Datenstrukturen

Guido Moerkotte

Literatur

Thomas H. Cormen, Charles E. Leiserson und R. L. Rivest:

Introduction to Algorithms

MIT Press

- ISBN: 0-262-53091-9 pb, ca. 75,- DM
- ISBN: 0-262-03141-8 hd, ca. 146,- DM

Lernziele

1. Für jedes Problem gibt es viele unterschiedlich gute Lösungen.
2. Spezifizieren/Quantifizieren von “gut”.
3. oft benötigte Datenstrukturen und Algorithmen
4. Probleme selber lösen, Lösungen beurteilen

Algorithmen

Ein Algorithmus ist ein

- Verfahren (ähnlich Kochrezept)

zur Lösung eines Problems.

Üblicherweise:

- EINGABE $\longrightarrow$ AUSGABE

EINGABE durch Problemstellung beschrieben

$\longrightarrow$ durch Algorithmus beschrieben.

AUSGABE durch Problemstellung beschrieben

Ein Algorithmus ist kein Programm:

- Abstraktion von der Darstellung

Datenstrukturen

Datenstrukturen dienen der Darstellung

- der Eingabedaten
- der Hilfsdaten
- der Ausgabedaten

Am wichtigsten: Hilfsdaten, da auf ihnen gearbeitet wird.

Die Darstellung beeinflußt erheblich die (Effizienz der) Arbeitsweise mit den Daten.

Beispiel:

- Telefonbuch nach Telefonnummern sortiert

Suchen der Telefonnummer von **Guido Moerkotte** wird sehr ineffizient.

Problemstellung Sortieren

input: Eine Folge $\langle a_1, \ldots, a_n \rangle$ von Zahlen

output: Eine Permutation

$$a'_1, \ldots, a'_n$$

der Zahlen

$$a_1, \ldots, a_n$$

mit $a'_i \leq a'_{i+1}$ für $0 < i < n$.

Wahl der Datenstruktur:

- Array, Feld

Sortieren durch Einfügen

Idee: Wie beim Kartenspielen:

1. Nimmt die erste Karte auf.
2. Dann füge jede weitere Karte von **rechts** in die bereits aufgenommenen Karten so ein, daß die Folge auf der Hand immer sortiert ist.

Merke: Die Karten müssen im 2. Schritt verschoben werden, damit sie sich nicht verdecken.

Pseudo-Code

Wir müssen Algorithmen hinschreiben können.

Dazu benutzen wir Pseudo-Code.

1. $\triangleright$codiert einen Kommentar
2. $\leftarrow$codiert eine Zuweisung
3. $[\cdot]$ ist Array-Zugriff
4. Datenstrukturen haben Eigenschaften (Attribute). Diese werden über Funktionen abgerufen. (Bsp.: **length(A)** für ein Array A).

Zusätzlich benutzen wir Kontrollkonstrukte wie **if-then-else**, **while**, **for** etc.

Sortieren durch Einfügen

Algorithmus:

```
INSERTION-SORT (A)
for j ← 2 to length[A]
   do key ← A[j]
      ▷ Insert A[j] into sorted sequence A[1..j − 1].
      i ← j − 1
      while i > 0 and A[i] > key
         do A[i + 1] ← A[i]
            i ← i − 1
      A[i + 1] ← key
```

Sortieren durch Einfügen

Beispielanwendung:

5 2 4 6 1 3

2 5 4 6 1 3

2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6

Was jetzt?

Wir haben:

- Einen Algorithmus zum Sortieren einer Folge.

Fragen:

1. Wie "**gut**" ist **sortieren durch einfügen**?
2. Gibt es "**bessere**" Algorithmen?

Analyse von Algorithmen

Was heißt "**gut**"?

Wie messen wir "**gut**"?

- Unser Interesse: Laufzeit
- Gemessen in: Schritten
- Vorteil: Maschinenunabhängigkeit

Wir gehen dabei von **random access machine**s aus.

Random Access Machine (RAM)

- Es wird ein Schritt nach dem anderen ausgeführt.
- Schritt: Test, Zuweisung usw.
- Jeder Schritt i bekommt c_i
- k-maliges Ausführen hat dann Kosten $k * c_i$
- Gesamtkosten $= \Sigma$

Eine Beispielanalyse

Laufzeit-/Komplexitäts-

Analyse von **INSERTION-SORT**:

1. komplexe Formel herleiten
2. Darstellung vereinfachen
3. erlaubt einfacheren Vergleich verschiedener Algorithmen

Analyse von INSERTION-SORT

Sei t_j für $j = 2, \ldots, n$ die Anzahl der Ausführungen des Schleifentests in Anweisung 5 für ein gegebenes j. Wir definieren nun folgende Kostenkonstanten und stellen folgende Ausführungsanzahlen fest:

INSERTION-SORT (A)	**cost**	*times*
1 **for** $j \leftarrow 2$ **to length**$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 $\triangleright$ Insert $A[j]$ into the sorted		
$\triangleright$ sequence $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] >$ **key**	c_5	$\Sigma_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\Sigma_{j=2}^{n}(t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\Sigma_{j=2}^{n}(t_j - 1)$
8 $A[i+1] \leftarrow$ **key**	c_8	$n-1$

Analyse von INSERTION-SORT

Wir addieren die Produkte

Kosten mal Anzahl

und erhalten als Laufzeit $T(n)$:

$$\begin{aligned} T(n) \;=\; & c_1 n + c_2(n-1) + c_4(n-1) + \\ & c_5 * \sum_{j=2}^{n} t_j + \\ & (c_6 + c_7) * \sum_{j=2}^{n} (t_j - 1) + \\ & c_8(n-1) \end{aligned}$$

Was sagt uns diese Formel?

Was machen wir mit t_j?

Analyse von Algorithmen

Die Analyse von Algorithmen ist nicht einfach!

Folgendes erschwert die Analyse:

Wir beobachten beim Sortieren durch Einfügen:

- Die Anzahl der Schritte ist abhängig von der Folge selbst. Bei gleicher Anzahl von Elementen benötigt der Algorithmus für unterschiedliche Folgen unterschiedlich viele Schritte. Diese Abhängigkeit sieht man deutlich in Schritt 5.

Daher benötigen wir Möglichkeiten verschiedene Ergebnisse für Eingaben gleicher Größe zusammenzufassen.

Analyse von Algorithmen

Zum Zusammenfassen hat man mehrere Möglichkeiten:

best-case Es wird für jede Eingabegröße das Minimum der Anzahl der Schritte gezählt, die die Eingaben dieser Größe verursachen.

worst-case Es wird für jede Eingabegröße das Maximum der Anzahl der Schritte gezählt, die die Eingaben dieser Größe verursachen.

average-case Es wird die durschnittliche Anzahl der Schritte aller Eingaben einer Eingabegröße gezählt.

Wir werden hauptsächlich **worst-case**-Analysen durchführen.

Analyse von INSERTION-SORT

Den **best-case** erhält man, falls die Folge bereits sortiert ist. Dann ist $t_j = 1$ für alle $j = 2, \dots, n$. Aus

$$
\begin{aligned}
T(n) \;=\;& c_1 n + c_2(n-1) + c_4(n-1) + \\
& c_5 * \sum_{j=2}^{n} t_j + \\
& (c_6 + c_7) * \sum_{j=2}^{n} (t_j - 1) + \\
& c_8(n-1)
\end{aligned}
$$

wird dann

$$
\begin{aligned}
T(n) \;=\;& c_1 n + c_2(n-1) + c_4(n-1) + \\
& c_5(n-1) + c_8(n-1) \\
=\;& (c_1 + c_2 + c_4 + c_5 + c_8)n - \\
& (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

Wir erhalten also eine Funktion, die in n **linear** ist.

Den **worst-case** erhält man, falls die Folge der umgekehrten Sortierung entspricht. Dann ist $t_j = j$ für alle $j = 2, \ldots, n$. Mit $\Sigma_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ und $\Sigma_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$ erhalten wir

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \\ & \quad c_5 * \sum_{j=2}^{n} t_j + (c_6 + c_7) * \sum_{j=2}^{n} (t_j - 1) + \\ & \quad c_8(n-1) \end{aligned}$$

wird dann

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \\ & \quad c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) + \\ & \quad c_8(n-1) \\ &= \frac{c_5 + c_6 + c_7}{2} n^2 + \\ & \quad \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right) n - \\ & \quad (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Also ist $T(n)$ quadratisch in n.

Wachstumsordnung

Da wir nicht an den Details einer Funktion $an^2 + bn + c$ für Konstanten a, b, c interessiert sind, wollen wir die Konstanten ignorieren.

Da des weiteren n^2 schneller wächst als n oder jede Konstante, vernachlässigen wir auch die kleineren Terme.

Wir sind also nur an der Wachstumsordnung (-rate) n^2 interessiert. Hierfür führen wir die Notation $\Theta(n^2)$ (zunächst informell) ein.

