WS 2008/09

Algorithmen und Datenstrukturen (EI)

Ernst W. Mayr

Fakultät für Informatik
TU München

http://wwwmayr.in.tum.de/lehre/2008WS/ads-ei/

Wintersemester 2008/09

Kapitel 0 Organisatorisches

1. Termine

- Vorlesung:
  - 4SWS Mo 09:15–10:45, Do 08:15–09:45 (HS 1200)
    Pflichtvorlesung B.Sc. Elektro- und Informationstechnik
- Übung:
  - 2SWS Zentralübung: Mi 10:30–12:00 (HS 1200 Carl von Linde-Hörsaal)
  - Übungsleitung: Dr. Stefan Schmid, Dmytro Chibisov
- Umfang:
  - 4V+2ZÜ, 6 ECTS-Punkte
- Programmierpraktikum C:
  - 2SWS Praktikum: Mi 14:00–16:00 (Audimax)
  - Leitung: W. Bamberger, M. Durkovic

- Übungsleitung:
  - Dr. Stefan Schmid, MI 03.09.057 (schmiste@in.tum.de)
    Sprechstunde: Freitag, 16:00Uhr
  - Dmytro Chibisov, MI 03.09.041 (chibisov@in.tum.de)
    Sprechstunde: Freitag, 16:00Uhr
- Sekretariat:
  - Frau Lissner, MI 03.09.052 (lissner@in.tum.de)
- Sprechstunde:
  - nach Vereinbarung

- Hausaufgaben:
  - Ausgabe jeweils am Donnerstag in der Vorlesung bzw. auf der Webseite der Vorlesung
  - Besprechung in der Zentralübung in der Woche darauf
  - vorauss. 13 Hausaufgabenblätter, das letzte am 22. Januar 2009
- Klausur:
  - Klausur (120min, Termin: 17. Februar 2009, 11:00Uhr)
  - bei der Klausur sind *keine* Hilfsmittel außer einem handbeschriebenen DIN-A4-Blatt zugelassen
- Leistungsnachweis:
  - erfolgreiche Teilnahme an Klausur und Praktikum

- Vorkenntnisse:
  - Mathematische Grundkenntnisse aus der Schule
  - Neugier!
- Weiterführende Vorlesungen:
  - Computertechnik
  - zahlreiche Module im Vertiefungsteil
- Webseite:

  http://www14.in.tum.de/lehre/2008WS/ads-ei/

Geplante Themengebiete

1. Boolesche Algebra und Logik
2. Automatentheorie
3. Formale Sprachen und Grammatiken
4. Entwurf und Analyse von Algorithmen
5. Datenstrukturen
6. Suchen und Sortieren
7. Algorithmen auf Graphen

2. Literatur


Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:
The design and analysis of computer algorithms,
Addison-Wesley Publishing Company: Reading (MA), 1974


Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
Introduction to algorithms,
McGraw-Hill, 1990


Hartmut Ernst:
Grundkurs Informatik: Grundlagen und Konzepte für die erfolgreiche IT-Praxis — Eine umfassende, praxisorientierte Einführung,
Vieweg Verlag: Braunschweig-Wiesbaden, 3. Auflage, 2003


Volker Heun:
Grundlegende Algorithmen: Einführung in den Entwurf und die Analyse effizienter Algorithmen,
2. Aufl., Vieweg: Braunschweig-Wiesbaden, 2003


John E. Hopcroft, Jeffrey D. Ullman:
Introduction to Automata Theory, Languages, and Computation,
Addison-Wesley Publishing Company: Reading (MA), 1979


Donald E. Knuth:
The art of computer programming. Vol. 1: Fundamental algorithms,
3. Auflage, Addison-Wesley Publishing Company: Reading (MA), 1997

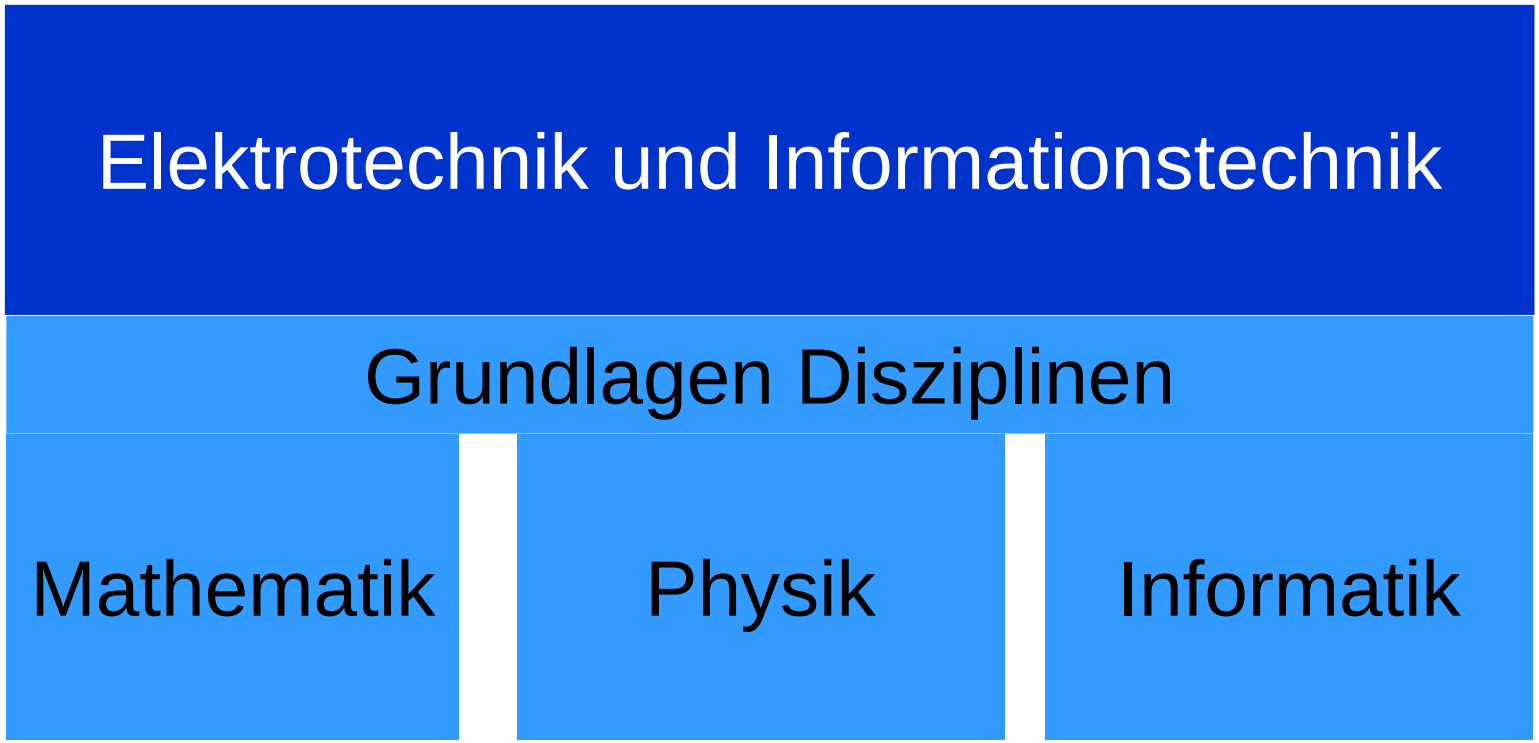

Kurt Mehlhorn, Peter Sanders:
Algorithms and Data Structures — The Basic Toolbox,
Springer-Verlag: Berlin-Heidelberg, 2008


Steven S. Skiena:
The algorithm design manual,
Springer-Verlag: Berlin-Heidelberg-New York, 1998

Weitere Originalarbeiten und Texte werden im Verlauf der Vorlesung angegeben.

Kapitel I Historisches und Begriffliches

1. EI, Informatik und ihre Struktur



Was ist Informatik?

- Die Wikipedia sagt:
  "Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, insbesondere der automatischen Verarbeitung mit Hilfe von Rechenanlagen"
  . . . mehr
- Die Gesellschaft für Informatik (GI) sieht die Informatik als
  - Grundlagenwissenschaft,
  - Ingenieurdisziplin, und als
  - Experimentalwissenschaft

  Mehr dazu hier!

Einige Teilgebiete der Informatik

- **Theoretische Informatik**
  Formale Sprachen, Automatentheorie, Komplexitätstheorie, Korrektheit und Berechenbarkeit, Algorithmik, Logik
- **Praktische Informatik**
  Betriebssysteme, Compiler, Datenbanken, Software-Engineering, Software-Entwicklung, ...
- **Technische Informatik**
  Digitale Schaltungen, Hardware-Komponenten, Mikroprogrammierung, Rechnerarchitekturen, Netzwerke, ...
- **Angewandte Informatik**
  Anwendungen von Informationsverarbeitung in Verwaltung, Büro, Fertigung, Medizin, (Molekular-) Biologie, Modellierung, Visualisierung, ...

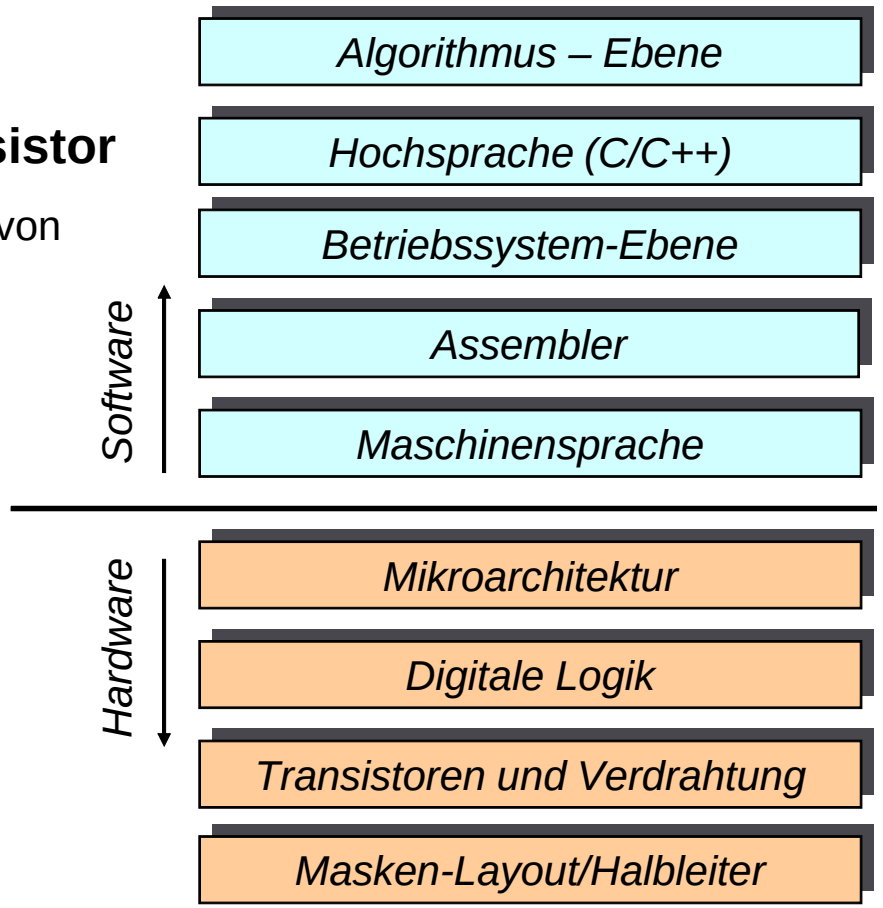
Ziele von Vorlesung und Praktikum

1. Kenntnis der Werkzeuge und Methoden der Informatik . . . , die für einen Ingenieur praxisrelevant sind
2. Methodenwissen
   - zum Entwurf von effizienten Algorithmen
   - und deren Implementierung
3. Erste praktische Erfahrungen im Umgang mit der Programmiersprache C

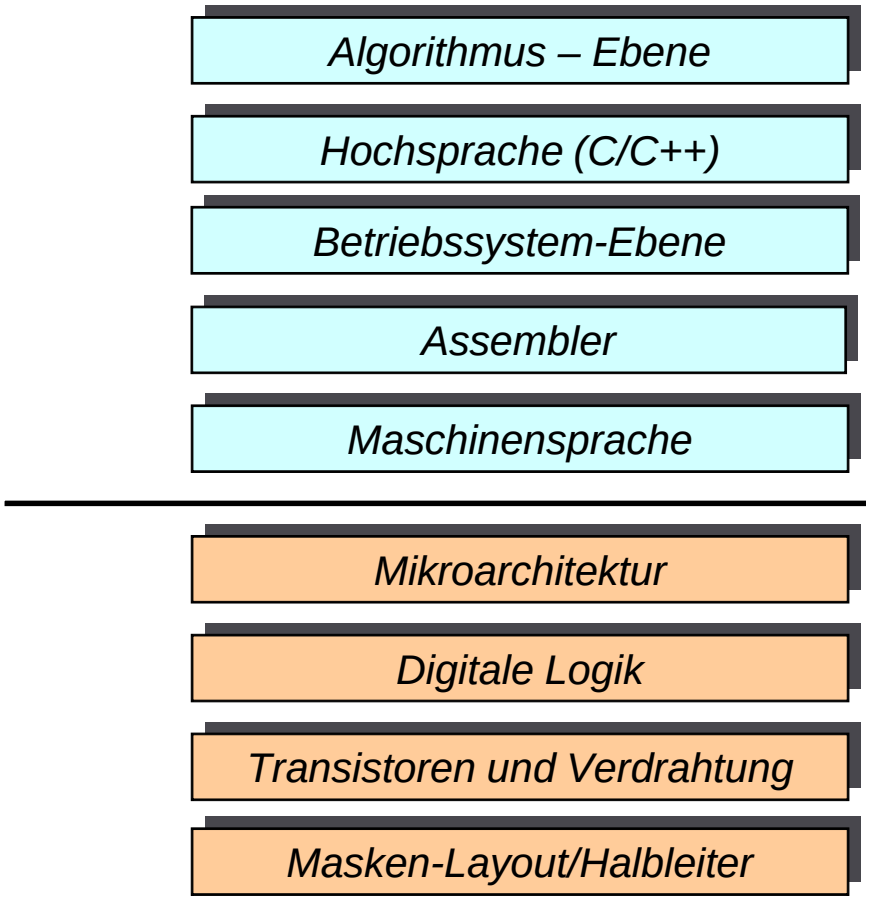
2. Abstraktionsebenen

- **Von der Formel zum Transistor**

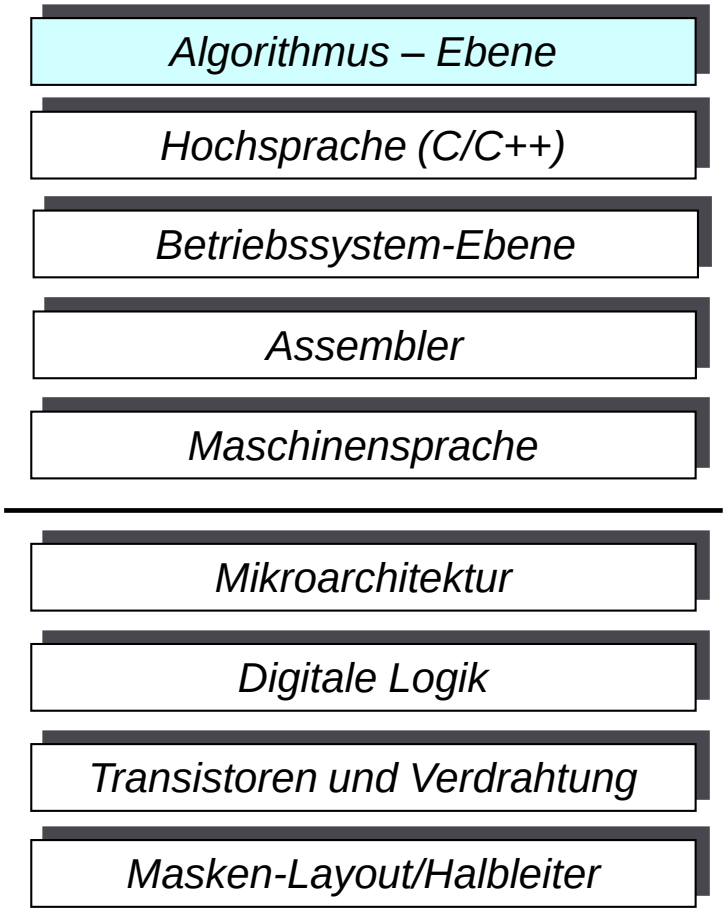
  Eine Reise durch die Hierarchie von Abstraktions-Ebenen



Abstraktionsebenen



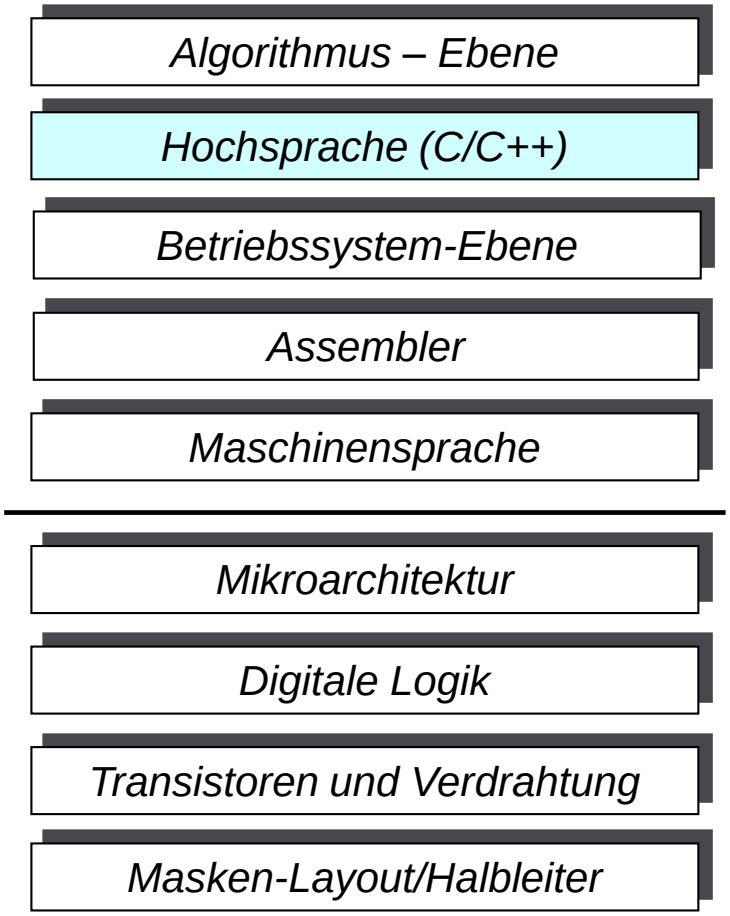
Abstraktionsebenen



Skalarprodukt (Mathematik)

$$\mathbf{a}^T \cdot \mathbf{b} = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \sum_{i=1}^{4} a_i \cdot b_i$$

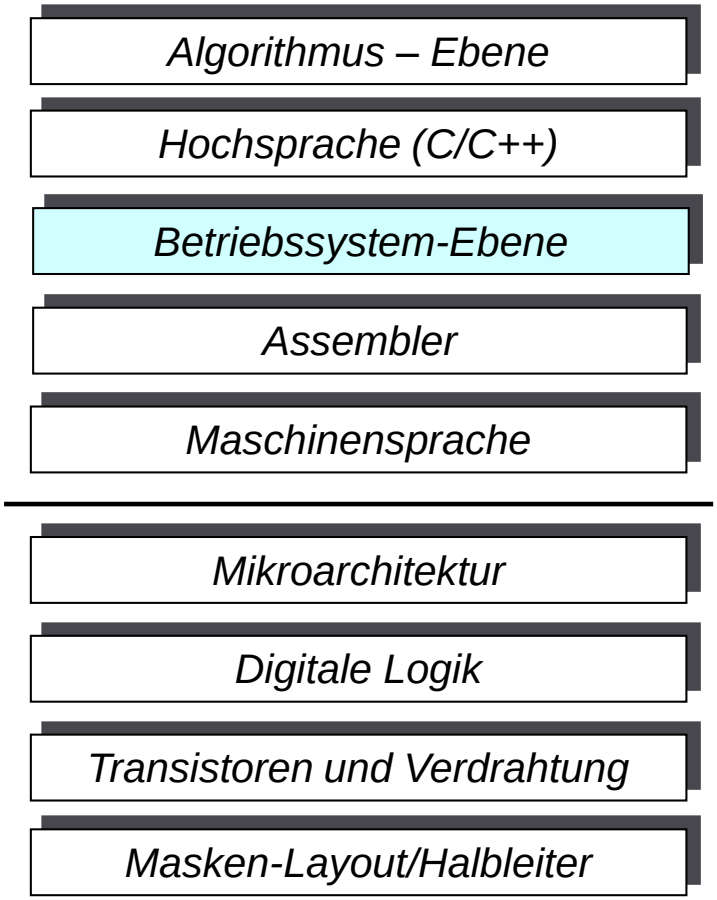
Abstraktionsebenen



C-Programm

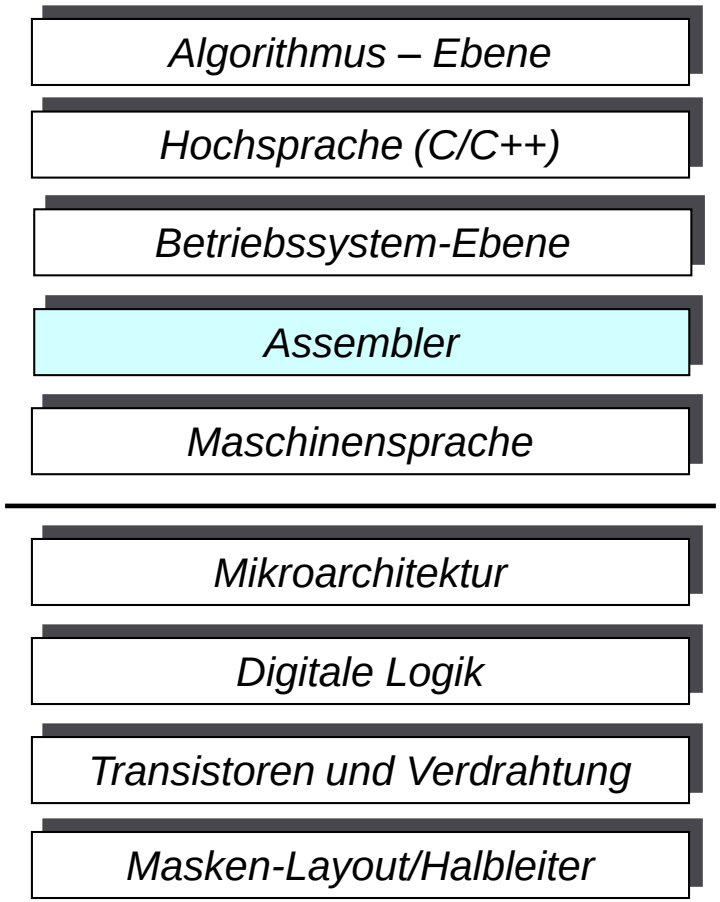
```
void main()
{
  int w, x;
  const int n = 4; % Dimens.
  int a[n] = {1,2,3,4};
  int b[n] = {10,20,30,40};

  w = n;
  x = 0;
  while(w){
     w = w - 1;
     x = x + a[w] * b[w];
  }
  %  in x steht der Wert des
  %  Skalarproduktes
}
```

Abstraktionsebenen



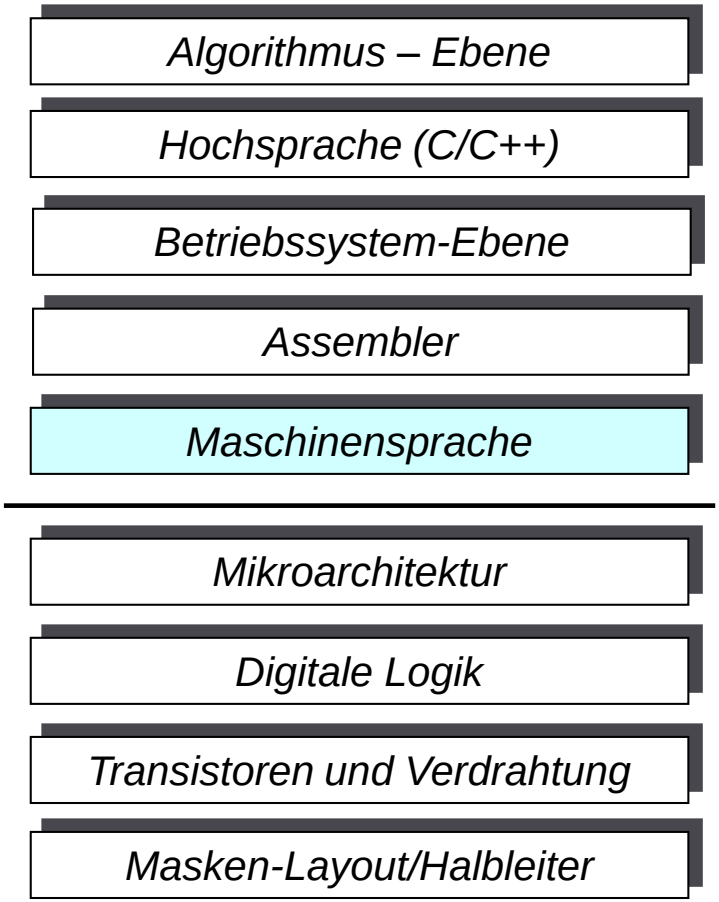
- **Verwaltung und Zuweisung von Ressourcen**
- **Laden des Programms**
- **Speicherverwaltung**
- **Zugriff auf Festplatte, Drucker, etc.**
- **Multi-Tasking – Zuteilung der CPU**
- **:**
- **:**

Abstraktionsebenen




```
            LOC       Data_Segment
            GREG      @
N           OCTA      4          Vektordimension
ADR_A1      OCTA      1          a1
            OCTA      2          a2
            OCTA      3          a3
            OCTA      4          a4
ADR_B1      OCTA      10         b1
            OCTA      20         b2
            OCTA      30         b3
            OCTA      40         b4
u           IS        $1         Parameter 1
v           IS        $2         Parameter 2
w           IS        $3         Parameter 3
x           IS        $4         Ergebnis
y           IS        $5         Zwischenberech.
z           IS        $6         Zwischenberech.
            LOC       #100
            GREG      @
Main        SETL      x,0        Initialisierung
            LDA       u,ADR_A1   Ref. a1 in u
            LDA       v,ADR_B1   Ref. B1 in v
            LDO       w,N        Lade Vektordim.
            MUL       w,w,8      8Byte Wortlänge
Start       BZ        w,Ende     wenn fertig End
            SUB       w,w,8      w=w-8
            LDO       y,u,w      y=<u+w>
            LDO       z,v,w      z=<v+w>
            MUL       y,y,z      y=<u+w>*<v+w>
            ADD       x,x,y      x=x+<u+w>*<v+w>
            JMP       Start
Ende        TRAP      0,:HALT,0  Ergebnis in x
```

Abstraktionsebenen




```
0x0000000000000100: e3040000
0x0000000000000104: 2301fe08
0x0000000000000108: 2302fe28
0x000000000000010c: 8d03fe00
0x0000000000000110: 19030308
0x0000000000000114: 42030007
0x0000000000000118: 25030308
0x000000000000011c: 8c050103
0x0000000000000120: 8c060203
0x0000000000000124: 18050506
0x0000000000000128: 20040405
0x000000000000012c: f1fffffa
0x0000000000000130: 00000000
 .....              ...
0x2000000000000000: 00000004
0x2000000000000004: 00000001
0x2000000000000008: 00000002
0x200000000000000c: 00000003
0x2000000000000010: 00000004
0x2000000000000014: 0000000a
0x2000000000000018: 00000014
0x200000000000001c: 0000001e
0x2000000000000020: 00000028
```

Abstraktionsebenen

Algorithmus – Ebene

Hochsprache (C/C++)

Betriebssystem-Ebene

Assembler

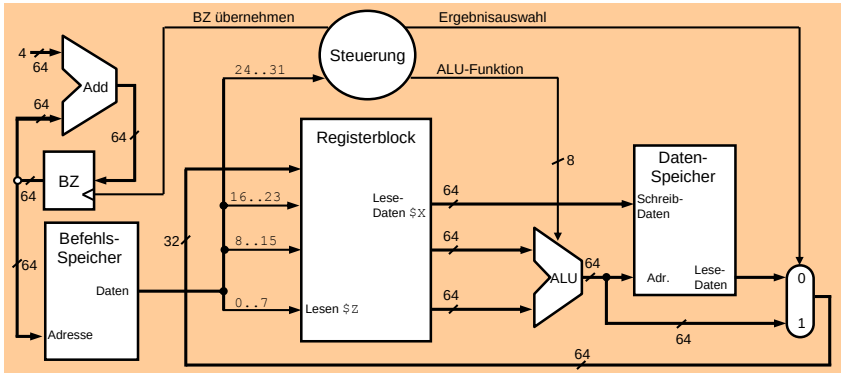
Maschinensprache

Mikroarchitektur

Digitale Logik

Transistoren und Verdrahtung

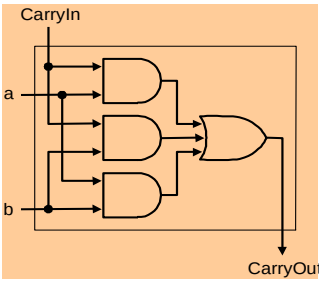
Masken-Layout/Halbleiter

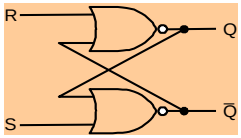


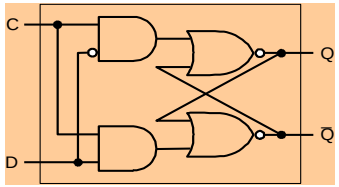
Abstraktionsebenen

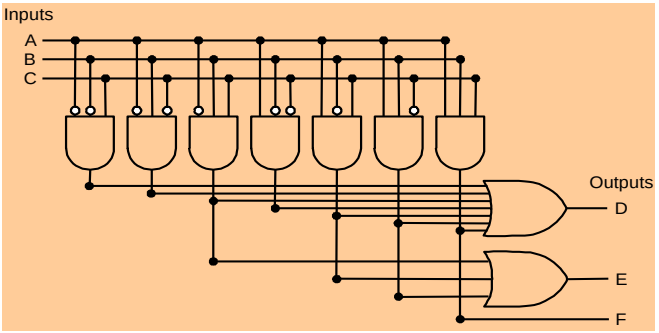
- *Algorithmus – Ebene*
- *Hochsprache (C/C++)*
- *Betriebssystem-Ebene*
- *Assembler*
- *Maschinensprache*

- *Mikroarchitektur*
- *Digitale Logik*
- *Transistoren und Verdrahtung*
- *Masken-Layout/Halbleiter*

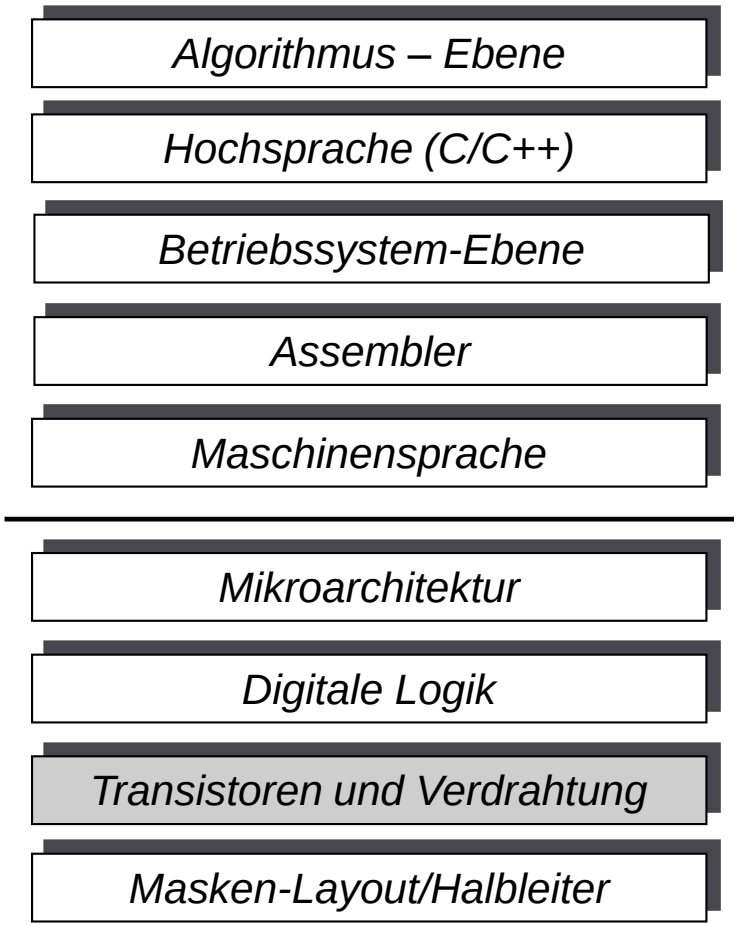


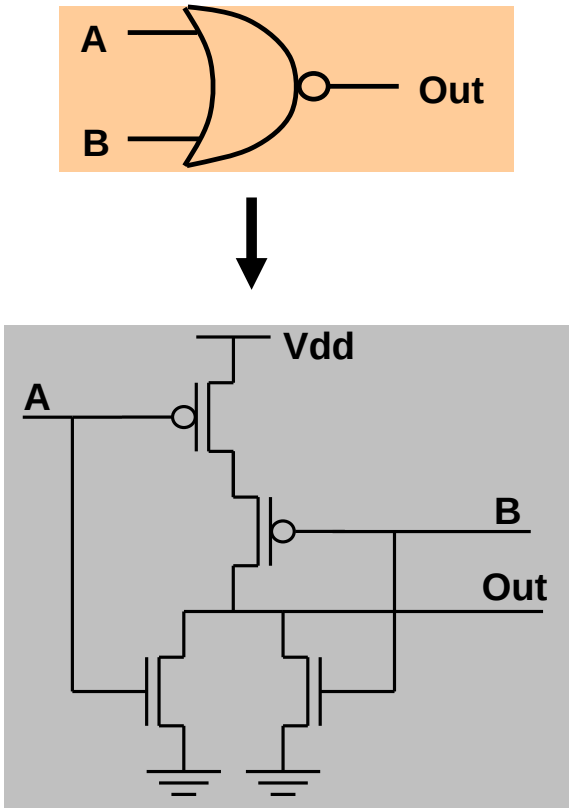




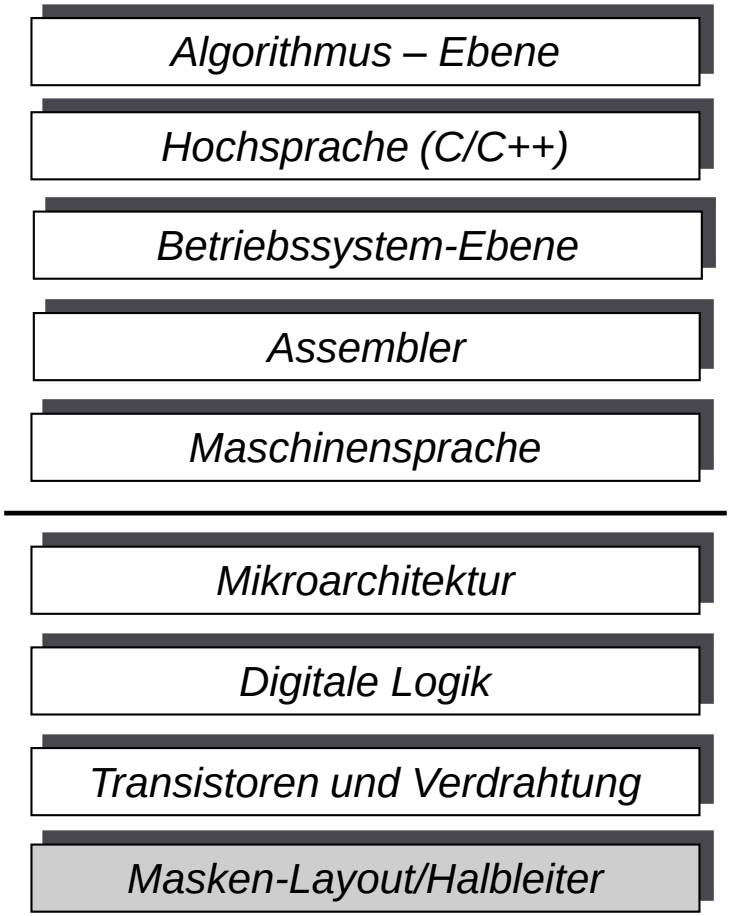


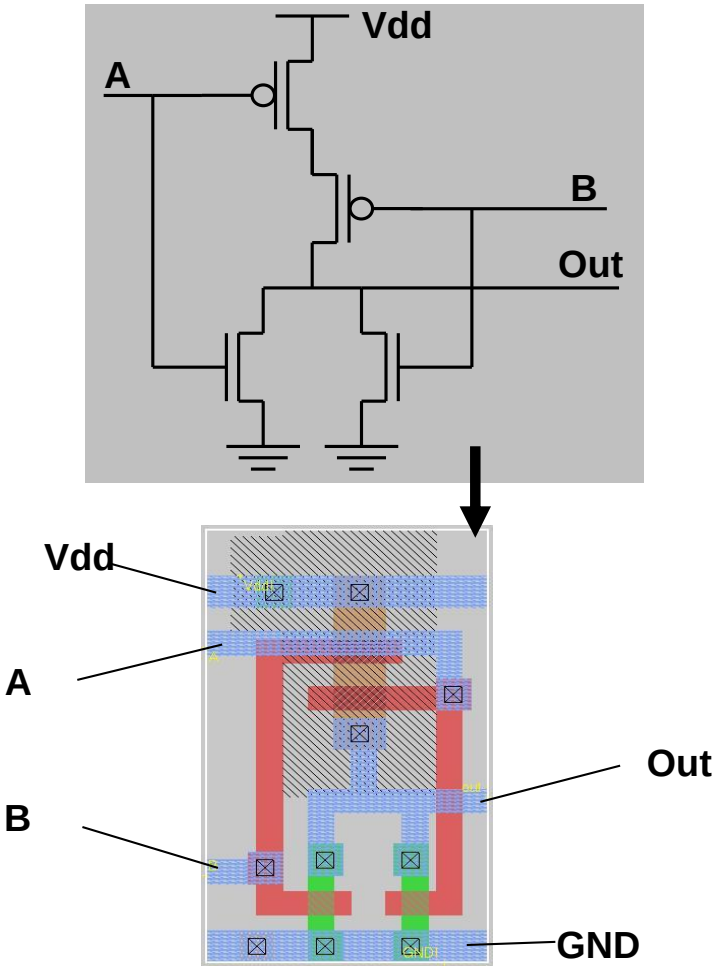
Abstraktionsebenen





Abstraktionsebenen

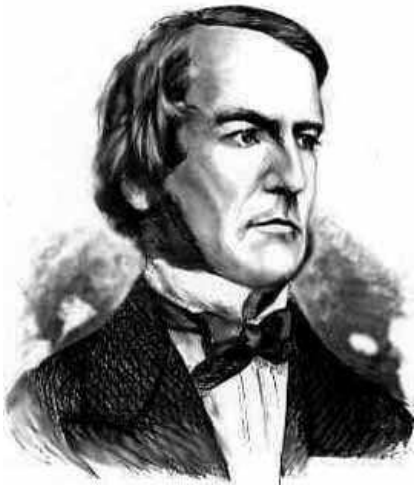




Kapitel II Boolesche Algebra und Logik

1. Einleitung

- besondere Rechenregeln für Systeme mit nur zwei möglichen Werten pro Symbol (vgl. Binärsystem)
- Grundlage für den Entwurf digitaler Schaltungen
- mathematisches Hilfsmittel für die algorithmische Behandlung von logischen Aussagen
- geht zurück auf den englischen Mathematiker George Boole



George Boole

lived from 1815 to 1864

Boole approached logic in a new way reducing it to a simple algebra, incorporating logic into mathematics. He also worked on differential equations, the calculus of finite differences and general methods in probability.

more on George Boole

2. Algebren

2.1 Grundbegriffe

Definition 1

Eine Algebra besteht aus einer Trägermenge S und einer Menge Φ von Operationen auf S (der Operatorenmenge). Dabei gilt: Jeder Operator ist eine (totale) Abbildung

$$S^m \to S$$

der Stelligkeit (Arität, arity) $m \in \mathbb{N}_0$.

- Nullstellige Operatoren sind Konstanten, z. B. 0, 47, $\perp$.
- Einstellige Operatoren sind unäre Operatoren, z. B. $x \mapsto 2^x$, $x \mapsto \neg x$, $A \mapsto 2^A$.
- Zweistellige Operatoren sind binäre Operatoren, z. B. $(x, y) \mapsto \max\{x, y\}$, $(x, y) \mapsto \mathrm{ggT}(x, y)$, $(x, y) \mapsto x + y$.
- Dreistellige Operatoren sind ternäre Operatoren, z. B. $(x, y, z) \mapsto$ **if** x **then** y **else** z **fi**

Beispiel 2

Sei U eine Menge, F die Menge der Funktionen von $U \to U$.
$(F, \circ)$ ist eine Algebra mit $\circ$ als Komposition von Funktionen.

Beispiel 3

Boolesche Algebra:
$\langle \{t, f\}, \{t, f, \neg, \wedge, \vee\} \rangle$ ist eine (endliche) Algebra.

2.2 Eigenschaften

Signatur einer Algebra

Definition 4

Die Signatur einer Algebra besteht aus der Liste der Stelligkeiten der Operatoren.

Beispiel 5

$\langle \mathbb{B}, \{t, f, \neg, \wedge, \vee\} \rangle$ (Boolesche Algebra, $\mathbb{B} = \{t, f\}$): 0, 0, 1, 2, 2

$$\begin{array}{lcll} \neg & : & \mathbb{B} & \to \mathbb{B} \\ \wedge & : & \mathbb{B} \times \mathbb{B} & \to \mathbb{B} \\ \vee & : & \mathbb{B} \times \mathbb{B} & \to \mathbb{B} \end{array}$$

Beispiel 6

$\langle 2^U, \{U, \emptyset, \bar{}, \cap, \cup\} \rangle$: 0, 0, 1, 2, 2

$$\begin{array}{lcll} \bar{} & : & 2^U & \to 2^U \\ \cap & : & 2^U \times 2^U & \to 2^U \\ \cup & : & 2^U \times 2^U & \to 2^U \end{array}$$

Diese beiden Algebren haben dieselbe Signatur; die Trägermenge ist unwesentlich, es kommt nur auf die Reihenfolge der Stelligkeiten an.

Einselement, Nullelement, Inverses

Sei $\langle S, \circ \rangle$ eine Algebra, $\circ$ beliebiger zweistelliger Operator.

Definition 7

- Ein Element $1 \in S$ heißt linkes (bzw. rechtes) Einselement für den Operator $\circ$, falls

$$(\forall a \in S) \quad 1 \circ a = a \text{ (bzw. } a \circ 1 = a)$$

  1 heißt Einselement, falls es linkes und rechtes Einselement ist.
- Ein Element $0 \in S$ heißt linkes (bzw. rechtes) Nullelement für den Operator $\circ$, falls

$$(\forall a \in S) \quad 0 \circ a = 0 \text{ (bzw. } a \circ 0 = 0)$$

  0 heißt Nullelement, falls es linkes und rechtes Nullelement ist.
- Sei 1 Einselement. Für $a \in S$ heißt $a^{-1} \in S$ Rechtsinverses von a, falls

$$a \circ a^{-1} = 1$$

  Analog: Linksinverses

Beispiel 8

Betrachte $F(U)$, d. h. die Menge aller Abbildungen $U \to U$. Dann gilt (mit der Komposition als Operator):

1. $f \in F(U)$ hat genau dann ein Rechtsinverses, wenn f surjektiv ist.

   $$f \circ f^{-1} = id$$

   (Wähle für f^{-1} irgendeine Funktion g, so dass gilt: $g(x)$ wird von f auf x abgebildet.)

2. $f \in F(U)$ hat genau dann ein Linksinverses, wenn f injektiv ist.

   $$f^{-1} \circ f = id$$

   (Wähle für f^{-1} irgendeine Funktion g, so dass gilt: $f(x)$ wird von g auf x abgebildet.)

Ist f bijektiv, dann stimmen die beiden f^{-1} aus (1) und (2) überein.

Bemerkungen:

Wir erinnern uns an folgende Definitionen:

- Eine Funktion $f : U \to V$ heißt injektiv, wenn gilt:

$$(\forall x, y \in U)[x \neq y \Rightarrow f(x) \neq f(y)]$$

- Eine Funktion $f : U \to V$ heißt surjektiv, wenn gilt:

$$(\forall y \in V \exists x \in U)[y = f(x)]$$

- Eine Funktion $f : U \to V$ heißt bijektiv, wenn gilt:

  f ist sowohl injektiv als auch surjektiv.

Wir hätten auch sagen können (mit $x \in f^{-1}(y) \Leftrightarrow f(x) = y$):
Sei f eine totale Funktion von U nach V (d.h., $f(x)$ ist für alle $x \in U$ definiert). Dann gilt:

- f injektiv: $(\forall y \in V)\left[\left|f^{-1}(y)\right| \leq 1\right]$
- f surjektiv: $(\forall y \in V)\left[\left|f^{-1}(y)\right| \geq 1\right]$
- f bijektiv: $(\forall y \in V)\left[\left|f^{-1}(y)\right| = 1\right]$, d.h. injektiv und surjektiv
- Ist $f : U \to V$ eine Bijektion, dann ist auch f^{-1} eine bijektive Funktion.

Wir erinnern ebenso an folgende Festlegungen:
Seien $f : U \to V$ und $g : V \to W$ jeweils Funktionen.

- Das Urbild von $y \in V$: $f^{-1}(y) = \{x \in U; f(x) = y\}$.
- Schreibweisen: $(U' \subseteq U, V' \subseteq V)$
  - $f(U') = \bigcup_{u \in U'} \{f(u)\}$
  - $f^{-1}(V') = \bigcup_{y \in V'} f^{-1}(y)$

Also (Erklärung zu Beispiel 8):
$f \in F(U)$ hat genau dann ein Rechtsinverses, wenn f surjektiv ist.

$$f \circ f^{-1} = id$$

(Wähle für f^{-1} irgendeine Funktion g, so dass gilt: $g(x)$ wird von f auf x abgebildet.)
$f \in F(U)$ hat genau dann ein Linksinverses, wenn f injektiv ist.

$$f^{-1} \circ f = id$$

(Wähle für f^{-1} irgendeine Funktion g, so dass gilt: $f(x)$ wird von g auf x abgebildet.)

Satz 9

Falls c linkes Einselement ist und d rechtes Einselement (bezüglich des binären Operator $\circ$), dann ist

$$c = d \; .$$

Beweis:

$$d = c \circ d = c \; .$$

□

Satz 10

Falls c linkes Nullelement und d rechtes Nullelement (bezüglich $\circ$) ist, dann ist

$$c = d\ .$$

Beweis:

$$c = c \circ d = d\ .$$

□

Beispiel 11

Betrachte $\langle \{b, c\}, \{\bullet\} \rangle$ mit

$\bullet$	b	c
b	b	b
c	c	c

Es gilt: b und c sind linke Nullelemente, und b und c sind rechte Einselemente.

Abgeschlossenheit

Definition 12

Sei $\langle S, \Phi \rangle$ eine Algebra, T eine Teilmenge von S.

- T ist unter den Operatoren in Φ abgeschlossen (stabil), falls ihre Anwendung auf Elemente aus T wieder Elemente aus T ergibt.
- $\langle T, \Phi \rangle$ heißt Unteralgebra von $\langle S, \Phi \rangle$, falls $T \neq \emptyset$ und T unter den Operatoren $\in \Phi$ abgeschlossen ist.

Beispiel 13

- $\langle \mathbb{N}_0, + \rangle$ ist Unteralgebra von $\langle \mathbb{Z}, + \rangle$
- $\langle \{0,1\}, \cdot \rangle$ ist Unteralgebra von $\langle \mathbb{N}_0, \cdot \rangle$
- $\langle \{0,1\}, + \rangle$ ist keine Unteralgebra von $\langle \mathbb{Z}, + \rangle$, da sie nicht abgeschlossen ist $(1 + 1 = 2)$.

2.3 Vereinbarung

Wir verwenden die Wertesymbole "1" und "0" oder die Begriffe "wahr" und "falsch" bzw. "true" und "false".

2.4 Schreibweisen und Sprachgebrauch

In der Literatur existieren verschiedene Schreibweisen für die Booleschen Operatoren:

Negation nicht a not a ¬a !a â $\overline{a}$

Konjunktion (Und-Verknüpfung, entspricht Multiplikation)
a und b a and b a·b a*b a∧b ab

Disjunktion (Oder-Verknüpfung, entspricht Addition)
a oder b a or b a+b a∨b

Die Operatoren werden in der Reihenfolge *nicht* - *und* - *oder* angewandt.

2.5 Beispiele für Axiome einer Algebra

Für alle $a, b, c \in S$ gilt:

Kommutativität	$a \cdot b = b \cdot a$ $a + b = b + a$
Assoziativität	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$ $(a + b) + c = a + (b + c)$
Distributivität	$a \cdot (b + c) = (a \cdot b) + (a \cdot c) = a \cdot b + a \cdot c$ $a + (b \cdot c) = (a + b) \cdot (a + c)$
Neutrales Element $n, e \in S$	$a \cdot e = e \cdot a = a$ $a + n = n + a = a$

2.6 Boolescher Verband

Unter einem Verband versteht man eine Algebra mit zwei binären Operatoren (z.B. $\cap$, $\cup$), die kommutativ und assoziativ sind.

Existieren in einem Verband die neutralen Elemente n und e (bzw. 0 und 1, oder $\emptyset$ und U) und zu jedem Element das komplementäre Element, so spricht man von einem komplementären Verband.

Besitzt ein komplementärer Verband auch die Eigenschaft der Distributivität, so ist dies ein Boolescher Verband bzw. eine Boolesche Algebra.

2.7 Boolesche Algebren

Eine Boolesche Algebra ist eine Algebra

$$\langle U, \oplus, \otimes, \sim, 0, 1 \rangle,$$

$\oplus, \otimes$ sind binäre, $\sim$ ist ein unärer Operator, 0 und 1 sind Konstanten. Es gilt:

1. $\oplus$ und $\otimes$ sind assoziativ und kommutativ.
2. 0 ist Einselement für $\oplus$, 1 ist Einselement für $\otimes$.
3. für $\sim$ gilt (Komplementarität):

$$\begin{aligned} b \oplus \sim b &= 1 \\ b \otimes \sim b &= 0 \quad \forall b \in U. \end{aligned}$$

4. Distributivgesetz:

$$b \otimes (c \oplus d) = (b \otimes c) \oplus (b \otimes d)$$
$$b \oplus (c \otimes d) = (b \oplus c) \otimes (b \oplus d)$$

Absorption	$a \cdot (a + b) = a$ $a + (a \cdot b) = a$
Involution	$\overline{\overline{a}} = a$
Idempotenz	$a \cdot a = a$ $a + a = a$
De Morgan	$\overline{a \cdot b} = \overline{a} + \overline{b}$ $\overline{a + b} = \overline{a} \cdot \overline{b}$
Dominanz	$a \cdot 0 = 0$ $a + 1 = 1$
0- und 1-Komplemente	$\overline{0} = 1$ $\overline{1} = 0$
Dualitätsprinzip	gleichzeitiges Vertauschen von $+$ und $\cdot$ und von 0 und 1 ergibt neues Gesetz der Booleschen Algebra $\rightarrow$ s. De Morgan

Auguste de Morgan
(1806–1871)

In 1838 De Morgan defined and introduced the term 'mathematical induction' putting a process that had been used without clarity on a rigorous basis.

De Morgan was always interested in numerical curiosities! Thus, he was

n years old in the year n^2 (for $n = 43$).

2.8 Boolesche Ausdrücke und Funktionen

Boolescher Ausdruck durch logische Verknüpfungen $+$ und $\cdot$ verbundene Variablen (auch in negierter Form)

Boolesche Umformung Anwendung der booleschen Gesetze auf boolesche Ausdrücke

Jede n-stellige boolesche Funktion bildet alle Kombinationen der Werte der n Eingangsgrößen auf einen Funktionswert aus $\{0,1\}$ ab.

$$f : \mathbb{B}^n \ni (x_1, \ldots, x_n) \mapsto f(x_1, x_2, \ldots, x_n) \in \mathbb{B}$$

Beobachtung: Da $|\mathbb{B}| = 2$, gibt es genau 2^n verschiedene Tupel in $\mathbb{B}^n$.
Da wir für jedes dieser Tupel den Funktionswert beliebig $\in \mathbb{B}$ wählen können, gibt es genau 2^{2^n} verschiedene (totale) Boolesche Funktionen mit n Argumenten.

2.8.1 Boolesche Funktionen mit einem Argument

Nach der Formel aus 8 gibt es $2^{2^1} = 4$ boolesche Funktionen mit einem Argument:

x	f_1	f_2	f_3	f_4
0	0	1	0	1
1	0	1	1	0

f_1: "falsch"-Funktion
f_2: "wahr"-Funktion
f_3: Identität
f_4: Negation

2.8.2 Boolesche Funktionen mit zwei Argumenten

Nach der Formel aus 8 gibt es $2^{2^2} = 16$ boolesche Funktionen mit zwei Argumenten:

x	y	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Bezeichnungen für einige dieser Funktionen:

f_2:	AND	$a \cdot b$
f_8:	OR	$a + b$
f_9:	NOR	$\overline{a + b}$
f_{15}:	NAND	$\overline{a \cdot b}$
f_7:	XOR	$(a + b) \cdot \overline{a \cdot b}$
f_{10}:	Äquivalenz	$(a \cdot b) + \overline{a + b}$
f_{14}:	Implikation	a impliziert b: $\overline{a} + b$
f_{12}:	Replikation	b impliziert a: $a + \overline{b}$

3. Normalformen boolescher Funktionen

- Jeder boolesche Ausdruck kann durch (äquivalente) Umformungen in gewisse Normalformen gebracht werden!

3.1 Vollkonjunktion und disjunktive Normalform (DNF)

Eine Vollkonjunktion ist ein boolescher Ausdruck,

- in dem **alle** Variablen **einmal** vorkommen (jeweils als negiertes oder nicht negiertes Literal),
- alle Literale durch Konjunktionen $\cdot$ ("und") verbunden sind.

Die disjunktive ("oder") Verbindung von Vollkonjunktionen nennt man disjunktive Normalform (DNF).

$$f(a,b,c) = \underbrace{\underbrace{(a \cdot b \cdot \overline{c})}_{\textit{Vollkonjunktion}} + \underbrace{(\overline{a} \cdot b \cdot \overline{c})}_{\textit{Vollkonjunktion}} + \underbrace{(\overline{a} \cdot \overline{b} \cdot c)}_{\textit{Vollkonjunktion}}}_{\textit{disjunktive Verknüpfung der Vollkonjunktionen}}$$

3.2 Ableitung der disjunktiven Normalform aus einer Wertetabelle

- jede Zeile der Wertetabelle entspricht einer Vollkonjunktion
- Terme mit Funktionswert "0" tragen nicht zum Funktionsergebnis bei ("oder" von 0)

a	b	f(a,b)
0	0	0
0	1	1
1	0	1
1	1	0

  - bilde Vollkonjunktionen für Zeilen mit Funktionswert "1" $\rightarrow$ Zeilen 2 und 3 ("0" in Tabelle $=$ Negation der Variable)
  - Zeile 2: $\overline{a} \cdot b$
  - Zeile 3: $a \cdot \overline{b}$
  - disjunktive Verknüpfung der Vollkonjunktionen: $f(a,b) = \overline{a} \cdot b + a \cdot \overline{b}$

3.3 Volldisjunktion und konjunktive Normalform (KNF/CNF)

Eine Volldisjunktion ist ein boolescher Ausdruck,

- in dem **alle** Variablen **einmal** vorkommen (in Form eines negierten oder nicht negierten Literals),
- alle Variablen durch eine Disjunktion $+$ ("oder") verbunden sind.

Die konjunktive ("und") Verbindung von Volldisjunktionen nennt man konjunktive Normalform, kurz KNF (engl.: CNF).

$$f(a,b,c) = \underbrace{\underbrace{(a+b+\overline{c})}_{\textit{Volldisjunktion}} \cdot \underbrace{(\overline{a}+b+\overline{c})}_{\textit{Volldisjunktion}} \cdot \underbrace{(\overline{a}+\overline{b}+c)}_{\textit{Volldisjunktion}}}_{\textit{konjunktive Verknüpfung der Volldisjunktionen}}$$

3.4 Ableitung der konjunktiven Normalenform aus einer Wertetabelle

- jede Zeile der Wertetabelle entspricht einer Volldisjunktion
- Terme mit Funktionswert "1" tragen nicht zum Funktionsergebnis bei ("und" mit 1)

a	b	$f(a,b,c)$
0	0	0
0	1	1
1	0	1
1	1	0

- bilde Volldisjunktionen für Zeilen mit Funktionswert "0" $\rightarrow$ Zeilen 1 und 4 ("1" in Tabelle $=$ Negation der Variable)
- Zeile 1: $a + b$
- Zeile 4: $\overline{a} + \overline{b}$
- konjunktive Verknüpfung der Volldisjunktionen:
- $f(a,b) = (a + b) \cdot (\overline{a} + \overline{b})$

3.5 Vergleich von DNF und KNF

	DNF	**KNF**
wähle Zeilen mit Funktionswert	1	0
Bildung der Teil-Terme	Negation der "0" Einträge Verknüpfung der Variablen mit "und"	Negation der "1" Einträge Verknüpfung der Variablen mit "oder"
Verknüpfung der Teil-Terme	mit "oder"	mit "und"

Bemerkungen:

1. Beim Entwurf digitaler Schaltungen ist man oft daran interessiert, eine zu implementierende Funktion in einer Normalform (oder dazu ähnlichen Form) möglichst geringer Größe darzustellen.
2. Dazu dienen z.B. die Verfahren von Karnaugh-Veitch und Quine und McCluskey, die aber nicht Gegenstand dieser Vorlesung sind.

3.6 Bemerkungen zur Umformung boolescher Formeln (NAND):

Häufig verwendeten Umformungen sind:

Idempotenz	$a = a \cdot a$
doppelte Negation	$a = \overline{\overline{a}}$
De Morgan	$\overline{a} + \overline{b} = \overline{a \cdot b}$

- Gemeinsame Anwendung von Idempotenz und doppelter Negation zeigt, dass eine UND-Verknüpfung durch drei NAND-Verknüpfungen dargestellt werden kann:

$$a \cdot b = \overline{\overline{(a \cdot b)}} = \overline{\overline{(a \cdot b)} \cdot \overline{(a \cdot b)}}$$

- Gemeinsame Anwendung aller drei Umformungen erlaubt auch Umwandlung einer ODER-Verknüpfung in drei NAND-Verknüpfungen:

$$a + b = \underbrace{\overline{\overline{a + b}}}_{\text{doppelte Negation}} = \underbrace{\overline{\overline{a} \cdot \overline{b}}}_{\text{De Morgan}} = \underbrace{\overline{\overline{(a \cdot a)} \cdot \overline{(b \cdot b)}}}_{\text{Idempotenz}}$$

- Dies gilt auch für mehr als 2 Variablen bzw. allgemein für Literale:

$$\overline{a} + b + c = \underbrace{\overline{\overline{\overline{a} + b + c}}}_{\text{doppelte Negation}} = \underbrace{\overline{\overline{\overline{a} \cdot \overline{b} \cdot \overline{c}}}}_{\text{DeMorgan}} = \underbrace{\overline{a \cdot \overline{(b \cdot b)} \cdot \overline{(c \cdot c)}}}_{\text{Involution } und Idempotenz}$$

3.7 Umformung zur "NOR-Darstellung"

Analog existieren folgende Umformungen:

Idempotenz	$a = a + a$
doppelte Negation	$a = \overline{\overline{a}}$
De Morgan	$\overline{a} \cdot \overline{b} = \overline{a + b}$

Die Umwandlungen von OR nach NOR sowie AND nach NOR ergeben sich aus den oben beschriebenen Regeln durch Vertauschung der Operatoren "AND" und "OR".

Bemerkungen:

1. Aus dem gerade Gezeigten ergibt sich, dass NOT, AND und OR jeweils durch NAND dargestellt werden können. Das Gleiche gilt für NOR.
2. Eine (minimale) Menge boolescher Funktionen, mittels derer sich alle anderen booleschen Funktionen darstellen lassen, heißt eine Basis.
3. $\{\mathrm{NOT}, \mathrm{AND}\}$, $\{\mathrm{NOT}, \mathrm{OR}\}$, $\{\mathrm{NAND}\}$ und $\{\mathrm{NOR}\}$ sind dementsprechend solche Basen.
4. $\{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}\}$ genügt, um alle anderen booleschen Funktionen darzustellen, ist aber nicht minimal.
5. In der Praxis ist es oft wichtig, nur Gatter möglichst wenig verschiedener Typen zu verwenden.
6. $\{\mathrm{AND}, \mathrm{OR}\}$ genügt nicht, um alle anderen booleschen Funktionen darzustellen, da alle damit erzeugten Funktionen monoton sind.

Bemerkung:
Die Umformung boolescher Ausdrücke z.B. zwischen verschiedenen Normalformeln kann ihre Länge exponentiell wachsen lassen.

Beispiel 14

$$\prod_{i=1}^{n}(a_i + b_i) = \sum_{I \subseteq 2^{\{1,\ldots,n\}}} \prod_{i \in I} a_i \prod_{i \notin I} b_i$$

Auch ist das Auffinden einer minimalen solchen Darstellung (vgl. Quine, McCluskey) ein algorithmisch notorisch schwieriges Problem (es gehört zur Klasse der so genannten NP-harten Probleme).

3.8 Addierer als Anwendungsbeispiel

3.8.1 Halbaddierer

- Addition von zwei einstelligen Binärzahlen
- zwei Ausgänge, je einer für Summe und Übertrag (Carry)

	x	$\overline{x}$
y		1
$\overline{y}$	1	

Summenfunktion
Ausgang = 1 falls
genau ein Eingang 1
(XOR-Funktion)

	x	$\overline{x}$
y	1	
$\overline{y}$		

Übertragsfunktion
Ausgang = 1 falls
beide Eingänge 1
(AND-Funktion)

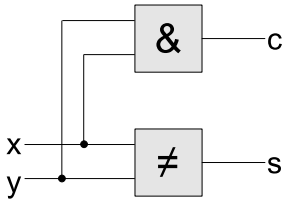


Abbildung: Halbaddierer – Funktionstabellen und Schaltbild

3.8.2 Volladdierer

- Addition von zwei einstelligen Binärzahlen unter Berücksichtigung des Carry-Flags
- zwei Ausgänge, je einer für Summe und Übertrag (Carry)
- besteht aus zwei verschalteten Halbaddierern und einem Oder-Gatter

	x	$\overline{x}$	
y	1		c
$\overline{y}$		1	
	1		$\overline{c}$
y		1	

Summenfunktion mit Carry-Bit c

	x	$\overline{x}$	
y	1	1	c
$\overline{y}$	1		
			$\overline{c}$
y	1		

Übertragsfunktion mit Carry-Bit c

Dem ersten Halbaddierer wird ein weiterer HA nachgeschaltet um Carry einer vorangehenden Stufe zu verarbeiten.

Das ODER-Glied fasst die beiden Carry-Flags der HA zusammen.

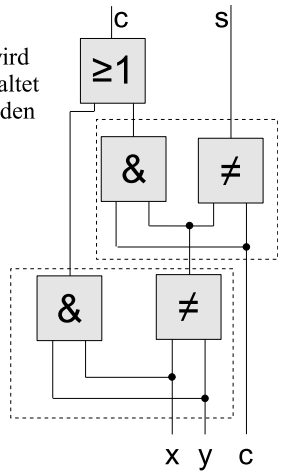


Abbildung: Volladdierer – Funktionstabellen und Schaltbild

4. Morphismen

Seien $A = \langle S, \Phi \rangle$ und $\tilde{A} = \langle \tilde{S}, \tilde{\Phi} \rangle$ zwei Algebren mit derselben Signatur.

4.1 Isomorphismus

Definition 15
Eine Abbildung

$$h : S \to \tilde{S}$$

heißt ein Isomorphismus von A nach $\tilde{A}$, falls

- h bijektiv ist und
- h mit den in Φ und $\tilde{\Phi}$ einander entsprechenden Operatoren vertauschbar ist (kommutatives Diagramm):

$$\begin{array}{ccc} S^m & \xrightarrow{\circ} & S \\ {\scriptstyle (h,\ldots,h)}\downarrow & & \downarrow{\scriptstyle h} \\ \tilde{S}^m & \xrightarrow{\tilde{\circ}} & \tilde{S} \end{array}$$

h ist also ein Isomorphismus gdw

- $h(c) = \tilde{c}$ für alle nullstelligen Operatoren (Konstanten) c
- $h\big(u(x)\big) = \tilde{u}\big(h(x)\big)$ für alle unären Operatoren $u \in \Phi$, $\forall x \in S$
- $h\big(b(x,y)\big) = \tilde{b}\big(h(x), h(y)\big)$ für alle binären Operatoren $b \in \Phi$, $\forall x, y \in S$

Notation: $A \cong \tilde{A}$: „A isomorph zu $\tilde{A}$“, d. h. es existiert ein Isomorphismus von A nach $\tilde{A}$ (und von $\tilde{A}$ nach A).

Ein Isomorphismus von A nach A heißt Automorphismus.

Zur Vereinfachung der Notation schreiben wir statt $\langle S, \{o_1, \ldots, o_k\}\rangle$ auch

$$\langle S, o_1, \ldots, o_k\rangle \ ,$$

solange keine Verwechslung zu befürchten ist.

Beispiel 16

$\langle \mathbb{N}_0, + \rangle$ und $\langle 2 \cdot \mathbb{N}_0, + \rangle$ ($2 \cdot \mathbb{N}_0$: gerade Zahlen) mit

$$h : \mathbb{N}_0 \ni n \mapsto 2 \cdot n \in 2\mathbb{N}_0$$

ist ein Isomorphismus zwischen den beiden Algebren.

Beispiel 17

$\langle \mathbb{R}^+, \cdot \rangle$ und $\langle \mathbb{R}, + \rangle$ $\left(\mathbb{R}^+ = \{x \in \mathbb{R}; x > 0\}\right)$

$$h : \mathbb{R}^+ \ni x \mapsto \log x \in \mathbb{R}$$

ist ein Isomorphismus (der sog. Rechenschieberisomorphismus)

Satz 18

Ein Algebra-Isomorphismus bildet Einselemente auf Einselemente, Nullelemente auf Nullelemente und Inverse auf Inverse ab.

