ZUSAMMENFASSUNG
GGI1
WS 2011/2012

DENNIS PÖTTER

Allgemein

- **Imperative Sprachen**: konkrete Beschreibung eines Algorithmus als Folge von Anweisungen
- **Deklarative Sprachen:** Beschreibung des erwünschten Programmverhaltens, konkrete Umsetzung erfolgt teilweise durch den Compiler und Laufzeitsystem
- **Objektorientierte Sprachen:** Zusammenfassung von Algorithmen und Daten zu Objekten, dadurch einfachere Programmwartung

- **Algorithmus:** Abstrakte Handlungsanweisung, um ein Problem zu lösen. In endlicher Zeit, eindeutig und deterministisch
- **Programm**: Codierung eines Algorithmus in Programmiersprache.
- **Datenstruktur:** Mathematisches Objekt zur Speicherung von Daten.

Ein **Problem ist per Computer lösbar** wenn die Problemlösung mathematisch exakt als Algorithmus beschreibbar ist.

Häufig wiederverwendeter Code wird in **Libraries** abgelegt. Library-Code wird nur einmal compiliert und kann dan von allen Programme eingebunden werden. Dies erfolgt nach der Assemblierung durch den Linker.

Die **main-Funktion** ist der Einstiegspunkt eines jeden C-Programmas. Steht an beliebiger Stelle, definiert immer eindeutig, wo bei der Programmausführung begonnen wird.

Variablen sind benannte Speicherplätze für Werte und müssen vor Verwendung deklariert werden.

C-Schlüsselwörter haben für den Compiler eine vordefinierte Bedeutung und weisen ihn an, bestimmte Code-Strukturen zu generieren.

Printf-Funktion ist in Standard-Library und gibt Daten auf dem Bildschirm aus. **Fprintf** gibt Daten in Files aus.

➔ Prinf-Funktion werden stehts zwei Parameter übergeben: 1. Art der Ausgabe 2. Die auszugebenden Daten.

Der **C-Präprozessor (cpp)** läuft vor dem Compiler über den Quellcode. Cpp-Anweisungen beginnen mit “#”.
Anwendungen:

➔ Symbolische Definitionen
➔ Makros
➔ Einbinden von Standard-Libraries

Übersetzung und Ausgabe:

1) Präprozessor und Assembler werden implizit aufgerufen
2) Zielprogramm “a.out” wird erzeugt
3) Ausführung von “a.out” generiert die gezeigte Ausgabe

In C sind Zuweisungen auch Ausdrücke. Ergebnis von Ziel = Audruck ist Ziel. Also ist a=b+c+(x=d+e) möglich.

Compile time: Zeitpunkt das der Compiler läuft
Run Time: Programm-Laufzeit

KAPITEL 3

3.1 Datentype

Ein **Datentyp (DT)** ist eine Menge von Werten und eine Sammlung von Operationen auf diesen Werten.

Ein **abstrakter Datentyp (ADT)** ist ein Datentyp der nur über eine Schnittstelle zugänglich ist. Implementierung bleibt verborgen. In C sind Daten un Operationen getrennt, daher keine tatsächlichen ADT möglich.

- Variablen, Konstanten und Programmcode werden im Speicher abgelegt.
- Ein Objekt ist ein benannter Speicherbereich
- Auf Objekte wird direkt (per Namen) oder indirekt (per Zeiger) zugegriffen. Zeiger sind selbst Objekte.
- Im Speicher befinden sich nur Binärzahlen. Interpretation ist den Operationen überlassen.

Der benötigte Speicherplatz eines Objektes sowies die darauf zulässigen Operationen hängen vom Typ ab. Alle Datentypen in C können auf die **Basis-Datentypen zurückgeführt werden**:

- **Char:** Zeichen (character) 1 Byte
- **Int:** ganze Zahl (integer) 4 Byte
  - Short int: kleinerer Wertebereich (mind. 16 Bit) 2 Byte
  - Long int: größerer Wertebereich (mind. 32 Bit)
- **Float:** Fließkommazahl (floating point) 4 Byte
- **Double**: größerer Wertebereich 8 Byte
- **Long Double**: Noch größerer Wertebereich
- Char und Int können vorzeichenbehaftet (Standardfall) oder vorzeichenlos sein:
  - **Signed Short Int** $[-2^{(16-1)}, 2^{(16-1)}-1] = [-32768, 32767]$
  - **Unsigned Short Int** $[0, 2^{16}-1] = [0, 65535]$

1 Byte = 8 Bit

Benutzerdefinierter int-Typ: *enum ID{Wert1=x, Wert2=y}*. Wert1 und Wert2 als symbolische Konstanten für x und y. Standard-Aufzählungsbeginn ist 0. Der Compiler (nicht der Präprozessor!) ersetzt die Symbole dann durch die konkreten Konstanten.

Ganze Zahlen werden meist im 2er-Komplement dargestellt. Das erste Bit gibt das Vorzeichen an (1= negativ). 0 Wird also als positive Zahl aufgefasst.

3.2 Variablen und Konstanten

Variablen speichern Werte eines bestimmten Datentyps. Sie müssen vor Verwendung deklariert werden. Variablen besitzen einen eindeutigen Namen, und Schlüsselwörter sind als Namen nicht erlaubt. Je nach Compiler sind nur die ersten 31 Zeichen signifikant.

Mittels regulärer Ausdrücke (RAs) kann der C-Identifier kompakt beschrieben werden. Aus zwei RAs R1 und R2 kann man einen neuen RA R3 formen:

- **Alternative**: R3 = R1|R2 (R1 oder R2)
- **Konkatenation**: R3 = R1 R2 (hintereinanderschalten von R1 und R2)
- **Hülle**: R3 = (R1)* (beliebig vielen Wiederholungen von R1

Aufbau von Variablennamen:

- L = a|...|z|A|...|Z|_ *(letter)*

- D = 0|...|9 *(digit)*
- ID = L(L|D)* *(identifier, eindeutige Namen für Variablen, Konstanten, Datentypen)*

Deklaration bestimmt benötigten Speicherplatz und ermöglicht Korrektheitsprüfungen durch Compiler. Im Wesentlichen setzt eine Deklaration einen Datentyp in Verbindung zu einem Identifier. Allgemeine Form der Deklaration: *Typ ID (, ID)**;

Typedef wird verwendet um einen Datentyp unter einem neuen Namen bekannt zu machen. Beispiel: typedef unsigned long int ULI (ULI als Abkürzung für unsigned long int)

Falls Werte sich innerhalb eines Programms nie ändern, so werden diese als **Konstanten** statt Variablen dargestellt. Diese benötigen keine Deklaration, da ihr Typ implizit ist. Es gibt verschiedene Formen:

- Symbolische Konstanten im Präprozessor (Textersetzung). z.B *#define MAXCOUNT 1000*
- Konstante Ausdrücke. Werden vom Compiler vorab ausgerechnet, nicht zur Programm-Laufzeit (compile time-Konstante). z.B MAXCOUNT + 1
- Konstante Variablen. Werden im Read Only Speicher abgelegt (ROM). z.B *const int a = 100* („const“ ist ein type qualifier)

3.3 Arithmetische und logische Operationen und Zuweisungen

Auf Variablen und Konstanten können verschiedene Rechenoperationen angewendet werden: - + - * / und % (modulo, nicht bei float-Werten). Komplexere Operationen sind in Libraries verfügbar.

