Informatik

11.Klasse

Arne Terkowski

I. Python

→ Python Kurzreferenz verteilen

Unterschiede zwischen Python 2.7 und 3.X:
print(„test“) statt print „test“, analog mit input()
3/4 ergibt in 3.X 0.75 und niht mehr 0

→ Python: Blatt Objektorientierung verteilen und bearbeiten lassen.

Dauer: ca. 2 Doppelstunden

II. Listen

1. Warteschlangen

Einführender Auftrag:
Wie funktioniert das Dateisystem FAT? → führt zu einfach verkettete Liste hin

Eigenschaften von Warteschlangen:

- Begrenzte Länge
- FIFO-Prinzip (First In First Out)
- Implementierung durch Array (Feld)

Modellierung:

Python: „warteschlange.py“ ; Rumpf: „warteschlange_rumpf.py“

Nachteile:

- Wenig flexibel wegen fester Länge (evtl. Verschwendung von Speicherplatz, evtl. zu kurz)
- Schlechte Performance beim Entfernen eines Elements

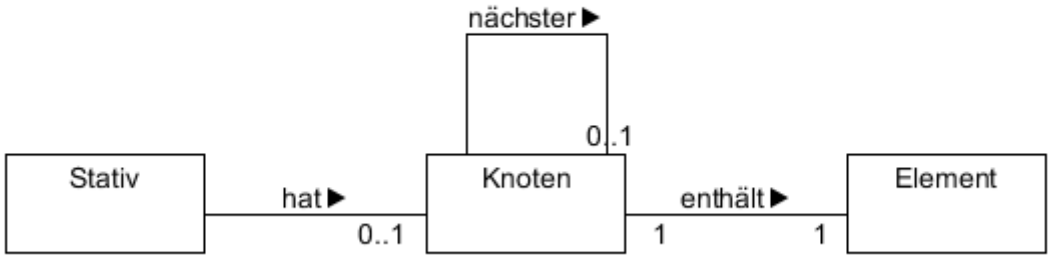
2. Einfach verkettete Listen

Beispiel (Polonaise):

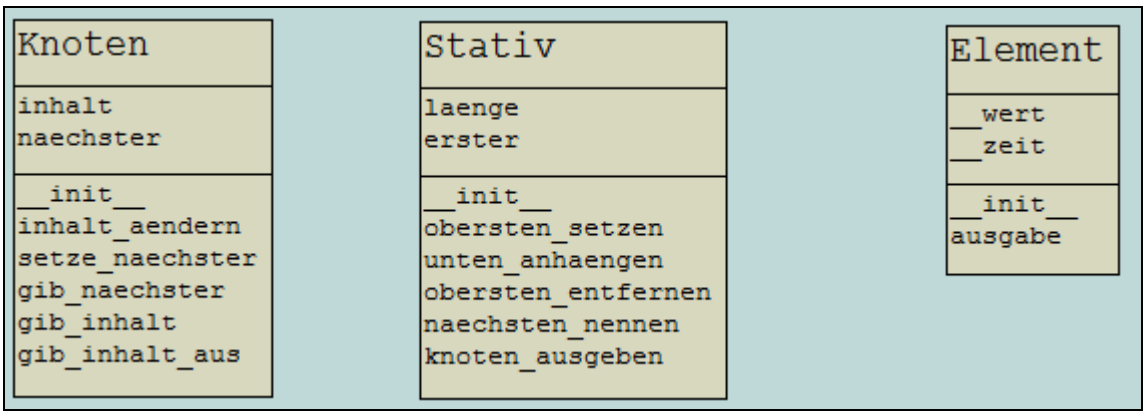
Wie erreicht der der letzte der Reihe, dass die Polonaise langsamer geht?

```
methode langsamer():
  wenn selbst.vordermann:
    selbst.vordermann.langsamer()
  ansonsten:
    selbst.geschwindigkeit--
```

Klassendiagramm einer Korbkette Obst):

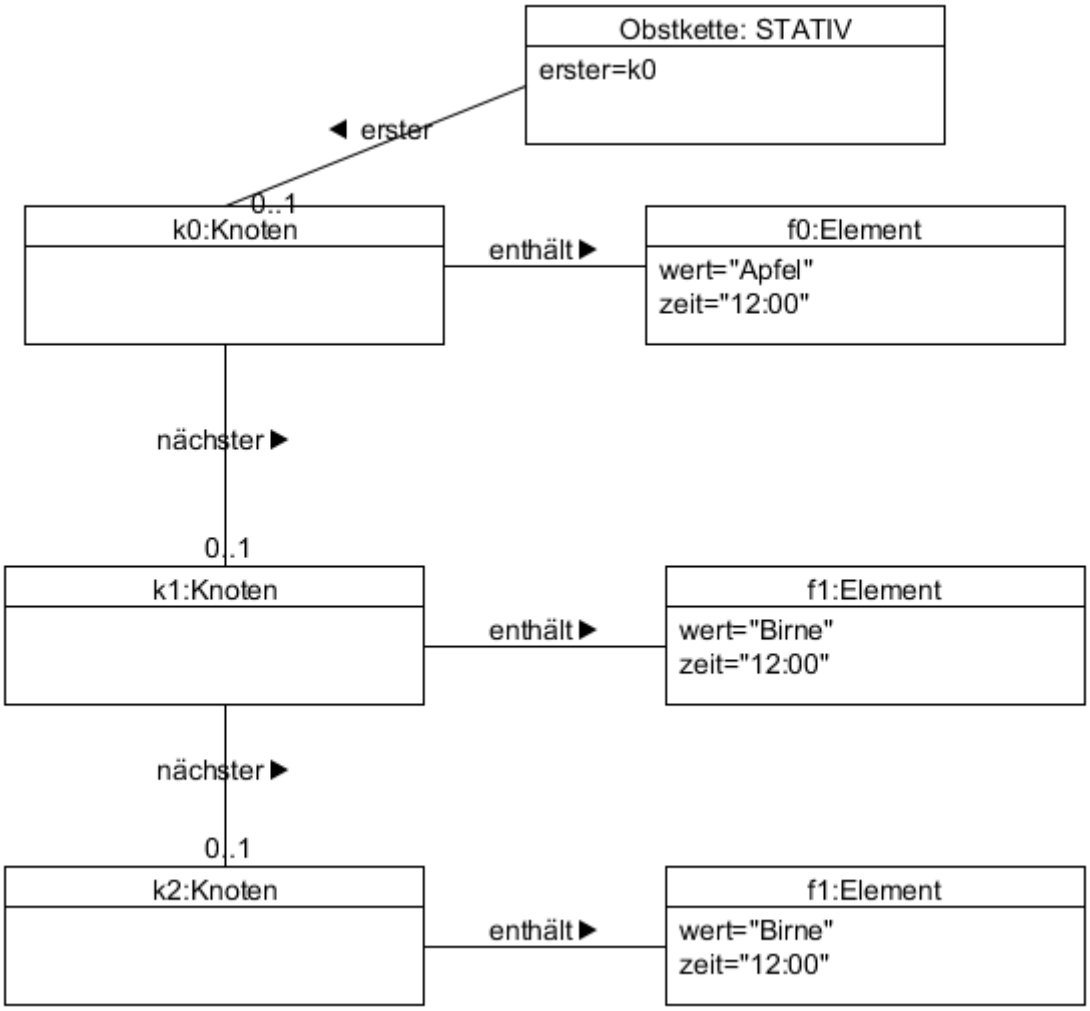


(erstellt mit UMLet)



(erstellt mit pyNsource 1.61)

Objektdiagramm:



Implementierung in Python: "verkettete_List_Obst.py"

Einfach verkettete Listen bilden eine ***rekursive*** Klassenstruktur, da jedes Objekt mit einem anderen Objekt derselben Klasse in Beziehung stehen kann.

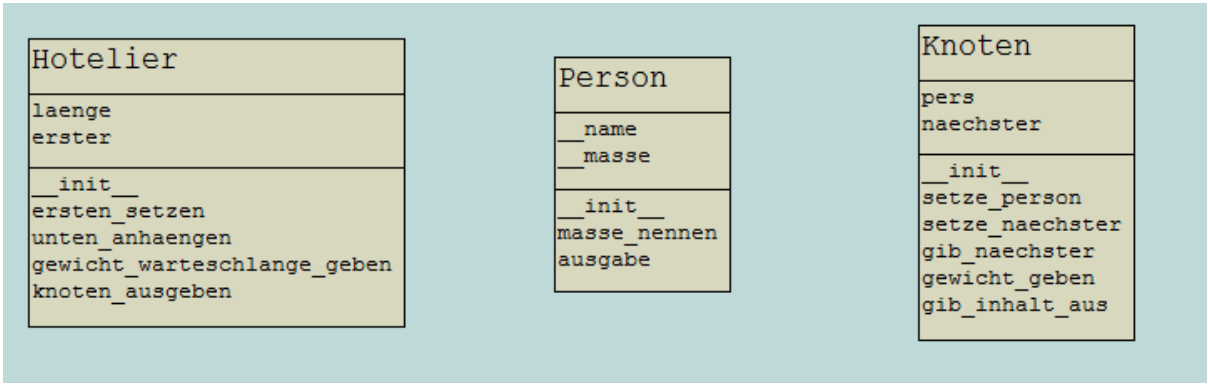
Begriff:
Unter Iteration versteht man die wiederholte Anwendung desselben Rechenverfahrens.

Anwendung:
Auf einen bestimmten Knoten der verketteten Liste greift man zu, indem man die Liste – einen Knoten nach dem anderen – durchläuft.

Aufgabe S.21/6 (Aufzug)

Python: " verkettete_List_Aufzug.py"

Grundmodell:



3. Rekursive Funktionen

Einführungsbeispiel: Berechnung der Fakultät einer Zahl

- iterativ:

- rekursiv

  Berechnungsvorschrift:

  fakult(n) = fakult(n–1)*n wenn n > 1
  fakult(1) = 1 (wenn n = 1)

  Beispiel:

  fakult(4) = fakult(3)*4 = fakult(2)*3 * 4 = fakult(1)*2 *3*4 = 1*2*3*4 = 24

  Python:

```
def fakult(n):
    if n > 1:
        return n*fakult(n-1)
    else:
        return 1
```

Fibonacci-Zahlen 1,1,2,3,5,8,13,21,34, ...

Berechnungsvorschrift für die n-te Fibonaccizahl:

fib(n) = fib(n–1) + fib(n–2) wenn n > 2
fib(2) = 1 (wenn n = 2)
fib(1) = 1 (wenn n = 1)

Python:

```
def fib(n):
    if n > 2:
        return fib(n-1)+fib(n-2)
    elif n==2:
        return 1
    else:
        return 1
```

ggt (größter gemeinsamer Teiler)

Berechnungsvorschrift:

ggT(a,b) = ggt(a–b,b) wenn a>b
ggT(a,b) = ggt(a,b–a) wenn b>a
ggT(a,b) = a wenn a=b

Python:

```
def ggt(a,b):
    if a>b:
        return ggt(a-b,b)
    elif a<b:
        return ggt(a,b-a)
    else:
        return a
```

4. Rekursive Methoden einer Liste

Begriff (rekursive Methode): Innerhalb der Methodendefinition wird dieselbe Methode eines referenzierten Objekts aufgerufen.

Beispiel: Methode `unten_anhaengen` (bzw, `hinten_anfuegen`) der Klasse Stativ.

