

## Info II

Die Mitarbeiter von <http://mitschriebwiki.nomeata.de/>

10. Januar 2017



# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>2</b>
<b>I. Über dieses Skriptum</b>	<b>7</b>
I.1. Wer	7
I.2. Wo	7
<b>II. Prädikatenlogik</b>	<b>9</b>
II.1. Syntax prädikatenlogischer Formeln	9
II.1.1. Grundsymbole	9
II.1.2. Terme	9
II.1.3. Atome	10
II.1.4. Formeln	10
II.2. Semantik prädikatenlogischer Formeln	11
II.2.1. Definition	11
II.2.2. Interpretation	12
II.2.3. Definition (Fortsetzung)	13
II.3. Normalformen	15
II.3.1. Definition (Äquivalenzbegriff)	15
II.3.2. Satz	15
II.3.3. Definition (NNF, KNF, DNF)	16
II.3.4. Definition (Substitution in Formeln)	16
II.3.5. Lemma (gebundene Umbenennung)	16
II.3.6. Lemma	16
II.3.7. Definition (Pränexform)	17
II.3.8. Satz	17
II.3.9. Definition (Skolemform)	17
II.3.10. Satz	18
II.3.11. Definition (Herbrand-Universum)	18
II.3.12. Definition (Herbrand-Expansion)	18
II.3.13. Satz (Gödel-Herbrand-Skolem)	18
II.4. Prolog und Prädikatenlogik	19
II.4.1. Logikprogrammierung	19
II.4.2. Pure Prolog	19
II.4.3. Prolog und Logik	19
II.4.4. Prädikatenlogik 1. Stufe	19
II.4.5. Formeln der Prädikatenlogik	19
II.4.6. Klauseln	20
II.4.7. Hornklauseln	20
II.4.8. Konjunktive Normalform	20
II.4.9. Logische Äquivalenzregeln	20
II.4.10. Quantorengesetze	21
II.4.11. Pränex-Normalform	22

II.4.12. Skolemisierung . . . . .	22
II.5. Aussagenlogische Resolution . . . . .	22
II.5.1. Definition (Resolvent) . . . . .	23
II.5.2. Definition ( $\text{Res}(F)$ ) . . . . .	23
II.5.3. Resolution in der Prädikatenlogik . . . . .	23
II.5.4. Allgemeinsten Unifikator . . . . .	24
II.6. Prädikatenlogische Resolution . . . . .	25
<b>III. Algorithmen</b> . . . . .	<b>27</b>
III.1. Definition . . . . .	27
III.2. Analyse von Algorithmen . . . . .	28
III.2.1. Das Sortierproblem . . . . .	28
III.2.2. Implementierung: Insertion-Sort . . . . .	28
III.2.3. Aufwandsklassen . . . . .	28
III.2.4. Analyse von Insertion Sort . . . . .	29
III.3. Aufwandsanalyse . . . . .	29
III.3.1. Methode: Teile und Beherrsche . . . . .	29
III.3.2. Laufzeiten . . . . .	30
III.3.3. Implementierung: MERGE-SORT . . . . .	30
III.3.4. Laufzeitanalyse . . . . .	31
III.4. Wachstum von Funktionen . . . . .	31
III.4.1. Asymptotische Notation - $\Theta$ -Notation . . . . .	31
III.4.2. Obere Asymptotische Schranke - O-Notation . . . . .	32
III.4.3. Untere Asymptotische Schranke: $\Omega$ -Notation . . . . .	32
III.4.4. Verhältnis der Mengen . . . . .	32
III.5. Rekurrenzen - Rekursionsgleichungen . . . . .	32
III.5.1. 1. Methode: „Raten und Induktion“ . . . . .	32
III.5.2. Rekursionsbaummethode . . . . .	33
III.5.3. Weitere Methoden . . . . .	33
III.6. Die $o$ -Notation . . . . .	33
III.7. Die $\omega$ -Notation . . . . .	33
III.8. Lösen von Rekurrenzen mit der Generierenden-Funktion . . . . .	33
III.9. Notationen . . . . .	35
III.10. Die Mastermethode . . . . .	35
III.11. Mastertheorem . . . . .	35
III.12. Probabilistische Algorithmen (zufallsgesteuerte Algorithmen) . . . . .	36
III.12.1. Einführung . . . . .	36
III.12.2. Macao-Algorithmen („Nächstes-Paar“-Algorithmus) . . . . .	37
III.12.3. Brute-Force-Methode („Brutaler Zwang“-Methode) . . . . .	37
III.12.4. Deterministische Algorithmen (Yuval) . . . . .	37
III.12.5. Monte-Carlo-Algorithmus . . . . .	38
III.12.6. Las-Vegas-Algorithmen . . . . .	39
III.13. Gierige Algorithmen . . . . .	40
III.13.1. Beispiel: . . . . .	41
III.13.2. Gierige Algorithmen abstrakt: . . . . .	41
III.13.3. Beispiel . . . . .	41
III.13.4. Kruskalscher Algorithmus . . . . .	42
III.13.5. Primscher Algorithmus . . . . .	44
III.13.6. Zeitplanerstellung (Scheduling) . . . . .	44
III.13.7. Greedy-Algorithmus . . . . .	45

III.13.8Zeitplanerstellung mit Schlußterminen (deadline)	45
III.14Teile und Herrsche	46
III.14.1Quicksort (C.A.R. Hoare, 1960)	47
III.14.2Selektion und Median	47
III.14.3Langzahlarithmetik	48
III.14.4Matrixmultiplikation	49
III.15Abstrakte Datentypen (ADT)	50
III.15.1Bool	50
III.15.2Schlange (queue, fifo)	50
III.15.3First In Last Out – Keller, Stack	51
III.15.4Liste	52
III.15.5Konkrete Implementierung	52
III.16Hash-Funktionen	53
III.17Graphenalgorithmen und Datenstrukturen für Graphen	55
III.17.11. Möglichkeit: Adjazenzliste	55
III.17.22. Möglichkeit: Adjazenzmatrix	55
III.17.3Speicherbedarf:	55
III.17.4Zugriff auf eine Kante:	56
III.17.5Einfache Graphenalgorithmen:	56
III.17.6Einfache Strategien:	56
III.18Binäre Suchbäume	57
III.18.1Definition/Einführung	57
III.18.2Traversierung	57
III.18.3Suchen	58
III.18.4Minimum und Maximum	59
III.18.5Vorgänger und Nachfolger	60
III.18.6Theorem	60
III.18.7Einfügen und Löschen	60
III.18.8Theorem	61
III.19Rot.Schwarz-Bäume	61
III.19.1Eigenschaften von R.S.Bäumen	61
III.19.2Rotationen	62
III.19.3Einfuegen	62
III.20Dynamisches Programmieren	66
III.20.1Optimierungsproblem:	66
III.20.2Beispiel:	67
III.20.3Beispiel 2:	68
III.21Vorbestimmung und Vorberechnung	69
III.21.1Vorbestimmung	69
III.21.2Vorberechnung für Zeichenreihe-Suchprobleme	72
III.22Vorberechnung für Zeichenreihen-Suchprobleme	74
III.22.1Algorithmus von Knuth-Morris-Pratt	74
III.22.2Algorithmus von Boyer-Moore	75
<b>IV. Objektorientierte Programmierung</b>	<b>77</b>
IV.1. Klassen	77
IV.1.1. Definition	77
IV.1.2. Gleichheit und Identität von Objekten	77
IV.2. Objektdiagramm	78
IV.3. Verwendung von Klassen	78



# I. Über dieses Skriptum

Dies ist ein erweiterter Mitschrieb der Vorlesung „Info II“ von Herrn Prof. Calmet im Sommersemester 05 an der Universität Karlsruhe (TH). Die Mitschriften der Vorlesung werden mit ausdrücklicher Genehmigung von Herrn Calmet hier veröffentlicht, Herr Calmet ist für den Inhalt nicht verantwortlich.

## I.1. Wer

Gestartet wurde das Projekt von Joachim Breitner. Beteiligt am Mitschrieb sind außer Joachim noch Felix Brandt und andere.

## I.2. Wo

Alle Kapitel inklusive  $\text{\LaTeX}$ -Quellen können unter <http://mitschriebwiki.nomeata.de> abgerufen werden. Dort ist ein *Wiki* eingerichtet und von Joachim Breitner um die  $\text{\LaTeX}$ -Funktionen erweitert. Das heißt, jeder kann Fehler nachbessern und sich an der Entwicklung beteiligen. Auf Wunsch ist auch ein Zugang über *Subversion* möglich.





# II. Prädikatenlogik

## II.1. Syntax prädikatenlogischer Formeln

### II.1.1. Grundsymbole

11.04.2005

Die Syntax definiert eine Menge von Grundsymbolen. Diese lassen sich weiter aufteilen:

#### logische Symbole

$\vee$	Disjunktion	(„oder“)
$\wedge$	Konjunktion	(„und“)
$\neg$	Negation	(„nicht“)
$\exists$	Existenz	(„es existiert“)
$\forall$	Allquantor	(„für all“)
$\rightarrow$	Implikation	(„dann“)
$\leftarrow$	(umgekehrte) Implikation	(„wenn“)
$\leftrightarrow$	Äquivalenz	(„genau dann wenn“)
$()$	Hilfssymbol	

#### frei wählbare, nicht-logische Symbole

- Symbole, die für ein Individuum (Objekt) stehen
  - Konstanten, z. B.  $a, b, c$ , John, 0,1,2, ...
  - Variablen, z. B.  $x, y, z$ , Name, Organisation, ...
- Funktionssymbole, z. B.  $f, g, h$ , mother\_of, sin, +, -, ...
- Prädikatensymbole, z. B.  $p, q, r$ , father, =, <, ≤, ...

Die freie Wählbarkeit wird durch Konventionen eingeschränkt, zum Beispiel in **Prolog**: Variablen beginnen mit Großbuchstaben, Konstanten sowie Funktions- und Prädikatensymbole mit Kleinbuchstaben.

### II.1.2. Terme

- Konstanten
- Variablen
- strukturierte Terme der Form  $\varphi(\tau_1, \tau_2, \dots, \tau_n)$ , wobei  $\varphi$  ein  $n$ -stelliges Funktionssymbol und  $\tau_1, \tau_2, \dots, \tau_n$  Terme sind.

Konvention nach Schöning:  $f$  ist ein Funktionssymbol mit Stelligkeit  $k$  und  $\tau_i$  ( $i = 1 \dots k$ ) sind Terme, dann ist auch  $f^k(\tau_1, \tau_2, \dots, \tau_k)$  ein Term.

### II.1.3. Atome

Auch atomare Formeln oder Primformeln genannt. Atome stehen für die Beziehung zwischen Individuen. Sie haben die Form  $\pi(\tau_1, \tau_2, \dots, \tau_n)$  wobei  $\pi$  ein  $n$ -stelliges Prädikatensymbol ist und  $\tau_i$  Terme sind.

### II.1.4. Formeln

Formeln erlauben eine logische Verknüpfung von Atomen und Termen. Im folgenden sind  $\Phi$  und  $\Psi$  Formeln und  $x$  eine Variable.

- Atome
- $(\Phi \vee \Psi)$
- $(\Phi \wedge \Psi)$
- $(\neg \Phi)$
- $(\exists x \Phi)$  („Es gibt ein  $x$ , so dass  $\Phi$  gilt“)
- $(\forall x \Phi)$  („Für alle  $x$  gilt  $\Phi$ “)

Die folgenden Formeln sind nur Abkürzungen:

- $(\Phi \rightarrow \Psi)$  Abkürzung für  $(\neg \Phi \vee \Psi)$ . („ $\Phi$  impliziert  $\Psi$ “), („wenn  $\Phi$ , dann  $\Psi$ “).
- $(\Phi \leftarrow \Psi)$  Abkürzung für  $(\Phi \vee \neg \Psi)$ . („ $\Phi$  wenn  $\Psi$ “), („ $\Phi$  falls  $\Psi$ “)
- $(\Phi \leftrightarrow \Psi)$  Abkürzung für  $((\Phi \leftarrow \Psi) \wedge (\Phi \rightarrow \Psi))$ . („ $\Phi$  genau, dann wenn  $\Psi$ “)

Ferner gelten folgende Konventionen:

- $\exists$  und  $\forall$  binden stärker als  $\neg$
- $\neg$  bindet stärker als  $\wedge$
- $\wedge$  bindet stärker als  $\vee$
- $\vee$  bindet stärker als  $\leftarrow, \rightarrow, \leftrightarrow$
- „unnötige“ Klammern werden weggelassen

Dies ist die Syntax der Prädikatenlogik *erster Stufe* (First Level Order). Erste Stufe bedeutet, dass Variablen nur für Individuen stehen können, nicht aber für Funktionen oder Prädikate.

**Beispiele** für Formeln

1. loves(romeo,juliet)
2.  $\forall x$  ancestor(adam, $x$ )
3.  $\forall x$  (person( $x$ )  $\rightarrow$  ancestor(adam, $x$ ))
4.  $\forall x$  (is\_mother( $x$ )  $\rightarrow \exists y$  (is\_child( $y$ )  $\wedge$  mother( $x,y$ )))
5.  $\forall$  sub  $\forall$  super (subset(sub,super)  $\leftarrow \forall x$  (member( $x$ ,sub)  $\rightarrow$  member( $x$ ,super)))

**Definition** einige weitere Begriffe.

- Die Menge der Variablen eines Terms  $\tau$ ,  $\text{vars}(\tau)$  kann induktiv wie folgt definiert werden:
  - Für eine Konstante  $c$  gilt:  $\text{vars}(c) := \{\}$
  - Für eine Variable  $x$  gilt:  $\text{vars}(x) := \{x\}$
  - Für einen Term  $\varphi(\tau_1, \tau_2, \dots, \tau_n)$  gilt:  
 $\text{vars}(\varphi(\tau_1, \tau_2, \dots, \tau_n)) := \text{vars}(\tau_1) \cup \text{vars}(\tau_2) \cup \dots \cup \text{vars}(\tau_n)$
- Eine Variable, die nicht durch einen Quantor gebunden ist, heißt *frei* (free). Die Menge der freien Variablen einer Formel  $\Phi$ ,  $\text{free}(\Phi)$ , ist induktiv wie folgt definiert:
  - Für ein Atom  $\pi(\tau_1, \tau_2, \dots, \tau_n)$  gilt:  
 $\text{free}(\pi(\tau_1, \tau_2, \dots, \tau_n)) := \text{vars}(\tau_1) \cup \text{vars}(\tau_2) \cup \dots \cup \text{vars}(\tau_n)$
  - $\text{free}(\neg\Phi) := \text{free}(\Phi)$
  - $\text{free}(\Phi \vee \Psi) := \text{free}(\Phi \wedge \Psi) := \text{free}(\Phi) \cup \text{free}(\Psi)$
  - $\text{free}(\exists x\Phi) := \text{free}(\forall x\Phi) := \text{free}(\Phi) \setminus \{x\}$
- In  $\forall x\Phi$  bzw.  $\exists x\Phi$  heißt die Variable  $x$  durch den jeweiligen Quantor gebunden (bound).
- Eine Formel mit freien Variablen heißt offen (open).
- Eine Formel ohne freie Variablen heißt geschlossen (closed).

## II.2. Semantik prädikatenlogischer Formeln

### II.2.1. Definition

(exzerpt aus “Logik für Informatiker”, Seite 52 ff) Eine Struktur ist ein Paar  $A(U_A, I_A)$ , wobei  $U_A$  eine beliebige nicht-leere Menge ist, das *Universon* (auch Grundmenge oder Grundbereich) genannt wird. Ferner ist  $I_A$  eine Abbildung, die

- jedem  $k$ -stelligen Prädikatsymbol  $P$  ein  $k$ -stelliges Prädikat über  $U_A$  zuordnet,
- jedem  $k$ -stelligen Funktionssymbol  $F$  eine  $k$ -stellige Funktion über  $U_A$  zuordnet sowie
- jeder Variablen  $x$  ein Element der Grundmenge  $U_A$  zuordnet.

Wir schreiben abkürzend  $P^A$  statt  $I_A(P)$ ,  $F_A$  statt  $I_A(F)$  und  $x^A$  statt  $I_A(x)$ .

**Das Herbrand-Universum** ist die Menge aller aus den Objekt- und Funktionssymbolen einer Formelmengen  $\Delta$  konstruierbaren Objekte.

Sei  $F$  eine Formel und  $A = (U_A, I_A)$  eine Struktur.  $A$  heißt zu  $F$  passend, falls  $I_A$  für alle in  $F$  vorkommenden Symbole und freien Variablen definiert ist.

**Beispiel:**

$$F = \forall x P(x, f(x)) \wedge Q(g(a, z))$$

ist eine Formel. Eine zu  $F$  passende Struktur ist dann Beispielsweise:

$$\begin{aligned}
 U_A &= \{1, 2, 3, \dots\} \\
 I_A(P) &= \{(m, n) \mid m, n \in U_A \text{ und } m < n\} \\
 I_A(Q) &= \{n \in U_A \mid n \text{ ist Primzahl}\} \\
 I_A(f) &= f^A(n) = n + 1 \\
 I_A(g) &= g^A(m, n) = m + n \\
 I_A(a) &= 2 \\
 I_A(z) &= 3
 \end{aligned}$$

Beachte:  $x$  kommt gebunden vor.

### II.2.2. Interpretation

Die Interpretation einer prädikatenlogischen Formel ergibt sich, wenn man jeder verwendeten Konstanten ein bekanntes Objekt eines Diskursbereichs  $D$  zuordnet und jedes verwendete Prädikatsymbol als Beziehungstyp zwischen zwei oder mehreren Objekten ansieht. Die Abbildung der Formel auf  $D$  wird als beabsichtigte Interpretation (*intended interpretation*) bezeichnet.

**Beispiel:** Einfacher Diskursbereich

$$D := \{\mathbb{J}, \mathbb{R}\}$$

Eine Interpretation  $I$  einer prädikatenlogischen Sprache ist ein Paar  $\langle D, I \rangle$  wobei  $I$  eine Abbildung der nicht-logischen Symbole auf (nicht-leeres)  $D$  ist, wobei  $I$

- jeder Konstanten der Sprache ein Element von  $D$  zuordnet
- jedem  $n$ -stelligen Funktionssymbol eine Abbildung  $D^n \mapsto D$  zuordnet
- jedem  $n$ -stelligen Prädikatsymbol eine Abbildung  $D^n \mapsto \{F, T\}$  zuordnet

Für Grundterme der Form  $\varphi(\tau_1, \dots, \tau_n)$  wird die Interpretation wie folgt berechnet:

$$I(\varphi(\tau_1, \dots, \tau_n)) := I(\varphi)(I(\tau_1) \dots I(\tau_n))$$

Jeder Term  $\tau_i$  wird demzufolge durch Anwendung der Abbildung  $I$  auf ein Element  $I(\tau_i) \in D$  abgebildet. Auch  $\varphi$  wird durch  $I$  abgebildet auf  $I(\varphi)$ .

Für Grundatome  $\pi(\tau_1, \dots, \tau_n)$  wird  $I$  wie folgt berechnet:

$$I(\pi(\tau_1, \dots, \tau_n)) := I(\pi)(I(\tau_1, \dots, \tau_n))$$

**Beispiel:** Interpretation von Elementen der Prädikatenlogik

Konstanten:  $I(\text{juliet}) = \mathbb{J}$

$I(\text{romeo}) = \mathbb{R}$

Funktionen:  $I(\text{girlfriend\_of}) = \{\mathbb{R} \mapsto \mathbb{J}\}$

Prädikate:  $I(\text{woman}) = \{\mathbb{J} \mapsto T, \mathbb{R} \mapsto F\}$

$I(\text{loves}) = \{\langle \mathbb{J}, \mathbb{J} \rangle \mapsto F, \langle \mathbb{J}, \mathbb{R} \rangle \mapsto T, \langle \mathbb{R}, \mathbb{J} \rangle \mapsto T, \langle \mathbb{R}, \mathbb{R} \rangle \mapsto F\}$

Grundterme:  $I(\text{girlfriend\_of}(\text{romeo})) = \{\mathbb{R} \mapsto \mathbb{J}\}(\mathbb{R}) = \mathbb{J}$

Grundatome:  $I(\text{woman}(\text{juliet})) = \{\mathbb{J} \mapsto T, \mathbb{R} \mapsto F\}(\mathbb{J}) = T$

$I(\text{woman}(\text{romeo})) = \{\mathbb{J} \mapsto T, \mathbb{R} \mapsto F\}(\mathbb{R}) = F$

**II.2.3. Definition (Fortsetzung)**

(Exzerpt aus „Logik für Informatiker“, Schönig , Seite 53 ff.)

Sei  $F$  eine Formel und  $A$  eine zu  $F$  passende Struktur. Für jeden Term  $t$ , den man aus den Bestandteilen von  $F$  bilden kann, definieren wir nun den Wert von  $t$  in der Struktur  $A$ , den wir mit  $A(t)$  bezeichnen.

1. Falls  $t$  eine Variable ist (also  $t = x$ ), so ist  $A(t) = x^A$ .
2. Falls  $t$  die Form  $t = f(t_1, \dots, t_k)$  hat, wobei  $t_1, \dots, t_k$  Terme sind und  $f$  ein  $k$ -stelliges Funktionssymbol ist, so ist  $A(t) = f^A(A(t_1), \dots, A(t_k))$

Auf analoge Weise definieren wir den (Wahrheits-) Wert der Formel  $F$  wobei wir ebenfalls die Bezeichnung  $A(F)$  verwenden.

1. Falls  $F$  die Form  $F = P(t_1, \dots, t_k)$  mit Termen  $t_1, \dots, t_k$ , so ist

$$A(F) = \begin{cases} 1, & \text{falls } ((A(t_1), \dots, A(t_k)) \in P^A) \\ 0, & \text{sonst} \end{cases}$$

2. Falls „ $F = \neg G$ “ hat, so ist

$$A(F) = \begin{cases} 1, & \text{falls } A(G) = 0 \\ 0, & \text{sonst} \end{cases}$$

3. Falls „ $F = (G \wedge H)$ “ so ist

$$A(F) = \begin{cases} 1, & \text{falls } A(G) = 1 \text{ und } A(H) = 1 \\ 0, & \text{sonst} \end{cases}$$

4. Falls „ $F = (G \vee H)$ “ so ist

$$A(F) = \begin{cases} 1, & \text{falls } A(G) = 1 \text{ oder } A(H) = 1 \\ 0, & \text{sonst} \end{cases}$$

5. Falls „ $F = \forall x G$ “ so ist

$$A(F) = \begin{cases} 1, & \text{falls für alle } d \in U_A \text{ gilt } A_{[x/d]}(G) = 1 \\ 0, & \text{sonst} \end{cases}$$

6. Falls „ $F = \exists x G$ “ so ist

$$A(F) = \begin{cases} 1, & \text{falls es ein } d \in U_A \text{ gibt, mit } A_{[x/d]}(G) = 1 \\ 0, & \text{sonst} \end{cases}$$

Hierbei bedeute  $A_{[x/d]}$  diejenige Struktur  $A'$ , die überall mit  $A$  identisch ist, bis auf die Definition von  $x^A$ . Hier ist  $x^{A'} = d$  wobei  $d \in U_A = U_{A'}$

## Bemerkungen

1.  $F$  ist gültig  $\Leftrightarrow \neg F$  unerfüllbar ist.

