Inhalt

1 TEM 11

2 Zahlensysteme

2.1 Natürliche Zahlen

- Allgemein: Natürliche Zahlen n werden dargestellt durch:

$$n = \sum_{i=0}^{N-1} b_i \, B^i$$

n = natürliche Zahl
B = Basis des Zahlensystems
b = Ziffern
N = Anzahl Stellen
i = Position

N = 4
i= 3210

$$n = (2A03)_{16} = b_3 * 16^3 + b_2 * 16^2 + b_1 * 16^1 + b_0 * 16^0$$

$$n = (2A03)_{16} = 2 * 16^3 + A * 16^2 + 0 * 16^1 + 3 * 16^0 = 10755$$

$10 * 16^2$

2.2 Reele Zahlen

Festpunktzahl z.B. 17.439

Festpunktzahl Charakteristiken:

- der **Punkt** (d.h. das "**Komma**") steht immer an einer bestimmten festgelegten Stelle (zwischen z_0 und z_{-1})
- die Zahl hat die Länge n + m
  (n Stellen vor dem Punkt, m Stellen nach dem Punkt)

Formel gilt auch für 3er, 4er, ...x-er- System!

Allgemein:

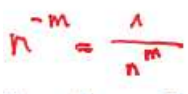
zahl = $(z_{n-1}z_{n-2}\ldots z_1 z_0 \, | \, z_{-1}z_{-2}\ldots z_{-m})_{(2)}$ d.h. $\sum_{i=-m}^{n-1} z_i 2^i$

Beispiel: (im 2-er System!)

$(11.011)_2 = 1\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 1\cdot 2^{-2} + 1\cdot 2^{-3}$
$= 2 + 1 + 0{\cdot}0.5 + 0.25 + 0.1\,2\,5$
$= (3.375)_{10}$

1 0 | -1 -2 -3
n Stellen | -m Stellen

Wichtige Formel:

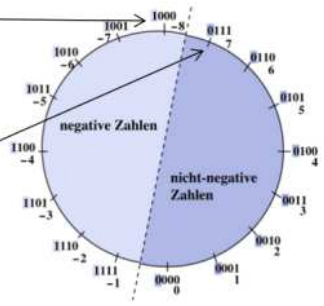


2.3 Grundlagen Zweierkomplement

- Allgemein:
  - Kleinste darstellbare negative Zahl: $-B^{s-1}$
  - Grösste darstellbare **positive** Zahl: $B^{s-1}-1$

B = Zahlensystem
s = Registergrösse (=Anzahl Bit)

- für Zweiersystem (B = 2) **kleinste negative Zahl**
  bei **s** = 4: −24−1 = −23 = **−8**
  bei **s** = 8: −28−1 = −27 = −128
  bei **s** = 16: −216−1 = −215 = −32768
  bei **s** = 32: −232−1 = −231 = −2147483648
- für Zweiersystem (B=2) **grösste positive Zahl**
  bei **s** = 4: 24−1 − 1 = 23 − 1 = **7**
  bei **s** = 8: 28−1 − 1 = 27 − 1 = 127
  bei **s** = 16: 216−1 − 1 = 215 − 1 = 32767
  bei **s** = 32: 232−1 − 1 = 231 − 1 = 2147483647



- Vorgehen: z.B. 2er-Komplement von 5:
  - jedes einzelne Bit wird für den Wert (hier z.B. 5) umgekehrt UND
  - nachher wird immer 1 dazugezählt

5	0	1	0	1
	1	0	1	0
+	0	0	0	1
2er-Komplement d.h. -5:	1	0	1	1

- 2er-Komplement von -6:
  - jedes einzelne Bit wird für den Wert (hier z.B. -6) umgekehrt UND
  - nachher wird immer 1 dazugezählt

-6	1	0	1	0
	0	1	0	1
+	0	0	0	1
			1	
2er-Komplement d.h. 6:	0	1	1	0

2.3.1 Beweis

$$\sum z_i \cdot 2^i = 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 10$$

2.4 IEEE Darstellung Gleitpunktzahl

Formel zur Darstellung einer Gleitpunktzahl im IEEE-Format:

$(-1)^S \cdot \underline{(2^{B-bias})} \cdot \underline{(1.f_N.....f_0)}$

S — *SIGN* S = 0 (positiv); S = 1 (negativ)

2^{B-bias} — *EXPONENT* bias = 127 (`float`), bias = 1023 (`double`)

$(1.f_N.....f_0)$ — *SIGNIFICAND* N=22 (`float`=23 Stellen), N=51 (`double`=52 Stellen)

B = Biased Exponent (zu speichernder Exponent)

- Beispiel:

  1 10001101 01100100000000000000000

  => -27776.0 d.h. $-1.6953125 * 2^{14}$

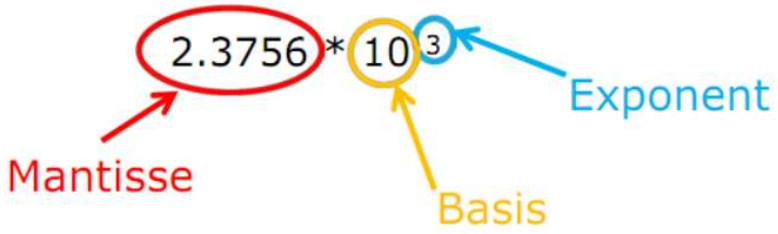
Beispiel: (im 2-er System!)

$(11.011)_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}$

$= 2 + 1 + 0 \cdot 0.5 + 0.25 + 0.125$

$= (3.375)_{10}$

1 0 — n Stellen; -1 -2 -3 — -m Stellen



2.5 Umrechung von Zahlensystemen

Umrechnung von Dual ins Dezimal:

Am einfachsten verwendet man unten stehende Liste:

Zweier-Potenzen-Reihe

2^0	=1	=	1	
2^1	=2	=	2	
2^2	=2*2	=	4	
2^3	=2*2*2	=	8	
2^4	=2*2*2*2	=	16	
2^5	=2*2*2*2*2	=	32	
2^6	=2*2*2*2*2*2	=	64	
2^7	=2*2*2*2*2*2*2	=	128	
2^8	=2*2*2*2*2*2*2*2	=	256	
2^9	=2*2*2*2*2*2*2*2*2	=	512	
2^{10}	=2*2*2*2*2*2*2*2*2*2	=	1.024	1 kB (Kilobyte)
2^{11}	=2*2*2*2*2*2*2*2*2*2*2	=	2.048	
2^{12}	=2*2*2*2*2*2*2*2*2*2*2*2	=	4.096	
2^{13}	=2*2*2*2*2*2*2*2*2*2*2*2*2	=	8.192	
2^{14}	=2*2*2*2*2*2*2*2*2*2*2*2*2*2	=	16.384	
2^{15}	=	=	32.768	
2^{16}	=	=	65.536	
2^{17}	=	=	131.072	
2^{18}	=	=	262.144	
2^{19}	=	=	524.288	
2^{20}	=	=	1.048.576	1 MB (Megabyte)
2^{21}	=	=	2.097.152	
2^{22}	=	=	4.194.304	
2^{23}	=	=	8.388.608	
2^{24}	=	=	16.777.216	
2^{25}	=	=	33.554.432	
2^{26}	=	=	67.108.864	
2^{27}	=	=	134.217.728	
2^{28}	=	=	268.435.456	
2^{29}	=	=	536.870.912	
2^{30}	=	=	1.073.741.824	1 GB (Gigabyte)
2^{31}	=	=	2.147.483.648	
2^{32}	=	=	4.294.967.296	
2^{33}	=	=	8.589.934.592	
2^{34}	=	=	17.179.869.184	
2^{35}	=	=	34.359.738.368	
2^{36}	=	=	68.719.476.736	
2^{37}	=	=	137.438.953.472	
2^{38}	=	=	274.877.906.944	
2^{39}	=	=	549.755.813.888	
2^{40}	=	=	1.099.511.627.776	1 TB (Terabyte)
2^{41}	=	=	2.199.023.255.552	

Umrechung Dual → Hex und Oktal → Hex

Beispiel 1:

(1011 0101 0001)$_B$
⇓ ⇓ ⇓
(B 5 1)$_H$

– **Wandlung: Dual- in Hexadezimalsystem**

Beispiel 2:

(5 2 1)$_O$
⇓ ⇓ ⇓
(101 010 001)$_B$

– **Wandlung: Oktal- in Dualsystem**

3 Algorithmen

3.1 Allgemein

Unterschied Effektive vs. Effizienz
Effektive = Ziel erreichen, wie, egal --> Irgendwann kommst du zum Ziel
Effizienz = Ziel erreichen, in vorgegebener Zeit --> Möglichst wenig Aufwand, wenig Zeit, wenig Geld erreichen

Beschreiben eine Verarbeitungsvorschrift, dies kann sein:
Vorschrift zur Lösung einer Aufgabe
Vorschrift zur Lösung eines Problems

3.1.1 Begriffe

Begriff	**Beschreibung**
terminiert	Algorithmus ist nach n-Schritten beendet, hat er kein bestimmtes Ende, spricht man einem nicht-terminiertem Algorithmus
Determinismus	Für alle Eingaben ist der Ablauf des Algorithmus eindeutig bestimmt, anderenfalls heisst er nicht-deterministisch
Determiniertheit	Ein Algorithmus heißt determiniert, wenn er bei gleichen zulässigen Eingabewerten stets das gleiche Ergebnis liefert. Andernfalls heißt er nicht-determiniert.

Wichtige Aussagen: Ein deterministischer Algorithmus ist immer determiniert,
d. h. er liefert bei gleicher Eingabe immer die gleiche Ausgabe.
Die "Umkehrung" aber gilt nicht: So gibt es Algorithmen, die nicht-deterministisch,
aber trotzdem determiniert sind (d. h. das gleiche Ergebnis liefern).

4 Algorithmus ||

Bezeichnet eine Verarbeitungsvorschrift.

Ein Algorithmus ist eine in der Beschreibung und Ausführung endliche, deterministische und effektive Vorschrift zur Lösung eines Problems, die effizient sein sollte.

endlich: nach einer endlichen Zeit wird der Algorithmus beendet,

deterministisch: nur definierte und reproduzierbare Zustände treten auf d.h. bei gleicher Eingabe folgt immer gleiche Ausgabe und zusätzlich wird die gleiche Folge von Zuständen durchlaufen. Zu jedem Zeitpunkt ist der nachfolgende Abarbeitungsschritt des Algorithmus eindeutig festgelegt. Es gibt auch "nicht-deterministische" Algorithmen!

effektiv: Grad (Mass) für die Zielerreichung: Es gibt Aufschluss darüber, wie nahe ein erzieltes Ergebnis dem angestrebten Ergebnis gekommen ist. Wir erwarten bem Programmieren implizit meist 100% ige Zielerreichung!

effizient: Mass für die Wirtschaftleichkeit z.B. geringer Verbrauch an Ressourcen wie Speicherplatz und Rechenzeit.

Abstrahierung: Ein Algorithmus löst eine ganze Klasse von gleichartigen Problemen. Die Wahl des speziellen Problems erfolgt über Parameter.

Finitheit statisch: Die Beschreibung des Algorithmus selbst ist endlich. - Finitheit dynamisch: Ein in Bearbeitung befindlicher Algorithmus hat zu jedem Zeitpunkt eine endliche Fülle von Datenstrukturen und Zwischenergebnissen. Er belegt deshalb endlich viele Ressourcen im System.

Sequenzialität: Ein Algorithmus ist aus Einzelschritten aufgebaut. In jedem dieser Schritte wird eine einfache Operation ausgeführt, wie z.B. eine Addition oder eine Zuweisung zu einer Variablen.

Realisierbarkeit: Die genannten Operationen müssen tatsächlich in der Praxis durchführbar sein. Die Quadratur des Kreises oder die Division durch Null sind also nicht algorithmisch lösbar, ebenso wenig wie die Bestimmung der Masse der Erde auf ein Gramm genau.

Terminierung: Normalerweise gehen wir davon aus, dass ein Algorithmus terminiert, das heisst, nach einer absehbaren Zeit kontrolliert abbricht. Gewisse Algorithmen – und Programme – laufen potenziell endlos wie z.B. Betriebssysteme, Prozessleitsysteme usw.
Determinismus: Ein Algorithmus ist dann deterministisch, wenn zu jedem Zeitpunkt nur eine Möglichkeit des weiteren Ablaufs, oder des Abbruchs, besteht. Ist ein nichtdeterministischer Algorithmus durch Wahrscheinlichkeiten oder Zufälle gesteuert, dann heisst er stochastisch.

Determiniertheit: Ein Algorithmus ist dann determiniert, wenn er bei gleichen Startparametern und Eingabewerten auf gleiche Art terminiert und gleiche Ergebnisse liefert (dabei aber möglicherweise nicht immer die gleiche Sequenz von Einzelschritten wählt).

Wichtig!!!

4.1 Rekursive Programmierung

Beispiel für rekursive Programmierung in Java:

```
public class Calculator {
    public static void main(String[] args) {
        System.out.println(getFactorial(32));
    }
    public static double getFactorial(double n){
        if(n==1){
            return 1;
        }
        return n * getFactorial(n-1);
    }

    public static int getFibonacci(int z){
            if(z==0){
                return 0;
            }
            if(z==1){
                return 1;
            }
            return getFibonacci(z-1) + getFibonacci(z-2);
        }
    }
}
```

Der base case ist das IF-Statement, die Rekursion natürlich der wiederholte Funktionsaufruf.

4.1.1 Zeichenketten Invertierung

Einfache Invertierung einer Zeichenkette

```
using System;

class InvertMain
{
   public static string Invert(string old)
   {
       if(old.Length < 2) return old;
       return old.Substring(old.Length-1) + Invert(old.Remove(old.Length-1, 1));
   }

   public static void Main()
   {
       string old = Console.ReadLine();
       Console.WriteLine(Invert(old));
   }
}
```

Prosit

Ermittelt wie oft die Gläser klingen, wenn n Personen, jede mit jedem anstossen

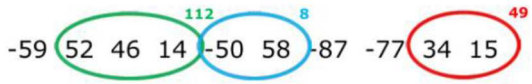
```
class PrositMain
{
    private static long Prosit(long n)
    {
        if (n > 2) return n-1 + Prosit(n-1);
        else return 1;
    }

    public static void Main()
    {
        Console.Write("Wie viele Gäste sind an der Party: ");
        int gaeste = Int32.Parse(Console.ReadLine());
        Console.WriteLine("Die Gläser klingen {0} mal.", Prosit(gaeste));
        Console.ReadKey();
    }
}
```

5 Ressourcenkomplexität

Die Komplexitätstheorie als Teilgebiet der Theoretischen Informatik befasst sich mit der Komplexität von algorithmisch behandelbaren Problemen auf verschiedenen mathematisch definierten formalen Rechnermodellen. Die Komplexität von Algorithmen wird in deren Ressourcenverbrauch gemessen, meist Rechenzeit oder Speicherplatzbedarf.

Beispiel – Zahlenreihe:



Nun möchte man herausfinden, welcher zusammenhängende Abschnitt von Zahlen, die grösste Summe aufweist.

5.1 Kubischer Algorithmus

```
int maxfolge1(int z[], int n) {
    int i , j , k, sum, max = −10000000;
        for ( i = 0; i < n; i++)
            for ( j = i ; j < n; j++) {
                sum = 0;
                for (k = i ; k <= j; k++)
                    sum += z[k];
                if (sum > max)
                    max = sum;
            }
        return max;
}
```

- Zeitaufwand: 3 geschachtelte for-Schleifen
- Aufwand: etwa proportional n^3

5.2 Quadratischer Algorithmuss

```
int maxfolge2(int z[], int n) {
    int i , j , sum, max = −10000000;
    for ( i=0; i<n; i++) {
        sum = 0;
        for ( j=i; j<n; j++) {
            sum += z[j];
            if (sum > max)
                max = sum;
        }
    }
    return max;
}
```

- Zeitaufwand: 2 geschachtelte for-Schleifen
- Aufwand: etwa proportional n^2

5.3 Linearer Algorithmus

```
int maxfolge3(int z[], int n) {
    int i , s, gesamtmax = −10000000, endesumme = 0;
    for ( i=0; i<n; i++) {
        if ((z[i] < 0 && Math.abs(z[i]) <  gr_min_zahl)){
          gr_min_zahl = Math.abs(z[i]);}
        endesumme = ((s=endesumme+z[i]) > 0) ? s : 0;
        if (endesumme > gesamtmax)
            gesamtmax = endesumme;
    }
    return gesamtmax;
}
```

- Zeitaufwand: 1 for-Schleife
- Aufwand: etwa proportional n

5.4 Exponentieller Algorithmus

```
int prim(int zahl, int teiler ) {
    if (zahl < 2 || zahl%2 == 0 || zahl%teiler == 0)
        return 0; /* keine Primzahl */
    else if ( teiler *teiler > zahl)
        return 1; /* Primzahl */
    return prim(zahl, teiler+1);
}
```

- Ist am schnellsten
- Zeitaufwand: 0 Schleifen (Schaltkreis etscheiden)
- Aufwand: 2^n

Die Wahl des Algorithmus ist gravierend für das Zeitverhalten des Programms. Zusammenfassend für 10'000 Zahlen kann man folgende Aussage machen:

`maxfolge1()`: $t(n) = 10000^3 = 10^{4^3} = 10^{12} = 1$ Billion

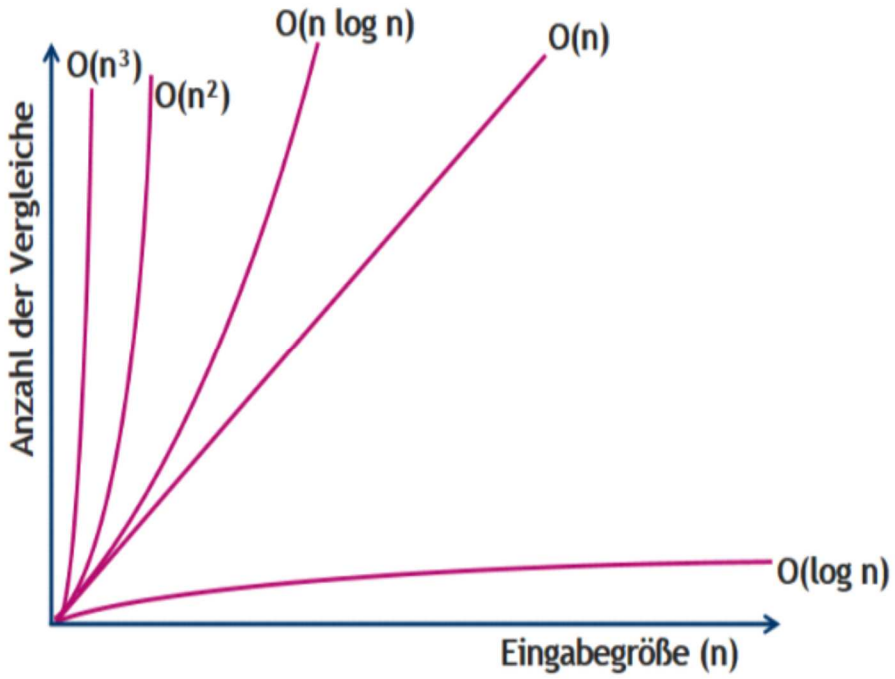
`maxfolge2()`: $t(n) = 10000^2 = 10^{4^2} = 10^8 = 100$ Millionen
also 10000 mal schneller als `maxfolge1()`

`maxfolge3()`: $t(n) = 10000^1 = 10^4$
also 10000 mal schneller als `maxfolge2()` und
100 Millionen mal schneller als `maxfolge1()`.