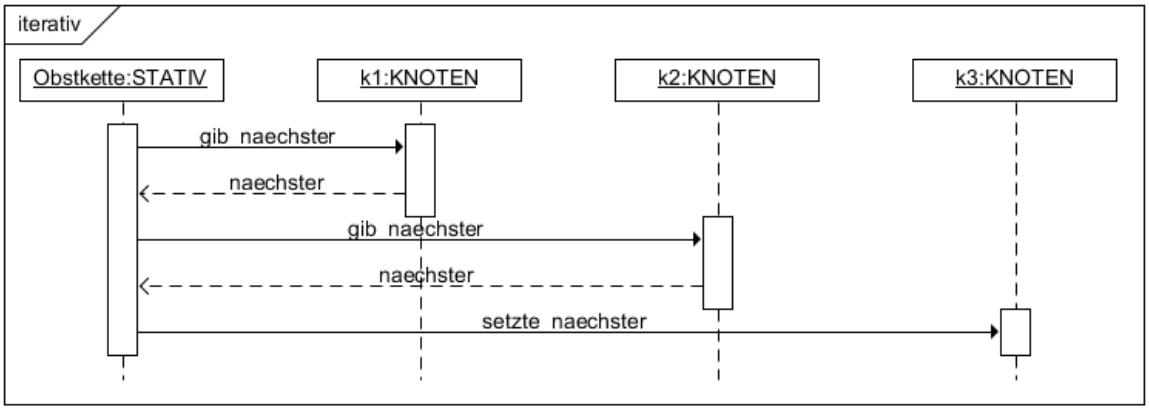
Iterativ:

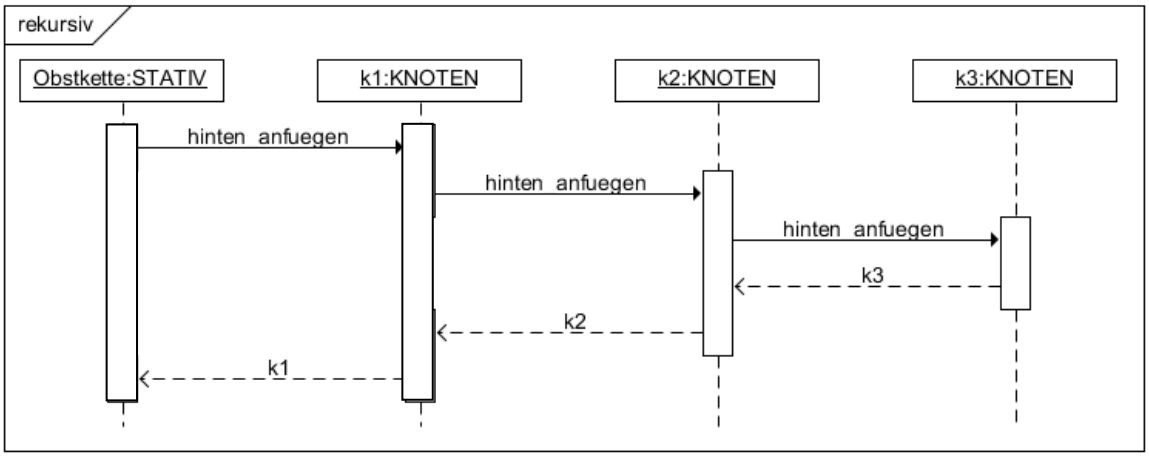
```
def unten_anhaengen(self, neu):
        k = self.erster
        for i in range (1,self.laenge):
            k=k.gib_naechster()

        k.setze_naechster(neu)
        neu.setze_naechster(None)
        self.laenge=self.laenge+1
```

Rekursiv:

```
def hinten_anfuegen(self,neu):
        """ fügt den Knoten an den letzten Knoten an"""
        if self.naechster!=None:
            self.naechster.hinten_anfuegen(neu)
        else:
            self.naechster=neu
```

Typische *Sequenzdiagramme* für beide Fälle:

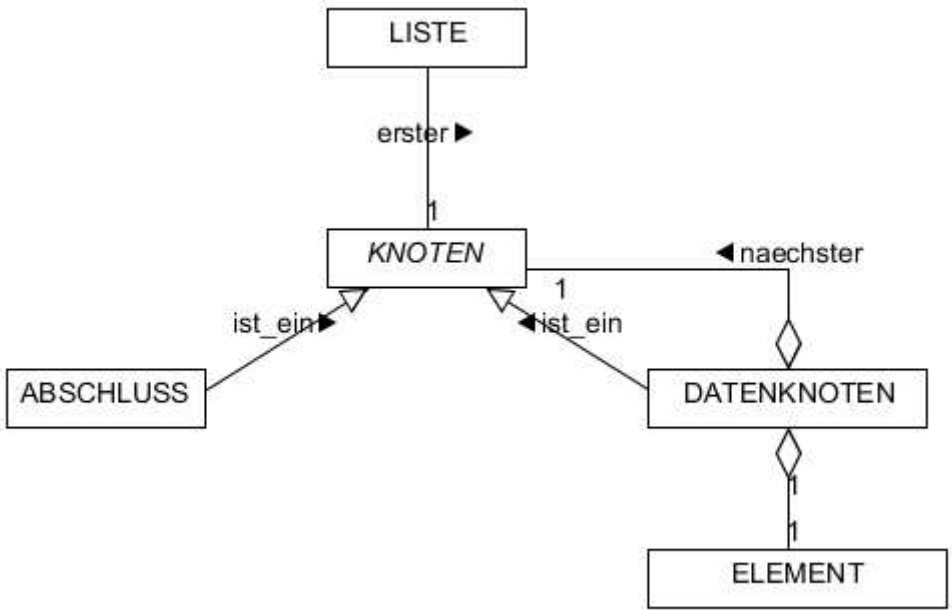




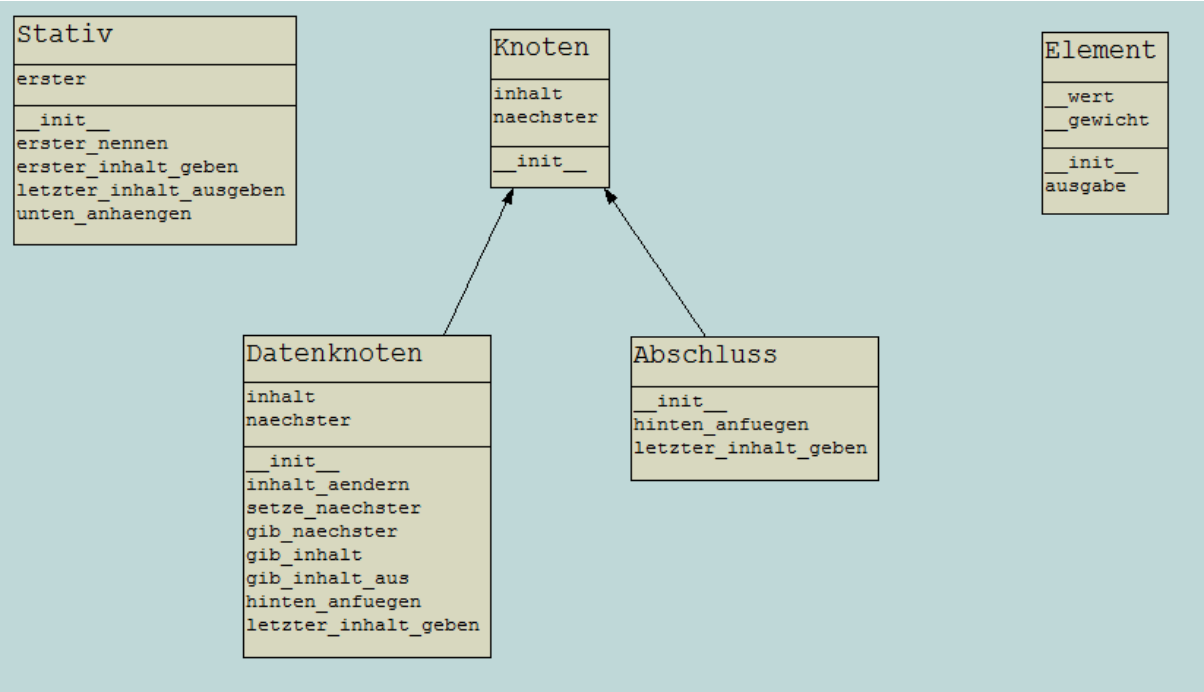
Erweiterung:
Jeder Knoten muss prüfen, ob er der Letzte ist. Oftmals löst man dies auch dadurch, einen „Abschlussknoten“ einzuführen.

Abschlussknoten und Datenknoten sind beides Objekte vom Typ Knoten.

Modell:



genauer:



Aufgaben:

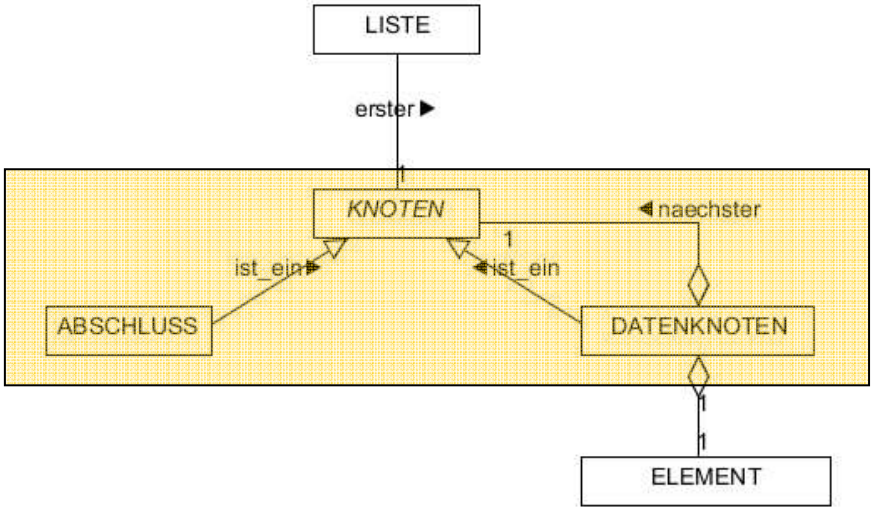
Programmiere die Methoden

- oben_einfuegen(Inhalt)
- oben_entnehmen()
- unten_entnehmen()

Bei den Entnahmemethoden soll jeweils der entfernte Inhalt ausgegeben werden.

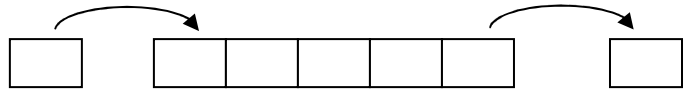
Begriff:

Der farblich hinterlegte Teil des Klassendiagramms kommt von der Struktur her in der Softwareentwicklung häufig vor. Das Muster trägt den Namen *Kompositum*.



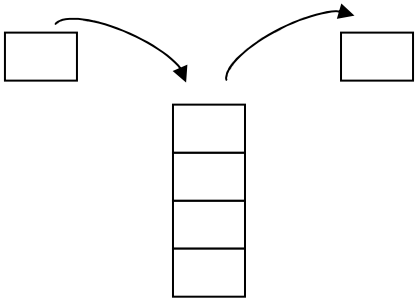
5. Dynamisch lineare Datenstrukturen

- Warteschlange (FIFO: First In First Out)



→ Taxischlange, moderne Autofähre, Kasse, Restaurantküche

- Stapel (LIFO: Last In First Out)
  andere Namen: Stack, Keller



→ Kofferraum, Aktenstapel, Türme von Hanoi

Beides kann durch die allgemeine Datenstruktur „Liste" realisiert werden.

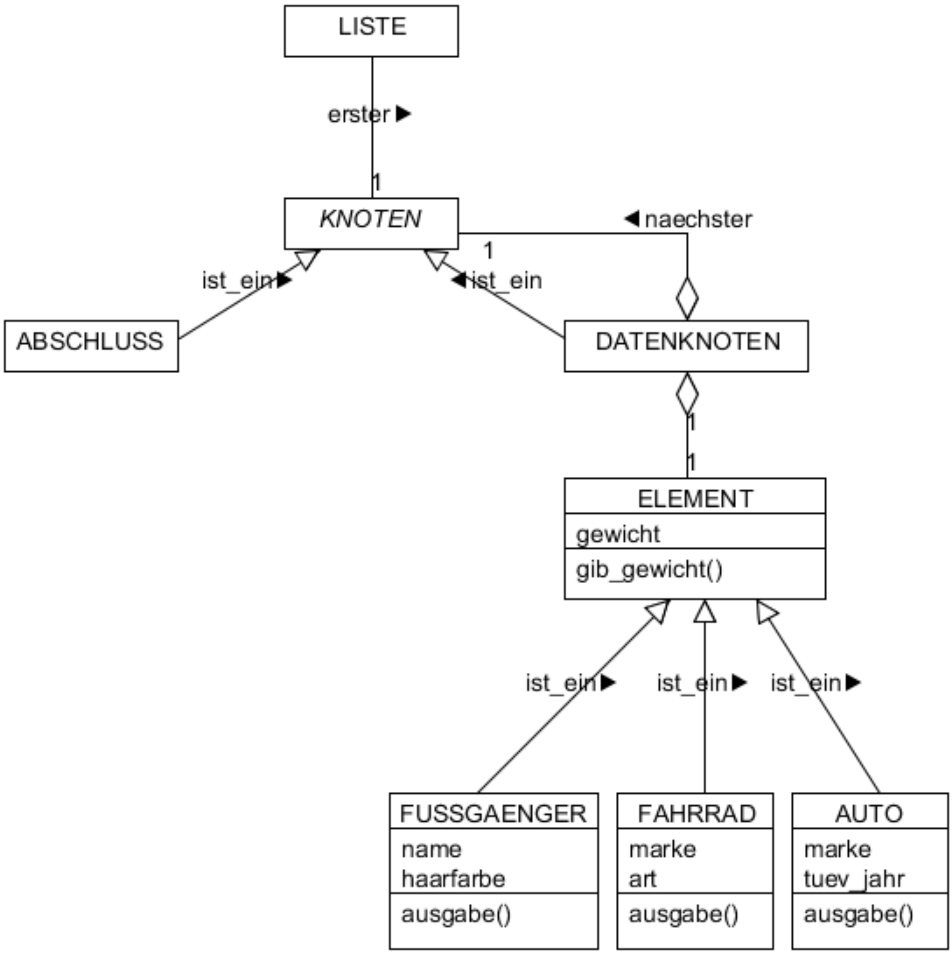
Im Fall einer Warteschlange wird die Methode „ankommen" das neue Element an das Ende der Liste fügen, bei einem Stapel an den Anfang.

