Zahlensysteme:

unsigned int : 65535

Binär	0,5	0,25	0,125	0,0625	0,03125	0,015625	0,0078125	0,00390625

Hex: $0xA_H=10_D$

Benötigte Stellen m für Zahl in Basis B: $m=\lfloor \log_B Z \rfloor+1$

1-er Komplement: Invertieren+VZ — Größte Zahl, mit m Stellen Basis B darstellbar: $Z_{max}=B^m-1$

2-er Komplement: Invertieren+VZ,dann 1 addieren

IEEE 754 - *S tan dard* :

Anzahl Werte : B^m

$$Z=a_n\cdot B^n+a_{n-1}\cdot B^{n-1}+\ldots+a_1\cdot B+a_0$$

Null : Exponent: 0...0, Mantisse: 0...0

Nicht normalisierte Zahlen : Exponent: 0...0, Mantisse: "nicht normalisierte Zahl"

Unendlich : Exponent: 1...1, Mantisse: 0...0 VZ:

Not a Number (*NaN*) : Exponent: 1...1, Mantisse: nicht 0...0

Wert : $W=(-1)^S\bullet(1+M)\bullet 2^E$

Festkommazahl: *Oktal* : 3 *Binärzahlen*

– bleibt als – *Hex* : 4 *Binärzahlen*

Exp.=0

	32Bit	64Bit
	1	1
	8	11
Gleitkommazahl:	23	52

Exponent(Charakteristik): +127

Mantisse normiert: 1,... (1 vor Komma wird nicht abgespeichert ⇒ mehr Genauigkeit)

Zahl : Kleinster darstellbarer Wert: $-(2^{e-1}-1)$

Größter darstellbarer Wert: $+(2^{e-1}-1)$

Doppelte Darstellung der Null: $1/0-0=0/0-0$

1- *Komplement* : Kleinster darstellbarer Wert: $-(2^{e-1}-1)$

Größter darstellbarer Wert: $+(2^{e-1}-1)$

Doppelte Darstellung der Null: 1111 = 0000

2 - *Komplement* : Kleinster darstellbarer Wert: $-(2^{e-1})$

Größter darstellbarer Wert: $+(2^{e-1}-1)$

Keine doppelte Darstellung der Null!

Boolesche Algebra:

UND: • ∧ (konj.)

ODER: + ∨ (disj.)

NEG vor UND vor ODER!!!

Kommutativ : $a\bullet b=b\bullet a;\ a+b=b+a$

Assoziativ : $(a+b)+c=a+(b+c)$

$(a\bullet b)\bullet c=a\bullet(b\bullet c)$

Absorbtion : $a\bullet(a+b)=a;\ a+(a\bullet b)=a$

Distributiv : $(b+c)\bullet a=(b\bullet a)+(c\bullet a)$

$(b\bullet c)+a=(b+a)\bullet(c+a)$

DeMorgan : $\overline{(a+b)}=\bar a\bullet\bar b;\ \overline{(a\bullet b)}=\bar a+\bar b$

NeutralesElement : $a\bullet 1=a;\ a+0=a$

Komplement.Element : $a\bullet\bar a=0;\ a+\bar a=1$

Involution : $\bar{\bar a}=a$

Idempotenz : $a\bullet a=a;\ a+a=a$

Domienanz : $a\bullet 0=0;\ a+1=1$

UMWANDLUNG : *Zuerst Distributivgesetz*!!!

KV-Diagramm:

	y	∧y	y	
x	1	1	1	1	z
∧x	1	1			
x	1	1		1	∧z
	w	∧w		

∗ *DNF* : *Funktionswert*1!

Bsp.: $(x\bullet y)+(\bar x\bullet\bar y)$

0-en negieren!

KNF : *Funktionswert*0!

Bsp.: $(x+y)\bullet(\bar x+\bar y)$

1-en negieren!

KNF erstellen :

$$(x+y)=(x+y)+(z\bullet\bar z)$$

$f_1(x,y)=0$ Nullfunktion

$f_2(x,y)=x\cdot y$ UND

$f_3(x,y)=x\cdot\bar y$ Inhibition

$f_4(x,y)=x$ Identität in x

$f_5(x,y)=\bar x\cdot y$ Inhibition

$f_6(x,y)=y$ Identität in y

$f_7(x,y)=(x+y)\cdot\overline{(x\cdot y)}$ xOR Antiv

$f_8(x,y)=x+y$ ODER

$f_9(x,y)=\overline{x+y}$ NOR

$f_{10}(x,y)=(x\cdot y)+\overline{(x+y)}$ Äquivalenz

$f_{11}(x,y)=\bar y$ Negation von y

$f_{12}(x,y)=x+\bar y$ Implikation $(y\to x)$

$f_{13}(x,y)=\bar x$ Negation von x

$f_{14}(x,y)=\bar x+y$ Implikation $(x\to y)$

$f_{15}(x,y)=\overline{(x\cdot y)}$ NAND

$f_{16}(x,y)=1$ Einsfunktion

Automatentheorie:

Zustandsübergangsgraph:

MealyAutomat : Ausgabe bei Zustandsübergang

MooreAutomat : Ausgabe im Zustand

Deter min istischerAutomat : Nur eine Übergangsmöglichkeit!

EndlicherAutomat : endliche Menge von Symbolen (Eingang/Ausgang)

Kellerautomat : enthält Speicher (Stack) um Zwischenergebnisse zu speichern (Ergebniszwischenspeicher)

[Oberstes Element ist zusätzliches Eingangssignal]

⊗Muss immer einen Haltezustand haben!!!

Turingmaschine :

endlicher Automat (Steuerautomat mit Steuerfunktion) & einfacher Schreib/Lesespeicher

(Kann alle bisher bekannten Algorithmen durchführen)

Muss immer Haltezustand haben!

Zum Zustandsübergang tragen aktuell gelesener Wert & innerer Zustand bei!

