Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Jakob Vogel

Computer-Aided Medical Procedures
Technische Universität München

Sortieren jenseits reiner Zahlen

- Bisher nutzen wir den „eingebauten“ Vergleichsoperator zur Sortierung. Dieser definiert eine Ordnung.
- Wir können auch eigene Vergleiche definieren, etwa zur Sortierung nach Absolutbetrag:

```
bool is_less(float a, float b)
{
    return std::abs(a) < std::abs(b);
}
```

- Beispiel:

$$\{-9, 8, 1, -2, 5, 3\} \quad \rightsquigarrow \quad \{1, -2, 3, 5, 8, -9\}$$

- Mit eigenen Vergleichsfunktionen kann man beliebige Daten sortieren. („Relation“, math. Anforderungen beachten!)

Eine Musikbibliothek...

- (Vereinfachende) Modellierung einer MP3-Datei:

```
struct Song
{
    std::string artist;
    std::string title;
    float duration;
};
```

- Die Playlist ist ein Array solcher Songs:

∧ Interpret	Titel	Dauer
Anton	Groovy	3:15:00
Berta	Funky	3:15:00
Caesar	De Bello Gallico	599:31:00

  - Vergleichsfunktion: `song1.artist.compare(song2.artist) < 0`

Eine Musikbibliothek...

- (Vereinfachende) Modellierung einer MP3-Datei:

```
struct Song
{
    std::string artist;
    std::string title;
    float duration;
};
```

- Die Playlist ist ein Array solcher Songs:

Interpret	$\wedge$ Titel	Dauer
Caesar	De Bello Gallico	599:31:00
Berta	Funky	3:15:00
Anton	Groovy	3:15:00

  - Vergleichsfunktion: `song1.title.compare(song2.title) < 0`

Eine Musikbibliothek...

- (Vereinfachende) Modellierung einer MP3-Datei:

```
struct Song
{
    std::string artist;
    std::string title;
    float duration;
};
```

- Die Playlist ist ein Array solcher Songs:

Interpret	Titel	$\wedge$ Dauer
Berta	Funky	3:15:00
Anton	Groovy	3:15:00
Caesar	De Bello Gallico	599:31:00

  - Vergleichsfunktion: `song1.duration < song2.duration`

Eine Musikbibliothek...

- (Vereinfachende) Modellierung einer MP3-Datei:

```
struct Song
{
    std::string artist;
    std::string title;
    float duration;
};
```

- Die Playlist ist ein Array solcher Songs:

Interpret	Titel	∧ Dauer
Anton	Groovy	3:15:00
Berta	Funky	3:15:00
Caesar	De Bello Gallico	599:31:00

  - Vergleichsfunktion: `song1.duration < song2.duration`

Eine Musikbibliothek...

- Welche Sortierung ist korrekt, wenn zwei Elemente gleichen Rang haben?
- Sortierung unter Nebenbedingungen
  - Falls Dauer gleich ist, sortiere nach Interpret...
- Behalte *vorherige* Ordnung im Falle gleichen Ranges...

Stabilität der Sortierung

Ein Sortierverfahren ist **stabil**, wenn bei gleichrangigen Elementen während der Sortierung die vorherige Ordnung erhalten bleibt.

- Der *Insertion Sort* ist **stabil**.
- Der *Selection Sort* ist **nicht stabil**.
  - Verrücken beim Einfügen des Minimums statt Vertauschen
- Der *Merge Sort* ist in der gegebenen Formulierung **nicht stabil**.
  - Forderung, dass im Fall gleichrangiger Anfangselemente immer aus dem ersten Teil gelesen wird
- Der *Quick Sort* ist **nicht stabil**.

Polynome

- Allgemeine Form eines *Polynoms*:

$$p(x) := \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0$$

- $a_i \in \mathbb{R}$ ist der *Koeffizient* für die i-te Potenz x^i
- Ein einzelner Summand $a_i x^i$ heißt *Monom*.
- Die höchste Potenz n ist der *Grad des Polynoms*.

Implementierung

Gegeben sei:

- Array a der Länge $n + 1$ mit Koeffizienten $a[i]$
- Auswertungsstelle x

Ausgabe:

- Wert des Polynoms $\sum_{i=0}^{n} a[i]x^i$

Naïve Implementierung

```
1       p = 0;
2       for i = n down to 0 {
3           m = 1;
4           for j = 1 to i {
5               m = m · x;
            }

6           m = a[i] · m;
7           p = p + m;
        }
```

Für jeden Durchlauf i der Schleife (ab Zeile 2):

- i Multiplikationen für die i-te Potenz in Zeilen 4 und 5
- 1 Multiplikation für den Koeffizienten in Zeile 6
- 1 Addition für die Summierung in Zeile 7

Naïve Implementierung

Gesamtaufwand bei $n+1$ Iterationen:

- Multiplikationen:

$$\left(\sum_{i=1}^{n} i\right) + (n+1) = \left(\frac{n(n+1)}{2}\right) + (n+1) = \frac{1}{2}n^2 + \frac{3}{2}n + 1$$

- Additionen:

$$n+1$$

- Laufzeitfunktion:

$$T(n) = O(n^2)$$

Inkrementelle Implementierung

```
1       p = 0;
2       h = 1;
3       for i = 0 to n {
4           m = a[i] · h;
5           h = h · x;
6           p = p + m;
        }
```

Für jeden Durchlauf i der Schleife (ab Zeile 3):

- 1 Multiplikation für den Koeffizienten in Zeile 4
- 1 Multiplikation für die Potenzierung in Zeile 5
- 1 Addition für die Summierung in Zeile 6

Inkrementelle Implementierung

Gesamtaufwand bei $n+1$ Iterationen:

- Multiplikationen:

$$2(n+1) = 2n+2$$

  Die minimale Anzahl an Multiplikationen ist $2n-1$.

