Abschnitt 4

Zeiger, Listen, Halde:

Objekte in der Halde,
Zeigertypen,
Listen und der Datentyp Sequenz,
Operationen auf Listen,
Graphen und ihre Darstellung,
Erstellen einer Adjazenzliste.

Keller und Halde

Variablen müssen irgendwo abgelegt werden. Wir haben bisher zwei Möglichkeiten kennen gelernt:

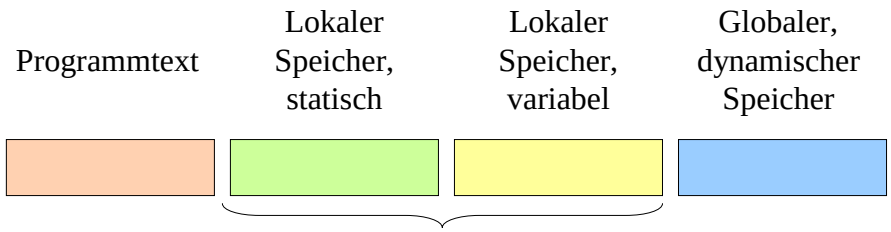
1. Deklaration am Anfang einer Programmeinheit (insbesondere zu Beginn eines Blockes) mit genauen Angaben über Typ und Größe zur Übersetzungszeit (statische Variable, z.B.: K: integer; A: array (4..17) of Character).
2. Auswertung einer Deklaration mit Festlegung von Größe und Typ zur Laufzeit (z.B. dynamische Felder: array (I..J) of ...).

Diese Variablen unterliegen der Blockstruktur und ihre Lebensdauer endet mit dem Verlassen des Blocks, in dem sie deklariert wurden.

Daneben möchte man oft Objekte haben, deren Länge flexibel bleibt oder die in inneren Blöcken (oder in Unterprogrammen) erzeugt werden und deren Lebensdauer erst dann endet, wenn sie nirgendwo im Programm mehr benötigt werden.

Solche Objekte werden in einem anderen Speicherbereich, der "Halde" (engl. Heap, in Ada "storage pool") abgelegt. Sie werden explizit mittels new erzeugt. Sie leben solange, bis sie entweder explizit (mittels einer "free"- oder "deallocate"-Anweisung) oder bei einer Speicherbereinigung oder mit dem Ende des Programms gelöscht werden.

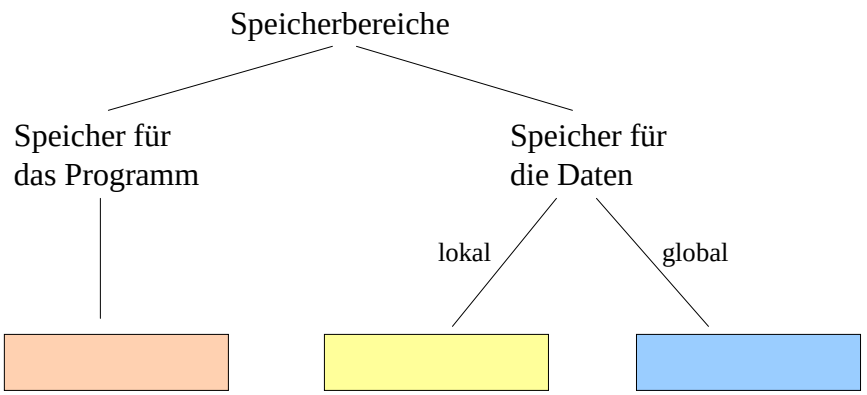
Solche Objekte brauchen keinen eigenen Namen im Programm zu haben. Sie sind über Zeiger (Verweise, Referenzen) erreichbar. Skizze hierzu:



Programmspeicher, wird beim Ablauf des Programms nicht verändert

Keller: Pulsiert mit dem Wechsel zwischen Blöcken und Prozeduren

Halde: Wo Platz ist, werden referenzierte Daten abgelegt. Wird dauernd aktualisiert.

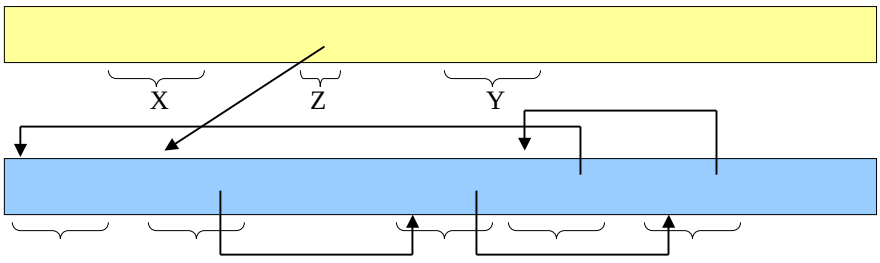


Programmspeicher, wird beim Ablauf des Programms nicht verändert

Keller: Pulsiert mit dem Wechsel zwischen Blöcken und Prozeduren

Halde: Wo Platz ist, werden referenzierte Daten abgelegt. Wird dauernd aktualisiert.

program ... X: integer; Z: ...; ...; declare Y: ...; Z := new TYP; ... end.



Wie pulsiert der Keller durch Blöcke und Prozeduren?
Wie werden die Daten in der Halde verwaltet?
Wie werden viele Programme verwaltet?

Der Datentyp Sequenz, Folgenbildung

Zu jeder Menge M kann man die Menge M* der endlichen Folgen (auch Menge der Wörter über M genannt) bilden:

$M^* = \{a_1\, a_2 \ldots a_n \mid n \geq 0 \text{ und } a_i \in M \text{ für } i = 1, 2, \ldots, n\}$.

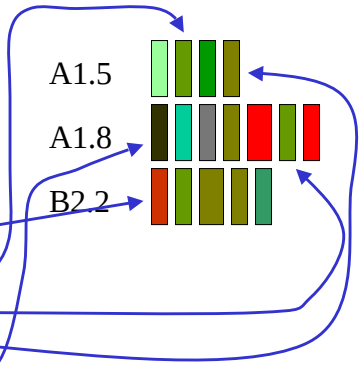
M möge zum Datentyp T gehören. Wir definieren den Datentyp seq of T durch folgende Festlegungen:

Zugrunde liegende Menge: M*

Nullstellige Operationen: Alle Elemente von M*. Man schreibt diese Wörter auf, indem man sie entweder in Anführungsstriche setzt und die Elemente von M durch Zwischenräume trennt oder indem man sie als Vektoren $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$ notiert. Das leere Wort erhält hierbei die Darstellung ε oder ( ). In der Programmierung schreibt man oft null oder nil für das leere Wort.

