Gliederung des Kapitels 1.3

1.3 Daten und ihre Strukturierung

1.3.3 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.

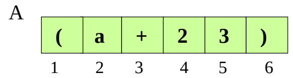
Relationen stellt man meist in Form von (gerichteten) Graphen dar.

1.3.3.1 Listen

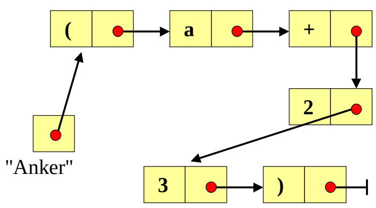
Um für eine Menge M die Menge M* der endlichen Folgen (auch Menge der Wörter oder freies Monoid über M genannt)
$M^* = \{a_1 a_2 \ldots a_n \mid n \geq 0 \text{ und } a_i \in M \text{ für } i = 1, 2, \ldots, n\}$.
verwenden zu können, kann man ein Feld
<u>type</u> M_Folgen <u>is</u> <u>array</u> (<Indexdatentyp>) <u>of</u> <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 "abstrakter Verweis" (Referenz, Zeiger).

Array-Darstellung A: <u>array</u> (1..6) <u>of</u> character

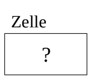


Listendarstellung:

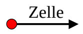


<u>Darstellung in Ada</u> mittels "access":

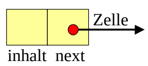
<u>type</u> Zelle;



<u>type</u> Ref_Zelle <u>is</u> <u>access</u> Zelle;



<u>type</u> Zelle <u>is</u> <u>record</u>
inhalt: character;
next: Ref_Zelle;
<u>end</u> <u>record</u>;



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

<u>type</u> Zelle;
<u>type</u> Ref_Zelle <u>is</u> <u>access</u> Zelle;
<u>type</u> Zelle <u>is</u> <u>record</u>
inhalt: character;
next: Ref_Zelle;
<u>end</u> <u>record</u>;

Z: Zelle; Anker: Ref_Zelle;

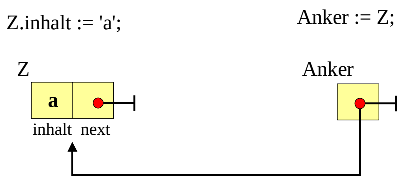
Hierdurch werden zwei Variablen angelegt (in Ada werden Zeigervariablen stets mit <u>null</u> (= <u>nil</u>) initialisiert:

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:

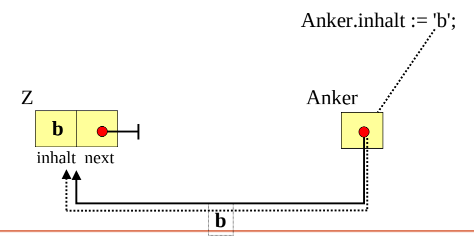
Anker := Z;

Dies bewirkt folgendes:



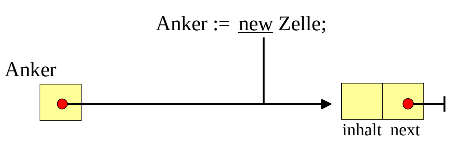
Man kann durch Anker.inhalt := 'b';
die Komponente inhalt der Variablen, auf die Anker verweist, mit dem Wert 'b' füllen.

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").

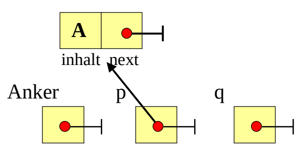
Anker: Ref_Zelle; ... Anker := new Zelle;
Veranschaulichung:



Nun können wir eine beliebig lange Kette von Verweisen bilden:

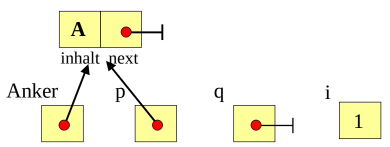

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop; ...
```

Veranschaulichung:



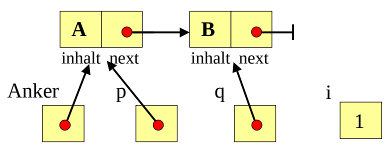

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop; ...
```

