

# Konzepte der Informatik

## Algorithmik Darstellung und Datenstrukturen

Barbara Pampel

Universität Konstanz, WiSe 2023/2024

---

# Inhalt

1 Begriffsbestimmung

2 Eigenschaften

3 Darstellung

4 Datenstrukturen

## Abgrenzung

ALGORITHMUS  
HANDLUNGSAUWEISUNG  
PROGRAMM

PROGRAMMERSPRACHE

# Eigenschaften

**Warum es manchmal schwer ist,  
ein Informatiker zu sein:**



Meine Mutter sagte:  
"Sei bitte so lieb und geh für mich zum Supermarkt. Kauf  
eine Flasche Milch und wenn sie Eier haben, bring sechs mit."

Ich kam mit sechs Flaschen Milch wieder.

Sie schimpft:  
"Warum zum Teufel, hast Du sechs Flaschen Milch  
gekauft?"

Meine Antwort:  
"Weil sie Eier hatten!"

## Spezifikation

- Eingabespezifikation
- Ausgabespezifikation

WAS eingegeben  
werden muss, damit  
DAS ausgegeben/getan wird.

---

**Algorithm 1:** wiederholte Minimumssuche

---

**Input:** int array  $M$  of size  $n$   
**Data:** int  $i, j$  and  $m$   
**Output:**  $k^{th}$  smallest element in  $M$

```
int  $k^{th}$  (array  $M$ ) begin
    for  $i = 1, \dots, k$  do
         $m \leftarrow i$ 
        for  $j = i + 1, \dots, n$  do
            if  $M[j] < M[m]$  then  $m \leftarrow j$ 
        swap  $M[i]$  and  $M[m]$ 
    print  $M[k]$ 
```

---

## Übertragbarkeit

- Parametrisierbarkeit

Verschiedene **Instanzen** eines Problems können eingegeben werden.

---

**Algorithm 2:** wiederholte Minimumssuche

---

**Input:** int array  $M$  of size  $n$   
**Data:** int  $i, j$  and  $m$   
**Output:**  $k^{th}$  smallest element in  $M$

```
int  $k^{th}$  (array  $M$ ) begin
    for  $i = 1, \dots, k$  do
         $m \leftarrow i$ 
        for  $j = i + 1, \dots, n$  do
            if  $M[j] < M[m]$  then  $m \leftarrow j$ 
        swap  $M[i]$  and  $M[m]$ 
    print  $M[k]$ 
```

---

## Beschreibung

- Finitheit
- Diskretheit

Endliche Folge von  
einzelnen Anweisungen.

---

**Algorithm 3:** wiederholte Minimumssuche

---

**Input:** int array  $M$  of size  $n$   
**Data:** int  $i, j$  and  $m$   
**Output:**  $k^{th}$  smallest element in  $M$

```
1 int  $k^{th}$  (array  $M$ ) begin
2   for  $i = 1, \dots, k$  do
3      $m \leftarrow i$ 
4     for  $j = i + 1, \dots, n$  do
5       if  $M[j] < M[m]$  then  $m \leftarrow j$ 
6       swap  $M[i]$  and  $M[m]$ 
7   print  $M[k]$ 
```

---

## Ausgabe und Ablauf

- Determiniertheit
- Determinismus

Gleiche Ausgabe  
bei gleicher Eingabe.

Gleicher Ablauf.

---

**Algorithm 4:** wiederholte Minimumssuche

---

**Input:** int array  $M$  of size  $n$   
**Data:** int  $i, j$  and  $m$   
**Output:**  $k^{\text{th}}$  smallest element in  $M$

```
int  $k^{\text{th}}$  (array  $M$ ) begin
    for  $i = 1, \dots, k$  do
         $m \leftarrow i$ 
        for  $j = i + 1, \dots, n$  do
            if  $M[j] < M[m]$  then  $m \leftarrow j$ 
        swap  $M[i]$  and  $M[m]$ 
    print  $M[k]$ 
```

---

## Korrektheit

- Partielle Korrektheit  
Korrekte Einzelschritte
- Terminieren  
Algorithmus terminiert bei korrekter Eingabe.

Wir werden hier nur kurz ins Thema Korrektheit einführen und konkrete Beweise später an Beispielen führen.

## Korrektheit

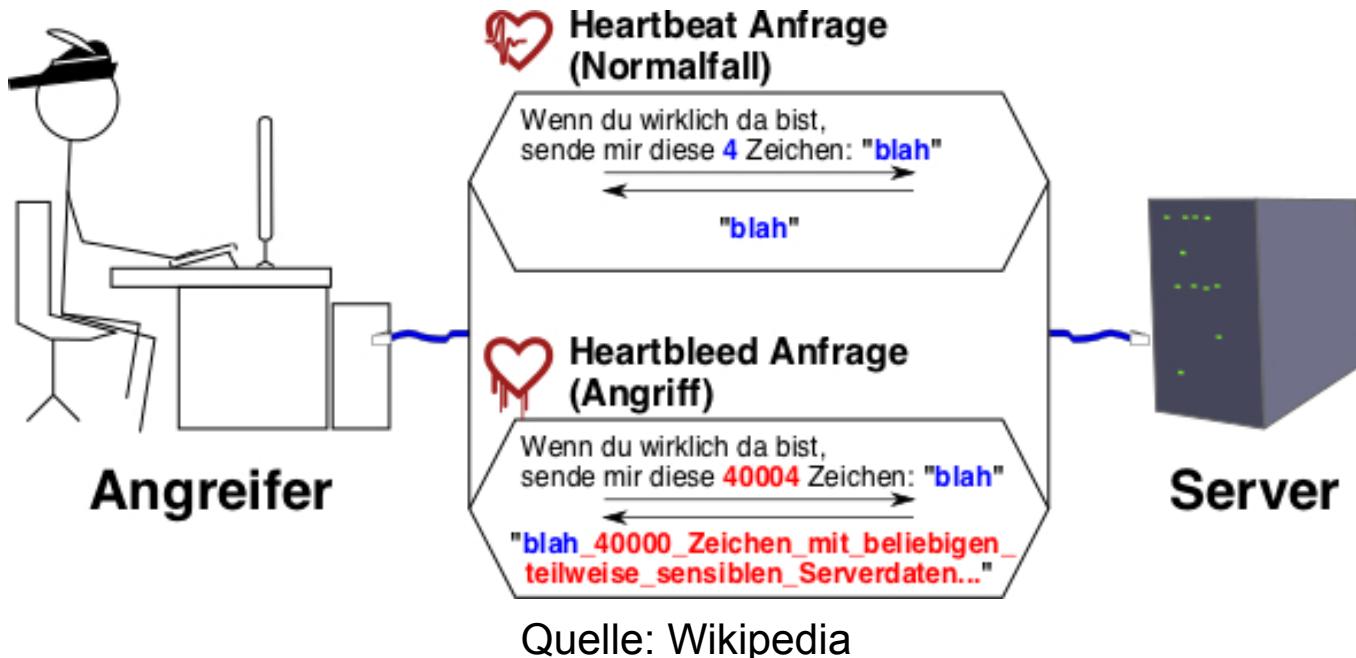
- 22. Juli 1962: NASA Mariner 1 stürzt ab
  - Überstrich in einer Formel beim Abschreiben der Spezifikation vergessen
  - Spezifikation zwar korrekt implementiert, aber dennoch falsches Verhalten
  - während des Starts verliert die Rakete einen Booster
- 4. Juni 1996: Ariane V88 stürzt beim Start ab
  - Überlauf bei Umwandlung einer Gleitkommazahl in eine Ganzzahl
  - wurde bis an oberste Ebene propagiert, das Steuerungsprogramm hat sich beendet
- Das Jahr-2000-Problem
  - viele Programme, auch wichtige Steuerungssysteme, speicherten das Jahr nur zweistellig
  - es war unklar, was beim Übergang von 99 auf 0 passieren würde
  - Schaden durch die Hysterie war viel größer als der durch das eigentliche Problem

## Korrektheit

- 1980er Jahre: Therac-25 Linearbeschleuniger zur Strahlentherapie
  - Fehler in der Steuerungssoftware kostet 3 Patienten das Leben
  - Messwerterfassung und Steuerung erfolgen nebenläufig, waren aber ungenügend synchronisiert
  - Probleme traten erst auf, nachdem der Bediener schneller mit dem Gerät arbeiten konnte
  - Programmierer waren auch für Tests zuständig
- Stromausfall in ganz Nordamerika
  - verursacht durch einen lokalen Stromausfall auf Grund eines nicht synchronisierten kritischen Abschnitts in einem Steuerungsprogramms