- Additionen:

$$n+1$$

- Laufzeitfunktion:

$$T(n) = O(n)$$

Horner-Schema

```
1       p = a[n];
2       for i = n − 1 down to 0 {
3               p = p · x;
4               p = p + a[i]
        }
```

Für jeden Durchlauf i der Schleife (ab Zeile 2):

- 1 Multiplikation mit einem ausgeklammerten Faktor der Potenz in Zeile 3
- 1 Addition für die Summierung in Zeile 4

Horner-Schema

Gesamtaufwand bei nur n (!) Iterationen:

- Multiplikationen:

$$n$$

- Additionen:

$$n$$

- Laufzeitfunktion:

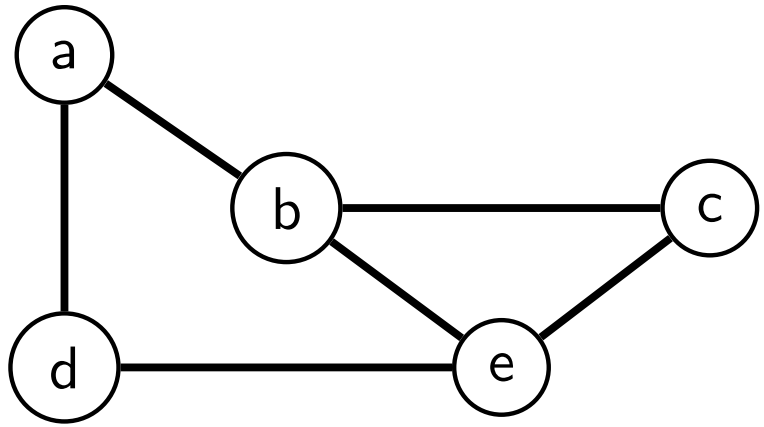
$$T(n) = O(n)$$

Vergleich

- Das naïve Schema verliert mit $O(n^2)$ eindeutig.
- Die anderen beiden Methoden sind $O(n)$.

- Zwei $O(n)$-Funktionen wachsen ab gewisser Größe bis auf konstante Faktoren gleich schnell.
- Das Horner-Schema ist etwa doppelt so schnell wie die inkrementelle Methode!
- Lediglich der Beschleunigungsfaktor ist unabhängig von der Problemgröße!

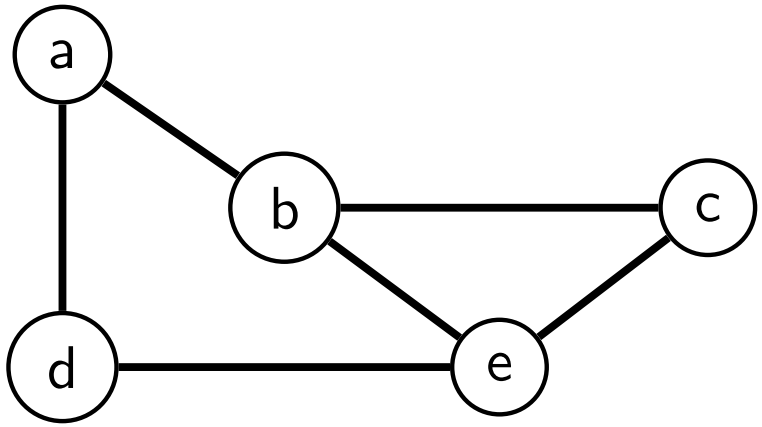
Graphen



- Paar $G = (V, E)$ mit Knoten V und Kanten E ist ein *Graph*
- $E \subseteq \{\{u, v\} : u, v \in V\}$ definiert *ungerichteten* Graphen
- $E \subseteq V \times V$ definiert *gerichteten* Graphen

$$
\begin{aligned}
V &= \{a, b, c, d, e\} \\
E &= \{\{a, b\}, \{a, d\}, \{b, c\}, \{b, e\}, \{c, e\}, \{d, e\}\}
\end{aligned}
$$

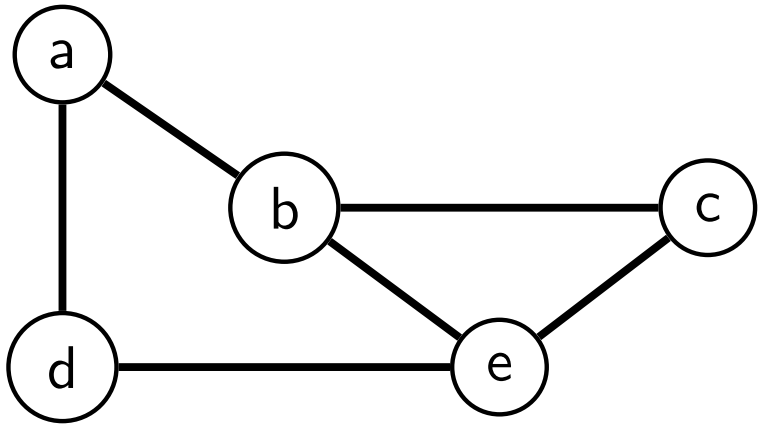
Adjazenz-Matrix I



Adjazenz-Matrix enthält 1, falls Kante zwischen den Knoten existiert. Bei ungerichteten Graphen ist sie *symmetrisch*:

	a	b	c	d	e
a	0	1	0	1	0
b	1	0	1	0	1
c	0	1	0	0	1
d	1	0	0	0	1
e	0	1	1	1	0

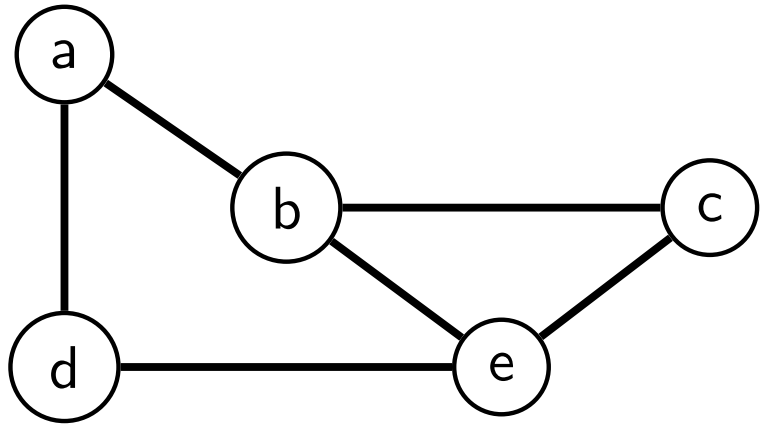
Adjazenz-Matrix I



Adjazenz-Matrix enthält 1, falls Kante zwischen den Knoten existiert. Bei ungerichteten Graphen ist sie *symmetrisch*:

	a	b	c	d	e
a	0	1	0	1	0
b	1	0	1	0	1
c	0	1	0	0	1
d	1	0	0	0	1
e	0	1	1	1	0

Adjazenz-Liste



$$
\begin{array}{l}
a \longrightarrow b \rightarrow d \\
\downarrow \\
b \longrightarrow a \rightarrow c \rightarrow e \\
\downarrow \\
c \longrightarrow b \rightarrow e \\
\downarrow \\
d \longrightarrow a \rightarrow e \\
\downarrow \\
e \longrightarrow b \rightarrow c \rightarrow d
\end{array}
$$

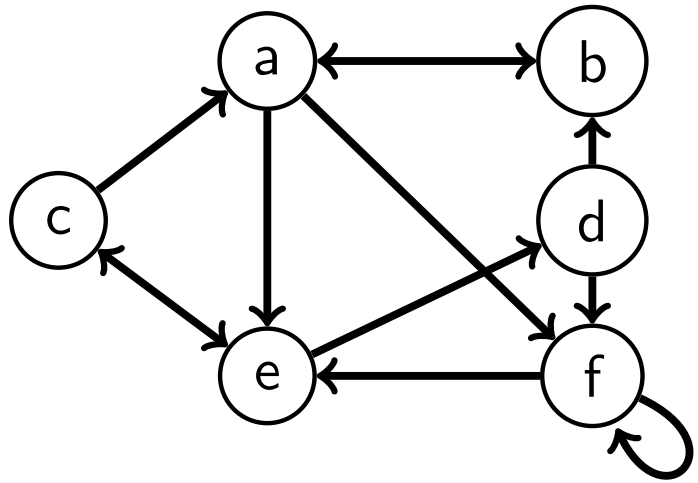
Adjazenz-Matrix II

	a	b	c	d	e	f
a	0	1	0	0	1	1
b	1	0	0	0	0	0
c	1	0	0	0	1	0
d	0	1	0	0	0	1
e	0	0	1	1	0	0
f	0	0	0	0	1	1

- Die Matrix ist offensichtlich **nicht** symmetrisch!
- Der Graph ist folglich gerichtet!
- Beispiel: Kante $c \to a$ existiert, da $A(c, a) = 1$, aber Kante $a \to c$ existiert **nicht**, da $A(a, c) = 0$.

Gerichteter Graph

	a	b	c	d	e	f
a	0	1	0	0	1	1
b	1	0	0	0	0	0
c	1	0	0	0	1	0
d	0	1	0	0	0	1
e	0	0	1	1	0	0
f	0	0	0	0	1	1

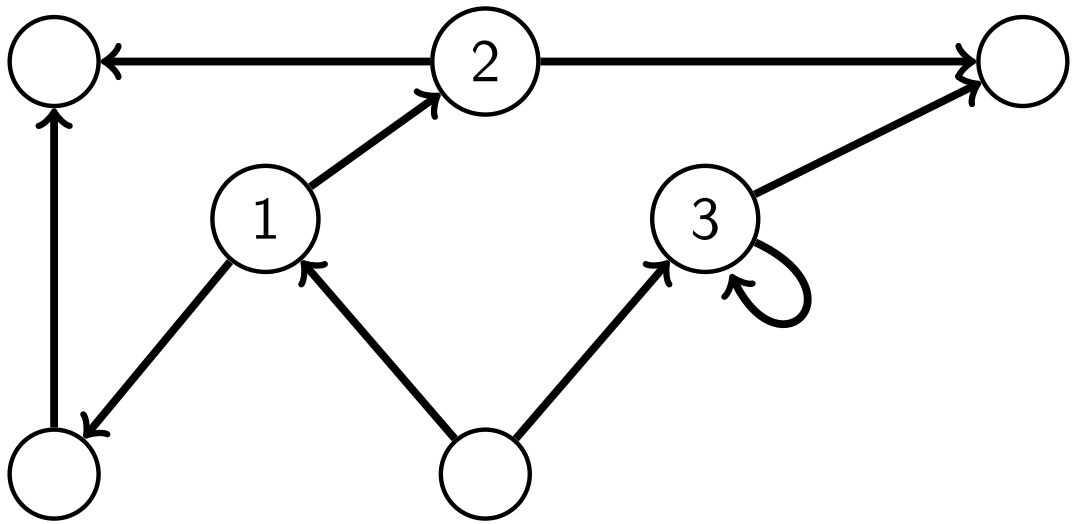


Adjazenz-Liste

	a	b	c	d	e	f
a	0	1	0	0	1	1
b	1	0	0	0	0	0
c	1	0	0	0	1	0
d	0	1	0	0	0	1
e	0	0	1	1	0	0
f	0	0	0	0	1	1

