2 Implementierung von Datentypen

Thema: Transformation beliebiger Datentypen und -strukturen auf den Speicher der Maschine.
Wie bringt man Records, Listen, Bäume, Graphen u.v.m. im Speicher unter?

2.1 Die PASCAL-Maschine

s. Skript.

Beispiele:

a)

```
program ggt(input,output);
var x,y: integer;
function rest (x,y: integer):integer;
begin
    if x>=0 and y>0 then
        while x>=y do x:=x-y;
    rest:=x
end;
function ggt (x,y: integer): integer;
var r: integer;
begin
    if x>=0 and y>0 then
        while y<>0 do
        begin
            r:=rest(x,y);
            x:=y;
            y:=r
        end;
    ggt:=x
end;
begin
    readln(x,y);
    writeln(ggt(x,y))
end.
```

b)

```
program ggt(input,output);
var x,y: integer;
function ggt (x,y: integer): integer;
begin
    if y=0 then ggt:=x else
        if y<=x then ggt:=ggt(y,x) else ggt:=ggt(x,y-x)
end;
begin
    readln(x,y);
    writeln(ggt(x,y))
end.
```

c)

```
program sort (input, output);
type feld = array [1..10] of integer;
var  a: feld;
     k: 1..10;
procedure sortiere(var a: feld);
var i, j: 1..10;
    procedure tausche(var x,y: integer);
    hilf: integer;
    begin
        hilf:= x; x:= y; y:= x
    end;
begin
    for i:= 1 to 9 do
        for j:= i+1 to 10 do
            if a[i] > a[j] then tausche(a[i],a[j])
end;
begin
    for k:= 1 to 10 do read (a[k]);
    sortiere(a);
    for k:= 1 to 10 do write (a[k])
end.
```

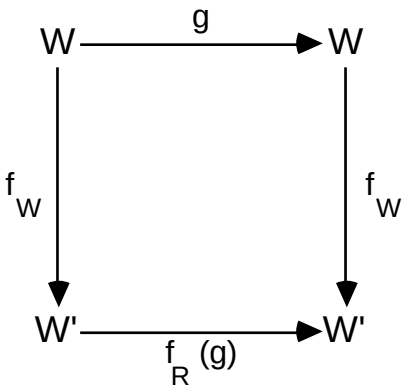
2.2 Begriff der Implementierung von Datentypen

Implementierung D' eines Datentyps D --> Darstellung der Werte von D durch die Werte von D' und Simulation der Operationen von D durch Operationen oder Operationsfolgen von D', so daß sich D' nach außen genauso verhält wie D.

Definition A:
Für eine Menge M von Funktionen sei $M^\circ=\{f_1 \circ f_2 \circ ... \circ f_r \mid f_i \in M, r \geq 1\}$ die Menge aller Kompositionen von Funktionen aus M.
Die **Implementierung** eines Datentyps D=(W,R) durch einen Datentyp D'=(W',R') ist eine Abbildung f: $D \rightarrow D'$ mit $f=(f_W,f_R)$, wobei f_W: $W \rightarrow W'$, f_R: $R \rightarrow R'^\circ$, so daß für alle $x \in W$, $g \in R$ gilt:

$$f_W(g(x))=f_R(g)(f_W(x)).$$

Notiert man die Funktionalität der in Definition A beteiligten Abbildungen graphisch, so erhält man ein sog. *kommutierendes Diagramm:*



Zu klären: Speicherstruktur eines realen Rechners.
Hier: Folge von Speicherzellen gleicher Größe, in PASCAL definiert durch:

type Speicher=array Adressen of Wort.

Hierbei sind die Typen Adressen und Wort maschinenabhängig; in der Regel ist Adressen ein Restriktionstyp von nat, z.B

type Adressen=0..1048575.

Wort ist ein Typ, der die kleinste adressierbare Einheit des Speichers beschreibt.
Aufgabe: Definiere für jeden Datentypen D Implementierungen durch den Datentyp Speicher der Form $f=(f_W,f_R)$: $D \rightarrow$ Speicher.

2.3 Implementierung von Arrays

1-dimensionale Arrays.
Gegeben:

typ A≡array nat [0..n] of T.

Annahme zunächst: Werte von T passen jeweils in ein Wort des Speichers.
Dann bestimme Implementierung

$f=(f_W,f_R)$: A→Speicher

mit: Für $a \in A$ und $sp \in$ Speicher mit

$f_W(a)=sp$

gilt:

$f_W(a(i))=sp[f_R(i)]$ für alle i mit $0 \le i \le n$.

Eigentlich: $f_W(\pi_{i,n}(a))=f_R(\pi_{i,n})(f_W(a))=f_R(\pi_{i,n})(sp)=sp[i']$.
g ist hier die Projektion $\pi_{i,n}$ auf die i-te Komponente von a.

Hier: Implementierung = geeignete Umrechnung der Indizes $i \to i'=f_R(\pi_{i,n})$, so daß $a(i)=sp[i']$. (**Adreßfunktion** (location-function, Abk. loc))

Sei a_0 : Anfangsadresse, ab der a in sp abgelegt werden soll.
Dann:

$i'=loc(i)=f_R(i)=a_0+i$, $0 \le i \le n$.

Allgemein: Werte von T benötigen c Worte:

$i'=loc(i)=a_0+c \cdot i$, $0 \le i \le n$,

und

$f_W(a(i))=sp(f_R(i))=sp(loc(i))...sp(loc(i)+c-1)$.

Mehrdimensionale Arrays.

2-dimensionales Array

typ A≡array (nat [0..m],nat [0..n]) of T.

Nötig: geschickte lineare Anordnung der Elemente eines Objekts a vom Typ A

1. zeilenweise (Abb. 2)
2. spaltenweise
3. diagonalenweise.

Hier: zeilenweise Anordnung (andere Anordnungen siehe Übungen).

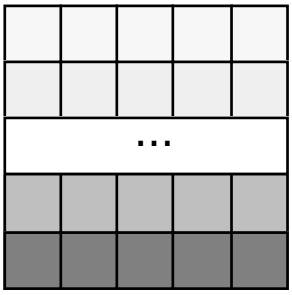


Abb.: Zeilenweise Anordnung eines 2-dimensionalen Arrays im Speicher

Wieder a_0 Anfangsadresse

Dann:

$$loc(i,j)=a_0+c(n+1)\cdot i+c\cdot j,\ 0\le i\le m,\ 0\le j\le n,$$

und

$$f_W(a(i,j))=sp(f_R(i,j))=sp(loc(i,j))...sp(loc(i,j)+c-1).$$

Allgemeiner Fall:

typ A≡array (nat [0..n_1],nat [0..n_2],...,nat [0..n_k]) of T.

Adreßfunktion:

$$\begin{aligned} loc(d_1,d_2,...,d_k) = \; & a_0 \\ & +c\cdot i_1(n_2+1)\cdot...\cdot(n_k+1) \\ & +c\cdot i_2(n_3+1)\cdot...\cdot(n_k+1) \\ & \ldots \\ & +c\cdot i_{k-1}(n_k+1) \\ & +c\cdot i_k \\ = \; & a_0+\sum_{j=1}^{k} c_j\cdot i_j \end{aligned}$$

mit $c_j=c\prod_{l=j+1}^{k}(n_l+1)$.

Beachte Effizienz: loc-Funktion sollte lineare Funktion sein, um beim Zugriff auf ein Array-Element schnelle Auswertung zu sichern.

2.4 Implementierung von Records

Gegeben:

$\underline{typ}\ T \equiv (t_1 : T_1, t_2 : T_2, ..., t_n : T_n)$

Implementierung: Lege Komponenten $x.t_1,...,x.t_n$ eines Objekts $x \in T$ beginnend bei a_0 sequentiell ab:

$loc(t_i)=a_0+c_1+c_2+...+c_{i-1}$.

Hier benötigt jedes Objekt aus T_i jeweils c_i Speicherzellen.

$c_1+c_2+...+c_{i-1}$ heißen **Offsets**.

	a_0
t_1	0
t_2	c_1
...	...
t_n	$c_1+c_2+...+c_n$ -1

Abb.: Tabelle mit Anfangsadresse und Offsets

2.5 Implementierung von Mengen

Gegeben:

$\underline{\text{typ}}\ D \equiv 2^{D'}$

Vorauss.: Wertemenge des Grundtyps D' **endlich**.

Implementierung einer Menge $M \in D$ durch Übergang zur *charakteristischen Funktion*

$\chi_M: D' \rightarrow \text{bool}$ mit

$$\chi_M(x) = \begin{cases} \text{true}, & \text{falls } x \in M, \\ \text{false}, & \text{sonst.} \end{cases}$$

D' endlich ==> charakteristische Funktion durch endliche Folge von n booleschen Werten (0-1-Folgen (0 für false, 1 für true)) in einem oder mehreren Speicherworten darstellen:

$\chi_M = (\chi_M(d_1),...,\chi_M(d_n))$

Beispiel:

$M = \{1,3,4,7\} \in 2^{\{1,2,3,4,5,6,7,8,9,10\}}$

Darstellung:

1 0 1 1 0 0 1 0 0 0.