## Korrektheit

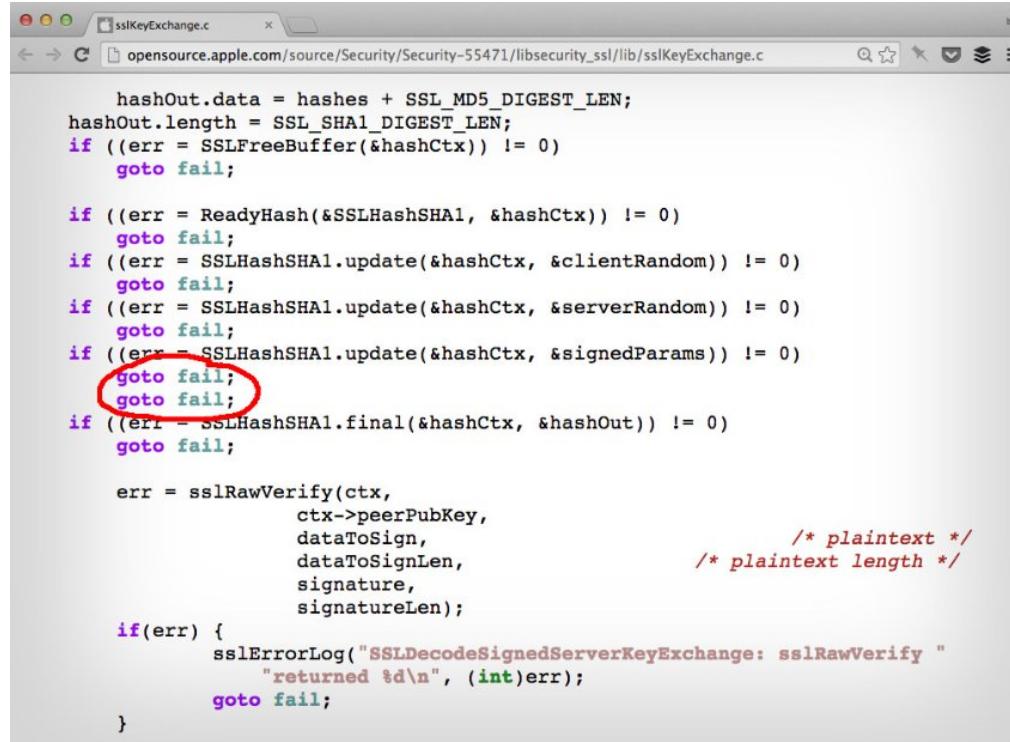
- Heartbleed



## 2 Eigenschaften - 2.5 Korrektheit

### Korrektheit

- goto fail



```
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(ctx,
                    ctx->peerPubKey,
                    dataToSign,
                    dataToSignLen,
                    signature,
                    signatureLen);
if(err) {
    sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                "returned %d\n", (int)err);
    goto fail;
}
```

Quelle: Spiegel

## Korrektheit

- Shellshock
  - Sicherheitslücke in Linux-Bash-Shell
  - betroffen sind (je nach Version) Linux und MacOS-Systeme, evtl. auch Windows mit uwin
  - Fehler existiert seit 1989
  - wurde im September 2014 von Stéphane Chazelas entdeckt

### Testcode

```
env x='() { :;}; echo vulnerable' bash -c 'echo hello'
```

- Wird 'vulnerable' und 'hello' angezeigt, ist man betroffen.
- Wird nur "hello" angezeigt, ist man NICHT (mehr) betroffen.

## Verifikationsmethode nach Floyd

Methode von Robert Floyd (1967) zur Verifikation von iterativen Programmen

- Finde eine geeignete Formel  $\mathcal{INV}$  und zeige, dass sie eine Schleifeninvariante ist
  - muss zunächst innerhalb der Schleife gelten
- Zeige, dass aus der Eingabespezifikation folgt, dass  $\mathcal{INV}$  auch vor dem ersten Schleifendurchgang gültig ist
- Zeige, dass nach dem letzten Schleifendurchgang aus  $\mathcal{INV}$  und aus der Negation Schleifenbedingung  $b$ , die Gültigkeit aus der Ausgabespezifikation folgt
- Zeige, dass die Schleife terminiert

## Komplexität

- Zeitkomplexität
  - (Operations-)Aufwand, den der konkrete Algorithmus braucht
  - abhängig von der Implementierung
- Speicherkomplexität
  - benötigter Speicherplatz während der Berechnung

Wir werden zunächst nur recht informell den Aufwand und des Speicherbedarf eines Algorithmus ermitteln, später dann formal fassen.

## Das Wichtigste in Kürze

- Eigenschaften
  - Ein- und Ausgabespezifikation
  - Parametrisierbarkeit
  - Finitheit und Diskretheit
  - Determiniertheit und Determinismus
  - Korrektheit
  - Komplexität

## Inhalt

1 Begriffsbestimmung

2 Eigenschaften

**3 Darstellung**

4 Datenstrukturen

## Pseudocode

Beschreibung aus natürlicher Sprache und mathematischer Notation. Nicht zur maschinellen Interpretation, sondern zur Veranschaulichung.

---

**Algorithm 5:**

**Input:** Eingabe

**Data:** Hilfsvariablen

**Output:** Ausgabe

**begin**

Vorbereitung

**while** *Nicht Trivialfall* **do**

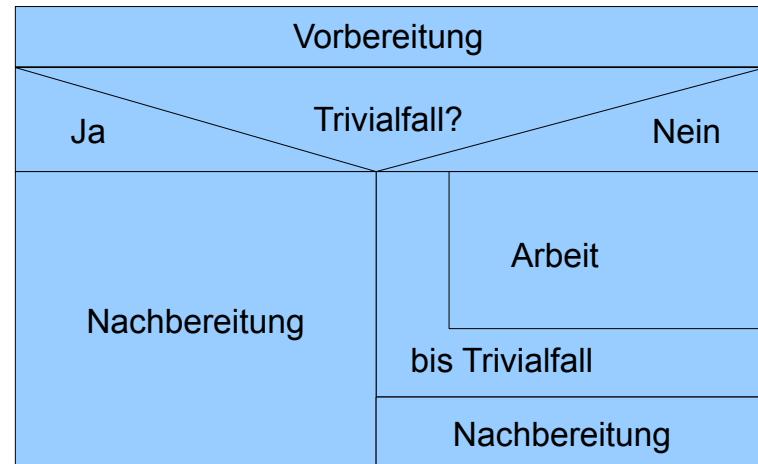
  Arbeit

  Nachbereitung

---

## Struktogramm

- Graphische, programmiersprachenunabhängige Darstellung von Algorithmen
- Zerlegung elementare Grundstrukturen wie Sequenzen und Kontrollstrukturen ⇒ Strukturblöcke
- Auch **Nassi-Shneidermann-Diagramme**



# Konstruktionselemente I

- Anweisungen

Betrachte den unsortierten Teil des Feldes

- Methoden, Unterprogramme, Funktionen

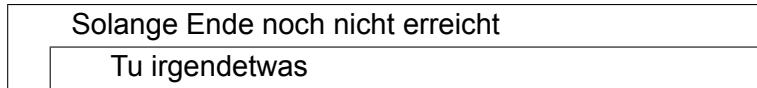
Kleinstes Element suchen

- Rückgabewert von Methoden

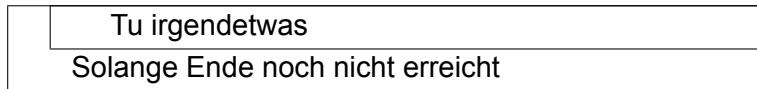
Kleinstes Element zurückgeben

## Konstruktionselemente II

- while-Schleifen



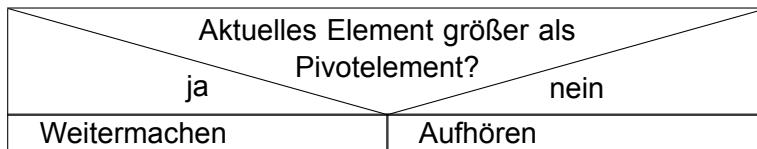
- do-Schleifen



- Sprung/Schleife beenden

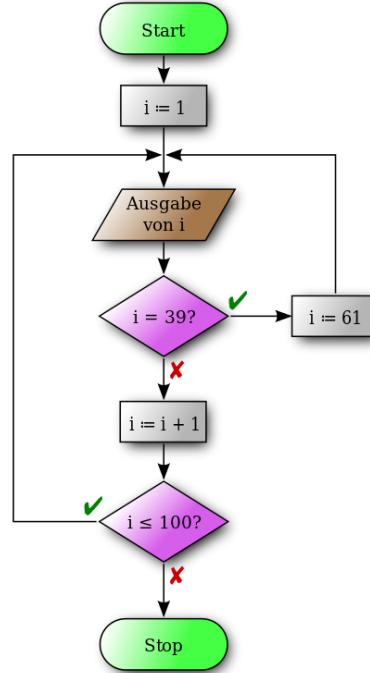


- Bedingte Ausführung



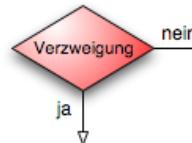
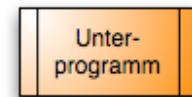
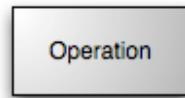
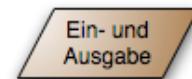
## Flussdiagramm

- graphische Darstellung des Ablaufes mit Hilfe von Symbolen und Übergängen
- Symbole für Programmablaufpläne sind in der DIN 66001 genormt



## Konstruktionselemente

- Programmstart bzw. Programmende
- Daten
- Ausführen von Operationen
- Ausführen eines Unterprogramms
- Bedingte Verzweigungen



## Das Wichtigste in Kürze

- Eigenschaften
  - Ein- und Ausgabespezifikation
  - Parametrisierbarkeit
  - Finitheit und Diskretheit
  - Determiniertheit und Determinismus
  - Korrektheit
  - Komplexität
- Darstellung
  - Pseudocode
  - Struktogramme
  - Flussdiagramme

## Inhalt

1 Begriffsbestimmung

2 Eigenschaften

3 Darstellung

**4 Datenstrukturen**

## Abstrakte Datenstrukturen

- Daten zweckdienlich angeordnet, kodiert und miteinander verknüpft
- Verwaltung von bzw. dem Zugriff auf die Daten in geeigneter Weise
- Spezifikation ist unabhängig von ihrer Implementierung und der konkreten Verwendung

ARRGH! MY MAP OF LISTS OF MAPS  
TO STRINGS IS TOO HARD TO  
ITERATE THROUGH! I'LL JUST ASSIGN  
EVERYTHING A NUMBER AND USE  
A !@#\*!@ ARRAY



# Arrays

## Definition

Namentliche Zusammenfassung von gleichartigen Objekten eines Datentyps.