Beispiel: „Die Türmen von Hanoi" (Buch S. 37)

6. Heterogene Listen

Ziel:
In der Liste sollen verschiedenartige Elemente abgelegt werden können.

Lösung:
ELEMENT wird als abstrakte Klasse definiert. Für die einzelnen Elementarten werden Unterklassen von ELEMENT definiert.



7. Sortierte Listen

Die Liste soll nach gewissen Kriterien sortiert vorliegen. Sinnvoll ist es unter Umständen, die Liste gleich sortiert aufzubauen.

Beispiele für Sortierkriterien:

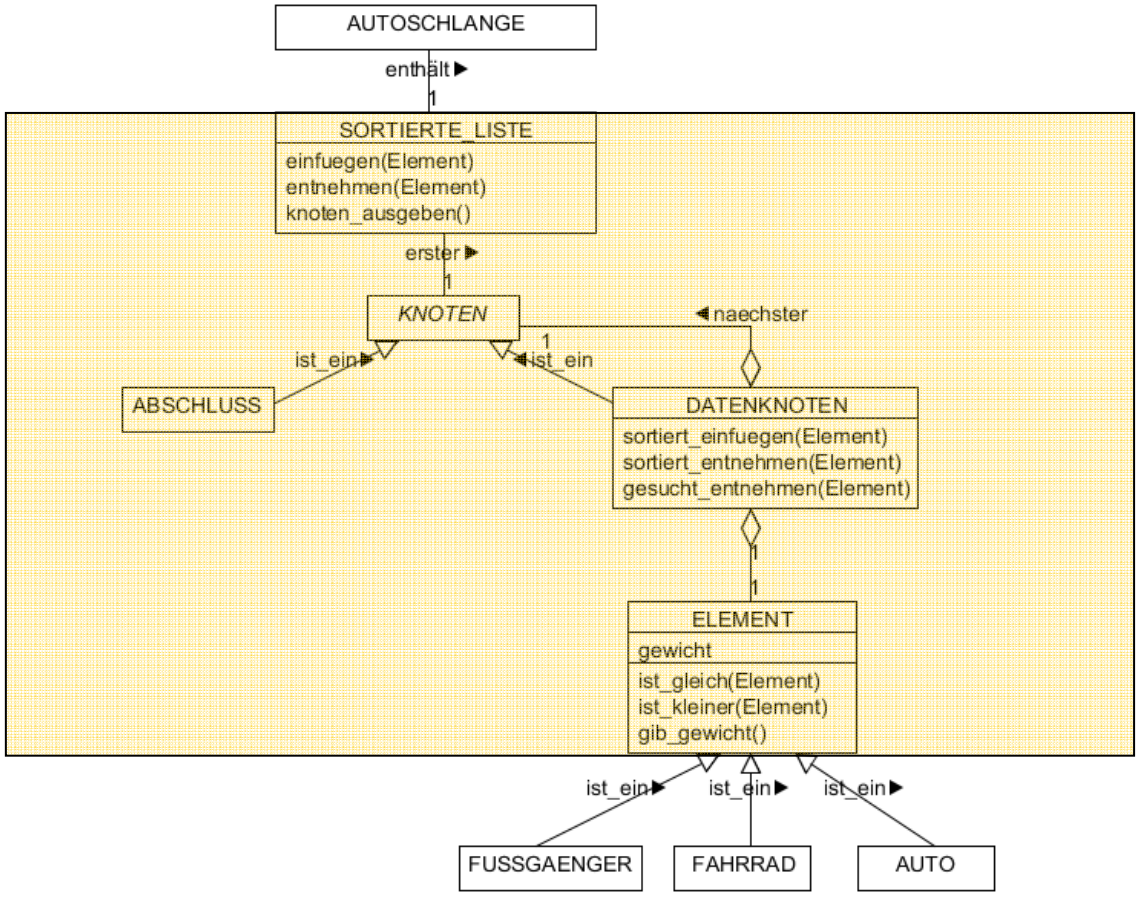
- Datum
- Zahl
- Alphanumerische Sortierung

benötigte Methoden:

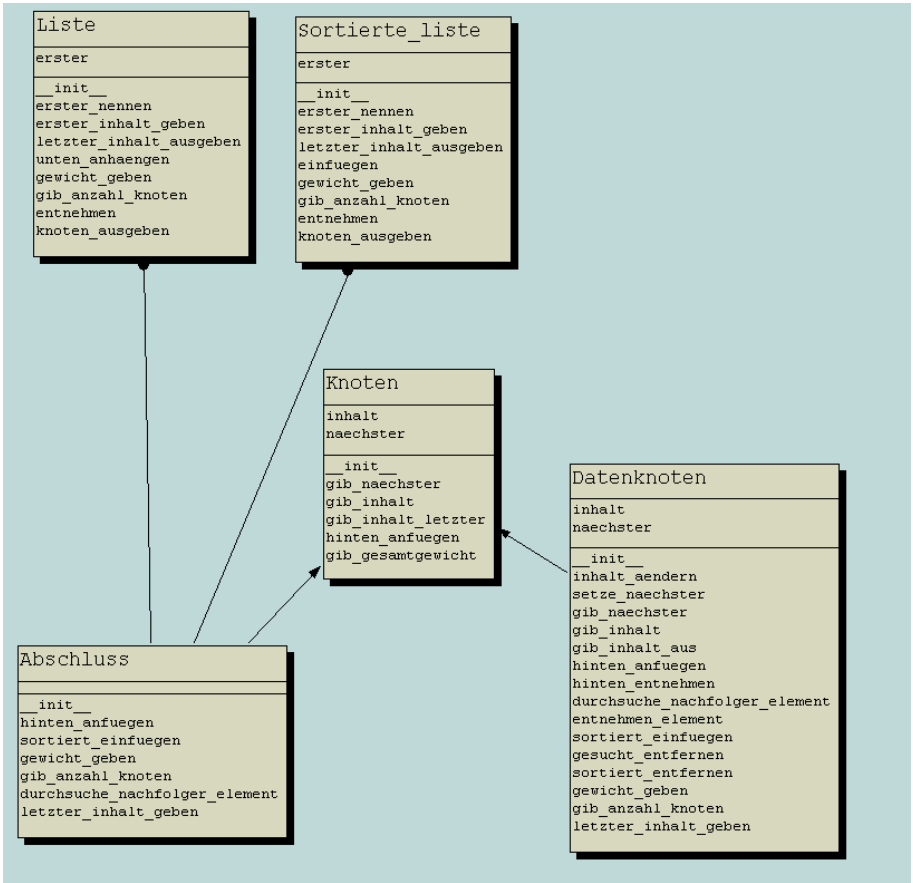
- sortiert_einfuegen(Element)

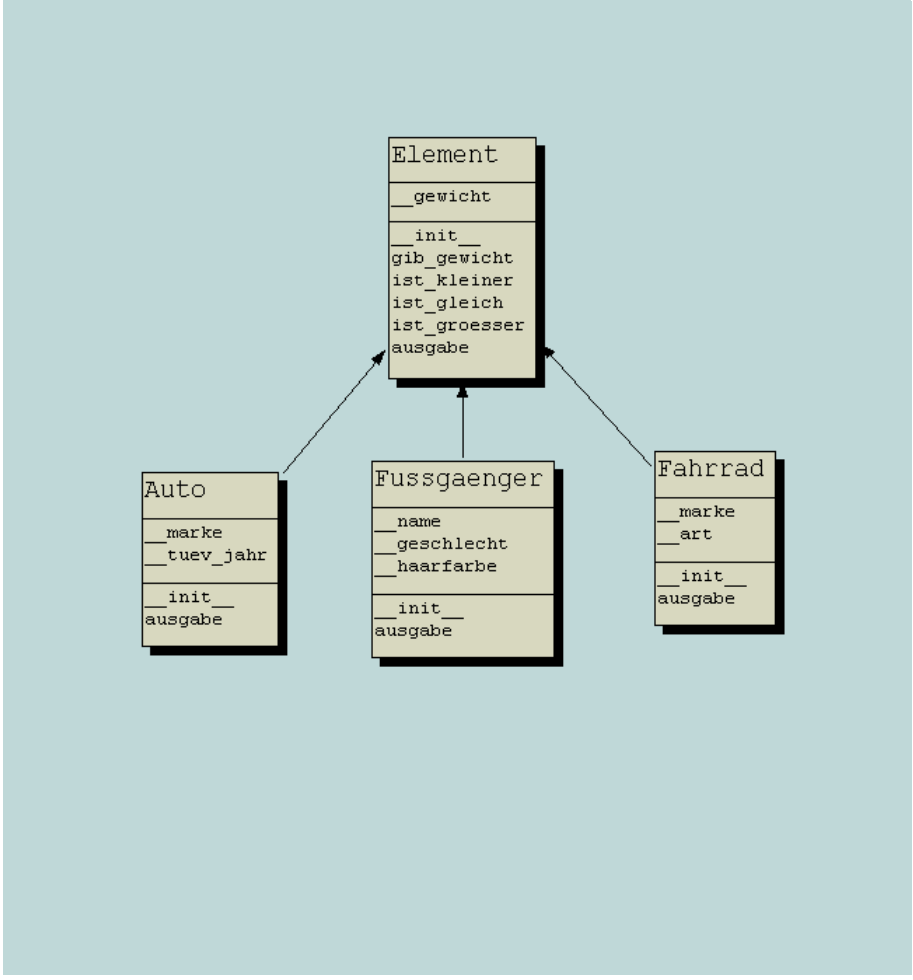
- sortiert_entfernen(Element)
  → hier wird das erste Element gelöscht, dessen beim Sortieren betrachteter Attributwert mit dem des übergebenen Elements übereinstimmt.
  → das Entfernen des ersten Elements hinter der Aufhängung der Liste muss gesondert betrachtet werden.

- gesucht_entfernen(Element)
  → es wird das Element gelöscht, das mit dem zu suchenden übereinstimmt.
  → das Entfernen des ersten Elements hinter der Aufhängung der Liste muss gesondert betrachtet werden.



Modellierung mit PyNSource:

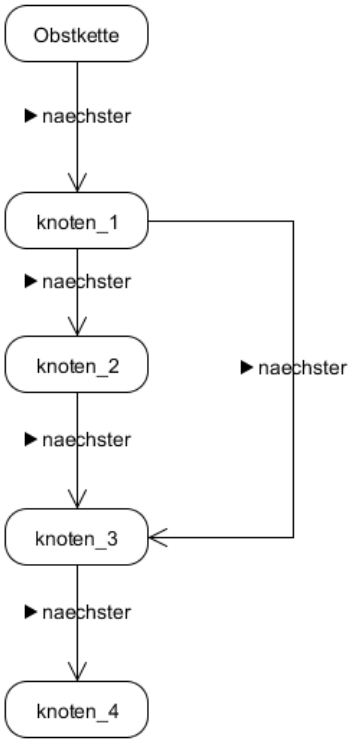


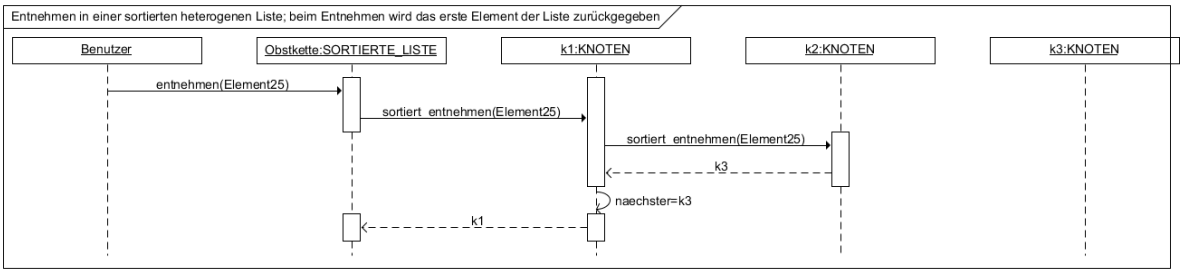


Sequenzdiagramm:

Annahme:

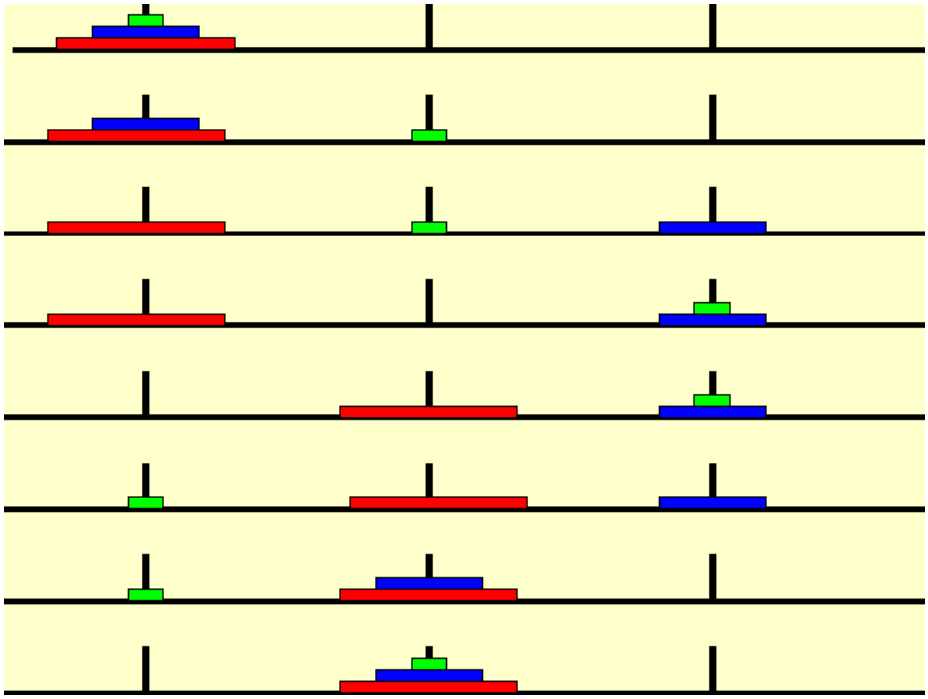
→ Element25 steckt in Knoten k2





8. Übungen

Die Türme von Hanoi



- Die Objekte vom Typ STAPEL heißen stapel_1, stapel_2 und stapel_3
- Es gibt drei Elemente der Klasse Scheibe (s0,s1,s2), daneben ein gleichartiges Hilfsobjekt namens move.
- Die Klasse STAPEL hat die Methoden einfügen und entnehmen, die wie bei einem vernünftigen Stack/Kellerspeicher funktionieren.
- Move ist ein Hilfsobjekt der Klasse Scheibe.