Wir sagen also beispielsweise für INSERTION-SORT, daß er eine worst-case Laufzeitkomplexität von $\Theta(n^2)$ hat.

In dieser Notation sehen wir schnell, daß ein Algorithmus mit Laufzeitkomplexität $\Theta(n^3)$ schlechter ist als einer mit $\Theta(n^2)$.

Wie entwirft man Algorithmen?

Algorithmen zu entwerfen ist eine Kunst.

Es gibt nur wenige Verfahren.

Ein Verfahren (noch aus dem alten Rom):

- divide et impera (teile und herrsche, divide and conquer)

Vorgehen:

Teile: Zerlege ursprüngliches Problem in Teilprobleme.

Herrsche: Löse die Teilprobleme getrennt.

Kombiniere: Füge die Teillösungen zusammen.

Teile und Herrsche: Beispiel

Problemstellung:

- Sortiere eine Folge

Sortieren durch Mischen:

Teile: Zerlege Folge in zwei Teilfolgen

Herrsche: Sortiere Teilfolgen REKURSIV

Kombiniere: Füge sortierte Teilfolgen durch Mischen zu einer sortierten Gesamtfolge zusammen

Beachte: Folge der Länge 0 oder 1 ist bereits sortiert.

Sortieren durch Mischen

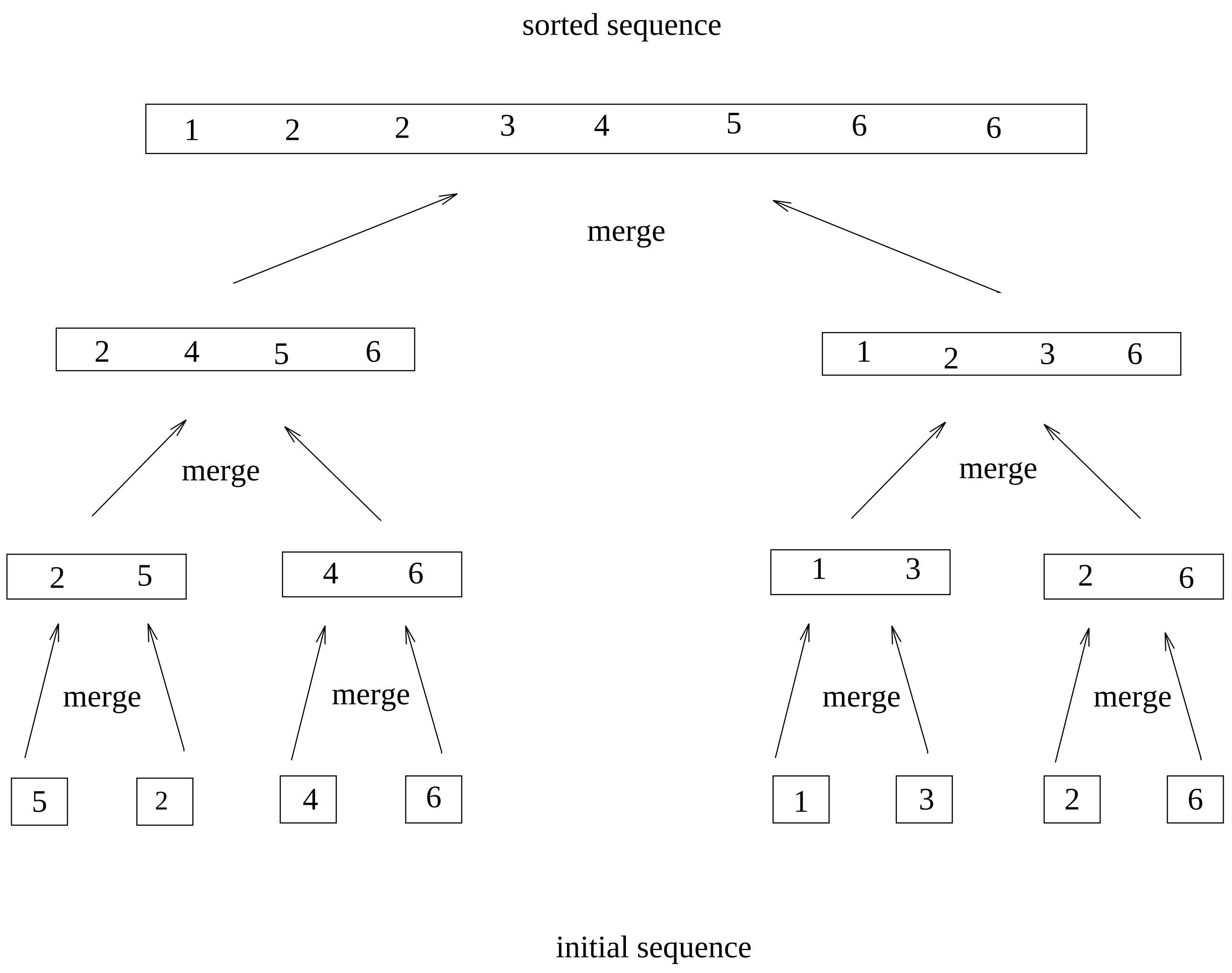
MERGE-SORT(**A,p,r**)
if $p < r$
  then $q \leftarrow \lfloor (p+r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q+1, r)$
    MERGE(A, p, q, r)

Dabei benutzen wir die Prozedur **MERGE** um die zwei Teilfolgen $A[p \ldots q]$ und $A[q+1 \ldots r]$ von A zu mischen. Aufwand: $\Theta(n)$ mit $n = r - p + 1$. (s. Übung)

Prinzipielles Vorgehen beim Mischen zweier Folgen:

> Nimm immer das kleines Element beider Folgen als nächstes Element der zu konstruierenden Gesamtfolge.

Sortieren durch Mischen



Analyse von Teile-und-Herrsche-Algorithmen

- Teile-und-Herrsche-Algorithmen sind oft rekursiv.
- Daher kann man ihre Laufzeit auch am besten durch “rekursive” Gleichungen beschreiben.
- Diese heißen **rekurrente Gleichungen**, oder kurz **Rekurrenzen**.
- Mathematische Werkzeuge helfen beim Lösen.

Analyse von Teile-und-Herrsche-Algorithmen

Analog der Vorgehensweise beim Entwurf werden die Rekurrenzen erstellt:

- Gesamtaufwand sei $T(n)$ für Problemgröße n
- Für kleine Probleme ($n < c$) für eine Konstante c ist der Aufwand konstant.
  Dies notieren wir durch $\Theta(1)$.
- Wir teilen das Problem in a Teile der Größe $1/b$ der Ursprungsgröße.
- $D(n)$ sei der Aufwand für das Teilen.
- $C(n)$ sei der Aufwand für das Kombinieren der Teillösungen.

erhalten wir die Rekurrenz:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{sonst} \end{cases}$$

Analyse von MERGE-SORT

Annahme:

- n ist Zweierpotenz, d.h. $n = 2^k$ für geeignetes k.

Das vereinfacht unsere Analyse, da dann die Teilfolgen genau die Größe $n/2$ haben.

Bemerkung:

- Dies beeinflußt nicht Wachstumsordnung.

Analyse von MERGE-SORT

Die Teile:

Teile: die Mitte der Folge kann in konstanter Zeit ermittelt werden.
Daher: $D(n) = \Theta(1)$

Herrsche: Wir teilen in 2 Teile der Größe $n/2$.
Dies ergibt den Anteil: $2T(n/2)$

Kombiniere: MERGE hat **linearen** Aufwand,
d.h.: $C(n) = \Theta(n)$

Vereinfachung:

- Da $\Theta(n) + \Theta(1)$ immer noch linearen Aufwand hat, können wir diese Summe durch $\Theta(n)$ ersetzen.

Die Summe:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Die Lösung der Rekurrenz ist

$$T(n) = \Theta(n \lg n)$$

(lg ist Log. zur Basis 2)

Folgerung:

- Für große n ist
  - MERGE-SORT mit $\Theta(n \lg n)$ besser als
  - INSERTION-SORT mit $\Theta(n^2)$ worst-case-Verhalten.

So sieht das aus:

n	$16 = 2^4$	$1024 = 2^{10}$	$1.048.576 = 2^{20}$
$n \lg n$	64	10.240	20.971.520
n^2	256	1.048.576	1.099.511.627.776

(Letzter Eintrag: Telefonbucheinträge von Hamburg sortieren.)

Abgleich mit Lernzielen

1. Zwei Lösungen für **Sortierproblem**.
2. MERGE-SORT schneller als INSERTION-SORT
3. Laufzeit quantifizierbar in $\Theta(\cdot)$.
4. Sortieren ist häufig vorkommendes Problem.
   Für dieses kennen wir jetzt zwei Algorithmen.
5. "Probleme selber lösen, Lösungen beurteilen:"
   Dafür fehlt noch die Übung.

Offene Fragen:

1. Was ist $\Theta(\cdot)$ genau?
2. Wie löst man Rekurrenzen?

Die Antworten kommen in Kürze.

