

# **Computerorientierte Mathematik**

**Lernwissen zur Klausur**

Stefan Hildebrand

24. September 2013

Wichtigste Quelle:

Prof. Dr. Martin Skutella, Dr. Britta Peis:

Vorlesungen der Computerorientierten Mathematik I und II

TU Berlin, WiSe 2012 - SoSe 2013

# Inhaltsverzeichnis

<b>1 Kontrollstrukturen</b>	<b>6</b>
1.1 Rekursion . . . . .	6
1.1.1 Idee . . . . .	6
1.1.2 Backtracking . . . . .	6
<b>2 Kodierung</b>	<b>7</b>
2.1 Eindeutige Decodierbarkeit . . . . .	7
2.2 Block- Code . . . . .	7
2.3 Präfix- Code . . . . .	7
2.4 Datenkompression mit Huffman- Algorithmus . . . . .	8
<b>3 Datenstrukturen</b>	<b>9</b>
3.1 Array . . . . .	9
3.2 Liste . . . . .	10
3.3 Stack . . . . .	10
3.4 Queue . . . . .	11
3.5 Graphen . . . . .	11
3.5.1 Weg . . . . .	11
3.5.2 Kantenbewertungen . . . . .	11
3.5.3 Länge eines Weges . . . . .	11
3.5.4 Kürzester Weg von Knoten i zu Knoten j . . . . .	11
3.5.5 Gerichtete (Di-) Graphen . . . . .	11
Zykel . . . . .	12
3.5.6 Datenstrukturen für Digraphen . . . . .	12
Adjazenzliste . . . . .	12
Adjazenzmatrix . . . . .	12
Entfernungsmatrix . . . . .	12
3.5.7 Bellman- Gleichungen . . . . .	12
3.6 Bäume . . . . .	12
3.6.1 voller binärer Baum: . . . . .	12
3.7 Suchbäume . . . . .	13
3.7.1 Suchen . . . . .	13
3.7.2 Einfügen . . . . .	13
3.7.3 Löschen . . . . .	14
3.7.4 Rotation . . . . .	15
3.7.5 AVL- Bäume . . . . .	16
Definition Balance . . . . .	16
Definition AVL- Baum . . . . .	16
Idee des Beweises . . . . .	16
Rotationslemma für AVL- Bäume . . . . .	18
Implementation der Basis- Operationen in AVL- Bäumen .	20
3.7.6 Optimale statische Suchbäume . . . . .	21
Ziel: . . . . .	21

Struktur von optimalen Suchbäumen . . . . .	22
Aufwand: . . . . .	25
3.8 Priority Queue . . . . .	26
3.9 Heap . . . . .	26
3.9.1 Definition: Array als Heap . . . . .	27
<b>4 Funktionen auf Datenstrukturen</b>	<b>28</b>
4.1 auf Graphen . . . . .	28
4.1.1 Bellman- Ford- Algorithmus . . . . .	28
4.1.2 Ermittlung negativer Zykel . . . . .	29
4.1.3 Floyd- Warshall- Algorithmus . . . . .	29
Pseudocode . . . . .	29
Tree- Matrix . . . . .	30
Adjazenzlisten . . . . .	30
4.1.4 Tiefensuche im Graphen . . . . .	30
4.1.5 Breitensuche . . . . .	31
4.2 Suchalgorithmen auf linearen Datenstrukturen . . . . .	32
4.2.1 Sequentielle Suche . . . . .	32
4.2.2 Binäre Suche . . . . .	32
4.3 Sortieralgorithmen . . . . .	32
4.3.1 Bubble Sort . . . . .	32
4.3.2 SelectionSort . . . . .	32
4.3.3 InsertionSort . . . . .	33
4.3.4 MergeSort . . . . .	33
4.3.5 QuickSort . . . . .	33
4.3.6 HeapSort . . . . .	34
Algorithmus zur Herstellung der Heap- Eigenschaft . . . . .	34
Der eigentliche Sortiervorgang . . . . .	34
4.3.7 Untere Komplexitätsschranken für das Sortieren . . . . .	34
4.3.8 BucketSort . . . . .	35
Aufwand: . . . . .	35
4.4 Hashing . . . . .	35
4.4.1 Grundidee . . . . .	35
4.4.2 Kriterium für eine gute Hash- Funktion . . . . .	36
4.4.3 Methoden zum Erzeugen einer Hashfunktion . . . . .	36
Divisionsmethode . . . . .	36
Multiplikationsmethode . . . . .	36
4.4.4 Umgang mit Kollisionen . . . . .	37
Hashing mit Überlauf („chaining“) . . . . .	37
Hashing mit Ersatzadresse („open addressing“) . . . . .	37
Perfektes Hashing: . . . . .	44
<b>5 Berechenbarkeit und Komplexität</b>	<b>47</b>
5.1 Computer . . . . .	47
5.1.1 Grundidee . . . . .	47

5.1.2	Turing- Maschine . . . . .	47
	Mehrspur- Turing- Maschinen . . . . .	48
	Mehrband- Turing- Maschinen . . . . .	48
	Universelle Turing- Maschine . . . . .	49
5.2	Problem . . . . .	50
5.2.1	Spezialfall Funktion . . . . .	50
5.2.2	Entscheidungsprobleme . . . . .	50
	zugehörige Sprache . . . . .	50
5.2.3	Es gibt unlösbare Probleme . . . . .	50
5.3	Berechenbarkeit . . . . .	51
	Definitionen . . . . .	51
5.3.1	Unentscheidbare Probleme . . . . .	51
	Die Diagonalsprache . . . . .	51
	Das Halteproblem . . . . .	52
	Das spezielle Halteproblem . . . . .	52
	Die universelle Sprache . . . . .	53
	Satz von Rice . . . . .	53
5.4	Der Gödel'sche Unvollständigkeitssatz . . . . .	55
5.4.1	Satz von Church . . . . .	55
	Definition Sprache der wahren Sätze . . . . .	55
	Satz von Church . . . . .	55
5.4.2	Annahmen über Beweise . . . . .	55
	Satz . . . . .	55
	Beweis . . . . .	55
5.4.3	Gödel'scher Unvollständigkeitssatz . . . . .	55
	Satz . . . . .	55
	Beweis . . . . .	56
5.5	Komplexitätsklasse P und NP- Vollständigkeit . . . . .	56
5.5.1	Definitionen . . . . .	56
5.5.2	Ist ein Problem effizient lösbar? . . . . .	56
	Das Cliquenproblem . . . . .	57
5.5.3	Komplexitätsklasse NP . . . . .	58
5.5.4	Nichtdeterministische Turing- Maschine . . . . .	58
	Definition Nichtdeterministische TM . . . . .	58
5.5.5	Komplexitätsklasse NP . . . . .	59
	Definition . . . . .	59
	Satz . . . . .	59
	NP- Vollständigkeit . . . . .	59
	Satisfiability Problem (SAT) . . . . .	60
<b>6</b>	<b>Anhang</b>	<b>64</b>
6.1	Grundbegriffe . . . . .	64
6.2	Pseudocode . . . . .	64

6.3	Beweis der Richtigkeit von Algorithmen . . . . .	66
6.3.1	Terminologie . . . . .	66
partielle Korrektheit . . . . .	66	
Terminieren . . . . .	66	
totale Korrektheit . . . . .	66	
6.3.2	Schleifeninvariante . . . . .	66
6.3.3	Widerspruchsbeweis . . . . .	66
6.4	Laufzeitanalyse . . . . .	67
6.4.1	Asymptotische Notation . . . . .	67
6.4.2	Unterscheidung von Größenordnungen . . . . .	68
6.4.3	untere Schranken . . . . .	68
6.5	Begriffe der Wahrscheinlichkeitstheorie . . . . .	68
(Endlicher Fall) . . . . .	68	
Abzählbar unendlicher Fall . . . . .	68	
6.6	Optimalitätsbeweis des Huffman- Algorithmus' . . . . .	69
6.7	Zu BucketSort . . . . .	74
6.7.1	Beweis der Korrektheit von BucketSort . . . . .	74
Induktionsanfang . . . . .	74	
Induktionsschluss . . . . .	74	
6.7.2	Erweiterung von BucketSort zum Sortieren von Strings . . . . .	74
Definition . . . . .	74	
Der Algorithmus . . . . .	75	
Beweis der Korrektheit . . . . .	75	
Aufwand . . . . .	75	
Umsetzung . . . . .	75	
6.8	Schlussfolgerungen über die Berechenbarkeiten „verwandter“ Sprachen	77
6.9	Zum Gödel'schen Unvollständigkeitssatz . . . . .	78
6.9.1	Mathematische Ausdrücke (Sätze) . . . . .	78
6.9.2	Hilberts Programm . . . . .	78
6.10	Sonstiges . . . . .	78

# 1 Kontrollstrukturen

Die wichtigsten (zur Konstruktion der Turing- Maschine) sind:

- Zusammengesetzte Anweisungen  
"Normaler Programmablauf": Werden sequentiell abgearbeitet.
- bedingte Anweisungen  
"*IF THEN ELSE ENDIF*": Stellen eine Entscheidung dar, von der nur eine abgearbeitet wird.
- Schleifen  
"*WHILE(Bedingung) DO ENDWHILE*": Anweisungsblock wird so lange abgearbeitet, bis die *Bedingung false* wird.  
Ausformuliert als *WHILE*, *DO – WHILE* und *FOR(...)*

Alle Programmiersprachen, die diese 3 strukturellen Anweisungen kennen, sind gleich mächtig. Es gibt aber auch äquivalente Alternativen (z.B. *goto*- Sprungbefehle). In Java gibt es *goto* zwar nicht, aber stattdessen *continue* (springt sofort zum Schleifenanfang) und *break* (beendet Schleife/ bedingten Anweisungsblock).

## 1.1 Rekursion

### 1.1.1 Idee

Um Rekursion zu verstehen, muss man erst Rekursion verstanden haben:

Ein rekursiver Methodenaufruf bedeutet, dass sich eine Methode selbst aufruft. Dabei sollte (um ein Terminieren zu gewährleisten) entweder der übergebene Parameter bei jedem Durchlauf oder eine Klassenvariable, sozusagen der „Zustand“ des Programms verändert werden. Mathematisch: Eine Funktion wird (in Teilen) durch sich selbst definiert.

Selten ist eine Rekursion effizienter als eine Iteration, vor allem, wenn der Rekursionsbaum eine Liste darstellt (z.B. Endrekursion). Viele Probleme lassen sich dadurch jedoch eleganter und mit wesentlich weniger Programmieraufwand lösen, darüber hinaus ist Rekursion ein wichtiger Bestandteil des „Divide- and Conquer“- Prinzips.

Es gilt:

$$\text{Anzahl rekursiver Aufrufe} \quad \hat{=} \quad \# \text{ Knoten im Rekursionsbaum} = T(n)$$

$$\text{Rekursionstiefe} = h(n) \quad \hat{=} \quad \text{Höhe des Stacks/ Aufrufbaums} + 1$$

### 1.1.2 Backtracking

Wichtiger Problemlösungsansatz, basiert auf Trial and Error, gut rekursiv zu implementieren als eine Art Tiefensuche:

Wenn du dir irgendwo nicht sicher bist, probiere alle Möglichkeiten aus. Gibt es einen Fehler, nimm die letzte Änderung zurück und wähle eine andere Möglichkeit. Geht alles glatt, ist die Lösung korrekt.

## 2 Kodierung

### 2.1 Eindeutige Decodierbarkeit

Ein Code heißt eindeutig decodierbar  $\Leftrightarrow$  Verschiedene Original- Dateien führen zu verschiedenen codierten Dateien.

Eine injektive Codierung wird gefordert.

### 2.2 Block- Code

- Jedes Zeichen hat die selbe Länge in der Binär- Darstellung
- Dadurch sehr einfache Kodierung und Dekodierung
- Häufig verwendete Codierung
- Beispiele sind ASCII, Unicode, etc.
- Nachteil: „Platzverschwendungen“

### 2.3 Präfix- Code

- Variable Länge: Zeichen *können* unterschiedliche viele Bits in der Binär- Darstellung belegen.
- Eine wichtige Klasse von eindeutig decodierbaren Codes
- Ein Code heißt Präfix- Code, wenn kein Codewort als Präfix eines anderen Codeworts auftaucht.
- Jeder Blockcode ist ein Präfix- Code
- Präfix- Codes lassen sich mit binären Bäumen identifizieren:

Ein Blatt im Baum entspricht einem Zeichen.

Zeichen  $c \in C \leftrightarrow$  Weg im Baum von der Wurzel bis zum zugehörigen Blatt.  
Die Kanten auf dem Weg geben die Bitfolge: 0 nach links, 1 nach rechts.

Dabei garantiert die Präfix- Code- Eigenschaft, dass Zeichen bei der Konstruktion des Baums nur in den Blättern auftauchen.

## 2.4 Datenkompression mit Huffman- Algorithmus

**Idee** gängige Block- Codes (z.B. ASCII) umwandeln in Präfix- Code variabler Länge. Die kürzesten Präfix- Code- Zeichen werden für die häufigsten Block- Code- Zeichen benutzt.

Suche einen binären Baum  $T$ , dessen Blätter dem Zeichensatz  $C$  entsprechen und der die Größe  $B(T) = \sum_{c \in C} f(c) \cdot h_T(c)$  minimiert, d.h.  $B(T) \leq B(T') \forall$  Präfixcodes  $T'$

### Algorithmus

**Input:** Ein Zeichensatz  $C$ , Häufigkeiten des Auftretens  $f : C \rightarrow \mathbb{N}, c \mapsto f(c)$

**Output:** Ein optimaler Präfixcode als Huffman Baum

Fasse jedes Zeichen  $c \in C$  als einelementigen Baum auf und füge es in eine Priority Queue  $Q$  ein, wobei die Häufigkeit  $f(c)$  als Schlüsselwert dient.

**while**  $Q$  enthält mehr als einen Baum **do**

Wähle die beiden Bäume  $T_1$  und  $T_2$  mit den kleinsten Häufigkeiten (muss nicht eindeutig sein).

Entferne sie aus  $Q$ .

Konstruiere einen neuen Baum aus den  $T_1, T_2$  als Teilbäume unter einer neuen Wurzel und gebe ihm die Häufigkeit  $f(T_1) + f(T_2)$ .

Füge diesen Baum in  $Q$  ein.

**end while**

**return** (Den einzig übrig gebliebenen Baum)  $T$  in  $Q$ .

Dieser Baum ist der Huffman Code/ Huffman Baum und ein optimaler Präfixcode.

### Bemerkungen zu Huffman- Codes

Huffman	alternativ
zeichenweise Codierung	nicht zeichenweise (kodiere ganze Strings mit einem Code- Wort)
statische Kodierung	dynamisch: „on the fly“
Kodierung gilt für gesamte Datei, nach Häufigkeitsanalyse	Codeworte werden beim Lesen der Datei erzeugt und dynamisch verändert („adaptiv“)
verlustfrei	verlustbehaftet

### 3 Datenstrukturen

In Datenstrukturen können Daten in großer Menge und durchaus sortiert auftreten.

Sie alle haben

- einen Wertebereich
- Operationen
- Komponenten- Daten (atomar oder selbst strukturiert)
- Regeln, die das Zusammenwirken der Komponenten zur Struktur definieren

Lineare Strukturen haben darüber hinaus eine Ordnung, die eine erste und letzte Komponente sowie einen Vorgänger und Nachfolger für jede Komponente definiert.

Lineare Strukturen ermöglichen beliebigen Zugriff auf eine einzelne Komponente (random access), im Gegensatz zum sequentiellen Zugriff, bei dem vor der n-ten Komponente erst auf die n-1 ersten Komponenten zugegriffen werden muss.

Homogene Komponenten sind untereinander alle gleichartig (von gleichem Typ)

#### 3.1 Array

Arrays sind besondere lineare Datenstrukturen. Sie kennzeichnet insbesondere:

- eine feste Komponentenzahl. Sie wird zur Laufzeit festgelegt.
- der direkte Zugriff auf Komponenten per Indizes (vom Typ *int*).
- ein homogener Grundtyp.
- die Möglichkeit, Indizes zu berechnen
- die Erzeugung einer `ArrayIndexOutOfBoundsException`, wenn der Index bei Zugriff nicht im gültigen Bereich liegt.
- das einfache Durchlaufen per *for-* Schleife.

Dieses Konstrukt entspricht in der Mathematik einem Vektor.

## 3.2 Liste

Dies ist keine lineare Datenstruktur. Hier hat jedes Datum seinen Nachfolger (und evtl. Vorgänger) selbst als Referenz abgespeichert. Bei Durchlaufen der Liste werden die Referenzen kopiert und so mit den Daten gearbeitet.

Auf der einen Seite ist kein random access möglich, stattdessen kann die Liste besonders schnell verlängert werden und Komponenten können sehr einfach eingefügt und entfernt werden.

Als Modell kann ein Seil aus zusammengeknoteten Fäden oder Bettlaken, (wie bei Gefängnisbrüchen) dienen. Ein neues Stück lässt sich an einer bestimmten Stelle leicht anknoten, genauso wie sich ein Stück leicht entfernen lässt.

Operationen auf einer Liste:

- Erzeugung einer leeren Liste
- Einfügen vor oder hinter einem Element
- Löschen eines Elements
- Prüfen, ob Liste leer ist
- Test auf Ende der Liste
- an den Anfang der Liste gehen
- ein Element weiter gehen

## 3.3 Stack

Der Stack ist eine besondere Art Liste, bei der nur am Kopf zugegriffen werden kann (LIFO). Er wird zum Beispiel als Runtime- Stack oder zum Finden von Paaren in einer Menge verwendet.

Die wichtigen Operationen lauten

- push (Element hinzufügen)
- peek (Element nur lesen)
- pop (Element lesen und entfernen).

Anwendungsmöglichkeiten:

- Laufzeit- Stack in Java
- Überprüfen korrekter Klammerung (Erkennen korrespondierender Klammernpaare)
- Realisierung von Rekursion
- Auswertung arithmetischer Ausdrücke
- ...
- außerhalb der Informatik: Rangierbahnhöfe

Anschauliche Darstellungen sind ein Stapel oder ein Kellerautomat, bzw. ein halb verstelltes Bücherregal.

## 3.4 Queue

Die Queue oder Schlange ist eine weitere spezielle Liste. Hier wird am Schwanz eingefügt, am Kopf gelesen und entfernt.

Die Schlange findet gerne Anwendung, um Wartezeiten zu verteilen, also als Warteschlange.

## 3.5 Graphen

Bestehen aus Knoten und gewichteten Kanten

$$V = \text{Knotenmenge} = \{1, \dots, n\} \text{ bzw. } \{0, \dots, n - 1\}$$

$$E = \text{Kantenlänge}; E \subseteq \{u, v; u, v \in V\}$$

### 3.5.1 Weg

Der Weg in einem Graphen ist eine Folge von Kanten und eine Richtung pro Kante, sodass der Endpunkt einer Kante der Anfangspunkt der nachfolgenden Kante ist. Wiederholungen von Knoten und Kanten sind zulässig.

### 3.5.2 Kantenbewertungen

Pro Kante wird eine Zahl ("Entfernung") eingeführt.

### 3.5.3 Länge eines Weges

Die Summe der Bewertungen der Kanten in der Folge.

### 3.5.4 Kürzester Weg von Knoten i zu Knoten j

Der Weg mit der kürzesten Länge unter allen Wegen unter i nach j (es kann mehrere kürzeste Wege geben).

### 3.5.5 Gerichtete (Di-) Graphen

V, E bleiben gleich. Aber es werden gerichtete Kanten (Bögen) eingeführt.

Ein (endlicher) Digraph G besteht aus:

$$(\text{endlicher}) \quad \text{Knotenmenge} \quad V$$

$$\text{Kantenmenge} \quad E \subseteq V \times V \quad \{(v, v) | v \in V\}$$

Schreibweise:  $G = (V, E)$

Kantenbewertung: Funktion  $w : E \rightarrow \mathbb{R}$

Nun werden Richtungen eingeführt, sodass der Durchlauf nur in einer Richtung möglich ist (wie auf Abbiegespuren, Einbahnstraßen). Der Begriff vom kürzesten Weg gilt entsprechend.