Beweis:

Sei die Abbildung $h : S \to \tilde{S}$ ein Isomorphismus von $A = \langle S, \Phi \rangle$ nach $\tilde{A} = \langle \tilde{S}, \tilde{\Phi} \rangle$.
Sei 1 ein rechtes Einselement für den Operator $\circ \in \Phi$ in A. Dann gilt für alle $\tilde{b} \in \tilde{S}$:

$$\tilde{b} \circ h(1) = h(b) \circ h(1) = h(b \circ 1) = h(b) = \tilde{b}$$

Also ist $h(1)$ ein rechtes Einselement in $\tilde{A}$. Die Argumentation für linke Einselemente, Nullelemente und Inverse ist analog. □

4.2 Wie viele Boolesche Algebren gibt es?

Definition 19

Sei $A = \langle S, \oplus, \otimes, \sim, 0, 1 \rangle$ eine endliche Boolesche Algebra. Dann definiert man:

$$a \leq b \iff a \otimes b = a$$
$$a < b \iff a \leq b \ \wedge \ a \neq b$$

Satz 20

Durch $\leq$ ist auf A eine partielle Ordnung definiert, d. h. eine reflexive, antisymmetrische und transitive Relation.

Beweis:

(a) Reflexivität: Zu zeigen ist, dass für alle $a \in S$ gilt $a \leq a$, d. h. $a \otimes a = a$ (Idempotenzgesetz bzgl. $\otimes$)

(b) Antisymmetrie: Sei $a \leq b \ \wedge \ b \leq a$. Damit gilt: $a \otimes b = a$ und $b \otimes a = b$ nach Definition. Damit:

$$a = a \otimes b = b \otimes a = b$$

(c) Transitivität: Sei $a \leq b \ \wedge \ b \leq c$, dann gilt: $a \otimes b = a$ und $b \otimes c = b$. Es ist zu zeigen, dass $a \leq c$, d.h. $a \otimes c = a$.

$$a \otimes c = (a \otimes b) \otimes c = a \otimes (b \otimes c) = a \otimes b = a$$

□

Definition 21

Ein Element $a \in S$, $a \neq 0$ heißt ein Atom, i. Z. $\mathrm{atom}(a)$, falls

$$(\forall b \in S \setminus \{0\}) \Big[b \leq a \;\Rightarrow\; b = a \Big].$$

Satz 22

Es gilt:

1. $\mathrm{atom}(a) \;\Rightarrow\; (\forall b \in S) \quad [a \otimes b = a \;\vee\; a \otimes b = 0]$
2. $\mathrm{atom}(a) \wedge \mathrm{atom}(b) \wedge a \neq b \;\Rightarrow\; a \otimes b = 0$
3. *Falls gilt:* $(\forall a \in S)[\mathrm{atom}(a) \;\Rightarrow\; a \otimes b = 0]$, *dann* $b = 0$.

Beweis:

[Wir zeigen nur die erste Teilbehauptung]

1. Sei a ein Atom. Nach Voraussetzung gilt (mit $a \otimes b$ statt b):

$$a \otimes b \neq 0 \quad \Longrightarrow \quad \big(a \otimes b \leq a \;\Rightarrow\; a \otimes b = a\big)$$

Da aber $a \otimes b \leq a$ ist (Idempotenz), folgt insgesamt

$$(a \otimes b = 0) \vee (a \otimes b = a).$$

□

Satz 23 (Darstellungssatz)

Jedes Element x einer endlichen Booleschen Algebra $\langle S, \oplus, \otimes, \sim, 0, 1\rangle$ lässt sich in eindeutiger Weise als $\oplus$-Summe von Atomen schreiben:

$$x = \bigoplus_{\substack{a \in S \\ \mathrm{atom}(a) \\ a \otimes x \neq 0}} a$$

ohne Beweis!

Korollar 24

Jede endliche Boolesche Algebra mit n Atomen enthält genau 2^n Elemente.

Korollar 25

Jede endliche Boolesche Algebra $A = \langle S, \oplus, \otimes, \sim, 0, 1\rangle$ mit n Atomen ist isomorph zur Potenzmengenalgebra

$$\mathcal{P}_n := \langle 2^{\{1,\ldots,n\}}, \cap, \cup, {}^-, \emptyset, \{1,\ldots,n\}\rangle$$

Beweis:

Seien $a_1, \ldots, a_n$ die Atome von A. Definiere die Abbildung

$$h : S \ni \bigoplus_{i\in I} a_i \mapsto I \in 2^{\{1,\ldots,n\}}$$

Diese Abbildung ist ein Isomorphismus (leicht nachzurechnen). □

5. Ergänzungen zu booleschen Funktionen

Neben UND ($\cdot, \wedge$), ODER ($+, \vee$) und NOT ($\neg, ^-$) sind die

Implikation ($\Rightarrow$) mit

p	q	$p \Rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

und die

Äquivalenz ($\equiv$) mit

p	q	$p \equiv q$
0	0	1
0	1	0
1	0	0
1	1	1

mit die wichtigsten binären booleschen Funktionen.

Implikation und Äquivalenz werden auch sehr häufig in Beweisen, Argumentationen und Ableitungen benützt. Während die logische Bedeutung sich dabei nicht ändert, gibt es in der Notation jedoch ein paar Besonderheiten, die zu beachten sind.
Wir verwenden im Folgenden die folgende Präzedenz der angegebenen binären Operatoren:

$$\neg \text{ vor } \wedge \text{ vor } \vee \text{ vor } \Rightarrow \text{ vor } \equiv$$

Wenn wir dabei $p = q$ schreiben, bedeutet das, dass mit den gegebenen Axiomen und Ableitungsregeln (sowie Substitution) q aus p sowie p aus q abgeleitet werden kann.

Beispiel 26

$$p \wedge q \vee p \wedge \neg q \equiv p \;=\; (((p \wedge q) \vee (p \wedge (\neg q))) \equiv p\,.$$

In Beweisen verwenden wir häufig Ketten der Form

$$\begin{aligned} A_0 &\Rightarrow A_1 \\ &\Rightarrow A_2 \\ &\Rightarrow A_3 \\ &\equiv A_4 \\ &\vdots \\ &\Rightarrow A_n \end{aligned}$$

Diese Kette steht für

$$A_0 \Rightarrow A_1 \text{ und } A_1 \Rightarrow A_2 \text{ und } \ldots \text{ und } A_{n-1} \Rightarrow A_n \ ,$$

während für den booleschen Ausdruck

$$A_1 \Rightarrow A_2 \Rightarrow A_3 \equiv A_4 \equiv A_5 \Rightarrow A_6$$

keine eindeutige Klammerung definiert ist!

Es ist jedoch leicht aus der entsprechenden Wertetabelle zu sehen, dass $\equiv$ assoziativ ist, d.h.

$$(p \equiv q) \equiv r \;=\; p \equiv (q \equiv r)\;,$$

während Assoziativität für $\Rightarrow$ nicht gilt! $\Rightarrow$ ist auch nicht

- kommutativ (bzw. symmetrisch) (symmetrisch: aus $p \Rightarrow q$ folgt stets $q \Rightarrow p$)
- antisymmetrisch (antisymmetrisch: wenn $p \Rightarrow q$ und $q \Rightarrow p$, dann $p = q$)
- asymmetrisch (asymmetrisch: aus $p \Rightarrow q$ folgt stets $q \not\Rightarrow p$)

Sprachliche Umsetzung der Implikation

Die boolesche/materiale Implikation $p \Rightarrow q$ wird sprachlich ausgedrückt durch

- (schon) wenn p, dann q;
- p impliziert q;
- aus p folgt q;
- p ist hinreichend für q;
- p nur wenn q;
- q ist notwendig für p;
- $\neg q$ impliziert $\neg p$;
- ...

Einige formale Eigenschaften der Implikation

1. $$p \Rightarrow q \;=\; \neg p \vee q \;.$$

   Beweis durch Wertetabelle!

2. $$p \Rightarrow q \;=\; \neg q \Rightarrow \neg p \;,$$

   da (mit (1)) $\neg q \Rightarrow \neg p = q \vee \neg p = \neg p \vee q = p \Rightarrow q.$

3. $$p \Rightarrow q \;=\; \neg(p \wedge \neg q) \;,$$

   da (mit De Morgan)
   $\neg(p \wedge \neg q) = \neg p \vee \neg\neg q = \neg p \vee q = p \Rightarrow q.$

4

$$p \Rightarrow q \;=\; p \wedge q \equiv p \;,$$

da

$$\begin{aligned} p \wedge q \equiv p &= (p \wedge q \Rightarrow p) \wedge (p \Rightarrow p \wedge q) \\ &= (\neg(p \wedge q) \vee p) \wedge (\neg p \vee p \wedge q) \\ &= (\neg p \vee \neg q \vee p) \wedge (\neg p \vee p \wedge q) \\ &= \neg p \vee p \wedge q = (\neg p \vee p) \wedge (\neg p \vee q) \\ &= \neg p \vee q \\ &= p \Rightarrow q \;. \end{aligned}$$

5. $$p \Rightarrow q \;=\; \neg p \wedge \neg q \equiv \neg q \;;$$

   die Argumentation verläuft hier analog zum vorherigen Fall, da $\neg p \wedge \neg q \equiv \neg q \equiv \neg q \Rightarrow \neg p$.

6. $$p \Rightarrow (q \Rightarrow r) \equiv (p \Rightarrow q) \Rightarrow (p \Rightarrow r) \;.$$

   Übungsaufgabe!

7. $$p \Rightarrow (q \Rightarrow r) \equiv p \wedge q \Rightarrow r \;.$$

   Übungsaufgabe!

Ebenso lässt sich leicht zeigen:

8. $$p \Rightarrow 1 \;=\; 1\,,$$

   d.h. 1 ist Rechtsnullelement von $\Rightarrow$, und

9. $$1 \Rightarrow p \;=\; p\,,$$

   d.h. 1 ist Linkseinselement von $\Rightarrow$, sowie

10. $$p \Rightarrow 0 \;=\; \neg p$$

    (das Prinzip des Widerspruchsbeweises), und

11. $$0 \Rightarrow p \;=\; 1\,,$$

    und zwar für alle p! (Aus etwas Falschem kann man alles folgern!)

Eine wichtige Ableitungsregel ist schließlich der

modus ponens:

$$p \wedge (p \Rightarrow q) \Rightarrow q \ .$$

Kapitel III Automatentheorie

1. Begriff des Automaten

- der Automat ist ein abstraktes Modell eines sehr einfachen Computers
- der Automat verfügt über Eingabe- und Ausgabemöglichkeiten
- Eingabe und Ausgabe hängen durch den inneren Zustand des Automaten zusammen

2. Eigenschaften von Automaten

2.1 Ein- und Ausgabe

- ein Automat kann eingehende Informationen in Form von Symbolen an einem (oder mehreren) Eingängen verarbeiten
- die an den Eingängen erlaubten Symbole sind durch das Eingabealphabet bestimmt
- die interne Berechnung kann den internen Zustand des Automaten verändern
- das Ergebnis der internen Berechnung kann als Symbol ausgegeben werden
- alle möglichen Ausgabesymbole bilden zusammen das Ausgabealphabet

2.2 Innere Zustände

- die Abhängigkeit der Ausgabe alleine von der Eingabe ermöglicht keine Aufgaben, die ein "Gedächtnis" benötigen
- verschiedene innere Zustände eines Automaten erlauben eine Abhängigkeit der Ausgabefunktionen von vorhergehenden Ereignissen
- die Menge der internen Zustände kann als "Kurzzeitgedächtnis" angesehen werden
- der Zustand transportiert Informationen zwischen zwei zeitlich aufeinanderfolgenden Operationen
- einer der inneren Zustände muss als Startzustand definiert sein
- für besondere Automaten werden spezielle Zustände als Endzustände bezeichnet

2.3 Ausgabefunktion

- definiert das nach jedem Schritt auszugebende Zeichen
-
  - Moore-Automat: die Ausgabefunktion hängt nur vom inneren Zustand des Automaten ab
  - Mealy-Automat: die Ausgabefunktion hängt vom inneren Zustand und der aktuell gelesenen Eingabe (an allen Eingängen) ab

2.4 Zustandsübergangsfunktion

- abhängig von allen vorhandenen Eingängen sowie dem aktuellen inneren Zustand
- definiert den Zustand, in den nach Abschluss des Rechenschrittes gewechselt werden soll

2.5 Determinismus

- Determinismus garantiert die Eindeutigkeit des Zustandsübergangs, d.h. für jede Kombination von Eingangswerten gibt es maximal einen Zustandsübergang
- nicht-deterministische Automaten erlauben mehr als einen Zustandsübergang für ein und dieselbe Kombination von Eingangswerten, die Zustandsübergangsfunktion und die Ausgabefunktion werden dann zu Relationen
- jeder nicht-deterministische endliche Automat kann in einen deterministischen Automaten überführt werden (zeigen wir später)

2.6 Verdeutlichung verwendeter Begriffe

- endlich/finit: die Mengen der Zustände und der Ein- bzw. Ausgabezeichen sind endlich
- synchron: die Ausgabezeichen erscheinen synchron mit dem Einlauf der Eingabezeichen
- sequentiell: die Eingabezeichen werden seriell verarbeitet und bilden eine Eingangsfolge

3. Darstellung

3.1 Graphische Darstellung

- ein Automat kann durch den so genannten Zustandsübergangs-Graphen dargestellt werden
- die Knoten des Graphen entsprechen den Zuständen
- jede Kante des Graphen entspricht einem Arbeitsschritt
- am Beginn der Kante wird/werden das/die Eingangssymbol/e eingetragen, für welche dieser Übergang gültig ist
- bei Mealy-Automaten wird das Ausgangssymbol an der Kante angetragen, welches bei Benutzung dieses Überganges ausgegeben wird, bei Moore-Automaten wird das Symbol in den Zielknoten geschrieben
- der Startzustand wird durch einen einlaufenden Pfeil (Pfeil, der im leeren Raum beginnt) gekennzeichnet
- Endzustände werden durch einen Doppelkreis dargestellt

3.2 Tabellarische Darstellung

- die Menge der möglichen Zustände und Eingangssymbole ist endlich, alle möglichen Kombinationen von Eingangssymbolen und Zuständen lassen sich aufschreiben
- jeder solchen Kombination wird durch die Ausgabefunktion ein Ausgabesymbol und durch die Zustandsübergangsfunktion ein Folgezustand zugeordnet
- diese Abhängigkeiten werden tabellarisch dargestellt
- diese Darstellung eignet sich für maschinelle Bearbeitung natürlich besser als eine graphische Schreibweise
- ist die Zustandsübergangsfunktion nur partiell gegeben, so kann sie durch Einführung eines Fangzustandes total gemacht werden

3.3 Mathematische Darstellung

- Eingabe- und Ausgabezeichen, sowie Zustände und Endzustände werden als endliche Mengen von Symbolen geschrieben
- Startzustand ist ein Symbol aus der Menge der Zustände
- die Ausgabefunktion/-relation und die Zustandsübergangsfunktion/-relation sind mathematische Funktionen bzw. Relationen

4. Deterministische endliche Automaten

Definition 27

Ein deterministischer endlicher Automat (englisch: deterministic finite automaton, kurz DFA) wird durch ein 7-Tupel $M = (Q, \Sigma, \delta, \Gamma, g, q_0, F)$ beschrieben, das folgende Bedingungen erfüllt:

1. Q ist eine endliche Menge von Zuständen.
2. Σ ist eine endliche Menge, das Eingabealphabet, wobei $Q \cap \Sigma = \emptyset$.
3. $q_0 \in Q$ ist der Startzustand.
4. $F \subseteq Q$ ist die Menge der Endzustände.
5. $\delta : Q \times \Sigma \to Q$ heißt Übergangsfunktion.
6. $g : Q \to \Gamma$ (bzw. $g : Q \times \Sigma \to \Gamma$) ist die Ausgabefunktion, Γ das Ausgabealphabet.

Die von M akzeptierte Sprache ist

$$L(M) := \{w \in \Sigma^*;\ \hat{\delta}(q_0, w) \in F\}\ ,$$

wobei $\hat{\delta} : Q \times \Sigma^* \to Q$ induktiv definiert ist durch

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q && \text{für alle } q \in Q \\ \hat{\delta}(q, ax) &= \hat{\delta}(\delta(q, a), x) && \text{für alle } q \in Q, a \in \Sigma \\ & && \text{und } x \in \Sigma^* \end{aligned}$$

Bemerkung: Endliche Automaten können durch (gerichtete und markierte) Zustandsgraphen (aka Cayley-Diagramme) veranschaulicht werden:

- Knoten $\widehat{=}$ Zuständen
- Kanten $\widehat{=}$ Übergängen
- genauer: die mit $a \in \Sigma$ markierte Kante (u, v) entspricht $\delta(u, a) = v$

Zusätzlich werden der Anfangszustand durch einen Pfeil, Endzustände durch doppelte Kreise gekennzeichnet.

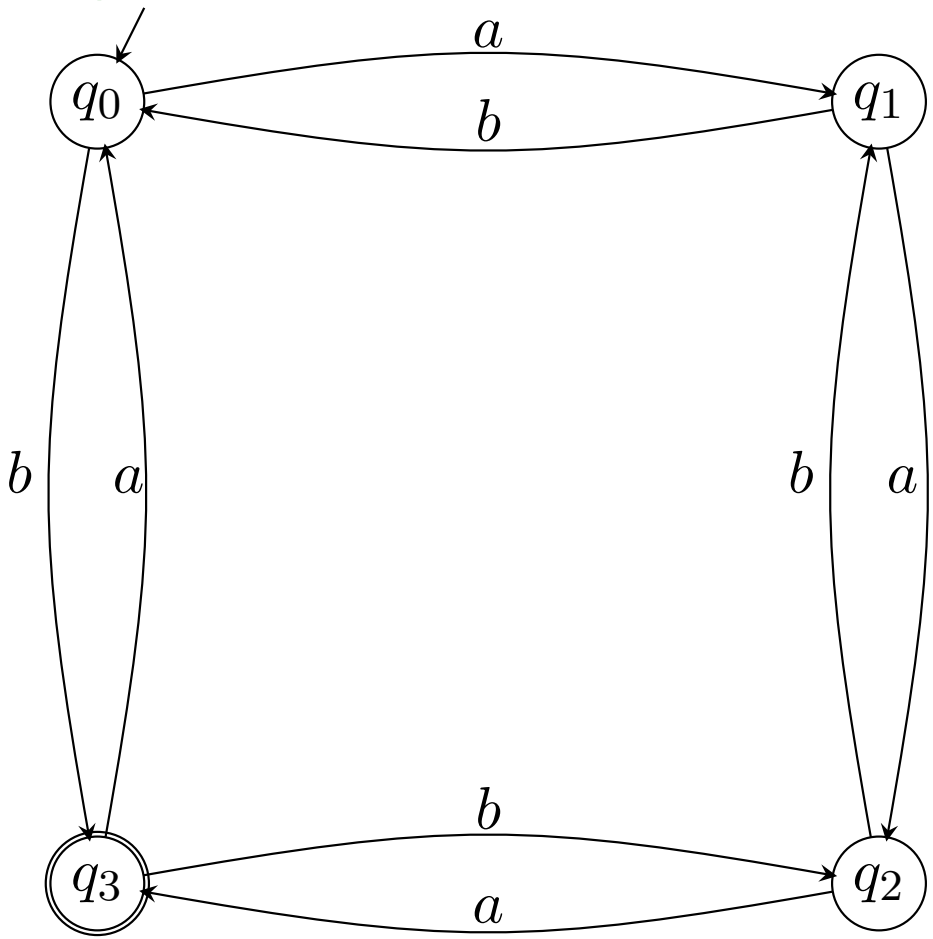
Weitere Informationen zu Arthur Cayley und zu Cayley-Diagrammen, deren ursprüngliches Anwendungsgebiet ja die algebraische Gruppentheorie ist, finden sich unter

1. Arthur Cayley (weitere Bilder von Arthur Cayley)
2. Grundbegriff des Cayley-Diagramms

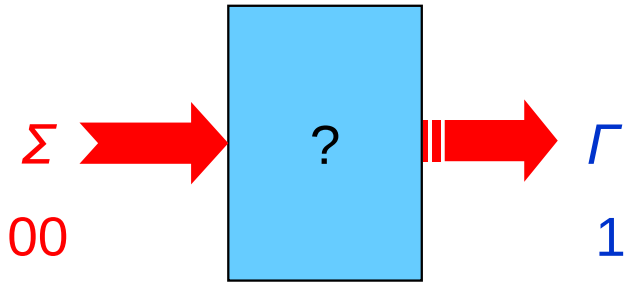
Beispiel 28 (DEA ohne Ausgabe)

Sei $M = (Q, \Sigma, \delta, q_0, F)$, wobei

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3\} \\
\Sigma &= \{a, b\} \\
F &= \{q_3\} \\
\delta(q_0, a) &= q_1 \\
\delta(q_0, b) &= q_3 \\
\delta(q_1, a) &= q_2 \\
\delta(q_1, b) &= q_0 \\
\delta(q_2, a) &= q_3 \\
\delta(q_2, b) &= q_1 \\
\delta(q_3, a) &= q_0 \\
\delta(q_3, b) &= q_2
\end{aligned}
$$



Beispiel 29



- **eingegebene Zeichenfolge ($e_i \in \Sigma$ Eingabe)**

	e_6	e_5	e_4	e_3	e_2	e_1
=	01	10	10	00	11	00

- **Ausgabezeichenfolge ($a_i \in \Gamma$ Ausgabe)**

	a_6	a_5	a_4	a_3	a_2	a_1
=	1	1	1	1	0	1

- **Eingabe und Ausgabe erfolgen synchron**

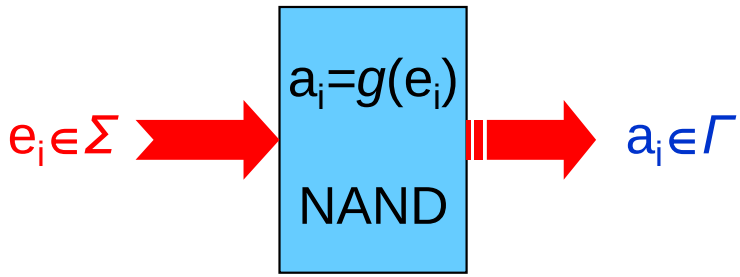
NAND-Automat

- **Endlicher Automat:** $\mathcal{A} = (Q, \Sigma, \delta, \Gamma, g, q_0, F)$
  - Eingabealphabet Σ
  - Ausgabealphabet Γ
  - Ausgabefunktion: $g: \Sigma \rightarrow \Gamma$

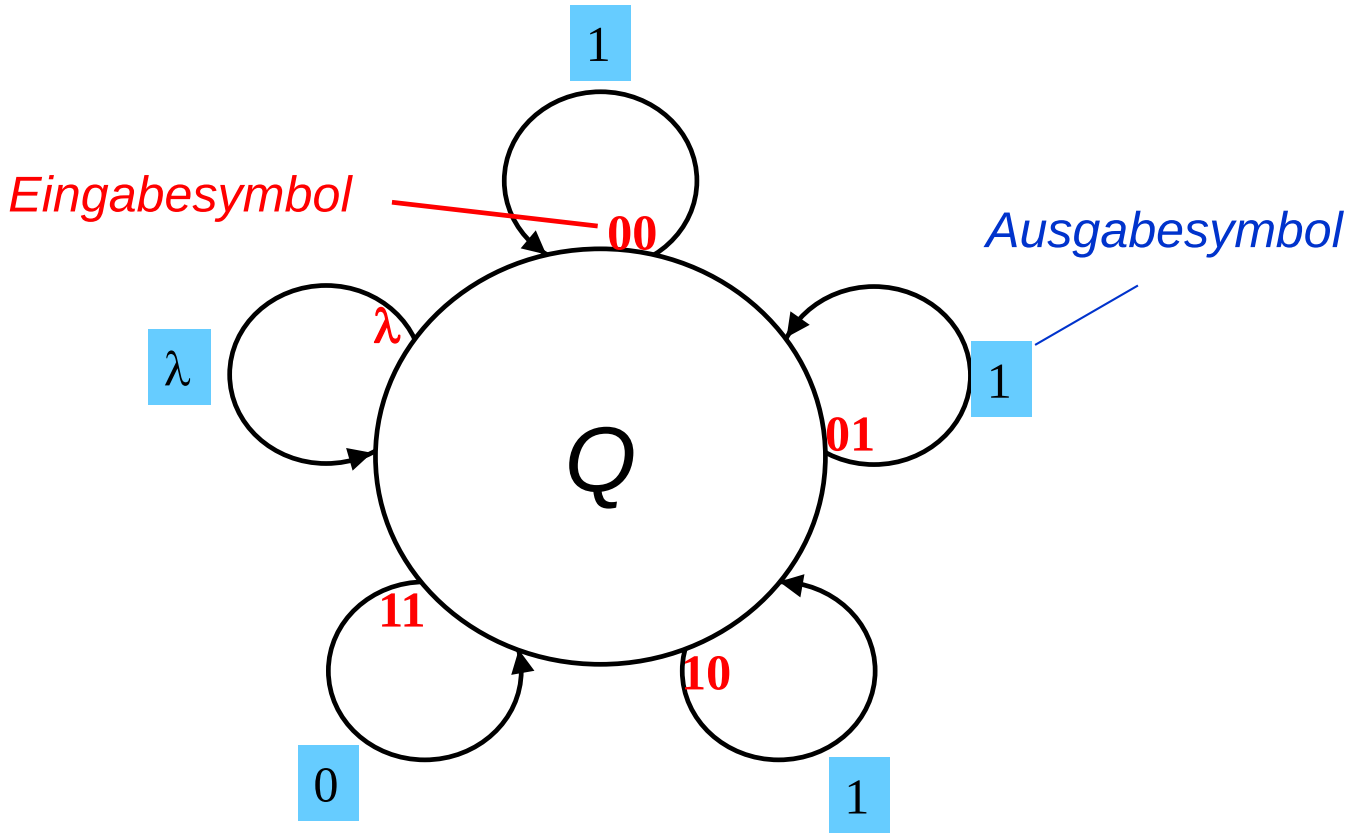
e	01	10	10	00	11	00
a	1	1	1	1	0	1

g Σ	Γ
00	1
01	1
10	1
11	0

NAND-Funktion



Cayley-Diagramm für den NAND-Automaten



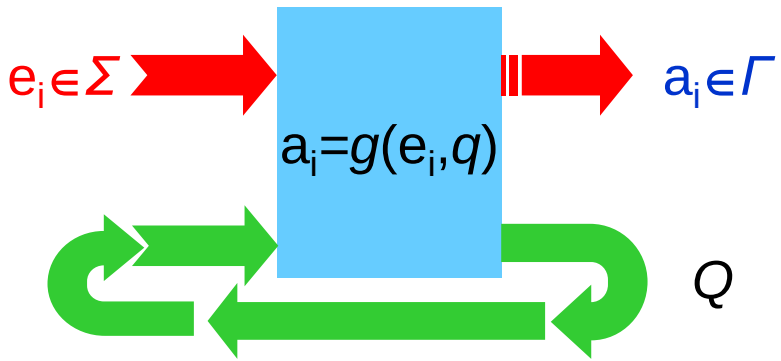
Beispiel 30

- **eingegebene Zeichenfolge ($e_i \in \Sigma$)**

...	λ	λ	e_6	e_5	e_4	e_3	e_2	e_1
= ...	λ	λ	01	10	10	11	00	00

- **Ausgabezeichenfolge ($a_i \in \Gamma$)**

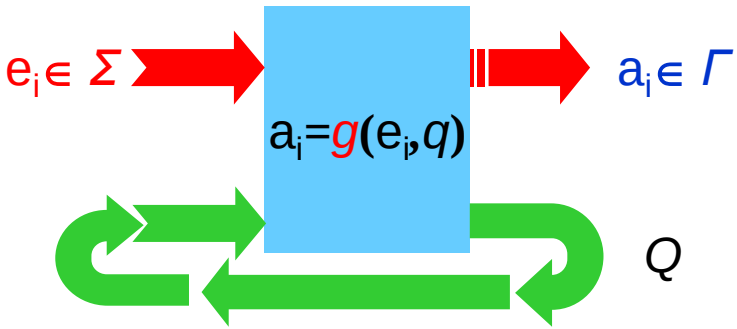
...	λ	a_7	a_6	a_5	a_4	a_3	a_2	a_1
= ...	λ	1	0	0	0	0	0	0



Bitserielle Addition

- **Automat:** $\mathcal{A} = (Q, \Sigma, \delta, \Gamma, g, q_0, F)$

Q	endliche Zustandsmenge
q_0	Anfangszustand
F	Menge der End- oder Finalzustände $F \subseteq Q$
$g : \Sigma \times Q \to \Gamma$	Ausgabefunktion
$\delta : \Sigma \times Q \to Q$	Zustandsübergangsfunktion



Funktion g
bit-serielle Addition
→ ADD-Funktion

Bitserielle Addition

- **Für den Automaten ADD gilt**

Σ	{00,01,10,11} $\cup$ {λ}
Γ	{0,1} $\cup$ {λ}
q_0	$\bar{\text{ü}}$
F	{$\bar{\text{ü}}$}
$g : \Sigma \times Q \rightarrow \Gamma$	*Ausgabefunktion*
$\delta : \Sigma \times Q \rightarrow Q$	*Zustandsübergangsfunktion*

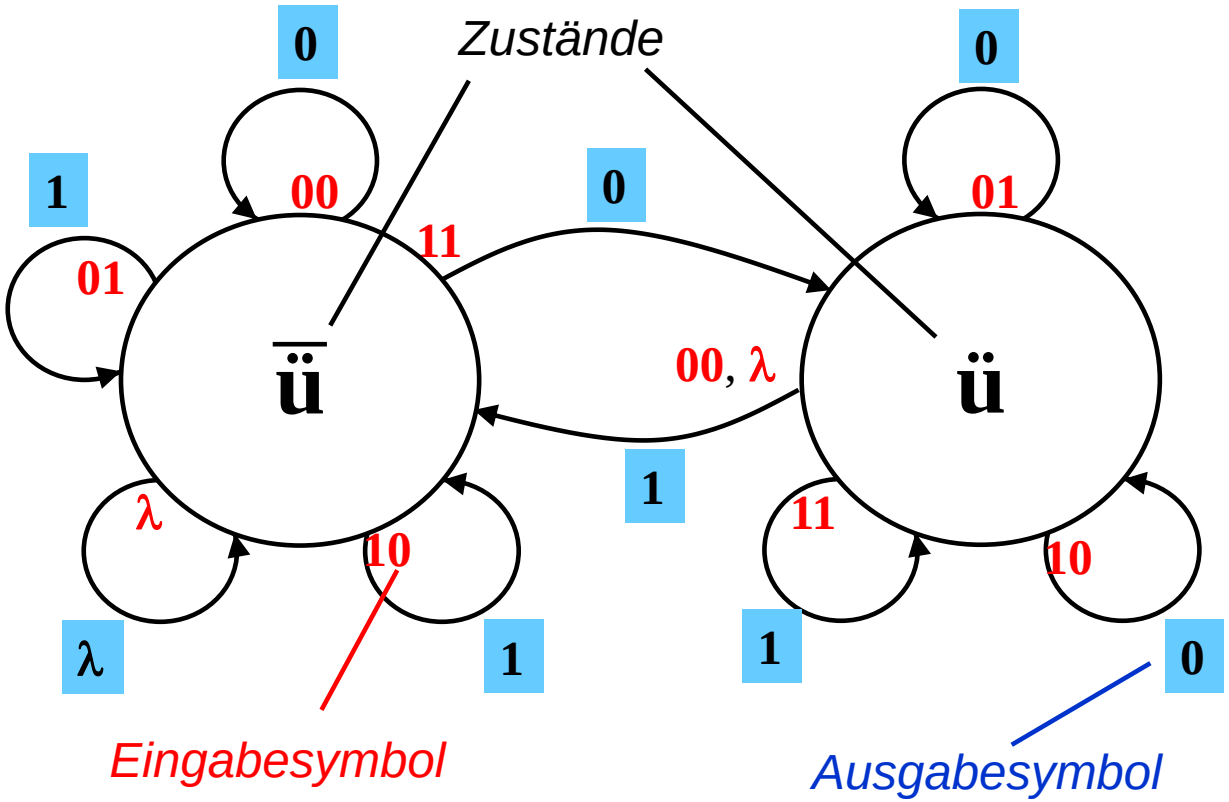
λ : leeres Zeichen

Übertrag vorhanden (ü=1)

oder nicht (ü=0)

Bitserielle Addition

- **Cayley-Diagramm für den ADD-Automaten**

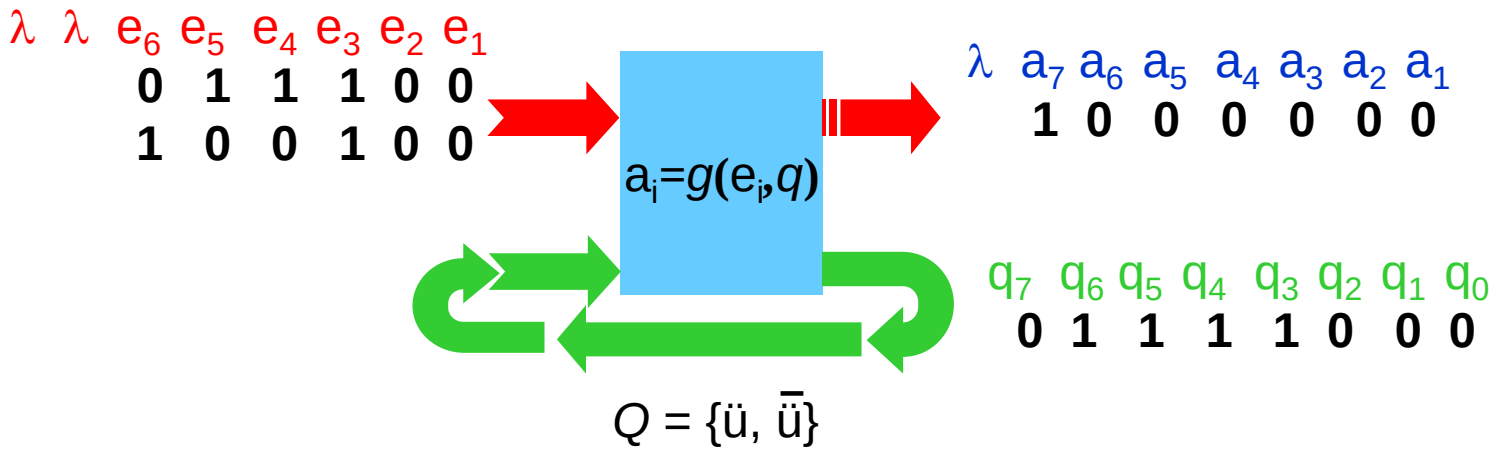


Addition binärer Zahlen

	1	1	1	1	0	0	0
		0	1	1	1	0	0
+		1	0	0	1	0	0
	1	0	0	0	0	0	0

Übertrag - Zustand

Realisierung als endlicher Automat



$Q = \{ü, \bar{ü}\}$

Bitserielle Addition

Ausgabefunktion des DEA:

g :

	λ	00	01	10	11
$\overline{\ddot{u}}$	λ	0	1	1	0
$\ddot{u}$	1	1	0	0	1

Zustandsübergangstabelle des DEA:

δ :

	λ	00	01	10	11
$\overline{\ddot{u}}$	$\overline{\ddot{u}}$	$\overline{\ddot{u}}$	$\overline{\ddot{u}}$	$\overline{\ddot{u}}$	$\ddot{u}$
$\ddot{u}$	$\overline{\ddot{u}}$	$\overline{\ddot{u}}$	$\ddot{u}$	$\ddot{u}$	$\ddot{u}$

Dieser Automat stellt aus dem Eingabewort

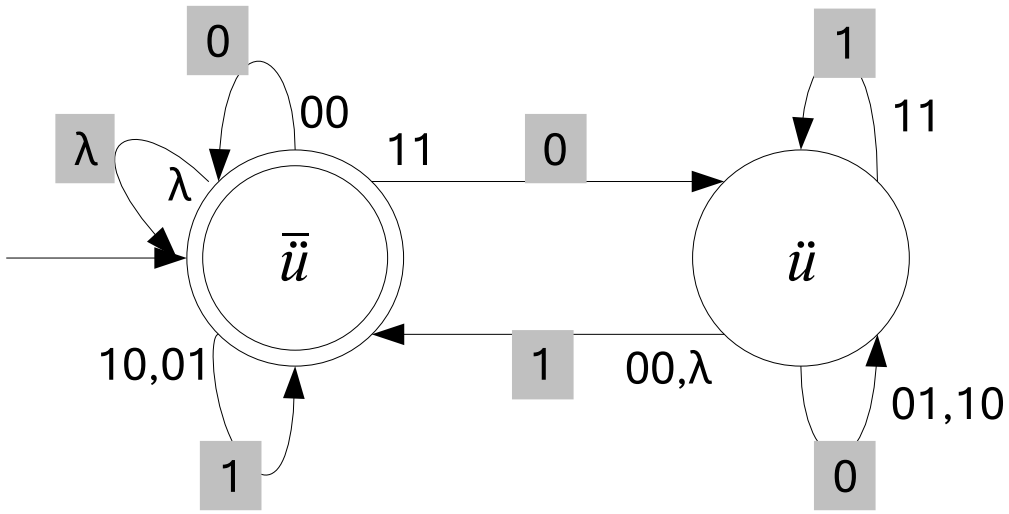
$$\begin{array}{cccccccccc} & \cdots & \lambda & \lambda & e_6 & e_5 & e_4 & e_3 & e_2 & e_1 \\ = & \cdots & \lambda & \lambda & 01 & 10 & 10 & 11 & 00 & 00 \end{array}$$

und dem Anfangszustand $\overline{\text{ü}}$ das Ausgabewort

$$\begin{array}{cccccccccc} & \cdots & \lambda & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 \\ = & \cdots & \lambda & 1 & 0 & 0 & 0 & 0 & 0 & 0 \text{ (Summe)} \end{array}$$

den Endzustand $\overline{\text{ü}}$ her.

Der Automat kann auch durch seinen Zustandsübergangsgraphen dargestellt werden:



Zustandsübergangsgraph des bitseriellen Addierers

4.1 Asynchroner Automat

- der zeitliche Zusammenhang zwischen Eingabe und Ausgabe ist aufgehoben
- zur mathematischen Abbildung werden das Eingabe- und das Ausgabesalphabet durch das leere Wort λ (bzw. ε) ergänzt
- der asynchrone Charakter spiegelt sich in den Funktionen g und δ wie folgt wider:

$g(\lambda, q_j) = w$	Ausgabe eines Wortes ohne Eingabe
$g(\lambda, q_j) = \varepsilon$	keine Eingabe, keine Ausgabe
$\delta(\lambda, q_j) = q_k$	spontaner Übergang ohne Eingabe

Bemerkung:
Endliche Automaten mit ε-Übergängen können in äquivalente (bzgl. der akzeptierten Sprache) deterministische endliche Automaten ohne ε-Übergänge umgewandelt werden.

5. Turing-Maschinen

- Abstraktes Modell für das prinzipielle Verhalten einer rechnenden Maschine
- Alle bisher bekannten Algorithmen können von einer Turing-Machine durchgeführt werden
- Alles, was eine Turing-Maschine tun kann, soll den Begriff Algorithmus beschreiben
- Alle Funktionen/Relationen, die sich dabei als Resultat ergeben, sollen berechenbar heissen.

Die Gleichsetzung des intuitiven Begriffs berechenbar mit dem durch Turingmaschinen gegebenen Berechenbarkeitsbegriff wird als Churchsche These bezeichnet.

5.1 Alan Turing

Geboren: 23. Juni 1912 in London, England
Gestorben: 7. Juni 1954 in Wilmslow, Cheshire, England

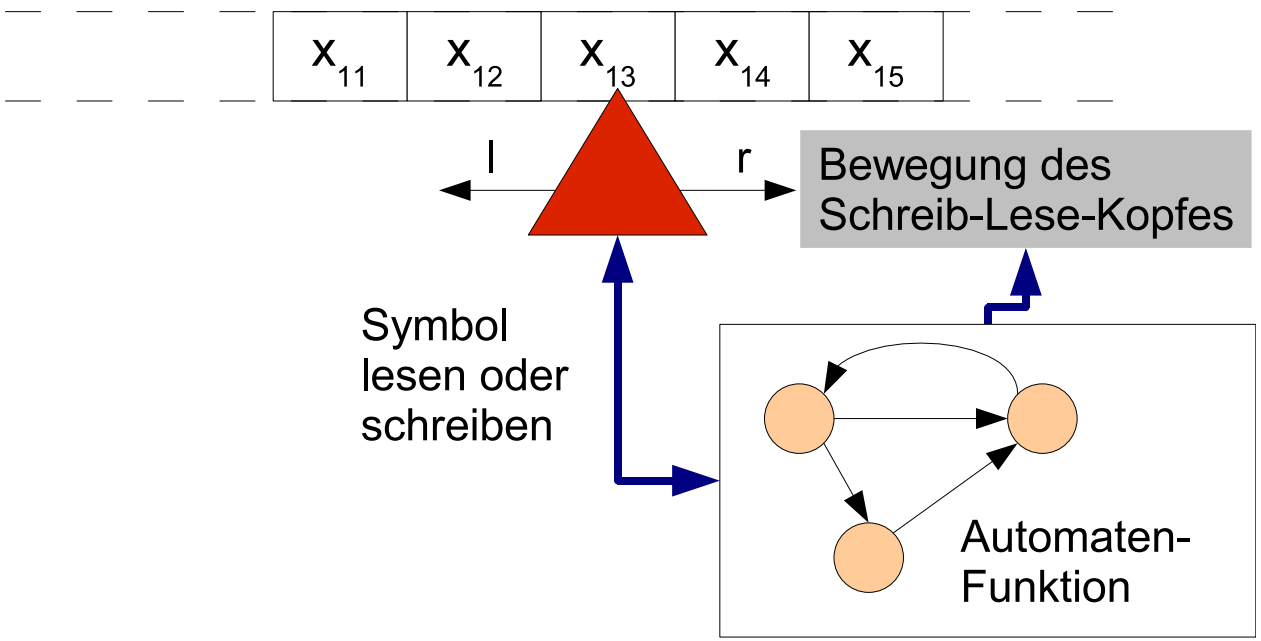
- Turing-Maschine 1936
- Government Code and Cypher School at Bletchley Park
- Entschlüsselung des Enigma Codes (2. WK)
- Hervorragender Sportler
- Turing Test 1950
- 1952 eingesperrt wegen Homosexualität
- Sicherheitsrisiko
- Tod unter teilweise ungeklärten Umständen

Und hier sind noch mehr Details zu Alan Turing.

5.2 Struktur einer Turing-Maschine

Eine Turing-Maschine besteht aus einem

- Band, auf dem Zeichen stehen
- beweglichen (um maximal eine Position) Schreib-Lese-Kopf
- Steuerautomaten



Definition 31

Eine nichtdeterministische Turingmaschine (kurz TM oder NDTM) wird durch ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ beschrieben, das folgende Bedingungen erfüllt:

1. Q ist eine endliche Menge von Zuständen.
2. Σ ist eine endliche Menge, das Eingabealphabet.
3. Γ ist eine endliche Menge, das Bandalphabet, mit $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$ ist die Übergangsfunktion.
5. $q_0 \in Q$ ist der Startzustand.
6. $\square \in \Gamma \setminus \Sigma$ ist das Leerzeichen.
7. $F \subseteq Q$ ist die Menge der (akzeptierenden) Endzustände.

Eine Turingmaschine heißt deterministisch, falls gilt

$$|\delta(q, a)| \leq 1 \qquad \text{für alle } q \in Q, a \in \Gamma.$$

Erläuterung:
Intuitiv bedeutet $\delta(q,a) = (q',b,d)$ bzw. $\delta(q,a) \ni (q',b,d)$:
Wenn sich M im Zustand q befindet und unter dem Schreib-/Lesekopf das Zeichen a steht, so geht M im nächsten Schritt in den Zustand q' über, schreibt an die Stelle des a's das Zeichen b und bewegt danach den Schreib-/Lesekopf um eine Position nach rechts (falls $d = R$), links (falls $d = L$) bzw. lässt ihn unverändert (falls $d = N$).

Beispiel 32

Es soll eine TM angegeben werden, die eine gegebene Zeichenreihe aus $\{0,1\}^+$ als Binärzahl interpretiert und zu dieser Zahl 1 addiert. Folgende Vorgehensweise bietet sich an:

1. Gehe ganz nach rechts bis ans Ende der Zahl. Dieses Ende kann durch das erste Auftreten eines Leerzeichens gefunden werden.
2. Gehe wieder nach links bis zur ersten 0 und ändere diese zu einer 1. Ersetze dabei auf dem Weg alle 1en durch 0.

Also:

$$\begin{aligned} \delta(q_0,0) &= (q_0,0,R) & \qquad \delta(q_1,1) &= (q_1,0,L) \\ \delta(q_0,1) &= (q_0,1,R) & \qquad \delta(q_1,0) &= (q_f,1,N) \\ \delta(q_0,\Box) &= (q_1,\Box,L) & \qquad \delta(q_1,\Box) &= (q_f,1,N) \end{aligned}$$

Damit ist $Q = \{q_0, q_1, q_f\}$ und $F = \{q_f\}$.

Definition 33

Eine Konfiguration einer Turingmaschine ist ein Tupel $(\alpha, q, \beta) \in \Gamma^* \times Q \times \Gamma^*$.

Das Wort $w = \alpha\beta$ entspricht dem Inhalt des Bandes, wobei dieses rechts und links von w mit dem Leerzeichen $\Box$ gefüllt sei. Der Schreib-/Lesekopf befindet sich auf dem ersten Zeichen von $\beta\Box^\infty$.

Die Startkonfiguration der Turingmaschine bei Eingabe $x \in \Sigma^*$ entspricht der Konfiguration

$$(\epsilon, q_0, x)\,,$$

d.h. auf dem Band befindet sich genau die Eingabe $x \in \Sigma^*$, der Schreib-/Lesekopf befindet sich über dem ersten Zeichen der Eingabe und die Maschine startet im Zustand q_0.

Je nach aktuellem Bandinhalt und Richtung $d \in \{L, R, N\}$ ergibt sich bei Ausführung des Zustandsübergangs $\delta(q, \beta_1) = (q', c, d)$ folgende Änderung der Konfiguration:

$$(\alpha_1 \cdots \alpha_n, q, \beta_1 \cdots \beta_m) \rightarrow \begin{cases} (\alpha_1 \cdots \alpha_n, q', c\beta_2 \cdots \beta_m) & \text{falls } d = N, \\ & n \geq 0, m \geq 1 \\ (\epsilon, q', \square c\beta_2 \cdots \beta_m) & \text{falls } d = L, \\ & n = 0, m \geq 1 \\ (\alpha_1 \cdots \alpha_{n-1}, q', \alpha_n c\beta_2 \cdots \beta_m) & \text{falls } d = L, \\ & n \geq 1, m \geq 1 \\ (\alpha_1 \cdots \alpha_n c, q', \square) & \text{falls } d = R, \\ & n \geq 0, m = 1 \\ (\alpha_1 \cdots \alpha_n c, q', \beta_2 \cdots \beta_m) & \text{falls } d = R, \\ & n \geq 0, m \geq 2 \end{cases}$$

Der Fall $m = 0$ wird mittels $\beta_1 = \square$ abgedeckt.

Definition 34

Die von einer Turingmaschine M akzeptierte Sprache ist

$$L(M) = \{x \in \Sigma^*;\ (\epsilon, q_0, x) \to^* (\alpha, q, \beta) \text{ mit } q \in F, \alpha, \beta \in \Gamma^*\}$$

Die Turing-Maschine kann

- eine (ggf partielle!) Funktion berechnen
- (formale) Sprachen akzeptieren bzw. erkennen (Sprachakzeptor)

Es gibt viele Variationen der Turing-Maschine, z.B.

- Mehrbandige Turing-Maschine
- Turing-Maschine mit einseitig oder zweiseitig unendlichem Band
- Mehrdimensionale Turing-Maschine
- Mehrköpfige Turing-Maschine
- Turing-Maschine mit separatem Ein- und Ausgabeband
- Online-Turing-Maschine

Bemerkung: Jede NDTM N (die die Sprache $L(N)$ akzeptiert) kann in eine deterministische Turing-Maschine M konvertiert werden, die ebenfalls genau die Sprache $L(N)$ akzeptiert.

Beweisidee: Die Berechnungspfade der NDTM werden von der DTM Zeitschritt für Zeitschritt simuliert, und zwar in BFS-Manier (breadth-first-search), d.h., es werden alle Berechnungspfade um einen Zeitschritt verlängert, bevor der nächste Zeitschritt in Angriff genommen wird.

6. Linear beschränkte Automaten

Definition 35

Eine Turingmaschine heißt linear beschränkt (kurz: LBA), falls für alle $q \in Q$ gilt:

$$(q', c, d) \in \delta(q, \Box) \qquad \Longrightarrow \qquad c = \Box\,.$$

Ein Leerzeichen wird also nie durch ein anderes Zeichen überschrieben. Mit anderen Worten: Die Turingmaschine darf ausschliesslich die Positionen beschreiben, an denen zu Beginn die Eingabe x steht.

7. Kellerautomaten

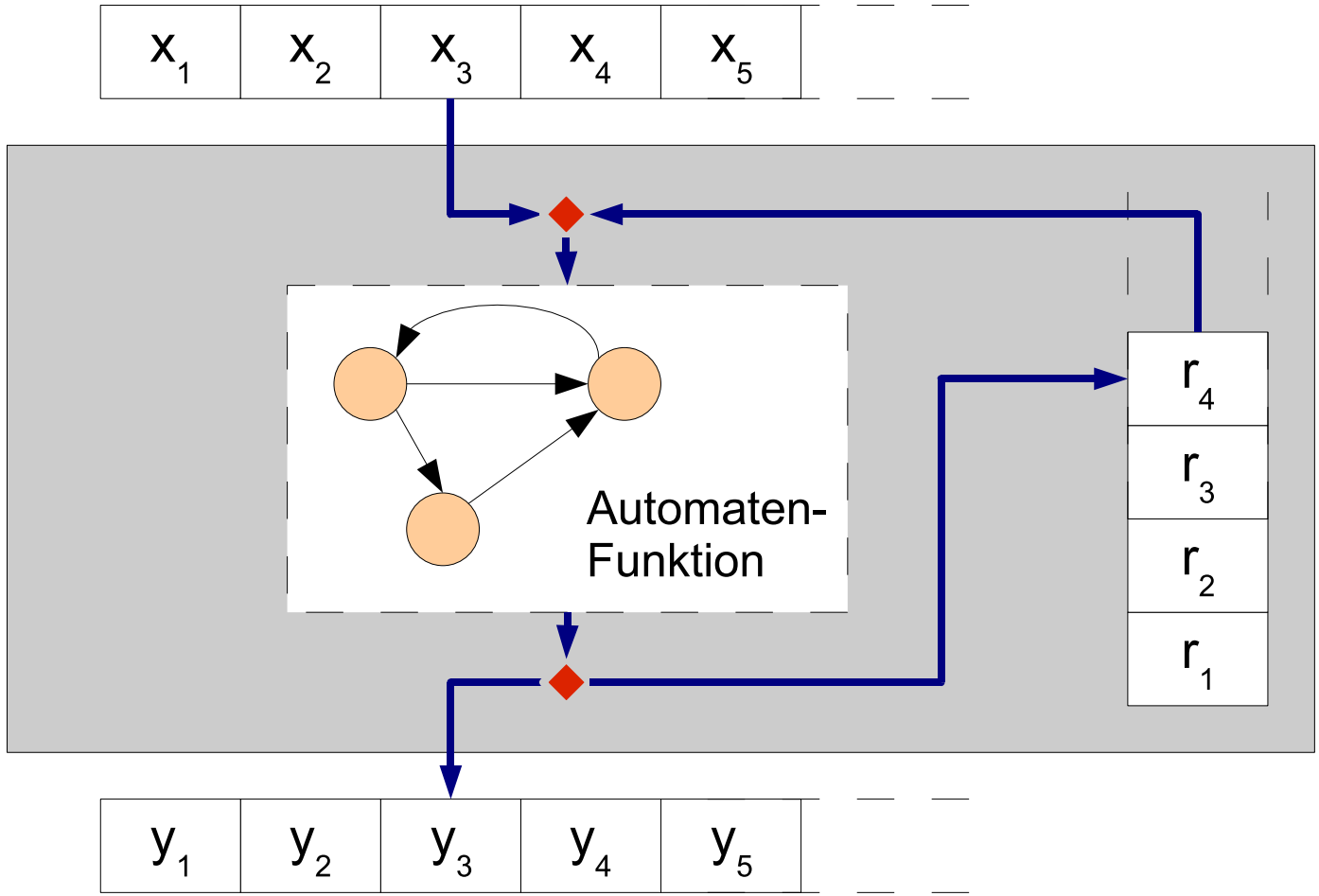
In der Literatur findet man häufig auch die Bezeichnungen Stack-Automat oder Pushdown-Automat. Kellerautomaten sind, wenn nichts anderes gesagt wird, nichtdeterministisch.

Definition 36

Ein NPDA = PDA (= Nichtdeterministischer Pushdown-Automat) besteht aus:

Q	endliche Zustandsmenge
Σ	endliches Eingabealphabet
Δ	endliches Stackalphabet
$q_0 \in Q$	Anfangszustand
$Z_0 \in \Delta$	Initialisierung des Stack
δ	Übergangsrelation
	Fkt. $Q \times (\Sigma \cup \{\epsilon\}) \times \Delta \to 2^{Q \times \Delta^*}$
	wobei $\lvert\delta(q,a,Z)\rvert + \lvert\delta(q,\epsilon,Z)\rvert < \infty \quad \forall q,a,Z$
$F \subseteq Q$	akzeptierende Zustände

Der Kellerautomat



Konfiguration:

Tupel (q, w, α) mit

$$\begin{aligned} q &\in Q \\ w &\in \Sigma^* \\ \alpha &\in \Delta^* \end{aligned}$$

Schritt:

$(q, w_0 w', Z\alpha') \to (q', w', Z_1 \dots Z_r \alpha')$
wenn $(q', Z_1 \dots Z_r) \in \delta(q, w_0, Z)$
bzw.:
$(q, w, Z\alpha') \to (q', w, Z_1 \dots Z_r \alpha')$
wenn $(q', Z_1 \dots Z_r) \in \delta(q, \epsilon, Z)$

Definition 37

1. Ein NPDA A akzeptiert $w \in \Sigma^*$ durch leeren Stack, falls

$$(q_0, w, Z_0) \to^* (q, \epsilon, \epsilon) \text{ für ein } q \in Q \ .$$

2. Ein NPDA A akzeptiert $w \in \Sigma^*$ durch akzeptierenden Zustand, falls

$$(q_0, w, Z_0) \to^* (q, \epsilon, \alpha) \text{ für ein } q \in F,\ \alpha \in \Delta^* \ .$$

3. Ein NPDA heißt deterministisch (DPDA), falls

$$|\delta(q, a, Z)| + |\delta(q, \epsilon, Z)| \leq 1 \qquad \forall (q, a, Z) \in Q \times \Sigma \times \Delta \ .$$

Beispiel 38

Der PDA mit

$$
\begin{array}{lcll}
\delta(q_0, a, *) & = & \{(q_0, a*)\} & \text{für } a \in \{0,1\}, * \in \{0,1,Z_0\} \\
\delta(q_0, \#, *) & = & \{(q_1, *)\} & \text{für } * \in \{0,1,Z_0\} \\
\delta(q_1, 0, 0) & = & \{(q_1, \epsilon)\} & \\
\delta(q_1, 1, 1) & = & \{(q_1, \epsilon)\} & \\
\delta(q_1, \epsilon, Z_0) & = & \{(q_1, \epsilon)\} & \text{akzeptiert mit leerem Stack}
\end{array}
$$

die Sprache

$$L = \{w\#w^R;\ w \in \{0,1\}^*\}\,.$$

Beispiel 38

Der PDA mit

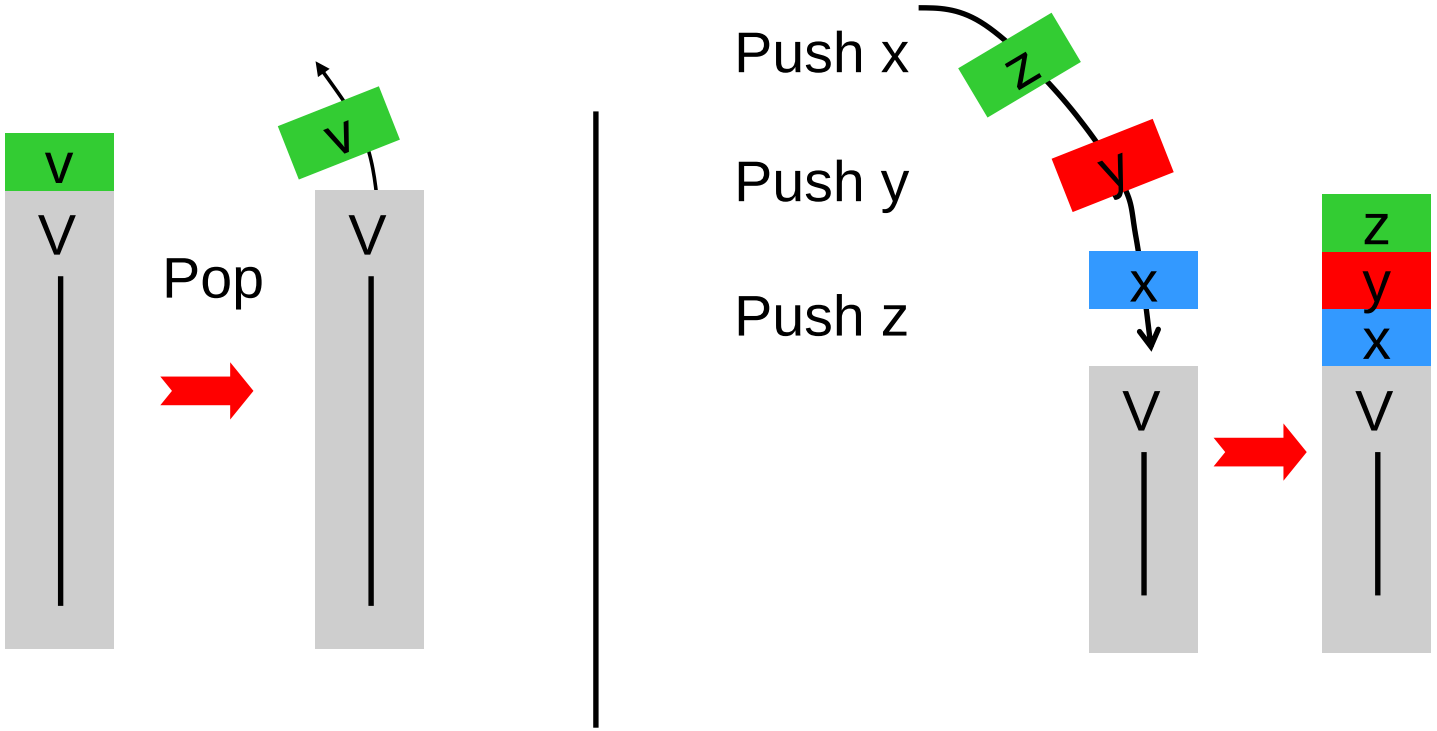
$\delta(q_0, a, *)$	$=$	$\{(q_0, a*)\}$	für $a \in \{0,1\}, * \in \{0,1,Z_0\}$
$\delta(q_0, \#, *)$	$=$	$\{(q_1, *)\}$	für $* \in \{0,1,Z_0\}$
$\delta(q_1, 0, 0)$	$=$	$\{(q_1, \epsilon)\}$	
$\delta(q_1, 1, 1)$	$=$	$\{(q_1, \epsilon)\}$	
$\delta(q_1, \epsilon, Z_0)$	$=$	$\{(q_a, \epsilon)\}$	akzeptiert mit akzeptierendem Zustand ($F = \{q_a\}$) (und leerem Stack)

die Sprache

$$L = \{w\#w^R;\ w \in \{0,1\}^*\}\,.$$

Die möglichen Operationen eines Kellerautomaten sind also

- Einlesen eines Eingabesymbols x_i, evtl. des Leersymbols λ
- Bestimmung des nächsten Zustandes
- Zustandsübergang hängt (i.a. nichtdeterministisch) vom Zustand, vom Eingabesymbol und vom obersten Symbol des Kellerspeichers ab
- Entfernen des Symbols am oberen Ende des Kellerspeichers (POP)
- Symbol(e) auf das obere Ende des Kellerspeichers schreiben (PUSH)



Push- und Pop-Operationen auf dem Kellerspeicher

8. Deterministische Kellerautomaten

Wir haben bereits definiert:
Ein PDA heißt deterministisch (DPDA), falls

$$|\delta(q,a,Z)| + |\delta(q,\epsilon,Z)| \leq 1 \qquad \forall (q,a,Z) \in Q \times \Sigma \times \Delta \; .$$

Die von einem DPDA, der mit leerem Keller akzeptiert, erkannte Sprache genügt der Fano-Bedingung, d.h. kein Wort in der Sprache ist echtes Präfix eines anderen Wortes in der Sprache.

Festlegung:
Da wir an einem weniger eingeschränkten Maschinenmodell interessiert sind, legen wir fest, dass ein DPDA stets mit akzeptierenden Zuständen akzeptiert.

Definition 39

Ein DPDA ist in Normalform, falls gilt:

1. $(q', \alpha) = \delta(q, e, X)$ für $e \in \Sigma \cup \{\epsilon\}$, $q, q' \in Q$, $X \in \Delta$

$$\Rightarrow \alpha \in \{\epsilon, X, YX\} \text{ für } Y \in \Delta \; .$$

2. Der Automat liest jede Eingabe vollständig.

Satz 40

Zu jedem DPDA $A = (Q, \Sigma, \Delta, q_0, Z_0, \delta, F)$ kann ein äquivalenter DPDA in Normalform konstruiert werden.

Beweis:

Erste Schritte der Konstruktion:

1. Werden von A in einem Schritt mehr als zwei Symbole auf dem Stack abgelegt, wird dies von A' durch eine Folge von Schritten mit je 2 Stacksymbolen ersetzt.
2. Werden zwei oder ein Stacksymbol abgelegt und dabei das oberste Stacksymbol X geändert, entfernen wir zunächst in einem eigenen Schritt das oberste Stacksymbol und pushen dann die gewünschten Symbole. (Das „Merken“ erfolgt in der Zustandsmenge Q'.)
3. Wir vervollständigen δ' mittels eines (nicht akzeptierenden) Fangzustandes. Es könnte hier noch sein, dass der DPDA ab einem Zeitpunkt nur mehr und unbegrenzt viele ϵ-Übergänge ausführt.

Beweis (Forts.):

Hilfsbehauptung:

Der DPDA A führt ab einer bestimmten Konfiguration (q, ϵ, β) unendlich viele direkt aufeinander folgende ϵ-Übergänge genau dann aus, wenn

$$\begin{array}{rcll} (q, \epsilon, \beta) & \rightarrow^* & (q', \epsilon, X\beta') & \text{und} \\ (q', \epsilon, X) & \rightarrow^+ & (q', \epsilon, X\alpha) & \text{für } q, q' \in Q \\ & & & X \in \Delta,\ \alpha, \beta, \beta' \in \Delta^* \end{array}$$

„$\Leftarrow$“: klar

Beweis (Forts.):

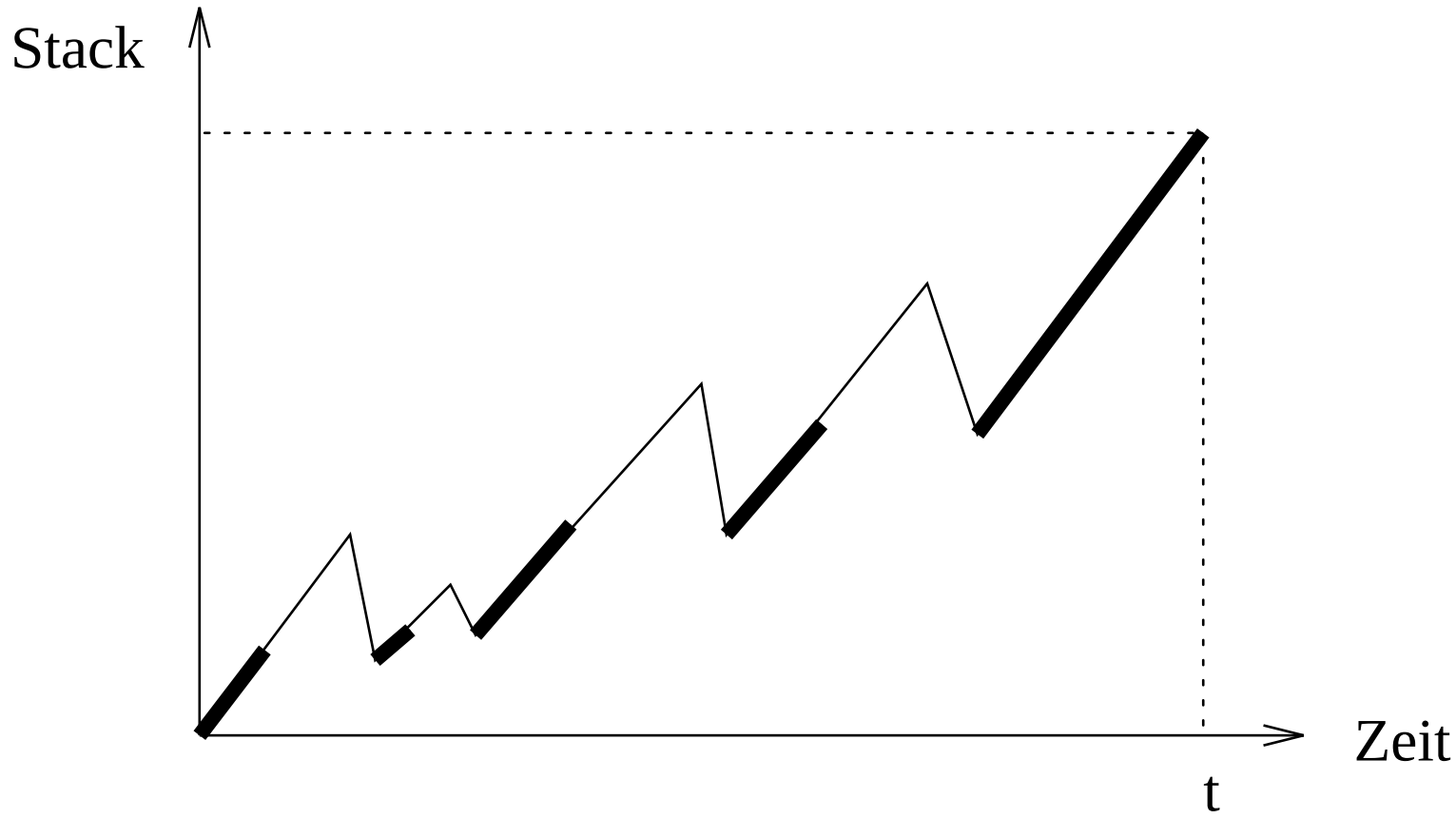
„$\Rightarrow$“: Betrachte eine unendlich lange Folge von ϵ-Übergängen.
Sei $n := |Q| \cdot |\Delta| + |\beta| + 1$.

Wird die Stackhöhe n nie erreicht, so muss sich sogar eine Konfiguration des DPDA's wiederholen. Daraus folgt die Behauptung.