1	2	3	4	5	6	7	8	9	10	11	12	13

## Methoden

### Integer-Array der Länge 10

```
int [] array = new int [10];
```

Direktes Abbild des Speichersystems. Ein Arrayzugriff wird in nur wenige Maschinenanweisungen übersetzt ⇒ Programme mit Arrays werden in effiziente Maschinenprogramme übersetzt.

Einfügen / Ändern eines Elements durch normale Zuweisung:

```
array[i] = 5;
```

Ausgeben als Zugriff:

```
j = array[i];
```

## Beispiel - Problemstellung

Finden von Primzahlen mit Primzahlensieb des Eratosthenes

- benannt nach Eratosthenes (ca. 200 v. Chr.)
- ist der älteste und bekannteste Siebalgorithmus

## Beispiel - Problemstellung

Finden von Primzahlen mit Primzahlensieb des Eratosthenes

- benannt nach Eratosthenes (ca. 200 v. Chr.)
- ist der älteste und bekannteste Siebalgorithmus

Prinzip von Siebalgorithmen:

- Aufteilen in *gute* und *schlechte* Elemente
- sukzessives eliminieren von schlechten Elementen (leichter zu finden)
- übrig bleiben die guten

Hier:

- Primzahlen als gute
- nicht-prime Zahlen als schlechte Elemente

## Beispiel - Lösung

---

**Algorithm 6:** Sieb des Eratosthenes

---

**Input:** Array A der Länge  $n$   
**Output:** alle Primzahlen  $\leq n$

**begin**

$j = 2$

**while**  $j \leq \sqrt{n}$  **do**

**for**  $k = j, \dots, \lfloor n/j \rfloor$  **do**

$A[k \cdot j] = 0$

$j = j + 1$

**while**  $A[j] == 0$  **do**

$j = j + 1;$

                // Vielfache von  $j$  aussieben

                // kleinste verbleibende Zahl prim

## Beispiel - $n = 120$

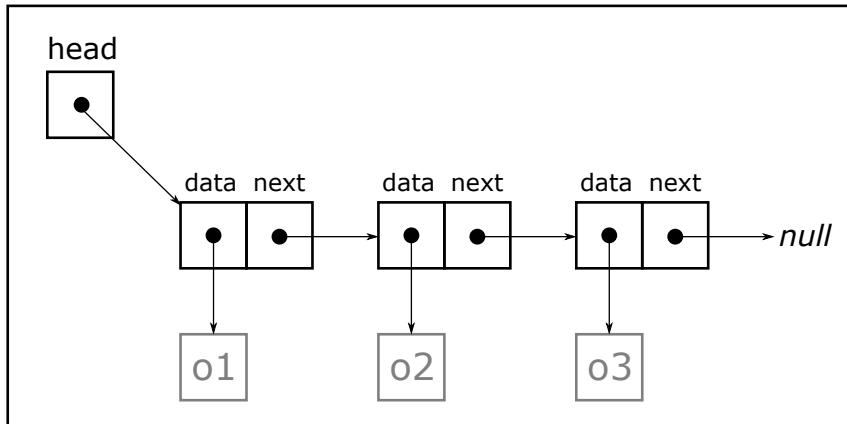
Quelle: Wikipedia

## Verkettete Listen (linked list)

### Definition

Folge von Objekten, durch Zeiger miteinander verknüpft. Dynamisch, da sie während des Programmverlaufs ihre innere Struktur ändern und verlängert oder verkürzt werden kann.

LinkedList



### Java-Klasse

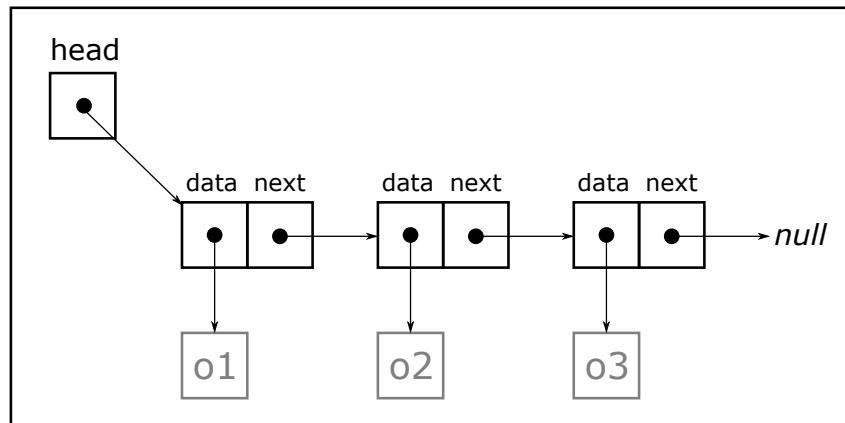
```
public class LinkedList<T> {  
    static class ListElement<T> {...}  
    private ListElement<T> head;  
}
```

## Erzeugen

Erzeugen eines Listenobjektes mit

- Referenz auf tatsächlich zu speicherndes Objekt
- Referenz auf Nachfolger in der Liste

LinkedList



### in Java

```
class ListElement<T> {  
    ListElement<T> next;  
    T data;  
}
```

## Erzeugen

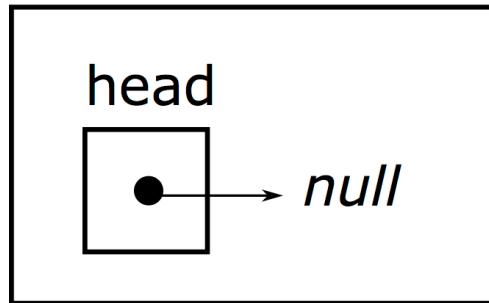
Erzeugen eines Listenobjektes mit

- Referenz auf tatsächlich zu speicherndes Objekt
- Referenz auf Nachfolger in der Liste

Erzeugen einer leeren Liste als

- Referenz auf erstes Listenobjekt
- `head = null`  $\Rightarrow$  Liste ist leer

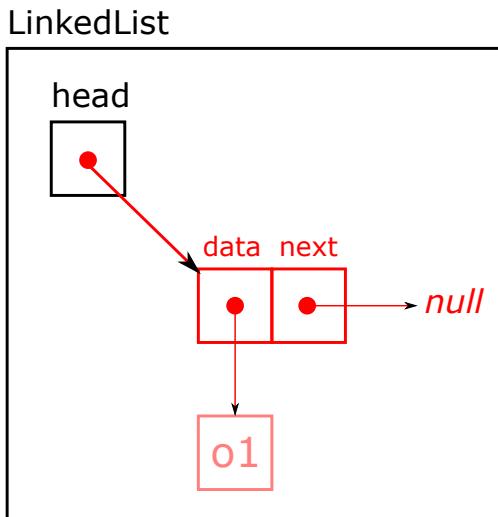
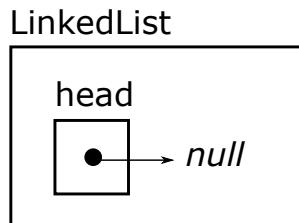
### LinkedList



```
LinkedList() {  
    head = null;  
}
```

## Einfügen - leere Liste

- Erzeugen eines neuen Listenelements
- Kopreferenz auf das neue Listenelement setzen

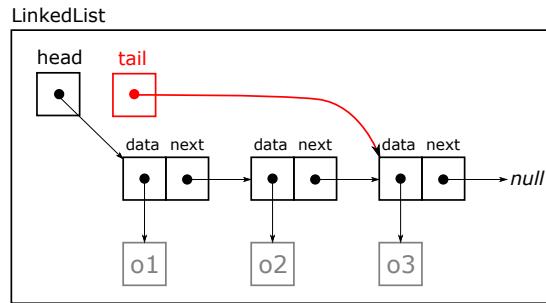


## Einfügen – am Ende

- Neues Listenelement erzeugen
- Ende der Liste finden
- Referenz auf nächstes Element des noch-letzten Elements auf neues Listenelement setzen
- Problem: wie findet man das Ende der Liste?