⇒ Ungerichtete Graphen können als Spezialfall von Digraphen gelten, die antiparallele Kanten haben, bzw. symmetrisch gerichtet sind. Somit reicht es aus, die Probleme nur für gerichtete Graphen zu diskutieren.

Achtung: Es sind auch negative Kantenbewertungen möglich.

**Zykel** Ein gerichteter Weg mit gleichem Anfangs- und Endknoten.

Enthält ein Weg zwischen i und j einen Zykel negativer Länge (negative Zykel), so gibt es keinen kürzesten Weg zwischen i und j.

Algorithmen liefern entweder den kürzesten Weg oder den negativen Zykel.

### 3.5.6 Datenstrukturen für Digraphen

**Adjazenzliste** Für jeden Knoten i gibt es eine Liste  $Adj(i)$  der Knoten j, die Endpunkt einer Kante  $(i, j)$  sind. Es ergibt sich ein Array von Listen, die man in einer Adjazenzmatrix zusammenfassen kann.

**Adjazenzmatrix**  $A = (a_{ij})$  mit  $a_{ij} = 1$ , falls  $(i, j)$  Kante, 0 sonst.

**Entfernungsmatrix**  $a_{ij}$  = Kantenbewertung, falls  $(i, j)$  Kante; 0, falls  $i = j$ ;  $\infty$  sonst

### 3.5.7 Bellman- Gleichungen

$u_{ij}^{(m)}$  := Länge eines kürzesten Weges von i nach j mit höchstens m Kanten, falls dieser existiert;  $\infty$  sonst

Bellman- Gleichung:

$$u_{ij}^{(1)} = a_{ij}$$

$$u_{ij}^{(m+1)} = \min_{k=0, \dots, n-1} [u_{ik}^{(m)} + a_{kj}] \quad \text{für } m \geq 1$$

Es gilt das Prinzip der optimalen Teilstuktur: In einem kürzesten Weg sind alle Teilwege selbst kürzeste Wege.

## 3.6 Bäume

Spezieller Digraph, der genau einen Wurzelknoten hat. Zu jedem anderen Knoten führt genau eine Kante und beliebig viele führen weg.

### 3.6.1 voller binärer Baum:

Ein binärer Baum, bei dem alle Schichten (bis auf die letzte) voll besetzt sind.  
„Jeder Knoten hängt so hoch wie möglich.“

**vollständiger Baum:** Jeder Knoten hat 0 oder 2 Kinder → Heap ist kein vollständiger Baum.

## 3.7 Suchbäume

- Verwaltet dynamische Daten
- Unterstützt folgende 3 Operationen in  $O(\log n)$ :
  - suchen
  - einfügen
  - löschen

Idee:

- binäre Bäume
- Knoten eines Baumes  $\hat{=}$  Datensätze / Elemente mit Schlüsseln

**Suchbaumeigenschaft:** Für jeden Knoten  $v$  des Baumes haben alle Elemente im linken Teilbaum von  $v$  einen kleineren Schlüssel als  $v$ , die Elemente im rechten Teilbaum einen größeren Schlüssel  $\Leftrightarrow$  In Inorder- Traversierung sind die Schlüssel aufsteigend sortiert.

### 3.7.1 Suchen

**Input:** Suchbaum  $T$ , gesuchter Schlüssel  $key$

**Output:** gesuchtes Element oder -1, falls Element nicht vorhanden

```
Vergleiche gesuchten Schlüssel mit Schlüssel in Wurzel
if Gleichheit then
    return Wurzelement
else
    if key < Root.key then
        Suche rekursiv im linken Teilbaum weiter
    else
        Suche rekursiv im rechten Teilbaum weiter
    end if
end if
return -1
```

**Aufwand:** Suchen nach Knoten  $v$  erfordert genau  $h_T(v) + 1$  Vergleiche.

### 3.7.2 Einfügen

Idee: Suche nach einzufügendem Knoten, bis in Blatt angekommen. Erzeuge entsprechenden Kind- Knoten und hänge ihn auf der richtigen Seite an das Blatt an.

**Aufwand:** Wie Suchen  $O(h(T))$  Vergleiche

### 3.7.3 Löschen

Suche zu löschenen Knoten

**if** Knoten ist Blatt **then**

    Blatt löschen, fertig.

**end if**

**if** Knoten hat genau 1 Kind **then**

    Ersetze Knoten durch sein Kind, fertig.

**end if**

**if** Knoten  $v$  hat genau 2 Kinder **then**

    Suche im rechten Teilbaum von  $v$  den linkesten Knoten  $w$ , d.h. mit kleinstem Schlüssel im Teilbaum

    Vertausche Inhalte von  $v$  und  $w$

    Lösche  $w$ , das kein linkes Kind hat.

**end if**

Beachte:  $w$  ist Nachfolger von  $v$  beim Inorder- Durchlauf  $\rightarrow$  Suchbaumeigenschaft bleibt erfüllt.

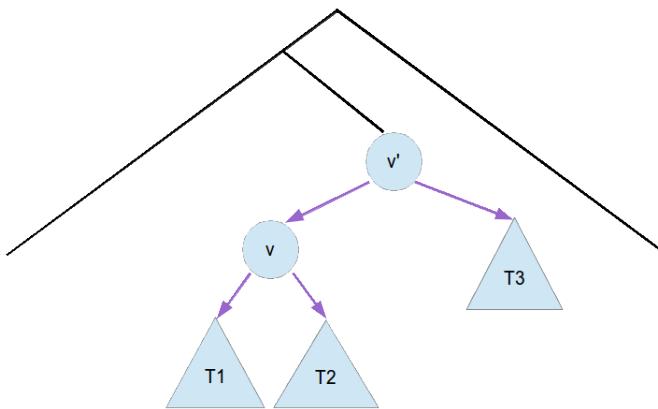
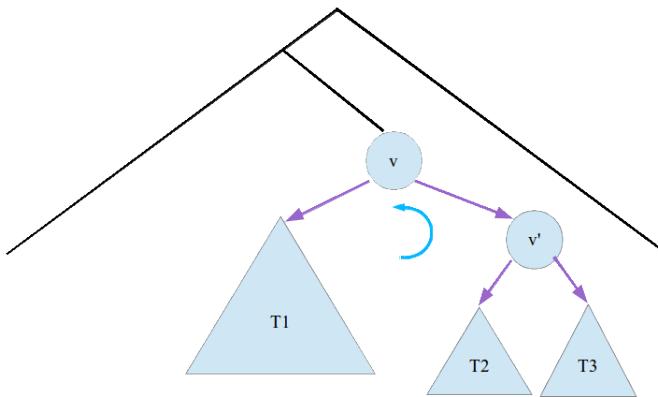
#### Aufwand:

- Suchen von  $v$  und  $w$  in  $O(h(T))$
- Austauschen der Inhalte:  $O(1)$
- Löschen von  $w$ :  $O(1)$

Insgesamt  $O(h(T))$ .

Alle 3 Operationen benötigen den Aufwand  $O(h(T))$ . Versuche also, Suchbäume mit spezieller Struktur zu verwenden, sodass  $h(T) \in O(\log n)$ .

### 3.7.4 Rotation



Rechtsrotation symmetrisch dazu und inverse Operation.

Beachte: Suchbaumeigenschaft bleibt erfüllt, denn „ $T_1 < v < T_2 < v' < T_3$ “.  
Der Inorder- Durchlauf bleibt erhalten.

Links- und Rechtsrotation sind zusammen mächtig genug, um einen Suchbaum in jede gewünschte Gestalt zu bringen.

**Idee:** Bei dieser Operation wird der linke Teilbaum höher und der rechte weniger hoch.

**Aufwand:** Referenz auf  $v$  gegeben, aktualisiere 3 Referenzen:  $O(1)$

### 3.7.5 AVL- Bäume

Eine balancierte Klasse von Suchbäumen, die die folgenden Eigenschaften hat:

1. Für jedes  $n$  gibt es in der Klasse einen balancierten Suchbaum  $T$  mit  $h(T) \in O(\log n)$ .
2. Die drei Basis- Operationen können auf dieser Klasse in  $O(h(T))$  durchgeführt werden.
3. Die Klasse ist abgeschlossen unter den drei Basis- Operationen, d.h. nach Einfügen und/ oder Löschen eines Knotens muss der resultierende Baum wieder zu dieser Klasse gehören.

#### Definition Balance

$T$  binärer Suchbaum ,  $v \in V$  Knoten ,  $T_l(v), T_r(v)$  - evtl. leere Teilbäume von  $v$

$$\Rightarrow \beta(v) := h(T_r(v)) - h(T_l(v)), \quad h(\emptyset) = -1$$

**Definition AVL- Baum** Ein binärer Suchbaum  $T$  heißt AVL- Baum, falls für jeden Knoten  $v$  in  $T$  gilt:  $|\beta(v)| \leq 1$

Für einen AVL- Baum  $T$  mit  $n$  Knoten gilt:  $h(T) \leq 2 \cdot \log(n)$

**Idee des Beweises** Betrachte extreme AVL- Bäume, die zu vorgegebener Höhe eine minimale Anzahl Knoten enthalten.

**Definition**  $T$  heißt extremer AVL- Baum der Höhe  $H$ , falls  $T$  AVL- Baum ist und  $n(T) = \min\{n(T') \mid T' \text{ AVL - Baum}, h(T') = h\}$

Vermutete Struktur eines extremalen AVL- Baums:

- 1 Höhendifferenz (Balance) ist in jedem inneren Knoten 1 oder -1.
- 2 Für  $h \geq 2$  sind  $T_l$  und  $T_r$  extreme AVL- Bäume zur Höhe  $h - 1$  und  $h - 2$

**Lemma** Ein extremer AVL- Baum zur Höhe  $h \geq 2$  hat als Teilbäume extreme AVL- Bäume zur Höhe  $h - 1$  und  $h - 2$ .

**Beweis** Es sei  $v$  Wurzel von  $T$ .

Falls  $\beta(v) = 0$ , dann kann man aus dem rechten Teilbaum alle Knoten auf dem Level  $h$  löschen und erhält wieder einen AVL- Baum der Höhe  $h$  mit weniger Knoten → **Widerspruch!**

⇒ Teilbäume  $T_l$  und  $T_r$  haben die Höhen  $h - 1$  und  $h - 2$ .

Annahme:  $T_l$  kein extremer AVL- Baum zur Höhe  $H(T_l)$ .

Dann ersetze  $T_l$  im Suchbaum  $T$  durch extremalen AVL- Baum der Höhe  $H(T_l)$ . Dieser besitzt weniger Knoten, also besitzt  $T$  nach dem Ersetzen weniger Knoten, ist aber weiterhin ein AVL- Baum zur Höhe  $h \rightarrow \mathbf{Widerspruch!}$

Genauso:  $T_r$  ist extremal.

Rekursionsformel für die Anzahl der Knoten eines extremalen AVL- Baumes:

$$n(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ 1 + n(h-1) + n(h-2) & h \geq 2 \end{cases}$$

**Lemma** Die minimale Knotenzahl  $n(h)$  eines AVL- Baums der Höhe  $h$  erfüllt:

$$n(h) \geq 2^{\frac{h}{2}}$$

## Beweis

Beweis: Per Induktion über  $\mathbb{R}$

$$\begin{array}{ll} \text{I.A. } h=0 & n(h) = 1 = 2^{\frac{0}{2}} \checkmark \\ h=1 & n(h) = 2 = 2^{\frac{1}{2}} = \sqrt{2} \checkmark \end{array}$$

$$\begin{aligned}
 \text{I.S.} \quad h(R) &= 1 + h(R-1) + h(R-2) \\
 h \geq 2 \quad &= 1 + 1 + h(R-2) + h(R-3) + h(R-2) \\
 &= 2 + \cancel{2 \cdot h(R-2)} + h(R-3) \\
 &\geq 2 \cdot h(R-2) \\
 &\geq 2 \cdot 2^{\frac{R-2}{2}} = 2^{\frac{R}{2}}
 \end{aligned}$$

Kos. → Auszüge des Satzes:

$$\log(nGh) = \frac{q}{2}$$

man kann lernen  
Schwierigkeiten... logisch

$$44 \dots \log_4 \rightarrow 44 \leq 2 \cdot \log(n \cdot h)$$

$$\text{Rels-Gly, York} \rightarrow h \leq 2 \cdot \log(n) \text{ f. dcl}$$

Wolke 11.000ft 11-08

Hohe parallele zu den VL-Bauweisen der Hohen Rauten und der Filonaci-Zellen (da  $n \geq 4$  ist).

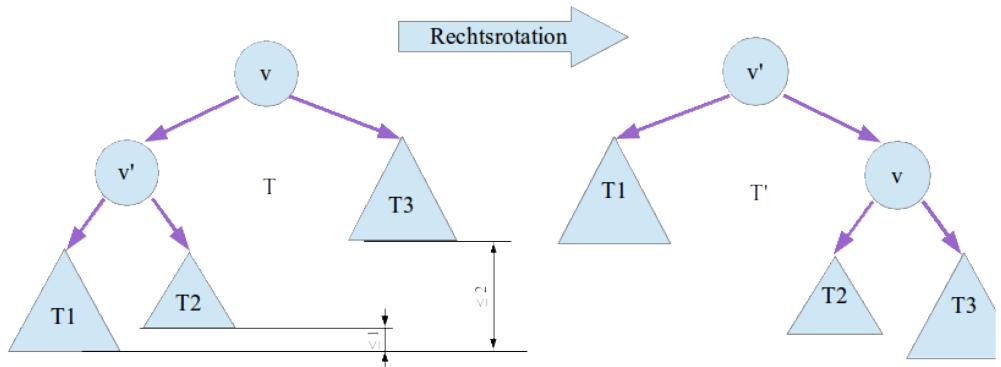
**Rotationslemma für AVL- Bäume** Es gilt, falls  $|\beta(v)| = 2$ :

- a Rotation oder Doppelrotation:  $T \rightarrow$  AVL- Baum,  $h(T') \leq h(T)$
- b Art der Rotation ist in  $O(1)$  zu ermitteln.
- c Rotation braucht  $O(1)$ - Aufwand.
- d veränderte Balancen in  $O(1)$  aus ursprünglichen zu berechnen.

4 Fälle in Abhängigkeit von den ersten beiden Kanten des Weges von der Wurzel  $v$  in die tiefste Tiefe des Baumes.

1. Der Weg von  $v$  in die größte Tiefe ist links- links.

Einfache Rechtsrotation



Veränderte Situation in  $T'$ :

$$\beta'(v) = \begin{cases} -1 & \text{falls } \beta(v') = 0 \\ 0 & \text{falls } \beta(v') = -1 \end{cases}$$

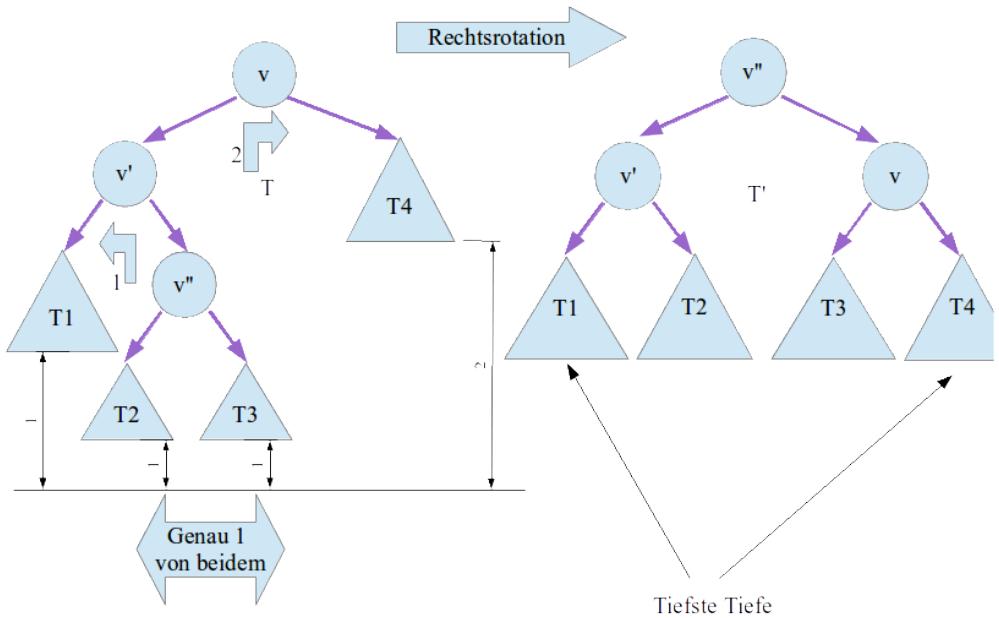
$$\beta'(v') = \begin{cases} 1 & \text{falls } \beta(v') = 0 \\ 0 & \text{falls } \beta(v') = -1 \end{cases}$$

$$\beta'(u) = \beta(u) \quad \forall \quad u \neq v, v'$$

$$h(T') = \begin{cases} h(T) & \text{falls } \beta(v') = 0 \\ h(T) - 1 & \text{falls } \beta(v') = -1 \end{cases}$$

2. Weg in die tiefste Tiefe geht links- rechts.

Links- Rechtsrotation



- $T_1$  verändert Tiefe nicht
- $T_4$  rutscht 1 nach unten
- $\Rightarrow$  beide liegen am Ende auf derselben Höhe.

Veränderte Situation in  $T'$ :

$$\beta'(v'') = \begin{cases} 0; & \beta(v'') = 0 \\ 0; & \beta(v'') = 1 \\ 0; & \beta(v'') = -1 \end{cases}$$

$$\beta'(v') = \begin{cases} 0; & \beta(v'') = 0 \\ -1; & \beta(v'') = 1 \\ 0; & \beta(v'') = -1 \end{cases}$$

$$\beta'(v) = \begin{cases} 0; & \beta(v'') = 0 \\ 0 & \beta(v'') = 1 \\ 1; & \beta(v'') = -1 \end{cases}$$

$$\beta'(u) = \beta(u) \quad \forall \quad u \neq v, v', v''$$

$$h(T') = h(T) - 1$$

**Beachte:** Berechnung der neuen Balancen in  $T'$  und neue Höhe  $h(T')$  in beiden Fällen in  $O(1)$  zu berechnen.

**Bemerkung:** Falls der Weg in die tiefste Tiefe sowohl links- links, als auch links- rechts geht, so muss wie in Fall 1 verfahren werden, denn sonst reicht allein  $T_1$  in die tiefste Tiefe und möglicherweise eins zu weit. Dann würde bei dem Fall des flachen  $T_2$  gelten:  $\beta(v') = -2$

3. Weg in die tiefste Tiefe beginnt rechts- rechts. → Analog zu 1. einfache Linksrotation.
4. Weg in die tiefste Tiefe beginnt rechts- links und *nicht* rechts- rechts. → Analog zu 2. Doppel- Links- Rotation

### Implementation der Basis- Operationen in AVL- Bäumen

**Suchen:** Wie bei allgemeinen Suchbäumen mit Aufwand  $O(h(T)) = O(\log n)$

**Einfügen:** Suche nach Schlüssel des neuen Elements endet erfolglos in einem Blatt. Hänge neues Element als linkes oder rechtes Kind an dieses Blatt an.

**Beachte:** Balancen haben sich höchstens auf dem Pfad von der Wurzel zum neuen Element geändert. → Gehe von neuem Element entlang der Spur zurück zur Wurzel und update und repariere die Balancen, falls nötig.

D.h. falls  $\beta(v) \in \{-2; 2\}$  repariere dies sofort mit Rotationslemma.

<b>Aufwand:</b>	Suchen	$O(\log n)$
	Update & Balance- Korrektur entlang Spur	$O(\log n)$

### Praktische Verbesserungsmöglichkeiten:

Stellt man beim Verfolgen der Spur fest, dass sich die Tiefe des Teilbaums an einem Knoten  $v$  nicht verändert hat, so kann man an dieser Stelle abbrechen.

Findet man beim Verfolgen der Spur nach oben einen Knoten  $v$  mit  $\beta(v) \in \{2; -2\}$  und behebt dies mithilfe des Rotationslemmas, so kann man danach stoppen, da die Höhe des Teilbaums an diesem Knoten wieder der Ausgangshöhe entspricht.