**Beweis:** es gilt  $F$  ist gültig

$\Leftrightarrow$  jede zu  $F$  passende Belegung ist ein Modell für  $F$

$\Leftrightarrow$  jede zu  $F$  und damit auch zu  $\neg F$  passende Belegung ist kein Modell für  $\neg F$

$\Leftrightarrow \neg F$  besitzt kein Modell

$\Leftrightarrow \neg F$  ist unerfüllbar.

2. Nicht jede mathematische Aussage kann im Rahmen der Prädikatenlogik formuliert werden. Erst wenn auch Quantoren über Prädikaten und Funktionssymbolen erlaubt werden ist dies möglich. Dies ist dann die Prädikatenlogik der zweiten Stufe. Die obige ist die Prädikatenlogik der ersten Stufe.

## Beispiel Interpretation eines Atoms mit einer Variablen

Die Notation für eine Variablenbelegung (*valuation*) ist  $[x \mapsto c]$  ( $c \in D$ ).

Atom mit Variable

Für die Variablenbelegung  $\mu = [x \mapsto \mathbb{J}]$  ist  $I_\mu(\text{woman}(x)) = T$

Für die Variablenbelegung  $\nu = [x \mapsto \mathbb{R}]$  ist  $I_\nu(\text{woman}(x)) = F$

Seien  $\Phi$  und  $\Psi$  Formeln, deren Interpretationen  $I_\mu(\Phi)$  und  $I_\mu(\Psi)$  unter der Variablenbelegung  $\mu$  bekannt sind, dann folgt daraus, dass eine Interpretation für „komplexe“ Formeln existiert und induktiv wie folgt bestimmt werden kann:

$$\begin{aligned}
 I_\mu(\Phi \vee \Psi) &:= T \Leftrightarrow I_\mu(\Phi) = T \vee I_\mu(\Psi) = T \\
 I_\mu(\Phi \wedge \Psi) &:= T \Leftrightarrow I_\mu(\Phi) = T \wedge I_\mu(\Psi) = T \\
 I_\mu(\neg \Phi) &:= T \Leftrightarrow I_\mu(\Phi) = F \\
 I_\mu(\Phi \rightarrow \Psi) &:= T \Leftrightarrow I_\mu(\Phi) = F \vee I_\mu(\Psi) = T \\
 I_\mu(\Phi \leftarrow \Psi) &:= T \Leftrightarrow I_\mu(\Phi) = T \vee I_\mu(\Psi) = F \\
 I_\mu(\Phi \leftrightarrow \Psi) &:= T \Leftrightarrow (I_\mu(\Phi) = T \vee I_\mu(\Psi) = T) \wedge (I_\mu(\Phi) = F \vee I_\mu(\Psi) = F) \\
 I_\mu(\exists x \Phi) &:= T \Leftrightarrow \text{ein } c \in D \text{ existiert, so dass } I_{\mu[x \mapsto c]}(\Phi) = T \\
 I_\mu(\forall x \Phi) &:= T \Leftrightarrow \text{für alle } c \in D \text{ gilt } I_{\mu[x \mapsto c]}(\Phi) = T
 \end{aligned}$$

**Bemerkung:** Eine Formel kann auf verschiedene Arten interpretiert werden.

## Beispiel: verschiedene Interpretationen einer Formel

Sei  $\Phi = \forall x(g(x, f, t) \rightarrow l(t, x))$

Die Interpretation  $I$  sei wie folgt definiert:

- $I(f) := \text{„(Katzen-)Futter“}$
- $I(t) := \text{„Katze namens Tom“}$
- $I(g) := \text{„Die Abbildung, die einem Tripel } \langle X, Y, Z \rangle \text{ } T \text{ zuordnet, falls Objekt } X \text{ dem Objekt } Z \text{ das Objekt } Y \text{ gibt und } F \text{ sonst“}$
- $I(l) := \text{„Die Abbildung, die einem Paar } \langle X, Y \rangle \text{ } T \text{ zuordnet, falls das Objekt } X \text{ das Objekt } Y \text{ liebt und } F \text{ sonst“}$

Die Aussage „Falls  $X$  der Katze namens Tom Katzenfutter gibt, dann liebt Tom dieses  $X$ “ kann sehr wohl für jede Variablenbelegung von  $X$  aus einem bestimmten Diskursbereich als wahr betrachtet werden. Damit wäre die Formel  $\Phi$  bezüglich der Interpretation  $I$  erfüllt, das heißt  $I(\Phi) = T$ .

$A$	$B$	$A \vee B$	$A \wedge B$	$A \rightarrow B \ (\equiv \neg A \vee B)$	$A \leftrightarrow B \ (\equiv (A \wedge B) \vee (\neg A \wedge \neg B))$
W	W	W	W	W	W
W	F	W	F	F	F
F	W	W	F	W	F
F	F	F	F	W	W

Abbildung II.1.: Wahrheitswerte der Aussagen

## II.3. Normalformen

18.04.2005

### II.3.1. Definition (Äquivalenzbegriff)

Zwei prädikatenlogische Formeln  $F$  und  $G$  sind äquivalent, falls für alle sowohl zu  $F$  als auch zu  $G$  passenden Strukturen  $A$  gilt:

$$A(F) = A(G)$$

#### Beispiel (de Morgansches Gesetz)

$$\neg(F \wedge G) \equiv \neg F \vee \neg G$$

### II.3.2. Satz

Seien  $F$  und  $G$  beliebige Formeln. Dann gilt:

1.

$$\begin{aligned}\neg \forall x F &\equiv \exists x \neg F \\ \neg \exists x F &\equiv \forall x \neg F\end{aligned}$$

2. Falls  $x$  in  $G$  nicht frei vorkommt, gilt:

$$\begin{aligned}(\forall x F \wedge G) &\equiv \forall x (F \wedge G) \quad (*) \\ (\forall x F \vee G) &\equiv \forall x (F \vee G) \\ (\exists x F \wedge G) &\equiv \exists x (F \wedge G) \\ (\exists x F \vee G) &\equiv \exists x (F \vee G)\end{aligned}$$

3.

$$\begin{aligned}(\forall x F \wedge \forall x G) &\equiv \forall x (F \wedge G) \\ (\exists x F \vee \exists x G) &\equiv \exists x (F \vee G)\end{aligned}$$

4.

$$\begin{aligned}\forall x \forall y F &\equiv \forall y \forall x F \\ \exists x \exists y F &\equiv \exists y \exists x F\end{aligned}$$

**Beweis** (\*) (exemplarisch)

Sei  $A = (U_A, I_A)$  eine zu den beiden Seiten der zu beweisenden Äquivalenz passende Struktur. Es gilt:

$$\begin{aligned}
 & A(\forall x F \wedge G) = 1 \\
 \Leftrightarrow & A(\forall x F) = 1 \text{ und } A(G) = 1 \\
 \Leftrightarrow & \text{für alle } d \in U_A \text{ gilt:} \\
 & A_{[x/d]}(F) = 1 \text{ und } A(G) = 1 \\
 \Leftrightarrow & \text{für alle } d \in U_A \text{ gilt:} \\
 & A_{[x/d]}(F) = 1 \text{ und } A_{[x/d]}(G) = 1 \\
 & \text{(\textbf{Bem.:} da } x \text{ in } G \text{ nicht frei vorkommt, ist nämlich } A(G) = A_{[x/d]}(G)) \\
 \Leftrightarrow & \text{für alle } d \in U_A \text{ gilt:} \\
 & A_{[x/d]}(F \wedge G) = 1 \\
 \Leftrightarrow & A(\forall x (F \wedge G)) = 1
 \end{aligned}$$

### II.3.3. Definition (NNF, KNF, DNF)

- Eine Formel ist in *Negationsnormalform*, wenn sie außer  $\neg, \wedge, \vee$  keine Junktoren enthält und wenn in ihr  $\neg$  nur vor atomaren Teilformen auftritt (NNF).
- Eine Formel ist in *konjunktiver* (bzw. *disjunktiver*) *Normalform*, wenn sie eine Konjunktion (bzw. Disjunktion) von Disjunktionen (bzw. von Konjunktionen) von Literalen ist.

### II.3.4. Definition (Substitution in Formeln)

Sei  $F$  eine Formel,  $x$  eine Variable und  $t$  ein Term. Dann bezeichnet  $F_{[x/t]}$  diejenige Formel, die man aus  $F$  erhält, indem man jedes freie Vorkommen der Variablen  $x$  in  $F$  durch den Term  $t$  ersetzt.  $[x/t]$  beschreibt eine *Substitution*. Substitutionen (oder Folgen von Substitutionen) behandeln wir auch als selbstständige Objekte, z. B.  $sub = [x/t_1][y/t_2]$  (wobei  $t_1$  auch  $y$  enthalten darf).

### II.3.5. Lemma (gebundene Umbenennung)

Sei  $F = QxG$  eine Formel mit  $Q \in \{\exists, \forall\}$ . Es sei  $y$  eine Variable, die in  $G$  nicht vorkommt. Dann gilt:

$$F \equiv QyG_{[x/y]}$$

### II.3.6. Lemma

Zu jeder Formel  $F$  gibt es eine äquivalente Formel  $G$  in *bereinigter* Form.

Hierbei heißt die Formel bereinigt, sofern es keine Variable gibt, die in der Formel sowohl gebunden als auch frei vorkommt, und sofern hinter allen vorkommenden Quantoren verschiedene Variablen stehen.

**Beispiel**

$$\begin{aligned}
 F &:= \forall x \exists y (P(x, y) \wedge Q(x, a)) \\
 G &:= \exists u (\forall v P(u, v) \vee Q(v, v))
 \end{aligned}$$

$F$ :  $x, y$  sind gebunden,  $a$  ist frei  $\rightarrow F$  ist bereinigt.

$G$ : alle Vorkommen der Variablen  $u$  sind gebunden, allerdings tritt  $v$  sowohl gebunden als auch frei auf  $\rightarrow G$  ist nicht in bereinigter Form.



### II.3.7. Definition (Pränexform)

Eine Formel heißt pränex oder in Pränexform, falls sie die Form

$$Q_1 y_1 Q_2 y_2 \dots Q_n y_n F$$

hat, wobei  $Q_i \in \{\exists, \forall\}$ ,  $n \geq 0$ ,  $y_i$  Variablen sind. Es kommt ferner kein Quantor in  $F$  vor.

### II.3.8. Satz

Für jede Formel  $F$  gibt es eine äquivalente (und bereinigte) Formel  $G$  in Pränexform.

**Beweis** (Induktion über den Formelaufbau)

**Induktionsanfang:**  $F$  ist atomare Formel. Dann liegt  $F$  bereits in der gewünschten Form vor.  
Wähle also  $G = F$ .

**Induktionsschritt:** Wir betrachten wieder die verschiedenen Fälle.

1.  $F$  hat die Form  $\neg F_1$  und  $G_1 = Q_1 y_1 Q_2 y_2 \dots Q_n y_n G'$  sei die nach Induktionsvoraussetzung existierende zu  $F_1$  äquivalente Formel. Dann gilt:

$$F \equiv \overline{Q}_1 y_1 \overline{Q}_2 y_2 \dots \overline{Q}_n y_n \neg G'$$

wobei  $\overline{Q}_i = \exists$ , falls  $Q_i = \forall$  und  $\overline{Q}_i = \forall$  falls  $Q_i = \exists$ . Diese Formel hat die gewünschte Form.

2.  $F$  hat die Form  $(F_1 \circ F_2)$  mit  $\circ \in \{\wedge, \vee\}$ . Dann gibt es zu  $F_1$  und  $F_2$  äquivalente Formeln  $G_1$  und  $G_2$  in bereinigter Pränexform. Durch gebundenes Umbenennen können wir die gebundenen Variablen von  $G_1$  und  $G_2$  disjunkt machen. Dann hat:

- $G_1$  die Form  $Q_1 y_1 Q_2 y_2 \dots Q_k y_k G'_1$
- $G_2$  die Form  $Q'_1 z_1 Q'_2 z_2 \dots Q'_l z_l G'_2$

mit  $Q_i, Q'_j \in \{\exists, \forall\}$ . Daraus folgt, dass  $F$  zu

$$Q_1 y_1 Q_2 y_2 \dots Q_k y_k Q'_1 z_1 \dots Q'_l z_l (G'_1 \circ G'_2)$$

äquivalent ist.

3.  $F$  hat die Form  $Qx F_1$ ,  $Q \in \{\exists, \forall\}$ . Die nach Induktionsvoraussetzung existierende bereinigte Pränexform habe die Bauart  $Q_1 y_1 Q_2 y_2 \dots Q_k y_k F'_1$ . Durch gebundenes Umbenennen kann die Variable  $x$  verschieden gemacht werden von  $y_1, \dots, y_k$ . Dann ist  $F$  zu  $Qx Q_1 y_1 Q_2 y_2 \dots Q_k y_k F'_1$  äquivalent.

### II.3.9. Definition (Skolemform)

Für jede Formel  $F$  in BPF (bereinigte Pränexform) definieren wir ihre *Skolemform(-el)* als das Resultat der Anwendung folgenden Algorithmus' auf  $F$ :

SKOLEMFORM-ALGORITHMUS

- 1 **while**  $F$  enthält einen Existenzquantor
- 2     **do**  $F$  habe die Form  $F = \forall y_1 \forall y_2 \dots \forall y_n \exists z G$  für eine Formel  $G$  in BPF  
und  $n \geq 0$  (der Allquantorblock kann auch leer sein);
- 3     Sei  $f$  ein neues, bisher in  $F$  nicht vorkommendes  $n$ -stelliges  
Funktionssymbol;
- 4      $F := \forall y_1 \forall y_2 \dots \forall y_n G_{[z/f(y_1, y_2, \dots, y_n)]}$ ; (d.h. der Existenzquantor in  $F$  wird  
gestrichen und jedes Vorkommen der Variable  $z$  in  $G$  durch  
 $f(y_1, y_2, \dots, y_n)$  ersetzt.

**II.3.10. Satz**

Für jede Formel  $F$  in BPF gilt:  $F$  ist erfüllbar  $\Leftrightarrow$  die Skolemform ist erfüllbar.

**Beweis** siehe Schöning, U.: „Logik für Informatiker“

**II.3.11. Definition (Herbrand-Universum)**

Das Herbrand-Universum  $D(F)$  einer geschlossenen Formel  $F$  in Skolemform ist die Menge aller variablenfreien Terme, die aus den Bestandteilen von  $F$  gebildet werden können. Falls keine Konstante vorkommt, wählen wir eine beliebige Konstante, z. B.  $a$  und bilden dann die variablenfreien Terme.

$D(F)$  wird wie folgt induktiv definiert:

1. Alle in  $F$  vorkommenden Konstanten sind in  $D(F)$ .  
Falls  $F$  keine Konstanten enthält, so ist  $a$  in  $D(F)$ .
2. Für jedes in  $F$  vorkommene  $n$ -stellige Funktionssymbol  $f$  und Terme  $t_1, t_2, \dots, t_n$  in  $D(F)$  ist der Term  $f(t_1, t_2, \dots, t_n)$  in  $D(F)$ .

**Beispiel**

$$\begin{aligned} F &= \forall x \forall y \forall z R(x, f(y), g(z, x)) \\ G &= \forall x \forall y Q(c, f(x), h(y, b)) \end{aligned}$$

Für  $F$  liegt der Spezialfall aus 1. vor (kein Vorkommen einer Konstanten).

$$\begin{aligned} D(F) &= \{a, f(a), g(a, a), f(g(a, a)), f(f(a)), \\ &\quad g(a, f(a)), g(f(a), a), g(f(a)), \dots\} \\ D(G) &= \{c, b, f(c), f(b), h(c, c), h(c, b), h(b, c), \\ &\quad h(b, b), f(f(c)), f(f(b)), f(h(c, c)), f(h(c, b)), \dots\} \end{aligned}$$

**II.3.12. Definition (Herbrand-Expansion)**

Sei  $F = \forall y_1 \forall y_2 \dots \forall y_n F^*$  eine Aussage in Skolemform, dann ist  $E(F)$  die Herbrand-Expansion von  $F$  definiert als

$$E(F) = \{F^*_{[y_1/t_1][y_2/t_2]\dots[y_n/t_n]} | t_1, t_2, \dots, t_n \in D(F)\}$$

Anmerkung, nicht im Skript: Es werden also alle gebundenen Variablen von  $F$  in der Matrix  $F^*$  von  $F$  durch beliebige Terme aus dem Herbrand-Universum  $D(F)$  ersetzt.

**II.3.13. Satz (Gödel-Herbrand-Skolem)**

Für jede Aussage  $F$  in Skolemform gilt:  $F$  ist erfüllbar genau dann, wenn die Formelmengende  $E(F)$  (im aussagenlogischen Sinn) erfüllt ist.

**Beispiel**

$$F = \forall x \forall y \forall z P(x, f(y, y), g(z, x))$$

Die einfachsten Formeln in  $E(F)$  sind die folgenden:

$$\begin{aligned} P(a, f(a, a), g(a, a)) &\quad \text{mit} \quad [x/a], \dots, [z/a] \\ P(f(a), f(a, a), g(a, f(a))) &\quad \text{mit} \quad [x/f(a)], [y/a], [z/a] \end{aligned}$$

## II.4. Prolog und Prädikatenlogik

### II.4.1. Logikprogrammierung

Prolog wurde um 1970 von Alain Colmerauer und seinen Mitarbeitern mit dem Ziel entwickelt, die Programmierung von Computern mit den Mitteln „der Logik“ zu ermöglichen.

### II.4.2. Pure Prolog

Das sogenannte Pure Prolog oder Database-Prolog entspricht einer Teilmenge der Sprachdefinition eines praktischen Prolog-Entwicklungssystems und enthält keine extra- oder metalogischen Komponenten wie:

- Cut, Type-Checking
- Arithmetische Operationen
- Datenbasismanipulation zur Laufzeit

### II.4.3. Prolog und Logik

Pure Prolog-Programme entsprechen den Ausdrücken der Hornklausellogik, die eine Teilmenge der Prädikatenlogik 1. Stufe ist. Das Beweisverfahren Resolution ermöglicht Inferenzen aufgrund von Prologprogrammen oder Hornklauseln.

**Bemerkung** Zur Transformation von prädikatenlogischen Sachverhalten nach Prolog wird der Weg über die Hornklauseln genutzt.

$$\text{Prädikatenlogik 1. Stufe} \xrightarrow{\text{KNF}} \text{Hornklauseln} \implies \text{Prolog}$$

### II.4.4. Prädikatenlogik 1. Stufe

**Inventar der Syntax:**

- Individuenkonstanten  $a, b, c, \dots$
- Individuenvariablen  $x, y, z, \dots$
- Prädikate  $P(t_1, \dots, t_n)$ ,  $t_i \in \text{TERM}$
- Quantoren  $\forall, \exists$
- Junktoren  $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$

### II.4.5. Formeln der Prädikatenlogik

- Wenn  $P$  ein  $n$ -stelliges Prädikat ist und  $t_1, \dots, t_n$  Terme sind, dann ist  $P(t_1, \dots, t_n)$  ein Literal.
- Literale sind Formeln.
- Wenn  $\Phi$  und  $\Psi$  Formeln sind, dann sind auch  $\neg\Phi$ ,  $\Phi \vee \Psi$ ,  $\Phi \wedge \Psi$ ,  $\Phi \rightarrow \Psi$ ,  $\Phi \leftrightarrow \Psi$  Formeln.
- Wenn  $\Phi$  eine Formel ist und  $x$  eine Variable, dann sind auch  $\forall x\Phi$ ,  $\exists x\Phi$  Formeln.

### II.4.6. Klauseln

- Wenn  $P$  ein  $n$ -stelliges Prädikat ist und  $t_1, \dots, t_n$  Terme sind, dann ist  $P(t_1, \dots, t_n)$  ein Literal.
- Literale sind Klauseln.
- Wenn  $\Phi$  ein Literal ist, dann sind auch  $\neg\Phi$ , eine Klausel.
- Wenn  $\Phi$  und  $\Psi$  Klauseln sind, dann ist auch  $\Phi \vee \Psi$  eine Klausel.

### II.4.7. Hornklauseln

Hornklauseln sind Klauseln, die genau ein nicht-negiertes Literal und beliebig viele negierte Literale enthalten.

#### Beispiele

- $\text{vater}(x, y) \vee \neg\text{elternteil}(x, y) \vee \neg\text{männlich}(x)$
- $\text{sterblich}(\text{sokrates})$

Anmerkung, nicht in der Vorlesung: Hornklauseln lassen sich als Implikationen darstellen. Die beiden Beispielen sind zu den folgenden äquivalent:

- $(\text{elternteil}(x, y) \wedge \text{Männlich}(x)) \Rightarrow \text{vater}(x, y)$
- $\text{wahr} \Rightarrow \text{sterblich}(\text{sokrates})$

### II.4.8. Konjunktive Normalform

Eine Formel ist in konjunktiver Normalform, wenn sie eine Konjunktion von Klauseln repräsentiert.

$$K_1 \wedge \dots \wedge K_n, K_i \in \text{KLAUSEL}$$

Formeln der Prädikatenlogik können durch Anwendung logischer Äquivalenzregeln in die konjunktive Normalform gebracht werden.