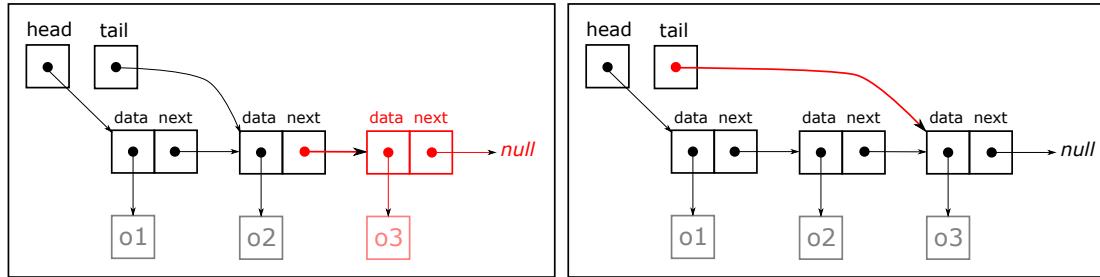
## Einfügen – am Ende

- Neues Listenelement erzeugen
- Ende der Liste finden
- Referenz auf nächstes Element des noch-letzten Elements auf neues Listenelement setzen
- Problem: wie findet man das Ende der Liste?
  - zweite Referenz in der Liste, die immer auf das aktuelle Ende zeigt



## Einfügen – am Ende

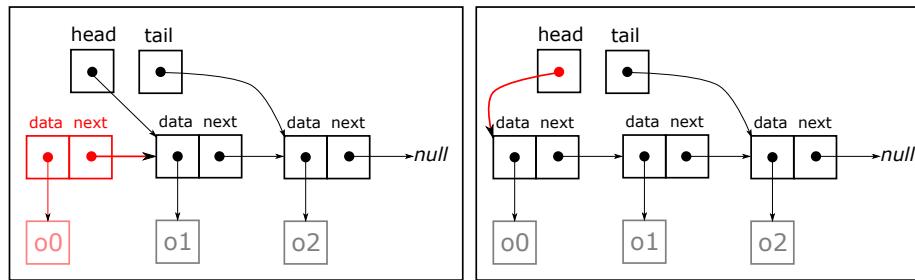
- Neues Listenelement erzeugen
- Referenz auf nächstes Element des noch-letzten Elements auf neues Listenelement setzen
- Sonderfall leere Liste beachten



## Einfügen - am Anfang

## Einfügen - am Anfang

- Neues Listenelement erzeugen
- Referenz auf nächstes Element ist die Kopfreferenz
- Kopfreferenz auf neues Listenelement setzen

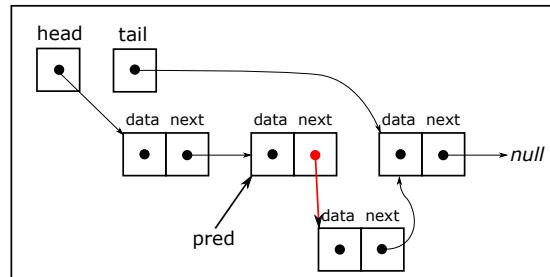
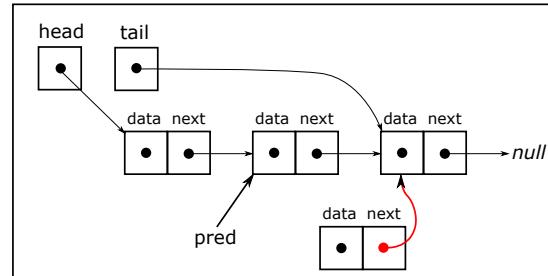
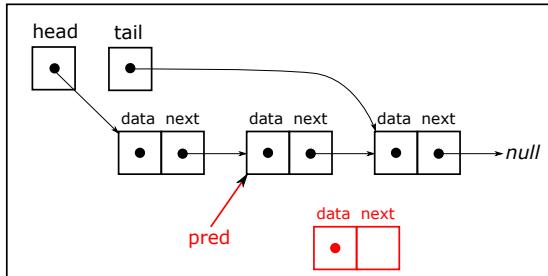


## Einfügen – an beliebiger Position

## Einfügen – an beliebiger Position

- Neues Listenelement erzeugen
- Position suchen (Durchlaufen der Liste)
- Nachfolgerreferenz des neuen Elements auf Nachfolger des Vorgängers setzen
- Nachfolgerreferenz des Vorgängers auf neues Element setzen
- Achtung auf Spezialfälle Anfang und Ende der Liste

## Einfügen – an beliebiger Position

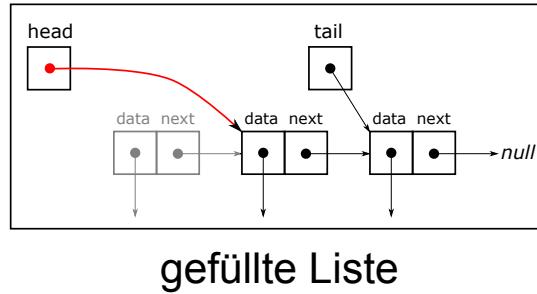


## Löschen – am Anfang

- Kopfreferenz ein Element weitersetzen

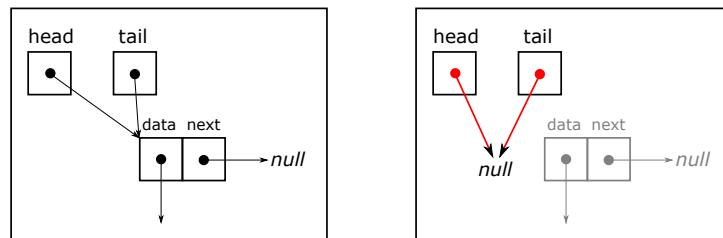
## Löschen – am Anfang

- Kopfreferenz ein Element weitersetzen
- Vorsicht bei leerer Liste oder nur einem Element
- Konvention: beim Entfernen wird das entfernte Objekt zurückgegeben



## Löschen – am Anfang

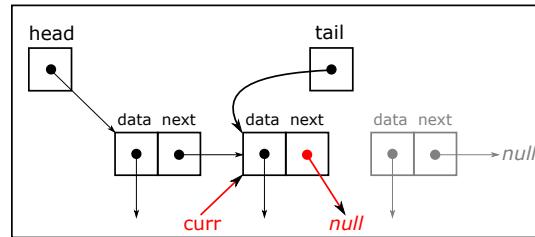
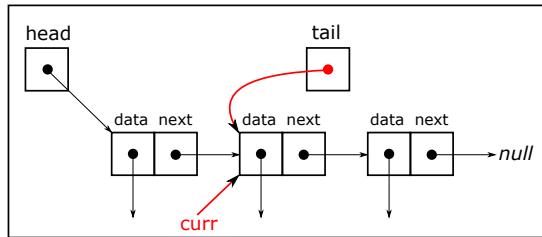
- Kopfreferenz ein Element weitersetzen
- Vorsicht bei leerer Liste oder nur einem Element
- Konvention: beim Entfernen wird das entfernte Objekt zurückgegeben



einelementige oder leere Liste

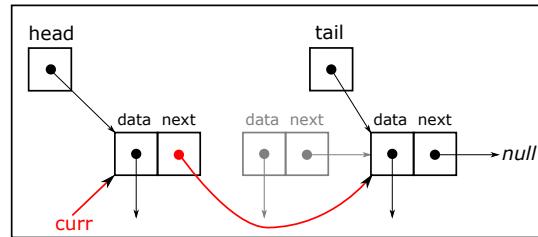
## Löschen – am Ende

- Komplizierter, da Vorgänger des letzten Elements nicht bekannt
- Suche nach vorletztem Element und Setzen der Endreferenz
- Nachfolgerreferenz auf `null` setzen



## Löschen – an beliebiger Position

- Suche nach dem Vorgänger des zu löschenen Elements
- Umsetzen der Nachfolgerreferenz
- Achtung falls das zu löschenes Element Kopf oder Ende der Liste ist

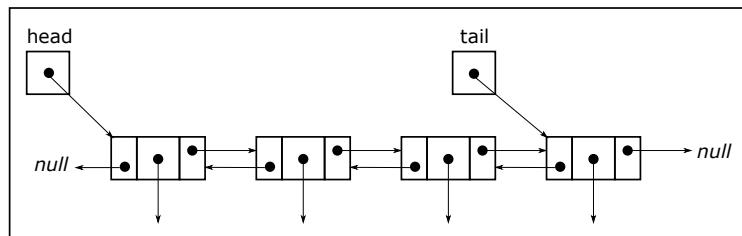


## Doppelt verkettete Listen

- Entfernen des letzten Elements bei einfach verketteter Liste immer noch aufwändig
  - vorletztes Element muss gefunden werden
- Iterieren über Listenelemente in umgekehrter Reihenfolge extrem teuer

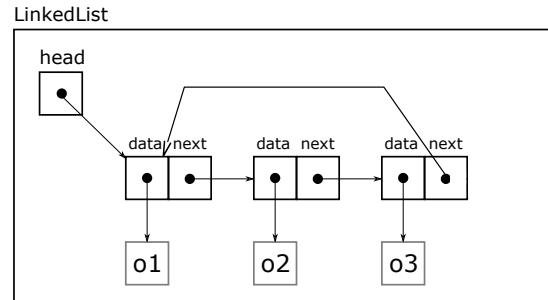
## Doppelt verkettete Listen

- Entfernen des letzten Elements bei einfach verketteter Liste immer noch aufwändig
  - vorletztes Element muss gefunden werden
- Iterieren über Listenelemente in umgekehrter Reihenfolge extrem teuer
- Lösung: doppelt verkettete Liste
  - jedes Listenelement hat zusätzlich einen Referenz auf seinen Vorgänger



## Ringliste

- zusätzliche Referenz vom letzten zum ersten Element



## Beispiel - Problemstellung

- Flavius Josephus

$n$  Personen begehen kollektiven Selbstmord, indem sie sich in einem Kreis aufstellen und jede  $m$ -te Person sich tötet. Nach dem Ausscheiden einer Person wird der Kreis wieder geschlossen. Die letzte Person möchte Josephus sein.