Mengenoperationen entsprechen booleschen Funktionen:

Vereinigung ==> stellenweise ODER-Funktion
Durchschnitt ==> stellenweise UND-Funktion
Komplement ==> stellenweise Negation
Elementabfrage $d_i \in M$ ==> Test, ob i-te Stelle =1

sehr effizient realisierbar durch spezielle Prozessoroperationen.

2.6 Implementierung der Parameterübergabe

Im folgenden x aktueller, a formaler Parameter.

call by value.
Implementierung: Kopieren des Wertes von x in einen neuen gleichgroßen Speicherbereich, der beim Aufruf angelegt und unter dem Bezeichner a angesprochen wird.

Falls x Objekt eines strukturierten Typs (etwa ein großes Array), so werden alle Werte einzeln kopiert (hoher Speicher- und Zeitaufwand)
Hier besser: call-by-reference-Übergabe.

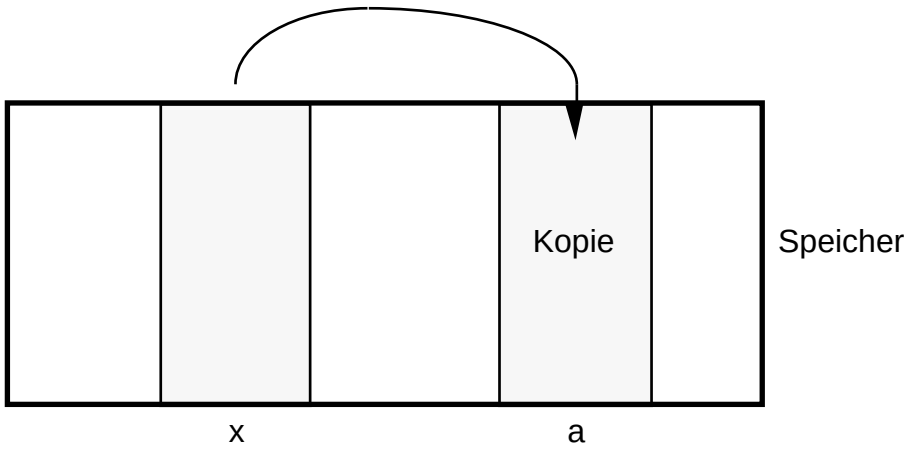


Abb.: call-by-value-Übergabe

call by reference.

Wirkung: formaler Parameter a innerhalb der Prozedur oder Funktion Synonym für den aktuellen Parameter x.

Implementierung: Anlegen einer neuer Speicherzelle, die unter dem Bezeichner a angesprochen und in der ein Verweis (ein **Zeiger**, eine **Referenz**) auf x abgelegt wird. Der Verweis besteht aus der Adresse der ersten Speicherzelle, an der x im Speicher beginnt. Keine Kopiervorgänge nötig.

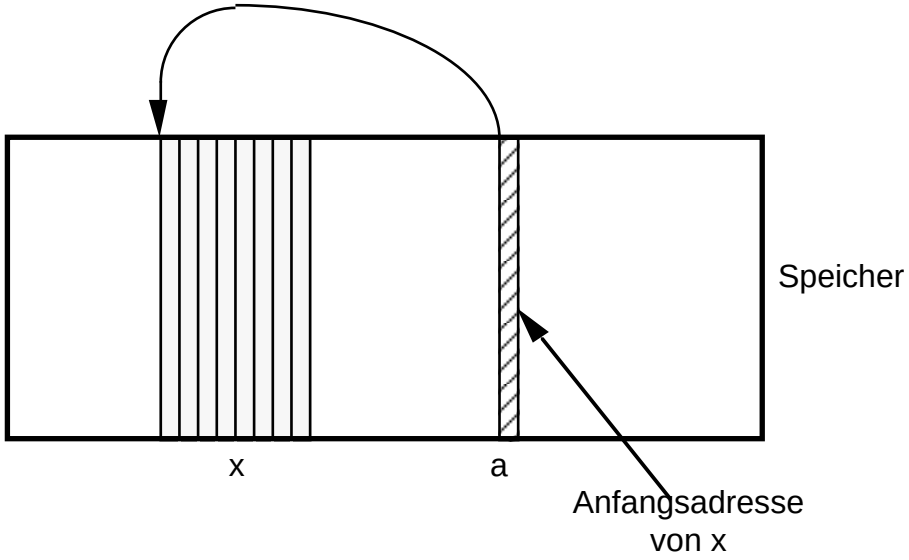


Abb.: call-by-reference-Übergabe

2.7 Implementierung von Stacks und Queues

1. **sequentielle Speicherverfahren:** Einträge werden der Reihe nach in aufeinanderfolgenden Speicherzellen abgelegt
2. **verkettete Speicherverfahren:** Einträge an beliebigen Stellen im Speicher verstreut, Herstellung der Reihenfolge durch Verweise/Zeiger/Referenzen zwischen den Daten

Stacks.

Implementierung durch Arrays:

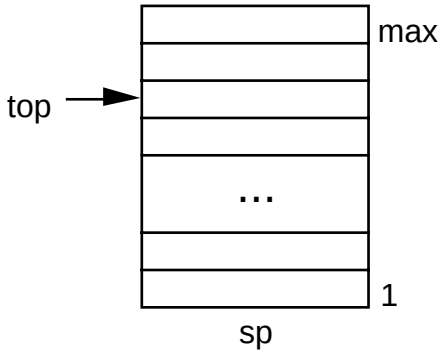


Abb.: Stack

Wahl der **Parameterübergabeart** bei push und pop: call-by-reference, weil Modifikationen des Stacks nach außen dringen sollen.

top und is_empty: call-by-reference-Übergabe, damit durch call-by-value-Übergabe keine Kopien angelegt werden:

```
const max=...;
type data=...;
     stack=record
               top: 0..max;
               sp: array [1..max] of data
             end;
function is_empty(var s: stack): boolean;
begin
     is_empty:=(s.top=0)
end;
procedure push(x: data; var s: stack);
begin
     with s do
     begin
          if top=max then "Overflow" else
          begin
               top:=top+1;
               sp[top]:=x
          end
     end
end;
procedure pop(var s: stack);
begin
     with s do
          if is_empty(s) then "Underflow" else top:=top-1
end;
function top(var s: stack): data;
begin
     if is_empty(s) then "Underflow" else top:=s.sp[s.top]
end;
procedure empty(var s: stack);
begin
     s.top:=0
end.
```

Initialisierung:

```
var s: stack;
empty(s).
```

Beispiel: Syntaxanalyse von Zeichenfolgen auf korrekte Klammerung.
Definition durch BNF-Grammatik G=(N,T,P,S) wie folgt: N={<Wort>}, T={(,),[,]}, S=<Wort>, und P enthält die Produktion:

<Wort>::= (<Wort>) <Wort> | [<Wort>] <Wort> | ε.

L(G) besteht aus allen Zeichenfolgen mit den Klammern (,),[,], die als korrekte Klammerung anzusehen sind.

Idee: "Klammer auf" -> push on stack
"Klammer zu" -> pop off stack
Ungleichheit/leerer Stack/nichtleerer Stack am Ende
-> Syntaxfehler

Das Programm:

```
program Klammersyntax (input,output);
type data=char;
<hier steht die Definition des Stacks und der Zugriffsprozeduren>
var s: stack;
    ch: char;
procedure error;
begin
    writeln('Syntaxfehler')
end;
begin
    empty(s);
    while not eof do
    begin
        read(ch);
        case ch of
            '(','[': push(s,ch);
            ')': if is_empty(s) then error else
                        if top(s)='(' then pop(s) else error
            ']': if is_empty(s) then error else
                        if top(s)='[' then pop(s) else error
            otherwise: error
        end;
    end;
    if is_empty(s) then writeln('Korrekte Klammerung') else error
end.
```

Queues.

Implementierung durch Array (als Ring zusammengebogen)
Zwei Indizes anfang und ende m
anfang weist immer auf das Feldelement unmittelbar *vor* Beginn der Queue.