Beweis (Forts.):

Ansonsten wird jede Stackhöhe $|\beta|, \ldots, n$ mindestens einmal erreicht (wegen der Normalform ist die Höhendifferenz pro Schritt $\in \{-1, 0, 1\}$).
Betrachte den Zeitpunkt t, in dem die Stackhöhe zum erstenmal n ist. Markiere für jedes $i \in \{|\beta|, \ldots, n\}$ den Zeitpunkt t_i, wo zum letzen Mal (vor Zeitpunkt t) die Stackhöhe $= i$ ist. Zu diesen Zeitpunkten t_i betrachte die Paare $(q, X) \in Q \times \Delta$, wobei q der Zustand des DPDA's und X das oberste Kellersymbol des DPDA's zu diesem Zeitpunkt ist.
Da es mehr als $|\Delta| \cdot |Q|$ markierte Paare gibt, taucht ein markiertes Paar (q', X) doppelt auf. Für dieses gilt dann
$(q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha)$.

Beweis (Forts.):



Beweis (Forts.):

Das gleiche Argument gilt, falls sich die Stackhöhe um $> |Q| \cdot |\Delta|$ erhöht.

Damit lassen sich alle Paare (q', X) finden, für die gilt:

$$(q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha),\ \alpha \in \Delta^*.$$

Da der DPDA nicht endlos weiterlaufen soll, ersetzen wir $\delta(q', \epsilon, X)$ durch einen ϵ-Übergang in einen neuen Zustand q'' (der genau dann akzeptierend ist, wenn in der Schleife $(q', \epsilon, X) \rightarrow^+ (q', \epsilon, X\alpha)$ ein akzeptierender Zustand auftritt) und einen ϵ-Übergang von q'' in den nichtakzeptierenden Fangzustand. Die Details dieser Konstruktion werden nun beschrieben.

Beweis (Forts.):

Wir modifizieren den DPDA A in mehreren Schritten wie folgt:

1. *A merkt sich das oberste Kellersymbol im Zustand:*

$$Q' := \{q_{\text{pop}};\ q \in Q\} \cup \{qX;\ q \in Q, X \in \Delta\}$$

Für die Übergangsrelation δ' setzen wir (für $e \in \Sigma \cup \{\epsilon\}$)

$$\delta'(qX, e, X) := \begin{cases} (pY, YX) \\ (pX, X) \\ (p_{\text{pop}}, \epsilon) \end{cases} \qquad \text{falls } \delta(q, e, X) = \begin{cases} (p, YX) \\ (p, X) \\ (p, \epsilon) \end{cases}$$

Im dritten Fall kommt noch

$$\delta'(p_{\text{pop}}, \epsilon, X) := (pX, X)$$

für alle $X \in \Delta$ dazu.

Beweis (Forts.):

Weiter

$$q_0' := q_0 Z_0$$
$$F' := \{qX;\ q \in F, X \in \Delta\}$$

Ein Zustand q' heißt spontan, falls q' von der Form p_{pop} (und damit $\delta'(q', \epsilon, X)$ für alle $X \in \Delta$ definiert) ist oder falls

$$q' = qX$$

und $\delta'(qX, \epsilon, X)$ definiert ist.

Beweis (Forts.):

Wir erweitern Q' um einen neuen Zustand f, $f \notin F'$, der als Fangzustand dient:

— für alle $q' = qX$, q' nicht spontan, $a \in \Sigma$, so dass $\delta'(qX, a, X)$ nicht definiert ist, setze

$$\delta'(qX, a, X) := (f, X)\,;$$

— setze

$$\delta'(f, a, X) := (f, X)$$

für alle $a \in \Sigma$, $X \in \Delta$.

Beweis (Forts.):

Bemerkungen:

1. f ist nicht spontan, f ist kein (akzeptierender) Endzustand.
2. Für alle nicht-spontanen $q' \in Q'$ von der Form $q' = qX$ ist $\delta'(q', a, X)$ für alle $a \in \Sigma$ definiert.
3. $\delta'(f, a, X)$ ist für alle $a \in \Sigma$ und $X \in \Delta$ definiert.

Falls sich der DPDA also in einem nicht-spontanen Zustand befindet und ein weiteres Eingabezeichen zur Verfügung steht, wird dieses gelesen!

2. *Endliche Gedächtniserweiterung:* Wir erweitern den DPDA so, dass er sich eine vorgegebene endliche Menge von Alternativen merken kann. Ersetzen wir z.B. Q' durch $Q' \times \{0,1\}^m$, so kann sich der Automat Information im Umfang von m Bits (also 2^m Alternativen) merken und diese bei Übergängen fortschreiben.

Der neue Anfangszustand, die Menge der (akzeptierenden) Endzustände und die neue Übergangsrelation werden entsprechend der intendierten Semantik des endlichen Speichers festgelegt.

Sei A' der DPDA vor der „Speichererweiterung“. Wir erweitern A' zu A'', so dass A'' sich ein zusätzliches Bit im Zustand merken kann. Dieses Bit ist im Anfangszustand von A'' gleich 0. Bei einem Zustandsübergang von A'', der einem Zustandsübergang von A' aus einem spontanen Zustand q' entspricht, gilt: Ist $q' \in F'$, so wird das Bit im Zustand nach dem Übergang gesetzt, ansonsten kopiert. Entspricht der Zustandsübergang von A'' einem Zustandsübergang von A' aus einem nicht-spontanen Zustand, so wird das Bit gelöscht.

Beweis (Forts.):

Der Fangzustand f mit gesetztem Bit (i.Z. $f^{(1)}$) wird nun (akzeptierender) Endzustand, f mit nicht gesetztem Bit (i.Z. $f^{(0)}$) bleibt Nicht-Endzustand.

3. *Entfernung unendlicher Folgen von ϵ-Übergängen:* Für alle Zustände $q' = qX$ von A', für die gilt

$$(q', \epsilon, X) \to^+ (q', \epsilon, X\alpha)\,,$$

setzen wir

$$\delta'(q', \epsilon, X) := (f, X)\,.$$

In A'' setzt dieser Übergang das Speicherbit, falls die obige Schleife einen Endzustand enthält (womit A'' in $f^{(1)}$ endet), ansonsten wird das Speicherbit kopiert.

Beweis (Forts.):

Bemerkung: Wenn wir weiter (und o.B.d.A.) voraussetzen, dass A (bzw. A', A'') seinen Keller nie leert, gilt: Der soeben konstruierte DPDA akzeptiert/erkennt ein Eingabewort w gdw er w in einem nicht-spontanen Zustand akzeptiert/erkennt. □

Satz 41

Die Klasse der deterministischen kontextfreien Sprachen (also der von DPDA's erkannten Sprachen) [DCFL] ist unter Komplement abgeschlossen.

Beweis:

Sei A ein DPDA, A' ein daraus wie oben beschrieben konstruierter äquivalenter DPDA. O.B.d.A. sind in A' alle Endzustände $q \in F'$ nicht spontan.

Sei $N \subseteq Q'$ die Menge aller nicht-spontanen Zustände von A'. Konstruiere den DPDA $\bar{A}$, indem in A' F' durch $N \setminus F'$ ersetzt wird. Dann ergibt sich aus der vorangehenden Konstruktion direkt

$$L(\bar{A}) = \overline{L(A)}\,.$$

□

Kapitel IV Formale Sprachen und Grammatiken

1. Begriffe und Notationen

Sei Σ ein (endliches) Alphabet. Dann

Definition 42

1. ist Σ^* das Monoid über Σ, d.h. die Menge aller endlichen Wörter über Σ;
2. ist Σ^+ die Menge aller nichtleeren endlichen Wörter über Σ;
3. bezeichnet $|w|$ für $w \in \Sigma^*$ die Länge von w;
4. ist Σ^n für $n \in \mathbb{N}_0$ die Menge aller Wörter der Länge n in Σ^*;
5. eine Teilmenge $L \subseteq \Sigma^*$ eine formale Sprache.

2. Sprachkonzept

- Eine (formale) Sprache L besteht aus Wörtern w über einem Alphabet Σ
- Eine formale Sprache L ist also i.a. einfach eine Teilmenge von Σ^*:

$$L \subseteq \Sigma^*$$

- Wir betrachten insbesondere formale Sprachen, deren Wörter entlang gewisser Regeln erzeugt werden
- Diese Regeln werden als Grammatik G für die Sprache bezeichnet
- Die Menge der so bildbaren Wörter heißt Wortschatz oder Sprachschatz der Grammatik

- Eine Grammatik wird durch
  1. Σ, eine endliche Menge von Terminalzeichen (Alphabet)
  2. V, eine endliche Menge von Nichtterminalzeichen (grammatische Begriffe)
  3. P, eine endliche Menge von Produktionen (Ableitungs-/Ersetzungsregeln, grammatische Regeln)
  4. S, ein spezielles Element aus V, das Axiom oder Startsymbol

  vollständig beschrieben.
- Die Vereinigungsmenge $V \cup \Sigma$ der Nichtterminale und Terminale heißt das Vokabular der Grammatik.
- V und Σ werden o.B.d.A. als disjunkt vorausgesetzt, d.h.

$$V \cap \Sigma = \emptyset$$

- die Produktionen $p \in P$ haben ganz allgemein die Gestalt $A ::= B$, mit $A, B \in (V \cup \Sigma)^*$

Beispiel 43

Wir betrachten folgende Grammatik:

$$
\begin{aligned}
\langle \text{Satz} \rangle &::= \langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle \\
\langle \text{Subjekt} \rangle &::= \langle \text{Artikel} \rangle \langle \text{Attribut} \rangle \langle \text{Substantiv} \rangle \\
\langle \text{Artikel} \rangle &::= \epsilon \\
\langle \text{Artikel} \rangle &::= \text{der} | \text{die} | \text{das} | \text{ein} | \ldots \\
\langle \text{Attribut} \rangle &::= \epsilon | \langle \text{Adjektiv} \rangle | \langle \text{Adjektiv} \rangle \langle \text{Attribut} \rangle \\
\langle \text{Adjektiv} \rangle &::= \text{gross} | \text{klein} | \text{schön} | \ldots
\end{aligned}
$$

Die vorletzte Ersetzungsregel ist rekursiv, die durch diese Grammatik definierte Sprache deshalb unendlich.
Zur Darstellungsform dieser Grammatik später mehr!

Beispiel 44

- $L_1 = \{aa, aaaa, aaaaaa, \ldots\} = \{(aa)^n, n \in \mathbb{N}\}$
  $(\Sigma_1 = \{a\})$
- $L_2 = \{ab, abab, ababab, \ldots\} = \{(ab)^n, n \in \mathbb{N}\}$
  $(\Sigma_2 = \{a, b\})$
- $L_3 = \{ab, aabb, aaabbb, \ldots\} = \{a^n b^n, n \in \mathbb{N}\}$
  $(\Sigma_3 = \{a, b\})$
- $L_4 = \{a, b, aa, ab, bb, aaa, aab, abb, bbb \ldots\}$
  $= \{a^m b^n, m, n \in \mathbb{N}_0, m + n > 0\}$ $\quad (\Sigma_4 = \{a, b\})$

2.1 Erzeugung und Sprachschatz

- Sei G eine Grammatik mit Terminalalphabet Σ, Nichtterminalalphabet V, Produktionen P, $p = A ::= B \ \in P$ und seien $x, y \in (\Sigma \cup V)^*$
- y heißt mittels G aus x in einem Schritt ableitbar, falls es $v, w \in (\Sigma \cup V)^*$ gibt, so dass

$$x = vAw \text{ und } y = vBw$$

- y wird also aus x erzeugt, indem man in x ein Vorkommen der linken Seite von p durch deren rechte Seite ersetzt
- Wir schreiben dafür i.Z. auch

$$x \rightarrow_G y$$

- Überhaupt schreiben wir dementsprechend auch für Produktionen $A ::= B$ oft $A \rightarrow_G B$ (oder nur $A \rightarrow B$)

- Mit $\rightarrow^*$ bezeichnen wir die reflexive, transitive Hülle von $\rightarrow$.
- Es gilt also $x \rightarrow^* y$ gdw es $x^{(0)}, x^{(1)}, \ldots, x^{(k)} \in (\Sigma \cup V)^*$ gibt, so dass $k \geq 0$, $x = x^{(0)}$ und $y = x^{(k)}$ und

$$x^{(i)} \rightarrow_G x^{(i+1)} \text{ für alle } i = 0, \ldots, k-1$$

- Falls S das Axiom der Grammatik G ist und $S \rightarrow_G x$ gilt, so nennen wir x auch eine (mittels G erzeugbare) Sprach- oder Satzform

2.2 Darstellungsformen

2.2.1 Backus-Naur-Form

- Zur Beschreibung der Programmiersprache ALGOL 60 erstmals benutzt
- Nichtterminale werden durch Winkelklammern kenntlich gemacht, z.B. $<$ Ausdruck $>$, diese können entfallen, wenn weiterhin die Eindeutigkeit der Regeln gegeben ist
- Produktionsregeln mit identischer linker Seite können mit dem „|“ Symbol zusammengefasst werden:
  $A ::= W_1$ und $A ::= W_2$ wird so zu $A ::= W_1|W_2$
- Im Rahmen der Vorlesung werden häufig Kleinbuchstaben für Terminale und Großbuchstaben für Nichtterminale gewählt

- Zur Vereinfachung von rekursiven Produktionen (das zu definierende Nichtterminal steht auch auf der rechten Seite), werden Metasymbole verwendet:
  - ( ) Klammer für Zusammenfassung
  - [ ] Inhalt der Klammer optional (0-mal oder 1-mal)
  - { } Inhalt der Klammer beliebig oft mal (0,1,2,...)
- Diese erweiterte Notation wird als ExtendedBNF oder EBNF bezeichnet

Beispiel 45 (Grammatik für die Menge aller arithmetischen Ausdrücke in BNF)

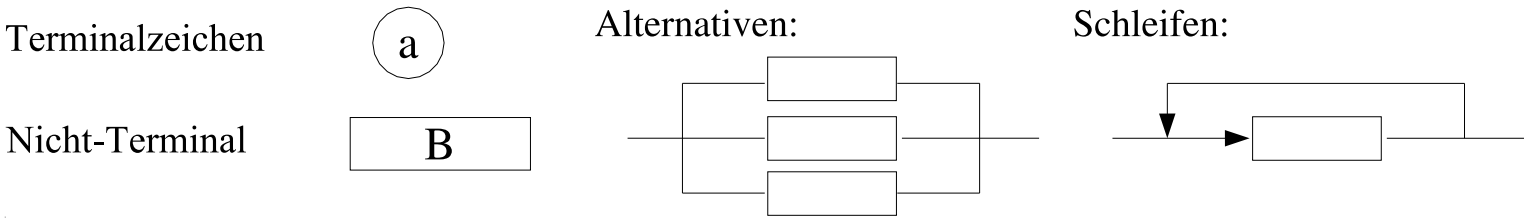
$$
\begin{aligned}
< \text{Terminal} > &::= + \mid - \mid * \mid / \mid \uparrow \mid (\mid) \mid v \mid z \\
< \text{Ausdruck} > &::= < \text{Term} > \mid + < \text{Term} > \mid - < \text{Term} > \mid \\
&\quad < \text{Ausdruck} > + < \text{Term} > \mid \\
&\quad < \text{Ausdruck} > - < \text{Term} > \\
< \text{Term} > &::= < \text{Faktor} > \mid < \text{Term} > * < \text{Faktor} > \mid \\
&\quad < \text{Term} > / < \text{Faktor} > \\
< \text{Faktor} > &::= < \text{Elementarausdruck} > \mid \\
&\quad < \text{Faktor} > \uparrow < \text{Elementarausdruck} > \\
< \text{Elementarausdruck} > &::= z \mid v \mid (< \text{Ausdruck} >)
\end{aligned}
$$

Einsatz der EBNF-Metasymbole vereinfacht die Produktionsregel für $<$ Faktor $>$ zu

$$< \text{Faktor} > ::= \{< \text{Faktor} > \uparrow\} < \text{Elementarausdruck} >$$

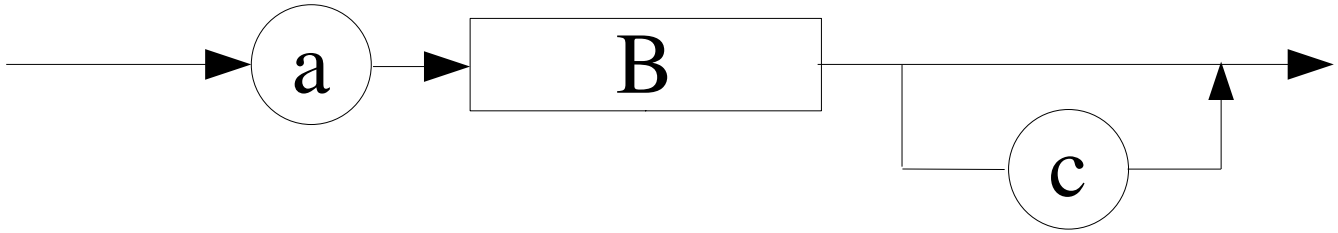
2.2.2 Syntaxdiagramme

- Produktionsregeln der BNF können direkt in ein Syntaxdiagramm überführt werden
- Terminalzeichen sind durch Kreise bzw. abgerundete Rechtecke, syntaktischen Variablen durch Rechtecke dargestellt
- Produktionsregeln mit Wiederholungen und Alternativen werden durch Verbindungen zwischen den Elementen dargestellt



Graphische Darstellung der BNF

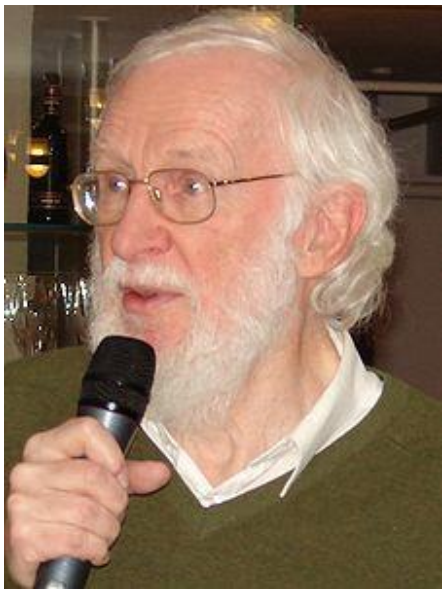
Beispiel 46



Syntaxdiagramm der Produktion $S ::= aB[c]$

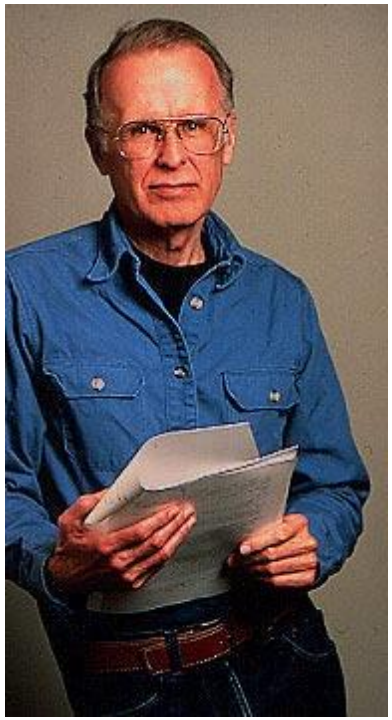
Peter Naur

- 25. Oktober 1928
- Dänischer Informatiker
- Niels Bohr Institute
  Technical University of Denmark
  Copenhagen University
- Entwicklung von ALGOL 60
- 2005 ACM Turing Award



John Backus

- 3. Dezember 1924 – 17. März 2007
- Amerikanischer Informatiker
- IBM-Mitarbeiter
- Entwicklung von FORTRAN
- 1977 ACM Turing Award



3. Die Chomsky-Hierarchie

Diese Sprachenhierarchie ist nach Noam Chomsky [MIT, 1976] benannt.

3.1 Phrasenstrukturgrammatik, Chomsky-Grammatik

Grammatiken bestehen aus

1. einem Terminalalphabet Σ (manchmal auch T), $|\Sigma| < \infty$
2. einem endlichen Vorrat von Nichtterminalzeichen (Variablen) V, $V \cap \Sigma = \emptyset$
3. einem Startsymbol (Axiom) $S \in V$
4. einer endliche Menge P von Produktionen (Ableitungsregeln) der Form $l \to r$ (oder $l ::= r$), mit $l \in (V \cup \Sigma)^*$, $r \in (V \cup \Sigma)^*$

Eine Phrasenstrukturgrammatik (Grammatik) ist ein Quadrupel $G = (V, \Sigma, P, S)$.

Sei $G = (V, \Sigma, P, S)$ eine Phrasenstrukturgrammatik.

Definition 47

Wir schreiben

1. $z \rightarrow_G z'$ gdw
   $(\exists x, y \in (V \cup \Sigma)^*, l \rightarrow r \in P)[z = xly, z' = xry]$
2. $z \rightarrow_G^* z'$ gdw $z = z'$ oder
   $z \rightarrow_G z^{(1)} \rightarrow_G z^{(2)} \rightarrow_G \ldots \rightarrow_G z^{(k)} = z'$. Eine solche Folge von Ableitungsschritten heißt eine Ableitung für z' von z in G (der Länge k).
3. Die von G erzeugte Sprache ist

$$L(G) := \{z \in \Sigma^*;\ S \rightarrow_G^* z\}$$

Zur Vereinfachung der Notation schreiben wir gewöhnlich $\rightarrow$ und $\rightarrow^*$ statt $\rightarrow_G$ und $\rightarrow_G^*$

Vereinbarung:
Wir bezeichnen Nichtterminale mit großen und Terminale mit kleinen Buchstaben!

Beispiel 48

Wir erinnern uns:

- $L_2 = \{ab, abab, ababab, \ldots\} = \{(ab)^n, n \in \mathbb{N}\}$
  $(\Sigma_2 = \{a, b\})$
- Grammatik für L_2 mit folgenden Produktionen:

$$S \to ab, S \to abS$$

Beispiel 48 (Forts.)

- $L_4 = \{a, b, aa, ab, bb, aaa, aab, abb, bbb \ldots\}$
  $= \{a^m b^n, m, n \in \mathbb{N}_0, m + n > 0\} \qquad (\Sigma_4 = \{a, b\})$
- Grammatik für L_4 mit folgenden Produktionen:

$$S \to A,\ S \to B,\ S \to AB,$$
$$A \to a,\ A \to aA,$$
$$B \to b,\ B \to bB$$

3.2 Die Chomsky-Hierarchie

Sei $G = (V, \Sigma, P, S)$ eine Phrasenstrukturgrammatik.

1. Jede Phrasenstrukturgrammatik (Chomsky-Grammatik) ist (zunächst) automatisch vom Typ 0.
2. Eine Chomsky-Grammatik heißt (längen-)monoton, falls für alle Regeln

$$\alpha \to \beta \in P \text{ mit } \alpha \neq S$$

gilt:

$$|\alpha| \leq |\beta| \,,$$

und, falls $S \to \epsilon \in P$, dann das Axiom S auf keiner rechten Seite vorkommt.

3. Eine Chomsky-Grammatik ist vom Typ 1 (auch: kontextsensitiv), falls sie monoton ist und für alle Regeln $\alpha \to \beta$ in P mit $\alpha \neq S$ gilt:

$$\alpha = \alpha' A \alpha'' \text{ und } \beta = \alpha' \beta' \alpha''$$

für geeignete $A \in V$, $\alpha', \alpha'' \in (V \cup \Sigma)^*$ und $\beta' \in (V \cup \Sigma)^+$.

4. Eine Chomsky-Grammatik ist vom Typ 2 (auch: kontextfrei), falls sie monoton ist und für alle Regeln $\alpha \to \beta \in P$ gilt:

$$\alpha \in V \ .$$

Bemerkung: Manchmal wird "kontextfrei" auch ohne die Monotonie-Bedingung definiert; streng monoton schließt dann die Monotonie mit ein, so dass ϵ nicht als rechte Seite vorkommen kann.

5. Eine Chomsky-Grammatik ist vom Typ 3 (auch: regulär, rechtslinear), falls sie monoton ist und für alle Regeln $\alpha \to \beta$ in P mit $\beta \neq \epsilon$ gilt:

$$\alpha \in V \text{ und } \beta \in \Sigma^+ \cup \Sigma^* V \ .$$

Auch hier gilt die entsprechende Bemerkung zur Monotonie-Bedingung.

Beispiel 49

- Die folgende Grammatik ist regulär:

$$S \to \epsilon,\ S \to A,$$
$$A \to aa,\ A \to aaA$$

- Eine Produktion

$$A \to Bcde$$

heißt linkslinear.

- Eine Produktion

$$A \to abcDef$$

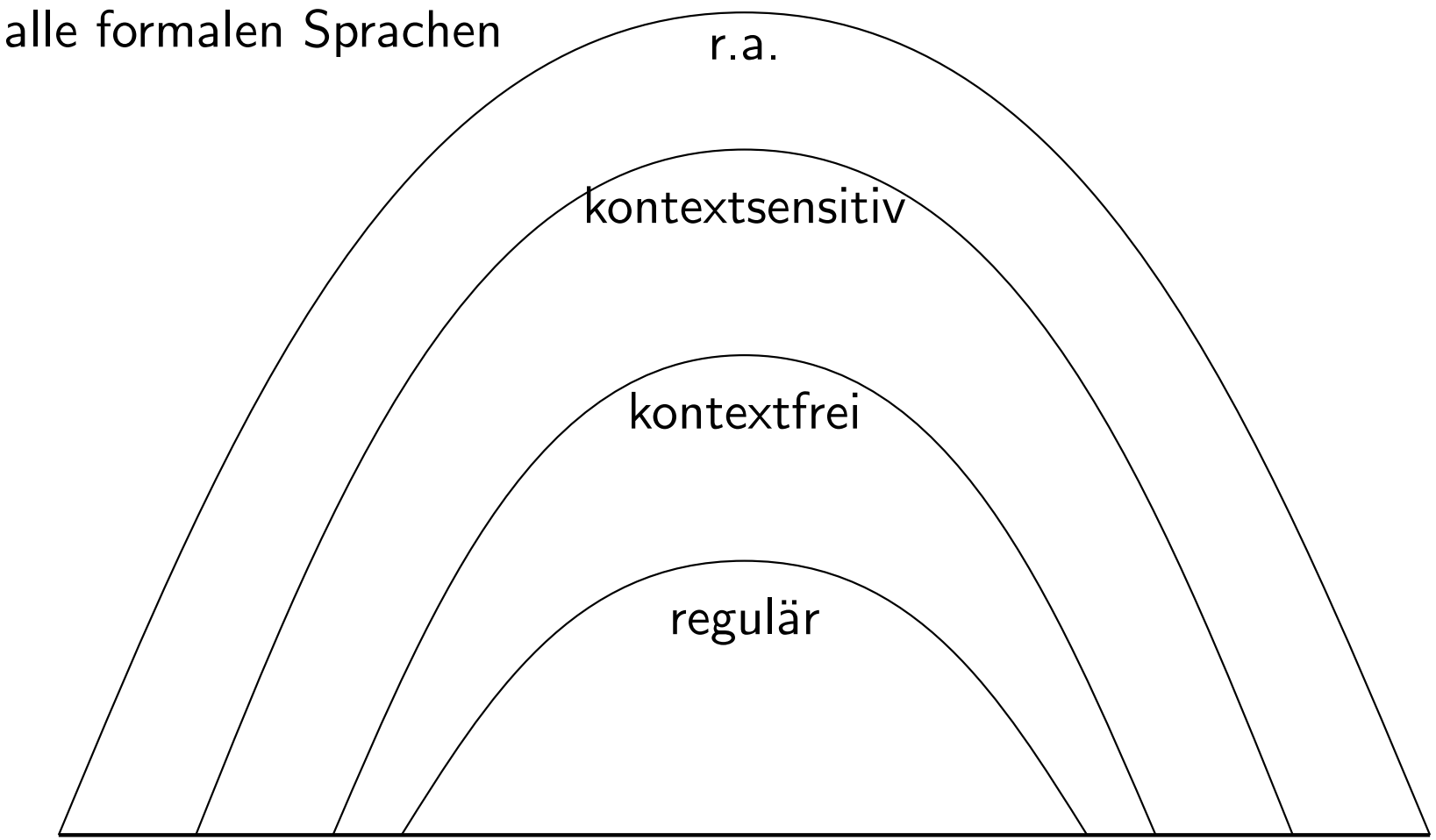
heißt linear.

Definition 50

Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ k, $k \in \{0, 1, 2, 3\}$, falls es eine Chomsky-k-Grammatik G mit $L(G) = L$ gibt.

In der Chomsky-Hierarchie bilden also die Typ-3- oder regulären Sprachen die kleinste, unterste Stufe, darüber kommen die kontextfreien, dann die kontextsensitiven Sprachen. Oberhalb der Typ-1-Sprachen kommen die Typ-0-Sprachen, die auch rekursiv aufzählbar oder semientscheidbar genannt werden. Darüber (und nicht mehr Teil der Chomsky-Hierarchie) findet sich die Klasse aller formalen Sprachen.

In Typ-3-Grammatiken müssen entweder alle Produktionen rechtslinear oder alle linkslinear sein.
Überlegen Sie sich eine lineare Grammatik, deren Sprache nicht regulär ist! (Beweismethode später!)


alle formalen Sprachen
r.a.
kontextsensitiv
kontextfrei
regulär

Lemma 51

Sei $G = (V, \Sigma, P, S)$ eine Chomsky-Grammatik, so dass alle Produktionen $\alpha \to \beta$ die Bedingung $\alpha \in V$ erfüllen. Dann ist $L(G)$ kontextfrei.

Beweis:

Definition 52

Ein $A \in V$ mit $A \to^* \epsilon$ heißt nullierbar.

Bestimme alle nullierbaren $A \in V$:

$N := \{A \in V;\ (A \to \epsilon) \in P\}$
$N' := \emptyset$
while $N \neq N'$ **do**
  $N' := N$
  $N := N' \cup \{A \in V;$
    $(\exists (A \to \beta) \in P)[\beta \in N'^*]\}$
od

Wie man leicht durch Induktion sieht, enthält N zum Schluss genau alle nullierbaren $A \in V$.

Sei nun G eine Grammatik, so dass alle linken Seiten $\in V$, aber die Monotoniebedingung nicht unbedingt erfüllt ist.
Modifiziere G zu G' mit Regelmenge P' wie folgt:

1. für jedes $(A \to x_1 x_2 \cdots x_n) \in P$, $n \geq 1$, füge zu P' alle Regeln $A \to y_1 y_2 \cdots y_n$ hinzu, die dadurch entstehen, dass für nicht-nullierbare x_i $y_i := x_i$ und für nullierbare x_i die beiden Möglichkeiten $y_i := x_i$ und $y_i := \epsilon$ eingesetzt werden, ohne dass die ganze rechte Seite $= \epsilon$ wird.
2. falls S nullierbar ist, sei T ein neues Nichtterminal; füge zu P' die Regeln $S \to \epsilon$ und $S \to T$ hinzu, ersetze S in allen rechten Seiten durch T und ersetze jede Regel $(S \to x) \in P'$, $|x| > 0$, durch $T \to x$.

Lemma 53

$G' = (V \cup T, \Sigma, P', S)$ ist kontextfrei, und es gilt

$$L(G') = L(G) \ .$$

Beweis:

Klar! □

Auch für reguläre Grammatiken gilt ein entsprechender Satz über die "Entfernbarkeit" nullierbarer Nichtterminale:

Lemma 54

Sei $G = (V, \Sigma, P, S)$ eine Chomsky-Grammatik, so dass für alle Regeln $\alpha \to \beta \in P$ gilt:

$$\alpha \in V \text{ und } \beta \in \Sigma^* \cup \Sigma^* V \ .$$

Dann ist $L(G)$ regulär.

Beweis:

Übungsaufgabe! □

Beispiel 55

Typ 3: $L = \{a^n;\ n \in \mathbb{N}\}$, Grammatik:

$$\begin{aligned} S &\to a, \\ S &\to aS \end{aligned}$$

Typ 2: $L = \{a^n b^n;\ n \in \mathbb{N}_0\}$, Grammatik:

$$\begin{aligned} S &\to \epsilon, \\ S &\to T, \\ T &\to ab, \\ T &\to aTb \end{aligned}$$

Wir benötigen beim Scannen *einen* Zähler.

Beispiel 55 (Forts.)

Typ 1: $L = \{a^n b^n c^n;\ n \in \mathbb{N}\}$, Grammatik:

$$\begin{aligned} S &\rightarrow aSXY, \\ S &\rightarrow abY, \\ YX &\rightarrow XY, \\ bX &\rightarrow bb, \\ bY &\rightarrow bc, \\ cY &\rightarrow cc \end{aligned}$$

Wir benötigen beim Scannen *mindestens zwei* Zähler.

Bemerkung: Diese Grammatik entspricht *nicht* unserer Definition des Typs 1, sie ist aber (längen-)monoton. Wir zeigen als Hausaufgabe, dass monotone und Typ 1 Grammatiken die gleiche Sprachklasse erzeugen!

4. Das Wortproblem

4.1 Die Existenz von Ableitungen eines Wortes

Beispiel 56 (Arithmetische Ausdrücke)

$$
\begin{array}{lcl}
\text{<expr>} & \rightarrow & \text{<term>} \\
\text{<expr>} & \rightarrow & \text{<expr>} + \text{<term>} \\
\text{<term>} & \rightarrow & (\text{<expr>}) \\
\text{<term>} & \rightarrow & \text{<term>} \times \text{<term>} \\
\text{<term>} & \rightarrow & a \mid b \mid \ldots \mid z
\end{array}
$$

Aufgabe eines Parsers ist nun, zu prüfen, ob eine gegebene Zeichenreihe einen gültigen arithmetischen Ausdruck darstellt und, falls ja, ihn in seine Bestandteile zu zerlegen.

Sei $G = (V, \Sigma, P, S)$ eine Grammatik.

Definition 57

1. **Wortproblem:** Gegeben ein Wort $w \in \Sigma^*$, stelle fest, ob

$$w \in L(G) \; ?$$

2. **Ableitungsproblem:** Gegeben ein Wort $w \in L(G)$, gib eine Ableitung $S \rightarrow_G^* w$ an, d.h. eine Folge

$$S = w^{(0)} \rightarrow_G w^{(1)} \rightarrow_G \cdots \rightarrow_G w^{(n)} = w$$

   mit $w^{(i)} \in (\Sigma \cup V)^*$ für $i = 1, \ldots, n$.

3. **uniformes Wortproblem:** Wortproblem, bei dem jede Probleminstanz sowohl die Grammatik G wie auch die zu testende Zeichenreihe w enthält. Ist G dagegen global festgelegt, spricht man von einem nicht-uniformen Wortproblem.

Bemerkung:
Das uniforme wie auch das nicht-uniforme Wortproblem ist für Typ-0-Sprachen (also die rekursiv-aufzählbare Sprachen) nicht entscheidbar. Wir werden später sehen, dass es zum Halteproblem für Turingmaschinen äquivalent ist.

Es gilt jedoch

Satz 58
Für kontextsensitive Grammatiken ist das Wortproblem entscheidbar.
Genauer: Es gibt einen Algorithmus, der bei Eingabe einer kontextsensitiven Grammatik $G = (V, \Sigma, P, S)$ und eines Wortes w in endlicher Zeit entscheidet, ob $w \in L(G)$.

Beweisidee:
Angenommen $w \in L(G)$. Dann gibt es eine Ableitung

$$S = w^{(0)} \rightarrow_G w^{(1)} \rightarrow_G \cdots \rightarrow_G w^{(n)} = w$$

mit $w^{(i)} \in (\Sigma \cup V)^*$ für $i = 1, \ldots, n$.
Da aber G kontextsensitiv ist, gilt (falls $w \neq \epsilon$)

$$|w^{(0)}| \leq |w^{(1)}| \leq \cdots \leq |w^{(n)}| \ ,$$

d.h., es genügt, alle Wörter in $(\Sigma \cup V)^*$ der Länge $\leq |w|$ zu erzeugen.

Beweis:

Sei o.B.d.A. $w \neq \epsilon$ und sei

$$T_m^n := \{w' \in (\Sigma \cup V)^*;\ |w'| \leq n \text{ und } w' \text{ lässt sich aus } S \text{ in } \leq m \text{ Schritten ableiten}\}$$

Diese Mengen kann man für alle n und m induktiv wie folgt berechnen:

$$\begin{aligned} T_0^n &:= \{S\} \\ T_{m+1}^n &:= T_m^n \cup \{w' \in (\Sigma \cup V)^*;\ |w'| \leq n \text{ und } w'' \to w' \text{ für ein } w'' \in T_m^n\} \end{aligned}$$

Beachte: Für alle m gilt: $|T_m^n| \leq \sum_{i=1}^{n} |\Sigma \cup V|^i$.
Es muss daher immer ein m_0 geben mit

$$T_{m_0}^n = T_{m_0+1}^n = \ldots =: T_n \ .$$

Beweis (Forts.):

Algorithmus:

$n := |w|$
$T := \{S\}$
$T' := \emptyset$
while $T \neq T'$ **do**
  $T' := T$
  $T := T' \cup \{w' \in (V \cup \Sigma)^+;\ |w'| \leq n, (\exists w'' \in T')[w'' \to w']\}$
od
if $w \in T$ **return** „ja“ **else return** „nein“ **fi**

□

Beispiel 59

Gegeben sei die Typ-2-Grammatik mit den Produktionen

$$S \to ab \text{ und } S \to aSb$$

sowie das Wort $w = abab$.

$$\begin{aligned} T_0^4 &= \{S\} \\ T_1^4 &= \{S, ab, aSb\} \\ T_2^4 &= \{S, ab, aSb, aabb\} \quad aaSbb \text{ ist zu lang!} \\ T_3^4 &= \{S, ab, aSb, aabb\} \end{aligned}$$

Also lässt sich das Wort w mit der gegebenen Grammatik nicht erzeugen!

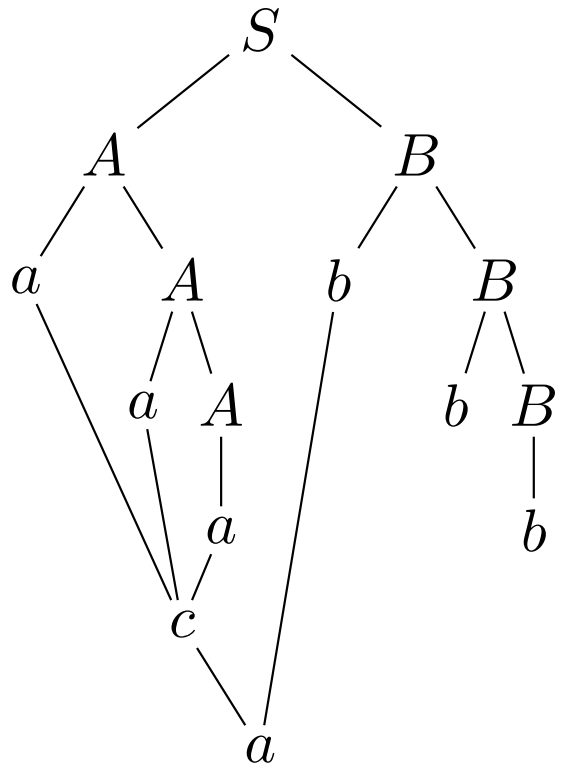
Bemerkung:
Der angegebene Algorithmus ist nicht sehr effizient! Für kontextfreie Grammatiken gibt es wesentlich effizientere Verfahren, die wir später kennenlernen werden!

4.2 Ableitungsgraph und Ableitungsbaum

Grammatik:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA \\
A &\rightarrow a \\
B &\rightarrow bB \\
B &\rightarrow b \\
aaa &\rightarrow c \\
cb &\rightarrow a
\end{aligned}
$$

Beispiel:

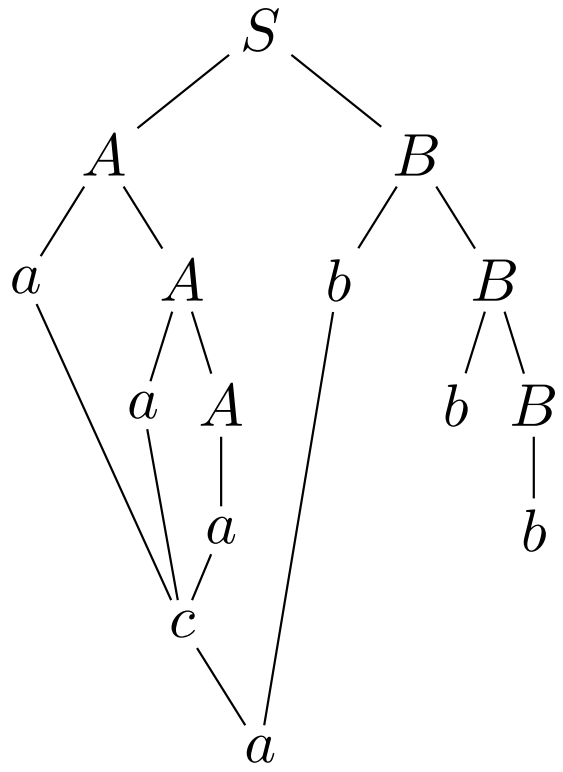


Die Terminale ohne Kante nach unten entsprechen, von links nach rechts gelesen, dem durch den Ableitungsgraphen dargestellten Wort.

Grammatik:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA \\
A &\rightarrow a \\
B &\rightarrow bB \\
B &\rightarrow b \\
aaa &\rightarrow c \\
cb &\rightarrow a
\end{aligned}
$$

Beispiel:



Dem Ableitungsgraph entspricht z.B. die Ableitung

$$
\begin{aligned}
S &\rightarrow AB \rightarrow aAB \rightarrow aAbB \rightarrow aaAbB \rightarrow aaAbbB \rightarrow \\
&\rightarrow aaabbB \rightarrow aaabbb \rightarrow cbbb \rightarrow abb
\end{aligned}
$$

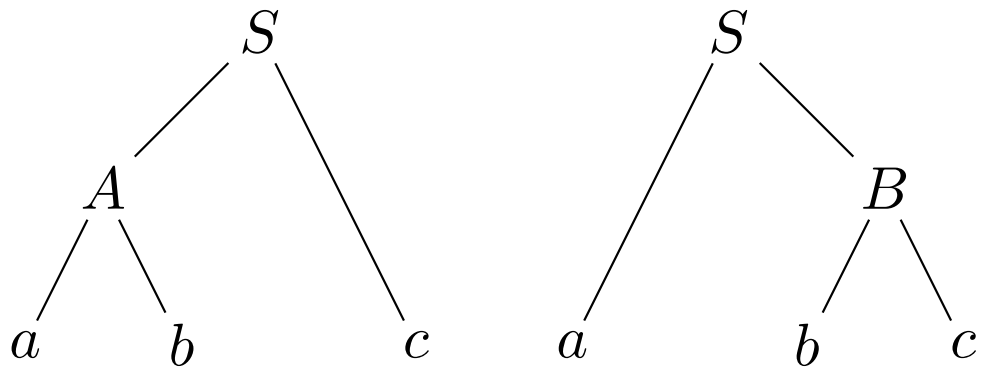
Beobachtung:
Bei kontextfreien Sprachen sind die Ableitungsgraphen immer Bäume.

Beispiel 60

Grammatik:

$$\begin{aligned} S &\to aB \\ S &\to Ac \\ A &\to ab \\ B &\to bc \end{aligned}$$

Ableitungsbäume:



Für das Wort abc gibt es zwei verschiedene Ableitungsbäume.

Definition 61

- Eine Ableitung

$$S = w^{(0)} \rightarrow w^{(1)} \rightarrow \cdots \rightarrow w^{(n)} = w$$

  eines Wortes w heißt Linksableitung, wenn für jede Anwendung einer Produktion $\alpha \rightarrow \beta$ auf $w^{(i)} = x\alpha z$ gilt, dass sich keine Regel der Grammatik auf ein echtes Präfix von $x\alpha$ anwenden lässt.
- Eine Grammatik heißt eindeutig, wenn es für jedes Wort $w \in L(G)$ genau eine Linksableitung gibt. Nicht eindeutige Grammatiken nennt man auch mehrdeutig.
- Eine Sprache L heißt eindeutig, wenn es für L eine eindeutige Grammatik gibt. Ansonsten heißt L mehrdeutig.

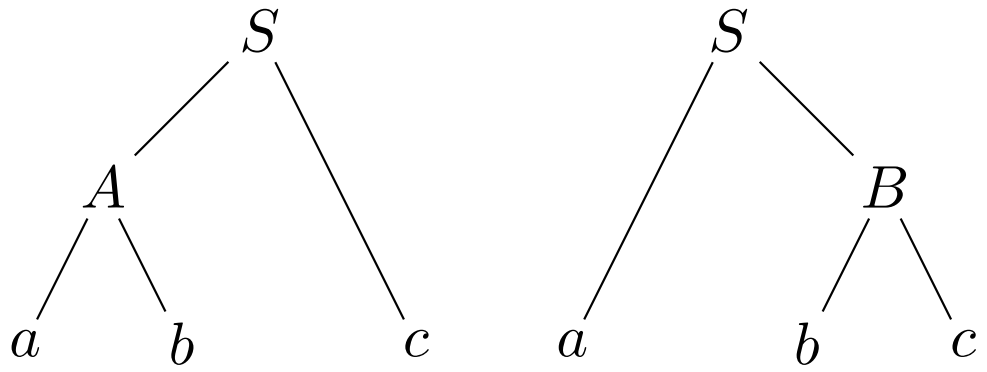
Bemerkung: Eindeutigkeit wird meist für kontextfreie (und reguläre) Grammatiken betrachtet, ist aber allgemeiner definiert.

Beispiel 62

Grammatik:

$$
\begin{aligned}
S &\rightarrow aB \\
S &\rightarrow Ac \\
A &\rightarrow ab \\
B &\rightarrow bc
\end{aligned}
$$

Ableitungsbäume:



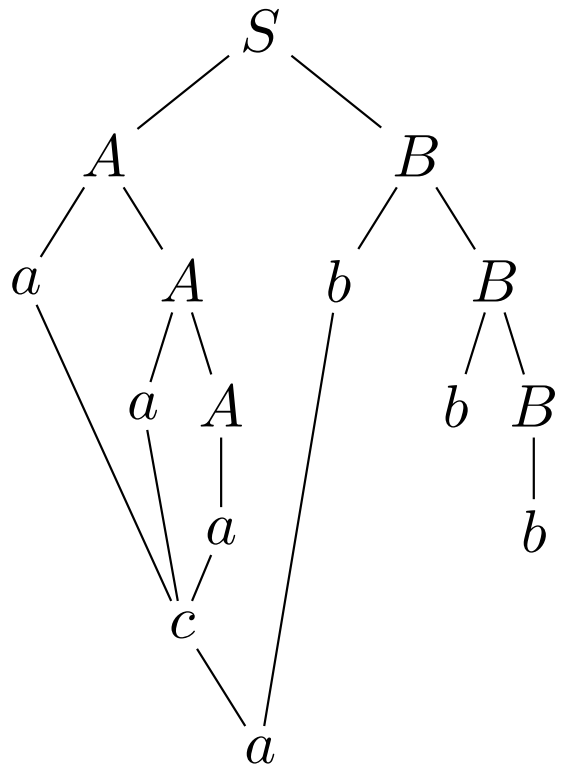
Beide Ableitungsbäume für das Wort abc entsprechen Linksableitungen.

Beispiel 63

Grammatik:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA \\
A &\rightarrow a \\
B &\rightarrow bB \\
B &\rightarrow b \\
aaa &\rightarrow c \\
cb &\rightarrow a
\end{aligned}
$$

Ableitung:



Eine Linksableitung ist

$$
\begin{aligned}
S &\rightarrow AB \rightarrow aAB \rightarrow aaAB \rightarrow aaaB \rightarrow cB \rightarrow \\
&\rightarrow cbB \rightarrow aB \rightarrow abB \rightarrow abb
\end{aligned}
$$

Beispiel 63

Grammatik:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA \\
A &\rightarrow a \\
B &\rightarrow bB \\
B &\rightarrow b \\
aaa &\rightarrow c \\
cb &\rightarrow a
\end{aligned}
$$

Eine andere Linksableitung für abb ist

$$S \rightarrow AB \rightarrow aB \rightarrow abB \rightarrow abb\ .$$

Beispiel 63

Grammatik:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA \\
A &\rightarrow a \\
B &\rightarrow bB \\
B &\rightarrow b \\
aaa &\rightarrow c \\
cb &\rightarrow a
\end{aligned}
$$

Die Grammatik ist also mehrdeutig.

5. Eigenschaften regulärer Automaten und Sprachen

5.1 Äquivalenz von NFA und DFA

Satz 64

Für jede von einem nichtdeterministischen endlichen Automaten akzeptierte Sprache L gibt es auch einen deterministischen endlichen Automaten M mit

$$L = L(M) \; .$$

Beweis:

Sei $N = (Q, \Sigma, \delta, q_0, F)$ ein NFA.

Definiere

1. $M' := (Q', \Sigma, \delta', q_0', F')$
2. $Q' := \mathcal{P}(Q)$ $\quad$ ($\mathcal{P}(Q) = 2^Q$ Potenzmenge von Q)
3. $\delta'(Q'', a) := \bigcup_{q' \in Q''} \delta(q', a)$ für alle $Q'' \in Q'$, $a \in \Sigma$
4. $q_0' := \{q_0\}$
5. $F' := \{Q'' \subseteq Q;\ Q'' \cap F \neq \emptyset\}$

Also

NFA N:	Q	Σ	δ	q_0	F
DFA M':	2^Q	Σ	δ'	q_0'	F'

Beweis (Forts.):

Es gilt:

$$\begin{aligned} w \in L(N) &\Leftrightarrow \hat{\delta}(S,w) \cap F \neq \emptyset \\ &\Leftrightarrow \widehat{\delta'}(q_0', w) \in F' \\ &\Leftrightarrow w \in L(M'). \end{aligned}$$

□

Der zugehörige Algorithmus zur Überführung eines NFA in einen DFA heißt Teilmengenkonstruktion, Potenzmengenkonstruktion oder Myhill-Konstruktion.

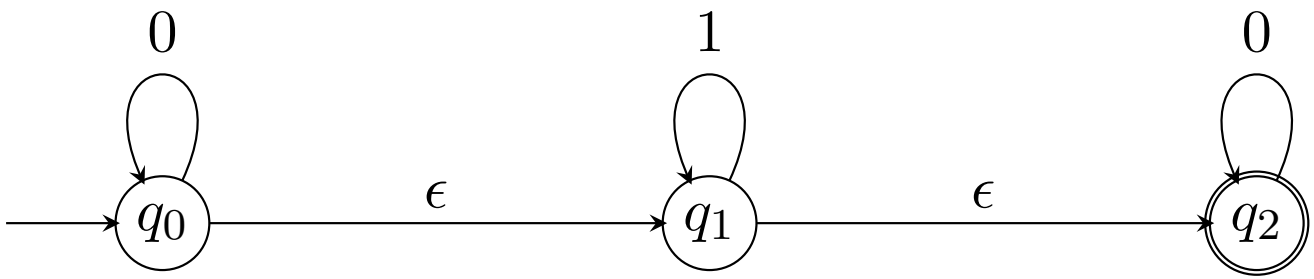
5.2 NFA's mit ϵ-Übergängen

Definition 65

Ein (nichtdeterministischer) endlicher Automat A mit ϵ-Übergängen ist ein 5-Tupel analog zur Definition des NFA mit

$$\delta : Q \times (\Sigma \uplus \{\epsilon\}) \to \mathcal{P}(Q) \ .$$

Ein ϵ-Übergang wird ausgeführt, ohne dass ein Eingabezeichen gelesen wird. Wir setzen o.B.d.A. voraus, dass A nur einen Anfangszustand hat.



Definiere für alle $a \in \Sigma$

$$\bar{\delta}(q, a) := \hat{\delta}(q, \epsilon^* a \epsilon^*) \ .$$

Falls A das leere Wort ϵ mittels ϵ-Übergängen akzeptiert, also $F \cap \hat{\delta}(q_0, \epsilon^*) \neq \emptyset$, dann setze zusätzlich

$$F := F \cup \{q_0\} \ .$$

Satz 66

$$w \in L(A) \Leftrightarrow \hat{\bar{\delta}}(S, w) \cap \ F \neq \emptyset \ .$$

Beweis:

Hausaufgabe! □

5.3 Entfernen von ϵ-Übergängen

Satz 67

Zu jedem nichtdeterministischen endlichen Automaten A mit ϵ-Übergängen gibt es einen nichtdeterministischen endlichen Automaten A' ohne ϵ-Übergänge, so dass gilt:

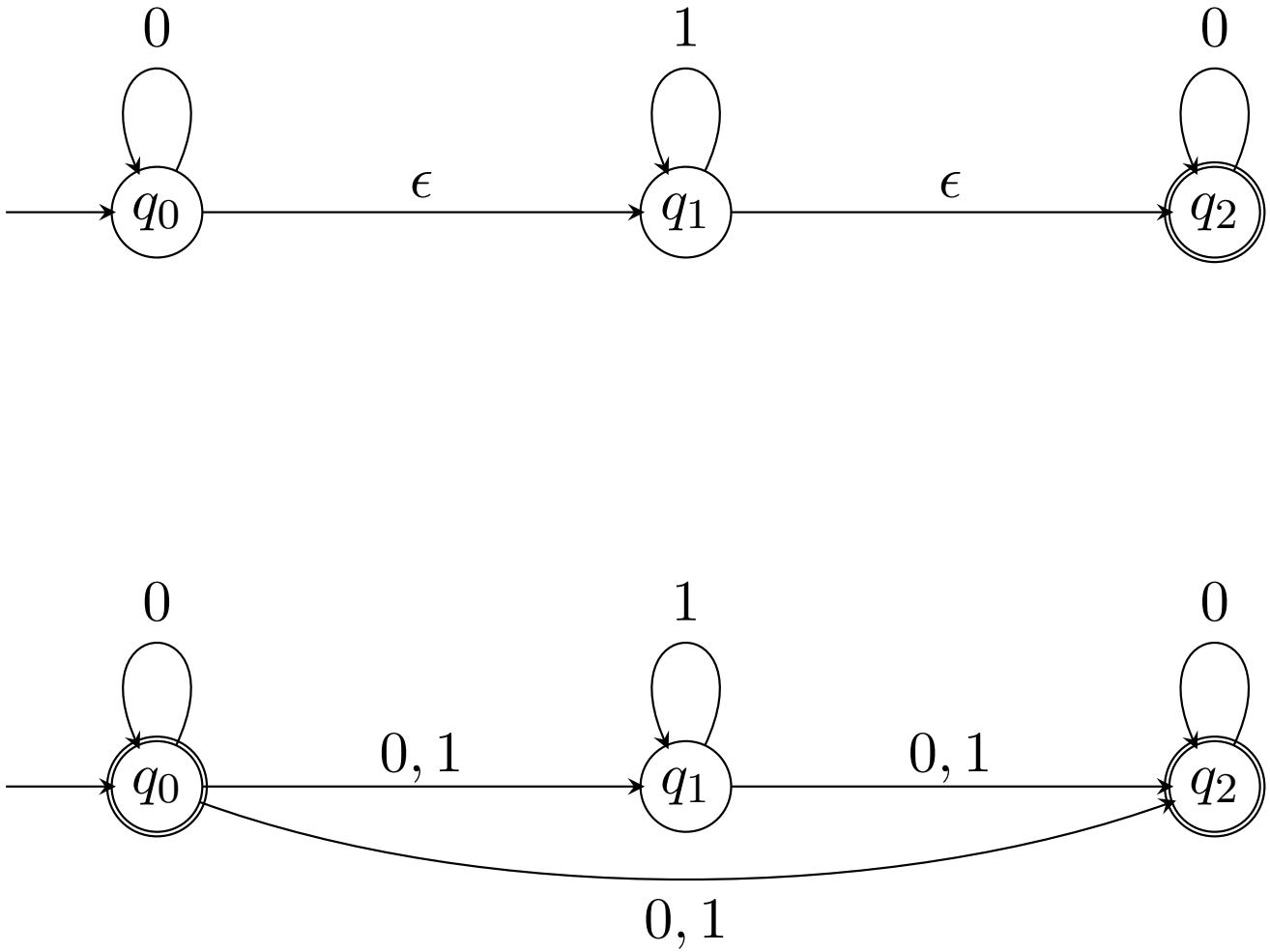
$$L(A) = L(A')$$

Beweis:

Ersetze δ durch $\bar{\delta}$ und F durch F' mit

$$F' = \begin{cases} F & \epsilon \notin L(A) \\ F \cup \{q_0\} & \epsilon \in L(A) \end{cases}$$

Beispiel 68



Satz 69

Ist $G = (V, \Sigma, P, S)$ eine rechtslineare (also reguläre) Grammatik (o.B.d.A. sind die rechten Seiten aller Produktionen aus $\Sigma \cup \{\epsilon\} \cup (\Sigma \cup \{\epsilon\})V$), so ist $N = (V \cup \{X\}, \Sigma, \delta, \{S\}, F)$, mit

$$F := \begin{cases} \{S, X\}, & \text{falls } S \to \epsilon \in P \\ \{X\}, & \text{sonst} \end{cases}$$

und, für alle $A, B \in V$, $a \in \Sigma \cup \{\epsilon\}$,

$$\begin{aligned} B \in \delta(A, a) \quad &\Longleftrightarrow \quad A \to aB \qquad \text{und} \\ X \in \delta(A, a) \quad &\Longleftrightarrow \quad A \to a \end{aligned}$$

ein nichtdeterministischer endlicher Automat, der genau $L(G)$ akzeptiert.

Beweis:

Aus der Konstruktion folgt, dass N ein NFA ist (i.A. mit ϵ-Übergängen, die sich bei Produktionen der Form $A \to B$ ergeben).
Durch eine einfache Induktion über n zeigt man, dass eine Satzform

$$a_1 a_2 \cdots a_{n-1} A \text{ bzw. } a_1 a_2 \cdots a_n$$

in G genau dann ableitbar ist, wenn für die erweiterte Übergangsfunktion $\hat{\delta}$ des zu N äquivalenten NFA *ohne* ϵ-Übergänge gilt:

$$A \in \hat{\delta}(S, a_1 a_2 \cdots a_{n-1})$$

bzw.

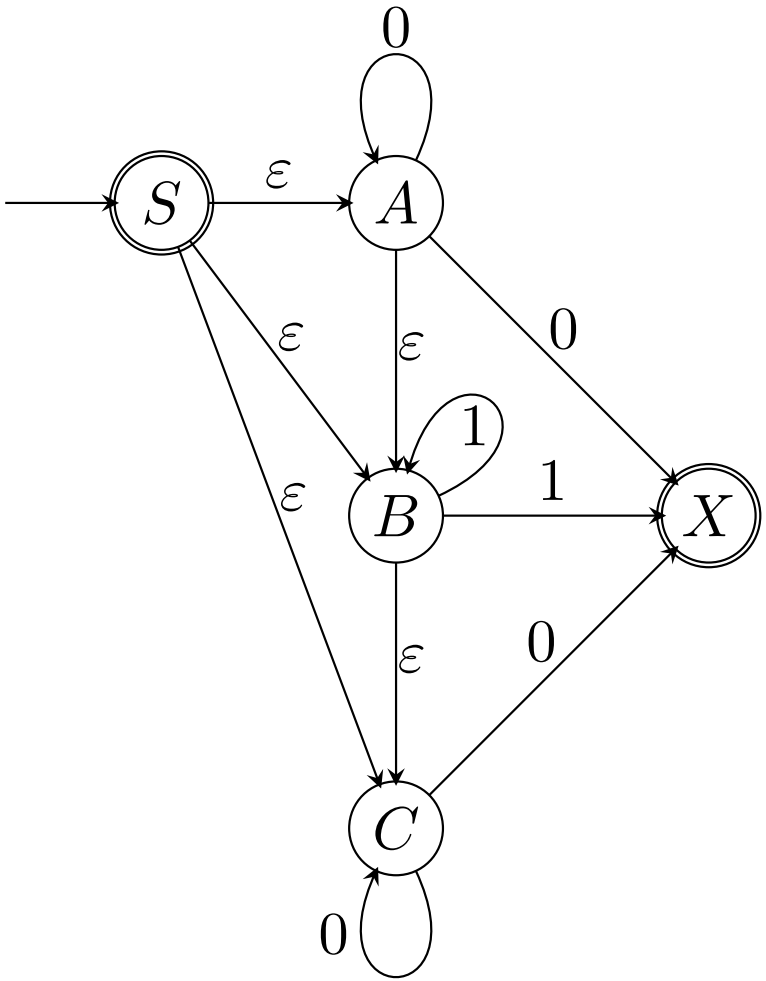
$$X \in \hat{\delta}(S, a_1 a_2 \cdots a_n)$$

(bzw., für $n = 0$, $F \cap \hat{\delta}(S, \epsilon) \neq \emptyset$). □

Beispiel 70

Wir betrachten die (reguläre) Grammatik G mit den Produktionen

$$
\begin{aligned}
S &\rightarrow \varepsilon \\
S &\rightarrow A \mid B \mid C \\
A &\rightarrow 0A \mid 0 \mid B \\
B &\rightarrow 1B \mid 1 \mid C \\
C &\rightarrow 0C \mid 0
\end{aligned}
$$



Zusammenfassend ergibt sich:

Satz 71

Die Klasse der regulären Sprachen (Chomsky-3-Sprachen) ist identisch mit der Klasse der Sprachen, die

- *von DFA's akzeptiert/erkannt werden,*
- *von NFA's akzeptiert werden,*
- *von NFA's mit ϵ-Übergängen akzeptiert werden.*

Beweis:

Wie soeben gezeigt. □

5.5 Reguläre Ausdrücke

Reguläre Ausdrücke sollen eine kompakte Notation für spezielle Sprachen sein, wobei endliche Ausdrücke hier auch unendliche Mengen beschreiben können.

Definition 72

Reguläre Ausdrücke sind induktiv definiert durch:

1. $\emptyset$ ist ein regulärer Ausdruck.
2. ϵ ist ein regulärer Ausdruck.
3. Für jedes $a \in \Sigma$ ist a ist ein regulärer Ausdruck.
4. Wenn α und β reguläre Ausdrücke sind, dann sind auch (α), $\alpha\beta$, $(\alpha|\beta)$ (hierfür wird oft auch $(\alpha + \beta)$ geschrieben) und $(\alpha)^*$ reguläre Ausdrücke.
5. Nichts sonst ist ein regulärer Ausdruck.

Bemerkung: Ist α atomar, so schreiben wir statt $(\alpha)^*$ oft auch nur α^*.

Zu einem regulären Ausdruck γ ist die zugehörige Sprache $L(\gamma)$ induktiv definiert durch:

Definition 73

1. Falls $\gamma = \emptyset$, so gilt $L(\gamma) = \emptyset$.
2. Falls $\gamma = \epsilon$, so gilt $L(\gamma) = \{\epsilon\}$.
3. Falls $\gamma = a$, so gilt $L(\gamma) = \{a\}$.
4. Falls $\gamma = (\alpha)$, so gilt $L(\gamma) = L(\alpha)$.
5. Falls $\gamma = \alpha\beta$, so gilt
$$L(\gamma) = L(\alpha)L(\beta) = \{uv;\ u \in L(\alpha), v \in L(\beta)\}\ .$$
6. Falls $\gamma = (\alpha \mid \beta)$, so gilt
$$L(\gamma) = L(\alpha) \cup L(\beta) = \{u;\ u \in L(\alpha) \vee u \in L(\beta)\}\ .$$
7. Falls $\gamma = (\alpha)^*$, so gilt
$$L(\gamma) = (L(\alpha))^* = \{u_1 u_2 \dots u_n;\ n \in \mathbb{N}_0, u_1, \dots, u_n \in L(\alpha)\}\ .$$

Beispiel 74

Sei das zugrunde liegende Alphabet $\Sigma = \{0, 1\}$.

- alle Wörter, die gleich 0 sind oder mit 00 enden:

$$(0 \mid (0 \mid 1)^*00)$$

- alle Wörter, die 0110 enthalten:

$$(0|1)^*0110(0|1)^*$$

- alle Wörter, die eine gerade Anzahl von 1'en enthalten:

$$(0^*10^*1)^*0^*$$

- alle Wörter, die die Binärdarstellung einer durch 3 teilbaren Zahl darstellen, also

$$0, 11, 110, 1001, 1100, 1111, 10010, \ldots$$

  Hausaufgabe!

Satz 75

Eine Sprache $L \subseteq \Sigma^*$ ist genau dann durch einen regulären Ausdruck darstellbar, wenn sie regulär ist.

Beweis:

"$\Longrightarrow$":

Sei also $L = L(\gamma)$.

Wir zeigen: $\exists$ NFA N mit $L = L(N)$ mit Hilfe struktureller Induktion.

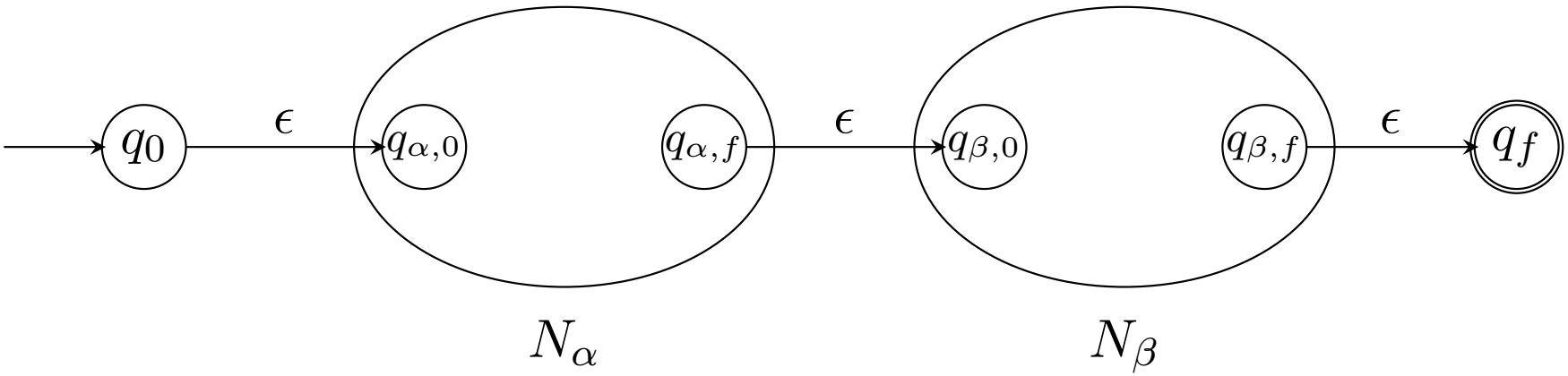
Induktionsanfang: Falls $\gamma = \emptyset$, $\gamma = \epsilon$, oder $\gamma = a \in \Sigma$, so folgt die Behauptung unmittelbar.

