Algorithmen und Datenstrukturen
Vorlesungsskript WS/SS 99-00

Gunter Saake, Kai-Uwe Sattler
Universität Magdeburg

4. Juli 2000

Vorwort

Das vorliegende Skript ist zum großen Teil erst während der aktuellen Vorlesungsvorbereitung entstanden. Dies war nur dadurch möglich, daß die Autoren auf eine Reihe von Vorarbeiten zurückgreifen konnten.

Teile des Skriptes, insbesondere Beispiele und Definitionen, sind aus dem Material anderer Skripte entnommen und geeignet abgewandelt worden. Besonders nennen möchten wir hierbei

- Unterlagen zu Einführungsvorlesungen von Hans-Dieter Ehrich, die in den letzten Jahren an der TU Braunschweig statgefunden haben,
- die Unterlagen der vorherigen Vorlesungen von Jürgen Dassow [zu nennen sind insbesondere der Abschnitt über die Registermaschinen], Rudolf Kruse und Maritta Heisel [insbesondere abstrakte Datentypen],
- Unterlagen zu einer Vorlesung zum Thema "Information Retrieval: Datenstrukturen und algorithmische Grundlagen" von Peter Becker von der Univ. Tübingen (http://sunburn.informatik.uni-tuebingen.de/~becker/ir/) [Algorithmen zur Textsuche].

Weitere Beispiel und Algorithmen wurden einer Reihe von Lehrbüchern entlehnt, von denen wir hier nur die wichtigsten nennen wollen:

- Goldschlager / Lister [GL90] [insbesondere Grundkonzepte von Algorithmen],
- Schülerduden Informatik und Duden Informatik [dB86, Lek93] [einige grundlegende Definitionen und Beispiele],
- Goodrich / Tamassia [GT98] [grundlegende Datenstrukturen in Java]
- Saake / Heuer: Datenbankimplementierungskonzepte [SH99] [dynamische Datenstrukturen: B-Bäume, digitale Bäume, dynamische Hash-Verfahren, DBMS-Techniken],
- Aho / Ullman: Informatik. Datenstrukturen und Konzepte der Abstraktion [Material über Datenstrukturen] [AU96],
- Thomas Cormann, Charles Leiserson, Ronald Rivest: Introduction to Algorithms, McGraw-Hill, 0-07-013143-0 [CLR90] [Durchlauf durch Graphen, topologisches Sortieren],
- M.A. Weiss: Data Structures & Algorithm Analysis in Java, Addison Wesley [AVL-Realisierung in Java] [Wei98],
- J. Friedl: Mastering Regular Expressions [Fri97],

Wir danken ferner allen Hörerinnen und Hörern der Vorlesung, die uns auf Fehler und Ungenauigkeiten auf Folien und im Skript aufmerksam gemacht haben — wir hoffen daß wir dadurch alle gefunden (und korrekt beseitigt) haben!

Inhaltsverzeichnis

Teil I.

Algorithmen

1. Einführung

1.1. Vorbemerkungen

Informatik

- Kunstwort aus den 60ern (Informatik → Information + Technik *oder* Informatik → Information + Mathematik)
- beabsichtigt: Gegensatz zur amerikanischen *Computer Science*: nicht nur auf Computer beschränkt
- Theoretische / Praktische / Angewandte / Technische Informatik; 'Bindestrich-Informatiken'

Informatik hat zentral zu tun mit

- systematischer Verarbeitung von Informationen
- Maschinen, die diese Verarbeitung automatisch leisten (→ Computer)

Hier: maschinenunabhängige Darstellung

> Die "systematische Verarbeitung" wird durch den Begriff *Algorithmus* präzisiert, Information durch den Begriff *Daten*.

Algorithmus (erste Näherung):

> Eindeutige Beschreibung eines Verarbeitungsvorganges.

In der Informatik speziell: Berechnungsvorgänge statt Bearbeitungsvorgänge, Schwerpunkt auf *Ausführbarkeit* durch (abstrakte) Maschinen.

> Ein *Prozessor* führt einen Prozeß (Arbeitsvorgang) auf Basis einer eindeutig interpretierbaren Beschreibung (dem Algorithmus) aus.

Typische algorithmisierbare Prozesse;

- Kochrezepte
- Bedienungsanleitungen
- Rechnerprogramm (Prozessor: Computer!)

Fragestellungen:

- Notation für Beschreibung
- Ausdrucksfähigkeit (man vergleiche Notationen der Bienensprache, dressierte Hunde)
- Korrektheit / Genauigkeit / Eindeutigkeit
- Zeitbedarf / Geschwindikeit

In dieser Vorlesung:

Algorithmen für Rechner

also

- schnelle (aber dumme) Prozessoren; 'Hochgeschwindigkeitstrottel'
- mathematisch / formale Grundlagen (Rechner 'versteht' nur Bits)
- 'Kunst des Programmierens'
  - große Systeme
  - korrekte Systeme
  - benutzerfreundliche Systeme
  - Programmerstellung im Team
  - wartbare / vertändliche Programme

1.2. Historischer Überblick: Algorithmen

- 300 v. Chr.: Euklids Algorithmus zur Bestimmung des **ggT**, (7. Buch der Elemente):

  $$\textbf{ggT}(300, 200) = 100$$

- 800 n. Chr.: Muhammed ibn Musa abu Djafar alChoresmi:[1]

  Aufgabensammlung für Kaufleute und Testamentsvollstrecker (lat.: Liber Algorithmi, Kunstwort aus dem Namen und griechisch 'arithmos' für Zahl)
- 1574: Adam Rieses Rechenbuch
- 1614 Logarithmentafeln (30 Jahre für Berechnung!)
- 1703 Binäres Zahlensysteme (Leibnitz)
- 1931 Gödels Unvollständigkeitssatz
- 1936 Church'sche These
- danach Ausbau der Algorithmentheorie

[1] Oft auch als 'al Chworesmi' geschrieben.

1.3. Java – ein Überblick

Java - was ist das ?

- Objektorientierte Programmiersprache
  - entwickelt von Sun Microsystems
  - Internet-Programmiersprache
- Plattform
  - Umgebung zur Ausführung von Java-Programmen für PC, Workstation, Handhelds, Set-Top-Boxen, ...
  - Bibliothek von nützlichen Klassen/Funktionen, z.B. Datenbankzugriff, 2D/3D-Grafik, verteilte Verarbeitung, ...

Java – Algorithmen & Datenstrukturen

- Vorlesung
  - Umsetzung von Algorithmen
  - Implementierung von Datenstrukturen
  - Einführung in die Prgrammierung
- Übungen
  - Lösung der praktischen Aufgaben
- Praktikum
  - Belegaufgaben
  - Teilnahme am Programmierwettbewerb

Historisches

- 1990: Oak – als Programmiersprache für Consumer Electronics (Sun Microsystems)
- 1993: Entwickung des World Wide Web
  - Umorientierung auf Web-Anwendungen
  - Umbenennung in Java
- 1995: Freigabe des HotJava-Browsers
  - Aktive Web-Inhalte (Applets)
  - Erste größere Java-Anwendung

- 1995: Netscape Navigator 2.0 mit Applet-Unterstützung
- 1997: Freigabe des JDK 1.1
  - Unterstützung durch alle großen Firmen: IBM, Oracle, Microsoft, ...
- 1998: Java 2
- 1999: OpenSource-Lizenz

Applet vs. Applikation

- **Applet**
  - Java-Programm, das in andere Applikation eingebettet ist
  - Bsp.: Applets in Web-Dokumenten: werden vom Web-Server in den Browser geladen
  - Sicherheitsrestriktionen: kein Zugriff auf lokalen Computer
  - Anwendung: Frontend zu Internet-Diensten, Präsentation, aktive Dokumente (3D-Welten), ...
- **Applikation**
  - Java-Programme, die unabhängig von anderen Anwendungen ausgeführt werden können (*standalone*)
  - keine Sicherheitsrestriktionen
  - Anwendung: Entwicklungswerkzeuge, Office-Anwendungen, Browser, ...

Eigenschaften von Java

- einfach
  - automatisierte Speicherverwaltung
  - Verzicht auf Zeiger
  - Verzicht auf `goto`
- objektorientiert
  - Klassen als Abstraktionskonzept
- robust und sicher
  - starke Typisierung
  - Laufzeitüberprüfung von Zugriffen
- interpretiert und dynamisch

- virtuelle Java-Maschine
- einfache, schnelle Programmenticklung
- „kleine“ Programme

- architekturneutral und portabel
  - plattformunabhängiger Zwischencode (Bytecode)
  - Programme sind ohne Änderungen ablauffähig unter Windows, Unix, MacOS, ...

Java-Werkzeuge

- Java-Compiler `javac`
  - überprüft Quelltext auf Fehler
  - übersetzt Quelltext in plattformneutralen Zwischencode (Bytecode)
- Java-Interpreter `java`
  - intepretiert Bytecode
  - implementiert virtuelle Java-Maschine

Übersetzung und Ausführung

```java
// Hello.java – Das erste Java Programm.

// Jedes Java-Programm besteht aus mind. einer Klasse.
public class Hello {
    // Eine Standalone-Anwendung muß eine main-Methode besitzen
    public static void main (String[] args) {
        // Zeichenkette ausgeben
        System.out.println ("Hello Java !");
    }
}
```

1. Quelltext in Datei `Hello.java` speichern
   - Dateiname entspricht Klassennamen !
   - Klasse muß eine Methode `main` als Startpunkt der Ausführung besitzen
2. Quelltext kompilieren
   - Quelltext in Bytecode übersetzen
   - liefert Datei `Hello.class` Java in den FIN-Pools

3. Java-Programm ausführen
   - `java Hello`
   - Interpretieren des Bytecodes
4. Ergebnis
   `Hello Java !`

2. Algorithmische Grundkonzepte

2.1. Intuitiver Algorithmus-Begriff

2.1.1. Beispiele für Algorithmen

Beispiel 2.1 Algorithmen im täglichen Leben:

- Bedienungsanleitungen
- Gebrauchsanleitungen
- Bauanleitungen
- Kochrezepte □

Intuitive Begriffsbestimmung:

> Ein *Algorithmus* ist eine präzise (d.h., in einer festgelegten Sprache abgefaßte) endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-) Schritte.

Beispiel 2.2 Bekannte Algorithmen:

1. Addition zweier positiver Dezimalzahlen (mit Überträgen)

$$\begin{array}{rcc} & 3 & 3 \\ + & 4 & 8 \\ & {\scriptstyle 1} & \\ \hline & 8 & 1 \end{array}$$

2. Test, ob einen gegebene natürliche Zahl eine Primzahl ist
3. Sortieren einer unsortierten Kartei (etwa lexikographisch)
4. Berechnung der Zahl $e = 2.7182\ldots$

Begriff der *Terminierung*:

> Ein Algorithmus heißt *terminierend*, wenn er (bei jeder erlaubten Eingabe von Parameterwerten) nach endlich vielen Schritten abbricht.

Begriff des *Determinismus*:

- deterministischer Ablauf: eindeutige Vorgabe der Schrittfolge
- determiniertes Ergebnis: bei vorgegebener Eingabe eindeutiges Ergebnis (auch bei mehrfacher Durchführung)

Beispiel 2.3 Beispiel für nicht-deterministischen Ablauf:

> Sortieren: Wähle zufällig eine Karte, bilde zwei Stapel (lexikographisch vor der Karte, lexikographisch nach der Karte), sortiere dies (kleineren) Stapel, füge die sortierten Stapel wieder zusammen.

Beispiel für nicht-determiniertes Ergebnis:

> Wähle zufällig eine Karte. □

Nicht-deterministische Algorithmen mit determiniertem Ergebnis heißen *determiniert*.

Beispiel 2.4 Beispiel für nicht-determinierten Algorithmus:

1. Nehmen Sie eine Zahl x;
2. Addieren Sie 5 hinzu und multiplizieren Sie mit 3;
3. Schreiben Sie das Ergebnis auf.

Beispiel für determinierten, nicht-deterministischen Algorithmus:

1. Nehmen Sie eine Zahl x ungleich 0;
2. Entweder: Addieren Sie das Dreifache von x zu x und teilen das Ergebnis durch x
$$(3x + x)/x$$
Oder: Subtrahieren Sie 4 von x und subtrahieren das Ergebnis von x;
$$x - (x - 4)$$
3. Schreiben Sie das Ergebnis auf. □

Wichtige Klasse: deterministische, terminierende Algorithmen. Diese definieren jeweils eine Ein/Ausgabefunktion:

$$f : \text{Eingabewerte} \rightarrow \text{Ausgabewerte}$$

Bemerkung 2.1 Algorithmen geben eine konstruktiv ausführbare Beschreibung dieser Funktion, die Funktion heißt *Bedeutung* (*Semantik*) des Algorithmus. Es kann mehrere verschiedene Algorithmen mit der gleichen Bedeutung geben. □

Beispiel 2.5 Funktionen zu Algorithmen aus 2.2:

1. Addition zweier positiver Dezimalzahlen (mit Überträgen)

$$f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q} \;\text{ mit }\; f(p,q) = p + q$$

   $\mathbb{Q}$ seien die positiven Rationalzahlen

2. Test, ob eine gegebene natürliche Zahl eine Primzahl ist

$$f : \mathbb{N} \to \{\texttt{ja}, \texttt{nein}\} \;\text{ mit }\; f(n) = \begin{cases} \texttt{ja} & \text{falls } n \text{ Primzahl} \\ \texttt{nein} & \text{sonst} \end{cases}$$

3. Sortieren einer unsortierten Kartei (etwa lexikographisch)

   K Menge von Karteikarten

   S_K Menge von sortierten Karteien über K

   US_K Menge von unsortierten Karteien über K

$$f : US_K \to S_K$$

4. Berechnung der Stellen der Zahl $e = 2.7182\ldots$

   nicht terminierend! □

2.1.2. Bausteine für Algorithmen

Gängige Bausteine für Algorithmenbeschreibungen (am Beispiel von Kochrezepten):

- elementare Operationen

  ‘*Schneide Fleisch in kleine Würfel.*’

- sequentielle Ausführung (ein Prozessor!)

  ‘*bringe das Wasser zum Kochen, dann gib Paket Nudeln hinein, schneide das Fleisch, dann das Gemüse.*’

- parallele Ausführung (mehrere Prozessoren!)

  ‘*ich schneide das Fleisch, Du das Gemüse.*’

- bedingte Ausführung

  ‘*wenn Soße zu dünn, füge Mehl hinzu.*’

- Schleife

  ‘*rühre bis Soße braun*’

- Unter‘programm’

  ‘bereite Soße nach Rezept Seite 42’

- Rekursion: Anwendung des selben Prinzip auf kleinere Teilprobleme

  ‘gutes Beispiel aus Kochrezepten unbekannt; bitte Vorschläge an den Autor dieser Materialien schicken!’

  Beispiel: ‘Sortieren von Adreßkarten: Lege alle Karten mit Namen vor ‘M’ nach links, alle Karten mit Namen mit ‘M’ und danach nach rechts; sortiere dann die beiden kleineren Stapel’

  NEU: *Koch-Beispiel von Ivo Rössling*:

```
"Schneide Fleischstückchen":
---------------------------
   Wenn Länge* Fleischstückchen > 5mm,
   Dann
      - Teile Fleischstückchen in Mitte von Länge
      - Schneide Fleischstückchen #1
      - Schneide Fleischstückchen #2
   (sonst wirf Fleischstückchen in Kochtopf)
```

Die Konstrukte

- elementare Operationen + Sequenz + Bedingung + Schleifen

reichen aus (allerdings auch andere Kombinationen)!

2.1.3. Pseudo-Code-Notation für Algorithmen

Notation von Algorithmen

Wir nutzen eine semi-formale Notation, angelehnt an Goldschlager / Lister [GL90].

Sequenz

```
(1)   Koche Wasser
(2)   Gib Kaffeepulver in Tasse
(3)   Fülle Wasser in Tasse
```

Sequenz und Verfeinerung

```
(2)   Gib Kaffeepulver in Tasse
```

verfeinert zu

```
(2.1) Öffne Kaffeeglas
(2.2) Entnehme Löffel von Kaffee
(2.3) Kippe Löffel in Tasse
(2.4) Schließe Kaffeeglas
```

Entwurfsprinzip der *schrittweisen Verfeinerung*!

Sequenzoperator

Sequenzoperator: **`;`**

```
Koche Wasser;
Gib Kaffeepulver in Tasse;
Fülle Wasser in Tasse
```

Erspart die Durchnumerierung.

Auswahl / Selektion

```
falls Bedingung
   dann Schritt
```

bzw.

```
falls Bedingung
   dann Schritt a
   sonst Schritt b
```

Beispiel:

```
falls Ampel rot oder gelb
   dann stoppe
   sonst fahre weiter
```

Schachtelung:

```
falls Ampel ausgefallen
   dann fahre vorsichtig weiter
   sonst falls Ampel rot oder gelb
               dann stoppe
               sonst fahre weiter
```

`falls` ... **`dann`** ... **`sonst`** ... entspricht in Programmiersprachen den Konstrukten

```
if Bedingung then ...  else ...  fi
if Bedingung then ...  else ...  endif
if ( Bedingung ) ...  else ...
```

Schleifen / Iteration

```
wiederhole Schritte
bis Abbruchkriterium
```

Beispiel:

```
{* nächste Primzahl *}
wiederhole
   Addiere 1 ;
   Teste auf Primzahleigenschaft
bis Zahl Primzahl ist;
gebe Zahl aus
```

Die inneren Schritte heissen jeweils *Schleifenrumpf*.

Varianten der Iteration

```
solange Bedingung
führe aus Schritte
```

Beispiel:

```
{* Bestimmung der größten Zahl einer Liste *}
Setze erste Zahl als bislang größte Zahl
solange Liste nicht erschöpft
führe aus
   Lies nächste Zahl der Liste
   falls diese Zahl > bislang größte Zahl
      dann setze diese Zahl als bislang größte Zahl;
gebe bislang größte Zahl aus
```

Iteration über festen Bereich:

```
wiederhole für Bereichsangabe
Schleifenrumpf
```

typische Bereichsangaben: jede Zahl zwischen 1 und 100, jedes Wagenrad, jeden Hörer der Vorlesung, ...

Diese Schleifenkonstrukte entsprechen wieder jeweils Programmiersprachenkonstrukten:

```
wiederhole ...  bis          repeat ...  until ...
                             do ...  while ...
solange ...  führe aus       while ...  do ...
                             while (...)  ...
```

`wiederhole für`	`for each ... do ...`
	`for ... do ...`
	`for (...) ...`

Pseudo-Code-Notation

Deutsche Schlüsselworte entsprechen Programmiersprachenkonstrukten (**`if`**, **`then`** **`while`** etc), vorgestellte Notation entspricht somit einer vereinfachten *Pseudo-Code-Notation*.

2.1.4. Struktogramme

Graphische Notation für Sequenz, Bedingung, Schleife (siehe Informatik-Duden)

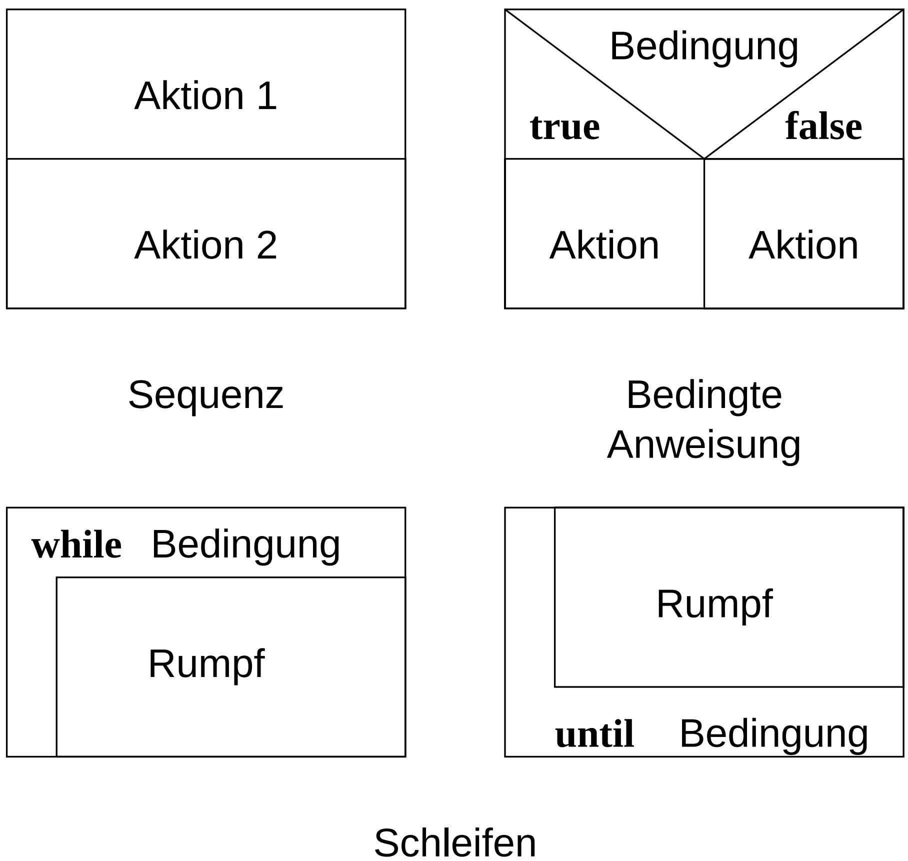


Abbildung 2.1.: Notation für Struktogramme

Notation für Struktogramme wird in Abbildung 2.1 eingeführt (weitere Konstrukte für Mehrfachverzweigung etc.). Elementare Aktionen entsprechen Rechtecken, Konstrukte werden beliebig ineinander geschachtelt.

2.1.5. Rekursion

Das Thema Rekursion wird später noch ausführlich behandelt. Hier nur ein kurzes Beispiel ("Die Türme von Hanoi", siehe Goldschlager Lister [GL90] Seiten 57 bis 59 in ausführlicher Beschreibung).

Regeln:

- Türme von Scheiben unterschiedlichem Umfangs auf drei Plätzen
- nur jeweils oberste Scheibe eines Turmes darf einzeln bewegt werden
- es darf niemals eine größere auf einer kleineren Scheibe liegen

Aufgabe: Bewegen eines Turmes der Höhe n (etwa $n = 64$ im Originalbeispiel) von einem Standort zum einem zweiten unter Benutzung eines dritten Hilfsstandorts.

Beispiel 2.6 Türme von Hanoi (rekursiv)

```
Modul Turmbewegung(n, Quelle, Senke, Arbeitsbereich)
   { Bewegt einen Turm der Höhe n von Quelle
     nach Senke unter Benutzung des
     Arbeitsbereichs }
falls n = 1
dann bewege Scheibe von Quelle zur Senke
sonst Turmbewegung(n-1, Quelle, Arbeitsbereich, Senke)
      bewege 1 Scheibe von Quelle zur Senke
      Turmbewegung(n-1, Arbeitsbereich, Senke, Quelle)
```

folgender Ablauf verdeutlicht die Vorgehensweise (Bewegung von A nach B, Rekursion durch Einrückung verdeutlicht, nur Aufrufe der Turmbewegungen [als `Turm` abgekürzt] und Scheibenbewegungen).

```
Turm(3,A,B,C)
   Turm(2,A,C,B)
      Turm(1,A,B,C)
         bewege A → B
      bewege A → C
      Turm(1,B,C,A)
         bewege B → C
   bewege A → B
   Turm(2,C,B,A)
      Turm(1,C,A,B)
         bewege C → A
      bewege C → B
      Turm(1,A,B,C)
         bewege A → B
```

Bei 64 Scheiben ca. 600.000.000.000 Jahre bei einer Sekunde pro Bewegung einer Scheibe ($2^{64} - 1$ Sekunden!). □

2.2. Sprachen und Grammatiken

Beschreibung von Algorithmen muß

- 'verstehbar' und
- ausführbar

sein.

Hier Verstehbarkeit im Sinne der Festlegung einer Sprache, die von Rechner interpretiert werden kann.

- *Syntax*.

  Formale Regeln, welche Sätze gebildet werden können:

  - "Der Elefant aß die Erdnuß." (syntaktisch korrekt)
  - "Der Elefant aß Erdnuß die." (syntaktisch falsch)

- *Semantik*.

  (Formale) Regeln, welche Sätze eine *Bedeutung* haben:

  - "Der Elefant aß die Erdnuß." (semantisch korrekt, 'sinnhaft')
  - "Die Erdnuß aß den Elephant." (semantisch falsch, 'sinnlos')

Syntax und Semantik: Inhalt späterer Studienabschnitte, hier nur einige kurze Ausführungen.

Ziel: semantisch korrekte Sätze einer Algorithmensprache entsprechen ausführbaren Algorithmen!

2.2.1. Begriffsbildung

- Grammatik.

  Regelwerk zur Beschreibung der Syntax.

- Produktionsregel.

  Regel einer Grammatik zum Bilden von Sätzen.

  z.B.

  $$\text{Satz} \mapsto \text{Subjekt Prädikat Objekt}.$$

- generierte Sprache.

  Alle durch Anwendungen der Regeln einer Sprache erzeugbaren Sätze

Genaueres später; jetzt nur zwei Formalismen zur Beschreibung einfacher 'Kunst'-Sprachen, die im Laufe der Vorlesung eingesetzt werden.

2.2.2. Reguläre Ausdrücke

Einfache Konstrukte, um Konstruktionsregeln für Zeichenketten festzulegen:

- 'Worte' a, b, etc.
- Sequenz: pq
- Auswahl: $p + q$
- Iteration: p^* (0, 1 oder n-mal)
  Variante der Iteration: p^+ (1 oder n-mal)
- Klammerung zur Strukturierung, ansonsten 'Punkt- vor Strichrechnung'

Beispiele

- Mit L beginnende Binärzahlen über L und O:

  $$L(L + O)^*$$

  zusätzlich auch die O als einzige mit diesem Symbol startende Zahl:

  $$0 + L(L + O)^*$$

- Bezeichner einer Programmiersprache, die mit einem Buchstaben anfangen müssen:

  $$(a + b + ... + z)(a + b + ... + z + 0 + 1 + ...9)^*$$

Relevant für:

- Festlegung von Datenformaten für Programmeingaben
- Suchen in Texten
- Such-Masken in Programmierprachen

2.2.3. Backus-Naur-Form (BNF)

Festlegung der Syntax von Kunstsprachen:

- Ersetzungsregeln der Form

  $$\texttt{linkeSeite} ::= \texttt{rechteSeite}$$

- `linkeSeite` ist Name des zu definierenden Konzepts
- `rechteSeite` gibt Definition in Form einer Liste von Sequenzen aus Konstanten und anderen Konzepte (evtl. einschließlich dem zu definierenden!).

  Listenelemente sind durch $|$ getrennt.

- Beispiel (Bezeichner):

$$
\begin{array}{rcl}
\langle\texttt{Ziffer}\rangle & ::= & 1|2|3|4|5|6|7|8|9|0 \\
\langle\texttt{Buchstabe}\rangle & ::= & a|b|c|\ldots|z \\
\langle\texttt{Zeichenkette}\rangle & ::= & \langle\texttt{Buchstabe}\rangle| \\
 & & \langle\texttt{Ziffer}\rangle| \\
 & & \langle\texttt{Buchstabe}\rangle\langle\texttt{Zeichenkette}\rangle| \\
 & & \langle\texttt{Ziffer}\rangle\langle\texttt{Zeichenkette}\rangle| \\
\langle\texttt{Bezeichner}\rangle & ::= & \langle\texttt{Buchstabe}\rangle| \\
 & & \langle\texttt{Buchstabe}\rangle\langle\texttt{Zeichenkette}\rangle|
\end{array}
$$

- Sprechweise:
  - definierte Konzepte: Nichtterminalsymbole
  - Konstanten: Terminalsymbole
    (hier endet die Ersetzung)

Beispiel 2.7 Syntax für Pseudo-Code-Algorithmen

$$
\begin{array}{rcl}
\langle\texttt{atom}\rangle & ::= & \texttt{'addiere 1 zu x'}|\ldots \\
\langle\texttt{bedingung}\rangle & ::= & \texttt{'x=0'}|\ldots \\
\langle\texttt{sequenz}\rangle & ::= & \langle\texttt{block}\rangle;\langle\texttt{block}\rangle \\
\langle\texttt{auswahl}\rangle & ::= & \textbf{falls}\langle\texttt{bedingung}\rangle\textbf{dann}\langle\texttt{block}\rangle| \\
 & & \textbf{falls}\langle\texttt{bedingung}\rangle\textbf{dann}\langle\texttt{block}\rangle\textbf{sonst}\langle\texttt{block}\rangle \\
\langle\texttt{schleife}\rangle & ::= & \textbf{wiederhole}\langle\texttt{block}\rangle\textbf{bis}\langle\texttt{bedingung}\rangle| \\
 & & \textbf{solange}\langle\texttt{bedingung}\rangle\textbf{führe aus}\langle\texttt{block}\rangle \\
\langle\texttt{block}\rangle & ::= & \langle\texttt{atom}\rangle|\langle\texttt{sequenz}\rangle|\langle\texttt{auswahl}\rangle|\langle\texttt{schleife}\rangle
\end{array}
$$

□

Relevant für:

- Festlegung der Syntax für Programmiersprachen
- Definition komplexerer Dateiformate

BNF bildet spezielle Form kontextfreier Grammatiken (später im Studium).

Erweiterungen (oft EBNF für *Extended BNF*) integrieren Elemente regulärer Ausdrücke (optinale Teile mittels [...], Iterationen mittels {...}) in die einfache BNF-Notation (siehe etwa Eintrag Backus-Naur-Form in [Lek93]). *Syntaxdiagramme* bilden eine grphische Variante (siehe ebnfalls in [Lek93]).

2.3. Elementare Datentypen

Ausführbare elementare Schritte eines Algorithmus basieren meistens auf den Grundoperationen eines *Datentyps*.

2.3.1. Datentypen als Algebren

- Algebra = Wertemenge plus Operationen

  (mathematisches Konzept)

  – Beispiel: Natürliche Zahlen $\mathbb{N}$ mit $+, -, *, \div$, etc.

- Wertemengen werden in der Informatik als Sorten bezeichnet
- Operationen entsprechen Funktionen, werden durch *Algorithmen*
- *mehrsortige Algebra* = Algebra mit mehreren Sorten

  – Beispiel: Natürliche Zahlen plus Wahrheitswerte mit $+, -, *, \div$, auf Zahlen, $\neg, \wedge, \vee, \ldots$ auf Wahrheitswerten, $=, <, >, \leq, \ldots$ als Verbindung

- Datentyp (im Gegensatz zum mathematischen Konzept der Algebra): Interpretierbare Werte mit ausführbaren Operationen.

  (hier interessant: *durch Rechner* interpretierbar, *durch Rechner* ausführbar!)

2.3.2. Signaturen von Datentypen

Begriff der Signatur:

- Formalisierung der Schnittstellenbeschreibung eines Datentyps
- Angabe der (Bezeichner / Namen der) Sorten
- Angabe der (Bezeichner / Namen der) Operationen
- Stelligkeit der Operationen
- Sorten der einzelnen Parameter
- *Konstanten* als nullstellige Operationen

Beispiel 2.8 *Beispiel natürliche Zahlen*

```
type nat
sorts nat, bool
functions
   0 → nat
   succ : nat → nat
```

$+ : \mathit{nat} \times \mathit{nat} \rightarrow \mathit{nat}$
$\leq : \mathit{nat} \times \mathit{nat} \rightarrow \mathit{bool}$
...

Das Operationssymbol **succ** *steht für die Nachfolgerfunktion* successor, *also den unären Operator* '+1'.

Das Beispiel ist angelehnt an die algebraische Spezifikation von Datentypen (später im Detail!) □

graphische Notation durch *Signaturgraph* (Abb. 2.2)

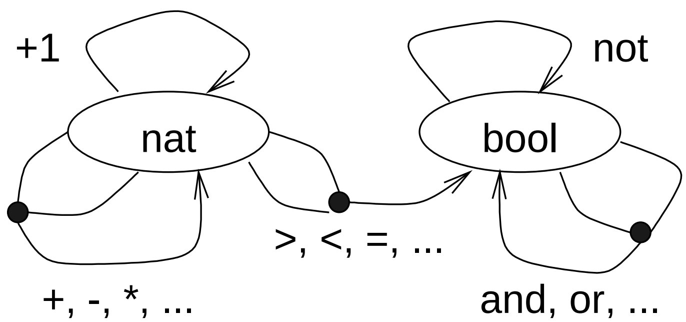


Abbildung 2.2.: Signaturgraph für natürliche Zahlen

2.3.3. Der Datentyp `bool`

- **boolean**, auch **bool**: Datentyp der Wahrheitwerte
- Werte: {**true**, **false**}
- Operationen:
  - $\neg$, auch **not**: logische Negation
  - $\wedge$, auch **and**: logisches Und
  - $\vee$, auch **or**: logisches Oder
  - $\Longrightarrow$: Implikation

Operationen definiert durch *Wahrheitstafeln*:

	$\neg$
false	**true**
true	**false**

$\wedge$	**false**	**true**
false	**false**	**false**
true	**false**	**true**

$\lor$	**false**	**true**
false	**false**	**true**
true	**true**	**true**

$p \implies q$ ist definiert als $\neg p \lor q$

2.3.4. Der Datentyp integer

- **integer**, auch **int**: Datentyp der ganzen Zahlen
- Werte: $\{\ldots, -2, -1, 0, 1, 2, 3, 4, \ldots\}$
- Operationen $+, -, *, \div,$ **mod**, **sign**, $>, <, \ldots$.
  - $+$: **int** $\times$ **int** $\to$ **int**
    Addition, $4 + 3 \mapsto 7$
  - **mod**: **int** $\times$ **int** $\to$ **int**
    Rest bei Division, 19 **mod** $7 \mapsto 5$
  - **sign**: **int** $\to \{-1, 0, 1\}$
    Vorzeichen, **sign**$(7) \mapsto 1$, **sign**$(0) \mapsto 0$, **sign**$(-7) \mapsto -1$,
  - $>$: **int** $\times$ **int** $\to$ **bool**
    Größerrelation: $4 > 3 \mapsto$ **true**
  - weiter übliche Operationen, insbesondere in folgenden Beispielen **abs** zur Berechnung des Absolutbetrags, **odd** und **even**[1] für ungerade bzw. gerade Zahlen, ...
  - ...

$\mapsto$ bedeutet hierbei "wird ausgewertet zu"

Bemerkung 2.2 *In einem Rechner sind stets nur endlich viele* **integer**-*Werte definiert.* □

Wir verwenden das Zeichen $\perp$, wenn das Ergebnis einer Auswertung *undefiniert* ist:

$$\begin{aligned} 19 \div 0 &\mapsto \perp \\ 2 * \perp &\mapsto \perp \\ \perp + 3 &\mapsto \perp \end{aligned}$$

[1] Merkregel: **odd** hat drei Buchstaben und steht für ungerade, **even** hat vier Buchstaben und steht für gerade!

2.3.5. Felder und Zeichenketten

Weitere Datentypen werden im Laufe der Vorlesung behandelt.

Für die praktischen Übungen und Beispiele sind folgende Datentypen relevant:

- **char**: Zeichen in Texten $\{A, .., Z, a, ..\}$ mit Operation $=$
- **string**: Zeichenketten über **char**

  Operationen:

  - Gleichheit $=$
  - Konkatenation ('Aneinanderhängen'): $+$
  - Selektion des i-ten Zeichens: $s[i]$
  - Länge: **length**
  - jeder Wert aus **char** wird als ein **string** der Länge 1 aufgefaßt, wenn er wie folgt notiert wird: `'A'`
  - **empty** als leerer **string**
  - weitere sinnvolle Operatoren, etwa **substring**
- **array**: 'Felder' mit Werten eines Datentyps als Eintrag, Ausdehnung fest vorgegeben

  Definition:

  ```
  array 1..3 of int;
  array 1..3, 1..3 of int;
  ```

  Operationen:

  - Gleichheit $=$
  - Selektion eines Elements: $A[n]$ oder $A[1, 2]$
  - Kontruktion eines Feldes: $(1, 2, 3)$ oder $((1, 2, 3), (3, 4, 5), (6, 7, 8))$

Letzterer Datentyp ist genau genommen ein *Datentypkonstruktor*, da mit ihm Felder verschiedener Ausdehnung, Dimensionalität und Basisdatentypen gebildet werden können. Diese Art von Datentypen wird später noch genauer behandelt.

2.3.6. Datentypen in Java *

zusätzliche Dokumente von Kai-Uwe Sattler

2.4. Terme

2.4.1. Ausdrücke / Terme

Wie setzt man Grundoperationen zusammen?

- Bildung von Termen:

$$7 + (9 + 4) * 8 - 14$$

$$13 - \mathbf{sign}(-17) * 15$$

$$\neg\mathbf{true} \vee (\mathbf{false} \vee (\neg\mathbf{false} \wedge \mathbf{trues}))$$

  Man beachte: Klammern und Prioritäten zur Festlegung der Auswertungsabfolge!

- Bedingte Terme:

  if b **then** t **else** u **fi**

  b boolescher Term, t und u zwei Terme gleicher Sorte

- Auswertung bedingter Terme (am Beispiel):

  if true then t **else** u **fi** $\mapsto t$
  if false then t **else** u **fi** $\mapsto u$
  if true then 3 **else** $\perp$ **fi** $\mapsto 3$
  if false then 3 **else** $\perp$ **fi** $\mapsto \perp$

  Im Gegensatz zu Operationen erzwingt ein Teilausdruck, der undefiniert ist, nicht automatisch die Undefiniertheit des Gesamtterms!

Notwendig: Formalisierung der Bildung und Auswertung von Termen (hier nur für **int**-Terme):

Definition 2.1 **int**-Terme

1. Die **int**-Werte $..,-2,-1,0,1,...$ sind **int**-Terme.
2. Sind t, u **int**-Terme, so sind auch $(t+u)$, $(t-u)$, **sign**(t) **int**-Terme.
3. Ist b ein **bool**-Term, und sind t, u **int**-Terme, so ist auch **if** b **then** t **else** u **fi** ein **int**-Term.
4. Nur die durch diese Regeln gebildeten Zeichenketten sind **int**-Terme. □

Diese Regeln ergeben vollständig geklammerte Ausdrücke. Klammereinsparungsregeln:

- Vorrangregeln: Punktrechnung vor Strichrechnung, $\neg$ vor $\wedge$ vor $\vee$, **if**-Konstruktor schwächer als alle anderen, etc
- Ausnutzung von Assoziativgesetzen (= Klammern unnötig da identischer Wert als Ergebnis)

$$(((\mathbf{abs}((7*9)+7)-6)+8)-17) \text{ kurz } \mathbf{abs}(7*9+7)-6+8-17$$

Der Multiplikationsoperator wird oft ebenfalls eingespart:

$$2*(2+3) \text{ kurz } 2(2+3)$$

2.4.2. Algorithmus zur Termauswertung

Auswertung von innen nach außen (in der Klammerstruktur), bei bedingten Termen wird zuerst die Bedingung ausgewertet.

$$\begin{array}{ll}
 & 1+\ \mathbf{if}\ \mathbf{true} \vee \neg\mathbf{false}\ \mathbf{then}\ 7*9+7-6\ \mathbf{else}\ \mathbf{abs}(3-8))\ \mathbf{fi} \\
\mapsto & 1+\ \mathbf{if}\ \mathbf{true} \vee \mathbf{true}\ \mathbf{then}\ 7*9+7-6\ \mathbf{else}\ \mathbf{abs}(3-8))\ \mathbf{fi} \\
\mapsto & 1+\ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ 7*9+7-6\ \mathbf{else}\ \mathbf{abs}(3-8))\ \mathbf{fi} \\
\mapsto & 1+7*9+7-6 \\
\mapsto & 1+63+7-6 \\
\mapsto & 64+7-6 \\
\mapsto & 71-6 \\
\mapsto & 65
\end{array}$$

Der Algorithmus ist nicht-deterministisch, determiniert, terminierend.

Beispiel für Nicht-Determinismus: $(7+9)*(4+10)$ kann über $16*(4+10)$ zu $(7+9)*14$ zu $16*14$ ausgewertet werden. Man kann den Algorithmus deterministisch machen, indem z.B. bie mehren Möglichkeiten jeweils immer der am weitesten links stehende auswertbare Teilterm ausgewertet wird.

2.5. Überblick über Algorithmenparadigmen

Zwei grundlegende Arten Algorithmen zu notieren:

- applikativ: Verallgemeinerung der Funktionsauswertung
- imperativ: basierend auf einem einfachen Maschinenmodell mit gespeicherten und änderbaren Werten

Weitere Paradigmen (logisch, objektorientiert, agentenorientiert) später in der Vorlesung.

Zu den Paradigmen korrespondieren jeweils Programmiersprachen, die diesen Ansatz realisieren. Moderne Programmiersprachen vereinen oft Ansätze mehrerer Paradigmen.

- Java: objektorientiert, imperativ, Elemente von applikativ.

2.6. Applikative Algorithmen

Idee: Definition zusammengesetzter Funktionen durch Terme:

$$f(x) = 5x + 1$$

In diesem Abschnitt beschränken wir uns zur Vereinfachung der Definitionen auf Funktionen über **int** und **bool**!

2.6.1. Terme mit Unbestimmten

Gegeben sind zwei (unendliche, abzählbare) Mengen von Symbolen ("Unbestimmte"):

- $x, y, z, ..$ vom Typ **int**
- $q, p, r, ...$ vom Typ **bool**

Ein Term mit Unbestimmten wird analog zu Termen ohne Unbestimmte gebildet:

$$x, x - 2, 2x + 1, (x + 1)(y - 1)$$

und

$$p, p \wedge \texttt{true}, (p \vee \texttt{true}) \implies (q \vee \texttt{false})$$

2.6.2. Funktionsdefinitionen

Definition 2.2 Sind $v_1, ..., v_n$ Unbestimmte vom Typ $\tau_1, .., \tau_n$ (**bool** oder **int**) und ist $t(v_1, ..., v_n)$ ein Term, so heißt

$$f(v_1, ..., v_n) = t(v_1, ..., v_n)$$

eine *Funktionsdefinition* vom Typ τ. τ ist dabei der *Typ* des Terms. □

- f heißt *Funktionsname*
- $v_1, ..., v_n$ heissen *formale Parameter*
- $t(v_1, ..., v_n)$ heißt *Funktionsausdruck*

Beispiel 2.9 Beispiele für Funktionsdefinitionen:

1. $f(p, q, x, y) =$ **if** $p \vee q$ **then** $2x + 1$ **else** $3y - 1$ **fi**
2. $g(x) =$ **if even**(x) **then** $x \div 2$ **else** $3x - 1$ **fi**
3. $h(p, q) =$ **if** p **then** q **else false fi** □

2.6.3. Auswertung von Funktionen

Eine Funktionsdefinition gemäß Definition 2.2 definiert eine Funktion

$$f : \tau_1 \times ... \times \tau_n \to \tau$$

Sind $a_1, .., a_n$ Werte vom Typ $\tau_1, .., \tau_n$, so ersetzt man bei der Auswertung von $f(a_1, .., a_n)$ im definierenden Term jedes Vorkommen von v_i durch a_1 und wertet $t(a_1, .., a_n)$ aus.

- $a_1, ..., a_n$ heissen *aktuelle Parameter*
- $f(a_1, ..., a_n)$ heißt *Funktionsaufruf*

Beispiel 2.10 Beispiele (Erweiterung von Beispiel 2.9):

1. $f(p, q, x, y) =$**if** $p \vee q$ **then** $2x + 1$ **else** $3y - 1$ **fi**

$$f : \textbf{bool} \times \textbf{bool} \times \textbf{int} \times \textbf{int} \to \textbf{int}$$

$$f(\textbf{true}, \textbf{true}, 3, 4) = 7$$

2. $g(x) =$**if even**(x) **then** $x \div 2$ **else** $3x - 1$ **fi**

$$g : \textbf{int} \to \textbf{int}$$

$$g(2) = 1, g(3) = 8$$

3. $h(p, q) =$**if** p **then** q **else false fi**

$$h : \textbf{bool} \times \textbf{bool} \to \textbf{bool}$$

$$h(\textbf{false}, \textbf{false}) = \textbf{false}$$

Bemerkung: $h(p, q) = p \wedge q$ □

2.6.4. Erweiterung der Funktionsdefinition

Erweiterung der Klassen der Terme und Funktionsdefinitionen: Aufrufe definierter Funktionen dürfen als Terme verwendet werden.

Beispiel 2.11 Erweiterte Funktionsdefinitionen:

$$\begin{aligned}
f(x, y) &= \textbf{if } g(x, y) \textbf{ then } h(x + y) \textbf{ else } h(x - y) \textbf{ fi} \\
g(x, y) &= (x = y) \vee \textbf{odd}(y) \\
h(x) &= j(x + 1) * j(x - 1) \\
j(x) &= 2x - 3
\end{aligned}$$

Beispiel für Auswertung:

$$\begin{array}{rcl} f(1,2) & \mapsto & \textbf{if } g(1,2) \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\ & \mapsto & \textbf{if } 1 = 2 \vee \textbf{odd}(2) \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\ & \mapsto & \textbf{if } 1 = 2 \vee \textbf{false} \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\ & \mapsto & \textbf{if } \textbf{false} \vee \textbf{false} \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\ & \mapsto & \textbf{if } \textbf{false} \textbf{ then } h(1+2) \textbf{ else } h(1-2) \textbf{ fi} \\ & \mapsto & h(1-2) \\ & \mapsto & h(-1) \\ & \mapsto & j(-1+1) * j(-1-1) \\ & \mapsto & j(0) * j(-1-1) \\ & \mapsto & j(0) * j(-2) \\ & \mapsto & (2*0-3) * j(-2) \\ & \mapsto^* & (-3)*(-7) \\ & \mapsto & 21 \end{array}$$

Mit $\mapsto^*$ bezeichnen wir die konsekutive Ausführung mehrerer elementarer Termauswertungsschritte. □

2.6.5. Rekursive Funktionsdefinitionen

Eine Funktionsdefinition f heißt *rekursiv*, wenn direkt (oder indirekt über andere Funktionen) ein Funktionsaufruf $f(..)$ in ihrer Definition auftritt.

Beispiel 2.12 Beispiel Rekursion

$$\begin{array}{rcl} f(x,y) & = & \textbf{if } x = 0 \textbf{ then } y \textbf{ else } (\\ & & \textbf{if } x > 0 \textbf{ then } f(x-1,y)+1 \textbf{ else } -f(-x,-y) \textbf{ fi}) \textbf{ fi} \end{array}$$

Auswertungen

$$\begin{array}{rcl} f(0,y) & \mapsto & y \text{ für alle } y \\ f(1,y) & \mapsto & f(0,y)+1 \mapsto y+1 \\ f(2,y) & \mapsto & f(1,y)+1 \mapsto (y+1)+1 \mapsto y+2 \\ & \ldots & \\ f(n,y) & \mapsto & n+y \text{ für alle } n \in \textbf{int}, n > 0 \\ f(-1,y) & \mapsto & -f(1,-y) \mapsto -(1-y) \mapsto y-1 \\ & \ldots & \\ f(x,y) & \mapsto & x+y \text{ für alle } x,y \in \textbf{int} \end{array}$$

□

2.6.6. Applikative Algorithmen

Definition 2.3 Ein *applikativer Algorithmus* ist eine Menge von Funktionsdefinitionen

$$\begin{aligned} f_1(v_{1,1}, ..., v_{1,n_1}) &= t_1(v_{1,1}, ..., v_{1,n_1}), \\ &\vdots \\ f_m(v_{m,1}, ..., v_{m,n_m}) &= t_m(v_{m,1}, ..., v_{m,n_m}). \end{aligned}$$

Die erste Funktion f_1 wird wie beschrieben ausgewertet und ist die Bedeutung (Semantik) des Algorithmus. □

Applikative Algorithmen sind die Grundlage einer Reihe von universellen Programmiersprachen, wie APL, LISP, Scheme, etc.

Beispiel 2.13 Beispiel Definiertheit

$$f(x) = \texttt{\textbf{if}}\ x = 0\ \texttt{\textbf{then}}\ 0\ \texttt{\textbf{else}}\ f(x-1)\ \texttt{\textbf{fi}}$$

Auswertungen:

$$\begin{aligned} f(0) &\mapsto 0 \\ f(1) &\mapsto f(0) \mapsto 0 \\ f(x) &\mapsto 0 \text{ für alle } x \geq 0 \\ f(-1) &\mapsto f(-2) \mapsto ... \text{ Auswertung terminiert nicht!} \end{aligned}$$

Also gilt

$$f(x) = \begin{cases} 0 & \text{falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}$$

□

Eine (möglicherweise) für einige Eingabewertekombinationen *undefinierte* Funktion heißt *partielle Funktion*.

2.6.7. Beispiele für applikative Algorithmen

Beispiel 2.14 (Fakultät)

$$x! = x(x-1)(x-2) \cdots 2 * 1 \text{ für } x > 0$$

Bekannte Definition: $0! = 1, x! = x * (x-1)!$.

Problem: negative Eingabewerte.

1. Lösung:

$$fac(x) \quad = \quad \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * fac(x-1) \textbf{ fi}$$

   Bedeutung:

$$fac(x) = \begin{cases} x! & \text{falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}$$

2. Lösung:

$$fac(x) \quad = \quad \textbf{if } x \leq 0 \textbf{ then } 1 \textbf{ else } x * fac(x-1) \textbf{ fi}$$

   Bedeutung:

$$fac(x) = \begin{cases} x! & \text{falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

□

Beispiel 2.15 Fibonacci-Zahlen:

$$f_0 = f_1 = 1, f_i = f_{i-1} + f_{i-2} \text{ für } i > 0$$

$$fib(x) \quad = \quad \textbf{if } (x = 0) \vee (x = 1) \textbf{ then } 1 \textbf{ else } fib(x-2) + fib(x-1) \textbf{ fi}$$

Bedeutung:

$$fib(x) = \begin{cases} x\text{-te Fibonacci-Zahl} & \text{falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

□

Beispiel 2.16 Produkt nur unter Verwendung der Addition. Ausnutzung folgender Regeln:

$$\begin{aligned} 0 * y &= 0 \\ x * y &= (x-1) * y + x \text{ für } x > 0 \\ x * y &= -((-x) * y) \text{ für } x < 0 \end{aligned}$$

$$\begin{array}{lll} prod(x, y) = & \textbf{if } x = 0 & \textbf{then } 0 \textbf{ else} \\ & \textbf{if } x > 0 & \textbf{then } prod(x-1, y) + y \\ & & \textbf{else } -prod(-x, y) \quad \textbf{fi fi} \end{array}$$

□

Beispiel 2.17 Größter gemeinsamer Teiler ggT.
Für $0 < x$ und $0 < y$ gilt:

$$\begin{aligned} ggT(x, x) &= x \\ ggT(x, y) &= ggT(y, x) \\ ggT(x, y) &= ggT(x, y - x) \text{ für } x < y \end{aligned}$$

$$
\begin{array}{lll}
ggT(x,y) & \textbf{if } (x \leq 0) \vee (y \leq 0) & \textbf{then } ggT(x,y) \textbf{ else} \\
 & \textbf{if } x = y & \textbf{then } x \textbf{ else} \\
 & \textbf{if } x > y & \textbf{then } ggT(y,x) \\
 & & \textbf{else } ggT(x, y-x) \quad \textbf{fi fi fi}
\end{array}
$$

ggT ist korrekt für positive Eingaben, bei negativen Eingaben ergeben sich nicht abbrechende Berechnungen (undefinierte Funktion).

$$ggT(39,15) \mapsto ggT(15,39) \mapsto ggT(15,24) \mapsto ggT(15,9) \mapsto ggT(9,15) \mapsto ggT(9,6)$$

$$\mapsto ggT(6,9) \mapsto ggT(6,3) \mapsto ggT(3,6) \mapsto ggT(3,3) \mapsto 3$$

Dieses Berechnungsschema ist Formalisierung des Originalverfahrens von Euklid (Euklid: Elemente, 7. Buch, Satz 2; ca 300 v.Chr.). □

Das folgende Beispiel demonstriert den Einsatz mehrere Funktionen:

Beispiel 2.18 Test, ob eine Zahl gerade ist: $even(x)$

$$
\begin{array}{rcl}
even(0) & = & \textbf{true} \\
odd(0) & = & \textbf{false} \\
even(x+1) & = & odd(x) \\
odd(x+1) & = & even(x)
\end{array}
$$

$$
\begin{array}{lll}
even(x) = & \textbf{if } x = 0 & \textbf{then true else} \\
 & \textbf{if } x > 0 & \textbf{then } odd(x-1) \\
 & & \textbf{else } odd(x+1) \quad \textbf{fi fi}
\end{array}
$$

$$
\begin{array}{lll}
odd(x) = & \textbf{if } x = 0 & \textbf{then false else} \\
 & \textbf{if } x > 0 & \textbf{then } even(x-1) \\
 & & \textbf{else } even(x+1) \quad \textbf{fi fi}
\end{array}
$$

Algorithmus für odd durch Vertauschen der Reihenfolge. □

Beispiel 2.19 Primzahltest.

$$
\begin{array}{lll}
prim(x) & \textbf{if abs}(x) \leq 1 & \textbf{then false else} \\
 & \textbf{if } x < 0 & \textbf{then } prim(-x) \\
 & & \textbf{else } pr(2,x) \quad \textbf{fi fi}
\end{array}
$$

$$
\begin{array}{lll}
pr(x,y) & \textbf{if } x \geq y & \textbf{then true else} \\
 & & \textbf{else } (y \textbf{ mod } x) \neq 0 \wedge pr(x+1,y) \quad \textbf{fi}
\end{array}
$$

- Hilfsfunktion $pr(x,y)$ ist wahr g.d.w. y ist durch keine Zahl z, $x \leq z < y$, teilbar
- $prim(x) \Leftrightarrow (x > 1) \wedge pr(2,x)$

- Lösung nicht sehr effizient....

  Verbesserungen:

  - Ersetze $x \geq y$ durch $x * x \geq y$ (kleinster Teiler muß kleiner als Quadratwurzel sein!).
  - Ersetze $pr(x+1, y)$ durch **if** $x = 2$ **then** $pr(3, y)$ **else** $pr(x+2, y)$ **fi**. □

Beispiel 2.20 Beispiel 'Terminierung'

$$f(x) = \textbf{if } f(x) = 0 \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi}$$

Aufgrund der Termauswertung (unendliche Berechnung) gilt:

$$f(x) = \bot \quad \text{für alle} \quad x \in \textbf{int}$$

Nich möglicher Schluß: Da $f(x) = \bot$, gilt etwa $f(7) \neq 0$ und damit der zweite Zweig der Auswahl $f(7) = 0$!

Vorsicht beim 'Rechnen' mit $\bot$!! Es gilt immer

$$\textbf{if } \bot \textbf{ then } t \textbf{ else } u \textbf{ fi} = \bot$$

Paßt auch zu $\bot$ als 'unendlich lange Berechnung'! □

Beispiel 2.21 Beispiel 'komplexe Bedeutung'

$$f(x) = \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } f(f(x+11)) \textbf{ fi}$$

Beispielberechnungen:

$$\begin{array}{rcl} f(100) & \mapsto & f(f(111)) \mapsto f(101) \mapsto 91 \\ f(99) & \mapsto & f(f(110)) \mapsto f(100) \mapsto \ldots \mapsto 91 \\ \ldots & \mapsto & \ldots \mapsto 91 \end{array}$$

Tatsächlich gilt:

$$f(x) = \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } 91 \textbf{ fi}$$

Beweis nicht ganz einfach! □

Beispiel 2.22 Beispiel 'knifflige Bedeutung'

$$f(x) = \textbf{if } x = 1 \textbf{ then } 1 \textbf{ else } f(g(x)) \textbf{ fi}$$

$$g(x) = \textbf{if } \textbf{even}(x) \textbf{ then } x/2 \textbf{ else } 3x + 1 \textbf{ fi}$$

Es gilt:

$$f(x) = \bot \quad \text{für} \quad x \leq 0$$

Beispielberechnungen:

$$
\begin{array}{lcl}
f(1) & \mapsto & 1 \\
f(2) & \mapsto & f(1) \mapsto 1 \\
f(3) & \mapsto & f(10) \mapsto f(5) \mapsto f(16) \mapsto f(8) \mapsto f(4) \mapsto f(2) \mapsto 1 \\
f(4) & \mapsto & ... \mapsto 1 \\
\dots & &
\end{array}
$$

Bisher unbewiesen, ob gilt:

$$f(x) = \mathbf{if}\ x > 0\ \mathbf{then}\ 1\ \mathbf{else}\ \perp\ \mathbf{fi}$$

Problem: kann man entscheiden ob zwei Algorithemn äquivalent sind, d.h., ob sie das selbe berechnen? □

Beispiel 2.23 Ackermann-Funktion.

$$
\begin{array}{lll}
f(x,y) & \mathbf{if}\ x \leq 0 & \mathbf{then}\ y+1\ \mathbf{else} \\
 & \mathbf{if}\ y \leq 0 & \mathbf{then}\ f(x-1,1) \\
 & & \mathbf{else}\ f(x-1, f(x, y-1))\quad \mathbf{fi}\ \mathbf{fi}
\end{array}
$$

- Werte wachsen unglaublich schnell an:

$$
\begin{array}{lcl}
f(0,0) & \mapsto & 1 \\
f(1,0) & \mapsto & f(0,1) \mapsto 2 \\
f(1,1) & \mapsto & f(0,f(1,0)) \mapsto f(0,f(0,1)) \mapsto f(0,2) \mapsto 3 \\
f(2,2) & \mapsto & f(1,f(2,1)) \mapsto f(1,f(1,f(2,0))) \mapsto f(1,f(1,f(1,1))) \\
 & & ... \mapsto f(1,5) \mapsto ... f(0,6) \mapsto 7 \\
f(3,3) & \mapsto & 61 \\
f(4,4) & \mapsto & 2^{2^{2^{2^{2^{2^{2}}}}}} - 3 = 2^{2^{2^{65536}}} - 3
\end{array}
$$

Ackermann-Funktion ist relevant in Komplexitätsbetrachtungen. □

2.7. Imperative Algorithmen

- Verbreiteteste Art, Algorithmen für Computer zu formulieren
- Basiert auf den Konzepten Anweisung und Variablen
- Wird durch Programmiersprachen C, PASCAL, FORTRAN, COBOL,... realisiert, hier nur Darstellung der Grundprinzipien

2.7.1. Grundlagen

Definition 2.4 Variablen

- Eine *Variable* besteht aus einem Namen (z.B. X) und einem veränderlichen *Wert* (momentaner Wert ist dann $|X|$)
- Ist t ein Term ohne Unbestimmte und $w(t)$ sein Wert, dann heißt das Paar $x := t$ eine *Wertzuweisung*. Ihre Bedeutung ist festgelegt durch

  Nach Ausführung von $x := t$ gilt $x = w(t)$.