III. Bäume

1. Binärbäume

Baumstruktur:

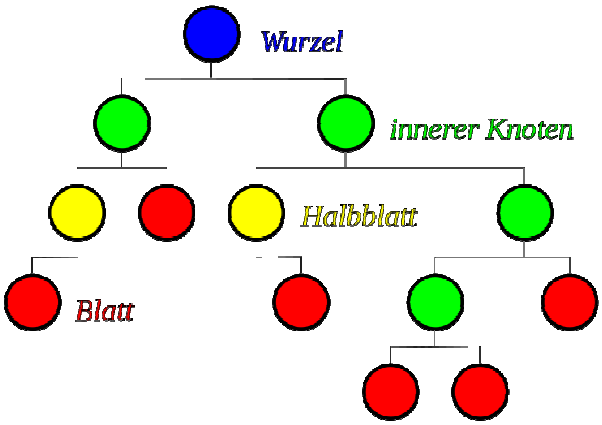
1. Es gibt genau eine Wurzel. Diese ist das einzige Objekt der Struktur, das nicht referenziert wird.

2. Von der Wurzel aus kann jedes Objekt der Struktur auf genau einem Weg durch sukzessives Verfolgen von Referenzen erreicht werden.

Binärbaum (zusätzlich):

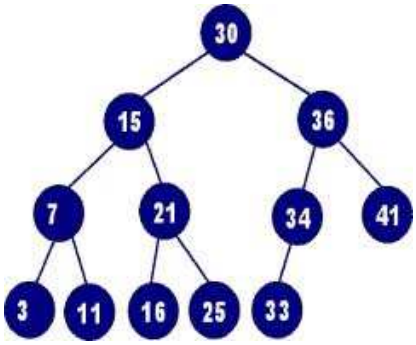
Jedes Objekt des Baums hat höchstens zwei Nachfolger.

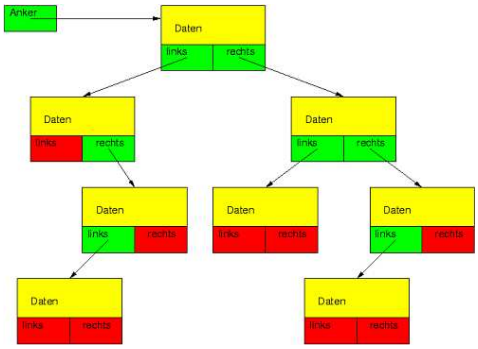
Bezeichnungen:



Jeder Knoten „hat“ genau einen *Elternknoten* und maximal zwei *Kindknoten*. (Genauer müsste man sagen *wird referenziert* bzw. *referenziert*). Blätter haben keine Kindknoten, innere Knoten derer mindestens einen.

Beispiele:





Traversieren eines Binärbaums:

Systematiches Durchsuchen der Knoten in einer bestimmten Reihenfolge.

Abkürzungen:

W: Wert des aktuellen Knotens ausgeben
LK: Gehe zum linken Kindknoten (=Wurzel des linken Teilbaumes), wenn dieser vorhanden ist
RK: Gehe zum rechten Kindknoten (=Wurzel des rechten Teilbaumes), wenn dieser vorhanden ist

PreOrder (Tiefensuche)
W–LK–RK (erst Wurzel, dann linker Teilbaum, dann rechter Teilbaum)

→ im Beispiel: 30, 15, 7, 3, 11, 21, 16, 25, 36, 34, 33, 41

→ Pseudocode:

```
preorder(k):
      bearbeite(k)
      wenn k.links ≠ null
                  preorder(k.links)
      wenn k.rechts ≠ null
                  preorder(k.rechts)
```

InOrder
LK–W–RK (erst linker Teilbaum, dann Wurzel, dann rechter Teilbaum)

→ im Beispiel: 3, 7, 11, 15, 16, 21, 25, 30, 33, 34, 36, 41

→ Pseudocode:

```
inorder(k):
      wenn k.links ≠ null
                  ineorder(k.links)
      bearbeite(k)
      wenn k.rechts ≠ null
                  ineorder(k.rechts)
```

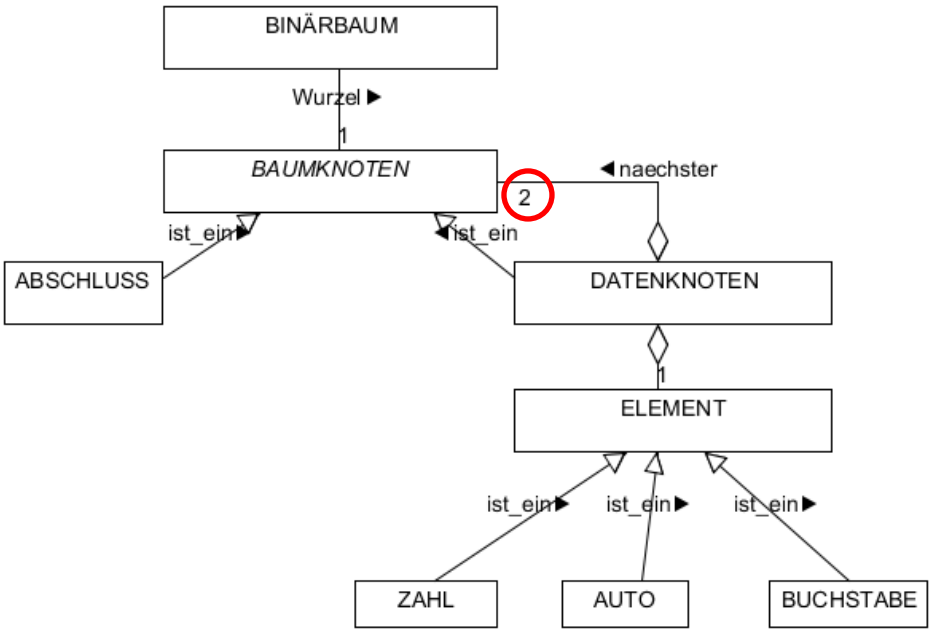
PostOrder (Breitensuche)
LK–RK–W

→ im Beispiel: 3, 11, 7, 16, 25, 21, 15, 33, 34, 41, 36, 30

→ Pseudocode:

```
inorder(k):
      wenn k.links ≠ null
                  postorder(k.links)
      wenn k.rechts ≠ null
                  postorder(k.rechts)
      bearbeite(k)
```

Implementieren von Binärbäumen



Bemerkung:
Die umringelte 2 deutet an, dass jeder Knoten (maximal) zwei Nachfolger hat. Man orientiert sich offensichtlich an dem Entwurfsmuster *Kompositum*.

Objektdiagramm:
→ vgl. Buch S.57 unten

InOrder Implementierung und Sequenzdiagramm
→ vgl. Buch S.58

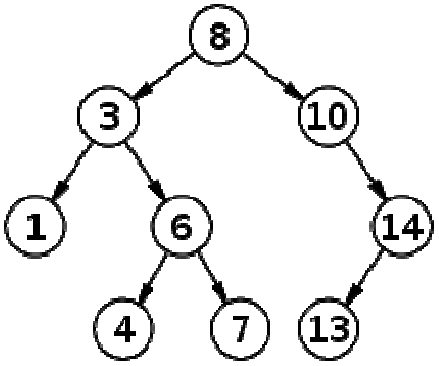
2. Geordnete Binärbäume (Suchbäume)

Ausgangspunkt:
Es existiert eine gewisse Menge von Objekten (Zahlen). Es soll untersucht werden, ob ein vorgegebenes Objekt bereits in der Menge enthalten ist.

Lösung 1:
Man vergleicht das vorgegebene Objekt mit jedem einzelnen Element der Menge.

Lösung 2:
Man überlegt sich eine geeignete Datenstruktur, bei der die Suche „effizienter" geht. Dies führt zum geordneten Binärbaum. Alternativ nennt man diesen auch Suchbaum.

Beispiel für einen geordneten Binärbaum:



Bemerkung:

Statt der Knotennamen bzw. Elementnamen sind hier der Einfachheit halber die Inhalte der Elemente aufgeschrieben. Das ist oft üblich.

Eigenschaften / Definition:

- Falls ein linker Teilbaum existiert, ist der Schlüsselwert jedes Knotens kleiner als der Schlüsselwert des aktuellen Knotens.
- Falls ein rechter Teilbaum existiert, ist der Schlüsselwert jedes Knotens größer als der Schlüsselwert des aktuellen Knotens.
- Linke und rechter Teilbaum sind auch Suchbäume.

Der Schlüsselwert kann z.B. eine Zahl, ein Datum, o.ä. sein – also allgemein ein Attribut der Baumelemente. Ebenso könnte er auch eine komplexere Struktur haben.
Je nach Art des Schlüsselwerts sehen die Methoden `ist_gleich`, `ist_kleiner`, … anders aus.

Bedeutung der InOrder Traversierung:

→ Im Beispiel: 1, 3, 4, 6, 7, 8, 10, 13, 14

Diese Traversierung liefert also eine Sortierung der Elemente der Größe nach.

Suchen in einem Suchbaum:

- vergleiche zunächst den Schlüsselwert der Wurzel mit dem Suchwert.
- wenn der Suchwert gleich dem Schlüsselwert der Wurzel ist, ist die Suche beendet.
- wenn der Suchwert kleiner als der Schlüsselwert der Wurzel ist, dann suche (rekursiv) weiter im linken Teilbaum.
- wenn der Suchwert größer als der Schlüsselwert der Wurzel ist, dann suche (rekursiv) weiter im rechten Teilbaum.

Bemerkung:
In obiger Version sind Duplikate nicht zugelassen. Man kann sich aber auch Suchbäume vorstellen, die Duplikate (d.h. Elemente mit denselben Schlüsselwerten enthalten).

3. Operationen von Suchbäumen

Einsortieren neuer Datenelemente

Neue Elemente werden immer als Blätter eingefügt!

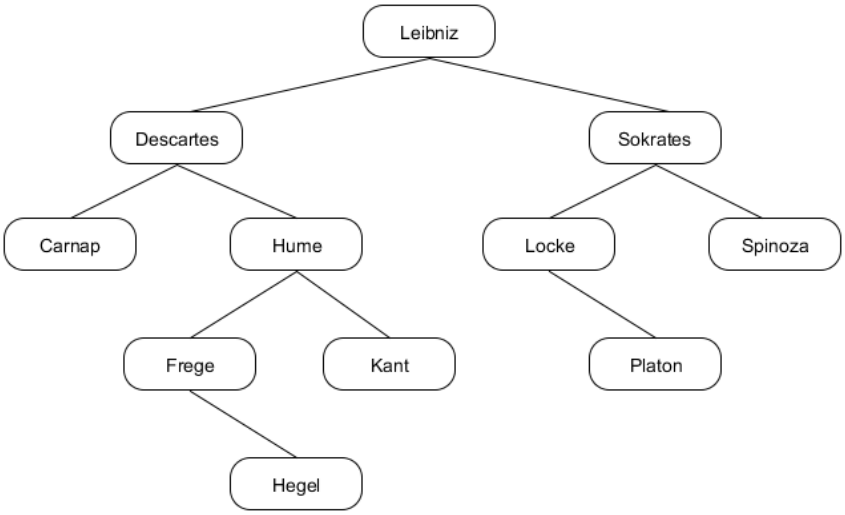
- vergleiche zunächst den Schlüsselwert des einzufügenden Elements (SE) mit dem Schlüsselwert der Wurzel (SW).
- wenn SE kleiner als SW ist, dann füge das Element im linken Teilbaum ein. Verfahre also im linken Teilbau rekursiv genauso weiter wie oben.
- wenn SE größer als SW ist, dann füge das Element im rechten Teilbaum ein. Verfahre also im rechten Teilbau rekursiv genauso weiter wie oben.
- Schlussendlich wird das neue Element als Blatt eingefügt.