Was fehlt noch:

- Mehr Beispiele und viel Übung.

Wiederholung Wachstumsordnung

- Ein Algorithmus benötigt für jedes Problem eine bestimmte Anzahl von Schritten.
- Die Anzahl der Schritte wird abgeschätzt.
- Die abgeschätzte Anzahl der Schritte wird durch Funktionen beschrieben.
- Wir interessieren uns nicht für Details.
  Bei einer Funktion $an^2 + bn + c$ interessieren uns nicht:
  - Konstanten (a, b, c)
  - Terme, die durch andere subsumiert werden (n durch n^2)
- Uns interessiert nur das asymptotische Verhalten.

Wozu das Ganze?

- Zählen der Schritte, damit wissen wir wie schnell ein Algorithmus ist.
- Zählen der Schritte, damit wir sagen können welcher Algorithmus besser ist.
- Weg mit den Details, damit
  - wir maschinenunabhängig sind
  - damit die Sache einfacher wird:
    * Zählen wird einfacher.
    * Vergleichen wird einfacher.

Funktionen für die Analyse

- Die Eingabegröße geben wir als natürliche Zahl n an.
  Beispiel: Länge der zu sortierenden Folge.
- Die Anzahl der Schritte geben wir als natürliche Zahl $T(n)$ an.
- Also verwenden wir zur Beschreibung der Laufzeit eines Algorithmus Funktionen $T : N \to N$.

Anmerkungen:

- Meistens führen wir eine worst-case-Analyse durch. (Da viel einfacher als mittlerer Fall.)
- Wir benutzen aber auch Funktionen die reelle Zahlen auf reelle Zahlen abbilden.
  Man kann sich vorstellen, daß diese auf natürliche Zahlen eingeschränkt werden, und das Bild mittels $\lceil \cdot \rceil$ auf eine natürliche Zahl abgebildet wird. Dies machen wir jedoch nicht explizit.

Die Θ-Notation

Def.:
Sei $g(n)$ eine Funktion. Wir definieren $\Theta(g(n))$ als eine **Menge von Funktionen**:

$$\begin{aligned}\Theta(g(n)) \;:=\; \{f(n)|\exists c_1, c_2, n_0 > 0 \forall n \geq n_0 \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}\end{aligned}$$

$\square$

Es gilt also $f(n) \in \Theta(g(n))$, falls $f(n)$ zwischen $c_1 g(n)$ und $c_2 g(n)$ eingeklemmt werden kann.

$\Theta(g(n))$ bildhaft:

Vereinbarung/Anmerkungen

- Wir schreiben auch $f(n) = \Theta(g(n))$ statt $f(n) \in \Theta(g(n))$.
- Falls $f(n) = \Theta(g(n))$, so ist $f(n)$ ab einem bestimmten $n > n_0$ nie weiter als einen konstanten Faktor von $g(n)$ entfernt. Falls dies gilt, so sagt man auch, daß $g(n)$ eine asymptotisch enge Grenze für $f(n)$ ist.
- Für $f(n) = \Theta(g(n))$ muß $f(n)$ asymptotisch nicht negativ sein. Für jedes $g(n)$, das nicht asymptotisch nicht negativ ist, ist $\Theta(g(n)) = \emptyset$.
  Daher setzen wir voraus, daß alle $g(n)$, die innerhalb von $\Theta(\cdot)$ auftauchen asymptotisch nicht negativ sind. Das gleiche gilt auch für andere asymptotische Notationen, die wir noch einführen werden.

Macht Θ was es soll?

Motiviert wurde Θ durch weglassen von Konstanten und Termen niedriger Ordnung. Daß dies durchaus richtig ist, demonstrieren wir an einem Beispiel. Wir zeigen:

$$1/2n^2 - 3n = \Theta(n^2)$$

Dazu benötigen wir Konstanten c_1, c_2, n_0 mit

$$c_1 n^2 \leq 1/2n^2 - 3n \leq c_2 n^2$$

für alle $n \geq n_0$. Dividieren durch n^2 ergibt

$$c_1 \leq 1/2 - 3/n \leq c_2$$

- Die rechte Gleichung gilt (bspw) für alle $n \geq 1$, falls $c_2 \geq 1/2$.
- Die linke Gleichung gilt (bspw) für alle $n \geq 7$, falls $c_1 \leq 1/14$.

Mit $c_1 = 1/14$, $c_2 = 1/2$ und $n_0 = 7$ haben wir also bewiesen, daß $1/2n^2 - 3n = \Theta(n^2)$.

Macht Θ was es soll?

Wir zeigen jetzt:

$$6n^3 \neq \Theta(n^2)$$

Annahme: $\exists c_2, n_0$, so daß

$$6n^3 \leq c_2 n^2$$

für alle $n \geq n_0$. Daraus folgt aber

$$n \leq c_2/6$$

Das kann aber nicht sein, da n beliebig groß werden kann und c_2 eine Konstante sein muß. Widerspruch.

Allgemeineres Beispiel

Für

$$f(n) = an^2 + bn + c$$

gilt

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

Dazu kann man bspw. die Konstanten

$$\begin{aligned} c_1 &= a/4 \\ c_2 &= 7a/4 \\ n_0 &= 2 * \max(|b|/a, \sqrt{|c|/a}) \end{aligned}$$

benutzen. (s. Übung)

Polynome

Satz: Sei
$$\sum_{i=0}^{d} a_i n^i$$
ein Polynom mit Konstanten a_i, wobei $a_d > 0$. Dann gilt
$$\sum_{i=0}^{d} a_i n^i = \Theta(n^d)$$
$\square$

(Beweis s. Übung)

Vereinbarung:

> Jede Konstante ist ein Polynoms n^0 0-ten Grades. Wir schreiben jedoch statt $\Theta(n^0)$ lieber $\Theta(1)$. Dabei identifizieren wir die Konstante 1 mit der Funktion, deren Funktionswerte konstant 1 sind.

Obere und untere Schranken

Θ klemmte eine Funktion nach oben und unten bis auf eine Konstante ein.

Manchmal ist das ein bischen viel. Daher führen wir noch Notationen ein für

- obere Schranken
- untere Schranken

Die O-Notation

sprich:

- groß Ohhh

O

- wird für asymptotische obere Schranken benutzt
- das ist nützlich für einfache worst-case Abschätzungen

Def.:

$$\begin{aligned} O(g(n)) \;=\; & \{f(n) | \exists c, n_0 \forall n \geq n_0 \\ & 0 \leq f(n) \leq c g(n)\} \end{aligned}$$

□

Die O-Notation

Anmerkungen:

- wir schreiben auch wieder $f(n) = O(g(n))$
- es gilt $\Theta(g(n)) \subseteq O(g(n))$
- Implikation: $an^2 + bn + c = O(n^2)$

$O(g(n))$ bildhaft:

Beispiel

Für

$$f(n) = an + b$$

gilt

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

Zu finden sind c, n_0 mit

$$0 \leq an + b \leq cn^2$$

Für $c = a + |b|$ gilt

$$0 \leq an + b \leq cn^2$$

für alle $n \geq 1$, da

$$\begin{aligned} an + b &\leq an + |b| \\ &\leq an + |b|n \\ &\leq (a + |b|)n \\ &\leq (a + |b|)n^2 \end{aligned}$$

für $n \geq 1$.

Anwendung von O

Der Algorithmus INSERTION-SORT hatte die Form

for $j \leftarrow$ **2 to** $length(A)$
  $key \leftarrow A[j]$
   while $i > 0$ **and** $a[i] > key$
    $A[i+1] \leftarrow A[i]$
    $i \leftarrow i - 1$
   $A[i+1] \leftarrow key$

Wir beobachten:

- Die beiden Schleifen werden jeweils maximal n mal durchlaufen (eigentlich viel weniger oft).
- Die beiden Schleifen sind geschachtelt.

Dies impliziert sofort für die Laufzeit $T(n)$:

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

Manchmal ist man schon mit so einem Ergebnis zufrieden.

Anmerkung:

> Es ist technisch gesehen ein Fehler zu sagen, daß die Laufzeit von INSERTION-SORT $O(n^2)$ ist, da die Laufzeit noch von der konkreten Eingabefolge abhängt.

Noch deutlicher:

- Die Laufzeit von INSERTION-SORT ist keine Funktion von n!!!

Wir werden jedoch über solche Details hinwegsehen.

Wie:

- Für die worst-case-Analyse können das Maximum aller Laufzeiten für alle folgen der Länge n als $T(n)$ nehmen.

Damit sind wir aus dem Schneider: Wir haben eine Funktion in n.

Die Ω-Notation

So wie O eine obere Schranke ist, ist Ω eine untere Schranke.