Beispiel: Bibliothek

Standort der Bücher:



Karteikarten oder
Computereinträge

...
Kaiser, Albert, "Über die ...", B2.2
Kaiser, Karl, "Die helle ...", A1.5
Kaiser, Wolfgang, "Das ...", A1.8
Kaisers, Emma, "Über die ...", A1.5
Kaisser, Fritz, "Das alte ...", A1.8
...

Karteikarten sind vom Datentyp "Name" oder "Adresse".

Einstellige Operationen

Für empty, square, removefirst, removelast: $M^* \to M^*$ gilt:
empty$(u) = \varepsilon$ für alle $u \in M^*$,
square$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = (a_{i_1}, a_{i_2}, ..., a_{i_n}, a_{i_1}, a_{i_2}, ..., a_{i_n})$,
removefirst$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = (a_{i_2}, ..., a_{i_n})$,
removelast$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = (a_{i_1}, a_{i_2}, ..., a_{i_{n-1}})$.
Für das leere Wort sind removefirst und removelast undefiniert.

first, last: $M^* \to M$ sind definiert durch
first$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = a_{i_1}$, sofern $n > 0$; first$(())$ ist undefiniert,
last$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = a_{i_n}$, sofern $n > 0$; last$(())$ ist undefiniert.

Statt "first" schreibt man meist "head", statt removefirst "tail".

Einstellige Operationen (Fortsetzung)

in: $M \to M^*$ mit in$(b) = (b)$, für alle $b \in M$ (Einbettung).

length: $M^* \to \mathbb{N}_0$ ist definiert durch
length$((a_{i_1}, a_{i_2}, ..., a_{i_n})) = n$; speziell gilt length$(()) = 0$.
Für jedes $a \in M$ ist $\#_a : M^* \to \mathbb{N}_0$ die Anzahlfunktion für das Element a, d.h., $\#_a(u)$ = Anzahl, wie oft a in u vorkommt.
Üblicherweise definiert man $\#_a$ rekursiv:
$\#_a(()) = 0$ und für alle $u \in M^*$ und alle $b \in M$:
$\#_a(ub) = \#_a(u) + 1$, falls $a = b$ ist,
$\#_a(ub) = \#_a(u)$, falls $a \neq b$ ist,

isempty: $M^* \to \mathbb{B}$ ist definiert durch
isempty(u) = true genau dann, wenn u das leere Wort ist.

Zweistellige Operationen

conc: $M^* \times M^* \to M^*$ (Konkatenation) ist definiert durch
$\text{conc}((a_{i_1}, a_{i_2}, ..., a_{i_n}), (b_{j_1}, b_{j_2}, ..., b_{j_m})) = (a_{i_1}, ..., a_{i_n}, b_{j_1}, ..., b_{j_m})$
append: $M^* \times M \to M^*$ (Anhängen eines Elements) ist definiert durch $\text{append}((a_{i_1}, a_{i_2}, ..., a_{i_n}), b) = (a_{i_1}, ..., a_{i_n}, b)$.

Die Vergleichsoperationen $=, \neq$: $M^* \times M^* \to \mathbb{B}$ sind wie üblich definiert. Falls M eine geordnete Menge ist, so kann man auch die anderen Vergleichsoperationen $<, \leq, >$ und $\geq$ verwenden.
(Man achte aber genau darauf, wie die Ordnung von M auf M* fortgesetzt wurde! Beispiel: lexikografisch oder längenlexikografisch.)

Beziehungen, Gesetzmäßigkeiten

Beispiele für Gesetzmäßigkeiten sind (stets für alle $u \in M^*$):
$\text{removelast}(\text{append}(u, b)) = u$ für alle $b \in M$.
$\text{conc}(u, u) = \text{square}(u)$.

$$\text{length}(u) = \sum_{a \in M} \#_a(u).$$

not $\text{isempty}(\text{append}(u,b))$ für alle $b \in M$.

$\text{length}(\text{conc}(u, v)) = \text{length}(u) + \text{length}(v)$ für alle $v \in M^*$.

$\text{first}(\text{in}(b)) = \text{last}(\text{in}(b)) = b$ für alle $b \in M$.

Die Datenstruktur "seq of T" in der Sprache Ada

Folgen werden in Ada durch "Listen" dargestellt, die mittels Zeigern (also mit access - Datentypen) realisiert werden, siehe unten.

Für die Operationen müssen geeignete Prozeduren und Funktionen geschrieben werden, sofern nicht eine vordefinierte Klasse für die "Listenverarbeitung" existiert.

Ein Spezialfall sind Folgen über dem Latin-1-Alphabet. In Ada gibt es für Zeichenketten den vordefinierten Datentyp "string", allerdings ist dieser nicht dynamisch, sondern die Länge des string muss bei der Deklaration bekannt sein. Kann man dies nicht angeben, so sollte man eine Liste verwenden.

Einschub: Deklarationen in Ada (Syntax 3.1)

```
basic_declaration ::=
    type_declaration                | subtype_declaration    |
    object_declaration              | number_declaration     |
    subprogram_declaration          |
    abstract_subprogram_declaration |
    package_declaration             | renaming_declaration   |
    exception_declaration           | generic_declaration    |
    generic_instantiation

type_declaration ::=  full_type_declaration    |
    incomplete_type_declaration               |
    private_type_declaration                  |
    private_extension_declaration
```

Einschub: Typdeklarationen in Ada (Syntax 3.1)

full_type_declaration ::=
'type' *defining*_identifier [known_discriminant_part] 'is'
type_definition ";" |
task_type_declaration | protected_type_declaration

type_definition ::=
enumeration_type_definition | integer_type_definition |
real_type_definition | array_type_definition |
record_type_definition | access_type_definition |
derived_type_definition

Hiervon fehlt bei uns nur noch die Definition für access_type:
access_type_definition ::= access_to_object_definition |
access_to_subprogram_definition

Zeiger in Ada95:

access_to_object_definition ::=
'access' [general_access_modifier] subtype_indication

general_access_modifier ::= 'all' | 'constant'

Nullstellige Operation: null (zugleich Initialisierung jedes Zeigers in Ada)

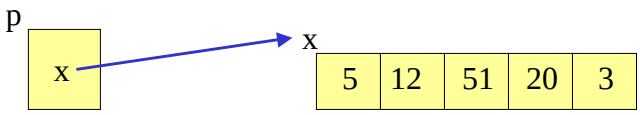
Einstellige Operation: Zugriff auf das Objekt, auf das der Zeiger verweist (Dereferenzierung, Schlüsselwort all. Ada dereferenziert automatisch, so dass "all" oft weggelassen werden kann.).

