

# **Grundlagen der Informatik**

**Veranstaltungsbegleitendes Skript**

© 2010 - 2025 Dipl.Phys. Gerald Kempfer  
Gastdozent an der Berliner Hochschule für Technik

Internet: [www.bht-informatik.de](http://www.bht-informatik.de)  
E-Mail: [gerald@kempfer.de](mailto:gerald@kempfer.de)

Stand: 26. August 2025

# Inhaltsverzeichnis

<b>1. GRUNDBEGRIFFE</b> .....	<b>5</b>
1.1. INFORMATIK.....	5
1.2. COMPUTERPROGRAMM.....	5
1.3. DATEN.....	5
1.4. ELEKTRONISCHE DATENVERARBEITUNG.....	6
1.5. PROGRAMMIERSPRACHE.....	6
<b>2. ZAHLENSYSTEME</b> .....	<b>7</b>
2.1. DEZIMALSYSTEM.....	7
2.2. BINÄRSYSTEM.....	7
2.3. HEXADEZIMALSYSTEM.....	8
2.4. OKTALSYSTEM.....	8
2.5. UMRECHNEN ZWISCHEN DEN ZAHLENSYSTEMEN.....	8
2.6. BINÄRDARSTELLUNG VON GANZEN ZAHLEN MIT UND OHNE VORZEICHEN.....	8
<b>3. BOOLE'SCHE ALGEBRA</b> .....	<b>9</b>
3.1. EINFÜHRUNG.....	9
3.2. DEFINITION.....	9
3.3. AND (KONJUNKTION).....	10
3.4. OR (DISJUNKTION).....	10
3.5. NOT (NEGATION).....	10
3.6. XOR (EXKLUSIV-ODER, KONTRAVALENZ).....	11
3.7. LOGISCH ODER BITWEISE.....	11
<b>4. PROGRAMMENTWICKLUNG</b> .....	<b>12</b>
<b>5. PROGRAMMIERSTILE</b> .....	<b>13</b>
5.1. EINFÜHRUNG.....	13
5.2. NEUN HILFREICHE REGELN.....	13
5.3. FUNKTIONSHEDER.....	14
5.4. MODULHEADER.....	16
<b>6. ANSI-STEUERSEQUENZEN</b> .....	<b>17</b>
6.1. ALLGEMEINES.....	17
6.2. ESCAPE- UND CONTROL-SEQUENZEN.....	18
6.3. ANSI-CONTROL-SEQUENZEN.....	19
<b>7. AUTOMATEN</b> .....	<b>27</b>
7.1. EINFÜHRUNG.....	27
7.2. TOURING-MASCHINE.....	27
7.3. HALTEPROBLEM.....	27
7.4. ENTSCHEIDUNGSPROBLEM.....	27
7.5. BERECHENBARKEIT.....	27
<b>8. KOMPLEXITÄT</b> .....	<b>28</b>
8.1. EINFÜHRUNG.....	28
8.2. ARTEN DER KOMPLEXITÄT.....	28
8.3. O-NOTATION.....	28
8.4. RECHENREGELN FÜR KOMPLEXITÄTEN.....	29
8.5. LAUFZEIT-KOMPLEXITÄTEN DER GRUNDBAUSTEINE EINER PROGRAMMIERSPRACHE.....	29
8.6. GRENZEN DER O-NOTATION.....	30
<b>9. SORTIER-ALGORITHMEN</b> .....	<b>31</b>
9.1. SORTIEREN DURCH DIREKTES EINFÜGEN.....	31
9.2. SORTIEREN DURCH DIREKTES AUSWÄHLEN.....	32

9.3. BUBBLE-SORT.....	33
9.4. SHELL-SORT.....	34
9.5. QUICK-SORT.....	35
<b>10. LISTEN.....</b>	<b>39</b>
10.1. EINFACH VERKETTETE LISTEN.....	39
10.2. DOPPELT VERKETTETE LISTEN.....	43
10.3. STAPEL (STACKS).....	44
10.4. WARTESCHLANGEN (QUEUES).....	44
<b>11. SUCHVERFAHREN.....</b>	<b>45</b>
11.1. SEQUENTIELLE SUCHE.....	45
11.2. BINÄRE SUCHE.....	49
11.3. BERECHNETE SUCHE (HASHING).....	50
<b>12. TEXTSUCHE.....</b>	<b>51</b>
12.1. EINFACHE TEXTSUCHE.....	51
12.2. KNUTH-MORRISON-PRATT-ALGORITHMUS.....	51
<b>13. EINFÜHRUNG IN DIE GRAPHENTHEORIE.....</b>	<b>52</b>
13.1. ALLGEMEINE BEGRIFFE.....	52
13.2. BAUMSPEZIFISCHE BEGRIFFE.....	52
<b>14. BÄUME.....</b>	<b>53</b>
14.1. UNSORTIERTE BÄUME.....	53
14.2. SORTIERTE BÄUME.....	53
14.3. SUCHE IN BÄUMEN.....	53
<b>15. ASCII-TABELLEN.....</b>	<b>54</b>

# 1. Grundbegriffe

Im folgenden werden einige Grundbegriffe der Informatik vorgestellt und geklärt.

## *1.1. Informatik*

Der Begriff **Informatik** ist von den Begriffen Information und Mathematik (und/oder Elektrotechnik) abgeleitet. Obwohl dieser Begriff bereits seit 1957 in Veröffentlichungen verwendet wurde, setzte er sich erst 1968 im deutschen Sprachraum durch.

Die Informatik unterteilt sich im wesentlichen in drei Unterbereiche: Theoretische Informatik, Praktische Informatik und Technische Informatik. Darauf aufbauend gibt es die Bereiche Angewandte Informatik (z.B. Wirtschaftsinformatik), Didaktik der Informatik, Künstliche Intelligenz sowie Informatik und Gesellschaft.

In der **Theoretischen Informatik** werden die Grundlagen der Informatik behandelt. So werden hier die Grundlagen der verschiedenen Algorithmen entwickelt und deren Leistungen untersucht sowie die Grenzen der Algorithmen und auch die Grenzen der Lösbarkeit von Problemen erforscht. Dazu gehören die Automatentheorie, Berechnungstheorie, Graphentheorie, Komplexitätstheorie, Kryptologie usw.

In der **Praktischen Informatik** werden Computerprogramme entwickelt, um konkrete Problemen der Informatik lösen zu können. Dabei werden Datenstrukturen und Algorithmen entwickelt, um Informationen bearbeiten und speichern zu können. Aber auch die Entwicklung von Betriebssystemen, Compilern und Programmierwerkzeugen gehören dazu.

In der **Technischen Informatik** werden die hardwareseitigen Grundlagen der Informatik behandelt. Dazu gehören u.a. die Mikroprozessortechnik und die Rechnerarchitekturen.

## *1.2. Computerprogramm*

Ein **Computerprogramm** (meist nur kurz Programm genannt) ist ganz allgemein eine vollständige Arbeitsanweisung an eine Datenverarbeitungsanlage, um mit ihrer Hilfe eine bestimmte Aufgabe lösen zu können. Das Programm setzt sich aus einzelnen Befehlen zusammen, die die Verarbeitungsoperationen der Datenverarbeitungsanlage bei der Ausführung des Programms auslösen. Das Problem bei der Erstellung des Programms besteht darin, die richtigen Befehle auszuwählen und diese in eine der Aufgabenstellung entsprechend richtige Reihenfolge zu bringen.

Dabei muss unterschieden werden, ob mit einem Programm eine ausführbare Datei oder eine Quelltextdatei gemeint ist. Bei einer ausführbaren Datei liegt das Programm im Maschinencode vor. Der Maschinencode enthält Befehle, die der Prozessor eines Rechners direkt versteht und verarbeiten kann. Quelltextdateien (auch Quellcode- oder Programmcode-dateien genannt) sind dagegen Textdateien, die das Programm in einer Programmiersprache beinhalten. Diese Dateien können vom Prozessor nicht verstanden und folglich auch nicht verarbeitet werden.

## *1.3. Daten*

**Daten** sind Informationen, die durch Zeichen oder kontinuierliche Funktionen aufgrund bekannter oder unterstellter Abmachungen dargestellt werden. Die Zeichen werden dabei als digitale Daten und die kontinuierlichen Funktionen als analoge Daten bezeichnet (nach DIN ISO/IEC 2382).

Dabei ist der Begriff Daten die Mehrzahl von Datum – nicht im Sinne eines Kalenderdatums, sondern im Sinne von Datenelement.

Nach dieser Definition bestehen Daten aus einer Größe (Zahl) und einer Einheit; beispielsweise ist eine Temperatur (bestehend aus Zahl und Einheit) ein Datum. Die Zahl 123 für sich alleine dagegen ist kein Datum.

## **1.4. Elektronische Datenverarbeitung**

Die **elektronische Datenverarbeitung** (kurz EDV) ist die Verarbeitung von Daten durch elektronische Systeme, z.B. durch Rechner. Dabei bezieht sich die Verarbeitung der Daten auf die Eingabe, Speicherung, Übertragung, Transformation und Ausgabe von Daten. Häufig wird dies auch das EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe) genannt.

## **1.5. Programmiersprache**

Eine **Programmiersprache** ist ein Kommunikationsmittel zwischen Menschen und Rechner. Jede Programmiersprache ist eine formale Sprache und hat eine streng definierte Grammatik, ihre Syntax.

Der Maschinencode (auch Maschinensprache genannt) kann vom Prozessor direkt verstanden werden, aber von Menschen kann er im Allgemeinen nicht gelesen werden. Dazu werden Programme in einer Programmiersprache geschrieben, die von Menschen gelesen werden können. Damit der Prozessor diese Programme verstehen kann, muss der Quelltext erst von einem Compiler in den Maschinencode übersetzt werden. Dieser Vorgang wird Compilieren genannt. Das Ergebnis wird in einer ausführbaren Datei gespeichert.

Alternativ kann auch ein Interpreter verwendet werden: Hierbei wird zur Programmlaufzeit Befehl für Befehl einzeln aus der Quelltextdatei in Maschinencode übersetzt und dann auch gleich ausgeführt. Damit sind Programme, die mit einem Interpreter ausgeführt werden, langsamer als Programme, die vorher mit einem Compiler übersetzt wurden.

Eine dritte Variante kommt z.B. bei Java zum Einsatz: Hier werden die Quelltextdateien in einen Zwischencode (Bytecode) übersetzt. Dieser Zwischencode wird zur Programmlaufzeit von einer virtuellen Java-Maschine (im Prinzip ein Interpreter) übersetzt und ausgeführt.

Es wird zwischen hardwarenahen und höheren Programmiersprachen unterschieden (genau genommen gibt es noch eine Vielzahl weiterer Unterscheidungen wie Skriptsprachen, Datenbanksprachen, CNC zur Steuerung von Werkzeugmaschinen, usw.). Während bei den hardwarenahen Programmiersprachen wie z.B. Assembler der Vorteil darin liegt, dass die Hardware direkt angesprochen werden kann, sind höhere Programmiersprachen wie z.B. Delphi dafür konzipiert, komplexere und strukturiertere Programme zu erstellen. Die Programmiersprache C wird beiden Bereichen zugeordnet.

Bei der Erstellung eines Programmes sollte immer darauf geachtet werden, dass ein Programm leicht zu lesen (d.h. leicht zu warten und zu erweitern) und auch leicht zu handhaben (d.h. bedienerfreundlich) ist.

## 2. Zahlensysteme

Im Allgemeinen werden ganze Zahlen im Dezimalsystem dargestellt, da wir im Alltag gewohnt sind, mit dieser Zahlendarstellung umzugehen. In der Informatik werden aber noch weitere Zahlensysteme verwendet. Um die anderen Zahlensysteme besser verstehen zu können, wird zuerst das Dezimalsystem betrachtet.

### 2.1. *Dezimalsystem*

Das **Dezimalsystem** hat seine Bezeichnung von der Zahl 10 (deka) erhalten. Dabei wird jede Ziffer einer Dezimalzahl als Multiplikator einer Potenz der Basis 10 angesehen. Erlaubte Ziffern für eine Dezimalzahl sind die Ziffern 0 bis 9 (allgemein von 0 bis Basis - 1).

#### **Beispiel:**

Die Zahl 1234 setzt sich zusammen aus  $1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 = 1234$ . D.h. die erste Ziffer von rechts wird mit der Basis hoch 0 multipliziert, die zweite Ziffer von rechts wird mit der Basis hoch 1 multipliziert, die dritte Ziffer von rechts wird mit der Basis hoch 2 multipliziert usw. Dann werden die einzelnen Werte summiert und ergeben den Zahlenwert.

Nach diesem Schema können nun die anderen Zahlensysteme betrachtet und schnell erklärt werden. Zur besseren Unterscheidung der Zahlen anderer Zahlensysteme zu den Dezimalzahlen werden diesen Zahlen die Basis als tiefgestellter Index angehängen.

### 2.2. *Binärsystem*

Das **Binärsystem** ist ein Zahlensystem zur Basis 2 (bi). Erlaubte Ziffern sind daher nur die Ziffern 0 und 1. Binärzahlen werden meistens in Achter- oder manchmal in Vierergruppen dargestellt.

#### **Beispiel:**

Die Zahl  $01010101_2$  setzt sich nach dem oben angegebenen Schema also wie folgt zusammen:  $0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 85_{10}$ .

Binärzahlen sind in der Informatik sehr wichtig; denn eine Binärziffer stellt die kleinste Einheit dar (0 oder 1). Eine Binärziffer wird **Bit** (*Binary Digit*; englisch für Binärziffer) genannt. Die Bedeutung kann vielfältig sein: Neben der Zahlendarstellung kann ein Bit auch anders gedeutet werden, z.B. als Wahrheitswert (0: falsch und 1: wahr; später wird jede Zahl ungleich 0 für wahr stehen) oder für An und Aus (0: Strom fließt nicht und 1: Strom fließt).

8 Bit ergeben ein **Byte** (auch Oktett genannt). Damit wird der Zahlenbereich von  $00000000_2$  (entspricht  $0_{10}$ ) bis  $11111111_2$  (entspricht  $255_{10}$ ) abgedeckt. Oder es werden 8 Wahrheitswerte in einer Zahl gespeichert; dann wird diese Zahl auch Bitmaske genannt.

Aufgrund des Oktetts werden Binärzahlen meistens in Achtergruppen dargestellt. Manchmal wird das Oktett auch in zwei Vierergruppen unterteilt; diese 4 Bit werden dann **Nibble** genannt. Jedes Nibble kann Werte von 0 bis 15 enthalten und lässt sich damit direkt auch als eine hexadezimale Ziffer darstellen (siehe nächsten Abschnitt).

Weitere häufig verwendete Werte sind 16, 32 und 64 Bit. Sie werden im Zusammenhang mit Betriebssystemen bzw. Prozessoren verwendet. Dabei gibt die Bitzahl an, welche Zahlen direkt vom Prozessor und damit wie viele Speicheradressen (zumindest theoretisch) angesprochen werden können. Bei 16 Bit sind es 65.536 Speicheradressen ( $2^{16}$  Byte = 64 kByte; z.B. PCs mit 8086-Prozessoren), bei 32 Bit sind es 4.294.967.296 Speicheradressen ( $2^{32}$  Byte = 4 GByte; z.B. PCs mit Pentium-Prozessoren) und bei 64 Bit sind es 18.446.744.073.709.551.616 Speicheradressen ( $2^{64}$  Byte = 16 EByte) – wie gesagt: theoretisch!

Bei größeren Werten werden die aus der Physik bekannten Vorsilben wie z.B: Kilo, Mega, Giga usw. verwendet. So sind 1.000 Byte gleich 1 Kilobyte. Um aber mit Binärzahlen einfacher arbeiten zu können,

wird die nächstgelegene Zweierpotenz verwendet: So wird anstelle von Kilobyte die Einheit kB (gesprochen K-Byte) verwendet. Dabei ist 1 kB gleich  $1.024 (= 2^{10})$  Byte, 1 MB gleich  $1.024 \text{ kB} = 1.048.576 (= 2^{20})$  Byte usw. Leider sprechen viele von Kilobyte und Megabyte, obwohl sie kB und MB meinen.

### 2.3. *Hexadezimalsystem*

Hexadezimale Zahlen sind Zahlen zur Basis 16 (hexa für 6 und deka für 10). Da für die erlaubten Ziffern die zehn Ziffern vom Dezimalsystem 0 bis 9 nicht ausreichen, werden die ersten 6 Buchstaben des Alphabets noch dazugenommen: A bis F. Dabei ist A = 10, B = 11, C = 12, D = 13, E = 14 und F = 15. Anstelle der Großbuchstaben A bis F können auch die Kleinbuchstaben a bis f verwendet werden.

Um hexadezimale Zahlen von Zahlen anderer Zahlensysteme zu unterscheiden, wird entweder ein 0x bzw. 0X vor die Zahl oder ein h hinter die Zahl gesetzt.