Quelle: Wikipedia

## Beispiel - Lösung

---

Algorithm 7: Josephus-Problem

---

**Input:** Ringliste  $L$  der Länge  $n$ , Schrittlänge  $m$

**Output:** zuletzt bleibende Position  $last$

**begin**

```
ListElement candidate = head
for i = 1 ... n - 1 do                                // lösche n - 1 Elemente
    for k = 1 ... m - 1 do                            // noch m - 1 Schritte
        candidate = candidate.next
        cNeighbour = candidate.next                    // da candidate gelöscht wird
        list.delete(candidate)
        candidate = cNeighbour                         // 1. Schritt
    last = cNeighbour                                // verbleibende Position
```

---

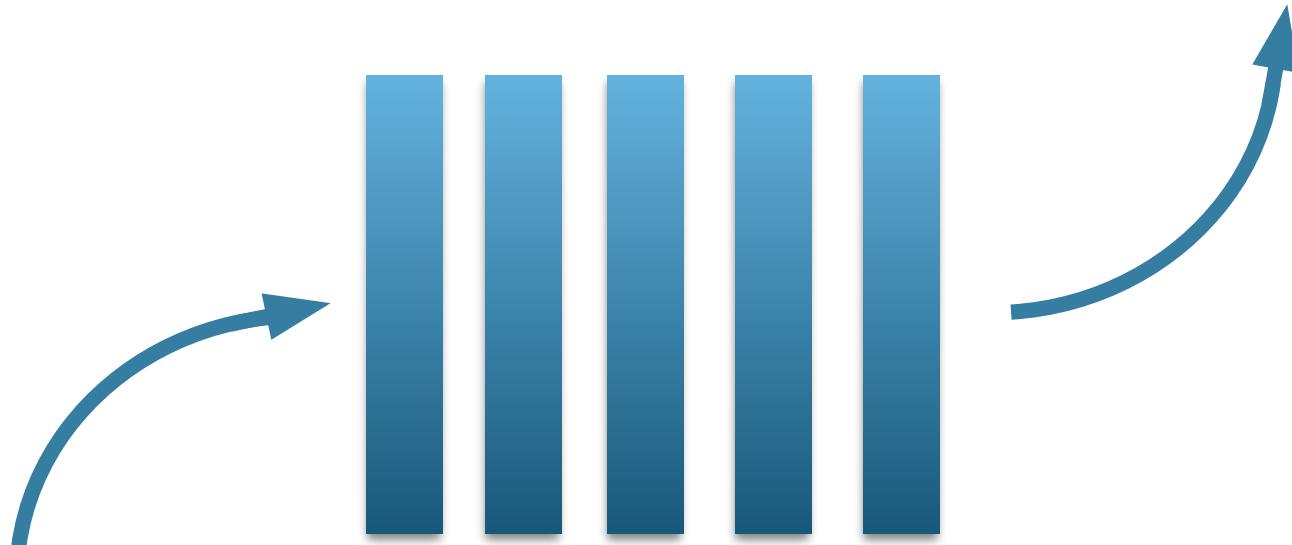
## Arrays vs. Verkettete Listen

- Primzahlensieb von Eratosthenes:  
Array vorteilhaft, weil die Effizienz des Algorithmus darauf beruht, schnell auf eine beliebige Arrayposition zugreifen zu können
- Josephus-Problem:  
Verkettete Listenvorteilhaft, weil die Effizienz des Algorithmus darauf beruht, schnell Elemente entfernen zu können. Achtung: Verkettete Listen könnten durchaus intern mit Arrays realisiert werden!

## Warteschlangen

## Warteschlangen

- kann "beliebige" Menge von Objekten aufnehmen
- gibt diese in der Reihenfolge ihres Einfügens wieder zurück  
(First In – First Out-Prinzip, kurz FIFO)



## Operationen

Java	Pseudocode	Operation
Queue()	new queue	eine neue, leere Warteschlange anlegen
add( <i>e</i> ) or offer( <i>e</i> )	enqueue	eine neues Element <i>e</i> in die Warteschlange einfügen
remove() or poll()	dequeue	das älteste Element aus der Warteschlange entfernen und ausgeben
element() or peek()	peek	ältestes Element anzeigen, ohne es zu entfernen
	isEmpty	gibt <i>true</i> aus, falls leer, sonst <i>false</i>

## Implementierung als Liste

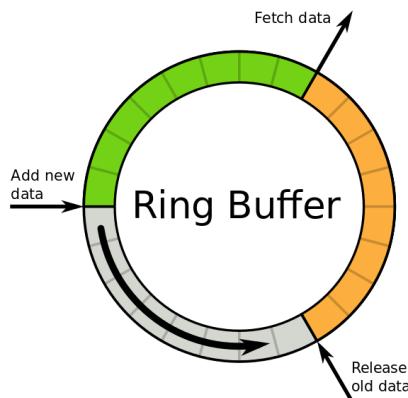
## Implementierung als Liste

- Einfügen eines Elements am Ende der Liste und Entnahme am Kopf sichern FIFO
- maximale Länge muss nicht bekannt sein

## Implementierung als Array

## Implementierung als Array

- bietet sich an, falls Maximallänge bekannt ist
- zirkuläres Array: *Ringpuffer*
- *write* und *read pointer* zeigen Anfang und Ende an
- Einfügen nur am write pointer und Entnahme nur am read pointer und sichern dadurch FIFO
- braucht zusätzliche Fehlermeldungen für volles Array



## Stapel (stacks)

## Stapel (stacks)

- kann beliebige Menge von Objekten aufnehmen
- gibt diese in der dem Einfügen entgegen gesetzten Reihenfolge zurück  
(Last In – First Out-Prinzip, kurz LIFO)



## Operationen

Java	Pseudocode	Operation
Stack()	new stack	einen neuen, leeren Stapel anlegen
push( <i>e</i> )	push( <i>e</i> )	eine neues Element <i>e</i> auf den Stapel legen
pop()	pop	das jüngstes Element vom Stapel nehmen und ausgeben
peek()	peek	jüngstes Element anzeigen, ohne es zu entfernen
empty()	isEmpty	gibt <i>true</i> aus, falls leer, sonst <i>false</i>

## Implementierung als Liste

## Implementierung als Liste

- Einfügen eines Elements am Kopf einer Liste
- Entnahme ebenfalls am Kopf sichert LIFO
- maximale Stackhöhe muss nicht bekannt sein

## Implementierung als Array

## Implementierung als Array

- bietet sich an, falls Maximalhöhe bekannt ist
- Array wird von "vorn" nach "hinten" gefüllt
- *stack pointer* zeigt Stabelspitze an
- Einfügen und Entnahme am stack pointer sicher LIFO
- braucht zusätzliche Fehlermeldungen für volles Array

## Beispiel - Problemstellung

Korrektheit eines Klammerausdrucks:  
für jede öffnende existiert **genau** eine schließende Klammer

korrekt: (((()((()))))  
nicht korrekt: ((())()

- Eingabe: Ein (nichtleerer) Klammerausdruck mit öffnenden und schließenden Klammern
- Ausgabe: *true* falls Klammerausdruck korrekt, *false* falls Klammerausdruck nicht korrekt

## Beispiel - Lösung

---

### Algorithm 8: Korrektheit eines Klammerausdrucks

---

**Input:** Klammerausdruck K als Array

**Data:** Stack S

**Output:** boolean für Korrektheit

**begin**

```
  for  $i = 1 \dots K.length$  do
    if  $K[i]$  ist öffnende Klammer then
       $S.push(K[i])$ 
    if  $K[i]$  ist schließende Klammer then
      if  $S.isEmpty()$  then
        return false
       $x = S.pop()$ 
  return true
```

---

## Beispiel - Lösung

---

### Algorithm 9: Korrektheit eines Klammerausdrucks

---

**Input:** Klammerausdruck K als Array

**Data:** Stack S

**Output:** boolean für Korrektheit

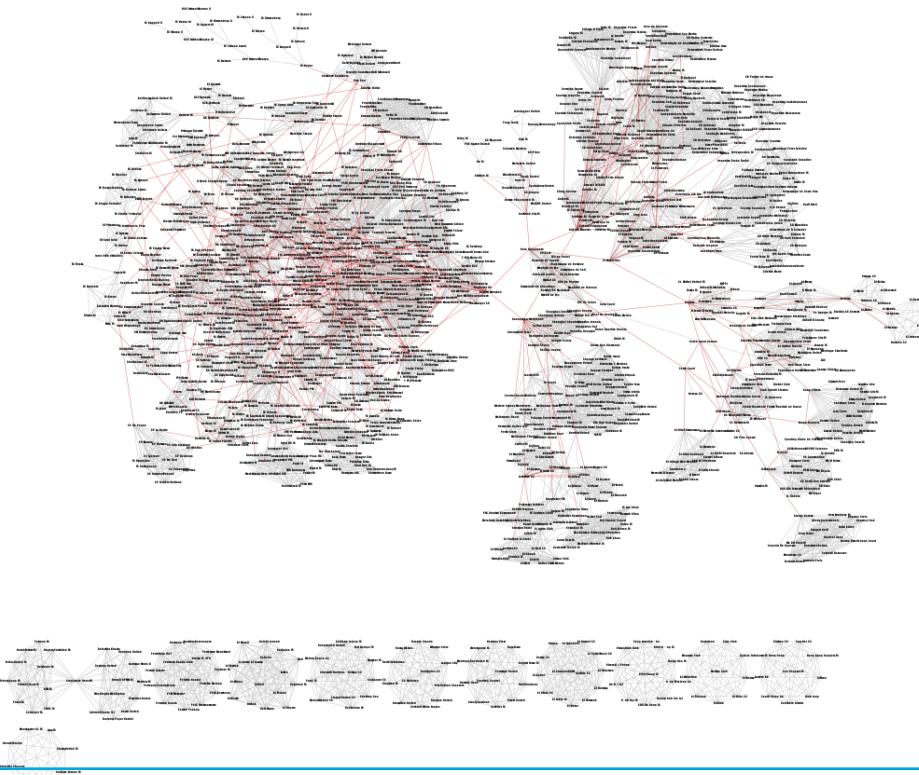
**begin**

```
for i = 1 . . . K.length do
    if K[i] ist öffnende Klammer then
        S.push(K[i])
    if K[i] ist schließende Klammer then
        if S.isEmpty() then
            return false
        x = S.pop()
return true
```