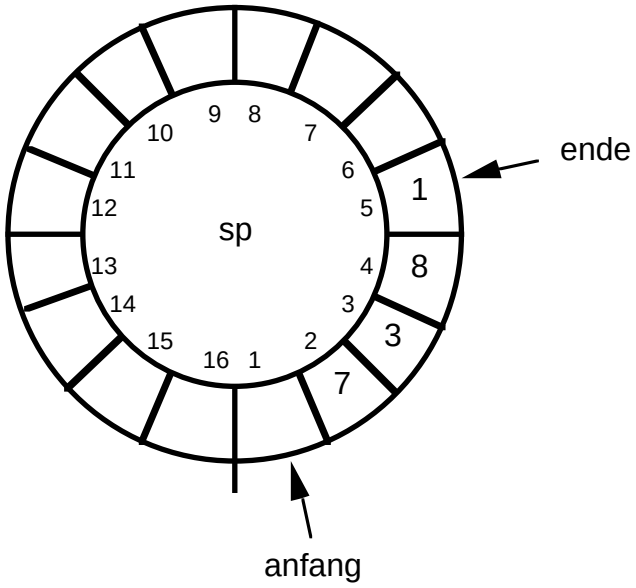


Abb. : Queue

Programm:

```
const max=...;
type data=...;
     queue=record
               anfang, ende: 1..max;
               sp: array [1..max] of data
           end;
procedure enter(x: data; var q: queue);
begin
     with q do
     begin
          if ende=max then ende:=1 else ende:=ende+1;
          if ende=anfang then "Overflow" else sp[ende]:=x
     end
end;
function is_empty(var q: queue): boolean;
begin
     is_empty:=(q.anfang=q.ende)
end;
procedure remove(var q: queue);
begin
     with q do
          if is_empty(q) then "Underflow" else
               if anfang=max then anfang:=1 else anfang:=anfang+1
end;
function first(var q: queue): data;
begin
     if is_empty(q) then "Underflow" else first:=q.sp[q.anfang]
end;
procedure empty(var q: queue);
begin
     q.anfang:=max; q.ende:=max
end.
```

Initialisierung:

```
var q: queue;
empty(q).
```

Overflows und Underflows.

Underflow (*Unterlauf*): Situation, aus einer leeren Datenstruktur ein Element entfernen zu wollen. Meist fehlerhaftes Programm
Overflow (*Überlauf*): in eine volle Datenstruktur ein Element einfügen zu wollen. Meist zuwenig Speicher, also zu kleinem $\max$.

Elegantere Lösung bei mehreren Stacks (oder Queues)

Gegeben: Gewisse Anzahl von Stacks hintereiannder im Speicher

Overflow, wenn die Summe der Längen aller Stacks $\max$ überschreitet.

langsames und schnelles Wachstum einzelner Stacks gleichen sich aus
=> bessere Ausnutzung des Speichers und
geringere Overflow-Häufigkeit.

Annahme:

- Speicher mit l Speicherzellen:

  type Speicher=array [1..l] of data.
- n homogene Stacks der Reihe nach
- top-Zeiger wie bisher (jetzt als Array)
- base-Zeiger für den Anfang jedes Stacks (als Array)
- base[n+1] nur zur Vereinfachung der Algorithmen. Beide Mengen implementieren wir als Arrays.
- Stack-Operationen um Nummer des angesprochenen Stacks ergänzen
- Überlauf ==> Einfügen eines Elementes führt zu Überschneidung zweier Stacks (top[i]>base[i+1]).

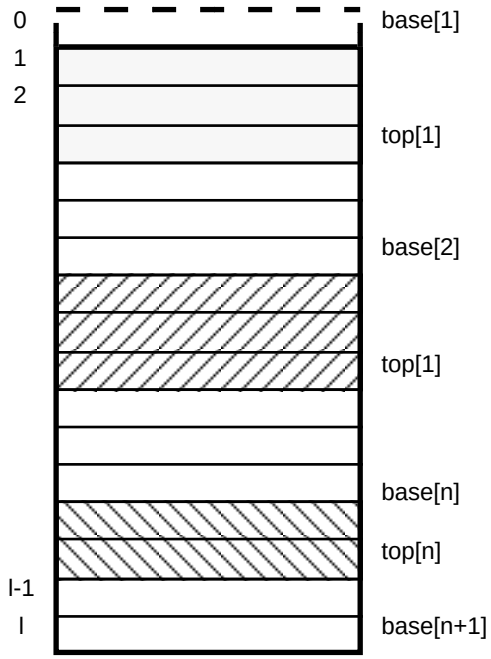


Abb.: n homogene Stacks

Definition:

```
type data=...;
     Speicher=array [1..l] of data;
     anzahl=1..n;
     n_stack=record
               top: array [anzahl] of 0..l+1;
               base: array [1..n+1] of 0..l;
               sp: Speicher
                end;
procedure push(x: data; i: anzahl; var s: n_stack);
begin
     with s do
     begin
          top[i]:=top[i]+1;
          if top[i]>base[i+1] then "Overflow" else sp[top[i]]:=x
     end
end;
function is_empty(i: anzahl; var s: n_stack): boolean;
begin
     is_empty:=(s.top[i]=s.base[i])
end.
procedure pop(i: anzahl; var s: n_stack);
begin
     with s do
          if is_empty(i,s) then "Underflow" else top[i]:=top[i]-1
end;
function top(i: anzahl; var s: n_stack): data;
begin
     with s do
          if is_empty(i,s) then "Underflow" else top:=sp[top[i]]
end;
procedure empty(var s: n_stack);
var i: integer;
begin
     with s do
     begin
          base[n+1]:=l;
          for i:=1 to n do
          begin
               base[i]:=(l*(i-1)) div n;
               top[i]:=(l*(i-1)) div n
          end
     end
end.
```

Initialisierung

```
var s: stack;
empty(s)
```

Reaktion bei Overflow beim Stack i:

1. Speicher ist vollständig belegt ==> Fehlermeldung
2. Speicher ist noch nicht vollständig belegt: Verschiebe alle Stacks, daß bei Stack i genau ein freier Platz entsteht ==> langfristig unklug.

Verbesserung: **Garwick-Algorithmus**

- Strebe längerfristige Lösung an
- Sorge dafür, daß bis zur nächsten Umordnung eine möglichst große Zahl von Stack-Operationen möglich wird.
- Weise jedem Stack mehrere Speicherzellen zu.
- Die genaue Zahl richtet sich nach dem Verhalten des Stacks in der Vergangenheit:
    - stark wachsende Stacks: mehr zusätzliche Plätze
    - schwach wachsende Stacks: weniger zusätzliche Plätze
      Hintergrund: *Lokalitätsprinzip*: Programme verhalten sich innerhalb einer gewissen zukünftigen Zeitspanne etwa so, wie sie sich innerhalb ihren unmittelbaren Vergangenheit verhalten haben.

Ergänzung der Implementierung für n_stacks um einige Komponenten:

```
type n_stack=record
                ...
                oldtop: array [anzahl] of 0..l;
                newbase: array [1..n+1] of 0..l;
                d: array [anzahl] of 0..l
                 end;
```

Die Prozedur zur Behandlung eines Overflows wird wie folgt implementiert:

```
procedure overflow(var s: n_stack);
var a,b: real;
    sum, inc: integer;
    j: 1..n;
begin
    with s do
    begin
        sum:=l; inc:=0;
        for j:=1 to n do
        begin
            sum:=sum-(top[j]-base[j]);
            if top[j]>oldtop[j] then
            begin
                d[j]:=top[j]-oldtop[j];
                inc:=inc+d[j]
            end else d[j]:=0;
        end;
        if sum<0 then "Fehler: Speicher ist voll" else
        begin
            a:=(0.1*sum)/n; b:=(0.9*sum)/inc;
            newbase[1]:=base[1];
            for j:=2 to n do
                newbase[j]:=newbase[j-1]+(top[j-1]-base[j-1])+
                        trunc(a)+trunc(b*d[j-1]);
            umordnen(s);
            for j:=1 to n do oldtop[j]:=top[j]
        end
    end
end;
procedure umordnen(var s: n_stack);
var j,j1: 2..n;
    k: 2..n+1;
begin
    with s do
    begin
        j:=2;
        while j<=n do
        begin
            k:=j;
            if newbase[k]<=base[k] then verschieben(s,k) else
            begin
                while newbase[k+1] >base[k+1] do k:=k+1;
                for j1:=k downto j do verschieben(s,j1)
            end;
            j:=k+1
        end
    end
```

```
end;
procedure verschieben(var s: n_stack; m: integer);
var delta: integer;
    j2:= 0..l;
begin
    with s do
    begin
        delta:=newbase[m]-base[m];
        if delta<>0 then
        begin
            if delta>0 then
                for j2:=top[m] downto base[m]+1 do
                    sp[j2+delta]:=sp[j2]
            else
                for j2:=base[m]+1 to top[m] do sp[j2+delta]:=sp[j2];
            base[m]:= newbase[m];
            top[m]:=top[m]+delta
        end
    end
end.
```

Analoge Implementierung für alle vergleichbaren Datentypen, z.B. Queues.

Algorithmische Analyse.

- Bisher keine exakten theoretischen Effizienzanalysen vor
- Ergebnisse experimenteller Untersuchungen: Garwick-Algorithmus sehr effizient, wenn der Speicher etwa zur Hälfte gefüllt ist.
- Problem: Kurz vor globalem Überlauf ungeheuerer Zeitverbrauch Möglicher Lösungsansatz: Melde bereits Überlauf, wenn die Zahl der freien Speicherplätze ein gewisses Minimum $min>0$ unterschreitet.

Zusammenfassung.

Merkmale sequentieller Verfahren:
- erlauben schnellen direkten Zugriff auf Datenelemente,
- erfordern kompliziertes Einfügen von Datenelementen, meist mit Umordnung,
- sorgen oft für unzureichende Speicherausnutzung, weil für evtl. einzutragende Elemente Platz reserviert werden muß.

