Algorithmen:
„Eine effektive Methode, ausgedrückt in einer endlichen Liste wohldefinierter Anweisungen zur Berechnung einer Funktion.“

Erstellung eines Algorithmus':

- Aufgabe
- Entwurf eines Algorithmus
- Darstellung (Pseudocode, Struktogramm, Programmablaufpläne (PAP))
- Codierung (⇒ für Compiler)
- Erfassung mit Editor programm.cpp
- Compilierung (1. Präprozessor, 2. Compiler) ⇒ Syntax programm.o
- Syntaxfehler-Korrektur und Neucompilierung
- Linker mit Laufzeitbibliotheken programm.exe
- Ausführen und Testen

Compiler analysiert Syntax, welche allgemein in Metasprache (BNF - Backus-Nauer-Form) definiert ist.

Listing 1: prinzipieller Aufbau

```
1 #include <stdio.h>
2 #include <image.h>
3 int main(int anz, char* para[])
4 {
5 /* Kommentar */
6 return 0; //Kommentar bis Zeilenende
7 }
```

Namen: 1. [Buchstabe | _] , dann {Ziffer|Buchstabe|_} beliebig oft.

Makros: {Großbuchstaben | _} beliebig oft.

#include<...> sucht im Compilerverzeichnis; **#include**"..." sucht im aktuellen Arbeitsverzeichnis.

printf("control␣string", argument list); und scanf("control␣string", &argument list) steuern die Standard Ein- und Ausgabe; dabei kann im control string folgendes stehen:

- Zeichenketten
- Formatbeschreiber (%f - float,%lf - double, %i - integer, %d - integer, %s - string, %c - char)
- Escape-Sequenzen (\n - newline, \\ - backslash, \" - Anführungszeichen, \o - oktal)

Ein-, Ausgabepuffer leeren: fflush (stdin), fflush (stdout).

Variablen verändern: **signed**, **unsigned**, **long**, **short**, **long long**, **short short**,...

wchar_t ist ein Unicode-Charakter; **char** name[80] ergibt dann einen String.

Konstanten: **double** a=3.12;, **#define** PI 3.141593 (ohne Semikolon, da Präprozessoranweisung!), **const int** a = 10;.

Arithmetische Operationen:

Zuweisung =

Operatoren $\underbrace{+,-}_{\text{Vorzeichen}}$ $,\cdot,/,+,-,$ $\underbrace{\%}_{\text{Modulo}}$ (Vorsicht bei ganzzahliger Division!)

Inkrement-Dekrement $--i$, $++i$ (Präfixnotation); $i--$, $i++$ (Postfixnotation, $i = i-1$, $i = i+1$); $x+=\Delta$; ($\Leftrightarrow x = x+\Delta$;).

C ist nahezu immer linksassoziativ! (x=y=10)

Prioritäten (hoch zu niedrig):	
· ()	
· +, −	(Vorzeichen)
· ++, −−	
· ∗, /, %	
· +, −	
· =	(Zuweisung)

Schleifen:

while **while** (n < 10) { ...; n++;}

for **for** (**int** i = 0; i < 10; i++) {...;}

do while **do** {...; n++} **while** (n < 10); (Durchlauf mind. 1 Mal.)

break; zum Verlassen einer Schleife; **continue**; zum Verlassen eines Schleifendurchlaufs; exit (1); zum Programmabbruch.

Eine einfache Alternative: **if** (...) {...;} **else** {...;}

Vergleichsoperatoren: <, >, <=, >=, ==, != (! zur Negation.)

if (a=b)... „=“ - Zuweisung!

Newton-Iterationsverfahren für die Wurzel einer Zahl z: $\lim_{i\to\infty} x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$, mit $f(x) = x^2 - z$.

Iterationsverfahren
= Relaxationsverfahren
= Lernverfahren

Abbruchkriterien:

1. absolute Fehler: $|x_{i+1} - x_i| \leq \varepsilon$
2. relative Fehler: $\frac{|x_{i+1}-x_i|}{|x_i|} \leq \varepsilon$

- „Unsicherheit / **Entropie** pro Zustand“: $H = \log_2(m)$, $[H] =$ bit für m Zustände;
- „(mittlere) Entropie“: $H_0 = -\sum_{i=1}^{m} p_i \log_2(p_i)$, mit $0 \leq H_0 \leq \log_2(m)$, p_i der Wahrscheinlichkeit für einen der Zustände m $(0 \cdot \log_2(0) = 0)$;
- „differentielle Entropie“: $H_0 = -\int_{-\infty}^{\infty} f(x) \log_2(f(x))\,\mathrm{d}x$, wobei $\exists H_0 < 0$!