Induktionsschritt:

$\gamma = \alpha\beta$:
nach Induktionsannahme $\exists$ NFA N_α und N_β mit

$$L(N_\alpha) = L(\alpha) \text{ und } L(N_\beta) = L(\beta) \ .$$

N_γ:

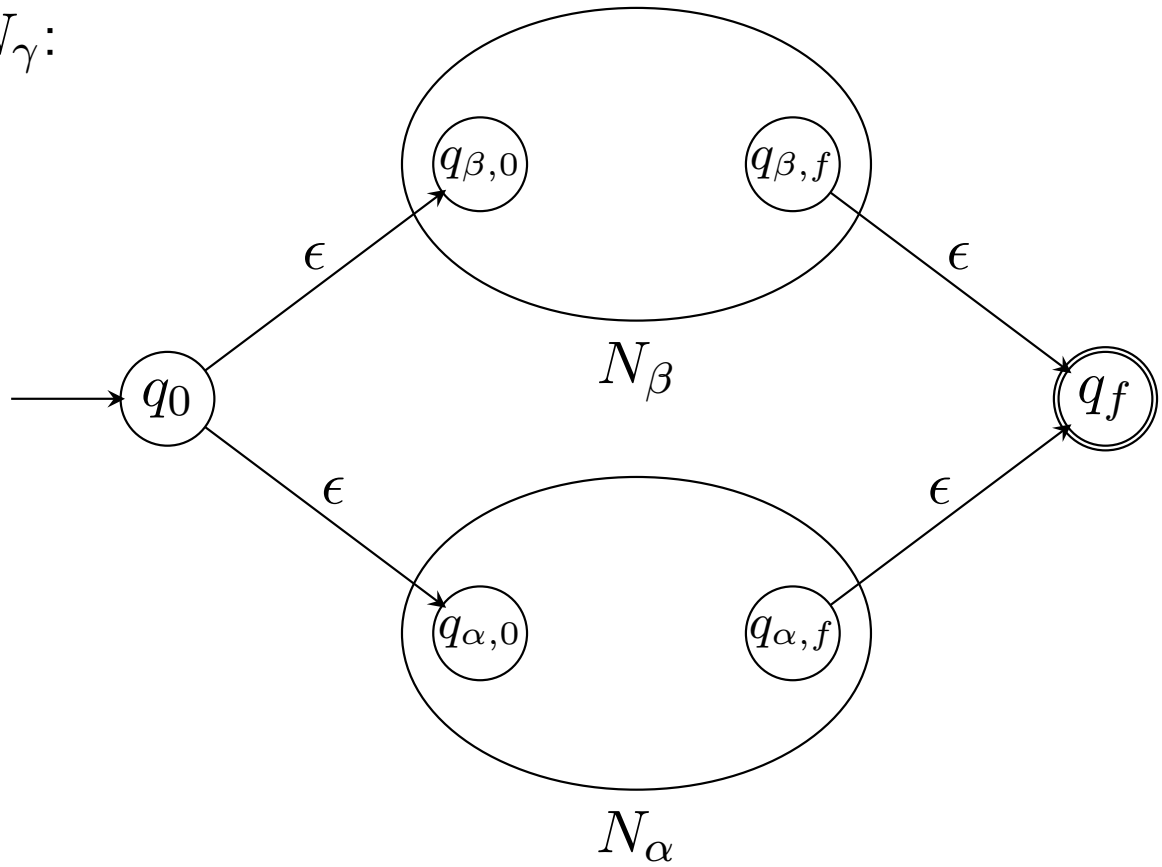


Induktionsschritt (Forts.):

$\gamma = (\alpha \mid \beta)$:
nach Induktionsannahme $\exists$ NFA N_α und N_β mit

$$L(N_\alpha) = L(\alpha) \text{ und } L(N_\beta) = L(\beta) \ .$$

N_γ:

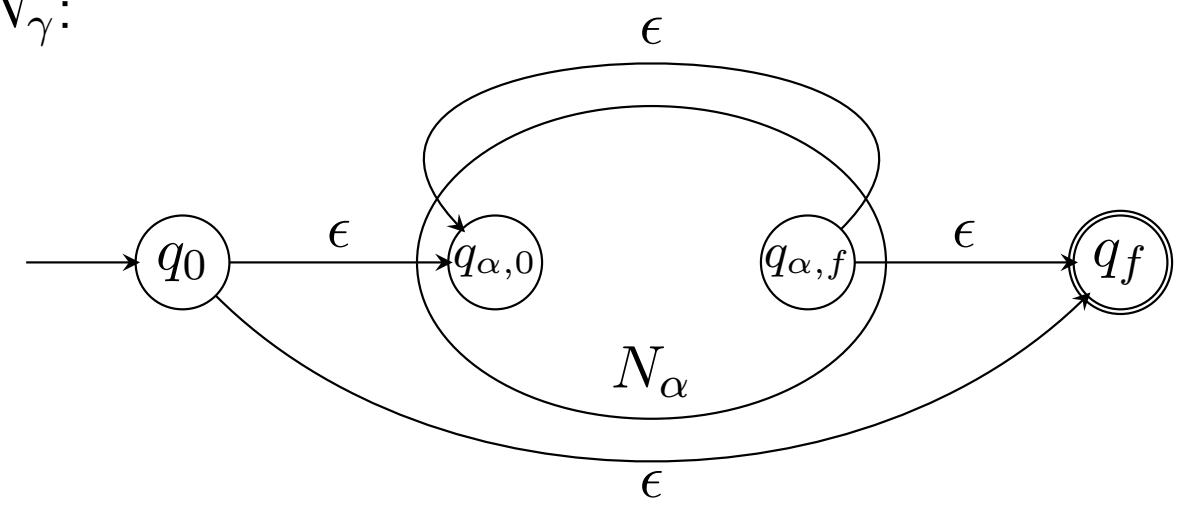


Induktionsschritt (Forts.):

$\gamma = (\alpha)^*$:
nach Induktionsannahme $\exists$ NFA N_α mit

$$L(N_\alpha) = L(\alpha) \; .$$

N_γ:



"$\Longleftarrow$":
Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat.
Wir zeigen: es gibt einen regulären Ausdruck γ mit $L(M) = L(\gamma)$.

Sei $Q = \{q_0, \ldots, q_n\}$. Wir setzen

$R_{ij}^k := \{w \in \Sigma^*;$ die Eingabe w überführt den im Zustand q_i gestarteten Automaten in den Zustand q_j, wobei alle zwischendurch durchlaufenen Zustände einen Index kleiner gleich k haben$\}$

Behauptung: Für alle $i, j \in \{0, \ldots, n\}$ und alle $k \in \{-1, 0, 1, \ldots, n\}$ gilt:
Es gibt einen regulären Ausdruck α_{ij}^k mit $L(\alpha_{ij}^k) = R_{ij}^k$.

Bew.:

Induktion über k:
$k = -1$: Hier gilt

$$R_{ij}^{-1} := \begin{cases} \{a \in \Sigma;\ \delta(q_i, a) = q_j\}, & \text{falls } i \neq j \\ \{a \in \Sigma;\ \delta(q_i, a) = q_j\} \cup \{\epsilon\}, & \text{falls } i = j \end{cases}$$

R_{ij}^{-1} ist also endlich und lässt sich daher durch einen regulären Ausdruck α_{ij}^{-1} beschreiben.

Bew.:

Induktion über k:
$k \Rightarrow k+1$: Hier gilt

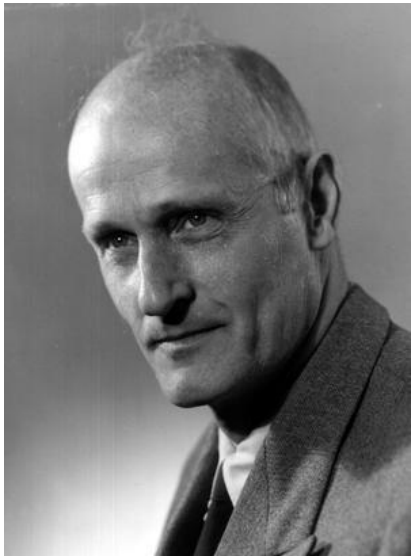
$$R_{ij}^{k+1} = R_{ij}^{k} \cup R_{i\,k+1}^{k}(R_{k+1\,k+1}^{k})^{*}R_{k+1\,j}^{k}$$
$$\alpha_{ij}^{k+1} = (\alpha_{ij}^{k} \mid \alpha_{i\,k+1}^{k}(\alpha_{k+1\,k+1}^{k})^{*}\alpha_{k+1\,j}^{k})$$

Somit gilt: $L(M) = L((\alpha_{0\,f_1}^{n} \mid \alpha_{0\,f_2}^{n} \mid \cdots \mid \alpha_{0\,f_r}^{n}))$, wobei $f_1, \ldots, f_r$ die Indizes der Endzustände seien.

□(Satz 75)

Stephen Cole Kleene

- Geboren am 5. Januar 1909 in Hartford, Connecticut
- Gestorben am 25. Januar 1994 in Madison, Wisconsin
- US-amerikanischer Mathematiker und Logiker
- Mitbegründer der theoretischen Informatik
  - Automatentheorie
  - Kleenesche Hülle A*
  - Arbeiten zum Lambda-Kalkül von A. Church
- 1934 Promotion in Mathematik an der Princeton University
  - Doktorvater: Alonzo Church (1903-1995)



5.6 Abschlusseigenschaften regulärer Sprachen

Satz 76

Seien $R_1, R_2 \subseteq \Sigma^*$ reguläre Sprachen. Dann sind auch

$$R_1 R_2, R_1 \cup R_2, R_1^*, \Sigma^* \setminus R_1 (=: \bar{R}_1), R_1 \cap R_2$$

reguläre Sprachen.

Beweis:

$R_1 R_2, R_1 \cup R_2, R_1^*$ klar.

$\Sigma^* \setminus R_1$: Sei $R_1 = L(A), A$ DFA, $A = (Q, \Sigma, \delta, q_0, F)$, δ vollständig.
Betrachte $A' = (Q, \Sigma, \delta, q_0, Q \setminus F)$.
Dann ist $L(A') = \Sigma^* \setminus L(A)$

$R_1 \cap R_2$: De Morgan

□

Definition 77

Substitution (mit regulären Mengen) ist eine Abbildung, die jedem $a \in \Sigma$ eine reguläre Sprache $h(a)$ zuordnet. Diese Abbildung wird kanonisch auf Σ^* erweitert.
Ein Homomorphismus ist eine Substitution mit
$\forall a \in \Sigma : |h(a)| = 1$, d.h. jedes $a \in \Sigma$ wird durch ein Wort $h(a)$ $(\in \Delta^*)$ ersetzt.

Satz 78

Reguläre Sprachen sind unter (regulärer) Substitution, Homomorphismus und inversem Homomorphismus abgeschlossen.

Beweis:

Wir zeigen (nur) die Behauptung für den inversen Homomorphismus.

Sei $h : \Delta \to \Sigma^*$ ein Homomorphismus, und sei $R \subseteq \Sigma^*$ regulär.

Zu zeigen: $h^{-1}(R) \subseteq \Delta^*$ ist regulär.

Sei $A = (Q, \Sigma, \delta, q_0, F), L(A) = R$.

Betrachte $A' = (Q, \Delta, \delta', q_0, F)$, mit

$$\delta'(q, a) = \delta(q, h(a)) \quad \forall q \in Q, a \in \Delta \ ,$$

wobei wir nunmehr der Einfachheit halber statt $\hat{\delta}$ nur δ schreiben.

Also gilt

$$\delta'(q_0, w) = \delta(q_0, h(w)) \in F \Leftrightarrow h(w) \in R \Leftrightarrow w \in h^{-1}(R)$$

□

Definition 79

Seien $L_1, L_2 \subseteq \Sigma^*$. Dann ist der Rechtsquotient

$$L_1/L_2 := \{x \in \Sigma^*;\ (\exists y \in L_2)[xy \in L_1]\}\ .$$

Satz 80

Seien $R, L \subseteq \Sigma^*$, R regulär. Dann ist R/L regulär.

Beweis:

Sei A DFA mit $L(A) = R$, $A = (Q, \Sigma, \delta, q_0, F)$.

$$\begin{aligned} F' &:= \{q \in Q;\ (\exists y \in L)[\hat{\delta}(q, y) \in F]\} \\ A' &:= (Q, \Sigma, \delta, q_0, F') \end{aligned}$$

Dann ist $L(A') = R/L$. □

Lemma 81

Es gibt einen Algorithmus, der für zwei (nichtdeterministische, mit ϵ-Übergängen) endliche Automaten A_1 und A_2 entscheidet, ob sie äquivalent sind, d.h. ob

$$L(A_1) = L(A_2) \ .$$

Beweis:

Konstruiere einen endlichen Automaten für $(L(A_1) \setminus L(A_2)) \cup (L(A_2) \setminus L(A_1))$ (symmetrische Differenz).
Prüfe, ob dieser Automat ein Wort akzeptiert. □

Satz 82 (Pumping Lemma für reguläre Sprachen)

Sei $R \subseteq \Sigma^*$ *regulär. Dann gibt es ein* $n > 0$, *so dass für jedes* $z \in R$ *mit* $|z| \geq n$ *es* $u, v, w \in \Sigma^*$ *gibt, so dass gilt:*

1. $z = uvw$,
2. $|uv| \leq n$,
3. $|v| \geq 1$, *und*
4. $\forall i \geq 0 : \; uv^i w \in R$.

Beweis:

Sei $R = L(A), A = (Q, \Sigma, \delta, q_0, F)$.

Sei $n = |Q|$. Sei nun $z \in R$ mit $|z| \geq n$.

Sei $q_0 = q^{(0)}, q^{(1)}, q^{(2)}, \ldots, q^{(|z|)}$ die beim Lesen von z durchlaufene Folge von Zuständen von A. Dann muss es $0 \leq i < j \leq n \leq |z|$ geben mit $q^{(i)} = q^{(j)}$.

Seien nun u die ersten i Zeichen von z, v die nächsten $j - i$ Zeichen und w der Rest.

$$\Rightarrow z = uvw, |v| \geq 1, |uv| \leq n, uv^l w \in R \quad \forall l \geq 0 \; .$$

□

Beispiel für die Anwendung des Pumping Lemmas:

Satz 83
$L = \{0^{m^2};\ m \geq 0\}$ *ist nicht regulär.*

Beweis:
Angenommen, L sei doch regulär.
Sei n wie durch das Pumping Lemma gegeben. Wähle $m \geq n$.
Dann gibt es ein r mit $1 \leq r \leq n$, so dass gilt:

$$0^{m^2+ir} \in L \text{ für alle } i \in \mathbb{N}_0 \ .$$

Aber (für $i = 1$):

$$m^2 < m^2 + r \leq m^2 + m < m^2 + 2m + 1 = (m+1)^2$$

und damit Widerspruch! □

Denkaufgabe:

$\{a^i b^i;\ i \geq 0\}$ ist nicht regulär.

Definition 84

Sei $L \subseteq \Sigma^*$ eine Sprache. Definiere die Relation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ durch

$$x \equiv_L y \Leftrightarrow (\forall z \in \Sigma^*)[xz \in L \Leftrightarrow yz \in L]$$

Lemma 85

$\equiv_L$ ist eine rechtsinvariante Äquivalenzrelation.

Dabei bedeutet rechtsinvariant:

$$x \equiv_L y \Rightarrow xu \equiv_L yu \text{ für alle } u \text{ .}$$

Beweis:

Klar! □

Satz 86 (Myhill-Nerode)

Sei $L \subseteq \Sigma^*$. Dann sind äquivalent:

1. *L ist regulär*
2. *$\equiv_L$ hat endlichen Index (= Anzahl der Äquivalenzklassen)*
3. *L ist die Vereinigung einiger der endlich vielen Äquivalenzklassen von $\equiv_L$.*

Beweis:

(1)⇒(2):

Sei $L = L(A)$ für einen DFA $A = (Q, \Sigma, \delta, q_0, F)$.

Dann gilt

$$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y) \quad \Rightarrow \quad x \equiv_L y \ .$$

Also gibt es höchstens so viele Äquivalenzklassen, wie der Automat A Zustände hat.

Beweis:

(2)⇒(3):

Sei $[x]$ die Äquivalenzklasse von x, $y \in [x]$ und $x \in L$.

Dann gilt nach der Definition von $\equiv_L$:

$$y \in L$$

Beweis:

(3)⇒(1):

Definiere $A' = (Q', \Sigma, \delta', q_0', F')$ mit

$$\begin{aligned} Q' &:= \{[x];\ x \in \Sigma^*\} \qquad (Q' \text{ endlich!}) \\ q_0' &:= [\epsilon] \\ \delta'([x], a) &:= [xa] \quad \forall x \in \Sigma^*, a \in \Sigma \qquad (\text{konsistent!}) \\ F' &:= \{[x];\ x \in L\} \end{aligned}$$

Dann gilt:

$$L(A') = L$$

□

5.7 Konstruktion minimaler endlicher Automaten

Satz 87

Der nach dem Satz von Myhill-Nerode konstruierte deterministische endliche Automat hat unter allen DFA's für L eine minimale Anzahl von Zuständen.

Beweis:

Sei $A = (Q, \Sigma, \delta, q_0, F)$ mit $L(A) = L$. Dann liefert

$$x \equiv_A y :\Leftrightarrow \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$

eine Äquivalenzrelation, die $\equiv_L$ verfeinert.
Also gilt: $|Q| = \text{index}(\equiv_A) \geq \text{index}(\equiv_L) =$ Anzahl der Zustände des Myhill-Nerode-Automaten. □

Algorithmus zur Konstruktion eines minimalen FA

Eingabe: $A(Q, \Sigma, \delta, q_0, F)$ DFA $\quad (L = L(A))$

Ausgabe: Äquivalenzrelation auf Q.

0. Entferne aus Q alle überflüssigen, d.h. alle von q_0 aus nicht erreichbaren Zustände. Wir nehmen nun an, dass Q keine überflüssigen Zustände mehr enthält.
1. Markiere alle Paare $\{q_i, q_j\} \in Q^2$ mit

$$q_i \in F \text{ und } q_j \notin F \text{ bzw. } q_i \notin F \text{ und } q_j \in F \ .$$

2. **for** alle unmarkierten Paare $\{q_i, q_j\} \in Q^2$, $q_i \neq q_j$ **do**
   if $(\exists a \in \Sigma)[\{\delta(q_i, a), \delta(q_j, a)\}$ ist markiert$]$ **then**
   markiere $\{q_i, q_j\}$;
   for alle $\{q, q'\}$ in $\{q_i, q_j\}$'s Liste **do**
   markiere $\{q, q'\}$ und lösche aus Liste;
   ebenso rekursiv alle Paare in der Liste von $\{q, q'\}$ usw.
   od
   else
   for alle $a \in \Sigma$ **do**
   if $\delta(q_i, a) \neq \delta(q_j, a)$ **then**
   trage $\{q_i, q_j\}$ in die Liste von $\{\delta(q_i, a), \delta(q_j, a)\}$ ein
   fi
   od
   fi
   od
3. Ausgabe: q äquivalent zu $q' \Leftrightarrow \{q, q'\}$ *nicht* markiert.

Satz 88

Obiger Algorithmus liefert einen minimalen DFA für $L(A)$.

Beweis:

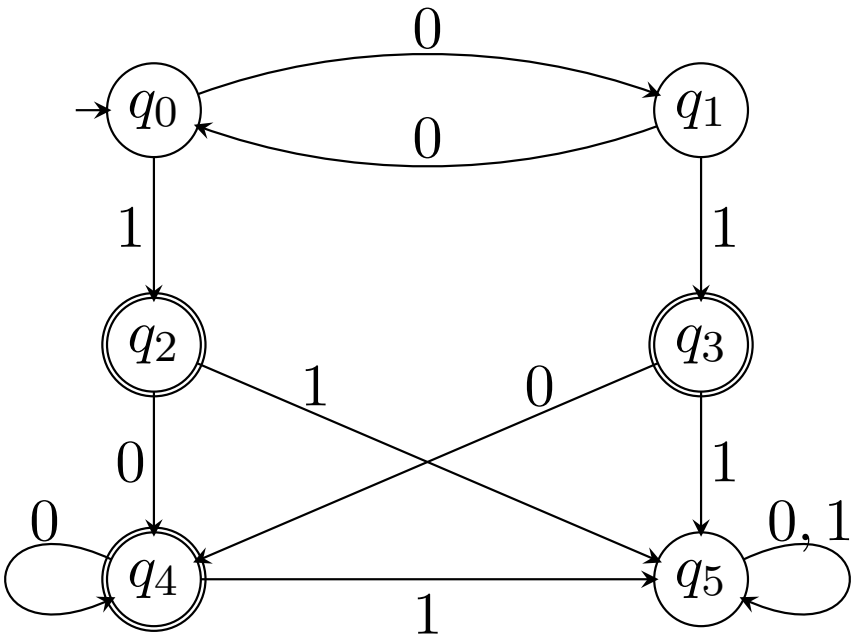
Sei $A' = (Q', \Sigma', \delta', q_0', F')$ der konstruierte Äquivalenzklassenautomat.
Offensichtlich ist $L(A) = L(A')$.
Es gilt: $\{q, q'\}$ wird markiert gdw

$$(\exists\, w \in \Sigma^*)[\delta(q, w) \in F \wedge \delta(q', w) \notin F \text{ oder umgekehrt}],$$

wie man durch einfache Induktion über $|w|$ sieht.
Also: Die Anzahl der Zustände von A' (nämlich $|Q'|$) ist gleich dem Index von $\equiv_L$. □

Beispiel 89

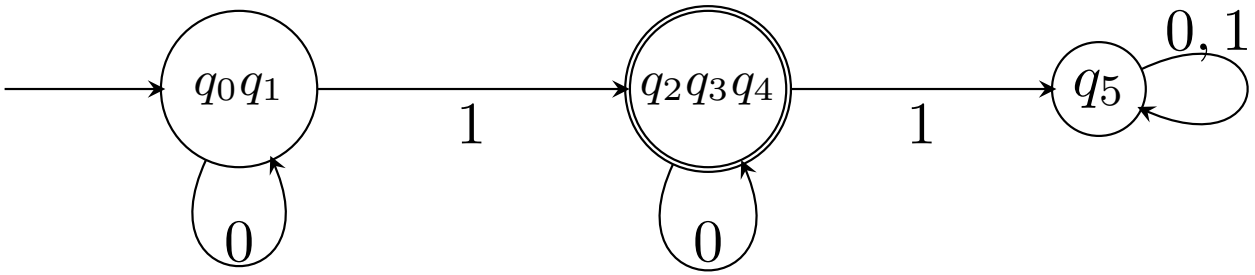
Automat A:



	q_0	q_1	q_2	q_3	q_4	q_5
q_0	/	/	/	/	/	/
q_1		/	/	/	/	/
q_2	×	×	/	/	/	/
q_3	×	×		/	/	/
q_4	×	×			/	/
q_5	×	×	×	×	×	/

Automat A':

$L(A') = 0^*10^*$



Satz 90

Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DFA. Der Zeitaufwand des obigen Minimalisierungsalgorithmus ist $O(|Q|^2|\Sigma|)$.

Beweis:

Für jedes $a \in \Sigma$ muss jede Position in der Tabelle nur konstant oft besucht werden. □

6. Einige Ergebnisse zur Entscheidbarkeit

Beispiel 91

Wie wir bereits wissen, ist das Wortproblem für reguläre Grammatiken entscheidbar. Wenn L durch einen deterministischen endlichen Automaten gegeben ist, ist dies sogar in linearer Laufzeit möglich. Allerdings gilt, dass bei der Überführung eines nichtdeterministischen endlichen Automaten in einen deterministischen endlichen Automaten die Komplexität exponentiell zunehmen kann.

Die folgenden Probleme sind für Chomsky-3-Sprachen (also die Klasse der regulären Sprachen) entscheidbar:

Wortproblem: Ist ein Wort w in $L(G)$ (bzw. $L(A)$)?

Das Wortproblem ist für alle Grammatiken mit einem Chomsky-Typ größer 0 entscheidbar. Allerdings wächst die Laufzeit exponentiell mit der Wortlänge n. Für Chomsky-2- und Chomsky-3-Sprachen (d.h. -Grammatiken) gibt es wesentlich effizientere Algorithmen.

Leerheitsproblem: Ist $L(G) = \emptyset$?

Das Leerheitsproblem ist für Grammatiken vom Chomsky-Typ 2 und 3 entscheidbar.
Für andere Typen lassen sich Grammatiken konstruieren, für die nicht mehr entscheidbar ist, ob die Sprache leer ist.

Endlichkeitsproblem: Ist $|L(G)| < \infty$?

Das Endlichkeitsproblem ist für alle regulären Grammatiken lösbar.

Lemma 92

Sei n eine geeignete Pumping-Lemma-Zahl, die zur regulären Sprache L gehört. Dann gilt:

$$|L| = \infty \text{ gdw } (\exists z \in L)[n \leq |z| < 2n] \; .$$

Beweis:

Wir zeigen zunächst $\Leftarrow$:
Aus dem Pumping-Lemma folgt: $z = uvw$ für $|z| \geq n$ und $uv^iw \in L$ für alle $i \in \mathbb{N}_0$. Damit erzeugt man unendlich viele Wörter.

Nun wird $\Rightarrow$ gezeigt:
Dass es ein Wort z mit $|z| \geq n$ gibt, ist klar (es gibt ja unendlich viele Wörter). Mit Hilfe des Pumping-Lemmas lässt sich ein solches Wort auf eine Länge $< 2n$ reduzieren. □

Damit kann das Endlichkeitsproblem auf das Wortproblem zurückgeführt werden.

Schnittproblem: Ist $L(G_1) \cap L(G_2) = \emptyset$?

Das Schnittproblem ist für die Klasse der regulären Grammatiken entscheidbar, nicht aber für die Klasse der Chomsky-2-Grammatiken.

Äquivalenzproblem: Ist $L(G_1) = L(G_2)$?

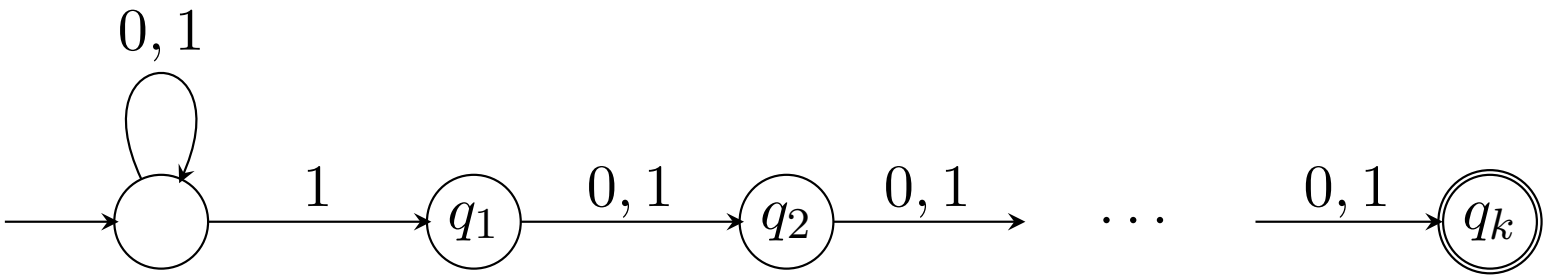
Das Äquivalenzproblem lässt sich auch wie folgt formulieren:

$$L_1 = L_2 \quad \Leftrightarrow \quad (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset$$

Wichtig für eine effiziente Lösung der Probleme ist, wie die Sprache gegeben ist. Hierzu ein Beispiel:

Beispiel 93

$L = \{w \in \{0,1\}^*$; das k-letzte Bit von w ist gleich $1\}$
Ein NFA für diese Sprache ist gegeben durch:



Insgesamt hat der NFA $k+1$ Zustände. Man kann nun diesen NFA in einen deterministischen Automaten umwandeln und stellt fest, dass der entsprechende DFA $\Omega(2^k)$ Zustände hat.

Da die Komplexität eines Algorithmus von der Größe der Eingabe abhängt, ist dieser Unterschied in der Eingabegröße natürlich wesentlich, denn es gilt:

kurze Eingabe wie beim NFA $\Rightarrow$ wenig Zeit für einen effizienten Algorithmus,

lange Eingabe wie beim DFA $\Rightarrow$ mehr Zeit für einen effizienten Algorithmus.

7. Das Wortproblem für kontextfreie Sprachen

7.1 Die Chomsky-Normalform

Sei $G = (V, \Sigma, P, S)$ eine kontexfreie Grammatik.

Definition 94

Eine kontextfreie Grammatik G ist in Chomsky-Normalform, falls alle Produktionen eine der Formen

$$A \to a \qquad A \in V, a \in \Sigma,$$
$$A \to BC \qquad A, B, C \in V, \text{ oder}$$
$$S \to \epsilon$$

haben.

Algorithmus zur Konstruktion einer (äquivalenten) Grammatik in Chomsky-Normalform

Eingabe: Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$

1. Wir fügen für jedes $a \in \Sigma$ zu V ein neues Nichtterminal Y_a hinzu, ersetzen in allen Produktionen a durch Y_a und fügen $Y_a \to a$ als neue Produktion zu P hinzu.

   /* linearer Zeitaufwand, Größe vervierfacht sich höchstens */

2. Wir ersetzen jede Produktion der Form

   $$A \to B_1 B_2 \cdots B_r \ (r \geq 3)$$

   durch

   $$A \to B_1 C_2,\ C_2 \to B_2 C_3, \ldots, C_{r-1} \to B_{r-1} B_r,$$

   wobei $C_2, \ldots, C_{r-1}$ neue Nichtterminale sind.

   /* linearer Zeitaufwand, Größe vervierfacht sich höchstens */

3. Für alle $C, D \in V$, $C \neq D$, mit

$$C \to^+ D,$$

füge für jede Produktion der Form

$$A \to BC \in P \text{ bzw. } A \to CB \in P$$

die Produktion

$$A \to BD \text{ bzw. } A \to DB$$

zu P hinzu.

/* quadratischer Aufwand pro A */

4. Für alle $\alpha \in V^2 \cup \Sigma$, für die $S \to^* \alpha$, füge $S \to \alpha$ zu P hinzu.
5. Streiche alle Produktionen der Form $A \to B$ aus P.

Zusammenfassend können wir festhalten:

Satz 95

Aus einer kontextfreien Grammatik $G = (V, \Sigma, P, S)$ der Größe $s(G)$ kann in Zeit $O(|V|^2 \cdot s(G))$ eine äquivalente kontextfreie Grammatik in Chomsky-Normalform der Größe $O(|V|^2 \cdot s(G))$ erzeugt werden.

7.2 Der Cocke-Younger-Kasami-Algorithmus

Der CYK-Algorithmus (oft auch Cocke-Kasami-Younger, CKY) entscheidet das Wortproblem für kontextfreie Sprachen, falls die Sprache in Form einer Grammatik in Chomsky-Normalform gegeben ist.

Eingabe: Grammatik $G = (V, \Sigma, P, S)$ in Chomsky-Normalform, $w = w_1 \dots w_n \in \Sigma^*$ mit der Länge n. O.B.d.A. $n > 0$.

Definition

$$V_{ij} := \{A \in V;\ A \to^* w_i \dots w_j\}\ .$$

Es ist klar, dass $w \in L(G) \Leftrightarrow S \in V_{1n}$.

Der CYK-Algorithmus berechnet alle V_{ij} rekursiv nach wachsendem $j - i$. Den Anfang machen die

$$V_{ii} := \{A \in V;\ A \to w_i \in P\},$$

der rekursive Aufbau erfolgt nach der Regel

$$V_{ij} = \bigcup_{i \le k < j} \{A \in V;\ (A \to BC) \in P \wedge B \in V_{ik} \wedge C \in V_{k+1,j}\} \quad \text{für } i < j.$$

Die Korrektheit dieses Aufbaus ist klar, wenn die Grammatik in Chomsky-Normalform vorliegt.

Zur Komplexität des CYK-Algorithmus

Es werden $\frac{n^2+n}{2}$ Mengen V_{ij} berechnet. Für jede dieser Mengen werden $|P|$ Produktionen und höchstens n Werte für k betrachtet. Der Test der Bedingung $(A \to BC) \in P \wedge B \in V_{ik} \wedge C \in V_{k+1,j}$ erfordert bei geeigneter Repräsentation der Mengen V_{ij} konstanten Aufwand. Der Gesamtaufwand ist also $O(|P|n^3)$.

Mit der gleichen Methode und dem gleichen Rechenaufwand kann man zu dem getesteten Wort, falls es in der Sprache ist, auch gleich einen Ableitungsbaum konstruieren, indem man sich bei der Konstruktion der V_{ij} nicht nur merkt, welche Nichtterminale sie enthalten, sondern auch gleich, warum sie sie enthalten, d.h. aufgrund welcher Produktionen sie in die Menge aufgenommen wurden.

7.3 Das Pumping-Lemma für kontextfreie Sprachen

Zur Erinnerung: Das Pumping-Lemma für reguläre Sprachen: Für jede reguläre Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so dass sich jedes Wort $z \in L$ mit $|z| \geq n$ zerlegen lässt in $z = uvw$ mit $|uv| \leq n$, $|v| \geq 1$ und $uv^*w \subseteq L$.

Zum Beweis haben wir $n = |Q|$ gewählt, wobei Q die Zustandsmenge eines L erkennenden DFA war. Das Argument war dann, dass beim Erkennen von z (mindestens) ein Zustand zweimal besucht werden muss und damit der dazwischen liegende Weg im Automaten beliebig oft wiederholt werden kann.

Völlig gleichwertig kann man argumentieren, dass bei der Ableitung von z mittels einer rechtslinearen Grammatik ein Nichtterminalsymbol (mindestens) zweimal auftreten muss und die dazwischen liegende Teilableitung beliebig oft wiederholt werden kann.

Genau dieses Argument kann in ähnlicher Form auch auf kontextfreie Grammatiken (in Chomsky-Normalform) angewendet werden:

Satz 96 (Pumping-Lemma)

Für jede kontextfreie Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so dass sich jedes Wort $z \in L$ mit $|z| \geq n$ zerlegen lässt in

$$z = uvwxy,$$

mit

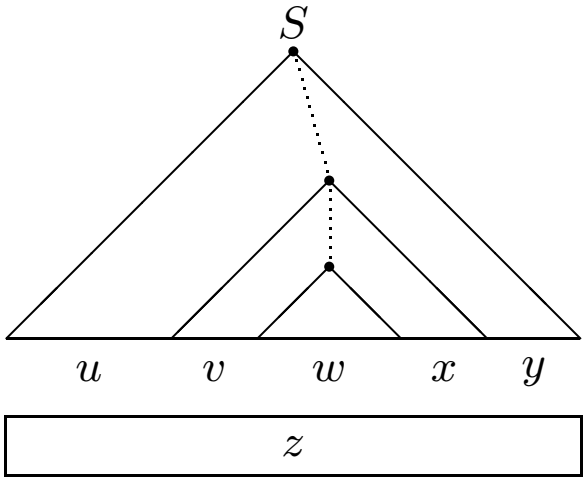
1. $|vx| \geq 1$,
2. $|vwx| \leq n$, *und*
3. $\forall i \in \mathbb{N}_0 : \; uv^i wx^i y \in L$.

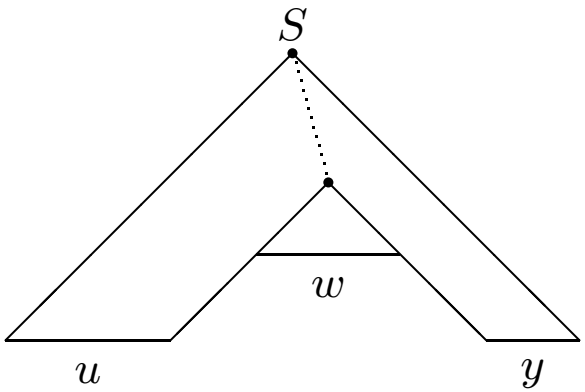
Beweis:

Sei $G = (V, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform mit $L(G) = L$. Wähle $n = 2^{|V|}$. Sei $z \in L(G)$ mit $|z| \geq n$. Dann hat der Ableitungsbaum für z (ohne die letzte Stufe für die Terminale) mindestens die Tiefe $|V| + 1$, da er wegen der Chomsky-Normalform den Verzweigungsgrad 2 hat.

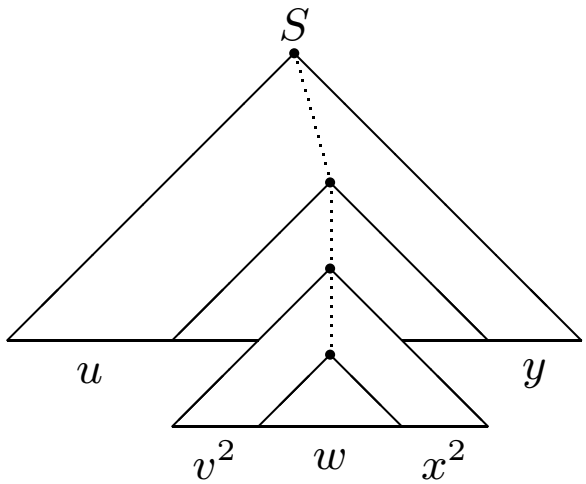
Auf einem Pfadabschnitt der Länge $\geq |V| + 1$ kommt nun mindestens ein Nichtterminal wiederholt vor. Die zwischen diesen beiden Vorkommen liegende Teilableitung kann nun beliebig oft wiederholt werden.

Beweis:





Dieser Ableitungsbaum zeigt
$uwy \in L$



Dieser Ableitungsbaum zeigt
$uv^2wx^2y \in L$

Beweis:

Sei $G = (V, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform mit $L(G) = L$. Wähle $n = 2^{|V|}$. Sei $z \in L(G)$ mit $|z| \geq n$. Dann hat der Ableitungsbaum für z (ohne die letzte Stufe für die Terminale) mindestens die Tiefe $|V| + 1$, da er wegen der Chomsky-Normalform den Verzweigungsgrad 2 hat.

Auf einem Pfadabschnitt der Länge $\geq |V| + 1$ kommt nun mindestens ein Nichtterminal wiederholt vor. Die zwischen diesen beiden Vorkommen liegende Teilableitung kann nun beliebig oft wiederholt werden.

Um $|vwx| \leq n$ zu erreichen, muss man das am weitesten unten liegende Doppelvorkommen eines solchen Nichtterminals wählen.

□

Beispiel 97

Wir wollen sehen, dass die Sprache

$$\{a^i b^i c^i;\ i \in \mathbb{N}_0\}$$

nicht kontextfrei ist.

Wäre sie kontextfrei, so könnten wir das Wort $a^n b^n c^n$ (n die Konstante aus dem Pumping-Lemma) aufpumpen, ohne aus der Sprache herauszufallen. Wir sehen aber leicht, dass dann für die Zerlegung $z = uvwxy$

$$\#_a(vx) = \#_b(vx) = \#_c(vx) > 0 \text{ und } v, x \in a^* + b^* + c^*$$

gelten muss, letzteres, damit die a's, b's und c's beim Pumpen nicht in der falschen Reihenfolge auftreten. Damit ergibt sich aber Widerspruch!

Zur Vereinfachung von Beweisen wie in dem gerade gesehenen Beispiel führen wir die folgende Verschärfung des Pumping-Lemmas ein:

Satz 98 (Ogdens Lemma)

Für jede kontextfreie Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n$ die folgende Aussage gilt: Werden in z mindestens n (beliebige) Buchstaben markiert, so lässt sich z zerlegen in

$$z = uvwxy,$$

so dass

1. *in vx mindestens ein Buchstabe und*
2. *in vwx höchstens n Buchstaben markiert sind und*
3. $(\forall i \in \mathbb{N}_0)[uv^i wx^i y \in L]$.

Bemerkung: Das Pumping-Lemma ist eine triviale Folgerung aus Ogdens Lemma (markiere alle Buchstaben in z).

Beweis:

Sei $G = (V, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform mit $L(G) = L$. Wähle $n = 2^{|V|+1}$. Sei $z \in L$ und seien in z mindestens n Buchstaben markiert. In einem Ableitungsbaum für z markieren wir alle (inneren) Knoten, deren linker *und* rechter Teilbaum *jeweils* mindestens ein markiertes Blatt enthalten. Es ist nun offensichtlich, dass es einen Pfad von der Wurzel zu einem Blatt gibt, auf dem mindestens $|V| + 1$ markierte innere Knoten liegen.

Beweis:

...

Wir betrachten die letzten $|V|+1$ markierten inneren Knoten eines Pfades mit maximaler Anzahl markierter Knoten; nach dem Schubfachprinzip sind zwei mit demselben Nichtterminal, z.B. A, markiert. Wir nennen diese Knoten v_1 und v_2. Seien die Blätter des Teilbaumes mit der Wurzel v_2 insgesamt mit w und die Blätter des Teilbaumes mit der Wurzel v_1 insgesamt mit vwx beschriftet. Es ist dann klar, dass die folgende Ableitung möglich ist:

$$S \to^* uAy \to^* uvAxy \to^* uvwxy\,.$$

Es ist auch klar, dass der Mittelteil dieser Ableitung weggelassen oder beliebig oft wiederholt werden kann.

Beweis:

...

Es bleibt noch zu sehen, dass vx mindestens einen und vwx höchstens n markierte Buchstaben enthält. Ersteres ist klar, da auch der Unterbaum von v_1, der v_2 nicht enthält, ein markiertes Blatt haben muss.

Letzteres ist klar, da der gewählte Pfad eine maximale Anzahl von markierten inneren Knoten hatte und unterhalb von v_1 nur noch höchstens $|V|$ markierte Knoten auf diesem Pfad sein können. Der Teilbaum mit Wurzel v_1 kann also maximal $2^{|V|+1} = n$ markierte Blätter haben. Formal kann man z.B. zeigen, dass ein Unterbaum, der auf jedem Ast maximal k markierte (innere) Knoten enthält, höchstens 2^k markierte Blätter enthält. □

Beispiel 99

$$L = \{a^i b^j c^k d^l;\ i = 0 \text{ oder } j = k = l\}\,.$$

Hier funktioniert das normale Pumping-Lemma nicht, da für z mit $|z| \geq n$ entweder z mit a beginnt und dann z.B. $v \in \{a\}^+$ sein kann oder aber z nicht mit a beginnt und dann eine zulässige Zerlegung $z = uvwxy$ sehr einfach gewählt werden kann.

Sei n die Konstante aus Ogdens Lemma. Betrachte das Wort $ab^n c^n d^n$ und markiere darin $bc^n d$. Nun gibt es eine Zerlegung $ab^n c^n d^n = uvwxy$, so dass vx mindestens ein markiertes Symbol enthält und $uv^2wx^2y \in L$.

Es ist jedoch leicht zu sehen, dass dies einen Widerspruch liefert, da vx höchstens zwei verschiedene der Symbole b, c, d enthalten kann, damit beim Pumpen nicht die Reihenfolge durcheinander kommt.

Bemerkung:
Wie wir gerade gesehen haben, gilt die Umkehrung des Pumping-Lemmas nicht allgemein (d.h., aus dem Abschluss einer Sprache unter der Pumpoperation des Pumping-Lemmas folgt i.A. nicht, dass die Sprache kontext-frei ist).
Es gibt jedoch stärkere Versionen des Pumping-Lemmas, für die auch die Umkehrung gilt. Siehe dazu etwa

David S. Wise:
A strong pumping lemma for context-free languages.
Theoretical Computer Science **3**, pp. 359–369, 1976

Richard Johnsonbaugh, David P. Miller:
Converses of pumping lemmas.
ACM SIGCSE Bull. **22**(1), pp. 27–30, 1990

Satz 100

Sei $G = (V, \Sigma, P, S)$ kontextfrei. Dann kann die Menge V' der Variablen $A \in V$, für die gilt:

$$(\exists w \in \Sigma^*)[A \to^* w]$$

in Zeit $O(|V| \cdot s(G))$ berechnet werden.

Beweis:

Betrachte folgenden Algorithmus:

$\Delta := \{A \in V;\ (\exists (A \to w) \in P \text{ mit } w \in \Sigma^*\};\ V' := \emptyset;$
while $\Delta \neq \emptyset$ **do**
  $V' := V' \cup \Delta$
  $\Delta := \{A \in V \setminus V';\ (\exists A \to \alpha) \in P \text{ mit } \alpha \in (V' \cup \Sigma)^*\}$
od

Induktion über die Länge der Ableitung. □

Definition 101

$A \in V$ heißt nutzlos, falls es keine Ableitung

$$S \to^* w, \quad w \in \Sigma^*$$

gibt, in der A vorkommt.

Satz 102

Die Menge der nutzlosen Variablen kann in Zeit $O(|V| \cdot s(G))$ bestimmt werden.

Beweis:

Sei V'' die Menge der nicht nutzlosen Variablen.
Offensichtlich gilt: $V'' \subseteq V'$ (V' aus dem vorigen Satz).
Falls $S \notin V'$, dann sind alle Variablen nutzlos.

Ansonsten:

$\Delta := \{S\}$; $V'' := \emptyset$;
while $\Delta \neq \emptyset$ **do**
$V'' := V'' \cup \Delta$
$\Delta := \{B \in V' \setminus V''$; $(\exists A \to \alpha B \beta) \in P$ mit $A \in V''$,
$\alpha, \beta \in (V' \cup \Sigma)^*\}$
od

Induktion über Länge der Ableitung: Am Ende des Algorithmus ist V'' gleich der Menge der nicht nutzlosen Variablen. □

Bemerkung: Alle nutzlosen Variablen und alle Produktionen, die nutzlose Variablen enthalten, können aus der Grammatik entfernt werden, ohne die erzeugte Sprache zu ändern.

Korollar 103

Für eine kontextfreie Grammatik G kann in Zeit $O(|V| \cdot s(G))$ entschieden werden, ob $L(G) = \emptyset$.

Beweis:

$L(G) = \emptyset \Longleftrightarrow S \notin V''$ (bzw. $S \notin V'$) □

Satz 104

Für eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ ohne nutzlose Variablen und in Chomsky-Normalform kann in linearer Zeit entschieden werden, ob

$$|L(G)| < \infty\,.$$

Beweis:

Definiere gerichteten Hilfsgraphen mit Knotenmenge V und

$$\text{Kante } A \to B \iff (A \to BC) \text{ oder } (A \to CB) \in P\,.$$

$L(G)$ ist endlich $\iff$ dieser Digraph enthält keinen Zyklus.

Verwende Depth-First-Search (DFS), um in linearer Zeit festzustellen, ob der Digraph Zyklen enthält. □

Satz 105

Seien kontextfreie Grammatiken $G_1 = (V_1, \Sigma_1, P_1, S_1)$ *und* $G_2 = (V_2, \Sigma_2, P_2, S_2)$ *gegeben. Dann können in linearer Zeit kontextfreie Grammatiken für*

1. $L(G_1) \cup L(G_2)$,
2. $L(G_1)L(G_2)$,
3. $(L(G_1))^*$

konstruiert werden. Die Klasse der kontextfreien Sprachen ist also unter Vereinigung, Konkatenation und Kleene'scher Hülle abgeschlossen.

Beweis:

Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $V_1 \cap V_2 = \emptyset$.

1. $V = V_1 \cup V_2 \cup \{S\}$; S neu
   $P = P_1 \cup P_2 \cup \{S \to S_1 | S_2\}$
2. $V = V_1 \cup V_2 \cup \{S\}$; S neu
   $P = P_1 \cup P_2 \cup \{S \to S_1 S_2\}$
3. $V = V_1 \cup \{S, S'\}$; S, S' neu
   $P = P_1 \cup \{S \to S' | \epsilon, S' \to S_1 S' | S_1\}$

Falls $\epsilon \in L(G_1)$ oder $\epsilon \in L(G_2)$, sind noch Korrekturen vorzunehmen, die hier als Übungsaufgabe überlassen bleiben. □

Satz 106

Die Klasse der kontextfreien Sprachen ist nicht abgeschlossen unter Durchschnitt oder Komplement.

Beweis:

Es genügt zu zeigen (wegen de Morgan (1806–1871)): nicht abgeschlossen unter Durchschnitt.

$$L_1 := \{a^i b^i c^j;\ i,j \geq 0\} \text{ ist kontextfrei}$$

$$L_2 := \{a^i b^j c^j;\ i,j \geq 0\} \text{ ist kontextfrei}$$

$$L_1 \cap L_2 = \{a^i b^i c^i;\ i \geq 0\} \text{ ist nicht kontextfrei}$$

□

Satz 107

Die Klasse der kontextfreien Sprachen ist abgeschlossen gegenüber Substitution (mit kontextfreien Mengen).

Beweis:

Ersetze jedes Terminal a durch ein neues Nichtterminal S_a und füge zu den Produktionen P für jedes Terminal a die Produktionen einer kontextfreien Grammatik $G_a = (V_a, \Sigma, P_a, S_a)$ hinzu.
Forme die so erhaltene Grammatik in eine äquivalente Chomsky-2-Grammatik um. □

7.5 Greibach-Normalform

Definition 108

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. G ist in Greibach-Normalform (benannt nach Sheila Greibach (UCLA)), falls jede Produktion $\neq S \to \epsilon$ von der Form

$$A \to a\alpha \text{ mit } a \in \Sigma,\ \alpha \in V^*$$

ist.

Lemma 109

Sei $G = (V, \Sigma, P, S)$ kontextfrei, $(A \to \alpha_1 B \alpha_2) \in P$, und sei $B \to \beta_1 | \beta_2 | \ldots | \beta_r$ die Menge der B-Produktionen (also die Menge der Produktionen mit B auf der linken Seite). Ersetzt man $A \to \alpha_1 B \alpha_2$ durch $A \to \alpha_1\beta_1\alpha_2 | \alpha_1\beta_2\alpha_2 | \ldots | \alpha_1\beta_r\alpha_2$, so ändert sich die von der Grammatik erzeugte Sprache nicht.

Lemma 110

Sei $G = (V, \Sigma, P, S)$ kontextfrei, sei $A \to A\alpha_1 | A\alpha_2 | \ldots | A\alpha_r$ die Menge der linksrekursiven A-Produktionen (alle $\alpha_i \neq \epsilon$, die Produktion $A \to A$ kommt o.B.d.A. nicht vor), und seien $A \to \beta_1 | \beta_2 | \ldots | \beta_s$ die restlichen A-Produktionen (ebenfalls alle $\beta_i \neq \epsilon$).

Ersetzen wir alle A-Produktionen durch

$$A \to \beta_1 | \ldots | \beta_s | \beta_1 A' | \ldots | \beta_s A'$$
$$A' \to \alpha_1 | \ldots | \alpha_r | \alpha_1 A' | \ldots | \alpha_r A' \,,$$

wobei A' ein neues Nichtterminal ist, so ändert sich die Sprache nicht, und die neue Grammatik enthält keine linksrekursive A-Produktion mehr.

Beweis:

Von A lassen sich vor der Transformation alle Zeichenreihen der Form

$$(\beta_1|\beta_2|\ldots|\beta_s)(\alpha_1|\alpha_2|\ldots|\alpha_r)^*$$

ableiten.

Dies ist auch nach der Transformation der Fall. Während vor der Transformation alle Zeichenreihen der obigen Form von rechts her aufgebaut werden, werden sie danach von links nach rechts erzeugt.

Die Umkehrung gilt ebenso. □

Satz 111

Zu jeder kontextfreien Grammatik G gibt es eine äquivalente Grammatik in Greibach-Normalform.

Beweis:

Sei o.B.d.A. $G = (V, \Sigma, P, S)$ mit $V = \{A_1, \ldots, A_m\}$ in Chomsky-Normalform und enthalte keine nutzlosen Variablen.

Bemerkung: Im folgenden Algorithmus werden ggf neue Variablen hinzugefügt, die Programmvariable m ändert sich dadurch entsprechend!

Beweis:

for $k = 1, \ldots, m$ **do**
for $j = 1, \ldots, k-1$ **do**
for all $(A_k \to A_j\alpha) \in P$ **do**
ersetze die Produktion gemäß der Konstruktion in Lemma 109
od
od
co die rechte Seite keiner A_k-Produktion beginnt nun noch mit einer Variablen A_j, $j < k$ **oc**

ersetze alle linksrekursiven A_k-Produktionen gemäß der Konstruktion in Lemma 110
co die rechte Seite keiner A_k-Produktion beginnt nun noch mit einer Variablen A_j, $j \leq k$ **oc**
od

Beweis (Forts.):

Da nun für jede Produktion der Form

$$A_k \to A_j \alpha$$

gilt:

$$j > k$$

(dies impliziert insbesondere, dass die rechte Seite jeder A_m-Produktion mit einem Terminalzeichen beginnt), können wir durch genügend oftmalige Anwendung der Konstruktion in Lemma 109 erreichen, dass jede rechte Seite mit einem Terminalzeichen beginnt. □

Korollar 112

Sei G eine kontextfreie Grammatik. Es gibt einen Algorithmus, der eine zu G äquivalente Grammatik in Greibach-Normalform konstruiert, deren rechte Seiten jeweils höchstens zwei Variablen enthalten.

Beweis:

Klar! □

Satz 113

Sei G eine CFG in Greibach-Normalform. Dann kann in linearer Zeit ein NPDA A konstruiert werden (welcher mit leerem Stack akzeptiert), so dass

$$L(A) = L(G) \; .$$

Beweis:

Sei o.B.d.A. $\epsilon \notin L(G)$.

Der Automat startet mit S auf dem Stack. Er sieht sich in jedem Schritt das oberste Stacksymbol A an und überprüft, ob es in G eine Produktion gibt, deren linke Seite A ist und deren rechte Seite mit dem Terminal beginnt, welches unter dem Lesekopf steht.

Sei also $G = (V, T, P, S)$.

Konstruiere NPDA $A = (Q, \Sigma, \Delta, q_0, Z_0, \delta)$ mit

$$\begin{aligned} Q &:= \{q_0\} \\ \Delta &:= V \\ \Sigma &:= T \\ Z_0 &:= S \end{aligned}$$

$$\delta(q_0, a, A) \ni (q_0, \alpha) \quad \text{für } (A \to a\alpha) \in P\ .$$

Beweis (Forts.):

Zu zeigen ist nun: $L(A) = L(G)$.

Hilfsbehauptung:

$$S \rightarrow_G^* w_1 \dots w_i A_1 \dots A_m \text{ mit } w_j \in T, A_j \in V \text{ per Linksableitung}$$

$$\Leftrightarrow (q_0, w_1 \dots w_i, Z_0) \rightarrow_A^* (q_0, \epsilon, A_1 \dots A_m)$$

Der Beweis erfolgt durch Induktion über die Anzahl der Schritte in der Linksableitung.

Beweis (Forts.):

Induktionsanfang ($i = 0$):

$$S \rightarrow_G^* S \qquad \Leftrightarrow \qquad (q_0, \epsilon, Z_0) \rightarrow_A^* (q_0, \epsilon, Z_0)$$

Beweis (Forts.):

Induktionsschritt $((i-1) \mapsto i)$:

$$
\begin{array}{lll}
 & S & \to_G^* w_1 \dots w_i A_1 \dots A_m \\
\Leftrightarrow & S & \to_G^* w_1 \dots w_{i-1} A' A_v \dots A_m \quad v \in \{1, \dots, m+1\} \\
 & & \to_G w_1 \dots w_i A_1 \dots A_m \\
 & & (\text{also } (A' \to w_i\, A_1 \dots A_{v-1}) \in P)
\end{array}
$$

gemäß Induktionsvoraussetzung

$$
\begin{array}{lll}
\Leftrightarrow & (q_0, w_1 \dots w_{i-1}, Z_0) & \to_A^* (q_0, \epsilon, A' A_v \dots A_m) \\
\Leftrightarrow & (q_0, w_1 \dots w_{i-1} w_i, Z_0) & \to_A^* (q_0, w_i, A' A_v \dots A_m) \\
 & & \to_A (q_0, \epsilon, A_1 \dots A_m) \\
 & & \text{da } (A' \to w_i\, A_1 \dots A_{v-1}) \in P) \\
 & & \\
\Leftrightarrow & (q_0, w_1 \dots w_i, Z_0) & \to_A^* (q_0, \epsilon, A_1 \dots A_m)
\end{array}
$$

Beweis (Forts.):

Aus der Hilfsbehauptung folgt

$$L(A) = L(G) \ .$$

□

Satz 114

Sei $A = (Q, \Sigma, \Delta, q_0, Z_0, \delta)$ ein NPDA, der mit leerem Keller akzeptiert. Dann ist $L(A)$ kontextfrei.

Beweis:

Wir definieren:

$G = (V, T, P, S)$

$T := \Sigma$

$V := Q \times \Delta \times Q \cup \{S\}$ wobei wir die Tupel mit $[,,]$ notieren

$P \ni S \to [q_0, Z_0, q]$ für $q \in Q$

$P \ni [q, Z, q_m] \to a[p, Z_1, q_1][q_1, Z_2, q_2] \cdots [q_{m-1}, Z_m, q_m]$

für $\delta(q, a, Z) \ni (p, Z_1 \cdots Z_m), \forall q_1, \ldots, q_m \in Q,$

mit $a \in \Sigma \cup \{\epsilon\}$.

Idee: Aus $[p, X, q]$ sollen sich alle die Wörter ableiten lassen, die der NPDA A lesen kann, wenn er im Zustand p mit (lediglich) X auf dem Stack startet und im Zustand q mit leerem Stack endet.

Beweis (Forts.):

Hilfsbehauptung:

$$[p, X, q] \rightarrow_G^* w \Leftrightarrow (p, w, X) \rightarrow_A^* (q, \epsilon, \epsilon)\,.$$

„$\Rightarrow$“: Induktion über die Länge l der Ableitung.

Induktionsanfang ($l = 1$):

$$\begin{aligned} & [p, X, q] \rightarrow_G w \\ \Rightarrow \quad & \delta(p, w, X) \ni (q, \epsilon) \\ \Rightarrow \quad & (p, w, X) \rightarrow_A (q, \epsilon, \epsilon) \end{aligned}$$

Beweis (Forts.):

Induktionsschritt $((l-1) \mapsto l)$:

Gelte

$$[p, X, q_{m+1}] \to_G a[q_1, X_1, q_2][q_2, X_2, q_3] \cdots [q_m, X_m, q_{m+1}]$$
$$\to_G^* aw^{(1)} \cdots w^{(m)} = w$$

mit $(q_1, X_1 \cdots X_m) \in \delta(p, a, X)$, $[q_i, X_i, q_{i+1}] \to_G^{l_i} w^{(i)}$ und $\sum l_i < l$.

Dann gilt gemäß Induktionsvoraussetzung

$$\Rightarrow \qquad (q_i, w^{(i)}, X_i) \to_A^{l_i} (q_{i+1}, \epsilon, \epsilon) \quad \forall i \in \{1, \ldots, m\}$$

$$\Rightarrow \quad (q, \underbrace{aw^{(1)} \cdots w^{(m)}}_{=w}, X) \to_A (q_1, w^{(1)} \cdots w^{(m)}, X_1 \cdots X_m)$$

$$\to_A^{<l} (q_{m+1}, \epsilon, \epsilon) \; .$$

Beweis (Forts.):

„$\Leftarrow$“: Induktion über die Länge l einer Rechnung des NPDAs A

Induktionsanfang ($l = 1$):

$$
\begin{aligned}
& (p, w, X) \to_A (q, \epsilon, \epsilon) \\
\Rightarrow \quad & (q, \epsilon) \in \delta(p, w, X) \qquad (\text{also } |w| \leq 1) \\
\Rightarrow \quad & ([p, X, q] \to w) \in P\,.
\end{aligned}
$$

Beweis (Forts.):

Induktionsschritt $((l-1) \mapsto l)$:

Sei

$$(p, w, X) \to_A (q_1, w', X_1 \cdots X_m)$$
$$\to_A^* (q, \epsilon, \epsilon)$$

eine Rechnung von A der Länge l, mit $w = ew'$ und $e \in \Sigma \cup \{\epsilon\}$.

Nach Definition gibt es

$$([p, X, q] \to e[q_1 X_1 q_2] \cdots [q_m X_m q_{m+1}]) \in P \quad \text{mit } q_{m+1} = q$$

und eine Zerlegung $w' = w^{(1)} \cdots w^{(m)}$, so dass $w^{(1)} \cdots w^{(i)}$ der von A zu dem Zeitpunkt verarbeitete Teilstring von w' ist, wenn X_{i+1} zum ersten Mal oberstes Stacksymbol (bzw., für $i = m$, der Stack leer) wird.

Beweis (Forts.):

Gemäß Induktionsvoraussetzung gilt also

$$(q_i, w^{(i)}, X_i) \to_A^{l_i} (q_{i+1}, \epsilon, \epsilon) \qquad \text{mit } \sum l_i < l \text{ und}$$
$$[q_i, X_i, q_{i+1}] \to_G^* w^{(i)} \ .$$

Also folgt:

$$\begin{aligned}[p, X, q] &\to_G e[q_1, X_1, q_2] \cdots [q_m, X_m, q_{m+1}] \quad \text{mit } q_{m+1} = q \\ &\to_G^{<l} ew^{(1)} \cdots w^{(m)} = w\end{aligned}$$

Aus der Hilfsbehauptung folgt der Satz. □

Satz 115

Folgende Aussagen sind äquivalent:

- L *wird von einer kontextfreien Grammatik erzeugt.*
- L *wird von einem NPDA akzeptiert.*

Beweis:

Folgt aus den vorhergehenden Sätzen. □

Beispiel (Anwendung von Satz 41)

Korollar 116

Die Sprache

$$L = \{0^i 1^j 2^k;\ i = j \text{ oder } j = k\} \subset \{0,1,2\}^*$$

ist kontextfrei, aber nicht deterministisch kontextfrei
($\in$ CFL $\setminus$ DCFL).

Beweis:

L wird erzeugt z.B. von der CFG mit den Produktionen

$$S \to A \mid B \mid AB \mid C \mid D \mid CD \mid \epsilon$$

$$A \to 01 \mid 0A1 \qquad B \to 2 \mid 2B$$

$$C \to 0 \mid 0C \qquad D \to 12 \mid 1D2$$

Beispiel (Anwendung von Satz 41)

Korollar 116

Die Sprache

$$L = \{0^i 1^j 2^k;\ i = j \text{ oder } j = k\} \subset \{0,1,2\}^*$$

ist kontextfrei, aber nicht deterministisch kontextfrei ($\in$ CFL $\setminus$ DCFL).

Beweis:

Wäre $L \in$ DCFL, dann auch

$$L' := \bar{L} \cap 0^*1^*2^* = \{0^i 1^j 2^k;\ i \neq j \text{ und } j \neq k\}.$$

Mit Hilfe von Ogden's Lemma sieht man aber leicht, dass dies nicht der Fall ist. □

Beispiel 117 (Grammatik für Arithmetische Ausdrücke)

Regeln:

$$
\begin{aligned}
S &\to A \\
A &\to E \mid A + E \mid A - E \\
E &\to P \mid E * P \mid E/P \\
P &\to (A) \mid a
\end{aligned}
$$

Wir betrachten für das Parsen einen bottom-up-Ansatz, wobei die Reduktionen von links nach rechts angewendet werden.

Beispiel 117 (Grammatik für Arithmetische Ausdrücke)

Dabei können sich bei naivem Vorgehen allerdings Sackgassen ergeben, die dann aufgrund des Backtracking zu einem ineffizienten Algorithmus führen:

Regeln:

$$S \to A$$
$$A \to E \mid A + E \mid A - E$$
$$E \to P \mid E * P \mid E/P$$
$$P \to (A) \mid a$$

Ableitung:

a	$+$	a	$*$	a	
P	$+$	a	$*$	a	
E	$+$	a	$*$	a	
A	$+$	a	$*$	a	
S	$+$	a	$*$	a	Sackgasse!

Beispiel 117 (Grammatik für Arithmetische Ausdrücke)

. . . oder auch

Regeln:

$$S \to A$$
$$A \to E \mid A + E \mid A - E$$
$$E \to P \mid E * P \mid E/P$$
$$P \to (A) \mid a$$

Ableitung:

$$\begin{array}{llllll}
a & + & a & * & a & \\
P & + & a & * & a & \\
E & + & a & * & a & \\
A & + & a & * & a & \\
A & + & P & * & a & \\
A & + & E & * & a & \\
 & A & & * & a & \\
 & A & & * & P & \\
 & A & & * & E & \text{Sackgasse!}
\end{array}$$

Beispiel 117 (Grammatik für Arithmetische Ausdrücke)

Zur Behebung des Problems führen wir für jede Ableitungsregel (besser hier: Reduktionsregel) einen Lookahead (der Länge k) ein (in unserem Beispiel $k = 1$) und legen fest, dass eine Ableitungsregel nur dann angewendet werden darf, wenn die nächsten k Zeichen mit den erlaubten Lookaheads übereinstimmen.

Beispiel 117 (Grammatik für Arithmetische Ausdrücke)

Produktion	Lookaheads (der Länge 1)
$S \to A$	ϵ
$A \to E$	$+, -,), \epsilon$
$A \to A + E$	$+, -,), \epsilon$
$A \to A - E$	$+, -,), \epsilon$
$E \to P$	beliebig
$E \to E * P$	beliebig
$E \to E/P$	beliebig
$P \to (A)$	beliebig
$P \to a$	beliebig

Beispiel 117 (Grammatik für Arithmetische Ausdrücke)

Damit ergibt sich

Produktion	Lookaheads
$S \to A$	ϵ
$A \to E$	$+, -,), \epsilon$
$A \to A + E$	$+, -,), \epsilon$
$A \to A - E$	$+, -,), \epsilon$
$E \to P$	beliebig
$E \to E * P$	beliebig
$E \to E/P$	beliebig
$P \to (A)$	beliebig
$P \to a$	beliebig

Ableitung:

$$
\begin{array}{ccccc}
a & + & a & * & a \\
P & + & a & * & a \\
E & + & a & * & a \\
A & + & a & * & a \\
A & + & P & * & a \\
A & + & E & * & a \\
A & + & E & * & P \\
A & + & & E & \\
 & & A & & \\
 & & S & &
\end{array}
$$

Definition 118

Eine kontextfreie Grammatik ist eine $LR(k)$-Grammatik, wenn man durch Lookaheads der Länge k erreichen kann, dass bei einer Reduktion von links nach rechts in jedem Schritt höchstens eine Produktion/Reduktion anwendbar ist.

Korollar 119

Jede kontextfreie Sprache, für die es eine $LR(k)$-Grammatik gibt, ist deterministisch kontextfrei.

Bemerkung:
Es gibt eine (im allgemeinen nicht effiziente) Konstruktion, um aus einer $LR(k)$-Grammatik, $k > 1$, eine äquivalente $LR(1)$-Grammatik zu machen.

Korollar 120

Die folgenden Klassen von Sprachen sind gleich:

- *die Klasse DCFL,*
- *die Klasse der $LR(1)$-Sprachen.*

7.8 $LL(k)$-Grammatiken

Beispiel 121 (Noch eine Grammatik)

Regeln:

$$
\begin{array}{ll}
S \to A & S \to A \\
A \to E \mid E + A & A \to EA' \\
 & A' \to +A \mid \epsilon \\
E \to P \mid P * E & E \to PE' \\
 & E' \to *E \mid \epsilon \\
P \to (A) \mid a & P \to (A) \mid a
\end{array}
$$

Wir betrachten nun für das Parsen einen top-down-Ansatz, wobei die Produktionen in Form einer Linksableitung angewendet werden.