#### **Beispiel:**

Die Zahl 0X1AF2 (alternative Darstellungen: 0x1af2, 1AF2h oder  $1AF2_{16}$ ) setzt sich wie folgt zusammen:  $1 \cdot 16^3 + 10 \cdot 16^2 + 15 \cdot 16^1 + 2 \cdot 16^0 = 6898_{10}$ .

### 2.4. *Oktalsystem*

Oktale Zahlen sind Zahlen zur Basis 8 (okto). Entsprechend sind nur die Ziffern von 0 bis 7 in diesem Zahlensystem erlaubt. Oktalzahlen wird eine 0 vorangestellt, um sie von den Dezimalzahlen zu unterscheiden.

#### **Beispiel:**

Die Oktalzahl 0171 ist (nicht nur die Vorwahl einer Mobilfunknummer sondern auch) die Dezimalzahl  $1 \cdot 8^2 + 7 \cdot 8^1 + 1 \cdot 8^0 = 121_{10}$ .

### 2.5. *Umrechnen zwischen den Zahlensystemen*

### 2.6. *Binärdarstellung von ganzen Zahlen mit und ohne Vorzeichen*

## 3. Boole'sche Algebra

### 3.1. *Einführung*

Die **Boole'sche Algebra** geht zurück auf den britischen Mathematiker George Boole (1815 - 1864). Er veröffentlichte 1847 sein Buch „The Mathematical Analysis of Logic“, in welchem er algebraische Methoden in der klassischen Logik und in der Aussagenlogik (algebraisches Logikkalkül) formulierte. Durch die Weiterentwicklung verschiedener Mathematiker ist die heutige Form der Boole'schen Algebra entstanden. So wurden z.B. erst Anfang des 20. Jahrhunderts die Symbole  $\wedge$  für das Und (AND),  $\vee$  für das Oder (OR) und  $\neg$  für die Negation (NOT) eingeführt; der Begriff Boole'sche Algebra selber wurde erst 1913 geprägt.

### 3.2. *Definition*

Die Boole'sche Algebra beinhaltet die Werte 0 und 1 sowie die zweistelligen (binären) Operatoren  $\wedge$  und  $\vee$  und den einstelligen (unären) Operator  $\neg$ . Dazu gelten folgende Axiome (Gesetze):

#### **Kommutativgesetz (Vertauschungsgesetz)**

$$A \wedge B = B \wedge A$$

$$A \vee B = B \vee A$$

#### **Assoziativgesetz (Verknüpfungsgesetz)**

$$(A \wedge B) \wedge C = A \wedge (B \wedge C)$$

$$(A \vee B) \vee C = A \vee (B \vee C)$$

#### **Idempotenzgesetz**

$$A \wedge A = A$$

$$A \vee A = A$$

#### **Distributivgesetz (Verteilungsgesetz)**

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

#### **Neutralitätsgesetz**

$$A \wedge 1 = A$$

$$A \vee 0 = A$$

#### **Extremalgesetz**

$$A \wedge 0 = 0$$

$$A \vee 1 = 1$$

#### **Gesetz der doppelten Negation**

$$\neg(\neg A) = A$$

#### **De Morgan'sches Gesetz (De Morgan'sche Regel)**

$$\neg(A \wedge B) = \neg A \vee \neg B$$

$$\neg(A \vee B) = \neg A \wedge \neg B$$

#### **Komplementärgesetz**

$$A \wedge \neg A = 0$$

$$A \vee \neg A = 1$$

#### **Dualitätsgesetz**

$$\neg 0 = 1$$

$$\neg 1 = 0$$

#### **Absorptionsgesetz**

$$A \vee (A \wedge B) = A$$

$$A \wedge (A \vee B) = A$$

Die Boole'sche Algebra wird in verschiedenen Bereichen angewendet:

In der **Aussagenlogik** wird der Wert 0 als Falsch (false) und der Wert 1 als Wahr (true) interpretiert. Die Verknüpfungen  $\wedge$ ,  $\vee$  und  $\neg$  entsprechen den logischen Verknüpfungen Und, Oder und Nicht. Logische Ausdrücke, die aus den beiden Werten und einer der Verknüpfungen bestehen, werden Boole'sche Ausdrücke genannt.

Auch bei digitalen Schaltungen kommt die Boole'sche Algebra zum Einsatz (**Schaltalgebra**); hierbei werden die Werte 0 und 1 als Spannungszustände Aus und An interpretiert. Jede digitale Schaltung kann durch einen Boole'schen Ausdruck dargestellt werden: Zwei Spannungszustände werden verknüpft und ergeben einen neuen Spannungszustand.

### 3.3. *AND (Konjunktion)*

Bei einer  $\wedge$ -Operation ist das Ergebnis nur dann 1, wenn beide Operanden gleich 1 sind. In allen anderen Fällen ist das Ergebnis gleich 0. In der Aussagenlogik bedeutet dies, dass eine mit Und verknüpfte Aussage nur dann wahr ist, wenn beide Teilaussagen wahr sind. Im allgemeinen werden diese Ergebnisse in einer sogenannten Wahrheitstabelle dargestellt.

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

### 3.4. *OR (Disjunktion)*

Bei einer  $\vee$ -Operation ist das Ergebnis dann 1, wenn mindestens einer der beiden Operanden gleich 1 ist. Nur in dem Fall, dass beide Operanden gleich 0 sind, ist auch das Ergebnis gleich 0. In der Aussagenlogik bedeutet dies, dass eine mit Oder verknüpfte Aussage dann wahr ist, wenn mindestens eine der beiden Teilaussagen wahr ist.

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

### 3.5. *NOT (Negation)*

Mit der  $\neg$ -Operation wird von einem Wert zum anderen gewechselt, d.h. von 0 zur 1 und von 1 zur 0. In der Aussagenlogik wird dabei von einer Negation (nicht zu verwechseln mit der mathematischen Negation) gesprochen, denn aus wahr wird falsch und umgedreht.

<b>A</b>	<b><math>\neg A</math></b>
0	1
1	0

### 3.6. *XOR (Exklusiv-Oder, Kontravalenz)*

Die Entweder-Oder-Verknüpfung wird auch ausschließende Disjunktion (zur Unterscheidung zur einschließenden Disjunktion, dem OR) oder Kontravalenz bzw. Antivalenz genannt. Sie gehört nicht zu den Grundoperationen, denn sie lässt sich durch die drei anderen Operatoren darstellen:

$$A \text{ xor } B = (A \vee B) \wedge \neg (A \wedge B)$$

<b>A</b>	<b>B</b>	<b>A xor B</b>
0	0	0
0	1	1
1	0	1
1	1	0

### 3.7. *Logisch oder bitweise*

## **4. Programmentwicklung**

Die Programmentwicklung verläuft durch verschiedene Phasen, von denen einige durchaus mehrfach durchlaufen werden. Diese sind:

- Problembeschreibung
- Modellbildung; dazu gehört der Entwurf von Datenstrukturen und Algorithmen
- Programmtext (auch Quelltext genannt) eingeben und bearbeiten
- Compilieren (d.h. Übersetzen) in eine Objektdatei (\*.o bzw. \*.obj)
- Linken (Zusammenbinden) der Objektdatei zu einem ausführbaren Programm (\*.exe)
- Testen des Programms
- fertiges Programm

Treten beim Compilieren, Linken oder Testen Fehler auf, muss zu einem der ersten drei Schritte zurückgegangen werden. Dies wird solange wiederholt, bis das fertige Programm fehlerfrei läuft.

## 5. Programmierstile

### 5.1. *Einführung*

Die Programmiersprache C wurde von Programmierern für Programmierer entwickelt. Sie lässt dem Benutzer der Sprache große Freiheiten in der Realisierung von Algorithmen. Konkret bedeutet dies auch, dass man C-Programme sehr klar und übersichtlich darstellen kann. Viele Programmierer sagen, dass sich ein gut formuliertes Programm bzw. eine Funktion "wie ein Buch" lesen lassen sollte. Diese Forderung lässt sich allerdings nur dann erfüllen, wenn der Programmierer sehr sorgfältig und vor allem diszipliniert arbeitet.

Bedauerlicherweise sieht die Praxis in vielen Fällen ganz anders aus. Es existieren C-Programme und -Funktionen, die bei vielen Magenkrämpfe und Augenzucken hervorrufen. Solche Programme stammen von Personen, die man gut und gerne als "Kraut und Rüben"-Programmierer bezeichnen kann. Vielleicht werden Sie jetzt sagen: "*Die Hauptsache ist, dass der Programmierer, der die Funktion geschrieben hat, weiß, was sie tut*". Da haben Sie schon - mit Einschränkungen! - recht, aber nach wenigen Wochen oder sogar schon nach Tagen kann der Programmierer meistens seinen eigenen Gedankengang zum Zeitpunkt der Programmentwicklung nicht mehr nachvollziehen. Er steht spätestens dann genau vor dem gleichen Problem, mit dem auch alle anderen zu kämpfen haben: Er muss sich neu einarbeiten und sich mit dem Programm auseinandersetzen.

Ferner ist es denkbar, dass der Verfasser eines Programmes und derjenige, der daran Änderungen vornehmen soll, nicht ein und dieselbe Person ist.

Ein Großteil der Literatur geht auf dieses Thema kaum oder überhaupt nicht ein. Dabei ist ein vernünftiger Programmierstil eine elementare Voraussetzung für professionelles Programmieren.

### 5.2. *Neun hilfreiche Regeln*

Die folgenden Regeln sind Möglichkeiten, die die Übersichtlichkeit und Lesbarkeit eines C-Programms steigern.

#### **Regel 1:**

Gehen Sie nicht zu sparsam mit dem Einfügen von Leerzeilen um. Beispielsweise können Sie in einer Funktion den Vereinbarungsteil (Deklaration von Variablen usw.) vom Anweisungsteil auch optisch mittels einer oder mehrerer Leerzeilen voneinander trennen.

#### **Regel 2:**

Rücken Sie logisch zusammengehörige Einheiten um einige Positionen ein (gewöhnlicherweise um 3 oder 4 Leerzeichen; bei vielen Editoren kann auch der Tabulator für die Einrückung verwendet werden). Somit können Sie die Verschachtlungstiefe von Anweisungsblöcken auch optisch darstellen.

#### **Regel 3:**

Fügen Sie innerhalb von Anweisungen an entsprechenden Stellen Leerzeichen ein. Das bietet sich besonders bei Ausdrücken an - z.B. zwischen den Variablen, den Konstanten und den Operatoren. So ist die Schreibweise

```
ergebnis=a*x*x+b*x+c;
```

nicht annähernd so übersichtlich wie

```
ergebnis = a * x * x + b * x + c;
```

#### **Regel 4:**

Wählen Sie für Variablen, Funktionen, Sprungmarken, usw. immer aussagekräftige Namen. Das bedeutet zwar in den meisten Fällen einen höheren Aufwand an Schreiarbeit, aber die Bezeichner des Ausdrucks

```
umfang_kreis = 2 * radius * pi;
```

sind selbstdokumentierend und wesentlich verständlicher als die der Anweisung

```
u = 2 * r * p;
```

### Regel 5:

"Mysteriöse" und "Nichtssagende" Zahlen und Zeichen sollten Sie durch symbolische Konstanten ersetzen. Es versteht sich von selbst, dass Sie dafür wieder aussagekräftige Bezeichner wählen sollten.

### Beispiel:

C bietet eine Reihe von Operatoren, deren Schreibweisen nicht immer leicht zu merken sind. Es steht Ihnen frei, für diese Operatoren aussagekräftige Namen zu vereinbaren und mit diesen in Ihrem Programm zu arbeiten.

```
#define AND    &&    /* logische UND-Verknuepfung          */
#define OR     ||    /* logische ODER-Verknuepfung         */
#define NOT    !     /* logische Negation                  */
#define BAND  &     /* bitweise UND-Verknuepfung         */
#define BOR   |     /* bitweise ODER-Verknuepfung        */
#define BNOT  ~     /* bitweise Negation (Bit-Komplement) */
#define BXOR ^     /* bitweise Exklusiv-ODER-Verknuepfung */
#define SHR  >>    /* Schieben der Bitfolge nach rechts  */
#define SHL  <<<    /* Schieben der Bitfolge nach links   */
#define MOD  %     /* Ganzzahliger Rest (Modulo)         */
```

### Regel 6:

Kommentieren Sie Stellen in Ihrem Programm, die etwas komplizierter sind. Arbeiten Sie nach dem Motto: *"Lieber etwas mehr Kommentar als zu wenig!"*

Zum Einen sollten Sie beim Kommentieren vor Anweisungsblöcken eine einzeilige Kommentarzeile als Überschrift schreiben. So braucht man bei der Fehlersuche oder bei Änderungen nicht den Anweisungsblock analysieren, sondern findet anhand dieser Überschriften den gesuchten Anweisungsblock schneller. Zum Anderen sollten kompliziertere Zeilen am Zeilenende kommentiert werden. Hier bietet es sich an, diese Kommentare nicht gleich hinter dem Anweisungsende, sondern immer in der gleichen Spalte zu beginnen. Somit kann direkt in dieser "Kommentarspalte" das Programm gelesen werden.

### Regel 7:

Jedes zu lösende Problem sollte zu Beginn gründlich analysiert und (soweit sinnvoll) in Unterprobleme aufgegliedert werden. Auch Unterprobleme können bzw. müssen in weitere Unterprobleme unterteilt werden. Eine C-Funktion stellt die Lösung eines Unterproblems dar. Die Funktionen, die zu einem Programm gehören, sind auf einem oder mehreren C-Modulen zu verteilen. Jede Funktion lässt sich damit einzeln auf korrekte Funktionalität prüfen. Diese Arbeitsweise erleichtert auch die Fehlersuche und -behebung.

### Regel 8:


Legen Sie zu jeder Funktion auch explizit deren Geltungsbereich und Ergebnistyp fest.

### Regel 9:

Vermeiden Sie den Einsatz von Sprunganweisungen (`goto`)!

## 5.3. Funktionsheader

Jeder selbstdefinierten Funktion sollte einen sogenannten **Funktionsheader** vorangestellt bekommen, der relevante Informationen zur Funktion enthält. Er hat dokumentierenden Charakter und soll die wesentlichen Eigenschaften der Funktion knapp und übersichtlich darstellen. Der Header ist ein Mittel zur einheitlichen Dokumentation von Programmen. Ein Header für eine C-Funktion könnte das folgende Aussehen haben. Hierbei handelt es sich um einen leeren Header, also um eine Art Rahmen, der für jede Funktion individuell ausgefüllt werden muss.

 `kap05_01.c`

```


01 /*****
02 * FUNKTION:
03 *-----
04 * BESCHREIBUNG:
05 * GELTUNGSBEREICH:
06 * PARAMETER:
07 * ERGEBNISTYP:
08 * ERGEBNISWERTE:
09 *   -> NORMALFALL:
10 *   -> FEHLERFALL:
11 *-----
12 * ERSTELLT VON:
13 *           AM:
14 * AENDERUNGEN :
15 *****/

```

Nach dem Wort FUNKTION steht der Funktionsname. In den nächsten Zeilen folgt eine kurze BESCHREIBUNG der Aufgabe, die die Funktion erledigt. Für gewöhnlich reichen dazu schon wenige Sätze bzw. Stichpunkte aus. Dies soll eine Ist-Beschreibung und keine Soll-Beschreibung sein! Der GELTUNGSBEREICH enthält Informationen darüber, ob die Funktion nur im eigenen Modul (also lokal) oder auch in anderen Modulen (global) verwendet werden kann. PARAMETER liefert Informationen über die Parameter der Funktion. Dabei werden die Namen der Parameter, deren Bedeutung sowie die Art der Übergabe (per Kopie, per Zeiger oder - nur in C++ - per Referenz) angegeben. ERGEBNISTYP legt den Datentyp des Rückgabewertes der Funktion fest. Falls die Funktion keinen Wert mittels der Anweisung return zurückliefert, muss dort void stehen. Die Angabe der ERGEBNISWERTE wird noch unterteilt in NORMALFALL und FEHLERFALL. So kann z.B. eine Funktion, die eine Datei öffnen soll, den Wert 1 im NORMALFALL (d.h. die Datei wurde geöffnet) und den Wert 0 im FEHLERFALL (d.h. die Datei wurde nicht geöffnet) zurückgeben. Gegebenenfalls muss noch die Bedeutung des Ergebnisses angegeben werden. ERSTELLT VON und AM gibt den Namen des Programmierers und das Erstellungsdatum an. Hinter AENDERUNGEN stehen alle Änderungen, die nach der ersten Erstellung der Funktion durchgeführt wurden - einschließlich Namen des Programmierers und Änderungsdatum. Dadurch entsteht im Laufe der Zeit die "Geschichte" (History) der Funktion.