Die verschiedenen Algorithmen lassen sich also in mathematische Funktionen unterteilen:

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

Eine Visualisierung der Anzahl der Vergleiche, die eine solcher Algorithmus macht lässt sich einfach aufzeigen:



6 Kodierung

Zeichensatz

- Liste von Zeichen

Zeichenkodierung

- Eindeutige Zuordnung von Zeichen zu einer Zahl

ASCII-American Standard for Coded Information Interchange

- Klein- und Grossbuchstaben, Ziffern + Sonderzeichen
- Codierung in 1 Byte => 256 Zeichen Möglich

ASCII-Tabelle:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00	□ 01	□ 02	□ 03	□ 04	□ 05	□ 06	□ 07	□ 08	09	0A	□ 0B	□ 0C	0D	□ 0E	□ 0F
16	□ 10	□ 11	□ 12	□ 13	□ 14	□ 15	□ 16	□ 17	□ 18	□ 19	□ 1A	□ 1B	□ 1C	□ 1D	□ 1E	□ 1F
32	20	! 21	" 22	# 23	$ 24	% 25	& 26	' 27	(28	) 29	* 2A	+ 2B	, 2C	- 2D	. 2E	/ 2F
48	0 30	1 31	2 32	3 33	4 34	5 35	6 36	7 37	8 38	9 39	: 3A	; 3B	< 3C	= 3D	> 3E	? 3F
64	@ 40	A 41	B 42	C 43	D 44	E 45	F 46	G 47	H 48	I 49	J 4A	K 4B	L 4C	M 4D	N 4E	O 4F
80	P 50	Q 51	R 52	S 53	T 54	U 55	V 56	W 57	X 58	Y 59	Z 5A	[5B	\ 5C	] 5D	^ 5E	_ 5F
96	` 60	a 61	b 62	c 63	d 64	e 65	f 66	g 67	h 68	i 69	j 6A	k 6B	l 6C	m 6D	n 6E	o 6F
112	p 70	q 71	r 72	s 73	t 74	u 75	v 76	w 77	x 78	y 79	z 7A	{ 7B	| 7C	} 7D	~ 7E	□ 7F
128	€ 80	? 81	‚ 82	ƒ 83	„ 84	… 85	† 86	‡ 87	ˆ 88	‰ 89	Š 8A	‹ 8B	Œ 8C	? 8D	Ž 8E	? 8F
144	? 90	‘ 91	’ 92	“ 93	” 94	• 95	– 96	— 97	˜ 98	™ 99	š 9A	› 9B	œ 9C	? 9D	ž 9E	Ÿ 9F
160	A0	¡ A1	¢ A2	£ A3	¤ A4	¥ A5	¦ A6	§ A7	¨ A8	© A9	ª AA	« AB	¬ AC	AD	® AE	¯ AF
176	° B0	± B1	² B2	³ B3	´ B4	µ B5	¶ B6	· B7	¸ B8	¹ B9	º BA	» BB	¼ BC	½ BD	¾ BE	¿ BF
192	À C0	Á C1	Â C2	Ã C3	Ä C4	Å C5	Æ C6	Ç C7	È C8	É C9	Ê CA	Ë CB	Ì CC	Í CD	Î CE	Ï CF
208	Ð D0	Ñ D1	Ò D2	Ó D3	Ô D4	Õ D5	Ö D6	× D7	Ø D8	Ù D9	Ú DA	Û DB	Ü DC	Ý DD	Þ DE	ß DF
224	à E0	á E1	â E2	ã E3	ä E4	å E5	æ E6	ç E7	è E8	é E9	ê EA	ë EB	ì EC	í ED	î EE	ï EF
240	ð F0	ñ F1	ò F2	ó F3	ô F4	õ F5	ö F6	÷ F7	ø F8	ù F9	ú FA	û FB	ü FC	ý FD	þ FE	ÿ FF

16					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
					0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
					0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
					0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
					0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	0	0	0	1	16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	QS	RS	US
2	0	0	1	0	32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	0	1	1	48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	0	1	0	0	64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	0	1	0	1	80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	0	1	1	0	96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	0	1	1	1	112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8	1	0	0	0	128															
9	1	0	0	1	144															
A	1	0	1	0	160															
B	1	0	1	1	176															
C	1	1	0	0	192															
D	1	1	0	1	208															
E	1	1	1	0	224															
F	1	1	1	1	240															

Abbildung 8: ASCII mit bin. / dez. / hex.-Notationen (Fischer, 2001)

A ≈ 65_{10} ≈ $0100\ 0001_2$ ≈ 41_{16}

6.1 Unicode

- Unicode ist ein internationaler Standard, in dem langfristig für jedes sinntragende Schriftzeichen oder Textelement aller bekannten Schriftkulturen und Zeichensysteme ein digitaler Code festgelegt wird.

- Ziel ist es, die Verwendung unterschiedlicher und inkompatibler Kodierungen in verschiedenen Ländern oder Kulturkreisen zu beseitigen. U

- Unicode wird ständig um Zeichen weiterer Schriftsysteme ergänzt (heute in der Version 6.2)
- Die Nummerierung ist hexadezimal in der Schreibweise U+XXXXXXXX

- z.B. Beispiel: U+00B6 ist das "Pilcrow-Zeichen", wie wir es aus Word kennen: "¶".
- Üblich sind eine 4-/6- oder 8-stellige Hex-Schreibung (16/24/32 Bits) – Obiges ist ein Beispiel

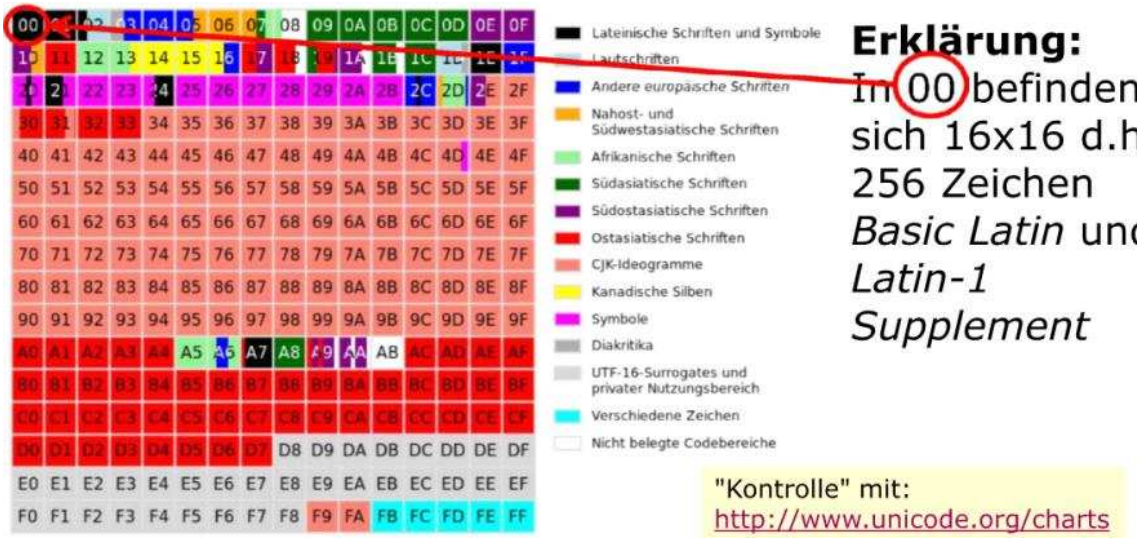
  für 4 Stellen.

- Führende Nullen können in dieser Schreibung paarweise weggelassen werden.

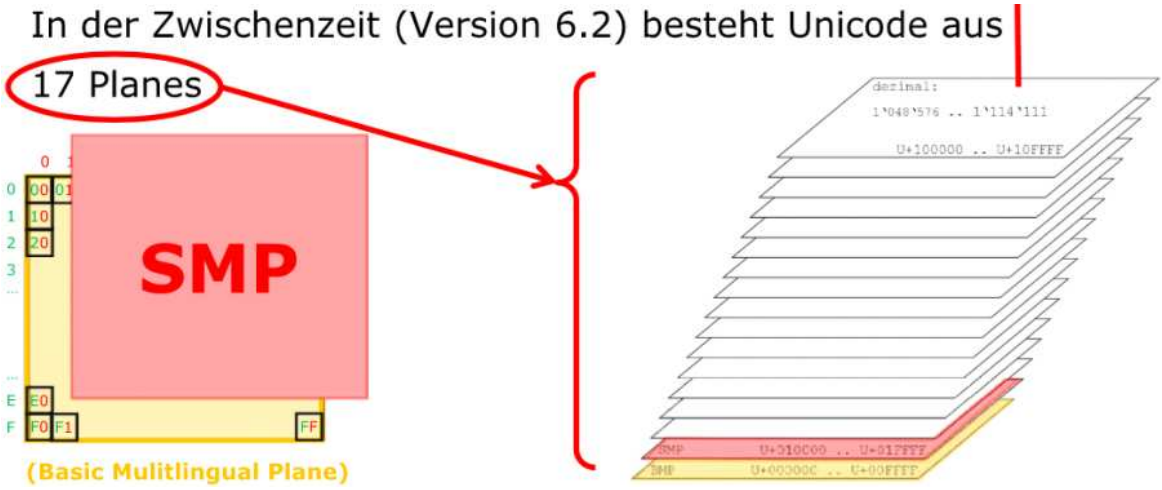
- Die Identifikation durch eine Bezeichnung ist definiert und kann übersetzt werden

6.1.1 Struktur

Die Zeichen des Unicode sind in so genannten Planes organisiert. Eine Plane ist eine quadratische Tabelle von 256 Zeilen zu 256 Spalten, also 216 = 65'536 Feldern.
Dies sind die Code Points und werden innerhalb der Plane fortlaufend nummeriert.
Unicode war ehemals ein 16-Bit Code, bestand also aus einer einzigen Plane.
Inzwischen, mit Version 5.0.0 vom Juli 2006, sind weitere 16 Planes dazugekommen!
Aufbau der BMP (Basic Multilingual Plane)



Aufbau SMP (Supplementary Multilingual Plane) -> Unicode wächst jedes Jahr um abertausende Zeichen.



6.1.2 UTF – Unicode Transformation Formate

UTF ist also eine Methode, Unicode-Zeichen auf Folgen von Bits abzubilden.
Für die Repräsentation der Unicode-Zeichen zum Zweck der elektronischen Datenverarbeitung gibt es verschiedene UTFs.

- UTF-8 empfiehlt sich für Texte mit vorwiegend lateinischen Buchstaben.
- UTF-16 empfiehlt sich für Texte mit z.B. asiatischen Zeichen.
- UTF-32 empfiehlt sich künftig für höchste Performanz (bei grösstem Speicherbedarf).

Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Abbildung 14: Umrechnung von Codepunkt in UTF-8 (http://www.ietf.org/rfc/rfc3629.txt)

UTF-8 muss gemäss den Internet-Behörden (IETF, IANA, Internet Mail Consortium, IMC) künftig durch alle Internet-Protokolle berücksichtigt werden. IANA gibt die Schreibweise als UTF-8 vor - wie auch die exakte Schreibweise der anderen Formate (siehe Abbildung weiter unten). Die RFC 3629 legt die Codierung und die zugehörigen Algorithmen fest.

Buchstabe *y*	U+0079	00000000 01111001	01111001	0x79
Buchstabe *ä*	U+00E4	00000000 11100100	11000011 10100100	0xC3 0xA4
Zeichen für eingetragene Marke ®	U+00AE	00000000 10101110	11000010 10101110	0xC2 0xAE
Euro-Zeichen €	U+20AC	00100000 10101100	11100010 10000010 10101100	0xE2 0x82 0xAC

7 Big und Little Endian

- Womit arbeiten MAC / Windows?

OS	Endian	Was ist besser
Windows	Little Endian	Also, because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision math routines are correspondingly easy to write.
Linux	Big Endian	The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.

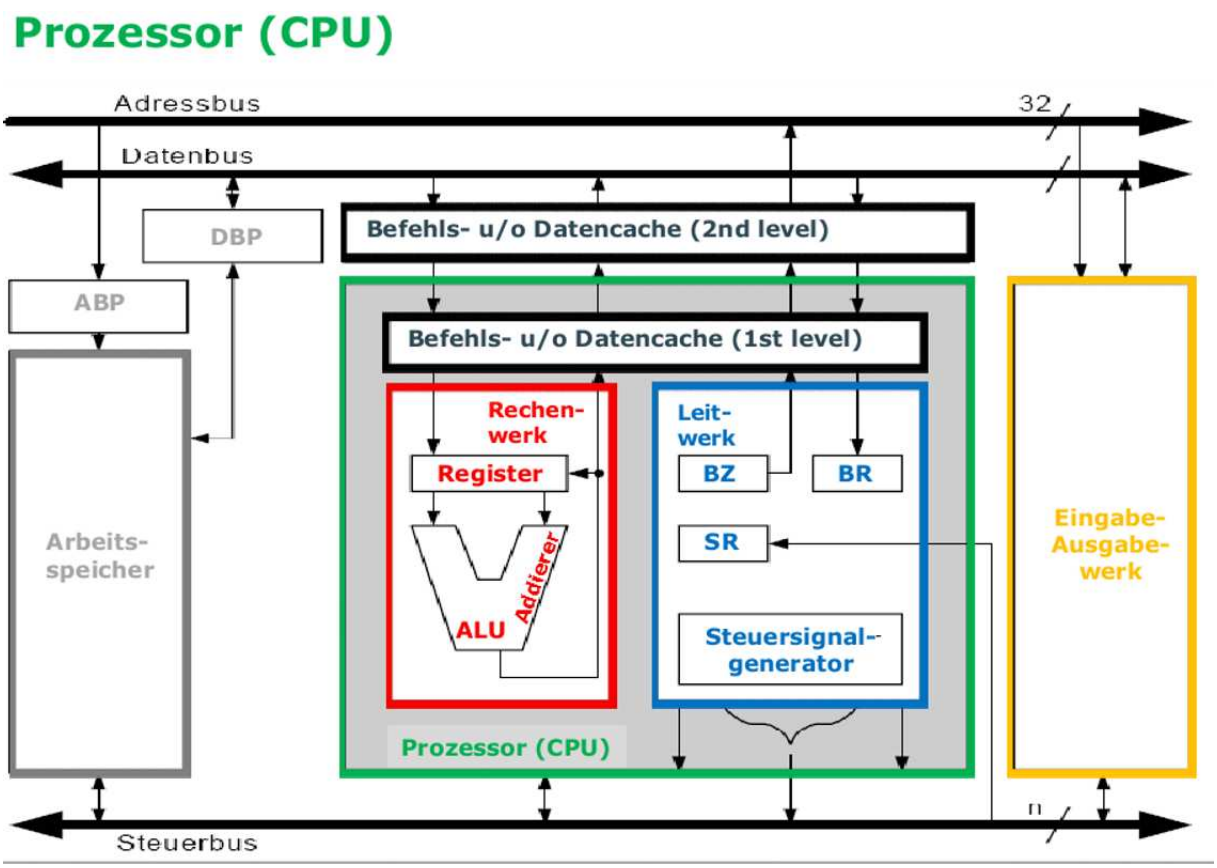
- Wofür braucht man BOM (Byte Order Mark)?
  Hint: http://www.unicode.org/faq/utf_bom.html
  - Gibt die verwendete Codierung an.

