GRUNDLAGEN DER INFORMATIK

Stoffzusammenfassung

Inhalt

Datenstrukturen und (Sortier-)Algorithmen

Grundlegende Datenstrukturen

Allgemeine Eigenschaften von Daten

„Interpretierbare Darstellung von Informationen in einer formalisierten Weise, geeignet für Kommunikation, Interpretation oder Verarbeitung“ [ISO/IEC 2382-1]

- Es existieren Basisdaten wie Zeichen, Wahrheitswerte (true, false), ganze Zahlen oder Gleitpunktzahlen.
- Daten können Beziehungen untereinander haben, wie z.B. Listen, hierarchische Datenstrukturen wie Bäume usw.

Algorithmen hängen in starkem Maße von der gewählten Datenstruktur ab. Durch geeignete Datenstrukturen können Algorithmen übersichtlicher und effizienter werden.

Basis-Datentypen

Ein Basistyp ist die Klassifizierung von Werten gleicher Art, wie z.B. ganzen Zahlen, Gleitpunktzahlen oder Zeichen. Basis-Datentypen sind:

char: Menge der Zeichen,
int: Menge der ganzen Zahlen, die im Rechner darstellbar sind,
float: Menge der darstellbaren Gleitkommazahlen mit einfacher Genauigkeit,
double: Menge der darstellbaren Gleitkommazahlen mit doppelter Genauigkeit,
Array: Zusammenfassung von zusammengehörigen Daten des gleichen Typs.

Datenstruktur

Unter einer Datenstruktur versteht man den Datentyp zusammen mit der Menge von Operationen, die auf diesem Datentyp erlaubt sind. Beispiele für Operationen der verschiedenen Datentypen:

- Ganze Zahlen (integer)
  - Grundrechenarten: +, -, *
  - Division mit Rest und Modulo
  - Vergleichsoperatoren: <, >, ==
- Gleitkommazahlen (single, double)
  - Grundrechenarten: +, -, *, /
  - Vergleichsoperatoren: <, >, ==
- Boolean
  - Logische Operatoren: NOT, AND, OR, XOR, NOR, NAND
  - Vergleichsoperatoren: ==, ~=
- Zeichen
  - Vergleichsoperatoren: <, >, ==
  - Konvertierung in INTEGER

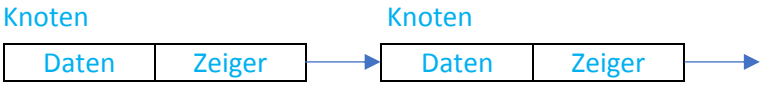
Gleitkommazahlen soll man nicht auf Gleichheit prüfen, denn durch den Floating point sind zwei gleiche Zahlen z.B. a=69.20 und (b=0.62 + c=68.58) unterschiedlich gerundet.

Die Wahl der richtigen Datenstruktur ist von fundamentaler Wichtigkeit bei der Lösung einer Aufgabe. Zur Datenstruktur gehören immer die Operationen, die auf ihr ausgeführt werden sollen. Diese Operationen implizieren Algorithmen, die für diese verwendet werden. Dieses Konzept bezeichnet man als *abstrakten Datentyp*.

Verkettete Listen

Eine verkettete Liste (linked list) ist eine Folge von Elementen, die dynamisch währen des Programmablaufs verlängert bzw. verkürzt werden kann. Anders als bei Arrays ist bei verkettete Listen nicht garantiert, dass die einzelnen Elemente hintereinander im Speicher liegen. Dies hat Geschwindigkeitsverluste beim Abarbeiten der Liste mit sich zu führen.

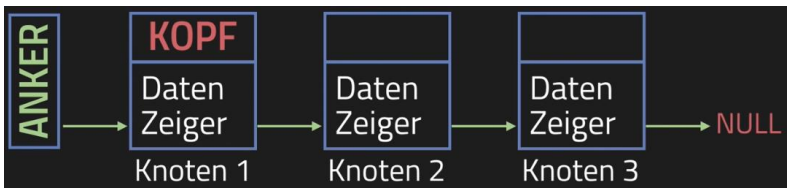
Eine Liste ist prinzipiell wie folgt aufgebaut:



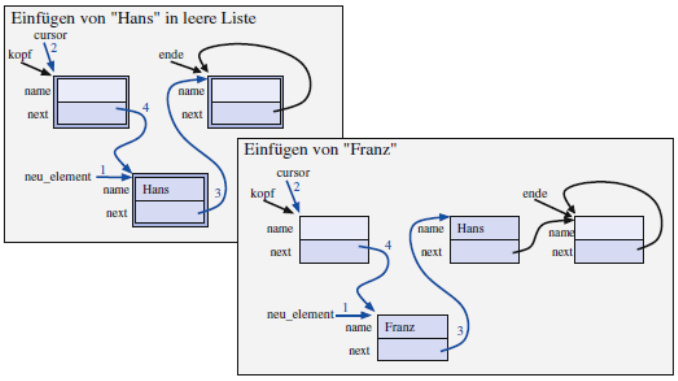
Um also eine Verbindung zwischen den Einträgen der Liste herstellen zu können, muss man einen Zeiger auf das nächste oder auch auf das vorherige Element halten. Dies wird so realisiert, dass sie die Speicheradresse des Elements enthalten, auf die sie zeigen.

Einfach verkettete Listen

Die einfach verkettete Liste besteht aus Anker („Cursor“), Kopf und Null. Der Anker ist der Anfangsknoten und zeigt auf den ersten Knoten (Kopf). Der Kopf kennt nur die Position des zweiten Knotens, die zweite nur die Position des dritten und so weiter. Nach dem letzten Knoten – dies kann natürlich auch der Kopf sein – folgt die Null. Diese zeigt an, dass die Liste zu Ende ist.



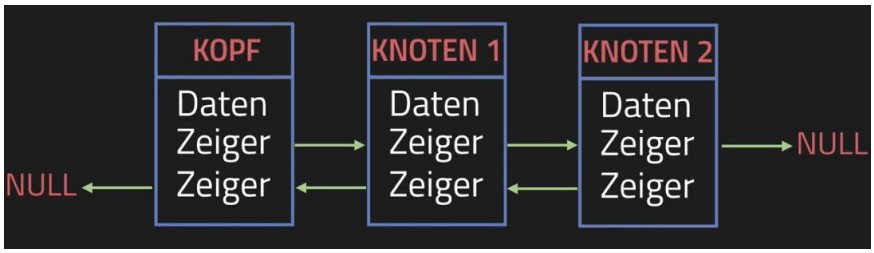
Der Vorteil dieser Liste ist, dass die Größe nicht bekannt sein muss. Die Befehle dieser Liste sind „Einfügen“ und „Löschen“ von Elementen und „Durchwandern“ der Liste (Suchen).



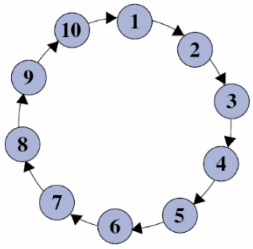
Beim Einfügen muss man zuerst dem neuen Element die Adresse des Nachfolgeelements geben, um Anschließend des Elements davor die Adresse des Neuen zu geben.
Beim Löschen muss man dem Element vor dem zu Löschenden die Adresse des Elements danach geben, bevor man dieses löscht.

Doppelt verkettete Listen

Die doppelt verkettete Liste besitzt zwei Zeiger. Einer verweist auf das vorherige Element, das andere auf das Nachfolgende. Ist kein Element verfügbar, zeigt jeweils ein Zeiger auf die Null. Der Vorteil dabei ist, dass das Löschen, Einfügen und die Suche schneller ist, als bei der einfach verketteten Liste.



Kreisförmig verkettete Listen



Bei der kreisförmig verketteten Liste wird der Zeiger des letzten Elements wieder auf den „Kopf“ gerichtet. Ein bekanntes Beispiel dieser Liste ist das Josephus-Problem. Im Jahre 67 n. Chr. mussten sich 40 Soldaten selbst richten um der Sklaverei zu entkommen. Josephus schlug die decimatio (Aussonderung jedes zehnten) vor und wollte sich und seinen Freund retten, indem sie sich an eine gewisse Stelle des Kreises stellten.

Vor- und Nachteile von Listen gegenüber Arrays

Vorteile:
Schnelles Einfügen und Löschen von Elementen, dynamisch, Speicher muss nicht zusammenhängen (bedingter Vorteil)

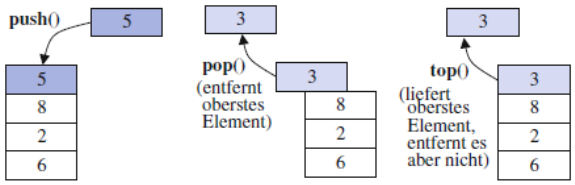
Nachteile:
Kein direkter Zugriff auf i. Element, Administration ist erforderlich, Cache unfreundlich (durch zu vieles einfügen und löschen wird der Vorgang langsamer; durch vieles Rumspringen -> viele „Cache misses“)

Stack (Stapel/Kellerspeicher)

Bei einem Stack handelt es sich um eine dynamische Datenstruktur, für die Operationen wie folgt definiert sind:

push(): legt ein Element an oberster Stelle auf dem Stack ab.
pop(): entfernt das oberste Element aus dem Stack.
top(): liefert oberstes Element des Stacks, entfernt dieses aber nicht.

Da diese Datenstruktur immer das zuletzt eingefügte Element vom Stapel als erstes entnimmt bezeichnet man dieses als LIFO (Last-In-First-Out).



Queue (Warteschlange)

Queue ist eine dynamische Datenstruktur, die dem FIFO (First-In-First-Out)-Prinzip folgt – deshalb wird sie auch Warteschlange genannt. Seine Operationen sind wie folgt definiert:

put(): fügt ein Element am Ende der Warteschlange hinzu.
get(): entnimmt ein Element am Ende der Warteschlange.

Bäume

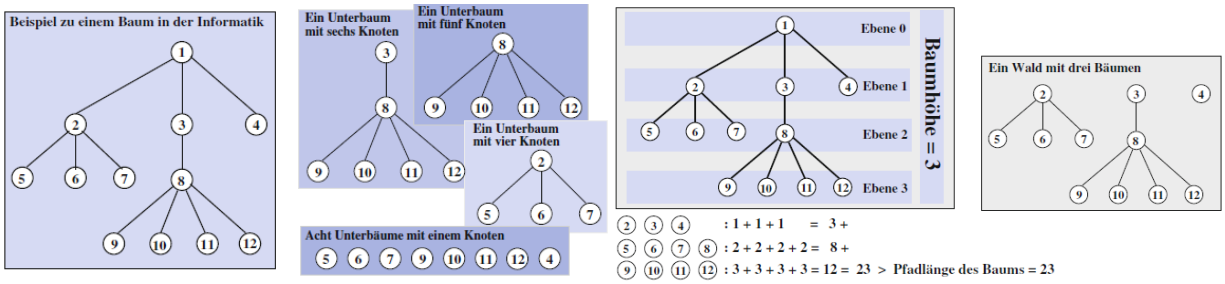
Bäume (trees) gehören zu den fundamentalen Datenstrukturen der Informatik. Bei Bäumen handelt es sich in gewisser Weise um eine zweidimensionale Erweiterung von verketteten Listen.

Folgende Definitionen stellen einen Baum auf:

- **Baum:** nicht-leere Menge von Knoten und Kanten, ohne Zyklen (=geschlossene Pfade)
- **Knoten:** Objekte mit Informationen
- **Kante:** gerichtete Verbindung zweier Knoten
- **Pfade:** Folge unterschiedlicher Knoten, die durch Kanten miteinander verbunden sind
- **Wurzel:** Ursprungsknoten

Gibt es zwischen der Wurzel und einem Knoten mehr als einen oder auch keinen Pfad, spricht man nicht von einem Baum, sondern um einen sogenannten gerichteten Graphen. Die Spezialform eines gerichteten Graphen ist der ungerichtete Graph. Dieser hat stets eine Hin- und Rückkante.

- Jeder Knoten (außer der Wurzel) hat einen *direkten Vorgänger* (parent) und (einen oder mehrere) *direkte Nachfolger* (child(ren)).
- Knoten ohne Nachfolger nennt man Endknoten, äußere Knoten oder Blätter (leafs).
- Entsprechend des Familienstammbaums spricht man auch von Großelternknoten und Geschwisterknoten.
- Innere Knoten sind Knoten mit mindestens einem Nachfolger.
- Jeder Knoten ist die Wurzel eines *Unterbaums*.
- Die Ebene eines Baums ist die Anzahl von Knoten auf dem Pfad, wobei die Wurzel Ebene 0 ist.
- Die Baumhöhe ist die Anzahl der Ebenen vom längsten Pfad.
- Entfernt man z.B. die Wurzel (mit seinen Kanten) ergeben sich drei Bäume, die man auch als *Wald* zusammenfasst.
- Geordnete Bäume sind Bäume, bei denen die Reihenfolge der direkten Nachfolger bei jedem Knoten festgelegt ist. Ist diese Reihenfolge nicht vorgeschrieben spricht man von einem ungeordneten Baum.
- Falls jeder Knoten nur eine bestimmte Anzahl n von direkten Nachfolgern haben darf, spricht man von einem n-ären Baum (z.B. Binärbaum).

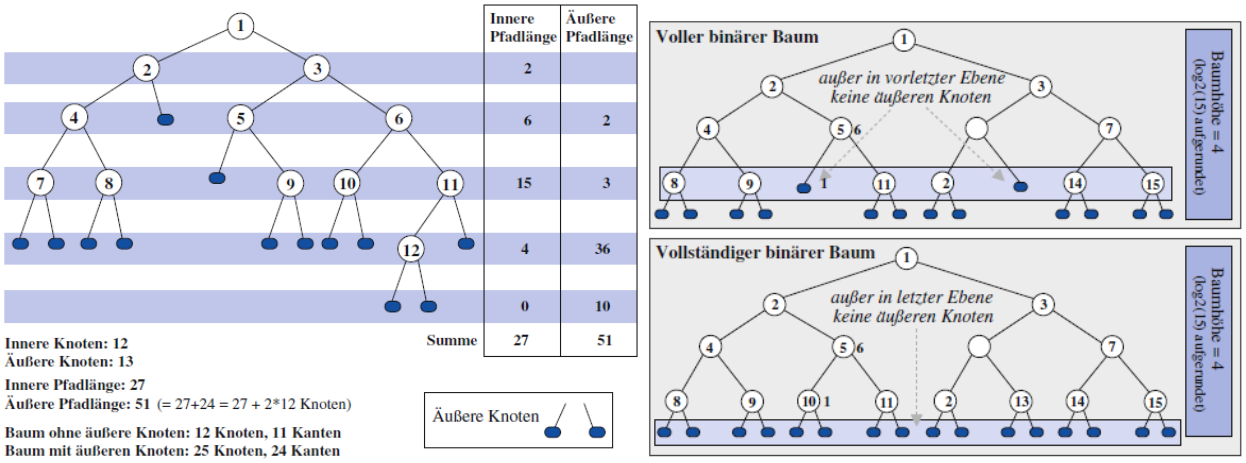


Binäre Bäume

Der einfachste Typ eines n-ären Baums ist der Binärbaum, für den Folgendes gilt:

- Es ist ein geordneter Baum.
- Er besitzt zwei Typen von Knoten
  - Innerer Knoten: haben immer einen oder maximal zwei Nachfolger, die man als linken und rechten Nachfolger bezeichnet
  - Äußerer Knoten: sind Knoten ohne Nachfolger (Blätter/leafs).
- Ein leerer Binärbaum besteht nur aus einem äußerem und keine inneren Knoten.

- Ein voller Binärbaum besitzt in der letzten und vorletzten Ebene Blätter
- Ein vollständiger Baum besitzt in der letzten Ebene ausschließlich Blätter.



Binärbäume finden in der Informatik häufig Anwendung, da sie die Vorteile von Arrays (schnelle Zugriffe auf bestimmte Elemente) und Listen (leichtes Einfügen bzw. Entfernen von Elementen) vereinigen. Binäre Bäume lassen sich am besten mittles Rekursion verwalten:

Ein binärer Baum ist entweder die Wurzel eines anderen binären Baum oder ein äußerer Knoten.

Fun-Facts:

- Ein binärer Baum mit n inneren Knoten hat n+1 Blätter
- Ein Baum mit n Knoten hat n-1 Kanten
- Die äußere Pfadlänge x mit n inneren Knoten ist immer um 2n größer als die innere Pfadlänge y (y=x+2n)
- Die Höhe eines vollen Binärbaums mit n inneren Knoten ist die nächste ganze Zahl zu $\log_2(n)$ bzw. $\frac{\log(n)}{\log(2)} = \frac{\ln(n)}{\ln(2)}$

Baum traversieren

Traversieren bezeichnet die systematischen Untersuchung der Knoten in einer bestimmten Reihenfolge. Es gibt verschiedene Möglichkeiten den Baum zu durchsuchen (Dabei ist „durchlaufen der Teilbäume“ l und r als rekursiver Aufruf zu verstehen):

- **pre-order** bzw. **Hauptreihenfolge** (N-l-r)
  Zuerst wird die Wurzel N betrachtet und anschließend der linke l, schließlich der rechte Teilbaum r durchlaufen.
- **post-order** bzw. **Nebenreihenfolge** (l-r-N)
  Zuerst wird der linke l, danach der rechte Teilbaum r durchlaufen und schließlich die Wurzel N betrachtet.
- **In-order** bzw. **symmetrische Reihenfolge** (l-N-r)
  Zuerst wird der linke l Teilbaum durchlaufen, danach wird die Wurzel N betrachtet und schließlich wird der rechte Teilbaum r durchlaufen.
- **level-order** bzw. **breadth-first (Breitensuche)**
  Beginnend von der Wurzel werden die Ebenen von links nach rechts durchlaufen.

Beim Löschen von Kanten wird unterschieden:

- Handelt es sich um ein Blatt, so kann dieses einfach entfernt werden.
- Hat die Kante ein Kind, so rutscht das Kind an die Stelle der gelöschten Kante.
- Hat die Kante zwei Kinder, so kann die Löschung sowohl über den linken als auch den rechten Teilbaum gehen. Es ist ein Abstieg bis zu einem Halbblatt (Kante mit einem Kind) unvermeidlich.

Beim Einfügen einer Kante, wird diese zum unmittelbaren Nachbar des mittelbaren Halbblatts.

Balancierte Binärbäume

Der Aufwand in einem Binärbaum zu suchen, wächst mit der Länge des Pfades von der Wurzel bis zu dem gesuchten Element. Bei einem balancierten Binärbaum besitzen alle Blätter in etwa die gleiche Tiefe, was bedeutet, dass der Binärbaum bei n Knoten eine Höhe von $\log_2(n+1)$ hat. Bei einem balancierten Baum sind somit maximal $\log_2(n+1)$ Vergleiche notwendig um ein Element zu finden.

Durch zu viel Löschen bzw. Hinzufügen von Elementen in einem Binärbaum, ohne dementsprechenden Maßnahmen, kann der Binärbaum (im schlimmsten Fall: zu einer Liste) degeneriert werden. Dies hat die Folge, dass der Suchaufwand enorm in die Höhe steigt.

Komplexität von Algorithmen und O-Notation

Bevor wir weitergehen zu den Sortieralgorithmen, ein kleiner Exkurs in die Komplexität dieser. Probleme, die in Wissenschaft, Technik und Wirtschaft auftreten, erfordern effiziente Algorithmen, die Lösungen innerhalb weniger Stunden oder sogar innerhalb weniger Millisekunden erfordern. Um zu beurteilen, ob Programme diesen Anforderungen genügen, muss man die Algorithmen hinsichtlich ihres Bedarfs and Zeit und Speicherplatz analysieren.

Zeitaufwand

Beim kubischen Algorithmus werden drei ineinander geschachtelten Schleifen verwendet. Da die Schleifen sich immer wieder selbst aufrufen müssen ist der Zeitaufwand n^3. Dies bedeutet, wenn der Algorithmus für 2 Zahlen aufgerufen wird, benötigt dieser $2^3=8$ mal solange um ans Ergebnis zu kommen.

Der Quadratische Algorithmus besitzt zwei ineinander verschachtelte Schleifen. Dies bedeutet, dass der Zeitaufwand n^2 ist. Für unser Beispiel würde das Ergebnis nach 4 Zeiteinheiten festsehen.

Bei dem linearen Algorithmus ist nur eine Schleife in Verwendung, was einen Zeitaufwand von n bedeutet.

Der Optimalfall wäre eine Konstante. Egal wie viele Zahlen benötigt werden, das Ergebnis wird stets nach der selben Zeit geliefert.

Wenn man eine Zahl aufruft, mit einer immer kleiner werdenden Zahl vergleicht, danach die Ursprungszahl um 1 erhöht und diese Schritte wiederholt, – so wie es bei einer einfachen Primzahltestung der Fall ist – ist dies ein exponentieller Algorithmus mit einem Zeitaufwand von x^n, wobei x eine beliebige natürliche Zahl ≥ 0 ist.

Speicherplatzaufwand

Ein geringerer Zeitaufwand beschreibt nicht unbedingt einen geringeren Speicherplatzaufwand. Benützt man z.B. einen Algorithmus mit mehreren Schleifen, jedoch ohne Hilfsarray, so ist zwar der Zeitaufwand $n^{\text{Schleifenanzahl}}$, der Speicherplatzaufwand jedoch proportional n. Benützt man vergleichsweise einen linearen Algorithmus, kann dieser, je nachdem wie viele Arrays benutzt werden, die Platzkomplexität exponentiell werden.

Klassifikation von Algorithmen

Typische Komplexitätsfunktionen zu Algorithmen

1	*konstant*	Jede Anweisung eines Programms wird höchstens einmal ausgeführt. Dies ist der Idealzustand für einen Algorithmus.
$\log n$	*logarithmisch*	Speicher- oder Zeitverbrauch wachsen nur mit der Problemgröße *n*. Die Basis des Logarithmus wird häufig 2 sein, d. h. vierfache Datenmenge verursacht doppelten Ressourcenverbrauch, 8-fache Datenmenge verursacht 3-fachen Verbrauch und 1024-fache Datenmenge 10-fachen Verbrauch.
n	*linear*	Speicher- oder Zeitverbrauch wachsen direkt proportional mit der Problemgröße *n*.
$n \log n$	$n \log n$	Der Ressourcenverbrauch liegt zwischen *n* (*linear*) und n^2 (*quadratisch*).
n^2	*quadratisch*	Speicher- oder Zeitverbrauch wachsen quadratisch mit der Problemgröße. Solche Algorithmen lassen sich praktisch nur für kleine Probleme anwenden.
n^3	*kubisch*	Speicher- oder Zeitverbrauch wachsen kubisch mit der Problemgröße. Solche Algorithmen lassen sich in der Praxis nur für sehr kleine Problemgrößen anwenden.
2^n	*exponentiell*	Bei doppelter, dreifacher und 10-facher Datenmenge steigt der Ressourcenverbrauch auf das 4-, 8- bzw. 1024-fache. Solche Algorithmen sind praktisch kaum verwendbar.

Neben diesen grundlegenden Komplexitätsformen existieren weitere Zwischenformen, wie z.B.:

$n^{\frac{3}{2}}$	Algorithmus mit n^2 Speicher- und n^3 Zeitverbrauch (bei großem *n* näher bei $n \log n$ als bei n^2).
$n \cdot \log^2 n$	Algorithmus, der ein Problem zweistufig in Teilprobleme zerlegt (bei großem *n* näher bei $n \log n$ als bei n^2).

In der Informatik wird meist als Basis für den Logarithmus die Zahl 2 hergenommen, welche mit lg bezeichnet wird ($\log_2(n)=\lg(n)$).

Die O-Notation

Bezeichnet man die Laufzeit eines Programm mit t(n), so ist es offensichtlich, dass mit steigendem n der Koeffizient der höchsten Potenz von n in t(n) zunehmend an Bedeutung für die Laufzeit gewinnt. Ist z.B. $t(n^3+20n^2+n+8)$, wäre die O-Notation $O(n^3)$ (sprich: groß-O von). Die O-Notation ersetzt den Ausdruck „ist proportional zu“.