Codes: (Graycode -> Speicherfehlerminimerung)

EBCDI-Code: 8 Bit pro Zeichen
ASCII-Code: 8 Bit pro Zeichen
UNICODE: 32 Bit pro Zeichen
} Blockcode

UTF-8 multibyte Code

B-adisches Zahlensystem: $z = \pm \sum_{k=-m}^{+n} a_k B^k$ mit a_k Ziffern, B Basis, m Stellen hinter Komma, n Stellen vor Komma.

Dualsystem: $B = 2, a_k \in \{0, 1\}$; Oktalsystem: $B = 8, a_k \in \{0, 1, ..., 7\}$; Dezimalsystem: $B = 10, a_k \in \{0, 1, ..., 9\}$; Hexadezimalsystem: $B = 16, a_k \in \{0, 1, ..., 9, A, ..., F\}$.

dezimal: $z = 0.\overline{9} \Rightarrow 9z = 10z - 1z = 9 \Rightarrow z = 1$
binär: $z = 0.\overline{011} \Rightarrow 2^3 z - z = 011 \Rightarrow z = \frac{011}{2^3-1}$

Umwandlung von Dezimalzahlen zu Dualzahlen:

1. Divisionsrestverfahren:
   $z = a_n\,2^n + a_{n-1}\,2^{n-1} + ... + a_1\,2^1 + a_0\,2^0$; dann: $z_{i+1} = z_i/2$ (ganzzahl. Div.) und $a_i = z_i \bmod 2$.
2. Linkspunktzahlen:
   $z = a_{-1}\,2^{-1} + a_{-2}\,2^{-2} + ...$; dann $z_{i+1} = z_i \cdot 2 - a_i$ und $a_i = 1$ wenn $(z_i \cdot 2) \geq 1$, $a_i = 0$ wenn $(z_i \cdot 2) < 1$.
3. Beliebige dezimale Festpunktzahlen: Kombination beider Verfahren.

$2^{10} \approx 10^3$

Zweierkomplement:
Sei $z > 0$, dann: $+z = z^*$, 1. Bit 0 ; mit $-2^{n-1} \leq z \leq 2^{n-1} - 1$.
$-z = z^* - 2^n$, 1. Bit 1

2er-Komplementbildung: Negation aller Bits und anschließende Addition von 1.

Vorteile: 0 eindeutig; Subtraktion auf Addition zurückführbar.

	0	1	0	0	0	(8)
+	1	1	1	1	1	(-1)
(1)	0	0	1	1	1	(7)

Binäre Gleitpunktzahlen nach IEEE 754: $z = \pm\text{Mantisse} \cdot \text{Basis}^{\pm\text{exp}}$ mit Basis=„2“, Mantisse=„$1, m$“ und exp=„$e - (2^{n-1}-1)$“; wobei e (≥ 1) und m die wirklich gespeicherten Größen sind. Die Darstellung von exp heißt „Verschiebecode“. Und die 0 hat dabei eine Sonderrelle: Sie wird nur mit Nullen dargestellt.

In limits.h stehen die maximalen und minimalen Werte der Zahlenformate. Mit **sizeof**(); kann man sich die Anzahl verwendeter Bytes aber auch ausgeben lassen.

int a = ... - dezimal; **int** a = 0... - oktal; **int** a = 0x... - hexadezimal.

(type)expression wandelt expression in das Format type um.

Runden: **double** c; **int** i = **int**(floor(c+0.5))

Listing 2: Mehrfachverzweigung

```
1 switch( ganzer Ausdruck );
2 {
3  case konst_1: ...; break;
4  case konst_2: ...; break;
5
6  case konst_n: ...; break;
7  default: ...;
8 }
```

Dabei sind die **case** konst_i: Sprungadressen und keine Bedingung für das Ausführen der nachfolgenden Befehle.

Bedingte Ausdrücke: x = condition ? ausdr_1 : ausdr_2; ;
wobei x = ausdr_1 wenn condition wahr und x = ausdr_2 wenn condition falsch.