### II.4.9. Logische Äquivalenzregeln

#### Kommutativität

$$\begin{aligned} P \wedge Q &\Leftrightarrow Q \wedge P \\ P \vee Q &\Leftrightarrow Q \vee P \end{aligned}$$

#### Assoziativität

$$\begin{aligned} (P \wedge Q) \wedge R &\Leftrightarrow P \wedge (Q \wedge R) \\ (P \vee Q) \vee R &\Leftrightarrow P \vee (Q \vee R) \end{aligned}$$

#### De Morgan

$$\begin{aligned} \neg(P \vee Q) &\Leftrightarrow \neg P \wedge \neg Q \\ \neg(P \wedge Q) &\Leftrightarrow \neg P \vee \neg Q \end{aligned}$$

**Konditional- & Bikonditionalgesetz**

$$P \rightarrow Q \Leftrightarrow \neg P \vee Q$$

$$P \leftrightarrow Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$$

**Idempotenz**

$$P \vee P \Leftrightarrow P$$

$$P \wedge P \Leftrightarrow P$$

**Identität**

$$P \vee 0 \Leftrightarrow P$$

$$P \vee 1 \Leftrightarrow 1$$

$$P \wedge 0 \Leftrightarrow 0$$

$$P \wedge 1 \Leftrightarrow P$$

**Komplementarität**

$$P \vee \neg P \Leftrightarrow 1 \text{ (Tautologie, allgemeingültig)}$$

$$P \wedge \neg P \Leftrightarrow 0 \text{ (Kontradiktion, Inkonsistenz)}$$

$$\neg \neg P \Leftrightarrow P \text{ (Doppelte Negation)}$$

**II.4.10. Quantorengesetze****Quantoren-Negation**

$$\neg \forall x \Phi \Leftrightarrow \exists x \neg \Phi$$

$$\forall x \Phi \Leftrightarrow \neg \exists x \neg \Phi$$

$$\neg \forall x \neg \Phi \Leftrightarrow \exists x \Phi$$

$$\forall x \neg \Phi \Leftrightarrow \neg \exists x \Phi$$

**Quantoren-Distribution**

$$\forall x (\Phi \wedge \Psi) \Leftrightarrow \forall x \Phi \wedge \forall x \Psi$$

$$\exists x (\Phi \vee \Psi) \Leftrightarrow \exists x \Phi \vee \exists x \Psi$$

$$\forall x \Phi \vee \forall x \Psi \Rightarrow \forall x (\Phi \vee \Psi)$$

$$\exists x (\Phi \wedge \Psi) \Rightarrow \exists x \Phi \wedge \exists x \Psi$$

**Quantoren-Dependenz**

$$\forall x \forall y \Phi \Leftrightarrow \forall y \forall x \Phi$$

$$\exists x \exists y \Phi \Leftrightarrow \exists y \exists x \Phi$$

$$\exists x \forall y \Phi \Rightarrow \forall y \exists x \Phi$$

## Quantoren-Bewegung

$$\begin{aligned}\Phi \rightarrow \forall x \Psi &\Leftrightarrow \forall x (\Phi \rightarrow \Psi) \\ \Phi \rightarrow \exists x \Psi &\Leftrightarrow \exists x (\Phi \rightarrow \Psi) \\ (\forall x \Phi) \rightarrow \Psi &\Leftrightarrow \exists x (\Phi \rightarrow \Psi) \\ (\exists x \Phi) \rightarrow \Psi &\Leftrightarrow \forall x (\Phi \rightarrow \Psi)\end{aligned}$$

### II.4.11. Pränex-Normalform

Eine prädikatenlogische Formel befindet sich in PNF, wenn alle Quantoren am Anfang der Formel stehen.

$$\begin{aligned}(\exists x F(x)) \rightarrow (\forall y G(y)) \\ \Downarrow \\ \forall y \forall x (F(x) \rightarrow G(y))\end{aligned}$$

### II.4.12. Skolemisierung

Existenzquantoren können eliminiert werden, indem existenzquantifizierte Variablen durch Skolemkonstanten substituiert werden. Liegt ein Existenzquantor im Skopus von Allquantoren, werden die Skolemkonstanten mit den jeweiligen allquantifizierten Variablen durch Parametrisierung in Abhängigkeit gebracht.

#### Beispiele

- $\exists y \forall x ((\text{man}(x) \wedge (\text{woman}(y)) \rightarrow \text{loves}(x, y))$   
 $\implies \forall x ((\text{man}(x) \wedge \text{woman}(G)) \rightarrow \text{loves}(x, G))$
- $\forall x (\text{man}(x) \rightarrow \exists y (\text{woman}(y) \wedge \text{loves}(x, y)))$   
 $\implies \forall x (\text{man}(x) \rightarrow (\text{woman}(G(x)) \wedge \text{loves}(x, G(x))))$

## II.5. Aussagenlogische Resolution

Sie dient zum Nachweis der Unerfüllbarkeit einer Formel. Viele Aufgaben können auf einen Test auf Unerfüllbarkeit reduziert werden:

- $F$  ist Tautologie gdw.  $\neg F$  ist unerfüllbar
- Folgt  $G$  aus  $F_1, F_2, \dots, F_n$  ?  
d.h.: Ist  $F_1 \wedge F_2 \wedge \dots \wedge F_n \rightarrow G$  eine Tautologie?  
d.h.: Ist  $F_1 \wedge F_2 \wedge \dots \wedge F_n \wedge \neg G$  unerfüllbar?

Die aussagenlogische Resolution ist eine Umformungsregel und geht von der *Klauselschreibweise* der KNF Formel aus. Die Formel  $F$  liegt in KNF vor, wenn gilt:

$$F = (L_{1,1} \vee \dots \vee L_{1,n}) \wedge \dots \wedge (L_{m,1} \vee \dots \vee L_{m,n})$$

wobei  $L_{i,j}$  Literale sind, also

$$L_{i,j} \in \{A_1, A_2, \dots\} \cup \{\neg A_1, \neg A_2, \dots\}$$

Die Klauselschreibweise von  $F$  ist dann:

$$F = \{\{L_{1,1}, \dots, L_{1,n}\}, \dots, \{(L_{m,1}, \dots, L_{m,n})\}\}$$

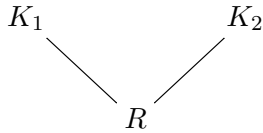
### II.5.1. Definition (Resolvent)

- Es seien  $K_1$  und  $K_2$  Klauseln mit  $A \in K_1$  und  $\neg A \in K_2$ , wobei  $A$  eine atomare Formel ist. Dann heit

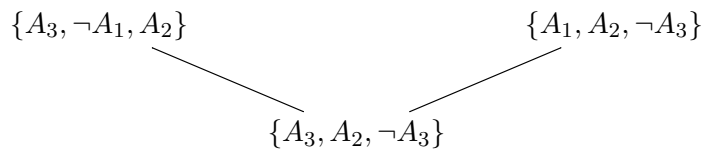
$$R = (K_1 \setminus \{A\}) \cup (K_2 \setminus \{\neg A\})$$

der *Resolvent* von  $K_1$  und  $K_2$  (bezüglich  $A$ ).

- Diagramm:



- Falls  $K_1 = \{A\}$  und  $K_2 = \{\neg A\}$ , dann ist  $R = \emptyset$
- Eine Klauselmengen, die  $\emptyset$  enthält, ist unerfüllbar.
- Beispiel:



### 11.5.2. Definition ( $\text{Res}(F)$ )

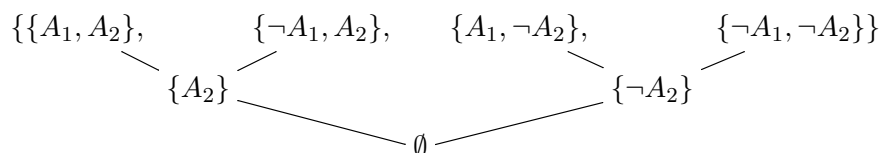
Für eine Klauselmenge  $F$  sei

$$\text{Res}(F) = F \cup \{R \mid R \text{ ist Resolvent zweier Klauseln in } F\}.$$

Weiter sei

- $\text{Res}^0(F) = F$
- $\text{Res}^{n+1}(F) = \text{Res}(\text{Res}^n(F))$
- $\text{Res}^*(F) = \bigcup_{n \geq 0} \text{Res}^n$

**Resolutionssatz der Aussagenlogik:** Eine Klauselmenge  $F$  ist genau dann unerfüllbar, wenn  $\emptyset \in \text{Res}^*(F)$



**Beispiel** Die Formel ist unerfüllbar!

### 11.5.3. Resolution in der Prädikatenlogik

Es gibt zwei Varianten

1. Zurückführen auf die aussagenlogische Resolution (Grundresolution)
2. Direkte Resolution (Unifikation)

## Grundresolution

- Stelle die Matrix der Formel  $F$  in KNF dar. Dann sind die Formeln in  $E(F)$  ebenfalls in KNF.
- Da die Formeln in  $E(F)$  variablenfrei sind kann die Resolution der Aussagenlogik verwendet werden.

**Bemerkung** Dieser Algorithmus kann nicht terminieren, falls die Formel erfüllbar ist!

## Beispiel

$$F = \forall x(P(x) \wedge \neg P(f(x)))$$

Matrix von  $F$  ist schon in KNF:

$$F^* = P(x) \wedge P(f(x))$$

als Klauselmenge:

$$\begin{aligned} F^* &= \{\{P(x)\}, \{\neg P(f(x))\}\} \\ D(F) &= \{a, f(a), f(f(a)), \dots\} \\ E(F) &= \{P(a) \wedge P(f(a)), P(f(a)) \wedge P(f(f(a))), \dots\} \end{aligned}$$

als Klauselmenge

$$E(F) = \{\{P(a)\}, \{\neg P(f(a))\}, \{P(f(a))\}, \{\neg P(f(f(a)))\}, \dots\}$$

wegen

$$\begin{array}{ccc} \{\{\neg P(f(a))\}, & & \{P(f(a))\}\} \\ & \searrow \quad \swarrow & \\ & \emptyset & \end{array}$$

ist  $F$  unerfüllbar.

**Verbesserung** Bei diesem Verfahren werden i. A. zu viele Klauseln generiert, die für die Herleitung von  $\emptyset$  nicht gebraucht werden, da die gleiche Substitution auf die ganze Klauselmenge  $F^*$  angewendet wird.

Im vorherigen Beispiel genügt es aber für die Klausel  $\{P(f(x))\}$  die Substitution  $[x/f(a)]$  und für die Klausel  $\{\neg P(f(x))\}$  die Substitution  $[x/a]$  durchzuführen um  $\emptyset$  herleiten zu können.

### II.5.4. Allgemeinsten Unifikator

- Es sei  $L = \{L_1, L_2, \dots, L_k\}$  eine Menge von prädikatenlogischen Literalen und  $\text{sub}$  eine Substitution. Wir schreiben  $L_{\text{sub}}$  für die Menge  $\{L_{1\text{sub}}, \dots, L_{k\text{sub}}\}$ .
- $\text{sub}$  ist ein *Unifikator* für  $L$ , falls  $|L_{\text{sub}}| = 1$ , dann heißt  $L$  *unifizierbar*
- Ein Unifikator  $\text{sub}$  von  $L$  heißt *allgemeinster Unifikator* von  $L$ , wenn es für jeden Unifikator  $\text{sub}'$  von  $L$  eine Substitution  $s$  gibt mit  $\text{sub}' = \text{sub}s$



## II.6. Prädikatenlogische Resolution

Es seien  $K_1, K_2$  prädikatenlogische Klauseln mit

1. Es gibt Substitutionen  $s_1$  und  $s_2$ , so dass  $K_1s_1$  und  $K_2s_2$  keine gemeinsamen Variablen enthalten.
2. Es gibt Literale  $L_1, \dots, L_m \in K_1s_1$  ( $m \geq 1$ ) und  $L'_1, \dots, L'_n \in K_2s_2$  ( $n \geq 1$ ), so dass  $L = \{\neg L_1, \dots, \neg L_m, L'_1, \dots, L'_n\}$  unifizierbar ist. Es sei  $sub$  ein allgemeinsten Unifikator von  $L$ .

Dann heißt die Klausel

$$R := ((K_1s_1 \setminus \{L_1, \dots, L_m\}) \cup (K_2s_2 \setminus \{L'_1, \dots, L'_n\}))sub$$

ein *prädikatenlogischer Resolvent* von  $K_1$  und  $K_2$ .



# III. Algorithmen

25.04.2005

## Literatur

- THOMAS H. CORMEN, CHARLES. E. LEISERSON, RONALD RIVEST, CLIFFORD STEIN: *Algorithmen – Eine Einführung*.
- DONALD E. KNUTH: *The Art of Computer Programming*.

## Konventionen Pseudocode

- Blockstruktur wird nur durch Einrücken gekennzeichnet (keine Klammern).
- Schleifenkonstrukte **while** und **repeat** wie üblich.
- Bei **for** bleibt der Wert nach Verlassen der Schleife erhalten.
- Alles nach „//“ ist Kommentar. (Buch: ▷[, %] ...)
- Mehrfachzuweisungen  $x \leftarrow y \leftarrow z$  bedeutet  $y \leftarrow z; x \leftarrow y$ .
- Variablen sind lokal (local).
- Zugriff auf die Feldelemente:  $A[i]$  das  $i$ -te Element.
- Datenattribute z. B.  $\text{länge}(A)$ .
- Parameter einer Prozedur: **call by value**.
- „und“ und „oder“ sind träge (lazy) Operatoren.

## Rundungsfunktion / Gaußklammer

$\lceil p/q \rceil$  „ceiling function“

$\lfloor p/q \rfloor$  „floor function“

$$p = 7, q = 3$$

$$p/q = 7/3 = 2,3...$$

$$\lceil p/q \rceil = \lceil 7/3 \rceil = 3$$

$$\lfloor p/q \rfloor = \lfloor 7/3 \rfloor = 2$$

## III.1. Definition

(mehrere Definitionen, Anzahl Bücher  $\gg 100$ )

- Algorithmus  $\equiv$  Berechnungsprozedur (allgemeine  $\rightarrow$  sehr spezialisierte)
- Prozedur ist deterministisch oder nicht
- deterministisch  $\equiv$  {endlich, definiert, eindeutig}

Algorithmus  $\equiv$  Analyse, Komplexität, effiziente Berechnungsmethoden

## III.2. Analyse von Algorithmen

### III.2.1. Das Sortierproblem

- Eingabe: Eine Folge von  $n$  Zahlen  $(a_1, a_2, \dots, a_n)$ .
- Ausgabe: Eine Permutation  $(b_1, b_2, \dots, b_n)$  der Eingabefolge mit  $b_1 \leq b_2 \leq \dots \leq b_n$ .

### III.2.2. Implementierung: Insertion-Sort

INSERTION-SORT( $A$ )

```

1  for  $j \leftarrow 2$  to  $\text{laenge}[A]$ 
2      do  $\text{schluessel} \leftarrow A[j]$ 
3           $\triangleright$  Füge  $A[j]$  in die sortierte Sequenz  $A[1..j-1]$  ein.
4           $i \leftarrow j-1$ 
5          while  $i > 0$  und  $A[i] > \text{schluessel}$ 
6              do  $A[i+1] \leftarrow A[i]$ 
7                   $i \leftarrow i-1$ 
8           $A[i+1] \leftarrow \text{schluessel}$ 

```

**Beispiel:**

(a) 

5	<b>2</b>	4	6	1	3
---	----------	---	---	---	---

(b) 

2	5	<b>4</b>	6	1	3
---	---	----------	---	---	---

(c) 

2	4	5	<b>6</b>	1	3
---	---	---	----------	---	---

(d) 

2	4	5	6	<b>1</b>	3
---	---	---	---	----------	---

(e) 

1	2	4	5	6	<b>3</b>
---	---	---	---	---	----------

(f) 

1	2	3	4	5	6
---	---	---	---	---	---

### III.2.3. Aufwandsklassen

- Obere asymptotische Schranke

$$O(g(n)) = \{f(n) \mid \text{es gibt } c, n_0 > 0 \text{ mit } 0 \leq f(n) \leq cg(n) \text{ für alle } n > n_0\}$$

- Untere asymptotische Schranke

$$\Omega(g(n)) = \{f(n) \mid \text{es gibt } c, n_0 > 0 \text{ mit } 0 \leq cg(n) \leq f(n) \text{ für alle } n \geq n_0\}$$

- Asymptotisch scharfe Schranke

$$\Theta(g(n)) = \{f(n) \mid \text{es gibt } c_1, c_2, n_0 > 0 \text{ mit } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ für alle } n \geq n_0\}$$

## III.2.4. Analyse von Insertion Sort

	Kosten	Zeit
0    INSERTION-SORT(A)		
1 <u>for</u> j <- 2 <u>to</u> länge[A]	$c_1$	$n$
2 <u>do</u> schlüssel <- A[j]	$c_2$	$n-1$
3        //setze A[j] ein ...	0	$n-1$
4        i <- j - 1	$c_4$	$n-1$
5 <u>while</u> i > 0 und A[i] > schlüssel	$c_5$	$\sum_{j=2}^n t_j$
6 <u>do</u> A[i + 1] <- A[i]	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7            i <- i - 1	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8        A[i + 1] <- schlüssel	$c_8$	$n-1$

## III.3. Aufwandsanalyse

Durch Summieren der Produkte aus Kosten und Zeit:

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

Günstigster Fall: Das Feld ist schon sortiert

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_6) \\ &\Rightarrow \text{lineare Laufzeit} \end{aligned}$$

Schlechtester Fall: Das Feld ist in umgekehrter Reihenfolge sortiert

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

Im Folgenden werden wir meistens nur die Laufzeit im schlechtesten Fall analysieren, denn

- der schlechteste Fall bietet eine obere Schranke für die maximale Laufzeit,
- für einige Algorithmen tritt der schlechteste Fall häufig auf: z. B. Suche in einer Datenbank,
- der „mittlere Fall“ ist oft annähernd genauso schlecht wie der schlechteste Fall.

## III.3.1. Methode: Teile und Beherrsche

- Teile das Problem in eine Anzahl von Teilproblemen auf
- Beherrsche die Teilprobleme durch rekursives Lösen bis sie so klein sind, dass sie direkt gelöst werden können.
- Verbinde die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

### III.3.2. Laufzeiten

- $\lg n$
- $\sqrt{n}$
- $n$
- $n \cdot \lg n$
- $n^2$
- $n^3$
- $2^n$
- $n!$

### III.3.3. Implementierung: MERGE-SORT

- **Teile** die zu sortierende Sequenz der Länge  $n$  in zwei Teilsequenzen der Länge  $\frac{n}{2}$
- **Beherrsche** durch rekursives Anwenden von MERGE-SORT auf die zwei Teilsequenzen
- **Verbinde** die zwei Teilsequenzen durch Mischen (merge)

#### Pseudocode

MERGE-SORT( $A, p, r$ )

```

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

```

MERGE( $A, p, q, r$ )

```

1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  Erzeuge die Felder  $L[1..(n_1 + 1)]$  und  $R[1..(n_2 + 1)]$ 
4  for  $i \leftarrow 1$  to  $n_1$ 
5      do  $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7      do  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow R[n_2 + 1] \leftarrow \infty$   $\triangleright$  Wächter
9   $i \leftarrow j \leftarrow 1$ 
10 for  $k \leftarrow p$  to  $r$ 
11     do if  $L[i] < R[j]$ 
12         then  $A[k] \leftarrow L[i]$ 
13              $i \leftarrow i + 1$ 
14         else  $A[k] \leftarrow R[j]$ 
15              $j \leftarrow j + 1$ 

```

### III.3.4. Laufzeitanalyse

- Im allgemeinen Teile- und Beherrsche-Fall gilt: Sei  $T(N)$  die Laufzeit für ein Problem der Größe  $n$ . Ist  $n$  hinreichend klein  $n \leq c$ , dann benötigt die direkte Lösung eine konstante Zeit  $\Theta(1)$ . Führt die Aufteilung des Problems zu  $a$  Teilproblemen der Größe  $1/b$  und braucht die Aufteilung  $D(n)$  Zeit und das Verbinden zum ursprünglichen Problem die Zeit  $C(n)$  so gilt:

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

- Im Fall von Merge-Sort ist  $a = b = 2$  und  $c = 1$ , also

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 2(T(n/2)) + dn & \text{sonst} \end{cases}$$

- Man kann die Problemgröße nur  $\log_2(n)$  oft aufteilen.
- Beim  $i$ -ten Aufteilen hat man  $2^i$  Teillisten der Größe  $n/2^i$  zu lösen und benötigt dafür  $dn$  Zeit
- Somit braucht man insgesamt  $dn \log_2 n + dn$  Zeit.

## III.4. Wachstum von Funktionen

27.04.2005

Zeitaufwand eines Algorithmus:

$$T(n), \quad n \in \mathbb{N}_0$$

### III.4.1. Asymptotische Notation - $\Theta$ -Notation

Asymptotisch scharfe Schranke:

$$\Theta(g(n)) = \{f(n) \mid \text{es gibt } c_1, c_2, n_0 > 0 \text{ mit } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ für alle } n \geq n_0\}$$

**Bemerkung:**  $f \in \Theta(g)$  folgt  $f$  ist asymptotisch nicht negativ, d.h. es gibt ein  $n_0$  mit  $f(n) \geq 0$  für alle  $n \geq n_0$

**Beispiele:**

- Konstanten:  $\Theta(c), c \geq 0$ :  $\Theta(c) = \Theta(1)$  ( $c_1 = c_2 = c, n_0 = 0$ )
- Monome:  $f(x) = ax^n$ . zu zeigen:  $f \in \Theta(x^n)$ . Wegen der Bemerkung gilt:  $a > 0$ . Somit  $c_1 = a, c_2 = a, n_0 = 0$ . Aber:  $ax^n \notin \Theta(x^{n+1})$ , denn für alle  $c > 0$  gilt: für alle  $x > \frac{a}{c_2}$  ist  $ax^n < c_2 x^{n+1}$
- Polynome:  $f(x) = \sum_{i=0}^n a_i x^i, a_n \neq 0$ . zu zeigen:  $f(x) \in \Theta(x^n)$ . Auch hier:  $a_n > 0$  wegen Bemerkung und Monomen. Wähle  $c_1 = \min_{i=0}^n |a_i|, c_2 = \sum_{i=0}^n |a_i|, n_0 > c_2$ .  $c_1 x^n \leq \sum_{i=0}^n a_i x^i \leq c_2 x^n$

### III.4.2. Obere Asymptotische Schranke - O-Notation

$$O(g(n)) = \{f(n) \mid \text{es gibt } c, n_0 > 0 \text{ mit } 0 \leq f(n) \leq cg(n) \text{ für alle } n > n_0\}$$

Klar:  $ax^k \in O(x^m)$  für  $k \leq m$ .

$a > 0$ , Die Bemerkung gilt auch hier! ( $c = a, n_0 = 0$ ). Ebenso:  $\sum_{i=0}^k a_i x^i \in O(x^m)$  für  $k \leq m$ .

### III.4.3. Untere Asymptotische Schranke: $\Omega$ -Notation

$$\Omega(g(n)) = \{f(n) \mid \text{es gibt } c, n_0 > 0 \text{ mit } 0 \leq cg(n) \leq f(n) \text{ für alle } n \geq n_0\}$$

Klar:  $a^k \in \Omega(x^m)$  für  $m \leq k$ . Wähle  $c = a, n_0 = 0$ .

ebenso:  $\sum_{i=0}^k a_i x^i \in \Omega(x^m)$  für  $m \leq k$ .

### III.4.4. Verhältnis der Mengen

für beliebige  $f(n)$  und  $g(n)$  gilt:

$$f(n) \in \Theta(g) \text{ genau dann, wenn } f(n) \in O(g) \text{ und } f(n) \in \Omega(g)$$

Anmerkung:  $\Theta$  ist eine Äquivalenzklasse,  $\Omega, O$  sind keine Äquivalenzklassen, da die Symmetriebedingung nicht erfüllt ist. <sup>1</sup>

## III.5. Rekurrenzen - Rekursionsgleichungen

**Problem:** Gegeben ist eine Rekurrenz  $F_n$ .

Gesucht:  $f(X)$  in geschlossener Form mit  $F_n \in \Theta(f)$ .

### III.5.1. 1. Methode: „Raten und Induktion“

**Beispiel:**  $F_0 = 1, F_1 = 1, F_{n+1} = F_n + F_{n-1}$

$n$	$F_n$		
1	1	+0	+1
2	1	+1	+0
3	2	+1	+1
4	3	+2	+1
5	5	+3	+2
6	8	+5	+3
7	13		

Vermutung:  $f(x) = ae^{bx} + c$

**Weiteres Beispiel:**

$n$	$F_n$		
0	-4	+1	·2
1	-3	+2	·2
2	-1	+4	·2
3	3	+8	·2
4	11	+16	·2
5	27		

Ansatz:  $F(n) = a2^n + c$

---

<sup>1</sup>War wohl zunächst in der Übung falsch, wurde aber gleich korrigiert.