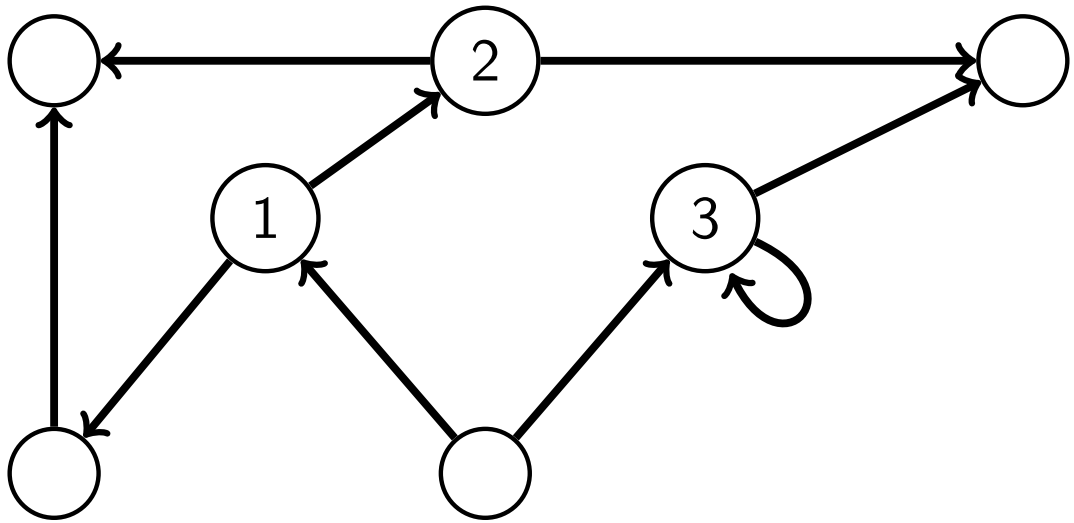
$$
\begin{array}{lllllll}
a & \longrightarrow & b & \rightarrow & e & \rightarrow & f \\
\downarrow & & & & & & \\
b & \longrightarrow & a & & & & \\
\downarrow & & & & & & \\
c & \longrightarrow & a & \rightarrow & e & & \\
\downarrow & & & & & & \\
d & \longrightarrow & b & \rightarrow & f & & \\
\downarrow & & & & & & \\
e & \longrightarrow & c & \rightarrow & d & & \\
\downarrow & & & & & & \\
f & \longrightarrow & e & \rightarrow & f & &
\end{array}
$$

Eingangs- und Ausgangsgrad von Knoten



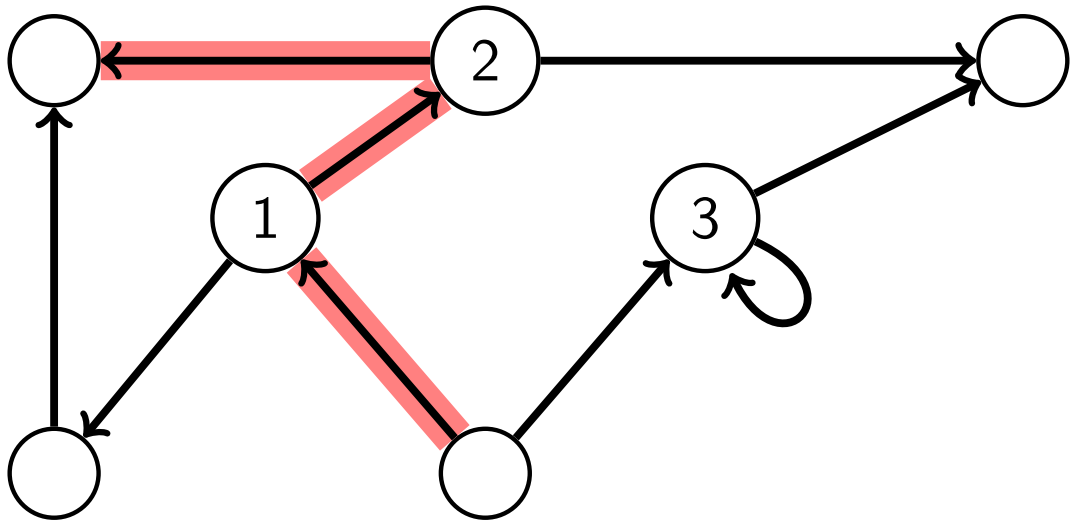
- Der Eingangsgrad $indeg(v)$ eines Knotens v ist die Anzahl der *zu* v führenden Kanten
- Der Ausgangsgrad $outdeg(v)$ die Anzahl der *von* v *weg* führenden Kanten