Elementare Sortieralgorithmen

Link für eine Visualisierung der im Folgenden vorgestellten Algorithmen. https://visualgo.net/de/sorting

Definitionen:

---

*Ein Algorithmus ist eine detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Problems, d. h. eine Vorschrift zur Lösung einer Aufgabe, die **präzise**, **in endlicher Form dargestellt** und **effektiv** ausführbar ist.*

Eine zufällige Anordnung von Zahlen nennt man Permutation.

Bubble-Sort

Der Bubble-Sort ist der bekannteste und ein sehr einfacher Sortieralgorithmus. Bei diesem werden zwei benachbarte Zahlen direkt verglichen und eventuell ausgetauscht, damit links die kleinere und rechts die größere Zahl steht. Somit wandert beim i-ten Durchlauf das i-größte Element nach hinten bzw. das i-kleinste nach vorne. Der Name dieses Algorithmus stammt durch das Aufsteigen einer Gasblase in einer Flüssigkeit (Bubble). Wird bei einem Durchlauf keine Zahl getauscht ist das Sortieren fertiggestellt.

2 8 9 4 1 | 2 8 9 4 1 | 2 8 9 4 1 | 2 8 4 9 1

2 8 4 1 9 | 2 8 4 1 9 | 2 4 8 1 9 | 2 4 1 8 9

2 4 1 8 9 | 2 4 1 8 9 | 2 1 4 8 9 | 2 1 4 8 9

2 1 4 8 9 | 1 2 4 8 9 | 1 2 4 8 9 | 1 2 4 8 9

Code:

```
void bubble_sort(int n, int z[]) {
      for (int i=n-1; i>0; i--) {
            for (int j=0; j<i-1; j++)
                  if (z[j] > z[j+1]) {
                        int t=z[j];
                        z[j]=z[j+1];
                        z[j+1]=t;
                  }
      }
}
```

Im besten Fall ist die Komplexität O(n)=n, im worst case ist sie $O(n)=n^2$.

Insertion-Sort

Beim Insertion-Sort nimmt das Programm der Reihe nach eine Zahl aus der Kette und vergleicht diese mit den Vorherigen und fügt diese anschließend in die passende Stelle ein.

8 2 9 4 1

2 8 9 4 1

2 4 8 9 1

1 2 4 8 9

Code:

```
void insert_sort(int n, int z[]) {
      for(inti=1; i<n; i++)
            for(int j=i; j>0 && z[j]<z[j-1]; j--) {
                  int t=z[j];
                  z[j]=z[j-1];
                  z[j-1]=t;
            }
}
```

Im besten Fall O(n)=n, im schlechtesten: $O(n)=n^2$

Selection-Sort

Der Selection-Sort vergleicht die Zahl der Reihe nach mit sich selbst. Wenn eine Zahl kleiner ist, wird diese in den Speicher gelegt. Der Zeiger läuft die Reihe jedoch noch durch und falls eine Zahl kleiner ist, als die im Speicher, so wird diese ausgetauscht. Ist die Zahlenreihe fertig durchgegangen, werden die erste Zahl und die Zahl im Speicher ausgetauscht.

8 13 7 20 4 6 14 1 19

1 13 7 20 4 6 14 8 19

1 4 7 20 13 6 14 8 19

1 4 6 7 20 13 14 8 19

1 4 6 7 8 13 14 20 19

1 4 6 7 8 13 14 19 20

Die Komplexität ist $O(n^2)$.

Quick-Sort

Der Quick-Sort ist der am häufigsten verwendete Sortieralgorithmus. Die Vorteile dieses Algorithmus ist einerseits die Geschwindigkeit und andererseits der geringe Speicherbedarf, da er die Daten im sortierenden Array direkt nur unter Zuhilfenahme eines kleinen Hilfs-Stack sortiert. Der Quick-Sort arbeitet nach dem Prinzip „Teile und Herrsche“.

- Das Array wird in zwei nicht-leere Teilarrays zerlegt, so dass alle Elemente im ersten Teilarray kleiner als alle in Teilarray zwei sind.
- Die Funktion, die für die Zerlegung des Arrays in Teilarrays zuständig ist, liefert dabei den Index des sogenannten Pivot-Elements, das die Trennstelle zwischen den beiden Teilarrays ist.
- Die Teilarrays werden nun ihrerseits wieder nach dem gleichen Verfahren durch rekursive Aufrufe des Quicksort sortiert.

Code:

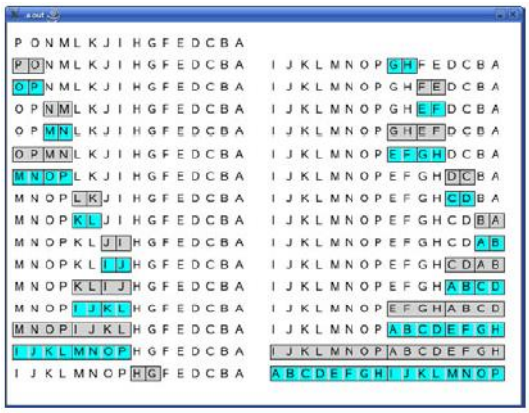
```
def partition(A, lo, hi):
      pivot = A[lo]
      i= lo -1
      j = hi + 1
      while True:
            while True:
                  j = j -1
                  if A[j] <= pivot:
                        break
            while True:
                  i= i+ 1
                  if A[i] >= pivot:
                        break
            if i< j:
                  A[i], A[j] = A[j], A[i]
            else:
                  return j

def quicksort(A, lo, hi):
      if lo < hi:
```

```
i= partition(A, lo, hi)
quicksort(A, lo, i)
quicksort(A, i+ 1, hi)
```

Die Komplexität im Durchschnittsfall ist O(n*log(n)), im schlechtesten Fall ist sie $O(n^2)$, da der Algorithmus rekursiv arbeitet.

Merge-Sort

Während der Quicksort die Datenmenge immer rekursiv in zwei Teilarrays zerlegt, die er dann sortiert, geht der Mergesort umgekehrt vor, indem er rekursiv zwei bereits sortierte Teilarrays mischt. Der Vorteil ist, dass das die Laufzeit im günstigsten sowohl auch im ungünstigsten Fall dieselbe ist. Der größte Nachteil dabei ist, dass man einen zu n proportional zusätzlichen Speicherplatz benötigt. Jedoch sollte man, wenn genügend Speicherplatz vorhanden ist, den Mergesort vor dem Quicksort ziehen.



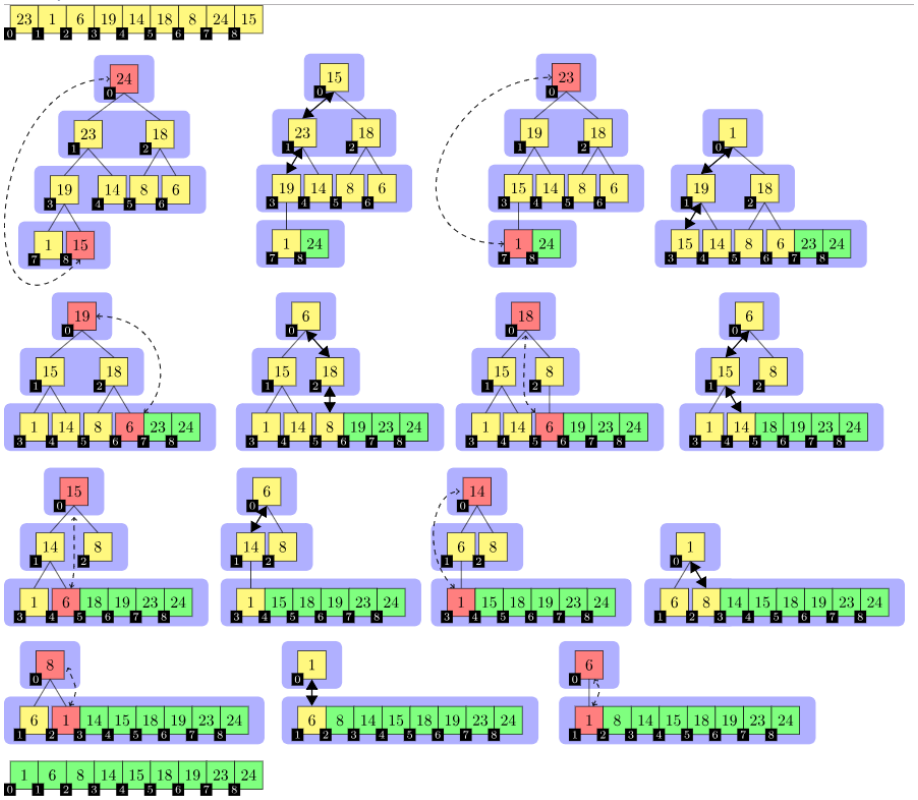
Grau: vor dem Sortiervorgang
Cyan: nach dem Sortiervorgang

Code:

```
void merge(int z[], int l, int m, int r) {
      int i, j, k;
      int h[] = …;
      for (i=m+1; i>l; i--)
            h[i-1] = z[i-1];
      for (j=m; j<r; j++)
            h[r+m-j] = z[j+1];
      for (k=l; k<=r; k++)
            z[k] = (h[i] < h[j]) ? h[i++] : h[j--];
}

void merge_sort(int z[], int l, int r) {
      if (l < r) {
            int mitte= (l+r)/2;
            merge_sort(z, l, mitte);
            merge_sort(z, mitte+1, r);
            merge(z, l, mitte, r);
      }
}
```

Die Komplexität ist im günstigsten und ungünstigsten Fall O(n*log(n)).

Heap-Sort



Beispiel zur Abarbeitung eines Heapsorts.

Die Komplexität des Heapsorts ist O(n*log(n)).

Automaten

Allgemeines

Automaten sind in der Informatik einfache Modelle von zustandsorientierten Maschinen, die sich nach bestimmten Regeln, dem Programm, verhalten.
Programme, die ein Programmierer erstellt, bezeichnet man als Quellprogramme (Sourcen), die in die für einen Menschen nicht lesbaren Maschinensprache der jeweiligen Prozessors übersetzt werden müssen.

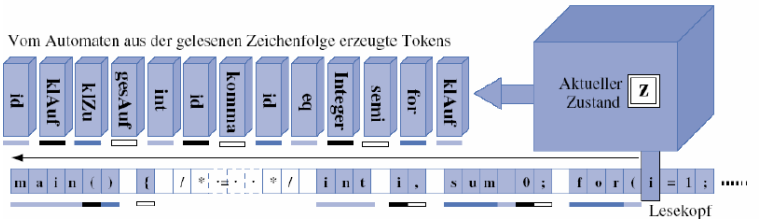
Alphabet einer Sprache: Als Alphabet einer Sprache bezeichnet man Zeichen (Groß-, Kleinbuchstaben, Zahlen und Sonderzeichen), die eine Programmiersprache akzeptiert.

Wörter einer Sprache: Aus diesem Alphabet werden die Wörter definiert, die zur Sprache gehören. Z.B.: Schlüsselwörter (**for**, **while**, **if**,...), Sonderzeichen (**+**, **-**, *****, **<**, **>**,...), Benutzerdefinierte Bezeichner (`zaehler`, `x12`, `kinder_zahl`,...) und Konstanten (`425` (Integer), `3.14` oder `2.3e7` (Gleitpunktzahlen), `'z'` (Zeichen), `„Guten Tag“` (String),...)

Der Übersetzer (Compiler oder Interpreter) überprüft als erstes, ob der Code den Regeln der entsprechenden Programmiersprache folgt. Dies geschieht mit lexikalischer und syntaktischer Analyse.

Lexikalisch: Code wird in einzelne Wörter getrennt, Kommentare und Trennzeichen („white spaces“ – Leerzeichen, Tabulator und Zeilenumbruch) werden entfernt und jedes Wort wird einem Token zugeteilt. Das Token gibt an, zu welcher Klasse von Wörtern das jeweilige Wort gehört.

```
main() { /* Summe aller ungeraden Zahlen zw. 1 und 100 */
    int i, sum=0;
    for (i=1; i<100; i = i+2)
        sum = sum + i;
}
```



Syntaktisch: Code wird auf die Regeln der Grammatik geprüft. Der Ausdruck **x = y + * z** entspricht zum Beispiel nicht der Grammatik der Programmiersprache C und ist somit unzulässig.

Reguläre Sprachen und endliche Automaten

Alphabet, Wort und Sprache

Ein Alphabet ∑ ist eine endliche Menge von Zeichen. Alphabete werden meist mit griechischen Buchstaben und Zeichen mit dem Anfangsbuchstaben des Alphabets (a, b, c,…) bezeichnet.

Ein Wort ist eine endliche Folge von Zeichen aus ∑. Die Menge über ∑ wird ∑* bezeichnet. Die Sonderform des Wortes ist das leere Wort ε. Wörter werden meist mit den Kleinbuchstaben u, v und w bezeichnet.

Konkatenation von Wörtern bedeutet das Zusammenfügen von zwei Wörtern u und v zu einem neuen Wort uv. Grundsätzlich gilt für jedes Wort: εu= uε=u. Notationen wie aw bzw. wa bedeuten, dass man dem Wort w das Zeichen a voranstellt bzw. anfügt.

Die Länge eines Wortes ist die Anzahl der Zeichen und wird mit $|u|$ bezeichnet. Es gilt $|uv|=|u|+|v|$ und $|\varepsilon|=0$.
Die Anzahl der Vorkommen von a in u wird mit $|u|_a$ beschrieben.
u heißt Präfix von w, wenn w=uv, u heißt Suffix von w, wenn w=vu und u heißt Infix von w, wenn w=vuv‘.

Sprache L über ∑ ist eine Menge von Wörtern über ∑.
Eine Sprache über ∑ ist somit eine Teilmenge von ∑*, wobei zu beachten ist, dass sowohl die leere Menge {} als auch ∑* (alle möglichen Wörter) selbst Sprachen sein können.

Reguläre Ausdrücke

Um komplizierte Sprachen aus einfacheren Sprachen konstruieren zu können, werden folgende Operatoren verwendet:

Vereinigung:	$L = L_1 \cup L_2$
Durchschnitt:	$L = L_1 \cap L_2$
Produkt (Konkatenation):	$L = L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}$
Potenzen:	$L^0 = \{\varepsilon\}; \quad L^{n+1} = L \cdot L^n$
Kleene-Stern:	$L^* = \{\varepsilon\} \cup \{L^n \mid n \in \mathcal{N}\}$
L mindestens einmal:	$L^+ = L \cdot (L^*)$
L einmal oder keinmal (optionales L):	$L? = L \cup \{\varepsilon\}$

- Ø (leer), {} und ε sind reguläre Ausdrücke

- A ist ein regulärer Ausdruck für jedes a ϵ Σ
- Sind r_1 und r_2 reguläre Ausdrücke, dann sind auch r_1+r_2, $r_1 \cdot r_2$ und r_1^* reguläre Ausdrücke.

* hat höhere Priorität als · und · höhere als +. Wie in der Mathematik kann man bei höheren Prioritäten Klammern setzen.

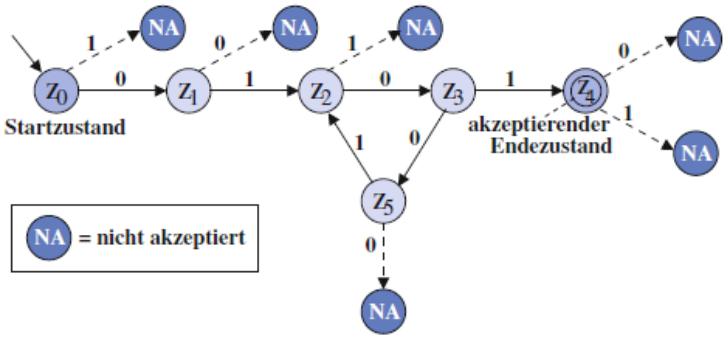
Anwendungsbereiche dieser Sprache sind z.B. Java, JavaScript und die Suchfunktion im Windows Explorer.

Endliche Automaten

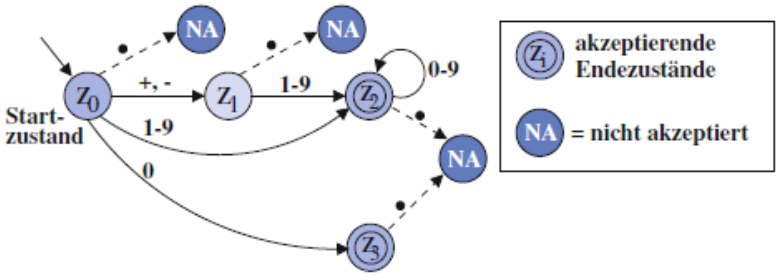
Ein endlicher Automat ist ein sehr einfaches Modell einer zustandsorientierten Maschine, die eine endliche Menge von inneren Zuständen hat. Er liest ein Eingabewort zeichenweise ein und führt bei jedem Zeichen, entsprechend seinem Programm, einen Zustandsübergang durch. Zusätzlich kann er bei jedem Zustandsübergang ein Ausgabesymbol ausgeben.
Ein endlicher Automat hat einen besonders gekennzeichneten Startzustand und eine Menge an Endzuständen. Erreicht der Automat nach einer Folge von Zustandsübergängen einen Endzustand, so bedeutet dies, dass das gelesene Wort in der Sprache vorhanden ist, also der Automat dieses akzeptiert hat.

Die Menge aller von einem Automaten akzeptierten Wörter bezeichnet man als die akzeptierte Sprache.

Beispiele:



Beispiel 1: Automat, der den Ausdruck 01(001)*01 erkennt. Es sind somit folgende Ausrücke akzeptiert: 0101, 0100101, 0100100101, usw.



Beispiel 2: Automat, der eine dezimale Konstante (ggf. mit Vorzeichen) erkennt. Der reguläre Ausdruck sieht wie folgt aus: [+-]?[1-9][0-9]*+0

Merksatz:

Jede Typ-3-Grammatik erzeugt eine reguläre Sprache und zu jeder regulären Sprache existiert ein endlicher Automat, der diese erzeugt.

Übungen: Postleitzahlen, Versicherungsnummern, Autokennzeichen,…

Zustand z_i	Gelesenes Zeichen 0	1	
0	1	−1	
1	−1	2	
2	3	−1	Folgezustände
3	5	4	
4	−1	−1	
5	−1	2	

Zustandsübergangstabelle zu Beispiel 1.

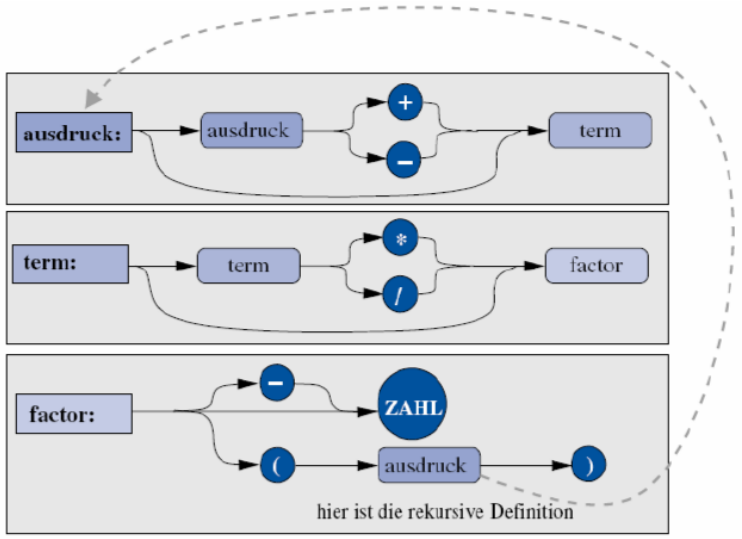
Ein endlicher Automat hat kein „Gedächtnis“. Er kann somit Aufgaben, die das Zählen von Klammern (zum Beispiel) erfordern, nicht (deterministisch) lösen.

Kontextfreie Sprachen und Kellerautomaten

Kontextfreie Grammatiken

Kontextfreie Sprachen können, wie erwähnt, Klammern zählen, da diese durch ihre kontextfreie Grammatik ein „Gedächtnis“ haben. Daher kommen diese Sprachen bei der Syntaxanalyse zum Einsatz. Um eine kontextfreie Grammatik zu beschreiben, existieren verschiedene Darstellungsmöglichkeiten – das Syntaxdiagramm und die Backus-Naur-Form (BNF; die Standardform einer Grammatik)

Syntaxdiagramm



- Ein *ausdruck* kann aus einem *ausdruck* bestehen, der mit + bzw. – mit einem *term* verknüpft ist. Alternativ kann ein *ausdruck* auch nur aus einem *term* bestehen.
- Ein *term* seinerseits kann wiederum aus einem *term* bestehen, der mittels * bzw. / mit einem *factor* verknüpft ist. Alternativ kann jedoch ein *term* nur aus einem *factor* bestehen.

- Ein *factor* kann entweder eine *ZAHL* sein, der optional ein Minuszeichen vorangestellt sein darf, oder aber wiederum ein geklammerter *ausdruck*, womit wir wieder, mittels Rekursion, beim ersten Syntaxdiagramm beim *ausdruck* sind.

Backus-Naur-Form (BNF)

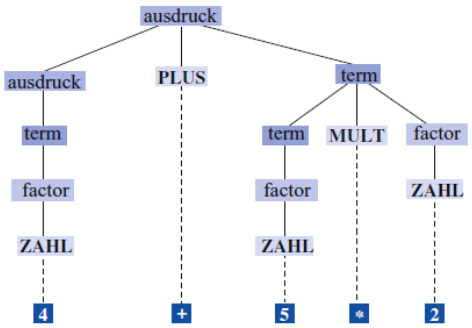
Die kontextfreie Grammatik hat in der BNF vier Komponenten:

1. Eine Menge von terminalen Symbolen: Terminale Symbole sind die Tokens (wie z.B. +(*OPERATOR*) oder *ZAHL* in der obenstehenden Grafik)
2. Eine Menge von nichtterminalen Symbolen: Nichtterminale Symbole sind Überbegriffe für Konstruktionen, die sich aus terminalen und/oder nichtterminalen Symbolen zusammensetzen, wie z.B. *ausdruck* oder *factor* in der obenstehenden Grafik. Sie sind sozusagen „Pseudosymbole“ bzw. Platzhalter.
3. Eine endliche Menge von Produktionen:
   a. Syntax: linker Teil → rechter Teil
   b. Semantik: rechter Teil ersetzt Linken
   c. Linker Teil: Nichtterminales Symbol
   d. Rechter Teil: Folge von terminalen und/oder nichtterminalen Symbolen
4. Startsymbol: Ein nichtterminales Symbol legt immer das Startsymbol fest, wie z.B. *ausdruck* in der obenstehenden Grafik.

Im folgenden Beispiel werden terminale Symbole **fett** und nichtterminale Symbole *kursiv* angegeben:

ausdruck ⟶ *ausdruck* **PLUS** *term*
| *ausdruck* **MINUS** *term*
| *term*
term ⟶ *term* **MULT** *factor*
| *term* **DIV** *factor*
| *factor*
factor ⟶ **ZAHL**
| **MINUS ZAHL**
| **KLAMAUF** *ausdruck* **KLAMZU**

In der folgenden Grafik wird der Ausdruck *4+5*2* mit Hilfe eines sogenannten *parse tree* hergeleitet:



Kontextfreie Grammatiken werden wie folgt gegliedert:

Griechische Kleinbuchstaben α, β, γ	für Satzformen
Kleinbuchstaben a, b, c, t	für Terminale
Großbuchstaben A, B, C, S	für Nichtterminale

Eine Satzform besteht aus einem Wort mit Terminalen und Nichtterminalen.