Def.

$$\begin{aligned}\Omega(g(n)) \;=\; \{f(n) | \exists c, n_0 \forall n \geq n_0 \\ 0 \leq cg(n) \leq f(n)\}\end{aligned}$$

Bildhaft:

Der erste wichtige Satz

Theorem 2.1:
Für je zwei Funktionen $f(n)$ und $g(n)$ gilt:

$f(n) = \Theta(g(n)) \prec\succ f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$

(Beweis: s. Übung)

Anwendung des Theorems: asymptotisch enge Schranken aus unteren und oberen Schranken gewinnen.

Manchmal geht es jedoch nicht so gut: INSERTION-SORT:

- best-case: $T(n) = \Omega(n)$
- worst-case: $T(n) = O(n^2)$

Verabredung: Wenn wir sagen die Laufzeit eines Algorithmus ist $\Omega(f(n))$, meinen wir den best-case. Analog für $O(f(n))$.

Asymptotische Notationen in Gleichungen

In der Einleitung haben wir bereits asymptotische Notationen in Gleichungen verwendet. Bsp.:

- $T(n) = \Theta(n) + \Theta(1)$
- $T(n) = 2T(n/2) + \Theta(n)$
- $n = O(n^2)$

Was ist die Semantik?

Asymptotische Notationen in Gleichungen

Für einen Fall haben wir es schon definiert:

- Falls auf der linken Seite der Gleichung eine Funktion steht, und auf rechten asymptotische Notation, so ist dies als Mengenzugehörigkeit zu interpretieren.

Ansonsten interpretieren wir eine asymptotische Notation auf der rechten Seite einer Gleichung als eine anonyme Funktion f, die wir nicht bezeichnet haben, und die in der entsprechenden Klasse enthalten ist.

Beispiel:

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

interpretieren wir als

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

für $f(n) \in \Theta(n)$.

(Dabei ist $f(n)$ implizit Existenzquantifiziert.)

Asymptotische Notationen in Gleichungen

Wichtig:

- Jedes Vorkommen einer asymptotischen Notation bedeutet eine originäre Funktion, selbst wenn der Ausdruck gleich sein sollte.

Extremes Beispiel: In

$$\sum_{i=1}^{n} O(i)$$

bezeichnet $O(i)$ **eine** Funktion.

Damit ist dieser Ausdruck ungleich von folgendem Ausdruck:

$$O(1) + O(2) + O(3) + \ldots + O(n)$$

Asymptotische Notationen in Gleichungen

Manchmal kommen asymptotische Notationen auf beiden Seiten einer Gleichung vor. Dann interpretieren wir dies wie folgt:

- Für alle möglichen Funktionen, die wir für die asymptotischen Notationen auf der linken Seite einsetzen können, existieren Funktionen, die wir für die asymptotischen Notationen auf der rechten Seite einsetzen können, so daß die resultierende Gleichung gilt.

Beispiel:

$$2n^2 + \Theta(n) = \Theta(n^2)$$

wird interpretiert als:

Für alle $f(n) \in \Theta(n)$ *existiert ein* $g(n) \in \Theta(n^2)$, *so daß*

$$2n^2 + f(n) = g(n)$$

gilt.

Gleichungsketten

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Dies resultiert in zwei Gleichungen.

Die erste Gleichung besagt, daß es eine Funktion $f(n) \in \Theta(n)$ gibt mit

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

Die zweite Gleichung besagt, daß es für jedes $f(n) \in \Theta(n)$ ein $g(n) \in \Theta(n^2)$ gibt mit

$$2n^2 + f(n) = g(n)$$

Diese beiden Aussagen implizieren:

Es gibt ein $g(n) \in \Theta(n^2)$ mit

$$2n^2 + 3n + 1 = g(n)$$

Dies ist genau das, was wir intuitiv erwarten würden.

Die o-Notation

Betrachte:

$$\begin{aligned} n &= O(n^2) \\ 2n^2 &= O(n^2) \end{aligned}$$

Die zweite Gleichung ist eng, die erste ist es nicht. o erfaßt nun nur die nicht engen asymptotischen oberen Schranken:

$$\begin{aligned} o(g(n)) &= \{f(n) | \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 \\ & \quad 0 \leq f(n) \leq cg(n)\} \end{aligned}$$

Man beachte, daß c gegen 0 gehen kann. $f(n) = o(g(n))$ impliziert also

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Die ω-Notation

Analog definieren wir die nicht engen unteren Schranken:

$$\begin{aligned} \omega(g(n)) &= \{f(n) | \forall c > 0 \exists n_0 > 0 \forall n > n_0 \\ & 0 \leq cg(n) < f(n)\} \end{aligned}$$

Beispiel:

$$\begin{aligned} n^2/2 &= \omega(n) \\ n^2/2 &\neq \omega(n^2) \end{aligned}$$

Die Definition von ω impliziert für $f(n) = \omega(g(n))$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Eigenschaften der asymptotischen Notationen

Viele Eigenschaften, die beispielsweise für die reellen Zahlen gelten, gelten auch für die asymptotischen Notationen:

Reflexivität:

$$\begin{aligned} f(n) &= \Theta(f(n)) \\ f(n) &= O(f(n)) \\ f(n) &= \Omega(f(n)) \end{aligned}$$

Symmetrie:

$$f(n) = \Theta(g(n)) \ \prec\!\succ\ g(n) = \Theta(f(n))$$

Eigenschaften der asymptotischen Notationen

Transitivität:

$$f(n) = \Theta(g(n)), g(n) = \Theta(h(n)) \succ f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)), g(n) = O(h(n)) \succ f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)), g(n) = \Omega(h(n)) \succ f(n) = \Omega(h(n))$$

$$f(n) = o(g(n)), g(n) = o(h(n)) \succ f(n) = o(h(n))$$

$$f(n) = \omega(g(n)), g(n) = \omega(h(n)) \succ f(n) = \omega(h(n))$$

Antisymmetrie:

$$f(n) = O(g(n)) \prec\succ g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \prec\succ g(n) = \omega(f(n))$$

Analogien zu den Reellen Zahlen

Wegen dieser Eigenschaften, kann man folgende (hinkende, siehe unten) Analogie aufstellen:

$$f(n) = O(g(n)) \;\approx\; a \le b$$

$$f(n) = \Omega(g(n)) \;\approx\; a \ge b$$

$$f(n) = \Theta(g(n)) \;\approx\; a = b$$

$$f(n) = o(g(n)) \;\approx\; a < b$$

$$f(n) = \omega(g(n)) \;\approx\; a > b$$

Eine Eigenschaft gilt jedoch nicht:

> Für je zwei reelle Zahlen a und b hat man $a = b$, $a < b$, oder $a > b$.

Es kann aber sein, daß für zwei verschiedene Funktionen $f(n)$ und $g(n)$ weder $f(n) = O(g(n))$ noch $f(n) = \Omega(g(n))$ gilt. Beispiel: n und $n^{1+\sin(n)}$.

Was brauchen wir noch bevor es richtig losgeht?

Das nächste Kapitel ist über

- Sortieren

Dazu brauchen wir

- Bäume.

Für Bäume brauchen wir

- Graphen.

Für Graphen brauchen wir

- Relationen.

Für Relationen brauchen wir

- Mengen.

Die setze ich als bekannt voraus.

Relationen (Def.)

Def.: Eine **binäre Relation** R zweier Mengen A und B ist eine Teilmenge des kartesischen Produkts $A \times B$, also

$$R \subseteq A \times B.$$

$\square$

- Für $(a, b) \in R$ schreiben wir auch aRb.
- eine binäre Relation R über A ist eine Teilmenge von $A \times A$.
- Beispiele über den natürlichen Zahlen: $=$, $\leq$ $<$.
  Letzteres:
  $<= \{(a, b) | a, b \in N, a < b\}$

Def.: Eine n-äre Relation über den Mengen $A_1, \ldots, A_n$ ist eine Teilmenge von $A_1 \times \ldots \times A_n$. $\square$

Relationen (Eigenschaften)

Sei R eine binäre Relation $R \subseteq A \times A$.

R heißt **reflexiv**, falls $\forall a \in A \quad aRa$

R heißt **symmetrisch**, falls $aRb \quad \succ \quad bRa$

R heißt **transitiv**, falls $aRb, bRc \quad \succ \quad aRc$

Ist R alles drei, so heißt R **Äquivalenzrelation**.

Beispiele:

- $=$ und $\leq$ sind Äquivalenzrelationen.
- $<$ ist nicht reflexiv

Jede Äquivalenzrelation partitioniert A in $n \geq 0$ disjunkte Teilmengen (**Partitionen**).