Zuweisung erfolgt durch Zuweisungsoperator „=“. *Ziel = Ausdruck*. Kurzformen sind:

- ++ (increment)
- - - (decrement)
- +=, *= etc. (x += y entspricht x = x+y)

Vergleichsoperationen geben als Ergebnis einen Booleschen Wert (true (von 0 verschiedener Wert) oder false (=0)). Vergleichsoperation sind:

- >, >=, <, <=
- == (Gleichheit), != (Ungleichheit)

Shift-Operationen sind nützlich für den Zugriff auf einzelne Bits eines Wertes oder Vereinfachung von Rechenoperationen. (x >> 1 = x/2, x<<2 = x*4).

- >> Rechtsshift
  - **Logisches Shift**: Auffüllen links mit 0
  - **Arithmetisches Shift**: Auffüllen links mit Vorzeichen
- << Linksschift

Bitweise logische Operationen werden meist in der maschinennahen Programmierung verwendet um einzelne Bits abszfragen oder zu manipulieren

- &: bitweise UND-Verknüpfung
- |: bitweise ODER-Verknüpfung
- ^: bitweise Exklusiv-Oder-Verknüpfung (XOR)

Expliziter **type cast** durch Programmierer: *Ziel = (Typ) Ausdruck;*

3.4 Kontrollanweisungen

Verzweigungen. Bedingung muss true oder false sein. Auf die {}-Klammerung darf verzichtet werden. Allgemeine Form

if(Bedingung)

{statement}*
else{statement}*

Komplexe Bedingungen können durch logische Verknüpfungen geprüft werden:
- ODER-Operator „||“ UND-Operator „&&“

Dangling else-Problem. „else“ gehört immer zum innersten freien „if“.
Verbindung auch über Labels und **goto-Statements** möglich. Möglichst vermeiden aber! Syntax wie folgt:
- ID: *statement* (ID ist Bezeichner des Labels)
- goto ID; (Sprung nach Position ID)

Switch Verzweigungen: Kette von if-then-else Statments. Hinter den statements soll mann das „break-Statement“ benutzen um aus der Switch Verzweigung raus zu springen. Allgemeine Form:

switch(Ausdruck){
case Konstante$_1$:
*statement**
…
case Konstante$_n$:
*statement**
*default: statemen**
}

Schleifen sind wiederholte Ausführungen von Anweisungsfolgen. C bietet drei Formen von strukturierten Schleifen:
- for-Schleife: falls die Anzahl der Durchläufe von vornherein feststeht
- do-Schleife: falls mindestens ein Durchlauf stattfinden muss und unbekannte Anzahl Iterationen
- while-Schleife: bei unbekannte Anzahl Iterationen

Spezielle Statements in Schleifen sind:
- Break: verlässt die Schleife komplett
- Continue: verlässt die aktuelle Iteration

Allgemeine Form der Schleifen:
for Schleife: *for(Ausdruck1; Ausdruck2; Ausdruck3){statement*}*
- Ausdruck1: Wird vor Schleifenbeginn einmal ausgeführt (Initialisierung)
- Ausdruck2: Bedingung für jeden Durchlauf (jede Iteration)
- Ausdruck3: Wird nach statement* ausgeführt

while Schleife: *while(Ausdruck){statement*}*
Ausdruk wird ausgewerter. Falls *true*, wird die Schleife betreten. Nach *statement** wird an den Anfang zurückgesprungen und *Ausdruck* wird erneut geprüft.

Do Schleife: *do{statement*}while(Ausdruck)*
*statement** wird aus0geführ, und *Ausdruck* wird ausgewertet. Falls true, so wird an den Anfang zurückgesprungen.

Falls die Abbruchbedingung einer Schleife nie erfüllt wird, terminiert es nicht: eine **Endlosschleife**.

3.5 Zeiger

Ein **Zeiger (pointer)** ist eine Variable, die die Speicheradresse einer Variablen enthält.

Jedes Objekt besitzt Adresse im Speicher. Häufig werden Objekte per Namen zugegriffen aber indirekter Zugriff über Adresse kann sinnvoll sein. Objekt-Adressen werden in **Zeigern** gespeichert. Zeiger sind selbst Objekte mit Adressen, daher sind auch Zeiger auf Zeiger möglich.

int x; //normale int-Variable
int p; //Zeiger auf einen int-Wert im Speicher*
p= &x; //p = Adresse(x), d.h. „p auf x zeigen lassen“.

Jetzt können x und *p synonym verwendet werden; es sind Decknamen.

Zeiger werden mittels „*“ deklariert. *Typ * ID;* deklariert einen Zeiger auf ein Objekt vom Typ *Typ*. Symbol „*“ gehört syntaktisch zu ID, nicht zum Typ. Der „*“-Operator „dereferenziert“ einen Zeiger. Zeiger auf x wird auch als Referenz auf x bezeichnet.

Erzeugung von Referenzen durch den Adressoperator „&“. & und * sind invers zueinander. **„&“ nur auf Objekte anzuwenden, „*“ nur auf Zeiger.** Zeiger sind im Prinzip identisch mit unsigned Integer-Zahlen.

Zwecke eines Zeigers:

- Mittels einem einzigen Objekt auf verschiedene andere Objekte zugreifen
- Zugriff auf Dynamischen Speicher. Der **Heap** ist ein dynamischer Speicherbereich. Anforderung mittels Funktion „malloc“. *int* p = (int*) malloc(1000);* p zeigt jetzt auf Anfang von freiem 1000-Byte Heap-Block. Bei der Standard-Library-Funktion „malloc“ ist ein type cast notwendig. Zunächst gibt sie einen „generischen“ Zeiger vom Typ „void*“ zurück. Dieser muss dann explizit auf den gewünschten Typ (hier „int*“) gecastet werden.

„**free()**“ ist das Gegenstück zu „malloc“. Die explizite Freigabe von nicht mehr benötigten Speicherbereichen.

4. Zusammengesetzte Datentypen

4.1 Arrays

Arrays definieren zusammengesetzte Datentypen, deren Elemente durch einen Index zugegriffen werden. Deklaration: *Typ ID [int-Konstante]*. Indizierung begint bei 0! **Benachbarte Arrayelemente** befinden sich stets auch benachbart im Speicher.

Index i bezeichnet den Abstand des i-ten Elementes von der Basisadresse V (= &V[0]). Hierdurch können Arrays auch nicht mit einem einzigen Befehl kopiert werden, aber muss Elementweise passieren. Daher:
V[i] = *(V+i) = *(i+V) = i[V].
V[3] = 123;, p[3] = 123; und *(p+3) = 123; sind äquivalent.