2.8 Implementierung von Rekursion

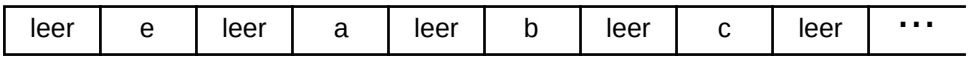
Rekursive Datentypen.

Motivation: Definition eines markierten binären Baumes:

typ Markierung ≡ {a,b,c,d,e};
typ Baum ≡ {leer} | (Baum,Markierung,Baum).

Ein Objekt vom Typ Baum:

B=((leer,e,leer),a,((leer,b,leer),c,(leer,b,leer))).



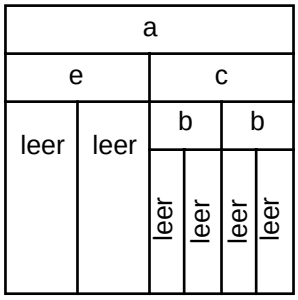
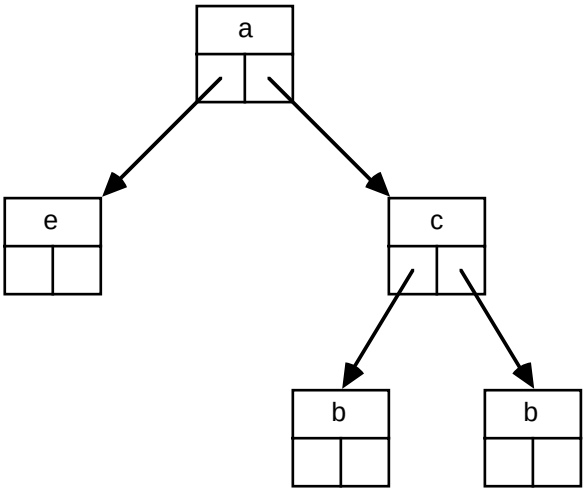


Abb.: Ungeeignete Speichermöglichkeiten für Bäume (Einfüge- und Löschproblem)

Lösung: Referenztechnik: anstelle einer Schachtelungsstruktur Nutzung einer verketteten Struktur



... oder als mögliche Anordnung im Speicher ...

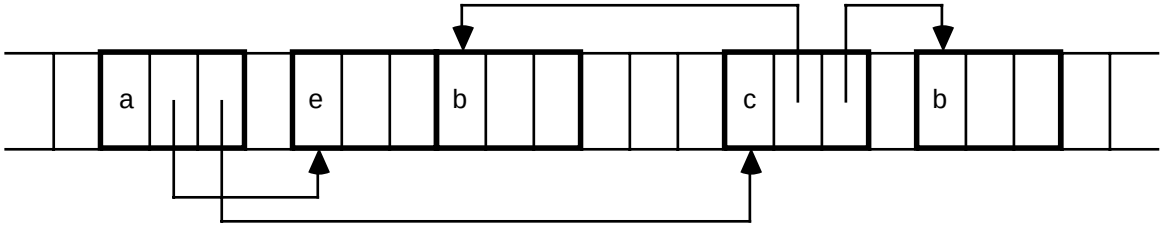


Abb.: Verkettete Speichermöglichkeit für Bäume

Vorteile:
- Bausteine eines Baumes sind identische, einfach zu beschreibende Objekte desselben Grundtyps;
- Bausteine können an beliebige freie Stellen im Speicher abgelegt werden. Ein- und Ausfügen wird vergleichsweise unkompliziert.

Definition in PASCAL:

```
type Baum=record
           marke: markierung;
           lt, rt: ↑Baum
         end.
```

Wertemenge: Menge aller Verweise auf Werte vom Typ Baum ist.
Aus Maschinensicht: Adressen von Speicherzellen oder Gruppen aufeinanderfolgender Speicherzellen, in denen Objekte vom Typ Baum abgelegt werden können.

2.8.1 Zeiger

Definitionsschema: T beliebiger Datentyp:

type refT=↑T

Wertemenge W von refT ist die Menge aller Zeiger auf Objekte vom Typ T.

Universelle Konstante

nil∈W.

(verweist zur Zeit auf kein Objekt (Beachte: nil ist nicht dasselbe wie undefiniert).

Operationsmenge:

- new

  new(p)

  erzeugt (neues) Objekt vom Typ T und weist p Referenz auf Objekt zu.
- dispose

  dispose(p)

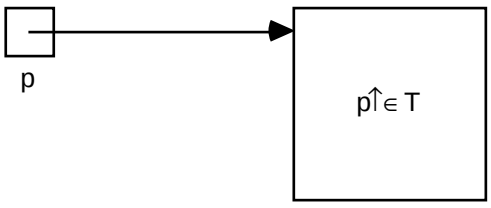
  löscht Speicherbereich, den das Objekt belegt, auf das p weist.
- ↑ (Dereferenzierung).

  (Postfix-)Operation ↑ realisiert Dereferenzierung eines Zeigerobjekts:

  p↑

  ist das Objekt, auf das p verweist; also p↑∈T.

  Unterscheide: p:=q und p↑:=q↑. Verboten sind p:=q↑ und p↑:=q.



... oder als Speicherbild ...

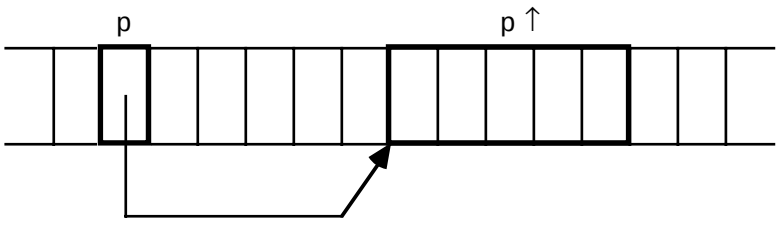


Abb.: Wirkung von new

Beispiel:

```
type T=record
          x: integer
       end;
var p, q, r: ↑T

new(p); p↑.x:=5;
new(q); q↑.x:=7;
new(r); r↑.x:=9;
p:=q;
r↑:=q↑;
```

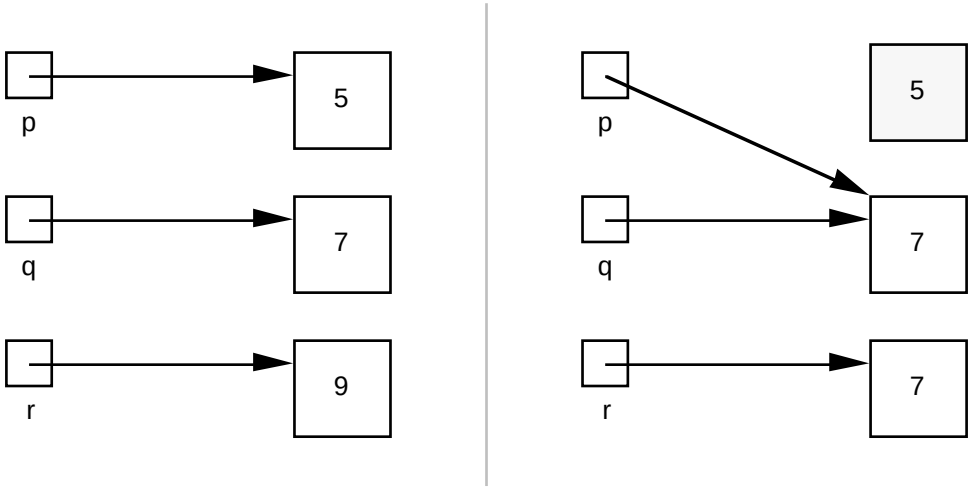


Abb.: Programmsituationen

Zeiger ermöglichen effiziente Implementierung einer Vielzahl rekursiver Datentypen.

2.8.2 Implementierung von Sequenzen

Polymorphe Linkssequenz der Form

typ L(D)≡{leer} | (D,L)

PASCAL-Implementierung:

```
type T=record
         inhalt: D
         next: ↑T
       end.
```

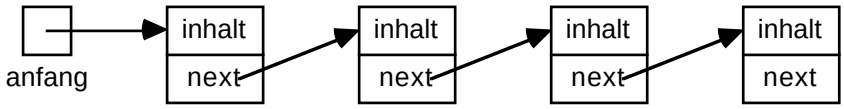


Abb.: Implementierung einer Links-/Rechtssequenz als (einfach verkettete) lineare Liste

Typische Operationen:

Vorausgesetzte Deklarationen

```
var p,q,r,anfang: ↑T
```

Aufbau einer Liste mit n Elementen, deren jeweiliger Inhalt $t_1,...,t_n$ ist:

```
anfang:=nil;
for i:=n downto 1 do
begin
    new(p);
    p↑.next:=anfang;
    p↑.inhalt:=ti;
    anfang:=p
end.
```

Einfügen eines Listenelementes mit dem Inhalt t hinter p↑:

```
new(q);
q↑.inhalt:=t;
q↑.next:=p↑.next;
p↑.next:=q.
```

Entfernen des Nachfolgers von p↑:

```
p↑.next:=p↑.next↑.next.
```

Entfernen des Elements p↑ selbst unter der Voraussetzung, daß p↑ einen Nachfolger besitzt:

p↑:=p↑.next↑.