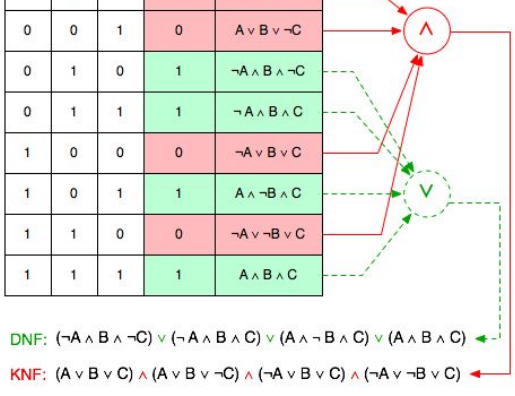
Boolesche Algebra:
Algebraische Struktur $B = \{a, b, ...\}$, Operationen „+“, „·“, Gesetze:

1.	$a + b = b + a$	$a \cdot b = b \cdot a$	Kommutativität
2.	$(a + b) + c = a + (b + c)$	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	Assozitivität
3.	$+a^m = a$	$\cdot a^m = a$	Idempotenz (Absorptionsgesetz)
4.	$a + (b \cdot c) = (a + b) \cdot (a + c)$	$a \cdot (b + c) = a \cdot b + a \cdot c$	Distributivität
5.	$\exists n \in B : a + n = a$	$\exists e \in B : a \cdot e = a$	$\forall a \in B$ Neutrale Elemente
6.	$a + \overline{a} = e$	$a \cdot \overline{a} = n$	unitäre Operation $\overline{a}$
7.	$\exists$ Gleichung G in $B \Rightarrow \exists \overline{G}$ mit $\begin{cases} \cdot \to + \\ + \to \cdot \\ n \to e \\ e \to n \end{cases}$		Dualitätsprinzip

Mengenalgebra:
$B = \{A, B, ...\}$, $+ \to \cup$, $\cdot \to \cap$, $\overline{A} \to$ Komplementärmenge, $n \to \emptyset$, $e \to E$ (Universalmenge).

Zweiwertige Logik:
$B = \{0, 1\}$, $+ \to$ Disjunktion (OR, $\vee$), $\cdot \to$ Konjunktion (AND, $\wedge$), $\overline{a} \to$ Negation ($\neg$), $n \to 0$, $e \to 1$.

A	B	C	Ergebnis	Klausel
0	0	0	0	A ∨ B ∨ C
0	0	1	0	A ∨ B ∨ ¬C
0	1	0	1	¬A ∧ B ∧ ¬C
0	1	1	1	¬A ∧ B ∧ C
1	0	0	0	¬A ∨ B ∨ C
1	0	1	1	A ∧ ¬B ∧ C
1	1	0	0	¬A ∨ ¬B ∨ C
1	1	1	1	A ∧ B ∧ C



Schaltfunktion aus gewünschtem Ergebnis erstellen:

1. Kanonisch disjunktive Normalform (KDNF):
   Disjunktion von Konjunktionsthermen ($\bigvee_i \bigwedge_j (\neg) x_{ij}$).
2. Kanonisch konjunktive Normalform (KKNF):
   Konjunktion von Disjunktionsthermen ($\bigwedge_i \bigvee_j (\neg) x_{ij}$).

(($\neg$) $\Leftrightarrow \neg$ optional)

Bit-Operatoren (nur für ganze Operanden sinnvoll):

- Shift-Operatoren: << , >> ;
- unitärer Negationsoperator: ~ ;
- Konjunktion: & ; Disjunktion: | ; XOR: ^ .

Boolsche-Operatoren (nur für ganze Operanden sinnvoll):

- AND: && ;
- OR: || ; (XOR gibt es so nicht)
- NOT: ! .

Lazy Evaluation: von links nach rechts! (**if** (x != 0 && y/x == 24) {...})

Variablenwerte tauschen:

```
int x,y;
x=x−y;
y=x+y;
x=y−x;
```

Arrays:
type name[num1][num2]; deklariert einen „Vektor“, dessen Elemente num1 „Vektoren“ mit num2 Elementen des Typs format sind; auf das i,j-te Element kann mit name[i−1][j−1] zugegriffen werden.
C speichert dieses Array intern zeilenweise [⇔ der hinterste ist der schnellste Index].
int x[42] = {1,2,3}; deklariert x, weist den ersten 3 Elementen $1,2,3$ zu und der Rest wird mit 0 aufgefüllt.