Steuerautomat: anderen Zustand wechseln; Zeichen an akt. Bandposition schreiben

um eine Position links (l) oder rechts (r) oder gar nicht (ε) bewegen

7 – *Tupel* :

$TM=\{Z,A,I,b,Z_0,Z_F,S\}$

$Z\triangleq$ Menge aller Zustände

$A\triangleq$ Menge aller Symbole

$I\triangleq$ Menge aller Eingabesymbole

$b\triangleq$ Leerzeichen

$Z_0\triangleq$ Anfangszustand

$Z_F\triangleq$ Endzustand

$S\triangleq$ Steuerfunktion $S: A\times Z\to A\times B\times Z$

(Zustandsübergangsfunktion)

$B=$ *Bewegungen* $\{r,l,\varepsilon\}$

Kellerautomat :

$(7-Tupel)M=\{Z,T,V,p,V_0,Z_0,F\}$

$Z\triangleq$ Menge aller Zustände

$T\triangleq$ Menge aller Eingabesymbole

$V\triangleq$ Kelleralphabet

$p\triangleq$ Arbeitsfunktionen $Z\times(T\cup\{\varepsilon\})\times V^*$

$V_0\triangleq$ Startsymbol im Keller

$Z_0\triangleq$ Startzustand

$F\triangleq$ *Finalzus tan d*

POP = Lesen&entfernen oberstes Element

PUSH = Symbol schreiben

Zus tan dsübergangsgraph :

ODER werden durch Komma ausgedrückt!!!

Start

Halt

| *ODER*

■ *ENDE*

Re *kursion* :

Bezeichner ::= Buchstabe | Bezeichner Buchstabe

Levenshtein – Alg orithmus

Berechnet Anzahl Zeichen, die verändert werden müssen, um von einer s zur anderen Zeichenkette t zu kommen.

1.*Schritt* : Matrix (m+1)*(n+1) erstellen; s hat n , t hat m Zeichen

2.*Schritt* : erste Zeile mit 0-n; erste Spalte mit 0-m auffüllen

3.*Schritt* : erstes Zeichen von s mit allen zeichen von t vergleichen

s[i]==t[i], cost=0

s[i]!=t[i], cost=1

4.*Schritt* : In Zelle das Minimum setzen von {obere Zelle + 1; linke Zelle +1; linke obere Zelle + cost}

5.*Schritt* : Unterschied beider Wörter in unterster letzter Zelle!!

Algorithmen (Berechnungs- & Bearbeitungsvorschriften):

Rekursionsgleichung:

$$T(n)=\begin{cases}b & \text{für } n=1\\ aT\left(n/c\right)+bn+k & \text{für } n>1\end{cases}\Rightarrow T(n)=\begin{cases}O(n) & \text{für } a<c\\ O(n\log n) & \text{für } a=c\\ O\left(n^{\log_c a}\right) & \text{für } a>c\end{cases}$$

ZIEL : Minimierung {Zeit; Speicher} Komplexität

ACHTUNG:Aufwand nicht größer als Gewinn!

Bei Analyse Überdeckung von Variablen beachten!

Komplexität :

$T(n)\triangleq$ *Zeitkomplexität*

$S(n)\triangleq$ *Speicherkomplexität*

AsymptotischeKomplexität : Komplexität für große n

$[\Rightarrow$ Obere Schranke: $T(n)=O(f(n))]$

$n\Rightarrow$ *linear*

$n^2\Rightarrow$ *polyno min al*

$n\log n\Rightarrow$ log *arithmisch*

$2^n\Rightarrow$ exp *onentiell*

Re *chenregeln* :

$O(f(n))+O(f(n))=O(f(n))$

$c\bullet O(f(n))=O(f(n))$

$a\triangleq$ *AnzahlTeilprobleme*

$b\triangleq$ *Zeitschrittefür* $n=1$

$c\triangleq 1/$ *GrößeTeilprobleme*

H int ereinander : $T_1(n)+T_2(n)=O(\max f(n),g(n))$

[Immer größtes relevant!]

VerschachteIteSchleifen : $T_1(n)\bullet T_2(n)=O(f(n)\bullet g(n))$

nicht rekursiver Unterprogrammaufruf : Von unten nach oben auswerten! Aufruf mit Laufzeit für Unterprogramm bewerten!

Re *kursion* :

Funktion, die sich selbst direkt (indirekt) aufruft

Iteration :

Wiederholte Ausführung identischer Programmschritte innerhalb einer Schleife!

Divide & Conquer : Aufteilen in Teilprobleme & Vereinigung Teilergebnisse (Aufwand nicht größer Gewinn)

Balanzierung : Teilprobleme in etwa gleich groß

Dynamische Pr ogrammierung : Problem der Ordnung n in nTeilprobleme mit Ordnung (n-1) aufteilen. Start bei kleinenn in Richtung größere. ⇒ Vorgang aufteilen nicht größer als Gewinn

Modularisierung :

Einzelne Teillösungen (Bausteine)

⇒ Zusammenfügen

(Module unabhängig!!)

Testen :

Testdaten {repräsentativ; ökonomisch; fehler int ensiv}

BlackBox : Ohne Kenntnis des Alg.

WhiteBox : Struktur wichtig! Alle Strukturen (Verzwei.) einmal getestet!

Entwurf :

TOP – DOWN – Entwurf :

Erst Abstrakt dann Verfeinerung bis nur aus zur verfügungstehenden Ablaufstrukturen & Elementar-operationen besteht.

BOTTOM – UP – Implementierung: Zusammenfügen von Teillösungen!