Aufbau eines Baums

Man startet mit einem Baum, der nur die Wurzel enthält und fügt nacheinander die Elemente hinzu.

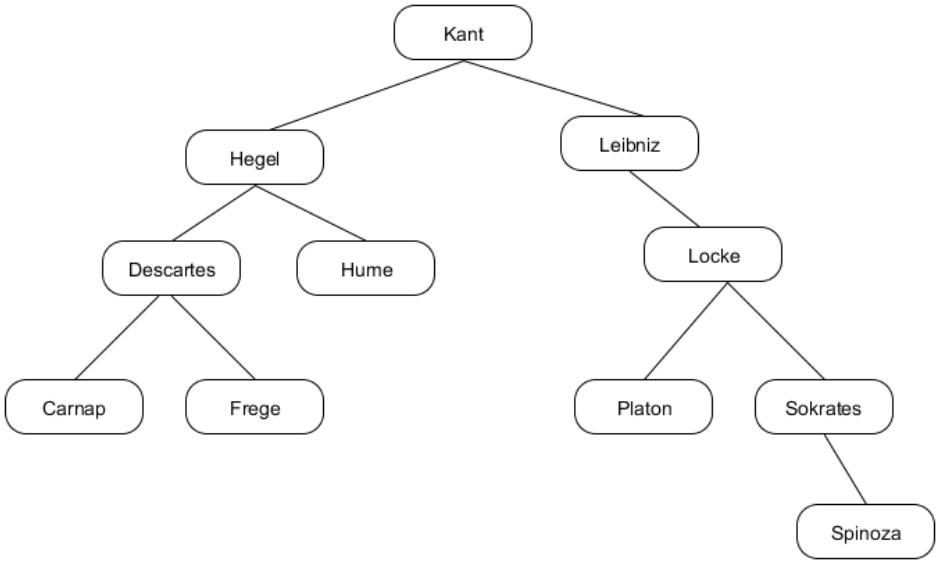
Erstelle einen Baum und füge nacheinander die Elemente folgenden Elemente hinzu:

Leibniz, Descartes, Carnap, Hume, Sokrates, Frege, Locke, Kant, Hegel, Platon, Spinoza



Eine andere Einfügereihenfolge führt zu einem anderen Ergebnis:

Kant, Leibniz, Hegel, Hume, Locke, Sokrates, Spinoza, Descartes, Carnap, Frege, Platon



Bemerkung:
Beim alphabetischen Einfügen entartet der Baum zur sortierten Liste.

Löschen von Datenelementen

Das Löschen ist am kompliziertesten:

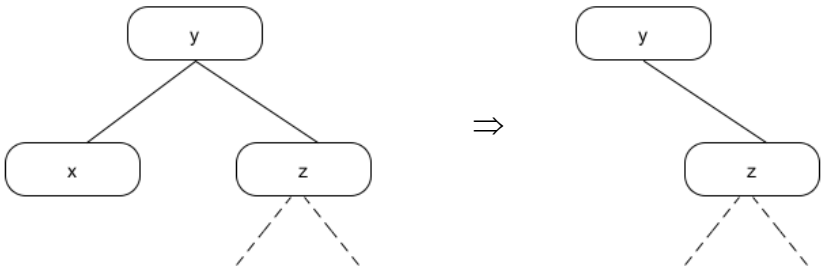
1. Fall: x ist ein Blatt

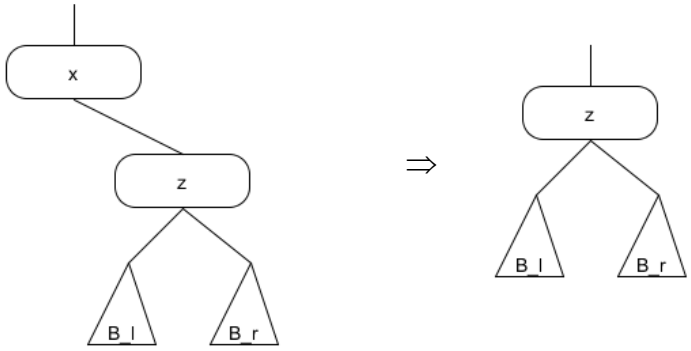
2. Fall: x hat leeren linken bzw. rechten Teilbaum

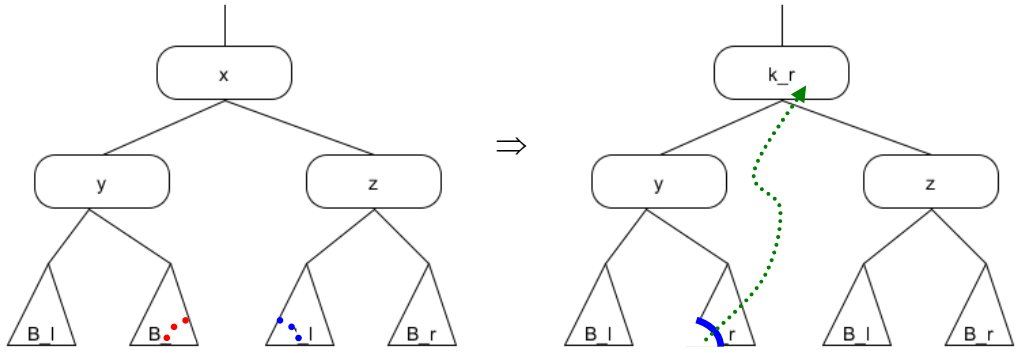
3. Fall: x hat zwei nichtleere Teilbäume.
Dann wird das Element mit dem kleinsten Schlüsselwert in rechten Teilbaum an die Stelle von x gesetzt. Danach wird im rechten Teilbaum das eben "kopierte" Element gelöscht.

Bemerkungen zu Fall 3:

- Man könnte auch das größte Element des linkes Teilbaums nehmen.
- Das Löschen funktioniert garantiert so wie in Fall 1 oder 2.





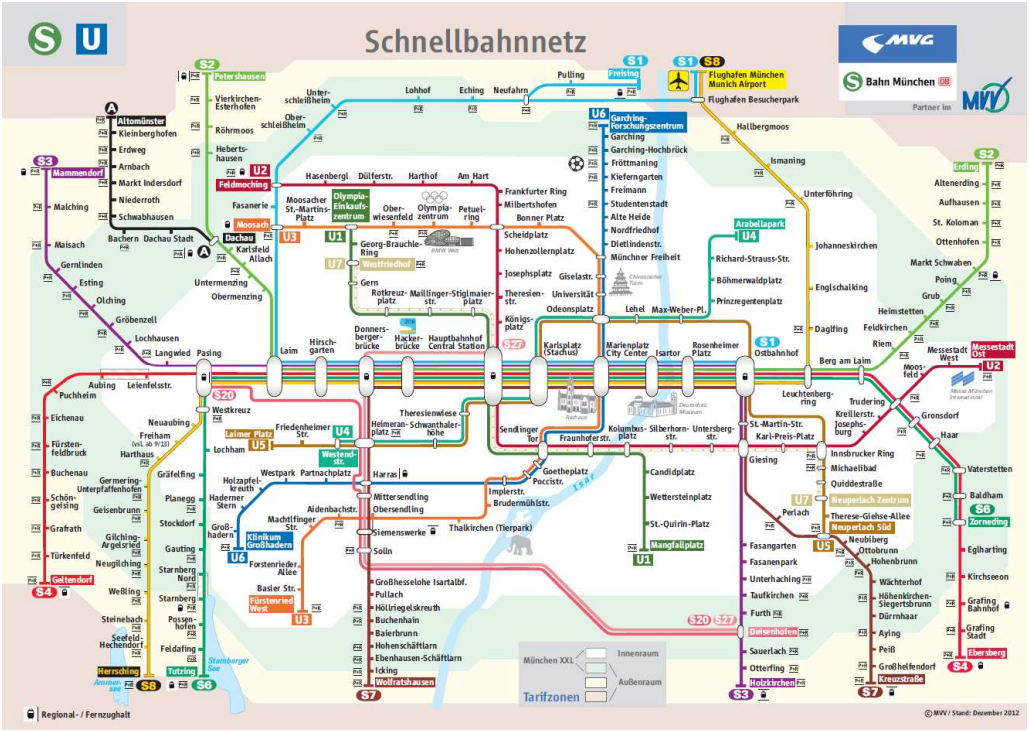


IV. Graphen

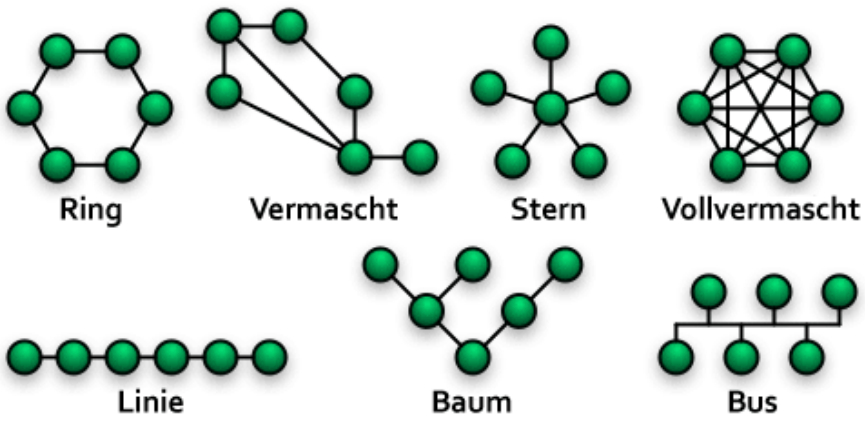
1. Grundlegende Eigenschaften

Beispiele:

1. MVV

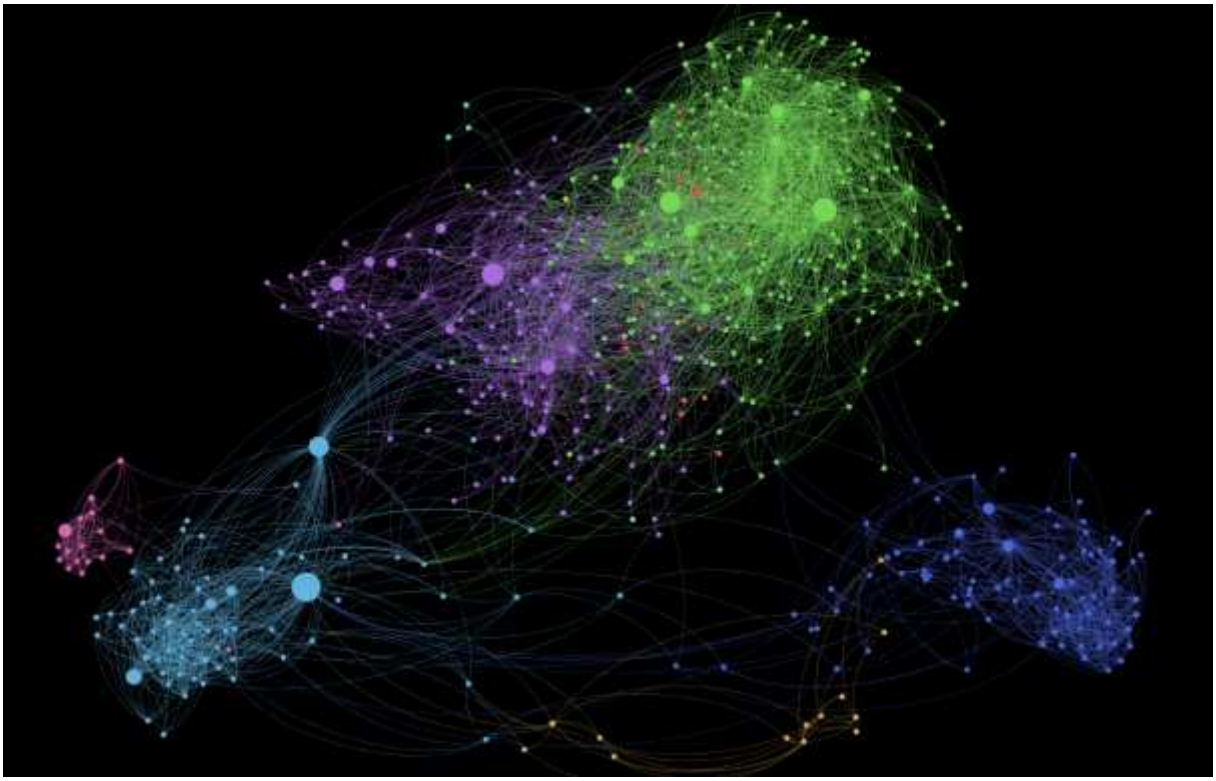


2. Netztopologie

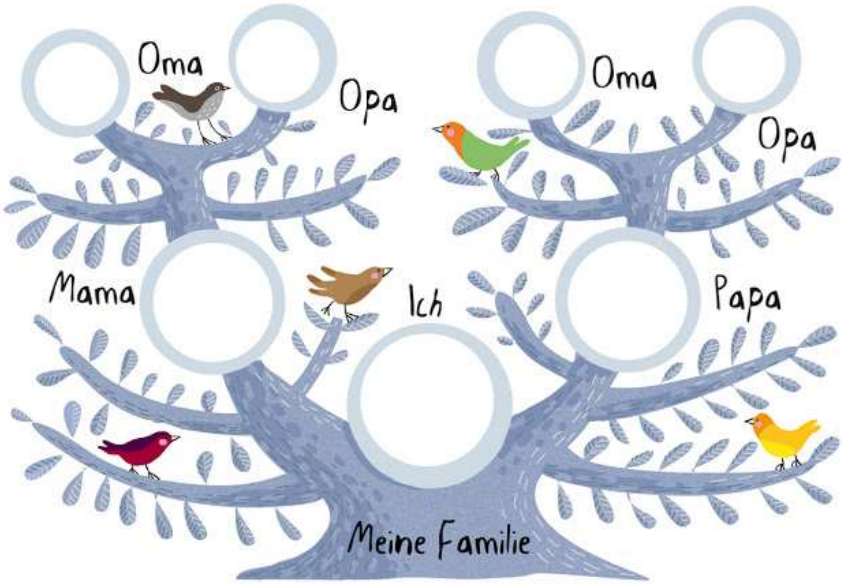


3. Facebook-Network-Visualisation (+1000 friends)

(Quelle: http://kimoquaintance.com/2011/08/22/what-can-we-learn-about-somalis-from-their-facebook-networks/)



4. Stammbaum

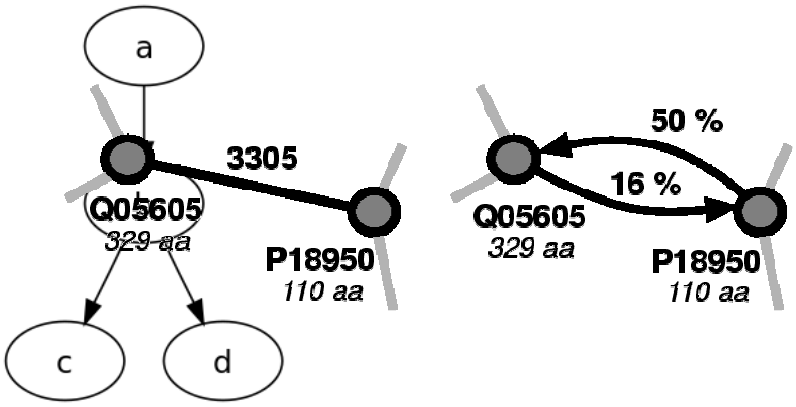


Definition:

Ein *Graph G* besteht aus einer Menge *V* von unterschiedlichen Knoten (*vertex* / *vertices*) und einer Menge *E* von Kanten (*edges*). Diese Kanten enden jeweils an Knoten.

Begriffe / Eigenschaften:

- Man unterscheidet gerichtete und ungerichtete Kanten. Bei gerichteten Kanten unterscheidet man Vorgänger– und Nachfolgerknoten. Ansonsten spricht man allgemein von Nachbarknoten.



gerichtete Kante ungerichtete Kanten

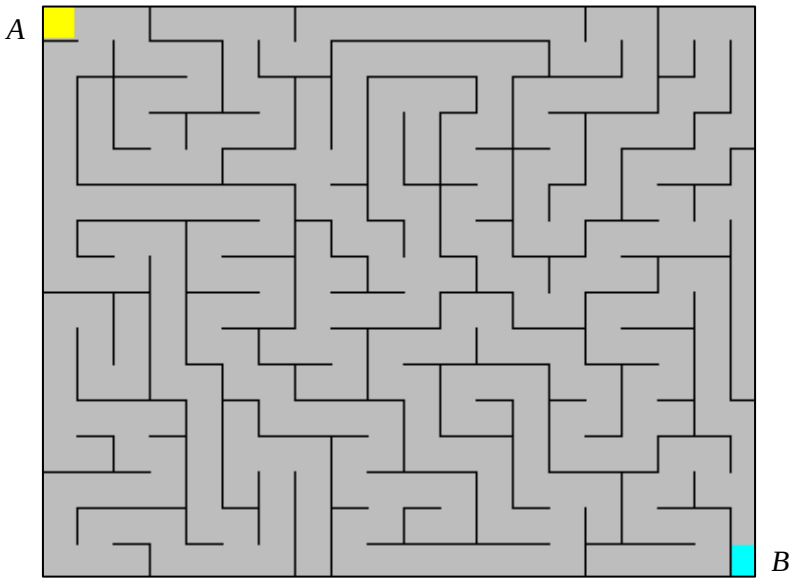
- Die Kanten (und seltener auch Knoten) können mit einem Gewicht versehen sein (vgl. oben: 50% bzw. 16%).

Aufgaben:

1. Diskutiere (kurz) die Vor– und Nachteile einiger verschiedener Netztopologien.
2. MVV-Netz
   - Sind die Kanten gerichtet oder ungerichtet?
   - Welches ist im MVV–Netz der Knoten mit den meisten Nachbarn?
   - Angenommen, es gelten folgende "Gewichte" für die Fahrzeiten:
     U-Bahn-Kante: 1
     S-Bahn-Kante (komplett im Innenraum): 2
     S-Bahn-Kante (Rest): 3
     Wie kommt man am schnellsten von Germering nach Ottobrunn, wenn keine Umstiegszeiten berücksichtigt werden und immer sofort Anschluss besteht? Was ändert sich bei sinnvolleren Annahmen?
3. Wann sind bei Autokarten gerichtete Graphen sinnvoll?

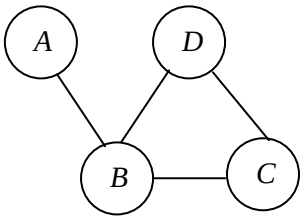
2. Wege durch Graphen

Typische Fragestellungen:

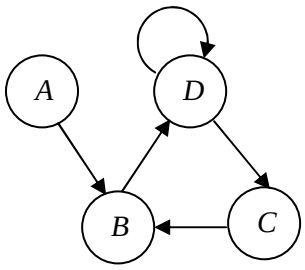


- *Gibt* es einen Weg von Knoten *A* zu Knoten *B* ?
- Welches ist der *kürzeste* Weg von *A* nach *B* ?
- Gibt es Zyklen?

Begriffe:



(I)



(II)

Sei z.B. die Menge der Knoten gleich $V = \{A, B, C, D\}$ und die Menge der Kanten gleich $E = \{\{A, B\}, \{B, C\}\{C, D\}\{D, B\}\}$

- *Wege / Pfade*:
  Unter einem Weg (Pfad) der Länge *n* versteht man eine Folge von *n* aufeinanderfolgenden Kanten.

  Im *ungerichteten* Graphen schreibt man z.B. $\{D, C\}, \{C, B\}, \{B, A\}$

  Im Falle von *gerichteten* Graphen schreibt man statt dessen $(D, C), (C, B), (B, D)$

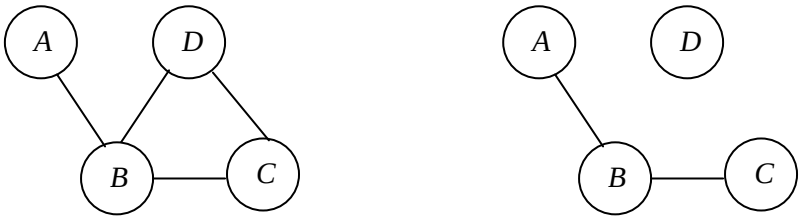
- *Zyklen:*
  Gelangt man auf einem Weg zum Ausgangspunkt zurück, spricht man von einem Zyklus.
  Vergleiche dazu das obige Beispiel zum gerichteten Graphen.
  Besitzt ein Graph mindestens einen Zyklus, spricht man von einem zyklischen Graphen.

- *Schlingen:*
  Eine Kante heißt Schlinge, wenn am selben Knoten endet, wie sie beginnt: vgl. Bsp. (II)

- *Erreichbarkeit:*
  Ein Knoten K heißt von einem Ausgangsknoten S aus erreichbar, wenn es einen Pfad von S nach K gibt. Im obigen Beispiel (II) ist A von C aus nicht erreichbar.

- *Zusammenhang von Graphen:*

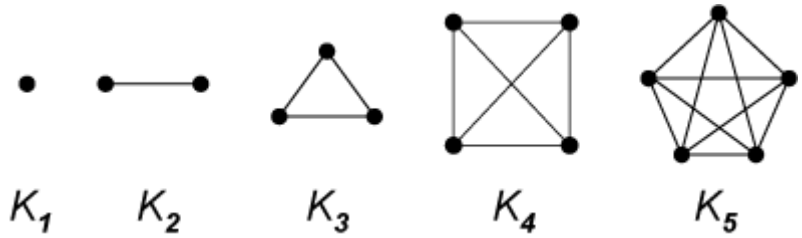
  ungerichtete Graphen:
  Ist in einem *ungerichteten* Graphen jeder Knoten von jedem anderen Knoten aus erreichbar, so spricht man von einem zusammenhängenden Graphen. Ansonsten heißt der Graph unzusammenhängend bzw. nicht zusammenhängend.



  gerichtete Graphen:
  Ist in einem *gerichteten* Graphen jeder Knoten von jedem anderen Knoten aus erreichbar, so spricht man von einem *stark* zusammenhängenden Graphen.
  Ist dies nicht der Fall, nennt man einen Graphen schwach zusammenhängend, wenn zumindest bei Ignorieren der Kantenrichtungen – also beim "zugehörigen" ungerichteten Graphen – ein zusammenhängender Graph entstehen würde.
  Hilft auch das nicht, ist der Graph unzusammenhängend.

- *Planarität*:
  Ein Graph heißt planar, wenn es (in der Ebene) keine sich kreuzenden Kanten gibt.

- *Vollständigkeit:*
  Ein Graph heißt vollständig, wenn jeder Knoten mit jedem anderen verbunden ist:

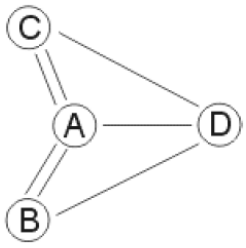


Link zum Königsberger Brückenproblem:
http://www.matheprisma.uni-wuppertal.de/Module/Koenigsb/

3. Repräsentation von Graphen

Adjazenzmatrix – ungerichteter Graph

Betrachte den ungerichteten Graphen zum Königsberger Brückenproblem. Hierbei handelt es sich um einen sogenannten Multigraphen.



Variante 1:

schreibe "x", wenn die betreffenden Knoten durch eine Kante verbunden sind.

	A	**B**	**C**	**D**
A		x	x	x
B	x			x
C	x			x
D	x	x	x	

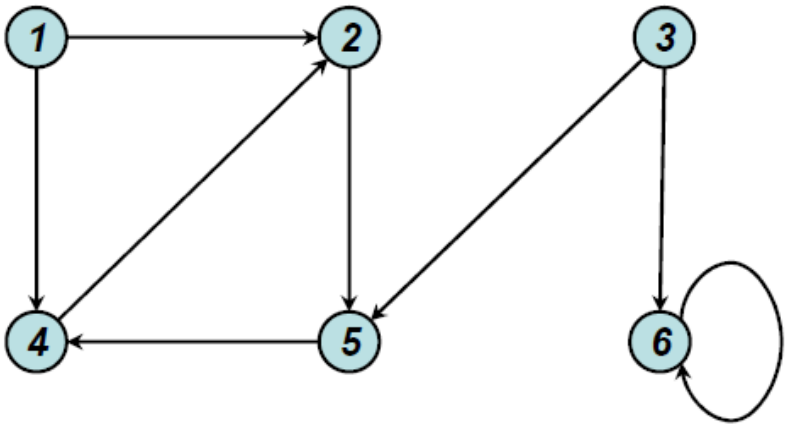
Variante 2:

schreibe "w", wenn die betreffenden Knoten durch eine Kante verbunden sind, sonst "f". Alternativ: 0 und 1

	A	**B**	**C**	**D**
A	f	w	w	w
B	w	f	f	w
C	w	f	f	w
D	w	w	w	f

Eigenschaft:
Die Adjazenzmatrix eines ungerichteten Graphen ist symmetrisch.