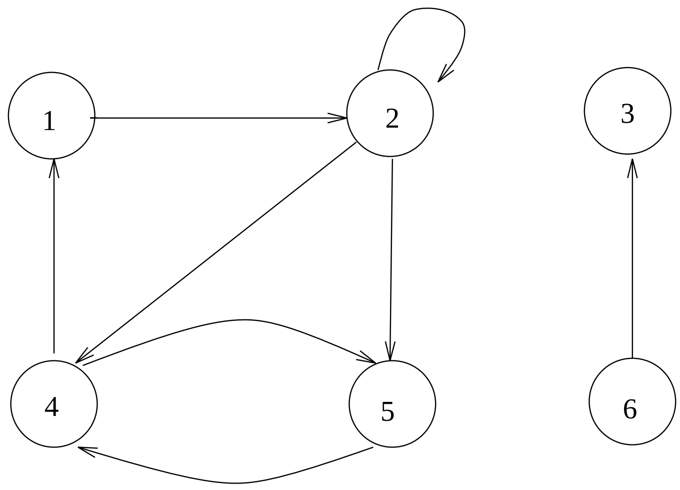
Gerichtete Graphen

Def. Ein **gerichteter Graph** ist ein Paar (V, E), wobei V eine endliche Menge ist und E eine binäre Relation über V, also $E \subseteq V \times V$.

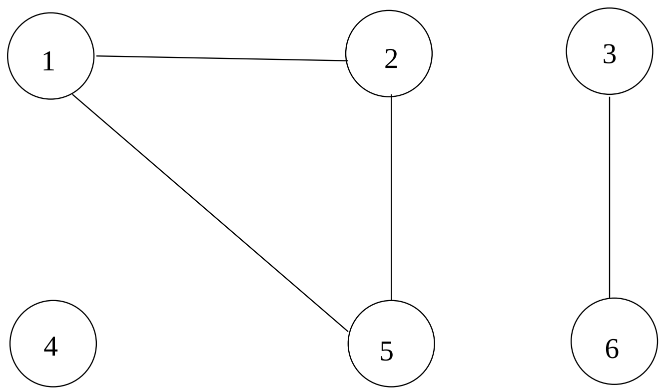
Die Elemente aus V heißen **Knoten**, die aus E **Kanten**.

Beispiel:

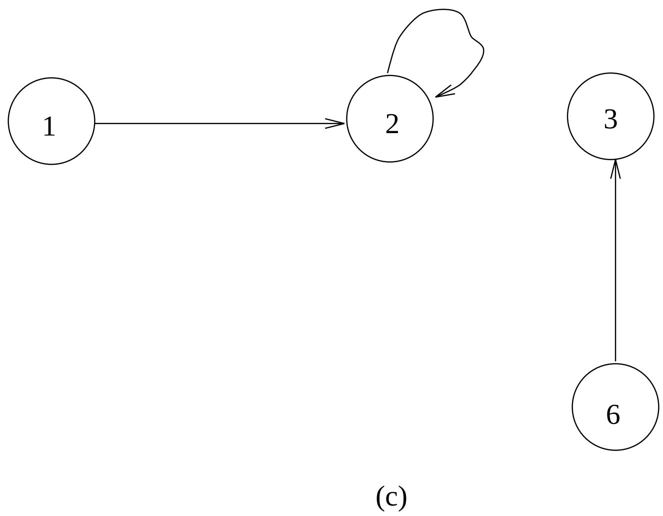
- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(4, 1), (1, 2), (2, 2), (2, 4), (4, 5), (5, 4), (3, 6)\}$



(a)



(b)



(c)

Ungerichtete Graphen

Def. Ein **ungerichteter Graph** ist ein Paar (V, E), wobei V eine endliche Menge ist und E eine ungerichtete Menge von Paaren (u, v) mit $u, v \in V$ und $u \neq v$ ist.

Die Elemente aus V heißen **Knoten**, die aus E **Kanten**.

Beispiel:

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$

Allgemeine Sprechweisen bei Graphen

Viele Sprechweisen sind bei gerichteten und ungerichteten Graphen die gleichen, haben aber u.U. leicht unterschiedliche Semantik.

Sei $(u, v) \in E$. Dann sagen wir:

- Es führt eine Kante von u nach v.
- Die Kante verläßt u.
- Die Kante führt zu v.
- Die Kante ist eine von u ausfallende Kante.
- Die Kante ist eine in v einfallende Kante.
- u ist benachbart zu v.
  (bei gerichteten Graphen ist dies nicht notwendig symmetrisch)
- u und v sind durch eine Kante verbunden (bei ungerichteten Graphen).
- Ist $u = v$, so heißt die Kante Schleife (bei gerichteten Graphen).

Grad

Für gerichtete Graphen definieren wir:

- Der In-Grad eines Knotens ist die Anzahl der einfallenden Kanten.
- Der Aus-Grad eines Knotens ist die Anzahl der ausfallenden Kanten.
- Der Grad eines Knotens ist die Summe aus In-Grad und Aus-Grad.

Für ungerichtete Graphen definieren wir:

- Der Grad eines Knotens ist die Menge aller Kanten, die mit dem Knoten verbunden sind.

Pfad

Def.: Ein **Pfad** der **Länge** k von einen Knoten u zu einem Knoten u' in einem Graphen $G = (V, E)$ ist eine Folge

$$\langle v_0, v_1, \ldots, v_k \rangle$$

mit folgenden Eigenschaften:

1. $v_0 = u$
2. $v_k = u'$
3. $(v_{i-1}, v_i) \in E$, für $1 \leq i \leq k$

$\square$

Anmerkung:

- Die Länge eines Pfades ist also die Anzahl der Kanten im Pfad.

Sprechweisen/Definitionen

- Der Pfad **enthält** die Knoten v_i.
- u' ist von u aus **erreichbar**.
- u und u' sind durch einen Pfad verbunden.
- Ein Pfad heißt **einfach**, falls er keinen Knoten mehr als einmal enthält.
- Ein Teilpfad ist eine Teilfolge von $\langle v_0, v_1, \ldots, v_k \rangle$, also eine Folge

  $$\langle v_i, v_{i+1}, \ldots, v_j \rangle$$

  mit $0 \le i \le j \le k$.
- Ein Zyklus ist ein Pfad mit $v_0 = v_k$.
- Ein einfacher Zyklus ist ein Zyklus, bei dem sonst keine zwei Knoten gleich sind.
- Ein Graph mit Zyklus heißt zyklisch, einer ohne Zyklus heißt azyklisch.

Zusammenhang

- Ein ungerichteter Graph heißt **zusammenhängend**, falls je zwei Knoten durch einen Pfad verbunden sind.
- Die **Zusammenhangskomponenten** eines Graphen sind die Äquivalenzklassen unter der **erreichbar**-Relation.
- Ein ungerichteter Graph ist genau dann zusammenhängend, wenn er nur eine Zusammenhangskomponente hat.
- Ein gerichteter Graph heißt **streng zusammenhängend**, falls je zwei Knoten voneinander erreichbar sind.
- Die **strengen Zusammenhangskomponenten** sind die Äquivalenzklassen der **gegenseitig-erreichbar**-Relation.

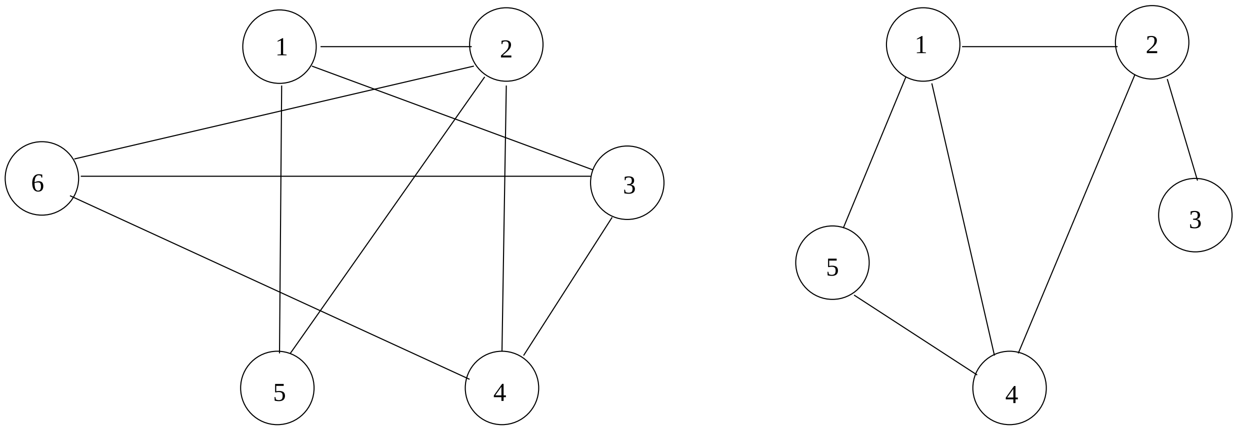
Graphisomorphie

Def.:
Zwei Graphen $G = (V, E)$ und $G' = (V', E')$ heißen **isomorph**, falls es eine Bijektion $f : V \to V'$ gibt, mit