Eigenschaften :

- Ausgabe in Relation zur Eingabe
- Effizienz!! (Abhängig von Kontroll- & Datenstruktur) Gleiches Resultat weniger Aufwand ⇒ effizienter!!
- In endlicher Zeit ausführbar
- Bestimmtheit jedes Schrittes
- In endlichen Schritten beendet
- Regeln vollständig (auch Fehler)

$D(n)$ Zeitkomplexität Divide|$C(n)$"Conquer|$T(n)$ Gesamtkomplexität

a=Anzahl Teilprobleme|n_i=Größtes i-tes Problem

Formale Sprachen & Grammatiken:

C hom sky – Hierachie :

Typ3: Reguläre Grammatik

Typ2: Kontextfreie Grammatik

Typ1: Kontextintensive Grammatik

Typ0: halbentscheidbare Grammatik

Freiheit ↓

Wort beliebig lang, aber nicht unendlich lang!!!

$G(L)=\{T,N,S,P\}$

$T\triangleq$ Terminale (Alphabet)

$N\triangleq$ Nichtterminale (Grammatik)

$S\triangleq$ Satzelement/Startelement $S\in N$

$P\triangleq$ Produktionsregeln

[Terminale in P: <...> | "..."]

Typ3 : Links: genau ein Nichtterminal

Rechts: Terminale und ein Nichtterminal [NT links $\triangleq$ linkslinear; NT rechts $\triangleq$ rechtslinear] | ε

Überprüfen mit endlich-deterministischem Automat A={T,N,F,S,P}

Typ2 : Links: genau ein Nichtterminal

Rechts: bel. Kmbination von Terminalen & Nichtterminalen | ε

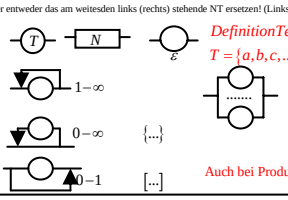
Überprüfen mit Kellerautomat (S → Wort: topdown; Wort → S: bottom-up) Analyseweg speichern!!

Typ1 : uAv::=uwv {w $\triangleq$ beliebige Kombination Terminale Nichtterminale; Kontext u,v muss erhalten bleiben, kann auch leer sein! keine Wortverkürzungen (kein ε); Endzustand nur wenn keine Nichtterminale mehr vorhanden!!!}

Typ0 : {Keine Wortverkürzungen (kein ε) Endzustand nur wenn keine Nichtterminale mehr vorhanden!!!; Semi-Thue-System: Regeln nur in eine Richtung! Thue-System: beide Richtungen!!!}

Eindeutigkeit : Immer entweder das am weitesden links (rechts) stehende NT ersetzen! (Links-/Rechtserzeugung)

EBNF:



DefinitionTer min ale :

$T=\{a,b,c,\ldots,0,\ldots,9,LZ\}$

Auch bei Produktionsregeln!!!

Datenstrukturen:

Datenelement (*Datensatz*):

Ein Datenelement besteht aus mehreren aufeinander folgenden Wörtern im Speicher, die in benannte Teile eingeteilt sind. Die benannten Teile eines Wortes werden als Felder bezeichnet. Im einfachsten Fall ist ein Element nur ein Wort im Speicher und hat nur ein Feld. ∧ ≙ Nullzeiger (ENDE)

X : Name einer Variablen

$pX=Loc(X)$:Zeigervariable pX

$Content(X)$:Inhalt des Speichers an Stelle X

$Loc(X)$: Adresse im Speicher an der X gespeichert ist

Nächste$(A)\Rightarrow$1Bit des auf A folgenden Datensatzes

Loc$(A)\Rightarrow$1Bit des Datensatzes A

Underflow : Versuch nicht vorhandenes Element zu lesen.(nicht bösartig)

Overflow : Versuch Information einzutragen in bereits volle Struktur.(bösartig) Informationsverlust!!! Ende Zeichen wird überschrieben!

SpezielleListen :

Schlange : verk.Liste; Elemente am Anfang entf. & am Ende eingefügt

Ringpuffer : seq. Liste; älteste Daten werden jeweils überschrieben

Stack : seq.Liste;Nur am Ende hinz., entf.

Sequentielle Listen : Reihenfolge im Speicher entspricht logischer Folge (folgen direkt aufeinander). Speicherverschwenderisch, da immer gesamter Speicher reserviert werden muss. Datensätze finden sehr einfach, da direkt aufeinanderfolgen (Über TOP Zeiger berechenbar) Schneller,weniger Speicher pro Datensatz! Einfügen/Entfernen nur am Ende. SpeicherBlock!!!

Verkettete Listen : TOP Zeiger auf ersten Element; Zeiger in jedem Datensatz, der auf nächstes Element zeigt (erstes Bit davon) flexible Speichernutzung.

ABER: Pro Zeiger 4 Byte zusätzlich!

Einfügen/Löschen sehr einfach (dynamisches Wachsen)

1 Zeichen entspricht einem Byte! ACHTUNG:

Für 50 gültige Zeichen werden 51 Zeichen benötigt,

da das ENDE Zeichen "/0" enthalten sein muss!!

E inf ach verkettete Listen : Listeneintrag kennt nur Nachfolger

Vorgänger kann nur mit nochmaligem Durchlauf von Anfang an der Liste gefunden werden. Bewegung/Suchen nur in eine Richtung.

Doppelt verkettete Listen : Listeneintrag kennt Vorgänger und Nachfolger

Bewegung in beide Richtungen möglich, aber aufwendigere Operationen zum Einfügen/Löschen (mehr Zeiger umbiegen) Links und Rechts sind Zeigervariablen, die auf das linke bzw. rechte Element zeigen.

(Jedes Element enthält zwei Zeiger) Vereinfachung des Durchlaufs in beide Richtungen, aber zusätzlicher Speicher nötig.

```
struct Name{
    Variable1;
    ....
    struct Name *next;
    struct Name *previous;
};
```

Neues Element nach *previous einfügen!

struct Klausur *neu = (*struct* Klausur*) *malloc*(*sizeof*(*struct* Klausur));

neu → next = previous → next;

neu → previous = previous;

previous → next = neu;

neu → next → previous = neu;

Kein Ende bei Durchlauf????

