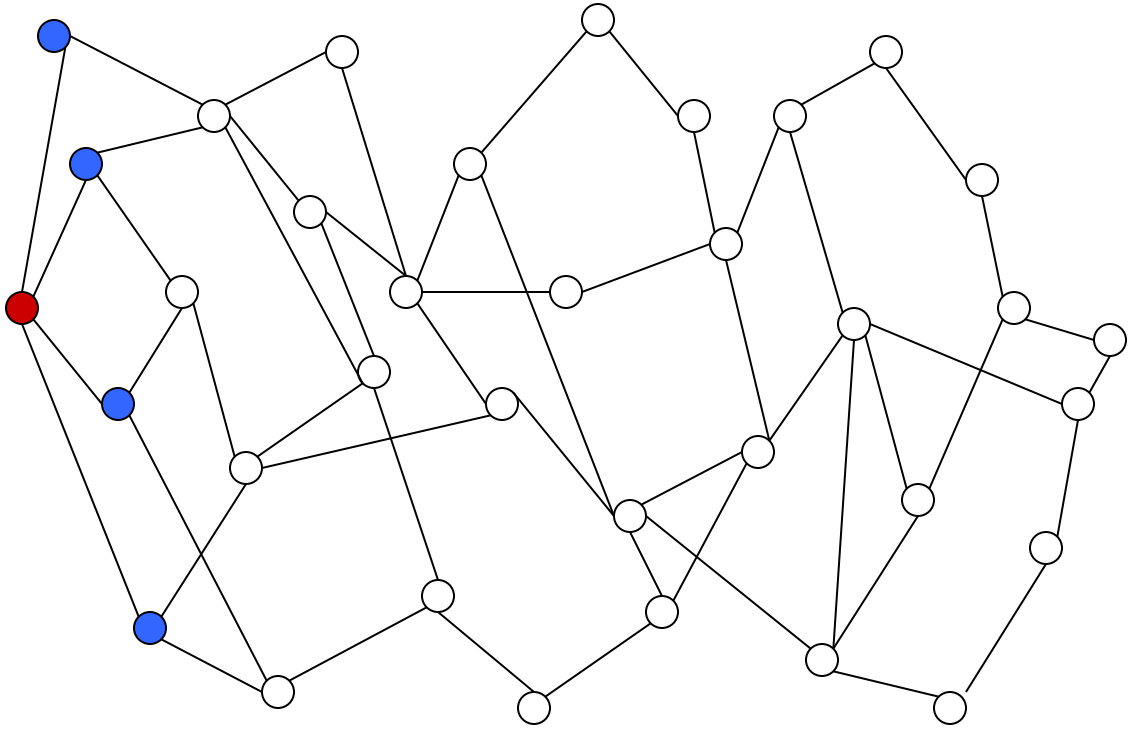
- Bei anderen Problemstellungen wird eine optimale Lösung erreicht.
- Beispiel:
  - **Problemstellung: Auf Geldbeträge < 1 Euro soll Wechselgeld herausgegeben werden.**
  - **Lösung mit Greedy: Solange noch ein Restbetrag da ist, gib die größte Münze zurück, die kleiner als der Restbetrag ist.**
- Hier kommt der Algorithmus zum optimalen Ergebnis.
- Bei einem Greedy-Algorithmus ist zu prüfen, ob:
  - **immer das optimale Ergebnis erreicht wird.**
  - **ein suboptimales Ergebnis für die Aufgabenstellung ausreicht.**

- Andere Namen: divide-et-impera, teile-und-herrsche

- allgemeiner Aufbau:
  - Problem in mehrere kleinere Teilprobleme derselben Art aufteilen (divide).
  - Dann: kleinere Teilprobleme (rekursiv) lösen.
  - Zuletzt aus Lösungen der Teilprobleme eine Lösung für Gesamtproblem konstruieren (conquer).

Lösung mit Divide and Conquer

- Zerlege in Teilprobleme:
  - **Versuche, von den blauen Punkten mit 4 Schritten so weit wie möglich nach rechts zu kommen (⇒ Rekursion).**
  - **Nimm davon das Maximum.**



Anderes Beispiel: QuickSort

- Zu sortierende Liste in zwei Teile (untere / obere Teilliste) aufteilen: Aufteilung geschieht so, dass
  - in unterer Teilliste nur Elemente $\leq$ einem Referenz-Element,
  - in oberer Teilliste nur Elemente $\geq$ Referenz-Element vorkommen

- So entstandene Teillisten werden nach dem gleichen Prinzip "sortiert" (rekursive Vorgehensweise).

Verfahren endet, wenn Teillisten weniger als zwei Elemente enthalten.