Die - für viele sehr aufwändige - Gestaltung des Funktionsheaders mit den vielen Sternchen dient zum schnelleren Auffinden der Funktionen innerhalb des Programmcodes, da ein kompletter Kasten immer sofort auffällt.

Der hier vorgestellte Funktionsheader entspricht keiner Norm und kann problemlos umgestellt, erweitert oder auch anders gestaltet werden. Nun folgt noch ein Beispiel für einen vollständig ausgefüllten Funktionsheader:

 kap05\_02.c


```

01 /*****
02 * FUNKTION:           file_open
03 *-----
04 * BESCHREIBUNG:      Oeffnet die angegebene Datei im
05 *                   angegebenen Modus.
06 * GELTUNGSBEREICH:  Global
07 * PARAMETER:        char *Dateiname
08 *                   char *Modus
09 * ERGEBNISTYP:      int
10 * ERGEBNISWERTE:
11 *   -> NORMALFALL:  1 (Datei wurde geoeffnet.)
12 *   -> FEHLERFALL:  0 (Datei konnte nicht geoeffnet werden.)
13 *-----
14 * ERSTELLT VON:      G. Kempfer
15 *           AM:      16.02.2004
16 * AENDERUNGEN :     18.03.2005 GK: Anpassung des Ergebnistyps
17 *****/

```

## 5.4. Modulheader


Analog zu den Funktionen sollte auch jedes C-Programm und -Modul einen eigenen **Modulheader** erhalten. Dieser enthält modulbezogene Informationen und steht am Anfang eines Moduls. Ein möglicher Modulheader (auch hier gibt es keine Norm) könnte Platz für die folgenden Informationen beinhalten:

 *kap05\_03.c*

```
01 /*****
02  ****
03  *** PROGRAMM:
04  *** MODUL:
05  *** VERSION:
06  *** BESCHREIBUNG:
07  *** GLOBALE FKT.:
08  *** LOKALE FKT.:
09  ***-----
10  *** ERSTELLT VON:
11  *** DATUM BEGINN:
12  ***           ENDE:
13  *** AENDERUNGEN :
14  ****
15  *****/
```

Hinter PROGRAMM steht der Name bzw. die Bezeichnung des Programms, zu dem das Modul gehört. Hinter MODUL steht der Modulname. Für gewöhnlich ist er mit dem Namen der Datei identisch, in dem der Quelltext abgespeichert ist. Als nächstes kommt die Versionsnummer (VERSION) und eine kurze BESCHREIBUNG. Anschließend werden die globalen und lokalen Funktionen aufgelistet (nur die Funktionsnamen). Der Name des Programmierers bzw. der Programmierer steht hinter ERSTELLT VON. Darunter kommt das Anfangs- und Enddatum der Modulentwicklung. Zuletzt werden bei den AENDERUNGEN die Änderungen und Erweiterungen des Moduls angegeben. Dazu gehören auch wieder Name des Programmierers und Änderungsdatum. Eine Änderung im Modul hat für gewöhnlich auch eine Änderung der Versionsnummer zur Folge.

Im Folgenden wird ein Beispiel für einen ausgefüllten Modulheader gezeigt:

 *kap05\_04.c*

```
01 /*****
02  ****
03  *** PROGRAMM:      Verwaltung von Dateien
04  *** MODUL:         file.c
05  *** VERSION:       1.1
06  *** BESCHREIBUNG: stellt elementare Funktionen zum
07  ***               Arbeiten mit Dateien zur Verfuegung
08  *** GLOBALE FKT.: file_open
09  ***               file_close
10  ***               file_create
11  ***               file_delete
12  *** LOKALE FKT.:  file_exists
13  ***-----
14  *** ERSTELLT VON:  G. Kempfer
15  *** DATUM BEGINN: 16.02.2004
16  ***           ENDE: 17.02.2004
17  *** AENDERUNGEN : 18.03.2005 GK: file_open angepasst
18  ****
19  *****/
```

## 6. ANSI-Steuersequenzen

### 6.1. *Allgemeines*

Dieses Kapitel greift auf Elemente zurück, die erst später im Skript erläutert werden. Für den Einstieg sind nur die ANSI-Control-Sequenzen zur Steuerung der Cursorposition und zum Löschen des Bildschirms interessant. Der Rest dieses Kapitels kann beim ersten Lesen übersprungen werden.

Ein Computer bildet die uns vertrauten Zeichen (Ziffern, Buchstaben, Sonderzeichen) auf ein eindeutiges Bitmuster ab, mit dem er intern arbeitet. Es muss folglich Vorschriften geben, die besagen, durch welchen Wert ein bestimmtes Zeichen im Speicher repräsentiert wird. Eine solche Vorschrift - Code genannt - bildet die Zeichen eines Zeichenvorrats (z.B. den der Menschen) auf die Zeichen eines zweiten Zeichenvorrats (z.B. den des Rechners) ab.

Einer der gebräuchlichsten Codes ist der vom American National Standards Institut (ANSI) definierte American Standard Code for International Interchange (ASCII). In der Tabelle im Anhang sind alle ASCII-Zeichen mit den zugehörigen Werten ersichtlich. Dieser Code ist ein 7-Bit-Code, d.h. ASCII nutzt von den insgesamt 8 Bits eines Bytes nur 7 Bits für die interne Darstellung eines Zeichens aus. Mit 7 Bits lassen sich  $2^7 = 128$  Kombinationen darstellen, wobei jeder einzelnen Kombination genau ein Zeichen zugeordnet ist. Später wurde der erweiterte ASCII-Code, bei dem alle 8 Bits eines Bytes ( $2^8 = 256$  verschiedene Kombinationen) verwendet werden, eingeführt. Der Zeichenvorrat setzt sich aus darstellbaren Zeichen (dezimale ASCII-Werte von 32 bis 126 und von 128 bis 254) und Steuerzeichen (dezimale ASCII-Werte von 0 bis 31 und 127) zusammen. Darstellbare Zeichen erscheinen auf dem Bildschirm in Form eines Pixelmusters. Im Gegensatz dazu sind den Steuerzeichen kontrollierende und steuernde Aufgaben zugeordnet. Die meisten dieser Steuerzeichen spielen zwar in der Datenübertragung eine Rolle, haben allerdings auf die Steuerung des Bildschirms keinen Einfluss. Werden diese an den Bildschirm geschickt, so wird das zugeordnete Zeichen ausgegeben, z.B. wird der ASCII-Wert 1 an den Bildschirm geschickt, erscheint ein Smiley-Gesicht. Die folgenden Steuerzeichen führen dagegen konkrete Aktionen aus, wenn sie an den Bildschirm geschickt werden.

ASCII-Wert dez.	hex.	Bezeichnung	Aktion
7	0x07	BEL (Bell)	erzeugt ein akustisches Signal
8	0x08	BS (Backspace)	Bewegt den Cursor um eine Position nach links. Falls sich der Cursor bereits am Zeilenanfang befindet, bleibt dieses Steuerzeichen wirkungslos.
9	0x09	HT (Horizontal Tab)	Positioniert den Cursor auf die nächste Tabulatormarke oder an den rechten Bildschirmrand, falls keine weiteren Tabulatormarken vorhanden sind.
10	0x0A	LF (Linefeed)	Setzt den Cursor an den Anfang der nächsten Zeile.
13	0x0D	CR (Carriage Return)	Verschiebt den Cursor an den Anfang der aktuellen Zeile.
26	0x1A	SUB (Substitute)	Kennzeichnet im Textmodus das Ende einer Datei.

Für die dezimalen ASCII-Werte 0 und 255 wird jeweils ein Leerzeichen ausgegeben. Für das Escape-Zeichen ESC (dezimaler ASCII-Wert 27) erfolgt keinerlei Ausgabe. Es spielt allerdings eine wichtige Rolle

bei den Escape- und Control-Sequenzen. Mit ihnen können erst beispielsweise die Cursorposition gesteuert oder Bildschirmattribute gesetzt werden.

## 6.2. *Escape- und Control-Sequenzen*

Bei den **Escape- und Control-Sequenzen** handelt es sich um eine Folge von Zeichen, die alle mit dem **Escape-Zeichen ESC** beginnen. Mit diesen Zeichenfolgen ist es möglich, den Cursor gezielt zu positionieren, Bildschirmattribute zu setzen oder die Tastaturbelegung zu manipulieren usw. Diese Zeichenfolgen erscheinen nicht auf dem Bildschirm, sondern üben steuernde und kontrollierende Aufgaben aus. ANSI hat die Escape- und Control-Sequenzen erstmals in den Standards X3.41-1974 und X3.64-1979 definiert. In der Praxis werden solche Zeichenfolgen meistens (fälschlicherweise) nur als Escape-Sequenzen bezeichnet. ANSI unterscheidet jedoch zwischen Escape- und Control-Sequenzen.

In diesem Abschnitt werden die Escape- und Control-Sequenzen nur allgemein vorgestellt und im nächsten Abschnitt werden einige ANSI-Control-Sequenzen und deren Umsetzung in C vorgestellt.

### Escape-Sequenzen

Das Format einer Escape-Sequenz hat den folgenden allgemeinen Aufbau:

ESC	I...I	F
27	32-47	48-126
escape sequence introducer (1 Zeichen)	intermediate characters (0 oder mehrere Zeichen)	final character (1 Zeichen)

Es wurden jeweils die englischen Bezeichnungen beibehalten. Die Zahlen in der zweiten Zeile stellen jeweils die dezimalen ASCII-Werte der möglichen Zeichen dar. Im folgenden werden die drei Bereiche genauer beschrieben.

**escape sequenz introducer:** Hierbei handelt es sich um das ESC-Zeichen mit dem dezimalen ASCII-Wert 27, mit dem die Escape-Sequenz eingeleitet wird. Zeichen, die danach folgen, werden als Teil der Escape-Sequenz betrachtet und nicht auf dem Bildschirm ausgegeben.

**intermediate characters:** Dies sind Zeichen aus dem dezimalen ASCII-Bereich von 32 bis 47, die als Teil der Escape-Sequenz automatisch gespeichert werden.

**final character:** Dieses Zeichen aus dem dezimalen ASCII-Bereich von 48 bis 126 schließt eine Escape-Sequenz ab. Die Kombination aus intermediate characters und final character spezifiziert die Aufgabe der Sequenz. Die durch die Escape-Sequenz festgelegte Operation wird ausgeführt und die nachfolgenden Zeichen erscheinen wieder auf dem Bildschirm, soweit es sich um darstellbare Zeichen handelt.

### Beispiel:

Das folgende Beispiel zeigt die Escape-Sequenz für den Befehl Select Character Set (SCS):

```
ESC (A
```

Dabei ist ESC der escape sequence introducer, '(' das intermediate character und 'A' das final character.

**Wichtiger Hinweis:** Zwischen den drei Bereichen dürfen keine Leerzeichen stehen!

### Control-Sequenzen

Der Aufbau von Control-Sequenzen ist mit dem von Escape-Sequenzen vergleichbar.

CSI	P...P	I...I	F
27 91	48-63	32-47	64-126
control sequence	parameter characters	intermediate characters	final character

introducer (2 Zeichen)	(0 oder mehrere Zeichen)	(0 oder mehrere Zeichen)	(1 Zeichen)
---------------------------	-----------------------------	-----------------------------	-------------

Im folgenden werden die vier Bereiche genauer beschrieben.

**control sequenz introducer:** Hierbei handelt es sich um das Escape-Zeichen ESC (dezimaler ASCII-Wert: 27) gefolgt von der eckigen Klammer auf '[' (dezimaler ASCII-Wert: 91). Diese beiden Zeichen zusammen leiten die Control-Sequenz ein. Zeichen, die danach folgen, werden als Teil der Control-Sequenz betrachtet und nicht auf dem Bildschirm ausgegeben.

**parameter characters:** Es wird zwischen zwei Arten von parameter characters unterschieden: numerische und selektive Parameter. Ein numerischer Parameter ist eine Dezimalzahl und besteht aus Ziffern aus dem dezimalen ASCII-Bereich von 48 bis 57. Für den selektiven Parameter kann nur ein Zeichen aus dem dezimalen ASCII-Bereich von 58 bis 63 verwendet werden. Treten in einer Control-Sequenz mehrere Parameter auf, müssen diese mit einem Semikolon ';' voneinander getrennt werden.

**intermediate characters:** Dies sind Zeichen aus dem dezimalen ASCII-Bereich von 32 bis 47, die als Teil der Control-Sequenz automatisch gespeichert werden.

**final character:** Dieses Zeichen aus dem dezimalen ASCII-Bereich von 64 bis 126 schließt eine Control-Sequenz ab. Zusätzlich macht dieses Zeichen eine Aussage darüber, ob es sich um eine ANSI-Standard-Control-Sequenz oder um eine sogenannte private Control-Sequenz – auf die nicht weiter eingegangen werden – handelt.

### Beispiel:

Das folgende Beispiel zeigt die Control-Sequenz für den Befehl Cursor Position (CUP; Näheres dazu im nächsten Abschnitt!):

```
ESC[4;10H
```

Dabei ist ESC[ der control sequence introducer, '4;10' die parameter characters (2 Parameter mit Semikolon getrennt) und 'H' das final character. In dieser Control-Sequenz sind keine intermediate characters enthalten.

**Wichtiger Hinweis:** Zwischen den Bereichen dürfen auch hier keine Leerzeichen stehen!

## 6.3. ANSI-Control-Sequenzen

Für die nachfolgend beschriebenen **ANSI-Control-Sequenzen** findet man häufig die Bezeichnung ANSI-Escape-Sequenzen, obwohl dies nicht korrekt ist. Ferner werden in diesem Abschnitt nur eine Teilmenge der von ANSI definierten Control-Sequenzen vorgestellt. Damit die ANSI-Control-Sequenzen funktionieren, muss unter DOS der Treiber ANSI.SYS installiert sein. In dem DOS-Fenster unter Windows sowie in der Konsole unter Linux sollte dies ohne weiteres funktionieren.

Zu jeder Control-Sequenz existiert eine von ANSI festgelegte mnemotechnische Abkürzung, die hier auch jeweils angegeben wird. Für die Realisierung unter C sind diese aber nicht relevant. Zu beachten ist, dass die Zeichen einer Sequenz in genau der gleichen Art (Groß- und Kleinschreibung) angegeben werden müssen. Ferner dürfen keine Leerzeichen innerhalb einer Sequenz stehen.

Die Sequenzen in diesem Abschnitt werden unterteilt in die Bereiche Cursor-Steuerfunktionen, Löschraktionen, Grafikmodusfunktionen und Funktionen zur Änderung der Tastaturbelegung.

### Cursor-Steuerfunktionen

Mit Hilfe der folgenden ANSI-Control-Sequenzen kann der Cursor auf dem Bildschirm positioniert werden. In DOS-Fenstern und Konsolen, in denen gescrollt werden kann, beziehen sie sich immer auf den aktuell sichtbaren Bereich.

#### 1. Cursor Position

Sequenz: ESC[P;PH Mnemonik: CUP (Cursor Position)

oder

Sequenz: ESC[P;Pf Mnemonik: HVP (Horizontal and Vertical Position)

Die Control-Sequenzen CUP und HVP positionieren den Cursor auf eine beliebige Stelle auf dem Bildschirm. Der erste Parameter P legt die Zeile und der zweite die Spalte fest. Der Standardwert für fehlende Parameter ist 1, d.h. ESC[H ist identisch mit ESC[1;1H.

Die folgende Funktion zeigt, wie die beschriebene Control-Sequenz in C realisiert werden kann. Hierzu muss die Control-Sequenz an den Bildschirm geschickt werden. Dies erfolgt mit Hilfe der Funktion printf. Das Escape-Zeichen ESC wird dabei in oktaler Schreibweise ('\033') angegeben.

```
void position(int Zeile, int Spalte)
{ printf("\033[%d;%dH", Zeile, Spalte);
}
```

Der Funktionsaufruf position(4,50); bewegt den Cursor auf die fünfzigste Spalte in der vierten Zeile. Funktionsaufrufe erfolgen allerdings erst zur Laufzeit des Programms und verzögern somit seine Ausführung. Makros hingegen werden schon während der Übersetzung vom Präprozessor (siehe Kapitel *Präprozessorbefehle*) aufgelöst. Das heißt, dass Makros für gewöhnlich schneller abgearbeitet werden als Funktionen. Die Ausgabe der oben erläuterten Control-Sequenz als Makro lautet wie folgt.

```
#define POSITION(Ze, Sp) printf("\033[%d;%dH", Ze, Sp)
```

Zusätzlich kann noch folgendes Makro definiert werden.

```
#define HOME printf("\033[H")
```

Innerhalb des Programms könnten die Makroaufrufe wie folgt aussehen.

```
POSITION(23, 12);
HOME;
```

Es sei hier darauf hingewiesen, dass zwischen dem Makronamen und der linken öffnenden runden Klammer **kein** Leerzeichen stehen darf.