Strings: in C als Array von chars. **char** name[15] = {'S','u','e','s','s','e','\0'}; ⇔ **char** name[15] = "Suesse";

Zeichenkettenverarbeitung:

- **#include** <string.h> ,
- gets(name) („get string“),
- puts(name) („put string“),
- strlen(name) („string length“),
- strcpy(to, from) („string copy“),
- strcat(to, add) (hängt add an to an),
- strcmp(name1, name2) (vergleicht von links zeichenweise, liefert $+1, 0, -1$),
- sprintf(string, "control␣string", arguments); (wie printf, aber auf Variable string).

Pointer: (≙ typisierten Adressen)
Deklaration: type* name; ; als Format auch **void** möglich (→ nicht typisierte Zeiger).
Adressoperator: &name gibt die Adresse einer Variablen name aus.
Dereferenzierung: *ptr dereferenziert, so dass hier der Wert an der Stelle ptr zurückgegeben wird.
Zeigerarithmetik: ptr += 5 verscheibt den Pointer um 5 Einheiten des Typs des Pointers weiter auf dem Speicher.
Null-Zeiger: ptr = NULL; (⇔ ptr = 0) wegen der Lesbarkeit.
Die „Namen“ von Arrays sind Pointerkonstanten!

Listing 3: Pointer

```
1 double x=25.0, y=0;
2 double* ptr_x;
3 ptr_x = &x;
4 y = *ptr_x;    //gibt 25.0 aus
5 *ptr_x = 10;   //speichert 10 an die Stelle
```

Listing 4: String kopieren

```
1 char string_1[80] = "Meyer"; char string_2[80];
2 char* pq = string_1; char* pz = string_2;          // Array−Name = Pointerkonstante
3 while (*pq) {*pz = *pq; pz++; pq++;}               // String−Ende ist NULL
4 *pz = '\0';
```

Feld von Pointern: **int*** ptr[50]; (äquivalent vielen: **int*** *ptr).

Konstante Pointer:

- Wert (a), wo Pointer hinzeigt, ist konstant, Pointer (ptr) nicht: **char** a = 'b'; **const char*** ptr;
- Pointer konstant (ptr), Wert überschreibbar (a): **char* const** ptr = &a;
- Pointer konstant (ptr), Wert konstant (a): **const char* const** ptr = &a;

Bei Konstanten ist natürlich die Initialisierung bei der Deklaration nötig!

Dynamische Feldvereinbarungen: (Speicherplatzzuweisung)

void* malloc(Anzahl Bytes); erfragt vom System Platz im Hauptspeicher für Anzahl Bytes („memory allocation").

Listing 5: Dynamischer Vektor

```
int* feld;
feld = (int*) malloc(n*sizeof(int));
...
free (feld);
```

Listing 6: Dynamische Matrix

```
int** matrix;
matrix = (int**) malloc(m*sizeof(int*));
for (int i = 0; i < m; i++)
{
 matrix[i] = (int*) malloc(n*sizeof(int));
}
...
for( int i = 0; i < m; i++)
{
 free(matrix[i]);
}
free(matrix);
```

Dabei sind die Zeilen der Matrix u.U. im Hauptspeicher verstreut; soll die Matrix als hintereinanderliegende Zeilenvektoren abgespeichert werden, so: matrix = (**int****) malloc(m*n*sizeof(**int***)); und einteilen der Pointer für Zeilen...

free( ) gibt den Speicherplatz wieder frei; eigentlich nur beim Verlassen von Unterprogrammen nötig.

Halbdynamische Zuweisung:

int (*matrix)[10]; erzeugt einen Pointer, der mit 10 Integern typisiert ist.

Listing 7: Dynamischer Vektor

```
scanf("%d",m);
matrix = (int (*) [10]) malloc(m*10*sizeof(int)); // Casten des zurückgegebenen Vektors
```

Bei der Verarbeitung von Matrizen Zeilenweise Abspeicherung ausnutzen (Zeilenweise, über *(matrix++))!

Sortierung:

Distribution Sort Adressraum (alle Möglichkeiten) vollständig im Speicher abbilden, dann einfach die Anzahl eines jeden Elements angeben (zu Deutsch: „Histogrammmethode").

Selection Sort Maximum suchen, mit dem am weitesten hinten stehenden, noch nicht bewegten Wert tauschen.

Insert Sort 1. Element als sortierte Liste annehmen; 2. passend einfügen (dazu u.U. die vorigen Elemente der Liste umkopieren); 3. passend einfügen ...

Bubble Sort Von vorn beginnend jeden Wert und den rechts daneben vergleichen und nach Sortierkriterium entweder vertauschen oder nicht; so viele Durchläufe, bis keine Vertauschung mehr stattfand.