Zweistellige Operationen:
Gleichheit oder Ungleichheit von referenzierten Objekten oder von Zeigern: p = q bedeutet "Verweisen p und q auf dasselbe Objekt?", während p.all = q.all angibt, ob p und q auf zwei Objekte zeigen, die komponentenweise gleich sind.

Beispiel:

type vekt is array (1..5) of Integer;
type A_Int_Feld is access vekt; ...
p: A_Int_Feld; ...
p := new vekt'(5,12,51,20,3);

Man kann den Namen (dieser beinhaltet vor allem die Adresse des Feldes im Speicher) eines Feldes, auf das p verweist, als Wert der Variablen p auffassen. Skizze:



(im Keller) (in der Halde)

Relationen, Graphen, Referenzen

Statt die Objekte direkt zu manipulieren, kann man auch mit Verweisen auf sie (oder mit ihren Namen) arbeiten.

Da der Name eines Objekts im Speicher eines Rechners durch die Position ("Adresse"), ab der das Objekt im Speicher steht, dargestellt wird, spricht man auch von der Adresse (anstelle des Namens) eines Objekts.

Mathematisch kann man dies als eine Relation auffassen, also eine Beziehung zwischen der Variablen, die den Namen enthält, und der durch den Namen bezeichneten Variablen.

Relationen stellt man meist in Form von (gerichteten) Graphen dar.

Listen

Um für eine Menge M die Menge M* der endlichen Folgen (auch Menge der Wörter oder freies Monoid über M genannt) $M^* = \{a_1 a_2 \ldots a_n \mid n \geq 0 \text{ und } a_i \in M \text{ für } i = 1, 2, \ldots, n\}$. verwenden zu können, kann man ein Feld

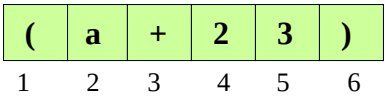
type M_Folgen is array (<Indexdatentyp>) of <Datentyp>

einführen, allerdings ist man dann begrenzt auf den (in der Praxis beschränkt großen) Indexbereich.

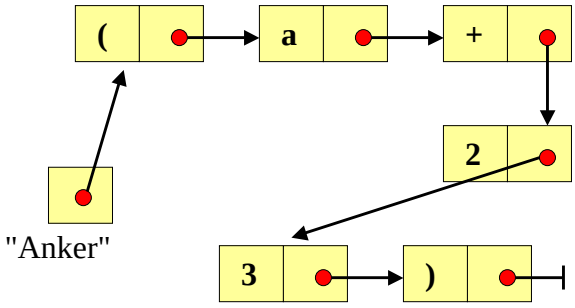
Alternativ kann man eine Folge so definieren, dass jedes Element durch einen Namen bezeichnet wird und man zu jedem Element den Namen des nachfolgenden Elements notiert. Der Name selbst braucht hierbei nicht bekannt gegeben zu werden, es genügt ein Zeiger.

Array-Darstellung A: array (1..6) of character

A

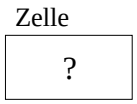


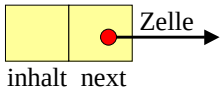
Listendarstellung:



Darstellung in Ada mittels "access":

```
type Zelle;

type Ref_Zelle is access Zelle;

type Zelle is record
                inhalt: character;
                next: Ref_Zelle;
end record;
```





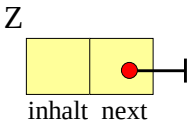
Prinzip: In jeder Definition müssen alle Bestandteile bekannt sein. Daher muss eine "Vor-information" type Zelle gegeben werden, die auf die kommende Präzisierung hinweist.

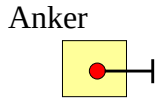

```
type Zelle;
type Ref_Zelle is access Zelle;
type Zelle is  record
          inhalt: character;
          next: Ref_Zelle;
end record;
```

Nicht ganz korrekt, es fehlt aliased bei Z

```
Z: Zelle;   Anker: Ref_Zelle;
```

Hierdurch werden zwei Variablen angelegt (in Ada werden Zeigervariablen stets automatisch mit null initialisiert:





Man kann durch Z.inhalt := 'a';
die Komponente inhalt von Z mit dem Wert 'a' füllen.

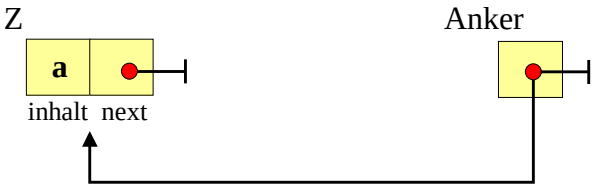
Der Variablen "Anker" kann man den Verweis (die Referenz) auf Z zuordnen:

Anker := Z;

Dies bewirkt folgendes:

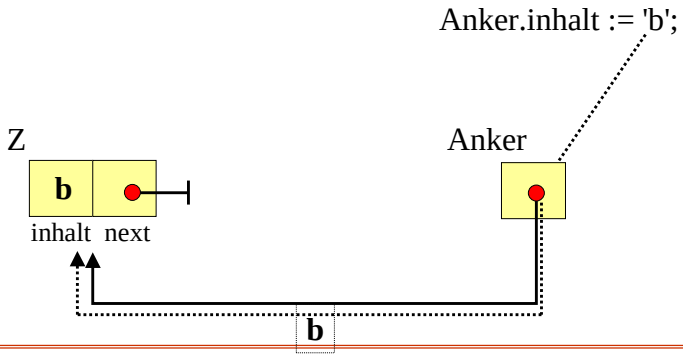
Z.inhalt := 'a';

Anker := Z;



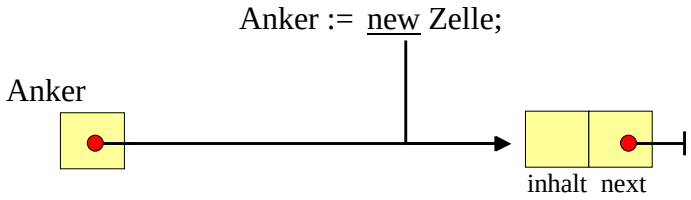
Man kann durch Anker.inhalt := 'b';
die Komponente inhalt der Variablen, auf die Anker verweist, mit dem Wert 'b' füllen.