Beispiel: Einlesen einer beliebigen Folge von Zeichen und Ausgabe in umgekehrter Reihenfolge:

```
program palindrom(input,output);
type zeichen=record
                inhalt: char;
                next: ↑zeichen
             end;
var p, anfang: ↑zeichen;
    ch: char;
begin
    anfang:=nil;
    while not eof do
    begin
        new(p); p↑.next:=anfang;
        read(ch); p↑.inhalt:=ch;
        anfang:=p
    end;
    while anfang≠nil do
    begin
        write(anfang↑.ch);
        anfang:=anfang↑.next
    end
end.
```

Weitere Implementierungen mit verbesserten Zugriffmöglichkeiten

Kreisförmige Verkettung.
next-Zeiger des letzten Listenelements verweist wieder auf den Anfang der Liste.

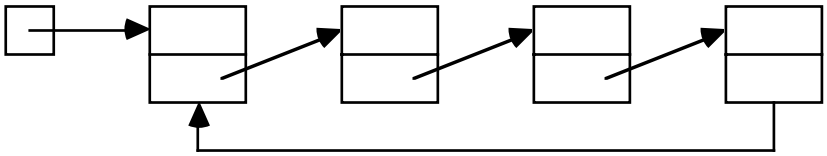


Abb.: Kreisförmig verkettete lineare Liste

Doppelte Verkettung.
Jedes Listenelement besitzt zwei Zeiger, einen auf den Vorgänger und einen auf den Nachfolger. Zusätzlich gibt es ein ausgezeichnetes Listenelement, den Kopf, der als Anker der Liste dient, keine Information trägt und nicht gelöscht werden darf. Bei doppelter Verkettung vereinfacht sich das Ein- und Ausfügen von Listenelementen. Zugleich ist ein beliebiges Vor- und Zurücklaufen möglich. Zur Implementierung wird auf die Übungen verwiesen.

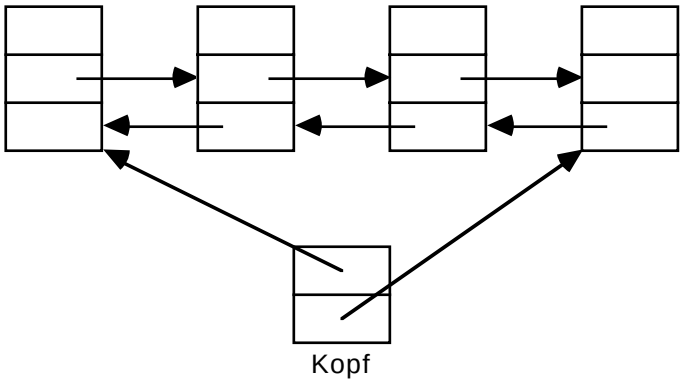


Abb.: Doppelt verkettete lineare Liste

2.8.3 Implementierung von Bäumen

Allgemeinster Fall: Anzahl der Söhne nicht bekannt

- ==> lineare Liste
- Array von Zeigern mit durchschnittlich vielen Komponenten. Überschreitet die Zahl der Söhne die Arraygröße, so erzeugt man einen Hilfsknoten.

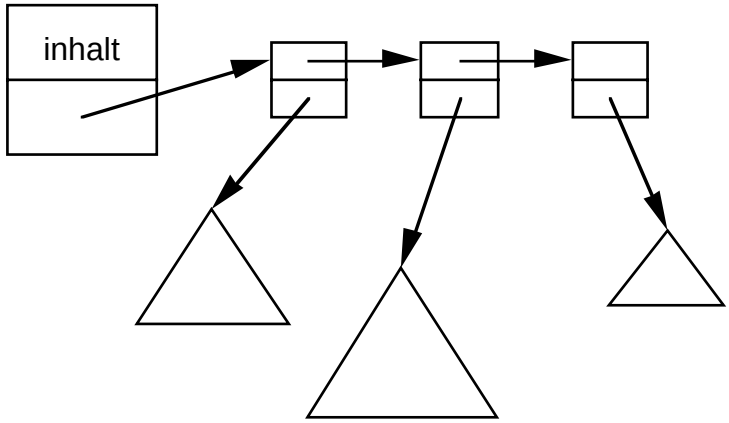


Abb.: Implementierung von Bäumen durch lineare Listen

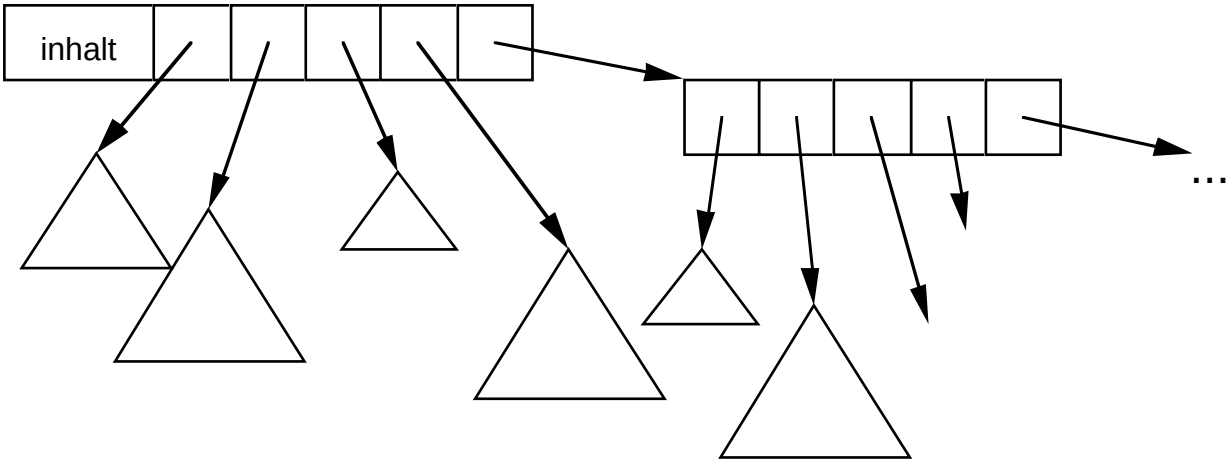


Abb.: Implementierung von Bäumen durch Arrays mit Überlauf

Nachteil: Speicherausnutzung.
- Entweder je Knoten mit k Söhnen 2k+1 Zeiger
- oder Vielzahl von Zeigern ungenutzt, nämlich die von den Blättern.

Beispiel:
- Sei B ein Baum mit n Knoten
- je Knoten mit Ausnahme der Blätter jeweils k Söhne.
- Dann besitzt B also nk Zeiger. Wegen der n Knoten sind genau n-1 Zeiger ungleich nil. Folglich sind n(k-1)+1 Zeiger gleich nil und damit praktisch überflüssig. Schon bei ternären Bäumen ca. 2/3 der Zeiger überflüssig.

Daher: binäre Bäume
- bestmögliche Speicherausnutzung
- effiziente Simulation von Bäumen beliebiger Ordnung

Idee: definieren einen der beiden Verweise um:

linker Zeiger <-> erster Sohn des Knotens
rechter Zeiger <-> Bruder des Knotens.

Idee überträgt sich auf Wälder. Formal:

Sei $F=(T_1,...,T_n)$ ein Wald. Für einen Baum T bezeichne w(T) die Wurzel, für einen Knoten x sei lt(x) der linke und rt(x) der rechte Sohn. Der zu F gehörige binäre Baum B(F) ist wie folgt definiert:

1) Für n=0 ist B(F) der leere Baum.
2) Für n>0 ist

   $w(B(F))=w(T_1)$,
   $lt(w(B(F)))=B(F')$ mit F'=Wald der Teilbäume von $w(T_1)$,
   $rt(w(B(F)))=B(T_2,...,T_n)$.

Beispiel: s. Übungen.

Im folgenden **immer binäre Bäume**, definiert wie folgt:

```
type data=...
    knoten=record
              inhalt: data;
              lt, rt: ↑knoten
           end.
```

Ein Objekt

```
var b: ↑knoten
```

repräsentiert dann einen Baum. Für b=nil ist der Baum leer.

Baumdurchlauf.

inorder-Durchlauf:

```
procedure inorder (b: ↑knoten);
begin
    if b<>nil then
    begin
        inorder(b↑.lt);
        write(b↑.inhalt);
        inorder(b↑.rt)
    end
end.
```

Aufbau eines Baumes.

Ordne Knoteninhalte z.B. in preorder-Reihenfolge an mit Ergänzung der Stellen, an denen ein Teilbaum leer ist (Bindestrich).

Beispiel:

ABC--DE--FG---HI--JKL--M--N--

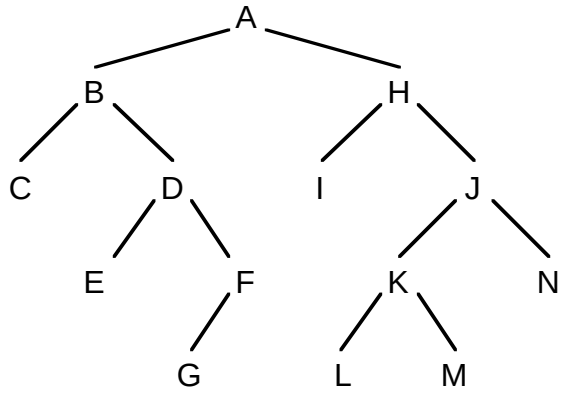


Abb.