Hauptbausteine dieser Technik:

- *Branching (Verzweigung):*
  Aufgabe in Teilaufgaben zerlegen.
  Diese wieder zerlegen $\Rightarrow$ Baum von Teilaufgaben

- *Bounding (Beschränkung):*
  Wenn die optimale Lösung einer Teilaufgabe nicht besser als eine schon bekannte Lösung sein kann: Diese Teilaufgabe nicht weiter betrachten.
  Um dies festzustellen, schätzt man den höchstens erreichbaren Gewinn für die betrachtete Teilaufgabe ab. Ein solcher Schätzwert heißt *obere Schranke*.

Branch-and-Bound

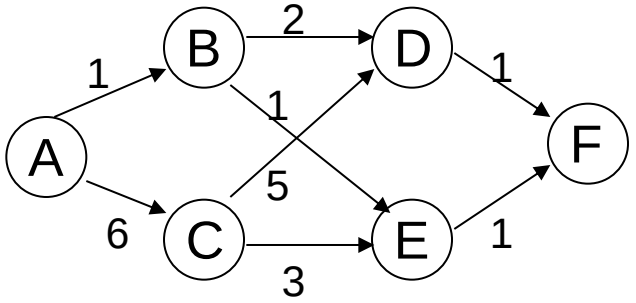
- Zerlege mögliche Fälle in disjunkte Klassen
- Bestimme Schranken für Teilbereiche, welche ganze Fälle ausschließen:
  - obere Schranken bei Minimierungsproblemen
  - untere Schranken bei Maximierungsproblemen

- Beispiel: minimaler Weg in einem Graphen

  Wenn Kosten eines Teilweges bereits größer sind als Kosten eines bereits bekannten kompletten Weges:
  alle Wege, die diesen Teilweg enthalten, nicht mehr betrachten

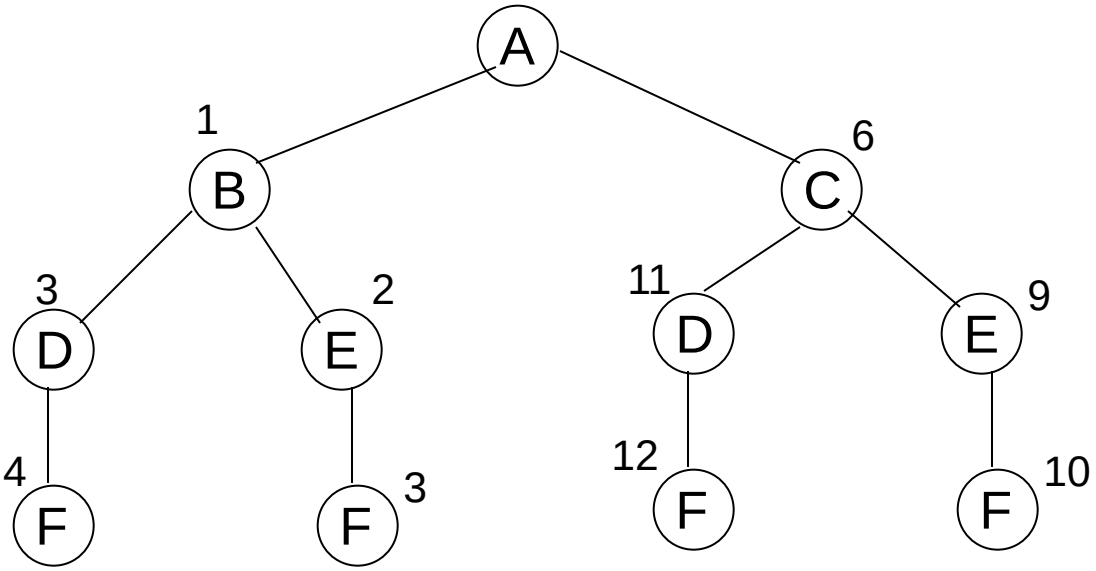
Beispiel zu Branch-and-Bound

Suche kürzesten Weg von A nach F



- Disjunkte Klassen: mögliche Wege, die man nehmen kann:
  1. Wege, die mit B beginnen (A-B-D-F, A-B-E-F)
  2. Wege, die mit C beginnen (A-C-E-F, A-C-D-F)
- Schranken: Kosten für Teilweg von A bis zum jeweiligen Knoten. Falls Kosten auf betrachtetem Weg von A zu einem Knoten X höher als für bereits bekannten Weg: Suche auf aktuell untersuchtem Weg abbrechen
- z.B. Weg A-B-E-F bereits bekannt mit Kosten 3:
  Da bereits A-C mit 6 teurer ist, braucht hier nicht weiter gerechnet werden

Suchbaum zu Branch-and- Bound-Bsp.



- Generiere alle möglichen Wege
- Breche Suche auf einem Weg ab, wenn Kosten für Teilweg bereits höher sind als eine bekannte Lösung

Bemerkungen zu Branch-and-Bound

- Anwendungsgebiete:
  - Spielbaumsuche
  - Optimierungsprobleme
  - im allgemeinen: sehr aufwendige Probleme (NP-vollständige Probleme)

- Wichtig: möglichst schnell gute Schranke finden, um so große Teile des Suchraumes auszuschließen