### III.5.2. Rekursionsbaummethode

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$$

Komischesschaubild womaneigentlich nichtserkennt und woraus man was folgern kann.

### III.5.3. Weitere Methoden

- Jordan-Normalform:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

- Z-Transformierte.

## III.6. Die o-Notation

02.05.2005

$$o(g(n)) = \left\{ \begin{array}{l} f(n) : \text{für jede positive Konstante } c > 0 \\ \text{existiert eine Konstante } n_0 > 0, \text{ sodass} \\ 0 \leq f(n) < c \cdot g(n) \text{ für alle } n \geq n_0 \end{array} \right\}$$

Die Definition der  $O$ -Notation und der  $o$ -Notation sind einander ähnlich. Der Unterschied besteht darin, dass in  $f(n) = O(g(n))$  die Schranke  $0 \leq f(n) \leq cg(n)$  für eine Konstante  $c > 0$  gilt, während sie in  $f(n) = o(g(n))$  für alle Konstanten gilt.

Die Funktion  $f(n)$  (in der  $o$ -Notation) ist unbedeutend gegenüber  $g(n)$ , wenn

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

## III.7. Die $\omega$ -Notation

$$\omega(g(n)) = \left\{ \begin{array}{l} f(n) : \text{für jede positive Konstante } c > 0 \\ \text{existiert eine Konstante } n_0 > 0, \text{ sodass} \\ 0 \leq c \cdot g(n) < f(n) \text{ für alle } n \geq n_0 \end{array} \right\}$$

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

$$[n^2/2 = \omega(n); n^2/2 \neq \omega(n^2)]$$

## III.8. Lösen von Rekurrenzen mit der Generierenden-Funktion

(generating function)

Gegeben sei eine Folge  $\langle g_n \rangle$

Gesucht ist eine geschlossene Form für  $g_n$ . Die folgenden 4 Schritte berechnen diese (*closed form*)

1. Finde eine einzige Gleichung, die  $g_n$  anhand anderer Elemente der Folge ausdrückt. Die Gleichung sollte unter der Annahme  $g_{-1} = g_{-2} = \dots = 0$  für alle ganzen Zahlen ( $\mathbb{Z}$ ) gelten.

### III. Algorithmen

- Multipliziere beide Seiten mit  $z^n$  und summiere über alle  $n$ . Auf der linken Seite steht nun

$$\sum_n g_n z^n = G(z) \text{ die generierende Funktion}$$

Die rechte Seite der Gleichung sollte nun so manipuliert werden, dass sie andere Ausdrücke in  $G(z)$  enthält.

- Löse die resultierende Gleichung und erhalte damit eine geschlossene Form für  $G(z)$ .
- Expandiere diese Form von  $G(z)$  in eine Potenzreihe und betrachte die Koeffizienten von  $z^n$ . Das ist eine geschlossene Form für  $g_n$ .

**Beispiel** Die Fibonacci-Zahlen

$$g_0 = 0; \quad g_1 = 1; \quad g_n = g_{n-1} + g_{n-2} \quad (n \geq 2)$$

Schritt 1: Die Gleichung  $g_n = g_{n-1} + g_{n-2}$  ist nur für  $n \geq 2$  zulässig, denn unter der Annahme  $g_{-1} = 0, \quad g_{-2} = 0$  ist  $g_0 = 0, \quad g_1 = 0, \quad \dots$

$$g_n \stackrel{?}{=} \begin{cases} 0, & \text{falls } n = 1 \\ 1, & \text{falls } n = 2 \\ g_{n-1} + g_{n-2}, & \text{sonst} \end{cases}$$

$\Rightarrow$  Nein

$$[n = 1] = \begin{cases} 1, & \text{falls } n = 1 \\ 0, & \text{sonst} \end{cases}$$

$$\Rightarrow g_n = g_{n-1} + g_{n-2} + [n = 1]$$

Schritt 2:

$$\begin{aligned} G(z) &= \sum_n g_n z^n = \sum_n g_{n-1} z^n + \sum_n g_{n-2} z^n + \sum [n = 1] z^n \\ &= \sum_n g_n z^{n+1} + \sum_n g_n z^{n+2} + z \\ &= z \sum_n g_n z^n + z^2 \sum_n g_n z^n + z \\ &= zG(z) + z^2 G(z) + z \end{aligned}$$

Schritt 3: ist hier einfach

$$G(z) = \frac{z}{1 - z - z^2}$$

Schritt 4: Gesucht ist eine Darstellung von

$$\frac{z}{1 - z - z^2} = \frac{z}{(1 - \Phi z)(1 - \hat{\Phi} z)} \quad (\Phi: \text{„Goldener Schnitt (engl.: golden ratio)“})$$

$$\text{mit } \Phi = \frac{1 + \sqrt{5}}{2}; \quad \hat{\Phi} = \frac{1 - \sqrt{5}}{2}$$

als formale Potenzreihe. Eine Partialbruchzerlegung ergibt

$$\frac{1/\sqrt{5}}{1 - \Phi z} - \frac{1/\sqrt{5}}{1 - \hat{\Phi} z}$$

Es existiert folgende Regel

$$\frac{a}{(1 - pz)^{m+1}} = \sum_{n \geq 0} \binom{m+n}{m} ap^n z^n$$

Somit ist

$$\frac{1/\sqrt{5}}{1 - \Phi z} - \frac{1/\sqrt{5}}{1 - \hat{\Phi} z} = \sum_{n \geq 0} \frac{1}{\sqrt{5}} \Phi^n z^n + \sum_{n \geq 0} \frac{1}{\sqrt{5}} \hat{\Phi}^n z^n$$

und für den  $n$ -ten Koeffizienten gilt  $F_n = \frac{\Phi^n - \hat{\Phi}^n}{\sqrt{5}}$

### III.9. Notationen

Die *floor* und die *ceiling* Funktion:

$\lceil z \rceil$	kleinste obere Ganzzahl the least integer greater than or equal to $z$
$\lfloor z \rfloor$	größte untere Ganzzahl the greatest integer less than or equal to $z$

### III.10. Die Mastermethode

„Rezept“: Methode zur Lösung von Rekurrenzgleichungen der Form

$$T(n) = aT(n/b) + f(n)$$

wobei  $a \geq 1$ ,  $b > 1$  und  $f(n)$  eine asymptotisch positive Funktion.

**Beispiel** Merge-Sort

$$a = 2, b = 2, f(n) = \Theta(n)$$

### III.11. Mastertheorem

Seien  $a \geq 1$  und  $b > 1$  Konstanten. Sei  $f(n)$  eine Funktion und sei  $T(n)$  über die nichtnegativen ganzen Zahlen durch die Rekursionsgleichung  $T(n) = aT(n/b) + f(n)$  definiert, wobei wir  $n/b$  so interpretieren, dass damit entweder  $\lfloor n/b \rfloor$  oder  $\lceil n/b \rceil$  gemeint ist. Dann kann  $T(n)$  folgendermaßen asymptotisch beschränkt werden.

1. Wenn  $f(n) = O(n^{\log_b a - \epsilon})$  für eine Konstante  $\epsilon > 0$  erfüllt ist, dann gilt  $T(n) = \Theta(n^{\log_b a})$
2. Wenn  $f(n) = \Theta(n^{\log_b a})$  erfüllt ist, dann gilt  $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$
3. Wenn  $f(n) = \Omega(n^{\log_b a + \epsilon})$  für  $\epsilon > 0$  erfüllt ist und wenn  $a \cdot f(n/b) \leq c \cdot f(n)$  für eine Konstante  $c < 1$  und hinreichend große  $n$  gilt, dann ist  $T(n) = \Theta(f(n))$ .

$\implies$  Im ersten Fall muss  $f(n)$  nicht nur kleiner als  $n^{\log_b a}$  sein, sondern sogar polynomial kleiner. Das heißt  $f(n)$  muss für  $t > 0$  und den Faktor  $n^t$  asymptotisch kleiner sein, als  $n^{\log_b a}$ .

$\implies$  Im dritten Fall muss  $f(n)$  nicht nur größer sein als  $n^{\log_b a}$ , sondern polynomial größer und zusätzlich die „Regularitätsbedingung“  $a \cdot f(n/b) \leq c \cdot f(n)$  erfüllen.

**Beispiel 1**  $T(n) = 9T(n/3) + n$

$a = 9$ ,  $b = 3$ ,  $f(n) = n$  und somit  $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$ . Da  $f(n) = O(n^{\log_3 9 - \epsilon})$  mit  $\epsilon = 1$  gilt, können wir Fall 1 anwenden und schlussfolgern, dass  $T(n) = \Theta(n^2)$  gilt.

**Beispiel 2**  $T(n) = T(2n/3) + 1$

$a = 1$ ,  $b = 3/2$ ,  $f(n) = 1$  also  $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$  da  $f(n) = \Theta(n^{\log_b a}) = \Theta(1) \Rightarrow$  Lösung:  $T(n) = \Theta(\lg n)$

**Beispiel 3**  $T(n) = 3T(n/4) + n \lg n$

$a = 3$ ,  $b = 4$ ,  $f(n) = n \lg n$  also  $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$  Da  $f(n) = \Omega(n^{\log_4 3 + \epsilon})$  mit  $\epsilon \approx 0,2$  gilt, kommt Fall 3 zur Anwendung  $\Rightarrow T(n) = \Theta(n \lg n)$

Die Mastermethode ist auf  $T(n) = 2T(n/2) + n \lg n$  nicht anwendbar auch wenn sie die korrekte Form hat:  $a = 2$ ,  $b = 2$ ,  $f(n) = n \lg n$ . Das Problem besteht darin, dass  $f(n)$  nicht polynomial größer ist als  $n^{\log_b a}$ . Das Verhältnis  $f(n)/(n^{\log_b a}) = (n \lg n)/n = \lg n$  ist asymptotisch kleiner als  $n^\epsilon$  für jede Konstante  $\epsilon > 0$ .

## III.12. Probabilistische Algorithmen (zufallsgesteuerte Algorithmen)

**Numerische Algorithmen**  $\approx 1950 \rightarrow$  Integration.

$$\int_0^1 \cdots \int_0^1 f(\alpha_1, \dots, \alpha_n) d\alpha_1, \dots, d\alpha_n$$

- Monte-Carlo Methode  $n = 2$   $\hat{=}$  komische Graphiken  $\hat{=}$

Das Konzept wurde 1976 von Rabin effektiver formuliert.

### III.12.1. Einführung

Wir beginnen mit der Definition eines deterministischen Algorithmus nach Knuth.

1. Eine Berechnungsmethode ist ein Quadrupel  $(Q, I, O, F)$  mit  $I \subset Q, O \subset Q, f : Q \rightarrow Q$  und  $f(p) = p', \forall p \in J$ .  $Q$  ist der Zustand der Berechnung,  $I$ : Eingabe,  $O$ : Ausgabe und  $f$ : Berechnungsregeln.
  - Jedes  $x \in J$  ergibt eine Folge  $x_0, x_1, \dots$  die durch  $x_{k+1} = f(x_k), k \geq 0, x_0 = x$  definiert ist.
  - Die Folge terminiert nach  $k$  Schritten ( $K$ : kleiner als die  $k$ 's, sodass gilt:  $x_k \in O$ )
2. Eine Algorithmus ist eine Berechnungsmethode, die in endlich vielen Schritten für alle  $x \in I$  terminiert.
3. Ein deterministischer Algorithmus ist eine formale Beschreibung für eine *endliche, definite* Prozedur, deren Ausgaben zu beliebigen Eingaben *eindeutig* sind.

Hauptmotivation zur Einführung von probalistischen Algorithmen stammt aus der Komplexitätsanalyse. Man unterscheidet zwischen dem *Verhalten im schlechtesten Fall* und dem *Verhalten im mittleren Fall* eines Algorithmus. Diese Fälle sind festgelegt, sobald das Problem und die Daten bestimmt sind.

Die hervorgehobenen Wörter in den Definitionen eines deterministischen Algorithmus sind die Schlüssel zur Definition von drei Arten von probalistischen Algorithmen:

1. Macao-Algorithmus (1. Art) (auch Sherwood Algorithmus)

- mindestens bei einem Schritt der Prozedur werden einige Zahlen zufällig ausgewählt (nicht definit)
- sonst: deterministisch

$\Rightarrow$  immer eine korrekte Antwort

Wird benutzt, wenn irgendein bekannter Algorithmus (zur Lösung eines bestimmten Problems) im mittleren Fall viel schneller als im schlechtesten Fall läuft.

2. Monte-Carlo-Algorithmus (2. Art)

- gleich wie Algorithmus 1. Art (nicht definit)
- mit einer Wahrscheinlichkeit von  $1 - \epsilon$ , wobei  $\epsilon$  sehr klein ist (nicht eindeutig)

$\Rightarrow$  immer eine Antwort, wobei die Antwort nicht unbedingt richtig ist  
( $\epsilon \rightarrow 0$  falls  $t \rightarrow \infty$ )

3. Las-Vegas-Algorithmus (3. Art)

- Gleich wie Macao-Algorithmus (nicht definit).
- Eine Folge von zufälligen Wahlen kann unendlich sein (mit einer Wahrscheinlichkeit  $\epsilon \rightarrow 0$ ) (nicht endlich).

### III.12.2. Macao-Algorithmen („Nächstes-Paar“-Algorithmus)

Problem:  $x_1, \dots, x_n$  seien  $n$  Punkte in einem  $k$ -dimensionalen Raum  $R^k$ . Wir möchten das nächste Paar  $x_i, x_j$  finden, sodass gilt:

$$d(x_i, x_j) = \min\{d(x_p, x_q)\} \quad (1 \leq p < q \leq n),$$

wobei  $d$  die gewöhnliche Abstandsfunktion aus  $R^k$  ist.

### III.12.3. Brute-Force-Methode („Brutaler Zwang“-Methode)

Evaluiert alle  $\frac{n(n-1)}{2}$  relevanten gegenseitigen Abstände.

$\Rightarrow$  minimaler Abstand

$\Rightarrow O(n^2)$

### III.12.4. Deterministische Algorithmen (Yuval)

$\Rightarrow O(n \log n)$

Idee: man wählt eine Hülle  $S = \{x_1, \dots, x_n\}$  und sucht das nächste Paar innerhalb dieser Hülle.

Schlüsselidee: eine Teilmenge von Punkten wird zufällig ausgewählt  $\Rightarrow$  Parl. (???) Alg. (Macao) mit  $O(n)$  und mit sehr günstiger Konstante.

1. Wähle zufällig  $S_1 = \{x_{i_1}, \dots, x_{i_m}\}$

$m = n^{2/3}$

$m$  = Kardinalität  $S_1$  (= Anzahl von Elementen in  $S_1$ )

### III. Algorithmen

2. Berechne  $\delta(S_1) = \min\{(x_p, x_q)\}$   
für  $x_p, x_q \in S_1 \Rightarrow O(n)$   
Wir iterieren einmal den gleichen Algorithmus für  $S_1$ , indem man  $S_2 \subset S_1$  mit  $c(S_2) = m^{2/3} = n^{4/9}$  zufällig auswählt.  
 $\dots O(n)$
3. Konstruieren eines quadratischen Verbandes  $\Gamma$  mit der Netzgröße (*mesh size*)  $\delta = \delta(S_1)$ .
4. Finde für jedes  $\Gamma_i$  die Dekomposition  $S = S_1^{(i)} \cup \dots \cup S_k^{(i)}, 1 \leq i \leq 4$  (anders als Hashing-Techniken)
5.  $\forall x_p, x_q \in S_j^{(i)}$  berechne  $d(x_p, x_q) \Rightarrow$  Das nächste Paar ist unter diesen Paaren zu finden.  
Leite ab aus  $\Gamma$  durch Verdopplung der Netzgröße auf  $2\delta$

**Lemma zu 3.:** Gilt  $\delta(S) \leq \delta$  ( $\delta$  ist Netzgröße von  $\Gamma$ ), so existiert ein Verbandspunkt  $y$  auf  $\Gamma$ , so dass das nächste Paar im Quadrupel von Quadraten aus  $\Gamma$  direkt und rechts von  $y$  liegt.  
 $\Rightarrow$  es ist garantiert, dass das nächste Paar  $x_i, x_j$  aus  $S$  innerhalb eines gleichen Quadrats aus  $\Gamma_i$  liegt ( $1 \leq i \leq 4$ )

#### III.12.5. Monte-Carlo-Algorithmus

$\rightarrow$  Miller-Rabin Primzahl Algorithmus

Ganze Zahlen: 2 Probleme

- Primzahltest
- Faktorisierung

Algorithmus stellt fest, ob eine Zahl  $n$  prim ist (pseudo-prim). In diesem Algorithmus werden  $m$  Zahlen  $1 \leq b_1, \dots, b_m < n$  zufällig ausgewählt. Falls für eine gegebene Zahl  $n$  und irgendein  $\epsilon > 0$   $\log_2 \frac{1}{\epsilon} \leq m$  gilt, dann wird der Algorithmus die korrekte Antwort mit Wahrscheinlichkeit größer als  $(1-\epsilon)$  liefern.

Grundidee: Ergebnisse aus Zahlentheorie (*Millers Bedingung*) für eine ganze Zahl  $b$ .

09.05.2005

#### Einschub Modular-Arithmetik (Gauß (1801))

**Definition** Seien  $a, b, N \in \mathbb{Z}$ . Dann schreibt man:

$$a \equiv b \pmod{N} \Leftrightarrow N \mid (a - b)$$

„ $\mid$ “ bedeutet „teilt“

#### Beispiele

- $7 \bmod 5 = 2$
- „mod 5“ bedeutet Rechnen mit 0, 1, 2, 3, 4
- $a \equiv b \bmod N \wedge c \equiv d \bmod N \Rightarrow a + c \equiv (b + d) \bmod N$
- $a \bmod N + b \bmod N = (a + b) \bmod N$

### Witnessfunktion

Für eine ganze Zahl  $b$  erfülle  $W_n(b)$  folgende Bedingungen:

1.  $1 \leq b < n$
2. a)  $b^{n-1} \not\equiv 1 \pmod{n}$  oder  
 b)  $\exists i : 2^i | (n-1)$  und  $1 < \text{ggT}(b^{\frac{n-1}{2^i}} - 1, n) < n$   $\left[ \frac{n-1}{2^i} \equiv m \right]$

Eine ganze Zahl, die diese Bedingung erfüllt, wird *Zeuge (witness)* für die Teilbarkeit von  $n$  genannt.

$\stackrel{2a}{\Rightarrow}$  Die *Fermat'sche Relation* ist verletzt. (Fermat:  $b^{n-1} \equiv 1 \pmod{n}$ )  $\stackrel{2b}{\Rightarrow}$   $n$  hat einen echten Teiler  $\Rightarrow$  ist  $n$  teilbar, so gilt  $W_n(b)$  (**FIXME:** Stimmt das so?)  $\Rightarrow n$  ist keine Primzahl.

Ist  $n$  teilbar, so gibt es viele Zeugen.

**Theorem (Anzahl der Zeugen)** Wenn  $n \geq 4$  teilbar ist, dann gilt

$$3(n-1)/4 \leq c(\{b | 1 \leq b < n, W_n(b) \text{ gilt} \})$$

$\Rightarrow$  nicht mehr als  $1/4$  der Zahlen  $1 \leq b < n$  sind keine Zeugen.

### Algorithmus: Rabins Algorithmus

*Eingabe:*  $n$  ungerade, ganze Zahl mit  $n > 1$

*Ausgabe:*  $b = \pm 1$ , falls entschieden ist, dass  $n$  prim ist  
 $b = 0$ , falls  $n$  teilbar ist

RABIN( $n$ )

- 1 Wähle zufällig  $a$  aus  $1 \leq a < n$
- 2 Faktorisiere  $(n-1)$  zu  $2^l m$  so, dass  $n-1 = 2^l m$ ,  $m$  ungerade
- 3 (Teste)  $b = a^m \pmod{n}$ ,  $i = 1$
- 4 **while**  $b \neq -1$  und  $b \neq 1$  und  $i < l$
- 5     **do**  $b \leftarrow b^2 \pmod{n}$
- 6      $i \leftarrow i + 1$
- 7 **if**  $b = 1$  oder  $b = -1$
- 8     **then**  $n$  ist Primzahl (prime)
- 9     **else**  $n$  ist *keine Primzahl (composite)*, ( $b = 0$ )

**Bemerkung:** Der Algorithmus braucht  $m(2+l) \log_2(n)$  Schritte  $\Rightarrow$  sehr effizient.

### III.12.6. Las-Vegas-Algorithmen

**Beispiel 1:**  $M$  sei eine  $n$ -elementige Menge,  $S_0, \dots, S_{k-1} \subseteq S$  mit  $|S_i| = r > 0$  seien paarweise verschiedene Teilmengen von  $S$ ,  $k \leq 2^{r-2}$

Wir wollen die Elemente von  $M$  so mit den Farben *rot* und *schwarz* färben, dass  $S_i$  wenigstens *ein* rotes und *ein* schwarzes Element enthält.

### III. Algorithmen

#### Beispiel 2: Rabin (1980)

Problem: irreduzible Polynome in endlichen Körpern zu finden.

*irreduzibel*:  $\exists$  kein Teiler, d.h.

$$n \text{ ist irreduzibel} \Leftrightarrow \forall b : \text{ nicht } b|n$$

d.h.  $Q(x)$  ist irreduzibel  $\Leftrightarrow \exists$  kein Polynom  $q(x)$  so dass  $q(x)|Q(x)$

#### Algorithmus: Irreduzibles Polynom

*Eingabe*: Primzahl  $p$  und ganze Zahl  $n$

*Ausgabe*: irreduzibles Polynom

#### Algorithmus

```
0  repeat
1    Generiere ein zufälliges Polynom  $g \in GF(p)[n]$  (FIXME: Stimmt das so?)
2    Teste die Irreduzibilität
3  until Erfolg
```

**Bemerkung:** Die Irreduzibilität wird durch 2 Theoreme geprüft:

**Theorem (Prüfung auf Irreduzibilität)** Seien  $l_1, \dots, l_n$  alle Primteiler von  $n$  und bezeichne  $n/l_i = m_i$ . Ein Polynom  $g(x) \in GF(\phi)[x]$  vom Grad  $n$  ist irreduzibel in  $GF(\phi) : \Leftrightarrow$

1.  $g(x)|(x^{p^n} - x)$
2.  $\text{ggT}(g(x), x^{p^{m_i}} - n) = 1$  für  $1 \leq i \leq k$

### III.13. Gierige Algorithmen

auch: *greedy algorithms* bzw. *Raffke-Algorithmen*

- normalerweise sehr einfach
- zum Lösen von Optimierungsproblemen

**Typische Situation:** Wir haben

- eine Menge von Kandidaten (Jobs, Knoten eines Graphen)
- eine Menge von Kandidaten, die schon benutzt worden sind
- eine Funktion, die feststellt, ob eine bestimmte Menge von Kandidaten eine Lösung zu diesem Problem ist
- eine Funktion, die feststellt, ob eine Menge von Kandidaten eine zulässige Menge ist, um die bisherige Menge so zu vervollständigen, dass mindestens eine Lösung gefunden wird
- eine Wahlfunktion (*selection function*), die in beliebiger Zeit den geeignetsten Kandidaten aus den unbenutzten Kandidaten bestimmt.
- eine Zielfunktion (*target function*), die den Wert einer Lösung ergibt (die Funktion, die zu optimieren ist)



**III.13.1. Beispiel:**

Wechselgeldausgabe an einen Kunden

**Kandidaten:** Menge von Geldstücken  $(1, 5, 10, \dots)$ , wobei jede Sorte aus mindestens einem Geldstück besteht

**Lösung:** Gesamtbetrag

**Zulässige Menge:** die Menge, deren Gesamtbetrag die Lösung nicht überschreitet

**Wahlfunktion:** Wähle das am höchsten bewertete Geldstück, das noch in der Menge der Kandidaten übrig ist

**Zielfunktion:** Die Anzahl der in der Lösung benutzten Geldstücke

**Bemerkung:** Gierige Algorithmen arbeiten schrittweise:

1. Zu Beginn ist die Liste der Kandidaten leer.
2. Bei jedem Schritt versucht man, mit Hilfe der Wahlfunktion den besten Kandidaten hinzuzufügen.
3. Falls die erwartete Menge nicht mehr zulässig ist, entfernen wir den gerade hinzugefügten Kandidaten. Er wird später nicht mehr berücksichtigt.
4. Falls die gewählte Menge noch zulässig ist, gehört der gerade Kandidat dieser Menge für immer an
5. Nachdem wir die Menge erweitert haben, überprüfen wir, ob die Menge eine Lösung des gegebenen Problems ist.