Operationen : [*e* inf *achverkettet*]:

NeuesE inf *ügen* : 1.Pointer bei neuem Datensatz auf das darauffolgende Element legen.

2.Zeiger von Datensatz davor umbiegen

Re *ihenfo* lg*e ändern* : 1.1Bit des neuen ersten Datensatzes zwischenspeichern (Temp)

2.Zeiger in altem ersten Datensatz auf neuen darauffolgenden ändern

3.Zeiger im neuen ersten Datensatz auf neuen darauffolgenden ändern

4.TOP Zeiger auf neuen ersten Datensatz umbiegen (Auslesen aus Temp)

Operationen : [*doppeltverkettet*]:

NeuesE inf *ügen* : Vorwärtsrichtung:

$Nächste(NEU)=Nächste(X)$

$Nächste(X)=Loc(NEU)$

Rückwärtsrichtung:

$Vorige(NEU)=Vorige(Y)$

$Vorige(Y)=Loc(NEU)$

RETRIEVE (p,L):Gibt Element an Position p der Liste L zurück

DELETE (p,L) : Löschen des Elements an der Position p der Liste L

NEXT (p,L):Gibt Position des Elements zurück, das nach Element an der Position p der Liste L ist

PREVIOUS (p,L):Gibt Position des Elements zurück, das vor Element an der Position p der Liste L ist

MAKENULL (L) : Initialisiert leere Liste

FIRST (L):Erste Position in der Liste L

PRINTLIST (L):Druckt Elemente der Liste L in der Reihenfolge ihres Auftretens

Graphen:

Darstellung von Relationen (symmetrisch(=ungerichtet), asymetrisch(=gerichtet))

Grad: Anzahl der Ein- & Ausgehenden Kanten {gerade ⇒ gerade Anzahl Kanten; ungerade ⇒ ungerade Anzahl Kanten}

Nachbarschaft: Knoten mit gemeinsamer Kante $N(v)=\{w\in V\mid(v,w)\in E\}$

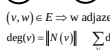
Gerichteter Graph: Ein-/Ausgangsgrad, Vorgänger, Nachfolger

Zusammenhängende Teilgraphen werden als Komponenten bezeichnet

Graph $G=(E,V)$

$E\triangleq$ *Kanten*(*Edges*)[Paare von Knoten (v,w)]

$V\triangleq$ *Knoten*(*Vertices*)



$(v,w)\in E\Rightarrow$ w adjazent (nachf.) zu v

$deg(v)=|N(v)|\quad\sum_{v\in V}deg(v)=2\cdot|E|$

Nachbarschaft :

Vorgänger(N^-) + *Nachfo* lg*er*(N^+)

$deg(v)\triangleq$ Grad von v

Minimum Spanning Tree : Minimales Gerüst um alle Knoten zu verbinden! (Pfadlänge beachten)

Cliquen in Graphen : Vollvermaschte Teile eines Graphen

Pfad : Folge von Knoten (Anfang ' Ende)

Länge des Pfades: n-1 (Anfangs- zu Endknoten)

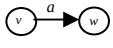
Durch Folge der Knoten oder Kanten beschrieben

Länge des Pfades ist Summe der Gewichte

Schleife / Zyklus : Pfad im Graph (Beginnt & Endet am selben Knoten)

Länge: min 1 (Bei ungerichtet: min 3)

In Adjazenzmatrix/-liste Gewichte anstelle von "1" eintragen



Adjazenzmatrix :

Dimension $|V|\times|V|$

Bestehend aus 0-en und 1-en

Element A[i,j] hat Wert 1, wenn Kante von Knoten i zu j vorhanden ist.

von \ nach		

Speicherplatz von $|V|^2$ nötig (auch bei wenig Kanten)!

Zeit um zu bestimmen ob eine Kante vorhanden ist ist unabhängig von $|V|$ und $|E|$

Adjazenzliste :

Verwenden wenn $|E|\ll|V|$

$|V|$ Listen für einen Graphen (pro Knoten eine Liste)

Speicherbedarf: $|V|+|E|$

i	Head	Next
1	A	4
2	B	6
3	C	
4	B	5
5	C	
6	C	

Bäume:

Wichtigste Struktur für die Entwicklung von Computer Algorithmen

Eingangsgrad aller Kanten ist 1 (ausser Wurzel) ⇒ Grad = Ausgangsgrad!

Geordneter Baum: Reihenfolge Nachfahren relevant

Orientierter Baum: Reihenfolge der Kinder vertauschbar

Grad eines Knoten: Anzahl der Unterbäume

Endknoten (Blatt): Knoten ohne echten Nachfahren

Verzweigungsknoten: min. 2 Nachfahren

BinärerBaum :

Immer zwei Nachfahren (sonst Blatt)

Vollständig : alle Ebenen voll besetzt!!!

Fast vollständig : sämtliche Ebenen ausser der letzten voll besetzt; letzte von links her besetzt!!!

Höhe : k

Ebenen : $k+1$

Knoten : $2^{k+1}-1$

Blätter : 2^k

Höhenbestimmung :

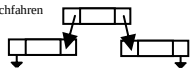
vonOben : Wurzel: 0

nächste Ebene: 1

Tiefe : Länge des Pfades von Wurzel aus

Höhe : Länge des längsten Pfades zu einem Knoten

HöheBaum : Höhe der Wurzel



Darstellung als lin.Anordnung :

Wurzel auf Position 1; Knoten auf Position i;

Position linker Sohn: $2\bullet i$

Position rechter Sohn: $2\bullet i+1$

Position Vater: $\lfloor i/2\rfloor$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	15	13	14	9	10	11	12						7	8	

Verwendung : Parser, Stammbaum, Heapsort, XML/HTML

Durchlauf :

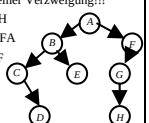
Vollständiger Durchlauf ⇒ lineare Anordnung der Knoten

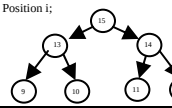
Achtung auf links/rechts bei nur einer Verzweigung!!!