Beim **Post-Inkrement** wird der Ausdruck zunächst ausgewertet und dann inkrementiert. *p++ = .. ist zu lesen wie *p = ..; p++;

String = Array mit char-Elementen. Element-Wert 0 (oder '\0' als char-Konstante) kennzeichnet String-Ende. Daher: char-Arrays immer ein Byte länger als „Nutzlast“-String.

Mehrdimensionale Arrays sind auch möchlich *Typ ID [X][Y]*.

4.2 Strukturen

Ein **Struct** ist eine Zusammenfassung einer Gruppe von Variablen zu einer Einheit. Adressierung per Namen, nicht per Index wie bei Array. Allgemeine Form einer struct-Deklaration: *struct*

ID{Variablendeklaration}.* ID heißt structure tag. Struct-Deklaration oft in Verbindung mit typedef:
struct point{float x, y;} p1, p2 ==>
typedef struct point{float x, y;} Point
Point p1, p2;

```
typedef struct person
        { char name[100], vorname[100];
        struct datum{int tag, monat, jahr;} gebdatum;
        } Person;
Person struct1, struct2, *struc_ptr;
```

Alle Komponenten können auf **einmal kopiert** werden *struct1 = struct2*. Die Adresse des Structs ist struct_ptr = &struct1.

Komponentenzugriff direkt (Operator „.“): *struct1.gebdatum.jahr = 1950;*
Komponentenzugriff per Zeiger (Operator „->“) *struct_ptr->gebdatum.monat = 12; // „p->“ entspricht „(*p).“*

5. Funktionen und Programmaufbau

5.1 Funktionen

Eine **Funktion** ist ein abgegrenztes Teilprogramm, welches aus einer Menge von Eingabeparametern eine fest definierte Ausgabe erzeugt. Zu jedem Zeitpunkt befindet sich das Programm innerhalb einer bestimmten Funktion f (zu Beginn die main-Funktion). Funktion f kann durch Aufruf Funktion g unterbrochen werden. F übergibt Parameter an g. Danach springt g zurück auf f.

Funktionen können nicht nur andere Funktionen, sondern auch sich selbst aufrufen: **Rekursion**.

- **Deklaration:** Name, Definitionsbereich, Wertebereich (d.h. Interface der Funktion nach außen). Allgemeine Form: *Typ ID (Parameter-Deklaration*);*
- **Definition:** Programmcode zur Implementierung durch gewünschten Funktionalität. Wenn diese vor ersten Verwendung erfolgt, darf Deklaration entfallen. Allgemeine Form: *Typ ID (Parameter-Deklaration*){statement*};*

Verlassen einer Funktion durch return-Statement. In Form: *return Ausdruck;* Ausdruck muss dem Wertebereich entsprechen. Verlassen nicht per goto! Wenn Funktion keinen Rückgabewert hat, ist der Funktionstyp „void“.

Kommunikation zwischen **caller** und **callee**:
void f(){ int x; … / ruft g auf */ };*
int g(int a, int b){return a + b};
a und b sind **formale Parameter** der Funktion g gemäß Deklaration. Es sind Platzhalter für die aktuellen Parameter, die beim Aufruf an die Funktion übergeben werden.

Zwei Arten von Variablen:
- **Globale Variablen** sind im gesament Programm gültig. Ein korrekter Compiler initialisert globale Variablen automatisch mit 0.
- **Lokale Variablen** sind nur in einer bestimment Funktion gültig. Diese können globale Variablen gleichen Namens verdecken.

5.2 Parameterübergabe

Bei Übergabe eines Wertes w an eine Funktion f wird dem formalen Parameter von f eine lokale Kopie von w zugewiesen (**call by value**). Falls Modifikation von w erwünscht: Übergabe als Referenzparameter (**call by reference**).

```
Void f1(int a)
{ a++; }
void f3(int* a)
{ (*a)++; }
main()
{       int a;
        a = 1;
        f1(a);          /* a = 1 */
        f3(&a);         /* a = 2 */
}
```

- **Call by value → f1**
- **Nachbildung Call by reference in C** → Call by Value mit Zeiger auf a → f3

- Änderungen an einem Parameter in Funktion erscheinen nicht in der aufrufenden Funktion
- Soll eine Funktion einen Wert dauerhaft ändern, so muss die Adresse des Wertes via Zeiger übergeben werden.
- Auch wenn Structs übergeben werden, wird eine Kopie erstellt → kostet viel Zeit und Arbeitsspeicher. Besser Adresse zu übergeben.

Arrays werden immer als Referenzparameter übergeben. Arrays sind als Rückgabewert unzulässig.

Funktionen besitzen eine Speicheradresse, also existieren **Zeiger auf Funktionen**. Der Name einer Funktion ist keine Variable, sondern (wie ein Arrayname) eine Adresskonstante.

```
int f(float, float);        /* Funktion f */
int(*p)(float,float);       /* Zeiger auf Funktion f */
p = &f;
```

5.3 Module und Programme

Module sind selbständig übersetzbare Programmeinheiten. Ein Modul bündelt typischerweise verschiedene logisch zusammengehörige Funktionen, die durch Programme bzw. andere Module genutzt werden können. Der C-Compiler übersetzt immer ein einzelnes C-Modul. Der Linker setzt komplettes C-Programm zusammen. Ein Modul muss nicht unbedingt main-Funktion enthalten, aber ein C-Programm muss genau eine main-Funktion enthalten. Globale Variablen und Funktionen gleichen Namens nur einmal im ganzen Programm erlaubt.

- **Interfaceteil**: beschreibt für andere Module sichtbar die Schnittstelle der zu exportierenden Ressourcen .
- **Implementationsteil:** Hier werden die Ressourcen realisiert. Die nicht im Interfaceteil beschriebenen Ressourcen des Moduls sind für andere Module nicht sichtbar.

Trennung z.B durch Deklarationen im **Header File** und Implementierung im **Source File**.

6. Analyse von Algorithmen

6.1 Laufzeitanalyse

Um Algorithmen mit einander vergleichen zu können benutzt man die **Komplexitätsanalyse von**

Algorithmen. Die Komplexität ist die Maß für die Effizienz. Es gibt zwei Analysemöglichkeiten:

1. Zählen der Einzelschritte, aber aufwendig, rechnerabhängig
2. Abstrakte Analyse: z.B. Anzahl Vergleiche, Anzahl zu vertauschende Paare; rechnerunabhängig, aber sehr grob.

Die Frage ist: Welcher Algorithmus ist schneller für bestimmte Eingabegröße (n)? Also für große n der Alg., dessen Laufzeit weniger mit n steigt.

6.2 O-Notation für Wachstumsordnungen

O-Notation (O = oder) ist das gebräuchlichste Hilfsmittel zur Wachstumsanalyse von Algorithmen.