**III.13.2. Gierige Algorithmen abstrakt:**

$C = \{\text{Menge aller Kandidaten}\}$

$S \leftarrow \emptyset \{\text{Lösungsmenge}\}$

GREEDY-ALGORITHM

```

1  while nicht  $Lösung(S)$  und  $C \neq \emptyset$ 
2      do  $x \leftarrow$  ein Element aus  $C$ , das  $Wahl(X)$  maximiert
3       $C \leftarrow C \setminus x$ 
4  if  $Lösung(S)$ 
5      then return  $S$ 
6  else return „keine Lösung“
```

**III.13.3. Beispiel**

Minimale, zusammenhängende Bäume

**Einführung** Sei  $G = \langle N, A \rangle$  ein zusammenhängender, ungerichteter Graph

- $N$ : Menge von Knoten
- $A$ : Menge von Kanten (Wobei jeder Kante eine nichtnegative Länge zugeordnet wird)

11.05.2005

**Problem** Finde eine Teilmenge  $T$  von  $A$ , so dass alle Knoten zusammenhängend bleiben, wenn man nur die Kanten aus  $T$  benutzt. Dabei soll die Kantenlänge aus  $T$  so klein wie möglich gehalten werden.

**FIXME:** Bild vom Baum

### Terminologie

1. Eine Menge von Kanten ist eine Lösung, wenn sie einen zusammenhängenden Baum bildet.
2. Sie ist zulässig, wenn sie keinen Zyklus enthält.
3. Eine zulässige Menge von Kanten heißt günstig  $\Rightarrow$  optimale Lösung.
4. Eine Kante berührt eine gegebene Menge von Kanten, wenn genau ein Ende der Kante ein Element aus dieser Menge ist.

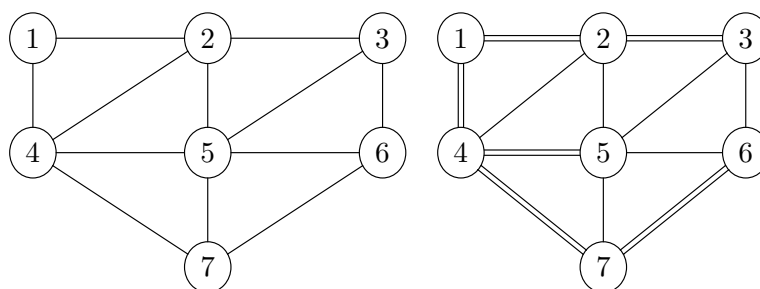
### III.13.4. Kruskalscher Algorithmus

**Beispiel** Die aufsteigende Reihenfolge der Kantenlänge ist:

$\{1, 2\}, \{2, 3\}, \{4, 5\}, \{6, 7\}, \{1, 4\}, \{2, 5\}, \{4, 7\}, \{3, 5\}, \{2, 4\}, \{3, 6\}, \{5, 7\}, \{5, 6\}$

Schritte	Berücksichtigte Kanten	Zusammengebundene Komponenten
Initialisierung	—	$\{1\}\{2\}\{3\} \dots \{7\}$
1	$\{1, 2\}$	$\{1, 2\}\{3\}\{4\} \dots \{7\}$
2	$\{2, 3\}$	$\{1, 2, 3\}\{4\}\{5\} \dots \{7\}$
3	$\{4, 5\}$	$\{1, 2, 3\}\{4, 5\}\{6\}\{7\}$
4	$\{6, 7\}$	$\{1, 2, 3\}\{4, 5\}\{6, 7\}$
5	$\{1, 4\}$	$\{1, 2, 3, 4, 5\}\{6, 7\}$
6	$\{2, 5\}$	nicht angenommen
7	$\{4, 7\}$	$\{1, 2, 3, 4, 5, 6, 7\}$

$T$  enthält die Kanten  $\{1, 2\}, \{2, 3\}, \{4, 5\}, \{6, 7\}, \{1, 4\}$  und  $\{4, 7\}$



Initialisierung

Schritt 7

### Algorithmus Kruskal

- $find(x)$ , der feststellt in welcher Komponente der Knoten  $x$  zu finden ist.
- $merge(A, B)$ : Mischen von zwei disjunkten Mengen

ALGORITHMUS KRUSKAL( $G = \langle N, A \rangle$ )

**Input:**  $N, A$  (mit Längenangabe)

**Output:** Menge von Kanten

**Initialisierung**

```

1  Sortiere  $A$  nach aufsteigender Länge
2   $n = \#N \triangleright \#$  Anzahl Knoten
3   $T \leftarrow \emptyset \triangleright$  Lösungsmenge
4  Initialisiere  $n$  disjunkte Mengen, wobei jede Menge ein Element aus  $N$  enthält.
5  repeat
6       $\{u, v\} \leftarrow$  noch nicht berücksichtigte, kürzeste Kanten
7       $ucomp \leftarrow \text{find}(u) \triangleright$  In der Menge der bereits verbundenen Kanten
8       $vcomp \leftarrow \text{find}(v)$ 
9      if  $ucomp \neq vcomp$ 
10         then  $\text{merge}(ucomp, vcomp)$ 
11          $T \leftarrow T \cup \{\{u, v\}\}$ 
12  until  $T = n - 1$ 
13 return  $T$ 
```

Analyse:  $n$  Knoten,  $a$  Kanten

- $O(a \log a)$ : Kanten zu sortieren  $\left[ n - 1 \leq a \leq \frac{n(n-1)}{2} \rightarrow \approx O(n \log n) \right]$
- $O(n)$ :  $n$  disjunkte Mengen zu initialisieren
- „find“ and „merge“. Höchstens  $2a$  Operationen (find),  $n - 1$  Operationen (merge)  $\rightarrow$  „worst case“  $O((2a + n - 1) \log^* n)$

Def:  $\log^{(0)} n = n$

$\log^{(k)} n = \log(\log^{(k-1)} n)$   $k \geq 1 \Rightarrow \log^* n = \min \left\{ i \mid \log^{(i)}(n) \leq 1 \right\}$

Man erhält:

$n$	$\log^* n$
1	0
2	1
3,4	2
$5 \rightarrow 16$	3
$17 \rightarrow 65536$	4

$\Rightarrow \log^*$  wächst sehr langsam. Höchstens  $O(a)$  für die restlichen Operationen.

### III.13.5. Prim'scher Algorithmus

ALGORITHMUS PRIM( $G = \langle N, A \rangle$ )

**Initialisierung**

- 1  $T \leftarrow \emptyset$
- 2  $B \leftarrow \{\text{ein willkürliches Element aus } N\}$

**Greedy-Schleife**

- 3 **while**  $B \neq N$
- 4     **do** Finde  $\{u, v\}$  von minimaler Länge, so dass  $u \in N \setminus B$  und  $v \in B$
- 5          $T \leftarrow T \cup \{\{u, v\}\}$
- 6          $B \leftarrow B \cup \{u\}$
- 7 **return**  $T$

Schritt	$\{u, v\}$	$B$
Initialisierung	-	$\{1\}$
1	$\{2, 1\}$	$\{1, 2\}$
2	$\{3, 2\}$	$\{1, 2, 3\}$
3	$\{4, 1\}$	$\{1, 2, 3, 4\}$
4	$\{5, 4\}$	$\{1, 2, 3, 4, 5\}$
5	$\{7, 4\}$	$\{1, 2, 3, 4, 5, 7\}$
6	$\{6, 7\}$	$\{1, 2, 3, 4, 5, 6, 7\}$

Analyse und Vergleich:

Hauptschleife (Prim)  $(n - 1)$ mal ausgeführt.

Bei jeder Iteration benötigen die for-Schleifen eine Zeit von  $O(n) \Rightarrow O(n^2)$  für Prim.

Kruskal  $O(a \log n)$

- Für dicht besetzte Graphen:

$$a \approx \frac{n(n-1)}{2} \Rightarrow O(n^2 \log n)$$

$\Rightarrow$  Prim ist „besser“.

- Für dünn besetzte Graphen:

$$a \approx n \Rightarrow O(n \log n)$$

$\Rightarrow$  Prim ist „weniger effizient“.

Kürzeste Pfade (im „Skript“)

(Dijkstra-Algorithmus)

### III.13.6. Zeitplanerstellung (Scheduling)

**Komplexitätsklassen:** P, NP

$$P \stackrel{?}{=} NP$$

Das P/NP-Problem ist ein offenes Problem der theoretischen Informatik, speziell der Komplexitätstheorie.

Es ist die Frage, ob die Klasse NP, der von nichtdeterministischen Turingmaschinen in Polynomialzeit entscheidbaren Probleme, mit der Klasse P, der von deterministischen Turingmaschinen in Polynomialzeit entscheidbaren Probleme, übereinstimmt.

Es ist also lediglich zu zeigen, dass das Finden einer Lösung für ein Problem wesentlich schwieriger ist, als nur zu verifizieren, ob eine gegebene Lösung korrekt ist. Dies ist allerdings bisher noch nicht gelungen. [...]

Das P/NP-Problem gilt derzeit als die wichtigste Fragestellung der Informatik überhaupt und wurde vom Clay Mathematics Institute in seine Liste der Millennium-Probleme aufgenommen. (Wikipedia)

**Problem:** Ein Server (Prozessor, KassiererIn einer Bank, ...) habe  $n$  Kunden in einem gegebenen System zu bedienen.

Bedienzeit für jeden Kunden ist bekannt: Bedienzeit  $t_i$  für Kunde  $i$  ( $1 \leq i \leq n$ )

$$T = \sum_{i=1}^n (\text{Gesamtzeit für Kunde } i)$$

Wir möchten  $T$  minimieren.

Beispiel:  $n = 3$ ,  $t_1 = 5$ ,  $t_2 = 10$ ,  $t_3 = 3 \Rightarrow 3! = 6$  Reihenfolgen möglich

Reihenfolge 123 bedeutet Kunde 1 wird bedient und Kunde 2 und 3 warten.

Reihenfolge	T	
123	$5 + (5+10) + (5+10+3) = 38$	
132	$5 + (5+3) + (5+3+10) = 31$	
213	$10 + (10+5) + (10+5+3) = 43$	
231	$10 + (10+3) + (10+3+5) = 43$	
312	$3 + (3+5) + (3+5+10) = 29$	← Optimum
321	$3 + (3+10) + (3+10+5) = 34$	

### III.13.7. Greedy-Algorithmus

Füge ans Ende des Zeitplans  $t_{i_1} + \dots + t_{i_m}$  den Kunden ein, der am meisten Zeit benötigt. Dieser triviale Algorithmus liefert die korrekte Antwort für  $\{3,1,2\}$ .

**Theorem:** Dieser Algorithmus ist immer optimal.

### III.13.8. Zeitplanerstellung mit Schlußterminen (deadline)

Beispiel:  $n = 4$

i	$g_i$	$d_i$
1	50	2
2	10	1
3	15	2
4	30	1

→ Reihenfolge (3,2) wird nicht berücksichtigt, da dann Auftrag 2 zum Zeitpunkt  $t = 2$  nach Schlußtermin  $t = 1$  verarbeitet wird.

### III. Algorithmen

Reihenfolge	Gewinn	
1	50	
2	10	
3	15	
4	30	
(1,3)	65	
(2,1)	60	
(2,3)	25	
(3,1)	65	
(4,1)	80	←
(4,3)	45	

⇒ es ist nicht notwendig alle  $n!$  Auftragsfolgen zu untersuchen. Es genügt eine Auftragsfolge in der Reihenfolge aufsteigender Schlußtermine zu untersuchen ( $\rightarrow (4,1)$ , aber nicht  $(1,4)$ ).

### III.14. Teile und Herrsche

18.05.2005

Die Effizienz der Teile und Herrsche-Methode liegt darin, dass Teilinstanzen schneller gelöst werden können als das Gesamtproblem.

ALGORITHMUS TEILE & HERRSCHE  $DQ(x)$

```
1  if ( $x$  ist klein genug oder einfach)
2    then return ADHOC( $x$ )
3  Teile  $x$  in kleinste Teilinstanzen,  $x_1, x_2, \dots, x_k$ 
4  for  $i \leftarrow 1$  to  $k$ 
5    do  $y_i \leftarrow DQ(x_i)$ 
6  Kombiniere die  $y_i$ s um eine Lösung  $y$  für  $x$  zu erhalten.
   ADHOC: Grundalgorithmus zur Lösung der Teilinstanzen
   Spezialfall: Wenn  $k = 1 \Rightarrow$  Vereinfachung statt Teile und Herrsche
```

#### Bedingungen

- Es ist möglich eine Instanz in Teilinstanzen zu teilen
- Es ist möglich die Teilergebnisse effizient zu kombinieren
- Die Größe der Teilinstanzen soll möglichst gleich sein
- Problem: Grundalgorithmus anstatt weiter rekursiv zu arbeiten
- Es muss gut überlegt werden, wie man den Grenzwert wählt

#### Beispiele

- Binäres Suchen
- Mergesort
- Quicksort

**III.14.1. Quicksort (C.A.R. Hoare, 1960)**

Die Funktionsweise und Analyse von Quicksort steht in nahezu allen Algorithmenbüchern.

QUICKSORT( $A, p, r$ )

```

1  if  $p < r$ 
2    then  $q \leftarrow \text{PARTITION}(A, p, r)$ 
3         QUICKSORT( $A, p, q - 1$ )
4         QUICKSORT( $A, q + 1, r$ )

```

PARTITION( $A, p, r$ )  $\Rightarrow A[p, \dots, r]$  neu geordnet.

```

1   $x \leftarrow A[r]$ 
2   $i \leftarrow p - 1$ 
3  for  $j \leftarrow p$  to  $r - 1$ 
4    do if  $A[j] \leq x$ 
5         then  $i \leftarrow i + 1$ 
6         exchange  $A[i] \leftrightarrow A[j]$ 
7  exchange  $A[i + 1] \leftrightarrow A[r]$ 
8  return  $i + 1$ 

```

ZEILEN 3-6

```

1  if  $p \leq k \leq i$ 
2    then  $A[k] \leq x$ 
3  if  $i + 1 \leq k \leq j - 1$ 
4    then  $A[k] > x$ 
5  if  $k = r$ 
6    then  $A[k] = x$ 

```

**III.14.2. Selektion und Median**

Sei  $T[1, \dots, n]$  eine Reihung der ganzen Zahlen.  $m$  ist der Median von  $T \Leftrightarrow$

1.  $m \in T$
2.  $\#\{i \in [1, \dots, n] \mid T[i] < m\} < n/2$  und  
 $\#\{i \in [1, \dots, n] \mid T[i] \leq m\} \leq n/2$

So sind auch die Möglichkeiten berücksichtigt, bei denen  $m$  ungerade ist oder nicht alle Elemente von  $T$  verschieden sind.

**Naiver Algorithmus**

Die Reihung ist in aufsteigender Ordnung zu sortieren und man erhält das  $\lceil \frac{n}{2} \rceil$ -te Element. Mit MERGESORT benötigt man dafür eine Zeit von  $O(n \log n)$ .

**Selektion-Problem**

$T$  ist Reihung der Größe  $n$ , sowie  $k \in \mathbb{Z}, 1 \leq k \leq n$ . Das  $k$ -te kleinste Element von  $T$  ist  $m$ , so dass

$$\begin{aligned} \#\{i \in [1, \dots, n] \mid T[i] < m\} &< k \text{ während} \\ \#\{i \in [1, \dots, n] \mid T[i] \leq m\} &\geq k \end{aligned}$$

### III. Algorithmen

Es ist also das  $k$ -te Element aus  $T$ , wenn die Reihung in aufsteigender Ordnung sortiert ist. Analog zum Quicksort ist es möglich folgenden Algorithmus zu entwerfen, um dieses Element zu finden:

```

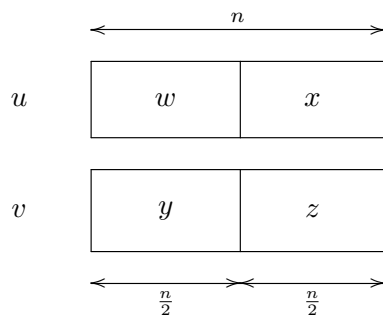
SELEKTION( $T[1, \dots, n], k$ )
1  if  $n$  ist klein
2      then SORT( $T$ )
3          return  $T[k]$ 
4   $p \leftarrow$  irgendein Element aus  $T[1, \dots, n]$ 
   { $p$  ist unser „Pivot“-Element}
5   $u \leftarrow \#\{i \in [1, \dots, n] \mid T[i] < p\}$ 
6   $v \leftarrow \#\{i \in [1, \dots, n] \mid T[i] \leq p\}$ 
7  if  $k \leq u$ 
8      then array  $U[1, \dots, n]$ 
9           $U \leftarrow$  die Elemente aus  $T$  kleiner als  $p$ 
          {das kleinste Element aus  $T$  ist auch das kleinste Element aus  $U$ }
10         return SELEKTION( $U, k$ )
11 if  $k \leq v$ 
12     then return  $p$  {Die Lösung}
13 else array  $V[1, \dots, n - v]$ 
14      $V \leftarrow$  die Elemente aus  $T$  größer als  $p$ 
       {das  $k$ -te kleinste Element aus  $T$ , ist auch das  $(k - v)$ -te kleinste Element aus  $V$ }
15     return SELEKTION( $V, k - v$ )

```

Welches Element aus  $T$  sollen wir als Pivotelement  $p$  benutzen? Die beste Wahl ist sicherlich der Median von  $T$ , so dass die Größen von  $U$  und  $V$  möglichst gleich sind.

#### III.14.3. Langzahlarithmetik

Multiplikationen zweier ganzen Zahlen von  $n$  Dezimalziffern wobei  $n$  sehr groß sein kann.



$$\begin{aligned}
 u &= 10^s w + x & 0 &\leq x \leq 10^s \\
 v &= 10^s y + z & 0 &\leq z \leq 10^s
 \end{aligned}$$

Wir suchen das Produkt

$$uv = 10^{2s}wy + 10^s(wz + xy) + xz$$

Das führt zum Algorithmus



```

MULT( $u, v$ )
1   $n \leftarrow$  die kleinste ganze Zahl so dass  $u$  und  $v$  von Größe  $n$  sind
2  if  $n$  klein
3      then return CLASSIC-PRODUCT( $u, v$ )
4   $s \leftarrow n \operatorname{div} 2$ 
5   $w \leftarrow u \operatorname{div} 10^s$ ;  $x \leftarrow u \bmod 10^s$ 
6   $y \leftarrow v \operatorname{div} 10^s$ ;  $z \leftarrow v \bmod 10^s$ 
7  return
8   $\text{MULT}(w, y) \cdot 10^{2s} + (\text{MULT}(w, z) + \text{MULT}(x, y)) \cdot 10^s + \text{MULT}(x, z)$ 

```

Eine triviale Verbesserung wird dadurch erreicht, dass man den letzten Schritt durch folgendes ersetzt:

```

1   $r \leftarrow \text{MULT}(w + x, y + z)$ 
2   $p \leftarrow \text{MULT}(w, y)$ 
3   $q \leftarrow \text{MULT}(x, z)$ 
4  return  $p \cdot 10^{2s} + (r - p - q) \cdot 10^s + q$ 

```

### Bemerkungen

- Die Komplexitätsanalyse zeigt, dass der Algorithmus eine Zeit von  $O(n^{\log_2 3}) = O(n^{1,59})$  benötigt
- mittels „Schneller Fourier-Transformation“ und Teile & Herrsche kann die Komplexität auf  $O(n \cdot \log n \cdot \log \log n)$  reduziert werden.
- Eine spezielle Version dieses Algorithmus ist als „Karatsuba-Algorithmus“ bekannt. Sei  $n$  gerade mit  $n = 2m$  und  $u, v$  ganze Zahlen der Länge  $n$  (in Bits):

$$\begin{aligned}
 u &= a2^m + b \\
 v &= c2^m + d \\
 w &= uv = y2^{2m} + (x - y - z)2^n + z
 \end{aligned}$$

wobei

$$\begin{aligned}
 x &= (a + b)(c + d) \\
 y &= ac \\
 z &= bd
 \end{aligned}$$

### III.14.4. Matrixmultiplikation

$A, B$ :  $2 \times n$ -Matrizen;  $C = A \cdot B$

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}; \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\begin{aligned}
 m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
 m_2 &= a_{11}b_{11} \\
 m_3 &= a_{12}b_{21} \\
 m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\
 m_5 &= (a_{21} - a_{22})(b_{12} - b_{11}) \\
 m_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
 m_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21})
 \end{aligned}$$

$$\Rightarrow AB = \begin{pmatrix} m_1 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 + m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Normalerweise hat der Algorithmus eine Komplexität von  $\Theta(n^2)$ . Hier jedoch nur  $\Theta(n)$ .

## III.15. Abstrakte Datentypen (ADT)

### III.15.1. Bool

#### Signaturen

- Konstruktoren:  
Wahr: Bool  
Falsch: Bool
- Destruktoren:  
 $\wedge, \vee : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$   
 $\neg : \text{Bool} \rightarrow \text{Bool}$

#### Axiome

$x \wedge \text{Wahr} = x$   
 $x \wedge \text{Falsch} = \text{Falsch}$   
 $x \vee \text{Wahr} = \text{Wahr}$   
 $x \vee \text{Falsch} = x$   
 $\neg \text{Wahr} = \text{Falsch}$   
 $\neg x = \text{Wahr}$   
 (von oben nach unten die erste passende Regel anwenden)

### III.15.2. Schlange (queue, fifo)

#### Signaturen

- Konstruktoren:  
 $\perp : \text{Schlange } a$   
Einf:  $a \times \text{Schlange } a \rightarrow \text{Schlange } a$

- Destruktoren:  
kopf: Schlange  $a \rightarrow a$   
schwanz: Schlange  $a \rightarrow$  Schlange  $a$
- Verhalten:  
länge: Schlange  $a \rightarrow \text{Int}$

**Axiome**

Seien  $S$ : Schlange  $a$ ,  $x$ :  $a$   
 $\text{kopf}(\text{Einf}(x, \perp)) = x$   
 $\text{kopf}(\text{Einf}(x, S)) = \text{kopf}(S)$   
 $\text{schwanz}(\text{Einf}(x, \perp)) = \perp$   
 $\text{schwanz}(\text{Einf}(x, S)) = \text{Einf}(x, \text{schwanz}(S))$   
 $\text{länge}(\perp) = 0$   
 $\text{länge}(\text{Einf}(x, S)) = 1 + \text{länge}(S)$

Eine konkrete Implementierung muss natürlich nicht  $\Theta(n)$  für Kopf haben.