  Vor der Ausführung einer Wertzuweisung gilt $x = \perp$ (undefiniert). □

Beispiele:

$$\begin{array}{ll} X := 7 & F := true \\ X := (3-7)*9 & Q := \neg(true \vee \neg false) \vee \neg\neg true \end{array}$$

Definition 2.5 Zustände

- Ist $\underline{X} = \{X_1, X_2, \dots\}$ eine Menge von Variablen(-namen), von denen jede Werte aus der Wertemenge W haben kann (alle Variablen vom gleichen Typ), dann ist ein Zustand Z eine partielle Abbildung

  $Z\colon \underline{X} \to W$ (Zuordnung des momentanen Wertes)

- Ist $Z\colon \underline{X} \to W$ ein Zustand und wählt man eine Variable X und einen Wert w, so ist der *transformierte Zustand* $Z_{\langle X\leftarrow w\rangle}$ wie folgt definiert:

$$\begin{aligned} Z_{\langle X\leftarrow w\rangle}\colon \underline{X} &\to w \\ Y &\mapsto \begin{cases} w, & \text{falls } X = Y \\ Z(Y), & \text{sonst} \end{cases} \end{aligned}$$

  Hier bezeichnet Z den *alten Zustand* vor der Ausführung der Wertzuweisung; $Z_{\langle X\leftarrow w\rangle}$ ist der *neue Zustand* nach Ausführung der Wertzuweisung.[2]

□

Anweisungen (Befehle)

Mit einer Anweisung α wird ein Zustand Z in einen (anderen) Zustand $[\![\alpha]\!](Z)$ überführt. Bei einer Wertzuweisung $X := t$ ist der neue Zustand

$$[\![X := t]\!](Z) = Z_{\langle X\leftarrow w(t)\rangle}$$

[2] In einer ersten Version des Skriptes war $Z_{\langle X\leftarrow w\rangle}$ als $Z\langle X \leftarrow w\rangle$ notiert. Da dies zur Verwirrung führte — der Ausdruck wurde von einigen Lesern nicht als *ein* Zustand erkannt — wurde in der Endversion die Subskript-Notation gewählt.

Ausdrücke

Um Werte, die Variablen als Zwischenergebnisse zugewiesen wurden, später wiederverwenden zu können, muß man Werte aufrufen bzw. auslesen können.

Definition 2.6 *Ausdrücke* entsprechen im wesentlichen den Termen einer applikativen Sprache, jedoch stehen an der Stelle von Unbekannten Variablen. □

Die Auswertung von Termen ist zustandsabhängig, an der Stelle der Variablen wird ihr Wert im gegebenen Zustand gesetzt.

Beispiel 2.24 gegebener Term $2X + 1$
Wert im Zustand Z ist $2 \cdot Z(X) + 1$ □

Der so bestimmte Wert eines Ausdrucks $t(X_1, \ldots, X_n)$ wird mit $Z(t(X_1, \ldots, X_n))$ bezeichnet.

Beispiel 2.25 Wert eines Terms:

$$Z(2 * X + 1) = 2 * Z(X) + 1$$

□

Man kann so auch Wertzuweisungen verwenden, auf deren rechter Seite ein Ausdruck mit Variablen steht:

$$X := t(X_1, \ldots, X_n)$$

Der transformierte Zustand ist

$$[\![X := t(X_1, \ldots, X_n)]\!](Z) = Z_{\langle X \leftarrow Z(t(X_1, \ldots, X_n)) \rangle}$$

Beispiel 2.26 (Anweisungen)
Wir zeigen die Transformationen für zwei elementare Anweisungen α_1 und α_2.

$$\alpha_1 = (X := 2 \cdot Y + 1) \text{ Transformation in } [\![\alpha_1]\!](Z) = Z_{\langle X \leftarrow 2Z(Y)+1 \rangle}$$
$$\alpha_2 = (X := 2 \cdot X + 1) \text{ Transformation in } [\![\alpha_2]\!](Z) = Z_{\langle X \leftarrow 2Z(X)+1 \rangle}$$

Bei der letzten Anweisung handelt es sich also *nicht* um eine rekursive Gleichung!
□

Wertzuweisungen bilden die *einzigen* elementaren Anweisungen; aus ihnen werden komplexe Anweisungen zusammengesetzt, die Algorithmen definieren.

2.7.2. Komplexe Anweisungen

1. Folge (Sequenz): Sind α_1 und α_2 Anweisungen, so ist $\alpha_1;\alpha_2$ auch eine Anweisung.

   Zustandstransformation

   $$[\![\alpha_1;\alpha_2]\!](Z) = [\![\alpha_2]\!]([\![\alpha_1]\!](Z))$$

2. Auswahl (Selektion): Sind α_1 und α_2 Anweisungen und B ein Boolescher Ausdruck, so ist

   if B **then** α_1 **else** α_2 **fi**

   eine Anweisung.

   Transformation:

   $$[\![\texttt{if } B \texttt{ then } \alpha_1 \texttt{ else } \alpha_2 \texttt{ fi}]\!](Z) = \begin{cases} [\![\alpha_1]\!](Z) & \text{falls } Z(B) = \texttt{true} \\ [\![\alpha_2]\!](Z) & \text{falls } Z(B) = \texttt{false} \end{cases}$$

   Hier wird vorausgesetzt, daß $Z(B)$ definiert ist. Ansonsten ist die Bedeutung der Auswahlanweisung undefiniert.

3. Wiederholung (Iteration): Ist α Anweisung und B Boolscher Ausdruck, so ist

   while B **do** α **od**

   eine Anweisung.

   Transformation:

   $$[\![\texttt{while } B \texttt{ do } \alpha \texttt{ od}]\!]\ (Z) = \begin{cases} Z, \text{falls } Z(B) = \texttt{false} \\ [\![\texttt{while } B \texttt{ do } \alpha \texttt{ od}]\!]([\![\alpha]\!](Z)), \text{sonst} \end{cases}$$

   Ist $Z(B)$ undefiniert, so ist die Bedeutung dieser Anweisung ebenfalls undefiniert.

Bemerkung 2.3 (Umsetzung in Programmiersprachen)

- In realen imperativen Programmiersprachen gibt es fast immer diese Anweisungen, meistens jedoch viel mehr.
- **while**-Schleifen sind rekursiv definiert, sie brauchen nicht zu terminieren.
- Die Verwendung von **if**-**fi** ist wegen der klaren Klammerung sauberer.
- In Algol68 gibt es auch “nichtdeterministische” Versionen der Sequenz
- Jedoch sind bereits Programmiersprachen mit diesen Sprachelementen *universell* (siehe weiter unten). □

Wir beschränken uns im folgenden auf die Datentypen **bool** und **int**.

2.7.3. Syntax imperativer Algorithmen

Imperative Algorithmen haben folgende Syntax:

$<$ Programmname $>$:
var $X, Y, \ldots$: **int**; $P, Q, \ldots$: **bool** $\Leftarrow$ Variablen-Deklaration
input $X_1, \ldots, X_n$; $\Leftarrow$ Eingabe-Variablen
α; $\Leftarrow$ Anweisung(en)
output $Y_1, \ldots, Y_m$. $\Leftarrow$ Ausgabe-Variablen

2.7.4. Semantik imperativer Algorithmen

Die Bedeutung (Semantik) eines imperativen Algorithmus ist eine partielle Funktion.

$$\boxed{\begin{array}{rcl} [\![\text{PROG}]\!]: & & W_1 \times \ldots \times W_n \mapsto V_1 \times \ldots \times V_m \\ [\![\text{PROG}]\!](w_1, \ldots, w_n) & = & (Z(Y_1), \ldots, Z(Y_m)) \\ \text{wobei } Z & = & [\![\alpha]\!](Z_0), \\ Z_0(X_i) & = & w_i,\ i = 1, \ldots, n, \\ \text{und } Z_0(Y) & = & \perp \text{ für alle Variablen} \\ & & \quad Y \neq X_i,\ i = 1, \ldots, n \end{array}}$$

wobei gilt

PROG	Programmname
$W_1, \ldots, W_n$	Wertebereiche der Typen von $X_1, \ldots, X_n$
$V_1, \ldots, V_m$	Wertebereiche der Typen von $Y_1, \ldots, Y_m$

2.7.5. Beispiele für imperative Algorithmen

Beispiel 2.27 Fakultät-Funktion $0! = 1$, $x! = x * (x-1)!$ für $x > 0$

```
FAC: var  X,Y: int;
     input  X;
     Y:=1;  while  X>1  do  Y:=Y*X;  X:=X-1 od
     output  Y.
```

$$\text{Hier ist } [\![FAC]\!]\ (x) \ = \begin{cases} x! \text{ für } x \geq 0 \\ 1 \ \text{ sonst} \end{cases}$$

Setzt man als Bedingung für die **while**-Schleife “$X \neq 0$”, so erhält man die Version

$$[\![FAC]\!]\ (x) \ = \begin{cases} x! \text{ für } x \geq 0 \\ \perp \ \text{ sonst} \end{cases}$$

□

Beispiel 2.28 Semantik imperativer Algorithmen am Beispiel.

Die folgenden Ausführungen verdeutlichen anhand des imperativen Algorithmus FAC aus Beispiel 2.27 die Festlegungen der Definition der formalen Semantik. Gesucht ist hierbei das Ergebnis von $FAC(3)$.

Wir verwenden die Abkürzung **while** β für die Zeile

```
while  X>1  do  Y:=Y*X;  X:=X-1 od
```

Es gilt:

$$[\![FAC]\!] : \texttt{int} \rightarrow \texttt{int}$$

mit

$$[\![FAC]\!](w) = Z(Y)$$

Der Endzustand Z ist dabei definiert als

$$Z = [\![\alpha]\!](Z_0)$$

wobei α die Folge aller Anweisungen des Algorithmus ist. Der intiale Zustand Z_0 ist definiert als $Z_0 = (X = w, Y = \perp)$. Zustände werden wir im folgenden abkürzend ohne Variablennamen notieren, also $Z_0 = (w, \perp)$.

Gesucht sei nun $[\![FAC]\!](3)$. Dazu müssen wir den Endzustand Z bestimmen. Die Bestimmung von Z ist in Abbildung 2.3 ausgeführt.

Bemerkungen zu Abbildung 2.3:

- der Übergang von der 3ten auf die 4te Zeile folgt der Definition der Sequenz.
- Nur in der 5ten Zeile wurde die Wertzuweisung formal umgesetzt, später einfach verkürzt ausgerechnet.
- In der 7ten Zeile haben wir die Originaldefinition der Iteration eingesetzt (nur mit α' statt α, da α bereits verwendet wurde).

  Im Beispiel gilt
  $$\alpha' = \{Y := Y * X; X : X - 1\}$$
- Das Z in der 7ten und 8ten Zeile steht für den Zustand $(3, 1)$ (in späteren Zeilen analog für den jeweils aktuellen Zustand).
- Man beachte: *Die Ausführung einer While-Schleife erfolgt analog einer rekursiven Funktionsdefinition!*

Damit gilt:

$$[\![FAC]\!](3) = Y(Z) = Y(X = 1, Y = 6) = 6$$

□

$$
\begin{aligned}
Z &= [\![\alpha]\!](Z_0) && (2.1)\\
&= [\![\alpha]\!](3,\perp) && (2.2)\\
&= [\![Y := 1; \mathbf{while}\ \beta]\!](3,\perp) && (2.3)\\
&= [\![\mathbf{while}\ \beta]\!]([\![Y := 1]\!](3,\perp)) && (2.4)\\
&= [\![\mathbf{while}\ \beta]\!]((3,\perp)_{\langle Y \leftarrow 1\rangle}) && (2.5)\\
&= [\![\mathbf{while}\ \beta]\!](3,1) && (2.6)\\
&= \begin{cases} Z, \text{ falls } Z(B) = \mathbf{false} \\ [\![\mathbf{while}\ B\ \mathbf{do}\ \alpha'\ \mathbf{od}]\!]([\![\alpha']\!](Z)), \text{ sonst} \end{cases} && (2.7)\\
&= \begin{cases} (3,1), \text{ falls } Z(X>1) = (3>1) = \mathbf{false} \\ [\![\mathbf{while}\ \beta]\!]([\![Y := Y * X; X := X-1]\!](Z)), \text{ sonst} \end{cases} && (2.8)\\
&= [\![\mathbf{while}\ \beta]\!]([\![Y := Y * X; X := X-1]\!](3,1)) && (2.9)\\
&= [\![\mathbf{while}\ \beta]\!]([\![X := X-1]\!]([\![Y := Y * X]\!](3,1))) && (2.10)\\
&= [\![\mathbf{while}\ \beta]\!]([\![X := X-1]\!](3,3)) && (2.11)\\
&= [\![\mathbf{while}\ \beta]\!](2,3) && (2.12)\\
&= \begin{cases} (2,3), \text{ falls } Z(X>1) = (2>1) = \mathbf{false} \\ [\![\mathbf{while}\ \beta]\!]([\![Y := Y * X; X := X-1]\!](Z)), \text{ sonst} \end{cases} && (2.13)\\
&= [\![\mathbf{while}\ \beta]\!]([\![Y := Y * X; X := X-1]\!](2,3)) && (2.14)\\
&= [\![\mathbf{while}\ \beta]\!]([\![X := X-1]\!]([\![Y := Y * X]\!](2,3))) && (2.15)\\
&= [\![\mathbf{while}\ \beta]\!]([\![X := X-1]\!](2,6)) && (2.16)\\
&= [\![\mathbf{while}\ \beta]\!](1,6) && (2.17)\\
&= \begin{cases} (1,6), \text{ falls } Z(X>1) = (1>1) = \mathbf{false} \\ [\![\mathbf{while}\ \beta]\!]([\![Y := Y * X; X := X-1]\!](Z)), \text{ sonst} \end{cases} && (2.18)\\
&= (1,6) && (2.19)\\
& && (2.20)
\end{aligned}
$$

Abbildung 2.3.: Semantik imperativer Algorithmen am Beispiel

Beispiel 2.29 Fibonacci-Zahlen

```
FIB:  var  X,A,B,C:  int;
      input  x;
      A:=1,  B:=1;
      while  X>0 do
            C:=A+B;  A:=B;  B:=C;  X:=X-1  od
      output  A
```

$$[\![FIB]\!]\,(x) \;=\begin{cases} \text{die x-te Fibonacci-Zahl,} & \text{falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

□

Beispiel 2.30 Größter gemeinsamer Teiler, Version 1

```
GGT1: var  X,Y: int;
      input  X,Y;
      while  X ≠ Y do
         while  X>Y  do  X:=X-Y  od;
         while  X<Y  do  Y:=Y-X  od  od;
      output  X.
```

X	Y
19	5
14	5
9	5
4	5
4	1
3	1
2	1
$\boxed{1}$	1

□

Beispiel 2.31 Größter gemeinsamer Teiler Version 2

```
GGT2: var  X,Y,R: int;
      input  X,Y;
      R:=1;
      while  R ≠ 0  do
         R:=X  mod  Y;  X:=Y;  Y:=R  od;
      output  X.
```

Einige Berechnungen für GGT2:

X	Y	R
19	5	1
5	4	4
4	1	1
$\boxed{1}$	0	0

X	Y	R
2	100	1
100	2	1
$\boxed{2}$	0	0

ist $X > Y$, so wird erst vertauscht: $X \leftrightarrow Y$

Wie ist das Verhalten für negative X oder Y?

$$[\![GGT2]\!](x,y) = \begin{cases} ggT(x,y), & \text{falls } x,y > 0 \\ y, & \text{falls } x = y \neq 0 \text{ oder } x = 0, y \neq 0 \\ \perp & \text{falls } y = 0 \\ ggT(|x|,|y|), & \text{falls } x < 0 \text{und } y > 0 \\ -ggT(|x|,|y|), & \text{falls } y < 0 \end{cases}$$

Ist GGT2 effizienter als Version 1? □

Beispiel 2.32 Was tut der folgende Algorithmus?

```
XYZ:  var  W,X,Y,Z:  int;
      input  X;
      Z:=0;  W:=1;  Y:=1;
      while  W ≤ X  do
         Z:=Z+1;  W:=W+Y+2;  Y:=Y+2  od;
      output  Z.
```

Einige Berechnungen für XYZ:

W	X	Y	Z
1	0	1	$\boxed{0}$
1	1	1	0
4		3	$\boxed{1}$
1	2	1	0
4		3	$\boxed{1}$
1	3	1	0
4		3	$\boxed{1}$

2.8. Begriffe des Kapitels

Als Lernkontrolle sollten folgende Begriffe eindeutig charakterisiert werden können (noch unvollständig):

- Algorithmus

- Pseudo-Code-Notation, Struktogramm
- Determinismus / Nicht-Determinismus
- Terminierung
- Determiniertheit
- Sprache, Syntax, Semantik, Grammatik, regulärer Ausdruck, BNF
- Datentyp, Algebra, Signatur
- Term, Termauswertung, bedingter Term
- applikativer Algorithmus
- formale Parameter
- Rekursion
- imperativer Algorithmus
- Variable, Anweisung, Sequenz, Auswahl, Wiederholung, Iteration
- Zustand, Zustandstransformation, Semantik
- ...

2.9. Beispiele in Java

Beispiel 2.33 Fakultät applikativ

```
import algds.IOUtils;

public class Fakultaet {
  public static int fak (int x) {
    System.out.println ("\tfak ("+x+")");
    if (x <= 1)
         return 1;
     else
         return x * fak (x - 1);
   }

  public static void main (String[] args) {
    int z;

    System.out.print ("Zahl: ");
    z = IOUtils.readInt ();
```

```
    System.out.println ("fakultaet ("+z+
                            ") = "+fak(z));
   }
}
```

□

Beispiel 2.34 ggT applikativ

```
import algds.IOUtils;
public class Ggt {
  public static int ggt (int x, int y) {
    System.out.println("\tggt ("+x+","+y+")");
    if ( x <= 0 || y <= 0) {
        return 0;
    }
    else {
     if (x==y)
       return x;
     else if (x > y)
             return ggt (y,x);
      else
             return ggt (x, y - x);
    }
  }
  public static void main (String[] args) {
    int z1, z2;
    System.out.print ("Zahl #1: ");
    z1 = IOUtils.readInt ();
    System.out.print ("Zahl #2: ");
    z2 = IOUtils.readInt ();
    System.out.println
     ("ggt("+z1+","+z2+") = "+ ggt(z1,z2));
  }
}
```

□

Beispiel 2.35 Fakultät imperativ

```
import algds.IOUtils;

public class FakImp {

    public static void main (String[] args) {
```

```
        int z;
        int y = 1;

        System.out.print ("Zahl: ");
        z = IOUtils.readInt ();
        System.out.print ("Fakultaet (" + z + ") = ");
        while(z>1) { y = y * z; z--; };
        System.out.println (y);
    }
}
```

□

3. Ausgewählte Algorithmen

In diesem Abschnitt der Vorlesung werden einige ausgewählte Algorithmen präsentiert. Ziel dieser Präsentation:

- Kennenlernen von Beispielalgorithmen, auch als Grundlage für praktische Übungen
- Vorbereitung der theoretischen Betrachtungen, etwa Komplexität von Algorithmen (mehr in Kapitel 4)
- informelle Diskussion von Design-Prinzipien (mehr in Kapitel 5)

In diesem Kapitel (und den folgenden) werden wir eine Mischung aus Pseudo-Code-Notation und der Notation imperativer Algorithmen benutzen, um Algorithmen abstrakt zu beschreiben, bevor diese in Java umgesetzt werden.

3.1. Suchen in sortierten Listen

Aufgabe: Finden eines Eintrags in einer sortierten Liste (vergleichbar mit Suche in einem Telefonbuch).

3.1.1. Sequentielle Suche

Einfachste Variante: Sequentieller Durchlauf durch alle Einträge, beginnend mit dem ersten.

```
algorithm SequentialSearch:
Starte am Anfang der Liste;
while nicht gefunden
do
   Prüfe nächsten Eintrag
od;
```

Aufwand in Anzahl Vergleichen bei einer Liste der Länge n:

- im besten Fall: 1
- im schlechtesten Fall: n
- Durchschnittswert (bei erfolgreicher Suche): $n/2$
- Durchschnittswert (bei erfolgloser Suche): n

3.1.2. Binäre Suche

Raffiniertere Variante: Wähle den mittleren Eintrag, und prüfe ob gesuchter Wert in der ersten oder in der zweiten Hälfte der Liste ist. Fahre rekursiv mit der Hälfte vor, in der sich der Eintrag befindet.

Diese Variante entspricht dem üblichen Umgang mit Telefonbüchern!

Wir zeigen die *iterative* Variante ohne explizite Rekursion:

```
algorithm BinSearch:
Initialisiere zwei Variablen:
   setze UNTEN = 1,
   setze OBEN = n;
while nicht gefunden and UNTEN < OBEN
do
   setze MITTE = (UNTEN + OBEN) / 2;
   if Eintrag[MITTE] = Suchwert then Wert gefunden fi;
   if Eintrag[MITTE] > Suchwert
      then setze OBEN = MITTE
      else setze UNTEN = MITTE
   fi
od;
```

Aufwand in Anzahl Vergleichen bei einer Liste der Länge n:

- im besten Fall: 1
- im schlechtesten Fall: $\log_2 n$
- Durchschnittswert (bei erfolgreicher Suche): $\log_2 n$
- Durchschnittswert (bei erfolgloser Suche): $\log_2 n$

3.1.3. Vergleich

Bereits einfache Verbesserungen von Algorithmen können drastische Effizienzgewinne bedeuten!

3.2. Sortieren

- Grundlegendes Problem in der Informatik
- Aufgabe:
  - Ordnen von Dateien mit *Datensätzen*, die *Schlüssel* enthalten
  - Umordnen der Datensätze, so daß klar definierte Ordnung der Schlüssel (numerisch/alphabetisch) besteht
- Vereinfachung: nur Betrachtung der Schlüssel,
  z.B. Feld von `int`-Werten

3.2.1. Sortieren: Grundbegriffe

Verfahren

- intern: in Hauptspeicherstrukturen (Felder, Listen)
- extern: Datensätze auf externen Medien (Festplatte, Magnetband)

Stabilität
Ein Sortierverfahren heißt *stabil*, wenn es die relative Reihenfolge gleicher Schlüssel in der Datei beibehält.

Beispiel: alphabetisch geordnete Liste von Personen soll nach Alter sortiert werden $\rightarrow$ Personen mit gleichem Alter weiterhin alphabetisch geordnet

3.2.2. Sortieren durch Einfügen

Erste Variante: Umsetzung der typischen menschlichen Vorgehensweise, etwa beim Sortieren eines Stapels von Karten:

```
Starte mit der ersten Karte einen neuen Stapel;
Nehme jeweils nächste Karte des Originalstapels und füge
   diese an der richtigen Stelle in den neuen Stapel ein;
```

In einer detaillierteren Version, angepaßt an die Aufgabe ein Feld von Zahlen der Länge n zu sortieren:

```
algorithm InsertionSort
setze MARKER = 2;
while MARKER ≤ n
do
   merke Feld[MARKER] als TEMP (Feld[MARKER] ist nun frei);
   for alle Elemente E im Bereich MARKER-1 bis 1 abwärts
      do if E ≥ TEMP
         then verschiebe E einen nach hinten
         else brich Schleife ab;
         fi;
      od;
   setze TEMP auf die freie Stelle;
   setze MARKER = MARKER + 1;
od;
```

3.2.3. Sortieren durch Selektion

Idee: Suche jeweils größten Wert, und tausche diesen an die letzte Stelle; fahre dann mit der um 1 kleineren Liste fort.

```
algorithm SelectionSort
setze MARKER = n;
while MARKER > 0
do
   bestimme das größte Element im Bereich 1 bis MARKER;
   tausche Feld[MARKER] mit diesem Element;
   setze MARKER = MARKER - 1;
od;
```

Auch: *Sortieren durch Auswählen*, [Lek93].

3.2.4. Bubble-Sort

Idee von Bubble-Sort: Verschieden große aufsteigende Blasen ('Bubbles') in einer Flüssigkeit sortieren sich quasi von alleine, da größere Blasen die kleineren 'überholen'.

```
algorithm BubbleSort
do
   for alle Positionen I
         if Feld[I] > Feld[I+1]
         then vertausche Werte von Feld[I] und Feld[I+1]
         fi;
      do
      od;
until keine Vertauschung mehr aufgetreten;
```

Optimierungen:

- Im ersten Durchlauf 'wandert' die größte Blase an die oberste Stelle, im zweiten die zweitgrößte, etc. Die **for**-Schleifen in späteren Durchläufen können diese garantiert korrekt besetzten Stellen auslassen.
- Bei großen Feldern kann die 'Lokalität' von Feldzugriffen positiv auf die Effizienz wirken. Dann sollte der Durchlauf als 'Jojo' jeweils abwechselnd auf und nieder laufen.

3.2.5. Merge-Sort

Sortieren durch Mischen; Grundidee: Teile die zu sortierende Liste in zwei Teillisten; Sortiere diese (rekursives Verfahren!); Mische die Ergebnisse

```
algorithm MergeSort (L: Liste):  Liste
if Liste einelementig
then return L
else
```

```
      teile L in L1 und L2;
      setze L1 = MergeSort (L1);
      setze L2 = MergeSort (L2);
      return Merge(L1,L2);
   fi;

   algorithm Merge (L1,L2:  Liste):  Liste
   setze ERGEBNIS = leere Liste;
   while L1 oder L2 nicht leer
   do
      entferne das kleinere von den Anfangselementen von
      L1 und L2 aus der jeweiligen Liste und hänge es an
      ERGEBNIS an;
   od;
   return ERGEBNIS;
```

Der Vorgang des Mischens erfordert in der Regel doppelten Speicherplatz (oder aufwendiges Verschieben)!

Auch: *Sortieren durch Verschmelzen*, [Lek93].

3.2.6. Quick-Sort

Quick-Sort arbeitet ähnlich wie Merge-Sort durch rekursive Aufteilung. Der Mischvorgang wird dadurch vermieden, daß die Teillisten bezüglich eines Referenzelementes in zwei Hälften aufgeteilt werden, von denen die eine alle Elemente größer als das Referenzelement enthält, die andere die kleineren.

```
algorithm QuickSort1 (L: Liste):  Liste
bestimme Teilungselement PIVOT;
teile L in L1 und L2 so daß gilt:
   alle Elemente in L1 sind kleiner als PIVOT, und
   alle Elemente in L2 sind größer als PIVOT;
if L1 oder L2 hat mehr als ein Element
then
   setze L1 = QuickSort (L1);
   setze L2 = QuickSort (L2);
fi;
return L1 + [ PIVOT ] + L2;
```

`PIVOT` kann etwa als erstes Element des Feldes gewählt werden, oder auch als mittleres Element (besonders günstig, falls die Eingabe auch vorsortiert sein könnte!), oder als Mittelwert des ersten, des letzten und des mittleren Elements.

Die Realisierung mit einem Feld (array of int) kann die Aufteilung in L1 und L2 ohne weiteren Speicherplatzbedarf innerhalb des Feldes erfolgen:

```
sei n die Anzahl der Elemente des Feldes
   und 1 der untere Index;
setze PIVOT = Feld[ z.B. beliebig aus 1 bis n ];
setze UNTEN = 1;
setze OBEN = n;
repeat
   while Feld[UNTEN] < PIVOT do UNTEN = UNTEN + 1;
   while Feld[OBEN] > PIVOT do OBEN = OBEN - 1;
   if UNTEN < OBEN
   then vertausche Inhalte von Feld[UNTEN] und Feld[OBEN];
        UNTEN = UNTEN + 1; OBEN = OBEN - 1;
   fi
until UNTEN > OBEN;
```

Danach kann wie folgt aufgeteilt werden:

```
L1 = Feld[1..OBEN]
L2 = Feld[UNTEN..n]
```

Da die Sortierung *innerhalb* des Feldes erfolgt, benötigt man am Ende keine Konkatenation der Ergebnislisten wie in der ersten Version `QuickSort1` des Algorithmus.

3.2.7. Sortier-Verfahren im Vergleich

	Insertion	**Selection**	**Bubble**	**Merge**	**Quick**
Stabilität	stabil	instabil	stabil	stabil	instabil
Vergleiche					
minimal	$n-1$	$\frac{n^2-n}{2}$	$n-1$	$O(n \lg n)$	$O(n \lg n)$
maximal	$\frac{n^2-n}{2}+n-1$	$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$	$O(n \lg n)$	$O(n^2)$
mittel	$\frac{n^2+n-2}{2}$	$\frac{n^2-n}{2}$	$O(n^2)$	$O(n \lg n)$	$O(n \lg n)$

3.3. Java-Realisierungen der Beispiele

3.3.1. Such-Algorithmen

```
import algds.IOUtils;

public class Suche {
  // Ausgabe der Vergleiche bei verbose == true
  static boolean verbose = true;
```

```
// Anlegen und initialisieren eines Feldes.
public static int[] init (int anzahl) {
  int[] feld = new int[anzahl];
  int index;
  for (index = 0; index < anzahl; index = index + 1)
    feld[index] = index + 1;
  return feld;
}

// Sequentielles Durchsuchen des Feldes feld
// nach den Wert wert.
public static boolean sequentiell (int wert, int[] feld) {
  int index;
  boolean gefunden = false;
  for (index = 0; index < feld.length && ! gefunden;
      index = index + 1) {
    if (verbose)
      System.out.println ("  Vergleich mit "
                         + feld[index]);
    if (feld[index] == wert)
      gefunden = true;
  }
  return gefunden;
}

// Binaeres Durchsuchen des Feldes feld
// nach den Wert wert.
public static boolean binaer (int wert, int[] feld) {
  int unten = 0;
  int oben = feld.length - 1;
  int mitte;
  boolean gefunden = false;
  while (oben >= unten && ! gefunden) {
    mitte = (oben + unten) / 2;

    if (verbose)
      System.out.println ("  Vergleich mit "
                          + feld[mitte]);
    if (feld[mitte] == wert)
      gefunden = true;
    else {
      if (feld[mitte ] < wert)
        unten = mitte + 1;
```

```
        else
          oben = mitte - 1;
      }
    }
    return gefunden;
  }

  public static void main (String[] args) {
    int wert, anzahl;
    boolean gefunden = false;
    int[] feld = null;

    System.out.print ("Feldgroesse: ");
    anzahl = IOUtils.readInt ();
    feld = init (anzahl);

    do {
      System.out.print ("gesuchter Wert: ");
      wert = IOUtils.readInt ();
      if (wert == 0)
        break;
      System.out.println ("Sequentielle Suche !");
      gefunden = sequentiell (wert, feld);
      if (gefunden)
        System.out.println ("Wert gefunden!");
      else
        System.out.println ("Wert nicht gefunden!");

      System.out.println ("Binaere Suche !");
      gefunden = binaer (wert, feld);
      if (gefunden)
        System.out.println ("Wert gefunden!");
      else
        System.out.println ("Wert nicht gefunden!");
    } while (wert > 0);
  }
}
```

3.3.2. Sortier-Algorithmen

Sortieren in Java

```
public class Sort {
```

```
/*
 * Hilfsmethode: Initialisierung eines Feldes mit Zufallszahlen
 */
static int[] initArray (int num) {
  int[] result = new int[num];
  for (int i = 0; i < num; i++)
    result[i] = (int) (Math.random () * 100.0);
  return result;
}

/*
 * Hilfsmethode: Ausgabe der Elemente eines Feldes
 */
static void printArray (int[] array) {
  for (int i = 0; i < array.length; i++)
    System.out.print (array[i] + " ");
  System.out.println ();
}

/*
 * Hilfsmethode: Austauch zweier Feldelemente
 */
static void swap (int[] array, int idx1, int idx2) {
  int tmp = array[idx1];
  array[idx1] = array[idx2];
  array[idx2] = tmp;
}

static void insertionSort (int[] array) {
  for (int i = 1; i < array.length; i++) {
    int marker = i;
    int temp = array[i];
    // fr alle Elemente links vom Marker-Feld
    while (marker > 0 && array[marker - 1] > temp) {
      // verschiebe alle greren Element nach hinten
      array[marker] = array[marker - 1];
      marker--;
    }
    // setze temp auf das freie Feld
    array[marker] = temp;
  }
}
```

```
/*
 * Implementierung des SelectionSort
 */
static void selectionSort (int[] array) {
  int marker = array.length - 1;
  while (marker >= 0) {
    // bestimme grtes Element
    int max = 0;
    for (int i = 1; i <= marker; i++)
      if (array[i] > array[max])
        max = i;

    // tausche array[marker] mit diesem Element
    swap (array, marker, max);
    marker--;
  }
}

/*
 * Implementierung des Bubble-Sort
 */
static void bubbleSort1 (int[] array) {
  boolean swapped;

  do {
    swapped = false;

    for (int i = 0; i < array.length - 1; i++) {
      if (array[i] > array[i + 1]) {
        // Elemente vertauschen
        swap (array, i, i + 1);
        swapped = true;
      }
    }
    // solange Vertauschung auftritt
  } while (swapped);
}

/*
 * Verbesserte Implementierung des Bubble-Sort
 */
static void bubbleSort2 (int[] array) {
  boolean swapped; // Vertauschung ?
  int max = array.length - 1; // obere Feldgrenze
```

```
  do {
    swapped = false;

    for (int i = 0; i < max; i++) {
      if (array[i] > array[i + 1]) {
        // Elemente vertauschen
        swap (array, i, i + 1);
        swapped = true;
      }
    }
    max--;
    // solange Vertauschung auftritt
  } while (swapped);
}

/*
 * Implementierung des MergeSort
 */

// Hilfsmethode fr rekursives Sortieren durch Mischen
static void msort (int[] array, int l, int r) {
  int i, j, k;
  int[] b = new int[array.length];

  if (r > l) {
    // zu sortierendes Feld teilen
    int mid = (r + l) / 2;
    // Teilfelder sortieren
    msort (array, l, mid);
    msort (array, mid + 1, r);

    // Ergebnisse mischen ber Hilfsfeld b
    for (i = mid + 1; i > l; i--)
      b[i - 1] = array[i - 1];
    for (j = mid; j < r; j++)
      b[r + mid - j] = array[j + 1];
    for (k = l; k <= r; k++)
      if (b[i] < b[j])
        array[k] = b[i++];
      else
        array[k] = b[j--];

  }
}
```

```
static void mergeSort (int[] array) {
  msort (array, 0, array.length - 1);
}

/*
 * Implementierung des QuickSort
 */

// Hilfsmethode fr rekursives Sortieren
static void qsort (int[] array, int l, int r) {
  int lo = l, hi = r;

  if (hi > lo) {
    // Pivotelement bestimmen
    int mid = array[(lo + hi) / 2];

    while (lo <= hi) {
      // Erstes Element suchen, das grer oder gleich dem
      // Pivotelement ist, beginnend vom linken Index
      while ((lo < r) && (array[lo] < mid))
        ++lo;

      // Element suchen, das kleiner oder gleich dem
      // Pivotelement ist, beginnend vom rechten Index
      while (( hi > l ) && ( array[hi] > mid))
        --hi;

      // Wenn Indexe nicht gekreuzt --> Inhalte vertauschen
      if (lo <= hi) {
        swap(array, lo, hi);
        ++lo;
        --hi;
      }
    }

    // Linke Partition sortieren
    if (l < hi)
      qsort (array, l, hi);

    // Rechte Partition sortieren
    if (lo < r)
      qsort( array, lo, r);
  }
```

```
  }

  static void quickSort (int[] array) {
    qsort (array, 0, array.length - 1);
  }

  public static void main(String[] args) {
    int[] array = null;

    array = initArray (20);
    mergeSort (array);
    printArray (array);

    array = initArray (20);
    quickSort (array);
    printArray (array);

    array = initArray (20);
    selectionSort (array);
    printArray (array);

    array = initArray (20);
    insertionSort (array);
    printArray (array);

    array = initArray (20);
    bubbleSort2 (array);
    printArray (array);
  }
}
```

4. Eigenschaften von Algorithmen

4.1. Formale Algorithmenmodelle

Ziel: Modelle für Maschinen, die Algorithmen ausführen

- näher an tatsächlichen Computern, oder
- einfacher mathematisch nutzbar für Beweisführungen

4.1.1. Registermaschinen

- Registermaschinen: nahe an tatsächlichen Computern ("idealisierte" Version eines Prozessors mit Maschine-Code)

Eine (maschinennahe) Präzisierung des Algorithmenbegriffs kann mittels *Registermaschinen* vornehmen. Diese stellen eine relativ simple Abstraktion der programmierbaren Rechner dar.

Definition 4.1 Definition Registermaschine:

i) Eine Registermaschine besteht aus den Registern

$$B,\ C_0,\ ,C_1, C_2, \ldots, C_n, \ldots$$

und einem Programm.

ii) B heißt Befehlszähler, C_0 heißt Arbeitsregister oder Akkumulator, und jedes der Register C_n, $n \geq 1$ heißt Speicherregister.

Jedes der Register enthält als Wert eine natürliche Zahl. Enthält das Register B die Zahl b und für $n \geq 0$ das Register C_n die Zahl c_n, so heißt das unendliche Tupel

$$(b, c_0, c_1, \ldots, c_n, \ldots)$$

Konfiguration der Registermaschine.

iii) Das Programm ist eine endliche Folge von Befehlen. Durch die Anwendung eines Befehls wird die Konfiguration der Registermaschine geändert.

Die folgende Liste gibt die zugelassenen Befehle und die von ihnen bewirkte Änderung der Konfiguration

$$(b, c_o, c_1, \ldots c_n, \ldots)$$

in die Konfiguration

$$(b', c'_0, c'_1, \ldots, c'_n, \ldots),$$

geschrieben als

$$(b, c_o, c_1, \ldots c_n, \ldots) \vdash (b', c'_0, c'_1, \ldots, c'_n, \ldots),$$

an:

Ein- und Ausgabebefehle:

`LOAD` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = c_i$	$c'_j = c_j$ für $j \neq 0$,
`CLOAD` i ,	$i \in \mathbb{N}$	$b' = b + 1$	$c'_0 = i$	$c'_j = c_j$ für $j \neq 0$,
`STORE` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_i = c_0$	$c'_j = c_j$ für $j \neq i$,

Arithmetische Befehle:

`ADD` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = c_0 + c_i$	$c'_j = c_j$ für $j \neq 0$,
`CADD` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = c_0 + i$	$c'_j = c_j$ für $j \neq 0$,
`SUB` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = \begin{cases} c_0 - c_i & \text{für } c_0 \geq c_i \\ 0 & \text{sonst} \end{cases}$	$c'_j = c_j$ für $j \neq 0$,
`CSUB` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = \begin{cases} c_0 - i & \text{für } c_0 \geq i \\ 0 & \text{sonst} \end{cases}$	$c'_j = c_j$ für $j \neq 0$,
`MULT` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = c_0 * c_i$	$c'_j = c_j$ für $j \neq 0$,
`CMULT` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = c_0 * i$	$c'_j = c_j$ für $j \neq 0$,
`DIV` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = \lfloor c_0 / c_i \rfloor$	$c'_j = c_j$ für $j \neq 0$,
`CDIV` i ,	$i \in \mathbb{N}_+$	$b' = b + 1$	$c'_0 = \lfloor c_0 / i \rfloor$	$c'_j = c_j$ für $j \neq 0$,

Sprungbefehle:

`GOTO` i ,	$i \in \mathbb{N}_+$	$b' = i$	$c'_j = c_j$ für $j \geq 0$,
`IF` $c_0 = 0$ `GOTO` i ,	$i \in \mathbb{N}_+$	$b' = \begin{cases} i & \text{falls } c_0 = 0 \\ b + 1 & \text{sonst} \end{cases}$	$c'_j = c_j$ für $j \geq 0$,

Stopbefehl:

`END` $\quad b' = b \quad c'_j = c_j$ für $j \geq 0$

□

Bei den Eingabebefehlen `LOAD` i und `CLOAD` i wird der Wert des i-ten Registers bzw. die Zahl i in den Akkumulator geladen; bei `STORE` i wird der Wert des Akkumulators in das i-te Speicherregister eingetragen.

Bei den Befehlen `ADD` i, `SUB` i, `MULT` i und `DIV` i erfolgt eine Addition, Subtraktion, Multiplikation und Division des Wertes des Akkumulators mit dem Wert des i-ten Speicherregisters. Da die Operationen nicht aus dem Bereich der natürlichen Zahlen herausführen sollen, wird die Subtraktion nur dann wirklich ausgeführt,

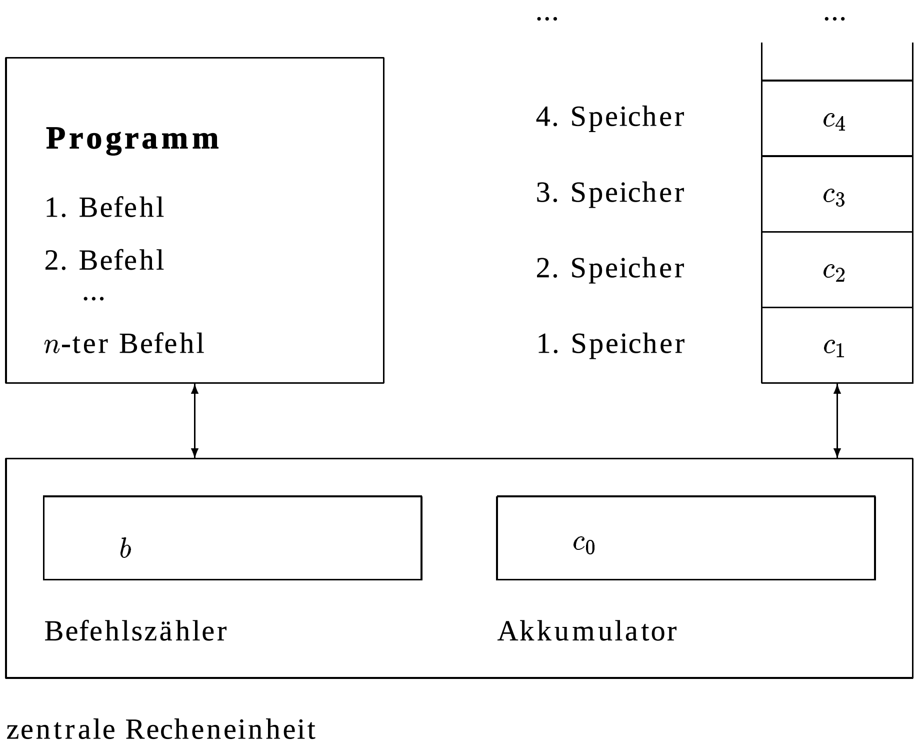


Abbildung 4.1.: Aufbau einer Registermaschine

wenn der Subtrahend nicht kleiner als der Minuend ist und sonst 0 ausgegeben; analog erfolgt die Division nur ganzzahlig.

Die Befehle `CADD` i, `CSUB` i, `CMULT` i und `CDIV` i arbeiten analog, nur daß anstelle des Wertes des i-ten Registers die natürliche Zahl i benutzt wird. Dadurch werden auch arithmetische Operationen mit Konstanten möglich.

In all diesen Fällen wird der Wert des Befehlsregisters um 1 erhöht, d.h. der nächste Befehl des Programms wird abgearbeitet. Dies ist bei den Sprungbefehlen grundsätzlich anders. Bei `GOTO` i wird als nächster Befehl der i-te Befehl des Programms festgelegt, während bei der `IF`-Anweisung in Abhängigkeit von dem Erfülltsein der Bedingung $c_0 = 0$ der nächste Befehl der i-te bzw. der $(b + 1)$-te Befehl des Programms ist.

Der Befehl `END` ist kein eigentlicher Stopbefehl. Er läßt die Konfiguration unverändert, so daß diese nun ad infinitum bestehen bleibt (und daher als das Ende der Berechnung angesehen werden kann).

Eine Registermaschine läßt sich entsprechend Abb. 4.1 veranschaulichen.

Beispiel 4.1 Wir betrachten die Registermaschine M_1 mit dem Programm

```
1  LOAD 1
2  DIV 2
3  MULT 2
4  STORE 3
5  LOAD 1
6  SUB 3
7  STORE 3
8  END
```

und untersuchen die Veränderung der Konfiguration, wobei wir uns auf das Befehlsregister, den Akkumulator und die ersten drei Speicherregister beschränken, da nur diese während der Berechnung verändert werden. Stehen in den Registern zuerst die Zahlen

$$b = 1,\ c_0 = 0,\ c_1 = 32,\ c_2 = 5,\ c_3 = 0,$$

so ergibt sich folgende Folge von Konfigurationen

$$\begin{aligned}(1, 0, 32, 5, 0, \dots) &\vdash (2, 32, 32, 5, 0, \dots) \vdash (3, 6, 32, 5, 0, \dots) \vdash (4, 30, 32, 5, 0, \dots)\\ &\vdash (5, 30, 32, 5, 30, \dots) \vdash (6, 32, 32, 5, 30, \dots) \vdash (7, 2, 32, 5, 30, \dots)\\ &\vdash (8, 2, 32, 5, 2, \dots),\end{aligned}$$

womit der „stoppende“ Befehl erreicht wird. Sind die Inhalte der Register dagegen

$$b = 1,\ c_0 = 0,\ c_1 = 100,\ c_2 = 20,\ c_3 = 0,$$

so ergibt sich folgende Folge von Konfigurationen

$$\begin{aligned}(1, 0, 100, 20, 0, \dots) &\vdash (2, 100, 100, 20, 0, \dots) \vdash (3, 5, 100, 20, 0, \dots)\\ &\vdash (4, 100, 100, 20, 0, \dots) \vdash (5, 100, 100, 20, 100, \dots)\\ &\vdash (6, 100, 100, 20, 100, \dots) \vdash (7, 0, 100, 20, 100, \dots)\\ &\vdash (8, 0, 100, 20, 0, \dots).\end{aligned}$$

Allgemeiner läßt sich folgendes feststellen. Wir betrachten eine Konfiguration, die durch

$$b = 1,\ c_0 = 0,\ c_1 = n,\ c_2 = m,\ c_3 = 0$$

gegeben ist, und nehmen an, daß $n = q \cdot m + r$ mit $0 \leq r < m$ gegeben ist, d.h. $q = \lfloor n/m \rfloor$ ist das ganzzahlige Ergebnis der Division von n durch m und r ist der verbleibende Rest bei dieser Division. Dann ergibt sich die Folge

$$\begin{aligned}(1, 0, n, m, 0, \dots) &\vdash (2, n, n, m, 0, \dots) \vdash (3, q, n, m, 0, \dots)\\ &\vdash (4, q \cdot m, n, m, 0, \dots) \vdash (5, q \cdot m, n, m, q \cdot m, \dots)\\ &\vdash (6, n, n, m, q \cdot m, \dots) \vdash (7, r, n, m, q \cdot m, \dots)\\ &\vdash (8, r, n, m, r, \dots).\end{aligned}$$

□

Diese Betrachtung legt folgendes nahe. Gehen wir von einer Konfiguration aus, die im Befehlsregister eine 1 enthält, d.h. es wird mit der Abarbeitung des Programms beim ersten Befehl begonnen, und in den beiden ersten Speicherregistern zwei natürliche Zahlen n und m enthält, so steht nach Abarbeitung des Programms (genauer: in der sich nicht mehr verändernden Konfiguration, die bei Erreichen der END-Anweisung erreicht wird) im dritten Speicherregister der Rest bei der ganzzahligen Division von n durch m. Daher ist es naheliegend zu sagen, daß die gegebene Registermaschine - genauer die durch das Programm gegebene Registermaschine - die Funktion $r = rest(n,m)$ berechnet, die durch

$$r = rest(n,m) \quad \text{genau dann, wenn} \quad 0 \leq r < n \text{ und } n = qm + r \text{ für ein } q \in \mathbb{N}$$

gegeben ist.

Wir verallgemeinern diese Idee in der folgenden Definition.

Definition 4.2 Eine Registermaschine M berechnet die Funktion $f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$ mit $f(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m)$, wenn es Zahlen $i_1, i_2, \ldots, i_m$ so gibt, daß M jede Konfiguration $(1, 0, x_1, x_2, \ldots, x_n, 0, 0, \ldots)$ in eine Konfiguration $(b, c_0, c_1, \ldots)$ überführt, für die b die Nummer einer END-Anweisung ist und $c_{i_j} = y_j$ für $1 \leq j \leq m$ gilt. □

Intuitiv bedeutet dies folgendes: Die Registermaschine beginnt die Abarbeitung ihres Programms mit dem ersten Befehl, wobei die Argumente bzw. Eingaben der Funktion in den ersten n Speicherregistern $C_1, C_2, \ldots, C_n$ stehen. Sie beendet ihre Arbeit bei Erreichen eines END-Befehls und die Resultate bzw. Ausgaben stehen in den vorab festgelegten Speicherregistern $C_{i_1}, C_{i_2}, \ldots, C_{i_m}$.

Die arithmetischen Grundfunktionen lassen sich in diesem Sinn einfach berechnen. Z.B. berechnet die Registermaschine mit dem Program

```
1  LOAD 1
2  ADD 2
3  STORE 3
4  END
```

aus den Werten x und y in den Registern C_1 und C_2 die Summe $x + y$ und legt diese im Register C_3 ab, realisiert also mit $i_1 = 3$ die Addition. Analog verfährt man bei den anderen Operationen.

Beispiel 4.2 Wir betrachten die Registermaschine M_2 mit dem Programm

```
1   CLOAD 1
2   STORE 3
3   LOAD 2
4   IF c0 = 0 GOTO 12
5   LOAD 3
6   MULT 1
7   STORE 3
8   LOAD 2
9   CSUB 1
10  STORE 2
11  GOTO 4
12  END
```

und bestimmen die Funktion $f_2 : \mathbb{N}^2 \longrightarrow \mathbb{N}$, die das Ergebnis im dritten Speicherregister enthält.

Wir betrachten zuerst die Entwicklung einer konkreten Konfiguration $(1, 0, 5, 3, 0, \dots)$, wobei wir uns erneut auf das Befehlsregister, den Akkumulator und die ersten drei Speicherregister beschränken, da nur diese vom Programm betroffen sind. Es ergibt sich folgende Folge:

$$
\begin{aligned}
(1,0,5,3,0,\dots) &\vdash (2,1,5,3,0,\dots) \vdash (3,1,5,3,1,\dots) \vdash (4,3,5,3,1,\dots)\\
&\vdash (5,3,5,3,1,\dots) \vdash (6,1,5,3,1,\dots) \vdash (7,5,5,3,1,\dots)\\
&\vdash (8,5,5,3,5,\dots) \vdash (9,3,5,3,5,\dots) \vdash (10,2,5,3,5,\dots)\\
&\vdash (11,2,5,2,5,\dots) \vdash (4,2,5,2,5,\dots) \vdash (5,2,5,2,5,\dots)\\
&\vdash (6,5,5,2,5,\dots) \vdash (7,25,5,2,5,\dots) \vdash (8,25,5,2,25,\dots)\\
&\vdash (9,2,5,2,25,\dots) \vdash (10,1,5,2,25,\dots) \vdash (11,1,5,1,25,\dots)\\
&\vdash (4,1,5,1,25,\dots) \vdash (5,1,5,1,25,\dots) \vdash (6,25,5,1,25,\dots)\\
&\vdash (7,125,5,1,25,\dots) \vdash (8,125,5,1,125,\dots) \vdash (9,1,5,1,125,\dots)\\
&\vdash (10,0,5,1,125,\dots) \vdash (11,0,5,0,125,\dots) \vdash (4,0,5,0,125,\dots)\\
&\vdash (12,0,5,0,125,\dots).
\end{aligned}
$$

Dieses konkrete Beispiel ergibt $f_1(5,3) = 125$.

Für den allgemeinen Fall, d.h. die Konfiguration $(1, 0, x, y, 0, \dots)$, gelten folgende Bemerkungen. Nach der Abarbeitung der Befehle 1 – 3 ergibt sich die Konfiguration $(4, y, x, y, 1, \dots)$.

Falls $y = 0$ ist, so wird zur END-Anweisung gegangen, und es ergibt sich aus der Konfiguration $(12, y, x, y, 1) = (12, 0, x, 0, 1)$ das Ergebnis 1 aus dem dritten Speicherregister. Falls $y \neq 0$ ist, so werden die Befehle 5 – 11 durchlaufen, wodurch ins dritte Speicherregister der Wert $1 \cdot x$ und ins zweite Speicherregister der Wert $y - 1$ geschrieben wird, d.h. die erreichte Konfiguration ist $(4, y - 1, x, y, 1 \cdot x, \dots)$.

Ist $y - 1 = 0$, so wird zur Konfiguration $(12, y-1, x, y-1, x, \dots) = (12, 0, x, 0, x, \dots)$ übergegangen und x als Ergebnis ausgegeben. Ist $y - 1 \neq 0$, so werden erneut die Befehle 5 – 11 abgearbeitet und $(4, y - 2, x, y - 2, x^2, \dots)$ erhalten.

Ist $y - 2 = 0$, so wird $(12, y-2, x, y-2, x^2, \dots) = (12, 0, x, 0, x^2, \dots)$ erreicht und das Ergebnis ist x^2; bei $y - 2 \neq 0$ werden erneut die Befehle 5 – 11 abgearbeitet.

Folglich bricht der Prozeß mit $(12, y-k, x, y-k, x^k, \dots) = (12, 0, x, 0, x^y, \dots)$ (wegen $y = k$) ab.

Somit ist die berechnete Funktion

$$f_2(x, y) = x^y.$$

Dies Beispiel zeigt daher, daß neben den arithmetischen Grundoperationen auch das Potenzieren eine durch Registermaschinen berechenbare Funktion ist. □

Beispiel 4.3 Offensichtlich berechnet die Registermaschine M_3 mit dem Programm

1 `LOAD` 1
2 `IF` $c_0 = O$ `GOTO` 4
3 `GOTO` 3
4 `END`

die Funktion

$$f_3(x) = \begin{cases} 0 & \text{falls } x = 0 \\ \text{undefiniert} & \text{sonst} \end{cases},$$

denn nur in dem Fall, daß die Eingabe 0 ist, wird zur `END`-Anweisung gesprungen, ansonsten werden die Befehle 2 und 3 abwechselnd unendlich lange hintereinander ausgeführt und damit kein Ergebnis produziert. □

Dieses Beispiel zeigt, daß die von Registermaschinen berechneten Funktionen partiell sein können, d.h. ihr Definitionsbereich ist eine echte Teilmenge von $\mathbb{N}^n$.

Beispiel 4.4 Wir wollen nun eine Registermaschine M_4 konstruieren, die die Funktion

$$f_4(x) = \begin{cases} 0 & \text{falls } x \text{ keine Primzahl ist} \\ 1 & \text{falls } x \text{ eine Primzahl ist} \end{cases}$$

berechnet.

Eine natürliche Zahl x ist bekanntlich genau dann eine Primzahl, wenn $x \geq 2$ ist und nur die Teiler 1 und x besitzt.

Hieraus resultiert folgendes Vorgehen: Wir überprüfen zuerst, ob die Eingabe $x \leq 1$ ist. Sollte dies der Fall sein, so ist x keine Primzahl und wir schreiben in das zweite Speicherregister eine 0 und beenden die Arbeit. Ist dagegen $x \geq 2$, so testen wir der Reihe nach, ob x durch $2, 3, 4, \dots x - 1$ teilbar ist. Gibt einer dieser Tests ein positives Resultat, so ist x keine Primzahl; fallen dagegen alle diese Tests negativ aus, so ist x prim.

Nachfolgendes Programm realisiert diese Idee (zur Illustration geben wir in der rechten Spalte kurze Erläuterungen zu den Befehlen):

1	`LOAD 1`	Laden von x
2	`CSUB 1`	Berechnen von $x - 1$
3	`IF` $c_0 = 0$ `GOTO 19`	Test, ob $x \leq 1$
4	`CLOAD 2`	Laden des ersten Testteilers t=2
5	`STORE 2`	Speichern des Testteilers t
6	`LOAD 1`	
7	`SUB 2`	Berechnen von $x - t$
8	`IF` $c_0 = 0$ `GOTO 21`	Test, ob $t < x$
9	`LOAD 1`	
10	`DIV 2`	die Befehle 9 – 14 berechnen den Rest
11	`MULT 2`	bei ganzzahliger Teilung von x durch t
12	`STORE 3`	(siehe Beispiel 4.1)
13	`LOAD 1`	
14	`SUB 3`	
15	`IF` $c_0 = 0$ `GOTO 19`	Test, ob t Teiler
16	`LOAD 2`	
17	`CADD 1`	Erhöhung des Testteilers um 1
18	`GOTO 5`	Start des nächsten Teilbarkeitstests
19	`STORE 2`	Speichern des Ergebnisses 0
20	`GOTO 23`	
21	`CLOAD 1`	
22	`STORE 2`	Speichern des Ergebnisses 1
23	`END`	

□

Bemerkung: Notationen etc an Skript von Dassow [**?**] angelehnt.

4.1.2. Abstrakte Maschinen

- Abstrakte Maschinen: universelles Rahmen-Modell für verschiedene Algorithmen-Notationen

Wir definieren ein recht allgemeines mathematisches Modell für deterministische Algorithmen, "abstrakte Maschine" genannt.

Definition 4.3 Eine *abstrakte Maschine* M ist ein 7-Tupel

$$M \;=\; (X, Y, K, \alpha, \omega, \tau, \sigma),$$

wobei gilt:

X	ist eine Menge von *Eingabewerten*
Y	ist eine Menge von *Ausgabewerten*
K	ist eine Menge von *Konfigurationen*
α:	$X \to K$ ist die *Eingabefunktion*
ω:	$K \to Y$ ist die *Ausgabefunktion*
τ:	$K \to K$ ist die *Transitionsfunktion*
σ:	$K \to$ **`bool`** ist die *Stopfunktion*

□

Definition 4.4 Die Menge der *Endkonfigurationen* zu M ist

$$E = \{k \in K \mid \sigma(k) = \texttt{true}\}$$

□

Eine abstrakte Maschine M läßt sich auch durch ein Diagramm darstellen:

$$\begin{tikzcd} X \arrow[r, "\alpha"] & K \arrow[loop above, "\tau"] \arrow[r, "\omega"] \arrow[d, "\sigma"] & Y \\ & \{\texttt{true}, \texttt{false}\} & \end{tikzcd}$$

Eine abstrakte Maschine *arbeitet* folgendermaßen:

1. Ein Eingabewert $x \in X$ bestimmt die Anfangskonfiguration $k_0 = \alpha(x) \in K$.
2. Wir überführen mittels τ Konfigurationen in Folgekonfigurationen , also

$$k_1 = \tau(k_0), k_2 = \tau(k_1), \ldots$$

   bis zum erstenmal eine Endkonfiguration $k_i \in E$ erreicht wird. Dies braucht natürlich niemals einzutreten.
3. Wird eine Endkonfiguration $k_1 \in E$ erreicht, so wird der Ausgabewert $\omega(k_i) \in Y$ ausgegeben.

Bei Eingabe von $x \in X$ gibt es also zwei Möglichkeiten:

1. Die Maschine hält nie an.
2. Die Maschine hält und gibt einen eindeutig bestimmten Ausgabewert $y \in Y$ aus.

Auf diese Weise berechnet die Maschine M eine partielle Funktion

$$f_M : X \to Y$$

Wir präzisieren diese Verhaltensweise mathematisch.

Definition 4.5 Die *Laufzeit* einer abstrakten Maschine M für die Eingabe $x \in X$ ist

$$t_M(x) \;=\; (\mu n)[\sigma(\tau^n(\alpha(x)))]$$

□

Hierbei ist $(\mu n)[B]$ die kleinste natürliche Zahl $n \in \mathbb{N} \;=\; \{0, 1, 2, \dots\}$, so daß die Bedingung $B = \texttt{true}$ wird und B für alle $m \leq n$ definiert ist. Gibt es keine solche Zahl $n \in \mathbb{N}$, so ist $t_M(x) = \bot$ (undefiniert).