**Begründung:** Sei  $v_1$  der erste gefundene Knoten mit  $\beta(v_1) \in \{2; -2\} \rightarrow$  Höhe des TB an  $v_1$  um 1 gewachsen durch Einfügen von  $v$ . Unterscheide symmetriebedingt folgende zwei Fälle:

- links- rechts- Weg: Die Tiefe des Baumes schrumpft um 1
- links- links- Weg: Bemerke, dass beide TB von  $w$  die selbe Tiefe hatten, d.h. vor Einfügen von  $v$  perfekt balanciert waren.  
Bei Rotation schrumpft die Tiefe des TB an dem Knoten um 1.

## Löschen

- wie in allgemeinen Suchbäumen
- aber: Balance möglicherweise verletzt  $\rightarrow$  gehe daher entlang der Spur vom gelöschten Element zurück zur Wurzel, update Balancen, repariere ggf.
- Aufwand:  $O(\log n)$

**Bemerkung:** Leider gilt das schöne Resultat für das Einfügen *nicht* für den Fall des Löschens, denn es gibt AVL- Bäume, bei denen nach dem Löschen eines Blattes an jedem Knoten der Spur das Rotationslemma angewendet werden muss.

Worst- Case- Beispiel: Lösche Blatt minimaler Tiefe in einem extremalen AVL- Baum.

### 3.7.6 Optimale statische Suchbäume

- Kenne alle Suchschlüssel  $S = \{s_1, \dots, s_n\}$
- Kenne deren Zugriffshäufigkeiten  $\beta_1, \dots, \beta_n$ , mit denen auf  $s_1, \dots, s_n$  zugegriffen wird.
- Beispiel: Daten auf CD
- Ziel: Bestimme Suchbaum  $T$  für  $S$  und zu gegebenen  $\beta_i$ , sodass die mittlere Zugriffszeit klein ist.

Die Anzahl nötiger Vergleiche beim Suchen nach  $s_i$  beträgt  $h_T(s_i) + 1$ .

Die mittlere Zugriffszeit ist:

$$E_p(T) = \frac{\sum_{i=1}^n \beta_i \cdot (h_T(s_i) + 1)}{\sum_{i=1}^n \beta_i}$$

Die konkreten Zugriffswahrscheinlichkeiten berechnen sich folgendermaßen:

$$p_i = \frac{\beta_i}{\sum_{j=1}^n \beta_j}$$

**Ziel:** Finde Suchbaum  $T$ , sodass  $E_p(T)$  minimal. Entwickle Algorithmus, der diesen  $T$  findet.

$$\begin{aligned} T \text{ ist optimaler statischer Suchbaum, falls } E_p(T) \leq E_p(T') & \quad \forall T' \text{ Suchbäume} \\ \Leftrightarrow \underbrace{\sum_{i=1}^n \beta_i (h_T(s_i) + 1)}_{:=c(T)} & \leq \underbrace{\sum_{i=1}^n \beta_i (h_{T'}(s_i) + 1)}_{:=c(T')} \end{aligned}$$

Betrachte im Folgenden absolute Häufigkeiten  $\beta_i$  statt  $p_i$  und  $c(T)$  statt  $E_p(T)$ .

**Struktur von optimalen Suchbäumen** Prinzip der optimalen Substruktur:

Sei  $T$  Suchbaum für  $S$ ,  $T'$  Teilbaum von  $T$ .

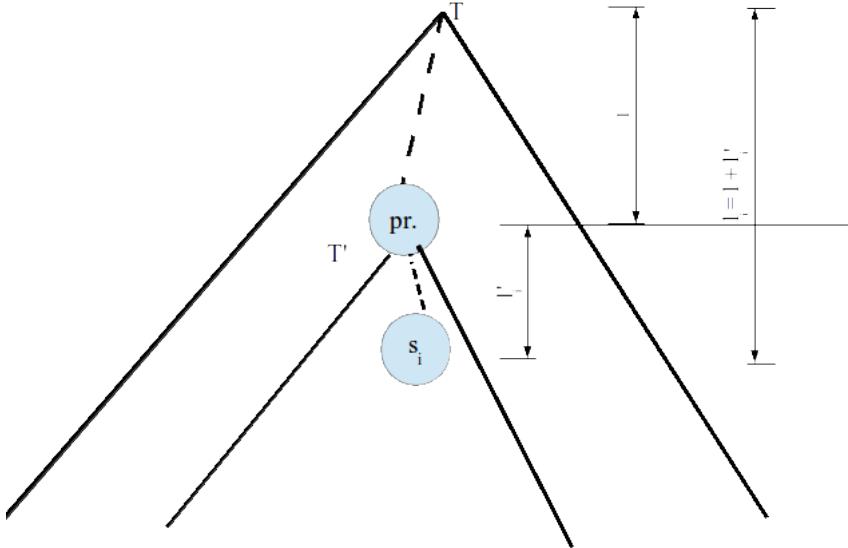
$\Rightarrow$  Die Schlüsselmenge  $S'$  der Schlüssel in  $T'$  bildet konsekutive Teilsequenz (Intervall)  $s_i < s_{i+1} < \dots < s_j$  von  $s_1 < s_2 < \dots < s_n$ .

**Beweis:**

Der Inorder- Durchlauf durch  $T$  liefert die Schlüssel in  $S$  in aufsteigender Sortierung (Suchbaum- Eigenschaft). Dieser Durchlauf besucht die Knoten in einem Teilbaum  $T'$  nacheinander ohne Unterbrechung durch Knoten außerhalb von  $T'$ .

$\Rightarrow T$  ist optimal für  $S$  und  $\beta_1, \dots, \beta_n \Rightarrow T'$  optimal für  $S' = \{s_i, s_{i-1}, \dots, s_j\}$  und  $\beta_i, \beta_{i+1}, \dots, \beta_j$

Beweis:



$$l_i := h_T(s_i) + 1; \quad l'_i := h_{T'}(s_i) + 1; \quad l := h_T(s_k)$$

$$\Rightarrow l_i = l'_i + l$$

$$\begin{aligned}
 C(T) &= \sum_{q=1}^n \beta_q \cdot l_q = \underbrace{\sum_{q=1}^{i-1} \beta_q \cdot l_q}_{V(\text{ vor } T')} + \underbrace{\sum_{q=i}^j \beta_q \cdot l_q}_{N(\text{ nach } T')} + \underbrace{\sum_{q=j+1}^n \beta_q \cdot l_q}_{N(\text{ nach } T')} \\
 &= V + \sum_{q=i}^j \beta_q(l + l'_q) + N = \underbrace{V + N + \sum_{q=i}^j \beta_q \cdot l}_{\text{Konstante unabhängig von } T'} + \underbrace{\sum_{q=i}^j \beta_q \cdot l'_q}_{C(T'), \text{ minimal} \Rightarrow C(T') \text{ minimal}}
 \end{aligned}$$

$$C(T') \leq C(T'') \quad \forall \quad \text{Suchbäume } T'' \text{ für } s_i, \dots, s_j; \beta_i, \dots, \beta_j$$

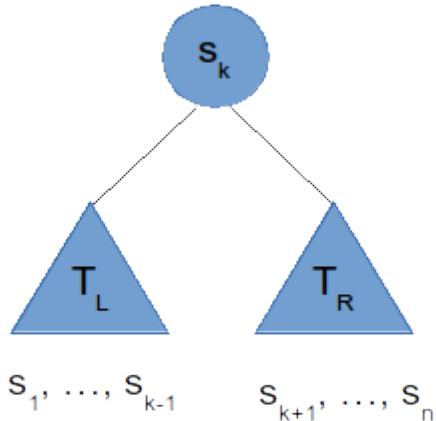
Sonst wähle  $\bar{T}$  mit  $T''$  als Teilbaum statt  $T'$ . Dafür gilt

$$C(T) > C(\bar{T}), \text{ falls } C(T') > C(T'')$$

Nutze die Einsicht: Falls  $s_k$  Wurzel des optimalen statischen Suchbaums ist, dann sieht er so aus: Dabei ist

$T_l$  opt. stat. SB f.  $s_1, \dots, s_{k-1}; \beta_1, \dots, \beta_{k-1}$  Das heißt  
 $T_r$  opt. stat. SB f.  $s_{k+1}, \dots, s_n; \beta_{k+1}, \dots, \beta_n$

$$C(\text{opt. SB}) = \min_{h=1, \dots, n} \left[ C(\text{opt. SB für } s_1, \dots, s_{k-1}) + C(\text{opt. SB für } s_{k+1}, \dots, s_n) + \sum_{i=1}^n \beta_i \right]$$



Für den Algorithmus bedeutet dies:

- Betrachte jede mögliche Wurzel  $s_k, k = 1, \dots, n$
- Für diese Wurzel hänge optimale Teilbäume an
- Wurzel auswählen, die  $C$  minimiert.

Es sei  $T_{ij}$  Suchbaum für  $s_{i+1}, \dots, s_j$ . Wenn  $T_{ij}$  die Wurzel  $s_k$  hat, dann sind die Teilbäume  $T_{i,k-1}$  und  $T_{k,j}$ . Zugehörige Kosten:

$$C(T_{ij}) =: C_{ij} = \beta_{i+1} \cdot l_{i+1}^{ij} + \dots + \beta_j \cdot l_j^{ij} = \beta_{i+1} \cdot (l_{i+1}^{ij} - 1) + \dots + 0 + \beta_j (l_j^{ij} - 1) + \underbrace{\sum_{q=i+1}^j \beta_q}_{W_{ij}}$$

$$C_{ij} = C_{i,k-1} + C_{k,j} + W_{ij}$$

Wende Prinzip der optimalen Substruktur an:

$$C_{ij}^{opt} = \min_{i+1 \leq k \leq j} \left( C_{i,k-1}^{opt} + C_{kj}^{opt} + W_{ij} \right)$$

Idee des Algorithmus': Berechne alle  $C_{ij}^{opt}$  und zugehörige  $T_{ij}^{opt}$  (optimale Teilbäume) nach aufsteigender Länge des Intervalls  $i+1, \dots, j$ .

Dieser Algorithmus ist Beispiel für das Prinzip der dynamischen Optimierung (Programmierung).

n	Bäume mit 1 Knoten	n
n-1	Bäume mit 2 Knoten	n-1
n-2	Bäume mit 3 Knoten	n-2
<b>Aufwand:</b>	...	...
2	n-1	2
1	n	1
	Summe	$\frac{n(n+1)}{2} \in O(n^2)$

Aufwand für die Berechnung eines dieser optimalen Teilbäume:  $O(n)$  (Minimum über  $n$  Terme)

**Gesamt:**  $O(n^2 \cdot n) = O(n^3)$

### 3.8 Priority Queue

Ist eine dynamische Datenstruktur. Implementationsmöglichkeiten als

- Array (semi- dynamisch)
- voller binärer Baum
- sortierte Liste

Folgende Operationen werden unterstützt:

#### Implementation als voller binärer Baum - Aufwand:

Operation	Name	Komplexität
Einfügen einer Komponente	insert()	$O(\log n)$
Zugriff auf Komponente mit größtem Schlüssel	max()	$O(1)$
Entferne Komponente mit größtem Schlüssel	extractMax()	$O(\log n)$
Ändern eines Schlüssels	changeKey	$O(\log n)$

Implementierungsdetails:

- Die Komponente mit größtem Schlüssel ist in der Wurzel.
- *insert()* - Hangle vom Zeiger *last* auf letztes Blatt nach oben, bis Knoten linkes Kind ist. Gehe dann in rechten Ast, möglichst weit nach links und füge ein. Rufe dann *heapify()* auf.

#### Implementation als sortierte Liste - Aufwand:

Operation	Name	Komplexität
Einfügen einer Komponente	insert()	$O(n)$
Zugriff auf Komponente mit größtem Schlüssel	max()	$O(1)$
Entferne Komponente mit größtem Schlüssel	extractMax()	$O(1)$
Ändern eines Schlüssels	changeKey	$O(n)$

### 3.9 Heap

- spezielle Variante einer Priority Queue
- homogener Komponententyp
- alle Komponenten haben einen Schlüssel
- Operationen wie Priority Queue als voller binärer Baum:

**Die Heap- Bedingung:** Für jeden Knoten gilt: Die Schlüssel seiner Kinder sind kleiner als sein eigener.

Implementierung als Array mit folgender Belegung: Die Kinder von Knoten  $i$  haben die Indizes  $2i + 1$  und  $2i + 2$ . Diese Implementierung eignet sich besonders für HeapSort, da dann keine weiteren Einträge hinzukommen können und auch dieses Sortierverfahren durch Umdefinieren der Array- Einträge ohne zusätzlichen Speicherplatz auskommt.

### 3.9.1 Definition: Array als Heap

Ein Array erfüllt die Heap- Eigenschaft  $\Leftrightarrow$  Jeder Knoten in dem zugehörigen vollen binären Baum besitzt einen größeren Schlüssel als seine Kinder:

$$\forall i : \text{vec}[i].key > \text{vec}[2i + 1].key \wedge \text{vec}[i].key > \text{vec}[2i + 2].key$$

für  $(2i + 1), (2i + 2) > n$  (Länge des Arrays)

Dann gilt:

- Größtes Element steht in  $\text{vec}[0]$
- Entlang eines Weges von der Wurzel zu einem Blatt können die Schlüssel nur (streng) monoton fallen.

# 4 Funktionen auf Datenstrukturen

## 4.1 auf Graphen

### 4.1.1 Bellman- Ford- Algorithmus

Um die Matrix  $U^{(n)}$  herzustellen, verwendet man Folgendes:

$$U^{(1)} = A$$

$$U^{(n+1)} = U^{(n)} \otimes A$$

Dabei ist die Bellman- Multiplikation so definiert:

$$A \otimes B = C \quad \text{mit} \quad c_{ij} = \min_{k=1,\dots,n} [a_{ik} + b_{kj}]$$

```
1 public static double[][] bellman (double[][] a) {
2     int n = a.length; //number of nodes
3     double[][] u = new double[n][n];
4     //initialize u
5     for (int i = 0; i < n; i++) {
6         for (int j = 0; j < n; j++)
7             u[i][j] = a[i][j];
8     }
9     for (int m = 2; m < n; m++) {
10        for (int i = 0; i < n; i++) {
11            double[] tmp = new double[n];
12            //tmp array to calculate min[...] in
13            //Bellman equation
14            for (int j = 0; j < n; j++)
15            {
16                tmp[j] = u[i][j];
17                for (int k = 0; k < n; k++)
18                {
19                    if (tmp[j] > u[i][k] + a[k][j])
20                        tmp[j] = u[i][k] + a[k][j];
21                }
22            } //endfor k
23        } //endfor j
24        //store tmp in u[i]
25        for (int j = 0; j < n; j++)
26        {
27            u[i][j] = tmp[j];
28        }
29    } //endfor i
30 } //endfor m
31 return u;
32 }
```

**Elementarer Zykel** Ein Zykel, in dem jeder beteiligte Knoten genau ein mal besucht wird (außer dem Anfangs- und Endknoten, der wird genau zwei mal besucht).

#### 4.1.2 Ermittlung negativer Zykel

**Satz:** G hat einen negativen Zykel  $\Leftrightarrow$  G hat einen *elementaren* negativen Zykel  $\Leftrightarrow$  Es gibt einen Knoten i mit  $u_{ii}^{(n)} < 0$ .

Somit wird die Existenz negativer Zykel mit einer zusätzlichen Iteration ( $m = n$ ) und Überprüfung der Diagonalelemente festgestellt. Alternativ kann man auch eine Änderung der Matrix während der letzten Iteration überprüfen.

#### 4.1.3 Floyd- Warshall- Algorithmus

**Idee:**  $dist(v, w, X) :=$  Länge eines kürzesten v- w- Weges, der nur Knoten aus  $\{v, w\} \cup X$  besucht.

Es wird definiert:

$$dist(v, w, \emptyset) := a_{vw}$$

$$dist(v, w, \{X\}) := \min \left[ dist(v, w, \emptyset), dist(v, x, \emptyset) + dist(x, w, \emptyset) \right]$$

Ein erlaubter Zwischenstopp oder ohne Zwischenstopp

$$dist(v, w, \{x_1, \dots, x_k\})$$

$$= \min \left[ dist(v, w, \{x_1, \dots, x_{k-1}\}), dist(v, x_k, \{x_1, \dots, x_{k-1}\}) + dist(x_k, w, \{x_1, \dots, x_{k-1}\}) \right]$$

Entweder bekannte Wege oder ein neuer Umweg  $\Rightarrow$  Rückführung auf den Fall mit einem Zwischenstopp weniger.

**Pseudocode** **Input:** Adjazenzmatrix A **Output:** Distanzmatrix Dist

```

for (v = 0, ..., n-1) do
    for (w = 0, ..., n-1) do
         $dist(v, w, \emptyset) := a_{vw}$                                  $\triangleright$  (Initialisieren)
        end for
    end for
    for (i = 1, ..., n-1) do
        for (v = 0, ..., n-1) do
            for (w = 0, ..., n-1) do
                 $dist(v, w, \{0, \dots, i\}) := \min \left[ dist(v, w, \{0, \dots, i-1\}), dist(v, i, \{0, \dots, i-1\}) + dist(i, w, \{0, \dots, i-1\}) \right]$ 
            end for
        end for
    end for

```

```

    end for
end for

```

**Laufzeit:**  $O(n^3)$

**Laufzeit von Bellman- Ford:**  $O(n^3 \log(n))$  bzw.  $O(n^4)$

**Tree- Matrix**  $tree_{ij} :=$  Vorgängerknoten von  $j$  auf dem  $i-j$ -Weg

Initialisierung:  $tree_{ij} = i$ , falls  $(i, j) \in E$ ; -1, sonst

Updates um Bellman- Ford- Algorithmus:

```

1 for (int k = 0; k < n; k++)
2 {
3     if (u[i][j] > u[i][k] + a[k][j])
4     {
5         u[i][j] = u[i][k] + a[k][j];
6         tree[i][j] = k;
7     }
8 }

```

⇒ Für jeden Startknoten  $v$  lässt sich aus  $tree[v]$  ein Kürzester- Wege- Baum basteln.

**Adjazenzlisten** **Idee:** Platz sparen.

**Umsetzung:** Jeder Knoten erhält eine Liste seiner Nachbarn.

**Vorteile:** Schneller bei dünnen Graphen, wenn Algorithmen die Nachbarn eines Knoten oder alle Knoten suchen. Spart Speicher

**Nachteile:** Bei dichten Graphen speicherhungriger durch Overhead; Suche nach einer speziellen Kante im Graphen dauert länger.

#### 4.1.4 Tiefensuche im Graphen

Erzeuge einen Stack, packe Knoten  $s$  darauf.

**while** (Stack ist nicht leer) **do**

**if** (oberster Knoten  $n$  weiß) **then**

$n$  orange färben, seine weißen Nachbarn auf den Stack legen

**else if** ( $n$  orange) **then**

$n$  rot färben, vom Stack entfernen

**else if** ( $n$  rot) **then**

$n$  vom Stack nehmen

**end if**

**end while**

**Mittels Rekursion, ohne einzelnen Stack:**

Färbe alle Knoten weiß

Rufe Tiefensuche für den Startknoten  $s$  auf

*Tiefensuche- Methode für einen Knoten  $v$ :*

Färbe v orange

Rufe für jeden weißen Nachbaren w die Tiefensuche- Methode auf

Färbe v rot

**Laufzeit:** jede Kante wird genau einmal betrachtet

**Speicherbedarf:** pro Kante wird maximal 1 Knoten auf den Stack gelegt

⇒ Die Laufzeit ist linear von der Anzahl der Kanten abhängig.