Kellerautomat

Da endliche Automaten nicht in der Lage sind kontextfreie Grammatiken zu erkennen, werden sogenannte *Kellerautomaten* (*Stackautomaten*, und engl. *Pushdown automaton (PDA)*) benötigt. Ein Kellerautomaten liest – wie ein endlicher Automat – die Eingabe Zeichenweise von links nach rechts, wobei, wenn möglich, das jeweilige Eingabezeichen sofort verarbeitet wird. Ist die Bearbeitung eines Zeichens wie z.B. einer öffnenden Klammer nicht sofort möglich, wird dieses Zeichen in einem eigenen Stack abgelegt und dessen Bearbeitung aufgeschoben, bis die dazugehörige schließende Klammer gelesen wird.
Die möglichen Aktionen hängen, wie beim endlichen Automaten, vom momentanen Eingabezeichen, vom momentanen Zustand und (anders als beim EA) vom Inhalt des Stacks ab. Immer nur das oberste Zeichen des Stacks ist Relevant.
Die Verarbeitung beginnt logischerweise mit einem leeren Stack.

Formal wird ein *nichtdeterministischer Kellerautomat (NPDA)* als ein 7-Tupel $K = (Z, \Sigma, \Gamma, \delta, z_0, \#, E)$ definiert, wobei Folgendes gilt:

Z *eine endliche Menge von Zuständen,*

Σ *ein endliches Eingabealphabet,*

Γ *ein endliches Stackalphabet,*

δ *eine nichtdeterministische Übergangsfunktion* $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow (Z \times \Gamma^*)$,

$z_0 \in Z$ der Startzustand,

\# das Anfangssymbol im Stack (für leeren Stack) und

$E \subseteq Z$ eine Menge von akzeptierenden Endezuständen.

Beispiele:

```
Gib die oeffnenden Symbole ein: ([{
Gib schliessende Symbole korrespondierend zu den oeffnenden ein: )]}
Gib den Ausdruck mit Klammern ein: (ab{(c)d)ee)x[(a)]
gelesen:  (.... push('(')   Stack: (,
gelesen: a ....             Stack: (,
gelesen: b ....             Stack: (,
gelesen:  {.... push('{')   Stack: {,  (,
gelesen:  {.... push('{')   Stack: {,  {,  (,
gelesen: c ....             Stack: {,  {,  (,
gelesen:  }.... pop()='{'   Stack: {,  (,
gelesen: d ....             Stack: {,  (,
gelesen:  }.... pop()='{'   Stack: (,
gelesen: e ....             Stack: (,
gelesen: e ....             Stack: (,
gelesen:  ).... pop()='('   Stack:
gelesen: x ....             Stack:
gelesen:  [.... push('[')   Stack: [,
gelesen:  (.... push('(')   Stack: (,  [,
gelesen: a ....             Stack: (,  [,
gelesen:  ).... pop()='('   Stack: [,
gelesen:  ].... pop()='['   Stack:
-----------------------------------------> akzeptiert
```

```
Gib die oeffnenden Symbole ein: ([{
Gib schliessende Symbole korrespondierend zu den oeffnenden ein: )]}
Gib den Ausdruck mit Klammern ein: (a[ab]a))
gelesen:  (.... push('(')   Stack: (,
gelesen: a ....             Stack: (,
gelesen:  [.... push('[')   Stack: [,  (,
gelesen: a ....             Stack: [,  (,
gelesen: b ....             Stack: [,  (,
gelesen:  ].... pop()='['   Stack: (,
gelesen: a ....             Stack: (,
gelesen:  ).... pop()='('   Stack:
gelesen:  ).... pop()='#'
-----------------------------------------> nicht akzeptiert
```

Bei den folgenden Ablaufbeispielen deckt der Automat die Sprache $L = \{a^n b^n | n > 0\}$ ab:

```
Gib die oeffnenden Symbole ein: a
Gib schliessende Symbole korrespondierend zu den oeffnenden ein: b
Gib den Ausdruck mit Klammern ein: aaabbb
gelesen: a .... push('a')   Stack: a,
gelesen: a .... push('a')   Stack: a, a,
gelesen: a .... push('a')   Stack: a, a, a,
gelesen: b .... pop()='a'   Stack: a, a,
gelesen: b .... pop()='a'   Stack: a,
gelesen: b .... pop()='a'   Stack:
----------------------------------------> akzeptiert
```

```
Gib die oeffnenden Symbole ein: a
Gib schliessende Symbole korrespondierend zu den oeffnenden ein: b
Gib den Ausdruck mit Klammern ein: abb
gelesen: a .... push('a')   Stack: a,
gelesen: b .... pop()='a'   Stack:
gelesen: b .... pop()='#'
----------------------------------> nicht akzeptiert
```

Kellerautomaten erkennen genau die kontextfreien Sprachen, sind somit mächtiger als die endlichen Automaten, aber weniger mächtig als die Turingmaschinen, welche z.B. auch rekursiv aufzählbare und kontextsensitive Sprachen erkennt.

Es gibt formale Sprachen, die von keinem Kellerautomaten erkannt werden können.

So kann z.B. die kontextsensitive Sprache $L = (a^n b^n c^n | n \geq 0)$ nicht von einem Kellerautomaten erkannt werden.
Hebt man die Einschränkung des Kellerautomaten auf, dass das Band nur in eine Richtung bewegbar ist und beim Stack nur das oberste Element zugreifbar ist, so ist es eine Turingmaschine.

Merksatz:

Jede Typ-2-Grammatik erzeugt eine kontextfreie Sprache und zu jeder kontextfreien Sprache gibt es einen Kellerautomaten, die diese erzeugt.

Eine Teilemenge dieser Typ-2-Grammatiken bildet die theoretische Grundlage für dies Syntax der meisten Programmiersprachen.

Chomsky-Hierarchie

Klasse	Grammatiken	Sprachen	Minimaler Automat	Zeit
Typ 0	uneingeschränkt	rekursiv	Turingmaschine	n.m.
Typ 1	kontextsensitiv	kontext-sensitiv	Linear beschränkte TM	$2^{O(n)}$
Typ 2	kontextfrei	kontextfrei	Kellerautomat	$O(n^3)$
Typ 3	regulär	regulär	Endlicher Automat	O(n)

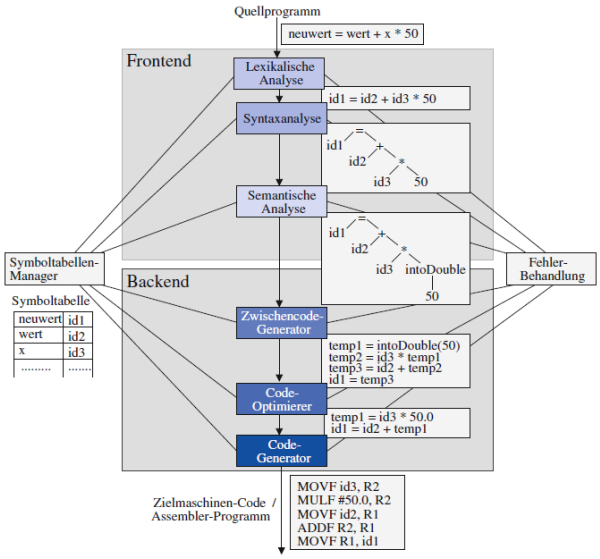
Typ 0: $L = \{a^n b^n c^n | n \geq 0\}$
Typ 2: $L = \{a^n b^n | n \geq 0\}$
Typ 3: $L = \{a^n | n \geq 0\}$

Letzter Merksatz:

Eine rekursive Sprache ist eine Typ-0-Grammatik und wird von einer Turingmaschine erkannt.

Die unterschiedlichen Phasen eines Compilers

Im allgemeinen ist der Übersetzungsvorgang eines Compilers in verschiedene Phasen unterteilt. Die nachstehende Grafik zeigt die Phasen eines Compilers.



Analysephase („Frontend“)

1. Lexikalische Analyse (lineare Analyse, Scanning): Der Zeichenstrom des Quellprogramms wird dabei von links nach rechts gelesen und ins Symbole (Tokens) zerlegt.
2. Syntaktische Analyse (hierarchische Analyse, Parsing): Die lexikalische Analyse zerlegt zwar den Eingabetext in einzelne Tokens, führt jedoch keine Prüfungen durch, ob die Reihenfolge der Tokens sinnvoll ist. Auch können Vorrangsregeln wie „Punkt vor Strich“ bei der lexikalischen Analyse nicht berücksichtigt werden.
3. Semantische Analyse: Die semantische Analyse muss im Anschluss z.B. erkennen, dass ein Operand eventuell zuerst von einer ganzen Zahl in eine Kommazahl (Gleitpunktzahl) umzuwandeln ist, bevor er mit den anderen Operanden eines Ausdrucks verknüpft werden kann.

Synthesephase („Backend“)

Nach der Analyse folgt die Synthese, in der aus dem parse tree das gewünschte Zielprogramm konstruiert wird.

1. Zwischencode-Generator: Der pars tree soll nun in ein Zielprogramm übersetzt werden. Der Zwischencode-Generator übersetzt dazu das Quellprogramm in eine Zwisdchensprache. Es gibt dabei verschiedene Möglichkeiten, darunter den sogenannten „Drei-Adress-Code“. Dieser ähnelt der Assembler-Sprache für eine Maschine. Dieser Code ist eine Folge von Befehlen, bei denen jeder Befehl höchstens drei Operanden und neben der Zuweisung „=“ höchstens einen Operator (*,+,…) hat, wie z.B.:

```
temp1 = intoreal(50)  /* wandelt Ganzahl 50 in 50.0 um */
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

temp1, temp2 und temp3 sind dabei temporäre Namen für Speicherplätze (Register). Ein Zwischencode hat den Vorteil, dass man unnötigen Aufwand vermeidet, wenn man Compiler zu anderen Sprachen bzw. einen Compiler für anderer Prozessoren anbieten möchte.

2. Code-Optimierer: Der erzeugte Zwischencode wird dann vom Code-Optimierer verbessert, indem z.B. redundante Um- und Zwischenspeicherungen erkannt und beseitigt werden.

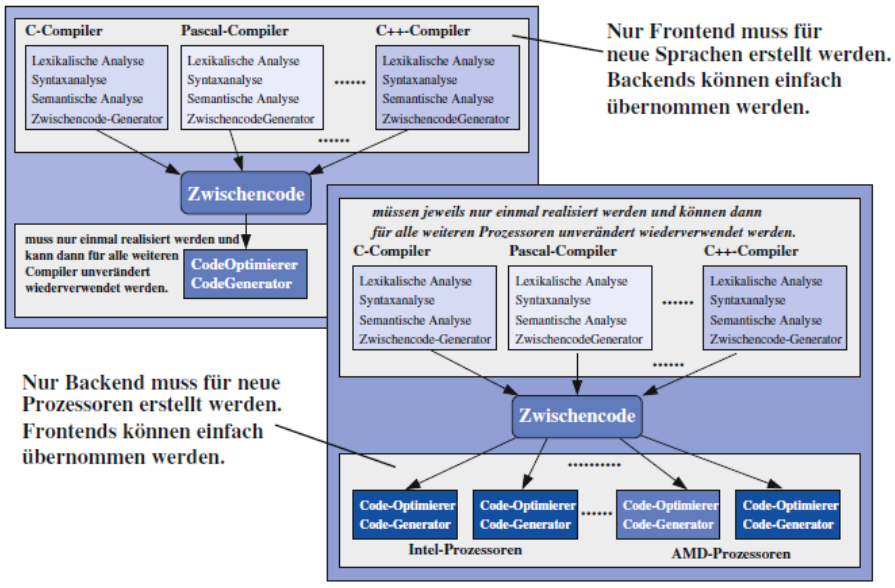
```
temp1 = id3 * 50.0
id1 = id2 + temp1
```

3. Code-Generator: Der Code-Generator übersetzt schließlich das in der Zwischensprache vorgelegte Programm in den Code der Zielmaschine oder in ein Assemblerprogramm.

```
MOVF id3, R2 ;     Lade Inhalt von id3 in das Register R2
MULF #50.0, R2 ;   Multipliziere Inhalt von Reg. R2. mit der Zahl 50.0
MOVF id2, R1 ;     Lade Inhalt von id2 in das Register R1
ADDF R2, R1  ;     Addiere auf Reg. R2 den Inhalt von Reg. R1 auf
MOVF R1, id1 ;     Speichere Inhalt von Reg. R1 in id1
```

Das angehängte F im Pseudo-Assemblercode bedeutet, dass es sich um eine float-Zahl handelt.

Vorteile bei Verwendung eines Zwischencodes:



Parallel zu diesen sechs Phasen hat der Compiler noch zwei Aufgaben:

- Symboltabellenverwaltung
  Dies ist die zentrale Datenstruktur, die für jeden Bezeichner eines Quellprogramms einen Platz vorsieht, in dem Name, Datentyp und sonstige Attribute festgehalten werden.
- Fehlerbehandlung
  Dieses Programmteil enthält üblicherweise die zu allen möglichen Fehlerarten gehörigen Nummern und Meldungen. Stößt der Compiler bei der Bearbeitung des ihm vorgelegten

Quellprogramms auf einen Fehler, meldet er dies. Bei den meisten Fehlern kann der Compiler mit dem kompilieren fortfahren, um weitere Fehler zu entdecken.

Zahlensysteme

Als Beginn der Datenverarbeitung kann die Erfindung des Zahlensystems angesehen werden. Zahlensystem wurden in der Vergangenheit sehr unterschiedlich konzipiert. Fast alle beruhen aber auf dem Abzählen der Finger. In der Informatik werden hauptsächlich das Dual-, Oktal-, Dezimal- und das Hexadezimalsystem verwendet.

Verschiedene Zahlensysteme haben unterschiedliche Basen und Bereiche, in denen Ziffern vorkommen.

Binär:	Basis 2, 0-1
Oktal:	Basis 8, 0-7
Dezimal:	Basis 10, 0-9
Hexadezimal:	Basis 16, 0-9 und A-F

Positionssysteme

Ein Positionssystem mit der Basis B ist ein Zahlensystem, in dem eine Zahl x nach Potenzen von B zerlegt wird. Das von uns verwendete Zehnersystem ist ein solches Positionssystem. Dies bedeutet, dass jeder Position in einer Zahl ein bestimmter Wert zugeordnet wird, der eine Potenz von 10 ist.

Bespiele:
dezimal

$$\begin{aligned} n = (2017)_{10} &= 2\cdot 10^3 + 0\cdot 10^2 + 1\cdot 10^1 + 7\cdot 10^0 \\ \text{in Kurzform} &: 2\cdot 10^3 + \quad + 1\cdot 10^1 + 7\cdot 10^0 \\ \text{oder} &: 2000 + \quad + 10 + 7 \end{aligned}$$

oktal

$$\begin{aligned} n = (315)_8 &= 3\cdot 8^2 + 1\cdot 8^1 + 5\cdot 8^0 \\ &= 3\cdot 64 + 1\cdot 8 + 5\cdot 1 \\ &= 192 + 8 + 5 = (205)_{10} \end{aligned}$$

Tabelle für die Zahlendarstellung in fünf verschiedenen Zahlensystemen

Dual	Oktal	Dezimal	Hexadezimal	Zwölfersystem
0	0	0	0	0
1	1	1	1	1
10	2	2	2	2
11	3	3	3	3
100	4	4	4	4
101	5	5	5	5
110	6	6	6	6
111	7	7	7	7
1000	10	8	8	8
1001	11	9	9	9
1010	12	10	a	a
1011	13	11	b	b
1100	14	12	c	10
1101	15	13	d	11
1110	16	14	e	12
1111	17	15	f	13
10000	20	16	10	14
10001	21	17	11	15

Dual-, Oktal- und Hexadezimalsystem

Dualsystem

Da das Zehnersystem , in dem Zehn verschiedene Ziffern 0, 1, 2,…, 9 existieren, technisch schwer zu realisieren ist, benutzt man in Rechnern intern das Dualsystem, bei dem nur zwei Ziffern, 0 (kein Strom/keine Spannung) und 1 (Strom/Spannung), verwendet werden.

Umrechnen ins Oktalsystem

Um eine im Dualsystem dargestellte Zahl ins Oktalsystem zu konvertieren, bildet man von rechts beginnend sogenannte Dualtriaden (Dreiergruppen).

```
110  111  001  110  010  | Dualzahl
 6    7    1    6    2  | Oktalzahl
```

Bei der Umwandlung vom Oktalsystem ins Dualsystem wird der umgekehrte Weg gegangen.

Umrechnen ins Hexadezimalsystem

Um ins Hexadezimalsystem zu gelangen bildet man Vierergruppen.

Umrechnen von Dezimal in andere Positionssysteme

Für die Umwandlung einer Dezimalzahl x in ein Zahlensystem mit der Basis n kann folgender Algorithmus verwendet werden.

1. $x : n = y \qquad Rest\ z$
2. Mache y zum neuen x und fahre wieder mit Schritt 1 fort, solange y>0, sonst fahre mit Schritt 3 fort.
3. Die ermittelten Reste z von unten nach oben nebeneinander geschrieben ergeben dann die entsprechende Dualzahl.

Konvertieren echt gebrochener Zahlen

Für die Umwandlung des „Nachkommateils“ einer Dezimalzahl kann folgender Algorithmus verwendet werden, wobei B die Basis darstellt:

1. $x * B = y \quad Überlauf\ z$ (z = ganzzahliger Anteil)
2. Der Nachkommateil von y ist nun das neu x. Fahre mit Schritt 1 solange fort, bis y=0, ansonsten mit Schritt 3
3. Schreibe die ermittelten Überläufe von oben nach unten nebeneinander.

Manche gebrochene Zahlen, die sich ganz genau im Dezimalsystem darstellen lassen, lassen sich nicht ganz genau als Dualzahl darstellen wie z.B. $0.1_{(10)} = 0.0001100110011..._{(2)}$

Um eine unecht gebrochene Zahl (z.B. 12,25) zu konvertieren muss man die Zahl in den ganzzahligen Teil und den echt gebrochenen Teil aufteilen, die dann getrennt voneinander konvertiert werden.

Rechenoperationen im Dualsystem

Addition

Für die duale Addition gilt allgemein:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1

1 + 1 = 0 Übertrag 1
1 + 1 + 1 = 1 Übertrag 1

Subtraktion mit Hilfe des 2er-Komplements

Bei der Subtraktion wird vom Subtrahend der Kehrwert gebildet, d.h. 0 = 1 und 1 = 0. Das Ergebnis wird mit 1 addiert. Dieses wird anschließend mit dem Minuend addiert. Das Ergebnis besitzt um einen Bit mehr als der Minuend und Subtrahend, welches einfach zu ignorieren ist.

Beispiel:
37 – 21 = 16

37 = 100101
21 = 010101 (Komplement: 101010)

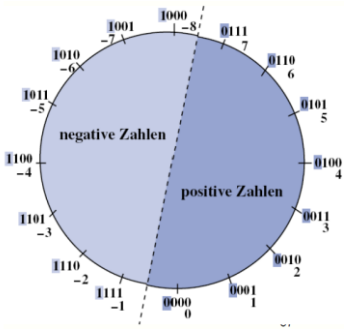
101010 + 1 = 101011

```
        100101
+       101011
--------------
      1|010000  →  1000(2)=16(10)
```

Negation

Wie im Dezimalsystem gibt es auch im Binärsystem negative Zahlen. Diese negative Zahlen werden jedoch nicht mit einem „-“ gekennzeichnet, sondern mit einer führenden „1“ (Vorzeichenbit).

0000 = 0	1000 = −8
0001 = 1	1001 = −7
0010 = 2	1010 = −6
0011 = 3	1011 = −5
0100 = 4	1100 = −4
0101 = 5	1101 = −3
0110 = 6	1110 = −2
0111 = 7	1111 = −1

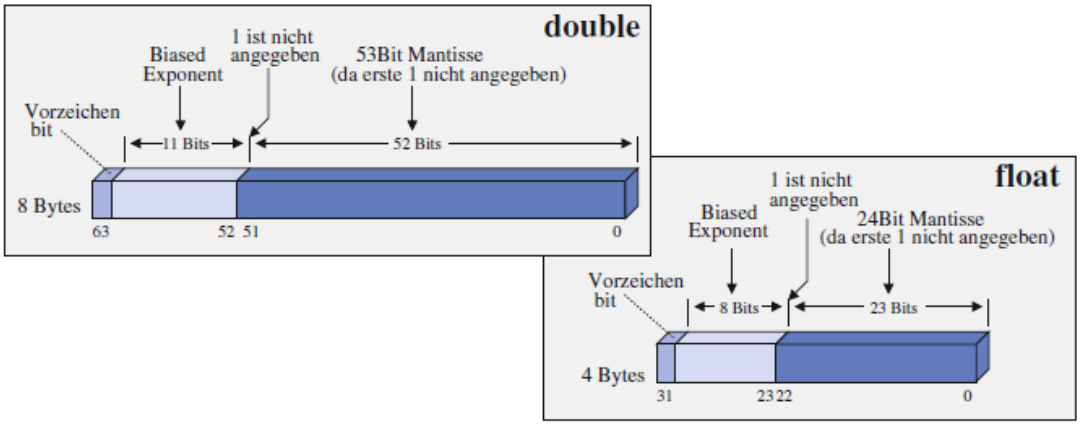


IEEE 754

Das Institute of Electrical and Electronics Engineering definierte mit der IEEE 754 die standartisierte Darstellung von binäre Gleitkommazahlen. Es gibt dazu die Formate *single* und *double*, die im Anschluss anhand eines Beispiels erklärt werden.

Im Prinzip besteht die IEEE 754 aus Vorzeichen, Mantisse, Basis und Exponent.

Im single floating point besteht die Mantisse aus 23 Bits, im double floating point aus 52.



Single (Float):

Zuerst wird die Zahl 147,963 in das binäre Format konvertiert.

```
147,963 = 10010011,11110110100001110010101100000010000011000100 1
```

1. Nun wird diese Zahl „normiert“ bzw. „normalisiert“. Das bedeutet, dass die Mantisse ermittelt wird.
   (Die Mantisse der Dezimalzahl 147,963 (bzw. 147963 * 10^-3) ist 147963.)

```
1,00100111111011010000111001010110000001000000110001001 x 10^7
```

Da die 1 immer davor steht, kann diese als Hidden-Bit angesehen werden.

2. Nun wird der Versatz (Bias) ermittelt. Um wie viel der Wertebereich verschoben wird, hängt von der Genauigkeit ab. Da der Zahlenbereich „single“ (32 Bit) anzugeben ist, benötigen wir den Versatz von 127. Wir müssen nun Bias + Exponent rechnen, um den neuen Exponent (Charakteristik) erhalten. Diese Charakteristik wird dann ins Binärformat konvertiert. Die Stellen des Bias‘ betragen beim „single“ 8 Bit.

   ```
   127 + 7 = 134
   ```

   ```
   134 : 2 = 67    0 Rest
    67 : 2 = 33    1 Rest
    33 : 2 = 16    1 Rest
    16 : 2 =  8    0 Rest
     8 : 2 =  4    0 Rest
     4 : 2 =  2    0 Rest
     2 : 2 =  1    0 Rest
     1 : 2 =  0    1 Rest
   ```

   ```
   Ergebnis: 10000110
   ```

3. Nächster Schritt ist die Vorzeichenbestimmung. Positiv = 0; Negativ = 1 => Die Zahl ist positiv, also Vorzeichen = 0

4. Zum Schluss wird die Gleitkommazahl aufgeschrieben

   ```
   Format: Vorzeichen  Charakteristik  Mantisse (23 Stellen)
   ```

   ```
   0 10000110 00100111111011010000111
   ```

Double

1. Zuerst wird die Zahl 147,963 in das binäre Format konvertiert. Da dies schon in Aufgabe 3.1.1. erledigt wurde, kann man diesen Wert ohne weiteres übernehmen.

```
147,963 = 10010011,11110110100001110010101100000010000011000100 1
```

2. Nun wird diese Zahl - wie in 3.1.2 - „normiert“ bzw. „normalisiert“.