- Was ist das Unicode Consortium?
  - Zielsetzung
    - Entwicklung eines weltweit gültigen Zeichensatzes für alle lebenden und toten Sprachen.
  - Seit wann?
    - 3. Januar 1991
  - Wo wurde es entwickelt?
    - Sitz in Kalifornien, bestehend aus den Big Playern der Branche

8 Architektur Mikropozessor

Processor besteht immer aus einer CPU (central processing unit) und ALU (Arithmetik locigal Unit) heut auch FPU (Floating Point Unit)

64bit bezieht sich nur auf die Bandbreite des Datenbuses



8.1 Rechenwerk und Cache

ALU

- Arithmetic Ligical Unit
- Kann nur addieren
- Trifft logische Enscheide (XOR)
- FPU = Floating Point Unit
- Moderne Prozessoren bestehen aus mehreren ALU's und FPU's
- Register = Prozessor eigenen 64bit Speicherbereiche

8.2 Leitwerk und Steuerwerk

- BZ = Befehlszähler (Counter)
  Zeigt auf den nächsten Befehl

- BR = Befehlsregister
  Holt Befehl aus Katalog
- SR = Statusregister
  Status der Hardware

8.3 Arbeitsspeicher

- Aufbau wie ein Hochhaus
- Daten werden von oben nach unten gespeichert
- Programmcode wird von unten nach oben gespeichert

8.4 Bus

Parallele Leistungssystem

- Es gibt Adress-, Daten- und Steuerbus
- Heutige Busse sind 64 Bit
- Der Holzi hat ein nur ein Steuerbus für Read und Write (minimal)

Busstypen:

- Adressbus - Verantwortlich für die Geschwindigkeit, überträgt Adressen (i7 auf 36 Leitungen)
- Datenbus - Verantwortlich für das Speichervolumen, überträgt Daten (i7 auf 64 Leitungen)

9 Hardware-Komponenten

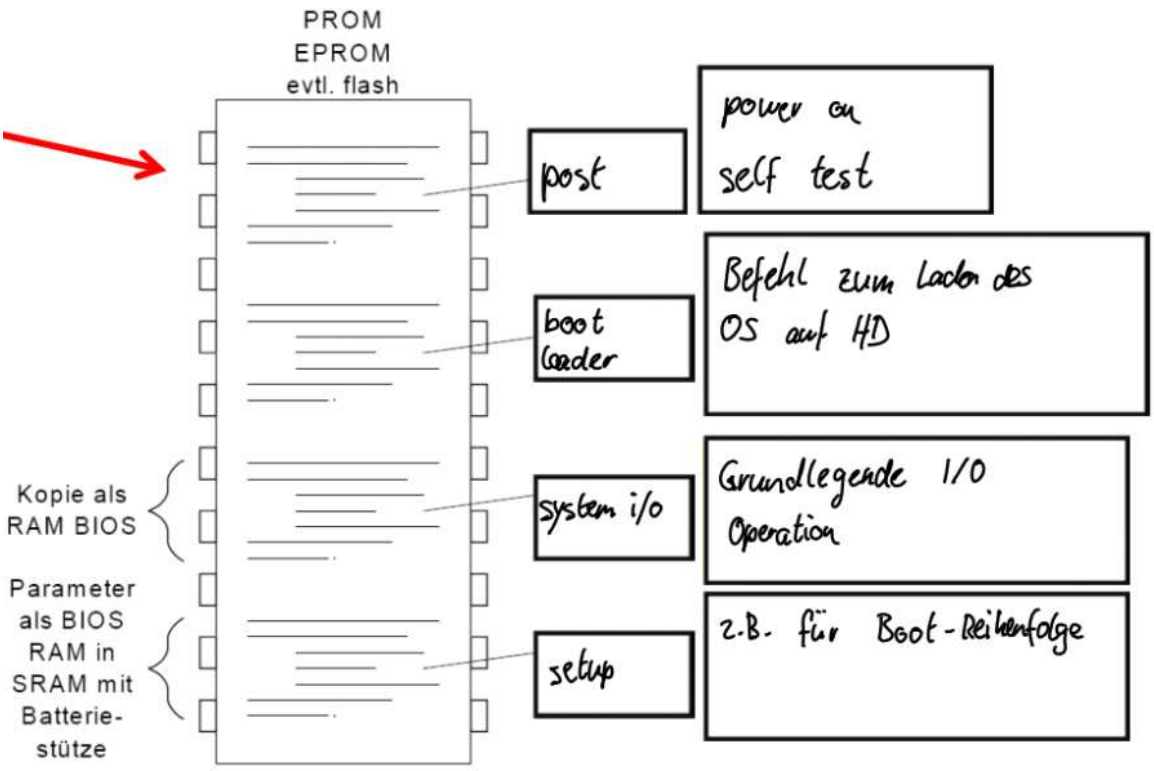
9.1 BIOS

Ein Chip der die Computer Firmware enthält.

Neuerdings heissen diese:

- EFI = Extensible Firmware Interface
- UEFI = Universal EFI (ab 1998 von Intel)

Architektur des EPROM/PROM Speichers, welcher diese Komponente enthält:



10 Codegewinnung

10.1 Diskreditisierung

Bei der Diskretisierung wird in verschiedenen zeitlichen Abstand anhand einer Referenztabelle ein Signal digitalisiert.

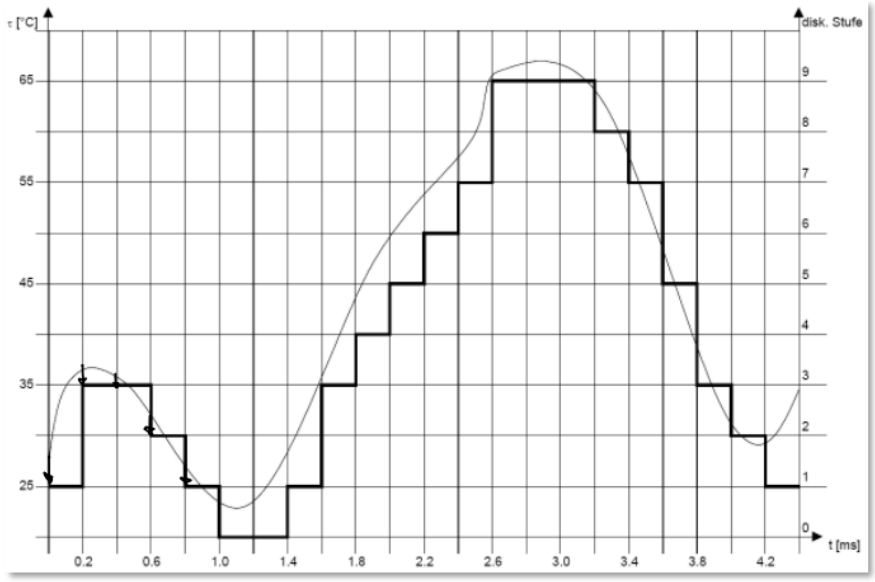


Abbildung 28: Diskretisierung eines kontinuierlichen Signals

Immer abrunden

diskrete Stufe	binäres Wort
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Abbildung 29: Digitalisierungstabelle

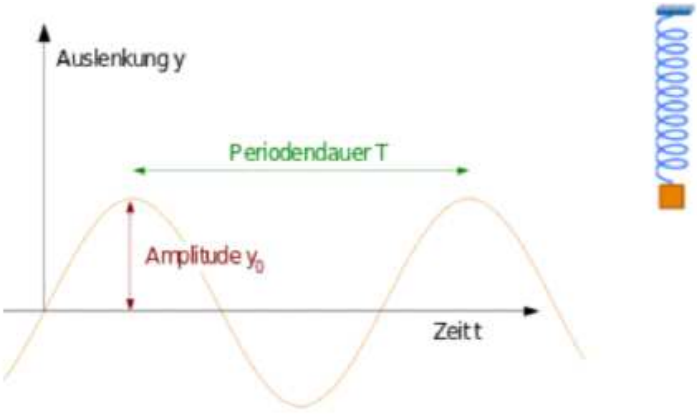
Zeitpunkt	*0.0 ms*	*0.2 ms*	*0.4 ms*	*0.6 ms*	*0.8 ms*	*1.0 ms*	*1.2 ms*	*1.4 ms*	*1.6 ms*	*1.8 ms*	*2.0 ms*
diskreter Wert	1	3	3	2	1	0	0	1	3	4	5
binärer Wert	0001	0011	0011	0010	0001	0000	0000	0001	0011	0100	0101

Abbildung 30: Übertragungsrate als Funktion von Abtastfrequenz *und* Auflösung

10.2 Modulierung

Beschreibt einen Vorgang der in der Nachrichtentechnik zur Übertragung eines Nutzsignals. Dabei veränder das Nutzsignal seinen Träger.

10.2.1 Harmonische Schwingung

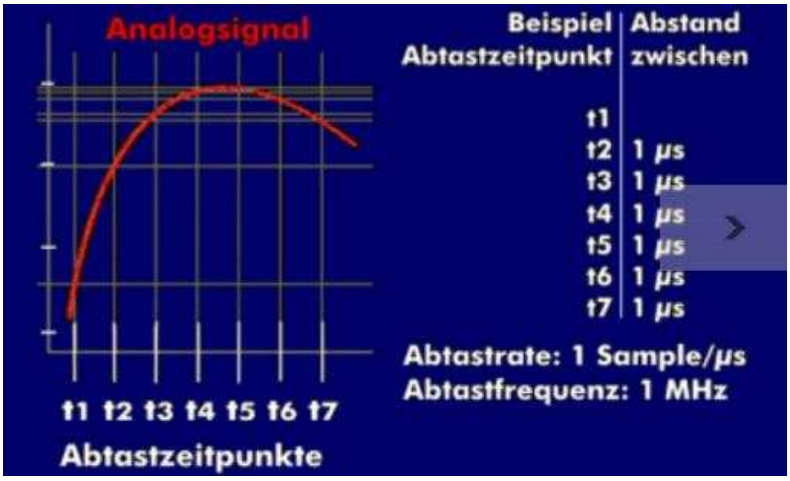


- Der Kehrwert der Periodendauer *T* ist die **Frequenz f**, also: **f = 1/T**
- Einheit der Frequenz ist das **Hertz** (**1 Hz = 1 s^{-1} = 1 * 1/s**)
  in Worten:
  1 Hz = 1 Schwingung pro Sek.
  = 1 Sample pro Sek.

10.2.2 Abtastrate

Beschreibt die Frequenz mit der ein analoges Signal gemessen wird.

Samples per Second S/s



Eine ergänzende Methode ist **sample and hold**

Dabei wird bei einer Entnahme einer Signalprobe aus einer analogen Spannung der Abgetastete Spannungswert für einen bestimmten Zeitraum gehalten.

Die Digitalisierung von analogen Signalen ist mit AD-Wandlern möglich.

11 Codierung

Ziel der Codierung ist es Daten mit grösstmöglicher Qualität zu übertragen.

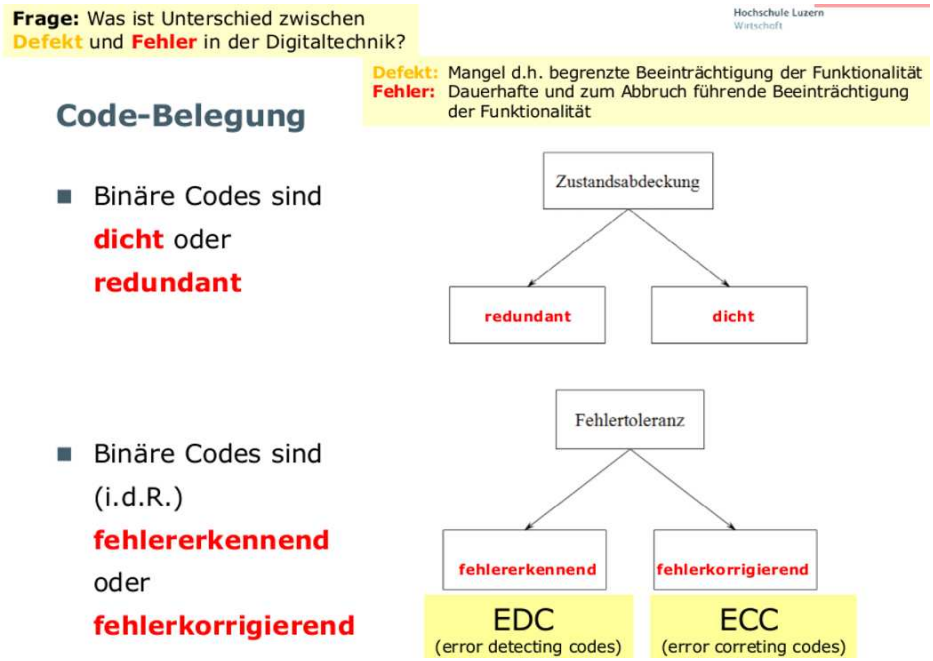
11.1 Redundanz

Kanalcodierung bezeichnet man in der Nachrichtentechnik das Verfahren digitale Daten bei der Übertragung über gestörte Kanäle mit Redundanz zu schützen.

+ Daten sind geschützt
- Es braucht mehr Bandbreite

11.2 Codesicherung

Transportkomponenten haben und machen Fehler, meist in Form der Invertierung einzelner Bits.



Defekt = Mangel mehrfach versuchen jedoch Erfolg
Fehler = Abbruch

11.3 Error dedecting Code

Beispiel einer Korrektur mit mehrere Parity-Bits

Codierung von „par“ mit ASCII:

p: 0111 0000 > even 1
a: 0110 0001 > even 1
r: 0111 0010 > even 0

Mit dem Parity-Even wird ein zusätzliches Bit mit der Information ob Quersumme an erster Stelle ungerade ist oder nicht.

Die Fehlerkorrektur hat immer eine Nachfrage zurfolge, d.h. der Datenblock muss nochmals übertragen werden.

11.4 Error correcting Code

Zwei Arten von Codes, Block-Code und Faltungs-Code.

Ein Blockcode ist eine Art von Kanalcodierung, wobei alle benutzten Codewörter diselbe Anzahl an Symbolen (z.B. 16 Bit) aus einem Alphabet (z.B. {0,1} haben).

12 Masseinheit für Bytes

Maßeinheiten für Bytes

Maßeinheit		Anzahl von Bytes	KBytes	MBytes
Byte		1		
Kilobyte (KByte)	2^{10}	1024	1	
Megabyte (MByte)	2^{20}	1.048.576	1024	1
Gigabyte (GByte)	2^{30}	1.073.741.824	1.048.576	1024
Terabyte (TByte)	2^{40}	1.099.511.627.776	1.073.741.824	1.048.576
Petabyte (PByte)	2^{50}	1.125.899.906.842.624	1.099.511.627.776	1.073.741.824
Exabyte (EByte)	2^{60}	1.152.921.504.606.846.976	1.125.899.906.842.624	1.099.511.627.776

13 Fehlerkorrektur

Bei der Übertragung können Signale ungenauer, verzerrt oder verzögert werden.

13.1 Fehlerursachen

Rauschen

In einem Medium können die Leitmaterialien durch Verschmutzung zur Verzerrung des Signals führen.

Kurzzeitstörung

Elektronischer Funken oder Kratzer auf CDs die kurzzeitig die Spannung beinflussen

Signalverformung

Das Übertragungssignal wird verformt

Nebensprechen

Benachbarte Digitalkanäle haben Einfluss auf das zu übertragende Signal z.B. durch kapazitive Kopplung.

13.2 Fehlerarten

Einzelbitfehler

Treten unabhängig von anderen auf.

Bündelfehler

Treten abhängig von anderen Fehler auf >> Folgefehler.

Synchronisationsfehler

Ist ein längerer Bündelfehler bei dem neben dem Verlust des Inhalts auch die Information verloren geht wieviele Symbole verloren gegangen sind.
Dies hat zur Folge, dass auch die folgenden Informationen nicht mehr korrekt gelesen werden können.

13.3 Fehlerkorrektur-Codes (ECC)

- ***Annäherung 1*: ECC auf dem Stern Duoland**
- Das Alphabet auf Duoland kennt die Buchstaben j und n.
- Zwecks Fehlerkorrektur wird ein **Drei-Bit-Code** fürs Kommunizieren im Duonet vereinbart:
- j = 111, n = 000
- Bei der Übermittlung können 1-Bit-Fehler vorkommen.

Empfangen	Gesendet
000	000
001	000
010	000
100	000
011	111
101	111
110	111
111	111

Beispiel einer einfachen 1-Bit-ECC-Codierung

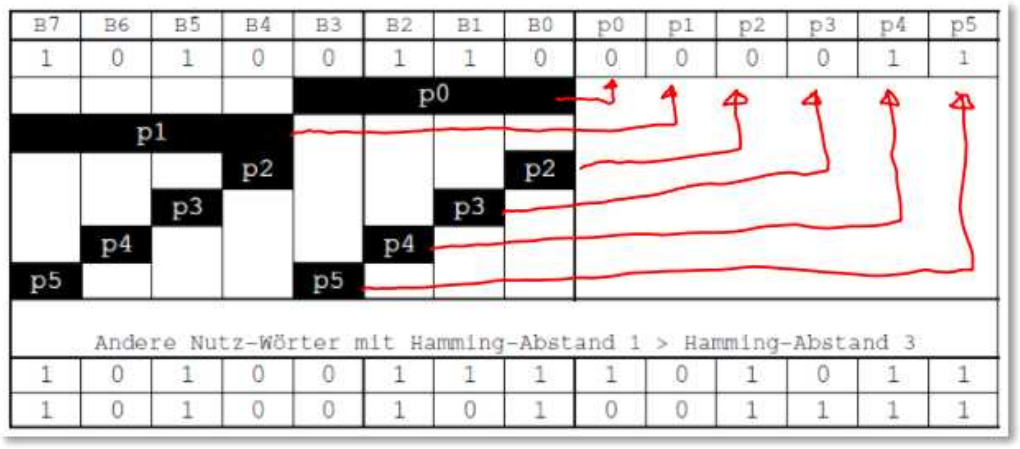


Abbildung 37: Illustration der Idee der fehlerkorrigierenden Codierung (www.pfischer.doz.fhz.ch)

Jedes Bit wird doppelt kontrolliert.