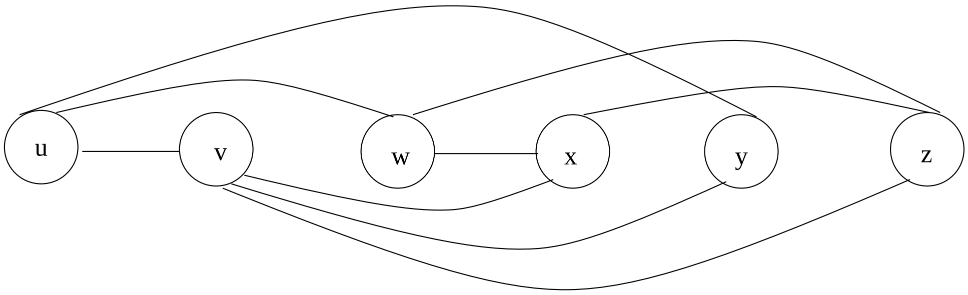
$$(u, v) \in E \prec\succ (f(u), f(v)) \in E'$$

$\square$

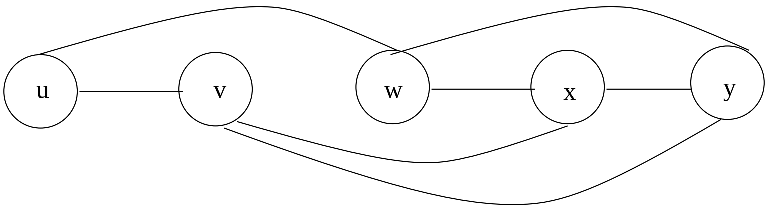
G



G’



(a)



(b)

Teilgraphen

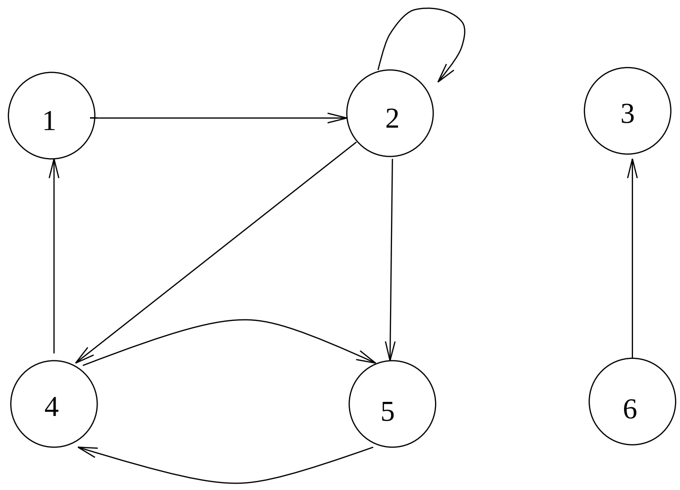
Def.:
Ein Graph $G' = (V', E')$ heißt **Teilgraph** (Unter-, Sub-) eines Graphen $G = (V, E)$, falls

1. $V' \subseteq V$
2. $E' \subseteq E$

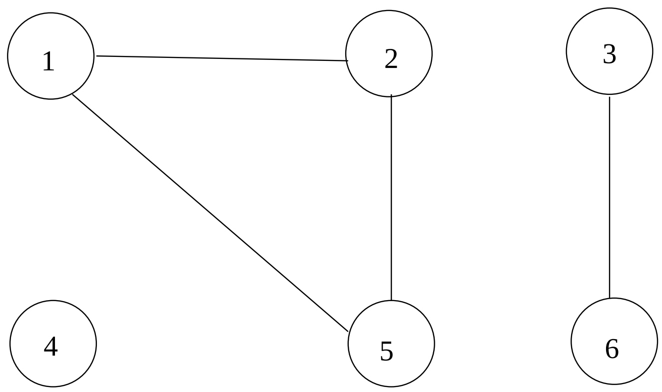
$\square$

Gegeben eine Menge $V' \subseteq V$, dann definiert man den von V' **induzierten** Teilgraphen von $G = (V, E)$ als den Graphen $G' = (V', E')$ mit

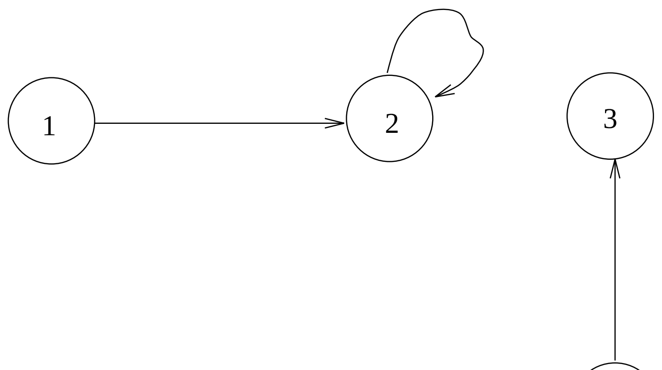
$$E' = \{(u, v) \in E | u, v \in V'\}$$



(a)



(b)



(c)

Richten und Unrichten

Def.:
Sei $G = (V, E)$ ein ungerichteter Graph. Dann heißt $G' = (V, E')$ mit

$$(u, v) \in E' \prec\succ (u, v) \in E$$

die **gerichtete Version** von G.
$\square$

Es wird also jede ungerichtete Kante durch zwei gerichtete ersetzt.

Richten und Unrichten

Def.:
Sei $G = (V, E)$ ein gerichteter Graph. Dann heißt $G' = (V, E')$ mit

$$(u, v) \in E' \prec\succ (u, v) \in E \wedge u \neq v$$

die **ungerichtete Version** von G.
$\square$

Man beachte, daß keine Kante "doppelt" vorkommt, da ja (u, v) und (v, u) in einem ungerichteten Graphen die gleiche Kante sind.

In einem gerichteten Graphen $G = (V, E)$ ist der Knoten u ein **Nachbar** von Knoten v, falls in der ungerichteten Version von G die Knoten u und v benachbart sind.

Spezielle Graphen (Sonderfälle)

- Ein **vollständiger Graph** ist ein ungerichteter Graph, in dem je zwei Knoten benachbart sind.
- Ein **bipartiter Graph** ist ein ungerichteter Graph $G = (V, E)$, in dem V in zwei disjunkte Teilmengen (Partitionen) V' und V'' zerlegt werden kann mit

$$\begin{aligned}(u, v) \in E \succ\ & (u \in V' \land v \in V'') \lor \\ & (u \in V'' \land v \in V')\end{aligned}$$

Spezielle Graphen (Sonderfälle)

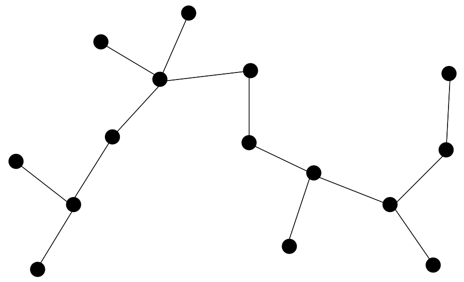
- azyklischer, ungerichteter Graph heißt auch **Wald**.
- verbundener, azyklischer, ungerichteter Graph heißt **(freier) Baum**.
- **Multigraph** ist analog zum ungerichteten Graphen definiert, nur daß er Mehrfachkanten und Schleifen enthalten kann.
- **Hypergraph** ist analog zum ungerichteten Graphen definiert, nur daß eine Kante mehr als zwei Knoten beinhalten kann (also eine beliebige Menge). Eine solche Kante heißt dann **Hyperkante**.

Bäume sind so wichtig, daß wir sie noch ausführlicher behandeln.

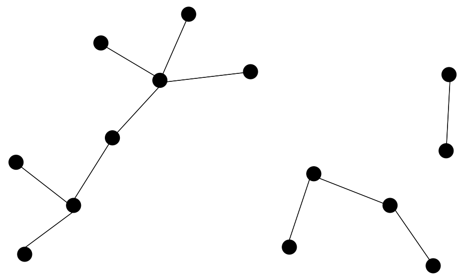
Freie Bäume

Def.: Ein freier Baum ist ein verbundener, azyklischer, ungerichteter Graph.

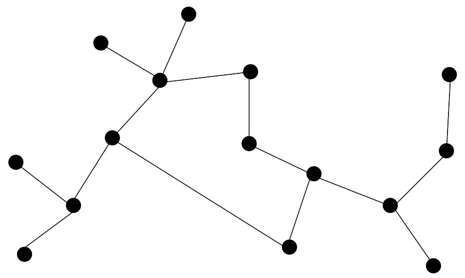
(a)



(b)



(c)



Bäume

Satz: Sei $G = (V, E)$ ein ungerichteter Graph. Dann sind folgende Aussagen äquivalent:

1. G ist ein freier Baum.
2. Je zwei Knoten in G sind durch einen eindeutigen einfachen Pfad verbunden.
3. G ist zusammenhängend, aber falls nur eine Kante entfernt wird, so ist der resultierende Graph nicht mehr zusammenhängend.
4. G ist zusammenhängend und $|E| = |V| - 1$.
5. G ist azyklisch und $|E| = |V| - 1$.
6. G ist azyklisch, aber falls nur eine Kante zu E hinzugefügt wird, so ist der resultierende Graph zyklisch.