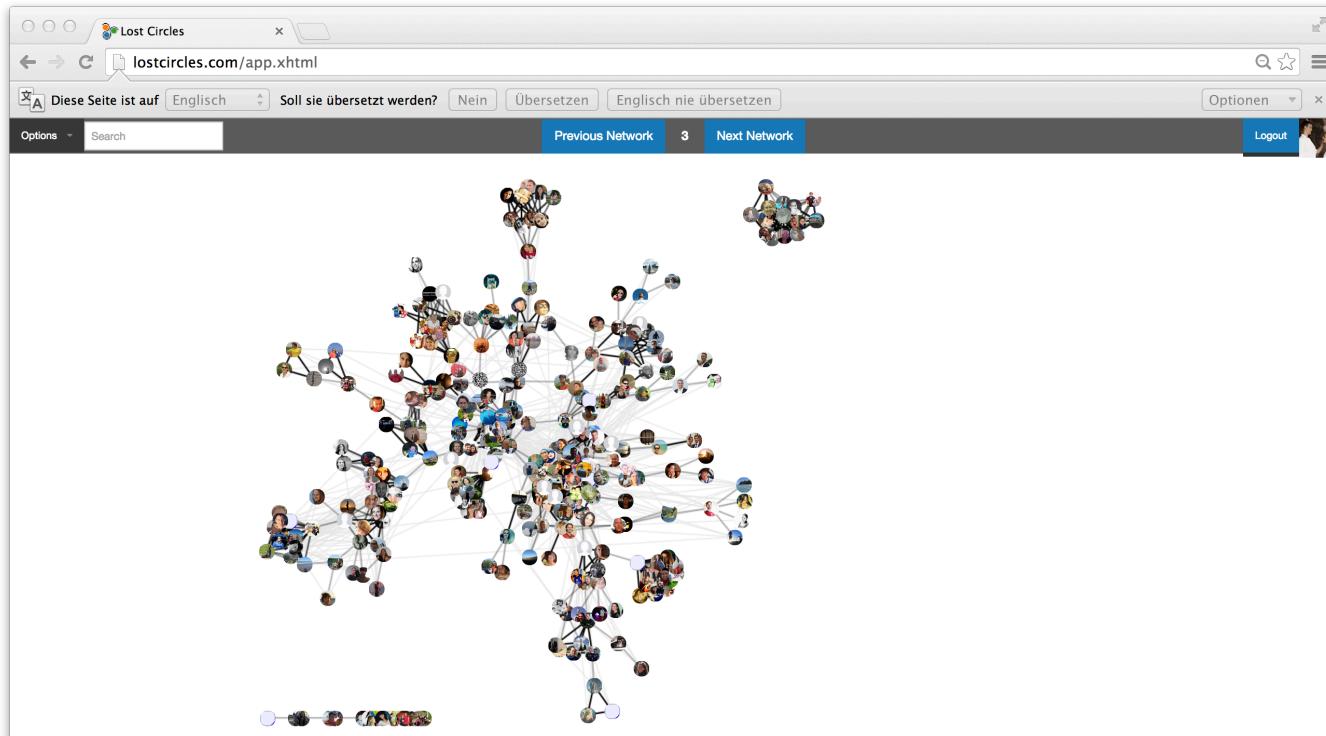
---

- Kleiner Fehler im Code => Übungsaufgabe

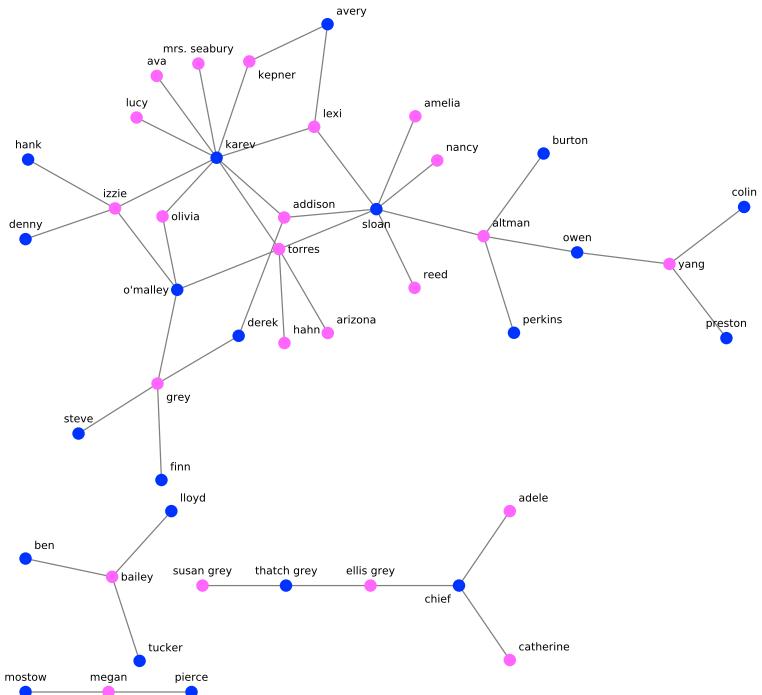
## Graphen - Beispiele



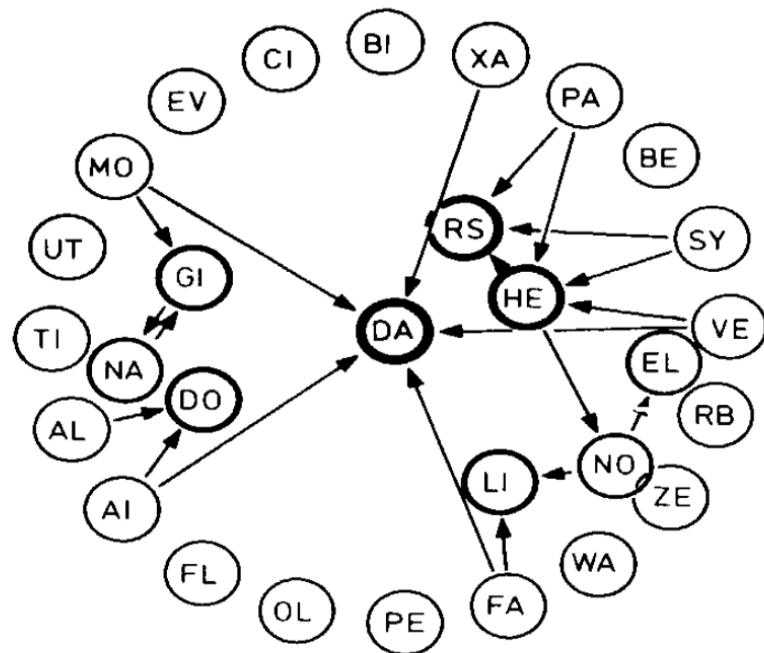
## Graphen - Beispiele



## Graphen - Beispiele

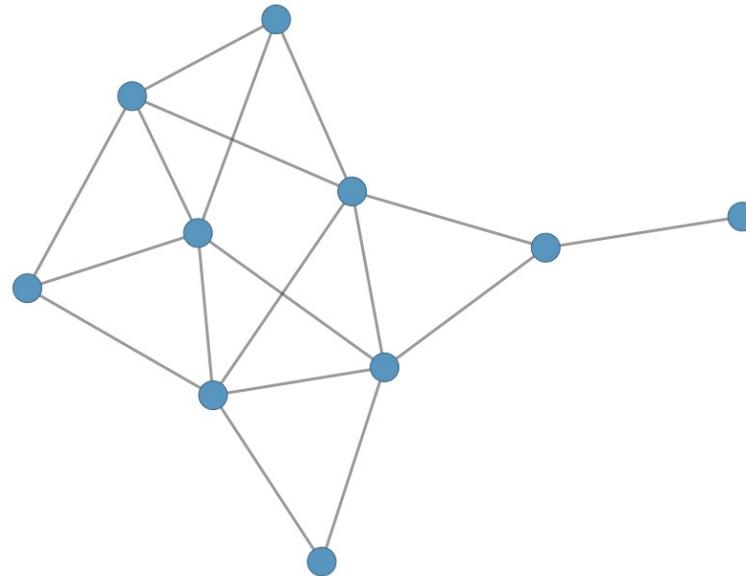


## Graphen - Beispiele



## Graphen (graphs)

- besteht aus Knoten und Kanten
- besonders geeignet, um komplexe Beziehungsstrukturen zu modellieren



## Graphen (graphs)

- Knoten repräsentieren beliebige Objekte
  - Städte, Zustände, Atome, Prozessschritte,
  - können beschriftet oder unbeschriftet sein
- Kanten
  - repräsentieren Beziehungen zwischen je zwei Knoten
  - Straßen zwischen zwei Städten
  - mögliche Zustandsübergänge
  - Bindungen zwischen Atomen
  - Abhängigkeiten
  - können beschriftet oder unbeschriftet sein
    - Abstand, Eingabesymbole, Bindungsart, ...
    - können gerichtet oder ungerichtet sein

## Graphen - formal

Ein Graph  $G$  ist ein Tupel  $G = (V, E)$ . Dabei ist

- $V$  die Menge aller Knoten (vertices),
- $E$  die Menge aller Kanten (edges)

Bei gerichteten Graphen besteht  $E$  aus geordneten Knotenpaaren  $(v_1, v_2)$  (d.h. die Kante geht von  $v_1$  nach  $v_2$ ), bei ungerichteten Graphen besteht  $E$  aus ungeordneten Knotenmengen  $\{v_1, v_2\}$ .