13.4 ECC-Parity-Prüfung

13.4.1 Eindimensionale Prüfung

Ziel ist es aus einem "Nicht-fehlererkennenden" code einen 1-fehlererkennenden Code zu machen.

Die Umsetzung erfolgt mit dem Hinzufügen von einem zusätlichen Parity-Bit.

Für dieses Bit filt die even parity Regel:

```
Anzahl 1 in Codewort + Parity-Bit = Gerade
```

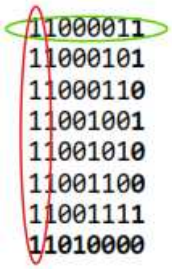
Beispiel mit einem 7 Bit Codewort:

```
1001 0000
1001 0011
```

13.4.2 Zweidimensionale Prüfung

Bei der Zweidimensionalen Prüfung werden ganze Code-Blöcke überprüft.

Dabei wird mit der gleichen parity Regel für jede Zeile eine Parity-Bit gesetzt und in der SchlussZeile Wird für jede Spalte ein Bit gesetzt.



Mit dieser Art von Prüfung können nun die Fehlerhaften Bits gefunden werden.
Codes mit zweidimensionaler Parity-Prüfung haben mindestens den Hammingabstand 3.

13.4.3 K aus n Code

n ist die länge des Bitwortes
k ist die Anzahl gesetzten (1) Bits im Codewort

00011 >> n=5, k=2

13.4.4 Hammingabstand eines Codes

Das Hamminggewicht entspricht dem k.

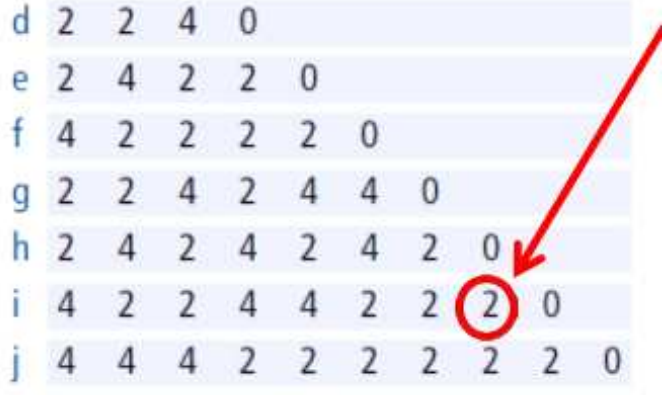
Der Hammingabstand entspricht den Anzahl von Binärstellen an denen sich zwei Bitwörter unterscheiden.

Beispiel mit 2 aus 5 Code:

c 00110
g 10**001**

>> Hammingabstand ist 4
Mit den Codewörtern {a, …, j} kann man eine Tabelle erstellen:

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



Der Hammingabstand d dieses Codes ist der kleinste auftretende Abstand, sprich 2.

Es können Fehler erkannt werden, die weniger als d Bits betreffen (<2).

Es können Fehler korrigiert werden, die weniger als d/2 Bits betreffen (<2/2)

>> Der 2 aus 5 Code kann 1 Fehler erkennen.
>> Der Code kan keine Korrektur durchführen.

14 Hamming Codes

Der einfachste Hamming-Code ist ein (7,4)-Code, der eine Länge von 7 Bits hat, wovon allerdings nur 4 Bits
Nutzinformationen sind und die restlichen 3 Bits zur Fehlerkorrektur dienen.
Dieser Hamming-Code ist ein 1-fehlerkorrigierender Code mit einem Hammingabstand von 3.

14.1 Berechnung

Im Allgemeinen gilt:

Es gibt Hamming-Codes der Länge:

2r-1

Sofern folgendes gilt:

r>=2

Anzahl Korrekturbits:

r

Informationsbits:

2r-1-r

Beispiel:

10011010

Parity Bit Positions:

1,2,4,8,16,32,64 etc.

Leave Space for PB :

_ _ 1 _ 001 _ 1010

- Position 1 checks bits 1,3,5,7,9,11:
  ? _ **1** _ **0** 0 **1** _ **1** 0 **1** 0. Even parity so set position 1 to a
  0: **0** _ **1** _ **0** 0 **1** _ **1** 0 **1** 0
- Position 2 checks bits 2,3,6,7,10,11:
  0 **?** **1** _ 0 **0** **1** _ 1 **0** **1** 0. Odd parity so set position 2 to a 1: 0 **1** **1** _ 0 **0** **1** _ 1 **0** **1** 0
- Position 4 checks bits 4,5,6,7,12:
  0 1 1 **?** **0** **0** **1** _ 1 0 1 **0**. Odd parity so set position 4 to a 1: 0 1 1 **1** **0** **0** **1** _ 1 0 1 **0**
- Position 8 checks bits 8,9,10,11,12:
  0 1 1 1 0 0 1 **?** **1** **0** **1** **0**. Even parity so set position 8 to a 0: 0 1 1 1 0 0 1 **0** **1** **0** **1** **0**
- Code word: 011100101010.

14.2 Fixing

- Received: 011100101110 → Nach Prüfung → Fehler (Parity Bit nicht even) an Position 2 + 8 = 10 → Fehler an der Position 10

Beweiss:

Code-Wort: 011100101010 (So wäre es richtig)

Received : 011100101110 (Falsches bit)

15 CRC-Kodierung

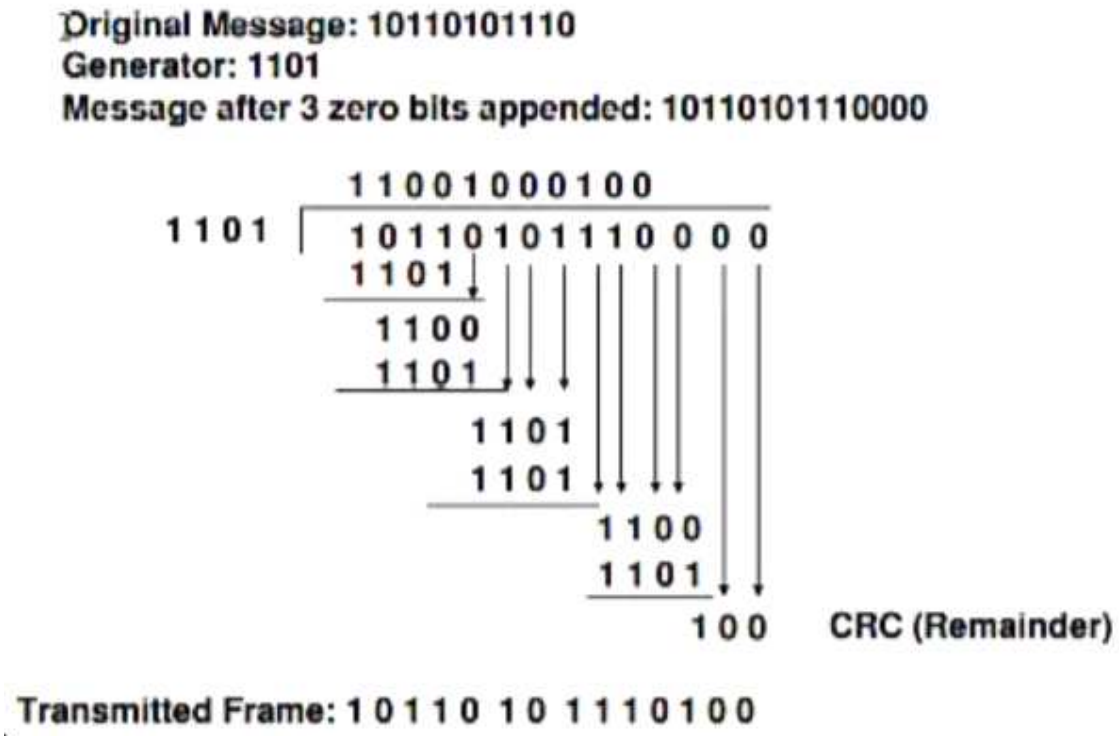
- CRC erkennt Fehler bei der Übertragung oder Speicherung
- CRC kommst gerade beim Ethernet Standard zum Einsatz.
- Die Berechnung des CRC Werts beruht auf Polynomdivision.

  Beispiel:
  https://www.youtube.com/watch?v=MSAog5MEhrs

Als erstes wird ein neues Codewort durch Polynominal:

10111
X^4 + x^2 + x^1 + x^0

- CRC = 4 (höchstes Polinomgrad)
  - An Originalmessage werden also 4 Bit angehängt.



16 Booleschen Algebra

In der Booleschen Algebra werden mit logischen Operatoren gerechnet.

OR, AND und NOT

Wahrheitstabelle **OR**:

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

Symbol **OR**:

Wahrheitstabelle **AND**:

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

Symbol **AND**:

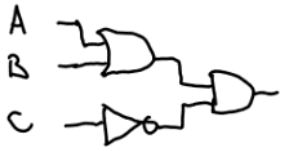
Wahrheitstabelle **NOT**:

a	NOT a
0	1
1	0

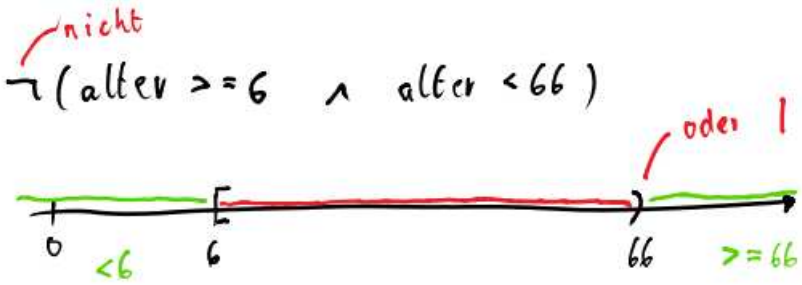
Symbol **NOT**:

Beispiel:

Zeichnen Sie mit Bleistift und Papier und offiziellen Symbolen den booleschen Ausdruck **(A + B)** $\overline{\mathbf{C}}$



Beispiel Und-Audruck:

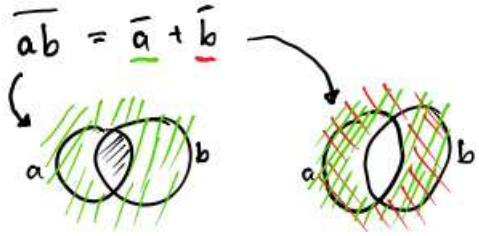


16.1 Wichtige Gesetzte

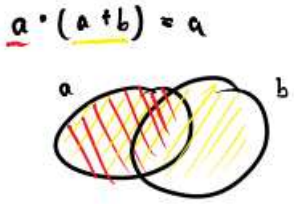
Name	Formel	Kurzschreibweise
Kommutativgesetze	a * b = b * a a + b = b + a	$ab = ba$
Assoziativgesetze	a * (b * c) = (a * b) * c a + (b + c) = (a + b) + c	$a(bc) = (ab)c$
Distributivgesetze	a * (b + c) = (a * b) + (a * c) a + (b * c) = (a + b) * (a + c)	$a(b + c) = ab + ac$ $a + bc = (a + b)(a + c)$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

* = UND
+ = ODER
- = NOT

Beweis De Morgan'sche Gesetzt:



Beweis Verschmelzungsgesetzt:



Wahrheitstabelle

$\overline{a \cdot b} = \bar{a} + \bar{b}$

a	b	$\bar{a}$	$\bar{b}$	$\bar{a} + \bar{b}$	$a \cdot b$	$\overline{a \cdot b}$
0	0	1	1	1	0	1
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	0	1	0

How to real:

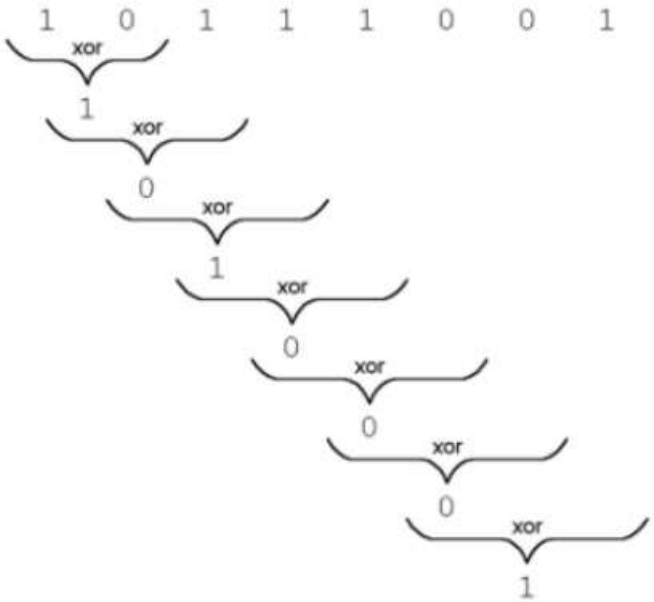
a+(-a+b)=a+b

a+b:
>0111

u:-a*b:	a+u
1100	0011
0101	0100
>0100	**>0111**

16.2 Antivalenz

So bestimmt man die gerade parität eines n-Bitworts.



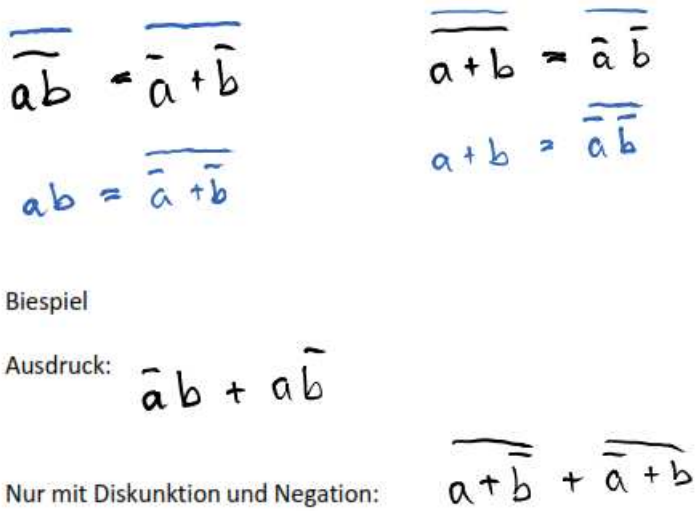
P(n-Wort) = LSB xor P(n-Wort div 2)
P(0) = 0

17 Disjunktive Normalform

Folgende Sätze existieren:
1: Alle zweistelligen booleschen Funktionen können mit Hilfe der Negation (-), der Konjunktion (* und der Diskunktion (+) dargestellt werden.

2: Alle zweistelligen booleschen Funktionen können entweder mit Hilfe der Negation und der Konjunktion, oder mit Hilfe der Negation und der Disjunktion dargestellt werden.

3: Alle zweistelligen booleschen Funktionen können entweder mit Hilfe der NAND-Verknüpfung oder mit Hilfe der NOR-Verknüpfung dargestellt werden.



17.1 Liftsteuerung

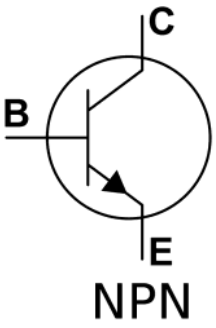
Ein Steuerung für einen Lift der nur im 1 und 4-7 Stock halten soll wird mit einer UND Schaltung gebaut. Der ganze Ausdruck wird dann verkürtzt.

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

$$
\begin{aligned}
s &= \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c + ab\bar{c} + abc \\
&= \bar{a}\bar{b}\bar{c} + a\bar{b}(\bar{c} + c) + ab(\bar{c} + c) \\
&= \bar{a}\bar{b}\bar{c} + a\bar{b} + ab \\
&= \bar{a}\bar{b}\bar{c} + a(\bar{b} + b) \\
&= \bar{a}\bar{b}\bar{c} + a
\end{aligned}
$$

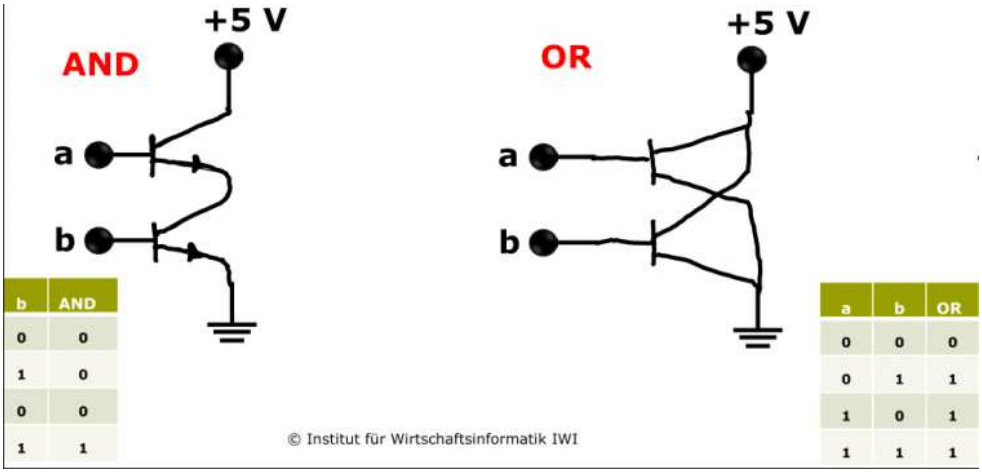
18 Le Transistor

Mittels geschickter Kombination von Transisitoren kann man NOT, AND und OR-Schaltungen erzeugen.
Transistoren sind elektronische Schalter, die zwei Zustände haben: Ein und Aus.