## 2. Cursor Up

Sequenz: ESC[PA Mnemonik: CUU (Cursor Up)

Diese Control-Sequenz bewegt den Cursor innerhalb der aktuellen Spalte um P Zeilen nach oben. Falls sich der Cursor bereits in der ersten Zeile des Bildschirms befindet, bleibt die Sequenz wirkungslos. Fehlt der Parameter P, so wird als Standardwert 1 angenommen.

```
#define UP(Anz) printf("\033[%dA", Anz)
#define UP_LINE printf("\033[A")
```

## 3. Cursor Down

Sequenz: ESC[PB Mnemonik: CUD (Cursor Down)

Diese Control-Sequenz bewegt den Cursor innerhalb der aktuellen Spalte um P Zeilen nach unten. Falls sich der Cursor bereits in der letzten Zeile des Bildschirms befindet, bleibt die Sequenz wirkungslos. Fehlt der Parameter P, so wird als Standardwert 1 angenommen.

```
#define DOWN(Anz) printf("\033[%dB", Anz)
#define DOWN_LINE printf("\033[B")
```

## 4. Cursor Forward

Sequenz: ESC[PC Mnemonik: CUF (Cursor Forward)

Diese Control-Sequenz bewegt den Cursor innerhalb der aktuellen Zeile um P Spalten nach rechts. Falls sich der Cursor bereits in der letzten Spalte der Zeile befindet, bleibt die Sequenz wirkungslos. Fehlt der Parameter P, so wird als Standardwert 1 angenommen.

```
#define RIGHT(Anz) printf("\033[%dC", Anz)
#define ONE_POS_RIGHT printf("\033[C")
```

## 5. Cursor Backward

Sequenz: ESC[PD Mnemonik: CUB (Cursor Backward)

Diese Control-Sequenz bewegt den Cursor innerhalb der aktuellen Zeile um P Spalten nach links. Falls sich der Cursor bereits in der ersten Spalte der Zeile befindet, bleibt die Sequenz wirkungslos. Fehlt der Parameter P, so wird als Standardwert 1 angenommen.

```
#define LEFT(Anz)          printf("\033[%dD", Anz)
#define ONE_POS_LEFT      printf("\033[D")
```

## 6. Save Cursor Position

Sequenz: ESC[s Mnemonik: SCP (Save Cursor Position)

Die Position des Cursors (Zeile und Spalte) zum Zeitpunkt der Ausführung der Control-Sequenz wird intern gespeichert. Mit Hilfe der RCP-Sequenz kann der Cursor auf die gespeicherte (alte) Position zurückgebracht werden.

```
#define STORE_POS          printf("\033[s")
```

## 7. Restore Cursor Position

Sequenz: ESC[u Mnemonik: RCP (Restore Cursor Position)

Der Cursor wird auf die zuvor mittels der SCP-Sequenz gespeicherten Position (Zeile und Spalte) gesetzt.

```
#define RESTORE_POS        printf("\033[u")
```

## 8. Device Status Report

Sequenz: ESC[6n Mnemonik: DSR (Device Status Report)

Die Control-Sequenz DSR fordert die aktuelle Position des Cursors an. Diese wird mit Hilfe der Control-Sequenz CPR in der Form ESC[P;PR in den Tastaturpuffer geschrieben. Von dort kann die Position des Cursors ausgelesen werden. Die erste Parameter P gibt die Zeile und der zweite die Spalte an. In dem anschließenden Beispiel wird gezeigt, wie die Control-Sequenzen DSR und CPR angewendet werden.


```
#define ACT_POS            printf("\033[6n")
```

## 9. Cursor Position Report

Sequenz: ESC[P;PR Mnemonik: CPR (Cursor Position Report)

Die Control-Sequenz CPR ist die Antwort auf die Control-Sequenz DSR und wird in der Standardeingabe (normalerweise im Tastaturpuffer) abgelegt. Dabei gibt der erste Parameter P die Zeile und der zweite die Spalte des Cursors an. Im folgenden Beispiel wird gezeigt, wie die Control-Sequenzen DSR und CPR angewendet werden.

In der folgenden Beispiel-Funktion `Cursor_Position` wird die aktuelle Cursorposition mit Hilfe des Makros `ACT_POS` (siehe oben) angefordert. Die Antwort wird aus dem Tastaturpuffer gelesen und an die aufrufende Funktion zurückgegeben. Die Funktion gibt zusätzlich einen Statuswert zurück, der angibt, ob ein Fehler aufgetreten ist oder nicht.

 *kap06\_01.c*

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 #define FEHLER          0
05 #define OK              1
06
07 #define ESC             '\033'
08 #define LEFT_BRACKET   '['
09 #define NUL             '\0'
10 #define SEMIKOLON      ';'
11 #define FINAL_CHAR     'R'
```

```

12
13 #define ACT_POS          printf("\033[6n")
14
15 int Cursor_Position(int *Zeile, int *Spalte)
16 {
17     int Zeichen, i;
18     char Parameter1[5], Parameter2[5];
19
20     /* Control-Sequenz DSR: akt. Cursor-Position anfordern */
21     ACT_POS;
22
23     /* erstes Zeichen der CPR-Sequenz lesen (= ESC) */
24     if ((Zeichen = getch()) != ESC)
25         return FEHLER;
26
27     /* zweites Zeichen der CPR-Sequenz lesen (='[') */
28     if ((Zeichen = getch()) != LEFT_BRACKET)
29         return FEHLER;
30
31     /* ersten Parameter holen */
32     for (i = 0; isdigit(Zeichen = getch()); i++)
33         Parameter1[i] = (char) Zeichen;
34     Parameter1[++i] = NUL;
35
36     /* pruefen, ob Semikolon zwischen den Parametern steht */
37     if (Zeichen != SEMIKOLON)
38         return FEHLER;
39
40     /* zweiten Parameter holen */
41     for (i = 0; isdigit(Zeichen = getch()); i++)
42         Parameter2[i] = (char) Zeichen;
43     Parameter2[++i] = NUL;
44
45     /* pruefen, ob letztes Zeichen gleich 'R' */
46     if (Zeichen != FINAL_CHAR)
47         return FEHLER;
48
49     /* CPR-Sequenz ist ok -> Parameter uebergeben */
50     *Zeile = atoi(Parameter1);
51     *Spalte = atoi(Parameter2);
52     return OK;
53 }
54
55 int main(void)
56 {
57     int Zeile, Spalte, Sequenz_OK;
58
59     /* akt. Cursorposition holen */
60     Sequenz_OK = Cursor_Position(&Zeile, &Spalte);
61
62     /* Cursorposition ausgeben */
63     if (Sequenz_OK)
64     {
65         printf("Akt. Cursorposition:\n");
66         printf("Zeile:  %d\n", Zeile);
67         printf("Spalte: %d\n", Spalte);
68     }
69     else

```

```

70     printf("Akt. Position konnte nicht ermittelt werden!\n");
71
72     return 0;
73}

```

## Löschfunktionen

Die nachfolgend beschriebenen ANSI-Control-Sequenzen löschen bestimmte Bereiche des Bildschirms.

### 1. Erase Display

Sequenz: ESC[2J    Mnemonik: ED (Erase Display)

Diese Sequenz hat das Löschen des kompletten (sichtbaren) Bildschirms zur Folge.

```
#define CLEAR           printf("\033[2J")
```

### 2. Erase Line

Sequenz: ESC[K    Mnemonik: EL (Erase Line)

Die Zeile, in der sich der Cursor aktuell befindet, wird von einschließlich der Cursorposition bis zum Zeilenende gelöscht.

```
#define CLEAR_LINE     printf("\033[K")
```

## Grafikmodusfunktionen

Mit Hilfe der folgenden ANSI-Control-Sequenzen kann die Bildschirmdarstellung verändert werden. Da einige dieser Sequenzen für DOS definiert wurden, funktionieren unter Umständen nicht alle Sequenzen auch in DOS-Fenstern bzw. in Konsolen.

### 1. Set Graphics Rendition

Sequenz: ESC[P;...;Pm    Mnemonik: SGR (Set Graphics Rendition)

Die SGR-Sequenz gestattet das Einstellen spezieller grafischer Darstellungen. Die Art der Funktion wird durch den Wert des Parameters P bestimmt. Die folgende Liste zeigt mögliche Werte, die der Parameter P annehmen kann.

Parameter P	Wirkung
0	Alle Attribute werden zurückgesetzt
1	Verstärkte Intensität (Fett) einschalten
4	Unterstreichen einschalten (nur Monochrom-Bildschirme)
5	Blinken einschalten
7	Inversdarstellung einschalten
8	Unsichtbare (versteckte) Ausgabe
30	Vordergrundfarbe schwarz
31	Vordergrundfarbe rot
32	Vordergrundfarbe grün
33	Vordergrundfarbe gelb
34	Vordergrundfarbe blau
35	Vordergrundfarbe violett
36	Vordergrundfarbe kobaltblau
37	Vordergrundfarbe weiß
40	Hintergrundfarbe schwarz

41	Hintergrundfarbe rot
42	Hintergrundfarbe grün
43	Hintergrundfarbe gelb
44	Hintergrundfarbe blau
45	Hintergrundfarbe violett
46	Hintergrundfarbe kobaltblau
47	Hintergrundfarbe weiß

Hier nur einige Beispiele für die Umsetzung in die entsprechenden C-Makros:

```
#define BOLD          printf("\033[1m")
#define UNDERSCORE   printf("\033[4m")
#define BLINK         printf("\033[5m")
#define INVERSE       printf("\033[7m")
#define BOLD_INVERSE printf("\033[1;7m")
#define RED_ON_WHITE  printf("\033[31;47m")
#define ATTRIBUTE_OFF printf("\033[0m")
```

Der Nachteil der letzten Sequenz besteht darin, dass damit alle gerade aktiven Attribute zurückgesetzt werden. Hier müssten parallel noch Informationen über die gerade gesetzten Attribute verwaltet werden (z.B. in einem Bitfeld). Möchte man nun ein Attribute zurücksetzen, so muss man dazu mit dem Makroaufruf `ATTRIBUTE_OFF` alle zurückgesetzt werden und anschließend die restlichen nacheinander wieder explizit aktivieren.

## 2. Set Mode

Sequenz: `ESC[=Ph` Mnemonik: `SM` (Set Mode)

Sowohl die Bildschirmbreite als auch der `-typ` können mit der `SM`-Sequenz eingestellt werden. Der Parameter `P` kann folgende Werte (für `CGA`-Bildschirme; für `VGA`-Bildschirme usw. gibt es weitere Werte) annehmen:

Parameter P	Wirkung
0	Textbildschirm: 25 Zeilen * 40 Spalten (schwarz/weiß).
1	Textbildschirm: 25 Zeilen * 40 Spalten (Farbe).
2	Textbildschirm: 25 Zeilen * 80 Spalten (schwarz/weiß).
3	Textbildschirm: 25 Zeilen * 80 Spalten (Farbe).
4	Grafikbildschirm: 320 * 200 Punkte (Farbe).
5	Grafikbildschirm: 320 * 200 Punkte (schwarz/weiß).
6	Grafikbildschirm: 640 * 200 Punkte (schwarz/weiß).
7	Wrap-Modus einschalten. Sobald das Ende einer Zeile erreicht ist, erfolgt automatisch ein Zeilenvorschub. Das bedeutet, dass Zeichen, die über die aktuelle Zeile hinausgehen, in die nächste Zeile übernommen werden.

Hier wieder nur einige Beispiele für die Umsetzung in die entsprechenden C-Makros:

```
#define TEXT_COLOR_25_80 printf("\033[=3h")
#define WRAP_MOD_ON      printf("\033[=7h")
```

## 3. Reset Mode

Sequenz: `ESC[=Pl` Mnemonik: `RM` (Reset Mode)

Die `RM`-Sequenz setzt die mit der `SM`-Sequenz gesetzten Attribute zurück. Die Parameterwerte für den Parameter `P` sind die gleichen wie bei der zuvor beschriebenen `SM`-Sequenz.

```
#define WRAP_MOD_OFF          printf("\033[=71")
```

## Änderung der Tastaturbelegung

Folgende Control-Sequenz erlaubt eine Veränderung der Tastaturbelegung:

Sequenz: ESC[#;...;#p

Zu beachten ist, dass das Symbol # sowohl einen numerischen Parameter (also eine Dezimalzahl) als auch eine Zeichenfolge (von Anführungszeichen eingeschlossen) repräsentieren kann. Diese Sequenz ist in keiner ISO- oder ANSI-Norm zu finden, entspricht jedoch der Vorschrift zur Bildung einer ANSI-Control-Sequenz. Das abschließende Zeichen p spezifiziert diese Sequenz als eine private ANSI-Control-Sequenz.

Der erste Parameter # ist der Tastencode als dezimaler Wert. Handelt es sich hierbei um eine Funktions- (F1 bis F12) oder Sondertaste (z.B. Cursor- bzw. Pfeiltasten, Ins (Einf), Del (Entf), ...), die durch zwei ASCII-Zeichen gekennzeichnet ist, ist der erste Parameter eine 0 und als nächster Parameter folgt der Tastencode. Alle weiteren Parameter (Dezimalzahlen und/oder Zeichenketten) der Sequenz stellen die Zeichenfolge dar, die nach dem Drücken der durch den ersten bzw. den beiden ersten Parametern beschriebenen Taste erzeugt werden sollen. Hierzu nun einige Beispiele:

Die folgenden Sequenzen verändern die Belegung der Tasten 'z', 'Z', 'y' und 'Y'. Das bedeutet, dass nach dem Drücken der Taste 'z' bzw. 'z' das Zeichen 'Y' bzw. 'y' erscheint (und umgekehrt).

```
printf("\033[122;121p")    /* 'z' wird zu 'y' */
printf("\033[90;89p")     /* 'Z' wird zu 'Y' */
printf("\033[121;122p")   /* 'y' wird zu 'z' */
printf("\033[89;90p")     /* 'Y' wird zu 'Z' */
```

Die nächste Sequenz legt auf die Funktionstaste F8 den DOS-Befehl dir, abgeschlossen mit einem RETURN (ASCII-Wert 13), damit die sofortige Ausführung des Befehls veranlasst wird. Der Tastencode für die Funktionstaste F8 lautet 0;66.

```
printf("\033[0;66;"dir";13p")
```

Hier werden noch einmal alle Escape-Sequenzen zusammengestellt und als Headerdatei aufgelistet:

### *escapesequenzen.h*

```
01 #ifndef escapesequenzen_h
02     #define escapesequenzen_h escapesequenzen_h
03
04     #define POSITION(Ze, Sp)      printf("\033[%d;%dH", Ze, Sp)
05     #define HOME                printf("\033[H")
06     #define UP(Anz)             printf("\033[%dA", Anz)
07     #define UP_LINE             printf("\033[A")
08     #define DOWN(Anz)           printf("\033[%dB", Anz)
09     #define DOWN_LINE           printf("\033[B")
10     #define RIGHT(Anz)          printf("\033[%dC", Anz)
11     #define ONE_POS_RIGHT       printf("\033[C")
12     #define LEFT(Anz)           printf("\033[%dD", Anz)
13     #define ONE_POS_LEFT        printf("\033[D")
14
15     #define STORE_POS            printf("\033[s")
16     #define RESTORE_POS         printf("\033[u")
17     #define ACT_POS             printf("\033[6n")
18
19     #define CLEAR                printf("\033[2J")
20     #define CLEAR_LINE          printf("\033[K")
21
22     #define ATTRIBUTE_OFF        printf("\033[0m")
23     #define BOLD                 printf("\033[1m")
24     #define UNDERSCORE           printf("\033[4m")
```