**Einsatzmöglichkeiten:** Gerichtete Zykel entfernen, alle Knoten durchsuchen, ...

#### 4.1.5 Breitensuche

- wendet Queues an
- traversiert einen Graphen
- Berechnet kürzeste Wege in ungewichteten Graphen

Ideen:

- Beginne bei Startknoten
- Benutze Adjazenzlisten, um neue Knoten zu finden
- Markiere Knoten:
  - Weiß für noch nicht besuchte Knoten
  - Orange für besuchte Knoten, von denen weitergesucht wird
  - Rot für besuchte Knoten, von denen nicht weitergesucht wird
- Benutze zwei Queues, um Knoten der aktuellen Phase und Knoten der nächsten Phase zu merken
- Nummeriere der Knoten nach der Phase, in der sie bearbeitet werden (→ Distanz)

Basisalgorithmus mit Startknoten s:

- Färbe alle Knoten weiß
- Erzeuge zwei leere Queues
- Füge s in die zweite Queue ein

**while** zweite Queue nicht leer **do**

Verschiebe die Knoten der zweiten Queue in die erste Queue

Nächste Phase beginnt

**while** erste Queue nicht leer **do**

Betrachte den ersten Knoten v der ersten Queue

Färbe den Knoten rot und gib ihm die aktuelle Phasennummer

Weisse Nachbarn werden orange und zur zweiten Queue hinzugefügt

**end while**

**end while**

## 4.2 Suchalgorithmen auf linearen Datenstrukturen

### 4.2.1 Sequentielle Suche

Gegeben: Feld a, Wert x

Gesucht: Index k mit  $a[k] = x$ ; Falls  $\neq -1$

Durchlaufe Feld von  $a[0]$  bis  $a[n-1]$  der Reihe nach.

Falls  $a[k] = x$ , gib k zurück

Ist  $a[n-1] \neq x$ , gib -1 zurück.

Aufwand: Im worst Case:  $n-1$

### 4.2.2 Binäre Suche

Feld ist sortiert, d.h.  $a[0] < a[1] < \dots < a[n-1]$ .

Finde Index k mit  $a[k] = x$ . Sonst gib -1 zurück.

Wähle mittleren Index k und vergleiche x und  $a[k]$ .

Ist  $a[k] = x$ , gib k zurück.

Ist  $x < a[k]$ , so suche im Teilstück  $a[0] \dots a[k-1]$ .

Ist  $a[k] < x$ , suche im Teilstück  $a[k+1] \dots a[n-1]$ .

Verfahren so weiter, bis x gefunden. Ist der noch zu durchsuchende Teil leer, gib 1-aus.

Aufwand:  $C(n) \leq \lfloor \log n \rfloor + 1$

## 4.3 Sortieralgorithmen

### 4.3.1 Bubble Sort

```
for (i := 1 to n -1) do
    for (j := 1 to n-1) do
        if (a[j].key > a[j+1].key) then
            vertausche (a[j], a[j+1])
        end if
    end for
end for
return a
```

### 4.3.2 SelectionSort

```
for (i := 1 to n-1) do
    min := i
    for (j := i+1 to n) do
        if (a[j].key < a[min].key) then
```

```

        min := j
    end if
end for
vertausche (a[i], a[min])
end for
return a

```

#### 4.3.3 InsertionSort

```

for (i := 1 to n-1) do
    temp := a[i]
    j:= i-1
    while (j > 0 ∧ a[j].key > temp.key) do
        vertausche (a[j], a[j+1])
        j := j-1
    end while
end for
return a

```

#### 4.3.4 MergeSort

```

if (n = 1) then
    return a
end if
a1 := firstHalf (a)
a2 := secondHalf (a)
a1 := MergeSort (a1)
a2 := MergeSort (a2)
a := merge (a1, a2)
return a

```

#### 4.3.5 QuickSort

```

if (n = 1) then
    return a
end if
p := a[n/2]
a1 := QuickSort({a[i] | a[i] ≤ p})
a2 := QuickSort({a[i] | a[i] ≥ p})
a := a1, a2
return a

```

Anmerkung: im Average Case  $O(n \log n)$ ,  
im Worst Case (Pivotelement stets das Maximum oder Minimum)  $O(n^2)$ .  
Praktisch der schnellste.

#### 4.3.6 HeapSort

##### Algorithmus zur Herstellung der Heap- Eigenschaft

**Input:** (Item[] vec, int top, int bottom)

**Output:** Array Item[] vec, das die Heap- Eigenschaft erfüllt

Setze voraus, dass Heap- Eigenschaft im Teilarray von  $top + 1$  bis  $bottom$  bereits erfüllt ist.

Stelle dann sicher, dass Heap- Eigenschaft erfüllt ist von  $top$  bis  $bottom$ .

1) Ermittle das größere der beiden Kinder von  $top$  (falls Kinder im Teilarray existieren), nenne es child.

2) Vergleiche vec[top].key mit vec[child.key]

IF (vec[top].key > vec[child])

→ Fertig! (RETURN)

ELSE

→ Vertausche Komponenten von  $top$  und  $child$

3) Wende heapify rekursiv auf den Bereich von  $child$  bis  $bottom$  an.

Die Heapeigenschaft eines Arrays kann durch wiederholtes Aufrufen dieser Methode mit einer Laufzeit in  $O(n)$  hergestellt werden.

**Der eigentliche Sortiervorgang** Tausche im ersten Durchlauf das größte Element an die letzte Stelle des Arrays, im zweiten Schritt das zweitgrößte an die vorletzte, usw.

**Aufwand:**  $O(n) + n$  Aufrufe von heapify für Element an der Stelle  $0 \leq O(n) + n \cdot O(h) \in O(n \log n)$

#### 4.3.7 Untere Komplexitätsschranken für das Sortieren

Es wurde gezeigt, dass jedes Sortierverfahren, das auf paarweisen Vergleichen beruht, im worst case wie auch im average case mindestens  $\Omega(n \log n)$  Vergleiche benötigt.

Beweisidee: Im Entscheidungsbaum, den ein solcher Algorithmus zwangsläufig abläuft, sind die tiefsten Blätter sowie der Höhendurchschnitt der  $n!$  Blätter in einer Höhe von  $\Omega(n \log n)$  zu finden.

### 4.3.8 BucketSort

Ziel ist keine lineare Auflistung, sondern das Einsortieren der Einträge in der Eingabe (Liste, Array, etc.) in "Kategorien" (Listen, Queues), sodass in manchen Kategorien mehrere Einträge auftreten können, in anderen gar keine. Als Veranschaulichung: Ein Briefträger sortiert die Briefe einer Straße nach Hausnummern.

**Input:**  $n$  Objekte  $a_1, \dots, a_n$  (Liste mit Schlüsseln), wobei die Schlüssel  $s(a_1), \dots, s(a_n) \in \{0, \dots, m - 1\}$

**Output:** Eine Liste, die an Stelle  $i$  alle Objekte mit dem Schlüssel  $i$  in einer gesonderten Liste enthält.

**Aufwand:**  $O(n + m)$  Insbesondere, falls  $m \leq n$ ,  $m \in O(n) \Rightarrow$  Aufwand liegt in  $O(n)$ .

## 4.4 Hashing

- gestreute Speicherung
- Alternative zu Suchbäumen
- Unterstützt
  - Suchen
  - Einfügen
  - Löschen (partiell)
- Vorteil: Aufwand pro Operation im Mittel  $O(1)$
- Nachteil: Im Worst Case Aufwand  $O(n)$

### 4.4.1 Grundidee

- Reserviere Hashtabelle mit  $m$  Plätzen im Speicher, in der die abgespeicherten Daten stehen (ähnlich einem Array).
- Gegeben: Datensätze mit Schlüssel, die aus Universum  $U$  stammen.
- Man kennt aber a priori nicht die Schlüssel der  $n$  Datensätze. Ganz  $U$  kommt in Frage
- $n := \#$  Datensätze in Teilmenge  $<< |U|$
- Die Hashfunktion berechnet die Speicheradresse aus dem Schlüssel:  
$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$
- Hier: Schlüssel  $k$  sind ganze Zahlen

## Zur Umsetzung

- Einfachste Lösung („direct addressing“):  $h$  bijektive Abbildung.  
Belegt aber viel mehr Speicher als nötig, falls  $U$  deutlich größer als die interessante Teilmenge.
- Ziel: Größe der Hashtabelle soll ungefähr Anzahl der Datensätze entsprechen.
- Es gilt i.A.:  $h$  nicht injektiv, da  $|U| > m$
- i.A.  $\exists k_1 \neq k_2$  mit  $h(k_1) = h(k_2)$ , d.h. Kollision
- Kollision tritt fast sicher auf, kaum zu vermeiden. Hashverfahren muss mit Kollisionen umgehen können

### 4.4.2 Kriterium für eine gute Hash- Funktion

$h$  streut gleichmäßig über den Adressraum: Jede Adresse wird mit der gleichen Wahrscheinlichkeit  $\frac{1}{m}$  gewählt („uniform hashing“).

In der Praxis nur angenähert zu realisieren. Nutze diese Annahme für die Analyse:

$$\forall j \quad \underbrace{\sum_{k \in U : h(k)=j} p(k)}_{\text{Wahrscheinlichkeit, dass Speicherstelle } j \text{ getroffen wird.}} = \frac{1}{m}$$

Mit  $p(k)$  = Wahrscheinlichkeit für Auftreten von Schlüssel  $k$  in einer Iteration.

### 4.4.3 Methoden zum Erzeugen einer Hashfunktion

#### Divisionsmethode

$$h(k) = k \mod m$$

Dadurch wird Projektion von  $\mathbb{N}$  auf  $\{0, \dots, m-1\}$  erreicht.

**Beispiel:**  $m = 2^q$ ,  $k$  Binärzahl.  $h$  schneidet dann die vordersten Bits ab und behält die  $q$  kleinsten Bits.

Kann u.U. zu Häufung von Kollisionen führen.

**Empirisch gute Wahl für  $m$ :** Primzahl weit entfernt von 2er- Potenzen

**Multiplikationsmethode** Wähle festes  $0 < A < 1$  und definiere:

$$h(k) = \lfloor m \cdot \underbrace{(k \cdot A - \lfloor k \cdot A \rfloor)}_{\in [0,1)} \underbrace{\}_{\in [0,m)} \in \{0,1,\dots,m-1\}$$

**Beispiel:**  $m = 2^q$  Empirisch gute Wahl:  $A = \frac{\sqrt{5}-1}{2} \approx 0,618\dots$

#### 4.4.4 Umgang mit Kollisionen

##### Hashing mit Überlauf („chaining“)

**Idee** Lege pro Platz in der Hashtabelle eine Liste an. Speichere alle Datensätze  $k$  mit  $h(k) = j$  in der Liste zum Speicherplatz  $j$  ab.

**Aufwand für das Einfügen** (am Beginn der Liste):  $O(1)$

**Aufwand für das Suchen** (im Mittel) Annahme:

- Hashfunktion streut gut (uniform hashing)
  - Es sind  $n$  Datensätze in der Hash-Tabelle
- ⇒ Im Mittel sind  $\frac{n}{m}$  Datensätze pro Liste vorhanden.
- ⇒ Erfolgsuche erfordert im Mittel Aufwand  $O(1 + \frac{n}{m})$  = Berechnung von  $h(k)$  + Durchgehen der Liste
- $\alpha = \frac{n}{m}$  Auslastung der Hashtabelle
- Wähle typischerweise  $m \in \Theta(n)$
- Aufwand  $O(1 + \alpha) = O(1)$  mit  $\alpha$  Konstante

**Aufwand für erfolgreiche Suche** Beachte: Die Liste  $h(k)$  enthält mindestens ein Element (nämlich Element mit Schlüssel  $k$ ). Daher ist die mittlere Länge der Liste an der Stelle  $h(k)$  durch  $1 + \frac{n-1}{m}$  beschränkt. ⇒ Aufwand:  $O(1 + 1 + \alpha)$

##### Aufwand für die 3 Operationen

Einfügen  $O(1)$

Suchen  $O(1 + \alpha)$  im Mittel

Löschen  $O(1 + \alpha)$

Alle Operationen erfordern im Mittel konstanten Aufwand.

**Aber:** Worst-Case- Aufwand für Suchen ist  $\Omega(n)$  (alle  $n$  Schlüssel in derselben Liste)

**Generell:** Tradeoff zwischen Größe der Hash-Tabell (Speicher) und Anzahl Vergleich beim Suchen (Zeit).

##### Hashing mit Ersatzadresse („open addressing“)

## Idee

- Speichere alle Datensätze in der Hashtabelle (d.h. keine Überlauf- Listen)
- $\Rightarrow m \geq n$
- Bei Kollision: suche Ersatzadresse

$$h : \underset{\text{Schlüssel}}{U} \times \underset{\substack{\{0, 1, \dots, m-1\} \\ \# \text{ erfolgreicher Versuche bei Suche nach freiem Eintrag}}}{\{0, 1, \dots, m-1\}} \rightarrow \underset{\text{Adresse}}{\{0, 1, \dots, m-1\}}$$

$h(k, i) \hat{=} (i + 1)\text{-ter Versuch, eine freie Adresse für } k \text{ zu finden.}$

Dieses Verfahren berechnet nacheinander die Adressen  $h(k, 0), h(k, 1), \dots, h(k, i), \dots$  bis eine freie Adresse gefunden wird. Damit auf diese Art immer eine freie Adresse gefunden wird (unter der Annahme, dass  $n \leq m$ ), muss gelten:

### Permutationsbedingung

$h(k, 0), h(k, 1), \dots, h(k, m-1)$  ist Permutation von  $0, 1, \dots, m-1$

**Achtung:** Dies erschwert das Löschen!

### Lineares Sondieren

$$h(k, i) := (h'(k) + i) \mod m$$

- $h'(k) = h(k, 0)$  ist gewöhnliche Hashfunktion.
- Permutationsbedingung offensichtlich erfüllt.
- Nachteil: Primäres Clustering, d.h. es bilden sich immer größere Blöcke konsekutiv belegter Adressen, denn:

Falls die Adressen  $a, a+1, a+2, \dots, a+i-1$  bereits belegt sind und als nächstes ein Datenelement mit zufälligem Schlüssel eingefügt wird, so landet es mit Wahrscheinlichkeit  $\frac{i+1}{m}$  an der Stelle  $a + i$ .

### Quadratisches Sondieren:

$$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$$

- Sprungweite nimmt quadratisch zu
- Beachte: Permutationsbedingung hängt von  $m, c_1, c_2$  ab und muss im Einzelfall überprüft werden.
- Vorteil: Kein primäres Clustering
- Nachteil: Sekundäres Clustering:

Schlüssel, die auf dieselbe erste Adresse abgebildet werden, haben auch dieselbe Folge von Ersatzadressen.

## Doppel- Hash

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

- $h_1, h_2$  herkömmliche Hashfunktion
- Vermeidet primäres und sekundäres Clustering,  
d.h.  $h_1(k) = h_1(k')$  führt zu unterschiedlichen Ersatzadressen, falls  $h_2(k) \neq h_2(k')$

### Zur Erfüllung der Permutationsbedingung:

Doppelhash erfüllt die Permutationsbedingung  $\Leftrightarrow \forall k$  sind  $m$  und  $h_2(k)$  relativ prim, d.h.  $ggT(m, h_2(k)) = 1$

- Immer erfüllt, wenn  $m$  Primzahl
- z.B. auch erfüllt, wenn  $m = 2^q$  und  $h_2(k)$  ungerade  $\forall k$

#### Beweis:

$\Rightarrow$  : Annahme:  $\exists k$ , so dass  $ggT(m, h_2(k)) = d > 1$

$$\Rightarrow m = p \cdot d, h_2(k) = q \cdot d \Rightarrow p \cdot h_2(k) = q \cdot p \cdot d = q \cdot m \equiv 0 \mod m$$

D.h. die  $p$ -te Ersatzadresse für  $k$  stimmt überein mit  $h_1(k)$ . Beachte, dass  $p < m \Rightarrow$  nicht alle Adressen besucht.  $\Rightarrow$  Permutationsbedingung verletzt

#### Widerspruch!

$\Leftarrow$ : Annahme: Permutationsbedingung nicht erfüllt, d.h.

$\exists k$  und  $0 \leq i < j \leq m - 1$  mit

$$h_1(k) + i \cdot h_2(k) \equiv h_1(k) + j \cdot h_2(k) \mod m$$

$$\Leftrightarrow (j - i) \cdot h_2(k) \equiv 0 \mod m$$

$$\Leftrightarrow (j - i) \cdot h_2(k) = q \cdot m, \quad q \in \mathbb{N}$$

$$\Leftrightarrow h_2(k) = \frac{q \cdot m}{j - i}$$

Da  $j - i < m$  gilt also  $ggT(h_2(k), m) > 1$  **Widerspruch!**

□

**Analyse des open addressing** Einfügen: Wie viele Sondierungen sind nötig, bis freie Adresse gefunden? Benutze dazu die Gleichverteilungsannahme, d.h. nächste Sondierung wählt immer gleichverteilt aus  $\{0, \dots, m - 1\}$ , jede Adresse wird mit Wahrscheinlichkeit  $\frac{1}{m}$  gewählt.

Bei Auslastung  $\alpha = \frac{n}{m} < 1$  ist die erwartete Anzahl Sondierungen beim Einfügen höchstens  $\frac{1}{1-\alpha}$ .

## Beweis:

### Urnenmodell:

$m$  Kugeln,  $w$  weiße,  $s$  schwarze ( $m = w + s$ ) ziehe gleichverteilt mit Zurücklegen.

$$E(\# \text{ Ziehungen bis weiße Kugel gezogen}) = \frac{m}{w}$$

Kugel ist weiß mit Wahrscheinlichkeit  $\frac{w}{m} =: p$ , schwarz mit Wahrscheinlichkeit  $\frac{s}{m} = 1 - p =: q$ .  $X = \# \text{ Ziehungen, bis weiße Kugel gezogen}$ .

Werte von $X$		Wahrscheinlichkeit dafür
1	w	$p_1 = p$
2	sw	$p_2 = q \cdot p$
3	ssw	$p_3 = q^2 \cdot p$
$\vdots$	$\vdots$	$\vdots$
$i$	$\underbrace{\text{S...S}}_{i-1} \text{ w}$	$p_i = q^{i-1} \cdot p$

$$\sum_{i=1}^{\infty} p_i = \sum_{i=1}^{\infty} q^{i-1} \cdot p = p \cdot \sum_{i=0}^{\infty} q^i = p \cdot \frac{1}{1-q} = \frac{p}{p} = 1$$

$$E(X) = \sum_{i=1}^{\infty} i \cdot q^{i-1} \cdot p = p \cdot \underbrace{\sum_{i=1}^{\infty} i \cdot q^{i-1}}_{=:s}$$

$$\begin{aligned} S &= 1 \cdot q^0 + 2 \cdot q^1 + 3 \cdot q^2 + 4 \cdot q^3 + \dots \\ &= q^0 + q^1 + q^2 + q^3 + \dots \quad \{1 \cdot \sum q^i\} \\ &\quad + q^1 + q^2 + q^3 + \dots \quad \{q \cdot \sum q^i\} \\ &\quad + q^2 + q^3 + \dots \quad \{q^2 \cdot \sum q^i\} \\ &\quad + q^3 + \dots \quad \{q^3 \cdot \sum q^i\} \end{aligned}$$

$$= \left( \sum_{i=0}^{\infty} q^i \right) \cdot \left( \sum_{i=0}^{\infty} q^i \right) = \frac{1}{p} \cdot \frac{1}{p} = \frac{1}{p^2}$$

$$\Rightarrow E(X) = \frac{1}{p} = \frac{m}{w}$$

**Anwendung auf open addressing:** Sondierungen  $\hat{=} \text{ Ziehen aus Urne}$

$$E(\# \text{ Ziehungen bis freier Platz gefunden}) = \frac{m}{m-n} = \frac{1}{1-\frac{n}{m}} = \frac{1}{1-\alpha}$$

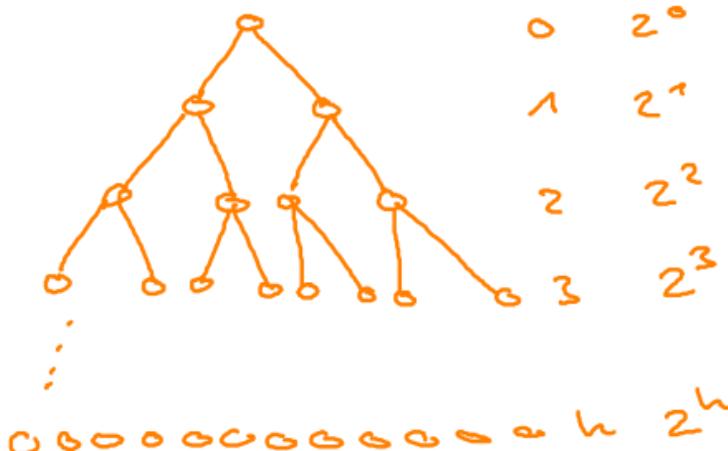
$\Rightarrow$  Satz □

- Einfügen: im Mittel konstant (falls  $\alpha < 1$  konstant)
  - Suchen: dieselbe Folge von Sondierungen wie Einfügen (sofern in der Einfügefolge nicht bereits Elemente gelöscht wurden)
  - Löschen bereitet daher Schwierigkeiten!