PreOrder: W,L,R ⇒ ABCDEFGH

PostOrder: L,R,W ⇒ DCEBHGFA

InOrder: L,W,R ⇒ CDBEAGHF





Für Gleichungen InOrder!!!

FVBB :

Knoten : {*Max* : $2^{h+1}-1$; *Min* : 2^h}

C-Programmierung:

\n	Newline
\f	Formfeed
\a	Bell
\t	Tab
\'	Apostroph
\"	Anführungsz

Anweisungen:

printf("Text %<insg>.<Nachk><Format>", Variable);

scanf("%d", &Variable);

while(i==2){...} [0-∞]

do{...}while(i==2); [1-∞]

for(x=0;x<10;x++){...}

if(x==0){...} else{if(x==1){...}} && = UND || = ODER

switch (*Variable*){case Wert: Anweisung; break; ... default: Anweisung; break;}

#include <...>

int main (){...};

/*Kommentar*/

ASCII

A ≙ 0x41

K ≙ 0x4B

LZ ≙ 0x20

festeZeichenlänge!!

0xkannWeglassen!!!

Datentypen:

const int	Konstanten
static int	statische Varibale (bleibt in Funktionen erhalten)
int	neg,pos ganze Zahlen
long int	32 Bit
short int	16 Bit
unsigned int	max. 0-65535
float	GKZ 16 Bit
double	GKZ 32 Bit
char	Zeichen 8 Bit

char *arrv[3]; Feld mit 3 Elementen je ein Zeiger auf char

string Speicherung in char-Feld

int a[n] [k] int Datenfeld (Array) 2D

struct Datum{ int Tag;...} Struktur (Zugriff: d1.tag)

struct Datum d1,d2;

enum Tag {Mo,Di,...} Aufzählung, wobei

enum Tag t; t nur Werte aus Tag annehmen kann. Mo=0

%d: verzeichenb. Integerzahl

%x: hex Integerzahl

%o: okt. Integerzahl

%u: unsigned Int

%ld: long

%lu: unsigned long

%f: double,float (scanf ⇒ *%lf*)

%e: GKZ in Expdarst.

%c: Zeichen char

%s: strings

Funktionen(Unterprogramme):

void Name (int *n*, int *t*){...*return*;}

double Name (int *n*, int *t*){...*return* Var;}

In Funktion *n* & *t* verwenden!

Unbedingt vorher deklarieren: *double* Name (int *n*, int *t*);

Implementierung kann später erfolgen!

ACHTUNG: Variablenüberdeckung!!!

[lokale Variablen gehen verloren!]

Funktionen in Bibliotheksfunk. enthalten;

Für Deklaration Headerfiles!!! (enthalten nur Deklarationen; Funktion wird über Linker hinzugefügt)

Module:

- Programm besteht aus mehreren Modulen
- Module separat vom Hauptprogramm-Modul erstellbar
- Änderungen der externen Module separat machbar
- Bei gleicher Schnittstelle sind Module beliebig austauschbar
- Schnittstellen (globale Variablen, Typendefinitionen, Funktionsdek.)

Zusammenfuhren von Modulen: (Übersetzung in mehreren Schritten)

- 1. Jedes Modul einzeln kompilieren ⇒ *.o ObjektDatei
- 2. Objektdateien zu ausführbaren Dateien linken (Makefile!!!)

Pointer(Zeiger):

Variable in der die Adresse einer Speicherzelle abgelegt ist

& : *Adressoperator*

* : *Zeiger – / Dereferenzierungsoperator*

Deklaration: int *iptr;

Zuweisung: iptr=&i; (iptr ist Zeiger auf i)

⇒ Wertzuweisung: *iptr=123; oder i=123;

ACHTUNG:

iptr1=iptr2; ⇒ Beide Zeiger zeigen auf gleiche Bezugsvariable

*iptr1=*iptr2; ⇒ Bezugsvariable iptr1 wird Wert von iptr2 zugewiesen

Zeiger & Felder:

Zugriff auf Feldelemente auch über Zeiger;

iptr= feld (iptr=&feld[o]) ⇒ *(iptr+i) Wert von feld[i]

char c='...';

int a = (int) c;

a ≙ ASCII Wert der Var. c

int *Variable; (=ZeigerVariable)

*Variable=123; (=Wertzuweisung)

Variable=&a; (=Variable ist Zeiger auf a)

a=123; (=Wertzuweisung)

Zeiger & Array:

int feld[20], *pi;

pi=feld; {pi=&feld[0]}

*pi ≙ 1.Feldelement(feld[0])

*pi++ ≙ 2.Feldelement(feld[1])

Zeiger & Struct:

struct varst{.Teil.};

struct varst *Variable;

Variable->Teil=...; (=Zuweisung)

Zeiger & Datenstrukturen:

Speicher reservieren: int *p; p=malloc(sizeof(int));

Dynamische Variable der Größe int, auf die p zeigt;

Ermittlung benötigter Platz: sizeof(Typ od. Variable)

Speicher wieder frei geben: free(p); Speicherplatz der Variable auf die p zeigt wieder freigeben.

ACHTUNG: Nur einmal wieder freigeben!!!