Veranschaulichung:



Will man eine Variable anlegen, deren Name nicht interessiert und auf die nur verwiesen werden soll, so verwendet man das Schlüsselwort new mit Angabe des zu erzeugenden Datentyps (new ist ein "Generator für Speicherplatz", engl. "allocator").

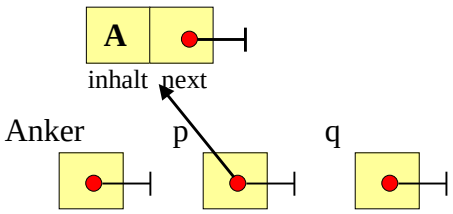
Anker: Ref_Zelle; ... Anker := new Zelle;
Veranschaulichung:



Nun können wir eine beliebig lange Kette von Verweisen bilden:

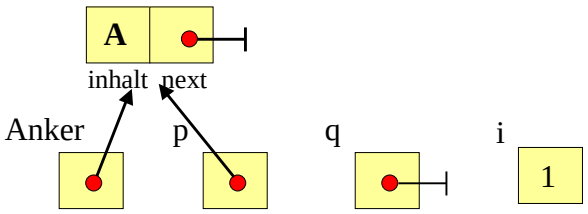

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



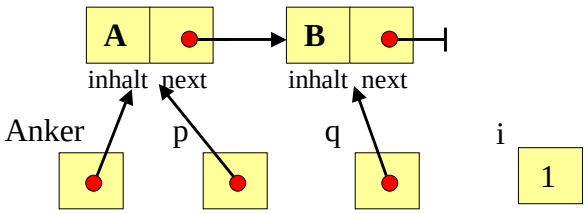
```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



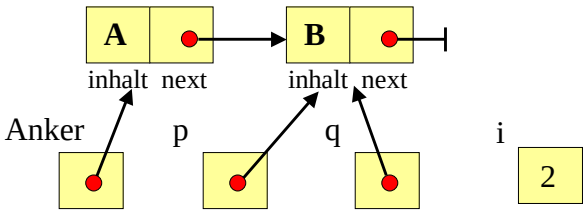

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



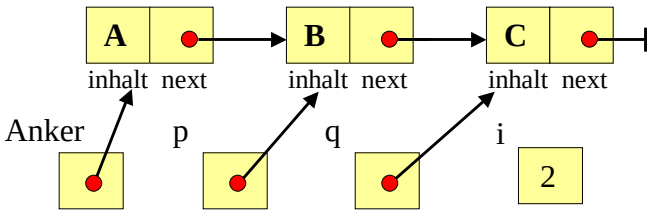
```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



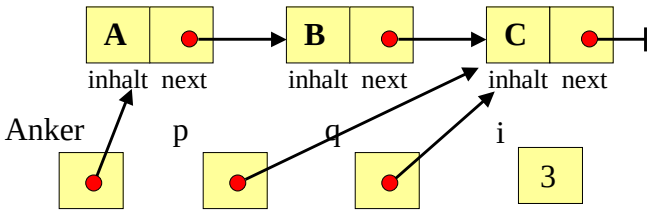

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



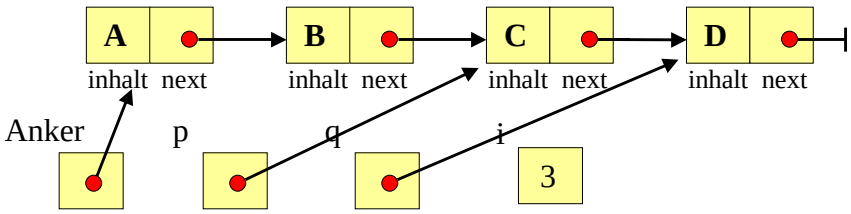
```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



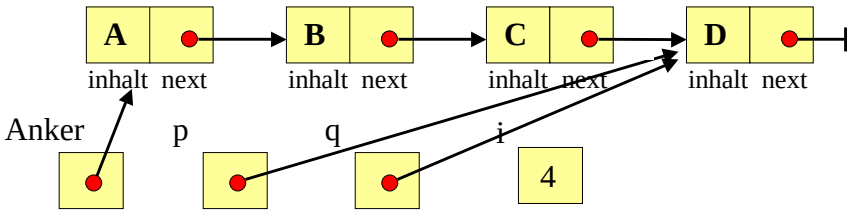

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



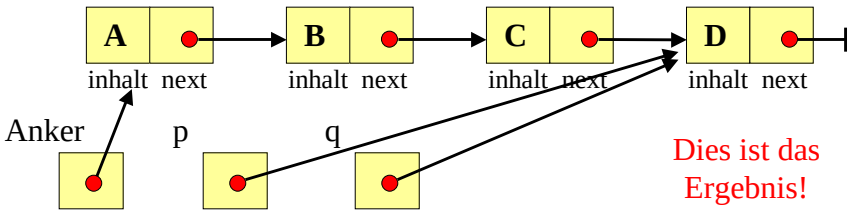
```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:



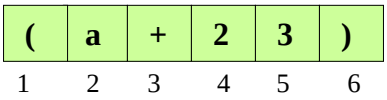

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop;  ...
```

Veranschaulichung:

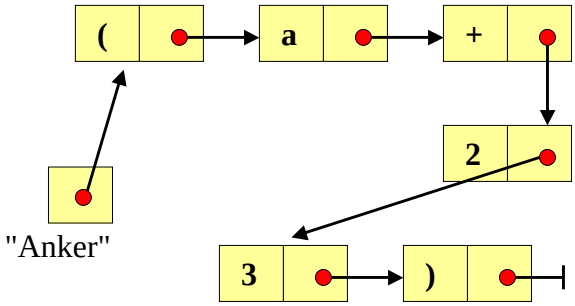


Zurück zu unserem Beispiel: Wir wollen anstelle des Feldes

A



eine Liste der folgenden Form aufbauen:




```
with Ada.Text_IO; use Ada.Text_IO;

procedure Char_Liste is
type Zelle;  type Ref_Zelle is access Zelle;
type Zelle is record inhalt: character; next: Ref_Zelle; end record;
Anker, p, q: Ref_Zelle;  Anzahl: natural;

begin  ...
     p := new Zelle; get(p.inhalt); Anker := p; Anzahl := 1;
     while not End_Of_File loop
          q:= new Zelle;
          get(q.inhalt); Anzahl := Anzahl+1;
          p.next := q;  q.next := null;  -- q.next:=null; ist hier überflüssig
          p := q;
     end loop;
...
end;
```

Übliche Operationen auf Listen

(1) Leeren einer Liste: Anker := null;

(2) Abfragen auf Leerheit einer Liste: Anker = null

(3) Hinzufügen eines Elements am Anfang:

```
p := new Zelle; p.inhalt := ... ;
p.next := Anker; Anker := p;
```

(4) Hinzufügen eines Elements am Ende:

```
p := Anker; q := null;
while p /= null loop q := p; p := p.next; end loop;
if q = null then   Anker := new Zelle;
                   Anker.inhalt := ... ;
else   q.next := new Zelle; q.next.inhalt := ... ; end if;
```

(p zeigt auf das aktuell betrachtete Element, q auf das davor.)

Übliche Operationen auf Listen (Fortsetzung)

(5) Suchen eines Elements mit dem Inhalt s:

```
p := Anker;
while p /= null and then p.inhalt /= s loop
                   p := p.next;  end loop;
if p = null then < s nicht in der Liste enthalten >
else < p verweist auf das erste Element mit Inhalt s > end if;
```

(6) Entfernen des ersten Elements mit dem Inhalt s:

```
p := Anker; q := Anker;
while p /= null and then p.inhalt /= s loop
        q := p;  p := p.next;  end loop;
if p /= null then q.next := p.next;  end if;
end if;
```

Bezeichnungen

Eine Liste, in der jedes Element nur auf seinen Nachfolger verweist, heißt einfach verkettete Liste.

Eine Liste, bei der das letzte Element nicht auf null, sondern auf das erste Element der Liste (zurück) verweist, heißt zyklische Liste.

Eine Liste, bei der jedes Element sowohl auf den Vorgänger in der Liste als auch auf den Nachfolger verweist, heißt doppelt verkettete Liste. Beispiel für deren Deklaration:

```
type DZelle;
type Ref_DZelle is access DZelle;
type DZelle is  record
                inhalt: character;
                vor, nach: Ref_DZelle;
end record;
```


Aufbau einer doppelt verketteten zyklischen Liste (den Anker setzen wir "konstant", so dass dieser Verweis nicht verändert werden kann - wohl aber die Inhalte der referierten DZelle):

```
Anker: constant Ref_DZelle := new DZelle;
p, q: Ref_DZelle;
begin if not End_Of_File then
  get(Anker.inhalt); p := Anker; Anzahl := 1;
  p.vor := p; p.nach := p;
  while not End_Of_File loop
        q:= new DZelle;  get(q.inhalt); Anzahl := Anzahl+1;
        q.nach := p.nach;  q.vor := p;
        p.nach := q; Anker.vor := q;  p := q;
  end loop;
end if;  ...
```

Hinweis: Man kann p.vor:=p und Anker.vor:=q streichen und dafür nach der Schleife Anker.vor := p; hinzufügen.

Eine Struktur, bei der die Elemente nacheinander durch Verweise angeordnet sind (also eine Folge bilden), heißt lineare Liste.

Einfach und doppelt verkettete, nicht-zyklische oder zyklische Listen sind also lineare Listen.

Lineare Listen sind die "gängige" Datenstruktur, um die Menge der Folgen M* in einer Programmiersprache zu realisieren.

Spezielle Listen: Keller und Schlangen (stack und queue)

Definition: Eine lineare Liste heißt Keller oder Stapel (engl.: stack oder pushdown), wenn auf ihr genau die folgenden fünf Operationen zugelassen sind:

(1) "empty" = Leeren der Liste.

(2) "isempty" = Abfragen auf Leerheit der Liste.

(3) "top" = Kopieren des letzten Elements der Liste.

(4) "push" = Hinzufügen eines Elements am Ende der Liste.

(5) "pop" = Löschen des letzten Elements der Liste.

(Realisierung durch Ada-Programmstücke wie oben.)

Man sagt auch, ein Keller ist eine Liste, die nach dem LIFO-Prinzip arbeitet.

LIFO = Last in, first out.
Dies bedeutet: die Elemente, die als letzte in den Keller eingefügt wurden, müssen als erste wieder herausgenommen werden.

Ein Beispiel ist der Ablagekorb auf dem Schreibtisch: Die Akten werden in der umgekehrten Reihenfolge, in der sie in den Ablagekorb gelegt wurden, herausgeholt und bearbeitet.

Beispiel: Spiegeln eines Textes

Sei K eine lineare Liste, auf der nur die fünf genannten Operationen verwendet werden dürfen. B sei eine Variable vom Typ Character.

```
empty(K);
while not End_Of_File loop
      get(B);
      push(K,B);
end loop;
while not isempty(K) loop
      put(top(K));
      pop(K);
end loop;
```

Definition: Eine lineare Liste heißt Schlange (engl.: queue), wenn auf ihr genau die folgenden fünf Operationen zugelassen sind:

(1) "empty" = Leeren der Liste.

(2) "isempty" = Abfragen auf Leerheit der Liste.

(3) "first" = Kopieren des ersten Elements der Liste.

(4) "enter" = Hinzufügen eines Elements am Ende der Liste.

(5) "remove" = Löschen des ersten Elements der Liste.

Die Realisierung durch Ada-Programmstücke ist einfach.

Man sagt auch, eine Schlange ist eine Liste, die nach dem FIFO-Prinzip arbeitet.

FIFO = First in, first out.

Wie der Name schon besagt, setzt man Schlangen für alle Situationen ein, bei denen Elemente nacheinander aufgestellt werden und in einer Reihe warten müssen, bis sie bearbeitet werden.

Beispiele sind Simulationen von Warteschlangen, etwa vor einem Schalter, im Verkehr oder beim Lernen.

Beispiel: Systematische Aufzählung aller von einer Grammatik G erzeugten Wörter. (Wörter realisiert man als Verweis auf String; u und v seien solche Verweise.)

G = (V, Σ, P, S) sei eine allgemeine Grammatik. Sei Q eine lineare Liste für Wörter über V∪Σ, auf der nur die fünf genannten Operationen für Schlangen verwendet werden dürfen.