Veranschaulichung:



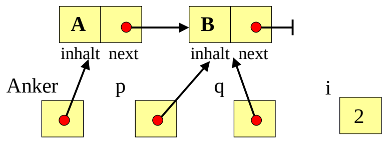

```
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
        q:= new Zelle;
        q.inhalt := character'Val (65+i);
        p.next := q; p:=q;
end loop; ...
```

Veranschaulichung:



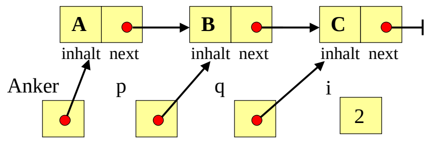
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
    q:= new Zelle;
    q.inhalt := character'Val (65+i);
    p.next := q; p:=q;
end loop; ...

Veranschaulichung:



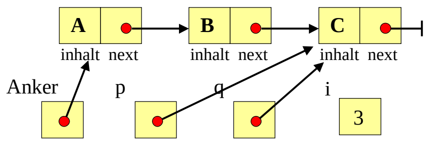
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
    q:= new Zelle;
    q.inhalt := character'Val (65+i);
    p.next := q; p:=q;
end loop; ...

Veranschaulichung:



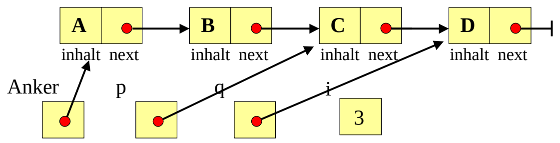
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
    q:= new Zelle;
    q.inhalt := character'Val (65+i);
    p.next := q; p:=q;
end loop; ...

Veranschaulichung:



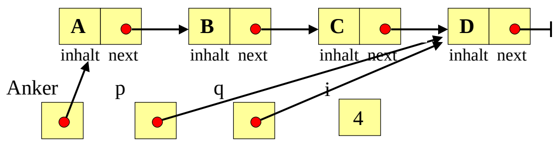
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
    q:= new Zelle;
    q.inhalt := character'Val (65+i);
    p.next := q; p:=q;
end loop; ...

Veranschaulichung:



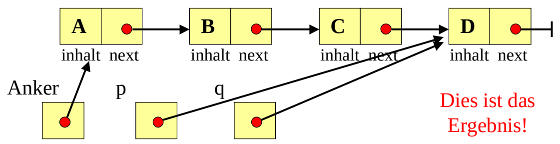
Anker, p, q: Ref_Zelle; ...
p := new Zelle; p.inhalt := 'A'; p.next := null; Anker := p;
for i in 1..3 loop
    q:= new Zelle;
    q.inhalt := character'Val (65+i);
    p.next := q; p:=q;
end loop; ...

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:



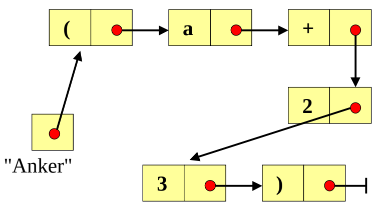
Dies ist das Ergebnis!

Zurück zu unserem Beispiel: Wir wollen anstelle des Feldes

A

(	a	+	2	3	)
1	2	3	4	5	6

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

```
Anker, p, q: Ref_DZelle;
p := new DZelle; get(p.inhalt); Anker := p; 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;
```

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. Diese Verweise nennt man meist "Zeiger" oder "Pointer" oder Referenzen.

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.

1.3.3.2 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.

Die Realisierung durch Ada-Programmstücke ist einfach, analog zu 1.3.3.1.

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.

G = (V, Σ, P, S) sei eine allgemeine Grammatik (1.1.5.14). 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.

1.3.3.3 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.

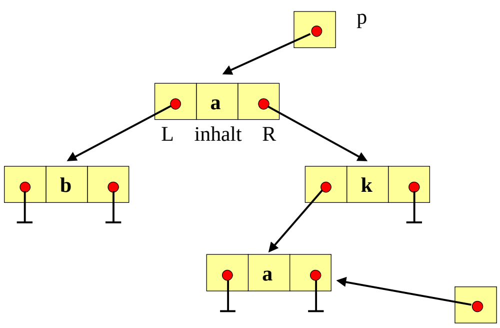
Mathematisch gesehen sind dies Relationen über der Menge M des zugrunde liegenden Datentyps, siehe 1.3.3.4.