Def: Sind f: N → N und g : N → N zwei Funktionen, so ist $f \in O(g)$, falls $\exists\, n_0 \in N$, $\exists\, c \in R$, so dass $\forall\, n \geq n_0$: $f(n) \leq c * g(n)$

$f \in O(g)$ heißt: „f wächst nicht schneller als g“
Komplexitätsklassen: Wie verändert sich der Rechenaufwand, wenn man die Eingabegröße um einen bestimmten Faktor vergrößert.

O(g) ist die Menge aller Funktionen, die höchstens so schnell wachsen wie c*g. Die Konstanten werden also vernachlässigt. Der stärkste ansteigende Term setzt sich durch.

Mittels O-Notation können auch andere Maße für die Qualität eines Algorithmus im Vergleich zu anderen abgeschätzt werden, z.B. Speicherbedarf. Außerdem kann die O-Notation aussagen, ob sich die Suche nach einem besseren Algorithmus für ein Problem lohnt.

Rechenregeln:

- „O“ im Ausdruck vernachlässigen
- Umformen des Ausdrucks wie gewohnt
- Nur den führenden Term (höchster Exponent) behalten
- $f+g \in O(\max\{f,g\})$
- $f*g \in O(f*g)$
- Linearität: $f(n) = c_1 g(n) + c_2$ mit c_1 und c_2 aus R, so ist $f \in O(g)$

Wichtige Wachstumsordnungen sind: $O(1) \subset O(\log_2 n) \subset O(\text{sqrt}(n)) \subset O(n) \subset O(n\log_2 n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$.

Beispiele ab seite 133 im Skript

7. Sortieralgorithmen

7.1 Einfache Verfahren

Gegeben: Folge S = (S1,...,Sn) von Datensätzen (structs) Jeder struct besitzt eine Komponente Si.key eines mittels einer Ordnungsrelation linear geordneten Datentyps. Gesucht: Permutation (Anordnung) S' = (Si1,...Sin) der Elemente von S, so dass

$$si1.key \leq si2.key \leq \ldots \leq sin.key$$

Hierbei dürfen gleiche Schlüssel mehrfach vorkommen.

Ein Sortieralgorithmus, der die ursprüngliche Reihenfolge bei gleichem Schlüssel beibehält, **heißt stabil.**
Analyse: Die Anzahl der Vergleichsoperationen ist eine obere Schranke für die Anzahl der Vertauschungen, da jede evtl. notwendige Vertauschung immer erst nach einem Vergleich durchgeführt werden kann.

Die **Exchange-Funktion** die wir benutzen werden in den folgenden Algorithmen ist:

```
void exchange(int *x,int *y){
        int temp;
        temp = *y;
        *y = *x;
        *x=temp;
}
```

Selection Sort:

Der Algorithmus durchläuft Array A vom linken Rand 1 bis zum rechten rand r-1. In jedem Durchlauf i wird zunächst das Minimum auf den Index i gesetzt. Im Folgenden wird das Array von der i-ten Position aus nach rechts durchlaufen und das Minimum mit dem aktuellen Element verglichen und u.U. Aktualisiert. Am Ende des Arrays angelangt wird das i-te Element mit dem gefunden (lokal) minimalen Element vertauscht.

Laufzeit: $O(N^2)$ **Stabil**: Nein!

```
void selection_sort(int A[], int l, int r)
{       int i, j, min;
        for (i = l; i < r; i++)
        {       min = i;
                for (j = i+1; j <= r; j++)
                {       if(A[j] < A[min]) min = j;
                }
                exchange(&A[i],&A[min]);   /* vertauschen */
        }
}
```

Insertion Sort:

Durchlaufen des Arrays v.l.n.r (start bei i = l+1). Elemente links vom aktuellem Element i werden so lange nach rechts rücken, bis das Element an der richtigen Stelle ist. Also bis A[i] größer gleich eins der Elemente links von sich ist.

Laufzeit: $O(N^2)$ **Stabil**: Ja!

```
void insertion_sort(int A[], int l, int r)
{       int i, j, v;
        for (i = l+1; i <= r; i++)
        {       v = A[i];
                for (j = i-1; j >= l; j--)
                {       if (v < A[j]) A[j+1] = A[j];
                        else break;
                }
                A[j+1] = v;
        }
}
```

Bubblesort

Sortieren durch fortgesetztes paarweises Vertauschen von rechts nach links, kleinstes Element wird zunächst nach ganz links getauscht, danach das zweitkleinste usw.

Laufzeit: $O(N^2)$ **Stabil**: Ja!

```
void bubble_sort(int A[], int l, int r)
{       int i, j;
        for (i = l; i < r; i++)
        {       for (j = r; j > i; j--)
                {       if (A[j-1] > A[j]) exchange(&A[j-1],&A[j]);
                }
        }
}
```

7.2 Quicksort

Die Idee ist: Divide et impera. Teile und herrsche → Mehrere kleine Probleme sind insgesamt leichter zu lösen als ein großes. Teile also ein großes Problem. Hier wird das zu sortierende Array A in zwei Teile A' und A" getrennt und unabhängig sortiert.

1. Trennelement A[k] wählen.
2. Andere Elemente so vertausche, dass alle Elemente links van A[k] kleiner gleich sind als A[k] und alle Elemente rechts von A[k] größer gleich als A[k].
3. Für die neuen Teilarrays rekursive Aufrufe von Quicksort

Laufzeit Worst Case → Trennelement ist das Maximum des Arrays → $O(N^2)$ (Kann vermieden werden durch den Median von 3 Elementen als Trennelement zu benutzen)
Laufzeit Best und Average Case → O(NlogN)
Stabil: Nein!

```
int partition(int A[], int l, int r)
{       int i,j,k,v;
        k = r; v = A[k];        /* willkürliches Trennelement */
        i = l;                  /* starte am linken Rand */
        j = r-1;                /*starte am rechten Rand -1 */
        while (1)               /* durchlaufen bis Abbruch */
        {       while(A[i] <= v && i < r) i++;
                while(A[j] >= v && j >= l) j--;
                if(i >= j)      /* aneinander vorbeigelaufen ? */
                        break; /* Abbruck der while-Schleife */
                else
                        exchange(&A[i],&A[j]);
        }
        exchange(&A[i], &A[k]);
        return i;
}

void quick_sort(int A[], int l, int r)
{       int k;
        if(r <= l) return;
        k = partition(A, l, r);
        quick_sort(A,l,k-1);
        quick_sort(A,k+1,r);
}
```

7.3 Heapsort

Das Heapsort ist im wesentlichen ein verbessertes Selection Sort. Grundlage hierfür ist die „Heap-Eigenschaft“ des Arrays. Das Array ist unsortiert hat aber die folgende Eigenschaft (**(Minimums-)Heap**):

- $A[i] \leq A[2i]$, falls $2i \leq N$
- $A[i] \leq A[2i+1]$, falls $2i+1 \leq N$

Jedes Baumelement A[i] besitzt 0, 1 oder 2 Söhne. Söhne besitzen Arrayindex 2i bzw. 2i+1.