Adjazenzmatrix – gerichteter Graph



nach / von	**1**	**2**	**3**	**4**	**5**	**6**
1	1	0	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Bewertete (gewichtete) Graphen

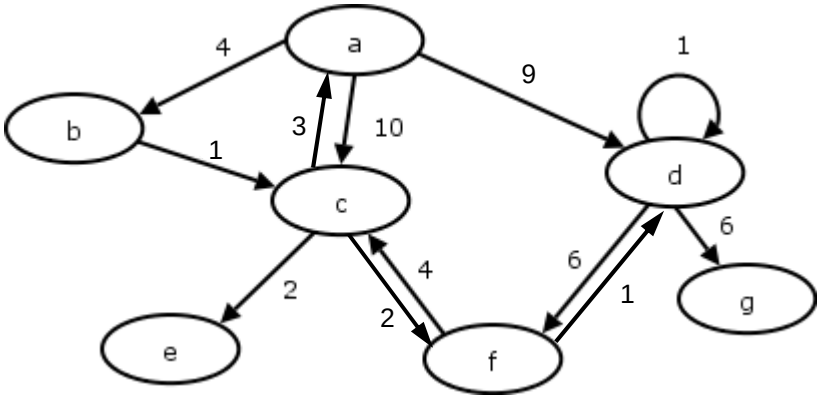
Jeder Kante wird ein bestimmter Wert zugeordnet. Dieser ergibt sich aus dem Sachzusammenhang.

Beispiele:

- Entfernungen beim Verkehrsnetz
- Kosten beim Bau von Stromtrassen

Ziel:
Ermittlung optimierter Wege (hinsichtlich Distanz, Kosten, ...)

Beispiel (*) eines bewerteten Graphen:



nach / von	**a**	**b**	**c**	**d**	**e**	**f**	**g**
a		4	10	9			
b			1				
c	3				2	2	
d				1		6	6
e							
f			4	1			
g							

- Gibt es einen Weg von **d** nach **b** ?
- Welches ist der kürzeste Weg von **a** nach **g** ?

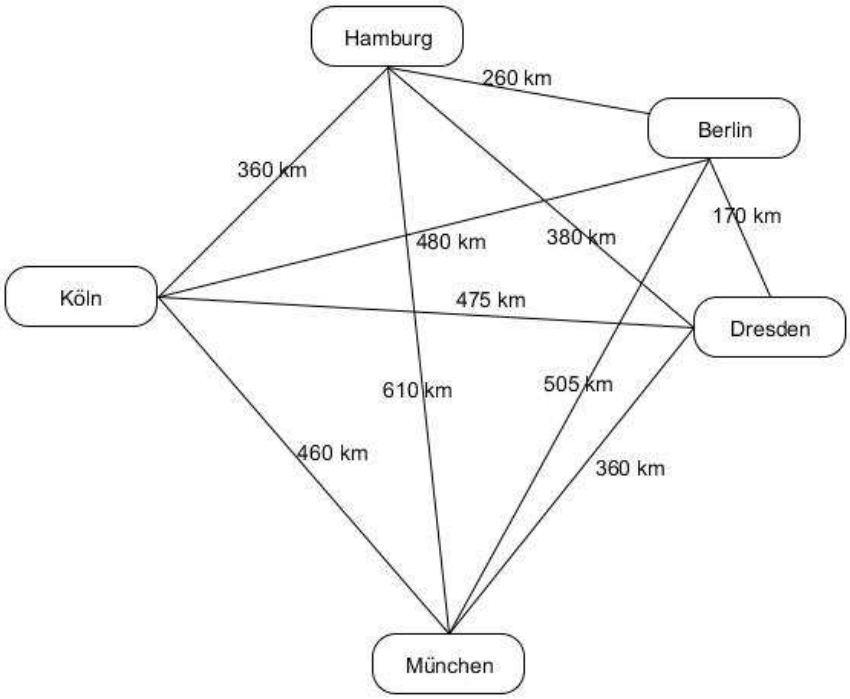
Aufgabe:
Erstelle einen bewerteten vollständigen Graphen, der als Knoten die Städte *München*, *Köln*, *Dresden*, *Berlin*, und *Hamburg* hat. Die Kantengewichte sollen die direkte Entfernung (Luftlinie) zwischen den Orten sein.

a) Schätze zunächst die folgenden Entfernungen: MK, MH, MD, MB, KB, HB

b) Ermittle alle wahren Entfernungen und skizziere den Graphen.

c) Gib die Adjazenzmatrix an.

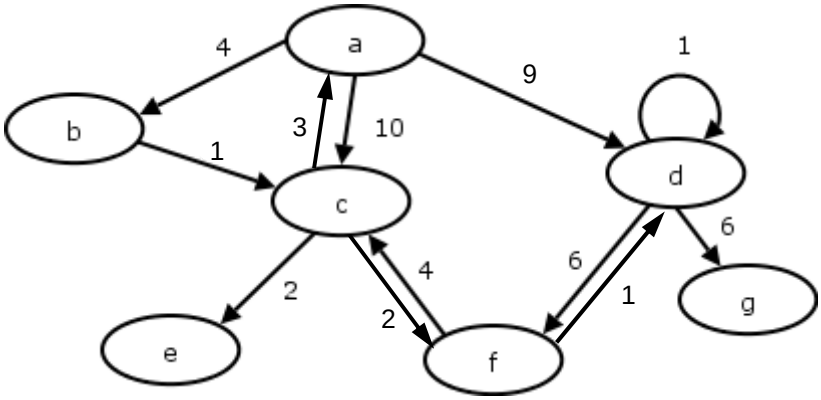
Literaturhinweis: http://www.luftlinie.org/Berlin_Dresden



nach / von	**M**	**K**	**D**	**H**	**B**
M	0	460	360	610	505
K	460	0	475	360	480
D	360	475	0	380	170
H	610	360	380	0	260
B	505	480	170	260	0

Adjazenzlisten

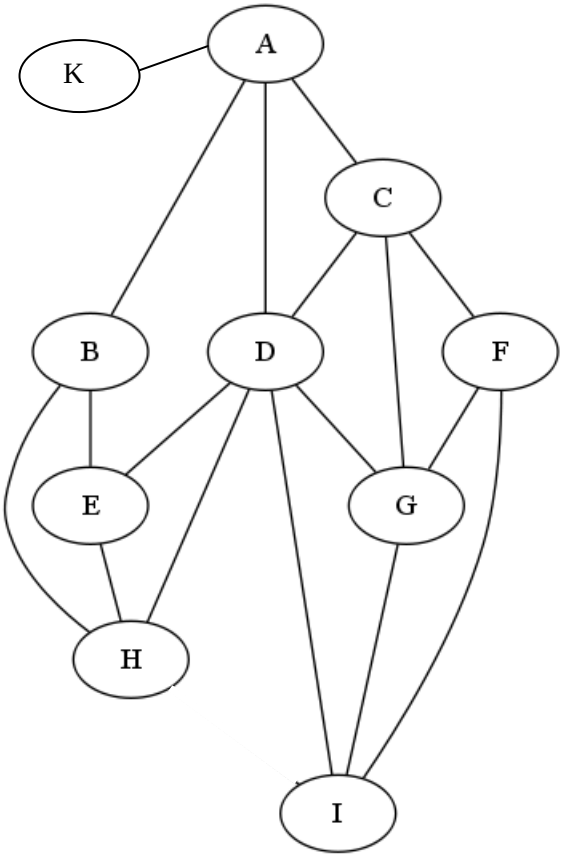
- Lies im Buch dazu auf S.102
- Erstelle eine Adjazenzliste des Graphen aus obigem Beispiel (*).



Lösung:

$a \rightarrow b\ c\ d$
$b \rightarrow c$
$c \rightarrow a\ e\ f$
$d \rightarrow d\ f\ g$
$e \rightarrow$
$f \rightarrow c\ d$
$g \rightarrow$

4. Graphentraversierung – Suchen in Graphen



Adjazenzliste

A	→	B D C K
B	→	A E H
C	→	A D F G
D	→	A C E G H I
E	→	B D H
F	→	C G I
G	→	D C F I
H	→	B D E
I	→	D G F
K	→	A

Tiefensuche (DFS: Depth First Search)

Algorithmus:

1. Wähle einen Startknoten *s* aus.
2. Markiere alle Knoten als "unbesucht".
3. Bearbeite den Startknoten *s* gemäß Punkt 4.
4. Bearbeiten eines beliebigen Knotens *k.*
    - Markiere *k* als besucht.
    - Bearbeite (rekursiv) alle Nachfolger bzw. Nachbarknoten von *k.*

Bemerkungen:

- In welcher Reihenfolge die Nachfolger bzw. Nachbarknoten bei Punkt 4 bearbeitet werden wird vom Programmierer festgelegt.
- Alle Knoten, die noch zu bearbeiten sind werden auf einem Stapel (Stack) abgelegt. Durch den rekursiven Aufruf wird dies automatisch geleistet.

zurück zum Beispiel:

- Starte bei A
- Markiere alle Knoten als unbesucht

- Markiere A als besucht

- Gehe zu B
- Markiere B ; Merke dir, dass auch E, D, C und K noch zu bearbeiten sind

- Gehe zu E
- Lege K, D, C auf den Stapel
- Markiere E; Merke dir, dass auch H (und D) noch zu besuchen sind

C
D
K

- Gehe zu H
- Es muss nichts auf den Stapel gelegt werden.
- Markiere H; D ist der einzige unbesuchte Nachbarknoten

- Gehe zu D
- Markiere D; Merke dir, dass auch G, I (und C) noch zu besuchen sind.

- Gehe zu I
- Lege G auf den Stapel
- Markiere I; Merke dir, dass auch G und F noch zu besuchen sind.

G
C
D
K

- Gehe zu G
- Lege F auf den Stapel
- Markiere G; Merke dir, dass auch C und F noch zu besuchen sind.

- Gehe zu F
- Markiere F; C ist der einzige unbesuchte Nachbarknoten

F
G
C
D
K

- Gehe zu C

- Markiere C

- Alle Nachbarknoten sind besucht.

- Nimm dir den obersten Knoten vom Stapel (F)
- Auch dieser ist bereits besucht.

- Nimm nacheinander die weiteren Knoten vom Stapel

...

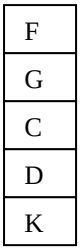
- Alle Knoten sind bereits besucht

- Nimm K vom Stapel

- Gehe zu K
- Markiere K ; Es gibt keine unbesuchten Knoten

- Es gibt keinen Knoten mehr auf dem Stapel

- FERTIG



Breitensuche (BFS: Broad First Search)

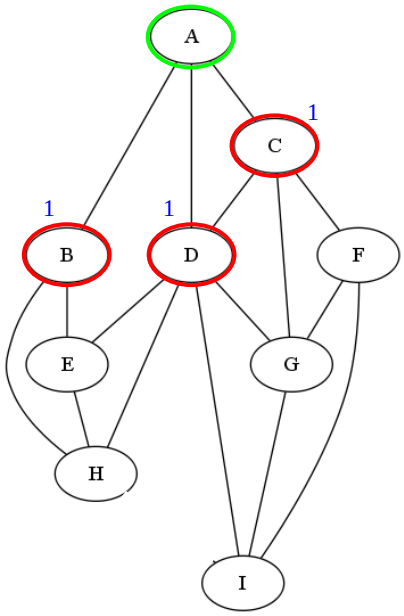
- Es können sehr leicht alle Knoten ausgegeben werden, die vom Startknoten einen Abstand kleiner gleich einem vorgegebenen *d* haben.
- Bei der Breitensuche wird beim Durchlauf die minimale Entfernung (= Anzahl der Kanten) zum Ausgangsknoten gespeichert.
- Ebenso wird für jeden Knoten der Vorgänger mitprotokolliert.

Algorithmus:

1. Wähle einen Startknoten *s* aus.
   - Zustand = "bearbeitet: grün"
   - Distanz = "0"
   - Vorgängerknoten = "null"
2. Markiere alle Knoten (außer dem Startknoten) folgendermaßen:
   - Zustand = "unbesucht: weiß"
   - Distanz = "unendlich"
   - Vorgängerknoten = "null"
3. Füge den Startknoten in eine Warteschlange ein.
4. Entnahme des vordersten Knotens *k* der Warteschlange. Führe für diejenigen Nachbarn *n* von *k*, die "unbesucht" sind, folgende Schritte aus:
   - Zustand von *n* = "in Arbeit: rot"
   - Distanz von *n* = Distanz von *k* + 1
   - Vorgängerknoten von *n* = *k*
   - Füge *n* in die Warteschlange ein.
   - Zustand von *k* = "bearbeitet: grün"
5. Wiederhole 4, solange die Warteschlange nicht leer ist.