```
empty(Q); enter (Q,"S");   -- die Schlange erhält anfangs das Wort "S"
while not isempty(Q) loop
      u := first(Q);          -- u ist das Wort, das am Anfang von Q steht
      remove(Q);
      if u ∈ Σ* then put(u);      -- u gehört dann zur erzeugten Sprache
      else
      for  alle Wörter v, die man aus u mit Hilfe der Produktionen
           aus P in einem Schritt ableiten kann  loop
                enter(Q, v); end loop;
      end if;
end loop;
```


Definition: Eine lineare Liste heißt Doppelschlange (engl.: deque), wenn auf ihr genau die folgenden acht Operationen zugelassen sind:

(1) "empty" = Leeren der Liste.

(2) "isempty" = Abfragen auf Leerheit der Liste.

(3) "first" = Kopieren des ersten Elements der Liste.

(4) "last" = Kopieren des letzten Elements der Liste.

(5) "enter_front" = Hinzufügen eines Elements am Anfang.

(6) "removefirst" = Löschen des ersten Elements der Liste.

(7) "enter_back" = Hinzufügen eines Elements am Ende.

(8) "removelast" = Löschen des letzten Elements der Liste.

Geflechte

Wir müssen die Elemente nicht wie an einer Perlenschnur aufreihen (lineare Liste), sondern können Verweise auch auf beliebige Elemente des zugrundeliegenden Datentyps setzen.

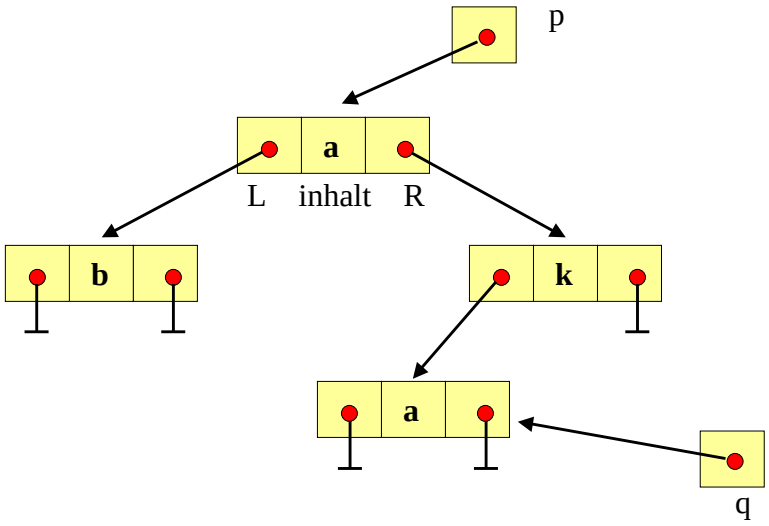
Dadurch entstehen beliebig vernetzte Gebilde. Diese werden Geflechte oder (meist) Graphen genannt.

Sie bestehen aus den Elementen des Datentyps ("Knoten" genannt), die durch Verweise ("Kanten", Ecken oder Pfeile genannt) miteinander verbunden sind.

Mathematisch gesehen sind dies Relationen über der Menge M des zugrunde liegenden Datentyps.

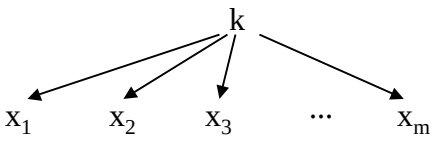
Wir betrachten ein Beispiel in Ada:


```
with Ada.Text_IO; use Ada.Text_IO;

procedure Char_Liste is
type BZelle;  type Ref_BZelle is access BZelle;
type BZelle is record inhalt: character; L,R: Ref_BZelle; end record;
p, q: Ref_BZelle;

begin  p := new BZelle;  p.inhalt:='a';
       q:= new BZelle;   p.L := q;      q.inhalt := 'b';
       q:= new BZelle;   p.R := q;      q.inhalt := 'k';
       q:= new BZelle;   p.R.L := q;    q.inhalt := 'a';
...
end;
```



Situation nach Ausführen des Programmstücks

Auf diese Weise lassen sich beliebig verflochtene Strukturen darstellen. Insbesondere können wir hiermit Bäume beschreiben: Ein Zeiger "Wurzel" verweist auf die Wurzel des Baumes und von dort wird weiter auf die Nachfolger verwiesen usw.

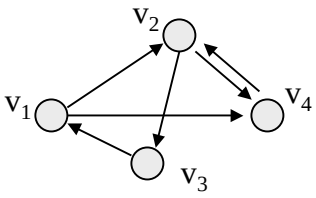