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;
```

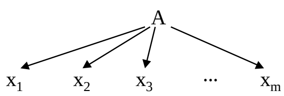



Situation nach Ausführen des Programmstücks

Auf diese Weise lassen sich beliebig verflochtene Strukturen darstellen und manipulieren.

Insbesondere können wir hiermit Bäume (1.1.5.7) beschreiben: Ein Zeiger "Wurzel" verweist auf die Wurzel des Baumes und von dort wird weiter auf die Nachfolger verwiesen usw.

Beispielsweise würde ein Produktion gemäß 1.1.5.8



dargestellt werden durch den Datentyp:

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

Auf Beispiele gehen wir in den Übungen ein. Dort werden wir Bäume aufbauen und zum Sortieren von beliebigen Elementen verwenden.

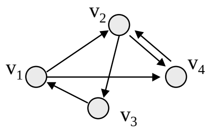
Wir wollen abschließend die dahinter liegenden Strukturen, die sog. Graphen betrachten.

1.3.3.4 Graphen

Also: 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) $\Leftrightarrow$
1. V ist ein endliche nicht-leere Menge,
2. $E \subseteq V \times V$.

[Digraph kommt von "directed graph" = gerichteter Graph.]

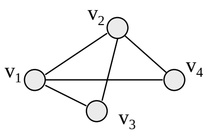


$V = \{v_1, v_2, v_3, v_4\}$

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

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 $\Leftrightarrow$
1. V ist ein endliche nicht-leere Menge,
2. $E \subseteq \{ \{x,y\} \mid x,y \in V, x \neq y \} \cup \{ \{x\} \mid x \in V \}$.