**Frage:** Wie kann ich Graphen speichern?

# Adjazenzmatrix

## Definition

Ist  $G = (V, E)$  ein Graph, dann heißen zwei Knoten  $u, v \in V$  **adjazent** (auch: *benachbart*), falls es eine Kante  $\{u, v\} \in E$  gibt, und die Matrix  $A(G) = (a_{v,w})_{v,w \in V}$  **Adjazenzmatrix** des Graphen mit

$$a_{v,w} = \begin{cases} 1 & \text{falls } \{v, w\} \in E \\ 0 & \text{sonst} \end{cases}$$

$$\begin{array}{cccccccc} & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \\ v_1 & \left( \begin{array}{cccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array} \right) \end{array}$$

## Adjazenzlisten

- Knotenarray, in dem zu jedem Knoten eine lineare Liste mit seinen Nachbarn gespeichert wird
- komprimierte Darstellung der Adjazenzmatrix, in der alle Nullen übersprungen werden

$v_1$	$\rightarrow v_2$
$v_2$	$\rightarrow v_1$
$v_3$	$\rightarrow v_4 \rightarrow v_5$
$v_4$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_7$
$v_5$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_8$
$v_6$	$\rightarrow v_4 \rightarrow v_5 \rightarrow v_7$
$v_7$	$\rightarrow v_4 \rightarrow v_6 \rightarrow v_8$
$v_8$	$\rightarrow v_5 \rightarrow v_7$

## Adjazenzmatrix vs. Adjazenzlisten

Jede der Speichermethoden hat Vor- und Nachteile

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$
$v_1$	0	1	0	0	0	0	0	0
$v_2$	1	0	0	0	0	0	0	0
$v_3$	0	0	0	1	1	0	0	0
$v_4$	0	0	1	0	0	1	1	0
$v_5$	0	0	1	0	0	1	0	1
$v_6$	0	0	0	1	1	0	1	0
$v_7$	0	0	0	1	0	1	0	1
$v_8$	0	0	0	0	1	0	1	0

$v_1$	$\rightarrow v_2$
$v_2$	$\rightarrow v_1$
$v_3$	$\rightarrow v_4 \rightarrow v_5$
$v_4$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_7$
$v_5$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_8$
$v_6$	$\rightarrow v_4 \rightarrow v_5 \rightarrow v_7$
$v_7$	$\rightarrow v_4 \rightarrow v_6 \rightarrow v_8$
$v_8$	$\rightarrow v_5 \rightarrow v_7$

Zwei mögliche Problemstellungen:

- Existiert zwischen zwei Knoten  $v_i$  und  $v_j$  eine Kante?

## Adjazenzmatrix vs. Adjazenzlisten

Jede der Speichermethoden hat Vor- und Nachteile

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$
$v_1$	0	1	0	0	0	0	0	0
$v_2$	1	0	0	0	0	0	0	0
$v_3$	0	0	0	1	1	0	0	0
$v_4$	0	0	1	0	0	1	1	0
$v_5$	0	0	1	0	0	1	0	1
$v_6$	0	0	0	1	1	0	1	0
$v_7$	0	0	0	1	0	1	0	1
$v_8$	0	0	0	0	1	0	1	0

$v_1$	$\rightarrow v_2$
$v_2$	$\rightarrow v_1$
$v_3$	$\rightarrow v_4 \rightarrow v_5$
$v_4$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_7$
$v_5$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_8$
$v_6$	$\rightarrow v_4 \rightarrow v_5 \rightarrow v_7$
$v_7$	$\rightarrow v_4 \rightarrow v_6 \rightarrow v_8$
$v_8$	$\rightarrow v_5 \rightarrow v_7$

Zwei mögliche Problemstellungen:

- Existiert zwischen zwei Knoten  $v_i$  und  $v_j$  eine Kante?
  - Adjazenzmatrix: direkter Zugriff
  - Adjazenzlisten: Durchlaufen aller Nachbarn

## Adjazenzmatrix vs. Adjazenzlisten

Jede der Speichermethoden hat Vor- und Nachteile

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$
$v_1$	0	1	0	0	0	0	0	0
$v_2$	1	0	0	0	0	0	0	0
$v_3$	0	0	0	1	1	0	0	0
$v_4$	0	0	1	0	0	1	1	0
$v_5$	0	0	1	0	0	1	0	1
$v_6$	0	0	0	1	1	0	1	0
$v_7$	0	0	0	1	0	1	0	1
$v_8$	0	0	0	0	1	0	1	0

$v_1$	$\rightarrow v_2$
$v_2$	$\rightarrow v_1$
$v_3$	$\rightarrow v_4 \rightarrow v_5$
$v_4$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_7$
$v_5$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_8$
$v_6$	$\rightarrow v_4 \rightarrow v_5 \rightarrow v_7$
$v_7$	$\rightarrow v_4 \rightarrow v_6 \rightarrow v_8$
$v_8$	$\rightarrow v_5 \rightarrow v_7$

Zwei mögliche Problemstellungen:

- Existiert zwischen zwei Knoten  $v_i$  und  $v_j$  eine Kante?
  - Adjazenzmatrix: direkter Zugriff
  - Adjazenzlisten: Durchlaufen aller Nachbarn
- Wie viele Kanten hat ein Knoten  $v_i$ ?

## Adjazenzmatrix vs. Adjazenzlisten

Jede der Speichermethoden hat Vor- und Nachteile

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$
$v_1$	0	1	0	0	0	0	0	0
$v_2$	1	0	0	0	0	0	0	0
$v_3$	0	0	0	1	1	0	0	0
$v_4$	0	0	1	0	0	1	1	0
$v_5$	0	0	1	0	0	1	0	1
$v_6$	0	0	0	1	1	0	1	0
$v_7$	0	0	0	1	0	1	0	1
$v_8$	0	0	0	0	1	0	1	0

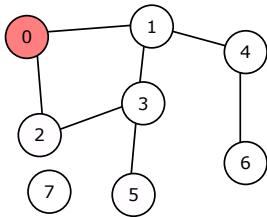
$v_1$	$\rightarrow v_2$
$v_2$	$\rightarrow v_1$
$v_3$	$\rightarrow v_4 \rightarrow v_5$
$v_4$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_7$
$v_5$	$\rightarrow v_3 \rightarrow v_6 \rightarrow v_8$
$v_6$	$\rightarrow v_4 \rightarrow v_5 \rightarrow v_7$
$v_7$	$\rightarrow v_4 \rightarrow v_6 \rightarrow v_8$
$v_8$	$\rightarrow v_5 \rightarrow v_7$

Zwei mögliche Problemstellungen:

- Existiert zwischen zwei Knoten  $v_i$  und  $v_j$  eine Kante?
  - Adjazenzmatrix: direkter Zugriff
  - Adjazenzlisten: Durchlaufen aller Nachbarn
- Wie viele Kanten hat ein Knoten  $v_i$ ?
  - Adjazenzmatrix: Durchlaufen aller Knoten
  - Adjazenzlisten: Durchlaufen aller Nachbarn

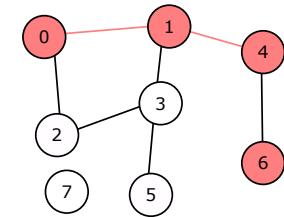
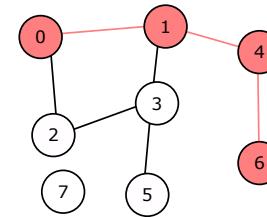
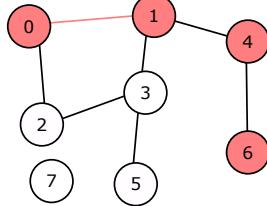
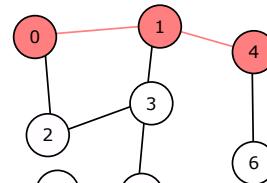
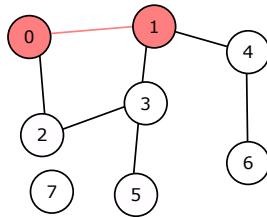
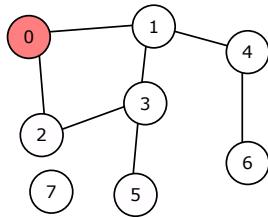
## Graphendurchäufe

Existiert ein Pfad zwischen Knoten 0 und 5?



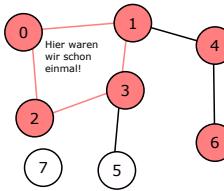
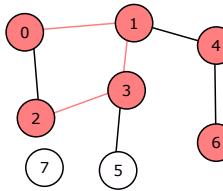
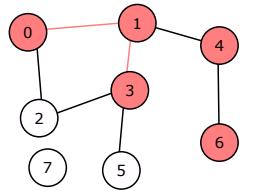
## Graphendurchäufe

Existiert ein Pfad zwischen Knoten 0 und 5?