Definition 4.6 Die von einer abstrakten Maschine M *berechnete Funktion*

$$f_M : X \rightarrow Y$$

ist gegeben durch

$$f_M(x) \;=\; \omega(\tau^{t_M(x)}(\alpha(x))) \;\;;\quad \text{ist} t_M(x) \;=\; \bot, \text{ so ist } f_M(x) \;=\; \bot.$$

□

Applikative und imperative Algorithmen sind Beispiele dieses allgemeinen Algorithmenmodells, d.h. sie lassen sich als abstrakte Maschinen auffassen. Wir skizzieren den Zusammenhang grob ohne ins Detail zu gehen.

Beispiel 4.5 Applikative Algorithmen (nur ganzzahlige E/A)

$$f_i(u_{i,1}, \dots, u_{i,n_i}) \;=\; t_i(u_{i,1}, \dots, u_{i,n_i}), \quad i = 1, \dots, m.$$

- $X \;=\; \mathbb{Z}^{n_1}$
- $Y \;=\; \mathbb{Z}$
- $K \;=\;$ Terme ohne Unbekannte
- $\alpha(a_1, \dots, a_{n_1}) \;=\;$ der Term "$f_1(a_1, \dots a_{n_1})$"
- $\omega(t) \;=\;$ der Wert von t
- τ: Termauswertung aufgrund der Funktionsdefinitionen.

  Da diese nichtdeterministisch ist, müssen wir sie durch eine zusätzliche "Berechnungsvorschrift" deterministisch machen, z.B. durch die Forderung, das erste Auftreten von links eines Funktionsaufrufes mit Konstanten. $f_i(b_1, \dots, b_{n_i})$ mit $b_j \in \mathbb{Z}$, $j = 1, \dots, n_i$, durch die rechte Seite $t_i(b_1, \dots, b_{n_i})$ zu ersetzen.
- $\sigma(t) \;=\; \begin{cases} \texttt{true}, \text{ falls } t = b \in \mathbb{Z} \text{ (Konstante) ist} \\ \texttt{false} \text{ sonst} \end{cases}$ □

Beispiel 4.6 Imperative Algorithmen (nur ganzzahlige E/A)

```
PROG: var    V, W, ... :  int;
             P,Q, ... :  bool;
      input  X_1, ..., X_n;
             β̄;
      output Y_1, ..., Y_m.
```

- $X = \mathbb{Z}^n$
- $Y = \mathbb{Z}^m$
- $K = \{(Z, \beta) \mid Z$ Zustand, β Anweisung $\}$

  Z bezeichnet den aktuellen Zustand,
  β die noch auszuführende Anweisung

- $\alpha(a_1, \ldots, a_n) = (Z_0, \overline{\beta})$, wobei

$$\begin{aligned} Z_0(X_i) &= a_i, i = 1, \ldots, n, \text{ und} \\ Z_0(Y) &= \perp \text{ für } Y \neq X_i, i = 1, \ldots, n. \end{aligned}$$

- $\omega(Z, \beta) = (Z(Y_1), \ldots, Z(Y_m))$
- $\tau(Z, \beta) = (Z', \beta')$, wobei

$$\begin{aligned} Z' &= \text{Zustand nach Ausführung der nächsten Anweisung} \\ \beta' &= \text{Rest der noch auszuführenden Anweisung} \end{aligned}$$

- $\sigma(t) = \begin{cases} \textbf{true}, \text{ falls } \beta \text{ leer ist (keine Anweisungen mehr)} \\ \textbf{false} \text{ sonst} \end{cases}$ □

Auch Register-Maschinen lassen sich direkt als Spezialfall einer abstrakten Maschine beschreiben.

4.1.3. Markov-Algorithmen

- einfaches mathematisch orientiertes Modell als Spezialisierung abstrakter Maschinen
- programmtechnisch einfach in einen Interpretierer für Markov-Tafeln umzusetzen

Sei $A = (a_1, \ldots, a_n)$ ein Alphabet, und sei A^* die Menge der Worte (Texte) über A. Wir definieren die Klasse der *Markov-Algorithmen* (Markov 1954). Die elementaren Einzelschritte dieser Algorithmen beruhen auf *Regeln* der Form $\varphi \to \psi$, wobei $\varphi, \psi \in A^*$ sind. Diese Angaben bedeuten, daß das Wort φ durch das Wort ψ ersetzt werden soll. Angewendet auf ein Wort $\xi \in A^*$ entsteht somit auf eindeutige Weise ein neues Wort $g[\varphi \to \psi](\xi)$, das wie folgt definiert ist:

1. Ist φ ein Teilwort von ξ, also $\xi = \mu\varphi\nu$ für $\mu, \nu \in A^*$, und ist φ an dieser Stelle nach μ das erste Auftreten (von links) von φ in ξ, so ist $g[\varphi \to \psi](\xi) \;=\; \mu\psi\nu$, d.h. φ wird (nur!) an dieser Stelle durch ψ ersetzt.

2. Ist φ kein Teilwort von ξ, so ist $g[\varphi \to \psi](\xi) \;=\; \xi$, d.h., es passiert nichts.

Beispiel 4.7 Sei $A = (0, 1)$. Wir wenden die Regel $01 \to 10$ sukzessive an.

$$
\begin{array}{lcl}
1\,1\,0\,\underline{0\,1}\,0\,1\,0\,1\,1 & \to & 1\,1\,\underline{0\,1}\,0\,0\,1\,0\,1\,1 \\
 & \to & 1\,1\,1\,0\,0\,\underline{0\,1}\,0\,1\,1 \\
 & \to & 1\,1\,1\,0\,\underline{0\,1}\,0\,0\,1\,1 \\
 & \to & 1\,1\,1\,\underline{0\,1}\,0\,0\,0\,1\,1 \\
 & & \text{etc.}
\end{array}
$$

□

Ein Markov-Algorithmus besteht nun aus einer Menge solcher Regeln, zusammen mit einer “Kontrollstruktur”, die eindeutig festlegt, wann welche Regel anzuwenden ist. Diese Information wird übersichtlich in einer Tabelle, der *Markov-Tafel*, zusammengefaßt. Diese hat fünf Spalten und beliebig viele (endlich viele) Zeilen. Eine Zeile ist ein 5-Tupel der Form

$$k \;\; \varphi \;\; \psi \;\; i \;\; j,$$

wobei $i, j, k \in \mathbb{N}$ sind, k ist die Zeilennummer, φ und ψ stellen die Regel $\varphi \to \psi$ dar, i ist die Nummer der nächsten Zeile, falls das Teilwort φ gefunden (die Regel also angewandt) wurde, und j ist die Nummer der nächsten Zeile, falls φ nicht auftrat.

Die *Ausführung* der Markov-Tafel beginnt in der ersten Zeile (Nummer 0) und stoppt sobald zu einer nicht vorhandenen Zeilennummer (i oder j) gegangen werden soll. Dazwischen werden die Regeln der angelaufenen Zeilen auf das Eingabewort nacheinander angewandt.

Bevor wir Beispiele betrachten definieren wir Markov-Algorithmen zunächst als abstrakte Maschinen:

Definition 4.7 Ein *Markov-Algorithmus* über dem Alphabet A ist gegeben durch

$X \subseteq A^*$	Eingabemenge
$Y \subseteq A^*$	Ausgabemenge
$K \subseteq \{(z, n) \mid z \in A^*, n \in \mathbb{N}\}$	Konfigurationen
$\alpha(x) \;=\; (x, 0)$	Eingabefunktion
$\omega(z, n)$	Ausgabefunktion
$\tau : K \to K$	Transitionsfunktion $\tau(z, n) \;=\; (g[\varphi \to \psi](z), n')$, wobei $\varphi \to \psi$ die Regel in der Zeile n ist und n' die Folgenummer (s.o., i bzw. j)
$\sigma(z, n)$	$\sigma(z, n) \;=\; \begin{cases} \text{true, falls n keine Zeilennummer ist} \\ \text{false, sonst} \end{cases}$

□

Beispiel 4.8 ”Addiere |”.

$$A = \{|\};\ X = A^*,\ Y = A^+ = A^* - \{\epsilon\}$$

k	φ	ψ	i	j
0	ϵ	$\vert$	1	$-$

berechnet die Funktion $f(|^n) = |^{n+1}$, $n \in \mathbb{N}$

Bemerkung 4.1 Das leere Wort ϵ kommt in jedem Wort vor. Das erste Auftreten ist ganz am Anfang. Der Algorithmus schreibt also ein | *vor* das Eingabewort $|^n = |\,|\,\ldots\,|$. Der j-Eintrag der Tabelle kann niemals angelaufen werden und ist daher irrelevant. Dies deuten wir durch das Zeichen $-$ in der Tabelle an. □

Beispiel 4.9 “Addition”.

$$A_0 = \{|\}; A = A_0 \cup \{+\}; X = A_0^* + A_0^* \ (= \{\mu + \nu \mid \mu, \nu \in A_0^*\}); Y = A_0^*$$

k	φ	ψ	i	j
0	$+$	ϵ	1	$-$

Der Algorithmus löscht das erste $+$ im Eingabewort, also:

$$f(|^n + |^m) = |^{n+m}$$

□

Beispiel 4.10 “Verdopplung”.

$$A_0 = \{|\}; A = A_0 \cup \{\#\}; X = Y = A_0^*$$

Das Zeichen # spielt die Rolle einer Markierung, die einmal von links nach rechts durchwandert und dabei die Zeichen | verdoppelt. In den Kommentaren der folgenden Tafel gilt $p = (n - q)$.

k	φ	ψ	i	j	Kommentar
0	$\vert$	$\#\vert$	1	3	$\vert^n \to \#\vert^n$
1	$\#\vert$	$\vert\,\vert\#$	1	2	$\vert^{2p}\#\vert^q \to \vert^{2(p+1)}\#\vert^{q-1}$ wiederholen bis $q = 1$
2	$\#$	ϵ	3	$-$	$\vert^{2p}\# \to \vert^{2n}$

Eine Berechnung mit dem Eingabewort | | | ergibt:

$$|\,|\,| \underset{0}{\to} \#|\,|\,| \underset{1}{\to} |\,|\#|\,| \underset{1}{\to} |\,|\,|\,|\#| \underset{1}{\to} |\,|\,|\,|\,|\,|\# \underset{2}{\to} |\,|\,|\,|\,|\,|$$

Allgemein wird die Funktion $f(|^n) = |^{2n}$ berechnet. □

Beispiel 4.11 “Multiplikation”.

$$A_0 \;=\; \{|\};\; A \;=\; A_0 \cup \{*, \#\}; X \;=\; A_0^* * A_0^*; \quad Y \;=\; A_0^*$$

Der folgende Markov-Algorithmus berechnet die Funktion $f(|^n * |^m) \;=\; |^{n*m}$:

k	φ	ψ	i	j	Kommentar
0	$*$	$**$	1	$-$	$\longrightarrow \; |^n * * |^m$
1	ϵ	$*$	2	$-$	$\longrightarrow \; * |^n * * |^m$
2	$**|$	$\#**$	3	6	
3	$|\#$	$\#|$	4	5	Faktor vorn
4	ϵ	$|$	3	$-$	kopieren
5	$\#$	ϵ	2	$-$	
6	$*|$	$*$	6	7	1. Faktor löschen
7	$***$	ϵ	8	$-$	Hilfsmarkierungen löschen

Mit der Eingabe $|||*||$ ergibt sich z.B. folgende Berechnungsfolge:

$$\begin{aligned}
|||*|| \;&\xrightarrow[0]{}\; |||**|| \;\xrightarrow[1]{}\; *|||**|| \\
&\xrightarrow[2]{}\; *|||\#**| \;\xrightarrow[3,4]{}\; |*||\#|*| \\
&\xrightarrow[3,4]{}\; ||*|\#||**| \;\xrightarrow[3,4,5]{}\; |||*|||**| \\
&\xrightarrow[2]{}\; |||*|||\#** \;\xrightarrow[3,4]{}\; ||||*||\#|** \\
&\xrightarrow[3,4]{}\; |||||*|\#||** \;\xrightarrow[3,4,5]{}\; ||||||*|||** \\
&\xrightarrow[(2)6]{}\; ||||||*||** \;\xrightarrow[6]{}\; ||||||*|** \;\xrightarrow[6]{}\; ||||||*** \;\xrightarrow[7]{}\; ||||||
\end{aligned}$$

□

Beispiel 4.12 “Kopieren”.

$$A_0 \;=\; \{0, 1\}; \quad A \;=\; A_0 \cup \{*, \#\}; X \;=\; A_0^*; \quad Y \;=\; A_0^* * A_0^*$$

Der folgende Markov-Algorithmus berechnet die Funktion

$$f(\mu) \;=\; \mu * \mu,\; \mu \in A_0^*$$

k	φ	ψ	i	j	Kommentar
0	ϵ	$*\#$	1	$-$	
1	$\#0$	$0\#$	2	3	kopiere nachstes Zeichen 0 vor $*$
2	$*$	$0*$	1	$-$	und wiederhole
3	$\#1$	$1\#$	4	5	kopiere nachstes Zeichen 1 vor $*$
4	$*$	$1*$	1	$-$	und wiederhole
5	$\#$	ϵ	6	$-$	

Mit der Eingabe 0 1 0 ergibt sich folgende Berechnung:

$$
\begin{aligned}
0\,1\,0 \quad & \xrightarrow[0]{} \ast \# 0\,1\,0 \xrightarrow[1]{} \ast\, 0 \# 1\,0 \xrightarrow[2]{} 0 \ast 0 \# 1\,0 \\
& \xrightarrow[(1)3]{} 0 \ast 0\,1 \# 0 \xrightarrow[4]{} 0\,1 \ast 0\,1 \# 0 \\
& \xrightarrow[1]{} 0\,1 \ast 0\,1\,0 \# \xrightarrow[2]{} 0\,1\,0 \ast 0\,1\,0 \# \\
& \xrightarrow[(1,3)5]{} 0\,1\,0 \ast 0\,1\,0
\end{aligned}
$$

□

4.1.4. CHURCH'sche These

> *Die Klasse der* intuitiv berechenbaren *Funktionen stimmt mit den formalen Klassen der (Markov-, imperativ, applikativ, Registermaschinen-, etc.) berechenbaren Funktionen überein.*
>
> *Derartige Algorithmenmodelle heissen* universell.

Welche Funktionen können nun durch Markov-Algorithmen berechnet werden? Leisten Markov-Algorithmen mehr oder weniger als applikative bzw. imperative Algorithmen? Wie kann man überhaupt die Leistungen von Algorithmenmodellen mieteinander vergleichen?

Einem direkten Leistungsvergleich steht zunächst der Umstand im Wege, daß die Algorithmen auf ganz verschiedenen Argument- und Wertebereichen definiert sind:

- applikative Algorithmen:	$f: \mathbb{Z}^n \to \mathbb{Z}$, n variabel
- imperative Algorithmen:	$f: \mathbb{Z}^n \to \mathbb{Z}^m$, m,n, variabel
- Markov-Algorithmen:	$f: X \to Y, \quad X, Y \subseteq A^*$
- Registermaschinen-Algorithmen	$f: \mathbb{Z}^n \to \mathbb{Z}^m$, m,n, variabel

Um vergleichen zu können, muß man zunächst die Argument- und Wertebereiche "vereinheitlichen", d.h. man muß einen Argument- und Wertebereich finden, mit dem sich alle Ein- und Ausgaben aller Algorithmenmodelle eindeutig und effektiv (i.e. berechenbar) *codieren* lassen.

Einen solchen für die Zwecke der Berechenbarkeit "universellen" Bereich bilden u.a. Texte A^* über einem festen Alphabet $A = \{a_1, \ldots, a_n\}$: derselbe wie für die Algorithmen!

Es gilt das folgende wichtige, grundlegende (und vielleicht überraschende) Ergebnis:

$$F_M = F_a = F_i = \ldots$$

Alle diese und weitere Algorithmenmodelle leisten also prinzipiell gleich viel.

Den Beweis dieses Satzes wollen wir hier nicht durchführen. Er wird konstruktiv geführt, indem man allgemeine “Übersetzer” angibt, die jeden Algorithmus des einen Modells in einen gleichwertigen des anderen Modells überführen, der dieselbe Funktion berechnet. Die Konstruktion solcher Übersetzer ist - auch für einfache mathematische Algorithmenmodell - sehr aufwendig und langwierig.

Bemerkung 4.2 Auch viele andere Algorithmenmodelle führen auf diese gleiche Klasse berechenbarer Funktionen, z.B. Rekursive Funktionen, λ-Kalkül, Turing-Maschinen, Post’sche Normalsysteme, Random-Access-Maschinen, Universelle Register-Maschinen, *goto*-Programme, etc.

Derartige Ergebnisse haben 1936 den Logiker und Mathematiker A. Church veranlaßt, folgende - prinzipiell nich beweisbare - These aufzustellen:

> *Church’sche These:* Die Klasse der *intuitiv berechenbaren* Funktionen stimmt mit dieser Klase der (Markov-, applikativ, imperativ, etc.) berechenbaren Funktionen überein.

Bemerkung 4.3 Diese These wurde nahezu gleichzeitig unabhängig von mehreren Personen aufgestellt. Sie ist daher auch als Turing-Church-These bekannt. □

Daß jede in einem der Algorithmenmodelle berechnbare Funktion auch intuitiv berechenbar ist, ist klar. Die wesentliche Aussage der These besagt, daß jede Funktion, die man intuitiv als berechenbar ansehen würde, tatsächlich auch in jedem dieser Algorithmenmodelle berechenbar ist. Das besagt im Grunde, daß der intuitive Begriff der Berechenbarkeit durch die mathmatische Präzisierung korrekt und vollständig erfaßt ist.

Definition 4.8 Ein Algorithmenmodell heißt *universell*, wenn es alle berechenbaren Funktionen zu beschreiben gestattet. □

Bemerkung 4.4 Nahezu alle Programmiersprachen, höhere sowie niedere, sind universell. □

4.2. Berechenbarkeit und Entscheidbarkeit

4.2.1. Existenz nichtberechenbarer Funktionen

Wir gehen der Frage nach, welche Funktionen sich überhaupt durch Algorithmen — applikative, imperative oder andere — berechnen lassen.

Gibt es z.B. Funktionen, die sich nicht durch irgendeinen Algorithmus berechnen lassen?

Daß dies tatsächlich der Fall sein muß, läßt sich unschwer einsehen. Wir benutzen hierzu nur eine — sicherlich unabdingbare — Eigenschaft eines jeden Algorithmus, nämlich:

Jeder Algorithmus läßt sich durch einen endlichen Text über einem festen, endlichen Alphabet beschreiben.

Sei $A = \{a_1, \ldots, a_n\}$ ein Alphabet mit der alphabetischen Ordnung $a_1 < a_2 < \ldots < a_n$. Mit A^* bezeichnen wir die Menge der *Texte* (Zeichenketten, endlichen Folgen, Worte) über A:

$$A^* = \{\epsilon, a_1, \ldots, a_n, a_1a_1, a_1a_2, \ldots, a_5a_3a_3a_1a_5a_2, \ldots\}$$

Hierbei ist ϵ der leere Text der Länge 0.

Wir können A^* nun auflisten, und zwar

1. der Länge nach. Zu jeder Länge l gibt es n^l verschiedene Texte über A, also endlich viele.

2. lexikographisch innerhalb der Texte gleicher Länge:

$$\begin{aligned} b_1b_2\ldots b_l \;<\; c_1c_2\ldots c_l \;:\Leftrightarrow\; & b_1 \;<\; c_1 \\ & \vee(b_1 = c_1 \;\wedge\; b_2 < c_2) \\ & \vee(b_1 = c_1 \;\wedge\; b_2 = c_2 \;\wedge\; b_3 < c_3) \\ & \ldots \\ & \vee(b_1 = c_1 \;\wedge\; \ldots \;\wedge\; b_{l-1} = c_{l-1} \;\wedge\; b_l < c_l) \end{aligned}$$

Hierdurch ist eine Ordnung auf A^* eindeutig bestimmt, so daß wir von dem ersten, zweiten, dritten, etc. Text über A^* (gemäß dieser Ordnung) reden können. Daraus folgt:

Satz 4.1 A^* ist abzählbar. □

Da es nicht mehr berechenbare Funktionen als sie berechnende Algorithmen und nicht mehr Algorithmen als sie beschreibende Texte geben kann, gilt offenbar folgende Aussage:

Satz 4.2 Die Menge der berechenbaren Funktionen ist abzählbar. □

Betrachten wir nun speziell einstellige Funktionen auf $\mathbb{Z}$, $f : \mathbb{Z} \to \mathbb{Z}$. Wie bewiesen, gibt es nur abzählbar viele berechenbare solche Funktionen. Funktionen insgesamt gibt es jedoch weit mehr, wie der folgende Satz zeigt:

Satz 4.3 Die Menge $F = \{f \mid f : \mathbb{Z} \to \mathbb{Z}\}$ der einstelligen Funktionen auf $\mathbb{Z}$ überabzählbar.

Beweis: Wir nehmen an, F sei abzählbar, so daß wir nun alle $f \in F$ auflisten können, $F = \{f_0, f_1, \ldots\}$, d.h. jedes f hat eine Nummer i in dieser Folge. Sei nun $g : \mathbb{Z} \to \mathbb{Z}$ definiert durch

$$g(x) \quad = \quad f_{abs(x)}(x) + 1$$

Dann ist für $i = 1, 2, \ldots g(i) \neq f_i(i)$, also ist für $i = 1, 2, \ldots$ immer $g \neq f_i$. g kommt demnach in der obigen Folge nicht vor. Offensichtlich ist g aber eine einstellige Funktion auf $\mathbb{Z}$ und müßte somit in der obigen Folge *aller* dieser Funktionen vorkommen. Der Widerspruch läßt sich nur lösen, wenn wir die Annahme fallenlassen, F sei abzählbar. □

Berechenbare Funktionen sind also unter allen Funktionen genauso “seltene” Ausnahmen wie ganze (oder natürliche oder rationale) Zahlen unter den reellen.

Bemerkung 4.5 Die obige Beweismethode ist unter dem Namen “Cantorsches Diagonalverfahren” bekannt. Auf ähnliche Weise hat Cantor erstmals bewiesen, daß die reellen Zahlen überabzählbar sind.

Nachdem wie wissen, daß berechenbare Funktionen eher seltene Ausnahmen sind, ist die Frage naheliegend, ob sich nicht-berechenbare Funktionen *konkret angeben* lassen. Um zu beweisen, daß eine gegebene Funktion berechenbar ist, braucht man nur einen Algorithmus anzugeben, der sie berechnet. So schwierig dies in Einzelfall sein mag, so ist es doch prinzipiell schwieriger zubeweisen, daß eine gegebene Funktion *nicht* berechenbar ist. Die erfordert nämlich eine Beweisführung, über *alle denkbaren und möglichen Algorithmen!*

Derartige Ergebnisse über Algorithmen lassen sich nur gewinnen, wenn man den Begriff des Algorithmus mit hinreichender mathematischer Genauigkeit definiert, wie dies am Anfang dieses Kapitels geschah.

4.2.2. Konkrete Nicht-berechenbare Funktionen

Nun kehren wir zu dem Problem zurück, eine nicht berechenbare Funktion konkret anzugeben. Wir benutzen dazu ein beliebiges Algorithmenmodell, welches folgenden Kriterien genügt:

1. Berechnet werden partielle Funktionen $f : A^* \to A^*$ über einem festen Alphabet A.

2. Auch die Algorithmen selbst lassen sich als Text über A darstellen.

Damit können Algorithmentexte wiederum als Eingaben für Algorithmen genommen werden!

Bemerkung 4.6 Z.B. lassen sich Markov-Algorithmen leicht als Text über einem geeigneten Alphabet A codieren. Codierungen von Algorithmen durch natürliche Zahlen gehen auf K. Gödel (1931) zurück. Man spricht von “Gödelisierung”. □

Definition 4.9 Sei $x \in A^*$. Dann bezeichnet φ_x die *vom Algorithmus mit Text x berechnete Funktion*. Ist x kein sinnvoller Algorithmentext, so sei φ_x überall undefiniert.

Definition 4.10 Sei $f: A^* \to A^*$ eine partielle Funktion. Dann ist

$$dom\ f := \{x \in A^* \mid f(x) \text{ ist definiert } \}$$

der *Urbildbereich* von f ("domain"). □

Nun sind wir endlich in der Lage, eine nicht berechenbare Funktion konkret anzugeben. Dazu sei $a \in A$ ein fest gewählter Buchstabe.

Satz 4.4 Die (totale) Funktion $h: A^* \to A^*$,

$$h(x) = \begin{cases} \epsilon, & \text{falls } x \in dom\ \varphi_x \\ a & \text{sonst} \end{cases} \tag{4.1}$$

ist nicht berechenbar. □

Die obige Funktion h ist mathematischer Ausdruck eines recht anschaulichen Problems, des sog. *Halteproblems*. h entscheidet durch die Ausgabe ϵ oder a, ob $x \in dom\ \varphi_x$ ist oder nicht. Nun bedeutet $x \in dom\ \varphi_x$, daß $\varphi_x(y)$ definiert ist, und das heißt, daß der Algorithmus mit Text x bei Eingabe von y irgendwann anhält. Die Funktion h drückt also die Frage aus:

"Hält ein Algorithmus irgendwann an,
wenn man ihm seinen eigenen Text eingibt?"

4.2.3. Das Halteproblem

Entscheidungsprobleme, derern Entscheidungsfunktion nicht berechenbar ist, nennt man *nicht entscheidbar*. Es gibt eine große Fülle nicht entscheidbarer Probleme. Das Problem $x \in dom\ \varphi_x$ ist ein spezielles Halteproblem, auch *Selbstanwendungsproblem* genannt. Das *allgemeine Halteproblem* ist $y \in dom\ \varphi_x$, also:

"Hält Algorithmus x bei der Eingabe von y ?"

Korollar 4.5 Das allgemeine Halteproblem ist nicht entscheidbar.

Beweis: Gäbe es einen Entscheidungsalgorithmus für $y \in dom\ \varphi_x$, so könnte man damit auch speziell $x \in dom\ \varphi_x$ entscheiden. □

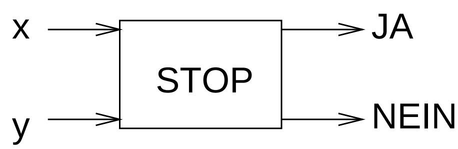
Bemerkung 4.7 So einfach dieser Beweis ist, er zeigt das typische Grundmuster, nämlich "Reduktion auf ein als nicht entscheidbar bewiesenes Problem". Fast alle Nicht-Entscheidbarkeits-Beweise sind so konstruiert: "Wäre das (neue) Problem Y entscheibar, so auch das (alte) Problem X". Das Halteproblem spielt hierbei die Rolle einer Wurzel, auf die letzten Endes alle Nicht-Entscheidbarkeits-Probleme reduziert werden. □

Wegen der grundlegenden Bedeutung des Halteproblems für die Theorie der Entscheidbarkeit wollen wir zunächst einen "anschaulichen" Beweis für seine Nicht-Entscheidbarkeit liefern.

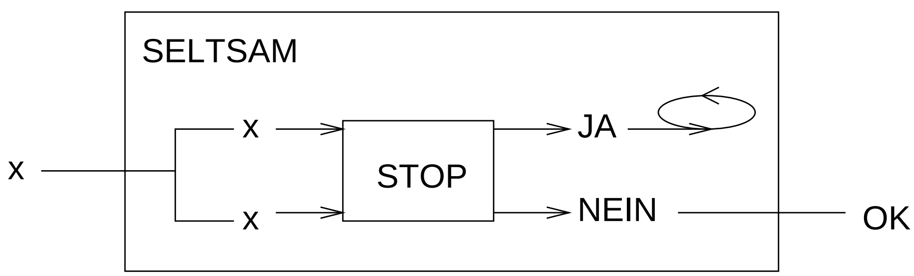
Angenommen, wir hätten eine Maschine (Algorithmus) STOP mit zwei Eingaben, nämlich einem Algorithmentext x und einer Eingabe y für x, sowie zwei Ausgaben:

JA: x stoppt bei Eingabe von y
NEIN: x stoppt nicht bei Eingabe von y

Graphisch dargestellt:



Mit dieser Maschine STOP könnten wir dann eine neue Maschine SELTSAM konstruieren:



Bei Eingabe von x wird getestet, ob x bei Eingabe von x stoppt. Im JA-Fall wird in eine Endlosschleife gegangen, die nie anhält. Im NEIN-Fall hält SELTSAM an mit der Anzeige OK.

Nun geben wir SELTSAM als Eingabe für sich selbst ein und fragen:

Hält SELTSAM bei der Eingabe von SELTSAM?

1. Wenn ja, so wird der JA-Ausgang von STOP angelaufen und SELTSAM gerät in die Endlosschleife, hält also nicht. Widerspruch!

2. Wenn nein, so wird der NEIN-Ausgang von STOP angelaufen und SELTSAM stoppt mit OK. Widerspruch!

Diese Widersprüche folgen aus der Annahme, daß eine STOP-Maschine existiert, was daher verneint werden muß.

Beweis 4.6 Mathematischer Beweis von Satz 4.4: Wir nehmen an, h sei berechenbar, dann ist aufgrund der Church'schen These auch die Funktion $g: A^* \to A^*$

$$g(x) = \begin{cases} \varphi_x(x)a, & \text{falls } h(x) = \epsilon \\ \epsilon & \text{sonst} \end{cases}$$

berechenbar. Nun ist sicherlich $g(x) \neq \varphi_x(x)$ für alle $x \in A^*$. Da g berechenbar ist, gibt es eine Algorithmus mit einem Text $y \in A^*$, der g berechnet, d.h. es ist $g = \varphi_y$. Damit folgt nun:

$$\varphi_y(y) = g(y) \neq \varphi_y(y)$$

Dies ist offenbar ein Widerspruch, der sich nur dadurch lösen läßt, daß wir die Annahme aufgeben, h sei berechenbar. □

Bemerkung 4.8 Dies ist wiederum ein Diagonalbeweis.

4.2.4. Nicht-entscheidbare Probleme

Das Halteproblem ist ein Beispiel für ein *semantisches* Problem von Algorithmen, nämlich ein Problem der Art:

Kann man anhand eines Programmtextes (Syntax) entscheiden, ob die berechnete Funktion (Semantik) eine bestimmte Eigenschaft hat?

Beim Halteproblem ist dies die Eigenschaft, ob die berechnete Funktion an einer bestimmten Stelle definiert ist. Wie steht es nun mit anderen semantischen Eigenschaften von Algorithmen?

Aus einem tiefliegenden Satz der Algorithmentheorie (Satz von Rice) folgt die folgende bemerkenswerte Aussage:

Jede nicht triviale semantische Eigenschaft von Algorithmen ist nicht-entscheidbar!

Dabei ist eine Eigenschaft genau dann trivial, wenn entweder jede oder keine berechnete Funktion sie hat. Nichttriviale semantische Eigenschaften sind demnach solche, die manche berechneten Funktionen haben und manche nicht.

Nicht entscheidbar sind also u.a. folgende Probleme:

1. Ist die berechnete Funktion total? überall undefiniert? injektiv? surjektiv? bijektiv? etc. etc.

2. Berechnen zwei gegebene Algorithmen dieselbe Funktion?

3. Ist ein gegebener Algorithmus korrekt, d.h. berechnet er die gegebene (gewünschte) Funktion?

Diese Ergebnisse bedeuten nicht, daß man solche Fragen nicht *im Einzelfall* entscheiden könnte! Dies ist durchaus möglich. So behandeln wir z.B. im nächsten Abschnitt das Problem, die Korrektheit von Algorithmen nachzuweisen. Es ist jedoch prinzipiell unmöglich, eine allgemeine Methode hierfür zu finden, also z.B. *einen* Algorithmus, der die Korrektheit *aller* Algorithmen nachweist (und damit auch seine eigene!).

4.2.5. Post'sches Korrespondenzproblem

Ein besonders einfaches und verblüffendes nicht-entscheidbares Problem ist das *Post'sche Korrespondenzproblem* (E. Post): Gegeben seien ein Alphabet A und zwei gleichlange Listen von Worten über A:

$$\begin{aligned} \alpha &= (\alpha_1, \alpha_2, \ldots, \alpha_n) \\ \beta &= (\beta_1, \beta_2, \ldots, \beta_n) \end{aligned}$$

wobei $\alpha_i, \beta_i \in A^+ = A^* - \{\epsilon\}$ und $n \geq 1$. Das Problem besteht darin, eine "Korrespondenz" zu finden, d.h. eine endliche Folge $(i_1, i_2 \ldots, i_k)$, $i_j \in \{1, \ldots, n\}$ für $j = 1, 2, \ldots, k$, so daß gilt:

$$\alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_k} = \beta_{i_1} \beta_{i_2} \ldots \beta_{i_k}$$

Beispiel 4.13

$$A = \{0,1\} \quad \begin{aligned} \alpha &= (1\ ,\ 1\,0\,1\,1\,1\ ,1\,0) \\ \beta &= (1\,1\,1\ ,\ 1\,0\ ,\ 0) \end{aligned}$$

Dieses Post'sche Korrespondenzproblem besitzt eine Lösung, nämlich (2,1,1,3):

$$1\,0\,1\,1\,1\,.\,1\,.\,1\,.\,1\,0 = 1\,0\,.\,1\,1\,1\,.\,1\,1\,1\,.\,0$$

□

Beispiel 4.14

$$A = \{0,1\} \quad \begin{aligned} \alpha &= (1\,0\ ,\ 0\,1\,1\ ,\ 1\,0\,1) \\ \beta &= (1\,0\,1\ ,\ 1\,1\ ,\ 0\,1\,1) \end{aligned}$$

Dieses Post'sche Korrespondenzproblem besitzt keine Lösung. Gäbe es nämlich eine, so müßte sie mit 1 anfangen:

$$1\,0\ \ldots \stackrel{?}{=} 1\,0\,1\ \ldots$$

Als zweites müßte ein Index i mit $\alpha_i = 1\ldots$ gewählt werden, also 1 oder 3. Aber $(1,\ 1,\ \ldots)$ ergibt

$$1010\ldots \neq 101101\ldots$$

und $(1,\ 3,\ldots)$ ergibt

$$10101\ldots \stackrel{?}{=} 101011\ldots$$

Die gleiche Überlegung wie oben führt dazu, daß wieder 3 gewählt werden muß. $(1,\ 3,\ 3,\ldots)$ ergibt

$$10101101\ldots \stackrel{?}{=} 101011011\ldots$$

Wieder müßte 3 gewählt werden, usw. Es ist nun zu sehen, daß niemals eine Lösung entstehen kann, da die rechte Seite der linken immer um 1 voraus ist. □

Während wir in diesen Beispielen die Lösbarkeit entscheiden können, ist das allgemeine Problem, nämlich zu entscheiden, ob eine Lösung für ein gegebenes Post'sches Korrespondenzproblem über einem Alphabet mit mindestens zwei Buchstaben existiert, nicht entscheidbar.

4.3. Korrektheit von Algorithmen

Es ist ganz wesentlich, daß Algorithmen, die wir für eine bestimmte Aufgabe entworfen haben, korrekt sind, d.h. daß sie sich genauso verhalten, wie wir es beabsichtigt hatten. Bei einigen hochsensiblen und kritischen Anwendungen(z.B. in der medizinischen Informatik, bei der Steuerung von Waffensystemen, aber auch in der Baustatik) können durch inkorrekte Algorithmen hervorgerufene Fehlfunktionen schwere Schäden anrichten oder gar Katastrophen auslösen.

Nun haben wir im vorherigen Abschnitt gesehen, daß die Korrektheit von Programmen im allgemeinen nicht entscheidbar ist. In vielen Anwendungsfällen wird man mit pragmatischem Austesten eine hinreichende Sicherheit erreichen können. Will man jedoch die Korrektheit eines Algorithmus' mit mathematischer Exaktheit und Strenge nachweisen, so ist man auf ad-hoc-Beweisführung im Einzelfall angewiesen. Dies ist nicht nur bei sicherheitskritischen Anwendungen angezeigt, sondern z.B. auch bei Standard-Algorithmen, die hardwaremäßig, etwa als VLSI-Chip, realisiert und in großer Serie produziert werden sollen. Ein großer Aufwand bei dem Nachweis der Korrektheit lohnt sich hier, da Fehler große wirtschaftliche Verluste bewirken.

In diesem Abschnitt wollen wir einige grundlegende Konzepte und Vorgehensweisen für Korrektheitsbeweise für Algorithmen erörtern. Die Problematik kann hier nicht tief und ausführlich abgehandelt werden. Wir beschränken uns darauf, Problematik und Vorgehensweise anhand einfacher Beispiele zu demonstrieren.

4.3.1. Relative Korrektheit

Der Nachweis der *Korrektheit* eines Algorithmus bezieht sich immer auf eine *Spezifikation* dessen, was er tun *soll*. Insofern handelt es sich immer um eine *relative* Korrektheit.

- *Verifikation*: formaler Beweis der Korrektheit bezüglich einer formalen Spezifikation
- *Validation*: (nicht-formaler) Nachweis der Korrektheit bezüglich einer informellen oder formalen Spezifikation (etwa systematisches Testen)

4.3.2. Korrektheit von imperativen Algorithmen

Wir behandeln zunächst imperative Algorithmen. Eine verbreitete Methode, die gewünschten Eigenschaften von Algorithmen zu spezifizieren, ist die Angabe von Vor- und Nachbedingungen:

$$(*) \qquad \{\text{ VOR }\} \quad \text{ANW} \quad \{\text{ NACH }\}$$

VOR und NACH sind dabei Aussagen über den Zustand vor bzw. nach Ausführung der Anweisung ANW. Genauer bedeutet die Aussage $(*)$:

Gilt VOR unmittelbar vor Ausführung von ANW und terminiert ANW, so gilt NACH unmittelbar nach Ausführung von ANW.

Terminiert ANW nicht, so ist $(*)$ trivialerweise wahr, wie auch immer VOR und NACH aussehen!

Entsprechend ist $(*)$ trivialerweise wahr, wenn VOR nicht gilt, gleichgültig, ob ANW terminiert oder nicht und ob NACH gilt oder nicht!

Beispiel 4.15 Anweisungen über $\mathbb{Z}$

$\{X = 0\}$	$X := X + 1$	$\{X = 1\}$	ist wahr
$\{\text{true}\}$	$X := Y$	$\{X = Y\}$	ist wahr
$\{Y = a\}$	$X := Y$	$\{X = a \wedge Y = a\}$	ist wahr für alle $a \in \mathbb{Z}$
$\{X = a \wedge Y = b$ $\wedge a \neq b\}$	$X := Y;\ Y := X$	$\{X = b \wedge Y = a\}$	ist *falsch* für alle $a, b \in \mathbb{Z}$
$\{X = a \wedge Y = b\}$	$Z := X;\ X := Y;$ $Y := Z$	$\{X = b \wedge Y = a\}$	ist wahr für alle $a, b \in \mathbb{Z}$
$\{\text{false}\}$	ANW	$\{NACH\}$	ist wahr für alle ANW, NACH
$\{\text{true}\}$	ANW	$\{\text{false}\}$	ist genau dann wahr, wenn ANW nicht terminiert
$\{X = a\}$ **while**	$X \neq 0$ **do** $X := X - 1$	**od** $\{X = 0\}$	ist wahr für alle $a \in \mathbb{Z}$! Auch für $a < 0$, da dann die **while**-Schleife nicht terminiert!)

□

Bei der Korrektheit *imperativer* Algorithmen bzgl. gegebener Vor- und Nachbedingungen unterscheidet man zwischen *partieller* und *totaler* Korrektheit. Zur Definition dieser Begriffe sei ein Algorithmus PROG gegeben:

```
PROG: var  X,Y, ... :  int ;
             P,Q ... :  bool ;
      input  X_1,...,X_n;
                α ;
      output  Y_1,...,Y_m  .
```

Ferner gegeben seien eine Vorbedingung VOR und eine Nachbedingung NACH für die Anweisung α von Prog gegeben.

Definition 4.11 PROG heißt *partiell korrekt* bzgl. VOR und NACH gdw.

$$\{VOR\}\alpha\{NACH\}$$

wahr ist (s.o.).

PROG heißt *total korrekt* bzgl. VOR und NACH gdw. PROG partiell korrekt ist bzgl. VOR und NACH, und wenn α darüberhinaus immer dann terminiert, wenn vorher VOR gilt. □

Im Rahmen dieser Vorlesung wird darauf verzichtet, ein Regelwerk zum Beweisen von Vor- und Nachbedingungen anzugeben. Es sei stattdessen auf spätere Vorlesungen und weiterführende Literatur vertröstet. Hier nur folgende Bemerkungen:

- Das Finden von geeigneten Vor- und Nachbedingungen ist nicht einfach!
- Für jedes Konstrukt imperativer Algorithmen muß eine Beweisvorgehensweise entwickelt werden. Wir werden dies für den Fall der Schleifen demonstrieren.
- Für eine atomare Anweisung $X := T$ kann man eine Vorbedingung in eine Nachbedingung transferieren, in dem jedes Auftreten von T in VOR durch X ersetzt wird. So kann man etwa folgende Bedingungen zeigen:

  $$\{3 > 0\}X := 3\{X > 0\}$$

  $$\{X + 1 > 0\}X := X + 1\{X > 0\}$$

  Dies ist natürlich oft nicht praktikabel, da der Term T in der Vorbedingung auftauchen muß, um eine geeignete Nachbedingung zu erzeugen (dies kann man mittels Tautologien und Umformungen erreichen).

  Praktikabler ist es, in der Nachbedingung die Variablen X als 'neue Werte' X' zu kennzeichnen, und deren Eigenschaften dann auf die alten Werte der Vorbedingung zurückzuführen. Die folgenden Beispiele zeigen diese Vorgehensweise.

4.3.3. Schleifeninvarianten

Um eine Bedingung über eine Schleife der Form

```
{ VOR }
while B do β od
{ NACH }
```

zu überprüfen, müssen wir eine *Schleifeninvariante* P finden, die auch bei mehrfachen Schleifendurchläufen jeweils ihre Gültigkeit bewahrt. Im Einzelnen müssen wir dann folgendes prüfen:

1. $VOR \implies P$.

   Die Vorbedingung garantiert die Gültigkeit von P beim ersten Eintritt in die Schleife.

2. $\{P \wedge B\}\alpha\{P\}$.

   Der Schleifenrumpf bewahrt die Schleifenvariante (nur in den Fällen notwendig, in denen der Rumpf tatsächlich durchlaufen wird).

3. $\{P \wedge \neg B\} \implies NACH$.

4.3.4. Korrektheit imperativer Algorithmen an Beispielen

Beispiel 4.16 Wir betrachten als Beispiel den Algorithmus MULT, der die Multiplikation nur mittels Addition und Subtraktion realisiert.

```
MULT: var  W,X,Y,Z : int ;
      input  X, Y
      Z:=0;
      W:=Y;
      while  W ≠ 0  do  Z:=Z+X;  W:=W-1 od;
      output  Z.
```

Der Schleifenrumpf sei wieder mit β bezeichnet.

- Vorbedingung: $\{Y > 0\}$
- Nachbedingung: $\{Z = X * Y\}$
- Schleifeninvariante: $P = (X * Y = Z + W * X)$

Zu beweisen:

1. $\{Y > 0\}Z := 0; W := Y\{X' * Y' = Z' + W' * X' = 0 + Y * X\}$

   P gilt vor dem ersten Schleifendurchlauf!

2. $\{X * Y = Z + W * X \land W \neq 0\}$
   $Z := Z + X;\ W := W - 1$
   $\{X * Y = X' * Y' = Z' + W' * X' =$
   $(Z + X) + (W - 1) * X = Z + X + W * X - X = Z + W * X\}$

   Der Rumpf erhält P.

3. $(P \land \neg B) \equiv (X * Y = Z + W * X \land W = 0) \equiv (X * Y = Z)$

Dies beweist die partielle Korrektheit. Einen Beweis einer totalen Korrektheit zeigt das nächste Beispiel. □

Als zweites Beispiel wird die Korrektheit des Algorithmus XYZ aus Abschnitt 2.7 betrachtet.

Beispiel 4.17 Algorithmus XYZ

```
XYZ:  var  W,X,Y,Z : int ;
      input  X
      Z:=0;  W:=1;  Y:=1 ;
      while  W ≤ X  do  Z:=Z+1;  W:=W+Y+2;  Y:=Y+2  od;
      output  Z.
```

Seien

$$VOR = (X \geq 0)$$

und

$$NACH = (Z^2 \leq X < (Z+1)^2) = (Z = \lfloor\sqrt{X}\rfloor).$$

Wir beweisen, daß XYZ total korrekt ist bzgl. VOR und NACH. Dazu sei

$$\begin{aligned}
\alpha &= (Z := 0;\ W := 1;\ Y := 1)\\
\beta &= (Z := Z + 1;\ W := W + Y + 2;\ Y := Y + 2)\\
P &= (Z^2 \leq X \ \land\ (Z+1)^2 = W \ \land\ 2 \cdot Z + 1 = Y \ \land Y > 0)
\end{aligned}$$

P ist eine “Zwischenbedingung”, von der wir zeigen, von der wir zeigen, daß sie am Anfang jeden Durchlaufs durch die **while**-Schleife gilt (die sogenannte “*Schleifeninvariante*”), und daß sie nach Ablauf der **while**-Schleife die Nachbedingung NACH impliziert. Daß heißt, wir zeigen:

1. $\{VOR\}\ \alpha\ \{P\}$
2. $\{P \land W \leq X\}\ \beta\ \{P\}$
3. $P \land W > X \ \Rightarrow\ NACH$

Haben wir dies bewiesen, so folgt daraus, daß XYZ partiell korrekt ist bzgl. VOR und NACH.

zu 1. Mit den Werten $Z = 0,\ W = Y = 1$ nach α gilt dort P.

zu 2. Offenbar gilt:

$\{(Z+1)^2 = W \ \wedge\ W \leq X\}\ \beta\ \{Z^2 \leq X\}$

$\{(Z+1)^2 = W \ \wedge\ 2 \cdot Z + 1 = Y\}\ \beta$

$\{W = Z^2 + 2\ \cdot (Z-1) + 1 + 2\ =\ Z^2 + 2 \cdot Z + 1\ =\ (Z+1)^2\}$

$\{2 \cdot Z + 1 = Y\}\ \beta\ \{Y = 2 \cdot (Z-1) + 1 + 2\ =\ 2 \cdot Z + 1\}$

$\{Y > 0\}\ \beta\ \{Y > 0\}$

Hieraus folgt insgesamt $\{P \ \wedge\ W \leq X\}\ \beta\ \{P\}$

zu 3. $P \ \wedge\ W > X \ \Rightarrow\ Z^2 \leq X \ \wedge\ X < (Z+1)^2 \ \Rightarrow\ NACH$

Damit ist die partielle Korrektheit bewiesen. Man beachte, daß hierfür die Bedingung $Y > 0$ nicht benötigt wurde. Sie wird erst zu *Beweis der Terminierung* benötigt.

Hierzu zeigen wir, daß für den Wert von $u = X - W$ gilt:

1. $W \leq X \ \Rightarrow\ u \geq 0$, d.h. u bleibt bei allen Schleifendurchläufen nichtnegativ.

2. u wird bei jedem Schleifendurchlauf echt kleiner.

Haben wir dies bewiesen, so folgt daraus, daß es nur endlich viele Schleifendurchläufe geben kann, daß XYZ also (bzgl. VOR) terminiert.

zu 1. Dies ist offensichtlich.

zu 2. Sei u der Wert unmittelbar vor Ablauf von β, und sei u' der Wert unmittelbar danach. Dann gilt:

$u\ =\ X - W$

$u'\ =\ X - (W + Y + 2)\ =\ X - W - Y - 2$

Da $Y > 0$ ist (!), ist $u' < u$.

Dies vervollständigt der Beweis der totalen Korrektheit von XYZ bzgl. VOR und NACH. □

4.3.5. Korrektheit applikativer Algorithmen

Für den Nachweis von Eigenschaften — wie z.B. der Korrektheit — *applikativer* Algorithmen verwendet man typischerweise *Induktionsbeweise*, die der Struktur der rekursiven Funktionsdefinition folgen.

Beispiel 4.18 Mc'Carthy's 91-Funktion:

$$f(x) \; = \; \textbf{if} x > 100 \; \textbf{then} \; x - 10 \; \textbf{else} \; f(f(x+11)) \; \textbf{fi}.$$
$$\text{Sei } g(x) \; = \; \textbf{if} \; x > 100 \; \textbf{then} \; x - 10 \; \textbf{else} \; 91 \; \textbf{fi}.$$

Wir beweisen: $f(x) = g(x)$ für alle $x \in \mathbb{Z}$.

1. Induktionsanfang:

   Für $x > 100$ ist $f(x) \; = \; x - 10 \; = \; g(x)$.

   Die Induktion verläuft "rückwärts", d.h. wir zeigen:

   Gilt $f(y) = g(y)$ für $y \geq x$, so gilt auch $f(x-1) = g(x-1)$; also:

2. Induktionsannahme:

   Es gelte $f(y) = g(y)$ für $y \geq x$

3. Induktionsschritt:

   1. Fall: Sei $101 \geq x \geq 91$. Dann gilt:
      $f(x-1) \; = \; f(f(x-1+11)) \; = \; f(f(x+10)) \; = \; f(x+10-10) \; = \; f(x) \; = \; g(x) \; = \; 91 \; = \; g(x-1)$
   2. Fall: Sei $90 \geq x$. Dann gilt: $f(x-1) \; = \; f(f(x+10))$
      Aufgrund der Induktionsannahme ist $f(x+10) \; = \; g(x+10) \; = \; 91$, also folgt:
      $f(x-1) \; = \; f(91) \; = \; g(91) \; = \; 91 \; = \; g(x-1)$

Dies beweist $f(x) = g(x)$ für alle $x \in \mathbb{Z}$. □

4.4. Komplexität

Für die algorithmische Lösung eines gegebenen Problems ist es *unerläßlich*, daß der gefundene Algorithmus das Problem *korrekt* löst. Darüber hinaus ist es natürlich *wünschenswert*, daß er dies mit möglichst geringem *Aufwand* tut. Die Theorie der Komplexität von Algorithmen beschäftigt sich damit, gegebene Algorithmen hinsichtlich ihres Aufwands abzuschätzen und - darüber hinaus - für gegebene Problemklassen anzugeben, mit welchem Mindestaufwand Probleme dieser Klasse gelöst werden können.

4.4.1. Motivierendes Beispiel

Beispiel 4.19 Suche in Listen
Gegeben seien:

- eine Zahl $n \geq 0$,

- n Zahlen $a_1, \ldots, n$, alle verschieden
- eine Zahl b

Gesucht wird der Index $i = 1, 2, \ldots, n$, so daß $b = a_i$ ist, sofern ein solcher Index existiert. Sonst soll $i = n + 1$ ausgegeben werden.

Eine sehr einfache Lösung dieses Suchproblems ist die folgende (wobei standardmäßig $a_{n+1} = 0$ gesetzt sei:

$$i := 1; \textbf{ while } i \leq n \ \wedge \ b \neq a_i \textbf{ do } i := i + 1 \textbf{ od}$$

Am Ende hat i den gesuchten Ausgabewert.

Diese Lösung hängt von der Eingabe ab, d.h. von $n, a_1, \ldots, a_n$ und b, und zwar gilt:

1. Bei *erfolgreicher* Suche, wenn $b = a_i$ ist, werden $S = i$ Schritte benötigt.
2. Bei *erfolgloser* Suche werden $S = n + 1$ Schritte benötigt.

Die Aussage 1 hängt noch von zu vielen Parametern ab, um aufschlußreich zu sein. Man ist bemüht, globalere Aussagen zu finden, die nur von *einer* einfachen Größe abhängen. Hier bietet sich die Länge n der Eingabeliste an, und man beschränkt sich auf Fragen der Art:

A: Wie groß ist S für gegebenes n *im schlechtesten Fall?*

B: Wie groß ist S für gegebenes n *im Mittel?*

Wir analysieren den Fall der erfolgreichen Suche:

zu A: Im schlechtesten Fall wird b erst im letzten Schritt gefunden, d.h. $b = a_n$. Also gilt:

$$S = n \textit{ im schlechtesten Fall.}$$

zu B: Um eine *mittlere* Anzahl von Schritten zu berechnen, muß man Annahmen über die Häufigkeit machen, mit der - bei wiederholter Verwendung des Algorithmus mit verschiedenen Eingaben - b an erster, zweiter, ..., letzter Stelle gefunden wird. Wird b häufiger am Anfang der Liste gefunden, so ist die mittlere Anzahl von Suchschritten sicherlich kleiner, als wenn b häufiger am Ende der Liste gefunden wird. Als einfaches Beispiel nehmen wir die Gleichverteilung an, d.h.:

Läuft der Algorithmus N-mal, $N >> 1$, so wird n gleich häufig an erster, zweiter ..., letzter Stelle gefunden, also jeweils $\frac{N}{n}$-mal.

Dann werden für alle N Suchvorgänge insgesamt

$$M = \frac{N}{n} \cdot 1 + \frac{N}{n} \cdot 2 + \ldots + \frac{N}{n} \cdot n$$

Schritte benötigt. Die Auswertung ergibt:

$$M = \frac{N}{n}(1+2+\ldots+n) = \frac{N}{n} \cdot \frac{n(n+1)}{2} = N \cdot \frac{n+1}{2}$$

Für *eine* Suche werden im Mittel demnach $S = \frac{M}{N}$ Schritte benötigt, also:

$$S = \frac{n+1}{2} \quad im\ Mittel \qquad \text{(bei Gleichverteilung)}$$

□

Bemerkung 4.9 Unser Algorithmus ist für das gestellte Problem keineswegs der schnellste, sondern eher einer der langsameren. Es gibt Algorithmen, die die Suche in Listen (oder Tabellen) mit n Einträgen unter speziellen Voraussetzungen i.w. mit einer konstanten, nicht von n anhängigen, Anzahl von Schritten lösen! Die besten Suchverfahren für direkten Zugriff, die auch sequentielle Verarbeitung in der Ordnung der Schlüssel zulassen, benötigen eine logarithmische Anzahl von Schritten,d.h. $S = a \cdot \log_2(b \cdot n) + c$, wobei $a, b, c \in \mathbb{R}$ Konstante sind. □

4.4.2. Komplexitätsklassen

Meist geht es bei der Analyse der Komplexität von Algorithmen bzw. Problemklassen darum, als Maß für den Aufwand eine Funktion

$$f : \mathbb{N} \to \mathbb{N}$$

anzugeben, wobei $f(n) = a$ bedeutet: "Bei einem Problem der Größe n ist der Aufwand a''. Die *"Problemgröße"* n bezeichnet dabei in der Regel ein grobes Maß für den Umfang einer Eingabe, z.B. die Anzahl der Elemente in einer Eingabeliste oder dir Größe eines bestimmten Eingabewertes. Der *"Aufwand"* a ist in der Regel ein grobes Maß für die *Rechenzeit*, jedoch ist auch der benötigte *Speicherplatz* zuweilen Gegenstand der Analyse. Die Rechenzeit wird meist dadurch abgeschätzt, daß man zählt, wie häufig eine bestimmte Art von Operation ausgeführt wird, z.B. Speicherzugriffe, Multiplikationen, Additionen, Vergleiche, etc. Auf diese Weise erhält man ein maschinenunabhängiges Maß für die Rechenzeit.

Zur Illustration betrachten wir einige Beispiele. Wir benutzen dabei eine einfache **for**-Schleife, die folgendermaßen definiert ist:

$$\textbf{for } i := 1 \textbf{ to } u \textbf{ do } \alpha \textbf{ od} \quad = \quad i := 1;\ \textbf{while } i \leq u \textbf{ do } \alpha;\ i := i + 1 \textbf{ od}$$

Beispiel 4.20 Wie oft wird die Wertzuweisung “$x := x + 1$” in folgenden Anweisungen ausgeführt?

1.	$x := x + 1$	1mal
2.	**for** $i := 1$ **to** n **do** $x := x + 1$ **od**	n-mal
3.	**for** $i := 1$ **to** n **do** **for** $j := 1$ **to** n **do** $x := x + 1$ **od od**	n^2-mal

□

Die Aufwandsfunktion $f : \mathbb{N} \to \mathbb{N}$ läßt sich in den wenigsten Fällen exakt bestimmen. Vorherrschende Analysemethoden sind

- Abschätzungen des *Aufwandes im schlechtesten Fall*
- Abschätzungen des *Aufwandes im Mittel*

Selbst hierfür lassen sich im allgemeinen keine exakten Angaben machen. Man beschränkt sich dann auf “ungefähres Rechnen in Größenordnungen”. Das folgende Beispiel illustriert die Schwierigkeit, von exakten Analysen, insbesondere bei Abschätzungen im Mittel, bereits in recht einfachen Fällen.

Beispiel 4.21 Beispiel für Rechnen in Größenordnungen

Gegeben: $n \geq 0,\ a_1, \ldots, a_n \in \mathbb{Z}$

Gesucht: Der Index i der (ersten) größten Zahl unter den $a_1,\ i = 1, \ldots, n$.

Lösung

```
max := 1;
for i := 2 to n do
    if a_max < a_i then max := i fi
od
```

Am Ende enthält max den gesuchten Index (bzw. die Zahl 1, falls n=0 ist).

- *Analyse:* Wie oft wird die Anweisung “$max := i$” im Mittel ausgeführt, abhängig von n?

  Diese gesuchte mittlere Anzahl sei T_n. Offenbar gilt: $1 \leq T_n \leq n$.

- *Beobachtung:* “$max := i$” wird genau dann ausgeführt, wenn a_i das größte der Elemente $a_1, \ldots, a_i$ ist.

- *Annahme* (Gleichverteilung): Für jedes $i = 1, \ldots, n$ hat jedes der Element $a_1, \ldots, a_n$ die gleiche Chance, das größte zu sein.

Das heißt, daß bei N Durchläufen durch den Algorithmus, $N >> 1$, insgesamt $\frac{N}{n}$-mal die Anweisung "$max := i$" ausgeführt wird für $i = 1, \ldots, n$. Das heißt:

$max := 1$ wird N-mal, d.h. immer, ausgeführt
$max := 2$ wird $\frac{N}{2}$-mal ausgeführt
etc.

Daraus folgt für die gesamte Anzahl $N \cdot T_n$ von Ausführungen von "$max := i$" (für irgendein i) bei N Durchläufen:

$$\begin{aligned} N \cdot T_N &= N + \frac{N}{2} + \frac{N}{3} + \ldots + \frac{N}{n} \\ &= N \cdot (1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}) \end{aligned}$$

Damit haben wir $T_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$,
und dies ist H_n, die n-te harmonische Zahl. Für H_n ist keine exakte geschlossene Formel bekannt, jedoch eine ungefähre Abschätzung:

$$T_n = H_n \approx \ln n + \gamma,$$

wobei $\gamma = 0.57721566$ die Eulersche Konstante ist.