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

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

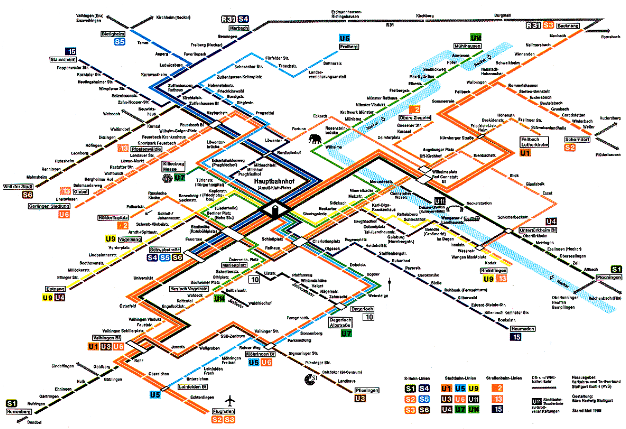
Umschalten zwischen gerichteten und ungerichteten Graphen:

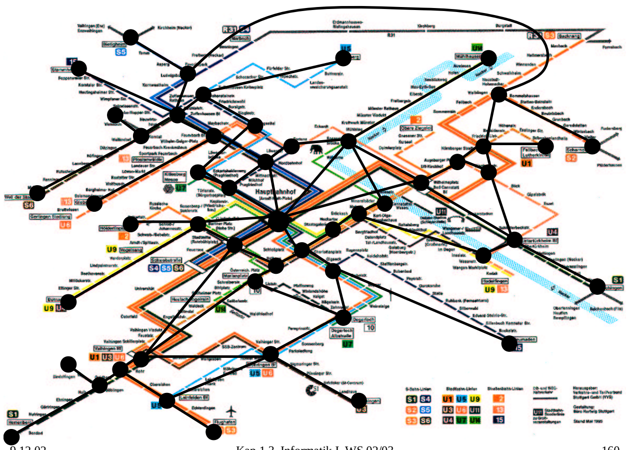
Es sei G = (V, E) ein ungerichteter Graph. Der gerichtete Graph $G_{ger} = (V, E_{ger})$ mit
$E_{ger} = \{(x,y), (y,x) \mid \{x,y\} \in E\} \cup \{(x,x) \mid \{x\} \in E\}$
heißt gerichtete Version des Graphen G.

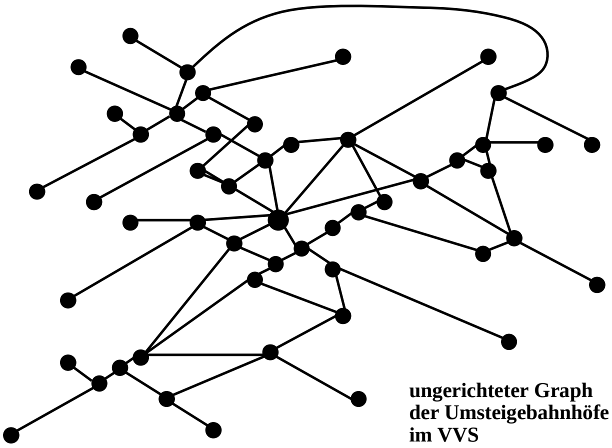
Es sei G = (V, E) ein gerichteter Graph. Der ungerichtete Graph $G_{ung} = (V, E_{ung})$ mit
$E_{ung} = \{\{x,y\} \mid (x,y) \in E \text{ oder } (y,x) \in E\} \cup \{\{x\} \mid (x,x) \in E\}$
heißt ungerichtete Version des Graphen G.

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

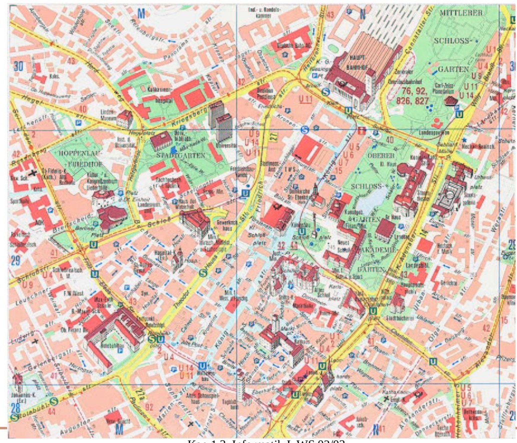
Ein gerichteter Graph H heißt Orientierung oder Ausrichtung des ungerichteten Graphen G, wenn G die ungerichtete Version von H ist. (Zu G gibt es $2^{|E|}$ Orientierungen mit |E| Kanten.)

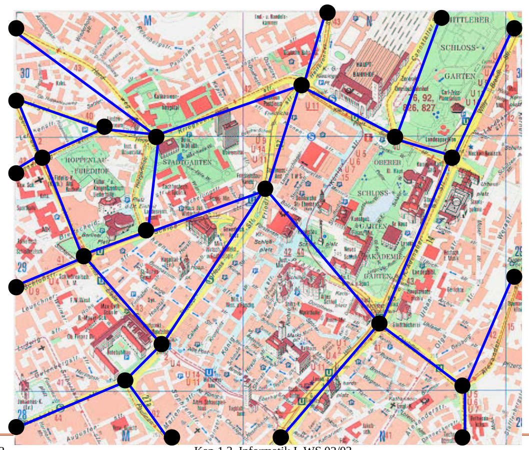


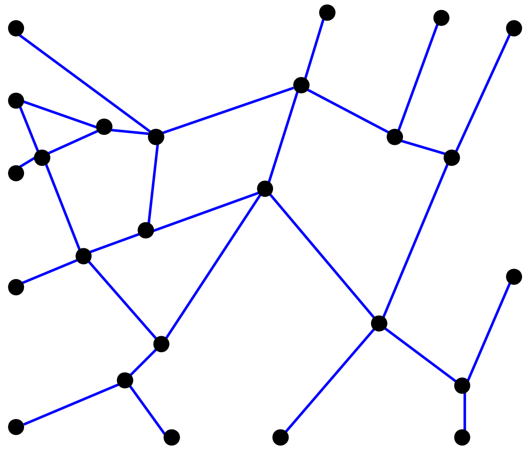


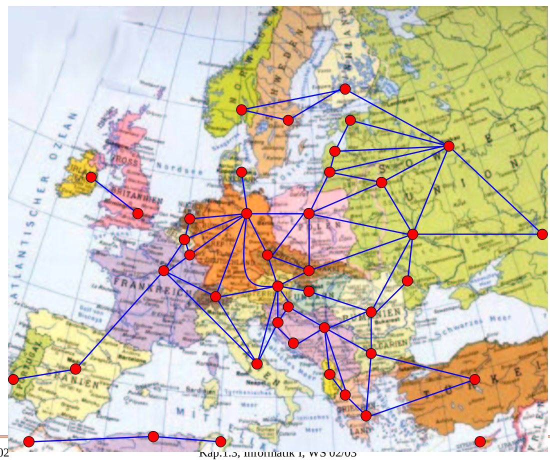


ungerichteter Graph der Umsteigebahnhöfe im VVS

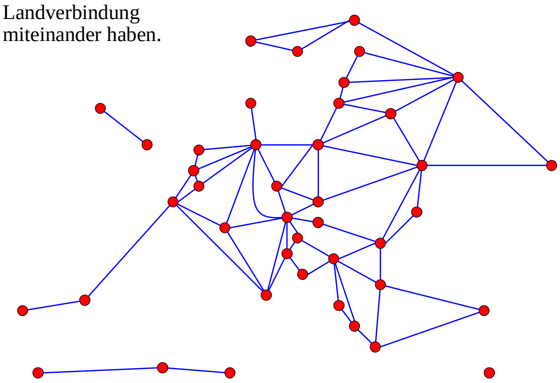








Ungerichteter Graph der Länder, die eine Landverbindung miteinander haben.



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. gerichtete 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.