Eine Eigenschaft ist, dass das globale Array-Minimum sich ganz oben im Heap befindet (die Wurzel)

Ablauf vom Heapsort (absteigende Sortierreihenfolge):

1) Verwandeln des unsortierten Arrays in einen Heap. Wurzel A[1] ist das Minimum
2) Vertausche Wurzel und A[N]
3) Min. befindet sich danach an der richtigen Stelle N
4) A[2...N-1] ist noch korrekter Heap, jedoch neues Element A[1] evtl. falsch
5) Wiederherstellen der Heap-Eigenschaft von A[1...N-1] (durch einsinken lassen)
6) Fortsetzen ab Schritt 2 mit nächstem Element N-1 und neuer Wurzel

Beim einsinken läuft eine Schleife so lange bis der Knoten k seinen richtigen Platz im Heap gefunden hat. Im wesentlichen wird eine Fallunterscheidung durchgeführt: hat der konnten 0, 1 oder 2 Söhne und muss noch mit einem Sohn vertauscht werden?

Laufzeit: O(NlogN) **Stabil**: Nein!

```
void sink(int A[], int k, int N) /* A[0] nicht benutzt */
{       int son;                        /* speichert kleinsten Sohn falls vorhanden */
        while(1)                        /* durchlaufen bis Abbruch */
        {       if(2*k >N) break;       /* Knoten k hat keinen Sohn */
                if(2*k+1 <= N)          /* Knoten k hat 2 Söhne */
                {       if(A[2*k] < A[2*k+1]) son = 2*k;
                        else     son = 2*k+1;
                }
                else son = 2*k;         /* 2k <= N, Knoten k hat 1 Sohn */
                if (A[k] > A[son])      /* Vertauschen notwendig? */
                {       exchange(&A[k],&A[son]);
                        k = son;        /* Knoten k evtl. weiter sinken lassen */
                }
                else break;             /* sont: richtige Pos. für k gefunden */
        }
}

void heap_sort(int A[], int N)          /* A[0] nicht benutzt */
{       int i;
        for (i = N/2; i > 0; i--)       /* Heap aufbauen */
        {       sink(A,i,N);
        }
        for (i = N; i > 1; i--)
        {       exchange(&A[1], &A[i]);     /* Min. ans akt. Heap-Ende setzen */
                sink(A,1,i-1);          /* Heap reparieren */
        }
}
```

8. Lineare Datentypen

8.1 Listen

Listen werden gebraucht zur Darstellung von Mengen von Objekten mit bestimmten Operationen (Einfügen, Suchen, Entfernen und zusammenfügen). Bei Arrays ist dies schwierig. **Eine (verkettete) Liste** ist entweder leer oder besteht aus einer Referenz auf einen Knoten, der ein Element und eine Referenz auf eine verkettete Liste enthält.

Modellierung eines Knotens für ein Element vom Typ T: *typedef struct node{T* data; struct node* next} *nodeptr*

- ➔ Data ist Zeiger auf Listenelement vom Typ T
- ➔ Next ist Zeiger auf Nachfolgerknoten.

Eine Liste ist generell eine rekursive Datenstruktur. Sie besteht aus einem Kopf und einer Restliste.

Definition des Datentyps Liste, Element vom Typ T:

```
typedef struct node{
        T* date;
        struct node* next; } Node, *nodeptr;

typedef struct list {
        nodeptr first, last;} List, *listptr
```

Erzeugung einer leeren Liste und Leerheitstest

```
List L;
void Init(listptr L)
{       L->first = NULL; L->last = NULL; }

int IsEmpty(List L)
{ return (L.first == NULL && L.last == NULL;}
```

Neuen Listenknoten erstellen

```
nodeptr newnode(T* item)
{       nodeptr np;
        np = (nodeptr)malloc(sizeof(Node));
        np->data = item;
        np->next = NULL;
        return np;
}
```

Einfügen am Listenanfang

```
void AppendFirst(T* item, listptr L)
{       nodeptr np = newnode(item);
        if (IsEmpty(*L))
        {       L->first = np; L->last = np;
        }
        else
        {       np->next = L->first;
                L->first = np;
        }
}
```

Einfügen am Listenende

```
void AppendLast(T* item, listptr L)
{       nodeptr np = newnode(item);
        if(IsEmpty(*L))
        {       L->first = np; L->last = np;
```

```
        }
        else
        {       L->last->next = np;
                L->last = np;
        }
}
```

Prüfen ob Element in Liste enthalten ist

```
int IsIn(T* item, List L)
{       nodeptr np;
        if(IsEmpty(L)) return 0;        /*  false */
        np = L.first;
        while(np != NULL)
        {       if(Equal(np->data, item)) return 1;   /* true */
                np = np->next;
        }
        return 0; /* false */
}
```

Laufzeit: O(N) in worst und average case, O(1) in best case.

Einfügen hinter ein bestimmtes Listenelement

```
/* item2 hinter item1 einfügen */
void InsertBehind(T* item1, T* item2, listptr L)
{       nodeptr np, newnp;
        if(!IsIn(item1, *L))
        {       printf(„FEHLER!!!EINZ11!11!11!!!“); return;
        }
        np = L->first;
        while(np!=NULL)
        {       if(Equal(np->data, item1))
                {       newnp = newnode(item2);
                        newnp->next = np->next;
                        np->next = newnp;
                        if(np == L->Last)
                        {       L->last = newnp; }
                        break;
                }
                n = np->next;
        }
}
```

Löschen eines Listenelementes

```
void Delete(T* item, listptr L)
{       nodeptr np1, np2;
        if(IsEmpty(*L)) return;
        np1 = L->first;
        if(Equal(np1->data,item))
        {       L->first = np1->next;
                if(L->first == NULL)
                {       L->last == NULL; }
                free(np1);
                return;
        }
        np2 = np1->next;
        while(np2 != NULL)
```

```
        {       if(Equal(np2->data,item))
                {       np1->next = np2->next;
                        if(np2==L->last)
                        {       L->last = np1; }
                        free(np2);
                        break;
                }
                np1 = np2;
                np2 = np2->next;
        }
}
```

Zusammenfügen zweier Listen

```
listptr Union(listptr L1, listptr L2)
{       if(IsEmpty(*L2)) return L1;
        if(IsEmpty(*L1)) return L2;
        L1->last->next = L2->first;
        L1->last = L2->last;
        return L1;
}
```

Beachten: Bei Mengen gibt es kein Mehrfachvorkommen von Elementen!

Schnittmengenberechnung mittels Listen

```
listptr Intersect(listptr L1, listptr L2)
{       nodeptr np;
        listptr L = (listptr)malloc(sizeof(List));
        Init(L);
        np = L1->first;
        while(np != NULL)
        {       if(IsIn(np->data, *L2))
                        AppendLast(np->data,L);
                np = np->next;
        }
        return L;
}
```

8.2 Stacks

Ein Stack ist ein ADT. Es ist eine LIFO-Datenstruktur: was zuletzt auf den Stack gelegt wird, wird zuerst entnommen. Es unterstützt folgende Grundoperationen:

1) Einfügen eines neuen Elementes („Push“)
2) Entfernen des zuletzt eingefügten Elementes („Pop“)

Ein Beispiel eines Stacks ist der **Laufzeit-Stack**: Funktionsaufruf → Push, Funktionsende → Pop. Bei einer korrekten Programmabarbeitung sollte am Ende in jedem Fall der gleiche Stack-Zustand wie am Anfang herrschen.