Da der Aufwand schon vom Ansatz her ohnehin nur grob abgeschätzt werden kann, macht eine Feinheit wie "$+\gamma$" im obigen Beispiel nicht viel Sinn. Interessant ist lediglich, daß T_n logarithmisch von n abhängt, nicht so sehr, wie dieser Zusammenhang im Detail aussieht. Man schreibt dafür:

$$T_n = O(\log n),$$

und dies bedeutet, daß T_n "von der Ordnung $\log n$" ist, wobei multiplikative und additive Konstante sowie die Basis des Logarithmus unspezifiziert bleiben. □

Die O-Notation läßt sich mathematisch exakt definieren. Seien $f, g : \mathbb{N} \to \mathbb{N}$ gegeben.

Definition 4.12 O-Notation

$$f(n) = O(g(n)) \;:\Leftrightarrow\; \exists c, n_0 \, \forall n \geq n_0 : \; f(n) \leq c \cdot g(n)$$

□

Das heißt, daß $\frac{f(n)}{g(n)}$ für genügend große n durch eine Konstante c beschränkt ist. Anschaulich bedeutet dies, daß f "nicht stärker wächst" als g.

Diese Begriffsbildung wendet man bei der Analyse von Algorithmen an, um Aufwandsfunktionen $f : \mathbb{N} \to \mathbb{N}$ durch Angabe einer *einfachen Vergleichsfunktion* $g : \mathbb{N} \to \mathbb{N}$ abzuschätzen, so daß $f(n) = O(g(n))$ gilt also das Wachstum von f durch das von g beschränkt ist.

Beispiel 4.22 Für das obige Bespiel gilt genauer

$$T_n = H_n = \ln n + \gamma$$
$$T_n = O(\log n)$$

Die Basis des Logarithmus ist unerheblich, da ein Basiswechsel gleichbedeutend ist mit der Multiplikation mit einer Konstanten: $\log_b n = \log_a n \cdot \log_b a$. □

Gebräuchliche Vergleichsfunktionen sind die folgenden:

$O(1)$	konstanter Aufwand
$O(\log n)$	logarithmischer Aufwand
$O(n)$	linearer Aufwand
$O(n \cdot \log n)$	
$O(n^2)$	quadratischer Aufwand
$O(n^k)$für ein $k \geq 0$	polyminaler Aufwand
$O(2^n)$	exponentieller Aufwand

Zur Veranschaulichung des Wachstums geben wir einige Werte ($\text{ld} = \log_2$).

$f(n)$	$n = 2$	$2^4 = 16$	$2^8 = 256$	$2^10 = 1024$	$2^20 = 1048576$
$\text{ld}n$	1	4	8	10	20
n	2	16	256	1024	1048576
$n \cdot \text{ld}n$	2	64	1808	10240	20971520
n^2	4	256	65536	1048576	$\approx 10^{12}$ *
n^3	8	4096	16777200	$\approx 10^9$ *	$\approx 10^{18}$ *
2^n	4	65536	$\approx 10^{77}$ *	$\approx 10^{308}$ *	$\approx 10^{315653}$ *

*: Der Wert liegt jenseits der Grenze der “praktischen Berechenbarkeit”.

Recht illustrativ ist es auch, sich vor Augen zu führen, welche Größenordnung ein Problem haben kann, wenn man die Zeit durch eine maximale Dauer T begrenzt. Wir nehmen dazu an, daß ein “Schritt” des Aufwands genau eine Mikrosekunde (1 μs) dauert. Sei G das größte lösbare Problem in der Zeit T. Den Aufwand $g(n) = \log n$ lassen wir weg, da man hier praktisch unbegrenzt große Probleme in “vernünftiger” Zeit bearbeiten kann.

G	T = 1 Min.	1 Std.	1 Tag	1 Woche	1 Jahr
$g(n) = n$	$6 \cdot 10^7$	$3,6 \cdot 10^9$	$8,6 \cdot 10^{10}$	$6 \cdot 10^{11}$	$3 \cdot 10^{13}$
n^2	7750	$6 \cdot 10^4$	$2,9 \cdot 10^5$	$7,8 \cdot 10^5$	$5,6 \cdot 10^6$
n^3	391	1530	4420	8450	31600
2^n	25	31	36	39	44

Zu den einzelnen Komplexitätsklassen listen wir einige typische Klassen von Problemen auf, die sich mit diesem Aufwand, jedoch nicht mit geringerem, lösen lassen.

Aufwand	Problemklasse
$O(1)$, konstant	einige Suchverfahren für Tabellen (“Hashing”)
$O(\log n)$, logarithmisch	allgemeinere Suchverfahren für Tabellen (Baum-Suchverfahren)
$O(n)$, linear	sequentielle Suche, Suche in Texten, syntaktische Analyse von Programmen (bei ”guter” Grammatik)
$O(n \cdot \log n)$	Sortieren
$O(n^2)$, quadratisch	einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume) Multiplikation Matrix-Vektor (einfach)
$O(n^3)$	Matrizen-Multiplikation (einfach)
$O(2^n)$, exponentiell	viele Optimierungsprobleme (z.B. optimale Schaltwerke) automatisches Beweisen (im Prädikatenkalkül 1. Stufe) etc.

Bemerkung 4.10 Es ist, streng genommen, ein offenes Problem, ob sich diese exponentiellen Probleme nicht doch mit polynomialem Aufwand lösen lassen. Es wäre jedoch eine große Überraschung, wenn sich dies herausstellen sollte. □

5. Entwurf von Algorithmen

- Schrittweise Verfeinerung
- Einsatz von Algorithmen-Mustern
  - Greedy-Algorithmen
  - Rekursive Algorithmen
  - und weitere

5.1. Schrittweise Verfeinerung

Prinzip: schrittweise Verfeinerung von Pseudo-Code-Algorithmen

- Ersetzen von Pseudo-Code-Teilen durch
  - verfeinerten Pseudo-Code
  - Programmiersprachen-Code

bereits vorgeführt!

5.2. Einsatz von Algorithmen-Mustern

Idee:

> Anpassung von generischen Algorithmenmustern für bestimmte Problemklassen an eine konkrete Aufgabe.

5.2.1. Greedy-Algorithmen am Beispiel

Auf Geldbeträge unter 1 DM soll Wechselgeld herausgegeben werden.

Zur Verfügung stehen ausreichend Münzen mit den Werten 50, 10, 5, 2, 1 Pfennig.

Das Wechselgeld soll aus so wenig Münzen wie möglich bestehen.

Also: 78 Pfennig = $50 + 2 * 10 + 5 + 2 + 1$.

Greedy-Algorithmus: Nehme jeweils immer die größte Münze unter Zielwert, und ziehe sie von diesem ab. Verfahre derart bis Zielwert gleich Null.

Greedy-Algorithmen berechnen lokales Optimum!

Beispiel: Münzen 11, 5, und 1; Zielwert 15

Greedy: $11 + 1 + 1 + 1 + 1$

Optimum: $5 + 5 + 5$

Aber: in vielen Fällen entsprechen lokale Optima den globalen, bzw. reicht ein lokales Optimum aus!

Greedy = 'gierig'!

Greedy-Algorithmen

1. Gegebene Menge von Eingabewerten.
2. Menge von Lösungen, die aus Eingabewerten aufgebaut sind.
3. Lösungen lassen sich schrittweise aus partiellen Lösungen, beginnend bei der leeren Lösung, durch Hinzunahme von Eingabewerten aufbauen.
4. Bewertungsfunktion für partielle und vollständige Lösungen.
5. Gesucht wird die / eine optimale Lösung.

Greedy-Beispiel: Kommunikationsnetz

Zwischen n Knotenpunkten $P_1, ..P_n$ soll ein *möglichst billiges Kommunikationsnetz* geschaltet werden, so daß jeder Knotenpunkt mit jedem anderen verbunden ist, ggf. auf einem Umweg über andere Knotenpunkte.

Bekannt sind Kosten $d_{i,j}$ für die direkte Verbindung zwischen P_i und P_j, $1 \leq i, j \leq n$.

Alle Kosten $d_{i,j}$ seien verschieden und größer als Null.

Beispiel: Eingabe für Kommunikationsnetz

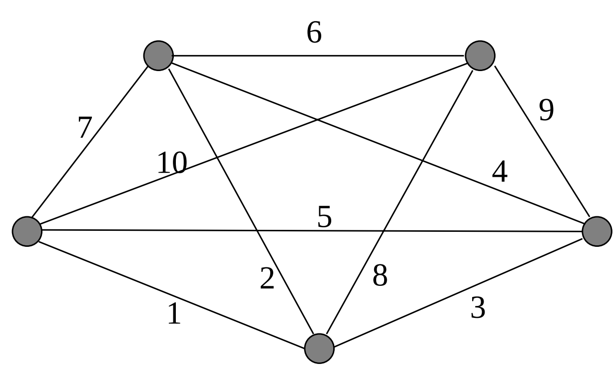


Abbildung 5.1.: Eingabe für Kommunikationsnetz

Beispiel: Eingabe für Kommunikationsnetzaufbau in Abbildung 5.1.

Aufspannender minimaler Baum

Erste Greedy-Variante:

```
[ Teilbaum R besteht anfangs aus beliebigem Knoten ]
while [ R noch nicht K_n aufspannt ]
do [ Suche billigste von R ausgehende Kante ];
   [ Füge diese zu R hinzu ]
od
```

Verfeinerung der Suche nach der billigsten Kante notwendig!

Ergebnis: Kommunikationsnetz

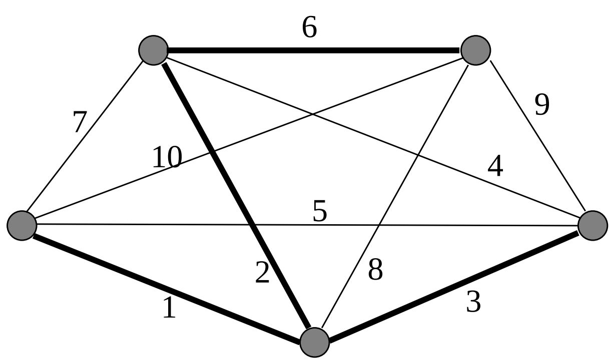


Abbildung 5.2.: Errechnetes Kommunikationsnetz

Beispiel: Errechnetes Kommunikationsnetz 5.2.

Suche nach billigster Kante

Intuitive Vorgehensweise erfordert jeweils $k(n-k)$ Vergleiche, also Gesamtlaufzeit $O(n^3)$!

Beschränkung der Suche auf Teilmenge V derart daß:

1. V enthält immer die billigste aus R ausgehende Kante.
2. V enthält wesentlich weniger Kanten als $k(n-k)$.
3. V ist einfach anpaßbar im Verlaufe des Algorithmus.

Wahl von V

A. V enthält für jeden Knoten P in R die billigste von P aus R herausführende Kante.

B. V enthält für jeden Knoten P außerhalb R die billigste von P in R hineinführende Kante.

Alternative A: mehrere Kanten können zum gleichen Knoten herausführen — redundant und änderungsaufwendig!

Daher: Wahl von Alternative B.

Erste Verfeinerung

```
[ R := ein beliebiger Knoten P ]
[ V := alle n−1 nach P führenden Kanten ]
for i:= 1 to n-1
do [ Suche billigste Kante b in V ];
   [ Füge b zu R hinzu ];
   [ Ändere V ]
od
```

"Ändere V"

b aus V entfernen [Anzahl Verbindungen ist okay].

Neu verbundener Knoten P:

Jeder noch nicht verbundener Knoten Q hat billigste Verbindung entweder wie zuvor, oder aber mit P!

Zweite Verfeinerung

```
[ R := ein beliebiger Knoten P ]
[ V := alle n−1 nach P führenden Kanten ]
for i:= 1 to n-1
do [ Suche billigste Kante b in V (Endknoten sei P) ];
   [ Füge b zu R hinzu ];
   [ Entferne b aus V ];
   for [ alle Kanten c in V mit Endknoten Q ] do
      if [ Kosten von Q-P ] < [ Kosten von c ]
      then [ Ersetze c durch Q-P ]
      fi
   od
od
```

Bemerkungen zum Beispiel

Für Realisierung benötigen wir 'abstrakten' Datentyp *Graph* mit Knoten, Kanten, Operationen etc. → zweites Semester.

Verfahren ist quadratisch im Aufwand: $O(n^2)$

Algorithmus stammt von R.C. Prim (1957)

5.3. Rekursive Algorithmen

Rekursive Algorithmen:

> Rekursives Anwenden einer Problemlösungsstrategie auf Teilproblem.

Verwandter Begriff: Selbstähnlichkeit.

5.3.1. Prinzip der Rekursion am Beispiel

Beispiel: Pythagorasbäume, Abbildung 5.3 zeigt Ausgabe eines Java-Programms (Programm über die WWW-Seite der Vorlesung herunterladbar).

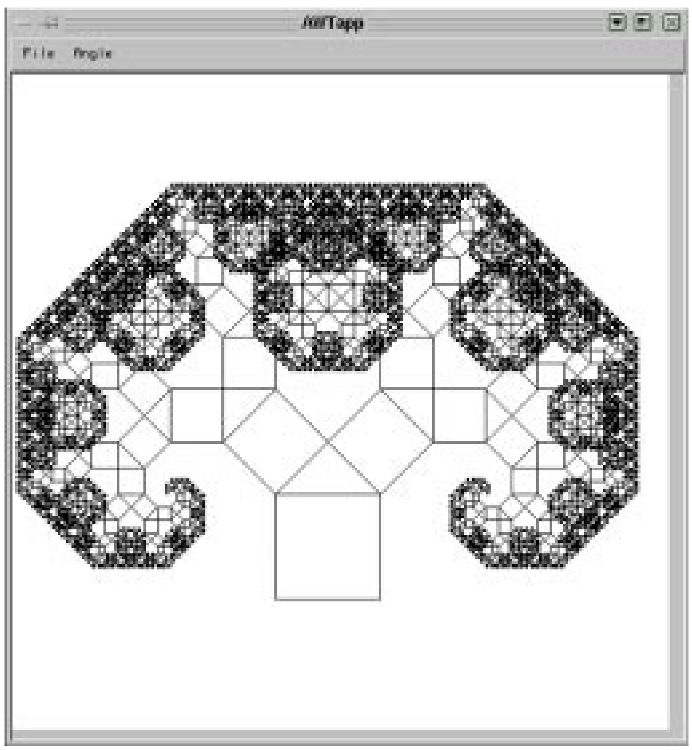


Abbildung 5.3.: Pythagorasbaum

Erläuterung Pythagorasbäume

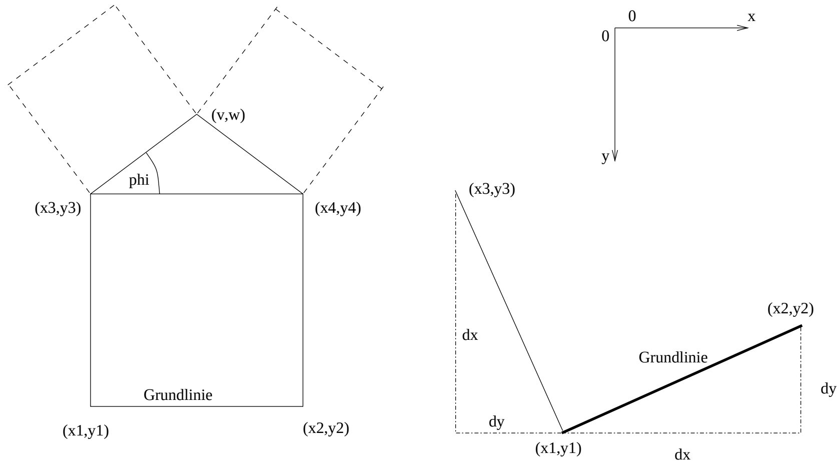


Abbildung 5.4.: Erläuterungen zu Pythagorasbäumen

Einige Erläuterung zu der folgenden Implementierung der Pythagorasbäume, insbesondere Variablenbezeichnungen, können in Abbildung 5.4 gefunden werden.

Pythagorasbäume: Java

Der folgende Abschnitt zeigt nur den Teil des Java-Progarmms, in dem ein Quadrat (festgelegt durch eine Grundlinie) berechnet und gezeichnet wird, und, solange die Kantenlänge bnicht zu klein ist, rekursiv weiter (kleinere) Teilbäume an den Schenkeln eines Dreiecks auf dem Quadrat gezeichnet werden.

```
public void paintTree (Graphics g,
                  double x1, double y1,
                  double x2, double y2 ) {
  double dx = x2 - x1;
  double dy = y1 - y2;
  double x3 = x1 - dy;
  double y3 = y1 - dx;
  double x4 = x2 - dy;
  double y4 = y2 - dx;

  g.drawLine((int)x1,(int)y1,(int)x2,(int)y2);
  g.drawLine((int)x2,(int)y2,(int)x4,(int)y4);
  g.drawLine((int)x4,(int)y4,(int)x3,(int)y3);
  g.drawLine((int)x1,(int)y1,(int)x3,(int)y3);

  double v = ((x3 + x4)/2 - (dy/2 * tanphi));
```

```
  double w = ((y3 + y4)/2 - (dx/2 * tanphi));

  if ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) >2 ) {
    paintTree (g,x3,y3,v,w);
    paintTree (g,v,w,x4,y4);
  }
}
```

5.3.2. Divide and Conquer

Quick-Sort und Merge-Sort als typische Vertreter

Teile und Herrsche; divide et impera

Prinzip:

> rekursive Rückführung auf identisches Problem mit kleinerer Eingabemenge

Divide-and-Conquer-Algorithmen

Grundidee: *Teile* das gebene Problem in mehrere getrennte Teilprobleme auf, *löse* dese *einzeln* und *setze* die Lösungen des ursprünglchen Problems aus den Teillösungen zusammen.

Wende dieselbe Technik auf jedes der Teilprobleme an, dann auf deren Teilprobleme usw., bis die Teilprobleme klein genug sind, daß man eine Lösung explizit angeben kann.

Trachte danach, daß jedes Teilproblem *von derselben Art* ist wie das ursprüngliche Problem, so daß es mit demselben Algorithmus gelöst werden kann.

Divide-and-Conquer-Muster

```
procedure DIVANDCONQ (P: problem)
begin
   ...
   if [ P klein ]
   then [ explizite Lösung ]
   else [ Teile P auf in P1, ...  Pk ];
        DIVANDCONQ ( P1 );
        ...;
        DIVANDCONQ ( Pk );
        [ Setze Lösung für P aus Lösungen
          für P1,.., Pk zusammen ]
   fi
end
```

Beispiel: Spielpläne für Turniere

$n = 2^k$ Spieler

$n - 1$ Turniertage
jeder Spieler spielt gegen jeden anderen
Annahme: Turnierplan T_k bekannt
Aufgabe: konstruiere T_{k+1} für $m = 2n = 2^{k+1}$
Spielplan als Matrix, Eintrag $s_{i,j}$ ist Gegner des Spielers i am Tage j

Spielplan T_2

	Tag 1	Tag 2	Tag 3
Spieler 1	2	3	4
Spieler 2	1	4	3
Spieler 3	4	1	2
Spieler 4	3	2	1

Idee des Algorithmus

Konstruiere T_{k+1} aus T_k wie folgt:

	$1 \cdots n-1$	$n \cdots m-1$
1 $\vdots$ $n = 2^k$	T_k	S_k
$n+1$ $\vdots$ $m = 2^{k+1}$	$T_k^{[+n]}$	Z_k

Dabei gilt:

- $T_k^{[+n]}$: T_k mit jeweils um n erhöhten Elementen
- Z_k : $(n \times n)$-Matrix, konstruiert durch zyklisches Verschieben der Zeile $(1, 2, ..., n)$
- S_k : $(n \times n)$-Matrix, konstruiert durch zyklisches Verschieben der Spalte $(n + 1, ..., m)$ für $n = 2^k$ und $m = 2^{k+1}$

Also z.B.:

$$Z_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{pmatrix} \qquad S_2 = \begin{pmatrix} 5 & 8 & 7 & 6 \\ 6 & 5 & 8 & 7 \\ 7 & 6 & 5 & 8 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

Spielplan T_1

	Tag 1
Spieler 1	2
Spieler 2	1

Spielplan T_3

	Tag 1	Tag 2	Tag 3	Tag 4	Tag 5	Tag 6	Tag 7
Spieler 1	2	3	4	5	8	7	6
Spieler 2	1	4	3	6	5	8	7
Spieler 3	4	1	2	7	6	5	8
Spieler 4	3	2	1	8	7	6	5
Spieler 5	6	7	8	1	2	3	4
Spieler 6	5	8	7	2	3	4	1
Spieler 7	8	5	6	3	4	1	2
Spieler 8	7	6	5	4	1	2	3

Weitere Beispiele für Divide & Conquer

- Merge-Sort
- Quick-Sort
- Türme von Hanoi (rekursive Lösung)

Merge-Sort: Algorithmus

(zur Erinnerung, aus Kapitel 3)

```
algorithm MergeSort (L: Liste):  Liste
if Liste einelementig
then return L
else
   teile L in L1 und L2;
   setze L1 = MergeSort (L1);
   setze L2 = MergeSort (L2);
   return Merge(L1,L2);
fi;
```

Backtracking

Backtracking (“Zurückverfolgen”): Allgemeine systematische Suchtechnik

- KF Menge von Konfigurationen K
- K_0 Anfangskonfiguration

- für jede Konfiguration K_i direkte Erweiterungen $K_{i,1}, K_{i,n_i}$
- für jede Konfiguration ist entscheidbar, ob sie eine Lösung ist

Aufruf: BACKTRACK(K_0)

Labyrinth-Suche

Wie findet die Maus den Käse?

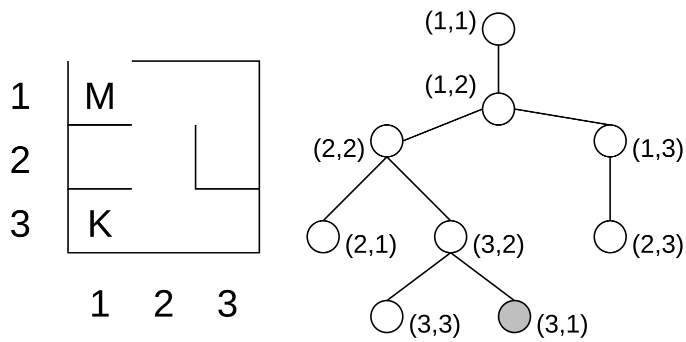


Abbildung 5.5.: Labyrinth-Suche

Abbildung 5.5 zeigt Labyrinth und zu durchlaufende Konfigurationen (= Positionen im Labyrinth) bei vollständigem Durchlauf durch das Labyrinth.

Backtracking-Muster

```
procedure BACKTRACK (K: konfiguration)
begin
   ...
   if [ K ist Lösung ]
   then [ gib K aus ]
   else
      for each [ jede direkte Erweiterung K' von K ]
         do
            BACKTRACK(K')
         od
   fi
end
```

Typische Einsatzfelder des Backtracking

- Spielprogramme (Schach, Dame, ...)
- Erfüllbarkeit von logischen Aussagen (logische Programmiersprachen)

- Planungsprobleme, Konfigurationen

Beispiel: Acht-Damen-Problem

Gesucht: Alle Konfigurationen von 8 Damen auf einem 8x8-Schachbrett, so daß keine Dame eine andere bedroht (Abbildung 5.6)

	1	2	3	4	5	6	7	8
1			D					
2						D		
3								D
4	D							
5				D				
6							D	
7					D			
8		D						

	1	2	3	4	5	6	7	8
1				D				
2						D		
3								D
4		D						
5							D	
6	D							
7			D					
8					D			

Abbildung 5.6.: Acht-Damen-Problem

Menge der Konfigurationen KF

Gesucht: geeignetes KF

1. $L \subseteq KF$ für Lösungskonfigurationen L
2. Für jedes $k \in KF$ ist leicht entscheidbar ob $k \in L$
3. Konfigurationen lassen sich schrittweise erweitern $\rightarrow$ hierarchische Struktur
4. KF sollte nicht zu groß sein

Menge der Lösungskonfigurationen L

L1 Es sind 8 Damen auf dem Brett.

L2 Keine zwei Damen bedrohen sich.

Geschickte Wahl von KF:

> Konfigurationen mit je einer Dame in den ersten n, $0 \leq n \leq 8$, Zeilen, so daß diese sich nicht bedrohen.

Nicht alle Konfigurationen lassen sich zu einer Lösung erweitern!

Abbildung 5.7 zeigt eine nicht eerweiterbare Konfiguration: ... jedes Feld in Zeile 7 ist bereits bedroht!

	1	2	3	4	5	6	7	8
1				D				
2	D							
3			D					
4					D			
5								
6								
7								
8								

Abbildung 5.7.: Nicht alle Konfigurationen lassen sich zu einer Lösung erweitern!

Acht-Damen-Problem: Algorithmus

```
procedure PLAZIERE (i :  [ 1..8 ] );
begin
   var h :  [ 1..8 ];
   for h:=1 to 8 do
      if [ Feld in Zeile i, Spalte h nicht bedroht ]
      then
         [ Setze Dame auf diese Feld (i,h) ];
         if [ Brett voll ] /* i = 8 */
         then [ Gib Konfiguration aus ]
         else PLAZIERE (i+1)
         fi
      fi
   od
end
```

Beispiel: Vier-Damen-Problem

Der Ablauf des Algorithmus bei der vereinfachten Version eines Vier-Damen-Problems wird in Abbildung 5.8 skizziert. Die zweite und die vierte Konfiguration der ersten Stufe werden hier nicht weiter verfeinert, da sie sich symmetrisch zu den beiden vollständig gezeigten Teilbäumen entwickeln.

Acht-Damen-Problem - Bemnerkungen

- Initialer Aufruf mit `PLAZIERE(1)`
- Es gibt 92 Lösungen ...

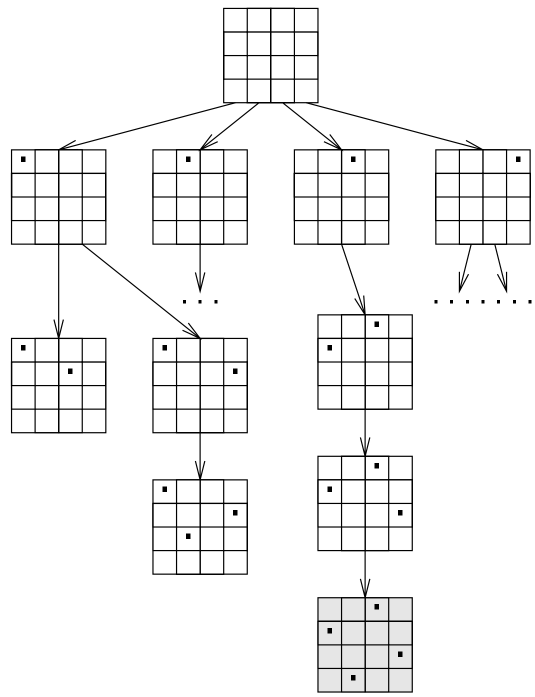


Abbildung 5.8.: Vier-Damen-Problem

- Konfigurationen etwa als zweidimensionales boolesches Arrary, oder als eindimensionales Array mit Damenposition pro Zeile realisierbar
- Optimierungspotential: redundante Informationen über bedrohte Spalten und Diagonalen

Varianten des Backtracking

- bewertete Lösungen; nach Lauf des Algorithmus wird beste ausgewählt
- Abbruch nach erster Lösung
- ‘Branch-and-Bound’: nur Zweige verfolgen, die eine Lösung prinzipiell zulassen
- maximale Rekursionstiefe (etwa bei Schachprogrammen!)

6. Verteilte Berechnungen

- Kommunizierende Prozesse als Ausführungsmodell
- Petri-Netze als Beschreibungsformalismus
- Programmierkonzepte

6.1. Kommunizierende Prozesse

Bisher: *Eine* Sequenz von Ausführungsschritten als Berechnung
Kommunizierende Prozesse:

- *jeder* Prozess eine Ausführungssequenz
- Schritte mehrerer Prozesse prinzipiell unabhängig:
  $\rightarrow$ zeitliche Reihenfolge oft nicht bestimmt!
- Synchronisation durch Kommunikation

Beispiele für kommunizierende Prozesse

- Kommunizierende Rechner
  - Web-Server und Browser
  - Client-Server-Anwendungen
- Produzenten-Verbraucher-Systeme (Logistik-Anlagen)
- Agenten-Systeme (Internet, Simulationen, Spiele)
- Betriebssysteme / Benutzeroberflächen

6.2. Modell der Petri-Netze

1962 von C. A. Petri vorgeschlagen
Modell zur Beschreibung von Abläufen mit *nebenläufigen* und *nichtdeterministischen Vorgängen*

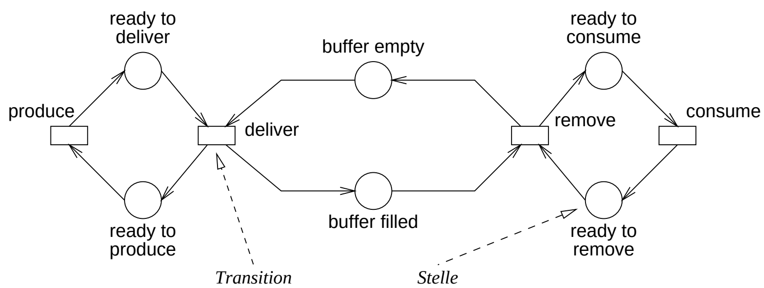
Petri-Netze

Petri-Netz ist gerichteter Graph

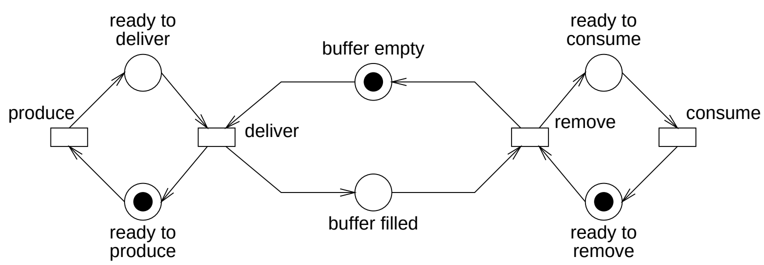
- Stellen: Zwischenlager von Daten → Kreise
- Transitionen: Verarbeitung von Daten → Balken
- (gerichtete) Ein- und Ausgabe-Kanten von Transitionen
  Kanten verbinden immer Transitionen mit Stellen!
- Markierungen der Stellen

Petri-Netz: Graphische Notation

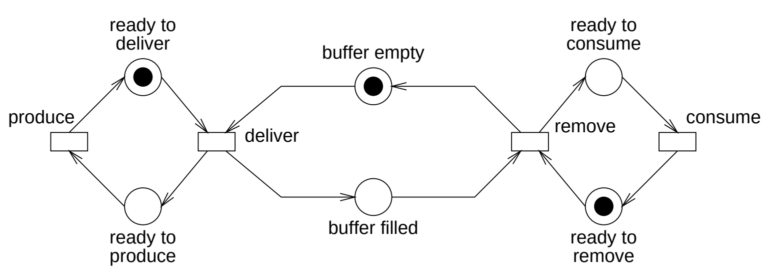
Beispiel-Netz (Produzent-Verbraucher mit Puffer der Größe 1):



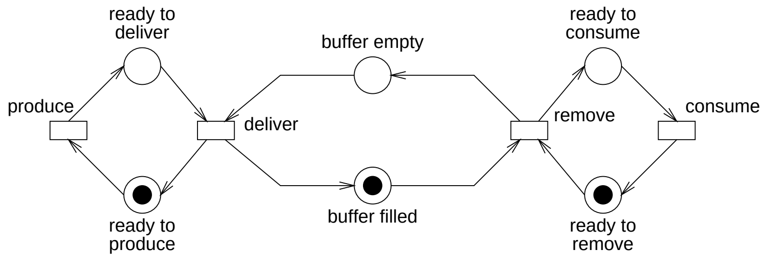
Petri-Netz mit Marken



Petri-Netz nach Feuern von produce



Petri-Netz nach Feuern von `deliver`



Petri-Netze

Arten von Petri-Netzen

- Bedingungs-Ereignis-Netz

  Stellen sind boolesche Variablen (bisheriges Beispiel)

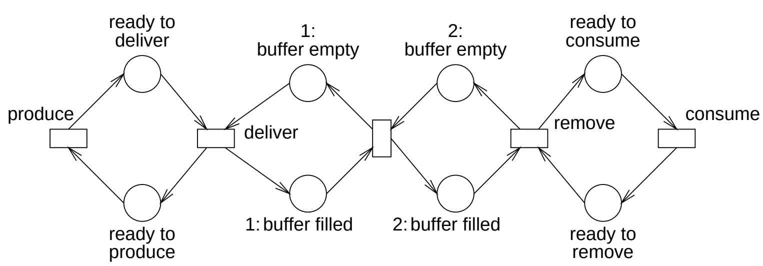
- Stellen-Transitions-Netz

  Stellen nehmen Integer-Werte an, eventuell mit ‘Kapazitäten’

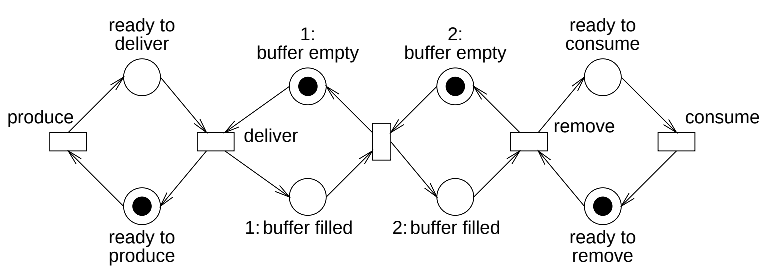
- höhere Petri-Netze

  Stellen sind Container für (strukturierte) Werte

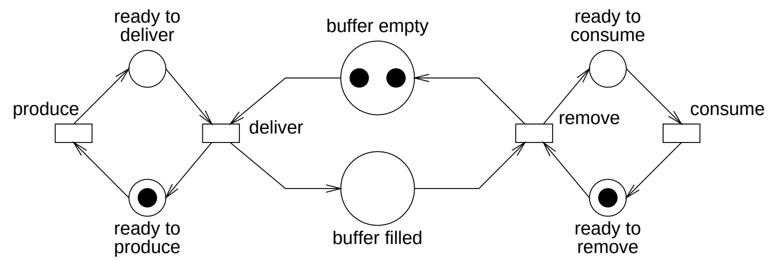
Petri-Netz mit Puffergröße 2



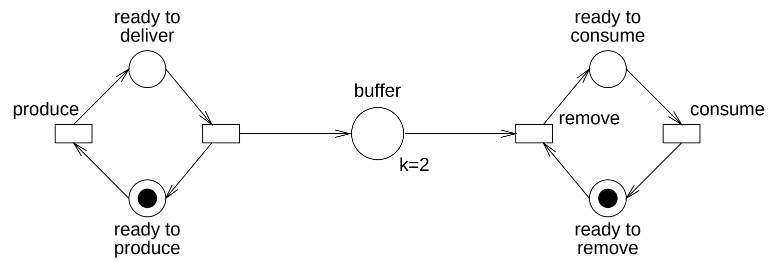
Petri-Netz mit Puffergröße 2 mit Marken



Stellen-Transitions-Netz für Puffergröße 2



Stellen-Transitions-Netz mit Kapazität



Schaltregel

“Token-Game”:

1. Eine Transition t kann schalten / feuern, wenn jede Eingabestelle von t mindestens eine Marke enthält.
2. Schaltet eine Transition, dann wird aus jeder Eingabestelle eine Marke entfernt und zu jeder Ausgabe stelle eine Marke hinzugefügt.

   Bei Stellen mit Kapazitäten darf das Hinzufügen nicht die Kapazitätsbegrenzung verletzen!

Erweiterung um gewichtete Kanten in naheliegender Weise...

Fragestellungen

- Terminierung: Hält das Netz ausgehend von eine Startmarkierung nach endlich vielen Schritten? Für beliebige Startmarkierung?
- Lebendigkeit: Kann man die Transitionen jeweils so schalten, daß eine Transition nochmals schalten kann? Lebendigkeit einer bestimmten Transition; Existenz von toten Transitionen?
- Verklemmungen / Deadlocks?
- Erreichbarkeit einer bestimmten Markierung?
- Beschränktheit: Obergrenze für Anzahl Marken im Netz?

Petri-Netz formal

$$P = (S, T, A, E, M)$$

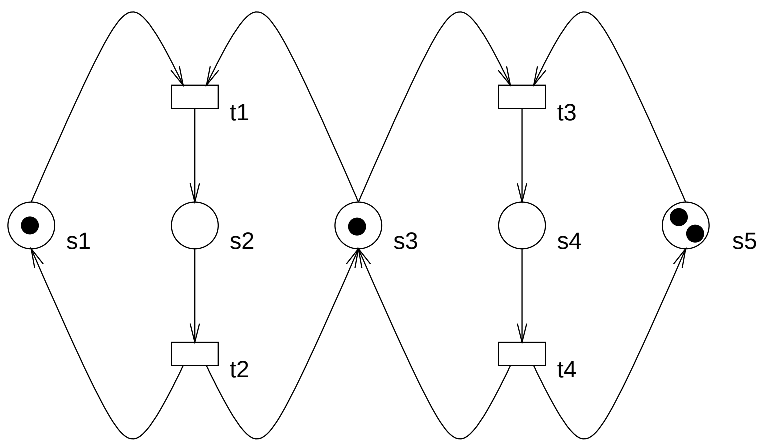
- S ist nichtleere Menge von **S**tellen
- T ist nichtleere Menge von **T**ransitionen mit $S \cap T = \emptyset$
- $A \subset S \times T$ **A**usgangskanten von Stellen; Eingangskanten von Transitionen
- $E \subset T \times S$ **E**ingangskanten von Stellen; Ausgangskanten von Transitionen
- $m : S \to \mathbb{N}_0$ Start**m**arkierung

Oft: eine **F**lußrelation $F = A \cup E$ statt A und E separat

Petri-Netz formal: Beispiel

$P = (S, T, A, E, M)$
$S = \{s_1, s_2, s_3, s_4, s_5\}$
$T = \{t_1, t_2, t_3, t_4\}$
$A = \{(s_1, t_1), (s_2, t_2), (s_3, t_1), (s_3, t_3), (s_4, t_4), (s_5, t_3)\}$
$E = \{(t_1, s_2), (t_2, s_1), (t_2, s_3), (t_3, s_4), (t_4, s_3), (t_4, s_5)\}$
$M(s_1) = M(s_3) = 1, \quad M(s_5) = 2, \quad M(s_2) = M(s_4) = 0$
$\to$ gegenseitiger Ausschluß zweier Prozesse

Petri-Netz formal: Beispiel graphisch



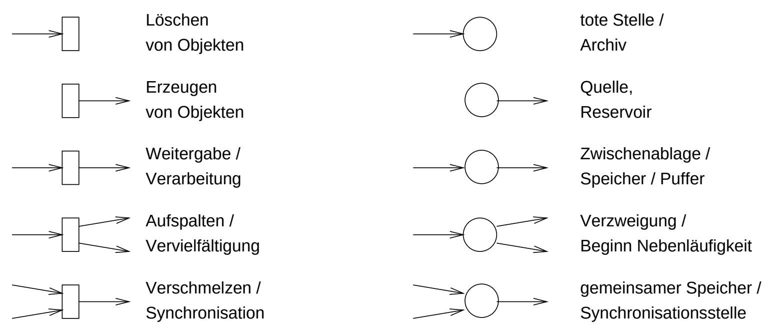
Formalisierung der Schalt-Regel

$\bullet t = \{s \in S | (s, t) \in A\}$
$t \bullet = \{s \in S | (t, s) \in E\}$

Schalt-Regel

$$M'(s) = \begin{cases} M(s)+1 & \text{falls } s \in t\bullet\text{, aber } s \notin \bullet t \\ M(s)-1 & \text{falls } s \in \bullet t\text{, aber } s \notin t\bullet \\ M(s) & \text{sonst} \end{cases}$$

Modellierungsprimitive für Petri-Netze



Fünf Philosophen

Fünf Philosophen am runden Tisch

Eine Schüssel Spaghetti

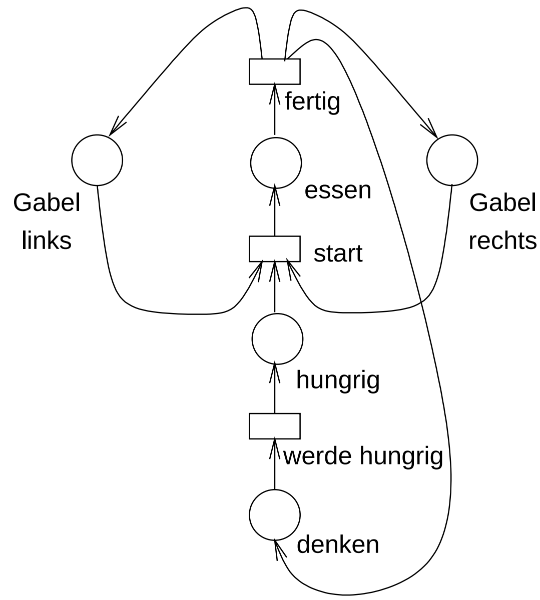
Fünf Gabeln, je eine zwischen zwei Philosphen

Essen nur mit zwei Gabeln möglich

Philosphen: denken $\rightarrow$ hungrig $\rightarrow$ essen $\rightarrow$ denken ...

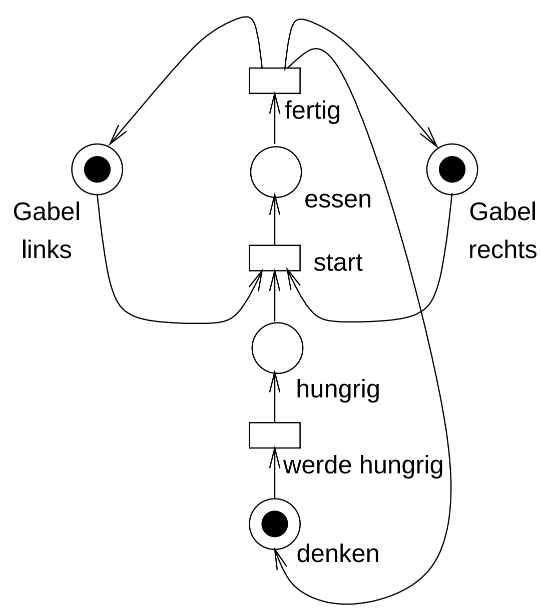
Philosophenproblem als Petri-Netz

Ein Philosoph:

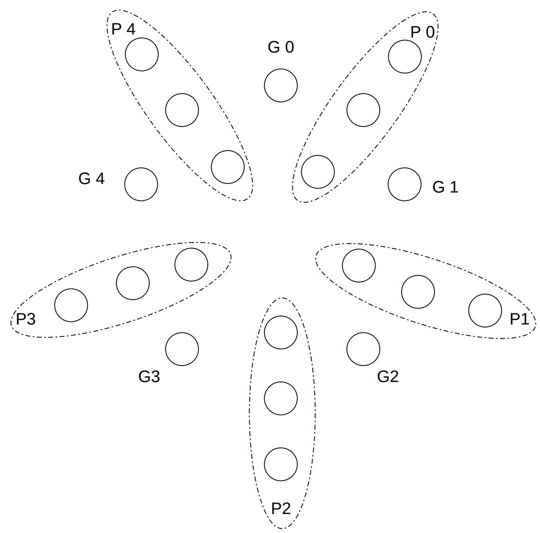


Philosophenproblem als Petri-Netz II

Ein Philosoph im initialen Zustand:



Stellen aller 5 Philosophen



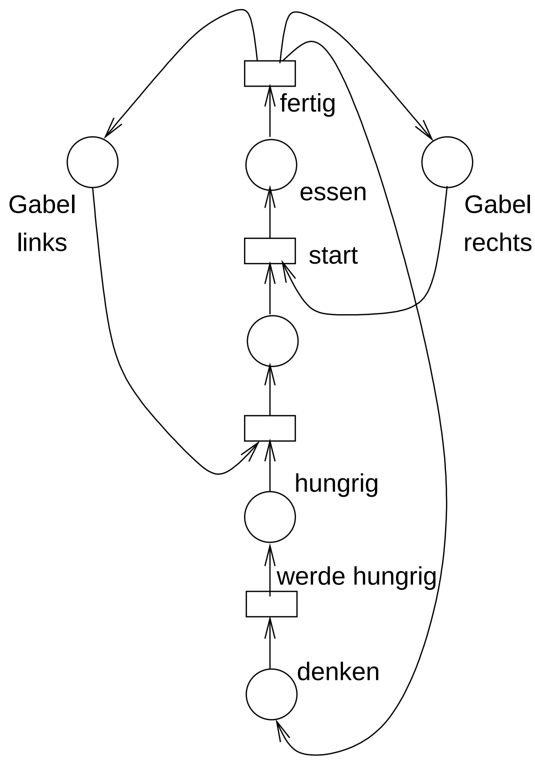
Eigenschaften der Philosophenmodellierung

Bisher: Abstrakte Modellierung des Problems

- beide Gabeln werden gleichzeitig aufgenommen

Realisierung der Gleichzeitigkeit nicht möglich, also Gabeln nacheinander aufnehmen → Verklemmungsmöglichkeit!

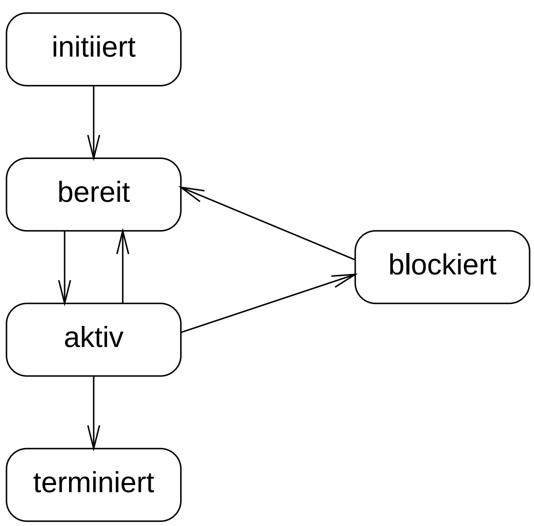
Fünf Philosophen mit Verklemmungsmöglichkeit



6.3. Programmieren von nebenläufigen Abläufen

nebenläufige Prozesse: mehrere *kommunizierende* Automaten *koordiniert* z.B. über Betriebssystem

Zustände von Prozessen



Programmieren mit Semaphoren

Semaphor (nach Dijkstra): ganzzahlige Variable plus *Warteschlange* von Prozessen
Operationen:

- **down**; Betreten; auch **P** für Passeer (holländisch)

  Warteoperation; Beginn eines kritischen Abschnitts

- **up**; Verlassen; auch **V** für Verlaat (holländisch)

  Signaloperation; Verlassen eines kritischen Abschnitts

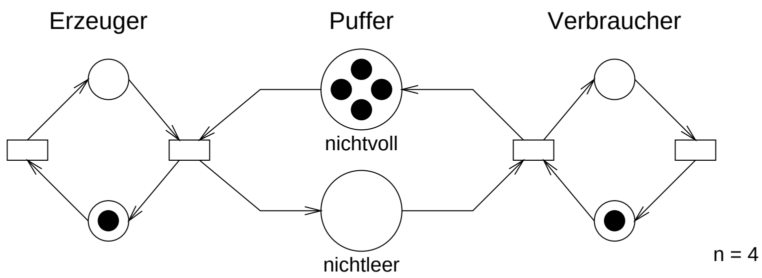
Semaphoren

```
type semaphor = 0..maxint;
procedure down (var s:  semaphor);
begin
   if s ≥ 1
   then s := s - 1
   else
      ''Stoppe den ausführenden Prozeß'';
      ''Trage den Prozeß in die
      Warteschlange W(s) ein''
   fi
end;
procedure up (var s:  semaphor);
begin
   s := s + 1;
   if ''Warteschlange W(s) nicht leer''
   then
      ''Wähle Prozeß Q aus W(s) aus'';
      ''Springe zur down-Operation in Q,
      durch die Q gestoppt wurde''
   fi
end;
```

Erzeuger-Verbraucher-System mit begrenztem Puffer mit Semaphoren

```
var nichtvoll, (* Puffer ist nicht voll *)
   nichtleer, (* Puffer ist nicht voll *)
   gesperrt (* Puffer ist nicht voll *)
      :  semaphor;
   nichtvoll = n; (* Puffer mit n Plätzen *)
   nichtleer := 0;
   gesperrt := 1;
```

Erzeuger-Verbraucher-System als Petri-Netz



Erzeuger

```
repeat
   ‘‘Erzeuge Marke’’;
   down (nichtvoll);
   down (gesperrt);
   ‘‘Transportiere Marke in Puffer‘‘;;
   up (gesperrt);
   up (nichtleer);
until false;
```

Verbraucher

```
repeat
   down (nichtleer);
   down (gesperrt);
   ‘‘Entnehme Marke dem Puffer‘‘;;
   up (gesperrt);
   up (nichtvoll);
   ‘‘Verbrauche Marke’’;
until false;
```

Philosophenproblem mit Semaphoren

- Gabeln als Semaphoren initialisiert mit 1

  ... Verklemmungsmöglichkeit falls alle Philosophen dem selben Programm folgen!

- Eine Semaphore für den Spaghettitopf ....

  ..wenig Nebenläufigkeit wenn lange gegessen wird.

- Eine (globale) Semaphore für Überprüfung ob beide Nachbargabeln frei sind, schützt auch die Aufnahme der Gabeln aber *nicht* die gesamte Eßphase ...

  ... mehr Nebenläufigkeit, keine Verklemmungen.

Philosoph i

```
repeat
   denken;
   hungrig werden;
   nimm Gabel i;
   nimm Gabel (i+1) mod 5;
   essen;
   lege Gabel i zurück;
   lege Gabel (i+1) mod 5 zurück
until false.
```

Philosoph i mit Semaphoren

```
repeat
   denken;
   hungrig werden;
   down (g_i);
   down (g_((i+1)%5));
   essen;
   up (g_i);
   up (g_((i+1)%5))
until false.
```

Verklemmungen möglich!

Verklemmungsfreie Philosophen: Idee

N Philosophen, eine Semaphore `ausschluss` zum Synchronisieren der Gabelaufnahme

Prozedur `prüfe` testet ob beide Nachbarn essen; falls nicht aktiviert Philosoph für Essensphase

je Philosoph eine Semaphore `phil` umd die Blockade und das Aktivieren zu realsieren

Aktivierung: Philosoph aktiviert (wenn möglich) beide Nachbarn sobald er fertig gegessen hat

Verklemmungsfreie Philosophen: Varablen

```
int zustand [N];
   (* denkend, hungrig, essend *)
semaphor ausschluss = 1;
   (* Synchronisation aller Philosophen *)
semaphor phil [N] = 0;
   (* Blockade eines Philosophen *)
```

Verklemmungsfreie Philosophen I

```
void philosoph (int i)
{
   while (true) {
      denke();
      nimmGabeln (i);
      esse();
      legeGabeln (i);
   }
}

void nimmGabeln (int i)
{
   down ( ausschluss );
   zustand[i] = hungrig;
   prüfe(i);
   up ( ausschluss );
   down ( phil[i] );
      (* Blockiere falls Gabeln nicht frei *)
}

void legeGabeln (int i)
{
   down ( ausschluss );
   zustand[i] = denkend;
   prüfe( (i-1) % N ); (* aktiviere ggf.  Nachbarn *)
   prüfe( (i+1) % N ); (* aktiviere ggf.  Nachbarn *)
   up ( ausschluss );
}

void prüfe (int i)
{
   if ( zustand[i] == hungrig &&
         zustand[i-1 % 5] != essend &&
         zustand[i+1 % 5] != essend )
   {
      zustand[i] = essend;
      up ( phil[i] );
         (* aktiviert bei Blockade!  *)
   }
}
```

Realisierung in Java

- in Java: geschützte Objekte (entsprechen geschützten Bereichen)

- Lösung: Schützen des Besteckkasten durch **synchronized**-Anweisung
  Diese implementiert quasi einen Monitor analog einer Semaphore.
- kritischer Bereich
  → alle Zugriffe synchronisiert mittels Semaphore mit Initialwert 1

Der Besteckkasten

- Verwaltung der 5 Gabeln
- Operationen zum Aufnehmen, Ablegen und Testen

```
class Canteen {
  // 5 Gabeln: true - frei, false - belegt
  boolean forks[] = {
    true, true, true, true, true
  };

  // Testet, ob Gabel verfügbar ist.
  boolean isAvailable (int f) {
    return forks[f];
  }

  // Aufnehmen der Gabel.
  void takeFork (int f) {
    forks[f] = false;
  }

  // Ablegen der Gabel.
  void putFork (int f) {
    forks[f] = true;
  }
}
```

Ein Philosoph

```
class Philosopher extends Thread {
  // Konstanten für die möglichen Zustände
  public final static int THINKING = 1;
  public final static int HUNGRY   = 2;
  public final static int EATING   = 3;

  Canteen canteen; // der Besteckkasten
  int leftFork, rightFork;
```

```
                // die linke und rechte Gabel
  int id; // die eigene ID

  /**
   * Konstruktor: initialisiert alle
   * Attribute und startet den Thread
   */
  public Philosopher (int id, int left,
                int right, Canteen can) {
    this.id = id;
    this.leftFork = left;
    this.rightFork = right;
    this.table = tbl;
    this.canteen = can;
    start ();
  }
  ...
}
```

Ablauf eines Philosophen

```
public void run () {
  // Anfangszustand: THINKING
  while (true) {
    // der Besteckkasten ist geschützt !
    synchronized (canteen) {
      // Warten, bis beide Gabeln verfügbar sind
      while (! canteen.isAvailable (leftFork) ||
            !canteen.isAvailable (rightFork)) {
        // Gabeln sind belegt: Zustand ist
        // HUNGRY - wir müssen warten
        try {
          canteen.wait ();
        } catch (InterruptedException exc) { }
      }
      // beide Gabeln aufnehmen
      canteen.takeFork (leftFork);
      canteen.takeFork (rightFork);
    }

    // jetzt können wir eine Weile essen
    // Zustand: EATING
    try {
      sleep ((int) (Math.random () * 3000.0));
```

```
    } catch (InterruptedException exc) { }

    // Besteckkasten ist wieder geschützt
    synchronized (canteen) {
      // Gabeln niederlegen
      canteen.putFork (leftFork);
      canteen.putFork (rightFork);
      // alle wartenden Philosophen aufwecken
      canteen.notifyAll ();
    }

    // wieder eine Weile nachdenken: THINKING
    try {
      sleep ((int) (Math.random () * 5000.0));
    } catch (InterruptedException exc) { }
  }
}
```

Starten der Philosophen

```
/**
 * Start Applet: Philosophen-Threads erzeugen
 */
public void start ()
  if (philosophers == null)
    // 5 Philosophen erzeugen und ihnen ihre
    // Gabeln zuweisen
    philosophers = new Philosopher[5];
    for (int i = 0; i < 5; i++)
     philosophers[i] =
       new Philosopher(i,i,(i+1 < 5 ? i+1 : 0),
                       table, canteen);



/**
 * Stop des Applets: alle Threads anhalten
 */
public void stop ()
  if (philosophers != null)
    for (int i = 0; i < 5; i++)
      philosophers[i].stop ();
  philosophers = null;
```

Philosophen: Erläuterung

- jeder Philosoph als *Thread*
  - einzelner, unabhängiger Kontrollfluß
  - leichtgewichtig: läuft im Kontext eines Betriebssystemprozesses
  - in Java: mehrere parallele Threads möglich
  - Hauptschleife eines Threads: Methode `run`
  - Starten mit `start`, Anhalten mit `stop`
- parallele Ausführung von 5 Threads
- Synchronisation über Monitore
  - Schutz eines kritischen Bereiches vor konkurrierenden Zugriffen

    ```
    synchronized (zu sperrendes Objekt) {
      Anweisungen
    }
    ```
  - Sperre beim Eintritt: gesicherte Ausführung der Anweisungen
  - zu schützender Bereich: Besteckkasten
- Warten über Methode `wait` bis Aufwecken mit `notifyAll`

Teil II.

Datenstrukturen

7. Abstrakte Datentypen

Abstrakte Datentypen wurden bereits im ersten Semester kurz eingeführt. Hier soll dieser Stoff rekapituliert und vertieft werden.

Um Programme möglichst wiederverwendbar zu gestallten und von unnötigen Details zu abstrahieren, spezifizieren wir Datenstrukturen unabhängig von ihrer späteren Implementierung in einer konkreten Programmiersprache. Die Beschreibung ist so gestaltet, daß Kunden sich der Datenstrukturen bedienen können, um Probleme zu lösen, daß aber die Implementierung jederzeit geändert werden kann, ohne daß die Kunden etwas davon merken (→ Geheimnisprinzip, programming by contract).

Ziel kurzgefaßt:

> Beschreibung von Datenstrukturen unabhängig von ihrer späteren Implementierung in einer konkreten Programmiersprache

Mittel hierzu sind *ADT*, *Abstrakte Datentypen*.
Neu entwickelte Datentypen:

- *konkrete Datentypen*: konstruiert aus Basisdatentypen bzw. Java-Klassen
- *abstrakte Datentypen*: Spezifikation der Schnittstelle nach außen: Operationen und ihre Funktionalität

Prinzipien von ADTen

- ADT = Software-Modul
- **Kapselung**: darf nur über Schnittstelle benutzt werden
- **Geheimnisprinzip**: die interne Realisierung ist verborgen

... somit sind ADTen Grundlage des Prinzips der objektorientierten Programmierung!

7.1. Spezifikation von ADTen

Gerade für objektorientierte Spezifikation ist die Definition von abstrakten Datentypen eine geeignete Methode, da ADT in natürlicher Weise durch Klassen implementiert werden können.

Die Spezifikation eines abstrakten Datentyps kann hierbei aus vier Teilen bestehen (angelehnt an Bertrand Meyer: Object oriented software construction, 2.Aufl., Kap.6):

1. **Type**: Hier wird der Name des neuen Typs genannt, zusammen mit eventuellen Parametern.

2. **Functions**: Hier werden die *Signaturen* der auf dem abstrakten Datentyp definierten Funktionen angegeben, das heißt für jede Funktion der Name, Typen und Reihenfolge der Parameter, Ausgabetyp sowie ggf. die Information, ob eine Funktion eine totale ($\rightarrow$) oder partielle ($\not\rightarrow$) Funktion ist.

3. **Axioms**: beschreiben *Eigenschaften* der eingeführten Funktionen als prädikatenlogische Formeln (zumeist Gleichungen). Jede Implementierung eines abstrakten Datentyps muß die Einhaltung der Axiome garantieren.

4. **Preconditions**: Für jede partielle Funktion muß eine Vorbedingung angegeben werden, die sagt, wann diese Funktion definiert ist.

Die ersten beiden Punkte legen die *Signatur* des ADT fest; die beiden anderen Teile die *Bedeutung*.

Wir betrachten jetzt einige Beispiele.