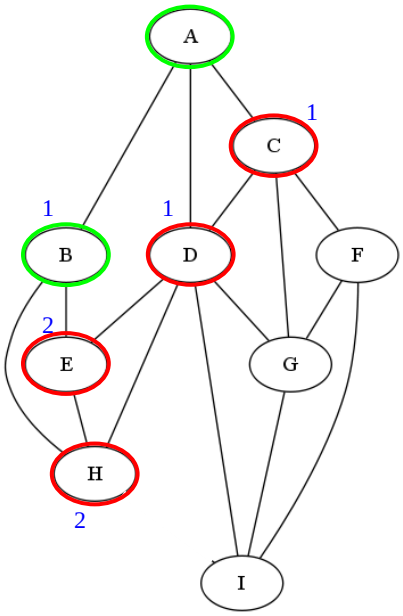
Bemerkung:
In welcher Reihenfolge die Nachfolger bzw. Nachbarknoten bei Punkt 4 bearbeitet werden wird vom Programmierer festgelegt.

Starte mit A:



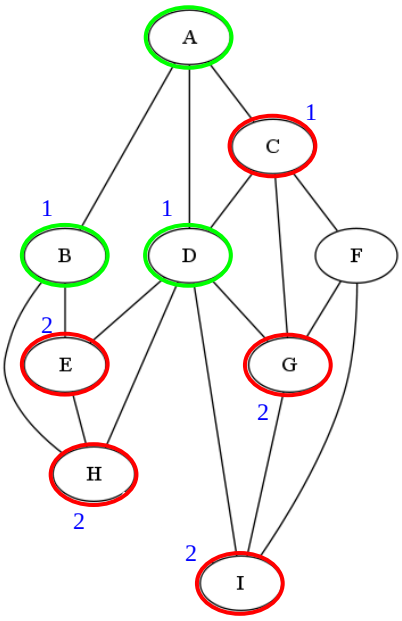
Schlange: CDB

Weiter mit B:



Schlange: EHCD

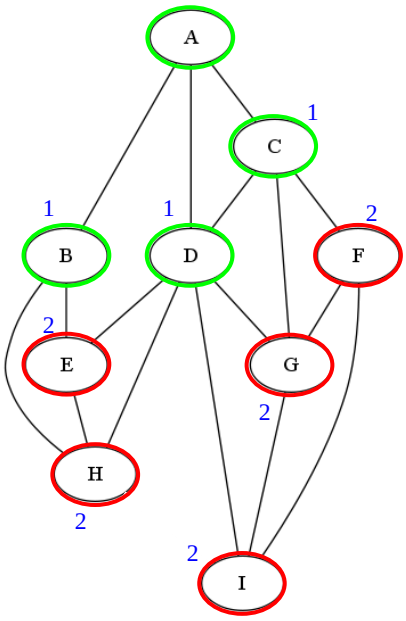
Weiter mit D:



Schlange: IGEHC

(C und H werden nicht eingefügt,
weil sie nicht unbesucht sind)

Weiter mit C:



Schlange: FIGEH

(nur F wird eingefügt, da sonst kein
Nachbar unbesucht ist)

Nun wird die Schlange weiter abgearbeitet. Dabei stellt man fest, dass kein Knoten einen unbesuchten Nachbarn hat. Alle roten Knoten werden grün. Somit sieht die Schlange nach und nach so aus:

FIGEH → FIGE → FIG → FI → F → *leer*

5. Kürzeste Wege – Der Algorithmus von Dijkstra

Einführung:
Video von Herrn Rau "Dijkstra_Algorithmus.mpeg" oder ".avi"

oder:

http://www.inf-schule.de/index.php?version=0&seite=informatik/graphen/wegeingraphen/station_dijkstra

Algorithmus:

1. Wähle einen Startknoten *s* aus.
   - Zustand = "besucht: grün"
   - Distanz = "0"
   - Vorgängerknoten = *s*
2. Initialisierung, zweiter Teil
   - Zustand = "unbesucht: weiß"
   - Distanz = "unendlich"
   - Vorgängerknoten = "null"
3. Füge den Startknoten in eine leere Warteschlange ein.
4. Entnahme des Knotens *k* aus der Warteschlange, dessen Distanzwert am *kleinsten* ist.
   Führe für diejenigen Nachbarn *n* von *k*, die "unbesucht" sind, folgende Schritte aus:
   - Wenn Kantengewicht von *k* nach *n* + Distanz von *k* < Distanz von *n*:
     Distanz von *n* = Distanz von *k* + Kantengewicht
     Vorgängerknoten von *n* = *k*
   - Füge *n* in die Warteschlange ein.
   - Setze, nachdem alle Nachbarn von *k* bearbeitet wurden, den Zustand von *k* auf "besucht: grün"
5. Wiederhole 4. , solange die Warteschlange nicht leer ist.

V. Kooperative Arbeitsabläufe

1. Projektarbeit

Einführung
Vorbereitung
Durchführung
Aufwandsschätzung

→ siehe Buch LS S.130 bis S. 147

2. Synchronisierung

Einführung:

Eine einfache Datenbank zur Verwaltung einer Musiksammlung bietet folgende Möglichkeiten:

- Neueingabe von Alben
- Veränderung / Löschen der Daten bestehender Alben (Interpret, Trackliste, etc.)
- Erstellen / Ändern eines CD-Covers
- Druck eines CD-Covers

Die Datenbank liegt im Heimnetz, sodass verschiedene Rechner / Nutzer darauf zugreifen können.

Was passiert, wenn ...

- zwei Nutzer verschiedene bzw. gleiche CD-Cover drucken wollen?
- ein Nutzer zwei Instanzen des Programms öffnet und in der einen die Trackliste desselben Albums ändern und im anderen diese Trackliste drucken will?
- zwei Nutzer die Trackliste desselben Albums verändern möchten?

Ähnliche Beispiele:

vgl. hierzu auch:
http://os.ibds.kit.edu/downloads/LN_Kapitel-06_Nebenlaeufigkeit-Synchronisation-Wechselseitiger-Ausschluss.pdf

1. Leser–/Schreiberproblem

   Erstelle eine Datei "test.txt" mit einem Editor und schreibe etwas hinein. Öffne die Datei zusätzlich mit Word. Versuche die Datei mit beiden Programmen zu ändern und zu speichern. Was passiert?

   Lösungsansätze (je nachdem, was man für wichtiger hält):

   Leservorrang:
   Alle Schreiber werden blockiert, solange mindestens ein Leser aktiv ist. Der letzte Leser gibt dann das Signal an den am längsten wartenden Schreiber, der dann exklusiv schreiben darf. Dann wird geprüft, ob wieder mindestens ein Leser wartet. Diese würden dann "aufgeweckt" und dürften dann lesen. Andernfalls würde der nächste Schreiber aktiviert.

   Schreibervorrang:
   Sobald ein Schreiber seinen Schreibwunsch geäußert hat und das Dokument aktuell gerade von mindestens einem Leser gelesen wird, werden keine weiteren Leser oder Schreiber mehr zugelassen, bis der am längsten wartende Schreiber nach Beendigung des letzten aktuellen Lesevorgangs seine exklusive schreibende Tätigkeit aufnehmen kann.
   Nach seinem erfolgreichen Schreiben wird zunächst geprüft, ob weitere Schreiber darauf warten, das Dokument zu modifizieren. Wenn dies der Fall ist, werden die Schreiber den wartenden Lesern vorgezogen. Ansonsten werden wiederum alle wartenden Leser auf einmal auf das Dokument zum parallelen Lesen zugelassen.

2. Schleibrücke bei Lindaunis:

3. Gemeinsamer Zugriff auf einen Drucker (allgemein: Ressource bzw. ein Betriebsmittel)

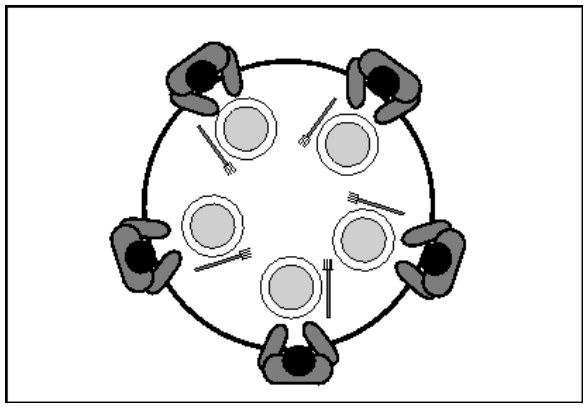
4. Produzenten–/Konsumentenproblem

   Ausgangspunkt ist ein Zwischenlager, das in der Praxis nur eine begrenzte Anzahl von Produkten zwischenlagern kann. In unregelmäßigen Abständen kommen Lieferwagen angerollt, um ihre Produkte abzuliefern. Auf der anderen Seite kommen in ebenfalls unregelmäßigen Abständen Kunden an, die Produkte mit nach Hause nehmen wollen.

   Probleme:
   Zwei oder mehr Kunden können nicht das gleiche Produkt nehmen bzw. zwei oder mehr Produzenten können ihr Produkt nicht auf den gleichen Lagerplatz abstellen.

5. Philosophenproblem

Fünf Philosophen sitzen an einem runden Tisch. Zwischen ihnen liegt jeweils eine Gabel. Um aus einer Schüssel in der Mitte des Tisches Nudeln zu bekommen benötigt jeder Philosoph zwei Gabeln.



Alle Philosophen sind durch identische Prozesse "modelliert", die "parallel" laufen.

<u>Strategie 1:</u>
die Philosophen sind nicht "gierig" und greifen sich die jeweils rechte Gabel, wenn sie hungrig sind. Sie warten dann, ob die linke Gabel frei ist und nehmen die, wenn das der Fall ist.

Es kommt zu einer Verklemmung (deadlock), da alle Philosophen genau eine Gabel in der hand haben, aber keine weiter mehr bekommen können.

<u>Strategie 2:</u>
Es gibt genau einen Essring (token), der es einem Philosophen erlaubt, sich zwei Gabeln zu nehmen. Währenddessen darf kein weiterer Philosoph Gabeln in Händen halten. Hierbei werden aber die Resourcen (5 Gabeln) nicht optimal ausgenutzt, da immer nur zwei in Betrieb sind.

Begriff:

Nebenläufige Prozesse heißen auch *potentiell parallel*. Sie können entweder auf mehreren Prozessoren tatsächlich gleichzeitig (parallel) ausgeführt werden oder auf einem einzigen Prozessor pseudo-parallel laufen. Ihre zeitliche Reihenfolge ist dabei nicht von vornherein festgelegt.

Grundlegende Fragestellung

- Greifen die parallelen Prozesse auf gemeinsame Ressourcen (Betriebsmittel: Speicher, Dateien, Drucker, ...) zu?
- Wie erfolgt dann die Kommunikation, Koordination bzw. Synchronisation der Prozesse?

In der Regel tritt beim Parallellauf von Prozessen nur in bestimmten Situationen ein Problem auf, nämlich dann, wenn gemeinsame Ressourcen gleichzeitig genutzt werden sollen.
Man spricht hier von einem *kritischen Bereich*.

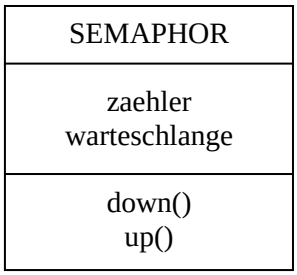
So ist es bei der Schleibrücke aus obigem Beispiel kritisch, wenn zwei Prozesse (Fahrzeuge) die Ressource (Brücke) gleichzeitig beanspruchen. Geschieht der Zugriff auf die Ressource im wechselseitigen Ausschluss, so kann das Problem vermieden werden.

Lösungsmöglichkeiten (in der Informatik)

- Softwarelösung
- Hardwarelösung
- Integration ins Betriebssystem

Semaphore

Semaphore stellen etwas wie Eintrittstickets in die kritischen Bereiche dar.



In der *Warteschlange* befinden sich die Objekte, die Einlass in den kritischen Bereich verlangen bzw. auf eine gemeinsame Ressource zugreifen wollen. Die Schlange wird über die Methoden down() und up() verwaltet.

Zaehler ist eine Integer-Variable, die zunächst mit der Anzahl der Objekte belegt ist, die sich gleichzeitig im kritischen Bereich befinden dürfen. Beim wechselseitigen Ausschluss kann Zaehler nur die Werte 0 und 1 haben.

down() – Methode:
Wann immer ein Objekt in einen kritischen Bereich eintreten will, ruft es die Methode down() auf. Wenn der Zähler 0 ist, wird das aufrufende Objekt (hinten) in die Warteschlange eingereiht. Ansonsten wird der Zähler um 1 verringert und das aufrufende Objekt kann in den kritischen Bereich eintreten.

up()–Methode:
Erhöhung des Zählers um 1. Das nächste Objekt in der Warteschlange wird aktiviert (aufgerufen

Bemerkung:
down() und up() dürfen keinesfalls unterbrochen werden – auch beim Multitasking nicht.