Eingangs- und Ausgangsgrad von Knoten



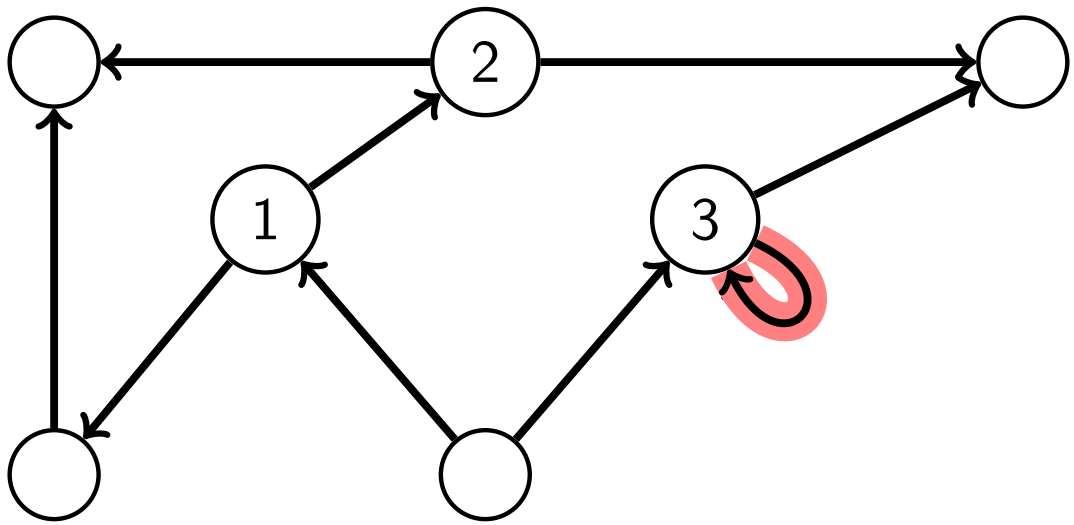
$$outdeg(1) = 2 \qquad indeg(1) = 1$$
$$outdeg(2) = 2 \qquad indeg(2) = 1$$
$$outdeg(3) = 2 \qquad indeg(3) = 2$$

Pfade, Zyklen, und Zusammenhang



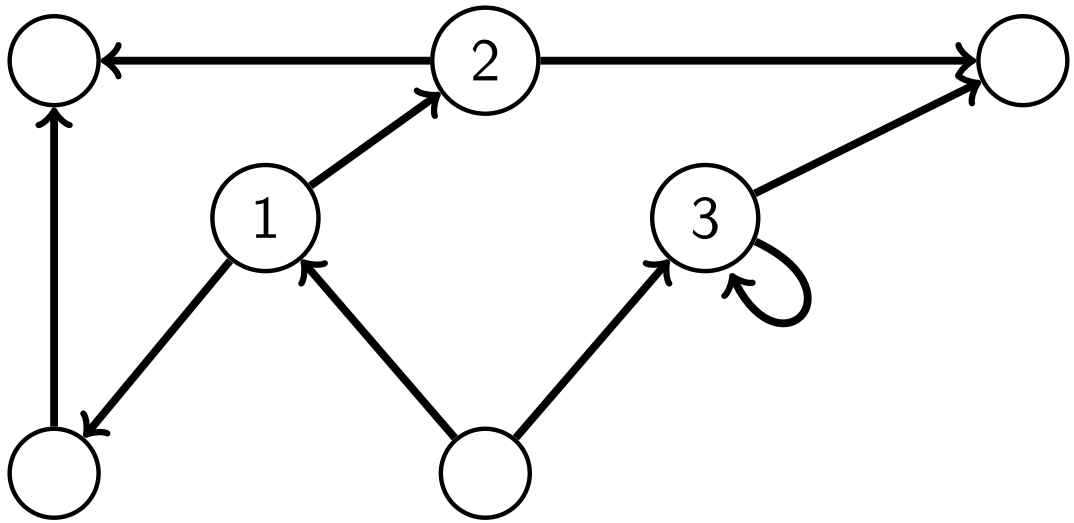
- Ein Pfad von v nach u ist eine Folge von Knoten $(v_0, ..., v_k)$ mit $v_0 = v$ und $v_k = u$ sowie $\{v_i, v_{i+1}\} \in E$ bzw. $(v_i, v_{i+1}) \in E$
- Die *Länge* dieses Pfads ist k.

Pfade, Zyklen, und Zusammenhang



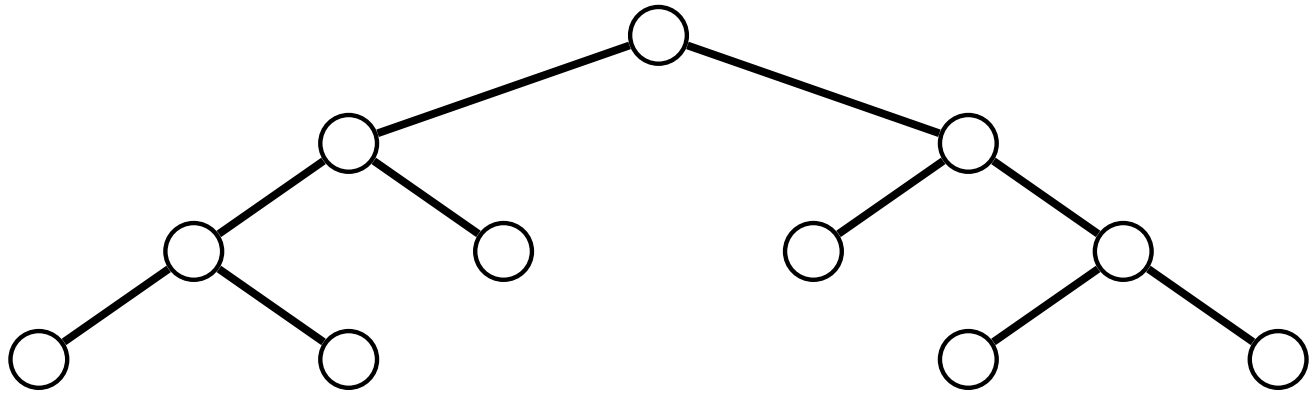
- Ein *Zyklus* ist ein Pfad $(v_0, ..., v_k)$ mit $v_0 = v_k$, der mindestens eine Kante enthält.
- Siehe Vorlesung für vollständige Definition!
- Hier ist die Schleife ein Zyklus!