```

25 #define BLINK printf("\033[5m")
26 #define INVERSE printf("\033[7m")
27 #define INVISIBLE printf("\033[8m")
28
29 #define FORECOLOR_BLACK printf("\033[30m")
30 #define FORECOLOR_RED printf("\033[31m")
31 #define FORECOLOR_GREEN printf("\033[32m")
32 #define FORECOLOR_YELLOW printf("\033[33m")
33 #define FORECOLOR_BLUE printf("\033[34m")
34 #define FORECOLOR_VIOLETT printf("\033[35m")
35 #define FORECOLOR_KOBALT printf("\033[36m")
36 #define FORECOLOR_WHITE printf("\033[37m")
37 #define BACKCOLOR_BLACK printf("\033[40m")
38 #define BACKCOLOR_RED printf("\033[41m")
39 #define BACKCOLOR_GREEN printf("\033[42m")
40 #define BACKCOLOR_YELLOW printf("\033[43m")
41 #define BACKCOLOR_BLUE printf("\033[44m")
42 #define BACKCOLOR_VIOLETT printf("\033[45m")
43 #define BACKCOLOR_KOBALT printf("\033[46m")
44 #define BACKCOLOR_WHITE printf("\033[47m")
45
46 #define TEXT_BW_25_40 printf("\033[=0h")
47 #define TEXT_COLOR_25_40 printf("\033[=1h")
48 #define TEXT_BW_25_80 printf("\033[=2h")
49 #define TEXT_COLOR_25_80 printf("\033[=3h")
50 #define GRAFIC_COLOR_320_200 printf("\033[=4h")
51 #define GRAFIC_BW_320_200 printf("\033[=5h")
52 #define GRAFIC_BW_640_200 printf("\033[=6h")
53 #define WRAP_MODE_ON printf("\033[=7h")
54 #define WRAP_MODE_OFF printf("\033[=7l")
55 #endif

```

## **7. Automaten**

**7.1. *Einführung***

**7.2. *Touring-Maschine***

**7.3. *Halteproblem***

**7.4. *Entscheidungsproblem***

**7.5. *Berechenbarkeit***

# 8. Komplexität

## 8.1. *Einführung*

Wenn die Laufzeiten (und damit auch die Effizienz) von Algorithmen verglichen werden, reicht es nicht aus zu sagen, dass ein Algorithmus schneller ist als ein anderer. So wird z.B. das Bubblesort-Verfahren eine Datenmenge von 10 Datensätzen auf einem sehr schnellen Computer schneller sortiert haben als das Quicksort-Verfahren eine Datenmenge von 1 Million Datensätzen auf einem langsamen Computer. D.h. die tatsächliche Laufzeit hängt von mehr Faktoren als nur vom Algorithmus und der Taktfrequenz des Computers ab.

Um nun Algorithmen besser vergleichen zu können, wurde der Begriff der **Komplexität** eingeführt. Die Komplexität eines Algorithmus ist die Abschätzung des Aufwandes seiner Realisierung auf einem Computer. Dabei bedeutet die Abschätzung des Aufwandes, wie sich der Aufwand verändert, wenn sich die Menge der Daten verdoppelt. Der Laufzeitaufwand z.B. wird durch Zählen von der Anzahl von Durchläufen (meistens Schleifendurchläufen) ermittelt. Dabei werden Operationen, die nicht von der Anzahl der Datenmenge abhängen, nicht mitgezählt, da sich diese bei Verdopplung der Datenmenge nicht ändern.

## 8.2. *Arten der Komplexität*

Bei der Komplexität wird im wesentlichen zwischen drei verschiedenen Arten unterschieden:

Die **Laufzeit-Komplexität** ist die Abschätzung der Computerrechenzeit (Laufzeit des Programms) und ist die Komplexität, die am häufigsten angegeben wird. So wird auch hier im folgenden die Laufzeit-Komplexität betrachtet.

Die **Raum-Komplexität** bezieht sich auf die Abschätzung des Speicherbedarfs der Daten bzw. der Informationen, die für einen Algorithmus benötigt wird.

Die **psychologische Komplexität** spiegelt das Programmverständnis des Programmierers und die Handhabbarkeit eines Algorithmuses wieder. Gerade die psychologische Komplexität wird häufig deutlich unterschätzt. Sie wird gering gehalten, wenn der Programmierer sich an die Vorgaben der Programmierstile hält. Dazu gehören u.a.

- Strukturierung des Quelltextes durch Einrückung und Leerzeilen
- Kommentierung des Quelltextes
- Unterteilung des Programmes in logische Einheiten (Module) und Funktionen

Einige dieser Aspekte wurden im Kapitel 5: *Programmierstile* bereits angesprochen.

## 8.3. *O-Notation*

Die Komplexität wird durch die **O-Notation** ausgedrückt. Diese wurde von dem deutschen Mathematiker Paul Bachmann (\* 1837 † 1920) im Jahr 1894 im Rahmen seines Buches über die analytische Zahlentheorie eingeführt und später von dem ebenfalls deutschen Mathematiker Edmund Landau (\* 1877 † 1938) weiter entwickelt. Dabei gibt die O-Notation den schlechtesten Fall (*worst case*) bei einem Algorithmus an; z.B. wenn bei einem unsortierten Array nach einem Eintrag gesucht wird und der letzte Eintrag im Array ist das gesuchte Element. Durch die Angabe des schlechtesten Falles wird eine obere Grenze angegeben, die von dem Algorithmus nicht überschritten werden kann.

Daneben gibt es noch den besten Fall (*best case*), der aber wegen seiner Seltenheit und seiner geringen Aussagekraft nicht weiter betrachtet wird, und den durchschnittlichen Fall (*average case*), für den die  $\Omega$ -Notation eingeführt wurde.

Im allgemeinen wird zwischen den folgenden O-Notationen unterschieden (kein Anspruch auf Vollständigkeit!). Dazu wird jeweils ein Anwendungsbeispiel angegeben, das im allgemeinen die angegebene Komplexität hat.

O-Notation	Aufwand	Anwendungsbeispiel
$O(1)$	konstanter Aufwand	Operationen, die nicht von der Datenmenge abhängen
$O(n)$	linearer Aufwand	lineare (sequentielle) Suchverfahren
$O(\log n)$	logarithmischer Aufwand	binäre Suchverfahren
$O(n * \log n)$	quasilinearer Aufwand	schnelle Sortierverfahren
$O(n^2)$	quadratischer Aufwand	langsame Sortierverfahren, Vektormultiplikation
$O(n^3)$	kubischer Aufwand	Matrizenmultiplikation
$O(n^k)$	polynomialer Aufwand	$(k-1)$ -dimensionale Matrizenmultiplikation
$O(2^n)$	exponentieller Aufwand	entscheidungsbasierte Spiele (z.B. Schach)
$O(n!)$	fakultätischer Aufwand	
$O(n^n)$	superexponentieller Aufwand	

Die folgende Tabelle zeigt recht eindrücklich, wie sich die einzelnen Komplexitäten schon für kleine Datenmengen  $n$  entwickeln.

$n$	$\log n$	$n * \log n$	$n^2$	$n^3$	$2^n$	$n!$	$n^n$
2	1	2	4	8	4	2	4
4	2	8	16	64	16	24	256
8	3	24	64	512	256	40.320	16.777.216
16	4	64	256	4.096	65.536	20.922.789.888.000	18.446.744.073.709.551.616
32	5	160	1.024	32.768	4.294.967.296	$\sim 2,6 * 10^{35}$	$\sim 1,5 * 10^{48}$

#### 8.4. Rechenregeln für Komplexitäten

Um die Komplexitäten von mehrschrittigen Funktionen schneller und einfacher abschätzen zu können, wurden zur O-Notation auch Rechenregeln entwickelt. Einige davon werden im folgenden vorgestellt. Dabei wird eine unbekannte oder allgemeine Komplexität mit  $O(f(n))$  bezeichnet.

- $O(1) = c$ , d.h. der konstante Aufwand wird mit der Konstanten  $c$  bezeichnet.
- $c * O(f(n)) = O(c * f(n)) = O(f(n))$ , da bei der Betrachtung der Verdopplung der Datenmenge  $n$  die Konstante nicht ins Gewicht fällt und daher vernachlässigt werden kann.
- $O(f(n)) + O(f(n)) = 2 * O(f(n)) = O(f(n))$  (siehe vorige Regel)
- Bei  $f(n) = c_k * n^k + c_{k-1} * n^{k-1} + \dots + c_0$  gilt  $O(f(n)) = O(n^k)$ , d.h. es wird für die Komplexität immer nur die höchste Potenz berücksichtigt, die niedrigeren Potenzen werden vernachlässigt.
- $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$ , auch hier wird nur die höhere Komplexität berücksichtigt.
- $O(f(n)) * O(g(n)) = O(f(n) * g(n))$

#### 8.5. Laufzeit-Komplexitäten der Grundbausteine einer Programmiersprache

Um die Laufzeit-Komplexität abschätzen zu können, müssen die Komplexitäten der grundlegenden Elemente bekannt sein.

## Einfache Anweisungen

Einfache Anweisungen (Zuweisungen, Berechnungen, Vergleiche, Deklarationen und Definitionen, Ein- und Ausgabe, usw.) sind nicht von der Menge der Daten abhängig. Sie haben daher einen konstanten Aufwand bzw. die Komplexität  $O(1)$ .

## Sequenz

Eine Sequenz ist eine Aneinanderreihung von einfachen Anweisungen oder von mehreren Algorithmen, die von der Datenmenge  $n$  abhängen. Die Summe der Komplexitäten ist nach den Rechenregeln vom vorigen Abschnitt  $O(f(n)) + O(g(n)) + O(h(n)) = O(\max\{f(n), g(n), h(n)\})$ . D.h. es wird in einer Sequenz nur der Algorithmus mit der größten Komplexität betrachtet.

## Schleife

Die Komplexität einer Schleife, die  $n$ -mal durchlaufen wird, ist im allgemeinen gleich  $O(n)$ . Nun kann der Schleifenkörper selbst von der Datenmenge  $n$  abhängig sein, z.B. durch eine weitere innere Schleife, die ebenfalls  $n$ -mal durchlaufen wird. Dann gilt  $O(n) * O(n) = O(n^2)$ . Verallgemeinert kann also gesagt werden, dass durch eine Schleife die Komplexität des Schleifenkörpers mit  $n$  multipliziert wird:  $O(n) * O(f(n)) = O(n * f(n))$ .

## Verzweigung

Bei einer Verzweigung wird nur ein Zweig der Verzweigung abgearbeitet. Da mit der O-Notation immer der schlimmste Fall betrachtet wird, ist die Gesamt-Komplexität der Verzweigung gleich der Komplexität von dem Zweig, der die höhere Komplexität hat, d.h. gleich  $O(\max\{f(n), g(n)\})$ .

## 8.6. Grenzen der O-Notation

Obwohl die O-Notation ein mathematisches Werkzeug ist, bedarf es des Mitdenkens und der Interpretation. Folgendes sollte bei der Anwendung der Komplexitäten immer berücksichtigt werden:

1. Die O-Notation gibt den Aufwand im schlimmsten Fall (*worst case*) an. Damit ist sie eine obere Grenze, was nicht heißen muss, dass der Algorithmus jemals die abgeschätzte Komplexität erreichen wird. Vielmehr ist es die Angabe, dass der Algorithmus niemals mehr Zeit oder mehr Speicher benötigen wird. Daher macht es durchaus Sinn, sich auch den durchschnittlichen und auch den besten Fall anzusehen.
2. Die oben verwendete Konstante, die die Komplexität  $O(1)$  (konstanter Aufwand) hat, wurde bisher immer vernachlässigt, da sie unabhängig von der Datenmenge ist. In dieser Konstanten ist die tatsächliche Laufzeit (oder der tatsächlich benötigten Speicherbedarf) enthalten. Es sollte natürlich klar sein, dass ein Algorithmus mit der Komplexität  $O(n^3)$ , bei dem für jeden Durchlauf nur einen Bruchteil einer Sekunde benötigt wird, einem Algorithmus mit der Komplexität  $O(n)$ , der für jeden Durchlauf ein Jahr benötigt, vorzuziehen ist.

### Beispiel:

Ein Algorithmus mit der Komplexität  $O(n^3)$  benötigt pro Durchlauf 1 Mikrosekunde. Bei einer Datenmenge von  $n = 100.000$  ergibt sich dann eine Laufzeit von knapp 32 Jahren. Dagegen benötigt ein Algorithmus mit der Komplexität  $O(n)$  und einer Laufzeit von 1 Jahr pro Durchlauf bei der gleichen Datenmenge insgesamt 1.000.000 Jahre!

Daher sollte auch immer ein Blick auf den konstanten Aufwand geworfen werden.

3. Auch bei der Vernachlässigung der Komplexitäten mit geringeren Potenzen (siehe Rechenregeln; es wird bei einer Sequenz immer nur die höchste Komplexität berücksichtigt) kann es zu falschen Ergebnissen kommen, wenn deren Faktoren deutlich größer sind als der Faktor der höchsten Komplexität – gerade bei kleinen Datenmengen wirkt sich dies sehr stark aus.
4. Denn die O-Notation ist eine Aufwandsabschätzung für sehr große Datenmenge – eigentlich sogar der Grenzwert für Datenmengen gegen Unendlich. Daher sollte auch immer die tatsächliche Datenmenge bei den Betrachtungen berücksichtigt werden.

# 9. Sortier-Algorithmen

Das Motto „Ordnung ist das halbe Leben“ hat, wenn nicht im Leben, zumindest beim Programmieren seine Berechtigung. Sortieren von Datensätzen nach verschiedenen Kriterien oder zum Ordnen von Tabellen oder für schnellen Zugriff über binäres Suchen gehören zum täglichen Brot.

Zwar werden von den meisten C-Compilern Standard-Algorithmen in Form von Bibliotheks-Funktionen mitgeliefert (zum Beispiel „QuickSort“); dennoch ist es wichtig, einen groben Überblick über die Problematik des Sortierens zu haben, damit die Leistungsfähigkeit der verschiedenen Verfahren beurteilt werden kann.

Zudem gehören Sortierverfahren neben Suchverfahren mit zu den am besten und ausführlichsten untersuchten Algorithmen in der Informatik.

Bei Sortierverfahren unterscheidet man zunächst „interne“ und „externe“ Verfahren. Bei internen Verfahren kann die gesamte zu sortierende Datenmenge im Arbeitsspeicher des Rechners gehalten werden (Sortieren von Arrays). Damit ist die maximale Anzahl der zu sortierenden Elemente beschränkt. Externe Verfahren (Bandsortierverfahren) unterliegen dieser Beschränkung nicht. Die Daten werden in sequentiellen Dateien abgelegt und durch Mischen dieser Dateien und einiger während des Sortierens anzulegender Hilfsdateien sortiert. Auf die externen Verfahren wird hier nicht weiter eingegangen.

In diesem Kapitel werden einige interne Sortierverfahren vorgestellt, die außerdem noch die Eigenschaft haben, dass außer dem Daten-Array kein weiterer Speicherplatz benötigt wird. Zur besseren Übersicht werden die hier vorgestellten Sortierverfahren nach der jeweils zugrunde liegenden Idee und nach ihrer **Komplexität** (siehe voriges Kapitel) eingeteilt:

	Einfache Verfahren		Höhere Verfahren	
Einfügen	direktes Einfügen	$O(n^2)$	Shell-Sort	$O(n^{1.2})$
Auswählen	direktes Auswählen	$O(n^2)$	Heap-Sort	$O(n \cdot \log_2(n))$
Austauschen	Bubble-Sort	$O(n^2)$	Quick-Sort	$O(n \cdot \log_2(n))$

Bei jedem Algorithmus ist die Komplexität des Verfahrens angegeben (zum Beispiel  $O(n)$ ). Die Komplexität ist ein Maß dafür, wie die Laufzeit eines Programmes von der Größe des zu lösenden Problems abhängt.

Im Falle der Sortierverfahren bedeutet dies, dass zum Beispiel die Laufzeit von Bubble-Sort ungefähr quadratisch mit der Anzahl der zu sortierenden Elemente zunimmt. Eine gute Grundlage für die Bestimmung der Komplexität von Sortier-Algorithmen sind die Anzahl der notwendigen Vergleiche und Vertauschungen von Elementen – die beiden wesentlichen Operationen beim Sortieren.

In realen Anwendungen werden die höheren Verfahren Heap-Sort und vor allem Quick-Sort eingesetzt. Bei detaillierten Untersuchungen hat sich die Überlegenheit von Quick-Sort über alle anderen hier vorgestellten Algorithmen herausgestellt. Quick-Sort schlägt Heap-Sort in der Geschwindigkeit noch um einen Faktor 2 bis 3.

Am schlechtesten schneidet fast immer Bubble-Sort ab. Allerdings benötigen Heap- und Quick-Sort eine gewisse Mindestgröße des zu sortierenden Arrays zum Ausspielen ihrer Leistungsfähigkeit. Bei sehr kleinen Arrays ist es dagegen z.B. sinnvoll, Sortieren durch direktes Auswählen einzusetzen.

## 9.1. *Sortieren durch direktes Einfügen*

Dieser Methode liegt folgende Idee zugrunde: Das zu sortierende Array wird gedanklich in einen linken und rechten Teil getrennt. Der linke Teil ist eine bereits sortierte Sequenz, in die die Elemente des rechten Teils nacheinander einsortiert werden müssen.

Zu Beginn des Sortiervorgangs besteht der bereits sortierte linke Teil nur aus dem ersten Element des Arrays (jedes Element stellt für sich allein eine sortierte Sequenz der Länge eins dar). Alle anderen Elemente gehören zum unsortierten rechten Teil.

Solange nun der rechte Teil nicht leer ist, wird das jeweils erste Element des rechten Teils in die sortierte Sequenz eingefügt. Einfügen heißt dabei, das Element solange mit seinem linken Nachbarn zu vertauschen, bis dieser entweder kleiner ist oder der linke Rand des Arrays erreicht wird (dieses Element ist dann das kleinste bisher gefundene).

Das folgende Beispiel soll dieses Verfahren verdeutlichen. Dabei gibt der senkrechte Strich die Trennung zwischen sortierten und unsortierten Teil an.


**Beispiel:**