(Median: das mittlere Element einer sortierten Liste
(bei gerader Anzahl meist der Mittelwert der beiden mittleren Elemente))

Pointersortierung:
Feld von Pointern auf die Datensätze anlegen; die Datensätze über dereferenzierte Pointer vergleichen und diese dann entsprechend sortieren (mit einem beliebigen Algorithmus).
Anschließend Sortierung der Daten durch Vertauschung analog den Pointern möglich (→ Permutationen, Zykelschreibweise).

Modulkonzept:
Ein Modul heißt in C „Funktion“ oder Unterprogramm.

Listing 8: Funktion

```
double name( double var1, int var2)
        {
        double variable;
        ...
        return variable;
        }
```

var1 und var2 heißen „formale Parameter“ und das **double** vor dem Funktionsnamen name deklariert den Rückgabewerttyp der Funktion.
int ist das Standardrückgabeformat (wenn nichts deklariert wird).
Soll nichts zurückgegeben werden, so muss das Rückgabeformat mit **void** definiert werden und man darf **return** nicht verwenden.

void print_line() { printf("\n"); }

Um Funktionen in einem anderen Programm verwenden zu können, muss man über dem Programm eine Prototypendeklaration (→ **Header-Files**) für den Compiler einfügen; diese sieht genau wie der Kopf der Funktion aus, nur die Namen der Variablen können weggelassen werden; z.B.: **double** name(**double** , **int**) ; .

Damit bei Mehrfachaufruf von Header-Dateien (***.h) keine Doppeldeklarationsversuche auftreten, wird anfangs abgefragt, ob diese schon aufgerufen wurden:

Listing 9: Typische Header-Datei

```
#ifndef _IMAGE_H
#define _IMAGE_H
 ...
#endif
```

Definiert man ein Feld array [20][30] und möchte dies an eine Unterfunktion übergeben, so muss man zusätzlich die Zeilenlänge (und bei mehr Dimensionen auch die weiteren) übergeben, damit der Compiler damit arbeiten kann:
double max_array (**double** a[][30] , **int** m , **int** n) { ... } , der Aufruf: **double** y = max_array(array, 5, 7);
Die Angabe der Zeilen ist dabei optional und hat keine Auswirkung (nur wegen der Lesbarkeit...).

Möchte man dies umgehen, so kann man auch einen Pointer auf einen Vektor mit Pointern auf die Zeilenanfänge der Matrix übergeben (auf höhere Dimensionen verallgemeinerbar); man muss nur vorher diesen Vektor erstellen:

Listing 10: Pointer-Vektor

```
double* matrix[20];                          // Pointervektor
for (int i = 0; i < 20; i++)
        {
        matrix[i] = array[i];                // siehe unten
        }
```

Dabei ist „array[i]“ äquivalent „&array[i][0] “ und gibt den Pointer auf den Anfang der i-ten Zeile zurück.

Funktionspointer: **int** (∗fp)(**int**, **double**)
; vorne Ausgabeargumentetyp, hinten Eingabeargumententypen.
fp++, ∗fp, ... haben keinen Sinn und sind daher nicht erlaubt.

Unterprogramm:
void double_it(double∗ ptr_x) ∗ptr_x ∗= 2.0;
Hauptprogramm:
y = 10; double_it(&y);

Damit können auch Funktionen an Unterfunktionen übergeben werden:
double extreme_value(**double** a, **double** b, **double** (∗fp)(**double**)) { ...; c=fp(a); ...; }.

Feld von Funktionspointern: **double** (∗fp[42])(**double** x[], **int** n);

Steuerung von Varianten in Funktionen - Bitmasken:

Header-Deklaration: **#define** CASE1 1
#define CASE2 (1<<1)
#define CASE3 (1<<2)

Unterprogramm:
```
int prog (double param, unsigned int mode)
{
if(mode & CASE1) { ... }
if(mode & CASE2) { ... }
if((mode & CASE3) && (mode & CASE2)) { ... }
}
```

Programmaufruf: prog(param, CASE1);
prog(param, CASE2)
prog(param, CASE2 | CASE3)

int main(**int** argc, **char**∗ ∗argv) im Programmaufruf übergibt der Funktion die Anzahl Parameter (argc), die mit den Pointern ab ∗argv referenziert werden.

Im Terminal der Aufruf: > Prog1 Param1 Param2 ... ; wobei argv[0] standerdmäßig der Pointer auf den Programmnamen ist.