```
1,00100111111011010000111001010110000001000001100010011 x 10^7
```

3. Nun wird der Versatz (Bias) ermittelt. Um wie viel der Wertebereich verschoben wird, hängt von der Genauigkeit ab. Da der Zahlenbereich „double“ (64 Bit) anzugeben ist, benötigen wir den Versatz von 1023. Wir müssen nun Bias + Exponent rechnen, um den neuen Exponent (Charakteristik) erhalten. Diese Charakteristik wird dann ins Binärformat konvertiert. Die Stellen des Bias‘ betragen beim „double“ 11 Bit.

```
1023 + 7 = 1030

1030 : 2 = 515   0 Rest
 515 : 2 = 257   1 Rest
 257 : 2 = 128   1 Rest
 128 : 2 =  64   0 Rest
  64 : 2 =  32   0 Rest
  32 : 2 =  16   0 Rest
  16 : 2 =   8   0 Rest
   8 : 2 =   4   0 Rest
   4 : 2 =   2   0 Rest
   2 : 2 =   1   0 Rest
   1 : 2 =   0   1 Rest

Ergebnis: 10000000110
```

4. Nächster Schritt ist die Vorzeichenbestimmung. Positiv = 0; Negativ = 1 => Die Zahl ist positiv, also Vorzeichen = 0
5. Zum Schluss wird die Gleitkommazahl aufgeschrieben

```
Format: Vorzeichen  Charakteristik  Mantisse (52 Stellen)