Bewusst ausgelassen wurde  $\text{schwanz}(\perp)$  da die Behandlung dieses Falles den Umfang der Axiome erhöhen würde da, würde man einen Fehler einführen, dieser sich durch alle Axiome durchschlängeln müsste. (Was 6 zusätzliche Axiome bedeuten würde)

**III.15.3. First In Last Out – Keller, Stack****Signaturen**

Stack  $a$

- Konstruktoren:  
 $\perp$ : Keller  $a$   
push:  $(a \times \text{Keller } a) \rightarrow \text{Keller } a$
- Destruktoren:  
top: Keller  $a \rightarrow a$   
pop: Keller  $a \rightarrow \text{Keller } a$
- Verhalten:  
laenge: Keller  $a \rightarrow \text{Int}$

**Axiome**

Seien  $K$ :Keller  $a$ ,  $x$ : $a$   
 $\text{top}(\text{push}(x, K)) = x$   
 $\text{pop}(\text{push}(x, K)) = K$   
 $\text{laenge}(\perp) = 0$   
 $\text{laenge}(\text{push}(x, K)) = 1 + \text{laenge}(K)$

#### Normalformen

Kann man auf einen Ausdruck A kein Axiom mehr anwenden, so ist eine Normalform erreicht.

#### Beispiel

- $A = \text{push}(3, \text{pop}(\text{push}(2, \perp)))$   
 $= \text{push}(3, \perp)$  Normalform!!
- $\text{top}(\perp)$

#### III.15.4. Liste

##### Signaturen

list  $a$

- Konstruktoren:  
 $\perp : \text{list } a$   
 $\text{Cons}: a \times \text{list } a \rightarrow \text{list } a$
- Destruktoren:  
 $\text{head}: \text{list } a \rightarrow a$   
 $\text{tail}: \text{list } a \rightarrow \text{list } a$

##### Axiome

$\text{head}(\text{cons}(x, L)) = x$   
 $\text{tail}(\text{cons}(x, L)) = L$

#### III.15.5. Konkrete Implementierung

##### Verkettete Liste

$\rightarrow [\text{Element}][\text{Zeiger}] \rightarrow$

Die Liste ist dann ein Verweis (Zeiger) auf ein Listenelement. **FIXME:** skizze von listenelement

- type Listenelement  $a$   
Element:  $a$   
next:  $\uparrow \text{Listenelement } a$
- type Liste  $a$   
kopf:  $\uparrow \text{Listenelement } a$

Wie geht man mit einer leeren Liste um?

1. Möglichkeit:  
spezieller Speicherbereich (Nil, Null, NULL, ...)
2. Möglichkeit: Selbstverweis **FIXME:** skizze

(Wir entscheiden uns für erste Möglichkeit)

- `tail(L: Liste a) : Liste a`  
`if L.kopf = Nil then error "..."`  
`else`  
`L.kopf = L.Kopf.next`  
`return L`
- `head(L:Liste a):a`  
`if L.kopf = Nil then error "..."`  
`else`  
`return L.kopf.Element`
- `cons (x:a, l:Liste a):Liste a`  
`ne = new(Listenelement a)`  
`ne.element = x`  
`ne.next = l.Kopf`  
`L.Kopf=↑ne`  
`return l`

## III.16. Hash-Funktionen

### Problem:

**Gegeben:**  $D_1, \dots, D_n$  Datensätze,  $n \in \{1, \dots, N\}$  mit zugehörigen Schlüsseln:  $k_i, i \in \{1, \dots, n\}$   
 $\{\text{Schlüssel: Indexierbarer Datentyp, d.h. } k_i \text{ sind geordnet und } D_i \neq D_j \Rightarrow k_i \neq k_j\}$

### Gesucht:

Ist Datensatz  $D$  in  $D_1, \dots, D_n$  enthalten?

### Lösung:

- Speichere  $D_1, \dots, D_n$  in Liste
- Suchaufwand im worst case  $\Theta(n)$

**Vereinfachung:**  $N$  ist zwar gross aber nicht riesig groß...

- $A$ : Reihung (Array)  $[1, \dots, N]$
- Speichere in  $A$  an alle Plätze "leer"
- Füge alle  $D_1, \dots, D_n$  in Stelle  $k_i$  ein  $i = 1, \dots, n$

$\Rightarrow$  Suchaufwand  $\Theta(1)$

### Problem:

Meist ist  $N$  zu groß! z.b.  $2^{64}$  mögliche Schlüssel, aber „nur“ bla  $2^{20}$  Datensätze...

### Deshalb: Hash-Funktion

Suche eine Funktion  $h$  mit  $h : N \rightarrow \{1, \dots, m\}$  mit  $m > n$  aber  $m \ll N$

Dann initialisiere die Reihung  $A[1, \dots, m]$  wieder mit „leer“ und füge  $D_i$  an Stelle  $h(k_i)$  ein  
 $D_i = A[h(k_i)]$

### Problem:

$h$  kann nicht injektiv sein. d. h.:

Es gibt  $y$  und  $z$  mit  $h(x) = h(y)$  aber  $x \neq y \Rightarrow$  Kollision ( $h$ : Hashfunktion)

### Lösung:

verkettetes Hashen statt  $D_i$  speichere Liste  $D_i$

**FIXME:** skizze  $[ ][h(k_i)][ ][ ][ ][ ] \rightarrow [D_i, D_j]$

$h(k_i) = h(k_j), i \neq j$

$h$  muss gut gewählt werden, sonst verkettete Liste, also  $\Theta(n)!$

### 2. Lösung (Sondierung):

**Idee:** Im Fall einer Kollision suche deterministisch den nächsten freien Platz.

**Konkret:** Sondierungsfunktion  $S : \{0, \dots, m\} \rightarrow \mathbb{N}$

Ist der Platz  $h(k_i)$  schon belegt, dann teste für  $j = 1, 2, 3 \dots$

$$h'(k_i, j) = (h(k_i) + S(j)) \bmod m$$

Sobald eine Stelle frei ist, speichere den Datensatz  $D_i$  dort. Es ist dann  $h'(k_i, j) \neq h(k_i)$ , d.h. der Datensatz steht auch später nachvollziehbar an der falschen Stelle. Ist  $j = m$  und  $S$  ist injektiv, dann ist die Tabelle voll. Folgende Situationen können auftreten:

1. Beim Versuch  $D_i$  an der Position  $h(k_i)$  zu speichern, ist die Position mit einem Datensatz  $D_j$  belegt und  $h(k_i) = h(k_j)$ : *Kollision erster Ordnung*
2. Ist  $h(k_i) \neq h(k_j)$ : *Kollision zweiter Ordnung*

**Wie suche ich?** Tritt eine Kollision auf, d. h.  $h(k_i)$  ist belegt, suche weiter bei  $h'(k_i, j)$  für  $j = 1, 2, 3 \dots$  bis entweder

- $D_i$  gefunden,
- die aktuelle Position leer ist  $\Rightarrow D_i$  ist nicht in der Tabelle,
- $j = m \Rightarrow$  Tabelle ist voll

### Beispiele:

- Lineares Sondieren:  $h'(k_i, j) = (h(k_i) + j) \bmod m$
- Quadratisches Sondieren:  $h'(k_i, j) = (h(k_i) + c_1 j^2 + c_2 j) \bmod m$
- Doppeltes Hashen:  $h'(k_i, j) = (h(k_i) + j h_1(k_i)) \bmod m$ ,  $h_1$  eine weitere Hashfunktion

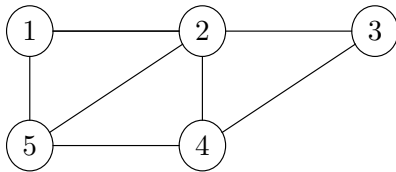
**Problem:** Wie lösche ich ein Element? Wenn einfach „leer“ in die tabelle eingetragen wird, dann werden Elemente unter Umständen nicht mehr gefunden. Die Suche kann fehlschlagen.

**Lösung:** Trage „gelöscht“ ein.

**Beispiel:**  $0, \dots, D_i, D_j, \dots, m, h(k_i) = h(k_j)$  (Lineares Sondieren) Nach Löschen von  $D_i$  kann  $D_j$  nicht mehr gefunden werden, falls nicht „gelöscht“ eingetragen wurde.

## III.17. Graphenalgorithmen und Datenstrukturen für Graphen

**Frage:** Wie speichere ich einen Graphen im Rechner?



Beispielgraph

### III.17.1. 1. Möglichkeit: Adjazenzliste

1 → [2,5]  
 2 → [1,5,4,3]  
 3 → [2,4]  
 4 → [5,2,3]  
 5 → [1,2,4]

### III.17.2. 2. Möglichkeit: Adjazenzmatrix

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

### III.17.3. Speicherbedarf:

- Worst-Case
  - Adjazenzliste:  $\Theta(n^2)$  Knoten
  - Adjazenzmatrix:  $\Theta(n^2)$  Bits
- Best-Case
  - Adjazenzliste:  $\Theta(n)$
  - Adjazenzmatrix:  $\Theta(n^2)$

#### III.17.4. Zugriff auf eine Kante:

- Worst-Case
  - Adjazenzliste:  $\Theta(n)$
  - Adjazenzmatrix:  $\Theta(1)$
- Best-Case
  - Adjazenzliste:  $\Theta(1)$
  - Adjazenzmatrix:  $\Theta(1)$

#### III.17.5. Einfache Graphenalgorithmen:

**Suche im Graphen:** z.B. gibt es einen Ausweg aus dem Labyrinth?  
Einfacher: Gibt es einen Weg von Knoten  $A$  nach Knoten  $B$ ?

#### III.17.6. Einfache Strategien:

##### Tiefensuche:

**Idee:** Im Knoten  $k$  mit Kanten  $E_1, \dots, E_j$ :  
Nimm Kante  $E_1$  und suche dort weiter.  
Stoße ich auf eine Sackgasse, gehe eins zurück und nimm dort  $E_2$  usw.

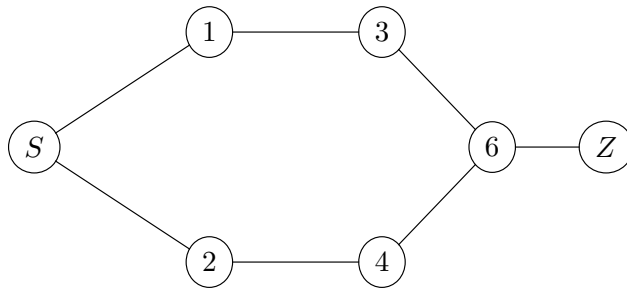
**Konkret:** Benutze einen Keller  $K$  (Knoten)  
Am Anfang enthält der Keller den Startknoten.  
Wiederhole:  
Ist  $K_j = \text{top}(K)$  das Ziel  $\Rightarrow$  fertig  
Sonst:  $\text{pop}(K)$ ,  $\text{push}(K_{j1}, \dots, K_{jl}, K)$ , wobei  $K_{j1}, \dots, K_{jl}$  die Folgeknoten von  $K_j$  sind.

##### Breitensuche:

**Idee:** Gehe erst einmal einen Schritt bei allen Nachfolgeknoten. Ist das Ziel dann noch nicht gefunden, gehe zwei Schritte usw.

**Konkret:** Benutze eine Schlange  $Schl$   
Am Anfang enthält  $Schl$  nur den Startknoten.  
Wiederhole:  
 $K_i = \text{Kopf}(Schl)$   
ist  $K_i$  das Ziel  $\Rightarrow$  Heureka!  
sonst  $\text{Einfügen}(K_{i1}, \dots, K_{ij}, \text{tail}(S))$ , wobei  $K_{i1}, \dots, K_{ij}$  die Folgeknoten von  $K_i$  sind.



**Beispiele:**

Beispielgraph

**Tiefensuche:** Zu Beginn:  $[S] \leftarrow \text{Keller} \Rightarrow \text{push}(1, 2, K) \Rightarrow K = [2, 1, S] \Rightarrow \text{push}(4, S, \text{pop}(K)) \Rightarrow K = [1, 4, S] \Rightarrow$  Der nächste Schritt beginnt wieder bei  $S$ . Ausweg: Zufällige Reihenfolge beim *push*

**Breitensuche:** Zu Beginn:  $[S] \leftarrow \text{Schlange}$   
 $\Rightarrow \text{einfügen}(1, 2, \text{Schl}) \Rightarrow \text{Schl} = [2, 1]$   
 $\Rightarrow \text{einfügen}(3, S, \text{tail}(\text{Schl})) \Rightarrow \text{Schl} = [S, 3, 2]$   
 $\Rightarrow \text{einfügen}(S, 4, \text{tail}(\text{Schl})) \Rightarrow \text{Schl} = [4, S, S, 3] \dots$

**III.18. Binäre Suchbäume**

30.05.2005

(nach Cormen) Suchbäume sind Datenstrukturen. Sie sind nützlich für Operationen auf dynamischen Mengen (Größe und Elemente sind nicht fest!) wie z.B. SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT oder DELETE.

**III.18.1. Definition/Einführung**

Ein binärer Suchbaum (B.S.) ist als binärer Baum (siehe Abbildung III.1) organisiert. Außer dem Attribut *Schlüssel* sind für jeden Knoten noch die Attribute *links* (linker Sohn), *rechts* (rechter Sohn) und *p* (Vater) gegeben. Wenn ein Sohn oder der Vater fehlt, erhält dieses Attribut den Wert *NIL*<sup>2</sup>. Der Wurzelknoten ist der einzige Knoten dessen Vater *NIL* ist.

Die Schlüssel werden immer so gespeichert dass die folgende Binäre-Suchbaum-Eigenschaft (B.S.E.) immer erhalten bleibt:

- Sei  $x$  Knoten in einem binären Suchbaum. Wenn  $y$  ein Knoten im linken Teilbaum von  $x$  ist, dann gilt

$$\text{schlüssel}[y] \leq \text{schlüssel}[x]$$

- Wenn  $y$  ein Knoten im rechten Teilbaum von  $x$  ist, dann gilt

$$\text{schlüssel}[x] < \text{schlüssel}[y]$$

**III.18.2. Traversierung**

Die B.S.E. erlaubt mittels einem einfachen rekursiven Algorithmus alle Schlüssel eines B.S. in sortierter Reihenfolge auszugeben.

---

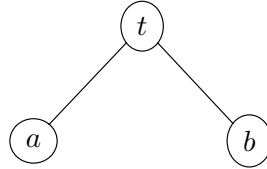
<sup>2</sup>Not In List

### III. Algorithmen

⇒ In-Order-Traversierung (es wird zuerst der linke Teilbaum, dann die Wurzel und danach der rechte Teilbaum ausgegeben  $(a, t, b)$ )

Es gibt auch noch zwei andere Arten der Traversierung (auf diese wird hier aber nicht näher eingegangen):

- Pre-Order-Traversierung (die Wurzel wird vor den beiden Teilbäumen ausgegeben  $(t, a, b)$ )
- Post-Order-Traversierung (die Wurzel wird nach den beiden Teilbäumen ausgegeben  $(a, b, t)$ )



INORDERTREEWALK( $x$ )

```
1  if  $x \neq NIL$ 
2    then INORDERTREEWALK(links[x])
3  print schlüssel[x]
4  INORDERTREEWALK(rechts[x])
```

**Beispiel** Die InOrder-Traversierung des Baumes in Abbildung III.1 ergibt: 2, 3, 5, 5, 7, 8

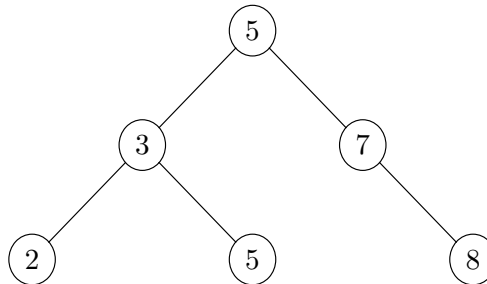


Abbildung III.1.: Beispiel InOrder-Traversierung

**Theorem** Wenn  $x$  die Wurzel eines Teilbaums mit  $n$  Knoten ist, dann benötigt INORDERTREEWALK die Zeit  $O(n)$ .

#### III.18.3. Suchen

Abfragen in einem B.S. machen sich ebenfalls die B.S.E. zu nutze um schneller zum Ziel zu gelangen.

TREESearch( $x, k$ )

```
1  if  $x = NIL$  oder  $k = schlüssel[x]$ 
2    then return  $x$ 
3  if  $k < schlüssel[x]$ 
4    then return TREESearch(links[x],  $k$ )
5  else return TREESearch(rechts[x],  $k$ )
```

**Beispiel** Die Abfragen nach 13 und 8 liefern für den Baum  $x$  aus Abbildung III.2 folgende Ergebnisse:

- $\text{TREESEARCH}(x, 13)$ :  $15 \xrightarrow{L} 6 \xrightarrow{R} 7 \xrightarrow{R} 13 \Rightarrow 13$
- $\text{TREESEARCH}(x, 8)$ :  $15 \xrightarrow{L} 6 \xrightarrow{R} 7 \xrightarrow{R} 13 \xrightarrow{L} \text{NIL} \Rightarrow \text{NIL}$

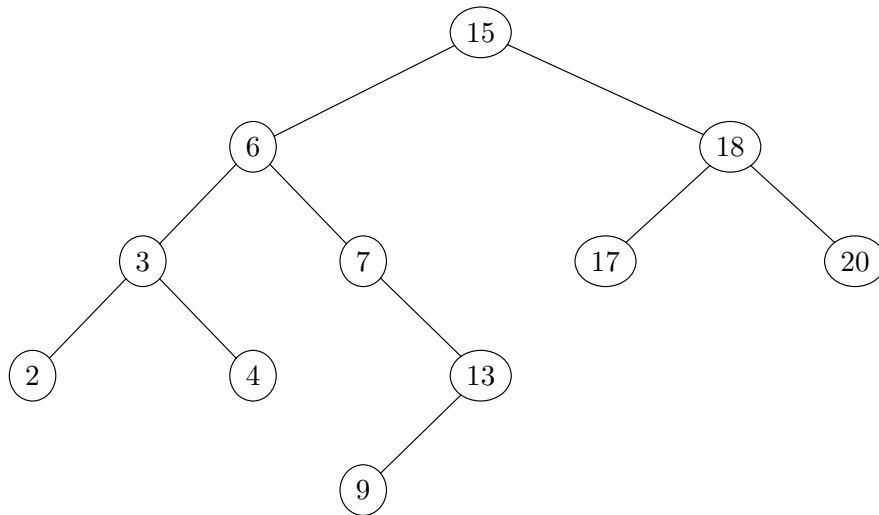


Abbildung III.2.: Beispiel-Baum  $x$

Das Finden des Elementes dauert dabei  $O(h)$ , wobei  $h$  für die Höhe des Baumes steht.

Die gleiche Prozedur kann auch iterativ geschrieben werden (d.h. in Form einer **while**-Schleife)

$\text{ITERATIVE\_TREESEARCH}(x, k)$

```

1  while  $x \neq \text{NIL}$  und  $k \neq \text{schlüssel}[x]$ 
2      do if  $k < \text{schlüssel}[x]$ 
3          then  $x \leftarrow \text{links}[x]$ 
4          else  $x \leftarrow \text{rechts}[x]$ 
5  return  $x$ 

```

#### III.18.4. Minimum und Maximum

$\text{TREEMINIMUM}(x)$

```

1  while  $\text{links}[x] \neq \text{NIL}$ 
2      do  $x \leftarrow \text{links}[x]$ 
3  return  $x$ 

```

$\text{TREEMAXIMUM}(x)$

```

1  while  $\text{rechts}[x] \neq \text{NIL}$ 
2      do  $x \leftarrow \text{rechts}[x]$ 
3  return  $x$ 

```

### III.18.5. Vorgänger und Nachfolger

TREESUCCESSOR( $x$ )

```

1  if rechts[ $x$ ]  $\neq$  NIL
2      then return TREEMINIMUM(rechts[ $x$ ])
3   $y \leftarrow p[x]$ 
4  while  $y \neq$  NIL und  $x = \textit{rechts}[y]$ 
5      do  $x \leftarrow y$ 
6       $y \leftarrow p[y]$ 
7  return  $y$ 

```

TREEPREDECESSOR( $x$ )

```

1  if links[ $x$ ]  $\neq$  NIL
2      then return TREEMAXIMUM(links[ $x$ ])
3   $y \leftarrow p[x]$ 
4  while  $y \neq$  NIL und  $x = \textit{links}[y]$ 
5      do  $x \leftarrow y$ 
6       $y \leftarrow p[y]$ 
7  return  $y$ 

```

### III.18.6. Theorem

Die Operationen SEARCH, MINIMUM, MAXIMUM, PREDECESSOR und SUCCESSOR für dynamische Mengen, können auf einem B.S. der Höhe  $h$  in der Zeit  $O(h)$  ausgeführt werden.

### III.18.7. Einfügen und Löschen

WICHTIG: Beim Einfügen und Löschen soll die B.S.E. beibehalten werden.

$\Rightarrow$  „Einfügen“: relativ unkompliziert

$\Rightarrow$  „Löschen“: etwas knifflig

**Einfügen** Mit der Prozedur TREEINSERT soll ein Wert  $v$  in den B.S.  $T$  eingefügt werden. Als Parameter bekommt sie den Knoten  $z$ , für den  $\textit{schlüssel}[z] = v$ ,  $\textit{links}[z] = \textit{NIL}$  und  $\textit{rechts}[z] = \textit{NIL}$  gilt.

TREEINSTERT( $T, z$ )

```

1   $y \leftarrow \textit{NIL}$ 
2   $x \leftarrow \textit{Wurzel}[T]$ 
3  while  $x \neq \textit{NIL}$ 
4      do  $y \leftarrow x$ 
5          if  $\textit{schlüssel}[z] < \textit{schlüssel}[x]$ 
6              then  $x \leftarrow \textit{links}[x]$ 
7              else  $x \leftarrow \textit{rechts}[x]$ 
8   $p[z] \leftarrow y$ 
9  if  $y = \textit{NIL}$ 
10     then  $\textit{Wurzel}[T] \leftarrow z$ 
        { $T$  war also leer}
11 if  $\textit{schlüssel}[z] < \textit{schlüssel}[y]$ 
12     then  $\textit{links}[y] \leftarrow z$ 
13     else  $\textit{rechts}[y] \leftarrow z$ 

```

Der Algorithmus beginnt an der Wurzel des Baum und arbeitet sich an einem Pfad nach unten. Der Zeiger  $x$  verfolgt den Pfad und  $y$  wird als Vater von  $x$  gehalten.