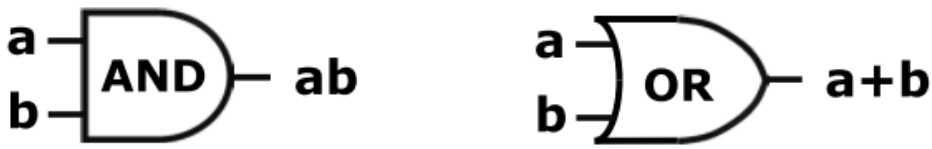
Kleiner Basistrom (B-Basis) kann grossen Stromfluss (C-Collector) steuern. Die Ausgabe (E-Emitter) wird dann an den Verbraucher geleitet.
Das verhältniss von E und B ist die Verstärkung

18.1.1 Logische Schaltung



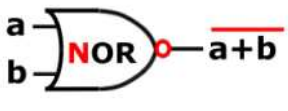
b	AND
0	0
1	0
0	0
1	1

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



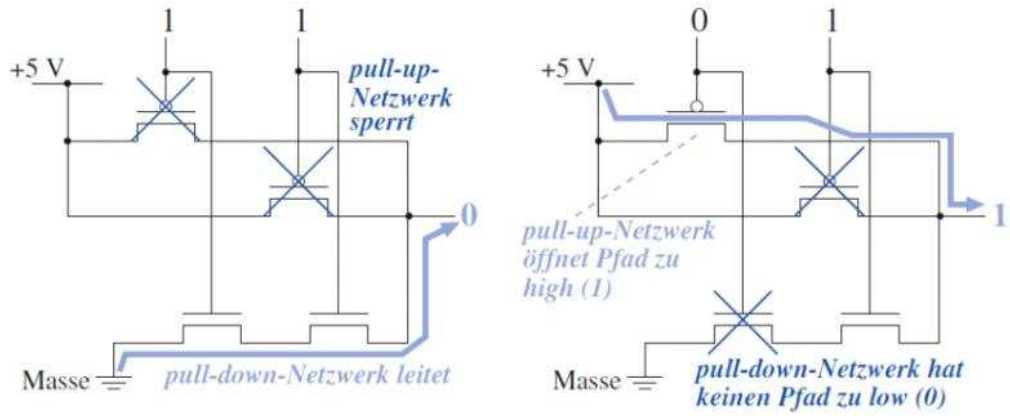
NAND –und NOR-Schaltung

a	b	NAND
0	0	1
0	1	1
1	0	1
1	1	0



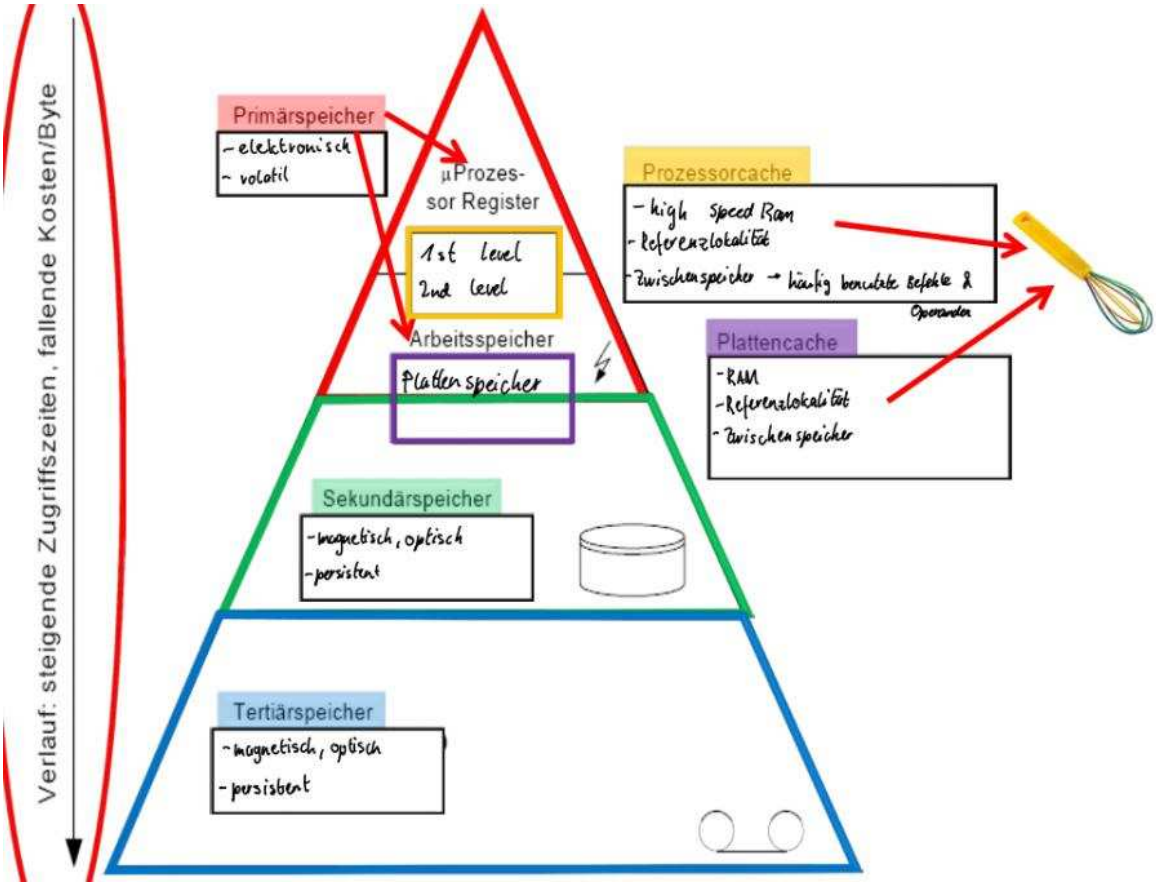
a	b	NOR
0	0	1
0	1	0
1	0	0
1	1	0

18.1.2 Die NAND-Schaltung im Detail

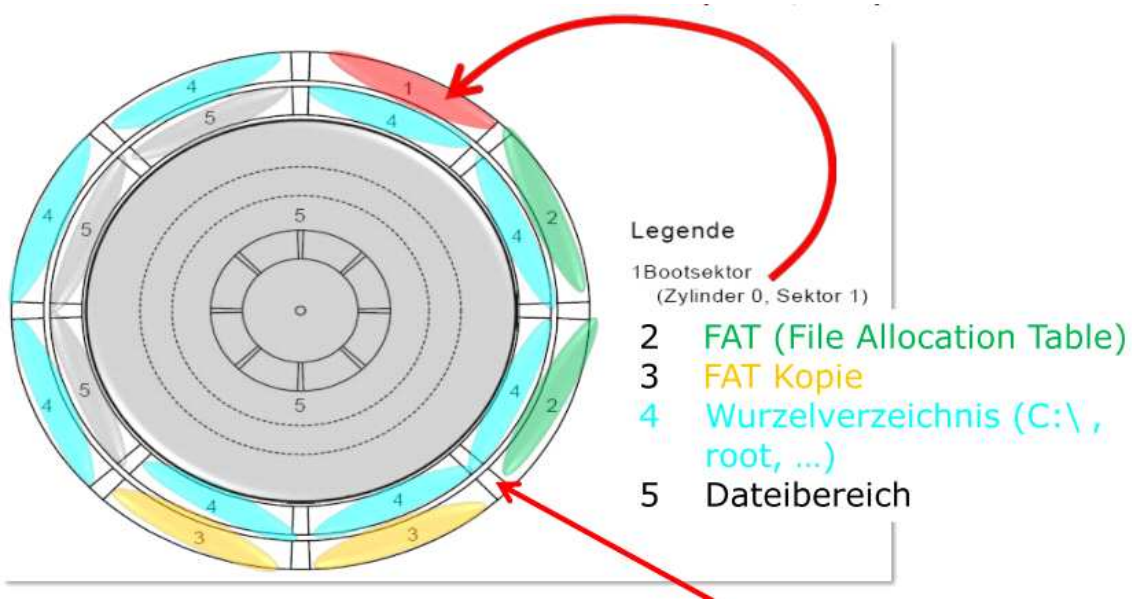


19 Speicher

Die Speicherhierarchi beschreibt den Aufbau von Speicher im Verlauf der Zugriffszeiten.
Dabei gilt: Je längere Zugriffszeiten, desto billiger ist der Speicher.

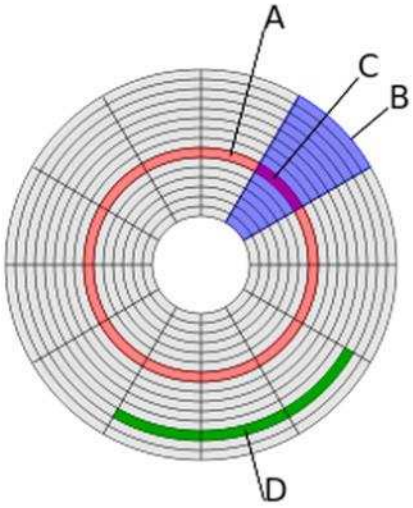


19.1 Aufbau nach Formatierung



19.2 Disk-Aufbau

In computer file systems, a cluster or allocation unit is a unit of disk space allocation for files and directories. To reduce the overhead of managing on-disk data structures, the filesystem does not allocate individual disk sectors by default, but contiguous groups of sectors, called clusters.



Disk structure:
(A) track
(B) geometrical sector
(C) track sector
(D) cluster

Track (Spur): ein Ring auf einer Disk
Cylinder: meherere Disks gestapelt
Zone (geomtrischer Sektor): Ein Auschnitt der Disk
Sektor: Stück von Track in einem geometrischen Sektor
Cluster (sektorenverbund): mehrere Sektoren verbunden
FAT32 -> 228 -> 268'435'456 Sektoren
Cluster Grösse 512 Byte (-32768 Byte -> 31kB)
2^28*32768=8E12
HD Grösse -> 2^28*512=1E11
Je nach Grösse der Datensätze lohnt es ich die Cluster Grösse zu vergrössern oder zu verkleinern.

19.2.1 Formatierung

Schnellformatierung: Löschen der FAT
Volle Formatierung: Löschen aller Daten auf der Disk
Dateilöschung: Adresse zu Datei wird gelöscht
Dateilöschung aus Papierkorb: Datei wird physisch auf Disk gelöscht

- Low Level Formatierung
- initialisiert die Platte magnetisch
- trägt die physikalischen Eigenschaften wie
    - Interleaving-Faktor und○
    - Zone Bit Recording (siehe je Lexikon)
    - oder vergleichbare Technologien auf.
- Sie wird heute durch die Hersteller vorgenommen und kann/sollte vom Anwender nicht mehr verändert werden.

Volle (High Level) Formatierung

- löscht alle Dateieintragungen
- löscht damit auch den Papierkorb
- überprüft die Qualität der Clusters und markiert sie u.U. als schlecht
- nullt alle Status-Eintragungen (ausser Bad Cluster)
- trägt ein neues Dateisystem mit einem leeren Index ein.

Schnellformatierung

- löscht alle Dateieintragungen im Index
- löscht damit auch den Papierkorb
- nullt (selten) alle Statuseintragungen (ausser Bad Cluster)
- trägt (selten) ein neues Dateisystem mit einem leeren Index ein.

Eine Dateilöschung (in den Papierkorb)

- verschiebt die Datei in ein anderes (meist verstecktes) Verzeichnis (recycled, trash, ...)
- ist auf der Ebene des ganzen Dateisystems also eine reine Umbenennung
- gleich benannte Dateien werden dabei (versteckt) umbenannt
- der Papierkorb gehört mal dem ganzen Dateisystem, mal nur einem Laufwerk.

Eine Dateilöschung (aus dem Papierkorb)

- markiert die Datei als gelöscht - oft durch eine Änderung des Dateinamens
- lässt ansonsten alles intakt inkl. der Clusterverkettung
- gibt die Clusters aber frei.

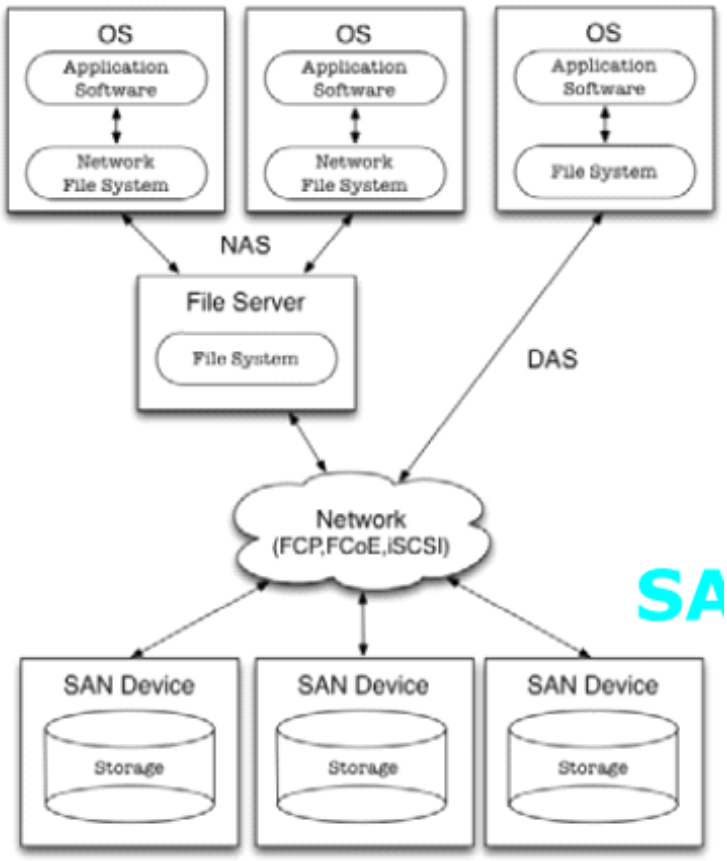
In einer hierarchischen Dateistruktur

- hat das Wurzelverzeichnis einen festen Standort: es ist statisch und hat eine beschränkte Anzahl Einträge
- werden die Unterverzeichnisse als spezielle Dateien im Datenbereich eingetragen: ihre Ablage ist dynamisch und ihre Anzahl lediglich durch die Anzahl möglicher Dateien - also letztlich durch die Anzahl Clusters - beschränkt
- haben Dateien einen partitionsweit einmaligen Namen - den absoluten Dateinamen
- ist eine Dateiverschiebung eine Umbenennung
- ist eine Verschiebung in den Papierkorb eine Umbenennung.

19.2.2 Business Storage

DAS: Direct Attached Storage
NAS: Network Attached Storage
SAN: Sorage Attached Network

19.2.3 Cloud Storage

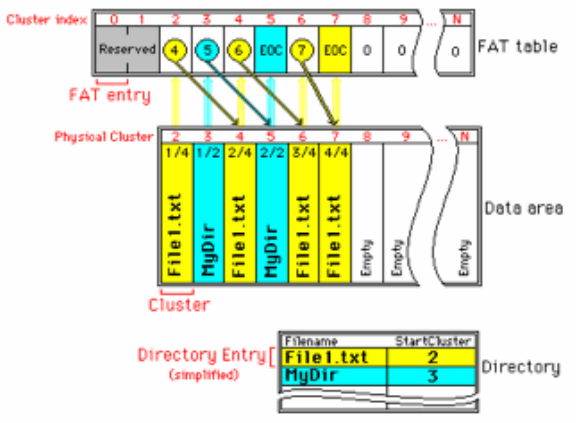


20 Fragmentierung

20.1 Dateiindex

Such nach einer Datei geht wie folgt:

1. Such anhand Dateinamens im Dateiindex
2. Lesen der Attribute und Adresse des ersten Clusters
3. Sprung zum Index und Nachschlagen der Cluster Adress-Kette
4. Cluster-Kette sequenziell auslesen



Der Cluster index kann wie folgt aussehen:

Status	*Bedeutung*
0000	frei
XXXX	Adresse Folge-Cluster
FFF7	Bad Cluster
FFF8	Reserved Cluster
FFFF	End of File

In der FAT table wird angezeigt wo eine Datei im Data area gespeichert ist. Mit EOC wird angezeigt, dass der letzte Cluster der Datei and dieser Speicherstelle liegt.