Daher ist open addressing meistens beschränkt auf Hashing ohne Löschen.

**Zusammenfassung** Aufwand pro Operation im Mittel  $O(1)$ , aber im Worst Case  $\Omega(n)$

Vergleich zu Suchbäumen: im Worst Case  $O(\log n)$ , mittlerer Aufwand  $\Omega(\log n)$ , denn:



Beobachtung: Mindestens die Hälfte aller Knoten liegt in Tiefe  $\geq \log n$   
 $\Rightarrow$  im Average Case ist der Aufwand für Suchen  $\Omega(\log n)$

**Universelles Hashing** Idee: Vermeide Worst Case. Stelle bösen Gegenspieler vor, der Hashing-Verfahren und durch Auswahl spezieller Schlüssel in den Worst Case treibt.  $\Rightarrow$  Randomisiere zur Laufzeit des Verfahrens, d.h. wähle eine zufällige Hash-Funktion. Diese wird dann für die gesamte Dauer des Verfahrens verwendet.

## **Definition:**

Eine Klasse  $H$  von Hash- Funktionen heißt universell bezüglich der Größe  $m$  der Hash- Tabelle, falls  $\forall$  Schlüsselpaare  $x, y \in U$  mit  $x \neq y$  gilt:

$$|\{h \in H | h(x) = h(y)\}| \leq \frac{|H|}{m}$$

**Idee:** Wähle  $h \in H$  zufällig und gleichverteilt, dann ist die Wahrscheinlichkeit für eine Kollision von  $x$  und  $y \leq \frac{1}{m}$ .

**Satz:**  $H$  sei universell für  $m, h \in H$  zufällig und gleichverteilt gewählt. Werden nun  $n < m$  Schlüssel mit  $h$  eingefügt und ist  $x$  ein fester Schlüssel, so gilt:

$$E(\# \text{ Kollisionen mit } x) < 1$$

**Beweis:**

Für Schlüsselpaar  $y \neq z$  sei

$$C_{yz} = \begin{cases} 1; & \text{falls } h(y) = h(z) \\ 0; & \text{sonst} \end{cases} \quad \begin{array}{c} \text{Wahrscheinlichkeit} \\ \dots \end{array} \quad \begin{array}{l} \leq \frac{1}{m} \\ \geq \frac{m-1}{m} \end{array}$$

$$E(C_{yz}) \leq \frac{1}{m}$$

Sei  $C_x = \#$  Kollisionen mit  $x$ , d.h.  $C_x = \sum \lim_{y \neq x} C_{yx}$

$$E(C_x) = E\left(\sum \lim_{y \neq x} C_{yx}\right) = \sum \lim_{y \neq x} E(C_{yx}) \leq \frac{n-1}{m} < 1$$

□

Angewendet auf Hashing mit chaining: Mittlere Listenlänge  $< 2$ . Worst Case tritt im Erwartungswert nicht auf, Absicherung gegen bösen Gegenspieler klappt.

Zeige, dass  $\exists$  universelle Klasse von Hashfunktionen für beliebiges  $m$ :

Wähle dazu zunächst Primzahl  $p$  mit  $p > x$  für alle  $x \in U$ . Für beliebiges  $a \in \mathbb{Z}_p^* = \{1, \dots, p-1\}$  und  $b \in \mathbb{Z}_p = \{0, 1, \dots, p-1\}$  sei die Hash- Funktion  $h_{a,b}$  wie folgt definiert:

$$h_{a,b}(x) = ((a \cdot x + b) \mod m)$$

Es sei  $H = \{h_{a,b} | a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$

**Satz:**  $H$  ist unversell bezüglich  $m$ .

**Beweis:**

Betrachte zwei Schlüssel  $x \neq y$  aus  $\{0, \dots, p-1\}$  und  $h_{a,b} \in H$ . Definiere

$$r := a \cdot x + b \mod p$$

$$s := a \cdot y + b \mod p$$

Behauptung:  $x \neq y \Rightarrow r \neq s$

Beweis: Annahme:  $r = s$

$$\Rightarrow a \cdot x + b \equiv a \cdot y + b \mod p$$

$$\stackrel{a \in \mathbb{Z}_p^*}{\Rightarrow} x \equiv y \mod p \Rightarrow x = y \text{ Widerspruch!} \blacksquare$$

Behauptung:  $a$  und  $b$  sind eindeutig durch  $x, y, r, s$  bestimmt.

Beweis: Aus der Definition folgt:

$$(r - s) \equiv a \cot \underbrace{(x - y)}_{\in \mathbb{Z}_p^*} \mod p$$

$$\Rightarrow a = (x - y)^{-1} \cdot (r - s) \in \mathbb{Z}_p$$

$$r = a \cdot x + b \Rightarrow b = r - (x - y)^{-1} \cdot (r - s) \cdot x \blacksquare$$

Es gibt  $(p-1) \cdot p$  Paare  $r, s \in \{0, \dots, p-1\}$  mit  $r \neq s$ .

$\Rightarrow$  Es gibt Bijektion zwischen möglichen Paaren  $(a, b)$  und  $(r, s)$ .

$\Rightarrow$  Wählt man eine zufällige Hash-Funktion aus  $H$ , also ein zufälliges Paar  $(a, b)$ , dann erhält man ein zufälliges Paar  $(r, s)$ .

Es gilt:

$$\Pr(h_{a,b}(x) = h_{a,b}(y)) = \Pr(r \equiv s \mod m)$$

Für festes  $r$  gibt es  $\leq \lceil \frac{p}{m} \rceil - 1$  mögliche  $s \neq r$  mit  $r \equiv s \mod m$ .

Es gilt:

$$\lceil \frac{p}{m} \rceil - 1 \leq \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$$

Daraus folgt: Die Wahrscheinlichkeit dafür, dass ein zufälliges  $s \in \{0, \dots, p-1\} \setminus \{r\}$  zu einer Kollision führt ist  $\leq \frac{1}{m}$ .

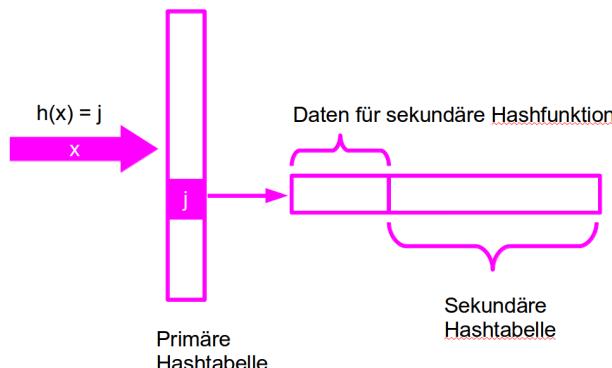
$\Rightarrow H$  ist universell bezüglich  $m$ .  $\square$

## Perfektes Hashing:

- Nehme an, dass Schlüsselmenge statisch und a priori bekannt ist, z.B.
  - reservierte Worte in Java (Compiler verwendet Hashing)
  - Namen aller Dateien auf DVD
- Ziel: Verbessere den Worst- Case- Aufwand für die Suche (im Mittel Aufwand  $O(1)$ )
- Ein Hashing- Verfahren heißt perfekt,  $\Leftrightarrow$  Worst- Case- Aufwand für das Suchen ist  $O(1)$ .

## Umsetzung

- Mithilfe eines zweistufigen Verfahrens:



- Daten für sekundäre Hashfunktion  $h_j$  und Länge der sekundären Hash-Tabelle werden in der sekundären Hash- Tabelle zu Beginn gespeichert.
- Die sekundäre Hash- Tabelle hat die Länge  $m_j$ , dieses hängt ab von  $h_j := \#$  Schlüssel mit  $h(x) = j$ , nämlich  $m_j = h_j^2$
- Bemerkung:  $h_j$  kann man vorab (nach Wahl von  $h$ ) bestimmen, da alle Schlüssel bekannt.
- Die Primäre Hashfunktion  $h$  und sekundäre Hashfunktionen  $h_j, j = 1, \dots, m$  sind aus universeller Klasse gewählt:

$$h(x) = ((a \cdot x + b) \mod p) \mod m$$

$$h_j(x) = ((a_j \cdot x + b_j) \mod p) \mod m_j$$

$$- h(x) \in H_{p,m}$$

$$- h_j(x) \in H_{p,m_j}$$

$$- p \text{ Primzahl} > x \quad \forall \text{ Schlüssel } x$$

- $m$  Länge primärer Hashtabelle
- $m_j = h_j^2$
- Spezialfall: Falls  $h_j = 1$ , wähle  $a_j = b_j = 0, m_j = h_j^2 = 1$
- Zwei mögliche Probleme
  - A Wie stellt man sicher, dass beim sekundären Hashing keine Kollision auftritt?
  - B Kann der benötigte Speicherplatz sinnvoll beschränkt werden?

Zu A: **Satz:** Eigenschaft der sekundären Hashfunktion mit  $n = n_j, m = m_j$

$n$  Schlüssel: werden mit zufällig gewählter Hashfunktion  $h \in H_{p,m}$  in Hashtabelle der Länge  $m = n^2$  gespeichert. Dann ist die Wahrscheinlichkeit, dass eine Kollision auftritt, klein, d.h.  $< \frac{1}{2}$ .

Das heißt, dass bei wiederholter zufälliger Wahl von  $h \in H_{p,m}$  **mit großer Wahrscheinlichkeit nach wenigen Wiederholungen** eine kollisionsfreie Hashfunktion gefunden wird.

**Beweis:**

$\exists$  insgesamt  $\binom{n}{2}$  mögliche Kollisionen.

$H_{p,m}$  ist universell  $\Rightarrow x \neq y$  kollidieren mit Wahrscheinlichkeit  $\leq \frac{1}{m}$ .

Sei  $X = \#$  Kollisionen, dann gilt:

$$E(X) = E\left(\sum_{x \neq y} C_{x,y}\right) = \sum_{x \neq y} E(C_{x,y}) \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$$

Markov'sche Ungleichung:

Für eine Zufallsgröße  $X \geq 0$  und  $t > 0$  gilt:

$$P_r(X \geq t) \leq \frac{E(X)}{t}$$

In unserem Fall:

$$P_r(X \geq 1) \leq \frac{E(X)}{1} < \frac{1}{2}$$

Problem A gelöst: □

**Satz:** Die Wahrscheinlichkeit für das Auftreten eines Konfliktes in einer sekundären Hashtabelle ist bei zufälliger Wahl von  $a_j$  und  $b_j$  kleiner als  $\frac{1}{2}$ .  
 $\Rightarrow \exists a_j, b_j$  mit konfliktfreier sekundärer Hashfunktion.

**Satz:** Die Wahrscheinlichkeit für das Auftreten eines Konfliktes in einer sekundären Hashtabelle ist bei zufälliger Wahl von  $a_j$  und  $b_j$  kleiner als  $\frac{1}{2}$ .  
 $\Rightarrow \exists a_j, b_j$  mit konfliktfreier sekundärer Hashfunktion.

Zu B: **Satz:**  $n$  Schlüssel werden mit zufälliger Hashfunktion  $h \in H_{p,m}$  in eine Hashtabelle der Größe  $m = n$  gespeichert. Dann gilt:  $E\left(\sum_{j=0}^{m-1} n_j^2\right) < 2 \cdot n$  wobei  $m_j := |\{x | h(x) = j\}|$

**Beweis:**

Für  $a \in \mathbb{N}$  gilt:  $a^2 = a + 2\binom{a}{2}$  Unter Betrachtung der Linearität des Erwartungswertes gilt:

$$\begin{aligned} E\left(\sum_{j=0}^{m-1} n_j^2\right) &= E\left(\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right) = \\ E\left(\underbrace{\sum_{j=0}^{m-1} n_j}_n\right) &+ 2 \cdot E\left(\sum_{j=0}^{m-1} \underbrace{\binom{n_j}{2}}_{\substack{\# \text{Kollisionen an Stelle } j \\ \# \text{Kollisionen, insgesamt}}}\right) = \\ n + 2 \cdot E\left(\sum_{j=0}^{m-1} \underbrace{\binom{n_j}{2}}_{\substack{\# \text{Kollisionen an Stelle } j \\ \# \text{Kollisionen, insgesamt}}}\right) &\leq n + 2 \cdot \binom{n}{2} \cdot \frac{1}{m} \\ = n + (n - 1) &= n + n - 1 = 2n - 1 < 2n \end{aligned}$$

□

**Korollar:** Für das zweistufige Hashverfahren mit primärer Länge  $m = n$  und sekundärer Länge  $m_j = n_j^2$  gilt: Der erwartete Speicherbedarf ist linear in  $n$ , d.h.  $O(n)$ .

**Satz:** Für das zweistufige Hash- Verfahren gilt:

$$P_r(\text{Speicherbedarf für sekundäre Hashtabelle} \geq 4n) < \frac{1}{2}$$

$\Rightarrow$  Probiere so lange zufällige primäre Hash- Funktion aus, bis Speicher  $< 4n$  (Beweis: Markov'sche Ungleichung) □

$\Rightarrow$  Durch wiederholtes Ziehen einer zufälligen primären Hash- Funktion  $h$  kann man garantieren, dass der Speicherbedarf  $\leq 4n$  ist.

# 5 Berechenbarkeit und Komplexität

Grundfrage dieses Kapitels: Welche Probleme lassen sich mit Computern lösen? Welche Probleme lassen sich effizient lösen? Dazu müssen aber diese beiden Fragen geklärt sein:

- Was ist ein Computer?
- Was ist ein Problem?

## 5.1 Computer

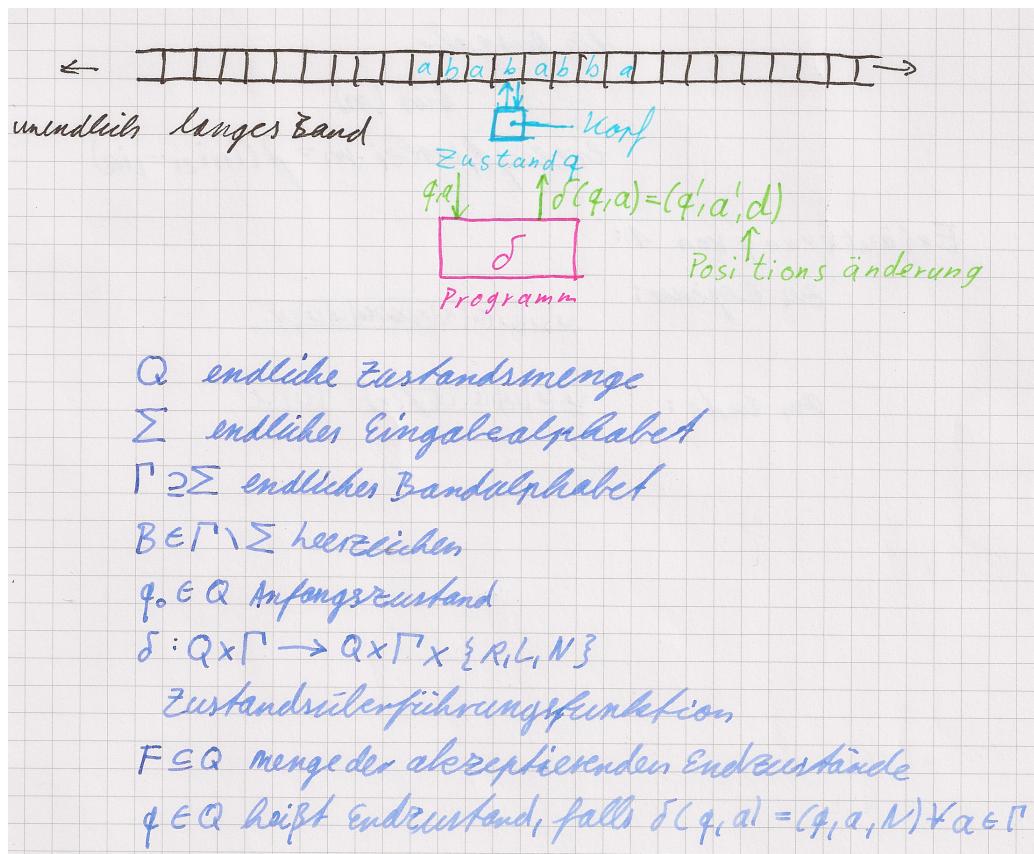
### 5.1.1 Grundidee

Eingabe → Computer → Ausgabe  
 $\in \Sigma^*$        $\in \Sigma^*$

$\Sigma$  - endliches Alphabet, z.B.  $\{0, 1\}$

$\Sigma^* := \bigcup_{i=1}^{\infty} \Sigma^i$  - Menge der Strings endlicher Länge über  $\Sigma$

### 5.1.2 Turing- Maschine



**Bandalphabet**  $\Gamma = \Sigma \cup \{B\} \cup \{\dots\}$ , wie Alphabet  $\Sigma$  endlich.

**Programm**  $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, N\}$ : Eine eindeutige Handlungsanweisung, die ausgehend vom aktuellen Zustand der Maschine und dem gelesenen Zeichen der Maschine einen neuen Zustand, ein zu schreibendes Zeichen und eine Bewegungsanweisung vorgibt. Bewegungsanweisung kann ein Feld nach rechts oder links oder auch stehen bleiben bedeuten.

**Deterministische Turing- Maschine (DTM)** Jeder Schritt wird nur durch Eingabe und Programm vorherbestimmt, es gibt keine zufälligen Schritte.

**Nicht- Terministische Turing- Maschine** Hier können Schritte zufallsgesteuert ablaufen, man kann „auf einen Zufallsgenerator zugreifen“.

**Konfiguration einer DTM** Besteht aus  $\alpha, q, \beta$  mit  $\alpha, \beta \in \Sigma^*, q \in Q$ .  $\alpha$  die Symbole auf dem Band links vom Kopf,  $\beta$  die restlichen, beginnend unter dem Kopf bis zum rechten Ende des Bereichs.  $q$  ist der Zustand der DTM.

**Direkte Nachfolgekonfiguration** von  $\alpha, q, \beta$  ist  $\alpha', q', \beta'$ , falls man in einem Schritt von  $\alpha, q, \beta$  zu  $\alpha', q', \beta'$  kommt.  $\alpha, q, \beta \vdash \alpha', q', \beta'$

**Nachfolgekonfiguration** von  $\alpha, q, \beta$  ist  $\alpha'', q'', \beta''$ , falls man in endlich vielen Schritten von  $\alpha, q, \beta$  zu  $\alpha'', q'', \beta''$  kommt.  $\alpha, q, \beta \vdash^* \alpha'', q'', \beta''$

**Mehrspur- Turing- Maschinen** Modelliere  $k$  Spuren durch Ersetzen von  $\Gamma$  durch  $\Gamma^k$ .

**Mehrband- Turing- Maschinen** Allgemein:  $k$  Bänder,  $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, N\}^k$

**Satz:** Eine  $k$ - Band- TM  $M$ , die mit Rechenzeit  $t(n)$  und Platz  $s(n)$  auskommt, kann von einer TM  $M'$  mit Rechenzeit  $O(t(n)^2)$  und Platz  $O(s(n))$  emuliert werden.