Beispiel 121 (Noch eine Grammatik)

Regeln:

$$S \to A$$
$$A \to E \mid E + A$$
$$E \to P \mid P * E$$
$$P \to (A) \mid a$$

Ableitung:

$$S$$
$$A$$
$$E$$
$$P$$
$$a \quad + \quad a \quad * \quad a$$

Beispiel 121 (Noch eine Grammatik)

Wir bestimmen nun für jede Produktion $A \to \alpha$ ihre Auswahlmenge, das heißt die Menge aller terminalen Präfixe der Länge $\leq k$ der von α ableitbaren Zeichenreihen.
Es ergibt sich (der Einfachheit lassen wir ϵ-Produktionen zu):

$$\begin{array}{lll}
S & \to A & \{a, (\} \\
A & \to EA' & \{a, (\} \\
A' & \to +A & \{+\} \\
A' & \to \epsilon & \{), \epsilon\} \\
E & \to PE' & \{a, (\} \\
E' & \to *E & \{*\} \\
E' & \to \epsilon & \{+,), \epsilon\} \\
P & \to (A) & \{(\} \\
P & \to a & \{a\}
\end{array}$$

Beispiel 121 (Noch eine Grammatik)

Damit ergibt sich

$$
\begin{array}{lll}
S & \to A & \{a, (\} \\
A & \to EA' & \{a, (\} \\
A' & \to +A & \{+\} \\
A' & \to \epsilon & \{), \epsilon\} \\
E & \to PE' & \{a, (\} \\
E' & \to *E & \{*\} \\
E' & \to \epsilon & \{+,), \epsilon\} \\
P & \to (A) & \{(\} \\
P & \to a & \{a\}
\end{array}
$$

Ableitung:

$$
\begin{array}{l}
S \\
EA' \\
aE'A' \\
\quad \vdots \\
a + PE'A' \\
\quad \vdots \\
a + a * PE'A' \\
\quad \vdots \\
a + a * a
\end{array}
$$

Bemerkungen:

1. Parser für $LL(k)$-Grammatiken entsprechen der Methode des rekursiven Abstiegs (recursive descent).
2. $LL(k)$ ist eine strikte Teilklasse von $LR(k)$.
3. Es gibt $L \in$ DCFL, so dass $L \notin LL(k)$ für alle k.

Bemerkung
Für die Praxis (z.B. Syntaxanalyse von Programmen) sind polynomiale Algorithmen wie CYK noch zu langsam. Für Teilklassen von CFLs sind schnellere Algorithmen bekannt, z.B.

Jay Earley:
An Efficient Context-free Parsing Algorithm.
Communications of the ACM **13**(2), pp. 94–102, 1970

8. Das Halteproblem

Definition 122

Unter dem speziellen Halteproblem H_s versteht man die folgende Sprache:

$$H_s = \{w \in \{0,1\}^*;\ M_w \text{ angesetzt auf } w \text{ hält}\}$$

Hierbei ist $(M_\epsilon, M_0, M_1, \ldots)$ eine berechenbare Auflistung der Turing-Maschinen.

Wir definieren weiter

Definition 123

$$L_d = \{w \in \Sigma^*;\ M_w \text{ akzeptiert } w \text{ nicht}\}$$

Satz 124

L_d ist nicht rekursiv aufzählbar.

Beweis:

Wäre L_d r.a., dann gäbe es ein w, so dass $L_d = L(M_w)$.
Dann gilt:

$$\begin{aligned} M_w \text{ akzeptiert } w \text{ nicht} &\Leftrightarrow w \in L_d \\ &\Leftrightarrow w \in L(M_w) \\ &\Leftrightarrow M_w \text{ akzeptiert } w \end{aligned}$$

$\Longrightarrow$ Widerspruch!

Korollar 125

L_d ist nicht entscheidbar.

Satz 126

H_s ist nicht entscheidbar.

Beweis:

Angenommen, es gäbe eine Turing-Maschine M, die H_s entscheidet. Indem man i.W. die Antworten von M umdreht, erhält man eine TM, die L_d entscheidet. Widerspruch!

9. Unentscheidbarkeit

Definition 127

Unter dem (allgemeinen) Halteproblem H versteht man die Sprache

$$H = \{\langle x, w\rangle \in \{0,1\}^*;\ M_x \text{ angesetzt auf } w \text{ hält}\}$$

Satz 128

Das Halteproblem H ist nicht entscheidbar.

Beweis:

Eine TM, die H entscheidet, könnten wir benutzen, um eine TM zu konstruieren, die H_s entscheidet.

Bemerkung: H und H_s sind beide rekursiv aufzählbar!

Definition 129

Seien A, $B \subseteq \Sigma^*$. Dann heißt A (effektiv) reduzierbar auf B gdw $\exists f : \Sigma^* \to \Sigma^*$, f total und berechenbar mit

$$(\forall w \in \Sigma^*)[w \in A \Leftrightarrow f(w) \in B]\,.$$

Wir schreiben auch

$$A \hookrightarrow_f B \text{ bzw. } A \hookrightarrow B\,.$$

bzw. manchmal

$$A \leq B \text{ oder auch } A \preceq_f B\,.$$

Ist A mittels f auf B reduzierbar, so gilt insbesondere

$$f(A) \subseteq B \text{ und } f(\bar{A}) \subseteq \bar{B}\,.$$

Satz 130

Sei $A \hookrightarrow_f B$.

(i) B *rekursiv* $\Rightarrow$ A *rekursiv.*

(ii) B *rekursiv aufzählbar* $\Rightarrow$ A *rekursiv aufzählbar.*

Beweis:

(i) $\chi_A = \chi_B \circ f$.

(ii) $\chi'_A = \chi'_B \circ f$.

Definition 131

Das Halteproblem auf leerem Band H_0 ist

$$H_0 = \{w \in \{0,1\}^*;\ M_w \text{ hält auf leerem Band}\}\,.$$

Satz 132

H_0 ist unentscheidbar (nicht rekursiv).

Beweis:

Betrachte die Abbildung f, die definiert ist durch:

$$\{0,1\}^* \ni w \mapsto f(w),$$

$f(w)$ ist die Gödelnummer einer TM, die, auf leerem Band angesetzt, zunächst $c_2(w)$ auf das Band schreibt und sich dann wie $M_{c_1(w)}$ (angesetzt auf $c_2(w)$) verhält. Falls das Band nicht leer ist, ist es unerheblich, wie sich $M_{f(w)}$ verhält.

f ist total und berechenbar.

Es gilt: $w \in H \Leftrightarrow M_{c_1(w)}$ angesetzt auf $c_2(w)$ hält
$\Leftrightarrow M_{f(w)}$ hält auf leerem Band
$\Leftrightarrow f(w) \in H_0$

also $H \hookrightarrow_f H_0$ und damit H_0 unentscheidbar.

Bemerkung

Es gibt also keine allgemeine algorithmische Methode, um zu entscheiden, ob ein Programm anhält.

Kapitel V Algorithmen und Datenstrukturen

1. Analyse von Algorithmen

Wir wollen die Ressourcen bestimmen, die, in Abhängigkeit von der Eingabe, ein Algorithmus benötigt, z.B.

1. Laufzeit
2. Speicherplatz
3. Anzahl Prozessoren
4. Programmlänge
5. Tinte
6. ...

Beispiel 133

Prozedur für Fakultätsfunktion

func $fak(n)$
  $m := 1$
  for $i := 2$ **to** n **do**
    $m := m * i$
  do
  return (m)

Diese Prozedur benötigt $O(n)$ Schritte bzw. arithmetische Operationen.

Jedoch: die Länge der Ausgabe ist etwa

$$\lceil \log_2 n! \rceil = \Omega(n \log n) \text{ Bits.}$$

Bemerkung:

Um die Zahl $n \in \mathbb{N}_0$ in Binärdarstellung hinzuschreiben, benötigt man

$$\ell(n) := \begin{cases} 1 & \text{für } n = 0 \\ 1 + \lfloor \log_2(n) \rfloor & \text{sonst} \end{cases}$$
$$= \begin{cases} 1 & \text{für } n = 0 \\ \lceil \log_2(n+1) \rceil & \text{sonst} \end{cases}$$

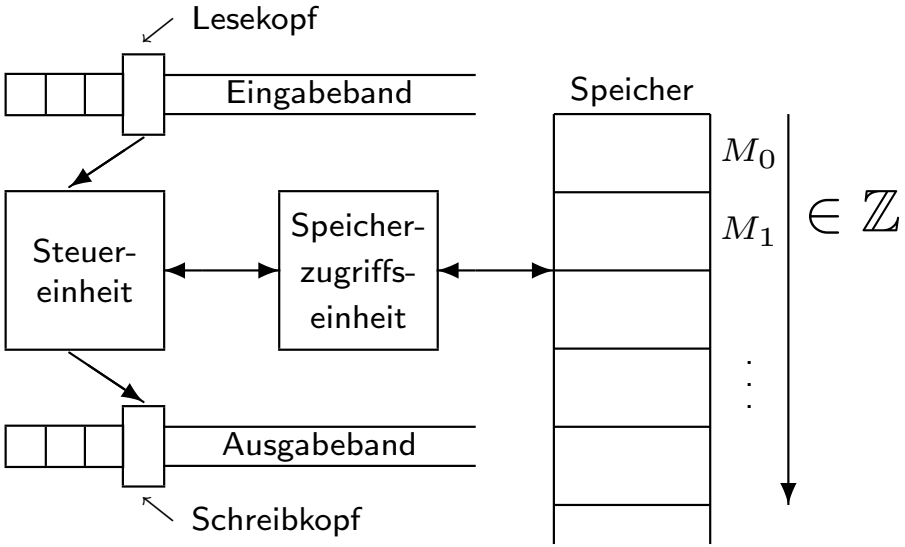
Um die Notation zu vereinfachen, vereinbaren wir im Zusammenhang mit Komplexitätsabschätzungen

$$\log(0) := 0\,.$$

1.1 Referenzmaschine

Wir wählen als Referenzmaschine die Registermaschine (engl. random access machine, RAM) oder auch WHILE-Maschine, also eine Maschine, die WHILE-Programme verarbeiten kann, erweitert durch

- IF ... THEN ... ELSE ... FI
- Multiplikation und Division
- indirekte Adressierung
- arithmetische Operationen wie $\sqrt{n}$, $\sin n$,...



Registermaschine

1.2 Wachstumsverhalten von Funktionen

f, g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_+$.

- $g = \mathcal{O}(f)$ [auch: $g(n) = \mathcal{O}(f(n))$ oder $g \in \mathcal{O}(f)$] gdw.

$$(\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \leq c \cdot f(n)]$$

- $g = \Omega(f)$ [auch: $g(n) = \Omega(f(n)$ oder $g \in \Omega(f)$] gdw.

$$(\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \geq c \cdot f(n)]$$

- $g = \Theta(f)$ gdw. $g = \mathcal{O}(f)$ und $g = \Omega(f)$

f, g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_+$.

- $g = o(f)$ gdw.

$$(\forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \leq c \cdot f(n)]$$

- $g = \omega(f)$ gdw.

$$(\forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \geq c \cdot f(n)]$$

- $g = \Omega_\infty(f)$ gdw.

$$(\exists c > 0)[g(n) \geq c \cdot f(n) \text{ für unendlich viele } n]$$

- $g = \omega_\infty(f)$ gdw.

$$(\forall c > 0)[g(n) \geq c \cdot f(n) \text{ für unendlich viele } n]$$

Beispiel 134

- n^3 ist nicht $\mathcal{O}\left(\frac{n^3}{\log n}\right)$.
- $n^3 + n^2$ ist nicht $\omega(n^3)$.
- $100n^3$ ist nicht $\omega(n^3)$.

Bemerkung:
Die Groß-O-Notation wurde von D. E. Knuth in der Algorithmenanalyse eingeführt, siehe z.B.

Donald E. Knuth:
Big omicron and big omega and big theta.
SIGACT News **8**(2), pp. 18–24, ACM SIGACT, 1976

Sie wurde ursprünglich von Paul Bachmann (1837–1920) entwickelt und von Edmund Landau (1877–1938) in seinen Arbeiten verbreitet.

1.2 Zeit- und Platzkomplexität

Beim Zeitbedarf zählt das uniforme Kostenmodell die Anzahl der von der Registermaschine durchgeführten Elementarschritte, beim Platzbedarf die Anzahl der benutzten Speicherzellen.

Das logarithmische Kostenmodell zählt für den Zeitbedarf eines jeden Elementarschrittes

$$\ell(\text{größter beteiligter Operand}),$$

beim Platzbedarf

$$\sum_x \ell(\text{größter in } x \text{ gespeicherter Wert}) \,,$$

also die maximale Anzahl der von allen Variablen x benötigten Speicherbits.

Beispiel 135

Wir betrachten die Prozedur

func $dbexp(n)$
  $m := 2$
  for $i := 1$ **to** n **do**
    $m := m^2$
  do
  return (m)

Die Komplexität von $dbexp$, die $n \mapsto 2^{2^n}$ berechnet, ergibt sich bei Eingabe n zu

	Zeit	Platz
uniform	$\Theta(n)$	$\Theta(1)$
logarithmisch	$\Theta(2^n)$	$\Theta(2^n)$

Bemerkung:
Das (einfachere) uniforme Kostenmodell sollte also nur verwendet werden, wenn alle vom Algorithmus berechneten Werte gegenüber den Werten in der Eingabe nicht zu sehr wachsen, also z.B. nur polynomiell.

1.3 Worst Case-Analyse

Sei A ein Algorithmus. Dann sei

$$T_A(x) := \text{Laufzeit von } A \text{ bei Eingabe } x\,.$$

Diese Funktion ist i.A. zu aufwändig und zu detailliert. Stattdessen:

$$T_A(n) := \max_{|x|=n} T_A(x) \quad (= \text{maximale Laufzeit bei Eingabelänge } n)$$

1.4 Average Case-Analyse

Oft erscheint die Worst Case-Analyse als zu pessimistisch. Dann:

$$T_A^{\text{avg}}(n) = \frac{\sum_{x;\,|x|=n} T_A(x)}{|\{x;\ |x| = n\}|}$$

oder allgemeiner

$$\begin{aligned} T_A^{\text{avg}}(n) &= \sum T_A(x) \cdot \text{Pr}\,\{x \mid |x| = n\} \\ &= \mathbf{E}_{|x|=n}\left[T_A(x)\right]\ , \end{aligned}$$

wobei eine (im Allgemeinen beliebige) Wahrscheinlichkeitsverteilung zugrunde liegt.

Bemerkung:
Wir werden Laufzeiten $T_A(n)$ meist nur bis auf einen multiplikativen Faktor genau berechnen, d.h. das genaue Referenzmodell, Fragen der Implementierung, usw. spielen dabei eine eher untergeordnete Rolle.

2. Sortierverfahren

Unter einem Sortierverfahren versteht man ein algorithmisches Verfahren, das als Eingabe eine Folge $a_1, \ldots, a_n$ von n Schlüsseln $\in \Sigma^*$ erhält und als Ausgabe eine auf- oder absteigend sortierte Folge dieser Elemente liefert. Im Folgenden werden wir im Normalfall davon ausgehen, dass die Elemente aufsteigend sortiert werden sollen. Zur Vereinfachung nehmen wir im Normalfall auch an, dass alle Schlüssel paarweise verschieden sind.

Für die betrachteten Sortierverfahren ist natürlich die Anzahl der Schlüsselvergleiche eine untere Schranke für die Laufzeit, und oft ist letztere von der gleichen Größenordnung, d.h.

$$\text{Laufzeit} = O(\text{Anzahl der Schlüsselvergleiche}) .$$

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

Satz 136

SelectionSort *benötigt zum Sortieren von* n *Elementen genau* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Feldes A) zur Bestimmung des maximalen Schlüssels in $A[1..i]$ ist $i-1$.

Damit ergibt sich die Laufzeit von SelectionSort zu

$$T_{\text{SelectionSort}} = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

Der Rang von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$ kann trivial mit $i-1$ Vergleichen bestimmt werden. Damit ergibt sich

Satz 137

INSERTIONSORT *benötigt zum Sortieren von* n *Elementen maximal* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Übungsaufgabe!

Die Rangbestimmung kann durch binäre Suche verbessert werden. Dabei benötigen wir, um den Rang eines Elementes in einer k-elementigen Menge zu bestimmen, höchstens

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

Vergleiche, wie man durch Induktion leicht sieht.

Satz 138

InsertionSort *mit binärer Suche für das Einsortieren benötigt zum Sortieren von n Elementen maximal*

$$n \lceil \mathrm{ld}\, n \rceil$$

Vergleiche.

Beweis:

Die Abschätzung ergibt sich durch einfaches Einsetzen.

Achtung: Die Laufzeit von InsertionSort ist dennoch auch bei Verwendung von binärer Suche beim Einsortieren im schlechtesten Fall $\Omega(n^2)$, wegen der notwendigen Verschiebung der Feldelemente.

Verwendet man statt des Feldes eine doppelt verkettete Liste, so wird zwar das Einsortieren vereinfacht, es kann jedoch die binäre Suche nicht mehr effizient implementiert werden.

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$**oc**
  if $s \leq r$ **return fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Satz 139

MergeSort *sortiert ein Feld der Länge* n *mit maximal* $n \cdot \lceil \mathrm{ld}(n) \rceil$ *Vergleichen.*

Beweis:

In jeder Rekursionstiefe werde der Vergleich dem kleineren Element zugeschlagen. Dann erhält jedes Element pro Rekursionstiefe höchstens einen Vergleich zugeschlagen.

2.4 Quick-Sort

Beim Quick-Sort-Verfahren wird in jeder Phase ein Element p der zu sortierenden Folge als Pivot-Element ausgewählt (wie dies geschehen kann, wird noch diskutiert). Dann wird in situ und mit einer linearen Anzahl von Vergleichen die zu sortierende Folge so umgeordnet, dass zuerst alle Elemente $< p$, dann p selbst und schließlich alle Elemente $> p$ kommen. Die beiden Teilfolgen links und rechts von p werden dann mit Quick-Sort rekursiv sortiert (Quick-Sort ist also ein Divide-and-Conquer-Verfahren).

Quick-Sort benötigt im schlechtesten Fall, nämlich wenn als Pivot-Element stets das kleinste oder größte der verbleibenden Elemente ausgewählt wird,

$$\sum_{i=1}^{n-1}(n-i) = \binom{n}{2}$$

Vergleiche.

Satz 140

QuickSort *benötigt zum Sortieren eines Feldes der Länge* n *durchschnittlich nur*

$$2\ln(2) \cdot n \operatorname{ld}(n) + O(n)$$

viele Vergleiche.

Beweis:

Siehe Vorlesung Diskrete Wahrscheinlichkeitstheorie (DWT).

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
   Analyse Übungsaufgabe!
3. Wähle ein zufälliges Element als Pivotelement
   liefert die o.a. durchschnittliche Laufzeit, benötigt aber einen Zufallsgenerator.

2.5 Heap-Sort

Definition 141

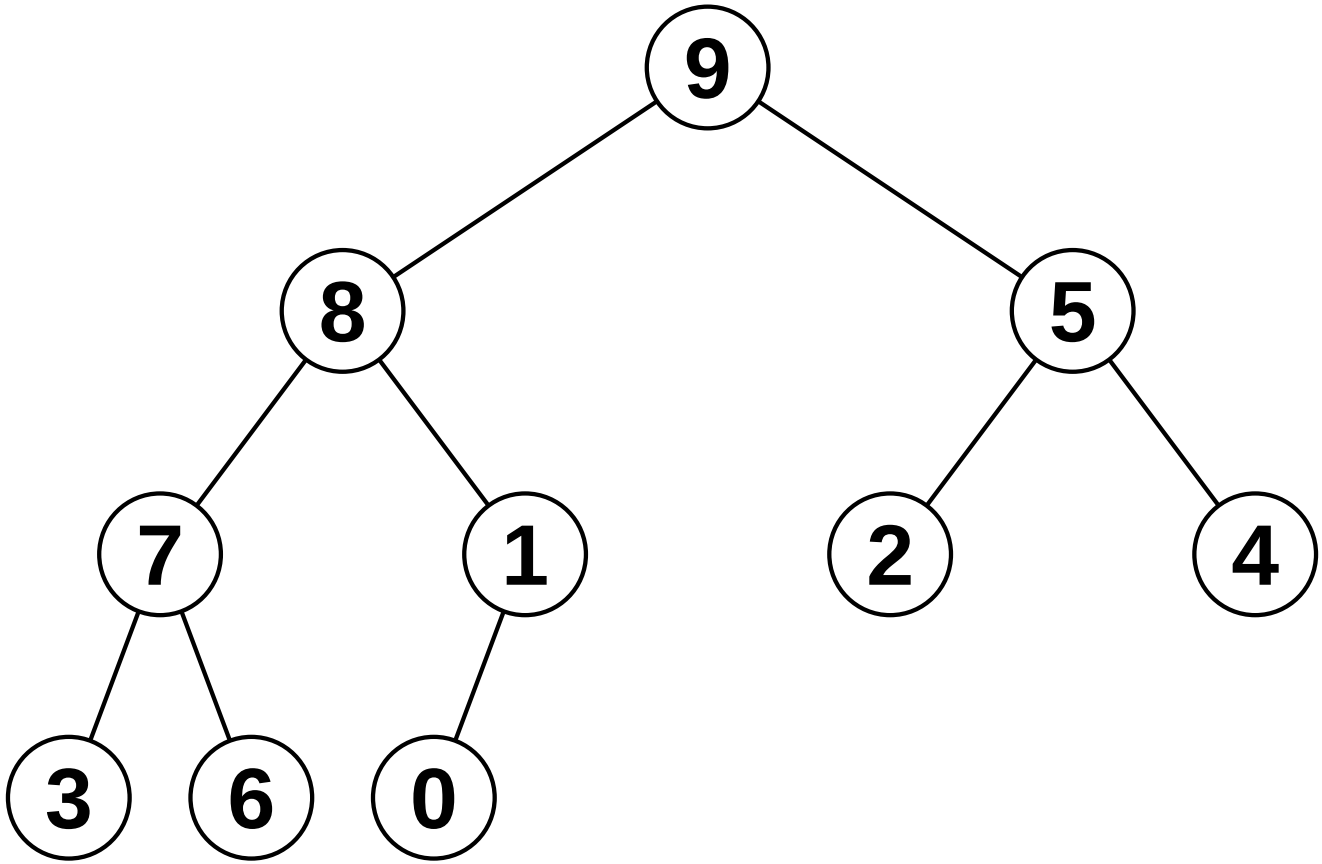
Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

Bemerkungen:

1. Die hier definierte Variante ist ein max-Heap.
2. Die Bezeichnung heap wird in der Algorithmentheorie auch allgemeiner für Prioritätswarteschlangen benutzt!

Beispiel 142



Der Algorithmus HeapSort besteht aus zwei Phasen.

1. In der ersten Phase wird aus der unsortierten Folge von n Elementen ein Heap gemäß Definition aufgebaut.
2. In der zweiten Phase wird dieser Heap ausgegeben, d.h. ihm wird n-mal jeweils das größte Element entnommen (das ja an der Wurzel steht), dieses Element wird in die zu sortierende Folge aufgenommen und die Heap-Eigenschaften wird wieder hergestellt.

Betrachten wir nun zunächst den Algorithmus REHEAP zur Korrektur der Datenstruktur, falls die Heap-Bedingung höchstens an der Wurzel verletzt ist.

Algorithmus REHEAP

sei v die Wurzel des Heaps;
while Heap-Eigenschaft in v nicht erfüllt **do**
  sei v' das Kind von v mit dem größeren Schlüssel
  vertausche die Schlüssel in v und v'
  $v := v'$
od

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

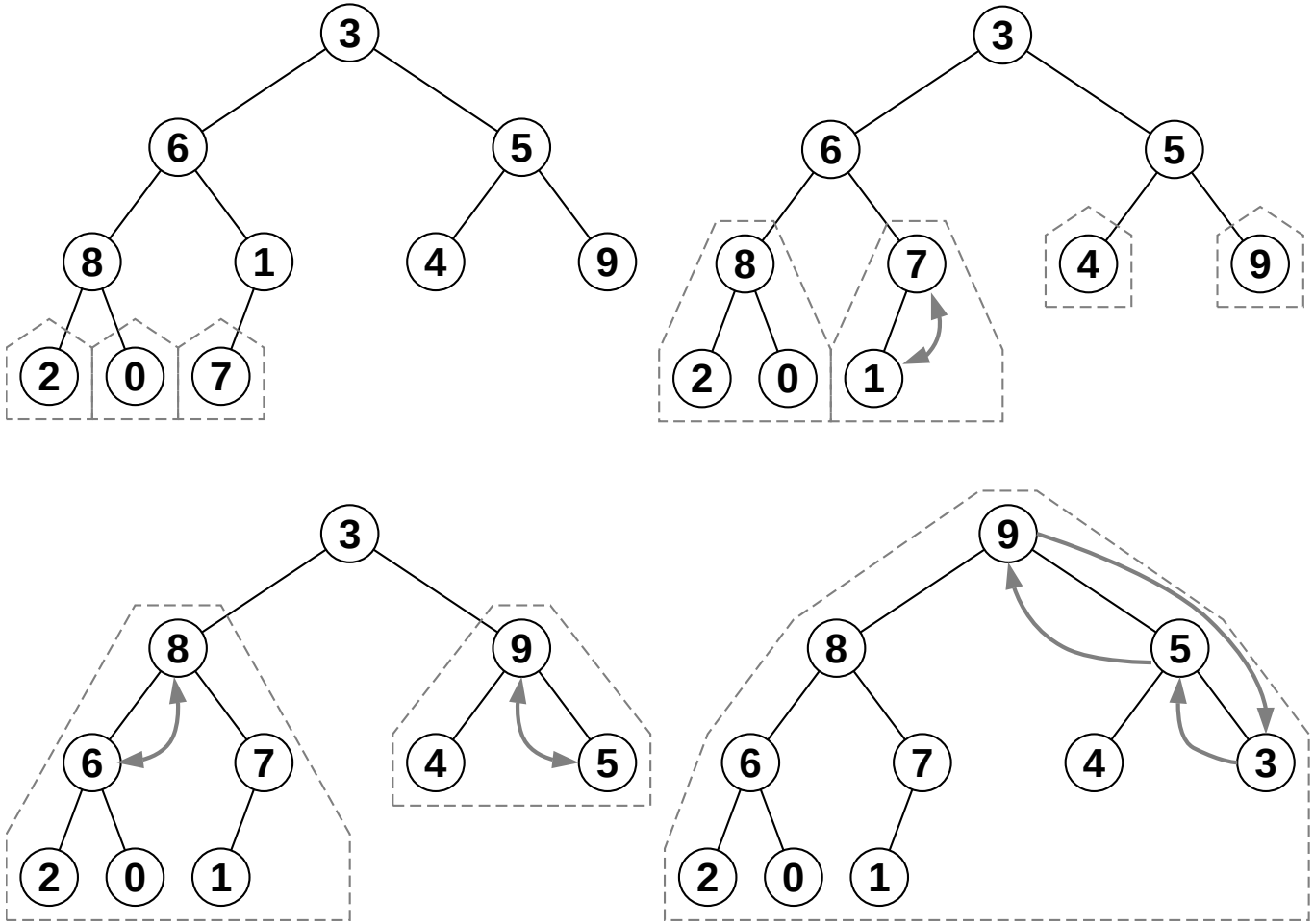
Mit geeigneten (kleinen) Modifikationen kann HeapSort in situ implementiert werden. Die Schichten des Heaps werden dabei von oben nach unten und von links nach rechts im Feld A abgespeichert.

In der ersten Phase von HeapSort müssen wir mit den gegebenen n Schlüsseln einen Heap erzeugen.

Wir tun dies iterativ, indem wir aus zwei bereits erzeugten Heaps und einem weiteren Schlüssel einen neuen Heap formen, indem wir einen neuen Knoten erzeugen, der die Wurzel des neuen Heaps wird. Diesem neuen Knoten ordnen wir zunächst den zusätzlichen Schlüssel zu und und machen die beiden alten Heaps zu seinen Unterbäumen. Damit ist die Heap-Bedingung höchstens an der Wurzel des neuen Baums verletzt, was wir durch Ausführung der Reheap-Operation korrigieren können.

Beispiel 143

[Initialisierung des Heaps]



Lemma 144

Die Reheap-Operation erfordert höchstens $O($Tiefe des Heaps$)$ Schritte.

Beweis:

Reheap führt pro Schicht des Heaps nur konstant viele Schritte aus.

Lemma 145

Die Initialisierung des Heaps in der ersten Phase von HeapSort *benötigt nur $O(n)$ Schritte.*

Beweis:

Sei d die Tiefe (Anzahl der Schichten) des (n-elementigen) Heaps. Die Anzahl der Knoten in Tiefe i ist $\leq 2^i$ (die Wurzel habe Tiefe 0). Wenn ein solcher Knoten beim inkrementellen Aufbau des Heaps als Wurzel hinzugefügt wird, erfordert die Reheap-Operation $\leq d - i$ Schritte, insgesamt werden also

$$\leq \sum_{i=0}^{d-1} (d-i)2^i = O(n)$$

Schritte benötigt.

Satz 146

HeapSort *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt.

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n \operatorname{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HeapSort beschrieben, die mit $n \operatorname{ld} n + O(n \log \log n)$ Vergleichen auskommt:

   Svante Carlsson:
   A variant of heapsort with almost optimal number of comparisons.
   Inf. Process. Lett., **24**(4):247–250, 1987

2.6 Vergleichsbasierte Sortierverfahren

Alle bisher betrachteten Sortierverfahren sind vergleichsbasiert, d.h. sie greifen auf Schlüssel k, k' (außer in Zuweisungen) nur in Vergleichsoperationen der Form $k < k'$ zu, verwenden aber nicht, dass etwa $k \in \mathbb{N}_0$ oder dass $k \in \Sigma^*$.

Satz 147

Jedes vergleichsbasierte Sortierverfahren benötigt im worst-case mindestens

$$n \operatorname{ld} n + O(n)$$

Vergleiche und hat damit Laufzeit $\Omega(n \log n)$.

Beweis:

Wir benutzen ein so genanntes Gegenspielerargument (engl. adversary argument). Soll der Algorithmus n Schlüssel sortieren, legt der Gegenspieler den Wert eines jeden Schlüssels immer erst dann fest, wenn der Algorithmus das erste Mal auf ihn in einem Vergleich zugreift. Er bestimmt den Wert des Schlüssels so, dass der Algorithmus möglichst viele Vergleiche durchführen muss.

Am Anfang (vor der ersten Vergleichsoperation des Algorithmus) sind alle $n!$ Sortierungen der Schlüssel möglich, da der Gegenspieler jedem Schlüssel noch einen beliebigen Wert zuweisen kann.

Beweis:

Seien nun induktiv vor einer Vergleichsoperation $A[i] < A[j]$ des Algorithmus noch r Sortierungen der Schlüssel möglich.

Falls der Gegenspieler die Werte der in $A[i]$ bzw. $A[j]$ gespeicherten Schlüssel bereits früher festgelegt hat, ändert sich die Anzahl der möglichen Sortierungen durch den Vergleich nicht, dieser ist redundant.

Beweis:

Andernfalls kann der Gegenspieler einen oder beide Schlüssel so festlegen, dass immer noch mindestens $r/2$ Sortierungen möglich sind (wir verwenden hier, dass die Schlüssel stets paarweise verschieden sind).

Nach k Vergleichen des Algorithmus sind also immer noch $n!/2^k$ Sortierungen möglich. Der Algorithmus muss jedoch Vergleiche ausführen, bis nur noch eine Sortierung möglich ist (die dann die Ausgabe des Sortieralgorithmus darstellt).

Damit

$$\#\text{Vergleiche} \geq \lceil \mathrm{ld}(n!) \rceil = n \,\mathrm{ld}\, n + O(n)$$

mit Hilfe der Stirlingschen Approximation für $n!$. □

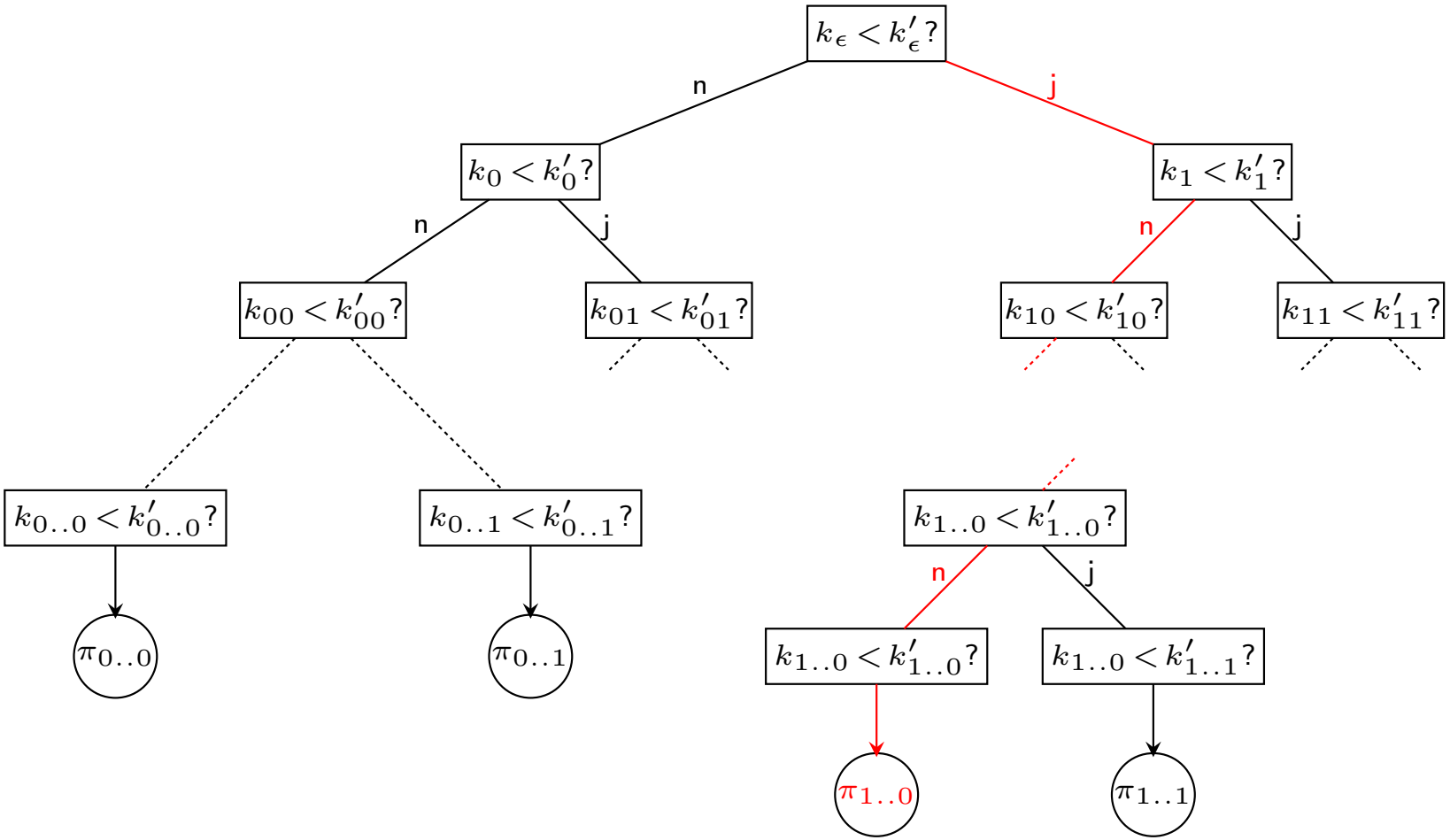
Alternativer Beweis mit Hilfe des Entscheidungsbaums

Ein vergleichsbasierter Algorithmus kann auf die Werte der Schlüssel nur durch Vergleiche $k < k'$ zugreifen. Wenn wir annehmen, dass alle Schlüssel paarweise verschieden sind, ergeben zu jedem Zeitpunkt die vom Algorithmus erhaltenen binären Antworten auf seine Anfragen der obigen Art an die Schlüsselmenge seine gesamte Information über die (tatsächliche) Ordnung der Schlüssel.

Am Ende eines jeden Ablaufs des Algorithmus muss diese Information so sein, dass die tatsächliche Ordnung der Schlüssel eindeutig festliegt, dass also nur eine Permutation der n zu sortierenden Schlüssel mit der erhaltenen Folge von Binärantworten konsistent ist.

Wir stellen alle möglichen Abläufe (d.h., Folgen von Vergleichen), die sich bei der Eingabe von n Schlüsseln ergeben können, in einem so genannten Entscheidungsbaum dar.

Entscheidungsbaum:



Damit muss es für jede der $n!$ Permutationen mindestens ein Blatt in diesem Entscheidungsbaum geben. Da dieser ein Binärbaum ist, folgt daraus:

- Die Tiefe des Entscheidungsbaums (und damit die Anzahl der vom Sortieralgorithmus benötigten Vergleiche im worst-case) ist
$$\geq \lceil \mathrm{ld}(n!) \rceil = n \,\mathrm{ld}\, n + O(n)\,.$$
- Diese untere Schranke gilt sogar im Durchschnitt (über alle Permutationen).

2.7 Bucket-Sort

Bucket-Sort ist ein nicht-vergleichsbasiertes Sortierverfahren. Hier können z.B. n Schlüssel aus

$$\{0, 1, \ldots, B-1\}^d$$

in Zeit $O(d(n+B))$ sortiert werden, indem sie zuerst gemäß dem letzten Zeichen auf B Behälter verteilt werden, die so entstandene Teilsortierung dann stabil gemäß dem vorletzten Zeichen auf B Behälter verteilt wird, usw. bis zur ersten Position. Das letzte Zeichen eines jeden Schlüssels wird also als das niedrigwertigste aufgefasst.

Bucket-Sort sortiert damit n Schlüssel aus $\{0, 1, \ldots, B-1\}^d$ in Zeit $O(nd)$, also linear in der Länge der Eingabe (gemessen als Anzahl der Zeichen).

3. Suchverfahren

Es ist eine Menge von Datensätzen gegeben, wobei jeder Datensatz D_i durch einen eindeutigen Schlüssel k_i gekennzeichnet ist. Der Zugriff zu den Datensätzen erfolgt per Zeiger über die zugeordneten Schlüssel, so dass wir uns nur mit der Verwaltung der Schlüssel beschäftigen.

I. A. ist die Menge der Datensätze dynamisch, d.h. es können Datensätze neu hinzukommen oder gelöscht werden, oder Datensätze bzw. Schlüssel können geändert werden.

Definition 148

Ein Wörterbuch (engl. dictionary) ist eine Datenstruktur, die folgende Operationen auf einer Menge von Schlüsseln effizient unterstützt:

1. is_member(k): teste, ob der Schlüssel k in der Schlüsselmenge enthalten ist;
2. insert(k): füge den Schlüssel k zur Schlüsselmenge hinzu, falls er noch nicht vorhanden ist;
3. delete(k): entferne den Schlüssel k aus der Schlüsselmenge, falls er dort vorhanden ist.

Es gibt zwei grundsätzlich verschiedene Ansätze, um Wörterbücher zu implementieren:

- Suchbäume
- Hashing (Streuspeicherverfahren)

Wir betrachten zuerst Suchbäume. Wir nehmen an, dass die Schlüssel aus einer total geordneten Menge, dem Universum $\mathcal{U}$ stammen.

Sind die Schlüssel nur in den Blättern des Suchbaums gespeichert, sprechen wir von einem externen Suchbaum, ansonsten von einem internen Suchbaum (wo dann der Einfachheit halber Schlüssel *nur* in den internen Knoten gespeichert werden).

3.1 Binäre/natürliche Suchbäume

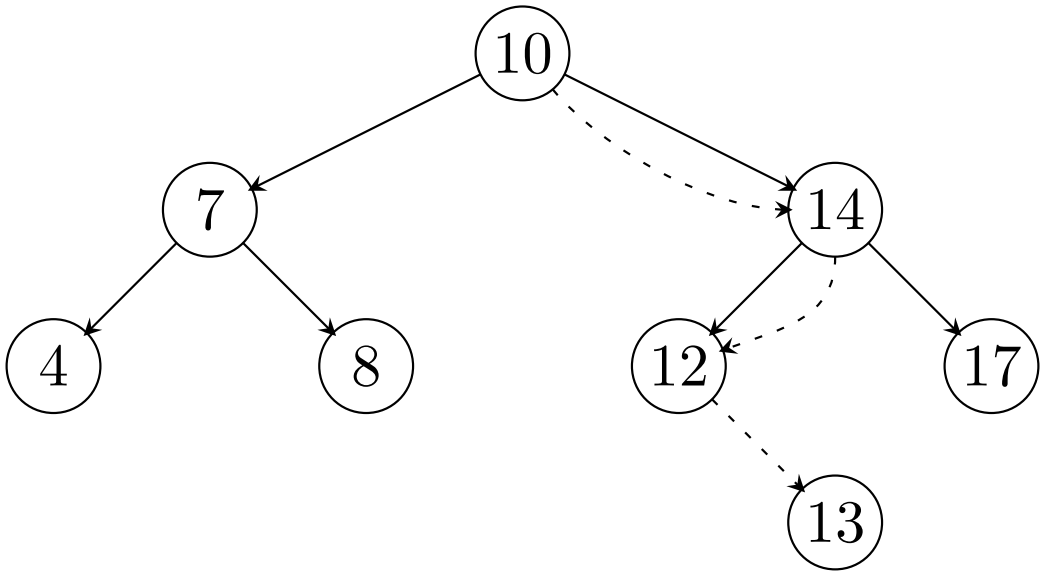
In binären Suchbäumen gilt für alle Knoten x

- $key(x)$ ist größer als der größte Schlüssel im linken Unterbaum von x;
- $key(x)$ ist kleiner als der kleinste Schlüssel im rechten Unterbaum von x.

Die Wörterbuch-Operationen werden wie folgt realisiert:

1. is_member(k): beginnend an der Wurzel des Suchbaums, wird der gesuchte Schlüssel k mit dem am Knoten gespeicherten Schlüssel k' verglichen. Falls $k < k'$ ($k > k'$), wird im linken (rechten) Unterbaum fortgefahren (falls der Unterbaum leer ist, ist der Schlüssel nicht vorhanden), ansonsten ist der Schlüssel gefunden.
2. insert(k): es wird zuerst geprüft, ob k bereits im Suchbaum gespeichert ist; falls ja, ist die Operation beendet, falls nein, liefert die Suche die Position (den leeren Unterbaum), wo k hinzugefügt wird.
3. delete(k): es wird zuerst geprüft, ob k im Suchbaum gespeichert ist; falls nein, ist die Operation beendet, falls ja, sei x der Knoten, in dem k gespeichert ist, und es sei x' der linkste Knoten im rechten Unterbaum von x (x' ist nicht unbedingt ein Blatt!); dann wird $key(x')$ im Knoten x gespeichert und x' durch seinen rechten Unterbaum ersetzt (falls vorhanden) bzw. gelöscht. Spezialfälle, wie z.B., dass x' nicht existiert, sind kanonisch zu behandeln.

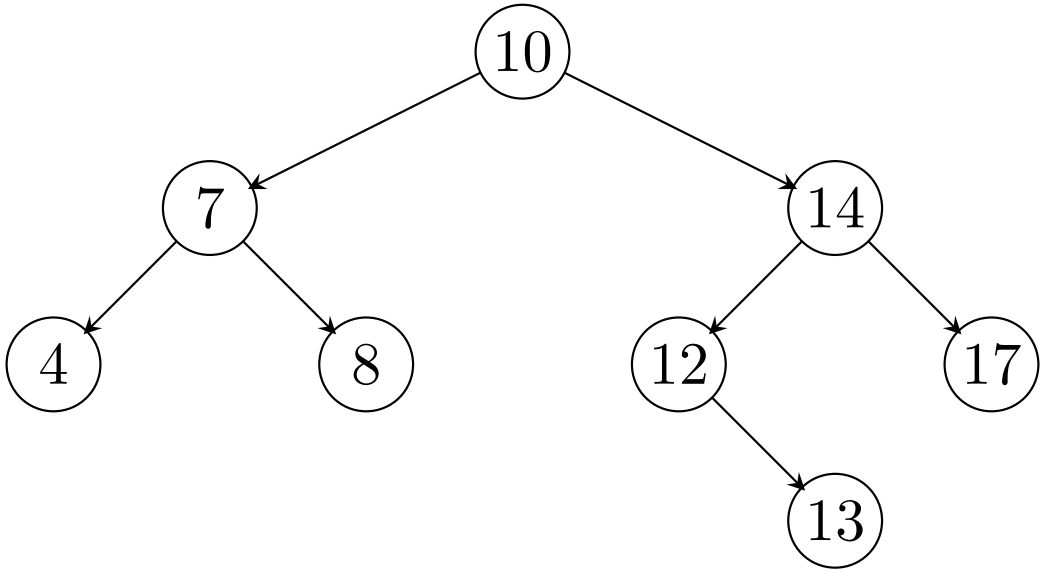
Beispiel 149



Der Schlüssel 13 wird hinzugefügt.

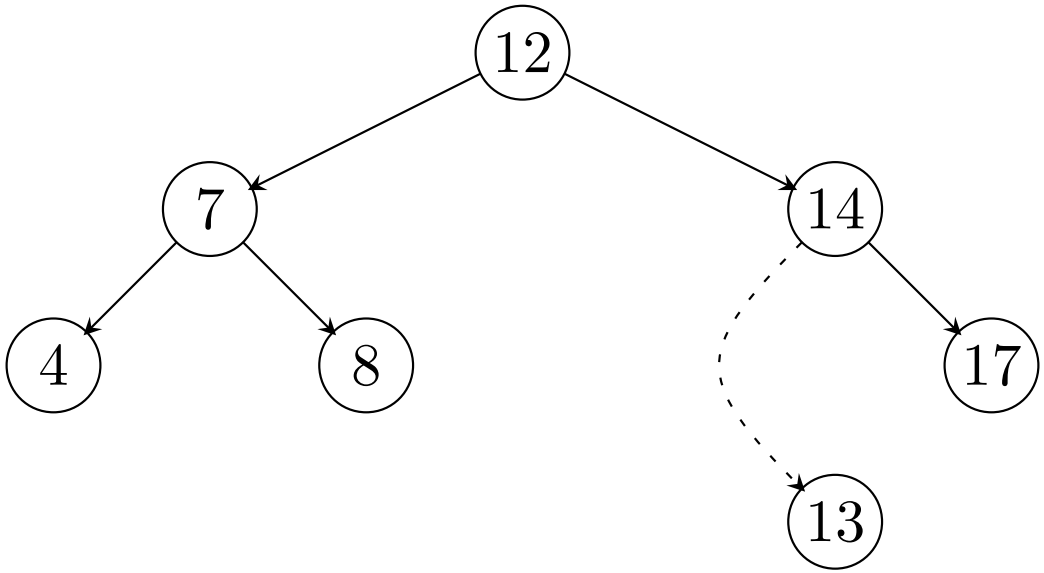
Beispiel

Der Schlüssel 10 (also die Wurzel) wird gelöscht:



Beispiel

Der Schlüssel 10 (also die Wurzel) wird gelöscht:



Satz 150

In einem natürlichen Suchbaum der Höhe h benötigen die Wörterbuch-Operationen jeweils Zeit $O(h)$.

Beweis:

Folgt aus der obigen Konstruktion!

Bemerkung:

Das Problem bei natürlichen Suchbäumen ist, dass sie sehr entartet sein können, z.B. bei n Schlüsseln eine Tiefe von $n-1$ haben. Dies ergibt sich z.B., wenn die Schlüssel in aufsteigender Reihenfolge eingefügt werden.

3.2 AVL-Bäume

AVL-Bäume sind interne binäre Suchbäume, bei denen für jeden Knoten gilt, dass sich die Höhe seiner beiden Unterbäume um höchstens 1 unterscheidet. AVL-Bäume sind nach ihren Erfindern G. Adelson-Velskii und Y. Landis (1962) benannt.

Satz 151

Ein AVL-Baum der Höhe h enthält mindestens $F_{h+3} - 1$ und höchstens $2^{h+1} - 1$ Knoten, wobei F_n die n-te Fibonacci-Zahl ($F_0 = 0, F_1 = 1$) und die Höhe die maximale Anzahl von Kanten auf einem Pfad von der Wurzel zu einem Blatt ist.

Beweis:

Die obere Schranke ist klar, da ein Binärbaum der Höhe h höchstens

$$\sum_{j=0}^{h} 2^j = 2^{h+1} - 1$$

Knoten enthalten kann.

Beweis:

Induktionsanfang:

1. ein AVL-Baum der Höhe $h = 0$ enthält mindestens einen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$
2. ein AVL-Baum der Höhe $h = 1$ enthält mindestens zwei Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$

Beweis:

Induktionsschluss: Ein AVL-Baum der Höhe $h \geq 2$ mit minimaler Knotenzahl hat als Unterbäume der Wurzel einen AVL-Baum der Höhe $h-1$ und einen der Höhe $h-2$, jeweils mit minimaler Knotenzahl. Sei

$$f_h := 1 + \text{minimale Knotenanzahl eines AVL-Baums der Höhe } h\,.$$

Dann gilt demgemäß

$$\begin{aligned} f_0 &= 2 && = F_3 \\ f_1 &= 3 && = F_4 \\ f_h - 1 &= 1 + f_{h-1} - 1 + f_{h-2} - 1\,, && \text{also} \\ f_h &= f_{h-1} + f_{h-2} && = F_{h+3} \end{aligned}$$

□

Korollar 152

Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

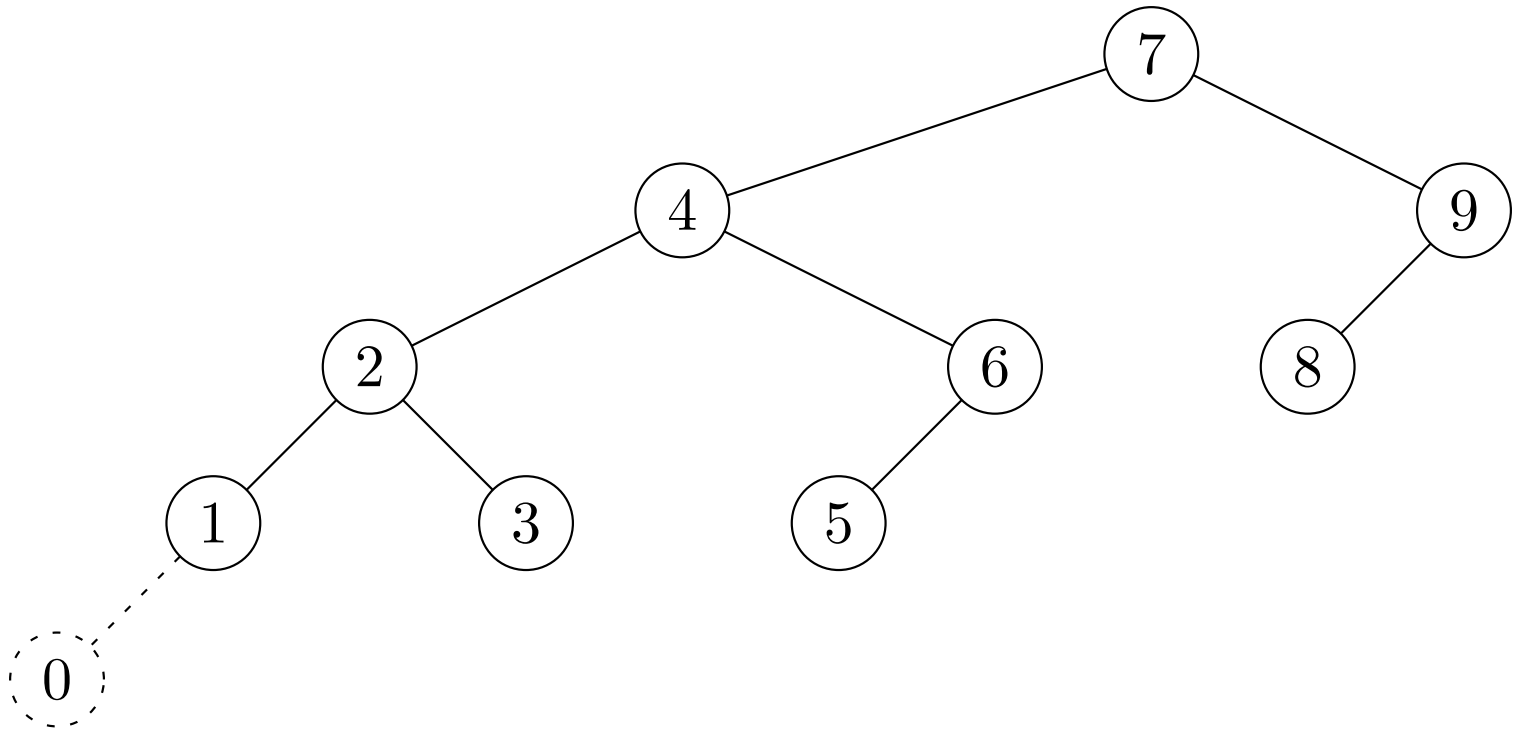
Satz 153

In einem AVL-Baum mit n Schlüsseln kann in Zeit $O(\log n)$ festgestellt werden, ob sich ein gegebener Schlüssel in der Schlüsselmenge befindet oder nicht.

Beweis:

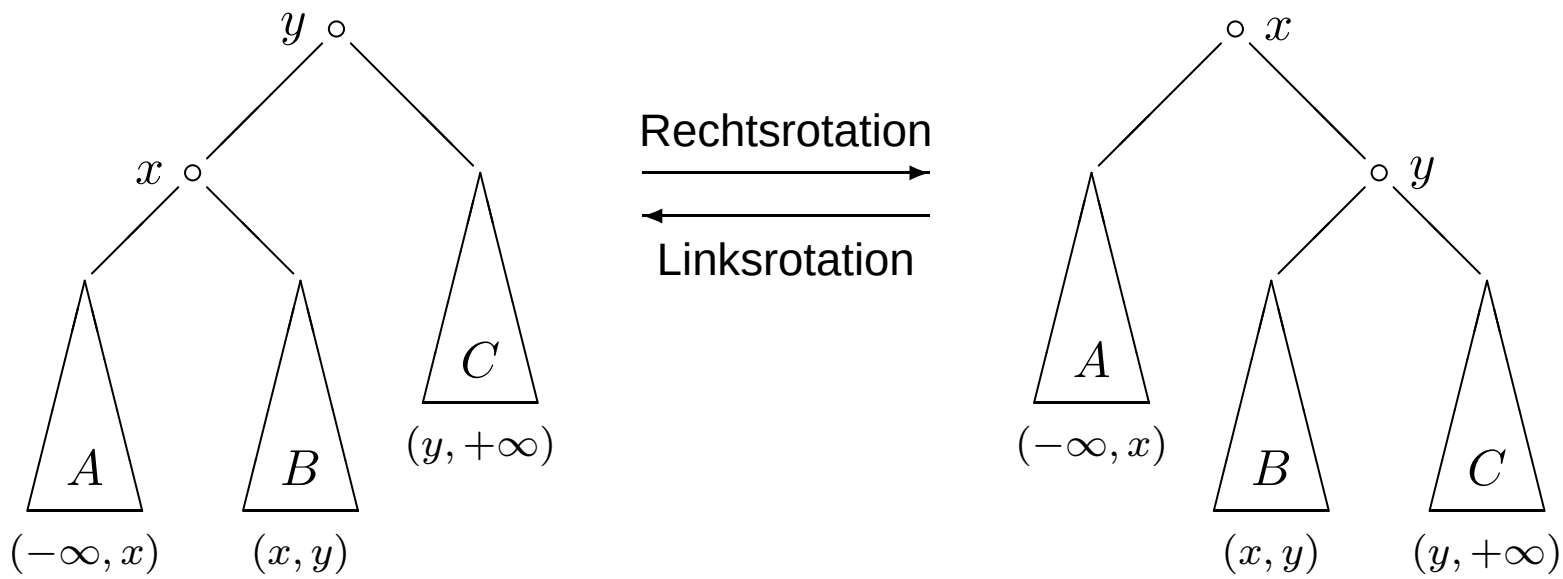
Klar!

Beispiel 154 (Einfügen eines Knotens in AVL-Baum)

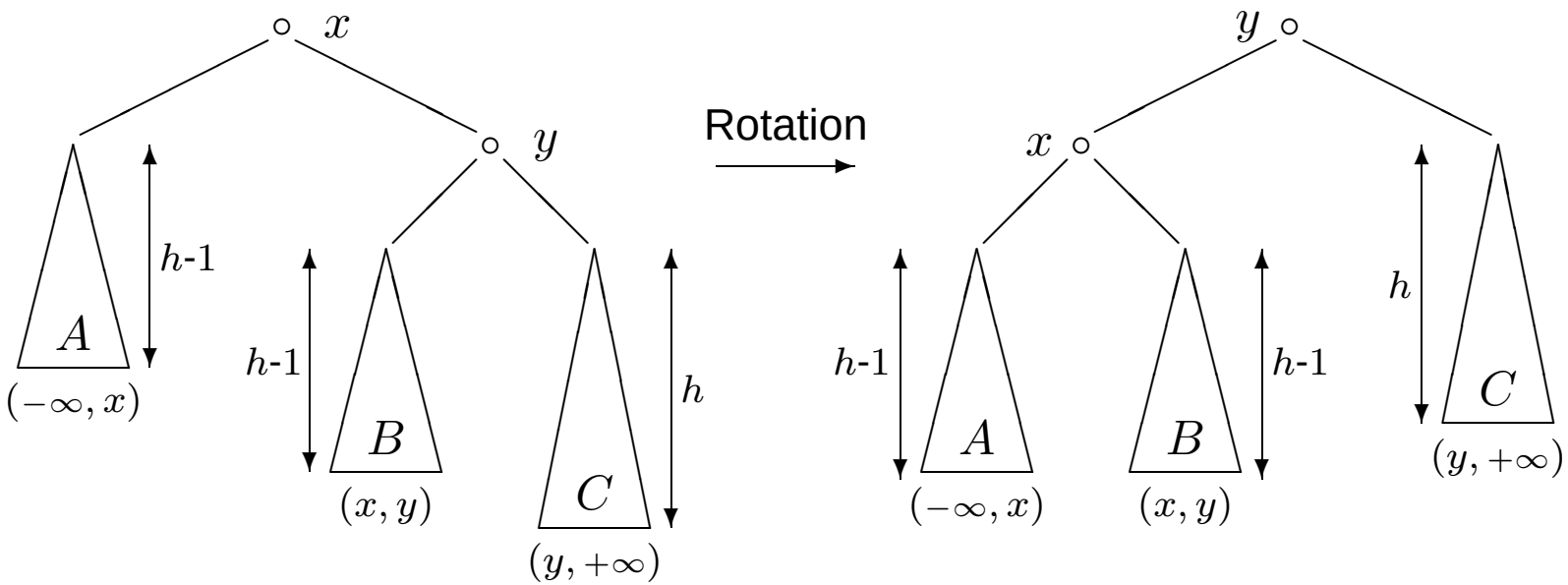


Zur Wiederherstellung der Höhenbedingung benutzen wir so genannte Rotationen und Doppelrotationen.

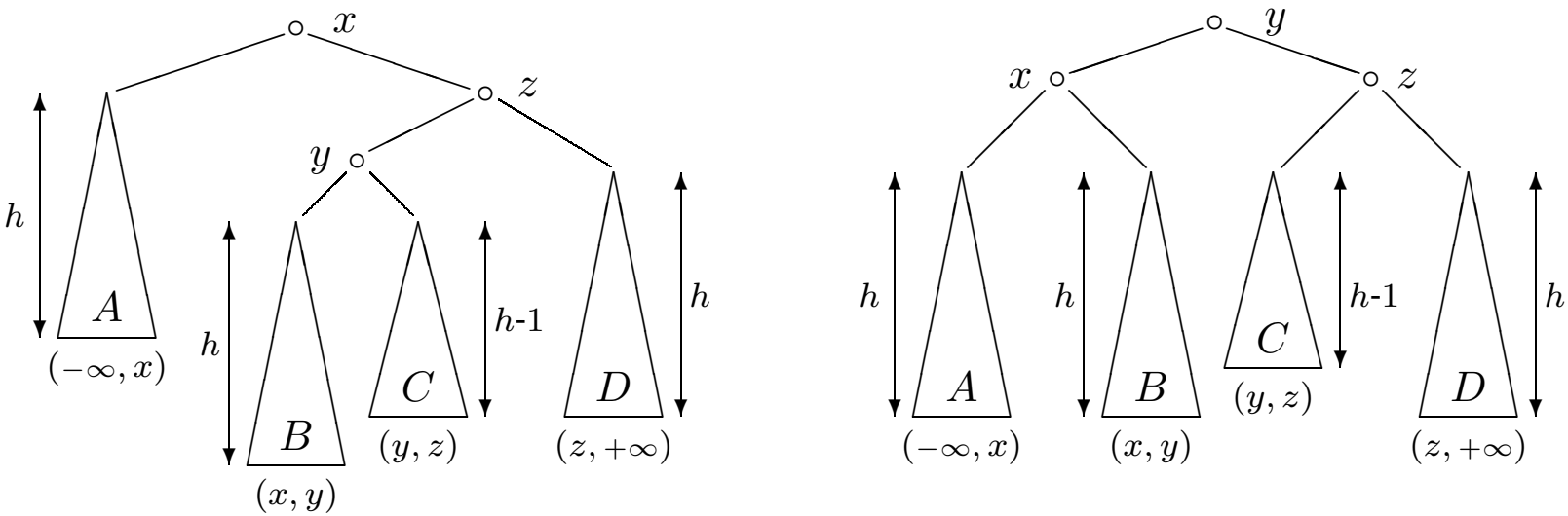
Beispiel 155 (Rotation um (x, y))



Beispiel 156 (Wiederherstellung der Höhenbedingung)



Beispiel 157 (Doppelrotation zur Rebalancierung)



Zur Rebalancierung des AVL-Baums sind Rotationen und Doppelrotationen nur entlang des Pfades zum eingefügten Knoten erforderlich. Damit ergibt sich

Lemma 158

In einen AVL-Baum mit n Knoten kann ein neuer Schlüssel in Zeit $O(\log n)$ eingefügt werden.

Ebenso kann man zeigen

Lemma 159

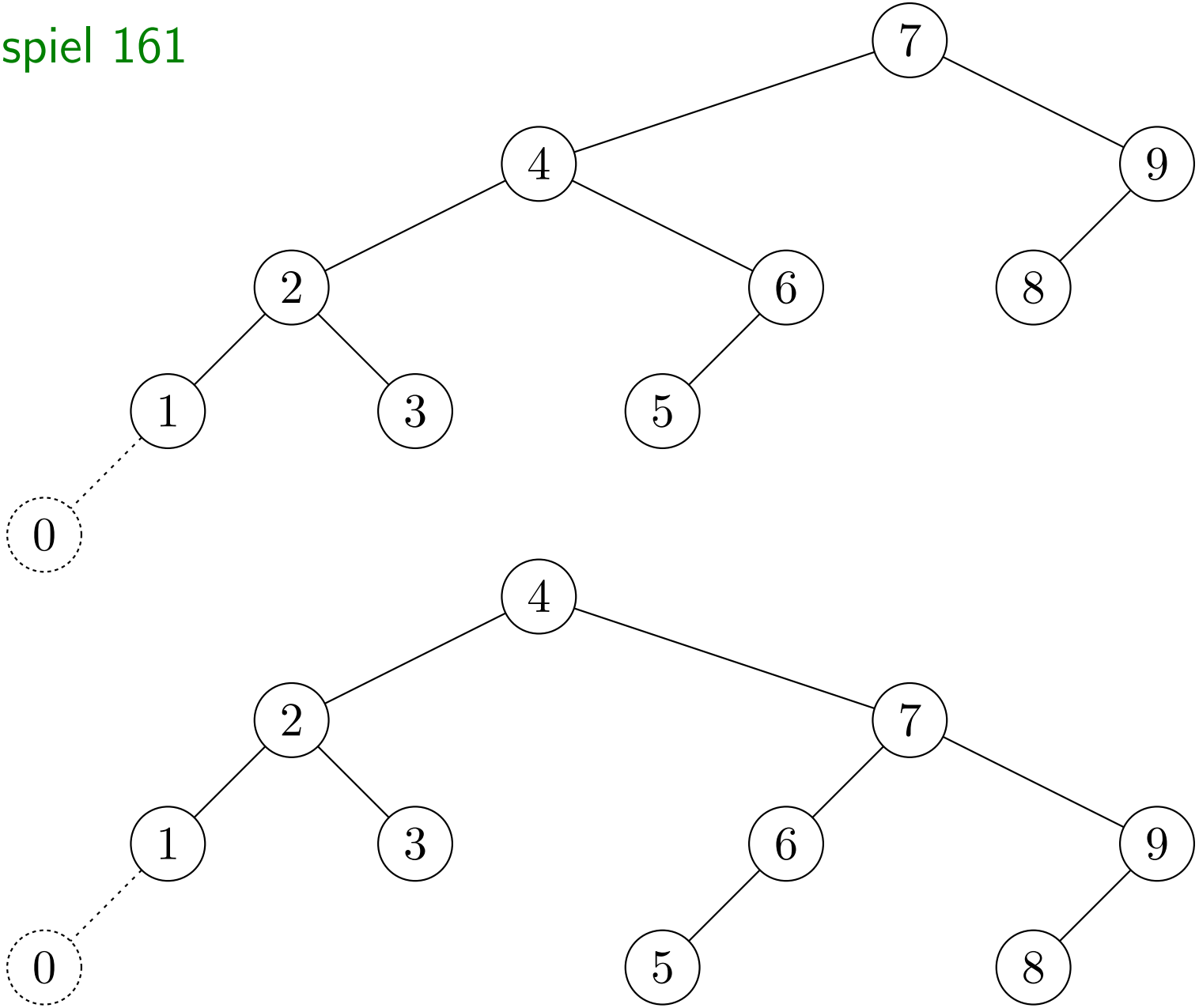
In einen AVL-Baum mit n Knoten kann ein im Baum vorhandener Schlüssel in Zeit $O(\log n)$ gelöscht werden.