Ein einfacher, nicht rekursiver, nicht generischer Datentyp ist der Typ `POINT`.

```
Type POINT
Functions
   create:  REAL × REAL → POINT                  Konstruktor
   get_x:  POINT → REAL                          Selektor
   get_y:  POINT → REAL                          Selektor
   is_origin:  POINT → BOOL                      Prädikat
   translate:  POINT × REAL × REAL → POINT
   scale:  POINT × REAL → POINT
   distance:  POINT × POINT → REAL
Axioms ∀ x,y,a,b,z,w:  REAL
   get_x(create(x,y))=x
   get_y(create(x,y))=y
   is_origin(create(x,y))⇔ x = 0 ∨ y = 0
   translate(create(x,y),a,b)=create(x+a,y+b)
   scale(create(x,y),a)=create(x· a,y· a)
   distance(create(x,y),create(z,w))=√((x − z)² + (y − w)²)
Preconditions
   keine
```

Die Funktionen, die im **`Functions`**-Teil der ADT-Spezifikation eingeführt werden, können verschiedene Rollen spielen.

Konstruktorfunktionen sind Funktionen, die dazu benutzt werden, Elemente des ADT *aufzubauen*. Jedes Element eines spezifizierten ADT kann unter ausschließlicher Verwendung von Konstruktorfunktionen aufgebaut werden. Konstruktorfunktionen erkennt man oft daran, daß es keine Axiome gibt, die sie einschränken. Ausnahmen gibt es bei sogenannnten *nicht-freien*[1] *Datentypen* (z.B. Mengen); dort sind auch Konstruktoren mit Axiomen möglich. Alle anderen Funktionen werden hingegen im allgemeinen unter Bezug auf die Konstruktorfunktionen beschrieben.

Beim ADT `POINT` ist `create` die Konstruktorfunktion. Jeder Punkt wird mit `create` gebildet, und alle anderen Funktionen sind unter Bezugnahme auf `create` beschrieben. *Selektorfunktionen* sind sozusagen die Inversen der Konstruktorfunktionen. Sie zerlegen ein Element des Datentyps in seine „Einzelteile". Für `POINT` sind dies `get_x` und `get_y`. *Prädikate* haben als Wertebereich den Typ `BOOL`. Für `POINT` ist dies `is_origin`.

Alle anderen Funktionen bezeichnen wir als sonstige Funktionen. Für `POINT` sind dies `translate`, `scale` und `distance`.

Ein weiterer wichtiger Datentyp sind die natürlichen Zahlen.

```
Type NAT
Functions
   zero:  NAT                        Konstruktor
   succ:  NAT → NAT                  Konstruktor
   pred:  NAT ↛ NAT                  Selektor
   less:  NAT × NAT → BOOL           Prädikat
   add:  NAT × NAT → NAT
   mult:  NAT × NAT → NAT
Axioms ∀ i,j:  NAT
   pred (succ(i))=i
   less(zero,succ(i))=true
   less(j,zero)=false
   less(succ(i),succ(j))=less(i,j)
   add(zero,j)=j
   add(succ(i),j)=succ(add(i,j))
   mult(zero,j)=zero
   mult(succ(i),j)=add(j,mult(i,j))
Preconditions ∀ i:  NAT
   pre(pred(i)):  less(zero,i)
```

Die Spezifikation von `NAT` zeigt einen wichtigen Spezialfall der ADT-Spezifikation: Die Axiome sind ausschließlich Gleichungen über aus den neu definierten Funktionen zusammengesetzten Termen. Diese Art der Spezifikation (die Gleichungsspezifikation) wird oft auch mit der *algebraische Spezifikation* gleichgesetzt, obwohl in den diversen Methoden der algebraischen Spezifikation weitere Axiomtypen (Ungleichungen, bedingte Gleichungen) verwendet werden.

[1] Das sind solche, wo die Konstruktorfunktionen nicht injektiv sind.

7.2. Signaturen und Algebren

Einige Begriffe:

- Signatur: formale Schnittstelle
- Algebra: Modell für Spezifikation + Signatur

  Für eine gegebene Signatur gibt es viele Algebren als mögliche Modelle!
- Axiome: Formeln die bestimmte Modelle ausschließen
- ADT-Spezifikation: Signatur plus Axiome

  Für eine gegebene Spezifikation gibt es in der Regel ebenfalls mehrere (noch zu viele) Algebren als mögliche Modelle!
- Auswahl einer Algebra als *Standard-Modell* notwendig!

Im folgenden wird dies anhand der Gleichungsspezifikation (Spezialfall der algebraische Spezifikation) kurz erläutert (nicht vertieft — würde Vorlesung sprengen!)

Signatur mathematisch

$$\Sigma = (S, \Omega)$$

- S Sorten
- $\Omega = \{f_{S^*,S}\}$ Funktionssymbole für Funktionen $f\colon s_1 \cdots s_n \rightarrow s$ mit Parametersorten $s_1, ..., s_n$ und Ergebnissorte s

  Funktionen ohne Parameter heißen *Konstanten*

Algebra mathematisch

$$A_\Sigma = (A_S, A_\Omega)$$

- A_S Trägermengen der Sorten
- $A_\Omega = \{A_f\colon A_{s_1} \cdots A_{s_n} \rightarrow A_s\}$ Funktionen auf den Trägermengen

7.3. Algebraische Spezifikation

Gleichungsspezifikation:

- Angabe der Signatur: Typen, Funktionssignaturen
- Gleichungen
- evtl. Import anderer Spezifikationen

7.3.1. Spezifikationen und Modelle

Beispiel Wahrheitswerte:

```
type bool
functions
   true:  → bool,
   false:  → bool
```

Modelle für diese Spezifikation?

1. $A_{\texttt{bool}} = \{T, F\}; A_{\texttt{true}} := T; A_{\texttt{false}} := F$

   ... gewünschte Algebra....

2. $A_{\texttt{bool}} = \mathbb{N}; A_{\texttt{true}} := 1; A_{\texttt{false}} := 0$

   ? eventuell akzeptabel, aber Trägermenge zu groß

3. $A_{\texttt{bool}} = \{1\}; A_{\texttt{true}} := 1; A_{\texttt{false}} := 1$

   ? sicher nicht erwünscht!

Ein etwas komplexeres Beispiel mit 'echten' Funktionen (Zähler der hoch- und herunterzählen[2] kann):

```
type counter
functions
   zero → counter,
   inc:  counter → counter,
   dec:  counter → counter
```

Modelle für diese Spezifikation?

1. $A_{\texttt{counter}} = \mathbb{N}$

   ```
   reset := 0;
   inc(n) := n+1;
   dec(0) := 0;
   dec(n+1) := n;
   ```

   Realisierung mit natürlichen Zahlen

2. $A_{\texttt{counter}} = Z$

   ```
   reset := 0;
   inc(n) := n+1;
   dec(n) := n-1;
   ```

   gewünschtes Modell mit ganzen Zahlen!

[2] `increment`: hochzählen, `decrement`: herunterzählen

3. $A_{\texttt{counter}} = Z$

```
reset := 0;
inc(n) := n-1;
dec(n) := n+1;
```

? mathematisch äqivalent zur vorherigen Version!

4. $A_{\texttt{counter}} = \{0, 1, \ldots, p\}$

```
reset := 0;
inc(n) :=n+1 für n<p;
         0 für n=p;
dec(n) :=n-1 für n>0;
         p für n=0;
```

in Programmen implementierter Zähler mit Obergrenze — aber nicht aus Axiomen herleitbar!

Fragestellung: wie komme ich zu einem 'kanonischen' Standardmodell (also genau einer Algebra, die die Axiome respektiert) für eine gegebene Spezifikation?

7.3.2. Termalgebra und Quotiententermalgebra

Ziel: Konstruktion *einer* ausgewählten Algebra für eine Spezifikation.
Beispiel: nat-Algebra

```
type Nat
operators
   0:  -> Nat
   Suc:  Nat -> Nat
   Add:  Nat × Nat -> Nat
axioms
   Add( i, 0 ) = i
   Add( i, Suc(j) ) = Suc( Add( i, j ) )
```

Termalgebra

Termalgebra definiert durch

- Trägermenge: alle korrekt gebildeten Terme einer Signatur

  Nicht interpretiert!

- Funktionsanwendung: Konstruktion des zugehörigen Terms

Quotienten-Termalgebra

QTA: Äquivalenzklassen gemäß '='

- Start: Termalgebra
- können zwei Terme durch die Datentyp-Gleichungen gleichgesetzt werden, kommen Sie in die selbe Äquivalenzklasse
- die Äquivalenzklassen bilden die Werte der Trägermengen der Sorten
- Konstruktorfunktionen konstruieren jeweils genau einen Repräsentanten pro Äquivalenzklasse

```
0 repräsentiert { 0, Add(0,0),
   Add(Add(0,0),0), ...  }
Suc(0) repräsentiert { Suc(0),
   Add(0,Suc(0)),
   Add(Add(0,Suc(0)),0), ...  }
Suc(Suc(0)) ...
```

Bemerkung: Initialität der QTA (Prinzip der maximalen (erlaubten) Ungleichheit).

Folge: Jede die Gleichungen erfüllbare Algebra ist Bild einer totalen Abbildung der QTA (Morphismus, d.h. Gültigkeit der Funktionsanwendungen bleibt erhalten)

Terminalität als duales Prinzip (notwendig z.B. für Spezifikation mit `true` $\neq$ `false`!).

7.3.3. Probleme mit initialer Semantik

Das Mengen-Beispiel.

```
type Set[Item]
operators
   Create:  -> Set
   Isempty:  Set -> Boolean
   Insert:  Set × Item -> Set
   Isin:  Set × Item -> Boolean
axioms
   Isempty(Create) = true
   Isempty(Insert(s,i)) = false
   Isin(Create,i) = false
   Isin(Insert(s,i),j) =
      if i=j then true else Isin(s,j)
```

Folgendes gilt aufgrund der Mengensemantik:

```
Insert(Insert(s,i),j) = Insert(Insert(s,j),i)
Insert(Insert(s,i),i) = Insert(s,i)
```

Kann aber nicht mit den Gleichungen bewiesen werden → unterschiedliche Elemente in der QTA!

Kann mit terminaler Semantik behoben werden (Maximierung der Gleichheit), erfordert aber zumindest die Ungleichheit $true \neq false$, da sonst entartete Algebra mit nur einem einzigen Wert resultiert).

7.4. Beispiele für ADTen

Listen von Elementen

```
type List(T)
import Nat
operators
   [] :  → List
   _ :  _ :  T × List → List
   head :  List → T
   tail :  List → List
   length :  List → Nat
axioms
   head(x :  l) = x
   tail(x :  l) = l
   length( [] ) = 0
   length( x :  l ) = succ(length( l ))
```

_ : _ ist Notation für Infix-Operatoren (wie + in artithmetischen Ausdrücken)
import importiert bereits definierten Datentypen

Kellerspeicher (Stack)

LIFO-Prinzip: Last-In-First-Out-Speicher

```
type Stack(T)
import Bool
operators
   empty :  → Stack
   push :  Stack × T → Stack
   pop :  Stack → Stack
   top :  Stack → T
   is_empty :  Stack → Bool
axioms
   pop(push(s,x)) = s
   top(push(s,x)) = x
```

```
is_empty(empty) = true
is_empty(push(s,x)) = false
```

Keller implementiert über Liste

```
empty == []
push(l,x) == x:l
pop(x:l) == l
top(x:l) == x
is_empty([]) == true
is_empty(x:l) == false
```

Beispiel für Keller-Nutzung

Auswertung einfacher arithmetischer Ausdrücke
analysierte Sprache:

```
A ::= B =
B ::= Int | (B+B) | (B*B)
```

Beispiel:

$$((3 + (4 * 5)) * (6 + 7)) =$$

Beispiel für Keller: Regeln

Notation:

- $z_1..z_n S = push(...(push(S, z_n), ...z_1)$
- $[\![x + y]\!]$ berechnet Wert der Addition

Bearbeitungsregeln:

$$\begin{aligned}
value(t) &= eval\langle t, empty\rangle \\
eval\langle (t, S\rangle &= eval\langle t, (S\rangle \\
eval\langle *t, S\rangle &= eval\langle t, *S\rangle \\
eval\langle +t, S\rangle &= eval\langle t, +S\rangle \\
eval\langle)t, y * xS\rangle &= eval\langle)t, [\![x * y]\!]S\rangle \\
eval\langle)t, y + xS\rangle &= eval\langle)t, [\![x + y]\!]S\rangle \\
eval\langle)t, x(S\rangle &= eval\langle t, xS\rangle \\
eval\langle =, x \;\; empty\rangle &= x \\
eval\langle xt, S\rangle &= eval\langle t, xS\rangle
\end{aligned}$$

Bemerkungen zum Stack-Beispiel

- Regeln nicht ganz korrekt: eigentlich darf nur ein Element pro Schritt vom Keller genommen / gelesen werden!

  aber: Umformung in korrekte Form natürlich leicht möglich....

- Regeln werden jeweils von oben nach unten durchsucht, bis eine passende linke Seite gefunden wird

Keller: Komplexeres Beispiel

- analysierte Sprache:

  ```
  A ::= B =
  B ::= Int | B+B | B*B | (B)
  ```

  arithmetische Ausdrücke ohne vollständige Klammerung

- Lösungsansatz: zwei Stacks, einer für Operatoren, der zweite für Operanden

- Bemerkung: weiter verbesserte Methode wird z.B. in Taschenrechnern eingesetzt!

- Beispiel:

$$(3 + 4 * 5) * (6 + 7) =$$

Regeln für verbesserten Term-Auswerter

$$
\begin{aligned}
value(t) &= eval\langle t, empty, empty\rangle \\
eval\langle (t, S_1, S_2\rangle &= eval\langle t, (S_1, S_2\rangle \\
eval\langle *t, S_1, S_2\rangle &= eval\langle t, *S_1, S_2\rangle \\
eval\langle +t, *S_1, yxS_2\rangle &= eval\langle +t, S_1, [\![x * y]\!]S_2\rangle \\
eval\langle +t, S_1, S_2\rangle &= eval\langle t, +S_1, S_2\rangle \\
eval\langle)t, +S_1, yxS_2\rangle &= eval\langle)t, S_1, [\![x + y]\!]S_2\rangle \\
eval\langle)t, *S_1, yxS_2\rangle &= eval\langle)t, S_1, [\![x * y]\!]S_2\rangle \\
eval\langle)t, (S_1, S_2\rangle &= eval\langle t, S_1, S_2\rangle \\
eval\langle = t, +S_1, yxS_2\rangle &= eval\langle =, S_1, [\![x + y]\!]S_2\rangle \\
eval\langle = t, *S_1, yxS_2\rangle &= eval\langle =, S_1, [\![x * y]\!]S_2\rangle \\
eval\langle =, empty, x\ \ empty\rangle &= x \\
eval\langle xt, S_1, S_2\rangle &= eval\langle t, S_1, xS_2\rangle
\end{aligned}
$$

Warteschlange (Queue)

FIFO-Prinzip: First-In-First-Out-Speicher

```
type Queue(T)
import Bool
operators
   empty :  → Queue
   enter :  Queue × T → Queue
   leave :  Queue → Queue
   front :  Queue → T
   is_empty :  Queue → Bool
axioms
   leave(enter(empty,x)) = empty
   leave(enter(enter(s,x),y)) =
      enter(leave(enter(s,x)),y)
   front(enter(empty,x)) = x
   front(enter(enter(s,x),y)) =
      front(enter(s,x))
   is_empty(empty) = true
   is_empty(enter(s,x)) = false
```

Einsatz von Warteschlangen

Abarbeiten von Aufträgen in Eingangsreihenfolge: Prozeßverwaltung in Betriebssystemen, Speicherverwaltung, Druckeraufträge, Kommunikations-Software, etc.

7.5. Parametrisierte Datentypen in Meyer-Notation

- bisher: nur der fehlerfreie Fall spezifiziert (kein Auslesen aus einem leeren Stack)
- notwendig:
  - Angabe der fehlerträchtigen Operatoren (als partielle Funktionen)
  - **Preconditions** zur Charakterisierung der problemlosen Aufrufe derartiger Operatoren
  - in Java: Ausnahmebehandlung!
- zur Analyse sinnvoll: Festlegung von Konstruktoren, Selektoren, Prädikaten (bool'sche Selektoren)

Im Folgenden werden wir oft mit *Containertypen* zu tun haben. Das sind Datenstrukturen, die dazu benutzt werden, andere Daten zu speichern, zum Beispiel Arrays, Listen, Mengen usw. Für die Spezifikation dieser Typen ist es unerheblich, welche Art von Daten sie speichern (ob ganze Zahlen oder Bankkonten oder Personaldatensätze). Deswegen haben die Spezifikationen solcher Typen im allgemeinen einen (in Ausnahmefällen auch mehrere) *Parameter*. Man nennt diese Datentypen *generisch* oder *parametrisiert*. Generische ADT-Spezifikationen spezifizieren genau genommen keinen einzelnen ADT, sondern einen ADT für jede Instanz des Parameters. Sie sind also eigentlich *Datentypgeneratoren*.

```
Type ARRAY[ITEM]
Functions
   create: INT × INT → ARRAY[ITEM]                    Konstruktor
   put: ARRAY[ITEM] × INT × ITEM ↛ ARRAY[ITEM]        Konstruktor
   lower: ARRAY[ITEM] → INT                           Selektor
   upper: ARRAY[ITEM] → INT                           Selektor
   get: ARRAY[ITEM] × INT ↛ INT                       Selektor
   empty: ARRAY[ITEM] → BOOL                          Prädikat
Axioms ∀ i,j,k : INT, a : ARRAY[ITEM], x : ITEM
   lower(create(i,j))=i
   lower(put(a,i,x))=lower(a)
   upper(create(i,j))=j
   upper(put(a,i,x))=upper(a)
   k=i→ get(put(a,i,x),k)=x
   k≠ i → get(put(a,i,x),k)=get(a,k)
   empty(create(i,j))
   ¬ empty(put(a,i,x))
Preconditions ∀ a: ARRAY[ITEM]; i: INT; x: ITEM
   pre(put(a,i,x)): lower(a)≤ i ≤ upper(a)
   pre(get(a,i)): lower(a)≤ i≤ upper(a)
```

Was passiert, wenn man ein Array-Element überschreibt? `get` liefert den korrekten Wert zurück, aber nach den bisherigen Axiomen können wir nicht zeigen, daß gilt:

$$put(put(a,j,x),j,y) = put(a,j,y)$$

Wenn wir wollen, daß diese Gleichheit gilt, und daß außerdem die Reihenfolge des Hinzufügens von Elementen egal ist, müssen wir folgende Axiome hinzufügen:

$$\begin{array}{lll} i \neq j & \rightarrow & put(put(a,i,x),j,y) = put(put(a,j,y),i,x) \\ i = j & \rightarrow & put(put(a,i,x),j,y) = put(a,j,y) \end{array}$$

Weiterhin ist die Frage interessant, was passiert, wenn man mit `get(i,a)` auf ein Arrayelement zugreifen will, das nicht vorher mit `put(a,i,x)` initialisiert wurde. Hier ist die Antwort, daß der Funktionsaufruf `get(i,a)` zwar ein Element des Typs `ITEM` liefert, aber über dieses Element nichts bekannt ist.

7.5.1. Weitere Beispiele wichtiger Containertypen

```
Type SET[ITEM] -- endliche Mengen
Functions
   mt_set: SET[ITEM]                             Konstruktor
   add: SET[ITEM] × ITEM → SET[ITEM]             Konstruktor
   is_in: ITEM × SET[ITEM] → BOOL                Prädikat
   delete: ITEM × SET[ITEM] → SET[ITEM]
Axioms ∀ s: SET[ITEM]; x,y: ITEM
   x=y→ add(add(s,x),y)=add(s,x)
   x≠ y→ add(add(s,x),y)=add(add(s,y),x)
   is_in(x,mt_set)=false
   x=y→ is_in(x,add(s,y))=true
   x≠ y→ is_in(x,add(s,y))=is_in(x,s)
   delete(x,mt_set)=mt_set
   x=y→ delete(x,add(s,y))=s
   x≠ y→ delete(x,add(s,y))=add(delete(x,s),y)
Preconditions
   keine
```

Die nächste Container-Datenstruktur heißt Keller oder Stack und funktioniert nach dem LIFO-Prinzip (last in, first out).

```
Type STACK[X]
Functions
   mt_stack: STACK[X]                    Konstruktor
   push: X × STACK[X] → STACK[X]         Konstruktor
   pop: STACK[X] ↛ STACK[X]              Selektor
   top: STACK[X] ↛ X                     Selektor
   empty: STACK[X] → BOOL                Prädikat
Axioms ∀ x: X; s: STACK[X]
   pop(push(x,s))=s
   top(push(x,s))=x
   empty(mt_stack)=true
   empty(push(x,s))=false
Preconditions ∀ s: STACK[X]
   pre(pop(s)): ¬ empty(s)
   pre(top(s)): ¬ empty(s)
```

Bei dieser Spezifikation kann ein Keller beliebig viele Elemente enthalten. Wenn man hingegen nur beschränkt viel Platz zur Verfügung hat, muß man auch *push* mit einer Beschränkung belegen:

```
Type FSTACK[X]
Functions
```

```
   mt_stack:  NAT → FSTACK[X]                  Konstruktor
   push:  X × FSTACK[X] ⇸ FSTACK[X]            Konstruktor
   pop:  FSTACK[X] ⇸ FSTACK[X] x X             Selektor
   top:  FSTACK[X] ⇸ X                         Selektor
   capacity:  FSTACK[X] → NAT
   push_count:  FSTACK[X] → NAT
   empty:  FSTACK[X] → BOOL                    Prädikat
   full:  FSTACK[X] → BOOL                     Prädikat
Axioms ∀ x:  X; s:  FSTACK[X]; k:  NAT
   push_count(s)≤ capacity(s) ⇒ pop(push(x,s))=s
   push_count(s)≤ capacity(s) ⇒ top(push(x,s))=x
   capacity(mt_stack(k))=k
   capacity(push(x,s))=capacity(s)
   push_count(mt_stack(k))=0
   push_count(push(x,s))=push_count(s)+1
   empty(s)=true⇔ push_count(s) = 0
   full(s)=true⇔ push_count(s)≥ capacity(s)
Preconditions ∀ x:  X; s:  FSTACK[X]
   pre(push(x,s)):  ¬ full(s)
   pre(pop(s)):  ¬ empty(s)
   pre(top(s)):  ¬ empty(s)
```

7.6. Entwurf von ADT

- Wie komme ich zu den Gleichungen?
- Wann habe ich genug Gleichungen?
- Wann fehlt mir ein Axiom?

→ leider keine einfache Antworten zu erwarten.

Systematische Vorgehensweise

1. Festlegung der Konstruktoren
2. Definition geeigneter Selektoren
3. eventuell: Axiome zur Gleichsetzung von Konstruktortermen (vergleiche `Set`)
4. Selektoren auf Konstruktoren zurückführen
5. weitere Operationen festlegen
6. Fehlersituationen abfangen

Festlegung weiterer Operationen

- wenn möglich: direkt auf Konstruktoren und Selektoren zurückführen
- Regeln von links nach rechts als Ersetzungsregeln aufbauen
  - rechte Seite ‘einfacher’ als linke Seite
  - bei Rekursion:
    - ▹ Argumente ‘einfacher’ (echt monoton schrumpfend)
    - ▹ Abbruch garantiert? (Vorsicht! — wird oft vergessen)
  - oft: Vorgehen analog zu applikativen Algorithmen
- vollständige Fallunterscheidung: jeder Konstruktor wird berücksichtigt

8. Grundlegende Datenstrukturen

Grundprinzipien anhand einfacher Datenstrukturen
Implementierung von Stacks und Queues
basierend auf Arrays und verketteten Listen
Java-Beispiele aus Goodrich / Tamassia

8.1. Stack und Queue über Array

Wiederholung: Array
feste Größe
direkter Zugriff in $O(1)$

Stack

Stack: LIFO-Prinzip
objektorientierter Stack: Parameter Stack ist implizit!
daher Schnittstelle:

```
push(object o):  void
pop():  object
size():  integer
isEmpty():  boolean
top():  object
```

Änderung zu vorherigem Kapitel: `pop()` liefert oberstes Element zurück; `size()` als zusätzlicher Operator liefert Anzahl gespeicherter Objekte

Stack implementiert auf Array

Idee:
Array fester Größe `S`
Zeiger `t` zeigt auf Top-Element, initialisiert mit -1
Stack wächst und schrumpft von `S[0]` aus

Stack über `S[]` und `t`

```
push(object o):  void
   t:=t+1; S[t]:=o
pop():  object
   e:=S[t]; S[t]:= null; t:=t-1; return e
size():  integer
   return t+1
isEmpty():  boolean
   return t < 0
top():  object
   return S[t]
```

Stack über `S[]` und `t`: Bemerkungen

Komplexität aller Operationen: $O(1)$
Java-Implementierung inklusive Fehlerbehandlung : [GT98] Seite 72, Code 3.4

Queue

Queue: FIFO-Prinzip
Schnittstelle:

```
enqueue(object o):  void
dequeue():  object
size():  integer
isEmpty():  boolean
front():  object
```

Queue über Array: Idee

je ein Zeiger für Beginn `f` und Ende `r`
Queue ‘wandert’ durch das Array (zyklisch)
realisiert durch Modulo-Rechnung bzgl. Array-Größe `N`

Queue über Array: Realisierung

```
enqueue(object o):  void
   Q[r]:=o; r:= (r+1) mod N
dequeue():  object
   e:=Q[f]; Q[f]:=null; f:=(f+1) mod N; return e
size():  integer
   return (r - f + N ) mod N
isEmpty():  boolean
   return ( f=r )
```

```
front(): object
   return Q[f]
```

plus Abfangen der Fehlerfälle (volle Queue / leere Queue)

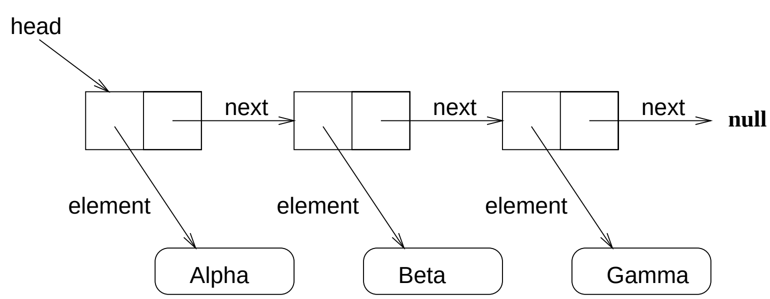
Bewertung der Array-Implementierungen

alle Operationen $O(1)$
Platzbedarf immer N unabhängig von gespeicherten Daten
Variante mit Erzeugen eines neuen Arrays bei Überlauf ist nicht mehr $O(1)$!
gesucht: dynamisch erweiterbare Speicherstruktur

8.2. Verkettete Listen

Idee: Liste von verketteten Knoten
Knoten: `element` + `next`
`element`: “Nutzdaten”
`next`: “Zeiger” auf nächsten Knoten
Notwendig: Verankerung der Liste mit head
englisch: ‘linked list’

Verkettete Listen graphisch



Operationen auf Knoten

```
Node(Object e, Node n)
setElement(Object e)
setNext(Node n)
getElement()
getNext()
```

Knotenoperationen realisieren Operationen auf verketteten Listen

Stack implementiert mit linked list

Variablen: `Node top`, `int size`

```
push(object o):  void
   Node n:=new Node();
   n.setElement(o);
   n.setNext(top); top:=n;
   size:=size+1
pop():  object
   e:=top.getElement();
   top:=top.getNext();
   size:=size-1; return e
size():  integer
   return size
isEmpty():  boolean
   return top = null
top():  object
   return top.getElement()
```

Interface `Stack`

```
public interface Stack {
  public int size();
  public boolean isEmpty();
  public Object top()
    throws StackException;
  public void push(Object element);
  public Object pop()
    throws StackException;
}
```

StackException

```
public class StackException
    extends RuntimeException {
  public StackException(String err) {
      super(err);
  }
}
```

ListStack

```
import java.awt.*;
import java.awt.event.*;
class ListStack implements Stack{
  private Node head;
  private int size;
  public ListStack () { /* Konstruktor */
    head = null; size = 0;
  }
  public int size(){
    return size;
  }
  public boolean isEmpty() {
    return (size == 0);
  }
  public Object pop() throws StackException {
    Object elem;
    if (isEmpty())
     throw new StackException("Keller ist leer");
    elem = head.getElement();
    head = head.getNext(); size--;
    return elem;
  }
  public Object top() throws StackException {
    if (isEmpty())
    throw new StackException("Keller ist leer");
    return  head.getElement();
  }
  public void push(Object element){
    Node n = new Node(element,head);
    head = n;  size++;
  }
}
```

Queue implementiert mit linked list

```
enqueue(object o):  void
   hinten Element anfügen (Spezialfall leere Liste)
dequeue():  object
   vorne Element entfernen
```

aus Effizienzgründen zwei Zeiger verwalten: `head` und `tail`

Enqueue mit einem Anker

```
public void enqueue(Object element){
Node n = new Node(element,null);
Node x = head;
if (isEmpty()){  head = n;}
else
 { while ( ! (x.getNext() == null) )
```

```
      {
         x = x.getNext();
      };
   x.setNext(n);
   };
  size++;
  }
```

Enqueue mit zwei Ankern

```
public void enqueue(Object element){
Node n = new Node(element,null);
if (isEmpty()){  head = n; tail = n ; }
else
    { tail.setNext(n);
    };
tail = n;
size++;
}
```

Effizienzunterschied der Queue-Realisierungen

- verkettete Liste mit einem Anker:

  fast alle Operationen $O(1)$:

  Enqueue ist $O(n)$!

- Verkettete Liste mit zwei Ankern:

  alle Operationen $O(1)$

  ... aber erhöhter Verwaltungsaufwand (Verwalten zweier Anker, Spezialbehandlung bei Löschen in einelementiger Liste notwenig)

Bemerkung: Folgender Java-Code ohne `import` und ohne Kommentare!

SListQueue.java

```
/* Queue "uber Liste mit einem Anker */
class SListQueue implements Queue{
  private Node head;
  private int size;
  public   SListQueue () {
    head = null; size = 0;
  }
  public int size(){
    return size;
  }
  public boolean isEmpty() {
    return (size == 0);
  }
```

```
  public Object dequeue()
    throws QueueException {
    Object elem;
    if (isEmpty())
      throw new QueueException("Schlange ist leer");
    elem = head.getElement();
    head = head.getNext(); size--;
    return elem;
  }
  public Object front()
    throws QueueException {
    if (isEmpty())
    throw new QueueException("Schlange ist leer");
    return  head.getElement();
  }
  public void enqueue(Object element){
    Node n = new Node(element,null);
    Node x = head;
    if (isEmpty()){  head = n;}
    else {
     while ( ! (x.getNext() == null) )
       x = x.getNext();
     x.setNext(n);
    };
    size++;
  }
}
```

ListQueue.java

```
/* Queue "uber Liste mit zwei Ankern */
class ListQueue implements Queue {
  private Node head;
  private Node tail;
  private int size;
  public   ListQueue () {
    head = null; tail = null;  size = 0;
  }
  public int size(){
    return size;
  }
  public boolean isEmpty() {
    return (size == 0);
  }
  public Object dequeue()
    throws QueueException {
    Object elem;
    if (isEmpty())
     throw new QueueException("Schlange ist leer");
    elem = head.getElement();
    head = head.getNext();
    size--;
    if (isEmpty()) { tail = null; };
    return elem;
  }
  public Object front()
    throws QueueException {
    if (isEmpty())
    throw new QueueException("Schlange ist leer");
    return  head.getElement();
  }
```

```
  public void enqueue(Object element){
     Node n = new Node(element,null);
    if (isEmpty()){  head = n; tail = n ; }
    else
      { tail.setNext(n);
      };
     tail = n;  size++;
  }
}
```

8.3. Doppelt verkettete Listen

Motivation: Deque (sprich “Deck”) für Double-ended queue

Generalisierung von Stack und Queue

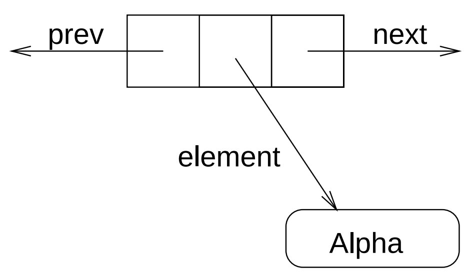
Operationen: `insertFirst`, `insertLast`, `removeFirst`, `removeLast`, `first`, `last`, `size`, `isEmpty`

Realisierung mit einfach verketteten Listen möglich: aber nicht mehr $O(1)$ für alle Operationen!

daher: Implementierung mit *doppelt verketteten Listen*

Deque-Knoten: neben Zeiger `next` auch Zeiger `prev`

Knoten einer doppelt verketteten Liste



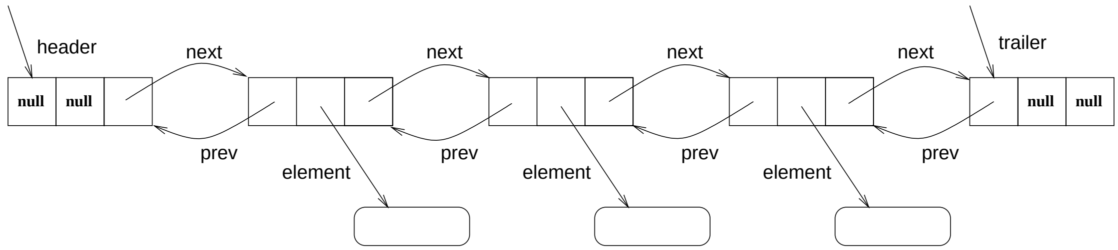
Realisierung einer Deque über doppelt verketteter Liste

Problem: Spezialfälle bei leerer Liste, einelementiger Liste

Daher: zwei Verwaltungs-Knoten ohne Daten `header` und `trailer`

Implementierung siehe [GT98], Seite 99, Code Fragment 3.14

Deque über doppelt verketteter Liste



Vergleich von Deque-Realisierungen

	einfach verkettet		doppelt
	ein Anker	zwei Anker	verkettet
`insertFirst`	$O(1)$	$O(1)$	$O(1)$
`insertLast`	$O(n)$	$O(1)$	$O(1)$
`deleteFirst`	$O(1)$	$O(1)$	$O(1)$
`deleteLast`	$O(n)$	$O(n)$	$O(1)$

8.4. Sequenzen

weiterer Container-Datentyp: Sequenzen

Goodwich / Tamassia Kapitel 4

ranked sequences: Einfügen an Position, Zugriff über Position (‘rank’), Löschen an Position

Array-Implementierung: Verschieben der ‘Rest’-Liste bei `insertElemAtRank` und `removeElemAtRank`

doppelt-verkettete Liste: Einhängen an Position

Sortierte Sequenzen

	über `Deque`	über `Array`
absolute Position	$O(n)$	$O(1)$
Suchen einer Position über Wert	$O(n)$	$O(\log n)$ binäre Suche
Einfügen an gefundener Position	$O(1)$	$O(n)$ Werte verschieben
Löschen an gefundener Position	$O(1)$	$O(n)$ Werte verschieben

9. Bäume

Begriffe und Konzepte
Spezifikation von Bäumen
Realisierung und Operationen für Bäume
Suchbäume

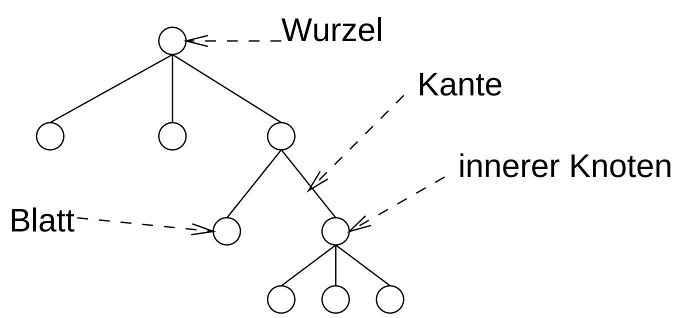
9.1. Bäume: Begriffe und Konzepte

Baum: Menge durch *Kanten* verbundenen *Knoten*

- *Wurzel*: genau ein ausgezeichneter Knoten
- jeder Knoten außer der Wurzel ist durch genau eine *Kante* mit seinem *Vaterknoten* (auch kurz Vater, Elter, Vorgänger) verbunden

  er wird dann als *Kind* (Sohn, Nachfolger) dieses Knotens bezeichnet
- ein Knoten ohne Kinder heißt *Blatt*
- alle anderen Knoten werden als *innere Knoten* bezeichnet

Begriffe am Beispiel



Weitere Begriffe

- *Vorfahren* sind Knoten in einer mehrstufigen Vaterbeziehung, *Nachfahren* / Nachkommen analog für Kinder
- jeder Knoten außer der Wurzel muß Nachfahre der Wurzel sein:

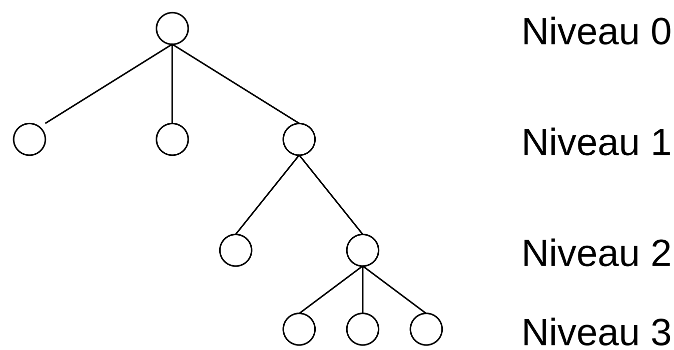
  - Baum ist zusammenhängend

- kein Knoten ist Nachfahre von sich selbst
  - keine Zyklen
- üblich: *geordnete Bäume*
  - Kinder sind als Liste angeordnet

... und noch mehr Begriffe

- *Niveau* eines Knotens (auch: Höhe des Knotens):
  Länge des *Pfades* von ihm zur Wurzel
- *Höhe eines Baums*:
  größtes Niveau eines Blattes plus 1

Niveaus in einem Baum der Höhe 4



Beispiele für Bäume

arithmetische Ausdrücke als Baum

rekursive Funktionsauswertung als Baum, etwa für Fibonacci-Zahlen

$$F(1) = 1; \quad F(2) = 2; \quad F(n+2) = F(n+1) + F(n)$$

Dateianordnung in einem File-System

Kapitel / Abschnitte eines Buches

Teileliste einer Maschine

Pseudo-Code für allgemeinen Baum

```
class TreeKnoten {
   int anzahlKinder;
   TreeKnoten[] Kind; /* Zeiger auf Kinder */
   Object element; /* Nutzdaten */
}
```

statt Array of: Liste mit `nextKind`-Operation
oft auch zusätzliche Attribute:

Binäre Bäume

Binäre Bäume:

- spezieller geordneter Baum
- jeder Knoten hat maximal zwei Kinder
- diese bilden Wurzeln des *linken Teilbaums* und des *rechten Teilbaums*

9.2. ADT für Binär-Bäume

Eigenschaften von binären Bäumen mittels ADT
wichtige Baum-Algorithmen anhand binärer Bäume
binäre Bäume:

- Knoten mit zwei Zeigern `left` und `right`

  in Java analog zu Knoten einer doppelt-verketten Liste!
- Wurzelknoten als Einstiegspunkt
- keine Zyklen

Binärbaum

Bäume mit Verzweigungsgrad zwei

```
type Tree(T)
import Bool
operators
   empty :  → Tree
   bin :  Tree × T × Tree → Tree
   left :  Tree → Tree
   right :  Tree → Tree
   value :  Tree → T
```

```
    is_empty :  Tree → Bool
axioms
    left(bin(x,b,y)) = x
    right(bin(x,b,y)) = y
    value(bin(x,b,y)) = b
    is_empty(empty) = true
    is_empty(bin(x,b,y)) = false
```

Beispiel für Binärbaum

Einlesen eines Terms in Binärbaum
→ Modifikation des zweiten Stack-Beispiels (Folie 146 bis 146)

- zweiter Stack enthält Bäume, keine Operanden
- statt Termauswertung:

$$\begin{array}{rll} [\![x]\!] & \text{bedeutet} & bin(empty, x, empty) \\ [\![x * y]\!] & \text{bedeutet} & bin(x, *, y) \\ [\![x + y]\!] & \text{bedeutet} & bin(x, +, y) \end{array}$$

Regeln für Baum-Einlesen

$$\begin{array}{rcl} value(t) & = & eval\langle t, mt, mt\rangle \\ eval\langle (t, S_1, S_2\rangle & = & eval\langle t, (S_1, S_2\rangle \\ eval\langle * t, S_1, S_2\rangle & = & eval\langle t, * S_1, S_2\rangle \\ eval\langle + t, * S_1, yxS_2\rangle & = & eval\langle + t, S_1, [\![x * y]\!]S_2\rangle \\ eval\langle + t, S_1, S_2\rangle & = & eval\langle t, + S_1, S_2\rangle \\ eval\langle)t, + S_1, yxS_2\rangle & = & eval\langle)t, S_1, [\![x + y]\!]S_2\rangle \\ eval\langle)t, * S_1, yxS_2\rangle & = & eval\langle)t, S_1, [\![x * y]\!]S_2\rangle \\ eval\langle)t, (S_1, S_2\rangle & = & eval\langle t, S_1, S_2\rangle \\ eval\langle = t, + S_1, yxS_2\rangle & = & eval\langle =, S_1, [\![x + y]\!]S_2\rangle \\ eval\langle = t, * S_1, yxS_2\rangle & = & eval\langle =, S_1, [\![x * y]\!]S_2\rangle \\ eval\langle =, mt, x\ \ mt\rangle & = & x \\ eval\langle xt, S_1, S_2\rangle & = & eval\langle t, S_1, [\![x]\!]S_2\rangle \end{array}$$

mt steht wieder für *empty*

Auslesen eines Baumes in unterschiedlichen Notationen I

Infix-Schreibweise:

```
term(empty) = ' '
term([[x*y]]) = '(' ++ term(x) ++ '*' ++ term(y) ++ ')'
term([[x+y]]) = '(' ++ term(x) ++ '+' ++ term(y) ++ ')'
term([[x]]) = x
```

rekursiver Aufruf von `term` zur Konstruktion eines Strings

Auslesen eines Baumes in unterschiedlichen Notationen II

Präfix-Schreibweise:

```
term(empty) = ' '
term([[x*y]]) = '*' ++ term(x) ++ term(y)
term([[x+y]]) = '+' ++ term(x) ++ term(y)
term([[x]]) = x
```

Postfix-Schreibweise:

```
term(empty) = ' '
term([[x*y]]) = term(x) ++ term(y) ++ '*'
term([[x+y]]) = term(x) ++ term(y) ++ '+'
term([[x]]) = x
```

Verallgemeinerung der Auslesevarianten

systematisches Abarbeiten aller Knoten
(engl. *Traversal*, Traversierung, 'Baumdurchlauf', kurz Durchlauf)

- Infix-Anordnung: *Inorder-Durchlauf*
- Präfix-Anordnung: *Preorder-Durchlauf*

  Beispiel: Stückliste dargestellt mit Einrückungen
- Postfix-Anordnung: *Postorder-Durchlauf*

  Beispiel: Stückliste mit Aufsummierung (Bsp.: disk usage `du` in Linux)

Rekursiver Pseudo-Code für Durchläufe

```
inorder(k):
   Besuche linken Teilbaum mit inorder(k.left);
   Verarbeite k.Element;
   Besuche rechten Teilbaum mit inorder(k.right);
preorder(k):
   Verarbeite k.Element;
   Besuche linken Teilbaum mit preorder(k.left);
   Besuche rechten Teilbaum mit preorder(k.right);
```

```
postorder(k):
   Besuche linken Teilbaum mit postorder(k.left);
   Besuche rechten Teilbaum mit postorder(k.right);
   Verarbeite k.Element;
```

Binärbaum als ADT nach Meyer I

Types $BINTREE[T]$
Functions

$mt\_tree : BINTREE[T]$
$make : BINTREE[T] \times T \times BINTREE[T] \to BINTREE[T]$
$left : BINTREE[T] \nrightarrow BINTREE[T]$
$right : BINTREE[T] \nrightarrow BINTREE[T]$
$data : BINTREE[T] \nrightarrow T$
$is\_mt : BINTREE[T] \to BOOL$
$is\_in : T \times BINTREE[T] \to BOOL$
$height : BINTREE[T] \to INT$
$balanced : BINTREE[T] \to BOOL$
$subtrees : BINTREE[T] \to SET[BINTREE[T]]$
$preorder\_traversal : BINTREE[T] \to LIST[T]$
$inorder\_traversal : BINTREE[T] \to LIST[T]$
$postorder\_traversal : BINTREE[T] \to LIST[T]$

Binärbaum als ADT nach Meyer II

Axioms $\forall\, t_1, t_2 : BINTREE[T];\ x, y : T$

$left(make(t_1, x, t_2)) = t_1$
$right(make(t_1, x, t_2)) = t_2$
$data(make(t_1, x, t_2)) = x$
$is\_mt(mt\_tree) = true$
$is\_mt(make(t_1, x, t_2)) = false$
$is\_in(x, mt\_tree) = false$
$x = y \Rightarrow is\_in(x, make(t_1, y, t_2)) = true$
$x \neq y \Rightarrow (is\_in(x, make(t_1, y, t_2)) \Leftrightarrow is\_in(x, t_1) \vee is\_in(x, t_2))$
$height(mt\_tree) = 0$
$height(make(t_1, x, t_2)) = max\{height(t_1), height(t_2)\} + 1$
$balanced(mt\_tree) = true$
$balanced(make(t_1, x, t_2)) \Leftrightarrow$
$\quad |height(t_1) - height(t_2)| \leq 1 \wedge balanced(t_1) \wedge balanced(t_2)$
$subtrees(mt\_tree) = \{mt\_tree\}$
$subtrees(make(t_1, x, t_2)) = \{make(t_1, x, t_2)\} \cup subtrees(t_1) \cup subtrees(t_2)$
$preorder\_traversal(mt\_tree) = nil$
$preorder\_traversal(make(t_1, x, t_2))$
$\quad = \langle x \rangle \frown preorder\_traversal(t_1) \frown preorder\_traversal(t_2)$
$inorder\_traversal(mt\_tree) = nil$
$inorder\_traversal(make(t_1, x, t_2))$
$\quad = inorder\_traversal(t_1) \frown \langle x \rangle \frown inorder\_traversal(t_2)$
$postorder\_traversal(mt\_tree) = nil$
$postorder\_traversal(make(t_1, x, t_2))$
$\quad = postorder\_traversal(t_1) \frown postorder\_traversal(t_2) \frown \langle x \rangle$

Preconditions $\forall\, t : BINTREE[T]$

pre$(left(t)) :\ \neg is\_mt(t)$
pre$(right(t)) :\ \neg is\_mt(t)$
pre$(data(t)) :\ \neg is\_mt(t)$

9.3. Suchbäume

Unterstützung der Suche mittels eines *Schlüsselwerts*

- Schlüsselwert: key
- Element eines Knotens enthält key und Nutzdaten
- Beispiel: Telefonbuch, Adressenliste, Studentenverzeichnis (Schlüssel: Matrikelnummer)

Derartige Datenstrukturen werden auch als Dictionaries / Wörterbücher bezeichnet

Binäre Suchbäume

im Falle binärer Bäume für inneren Knoten k:

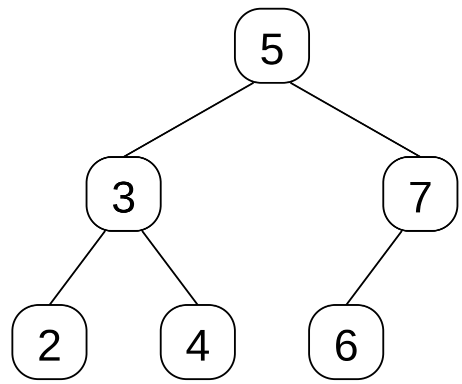
- $k.element$ enthält Schlüsselwert $k.key$
- alle Schlüsselwerte im linken Teilbaum $k.left$ sind kleiner als $k.key$
- alle Schlüsselwerte im rechten Teilbaum $k.right$ sind größer als $k.key$

ab jetzt: Suchbäume als binäre Suchbäume

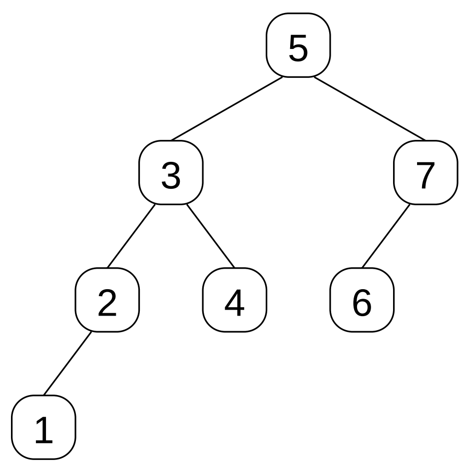
Suche in Suchbäumen als applikativer Algorithmus

```
suche(k,x):
if k = null
then false
else if k.key = x
     then true
     else if k.key > x
          then suche(k.left,x)
          else suche(k.right,x)
fi   fi   fi
```

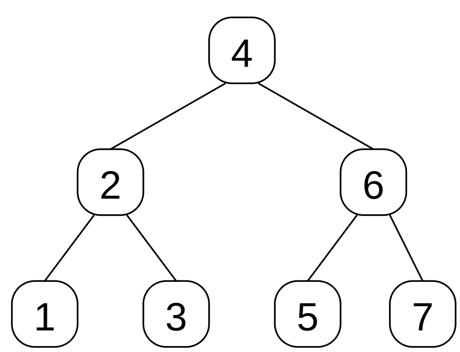
Beispiel Suchbaum für { 2...7 }



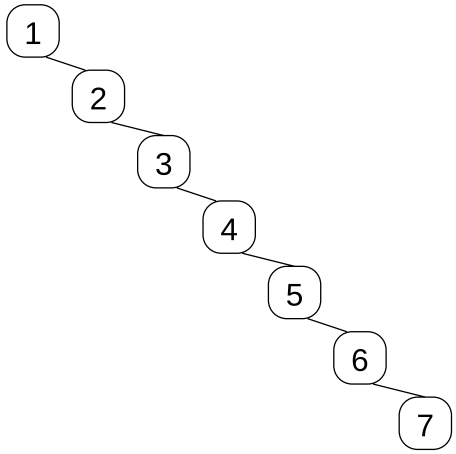
Suchbaum nach Einfügen von 1



Besserer Suchbaum für { 1...7 }



Entarteter Suchbaum für { 1...7 }



Rekursive Suche mit Knotenrückgabe

```
baumSuche(k,x):
if k = null or k.key = x
then return k
else if k.key > x
     then return baumSuche(k.left,x)
     else return baumSuche(k.right,x)
     fi
fi
```

Iterative Suche mit Knotenrückgabe

```
iterativBaumSuche(k,x):
while not (k = null) and not (k.key = x)
   do if k.key > x
      then k := k.left
      else k := k.right
      fi
   od
return k
```

Minimales Element

```
baumMin(k,x):
while not ( k.left = null)
   do k := k.left
   od
return k
```

Maximales Element

```
baumMax(k,x):
while not ( k.right = null)
   do k := k.right
   od
return k
```

Weitere Suchoperationen

- Successor: nächstgrößeres Element
- Predecessor: nächstkleineres Element

nur effizient möglich wenn jeder Knoten auch einen Zeiger `parent` besitzt!

```
successor(x):
if x.right ≠ null
then return baumMin (x.right )
else node y := x.parent;
     while y ≠ null and x = y.right
      do x := y ;
         y := y.parent
      od
     return y
fi
```

`predecessor` symmetrisch.

Wann ist ein Suchbaum 'gut'?

Suchzeit bestimmt durch *Höhe* des Baumes!
Wie hoch kann ein Baum werden bei n Knoten?

- entarteter Baum: Höhe n
- gut ausgeglichener Baum: $\log n$
  - Knoten des Niveaus 0: 1
  - Knoten des Niveaus 1: 2
  - Knoten des Niveaus 2: 4
  - ...
  - Knoten des Niveaus k: 2^k

  ein Baum der Höhe $k+1$ kann somit maximal $2^k + 2^{k-1} + \cdots 4 + 2 + 1 = 2^{k+1}$ Suchwerte fassen!

Operationen auf Suchbäumen

Suchen

- Operation `suche(k,x)`, rekursiv oder iterativ

Einfügen

- wie Operation `suche(k,x)`, aber im Falle $k = null$ Einhängen eines neuen Knotens (beim Vater!)

  ... siehe Pseudo-Code

Einfügen in binärem Suchbaum

```
baumEinfügen(T,x):
/* T ist Tree, x ist neuer Knoten */
vater := null;
sohn := T.root;
while not ( sohn = null)
   do vater := sohn;
      if x.key < sohn.key
      then sohn := sohn.left
      else sohn := sohn.right
      fi
   od
if vater = null
then T.wurzel := x
else if x.key < vater.key
   then vater.left := x
   else vater.right := x
fi fi
```

Operationen auf Suchbäumen II

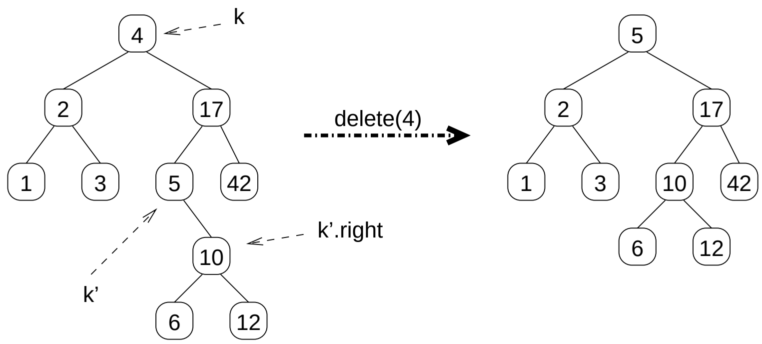
Löschen des Elements in Knoten k

- falls k ein Blatt: trivial.
- falls k kein Blatt, aber nur ein Sohn:

  Entferne k, ziehe Sohn auf die Position von k hoch
- falls k kein Blatt:
  - suche ersten (nach Inorder) Knoten k' im rechten Teilbaum von k, der keinen linken Sohn besitzt (= `baumMin(k.right)` = Successor von k!)
  - k' beinhaltet nächsten Wert in Sortierreihenfolge

- tausche Werte von k und k'
- ersetze Knoten k' durch rechtes Kind von k'

Beispiel für Löschen



Details beim Löschen in binärem Suchbaum

Suche des zu löschenden Knotens als Vorbereitung

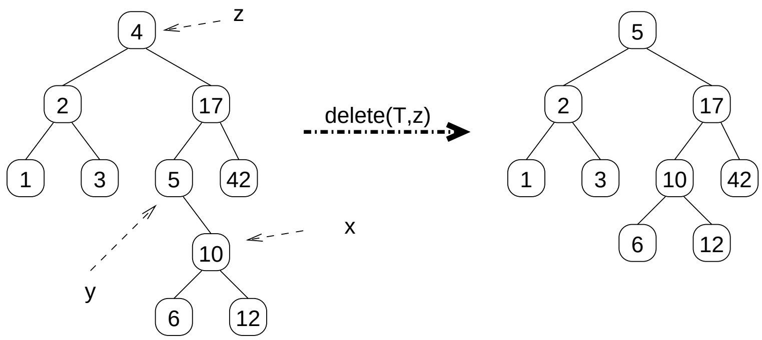
Zeiger `parent` als Zeiger zum Vater eines Knotens vereinfacht den Algorithmus, kann aber auch durch äquivalente Methode realisiert werden

Löschen in binärem Suchbaum

```
baumLöschen(T,z):
/* T ist Tree, Inhalt von z ist zu loeschen */
node y; /* y ist der zu entfernende Knoten */
node x; /* x ist der hochzuziehende Knoten */
if z.left = null or z.right = null
then y := z
else y := successor (z)
fi; /* y korrekt gesetzt */
if y.left ≠ null
then x := y.left
else x := y.right
fi; /* x korrekt bestimmt */
if x ≠ null
then x.parent := y.parent
fi;
if y.parent= null
then T.root := x
else if y = y.parent.left
     then y.parent.left := x
     else y.parent.right := x
     fi
fi; /* x wurde hochgezogen */
if y ≠ z
```

```
then z.key := y.key;
     ...  /* kopiere weitere Nutzdaten */
fi /* falls notwendig:
      Austausch der Daten y und z */
```

Beispiel für Löschalgorithmus



Komplexität der Operationen

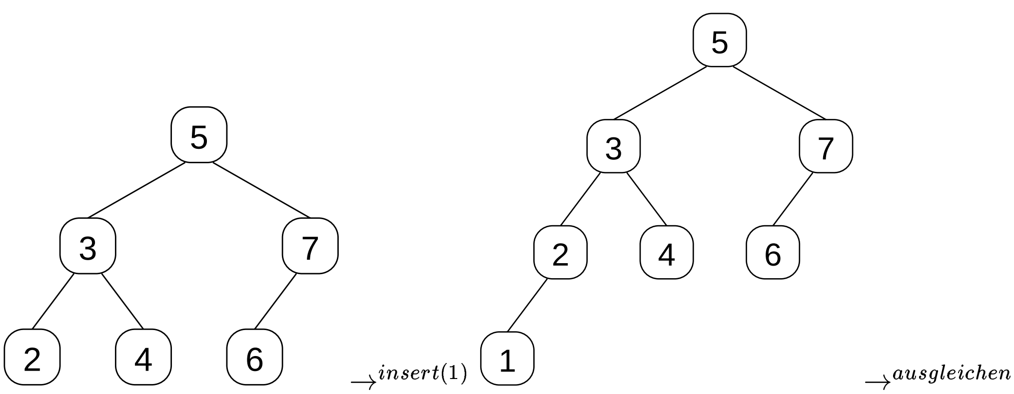
Alle vorgestellten Operationen auf binären Suchbäumen haben die Komplexität $O(h)$ bei einer maximalen Baumhöhe h.

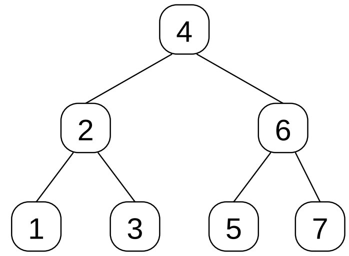
Alle Operationen bearbeiten nur einen Pfad von der Wurzel zu einem Knoten (Suche, Einfügen) bzw. zusätzlich den direkten Pfad zum Successor-Knoten.

Ziel: h möglichst klein bei vorgegebener Zahl n von gespeicherten Schlüsseln.

9.4. Ausgeglichene Bäume

Wie verhindert man daß Suchbäume entarten??
jeweils ausgleichen ist zu aufwendig:





... diese Folge zeigt, daß beim Ausgleichen eventuell jeder Knoten bewegt werden müßte!