**Beweisidee:**

Benutze Mehr- Spur- TM  $M'$  mit  $2k + 1$  Spuren

→ Fahre Band ab, markiere einzelne Kopf-Positionen (Sprung)

→ Fahre dann nochmal ab und führe Mehrband- Programm aus

Größe des Bereichs  $\in O(t(n))$

**Universelle Turing- Maschine** Betrachte Turing- Maschine  $M$  mit (o.B.d.A.):

$$\Sigma = \{0, 1\} \text{ und } \Gamma = \{0, 1, B\} = \{x_1, x_2, x_3\}$$

$$D_1 = L; D_2 = R; D_3 = N$$

$Q = \{q_1, q_2, \dots, q_t\}; F = \{q_2\}$  Kodiere Programm  $\delta$  von  $M$  wie folgt:

$$\delta(q_i, x_j) = (q_n, x_l, D_M) \quad z = 1, \dots, s$$

$$0^i 10^j 10^l 10^m = \text{Code}(z)$$

**Definition Gödelnummer:** Die Gödel- Nummer der Turing- Maschine  $M$  ist:

$$111\text{Code}(1)11\text{Code}(2)11\dots 11\text{Code}(s)111 =: < M >$$

**Definition Universelle Turing- Maschine:** Eine universelle Turing- Maschine  $U$  erwartet als Eingabe eine Gödelnummer und ein Wort  $W$  als Paar: ( $< M >, W$ ), wobei  $M$  eine TM und  $W \in \{0, 1\}^*$ .  $U$  simuliert  $M$  auf der Eingabe  $W$ .

### Realisierung von $U$ als Drei- Band- TM

- 1 Teste Eingabe auf Korrektheit:  $i, k \geq 1; j, l, m \in \{1, 2, 3\}$ 
  - Keine zwei Codes dürfen mit demselben Präfix  $0^i 10^j$
  - ...
- 2 Kopiere  $< M >$  auf Band 2 und überschreibe sie auf Band 1 mit Leerzeichen
- 3 Schreibe Zustand  $q_i$  von  $M$  auf Band 3, kodiert als  $0^i$ .
- 4 In jedem Schritt:
  - Lese  $x_j$  auf Band 1, suche auf Band 2 den zugehörigen String  $110^i 10^j 1\dots$ , wobei  $0^i$  auf Band 3 steht.
  - Akzeptiere, falls  $i = 2$

Andernfalls:

  - Lese  $0^k 10^l 10^m 11$  auf Band 2
  - Schreibe  $0^k$  auf Band 3
  - Schreibe  $x_l$  auf Band 1
  - Bewege Kopf von Band 1 in Richtung  $D_m$

**Beobachtung** Für festes  $M$  ist die Rechenzeit von  $U$  auf der Eingabe  $< M >, W$  nur um einen konstanten Faktor größer, als die Rechenzeit von  $M$  auf  $W$ .

## Church'sche These

- Die durch die formale Definition der Turing- Maschine erfasste Klasse berechenbarer Funktionen stimmt überein mit der Klasse der intuitiv berechenbaren Funktionen.
- Die Turing- Maschine erfasst die Rechenzeit bis auf polynomielle Faktoren richtig.

## 5.2 Problem

Eine Relation  $R$  auf  $\Sigma^* \times \Sigma^*$  mit  $(x, y) \in R \Leftrightarrow y \in \Sigma^*$  ist zulässige Ausgabe zur Eingabe  $x \in \Sigma^*$

### 5.2.1 Spezialfall Funktion

$$f : \Sigma^* \rightarrow \Sigma^*$$

Gegeben:  $x \in \Sigma^*$

Gesucht:  $y := f(x) \in \Sigma^*$

### 5.2.2 Entscheidungsprobleme

Ein Spezialfall der Funktionen.

$$f : \Sigma^* \rightarrow \{0, 1\}$$

Gegeben:  $x \in \Sigma^*$  Entscheide, ob  $f(x) = 1$ .

**zugehörige Sprache**  $L := f^{-1}(1) \subseteq \Sigma^*$

### 5.2.3 Es gibt unlösbare Probleme

Computerprogramme  $\subseteq \{0, 1\}^*$  Es gibt „nur“ abzählbar unendlich viele Computerprogramme.

Probleme  $\supseteq \{0, 1\}^{\{\{0, 1\}^*\}} \cong \mathbb{R} \Rightarrow$  Fast alle Probleme sind unlösbar.

## 5.3 Berechenbarkeit

**Definitionen** DTM = deterministische Turing- Maschine

- 1  $f : \Sigma^* \rightarrow \Sigma^*$  heißt total rekursiv oder berechenbar, falls  $\exists$  DTM, die zur Eingabe  $x \in \Sigma^*$  die Ausgabe  $f(x) \in \Sigma^*$  erzeugt.
- 2  $f : \mathbb{N}^* \rightarrow \mathbb{N}$  heißt total rekursiv, falls  $\exists$  DTM, die zur Eingabe  $bin(i_1)\#bin(i_2)\#\dots\#bin(i_k)$  die Ausgabe  $bin(m)$  erzeugt, wobei  $m = f(i_1, i_2, \dots, i_k)$ .
- 3 Eine Sprache  $L \subseteq \Sigma^*$  heißt rekursiv (entscheidbar), wenn  $\exists$  DTM, die auf alle Eingaben  $\in \Sigma^*$  stoppt und die Eingabe  $w \in \Sigma^*$  genau dann akzeptiert, wenn  $w \in L$ .
- 4 Eine Sprache  $L \subseteq \Sigma^*$  heißt rekursiv aufzählbar (semi- entscheidbar), wenn  $\exists$  DTM, die genau die Eingaben aus  $L$  akzeptiert. (Stopp bei Eingaben  $\in \Sigma^* \setminus \{L\}$  nicht gefordert.)

### 5.3.1 Unentscheidbare Probleme

#### Die Diagonalsprache

**Definition Kanonische Ordnung** Für  $w, w' \in \Sigma^*$  schreibe  $w \leq w'$ , falls  $w$  kürzer ist als  $w'$  oder beide gleich lang sind und  $w$  lexikographisch kleiner ist als  $w'$ . Definiert totale Ordnung auf  $\Sigma^*$ .

**Definition Diagonalsprache** Sei  $M_i$  die DTM, deren Gödelnummer an  $i$ - ter Stelle in der Liste aller Gödelnummern steht,  $w_i$  das Wort aus  $\Sigma^*$ , das an  $i$ - ter Stelle in der kanonischen Ordnung von  $\Sigma^*$  steht. Die Diagonalsprache ist definiert als

$$D := \{w_i \mid M_i \text{ akzeptiert } w_i \text{ nicht}\}$$

**Satz**  $D$  ist nicht rekursiv.

**Beweis** Angenommen,  $\exists$  DTM, die  $D$  entscheidet.

Es gilt:  $M = M_j$  für ein  $j \in \mathbb{Z} > 0$ .

Frage: Was liefert  $M_j$  bei Eingabe  $w_j$ ?

1. Fall  $w_j \in D \Rightarrow M_j$  akzeptiert  $w_j$

aber nach Definition von  $D$  akzeptiert  $M_j w_j$  nicht.  $\Rightarrow$  **Widerspruch!** ■

2. Fall  $w_j \notin D \Rightarrow M_j$  akzeptiert  $w_j$  nicht

aber nach Definition von  $D$  akzeptiert  $M_j w_j$ .  $\Rightarrow$  **Widerspruch!** ■

Es gibt keine DTM, die  $D$  entscheidet. □

**Korollar**  $\bar{D} := \Sigma^* \setminus D$  ist auch nicht rekursiv.

**Beweis** Eine Sprache  $L$  ist rekursiv  $\Leftrightarrow$  ihr Komplement ist rekursiv.

(Vertausche akzeptierende und nicht akzeptierende Endzustände der zugehörigen DTM.)  $\square$

## Das Halteproblem

### Definition Halteproblem

$$H := \{ \langle M \rangle w \mid M \text{ hält auf } w \}$$

**Satz**  $H$  ist nicht rekursiv.

**Beweis** Angenommen,  $M$  ist DTM, die  $H$  entscheidet. Mit  $M$  konstruiere DTM  $M'$ , die das Komplement der Diagonalsprache entscheidet.

Es sei  $w \in \Sigma^*$ , entscheide ob  $w \in \bar{D}$ .

- Berechne  $i$  mit  $w = w_i$
- Berechne  $\langle M_i \rangle$
- Wende  $M$  an auf  $\langle M_i \rangle w$ 
  - Falls  $\langle M_i \rangle w \notin H$ , so ist  $w \notin \bar{D}$ .
  - Falls  $\langle M_i \rangle w \in H$  simuliere  $M_i$  auf Eingabe  $w$  (mit Hilfe der universellen DTM).  $M_i$  akzeptiert  $w_i \Leftrightarrow w_i \in \bar{D}$ .

Damit ließe ich  $D$  entscheiden.  $\Rightarrow$  **Widerspruch!**  $\square$

## Das spezielle Halteproblem

**Definition spezielles Halteproblem** Das spezielle Halteproblem  $H_\epsilon$  ist definiert durch

$$H_\epsilon = \{ \langle M \rangle \mid M \text{ hält auf } \epsilon \text{ an} \}$$

Dabei ist  $\epsilon$  das leere Wort.

**Satz**  $H_\epsilon$  ist nicht rekursiv.

**Beweis** Annahme: DTM  $M'$  entscheidet  $H_\epsilon$ . Konstruiere daraus TM  $M''$ , die  $H$  entscheidet:

- $M''$  berechnet auf Eingabe  $\langle M \rangle w$  zunächst  $\langle M''_w \rangle$ , die Folgendes leistet:

Schreibe  $w$  auf das Band und simuliere  $M$  auf  $w$ .

- $M''$  simuliert  $M'$  auf  $\langle M''_w \rangle$

→ Dann gilt:  $M''$  akzeptiert  $\langle M \rangle w \Leftrightarrow M''_w$  hält auf  $\epsilon$  an  $\Leftrightarrow M$  hält auf  $w$  an.  $\square$

**Korollar**  $\bar{H}_\epsilon$  ist nicht rekursiv.

## Die universelle Sprache

### Definition universelle Sprache

$$U = \{\langle M \rangle w \mid M \text{ akzeptiert } w\}$$

**Satz**  $U$  ist nicht rekursiv.

**Beweis** Andernfalls  $\exists$  DTM  $M$ , die  $\bar{D}$  entscheidet.

Gegeben:  $w \in \Sigma^*$ , ist  $w \in \bar{D}$ ?

Berechne  $i$ , sodass  $w = w_i$  und  $\langle M_i \rangle$ . Entscheide, ob  $\langle M_i \rangle w_i \in U$ .

**Satz**  $U$  ist rekursiv aufzählbar (semi- entscheidbar).

**Beweis** Wende die universelle DTM an auf  $\langle M \rangle w$ . Falls  $Mw$  akzeptiert geschieht das nach endlich vielen Schritten → Ausgabe nach endlich vielen Schritten.

## Satz von Rice

**Definition zur Berechenbarkeit** Sei  $Q \subseteq \Sigma^*, f : Q \rightarrow \Sigma^*$ .

$f$  heißt berechenbar, wenn  $\exists$  DTM, die auf die Eingabe  $x \in Q$  die Ausgabe  $f(x)$  liefert und für Eingabe  $x \notin Q$  nicht anhält oder nicht akzeptiert.

**Spezielles Beispiel:**  $Q = \emptyset$  (leere Menge)  
 $U : \emptyset \rightarrow \Sigma^*$  ist berechenbar.

Zugehörige DTM: Gehe im ersten Schritt in nicht- akzeptierenden Endzustand.

**Satz von Rice** Sei  $R$  die Menge aller berechenbaren Funktionen und  $\emptyset \neq S \subsetneq R$  eine nichttriviale Teilmenge von  $R$ . Dann ist die Sprache  $L(S) := \{\langle M \rangle \mid M \text{ berechnet Funktion aus } S\}$  nicht rekursiv.

Das heißt: Ein Computer kann keinerlei bedeutende Aussagen über einen gegebenen Algorithmus treffen.

**Beweis** Erfolgt per Widerspruch:

**Annahme:**  $M_S$  entscheidet  $L(S)$ . Konstruiere daraus eine DTM  $M'$ , die  $\bar{H}_\epsilon$  entscheidet.

Betrachte im Folgenden die spezielle Funktion  $U \in R$  mit

$$U : \emptyset \rightarrow \Sigma^*$$

Nehme o.B.d.A. an, dass  $u \in S$ . Andernfalls ersetze  $S$  durch  $R \setminus S$  und beachte, dass  $L(R \setminus S) = \overline{L(S)}$ .

Weiter sei  $f \in R \setminus S$  und  $M_f$  eine DTM, die  $F$  berechnet.

Die DTM  $M'$  arbeitet wie folgt (bei Eingabe  $\langle M \rangle$ ):

- 1 Berechne Gödelnummer  $\langle M'' \rangle$  der DTM  $M''$ , die auf Eingabe  $x$  Folgendes tut:
  - $M''$  simuliert  $M$  auf Eingabe  $\epsilon$
  - Falls  $M$  auf  $\epsilon$  anhält, so simuliert  $M''$  die DTM  $M_f$  auf  $x$ .
- 2 Simuliere DTM  $M_S$  auf  $\langle M'' \rangle$ .

**Dann gilt:**

1. Fall:  $M$  hält auf  $\epsilon$  nicht an

$$\begin{aligned} &\Rightarrow M'' \text{ berechnet } U \in S \\ &\Rightarrow M_S \text{ akzeptiert } \langle M'' \rangle \\ &\Rightarrow M' \text{ akzeptiert } \langle M \rangle \end{aligned}$$

2. Fall:  $M$  hält auf  $\epsilon$  an

$$\begin{aligned} &\Rightarrow M'' \text{ berechnet } f \in R \setminus S \\ &\Rightarrow M_S \text{ akzeptiert } \langle M'' \rangle \text{ nicht} \\ &\Rightarrow M' \text{ akzeptiert } \langle M \rangle \text{ nicht}. \end{aligned}$$

Folglich entscheidet  $M'$  die Sprache  $\overline{H}_\epsilon$ .  $\Rightarrow$  **Widerspruch!**

□

## 5.4 Der Gödel'sche Unvollständigkeitssatz

### 5.4.1 Satz von Church

**Definition Sprache der wahren Sätze** Für eine fest vorgegebene Menge an Relationen  $\{R; 1, \dots, R_k\}$  sei  $L$  die Sprache der wahren Sätze:  $L = \{Q_1x_1\dots Q_lx_l\Phi = S \mid S \text{ ist wahrer Satz}\}$

**Satz von Church**  $L$  ist nicht rekursiv, falls unter den Relationen  $R_1, \dots, R_k$  die Additionsrelation und die Multiplikationsrelation ist.

**Beispiel** Es ist nicht entscheidbar, ob ein Polynom über mehreren Variablen eine ganzzahlige Nullstelle besitzt.

### 5.4.2 Annahmen über Beweise

Der Gödelsche Unvollständigkeitssatz beruht auf folgenden Annahmen über „Beweise“ von Sätzen:

1 Ein Beweis  $\Pi$  eines Satzes  $S$  kann von einer DTM überprüft werden, d.h.

$$\{S\Pi \mid \Pi \text{ ist Beweis von } S\}$$

ist rekursiv.

2 Ist  $\Pi$  ein Beweis von  $S$ , so ist  $S$  wahr.

Unter diesen Annahmen gilt:

**Satz** Die Sprache  $L'$  aller beweisbaren Sätze ist rekursiv aufzählbar.

$$L' := \{S \mid \exists \text{ Beweis } \Pi \text{ für } S\}$$

**Beweis** Zu einem gegebenen Satz überprüft man alle möglichen „Beweise“ der Reihe nach, nach aufsteigender Länge mithilfe der DTM aus Annahme 1.

Gibt es also einen gültigen Beweis für  $S$ , so wird dieser in endlicher Zeit gefunden.

### 5.4.3 Gödel'scher Unvollständigkeitssatz

**Satz** Es sei  $L$  die Sprache der wahren Sätze, die mittels Additions- und Multiplikationsrelation formuliert werden können.

Dann gibt es Sätze in  $L$ , die nicht beweisbar sind.

**Beweis** Durch Widerspruch

Nehme an, dass  $L = L'$ . Betrachte gegebenen Satz  $S$ . Dann ist entweder  $S$  oder seine Negierung wahr.

Wende die DTM aus dem vorigen Beweis parallel an auf  $S$  und die Negierung von  $S$ .

Da einer von beiden Sätzen wahr ist, gibt es (unter der Widerspruchsannahme  $L = L'$ ) einen Beweis für einen von beiden, der in endlicher Zeit gefunden wird.

$\Rightarrow L$  ist rekursiv.

$\Rightarrow$  Widerspruch zum Satz von Church

□

## 5.5 Komplexitätsklasse P und NP- Vollständigkeit

### 5.5.1 Definitionen

**Definition** Sei  $M$  DTM auf Eingabealphabet  $\Sigma$ . Die worst-case-Rechenzeit  $t_M(n)$  ist die maximale Anzahl Rechenschritte, die  $M$  auf einer Eingabe aus  $\Sigma^n$  durchführt.

**Definition**  $P$  ist die Klasse der Sprachen  $L$  (Entscheidungsprobleme), für die  $\exists$  DTM  $M$ , die  $L$  entscheidet und

$$t_M(n) \leq p(n)$$

für eine Polynomfunktion  $p$ .

**Definition** Eine Sprache (ein Problem) in  $P$  heißt **effizient entscheidbar**. Der zugehörige Algorithmus (DTM) wird dann **effizient** genannt.

### 5.5.2 Ist ein Problem effizient lösbar?

Wie zeigt man, dass ein Problem nicht effizient lösbar ist?

- 1 Zeige, dass das Problem nicht entscheidbar ist.
- 2 Zeige, dass die erwartete Ausgabe nicht polynomial in der Eingabegröße beschränkt ist.  
(Normalerweise ist so ein Problem dann falsch gestellt.)
- 3 Direkte Beweise (nur in seltenen Fällen bekannt, z.B. Pressburger Arithmetik)

Keiner dieser drei Ansätze ist praktisch zufriedenstellend.

## Das Cliquesproblem

**Definition** Es sei  $G = (V, E)$  ein ungerichteter Graph. Dann bildet  $V' \subseteq V$  eine Clique, wenn  $\forall$  Paare  $u, v \in V'$  mit  $u \neq v$  eine Kante zwischen  $u$  und  $v$  existiert.

### Varianten

**1. Variante:** Gegeben  $G = (V, E), k \in \mathbb{N}$ .

Gibt es eine Clique  $V' \subseteq V$  mit  $|V'| = k$ ?  
(Entscheidungsproblem)

**2. Variante** Gegeben  $G = (V, E)$

Berechne das größte  $k \in \mathbb{N}$ , sodass  $G$  eine Clique der Größe  $K$  besitzt.  
(Optimierungsproblem)

**3. Variante** Gegeben  $G = (V, E)$

Finde  $V' \subseteq V$  in  $G$ , sodass  $V'$  eine maximale Clique ist, d.h.  $|V'|$  maximal.  
(Optimierungsproblem)

**Satz** Gibt es für eine der 3 Varianten des Cliquesproblems einen effizienten Algorithmus, dann auch für die anderen beiden.

**Bemerkung:** Beachte, dass  $\exists$  unterschiedliche Arten, einen Graphen zu codieren (z.B., Adjazenzmatrix, Adjazenzlisten, ...). Diese unterscheiden sich in der Größe jedoch nur um polynomiale Faktoren.

### Beweis

- Kann man Variante 3 effizient lösen, dann auch Variante 1 und 2.
- Ein effizienter Algorithmus für Variante 2 löst auch Variante 1 effizient.
- Zeige noch, dass ein effizienter Algorithmus für Variante 1 auch Variante 3 effizient löst.

Gegeben effiziente Unterroutine für Variante 1, konstruiere effizienten Algorithmus für Variante 3:

- 1 For  $k = 1$  to  $n$ : Rufe Unterroutine auf für  $G = (V, E)$  und  $k$ .  
Bestimme dabei  $\max k = k^*$ , für das die Antwort ja ist.
- 2 For all  $v \in V$ : Rufe Unterroutine auf für den Graphen  $G'$ , der durch Löschen von  $v$  entsteht, und für  $k^*$ .  
Falls „ja“, dann lösche  $v$  aus  $G$ .