Übergabe:

DynamischeDatenstrukturen:

int *p;

p=malloc(sizeof(int));

*p=23;

Freigeben:

free(p);

Zeiger & Funktionen:

Verwendung für Ein- & Ausgabe; Funktion kann auf Daten im aufrufenden Programm zugreifen und sie verändern;

void funktion(int *iptr){...};

int main ( ) {funktion(&wert);}

DatenZuweisen:

sprintf(newRec → name,"%s","Maus");

sprintf(newRec → name,"%s","Maus");

Datenkompression:

Datenkompression:

- Digitalisierung von Signalen
- Mittel zur effizienten Übertragung, Speicherung, Verarbeitung

Kompressionsarten:

⊗Verlustlose Kompression:

Codierung mit variabler Wortlänge (wahrscheinlichere Symbole kurze Wortlänge, weniger wahr. längere Wortlänge

⇒ Mittelwert der Codewortlänge wird minimiert)=Huffman

Dekompression ergibt genaue Originaldaten

ACHTUNG: Kann auch zu Datenexpansion führen

Bsp.: Morse,Huffman

⊗Verlustbehaftete Kompression:

Nicht invertierbar ⇒ Informationsverlust (Bei ausreichend Bits nicht spürbar) ⇒ höhrer Kompression erzielbar

ZIEL: Max Qualität bei geg. Bitrate, Min Bitrate bei geg. Qualität (Rate Distortion Theorie)

Bsp.: ATRAC, MPEG

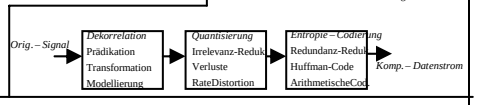
Kompression: $L \geq H$!!!!

Kompfaktor: Längen Symbole vergleichen (L)!!!

L<H ⇒ Verlust

Kriterien:

- Kompressions-Effizienz
- Mittlere Verzerrung
- Komplexität
- Speicherbedarf
- Verzögerung, Latenz
- Störanfälligkeit



Sortieren & Sortieralgorithmen:

Stabiles Sortierverfahren:

Datensätze mit gleichgroßen Elementen behalten ihre Reihenfolge bei!!!

Suchen:

seq.Liste: durch Teilen der Liste

verk.Liste: linearer Durchlauf

Binärbaum: wie beim Sortieren

Selektion - Ordnungsstatistik:

Aus Folge von n Elementen das Element mit Rang k heraussuchen (k-größte Element)!

Für n >1400 ist Sortieren und Auslesen zu langsam!

Partitionierung über Quicksort: Für n Elemente n Vergleiche

⇒ *BFPRT - Algorithmus* (Partitionierung):

Grundprinzip:

- Teilen der Folge in $\lceil \frac{n}{5} \rceil$ Gruppen von max 5 oder min 4 Elementen
- Bestimme Median jeder Gruppe und sortiere diese relativ dazu (vgl. Quicksort)
- Bestimme Median der Mediane der $\lceil \frac{n}{5} \rceil$ Teilgruppen
- Bestimme Rang des Medians der Teilmediane m

Rekursives Vorgehen

$RangM = k$ ⇒ fertig

$RangM < k$ Verwerfe Elemente kleiner als M

$RangM > k$ Verwerfe Elemente größer als M

Elemente ≤ M

Elemente ≥ M

Mediane sind aufsteigend angeordnet von links nach rechts, von oben nach unten. Für das Auffinden des k-ten Elements werden max 25*n Vergleiche benötigt. $O(n)$

Median aus 3 ⇒ 3 Vergl.

Median aus 4 ⇒ 5 Vergl.

Median aus 5 ⇒ 7 Vergl.

Median aus 7 ⇒ 11 Vergl.

Hashing:

- Speicherfunktion
- Elemente einer Menge S werden einer endlichen Menge von Klassen B (0,1,...B-1) zugeordnet
- Abbildung auf kleinere Menge (nicht umkehrbar)
- HashFunktion h ordnet jedem x aus S eine Klasse zu
- Funktion muss leicht berechenbar & eindeutig sein
- Ist Menge S >> Klassen B ⇒ h(x) ist surjektiv (⇒ Kollisionen)
- Kollisionsfreiheit: streuende HashFunktion! kleine Änd → groß Untersch

Überprüfen ob Element in Hash-Tabelle:

Elemente dürfen nicht gelöscht werden. ⇒ Klassen werden sukzessiv überprüft, ist die Klasse nicht leer und nicht gleich dem gesuchten Element, wird die mit Rehash ermittelte nächste Klasse überprüft.

Elemente dürfen gelöscht werden. ⇒ In die Klasse wird nach dem Löschen ein Eintrag (Marke "gelöscht"), der für gelöscht steht eingetragen.

HashTabelle:

h(x)	x
0	b
1	
2	a
3	c

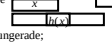
Zuweisung eines Elements x zur Klasse h(x). Bei Kollision ReHash-Funktion!

HashFunktionen:

- Quersumme über die Zeichen modulo B
- "Middle Bits" x hat 10Bits; x*x hat 20Bits ⇒ 5Bits in der Mitte von x*x ⇒ gut verteilte Werte und niedrige Kollisionsrate
- h(x)=x mod B (abhängig von der Wahl der B's:

B ist gerade Zahl ⇒ h gerade/ungerade wenn s gerade/ungerade;

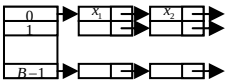
B ist Primzahl ⇒ gute Wahl)



Kollisionen:

- einfache ReHashFunktion: $h_i(x) = h(x) + i \bmod B$
- verkettetesHashing: x ist in Liste i, wenn h(x)=i

streuende Hashfunktion ⇒ jede Klasse enthält gleichviele Elemente = N/B



Anwendungen:

- Datenbankindex - schnelles Durchsuchen
- CRC Fehlererkennung (Polynom P(x) Generatorpolynom) Darf für benachbarte Datenworte die um weniger als 7 Stellen differieren nicht zur Kollision führen (=Prüfsummen)
- Kryptologie (digitale Signatur) Kollision muss praktisch unmöglich sein, einfache Berechenbarkeit