Nach der Initialisierung bewirkt die **while**-Schleife (Zeilen 3-7) dass diese beiden Zeiger im Baum abwärts laufen und dabei nach links oder rechts gehen (durch Vergleich der Schlüssel).

Dies geschieht so lange bis  $x$  gleich  $NIL$  gesetzt wird. Dieses  $NIL$  belegt den Platz an dem wir das Eingabeelement  $z$  einfügen wollen. Die Zeilen 8-13 setzen die Zeiger so, dass das Element  $x$  gerade an der Stelle eingeführt wird  $\Rightarrow$  Zeitaufwand  $O(h)$ .

01.06.05

**Löschen** Argument: Zeiger auf  $z$

Die Prozedur unterscheidet die drei (in 12.4) gezeigten Fälle.

- Falls  $z$  keine Kinder hat, modifizieren wir seinen Vater  $p[z]$  so, dass sein Kind durch  $NIL$  ersetzt wird.
- Falls  $z$  nur ein Kind hat, schneiden wir  $z$  aus, indem wir eine Verbindung zwischen seinem Kind und seinem Vater einfügen.
- Falls  $z$  zwei Kinder hat, nehmen wir  $z$ 's Nachfolger  $y$  heraus, der kein linkes Kind hat und ersetzen den Schlüssel und die Satellitendaten von  $z$  durch Schlüssel und Satellit von  $y$ .

### III.18.8. Theorem

Die Operationen INSERT und DELETE können so implementiert werden, dass sie auf einem binären Suchbaum der Höhe  $h$  in der Zeit  $O(h)$  laufen.

## III.19. Rot.Schwarz-Bäume

Binäre Suchbäume  $\Rightarrow$  SEARCH, SUCCESSOR, PREDECESSOR, MINIMUM, MAXIMUM, INSERT und DELETE  $\Rightarrow O(n)$

Verbesserung ist möglich  $\Rightarrow O(\lg n)$

### III.19.1. Eigenschaften von R.S.Bäumen

Ein R.S-Baum ist ein binärer Suchbaum, der ein zusätzliches Bit Speicherplatz zur Verfügung stellt. Diese Bit dient seiner Farbe: rot oder schwarz.

**FIXME:** Bild: R.S-Baum

**Attribute:** farbe, schlüssel, links, rechts und p

Wenn ein Kind oder der Vater eines Knotens nicht existiert, dann erhält das entsprechende Zeigerattribut den Wert  $NIL$ .

**Eigenschaften** Ein binärer Suchbaum ist ein R.S-Baum, falls er die folgenden Eigenschaften, die als R.S-Eigenschaften bezeichnet werden, erfüllt.

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt ( $NIL$ ) ist schwarz.
4. Wenn ein Knoten rot ist, dann sind seine beiden Kinder schwarz.

### III. Algorithmen

5. Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaums dieses Knotens enden, die gleiche Anzahl schwarzer Knoten.

#### Lemma

Ein R.S-Baum mit  $n$  inneren Knoten hat höchstens die Höhe  $2lg(n + 1)$

**Beweis:** Der Teilbaum zu einem beliebigen Knoten  $x$  hat mindestens  $2^{bh(x)} - 1$  innere Knoten.

IA Höhe von  $x = 0$ :

$$2^0 - 1 = 0$$

IV es ist wahr für  $n$ :

$$2^{bh(x)} - 1 \text{ innere Knoten}$$

IS nächster Schritt:

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$$

$\Rightarrow$  Operationen SEARCH, SUCCESSOR, PREDECESSOR, MINIMUM, MAXIMUM für dynamische Mengen auf R.S-Bäumen können in der Zeit  $O(lgn)$  implementiert werden, da sie auf einem Suchbaum der Höhe  $h$  in der Zeit  $O(h)$  laufen.

#### III.19.2. Rotationen

Um die Zeigerstruktur zu verändern, benutzen wir eine Rotation.

06.06.2005 **FIXME:** Beispiel

#### III.19.3. Einfuegen

“Introduction to Algorithms, Chapter 13.3, Second Edition”

- Rot-Schwarz-Baum mit  $n$  Knoten  $\Rightarrow O(lgn)$
- eine leicht modifizierte Version der Tree-Insertion
- Anschliessend färben wir den Knoten rot. Um sicherzustellen, dass die Rot-Schwarz-Eigenschaften erhalten bleiben rufen wir  $RB_{InsertFixup}(T, z)$  um die Knoten neu zu färben.  $\Rightarrow$  um die Knoten neu zu färben Rotation auszuführen.

```

RB-INSERT( $T, z$ )
1   $y \leftarrow \text{nil}[T]$ 
2   $x \leftarrow \text{wurzel}[T]$ 
3  while  $x \neq \text{nil}[T]$ 
4      do  $y \leftarrow x$ 
5          if  $\text{schlüssel}[z] < \text{schlüssel}[x]$ 
6              then  $x \leftarrow \text{links}[x]$ 
7              else  $x \leftarrow \text{rechts}[x]$ 
8   $p[z] \leftarrow y$ 
9  if  $y = \text{nul}[T]$ 
10     then  $\text{wurzel}[T] \leftarrow z$ 
11 elseif  $\text{schlüssel}[z] < \text{schlüssel}[y]$ 
12     then  $\text{links}[y] \leftarrow z$ 
13     else  $\text{rechts}[y] \leftarrow z$ 
14  $\text{links}[z] \leftarrow \text{nil}[T]$ 
15  $\text{rechts}[z] \leftarrow \text{nil}[T]$ 
16  $\text{farbe}[z] \leftarrow \text{ROT}$ 
17 RB-INSERT-FIXUP( $T, z$ )

```

Es gibt vier Unterschiede zwischen TREE-INSERT und RB-INSERT

1. alle Instanzen von NIL in TREE-INSERT werden durch  $\text{nil}[T]$  ersetzt
2. wir setzen  $\text{links}[z]$  und  $\text{rechts}[z]$  (Zeilen 14,15 RB-INSERT) auf  $\text{nil}[T]$  um die korrekte Baumstruktur aufrecht zu erhalten.
3. wir färben  $z$  rot (in Zeile 16 von RB-INSERT)
4. wegen einer mögliche Verletzung der Rot-Schwarz-Eigenschaft durch Schritt 3 rufen wir RB-INSERT-FIXUP( $T, z$ ) auf, um die RS-Eigenschaften wieder herzustellen.

```

RB-INSERT-FIXUP( $T, z$ )
1  while  $\text{farbe}[p[z]] = \text{ROT}$ 
2      do if  $p[z] = \text{links}[p[p[z]]]$ 
3          then  $y \leftarrow \text{rechts}[p[p[z]]]$ 
4              if  $\text{farbe}[y] = \text{ROT}$ 
5                  then  $\text{farbe}[p[z]] \leftarrow \text{SCHWARZ}$  ▷ FALL1
6                       $\text{farbe}[y] \leftarrow \text{SCHWARZ}$  ▷ FALL1
7                       $\text{farbe}[p[p[z]]] \leftarrow \text{ROT}$  ▷ FALL1
8                       $z \leftarrow p[p[z]]$  ▷ FALL1
9                  elseif  $z = \text{recht}[p[z]]$ 
10                     then  $z \leftarrow p[z]$  ▷ FALL2
11                     LEFT-ROTATE( $T, z$ ) ▷ FALL2
12                      $\text{farbe}[p[z]] \leftarrow \text{SCHWARZ}$  ▷ FALL3
13                      $\text{farbe}[p[p[z]]] \leftarrow \text{ROT}$  ▷ FALL3
14                     RIGHT-ROTATE( $T, p[p[z]]$ ) ▷ FALL3
15     else (wie then-Zweig mit “rechts” und “links” vertauscht)
16  $\text{farbe}[\text{wurzel}[T]] \leftarrow \text{SCHWARZ}$ 

```

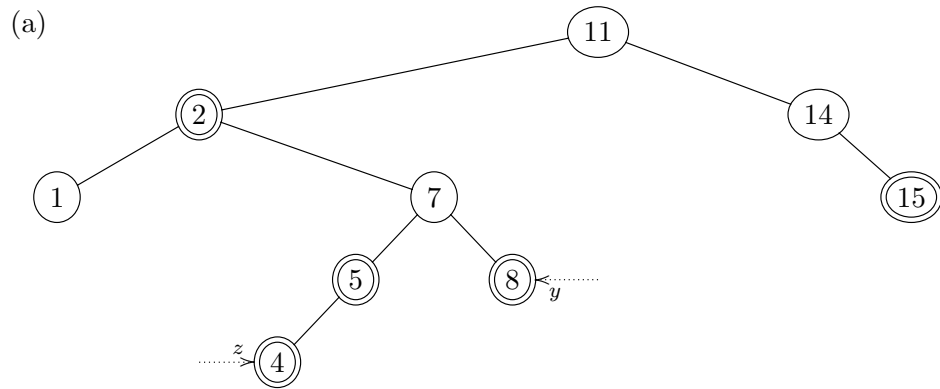


Abbildung III.3.: 13.4 (a)

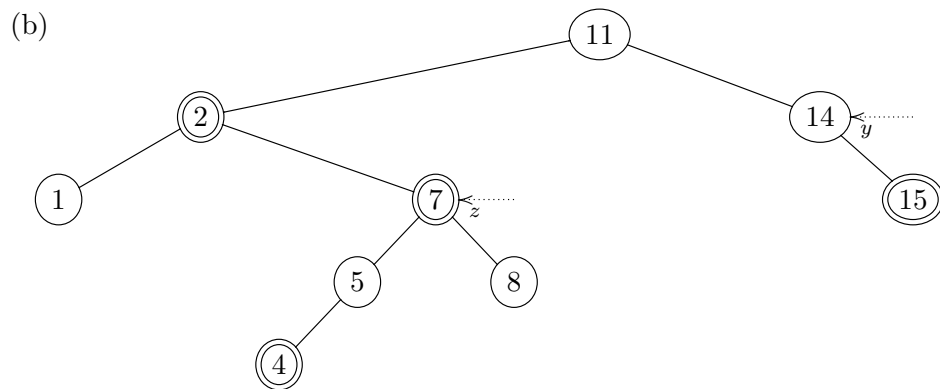


Abbildung III.4.: 13.4 (b)

- (a) da  $z$  und sein Vater  $p[z]$  beide rot sind erfolgt eine Verletzung der Eigenschaft 4. Da der Onkel  $y$  von  $Z$  rot ist, kann Fall 1 des Codes angewendet werden. Die Knoten werden neu gefärbt und der Zeiger  $z$  wird im Baum nach oben bewegt, wodurch der in (b) gezeigte Baum entsteht.
- (b) wieder sind  $z$  sein Vater  $p[z]$  beide rot, aber  $z$ 's Onkel  $y$  ist schwarz. Da  $z$  das rechte Kind von  $p[z]$  ist kann Fall 2 angewendet werden. Eine LinksRotation wird durchgeführt und der dadurch entstandene Baum wird in (c) gezeigt.
- (c) Nun ist  $z$  das linke Kind seines Vaters und Fall 3 kann angewendet werden.
- (d) Eine Rechtsrotation führt zu dem Baum gezeigt in (d), welches ein korrekter Baum ist.

Welche rot-schwarz Eigenschaften koennen durch den Aufruf von RB-INSERT-FIXUP verletzt sein?

Eigenschaft 1 und 3 gelten mit Sicherheit weiterhin, denn beide Kinder des neu eingefärbten roten Knotens sind der Waechter  $nil[T]$ . Eigenschaft 5 ist erfuehlt, weil der Knoten  $z$  den (schwarzen) Waechter ersetzt und Knoten  $z$  rot ist mit Waechter-Kindern.

Also koennen nur die Eigenschaften 2 (Wurzel ist schwarz  $\rightarrow$  falls  $z$  die Wurzel ist) ) und 4 (keine roten Kinder fuer rot Vater  $\rightarrow$  falls der Vater von  $z$  rot ist) verletzt sein. Eigenschaft 2 ist Verletzt, wenn  $z$  die Wurzel ist, Eigenschaft 4 wenn  $z$ 's Vater rot ist.

Die while-Schleife (Zeilen 1 - 15) enthält die folgende dreiteilige Schleifeninvariante:  
Zu Beginn jeder Iteration der Schleife gilt:



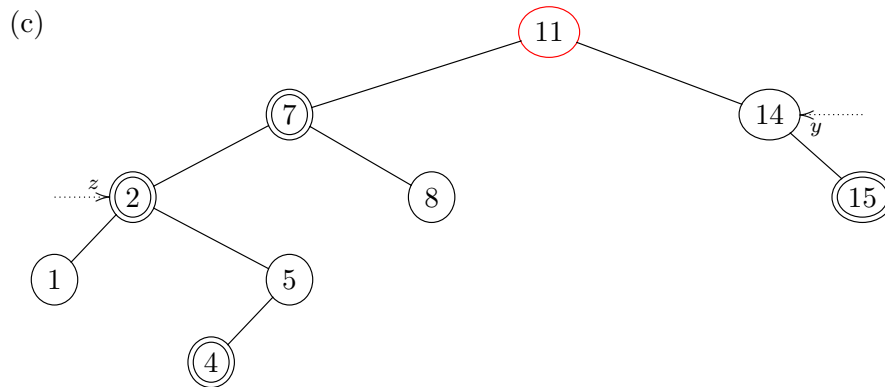


Abbildung III.5.: 13.4 (c)

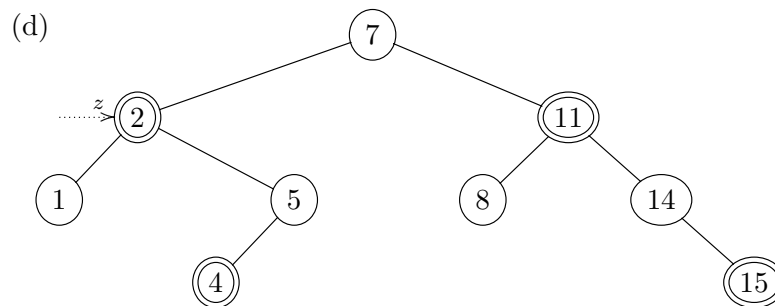


Abbildung III.6.: 13.4 (d)

- Knoten  $z$  ist rot.
- falls  $p[z]$  die wurzel ist, dann ist  $p[z]$  schwarz.
- falls es eine Verletzung der rot-schwarz Eigenschaften gibt, so gibt es hoechstens eine solche Verletzung. Entweder von Eig. 2 oder von Eig. 4. Wenn Eig. 2 verletzt wird, dann weil  $z$  die Wurzel ist und weil  $z$  rot ist. Eine Verletzung der Eigenschaft 4 tritt auf, wenn  $z$  und  $p[z]$  beide rot sind.

**Initialisierung:**

rot-schwarz-Baum ohne Verletzungen und roter Knoten  $z$  hinzugefuegt. Wir zeigen, dass zu dem Zeitpunkt zu dem RB-INSERT-FIXUP angerufen wird, jeder Teil der Schleifeninvariante gilt:

- Wenn RB-INSERT-FIXUP aufgerufen wird ist  $z$  der rote Knoten der hinzugefuegt wurde.
- wenn  $p[z]$  die Wurzel ist war  $p[z]$  anfangs schwarz und seine Farbe hat sich vor dem Aufruf von RB-INSERT-FIXUP nicht geaendert.
- Wir haben schon gesehen, dass die Eigenschaften 1,3 und 5 gelten.  
Wenn es eine Verletzung der Eig. 2 gibt, dann muss die rote Wurzel der neu hinzugefuegte Knoten  $z$  sein, welches der einzige innere Knoten des Baumes ist. Da der Vater und beide Kinder von  $z$  der (schwarze) Waechter sind gibt es nicht auch noch eine Verletzung der Eigenschaft 4.  
 $\Rightarrow$  die Verletzung der Eigenschaft 2 ist die einzige Verletzung der rot-schwarz Eigenschaften.

Wenn es eine Verletzung der Eigenschaft 4 gibt, dann weil die Kinder des Knoten der schwarze Waechter sind und der Baum vor dem hinzufügen von  $z$ , keine andere Verletzungen hatte. Die Verletzung muss daher ruehren, dass  $z$  und  $p[z]$  rot sind. Weiterhin gibt es keine Verletzungen der rot-schwarz Eigenschaften.

#### Terminierung:

keine moegliche Verletzung der 4. (weil  $p[z]$  schwarz ist )  
Die einzige Verletzung ist fuer 2.  
aber zeilen 16  $\Rightarrow$  alles ist gut.

#### Fortsetzung:

Es existieren 6 Faelle die zu beachten sind. 3 davon sind zu den anderen symmetrisch. (links rechtsch)

je nachdem, ob  $z$ 's Vater  $p[z]$  ein linkes oder rechtes Kind von  $z$ 's Grossvater  $p[p[z]]$  ist. Fall 1 unterscheidet sich von Faellen 2 und 3 durch die Farbe, die der Bruder von  $z$ 's Vater (Onkel) hat. Zeile 3 sorgt dafuer, dass  $y$  auf  $z$ 's Onkel rechts $[p[p[z]]]$  zeigt. in zeile 4 wird die Farbe getestet. Falls  $y$  rot ist wird Fall 1 ausgefuehrt. Anderenfalls sind die Faelle 2 und 3 auszufuehren. In allen drei Faellen ist der Grossvater  $p[p[z]]$  von  $z$  schwarz da  $p[z]$  rot ist. und eigenschaft 4 nur zwischen  $z$  und  $p[z]$  verletzt ist.

**FIXME:** Beispiel page 285 fig 13.5 Fall 1:  $z$ 's Onkel ist rot. Fall 2:  $z$  ist ein rechtes Kind Fall 3:  $z$ 's Onkel ist schwarz und  $z$  ist ein linkes Kind.

**FIXME:** 3. bild (Falle 2 und 3 der RB-Insert) page 286 fig 13.6

## III.20. Dynamisches Programmieren

### III.20.1. Optimierungsproblem:

Zu einem Problem gibt es viele Lösungen, gesucht ist eine optimale (maximal, minimal) Lösung.

**Beispiel:** Labyrinth

Es gibt viele mögliche Wege zum Ausgang, gesucht ist der kürzeste.

**Lösung:** Vollständige Suche (Tiefensuche, Breitensuche).

Oft kann man Optimierungsprobleme mit der Methode der dynamischen Programmierung effizienter lösen. Entwicklung eines dynamischen Programms in vier Schritten (nach Cormen):

1. Charakterisiere die Struktur einer optimalen Lösung.
2. Definiere den Wert einer optimalen Lösung rekursiv.
3. Berechne den Wert „bottom-up“.
4. Verwendung von Zwischenergebnissen.

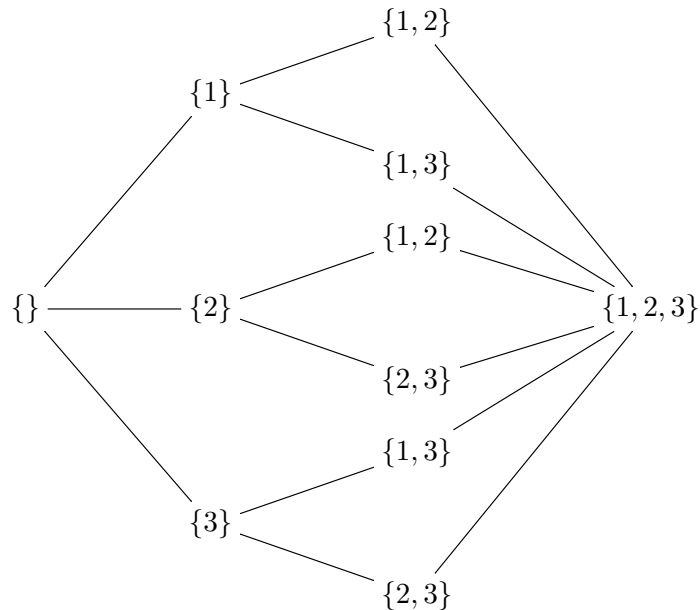
**III.20.2. Beispiel:**

Gegeben: Matrizen  $A_1, \dots, A_n$  passend

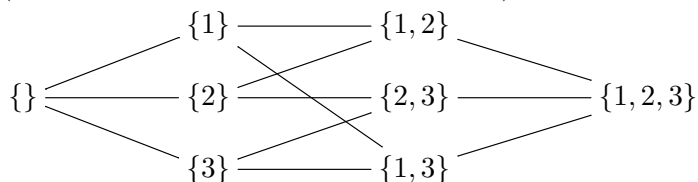
Gesucht:  $A_{1..n} := A_1 * \dots * A_n$  Optimierung: Möglichst wenig skalare Multiplikationen Beispiel:  $A_1$  mit Format  $10 \times 100$ ,  $A_2$  mit Format  $100 \times 5$  und  $A_3$  mit Format  $5 \times 50$ .  $(A_1 * A_2) * A_3$  benötigt 7500 Multiplikationen  $A_1 * (A_2 * A_3)$  benötigt 75000 Multiplikationen

**Aufstellen eines Baumes:**

Idee: Nummeriere die Operatoren, bilde Mengen, z.B. bedeutet  $\{i_1, i_2\}$ , dass die Operationen  $i_1$  und  $i_2$  ausgeführt werden. Beispiel:  $A_1 \dots A_4$



Dieser Baum wird in einen sog. *Trellis* (*Tree-like-structure*) überführt:  
(trellis: englisch fuer Gitter, Rankgitter)



Es gibt hier zwei Möglichkeiten, um z.B. zum Knoten  $\{1,2\}$  zu kommen:

1. Mit Kosten von  $\{1\}$
2. Mit Kosten von  $\{2\}$

Wähle die bessere!

**Situation FIXME:** Hier war ein Bild, aber ich kanns nicht entziffern!

Für die optimale Lösung ist nur der Pfad mit den geringsten Kosten interessant. Die anderen Pfade werden verworfen. Eine optimale Lösung muss einen optimalen Anfang haben.

**Algorithmus:**

Es ist nicht nötig, den Trellis aufzustellen. Es genügt jeweils eine Ebene, falls zu jedem Knoten alle Vorgänger und alle Nachfolger berechnet werden können.

**Konkret:**

- Vorgänger von  $\{i_1, \dots, i_k\}$   
Alle Teilmengen mit 1 Element weniger
- Nachfolger von  $\{i_1, \dots, i_k\}$   
Alle Teilmengen  $N$  von  $\{1, \dots, n\}$  mit  $i_1, \dots, i_k \in N$  und  $|N| = k + 1$

**III.20.3. Beispiel 2:**

Fehlerkorrigierende Codes:

**Begriffe:**

- *Informationswort*  
 $(i_0, \dots, i_k) \in \mathbb{F}_2^k$
- *Darstellung als Polynom*  
 $i(x) = \sum_{j=0}^k i_j x^j$
- *Generatorpolynom*  
 $g(x) = \sum_{j=0}^{n-k} g_j x^j$
- *Codewort als Polynom*  
 $c(x) = i(x)g(x) := \sum_{j=0}^n c_j x^j$   
 $c = (c_1, \dots, c_n) \in \mathbb{F}_2^n$

**Kanalmodell:**

Codewort  $c \in \mathbb{F}_2^n \rightarrow$  modulierte Codewort  $m \in \mathbb{R}^n \rightarrow$  Empfangswort  $y \in \mathbb{R}^n$  mit  $y = e + m$ , Fehler  $e \in \mathbb{R}^n$

**Fragen:** Wie komme ich zurück zu 0, 1? Welches Codewort passt am besten zu  $y$ ?

$c \rightarrow m, 0 \mapsto 1 \in \mathbb{R}, 1 \mapsto -1 \in \mathbb{R}$

Wie suche ich ein passendes Codewort?