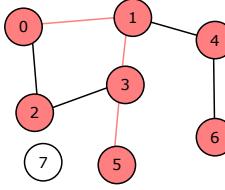
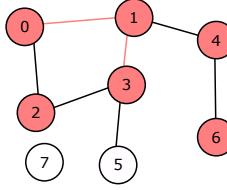
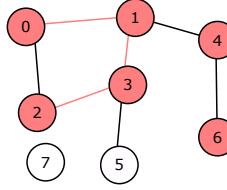
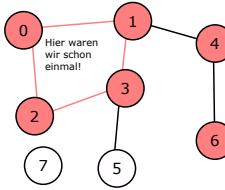
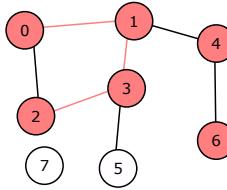
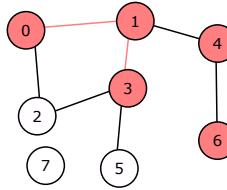
## Graphendurchäufe

- Existiert ein Pfad zwischen Knoten 0 und Knoten 5?



## Graphendurchäufe

- Existiert ein Pfad zwischen Knoten 0 und Knoten 5?



## Graphendurchlauf mit Warteschlange

### Algorithm 10: Breitensuche

**Input:** Graph  $G = (V, E)$ , Wurzel  $s \in V$

**Data:** Knotenwarteschlange  $Q$

**Output:** Breitensuchnummern BFS (anfänglich  $\infty$ )

Vorgänger parent (anfänglich nil)

$BFS[s] = 0$

markiere  $s$ ;  $Q.enqueue(s)$

// initialisiere für Startknoten  $s$

**while**  $!Q.isEmpty$  **do**

// ältester Knoten in der Schlange

// laufe noch unbesuchte Kante

$v = Q.dequeue$

**for** unmarkierte Kanten  $\{v, w\} \in E$

**do**

      markiere  $\{v, w\}$

**if**  $w$  nicht markiert

// falls Knoten unbesucht

**then**

$BFS[w] = BFS[v] + 1$

        markiere  $w$ ;  $Q.enqueue(w)$

        parent[w] =  $v$

// markiere und reihe ein

// merke Vorgänger

## Graphendurchläufe

**Breitensuche:** Strategie, um alle Knoten eines Graphen zu besuchen

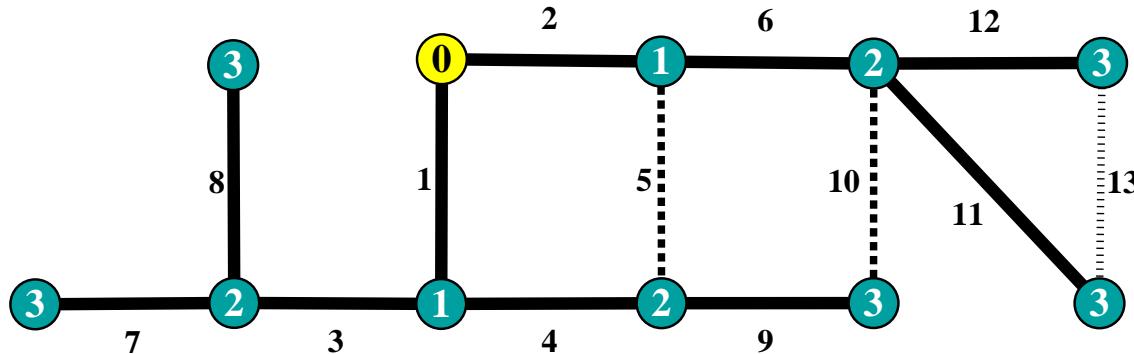
- ausgehend von einem Knoten  $v$
- besuche zuerst den Nachbarn  $u_1$  von  $v$
- dann den Nachbarn von  $u_2$  von  $v$
- usw.
- besuche dann den Nachbarn  $w_1$  von  $u_1$
- dann den Nachbarn von  $w_2$  von  $u_1$
- usw.

Knoten haben zwei Zustände:

*unmarkiert/unbesucht* und *markiert/besucht*

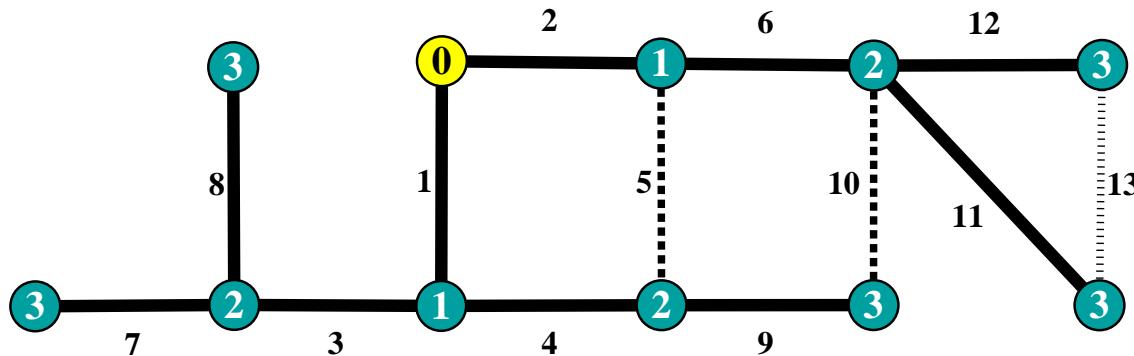
## Beispiel

Breitensuche vom hellen Knoten aus: die Knoten sind mit ihren Breitensuchnummern und die Kanten in Reihenfolge ihres Durchlaufs beschriftet.



## Beispiel

Breitensuche vom hellen Knoten aus: die Knoten sind mit ihren Breitensuchnummern und die Kanten in Reihenfolge ihres Durchlaufs beschriftet.



Beobachtung: Die Breitensuchnummern sind gerade die Längen kürzester Wege von der Wurzel aus.

## Graphendurchläufe

**Tiefensuche:** Strategie, um alle Knoten eines Graphen zu besuchen

- ausgehend von einem Knoten  $v$
- besuche zuerst den Nachbarn  $u_1$  von  $v$
- dann alle Nachbarn von  $u_1$  (aber den nächsten auch erst, wenn ich von einem nicht mehr weiter komme)
- usw.
- besuche dann den Nachbarn  $u_2$  von  $v$
- dann alle Nachbarn von  $u_2$
- usw.

Knoten haben zwei Zustände:

*unmarkiert/unbesucht* und *markiert/besucht*

## Graphendurchlauf mit Stapel

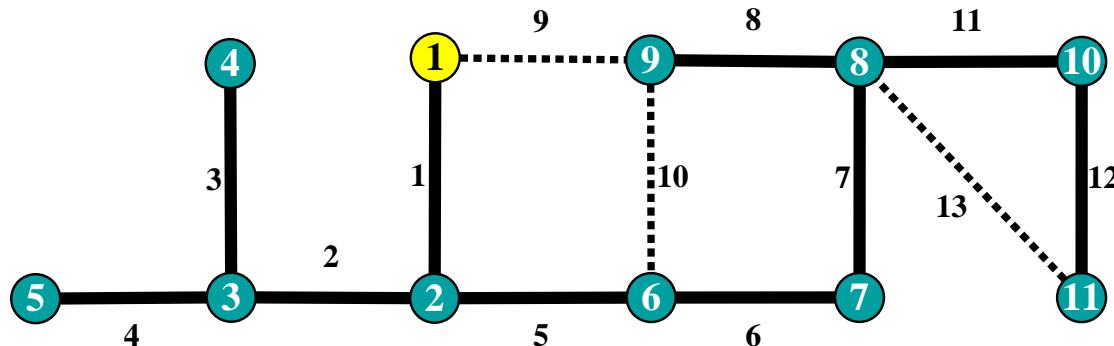
### Algorithm 11: Tiefensuche

**Input:** Graph  $G = (V, E)$ , Wurzel  $s \in V$   
**Data:** Knotenstack  $S$ , Zähler  $d$   
**Output:** Tiefensuchnummern DFS (anfänglich  $\infty$ )  
Vorgänger parent (anfänglich nil)

```
DFS[s] = 1;  d = 2
markiere s;  S.push(s)                                // initialisiere für Startknoten s
while !S.isEmpty do
    v = peek(S)                                         // noch nicht entfernen, bis alle Kanten besucht
    if es ex. unmarkierte Kante {v, w} ∈ E            // laufe unbesuchte Kante
        then
            markiere {v, w}
            if w nicht markiert                         // falls Knoten unbesucht
                then
                    DFS[w] = d;  d = d + 1
                    markiere w;  S.push(w)                  // markiere und lege auf
                    parent[w] = v                           // merke Vorgänger
            else v=S.pop
```

## Beispiel

Tiefensuche vom hellen Knoten aus: die Knoten sind mit ihren Tiefensuchnummern und die Kanten in Reihenfolge ihres Durchlaufs beschriftet.



Hilfreich zum Beispiel bei der Suche nach 2-fach zusammenhängenden Teilgraphen  
(in Vorlesung *Zeichnen von Graphen*)

## Das wichtigste in Kürze

- Eigenschaften
- Darstellung
  - Pseudocode
  - Struktogramm
  - Flussdiagramm
- Abstrakte Datenstrukturen
  - Felder (arrays)
  - Verkettete Listen (linked lists)
  - Stapel (stacks)
  - Graphen (graphs)