Den Aufbau leistet nun die Prozedur:

```
procedure Aufbau(var p: ↑knoten);
var ch: char;
begin
    read(ch);
    if ch<>'-' then
    begin
        new(p); p↑.inhalt:=ch;
        Aufbau(p↑.lt);
        Aufbau(p↑.rt)
    end else p:=nil
end.
```

2.8.4 Rekursion im Kontrollbereich

Entscheidendes Hilfsmittel zur effizienten Realisierung von Rekursio: **Stack**.

Beispiel:

```
funktion ggT (a:nat,b:nat) → nat ≡
        wenn a=b dann a sonst
                wenn a<b dann ggT b a sonst ggT (a-b,b) ende ende.
```

Vorgehen:

1. Schritt: Man beseitigt geschachtelte Funktionsaufrufe. Ggf. lokale Hilfsvariablen einführen.
   Beispiel: Ersetze den Aufruf "f(g(x))" durch "h:=g(x); f(h)".

2. Schritt: Der Rumpf der Funktion wird am Programmanfang eingefügt und erhält zu Beginn eine Marke X als Sprungziel. Eine weitere Marke Y steht unmittelbar hinter dem Rumpf. Vor den Rumpf wird ein Sprung auf Y eingefügt. Der Rumpf soll ja nicht gleich zu Beginn ausgeführt werden.

3. Schritt: Definiere Stack, bei dem jeder Eintrag in Form eines Records alle Parameter, alle lokalen Variablen, eine Variable für den berechneten Funktionswert einer Inkarnation sowie die Rücksprungadresse enthält, zu der nach Abarbeitung der Funktion verzweigt werden muß (direkt hinter der Aufrufstelle). Der Funktionsrumpf arbeitet nur mit dem obersten Stackelement. Eigene Variablen oder Parameter besitzt er nicht.

   Beispiel: Definiere den Typ data für die Funkton ggT durch

```
type data=record
            a,b: integer;          für die Parameter
            marke: (M1,M2,M3); für die drei möglichen
                                   Rücksprungmarken zu den drei
                                   Aufrufen von ggT
            ggT: integer           für den Funktionswert
          end.
```

4. Schritt: Jeder Funktionsaufruf wird in dieser Reihenfolge ersetzt durch
   a) eine Zuweisung der Parameter und der Rücksprungadresse (s. d)) an ein neues Record-Element
   b) eine push-Operation dieses Records auf den Stack
   c) einen Sprung zur Anfangsmarke des Funktionsrumpfs
   d) die Einführung einer Rücksprungmarke.

5. Schritt: Am Schluß des Funktionsrumpfes wird folgende Anweisungsfolge eingefügt:
   a) Auslesen der Rücksprungadresse aus dem obersten Stackelement
   a) Auslesen des Funktionswertes aus dem obersten Stackelement
   b) Entfernen des obersten Stackelementes
   c) Weitergabe des Funktionswertes an das jetzt oberste Stackelement
   d) Sprung zur Rücksprungadresse.

Bemerkung: Nicht alle oben beschriebenen Schritte funktionieren unmittelbar in PASCAL. Dies betrifft vor allem die Verwendung von Marken. Zur konkreten Realisierung schauen Sie am besten in ein PASCAL-Handbuch.

Beispiel:

```
funktion ggT (a:nat,b:nat) → nat ≡
wenn a=b dann a sonst
    wenn a<b dann ggT (b,a) sonst ggT (a-b,b) ende ende.
```

1. Schritt: Eigentlich ist hier ggT (a-b,b) durch c:=a-b; ggT (c,b) zu ersetzen. Dies können wir uns aber sparen, weil die Subtraktion elementar ist (wir haben in diesen Fällen auch bei der Formularmaschine kein neues Formular angelegt.)

2. Schritt: Aufstellung des Programms:

```
program ggt(input,output);
begin
    goto B;
    A: wenn a=b dann a sonst
        wenn a<b dann ggT (b,a) sonst ggT (a-b,b) ende ende;
    B: readln(a,b);
    h:=ggt(a,b); writeln(h)
end.
```

3. Schritt: Stackdefinition:

```
program ggt(input,output);
type data=record
            a,b: integer;
            marke: (M1,M2,M3);
            ggt: integer
        end;
    stack=... <übliche Definition unter Verwendung von data>
begin
    goto B;
    A: if a=b then a else
        if a<b then ggT (b,a) else ggT (a-b,b);
    B: readln(a,b);
    h:=ggt(a,b); writeln(h)
end.
```

4. und 5. Schritt: Modifikation der Aufrufe und Beendigung des Funktionsrumpfes:

```
program ggt(input,output);
type data=record
            a,b: integer;
            marke: (M1,M2,M3);
            ggt: integer
         end;
     stack=... <übliche Definition unter Verwendung von data>
var x: data;
     s: stack;
begin
     goto B;
     A: if s.sp[s.top].a=s.sp[s.top].b then s.sp[s.top].ggT:=s.sp[s.top].a
         else
         if s.sp[s.top].a<s.sp[s.top].b then
         begin
            x.a:=s.sp[s.top].b; x.b:=s.sp[s.top].a; x.marke:=M1;
            push(x,s);
            goto A;
            M1:
         end else
         begin
            x.a:=s.sp[s.top].a-s.sp[s.top].b; x.b:=s.sp[s.top].b;
            x.marke:=M2; push(x,s);
            goto A;
            M2:
         end;
     x:=top(s);
     pop(s);
     s.sp[s.top].ggT:=x.ggT
     goto x.marke;
     B: readln(a,b);
     x.a:=a; x.b:=b; x.marke:=M3; push(x,s);
     goto A;
     M3:
     h:=s.sp[s.top].ggT; writeln(h)
end.
```

Kritische Größe: Stack s, wegen Speicherbedarf und wegen Laufzeit für die Zugriffsoperationen

Grundsätzliche Vorüberlegung:

- Ist die Rekursion effizient?
- Haben Rekursionstiefe und -breite einen relativ zum Problem vernünftigen Umfang? Ggf. Übergang zu einem iterativen Algorithmus.

Beispiel: Fibonacci-Funktion:

$f(0)=0$,
$f(1)=1$,
$f(n)=f(n-1)+f(n-2)$ für $n \geq 2$,

Größe des Stacks wächst exponentiell mit n.
Eine effiziente nicht-rekursive Lösung ist ebenso übersichtlich wie die rekursive.

Rekursionsvermeidung (ohne Stack)

Typenklasse: rekursiver Aufruf stets am Anfang (*head recursion*) oder stets am Ende (*tail recursion*).

==> Umwandlung in while- oder repeat-Schleife.

Genauer: Das Funktionsschema

```
funktion P x → ... ≡
wenn B dann a sonst g(x,P(f(x))) ende
```

mit der *kommutativen* Funktion g und dem Aufruf P(y) ersetzt man durch das Schleifenschema

```
P:=a; j:=y;
while not B do
begin
    P:=g(j,P);
    j:=f(j)
end.
```

Beispiel: Für die Fakultätsfunktion

```
funktion fak x: nat→nat ≡
wenn x=0 dann 1 sonst x*fak (x-1) ende.
```

erhält man für den Aufruf fak n das Programmstück

```
fak:=1; j:=n;
while not (x=0) do
begin
    fak:=j*fak;
    j:=j-1
end.
```

2.9 Graphen

Mathematisches Modell zur Beschreibung von Objekten, die untereinander in gewisser Beziehung stehen:

- chemische Strukturformeln
- Verkehrsnetze
- Verwandschaftsbeziehungen.