**Beispiel:**

$i = (0, 1), i(x) = x, g(x) = x + 1, c(x) = g(x)i(x) = x^2 + x \Rightarrow c = (0, 1, 1), m = (1, -1, -1)$

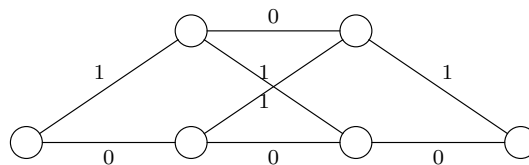
$00 \mapsto (0, 0, 0)$

$01 \mapsto (0, 1, 1)$

$10 \mapsto (1, 1, 0)$

$11 \mapsto (1, 0, 1)$

Trellis:



**Aufwandsanalyse:**

Paritätscode:  $k + 1$ -Bits

Es gibt  $2^{k+1}$  Codeworte,  $n = k + 1$ . Der Aufwand für das Durchsuchen ist im Trellis  $(n + 2)2$ .

**FIXME:** Hat jemand das mit dem Reed Solomon Code verstanden??

13.06.05

**III.21. Vorbestimmung und Vorberechnung**

(vergleiche: AlgotechSkript Calmet 99/00, Kapitel 5)

**III.21.1. Vorbestimmung****Einfuehrung**

Sei  $I$ :Menge von Instanzen eines gegebenen Problems. Angenommen  $i \in I$  kann in zwei Komponenten  $j \in J$  und  $k \in K$  aufgeteilt werden, d.h.:  $I \subseteq J \times K$ .

Ein "Vorbestimmungsalgorithmus" (fuer dieses Problem) ist ein Algorithmus  $A$  der irgendein Element  $j \in J$  als Eingabe akzeptiert und einen neuen Algorithmus  $B_j$  als Ausgabe liefert.

$B_j$  genügt also folgender Bedingung:

$k \in K$  und  $\langle j, k \rangle \in I \Rightarrow$  Anwendung von  $B_j$  auf  $k$  liefert die Lösung zu  $\langle j, k \rangle$  (Originalproblem)

**Beispiel:**

$J$ : Menge von Grammatiken für eine Familie von Programmiersprachen. (z.B.: Grammatiken in Backus-Naur-Form fuer Sprachen wie Algol, Pascal, Simula, ...)

$K$ : Menge von Programmen

Allgemeines Problem: Ist ein gegebenes Programm bezüglich einer Sprache syntaktisch korrekt? In diesem Fall ist  $I$  die Menge von Instanzen des Typs

"Ist  $k \in K$  ein gültiges Programm in der Sprache, die durch die Grammatik  $j \in J$  definiert ist?"

Ein Möglicher Vorbestimmungsalgorithmus ist ein Compiler-Generator:

Angewendet auf die Grammatik  $j \in J$  generiert er einen Compiler  $B_j$ . Danach um festzustellen, ob  $k \in K$  ein Programm in der Sprache  $J$  ist, wenden wir einfach den Compiler  $B_j$  auf  $k$  an.

Es seien

$$\begin{aligned} a(j) &= \text{Zeit, um } B_j \text{ zu produzieren.} \\ b_j(k) &= \text{Zeit, um } B_j \text{ auf } k \text{ anzuwenden.} \\ t(j, k) &= \text{Zeit, um } \langle j, k \rangle \text{ direkt zu loesen.} \end{aligned}$$

Normalerweise gilt:

$$b_j(k) \leq t(j, k) \leq a(j) + b_j(k).$$

Vorbestimmung ist Zeitverschwendung wenn

$$b_j(k) > t(j, k).$$

Vorbestimmung kann in zwei Situationen sinnvoll sein:

### III. Algorithmen

- (a) Man muss  $i \in I$  sehr schnell lösen. (schnelle Antwort in Echtzeitanwendungen, wo es manchmal unpraktisch alle  $\#I$  Lösungen zu den relevanten Instanzen im voraus zu berechnen und zu speichern.  $\#J$  Algorithmen vorzubestimmen koennte jedoch von Vorteil sein. Z.B.:
- (i) Einen laufenden Kern-Reaktor zu stoppen.
  - (ii) Die von einem Studenten verbrauchte Zeit zur Vorbereitung einer Prüfung.
- (b) Wir müssen eine Reihe von Instanzen  $\langle j, k_1 \rangle, \langle j, k_2 \rangle, \dots, \langle j, k_n \rangle$  mit gleichem  $j$  lösen.  
Ohne Vorbestimmung:  $t_1 = \sum_{i=1}^n t(j, k_i)$ .  
Mit Vorbestimmung:  $t_2 = a(j) + \sum_{i=1}^n b_j(k_i)$   
Wenn  $n$  gross genug ist, dann ist  $t_2$  oft viel kleiner als  $t_1$ .

#### Beispiel:

$J$ : Menge von Schlüsselwörtermengen

$J = \{\{if, then, else, endif\}, \{for, to, by\}, \dots\}$

$K$ : Menge von Schlüsselwörtern

$K = \{begin, function, \dots\}$  Wir müssen eine grosse Anzahl von Instanzen des folgenden Typs lösen:

“Gehört das Schlüsselwort  $k \in K$  der Menge  $j \in J$  an?”

Wenn wir jede Instanz direkt lösen, kommen wir auf  $t(j, k) \in \Theta(n_j)$ , wobei  $n_j$  die Anzahl der Elemente in der Menge  $j$  ist.

Idee: zunächst  $j$  Sortieren ( $\Theta(n_j \log(n_j))$ ) (Vorbestimmung) dann koennen binären Suchalgorithmus benutzen.  $\Rightarrow$

$$\begin{aligned} a(j) &\in \Theta(n_j \log(n_j))(\text{Sortieren}) \\ b_j(k) &\in \Theta(\log(n_j))(\text{Suchen}) \end{aligned}$$

Muss oft in der gleichen Menge(Instanz) gesucht werden, so ist das vorherige Sortieren gar nicht so dumm.

#### Vorgänger in einem Wurzelbaum

$J$ : Menge aller Bäume

$K$ : Menge von Knotenpaaren (Kanten)  $\langle v, w \rangle$

Für ein gegebenes Paar  $k = \langle v, w \rangle$  und einm gegebenes Baum  $j$  möchten wir feststellen, ob Knoten  $v$  der Vorgänger von  $w$  im Baum  $j$  ist. (Per def.: Jeder Knoten ist sein eigener Vorgänger)

Direkte Lösung dieser Instanz (schlimmster Fall):  $\Omega(n)$ . ( $j$  besteht aus  $n$  Knoten)

Es ist immer möglich, Vorbestimmung für  $j$  in Zeit  $\Theta(n)$  durchzuführen, so dass wir eine bestimmte Instanz des Problems bezüglich  $j$  in  $\Theta(1)$  lösen können.

**FIXME:** Beispielbaum mit 13 Knoten mit preorder(links) und postorder(rechts) nummerierung...

Vorbestimmung:

Pre-Order-Traversierung und Post-Order-Traversierung. Dabei nummerieren wir die Knoten wie sie durchlaufen werden. Für Knoten  $v$  sei  $prenum[v]$  und  $postnum[v]$  die ihm zugeordneten Nummern.

- Bei der Pre-Order-Traversierung nummerieren wir zunächst einen Knoten und dann die Teilbäume von links nach rechts.
- Bei der Post-Order-Traversierung nummerieren wir die Teilbäume eines Knotens von links nach rechts und danach den Knoten.

Es gilt:

$$prenum[v] \leq prenum[w] \Leftrightarrow v \text{ ist ein Vorgänger von } w \text{ **oder** } v \text{ ist links von } w$$

$$postnum[v] \geq postnum[w] \Leftrightarrow v \text{ ist ein Vorgänger von } w \text{ **oder** } v \text{ ist rechts von } w$$

Und somit:

$$prenum[v] \leq prenum[w] \text{ **und** } postnum[v] \geq postnum[w] \Leftrightarrow v \text{ ist ein Vorgänger von } w.$$

Wurde der Baum so in  $\Theta(n)$  vorbereitet, so kann die Bedingung in  $\Theta(1)$  geprüft werden.

### Wiederholte Auswertung eines Polynoms

$J$ : Menge der Polynome (in einer Variablen  $x$ )

$K$ : Wertemenge von  $x$

Problem: Auswertung

Bedingungen:

- 1) ganzzahliger Koeffizient
- 2) normierte Polynome (führender Koeffizient = 1; "monic")
- 3) Grad  $n = 2^k - 1, k \in \mathbb{N}$

Wir messen die Effizienz der verschiedenen Methoden an der Anzahl von Multiplikationen (Anzahl der Additionen nur zweitrangig)

**Beispiel:**  $p(x) = x^7 - 5x^6 + 4x^5 - 13x^4 + 3x^3 - 10x^2 + 5x - 17$

Naive Methode: Berechne zunächst die Reihe von Werten  $x^2, x^3, \dots, x^7$ , damit  $5x, -10x^2, \dots$ , und dann  $p(x)$ .

$\Rightarrow$  12 Multiplikationen und 7 Additionen.

leichte Verbesserung:

$$p(x) = ((((((x - 5)x + 4)x - 13)x + 3)x - 10)x + 5)x - 17$$

$\Rightarrow$  6 Multiplikationen und 7 Additionen.

Noch besser:

$$p(x) = (x^4 + 2)[(x^2 + 3)(x - 5) + (x + 2)] + [(x^2 - 4)x + (x + 9)]$$

### III. Algorithmen

⇒ 5 Multiplikationen (2 zur Berechnung von  $x^2$  und  $x^4$ ) und 9 Additionen.

Wie:

$p(x)$  ist nach Voraussetzung ein normiertes Polynom vom Grad  $n = 2^k - 1$ . Also koennen wir  $p(x)$  so ausdrücken:

$$p(x) = (x^{\frac{n+1}{2}} + a)q(x) + r(x)$$

wobei  $a$ : konst.;  $q(x)$  und  $r(x)$  normierte Polynome vom Grad  $2^{k-1} - 1$

Nun wird  $p(x)$  als Polynom der Form  $(x^i + c)$ , wobei  $i$  eine 2-er Potenz ist, ausgedrückt:

$$p(x) = (x^4 + a)(x^3 + q_2 \cdot x^2 + q_1 \cdot x + q_0) + (x^3 + r_2 \cdot x^2 + r_1 \cdot x + r_0)$$

Koeffizientenvergleich:

$$q_2 = -5, q_1 = 4, q_0 = -13, a = 2, r_2 = 0, r_1 = -3, r_0 = 9$$

$$\Rightarrow p(x) = (x^4 + 2)(x^3 - 5x^2 + 4x - 13) + (x^3 - 3x + 9)$$

Ähnlich:  $x^3 - 5x^2 + 4 \cdot x - 13 = (x^2 + 3)(x - 5) + (x - 2)$ . Daraus erhält man das im Beispiel gegebene Ergebnis. (Diese Methode wurde von Belaga (im "Problemi Kibernetiki", vol 5, pp 7 - 15, 1961) vorgeschlagen.

#### III.21.2. Vorberechnung für Zeichenreihe-Suchprobleme

**Problem:**

$S = s_1 s_2 \dots s_n$  Zeichenreihe aus  $n$  Zeichen. (haystack, text)  $P = p_1 p_2 \dots p_m$  Zeichenreihe aus  $m$  Zeichen. (needle, pattern) "Ist  $P$  eine Teilzeichenreihe von  $S$ ?" und "Wo in  $S$  kommt  $P$  vor?"  
OBdA:  $n \geq m$

**Naiver Algorithmus:**

Liefert  $r$ , falls das erste Vorkommen von  $P$  in  $S$  in der Position  $r$  beginnt. ( $r$  ist die kleinste ganze Zahl, so dass  $s_{r+i-1} = p_i, i = 1, 2, \dots, m$  gilt) Und 0 falls  $P$  keine Teilzeichenreihe von  $S$  ist.

..

```
1  for  $i \leftarrow 0$  to  $n - m$ 
2      do
3           $ok \leftarrow \text{TRUE}$ 
4           $j \leftarrow 1$ 
5          while  $ok \ \& \ j \leq m$ 
6              do if  $p[j] \neq s[i + j]$ 
7                   $ok \leftarrow \text{FALSE}$ 
8              else  $j \leftarrow j + 1$ 
9  if  $ok$  return  $i + 1$ 
10 return 0
```

Jede Position in  $S$  wird geprüft. Worst-Case:  $\Omega(m(n - m)) \Rightarrow \Omega(nm)$ , falls  $n$  viel grösser als  $m$  ist.

Es ist eine bessere Ausführung möglich, indem man verschiedene Methoden anwendet.



**Signaturen**

Sei  $S = S_1, S_2, \dots, S_n$  in Teilzeichenreihen zerlegt und falls  $P$  in  $S$  vorkommt muss  $P$  vollständig in einer der Teilzeichenreihen  $S_l, 1 \leq l \leq n$  vorkommen. (z.b.: Zerlegung einer Textdatei in Zeilen) Grundidee: eine Boolesche Funktion  $T(P, S)$  benutzen. (sehr schnell berechnet)  $T(P, S_i) = false \Rightarrow P \notin S_i$ . Sonst:  $P \in S_i$  möglich. (dann naiver Algorithmus...)

Mit Signaturen findet man ein einfaches Verfahren zur Implementation eines solchen Algorithmus.

Angenommen:

- 1)  $P$  ist a,b,c,...,y,z,other ("other": nicht-alphabetische Zeichen).
- 2) 32 Bit Rechner.

Eine mögliche Definition einer Signatur:

**Definiton "Signatur":**

- (i) Definiere  $\text{val}(\text{"a"})=0, \text{val}(\text{"b"})=1, \dots, \text{val}(\text{"z"})=25, \text{val}(\text{"other"})=26$
- (ii) Falls  $c_1$  und  $c_2$  Zeichen sind, definiere:

$$B(c_1, c_2) = (27\text{val}(c_1) + \text{val}(c_2)) \bmod 32$$

- (iii) Definiere die Signaturen  $\text{sig}(C)$  einer Zeichenreihe  $C = c_1 c_2 \dots c_r$  als ein 32-Bit-Wort, wobei die Bits mit den Nummern  $B(c_1, c_2), B(c_2, c_3), \dots, B(c_{r-1}, c_r)$  auf 1 und sonst auf 0 gesetzt sind.

**Beispiel:**

$C = \text{"computers"}$

$$B(\text{"c"}, \text{"o"}) = (27 \times 2 + 14) \bmod 32 = 4$$

$$B(\text{"o"}, \text{"m"}) = (27 \times 14 + 12) \bmod 32 = 6$$

...

$$B(\text{"r"}, \text{"s"}) = (27 \times 17 + 18) \bmod 32 = 29$$

Wenn die Bit des Wortes von links nach rechts mit 0 bis 31 nummeriert werden, ist die Signatur dieser Zeichenreihe

$$\text{sig}(C) = 0000\ 1110\ 0100\ 0001\ 0001\ 0000\ 0000\ 0100$$

Nur 7 Bits sind auf 1 gesetzt, da  $B(\text{"e"}, \text{"r"}) = B(\text{"r"}, \text{"s"}) = 29$  ist.

Wir berechnen jedes  $\text{sig}(S_i)$ , sowie  $\text{sig}(P)$ . Wenn  $P \in S_i$  dann sind alle Bit die in der Signatur von  $P$  1 sind, auch in der Signatur  $S_i$  auf 1 gesetzt.

$$T(P, S_i) = [\text{sig}(P) \text{ and } \text{sig}(S_i) = \text{sig}(P)]$$

(and: bitweise Konjunktion von zwei ganzen Wörtern.)

Berechnung der Signaturen in  $O(n)$ . Für Muster  $P$  brauchen wir  $O(m)$ .  $T(P, S)$  geht fix.

## III.22. Vorberechnung für Zeichenreihen-Suchprobleme

20.06.2005

### III.22.1. Algorithmus von Knuth-Morris-Pratt

**Gegeben:**

- lange Zeichenkette  $S$  (Text), Zeichen  $1, \dots, n$
- kurze Zeichenkette  $P$  (Suchwort), Zeichen  $1, \dots, m$

**Gesucht:**

- Finde die erste Position  $i$ , so dass

$$S[i+j] = P[j], \quad j = 1, \dots, m$$

Also das erste Vorkommen von  $P$  in  $S$ .

#### Naive Methode

NAIVEALGORITHM( $S, P$ )

```

1   $n \leftarrow \text{length}(S)$ 
2   $m \leftarrow \text{length}(P)$ 
3   $i \leftarrow 0$ 
4   $j \leftarrow 1$ 
5  while  $j \leq m$  und  $i \leq n - m$ 
6      do if  $P[j] = S[i+j]$ 
7          then  $j \leftarrow j + 1$ 
8          else  $j \leftarrow 1$ 
9               $i \leftarrow i + 1$ 
        {  $j$  Zeichen stimmten überein }
10 if  $j > m$ 
11     then return  $i + 1$ 
        { Suchwort nicht gefunden }
12 return -1
```

Der Algorithmus vergleicht  $P$  also zeichenweise mit  $S$  und verschiebt  $P$  um eins nach hinten, sobald ein Fehler auftritt. Kann auf diese Weise  $P$  komplett in  $S$  gefunden werden bricht der Algorithmus mit Erfolg ab (Startposition  $i$  von  $P$  in  $S$ ,  $i \geq 1$ ). Ansonsten läuft er  $S$  komplett durch und endet mit einem Fehler ( $-1$ ).

$S$	=	A	B	R	A	K	A	D	A	B	R	A
$P$	=	A	K	A								
				A	K	A						
					A	K	A					
						A	K	A				

**Aufwand im Worst-Case** Das Wort  $P$  steht nicht in  $S$  und passt (fast) immer bis auf das letzte Zeichen  $\Rightarrow \Theta(n \cdot m)$ .

$\Rightarrow$  Versuche den Aufwand durch Vorberechnung zu reduzieren

$\Rightarrow$  KMP-Algorithmus

**Idee:** Optimierte das „Schieben“ und „Vergleichen“ indem man sich zuerst  $P$  genauer anschaut. Dazu wird eine Verschiebeliste ( $next[j]$ ) auf Basis von  $P$  aufgebaut, aus der abgelesen werden kann, ob im Falle der Ungleichheit ab einem bestimmten Zeichen in  $P$  (Zeile) anschließend eventuell um mehr als 1 verschoben werden kann (vgl. Zeile).

$j$	=	1	2	3	4	5	6	7	8	9	10						
$P[j]$	=	$a$	$b$	$c$	$a$	$b$	$c$	$a$	$c$	$a$	$b$						
			$a$	$b$	$c$	$a$	$b$	$c$	$a$	$c$	$a$	$b$					
				$a$	$b$	$c$	$a$	$b$	$c$	$a$	$c$	$a$	$b$				
					<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	$b$	$c$	$a$	$c$	$a$	$b$			
$next[j]$	=	0	1	1	0	1	1	0	5	0	1						

**Um wie viel wird nun verschoben?** Antwort  $j - next[j]$

**Aufwand:** Im schlimmsten Fall kommt das erste Zeichen von  $P$  in  $S$  nicht vor  $\Rightarrow$  es wird immer nur um 1 verschoben  $\Rightarrow \Theta(n)$ .

Für das Erstellen der Verschiebungstabelle fällt ein Zusatzaufwand an. Dieser kann aber vernachlässigt werden, da üblicherweise  $m \ll n$  ist.

### III.22.2. Algorithmus von Boyer-Moore

**Idee** Betrachte das Wort von hinten nach vorn,

DENN: wenn das aktuell betrachtete Zeichen (aus  $S$ ) in  $P$  nicht vorkommt kann ich gleich um  $m$  schieben.

T	H	I	S		I	S		A		D	E	L	I	C	A	T	E		T	O	P	I	C
C	A	T																					
			C	A	T																		
						C	A	T															
									C	A	T												
												C	A	T									
															C	A	T						

#### Aufwandsabschätzung

$$O\left(m + \frac{n}{m}\right) \text{ im best-case}$$

wenn das letzte Zeichen von  $P$  nicht in  $S$  vorkommt (falls  $P \notin S$ ).

$$O(n) \text{ im worst-case}$$

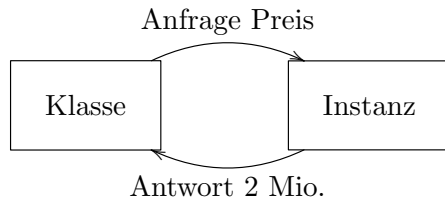
wenn das letzte Zeichen von  $P$  auf (fast) jedes Zeichen von  $S$  passt (falls  $P \notin S$ ).



## IV. Objektorientierte Programmierung

Ein Wert kann erst übergeben werden, wenn eine konkrete Instanz vorliegt.

**Wie kommt man an den Verkaufspreis?** Die Klasse sendet eine Botschaft an die konkrete Instanz.



**Was ist ein Objekt?** Ein Objekt (*Object*) ist ein Gegenstand des Interesses einer Beobachtung, Untersuchung oder Messung. Jedes Objekt hat eine eindeutige Objekt-Identität.

Der Zustand (*state*) eines Objektes wird durch seine Attribute bzw. Daten und Verbindung zu anderen Objekten bestimmt.

Das Verhalten (*behavior*) eines Objektes wird durch eine Menge von Operationen beschrieben.

Synonyme für den Begriff des Objekts sind Instanz oder Exemplar.

22.06.2005

### IV.1. Klassen

#### IV.1.1. Definition

Eine *Klasse* ist eine Gemeinsamkeit von Objekten mit den selben Eigenschaften (Attributen), dem selben Verhalten (Operationen bzw. Methoden) und den selben Beziehungen zu anderen Objekten (Vererbung).

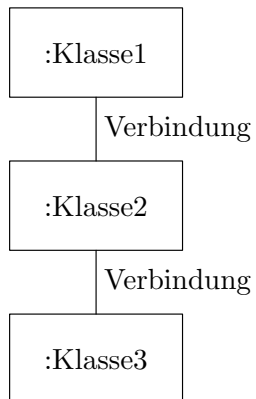
#### IV.1.2. Gleichheit und Identität von Objekten

Zwei Objekte sind *gleich*, wenn sie die gleichen Attributswerte besitzen aber unterschiedliche Objektidentitäten.

**Beispiel:** Zwei Firmen haben eine gemeinsame Tochterfirma.

**Analog für Gleichheit:** Zwei Firmen besitzen jeweils eine Tochterfirma gleichen Namens. Die Objekte sind somit gleich, aber nicht identisch.

## IV.2. Objektdiagramm



**Bemerkung:** Was die Verbindung bedeutet bleibt offen.

## IV.3. Verwendung von Klassen

→ Schablone

Ein Objekt kennt seine Klasse, aber die Klasse “weiß „nicht welche Objekt von ihr abgeleitet werden.