Beispiel:

script.doc hat die Grösse M
Mc ist die Memorygrösse des Clusters
Dabei gilt:
0 < M <= MC MC < M <= 2*MC 2*MC < M <= 3*MC

	Cluster	*Status*
	…	….
skript.doc ->	**3ABB**	**FFFF**
	3ABC	0000
	3ABD	FFF7
	3ABE	0000
	3ABF	0000
	3AC0	0000
	3AC1	0000
	3AC2	0000
	3AC3	0000
	3AC4	0000
	3AC5	0000
	3AC6	0000
	3AC7	0000
	3AC8	0000
	3AC9	0000

	Cluster	*Status*
	…	….
skript.doc ->	**3ABB**	**3ABC**
	3ABC	**FFFF**
	3ABD	FFF7
	3ABE	0000
	3ABF	0000
	3AC0	0000
	3AC1	0000
	3AC2	0000
	3AC3	0000
	3AC4	0000
	3AC5	0000
	3AC6	0000
	3AC7	0000
	3AC8	0000
	3AC9	0000

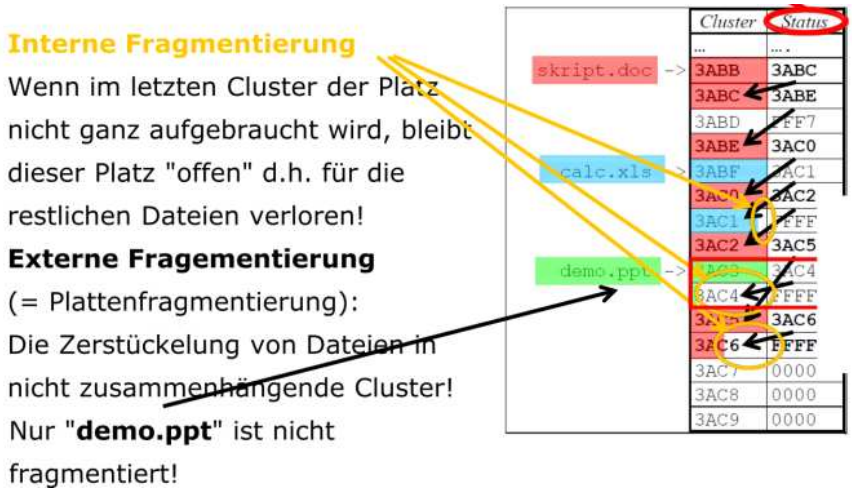
	Cluster	*Status*
	…	….
skript.doc ->	**3ABB**	**3ABC**
	3ABC	**3ABE**
	3ABD	FFF7
	3ABE	**FFFF**
	3ABF	0000
	3AC0	0000
	3AC1	0000
	3AC2	0000
	3AC3	0000
	3AC4	0000
	3AC5	0000
	3AC6	0000
	3AC7	0000
	3AC8	0000
	3AC9	0000

20.2 Fragmentierungstypen

Bei der Speicherung von Dateien werden Cluster fragmentiert, dabei unterscheidet man zwischen zwei Fragmentierungstypen.

Ein Cluster kann nur Informationen aus einer ganzen oder einem Teil einer datei aufnehmen. Bleibt "hinten" Platz frei, ist er verschwendet -> interne Fragmentierung.

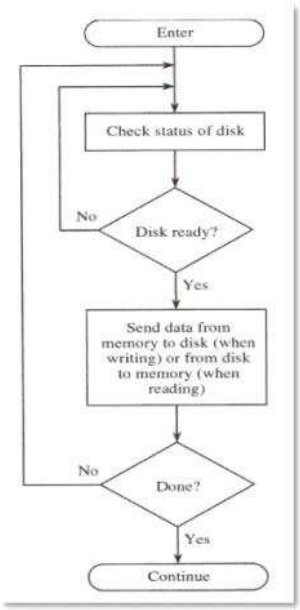
Erstreckt sich eine Datei über mehr als einen Cluster und sind diese nicht zusammenhängend ist das externe Fragmentierung.



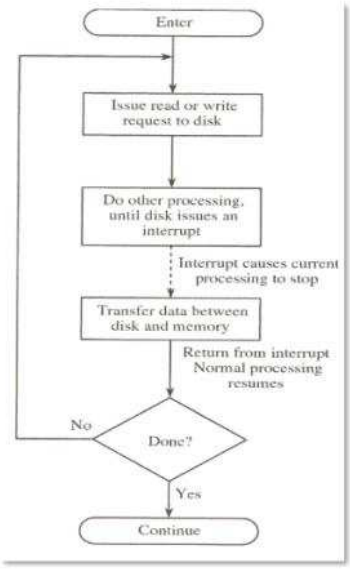
Bei der internen Fragmentierung ergibt sich also Speicherverlust (aus Sicht bytes ja, aus Sicht Cluster nein -> EOF).

21 Polling und Interrupting

Methoden wie 2 Partner kommunizieren können.



Polling (Busiy Waiting, Round Robin)



Interrupt Requesting

22 SSD

When deciding what type of flash to use for an application, it is important to understand the differences between flash technologies. The following document explains the pros and cons of the three types of flash, SLC, MLC and TLC.

SLC- Single Layer Cell

- High performance
- Lower power consumption
- Faster write speeds
- 100,000 program/erase cycles per cell
- Higher cost
- **A good fit for industrial grade devices**, embedded systems, critical applications.

MLC- Multi Layer Cell

- Lower endurance limit than SLC
- 10,000 program/erase cycles per cell
- Lower cost
- **A good fit for consumer products.** Not suggested for applications which require frequent update of data.

TLC- Three Layer Cell

- Higher density•
- Lower endurance limit than MLC and SLC•
- TLC has slower read and write speeds than conventional MLC•
- 5,000 program/erase cycles per cell•
- Best price point•
- **A good fit for low-end basic products.** Not suggested for critical or important applications at this time which require frequent updating of data.

SLC vs. MLC vs. TLC as explained with a glass of water

This glass of water analogy demonstrates how SLC NAND Flash outperforms MLC NAND Flash.

- SLC Flash has only two states: erased (empty) or programmed (full).
- MLC Flash has four states: erased (empty), 1/3, 2/3, and programmed (full).
- TLC Flash has eight states: erased (empty), 1/7, 2/7, 3/7, 4/7, 5/7, 6/7 and programmed (full).

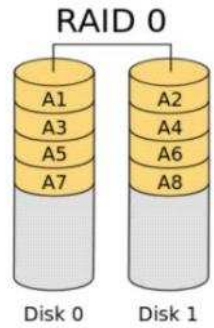
It's easier to read the correct fill status when a glass is either empty or full, as in SLC NAND Flash. When a glass is partially full, as in MLC NAND Flash, the fill status is more difficult to read, taking more time and energy.
Source: http://centon.com/flash-products/chiptype

23 Raid

Heisst: Redundant Array of Independent Disks
Es gibt verschiedene RAID-Levels

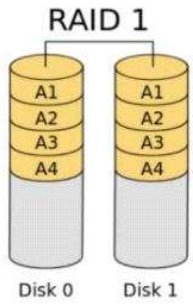
RAID 0 – Striping

- Dateien werden auf verschiedene physische Disk aufgeteilt.
- Keine Redundanz bei Disk-Verlust
- Gute Performance



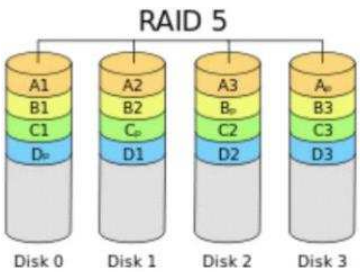
RAID 1 – Mirroring

- Disk werden vollständig dupliziert
- Braucht offensichtlich doppelten Speicherplatz
- Hohe Redundanz



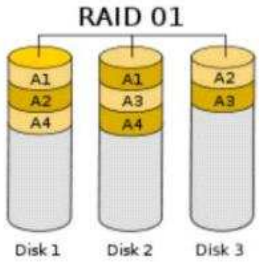
RAID 5 - Striping with Parity

- Vereint beide Ansprüche von 0 und 1
- Mithilfe des Parity-bits kann eine Disk wiederhergestellt werden
- Dateien werden nicht dupliziert, zusätzliche Benutzung durch Parity bit
- Effiziente Disk benutzung
- Parity berechnung kann unperformant sein

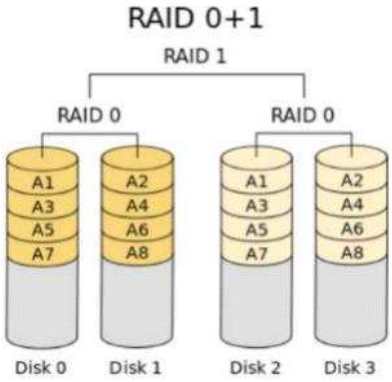


RAID 01

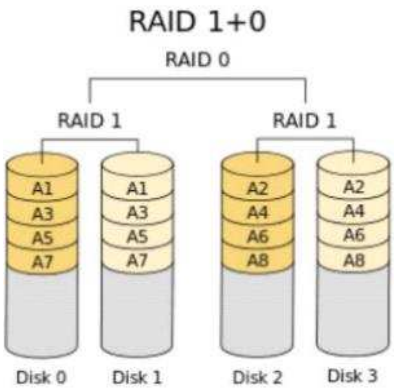
Wird selten benutzt.



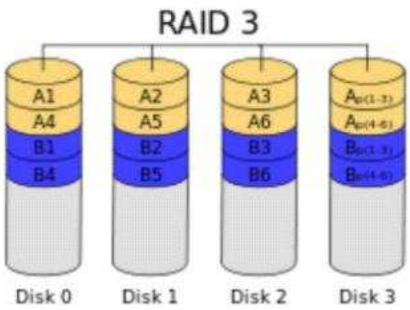
RAID 0+1



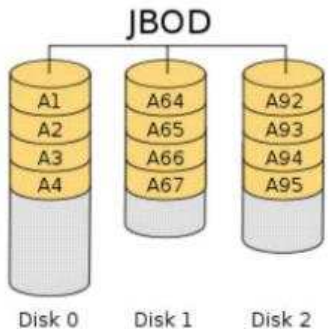
RAID 1+0



RAID 3



JBOD
Just a bound of disk



23.1 RAID Implementation

Software-RAID

- Eine OS-Feature
- Setzt keine spezielle Hardware voraus
- Schlechtere performance als hardware-Raid

Hardware-RAID

- Eine Drive-Controller feature
- Konfiguration unabhängig vom OS -> unsichtbar für OS
- Hohe performance -> designed for speed

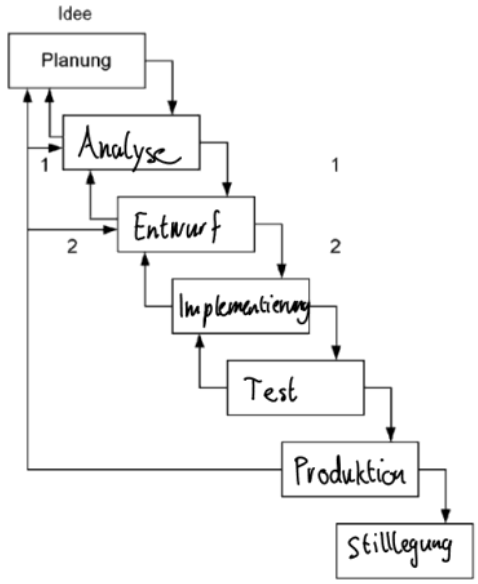
23.1.1 Wechsel

Hot Replacing, Swapping, Plugging -> Wechsel einer Disk im laufenden Betrieb
Host Spare (Sparing) -> Laufendes nicht verwendetes Laufwerk, einsatz bei Ausfall

24 Prozessmodelle

24.1 Schwergewichtige (klassische) Modelle

Wasserfall-Modell



V-Modell

- (ähnlich Wassefall jedoch kongruent v-seitig Testphasen zu jedem Step)

Inkrementelle und iterative Prozessmodelle

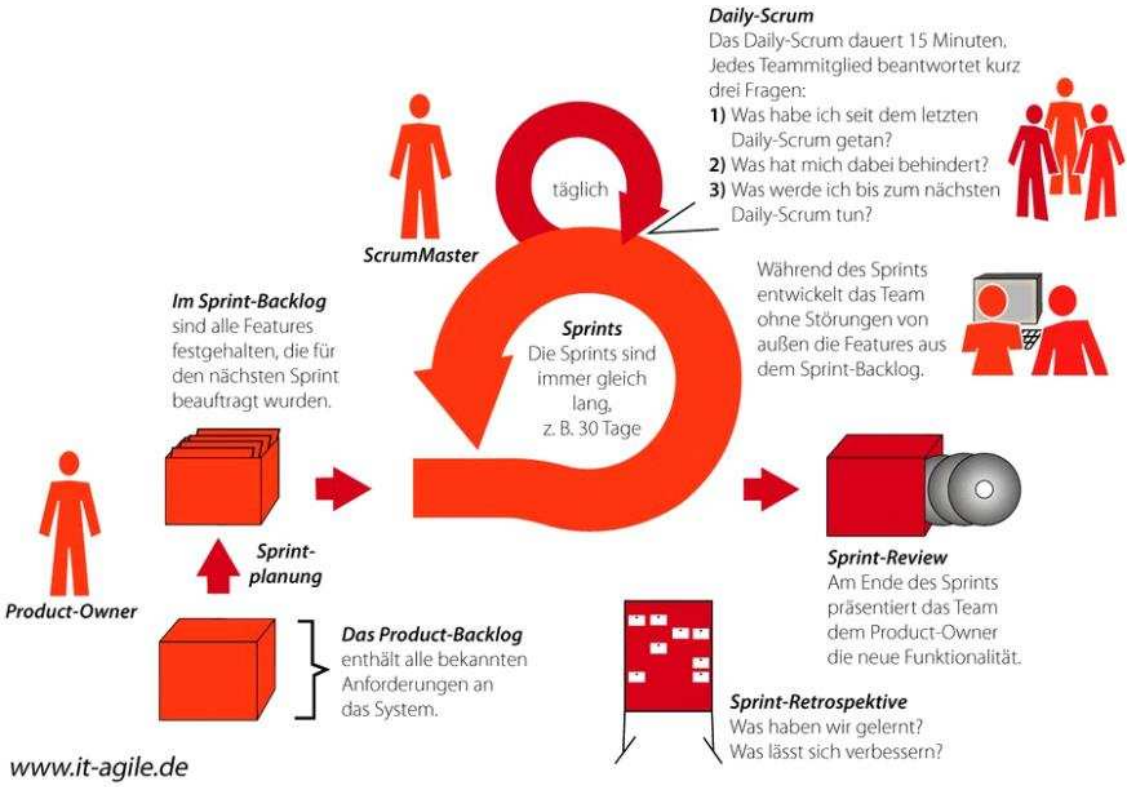
- Spiralmodell (böhm)
- Rational Unified Proces
- Open Unified Process

24.2 Leitgewichtige (agile) Vorgehensmodelle

- Extreme Programming (XP)
  - Programmierung zu zweit während eines Zeitraums -> Fazit: zu streng

24.3 SCRUM

Im Mittelpunkt von Scrum steht das selbstorganisierte Entwicklerteam, das ohne Projektleiter auskommt. Um dem Team eine störungsfreie Arbeit zu ermöglichen, gibt es den ScrumMaster, der als Methodenfachmann dafür sorgt, dass der Entwicklungsprozess nicht zerbricht.
Der ScrumMaster stellt auch die Schnittstelle zum Produktverantwortlichen (Product-Owner) dar, dem die Aufgabe zukommt, Anforderungen zu definieren, zu priorisieren und auch zu tauschen. Allerdings ist in Scrum klar geregelt, wann der Produktverantwortliche neue oder geänderte Anforderungen beauftragen darf - so gibt es ungestörte Entwicklungszyklen von 2-4 Wochen (Sprints), in denen ihm untersagt ist, das Entwicklerteam zu "stören". Während eines Sprints wird deshalb der Product Owner seine Vorstellungen von der weiteren Entwicklung ins Product Backlog eintragen und so für kommende Sprints einplanen.

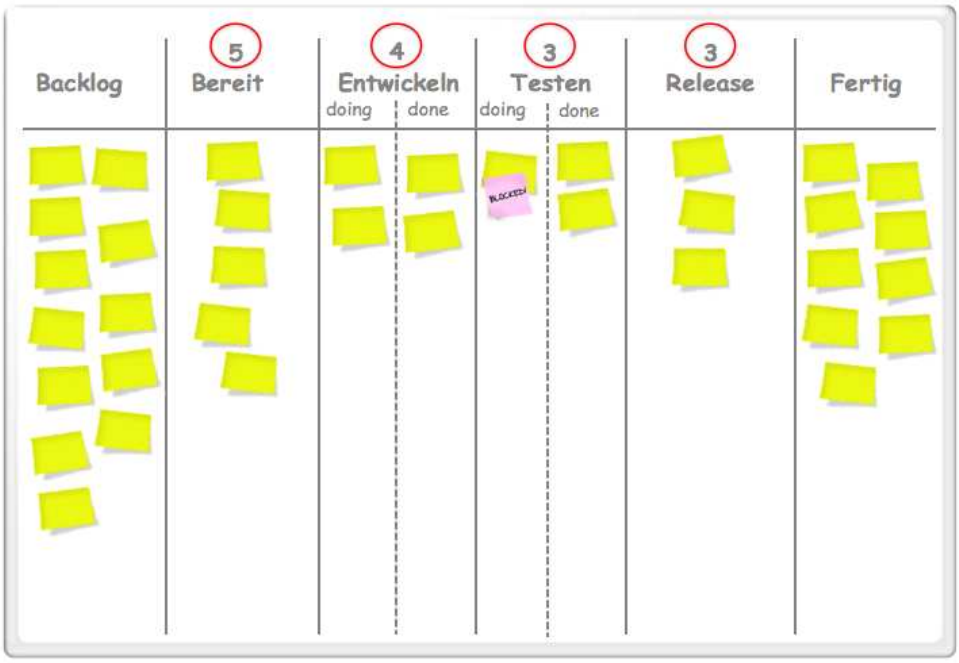


24.4 Kanban

- Kanban ist eine agile Methode für evolutionäres Change Management. Das bedeutet, dass der bestehende Prozess in kleinen Schritte (evolutionär) verbessert wird.
- Indem viele kleine Änderungen durchgeführt werden (anstatt einer großen), wird das Risiko für jede einzelne Maßnahme reduziert.
- Darüber hinaus führt der eher sanfte Stil von Kanban in der Regel zu weniger Widerständen bei den Beteiligten.