Nach Schritt 2 bleibt von  $G$  nur noch eine Clique der Größe  $k^*$  übrig.  $\square$   
Diese Beobachtung gilt im Wesentlichen für alle Optimierungsprobleme. Daher genügt es, die Komplexität der zugehörigen Entscheidungsprobleme zu studieren.

### 5.5.3 Komplexitätsklasse NP

### 5.5.4 Nichtdeterministische Turing- Maschine

**Beobachtung:** Es scheint algorithmisch schwer zu sein, das Cliquesproblem zu lösen, d.h. es ist kein effizienter Algorithmus bekannt.

**Aber:** Bekommt man eine Lösung (Clique) genannt, so kann man leicht überprüfen, ob dies eine zulässige Lösung ist.

**Definition Nichtdeterministische TM** Eine NTM ist wie eine DTM definiert, mit dem einzigen Unterschied, dass die Funktion

$$\delta : Q \times \Gamma \rightarrow (Q \times \Gamma \times \{L, R, N\})$$

durch eine Relastion  $\delta$  auf

$$(Q \times \Gamma) \times (Q \times \Gamma \times \{L, R, N\})$$

ersetzt wird.

Ist die NTM im Zustand  $q \in Q$  und liest  $a \in \Gamma$ , so kann sie sich in den Zustand  $q' \in Q$  begeben, das Zeichen  $a' \in \Gamma$  schreiben und den Kopf in Richtung  $d \in \{L, R, N\}$  bewegen, falls

$$((q, a), (q', a', d)) \in \delta$$

Die NTM hat daher unter Umständen mehrere Möglichkeiten in einem Rechenschritt. Daher ergeben sich für eine Eingabe mehrere Rechenwege und zugehörige Ausgaben.

**Definition** Eine NTM  $M$  akzeptiert die Eingabe  $w$ , falls es mindestens einen Rechenweg gibt, der zu einem akzeptierenden Endzustand führt.

$$L(M) := \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$$

### Intuition

Eine NTM testet parallel alle möglichen Rechenwege.

ODER

Eine NTM kann ein „Orakel“nach dem besten Rechenweg fragen.

**Definition** Betrachte NTM  $M$ , die die Sprache  $L$  akzeptiert. Die Rechenzeit von  $M$  auf Eingabe  $w$  ist wie folgt definiert:

- Falls  $w \in L$ , die Länge eines kürzesten akzeptierenden Rechenwegs.
- Falls  $w \notin L$ , 0

Wie bei DTMs ist die Laufzeitfunktion  $t_M(n)$  definiert als

$$t_M(n) := \max \text{ Rechenzeit auf Eingabe } \in \Sigma^*$$

### 5.5.5 Komplexitätsklasse NP

**Definition** NP ist die Klasse aller Sprachen  $L$ , für die es eine NTM  $M$  gibt, die  $L$  akzeptiert und  $t_M(n) \leq (N)\forall n \in \mathbb{N}$  für eine Polynomfunktion  $p$ .

**Satz** Das zu CLIQUE gehörende Entscheidungsproblem (Variante 1) ist in NP.

**Beweisidee** Die NTM rät zunächst  $k$  Knoten (nichtdeterministisch!) und überprüft dann (deterministisch), ob diese eine Clique bilden.  $\square$

**Satz**  $P \subseteq NP$

**Beweis** Es sei  $L \in P$  und  $M$  zugehörige DTM, fasse  $M$  als NTM auf.  $\square$

**Offenes Problem** Ist  $P \not\subseteq NP$  oder  $P = NP$ ? (Millennium- Problem)

**Satz** Für jede Sprache  $L \in NP$  gibt es ein Polynom  $P$  und eine DTM  $M$ , so dass  $M$  die Sprache  $L$  entscheidet und die Laufzeit von  $M$  beschränkt ist durch  $t_M(n) \leq 2^{p(n)}$ .

**Beweisskizze** Betrachte NTM  $M'$  für  $L$  mit polynomieller Laufzeit. Die DTM  $M$  probiert alle Rechenwege von  $M'$  sequentiell durch.  $\square$

### NP- Vollständigkeit

**Definition** Es seien  $L_1, L_2$  Sprachen über  $\Sigma_1, \Sigma_2$ . Dann heißt  $L_1$  polynomiell reduzierbar auf  $L_2$  ( $L_1 \leq_p L_2$ ), wenn es eine von einer DTM in polynomieller Zeit berechenbare Funktion

$$f : \Sigma_1^* \rightarrow \Sigma_2^*$$

gibt, sodass

$$\forall w \in \Sigma_1^* : w \in L_1 \Leftrightarrow f(w) \in L_2$$

**Satz**  $L_1 \leq_p L_2, L_2 \in P \Rightarrow L_1 \in P$

**Beweisskizze** Gegeben  $w \in \Sigma_1^*$ , berechne  $f(w) \in \Sigma_2^*$ , entscheide, ob  $f(w) \in L_2$ . Beachte: Länge von  $f(w)$  ist polynomiell beschränkt in der Länge von  $w$ .  $\square$

**Satz** Es gilt:

$$\text{Hamiltonkreisproblem} \leq_p TSP$$

**Lemma**  $L_1 \leq_p L_2, L_2 \leq_p L_3 \Rightarrow L_1 \leq_p L_3$

## Beweis

$$x \in L_1 \Leftrightarrow f_1(x) \in L_2$$

$$y \in L_2 \Leftrightarrow f_2(y) \in L_3$$

$$\Rightarrow x \in L_1 \Leftrightarrow f_2(f_1(x)) \in L_3$$

Sei  $f_3 := f_2 \circ f_1$ , dann ist  $f_3$  polynomiell berechenbar.

$$x \in L_1 \Leftrightarrow f_3(x) \in L_3$$

□

**Definition** Eine Sprache  $L$  heißt NP- schwer, falls  $L' \leq_p L \forall L' \in \text{NP}$ .

Eine NP- schwere Sprache  $L$  heißt NP- vollständig, falls  $L \in \text{NP}$ .

**Satz** Es sei  $L$  NP- vollständig.

- Falls  $L \in P$ , so ist  $P = NP$ .
- Falls  $L \notin P$ , so ist  $L' \notin P \forall$  NP- vollständige Sprachen  $L'$ .

## Satisfiability Problem (SAT)

**Gegeben** Boole'sche Variablen  $x_1, \dots, x_n$  und  $m$  Klauseln, d.h. Disjunktion von Literalen  $x_i$  und  $\overline{x_i} := \neg x_i, i = 1, \dots, n$

**Frage** Gibt es eine Belegung der Variablen, so dass alle  $m$  Klauseln erfüllt sind?

## Beispiel

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2) \wedge (x_2 \vee \overline{x_4} \vee x_5) \wedge (x_3 \vee x_4 \vee \overline{x_5}) \quad (n = 5, m = 4)$$

**Satz von Cook** SAT ist NP- vollständig.

**Beweis:** SAT  $\in \text{NP}$ : NTM „räät“erfüllende Belegung von  $x_1, \dots, x_n$  und überprüft diese.

Es sei nun  $L \in NP$ , zu zeigen:  $L \leq_p SAT$ !

Zu  $L$  gibt es eine NTM  $N$ , die  $L$  mit worst- case Rechenzeit  $p(n)$  entscheidet ( $p$  Polynomfunktion).

$$Q = \{q_0, q_1, \dots, q_l\}, F = \{q_0\}, q_1 \text{ Anfangszustand}$$

$$\Sigma = \{\sigma_1, \dots, \sigma_s\}$$

Zu gegebenem  $x \in \Sigma^*$  konstruiere Instanz von SAT  $f(x)$ , so dass

$$N \text{ akzeptiert } x : \Leftrightarrow x \in L \Leftrightarrow f(x) \in SAT$$

**Zeige:**  $f$  kann so in polynomieller Zeit konstruiert werden.

$f(x)$  enthält 8 Arten von Klauseln über den folgenden Boole'schen Variablen:

- |                |          |  |
|----------------|----------|--|
| $band_{i,j,t}$ | bedeutet | die $i$ -te Stelle des Bandes enthält zur Zeit $t$ das Symbol $\sigma_j$             |
| $cursor_{i,t}$ | bedeutet | der Cursor zeigt zur Zeit $t$ auf die $i$ -te Stelle des Bands                       |
| $zust_{k,t}$   | bedeutet | $N$ ist zur Zeit $t$ im Zustand $q_k$  |
| $wahl_{r,t}$   | bedeutet | $N$ wählt zur Zeit $t$ die $r$ -te Alternative aus $\Delta$ für den nächsten Schritt |

mit  $0 \leq i \leq p(|x|)$ ,  $1 \leq j \leq s$ ,  $0 \leq k \leq l$ ,  $0 \leq t \leq p(|x|)$ ,  $1 \leq r \leq |Q| \cdot |\Sigma| \cdot 3$

Verwende die folgenden Schreibweisen für Boole'sche Variablen  $P_1, \dots, P_a$ :

$$\left( \bigvee_i P_i \right) \wedge \bigwedge_{i < j} (\overline{P_i} \vee \overline{P_j})$$

$\Theta_1$ : Zu jeder Zeit  $t$  zeigt der Cursor von  $N$  auf genau eine Stelle des Bands:

$$\bigwedge_t \exists! i : \text{cursor}_{i,t}$$

$\Theta_2$ : Zu jeder Zeit steht an jeder Stelle des Bands genau ein Zeichen:

$$\bigwedge_{i,t} \exists! j : \text{band}_{i,j,t}$$

$\Theta_3$ : Zu jeder Zeit ist  $N$  in genau einem Zustand:

$$\bigwedge_t \exists! k : \text{zust}_{k,t}$$

$\Theta_4$ : Zu jeder Zeit wählt  $N$  genau eine der sich bietenden Möglichkeiten für den nächsten Schritt aus:

$$\bigwedge_t \exists! r : \text{wahl}_{r,t}$$

$\Theta_5$ : Zur Zeit 0 steht auf dem Band  $\triangleright x\epsilon = \sigma_{j_1}\sigma_{j_2}\dots\sigma_{j_{n+1}}\sigma_{j_{n+2}}\dots\sigma_{j_{p(n)}}$ . Der Anfangszustand ist  $s = q_1$ :

$$\text{zust}_{1,0} \wedge \text{cursor}_{0,0} \wedge \bigwedge_i \text{band}_{i,j_i,0}$$

$\Theta_6$ : Spätestens zur Zeit  $p(n)$  ist  $N$  im Zustand 'yes' =  $q_0$ :

$$\bigvee_t \text{zust}_{0,t}$$

$\Theta_7$ : Zu jeder Zeit kann nur das Symbol geändert werden, das an der Stelle steht, auf die der Cursor gerade zeigt:

$$\bigwedge_{i,j,t} \left( (\text{cursor}_{i,t} \vee \text{band}_{i,j,t} \vee \neg \text{band}_{i,j,t+1}) \wedge (\text{cursor}_{i,t} \vee \neg \text{band}_{i,j,t} \vee \text{band}_{i,j,t+1}) \right)$$

(Beachte, dass  $\neg X \Rightarrow (Y \Leftrightarrow Z)$  äquivalent ist zu  $(X \vee Y \vee \neg Z) \wedge (X \vee \neg Y \vee Z)$ .)

$\Theta_8$ : Zu jeder Zeit  $t$  folgt der nächste Schritt der Maschine einer Regel aus der Relation  $\mathcal{S}$  (ein gegebenes Paar  $(q_k, \sigma_j) \in \mathcal{Q} \times \Sigma$  eröffne die Möglichkeiten  $((q_k, \sigma_j), (q_{k_r}, \sigma_{j_r}, D_r)) \in \mathcal{S}$ ,  $r = 1, \dots, R$ ):

$$\begin{aligned} \bigwedge_{i,j,k,r,t} & \left( (\neg \text{zust}_{k,t} \vee \neg \text{cursor}_{i,t} \vee \neg \text{band}_{i,j,t} \vee \neg \text{wahl}_{r,t} \vee \text{zust}_{k_r,t+1}) \wedge \right. \\ & (\neg \text{zust}_{k,t} \vee \neg \text{cursor}_{i,t} \vee \neg \text{band}_{i,j,t} \vee \neg \text{wahl}_{r,t} \vee \text{band}_{i,j_r,t+1}) \wedge \\ & \left. (\neg \text{zust}_{k,t} \vee \neg \text{cursor}_{i,t} \vee \neg \text{band}_{i,j,t} \vee \neg \text{wahl}_{r,t} \vee \text{cursor}_{i+\delta_r,t+1}) \right) \end{aligned}$$

wobei

$$\delta_r := \begin{cases} 1 & \text{falls } D_r = \rightarrow, \\ 0 & \text{falls } D_r = -, \\ -1 & \text{falls } D_r = \leftarrow. \end{cases}$$

(Beachte, dass  $A_1 \wedge A_2 \wedge A_3 \wedge A_4 \Rightarrow B$  äquivalent ist zu  $\neg A_1 \vee \neg A_2 \vee \neg A_3 \vee \neg A_4 \vee B$ .)

Man kann sich davon überzeugen, dass es genau dann eine Belegung der Variablen gibt, die alle Klauseln erfüllt, wenn  $N$  mit Eingabe  $x$  in einen akzeptierenden Endzustand gelangen kann.  $\square$

**Lemma** Ist  $L_2 \in NP$  und  $L_1$  NP- vollständig und  $L_1 \leq_p L_2$ , so ist  $L_2$  NP- vollständig.

**Beweis** Transitivität von  $\leq_p$

□

**Satz** SAT  $\leq_p$  CLIQUE

**Korollar** CLIQUE ist NP- vollständig.

# 6 Anhang

## 6.1 Grundbegriffe

### Computer

- Maschine, die geistige Routinearbeiten durchführt
- führt einfache Operationen mit hoher Geschwindigkeit aus
- ⇒ Arbeit muss in Basisoperationen zerlegt werden und dem Rechner mitgeteilt werden (Algorithmus)

### Algorithmus

- präzise
- endlich
- enthält ausführbare elementare Arbeitsschritte

### Prozess

- Ausführung eines Algorithmus'
- besitzt stets einen Zustand, der den aktuellen Stand der Ausführung angibt

### Prozessor

- führt den Prozess aus
- kann Mensch oder Maschine sein

## 6.2 Pseudocode

Präzise, sprachunabhängige Beschreibung von Algorithmen.

Keine sprach- spezifische Syntax, keine Typdeklarationen, keine Effizienz- Tricks

### Grundform:

*Algorithmenname (Parameterliste)*

**Input:**

**Output:**

Programmschritte

	Bedeutung	Pseudocode	Java
<b>Befehle</b>	Zuweisung	$:=$	$=$
	Vergleich	$=, \neq, \leq, \geq, <, >$	$==, !=, \leq=, \geq=, <, >$
	Logisches UND	AND, $\wedge$	$\&\&$
	Logisches Oder	OR, $\vee$	$\ $
	Logisches Nicht	NOT, $\neg$	!
	Kommentar	//	//, /* */
	Rückgabe	RETURN	return

### Besondere Anweisungen:

#### Verzweigung

```
IF (condition) THEN
...
ELSE IF (condition2) THEN
...
ELSE
ENDIF
```

```
1 if ( condition ) {
2 } else if ( condition2 ) {
3 } else {
4 }
```

#### For- Schleife

```
FOR i := 1 TO n DO
...
ENDFOR
```

```
FORALL a ∈ A DO
...
ENDFOR
```

```
1 for ( int i = 1; i <=n; i = i+1) {
2 }
3
5 for ( Datentyp a : A) {
6 }
7 }
```

**While- Schleife** WHILE (condition) DO

```
...
ENDWHILE
```

DO

...

WHILE (condition)

```
1 while (condition) {  
2  
3  
4 do {  
5  
6 } while (condition);
```

## 6.3 Beweis der Richtigkeit von Algorithmen

Erfolgt, v.a. bei Rekursion, mit Induktion. Gerne werden dazu Schleifeninvarianten verwendet. Alternativ dazu gibt es Widerspruchsbeweise.

### 6.3.1 Terminologie

**partielle Korrektheit** Liefert das korrekte Ergebnis bei Terminieren.

**Terminieren** Der Algorithmus läuft in keinem Fall unendlich lange (Endlosschleifeö.ä.), sondern ist nach einer endlichen Zeit abgearbeitet.

**totale Korrektheit** Ist partiell korrekt und terminiert immer.

### 6.3.2 Schleifeninvariante

Ist eine Eigenschaft einer Schleife in einem Algorithmus, die zu einem bestimmten Zeitpunkt in jedem Schleifendurchlauf gültig ist.

### 6.3.3 Widerspruchsbeweis

Heißt auch indirekter Beweis oder reductio ad absurdum. Folgendes Vorgehen:

- Nimm das logische Gegenteil dessen an, was bewiesen werden soll (Widerspruchsannahme).
- Mit Widerspruchsannahme und den Voraussetzungen der Behauptung schlussfolgern.
- Widerspruch herleiten
- Fertig.

## 6.4 Laufzeitanalyse

A priori (Rechnerunabhängig, s.u.) oder A posteriori (empirisch getestet)

**Worst-Case Komplexität:** Obere Schranke für die Ausführungszeit in Form der Anzahl auszuführender Operationen in Abhängigkeit von der Größe des Inputs, gemessen in relevanten Parametern (z.B. Anzahl der zu sortierenden Objekte, Stellenzahl von Zahlen, etc.)

**Mittlere Komplexität:** Obere Schranke für die mittlere Ausführungszeit bei gewissen (Wahrscheinlichkeits-) Annahmen über das Auftreten der Problemdaten

**Untere Komplexitätsschranken:** Untere Schranken für die (worst-case oder mittlere) Ausführungszeit.

Im Idealfall liegen obere und untere Schranke nah beieinander, das ist in der Praxis jedoch nur schwer zu erreichen.

### 6.4.1 Asymptotische Notation

Sei  $g : \mathbb{N} \rightarrow \mathbb{N}$ . Dann bezeichnet man

$$O(g) := \{f : \mathbb{N} \rightarrow \mathbb{N} \mid \exists c > 0 \quad \text{und} \quad \exists n_0 \in \mathbb{N} \quad \text{mit} \quad f(n) \leq c \cdot g(n) \forall n \geq n_0\}$$

Liegt eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  zu dieser Menge, so schreibt man  $f \in O(g(n))$ .

Für eine Polynomfunktion  $f(n) = a_m \cdot n^m + a_{m-1} \cdot n^{(m-1)} + \dots + a_1 n + a_0$  mit  $a_m \neq 0$  gilt  $f \in O(n^m)$ .

Terme kleiner Ordnung sowie Konstanten werden vernachlässigt, weil bei großen  $n$  allein die Größenordnung ausschlaggebend ist und Konstanten und Terme niedriger Ordnung teils auch von der Maschine oder Programmiersprache abhängig sind.

## 6.4.2 Unterscheidung von Größenordnungen

Erfolgt per o- Notation:

$$o(g) := \{f : \mathbb{N} \rightarrow \mathbb{N} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \text{ mit } f(n) < c \cdot g(n) \forall n \geq n_0\}$$

$f \in o(g) \Leftrightarrow f$  ist von (echt) kleinerer Größenordnung als  $g$ .

Es gilt:  $O(f(n)) < O(g(n)) \Leftrightarrow f(n) \in o(g(n))$

## 6.4.3 untere Schranken

$$f \in \Omega(g) : \Leftrightarrow \exists c > 0, \exists n_0 \in \mathbb{N} : f(n) \geq c \cdot g(n) \forall n \geq n_0$$

Es gelte  $f \in O(g_1)$  und  $f \in \Omega(g_2)$ .