Lösungsideen

- abgeschwächtes Kriterium für ausgeglichene Höhe

  Beispiel: AVL Bäume

- ausgeglichene Höhe, aber unausgeglichener Verzweigungsgrad

  Beispiel: B-Bäume

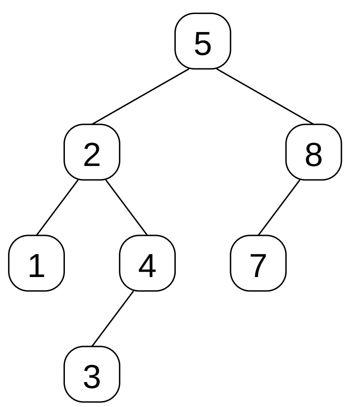
AVL-Bäume

AVL für Adelson-Velskii und Landis (russische Mathematiker)
binäre Suchbäume mit *AVL-Kriterium*:

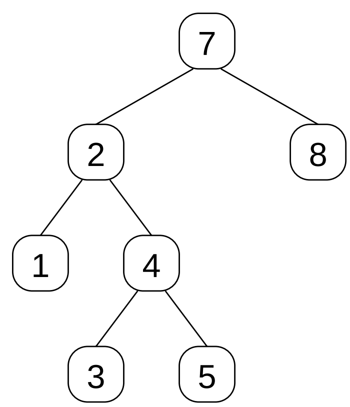
> für jeden (inneren) Knoten gilt: Höhe des linken und rechten Teilbaums differieren maximal um 1

Bemerkung: es reicht nicht dieses nur für die Wurzel zu fordern! Beide Teilbäume der Wurzel könnten entartet sein.

AVL-Eigenschaft am Beispiel



AVL



nicht AVL

Höhe von AVL-Bäumen

Wieviel Knoten k_{min} hat ein AVL-Baum der Höhe h mindestens?

- rekursive Beziehung:
  - $k_{min}(0) = 1$
  - $k_{min}(1) = 2$
  - $k_{min}(n) = k_{min}(n-1) + k_{min}(n-2) + 1$
  - wächst vergleichbar der Fibonacci-Reihe!
- exakter Wert (Vorsicht - nicht einfach zu berechnen!):

$$H_{AVL}(n) \leq 1.44 \log(n+2) - 0.328$$

Einfügen in AVL-Bäume

Einfügen eines Schlüssels mit üblichen Algorithmus
Danach kann die AVL-Eigenschaft verletzt sein:

- $Balance = left.height - right.height$
- AVL-Eigenschaft: $balance \in \{-1, 0, +1\}$
- nach Einfügen: $balance \in \{-2, -1, 0, +1, +2\}$

reparieren mittels *Rotation* und *Doppelrotation*

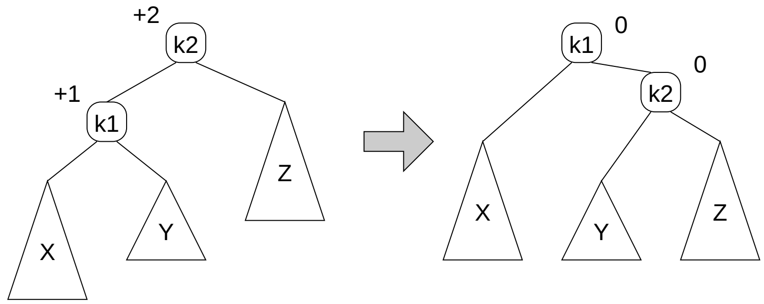
Fallunterscheidung beim Einfügen

Verletzung der AVL-Eigenschaft tritt ein bei

1. Einfügen in linken Teilbaum des linken Kindes
2. Einfügen in rechten Teilbaum des linken Kindes
3. Einfügen in linken Teilbaum des rechten Kindes
4. Einfügen in rechten Teilbaum des rechten Kindes

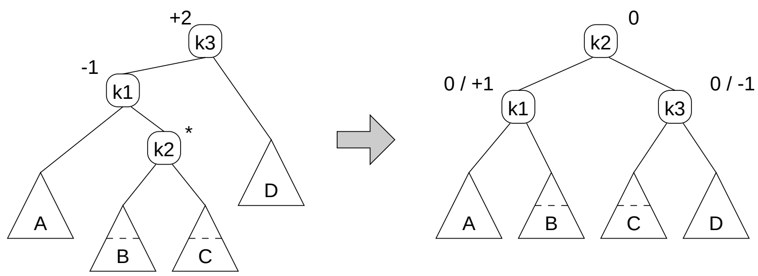
1 und 4 sowie 2 und 3 sind symmetrische Problemfälle

Einfache Rotation



Rotation mit linkem Kind nach rechts, analoge Operation nach links (spiegelbildlich)

Doppelrotation



Doppelrotation mit linkem Kind nach rechts, analoge Operation nach links (spiegelbildlich)

Rotationen am Beispiel

```
insert 3, 2, 1 → einfache Rotation nach rechts (2,3)
insert 4, 5 → einfache Rotation nach links (4,3)
insert 6 → einfache Rotation (Wurzel) nach links (4,2)
insert 7 → einfache Rotation nach links (6,5)
insert 16, 15 → Doppelrotation nach links (7,15,16)
insert 13 + 12 + 11 + 10 → jeweils einfache Rotationen
insert 8, 9 → Doppelrotation nach rechts
```

Rotationen in Einzelfällen

- Verletzung der AVL-Eigenschaft tritt ein bei
  - Einfügen in linken Teilbaum des linken Kindes
    → Rotation mit linkem Kind

- Einfügen in rechten Teilbaum des linken Kindes
  → Doppelrotation mit linkem Kind
- Einfügen in linken Teilbaum des rechten Kindes
  → Doppelrotation mit rechtem Kind
- Einfügen in rechten Teilbaum des rechten Kindes
  → Rotation mit rechtem Kind

AVL-Knoten in Java

```
class AvlNode {
   ...
   Comparable:  element;
   AvlNode right;
   AvlNode left;
   int height; // initialisiert zu 0 bei Blatt
   ..
   private static int height (AvlNode t )
      { return t == null ?  -1 :  t.height ; }
   ....
}
```

Einfügen (Skizze)

```
private AvlNode insert ( Comparable x, AvlNode t ) {
if ( t == null ) t = new AvlNode ( x, null, null );
else if (x.compareTo (t.element ) < 0 )
{
   t.left = insert ( x, t.left );
   if ( height(t.left) - height(t.right) == 2 )
      if ( x.compareTo (t.left.element) < 0 )
         t = rotateWithLeftChild ( t );
      else     t = doubleWithLeftChild ( t );
}
else if (x.compareTo (t.element ) > 0 )
{  analog mit rechts  };
t.height = max (height(t.left),height (t.right)) +1 ;
return t;
}
```

Einfache Rotation in Java

```
private static AvlNode rotateWithLeftChild(AvlNode k2)
{
 AvlNode k1 = k2.left;
 k2.left = k1.right;
```

```
 k1.right = k2;
 k2.height = max(height(k2.left),height (k2.right))+1;
 k1.height = max(height(k1.left), k2.height))+1;
 return k1;
}
```

Doppelte Rotation in Java

```
private static AvlNode doubleWithLeftChild(AvlNode k3)
{
   k3.left = rotateWithRightChild ( k3.left );
   return rotateWithLeftChild( k3 );
}
```

B-Bäume

Idee: Baumhöhe vollständig ausgeglichen, aber Verzweigungsgrad variiert

- Ausgangspunkt: ausgeglichener, balancierter Suchbaum
- *Ausgeglichen* oder *balanciert*: alle Pfade von der Wurzel zu den Blättern des Baumes gleich lang
- mehrere Zugriffsattributwerte auf einer Seite
- *Mehrweg-Bäume*

Prinzip des B-Baumes

- *B-Baum* von Bayer (B für balanciert, breit, buschig, Bayer, **NICHT:** binär)
- dynamischer, balancierter Suchbaum

Mehrwegebaum wäre völlig ausgeglichen, wenn

1. alle Wege von der Wurzel bis zu den Blättern gleich lang
2. jeder Knoten gleich viele Einträge

vollständiges Ausgleichen wäre zu aufwendig, deshalb *B-Baum-Kriterium*:

> *Jede Seite außer der Wurzelseite enthält zwischen m und $2m$ Schlüsselwerte*

Eigenschaften eines B-Baumes

- m heißt *Ordnung* des B-Baums
- i Schlüsselwerte ($m \leq i \leq 2m$) im inneren Knoten $\rightarrow i + 1$ Unterbäume
- Höhe des B-Baums bei minimaler Füllung: $log_m(n)$
- n Datensätze $\Rightarrow$ in $log_m(n)$ Seitenzugriffen von der Wurzel zum Blatt

Vorsicht: in einigen Büchern wird als Ordnung der maximale Verzweigungsgrad bezeichnet, also hier $2m + 1$!

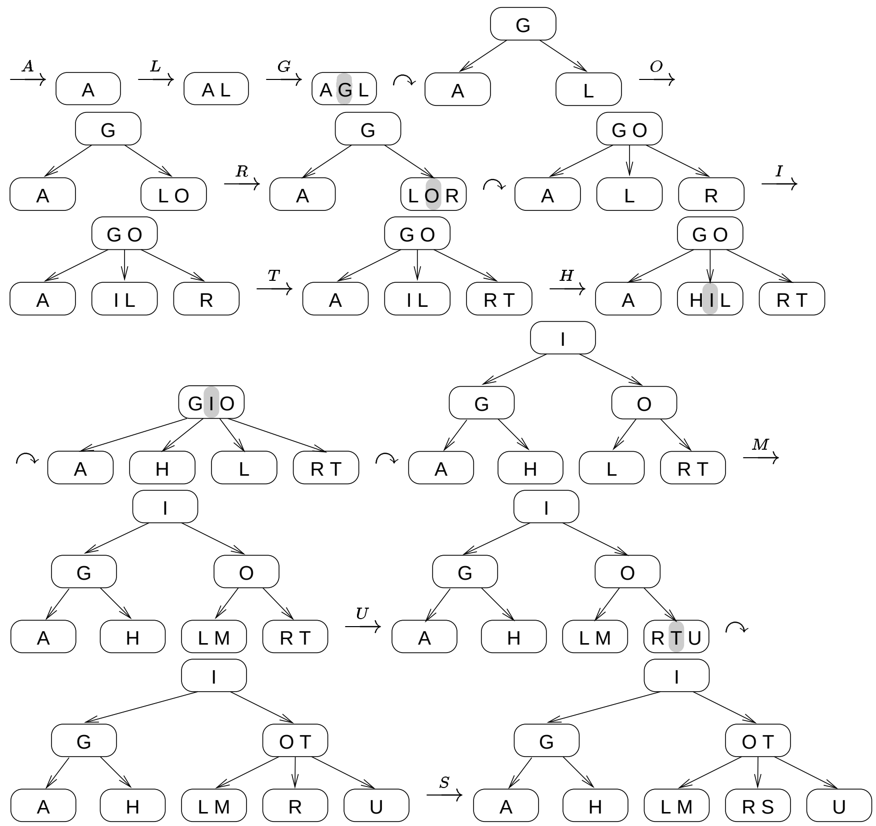
Definition B-Baum

B-Baum der Ordnung m:

1. Jede Seite enthält höchstens $2m$ Elemente.
2. Jede Seite, außer der Wurzelseite, enthält mindestens m Elemente.
3. Jede Seite ist entweder eine Blattseite ohne Nachfolger oder hat $i + 1$ Nachfolger, falls i die Anzahl ihrer Elemente ist.
4. Alle Blattseiten liegen auf der gleichen Stufe.

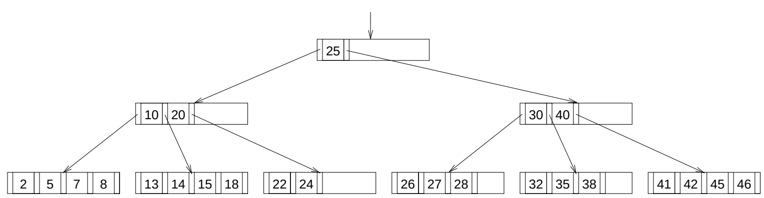
Einfügen in einen B-Baum: Ein Beispiel

Buchstaben des Wortes “ALGORITHMUS” der Reihe nach in einen B-Baum der Ordnung 1 einfügen



Suchen in B-Bäumen

- Startend auf Wurzelseite Eintrag im B-Baum ermitteln, der den gesuchten Schlüsselwert w überdeckt



Einfügen in B-Bäumen

Einfügen eines Wertes w

- Blattseite suchen
- passende Seite hat $n < 2m$ Elemente $\rightarrow$ w einsortieren
- passende Seite hat $n = 2m$ Elemente $\rightarrow$ neue Seite erzeugen
  - ersten m Werte auf Originalseite

  - letzten m Werte auf neue Seite
  - mittleres Element in Vaterknoten nach oben

- eventuell dieser Prozeß rekursiv bis zur Wurzel

  - B-Bäume wachsen an der Wurzel!

Löschen in B-Bäumen

Problem: bei weniger als m Elementen auf Seite: Unterlauf

- entsprechende Seite suchen
- w auf Blattseite gespeichert

  $\Rightarrow$ Wert löschen, eventuell Unterlauf behandeln

- w nicht auf Blattseite gespeichert $\Rightarrow$

  - Wert löschen
  - durch lexikographisch nächstkleineres Element von einer Blattseite ersetzen
  - eventuell auf Blattseite Unterlauf behandeln

  analog Löschen in klassischen Suchbaum!

Löschen in B-Bäumen: Unterlauf

Unterlaufbehandlung

- Ausgleichen mit einer benachbarten Seite (benachbarte Seite hat n Elemente mit $n > m$)
- oder

  - Zusammenlegen zweier Seiten zu einer (Nachbarseite hat $n = m$ Elemente)
  - das passende “mittlere” Element vom Vaterknoten dazu
  - in Vaterknoten eventuell (rekursiv) Unterlauf behandeln

  $\rightarrow$ B-Bäume schrumpfen an der Wurzel!

Komplexität der Operationen

- Aufwand beim Einfügen, Suchen und Löschen im B-Baum immer $O(log_m(n))$ Operationen

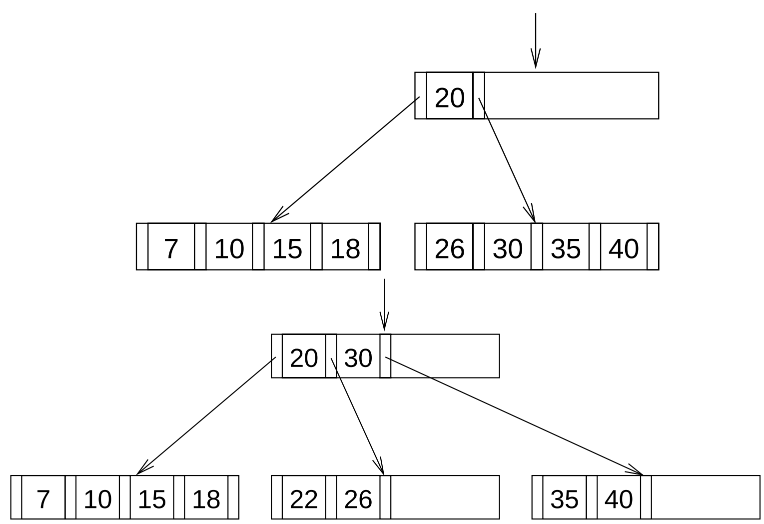
- entspricht genau der "Höhe" des Baumes

- beliebt für sehr große Datenbestände (mit großer Knotengröße):

  - Konkret : Seiten der Größe 4 KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger: zwischen 50 und 100 Indexeinträge pro Seite; Ordnung dieses B-Baumes 50
  - 1.000.000 Datensätze: $\log_{50}(1.000.000) = 4$ Seitenzugriffe im schlechtesten Fall
  - optimiert Anzahl von der Festplatte in den Hauptspeicher zu transferierenden Blöcke!

Beispiel für Einfügen und Löschen im B-Baum

Einfügen des Elementes 22; Löschen von 22



Praktischer Einsatz der vorgestellten Verfahren

AVL in Ausbildung und Lehrbüchern

B-Bäume "überall im Einsatz" (einfache Einfüge-Algorithmen; Knotengröße an Seitengröße von Hintergrundspeichern optimal anpaßbar)

9.5. Digitale Bäume

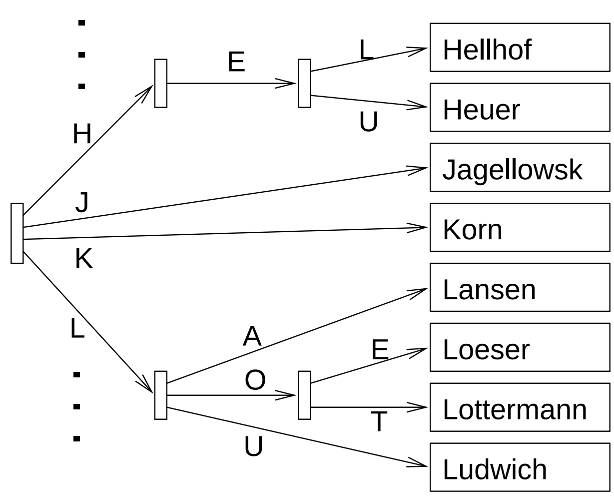
- digitale Bäume
  - spezielle (Mehrweg-) Suchbäume
  - feste Verzweigung *unabhängig* von gespeicherten Schlüsseln
  - nur für Datentypen bei denen eine derartige feste Verzweigung sinnvoll ist
    - ▷ insbesondere: Zeichenketten über festem Alphabet, Verzweigung nach jeweils erstem Buchstaben
- 'digital' von 'Finger' (10 Finger → Digitalbaum für numerische Zeichenketten)

Digital- und Präfixbäume

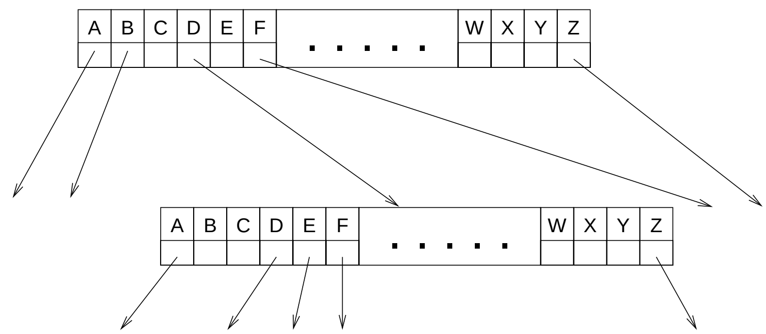
- B-Bäume: Probleme bei variabel langen Zeichenketten als Schlüssel
- Lösung: Digital- oder Präfixbäume
- Digitalbäume indexieren (fest) die Buchstaben des zugrundeliegenden Alphabets
  - können dafür unausgeglichen werden
  - Beispiele: Tries, Patricia-Bäume
- Präfixbäume indexieren Präfixe von Zeichenketten

Tries

- von "Information Retrieval", aber wie *try* gesprochen



Knoten eines Tries



- Probleme bei Tries: lange gemeinsame Teilworte, nicht vorhandenen Buchstaben und Buchstabenkombinationen, möglicherweise leere Knoten, sehr unausgeglichene Bäume

Patricia-Bäume

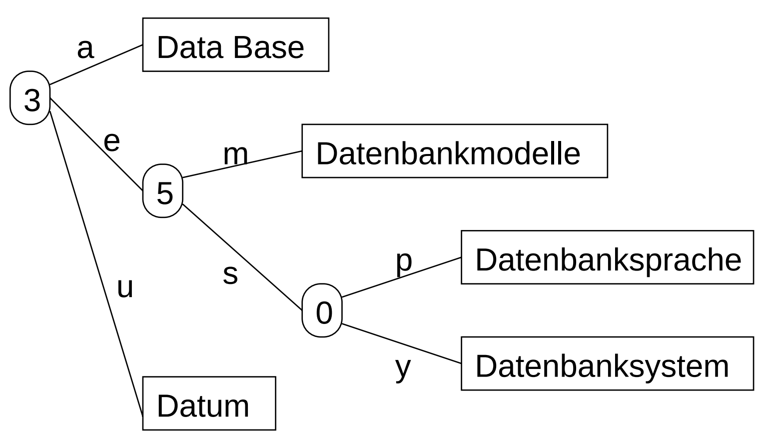
- Tries: Probleme beispielsweise bei künstlichen Schlüsseln (Teilekennzahlen), Pfadnamen, URLs (lange gemeinsame Teilworte), Bitfolgen

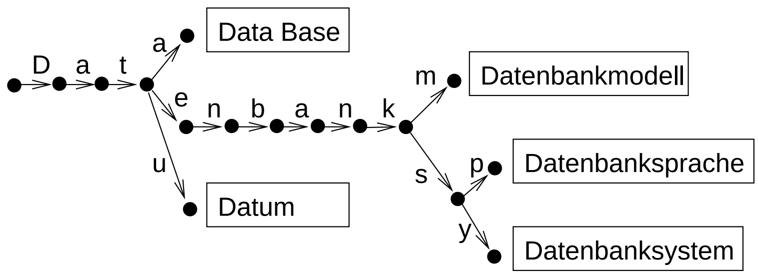
- Lösung: *Practical Algorithm To Retrieve Information Coded In Alphanumeric* (Patricia)

- Prinzip: Überspringen von Teilworten

- Problem (siehe Beispiel): Finden des Begriffs Datenbanksprache bei Suchbegriff Triebwerksperre

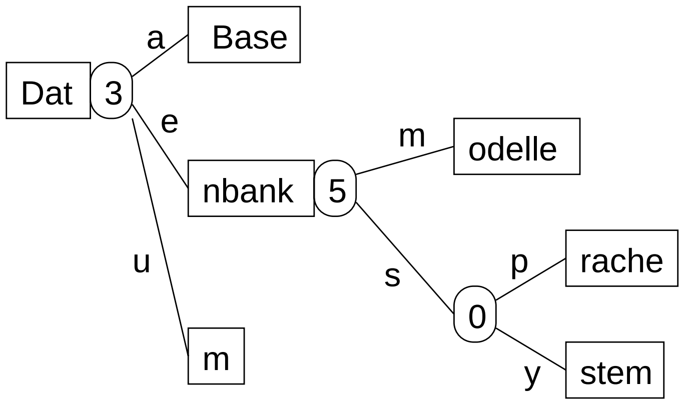
Patricia-Baum und Trie im Vergleich





Lösung des Suchproblems: übersprungene Teilworte zusätzlich speichern: Präfix-Bäume

Präfix-Bäume



Bemerkungen zu digitalen Bäumen

- nur bei Gleichverteilung ungefähr ausgeglichen (welcher Vorname beginnt mit dem Buchstaben 'Q'?)
- Einsatz insbesondere für Information Retrieval, Textindizierung (Suchmaschinen), Bitfolgen
- Alternative: 'Präfix-B-Bäume'
  - innere Knoten haben zur Verzweigung nur die notwendigen Präfixe als Schlüsseleinträge
  - echte Datensätze nur in den Blättern

10. Hash-Verfahren

Hash-Verfahren:

- Speicherung in Array 0 bis $N-1$, einzelne Positionen oft als ‘Buckets’ bezeichnet
- Hash-Funktion $h(e)$ bestimmt für Element e die Position im Array
- h sorgt für ‘gute’, kollisionsfreie bzw. kollisionsarme Verteilung der Elemente
- letzteres kann nur annäherungsweise durch Gleichverteilung erreicht werden, da Elemente vorher unbekannt!

Beispiel für Hashen

Beispiel: Array von 0 bis 9, $h(i) = i \mod 10$
Array nach Einfügen von 42 und 119:

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

Möglichkeit der *Kollision*: Versuch des Eintragens von 69

Beispiel für Hashen bei anderen Datentypen

Gegeben: Durchnumerierung der Buchstaben (A=1,B=2,C=3, ...)

- Hashfunktion für Vor- und Nachnamen:
  - Addiere Nummer des ersten Buchstabens des Vornamens mit der Nummer des ersten Buchstabens des Nachnamens

  - oder: Addiere Nummer des zweiten Buchstabens des Vornamens mit der Nummer des zweiten Buchstabens des Nachnamens
  - Hash-Wert ergibt sich dann modulo Feldgrösse, etwa $\mod 10$

- welche Hash-Funktion ist besser?

10.1. Grundlagen

Hash-Funktionen

- hängen vom Datentyp der Elemente und konkreter Anwendung ab
- für Integer oft
$$h(i) = i \mod N$$
(funktioniert gut wenn N eine Primzahl ist)
- für andere Datentypen: Rückführung auf Integer
  - Fließpunkt-Zahlen: Addiere Mantisse und Exponent
  - Strings: Addiere ASCII/Unicode-Werte der/einiger Buchstaben, evtl. jeweils mit Faktor gewichtet

Hash-Funktionen

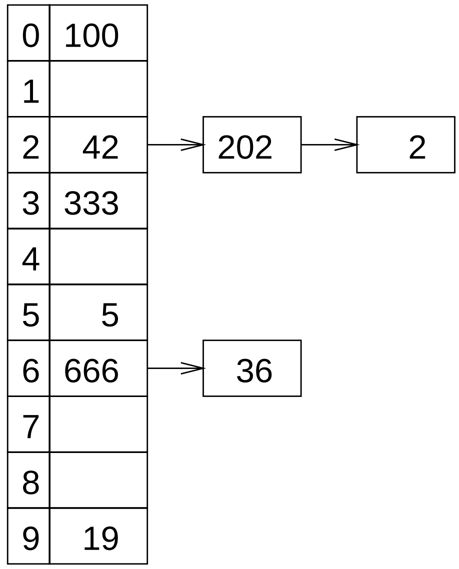
- sollen die Werte gut 'streuen'
- eventuell von Besonderheiten der Eingabewerte abhängig (Buchstaben des Alphabets in Namen tauchen unterschiedlich häufig auf!)
- sind effizient berechenbar (konstanter Zeitbedarf, *auf keinen Fall* abhängig von der Anzahl der gespeicherten Werte!)

10.2. Kollisionsstrategien

- Verkettung der Überläufer bei Kollisionen
- Sondieren (Suchen einer alternativen Position)
  - Lineares Sondieren
  - Quadratisches Sondieren
  - doppeltes Hashen, Sondieren mit Zufallszahlen, ...

Verkettung der Überläufer

Idee: alle Einträge einer Array-Position werden in einer verketteten Liste verwaltet



Lineares Sondieren

Idee: falls $T[h(e)]$ besetzt ist, versuche

$$T[h(e)+1], T[h(e)+2], T[h(e)+3], ...T[h(e)+i], ...$$

Varianten:

- $T[h(e)+c], T[h(e)+2c], T[h(e)+3c], ...$ für Konstante c
- $T[h(e)+1], T[h(e)-1], T[h(e)+2], T[h(e)-2], ...$

Lineares Sondieren II

	leer	insert 89	insert 18	insert 49	insert 58	insert 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratisches Sondieren

Idee: falls $T[h(e)]$ besetzt ist, versuche

$$T[h(e)+1], T[h(e)+4], T[h(e)+9], .., T[h(e)+i^2]...$$

Variante:

- $T[h(e)+1], T[h(e)-1], T[h(e)+4], T[h(e)-4], ...$

anfällig gegenüber falsch gewähltem N, Beispiel $N = 16 \rightarrow$ wähle Primzahlen für N

Quadratisches Sondieren II

	leer	insert 89	insert 18	insert 49	insert 58	insert 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Aufwand beim Hashen

- bei geringer Kollisionswahrscheinlichkeit:
  - Suchen in $O(1)$
  - Einfügen in $O(1)$
  - Löschen bei Sondierverfahren: nur Markieren der Einträge als gelöscht $O(1)$, oder ‘Rehashen’ der gesamten Tabelle notwendig $O(n)$
- Füllgrad über 80 % : Einfüge- / Such-Verhalten wird schnell dramatisch schlechter aufgrund von Kollisionen (bei Sondieren)

Aufwand beim Hashen: verkettete Überläufer

- Füllgrad (load factor) $\alpha = n/m$
  - n Anzahl gespeicherter Elemente
  - m Anzahl Buckets
- erfolglose Suche (Hashing mit Überlaufliste): $\Theta(1+\alpha)$
- erfolgreiche Suche ebenfalls in dieser Größenordnung

Aufwand beim Hashen: Sondieren

- Aufwand beim Suchen:
$$\frac{1}{1-\alpha}$$
- typische Werte:

  $\frac{1}{1-0.5} = 2$

  $\frac{1}{1-0.9} = 10$
- exakte Analyse basiert auf Aufsummierung der Wahrscheinlichkeiten, daß i Elemente das Bucket i belegen (bei Gleichverteilung)

Aufwand beim Hashen: Sondieren detailliert

- Wahrscheinlichkeiten basieren auf Füllgrad α
  - Bucket belegt: Wahrscheinlichkeit α
  - Bucket belegt und sondiertes Bucket auch belegt: ungefähr α^2
  - etc.

$$1 + \alpha + \alpha^2 + \alpha^3 + ... = \frac{1}{1-\alpha}$$

- erfolgreiche Suche besser, aber gleiche Größenordnung

10.3. Dynamische Hash-Verfahren

Probleme bei Hash-Verfahren:

- mangelnde Dynamik
- Vergrößerung des Bildbereichs erfordert komplettes Neu-Hashen
- Wahl der Hash-Funktion entscheidend; Bsp.: Hash-Tabelle mit 100 Buckets, Studenten über 6-stellige `MATRNR` (wird fortlaufend vergeben) hashen
  - ersten beiden Stellen: Datensätze auf wenigen Seiten quasi sequentiell abgespeichert
  - letzten beiden Stellen: verteilen die Datensätze gleichmäßig auf alle Seiten

dynamische Hash-Verfahren versuchen diese Probleme zu lösen

Motivation für die Behandlung dynamischen Hashens

dynamische Hash-Verfahren zeigen, wie verschiedene der bisher gezeigten Techniken sinnvoll kombiniert werden können

- unschlagbar gute Komplexität von Hashen beim Suchen
- dynamisches Wachstum ausgeglichener Baumstrukturen
- Implementierung von konzeptionellen Strukturen durch Basisdatenstrukturen

dynamische Hash-Verfahren gehören zu den fortgeschrittenen und ausgefeilteren Verfahren, und sind daher nicht in allen Lehrbüchern zu Datenstrukturen zu finden!

Erste Idee: Hash-Funktion mit dynamisch erweiterbaren Bildbereich

- Zielbereich der Hashfunktionen: Das Intervall $[0 \ldots 1)$
- Hashwerte werden binär notiert:

$$\begin{aligned} 0.0 &= b0.000000... \\ 0.5 &= b0.100000... \\ 0.75 &= b0.110000... \\ 0.99999... &= b0.111111... \end{aligned}$$

- erste i Bits hinter dem Punkt adressieren ein Array der Größe 2^i

$\rightarrow$ dynamisch erweiterbarer Bildbereich (... durch Verdoppeln)

Dynamisch erweiterbare Bildbereiche

Alternative mathematische Charakterisierung:

- $h_i : dom(\text{Wert}) \rightarrow \{0, \ldots, 2^i \times N - 1\}$ ist eine Folge von Hash-Funktionen mit $i \in \{0, 1, 2, \ldots\}$ und N als Anfangsgröße des Hash-Verzeichnisses. Der Wert von i wird als *Level* der Hash-Funktion bezeichnet.
- Für diese Hash-Funktionen gelten die folgenden Bedingungen:
  - $h_{i+1}(w) = h_i(w)$ für etwa die Hälfte aller $w \in dom(Prim)$
  - $h_{i+1}(w) = h_i(w) + 2^i \times N$ für die andere Hälfte

  Diese Bedingungen sind zum Beispiel erfüllt, wenn $h_i(w)$ als $w \mod (2^i \times N)$ gewählt wird.

Implementierung erweiterbarer Bildbereiche

übliche Realisierung:

- Hash-Werte sind Bit-Folgen (potentiell beliebig lang)
- beim Level i werden die ersten i Bits als Zahl interpretiert
  - erste Variante: Binärzahl in üblicher Schreibweise
  - zweite Variante: Bit-Folge in umgekehrter Reihenfolge interpretiert

erweiterbares Hashen wie unten beschrieben nutzt die erste Variante

Zweite Idee: Zielbereich kann mehr als einen Wert aufnehmen

Hash-Funktion liefert die Adresse eines *Blockes*
Block kann vorgegebene Anzahl von Elementen aufnehmen
... trotzdem Überlauf:

- Erweiterung des Bildbereichs ('ein Bit mehr')
- oder (übergangsweise) Überlaufliste verwalten

Dritte Idee: Erweiterung des Bildbereichs nur dort wo nötig

Beispiel: mehr Elemente zwischen 0 und 0.5 als zwischen 0.5 und 1:

wähle für Hash-Werte die (als Binärzahl) mit 0.0 beginnen weniger Bits zur Adressierung als für die mit 0.1 beginnenden

Vorsicht: jetzt kann man nicht mehr direkt ein Array adressieren, da unterschiedllich lange Bit-Folgen zur Adressierung genommen werden (Adressierung über 11 und 011 liefert in beiden Fällen die 3!)

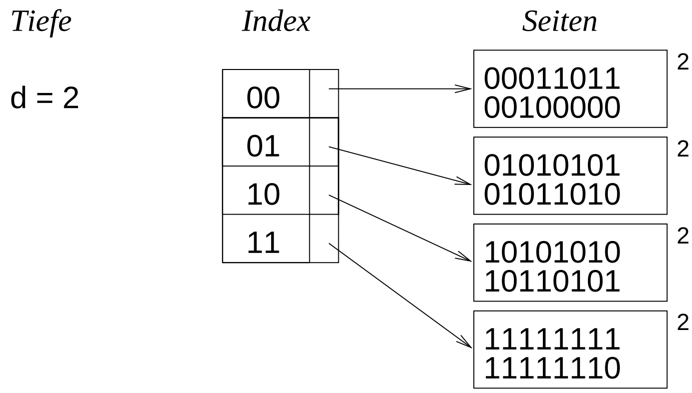
Vierte Idee: Binärer Trie zur Verwaltung der Blöcke

unterschiedliche lange Bit-Sequenzen können effizient über einen Trie verwaltet werden
Beobachtung: bei guter Hash-Funktion ist der Trie so gut wie ausgeglichen!

Fünfte Idee: Ausgeglichener Trie als Array gespeichert

Als Suchstruktur wird statt eines allgemeinen binären Präfix-Baum ein *ausgeglichener* binärer Präfix-Baum genommen, der als Array mit fester Größe 2^d realisiert wird.
d wird dabei als *Tiefe* des Baumes bezeichnet.

Erweiterbares Hashen: Beispiel der Tiefe $d = 2$



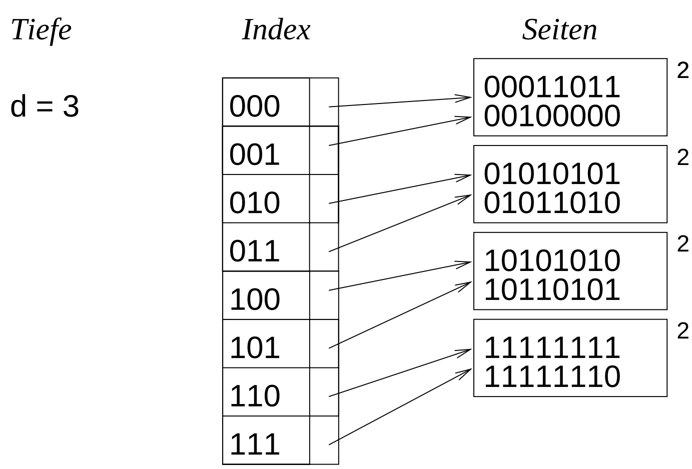
Verdopplung des Index-Arrays

Würde im Beispiel ein weiterer Wert eingefügt werden, müßte einer der Blöcke gesplittet werden.

Da der Index ein ausgeglichener Baum ist, muß hierfür die *Tiefe um eins erhöht werden*. Das Array verdoppelt damit seine Größe.

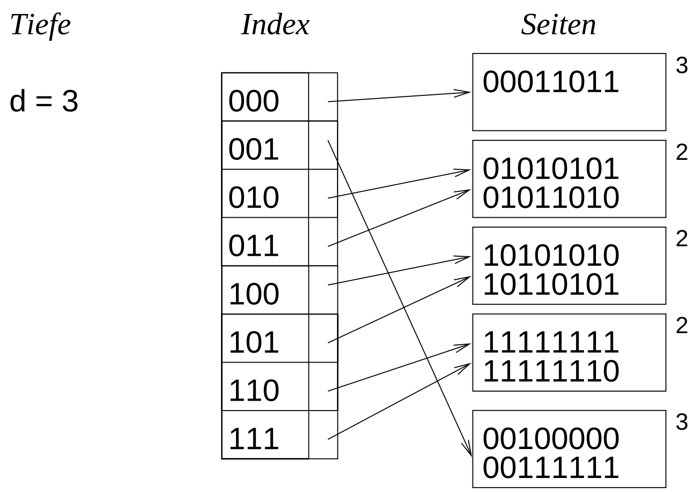
Um nun zu vermeiden, daß auch die Buckets verdoppelt werden, können Blätter gemeinsam auf den selben Bucket zeigen. Beim Verdoppeln des Index-Arrays geht man nun wie folgt vor: Ist einem Blatt x vor dem Verdoppeln das Bucket s zugeordnet, zeigen nach dem Verdoppeln die beiden Söhne von x, also $x0$ und $x1$, ebenfalls beide auf s.

Erweiterbares Hashen: Array-Größe verdoppelt



Erweiterbares Hashen: Aufteilen eines Blockes

nach Einfügen von 00111111:



Erweiterbares Hashen: Eigenschaften

- Bit-Folgen als Hash-Werte: Im Rechner effizient verarbeitbar
- Suchen in $O(1)$ bei guter Hash-Funktion
- Einfügen kann Verdoppeln des Index-Arrays bedeuten, aber *nicht* Umkopieren aller Datenblöcke!
- daher bei nicht völlig gleichmäßig verteilten Hash-Werten: Überlauflisten bei geringem Füllgrad statt sofortigem Verdoppeln des Index-Arrays

Zyklisches Verhalten

- kurz vor dem Erhöhen der Tiefe ist bei tatsächlicher Gleichverteilung der Hash-Werte die Speicherauslastung am besten

→ die Wahrscheinlichkeit, daß ein Einfügen das Splitten einer Seite bedeutet, ist am größten

- kann durch Umverteilen mittels $2^{h(k)} - 1$ gemildert werden

n	$2^n - 1$
0.0	0.0
0.1	0.0717735
0.2	0.1486984
0.3	0.2311444
0.4	0.3195079
0.5	0.4142136
0.6	0.5157166
0.7	0.6245048
0.8	0.7411011
0.9	0.866066
1.0	1.0

Lektionen aus erweiterbaren Hashen

Eigenschaften von Datenstrukturen sind nicht unumstößlich

- “Hash-Funktionen sind statisch”
- “ausgeglichene Suchbäume ermöglichen bestenfalls $O(\log n)$ Zugriff”
- “digitale Bäume sind nicht ausgeglichen”

... erweiterbares Hashen kombiniert die Ideen der drei Ansätze derart, daß die jeweiligen Nachteile wegfallen!

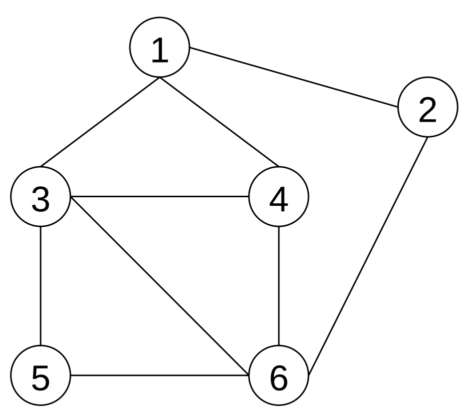
11. Graphen

- Arten von Graphen
- Implementierungsvarianten für Graphen
- Graph-Algorithmen

11.1. Arten von Graphen

- ungerichtete Graphenungerichteter Graph (oft einfach: Graphen)
  Straßenverbindungen, Nachbarschaft, Telefonnetz
- gerichtete Graphen
  Förderanlagen, Kontrollfluß in Programmen, Petri-Netze
- gerichtete azyklische Graphen (DAG für directed acyclic graph)
  Vorfahrenbeziehung, Vererbung in Programmiersprachen

Ungerichteter Graph



Graph G_u

Definition von (ungerichteten) Graphen

Graphen $G = (V, E)$

- V endliche Menge von Knoten (vertices oder nodes)
- E Menge von Kanten (edges)
  $e \in E$ ist zweielementige Teilmenge von V

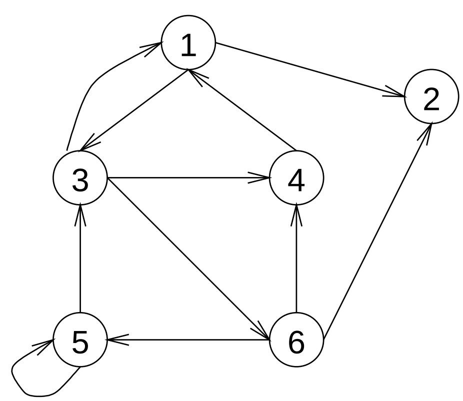
Bemerkung: Diese Definition läßt keine *Schleifen*, also Kanten von einem Knoten zu sich selber, zu. Erweiterte Definition möglich: ... *ein- oder zweielementige Teilmenge ...*

Beispiel für Graph

- $G_u = (V_u, E_u)$
- $V_u = \{1, 2, 3, 4, 5, 6\}$
- $E_u = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 6\}, \{4, 6\}, \{3, 6\}, \{5, 6\}, \{3, 4\}, \{3, 5\}\}$

Graph G_u von Folie 199

Gerichtete Graphen



Graph G_g

Definition von gerichteten Graphen

Graphen $G = (V, E)$

- V endliche Menge von Knoten (vertices oder nodes)
- E Menge von Kanten (edges)
  $e \in E$ ist Tupel (a, b) mit $a, b \in V$

Diese Definition läßt Schleifen (a, a) zu.

Beispiel für gerichteten Graph

- $G_g = (V_g, E_g)$
- $V_g = \{1, 2, 3, 4, 5, 6\}$
- $E_g = \{(1, 2), (1, 3), (3, 1), (4, 1), (3, 4), (3, 6), (5, 3), (5, 5), (6, 5), (6, 2), (6, 4)\}$

Graph G_g von Folie 200

Nützliche Informationen

- Ausgangsgrad eines Knotens $ag(i)$: Zahl der von diesem Knoten ausgehenden Kanten
- Eingangsgrad eines Knotens $eg(i)$: Zahl der zu diesem Knoten führenden Kanten

Datentyp für Graphen

- `empty`: leerer Graph $G = (\emptyset, \emptyset)$
- `insertNode(n,G)`: Einfügen eines Knotens
- `connect(n,m,G)`: Ziehen einer Kante
- `deleteNode(n,G)`: Löschen eines Knotens
- `disconnect(n,m,G)`: Entfernen einer Kante
- `isInNode?(n,G)`, `isInEdge?(n,m,G)` etc.

Gewichtete Graphen

- *Kantengewichte*: den Kanten zugeordnete Gewichte (etwa `int`- oder `real`-Werte)
- Graph nun z.B. $G = (V, E, \gamma)$ mit

$$\gamma\colon E \to \mathbb{N}$$

- neue Fragestellungen: kürzeste Wege, maximaler Fluß, minimaler aufspannender Baum

11.2. Realisierung von Graphen

- Konzentration auf gerichtete Graphen (ungerichtete Graphen oft analog)
- verschiedene Realisierungsvarianten
- Vergleich der Komplexität bei Operationen

Knoten- und Kantenlisten

Speicherung von Graphen als Liste von Zahlen (etwa in Array, oder verkettete Liste)

- Liste von Zahlen als Codierung des Graphen
- Knoten von 1 bis n durchnumeriert, Kanten als Paare von Knoten
- Varianten:
  - Kantenlisten
  - Knotenlisten

Kantenlisten

Liste: Knotenzahl, Kantenzahl, Liste von Kanten (je als zwei Zahlen)
Beispiel G_g von Folie 200:

$$6, 9, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4$$

Knotenlisten

- Liste: Knotenzahl, Kantenzahl, Liste von Knoteninformationen
- Knoteninformation: Ausgangsgrad und Zielknoten: $ag(i), v_{i_1}, .., v_{i_{ag(i)}}$

Beispiel G_g von Folie 200:

$$6, 9, 2, 2, 3, 0, 3, 1, 4, 6, 1, 1, 2, 3, 5, 3, 2, 4, 5$$

Teilfolge $2, 2, 3$ bedeutet “Knoten 1 hat Ausgangsgrad 2 und herausgehende Kanten zu den Knoten 2 und 3”

Adjazenzmatrix

Verbindungen als $n \times n$-Matrix

$$G_g = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Adjazenzmatrix für ungerichteten Graph

$$G_u = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

ungerichteter Graph von Folie 199

man beachte: gespiegelt an der Diagonalen, Diagonale selber ist leer $\rightarrow$ optimierte Speicherung möglich!

Graphen als dynamische Datenstrukturen

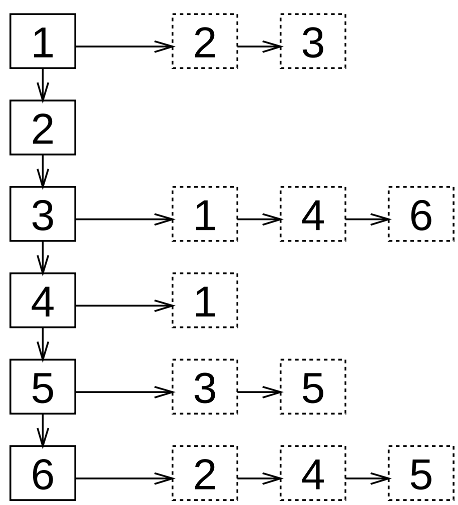
auch: *Adjazenzliste*

Realisierung durch $n + 1$ verkettete Listen

- Liste aller Knoten
- für jeden Knoten: Liste der Nachfolger

also $n + \sum_{i=1}^{n} ag(i) = n + m$ Listenelemente

Graphen als dynamische Datenstrukturen II



Transformationen zwischen Darstellungen

- im Wesentlichen:
  - Auslesen der einen Darstellung
  - Erzeugen der anderen Darstellung

- Aufwand variiert von $O(n+m)$ bis $O(n^2)$
  - im schlechtesten Fall gilt $m = n^2$
  - n^2 falls Matrixdarstellung beteiligt (da n^2 Elemente berücksichtigt werden müssen)

Vergleich der Komplexität

Operation	Kanten-liste	Knoten-liste	Adjazenz-matrix	Adjazenz-liste
Einfügen Kante	$O(1)$	$O(n+m)$	$O(1)$	$O(1)$ / $O(n)$
Löschen Kante	$O(m)$	$O(n+m)$	$O(1)$	$O(n)$
Einfügen Knoten	$O(1)$	$O(1)$	$O(n^2)$	$O(1)$
Löschen Knoten	$O(m)$	$O(n+m)$	$O(n^2)$	$O(n+m)$

Bemerkungen: Löschen eines Knotens löscht auch zugehörige Kanten, Aufwand bei Adjazenzmatrix bei Knoteneinfügung hängt von der Realisierung der Matrix ab, bei Adjazenzliste untercshiedlicher Aufwand falls Knotenliste als Array oder als verkettet Liste realisiert wird

11.3. Ausgewählte Graphenalgorithmen

- Breitendurchlauf
- Tiefendurchlauf
- Zyklenfreiheit
- Topologisches Sortieren

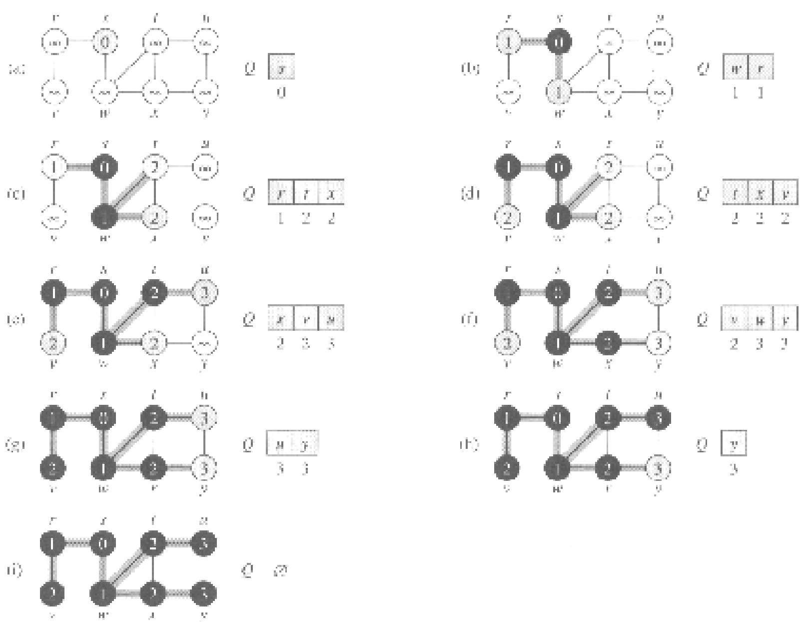
Breitendurchlauf

```
BFS(G,s):
for each Knoten u ∈ V[G] − s
   do { farbe[u]= weiß; d[u] = ∞; π[u] = null };
farbe[s] = grau; d[s] = 0; π[s] = null;
Q = emptyQueue; Q = enqueue(Q,s);
while not isEmpty(Q) do {
   u = front(Q)
   for each v ∈ ZielknotenAusgehenderKanten(u) do {
      if farbe(v) = weiß then
         { farbe[v] = grau; d[v] = d[u]+1;
           π[v] = u; Q = enqueue(Q,v); };
   };
   dequeue(Q); farbe[u] = schwarz;
}
```

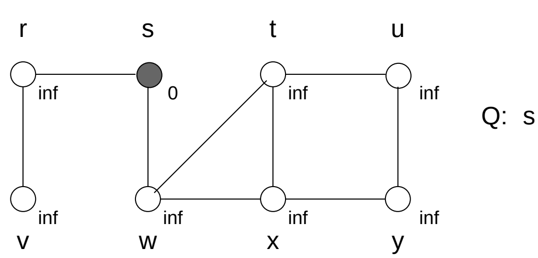
Breitendurchlauf: Erläuterungen

- BFS: breadth-first-search
- Hilfsinformation pro Knoten:
  - Farbwert: Bearbeitungsstatus
  - Distanz zum Startknoten d
  - Vorgänger (predecessor) π
- Warteschlange Q als Hilfsstruktur

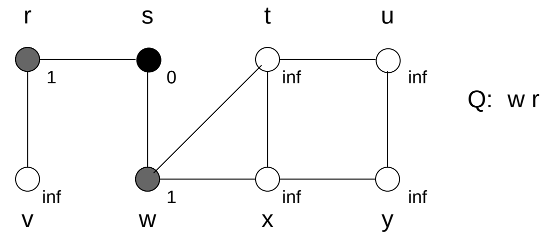
Breitendurchlauf: Beispiel



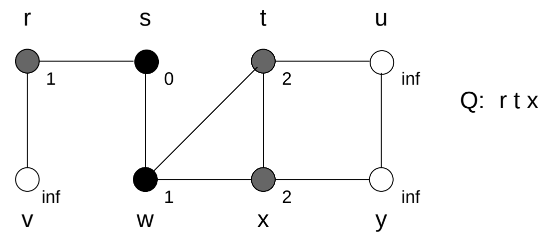
Breitendurchlauf: Schritt 1



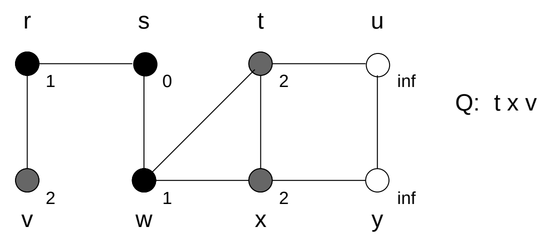
Breitendurchlauf: Schritt 2



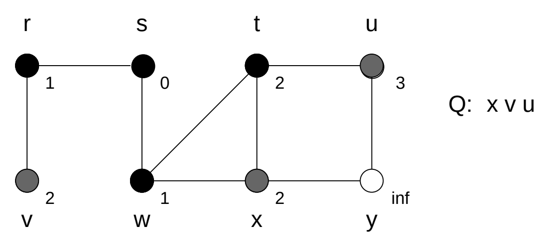
Breitendurchlauf: Schritt 3



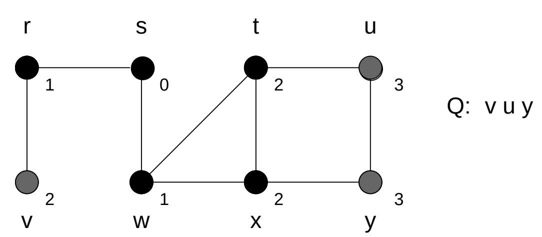
Breitendurchlauf: Schritt 4



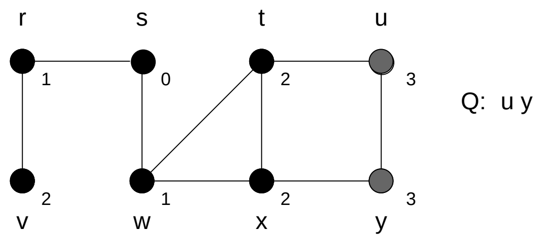
Breitendurchlauf: Schritt 5



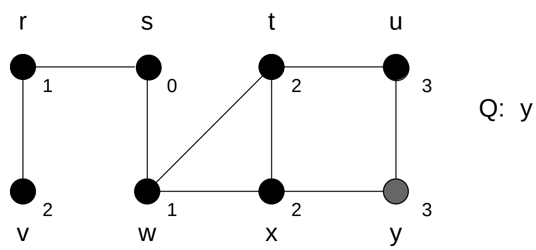
Breitendurchlauf: Schritt 6



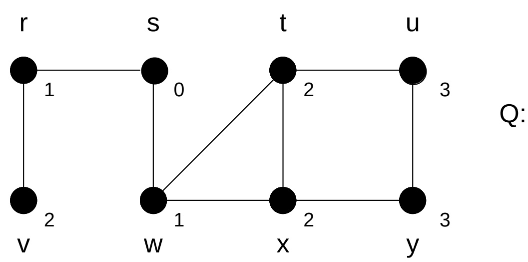
Breitendurchlauf: Schritt 7



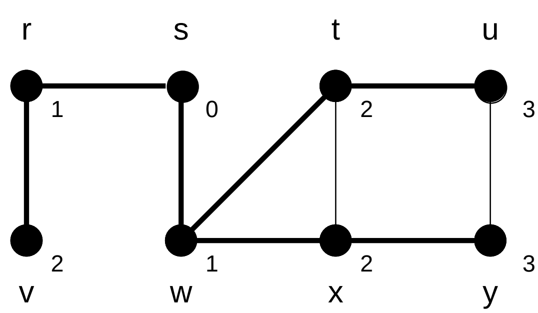
Breitendurchlauf: Schritt 8



Breitendurchlauf: Schritt 9



Breitendurchlauf: Erzeugter aufspannender Baum



Quelle der Beispiele

Thomas Cormann, Charles Leiserson, Ronald Rivest: Introduction to Algorithms, McGraw-Hill, 0-07-013143-0

Tiefendurchlauf

DFS: depth-first search

- Idee: rekursiver Durchlauf durch gerichteten Graphen
- Farbmarkierungen für den Bearbeitungsstatus eines Knotens
  - weiß: noch nicht bearbeitet
  - grau: in Bearbeitung
  - schwarz: fertig

Tiefendurchlauf DFS

```
DFS(G):
for each Knoten u ∈ V[G]
   do { farbe[u]= weiß; π[u] = null };
zeit = 0;
for each Knoten u ∈ V[G]
   do { if farbe[u]= weiß then DFS-visit(u) };

DFS-visit(u):
farbe[u]= grau; zeit = zeit+1; d[u]=zeit;
for each v ∈ ZielknotenAusgehenderKanten(u) do {
   if farbe(v) = weiß then
      { π[v] = u; DFS-visit(v); };
};
farbe[u] = schwarz; zeit = zeit+1; f[u]=zeit;
```

Tiefendurchlauf DFS (Erläuterungen)

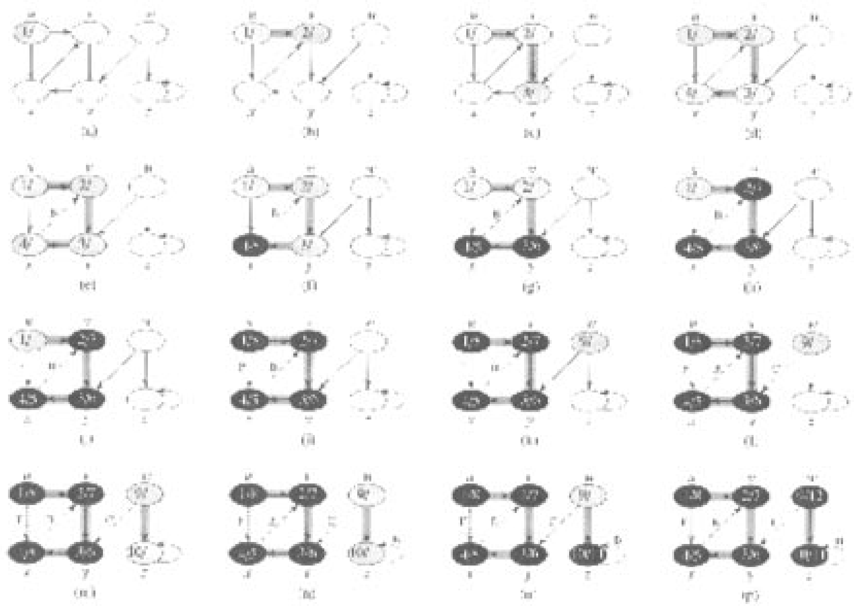
- Hilfsinformation pro Knoten:
  - Farbwert: Bearbeitungsstatus
  - d: Beginn der Bearbeitung des Knotens
  - f: Ende der Bearbeitung des Knotens

Tiefendurchlauf DFS (Erweiterung)

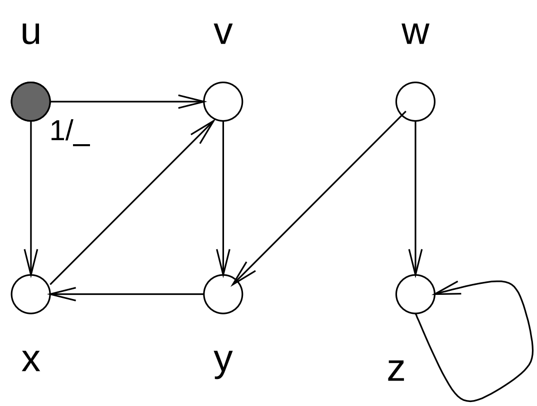
- Klassifikation der Kanten (u, v) während der Abarbeitung möglich:
  - Kanten des aufspannenden Baumes (Zielknoten weiß)
  - B: beim Test grau: Back-Edge / Rück-Kante im aufgespannten Baum (Zyklus)

- F: beim Test schwarz, paßt ins Intervall: Forward-Edge / Vorwärts-Kante in den aufgespannten Baum
- C: beim Test schwarz, paßt nicht ins Intervall ($d[u] > d[v]$): Cross-Edge (verbindet zwei aufspannende Bäume)