```

13 | 6      27      9      1
6  | 13     27      9      1
6  | 13     27 | 9      1
6  | 9      13     27 | 1
1  | 6      9      13     27 |

```

Die dazugehörige Sortierfunktion für ein zu sortierendes Zahlen-Array sieht wie folgt aus.

 *kap09\_01.c*

```

01 /*****
02 * Sortieren durch direktes Einfuegen
03 * Sortiert das angegebene Zahlen-Array in aufsteigender
04 * Reihenfolge.
05 * Parameter: Array - Zeiger auf das zu sortierende Array
06 *             Anzahl - Anzahl der Elemente im Array
07 * Rueckgabe: nichts
08 *****/
09 void InsertSort(int *Array, int Anzahl)
10 {
11     int i;        /* erstes Element im unsortierten Teil */
12     int j;        /* Laufindex zum Einsortieren */
13     int temp;     /* fuer Austausch zweier Elemente */
14
15     for (i = 1; i < Anzahl; i++)
16     {
17         j = i;
18         while (j > 0 && *(Array + j) < *(Array + j - 1))
19         {
20             /* Element an der Stelle j mit linkem Nachbarn tauschen */
21             temp = *(Array + j);
22             *(Array + j) = *(Array + j - 1);
23             *(Array + j - 1) = temp;
24             j--;
25         }
26     }
27 }

```

## 9.2. Sortieren durch direktes Auswählen


Auch bei dieser Methode unterscheidet man einen sortierten und unsortierten Teil des Arrays. Die bereits sortierte Sequenz wird beim direkten Auswählen dadurch vergrößert, dass man das kleinste Element des noch unsortierten Teils auswählt und mit einer einzigen Vertauschoperation in die sortierte Sequenz einfügt.

**Beispiel:**

		13		6		27		9		1
		1		6		27		9		13
		1		6		27		9		13
		1		6		9		27		13
		1		6		9		13		27

Der Vorteil von Sortieren durch Auswählen gegenüber Sortieren durch Einfügen liegt darin, dass pro Element zwar mehrere Vergleiche jedoch nur eine Vertauschung stattfindet. Dies spielt vor allem eine Rolle, wenn nicht ganze Zahlen sondern große Datensätze wie Adressen sortiert werden, da dann im allgemeinen die Vergleichsoperation schneller als das Vertauschen ist.

Hier die dazugehörige Funktion:

 kap09\_02.c

```

01 /*****
02  * Sortieren durch direktes Auswaehlen
03  * Sortiert das angegebene Zahlen-Array in aufsteigender
04  * Reihenfolge.
05  * Parameter: Array - Zeiger auf das zu sortierende Array
06  *           Anzahl - Anzahl der Elemente im Array
07  * Rueckgabe: nichts
08  *****/
09 void SelectSort(int *Array, int Anzahl)
10 {
11     int i;        /* erstes Element im unsortierten Teil */
12     int j;        /* Laufindex zum Suchen */
13     int min;      /* kleinstes gefundenes Element */
14     int temp;    /* fuer Austausch zweier Elemente */
15
16     for (i = 0; i < Anzahl - 1; i++)
17     {
18         min = i;
19         for (j = i + 1; j < Anzahl; j++)
20         {
21             /* suche kleinstes Element */
22             if (*(Array + j) < *(Array + min))
23                 min = j;
24         }
25         temp = *(Array + min);
26         *(Array + min) = *(Array + i);
27         *(Array + i) = temp;
28     }
29 }

```

### 9.3. *Bubble-Sort*


Das zu sortierende Array wird mehrmals von rechts nach links durchlaufen und das jeweilige Element mit seinem linken Nachbarn vertauscht, falls die beiden nicht in der richtigen Reihenfolge stehen. Die kleineren Elemente wandern also von rechts nach links. Stellt man sich das Feld senkrecht und die kleineren Elemente als 'leichtere' Blasen ('bubbles') vor, die im Feld in eine ihrem Gewicht entsprechende Stelle aufsteigen, so findet man die Erklärung für den Namen Bubble-Sort.

**Beispiel:**

		99		13		45		12		17		33		67		1
		1		99		13		45		12		17		33		67
		1		12		99		13		45		17		33		67
		1		12		13		99		17		45		33		67
		1		12		13		17		99		33		45		67

1	12	13	17	33		99	45	67
1	12	13	17	33		45		99
1	12	13	17	33		45		67
								99

Für Bubble-Sort gibt es verschiedene Varianten. Sie unterscheiden sich im wesentlichen im Laufbereich der Indizes. Hier die erste Variante:


 *kap09\_03.c*

```

01 /*****
02 * Bubble-Sort Variante 1
03 * Sortiert das angegebene Zahlen-Array in aufsteigender
04 * Reihenfolge.
05 * Parameter: Array - Zeiger auf das zu sortierende Array
06 *             Anzahl - Anzahl der Elemente im Array
07 * Rueckgabe: nichts
08 *****/
09 void BubbleSort1(int *Array, int Anzahl)
10 {
11     int i;        /* erstes Element im unsortierten Teil */
12     int j;        /* Index der aufsteigenden Blasen */
13     int temp;     /* fuer Austausch zweier Elemente */
14
15     for (i = 1; i < Anzahl; i++)
16         for (j = Anzahl - 1; j >= i; j--)
17             if (*(Array + j) < *(Array + j - 1))
18                 {
19                     temp = *(Array + j);
20                     *(Array + j) = *(Array + j - 1);
21                     *(Array + j - 1) = temp;
22                 }
23 }

```

Und hier die zweite Variante:

 *kap09\_04.c*

```

01 /*****
02 * Bubble-Sort Variante 2
03 * Sortiert das angegebene Zahlen-Array in aufsteigender
04 * Reihenfolge.
05 * Parameter: Array - Zeiger auf das zu sortierende Array
06 *             Anzahl - Anzahl der Elemente im Array
07 * Rueckgabe: nichts
08 *****/
09 void BubbleSort2(int *Array, int Anzahl)
10 {
11     int i;        /* erstes Element im unsortierten Teil */
12     int j;        /* Index der aufsteigenden Blasen */
13     int temp;     /* fuer Austausch zweier Elemente */
14
15     for (i = 0; i < Anzahl - 1; i++)
16         for (j = i + 1; j < Anzahl; j++)
17             if (*(Array + j) < *(Array + i))
18                 {
19                     temp = *(Array + i);
20                     *(Array + i) = *(Array + j);
21                     *(Array + j) = temp;

```

```

22     }
23 }

```

## 9.4. Shell-Sort

Shell-Sort ist eine Verfeinerung des Sortierens durch direktes Einfügen. Eine Effizienzsteigerung lässt sich beim Sortieren dadurch erreichen, dass Elemente in verkehrter Reihenfolge vorzugsweise über größere Distanzen ausgetauscht werden. Dieses nutzt Shell-Sort aus, indem zunächst alle Sequenzen von Elementen, die vier Positionen entfernt liegen, sortiert werden (4-Sortierung). Danach werden alle Elemente mit Abstand 2 (2-Sortierung) und zuletzt die Elemente mit Abstand 1 (1-Sortierung) sortiert.

### Beispiel:

```

99 13 45 12 17 33 67 1      unsortiert
17 13 45 1 99 33 67 12     4-Sortierung
17 1 45 12 67 13 99 33     2-Sortierung
1 12 13 17 33 45 67 99     1-Sortierung

```

Innerhalb der einzelnen  $k$ -Sortierungen wird durch "Sortieren durch direktes Einfügen" sortiert. Der Vorteil des Verfahrens liegt nun darin, dass entweder nur wenige Elemente sortiert werden müssen oder die Elemente bereits teilweise vorsortiert sind, so dass nur wenige Umstellungen notwendig sind.


Die Folge der Schrittweiten für die  $k$ -Sortierungen kann beliebig gewählt werden, sofern die letzte Schrittweite 1 ist. Im schlechtesten Fall verrichtet der Durchlauf mit Schrittweite 1 die ganze Arbeit. Eine gute Folge von Schrittweiten ist:

$$s(i) = 1, 3, 7, 15, 31, \dots$$

mit  $s(i) = 2 * s(i - 1) + 1$  und  $s(1) = 1$ .

Für diese Folge ergibt die mathematische Analyse des Algorithmus einen zu  $n^{1.2}$  proportionalen Aufwand.

Hier die Funktion für Shell-Sort:

 kap09\_05.c

```

01 /*****
02  * Shell-Sort
03  * Sortiert das angegebene Zahlen-Array in aufsteigender
04  * Reihenfolge.
05  * Parameter: Array - Zeiger auf das zu sortierende Array
06  *           Anzahl - Anzahl der Elemente im Array
07  * Rueckgabe: nichts
08  *****/
09 #define STEPS 5
10 void ShellSort1(int *Array, int Anzahl)
11 {
12     static steps[STEPS] = {1, 3, 7, 15, 31}; /* Schrittweiten */
13     int idx; /* Index der aktuellen Schrittweite */
14     int i; /* Index des naechsten einzusortierenden Elements */
15     int j; /* Laufindex zum Einsortieren */
16     int temp; /* fuer Austausch zweier Elemente */
17
18     for (idx = STEPS - 1; idx >= 0; idx--)
19         /* fuer alle Schrittweiten */
20         for (i = steps[idx]; i < Anzahl; i++)
21             {
22                 j = i;
23                 /* Einsortieren durch direktes Einfuegen */
24                 while (j > 0 && *(Array + j) < *(Array + j - steps[idx]))
25                     {

```

```

26         temp = *(Array + j);
27         *(Array + j) = *(Array + j - steps[idx]);
28         *(Array + j - steps[idx]) = temp;
29         j -= steps[idx];
30     }
31 }
32 }
33 #undef STEPS

```

## 9.5. Quick-Sort

Quick-Sort wird seinem Namen wirklich gerecht. Es ist die schnellste der vorgestellten Sortiermethoden und daher eine der am häufigsten verwendeten. Der Schlüssel zu Quick-Sort liegt im wiederholten Aufspalten des Arrays in zwei, wenn möglich etwa gleich große Teile, die für sich getrennt weiter sortiert werden. Dies ist dann möglich, wenn die Elemente einer Hälfte alle kleiner gleich den Elementen des anderen Teils sind.

Man muss also dafür sorgen, dass alle Elemente, die kleiner oder gleich einer bestimmten Schranke sind, in den linken Teil und alle Elemente größer gleich der Schranke in den rechten Teil des Arrays wandern. Dies wird durch die Funktion `partition` erreicht.

Nach dem Aufteilen des Arrays wird auf beide Teile das gleiche Verfahren wieder angewendet. Dies geschieht durch den rekursiven Aufruf der Quick-Sort-Routine (siehe auch Kapitel *Funktionen* Abschnitt *Rekursiver Funktionsaufruf*). Die Abbruchbedingung der Rekursion ist dann gegeben, wenn das zu sortierende Teil-Array leer ist oder nur noch aus einem einzigen Element besteht (was sollte dann noch sortiert werden?).

Die Problemgröße halbiert sich jeweils bei einer solchen Spaltung, woraus sich letztendlich auch die Effizienz des Verfahrens ergibt. Dabei muss man allerdings aufpassen, dass bei unglücklicher Wahl der Schranke nicht ein Teil leer bleibt, wenn z.B. alle Elemente kleiner bzw. größer als der Schranke sind.

Im einfachsten Fall wird das erste Element als Schranke gewählt und dieses nach dem Zerlegen des Arrays zwischen beide Teile gesetzt. Damit ist die Schranke selber bereits einsortiert und es müssen nur noch die beiden Teile vor und nach der Schranke sortiert werden.

Folgendes Beispiel zeigt die entstehenden Zerlegungen des Arrays: Wenn die einzelnen Partitionen am Ende wieder zusammengenommen werden, erhält man das sortierte Array. Dabei sind die Schranken immer fett und die bereits sortierten Teile unterstrichen gedruckt.

### Beispiel:

27	59	3	17	41	45	12	13	86	33	67	1	unsortiert
<b>27</b>	1	3	17	13	12	45	41	86	33	67	59	Schranke: 27
12	1	3	17	13	<u>27</u>	45	41	86	33	67	59	Schranke in der Mitte
<b>12</b>	1	3	17	13	<u>27</u>	<b>45</b>	41	33	86	67	59	Schranken: 12, 45
3	1	<u>12</u>	17	13	<u>27</u>	33	41	<u>45</u>	86	67	59	Schranken in der Mitte
<b>3</b>	1	<u>12</u>	<b>17</b>	13	<u>27</u>	<b>33</b>	41	<u>45</u>	<b>86</b>	67	59	Schranken: 3, 17, 33, 86
1	<u>3</u>	<u>12</u>	13	<u>17</u>	<u>27</u>	<u>33</u>	41	<u>45</u>	59	67	<b>86</b>	Schranken in der Mitte
<u>1</u>	<u>3</u>	<u>12</u>	<u>13</u>	<u>17</u>	<u>27</u>	<u>33</u>	<u>41</u>	<u>45</u>	<b>59</b>	67	<u>86</u>	Schranke: 59
<u>1</u>	<u>3</u>	<u>12</u>	<u>13</u>	<u>17</u>	<u>27</u>	<u>33</u>	<u>41</u>	<u>45</u>	<u>59</u>	<u>67</u>	<u>86</u>	Schranke in der Mitte


Wichtig für die Effizienz von Quick-Sort ist eine gute Wahl der Schranke. Quick-Sort ist am effizientesten, wenn das Array bei jeder Zerlegung in gleich große Teile partitioniert wird. Im schlechtesten Fall (z.B. wenn das Array bereits sortiert ist) wird bei jeder Zerlegung nur ein Element – nämlich die Schranke selber – abgespalten. Dann ist die Schnelligkeit von Quick-Sort verloren. Als Schranke sollte daher der mittlere Wert der Elemente gewählt werden. Allerdings kennt man diesen aber erst, wenn das Array sortiert ist. Eine

Möglichkeit ist es, drei Elemente aus dem Array herauszugreifen und den mittleren als Schranke zu verwenden. Im Fall des bereits sortierten Arrays wird die Leistung von Quick-Sort dadurch wesentlich verbessert.

Noch ein weiterer Punkt kann Probleme bereiten: Die rekursiven Aufrufe benötigen Speicherplatz für Funktionsparameter und Hilfsvariablen. Im oben geschilderten schlechtesten Fall entstehen  $n$  ( $n$  ist die Anzahl der Elemente) "hängende" rekursive Aufrufe, falls immer zuerst die größere der Partitionen weiter sortiert wird.

Außerdem kosten die ständigen Funktionsaufrufe Zeit. Dies kann mit einer iterativen Fassung (also Schleifen statt rekursive Funktionsaufrufe) vermieden werden.

Hier nun die Quick-Sort- und die dazugehörigen Funktionen:

 *kap09\_06.c*

```

01 /*****
02  * int partition(int *Array, int ui, int oi)
03  * Unterteilt das angegebene Array in zwei Teile, wobei
04  * im linken Teil alle Werte kleiner und im rechten Teil
05  * alle Werte groesser als die mittlere Schranke sind. Der
06  * Index der Schranke wird zurueckgegeben.
07  * Parameter: Array - das zu sortierende Array
08  *             ui   - der untere Index des Teils des
09  *                   Arrays, der sortiert werden soll
10  *             oi   - der obere Index (entsprechend ui)
11  * Rueckgabe: int  - Index der Schranke
12  *****/
13 int partition(int *Array, int ui, int oi)
14 {
15     int i = ui, j = oi;           /* Laufindizes                */
16     int temp;                    /* fuer Austausch zweier Elemente */
17     int *comp = Array + ui;      /* Zeiger auf Schranke          */
18
19     while (i <= j)
20     {
21         /* naechstes Element > comp von links suchen (im li. Teil) */
22         while (i <= j && *(Array + i) <= *comp)
23             i++;
24         /* naechstes Element < comp von rechts suchen (im re. Teil) */
25         while (j >= i && *(Array + j) >= *comp)
26             j--;
27         if (i < j)
28         {
29             temp = *(Array + i);
30             *(Array + i) = *(Array + j);
31             *(Array + j) = temp;
32             i++;
33             j--;
34         }
35     }
36     i--;
37     /* setze Vergleichselement (Schranke) zwischen beide Teile */
38     temp = *(Array + ui);
39     *(Array + ui) = *(Array + i);
40     *(Array + i) = temp;
41     return i;
42 }
43
44 /*****

```

```

45 * void qsort(int *Array, int ui, int oi)
46 * Unterteilt das Array in zwei Teile (Funktion
47 * partition) und ruft sich selber fuer beide Teile
48 * erneut auf.
49 * Parameter: Array - das zu sortierende Array
50 *             ui    - der untere Index des Teils des
51 *                 Arrays, der sortiert werden soll
52 *             oi    - der obere Index (entsprechend ui)
53 * Rueckgabe: keine
54 *****/
55 void qsort(int *Array, int ui, int oi)
56 {
57     int idx;                /* Schranke einer Zerlegung */
58
59     if (ui >= oi)          /* Abbruchbedingung der Rekursion */
60         return;
61     else
62     {
63         idx = partition(Array, ui, oi);
64         qsort(Array, ui, idx - 1); /* linken Teil rekursiv sortieren */
65         qsort(Array, idx + 1, oi); /* rechten Teil rekursiv sortieren */
66     }
67 }
68
69 *****/
70 * Quick-Sort
71 * Sortiert das angegebene Zahlen-Array in aufsteigender
72 * Reihenfolge.
73 * Parameter: Array - Zeiger auf das zu sortierende Array
74 *             Anzahl - Anzahl der Elemente im Array
75 * Rueckgabe: keine
76 *****/
77 void QuickSort(int *Array, int Anzahl)
78 {
79     qsort(Array, 0, Anzahl - 1);
80 }

```

# 10. Listen

In diesem Kapitel werden Strukturen vorgestellt, die während des Programmablaufs größer und auch wieder kleiner werden können. Diese Strukturen verbrauchen dabei immer nur so viel Speicherplatz, wie sie gerade benötigen. Solche Datenstrukturen werden **Dynamische Datenstrukturen** genannt.

Dynamische Datenstrukturen werden erst zur Laufzeit eines Programms aufgebaut, ihre Größe ist zur Zeit der Compilierung nicht bekannt. Im Kapitel *Dynamische Speicherverwaltung* wurden bereits mit den Funktionen zur Reservieren und Freigeben von Speicherbereichen die Grundlagen dafür gelegt.

## 10.1. *Einfach verkettete Listen*

Die bekannteste dynamische Datenstruktur ist die **einfach verkettete Liste**. Eine verkettete Liste besteht aus Elementen, die miteinander über Zeiger verbunden sind. Ein Element der Liste besteht dabei aus einer Datenstruktur mit den Daten - z.B. einer Adresse (bestehend aus Vorname, Nachname, Straße, usw.) - und dem Zeiger auf das nächste Element.

Im Folgenden werden die einzelnen Funktionen für eine einfach verkettete Liste vorgestellt. Dabei werden als Daten ein Index (sozusagen als Datensatznummer) und eine Fließkommazahl als Wert (Value) genommen.

Als erstes muss die Datenstruktur definiert werden.