Es gibt zwei Sondersituationen die zu vermeiden sind:

1) Etwas auf einen „vollen“ Stack abzulegen
2) Etwas von einem leeren Stack zu entnehmen

Stack-Implementierung durch Listen

Elemente auf Stack sind Listenelemente. Pushen heißt einfügen am Listenanfang. Poppen heißt entfernen des ersten Elementes.

```
List L;

void Push(T* item, listptr L)
{       AppendFirst(item,L); }

T* Pop(listptr L)
{       T* item;
        if(IsEmpty(*L)) return NULL;
        item = L->first->data;
        Delete(item,L);
        return item;
}
```

Stack-Implementierung durch Arrays

Vorgabe der max. Stackgröße notwendig. Elemente auf Stack sind Arrayelemente. Wesentliche Unterschied zu Listenimplementierung liegt darin dass der Speicherplatz statisch festliegt.

```
#define MAX 100
typedef struct stk
{       T* elems[max]; int top; } stack;

void Init(stack* s)
{       s->top = 0; }

int IsEmpty(stack s)
{       return (s.top == 0); }

void Push(T* item, stack* s)
{       if(s->top == MAX)
        {       prinf(„Stack voll!!111!1EINZ11!!!“); return;}
        s->elems[s->top]=item;
        s->top++;
}

T* Pop(stack* s)
{       if(IsEmpty(*s)) return NULL;
        s->top--;
        return s->elems[s->top];
}
```

8.3 Queues

Eine **Queue (Warteschlange)** ist ein ADT. Es ist eine FIFO-Datenstruktur: was zuerst auf den Stack wird gelegt, wird zuerst entnommen. Grundoperationen sind:

1) Einfügen eines neuen Elementes („Put“)
2) Entfernen des zuerst eingefügten Elementes („Get“)

Haben Verwandtschaft mit Stacks aber der Zugriff erfolgt quasi umgekehrt.

Sondersituationen die vermeidet werden müssen:

1) Etwas an eine „volle“ Queue zu putten.
2) Etwas aus einer leeren Queue zu getten.

Queue-Implementierung durch Listen:
Queue-Elemente = Listenelemente. Put ist das Einfügen am Listenende. Get ist das Entfernen des ersten Elementes.

List L;

```
void Put(T* item, listptr L)
{       AppendLast(item,L); }

T* Get(listptr L)
{       T* item;
        if(IsEmpty(*L)) return NULL;
        item = L->first->data;
        Delete(item, L);
        return item;
}
```

Queue-Implementierung durch Arrays:
Das problem bei Queues ist das diese immer weiter nach rechts „wandern“. Da immer hinten etwas an die Queue angehängt wird, und vorne etwas von der Queue entnommen wird. Arrays könnten hierdurch schnell „voll“ sein. Die Lösung ist: ein **ringförmiges Array** bzw. zyklische Adressierung. Dass heißt der nachfolger von A[N] ist nicht A[N+1] aber A[0].

Der Modulo Operator wird verwendet zur zyklischen Adressierung. Zwei Zeiger, first und last, verweisen auf aktuellen Anfang (-1) und Ende (+1) der Queue. Es muss auch auf Über- und Unterlauf geprüft werden.

```
#define MAX 100
typedef struct q
{       T* elems[MAX];
        int first, last; } queue;

void Init(queue* q)
{       q->first = 0; q->last=1; }

void Put(T* item, queue* q)
{       if(q->last == q->first) overflow();
        else
        {       q->elems[q->last] = item;
                q->last = (q->last+1) % MAX;
        }
}

T* Get(queue* q)
{       q->first = (q->first+1) % MAX;
        if(q->first == q->last) underflow();
        else return q->elems[q->first];
}
```

9. Bäume

9.1 Definition und Darstellung

Ein Baum besteht aus einem ausgezeichneten Knoten r (Wurzel, root) und $k \geq 0$ disjunkten Bäumen $T_1,....,T_k$. Bäume werden rekursiv definiert. Unterhalb der Wurzel befinden sich weitere Bäume. Auf der untersten Ebene haben die Bäume dan keine „Söhne“ mehr.

Ein Binärbaum $B = (r, B_L, B_R)$ ist entweder leer oder besteht aus einer Wurzel r sowie einem linken und rechten Teilbaum B_L bzw. B_R. Binärbäume sind auch rekursiv.

Bäume als dynamische Datenstruktur, ähnlich zu Listen: Zeiger „data“ auf Nutzdaten eines beliebigen Typs T. Zeiger „left“ und „right“ verweisen auf linken und rechten Teilbaum.

```
typedef struct tr
{       T* data;
        struct tr *left, *right; }
btree, *btreeptr;
```

9.2 Baumdurchläufe

Häufige Anwendung: Durchlaufen eines Baumes, d.h. Besuchen/Verarbeiten aller Knoten in bestimmter Reihenfolge. Es gibt drei prinzipielle Durchlaufschemata:

- **Inorder: L W R**

```
void inorder(btreeptr b)
{
if(b==NULL) return;
inorder(b->left);
visit(b->data);
inorder(b->right);
}
```

- **Preorder: W L R**

```
void preorder(btreeptr b)
{
if(b==NULL) return;
visit(b->data);
preorder(b->left);
preorder(b->right);
}
```

- **Postorder: L R W**

```
void postorder(btreeptr b)
{
if(b==NULL) return;
postorder(b->left);
postorder(b->right);
visit(b->data);
}
```

In/Pre/Post bzgl. des Zeitpunkts der Wurzel-Bearbeitung. L = linker Teilbaum, R = rechter Teilbaum, W = Wurzel. Mit:

```
void visit(T* item)
{       /* Verarbeitung von item */
```

}
Laufzeit: O(#Knoten)

9.3 Suchbäume

Suchbäume werden verwendet für folgende Mengenoperationen benutzt: Einfügen, Löschen, Suchen, Schnitt, Vereinigung. Bei großen Datenmengen sind Listen zur Mengenverwaltung zu ineffizient (Laufzeit: O(N)). In einem sortiertem Array kann man Suchen in Zeit O(logn) per binärer Suche, aber Einfügen und Löschen aufwendig.

Binäre Suche: Berechnung eines Entscheidungsbaums, d.h. In jedem Schritt Verzweigung in linkes oder rechtes Teilarray.

Def: Sei $B = (r, B_L, B_R)$ ein Binärbaum und bezeichne key(n) den Suchschlüssel für jeden Knoten n. B heißt Suchbaum, falls gilt:

1. Für alle $n \in B_L$: key(n) < key(r)
2. Für alle $n \in B_R$: key(n) > key(r)
3. B_L und B_R sind Suchbäume

Mehrfachvorkommen von Schlüsseln ist nicht erlaubt. Durch dieses Prinzip kann die Größe des noch zu untersuchenden Restbaumes halbiert werden mit jedem Schritt.