Pfade, Zyklen, und Zusammenhang



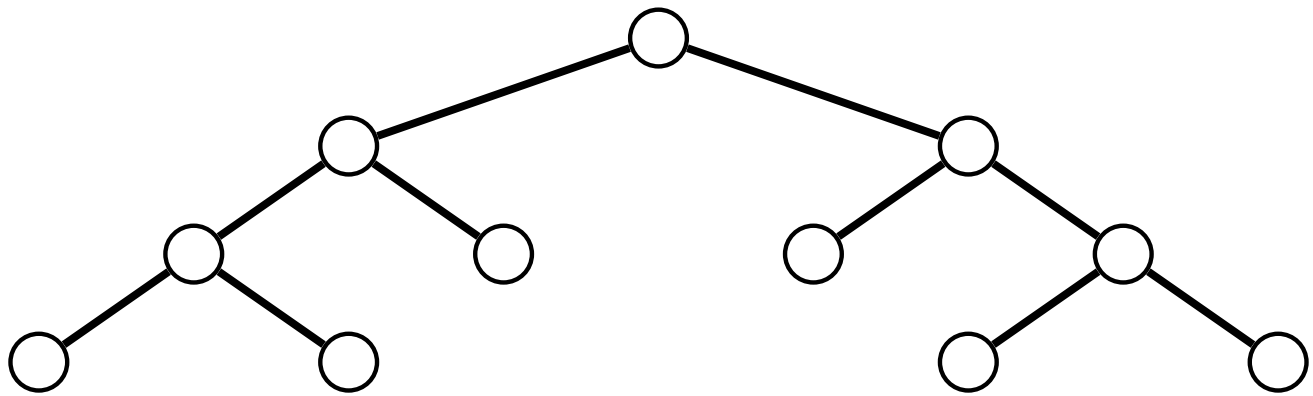
- Ein *ungerichteter Graph* ist *zusammenhängend*, falls jeder Knoten von jedem anderen erreichbar ist.
- Ein *gerichteter Graph* ist *stark zusammenhängend*, falls jeder Knoten – unter Beachtung der Kantenrichtungen! – von jedem anderen erreichbar ist.
- Er ist *schwach zusammenhängend*, falls der durch Nichtbeachtung der Richtungen entstehende ungerichtete Graph zusammenhängend ist. Das ist hier der Fall!

Bäume



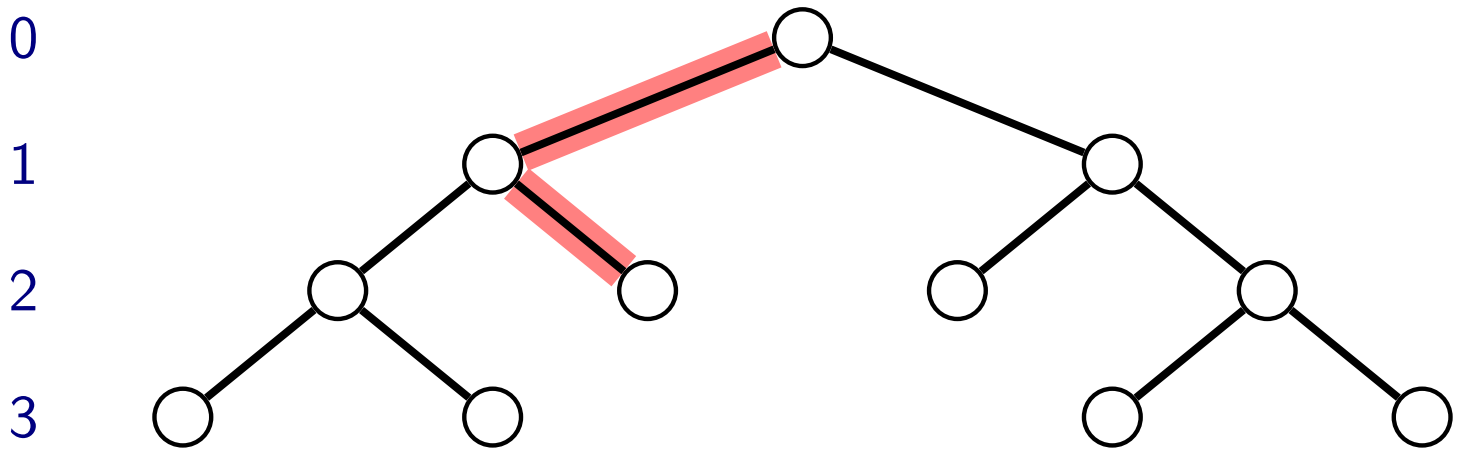
- Bäume sind *ungerichtete*, *azyklische*, *zusammenhängende* Graphen.
- Genau ein Knoten wird als *Wurzel* bezeichnet.
- Alle anderen Knoten haben genau eine Kante zu ihrem *Vaterknoten* v. Man nennt sie *Kind* von v.
- Die Anzahl der Kindknoten pro Vaterknoten ist nicht notwendigerweise eingeschränkt.
- Zwei Knoten sind durch genau einen Pfad verbunden.