Bsp.:Einfachverk. Liste:

```
struct karte {
  char name[20];
  struct karte *next;};
struct karte *kopf, *ende;
int main(void) {
/* Anlegen des Listenkopf und -ende */
  kopf = (struct karte*) malloc(sizeof(struct karte));
  ende = (struct karte*) malloc(sizeof(struct karte));
  kopf->next = ende->next = ende;
  char string[20];
  struct karte *neu_name, *list_zgr;
  printf("\nGeben Sie die Namen ein (Ende mit Leerzeile)\n\n");
  while (1) {
    printf("Name: ");
    gets(string);
    if (strlen(string) == 0)
      break;
/* Anlegen eines neuen Elementes */
    neu_name = (struct karte*) malloc(sizeof(struct karte));
    strcpy(neu_name->name, string);
/* Finden der Position wo neuer Name einzufügen */
    list_zgr = kopf;
    while(list_zgr->next != list_zgr->next->next) {
      if (strcmp(string, list_zgr->next->name) <= 0)
        break;
      list_zgr = list_zgr->next;}
/* Einfügen des neuen Namens in die Liste */
    neu_name->next = list_zgr->next;
    list_zgr->next = neu_name;}
  printf("\n----- Sortierte Liste -----\n");
  while (list_zgr != list_zgr->next) {
    printf("%s\n", list_zgr->name);
    list_zgr = list_zgr->next;}
  return(0);}
```

Decodierung: Vorderstes Bit beginnen!

Präfix-Code:

- Kein Codewort darf Präfix von anderem sein
- Code kann mit binärem Baum repräsentiert werden
- Alle Codeworte liegen an Blättern des Baumes
- ≙ Fano Bedingung
- ⇒ Eindeutigkeit bei Decodierung

Codierung: Alph.1 → Alph.2

verlustfrei: ein Code zu einem Zeichen

eindeutig (Grenzen klar): -PräfixCode -feste Wortlänge, -Trennzeichen

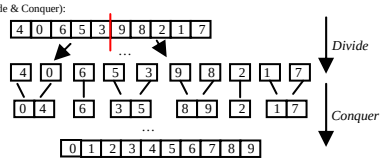
Mergesort:

Grundprinzip (Rekursiv):

Zusammenmischen von zwei sortierten Folgen in eine Folge (Reißverschluss)

Aufteilen in zwei unsortierte Folgen, sortieren der Teilfolge mit dem selben Prinzip (Rekursion)

(Divide & Conquer):

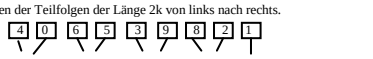


- Divide Schritt ist einfach
- Conquer Schritt benötigt viele Vergleiche
- Zusätzlich Speicher im Umfang der zu sortierenden Folge
- Maximal $O(n \log n)$ Vergleiche

Grundprinzip (Iterativ):

Man fängt direkt mit Mischen an, Aufteilung entfällt.

Mischen der Teilfolgen der Länge 2k von links nach rechts.



Quicksort:

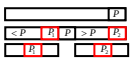
Aufteilen in Teilfolgen (Auswahl eines Pivotelements, immer letztes Element; Aufteilen in Teilfolge mit Elementen kleiner als Pivot und Teilfolge mit Elementen größer und gleich dem Pivot.

Rekursiv weiter durchführen.

- Sortieren der Teilfolgen mit dem selben Prinzip
- Zusammenfassen der sortierten Teilfolgen
- $O(n^2)$ aber unter Realbedingungen der schnellste $n \bullet \log n$
- Sortieren auf Speicherbereich der Liste

Durchschnittlich:

$V_{Quick} = 1,386 \cdot n \log n$



Algorithmus zur Bildung der Teillisten:

QUICKSORT(Liste)

if || Liste || = 1 then return: Liste

Wähle zufällig ein Element a aus der Liste

Setze die Merker i auf das erste und j auf das letzte Listenelement

while i ≤ j

while Element i < a bewege Merker i zum nächsten Element

while Element j ≥ a bewege Merker j zum vorigen Element

if i < j then

vertausche die Elemente i und j

bewege Merker i zum nächsten Element

bewege Merker j zum vorigen Element

return QUICKSORT(Teilliste(Anfang - Merker j)) +

QUICKSORT (Teilliste(Merker i - Ende))

Binärbaum:

Zur Speicherung wird wiederum eine Liste verwendet!

ACHTUNG: Wenn Elemente des Baumes fehlen, bleibt der Speicher leer, wird aber frei gehalten. ⇒ siehe Binärbaum!!!

$\begin{cases} \text{Verzweigung nach links, wenn } a_i < j \\ \text{Verzweigung nach rechts, wenn } a_i > j \end{cases}$

Bubblesort: (Iterativ)

Liste von oben nach unten abarbeiten; Dabei immer 1 mit 2, 2 mit 3, 3 mit 4,... vergleichen und der größe nach sortieren. Nach Durchlauf mit neu gewonnener Liste von oben wieder beginnen (selbes Verfahren)! Sehr einfach, aber auch sehr ineffizient! $O(n) = n^2$

Radixsort / Bucketsort: (Iterativ)

Endlicher Zeichenvorrat und endliche Anzahl von Zeichen; gut, wenn die Länge der Worte klein gegenüber ihrer Anzahl ist.

- Erzeuge leeren Stapel für jedes mögliche Zeichen des Vorrates
- Lese jedes Zeichen und lege es in den zugehörigen Stapel (Beim letzten Zeichen beginnen) Stapel von unten nach oben auffüllen!!
- Vereinige alle nicht leeren Stapel zu einer Liste (von unten nach oben nach rechts)

$O = ((m+n) \cdot k) \Rightarrow O(n) = n$

m ≙ Größe Zeichenvorrat

k ≙ Länge längstes Wort

n ≙ Anzahl Wörter

Informationstheorie:

Syntax ≙ Grammatik, formale Sprachen

Semantik ≙ Bedeutung von Symbolen

Pragmatik ≙ Zwischen den Zeilen

Quantitative Informationstheorie:

(Math. Beschr. & quant. Erfass. v. Informationen)

- Quellencodierung, Kanalcodierung, Kryptographie

Datenmenge ≠ Informationsgehalt

Informationsgehalt eines Symbols:

(Je seltener ein Symbol, desto größer die Information!!!)