Einfügen eines Elementes:

```
btreeptr insert(T* item, btreeptr b)
{       btreeptr n;
        if(b==NULL)
        {       n=(btreeptr)malloc(sizeof(btree));
                n->data = item;
                n->left = NULL; n->right = NULL;
                return n;
        }

        if(key(item) < key(b->data)))
                b->left = insert(item,b->left);

        if(key(item) > key(b->data))
                b->right = insert(item,b->right);

        return b;
}
```

Suchen eines Elementes

```
T* search(T* item, btreeptr b)
{
        if(b==NULL) return NULL;

        if(key(item) == key(b->data))
                return b->data;

        if(key(item) < key(b->data))
                return search(item, b->left);

        if(key(item) > key(b->data))
```

```
        return search(item,b->right);
}
```
Aufwand: Länge des Pfades von der Wurzel zum gesuchten Element.

Löschen eines Elementes: gleiches Prinzip wie beim Suchen, jedoch etwas aufwendiger beim Löschen von inneren Baumknoten (Wiederherstellung der Baumstruktur notwendig). **Aufwand:** Länge des Pfades von der Wurzel zum gesuchten Element.

Extremfälle sind:
- **Ausgeglichener Baum:** Höhe der Teilbäume unterscheidet sich max. um 1, auf jeder Ebene verdoppelt sich die Anzahl der Knoten, daher Suchzeit O(logn) bei n Knoten.
- **Entarteter Baum:** wie Liste, d.h. O(n)

Daher Laufzeit Suchen in binären Suchbaum: **O(logn) Average Case** und **O(n) in worst case.** In AVL Baum (ausgeglichener Baum) Laufzeit O(logN) bei worst case für Suchen, Einfügen, Löschen garantiert.

10. Graphen

10.1 Definition und Grapheigenschaften

Ein Graph ist eine graphische Darstellung einer Menge von Objekten und deren Beziehungen.

- **Def: Eine (binäre) Relation R** ist eine Teilmenge des kartesischen Produktes zweier Mengen: $R \subseteq AxB$. „aRb“ als Schreibweise für $(a,b) \in R$, „Kante“ zwischen a und b im Graphen. Spezialfall symmetrische Relation: $aRb \rightarrow bRa$
- **Def: Ein (gerichteter) Graph** $G = (V,E)$ besteht aus einer Menge V von Knoten (vertices) und einer Menge E von Kanten (edges) mit $E \subseteq VxV$
- **Def:** Gilt für alle Kanten (u,v) eines Graphen $G = (V,E)$ und $(u,v) \in E \rightarrow (v,u) \in E$ **so heißt G ungerichtet**. Schreibweise: $\{u,v\} \in E$. Also bei gerichteten Graphen weisen die Kanten in eine bestimmte Richtung.
- **Def:** Eine Folge $(v_1,...,v_n)$ von Knoten eines Graphen $G = (V,E)$ **heißt Pfad**, falls für alle i = 1..n-1: $\{v_i,v_{i+1}\} \in E$. Ein Pfad ist eine zusammenhängende Folge von Knoten in einem Graphen. Zwei durch eine Kante verbundene Knoten heißen „**adjazent**“.
- **Def:** Ein Graph $G = (V,E)$ heißt **zusammenhängend**, falls zwischen zwei beliebigen Knoten $u,v \in V$ mindestens ein Pfad existiert
- **Def:** Ein Graph $G = (V,E)$ heißt **zyklenfrei**, falls zwischen zwei beliebigen Knoten $u,v \in V$ höchstens ein Pfad existiert.
- **Def:** Ein zusammenhängender, **zyklenfreier** Graph $G = (V,E)$ heißt Baum.
- **Def:** Ein Graph $G = (V,E,w)$ mit einer Abbildung $w:E \rightarrow R$ heißt **(kanten)gewichteter Graph**.

10.2 Berechnung kürzester Pfad

Dijkstra-Algorithmus (single source shortest path). Wird benutzt zur Bestimmung der kürzesten Pfade von einem Startknoten V_0 aus zu allen anderen Graphknoten. Die Idee ist:
- In jedem Schritt Betrachtung einer „Nachbarschaft“ N von V_0, innerhalb deren die kürzesten Pfade zu V_0 bereits bekannt sind.
- Schrittweise Erweiterung von N um den jeweils am nächsten an N liegenden Knoten

Def: Ist $G = (V,E)$ ein Graph und $\{u,v\} \in E$, so heißt v **Nachfolger** von u, und u heißt **Vorgänger** von v.

Während der Abarbeitung des Algorithmus zerfällt der Graph in verschiedene Regionen:

- Der „**grüne“ Bereich**, in dessen Zentrum der Startknoten liegt, wurde bereits vollständig abgearbeitet, und alle kürzesten Wege darin sind bekannt.
- Der **„gelbe“ Bereich** bildet die Grenze von Knoten, die noch weiter untersucht werden müssen.
- Durch **„rote“ Kanten** werden jeweils die aktuell bekannten kürzesten Wege dargestellt.

Im folgenden Pseudo-Code kommen die folgenden Variablen vor:
- dist[v] bezeichnet den aktuell min. Abstand von Knoten v zum Startknoten
- cost(v,w) bezeichnet Gewichtung der Kante {v,w}
- Mengen GREEN, YELLOW und RED bezeichnen grüne und gelbe Knoten sowie rote Kanten. Die roten Kanten bilden stets einen Baum von aktuell kürzesten Pfaden vom Startknoten aus.

```
Void shortest_path(graph G = (V = {v0,...,vn},E))
{       set GREEN = {}, YELLOW = {v0}, RED = {}; dist[v0] =0;
        while(YELLOW != {})                     /*solange noch unbearbeitete Knoten */
        {       v=MinDist(YELLOW);              /* v ∈YELLOW mit min. dist-wert */
                Insert(v,GREEN);                /* GREEN = GREEN ∪{v} */
                Delete(v, YELLOW);              /* YELLOW = YELLOW \ {v} */

                for(w ∈Succ(v))                 /* für alle Nachfolger w von v */
                { if(!(w ∈YELLOW ∪GREEN))       /* neuer Knoten erreicht */
                  {     Insert({v,w}, RED);
                        Insert(w,YELLOW);
                        dist[w]=dist[v]+cost(v,w);
                  }
                  else if(w ∈YELLOW)            /* w erneut erreicht */
                  {     if(dist[v]+cost(v,w)<dist[w])
                        { Insert({v,w},RED);
                          e=PreviousEdge(w);    /* vorher rote Kante zu w */
                          Delete(e,RED);        /* RED = RED \ {e} */
                          dist[w]=dist[v]+cost(v,w);
                        }
                  }
                }
        }
}
```

De Algorithmus arbeitet in folgenden Schritten:
Zunächst werden die verwendeten Mengen initialisiert. Die äußere while-Scheife läuft dann solange, bis alle Knoten untersucht worden sind und somit alle kürzesten Pfade bestimmt sind.