Bäume



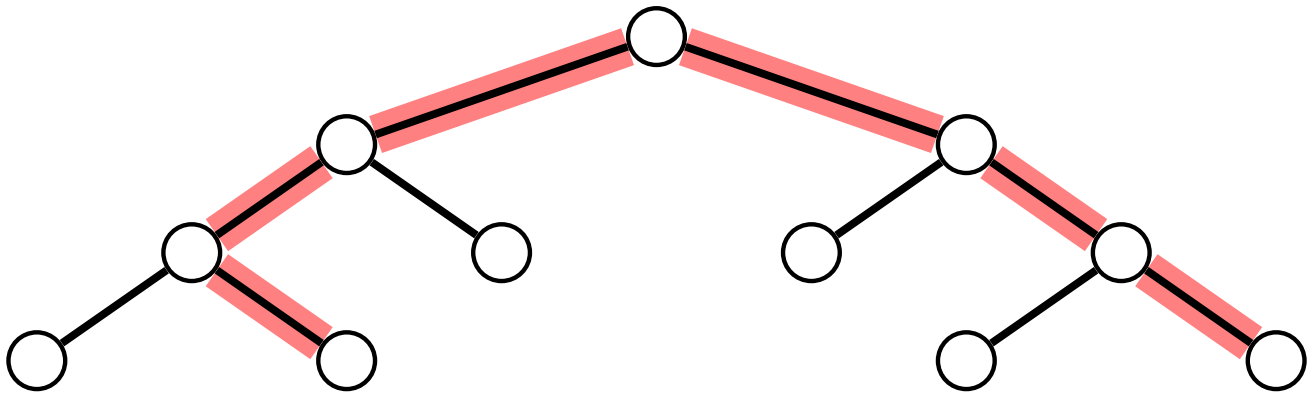
- Falls $|V| = n$, dann $|E| = n - 1$, bzw. $|E| = |V| - 1$
  - Alle Knoten bis auf die Wurzel haben je eine Kante zum Vaterknoten!

Bäume



- Die *Tiefe* eines Knotens v ist die Länge des (eindeutigen) Pfads von der Wurzel zu v.
- Alle Knoten der gleichen Tiefe bilden eine *Ebene*.
- Die *Höhe* eines Baums ist die maximale Tiefe. Hier 3.

Bäume



- Der längste Pfad verbindet zwei *Blätter* der Ebene mit höchstem Index über die Wurzel.
- Die Länge entspricht also der doppelten Baumhöhe.

Binärbäume

- Ein Knoten hat 0, 1 oder 2 Kinder.
- Falls der Binärbaum vollständig ist und n Blätter hat, dann...
  - ...hat die Ebene darüber $\frac{n}{2}$ Knoten.
  - ...hat jede weitere Ebene jeweils halb so viele Knoten wie die nächst-tiefere.
  - ...errechnet sich die Höhe x als Lösung von $2^x = n$.

  Insgesamt ergibt sich also eine Höhe $x = \log_2 n$.

Komplexität von Merge und Quick Sort

- Beide teilen das Problem jeweils hälftig auf. (Zumindest besteht beim Quick Sort die Hoffnung!)
- Beide haben im Mittel die Komplexität $O(n \log n)$.
- Der Faktor $\log n$ ergibt sich aus der binär(-baum-artig)-en Aufteilung.
- Der Faktor n ergibt sich...
  - ...beim Merge Sort durch die Zusammensortierung der einzeln sortieren Teile.
  - ...beim Quick Sort durch die Partitionierung gemäß Pivotwert.
- Diese Betrachtung ist natürlich informell!

Datenstruktur für Binärbäume

```
struct Node
{
    Node* parent;
    int payload;
    Node* left;
    Node* right;
};
```

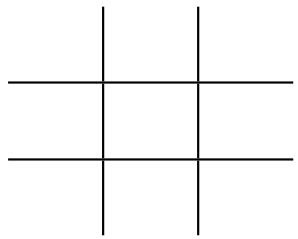
- Weitere Details hierzu in Vorlesung!

Anwendung und Traversierung

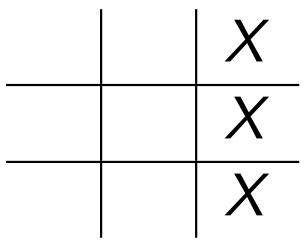
- Anwendungsbeispiele für Bäume aus der Vorlesung bekannt.
- Oft relevant ist dabei die Traversierung (das Durchlaufen) der Knoten. Breiten- oder Tiefensuche?
  - Betrachte ich zuerst das linke/erste Kind?
  - Betrachte ich zuerst das rechte/letzte Kind?
  - Verarbeite ich erst den assoziierten Datenwert?
- Wir betrachten nun ein Beispiel aus der KI.

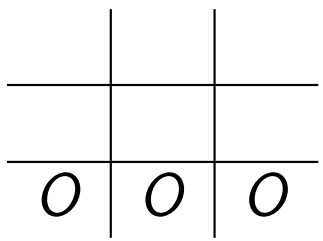
Tic–Tac–Toe

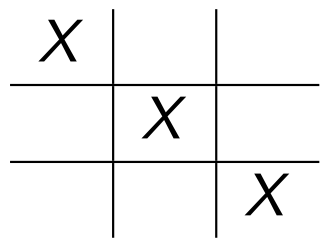
- Höchstwahrscheinlich wohl bekannt!
- Zwei Spieler füllen abwechselnd ein „Spielfeld“ mit ihren Symbolen X und O



- Wer zuerst drei Symbole vertikal, horizontal oder diagonal hat, gewinnt.