Damit

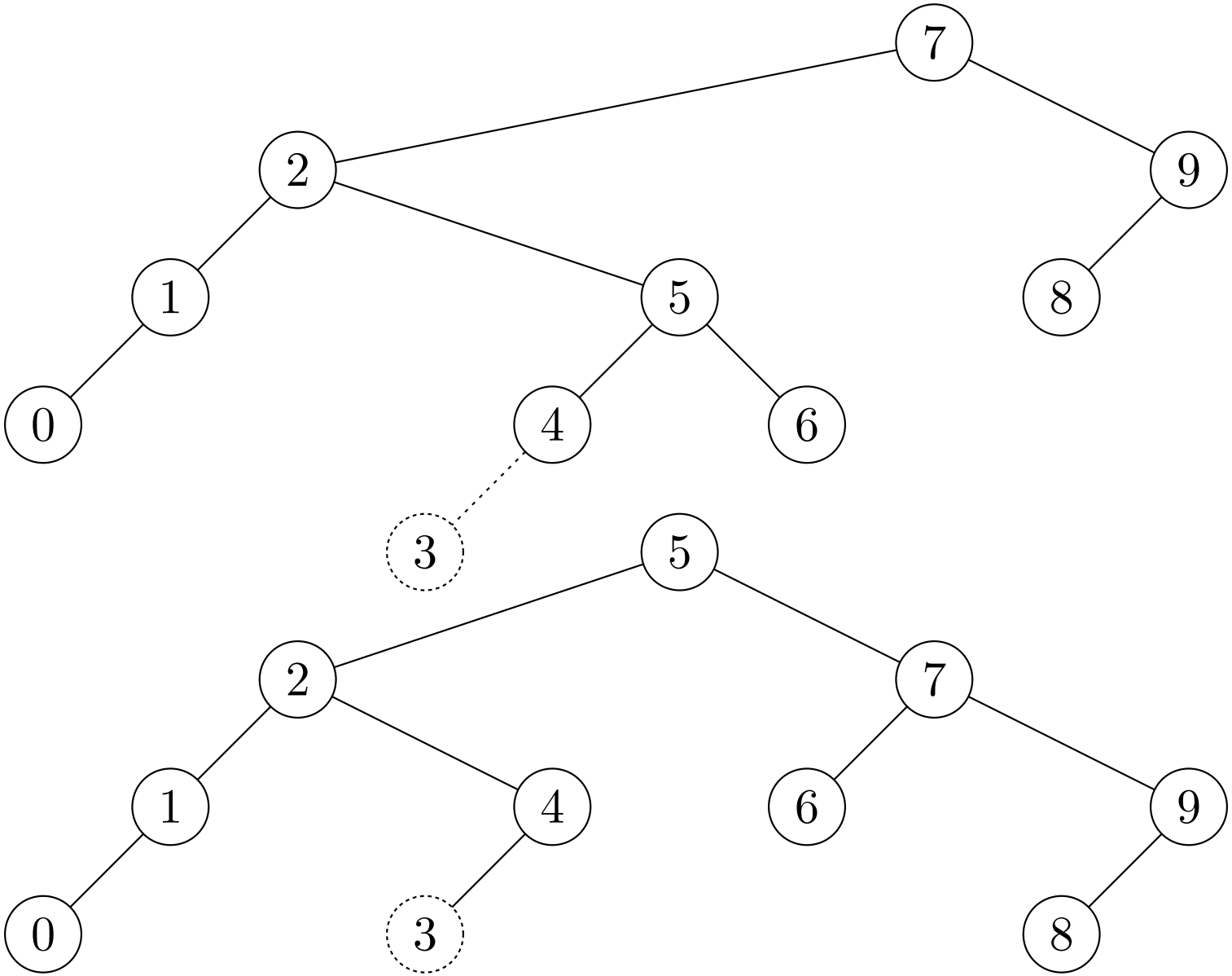
Satz 160

In einem AVL-Baum mit n Knoten kann jede Wörterbuch-Operation in Zeit $O(\log n)$ ausgeführt werden.

Beispiel 161



Beispiel 162 (Rebalancierung mit Doppelrotation)



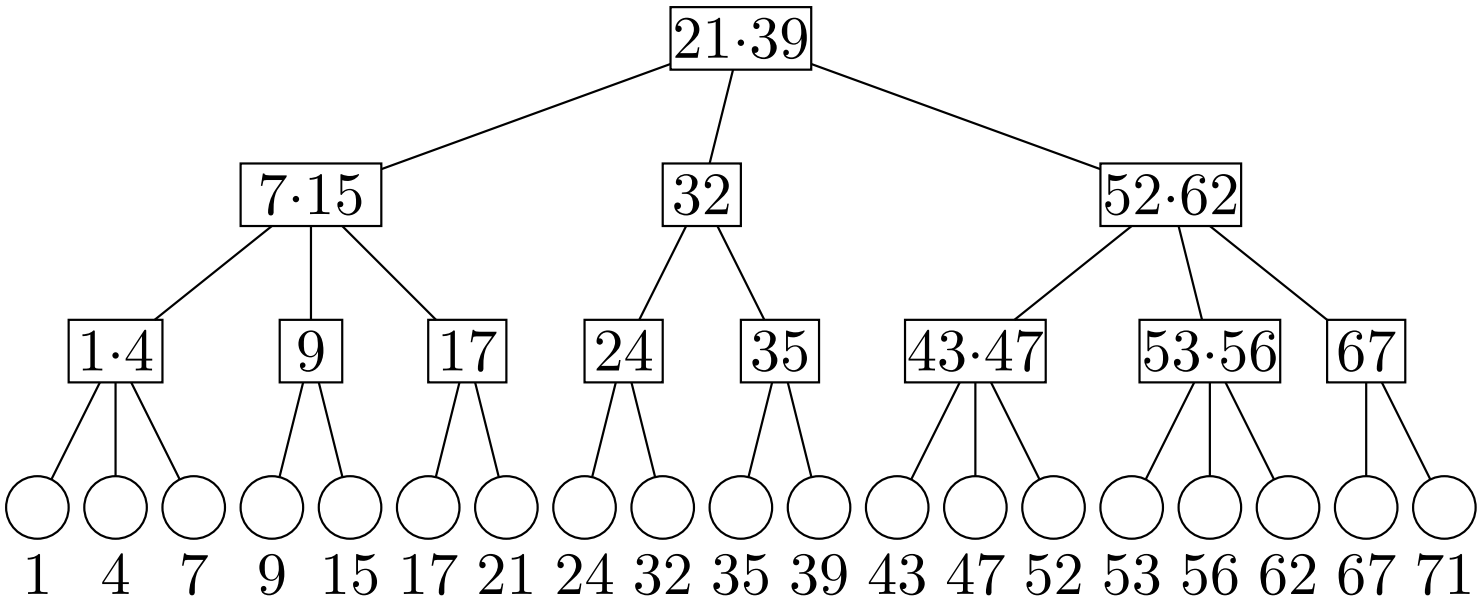
3.3 (a, b)-Bäume

Definition 163

Ein (a, b)-Baum ist ein externer Suchbaum, für den gilt:

1. alle Blätter haben die gleiche Tiefe
2. alle internen Knoten haben $\leq b$ Kinder
3. alle internen Knoten außer der Wurzel haben $\geq a$, die Wurzel hat ≥ 2 Kinder
4. $b \geq 2a - 1$
5. in jedem internen Knoten sind jeweils die größten Schlüssel seiner Unterbäume mit Ausnahme des letzten gespeichert

Beispiel 164

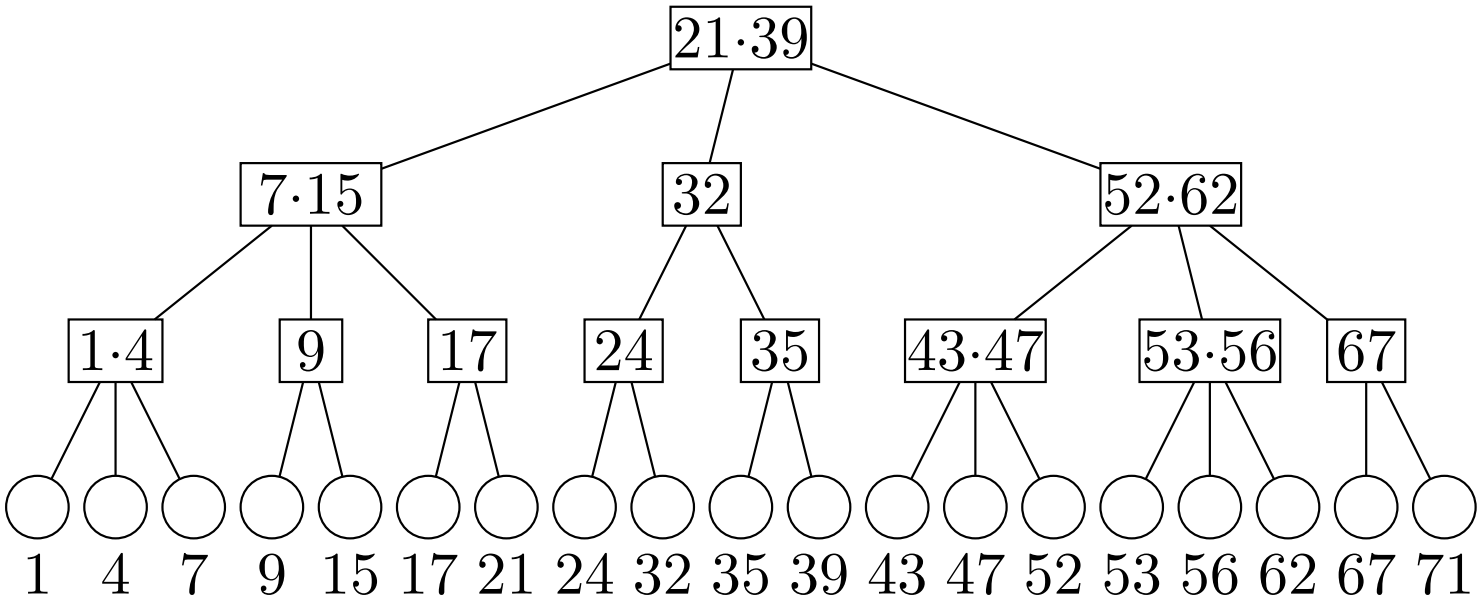


Bemerkung:
(a, b)-Bäume mit $b = 2a - 1$ heißen auch B-Bäume. Diese wurden erstmals in einer Arbeit von R. Bayer und E.M. McCreight im Jahr 1970 beschrieben. (2,3)-Bäume wurden von J. Hopcroft ebenfalls 1970 eingeführt.

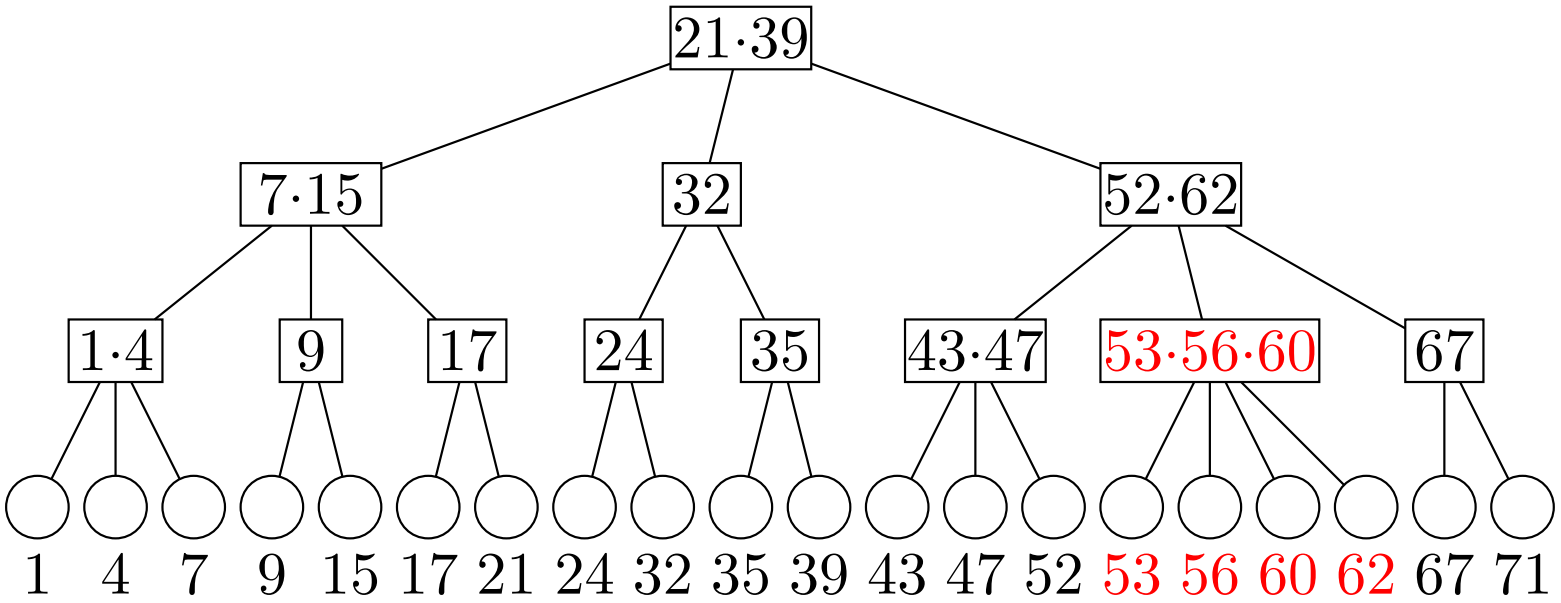
Insert-Operation: Übersteigt durch eine Insert-Operation ein Knoten die Anzahl der zulässigen Kinder, so wird er in zwei Knoten geteilt.

Delete-Operation: Fällt durch eine Delete-Operation die Anzahl der Kinder eines Knoten unter a, so wird ein Kind vom linken oder rechten Geschwister des Knoten adoptiert.

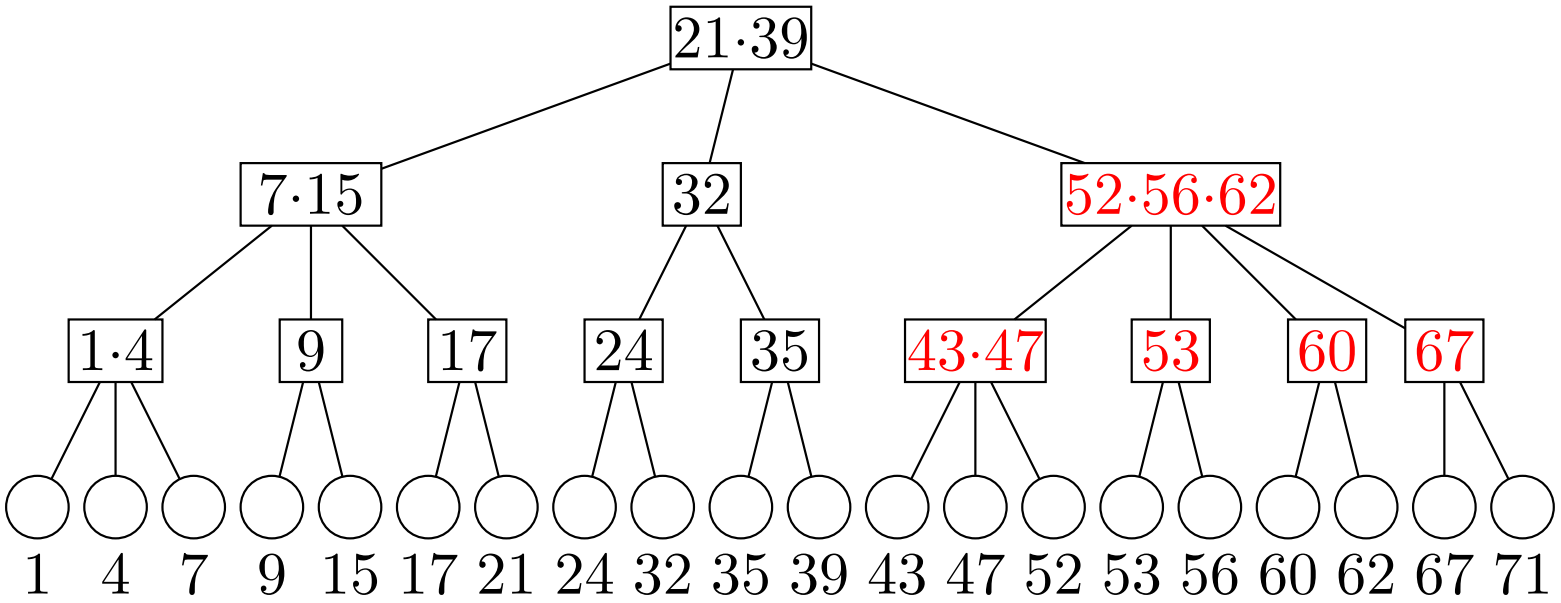
Beispiel 165 (Füge „60“ in (2,3)-Baum ein)



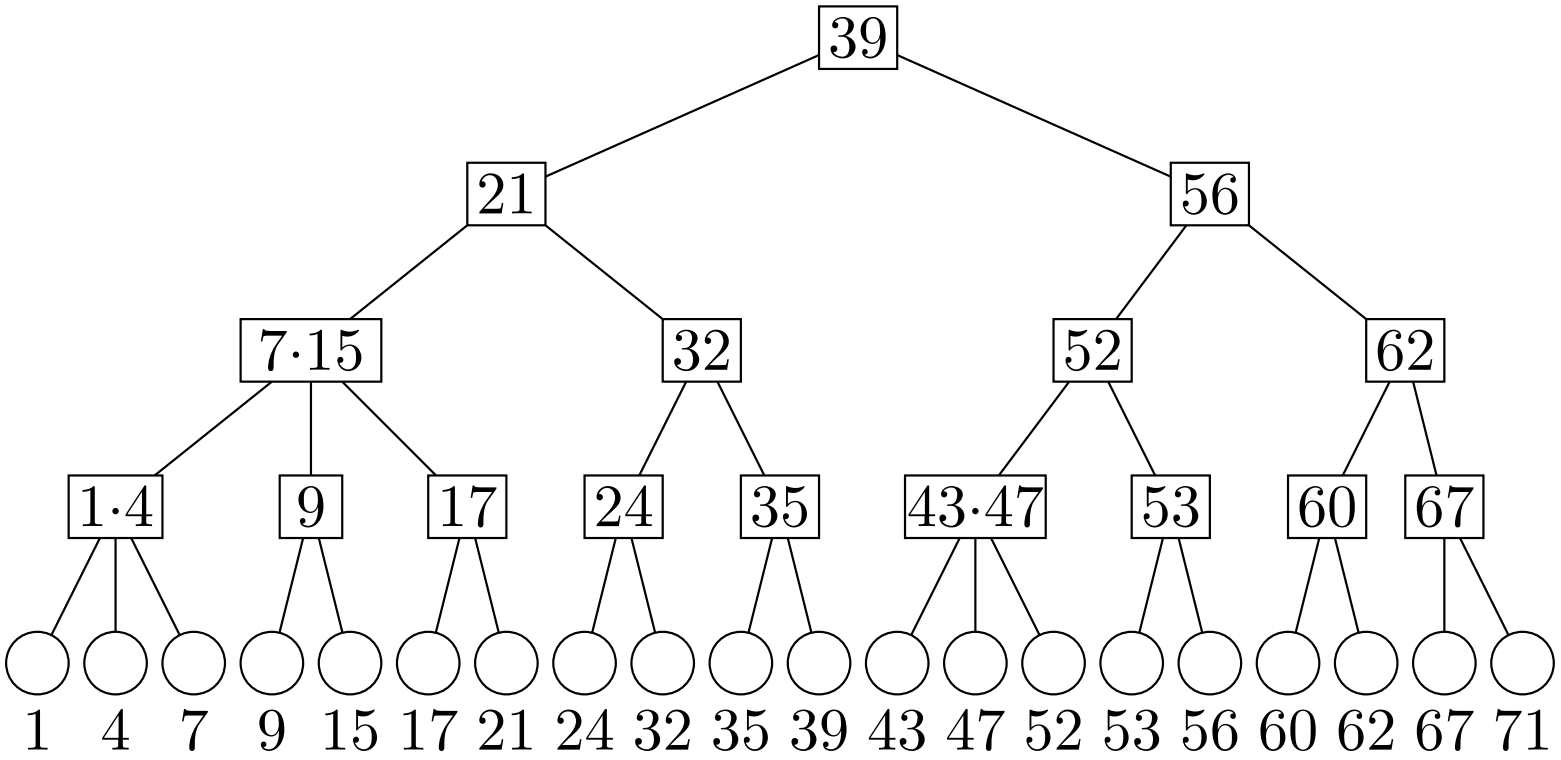
Beispiel 165 (Füge „60“ in (2,3)-Baum ein)



Beispiel 165 (Füge „60“ in (2,3)-Baum ein)



Beispiel 165 (Füge „60“ in (2,3)-Baum ein)



Korollar 166

In einem (a, b)-Baum mit n gespeicherten Schlüsseln können die Wörterbuchoperationen in Zeit $O(\log_a n)$ durchgeführt werden.

Bemerkung: Die Wahl von a und b hängt wesentlich von der Anwendung und der Größe des (a, b)-Baums ab.

- Liegt der (a, b)-Baum im RAM, dann wählt man b klein, um in jedem inneren Knoten den richtigen Unterbaum schnell zu finden.
- Andernfalls wählt man a groß. Der Baum hat dadurch nur geringe Tiefe, seine oberen Schichten können im RAM gehalten werden, und die Anzahl der Zugriffe auf den Sekundärspeicher (Festplatte) ist klein.

3.4 Hashing

Hash- oder auch Streuspeicherverfahren sind dadurch gekennzeichnet, dass jedem Schlüssel aus einem Universum U durch eine (effizient berechenbare) Funktion h (die Hashfunktion) eine Adresse $\in \{0, \ldots, m-1\}$ zugeordnet wird. Der Schlüssel k wird dann im Prinzip im Element $A[h(k)]$ des m-elementigen Feldes $A[0..m-1]$ gespeichert.

Im Idealfall sind dann die Wörterbuchoperationen effizient ausführbar, bestimmt durch den Aufwand für die Berechnung von $h(k)$.

Da $|U| \gg m$, kann h natürlich nicht injektiv sein, und es kommt zu Kollisionen:

$$h(x) = h(y) \text{ für gewisse } x \neq y\,.$$

Normalerweise sind wir nur an einer kleineren Teilmenge $S \subset U$ von Schlüsseln interessiert ($|S| = n$), und wir haben, der Speichereffizienz halber, z.B.

$$m \in \{n, \ldots, 2n\}$$

Selbst in diesem Fall sind Kollisionen jedoch zu erwarten, wie z.B. das so genannte Geburtstagsparadoxon zeigt:

In einer Menge von mindestens 23 zufällig gewählten Personen gibt es mit Wahrscheinlichkeit größer als $\frac{1}{2}$ zwei Personen, die am gleichen Tag des Jahres Geburtstag haben (Schaltjahre werden hier nicht berücksichtigt).

Das Geburtstagsparadoxon ist äquivalent dazu, dass in einer Hashtabelle der Größe 365, die mindestens 23 Einträge enthält, wobei die Hashadressen sogar zufällig gewählt sind, bereits mit einer Wahrscheinlichkeit von mehr als $\frac{1}{2}$ eine Kollision vorkommt.

Satz 167

In einer Hashtabelle der Größe m mit n Einträgen tritt mit einer Wahrscheinlichkeit von mindestens $1 - e^{-n(n-1)/(2m)}$ mindestens eine Kollision auf, wenn für jeden Schlüssel die Hashadresse gleichverteilt ist.

Beweis:

Die Wahrscheinlichkeit, dass auch der i-te eingefügte Schlüssel keine Kollision verursacht, ist

$$\frac{m-(i-1)}{m}.$$

Die Wahrscheinlichkeit, dass für alle n Schlüssel keine Kollision eintritt, ist daher

$$\Pr\{\text{keine Kollision}\} = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \exp\left(\sum_{i=0}^{n-1} \ln\left(1-\frac{i}{m}\right)\right)$$

Satz 167

In einer Hashtabelle der Größe m mit n Einträgen tritt mit einer Wahrscheinlichkeit von mindestens $1 - e^{-n(n-1)/(2m)}$ mindestens eine Kollision auf, wenn für jeden Schlüssel die Hashadresse gleichverteilt ist.

Beweis:

Da $\ln(1+x) \leq x$ für alle $x \in \mathbb{R}$, folgt damit

$$\begin{aligned}
\Pr\{\text{keine Kollision}\} &= \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right) \\
&\leq \exp\left(\sum_{i=0}^{n-1} \left(-\frac{i}{m}\right)\right) \\
&= \exp\left(-\frac{n(n-1)}{2m}\right) .
\end{aligned}$$

□

Korollar 168

Hat eine Hashtabelle der Größe m mindestens $\omega(\sqrt{m})$ Einträge, wobei für jeden Schlüssel die Hashadressen gleichverteilt sind, so tritt mit Wahrscheinlichkeit $1 - o(1)$ eine Kollision auf.

Definition 169

Eine Hashfunktion h heißt ideal, wenn gilt

$$(\forall i \in \{0, \ldots, m-1\}) \left[\sum_{\substack{k \in U \\ h(k)=i}} \Pr\{k\} = \frac{1}{m} \right] ,$$

wobei $\Pr\{k\}$ die Wahrscheinlichkeit ist, mit der der Schlüssel $k \in U$ vorkommt.

Beispiele für gebräuchliche Hashfunktionen:

1. **Additionsmethode:**

$$h(k) = \left(\sum_{i=0}^{r} \left((k \bmod 2^{\alpha_i}) \operatorname{div} 2^{\beta_i} \right) \right) \bmod m$$

für $\alpha_i \geq \beta_i \in \{0, \ldots, \ell(|U|) - 1\}$

2. **Multiplikationsmethode:** Sei $a \in (0, 1)$ eine Konstante.

$$h(k) = \lfloor m \cdot (ka - \lfloor ka \rfloor) \rfloor$$

Eine gute Wahl für a ist z.B.

$$a = \frac{\sqrt{5} - 1}{2} = \frac{1}{\varphi} \approx 0,618\ldots,$$

wobei $\varphi = \frac{1+\sqrt{5}}{2}$ der Goldene Schnitt ist.

Beispiele für gebräuchliche Hashfunktionen:

3. **Teilermethode:**

$$h(k) = k \bmod m\,,$$

wobei m keine Zweier- oder Zehnerpotenz und auch nicht von der Form $2^i - 1$ sein sollte. Eine gute Wahl für m ist i.A. eine Primzahl, die nicht in der Nähe einer Zweierpotenz liegt.

3.4.1 Kollisionsauflösung

Hashing durch Verkettung

In jedem Feldelement der Hashtabelle wird eine lineare Liste der Schlüssel abgespeichert, die durch die Hashfunktion auf dieses Feldelement abgebildet werden. Die Implementierung der Operationen is_member, insert und delete ist offensichtlich.

Sei $\alpha = \frac{n}{m}$ der Füllgrad der Hashtabelle. Dann beträgt, bei Gleichverteilung der Schlüssel, die Länge einer jeden der linearen Listen im Erwartungswert α und der Zeitaufwand für die Suche nach einem Schlüssel im

$$\text{erfolgreichen Fall } 1 + \frac{\alpha}{2}$$

$$\text{erfolglosen Fall } 1 + \alpha$$

Für die Operationen insert und delete ist darüber hinaus lediglich ein Zeitaufwand von $O(1)$ erforderlich.

Hashing durch Verkettung

In jedem Feldelement der Hashtabelle wird eine lineare Liste der Schlüssel abgespeichert, die durch die Hashfunktion auf dieses Feldelement abgebildet werden. Die Implementierung der Operationen is_member, insert und delete ist offensichtlich.

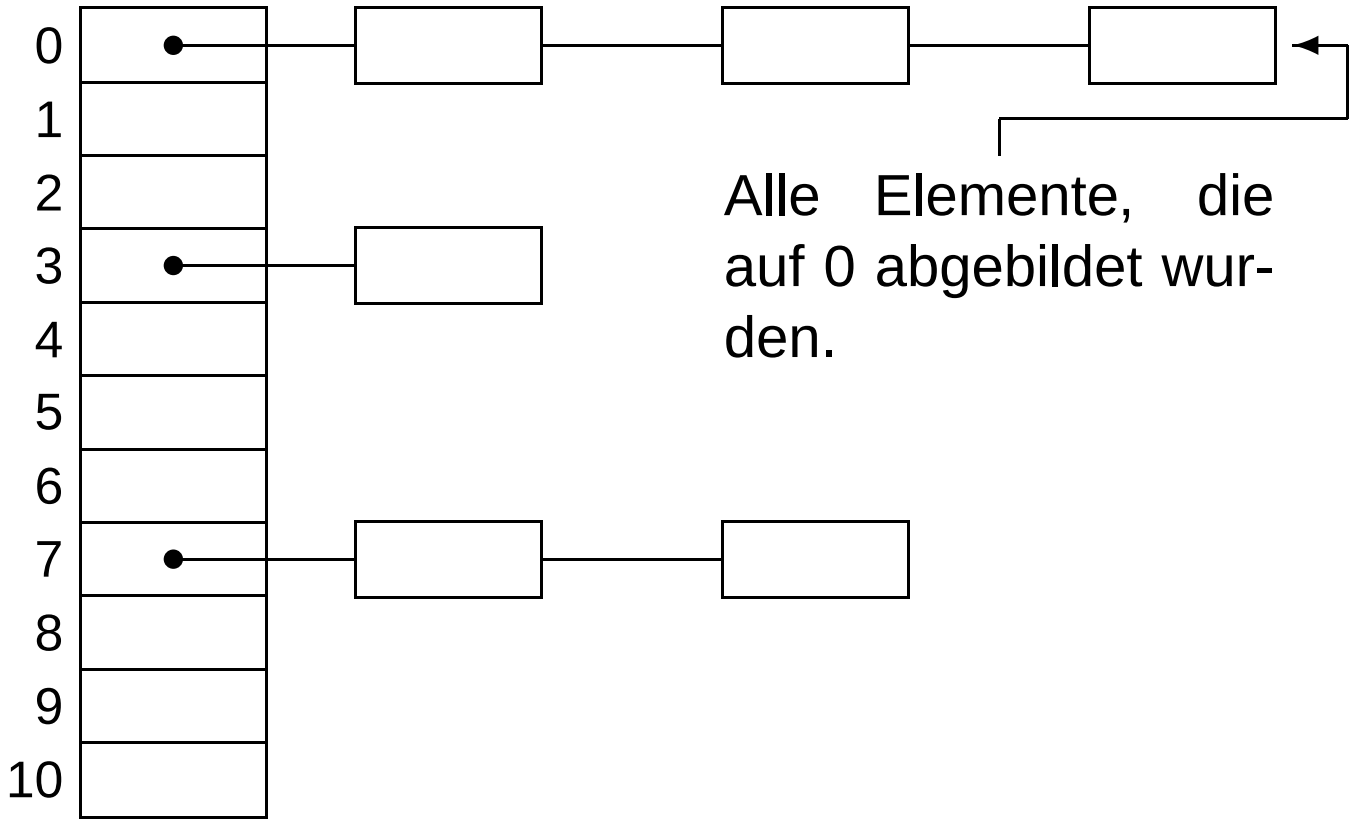
Sei $\alpha = \frac{n}{m}$ der Füllgrad der Hashtabelle. Dann beträgt, bei Gleichverteilung der Schlüssel, die Länge einer jeden der linearen Listen im Erwartungswert α und der Zeitaufwand für die Suche nach einem Schlüssel im

$$\text{erfolgreichen Fall } 1 + \frac{\alpha}{2}$$

$$\text{erfolglosen Fall } 1 + \alpha$$

Hashing durch Verkettung (engl. chaining) ist ein Beispiel für ein geschlossenes Hashverfahren.

Beispiel 170 (Hashing durch Verkettung)



Lineare Sondierung

Man führt eine *erweiterte Hashfunktion* $h(k, i)$ ein. Soll der Schlüssel k neu in die Hashtabelle eingefügt werden, wird die Folge $h(k, 0), h(k, 1), h(k, 2), \ldots$ durchlaufen, bis die erste freie Position in der Tabelle gefunden wird.

Beim Verfahren des linearen Sondierens (engl. linear probing) ist

$$h(k, i) = (h(k) + i) \pmod{m}.$$

Für den Zeitaufwand für die Suche nach einem Schlüssel kann man zeigen im

$$\text{erfolgreichen Fall: } \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$\text{erfolglosen Fall: } \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Lineare Sondierung

Man führt eine *erweiterte Hashfunktion* $h(k, i)$ ein. Soll der Schlüssel k neu in die Hashtabelle eingefügt werden, wird die Folge $h(k, 0), h(k, 1), h(k, 2), \ldots$ durchlaufen, bis die erste freie Position in der Tabelle gefunden wird.

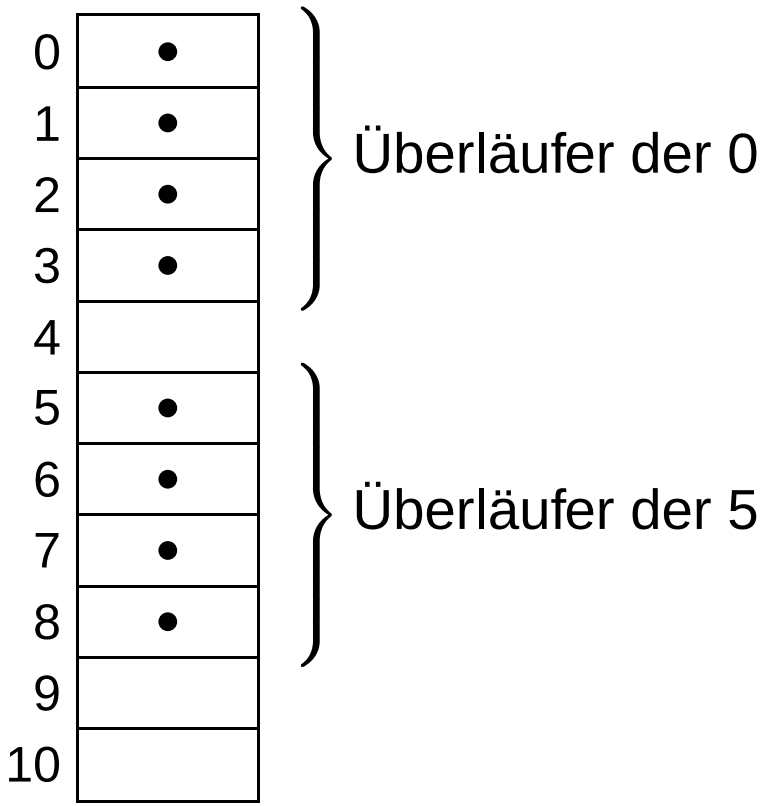
Beim Verfahren des linearen Sondierens (engl. linear probing) ist

$$h(k, i) = (h(k) + i) \pmod{m}.$$

Die Implementierung von insert ist kanonisch, bei delete ist jedoch Sorge zu tragen, dass keine der Sondierungsfolgen $h(k, 0), h(k, 1), h(k, 2), \ldots$ unterbrochen wird. Oft werden gelöschte Elemente daher nicht wirklich entfernt, sondern nur als *gelöscht* markiert.

Ein großes Problem der linearen Sondierung ist, dass sich durch Kollisionen große zusammenhängende Bereiche von belegten Positionen der Hashtabelle ergeben können, wodurch die Suchfolge bei nachfolgenden Einfügungen sehr lang werden kann und auch solche Bereiche immer mehr zusammenwachsen können. Dieses Problem wird als primäre Häufung (engl. primary clustering) bezeichnet.

Beispiel 171 (Lineare Sondierung)



Quadratische Sondierung

Beim Verfahren des quadratischen Sondierens (engl. quadratic probing) ist

$$h(k,i) = (h(k) - (-1)^i(\lceil i/2 \rceil^2)) \pmod m .$$

Surjektivität der Folge $h(k,0), h(k,1), h(k,2), \ldots$ kann garantiert werden, wenn z.B. m prim und $m \equiv 3 \bmod 4$ ist. Für den

Zeitaufwand für die Suche nach einem Schlüssel kann man zeigen im

$$\text{erfolgreichen Fall: } 1 - \frac{\alpha}{2} + \ln\left(\frac{1}{1-\alpha}\right)$$

$$\text{erfolglosen Fall: } 1 + \frac{\alpha^2}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right)$$

Für die Implementierung von delete gilt die gleiche Bemerkung wie beim linearen Sondieren.

Doppeltes Hashing

Beim Verfahren des doppelten Hashings (engl. double hashing) ist

$$h(k,i) = (h(k) + i \cdot h'(k)) \pmod m .$$

Dabei sollte $h'(k)$ für alle k teilerfremd zu m sein, z.B.

$h'(k) = 1 + (k \bmod (m-1))$ oder $h'(k) = 1 + (k \bmod (m-2))$.

Surjektivität der Folge $h(k,0), h(k,1), h(k,2), \ldots$ kann dann garantiert werden, wenn m prim ist. Für den Zeitaufwand für die Suche nach einem Schlüssel kann man zeigen im

$$\text{erfolgreichen Fall: } \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

$$\text{erfolglosen Fall: } \frac{1}{1-\alpha}$$

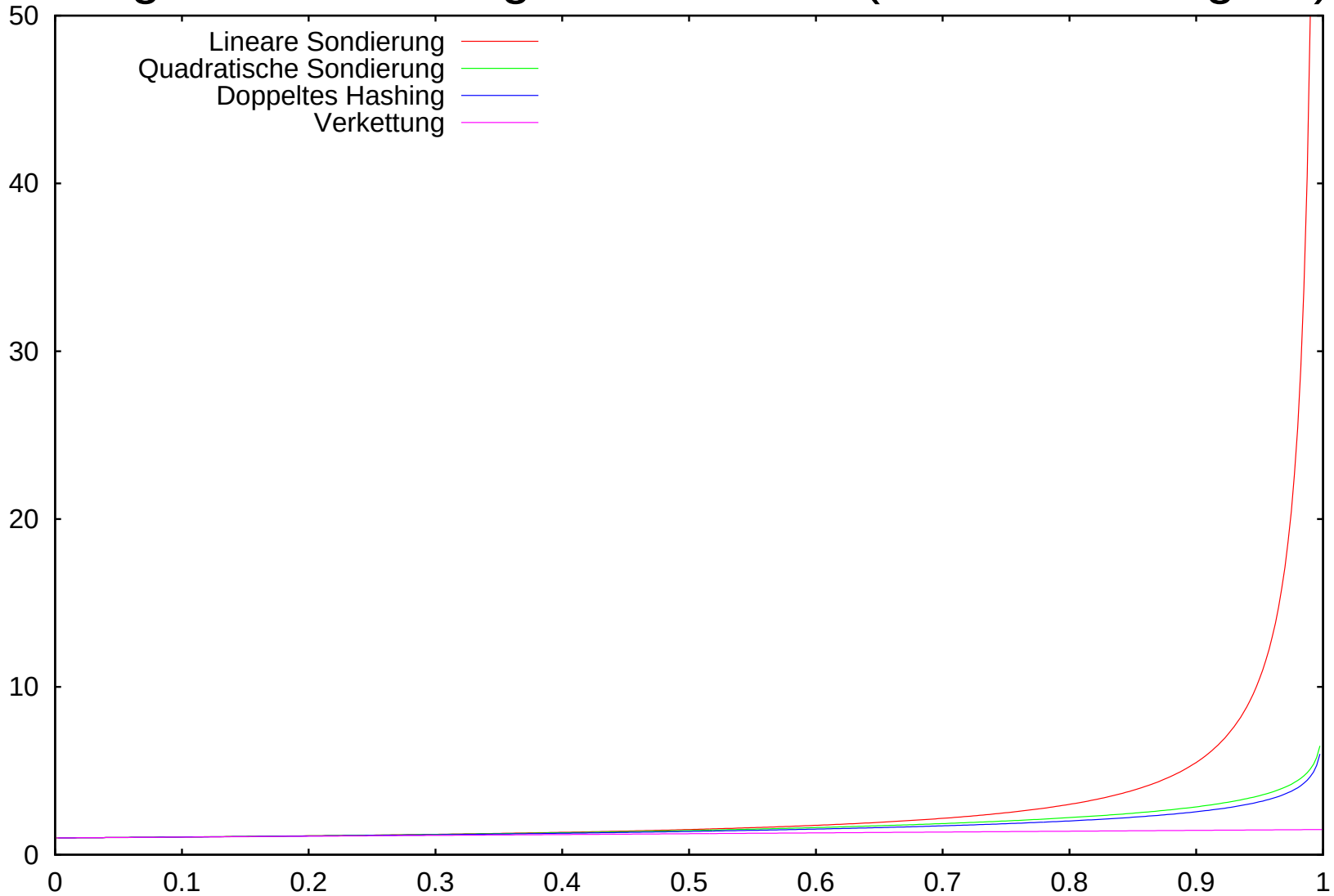
Für die Implementierung von delete gilt die gleiche Bemerkung wie beim linearen Sondieren.

Bemerkung:
Hashing mit Kollisionsauflösung durch lineares oder quadratisches Sondieren, wie auch doppeltes Hashing, sind Beispiele für so genannte

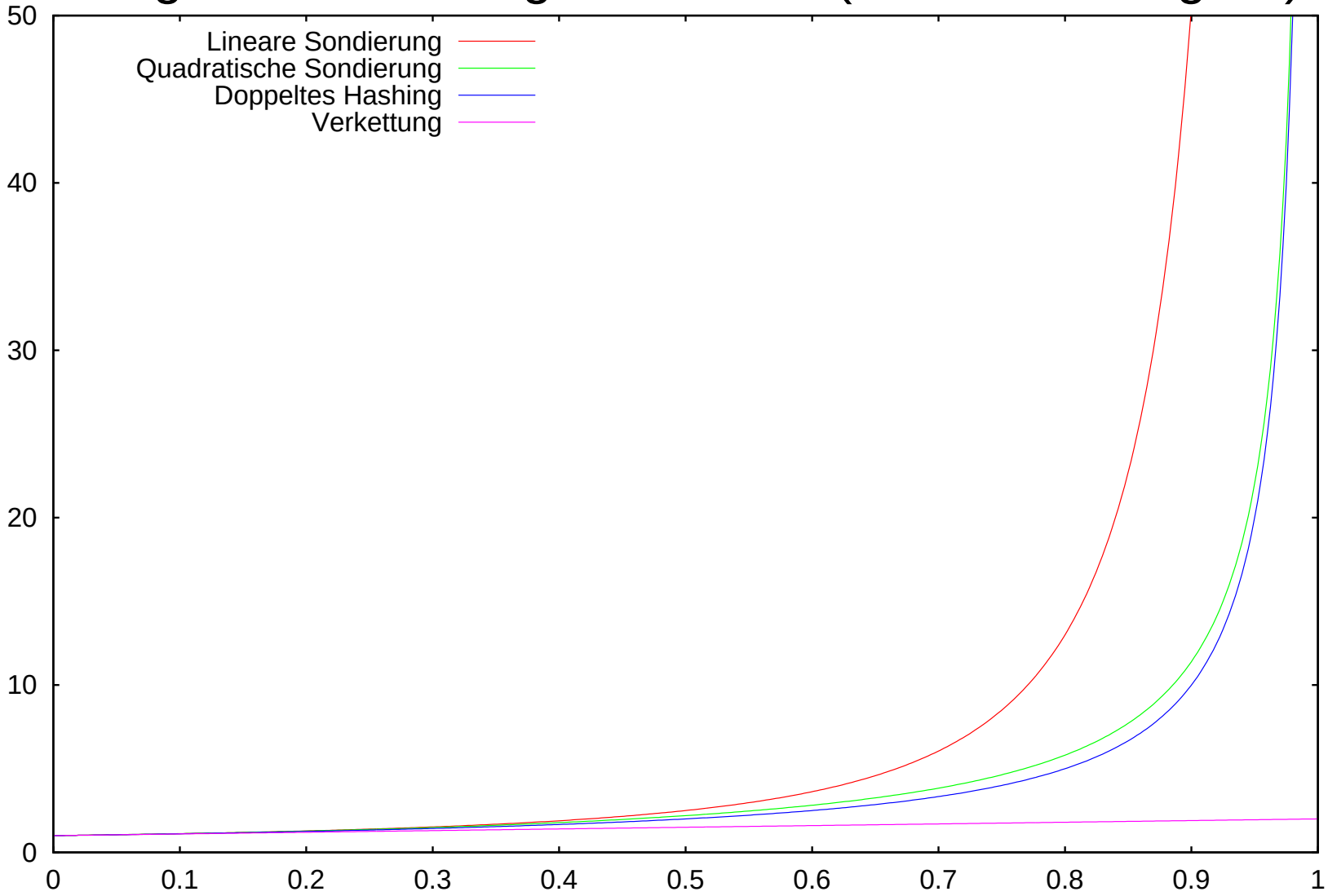
offene Hashverfahren,

da die Schlüssel nicht unbedingt an ihrer (ersten) Hashadresse abgespeichert werden.

Vergleiche für erfolgreiche Suche (abh. vom Füllgrad)



Vergleiche für erfolglose Suche (abh. vom Füllgrad)



3.4.1 Universelle Hashfunktionen

Definition 172

Eine Menge $\mathcal{H}$ von Hashfunktionen heißt universell, falls gilt:

$$\forall x, y \in U, x \neq y \underbrace{\frac{|\{h \in \mathcal{H};\ h(x) = h(y)\}|}{|\mathcal{H}|}}_{\Pr\{h(x)=h(y)\}} \leq \frac{1}{m}$$

Satz 173

Sei $\mathcal{H}$ eine universelle Familie von Hashfunktionen (für eine Hashtabelle der Größe m), sei $S \subset U$ eine feste Menge von Schlüsseln mit $|S| = n \leq m$, und sei $h \in \mathcal{H}$ eine zufällig gewählte Hashfunktion (wobei alle Hashfunktionen $\in \mathcal{H}$ gleich wahrscheinlich sind). Dann ist die erwartete Anzahl von Kollisionen eines festen Schlüssels $x \in S$ mit anderen Schlüsseln in S kleiner als 1.

Beweis:

Es gilt

$$\mathbb{E}[\#\text{ Kollisionen mit } x] = \sum_{\substack{y \in S \\ y \neq x}} \frac{1}{m} = \frac{n-1}{m} < 1\,.$$

Sei nun $m = p$ eine Primzahl. Dann ist $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ ein Körper.

Wir betrachten im Folgenden $U = \{0, 1, \ldots, p-1\}^{r+1}$.

Sei die Familie $\mathcal{H} = \{h_\alpha;\ \alpha \in U\}$ von Hashfunktionen definiert durch

$$h_\alpha(x_0, \ldots, x_r) = \left(\sum_{i=0}^{r} \alpha_i \cdot x_i \right) \bmod p$$

Satz 174

$\mathcal{H}$ ist universell.

Beweis:

Sei $x \neq y$, o.B.d.A. $x_0 \neq y_0$. Dann gilt $h_\alpha(x) = h_\alpha(y)$ gdw

$$\alpha_0(y_0 - x_0) = \sum_{i=1}^{r} \alpha_i(x_i - y_i)\,.$$

Für jede Wahl von $(\alpha_1, \ldots, \alpha_r)$ gibt es daher genau ein α_0, das zu einer Kollision führt.

Es gilt daher

$$\frac{|\{h \in \mathcal{H};\ h(x) = h(y)\}|}{|\mathcal{H}|} = \frac{p^r}{p^{r+1}} = \frac{1}{p} = \frac{1}{m}$$

4. Vorrangwarteschlangen (priority queues)

Definition 175

Eine Vorrangwarteschlange (priority queue) ist eine Datenstruktur, die die folgenden Operationen effizient unterstützt:

1. Insert
2. ExtractMin Extrahieren und Löschen des Elements mit dem kleinsten Schlüssel
3. DecreaseKey Verkleinern eines Schlüssels
4. Union (meld) Vereinigung zweier (disjunkter) Priority Queues

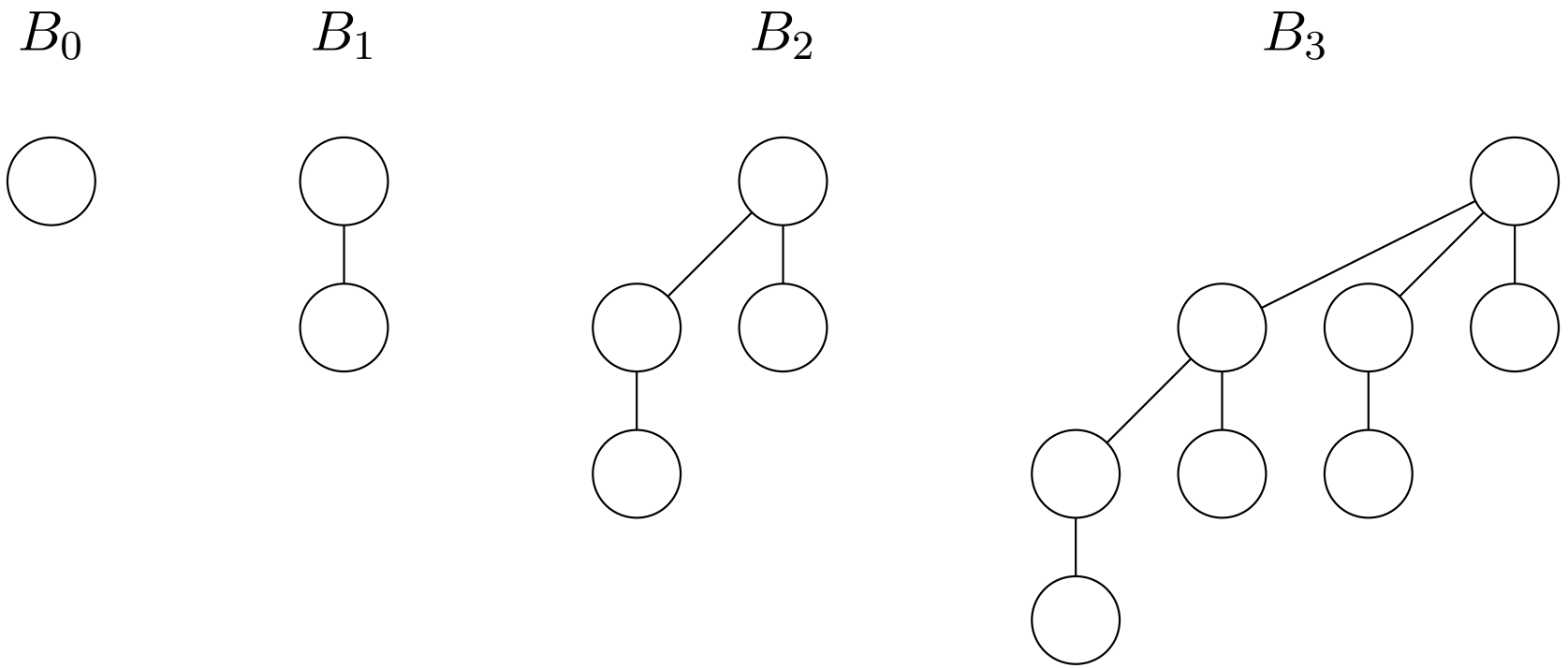
Wir besprechen die Implementierung einer Vorrangwarteschlange als Binomial Heap. Diese Implementierung ist (relativ) einfach, aber asymptotisch bei weitem nicht so gut wie modernere Datenstrukturen für Priority Queues, z.B. Fibonacci Heaps (die in weiterführenden Vorlesungen besprochen werden).

Hier ist ein kurzer Vergleich der Zeitkomplexitäten:

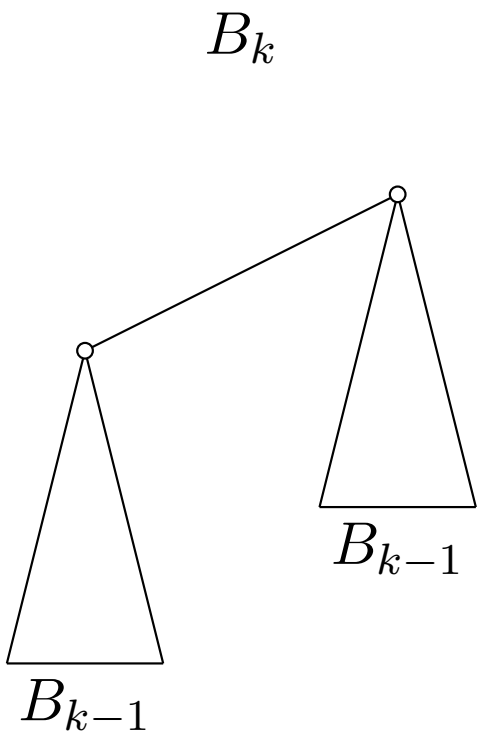
	BinHeap	FibHeap
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(1)$
Union	$O(\log n)$	$O(1)$
	worst case	amortisiert

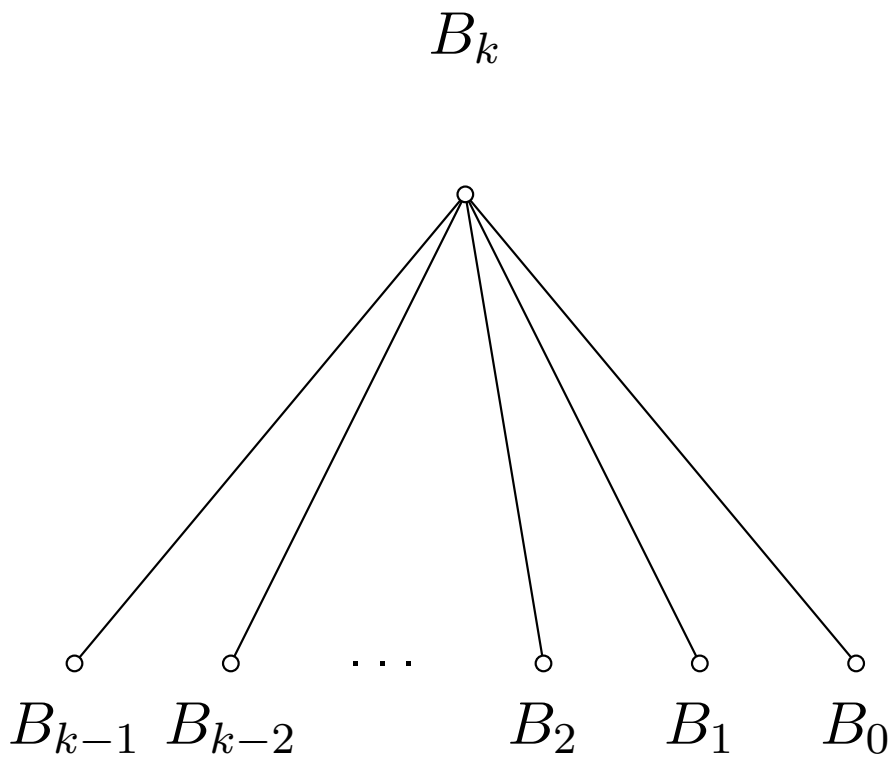
Definition 176

- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.

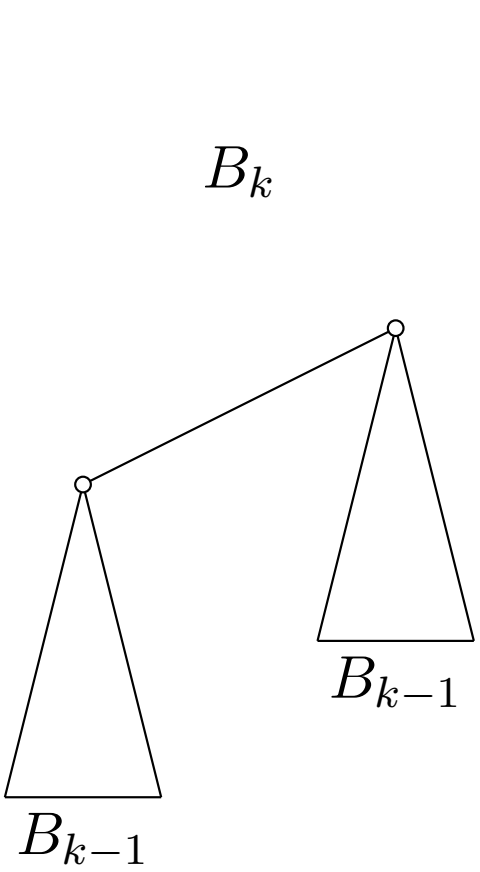


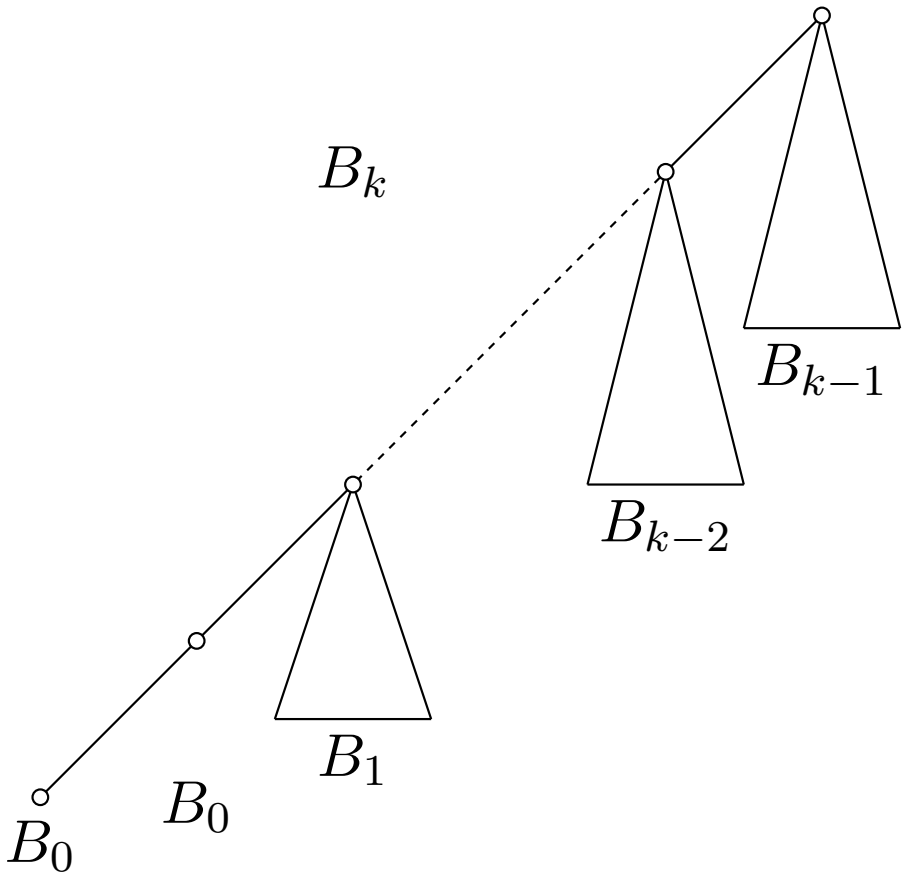
Rekursiver Aufbau des Binomialbaums B_k (rekursive Verfolgung des rechten Zweigs)





Rekursiver Aufbau des Binomialbaums B_k (rekursive Verfolgung des linken Zweigs)





Satz 177

Für den Binomialbaum B_k, $k \geq 0$, gilt:

1. *er hat Höhe k und enthält 2^k Knoten;*
2. *er enthält genau $\binom{k}{i}$ Knoten der Tiefe i, für alle $i \in \{0, \ldots, k\}$;*
3. *seine Wurzel hat k Kinder, alle anderen Knoten haben $< k$ Kinder.*

Beweis:

Der Beweis ergibt sich sofort aus dem rekursiven Aufbau des B_k und der Rekursionsformel

$$\binom{k}{i} = \binom{k-1}{i} + \binom{k-1}{i-1}$$

für Binomialkoeffizienten.

Definition 178

Ein Binomial Heap ist eine *Menge* $\mathcal{H}$ von Binomialbäumen, wobei jedem Knoten v ein Schlüssel $key(v)$ zugeordnet ist, so dass folgende Eigenschaften gelten:

1. jeder Binomialbaum $\in \mathcal{H}$ erfüllt die Heap-Bedingung und ist ein min-Heap:

$$(\forall \text{ Knoten } v, w)[v \text{ Vater von } w \Rightarrow key(v) \leq key(w)]$$

2. $\mathcal{H}$ enthält für jedes $k \in \mathbb{N}_0$ höchstens einen B_k

Für jeden Binomial Heap $\mathcal{H}$ gilt also

1. Enthält $\mathcal{H}$ n Schlüssel, $n \in \mathbb{N}$, so besteht $\mathcal{H}$ höchstens aus $\max\{1, \lceil \mathrm{ld}\, n \rceil\}$ Binomialbäumen.
2. In jedem von $\mathcal{H}$'s Binomialbäumen ist ein kleinster Schlüssel an der Wurzel gespeichert; verlinkt man daher die Wurzeln aller Binomialbäume von $\mathcal{H}$ in einer zirkulären Liste, kann ein minimaler Schlüssel in $\mathcal{H}$ durch einfaches Durchlaufen dieser Liste gefunden werden.

Korollar 179

In einem Binomial Heap mit n Schlüsseln benötigt FindMin *Zeit* $O(\log n)$.

Wir betrachten nun die Realisierung der Union-Operation. Hierbei wird vorausgesetzt, dass die Objektmengen, zu denen die Schlüssel in den beiden zu verschmelzenden Binomial Heaps $\mathcal{H}_1$ und $\mathcal{H}_2$ gehören, disjunkt sind. Es ist jedoch durchaus möglich, dass der gleiche Schlüssel für mehrere Objekte vorkommt.

1. Fall $\mathcal{H}_1$ und $\mathcal{H}_2$ enthalten jeweils nur einen Binomialbaum B_k (mit dem gleichen Index k):

In diesem Fall fügen wir den B_k, dessen Wurzel den größeren Schlüssel hat, als neuen Unterbaum der Wurzel des anderen B_k ein, gemäß der rekursiven Struktur der Binomialbäume. Es entsteht ein B_{k+1}, für den per Konstruktion weiterhin die Heap-Bedingung erfüllt ist.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls kein Binomialbaum mit Index k vorhanden ist: **noop**
2. falls genau ein Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls genau zwei Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls genau drei Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen; die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

Man beachte, dass nie mehr als 3 Binomialbäume mit gleichem Index auftreten können!

Bemerkung:
Es besteht eine einfache Analogie zur Addition zweier Binärzahlen, wobei das Verschmelzen zweier gleich großer Binomialbäume dem Auftreten eines Übertrags entspricht!

Lemma 180
Die Union-*Operation zweier Binomial Heaps mit zusammen* n *Schlüsseln kann in Zeit* $O(\log n)$ *durchgeführt werden.*

Die Operation Insert$(\mathcal{H}, k)$:

1. erzeuge einen neuen Binomial Heap $\mathcal{H}' = \{B_0\}$ mit k als einzigem Schlüssel
2. $\mathcal{H} :=$ Union$(\mathcal{H}, \mathcal{H}')$

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der Insert-Operation offensichtlich $O(\log n)$.

Die Operation ExtractMin($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. entferne diesen Binomialbaum aus $\mathcal{H}$
4. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen der Wurzel dieses Baums
5. $\mathcal{H} :=$ Union$(\mathcal{H}, \mathcal{H}')$

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der ExtractMin-Operation offensichtlich $O(\log n)$.

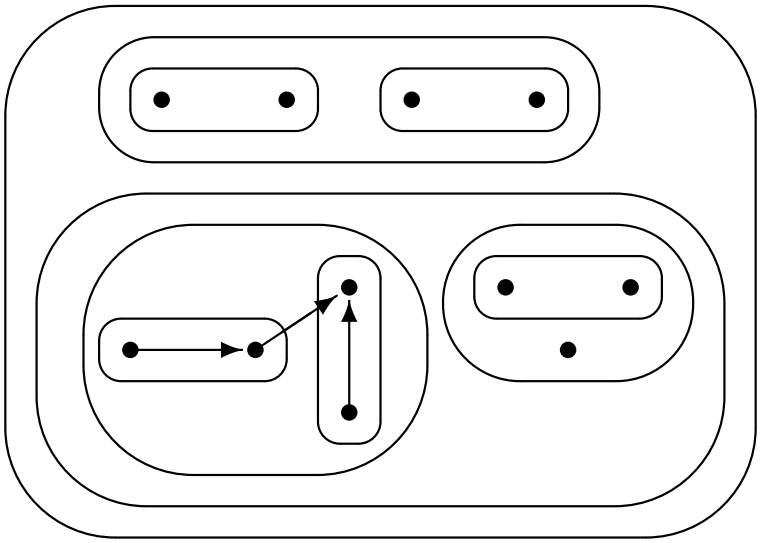
Die Operation DecreaseKey($\mathcal{H}, v, k$) (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der DecreaseKey-Operation offensichtlich $O(\log n)$.

5. Union/Find-Datenstrukturen

5.1 Motivation

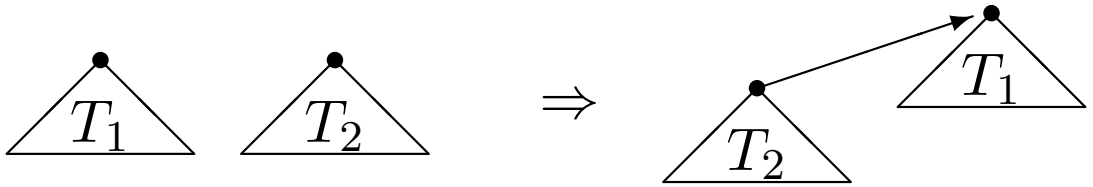


- *Union*(T_1, T_2): Vereinige T_1 und T_2
  $T_1 \cap T_2 = \emptyset$
- *Find*(x): Finde den Repräsentanten der (größten) Teilmenge, in der sich x gerade befindet.

5.2 Union/Find-Datenstruktur

5.2.1 Intrees

1. Initialisierung: $x \longrightarrow \bullet x$: Mache x zur Wurzel eines neuen (einelementigen) Baumes.
2. *Union*(T_1, T_2):



3. *Find*: Suche Wurzel des Baumes, in dem sich x befindet.

Bemerkung: Naive Implementation: worst-case-Tiefe $= n$

- Zeit für *Find* $= \Omega(n)$
- Zeit für *Union* $= \mathcal{O}(1)$

5.2.2 Gewichtete Union (erste Verbesserung)

Mache die Wurzel des kleineren Baumes zu einem Kind der Wurzel des größeren Baumes. Die Tiefe des Baumes ist dann $\mathcal{O}(\log n)$.

- Zeit für *Find* $= \mathcal{O}(\log n)$
- Zeit für *Union* $= \mathcal{O}(1)$

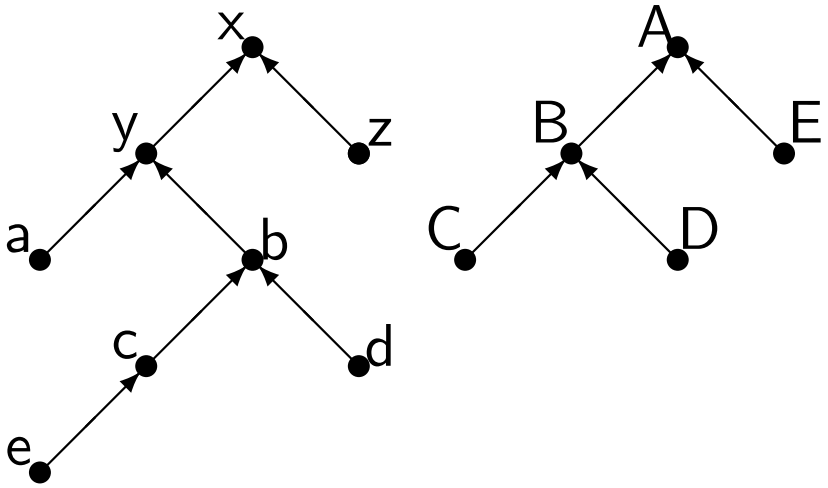
Es gilt auch: Tiefe des Baumes im worst-case:

$$\boxed{\Omega(\log n)}$$

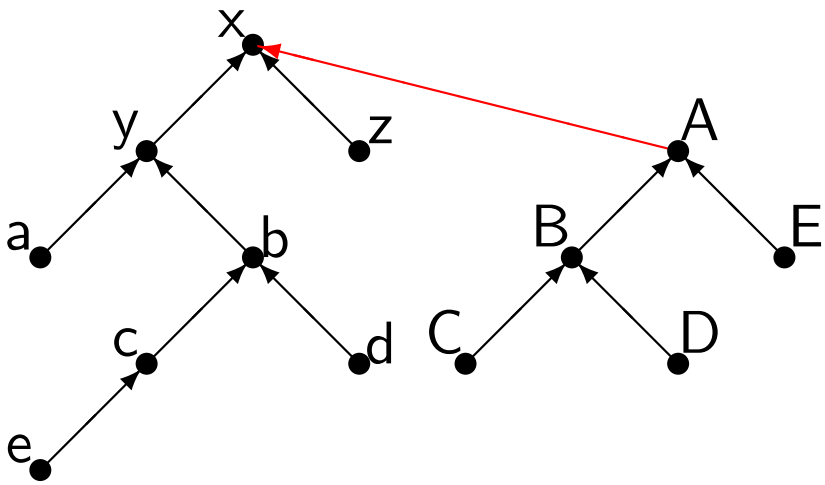
5.2.3 Pfad-Kompression mit gewichteter Union (zweite Verbesserung)

Wir betrachten eine Folge von k *Find*- und *Union*-Operationen auf einer Menge mit n Elementen, darunter $n-1$ *Union*.