So funktioniert Kanban:

Der erste Schritt bei der Einführung von Kanban besteht darin, den bestehenden Workflow, die vorhandene Arbeit sowie Probleme zu visualisieren.
Dies wird in Form eines Kanban-Boards getan, das z.B. aus einem einfachen Whiteboard und Haftnotizen oder Karteikarten besteht. Jede Karte auf dem Board repräsentiert dabei eine Aufgabe.

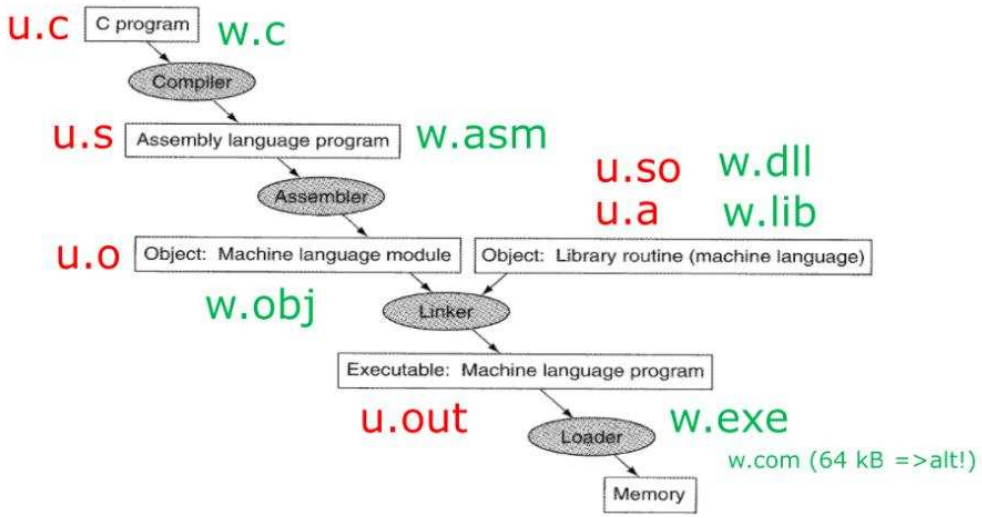


Allein diese einfache Maßnahme führt zu viel Transparenz über die Verteilung der Arbeit sowie bestehende Engpässe.

- Kanban ist ein übergreifender Ansatz, der prinzipiell auf jeden bestehenden Prozess aufgesetzt werden kann - egal ob agil oder nicht.
- Viele Prinzipien und Techniken aus Scrum ergänzen sich gut mit Kanban (etwa die Daily Scrums und die Burndown-Charts)
- und der ScrumMaster kann seine Rolle nutzen, um Kanban einzuführen.

25 Compilieren und Interpretieren

Entwicklung mit C:



Die Kompilierung erfolgt in 3 Schritten:

Lexikalische Analye (Scanner, Lexer)

- Strom aus Zeichen wird squentiell gelsen und in Symbole (Tokens) separiert
- Tokens werden mit Position in Quelltext assozieert.
- Lexikalischer Fehler: Zeichen oder Zeichenfolge, die keinem token zugeordnet werde kann, z.B. Bezeichner die mit Zahlen beginne "3foo".

Syntaktische Analyse (Parser)

- Symbole werden zu grammatikalischen Einheiten zusammengefasst
- Zusammenfassung erfolgt nach Syntaxregeln
- Visualisierung durch Syntaxbaum
- Syntaxfehler: Fehlende Klammer steht zu Beginn einer Methode oder es fehlt eine ";" am Ende einer Anweisung.
-

Semantische Analyse

- Erzeugung eines mit weiteren Attributen versehenen Sytanxbaum, d.h. attributierten Syntaxbaum
- Untersuchung von semantischen Fehler
  - Verwendete Variable muss deklariert sein
  - Datentypen müssen bei Zuweisung verträglich sein
  - Verwendung von nicht vorher definierten Bezeichner

Backend eines Compilers

- Auswertung des attributierten Syntaxbaumes
- Optimierung am Code (Syntaxbaum)
- Transformation des Syntaxbaumes in die Enddarstellung -> Maschinencode eines OS

26 Programmiersprachen

Die Unterteilung der Generationen von Programmiersprachen wird in Paradigment unterteilt, d.h. eine grundsätziche Denkweise.

Name	funktional	imperativ	objektorientiert	deklarativ	logisch	nebenläufig
Ada		X	X			X
C		X				
Prolog				X	X	
Scheme	X	X	(X)	X		(X)
Haskell	X	(X)		X		(X)

Dabei wird nebst OOP auch zwischen diesen Paradigmen unterschieden:

26.1.1 Prozedurale Programmierung

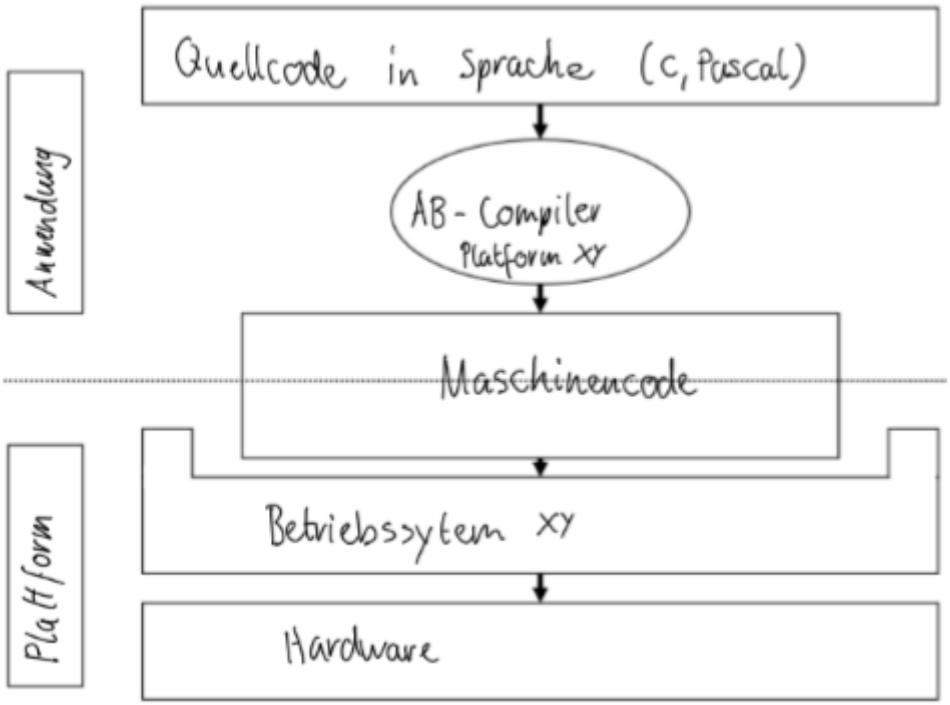
- Wird am nächsten mit Programmierung in Verbindung gesetzt
- Quellcode wird von oben nach unten gelsen -> Liste von Instruktionen
- Sprachen: C, C++ und Java -> nicht nur OOP sondern auch prozedural
    - Sind immer auch imperative Programmiersprachen
    - -> Befehl pro Linie an Compilero Anweisung
    - Anweisung kann man als Zustand des Programmes werten, z.B. Variable wird inkrementiert
- Imperative Programmierung beschreibt wie etwas berechnet werden soll
- Maschinencode

26.1.2 Funktionale Programmierung

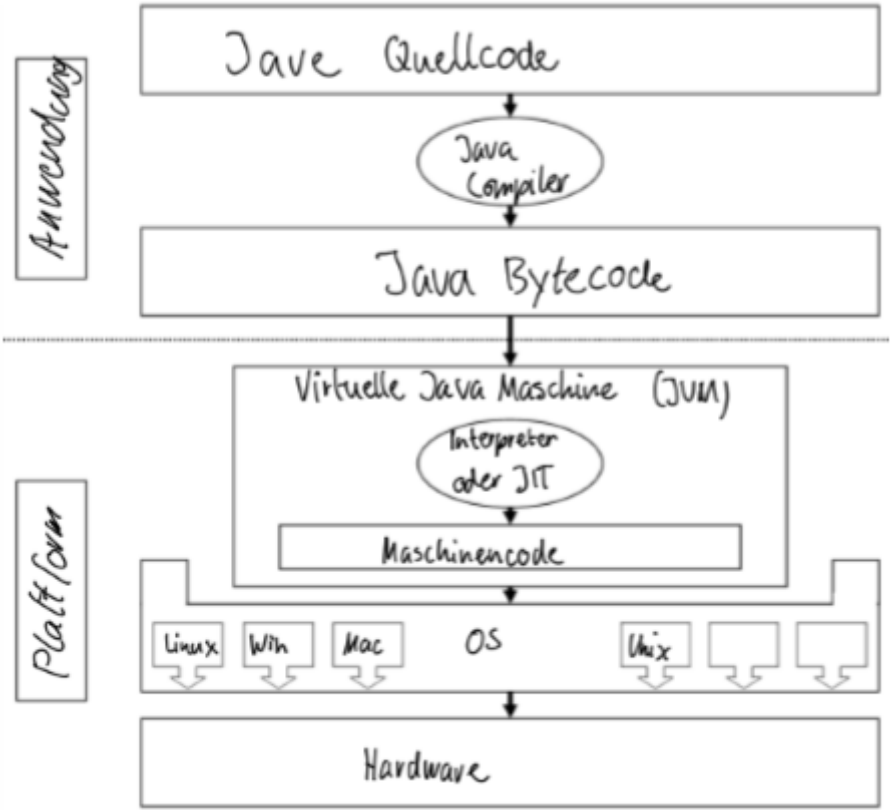
- Funktion im mathematischen Sine
- Zustandlos -> Eingabeparameter verändern Zustand des Programmes nicht
- Funktion kann ihren eigenen Zustand verändern -> concurrency, mehrere Instanzen laufen unabhängig
- Funktionen können andere Funktionen aufrufen (Rekursion)
- -> Entwicklung von Funktionen
- Funktionale Programmiersprachen sind immer deklarative Programmiersprachen
- -> Deklarative beschreibt was berechnet werden Soll
    - SQL z.B. beschreibt das Resulatat, der Lösungsweg mach die Programmiersprache
- Beispiele: Haskell, LISP oder Erlang -> beliebt in akademischen Krisen als Mittel der Mathematik

26.2 Enwicklungsparadigmen

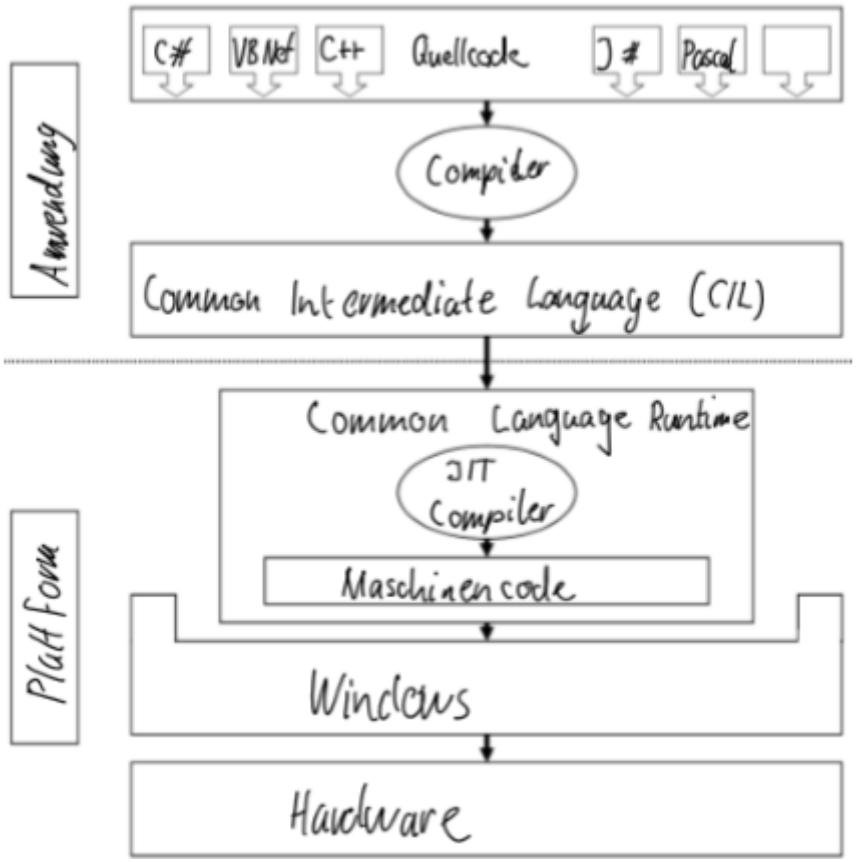
26.2.1 Klassisch:



26.2.2 JEE (Java Enterprise)

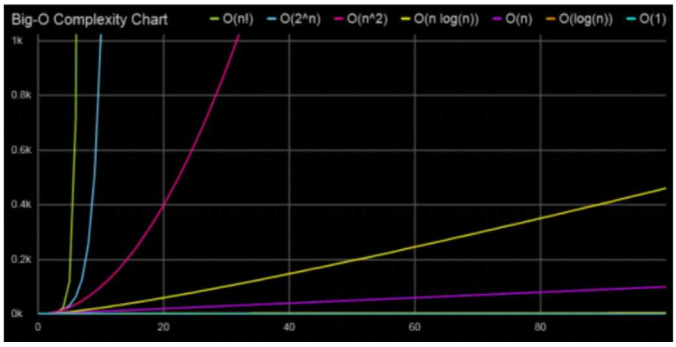


26.2.3 .Net (Microsoft)



28 Sortierung

Algorithm	Data Structure	Time Complexity			Worst Case Auxiliary Space Complexity
		Best	Average	Worst	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	Array	O(n+k)	O(n+k)	O(n^2)	O(nk)
Radix Sort	Array	O(nk)	O(nk)	O(nk)	O(n+k)



28.1 Bubble sort

```
BubbleSort(Array A)
for (n=A.size; n>1; n=n-1){
   for (i=0; i<n-1; i=i+1){
      if (A[i] > A[i+1]){
          A.swap(i, i+1)
      } // ende if
   } // ende innere for-Schleife
} // ende äußere for-Schleife
```

1. Beginnend beim ersten wird Element mit Nachbarverglichen
2. Grössere Zahl landet in der letzten Position
3. Erneuter Vergleich minus der Position bis alle Elemente durchiteriert worden sind

Komplexität: n^2

28.2 Ripple sort

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

Ähnlich dem Bubble-Sort

1. Jedes Element wird mit dem letzten Element verglichen und (wenn grösser) vertauscht
2. Nach jedem Durchgang liegt am Ende das grösste Element
3. Für jeweils eine Position weniger wird dieser Vorgang rekursiv durchgeführt

Komplexität: n^2

28.3 Insertion sort

```
void InsertionSort(vektor){
    for (int i = 1; i < vektor.Length;
                                    i++) {
        int temp = vektor[i];
        for (int j = i-1; j >= 0; j--) {
            if (vektor[j] > temp) {
                vektor[j+1] = vektor[j];
                vektor[j] = temp;
            }
        }
    }
}
```

Vergleichbar mit Jassen beim Aufnehmen der Karten

1. Schleife: Werte iterieren
2. Beginnend beim zweiten Elemente wird dieser mit allen vorgängigen Position verglichen
3. Der grössere Wert wird jeweils rechts eingeschoben
3. Dieser Vorgang wird rekursiv durchgeführt

Komplexität: n^2

28.4 Selection sort

```
SelectionSort(vektor){
   for (int i=vektor.size-1; i>0;i--){
      int max = 0;
      int temp =0;
         for (int j=1; j<=i;j++){
            if (vektor[j]>vektor[max]){
               max=j;  // merke
            }          // grösste Zahl
         }
      temp = vektor[i];
      vektor[i] = vektor[max];
      vektor[max] = temp;
  }
}
```

Ähnlich Ripple, nur wird erst kontrolliert um am Schluss getauscht.

1. Für jede Position wird geschaut welcher Werte der kleinste oder der grösste der vorausgehenden Position ist.
2. Das kleinste wird an die aktuelle Stelle platziert.
3. Das wird solange wiederholt bis man an der letzten Position ist.

Beim Ripple sort kann es zu mehreren Tausch kommen bevor der Maximal Wert an der entsprechenden Position liegt.

28.5 Shell sort

```
ShellSort(Array A)
static void  shellsort (int[] a, int n)
{
   int i, j, k, h, t;

   int[] spalten = {1743392200, 581130733, 193710244,
64570081, 21523360, 7174453, 2391484, 797161, 265720,
 88573, 29524, 9841, 3280, 1093, 364, 121, 40, 13, 4, 1};

   for (k = 0; k < spalten.length; k++)
   {
      h = spalten[k];
      // Sortiere die "Spalten" mit Insertionsort
      for (i = h; i < n; i++)
      {
         t = a[i];
         j = i;
         while (j >= h && a[j-h] > t)
         {
            a[j] = a[j-h];
            j = j - h;
         }
         a[j] = t;
      }
   }
}
```

...hmm:
3n + 1
eine Möglichkeit!

Bedient sich Prinzip vom Insertion sort.
Jedoch möchte man vermeiden, dass Elemente über weite Strecken verschoben werden müssen.
Deshalb wird die Sequenz in Untersequenzen zerlegt und einzeln sortiert.