0 10000110 0010011111101101000011100101011000000100000110001001
```

Single:

Vorzeichen + Hochzahl + Mantisse = Länge
1+8+23=32 Bits

Double:

1+11+52=64 Bits

Boolesche Algebra

George Boole war ein englischer Mathematiker, der sich Mitte des 19. Jahrhunderts mit der formalen Sicht heutiger digitaler Strukturen beschäftigte. Dabei entwickelte er die nach ihm benannte boolesche Algebra. In dieser existieren nur zwei Werte – 0 (falsch bzw. false) und 1 (richtig bzw. true). Sie bildet die Grundlage der heutigen Rechner-Hardware.

OR-Operator

Das Ergebnis des OR-Operator ist 1, sobald mindestens eine Variable 1 ist. Er wird mit + oder v geschrieben. Dieser wird auch als logische Summe bezeichnet.

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

AND-Operator

Das Ergebnis einer AND-Operation ist 1, sobald alles Variablen den Wert 1 besitzen. Er wird mit *, ∧ oder leer geschrieben. Dieser wird auch als logisches Produkt bezeichnet.

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

NOT-Operator

Der NOT-Operator negiert die einzelnen Variablen. Er wird als ā oder ¬ geschrieben. Aus diesem lassen sich NAND und NOR bilden.

a	NOT a
0	1
1	0

XOR-Operator

Die Ergebnisse einer XOR-Operation sind dann 1, wenn eines des Variablen 1 ist. Sind beide Variablen 0 oder 1 ist das Ergebnis XOR-Operation 0. XOR bedeutet e**X**clusive **OR**.

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Rechenregeln in der booleschen Algebra

Name	Formel	Kurzschreibweise
Kommutativgesetze	a * b = b * a a + b = b + a	$ab = ba$
Assoziativgesetze	a * (b * c) = (a * b) * c a + (b + c) = (a + b) + c	$a(bc) = (ab)c$
Distributivgesetze	a * (b + c) = (a * b) + (a * c) a + (b * c) = (a + b) * (a + c)	$a(b+c) = ab + ac$ $a + bc = (a+b)(a+c)$
Identitätsgesetze	a * 1 = a a + 0 = a	$a1 = a$
Null-/Einsgesetze	a * 0 = 0 a + 1 = 1	$a0 = 0$
Komplementärgesetze	a * -a = 0 a + -a = 1	$a\overline{a} = 0$ $a + \overline{a} = 1$
Idempotenzgesetze	a * a = a a + a = a	$aa = a$
Verschmelzungsgesetze	a * (a + b) = a a + (a * b) = a	$a(a+b) = a$ $a + ab = a$
De Morgan'sche Gesetze	-(a * b) = -a + -b -(a + b) = -a * -b	$\overline{ab} = \overline{a} + \overline{b}$ $\overline{a+b} = \overline{a}\,\overline{b}$
Doppeltes Negationsgesetz	-(-a) = a	$\overline{\overline{a}} = a$

Wie in der Mathematik hat das Produkt gegenüber der Addition Vorrang.

Aufbau von Computersystemen

Zentraleinheit und Peripheriegeräte

Die Hardware besteht grundsätzlich aus Zentraleinheit und Peripherie.
Zur Zentraleinheit zählen vor allem Mikroprozessor, Arbeitsspeicher (RAM), die verschiedenen Bus- und Anschlusssysteme sowie das Ein-/Ausgabesystem. Zur Peripherie gehören die Komponenten, die zusätzlich an die Zentraleinheit angeschlossen werden.

- Ausgabegerät
  Grafikkarte, Bildschirme, Projektoren, Drucker, Lautstärker, usw.
- Eingabegeräte
  Tastatur, Maus, Scanner, Mikrofon, Touchscreen, usw.
- Ein-/Ausgabegerät
  Festplatten, SSD, Modem, Soundkarten (Ansteuerung von Mikrofon und Lautsprecher), Multifunktionsdrucker, USB-Stick usw.

EVA

Computer arbeiten nach dem EVA-Prinzip:

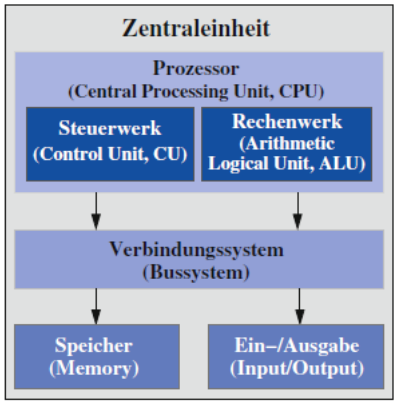
1. **E**ingabe: Über eine Eingabeeinheit z.B. Tastatur, Maus, USB-Stick usw. gelangen Daten in den Computer
2. **V**erarbeitung dieser Daten findet in der Zentraleinheit statt
3. **A**usgabe: über eine Ausgabeeinheit z.B. Monitor, Drucker, Festplatte usw.

Das EVA-Verfahren lässt sich durch die ganze Geschichte der Computer verfolgen.

Von-Neumann-Architektur

Der heutige Aufbau von Rechnern folgt weitgehend dieser klassischen von-Neumann-Architektur. Bei heutigen PCs befinden sich diese Grundeinheiten physikalisch auf der sogenannten Hauptplatine (Mainboard/Motherboard) und sind dort elektrisch verschaltet.

Die Grundkomponenten eines von-Neumann-Rechners:



- Prozessor (CPU): besteht aus Rechen- und Steuerwerk.
  Das Steuerwerk liest die Befehle und deren Operanden nacheinander ein und verarbeitet diese anhand seiner Befehlstabelle.
  Das Rechenwerk (ALU) führt die entsprechenden Algorithmen und logischen Operationen durch.
  Diese Rechen- und Steuerwerke sind auch heute noch die wichtigsten Komponenten von Mikroprozessoren.
- Arbeitsspeicher (RAM) enthält die Befehle von ablaufenden Programmen und den dazugehörigen Daten.
- Das Bussystem ist für den Transport von Daten zwischen den Einheiten
- Die Ein-/Ausgabeeinheit kommuniziert mit der Umwelt, um neue Daten/Programme entgegenzunehmen bzw. fertig verarbeitete Daten auszugeben.

Die Zentraleinheit besteht aus der CPU, dem Arbeitsspeicher (RAM=Random Access Memory; enthält die gerade ausgeführten Dateien und deren verwendeten Daten – wird beim herunterfahrendes PCs gelöscht), dem ROM-Speicher (ROM=Read Only Memory – früher: Betriebssystem, heute: BIOS (Hardware-Tests und Booten)), Busse und Schnittstellen (für Peripheriegeräte wie z.B. Grafikkarten, Festplatten usw.) und dem Chipsatz (fest auf dem Mainboard untergebrachte Schaltkreise).
Trotz Unterbringung auf dem Mainboard als sogenannte Onboard-Komponenten (z.B. Grafik-, Netzwerk- oder Soundkarten) werden diese nicht zur Zentraleinheit gezählt, sondern zur Peripherie.

Betriebssysteme und Rechnernetze

Betriebssysteme

> *"Unter Betriebssystem versteht man alle Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechnersystems bilden und insbesondere die Abwicklung von (Anwendungs-)Programmen steuern und überwachen."*

- Deutsche Industrie Norm (DIN)

Geschichte der Betriebssysteme

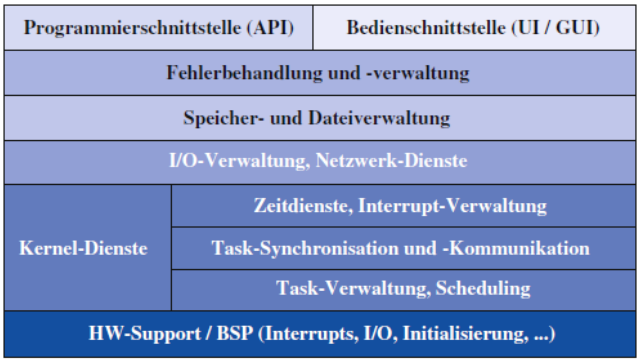
Die Geschichte der Betriebssysteme war stark von der Entwicklung der Computerarchitektur verwurzelt:

- 1. Generation (1940-1950): Röhren und Steckbretter
  Die ersten Generationen von Computern besaßen kein Betriebssystem. Es wurden die Programme in Maschinensprache geschrieben und die Grundfunktionen des Rechners wurden anhand von Umstecken von Kabeln bedient.
- 2. Generation (1950-1960): Transistoren und Stapelsysteme
  Mit Einführung der Lochkarte im Jahre 1950 wurde es möglich Programme auf eine Lochkarte zu schreiben und diese einzulesen. Der Programmierer schrieb das Programm und stanzte dieses auf eine Lochkarte. Der Lochkartenstapel wurde vom Operator zum Rechner gebracht, der die Lochkarten einlas und den entsprechenden Job (Programm) ablaufen lies. War dies erledigt, wurde das Programm vom Drucker in den Ausgaberaum geschickt, wo sie der Programmierer abholen konnte.
- 3./4. Generation (1960-1975): Integrierte Schaltkreise
  Das OD/360 von IBM war die erste größere Computerlinie, mit Millionen von Assemblerzeilen. Ein Problem ergab sich, dass der Prozessor – oft bis zu 90% der Gesamtzeit – auf die E/A-Operatoren warten musste. Dieses Problem wurde so gelöst, dass der Speicher in mehrere Teile aufgeteilt wurde. Wenn ein Job nun warten musste, konnte ein anderer Job ausgeführt werden und die CPU belasten.
  Eine andere neue Fähigkeit war, Jobs direkt von der Lochkarte auf die Festplatte zu schreiben, sobald diese in den Computerraum gebracht wurde. War ein Job beendet, griff der Rechner direkt auf die Festplatte zu, lud diesen in den leeren Speicherbereich und startete diesen. Diese „spooling“-Technik (**S**imultaneous **P**eripheral **O**peration **O**n **L**ine) wurde auch für die Ausgabe verwendet.
  Ebenfalls wurde erstmals das „Timesharing“ eingeführt. Dies diente dazu, dass nicht nur ein einzelner Programmierer wartet, bis sein Programm fertig durchgelaufen wurde, sondern es konnten mehrere Programme „gleichzeitig“ abgearbeitet werden. Dazu stellte die CPU Zeitscheiben für jeden Benutzer bereit und der Rechner teilte die Programme der Benutzer auf diese auf. Je nach Größe des Programms verbrauchte ein Programmierer für ein kurzes Programm nur eine Zeitscheibe, für ein längeres Programm mehrere.
- 5 Generation: (1975-heute):
  Eine große Neuheit dieser Generation sind die Computernetze, die sich über die ganze Welt hinaus erstrecken. Es gibt drei Standardbetriebssysteme, die den Höhenflug des PCs erst zu verdanken haben: MS-DOS (single-user BS von Microsoft), WindowsXX (Multitasking, Nachfolger von MS-DOS) und Unix (Multitasking und Multiuser, Gratis-Betriebssystem Linux entstand daraus)

Aufgaben eines Betriebssystems

- Benutzerschnittstelle durch das Graphical User Interface (GUI)
- Zugriff auf die Hardware des Rechners mittels Application Programming Interface (API)
- Verwaltung von Ressourcen:
  - Speicher
  - Dateien
  - Ein- und Ausgänge
  - Prozesse, Threads, Scheduling

Ein Betriebssystem ist üblicherweise in Schichten (bzw. Schalen) aufgebaut. Die Nutzung von Diensten erfolgt dabei von oben nach unten bzw. von außen nach innen. Das bedeutet, dass höhere Schichten Funktionen niedriger Schichten nutzen, aber nicht umgekehrt:



Prozess

Ein Schlüsselkonzept moderner Betriebssysteme ist das Prozess-Konzept. Programme werden in Form von Prozessen verwaltet und zum Ablauf gebracht.
Unterschied Programm und Prozess einfach erklärt: Man nehme an, man möchte einen Kuchen backen. Das dazu benötigte Rezept stellt das Programm dar. Der Bäcker ist der Prozessor, der den Prozess „Kuchen backen“ startet. Schließlich läutet es an der Tür und der Bäcker (Prozessor) startet den Prozess „Tür aufmachen“, da dieser wichtiger ist.

Thread

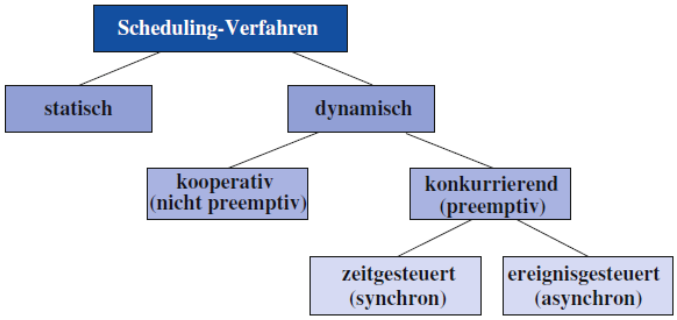
Ein Thread ist ein quasi parallel laufender Ablauffaden. Bei unserem vorigem Beispiel wäre es „Backrohr vorheizen“ und „Teig rollen“.

Scheduling

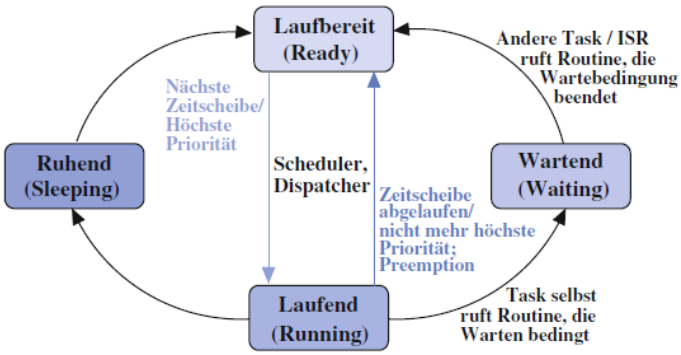
Bei einem Multitasking-BS können mehrere Prozesse abwechselnd bearbeitet werden. Die Umschaltung wird vom Scheduler geregelt. Dieser ist natürlich auch für das Zwischenspeichern der aktuellen Daten und Werte des unterbrochenen Prozesses zuständig, um beim Fortsetzen beim letzten Punkt fortzufahren.

Es gibt zwei Arten von Scheduling: statisch und dynamisch. Bei Embedded Systems – zum Beispiel – mit immer gleichen Aufgaben wird ein statisches Scheduling eingesetzt.
Dynamische Scheduler handeln nach folgenden unterschiedlichen Verfahren:

- Kooperatives Verfahren: Hier muss ein Prozess den Prozessor freiwillig abgeben, also den Scheduler aufrufen, um einen anderen Prozess/Ablauf zu ermöglichen.
- Konkurrierendes Verfahren: Hier entzieht der Scheduler dem Prozess die Rechte auf dem Prozess, was eine vorzeitige Unterbrechung (Preemtion) bewirkt. Hier muss der Scheduler für die Zwischenspeicherung des Prozesses sorgen. Es gibt hier zwei Arten, den Scheduler aufzurufen:
    - Zeitgesteuert (erhalten Rechenzeiz in abwechselnder Reihenfolge)
    - Ereignisgesteuert (erhalten Rechenzeit nach ihrer Wichtigkeit)



Für das grundsätzliche Scheduling werden folgende Task-Zustände benötigt, die zusammen mit ihren Übergängen dargestellt werden:



Anstelle von Prozess und Thread, wird häufig der Begriff **Task** verwendet.

Rechnernetze

Der Ursprung der Rechnernetzen entstand dabei, einen PC mit einem anderen verbinden zu können. Die Anfänge dafür bestimmte der **R**ecommended **S**tandard RS-232. Mit diesem konnte man mit folgenden Pinbelegungen zwei Rechner miteinander verbinden:

- Eine Hinleitung mit +12V (für logisch: 1) und -12V (für logisch: 0) = ausgehende Daten
- Eine Rückleitung mit denselben Voraussetzungen = eingehende Daten
- Gemeinsame Masse

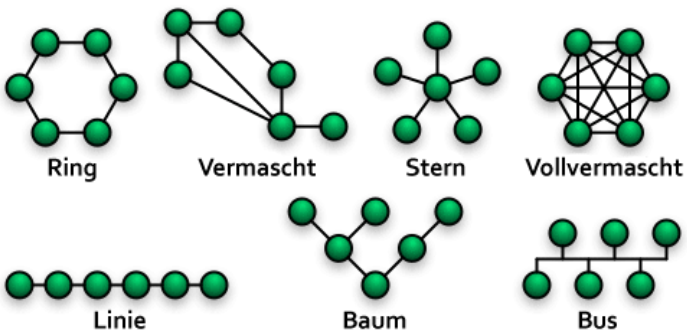
Die Übertragungsgeschwindigkeit ist vorgegeben.

Dieser Standard ist jedoch sehr veraltet, man kann mehrere Computer nur mit unübersichtlich viele Verbindungskabel verknüpfen – für 100 Computer würde man 4950 Kabel verbrauchen – und die Kabellänge soll nicht mehr als 100m betragen. Dennoch ist dieser noch weit verbreitet und sehr robust.

Da diese Erfindung nicht tragbar war, um viele Rechnernetze zu verbinden wurde ab Mitte der 1960-er Jahre am Internet entwickelt. Das Internet hatte ursprünglich militärische Zwecke und wurde erst in den 1970-er für die Öffentlichkeit „freigeschaltet“. Damit jedoch ein Rechner mit dem anderen über das Internet – unabhängig des Betriebssystems – kommunizieren kann, benötigt man einen gewissen Standard. Für das Internet ist dies das ISO/OSI-Modell.

Aufbau einfacher Topologien

Im Folgenden eine Grafik, um einen Überblick über die grundlegenden Topologien (spezifische Anordnung von Geräten und Leitungen, die ein Rechnernetz bilden) zu bekommen.



ISO/OSI-Modell

Das ISO/OSI-Modell wurde ab 1977 entwickelt und erst 1984 als internationaler Standard eingeführt.

Nr.	Schichten	Beispiele
7	Anwendungsschicht	Google Chrome, Thunderbird
6	Darstellungsschicht	http, smtp, ftp
5	Sitzungsschicht	„Check-Points“: RCP-Protokoll
4	Transportschicht	TCP, UDP
3	Vermittlungs-/Netzwerkschicht	IP
2	Sicherungssicht	Bridge, WAP, MAC
1	Bitübertragungsschicht	Netzwerkkabel

1. Schicht: Bitübertragungsschicht:

Diese Schicht dient, wie der Name schon vermuten lässt, zur Bitübertragung, also wie die Daten übertragen werden: zum Beispiel durch ein Netzwerkkabel aber auch Lichtwellenleiter, Antenne, Verstärker, Repeater und Hub.

2. Schicht: Sicherungsschicht:

Diese Schicht dient zur sicheren Übertragung von Daten. Dies hat zur Folge, dass die Daten in Blöcke (Frames) übertragen werden, um eine (halbwegs) fehlerfreie Übertragung zu gewährleisten. Der Switch und die Bridge sind ein Beispiel für die Hardware dieser Schicht.

3. Schicht: Vermittlungsschicht:

Diese Schicht ist für das Schalten der Verbindungen und bei paketorientierten Verbindungen für die Weitervermittlung zuständig. Unter diesem fällt auch das Internet Protocol (IP), welche als „Grundlage des Internets“ gilt.

4. Schicht: Transportschicht:

In dieser Schicht wird vor allem zwischen dem TCP (Transmission Control Protocol) und dem UDP (User Datagram Protocol) unterschieden:

- **TCP:**
  Bei diesem Protokoll ist vor allem die verlustfreie Übertragung der Datenpakete wichtig. Wird z.B. bei einer Website eine Seite von einer Tabelle nicht angezeigt, wird dies dem Überträger übermittelt und dieser schickt am Ende der folgenden Datenpakete das Fehlerhafte richtig nach, um somit den Datensatz richtig anzeigen zu können.
  Beim Verbindungsaufbau stellt der Client eine Anfrage an den Host (Server), dieser bestätigt diese und gibt ein OK an den Client zurück. Der Client bestätigt daraufhin die Verbindung. Die Anfragen werden Flags genannt.
- **UDP:**
  Bei diesem Protokoll ist die richtige Übertragung der Pakete trivial. Bei Online-Spiele oder beim Internet-Radio kommt es darauf an eine halbwegs flüssige Übertragung zu haben, so ist es im Beispiel vom Online-Gaming nicht von Vorteil, wenn ein Frame solange geladen wird, bis eine fehlerfreie Übertragung dieser Sequenz passiert. Man nimmt sozusagen Fehler in Kauf, um eine möglichst flüssige Datenübertragung zu gewährleisten.

5. Schicht: Sitzungsschicht:

Diese Schicht sorgt für die Prozesskommunikation zwischen zwei Systemen. Sie dient mit Hilfe von „Check-Points“ bei Ausfall der Übertragung als Wiederherstellungspunkt zur Synchronisation der Datenübertragung.

6. Schicht: Darstellungsschicht:

Diese Schicht ist für den Datenaustausch zwischen unabhängigen Systemen verantwortlich. Z.B. http(s), SMTP, usw. bei Windows zu OSX oder OSX zu Linux usw.

7. Schicht: Anwendungsschicht:

Auf dieser Schicht findet die Dateneingabe und -ausgabe statt, wie z.B. bei Webbrowsern (Google Chrome, Apple Safari) oder E-Mail Programme (z.B. Microsoft Outlook, Mozilla Thunderbird).

IP-Nummer

Jeder Rechner stellt im Internet eine Nummer zur Verfügung, unter die er eindeutig – für die Sitzung – identifiziert werden kann, so wie eine Adresse eines Hauses. Hierbei wird zwischen statischer und dynamischer IP-Adresse unterschieden: Die dynamische IP-Adresse wird jedes Mal vor einer Anmeldung im Internet vom Provider (Internetanbieter) automatisch vergeben und bei der Abmeldung wieder gelöscht. Dynamische IP-Adressen werden Hauptsächlich an Privatpersonen und kleinere Firmen von einem DHCP-Server (Dynamic Host Configuration Protocol) verliehen. An größeren Firmen und an Servern werden (nach Ansuchen beim Provider) statische IP-Adressen verliehen. Der Vorteil von statischen IP-Adressen ist, dass sich diese nach Abmeldung nicht ändert. Dadurch, dass jeden Tag immer mehr Rechner im Internet angemeldet sind werden diese jedoch nur mehr rar vergeben.
Auf Grund dass immer mehr Rechner im Internet registriert sind, hat dies zu Folge, dass die klassischen IP-Adressen (IPv4-Adressen) bald nicht mehr ausreichen. Deshalb wird schon seit einigen Jahren an IPv6 entwickelt. Da die Umstellung des alten Protokolls zum neuen sehr Zeitaufwendig ist, wird es noch ein paar Jahre dauern, bis IPv6 vollständig zum Einsatz kommt.

Unterschied IPv4 zu IPv6:

- **IPv4:**
  - 4 Zahlen (32 Bit), Dezimal
  - Mit Punkt voneinander getrennt
  - Zwischen 0.0.0.0 und 255.255.255.255
  - Bis zu 4 Mrd. Rechner gleichzeitig verbinden
- **IPv6:**
  - 8 Zahlen (128 Bit), Hexadezimal
  - Mit Doppelpunkt getrennt (sind 0-Gruppen vorhanden, werden diese mit ‚::‘ getrennt)
    - Z.B. 3ffe:353:0:0:0:0:0:1 = 3ffe:353::1

Duplex

Bei der Übertragung von Daten wird zwischen einigen Richtungsabhängigkeiten unterschieden:

- **Simplex:**
  Die Übertragung findet nur einseitig statt: z.B. Radio
- **Halbduplex:**
  Die Übertragung findet beidseitig statt, jedoch können die Endgeräte nur abwechselnd kommunizieren: z.B. Funkgerät
- **Vollduplex:**
  Die Übertragung findet beidseitig statt und kann auch gleichzeitig erfolgen: z.B. Telefon

Vom Programm zum Maschinenprogramm

Softwareprojekte werden in mehrere voneinander abgegrenzte Phasen unterteilt, diese werden Softwarelebenszyklus („Software Life Cycle“) genannt.

Phasen des SLC

Es gibt zwar einen eindeutigen Ablauf des SLC, jedoch keine eindeutige Form der Darstellung. Im Folgenden werden die grundsätzlichen Phasen vorgestellt

1. Problemanalyse
(auch Anforderungsanalyse oder Systemanalyse)
Diese Analyse wird mit dem Auftraggeber durchgeführt. Der Auftraggeber gibt die Vorstellungen über die Funktionen des zu entwickelnden Programms weiter. Eventuell werden hier auch Performance-Vorgaben (Geschwindigkeit, Antwortzeit, …) oder Entwicklungskosten mit einbezogen. Das Ergebnis ist ein „Lastenheft“ bzw. „Pflichtenheft“ (=Anforderungsbeschreibung).

2. Systementwurf
Hier werden die zu lösenden Aufgaben in Module aufgeteilt. Dies erhöht die Übersichtlichkeit und verbessert die Korrektheit und Zuverlässigkeit. Ein weiterer Vorteil ist, dass verschiedene Programmteams später parallel an genau festgelegten und gegeneinander abgrenzenden Teilaufgaben arbeiten können.
Das Ergebnis ist eine Systemspezifikation, die als Grundlage für die Implementierung gilt.

3. Programmentwurf
In dieser Phase werden einzelne Module verfeinert, indem die Datenstrukturen festgelegt und Algorithmen entwickelt werden.
Das Ergebnis besteht in mehreren Programmspezifikationen.

4. Implementierung und Test
In diesem Stadium werden die einzelnen Module programmiert und anhand ihrer jeweiligen Spezifikation getestet. Durch das Zusammensetzen der einzelnen Module erhält man das Programm.

5. Betrieb und Wartung
Diese Phase umfasst die Pflege der Software, in der vom Kunden entsprechende Erweiterungen und Änderungen eingebracht oder entdeckte Fehler behoben werden. Unter Umständen kann dies wieder zur Problemanalyse führen, wodurch ein Zyklus entsteht.

Programmierwerkzeuge

Der Computer, den wir als Maschine betrachten, die ein Problem löst, in dem sie Befehle (instructions) ausführt. Ein Satz von Befehlen nennt man ein Programm. Der Computer kann nur primitive Funktionen lösen (z.B. „Addiere zwei Zahlen! Speichere die Zahl, die an Stelle x steht, an die Stelle y.“), die der Programmierer in ein komplexes Programm zusammenfasst. Da der Computer höheren Programmiersprachen natürlich nicht versteht, muss der Code in, die für den Menschen schwer lesbare, Maschinensprache übersetzt werden. Dabei gibt es zwei verschiedene Möglichkeiten:

- **Kompilieren**
  Hierbei wird jeder Befehl des in der höheren Programmiersprache geschriebenen Programms in eine entsprechende Folge von Maschinenbefehle übersetzt. Diese Technik der Übersetzung nennt man *kompilieren*, für dies der *Compiler* benötigt wird. Der Compiler generiert eine separate Datei.

- **Interpretieren**
  Hierbei werden die Befehle erst bei der Ausführung in ein Maschinenprogramm einzeln nacheinander übersetzt und ausgeführt. Diese Technik nennt man *Interpretation*, das vom *Interpreter* ausgeführt wird. Der Interpreter generiert keine eigene Programmdatei.

Neuere Programmiersprachen, wie z.B. Java oder C#, verwenden mehr oder weniger eine Mischung dieser beiden Methoden. Diese kompilieren nicht direkt in den Maschinencodes eines bestimmten HW-Prozessors, sondern in eine sogenannte *virtuelle Maschinencode*. Erst beim Ablauf wird dieser Code von einem SW-Programm (*virtuelle Maschine*) interpretiert und in den realen Maschinencode umgesetzt. Vorteil: Plattformunabhängig (solange eine virtuelle Maschine existiert) und Überprüfung von Befehlen.

Compiler

Der Source Code wird für eine bestimmte Plattform kompiliert. Die Kompilierung umfasst im wesentlichen zwei Schritte:

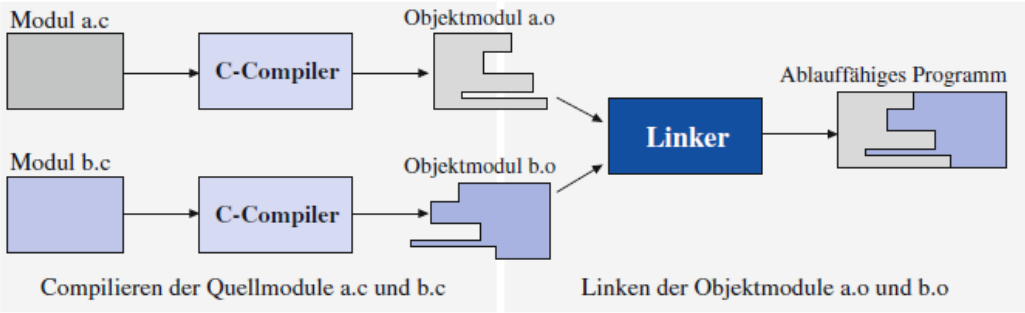
1. *Analyse*
   Das Quellprogramm wird in seine Bestandteile zerlegt und es wird eine Zwischendarstellung des Quellprogramms (*parse tree*) erzeugt.
2. *Synthese*
   Aus dem parse tree wird das gewünschte Zielprogramm konstruiert.

Besonderheiten:

- Das Programm kann aus mehreren Modulen bestehen, die sich in unterschiedlichen Dateien befinden und alle einzeln, getrennt voneinander, zu compilieren sind, bevor sie mit dem *Linker* „zusammengebunden“ werden.
- Wenn das Zielprogramm ein Assemblercode ist, muss eine weitere Übersetzung in Maschinencode erfolgen, was man als Assemblierung bezeichnet.

Linker

Die Grundidee in der Softwareentwicklung ist die getrennte Compilierung: Unterschiedliche Module können zu unterschiedlichen Zeitpunkten con verschiedenen Softwareentwicklern unabhängig voneinander compiliert werden. Später können die resultierenden *Objektdateien* mit dem Linker zu einem lauffähigen Programm zusammengebunden werden.



Man unterscheidet zwei folgende Arten von Linken:

- *Statisches Linken*
  Beim statischen Linken werden alle benötigten Funktionen fest zu einem Programm zusammengelinkt. Dies ist der sicherste Weg, wenn man vermeiden will, dass dem Benutzer eventuell benötigte Funktionen fehlen. Der Nachteil ist allerdings, dass die Datei sehr groß werden kann.

- *Dynamisches Linken*
  Beim dynamischen Linken werden nicht alles benötigten Funktionen sofort dazugelinkt, sondern man stellt dem Benutzer eine Bibliothek bereit, die dann erst zur Programmlaufzeit die fehlenden Abschnitte dynamisch dazulinkt. In diesem Fall spricht man von „dynamically linked libraries“ (DLLs) oder „shared libraries“. Dies hat den Vorteil, dass man die Bibliothek nachträglich leicht austauschen kann, die Programme kleiner werden und dass der Speicher benötigt wird, wenn mehrere Programme dieselbe Bibliothek verwenden. Der Nachteil dabei ist, dass man sicherstellen muss, dass auch der Benutzer die dynamische Bibliothek verwendet, und dazu noch die richtige Version davon.

Oft findet auch ein Mischen aus statischem und dynamischem Linken statt.

Der Linker kann an jeder Stelle des Arbeitsspeichers laden – dies ist die Aufgabe des Betriebssystems.

Lader und Locator

Das Ergebnis des Linkers- bzw. eines Assemblerverlaufs ist ein verschiebbarer Maschinencode, welcher an jeder Stelle des Arbeitsspeichers geladen werden kann. Wenn das Programm z.B. an die Adresse 3000 (Offset) geladen werden soll, muss es lociert bzw. relociert werden, indem zu allen Adressen im Code noch der Wert 3000 hinzuaddiert wird. Dies ist die Aufgabe des Laders.

Nach Starten des Rechners muss der Lader in den Hauptspeicher geladen werden. Dies geschieht hauptsächlich über ein eigenes Ladeprogramm, welches sich meist im BIOS befindet.
Bei Embedded Systems (wie z.B. Smartphones) werden diese meist bereits bei der Produktion in einem ROM-Speicher geladen und aus diesem direkt ausgeführt.
Der Lader eines Multitasking-Betriebssystems hat zudem noch die Aufgabe den Speicherplatz im Arbeitsspeicher für Programm- und Dienstsegmente zu reservieren.

Debugger

Eine einfache Möglichkeit, den ablaufenden Code zu betrachten, ist der Debugger. Dieses Programmiertool erlaubt es, das Programm schrittweise (=jeder Befehl einzeln) zu durchlaufen. Dabei können sowohl Befehle in Hochsprache, Maschinencode und Assemblersprache und die Adresse des Befehls im Arbeitsspeicher dargestellt werden.
Der Begriff Debugger heißt übersetzt „Entwanzer“. Er wird heute, wie damals, dazu eingesetzt, im Programm nach Fehlern zu suchen, indem er das Programm schrittweise durchläuft.

Komplexitätstheorie

P-Klasse

Ein Problem der P-Klasse, ist ein Problem, das nach **p**olynomialer Zeit lösbar ist. Ein Problem das berechenbar ist, kann mit einem Rechner gelöst werden, allerdings nur dann, wenn die Ressourcen ausreichen. Ein zwar berechenbares Problem ist z.B. nicht lösbar, wenn man mehr Speicherplätze benötigt, als das Universum Atome hat oder aber der schnellste Rechner der Welt eine Milliarde Jahre rechnen müsste.
Zur P-Klasse gehören z.B.:

- $O(\log(n))$ z.B. schnelles Potenzieren nach Legendre
- $O(n)$ z.B. sequentielle Suche
- $O(n \log(n))$ z.B. Sortieralgorithmen
- $O(n^2)$ z.B. Primzahlensieb nach Eratosthenes
- $O(n^3)$ z.B. klassische Matrizenmultiplikation

Ein Algorithmus der z.B. zur Komplexitätsklasse $O(2^n)$ gehört, wächst nicht polynomial, sondern exponentiell und gehört somit nicht in die Klasse P.

Für die Zukunft wird vorausgesagt, dass manche heute praktisch unlösbaren Probleme mit Hilfe eines Quantencomputers in erheblich kürzeren Zeiten lösbar sein können. Dazu werden aber Algorithmen benötigt, die bisher einen exponentiellen Aufwand benötigen, dann nur einen Polynomialen aufwenden. Diese Algorithmen werden aber nur für sehr spezielle Probleme erwartet.

NP-Problem

Als impraktikabel (praktisch undurchführbar) gelten die Algorithmen mit mindestens exponentiellem Aufwand. Obwohl sich die Klassen der praktikablen und impraktikablen Probleme unterscheiden lassen, ist es nicht immer einfach gewisse Probleme dementsprechend zuzuordnen. Es gibt daher eine Art Zwischenstellung: die NP-Klasse (*Nondeterministic Polynomial time solvable*).

Klasse NP ist die Menge aller Probleme, die ein nichtdeterministischer Algorithmus mit polynomialer Komplexität löst.

Nichtdeterministische Algorithmen dienen als Klassifizierungshilfe in der Informatik, um eine wichtige Klasse von algorithmisch lösbaren Problemen zu charakterisieren. Sie werden unter anderem deshalb verwendet, weil sie sich leicht als korrekt beweisen lassen. Hat man von einem nichtdeterministischen Algorithmus bewiesen, dass er ein Problem löst, dann ist dies auch mit jedem deterministischen Algorithmus möglich. Den entsprechenden deterministischen Algorithmus erhält man dann dadurch, dass man die willkürliche Auswahl im nichtdeterministischen jeweils durch feste Entscheidungen ersetzt, da $P \subseteq NP$ gilt. Solche schweren Probleme wie z. B. das SAT-Problem lassen sich nun in zwei Phasen lösen:

- Guess-Phase (Rate-Phase) in nichtdeterministischer polynomialer Zeit t_n
- Check-Phase (Prüf-Phase) in deterministischer polynomialer Zeit t_d

SAT-Problem

Das SAT-Problem (satisyability problem; Erfüllbarkeitsproblem) bezeichnet:

Beim SAT-Problem stellt sich zu einem gegebenen booleschen Ausdruck die Frage, ob eine Belegung der Variablen des Ausdrucks mit werten von true *und* false *existiert, sodass der Ausdruck den Wert* true *annimmt.*

- Für den booleschen Ausdruck $(\bar{a} + b) * (a + \bar{b} + \bar{c}) * \bar{c}$ liefert z.B. die Belegung a=false, b=true und c= false den Wert true. Dieser Ausdruck ist also erfüllbar.
- Der boolesche Ausdruck $a * (\bar{a} + b) * (\bar{b} + \bar{c}) * c$ dagegen ist nicht erfüllbar, da er für jede mögliche Variablenbelegung den Wert *false* liefert.

Ein Algorithmus, der alles Kombinationen von Belegungen überprüft, um festzustellen, ob der Ausdruck erfüllbar ist, benötigt $O(2^n)$ Schritte, wobei n die Anzahl der Variablen des Ausdrucks ist.
Bis heute ist kein Algorithmus bekannt, der dieses Problem in polynomialer Zeit löst.
Nehmen wir an, dass ein Rechner in einer Sekunde 1 Million Schritte durchführen kann, benötigt er abhängig von n folgende Zeiten:

$n = 20$: ~ 1 Sekunde	$n = 30$: ~ 18 Minuten	$n = 40$: ~ 12 Tage
$n = 50$: ~ 35 Jahre	$n = 60$: ~ 36 558 Jahre	$n = 100$: ~ 40 196 936 841 331 475 Jahre

Das SAT-Problem ist genauer gesagt ein NP-complete (NP-vollständig) -Problem.

NP-Vollständigkeit

Das SAT-Problem ist eines der schwierigsten Probleme in der Klasse NP. Da es eine Vielzahl von ähnlich schwierigen Problemen gibt, wurde eine eigene Klasse NP-vollständig eingeführt.

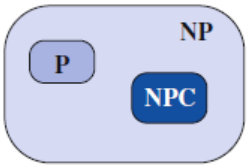
> *Ein Problem p heißt **NP-vollständig**, wenn das SAT-Problem auf p rückführbar ist. Die Klasse der NP-vollständigen Probleme wird mit **NPC** (C für complete) bezeichnet.*

Es gilt nun Folgendes:

1. Jedes NPC-Problem ist mindestens so schwierig wie das SAT-Problem.
2. Der Satz von Cook ("Jedes Problem in NP lässt sich polynomial auf das SAT-Problem zurückführen.") besagt wiederum, dass jedes NP-Problem höchstens so schwierig ist wie das SAT-Problem
3. Schließlich gilt: Jedes NPC-Problem ist gleich schwierig.

P=NP?

Nachdem Cook die Rückführbarkeit jedes NP-Problems auf das SAT-Problem und dieses damit als NP-vollständig nachgewiesen hatte, waren bereits im Jahr darauf über 20 NPC-Probleme bekannt. Heute kennt man mehrere Tausend von NPC-Problemen. Könnte man nur für ein einziges Problem der Klasse NPC nachweisen, dass es in der Klasse P (polynomial lösbar) liegt, dann wären die Problemklassen P und NP gleich, weswegen die folgende Beziehung

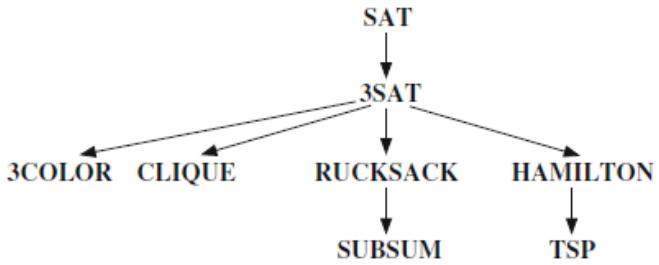


$$P = NP?$$

bis heute eines der größten ungelösten Rätsel der Informatik ist. Es spricht zwar vieles für $P \neq NP$, aber das konnte bis jetzt noch nicht bewiesen werden. Da die deterministischen Algorithmen ein Spezialfall der nichtdeterministischen sind, gilt sicher $P \subseteq NP$.

Weitere bekannte NP-Probleme

Dreifarbenproblem, 3SAT-Problem, Cliquenproblem, Rucksackproblem, Teilsummenproblem, Hamilton-Problem, Traveling Salesman Problem



Approximationsalgorithmen

Wird ein Problem in der theoretischen Informatik als „NP-vollständig“ klassifiziert, was „hoffnungslos schwierig“ bedeutet, muss man deswegen noch nicht aufgeben, denn es gibt meist auch so genannte Approximationsalgorithmen, die das jeweilige Problem zwar nicht optimal lösen, aber zumindest in einer polynomialen Zeit eine akzeptable Lösung finden.

Approximationsalgorithmus zum TSP

Nehmen wir als Beispiel das TSP, was auch in der Praxis eine wichtige Rolle spielt, wie z.B. bei Routenplan-Programmen. Um hier eine optimale Lösung zu finden, müsste man bei n Städten

$$n! = 1 * 2 * 3 * \ldots * n$$

Permutationen (Vertauschungen) der Zahlen 1, …, n durchprobieren, das es genauso viele Rundreisemöglichkeiten gibt. Zur Berechnung kann man hier die Stirling'sche Formel verwenden:

$$n! \approx \left(\frac{n}{e}\right)^n * \sqrt{2\pi n}$$

Bei n = 250 Städten würde dies folgende Anzahl an Permutationen bedeuten:

$$250! \approx \left(\frac{250}{e}\right)^n * \sqrt{500\pi} \approx 3{,}232 * 10^{492}$$

Wenn man z.B. annimmt, dass ein Rechner 1 Billion Permutationen in einer Sekunde erzeugen könnte, würde das entsprechende Programm immer noch über 10^{475} Jahre zur Lösung benötigen.

Ein möglicher Approximationsalgorithmus zum TSP ist nun z.B.:

Wähle immer als Nächstes die nächstgelegene Stadt, die noch nicht besucht wurde!

Diese Strategie wird auch als Greedy-Algorithmus bezeichnet.

Fehlertolerante Codes

Beim Übertragen, Speichern und Lesen von Daten können Fehler auftreten. Deshalb ist man bestrebt fehlertolerante Codes zu finden. Diese Codes ermöglichen dem Empfänger zu erkennen, ob bei der Übertragung ein Fehler aufgetreten ist und falls ja, diesen eventuell sogar zu korrigieren. Verfügt ein Code über die Eigenschaft einen Fehler zu erkennen, handelt es sich um einen fehlererkennenden Code. Ermöglicht er zusätzlich noch dem Empfänger, erkannte Fehler zu korrigieren, spricht man von einem fehlerkorrigierenden Code.

„k aus n"-Code

Manche Codes verwenden nur eine Teilmenge und nicht bei einer gegebenen Bitanzahl den ganzen möglichen Binärcode. Solche Codes können dazu verwendet werden, in vorliegenden Bitmustern gewisse Fehler zu erkennen und gegebenenfalls zu korrigieren. Durch Angaben wie die folgenden wird festgelegt, aus wie vielen n Bits ein Codewort jeweils besteht und wie viele k Bits in einem Codewort immer gesetzt sind:

$$\binom{10}{1}\,Code \qquad \text{oder:} \qquad \binom{5}{2}\,Code$$

Länge der 01–Tupel (1..30): **5** *[2 aus 5 Code]*
Wie viele Bits gesetzt (1..5): **2**
0 0 0 1 1 0 0 1 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 0
1 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 0 1 1 0 0 0

Länge der 01–Tupel (1..30):
4 *[1 aus 4 Code]*
Wie viele Bits gesetzt (1..4): **1**
0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0

Länge der 01–Tupel (1..30):
3 *[2 aus 3 Code]*
Wie viele Bits gesetzt (1..3): **2**
0 1 1 1 0 1 1 1 0

Hammingabstand

Die Anzahl der gesetzten Bits in einem Wort nennt man das *Hamminggewicht* des Wortes.
Die Anzahl von Binärstellen, an denen sich zwei Codewörter unterscheiden, heißt *Hammingabstand* d.
Aus dem Vergleich aller Codewörter untereinander kann dann der Hammingabstand des Codes bestimmt werden. Der Hammingabstand ist ein Maß für die Störsicherheit eines Codes. Hat ein Code den Hammingabstand d, so können

- Alle Fehler erkannt werden, die weniger als d Bits betreffen, und
- Alle Fehler korrigiert werden, die weniger als $\frac{d}{2}$ Bits betreffen.

Man spricht dann abhängig von der Anzahl n von falsch übertragenen Bits, die beim Code automatisch erkannt bzw. korrigiert werden können, von einem n-erkennenden bzw. n-korrigierenden Code.

Im Folgenden wird der Hammingabstand eines „2 aus 5“-Code bestimmt. Dazu werden folgende Zuordnungen der Zeichen zum Code genommen:

Zeichen	Code	Zeichen	Code	Zeichen	Code	Zeichen	Code	Zeichen	Code
a	00011	b	00101	c	00110	d	01001	e	01010
f	01100	g	10001	h	10010	i	10100	j	11000

Beispiele zur Bestimmung des Abstands zwischen Codewörtern:

	c **00110** g **10001**	a 00011 e 01010	d 01001 j 11000	e 01010 i 10100
Abstand	4	2	2	4

Tabelle zur Bestimmung des Hammingabstands:

	a	b	c	d	e	f	g	h	i	j
a	0									
b	2	0								
c	2	2	0							
d	2	2	4	0						
e	2	4	2	2	0					
f	4	2	2	2	2	0				
g	2	2	4	2	4	4	0			
h	2	4	2	4	2	4	2	0		
i	4	2	2	4	4	2	2	2	0	
j	4	4	4	2	2	2	2	2	2	0

Der Hammingabstand eines Codes ist der kleinste auftretende Abstand verschiedener gleichlanger Codewörter. In dieser Tabelle ist der kleinste Abstand zwischen den Codewörtern *2*. Da der Hammingabstand für den zuvor vorgestellten „2 aus 5“-Code *2* ist, kann ein Fehler, der eine Bitstelle betrifft, erkannt werden. Es handelt sich dabei also um einen 1-erkennenden-Code.

Der Hammingabstand des ASCII-Codes ist 1, d.h. man kann keine Übertragungsfehler erkennen.

1D-Parity-Prüfung

Eine häufig verwendetes Verfahren, um aus einem nicht fehlertoleranten Code einen 1-fehlererkennenden Code zu erhalten, ist die Verwendung eines sogenannten Parity-Bits.

Wenn die Anzahl der 1en im Code gerade, wird eine 0 angehängt, ist diese ungerade, wird eine 1 angehängt.

Als Beispiel ein 7-Bit ASCII Code. Da vorne eine 0 steht, wird diese nicht verwendet. Dafür wird hinten ein Parity-Bit angefügt:

A 1000001**0**	G 1000111**0**	M 1001101**0**	S 1010011**0**	Y 1011001**0**
B 1000010**0**	H 1001000**0**	N 1001110**0**	T 1010100**1**	Z 1011010**0**
C 1000011**1**	I 1001001**1**	O 1001111**1**	U 1010101**0**	
D 1000100**0**	J 1001010**1**	P 1010000**0**	V 1010110**0**	
E 1000101**1**	K 1001011**0**	Q 1010001**1**	W 1010111**1**	
F 1000110**1**	L 1001100**1**	R 1010010**1**	X 1011000**1**	

2D-Parity-Prüfung

Diese kommt bei Übertragung von Blöcken zum Einsatz. Sie verwendet zunächst für jedes einzelne Zeichen ein eigenes Parity-Bit. Zusätzlich wird aber, nachdem alle Zeichen übertragen wurden, noch ein weiteres Codewort übertragen, das die Parity-Bits zu allen Spalten des übertragenen Blocks enthält.

1	1	0	0	0	0	1	**1**	
1	1	0	0	0	1	0	**1**	
1	1	0	0	0	1	1	**0**	
1	1	0	0	1	0	0	**1**	
1	1	0	0	1	0	1	**0**	
1	1	0	0	1	1	0	**0**	
1	1	0	0	1	1	1	**1**	
1	1	0	1	0	0	0	**1**	
0	0	0	1	0	0	0	1	Spalten-Parity-Bits

Kippt beim Übertragen ein Bit, kann dieses bei der 2D Parity-Prüfung korrigiert werden.
Bei Vorliegen eines Einzelfehlers ist genau ein Zeilen- und ein Spalten-Parity-Bit falsch gesetzt. Einen falschen Wert hat dann in diesem Fall das Bit, das sich genau in der Zeile und in der Spalte mit den falschen Parity-Bits befindet. Daraus lässt sich schließen, dass ein Code mit zweidimensionaler Parity-Prüfung mindestens den Hammingabstand 3 hat.

1	1	0	0	0	0	1	1	
1	1	0	0	0	1	0	1	
1	1	0	1	0	1	1	0	⟸
1	1	0	0	1	0	0	1	
1	1	0	0	1	0	1	0	
1	1	0	0	1	1	0	0	
1	1	0	0	1	1	1	1	
1	1	0	1	0	0	0	1	
0	0	0	1	0	0	0	1	Spalten-Parity-Bits
			⇑					

Treten bei der Übertragung zwei oder drei Fehler auf, können diese zwar erkannt, aber nicht korrigiert werden.

Ein Code mit zweidimensionaler Parity-Prüfung ist 1-fehlerkorrigierend und 3-fehlererkennend.

Hamming-Codes

Der einfachste Hamming-Code ist ein (7,4)-Code, der eine Länge von 7 Bits hat, wovon allerdings nur 4 Bits Nutzinformationen sind und die restlichen 3 Bits zur Fehlerkorrektur dienen. Dieser Hamming-Code ist ein 1-fehlerkorrigierender Code mit dem Hammingabstand von 3.
Im Allgemeinen gibt es Hamming-Codes der Länge $2^r - 1$, wobei $r \geq 2$ sein muss. Davon sind dann r Bits Korrekturbits und die restlichen $2^r - 1 - r$ Bits Informationsbits. Für einen Code mit 3 Korrekturbits ergibt also die Gesamtlänge des Codes 7 Bits, wobei nur 4 Informations-Bits enthalten sind.

7 D	6 D	5 D	4 P	3 D	2 P	1 P	Parity-Bit an Position … (Dualzahl-Stellen, …
D	-	D	-	D	-	P	2^0 … die an letzter Stelle eine 1 haben)
D	D	-	-	D	P	-	2^1 … die an vorletzter Stelle eine 1 haben)
D	D	D	P	-	-	-	2^2 … die an drittletzter Stelle eine 1 haben)

- Parity-Bit an der Stelle 1 überprüft die Stellen (1), 3, 5 ,7 (jeden 2.)
- Parity-Bit an der Stelle 2 überprüft die Stellen (2), 3, 6, 7 (die ersten 2, lässt 2 frei und überprüft die nächsten 2)
- Parity-Bit an der Stelle 4 überprüft die Stellen (4), 5, 6, 7 (lässt die ersten frei und überprüft die nächsten 4)

Wenn die Anzahl der 1en gerade ist, wird wie bei der 1D- und 2D-Parity-Prüfung eine 0 geschrieben, ansonsten eine 1.
In folgender Tabelle wird die Information 1101 übertragen:

7 1	6 1	5 0	4 0	3 1	2 1	1 0
1	-	0	-	1	-	0
1	1	-	-	1	1	-
1	1	0	0	-	-	-

Anschließend wird von diesem Code ein Bit verändert, welches nicht das Hamming-Bit ist.
Es gibt nun zwei Möglichkeiten die Stelle des Fehlers festzustellen:

1. Man schreibt die Information, wie in der obenstehenden Tabelle und vergleicht, wo der Fehler auftreten kann:

7	6	5	4	3	2	1		
1	1	1	0	1	1	0	Paritätsbit	wird gewertet als
1	-	1	-	1	-	0	falsch	1
1	1	-	-	1	1	-	richtig	0
1	1	1	0	-	-	-	falsch	1

   Schreibt man nun diese Bitkombination aus der letzten Spalte dieser Tabelle „von unten nach oben“ hin, erhält man die Dualzahl 101, was der Dezimalzahl 5 entspricht, und genau im 5. Bit ist der Fehler aufgetreten.

2. Man schaut, welche Hamming-Bits „falsch“ sind, und addiert die Stellen der Hamming-Bits miteinander und erhält somit die Stelle des Fehlers:

7	6	5	4	3	2	1
1	1	1	0	1	1	0
1	-	1	-	1	-	0

1. Bit ist ungerade und sollte 1 sein – also ist dieses falsch.

7	6	5	4	3	2	1
1	1	1	0	1	1	0
1	1	-	-	1	1	-

2. Bit ist ungerade – also ist dies richtig.

7	6	5	4	3	2	1
1	1	1	0	1	1	0
1	1	1	0	-	-	-

4. Bit ist ungerade und sollte 1 sein – also ist dieses falsch.

Addiert man nun die zwei falschen Bits, erhält man die Stelle des Fehlers. $1 + 4 = 5$
Der Fehler ist, wie in Punkt 1 an der 5. Stelle.

Die folgende Tabelle zeigt noch den grundsätzlichen Aufbau eines (15,11)-Codes, wobei D für ein echtes Datenbit und P für ein Parity-Bit steht:

15	14	13	12	11	10	9	2^3	7	6	5	2^2	3	2^1	2^0
D	D	D	D	D	D	D	P	D	D	D	P	D	P	P
D	-	D	-	D	-	D	-	D	-	D	-	D	-	P
D	D	-	-	D	D	-	-	D	D	-	-	D	P	-
D	D	D	D	-	-	-	-	D	D	D	P	-	-	-
D	D	D	D	D	D	D	P	-	-	-	-	-	-	-

CRC-Kodierung

Der Cyclic Redundancy Check (CRC) ist der wohl am häufigsten verwendete fehlererkennende Code. Die Regeln einer CRC-Prüfung besagen, dass ein 16- oder 32-Bit-Prüfwer durch Division des Werts durch einen Divisor gebildet werden soll und diese Bildung über „Modulo-2“-Division geschieht. Auf diese Weise erhält man einen Quotienten und einen Rest:

$$Dividend\ modulo\ Divisor = Quotient + Rest$$

Bei der Division von Dualzahlen ist zur Ermittlung einer Stelle jeweils nur eine Subtraktion notwendig, da man sofort feststellen kann, ob der Divisor größer oder kleiner als der Teildividend ist.

Der Divisor wird auch Generatorpolynom genannt. Dieser dient als Schlüssel für Sender und Empfänger und wird vom Sender ausgegeben. Der Sender multipliziert seine Informationen mit dem Generatorpolynom und der Empfänger dividiert diese. Ist bei der Verbindung kein Fehler aufgetreten, so ist der Rest – logischerweise – null.
Für die CRC-Berechnung wird keine normale Arithmetik verwendet, wie im Kapitel Zahlensysteme erklärt ist, sondern die sogenannte Modulo-2-Arithmetik. Diese verwendet sowohl bei Addition als auch die Subtraktion als Differenz ein XOR-Gatter – man muss demnach nicht auf den Übertrag achten.

Erklärung anhand eines Beispiels:

Das Generatorpolynom lautet 110101 ($x^5+x^4+x^2+1$), ist also 5. Grades. Dem Rahmen (dies sind die Nutzdaten) werden noch n Nullen angehängt, wobei n dem Grad des Generatorpolynom entspricht.

Generatorpolynom:	110101	(wird vorher festgelegt; Schlüssel)
Rahmen:	11011	(Nutzdaten)
Rahmen mit Anhang:	1101100000	(das Generatorpolynom hat r Stellen, also werden $r-1=n$ Nullen ergänzt; n=Grad des Polynoms=5)

Nun wird der Rahmen mit Anhang von links her mit Modulo-2-Arithmetik mit dem Generatorpolynom dividiert. Modulo-2-Arithmetik bedeutet, dass ausschließlich mit XOR-Logik gerechnet wird.