```
struct ListElement
{   int Index;
    double Value;
    struct ListElement *Next;
};
```

Auf den ersten Blick ist es recht merkwürdig, dass in der Struktur ein Zeiger auf sich selber definiert wird. Aber der Zeiger zeigt ja nicht auf die eigene Struktur, sondern auf das nächste Element, das die gleiche Struktur hat.

Nun brauchen wir noch einen (Modul-globalen) Zeiger, der auf den Anfang der Liste - also auf das erste Listenelement - zeigt. Da am Anfang des Programms noch keine Listenelemente existieren, wird dieser Zeiger erst einmal auf NULL zeigen. Aus Effizienzgründen verwendet man häufig auch noch einen Zeiger auf das letzte Listenelement. Auch dieser Zeiger sollte am Anfang auf NULL zeigen.

```
struct ListElement *First = NULL, *Last = NULL;
```

Im Folgenden werden die drei gängigsten Listen-Operationen vorgestellt. Dabei müssen jeweils verschiedene Fälle berücksichtigt werden.

### *Neues Element anhängen*

Erst einmal wird ein neues Listenelement dynamisch erzeugt. Ein temporärer Zeiger zeigt auf diesen Speicherbereich. Nachdem sichergestellt ist, dass das Reservieren des Speicherbereiches erfolgreich war, können die Daten in das neue Listenelement eingesetzt werden. Ferner muss der Zeiger `Next` auf NULL gesetzt werden, um das Listenende zu kennzeichnen. Dann muss zwischen den beiden Fällen "Liste ist leer" und "Liste ist nicht leer" unterschieden werden.

1. Fall: Die Liste ist leer (`First == NULL`)

Die Zeiger `First` und `Last` zeigen am Anfang beide auf NULL und müssen nur auf das neue Listenelement "umgebogen" werden (d.h. sie zeigen anschließend auf das neue Listenelement).

2. Fall: Die Liste ist nicht leer (`First != NULL`)

Hier wird der Zeiger `Last->Next` (das bisherige Listenende) auf das neue Listenelement verbogen. Damit ist das neue Element an das Ende der Liste herangehangen. Nun muss nur noch der Zeiger `Last` auf das neue (letzte) Element zeigen.

Hier der Quelltext für eine entsprechende Funktion:

```

/*****
/* Hängt ein neues Element, das erst dynamisch erzeugt wird, */
/* an das Ende einer Liste. */
/* Parameter: NewIndex, NewValue - Daten des neuen Elementes */
/* Rückgabe : int - == Index, wenn erfolgreich */
/* == 0, wenn nicht erfolgreich */
*****/
int ListAppend(int NewIndex, double NewValue)
{
    struct ListElement *New = NULL;

    New = malloc(sizeof(struct ListElement));
    if (New != NULL)
    {
        New->Index = NewIndex; /* Daten */
        New->Value = NewValue;
        New->Next = NULL; /* neues Listenende */
        if (First == NULL)
        {
            /* Liste ist noch leer */
            First = Last = New;
        }
        else
        {
            /* Liste ist nicht leer */
            Last->Next = New;
            Last = New;
        }
        return NewIndex;
    }
    return 0;
}

```

### ***Neues Element einfügen***

Um eine Liste immer sortiert zu halten, müssen die neuen Elemente nicht am Ende herangehangen werden, sondern entsprechend der Sortierung eingefügt werden.

Das Einfügen funktioniert ganz ähnlich wie das Anhängen (erst einmal das neue Listenelement erzeugen, Daten setzen, usw.), allerdings müssen hier 4 Fälle unterschieden werden. In der folgenden Betrachtung wird nach dem Feld Wert aufsteigend sortiert.

1. Fall: Die Liste ist leer (`First == NULL`)

siehe 1. Fall vom Anhängen.

2. Fall: Neues Element am Anfang einfügen (`First->Value > NewValue`)

Zuerst muss der Zeiger Next des neuen Elements (also `Temp->Next`) auf das bisherige erste Element (`First`) zeigen. Dann muss der Zeiger First auf das neue Listenelement zeigen.

3. Fall: Neues Element am Ende einfügen (`Last->Wert < NewValue`)

siehe 2. Fall vom Anhängen.

4. Fall: Neues Element zwischen zwei Listenelementen einfügen

Hierfür werden 2 temporäre Zeiger benötigt. Der eine zeigt wie sonst auch auf das neue Listenelement (Zeiger New). Mit dem anderen wird die Liste erst einmal durchsucht, um die Stelle zu finden, an der das neue Element eingefügt werden soll. Dabei muss immer das Element geprüft werden, das auf das aktuelle Element (also das Element, auf das der Zeiger gerade zeigt) folgt. Ansonsten - wenn also immer das aktuelle Element geprüft wird - ist wohl der Zeiger auf das Element, vor dem eingefügt werden soll, vorhanden, aber es ist kein Zeiger auf das Element mehr vorhanden, nach dem eingefügt werden soll.

Für das eigentliche Einfügen sind dann zwei Schritte nötig: Zuerst wird der Zeiger Next des neuen Elementes auf das nachfolgende Listenelement und dann der Zeiger des aktuellen Elementes (also das Element, auf das der temporäre Zeiger gerade zeigt) auf das neue Element gesetzt.

Hier der Quelltext für eine entsprechende Funktion:

```

/*****
/* Fügt ein neues Element, das erst dynamisch erzeugt wird,      */
/* in eine Liste ein.                                             */
/* Parameter: NewIndex, NewValue - Daten des neuen Elementes    */
/* Rückgabe : int          - == Index, wenn erfolgreich          */
/*                    == 0, wenn nicht erfolgreich              */
*****/
int ListInsert(int NewIndex, double NewValue)
{
    struct ListElement *New = NULL, *Temp = NULL;

    New = malloc(sizeof(struct Listenelement));
    if (New != NULL)
    {
        New->Index = NewIndex; /* Daten */
        New->Value = NewValue;
        New->Next = NULL;      /* vorläufiges Listenende */
        if (First == NULL)
        {
            /* Liste ist noch leer */
            First = Last = New;
            return NewIndex;
        }
        if (First->Value > NewValue)
        {
            /* neues Element wird am Listenanfang eingefügt */
            New->Next = First;
            First = New;
            return NewIndex;
        }
        if (Last->Value <= NewValue)
        {
            /* neues Element wird am Listenende eingefügt */
            Last ->Next = New;
            Last = New;
            return NewIndex;
        }
        /* sonst: zwischen zwei Listenelemente einfügen */
        Temp = First;
        while (Temp->Next != NULL)
        {
            if (Temp->Next->Value > NewValue)
            {
                New->Next = Temp->Next;
                Temp->Next = New;
                return NewIndex;
            }
            Temp = Temp->Next;
        }
    }
    return 0;
}

```

## Element löschen

Als Parameter wird hier eine Angabe benötigt, welches Listenelement gelöscht werden soll; in dem bisherigen Beispiel könnte es der Index oder der Value sein.

Wie beim Anhängen und beim Einfügen muss zwischen verschiedenen Fällen unterschieden werden.

1. Fall: Die Liste ist leer (`First == NULL`)

In diesem Fall kann nichts gelöscht werden. Als Ergebnis wird eine 0 zurückgegeben.

2. Fall: Das erste Element wird gelöscht (`First->Value == DelValue`)

Mit einem temporären Zeiger wird auf das zu löschende Element gezeigt. Dann wird der Zeiger `First` auf das nachfolgende Element (`First->Next`) gesetzt. Wenn das erste Element auch das einzige Element war (Bedingung: `Last == Temp`), muss der Zeiger `Last` auf `NULL` gesetzt werden. Schließlich kann das gewünschte Element gelöscht werden.

3. Fall: Sonstige löschen

Hier werden wieder zwei temporäre Zeiger benötigt. Der eine zeigt auf das aktuelle Element, das geprüft wird, und der andere zeigt auf das vorige Element. Nun wird zuerst der Zeiger `Next` des vorigen Elements auf das nächste Element des aktuellen Elements gesetzt. Damit ist das aktuelle Element, das gelöscht werden soll, in der Liste überbrückt. Ist das zu löschende Element das letzte Listenelement (`Delete == Last`), muss noch der Zeiger `Last` auf das vorige Element (Zeiger `Prev`), das jetzt das neue letzte Element wird, gesetzt werden. Dann kann das aktuelle Element gelöscht werden.

Hier der Quelltext für eine entsprechende Funktion:

```
/* ***** */
/* Löscht das Listenelement, dessen Index dem DelIndex          */
/* entspricht. Der Speicherbereich des zu löschenden Elements   */
/* wird dynamisch freigegeben.                                  */
/* Parameter: DelIndex - Index des zu löschenden Elements      */
/* Rückgabe : int        - == Index, wenn erfolgreich           */
/*                   == 0, wenn nicht erfolgreich               */
/* ***** */
int ListDelete(int DelIndex)
{
    struct ListElement *Delete = NULL, *Prev = NULL;

    if (First == NULL)
        return 0;
    if (First->Index == DelIndex)
    {
        Delete = First;
        if (Last == First)
            /* nur ein Element in Liste */
            Last = NULL;
        First = First->Next;
        free(Delete);
        return DelIndex;
    }
    Prev = First;
    Delete = Prev->Next;
    while (Delete != NULL)
    {
        if (Delete ->Index == DelIndex)
        {
            Prev->Next = Delete->Next;
            if (Delete == Last)
                Last = Prev;
            free(Delete);
            return DelIndex;
        }
    }
}
```

```

    }
    Prev = Delete;
    Delete = Delete->Next;
}
return 0;
}

```

## 10.2. *Doppelt verkettete Listen*

Der Nachteil der einfach verketteten Liste besteht darin, dass man sie nur in Vorwärtsrichtung bearbeiten kann, weil die Information über das Vorgängerelement in den Listenelementen nicht vorhanden ist. Dies lässt sich aber durch Einfügen eines Zeigers `Prev` auf das Vorgängerelement in der Datenstruktur beheben. Dieser Zeiger muss nun in den jeweiligen Fällen korrekt gesetzt werden, dann kann eine Liste auch rückwärts durchlaufen werden.

### *Neues Element anhängen*

Auch hier wird erst einmal ein neues Listenelement dynamisch erzeugt. Ein temporärer Zeiger zeigt auf diesen Speicherbereich. Nachdem sichergestellt ist, dass das Reservieren des Speicherbereiches erfolgreich war, können die Daten in das neue Listenelement eingesetzt werden. Ferner müssen die Zeiger `Next` und `Prev` auf `NULL` gesetzt werden, um das Listenende zu kennzeichnen. Dann muss zwischen den beiden Fällen "Liste ist leer" und "Liste ist nicht leer" unterschieden werden.

1. Fall: Die Liste ist leer (`First == NULL`)

Die Zeiger `First` und `Last` zeigen am Anfang beide auf `NULL` und müssen nur auf das neue Listenelement "umgebogen" werden (d.h. sie zeigen anschließend auf das neue Listenelement). Die Zeiger `Next` und `Prev` brauchen nicht mehr verändert werden.

2. Fall: Die Liste ist nicht leer (`First != NULL`)

Hier wird der Zeiger `Last->Next` (das bisherige Listenende) auf das neue Listenelement verbogen. Damit ist das neue Element an das Ende der Liste herangehangen. Nun muss noch der `Prev`-Zeiger auf das alte Listenende `Last` zeigen. Als letztes muss nur noch der Zeiger `Last` auf das neue (letzte) Element zeigen.

Hier der Quelltext für eine entsprechende Funktion:

```

/*****
/* Hängt ein neues Element, das erst dynamisch erzeugt wird,   */
/* an das Ende einer Liste.                                     */
/* Parameter: NewIndex, NewValue - Daten des neuen Elementes   */
/* Rückgabe : int          - == Index, wenn erfolgreich          */
/*                   == 0, wenn nicht erfolgreich               */
*****/
int ListAppend(int NewIndex, double NewValue)
{
    struct ListElement *New = NULL;

    New = malloc(sizeof(struct ListElement));
    if (New != NULL)
    {
        New->Index = NewIndex; /* Daten */
        New->Value = NewValue;
        New->Prev = NULL;
        New->Next = NULL;      /* neues Listenende */
        if (First == NULL)
        {
            /* Liste ist noch leer */
            First = Last = New;
        }
    }
}

```

```
else
{
    /* Liste ist nicht leer */
    Last->Next = New;
    New->Prev = Last;
    Last = New;
}
return Index;
}
return 0;
}
```

***Neues Element einfügen***

***10.3. Stapel (Stacks)***

***10.4. Warteschlangen (Queues)***

# 11. Suchverfahren

Das Suchen nach Daten gehört mit zu den wichtigsten Grundlagen der Datenverarbeitung. Bevor nämlich ein Datensatz verändert werden kann, muss er zuvor lokalisiert werden. Suchen bedeutet, dass der Zugriffspfad (in Form einer Speicheradresse oder eines Indizes) zu dem gesuchten Datensatz ermittelt wird. Dazu werden die Werte der Datensätze mit einem vorgegebenen Wert (dem **Suchwert**) verglichen, bis entweder eine Übereinstimmung der miteinander verglichenen Werte eingetreten ist oder keine weitere Datensätze mehr vorhanden sind. Das Feld eines Datensatzes, dessen Wert mit dem Suchwert verglichen wird, heißt **Suchschlüssel**. Die Übereinstimmung des Schlüsselwertes mit dem entsprechenden Suchwert eines Datensatzes bezeichnet man als **key-matching**.

In allen Suchverfahren wird erst einmal davon ausgegangen, dass jedes Element nur einmal vorkommt. Sollten Elemente mehrfach vorkommen, wird hier immer nur das erste Element gefunden.

Für alle Suchverfahren definieren wir uns folgende Datenstruktur. In dieser Struktur können beliebige Daten (z.B. Adressendatensätze) stehen. Der Einfachheit halber wird hier nur eine ganze Zahl (z.B. als Datensatznummer verwendbar) eingesetzt. Der **Next**-Zeiger wird nur für die Suche in verketteten Listen benötigt.

```
struct D
{
    int Wert;
    /* weitere beliebige Daten */

    struct D *Next; /* nur für die verkettete Liste! */
};
```

```
typedef struct D Daten;
```

Desweiteren definieren wir für alle Suchverfahren eine Vergleichsfunktion. Diese erhält als Parameter einen Zeiger auf ein Element mit dem zu suchenden Wert sowie einen Zeiger auf das zu untersuchende Element. Die Funktion liefert eine negative Zahl, wenn das untersuchte Element kleiner als der Suchwert ist, eine positive Zahl, wenn das untersuchte Element größer als der Suchwert ist, und eine 0, wenn das untersuchte Element mit dem Suchwert übereinstimmt.