1. Aufteilung in z.B. 4er Blöcke
2. Sortierung blockweise -> 4-sortiert
3. Wiederholen für 2 und 1 Blöcke
4. Letzter als 1er Block entspricht dann wieder Insertion sort

28.6 Quick sort

Ganze Familie von sorts, es gibt etliche Versionen.
Ist ein Rekusiver Alogrithmus

```
int partition(int z [], int l , int r) {
  int x = z[r], i = l-1, j = r ;
  while (1) {
    while (z[++i] < x)
      ;
    while (z[--j] > x)
      ;
    if ( i < j)
      swap(&z[i], &z[j ]);
    else {
      swap(&z[i], &z[r ]);
      return i ;
    }
  }
}
void quick sort(int z[], int l , int r) {
  if ( l < r) {
    int pivot = partition(z, l , r );
    quick sort(z, l, pivot-1);
    quick sort(z, pivot+1, r);
  }
}
```

1. Teilen der Liste in zwei Teillisten (links und rechts)
2. Vergleich der Werte anhand Pivotelement (das letzte Element im Array)
3. Linke Liste enthält kleinere Elemente und rechte die Grösseren
4. Diesen Vorgang (1-3) nun für jede Teilliste rekursiv durchführen
5. Ist Grösse der Liste eins oder 0 erfolgt Abbruch der Rekursion

28.7 Merge sort

- Empfiehlt sich als Pivot Element

```
void merge(int z[], int l , int m, int r) {
  int i , j , k;
  for ( i=m+1; i>l; i--)
    hilf [ i-1] = z[i-1];
  for ( j=m; j<r; j++)
    hilf [r+m-j] = z[j+1];
  for (k=l; k<=r; k++)
    z[k] = ( hilf [ i ] < hilf[ j ]) ?
              hilf [ i++] : hilf [ j--];
}
void merge sort(int z[], int l , int r) {
  if ( l < r) {
    int mitte = ( l+r)/2;
    merge sort(z, l, mitte);
    merge sort(z, mitte+1, r);
    merge(z, l, mitte, r );
  }
}
```

Die gesamte Menge wir als erstes in kleinere Listen zerlegt.

2. Diese Teillisten werden sortiert
3. Die Teillisten werden dann nach dem Reisverschlussprinzip zusammengefügt.
4. Jede neue Kombination von Listen wird dann ebenfalls nach dem Reisverschlussprinzip zusammengefügt.

29 Datenstrukturen

Datenstrukturen können sein:

Arrays

```
Int [] array = {4, 7, 2, 1}
```

- Addressierbarkeit über Index
- 1 bis n dimensional
- Komponenten könnten primitive Datentypen oder ganze Objekte sein

Listen

Passendes Beispiel wäre LinkedList

- Homogen, gleiche Strukturen verketten
- dynamische, zur Laufzeit veränderbar

Stacks

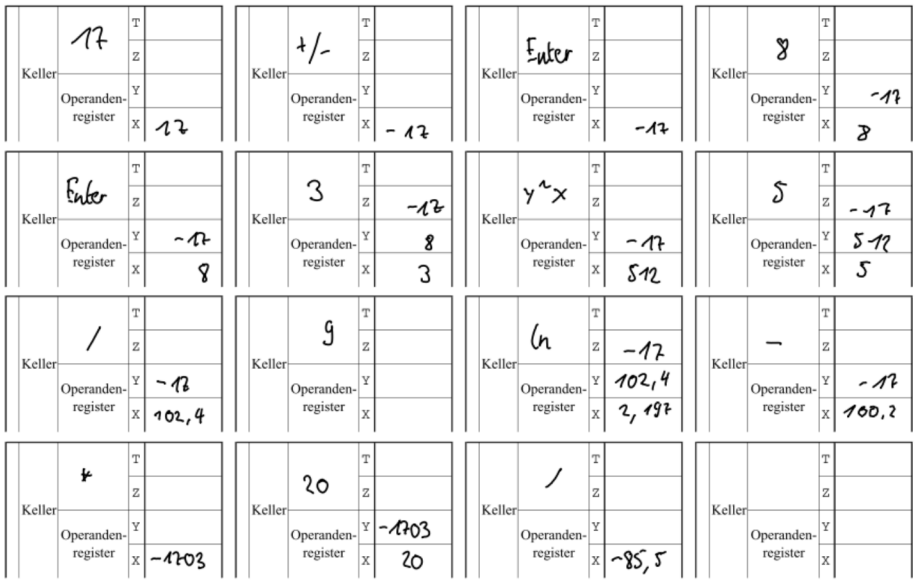
Für LIFO-Bearbeitung (Last In First Out) der Komponenten Ist ebenfalls homogen, dynamisch und abstrakt.

Beispiel UPN Rechner:

$$\frac{-17\cdot\left(\frac{8^3}{5}-\ln 9\right)}{20}$$

Früher kannten Taschenrechner noch keine Klammern, folge jeder Hersteller hatte eine eigenen Notation.

Dazu ein Beispiel von HP:



30 Trees (Bäume)

Bäume sind im Vergleich zu anderen Datenstrukturen nichtlinear.
In Bäumen kann man

- Nicht nur Daten, sondern auch
- relative Beziehungen der Daten untereinander,
- Ordnuungsgeziehrungen (siehe geordnete Bäume)
- hierarchische Beziehungen
- und Gewichte auf den Kanten (Zeile) speichern

Vorteil von Baumstrukturen bietet die Durchsuchung, ebenfalls könne viele Operationen mit niedriger O-Komplexität durchgeführt werden (Suchen, Insert, Delete, …)

30.1 Rekursive Definition

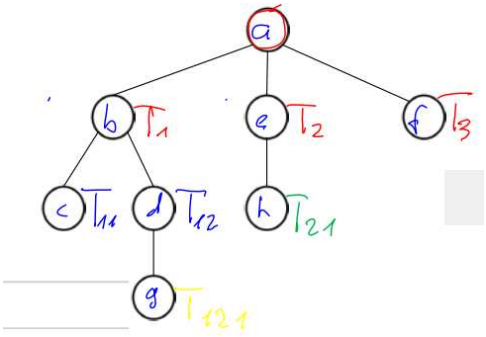
Ein endlicher Baum ist eine endliche Menge T, die in n+1 (n ∈N) paarweise disjunkte Teilmengen T0, ..., Tn zerlegt ist mit:
1. |T0| = 1
2. Ti ist wieder ein Baum (1<= i <= n)
Alle Teilbäume haben immer +1 Child.

Eine mathematische Notierung sieht dann so aus:

$$T = (t, T_1, ..., T_n) \; mit : T = \{t\} \cup \bigcup_{i=1}^{n} T_i$$

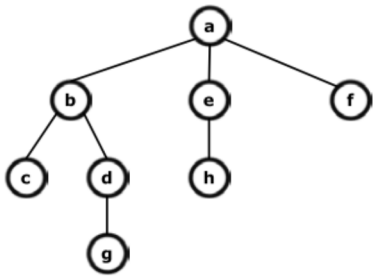
t: Wurzel (nur Name)
Ti: Teilbäume von T
n: z.B. n = 0 dann Besteht Baum aus nur einer Wurzel und hat keine Teilbäume.
u: Vereint
U: Summe

Eine Formale Definition kann dann so aussehen:



$$\begin{aligned} T &= (t, T_1, T_2, T_3) \\ &= (a, T_1, T_2, T_3) \\ &= (a, (b, T_{11}, T_{12}), (e, T_{21}), (f)) \\ &= (a, (b, (c), (d, T_{121})), (e, (h)), (f)) \\ &= (a, (b, (c), (d, (g))), (e, (h)), (f)) \end{aligned}$$

30.2 Elemente



Knoten (nodes): a-f bzw. N1-Ni
Kanten: Verbindungen (a->f)
a ist der Elternknoten des Baumes
b ist ein direkter Kindknoten
Ein Knoten ohne Kindknoten ist Blatt (leaf)
externe Knoten (äussere Knoten) und innere Knoten = Alle Knoten

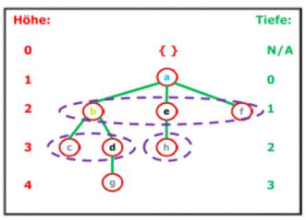
- externe Knoten (c, g, h, f)
- innere Knoten (b, d, e, a)

Knoten ohne Elterknoten ist die Wurzel (a)
Jeder Kindknoten ist Wurzel eines Teilbaums (ti) (b,c, d, g, e, h, f)
Eine Pfad (path) ist der Weg von einem Knoten zu einem anderen {h, e, a, b, c}
Die Länge l ist die Mächtigkeit der Menge der Knoten in einem Pfad (c zu h -> l = 5)
Betrachtet man nur absteigende Pfade so gilt:

- a nach b -> b ist Nachkomme (kind oder child) von a (successor)
- a ist Vorfahre (Eltern, parent, predecessor)
- d ist Vorfahre von g, f ist ein Nachkomme von a
- a ist ein Vorfahre von c, aber e kein direkter Vorfahre.

30.3 Höhe und Tiefe

Zwischen Höhe und Tiefe muss ganz klar unterschieden werden, je nach Definition unterscheiden sich diese Werte.



Tiefe ist Attribut des Knotens

Höhe ist Attribut des Baumes

- Die direkten Nachkommen eines Elternknotens (="Geschwister") bilden ein **Niveau** oder eine **Ebene** (*layer*).
- Die **Tiefe** eines Kontens ist die Anzahl Bögen (**Kanten**) bis zur Wurzel oder auch die Länge des Pfades von der Wurzel zu ihm.
- Ein Baum hat immer ein **Höhe** h:
  - Ein leerer Baum **{ }** hat die Höhe 0
  - ein Baum aus "nur" einem Wurzelknoten hat die Höhe 1
  - => unser Baum oben hat die **Höhe** 4
  - => unser Baum oben hat die **Tiefe** 3
- Es gilt die Gleichung: Höhe = Tiefe +1

Der Grad eines Graphen = die Anzahl der Kanten.

➔ **Array und LinkedList = entartetet Bäume Grad 1**

30.4 Eigenschaften

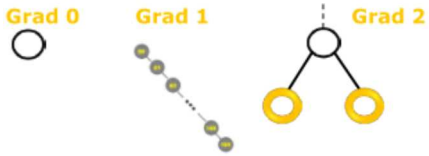
30.4.1 Geordnete Bäume

Bei geordneten Bäumen ist klar die Reihenfolge von direkten Nachfolger bei jedem Knoten festgelegt

30.4.2 Grad

- Auch Knotengrad oder Valen ist ein Begriff aus der Graphentheorie.
- Beschreibt die Anzahl ausgehenden Kanten von einem Knoten.
- Entspricht der maximalen zulässigen Anzahl direkter Kindknoten

Grad	**Baumtyp**
Grad 2	Binärbaum
>Grad 2	Vielwegbäume
Grad 1	LinkedList, Arrays (nur eine Liste)
Grad 0	Baum aus einem Blatt

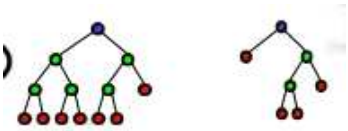


30.4.3 Balancierte Baum

Ist ein Spezialfall der Datenstruktur Baum.
Es gilt:

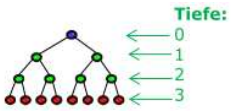
- Maximale Höhe c*log(n)
- Wobei n Anzahl Elemente
- C und n sind unabhängige Konstanten

Man bezeichnet ihn als voll, wenn jeder Knoten entweder Blatt ist (also kein Kind besitzt), oder aber zwei (also sowohl ein linkes wie ein rechtes) Kinder besitzt – es also kein Halbblatt gibt.

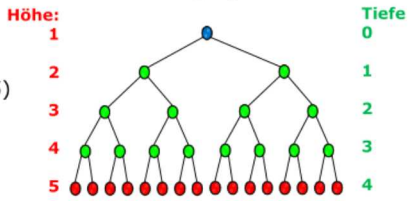


30.4.4 Vollständiger Binärbaum

Man bezeichnet volle Binärbäume als vollständig, wenn alle Blätter die gleiche Tiefe haben, wobei die Tiefe eines Knotens als die Anzahl der Bögen bis zur Wurzel definiert ist. --> kommt sehr selten vor



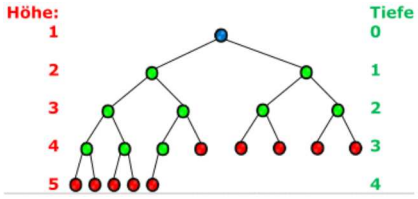
Bei einem **vollständiger Binärbaum**
mit der Höhe h gilt: (z.B. für h = 5)
Es gibt 2^h-1 Knoten => $2^5 - 1 = 31$
Es gibt 2^{h-1}-1 innere Knoten => $2^4 - 1 = 15$
Es gibt 2^{h-1} Blätter => $2^4 = 16$
Es gibt auf der **Höhe h** genau
2^{h-1} Knoten
z.B. h = 3 => $2^{3-1} = 4$
oder gleiche Aussage
Es gibt auf der **Tiefe t**
(0<= **t** <= h-1)
genau 2^t Knoten z.B. t = 2 => $2^2 = 4$
Es gilt ja: **h = t + 1**



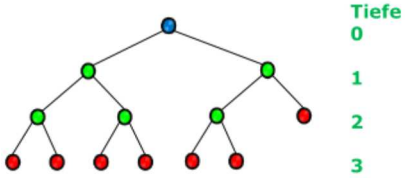
Die minimale Höhe h eines Binärbaumes lässt sich aus der Anzahl Knoten n berechnen:

$\log_2(n) = h$

Lösung immer abrunden.
Beispiel:



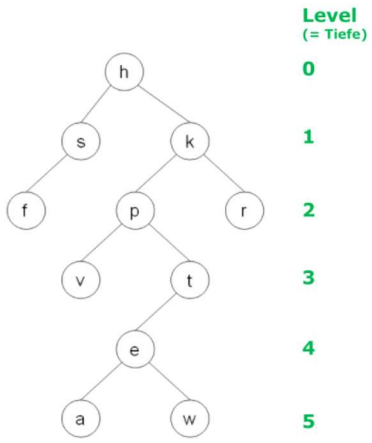
Log10(20)/log10(2)=4.321928094887362
Achtung!: Ergibt immer die Tiefe



Log10(13)/log10(2)=3.700439718141092

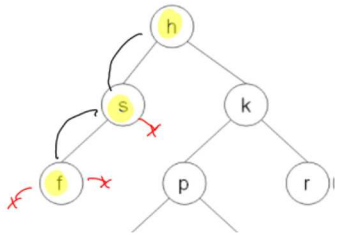
30.4.5 Traversierung von Bäumen

Gegeben ein nicht ausbalancierter Baum.
Dieser soll in verschiedenen Methoden durchlaufen werden:

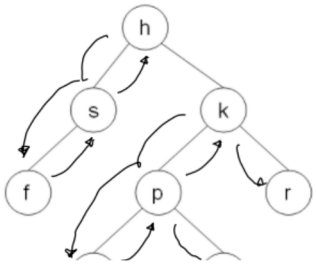


Allgemein gilt:
Ist kein Schritt durchführbar, geht man einen Teilbaum zurück.

Preorder (WLR)
Durchlaufe den Baum rekursiv in folgenden Schritten
1. Lies Wurzel
2. Traversiere linken Teilbaum in Preorder
3. Traversiere rechten Teilbaum in Preorder



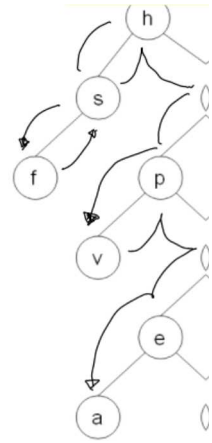
Inorder (LWR)
Inorder Führe Schritte durch bis nicht mehr möglich, dann nächsten Schritt
1. Traversiere linken Teilbaum in Inorder
2. Lies Wurzel
3. Traversiere rechten Teilbaum in Inorder



Postorder (LWR)

Postorder fange Alle Schritte von vorne an, wenn einer durchgeführt.

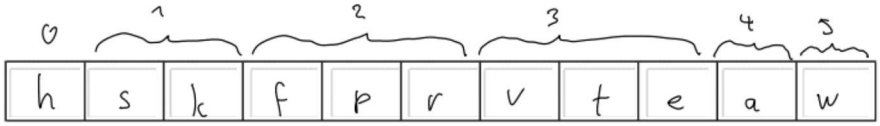
1. Traversiere linken Teilbaum in Postorder
2. Traversiere rechten Teilbaum in Postorder
3. Lies Wurzel



Levelorder

Beginnend beim ersten Level werden alle Elemente pro Level gelesen.

1. LevNo:=0
2. solange Ebene levNo
    a. Lies die Knoten EvenelevNo von links nach rechts
    b.levNo:=levNo+1



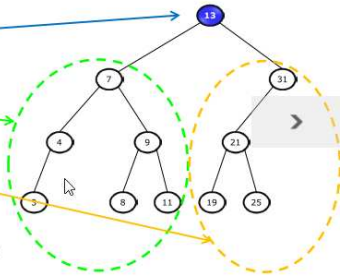
30.4.6 Binäre Suchbäume

Ein binärer Suchbaum hat **Grad 2** und jeder Teilbaum hat folgendes **Ordnungsprinzip**:

- Er hat eine **Wurzel**.
- Der Inhaltswert des linken Kindes ist **kleiner als der der Wurzel** (oder gleich gross).
- Der Inhaltswert des rechten Kindes ist **grösser als der der Wurzel** (oder gleich gross).

Diese "Regel" gilt für **alle** inneren Knoten!

Ein binärer Suchbaum liefert **Inorder** ausgelesen eine korrekte, aufsteigende Sortierung:



3	4	7	8	9	11	13	19	21	25	31

Frage: Was passiert jetzt bei **Insert** und **Delete**?

nächste Folie