```
1101100000
110101
------
0000110000
    110101
   -------
     00101  Rest
```

An den Rahmen ohne Anhang wird danach der Rest angehängt – dieser muss ebenfalls aus n Nullen bestehen.

Der übertragene Rahmen (mit Rest) ist also: 1101100101
Dieser Rahmen kann anschließend über ein Netzwerk übertragen werden.

Mittels Division des Generatorpolynoms kann der Empfänger der Information nun auf Fehler überprüfen.

Ist der Rest=0, so ist die Nachricht richtig übermittelt worden.

```
1101100101
110101
------
    110101
    110101
    ------
    000000 Rest
```

Der Empfänger muss also nun n Nullen (hier: 5, weil n=Grad=5) abziehen und erhält so die übertragene Zahl.

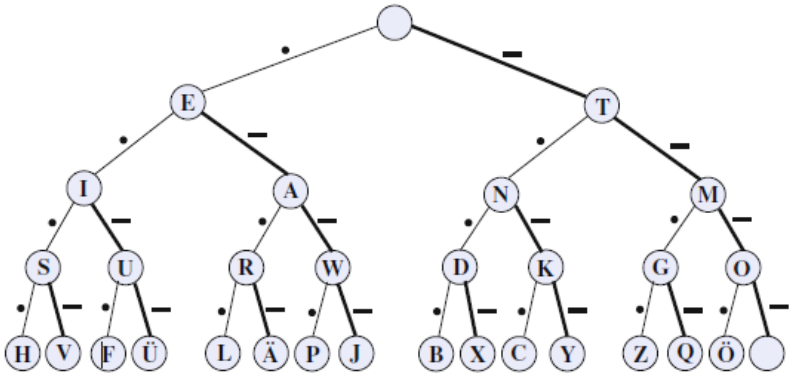
Datenkompression

Es wurde eine Vielzahl von Verfahren zur Datenkomprimierung entwickelt, die sich prinzipiell in zwei Gruppen unterteilen lassen.

- Verlustbehaftete Kompression (lossy compression): Hier geht ein Teil der Information verloren, was bedeutet, dass sich die dekodierte Daten von den Originaldaten unterscheiden. Dieser Informationsverlust kann z.B. bei Standbilder, Audio- oder Videodateien in Kauf genommen werden, da dies wahrnehmungspsychologisch trivial ist.
- Verlustlose Kompression (lossless compression): Bei diesen Verfahren unterscheiden sich die dekodierten Daten nicht von den Originaldaten. Dies wird verwendet, wenn der Verlust einzelner Bits die Qualität des Original wahrnehmbar beeinflusst z.B. bei der Kompression von Texten und Tabellen.

Morse Code

Je nach Sprache treten verschiedene Buchstaben unterschiedlich häufig auf. Je öfter der Buchstabe verwendet wird, desto kürzer sollte er also kodiert sein. Dieser Idee folgte Samuel F. B. Morse bereits im ersten Drittel des 19. Jahrhunderts. Morse entwickelte das nach ihm benannte Alphabet mit Hilfenahme von kurzen und langen Signalen, die graphisch mit „.“ und „-“ dargestellt werden.



Das Wort „MAUS“ sieht mit österreichischem Morse-Alphabet kodiert also wie folgt aus:

- -	. -	. . -	. . .
M	A	U	S

Fano-Bedingung

Codes mit verschieden langen Codewörtern müssen für die Kodierung eine, nach seinem Begründer, R. M. Fano, benannte, Bedingung erfüllen:

Ein Code erfüllt die Fano-Bedingung, wenn kein Wort aus dem Code der Anfang eines anderen Wortes ist.

Codes, die diese Bedingung erfüllen, nennt man auch *präfixfreie Codes*.
Der Morse-Code erfüllt diese Bedingung nicht. Das Wort „seen“ und „eier“ ist jeweils: „. . . . . - .“.

Lauflängenkodierung („run-length encoding“)

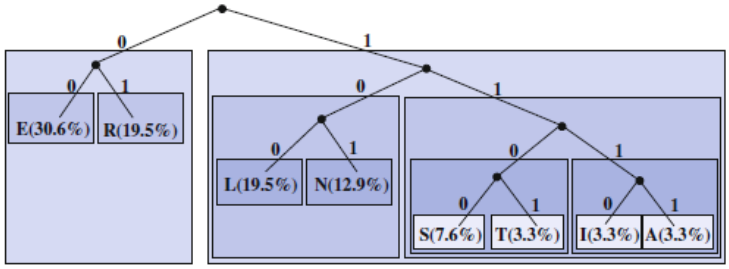
Eine einfache Methode zur verlustlosen Datenkomprimierung ist die Lauflängenkodierung, die bei Wiederholen eines Zeichens im Text durch einen Zähler vor dem entsprechenden Zeichen ersetzt wird. Die Zeichenfolge AAAAABBBBBBCCCDDDDABBCCCDDDD wird als 5A6B3C4DA2B3C4D übertragen.

Diese Vorgehensweise kann z.B. bei der Faxübertragung oder der Kompression eines Schwarz-Weiß-Bildes verwendet werden.

Shannon-Fano-Kodierung

Bei der Shannon-Faro-Kodierung werden die Buchstaben ihrer Auftrittswahrscheinlichkeit sortiert. Dazu werden die Zeichen zuerst in zwei Gruppen geteilt, sodass die Summer der Auftrittswahrscheinlichkeit beider Gruppen annähernd gleich ist. Nun erhalten alle Zeichen der ersten Gruppe ein Codewort beginnend mit 0 und alle Zeichen der zweiten Gruppe ein Codewort, das mit 1 beginnt. Die rechte Gruppe wird weiter in zwei Gruppen geteilt. Die erste Gruppe bekommt ein 0 hinzu, die zweite eine 1. Dieser Schritt wird rekursiv angewendet, wobei immer die zweite Gruppe unterteilt wird.

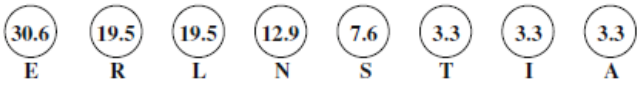
Zeichen	Häufigkeit	Codewort
E	30.6%	00
R	19.5%	01
L	19.5%	100
N	12.9%	101
S	7.6%	1100
T	3.3%	1101
I	3.3%	1110
A	3.3%	1111



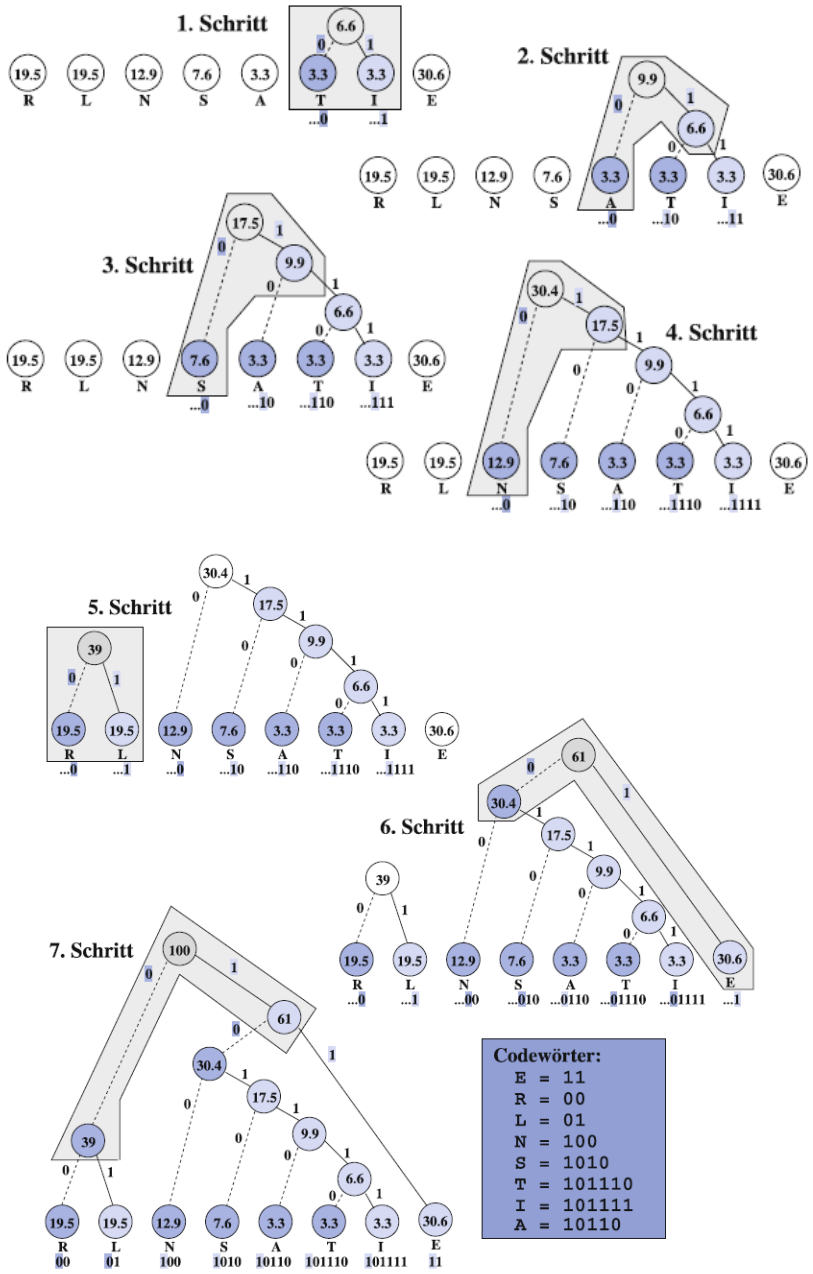
Huffman-Kodierung

Diese Kodierung ist der Shannon-Fano-Kodierung sehr ähnlich, nur dass sie die Codewörter nicht von vorne nach hinten, sondern umgekehrt von hinten nach vorne ermittelt. Das Verfahren baut dazu den Codebaum iterativ (wiederholend) von unten nach oben (*bottom up*) wie folgt auf:

1. Es wird zuerst für jedes Zeichen des Alphabets ein sogenanntes Blatt für den Codebaum angelegt, wobei dieses Blatt mit der Auftrittswahrscheinlichkeit dieses Zeichens beschriftet wird.



2. In jedem weiteren Schritt werden die beiden Konten mit den zwei kleinesten Wahrscheinlichkeiten zu einem neuen Knoten zusammengefasst. Die Beschriftung des neuen Knoten ist die Summe der beiden zusammengefassten Knoten. Die Linien werden jeweils mit (rechts) 1 bzw. (links) 0 beschriftet. Der entstandene „Kindsknoten“ wird nun mit dem nächsten Knoten der „kleinen“ Wahrscheinlichkeit verbunden. Dieser Schritt wird Wiederholt, bis man einen Knoten mit der Beschriftung 100% erhält.



Informationsgehalt und Entropie

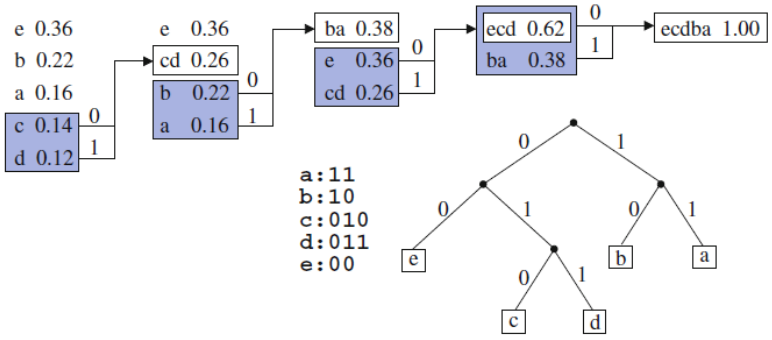
Der Informationsgehalt eines einzelnen Zeichens ist definiert durch folgende Formel:

$I(z_i) = \log_2(\frac{1}{p_i})$ wobei p_i die Auftrittswahrscheinlichkeit des Zeichens z_i ist

Für einen Datenquelle als Ganzes definiert man dann die so genannte Entropie H als den gewichteten Durchschnitt der Informationsgehalte alles Symbole S:

$$H(S) = \sum_{i=1}^{n} p_i I(z_i) = \sum_{i=1}^{n} p_i \log_2(\frac{1}{p_i})$$

Diese Entropieformel gibt die theoretisch optimal mögliche Kodierung an, also die kleinstmögliche Codewortlänge.
In der folgenden Grafik, die eine Kodierung mit 5 Zeichen zeigt, wird gezeigt, dass mit der Huffman-Codierung fast die theoretisch mögliche kleinste Codewortlänge erreicht wird. Die anschließende Tabelle zeigt die Berechnungen des Informationsgehalts I, der Entropie H und der Kompression K.



Berechnung der Entropie H und der Kompression K zur Abbildung 21.6

z_i	p_i	$I(z_i)$	$p_i I(z_i)$	Code	Bitanzahl	Bitanzahl · p_i
a	0.16	2.644	0.423	11	2	0.32
b	0.22	2.184	0.481	10	2	0.44
c	0.14	2.837	0.397	010	3	0.42
d	0.12	3.059	0.367	011	3	0.36
e	0.36	1.474	0.531	00	2	0.72
Summe	1.00	H =	2.198			K = 2.26

Der Huffman-Code ist ein optimaler, eindeutig dekodierbarer Code.

Nachteile der Huffman-Kodierung

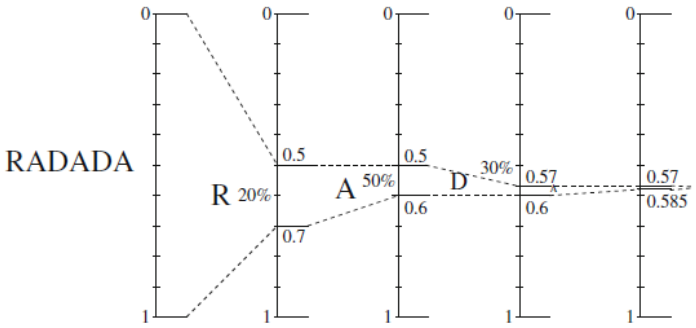
- Wenn der Codebaum nicht bekannt ist, muss man diesen Verarbeitungsaufwendig berechnen.
- Bei einer gleicher Wahrscheinlichkeit unterschiedlicher Knoten können unterschiedliche Codewörter für das gleiche Zeichen resultieren. Deshalb muss man bei Dekodieren denselben Algorithmus verwenden wie beim Kodieren oder der Huffman-Codebaum muss mit der kodierten Nachricht zusammen abgespeichert bzw. übertragen werden.

Die Huffman-Kodierung war Jahrzehnte lang das am häufigsten eingesetzte Verfahren für die Kodierung. In den letzten Jahren wurde sie aber immer mehr von der arithmetischen Kodierung abgelöst.

Arithmetische Kodierung

Die arithmetische Kodierung berechnet die Codewörter bereits beim Lesen der Daten („on-the-fly"), ohne dass sie die Codewörter aller Zeichenfolgen kennt. Das Prinzip der arithmetischen Kodierung ist die Darstellung von Daten als Intervalle in den rationalen Zahlen. Ausgehend vom Intervall [0,1) am Anfang wird dieses Intervall mit jedem neuen Symbol verkleinert, wobei es entsprechend der Auftrittswahrscheinlichkeiten proportional aufgeteilt wird.

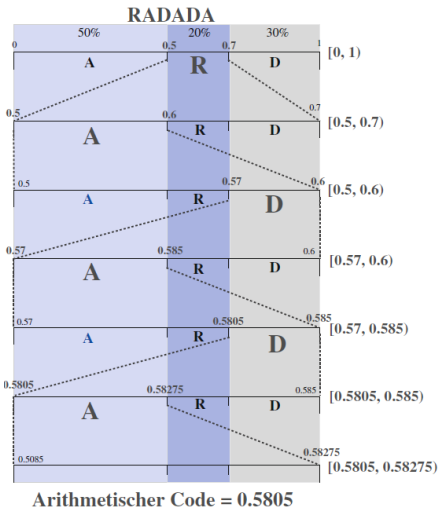
Mit folgenden Auftrittshäufigkeiten soll nun das Wort „RADADA“ arithmetisch kodiert werden: A: 50%; R: 20%; D:30%. Die Gleitpunktzahl, die die zu kodierende Nachricht darstellt, wird nun durch sequenzielle Intervallschachtelung bestimmt.



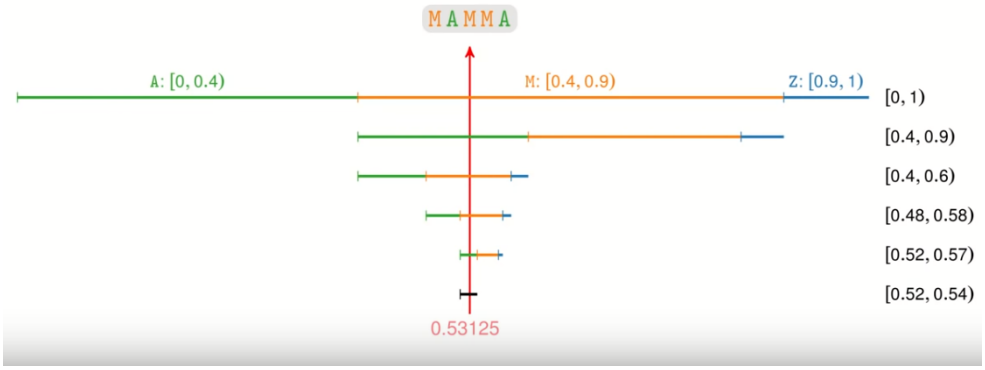
Zuerst wird das Zeichen eingelesen und das dazugehörige Teilintervall bestimmt. Dieses Teilintervall dient nun als Grundlage für das nächste Zeichen, indem die Zeichenwahrscheinlichkeit abgebildet werden. Dieser Vorgang wird nun bis zum Einlesen des Zeichens wiederholt. Zuletzt wird die Gleitpunktzahl des letzten Intervalls als „Code“ gewählt. Dabei sollte darauf geachtet werden, dass diese mit möglichst wenig Bits darstellbar ist.

Beim Dekodieren hat man das Problem, dass man nicht weiß, wann das Dekodieren zu beenden ist, also wann alle Zeichen dekodiert sind. Dieses Problem lässt sich auf zwei Arten lösen:

1. Man speichert bzw. überträgt die Anzahl der zu dekodierenden Zeichen zusammen mit der arithmetischen „Codezahl“.
2. Man kennzeichnet das Ende der dekodierten Zeichenfolge mit einem speziellen Zeichen, wie z.B. EOF (end-of-file)



Zur leichteren Verständnissen wird ein Beispiel mit dem Wort „MAMMA“ gezeigt.



Zwei Probleme der arithmetischen Kodierung:

- Die immer kleiner werdenden Teilintervalle mit jedem neuen Zeichen.
- Die Auftrittswahrscheinlichkeit muss dem Dekodierenden bekannt sein.
  Folgende Möglichkeiten können das Problem lösen:
  - Verwendung der immer gleichen Auftrittswahrscheinlichkeiten, was meist eine geringe Kompressionsrate mit sich zieht.
  - Semi-adaptives Verfahren: Hier liest der Kodierer zuerst die Nachricht ein, um die Auftrittswahrscheinlichkeiten festzustellen und nach dem Kodieren mitabzuspeichern bzw. zu übertragen.
  - Adaptives Verfahren: Nach einer Anzahl von gesendeten Blöcken wird eine neue Auftrittswahrscheinlichkeit berechnet und mitgesendet.

Lempel-Ziv-Kodierung

Die großen unterschiedlichen Gruppen der wörterbuchbasierenden kodierenden Verfahren gehören der Lempel-Ziv (LZ) Kodierung an. Bei diesem Verfahren handelt es sich um ein dynamisches Verfahren. Die zu kodierenden Daten werden während des Kodieren neu generiert.
Die LZ-Algorithmen lassen sich in zwei Gruppen einteilen:

1. Komprimierung von Wiederholungen (LZ77, LZSS)
   Diese Gruppe versucht die zu komprimierende Zeichenfolge in der schon verarbeiteten Datenmenge zu finden. Anstatt Wiederholungen wird dann lediglich ein Zeiger auf das letzte Auftreten derselben abgespeichert. Das Wörterbuch wird hier also als durch die bereits verarbeitete Datenstruktur repräsentiert (implicit dictionary).
2. Erzeugen eines Wörterbuchs aus Teilfolgen (LZ78, LZW, LZC)
   Bei dieser Gruppe wird während des Kompressionsvorganges ein Wörterbuch aus Teil-Zeichenfolgen erzeugt, die den zu komprimierenden Daten auftreten. Ist eine Zeichenfolge bereits im Wörterbuch vorhanden, wird diese durch den Index ihres Eintrags ersetzt. Dieses Wörterbuch wird während des Dekodierens dynamisch generiert.

Wichtige Begriffe des LZ-Algorithmus:

- Eingabedatenstrom: Folge von Zeichen, die komprimiert werden soll.
- Kodierungsposition: Position des Zeichens im Eingabedatenstrom, das gerade kodiert werden soll.
- Datenfenster: Intervall der Größe n, das den zuletzt verarbeiteten n Zeichen des Eingabedatenstromes entspricht.
- Zeiger: bezeichnet die Position einer Zeichenkette im Datenfenster; dieser Zeiger wird durch ein Tupel (Position, Länge) dargestellt.
- Vorausschaupuffer: Zeichenfolge von der Kodierungsposition bis zum Ende des Eingabedatenstroms.

- Wörterbuch: Tabelle aus Zeichenketten; zu jeder Zeichenkette im Wörterbuch gehört dabei genau ein Code als eindeutiger Index auf das Wörterbuch.
- Präfix: beliebig lange Folge von Zeichen, die genau einem Zeichen vorangestellt ist.

LZ77-Algorithmus

Der Lempel-Ziv-77-Algorithmus durchsucht das Datenfenster nach der längsten Zeichenkette, die mit der an der Kodierungsposition beginnenden Zeichenkette übereinstimmt. Da es natürlich vorkommen kann, dass überhaupt keine Übereinstimmungen gefunden werden, reicht es nicht aus nur den Zeiger (Position, Länge) auszugeben. Dazu wird nach dem Zeiger das darauffolgende Zeichen angefügt. Wird keine übereinstimmende Zeichenkette gefunden wird ein sogenannter Nullzeiger (0,0) gefolgt vom Zeichen an der Kodierungsposition ausgegeben.

Der LZ77-Algorithmus geht wie folgt vor:

1. Die Kodierungsposition des Eingabedatenstroms an den Anfang setzen
2. Die längste im Datenfenster übereinstimmende Zeichenkette im Vorausschaupuffer.
3. Den Zeiger (Pointer) und das erste nicht passende Zeichen ausgeben.
4. Die Kodierungsposition um die Länge der zuletzt gefundenen Zeichenkette +1 bewegen, bis das Ende erreicht wurde.

Beispiel:

```
Position:      0 1 2 3 4 5 6 7 8
Datenmenge:    A A B C A A A B C
```

Die Kleinbuchstaben im folgenden Bild ist der Inhalt des Datenfensters (DF), die Großbuchstaben der Inhalt des Vorausschaupuffers (VP) und der Doppelpunkt die Kodierungsposition.

Kodierungsposition	DF:VP	Position:Übereinstimmung	Ausgabe (Position, Länge)Zeichen
0	:AABCAAABC	-	(0,0)A
1	a:ABCAAABC	0:A	(0,1)B
3	aab:CAAABC	-	(0,0)C
4	aabc:AAABC	0:AA	(0,2)A
7	aabcaaa:BC	3:BC	(2,1)C[1]

1 Hier wird anstatt der gefundenen Zeichenkette *BC* nur der Code für *B* und das Zeichen *C* ausgegeben, da beim Dekodieren immer ein Code gefolgt von einem Zeichen erwartet wird. Wäre hier nicht das Ende des Eingabedatenstromes erreicht, würde natürlich der Code für *BC* ausgegeben werden.

Der Code sieht nach der Codierung folgender Maßen aus:

```
(0,0)A (0,1)B (0,0)C (0,2)A (2,1)C
```

Die Dekodierung ist einfach: Das Datenfenster wird auf dieselbe Weise verwendet wie bei der Codierung. Nachfolgend die schrittweise Dekodierung:

Eingabe	DF:Ausgabe
(0,0)A	-:A
(0,1)B	a:AB
(0,0)C	aab:C
(0,2)A	aabc:AAA
(2,1)C	aabcaaa:BC

Bekannte Komprimierungsprogramme, die den LZ77 verwenden: *arj, zip, pkzip, zoo*

LZ78-Algorithmus

Der LZ78-Algorithmus verweist mit dem Code auf einen Index im Wörterbuch, welches während des Kodieren und Dekodieren aufgebaut wird.
Zu Beginn ist das Wörterbuch und die Präfix-Zeichenkette leer und es wird ein Zeichen z aus dem Eingabedatenstrom gelesen. Ist die Zeichenkette „Präfix + z“ im Wörterbuch enthalten, so wird das Präfix um z erweitert. Dieses Einlesen und Erweitern wird so lange fortgesetzt, bis „Präfix + z“ nicht im Wörterbuch enthalten ist. In diesem Fall werden dann der dem Präfix repräsentierende Code und das momentane Zeichen z ausgegeben und „Präfix + z“ noch im Wörterbuch eingetragen. Anschließend wird mit einem neuen Präfix begonnen. Für den Sonderfall, dass das Präfix leer ist und z nicht im Wörterbuch vorhanden ist, wird ein spezieller Code, der einen leeren String repräsentiert, und z ausgegeben und z in das Wörterbuch eingetragen.

Der Algorithmus des LZ78 ist somit:

1. Zu Beginn sind Präfix und Wörterbuch leer
2. z:= nächstes Zeichen aus dem Eingabedatenstrom
3. Ist „Präfix + z“ im Wörterbuch?
   wenn ja: Präfix:= „Präfix + z“
   wenn nein: Gib den Code für Präfix und z aus.
   Trage „Präfix + z“ im Wörterbuch ein und leere Präfix
4. Ist das Ende des Eingabedatenstroms erreicht?
   wenn nein: Gehe zu Schritt 2
   wenn ja: Ist Präfix nicht leer, gib seinen korrespondierenden Code aus.

Beispiel:

```
Position:      0 1 2 3 4 5 6 7 8
Datenmenge:    A A B C A A A B C
```

Nachfolgend wird schrittweise die LZ78-Kodierung gezeigt. Kleinbuchstaben sind immer schon verarbeitete Zeichen und Großbuchstaben die noch zu verarbeitenden Zeichen.

Kodierungs-position	verarbeitet:nicht verarbeitet	Code:Wörterbucheintrag	Ausgabe Code,Zeichen
0	:AABCAAABC	1:A	0,A
1	a:ABCAAABC	2:AB	1,B
3	aab:CAAABC	3:C	0,C
4	aabc:AAABC	4:AA	1,A
6	aabcaa:ABC	5:ABC	2,C

Das erhaltene Wort aus der Kodierung lautet:
`0,A 1,B 0,C 1,A 2,C`

Bei der Dekodierung ist das Wörterbuch gleich aufgebaut, wie nach der Kodierung. Zuerst wird beim Dekodieren die durch (Position, Länge) angegebene Zeichenkette ausgegeben und anschließend das dem Zeiger z nachgestellten Zeichen.

Eingabe	Wörterbucheintrag	Ausgabe
0,A	1:A	A
1,B	2:AB	aAB
0,C	3:C	aabC
1,A	4:AA	aabcAA
2,C	5:ABC	aabcaaABC

LZW-Algorithmus

Der LZW ist die Weiterentwicklung des LZ78, der im Jahre 1984 von Terry Welch veröffentlicht wurde. Der LZW ist die bekannteste LZ-Variante. Der wesentliche Unterschied zu LZ78 besteht darin, dass nur mehr Codes ausgegeben werden und keine Zeichen. Dies setzt voraus, dass zu Beginn der Kodierung/Dekodierung für jedes im Eingabealphabet vorkommende Zeichen ein entsprechender Eintrag im Wörterbuch existiert. Da deshalb jede Kodierung mit einem Präfix der Länge 1 beginnt, wird bei der Kodierung immer zuerst im Wörterbuch nach Zeichenketten der Länge 2 gesucht. Außerdem muss das erste Zeichen des neuen Präfix mit dem letzten Zeichen des zuvor eingetragenen Zeichenkette ident sein, da sonst bei der Dekodierung dieses Zeichen, das bei der LZ78 explizit ausgegeben wurde, fehlt und das Wörterbuch nicht richtig aufgebaut werden könnte.
Der Algorithmus ist wie folgt:

1. Zu Beginn ist das Präfix leer und das Wörterbuch enthält für jedes im Dateneingabestrom vorkommende Zeichen einen Eintrag.
2. z:= nächstes Zeichen des Eingabedatenstroms.
3. Ist „Präfix + z“ im Wörterbuch?
   wenn ja: Präfix := „Präfix + z“
   wenn nein: Gib den Code für Präfix aus und trage „Präfix + z“ im Wörterbuch ein.
   Präfix := z
4. Ist das Ende des Eingabedatenstromes erreicht?
   wenn nein: Gehe zu Schritt 2.
   wenn ja: Ist Präfix nicht leer, gib seinen korrespondierenden Code aus.

Beispiel für die Zeichenfolge: ABBABABAC, wobei Z = Zeichen, P = Präfix, P + Z = „Präfix + Zeichen“:

			P+Z im Wörterbuch?				
			ja	nein			
Z	P	P+Z	Neues P	Ausgabe: … (für P aus Wörterbuch)	Wörterbuch-eintrag 1:A, 2:B, 3:C	Neues P	gelesen bisher
A		A	A		…		A
B	A	AB		1	…, 4:AB	B	AB
B	B	BB		2	…, 5:BB	B	ABB
A	B	BA		2	…, 6:BA	A	ABBA
B	A	AB	AB		…		ABBAB
A	AB	ABA		4	…, 7:ABA	A	ABBABA
B	A	AB	AB		…		ABBABAB
A	AB	ABA	ABA		…		ABBABABA
C	ABA	ABAC		7	…, 8:ABAC	C	ABBABABAC
-	C			3			(am Ende)

Der entstandene Code ist demnach: 1 2 2 4 7 3

Algorithmus zur LZW-Dekodierung

Da beim LZW auf Zeichen gänzlich verzichtet wurde, muss man das letzte Wort der vorherigen Zeichenfolge in den Wörterbucheintrag der aktuellen Zeichenfolge hinzufügen. Dies hat zur Folge, dass man beim Kodieren um einen Eintragt „hinterherhinkt“.
Algorithmus:

1. Das Wörterbuch enthält zu Beginn für jedes im Eingabedatenstrom vorkommendes Zeichen einen Eintrag.
2. Code := erster Code aus dem Eingabedatenstrom (immer 1 Zeichen)
3. Gib „Code“ aus.
4. Merke *Code* in *altCode*.
5. Code := nächster Code im Eingabedatenstrom.
6. Ist Code im Wörterbuch?
   wenn ja:
   a. Gib „Code“ aus.
   b. Präfix := „altCode“.
   c. Zeichen := erstes Zeichen von „Code“.
   d. Trage „Präfix + Zeichen“ in das Wörterbuch ein.

   wenn nein:

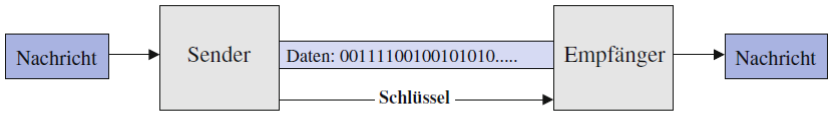
   a. Präfix := „altCode“
   b. Zeichen := erstes Zeichen von „altCode“.
   c. Trage „Präfix + Zeichen“ (= „Code“) in Wörterbuch ein UND gib es aus.
7. Ist Ende des Eingabedatenstroms noch nicht erreicht, dann gehe zu Schritt 4

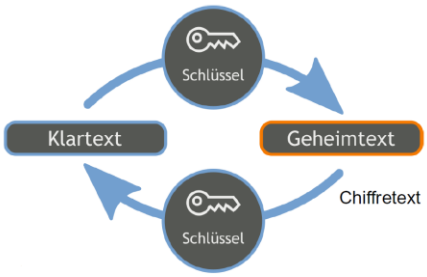
Bekennte Vertreter, die dieses Verfahren verwenden, sind GIF und TIFF.

YouTube-Link zu einem Beispiel: https://www.youtube.com/watch?v=dLvvGXwKUGw

Kryptografie

Elemente, die für eine sichere Kommunikation zwischen zwei Endpunkten erforderlich sind , nennt man in der Gesamtheit *Kryptosysteme*. Folgende Bilder zeigen die prinzipielle Struktur eines typischen Kryptosystems:





Der Sender sendet die Botschaft (Klartext) and den Empfänger, indem er den Klartext in einen sogenannten Chiffretext umwandelt, wozu er einen Verschlüsselungsalgorithmus und gewisse Schlüsselparameter verwendet. Um die Botschaft entschlüsseln zu können benötigt der Empfänger den passenden Entschlüsselalgorithmus und die gleichen Schlüsselparameter.

Allgemein gilt, dass ein Kryptosystem umso sicherer ist, seine Benutzung jedoch umso komplizierter, je mehr Schlüsselparameter vorhanden sind.

Einfache Verschlüsselungsmethoden

Cäsar-Chiffre

Zu den einfachsten und ältesten Verschlüsselungsmethoden gehört die sogenannte Cäsar-Chiffre: Wenn ein Buchstabe im Klartext der n-te Buchstabe des Alphabets ist, so ersetzt man diesen durch den (n+k)-ten Buchstaben, wobei k eine feste Zahl ist. Fun Fact: Julius Cäsar verwendete immer k = 3.

Beispiel mit k = 1:

Klartext: BOTSCHAFT
Chiffretext: CPUTDIBGU

Diese Methode ist ziemlich unzuverlässig, da nur k erraten werden muss um die Nachricht zu entschlüsseln.

Chiffre mit eigener Zuordnungstabelle

Eine weitaus bessere Methode, indem man jeden Buchstaben eine andere, zufällige Zuordnung vorgibt.

```
_ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
D I E K A T Z L G M S C H N _ B F J O P Q R U V W X Y
```

dann würde die Botschaft wie folgt aussehen:

Klartext: DIES_IST_EIN_GEHEIMTEXT
Chiffretext: AMTPDMPQDTM_DLTGTMNQTWQ

Diese Verschlüsselung ist wesentlich schwieriger zu entschlüsseln, wenn die entsprechende Zuordnungstabelle nicht bekannt ist. In diesem Fall müsste man ungefähr 27! (ca. 10^{28}) Tabellen ausprobieren, um eine sicherere Entschlüsselung zu gewährleisten.
Da jedoch gewisse Buchstaben öfter vorkommen (z.B. E) als andere (z.B. Q), können Kryptoanalytiker einen guten Anfang finden.

Vigenère-Verschlüsselung

Eine Möglichkeit, diesen Weg zur Entschlüsselung zu erschweren, besteht in der Verwendung von mehr als einer Tabelle. Diese werden im Allgemeinen Vigenère-Chiffre genannt.

Bei jedem Schritt wird der Index des Buchstaben im Schlüssel zum Index des Buchstabens im Klartext addiert.