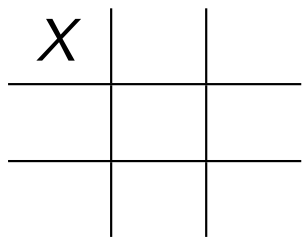


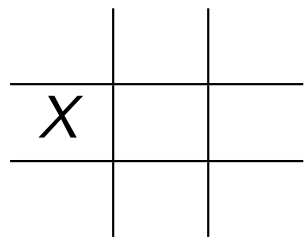


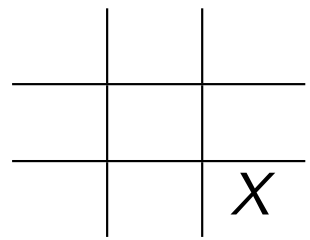
- Bei optimalen Spielern ist das Spiel nicht entscheidbar!

Spielverläufe

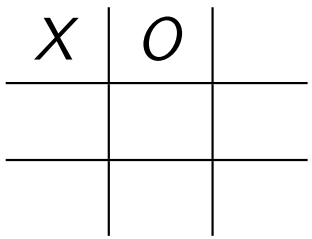
- Alle möglichen Spielverläufe passen in einen Baum.
- Die Wurzel ist das leere Spielfeld.
- Die erste Ebene enthält alle 9 möglichen ersten Züge, darunter:

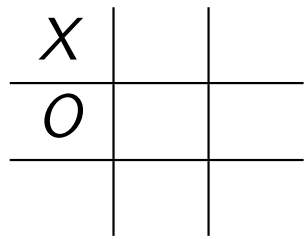


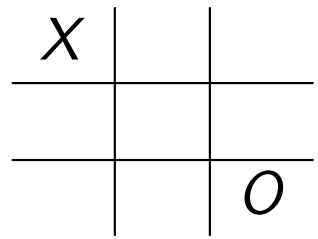




- Die zweite Ebene enthält für jeden dieser 9 Züge die 8 Reaktionen, etwa für den ersten Fall:







- Alle Pfade von der Wurzel zu den Blättern ergeben die Spielverläufe, die Baumhöhe ist also 9.

Problemgröße

- Falls das Feld immer komplett ausgefüllt wird:

0	$1 = 1$	Anfang
1	$9 = 9$	Zug 1 durch X
2	$72 = 9 \cdot 8$	Zug 2 durch O
3	$504 = 9 \cdot 8 \cdot 7$	Zug 3 durch X
4	$3.024 = 9 \cdot 8 \cdot 7 \cdot 6$	Zug 4 durch O
5	$15.120 = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5$	Zug 5 durch X
6	$60.480 = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4$	Zug 6 durch O
7	$181.440 = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3$	Zug 7 durch X
8	$362.880 = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2$	Zug 8 durch O
9	$362.880 = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$	Zug 9 durch X

  Also 362.880 Blätter bzw. Spielverläufe, insgesamt 986.410 Knoten.

- Bei Abbruch nach Sieg nur 255.168, bei Verwendung von Drehungen und Spiegelungen nur 31.896 Blätter.

Wie spielen?

- Ab hier nicht mehr Klausur-relevant!
- Spiele so, dass der Baum in Richtung eines Blattes durchlaufen wird, das unseren Sieg bedeutet.
- Der Gegenspieler versucht natürlich das gleiche und unterläuft unsere Strategie!
- Aus den Zügen der Spieler ergibt sich ein Pfad von der Wurzel zu einem Spielende/Blatt.

Utility-Funktion für Spielausgänge/Blätter

- Alle Spielenden mit 3 X-Feldern in einer Reihe bedeuten unseren Sieg. Sie haben den Nutzen 100%.
- Alle anderen Ausgänge sind Unentschieden oder bedeuten unsere Niederlage. Sie haben den Nutzen 0%.
- Das definiert die Nutzen-Funktion $\mathit{UTILITY}(b)$ für Blätter b!

Utility-Funktion für Zwischenstände/innere Knoten

- An jedem Vaterknoten entscheidet sich der an der Reihe befindliche Spieler für eines der Kinder!
- Entscheidung basiert auf Gewinnwahrscheinlichkeit bei Auswahl von Kindknoten/Zwischenzustand s:

$$UTILITY(s) = \frac{\#\text{ Siege im Teilbaum mit Wurzel } s}{\#\text{ Blätter im Teilbaum mit Wurzel } s}$$

- Auswertung erfordert Traversierung des Teilbaums!

Ausblick

- Man kann auch komplexere Nutzenfunktionen verwenden und zwischen Unentschieden und Niederlage unterscheiden.
  - Minimax-Algorithmus
- Die Suche kann etwa durch Verwerfen von Teilbäumen beschleunigt werden.
  - α-β-Pruning
- Komplexere Spiele wie Schach oder Go funktionieren theoretisch genau so, aber der Baum ist **gigantisch**!
  - Abschätzung des Nutzens im Spielzustand statt rekursiver Berechnung bzw. Traversierung.
- Der Prozess lässt sich auf Spiele mit mehr als 2 Teilnehmern erweitern.
- Spiele mit zufälligen Elementen (wie Würfeln oder Karten) lassen sich sehr ähnlich lösen.