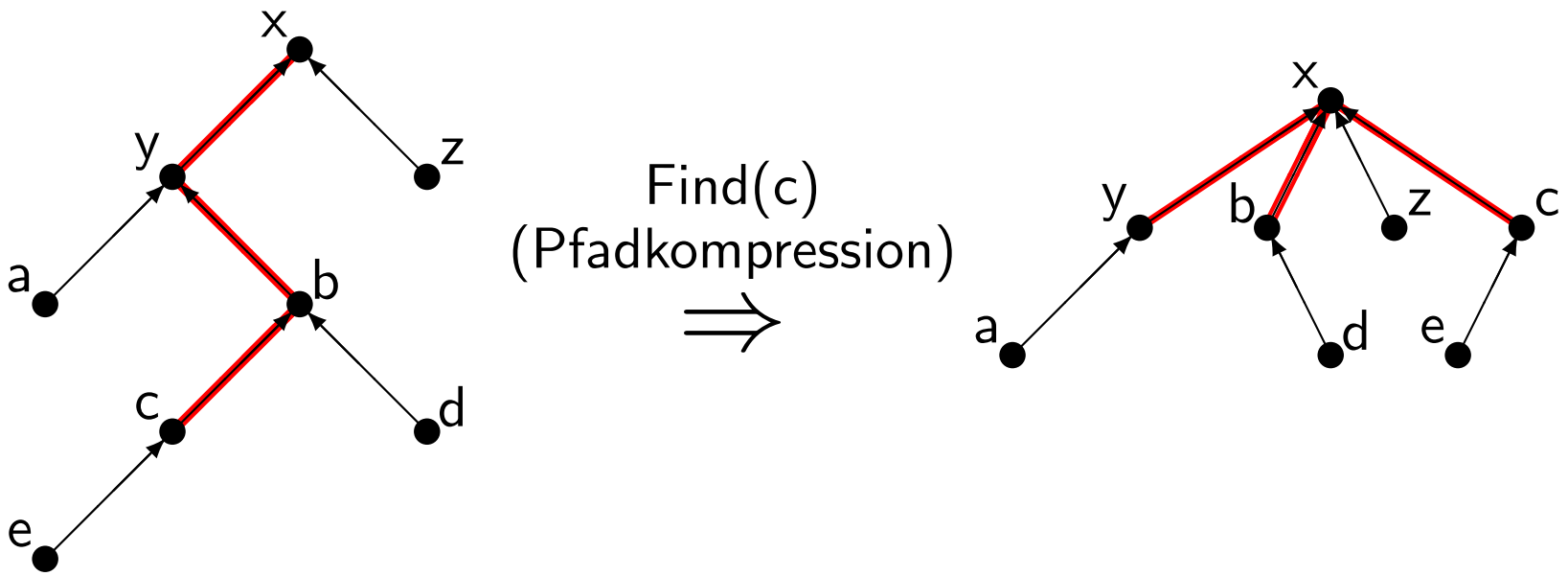
Implementierung: Gewichtete *Union* für Pfad-Kompression:



Union $\Rightarrow$



Implementierung: *Find* für Pfad-Kompression:



Bemerkung:
Nach Definition ist

$$\log^* n = \min\{i \geq 0; \underbrace{\log\log\log\ldots\log n}_{i \text{ log 's}} \leq 1\}$$

Beispiel 181

$$\log^* 0 = \log^* 1 = 0$$
$$\log^* 2 = 1$$
$$\log^* 3 = 2$$
$$\log^* 16 = 3 \qquad \text{da } 16 = 2^{2^2}$$
$$\log^* 2^{65536} = 5 \qquad \text{da } 2^{65536} = 2^{2^{2^{2^2}}}$$

Satz 182

Bei der obigen Implementierung ergibt sich eine amortisierte Komplexität von $\mathcal{O}(\log^* n)$ pro Operation.

Beweis:

(der Vollständigkeit halber, gehört nicht zum Vorlesungsstoff!)

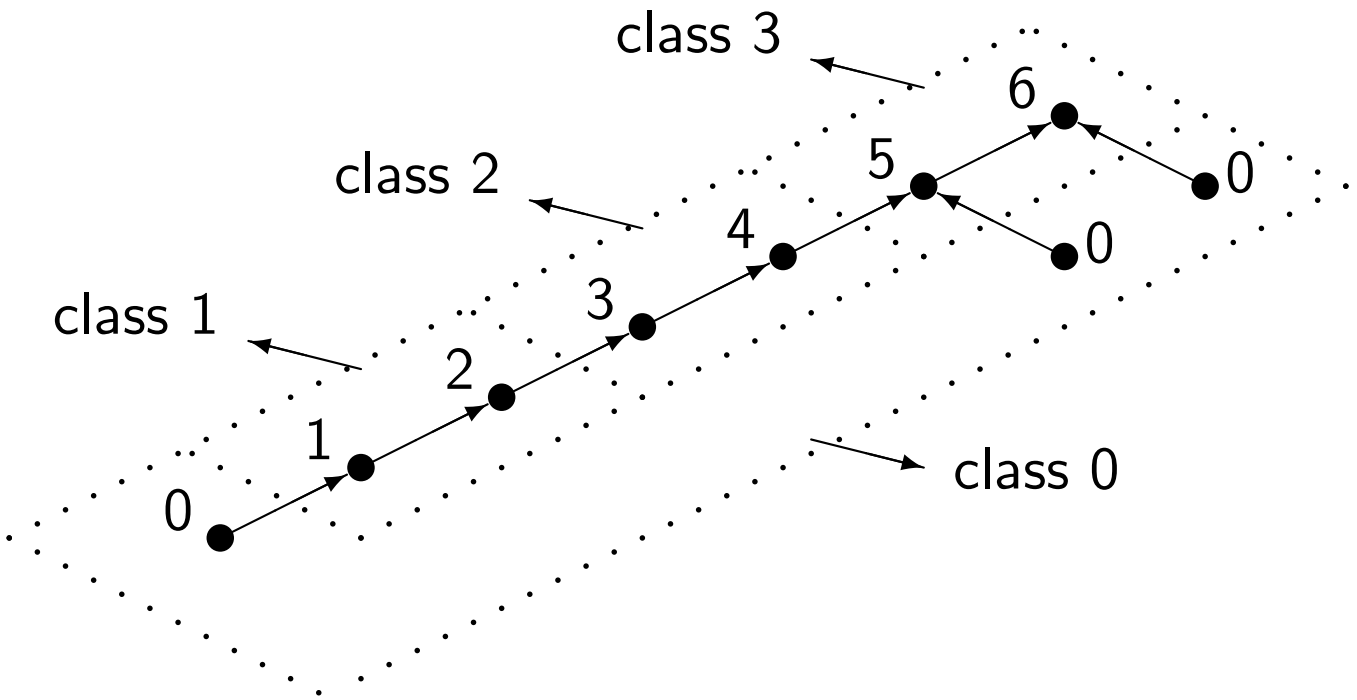
Sei T' der (endgültige) In-Baum, der durch die Folge der *Union*'s, ohne die *Find*'s, entstehen würde (also keine Pfad-Kompression).

Ordne jedem Element x drei Werte zu:

- rank(x):= Höhe des Unterbaums in T' mit Wurzel x
- class(x):= $\begin{cases} i \geq 1 & \text{falls } a_{i-1} < \text{ rank}(x) \leq a_i \text{ ist } (i \geq 1) \\ 0 & \text{falls rank}(x) = 0 \end{cases}$

  Dabei gilt: $a_0 = 0, a_i = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{i \ 2'\text{en}}$ für $i \geq 1$.

  Setze zusätzlich $a_{-1} := -1$.


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

Beweis (Forts.):

- dist(x) ist die Distanz von x zu einem Vorfahr y im momentanen Union/Find-Baum (mit Pfad-Kompression), so dass class(y) $>$ class(x) bzw. y die Wurzel des Baumes ist.

Definiere die Potenzialfunktion

$$\text{Potenzial} := c \sum_x \text{dist}(x), \quad c \text{ eine geeignete Konstante } > 0$$

Beweis (Forts.):

Beobachtungen:

i) Sei T ein Baum in der aktuellen Union/Find-Struktur (mit Pfad-Kompression), seien x, y Knoten in T, y Vater von x. Dann ist class$(x) \leq$ class(y).

ii) Aufeinander folgende *Find*(x) durchlaufen (bis auf eine) verschiedene Kanten. Diese Kanten sind (im wesentlichen) eine Teilfolge der Kanten in T' auf dem Pfad von x zur Wurzel.

Beweis (Forts.):

Amortisierte Kosten Find(x):

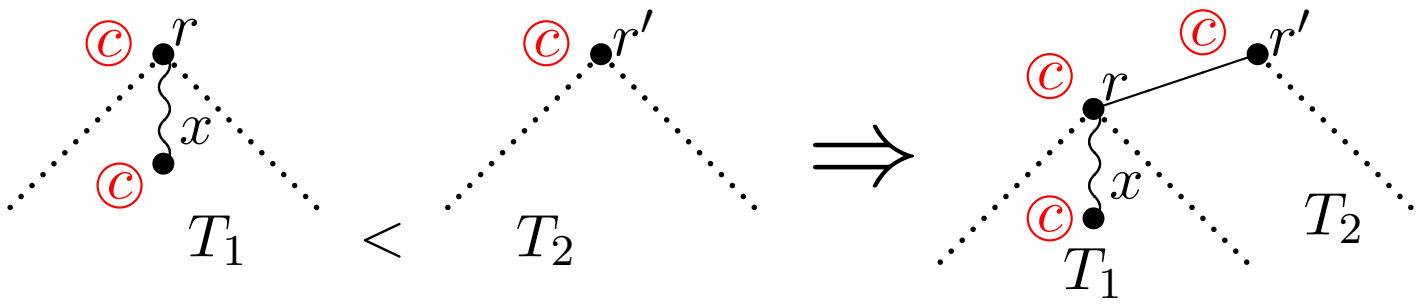
Sei $x_0 \to x_1 \to x_2 \ldots x_k = r$ der Pfad von x_0 zur Wurzel. Es gibt höchstens $\log^* n$-Kanten (x_{i-1}, x_i) mit class$(x_{i-1}) <$ class(x_i). Ist class$(x_{i-1}) =$ class(x_i) und $i < k$ (also $x_i \neq r$), dann ist dist(x_{i-1}) vor der *Find(x)*-Operation ≥ 2, nachher gleich 1.

Damit können die Kosten für alle Kanten (x_{i-1}, x_i) mit class(x_{i-1}) $=$ class(x_i) aus der Potenzialverringerung bezahlt werden. Es ergeben sich damit amortisierte Kosten

$$\boxed{\mathcal{O}(\log^* n)}$$

Beweis (Forts.):

Amortisierte Gesamtkosten aller $(n-1)$-Union's:



Die gesamte Potenzialerhöhung durch alle *Union*'s ist nach oben durch das Potenzial von T' beschränkt (Beobachtung ii).

Beweis (Forts.):

$$\begin{aligned}
\text{Potenzial}(T') &\leq c \cdot \sum_{i=0}^{\log^* n} \sum_{\text{rank}(x)=j=a_{i-1}+1}^{a_i} \text{dist}(x) \\
&\leq c \cdot \sum_{i=0}^{\log^* n} \sum_{\text{rank}(x)=j=a_{i-1}+1}^{a_i} \frac{n}{2^j} a_i \\
&\leq c \cdot n \sum_{i=0}^{\log^* n} a_i \frac{1}{2^{a_{i-1}}} = c \cdot n \sum_{i=0}^{\log^* n} 1 \\
&= \mathcal{O}(n \log^* n) \,.
\end{aligned}$$

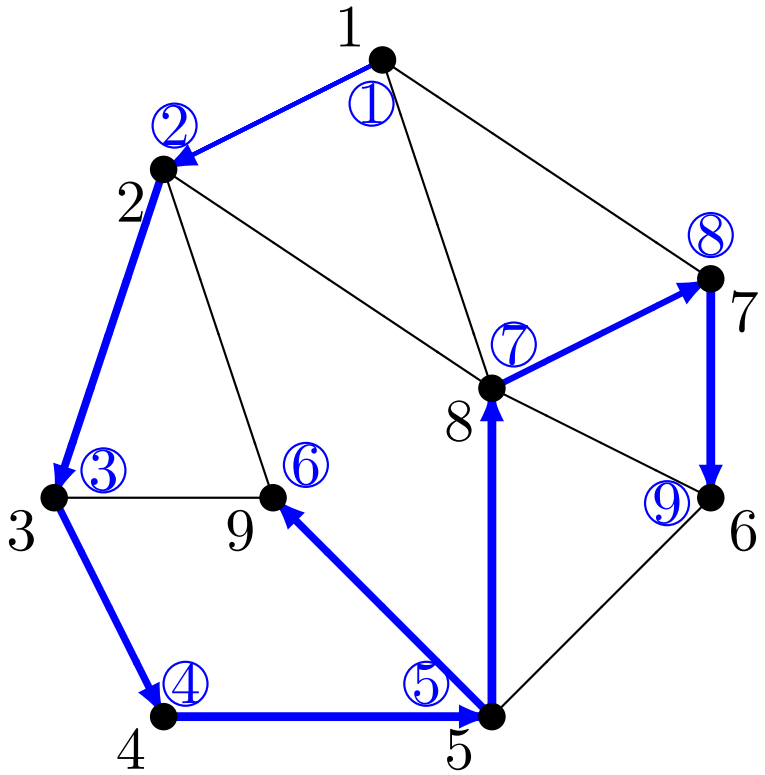
Die zweite Ungleichung ergibt sich, da alle Unterbäume, deren Wurzel x $\text{rank}(x) = j$ hat, disjunkt sind und jeweils $\geq 2^j$ Knoten enthalten. □

6. Traversierung von Graphen

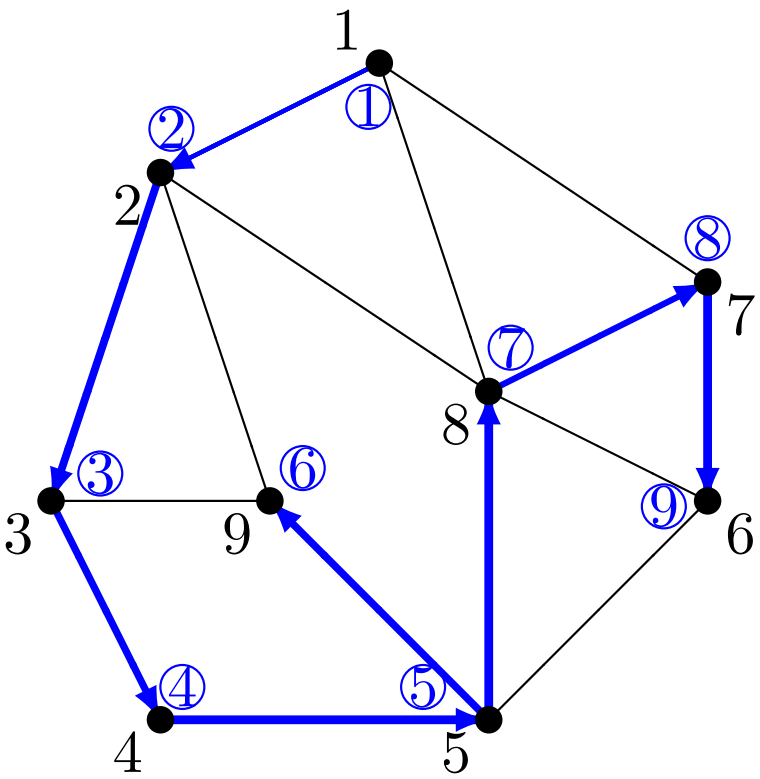
Sei $G = (V, E)$ ein ungerichteter Graph. Anhand eines Beipiels betrachten wir die zwei Algorithmen DFS (Tiefensuche) und BFS (Breitensuche).

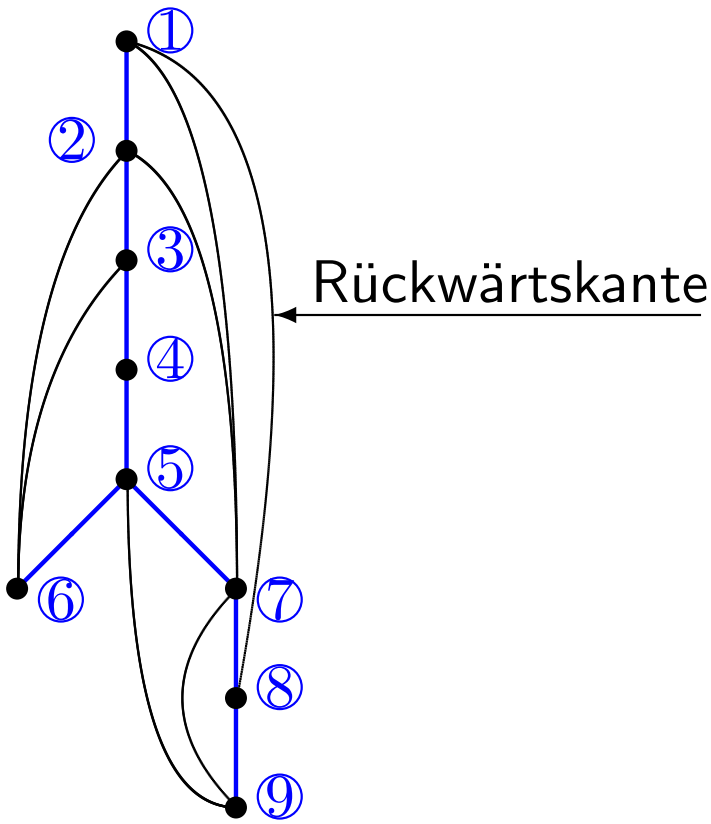
```
while ∃ unvisited v do
    r := pick (random) unvisited node
    push r onto stack
    while stack ≠ ∅ do
        v := pop top element
        if v unvisited then
            mark v visited
            push all neighbours of v onto stack
            perform operations DFS_Ops(v)
        fi
    od
od
```

Beispiel 183



Beobachtung: Die markierten Kanten bilden einen Spannbaum:

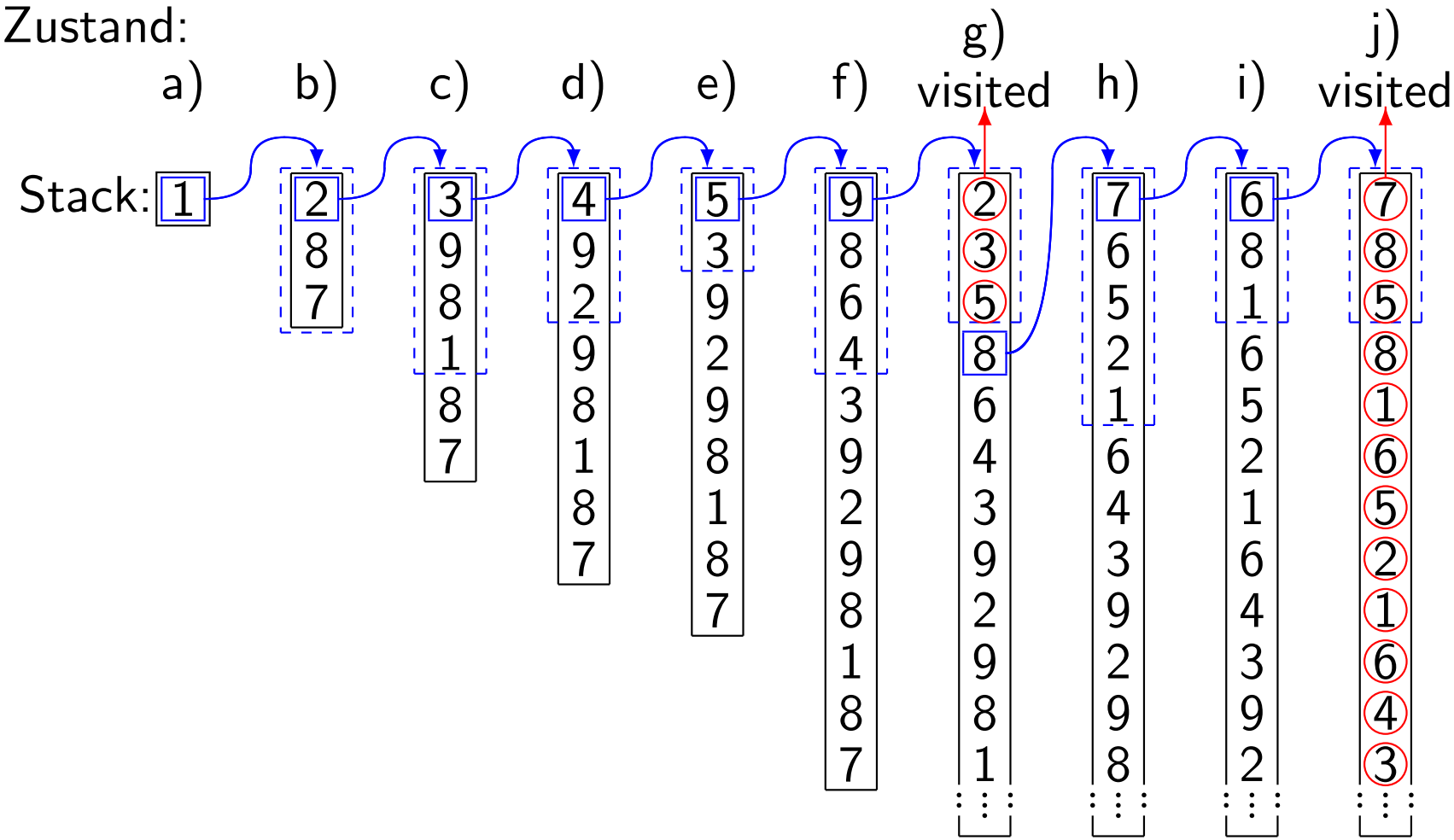




Folge der Stackzustände

: oberstes Stackelement : Nachbarn

: schon besuchte Knoten

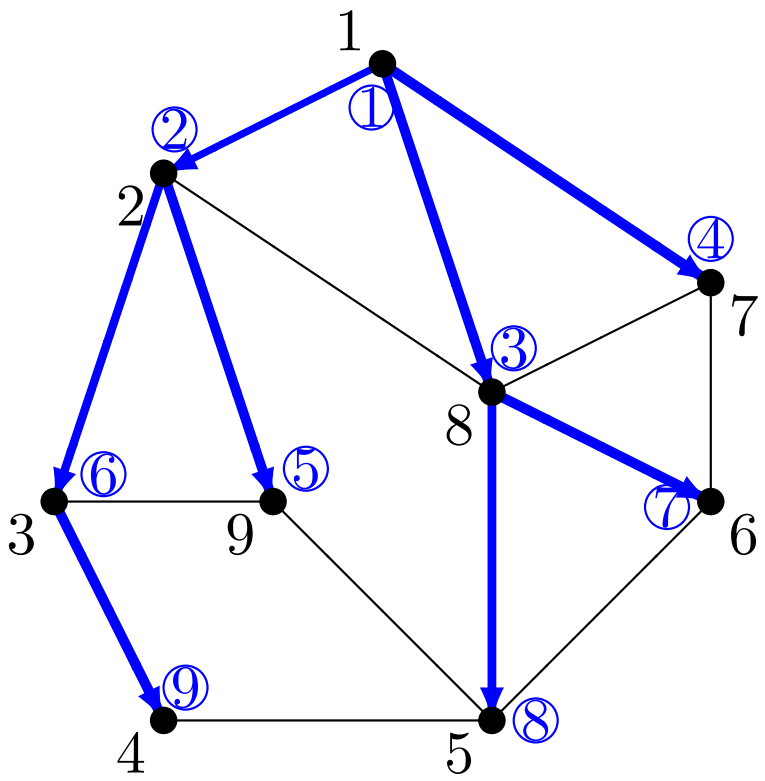


Wir betrachten den Stackzustand:
Im Zustand g) sind die Elemente 2, 3 und 5 als visited markiert (siehe Zustände b), c) und e)). Deswegen werden sie aus dem Stack entfernt, und das Element 8 wird zum obersten Stackelement. Im Zustand j) sind alle Elemente markiert, so dass eins nach dem anderen aus dem Stack entfernt wird.

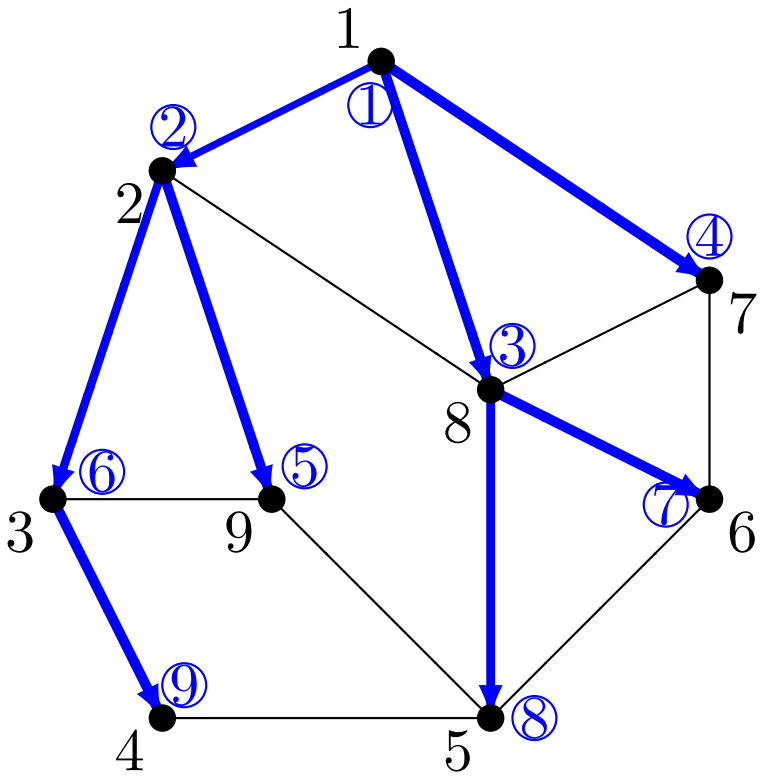
6.2 BFS-Algorithmus

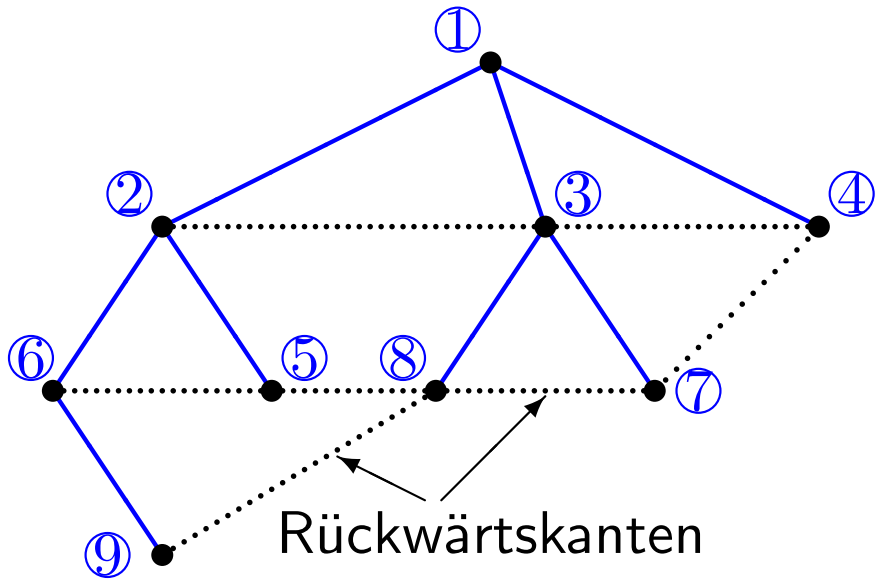
while $\exists$ unvisited v **do**
    $r :=$ pick (random) unvisited node
    push r into (end of) queue
    while queue $\neq \emptyset$ **do**
        $v :=$ remove front element of queue
        if v unvisited **then**
            mark v visited
            append all neighbours of v to end of queue
            perform operations BFS_Ops(v)
        fi
    od
od

Beispiel 184

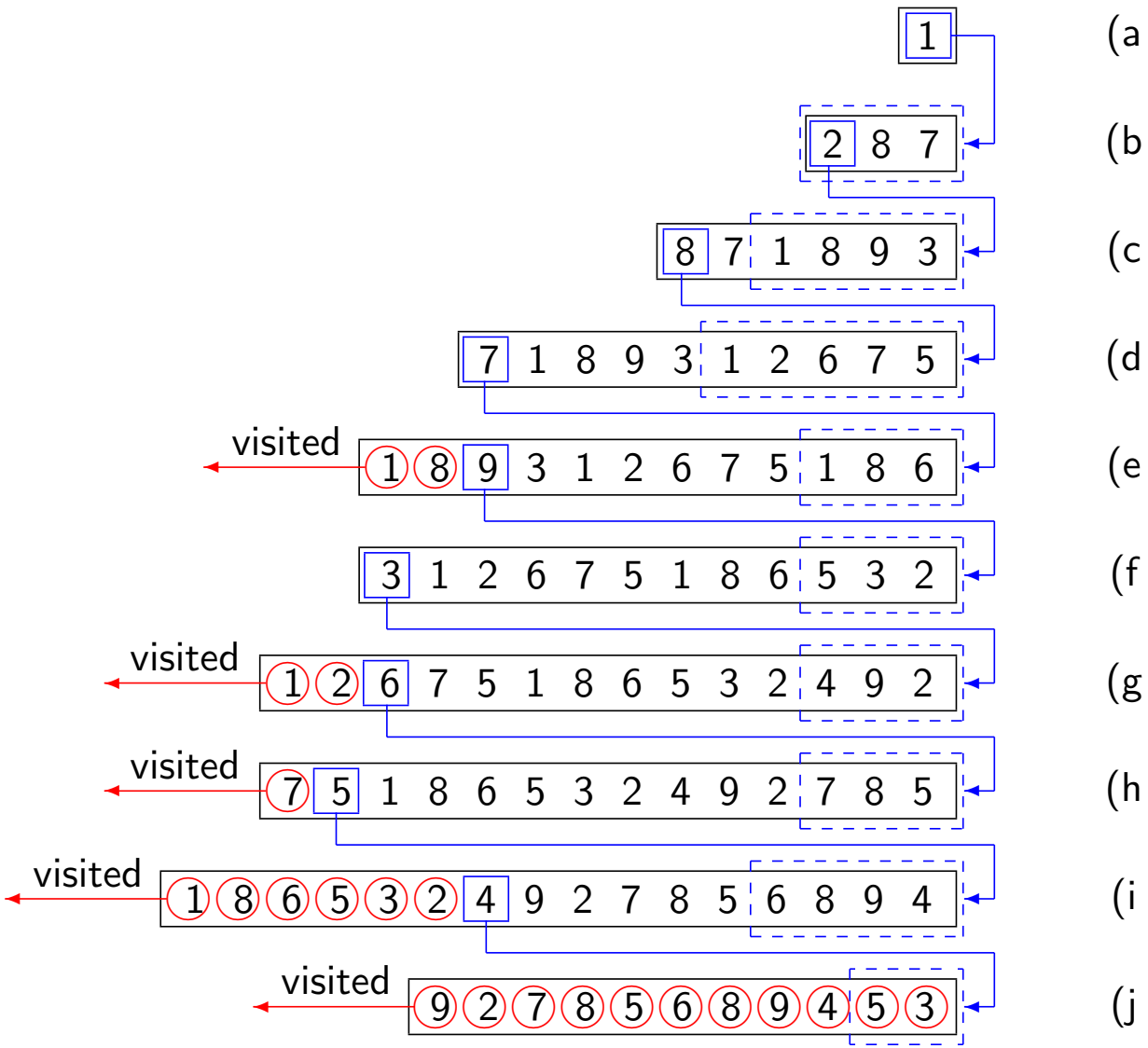


Beobachtung: Die markierten Kanten bilden einen Spannbaum:





Folge der Queuezustände



Wir betrachten die Folge der Queuezustände. Wiederum bedeutet die Notation:

 : vorderstes Queue-Element

 : schon besuchte Knoten

 : Nachbarn

Im Zustand e) sind die Elemente 1 und 8 als visited markiert (siehe Zustände a) und c)). Deswegen werden sie aus der Queue entfernt, und so wird das Element 9 das vorderste Queueelement. Das gleiche passiert in den Zuständen g), h) und i). Im Zustand j) sind alle Elemente markiert, so dass sie eins nach dem anderen aus der Queue entfernt werden.

7. Minimale Spannbäume

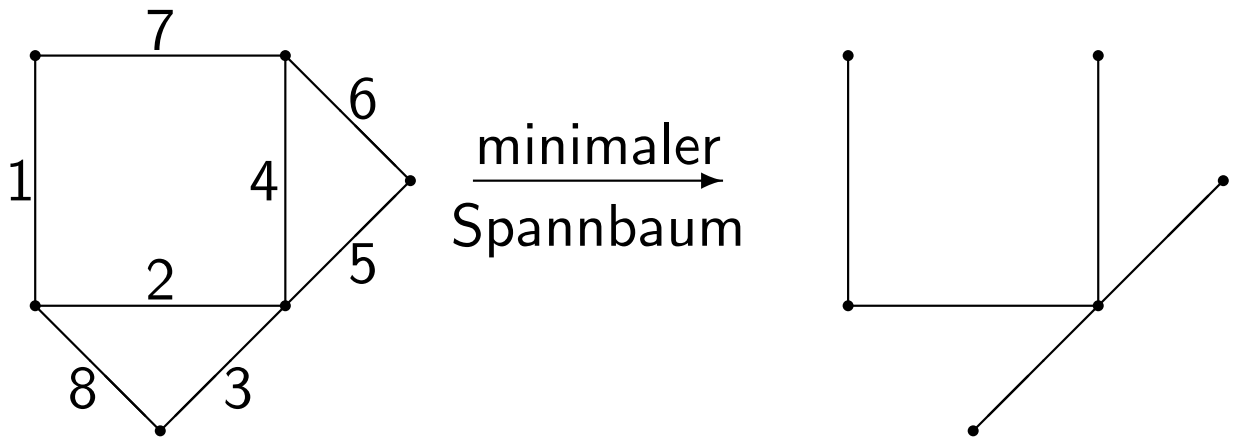
Sei $G = (V, E)$ ein einfacher ungerichteter Graph, der o.B.d.A. zusammenhängend ist. Sei weiter $w : E \to \mathbb{R}$ eine Gewichtsfunktion auf den Kanten von G.
Wir setzen

- $E' \subseteq E$: $w(E') = \sum_{e \in E'} w(e)$,
- $T = (V', E')$ ein Teilgraph von G: $w(T) = w(E')$.

Definition 185
T heißt minimaler Spannbaum (MSB, MST) von G, falls T Spannbaum von G ist und gilt:

$$w(T) \leq w(T') \text{ für alle Spannbäume } T' \text{ von } G.$$

Beispiel 186



Anwendungen:

- Telekom: Verbindungen der Telefonvermittlungen
- Leiterplatinen

7.1 Konstruktion von minimalen Spannbäumen

Es gibt zwei Prinzipien für die Konstruktion von minimalen Spannbäumen (Tarjan):

- „blaue“ Regel
- „rote“ Regel

Satz 187

*Sei $G = (V, E)$ ein zusammenhängender ungerichteter Graph, $w : E \to \mathbb{R}$ eine Gewichtsfunktion, $C = (V_1, V_2)$ ein Schnitt (d.h. $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, $V_1 \neq \emptyset \neq V_2$). Sei weiter $E_C = E \cap (V_1 \times V_2)$ die Menge der Kanten „über den Schnitt hinweg“. Dann gilt: (**„blaue“ Regel**)*

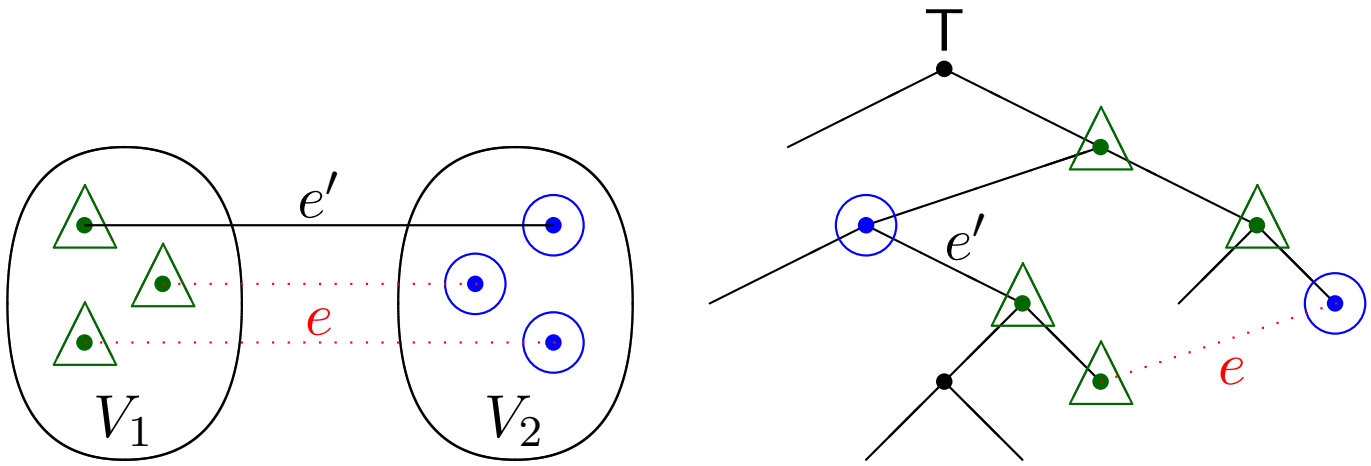
1. *Ist $e \in E_C$ die einzige Kante minimalen Gewichts (über alle Kanten in E_C), dann ist e in jedem minimalen Spannbaum für (G,w) enthalten.*
2. *Hat $e \in E_C$ minimales Gewicht (über alle Kanten in E_C), dann gibt es einen minimalen Spannbaum von (G,w), der e enthält.*

Beweis:

[durch Widerspruch]

1. Sei T ein minimaler Spannbaum von (G, w), sei $e \in E_C$ die minimale Kante. Annahme: $e \notin T$. Da T Spannbaum $\Rightarrow T \cap E_C \neq \emptyset$.
   Sei $T \cap E_C = \{e_1, e_2, \ldots, e_k\}$, $k \geq 1$. Dann enthält $T \cup \{e\}$ einen eindeutig bestimmten Kreis (den sogenannten durch e bzgl. T bestimmten Fundamentalkreis). Dieser Kreis muss mindestens eine Kante $\in E_C \cap T$ enthalten, da die beiden Endpunkte von e auf verschiedenen Seiten des Schnitts C liegen.

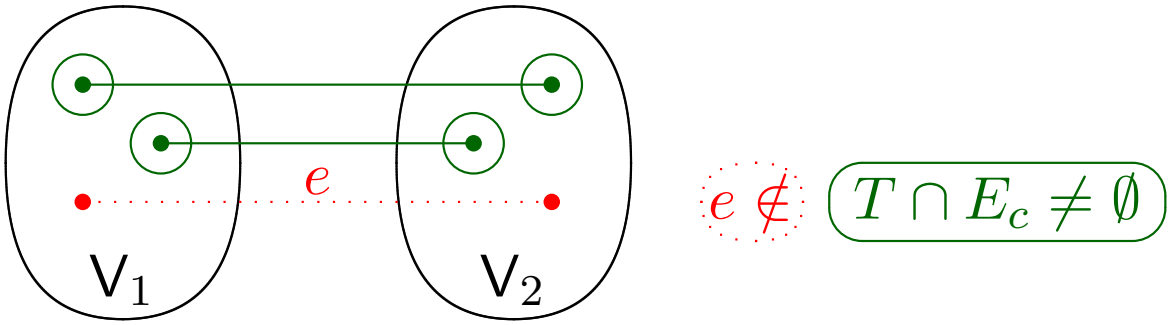
Beweis (Forts.):



Sei $e' \in E_C \cap T$. Dann gilt nach Voraussetzung $w(e') > w(e)$. Also ist $T' := T - \{e'\} \cup \{e\}$ ein Spannbaum von G, der echt kleineres Gewicht als T hat, Widerspruch zu „T ist minimaler Spannbaum“.

Beweis (Forts.):

2. Sei $e \in E_C$ minimal. Annahme: e kommt in keinem minimalen Spannbaum vor. Sei T ein beliebiger minimaler Spannbaum von (G, w).



$e \notin T \cap E_C \neq \emptyset$. Sei $e' \in E_C \cap T$ eine Kante auf dem durch e bezüglich T erzeugten Fundamentalkreis. Dann ist $T' = T - \{e'\} \cup \{e\}$ wieder ein Spannbaum von G, und es ist $w(T') \leq w(T)$. Also ist T' minimaler Spannbaum und $e \in T'$.

Satz 188

*Sei $G = (V, E)$ ein ungerichteter, gewichteter, zusammenhängender Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$. Dann gilt: (**„rote“ Regel**)*

1. *Gibt es zu $e \in E$ einen Kreis C in G, der e enthält und $w(e) > w(e')$ für alle $e' \in C \setminus \{e\}$ erfüllt, dann kommt e in keinem minimalen Spannbaum vor.*
2. *Ist $C_1 = e_1, \ldots, e_k$ ein Kreis in G und $w(e_i) = \max\{w(e_j);\ 1 \leq j \leq k\}$, dann gibt es einen minimalen Spannbaum, der e_i nicht enthält.*

Beweis:

1. Nehmen wir an, dass es einen minimalen Spannbaum T gibt, der $e = \{v_1, v_2\}$ enthält. Wenn wir e aus T entfernen, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2 mit $v_i \in T_i$, $i = 1, 2$. Da aber e auf einem Kreis in G liegt, muss es einen Weg von v_1 nach v_2 geben, der e nicht benützt. Mithin gibt es eine Kante $\hat{e} \neq e$ auf diesem Weg, die einen Knoten in T_1 mit T_2 verbindet. Verbinden wir T_1 und T_2 entlang $\hat{e}$, so erhalten wir einen von T verschiedenen Spannbaum $\hat{T}$. Wegen $w(\hat{e}) < w(e)$ folgt $w(\hat{T}) < w(T)$, im Widerspruch zur Minimalität von T.

Beweis (Forts.):

2. Wir nehmen an, e_i liege in jedem minimalen Spannbaum (MSB) von G, und zeigen die Behauptung durch Widerspruch.

   Sei T ein beliebiger MSB von G. Entfernen wir e_i aus T, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2. Da e_i auf einem Kreis $C_1 = e_1, \ldots, e_k$ in G liegt, können wir wie zuvor e_i durch eine Kante e_j des Kreises C_1 ersetzen, die T_1 und T_2 verbindet. Dadurch erhalten wir einen von T verschiedenen Spannbaum $\tilde{T}$, der e_i nicht enthält. Da nach Voraussetzung $w(e_j) \leq w(e_i)$ gilt, folgt $w(\tilde{T}) \leq w(T)$ (und sogar $w(\tilde{T}) = w(T)$, da T nach Annahme ein MSB ist). Also ist $\tilde{T}$ ein MSB von G, der e_i nicht enthält, im Widerspruch zur Annahme, e_i liege in *jedem* MSB von G.

□

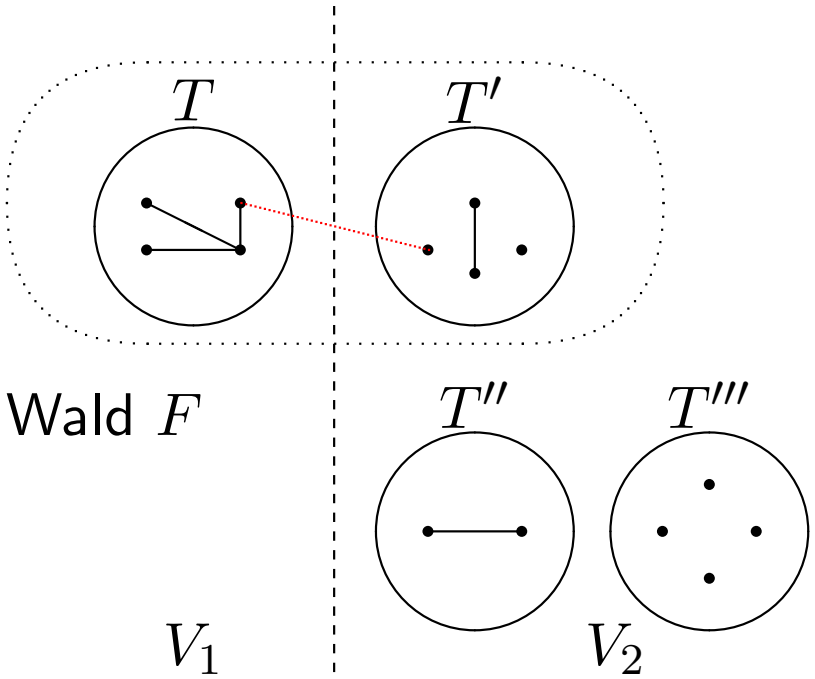
Literatur

Robert E. Tarjan:
Data Structures and Network Algorithms
SIAM CBMS-NSF Regional Conference Series in Applied Mathematics Bd. 44 (1983)

7.2 Generischer minimaler Spannbaum-Algorithmus

Initialisiere Wald F von Bäumen, jeder Baum ist ein singulärer Knoten
(jedes $v \in V$ bildet einen Baum)
while Wald F mehr als einen Baum enthält **do**
    wähle einen Baum $T \in F$ aus
    bestimme eine leichteste Kante $e = \{v, w\}$ aus T heraus
    sei $v \in T$, $w \in T'$
    vereinige T und T', füge e zum minimalen Spannbaum hinzu
od

Generischer MST-Algorithmus



7.3 Kruskal's Algorithmus

algorithm Kruskal $(G, w) :=$
  sortiere die Kanten nach aufsteigendem Gewicht in eine Liste L
  initialisiere Wald $F = \{T_i, i = 1, \ldots, n\}$, mit $T_i = \{v_i\}$
  MSB:=$\emptyset$
  for $i := 1$ **to** length(L) **do**
    $\{v, w\} := L_i$
    $x :=$ Baum $\in F$, der v enthält; **co** $x :=$*Find*(v) **oc**
    $y :=$ Baum $\in F$, der w enthält; **co** $y :=$*Find*(w) **oc**
    if $x \neq y$ **then**
      MSB:=MSB $\cup \{\{v, w\}\}$
      Union(x, y) **co** gewichtete Vereinigung **oc**
    fi
  od

Korrektheit: Falls die Gewichte eindeutig sind ($w(\cdot)$ injektiv), folgt die Korrektheit direkt mit Hilfe der "blauen" und "roten" Regel.
Ansonsten Induktion über die Anzahl $|V|$ der Knoten:
Ind. Anfang: $|V|$ klein: $\surd$

Sei $r \in \mathbb{R}$, $E_r := \{e \in E; w(e) < r\}$.
Es genügt zu zeigen:
Sei $T_1, \ldots, T_k$ ein minimaler Spannwald für $G_r := \{V, E_r\}$ (d.h., wir betrachten nur Kanten mit Gewicht $< r$). Sei weiter T ein MSB von G, dann gilt die

Hilfsbehauptung: Die Knotenmenge eines jeden T_i induziert in T einen zusammenhängenden Teilbaum, dessen Kanten alle Gewicht $< r$ haben.

Beweis der Hilfsbehauptung:

Sei $T_i =: (V_i, E_i)$. Wir müssen zeigen, dass V_i in T einen zusammenhängenden Teilbaum induziert. Seien $u, v \in V_i$ zwei Knoten, die in T_i durch eine Kante e verbunden sind. Falls der Pfad in T zwischen u und v auch Knoten $\notin V_i$ enthält (also der von V_i induzierte Teilgraph von T nicht zusammenhängend ist), dann enthält der in T durch Hinzufügen der Kante e entstehende Fundamentalkreis notwendigerweise auch Kanten aus $E \setminus E_r$ und ist damit gemäß der "roten" Regel nicht minimal! Da T_i zusammenhängend ist, folgt damit, dass je zwei Knoten aus V_i in T immer durch einen Pfad verbunden sind, der nur Kanten aus E_r enthält.

Zeitkomplexität: (mit $n = |V|, m = |E|$)

Sortieren	$m \log m = \mathcal{O}(m \log n)$
$\mathcal{O}(m)$ *Find*-Operationen	$\mathcal{O}(m \log n)$
$n - 1$ *Unions*	$\mathcal{O}(n \log n)$

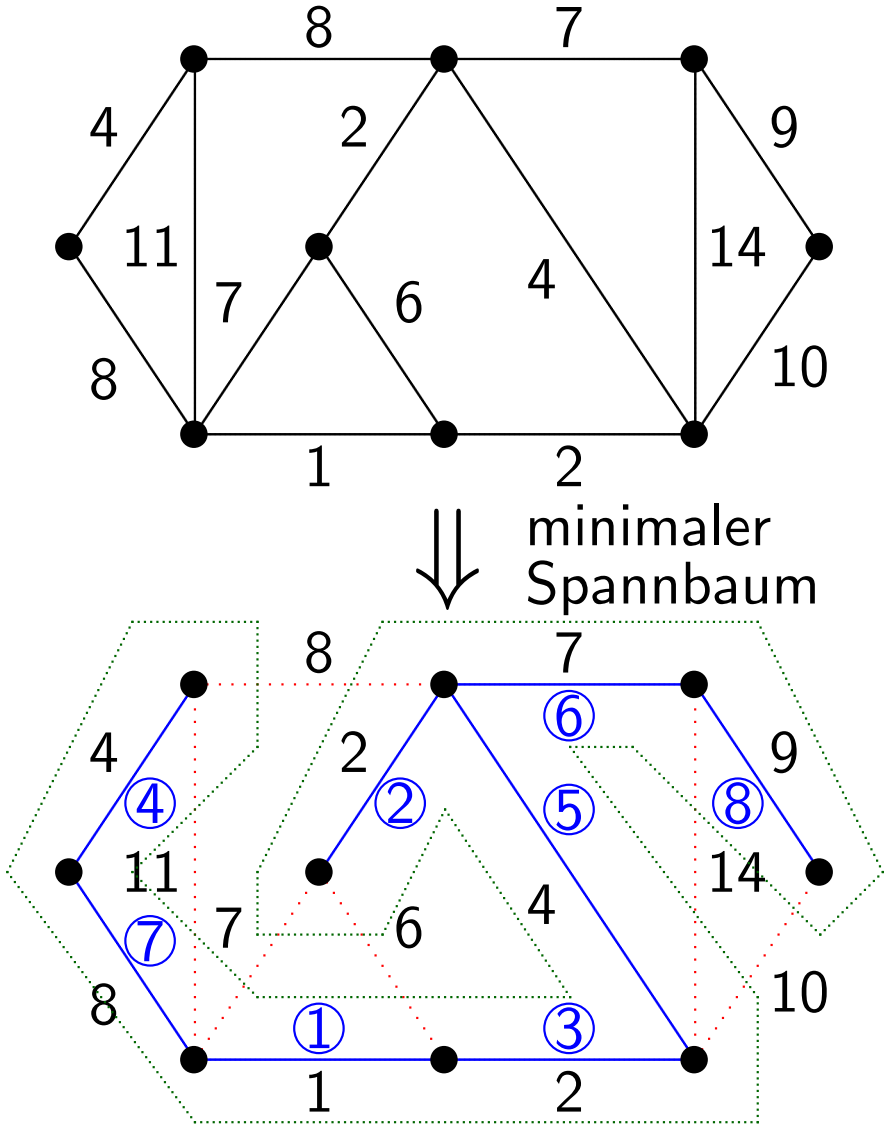
Satz 189

Kruskal's MSB-Algorithmus hat die Zeitkomplexität $\mathcal{O}((m + n) \log n)$.

Beweis:

s.o.

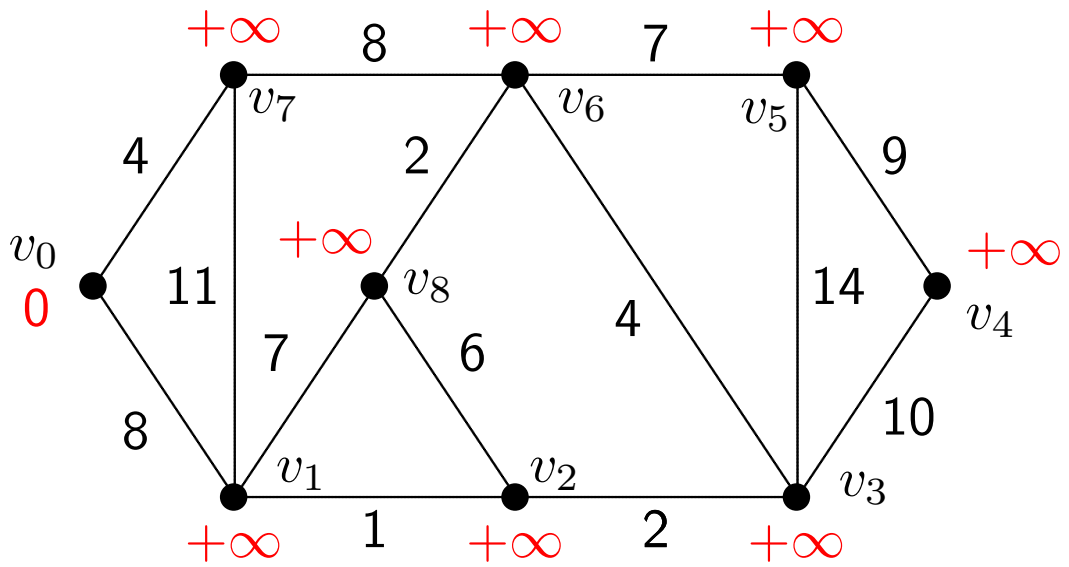
Beispiel 190 (Kruskals Algorithmus)



7.4 Prim's Algorithmus

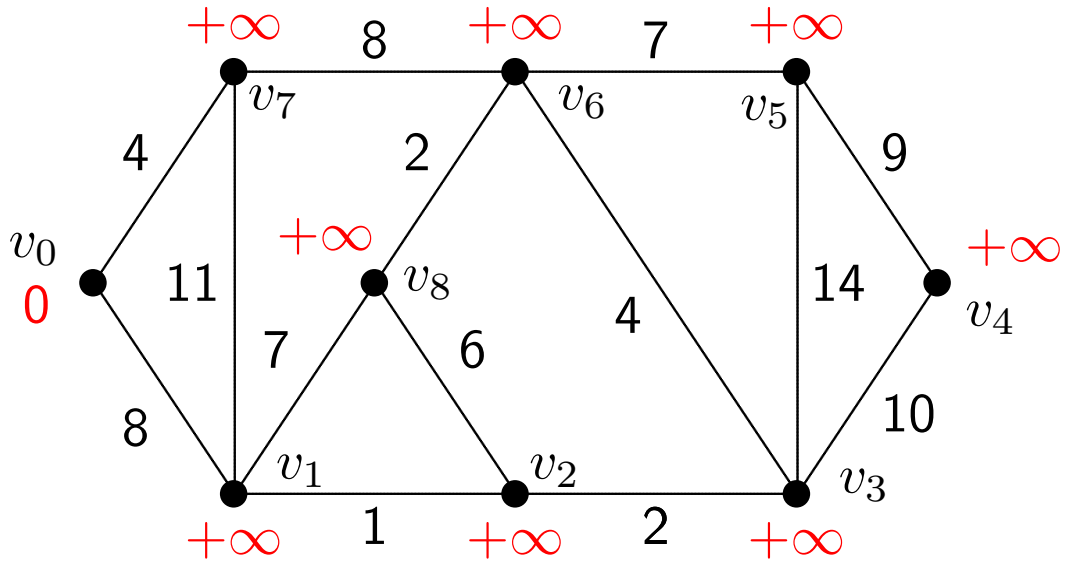
algorithm PRIM-MSB $(G, w) :=$
  initialisiere Priority Queue PQ mit Knotenmenge V und
    Schlüssel $+\infty$, $\forall v \in V$
  wähle Knoten r als Wurzel (beliebig)
  Schlüssel $k[r] := 0$
  Vorgänger$[r] :=$ **nil**
  while $PQ \neq \emptyset$ **do**
    $u :=$ *ExtractMin*(PQ)
    for alle Knoten v, die in G zu u benachbart sind **do**
      if $v \in PQ$ **and** $w(\{u, v\}) < k[v]$ **then**
        Vorgänger$[v] := u$
        DecreaseKey$(PQ, v, w(\{u, v\}))$
      fi
    od
  od

Beispiel 191 (Prim's Algorithmus)

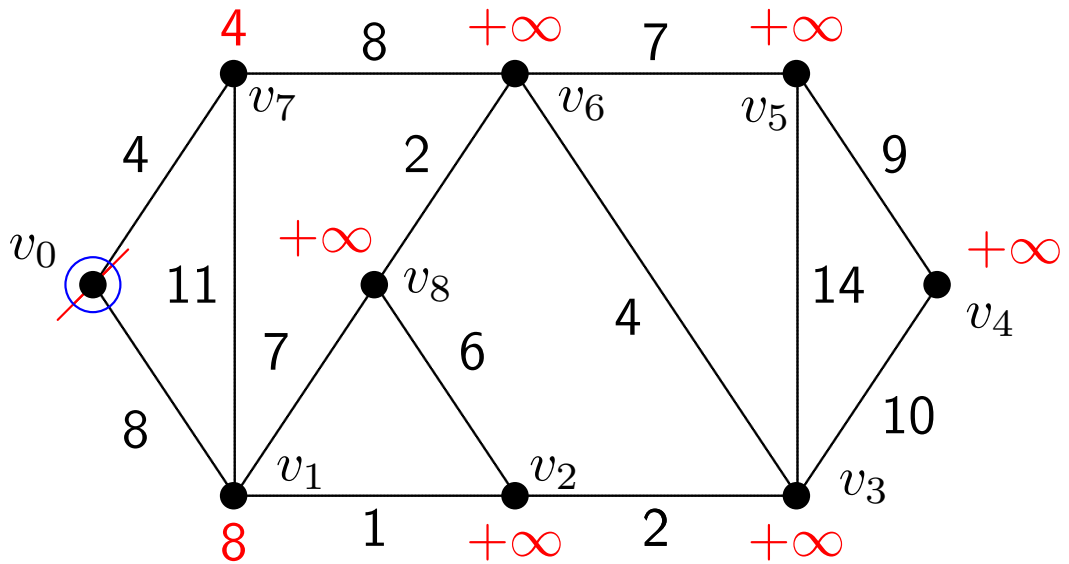


Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten u: ◉
Startknoten: r $(= v_0)$

Beispiel 191 (Prim's Algorithmus)

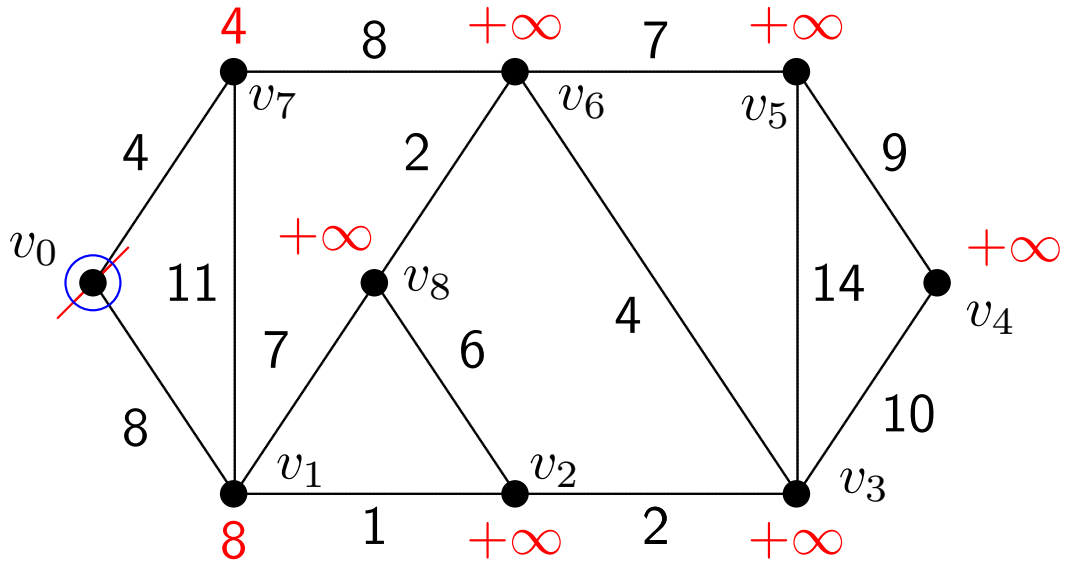


Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten u: ◉
Startknoten: r $(= v_0)$

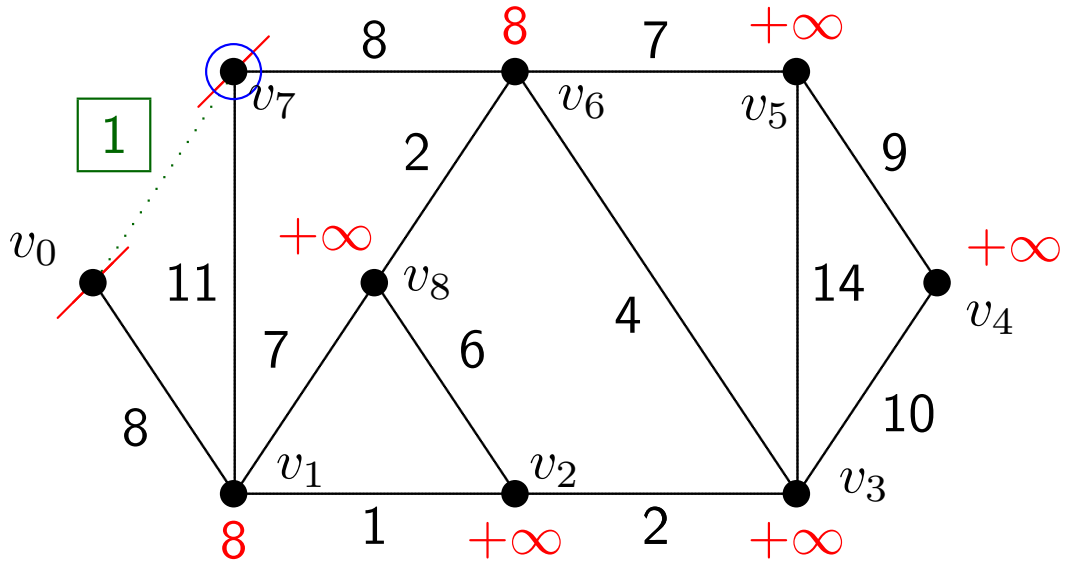


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_1 = 8, v_7 = 4)$

Beispiel 191 (Prim's Algorithmus)

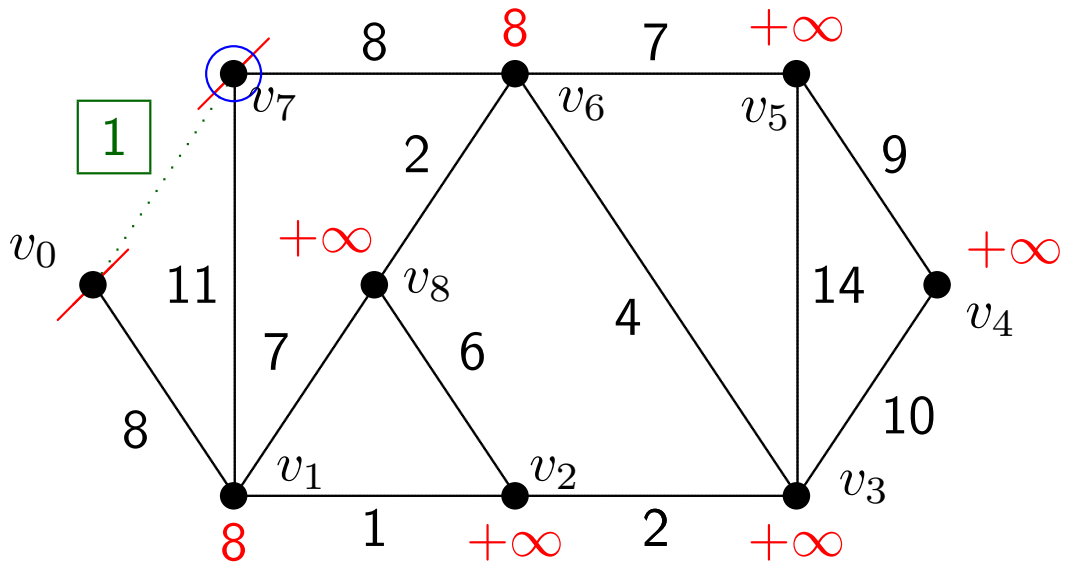


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $w(\{u,v\}) <$ Schlüssel$[v]$:
$(v_1 = 8, v_7 = 4)$

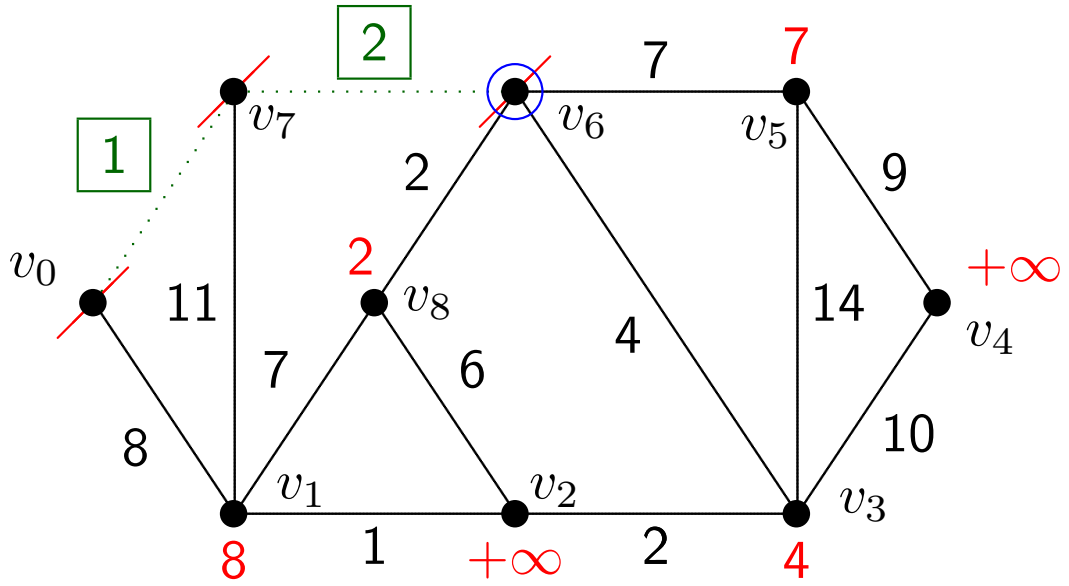


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $w(\{u,v\}) <$ Schlüssel$[v]$:
$(v_6 = 8)$