Der Knoten v mit minimalen dist-Wert wird aus der YELLOW-Menge in die GREEN-Menge überführt. Für diesen wird anschließend geprüft (indem alle Nachfolger betrachtet werden), ob er zu neuen kürzesten Wegen beitragen kann.

Fall 1 (if): Der Knoten w wurden bisher noch gar nicht erreicht. Der kürzeste Weg zu w führt somit per Definition über v, und die Distanz wird entsprechend vermerkt.

Fall 2 (else): w wurde bereits auf einem anderen Weg erreicht. Ist der neue Weg zu w über v kürzer als der bisher bekannte, so wird der Wert dist[w] entsprechend aktualisiert und die Menge der roten Kanten demzufolge angepasst. *Beispiel ab seite 246 im Skript.*

10.3 Datenstrukturen zur Graphrepräsentation

Datenstrukturen zur internen Speicherung von Graphen sind vor allem: **Adjazenzmatrix** und **Adjazenzlisten**.

Adjazenzmatrix

Def: Ist G=(V,E) ein Graph mit Knotenmenge V = $\{v_1,...,v_n\}$, so heißt die (nxn)-Matrix A = (a_{ij}) mit
a_{ij} = 1, falls $\{v_i, v_j\} \in E$ (es ist eine Kante vorhanden)
a_{ij} = 0, sonst. (es ist keine Kante vorhanden)
Adjazenzmatrix zu G

Es ist ein zweidimensionales Array. Die Knotenmenge ist die Anzahl der Zeilen/Spalten.
Vorteil: Adjazenz zweier Knoten kann in Zeit O(1) geprüft werden.
Nachteil: Hoher Platzbedarf ($O(|V|^2)$), unabhängig von |E|

Implementierung:

```
#define N 100

int adj_matrix[N][N];

int adjacent(int i, int j)
{
        return adj_matrix[i][j] == 1 ? 1 : 0;
}
```

Bedingter Ausruck in C:
Bedingung „?“ then-Ausdruck „:“ else-Ausdruck

Adjazenzlisten

Adjazenzlisten speichern für jeden Knoten Liste seiner Nachbarn.
Vorteil: Speicherplatzbedarf nur O(|V| + |E|)
Nachteil: Test auf Adjazenz erfordert im **worst case** O(|V|)

Implementierung:

```
#define N 100

List adj_list[N]

int adjacent(int i, int j)
{
        List L;

        L=adj_list[i];
        return IsIn(j,L);
}
```

10.4 Graphdurchläufe

Ein Algorithmus, der alle Knoten eines Graphen der Reihe nach „besucht“. Beispiel ist garbage collection im Heap-Speicher → Test auf nicht mehr erreichbare Speicherblöcke.

Depth first search (DFS, Tiefensuche): Ausgehend vom Startknoten möglichst lange Pfade verfolgen. Bei zusammenhängendem Graphen nur ein Aufruf von DFS nötig, sonst von allen Knoten aus.

```
int visited[N];                         /* implizit mit 0 initialisiert */

void dfs(int v)                         /* DFS beginnend bei v */
{       int w;
        visited[v]=1;                   /* markiere v als besucht */
        process(v);                     /* Verarbeitung von v */
        for({v,w} ∈E)                   /* für alle Nachbarn w */
        {                               /* rekursiver Aufruf von dfs: */
                if(!visited[w]) dfs(w);
        }
}

void main()
{       int v;
        for (v=0; v<N; v++)
                if(!visited[v]) dfs(v);
}
```

Breadth first search (BFS, Breitensuche): ausgehend vom Startknoten weitere Knoten in der Reigenfolge wachsenden Abstands vom Startpunkt besuchen. Eine Queue speichert noch zu bearbeitende Knoten auf nächster Ebene, während „Geschwisterknoten“ auf selber Ebene gerade bearbeitet werden.

```
Int visited[N];

void bfs(int v)                         /* BFS ab Knoten v */
{       int w; Queue Q;
        visited[v] = 1;
        process(v);
        Put(v,Q);
        while(!IsEmpty(Q))
        {       v=Get(Q);
                for({v,w} ∈E)
                {       if(!visited[w])
                        {       visited[w]=1;
                                process(w);
                                Put(w,Q);
                        }
                }
        }
}

void main()
{       int v;
        for (v = 0; v<N; v++)
        if(!visited[v]) bfs(v);
}
```

Laufzeit für beide Algorithmen: O(|V| + |E|)

10.4 Spannbäume

Ein Spannbaum in einem ungerichteten, zusammenhängenden und kantengewicteten Graphen G=(V,E,w) ist ein zyklenfreier, zusammenhängender Teilgraph G'=(V',E',w) mit V'=V und E'⊆E.

Ein minimaler Spannbaum in einem ungerichteten, zusammenhängenden und kantengewichteten Graphen G = (V,E,w) ist ein Spannbaum mit minimaler Kantengewichtssumme unter allen Spannbäumen zu G.

Ein Spannbaum dient dazu, alle Knoten in einem Graphen miteinander zu verbinden.

Idee: fortgesetze Auswahl von Kanten mit geringem Gewicht, dabei Zyklen vermeiden.

1. Beginne mit leerem Spannbaum T
2. Bilde Liste L der Graphkanten von G, sortiert nach aufsteigendem Gewicht (s. Sortieralgorithmen)
3. Betrachte jeweils nächstgrößere Kante e in L
4. Falls Einfügen von e in T einen Zyklus verursachen würde: e als ungültig betrachten und verwerfen
5. Sonst: e in T einfügen
6. Fortsetzen ab 3, bis T insgesamt n-1 Kanten enthält. Jetzt bildet T einen minimalen Spannbaum zu G.

Für das Spannbaumproblem sind effiziente Algorithmen bekannt. Einer davon ist der **Kruskal-Algorithmus**. Es handelt sich um einen **Greedy-Algorithmus**. Der Kruskal-Algorithmus ist ein Spezialfall eines Greedy-Algorithmus weil er immer die optimale Lösung findet.

```
Graph MinSpanningTree(Graph G = (V,E))
{       Graph T; List L; Edge e;
        T={};
        L = Sort(E);                                    /* sortiere Kantenliste */
        while(#Kanten in T < #Knoten in G-1 &&
                !IsEmpty(L))
        {       e = FirstElem(L);                       /* nächst-billigste Kante */
                Delete (e,L);
                if(!Cycle(e,T))                         /* kein Zyklus in T ∪{E}? */
                { T= T ∪{e};
                }
        }
        if(#Kanten in T < #Knoten in G-1)
        {       /* keine verfügbaren Kanten mehr, d.h. G ist nicht zusammenhängen */
                printf(„Fehler – kein Spannbaum möglich!“);
        }
        return T;
}
```

Laufzeit: O(|E| log |E|)

KAPITEL 11 STEHT NUR IM SKRIPT. KEINE ZUSAMMENFASSUNG!!!