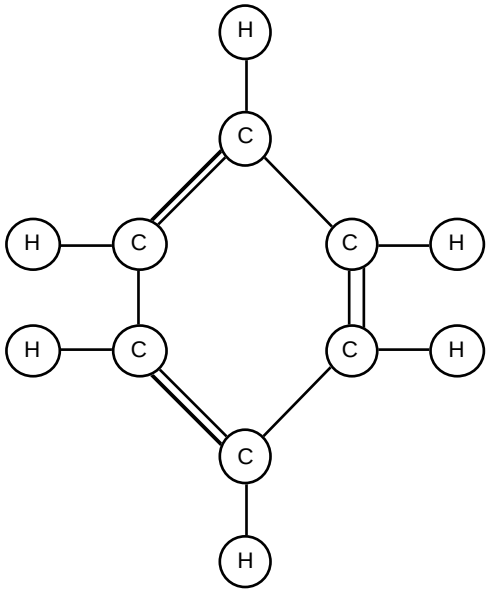


Abb.: Chemische Strukturformel für Benzol

Ursprung: Königsberger Brückenproblem (L. Euler)

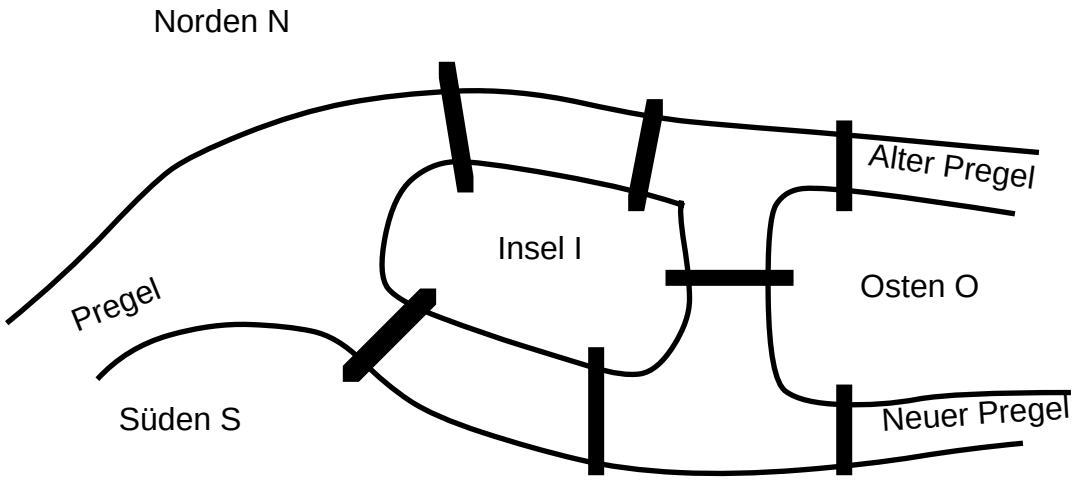


Abb.: Königsberger Brückenproblem

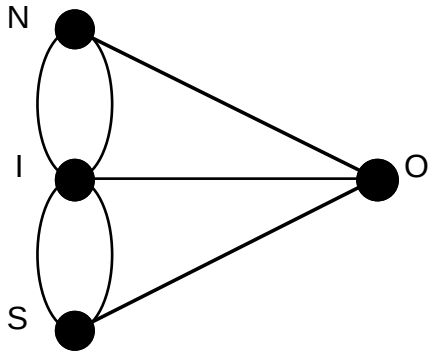


Abb.: Graph zum Königsberger Brückenproblem

Suche *Eulerschen Kreis*.
Eulerscher Kreis ex. <==> Grad jedes Knotens ist gerade

Definition B:
Sei $V \neq \emptyset$ eine endliche Menge und E eine Menge von ein- und zweielementigen Teilmengen von V. Dann heißt G=(V,E) **ungerichteter Graph**. V (oder V(G), wenn Unterscheidungen nötig sind) ist die Menge der **Knoten**, E (oder E(G)) die Menge der **Kanten**. Ist $\{x,y\} \in E$ eine Kante, so sind x und y **adjazent**. Der **Grad** d(v) eines Knotens v ist definiert durch $d(v)=|\{u \in V \mid \{u,v\} \in E\}|$.
Sei $P=(v_0,v_1,...,v_{n-1},v_n)$ ein (n+1)-Tupel von Knoten von G. P ist ein **Weg** der Länge n zwischen v_0 und v_n, wenn $\{v_{i-1},v_i\} \in E$ für $1 \leq i \leq n$. P heißt **einfach**, wenn $v_i \neq v_j$ für $0 \leq i < j \leq n$. Ein Weg $P=(v_0,v_1,...,v_{n-1},v_n)$ heißt **Zyklus** der Länge n, falls $n \geq 3$, $v_n=v_0$ und $(v_0,v_1,...,v_{n-1})$ ein einfacher Weg ist. Ein Graph ist **azyklisch**, wenn er keinen Zyklus enthält. u und v sind **verbunden**, falls es einen Weg zwischen u und v gibt. G ist **zusammenhängend**, wenn jedes Paar $x,y \in V$ verbunden ist.
Ein Graph G'=(V',E') heißt **Teilgraph** von G, falls $V' \subseteq V$ und $E' \subseteq E$ ist. Ein zusammenhängender Teilgraph von G heißt **Zusammenhangskomponente**.
Für eine Teilmenge $V' \subseteq V$ ist G\V' definiert als der Graph $(V \backslash V', \{\{u,v\} \in E \mid u,v \in V \backslash V'\})$.
Ein Graph G=(V,E) heißt **gerichteter Graph**, wenn $E \subseteq V \times V$ ist. Man unterscheidet hier den **Eingangsgrad** $d^+(v)=|\{(u,v) \mid (u,v) \in E\}|$ und den **Ausgangsgrad** $d^-(v)=|\{(v,u) \mid (v,u) \in E\}|$.
Die Begriffe von Weg, Zyklus, azyklisch usw. übertragen sich von ungerichteten auf gerichtete Graphen sinngemäß.

Beispiel: G=(V,E) mit V={a,b,c,d} und E={{a,c},{a,d},{b,d},{b},{c,d}} ungerichtet, gerichteter Graphen G=(V',E') mit V'=V und E'={(a,c),(d,a),(b,d),(b,b),(d,c)}.

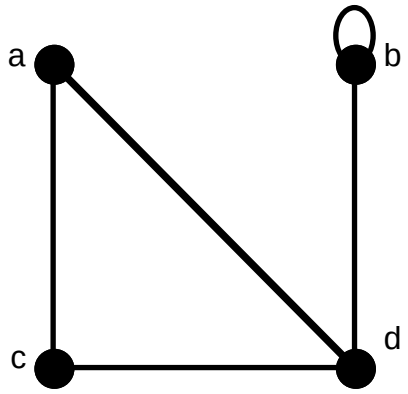


Abb.: Ungerichteter Graph

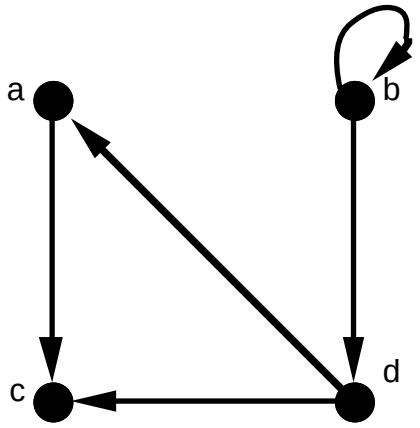


Abb.: Gerichteter Graph

Implementierung von Graphen.

- Fortlaufende Numerierung der Knoten beginnend bei 1
- boolesche Matrix für die Kantenbeziehungen, *Adjazenzmatrix* A=(a_{ij}) mit

$$a_{ij} = \begin{cases} \text{true}, & \text{falls } \{i,j\} \in E \text{ im ungerichteten Graphen bzw.} \\ & (i,j) \in E \text{ im gerichteten Graphen} \\ \text{false}, & \text{sonst.} \end{cases}$$

ungerichteter Graph ==> symmetrische Adjazenzmatrix, d.h. $a_{ij}=a_{ji}$ für alle i,j.

Beispiel: Adjazenzmatrix der obigen Graphen:

$$A = \begin{matrix} \text{false} & \text{false} & \text{true} & \text{true} \\ \text{false} & \text{true} & \text{false} & \text{true} \\ \text{true} & \text{false} & \text{false} & \text{true} \\ \text{true} & \text{true} & \text{true} & \text{false} \end{matrix} \qquad A' = \begin{matrix} \text{false} & \text{false} & \text{true} & \text{false} \\ \text{false} & \text{true} & \text{false} & \text{true} \\ \text{false} & \text{false} & \text{false} & \text{false} \\ \text{true} & \text{false} & \text{true} & \text{false} \end{matrix}$$

Nachteil: hoher Speicherplatzbedarf *unabhängig* von der Größe des Graphen: stets $|V|^2$ Speicherzellen.
Besser: verkettete Darstellungen, bei denen ein Speicherplatzbedarf entsteht, der nur linear mit der Größe des Graphen wächst, also ungefähr $c(|V|+|E|)$ für eine Konstante c.

Adjazenzlisten. Lineare Verkettung der Knoten mit jeweils Liste von Verweisen auf die adjazenten Knoten.

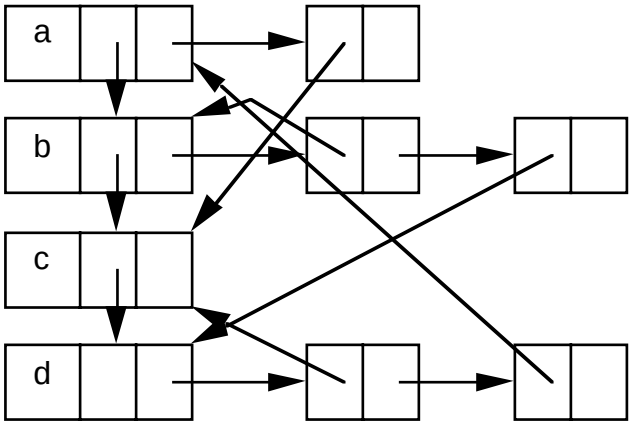


Abb.: Adjazenzlistendarstellung zum obigen Graphen

Durchlaufen von Graphen.
Zweck: systematisches Durchsuchen eines Graphen und ggf. Ausgabe aller Knotenmarkierungen.