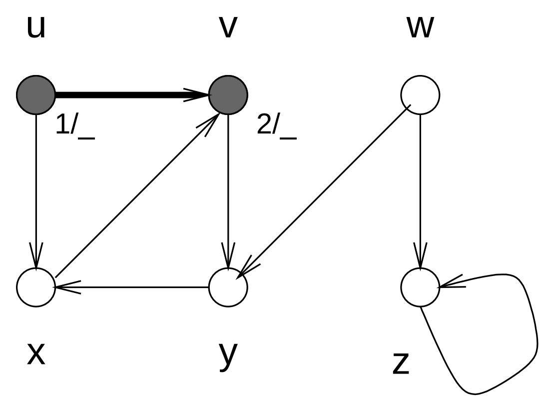
Tiefendurchlauf: Beispiel



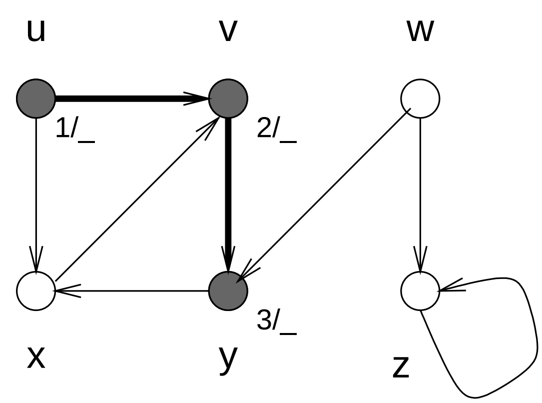
Tiefendurchlauf: Schritt 1



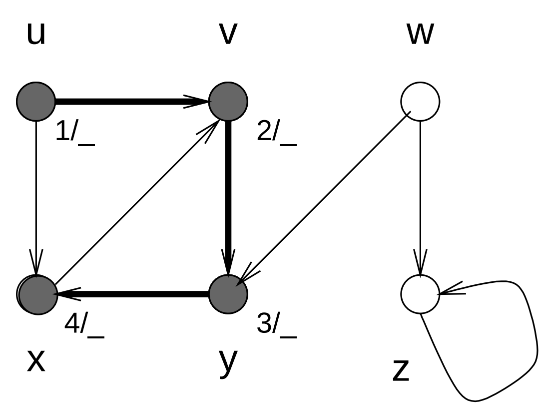
Tiefendurchlauf: Schritt 2



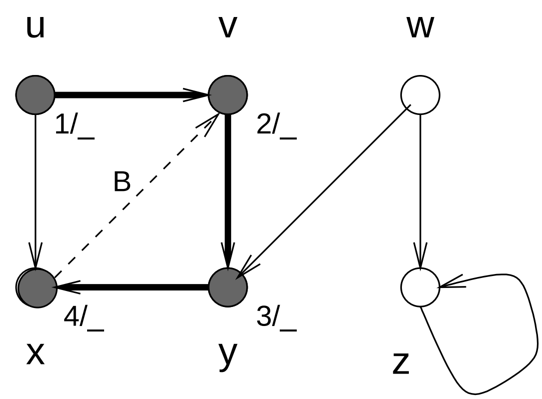
Tiefendurchlauf: Schritt 3



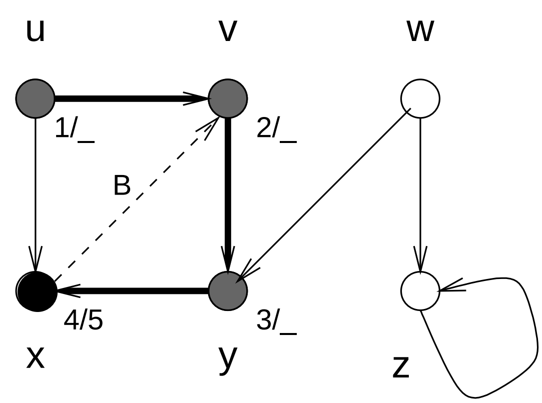
Tiefendurchlauf: Schritt 4



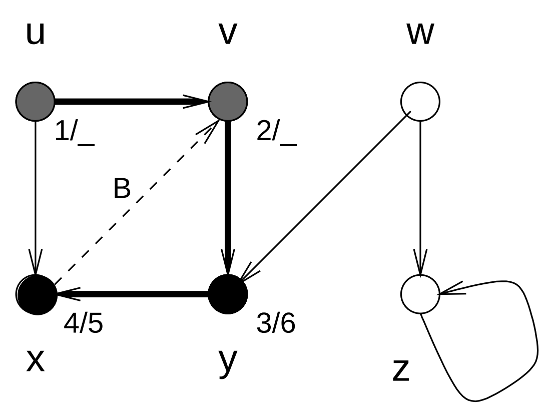
Tiefendurchlauf: Schritt 5



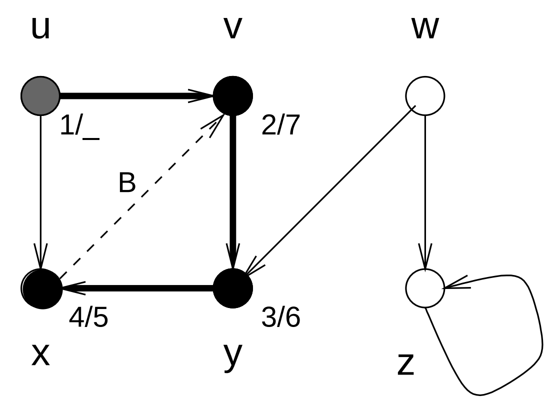
Tiefendurchlauf: Schritt 6



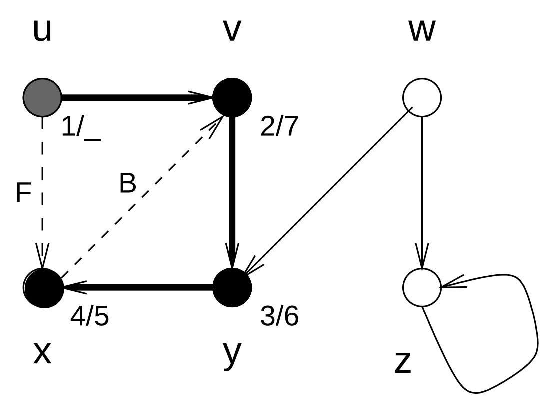
Tiefendurchlauf: Schritt 7



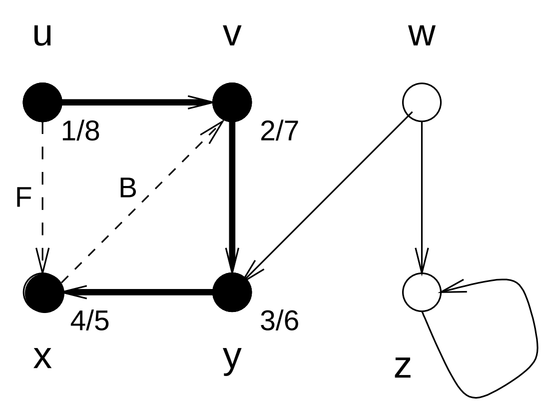
Tiefendurchlauf: Schritt 8



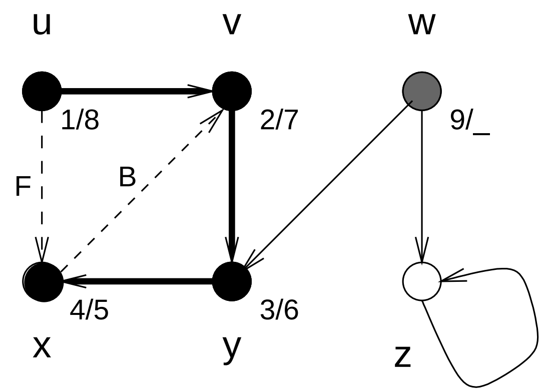
Tiefendurchlauf: Schritt 9



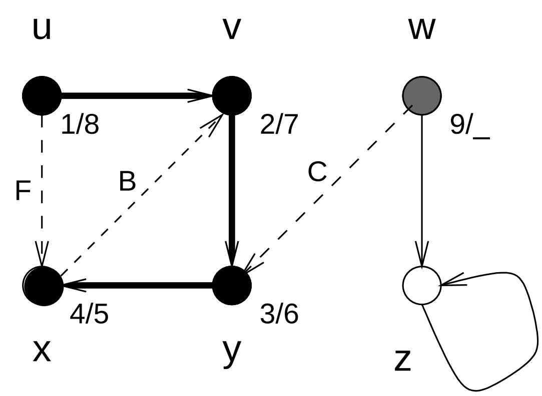
Tiefendurchlauf: Schritt 10



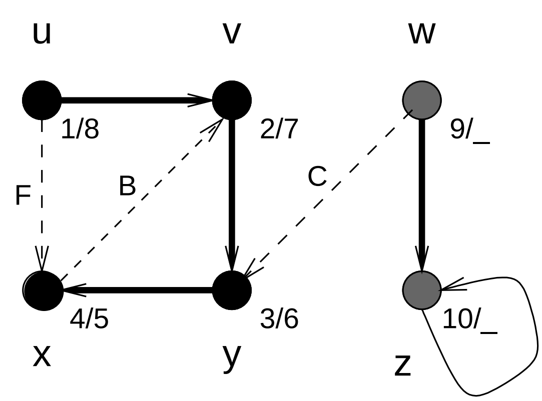
Tiefendurchlauf: Schritt 11



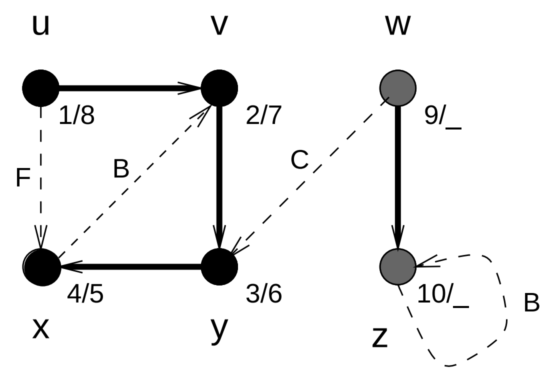
Tiefendurchlauf: Schritt 12



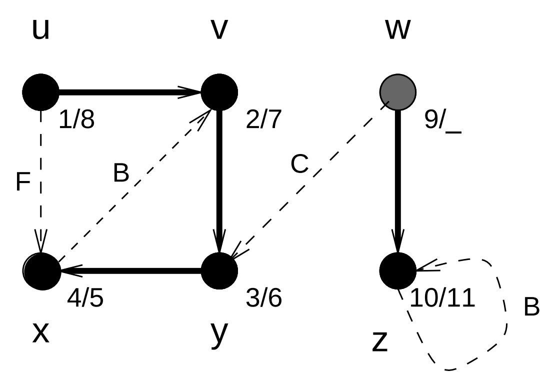
Tiefendurchlauf: Schritt 13



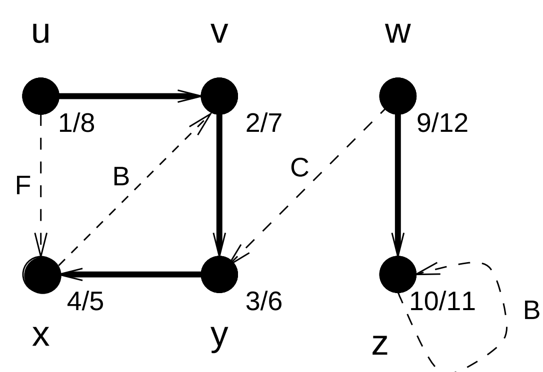
Tiefendurchlauf: Schritt 14



Tiefendurchlauf: Schritt 15



Tiefendurchlauf: Schritt 16



Zyklenfreiheit

- lasse DFS laufen
- G ist zyklenfrei $\iff$ keine Kante als B markiert

Topologisches Sortieren

Problem:

- Gegeben eine azyklischer gerichteter Graph
- finde Reihenfolge der Knoten, so daß jeder Knoten nach all seinen Vorgängern kommt
- auch: Scheduling-Problem

Topologisches Sortieren II

Topological-Sort:

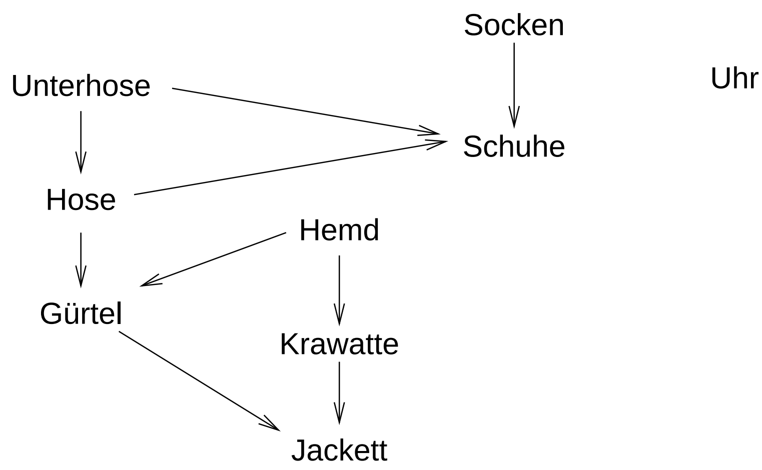
- lasse DFS(G) laufen um $f[v]$ für jeden Knoten v zu bestimmen
- $f[v]$ gibt eine topologische Sortierung vor
- daher:
  - bei jeder Abarbeitung eines Knotens wird er vorne in eine verkettet Liste eingehängt
  - die Liste gibt die Ordnung an

Beispiel für topologisches Sortieren

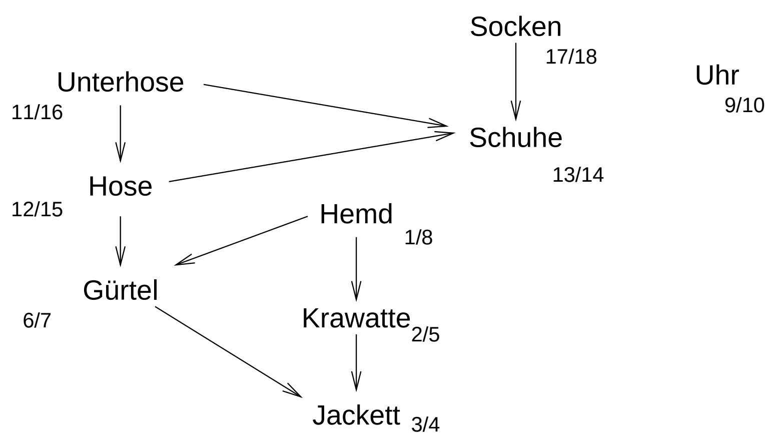
zerstreuter Professor legt die Reihenfolge beim Ankleiden fest:

```
Unterhose vor Hose
Hose vor Gürtel
Hemd vor Gürtel
Gürtel vor Jackett
Hemd vor Krawatte
Krawatte vor Jackett
Socken vor Schuhen
Unterhose vor Schuhen
Hose vor Schuhen
Uhr:  egal
```

Topologisches Sortieren: Eingabe



Topologisches Sortieren: nach DFS



Topologisches Sortieren: Ergebnis

f + Wert:

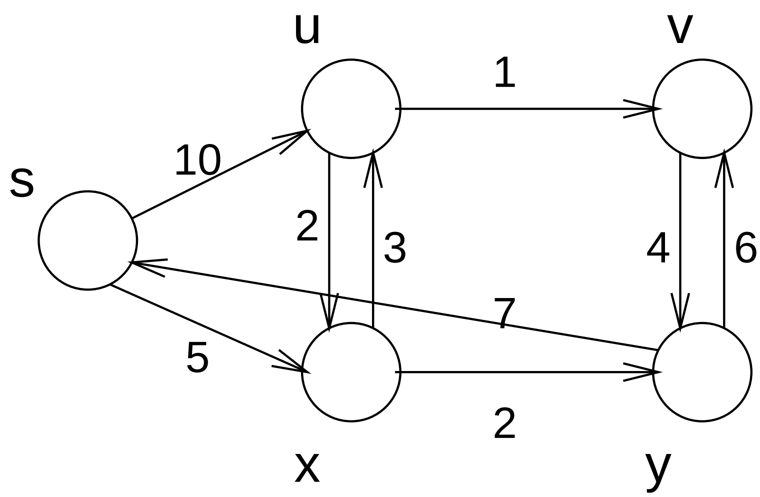
```
18 Socken
16 Unterhose
15 Hose
14 Schuhe
10 Uhr
8 Hemd
7 Gürtel
5 Krawatte
4 Jackett
```

11.4. Algorithmen auf gewichteten Graphen

gewichtete Graphen:

- ungerichtete gewichtet Graphen:
  z.B. Flugverbindungen mit Meilen / Preis, Straßen mit Kilometerangaben
- gerichtete gewichtete Graphen

Gewichteter Beispielgraph



Kürzeste Wege

- $G = (V, E, \gamma)$
- Weg $P = \langle (v_1, v_2), (v_2, v_3), (v_3, v_4), ..., (v_{n-1}, v_n) \rangle$ (P für path)
- Gewicht / Länge eines Pfades:

$$w(P) = \sum_{i=1}^{n-1} \gamma((v_i, v_{i+1}))$$

- Distanz $d(u, v)$: Gewicht des kürzesten Pfades von u nach v

Kürzeste Wege: Bemerkungen

- kürzeste Wege sind nicht eindeutig
- kürzeste Wege müssen nicht existieren
  - es existiert gar kein Weg
  - ein Zyklus mit negativem Gewicht existiert (und kann beliebig oft durchlaufen werden)

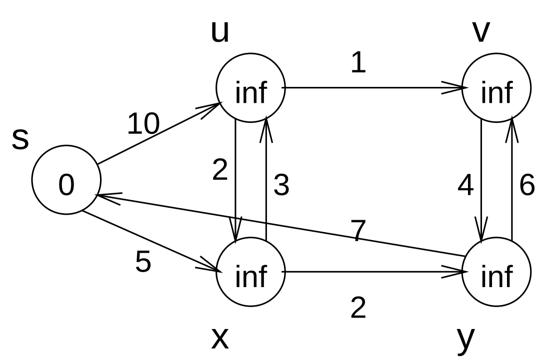
Dijkstra's Algorithmus

```
Dijkstra(G,s):
for each Knoten u ∈ V[G] − s do { D[u] = ∞; };
D[s] = 0; PriorityQueue Q = V;
while not isEmpty(Q) do {
   u = extractMinimal(Q)
   for each v ∈ ZielknotenAusgehenderKanten(u) ∩ Q do
   { if D[u] + γ((u,v)) < D[v] then
     {   D[v] = D[u] + γ((u,v));
         adjustiere Q an neuen Wert D[v] ; };
   };
}
```

Dijkstra's Algorithmus: Erläuterungen

- funktioniert nur für nichtnegative Gewichte
- arbeitet nach Greedy-Prinzip
- Prioritätswarteschlangen ermöglichen das Herauslesen des *minimalen* Elements; entsprechen sortierten Listen
- Algorithmus berechnet die Distanz aller Knoten zum Startknoten
- entspricht einer Breitensuche mit gewichteter Entfernung

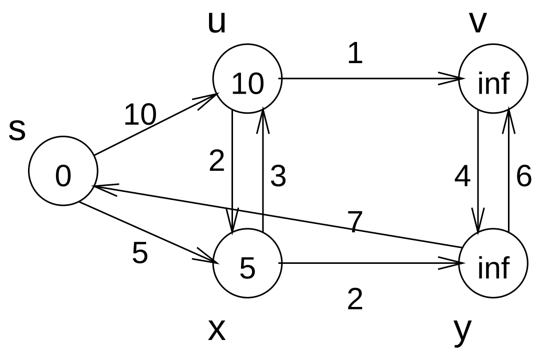
Dijkstra's Algorithmus: Initialisierung



$$Q = \langle (s:0), (u:\infty), (v:\infty), (x:\infty), (y:\infty) \rangle$$

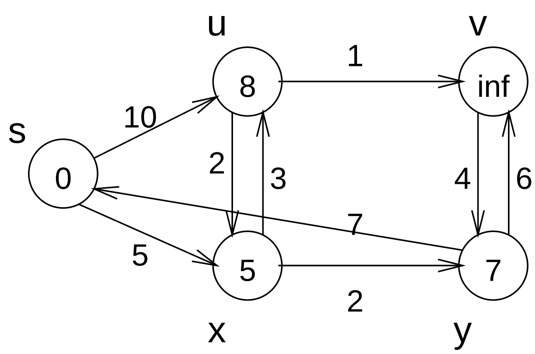
Bemerkung: ∞ in Abbildung als `inf` notiert

Dijkstra's Algorithmus: Schritt 1



$$Q = \langle (x:5), (u:10), (v:\infty), (y:\infty) \rangle$$

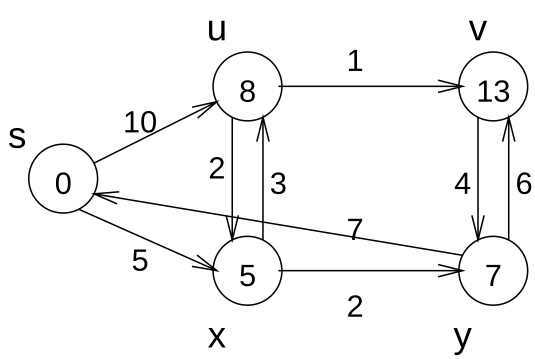
Dijkstra's Algorithmus: Schritt 2



$$Q = \langle (y, 7), (u:8), (v:\infty) \rangle$$

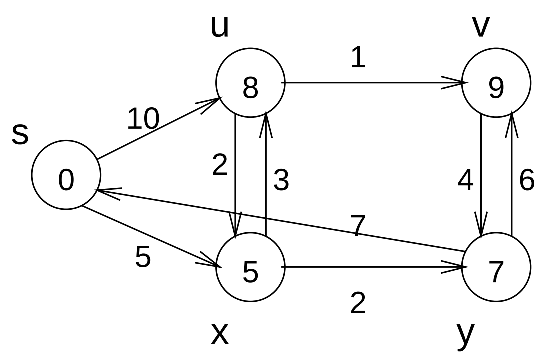
man beachte Anpassung von $D[u]$!

Dijkstra's Algorithmus: Schritt 3



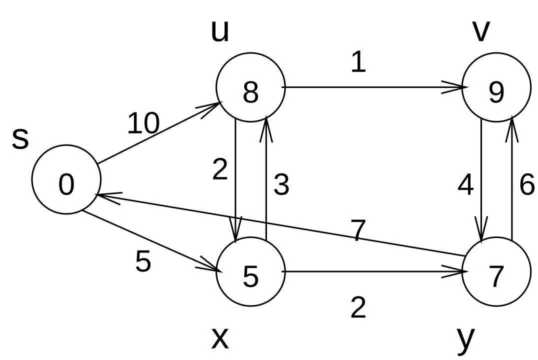
$$Q = \langle (u:8), (v:13) \rangle$$

Dijkstra's Algorithmus: Schritt 4



$$Q = \langle (v:9) \rangle$$

Dijkstra's Algorithmus: Ergebnis



$$Q = \langle \rangle$$

Dijkstra's Algorithmus: Warum funktioniert er?

- nur nichtnegative Kantengewichte
- wird die "billigste" Verbindung in einer Iteration hinzugenommen, so gilt:
  - es ist die billigste Verbindung des bereits konstruierten Graphen
  - jede Verbindung außerhalb diese Bereichs ist teurer als die gewählte (da Kosten mit zusätzlich hinzugenommenen Kanten nicht sinken können)
- man beachte: Argumentation gilt nicht für negative Kantengewichte!

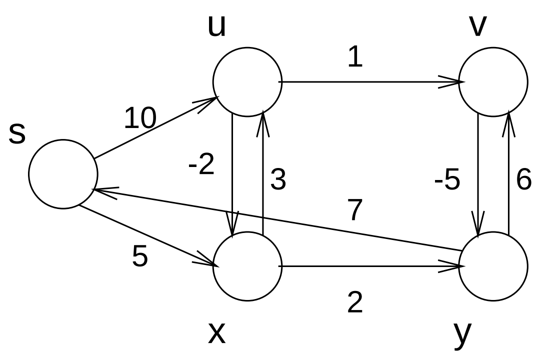
Kürzeste Wege mit negativen Kantengewichten

Bellman-Ford-Algorithmus:

```
BF(G,s):
D[s] = 0;
for i=1 to |E| − 1 do
   for each (u,v) ∈ E do // gerichtete Kanten
   { if D[u] + γ((u,v)) < D[v] then
     {   D[v] = D[u] + γ((u,v)); };
   }
```

funktioniert auch für negative Kantengewichte!

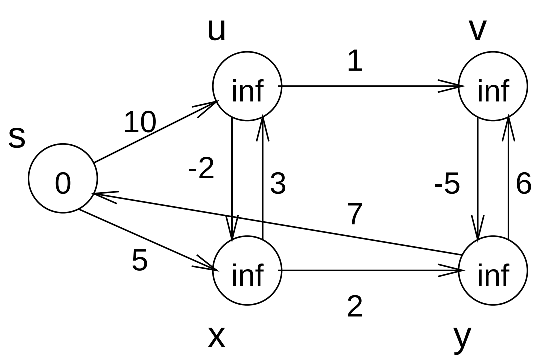
Graph mit negativen Kantengewichten



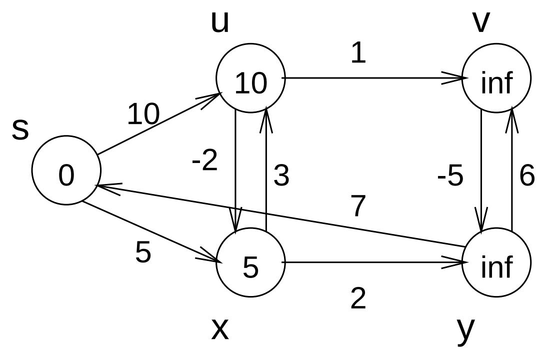
BFA: Bemerkungen zum folgenden Beispiel

- in jedem Schritt werden die D[] der letzten Iteration genommen (**for each** wird parallel ausgeführt)
- Schritte von 1 bis 5 entsprechen Initialisierung und Schleife von i=1 bis 4

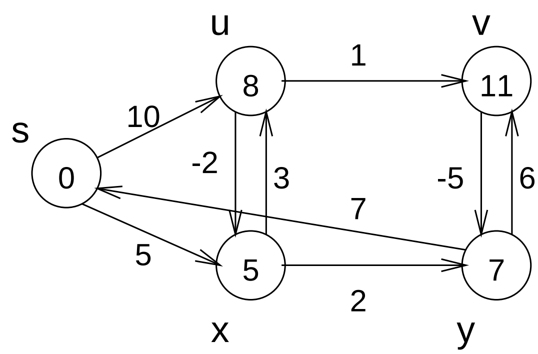
BFA: Schritt 1



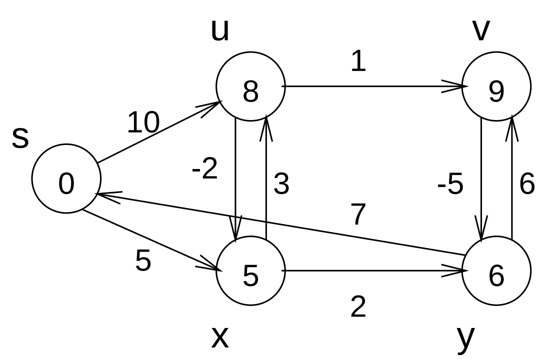
BFA: Schritt 2



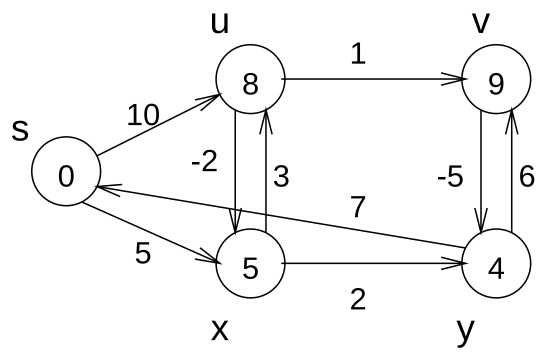
BFA: Schritt 3



BFA: Schritt 4



BFA: Schritt 5



Situation nach $|E|-1$ Iterationen

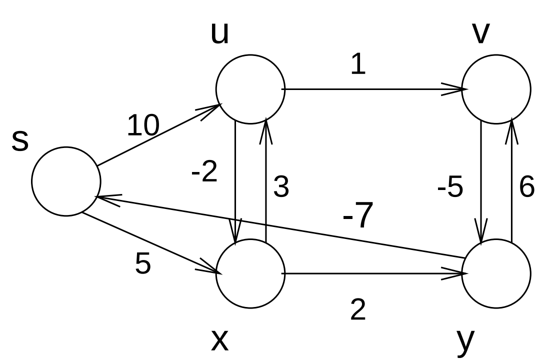
eine Kante könnte noch verbessert werden in einem Schritt

genau dann wenn

der Graph enthält einen Zyklus mit negativer Länge

Beweis: Argumentation über Länge möglicher kürzester Pfade (diese können maximal $|E|-1$ Kanten enthalten ohne Zyklen; deren Gewicht wird bei $|E|-1$ Iterationen korrekt berechnet)

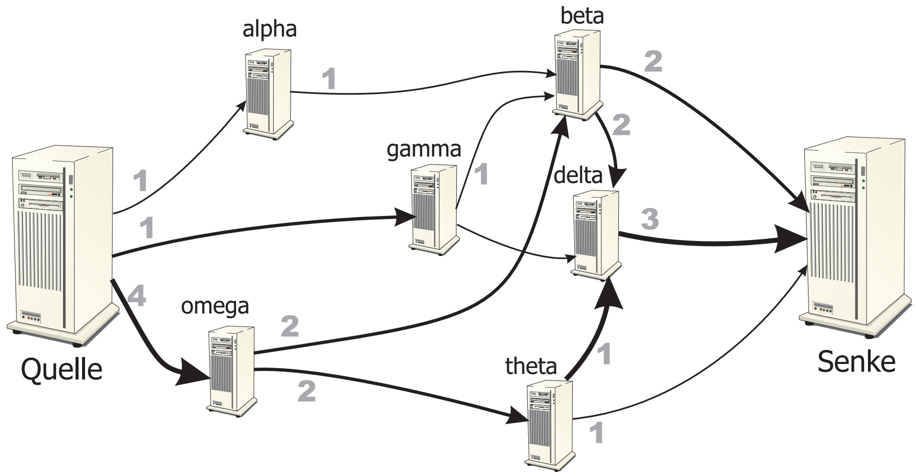
Graph mit negativ gewichteten Zyklus



Zyklus s, x, u, v, y, s hat Kosten $5+3+1-5-7=-3$

Maximaler Durchfluß: Motivation

Paketvermittlung in Computer-Netzwerk:



Maximaler Durchfluß

- Netzwerk (gerichteter Graph G) mit *Kapazitäten*
- Fluß F, für jede Kante
  - c maximale Kapazität
  - f aktueller *Fluß*
  - verbleibende Kapazität
- Verbindung kann in beiden Richtungen genutzt werden
  - zwei gerichtete Kanten mit identischer Kapazität
  - aktueller Fluß kann dabei negative Werte annehmen!

Korrekter Durchfluß

1. Kapazität wird eingehalten:

$$|f(u,v)| \leq C((u,v))$$

2. Konsistenz des Flusses:

$$f(u,v) = -f(v,u)$$

3. Bewahrung des Flusses für jeden Knoten $v \in V - \{q,z\}$ (Quelle q, Ziel z):

$$\sum_{u \in V} f(v,u) = 0$$

Maximaler Durchfluß

1. Wert eines Flusses für j Quelle q:

$$val(G, F) = \sum_{u \in V} f(s, u)$$

2. gesucht für gegebenen Graphen G, Quelle Q, Ziel z : maximaler Fluß

$$\mathbf{max}\{val(G, F) \mid F \in \text{Menge korrekter Flüße}\}$$

Ein Algorithmus

Ford-Fulkerson-Algorithmus:

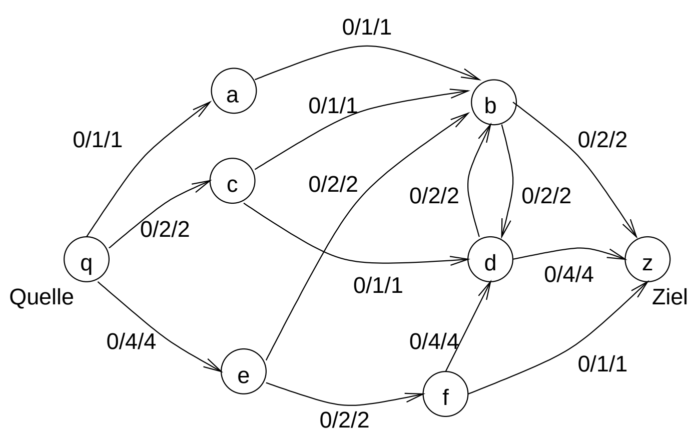
- nutzbarer Pfad: Pfad von q nach z mit verfügbarer Kapazität an allen Kanten > 1
- Pfad hat einen nutzbaren Fluß gegeben durch das Minimum der verfügbaren Kapazität der einzelnen Kanten
- füge solange verfügbare Pfade zum Gesamtfluß hinzu wie möglich

Ford-Fulkerson-Algorithmus

Verfahren im Pseudo-Code:

```
initialisiere Graph;
do
   wähle nutzbaren Pfad aus;
   füge Fluß des Pfades zum Gesamtfluß hinzu;
until kein nutzbarer Pfad mehr verfügbar.
```

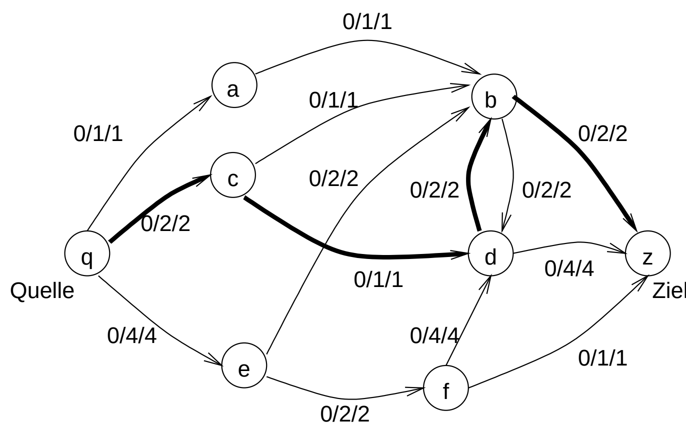
Graph mit Kapazitäten mit leerem Fluß



Graph mit Kapazitäten mit leerem Fluß: Erläuterungen

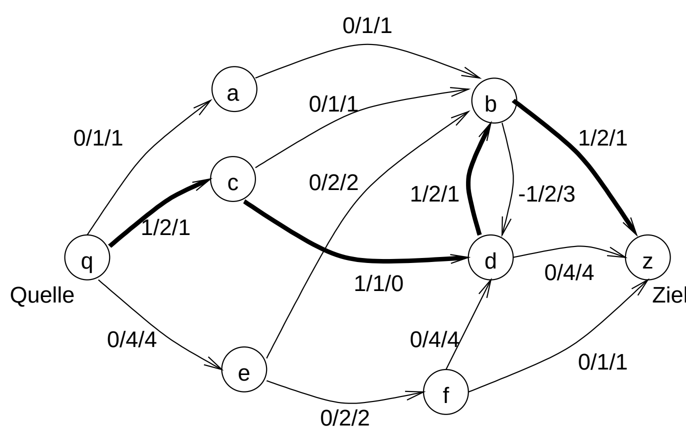
- Kanten mit drei Werten beschriftet:
    1. aktueller Fluß
    2. Kapazität
    3. noch verfügbare Kapazität
- im folgenden Beispiel:
    - ausgewählter nutzbarer Pfad fett dargestellt
    - Kanten ohne verbleibende Kapazität gestrichelt

FFA: Schritt 1 (Auswahl eines Pfades)



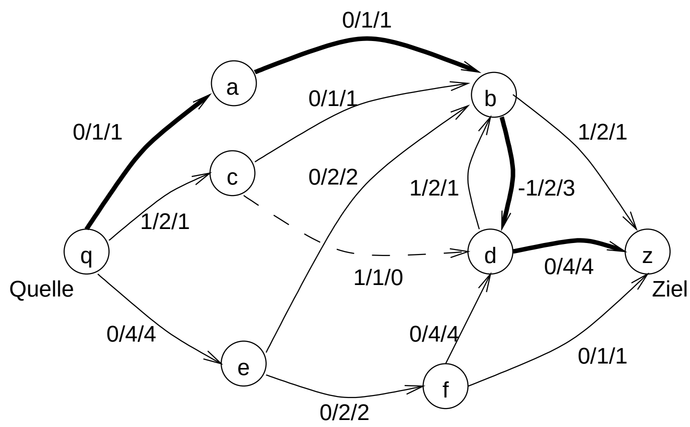
Pfad $q \to c \to d \to b \to z$ mit nutzbarer Kapazität 1

FFA: Schritt 2 (Adjustierung der Kantenbeschriftung)



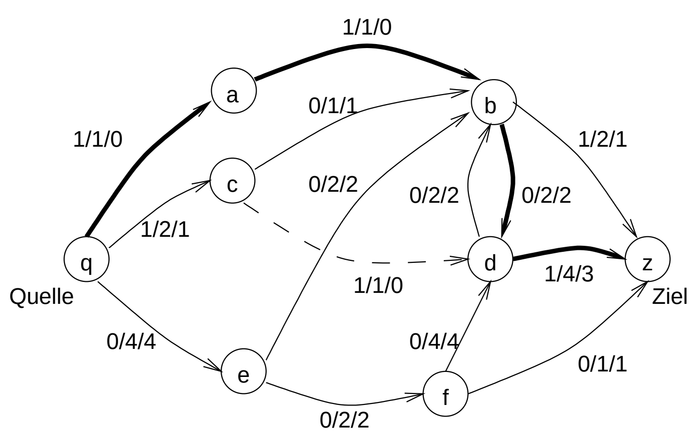
Adjustierung der Rückkante von b nach d mit negativem Wert!

FFA: Schritt 3 (Auswahl eines Pfades)



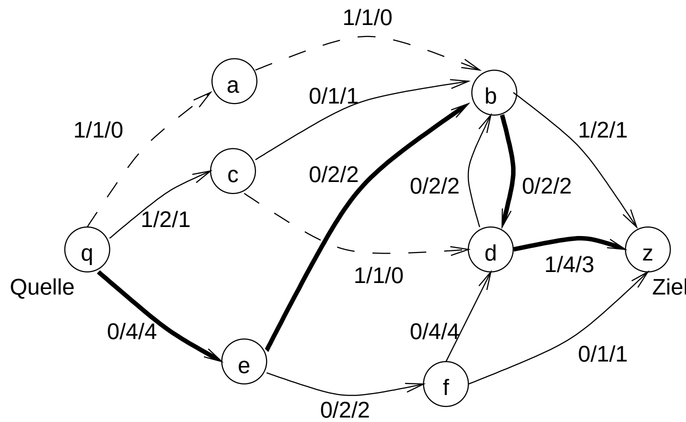
Pfad $q \to a \to b \to d \to z$ mit nutzbarer Kapazität 1

FFA: Schritt 4 (Adjustierung der Kantenbeschriftung)



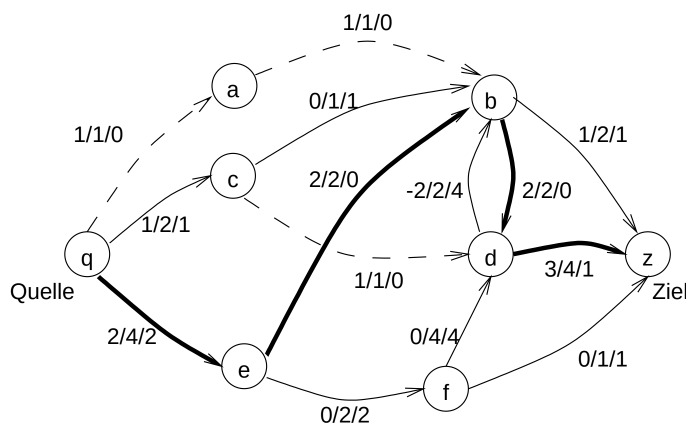
... beinhaltet Aufheben der Nutzung der $b - d$ Verbindung!

FFA: Schritt 5 (Auswahl eines Pfades)

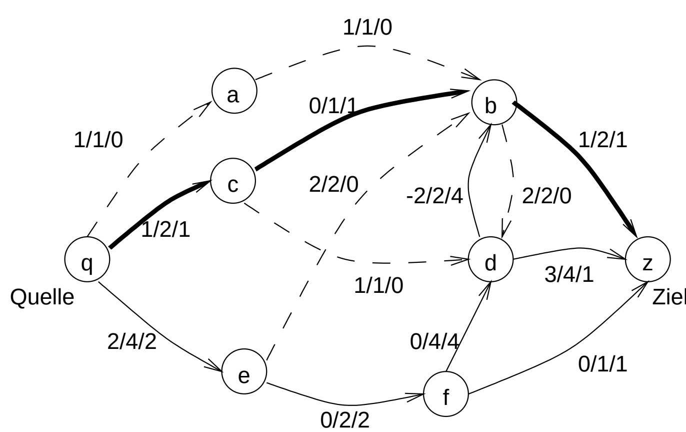


Pfad $q \to e \to b \to d \to z$ mit nutzbarer Kapazität 2

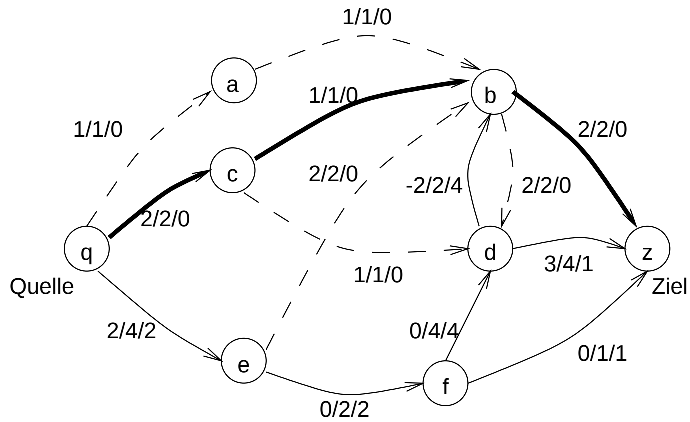
FFA: Schritt 6 (Adjustierung der Kantenbeschriftung)



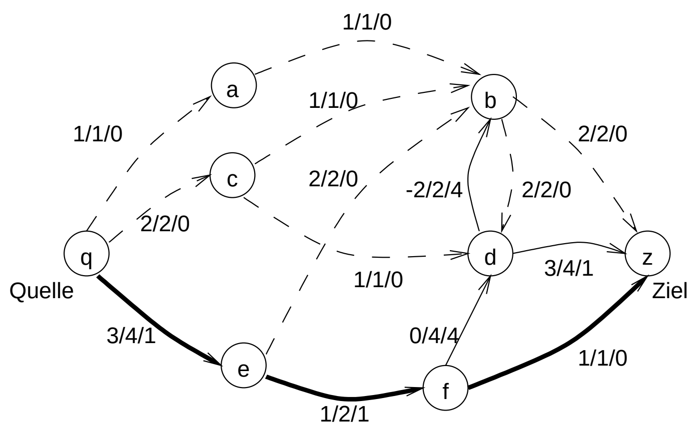
FFA: Schritt 7 (Auswahl eines Pfades)



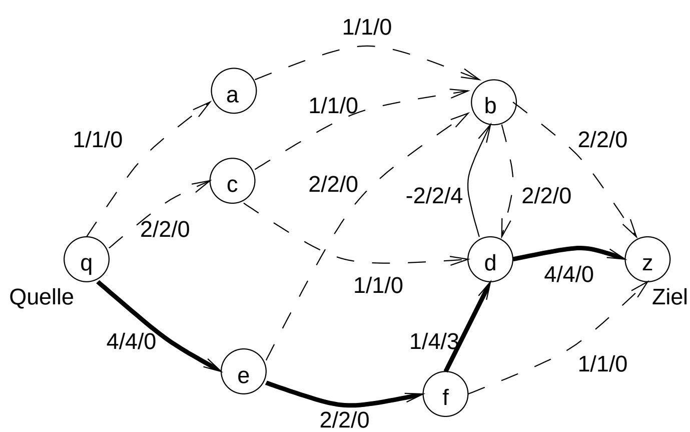
FFA: Schritt 8 (Adjustierung der Kantenbeschriftung)



FFA: Schritt 9/10 (Auswahl + Adjustierung)



FFA: Schritt 11/12 (Auswahl + Adjustierung)



11.5. Weitere Fragestellungen für Graphen

- Problem des Handlungsreisenden
- Planarisierung
- Einfärben von Graphen

Problem des Handlungsreisenden

- *travelling salesman problem*
  - Handlungsreisender soll n Städte nacheinander, aber jede nur einmal besuchen
  - am Ende kommt er in der Ausgangsstadt wieder an
  - Weg mit *minimalen Kosten* (oder unter vorgegebener Schranke)
- Problem ist NP-vollständig (nach heutigem Wissensstand: $O(2^n)$)
- üblich: Optimierungsverfahren für suboptimale Lösungen

Planare Graphen

- gegeben: beliebiger Graph G
- Problem: läßt sich dieser *planar* zeichnen, also ohne überschneidende Kanten?
- mathematische Charakterisierung bekannt: G enthält kein (isomorphes) Abbild von $\mathbf{K}_5$ oder $\mathbf{K}_{3,3}$ als Teilgraphen (siehe nächste Folie)
- Anwendungen: Chip-Design, Visualisierung von Zusammenhängen

Planare Graphen II

weitere Fragestellungen:

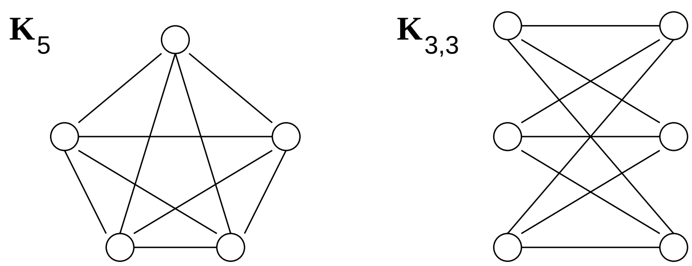
- “schöne” planare Darstellungen
  - Winkel zwischen Kanten an Knoten nicht zu klein
  - Kantenlänge nicht zu stark differierend
  - Knotenanordnung gemäß geometrischer Strukturen
  - etc.
- “schöne” dreidimensionale Darstellungen als Generalisierung?

Planare Graphen III

weitere Fragestellungen (contd.):

- minimale Anzahl von Kantenüberschneidungen
  - Minimierung der Anzahl von 'Brücken' in Förderanlagen
- Zerlegung eines nichtplanaren Graphen in minimale Anzahl planarer Teilgraphen
  - Chip-Design: Leiterbahnen in mehreren Ebenen

$\mathbf{K}_5$ und $\mathbf{K}_{3,3}$



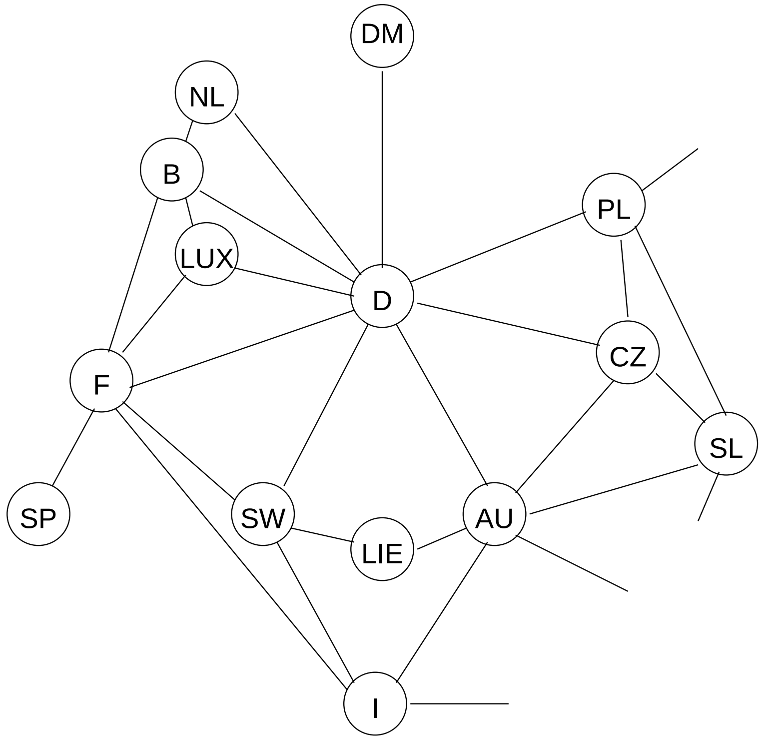
$\mathbf{K}_5$: vollständiger Graph mit 5 Knoten
$\mathbf{K}_{3,3}$: vollständiger bipartiter Graph der Ordnung $(3, 3)$

Einfärben von Graphen

- gegeben: beliebiger Graph G
- Problem: Einfärben der Knoten derart, daß keine zwei benachbarten Knoten die selbe Farbe haben
  - Einfärbung konstruieren
  - minimal benötigte Anzahl Farben bestimmen
- Anwendungen:
  - Vergabe von überschneidungsfreien Klausurterminen (Knoten sind Fächer, Kante gdw. ein Student beide Fächer hört)
  - Einfärben von Landkarten / Visualisierung

Einfärben von Landkarten

Ausschnitt der Ländernachbarschaft in Europa



12. Ausgesuchte algorithmische Probleme

- Spezielle Sortieralgorithmen: Heap-Sort
- Suchen in Texten
  - Probleme der Worterkennung
  - Worterkennung mit endlichen Akzeptoren
  - Verschiedene Algorithmen

12.1. Spezielle Sortieralgorithmen: Heap-Sort

- weiterer Sortieralgorithmus
- basiert auf Einsatz eines speziellen binären Baumes

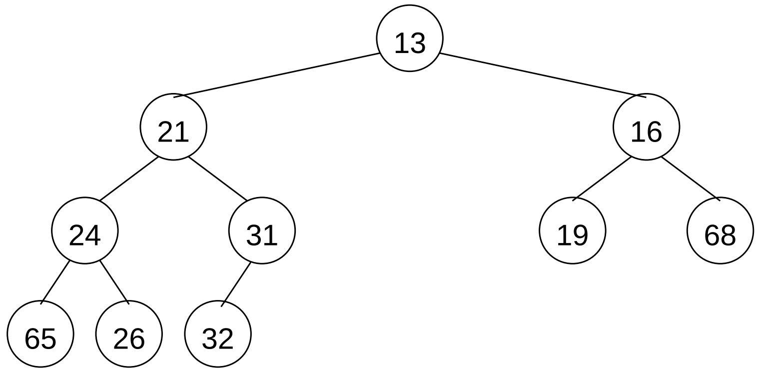
  Daher nicht im ersten Semester behandelt!

- Implementierbar trotzdem auf üblichem Array

Heap als Datenstruktur

- binärer Baum
  - vollständiger Baum (Blattebene von links nach rechts gefüllt)
  - Heap-Eigenschaft: Schlüssel eines jeden Knotens ist kleiner als die Schlüssel seiner Söhne (‘heap order’, ‘Heap-Ordnung’)
- kann zur Implementierung einer Prioritäts-Warteschlange genutzt werden

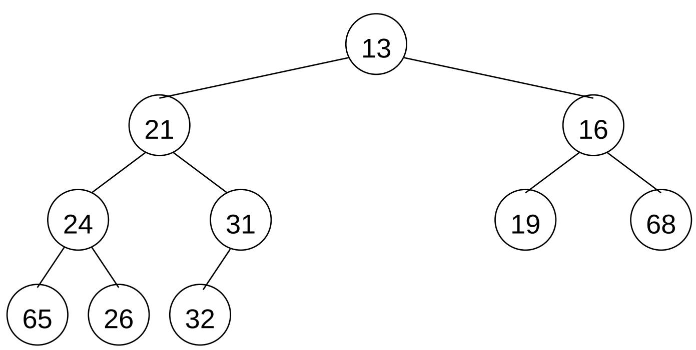
Heap: Beispiel



Heap implementiert über Array

- Wurzel an Position 1
- Söhne eines Knotens an der Position i befinden sich an den Positionen
  - $2i$ (linker Sohn)
  - $2i + 1$ (rechter Sohn)
- vollständiger Baum
  - keine Lücken im Array
  - nächster Knoten wird stets hinten angehängt

Heap als Array: Beispiel



	13	21	16	24	31	19	68	65	26	32		
0	1	2	3	4	5	6	7	8	9	10	11	12

Eigenschaften eines `Heap`

- `findMin` in konstantem Aufwand (Lesen der Wurzel)
- `insert` in $O(\log n)$
- `deleteMin` in $O(\log n)$

Begründung für diesen Aufwand nach Diskussion der Algorithmen!

Variation der Operationen

- `insertSloppy` in $O(1)$: Einfügen ohne Rücksicht auf Heap-Ordnung
- `deleteMin` muß dann `fixHeap` aufrufen (Kosten $O(n)$)

$\rightarrow$ sinnvoll bei vielen Einfügungen vor dem ersten Entnehmen

Operation `insertSloppy`

füge Knoten an nächster Position ein

- im Array: neuer Wert an Position $length + 1$, dann Länge hochzählen

  $\rightarrow$ $O(1)$

verletzt eventuell die Heap-Ordnung

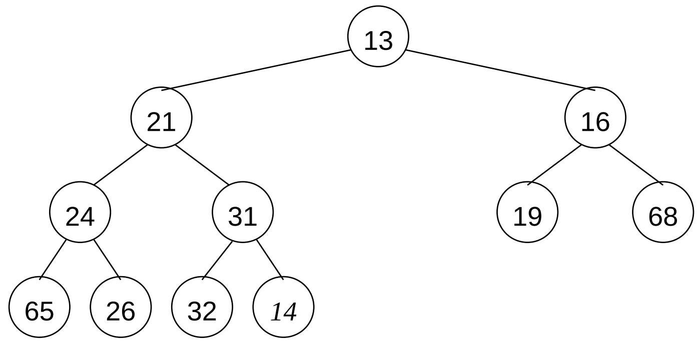
Operation `insert`

wie `insertSloppy` plus Korrektur

- Einfügen an nächster Position
- Wert wandert auf Pfad Richtung Wurzel bis Vorfahr einen kleineren Wert hat

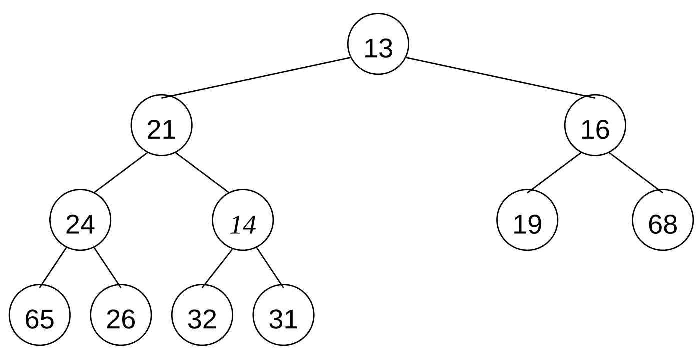
  $\rightarrow$ $O(\log n)$

`insert` in `Heap`: Schritt 1



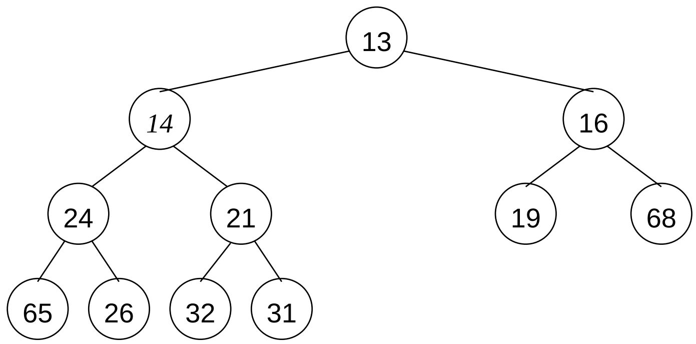
	13	21	16	24	31	19	68	65	26	32	*14*	
0	1	2	3	4	5	6	7	8	9	10	11	12

`insert` in `Heap`: Schritt 2



	13	21	16	24	*14*	19	68	65	26	32	31	
0	1	2	3	4	5	6	7	8	9	10	11	12

`insert` in `Heap`: Schritt 3



	13	*14*	16	24	21	19	68	65	26	32	31	
0	1	2	3	4	5	6	7	8	9	10	11	12

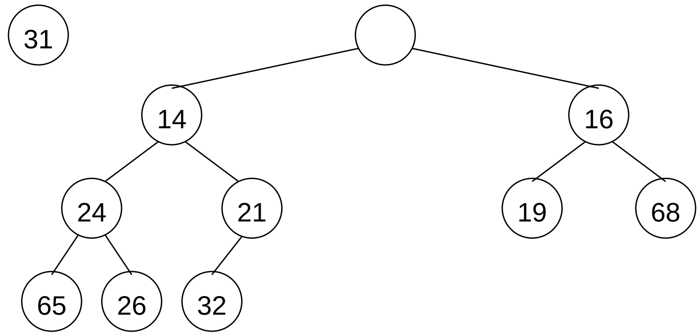
Operation `deleteMin`

- ‘leere’ die Wurzel (enthält Minimum)
- hänge den letzten Knoten (in der Array-Reihenfolge) ab
- Element des letzten Knotens in die Wurzel

  → lasse (nun zu großes) Element nach unten wandern

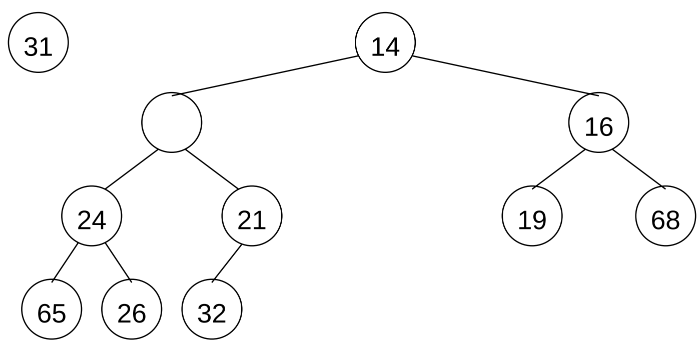
  → wandere jeweils in Richtung des kleineren Elements
- ebenfalls $O(\log n)$

`deleteMin` in `Heap`: Schritt 1



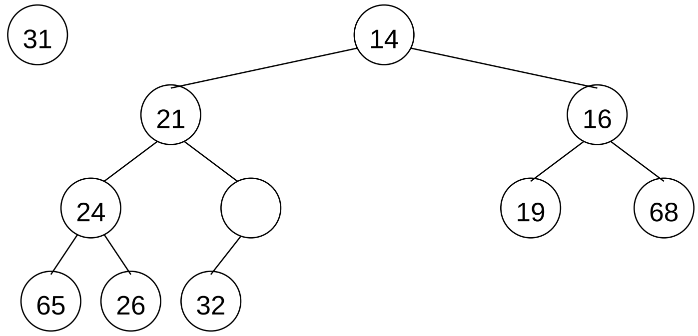
		14	16	24	21	19	68	65	26	32	31	
0	1	2	3	4	5	6	7	8	9	10	11	12

`deleteMin` in `Heap`: Schritt 2



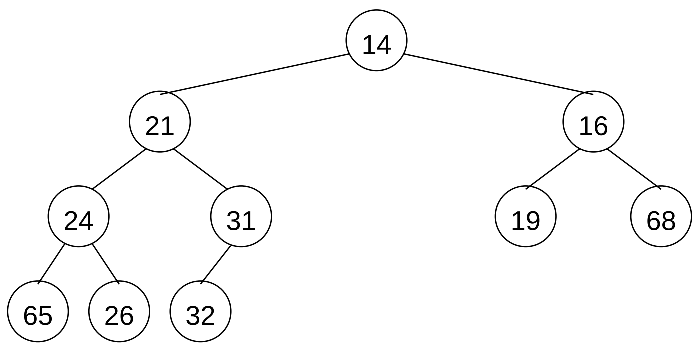
	14		16	24	21	19	68	65	26	32	31	
0	1	2	3	4	5	6	7	8	9	10	11	12

`deleteMin` in `Heap`: Schritt 3



	14	21	16	24		19	68	65	26	32	31	
0	1	2	3	4	5	6	7	8	9	10	11	12

`deleteMin` in `Heap`: Schritt 4



	14	21	16	24	31	19	68	65	26	32		
0	1	2	3	4	5	6	7	8	9	10	11	12

Operation `fixHeap`

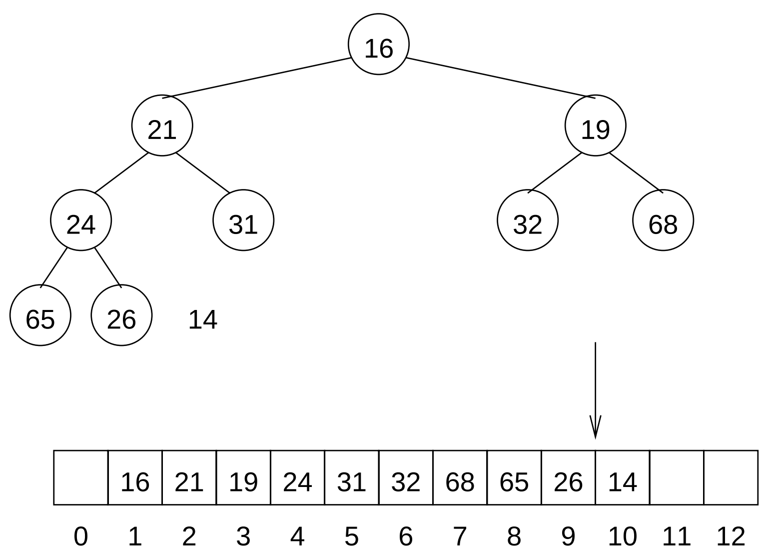
- betrachte in einer Schleife jede Ebene des ‘ungeordneten’ Heaps beginnend direkt über den Blättern
  - bei Verletzung der Heap-Ordnung lasse Element “in den Baum heruntersinken”
- entspricht einer Bottom-Up-Konstruktion des Heaps (in Goodrich/Tamassia)

- Aufwand $O(n)$ (Begründung erfordert etwas Nachdenken; im wesentlichen ist die Bewegung von Werten beim Heruntersinken auf *disjunkte* Pfade begrenzt, so daß der Aufwand durch die Gesamtanzahl Kanten im Baum begrenzt wird...)