```
Schlüssel:      ABCABCABCA
Klartext:       GEHEIMTEXT
Chiffretext:    HGKFKPUGAU
```

Wenn er Schlüsse genauso lange ist, wie der Klartext, so spricht man vom **Vernam-Chiffre**. Dies ist nachgewiesen die sicherste Methode einen Text zu verschlüsseln. Diese Methode wird jedoch nur selten eingesetzt, da durch die Übermittlung des Chiffretexts und des Schlüssel über dasselbe Medium, das Sicherheitsrisiko immens hebt. Wird der Schlüssel über z.B. durch die Post übermittelt liegt das Risiko am Übermittlungsweg.

Verschlüsselung mittels Zufallsfolgen

Hat man einen binären Klartext, benötigt man natürlich auch einen binären Schlüssel. In diesem Fall entsteht ein Chiffretext mittels XOR-Verknüpfung. Auch die Entschlüsselung und das Finden des Schlüssels wird die XOR-Verknüpfung verwendet, sobald man 2 Texte kennt, was eine ziemlich nützliche Eigenschaft ist.

- Verschlüsselung

  ```
  Schlüssel:  11010110
  Klartext:   01101111  (XOR)
  ----------------------
  Chiffretext: 10111001
  ```
- Entschlüsselung

  ```
  Schlüssel:  11010110
  Chiffretext: 10111001  (XOR)
  ----------------------
  Klartext:   01101111
  ```
- Finden des Schlüssels

  ```
  Chiffretext: 10111001
  Klartext:   01101111  (XOR)
  ----------------------
  Schlüssel:  11010110
  ```

Private Fernsehgesellschaften (Pay-TV) wenden oft dieses Verfahren an, um sich vor „Schwarzseher“ zu schützen. Dabei wird das gesendete Signal verschlüsselt und der Schlüssel selbst auch kodiert. Der Schlüssel wird meist jedes Monat geändert, wodurch eine Kopie des Schlüssels (wenn der Mitgliedsbeitrag bezahlt wurde) mittels Satellit übertragen wird und eine Kopie im Dekodiergerät gespeichert wird.

Kryptosysteme mit öffentlichen Schlüsseln

Bei den bisher kennengelernten Chiffresystemen gilt immer Folgendes:

1. Wer verschlüsseln kann, kann auch entschlüsseln.

2. Je zwei Partner müssen einen gemeinsamen geheimen Schlüssel austauschen.

Die zweite Eigenschaft ist ein Nachteil, während man die erste als Vorteil ansieht. Das Public-Key-System zeichnen sich dadurch aus, dass sie sich von der ersten Eigenschaft so weit wie möglich entfernen und die zweite überhaupt nicht ausweisen, was bedeutet, dass der Schlüssel nicht mehr geheim, sondern für jedermann zugänglich ist.

Die Idee der öffentlichen Kryptosysteme mit öffentlichen Schlüssel besteht in der Verwendung eines „Telefonbuchs mit Schlüsseln" für die Verschlüsselung. Jedermanns Schlüssel für die Verschlüsselung (mit P bezeichnet) ist öffentlich bekannt. Der Schlüssel einer Person könnte z.B. neben ihrer Nummer im Telefonbuch angegeben sein. Jedermann besitzt außerdem einen privaten Schlüssel zur Entschlüsselung (mit S bezeichnet), den sonst niemand kennt.

Um eine Botschaft M zu übermitteln, sucht der Absender den öffentlichen Schlüssel P des Empfängers heraus, verschlüsselt seine Botschaft damit, und übermittelt dann die Chiffre Botschaft $C = P(M)$. Der Empfänger verwendet seinen privaten Schlüssel S für das Entschlüsseln der chiffrierten Botschaft und erhält somit die originale Botschaft $M = S(C) = S(P(M))$. Damit dieses System funktioniert, müssen zumindest die folgenden Bedingungen erfüllt sein:

1. S(P(M)) = M für jede Botschaft M.
2. Alle Paare (S, P) sind verschieden.
3. Das Finden des privaten Schlüssels S bei Kenntnis von P ist nahezu unmöglich.
4. Das Entschlüsseln der Botschaft M ohne Kenntnis des Schlüssels S ist nahezu unmöglich.
5. Sowohl S als auch P lassen sich leicht berechnen. Diese Bedingung ist für die praktische Verwendbarkeit des Systems unverzichtbar.

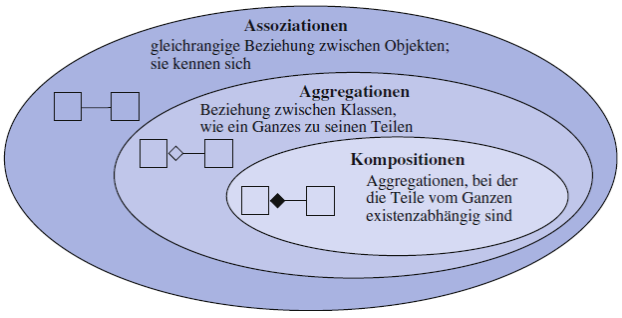
Solche Public-Key-Systeme wurden 1976 von W. Diffie und M. Hellman vorgeschlagen, jedoch wurde noch keine Methode angegeben, die alle diese Bedingungen erfüllt. Bald danach wurde eine derartige Methode von R. Rivest, A. Shamir und L. Adleman (RSA) gefunden.

Unified Modelling Language (UML)- Diagramme

Statische Modellierung

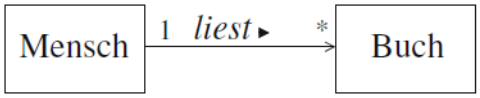
Bei einer statischen Modellierung handelt es sich um Diagramme, die eine strukturierte Aufstellung bzw. die Beziehungen zwischen einzelnen Klassen/Komponenten zeigen.

Grundbegriffe



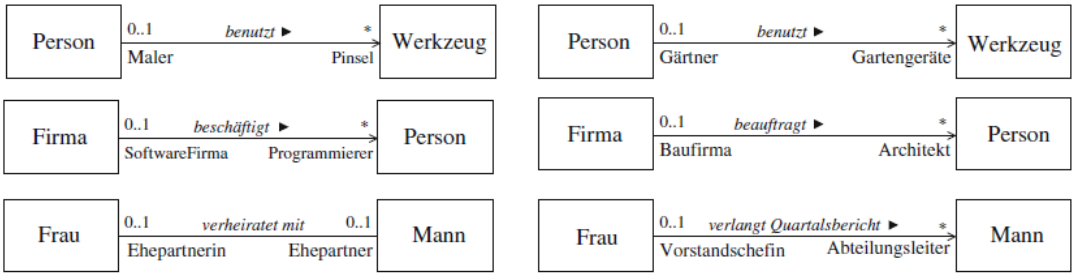
Assoziation

Über eine Assoziation zueinander kennen sich die Objekte zwar untereinander, haben aber sonst keinerlei stärkeren Beziehungen zueinander.



Im einfachsten Fall wird eine Assoziation nur in Form einer Linie zwischen zwei Klassen angegeben. Üblicherweise werden aber zusätzlich noch viele weiteren Informationen angegeben:

- Text auf der Linie: wird *kursiv* geschrieben, meist mit kleinem, ausgefüllten Dreieck daneben, dessen Spitze die Leserichtung angibt; beschreibt die Beziehung näher
- Anzahl (Kardinalität/Multiplizität): wie viele Objekte der einen Seite mit wie vielen Objekten der anderen Seite verbunden sind.
  1 genau eins
  1, 3, 5 eins oder drei oder fünf
  0..* keines, eines oder mehrere
  1..* eines oder mehrere
  0..3 null bis drei
  1..4,8 eins bis vier oder acht
  * entspricht 0..*
  Ein Fehlen dr Kardinalitätsangabe wird immer als 1 interpretiert. Liegt das Minimum bei null, ist die Beziehung optional.
- Rollenname: die Rolle der beteiligten Objekte z.B. Mensch, Buch, Gärtner, Werkzeug usw.

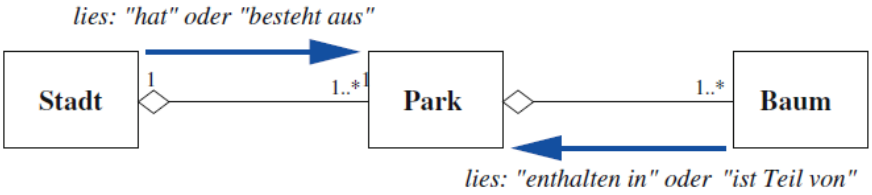


Aggregation

Eine Aggregation ist eine Sonderform der Assoziation. Bei einer Aggregation handelt es sich ebenfalls um eine Beziehung zwischen zwei Klassen, jedoch mit der Besonderheit, dass die Klassen zueinander in Beziehung stehen wie ein Ganzes zu seinen Teilen. Eine Aggregation ist die Zusammensetzung eines Objektes aus einer Menge von Einzelteilen, z.B.:

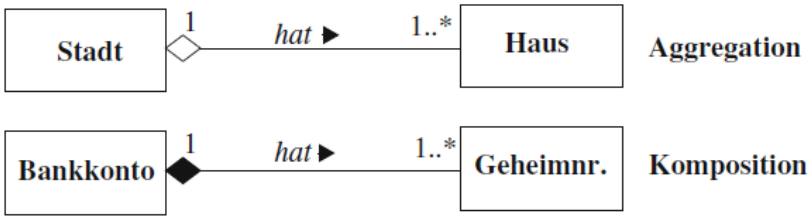
- Ein Ganzes zu seinen Teilen (z.B. Computer aus CPU, Speicher usw.)
- Ein Behälter und sein Inhalt (z.B. Stack und seine Elemente)
- Eine Kollektion und ihre Elemente (z.B. Verein und seine Mitglieder)

Aggregationen sind Hat-Beziehungen: Eine Stadt hat z.B. Straßen, Häuser, Parks usw. Eine Stadt ist also eine Aggregation aus Straßen, Häusern Parks usw. Auch diese Teile können wiederum Aggregationen sein: Ein Park z.B. aus Bäumen, Bänken, Fußwegen usw. Aggregationen werden manchmal auch als Teil-Ganze-Hierarchie bezeichnet. Um eine Beziehung als Aggregation zu kennzeichnen, wird auf der Seite des Ganzen eine (leere) Raute gezeichnet.



Komposition

Eine Sonderform der Aggregation wenn die Einzelteile vom Aggregat existenzabhängig sind. In diesem Fall spricht man von einer Komposition. Ein Beispiel dafür ist die Beziehung zwischen Geheimnummer und Bankkonto. Wenn das Ganze (z.B. das Bankkonto) gelöscht werden soll, so werden auch alle existenzberechtigten Einzelteile (z.B. die Geheimnummer) mitgelöscht. Bei einer normalen Aggregation würde dagegen nur das eine Objekt und die Beziehung zum anderen Objekt gelöscht werden, aber das andere Objekt würde weiterhin bestehen. Ein Beispiel für eine normale Aggregation ist die Beziehung. „Stadt hat Straßen“. Die Straßen gehören zwar notwendiger Weise zu einer Stadt, können aber auch ohne Stadt existieren, weshalb nun eine Aggregation und keine Komposition vorliegt.



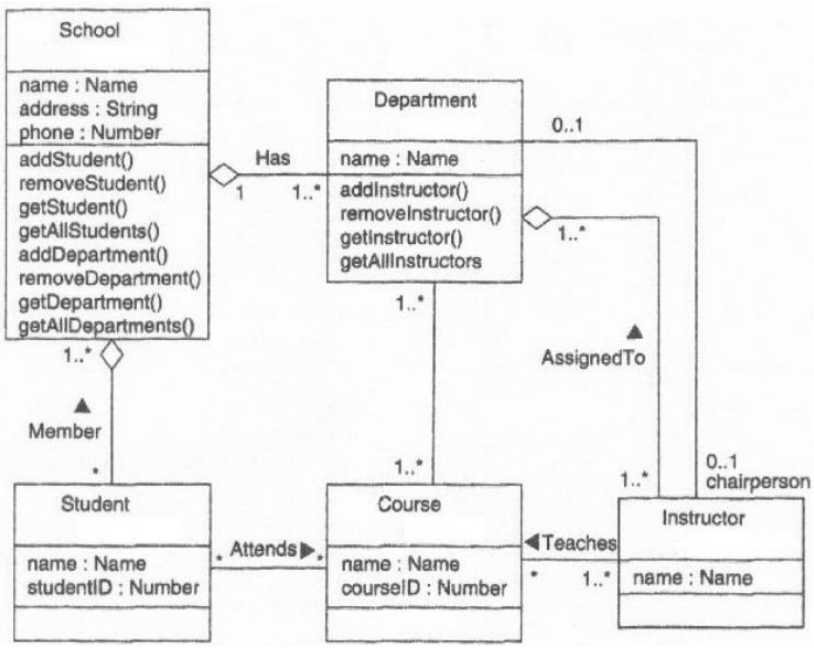
Klassendiagramm

Eine Klasse besteht aus drei verschiedenen Bestandteilen:

Klassenname
Attribute
Methoden

- Attribute:
  attributname: Datentyp (int, char, str, float, double, bool,…)
- Methoden:
  methodenname()
  Falls die Mehode Eingabeparameter besitzt:
  methodenname(attributname: Datentyp):Datentyp_rueckgabetyp //falls ein Rückgabetyp existiert.

Beispiel: Universität



Dynamische Modellierung

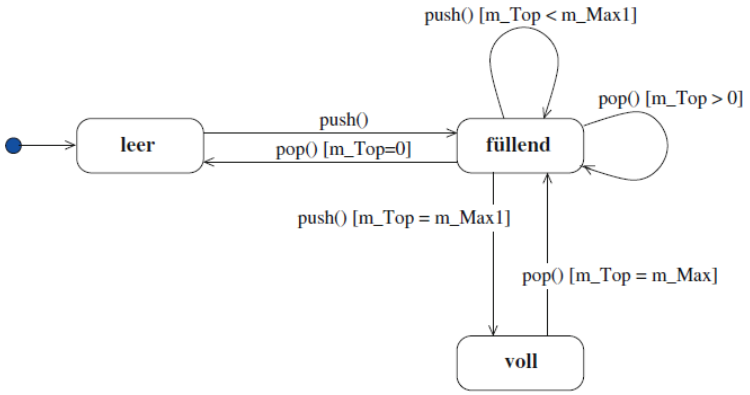
Die dynamische Modellierung dient zur funktionalen und verhaltensbasierten Darstellung von Funktionen. Wie z.B. im Zustandsdiagramm kommen Objekte von einem Zustand in den nächsten.

Zustandsdiagramm

Im Zustandsdiagramm wird neben werden neben den Zuständen auch die Zustandsübergänge (Transitionen) modelliert. Dabei wird am Zustandsübergang die Nachricht oder das Ereignis angegeben, durch die der Zustandsübergang ausgelöst wird. In […] kann dabei eine zusätzliche Wächterbedingung angegeben werden, die für den Zustandsübergang ebenfalls erfüllt sein muss. Das Zustandsdiagramm wird im Englischen auch state diagram genannt und wird auch Zustandsübergangsdiagramm (state transition diagram) oder Endlicher Automat genannt.
Zustandsdiagramme sind für folgende Anwendungsfälle sehr Hilfreich:

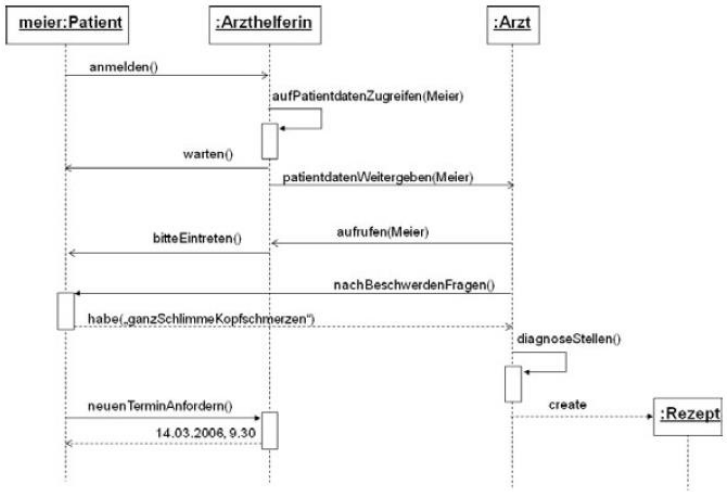
- Zur Überprüfung, ob sich ein Objekt stabil verhält und nicht in einen undefinierten Zustand gerät. Potenzielle Fehler können so im Vorfeld vermieden werden.
- Für den möglichst vollständigen Test einer Klasse. Eine Klasse sollte immer so entworfen werden, dass jede Methode zu jedem beliebigen Zeitpunkt aufgerufen werden kann, ohne dass das Objekt in einen undefinierten Zustand gerät.

Zustandsdiagramme sind also sehr hilfreich, um das Verhalten eines Objekts in seiner Gesamtheit zu betrachten.



Sequenzdiagramme

Sequenzdiagramme zeigen die Kommunikation zwischen ausgewählten Objekten für eine bestimmte Nachrichtensequenz, d. h. für ein bestimmtes Szenario auf.



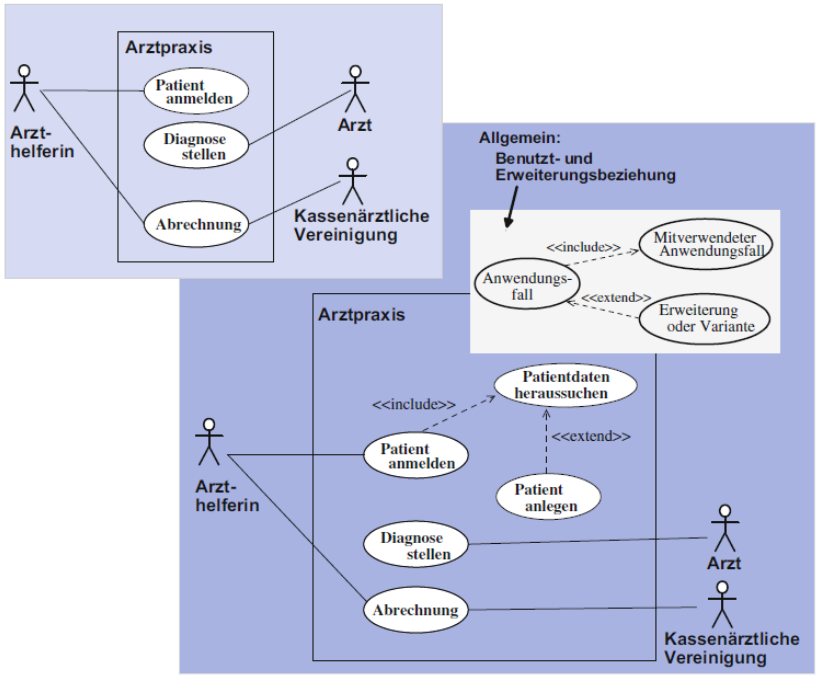
Bei Sequenzdiagrammen gelten unter anderem folgende Regeln:

- *Objekte* werden durch gestrichelte senkrechte Linien (Lebenslinien) notiert, an denen oben über der Linie der Name bzw. das Objektsymbol steht. Die Zeit verläuft von oben nach unten.
- Graue oder auch nicht ausgefüllte senkrechte Balken zeigen an, welches Objekt gerade die Programmkontrolle besitzt, d. h. welches Objekt gerade den *Steuerungsfokus* besitzt, also aktiv ist.
- *Nachrichten* werden als waagrechte Pfeile zwischen den Objekt-Linien gezeichnet. Auf ihnen wird die entsprechende Nachricht angegeben.
- *Synchrone Nachrichten*, die eine Rückantwort erwarten, damit ein Objekt seine Ausführung fortfahren kann, werden durch einen ausgefüllten Pfeil gekennzeichnet. Die Antwort auf eine Nachricht kann als Text (`antwort:=nachricht()`) oder als eigener gestrichelter Pfeil mit offener Pfeilspitze angegeben werden.
- *Asynchrone Nachrichten*, die keine Rückantwort erwarten, werden durch einen nicht ausgefüllten offenen Pfeil gekennzeichnet.
- Das Erzeugen von Objekten kann dargestellt werden, indem eine Nachricht direkt auf ein Objektsymbol trifft, das zuvor nicht existierte.

Anwendungsfalldiagramm (Use-Case-Diagram)

Dies zeigt den Zusammenhang zwischen Anwendungsfällen und den daran beteiligten Akteuren (Strichmännchen).

- Der Anwender wird als Akteur dargestellt. Ein Akteur ist eine außerhalb des Systems liegende „Klasse“.
- Ein Anwendungsfall beschreibt einen typischen Arbeitsablauf.



Die Beziehung zwischen dem eigentlichen Anwendungsfall und den herausgelösten Fällen wird durch <<include>> und <<extend>> beschrieben.

- <<include>> (Enthält-Beziehung): wird verwendet, wenn das gleiche Stück Anwendungsfallbeschreibung in unterschiedlichen Anwendungsfällen vorkommen kann. Der Anwendungsfall, von dem der Pfeil ausgeht, kann nie alleine ausgeführt werden, sondern nur mit dem Anwendungsfall, auf den der Pfeil zeigt.
- <<extend>> (Erweiterungs-Beziehung): wird verwendet, um Variationen eines Anwendungsfalls zu zeigen, beispielsweise Fehler- und Ausnahmesituationen, spezielle Abweichungen oder Erweiterungen des Standardfalles.

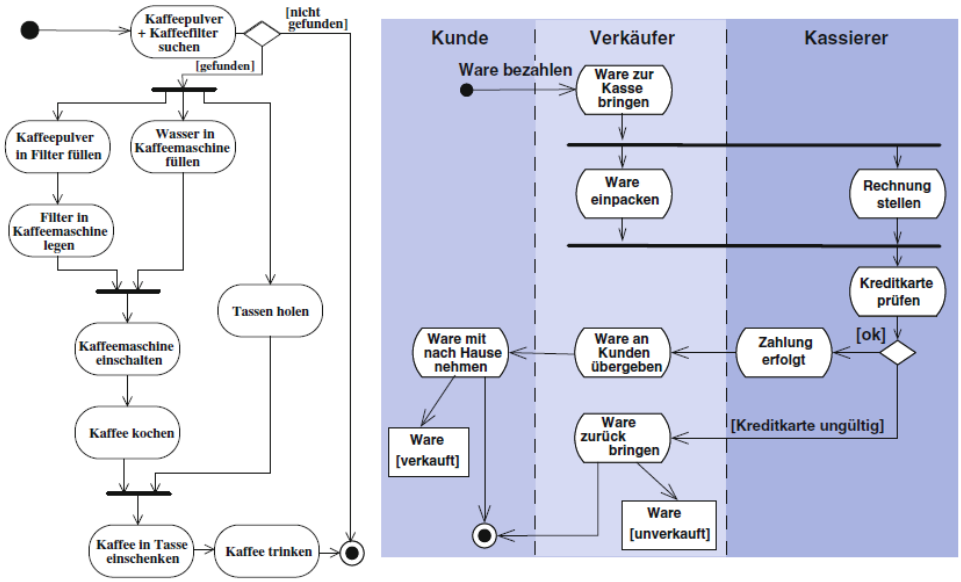
Wichtig ist, dass man sich über sämtlich möglichen Anwendungsfälle Gedanken macht und diese dem Kunden zur Gegenprüfung vorlegt, um sicherzustellen, dass das System vollständig analysiert ist. Das Use-Case-Diagramm zeigt eine Übersichtsdarstellung über Anwendungsfälle, vermittelt aber keine Details über den genauen Ablauf und die unterschiedlichen Szenarien, die dabei auftreten können. Diese werden für jeden Anwendungsfall in einer separaten Anwendungsfallbeschreibung per Text oder im **Aktivitätsdiagramm** vorgegeben.

Aktivitätsdiagramm

- Im Diagramm werden die einzelnen Aktionen und deren Zusammenhänge notiert, z.B. ob die Aktionen sequenziell oder parallel stattfinden, ob sie von irgendwelchen Bedingungen abhängen usw.

- Neben den Aktionen können auch Zustände notiert werden, wenn durch die Aktion ein Zustandswechsel erfolgt. Sie werden deshalb auch als spezielle Form des Zustandsdiagramm betrachtet. Der Fokus liegt hier aber eindeutig auf den Aktionen
- Sie ähneln prozeduralen Flussdiagrammen, allerdings sind im Aktivitätsdiagramm die Aktionen eindeutig Objekten zugeordnet.

Aktivitätsdiagramme können auch in Verantwortlichkeitsbereiche aufgeteilt werden (im folgenden Bild der rechte Teil), wobei jede Aktion dann genau einem Verantwortlichkeitsbereich zugeordnet ist.



Aktivitätsdiagramme werden auch oft im Zusammenhang mit Anwendungsfallbeschreibungen eingesetzt.

Anmerkungen

Das Kapitel „Betriebssysteme und Rechnernetzte“ sowie das Kapitel „Software Engineering (UML)“ wurden (noch) nicht zu 100% vervollständigt. Bei Ersterem fehlen noch Synchronisationsmechanismen, Speicherverwaltung, Dateiverwaltung und Echtzeitsysteme. Bei zweiterem fehlen noch z.B. das V-Modell, Wasserfallmodell, eXtreme Programming).

Diese Zusammenfassung enthält keine Erklärung für sog. Pseudocodes und enthält auch keine für die Prüfung relevante Zusammenfassung aller Codes. Codes und Pseudocodes sind (meist) subjektiv und jeder Programmierer programmiert verschieden. Deshalb wurde auf dieses Kapitel gänzlich verzichtet. Die angegebenen Codes sind also nicht im Pseudocode geschrieben.

Dies ist eine inoffizielle Zusammenfassung. Die Professoren der Lehrveranstaltung haben im Bezug dieser Zusammenfassung keine Beziehung und übernehmen demnach keine Haftung. Alle Angaben sind ohne Gewähr auf Vollständigkeit und Richtigkeit (sowohl Inhaltlich als auch Grammatikalisch und Orthographisch). Demnach ist das Bestehen der Prüfung(en) nur durch Benutzung dieser Fassung kein Garant. Einige Textpassagen sind 1:1 vom Buch „Grundlagen der Informatik“ von Helmut Herold, Bruno Lurz und Jürgen Wohlrab im Pearson-Verlag, welches auch der Literaturvorschlag der Lehrveranstaltung ist, (und nicht als Zitat angeführt) übernommen. Ebenfalls sind größtenteils der Grafiken und Bilder aus dem Buch entnommen. Ausnahmen:

- Jeweils eine Grafik zur einfach- und doppelt-verketteten Listen (im Kapitel „Datenstrukturen und (Sortier-)Algorithmen“) sowie bei der arithmetischen Kodierung (im Kapitel „Datenkompression“) sind Screenshots aus YouTube-Videos.
- Die Grafik der verschiedenen Topologien ist unter https://upload.wikimedia.org/wikipedia/commons/a/af/NetzwerkTopologien.png zu finden.
- Die Grafik bei Heap-Sort im Kapitel Algorithmen und (Sortier-)Algorithmen ist unter https://de.wikipedia.org/wiki/Heapsort#/media/File:Heapsort.svg zu finden.