```
int Vergleiche(Daten *SuchElement, Daten *Element)
{
    return (Element->Wert - SuchElement->Wert);
}
```

## 11.1. *Sequentielle Suche*

Die einfachste Suche ist die **sequentielle Suche**. Die sequentielle Suche kann nur auf Arrays und verkettete Listen angewendet werden. Hierbei werden die Elemente vom ersten Element beginnend sequentiell (also ein Element nach dem anderen) durchsucht, bis das gesuchte Element gefunden oder das letzte Element erreicht wurde. Die Elemente müssen nicht sortiert sein. Der Suchalgorithmus ist sehr einfach, dafür aber auch sehr langsam. Zu den schnelleren Suchverfahren gehören die **binäre Suche**, die aber eine sortierte Datenmenge voraussetzt, sowie die **berechnete Suche**, auf die in den nachfolgenden Abschnitten eingegangen wird.

### *Suchen in Arrays*

Der einfachste Fall ist die Suche in einem unsortierten Array. Hierzu müssen die einzelnen Elemente – mit dem ersten Element beginnend – mit dem Suchwert verglichen werden. Ist das Element gefunden, wird der Zeiger auf das Element zurückgegeben. Sollte das Element nicht gefunden werden, wird ein **NULL**-Zeiger

zurückgegeben. Alternativ kann auch der Index des gefundenen Elementes bzw. eine -1 zurückgegeben werden.

**Beispiel:**

```
#include <stdio.h>

#define MAX 10

struct D
{
    int Wert;
    /* weitere beliebige Daten */
};

typedef struct D Daten;

int Vergleiche(Daten *, Daten *);
Daten *Suche(Daten *, int, int (*VerglFkt)(Daten *, Daten *), Daten *);

int main()
{
    Daten Array[MAX] = { {17}, {25}, { 5}, {75}, {39},
                        {56}, {21}, {82}, {48}, {61} };
    Daten *Element = NULL;
    Daten SuchElement;

    SuchElement.Wert = 39; /* es wird nach dem Wert 39 gesucht! */
    Element = Suche(Array, MAX, Vergleiche, &SuchElement);
    if (Element != NULL)
        printf("Gefundener Wert: %i\n", Element->Wert);
    else
        printf("Suchwert %i nicht gefunden!\n", SuchElement.Wert);
}

int Vergleiche(Daten *SuchElement, Daten *Element)
{
    return (Element->Wert - SuchElement->Wert);
}

Daten *Suche(Daten *Array, int Anzahl,
             int (*VerglFkt)(Daten *, Daten *), Daten *SuchElement)
{
    Daten *Temp = Array; /* erstes Element */
    int i = 0;

    for ( ; i < Anzahl; i++)
    {
        if (VerglFkt (SuchElement, Temp) == 0)
            return Temp; /* gesuchtes Element gefunden */
        Temp++; /* nächstes Element */
    }
    return NULL; /* Element nicht gefunden */
}
```

Ist das Array sortiert, kann die Suchfunktion noch etwas modifiziert und damit beschleunigt werden für den Fall, dass das gesuchte Element nicht im Array enthalten ist. Dazu wird die Schleife in der Suchfunktion beendet, sobald das Ergebnis der Vergleichsfunktion nicht mehr negativ ist. Anschließend muss nur noch geprüft werden, ob das Ergebnis der Vergleichsfunktion gleich 0 (Element gefunden) oder größer 0 ist

(Element nicht gefunden). Die Suchfunktion könnte dann wie folgt aussehen (im Hauptprogramm müssen jetzt die Arrayelemente sortiert vorliegen!):

```
Daten *Suche(Daten *Array, int Anzahl,
             int (*VerglFkt)(Daten *, Daten *), Daten *SuchElement)
{
    Daten *Temp = Array; /* erstes Element */
    int i = 0;

    while (VerglFkt (SuchElement, Temp) < 0)
    {
        Temp++;          /* nächstes Element */
        i++;
        if (i == Anzahl)
            return NULL; /* Arrayende erreicht */
    }
    if (VerglFkt (SuchElement, Temp) == 0)
        return Temp;    /* gesuchtes Element gefunden */
    return NULL;       /* Element nicht gefunden */
}
```

Die Suche kann weiter beschleunigt werden, wenn in der Schleife auf die Abfrage nach dem Arrayende verzichtet werden kann. Dies kann erreicht werden, wenn als letztes Element im Array (notfalls muss das Array um ein Element vergrößert werden) das gesuchte Element eingetragen wird. Dadurch wird auf jeden Fall ein Element gefunden; es muss nur noch am Ende abgefragt werden, ob das gefundene Element das letzte Element ist oder nicht. Dieses letzte Element wird auch **Wächter** (im engl. *Sentinel*) genannt. Die Suchfunktion könnte dann folgendermaßen aussehen (im Hauptprogramm muss jetzt das Array um ein Element größer sein; ferner entfällt bei der Suchfunktion der Parameter Anzahl!):

```
Daten *Suche(Daten *Array, int (*VerglFkt)(Daten *, Daten *),
             Daten *SuchElement)
{
    Daten *Temp = Array; /* erstes Element */

    Array[MAX].Wert = SuchElement->Wert; /* Wächter setzen */

    while (VerglFkt (SuchElement, Temp) < 0)
        Temp++;          /* nächstes Element */

    if (Temp != &(Array[MAX])) /* nicht der Wächter? */
        return Temp;        /* gesuchtes Element gefunden */
    return NULL;           /* Element nicht gefunden */
}
```

Die Suche mit einem Wächter kann auch auf unsortierte Arrays angewendet werden.

### ***Suchen in verketteten Listen***

In diesem Abschnitt wird vorausgesetzt, dass die Daten in einer einfach verketteten Liste (siehe Kapitel *Listen* Abschnitt *Einfach verkettete Listen*) gespeichert sind. Die Datenstruktur muss gegenüber dem vorigen Abschnitt noch um den Zeiger auf das nächsten Datenelement erweitert werden.

#### **Beispiel:**

```
#include <stdio.h>
#include <malloc.h>

#define MAX 10

struct D
```

```

{
    int Wert;
    /* weitere beliebige Daten */

    struct D *Next;
};

typedef struct D Daten;

int Vergleiche(Daten *, Daten *);
Daten *Suche(Daten *, int, int (*VerglFkt)(Daten *, Daten *), Daten *);

int main()
{
    int Array[MAX] = { 17, 25, 5, 75, 39, 56, 21, 82, 48, 61 };
    int i;
    Daten *Wurzel, *Temp;
    Daten *Element = NULL;
    Daten SuchElement;

    Wurzel = malloc(sizeof(Daten));
    Wurzel->Wert = Array[0];
    Wurzel->Next = NULL;
    Temp = Wurzel;
    for (i = 1; i < MAX; i++)
    {
        Temp->Next = malloc(sizeof(Daten));
        Temp->Next->Wert = Array[i];
        Temp->Next->Next = NULL;
        Temp = Temp->Next;
    }

    SuchElement.Wert = 39; /* es wird nach dem Wert 39 gesucht! */
    Element = Suche(Wurzel, MAX, Vergleiche, &SuchElement);
    if (Element != NULL)
        printf("Gefundener Wert: %i\n", Element->Wert);
    else
        printf("Suchwert %i nicht gefunden!\n", SuchElement.Wert);
}

int Vergleiche(Daten *SuchElement, Daten *Element)
{
    return (Element->Wert - SuchElement->Wert);
}

Daten *Suche(Daten *Start, int Anzahl,
             int (*VerglFkt)(Daten *, Daten *), Daten *SuchElement)
{
    Daten *Temp = Start; /* erstes Element */
    int i = 0;

    for ( ; i < Anzahl; i++)
    {
        if (VerglFkt (SuchElement, Temp) == 0)
            return Temp; /* gesuchtes Element gefunden */
        Temp = Temp->Next; /* nächstes Element */
    }
}

```

```

    return NULL;          /* Element nicht gefunden */
}

```

Genauso wie bei der sequentiellen Suche in Arrays gibt es die Variationen, dass die Suche abgebrochen wird, sobald das Ergebnis der Vergleichsfunktion nicht mehr negativ ist (nur bei sortierten, verketteten Listen) sowie dass am Ende der Liste ein Wächter eingesetzt wird.

Bei der Suchfunktion könnte auch auf die Anzahl der Elemente als Parameter verzichtet werden. Es müsste dann als Abbruchbedingung nur abgefragt werden, ob der `Next`-Zeiger gleich dem `NULL`-Zeiger ist. Dann ließe sich die Suchfunktion aber nicht mehr auf Ringlisten anwenden.

## 11.2. Binäre Suche

Voraussetzung für die Binäre Suche ist ein sortiertes Datenarray. Bei komplexen Daten – z.B. Array einer Datenstruktur – kann das Array natürlich immer nur nach einem Feld der Datenstruktur sortiert sein. Gegebenenfalls muss das Array erst nach dem Suchschlüssel sortiert werden.

Die Binäre Suche bestimmt zunächst einmal das mittlere Element des ausgewählten Suchintervalls – am Anfang ist dies das komplette Datenarray. Dadurch entstehen zwei Teilintervalle: Im linken Teilintervall sind alle Elemente (bei aufsteigender Sortierung) wertmäßig kleiner als das mittlere Element und im rechten Teilintervall sind alle Elemente wertmäßig größer als das mittlere Element. Nun wird der Suchwert mit dem Wert des mittleren Elementes verglichen. Ist der Suchwert gleich dem Wert des mittleren Elementes, so ist das gesuchte Element gefunden und die Suche kann beendet werden. Ist der Suchwert kleiner als der Wert des mittleren Elementes, wird die Binäre Suche auf das linke Teilintervall angewendet, andernfalls (Suchwert ist größer als der Wert des mittleren Elementes) wird die Binäre Suche auf das rechte Teilintervall angewendet.

### Beispiel:

```

#include <stdio.h>

#define MAX 10

struct D
{
    int Wert;
    /* weitere beliebige Daten */
};

typedef struct D Daten;

int Vergleiche(Daten *, Daten *);
Daten *Suche(Daten *, int, int (*VerglFkt)(Daten *, Daten *), Daten *);

int main()
{
    Daten Array[MAX] = { { 5}, {17}, {21}, {25}, {39},
                        {48}, {56}, {61}, {75}, {82} };
    Daten *Element = NULL;
    Daten SuchElement;

    SuchElement.Wert = 39; /* es wird nach dem Wert 39 gesucht! */
    Element = Suche(Array, MAX, Vergleiche, &SuchElement);
    if (Element != NULL)
        printf("Gefundener Wert: %i\n", Element->Wert);
    else
        printf("Suchwert %i nicht gefunden!\n", SuchElement.Wert);
}

```

```

int Vergleiche(Daten *SuchElement, Daten *Element)
{
    return (Element->Wert - SuchElement->Wert);
}

Daten *Suche(Daten *Array, int Anzahl,
             int (*VerglFkt)(Daten *, Daten *), Daten *SuchElement)
{
    int ui = 0;                /* unterer Index          */
    int oi = Anzahl - 1;      /* oberer Index          */
    int VerglErg;             /* Vergleichsergebnis   */
    int VerglIndex;          /* Vergleichsindex       */
    Daten *VerglElement = NULL; /* Vergleichselement    */

    do
    {
        VerglIndex = (ui + oi) / 2;
        VerglElement = &(Array[VerglIndex]);
        VerglErg = VerglFkt (SuchElement, VerglElement);
        if (VerglErg == 0)      /* Element gefunden     */
            return VerglElement;
        if (VerglErg > 0)      /* linkes Teilintervall */
            ui = VerglIndex + 1;
        else                    /* rechtes Teilintervall */
            oi = VerglIndex - 1;
    } while (ui <= oi);
    return NULL;              /* Element nicht gefunden */
}

```

### ***11.3. Berechnete Suche (Hashing)***

## **12. Textsuche**

### ***12.1. Einfache Textsuche***

### ***12.2. Knuth-Morrison-Pratt-Algorithmus***

## **13. Einführung in die Graphentheorie**

### *13.1. Allgemeine Begriffe*

### *13.2. Baumspezifische Begriffe*

## **14. Bäume**

*14.1. Unsortierte Bäume*

*14.2. Sortierte Bäume*

*14.3. Suche in Bäumen*

# 15. ASCII-Tabellen

ASCII steht für **American Standard Code for Information Interchange**. Die ASCII-Tabelle gliedert sich auf in ASCII-Steuerzeichen und den druckbaren ASCII-Zeichen.

## ASCII-Steuerzeichen:

Nr.	hex	Abkürzung und Name	C/C++
0	0x00	NUL - Nullzeichen	\0
1	0x01	SOH - start of heading	\x01
2	0x02	STX - start of text	\x02
3	0x03	ETX - end of text	\x03
4	0x04	EOT - end of transmission	\x04
5	0x05	ENQ - enquiry	\x05
6	0x06	ACK - acknowledge	\x06
7	0x07	BEL - bell	\a
8	0x08	BS - backspace	\b
9	0x09	HT - horizontal tab	\t
10	0x0A	LF - line feed	\n
11	0x0B	VT - vertical tab	\v
12	0x0C	FF - form feed	\f
13	0x0D	CR - carriage return	\r
14	0x0E	SO - shift out	\x0E
15	0x0F	SI - shift in	\x0F
16	0x10	DLE - data link escape	\x10
17	0x11	DC1 - device control 1	\x11
18	0x12	DC2 - device control 2	\x12
19	0x13	DC3 - device control 3	\x13
20	0x14	DC4 - device control 4	\x14
21	0x15	NAK - negative acknowledge	\x15
22	0x16	SYN - synchronous idle	\x16
23	0x17	ETB - end of transmission block	\x17
24	0x18	CAN - cancel	\x18
25	0x19	EM - end of medium	\x19
26	0x1A	SUB - substitute	\x1A
27	0x1B	ESC - escape	\x1B
28	0x1C	FS - file separator	\x1C
29	0x1D	GS - group separator	\x1D
30	0x1E	RS - record separator	\x1E
31	0x1F	US - unit separator	\x1F
127	0x7F	DEL - delete	\x7F

## druckbare ASCII-Zeichen:

Nr.	hex	Zeichen	C/C++	Nr.	hex	Zeichen	C/C++
32	0x20			80	0x50	P	P
33	0x21	!	!	81	0x51	Q	Q
34	0x22	"	\"	82	0x52	R	R
35	0x23	#	#	83	0x53	S	S
36	0x24	\$	\$	84	0x54	T	T
37	0x25	%	%	85	0x55	U	U
38	0x26	&	&	86	0x56	V	V

39	0x27	'	\'	87	0x57	W	W
40	0x28	(	(	88	0x58	X	X
41	0x29	)	)	89	0x59	Y	Y
42	0x2A	*	*	90	0x5A	Z	Z
43	0x2B	+	+	91	0x5B	[	[
44	0x2C	,	,	92	0x5C	\	\\
45	0x2D	-	-	93	0x5D	]	]
46	0x2E	.	.	94	0x5E	^	^
47	0x2F	/	/	95	0x5F	_	_
48	0x30	0	0	96	0x60	`	`
49	0x31	1	1	97	0x61	a	a
50	0x32	2	2	98	0x62	b	b
51	0x33	3	3	99	0x63	c	c
52	0x34	4	4	100	0x64	d	d
53	0x35	5	5	101	0x65	e	e
54	0x36	6	6	102	0x66	f	f
55	0x37	7	7	103	0x67	g	g
56	0x38	8	8	104	0x68	h	h
57	0x39	9	9	105	0x69	i	i
58	0x3A	:	:	106	0x6A	j	j
59	0x3B	;	;	107	0x6B	k	k
60	0x3C	<	<	108	0x6C	l	l
61	0x3D	=	=	109	0x6D	m	m
62	0x3E	>	>	110	0x6E	n	n
63	0x3F	?	?	111	0x6F	o	o
64	0x40	@	@	112	0x70	p	p
65	0x41	A	A	113	0x71	q	q
66	0x42	B	B	114	0x72	r	r
67	0x43	C	C	115	0x73	s	s
68	0x44	D	D	116	0x74	t	t
69	0x45	E	E	117	0x75	u	u
70	0x46	F	F	118	0x76	v	v
71	0x47	G	G	119	0x77	w	w
72	0x48	H	H	120	0x78	x	y
73	0x49	I	I	121	0x79	y	y
74	0x4A	J	J	122	0x7A	z	z
75	0x4B	K	K	123	0x7B	{	{
76	0x4C	L	L	124	0x7C		
77	0x4D	M	M	125	0x7D	}	}
78	0x4E	N	N	126	0x7E	~	~
79	0x4F	O	O				