```
type Baum;  type Ref_Baum is access Baum;
type Baum is record
        inhalt: character;
        Nachf: array (1..m) of Ref_Baum;
end record;
```

Graphen

Graphen bestehen aus einer Knotenmenge V und einer Kantenmenge E (englisch: Knoten = **v**ertex oder node, Kante = **e**dge).

Definition: G = (V, E) heißt gerichteter Graph (oder Digraph)
⇔
1. V ist ein endliche nicht-leere Menge,
2. $E \subseteq V \times V$.
[Digraph kommt von "directed graph" = gerichteter Graph.]

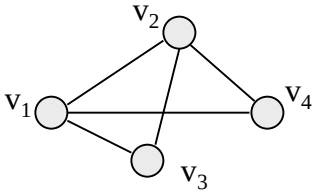


$V = \{v_1, v_2, v_3, v_4\}$

$E = \{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_1), (v_4, v_2)\}$

Im Falle, dass sowohl eine Kante von v nach v' als auch eine Kante von v' nach v führt, spracht man auch von einer ungerichteten Relation, bzw. von einem ungerichteten Graphen. In obigem Beispiel sind (v_2, v_4) und (v_4, v_2) solche Kanten.

Definition: G = (V, E) heißt ungerichteter Graph
⇔
1. V ist ein endliche nicht-leere Menge,
2. $E \subseteq \{ \{x, y\} \mid x, y \in V, x \neq y \} \cup \{ \{x\} \mid x \in V \}$.



$V = \{v_1, v_2, v_3, v_4\}$

$E = \{\{v_1, v_2\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_4, v_2\}\}$

Umschalten zwischen gerichteten und ungerichteten Graphen:

Es sei G = (V, E) ein ungerichteter Graph. Der gerichtete Graph $G_{ger} = (V, E_{ger})$ mit
$E_{ger} = \{(x,y), (y,x) \mid \{x, y\} \in E\} \cup \{(x,x) \mid \{x\} \in E\}$
heißt gerichtete Version des Graphen G.

Es sei G = (V, E) ein gerichteter Graph. Der ungerichtete Graph $G_{ung} = (V, E_{ung})$ mit
$E_{ung} = \{\{x,y\} \mid (x,y) \in E \text{ oder } (y,x) \in E\} \cup \{\{x\} \mid (x,x) \in E\}$
heißt ungerichtete Version des Graphen G.

Im ungerichteten Fall gehen wir also zu beiden Richtungen über, im gerichteten Fall ignorieren wir die Richtung.

Ein gerichteter Graph H heißt Orientierung oder Ausrichtung des ungerichteten Graphen G, wenn G die ungerichtete Version von H ist. (Zu G gibt es $2^{|E|}$ Orientierungen mit |E| Kanten.)

Definition: Teilgraph, induzierter Teilgraph

Es sei G = (V, E) ein ungerichteter bzw. gerichteter Graph. Ein ungerichteter bzw. gerichteter Graph G' = (V',E') heißt Teilgraph von G, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt.

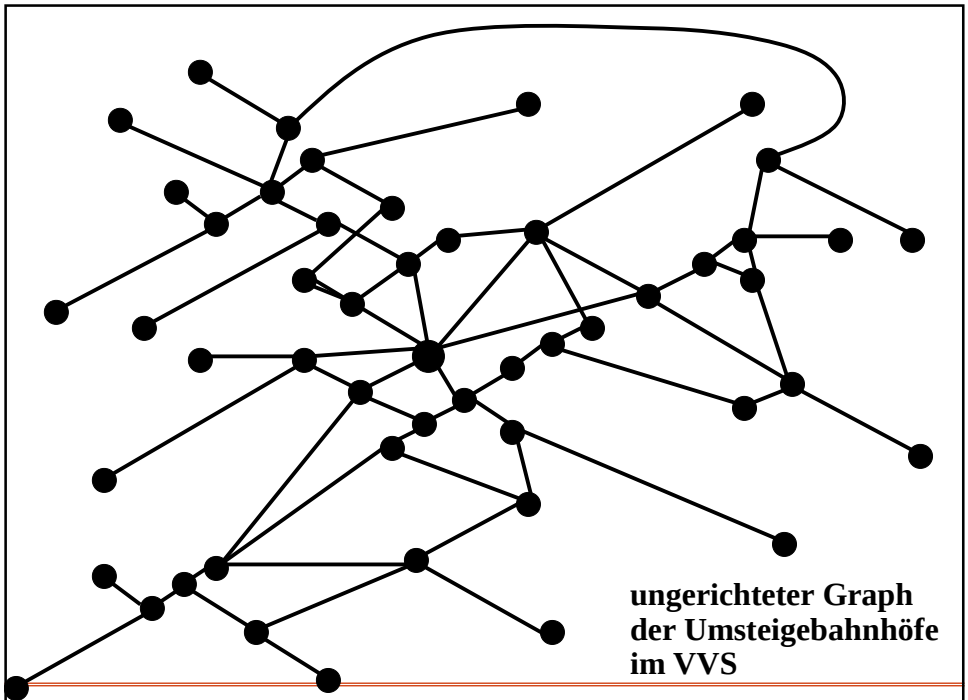
Es sei G = (V, E) ein ungerichteter bzw. gerichteter Graph und es sei $V' \subseteq V$. Der ungerichtete bzw. gerichteter Graph G' = (V',E') heißt der von V' induzierte Teilgraph von G, wenn im gerichteten Fall

$$E' = \{\{x,y\} \mid \{x,y\} \in E \text{ und } x,y \in V'\}$$

bzw. im gerichteten Fall

$$E' = \{(x,y) \mid (x,y) \in E \text{ und } x,y \in V'\}$$

gilt.



ungerichteter Graph der Umsteigebahnhöfe im VVS

Definition: (adjacent, neighbour, successor, predecessor)

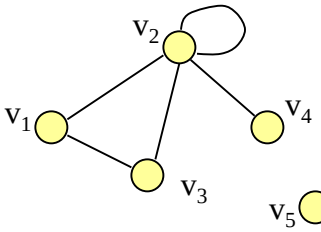
Jede Kante {x, y} bzw. (x,y) heißt inzident zu ihren Knoten x und y. Zwei Knoten x, y mit $\{x, y\} \in E$ (ungerichteter Fall) bzw. $(x,y) \in E$ oder $(y,x) \in E$ (gerichteter Fall) heißen adjazent.

Ungerichteter Fall: Die Menge $N(x) = \{ y \in V \mid \{x, y\} \in E \}$ heißt die Menge der Nachbarn (oder der Nachfolger) von x.

Gerichteter Fall: $N(x) = \{ y \in V \mid (x, y) \in E$ oder $(y, x) \in E\}$ heißt die Menge der Nachbarn von x.
$S(x) = \{ y \in V \mid (x, y) \in E\}$ heißt Menge der Nachfolger von x,
$P(x) = \{ y \in V \mid (y, x) \in E\}$ heißt Menge der Vorgänger von x.

Eine Kante {x} im ungerichteten Fall bzw. (x,x) im gerichteten Fall heißt Schlinge.

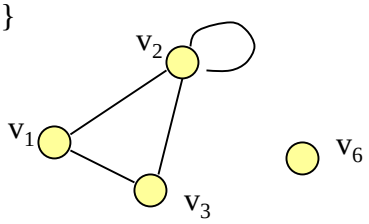
Beispiel: ungerichteter Fall



$V' = \{v_1, v_2, v_3,, v_4, v_5\ v_6\}$

$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_2\}, \{v_4, v_2\}, \{v_5, v_6\}\}$

Von $V' = \{v_1, v_2, v_3, v_6\}$ induzierter Teilgraph:



Definition: Grad d, Eingangs-, Ausgangsgrad, geordnet

Ungerichteter Fall: Für einen Knoten x heißt
$d(x) = |\{ y \in V \mid x \neq y, \{x, y\} \in E \}|$, falls $\{x\} \notin E$ bzw.
$d(x) = |\{ y \in V \mid x \neq y, \{x, y\} \in E \}| + 2$, falls $\{x\} \in E$
der (Knoten-) Grad von x ("degree"). Der maximale Knotengrad heißt Grad d(G) des Graphen G.

Gerichteter Fall: Für einen Knoten x heißt
$d^+(x) = |\{ y \in V \mid (x, y) \in E \}|$ der Ausgangsgrad und
$d^-(x) = |\{ y \in V \mid (y,x) \in E \}|$ der Eingangsgrad von x. Der Wert
$d(x) = d^+(x) + d^-(x)$ heißt auch der Grad von x.

Ein Graph heißt geordnet, wenn für jeden Knoten x im ungerichteten Fall die Menge der Nachbarn N(x) bzw. im gerichteten Fall die Menge der Nachfolger S(x) geordnet ist (und damit sind zugleich die zugehörigen Kanten geordnet).

Definitionen zur Darstellung von Graphen:

Graphen kann man durch ihre Adjazenzmatrix A, durch Adjazenzlisten oder durch Inzidenzlisten darstellen.

Es sei G = (V,E) mit V = $\{x_1, x_2, ..., x_n\}$ ein Graph.

Ungerichteter Fall: Die Adjazenzmatrix A = $(a_{i,j})$ ist definiert durch $a_{i,j} = 1$, falls $\{x_i, x_j\} \in E$, und $a_{i,j} = 0$ sonst (i, j = 1, ..., n).

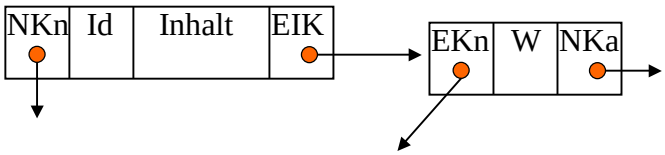
Gerichteter Fall: Die Adjazenzmatrix A = $(a_{i,j})$ ist definiert durch $a_{i,j} = 1$, falls $(x_i, x_j) \in E$, und $a_{i,j} = 0$ sonst (i, j = 1, ..., n).

Die *erweiterte Adjazenzmatrix* A' ist für $i \neq j$ gleich der Adjazenzmatrix, allerdings wird die Hauptdiagonale auf 1 gesetzt, d.h. $a'_{i,j} = a_{i,j}$ für $i \neq j$ und $a'_{ii} = 1$ (für i, j = 1, ..., n).

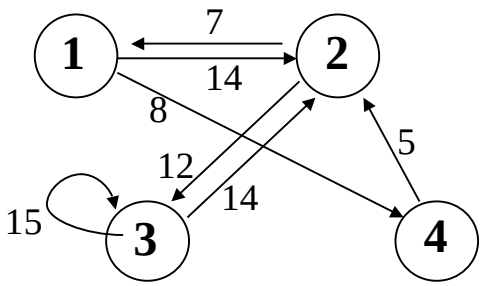
Datentyp für Graphen: Darstellung durch Adjazenzliste

Jeder Knoten erhält einen Identifikator Id (in der Regel ist dies eine natürliche Zahl) und einen Inhalt.
Die Knoten werden in Listen zusammengefasst und besitzen daher neben Id und Inhalt noch weitere Komponenten:
- Verweis auf den **N**ächsten **K**noten in der Liste "NKn",
- Verweis auf die **E**rste **I**nzidente **K**ante "EIK".

Die von einem Knoten ausgehende Kante muss enthalten: den "Endknoten der Kante" (EKn), ihren Wert W ("weight") und einen Verweis auf die nächste Kante "NKa".

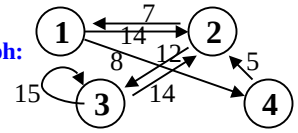


Beispiel: Gerichteter Graph mit Kantenwerten



Adjazenzliste hierzu

Zugehöriger Graph:



Anfang

NKn	Id	Inhalt	EIK	EKn	W	NKa	EKn	W	NKa
	1				14			8	
	4				5				
	2				12			7	
	3				14			15	

Die meisten Anwendungen notieren gewisse Informationen in den Graphen.

Sehr oft muss man sich zwei Zahlen merken, die Ergebnisse zuvor durchgeführten Durchläufen durch den Graphen sind, sowie einen Booleschen Wert "besucht", der angibt, ob dieser Knoten bereits zuvor erreicht ("besucht") worden ist.

Wir fügen diese Komponenten zu den Knoten hinzu und erhalten folgende Datentypen, formuliert in Ada:


```
type Knoten; type Kante;
type NextKnoten is access Knoten;
type NextKante is access Kante;

type Knoten is record
        Id: Knotenname;
        besucht: Boolean; zahl1, zahl2: integer;
        Inhalt: <weitere Komponenten>;
        NKn: NextKnoten;
        EIK: NextKante;
end record;

type Kante is record
        W: <Typ des Gewichts der Kanten>;
        EKn: NextKnoten;
        NKa: NextKante;
end record;
```

Beispiel:

Aufbau eines gerichteten Graphen in Adjazenzlistendarstellung aus seiner Adjazenzmatrix. Hierzu lesen wir zunächst die Anzahl der Knoten n und dann die n^2 Zeichen '0' oder '1' ein. Danach bauen wir die Knotenliste auf. Anschließend gehen wir die Adjazenzmatrix A zeilenweise durch und erzeugen hierbei die Kantenliste an jedem Knoten.

Die Deklarationen für Knoten, Kante, Nextknoten und Nextkante werden zu Beginn eingefügt mit:

Knotenname = Integer

Typ des Gewichts der Kante = Positive

Der Typ von Inhalt kann je nach Anwendung gewählt werden.


```
procedure Adj_List_Darst is
subtype Flag is Character range '0'..'1';
type Matrix is
   array (Integer range <>, Integer range <>) of Flag;
-- Deklarationen für Knoten, Kante usw. hier einfügen
n : Positive; Graph: Nextknoten;
begin  get(n);              -- Zahl der Knoten einlesen
   declare A: Matrix(1..n, 1..n);
   p: Nextknoten;           -- Hilfszeiger zum Aufbau
   begin                          -- Matrix einlesen
      for i in 1..n loop
         for j in 1..n loop get(A(i,j)); end loop;
      end loop;
```

```
Graph := new Knoten'(Id => 1); p:= Graph;
for i in 2..n loop          -- Aufbau der Knotenliste
   p.NKn := new Knoten'(Id => i); p := p.NKn;
end loop;
p := Graph;                 -- erster Knoten der Liste
for i in 1..n loop          -- für jeden Knoten
   for j in 1..n loop       -- Kanten durchgehen
   if A(i,j) = '1' then     -- falls Kante (i,j) existiert
      declare
      q: Nextknoten;        -- für Suche nach Knoten j
      begin
      while q /= null and then q.Id /= j loop
         q := q.NKn; end loop;
```



```
      if q = null then raise ...;     -- Fehlerbehandlung
      else                            -- Füge Kante ein
        p.EIK := new Kante'(EKn => q, NKa => p.EIK);
      end if;
      end;                  -- Ende innerer Block
   end if;                  -- Ende Test A(i,j)='1'
   end loop;                -- Ende Schleife mit j
   p := p.NKn;              -- nächsten Knoten nehmen
   end loop;                -- Ende Schleife mit i
 end;                       -- Ende mittlerer Block
   -- Hier kann man nun mit der Adjazenzliste arbeiten.
   -- Allerdings ist A hier nicht mehr bekannt.
...
end;                        -- Ende des Programms
```