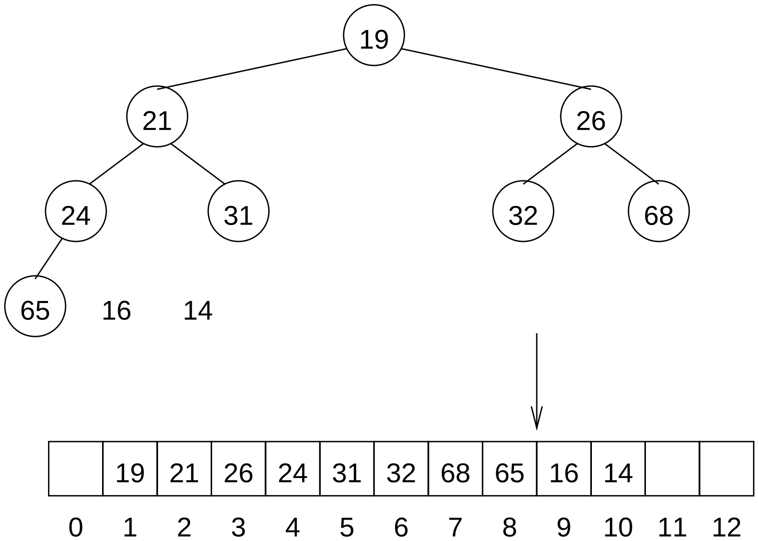
Heap-Sort

- Aufbau des Heaps
  - gegebenes Array; dann `fixHeap`: $O(n)$
  - n-mal `insert`: $O(n \log n)$
- Auslesen des Heaps mit `deleteMin`
  - n-mal `deleteMin`: $O(n \log n)$
  - ausgelesene Elemente können direkt in den frei werdenden Array-Positionen gespeichert werden
    - ▹ bei unserer Realisierung: größte Elemente stehen vorne im Array...
    - ▹ andere Reihenfolge: ein Max-Heap statt einem Min-Heap

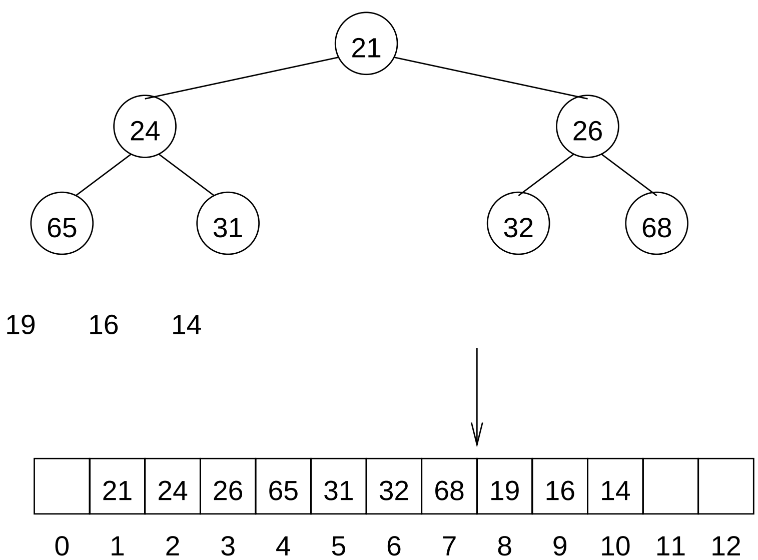
Auslesen beim Heap-Sort: Schritt 1



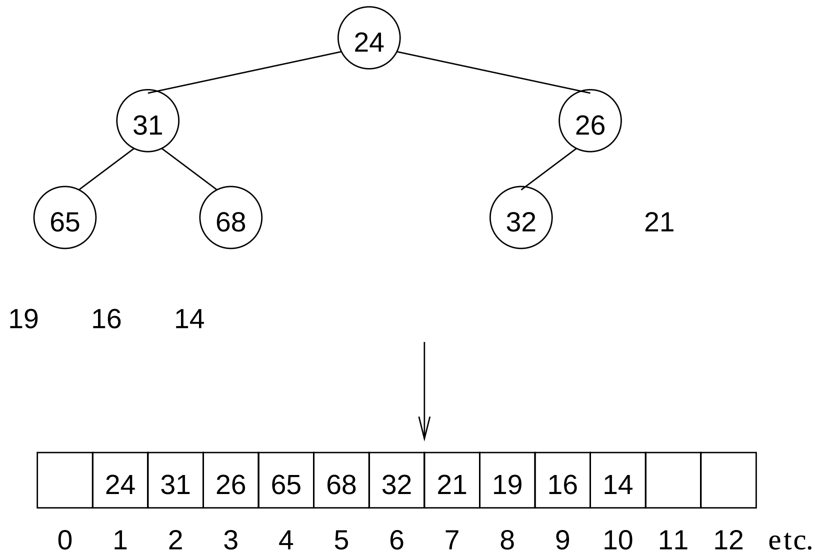
Auslesen beim Heap-Sort: Schritt 2



Auslesen beim Heap-Sort: Schritt 3



Auslesen beim Heap-Sort: Schritt 4



Heap-Sort: Eigenschaften

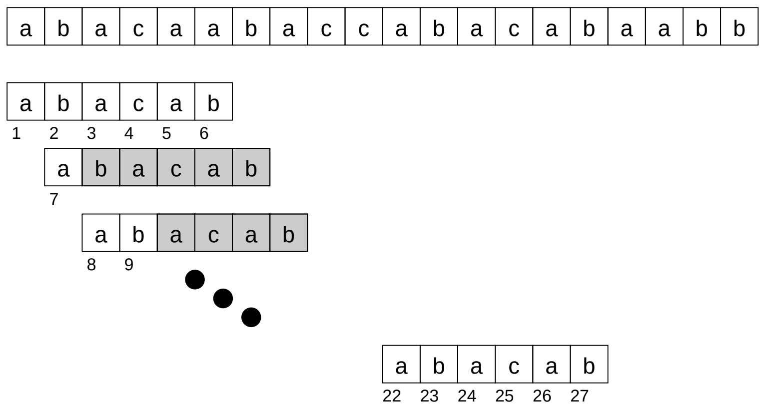
- Aufwand $O(n \log n)$
- einfache iterative Realisierung auf Array
- 'In-Place' Sortierung

12.2. Suchen in Texten

- Problem: Suche eines Teilwortes in einem (langem) anderen Wort
- typische Funktion der Textverarbeitung
  - $\rightarrow$ effiziente Lösung gesucht
  - Maß der Effizienz: Anzahl der Vergleiche zwischen Buchstaben der Worte
- Begriffe: *String-Matching* als Vergleich von Zeichenketten, *Mismatch* als nicht übereinstimmende Position

Probleme der Worterkennung

direkte Lösung (brute force):



Vorgegebene Daten

- Worte als Array:
  - `text[]` zu durchsuchender Text
  - `pat[]` ‘Pattern’, gesuchtes Wort
- Wortlängen:
  - `n` Länge des zu durchsuchenden Textes
  - `m` Länge des gesuchten Wortes
- Σ Alphabet, ϵ leerer String

Naiver Algorithmus

for i=1 **to** $n - m + 1$ **do**
  Prüfe ob pat = text[$i...i + m - 1$]

auch: *Brute Force* Algorithmus

Algorithmus: Brute Force

```
i := 1
WHILE i <= n-m+1 DO
  j := 1
  WHILE j <= m AND pat[j] = text[i+j-1] DO
    j := j+1
  END
  IF j = m+1 THEN RETURN TRUE
  i := i+1
END
RETURN FALSE
```

Algorithmus: Brute Force II

Analyse:

- Komplexität (worst case): $O((n-m)m) = O(nm)$
- zusätzlicher Platzbedarf: $O(1)$

Ziel: Verbesserung der Laufzeitkomplexität, ggf. unter Verschlechterung beim Platzbedarf

Verschiedene bessere Algorithmen

- Knuth-Morris-Pratt

  Idee: verschiebe Wort mehr als eine Stelle nach rechts basierend auf bisherigen Tests

- Boyer-Moore

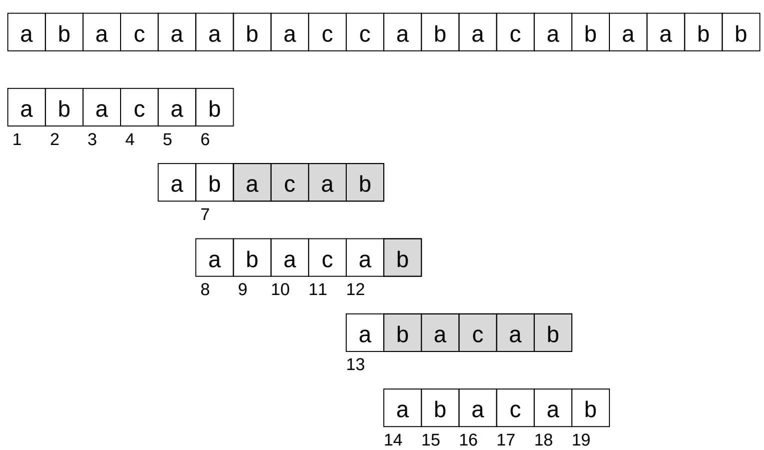
  Idee: Beginne hinten zu testen

String-Matching nach Knuth-Morris-Pratt

Idee: nutze bereits gelesene Information bei einem Mismatch
kommt es an Stelle j von pat zum Mismatch, so gilt:

$$pat[1...j-1] = text[i...i+j-2]$$

Knuth-Morris-Pratt



Knuth-Morris-Pratt: Realisierung mit Fehlerfunktion

bestimme für jedes j Länge des längsten Präfix von pat der Suffix von $pat[1...j]$ ist

j:	0	1	2	3	4	5
	a	b	a	c	a	b
f(j):	0	0	1	0	1	*2*

Fehler an Stelle $j \rightarrow$ verschiebe Suchposition auf $j = f[j-1]$

hier: Realisierung mit Fehlerfunktion nach Goodrich / Tamassia; später: elegantere Darstellung durch endliche Akzeptoren

Knuth-Morris-Pratt im Detail

- Preprocessing: Bestimme für jedes j, $1 \leq j \leq m$ das größte k so daß

  $$pat[1...k-1] \text{ ist echter Suffix von } pat[1...j-1]$$

  Genauer berechnet und als *border* bezeichnet:

  $$border[j] = \max_{1 \leq k \leq j-1} \{k \mid pat[1...k-1] = pat[j-k+1...j-1]\}$$

- bei Mismatch an Position j verschiebe um $j - border[j]$ Stellen (dies entspricht der Fehlerfunktion f)

Die `border`-Tabelle

Beispiel (drei Zeilen: j, $pat[j]$, $border[j]$):

1	2	3	4	5	6	7	8	9	10	11	12	13	j
a	b	a	a	b	a	b	a	a	b	a	a	b	pat[j]
0	1	1	2	2	3	4	3	4	5	6	7	5	border[j]

Bemerkung: Beispiel ist sogenannter Fibonacci-String F_7:

1. $F_0 = \epsilon$, $F_1 = b$, $F_2 = a$

2. $F_n = F_{n-1}F_{n-2}$

Berechnung von border

```
/* Computation of the Border-Table */
border[1] := 0
FOR j := 2 TO m DO border[j] := 1
j = 1; i = 2
WHILE i <= m DO
  WHILE i+j-1 <= m AND pat[j] = pat[i+j-1] DO
    j := j+1
    border[i+j-1] := j
  END
  i := i+j-border[j]
  j := MAX(border[j], 1)
END
```

sborder als Verbesserung von border

Problem:

```
pat:               a b a a b a - -
text: - - - - - - a b a a b c - -
```

Mismatch an Stelle $j = 6$, $border[6] = 3 \to$ verschiebe um $j - border[j] = 3$

```
pat:                     a b a a b a - -
text: - - - - - - a b a a b c - -
```

Resultat: sofort wieder Mismatch!

Verbesserungspotential: Wir wissen bereits, daß an der Mismatch-Stelle kein a stehen kann.

sborder als Verbesserung von border II

Verbesserung:

$$sborder[j] = \max_{1 \leq k \leq j-1} \{k \mid pat[1...k-1] = pat[j-k+1...j-1] \wedge pat[k] \neq pat[j]\}$$

falls kein derartiges k existiert dann 0

Beispiel (vier Zeilen: j, $pat[j]$, $border[j]$, $sborder[j]$):

```
1  2  3  4  5  6  7  8  9 10 11 12 13   j
a  b  a  a  b  a  b  a  a  b  a  a  b   pat[j]
0  1  1  2  2  3  4  3  4  5  6  7  5   border[j]

0  1  0  2  1  0  4  0  2  1  0  7  1   sborder[j]
```

Algorithmus: Knuth-Morris-Pratt I

```
/* Algorithm: Knuth/Morris/Pratt */
i := 1
j := 1
WHILE i <= n-m+1 DO
  WHILE j <= m AND pat[j] = text[i+j-1] DO
    j := j+1
  END
  IF j = m+1 THEN RETURN TRUE
  i := i+j-sborder[j]
  j := MAX(sborder[j],1)
END
RETURN FALSE
```

Algorithmus: Knuth-Morris-Pratt II

```
/* Computation of the Border-Table */
FOR j := 1 TO m DO sborder[j] := 0
j = 1; i = 2
WHILE i <= m DO
  WHILE i+j-1 <= m AND pat[j] = pat[i+j-1] DO
    j := j+1
    IF i+j-1 <= m THEN
      sborder[i+j-1] := sborder[j]
  END
  IF i+j-1 <= m THEN
    sborder[i+j-1] := MAX(j, sborder[i+j-1])
  i := i+j-sborder[j]
  j := MAX(sborder[j], 1)
END
```

Algorithmus: Knuth-Morris-Pratt III

- Komplexität: $O(n + m)$
  - $O(n)$ für Schleife
  - $O(m)$ für Berechnung von *border*
- Platzbedarf: $O(m)$

Beispiel KMP

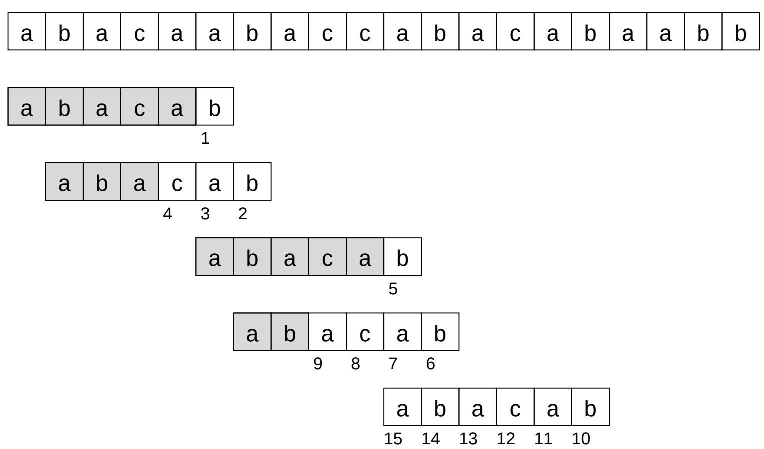
```
a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b

          a b a a b a b a a b a a b
```

```
a b a a b a b a a b a c a b a a b a b a a b a a b

                a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b

                    a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b

                        a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b
```

Beispiel KMP II

- erster Mismatch: $i = 1$, $j = 12$, $sborder[12] = 7$
  führt zu $i = 1 + 12 - sborder[12] = 6$ und $j = sborder[12] = 7$.
- zweiter Mismatch: $i = 6$, $j = 7$, $sborder[7] = 4$
  führt zu $i = 6 + 7 - sborder[7] = 9$ und $j = sborder[7] = 4$.
- dritter Mismatch: $i = 9$, $j = 4$, $sborder[4] = 2$
  führt zu $i = 9 + 4 - sborder[4] = 11$ und $j = sborder[4] = 2$.
- vierter Mismatch: $i = 11$, $j = 2$, $sborder[2] = 1$
  führt zu $i = 11 + 1 - sborder[2] = 11$ und $j = sborder[2] = 1$.

Boyer-Moore



Prinzip von Boyer-Moore

- Vergleich nun von rechts nach links
- Überlegungen: Shift-Tabelle $D[j]$ beinhaltet sichere Verschiebung falls $pat[j] \neq text[i + j - 1]$ und $pat[j + 1...m] = text[i + j...i + m - 1]$

```
1  2  3  4  5  6  7  8  9 10 11 12 13   j
a  b  a  a  b  a  b  a  a  b  a  a  b   pat[j]
8  8  8  8  8  8  8  3 11 11  6 13  1   D[j]
```

Beispiel für Shift-Tabellen

```
datenbank    compiler
999999991    88888881

kuckuck   rokoko   papa
3336661   662641   2241
```

Berechnung der Shift-Tabellen

1. Initialisierung mit dem maximalen Shift m
2. Berechne *rborder* (wie *border*, aber gespiegelt)
3. Wähle Minimum für Shift aus zwei Fällen:
   a) bisher akzeptierter Suffix taucht vollständig im Pattern erneut auf
   b) ein Endstück des bisher akzeptierten Suffix ist Anfangsstück des Patterns

Bemerkung: Berechnung auf den Folien enthalten (in der Notation der angebenen WWW-Seite), aber nicht detailliert Inhalt der Vorlesung.

Ausnutzung des Buchstabenauftretens

- `last`-Tabelle:
$$last[c] := \max_{1 \leq j \leq m} \{j \mid pat[j] = c\}$$
- Nutzung wird als *Occurence-Heuristik* bezeichnet

```
 a   b   c   Buchstaben
12  13   0   last[c] fuer F7
```

- Nun Verschiebungen um
  - vereinfachter (simple) Boyer-Moore:
$$j - last[text[i + j - 1]]$$
  - verbesserter Boyer-Moore:
$$\max(D[j], j - last[text[i + j - 1]])$$

  bringt Verbesserung bei großen Alphabeten

Algorithmus: Simple Boyer/Moore

```
/* Computation of the last table */
FOR c := 'a' TO 'Z' DO
  last[c] := 0;
END
i := 1
WHILE i <= m DO
  last[pat[i]] = i
  i := i+1
END
```

Algorithmus: Simple Boyer/Moore II

```
/* Simple Boyer Moore */
i := 1
WHILE i <= n-m+1 DO
  j := m
  WHILE j >= 1 AND pat[j] = text[i+j-1] D
    j := j-1
  END
  IF j = 0 THEN RETURN TRUE
  i := i+MAX(1, j-last[text[i+j-1]])
END
RETURN FALSE
```

Algorithmus: Simple Boyer/Moore III

- Komplexität: $O((n-m)m)$
- Platzbedarf: $O(1)$

  (Kardinalität des Alphabets ist konstant)

Algorithmus: Boyer/Moore

```
/* Computation of the Shift-Table (phase 1) */
rborder[m] := 0
D[m] := 0
FOR j := m-1 DOWNTO 0 DO
  rborder[j] := 1
  D[j] := m
END
j = 1
i = m-1
WHILE i >= j DO
  WHILE i >= j AND pat[m-j+1] = pat[i-j+1] DO
```

```
    j := j+1
    rborder[i-j+1] := j
  END
  IF j > 1 THEN
    D[m-j+1] = MIN(m-i, D[m-j+1])
  ENDIF
  i := i-j+rborder[m-j+1]
  j := MAX(rborder[m-j+1], 1)
END
```

Algorithmus: Boyer/Moore II

```
/* Computation of the Shift-Table (phase 2) */
t := rborder[0]
L := 1
WHILE t > 0 DO
  s = m-t+1
  FOR j := L TO s DO
    D[j] := MIN(D[j], s)
  END
  t := rborder[s]
  L := s+1
END
```

Algorithmus: Boyer/Moore III

```
/* Algorithm: Boyer/Moore */
i := 1
WHILE i <= n-m+1 DO
  j := m
  WHILE j >= 1 AND pat[j] = text[i+j-1] DO
    j := j-1
  END
  IF j = 0 THEN RETURN TRUE
  i := i+D[j]
END
RETURN FALSE
```

Algorithmus: Boyer/Moore IV

- Komplexität: $O(n+m)$
- Platzbedarf: $O(m)$

Algorithmus: Boyer/Moore with Occurrence Check

```
/* Computation of the Shift-Table (phase 1) */
rborder[m] := 0
D[m] := 0
FOR j := m-1 DOWNTO 0 DO
  rborder[j] := 1
  D[j] := m
END
j = 1
i = m-1
WHILE i >= j DO
  WHILE i >= j AND pat[m-j+1] = pat[i-j+1] DO
    j := j+1
    rborder[i-j+1] := j
  END
  IF j > 1 THEN
    D[m-j+1] = MIN(m-i, D[m-j+1])
  ENDIF
  i := i-j+rborder[m-j+1]
  j := MAX(rborder[m-j+1], 1)
END
```

Algorithmus: Boyer/Moore with Occurrence Check II

```
/* Computation of the Shift-Table (phase 2) */
t := rborder[0]
L := 1
WHILE t > 0 DO
  s = m-t+1
  FOR j := L TO s DO
    D[j] := MIN(D[j], s)
  END
  t := rborder[s]
  L := s+1
END
```

Algorithmus: Boyer/Moore with Occurrence Check III

```
/* Computation of the last table */
FOR c := ’a’ TO ’Z’ DO
  last[c] := 0;
END
i := 1
WHILE i <= m DO
  last[pat[i]] = i
  i := i+1
```

```
END
```

Algorithmus: Boyer/Moore with Occurrence Check IV

```
/* Algorithm: Boyer/Moore with Occurrence Check*/
i := 1
WHILE i <= n-m+1 DO
  j := m
  WHILE j >= 1 AND pat[j] = text[i+j-1] DO
    j := j-1
  END
  IF j = 0 THEN RETURN TRUE
  i := i+MAX(D[j], j-last[text[i+j-1]])
END
RETURN FALSE
```

Worterkennung mit endlichen Akzeptoren

Motivation:

- einheitliches Modell für die Mustererkennung
- erweiterte Suchfunktion nach regulären Ausdrucken
  - Wildcards: Überspringe einen Buchstaben
  - Wiederholung (0 bis n)
  - optionale Teile, Auswahl, etc
- Werkzeuge wie `grep`, `awk`, Standardfunktionen beim Auflisten von Dateinamen

Reguläre Ausdrücke in Unix-Systemen

- `.` : Wildcard
- `a*` : beliebige Anzahl von `a`
- `[abc]` : irgendein Buchstabe der Aufzählung
- Spezialzeichen für Zeilenbeginn, Zeilenende
- `a+` : beliebige Anzahl größer 0 von `a`
- `a?` : genau 0 oder 1 mal `a`
- `|` : Alternative, `()` : Gruppierung, Metazeichenumwandlung, etc

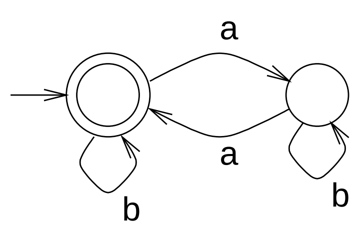
Endliche Akzeptoren

auch: endlicher (Zustands-) Automaten

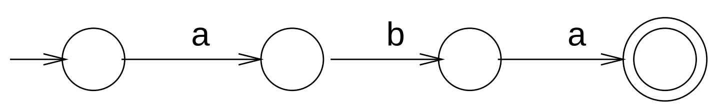
- S Menge von Zuständen (states)
- Alphabet Σ
- $s_i \in S$ initialer Zustand
- $S_f \subseteq S$ Menge von Endzuständen (oft auch: genau ein Endzustand s_f)
- Transitionsfunktion $\delta : S \times \Sigma \to S$
  - Transitionen mit ϵ
  - spezielle Transition mit * als Abkürzung: Übergang bei beliebigen Zeichen aus Σ

Beispiele für Akzeptoren

1)



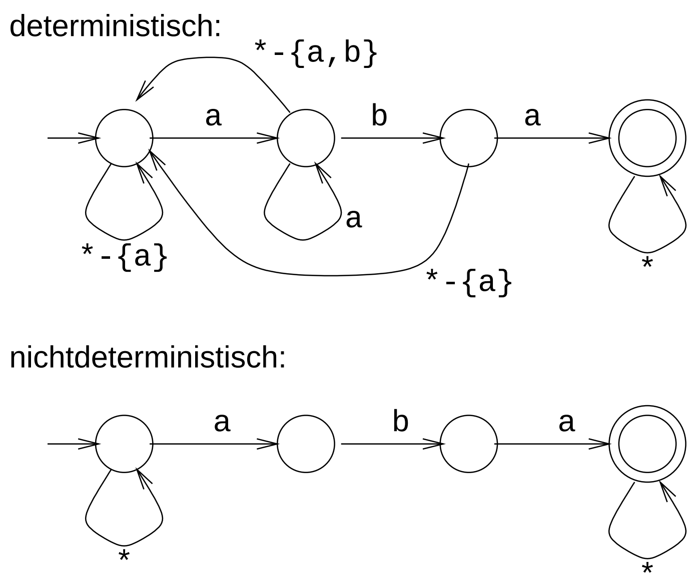
2)



1) erkennt Worte mit gerader Anzahl von a, 2) erkennt Wort aba

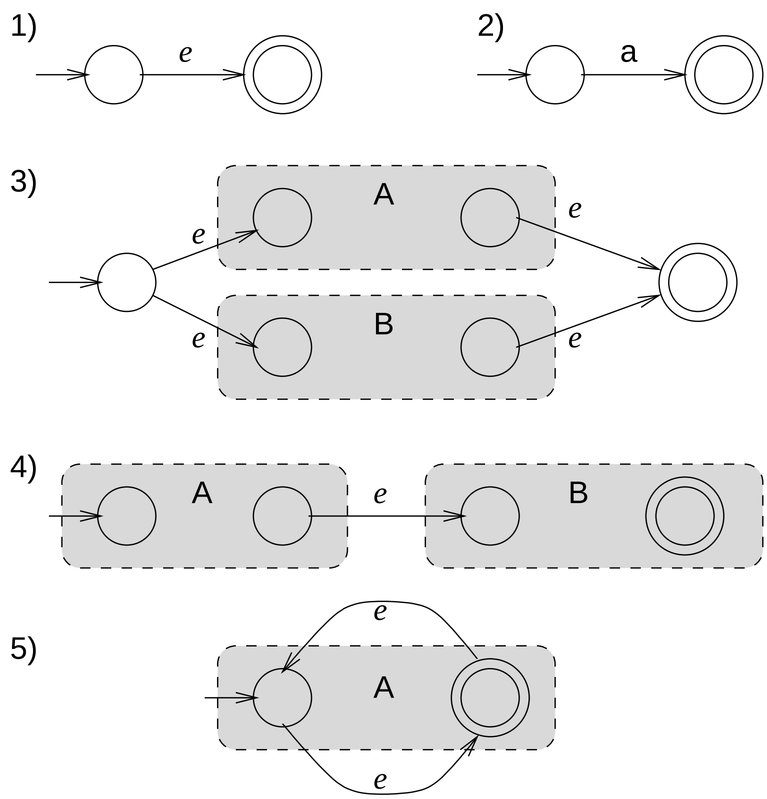
Nichtdeterministische Akzeptoren

deterministische Akzeptoren definieren genau einen Weg, nichtdeterministische fordern nur daß *ein* Weg zum Ziel führt



Beispiel: Erkennen von Worten die den Teil-String *aba* enthalten

Konstruktion von Akzeptoren



Konstruktion von Akzeptoren: 1) ϵ, 2) a, 3) $A + B$, 4) AB, 5) A^*

Bemerkungen zu Akzeptoren

- Trie kann als Akzeptor für die in ihm gespeicherten Worte genutzt werden
- Fehlerfunktion aus KMP-Algorithmus definiert einen speziellen Akzeptor

13. Verzeichnisse und Datenbanken

Prinzipien zur Verwaltung großer strukturierter Datenbestände

- Daten in Tabellenform
- effizienter Zugriff nach wahlfreien Kriterien
- SQL als Datenzugriffssprache

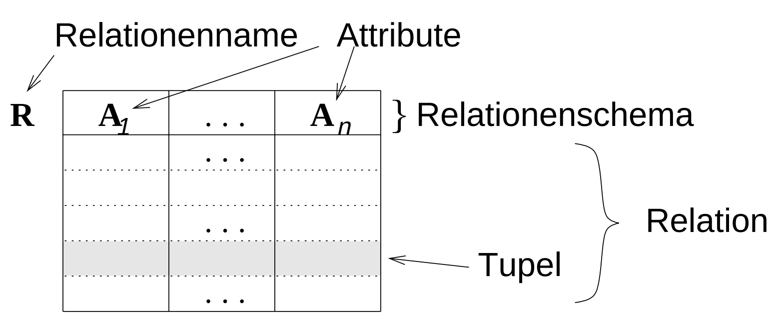
Warum in dieser Vorlesung?

- Datenbanken zeigen praktischen Einsatz vieler vorgestellter Methoden, Algorithmen und Datenstrukturen
- Datenzugriff ist fundamentaler Aspekt aller kommerziellen Anwendungssysteme
- SQL zeigt ein alternatives Berechnungsparadigma

13.1. Relationale Datenbanken

- Relation = Abstraktion einer Tabelle
  einfache, intuitive Datendarstellung
- effiziente Abspeicherung von Relationen
- wenige Basisoperationen (effizient implementiert)
- Abbildung einer höheren Sprache (SQL) auf diese Basisoperationen

Tabellen als Relationen: Begriffe



Beispiele für Relationen

AUSLEIH

INV.NR	NAME
4711	Meyer
1201	Schulz
0007	Müller
4712	Meyer

BUCH

INV.NR	TITEL	ISBN	AUTOR
0007	Dr. No	3-125	James Bond
1201	Objektbanken	3-111	Heuer
4711	Datenbanken	3-765	Vossen
4712	Datenbanken	3-891	Ullman
4717	PASCAL	3-999	Wirth

Operationen auf Tabellen: Selektion

SELEKTION: Zeilen (Tupel) auswählen

```
SEL [NAME = 'Meyer'] (AUSLEIH)
```

ergibt:

INVENTARNR	NAME
4711	Meyer
4712	Meyer

Operationen auf Tabellen: Projektion

PROJEKTION: Spalten (Attribute) auswählen

```
PROJ [INVENTARNR, TITEL] (BUCH)
```

ergibt:

INVENTARNR	TITEL
0007	Dr. No
1201	Objektbanken
4711	Datenbanken
4712	Datenbanken
4717	PASCAL

Operationen auf Tabellen: Projektion II

- Achtung: doppelte Tupel werden entfernt!
- das relationale Modell realisiert eine *Mengensemantik* für Relationen
  - Duplikate werden entfernt

- Projektion mit Duplikateliminierung ist $O(n \log n)$ (Duplikaterkennung durch Sortierung)
- aufgrund dieses Aufwands haben kommerzielle Systeme eine Multimengensemantik mit explizitem Operator zur Duplikatunterdrückung

Operationen auf Tabellen: Verbund

VERBUND (JOIN): Tabellen verknüpfen über gleichbenannte Spalten und gleiche Werte

$$\begin{gathered}\text{PROJ }[\text{INVENTARNR}, \text{TITEL}](\text{BUCH})\\ \text{JOIN}\\ \text{SEL }[\text{NAME} = \text{'Meyer'}](\text{AUSLEIH}).\end{gathered}$$

ergibt:

INVENTARNR	TITEL	NAME
4711	Datenbanken	Meyer
4712	Datenbanken	Meyer

auch: *natürlicher Verbund*

Relationale Algebra

- Grundoperationen Selektion σ, Projektion π, Verbund $\bowtie$
- Weitere Operationen: Vereinigung $\cup$, Differenz $-$, Durchschnitt $\cap$, Umbenennung β
- Operationen beliebig kombinierbar (sie bilden eine *Algebra* der Relationen)

13.2. SQL als Anfragesprache

Modell von SQL

- Multimengen statt Mengen
- Null-Werte
- Datentypen mit Arithmetik
- sortierte Ausgaben

`create table` — Beispiele

Definition einer Tabelle

```
create table Bücher
             ( ISBN char(10) not null,
             Titel varchar(200),
             Verlagsname varchar(30) )
```

`create table` — Beispiele II

Definition einer Tabelle mit (tabellenübergreifenden) Integritätsbedingungen

```
create table Bücher
   ( ISBN char(10) not null,
   Titel varchar(200),
   Verlagsname varchar(30),
   primary key (ISBN),
   foreign key (Verlagsname)
         references Verlage (Verlagsname) )
```

`create table` — Integritätsbedingungen

- Schlüssel **`primary key`**: identifizierendes Attribut

  Ein Buch wird durch die ISBN identifiziert.
- Fremdschlüssel **`foreign key`**: Verweis auf Schlüssel einer (anderen) Tabelle
- Das Attribut `Verlagsname` ist Schlüssel einer weiteren Tabelle, in der zusätzliche Informationen zu Verlagen (Ort, Adresse, etc.) stehen.

SQL-Kern

`select`

- Projektionsliste
- *arithmetische Operationen und Aggregatfunktionen*

`from`

- zu verwendende Relationen (für Verbund)
- eventuelle Umbenennungen (durch Tupelvariable oder ‘alias’)

`where`

- Selektionsbedingungen
- Verbundbedingungen (nötig da alle Attribute verschieden bezeichnet)
- Geschachtelte Anfragen (wieder ein SFW-Block)

group by

- *Gruppierung* für Aggregatfunktionen

having

- *Selektionsbedingungen an Gruppen*

SQL-Anfragen

Konzepte: Selektion, Projektion, Verbund, Tupelvariablen

```
select Bücher.ISBN, Titel, Stichwort
from Bücher, Buch_Stichwort
where Bücher.ISBN = Buch_Stichwort.ISBN
```

SQL-Anfragen mit Tupelvariablen

Tupelvariablen sind notwendig:

```
select eins.ISBN, zwei.ISBN
from Bücher eins, Bücher zwei
where eins.Titel = zwei.Titel
```

Paare von Büchern mit gleichem Titel.

exists-Prädikat

```
select ISBN
from Buch_Exemplare
where  exists
      ( select *
        from Ausleihe
        where Inventarnr = Buch_Exemplare.Inventarnr)

select ISBN
from Buch_Exemplare
where not  exists
      ( select *
        from Ausleihe
        where Inventarnr = Buch_Exemplare.Inventarnr)
```

Änderungsoperationen

- Einfügen von Tupeln in Basisrelationen **insert**
- Löschen von Tupeln aus Basisrelationen **delete**

- Ändern von Tupeln in Basisrelationen **update**

Diese Operationen jeweils als

- Eintupel-Operationen (etwa die Erfassung einer neuen Ausleihung)
- Mehrtupel-Operationen (erhöhe das Gehalt aller Mitarbeiter um 4.5%)

Beispiel für Änderungsoperationen

```
update Angestellte
   set Gehalt = Gehalt + 1000
   where Gehalt < 5000

update Angestellte
   set Gehalt = 6000
   where Name = 'Bond'

update Angestellte
   set Gehalt = 3000

delete from Ausleihe
   where Invnr = 4711

delete from Ausleihe
   where Name = 'Meyer'

delete from Ausleihe

insert into Buch
   values (4867,'Wissensbanken',
           '3-876','Karajan')

insert into Kunde
   ( select LName, LAdr, 0
     from Lieferant )
```

13.3. Algorithmen und Datenstrukturen in einem relationalen Datenbanksystem

- Aufgaben eines RDBMS
- Speicherung von Tabellen
- Optimierung und Auswertung von Anfragen

Aufgaben eines RDBMS

- effiziente Speicherung von Relationen
- Optimierung und Ausführung von SQL-Anfragen und Änderungen
- Datendefinition und Änderung der Datenspeicherung zur *Laufzeit*
- Kontrolle des Mehrbenutzerbetriebs, Datenschutz, Datensicherung, Datenkatalog, ...

Speicherung von Relationen

Dateiorganisationsform:

- als Heap (unsortiert)
- sortiert nach Schlüssel (etwa Matrikelnummer)
- B-Baum nach Schlüssel
- Hash-Tabelle nach Schlüssel

Speicherung von Relationen II

Zugriffsunterstützung durch *Indexe*:

- Tupelidentifikatoren verweisen auf Hauptdatei
- B-Bäume für Nicht-Schlüsselattribute zur Beschleunigung des Zugriffs (etwa für Studentenname)

Relevante Kosten

- wichtigster Kostenfaktor: Transfer vom Hintergrundspeicher in den Hauptspeicher, *Zugriffslücke*
- Transfereinheit: Block / Seite (z.B. 1 KiloByte)
- Kostemnberechnung basiert auf Anzahl zwischen Hauptspeicher und Festplatte bewegten Blöcken

Zugriffslücke

- Magnetplatten pro Jahr 70% mehr Speicherdichte
- Magnetplatten pro Jahr 7% schneller
- Prozessorleistung pro Jahr um 70% angestiegen

- Zugriffslücke zwischen Hauptspeicher und Magnetplattenspeicher beträgt 10^5
- Größen: ns für Nanosekunden (also 10^{-9} Sekunden, ms für Millisekunden (10^{-3} Sekunden), KB (KiloByte), MB (MegaByte), GB (GigaByte) und TB (TeraByte) für jeweils 10^3, 10^6, 10^9 und 10^{12} Bytes

Zugriffslücke in Zahlen

Speicherart	**typische Zugriffszeit**	**typische Kapazität**
Cache-Speicher	6 ns	512 KB bis 32 MB
Hauptspeicher	60 ns	32 MB bis 1 GB
	— *Zugriffslücke* 10^5 —	
Magnetplattenspeicher	12 ms	1 GB bis 10 GB
Platten-Farm oder -Array	12 ms	im TB-Bereich

Optimierer

Ausnutzung der Äquivalenz von Algebra-Termen

1. $\sigma_{A=Konst}(r(\mathsf{REL1}) \bowtie r(\mathsf{REL2}))$ /* *wobei* A *aus* REL1 */
2. $(\sigma_{A=Konst}(r(\mathsf{REL1}))) \bowtie r(\mathsf{REL2})$

(Relation REL1 mit 100 Tupeln, Relation REL2 mit 50 Tupeln, 10 verschiedene Werte für Attribut A)

heuristische Strategien: *“Führe Selektionen möglichst früh aus, da sie Tupelanzahlen in Relationen verkleinern”*

Optimierer II

1. Im ersten Fall erhalten wir $100 * 50 = 5.000$ Operationen als Zwischenergebnis für die Join-Ausführung. Das Zwischenergebnis enthält 5.000 Tupel, die für die Selektion alle durchlaufen werden müssen. Wir erhalten insgesamt 10.000 Operationen.
2. Laut unserer Annahme erfüllen zehn Tupel in REL1 die Bedingung $A = Konst$. Die Ausführung der Selektion erfordert 100 Zugriffe und die Ausführung des Verbundes nun zusätzlich $10 * 50 = 500$ Operationen. Insgesamt werden somit 600 Operationen benötigt.

Berechnung von Verbunden: Verbund durch Mischen

Idee *Merge-Join*:

- Merge-Join / *Verbund durch Mischen* von R_1 und R_2 (effizient, wenn eine oder beide Relation(en) sortiert nach den Verbund-Attributen vorliegen)

- Seien X Verbund-Attribute $X := R_1 \cap R_2$.
  - r_1 und r_2 werden nach X sortiert, falls diese nicht schon sortiert sind.
  - Dann werden r_1 und r_2 gemischt, d.h., beide Relationen werden parallel sequentiell durchlaufen und passende Paare in das Ergebnis aufgenommen.

Berechnung von Verbunden: Verbund mittels Schleifen

Idee *Nested-Loops-Join*:

- Nested-Loops-Join realisiert eine doppelte Schleife über beide Relationen.
- Liegt bei einer der beiden Relationen ein Zugriffspfad für die X-Attribute vor, kann die innere Schleife durch direkten Zugriff über diesen Zugriffspfad ersetzt werden.

Berechnung von Verbunden

Varianten der Verbundberechnung

- Nested-Loops-Verbund
- Block-Nested-Loops-Verbund
- Merge-Join
- Hash-Verbund
- ...

Nested-Loops-Verbund

doppelte Schleife iteriert über alle $t_1 \in r$ und alle $t_2 \in s$ bei einer Operation $r \bowtie s$
$r \bowtie_\varphi s$:

```
for each t_r ∈ r do
begin
   for each t_s ∈ s do
   begin
      if φ(t_r, t_s) then put(t_r · t_s) endif
   end
end
```

Funktioniert im Gegensatz zu den anderen Verfahren auch für Verbundbedingungen, die kein Gleichheitsprädikat benutzen!

Block-Nested-Loops-Verbund

statt über Tupel über Blöcke iterieren

```
for each Block $B_r$ of $r$ do
begin
   for each Block $B_s$ of $s$ do
   begin
      for each Tupel $t_r \in B_r$ do
      begin
         for each Tupel $t_s \in B_s$ do
         begin
            if $\varphi(t_r, t_s)$ then put($t_r \cdot t_s$)
            endif
         end
      end
   end
end
```

Aufwand (hängt von der Blockanzahl, nicht der Tupelanzahl ab): $b_r * b_s$

Merge-Techniken

Verbundattribute $X := R \cap S$; falls nicht bereits sortiert, zuerst Sortierung von r und s nach X

1. $t_r(X) < t_s(X)$, nächstes $t_r \in r$ lesen
2. $t_r(X) > t_s(X)$, nächstes $t_s \in s$ lesen
3. $t_r(X) = t_s(X)$, t_r mit t_s und allen Nachfolgern von t_s, die auf X mit t_s gleich, verbinden
4. beim ersten $t'_s \in s$ mit $t'_s(X) \neq t_s(X)$ beginnend mit ursprünglichem t_s mit den Nachfolgern t'_r von t_r wiederholen, solange $t_r(X) = t'_r(X)$ gilt

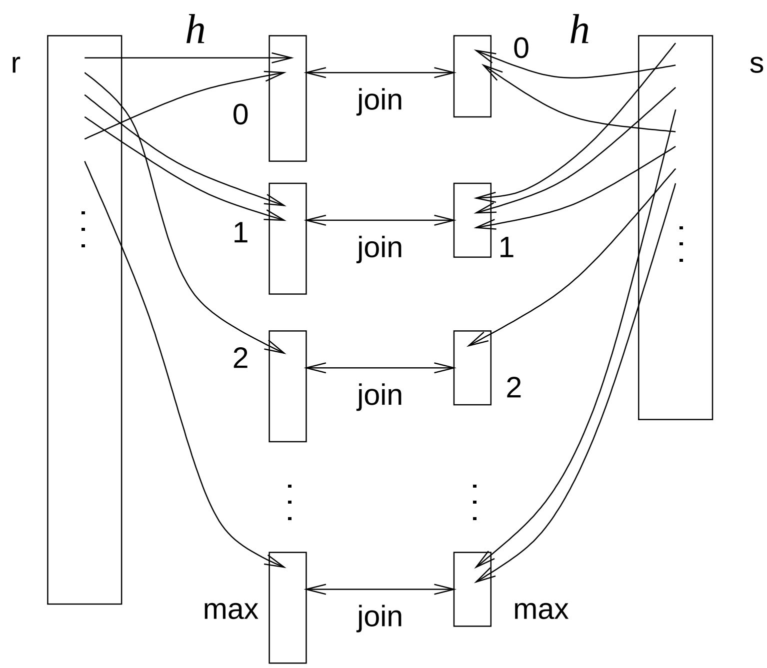
Merge-Techniken II

Aufwand:

- alle Tupel haben den selben X-Wert: $O(n_r \times n_s)$
- X Schlüssel von R oder S: $O(n_r \log n_r + n_s \log n_s)$
- bei vorsortierten Relationen sogar: $O(n_r + n_s)$

Hash-Verbund

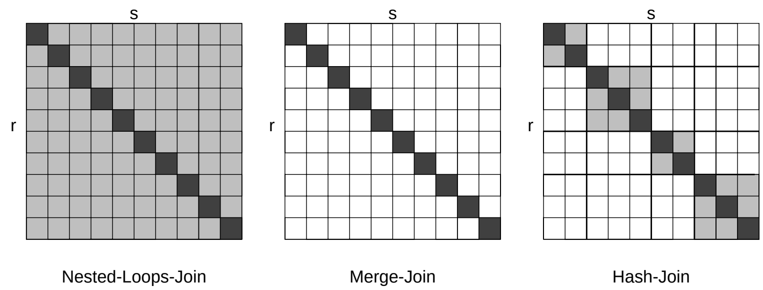
- Tupel aus r und s über X in gemeinsame Datei mit k Blöcken (*Buckets*) "gehasht"
- Tupel in gleichen Buckets durch Verbundalgorithmus verbinden



Hash-Verbund (2)

for each t_r **in** r **do**
   begin
      $i := h(t_r(X))$;
      $H_i^r := H_i^r \cup t_r(X)$;
   end;
for each t_s **in** s **do**
   begin
      $i := h(t_s(X))$;
      $H_i^s := H_i^s \cup t_s(X)$;
   end;
for each k **in** $0 \ldots$ **max do**
   $H_k^r \bowtie H_k^s$;

Vergleich der Techniken



Übersicht über Komplexität der Operationen

- Der Aufwand für die Selektion geht von konstantem Aufwand $O(1)$ bei einer Hash-basierten Zugriffsstruktur bis hin zu $O(n)$, falls ein sequentieller Durchlauf notwendig ist. In der Regel kann man von $O(\log n)$ als Resultat baumbasierter Zugriffspfade ausgehen.
- Der Aufwand für den Verbund reicht von $O(n + m)$ bei sortiert vorliegenden Tabellen bis zu $O(n * m)$ bei geschachtelten Schleifen.
- Die Komplexität der Projektion reicht von $O(n)$ (vorliegender Zugriffspfad oder Projektion auf Schlüssel) bis $O(n \log n)$ (Duplikateliminierung durch Sortieren).

Optimierungsarten

- *Logische Optimierung* nutzt nur algebraische Eigenschaften der Operationen, also *keine* Informationen über die realisierten Speicherungsstrukturen und Zugriffspfade. Beispiel: Entfernung redundanter Operationen.

  Statt exakter Optimierung: heuristische Regel, etwa Verschieben von Operationen derart, daß Selektionen möglichst früh ausgeführt werden

  $\rightarrow$ *algebraische Optimierung*.

Optimierungsarten II

- Die *interne Optimierung* nutzt Informationen über die vorhandenen Speicherungsstrukturen aus.

  Bei Verbunden kann die Reihenfolge der Verbunde nach Größe und Unterstützung der Relationen durch Zugriffspfade festgelegt werden. Bei Selektionen kann die Reihenfolge der Anwendung von Bedingungen nach der Selektivität von Attributen und dem Vorhandensein von Zugriffspfaden optimiert werden.

  Desweiteren wird in der internen Optimierung die Implementierungsstrategie einzelner Operationen ausgewählt.

14. Alternative Algorithmenkonzepte

deduktive Algorithmen

- basierend auf logischen Aussagen
- Programmiersprachen wie PROLOG

Aussagen und Aussageformen

- Aussage:
  Susi ist Tochter von Petra
- Aussageform mit Unbestimmten:
  X ist Tochter von Y
- Belegung:
  $$X \mapsto \text{Susi}; \quad Y \mapsto \text{Petra}$$
- atomare Formel:
  $$Tochter(X, Y)$$

Logik der Fakten und Regeln

- Alphabet
  - Unbestimmte $X, Y, Z, ...$
  - Konstanten $a, b, c, ...$
  - Prädikatensymbole $P, Q, R, ...$ mit Stelligkeit (analog Signatur von ADT)
  - logische Konnektive $\wedge$ und $\Longrightarrow$
- atomare Formeln: $P(t_1, .., t_n)$
- Fakten: atomare Formeln ohne Unbestimmte
- Regeln (α_i ist atomare Formel):
  $$\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_m \implies \alpha_0$$
  auch leere Prämissen möglich (entsprechen **`true`**)!

Beispiel

```
Fakten:
   1.  TOCHTER(Susi,Petra)
   2.  TOCHTER(Petra,Rita)
Regeln:
   3.  TOCHTER(X,Y) ∧ TOCHTER(Y,Z) ⟹ ENKELIN(X,Z)
```

Ableitung neuer Fakten

(vereinfachte Darstellung)

- finde Belegung der Unbestimmten einer Regel, so daß auf der linken Seite (Prämisse) bekannte Fakten stehen
- rechte Seite ergibt dann neuen Fakt

logische Regel: *modus ponens*

Beispiel II

```
Fakten:  1.  TOCHTER(Susi,Petra)
         2.  TOCHTER(Petra,Rita)
Regeln:  3.  TOCHTER(X,Y) ∧ TOCHTER(Y,Z) ⟹ ENKELIN(X,Z)
```

Belegung

```
X ↦ Susi, Y ↦ Petra, Z ↦ Rita
```

ergibt Fakt

```
ENKELIN(Susi,Rita)
```

Deduktiver Algorithmus

Ein *deduktiver Algorithmus* D ist eine Menge von Fakten und Regeln.

$F(D)$ sind alle direkt oder indirekt aus D ableitbaren Fakten.

Eine *Anfrage* γ ist eine Konjunktion von atomaren Formeln.

$$\gamma = \alpha_1 \wedge ... \wedge \alpha_m$$

Eine Antwort ist eine Belegung der Unbestimmten in γ, bei der aus allen α_i Fakten aus $F(D)$ werden.

Beispiel Addition

```
Fakten
   SUC(n,n+1)  für alle n ∈ ℕ
Regeln:
   (1)  ⟹  ADD(X,0,X)
   (2) ADD(X,Y,Z) ∧ SUC(Y,V) ∧ SUC(Z,W)  ⟹  ADD(X,V,W)
```

Addition: Anfragen

- `ADD(3,2,5)?` liefert **`true`**

  Ableitung: (1) mit X=3; (2) mit Belegung 3,0,3,1,4; (2) mit 3,1,4,2,5;

- `ADD(3,2,X)?` liefert $X \mapsto 5$
- `ADD(3,X,5)?` liefert $X \mapsto 2$
- `ADD(X,Y,5)?` liefert $(X, Y) \in \{(0,5),(1,4),(2,3),(3,2),(1,4),(5,0)\}$
- `ADD(X,Y,Z)?` liefert unendliches Ergebnis

Auswertung logischer Anfragen

- vollständiger Algorithmus würde Zeitrahmen dieser Vorlesung sprengen
- Idee:
  1. starte mit γ
  2. untersuche Belegungen die
     - einen Teil von γ mit Fakten gleichsetzen
     - einen Fakt aus γ mit einer rechten Seite einer Regel gleichsetzen
  3. wende passende Regeln "rückwärts" an (ersetze Konklusion durch Prämisse)
  4. entferne gefundene Fakten aus der Anfragemenge

  letzte Schritte wiederholen solange bis γ leer ist

Auswertung logischer Anfragen II

vollständige Lösung:

BACKTRACKING !

nur vereinfachter Formalismus: keine Negation; kein Erkennen von Endlosrekursionen, unendlichen Belegungsmengen etc.

Auswertung: Beispiel

add(3,2,5)

↓ (2)

add(3,y’,z’), suc(y’,2), suc(z’,5)

↓ y’ = 1 (Fakt)

add(3,1,z’), ~~suc(1,2)~~, suc(z’,5)

↓ z’ = 4 (Fakt)

add(3,1,4), ~~suc(4,5)~~

↓ (2)

add(3,y’’,z’’), suc(y’’,1), suc(z’’,4)

↓ y’’ = 0 (Fakt)
z’ = 3 (Fakt)

add(3,0,3)

↓ (1)

true

Auswertung: Beispiel 2

add(3,2,X)

↓ (2)

add(3,y’,z’), suc(y’,2), suc(z’,X)

↓ y’ = 1 (Fakt)

add(3,1,z’), ~~suc(1,2)~~, suc(z’,X)

↓ (2)

add(3,y’’,z’’), suc(y’’,1) suc(z’’,z’), suc(z’,X)

↓ y’’ = 0 (Fakt)

add(3,0,z’’)~~, suc(0,1),~~ suc(z’’,z’), suc(z’,X)

↓ z’’ = 3 (Fakt)

add(3,0,3), suc(3,z’), suc(z’,X)

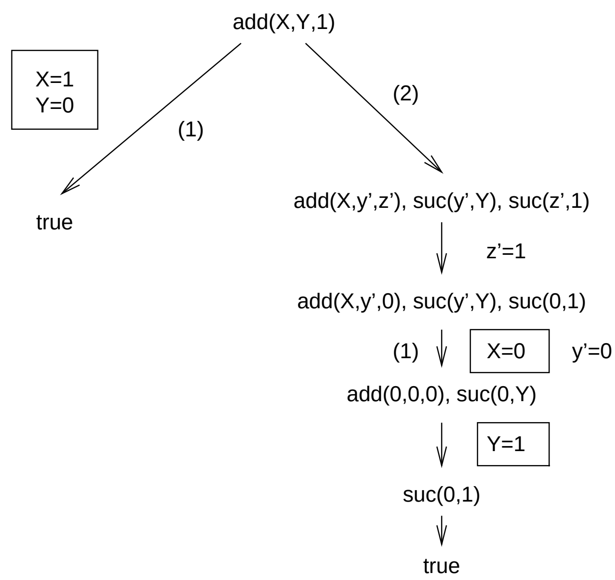
↓ (1) + z’ = 4

suc(3,4), suc(4,X)

↓ $\boxed{\text{X = 5}}$

true

Auswertung: Beispiel 3



A. Ergänzende und weiterführende Literatur

A.1. Grundlagen

- Starthilfe Informatik [ABCW98]
- Duden Informatik [Lek93]
- Einführungsbücher von Broy: [Bro98a, Bro98b]
- Einführungsbücher von Goos: [Goo97, Goo99]
- Skriptum Informatik: [AL98]
- Goldschlager / Lister: [GL90]

A.2. Programmieren und Java

- Goodrich und Tamassia: [GT98]

A.3. Spezielle Gebiete der Informatik

- Datenbanksysteme: [HS95, SH99]

Literaturverzeichnis

[ABCW98] Appelrath, H.-J.; Boles, ; Claus, V.; Wegner, : *Starthilfe Informatik*. Teubner, 1998.

[AL98] Appelrath, H.-J.; Ludewig, J.: *Skriptum Informatik — eine konventionelle Einführung*. Teubner, 4. Auflage, 1998.

[AU96] Aho, A. V.; Ullman, J. D.: *Informatik. Datenstrukturen und Konzepte der Abstraktion*. International Thomson Publishing, Bonn, 1996.

[Bro98a] Broy, M.: *Informatik. Eine grundlegende Einführung. Band 1: Programmierung und Rechnerstrukturen*. Springer, 2. Auflage, 1998.

[Bro98b] Broy, M.: *Informatik. Eine grundlegende Einführung. Band 2: Systemstrukturen und Theoretische Informatik*. Springer, 2. Auflage, 1998.

[CLR90] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.: *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[dB86] Bibliographischen Instituts, F. d. (Hrsg.): *Schülerduden. Die Informatik*. Bibliographisches Institut, Mannheim, 1986.

[Fri97] Friedl, J.: *Mastering Regular Expressions*. O'Reilly, 1997.

[GL90] Goldschlager, L.; Lister, A.: *Informatik. Eine moderne Einführung*. Hanser, 3. Auflage, 1990.

[Goo97] Goos, G.: *Vorlesungen über Informatik. Band 1: Grundlagen und funktionales Programmieren*. Springer, 2. Auflage, 1997.

[Goo99] Goos, G.: *Vorlesungen über Informatik. Band 2: Objektorientiertes Programmieren und Algorithmen*. Springer, 2. Auflage, 1999.

[GT98] Goodrich, M. T.; Tamassia, R.: *Data Structures and Algorithms in Java*. Wiley, 1998.

[HS95] Heuer, A.; Saake, G.: *Datenbanken — Konzepte und Sprachen*. International Thomson Publishing, Bonn, 1995.

[Lek93] Lektorat des BI-Wissenschafts-Verlags, (Hrsg.): *Duden Informatik*. B.I., 2. Auflage, 1993.

[SH99] Saake, G.; Heuer, A.: *Datenbanken — Implementierungstechniken*. MITP-Verlag, Bonn, 1999.

[Wei98] Weiss, M.: *Data Structures & Algorithm Analysis in Java*. Addison Wesley, 1998.

Sachindex