$I(x) = ld \frac{1}{p(x)} [Bit]$ $p(x)$ ≙ *Wahrscheinlichkeit*

Elementevorrat:

$EV \leq B^s$ (Wörter mit s Stellen, Stelle kann B Werte annehmen)

Entscheidungsgehalt:

$EG = ld(EV)$ (Stellenanzahl der binären Repräsentation eines Symbols)

Entscheidungsredundanz:

$R = S - EG$ $S = \lceil EG \rceil$

mittlererInformationsgehalt:

$H = \sum_{i=1}^{n} p(x_i) \cdot ld \frac{1}{p(x_i)} = \sum_{i=1}^{n} p(x_i) \cdot I(x_i)$

mittlereCodewortlänge:

$L = \sum_{i=1}^{n} p(x_i) \cdot l(x_i)$ $(l(x_i)$ Länge Symbol $i)$

Coderedundanz:

$R = L - H$

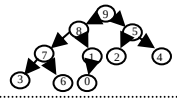
(*Shannon'sche Codierungstheorem*: $L \geq H$)

Heap: Fast vollständiger binärer Baum (die in einem Knoten gespeicherte Zahl ist nicht kleiner als die in seinen Kindern gespeicherten Zahlen) Werte der Knoten sind sortierbar. <Floyd 1964>

- Alle inneren Knoten, bis auf max. einen haben genau zwei Kinder
- Alle Knoten mit weniger als zwei Kindern befinden sich auf den größten beiden Leveln
- Die Blätter sind von links nach rechts aufgefüllt

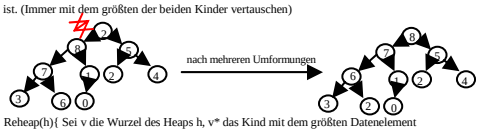
Grundoperationen:

- Reheap (Baum in Heap verwandeln)
- Delete_Max (Entfernen der Wurzel)
- Create_Heap (Aufbau eines Heaps)



Reheap:

Top-down Abarbeitung; Wurzel des Baumes wandert durch Vertauschen im Baum nach unten bis Heap-Eigenschaft erfüllt ist. (Immer mit dem größten der beiden Kinder vertauschen)



Reheap(h){ Sei v die Wurzel des Heaps h, v* das Kind mit dem größten Datenelement

While (Heap Eigenschaft nicht erfüllt)

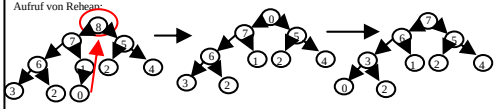
{Vertausche die Datenelemente in v und v*

Setze v=v*}}

Delete_Max:

Entfernen der Wurzel aus dem Heap; füllen des Loches mit rechtestem Blatt, rechtestes Blatt löschen;

Aufruf von Reheap:



Delete_Max(h){Sei r die Wurzel des Heaps h, l das rechteste Blatt

Kopiere l in die Wurzel r

Lösche l, dekrementiere heap_size

Reheap(h); }

Create_Heap:

Von den Blättern hin zur Wurzel; jedes Blatt erfüllt trivial die Heap Eigenschaft;

Immer eine Ebene von unten nach oben mehr;

Create_Heap(n,A){konstruiere fast vollständigen binären Baum H mit n Knoten

Heap_size=n

Fülle jeden Knoten mit einem Element von A

For (l=d(H):-1:1) {

For jedem Knoten auf dem Level l {

Reheap(Baum H mit Wurzel r)}}}

Datenstruktur für einen Heap: (lineares Datenfeld)

A[1] Wurzel des Baumes; A[i] eines Teilbaumes; A[2i] linker, A[2i+1] rechter Sohn Knoten i (Vertauschungen in der Liste)

i	1	2	3	4	5	6	7	8	9	10
	8	7	5	6	1	4	3	2	0	9

Heapsort:

Heap der Größe n auf einem Feld mit Indexpositionen [0,n-1] realisiert; Feld wird in zwei Teile geteilt

- Vorderer Teil: Heap (am Anfang ganzes Feld)
- Hinterer Teil: Aufbau der sortierten Folge

Anwendung von Create_Heap

Löschen des max. Elements

Einfügen der max. Elemente vor dem Anfang des sortierten Bereichs

Wiederherstellen der Heap Eigenschaft auf dem reduzierten Baum

KOMPLEXITÄT:

$V_{Reheap}(n,i) = 2 \cdot (\lfloor \log(n) \rfloor - \lfloor \log(i) \rfloor)$

$V_{Create}(n) \leq 5n$

$\Rightarrow V_{HS} \leq 2 \cdot n \log(n) + \frac{7}{2} n$

Heapsort benötigt $O(n \log(n))$ Vergleiche für n Elemente

4 zusätzliche Speicherplätze/Variablen nötig!

Verwendung: Prioritätswarteschlangen

CodeBaum:

- optimal, wenn keine freien Blätter auf niedrigeren Leveln (Optimierung durch verschieben!)

$ld3 = 1,585$

$ld5 = 2,322$

$ld7 = 2,807$

$ld9 = 3,170$

$ld11 = 3,459$

$ld13 = 3,700$

Huffman – Coding:

(es gibt Präfix-Code $L < H+1$)

1.Sortieren der Symbole entsp. Auftrittswahrsch.

2.Zwei Symbole mit geringst. Wahrsch. zu Hilfssymbol

3.Wahrsch. Hilfssymbol durch Addition

4.1-3 wdh. solange mehr als ein Symbol

5.CodeBaum (letzt. ist Wurzel); Symbole zuweisen (0/1);CodeTabelle (von Wurzel zu Symbol laufen)

Verb.: Durch Blockbildung, aber zu groß Alphabet; Redundanz bestimmt effizienz!