Vorgabe: Startknoten v.
Bedingung: Graph zusammenhängend (sonst je Komponente neu starten)

Verfahren 1: **Tiefendurchlauf** (**depth first search**, Abk. **dfs**)
besucht ausgehend von einem Knoten zunächst einen beliebigen adjazenten Knoten, macht von dort rekursiv weiter und besucht zu diesem Knoten einen beliebigen adjazenten Knoten usw. Erst wenn von einem Knoten kein unbesuchter Nachbar mehr gefunden wird, geht der Algorithmus zurück und versucht von bereits besuchten Knoten unbesuchte Nachbarn zu finden.
Merke: Erst "Tiefe" dann "Breite".

Vergahren 2: **Breitendurchlauf** (**breadth first search**, Abk. **bfs**)
geht von einem Knoten, der gerade besucht wird, zunächst zu allen adjazenten Knoten, bevor deren adjazente Knoten besucht werden.
Merke: Erst " Breite " dann " Tiefe ".

Implementierung dfs:
G=(V,E) ein Graph mit |V|=n Knoten.
Hilfsarray für besuchte Knoten:

```
var besucht: array [V] of boolean.
```

Programm dfs:

```
var besucht: array [V] of boolean;
procedure dfs(v: knoten);
begin
    besucht[v]:=true; write(v);
    for all w∈V: {v,w}∈E do
        if not besucht[w] then dfs(w)
end.
```

Implementierung bfs:
Queue für die jeweils adjazenten Knoten eines besuchten Knotens.
Knoten werden in der Reihenfolge besucht, in der sie in der Queue abgelegt sind:

Programm bfs:

```
var besucht: array [V] of boolean;
    q: queue of V;
procedure bfs(v: knoten);
begin
    besucht[v]:=true;
    enter(v,q);
    while not is_empty(q) do
    begin
        v:=first(q); write(v);
        remove(q);
        for all w∈V: {v,w}∈E do
            if not besucht[w] then
            begin
                besucht[w]:=true;
                enter(w,q)
            end
    end
end.
```

Bei geeigneter Implementierung; jeweils proportional zur Größe von G (=|V|+|E|) viele Schritte und ebenso viel Speicher (für Stack bzw. Queue).

Beispiel: Für u.g. Graphen mögliche Durchläufe:
dfs: 1 2 9 10 6 5 3 8 4 7
bfs: 1 2 4 6 9 3 7 8 5 10
In beiden Fällen hängt die genaue Besuchsreihenfolge von der Implementierung der Anweisung for all ab.

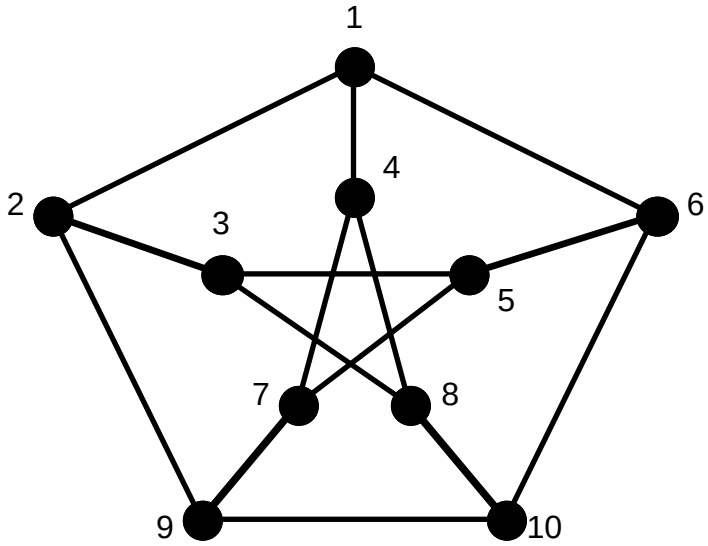


Abb.: Graph

2.10 Schlußbeispiel: Topologisches Sortieren

Form des Sortierens einer Menge $M=\{a_1,...,a_n\}$, wenn keine vollständige sondern nur eine partielle Ordnung vorliegt,

Ziel: Anordnung der Elemente der partiell geordnete Menge, daß für alle $i,j \in \{1,...,n\}$ mit $i \neq j$ aus $a_i < a_j$ immer $i<j$ folgt.

Beispiel:

$M=\{a_1,...,a_9\}$ mit
$a_1<a_3$, $a_3<a_7$, $a_7<a_4$, $a_7<a_5$, $a_4<a_6$, $a_9<a_2$, $a_9<a_5$,
$a_2<a_8$, $a_5<a_8$, $a_8<a_6$.

Topologische Sortierung z.B.:
$a_1,a_3,a_7,a_4,a_9,a_2,a_5,a_8,a_6$.

Andere Sortierung:
$a_9,a_1,a_2,a_3,a_7,a_5,a_8,a_4,a_6$.

Anwendungen:
1) *Projektplanung*. Hausbau.
2) *Schreiben eine Lehrbuchs*. ordne die Begriffsdefinitionen.
3) *Studienplan*. ordne Vorlesungen nach Voraussetzungen.

Modellierung durch gerichtete Graphen:
Elemente der Menge M --> Knoten
v<w --> zeichne Kante von v nach w.
Formal:

G=(V,E) mit V=M, E={(v,w) | v<w}.

G immer azyklisch (warum?).

Beispiel: partielle Ordnung aus obigem Beispiel als Graph und topologische Sortierung.

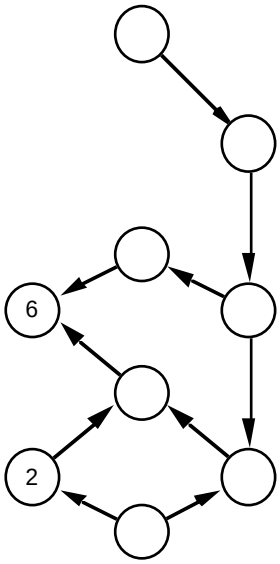


Abb.: Menge M mit partieller Ordnung als Graph

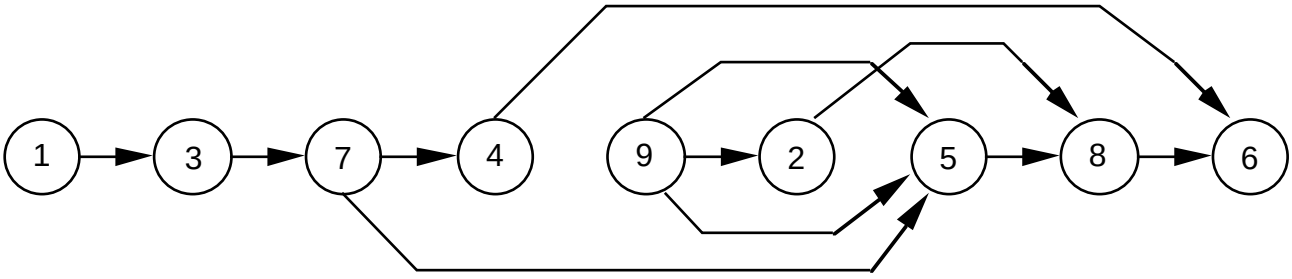


Abb.: topologische Sortierung von M als Graph

Lösungsidee:

- bestimme Knoten ohne Vorgänger, also mit Eingangsgrad 0. Knoten existiert immer, weil der Ausgangsgraph azyklisch ist.
- Entferne Knoten (und alle ausgehenden Kanten) und wende Verfahren auf den restlichen Graphen erneut an.

Algorithmus:

```
procedure topsort (G=(V,E): graph);
begin
    if V≠∅ then
    begin
        Wähle v∈ V mit d+(v)=0;
        writeln(v);
        topsort(G\{v})
    end
end.
```

Implementierung.

Aufgaben:

- Wähle geeignete Datenstruktur
- implementiere Schritt "Wähle $v \in V$ mit $d^+(v)=0$" geschickt:
- Idee:
    - bestimme einmalig zu Beginn alle Eingangsgrade
    - aktualisiere bei jedem Löschen eines Knotens die Eingangsgrade der adjazenten Knoten; jeweils vorliegende Knoten mit Eingangsgrad 0 werden in einer Queue abgelegt, nacheinander verarbeitet und aus der Queue entfernt.

Algorithmus:

```
var d+: array [V] of integer;
    q: queue of V;
begin
    read(G=(V,E));
    empty(q);
    for all v∈ V do d+(v):=0;
    for all v∈ V do
    begin
        for all w∈ V: (w,v)∈ E do d+(v):=d+(v)+1;
        if d+(v)=0 then enter(v,q)
    end;
    while not is_empty(q) do
    begin
        v:=first(q); remove(q);
        writeln(v,N);
        for all w∈ V: (v,w)∈ E do
        begin
            d+(w):=d+(w)-1;
            if d+(w)=0 then enter(w,q)
        end
    end
end.
```

Obiger Algorithmus erscheint effizienter als der erste.

Nächste Aufgabe:

- Präzisiere den Begriff Effizienz
- Entwickle Effizienzmaße und Meßmethoden