(getopt)

Bewertung von Algorithmen:
Zeitkomplexität: $g(n)$ - n Anzahl der Eingabeparameter, g die benötigte Zeit.

Bezeichung:
$$g \in \mathcal{O}(f) \quad \Leftrightarrow \quad \exists n_0 > 0, \exists c > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)$$
$$g \in \Omega(f) \quad \Leftrightarrow \quad \exists n_0 > 0, \exists c' > 0 : \forall n \geq n_0' : g(n) \geq c' \cdot f(n)$$
$$g \in \Theta(f) \quad \Leftrightarrow \quad g \in \mathcal{O}(f) \wedge g \in \Omega(f)$$

Typischerweise ist die Vergleichsfunktion f folgender Gestalt:

1	konstante Laufzeit
$\log(n)$	logarithmische Laufzeit
n	lineare Laufzeit
$n\log(n)$	fast-lineare Laufzeit
n^2 / n^3	quadratische / kubische Laufzeit
n^4, n^5, ...	polynomiale Laufzeit
a^n	exponentielle Laufzeit ($a > 1$)

($n!$, n^n, ... zu schlecht.)

Zusätzlich wird zwischem dem „best case“ , „average case“ und dem „worst case“ unterschieden.

($\mathcal{O}(n)$ - Skalarprodukt; $\mathcal{O}(n^3)$ - Matrixmultiplikation; $\mathcal{O}(n^2)$ - Selection Sort, Insert Sort;
$\mathcal{O}(n)$ - sequenzielles Suchen; $\mathcal{O}(\log(n))$ - binäres Suchen; $\mathcal{O}(n\log(n))$ - Merge-Sort.)

Datenstrukturen:
Listen-Charakteristika: einfach verkettet, doppelt verkettet, linear, zyklisch, ...

Bäume:

1. Mehrwege-Baum
2. Binärbaum: höchstens 2 Nachfolger je Knoten.
3. vollständiger Binärbaum: Binärbaum, dessen letzte Ebene „von links gefüllt“ ist.

Bei vollständigen Binärbäumen (oder welchen mit entsprechend benannten Knoten) gilt:

1. Vorgänger von Knoten n: $\left[\frac{n}{2}\right]$ (ganzzahlige Division)
2. Nachfolger von Knoten n: $2n$, $2n+1$.

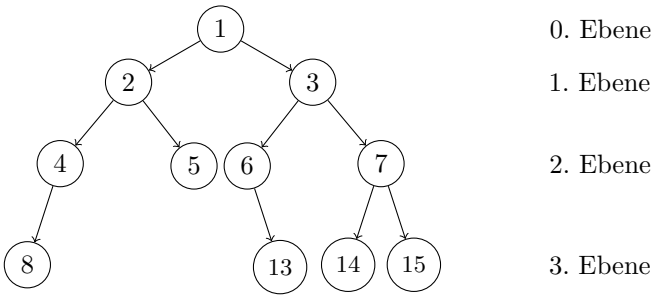


Abbildung 1: Binärbaum

gerichtete Bäume: Knoten mit Eingangsgrad 0 heißen **Wurzel**, Knoten mit Ausgangsgrad 0 heißen **Blätter**.
ungerichtete Bäume: Knoten mit Grad 1 heißen **Blätter**, alle anderen **innere Knoten**.

C Befehle

- Variablentypen: **int**, **double**, Image
- printf("control␣string", argument list);
- scanf("control␣string", &argument list);
- InfImgFile(name, dimx, dimy, ch);
- name = NewImg(dimx, dimy, 255);
- name = NewImg(pic, 0/1);
- ClearImg(name);
- Show(ON,name,"..."); / Show(OFF,name);
- GetVal(name, x, y); , PutVal(name, x, y, value);
- FreeImg(name);, ReadImg(filename, name);, WriteImg(name, filename)
- GetChar();
- Show(OVERLAY,pic,mark,"Schwerpunkt"); Marker(DEFAULT,sx,sy,farbe,groesse,bild);
- **#include** <math.h> ⇒ x = sqrt (...);
- **#include** <stdio.h> ⇒ d=InputD("\nText:␣");, i=Input("\nText:␣"); Printf("\nText:␣");
- GetChar()
- (type)(expression)
- **sizeof**()
- condition ? expression1 : expression2
- **int** sprintf = ("control␣string", argument list);
- **void**∗ malloc(m ∗ bytes);