Beispiel 191 (Prim's Algorithmus)

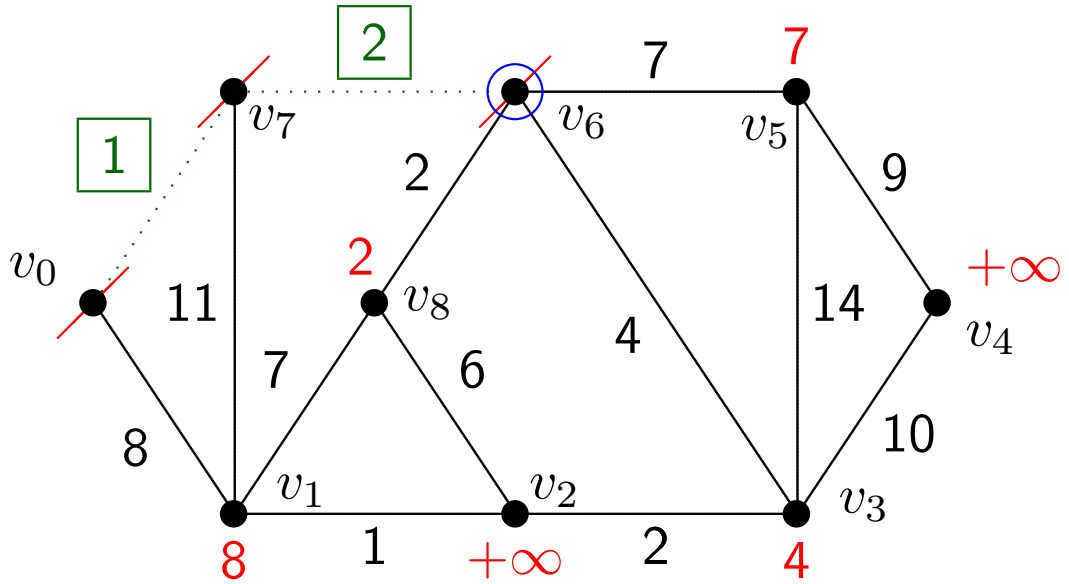


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_6 = 8)$

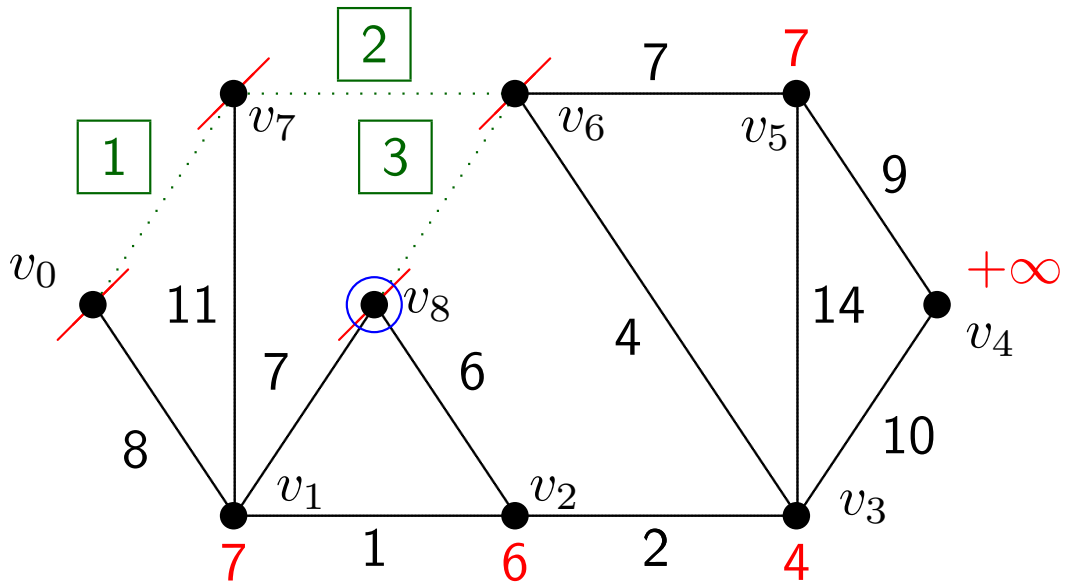


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_3 = 4, v_5 = 7, v_8 = 2)$

Beispiel 191 (Prim's Algorithmus)

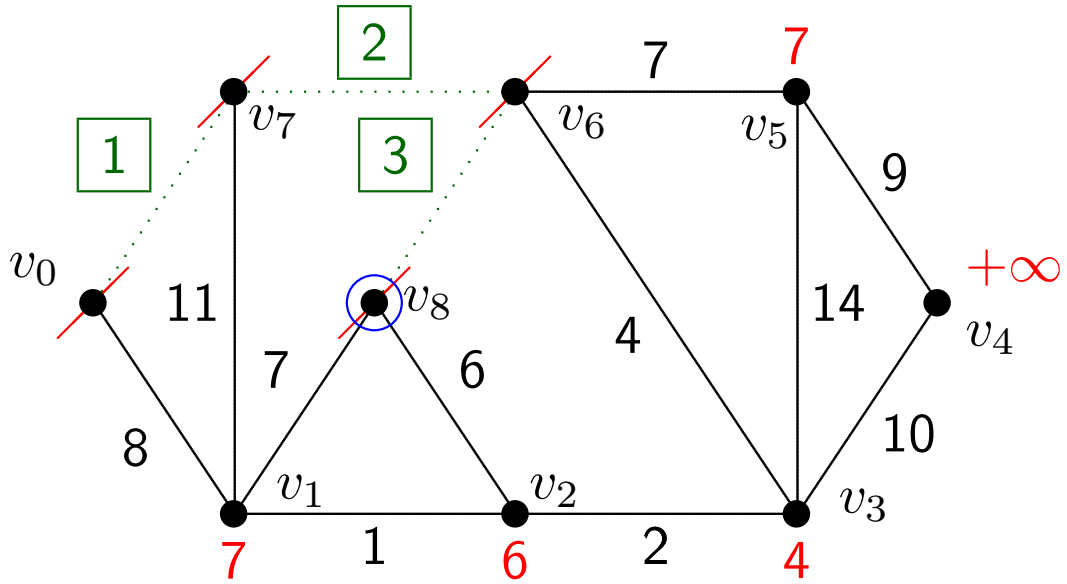


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_3 = 4, v_5 = 7, v_8 = 2)$

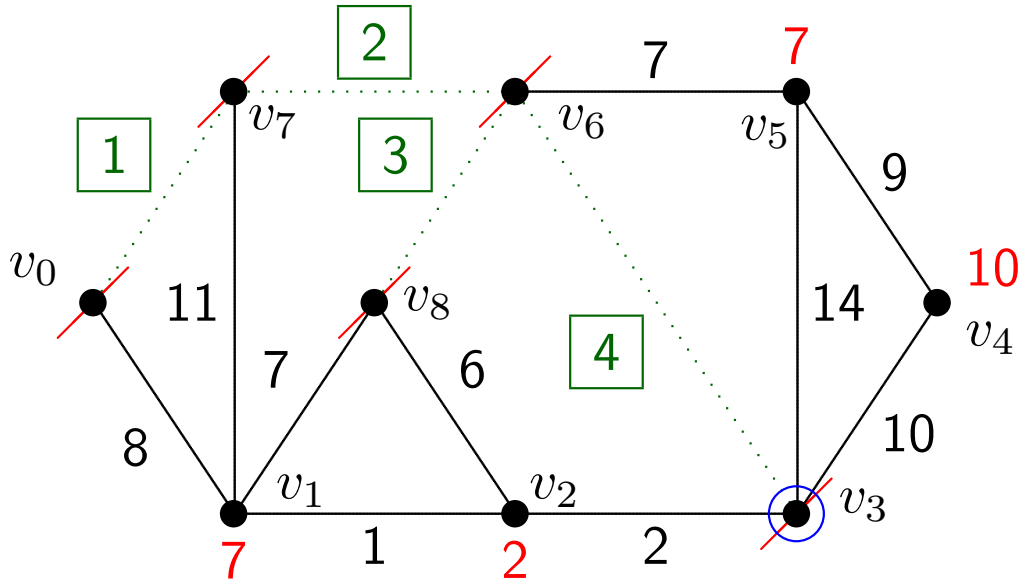


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_1 = 7, v_2 = 6)$

Beispiel 191 (Prim's Algorithmus)

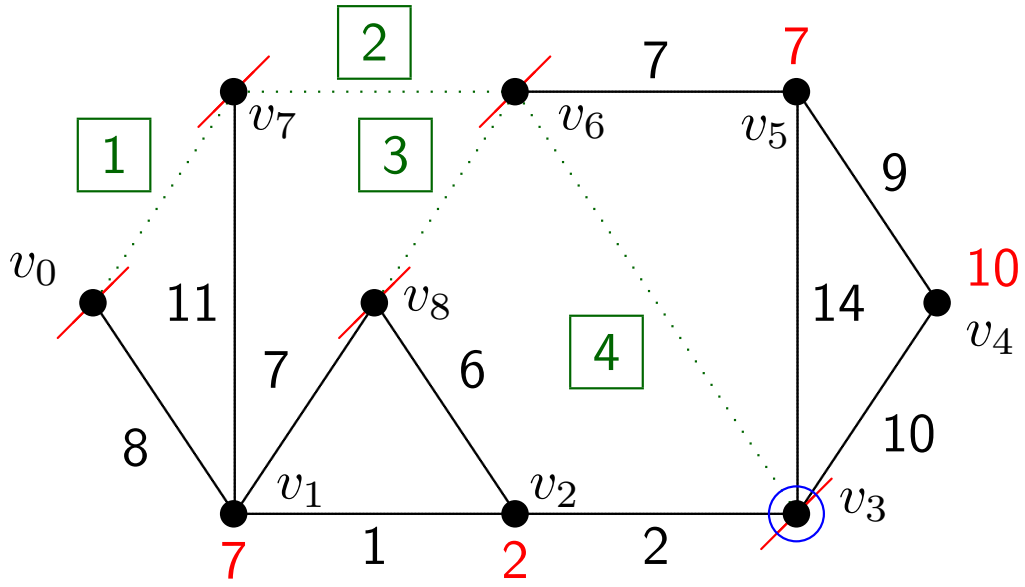


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_1 = 7, v_2 = 6)$

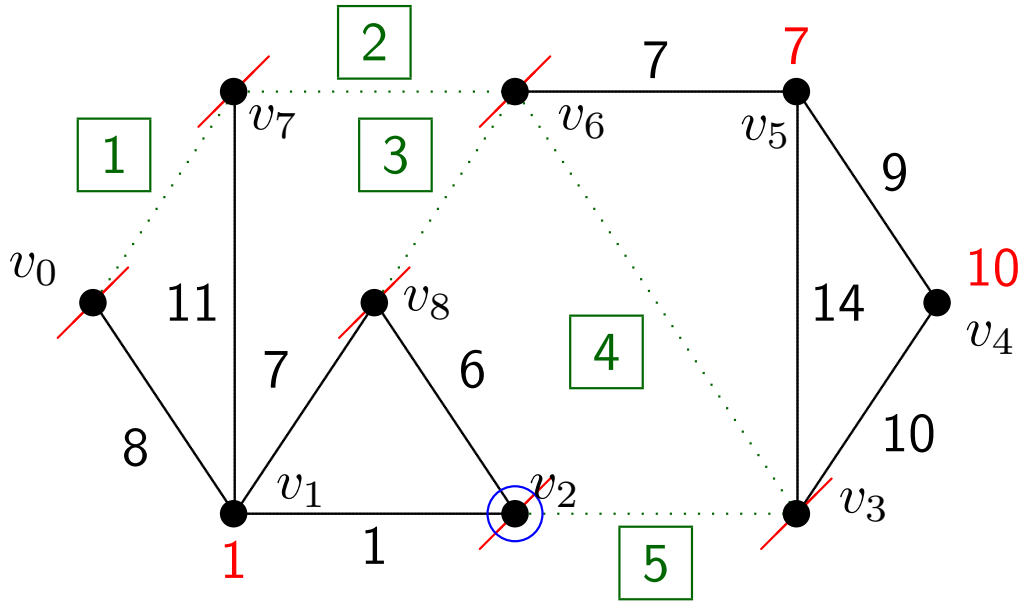


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_2 = 2, v_4 = 10)$

Beispiel 191 (Prim's Algorithmus)

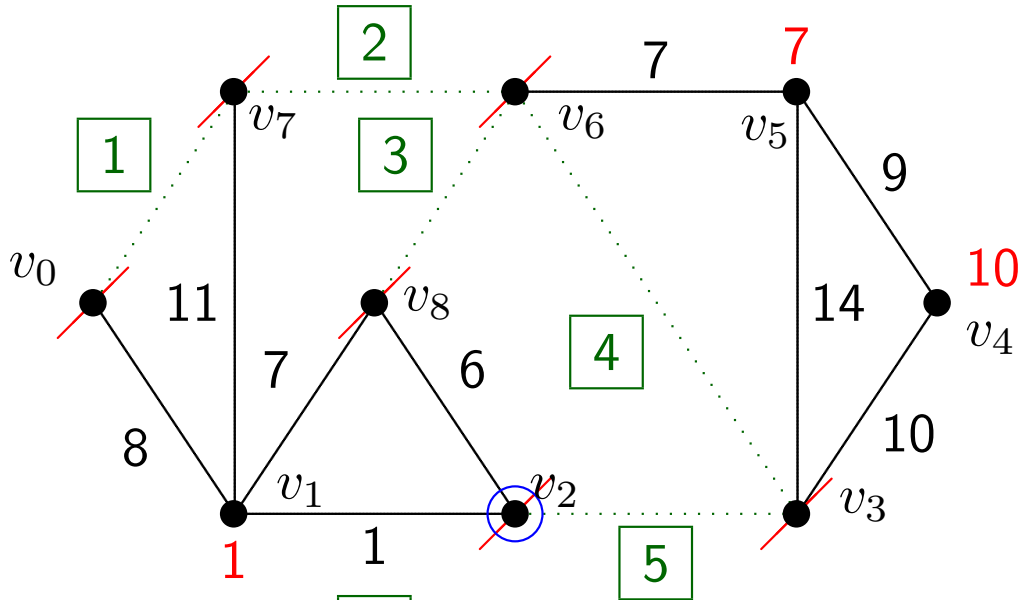


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_2 = 2, v_4 = 10)$

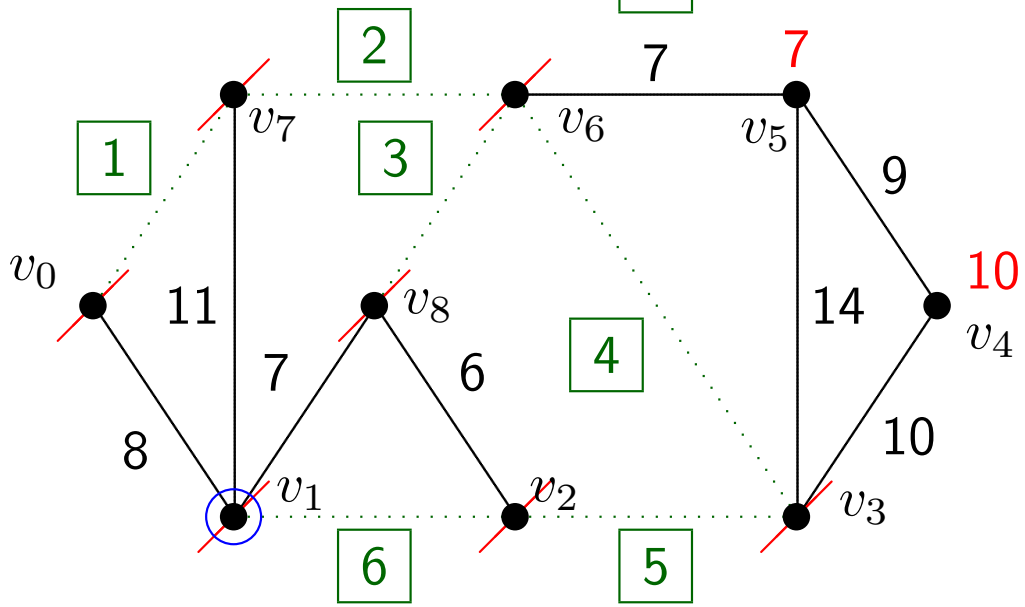


suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_1 = 1)$

Beispiel 191 (Prim's Algorithmus)

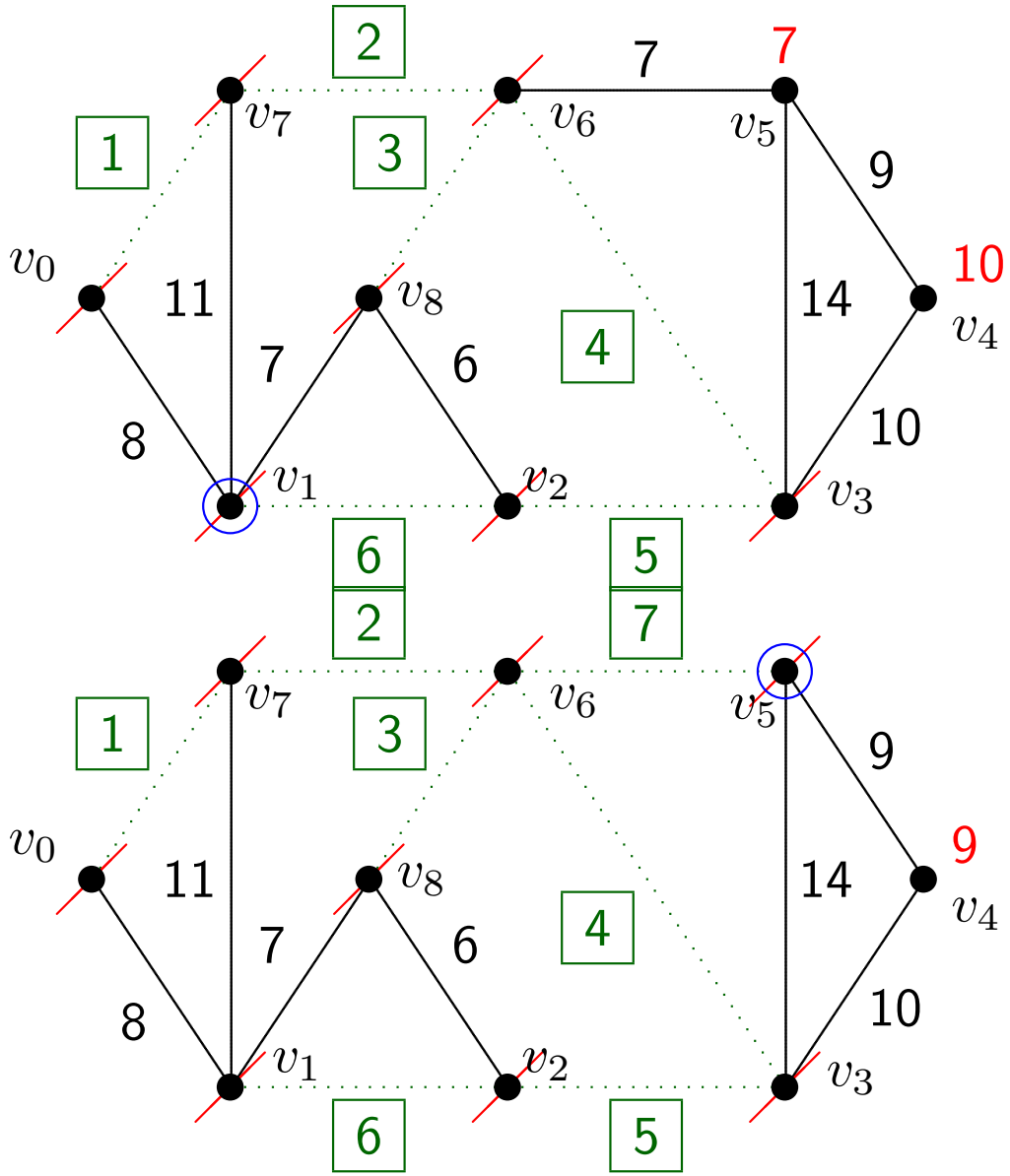


suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $\mathrm{w}(\{u,v\}) <$ Schlüssel$[v]$:
$(v_1 = 1)$



suche $u :=$ FindMin(PQ) und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit $\mathrm{w}(\{u,v\}) <$ Schlüssel$[v]$:
solche Nachbarn existieren nicht

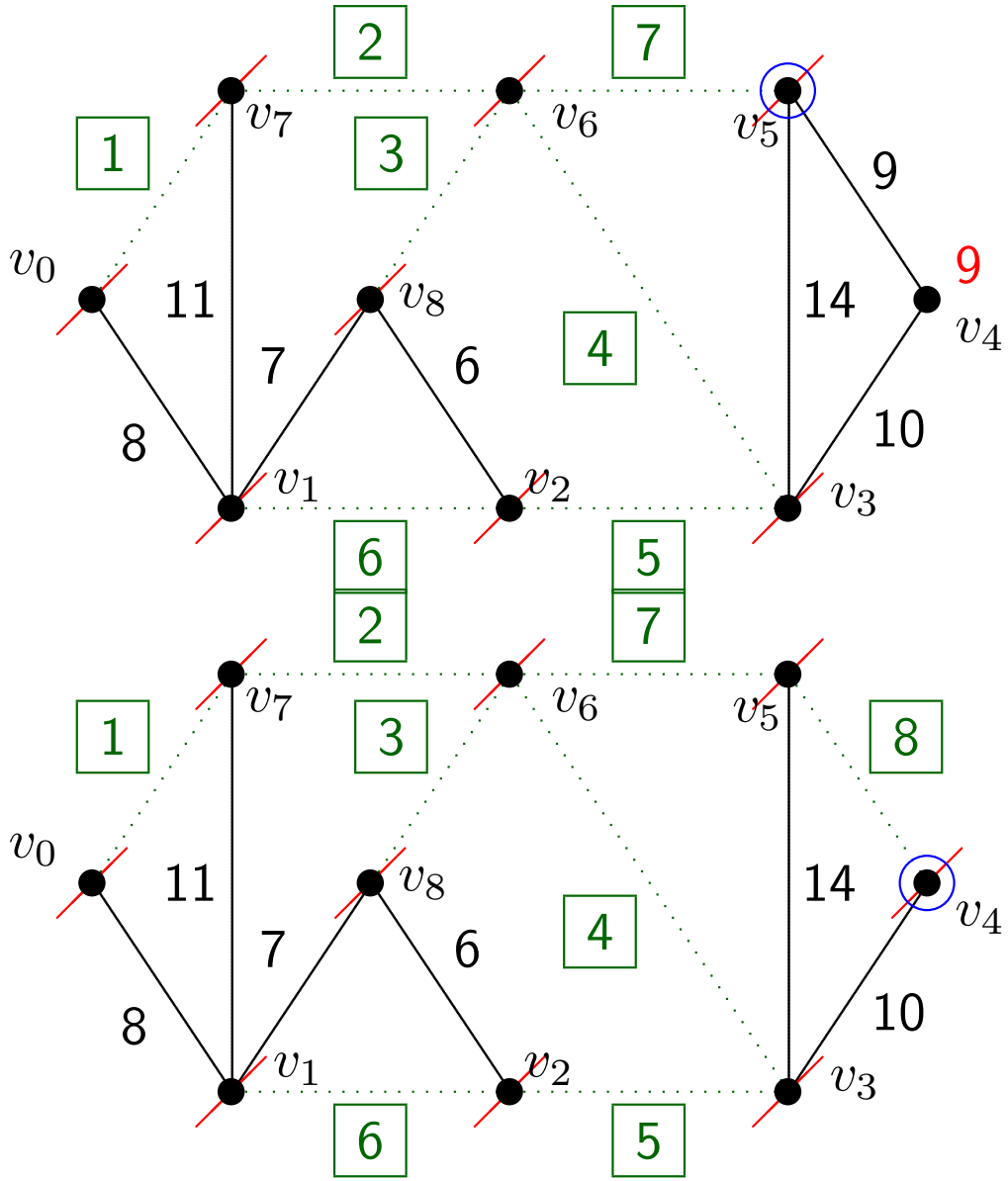
Beispiel 191 (Prim's Algorithmus)



suche $u := \text{FindMin}(PQ)$
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
$w(\{u, v\}) <$ Schlüssel$[v]$:
solche Nachbarn existieren nicht

suche $u := \text{FindMin}(PQ)$
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
$w(\{u, v\}) <$ Schlüssel$[v]$:
$(v_4 = 9)$

Beispiel 191 (Prim's Algorithmus)



suche $u :=$ FindMin(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w$(\{u, v\}) <$ Schlüssel$[v]$:
$(v_4 = 9)$

Endzustand:
suche $u :=$ FindMin(PQ)
und entferne u aus PQ,
damit ist PQ leer und der Algorithmus beendet

Korrektheit: ist klar.

Zeitkomplexität:

- n *ExtractMin*
- $\mathcal{O}(m)$ sonstige Operationen inclusive *DecreaseKey*

Implementierung der Priority Queue mittels Fibonacci-Heaps:

Initialisierung	$\mathcal{O}(n)$
ExtractMins	$\mathcal{O}(n \log n)$ $(\leq n$ Stück$)$
DecreaseKeys	$\mathcal{O}(m)$ $(\leq m$ Stück$)$
Sonstiger Overhead	$\mathcal{O}(m)$

Satz 192

Sei $G = (V, E)$ ein ungerichteter Graph (zusammenhängend, einfach) mit Kantengewichten w. Prim's Algorithmus berechnet, wenn mit Fibonacci-Heaps implementiert, einen minimalen Spannbaum von (G, w) in Zeit $\mathcal{O}(m + n \log n)$ (wobei $n = |V|,\ m = |E|$). Dies ist für $m = \Omega(n \log n)$ asymptotisch optimal.

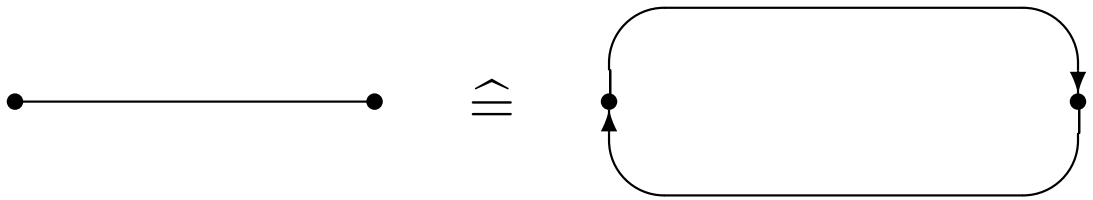
Beweis:

s.o.

8. Kürzeste Pfade

8.1 Grundlegende Begriffe

Betrachte Digraph $G = (V, A)$ oder Graph $G = (V, E)$.



Kante im ungerichteten Graph

entgegengesetzte (antiparallele) Kanten

Distanzfunktion: $\mathrm{d} : A \longrightarrow \mathbb{R}_0^+ \cup \{+\infty\}$ (bzw. $\longrightarrow \mathbb{R} \cup \{+\infty\}$)
O.B.d.A.: $A = V \times V$, $\mathrm{d}(x, y) = +\infty$ für Kanten, die eigentlich nicht vorhanden sind.
$\mathrm{dis}(v, w) :=$ Länge eines kürzesten Pfades von v nach w
$\in \mathbb{R}_0^+ \cup \{+\infty\}$.

Arten von Kürzeste-Pfade-Problemen:

1. single-pair-shortest-path (spsp). Beispiel: Kürzeste Entfernung von München nach Frankfurt.
2. single-source-shortest-path: gegeben G, d und $s \in V$, bestimme für alle $v \in V$ die Länge eines kürzesten Pfades von s nach v (bzw. einen kürzesten Pfad von s nach v) (sssp). Beispiel: Kürzeste Entfernung von München nach allen anderen Großstädten.
3. all-pairs-shortest-path (apsp). Beispiel: Kürzeste Entfernung zwischen allen Großstädten.

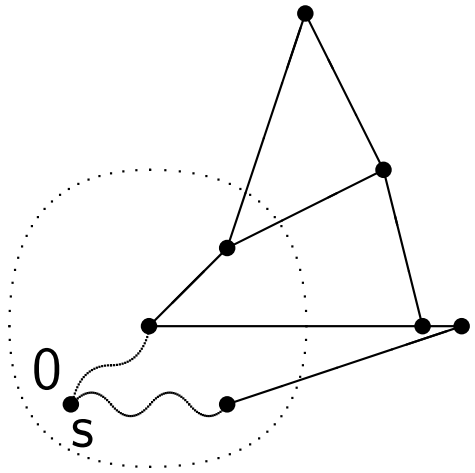
Bemerkung: Wir kennen keinen Algorithmus, der das single-pair-shortest-path berechnet, ohne nicht gleichzeitig (im worst-case) das single-source-shortest-path-Problem zu lösen.

8.2 Das single-source-shortest-path-Problem

Zunächst nehmen wir an, dass $\mathrm{d} \geq 0$ ist. Alle kürzesten Pfade von a nach b sind o.B.d.A. einfache Pfade.

8.2.1 Dijkstra's Algorithmus

Gegeben: $G = (V, A)$, $(A = V \times V)$, Distanzfunktion $\mathrm{d} : A \to \mathbb{R}_0^+ \cup \{+\infty\}$, Startknoten s, G durch Adjazenzlisten dargestellt.



algorithm sssp:=

$S := \{s\}$; $\mathrm{dis}[s] := 0$; initialisiere eine Priority Queue PQ, die alle Knoten $v \in V \setminus \{s\}$ enthält mit Schlüssel $\mathrm{dis}[v] := \mathrm{d}(s, v)$

for alle $v \in V - \{s\}$ **do** from$[v] := s$ **od**

while $S \neq V$ **do**

$v :=$ *ExtractMin*(PQ)

$S := S \cup \{v\}$

for alle $w \in V \setminus S$, $\mathrm{d}(v, w) < \infty$ **do**

if $\mathrm{dis}[v] + \mathrm{d}(v, w) < \mathrm{dis}[w]$ **then**

DecreaseKey$(w, \mathrm{dis}[v] + \mathrm{d}(v, w))$

co *DecreaseKey* aktualisiert $\mathrm{dis}[w]$ **oc**

from$[w] := v$

fi

od

od

Seien $n = |V|$ und $m =$ die Anzahl der wirklichen Kanten in G.

Laufzeit (mit Fibonacci-Heaps):

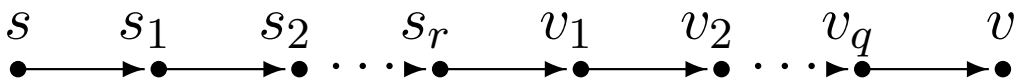
Initialisierung:	$\mathcal{O}(n)$
ExtractMin:	$n \cdot \mathcal{O}(\log n)$
Sonstiger Aufwand:	$m \cdot \mathcal{O}(1)$ (z.B. *DecreaseKey*)

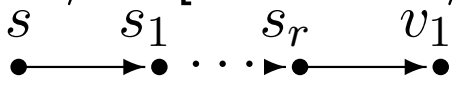
$\Rightarrow$ Zeitbedarf also: $\boxed{\mathcal{O}(m + n \log n)}$

Korrektheit: Wir behaupten, dass in dem Moment, in dem ein $v \in V \setminus \{s\}$ Ergebnis der *ExtractMin* Operation ist, der Wert $\mathrm{dis}[v]$ des Schlüssels von v gleich der Länge eines kürzesten Pfades von s nach v ist.

Beweis:

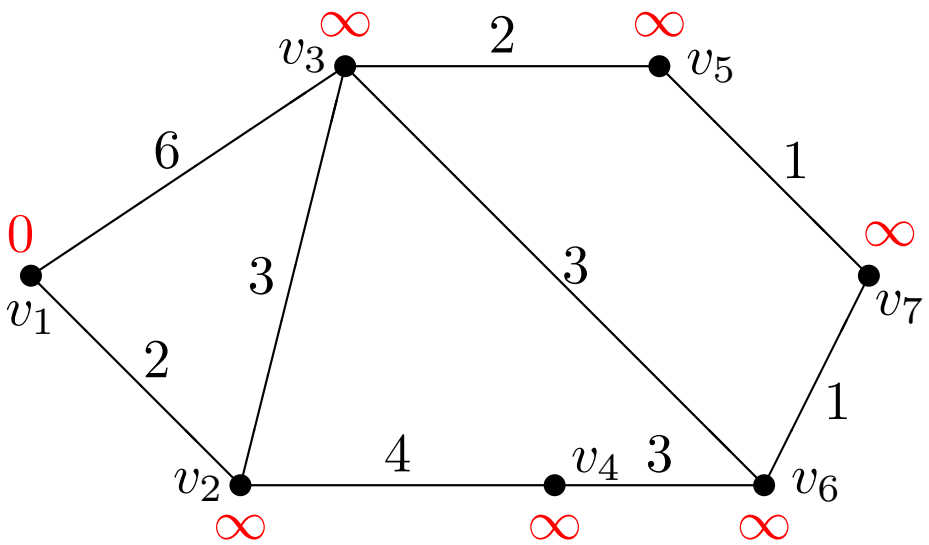
[durch Widerspruch] Sei $v \in V \setminus \{s\}$ der erste Knoten, für den diese Behauptung nicht stimmt, und sei



ein kürzester Pfad von s nach v, mit einer Länge $< \mathrm{dis}[v]$. Dabei sind $s_1, \ldots, s_r \in S$, $v_1 \notin S$ [$r = 0$ und/oder $q = 0$ ist möglich]. Betrachte den Pfad 

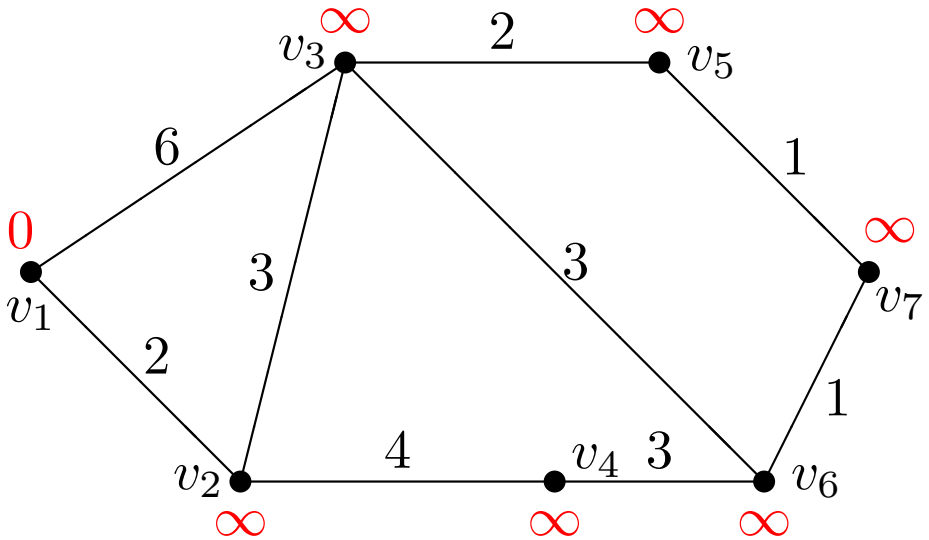
; seine Länge ist $< \mathrm{dis}[v]$, für $q \geq 1$ (ebenso für $q = 0$) ist also $\mathrm{dis}[v_1] < \mathrm{dis}[v]$, im Widerspruch zur Wahl von v.

Beispiel 193 (Dijkstra's Algorithmus)

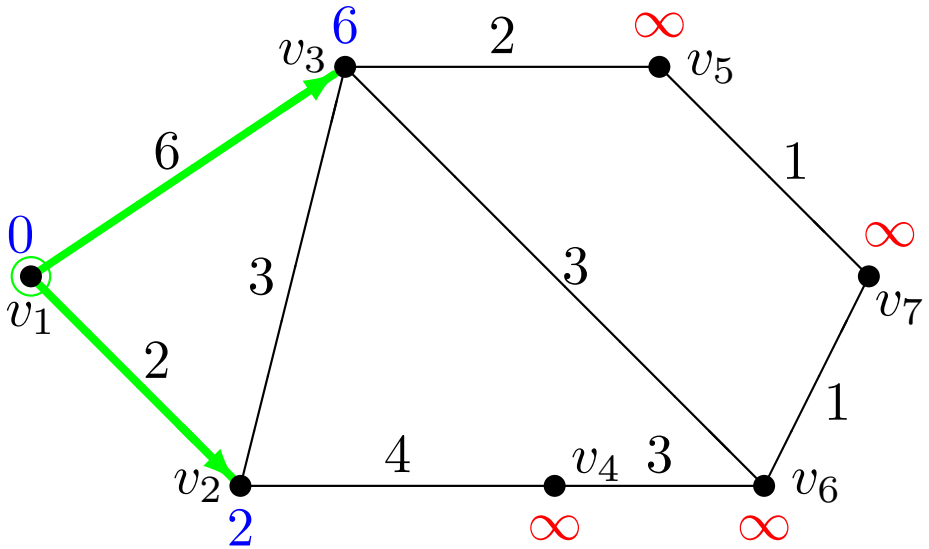


gegeben Graph G;
v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze $\mathrm{dis}[v_1] = 0$;
setze $\mathrm{dis}[\text{Rest}] = +\infty$;

Beispiel 193 (Dijkstra's Algorithmus)

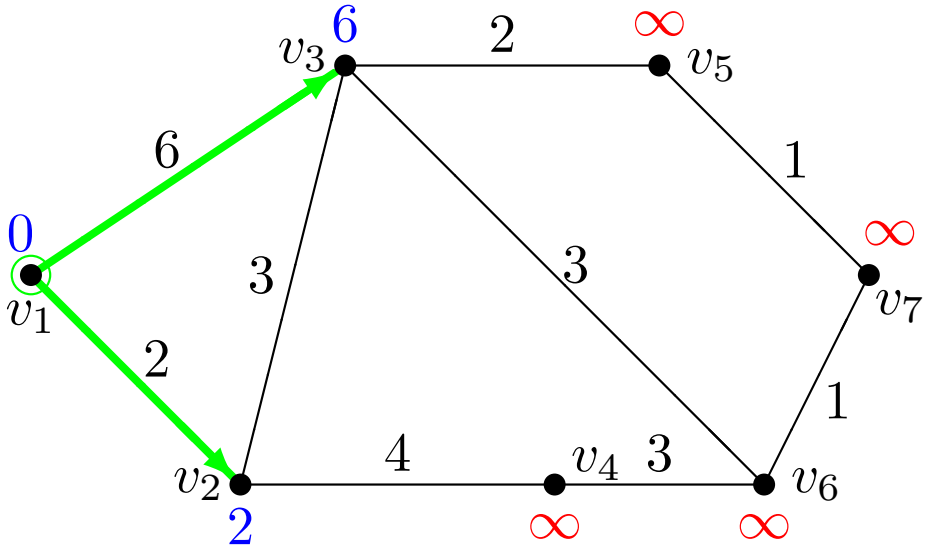


gegeben Graph G;
v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze $\mathrm{dis}[v_1] = 0$;
setze $\mathrm{dis}[\mathrm{Rest}] = +\infty$;

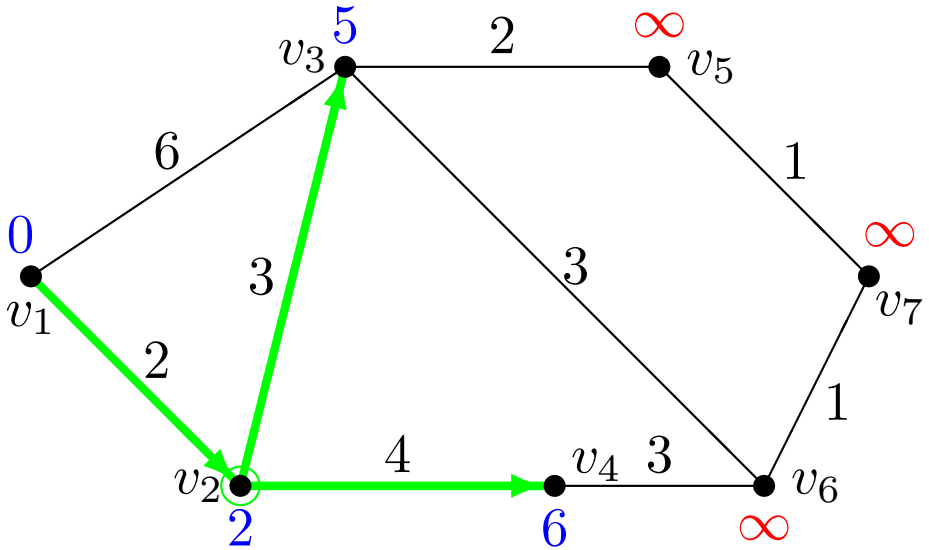


setze $\mathrm{dis}[v_2] = 2$;
markiere (v_1, v_2);
setze $\mathrm{dis}[v_3] = 6$;
markiere (v_1, v_3);
setze v_2 als Bezugsknoten,
da $\mathrm{dis}[v_2]$ minimal;

Beispiel 193 (Dijkstra's Algorithmus)

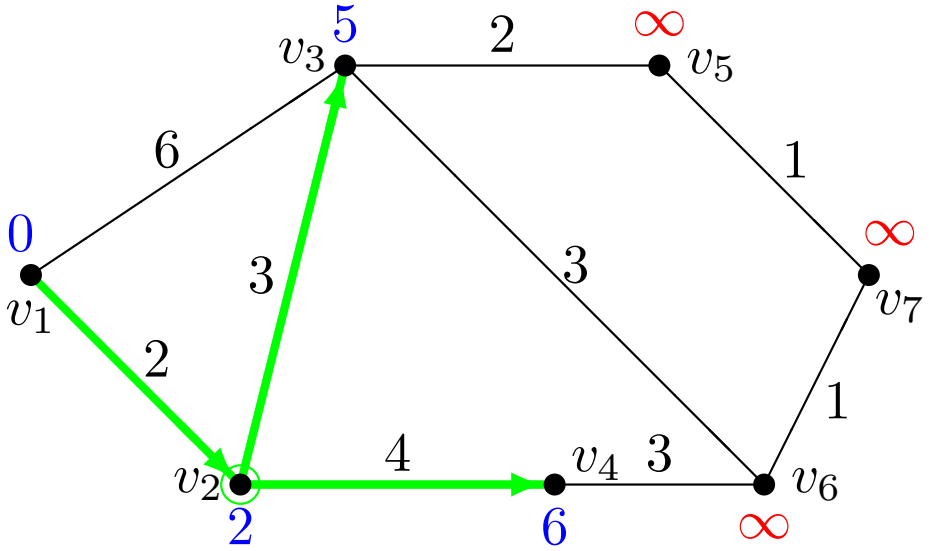


setze $\mathrm{dis}[v_2] = 2$;
markiere (v_1, v_2);
setze $\mathrm{dis}[v_3] = 6$;
markiere (v_1, v_3);
setze v_2 als Bezugsknoten,
da $\mathrm{dis}[v_2]$ minimal;

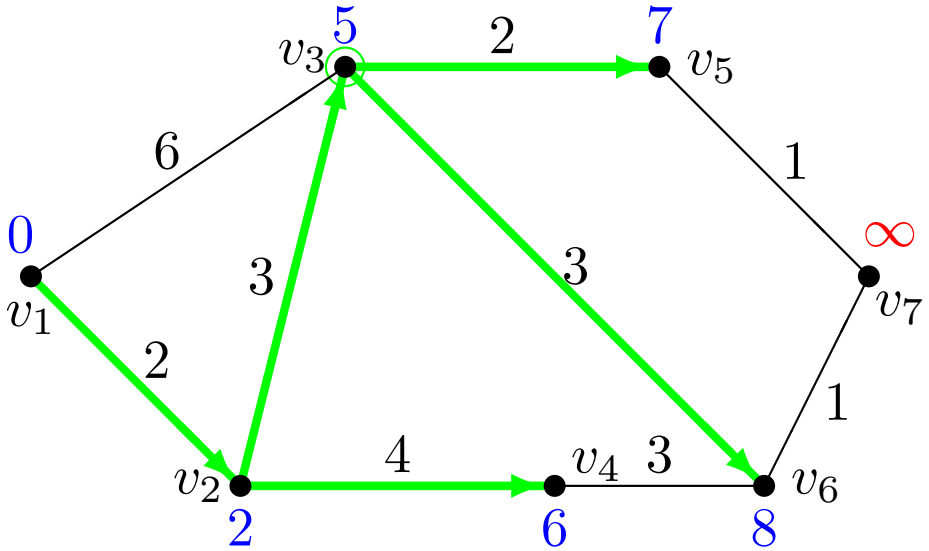


setze $\mathrm{dis}[v_3] = 2 + 3 = 5$;
markiere (v_2, v_3);
unmarkiere (v_1, v_3);
setze $\mathrm{dis}[v_4] = 2 + 4 = 6$;
markiere (v_2, v_4);
setze v_3 als Bezugsknoten,
da $\mathrm{dis}[v_3]$ minimal;

Beispiel 193 (Dijkstra's Algorithmus)

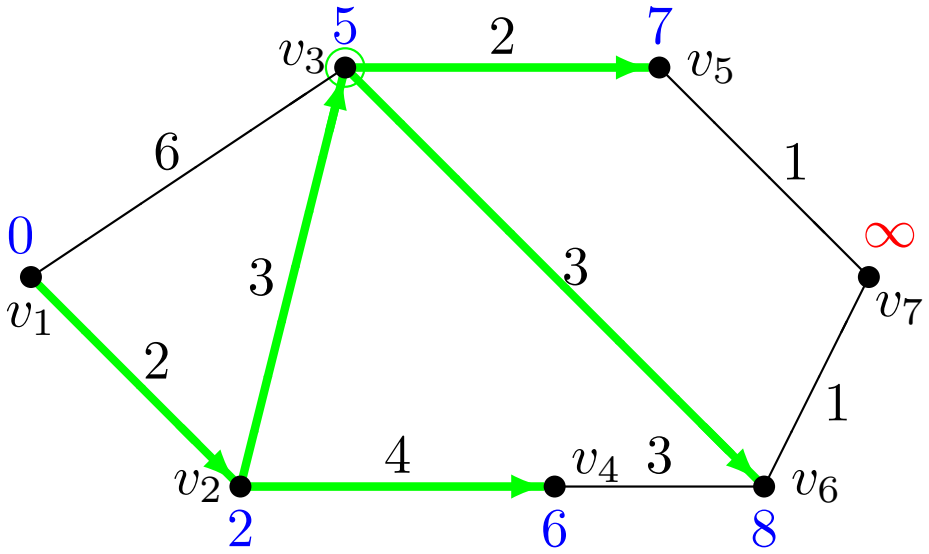


setze $\mathrm{dis}[v_3] = 2 + 3 = 5$;
markiere (v_2, v_3);
unmarkiere (v_1, v_3);
setze $\mathrm{dis}[v_4] = 2 + 4 = 6$;
markiere (v_2, v_4);
setze v_3 als Bezugsknoten,
da $\mathrm{dis}[v_3]$ minimal;

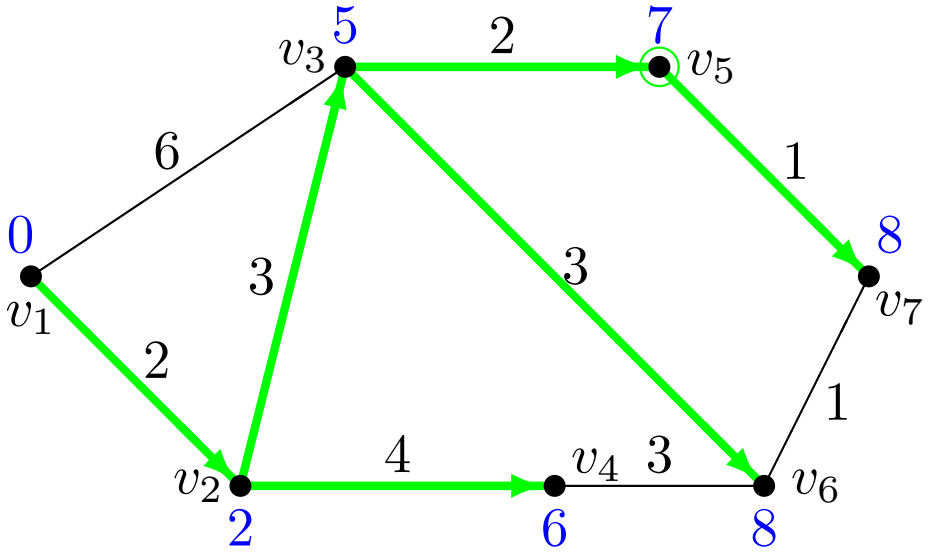


setze $\mathrm{dis}[v_5] = 5 + 2 = 7$;
markiere (v_3, v_5);
setze $\mathrm{dis}[v_6] = 5 + 3 = 8$;
markiere (v_3, v_6);
setze v_4, dann v_5
als Bezugsknoten;

Beispiel 193 (Dijkstra's Algorithmus)



setze $\mathrm{dis}[v_5] = 5 + 2 = 7$;
markiere (v_3, v_5);
setze $\mathrm{dis}[v_6] = 5 + 3 = 8$;
markiere (v_3, v_6);
setze v_4, dann v_5
als Bezugsknoten;



setze $\mathrm{dis}[v_7] := 7 + 1 = 8$;
markiere (v_5, v_7);
alle Knoten wurden erreicht:
$\Rightarrow$ Algorithmus zu Ende

Beobachtung:

- *ExtractMin* liefert eine (schwach) monoton steigende Folge von Schlüsseln $\mathrm{dis}[\cdot]$;
- Die Schlüssel $\neq \infty$ in PQ sind stets $\leq \mathrm{dis}[v] + C$, wobei v das Ergebnis der vorangehenden *ExtractMin*-Operation (bzw. s zu Beginn) und $C := \max_{(u,w)\in A}\{\mathrm{dis}(u,w)\}$ ist.

Satz 194

Dijkstra's Algorithmus (mit Fibonacci-Heaps) löst das single-source-shortest-path-Problem in Zeit $\mathcal{O}(m + n\log n)$.

8.2.2 Bellman-Ford-Algorithmus

Wir setzen (zunächst) wiederum voraus:

$$\mathrm{d} \geq 0 \; .$$

Dieser Algorithmus ist ein Beispiel für dynamische Programmierung.

Sei $B_k[i] :=$ Länge eines kürzesten Pfades von s zum Knoten i, wobei der Pfad höchstens k Kanten enthält.
Gesucht ist $B_{n-1}[i]$ für $i = 1, \ldots, n$ (o.B.d.A. $V = \{1, \ldots, n\}$).

Initialisierung:

$$B_1[i] := \begin{cases} \mathrm{d}(s,i) & , \text{ falls } \mathrm{d}(s,i) < \infty, i \neq s \\ 0 & , \text{ falls } i = s \\ +\infty & , \text{ sonst} \end{cases}$$

Iteration:

for $k := 2$ **to** $n-1$ **do**
  for $i := 1$ **to** n **do**

$$B_k[i] := \begin{cases} 0 & , \text{falls } i = s \\ \min_{j \in N^{-1}(i)} \{B_{k-1}[i], B_{k-1}[j] + \mathrm{d}(j,i)\} & , \text{sonst} \end{cases}$$

  od
od

Bemerkung: $N^{-1}(i)$ ist die Menge der Knoten, von denen aus eine Kante zu Knoten i führt.

Korrekheit:
klar (Beweis durch vollständige Induktion)

Zeitbedarf:
Man beachte, dass in jedem Durchlauf der äußeren Schleife jede Halbkante einmal berührt wird.

Satz 195

Der Zeitbedarf des Bellman-Ford-Algorithmus ist $\mathcal{O}(n \cdot m)$.

Beweis:

s.o.

8.3 Floyd's Algorithmus für das all-pairs-shortest-path-Problem

Dieser Algorithmus wird auch als „Kleene's Algorithmus" bezeichnet. Er ist ein weiteres Beispiel für dynamische Programmierung.

Sei $G = (V, E)$ mit Distanzfunktion $\mathrm{d} : A \to \mathbb{R}_0^+ \cup \{+\infty\}$ gegeben. Sei o.B.d.A. $V = \{v_1, \ldots, v_n\}$.

Wir setzen nun

$c_{ij}^k :=$ *Länge eines kürzesten Pfades von* v_i *nach* v_j*, der als innere Knoten (alle bis auf ersten und letzten Knoten) nur Knoten aus* $\{v_1, \ldots, v_k\}$ *enthält.*

algorithm floyd :=

for alle (i,j) **do** $c_{ij}^{(0)} := \mathrm{d}(i,j)$ **od** **co** $1 \leq i,j \leq n$ **oc**

for $k := 1$ **to** n **do**

for alle (i,j), $1 \leq i,j \leq n$ **do**

$$c_{ij}^{(k)} := \min\left\{c_{ij}^{(k-1)},\ c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right\}$$

od

od

Laufzeit: $\mathcal{O}(n^3)$

Korrektheit:

Zu zeigen: $c_{ij}^{(k)}$ des Algorithmus $= c_{ij}^{k}$ (damit sind die Längen der kürzesten Pfade durch $c_{ij}^{(n)}$ gegeben).

Beweis:

Richtig für $k = 0$. Induktionsschluss: Ein kürzester Pfad von v_i nach v_j mit inneren Knoten $\in \{v_1, \ldots, v_{k+1}\}$ enthält entweder v_{k+1} gar nicht als inneren Knoten, oder er enthält v_{k+1} genau einmal als inneren Knoten. Im ersten Fall wurde dieser Pfad also bereits für $c_{ij}^{(k)}$ betrachtet, hat also Länge $= c_{ij}^{(k)}$. Im zweiten Fall setzt er sich aus einem kürzesten Pfad P_1 von v_i nach v_{k+1} und einem kürzesten Pfad P_2 von v_{k+1} nach v_j zusammen, wobei alle inneren Knoten von P_1 und $P_2 \in \{v_1, \ldots, v_k\}$ sind. Also ist die Länge des Pfades $= c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}$.

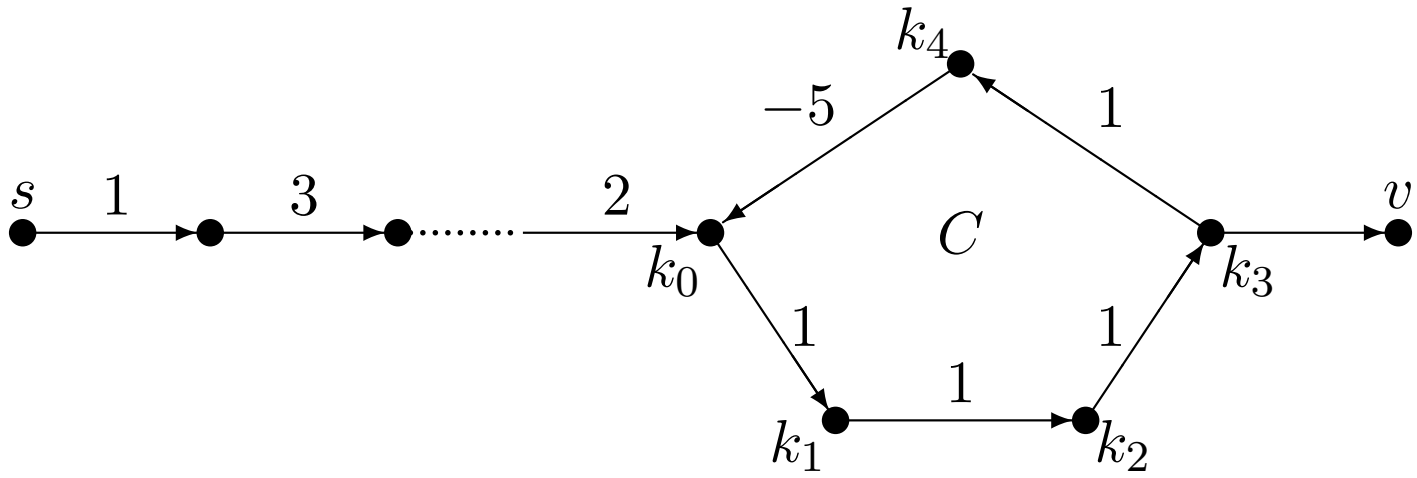
Satz 196

Floyd's Algorithmus für das all-pairs-shortest-path-Problem hat Zeitkomplexität $\mathcal{O}(n^3)$.

8.4 Digraphen mit negativen Kantengewichten

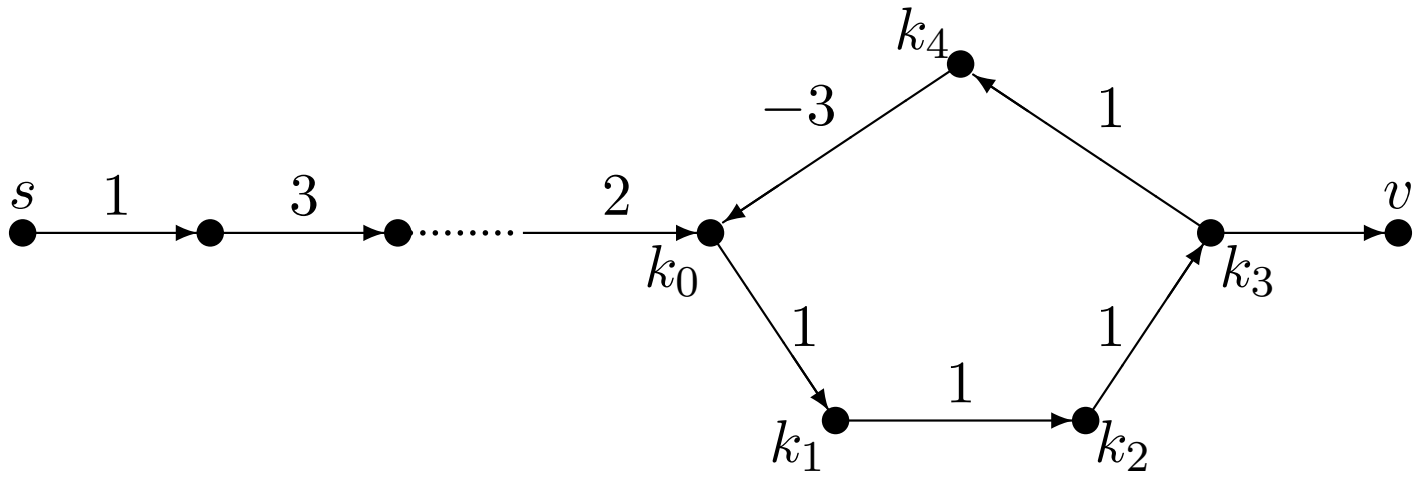
8.4.1 Grundsätzliches

Betrachte Startknoten s und einen Kreis C mit Gesamtlänge < 0.



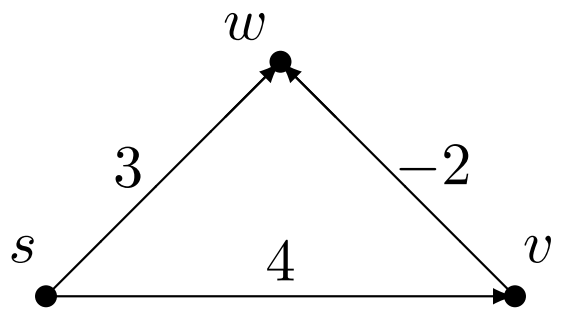
Sollte ein Pfad von s nach C und von C nach v existieren, so ist ein kürzester Pfad von s nach v nicht definiert.

Falls aber die Gesamtlänge des Kreises $C \geq 0$ ist,



dann ist der kürzeste Pfad (der dann o.B.d.A. als kreisfrei genommen werden kann) wohldefiniert. Probleme gibt es also nur dann, wenn G einen Zyklus negativer Länge enthält.

Dijkstra's Algorithmus funktioniert bei negativen Kantenlängen nicht:



Bei diesem Beispielgraphen (der nicht einmal einen negativen Kreis enthält) berechnet der Dijkstra-Algorithmus die minimale Entfernung von s nach w fälschlicherweise als 3 (statt 2).

8.4.2 Modifikation des Bellman-Ford-Algorithmus

$B_k[i]$ gibt die Länge eines kürzesten gerichteten s-i-Pfades an, der aus höchstens k Kanten besteht. Jeder Pfad, der keinen Kreis enthält, besteht aus maximal $n-1$ Kanten. In einem Graphen ohne negative Kreise gilt daher:

$$\forall i \in V : B_n[i] = B_{n-1}[i]$$

Gibt es hingegen einen (von s aus erreichbaren) Kreis negativer Länge, so gibt es einen Knoten $i \in V$, bei dem ein Pfad aus n Kanten mit der Länge $B_n[i]$ diesen Kreis häufiger durchläuft als jeder Pfad aus maximal $n-1$ Kanten der Länge $B_{n-1}[i]$. Demnach gilt in diesem Fall:

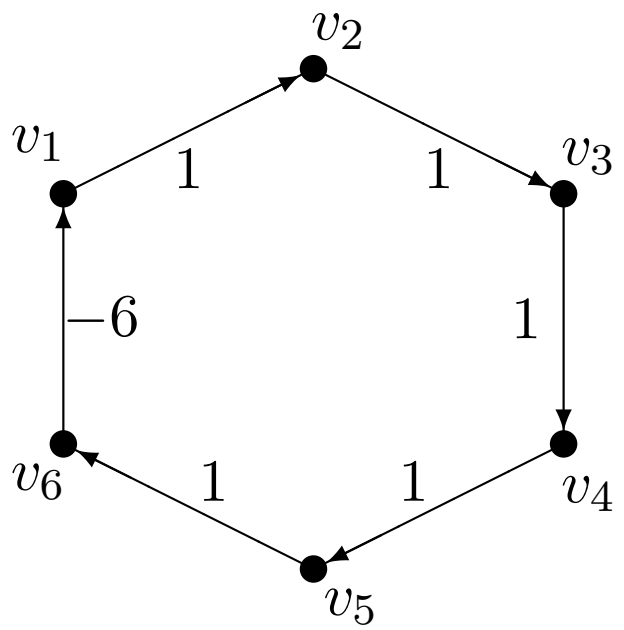
$$B_n[i] < B_{n-1}[i]$$

Man kann also in den Algorithmus von Bellman-Ford einen Test auf negative Kreise einbauen, indem man auch für alle $i \in V$ $B_n[i]$ berechnet und am Ende den folgenden Befehl einfügt:

for $i := 1$ **to** n **do**
  if $B_n[i] < B_{n-1}[i]$ **then** stop „Negativer Kreis“ **fi**

8.4.3 Modifikation des Floyd-Algorithmus

Falls kein negativer Kreis existiert, funktioniert der Algorithmus weiterhin korrekt.



$$c_{16}^6 = 5 = c_{16}^5$$

$$c_{61}^6 = -6 = c_{61}^5$$

$$c_{11}^6 = \min\{c_{11}^5, c_{16}^5 + c_{61}^5\} = -1$$

$\Rightarrow$ der Graph enthält einen negativen Kreis, gdw ein $c_{ii}^n < 0$ existiert.

Man kann also in den Algorithmus von Floyd einen Test auf negative Kreise einbauen, indem man am Ende den folgenden Befehl einfügt:

for $i := 1$ **to** n **do**
    if $c_{ii}^n < 0$ **then** stop „Negativer Kreis“ **fi**

8.4.4 Der Algorithmus von Johnson

Definition 197
Sei $\mathrm{d} : A \to \mathbb{R} \cup \{+\infty\}$ eine Distanzfunktion. Eine Abbildung

$$r : V \to \mathbb{R}$$

heißt Rekalibrierung, falls gilt:

$$(\forall (u,v) \in A)[r(u) + \mathrm{d}(u,v) \geq r(v)]$$

Beobachtung: Sei r eine Rekalibrierung (für d). Setze $\mathrm{d}'(u,v) := \mathrm{d}(u,v) + r(u) - r(v)$. Dann gilt:

$$\mathrm{d}'(u,v) \geq 0$$

Sei $u = v_0 \to \cdots \to v_k = v$ ein Pfad. Dann ist:

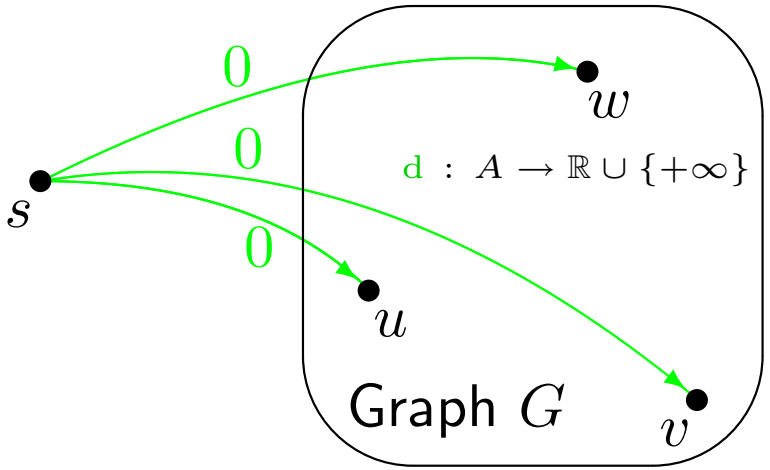
$$\text{d-Länge} := \sum_{i=0}^{k-1} \mathrm{d}(v_i, v_{i+1})$$

Demnach ist:

$$\begin{aligned}
\text{d}'\text{-Länge} &= \sum_{i=0}^{k-1} \mathrm{d}'(v_i, v_{i+1}) \\
&= \sum_{i=0}^{k-1} (\mathrm{d}(v_i, v_{i+1}) + r(v_i) - r(v_{i+1})) \\
&= \sum_{i=0}^{k-1} \mathrm{d}(v_i, v_{i+1}) + r(v_0) - r(v_k)
\end{aligned}$$

Also ist ein d-kürzester Pfad von u $(= v_0)$ nach v $(= v_k)$ auch ein d'-kürzester Pfad und umgekehrt. Nach einer Rekalibrierung kann man also auch die Algorithmen anwenden, die eine nichtnegative Distanzfunktion d voraussetzen (z.B. Dijkstra).

Berechnung einer Rekalibrierung:



Füge einen neuen Knoten s hinzu und verbinde s mit jedem anderen Knoten $v \in V$ durch eine Kante der Länge 0.

Berechne sssp von s nach allen anderen Knoten $v \in V$ (z.B. mit Bellman-Ford). Sei $r(v)$ die dadurch berechnete Entfernung von s zu $v \in V$. Dann ist r eine Rekalibrierung, denn es gilt:

$$r(u) + \mathrm{d}(u, v) \geq r(v)\,.$$

8.5 Zusammenfassung

	$d \geq 0$	d allgemein
sssp	D (Fibonacci): $\mathcal{O}(m + n \cdot \log n)$	B-F: $\mathcal{O}(n \cdot m)$
apsp	D: $\mathcal{O}(nm + n^2 \log n)$ F: $\mathcal{O}(n^3)^{(*)}$	J: $\mathcal{O}(n \cdot m + n^2 \log n)$ F: $\mathcal{O}(n^3)$

Bemerkung$^{(*)}$: In der Praxis ist der Floyd-Algorithmus für kleine n besser als Dijkstra's Algorithmus.