Definition: (adjacent, neighbour, successor, predecessor)

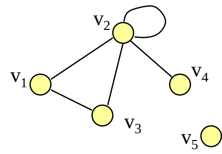
Jede Kante {x, y} bzw. (x,y) heißt inzident zu ihren Knoten x und y. Zwei Knoten x, y mit $\{x,y\} \in E$ (ungerichteter Fall) bzw. $(x,y) \in E$ oder $(y,x) \in E$ (gerichteter Fall) heißen adjazent.

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: $N(x) = \{ y \in V \mid (x,y) \in E \text{ oder } (y,x) \in E \}$ 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.

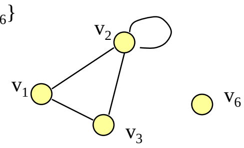
Beispiel: ungerichteter Fall



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

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

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).

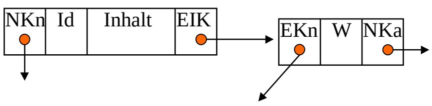
Gerichteter Fall: Die Adjazenzmatrix $A = (a_{i,j})$ ist definiert durch $a_{i,j} = 1$, falls $(x_i, x_j) \in E$, und $a_{i,j} = 0$ sonst (i, j = 1, ..., n).

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).

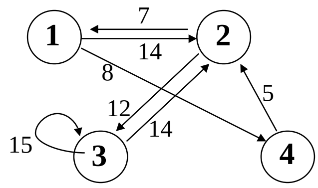
Datentyp für Graphen: Darstellung durch Adjazenzliste

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".

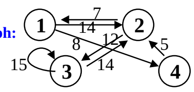


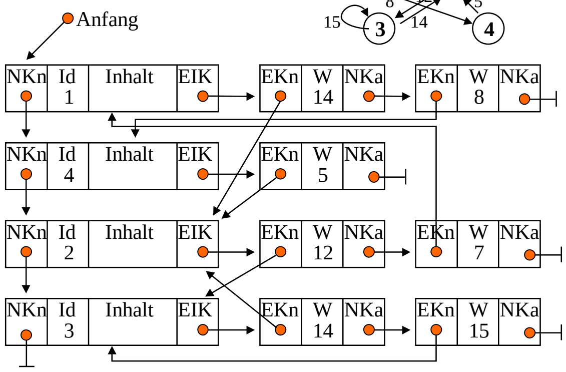
Beispiel: Gerichteter Graph mit Kantenwerten



Adjazenzliste hierzu

Zugehöriger Graph:





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;
```