Beweis

(1) $\Longrightarrow$ (2): Da ein Baum zusammenhängend ist, sind je zwei Kanten in G durch mindestens einen einfachen Pfad verbunden.

Seien u und v durch zwei einfache Pfade p_1 und p_2 verbunden:

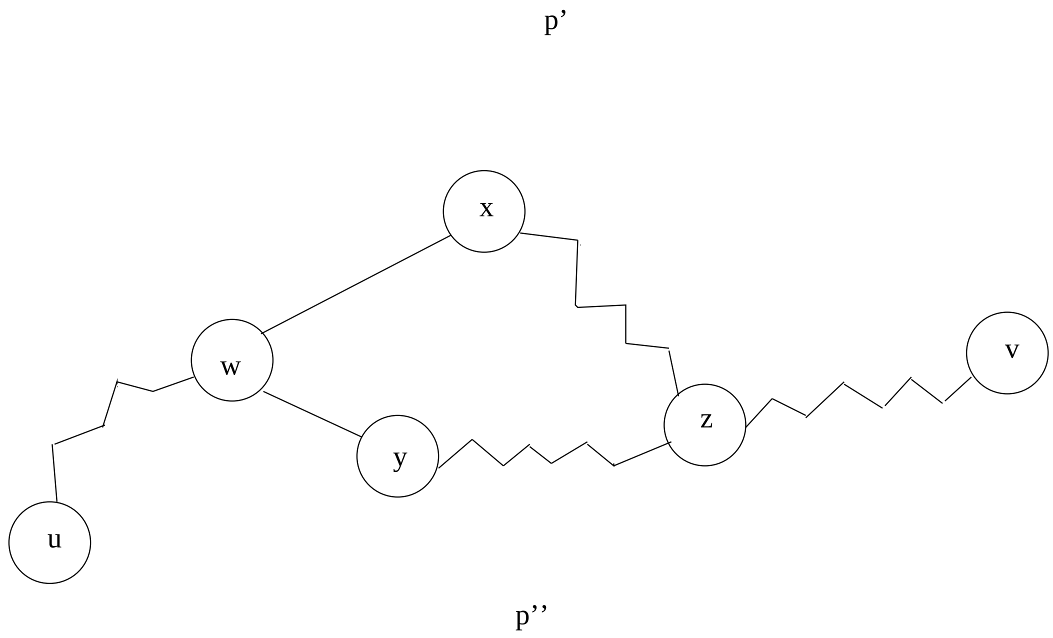
Sei w der Knoten, nach dem sich p_1 und p_2 unterscheiden.

Sei z der erste Knoten, an dem sich p_1 und p_2 wieder treffen.

Seien p' und p'' die Teilpfade von w nach z von p_1 und p_2.

Dann haben p' und p'' nur die Endpunkte gemeinsam.

Daher ist $p'p''$ ein Zyklus.


p'
x
w
v
z
y
u
p''

(2) $\Longrightarrow$ (3): Da je zwei Knoten durch einen einfachen Pfad verbunden sind, ist G verbunden. Da dieser Pfad eindeutig ist, führt das entfernen einer Kante auf diesem Pfad dazu, daß G unverbunden ist.

(3) $\Longrightarrow$ (4): G ist verbunden. Also (Übung) $|E| \geq |V| - 1$. Wir zeigen durch Induktion $|E| \leq |V| - 1$.

I.A.: Ein verbundener Graph mit $n = 1$ oder $n = 2$ Knoten hat $n - 1$ Kanten.

I.S.:
Annahme: G hat $n \geq 3$ Knoten. Jeder Graph mit $< n$ Knoten und (3) erfüllt $|E| \leq |V| - 1$.

Entfernen einer Kante läßt G in $k \geq 2$ Komponenten zerfallen. Jede Komponente erfüllt (3) (sonst würde G nicht (3) erfüllen).

I.H.: Anzahl der Knoten in allen Komponenten ist höchstens $|V| - k \leq |V| - 2$. Plus die entfernte Kante ergibt maximal $|V| - 1$ Kanten.

(4) $\Longrightarrow$ (5): Sei G ist verbunden mit $|E| = |V| - 1$ Kanten.

zu zeigen: G azyklisch.

Annahme: G enthält Zyklus $v_1, \ldots, v_n$.

Sei $G_k = (V_k, E_k)$ der Teilgraph von G, der nur den Zyklus enthält. Dann gilt $|V_k| = |E_k| = k$.

Falls $|V| < k$, gibt es einen Knoten $v_{k+1} \in V \setminus V_k$, mit $(v_{k+1}, v_i) \in E$ für ein $v_i \in V_k$, da G verbunden ist.

Definiere

$$
\begin{aligned}
G_{k+1} &= (V_{k+1}, E_{k+1}) \\
V_{k+1} &= V_k \cup \{v_{k+1}\} \\
E_{k+1} &= E_{k+1} \cup \{({}_{k+1}, v_i)\}
\end{aligned}
$$

Es gilt: $|V_{k+1}| = |E_{k+1}| = k_1$.

Falls $k + 1 < n$, machen wir weiter mit G_{k+2} usw.

Ergebnis: Teilgraph $G_n = (V_n, E_n)$ von G mit $|V_n| = |E_n| = n$. Widerspruch.

(5) $\Longrightarrow$ (6):

Sei G azyklisch mit $|E| = |V| - 1$.

Sei k die Anzahl der Verbundkomponenten von G.

Jede Verbundkomponente ist ein freier Baum (nach Definition). Da (1) $\Longrightarrow$ (5) ist die Summe aller Kanten in allen Komponenten $|V| - k$.

Also muß $k = 1$ gelten und G ist ein Baum.

Da (1) $\Longrightarrow$ (2) sind je zwei Knoten durch einen einfachen Pfad verbunden. Hinzufügen einer Kante erzeugt also einen Zyklus.

(6) $\Longrightarrow$ (1):

Sei G azyklisch, aber jedes Hinzufügen einer Kante erzeuge einen Zyklus.

zu zeigen: G verbunden.

Seien u und v zwei beliebige Kanten in G. Falls u und v nicht bereits benachbart sind, führt das Hinzufügen von (u, v) zu einem Zyklus.

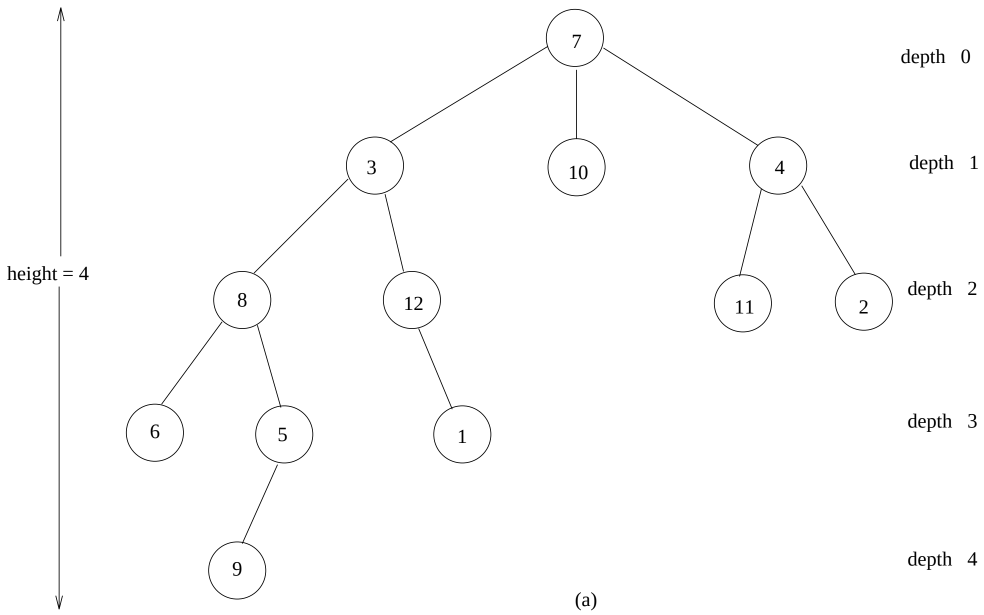
Also gab es bereits einen Pfad von u nach v.

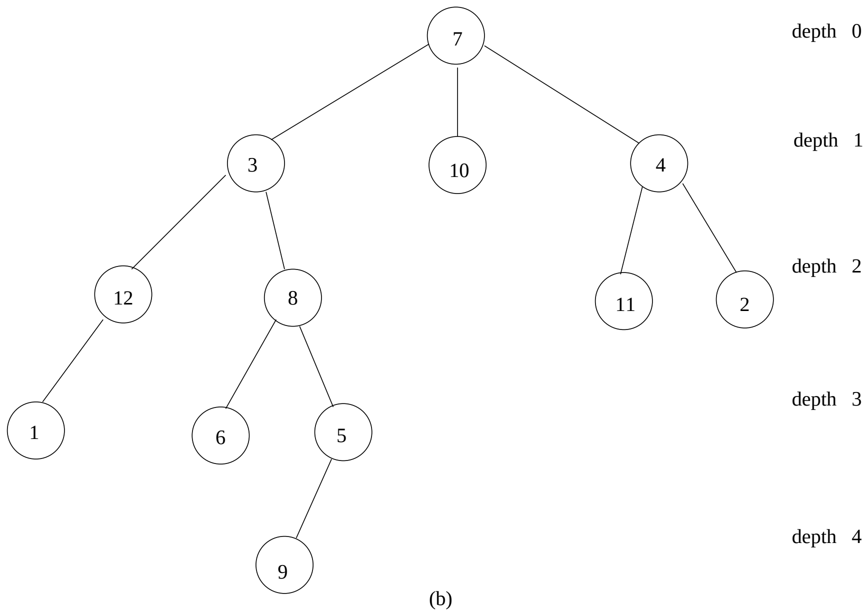
Da u und v beliebig gewählt waren ist G verbunden.