Gilt  $g_1 = g_2$ , so sagt man  $f \in \Theta(g_1)$ ,  
d.h.  $\Theta(g) := O(g) \cap \Omega(g)$ , d.h.

$$f \in \Theta(g) : \Leftrightarrow \exists c_1, c_2 > 0, n_0 \in \mathbb{N} : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0$$

Außerdem:  $f \in \omega(g) : \Leftrightarrow g \in o(f) \Leftrightarrow \forall c > 0 \exists n_0 \in \mathbb{N} : f(n) > c \cdot g(n) \forall n \geq n_0$

## 6.5 Begriffe der Wahrscheinlichkeitstheorie

### (Endlicher Fall)

- diskrete Zufallsgröße  $X$
- hat Werte  $x_1, x_2, \dots, x_n$ , die mit
- Wahrscheinlichkeiten  $p_1, p_2, \dots, p_n$  auftreten.
- Erwartungswert der Zufallsgröße  $X$  ist  $E(X) = \sum_{i=1}^n x_i \cdot p_i$

### Abzählbar unendlicher Fall

- hat Werte  $x_1, x_2, \dots, x_i, \dots$ , mit
- Wahrscheinlichkeiten  $p_1, p_2, \dots, p_i, \dots$
- $E(X) = \sum_{i=1}^{\infty} x_i \cdot p_i$ , falls Reihe absolut konvergent

## 6.6 Optimalitätsbeweis des Huffman-Algorithmus'

### Lemma 1

Es seien  $x, y \in C$  die Zeichen mit den kleinsten Häufigkeiten, d.h.

$$f(x) \leq f(y) = f(c) \quad \forall c \in C \setminus \{x, y\}$$

Dann gibt es einen optimalen Präfixcode  $T$ , in dem  $x$  und  $y$  auf der tiefsten Stufe liegen, d.h.  $h_T(x) = h_T(y) > h_T(c) \quad \forall c \in C \setminus \{x, y\}$  und Geschwister sind, (d.h. gemeinsamer Mutterknoten)

(Achtung: gilt nicht für jeden opt. Präfixcode)

Beweis: Es sei  $T$  ein optimaler Präfixcode  
( $\exists$  endl. viele Codes heißen "Bäume")

Es sei  $a \in C$  ein Blatt mit großer Höhe  $h_T(a)$   
 $h_T(a) > h_T(c) \quad \forall c \in C$

Beh.: Dann hat  $a$  einen Geschwister  $b \in C$ , denn

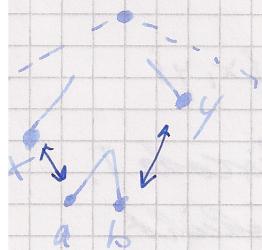


Somit ist  $T'$  nicht opt.  
lebhafte Baum  $T'$   
mit  $B(T') < B(T)$   $\nabla$   
Widerspruch zur Optimalität  
von  $T$ .

Fall 2:  $\{a, b\} = \{x, y\}$ , dann falsig.

Falls nicht, dann modifiziere  $T$  eingängig  
ohne die Optimalität zu verlieren.

$\rightarrow T'$  mit  $B(T') = B(T)$  und  
 $T'$  erfüllt Eigenschaft aus heuern.



Modifiziere in 2 Schritten:  
1) vertausche  $x$  mit  $a$ :  $T \rightarrow T_1$   
2)  $\text{---} \rightarrow y$  mit  $b$ :  $T_1 \rightarrow T'$

$$\begin{aligned}
 B(T) - B(T_1) &= \sum f(a) \cdot h_T(a) - \sum f(a) \cdot h_{T_1}(a) \\
 &= f(a) \cdot h_T(a) + f(x) \cdot h_T(x) - f(a) \cdot h_{T_1}(a) \\
 &= f(x) \cdot \underbrace{h_{T_1}(x)}_{h_T(a)} = \overline{h_{T_1}(x)} \\
 &= \underbrace{f(a) - f(x)}_{\geq 0} \cdot \underbrace{(h_T(a) - h_{T_1}(a))}_{\geq 0} \geq 0
 \end{aligned}$$

Opt. von  $T \Rightarrow B(T_1) = B(T)$

$B(T_1) - B(T') = 0$  genauer

$\Rightarrow B(T') = B(T_1) = B(T)$ .  $\square$

Lemma 2: Es seien  $x, y \in C$  Zeichen mit geringsten Häufigkeiten und  $T$  ein Präfix-Code, in dem  $x$  und  $y$  Geschwister sind, mit gemeinsamem Elternknoten  $z$ .

Es entstehe  $T'$  aus  $T$  durch Ablösen, der Blätter  $x$  und  $y$  und  $C' = (C \setminus \{x, y\}) \cup \{z\}$  und  $f'(c) = \begin{cases} f(c), & \text{falls } c \neq z \\ f(x) + f(y), & \text{falls } c = z \end{cases}$

Dann gilt:  $T$  ist optimal für  $C$  und  $f$  genau dann, wenn  $T'$  optimal ist für  $C'$  und  $f'$ .

Beweis: Es sei  $H$  die Menge aller Präfixcodes für  $C$ , die die Eigenschaft besitzen, dass  $x$  und  $y$  Geschwister sind mit Mutter  $z$ .

Es sei  $H'$  die Menge aller Präfixcodes für  $C'$ .

Vonn beschreibt  $H \rightarrow H'$  eine Abbildung von  $H \rightarrow H'$ . Diese Abbildung ist bijektiv (wenn man die Anordnung der Kinder  $x, y$  ignoriert oder o.B.d.h.  $\begin{matrix} x \\ y \end{matrix}$  annimmt).

Injectivität: klar

Surjektivität: zu gegebenem  $T'$  konstruiere  $T$  durch Antragen von  $x, y$  an Blatt  $z \Rightarrow T \mapsto T'$

Zeige jetzt:  $B(T) = B(T') + k$

(dann gilt offenbar  $B(T)$  minimal  
 $\stackrel{(*)}{\Leftrightarrow}$   
 $B(T')$  minimal)

$$B(T) - B(T')$$

$$= \sum_{c \in C} f(c) \cdot h_T(c) - \sum_{c \in C'} f(c) \cdot h_{T'}(c)$$

$$= f(x) \cdot h_T(x) + f(y) \cdot h_T(y) - f(z) \cdot h_{T'}(z)$$

$$= (f(x) + f(y)) h_T(x) - (f(x) + f(y)) \cdot (h_{T'}(x) - 1)$$

nach Def. Z-Wert für  $x$

$$= f(x) + f(y) =: k \quad \square$$

Konstante

Theorem: Der Huffman-Algorithmus konstruiert einen optimalen Präfix-Code.

Beweis: Induktion über  $n := |C|$

I. A.:  $n = 2$

opt. Präfixcode:  $\{ \overline{0}, \overline{1} \} \cong$  Ausgabe der Algo.

I. V.: Der Algo findet einen optimalen Präfixcode für bis zu  $n-1$  Zeichen für festes  $n \in \mathbb{N}$ .

### I.5: auf n

In einer Iteration des Algo: ermittle  
zwei Zeichen  $x, y$  mit geringster Häufigkeit  
und konstruiere plausibel kleinen Baum.

$\otimes \oplus$   $f(x) + f(y)$  und füge den neuen Baum in  
 $\mathcal{Q}$  ein.

$\rightarrow \mathcal{Q}$  enthält  $n - 1$  Bäume

Interpretiere  $\otimes \oplus$  als neues Zeichen,  $z$   
mit Häufigkeit  $f(x) + f(y)$

Im weiteren läuft der Huffman-Algorithmus  
genau so, wie er für  $c'$  und  $f'$  läuft.

Nach I. V. liefert er also einen optimalen  
Präfixcode  $T'$  für  $c'$  und  $f'$ .

Lemma 2  $\Rightarrow T$  ist ebenfalls optimal  
für  $c$  und  $f$ . ■

## 6.7 Zu BucketSort

### 6.7.1 Beweis der Korrektheit von BucketSort

Es gilt folgende Invariante: Nach der  $i$ -ten Iteration sind die Strings nach den letzten  $i$  Stellen lexikographisch sortiert.

Beweis per Induktion über  $i$

**Induktionsanfang**  $i = 1$ : einfacher BucketSort liefert Behauptung.

**Induktionsschluss** Betrachte zwei Fälle

- a) Zwei Strings A und B, die in Iteration  $i$  im selben Bucket liegen
- b) A und B in Iteration  $i$  in verschiedenen Buckets

Zu a)

A und B haben dasselbe Zeichen an der  $i$ -letzten Stelle.

$$a_1, \dots, a_l, \dots, a_k \quad b_1, \dots, b_l, \dots, b_k \quad l = k + 1 - i$$

Nach Induktion sind dann A und B lexikographisch sortiert gemäß letzter  $i - 1$  Zeichen.  $\Rightarrow$  Auch lexikographisch korrekt sortiert gemäß letzter  $i$  Stellen.

Zu b)

A und B haben verschiedene Zeichen an Position  $l = k + 1 - i \Rightarrow$  A und B wurden in Iteration  $i$  so angeordnet, dass sie lexikographisch sortiert sind gemäß letzter  $i$  Zeichen.

$\Rightarrow$  Algorithmus arbeitet korrekt. □

### 6.7.2 Erweiterung von BucketSort zum Sortieren von Strings

**Definition** A, B seien Strings über einem Alphabet (Zeichenmenge  $S$ ).  $S$  sei linear geordnet, d.h.  $s, t \in S, s \neq t \Rightarrow s < t$  oder  $s > t$ .

## Der Algorithmus

### Algorithmus 1.2 (BucketSort)

**Input:** Eine Liste  $L$  mit Strings  $A_1, A_2, \dots, A_n$  der Länge  $k$  mit  $A_i = a_{i1}, a_{i2}, \dots, a_{ik}$  und  $a_{ij} \in S = \{0, 1, \dots, m - 1\}$

**Output:** Eine Permutation  $B_1, \dots, B_n$  von  $A_1, \dots, A_n$  mit  $B_1 \leq_{lex} B_2 \leq_{lex} \dots \leq_{lex} B_n$

**Methode:**

1. Richte eine Queue  $Q$  ein und füge  $A_1, \dots, A_n$  in  $Q$  ein.<sup>1</sup>
2. Richte ein Array Bucket von  $m$  Buckets ein (wie beim einfachen BucketSort)
3. **for** jede Stelle  $r := k$  **downto** 1 **do**
  - 3.1 Leere alle Buckets  $\text{Bucket}[i]$
  - 3.2 **while**  $Q$  nicht leer ist **do**
    - Sei  $A_j$  das erste Element in  $Q$
    - Entferne  $A_j$  aus  $Q$  und füge es in  $\text{Bucket}[i]$  mit  $i = a_{jr}$  ein**endwhile**
  - 3.3 Konkateniere die nichtleeren Buckets in die Queue  $Q$**endfor**

**Beweis der Korrektheit** Dann ist  $A <_{lex} B$ , wenn:

- $A$  Präfix von  $B$
- $A$  und  $B$  unterscheiden sich an mindestens einer Stelle und an der ersten solchen Stelle gilt:  $a_l < b_l$

**Spezialfall** Alle Strings haben die gleiche Länge  $k$  und  $S = \{0, 1\}$

**Aufwand**  $k$  mal BucketSort:  $O(k(n + m))$

Bei paarweise verschiedenen Binärzahlen:  $k \geq \log(n)$

→ BucketSort bringt nur Vorteile bei vielen identischen Schlüsseln.

## Umsetzung

**Erste Idee** Verwende  $k$ - stellige Strings mit aufgefüllten kurzen Strings.

**Aufwand**  $O(l_{max}(n + m))$

## Bessere Variante

- 1 Sortiere Strings nach absteigender Länge  $O(\underbrace{n}_{<l_{total}} + \underbrace{l_{max}}_{<l_{total}}) \in O(l_{total})$
- 2 Verwende  $l_{max}$ - mal einfaches BucketSort, aber betrachte in Iteration  $r$  nur die Strings  $a_i$ , die an der  $r$ -ten Stelle ein Zeichen haben.
- 3 Vermeide leere Buckets: Bestimme vorab die benötigten Buckets für jede Iteration und konkateniere dann nur diese.

**Algorithmus 1.3** Input: Strings (Tupel)  $A_1, \dots, A_n$

$$A_i = (a_{i1}, a_{i2}, \dots, a_{i\ell_i}), \quad a_{ij} \in \{0, \dots, m-1\}$$

(oder auch ein beliebiges anderes Alphabet)

$$\ell_{max} = \max_i \ell_i$$

*bas abc a*

Output: Permutation  $B_1, \dots, B_n$  von  $A_1, \dots, A_n$  mit  $B_1 \leq_{lex} B_2 \leq_{lex} \dots \leq_{lex} B_n$

**NE**

1. Generiere ein Array von Listen  $\text{NONEEMPTY}[]$  der Länge  $\ell_{max}$  und für jedes  $\ell$ ,  $1 \leq \ell \leq \ell_{max}$  eine Liste in  $\text{NONEEMPTY}[\ell]$ , die angibt, welche Zeichen an einer der  $\ell$ -ten Stellen vorkommen und welche Buckets daher in der  $(\ell_{max}-\ell)$ -ten Iteration benötigt werden.

Dazu:

$$O(\ell_{max} + m)$$

- 1.1 Erschaffe für jedes  $a_{i\ell}$ ,  $1 \leq i \leq n$ ,  $1 \leq \ell \leq \ell_i$  ein Paar  $(\ell, a_{i\ell})$  (das bedeutet: das Zeichen  $a_{i\ell}$  kommt an  $\ell$ -ter Stelle in einem der Strings vor)
- 1.2 Sortiere die Paare lexikographisch mit Algorithmus 1.2, indem man sie als zweistellige Strings betrachtet.
- 1.3 Durchlaufe die sortierte Liste der  $(\ell, a_{i\ell})$  und generiere im Array  $\text{NONEEMPTY}[]$ , sortierte Listen, wobei das Array  $\text{NONEEMPTY}[\ell]$ ,  $1 \leq \ell \leq \ell_{max}$  eine sortierte Liste aller  $a_{i\ell}$  enthält. Dabei lassen sich auch gleich auf einfache Weise eventuell auftretende Duplikate entfernen.

*NE[1]: a<sub>11</sub>  
NE[2]: a<sub>12</sub>  
NE[3]: b<sub>11</sub>*

2. Bestimme Länge  $\ell_i$  jedes Strings und generiere Listen  $\text{LENGTH}[\ell]$  aller Strings mit Länge  $\ell$  (nur Referenzen auf die Strings in  $\text{LENGTH}[\ell]$  verwalten, daher nur  $O(1)$  für Referenzen umhängen)

$$L \quad O(n + \ell_{max})$$

3. Sortiere Strings analog zu Algorithmus 1.2.3, beginnend mit  $\ell_{max}$ . Aber:

$$O(\ell_{total})$$

- nach der  $r$ -ten Phase enthält  $Q$  nur die Strings der Länge  $\geq \ell_{max} - r + 1$ ; diese sind lexikographisch korrekt sortiert bezüglich der letzten  $r$  Komponenten.
- $\text{NONEEMPTY}[]$  wird benutzt, um die Listen in  $\text{BUCKET}[]$  neu zu generieren und außerdem zur schnelleren Konkatenation der Einzellisten. Dies ist nötig, weil wir nur die nichtleeren Buckets verwalten wollen.
- vor dem  $r+1$ -ten Durchlauf wird  $\text{LENGTH}[\ell_{max} - r]$  am Anfang<sup>2</sup> der Queue  $Q$  eingefügt. Die kurzen Strings stehen dann am Anfang und damit am lexikographisch richtigen Platz, falls sie mit anderen im selben Bucket landen.

**Korrektheit** Folgt aus Korrektheit des allgemeinen BucketSort und der Tatsache, dass die neuen (kurzen) Strings jeweils am Beginn der Liste eingefügt werden.

**Aufwand** Vorbereitung:  $O(l_{total}) + O(l_{total} + m)$

**Sortierphase:** An der Stelle  $l$ :

$n_l$  Strings,  $m_l$  Buckets

$$\rightarrow \text{Aufwand } O\left(\sum_{l=1}^{l_{max}} (n_l + \underbrace{m_l}_{\leq n_l})\right) = O\left(\sum_{l=1}^{l_{max}} \underbrace{n_l}_{l_{total}}\right) = O(l_{total})$$

## 6.8 Schlussfolgerungen über die Berechenbarkeiten „verwandter“ Sprachen

**Satz** Seien  $L, L_1, L_2$  rekursive Sprachen. Dann sind  $\bar{L}, L_1 \cup L_2$  und  $L_1 \cap L_2$  ebenfalls rekursiv.

### Beweis

$\bar{L}$  Klar: Vertausche akzeptierende und nichtakzeptierende Endzustände der DTM, die  $L$  entscheidet.

$L_1 \cup L_2$  Rufe zunächst die DTM auf, die entscheidet, ob Eingabe  $x \in L_1$ .

Falls ja  $\rightarrow$  fertig.

Falls nein  $\rightarrow$  Rufe die DTM auf, die entscheidet, ob  $x \in L_2$ . Falls ja  $\rightarrow$  ja, falls nein  $\rightarrow$  nein.

Alternativer Beweis: Lasse die beiden DTM für  $L_1$  und  $L_2$  „parallel“ auf einer 2- Band- DTM laufen. Akzeptiere, falls mindestens eine von beiden akzeptiert.

$L_1 \cap L_2$  Wie oben.

Alternativ:  $L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$

**Satz** Seien  $L_1, L_2$  rekursiv aufzählbar. Dann sind auch  $L_1 \cup L_2$  und  $L_1 \cap L_2$  rekursiv aufzählbar.

**Beweis** Nutze Parallelsimulation von oben.

**Lemma** Sind  $L$  und  $\bar{L}$  rekursiv aufzählbar, so ist  $L$  rekursiv.

**Beweis** Parallelsimulation, s.o.

**Satz** Die Menge der rekursiv aufzählbaren Funktionen ist nicht abgeschlossen bezüglich Komplementbildung. Insbesondere ist  $U$  rekursiv aufzählbar,  $\bar{U}$  jedoch nicht.

**Beweis**  $U$  ist rekursiv aufzählbar, aber nicht rekursiv.

**Korollar** Für jede Sprache  $L$  gibt es genau eine der folgenden 3 Eigenschaften:

- 1  $L$  und  $\bar{L}$  sind rekursiv.
- 2  $L$  und  $\bar{L}$  sind nicht rekursiv aufzählbar.
- 3 Genau eine der beiden Sprachen  $L$  und  $\bar{L}$  ist rekursiv aufzählbar (aber nicht rekursiv!)

## 6.9 Zum Gödel'schen Unvollständigkeitssatz

### 6.9.1 Mathematische Ausdrücke (Sätze)

#### Ein Beispiel

$$\forall q \in \mathbb{N}_0 \exists p \in \mathbb{N}_0 \forall x, y \in \mathbb{N}_0 (p > q \wedge (x, y > 1 \Rightarrow (x \cdot y \neq p, x \cdot y \neq p + 2)))$$

**Alphabet**  $\Sigma = \{\wedge, \vee, \neg, (,), \exists, \forall, x, R_1, \dots, R_k\}$

$R_1, \dots, R_k$  Relationen

z.B. Additionsrelation:  $R_1(x_i, x_j, x_l) = \text{"wahr"} \Leftrightarrow x_i + x_j = x_l$

**Definition** Ein Satz ist ein Ausdruck  $w \in \Sigma^*$  der Form:

$Q_1 x_1 \dots Q_l x_l \Phi$

Wobei  $Q_1, \dots, Q_l \in \{\forall, \exists\}$

und die Formel  $\Phi$  rekursiv wie folgt definiert ist:

- 1  $\Phi = R_j(x_{i_1}, x_{i_2}, \dots, x_{i_n})$  für  $1 \leq j \leq k, 1 \leq i_r \leq l$  ist Formel.
- 2  $\Phi = \Phi_1 \wedge \Phi_2$  oder  $\Phi = \Phi_1 \vee \Phi_2$  oder  $\Phi = \neq \Phi_1$ , wobei  $\Phi_1, \Phi_2$  kürzere Formeln sind.

**Ein Satz ist also ein mathematischer Ausdruck, der wahr oder falsch ist.**

### 6.9.2 Hilberts Programm

Baue die Mathematik von Grund auf neu, ausgehend von Axiomen, die konsistent und widerspruchsfrei sein müssen. Die Axiome sollen außerdem vollständig sein, d.h. alles Wahre soll daraus formell abgeleitet werden können.

Gödel zerstörte diesen Traum mit seinem Unvollständigkeitssatz.

## 6.10 Sonstiges

"Überabzählbar unendlich > abzählbar unendlich"