Wurzelbaum (rooted tree)

Def.: Ein **Wurzelbaum** ist ein freier Baum in dem ein Knoten als **Wurzel** ausgezeichnet wird.

Die Wurzel eines Wurzelbaumes (oder kurz Baumes) zeichnen wir nach oben.


height = 4
7
3
10
4
8
12
11
2
6
5
1
9
depth 0
depth 1
depth 2
depth 3
depth 4
(a)


7
3
10
4
12
8
11
2
1
6
5
9
depth 0
depth 1
depth 2
depth 3
depth 4
(b)

Baum

Betrachte einen Knoten x in einem Baum T mit Wurzel r.

- Jeder Knoten y auf einem Pfad von r zu x heißt **Vorgänger** (ancestor) von x.
- Falls y ein Vorgänger von x ist, so heißt x Nachfolger von y.
- Jeder Knoten ist Vorgänger und Nachfolger von sich selbst.
- Ist $x \neq y$, so sprechen wir von echten Vorgängern und Nachfolgern.
- Der **Teilbaum** des Knotens x ist der von den Nachfolgern von x induzierte Teilbaum von T.

Baum

- Falls die letzte Kante vom Pfad von r nach x (y, x) ist, so heißt y **Vaterknoten** (parent) von x und x heißt **Sohnknoten** (child) von y.
- Falls x und z den gleichen Vaterknoten haben, so heißen sie **Brüder** (sibling).
- Die Wurzel ist der einzige Knoten in einem Baum ohne Vaterknoten.
- Knoten ohne Sohnknoten heißen **äußere** Knoten oder **Blattknoten**.
- Alle anderen Knoten heißen **innere Knoten**.

Baum

- Die Anzahl der Sohnknoten eines Knotens x heißt **Grad** von x.
- Die Länge des Pfades von r nach x heißt Tiefe (Ebene, Level) von x.
- Die maximale Tiefe der Knoten in T heißt Höhe von T.

Baum

Def.: Ein **geordneter Baum** ist ein Wurzelbaum, bei dem die Sohnknoten geordnet sind.

Das heißt also, daß wir vom 1., 2., bis zum k-ten Sohn sprechen können.

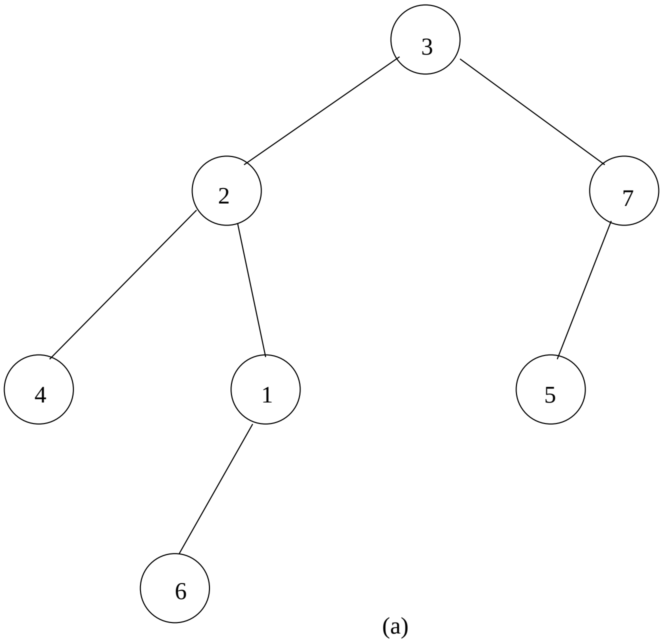
Binärbaum

Def.: Ein **Binärbaum** T ist über einer endlichen Menge N von Knoten wie folgt definiert:

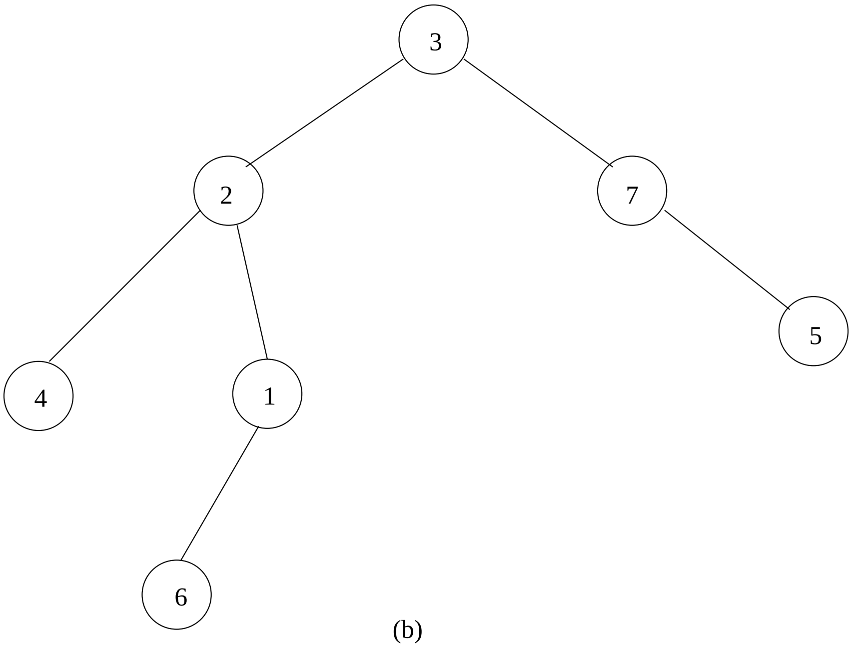
- N ist leer, oder
- N enthält drei disjunkte Teilmengen: einen Wurzelknoten r, und zwei Binärbäume L und R, die auch linker und rechter Teilbaum von T genannt werden.

Falls N leer ist, so heißt T auch **leerer Baum**. Wir bezeichnen ihn mit `NIL`.

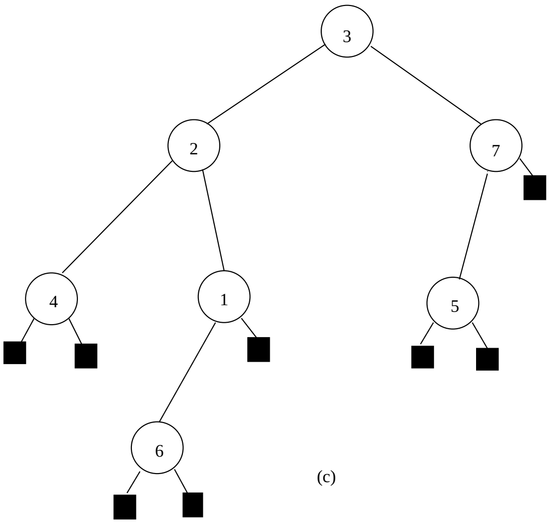
- Für einen Knoten x sprechen wir von **rechtem und linkem Sohn** für die Wurzeln nichtleerer linker und rechter Teilbäume.
- In einem **vollständiger Binärbaum** hat jeder Knoten entweder 2 oder 0 Söhne.



(a)



(b)



(c)

Binärbaum

Beachte:

- Ein Binärbaum ist ein bischen mehr als ein geordneter Baum mit Gradbeschränkung 2 für jeden Knoten:
  - Selbst wenn nur ein Sohn vorhanden ist, so ist dieser entweder linker oder rechter Sohn.
- Also können zwei verschiedene Binärbäume den gleichen geordneten Baum darstellen.

Positionsbäume

Die Art der Positionierung, die bei Binärbäumen benutzt wurde, kann auf beliebige Anzahl von Sohnknoten verallgemeinert werden. Dies führt zu Positionsbäumen oder k-ären Bäumen.

Nicht-Blattknoten in vollständige k-äre Bäumen haben genau k Nachfolger.

Positionsbäume

Die Anzahl der Knoten in einem vollständigen k-ären Baum der Höhe h bestimmt sich wie folgt:

$$
\begin{aligned}
1 + k + k^2 + k^3 + \ldots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\
&= \frac{k^h - 1}{k - 1}
\end{aligned}
$$

Vollständiger Binärbaum ($h = 2$)